YosysHQ Yosys

Version 0.66

YosysHQ GmbH

Jun 01, 2026

CONTENTS

1 What is Yosys **15**
- 1.1 What you can do with Yosys . . . 15
  - 1.1.1 Typical applications for Yosys . . . 16
  - 1.1.2 Things you can't do . . . 16
- 1.2 The Yosys family . . . 16
- 1.3 The original thesis abstract . . . 17
  - 1.3.1 Benefits of open source HDL synthesis . . . 18
  - 1.3.2 History of Yosys . . . 18

2 Getting started with Yosys **21**
- 2.1 Installation . . . 21
  - 2.1.1 CAD suite(s) . . . 21
  - 2.1.2 Building from source . . . 22
  - 2.1.3 Source tree and build system . . . 24
- 2.2 Synthesis starter . . . 25
  - 2.2.1 Demo design . . . 25
  - 2.2.2 Loading the design . . . 27
  - 2.2.3 Elaboration . . . 27
  - 2.2.4 Flattening . . . 33
  - 2.2.5 The coarse-grain representation . . . 34
  - 2.2.6 Hardware mapping . . . 42
  - 2.2.7 Final steps . . . 50
- 2.3 Scripting in Yosys . . . 52
  - 2.3.1 Script parsing . . . 53
  - 2.3.2 The synthesis starter script . . . 53

3 Using Yosys (advanced) **61**
- 3.1 Synthesis in detail . . . 61
  - 3.1.1 Synth commands . . . 61
  - 3.1.2 Converting process blocks . . . 62
  - 3.1.3 FSM handling . . . 65
  - 3.1.4 Memory handling . . . 68
  - 3.1.5 Optimization passes . . . 83
  - 3.1.6 Technology mapping . . . 89
  - 3.1.7 The extract pass . . . 91
  - 3.1.8 The ABC toolbox . . . 105
  - 3.1.9 Mapping to cell libraries . . . 108
- 3.2 More scripting . . . 113
  - 3.2.1 Loading a design . . . 113
  - 3.2.2 Selections . . . 116

3.2.3 Interactive design investigation . . . 128
3.2.4 Symbolic model checking . . . 147
3.2.5 Dataflow tracking . . . 152
3.3 Minimizing failing (or bugged) designs . . . 153
3.3.1 Before you start . . . 154
3.3.2 Minimizing RTLIL designs with bugpoint . . . 154
3.3.3 Minimizing Verilog designs . . . 157
3.3.4 Identifying issues . . . 159
3.4 Notes on Verilog support in Yosys . . . 160
3.4.1 Unsupported Verilog-2005 Features . . . 160
3.4.2 Verilog Attributes and non-standard features . . . 160
3.4.3 Non-standard or SystemVerilog features for formal verification . . . 164
3.4.4 Supported features from SystemVerilog . . . 164
3.5 Scripting with Pyosys . . . 165
3.5.1 Getting Pyosys . . . 166
3.5.2 Scripting and Database Inspection . . . 166
3.5.3 Modifying the Database . . . 167
3.5.4 Encapsulating as Passes . . . 169

4 Yosys internals **171**
4.1 Internal flow . . . 171
4.1.1 Flow overview . . . 171
4.1.2 Control and data flow . . . 172
4.1.3 The Verilog and AST frontends . . . 174
4.2 Internal formats . . . 183
4.2.1 The RTL Intermediate Language (RTLIL) . . . 184
4.3 Working with the Yosys codebase . . . 190
4.3.1 Writing extensions . . . 190
4.3.2 Compiling with Verific library . . . 199
4.3.3 Writing a new backend using FunctionalIR . . . 202
4.3.4 Identifying the root cause of bugs . . . 215
4.3.5 Contributing to Yosys . . . 219
4.3.6 Testing Yosys . . . 224
4.4 Techmap by example . . . 225
4.4.1 Mapping OR3X1 . . . 226
4.4.2 Conditional techmap . . . 228
4.4.3 Scripting in map modules . . . 229
4.4.4 Handling constant inputs . . . 231
4.4.5 Handling shorted inputs . . . 232
4.4.6 Notes on using techmap . . . 234
4.5 Hashing and associative data structures in Yosys . . . 234
4.5.1 Container classes based on hashing . . . 234
4.5.2 The hash function . . . 235
4.5.3 Making a type hashable . . . 235
4.5.4 Porting plugins from the legacy interface . . . 236

5 A primer on digital circuit synthesis **237**
5.1 Levels of abstraction . . . 237
5.1.1 System level . . . 238
5.1.2 High level . . . 238
5.1.3 Behavioural level . . . 238
5.1.4 Register-Transfer Level (RTL) . . . 239
5.1.5 Logical gate level . . . 239
5.1.6 Physical gate level . . . 240

5.1.7 Switch level . . . 240
5.1.8 Yosys . . . 240
5.2 Features of synthesizable Verilog . . . 240
5.2.1 Structural Verilog . . . 240
5.2.2 Expressions in Verilog . . . 241
5.2.3 Behavioural modelling . . . 241
5.2.4 Functions and tasks . . . 242
5.2.5 Conditionals, loops and generate-statements . . . 242
5.2.6 Arrays and memories . . . 242
5.3 Challenges in digital circuit synthesis . . . 243
5.3.1 Standards compliance . . . 243
5.3.2 Optimizations . . . 243
5.3.3 Technology mapping . . . 244
5.4 Script-based synthesis flows . . . 244
5.5 Methods from compiler design . . . 245
5.5.1 Lexing and parsing . . . 245
5.5.2 Multi-pass compilation . . . 246
5.5.3 Static Single Assignment (SSA) form . . . 247

6 RTLIL text representation **249**
6.1 Lexical elements . . . 249
6.1.1 Characters . . . 249
6.1.2 Identifiers . . . 249
6.1.3 Values . . . 249
6.1.4 Strings . . . 250
6.1.5 Comments . . . 250
6.2 File . . . 250
6.2.1 Autoindex statements . . . 250
6.2.2 Modules . . . 251
6.2.3 Attribute statements . . . 251
6.2.4 Signal specifications . . . 251
6.2.5 Connections . . . 251
6.2.6 Wires . . . 251
6.2.7 Memories . . . 252
6.2.8 Cells . . . 252
6.2.9 Processes . . . 252
6.2.10 Switches . . . 253
6.2.11 Syncs . . . 253

7 Auxiliary libraries **255**
7.1 BigInt . . . 255
7.2 dlfcn-win32 . . . 255
7.3 ezSAT . . . 255
7.4 fst . . . 255
7.5 json11 . . . 255
7.6 MiniSAT . . . 256
7.7 SHA1 . . . 256
7.8 SubCircuit . . . 256

8 Auxiliary programs **257**
8.1 yosys-config . . . 257
8.2 yosys-filterlib . . . 258
8.3 yosys-abc . . . 258
8.4 yosys-smtbmc . . . 258

8.5 yosys-witness . . . 262

9 Internal cell library **263**
9.1 Word-level cells . . . 263
9.1.1 Unary operators . . . 263
9.1.2 Binary operators . . . 269
9.1.3 Multiplexers . . . 289
9.1.4 Registers . . . 293
9.1.5 Memories . . . 303
9.1.6 Finite state machines . . . 315
9.1.7 Coarse arithmetics . . . 317
9.1.8 Arbitrary logic functions . . . 325
9.1.9 Specify rules . . . 327
9.1.10 Formal verification cells . . . 331
9.1.11 Debugging cells . . . 338
9.1.12 Wire cells . . . 341
9.2 Gate-level cells . . . 343
9.2.1 Combinatorial cells (simple) . . . 343
9.2.2 Combinatorial cells (combined) . . . 347
9.2.3 Flip-flop cells . . . 353
9.2.4 Latch cells . . . 408
9.2.5 Other gate-level cells . . . 420
9.3 Cell properties . . . 421

10 Command line reference **423**
10.1 Command reference . . . 424
10.1.1 Yosys environment variables . . . 425
10.1.2 Reading input files . . . 425
10.1.3 Writing output files . . . 449
10.1.4 Yosys kernel commands . . . 456
10.1.5 Formal verification . . . 456
10.1.6 Passes . . . 474
10.1.7 Technology libraries . . . 516
10.1.8 Internal commands for developers . . . 519
10.1.9 Writing command help . . . 524

Bibliography **533**

Property Index **535**

Internal cell reference **537**

Tag Index **541**

Command Reference **545**

Yosys is an open source framework for RTL synthesis. To learn more about Yosys, see *What is Yosys*. For a quick guide on how to get started using Yosys, check out *Getting started with Yosys*. For the complete list of commands available, go to *Command reference*.

Todo

look into command ref improvements

- Search bar with live drop down suggestions for matching on title / autocompleting commands
- Scroll the left sidebar to the current location on page load
- Also the formatting in pdf uses link formatting instead of code formatting

Todo

how does a filterlib rules-file work?

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/appendix/auxprogs.rst, line 22.)

Todo

see if we can get the two hanging appnotes as lit references

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/bib.rst, line 10.)

Todo

flip-flops with async load, `$_ALDFFE?_[NP]{2,3}_`

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/cell/gate_reg_ff.rst, line 234.)

Todo

Add information about *$alu*, *$fa*, *$macc_v2*, and *$lcu* cells.

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/cell/word_arith.rst, line 4.)

Todo

Describe formal cells

$check, *$assert*, *$assume*, *$live*, *$fair*, *$cover*, *$equiv*, *$initstate*, *$anyconst*, *$anyseq*, *$anyinit*, *$allconst*, and *$allseq*.

Also *$ff* and *$_FF_* cells.

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/cell/word_formal.rst, line 13.)

Todo

Describe *$fsm* cell

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/cell/word_fsm.rst, line 4.)

Todo

$specify2, *$specify3*, and *$specrule* cells.

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/cell/word_spec.rst, line 4.)

Todo

Add information about *$slice* and *$concat* cells.

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/cell/word_wire.rst, line 4.)

Todo

Can we warn on command groups that aren't included anywhere?

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/cmd_ref.rst, line 13.)

Todo

reconsider including the whole (~77 line) design like this

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/getting_started/example_synth.rst, line 23.)

Todo

fifo.v description

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/getting_started/example_synth.rst, line 31.)

Todo

consider a brief glossary for terms like adff

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/getting_started/example_synth.rst, line 199.)

Todo

`hierarchy` failure modes

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/getting_started/example_synth.rst, line 237.)

Todo

pending bugfix in `wreduce` and/or *`opt_clean`*

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/getting_started/example_synth.rst, line 427.)

Todo

`ice40_dsp` is pmgen

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/getting_started/example_synth.rst, line 501.)

Todo

look into command ref improvements

- Search bar with live drop down suggestions for matching on title / autocompleting commands
- Scroll the left sidebar to the current location on page load
- Also the formatting in pdf uses link formatting instead of code formatting

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/index.rst, line 10.)

Todo

nextpnr for FPGAs, consider mentioning openlane, vpr, coriolis

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/introduction.rst, line 76.)

Todo

Consider a less academic version of the History of Yosys

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/introduction.rst, line 206.)

Todo

pending merge of https://github.com/YosysHQ/yosys/pull/5068

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/bugpoint.rst, line 4.)

Todo

brief overview for the more scripting index

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/index.rst, line 4.)

Todo

troubleshooting document(?)

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/index.rst, line 6.)

Todo

interactive design opening text

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/interactive_investigation.rst, line 4.)

Todo

merge into *Scripting in Yosys* show section

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/interactive_investigation.rst, line 14.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/interactive_investigation.rst, line 363.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/interactive_investigation.rst, line 395.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/interactive_investigation.rst, line 483.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/interactive_investigation.rst, line 553.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/interactive_investigation.rst, line 597.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/interactive_investigation.rst, line 643.)

Todo

replace inline code?

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/interactive_investigation.rst, line 696.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/interactive_investigation.rst, line 721.)

Todo

link note to as-yet non-existent section on `RTLIL::Pass` under *Working with the Yosys codebase*

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/load_design.rst, line 84.)

Todo

figure out this example code block

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/load_design.rst, line 104.)

Todo

does *`write_file`* count?

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/load_design.rst, line 133.)

Todo

check text context

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/model_checking.rst, line 4.)

Todo

add/expand supporting text

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/model_checking.rst, line 25.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/model_checking.rst, line 47.)

Todo

add/expand supporting text

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/model_checking.rst, line 91.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/model_checking.rst, line 107.)

Todo

reduce overlap with *Scripting in Yosys* select section

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/selections.rst, line 10.)

Todo

pending discussion on whether rule ordering is a bug or a feature

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/selections.rst, line 340.)

Todo

reflow for not presentation

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/more_scripting/selections.rst, line 385.)

Todo

more about logic minimization & register balancing et al with ABC

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/abc.rst, line 102.)

Todo

find a Liberty pygments style?

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/cell_libs.rst, line 93.)

Todo

add/expand supporting text, also mention custom pattern matching and pmgen

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/extract.rst, line 12.)

Todo

add/expand supporting text

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/extract.rst, line 103.)

Todo

Make `memory_*` notes less quick

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/memory.rst, line 18.)

Todo

describe `memory` images

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/memory.rst, line 34.)

Todo

assorted enables, e.g. cen, wen+ren

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/memory.rst, line 233.)

Todo

“outlines these optimizations” or “outlines *some*..”?

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/opt.rst, line 7.)

Todo

unsure if this is too much detail and should be in *Yosys internals*

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/opt.rst, line 27.)

Todo

`$_DFF_` isn’t a valid cell

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/opt.rst, line 185.)

Todo

more on the other optimizations

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/opt.rst, line 234.)

Todo

describe `proc` images

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/proc.rst, line 29.)

Todo

comment on common `synth_*` options, like `-run`

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/synth.rst, line 4.)

Todo

less academic, check text is coherent

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/techmap_synth.rst, line 4.)

Todo

newer techmap libraries appear to be largely `.v` instead of `.lib`

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/using_yosys/synthesis/techmap_synth.rst, line 76.)

Todo

check text is coherent

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/extending_yosys/extensions.rst, line 7.)

Todo

update to use `/code_examples/extensions/test*.log`

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/extending_yosys/extensions.rst, line 9.)

Todo

mention coding guide

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/extending_yosys/extensions.rst, line 14.)

Todo

consider replacing inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/extending_yosys/extensions.rst, line 44.)

Todo

add/expand supporting text

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/extending_yosys/extensions.rst, line 62.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/extending_yosys/extensions.rst, line 84.)

Todo

use my_cmd.cc literalincludes

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/extending_yosys/extensions.rst, line 171.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/extending_yosys/extensions.rst, line 213.)

Todo

replace inline code

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/extending_yosys/extensions.rst, line 228.)

Todo

adding tests (makefile-tests vs seed-tests)

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/extending_yosys/test_suites.rst, line 4.)

Todo

less academic

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/flow/control_and_data.rst, line 4.)

Todo

less academic

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/flow/overview.rst, line 4.)

Todo

Synthesizing Verilog arrays

Add some information on the generation of *$memrd* and *$memwr* cells and how they are processed in the memory pass.

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/flow/verilog_frontend.rst, line 648.)

Todo

Synthesizing parametric designs

Add some information on the `RTLIL::Module::derive()` method and how it is used to synthesize parametric modules via the hierarchy pass.

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/flow/verilog_frontend.rst, line 654.)

Todo

less academic

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/index.rst, line 6.)

Todo

add RISC-V core example

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/index.rst, line 18.)

Todo

add/expand supporting text

(The original entry is located in /builddir/build/BUILD/yosys-0.66-build/yosys-86f2ddebce7e98ce7cacc27e8a5c14cb53b51b51/docs/source/yosys_internals/techmap.rst, line 25.)

CHAPTER

ONE

WHAT IS YOSYS

Yosys began as a BSc thesis project by Claire Wolf intended to support synthesis for a CGRA (coarse-grained reconfigurable architecture). It then expanded into more general infrastructure for research on synthesis.

Modern Yosys has full support for the synthesizable subset of Verilog-2005 and has been described as "the GCC of hardware synthesis." Freely available and open source, Yosys finds use across hobbyist and commercial applications as well as academic.

Note

Yosys is released under the ISC License:

A permissive license lets people do anything with your code with proper attribution and without warranty. The ISC license is functionally equivalent to the BSD 2-Clause and MIT licenses, removing some language that is no longer necessary.

Together with the place and route tool nextpnr, Yosys can be used to program some FPGAs with a fully end-to-end open source flow (Lattice iCE40 and ECP5). It also does the synthesis portion for the OpenLane flow, targeting the SkyWater 130nm open source PDK for fully open source ASIC design. Yosys can also do formal verification with backends for solver formats like SMT2.

Yosys, and the accompanying Open Source EDA ecosystem, is currently maintained by Yosys Headquarters, with many of the core developers employed by YosysHQ GmbH. A commercial extension, Tabby CAD Suite, includes the Verific frontend for industry-grade SystemVerilog and VHDL support, formal verification with SVA, and formal apps.

1.1 What you can do with Yosys

- Read and process (most of) modern Verilog-2005 code
- Perform all kinds of operations on netlist (RTL, Logic, Gate)
- Perform logic optimizations and gate mapping with ABC

1.1.1 Typical applications for Yosys

- Synthesis of final production designs
- Pre-production synthesis (trial runs before investing in other tools)
- Conversion of full-featured Verilog to simple Verilog
- Conversion of Verilog to other formats (BLIF, BTOR, etc)
- Demonstrating synthesis algorithms (e.g. for educational purposes)
- Framework for experimenting with new algorithms
- Framework for building custom flows (Not limited to synthesis but also formal verification, reverse engineering, . . .)

1.1.2 Things you can't do

- Process high-level languages such as C/C++/SystemC
- Create physical layouts (place&route)
  - Check out nextpnr for that
- Rely on built-in syntax checking
  - Use an external tool like verilator instead

Todo

nextpnr for FPGAs, consider mentioning openlane, vpr, coriolis

1.2 The Yosys family

As mentioned above, YosysHQ maintains not just Yosys but an entire family of tools built around it. In no particular order:

SBY for formal verification
: Yosys provides input parsing and conversion to the formats used by the solver engines. Yosys also provides a unified witness framework for providing cover traces and counter examples for engines which don't natively support this. SBY source | SBY docs

EQY for equivalence checking
: In addition to input parsing and preparation, Yosys provides the plugin support enabling EQY to operate on designs directly. EQY source | EQY docs

MCY for mutation coverage
: Yosys is used to read the source design, generate a list of possible mutations to maximise design coverage, and then perform selected mutations. MCY source | MCY docs

SCY for deep formal traces
: Since SCY generates and runs SBY, Yosys provides the same utility for SCY as it does for SBY. Yosys additionally provides the trace concatenation needed for outputting the deep traces. SCY source

1.3 The original thesis abstract

The first version of the Yosys documentation was published as a bachelor thesis at the Vienna University of Technology [Wol13].

Abstract

Most of today's digital design is done in HDL code (mostly Verilog or VHDL) and with the help of HDL synthesis tools.

In special cases such as synthesis for coarse-grain cell libraries or when testing new synthesis algorithms it might be necessary to write a custom HDL synthesis tool or add new features to an existing one. In these cases the availability of a Free and Open Source (FOSS) synthesis tool that can be used as basis for custom tools would be helpful.

In the absence of such a tool, the Yosys Open SYnthesis Suite (Yosys) was developed. This document covers the design and implementation of this tool. At the moment the main focus of Yosys lies on the high-level aspects of digital synthesis. The pre-existing FOSS logic-synthesis tool ABC is used by Yosys to perform advanced gate-level optimizations.

An evaluation of Yosys based on real-world designs is included. It is shown that Yosys can be used as-is to synthesize such designs. The results produced by Yosys in this tests where successfully verified using formal verification and are comparable in quality to the results produced by a commercial synthesis tool.

Yosys is a Verilog HDL synthesis tool. This means that it takes a behavioural design description as input and generates an RTL, logical gate or physical gate level description of the design as output. Yosys' main strengths are behavioural and RTL synthesis. A wide range of commands (synthesis passes) exist within Yosys that can be used to perform a wide range of synthesis tasks within the domain of behavioural, rtl and logic synthesis. Yosys is designed to be extensible and therefore is a good basis for implementing custom synthesis tools for specialised tasks.

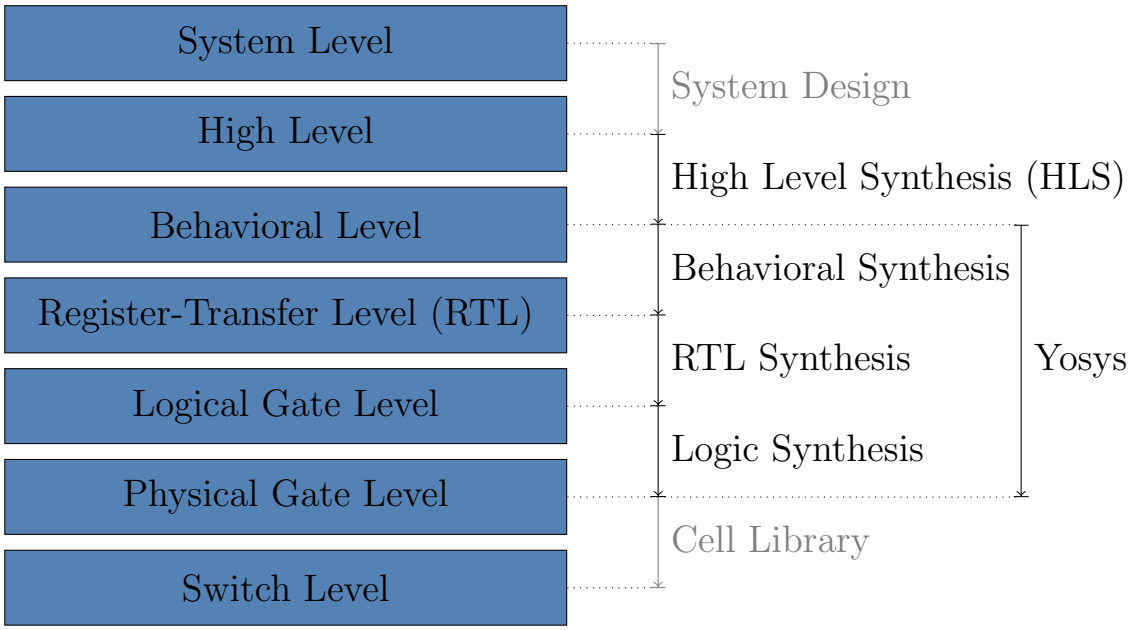


Fig. 1.1: Where Yosys exists in the layers of abstraction

1.3.1 Benefits of open source HDL synthesis

- Cost (also applies to `free as in free beer` solutions):

  Today the cost for a mask set in 180nm technology is far less than the cost for the design tools needed to design the mask layouts. Open Source ASIC flows are an important enabler for ASIC-level Open Source Hardware.

- Availability and Reproducibility:

  If you are a researcher who is publishing, you want to use tools that everyone else can also use. Even if most universities have access to all major commercial tools, you usually do not have easy access to the version that was used in a research project a couple of years ago. With Open Source tools you can even release the source code of the tool you have used alongside your data.

- Framework:

  Yosys is not only a tool. It is a framework that can be used as basis for other developments, so researchers and hackers alike do not need to re-invent the basic functionality. Extensibility was one of Yosys' design goals.

- All-in-one:

  Because of the framework characteristics of Yosys, an increasing number of features become available in one tool. Yosys not only can be used for circuit synthesis but also for formal equivalence checking, SAT solving, and for circuit analysis, to name just a few other application domains. With proprietary software one needs to learn a new tool for each of these applications.

- Educational Tool:

  Proprietary synthesis tools are at times very secretive about their inner workings. They often are `black boxes`. Yosys is very open about its internals and it is easy to observe the different steps of synthesis.

1.3.2 History of Yosys

Todo

Consider a less academic version of the History of Yosys

A Hardware Description Language (HDL) is a computer language used to describe circuits. A HDL synthesis tool is a computer program that takes a formal description of a circuit written in an HDL as input and generates a netlist that implements the given circuit as output.

Currently the most widely used and supported HDLs for digital circuits are Verilog [A+02, A+06] and VHDL (VHSIC HDL, where VHSIC is an acronym for Very-High-Speed Integrated Circuits) [A+04, A+09]. Both HDLs are used for test and verification purposes as well as logic synthesis, resulting in a set of synthesizable and a set of non-synthesizable language features. In this document we only look at the synthesizable subset of the language features.

In recent work on heterogeneous coarse-grain reconfigurable logic [WGS+12] the need for a custom application-specific HDL synthesis tool emerged. It was soon realised that a synthesis tool that understood Verilog or VHDL would be preferred over a synthesis tool for a custom HDL. Given an existing Verilog or VHDL front end, the work for writing the necessary additional features and integrating them in an existing tool can be estimated to be about the same as writing a new tool with support for a minimalistic custom HDL.

The proposed custom HDL synthesis tool should be licensed under a Free and Open Source Software (FOSS) licence. So an existing FOSS Verilog or VHDL synthesis tool would have been needed as basis to build upon.

The main advantages of choosing Verilog or VHDL is the ability to synthesize existing HDL code and to mitigate the requirement for circuit-designers to learn a new language. In order to take full advantage of any existing FOSS Verilog or VHDL tool, such a tool would have to provide a feature-complete implementation of the synthesizable HDL subset.

Basic RTL synthesis is a well understood field [HS96]. Lexing, parsing and processing of computer languages [ASU86] is a thoroughly researched field. All the information required to write such tools has been openly available for a long time, and it is therefore likely that a FOSS HDL synthesis tool with a feature-complete Verilog or VHDL front end must exist which can be used as a basis for a custom RTL synthesis tool.

Due to the author's preference for Verilog over VHDL it was decided early on to go for Verilog instead of VHDL[1]. So the existing FOSS Verilog synthesis tools were evaluated. The results of this evaluation are utterly devastating. Therefore a completely new Verilog synthesis tool was implemented and is recommended as basis for custom synthesis tools. This is the tool that is discussed in this document.

[1] A quick investigation into FOSS VHDL tools yielded similar grim results for FOSS VHDL synthesis tools.

CHAPTER

TWO

GETTING STARTED WITH YOSYS

This section covers how to get started with Yosys, from installation to a guided walkthrough of synthesizing a design for hardware, and finishing with an introduction to writing re-usable Yosys scripts.

2.1 Installation

This document will guide you through the process of installing Yosys.

2.1.1 CAD suite(s)

Yosys is part of the Tabby CAD Suite and the OSS CAD Suite! The easiest way to use yosys is to install the binary software suite, which contains all required dependencies and related tools.

- Contact YosysHQ for a Tabby CAD Suite Evaluation License and download link
- OR go to https://github.com/YosysHQ/oss-cad-suite-build/releases to download the free OSS CAD Suite
- Follow the Install Instructions on GitHub

Make sure to get a Tabby CAD Suite Evaluation License if you need features such as industry-grade SystemVerilog and VHDL parsers!

For more information about the difference between Tabby CAD Suite and the OSS CAD Suite, please visit https://www.yosyshq.com/tabby-cad-datasheet

Many Linux distributions also provide Yosys binaries, some more up to date than others. Check with your package manager!

Targeted architectures

The OSS CAD Suite releases nightly builds for the following architectures:

- **linux-x64** - Most personal Linux based computers
- **darwin-x64** - macOS 12 or later with Intel CPU
- **darwin-arm64** - macOS 12 or later with M1/M2 CPU
- **windows-x64** - Targeted for Windows 10 and 11
- **linux-arm64** - Devices such as Raspberry Pi with 64bit OS

For more information about the targeted architectures, and the current build status, check the OSS CAD Suite git repository.

2.1.2 Building from source

The Yosys source files can be obtained from the YosysHQ/Yosys git repository. ABC and some of the other libraries used are included as git submodules. To clone these submodules at the same time, use e.g.:

```
git clone --recurse-submodules https://github.com/YosysHQ/yosys.git  # ..or..
git clone https://github.com/YosysHQ/yosys.git
cd yosys
git submodule update --init --recursive
```

Note

As of Yosys v0.47, releases include a `yosys.tar.gz` file which includes all source code and all sub-modules in a single archive. This can be used as an alternative which does not rely on `git`.

Supported platforms

The following platforms are supported and regularly tested:

- Linux
- macOS

Other platforms which may work, but instructions may not be up to date and are not regularly tested:

- FreeBSD
- WSL
- Windows with (e.g.) Cygwin

Build prerequisites

A C++ compiler with C++20 support is required as well as some standard tools such as GNU Flex, GNU Bison (>=3.8), Make, and Python (>=3.11). Some additional tools: readline, libffi, Tcl and zlib; are optional but enabled by default (see **`ENABLE_*`** settings in Makefile). Graphviz and Xdot are used by the *`show`* command to display schematics.

Installing all prerequisites:

```
sudo apt-get install gawk git make python3 lld bison clang flex \
   libffi-dev libfl-dev libreadline-dev pkg-config tcl-dev zlib1g-dev \
   graphviz xdot
curl -LsSf https://astral.sh/uv/install.sh | sh
```

```
brew tap Homebrew/bundle && brew bundle
```

```
sudo port install bison flex readline gawk libffi graphviz \
   pkgconfig python311 zlib tcl
```

```
pkg install bison flex readline gawk libffi graphviz \
   pkgconf python311 tcl-wrapper
```

Note

On FreeBSD system use gmake instead of make. To run tests use: `MAKE=gmake CXX=cxx CC=cc gmake test`

Use the following command to install all prerequisites, or select these additional packages:

```
setup-x86_64.exe -q --packages=bison,flex,gcc-core,gcc-g++,git,libffi-devel,libreadline-
↪devel,make,pkg-config,python3,tcl-devel,zlib-devel
```

Warning

As of this writing, Cygwin only supports up to Python 3.9.16 while the minimum required version of Python is 3.11. This means that Cygwin is not compatible with many of the Python-based frontends. While this does not currently prevent Yosys itself from working, no guarantees are made for continued support. You may also need to specify `CXXSTD=gnu++17` to resolve missing `strdup` function when using gcc. It is instead recommended to use Windows Subsystem for Linux (WSL) and follow the instructions for Ubuntu.

Build configuration

The Yosys build is based solely on Makefiles, and uses a number of variables which influence the build process. The recommended method for configuring builds is with a `Makefile.conf` file in the root `yosys` directory. The following commands will clean the directory and provide an initial configuration file:

```
make config-clang    # ..or..
make config-gcc
```

Check the root Makefile to see what other configuration targets are available. Other variables can then be added to the `Makefile.conf` as needed, for example:

```
echo "ENABLE_ZLIB := 0" >> Makefile.conf
```

Using one of these targets will set the `CONFIG` variable to something other than `none`, and will override the environment variable for `CXX`. To use a different compiler than the default when building, use:

```
make CXX=$CXX        # ..or..
make CXX="g++-11"
```

Note

Setting the compiler in this way will prevent some other options such as `ENABLE_CCACHE` from working as expected.

If you have clang, and (a compatible version of) `ld.lld` available in PATH, it's recommended to speed up incremental builds with lld by enabling LTO with `ENABLE_LTO=1`. On macOS, LTO requires using clang from homebrew rather than clang from xcode. For example:

```
make ENABLE_LTO=1 CXX=$(brew --prefix)/opt/llvm/bin/clang++
```

By default, building (and installing) yosys will build (and install) ABC, using `yosys-abc` as the executable name. To use an existing ABC executable instead, set the `ABCEXTERNAL` make variable to point to the desired executable.

Running the build system

From the root `yosys` directory, call the following commands:

```
make
sudo make install
```

To use a separate (out-of-tree) build directory, provide a path to the Makefile.

```
mkdir build; cd build
make -f ../Makefile
```

Out-of-tree builds require a clean source tree.

See also

Refer to *Testing Yosys* for details on testing Yosys once compiled.

2.1.3 Source tree and build system

The Yosys source tree is organized into the following top-level directories:

`backends/`
: This directory contains a subdirectory for each of the backend modules.

`docs/`
: Contains the source for this documentation, including images and sample code.

`examples/`
: Contains example code for using Yosys with some other tools including a demo of the Yosys Python api, and synthesizing for various toolchains such as Intel and Anlogic.

`frontends/`
: This directory contains a subdirectory for each of the frontend modules.

`kernel/`
: This directory contains all the core functionality of Yosys. This includes the functions and definitions for working with the RTLIL data structures (`rtlil.h/cc`), the `main()` function (`driver.cc`), the internal framework for generating log messages (`log.h/cc`), the internal framework for registering and calling passes (`register.h/cc`), some core commands that are not really passes (`select.cc`, `show.cc`, ...) and a couple of other small utility libraries.

`libs/`
: Libraries packaged with Yosys builds are contained in this folder. See *Auxiliary libraries*.

`misc/`
: Other miscellany which doesn't fit anywhere else.

`passes/`
: This directory contains a subdirectory for each pass or group of passes. For example as of this writing the directory `passes/hierarchy/` contains the code for three passes: `hierarchy`, `submod`, and `uniquify`.

`techlibs/`
: This directory contains simulation models and standard implementations for the cells from the internal cell library.

`tests/`
: This directory contains the suite of unit tests and regression tests used by Yosys. See *Testing Yosys*.

The top-level Makefile includes `frontends/*/Makefile.inc`, `passes/*/Makefile.inc` and `backends/*/Makefile.inc`. So when extending Yosys it is enough to create a new directory in `frontends/`, `passes/` or `backends/` with your sources and a `Makefile.inc`. The Yosys kernel automatically detects all commands linked with Yosys. So it is not needed to add additional commands to a central list of commands.

Good starting points for reading example source code to learn how to write passes are `passes/opt/opt_dff.cc` and `passes/opt/opt_merge.cc`.

Users of the Qt Creator IDE can generate a QT Creator project file using make qtcreator. Users of the Eclipse IDE can use the "Makefile Project with Existing Code" project type in the Eclipse "New Project" dialog (only available after the CDT plugin has been installed) to create an Eclipse project in order to programming extensions to Yosys or just browse the Yosys code base.

2.2 Synthesis starter

This page will be a guided walkthrough of the prepackaged iCE40 FPGA synthesis script - `synth_ice40`. We will take a simple design through each step, looking at the commands being called and what they do to the design. While `synth_ice40` is specific to the iCE40 platform, most of the operations we will be discussing are common across the majority of FPGA synthesis scripts. Thus, this document will provide a good foundational understanding of how synthesis in Yosys is performed, regardless of the actual architecture being used.

See also

Advanced usage docs for *Synth commands*

2.2.1 Demo design

First, let's quickly look at the design we'll be synthesizing:

Todo

reconsider including the whole (~77 line) design like this

Listing 2.1: `fifo.v`

```
// address generator/counter
module addr_gen
#(  parameter MAX_DATA=256,
        localparam AWIDTH = $clog2(MAX_DATA)
) ( input en, clk, rst,
        output reg [AWIDTH-1:0] addr
);
        initial addr = 0;

```

(continues on next page)

(continued from previous page)

```
10          // async reset
11          // increment address when enabled
12          always @(posedge clk or posedge rst)
13                  if (rst)
14                          addr <= 0;
15                  else if (en) begin
16                          if ({'0, addr} == MAX_DATA-1)
17                                  addr <= 0;
18                          else
19                                  addr <= addr + 1;
20                  end
21  endmodule //addr_gen
22
23  // Define our top level fifo entity
24  module fifo
25  #(  parameter MAX_DATA=256,
26          localparam AWIDTH = $clog2(MAX_DATA)
27  ) ( input wen, ren, clk, rst,
28          input [7:0] wdata,
29          output reg [7:0] rdata,
30          output reg [AWIDTH:0] count
31  );
32          // fifo storage
33          // sync read before write
34          wire [AWIDTH-1:0] waddr, raddr;
35          reg [7:0] data [MAX_DATA-1:0];
36          always @(posedge clk) begin
37                  if (wen)
38                          data[waddr] <= wdata;
39                  rdata <= data[raddr];
40          end // storage
41
42          // addr_gen for both write and read addresses
43          addr_gen #(.MAX_DATA(MAX_DATA))
44          fifo_writer (
45                  .en     (wen),
46                  .clk    (clk),
47                  .rst    (rst),
48                  .addr   (waddr)
49          );
50
51          addr_gen #(.MAX_DATA(MAX_DATA))
52          fifo_reader (
53                  .en     (ren),
54                  .clk    (clk),
55                  .rst    (rst),
56                  .addr   (raddr)
57          );
58
59          // status signals
60          initial count = 0;
61
```

(continues on next page)

(continued from previous page)

```
62          always @(posedge clk or posedge rst) begin
63                  if (rst)
64                          count <= 0;
65                  else if (wen && !ren)
66                          count <= count + 1;
67                  else if (ren && !wen)
68                          count <= count - 1;
69          end
70
71  endmodule
```

Todo

fifo.v description

While the open source *read_verilog* frontend generally does a pretty good job at processing valid Verilog input, it does not provide very good error handling or reporting. Using an external tool such as verilator before running Yosys is highly recommended. We can quickly check the Verilog syntax of our design by calling `verilator --lint-only fifo.v`.

2.2.2 Loading the design

Let's load the design into Yosys. From the command line, we can call `yosys fifo.v`. This will open an interactive Yosys shell session and immediately parse the code from *fifo.v* and convert it into an Abstract Syntax Tree (AST). If you are interested in how this happens, there is more information in the document, *The Verilog and AST frontends*. For now, suffice it to say that we do this to simplify further processing of the design. You should see something like the following:

```
$ yosys fifo.v

-- Parsing `fifo.v' using frontend ` -vlog2k' --

1. Executing Verilog-2005 frontend: fifo.v
Parsing Verilog input from `fifo.v' to AST representation.
Storing AST representation for module `$abstract\addr_gen'.
Storing AST representation for module `$abstract\fifo'.
Successfully finished Verilog frontend.
```

See also

Advanced usage docs for *Loading a design*

2.2.3 Elaboration

Now that we are in the interactive shell, we can call Yosys commands directly. Our overall goal is to call `synth_ice40 -top fifo`, but for now we can run each of the commands individually for a better sense of how each part contributes to the flow. We will also start with just a single module; `addr_gen`.

At the bottom of the `help` output for `synth_ice40` is the complete list of commands called by this script. Let's start with the section labeled `begin`:

Listing 2.2: `begin` section

```
read_verilog -D ICE40_HX -lib -specify +/ice40/cells_sim.v
hierarchy -check -top <top>
proc
```

`read_verilog -D ICE40_HX -lib -specify +/ice40/cells_sim.v` loads the iCE40 cell models which allows us to include platform specific IP blocks in our design. PLLs are a common example of this, where we might need to reference `SB_PLL40_CORE` directly rather than being able to rely on mapping passes later. Since our simple design doesn't use any of these IP blocks, we can skip this command for now. Because these cell models will also be needed once we start mapping to hardware we will still need to load them later.

Note

`+/` is a dynamic reference to the Yosys `share` directory. By default, this is `/usr/local/share/yosys`. If using a locally built version of Yosys from the source directory, this will be the `share` folder in the same directory.

The addr_gen module

Since we're just getting started, let's instead begin with `hierarchy -top addr_gen`. This command declares that the top level module is `addr_gen`, and everything else can be discarded.

Listing 2.3: `addr_gen` module source

```
module addr_gen
#(  parameter MAX_DATA=256,
        localparam AWIDTH = $clog2(MAX_DATA)
) ( input en, clk, rst,
        output reg [AWIDTH-1:0] addr
);
        initial addr = 0;

        // async reset
        // increment address when enabled
        always @(posedge clk or posedge rst)
                if (rst)
                        addr <= 0;
                else if (en) begin
                        if ({'0, addr} == MAX_DATA-1)
                                addr <= 0;
                        else
                                addr <= addr + 1;
                end
endmodule //addr_gen
```

Note

`hierarchy` should always be the first command after the design has been read. By specifying the top module, `hierarchy` will also set the `(* top *)` attribute on it. This is used by other commands that need to know which module is the top.

Listing 2.4: `hierarchy -top addr_gen` output

```
yosys> hierarchy -top addr_gen

2. Executing HIERARCHY pass (managing design hierarchy).

3. Executing AST frontend in derive mode using pre-parsed AST for module `\addr_gen'.
Generating RTLIL representation for module `\addr_gen'.

3.1. Analyzing design hierarchy..
Top module:  \addr_gen

3.2. Analyzing design hierarchy..
Top module:  \addr_gen
Removing unused module `$abstract\fifo'.
Removing unused module `$abstract\addr_gen'.
Removed 2 unused modules.
```

Our `addr_gen` circuit now looks like this:

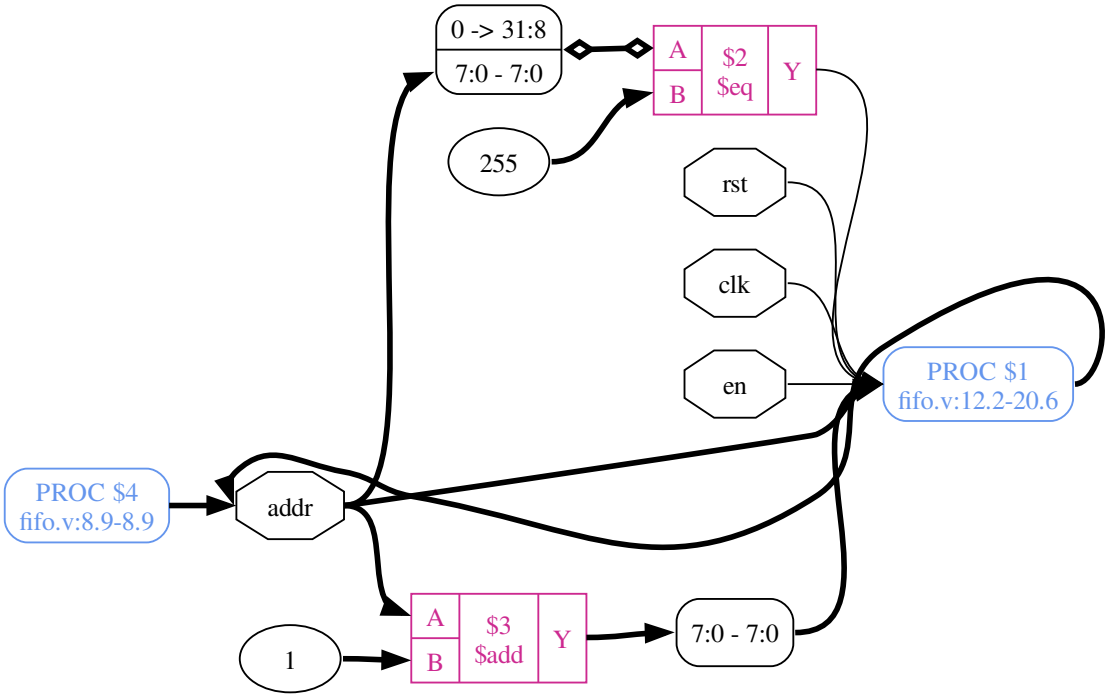


Fig. 2.1: `addr_gen` module after `hierarchy`

Simple operations like `addr + 1` and `addr == MAX_DATA-1` can be extracted from our `always @` block in *addr_gen module source*. This gives us the highlighted *`$add`* and *`$eq`* cells we see. But control logic (like the `if .. else`) and memory elements (like the `addr <= 0`) are not so straightforward. These get put into "processes", shown in the schematic as `PROC`. Note how the second line refers to the line numbers of the

start/end of the corresponding `always @` block. In the case of an `initial` block, we instead see the `PROC` referring to line 0.

To handle these, let us now introduce the next command: *proc - translate processes to netlists*. *`proc`* is a macro command like `synth_ice40`. Rather than modifying the design directly, it instead calls a series of other commands. In the case of *`proc`*, these sub-commands work to convert the behavioral logic of processes into multiplexers and registers. Let's see what happens when we run it. For now, we will call `proc -noopt` to prevent some automatic optimizations which would normally happen.

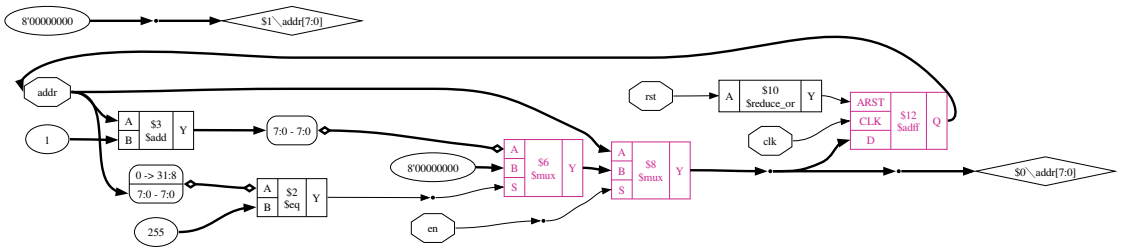


Fig. 2.2: `addr_gen` module after `proc -noopt`

There are now a few new cells from our `always @`, which have been highlighted. The `if` statements are now modeled with *`$mux`* cells, while the register uses an *`$adff`* cell. If we look at the terminal output we can also see all of the different `proc_*` commands being called. We will look at each of these in more detail in *Converting process blocks*.

Notice how in the top left of *addr_gen module after proc -noopt* we have a floating wire, generated from the initial assignment of 0 to the `addr` wire. However, this initial assignment is not synthesizable, so this will need to be cleaned up before we can generate the physical hardware. We can do this now by calling *`clean`*. We're also going to call *`opt_expr`* now, which would normally be called at the end of *`proc`*. We can call both commands at the same time by separating them with a colon and space: `opt_expr; clean`.

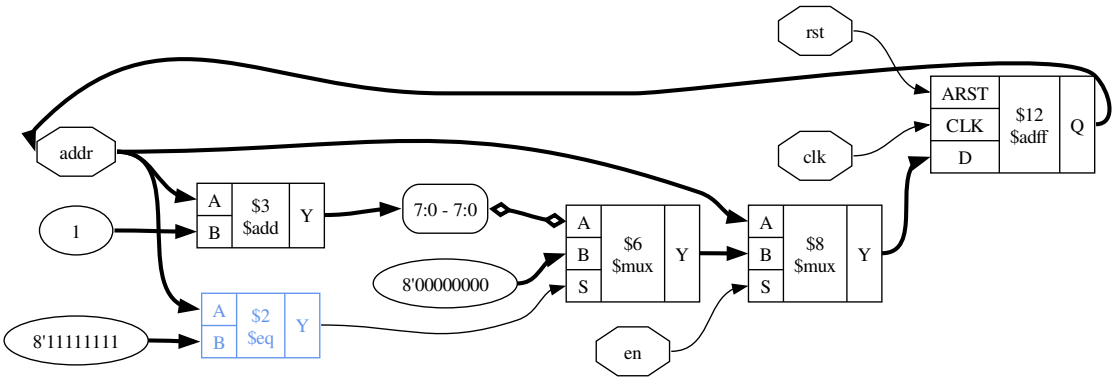


Fig. 2.3: `addr_gen` module after `opt_expr; clean`

You may also notice that the highlighted *`$eq`* cell input of `255` has changed to `8'11111111`. Constant values are presented in the format `<bit_width>'<bits>`, with 32-bit values instead using the decimal number.

This indicates that the constant input has been reduced from 32-bit wide to 8-bit wide. This is a side-effect of running *opt_expr*, which performs constant folding and simple expression rewriting. For more on why this happens, refer to *Optimization passes* and the *section on opt_expr*.

Note

clean - remove unused cells and wires can also be called with two semicolons after any command, for example we could have called `opt_expr;;` instead of `opt_expr; clean`. You may notice some scripts will end each line with `;;`. It is beneficial to run *clean* before inspecting intermediate products to remove disconnected parts of the circuit which have been left over, and in some cases can reduce the processing required in subsequent commands.

Todo

consider a brief glossary for terms like adff

See also

Advanced usage docs for

- *Converting process blocks*
- *Optimization passes*

The full example

Let's now go back and check on our full design by using `hierarchy -check -top fifo`. By passing the `-check` option there we are also telling the `hierarchy` command that if the design includes any non-blackbox modules without an implementation it should return an error.

Note that if we tried to run this command now then we would get an error. This is because we already removed all of the modules other than `addr_gen`. We could restart our shell session, but instead let's use two new commands:

- `design`, and
- *read_verilog - read modules from Verilog file*.

Listing 2.5: reloading `fifo.v` and running `hierarchy -check -top fifo`

```
yosys> design -reset

yosys> read_verilog fifo.v

11. Executing Verilog-2005 frontend: fifo.v
Parsing Verilog input from `fifo.v' to AST representation.
Generating RTLIL representation for module `\addr_gen'.
Generating RTLIL representation for module `\fifo'.
Successfully finished Verilog frontend.

yosys> hierarchy -check -top fifo
```

(continues on next page)

(continued from previous page)

```
12. Executing HIERARCHY pass (managing design hierarchy).

12.1. Analyzing design hierarchy..
Top module:  \fifo
Used module:     \addr_gen
Parameter \MAX_DATA = 256

12.2. Executing AST frontend in derive mode using pre-parsed AST for module `\addr_gen'.
Parameter \MAX_DATA = 256
Generating RTLIL representation for module `$paramod\addr_gen\MAX_DATA=s32
↪'00000000000000000000000100000000'.
Parameter \MAX_DATA = 256
Found cached RTLIL representation for module `$paramod\addr_gen\MAX_DATA=s32
↪'00000000000000000000000100000000'.

12.3. Analyzing design hierarchy..
Top module:  \fifo
Used module:     $paramod\addr_gen\MAX_DATA=s32'00000000000000000000000100000000

12.4. Analyzing design hierarchy..
Top module:  \fifo
Used module:     $paramod\addr_gen\MAX_DATA=s32'00000000000000000000000100000000
Removing unused module `\addr_gen'.
Removed 1 unused modules.
```

Notice how this time we didn't see any of those `$abstract` modules? That's because when we ran `yosys fifo.v`, the first command Yosys called was `read_verilog -defer fifo.v`. The `-defer` option there tells *`read_verilog`* only read the abstract syntax tree and defer actual compilation to a later `hierarchy` command. This is useful in cases where the default parameters of modules yield invalid code which is not synthesizable. This is why Yosys defers compilation automatically and is one of the reasons why hierarchy should always be the first command after loading the design. If we know that our design won't run into this issue, we can skip the `-defer`.

Todo

`hierarchy` failure modes

Note

The number before a command's output increments with each command run. Don't worry if your numbers don't match ours! The output you are seeing comes from the same script that was used to generate the images in this document, included in the source as `fifo.ys`. There are extra commands being run which you don't see, but feel free to try them yourself, or play around with different commands. You can always start over with a clean slate by calling `exit` or hitting `ctrl+d` (i.e. EOF) and re-launching the Yosys interactive terminal. `ctrl+c` (i.e. SIGINT) will also end the terminal session but will return an error code rather than exiting gracefully.

We can also run *`proc`* now to finish off the full *begin section*. Because the design schematic is quite large, we will be showing just the data path for the `rdata` output. If you would like to see the entire design for

yourself, you can do so with *show - generate schematics using graphviz*. Note that the *show* command only works with a single module, so you may need to call it with `show fifo`. *Displaying schematics* section in *Scripting in Yosys* has more on how to use *show*.

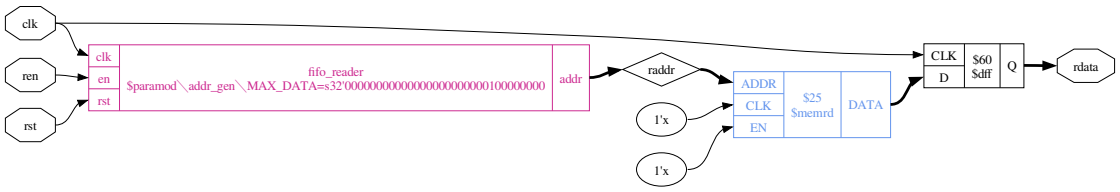


Fig. 2.4: `rdata` output after *proc*

The highlighted `fifo_reader` block contains an instance of the *addr_gen module after proc -noopt* that we looked at earlier. Notice how the type is shown as `$paramod\\addr_gen\\MAX_DATA=s32'...`. This is a "parametric module": an instance of the `addr_gen` module with the `MAX_DATA` parameter set to the given value.

The other highlighted block is a *$memrd* cell. At this stage of synthesis we don't yet know what type of memory is going to be implemented, but we *do* know that `rdata <= data[raddr];` could be implemented as a read from memory. Note that the *$memrd* cell here is asynchronous, with both the clock and enable signal undefined; shown with the `1'x` inputs.

See also

Advanced usage docs for *Converting process blocks*

2.2.4 Flattening

At this stage of a synthesis flow there are a few other commands we could run. In `synth_ice40` we get these:

Listing 2.6: `flatten` section

```
check
flatten
tribuf -logic
deminout
```

We start by runnning *check*. This doesn't affect the design, but it can identify a few obvious problems which will cause errors later. Calling it here lets us fail faster rather than wasting time on something we know is impossible. Next up is `flatten`. Flattening the design like this can allow for optimizations between modules which would otherwise be missed. Let's run `flatten;;` on our design.

Listing 2.7: output of `flatten;;`

```
yosys> flatten

15. Executing FLATTEN pass (flatten design).
Deleting now unused module $paramod\addr_gen\MAX_DATA=s32
↪'00000000000000000000000100000000.
```

(continues on next page)

(continued from previous page)

```
<suppressed ~2 debug messages>

yosys> clean
Removed 3 unused cells and 28 unused wires.
```

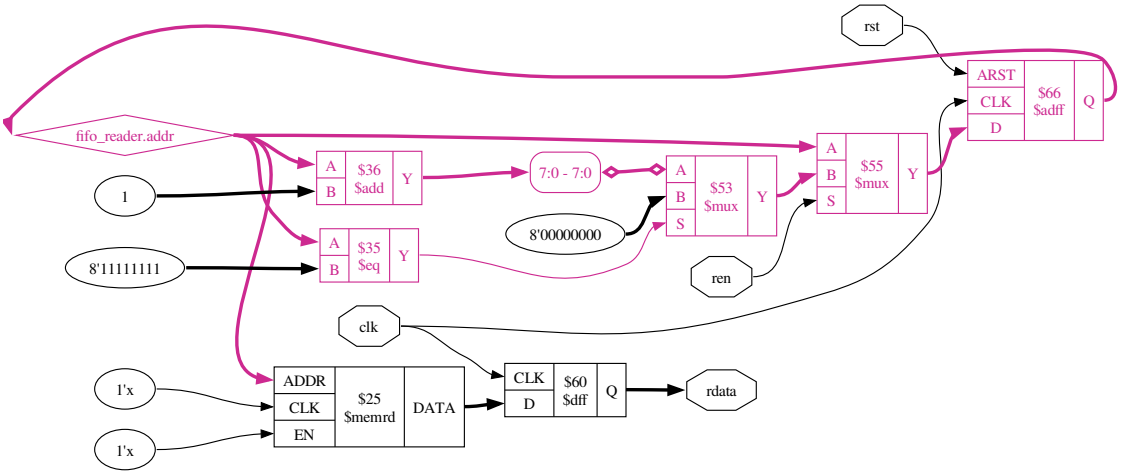


Fig. 2.5: `rdata` output after `flatten;;`

The pieces have moved around a bit, but we can see *addr_gen module after proc -noopt* from earlier has replaced the `fifo_reader` block in *rdata output after proc*. We can also see that the `addr` output has been renamed to `fifo_reader.addr` and merged with the `raddr` wire feeding into the *`$memrd`* cell. This wire merging happened during the call to *`clean`* which we can see in the *output of flatten;;*.

Note

`flatten` and *`clean`* would normally be combined into a single `yosys> flatten;;` output, but they appear separately here as a side effect of using `echo` for generating the terminal style output.

Depending on the target architecture, this stage of synthesis might also see commands such as `tribuf` with the `-logic` option and `deminout`. These remove tristate and inout constructs respectively, replacing them with logic suitable for mapping to an FPGA. Since we do not have any such constructs in our example running these commands does not change our design.

2.2.5 The coarse-grain representation

At this stage, the design is in coarse-grain representation. It still looks recognizable, and cells are word-level operators with parametrizable width. This is the stage of synthesis where we do things like const propagation, expression rewriting, and trimming unused parts of wires.

This is also where we convert our FSMs and hard blocks like DSPs or memories. Such elements have to be inferred from patterns in the design and there are special passes for each. Detection of these patterns can also be affected by optimizations and other transformations done previously.

Note

While the iCE40 flow had a *flatten section* and put *proc* in the *begin section*, some synthesis scripts will instead include these in this section.

Part 1

In the iCE40 flow, we start with the following commands:

Listing 2.8: `coarse` section (part 1)

```
opt_expr
opt_clean
check
opt -nodffe -nosdff
fsm
opt
```

We've already come across *opt_expr* and *check*, and *opt_clean* is the same as *clean* but with more verbose output. Next up is `opt -nodffe -nosdff` performing a set of simple optimizations on the design. This command also ensures that only a specific subset of FF types are included, in preparation for the next command: *fsm - extract and optimize finite state machines*. Both *opt* and *fsm* are macro commands which are explored in more detail in *Optimization passes* and *FSM handling* respectively.

Up until now, the data path for `rdata` has remained the same since *rdata output after flatten;;*. However the next call to *opt* does cause a change. Specifically, the call to *opt_dff* without the `-nodffe -nosdff` options is able to fold one of the *$mux* cells into the *$adff* to form an *$adffe* cell; highlighted below:

Listing 2.9: output of *opt_dff*

```
yosys> opt_dff

17. Executing OPT_DFF pass (perform DFF optimizations).
Adding EN signal on $procdff$59 ($adff) from module fifo (D = $0\count[8:0], Q = \count).
Adding EN signal on $flatten\fifo_writer.$procdff$66 ($adff) from module fifo (D =
↪$flatten\fifo_writer.$procmux$53_Y, Q = \fifo_writer.addr).
Adding EN signal on $flatten\fifo_reader.$procdff$66 ($adff) from module fifo (D =
↪$flatten\fifo_reader.$procmux$53_Y, Q = \fifo_reader.addr).
```

See also

Advanced usage docs for

- *FSM handling*
- *Optimization passes*

Part 2

The next group of commands performs a series of optimizations:

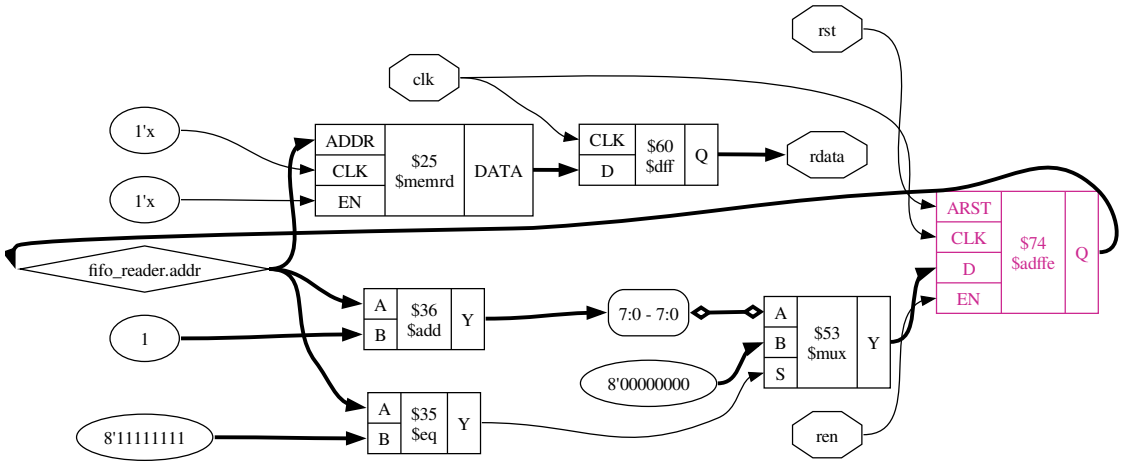


Fig. 2.6: `rdata` output after *opt_dff*

Listing 2.10: `coarse` section (part 2)

```
wreduce
peepopt
opt_clean
share
techmap
opt_expr
opt_clean
memory_dff
```

First up is `wreduce`. If we run this we get the following:

Listing 2.11: output of `wreduce`

```
yosys> wreduce

19. Executing WREDUCE pass (reducing word size of cells).
Removed top 1 bits (of 2) from port B of cell fifo.$auto$opt_dff.cc:235:make_patterns_
↪logic$72 ($ne).
Removed top 31 bits (of 32) from port B of cell fifo.$add$fifo.v:66$29 ($add).
Removed top 23 bits (of 32) from port Y of cell fifo.$add$fifo.v:66$29 ($add).
Removed top 31 bits (of 32) from port B of cell fifo.$sub$fifo.v:68$32 ($sub).
Removed top 23 bits (of 32) from port Y of cell fifo.$sub$fifo.v:68$32 ($sub).
Removed cell fifo.$flatten\fifo_writer.$procmux$55 ($mux).
Removed top 31 bits (of 32) from port B of cell fifo.$flatten\fifo_writer.$add$fifo.v:19
↪$36 ($add).
Removed top 24 bits (of 32) from port Y of cell fifo.$flatten\fifo_writer.$add$fifo.v:19
↪$36 ($add).
Removed cell fifo.$flatten\fifo_reader.$procmux$55 ($mux).
```

(continues on next page)

(continued from previous page)

```
Removed top 31 bits (of 32) from port B of cell fifo.$flatten\fifo_reader.$add$fifo.v:19
↪$36 ($add).
Removed top 24 bits (of 32) from port Y of cell fifo.$flatten\fifo_reader.$add$fifo.v:19
↪$36 ($add).
Removed top 24 bits (of 32) from wire fifo.$flatten\fifo_writer.$add$fifo.v:19$36_Y.
Removed top 23 bits (of 32) from wire fifo.$sub$fifo.v:68$32_Y.
Removed top 23 bits (of 32) from wire fifo.$add$fifo.v:66$29_Y.
Removed top 24 bits (of 32) from wire fifo.$flatten\fifo_reader.$add$fifo.v:19$36_Y.

yosys> show -notitle -format dot -prefix rdata_wreduce o:rdata %ci*

20. Generating Graphviz representation of design.
Writing dot description to `rdata_wreduce.dot'.
Dumping selected parts of module fifo to page 1.

yosys> opt_clean

21. Executing OPT_CLEAN pass (remove unused cells and wires).
Finding unused cells or wires in module \fifo..
Removed 0 unused cells and 6 unused wires.
<suppressed ~1 debug messages>

yosys> memory_dff

22. Executing MEMORY_DFF pass (merging $dff cells to $memrd).
Checking read port `\data'[0] in module `\fifo': merging output FF to cell.
    Write port 0: non-transparent.
```

Looking at the data path for `rdata`, the most relevant of these width reductions are the ones affecting `fifo.$flatten\fifo_reader.$add$fifo.v`. That is the *`$add`* cell incrementing the fifo_reader address. We can look at the schematic and see the output of that cell has now changed.

Todo

pending bugfix in `wreduce` and/or *`opt_clean`*

The next two (new) commands are `peepopt` and `share`. Neither of these affect our design, and they're explored in more detail in *Optimization passes*, so let's skip over them. `techmap -map +/cmp2lut.v -D LUT_WIDTH=4` optimizes certain comparison operators by converting them to LUTs instead. The usage of `techmap` is explored more in *Technology mapping*.

Our next command to run is *memory_dff - merge input/output DFFs into memory read ports*.

Listing 2.12: output of *`memory_dff`*

```
yosys> memory_dff

22. Executing MEMORY_DFF pass (merging $dff cells to $memrd).
Checking read port `\data'[0] in module `\fifo': merging output FF to cell.
    Write port 0: non-transparent.
```

As the title suggests, *`memory_dff`* has merged the output *`$dff`* into the *`$memrd`* cell and converted it to a

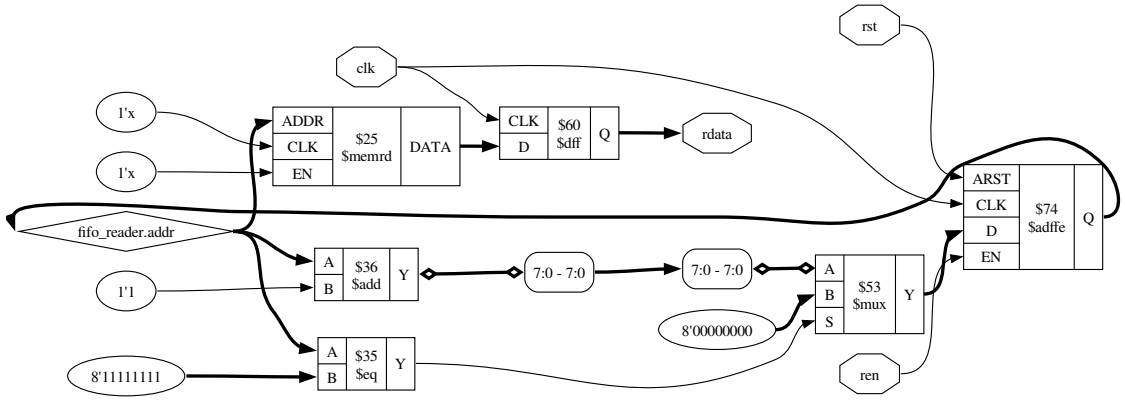


Fig. 2.7: `rdata` output after `wreduce`

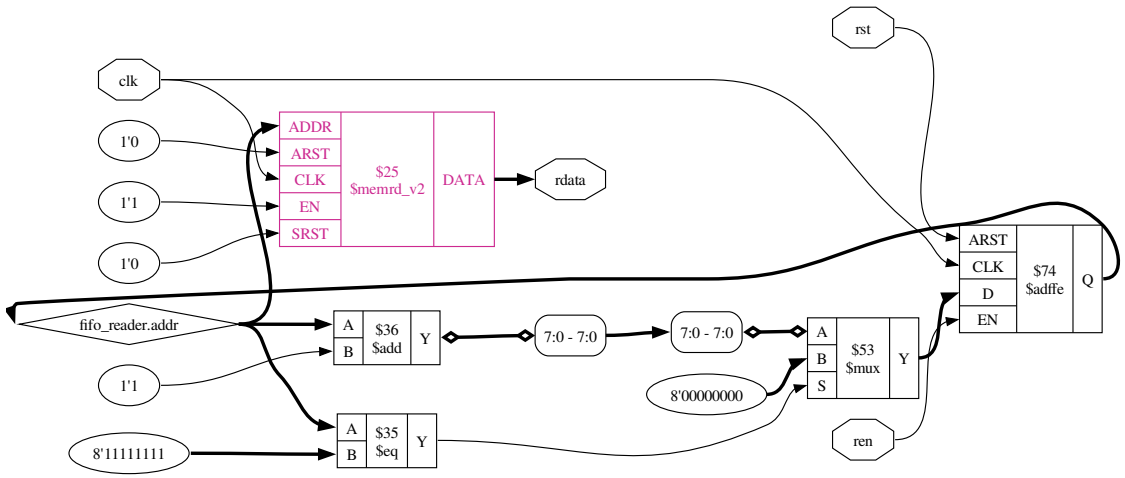


Fig. 2.8: `rdata` output after *`memory_dff`*

$memrd_v2 (highlighted). This has also connected the CLK port to the clk input as it is now a synchronous memory read with appropriate enable (EN=1'1) and reset (ARST=1'0 and SRST=1'0) inputs.

See also

Advanced usage docs for

- *Optimization passes*
- *Technology mapping*
- *Memory handling*

Part 3

The third part of the synth_ice40 flow is a series of commands for mapping to DSPs. By default, the iCE40 flow will not map to the hardware DSP blocks and will only be performed if called with the -dsp flag: synth_ice40 -dsp. While our example has nothing that could be mapped to DSPs we can still take a quick look at the commands here and describe what they do.

Listing 2.13: coarse section (part 3)

```
wreduce t:$mul
techmap
select a:mul2dsp
setattr -unset mul2dsp
opt_expr -fine
wreduce
select -clear
ice40_dsp
chtype -set $mul t:$__soft_mul
```

wreduce t:$mul performs width reduction again, this time targetting only cells of type *$mul*. techmap -map +/mul2dsp.v -map +/ice40/dsp_map.v ... -D DSP_NAME=$__MUL16X16 uses techmap to map *$mul* cells to $__MUL16X16 which are, in turn, mapped to the iCE40 SB_MAC16. Any multipliers which aren't compatible with conversion to $__MUL16X16 are relabelled to $__soft_mul before chtype changes them back to *$mul*.

During the mul2dsp conversion, some of the intermediate signals are marked with the attribute mul2dsp. By calling select a:mul2dsp we restrict the following commands to only operate on the cells and wires used for these signals. setattr removes the now unnecessary mul2dsp attribute. *opt_expr* we've already come across for const folding and simple expression rewriting, the -fine option just enables more fine-grain optimizations. Then we perform width reduction a final time and clear the selection.

Todo

ice40_dsp is pmgen

Finally we have ice40_dsp: similar to the *memory_dff* command we saw in the previous section, this merges any surrounding registers into the SB_MAC16 cell. This includes not just the input/output registers, but also pipeline registers and even a post-adder where applicable: turning a multiply + add into a single multiply-accumulate.

See also

Advanced usage docs for *Technology mapping*

Part 4

That brings us to the fourth and final part for the iCE40 synthesis flow:

Listing 2.14: `coarse` section (part 4)

```
alumacc
opt
memory -nomap [-no-rw-check]
opt_clean
```

Where before each type of arithmetic operation had its own cell, e.g. *$add*, we now want to extract these into *$alu* and *$macc_v2* cells which can help identify opportunities for reusing logic. We do this by running `alumacc`, which we can see produce the following changes in our example design:

Listing 2.15: output of `alumacc`

```
yosys> alumacc

24. Executing ALUMACC pass (create $alu and $macc cells).
Extracting $alu and $macc cells in module fifo:
  creating $macc model for $flatten\fifo_writer.$add$fifo.v:19$36 ($add).
  creating $macc model for $flatten\fifo_reader.$add$fifo.v:19$36 ($add).
  creating $macc model for $sub$fifo.v:68$32 ($sub).
  creating $macc model for $add$fifo.v:66$29 ($add).
  creating $alu model for $macc $add$fifo.v:66$29.
  creating $alu model for $macc $sub$fifo.v:68$32.
  creating $alu model for $macc $flatten\fifo_reader.$add$fifo.v:19$36.
  creating $alu model for $macc $flatten\fifo_writer.$add$fifo.v:19$36.
  creating $alu cell for $flatten\fifo_writer.$add$fifo.v:19$36: $auto$alumacc.
↪cc:512:replace_alu$88
  creating $alu cell for $flatten\fifo_reader.$add$fifo.v:19$36: $auto$alumacc.
↪cc:512:replace_alu$91
  creating $alu cell for $sub$fifo.v:68$32: $auto$alumacc.cc:512:replace_alu$94
  creating $alu cell for $add$fifo.v:66$29: $auto$alumacc.cc:512:replace_alu$97
  created 4 $alu and 0 $macc cells.
```

Once these cells have been inserted, the call to *opt* can combine cells which are now identical but may have been missed due to e.g. the difference between *$add* and *$sub*.

The other new command in this part is *memory - translate memories to basic cells*. *memory* is another macro command which we examine in more detail in *Memory handling*. For this document, let us focus just on the step most relevant to our example: *memory_collect*. Up until this point, our memory reads and our memory writes have been totally disjoint cells; operating on the same memory only in the abstract. *memory_collect* combines all of the reads and writes for a memory block into a single cell.

Looking at the schematic after running *memory_collect* we see that our *$memrd_v2* cell has been replaced with a *$mem_v2* cell named `data`, the same name that we used in *fifo.v*. Where before we had a single set of signals for address and enable, we now have one set for reading (`RD_*`) and one for writing (`WR_*`), as well as both `WR_DATA` input and `RD_DATA` output.

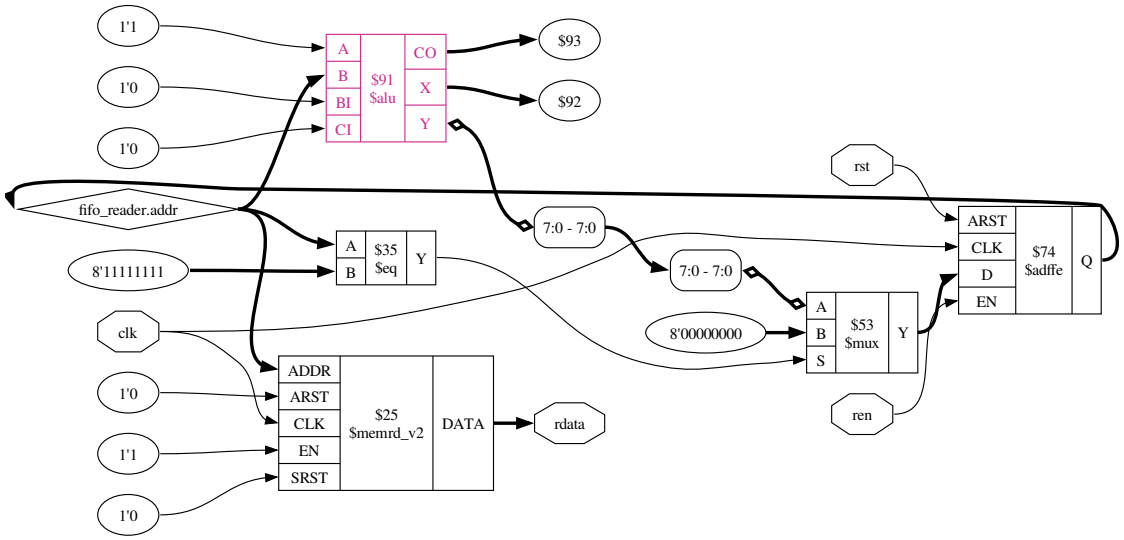


Fig. 2.9: `rdata` output after `alumacc`

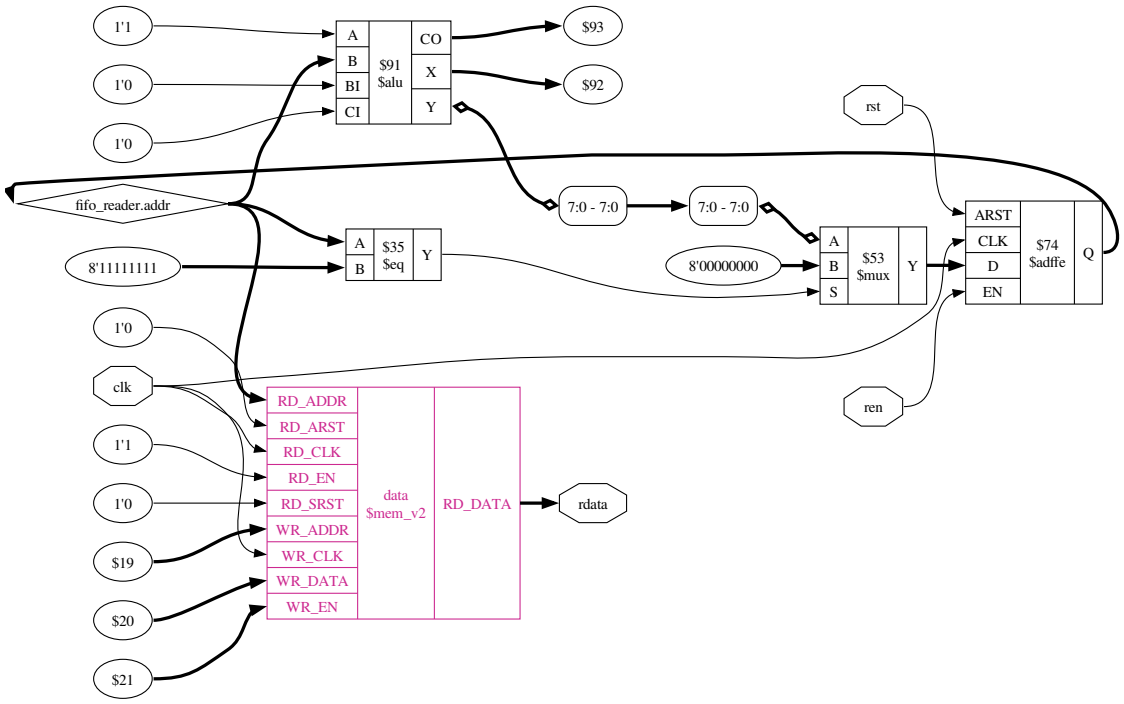


Fig. 2.10: `rdata` output after *`memory_collect`*

See also

Advanced usage docs for

- *Optimization passes*
- *Memory handling*

Final note

Having now reached the end of the the coarse-grain representation, we could also have gotten here by running `synth_ice40 -top fifo -run :map_ram` after loading the design. The `-run <from_label>:<to_label>` option with an empty `<from_label>` starts from the *begin section*, while the `<to_label>` runs up to but including the *map_ram section*.

2.2.6 Hardware mapping

The remaining sections each map a different type of hardware and are much more architecture dependent than the previous sections. As such we will only be looking at each section very briefly.

If you skipped calling `read_verilog -D ICE40_HX -lib -specify +/ice40/cells_sim.v` earlier, do it now.

Memory blocks

Mapping to hard memory blocks uses a combination of *`memory_libmap`* and `techmap`.

Listing 2.16: `map_ram` section

```
memory_libmap
techmap
ice40_braminit
```

The *map_ram section* converts the generic *`$mem_v2`* into the iCE40 `SB_RAM40_4K` (highlighted). We can also see the memory address has been remapped, and the data bits have been reordered (or swizzled). There is also now a *`$mux`* cell controlling the value of `rdata`. In *fifo.v* we wrote our memory as read-before-write, however the `SB_RAM40_4K` has undefined behaviour when reading from and writing to the same address in the same cycle. As a result, extra logic is added so that the generated circuit matches the behaviour of the verilog. *Synchronous SDP with undefined collision behavior* describes how we could change our verilog to match our hardware instead.

If we run *`memory_libmap`* under the *`debug`* command we can see candidates which were identified for mapping, along with the costs of each and what logic requires emulation.

```
yosys> debug memory_libmap -lib +/ice40/brams.txt -lib +/ice40/spram.txt -no-auto-huge
4. Executing MEMORY_LIBMAP pass (mapping memories to cells).
Memory fifo.data mapping candidates (post-geometry):
- logic fallback
  - cost: 2048.000000
- $__ICE40_RAM4K_:
  - option HAS_BE 0
  - emulation score: 7
  - replicates (for ports): 1
  - replicates (for data): 1
  - mux score: 0
```

(continues on next page)

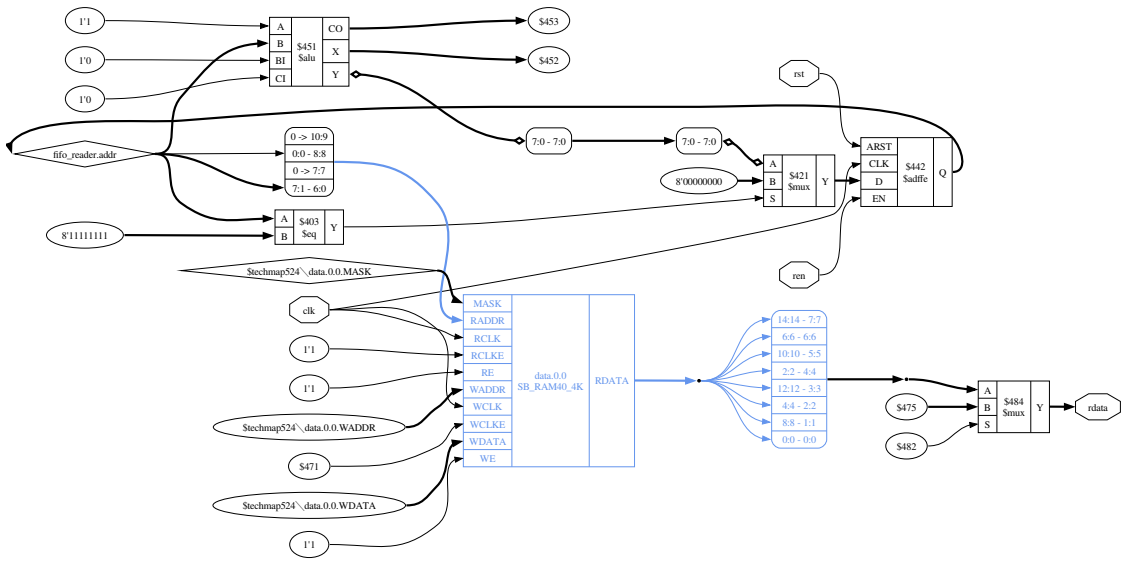


Fig. 2.11: `rdata` output after *map_ram section*

(continued from previous page)

```
  - demux score: 0
  - cost: 78.000000
  - abits 11 dbits 2 4 8 16
  - chosen base width 8
  - swizzle 0 1 2 3 4 5 6 7
  - emulate read-first behavior
  - write port 0: port group W
    - widths 2 4 8
  - read port 0: port group R
    - widths 2 4 8 16
    - emulate transparency with write port 0
- $__ICE40_RAM4K_:
  - option HAS_BE 1
  - emulation score: 7
  - replicates (for ports): 1
  - replicates (for data): 1
  - mux score: 0
  - demux score: 0
  - cost: 78.000000
  - abits 11 dbits 2 4 8 16
  - byte width 1
  - chosen base width 8
  - swizzle 0 1 2 3 4 5 6 7
  - emulate read-first behavior
  - write port 0: port group W
    - widths 16
  - read port 0: port group R
```

(continues on next page)

(continued from previous page)

```
    - widths 2 4 8 16
    - emulate transparency with write port 0
Memory fifo.data mapping candidates (after post-geometry prune):
- logic fallback
  - cost: 2048.000000
- $__ICE40_RAM4K_:
  - option HAS_BE 0
  - emulation score: 7
  - replicates (for ports): 1
  - replicates (for data): 1
  - mux score: 0
  - demux score: 0
  - cost: 78.000000
  - abits 11 dbits 2 4 8 16
  - chosen base width 8
  - swizzle 0 1 2 3 4 5 6 7
  - emulate read-first behavior
  - write port 0: port group W
    - widths 2 4 8
  - read port 0: port group R
    - widths 2 4 8 16
    - emulate transparency with write port 0
mapping memory fifo.data via $__ICE40_RAM4K_
```

The `$__ICE40_RAM4K_` cell is defined in the file techlibs/ice40/brams.txt, with the mapping to `SB_RAM40_4K` done by `techmap` using techlibs/ice40/brams_map.v. Any leftover memory cells are then converted into flip flops (the `logic fallback`) with *memory_map*.

Listing 2.17: `map_ffram` section

```
opt -fast -mux_undef -undriven -fine
memory_map
opt -undriven -fine
```

Note

The visual clutter on the `RDATA` output port (highlighted) is an unfortunate side effect of *opt_clean* on the swizzled data bits. In connecting the *$mux* input port directly to `RDATA` to reduce the number of wires, the `$techmap579\data.0.0.RDATA` wire becomes more visually complex.

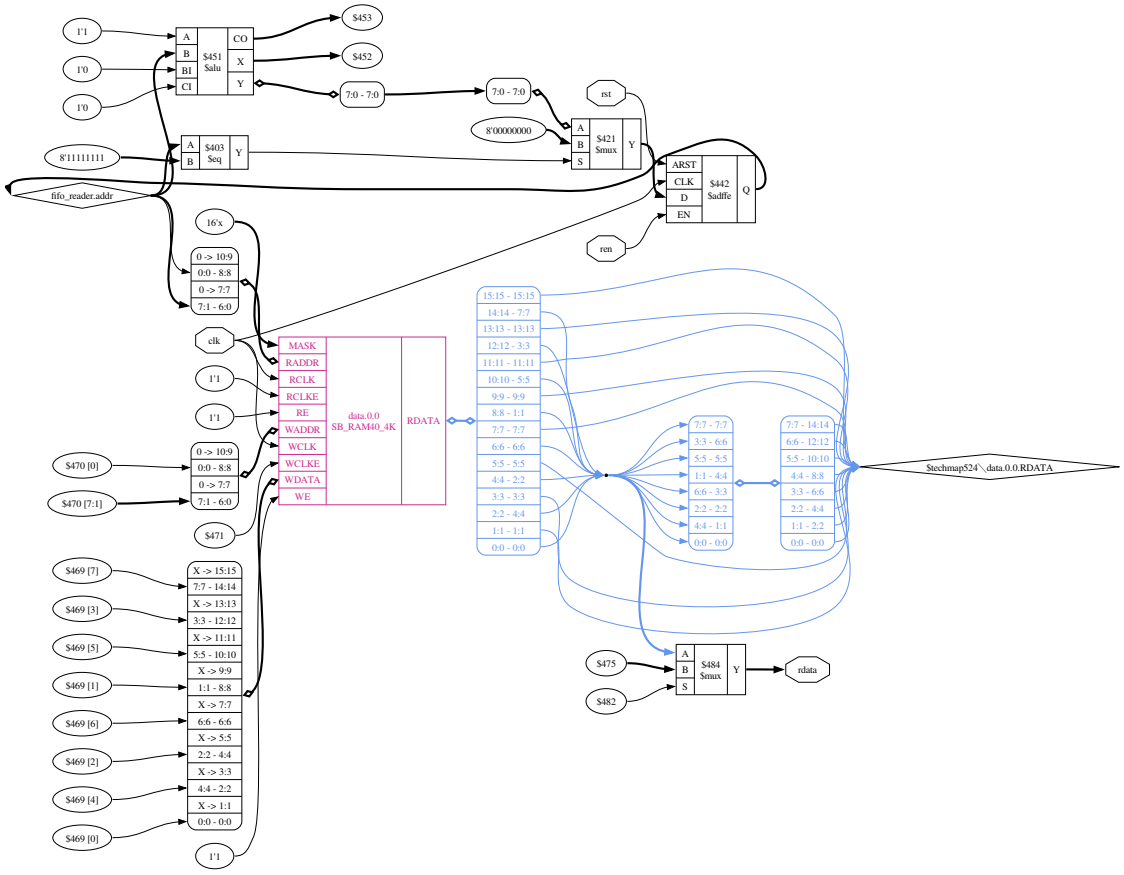


Fig. 2.12: `rdata` output after *map_ffram section*

See also

Advanced usage docs for

- *Technology mapping*
- *Memory handling*

Arithmetic

Uses `techmap` to map basic arithmetic logic to hardware. This sees somewhat of an explosion in cells as multi-bit *`$mux`* and *`$adffe`* are replaced with single-bit *`$_MUX_`* and *`$_DFFE_PP0P_`* cells, while the *`$alu`* is replaced with primitive *`$_OR_`* and *`$_NOT_`* gates and a *`$lut`* cell.

Listing 2.18: `map_gates` section

```
ice40_wrapcarry
techmap
opt -fast
abc -dff -D 1
ice40_opt
```

See also

Advanced usage docs for *Technology mapping*

Flip-flops

Convert FFs to the types supported in hardware with `dfflegalize`, and then use `techmap` to map them. In our example, this converts the *`$_DFFE_PP0P_`* cells to `SB_DFFER`.

We also run `simplemap` here to convert any remaining cells which could not be mapped to hardware into gate-level primitives. This includes optimizing *`$_MUX_`* cells where one of the inputs is a constant `1'0`, replacing it instead with an *`$_AND_`* cell.

Listing 2.19: `map_ffs` section

```
dfflegalize
techmap
opt_expr -mux_undef
simplemap
ice40_opt -full
```

See also

Advanced usage docs for *Technology mapping*

LUTs

`abc` and `techmap` are used to map LUTs; converting primitive cell types to use *`$lut`* and `SB_CARRY` cells. Note that the iCE40 flow uses `abc9` rather than `abc`. For more on what these do, and what the difference between these two commands are, refer to *The ABC toolbox*.

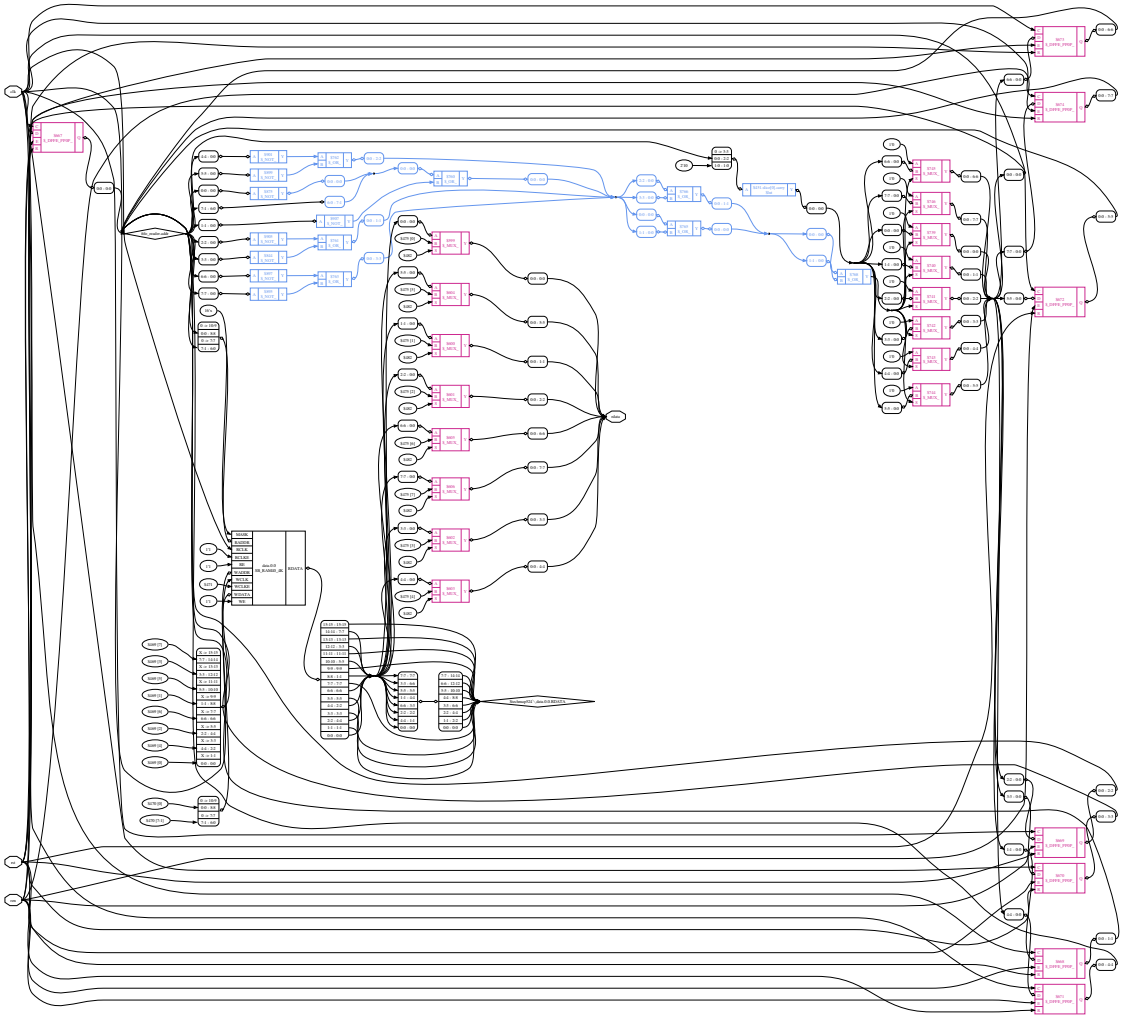


Fig. 2.13: `rdata` output after *map_gates section*

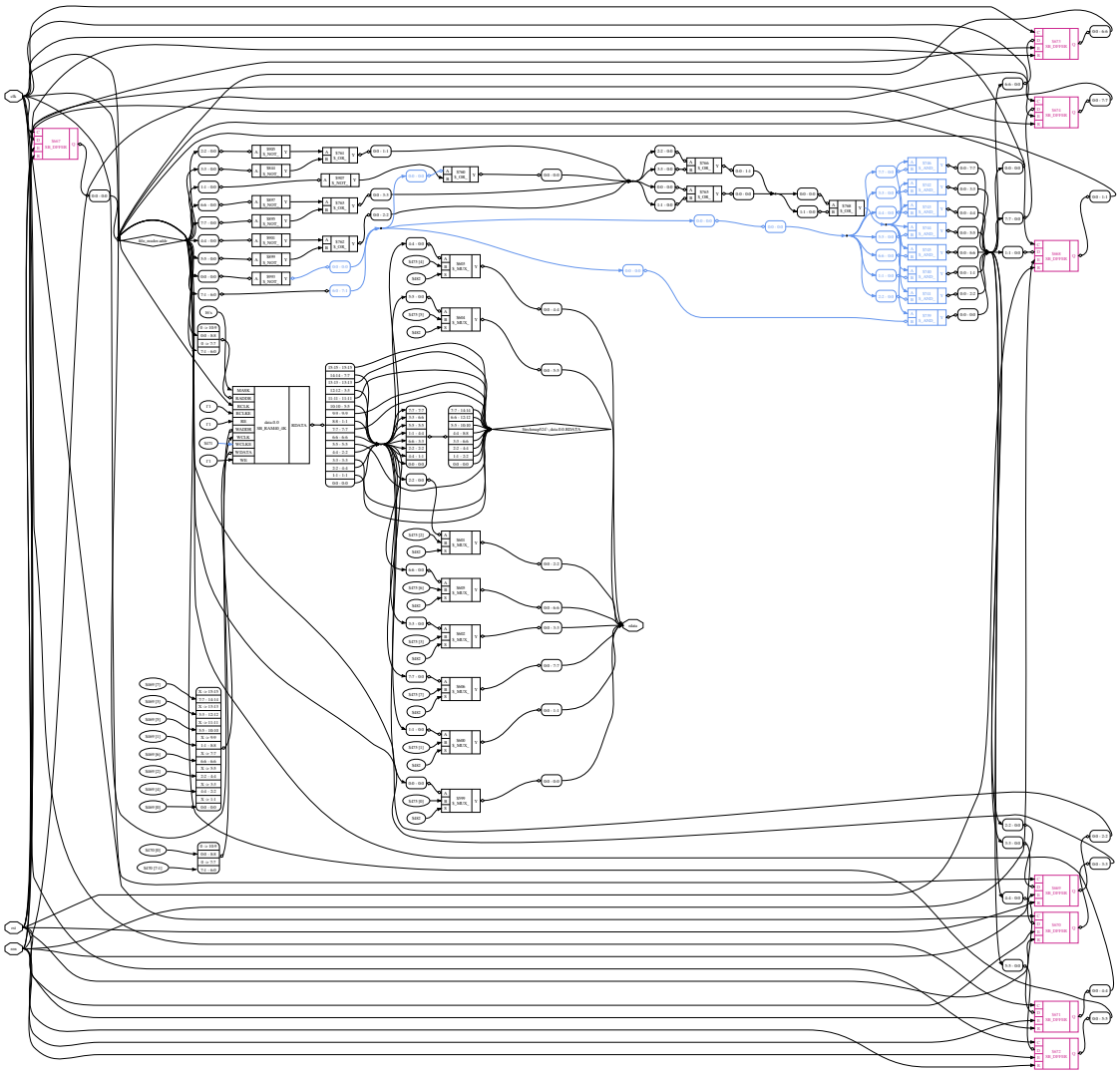


Fig. 2.14: `rdata` output after *map_ffs section*

Listing 2.20: `map_luts` section

```
abc
ice40_opt
techmap
simplemap
techmap
flowmap
read_verilog
abc9
ice40_wrapcarry -unwrap
techmap
clean
opt_lut -tech ice40
```

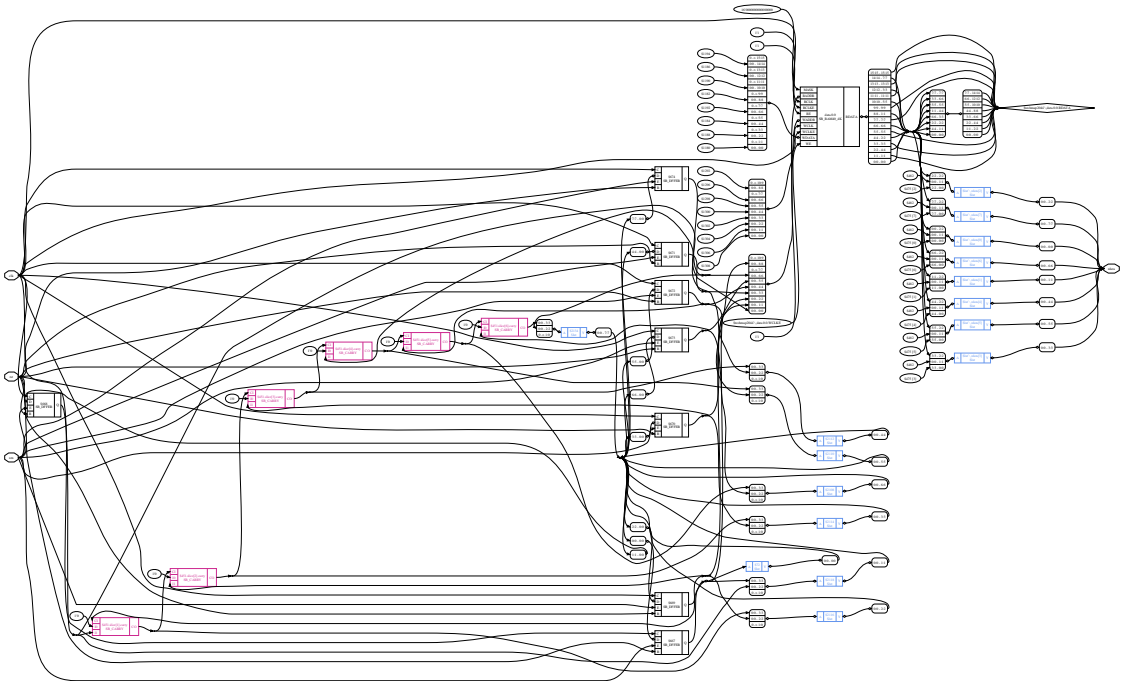


Fig. 2.15: `rdata` output after *map_luts section*

Finally we use `techmap` to map the generic *`$lut`* cells to iCE40 `SB_LUT4` cells.

Listing 2.21: `map_cells` section

```
techmap
clean
```

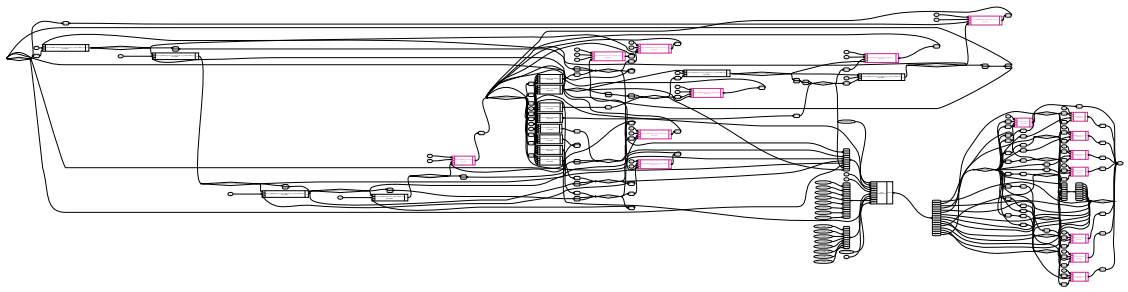


Fig. 2.16: `rdata` output after *map_cells section*

See also

Advanced usage docs for

- *Technology mapping*
- *The ABC toolbox*

Other cells

The following commands may also be used for mapping other cells:

`hilomap`
: Some architectures require special driver cells for driving a constant hi or lo value. This command replaces simple constants with instances of such driver cells.

`iopadmap`
: Top-level input/outputs must usually be implemented using special I/O-pad cells. This command inserts such cells to the design.

These commands tend to either be in the *map_cells section* or after the *check section* depending on the flow.

2.2.7 Final steps

The next section of the iCE40 synth flow performs some sanity checking and final tidy up:

Listing 2.22: `check` section

```
autoname
hierarchy -check
stat
check -noinit
blackbox =A:whitebox
```

The new commands here are:

- `autoname`,
- *stat - print some statistics*, and
- `blackbox`.

The output from *stat* is useful for checking resource utilization; providing a list of cells used in the design and the number of each, as well as the number of other resources used such as wires and processes. For this design, the final call to *stat* should look something like the following:

```
yosys> stat -top fifo

17. Printing statistics.

=== fifo ===

        +----------Local Count, excluding submodules.
        |
       95 wires
      264 wire bits
       95 public wires
      264 public wire bits
        7 ports
       29 port bits
      138 cells
        2   $scopeinfo
       26   SB_CARRY
       26   SB_DFF
       25   SB_DFFER
       58   SB_LUT4
        1   SB_RAM40_4K
```

Note that the `-top fifo` here is optional. *stat* will automatically use the module with the `top` attribute set, which `fifo` was when we called `hierarchy`. If no module is marked `top`, then stats will be shown for each module selected.

The *stat* output is also useful as a kind of sanity-check: Since we have already run *proc*, we wouldn't expect there to be any processes. We also expect `data` to use hard memory; if instead of an `SB_RAM40_4K` saw a high number of flip-flops being used we might suspect something was wrong.

If we instead called *stat* immediately after `read_verilog fifo.v` we would see something very different:

```
yosys> stat

2. Printing statistics.

=== fifo ===

        +----------Local Count, excluding submodules.
        |
       28 wires
      219 wire bits
        9 public wires
       45 public wire bits
        7 ports
       29 port bits
        1 memories
     2048 memory bits
        3 processes
        7 cells
        1   $add
```

(continues on next page)

(continued from previous page)

```
        2   $logic_and
        2   $logic_not
        1   $memrd
        1   $sub
        2 submodules
        2   addr_gen

=== addr_gen ===

        +----------Local Count, excluding submodules.
        |
        8 wires
       60 wire bits
        4 public wires
       11 public wire bits
        4 ports
       11 port bits
        2 processes
        2 cells
        1   $add
        1   $eq
```

Notice how `fifo` and `addr_gen` are listed separately, and the statistics for `fifo` show 2 `addr_gen` modules. Because this is before the memory has been mapped, we also see that there is 1 memory with 2048 memory bits; matching our 8-bit wide `data` memory with 256 values ($8 * 256 = 2048$).

Synthesis output

The iCE40 synthesis flow has the following output modes available:

- *write_blif*,
- *write_edif*, and
- *write_json*.

As an example, if we called `synth_ice40 -top fifo -json fifo.json`, our synthesized `fifo` design will be output as `fifo.json`. We can then read the design back into Yosys with *read_json*, but make sure you use `design -reset` or open a new interactive terminal first. The JSON output we get can also be loaded into nextpnr to do place and route; but that is beyond the scope of this documentation.

See also

`synth_ice40`

2.3 Scripting in Yosys

On the previous page we went through a synthesis script, running each command in the interactive Yosys shell. On this page, we will be introducing the script file format and how you can make your own synthesis scripts.

Yosys script files typically use the `.ys` extension and contain a set of commands for Yosys to run sequentially. These commands are the same ones we were using on the previous page like *read_verilog* and `hierarchy`.

2.3.1 Script parsing

As with the interactive shell, each command consists of the command name, and an optional whitespace separated list of arguments. Commands are terminated with the newline character, and anything after a hash sign `#` is a comment (i.e. it is ignored).

It is also possible to terminate commands with a semicolon `;`. This is particularly useful in conjunction with the `-p <command>` command line option, where `<command>` can be a string with multiple commands separated by semicolon. In-line comments can also be made with the colon `:`, where the end of the comment is a semicolon `;` or a new line.

Listing 2.23: Using the `-p` option

```
$ yosys -p 'read_verilog fifo.v; :this is a comment; prep'
```

> **Warning**
>
> The space after the semicolon is required for correct parsing. `log a;log b;` for example will display `a;log b` instead of `a` and `b` as might be expected.

Another special character that can be used in Yosys scripts is the bang `!`. Anything after the bang will be executed as a shell command. This can only be terminated with a new line. Any semicolons, hashes, or other special characters will be passed to the shell. If an error code is returned from the shell it will be raised by Yosys. *`exec`* provides a much more flexible way of executing commands, allowing the output to be logged and more control over when to generate errors.

> **Warning**
>
> Take care when using the `yosys -p` option. Some shells such as bash will perform substitution options inside of a double quoted string, such as `!` for history substitution and `$` for variable substitution; single quotes should be used instead to pass the string to Yosys without substitution.

2.3.2 The synthesis starter script

All of the images and console output used in *Synthesis starter* were generated by Yosys, using Yosys script files found in `docs/source/code_examples/fifo`. If you haven't already, let's take a look at some of those script files now.

Listing 2.24: A section of `fifo.ys`, generating the images used for *The addr_gen module*

```
10 echo on
11 hierarchy -top addr_gen
12 select -module addr_gen
13 select -list
14 select t:*
15 select -list
16 select -set new_cells %
17 select -clear
18 show -format dot -prefix addr_gen_show addr_gen
19 show -format dot -prefix new_cells_show -notitle @new_cells
20 show -color maroon3 @new_cells -color cornflowerblue p:* -notitle -format dot -prefix␣
```

(continues on next page)

(continued from previous page)

```
↪addr_gen_hier

# ======================================================
proc -noopt
select -set new_cells t:$mux t:*dff
show -color maroon3 @new_cells -notitle -format dot -prefix addr_gen_proc

# ======================================================
opt_expr; clean
select -set new_cells t:$eq
show -color cornflowerblue @new_cells -notitle -format dot -prefix addr_gen_clean

# ======================================================
```

The first command there, `echo on`, uses `echo` to enable command echoes on. This is how we generated the code listing for *hierarchy -top addr_gen output*. Turning command echoes on prints the `yosys> hierarchy -top addr_gen` line, making the output look the same as if it were an interactive terminal. `hierarchy -top addr_gen` is of course the command we were demonstrating, including the output text and an image of the design schematic after running it.

We briefly touched on *select* when it came up in `synth_ice40`, but let's look at it more now.

Selections intro

The *select* command is used to modify and view the list of selected objects:

```
yosys> select -module addr_gen

yosys [addr_gen]> select -list
addr_gen
addr_gen/$add$fifo.v:19$3
addr_gen/$eq$fifo.v:16$2
addr_gen/$1\addr[7:0]
addr_gen/$add$fifo.v:19$3_Y
addr_gen/$eq$fifo.v:16$2_Y
addr_gen/$0\addr[7:0]
addr_gen/addr
addr_gen/rst
addr_gen/clk
addr_gen/en
addr_gen/$proc$fifo.v:8$4
addr_gen/$proc$fifo.v:12$1

yosys [addr_gen]> select t:*

yosys [addr_gen]*> select -list
addr_gen/$add$fifo.v:19$3
addr_gen/$eq$fifo.v:16$2

yosys [addr_gen]*> select -set new_cells %

yosys [addr_gen]*> select -clear
```

When we call `select -module addr_gen` we are changing the currently active selection from the whole

design, to just the `addr_gen` module. Notice how this changes the `yosys` at the start of each command to `yosys [addr_gen]`? This indicates that any commands we run at this point will *only* operate on the `addr_gen` module. When we then call `select -list` we get a list of all objects in the `addr_gen` module, including the module itself, as well as all of the wires, inputs, outputs, processes, and cells.

Next we perform another selection, `select t:*`. The `t:` part signifies we are matching on the *cell type*, and the `*` means to match anything. For this (very simple) selection, we are trying to find all of the cells, regardless of their type. The active selection is now shown as `[addr_gen]*`, indicating some sub-selection of the `addr_gen` module. This gives us the *`$add`* and *`$eq`* cells, which we want to highlight for the *addr_gen module after hierarchy* image.

We can assign a name to a selection with `select -set`. In our case we are using the name `new_cells`, and telling it to use the current selection, indicated by the `%` symbol. We can then use this named selection by referring to it as `@new_cells`, which we will see later. Then we clear the selection so that the following commands can operate on the full design. While we split that out for this document, we could have done the same thing in a single line by calling `select -set new_cells addr_gen/t:*`. If we know we only have the one module in our design, we can even skip the `addr_gen/` part. Looking further down *the fifo.ys code* we can see this with `select -set new_cells t:$mux t:*dff`. We can also see in that command that selections don't have to be limited to a single statement.

Many commands also support an optional `[selection]` argument which can be used to override the currently selected objects. We could, for example, call `clean addr_gen` to have *`clean`* operate on *just* the `addr_gen` module.

Detailed documentation of the select framework can be found under *Selections* or in the command reference at *select - modify and view the list of selected objects*.

Displaying schematics

While the *`select`* command is very useful, sometimes nothing beats being able to see a design for yourself. This is where *`show`* comes in. Note that this document is just an introduction to the *`show`* command, only covering the basics. For more information, including a guide on what the different symbols represent, see *A look at the show command* and the *Interactive design investigation* page.

> **Note**
>
> The *`show`* command requires a working installation of GraphViz and xdot for displaying the actual circuit diagrams.

This is the first `show` command we called in `fifo.ys`, *as we saw above*. If we look at the log output for this image we see the following:

```
yosys> show -format dot -prefix addr_gen_show addr_gen

4. Generating Graphviz representation of design.
Writing dot description to `addr_gen_show.dot'.
Dumping module addr_gen to page 1.
```

Calling *`show`* with `-format dot` tells it we want to output a `.dot` file rather than opening it for display. The `-prefix addr_gen_show` option indicates we want the file to be called `addr_gen_show.*`. Remember, we do this in `fifo.ys` because we need to store the image for displaying in the documentation you're reading. But if you just want to display the images locally you can skip these two options. The `-format` option internally calls the `dot` command line program from GraphViz to convert to formats other than `.dot`. Check GraphViz output docs for more on available formats.

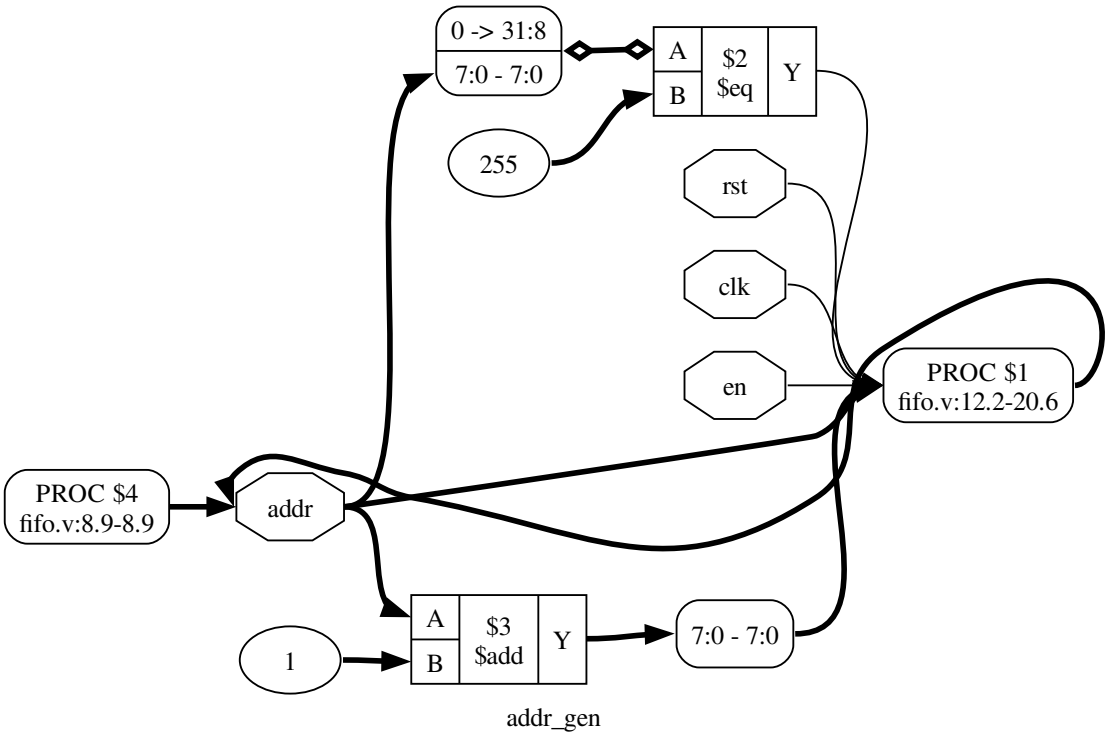


Fig. 2.17: Calling `show addr_gen` after `hierarchy`

Note

If you are using a POSIX based version of Yosys (such as for Mac or Linux), xdot will be opened in the background and Yosys can continue to be used. If it it still open, future calls to `show` will use the same xdot instance.

The `addr_gen` at the end tells it we only want the `addr_gen` module, just like when we called `select -module addr_gen` in *Selections intro*. That last parameter doesn't have to be a module name, it can be any valid selection string. Remember when we *assigned a name to a selection* and called it `new_cells`? We saw in the `select -list` output that it contained two cells, an *`$add`* and an *`$eq`*. We can call *`show`* on that selection just as easily:

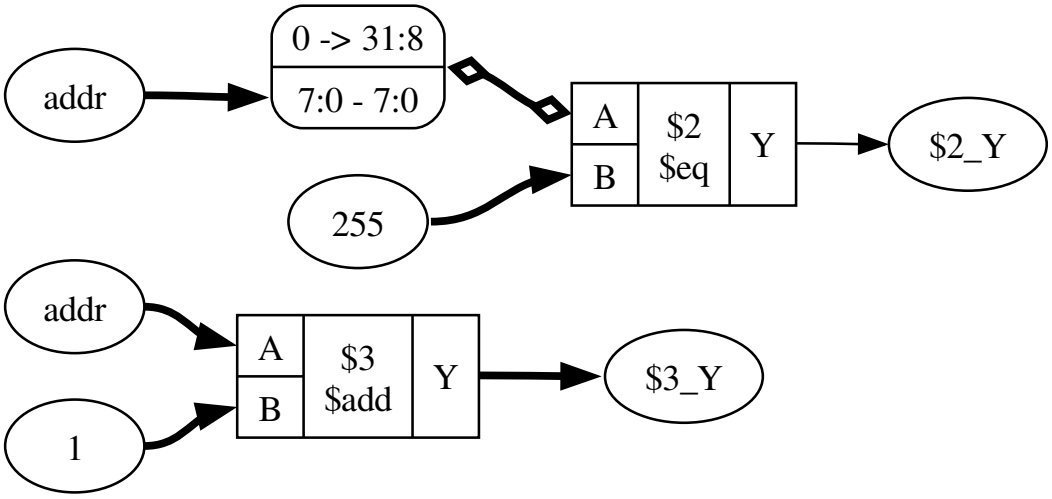


Fig. 2.18: Calling `show -notitle @new_cells`

We could have gotten the same output with `show -notitle t:$add t:$eq` if we didn't have the named selection. By adding the `-notitle` flag there we can also get rid of the `addr_gen` title that would have been automatically added. The last two images were both added for this introduction. The next image is the first one we saw in *Synthesis starter*: showing the full `addr_gen` module while also highlighting `@new_cells` and the two `PROC` blocks. To achieve this highlight, we make use of the `-color` option:

As described in the the `help` output for *`show`* (or by clicking on the *`show`* link), colors are specified as `-color <color> <object>`. Color names for the `<color>` portion can be found on the GraphViz color docs. Unlike the final *`show`* parameter which can have be any selection string, the `<object>` part must be a single selection expression or named selection. That means while we can use `@new_cells`, we couldn't use `t:$eq t:$add`. In general, if a command lists `[selection]` as its final parameter it can be any selection string. Any selections that are not the final parameter, such as those used in options, must be a single expression instead.

For all of the options available to *`show`*, check the command reference at *show - generate schematics using graphviz*.

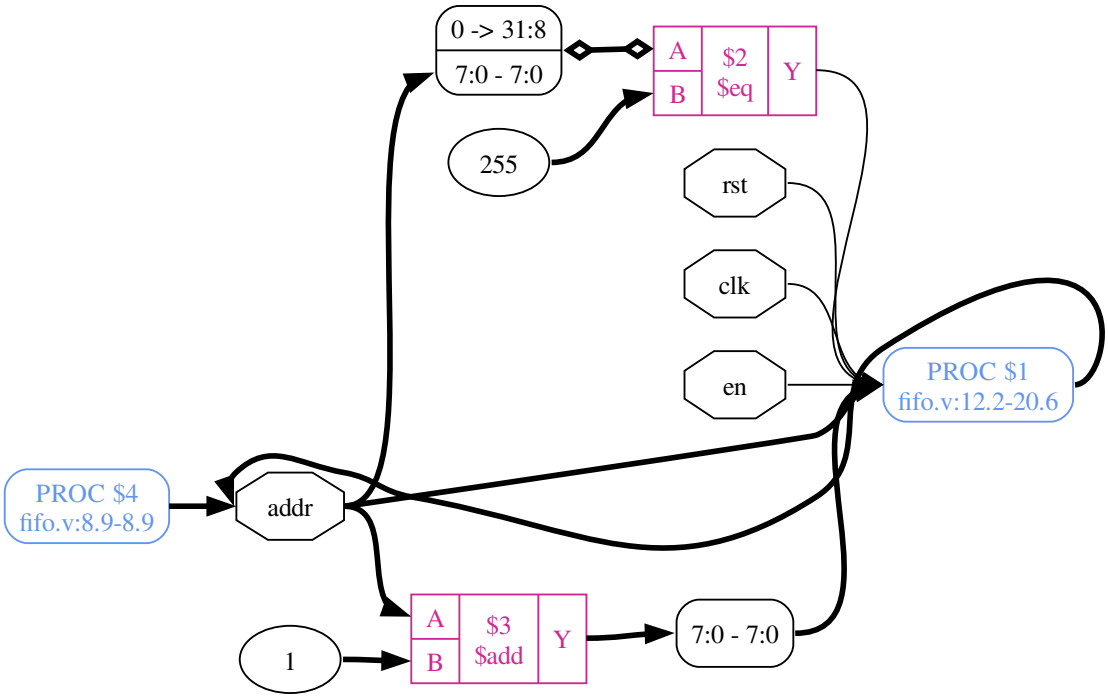


Fig. 2.19: Calling `show -color maroon3 @new_cells -color cornflowerblue p:* -notitle`

See also

A look at the show command on the *Interactive design investigation* page.

CHAPTER

THREE

USING YOSYS (ADVANCED)

While much of Yosys is focused around synthesis, there are also a number of other useful things that can be accomplished with Yosys scripts or in an interactive shell. As such this section is broken into two parts: *Synthesis in detail* expands on the *Synthesis starter* and goes into further detail on the major commands used in synthesis; *More scripting* covers the ways Yosys can interact with designs for a deeper investigation.

3.1 Synthesis in detail

Synthesis can generally be broken down into coarse-grain synthesis, and fine-grain synthesis. We saw this in *Synthesis starter* where a design was loaded and elaborated and then went through a series of coarse-grain optimizations before being mapped to hard blocks and fine-grain cells. Most commands in Yosys will target either coarse-grain representation or fine-grain representation, with only a select few compatible with both states.

Commands such as *proc*, *fsm*, and *memory* rely on the additional information in the coarse-grain representation, along with a number of optimizations such as `wreduce`, `share`, and `alumacc`. *opt* provides optimizations which are useful in both states, while `techmap` is used to convert coarse-grain cells to the corresponding fine-grain representation.

Single-bit cells (logic gates, FFs) as well as LUTs, half-adders, and full-adders make up the bulk of the fine-grain representation and are necessary for commands such as `abc`/`abc9`, `simplemap`, `dfflegalize`, and *memory_map*.

3.1.1 Synth commands

Todo

comment on common `synth_*` options, like `-run`

Packaged `synth_*` commands

A list of all synth commands included in Yosys for different platforms can be found under *Technology libraries*. Each command runs a script of sub commands specific to the platform being targeted. Note that not all of these scripts are actively maintained and may not be up-to-date.

General synthesis

In addition to the above hardware-specific synth commands, there is also `prep`. This command is limited to coarse-grain synthesis, without getting into any architecture-specific mappings or optimizations. Among other things, this is useful for design verification.

The following commands are executed by the `prep` command:

```
begin:
    hierarchy -check [-top <top> | -auto-top]

coarse:
    proc [-ifx]
    flatten    (if -flatten)
    future
    opt_expr -keepdc
    opt_clean
    check
    opt -noff -keepdc
    wreduce -keepdc [-memx]
    memory_dff    (if -rdff)
    memory_memx    (if -memx)
    opt_clean
    memory_collect
    opt -noff -keepdc -fast
    sort

check:
    stat
    check
```

Synthesis starter covers most of these commands and what they do.

3.1.2 Converting process blocks

The Verilog frontend converts `always`-blocks to RTL netlists for the expressions and "processess" for the control- and memory elements. The *proc* command then transforms these "processess" to netlists of RTL multiplexer and register cells. It also is a macro command that calls the other `proc_*` commands in a sensible order:

Listing 3.1: Passes called by *proc*

```
proc_clean # removes empty branches and processes
proc_rmdead # removes unreachable branches
proc_prune
proc_init # special handling of "initial" blocks
proc_arst # identifies modeling of async resets
proc_rom
proc_mux # converts decision trees to multiplexer networks
proc_dlatch
proc_dff # extracts registers from processes
proc_memwr
proc_clean # this should remove all the processes, provided all went fine
opt_expr -keepdc
```

After all the `proc_*` commands, *opt_expr* is called. This can be disabled by calling `proc -noopt`. For more information about *proc*, such as disabling certain sub commands, see *Converting process blocks*.

Many commands can not operate on modules with "processess" in them. Usually a call to *proc* is the first command in the actual synthesis procedure after design elaboration.

Example

Todo

describe `proc` images

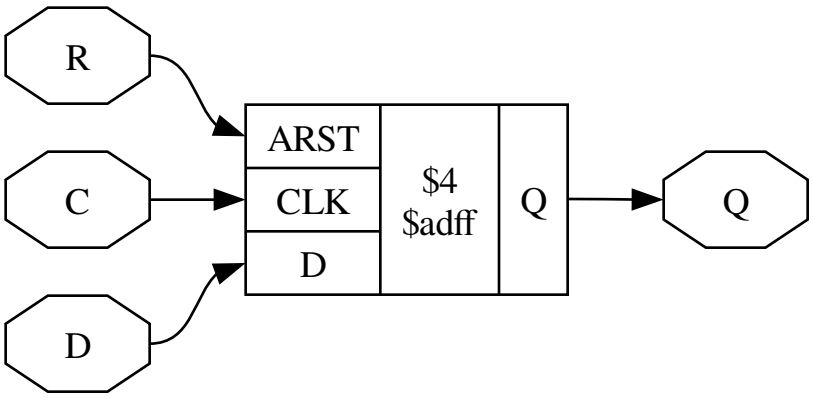
`docs/source/code_examples/synth_flow`.

Listing 3.2: `proc_01.v`

```
module test(input D, C, R, output reg Q);
    always @(posedge C, posedge R)
        if (R)
            Q <= 0;
        else
            Q <= D;
endmodule
```

Listing 3.3: `proc_01.ys`

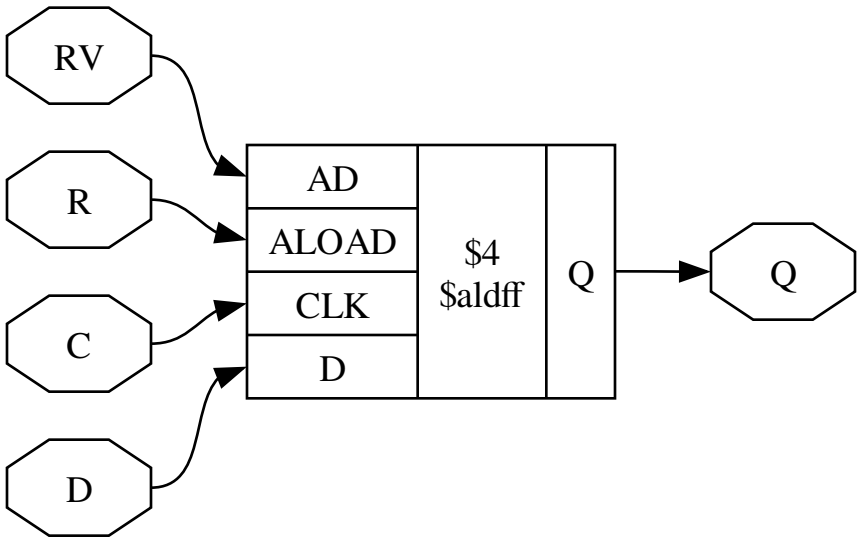
```
read_verilog proc_01.v
hierarchy -check -top test
proc;;
```



Listing 3.4: `proc_02.v`

```
module test(input D, C, R, RV,
            output reg Q);
    always @(posedge C, posedge R)
        if (R)
            Q <= RV;
        else
```

(continues on next page)



(continued from previous page)

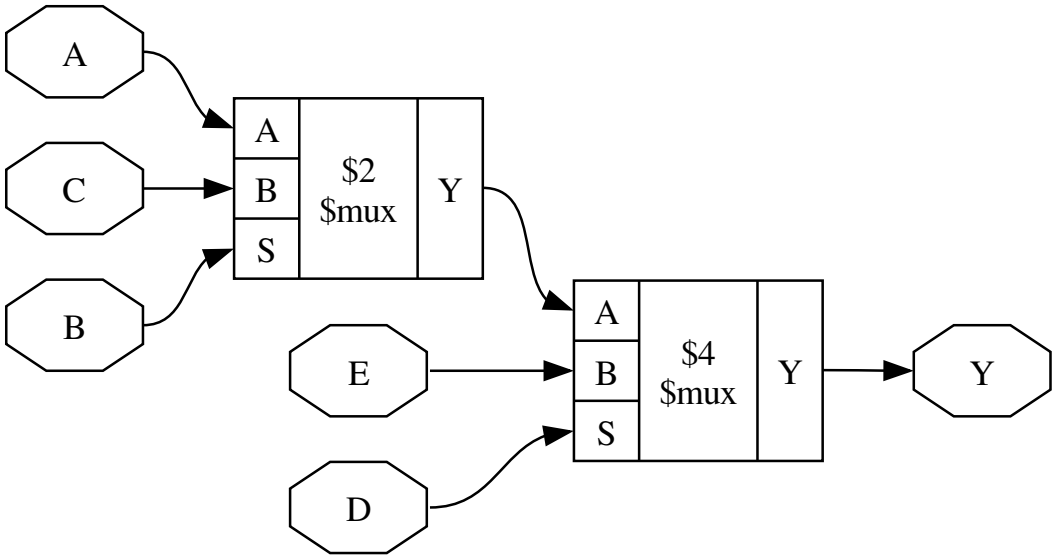
```
            Q <= D;
endmodule
```

Listing 3.5: `proc_02.ys`

```
read_verilog proc_02.v
hierarchy -check -top test
proc;;
```

Listing 3.6: `proc_03.ys`

```
read_verilog proc_03.v
hierarchy -check -top test
proc;;
```



Listing 3.7: `proc_03.v`

```
module test(input A, B, C, D, E,
            output reg Y);
    always @* begin
        Y <= A;
        if (B)
            Y <= C;
        if (D)
            Y <= E;
    end
endmodule
```

3.1.3 FSM handling

The *fsm* command identifies, extracts, optimizes (re-encodes), and re-synthesizes finite state machines. It again is a macro that calls a series of other commands:

Listing 3.8: Passes called by *fsm*

```
# Identify and extract FSMs:
fsm_detect
fsm_extract

# Basic optimizations:
fsm_opt
opt_clean
```

(continues on next page)

(continued from previous page)

```
fsm_opt

# Expanding to nearby gate-logic (if called with -expand):
fsm_expand
opt_clean
fsm_opt

# Re-code FSM states (unless called with -norecode):
fsm_recode

# Print information about FSMs:
fsm_info

# Export FSMs in KISS2 file format (if called with -export):
fsm_export

# Map FSMs to RTL cells (unless called with -nomap):
fsm_map
```

See also *FSM handling*.

The algorithms used for FSM detection and extraction are influenced by a more general reported technique [STGR10].

FSM detection

The *fsm_detect* pass identifies FSM state registers. It sets the `fsm_encoding = "auto"` attribute on any (multi-bit) wire that matches the following description:

- Does not already have the `fsm_encoding` attribute.
- Is not an output of the containing module.
- Is driven by single *$dff* or *$adff* cell.
- The `D`-Input of this *$dff* or *$adff* cell is driven by a multiplexer tree that only has constants or the old state value on its leaves.
- The state value is only used in the said multiplexer tree or by simple relational cells that compare the state value to a constant (usually *$eq* cells).

This heuristic has proven to work very well. It is possible to overwrite it by setting `fsm_encoding = "auto"` on registers that should be considered FSM state registers and setting `fsm_encoding = "none"` on registers that match the above criteria but should not be considered FSM state registers.

Note however that marking state registers with `fsm_encoding` that are not suitable for FSM recoding can cause synthesis to fail or produce invalid results.

FSM extraction

The *fsm_extract* pass operates on all state signals marked with the (`fsm_encoding != "none"`) attribute. For each state signal the following information is determined:

- The state registers
- The asynchronous reset state if the state registers use asynchronous reset
- All states and the control input signals used in the state transition functions

- The control output signals calculated from the state signals and control inputs
- A table of all state transitions and corresponding control inputs- and outputs

The state registers (and asynchronous reset state, if applicable) is simply determined by identifying the driver for the state signal.

From there the *$mux*-tree driving the state register inputs is recursively traversed. All select inputs are control signals and the leaves of the *$mux*-tree are the states. The algorithm fails if a non-constant leaf that is not the state signal itself is found.

The list of control outputs is initialized with the bits from the state signal. It is then extended by adding all values that are calculated by cells that compare the state signal with a constant value.

In most cases this will cover all uses of the state register, thus rendering the state encoding arbitrary. If however a design uses e.g. a single bit of the state value to drive a control output directly, this bit of the state signal will be transformed to a control output of the same value.

Finally, a transition table for the FSM is generated. This is done by using the ConstEval C++ helper class (defined in kernel/consteval.h) that can be used to evaluate parts of the design. The ConstEval class can be asked to calculate a given set of result signals using a set of signal-value assignments. It can also be passed a list of stop-signals that abort the ConstEval algorithm if the value of a stop-signal is needed in order to calculate the result signals.

The *fsm_extract* pass uses the ConstEval class in the following way to create a transition table. For each state:

1. Create a ConstEval object for the module containing the FSM
2. Add all control inputs to the list of stop signals
3. Set the state signal to the current state
4. Try to evaluate the next state and control output
5. If step 4 was not successful:
   - Recursively goto step 4 with the offending stop-signal set to 0.
   - Recursively goto step 4 with the offending stop-signal set to 1.
6. If step 4 was successful: Emit transition

Finally a *$fsm* cell is created with the generated transition table and added to the module. This new cell is connected to the control signals and the old drivers for the control outputs are disconnected.

FSM optimization

The *fsm_opt* pass performs basic optimizations on *$fsm* cells (not including state recoding). The following optimizations are performed (in this order):

- Unused control outputs are removed from the *$fsm* cell. The attribute `unused_bits` (that is usually set by the *opt_clean* pass) is used to determine which control outputs are unused.
- Control inputs that are connected to the same driver are merged.
- When a control input is driven by a control output, the control input is removed and the transition table altered to give the same performance without the external feedback path.
- Entries in the transition table that yield the same output and only differ in the value of a single control input bit are merged and the different bit is removed from the sensitivity list (turned into a don't-care bit).
- Constant inputs are removed and the transition table is altered to give an unchanged behaviour.

- Unused inputs are removed.

FSM recoding

The *fsm_recode* pass assigns new bit pattern to the states. Usually this also implies a change in the width of the state signal. At the moment of this writing only one-hot encoding with all-zero for the reset state is supported.

The *fsm_recode* pass can also write a text file with the changes performed by it that can be used when verifying designs synthesized by Yosys using Synopsys Formality.

3.1.4 Memory handling

The *memory* command

In the RTL netlist, memory reads and writes are individual cells. This makes consolidating the number of ports for a memory easier. The *memory* pass transforms memories to an implementation. Per default that is logic for address decoders and registers. It also is a macro command that calls the other common `memory_*` passes in a sensible order:

Listing 3.9: Passes called by *memory*

```
opt_mem
opt_mem_priority
opt_mem_feedback
memory_bmux2rom
memory_dff
opt_clean
memory_share
opt_mem_widen
memory_memx                          (when called with -memx)
opt_clean
memory_collect
memory_bram -rules <bram_rules>      (when called with -bram)
memory_map                           (skipped if called with -nomap)
```

Todo

Make `memory_*` notes less quick

Some quick notes:

- *memory_dff* merges registers into the memory read- and write cells.
- *memory_collect* collects all read and write cells for a memory and transforms them into one multi-port memory cell.
- *memory_map* takes the multi-port memory cell and transforms it to address decoder logic and registers.

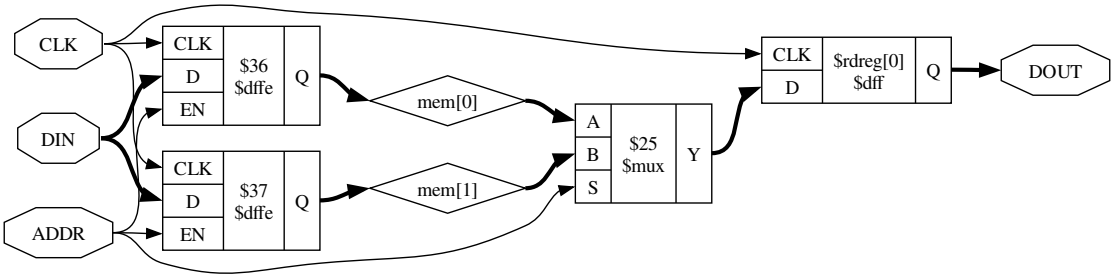
For more information about *memory*, such as disabling certain sub commands, see *Memory handling*.

Example

Todo

describe `memory` images

`docs/source/code_examples/synth_flow`.



Listing 3.10: `memory_01.ys`

```
read_verilog memory_01.v
hierarchy -check -top test
proc;; memory; opt
```

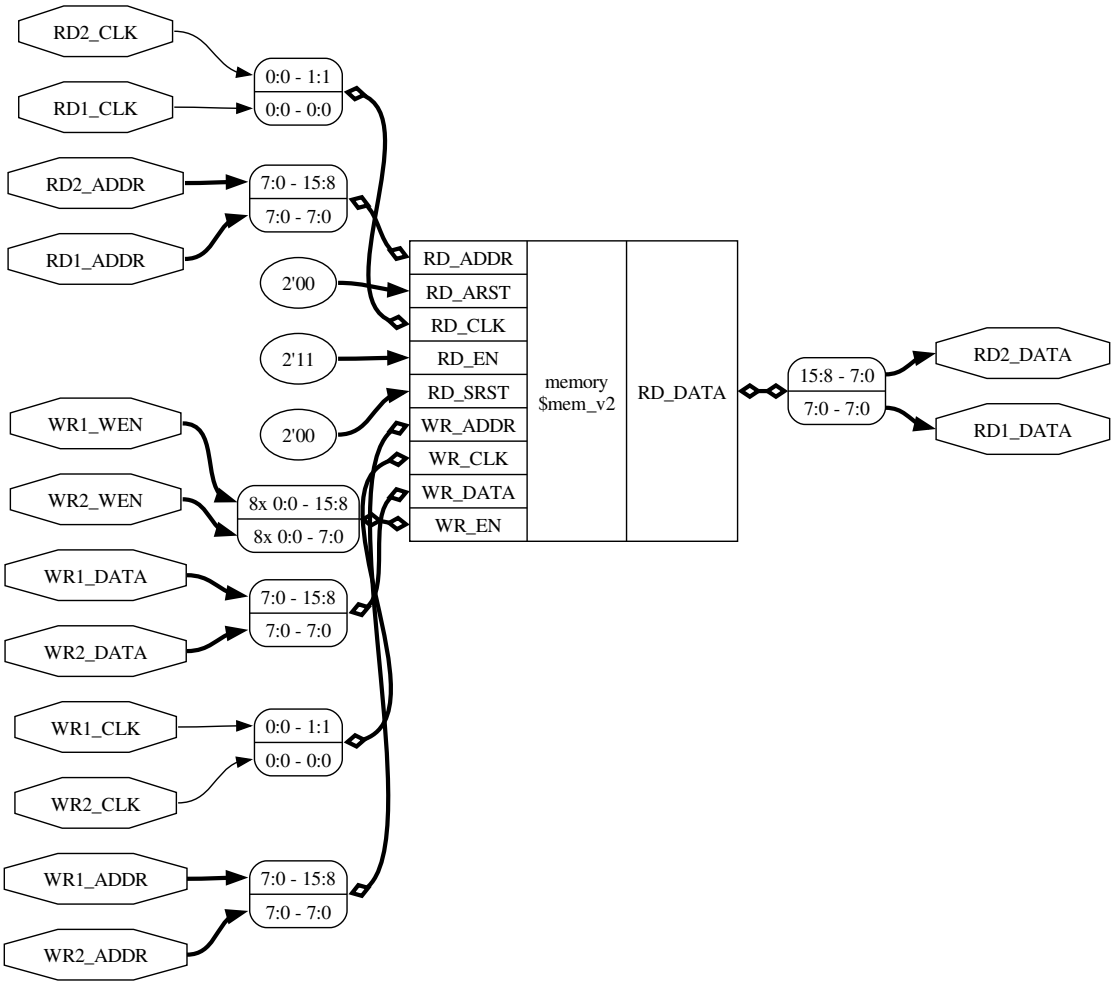
Listing 3.11: `memory_01.v`

```
module test(input      CLK, ADDR,
            input      [7:0] DIN,
            output reg [7:0] DOUT);
    reg [7:0] mem [0:1];
    always @(posedge CLK) begin
        mem[ADDR] <= DIN;
        DOUT <= mem[ADDR];
    end
endmodule
```

Listing 3.12: `memory_02.v`

```
module test(
    input             WR1_CLK,  WR2_CLK,
    input             WR1_WEN,  WR2_WEN,
    input      [7:0]  WR1_ADDR, WR2_ADDR,
    input      [7:0]  WR1_DATA, WR2_DATA,
    input             RD1_CLK,  RD2_CLK,
    input      [7:0]  RD1_ADDR, RD2_ADDR,
    output reg [7:0]  RD1_DATA, RD2_DATA
);
```

(continues on next page)


RD2_CLK
RD1_CLK
RD2_ADDR
RD1_ADDR
0:0 - 1:1
0:0 - 0:0
7:0 - 15:8
7:0 - 7:0
2'00
2'11
2'00
WR1_WEN
WR2_WEN
8x 0:0 - 15:8
8x 0:0 - 7:0
WR1_DATA
WR2_DATA
7:0 - 15:8
7:0 - 7:0
WR1_CLK
WR2_CLK
0:0 - 1:1
0:0 - 0:0
WR1_ADDR
WR2_ADDR
7:0 - 15:8
7:0 - 7:0
RD_ADDR
RD_ARST
RD_CLK
RD_EN
RD_SRST
WR_ADDR
WR_CLK
WR_DATA
WR_EN
memory
$mem_v2
RD_DATA
15:8 - 7:0
7:0 - 7:0
RD2_DATA
RD1_DATA

(continued from previous page)

```
reg [7:0] memory [0:255];

always @(posedge WR1_CLK)
    if (WR1_WEN)
        memory[WR1_ADDR] <= WR1_DATA;

always @(posedge WR2_CLK)
    if (WR2_WEN)
        memory[WR2_ADDR] <= WR2_DATA;

always @(posedge RD1_CLK)
    RD1_DATA <= memory[RD1_ADDR];

always @(posedge RD2_CLK)
    RD2_DATA <= memory[RD2_ADDR];

endmodule
```

Listing 3.13: `memory_02.ys`

```
read_verilog memory_02.v
hierarchy -check -top test
proc;; memory -nomap
opt -mux_undef -mux_bool
```

Memory mapping

Usually it is preferred to use architecture-specific RAM resources for memory. For example:

```
memory -nomap
memory_libmap -lib my_memory_map.txt
techmap -map my_memory_map.v
memory_map
```

memory_libmap attempts to convert memory cells (*$mem_v2* etc) into hardware supported memory using a provided library (`my_memory_map.txt` in the example above). Where necessary, emulation logic is added to ensure functional equivalence before and after this conversion. `techmap -map my_memory_map.v` then uses `techmap` to map to hardware primitives. Any leftover memory cells unable to be converted are then picked up by *memory_map* and mapped to DFFs and address decoders.

> **Note**
>
> More information about what mapping options are available and associated costs of each can be found by enabling debug outputs. This can be done with the *debug* command, or by using the `-g` flag when calling Yosys to globally enable debug messages.

For more on the lib format for *memory_libmap*, see passes/memory/memlib.md

Supported memory patterns

Note that not all supported patterns are included in this document, of particular note is that combinations of multiple patterns should generally work. For example, *wbe* could be used in conjunction with any of the simple dual port (SDP) models. In general if a hardware memory definition does not support a given configuration, additional logic will be instantiated to guarantee behaviour is consistent with simulation.

Notes

Memory kind selection

The memory inference code will automatically pick target memory primitive based on memory geometry and features used. Depending on the target, there can be up to four memory primitive classes available for selection:

- FF RAM (aka logic): no hardware primitive used, memory lowered to a bunch of FFs and multiplexers
  - Can handle arbitrary number of write ports, as long as all write ports are in the same clock domain
  - Can handle arbitrary number and kind of read ports
- LUT RAM (aka distributed RAM): uses LUT storage as RAM
  - Supported on most FPGAs (with notable exception of ice40)
  - Usually has one synchronous write port, one or more asynchronous read ports
  - Small
  - Will never be used for ROMs (lowering to plain LUTs is always better)
- Block RAM: dedicated memory tiles
  - Supported on basically all FPGAs
  - Supports only synchronous reads
  - Two ports with separate clocks
  - Usually supports true dual port (with notable exception of ice40 that only supports SDP)
  - Usually supports asymmetric memories and per-byte write enables
  - Several kilobits in size
- Huge RAM:
  - Only supported on several targets:
    - Some Xilinx UltraScale devices (UltraRAM)
      - Two ports, both with mutually exclusive synchronous read and write
      - Single clock

        - Initial data must be all-0
      - Some ice40 devices (SPRAM)
        - Single port with mutually exclusive synchronous read and write
        - Does not support initial data
      - Nexus (large RAM)
        - Two ports, both with mutually exclusive synchronous read and write
        - Single clock
    - Will not be automatically selected by memory inference code, needs explicit opt-in via ram_style attribute

In general, you can expect the automatic selection process to work roughly like this:

- If any read port is asynchronous, only LUT RAM (or FF RAM) can be used.
- If there is more than one write port, only block RAM can be used, and this needs to be a hardware-supported true dual port pattern
  - ... unless all write ports are in the same clock domain, in which case FF RAM can also be used, but this is generally not what you want for anything but really small memories
- Otherwise, either FF RAM, LUT RAM, or block RAM will be used, depending on memory size

This process can be overridden by attaching a ram_style attribute to the memory:

- `(* ram_style = "logic" *)` selects FF RAM
- `(* ram_style = "distributed" *)` selects LUT RAM
- `(* ram_style = "block" *)` selects block RAM
- `(* ram_style = "huge" *)` selects huge RAM

It is an error if this override cannot be realized for the given target.

Many alternate spellings of the attribute are also accepted, for compatibility with other software.

Initial data

Most FPGA targets support initializing all kinds of memory to user-provided values. If explicit initialization is not used the initial memory value is undefined. Initial data can be provided by either initial statements writing memory cells one by one of `$readmemh` or `$readmemb` system tasks. For an example pattern, see *sr_init*.

Write port with byte enables

- Byte enables can be used with any supported pattern
- To ensure that multiple writes will be merged into one port, they need to have disjoint bit ranges, have the same address, and the same clock
- Any write enable granularity will be accepted (down to per-bit write enables), but using smaller granularity than natively supported by the target is very likely to be inefficient (eg. using 4-bit bytes on ECP5 will result in either padding the bytes with 5 dummy bits to native 9-bit units or splitting the RAM into two block RAMs)

```
reg [31 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable[0])
                mem[write_addr][7:0] <= write_data[7:0];
        if (write_enable[1])
                mem[write_addr][15:8] <= write_data[15:8];
        if (write_enable[2])
                mem[write_addr][23:16] <= write_data[23:16];
        if (write_enable[3])
                mem[write_addr][31:24] <= write_data[31:24];
        if (read_enable)
                read_data <= mem[read_addr];
end
```

Simple dual port (SDP) memory patterns

Todo

assorted enables, e.g. cen, wen+ren

Asynchronous-read SDP

- This will result in LUT RAM on supported targets

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
always @(posedge clk)
        if (write_enable)
                mem[write_addr] <= write_data;
assign read_data = mem[read_addr];
```

Synchronous SDP with clock domain crossing

- Will result in block RAM or LUT RAM depending on size
- No behavior guarantees in case of simultaneous read and write to the same address

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge write_clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;
end

always @(posedge read_clk) begin
        if (read_enable)
                read_data <= mem[read_addr];
end
```

Synchronous SDP read first

- The read and write parts can be in the same or different processes.
- Will result in block RAM or LUT RAM depending on size
- As long as the same clock is used for both, yosys will ensure read-first behavior. This may require extra circuitry on some targets for block RAM. If this is not necessary, use one of the patterns below.

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;
        if (read_enable)
                read_data <= mem[read_addr];
end
```

Synchronous SDP with undefined collision behavior

- Like above, but the read value is undefined when read and write ports target the same address in the same cycle

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;

        if (read_enable) begin
                read_data <= mem[read_addr];

        if (write_enable && read_addr == write_addr)
                // this if block
                read_data <= 'x;
        end
end
```

- Or below, using the no_rw_check attribute

```
(* no_rw_check *)
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;

        if (read_enable)
                read_data <= mem[read_addr];
end
```

Synchronous SDP with write-first behavior

- Will result in block RAM or LUT RAM depending on size
- May use additional circuitry for block RAM if write-first is not natively supported. Will always use additional circuitry for LUT RAM.

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;

        if (read_enable) begin
                read_data <= mem[read_addr];
                if (write_enable && read_addr == write_addr)
                        read_data <= write_data;
        end
end
```

Synchronous SDP with write-first behavior (alternate pattern)

- This pattern is supported for compatibility, but is much less flexible than the above

```
reg [ADDR_WIDTH - 1 : 0] read_addr_reg;
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;
        read_addr_reg <= read_addr;
end

assign read_data = mem[read_addr_reg];
```

Single-port RAM memory patterns

Asynchronous-read single-port RAM

- Will result in single-port LUT RAM on supported targets

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
always @(posedge clk)
        if (write_enable)
                mem[addr] <= write_data;
assign read_data = mem[addr];
```

Synchronous single-port RAM with mutually exclusive read/write

- Will result in single-port block RAM or LUT RAM depending on size
- This is the correct pattern to infer ice40 SPRAM (with manual ram_style selection)
- On targets that don't support read/write block RAM ports (eg. ice40), will result in SDP block RAM instead

- For block RAM, will use "NO_CHANGE" mode if available

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[addr] <= write_data;
        else if (read_enable)
                read_data <= mem[addr];
end
```

Synchronous single-port RAM with read-first behavior

- Will only result in single-port block RAM when read-first behavior is natively supported; otherwise, SDP RAM with additional circuitry will be used
- Many targets (Xilinx, ECP5, ...) can only natively support read-first/write-first single-port RAM (or TDP RAM) where the write_enable signal implies the read_enable signal (ie. can never write without reading). The memory inference code will run a simple SAT solver on the control signals to determine if this is the case, and insert emulation circuitry if it cannot be easily proven.

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[addr] <= write_data;
        if (read_enable)
                read_data <= mem[addr];
end
```

Synchronous single-port RAM with write-first behavior

- Will result in single-port block RAM or LUT RAM when supported
- Block RAMs will require extra circuitry if write-first behavior not natively supported

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[addr] <= write_data;
        if (read_enable)
                if (write_enable)
                        read_data <= write_data;
                else
                        read_data <= mem[addr];
end
```

Synchronous read port with initial value

- Initial read port values can be combined with any other supported pattern
- If block RAM is used and initial read port values are not natively supported by the target, small emulation circuit will be inserted

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
reg [DATA_WIDTH - 1 : 0] read_data;
initial read_data = 'h1234;

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;
        if (read_enable)
                read_data <= mem[read_addr];
end
```

Read register reset patterns

Resets can be combined with any other supported pattern (except that synchronous reset and asynchronous reset cannot both be used on a single read port). If block RAM is used and the selected reset (synchronous or asynchronous) is used but not natively supported by the target, small emulation circuitry will be inserted.

Synchronous reset, reset priority over enable

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;

        if (read_reset)
                read_data <= 'h1234;
        else if (read_enable)
                read_data <= mem[read_addr];
end
```

Synchronous reset, enable priority over reset

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;
        if (read_enable)
                if (read_reset)
                        read_data <= 'h1234;
                else
                        read_data <= mem[read_addr];
end
```

Synchronous read port with asynchronous reset

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
```

(continues on next page)

(continued from previous page)

```
        if (write_enable)
                mem[write_addr] <= write_data;
end

always @(posedge clk, posedge read_reset) begin
        if (read_reset)
                read_data <= 'h1234;
        else if (read_enable)
                read_data <= mem[read_addr];
end
```

Asymmetric memory patterns

To construct an asymmetric memory (memory with read/write ports of differing widths):

- Declare the memory with the width of the narrowest intended port
- Split all wide ports into multiple narrow ports
- To ensure the wide ports will be correctly merged:
  - For the address, use a concatenation of actual address in the high bits and a constant in the low bits
  - Ensure the actual address is identical for all ports belonging to the wide port
  - Ensure that clock is identical
  - For read ports, ensure that enable/reset signals are identical (for write ports, the enable signal may vary — this will result in using the byte enable functionality)

Asymmetric memory is supported on all targets, but may require emulation circuitry where not natively supported. Note that when the memory is larger than the underlying block RAM primitive, hardware asymmetric memory support is likely not to be used even if present as it is more expensive.

Wide synchronous read port

```
reg [7:0] mem [0:255];
wire [7:0] write_addr;
wire [5:0] read_addr;
wire [7:0] write_data;
reg [31:0] read_data;

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;
        if (read_enable) begin
                read_data[7:0] <= mem[{read_addr, 2'b00}];
                read_data[15:8] <= mem[{read_addr, 2'b01}];
                read_data[23:16] <= mem[{read_addr, 2'b10}];
                read_data[31:24] <= mem[{read_addr, 2'b11}];
        end
end
```

Wide asynchronous read port

- Note: the only target natively supporting this pattern is Xilinx UltraScale

```
reg [7:0] mem [0:511];
wire [8:0] write_addr;
wire [5:0] read_addr;
wire [7:0] write_data;
wire [63:0] read_data;

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;
end

assign read_data[7:0] = mem[{read_addr, 3'b000}];
assign read_data[15:8] = mem[{read_addr, 3'b001}];
assign read_data[23:16] = mem[{read_addr, 3'b010}];
assign read_data[31:24] = mem[{read_addr, 3'b011}];
assign read_data[39:32] = mem[{read_addr, 3'b100}];
assign read_data[47:40] = mem[{read_addr, 3'b101}];
assign read_data[55:48] = mem[{read_addr, 3'b110}];
assign read_data[63:56] = mem[{read_addr, 3'b111}];
```

Wide write port

```
reg [7:0] mem [0:255];
wire [5:0] write_addr;
wire [7:0] read_addr;
wire [31:0] write_data;
reg [7:0] read_data;

always @(posedge clk) begin
        if (write_enable[0])
                mem[{write_addr, 2'b00}] <= write_data[7:0];
        if (write_enable[1])
                mem[{write_addr, 2'b01}] <= write_data[15:8];
        if (write_enable[2])
                mem[{write_addr, 2'b10}] <= write_data[23:16];
        if (write_enable[3])
                mem[{write_addr, 2'b11}] <= write_data[31:24];
        if (read_enable)
                read_data <= mem[read_addr];
end
```

True dual port (TDP) patterns

- Many different variations of true dual port memory can be created by combining two single-port RAM patterns on the same memory
- When TDP memory is used, memory inference code has much less maneuver room to create requested semantics compared to individual single-port patterns (which can end up lowered to SDP memory where necessary) — supported patterns depend strongly on the target

- In particular, when both ports have the same clock, it's likely that "undefined collision" mode needs to be manually selected to enable TDP memory inference
- The examples below are non-exhaustive — many more combinations of port types are possible
- Note: if two write ports are in the same process, this defines a priority relation between them (if both ports are active in the same clock, the later one wins). On almost all targets, this will result in a bit of extra circuitry to ensure the priority semantics. If this is not what you want, put them in separate processes.
  - Priority is not supported when using the verific front end and any priority semantics are ignored.

TDP with different clocks, exclusive read/write

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk_a) begin
        if (write_enable_a)
                mem[addr_a] <= write_data_a;
        else if (read_enable_a)
                read_data_a <= mem[addr_a];
end

always @(posedge clk_b) begin
        if (write_enable_b)
                mem[addr_b] <= write_data_b;
        else if (read_enable_b)
                read_data_b <= mem[addr_b];
end
```

TDP with same clock, read-first behavior

- This requires hardware inter-port read-first behavior, and will only work on some targets (Xilinx, Nexus)

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable_a)
                mem[addr_a] <= write_data_a;
        if (read_enable_a)
                read_data_a <= mem[addr_a];
end

always @(posedge clk) begin
        if (write_enable_b)
                mem[addr_b] <= write_data_b;
        if (read_enable_b)
                read_data_b <= mem[addr_b];
end
```

TDP with multiple read ports

- The combination of a single write port with an arbitrary amount of read ports is supported on all targets — if a multi-read port primitive is available (like Xilinx RAM64M), it'll be used as appropriate. Otherwise, the memory will be automatically split into multiple primitives.

```
reg [31:0] mem [0:31];

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] <= write_data;
end

assign read_data_a = mem[read_addr_a];
assign read_data_b = mem[read_addr_b];
assign read_data_c = mem[read_addr_c];
```

Patterns only supported with Verific

The following patterns are only supported when the design is read in using the Verific front-end.

Synchronous SDP with write-first behavior via blocking assignments

- Use *sdp_wf* for compatibility with Yosys Verilog frontend.

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr] = write_data;

        if (read_enable)
                read_data <= mem[read_addr];
end
```

Asymmetric memories via part selection

- Build wide ports out of narrow ports instead (see *wide_sr*) for compatibility with Yosys Verilog frontend.

```
reg [31:0] mem [2**ADDR_WIDTH - 1 : 0];

wire [1:0] byte_lane;
wire [7:0] write_data;

always @(posedge clk) begin
        if (write_enable)
                mem[write_addr][byte_lane * 8 +: 8] <= write_data;

        if (read_enable)
                read_data <= mem[read_addr];
end
```

Undesired patterns

Asynchronous writes

- Not supported in modern FPGAs
- Not supported in yosys code anyhow

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @* begin
        if (write_enable)
                mem[write_addr] = write_data;
end

assign read_data = mem[read_addr];
```

3.1.5 Optimization passes

Yosys employs a number of optimizations to generate better and cleaner results. This chapter outlines these optimizations.

Todo

"outlines these optimizations" or "outlines *some*.."?

The *opt* macro command

The Yosys pass *opt* runs a number of simple optimizations. This includes removing unused signals and cells and const folding. It is recommended to run this pass after each major step in the synthesis script. This macro command calls the following `opt_*` commands:

Listing 3.14: Passes called by *opt*

```
opt_expr
opt_merge -nomux

do
    opt_muxtree
    opt_reduce
    opt_merge
    opt_share  (-full only)
    opt_dff  (except when called with -noff)
    opt_hier (-hier only)
    opt_clean
    opt_expr
while <changed design>
```

Constant folding and simple expression rewriting - *opt_expr*

Todo

unsure if this is too much detail and should be in *Yosys internals*

This pass performs constant folding on the internal combinational cell types described in *Internal cell library*. This means a cell with all constant inputs is replaced with the constant value this cell drives. In some cases this pass can also optimize cells with some constant inputs.

Table 3.1: Const folding rules for *$_AND_* cells as used in *opt_expr*.

A-Input	B-Input	Replacement
any	0	0
0	any	0
1	1	1
X/Z	X/Z	X
1	X/Z	X
X/Z	1	X
any	X/Z	0
X/Z	any	0
a	1	*a*
1	*b*	*b*

Table 3.1 shows the replacement rules used for optimizing an *$_AND_* gate. The first three rules implement the obvious const folding rules. Note that 'any' might include dynamic values calculated by other parts of the circuit. The following three lines propagate undef (X) states. These are the only three cases in which it is allowed to propagate an undef according to Sec. 5.1.10 of IEEE Std. 1364-2005 [A+06].

The next two lines assume the value 0 for undef states. These two rules are only used if no other substitutions are possible in the current module. If other substitutions are possible they are performed first, in the hope that the 'any' will change to an undef value or a 1 and therefore the output can be set to undef.

The last two lines simply replace an *$_AND_* gate with one constant-1 input with a buffer.

Besides this basic const folding the *opt_expr* pass can replace 1-bit wide *$eq* and *$ne* cells with buffers or not-gates if one input is constant. Equality checks may also be reduced in size if there are redundant bits in the arguments (i.e. bits which are constant on both inputs). This can, for example, result in a 32-bit wide constant like `255` being reduced to the 8-bit value of `8'11111111` if the signal being compared is only 8-bit as in *addr_gen module after opt_expr; clean* of *Synthesis starter*.

The *opt_expr* pass is very conservative regarding optimizing *$mux* cells, as these cells are often used to model decision-trees and breaking these trees can interfere with other optimizations.

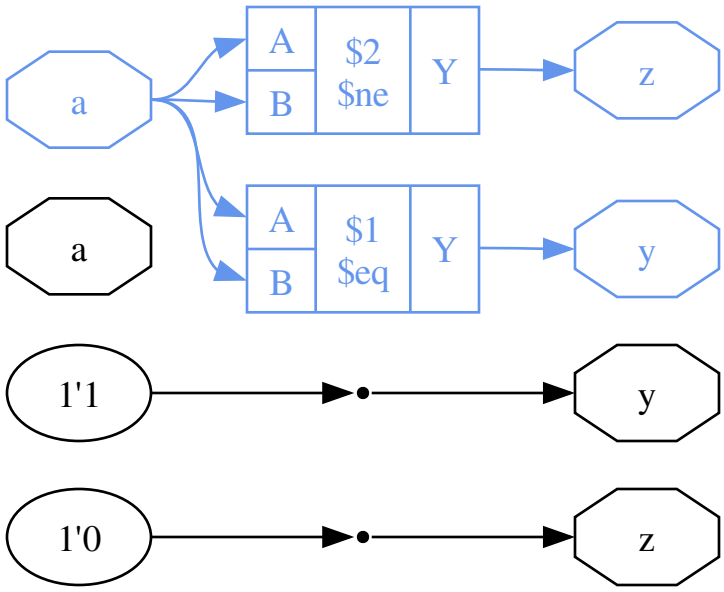


Fig. 3.1: Before and after *opt_expr*

Listing 3.15: example verilog for demonstrating *opt_expr*

```
module uut(
    input a,
    output y, z
);
    assign y = a == a;
    assign z = a != a;
endmodule
```

Merging identical cells - *opt_merge*

This pass performs trivial resource sharing. This means that this pass identifies cells with identical inputs and replaces them with a single instance of the cell.

The option `-nomux` can be used to disable resource sharing for multiplexer cells (*$mux* and *$pmux*.) This can be useful as it prevents multiplexer trees to be merged, which might prevent *opt_muxtree* to identify possible optimizations.

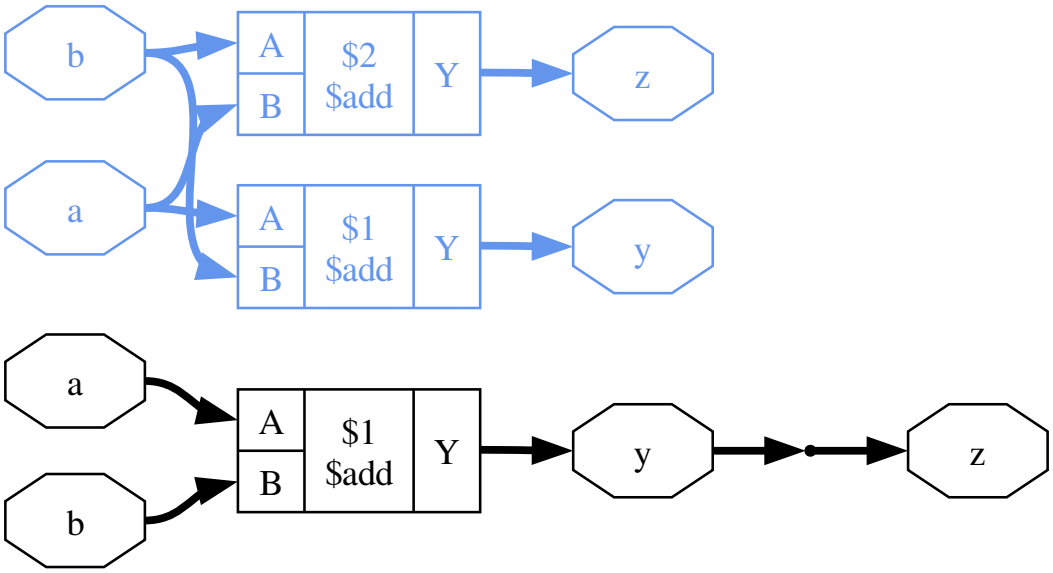


Fig. 3.2: Before and after *opt_merge*

Listing 3.16: example verilog for demonstrating *opt_merge*

```
module uut(
    input  [3:0] a, b,
    output [3:0] y, z
);
    assign y = a + b;
    assign z = b + a;
endmodule
```

Removing never-active branches from multiplexer tree - *opt_muxtree*

This pass optimizes trees of multiplexer cells by analyzing the select inputs. Consider the following simple example:

Listing 3.17: example verilog for demonstrating *opt_muxtree*

```
module uut(
    input a, b, c, d,
    output y
);
    assign y = a ? (a ? b : c) : d;
endmodule
```

The output can never be `c`, as this would require `a` to be 1 for the outer multiplexer and 0 for the inner

multiplexer. The *opt_muxtree* pass detects this contradiction and replaces the inner multiplexer with a constant 1, yielding the logic for `y = a ? b : d`.

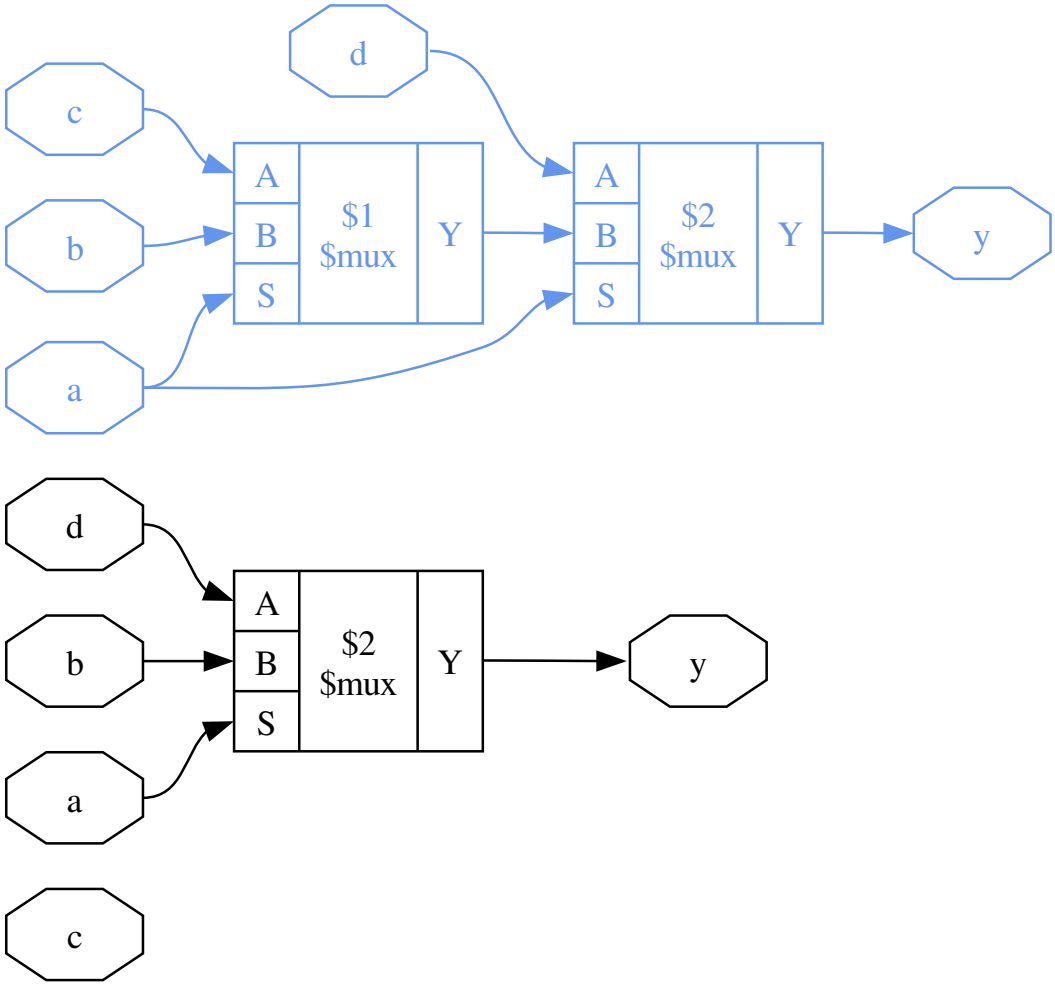


Fig. 3.3: Before and after *opt_muxtree*

Simplifying large MUXes and AND/OR gates - *opt_reduce*

This is a simple optimization pass that identifies and consolidates identical input bits to *$reduce_and* and *$reduce_or* cells. It also sorts the input bits to ease identification of shareable *$reduce_and* and *$reduce_or* cells in other passes.

This pass also identifies and consolidates identical inputs to multiplexer cells. In this case the new shared select bit is driven using a *$reduce_or* cell that combines the original select bits.

Lastly this pass consolidates trees of *$reduce_and* cells and trees of *$reduce_or* cells to single large *$reduce_and* or *$reduce_or* cells.

These three simple optimizations are performed in a loop until a stable result is produced.

Merging mutually exclusive cells with shared inputs - *opt_share*

This pass identifies mutually exclusive cells of the same type that:

a. share an input signal, and

b. drive the same *$mux*, *$_MUX_*, or *$pmux* multiplexing cell,

allowing the cell to be merged and the multiplexer to be moved from multiplexing its output to multiplexing the non-shared input signals.

Listing 3.18: example verilog for demonstrating *opt_share*

```
module uut(
    input  [15:0] a, b,
    input         sel,
    output [15:0] res,
);
    assign res = {sel ? a + b : a - b};
endmodule
```

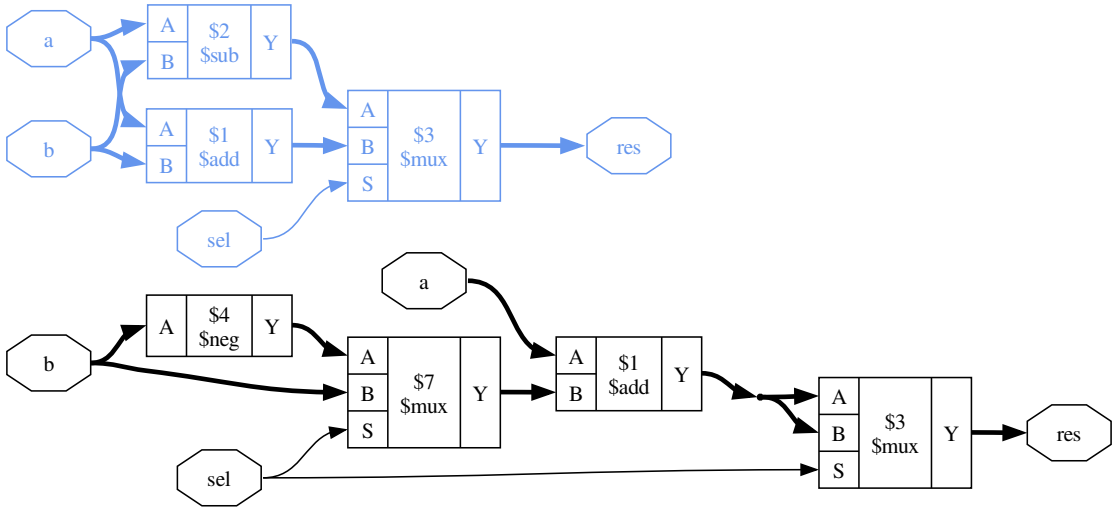


Fig. 3.4: Before and after *opt_share*

When running *opt* in full, the original *$mux* (labeled $3) is optimized away by *opt_expr*.

Performing DFF optimizations - *opt_dff*

Todo

`$_DFF_` isn't a valid cell

This pass identifies single-bit d-type flip-flops (`$_DFF_`, *`$dff`*, and *`$adff`* cells) with a constant data input and replaces them with a constant driver. It can also merge clock enables and synchronous reset multiplexers, removing unused control inputs.

Called with `-nodffe` and `-nosdff`, this pass is used to prepare a design for *FSM handling*.

Hierarchical optimization - *`opt_hier`* pass

This pass considers the design hierarchy and propagates unused signals, constant signals, and tied-together signals across module boundaries to facilitate optimization by other passes.

Removing unused cells and wires - *`opt_clean`* pass

This pass identifies unused signals and cells and removes them from the design. It also creates an `unused_bits` attribute on wires with unused bits. This attribute can be used for debugging or by other optimization passes.

When to use *`opt`* or *`clean`*

Usually it does not hurt to call *`opt`* after each regular command in the synthesis script. But it increases the synthesis time, so it is favourable to only call *`opt`* when an improvement can be achieved.

It is generally a good idea to call *`opt`* before inherently expensive commands such as *`sat`* or *`freduce`*, as the possible gain is much higher in these cases as the possible loss.

The *`clean`* command, which is an alias for *`opt_clean`* with fewer outputs, on the other hand is very fast and many commands leave a mess (dangling signal wires, etc). For example, most commands do not remove any wires or cells. They just change the connections and depend on a later call to clean to get rid of the now unused objects. So the occasional `;;`, which itself is an alias for *`clean`*, is a good idea in every synthesis script, e.g:

```
hierarchy; proc; opt; memory; opt_expr;; fsm;;
```

Other optimizations

Todo

more on the other optimizations

- Check *Optimization passes* for more.
- `abc` and `abc9`, see also: *The ABC toolbox*.

3.1.6 Technology mapping

Todo

less academic, check text is coherent

Previous chapters outlined how HDL code is transformed into an RTL netlist. The RTL netlist is still based on abstract coarse-grain cell types like arbitrary width adders and even multipliers. This chapter covers how an RTL netlist is transformed into a functionally equivalent netlist utilizing the cell types available in the target architecture.

Technology mapping is often performed in two phases. In the first phase RTL cells are mapped to an internal library of single-bit cells (see *Gate-level cells*). In the second phase this netlist of internal gate types is transformed to a netlist of gates from the target technology library.

When the target architecture provides coarse-grain cells (such as block ram or ALUs), these must be mapped to directly form the RTL netlist, as information on the coarse-grain structure of the design is lost when it is mapped to bit-width gate types.

Cell substitution

The simplest form of technology mapping is cell substitution, as performed by the techmap pass. This pass, when provided with a Verilog file that implements the RTL cell types using simpler cells, simply replaces the RTL cells with the provided implementation.

When no map file is provided, techmap uses a built-in map file that maps the Yosys RTL cell types to the internal gate library used by Yosys. The curious reader may find this map file as `techlibs/common/techmap.v` in the Yosys source tree.

Additional features have been added to techmap to allow for conditional mapping of cells (see *Technology mapping*). This can for example be useful if the target architecture supports hardware multipliers for certain bit-widths but not for others.

A usual synthesis flow would first use the techmap pass to directly map some RTL cells to coarse-grain cells provided by the target architecture (if any) and then use techmap with the built-in default file to map the remaining RTL cells to gate logic.

Subcircuit substitution

Sometimes the target architecture provides cells that are more powerful than the RTL cells used by Yosys. For example a cell in the target architecture that can calculate the absolute-difference of two numbers does not match any single RTL cell type but only combinations of cells.

For these cases Yosys provides the extract pass that can match a given set of modules against a design and identify the portions of the design that are identical (i.e. isomorphic subcircuits) to any of the given modules. These matched subcircuits are then replaced by instances of the given modules.

The extract pass also finds basic variations of the given modules, such as swapped inputs on commutative cell types.

In addition to this the extract pass also has limited support for frequent subcircuit mining, i.e. the process of finding recurring subcircuits in the design. This has a few applications, including the design of new coarse-grain architectures [GW13].

The hard algorithmic work done by the extract pass (solving the isomorphic subcircuit problem and frequent subcircuit mining) is performed using the SubCircuit library that can also be used stand-alone without Yosys (see *SubCircuit*).

Gate-level technology mapping

Todo

newer techmap libraries appear to be largely `.v` instead of `.lib`

On the gate-level the target architecture is usually described by a "Liberty file". The Liberty file format is an industry standard format that can be used to describe the behaviour and other properties of standard library cells .

Mapping a design utilizing the Yosys internal gate library (e.g. as a result of mapping it to this representation using the techmap pass) is performed in two phases.

First the register cells must be mapped to the registers that are available on the target architectures. The target architecture might not provide all variations of d-type flip-flops with positive and negative clock edge, high-active and low-active asynchronous set and/or reset, etc. Therefore the process of mapping the registers might add additional inverters to the design and thus it is important to map the register cells first.

Mapping of the register cells may be performed by using the dfflibmap pass. This pass expects a Liberty file as argument (using the -liberty option) and only uses the register cells from the Liberty file.

Secondly the combinational logic must be mapped to the target architecture. This is done using the external program ABC via the abc pass by using the -liberty option to the pass. Note that in this case only the combinatorial cells are used from the cell library.

Occasionally Liberty files contain trade secrets (such as sensitive timing information) that cannot be shared freely. This complicates processes such as reporting bugs in the tools involved. When the information in the Liberty file used by Yosys and ABC are not part of the sensitive information, the additional tool yosys-filterlib (see *yosys-filterlib*) can be used to strip the sensitive information from the Liberty file.

3.1.7 The extract pass

- Like the `techmap` pass, the `extract` pass is called with a map file. It compares the circuits inside the modules of the map file with the design and looks for sub-circuits in the design that match any of the modules in the map file.
- If a match is found, the `extract` pass will replace the matching subcircuit with an instance of the module from the map file.
- In a way the `extract` pass is the inverse of the techmap pass.

Todo

add/expand supporting text, also mention custom pattern matching and pmgen

Example code can be found in `docs/source/code_examples/macc`.

```
read_verilog macc_simple_test.v
hierarchy -check -top test;;
```

```
extract -constports -map macc_simple_xmap.v;;
```

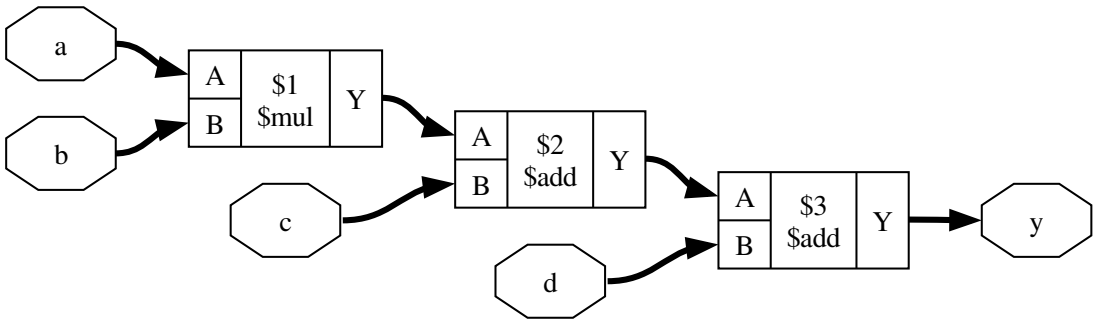


Fig. 3.5: before `extract`

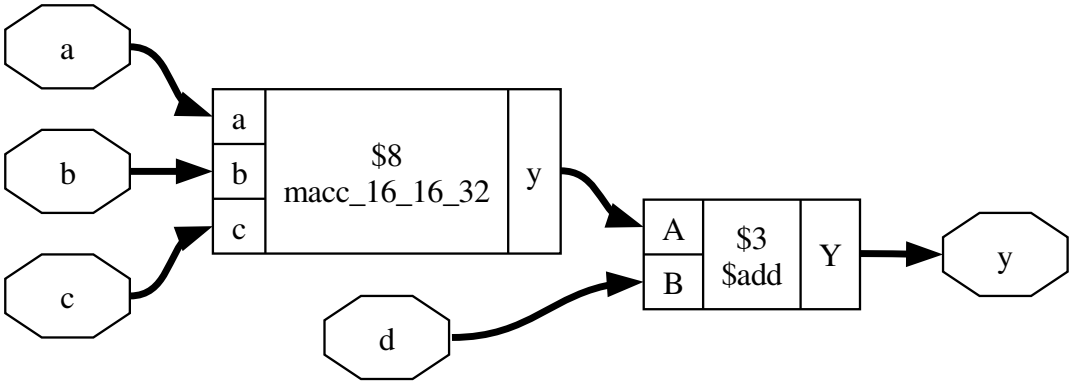


Fig. 3.6: after `extract`

Listing 3.19: `macc_simple_test.v`

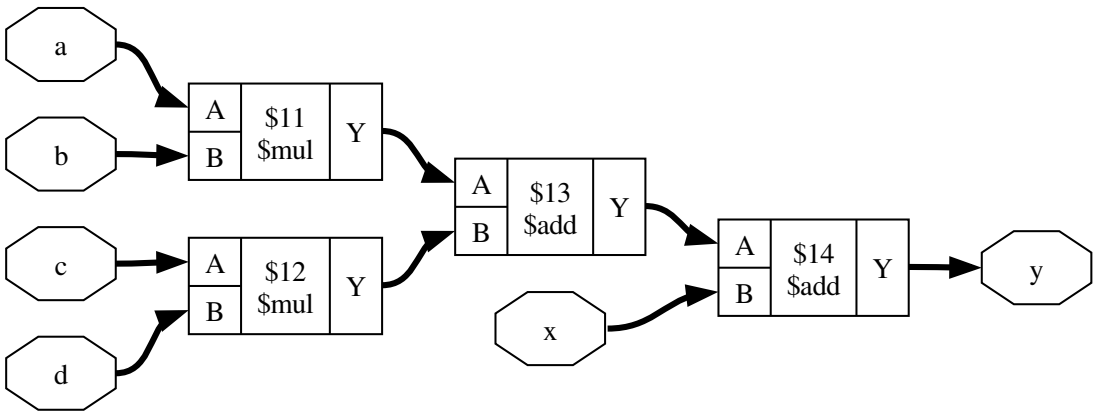
```
module test(a, b, c, d, y);
input [15:0] a, b;
input [31:0] c, d;
output [31:0] y;
assign y = a * b + c + d;
endmodule
```

Listing 3.20: `macc_simple_xmap.v`

```
module macc_16_16_32(a, b, c, y);
input [15:0] a, b;
input [31:0] c;
output [31:0] y;
assign y = a*b + c;
endmodule
```

Listing 3.21: `macc_simple_test_01.v`

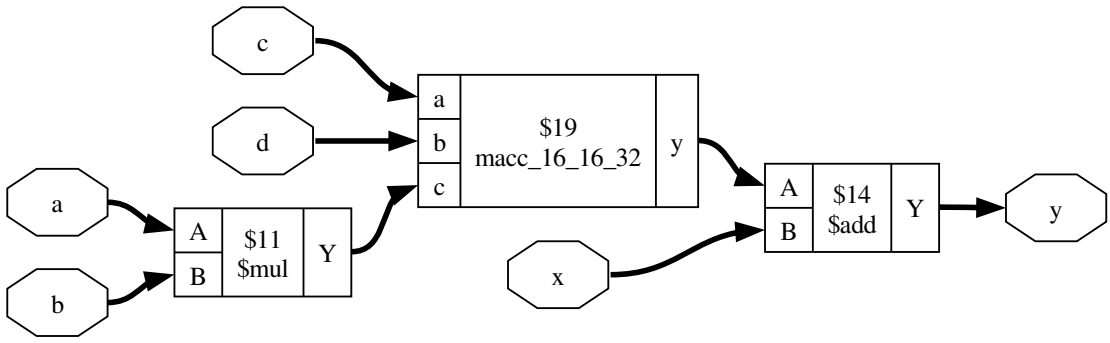
```
module test(a, b, c, d, x, y);
input [15:0] a, b, c, d;
input [31:0] x;
output [31:0] y;
assign y = a*b + c*d + x;
endmodule
```



Listing 3.22: `macc_simple_test_02.v`

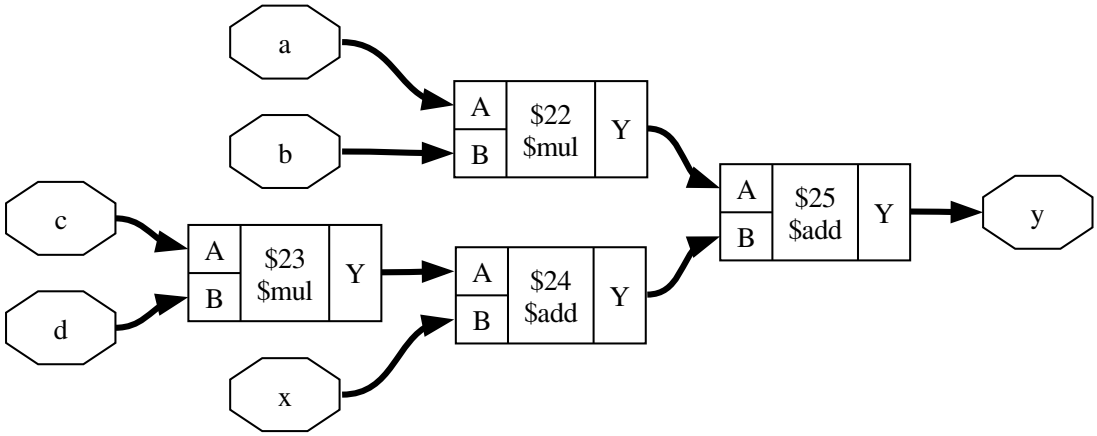
```
module test(a, b, c, d, x, y);
input [15:0] a, b, c, d;
input [31:0] x;
output [31:0] y;
```

(continues on next page)



(continued from previous page)

```
assign y = a*b + (c*d + x);
endmodule
```



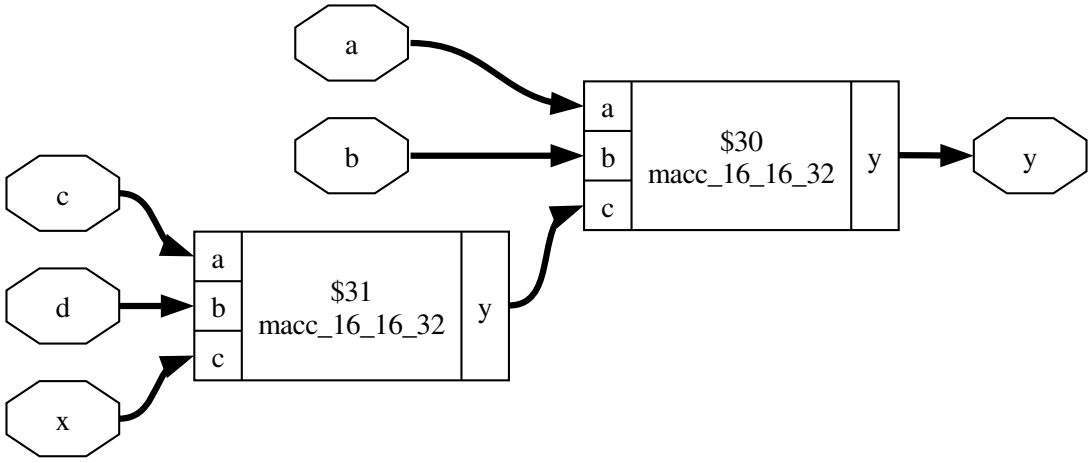
The wrap-extract-unwrap method

Often a coarse-grain element has a constant bit-width, but can be used to implement operations with a smaller bit-width. For example, a 18x25-bit multiplier can also be used to implement 16x20-bit multiplication.

A way of mapping such elements in coarse grain synthesis is the wrap-extract-unwrap method:

wrap
: Identify candidate-cells in the circuit and wrap them in a cell with a constant wider bit-width using `techmap`. The wrappers use the same parameters as the original cell, so the information about the original width of the ports is preserved. Then use the `connwrappers` command to connect up the bit-extended in- and outputs of the wrapper cells.

extract
: Now all operations are encoded using the same bit-width as the coarse grain element. The `extract`



command can be used to replace circuits with cells of the target architecture.

unwrap
The remaining wrapper cell can be unwrapped using `techmap`.

Example: DSP48_MACC

This section details an example that shows how to map MACC operations of arbitrary size to MACC cells with a 18x25-bit multiplier and a 48-bit adder (such as the Xilinx DSP48 cells).

Preconditioning: `macc_xilinx_swap_map.v`

Make sure `A` is the smaller port on all multipliers

Todo

add/expand supporting text

Listing 3.23: `macc_xilinx_swap_map.v`

```
(* techmap_celltype = "$mul" *)
module mul_swap_ports (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH = 1;
parameter B_WIDTH = 1;
parameter Y_WIDTH = 1;

input [A_WIDTH-1:0] A;
input [B_WIDTH-1:0] B;
output [Y_WIDTH-1:0] Y;
```

(continues on next page)

(continued from previous page)

```
wire _TECHMAP_FAIL_ = A_WIDTH <= B_WIDTH;

\$mul #(
        .A_SIGNED(B_SIGNED),
        .B_SIGNED(A_SIGNED),
        .A_WIDTH(B_WIDTH),
        .B_WIDTH(A_WIDTH),
        .Y_WIDTH(Y_WIDTH)
) _TECHMAP_REPLACE_ (
        .A(B),
        .B(A),
        .Y(Y)
);

endmodule
```

Wrapping multipliers: `macc_xilinx_wrap_map.v`

Listing 3.24: `macc_xilinx_wrap_map.v`

```
(* techmap_celltype = "$mul" *)
module mul_wrap (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH = 1;
parameter B_WIDTH = 1;
parameter Y_WIDTH = 1;

input [A_WIDTH-1:0] A;
input [B_WIDTH-1:0] B;
output [Y_WIDTH-1:0] Y;

wire [17:0] A_18 = A;
wire [24:0] B_25 = B;
wire [47:0] Y_48;
assign Y = Y_48;

wire [1023:0] _TECHMAP_DO_ = "proc; clean";

reg _TECHMAP_FAIL_;
initial begin
        _TECHMAP_FAIL_ <= 0;
        if (A_SIGNED || B_SIGNED)
                _TECHMAP_FAIL_ <= 1;
        if (A_WIDTH < 4 || B_WIDTH < 4)
                _TECHMAP_FAIL_ <= 1;
        if (A_WIDTH > 18 || B_WIDTH > 25)
                _TECHMAP_FAIL_ <= 1;
        if (A_WIDTH*B_WIDTH < 100)
                _TECHMAP_FAIL_ <= 1;
end
```

(continues on next page)

(continued from previous page)

```
\$__mul_wrapper #(
        .A_SIGNED(A_SIGNED),
        .B_SIGNED(B_SIGNED),
        .A_WIDTH(A_WIDTH),
        .B_WIDTH(B_WIDTH),
        .Y_WIDTH(Y_WIDTH)
) _TECHMAP_REPLACE_ (
        .A(A_18),
        .B(B_25),
        .Y(Y_48)
);

endmodule
```

Wrapping adders: `macc_xilinx_wrap_map.v`

Listing 3.25: `macc_xilinx_wrap_map.v`

```
(* techmap_celltype = "$add" *)
module add_wrap (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH = 1;
parameter B_WIDTH = 1;
parameter Y_WIDTH = 1;

input [A_WIDTH-1:0] A;
input [B_WIDTH-1:0] B;
output [Y_WIDTH-1:0] Y;

wire [47:0] A_48 = A;
wire [47:0] B_48 = B;
wire [47:0] Y_48;
assign Y = Y_48;

wire [1023:0] _TECHMAP_DO_ = "proc; clean";

reg _TECHMAP_FAIL_;
initial begin
        _TECHMAP_FAIL_ <= 0;
        if (A_SIGNED || B_SIGNED)
                _TECHMAP_FAIL_ <= 1;
        if (A_WIDTH < 10 && B_WIDTH < 10)
                _TECHMAP_FAIL_ <= 1;
end

\$__add_wrapper #(
        .A_SIGNED(A_SIGNED),
        .B_SIGNED(B_SIGNED),
        .A_WIDTH(A_WIDTH),
```

(continues on next page)

(continued from previous page)

```
        .B_WIDTH(B_WIDTH),
        .Y_WIDTH(Y_WIDTH)
) _TECHMAP_REPLACE_ (
        .A(A_48),
        .B(B_48),
        .Y(Y_48)
);

endmodule
```

Extract: `macc_xilinx_xmap.v`

Listing 3.26: `macc_xilinx_xmap.v`

```
module DSP48_MACC (a, b, c, y);

input [17:0] a;
input [24:0] b;
input [47:0] c;
output [47:0] y;

assign y = a*b + c;

endmodule
```

. . . simply use the same wrapping commands on this module as on the design to create a template for the `extract` command.

Unwrapping multipliers: `macc_xilinx_unwrap_map.v`

Listing 3.27: `$__mul_wrapper` module in `macc_xilinx_unwrap_map.v`

```
module \$__mul_wrapper (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH = 1;
parameter B_WIDTH = 1;
parameter Y_WIDTH = 1;

input [17:0] A;
input [24:0] B;
output [47:0] Y;

wire [A_WIDTH-1:0] A_ORIG = A;
wire [B_WIDTH-1:0] B_ORIG = B;
wire [Y_WIDTH-1:0] Y_ORIG;
assign Y = Y_ORIG;

\$mul #(
        .A_SIGNED(A_SIGNED),
        .B_SIGNED(B_SIGNED),
```

(continues on next page)

(continued from previous page)

```
        .A_WIDTH(A_WIDTH),
        .B_WIDTH(B_WIDTH),
        .Y_WIDTH(Y_WIDTH)
) _TECHMAP_REPLACE_ (
        .A(A_ORIG),
        .B(B_ORIG),
        .Y(Y_ORIG)
);

endmodule
```

Unwrapping adders: `macc_xilinx_unwrap_map.v`

Listing 3.28: `$__add_wrapper` module in `macc_xilinx_unwrap_map.v`

```
module \$__add_wrapper (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH = 1;
parameter B_WIDTH = 1;
parameter Y_WIDTH = 1;

input [47:0] A;
input [47:0] B;
output [47:0] Y;

wire [A_WIDTH-1:0] A_ORIG = A;
wire [B_WIDTH-1:0] B_ORIG = B;
wire [Y_WIDTH-1:0] Y_ORIG;
assign Y = Y_ORIG;

\$add #(
        .A_SIGNED(A_SIGNED),
        .B_SIGNED(B_SIGNED),
        .A_WIDTH(A_WIDTH),
        .B_WIDTH(B_WIDTH),
        .Y_WIDTH(Y_WIDTH)
) _TECHMAP_REPLACE_ (
        .A(A_ORIG),
        .B(B_ORIG),
        .Y(Y_ORIG)
);

endmodule
```

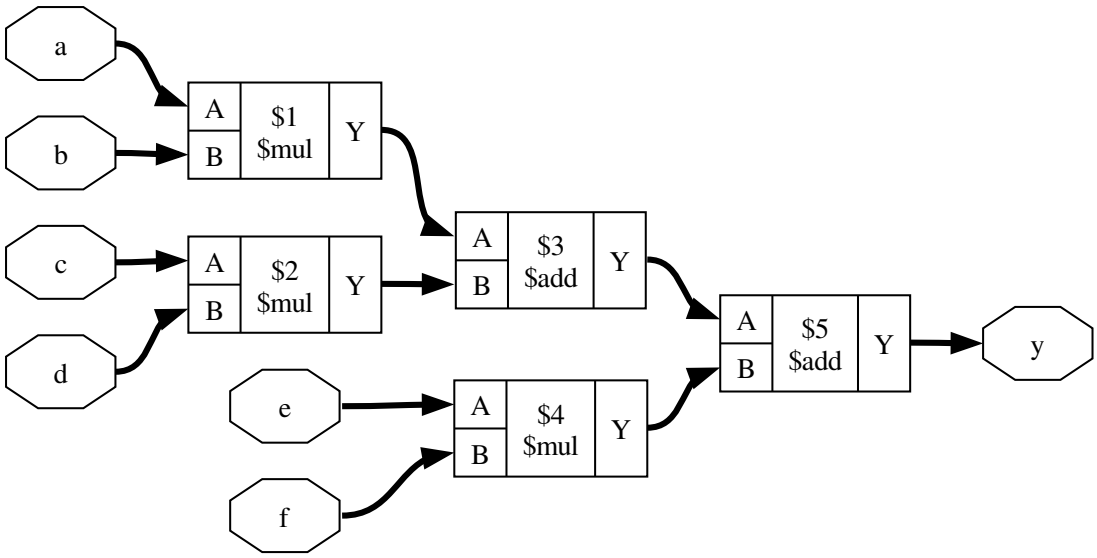
Listing 3.29: `test1` of `macc_xilinx_test.v`

```
module test1(a, b, c, d, e, f, y);
    input [19:0] a, b, c;
    input [15:0] d, e, f;
```

(continues on next page)

(continued from previous page)

```
    output [41:0] y;
    assign y = a*b + c*d + e*f;
endmodule
```



Listing 3.30: `test2` of `macc_xilinx_test.v`

```
module test2(a, b, c, d, e, f, y);
    input [19:0] a, b, c;
    input [15:0] d, e, f;
    output [41:0] y;
    assign y = a*b + (c*d + e*f);
endmodule
```

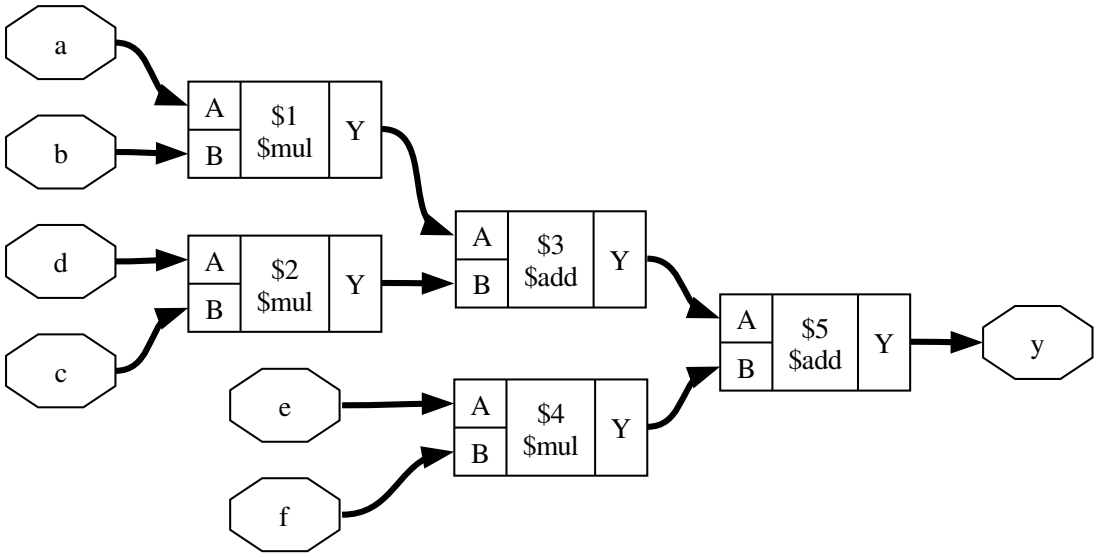
Wrapping in `test1`:

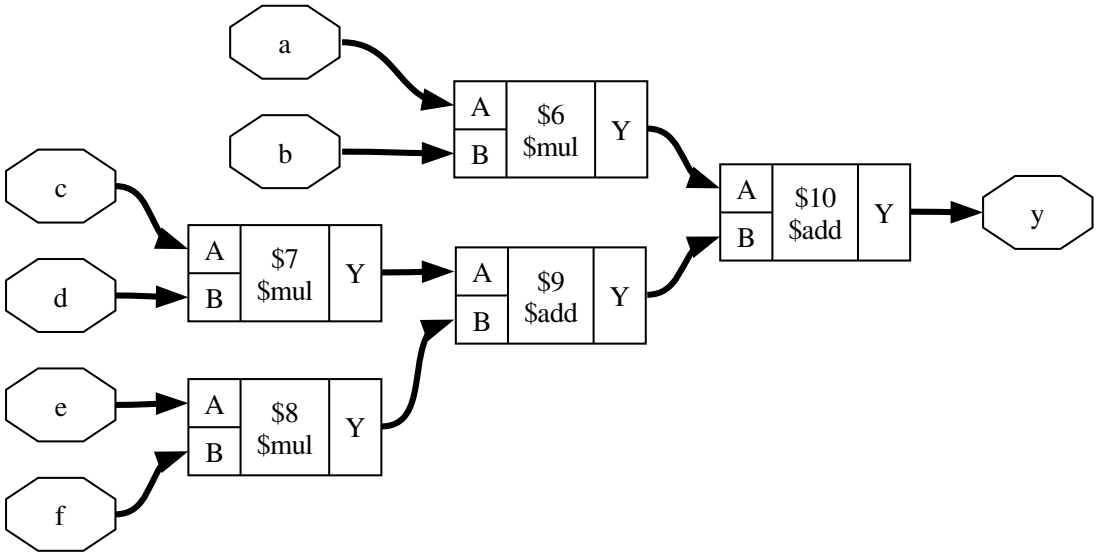
```
techmap -map macc_xilinx_wrap_map.v

connwrappers -unsigned $__mul_wrapper Y Y_WIDTH \
             -unsigned $__add_wrapper Y Y_WIDTH;;
```

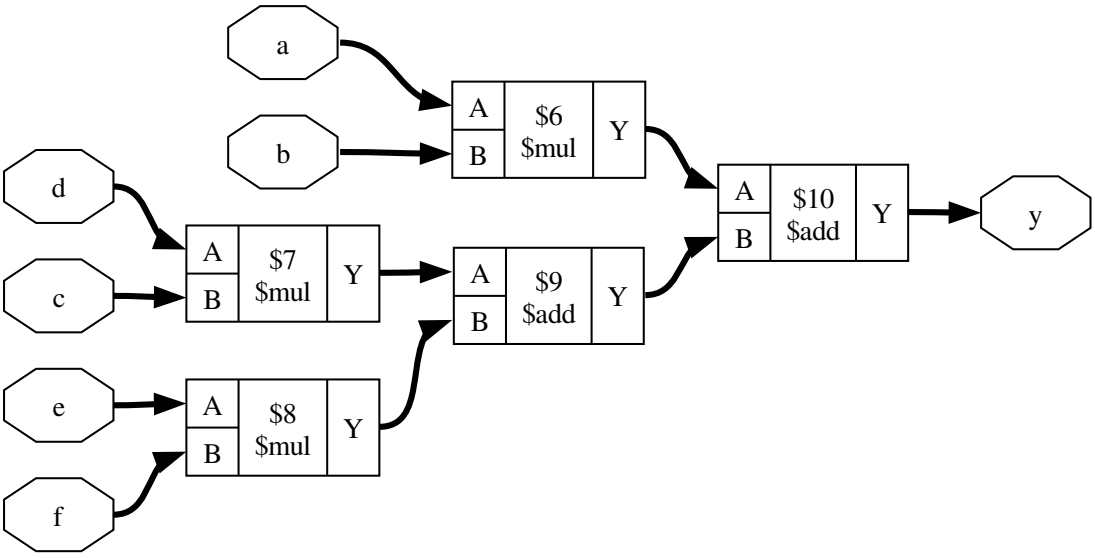
Wrapping in `test2`:

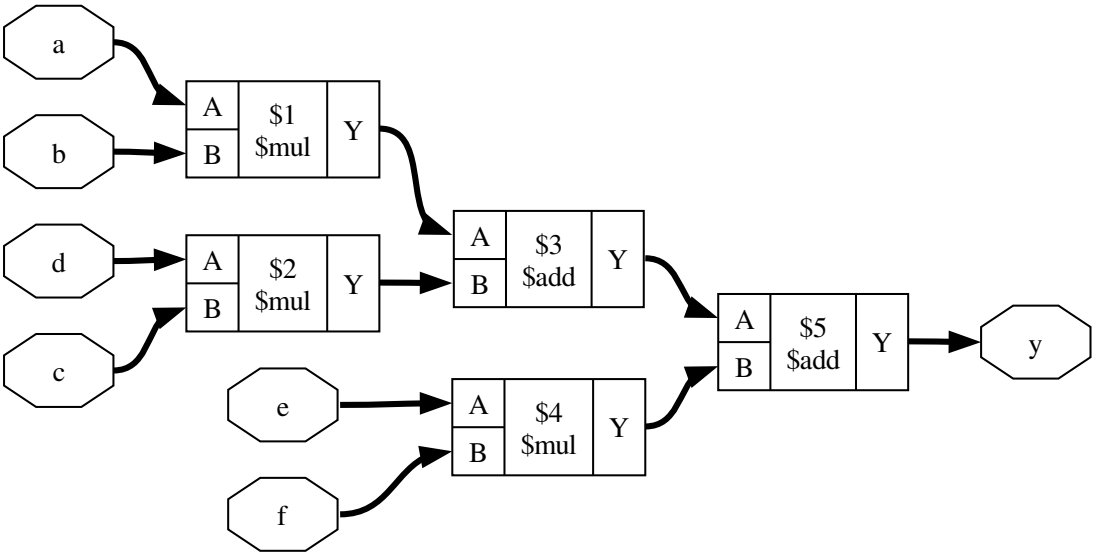
```
techmap -map macc_xilinx_wrap_map.v

connwrappers -unsigned $__mul_wrapper Y Y_WIDTH \
             -unsigned $__add_wrapper Y Y_WIDTH;;
```

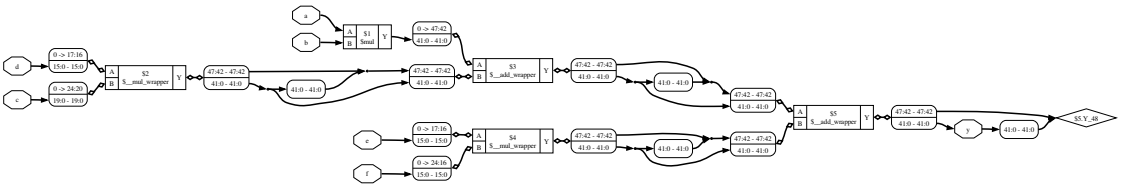
Extract in `test1`:

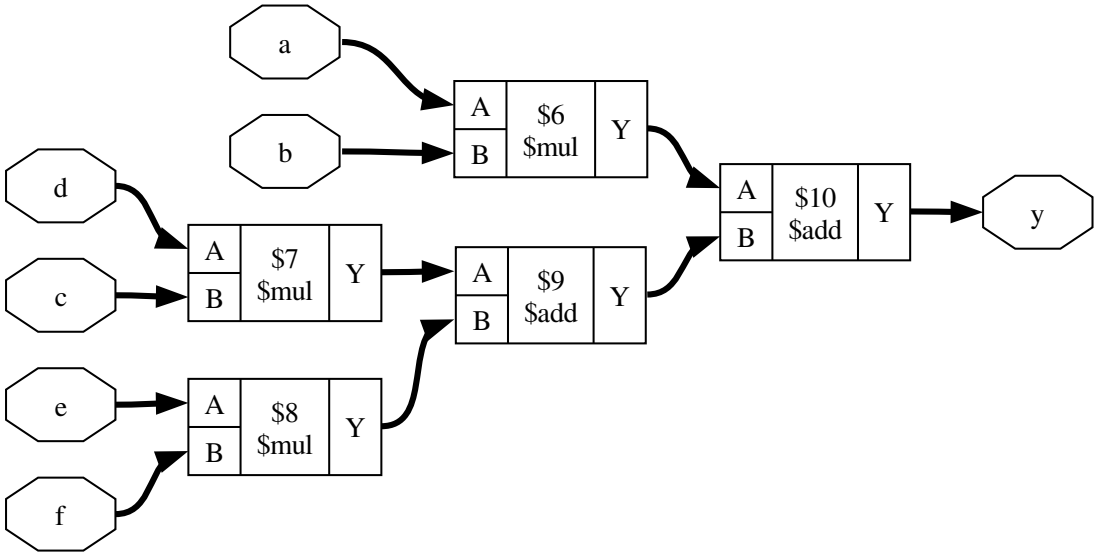

a
b
d
c
e
f
A
B
$1
$mul
Y
$2
$mul
$3
$add
$4
$mul
$5
$add
y

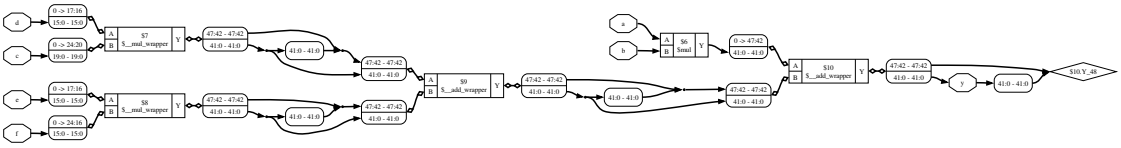

a
b
c
d
e
f
A
B
$6
$mul
Y
$7
$mul
$8
$mul
$9
$add
$10
$add
y

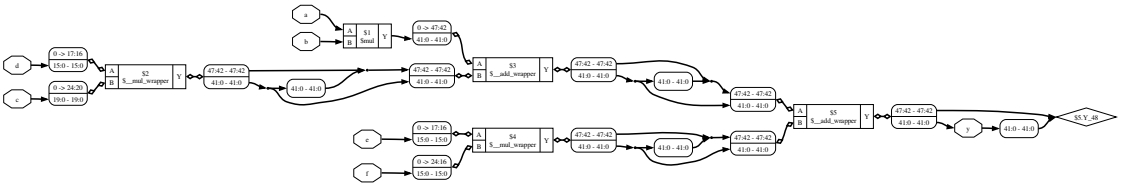

a
b
d
c
e
f
A
B
$6
$mul
Y
$7
$mul
$8
$mul
$9
$add
$10
$add
y


a
b
d
c
e
f
A
B
$1
$mul
$2
$mul
$3
$add
$4
$mul
$5
$add
Y
y

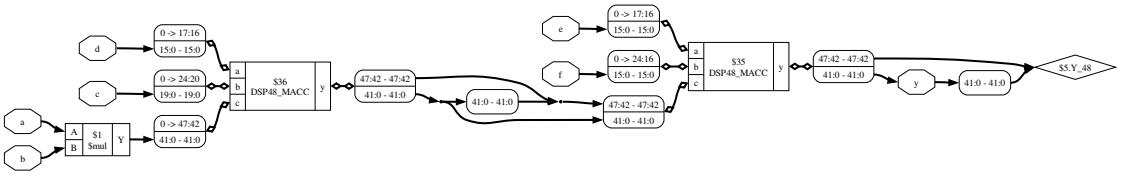



a
b
d
c
e
f
A
B
$6
$mul
Y
$7
$mul
$8
$mul
$9
$add
$10
$add
y

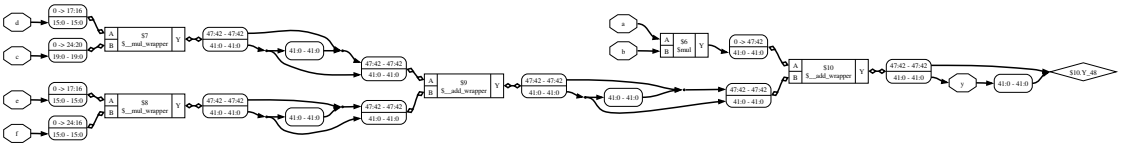




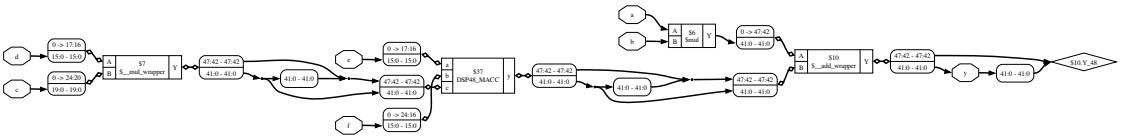
```
design -push
read_verilog macc_xilinx_xmap.v
techmap -map macc_xilinx_swap_map.v
techmap -map macc_xilinx_wrap_map.v;;
design -save __macc_xilinx_xmap
design -pop

extract -constports -ignore_parameters \
        -map %__macc_xilinx_xmap       \
        -swap $__add_wrapper A,B ;;
```



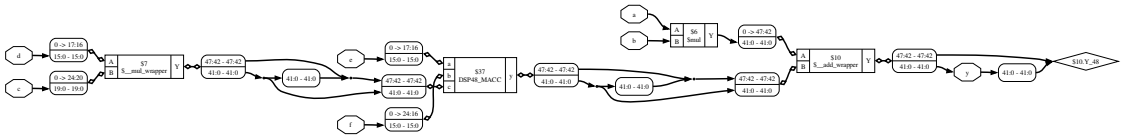
Extract in `test2`:



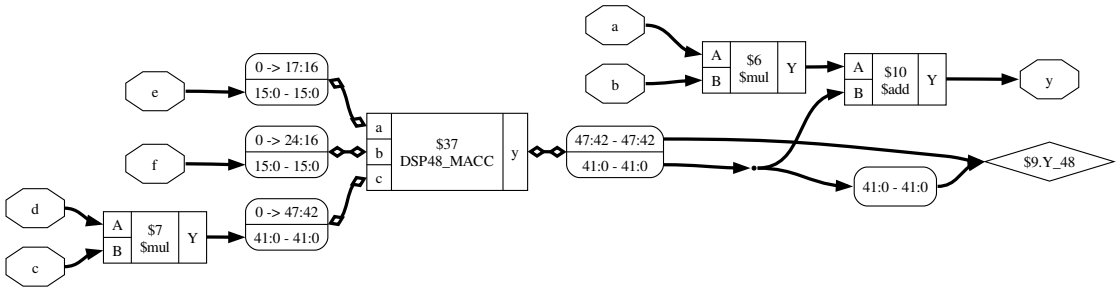

```
design -push
read_verilog macc_xilinx_xmap.v
techmap -map macc_xilinx_swap_map.v
techmap -map macc_xilinx_wrap_map.v;;
design -save __macc_xilinx_xmap
design -pop

extract -constports -ignore_parameters \
        -map %__macc_xilinx_xmap       \
        -swap $__add_wrapper A,B ;;
```



Unwrap in `test2`:



```
techmap -map macc_xilinx_unwrap_map.v;;
```



3.1.8 The ABC toolbox

ABC, from the University of California, Berkeley, is a logic toolbox used for fine-grained optimisation and LUT mapping.

Yosys has two different commands, which both use this logic toolbox, but use it in different ways.

The `abc` pass can be used for both ASIC (e.g. `abc -liberty`) and FPGA (`abc -lut`) mapping, but this page will focus on FPGA mapping.

The `abc9` pass generally provides superior mapping quality due to being aware of combination boxes and DFF and LUT timings, giving it a more global view of the mapping problem.

ABC: the unit delay model, simple and efficient

The `abc` pass uses a highly simplified view of an FPGA:

- An FPGA is made up of a network of inputs that connect through LUTs to a network of outputs. These inputs may actually be I/O pins, D flip-flops, memory blocks or DSPs, but ABC is unaware of this.
- Each LUT has 1 unit of delay between an input and its output, and this applies for all inputs of a LUT, and for all sizes of LUT up to the maximum LUT size allowed; e.g. the delay between the input of a LUT2 and its output is the same as the delay between the input of a LUT6 and its output.
- A LUT may take up a variable number of area units. This is constant for each size of LUT; e.g. a LUT4 may take up 1 unit of area, but a LUT5 may take up 2 units of area, but this applies for all LUT4s and LUT5s.

This is known as the "unit delay model", because each LUT uses one unit of delay.

From this view, the problem ABC has to solve is finding a mapping of the network to LUTs that has the lowest delay, and then optimising the mapping for size while maintaining this delay.

This approach has advantages:

- It is simple and easy to implement.
- Working with unit delays is fast to manipulate.
- It reflects *some* FPGA families, for example, the iCE40HX/LP fits the assumptions of the unit delay model quite well (almost all synchronous blocks, except for adders).

But this approach has drawbacks, too:

- The network of inputs and outputs with only LUTs means that a lot of combinational cells (multipliers and LUTRAM) are invisible to the unit delay model, meaning the critical path it optimises for is not necessarily the actual critical path.
- LUTs are implemented as multiplexer trees, so there is a delay caused by the result propagating through the remaining multiplexers. This means the assumption of delay being equal isn't true in physical hardware, and is proportionally larger for larger LUTs.
- Even synchronous blocks have arrival times (propagation delay between clock edge to output changing) and setup times (requirement for input to be stable before clock edge) which affect the delay of a path.

ABC9: the generalised delay model, realistic and flexible

ABC9 uses a more detailed and accurate model of an FPGA:

- An FPGA is made up of a network of inputs that connect through LUTs and combinational boxes to a network of outputs. These boxes have specified delays between inputs and outputs, and may have an associated network ("white boxes") or not ("black boxes"), but must be treated as a whole.
- Each LUT has a specified delay between an input and its output in arbitrary delay units, and this varies for all inputs of a LUT and for all sizes of LUT, but each size of LUT has the same associated delay; e.g. the delay between input A and output is different between a LUT2 and a LUT6, but is constant for all LUT6s.
- A LUT may take up a variable number of area units. This is constant for each size of LUT; e.g. a LUT4 may take up 1 unit of area, but a LUT5 may take up 2 units of area, but this applies for all LUT4s and LUT5s.

This is known as the "generalised delay model", because it has been generalised to arbitrary delay units. ABC9 doesn't actually care what units you use here, but the Yosys convention is picoseconds. Note the introduction of boxes as a concept. While the generalised delay model does not require boxes, they naturally fit into it to represent combinational delays. Even synchronous delays like arrival and setup can be emulated with combinational boxes that act as a delay. This is further extended to white boxes, where the mapper is able to see inside a box, and remove orphan boxes with no outputs, such as adders.

Again, ABC9 finds a mapping of the network to LUTs that has the lowest delay, and then minimises it to find the lowest area, but it has a lot more information to work with about the network.

The result here is that ABC9 can remove boxes (like adders) to reduce area, optimise better around those boxes, and also permute inputs to give the critical path the fastest inputs.

Todo

more about logic minimization & register balancing et al with ABC

Setting up a flow for ABC9

Much of the configuration comes from attributes and `specify` blocks in Verilog simulation models.

`specify` syntax

Since `specify` is a relatively obscure part of the Verilog standard, a quick guide to the syntax:

```
specify                             // begins a specify block
  (A => B) = 123;                   // simple combinational path from A to B with a delay␣
↪of 123.
  (A *> B) = 123;                   // simple combinational path from A to all bits of B␣
↪with a delay of 123 for all.
  if (FOO) (A => B) = 123;          // paths may apply under specific conditions.
  (posedge CLK => (Q : D)) = 123; // combinational path triggered on the positive edge␣
↪of CLK; used for clock-to-Q arrival paths.
  $setup(A, posedge CLK, 123);      // setup constraint for an input relative to a clock.
endspecify                          // ends a specify block
```

By convention, all delays in `specify` blocks are in integer picoseconds. Files containing `specify` blocks should be read with the `-specify` option to *`read_verilog`* so that they aren't skipped.

LUTs

LUTs need to be annotated with an `(* abc9_lut=N *)` attribute, where `N` is the relative area of that LUT model. For example, if an architecture can combine LUTs to produce larger LUTs, then the combined LUTs would have increasingly larger `N`. Conversely, if an architecture can split larger LUTs into smaller LUTs, then the smaller LUTs would have smaller `N`.

LUTs are generally specified with simple combinational paths from the LUT inputs to the LUT output.

DFFs

DFFs should be annotated with an `(* abc9_flop *)` attribute, however ABC9 has some specific requirements for this to be valid: - the DFF must initialise to zero (consider using `dfflegalize` to ensure this). - the DFF cannot have any asynchronous resets/sets (see the simplification idiom and the Boxes section for what to do here).

It is worth noting that in pure `abc9` mode, only the setup and arrival times are passed to ABC9 (specifically, they are modelled as buffers with the given delay). In `abc9 -dff`, the flop itself is passed to ABC9, permitting sequential optimisations.

Some vendors have universal DFF models which include async sets/resets even when they're unused. Therefore *the simplification idiom* exists to handle this: by using a `techmap` file to discover flops which have a constant driver to those asynchronous controls, they can be mapped into an intermediate, simplified flop which qualifies as an `(* abc9_flop *)`, ran through `abc9`, and then mapped back to the original flop. This is used in `synth_intel_alm` and `synth_quicklogic` for the PolarPro3.

DFFs are usually specified to have setup constraints against the clock on the input signals, and an arrival time for the `Q` output.

Boxes

A "box" is a purely-combinational piece of hard logic. If the logic is exposed to ABC9, it's a "whitebox", otherwise it's a "blackbox". Carry chains would be best implemented as whiteboxes, but a DSP would be best implemented as a blackbox (multipliers are too complex to easily work with). LUT RAMs can be implemented as whiteboxes too.

Boxes are arguably the biggest advantage that ABC9 has over ABC: by being aware of carry chains and DSPs, it avoids optimising for a path that isn't the actual critical path, while the generally-longer paths result in ABC9 being able to reduce design area by mapping other logic to slower cells with greater logic density.

3.1.9 Mapping to cell libraries

While much of this documentation focuses on the use of Yosys with FPGAs, it is also possible to map to cell libraries which can be used in designing ASICs. This section will cover a brief example project, available in the Yosys source code under `docs/source/code_examples/intro/`. The project contains a simple ASIC synthesis script (`counter.ys`), a digital design written in Verilog (`counter.v`), and a simple CMOS cell library (`mycells.lib`). Many of the early steps here are already covered in more detail in the *Synthesis starter* document.

> **Note**
>
> The `counter.ys` script includes the commands used to generate the images in this document. Code snippets in this document skip these commands; including line numbers to allow the reader to follow along with the source.
>
> To learn more about these commands, check out *A look at the show command*.

A simple counter

First, let's quickly look at the design:

Listing 3.31: `counter.v`

```
1  module counter (clk, rst, en, count);
2
3          input clk, rst, en;
4          output reg [1:0] count;
5
6          always @(posedge clk)
7                  if (rst)
8                          count <= 2'd0;
9                  else if (en)
10                         count <= count + 2'd1;
11
12 endmodule
```

This is a simple counter with reset and enable. If the reset signal, `rst`, is high then the counter will reset to 0. Otherwise, if the enable signal, `en`, is high then the `count` register will increment by 1 each rising edge of the clock, `clk`.

Loading the design

Listing 3.32: `counter.ys` - read design

```
1 # read design
2 read_verilog counter.v
3 hierarchy -check -top counter
```

Our circuit now looks like this:

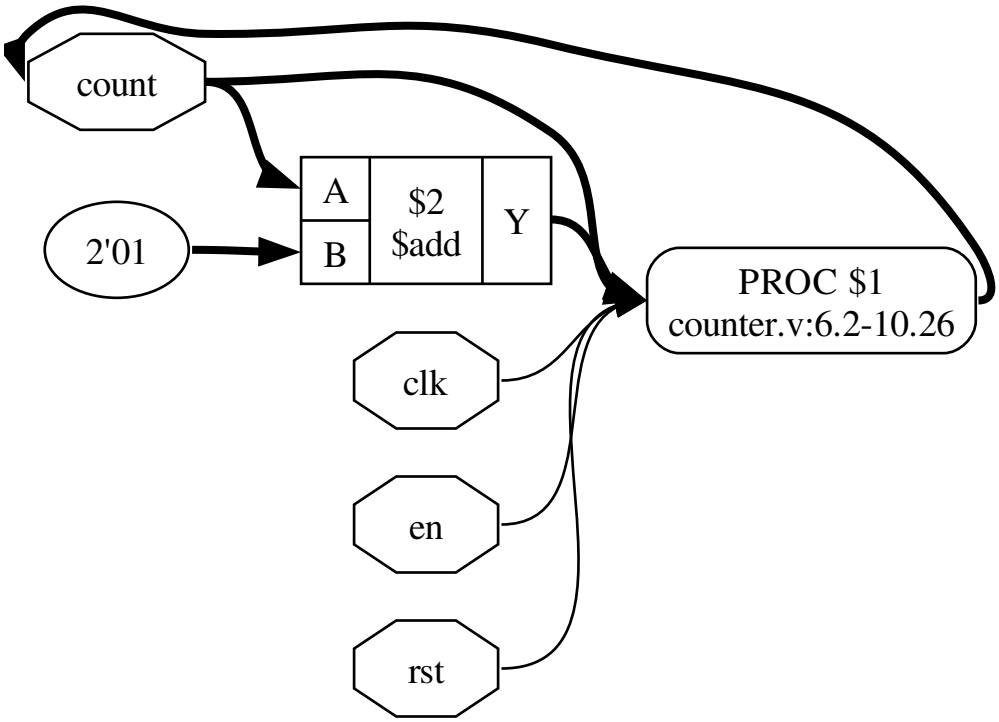


Fig. 3.7: `counter` after `hierarchy`

Coarse-grain representation

Listing 3.33: `counter.ys` - the high-level stuff

```
# the high-level stuff
proc; opt
memory; opt
fsm; opt
```

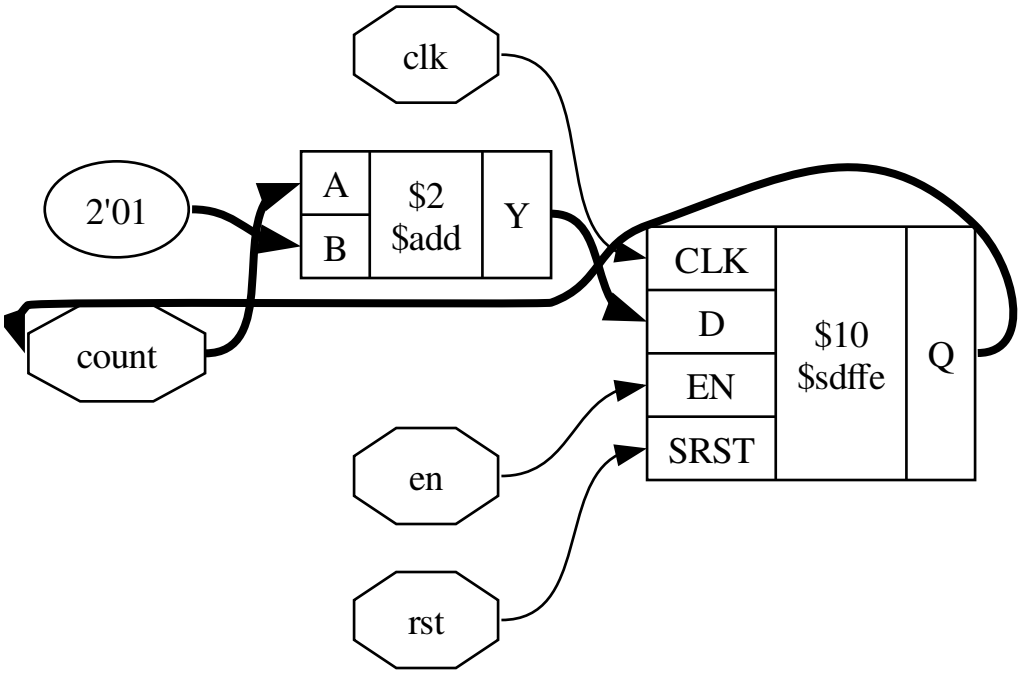


Fig. 3.8: Coarse-grain representation of the `counter` module

Logic gate mapping

Listing 3.34: `counter.ys` - mapping to internal cell library

```
# mapping to internal cell library
techmap; opt
```

Mapping to hardware

For this example, we are using a Liberty file to describe a cell library which our internal cell library will be mapped to:

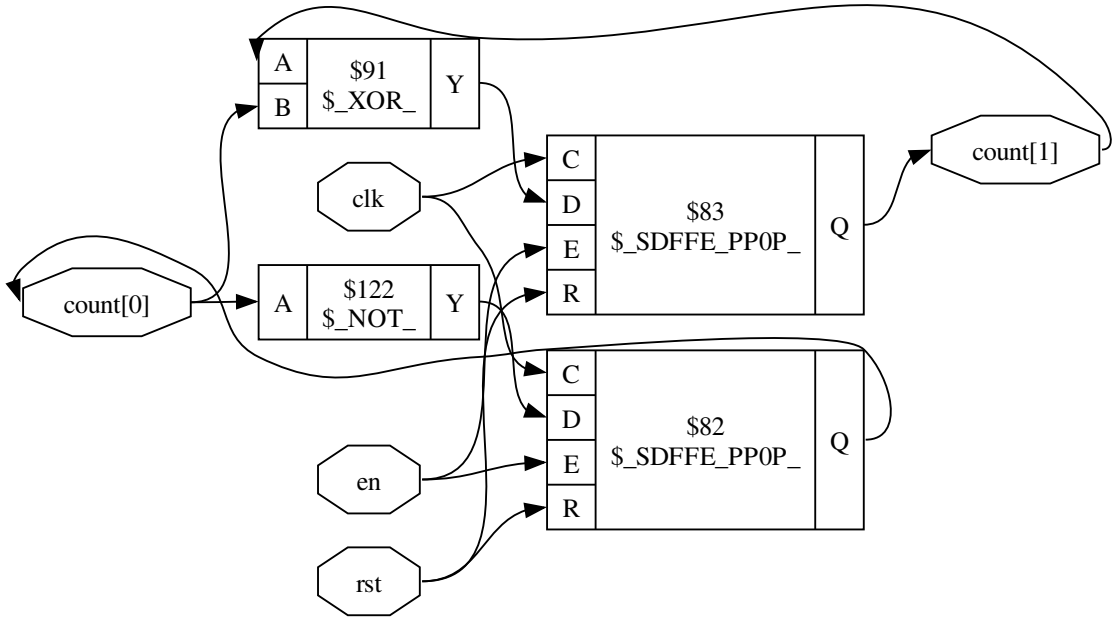


Fig. 3.9: `counter` after `techmap`

Todo

find a Liberty pygments style?

Listing 3.35: `mycells.lib`

```
library(demo) {
  cell(BUF) {
    area: 6;
    pin(A) { direction: input; }
    pin(Y) { direction: output;
             function: "A"; }
  }
  cell(NOT) {
    area: 3;
    pin(A) { direction: input; }
    pin(Y) { direction: output;
             function: "A'"; }
  }
  cell(NAND) {
    area: 4;
    pin(A) { direction: input; }
    pin(B) { direction: input; }
    pin(Y) { direction: output;
```

(continues on next page)

(continued from previous page)

```
            function: "(A*B)'"; }
  }
  cell(NOR) {
    area: 4;
    pin(A) { direction: input; }
    pin(B) { direction: input; }
    pin(Y) { direction: output;
             function: "(A+B)'"; }
  }
  cell(DFF) {
    area: 18;
    ff(IQ, IQN) { clocked_on: C;
                  next_state: D; }
    pin(C) { direction: input;
                clock: true; }
    pin(D) { direction: input; }
    pin(Q) { direction: output;
             function: "IQ"; }
  }
}
```

Recall that the Yosys built-in logic gate types are *$_NOT_* , *$_AND_* , *$_OR_* , *$_XOR_* , and *$_MUX_* with an assortment of dff memory types. *mycells.lib* defines our target cells as `BUF`, `NOT`, `NAND`, `NOR`, and `DFF`. Mapping between these is performed with the commands `dfflibmap` and `abc` as follows:

Listing 3.36: `counter.ys` - mapping to hardware

```
dfflibmap -liberty mycells.lib

# mapping logic to mycells.lib
abc -liberty mycells.lib

# cleanup
clean
```

The final version of our `counter` module looks like this:

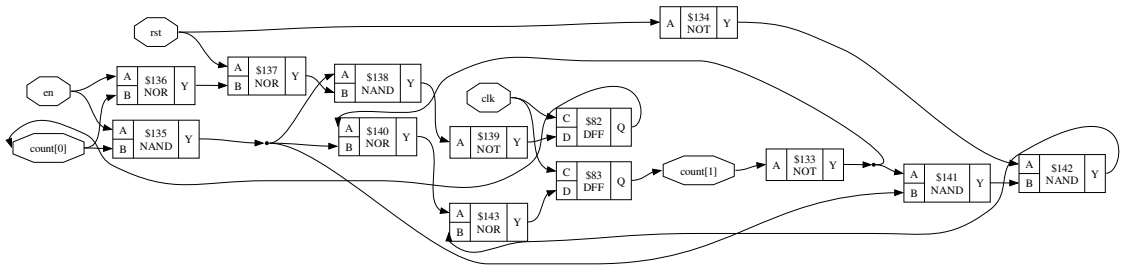


Fig. 3.10: `counter` after hardware cell mapping

Before finally being output as a verilog file with *write_verilog*, which can then be loaded into another

tool:

Listing 3.37: `counter.ys` - write synthesized design

```
30 # write synthesized design
31 write_verilog synth.v
```

3.2 More scripting

Todo

brief overview for the more scripting index

Todo

troubleshooting document(?)

3.2.1 Loading a design

Input files on the command line

- guesses frontend based on file extension
  - `.v` -> `read -vlog2k`
  - `.sv` -> `read -sv`
  - `.vhd` and `.vhdl` -> `read -vhdl`
  - `.blif` and `.eblif` -> *`read_blif`*
  - `.json` -> *`read_json`*
  - `.il` -> *`read_rtlil`* (direct textual representation of Yosys internal state)
- command line also supports
  - `.ys` -> `script`
  - `.tcl` -> `tcl`
  - `-` -> reads stdin and treats it as a script

The `read` command

- standard method of loading designs
- also for defining macros and include directories
- uses `verific` command if available
  - `-verific` and `-noverific` options to enforce with/without Verific
  - check `help read` for more about the options available and the filetypes supported
  - elaborate designs with `verific -import [options] <top>` (or use `hierarchy`)
- fallback to *`read_verilog`* with `-defer` option
  - does not compile design until `hierarchy` command as discussed in *Synthesis starter*

  - more similar to `verific` behaviour
- `read -define` et al mapped to `verific` or `verilog_defines`
- similarly, `read -incdir` et al mapped to `verific` or `verilog_defaults`

Note

The Verific frontend for Yosys, which provides the `verific` command, requires Yosys to be built with Verific. For full functionality, custom modifications to the Verific source code from YosysHQ are required, but limited useability can be achieved with some stock Verific builds. Check *Compiling with Verific library* for more.

Yosys frontends

- *Reading input files*
- typically start with `read_`
- built-in support for heredocs
  - in-line code with `<<EOT`
  - can use any eot marker, but EOT (End-of-Transmission) and EOF (End-of-File) are most common
- built-in support for reading multiple files in the same command
  - executed as multiple successive calls to the frontend
- compatible with `-f` command line option, e.g. `yosys -f verilog design.txt` will use the *`read_verilog`* frontend with the input file `design.txt`
- `verific` and `read` commands are technically not 'Frontends', but their behaviour is kept in sync

Note

'Frontend' here means that the command is implemented as a sub-class of `RTLIL::Frontend`, as opposed to the usual `RTLIL::Pass`.

Todo

link note to as-yet non-existent section on `RTLIL::Pass` under *Working with the Yosys codebase*

The *`read_verilog`* command

- *read_verilog - read modules from Verilog file*; also
  - `verilog_defaults`,
  - `verilog_defines`, and
  - *read_verilog_file_list - parse a Verilog file list*
- supports most of Verilog-2005
- limited support for SystemVerilog
- some non-standard features/extensions for enabling formal verification

- please do not rely on *read_verilog* for syntax checking
  - recommend using a simulator (for example Icarus Verilog) or linting with another tool (such as verilator) first

Todo

figure out this example code block

```
read_verilog file1.v
read_verilog -I include_dir -D enable_foo -D WIDTH=12 file2.v
read_verilog -lib cell_library.v

verilog_defaults -add -I include_dir
read_verilog file3.v
read_verilog file4.v
verilog_defaults -clear

verilog_defaults -push
verilog_defaults -add -I include_dir
read_verilog file5.v
read_verilog file6.v
verilog_defaults -pop
```

Other built-in `read_*` commands

- *read_rtlil - read modules from RTLIL file*
- *read_aiger - read AIGER file*
- *read_blif - read BLIF file*
- *read_json - read JSON file*
- *read_liberty - read cells from liberty file*
- *read_xaiger2 - (experimental) read XAIGER file*

Todo

does *write_file* count?

Externally maintained plugins

- GHDL plugin for VHDL (check `help ghdl`)
- yosys-slang plugin for more comprehensive SystemVerilog support (check `help read_slang`)
  - yosys-slang is implemented as a '*Frontend*,' with all the built-in support that entails
- both plugins above are included in OSS CAD Suite
- Synlig, which uses Surelog to provide SystemVerilog support
  - also implemented as a '*Frontend*'

3.2.2 Selections

The selection framework

Todo

reduce overlap with *Scripting in Yosys* select section

The *select* command can be used to create a selection for subsequent commands. For example:

```
select foobar         # select the module foobar
delete                # delete selected objects
```

Normally the *select* command overwrites a previous selection. The commands `select -add` and `select -del` can be used to add or remove objects from the current selection.

The command `select -clear` can be used to reset the selection to the default, which is a complete selection of everything in the current module.

This selection framework can also be used directly in many other commands. Whenever a command has `[selection]` as last argument in its usage help, this means that it will use the engine behind the *select* command to evaluate additional arguments and use the resulting selection instead of the selection created by the last *select* command.

For example, the command `delete` will delete everything in the current selection; while `delete foobar` will only delete the module foobar. If no *select* command has been made, then the "current selection" will be the whole design.

Note

Many of the examples on this page make use of the *show* command to visually demonstrate the effect of selections. For a more detailed look at this command, refer to *A look at the show command*.

How to make a selection

Selection by object name

The easiest way to select objects is by object name. This is usually only done in synthesis scripts that are hand-tailored for a specific design.

```
select foobar         # select module foobar
select foo*           # select all modules whose names start with foo
select foo*/bar*      # select all objects matching bar* from modules matching foo*
select */clk          # select objects named clk from all modules
```

Module and design context

Commands can be executed in *module/* or *design/* context. Until now, all commands have been executed in design context. The *cd* command can be used to switch to module context.

In module context, all commands only effect the active module. Objects in the module are selected without the `<module_name>/` prefix. For example:

```
cd foo                # switch to module foo
delete bar            # delete object foo/bar

cd mycpu              # switch to module mycpu
dump reg_*            # print details on all objects whose names start with reg_

cd ..                 # switch back to design
```

Note: Most synthesis scripts never switch to module context. But it is a very powerful tool which we explore more in *Interactive design investigation*.

Selecting by object property or type

Special patterns can be used to select by object property or type. For example:

- select all wires whose names start with `reg_`: `select w:reg_*`
- select all objects with the attribute `foobar` set: `select a:foobar`
- select all objects with the attribute `foobar` set to 42: `select a:foobar=42`
- select all modules with the attribute `blabla` set: `select A:blabla`
- select all *`$add`* cells from the module foo: `select foo/t:$add`

A complete list of pattern expressions can be found in *select - modify and view the list of selected objects*.

Operations on selections

Combining selections

The *`select`* command is actually much more powerful than it might seem at first glance. When it is called with multiple arguments, each argument is evaluated and pushed separately on a stack. After all arguments have been processed it simply creates the union of all elements on the stack. So `select t:$add a:foo` will select all *`$add`* cells and all objects with the `foo` attribute set:

Listing 3.38: Test module for operations on selections

```
module foobaraddsub(a, b, c, d, fa, fs, ba, bs);
  input [7:0] a, b, c, d;
  output [7:0] fa, fs, ba, bs;
  assign fa = a + (* foo *) b;
  assign fs = a - (* foo *) b;
  assign ba = c + (* bar *) d;
  assign bs = c - (* bar *) d;
endmodule
```

Listing 3.39: Output for command `select t:$add a:foo -list` on Listing 3.38

```
yosys> select t:$add a:foo -list
foobaraddsub/$add$foobaraddsub.v:6$3
foobaraddsub/$sub$foobaraddsub.v:5$2
foobaraddsub/$add$foobaraddsub.v:4$1
```

In many cases simply adding more and more stuff to the selection is an ineffective way of selecting the interesting part of the design. Special arguments can be used to combine the elements on the stack. For

example the `%i` arguments pops the last two elements from the stack, intersects them, and pushes the result back on the stack. So `select t:$add a:foo %i` will select all *`$add`* cells that have the `foo` attribute set:

Listing 3.40: Output for command `select t:$add a:foo %i -list` on Listing 3.38

```
yosys> select t:$add a:foo %i -list
foobaraddsub/$add$foobaraddsub.v:4$1
```

Some of the special `%`-codes:

- `%u`: union of top two elements on stack – pop 2, push 1
- `%d`: difference of top two elements on stack – pop 2, push 1
- `%i`: intersection of top two elements on stack – pop 2, push 1
- `%n`: inverse of top element on stack – pop 1, push 1

See *select - modify and view the list of selected objects* for the full list.

Expanding selections

Listing 3.41 uses the Yosys non-standard `{... *}` syntax to set the attribute `sumstuff` on all cells generated by the first assign statement. (This works on arbitrary large blocks of Verilog code and can be used to mark portions of code for analysis.)

Listing 3.41: Another test module for operations on selections

```
module sumprod(a, b, c, sum, prod);

  input [7:0] a, b, c;
  output [7:0] sum, prod;

  {* sumstuff *}
  assign sum = a + b + c;
  {* *}

  assign prod = a * b * c;

endmodule
```

Selecting `a:sumstuff` in this module will yield the following circuit diagram:

As only the cells themselves are selected, but not the temporary wire `$1_Y`, the two adders are shown as two disjunct parts. This can be very useful for global signals like clock and reset signals: just unselect them using a command such as `select -del clk rst` and each cell using them will get its own net label.

In this case however we would like to see the cells connected properly. This can be achieved using the `%x` action, that broadens the selection, i.e. for each selected wire it selects all cells connected to the wire and vice versa. So `show a:sumstuff %x` yields the diagram shown in Fig. 3.12:

Selecting logic cones

Fig. 3.12 shows what is called the `input cone` of `sum`, i.e. all cells and signals that are used to generate the signal `sum`. The `%ci` action can be used to select the input cones of all object in the top selection in the stack maintained by the *`select`* command.

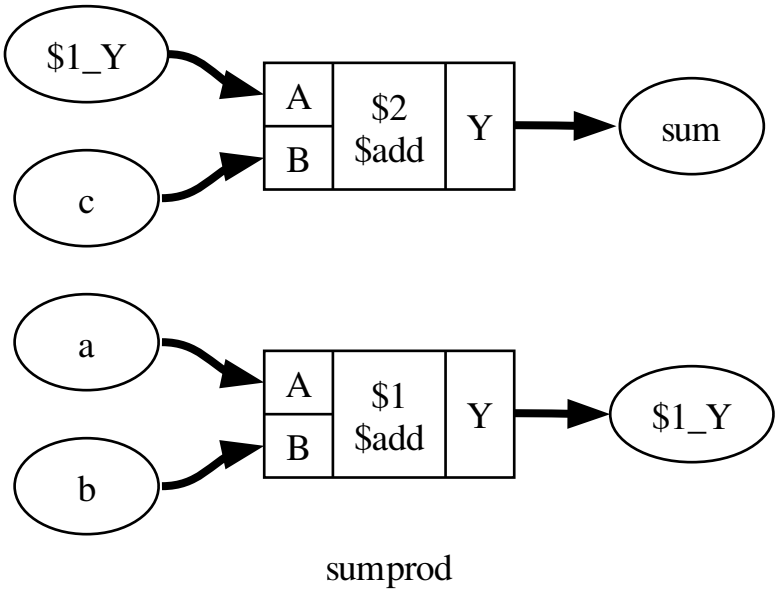


Fig. 3.11: Output of `show a:sumstuff` on Listing 3.41

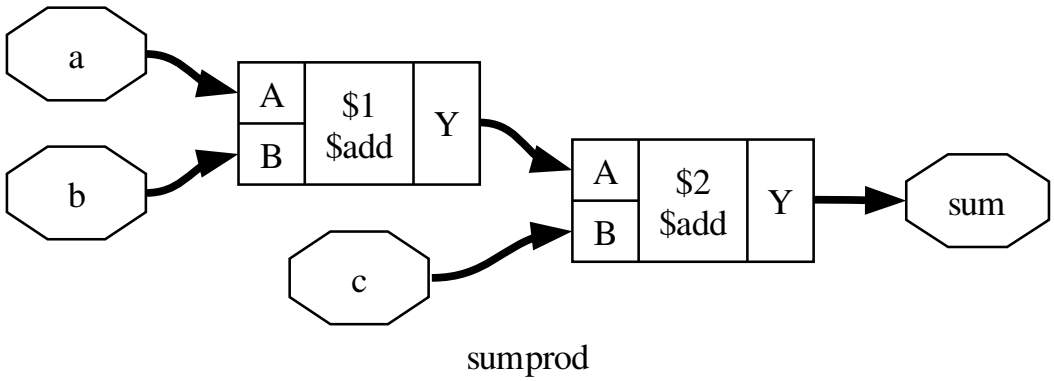


Fig. 3.12: Output of `show a:sumstuff %x` on Listing 3.41

As with the `%x` action, these commands broaden the selection by one "step". But this time the operation only works against the direction of data flow. That means, wires only select cells via output ports and cells only select wires via input ports.

The following sequence of diagrams demonstrates this step-wise expansion:

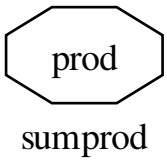


Fig. 3.13: Output of `show prod` on Listing 3.41

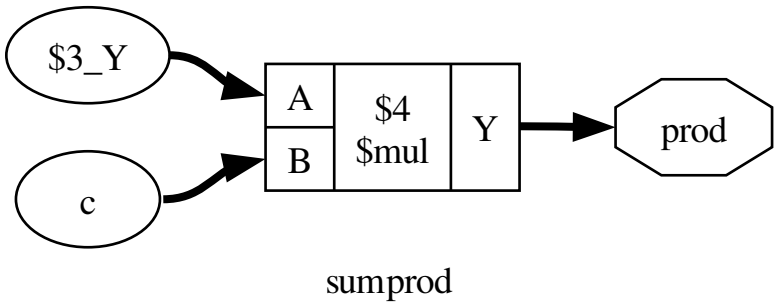


Fig. 3.14: Output of `show prod %ci` on Listing 3.41

Notice the subtle difference between `show prod %ci` and `show prod %ci %ci`. Both images show the *`$mul`* cell driven by some inputs `$3_Y` and `c`. However it is not until the second image, having called `%ci` the second time, that *`show`* is able to distinguish between `$3_Y` being a wire and `c` being an input. We can see this better with the `dump` command instead:

Listing 3.42: Output of `dump prod %ci`

```
attribute \src "sumprod.v:4.21-4.25"
wire width 8 output 5 \prod

attribute \src "sumprod.v:10.17-10.26"
cell $mul $mul$sumprod.v:10$4
  parameter \Y_WIDTH 8
  parameter \B_WIDTH 8
```

(continues on next page)

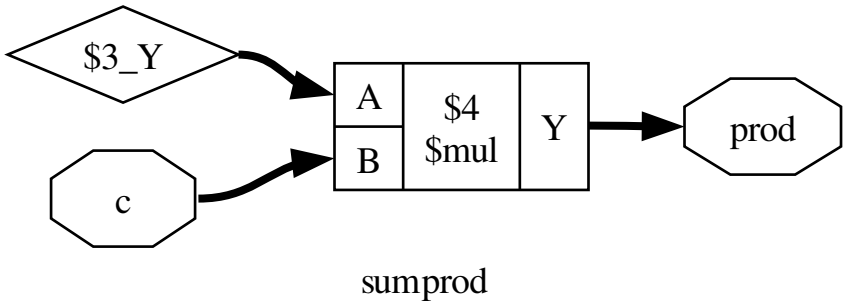


Fig. 3.15: Output of `show prod %ci %ci` on Listing 3.41

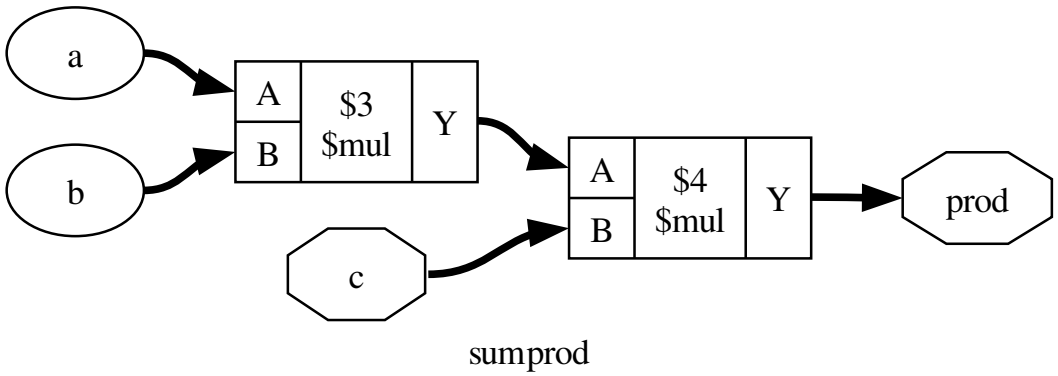


Fig. 3.16: Output of `show prod %ci %ci %ci` on Listing 3.41

(continued from previous page)

```
    parameter \B_SIGNED 0
    parameter \A_WIDTH 8
    parameter \A_SIGNED 0
    connect \Y \prod
    connect \B \c
    connect \A $mul$sumprod.v:10$3_Y
  end
```

Listing 3.43: Output of `dump prod %ci %ci`

```
  attribute \src "sumprod.v:4.21-4.25"
  wire width 8 output 5 \prod

  attribute \src "sumprod.v:3.21-3.22"
  wire width 8 input 3 \c

  attribute \src "sumprod.v:10.17-10.22"
  wire width 8 $mul$sumprod.v:10$3_Y

  attribute \src "sumprod.v:10.17-10.26"
  cell $mul $mul$sumprod.v:10$4
    parameter \Y_WIDTH 8
    parameter \B_WIDTH 8
    parameter \B_SIGNED 0
    parameter \A_WIDTH 8
    parameter \A_SIGNED 0
    connect \Y \prod
    connect \B \c
    connect \A $mul$sumprod.v:10$3_Y
  end
```

When selecting many levels of logic, repeating `%ci` over and over again can be a bit dull. So there is a shortcut for that: the number of iterations can be appended to the action. So for example the action `%ci3` is identical to performing the `%ci` action three times.

The action `%ci*` performs the `%ci` action over and over again until it has no effect anymore.

Advanced logic cone selection

In most cases there are certain cell types and/or ports that should not be considered for the `%ci` action, or we only want to follow certain cell types and/or ports. This can be achieved using additional patterns that can be appended to the `%ci` action.

Lets consider Listing 3.44. It serves no purpose other than being a non-trivial circuit for demonstrating some of the advanced Yosys features. This code is available in `docs/source/code_examples/selections` of the Yosys source repository.

Listing 3.44: `memdemo.v`

```
module memdemo(clk, d, y);

input clk;
input [3:0] d;
```

(continues on next page)

(continued from previous page)

```
output reg [3:0] y;

integer i;
reg [1:0] s1, s2;
reg [3:0] mem [0:3];

always @(posedge clk) begin
    for (i = 0; i < 4; i = i+1)
        mem[i] <= mem[(i+1) % 4] + mem[(i+2) % 4];
    { s2, s1 } = d ? { s1, s2 } ^ d : 4'b0;
    mem[s1] <= d;
    y <= mem[s2];
end

endmodule
```

The script `memdemo.ys` is used to generate the images included here. Let's look at the first section:

Listing 3.45: Synthesizing *memdemo.v*

```
read_verilog memdemo.v
prep -top memdemo; memory; opt
```

This loads Listing 3.44 and synthesizes the included module. Note that this code can be copied and run directly in a Yosys command line session, provided `memdemo.v` is in the same directory. We can now change to the `memdemo` module with `cd memdemo`, and call *`show`* to see the diagram in Fig. 3.17.

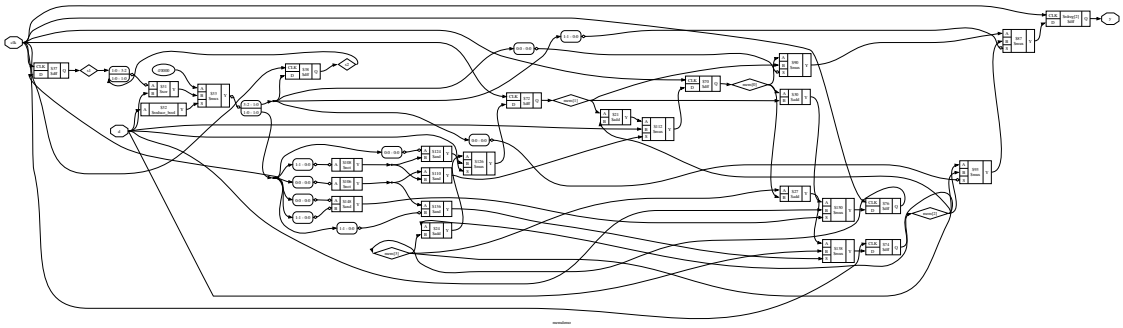


Fig. 3.17: Complete circuit diagram for the design shown in Listing 3.44

There's a lot going on there, but maybe we are only interested in the tree of multiplexers that select the output value. Let's start by just showing the output signal, `y`, and its immediate predecessors. Remember *Selecting logic cones* from above, we can use `show y %ci2`:

From this we would learn that `y` is driven by a *`$dff`* cell, that `y` is connected to the output port `Q`, that the `clk` signal goes into the `CLK` input port of the cell, and that the data comes from an auto-generated wire into the input `D` of the flip-flop cell (indicated by the `$` at the start of the name). Let's go a bit further now and try `show y %ci5`:

That's starting to get a bit messy, so maybe we want to ignore the mux select inputs. To add a pattern we

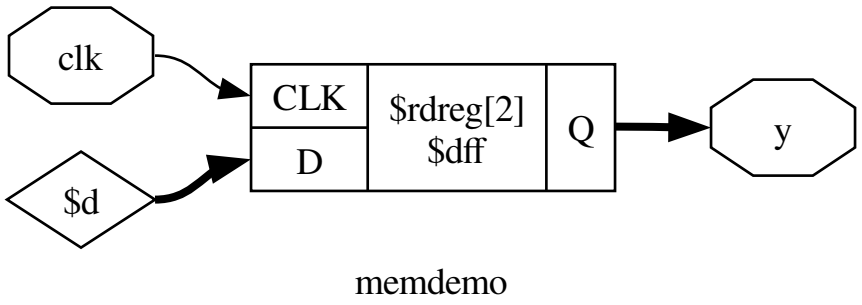


Fig. 3.18: Output of `show y %ci2`

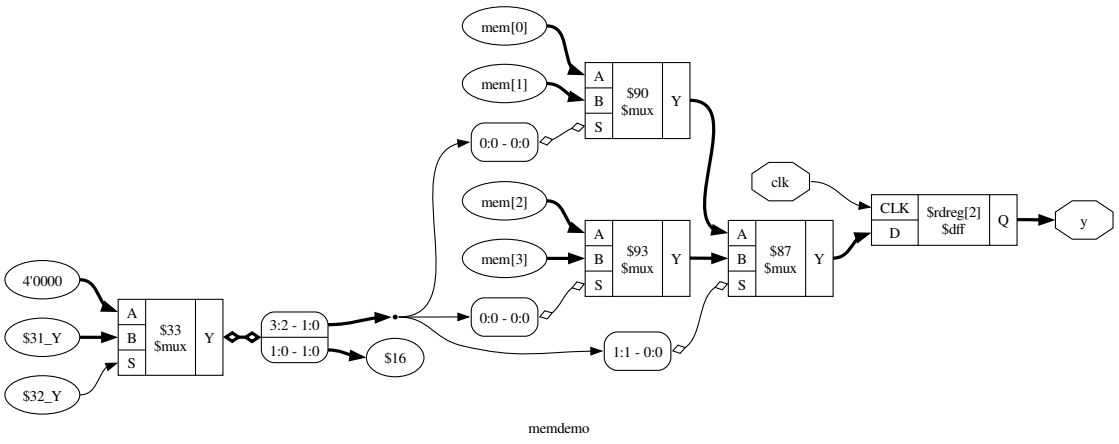


Fig. 3.19: Output of `show y %ci5`

add a colon followed by the pattern to the `%ci` action. The pattern itself starts with `-` or `+`, indicating if it is an include or exclude pattern, followed by an optional comma separated list of cell types, followed by an optional comma separated list of port names in square brackets. In this case, we want to exclude the `S` port of the *`$mux`* cell type with `show y %ci5:-$mux[S]`:

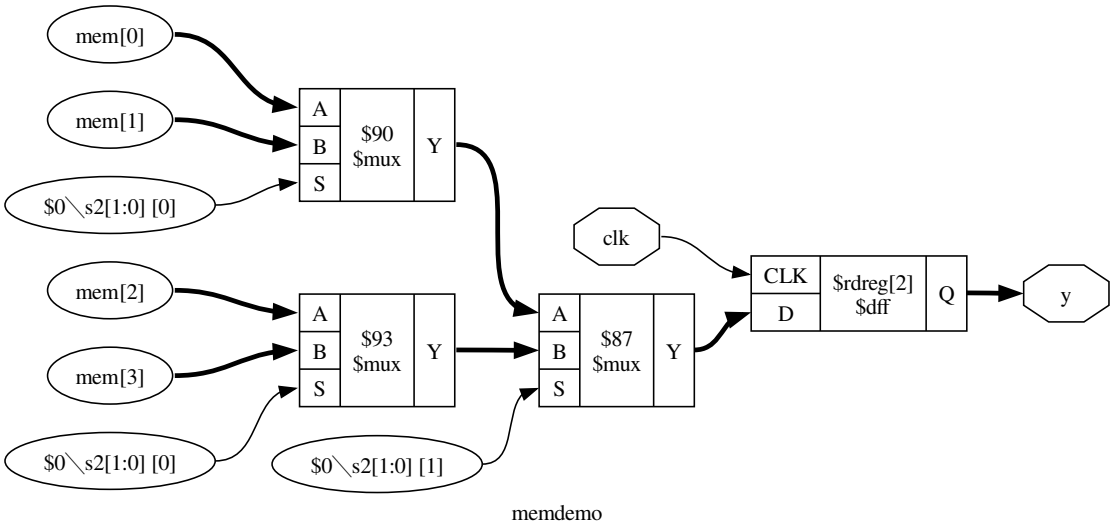


Fig. 3.20: Output of `show y %ci5:-$mux[S]`

We could use a command such as `show y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff` in which the first `%ci` jumps over the initial d-type flip-flop and the 2nd action selects the entire input cone without going over multiplexer select inputs and flip-flop cells:

Or we could use `show y %ci*:-[CLK,S]:+$dff:+$mux` instead, following the input cone all the way but only following *`$dff`* and *`$mux`* cells, and ignoring any ports named `CLK` or `S`:

Todo

pending discussion on whether rule ordering is a bug or a feature

Similar to `%ci` exists an action `%co` to select output cones that accepts the same syntax for pattern and repetition. The `%x` action mentioned previously also accepts this advanced syntax.

These actions for traversing the circuit graph, combined with the actions for boolean operations such as intersection (`%i`) and difference (`%d`) are powerful tools for extracting the relevant portions of the circuit under investigation.

Again, see *select - modify and view the list of selected objects* for full documentation of these expressions.

Incremental selection

Sometimes a selection can most easily be described by a series of add/delete operations. As mentioned previously, the commands `select -add` and `select -del` respectively add or remove objects from the current selection instead of overwriting it.

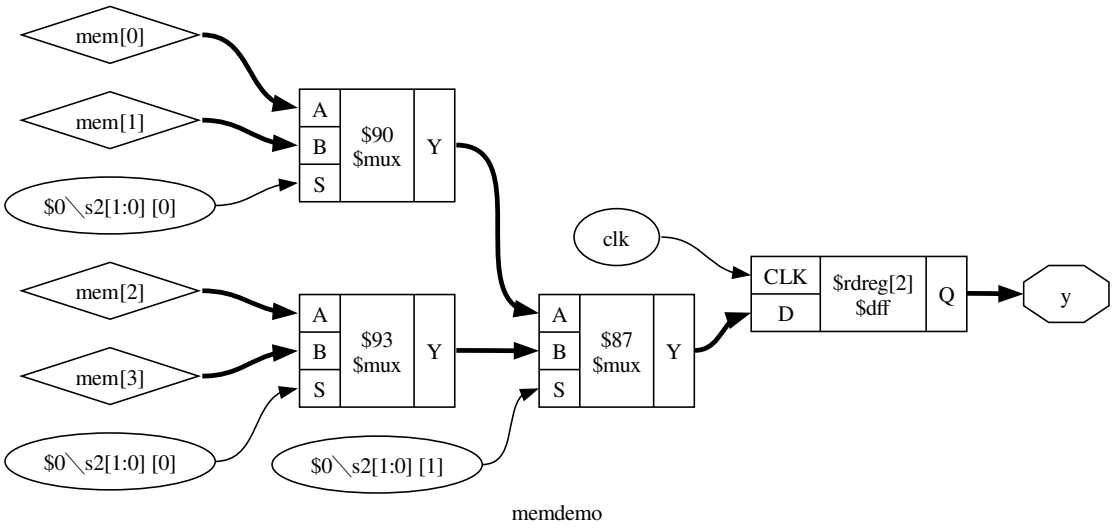


Fig. 3.21: Output of `show y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff`

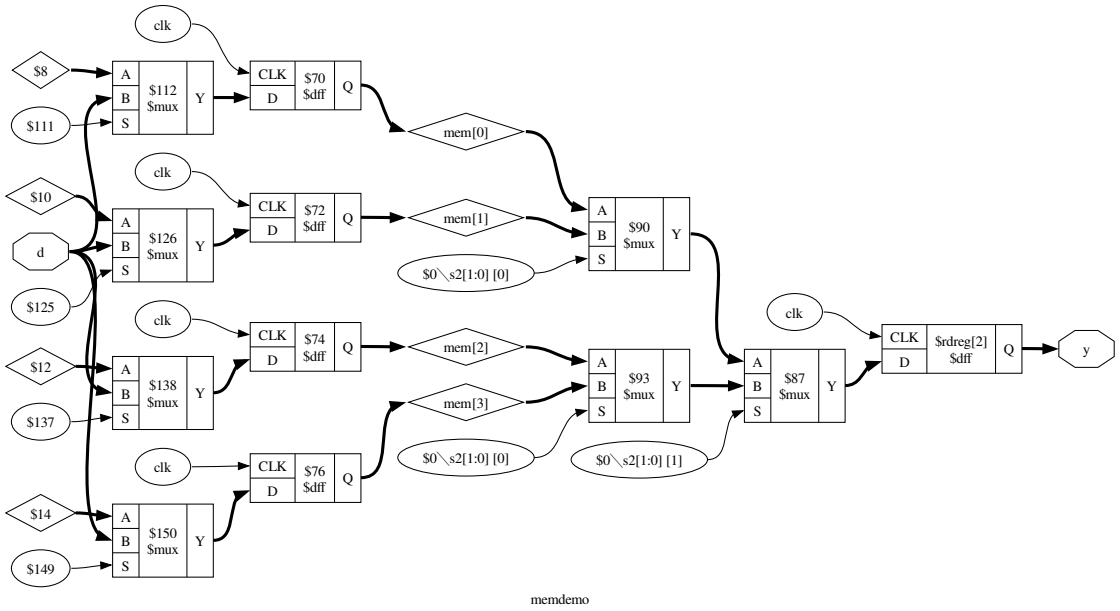


Fig. 3.22: Output of `show y %ci*:-[CLK,S]:+$dff,$mux`

```
select -none            # start with an empty selection
select -add reg_*       # select a bunch of objects
select -del reg_42      # but not this one
select -add state %ci   # and add more stuff
```

Within a select expression the token % can be used to push the previous selection on the stack.

```
select t:$add t:$sub    # select all $add and $sub cells
select % %ci % %d       # select only the input wires to those cells
```

Storing and recalling selections

Todo

reflow for not presentation

The current selection can be stored in memory with the command `select -set <name>`. It can later be recalled using `select @<name>`. In fact, the `@<name>` expression pushes the stored selection on the stack maintained by the *select* command. So for example `select @foo @bar %i` will select the intersection between the stored selections `foo` and `bar`.

In larger investigation efforts it is highly recommended to maintain a script that sets up relevant selections, so they can easily be recalled, for example when Yosys needs to be re-run after a design or source code change.

The `history` command can be used to list all recent interactive commands. This feature can be useful for creating such a script from the commands used in an interactive session.

Remember that select expressions can also be used directly as arguments to most commands. Some commands also accept a single select argument to some options. In those cases selection variables must be used to capture more complex selections.

Example code from docs/source/code_examples/selections:

Listing 3.46: `select.v`

```
module test(clk, s, a, y);
    input clk, s;
    input [15:0] a;
    output [15:0] y;
    reg [15:0] b, c;

    always @(posedge clk) begin
        b <= a;
        c <= b;
    end

    wire [15:0] state_a = (a ^ b) + c;
    wire [15:0] state_b = (a ^ b) - c;
    assign y = !s ? state_a : state_b;
endmodule
```

Listing 3.47: `select.ys`

```
read_verilog select.v
prep -top test

cd test
select -set cone_a state_a %ci*:-$dff
select -set cone_b state_b %ci*:-$dff
select -set cone_ab @cone_a @cone_b %i
show -prefix select -format dot -notitle \
     -color red @cone_ab -color magenta @cone_a \
     -color blue @cone_b
```

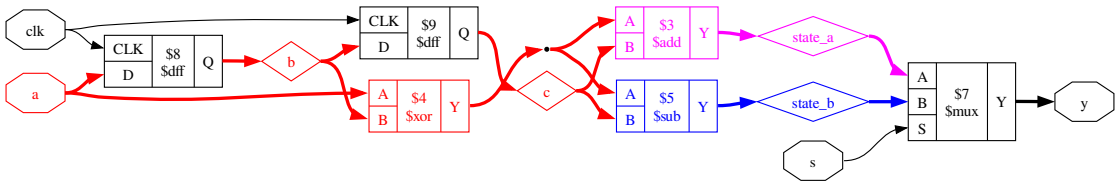


Fig. 3.23: Circuit diagram produced by Listing 3.47

3.2.3 Interactive design investigation

Todo

interactive design opening text

A look at the show command

Todo

merge into *Scripting in Yosys* show section

This section explores the *show* command and explains the symbols used in the circuit diagrams generated by it. The code used is included in the Yosys code base under `docs/source/code_examples/show`.

A simple circuit

example.v below provides the Verilog code for a simple circuit which we will use to demonstrate the usage of *show* in a simple setting.

Listing 3.48: `example.v`

```
module example(input clk, a, b, c,
               output reg [1:0] y);
```

(continues on next page)

(continued from previous page)

```
    always @(posedge clk)
        if (c)
            y <= c ? a + b : 2'd0;
endmodule
```

The Yosys synthesis script we will be running is included as Listing 3.49. Note that *show* is called with the `-pause` option, that halts execution of the Yosys script until the user presses the Enter key. Using `show -pause` also allows the user to enter an interactive shell to further investigate the circuit before continuing synthesis.

Listing 3.49: `example_show.ys`

```
read_verilog example.v
show -pause # first
proc
show -pause # second
opt
show -pause # third
```

This script, when executed, will show the design after each of the three synthesis commands. We will now look at each of these diagrams and explain what is shown.

Note

The images uses in this document are generated from the `example.ys` file, rather than `example_show.ys`. `example.ys` outputs the schematics as `.dot` files rather than displaying them directly. You can view these images yourself by running `yosys example.ys` and then `xdot example_first.dot` etc.

The first output shows the design directly after being read by the Verilog front-end. Input and output ports are displayed as octagonal shapes. Cells are displayed as rectangles with inputs on the left and outputs on the right side. The cell labels are two lines long: The first line contains a unique identifier for the cell and the second line contains the cell type. Internal cell types are prefixed with a dollar sign. For more details on the internal cell library, see *Internal cell library*.

Constants are shown as ellipses with the constant value as label. The syntax `<bit_width>'<bits>` is used for constants that are not 32-bit wide and/or contain bits that are not 0 or 1 (i.e. `x` or `z`). Ordinary 32-bit constants are written using decimal numbers.

Single-bit signals are shown as thin arrows pointing from the driver to the load. Signals that are multiple bits wide are shown as thick arrows.

Finally *processes* are shown in boxes with round corners. Processes are Yosys' internal representation of the decision-trees and synchronization events modelled in a Verilog `always`-block. The label reads `PROC` followed by a unique identifier in the first line and contains the source code location of the original `always`-block in the second line. Note how the multiplexer from the `?:`-expression is represented as a *$mux* cell but the multiplexer from the `if`-statement is yet still hidden within the process.

The *proc* command transforms the process from the first diagram into a multiplexer and a d-type flip-flop, which brings us to the second diagram:

The Rhombus shape to the right is a dangling wire. (Wire nodes are only shown if they are dangling or have "public" names, for example names assigned from the Verilog input.) Also note that the design now contains two instances of a `BUF`-node. These are artefacts left behind by the *proc* command. It is quite usual to see such artefacts after calling commands that perform changes in the design, as most commands

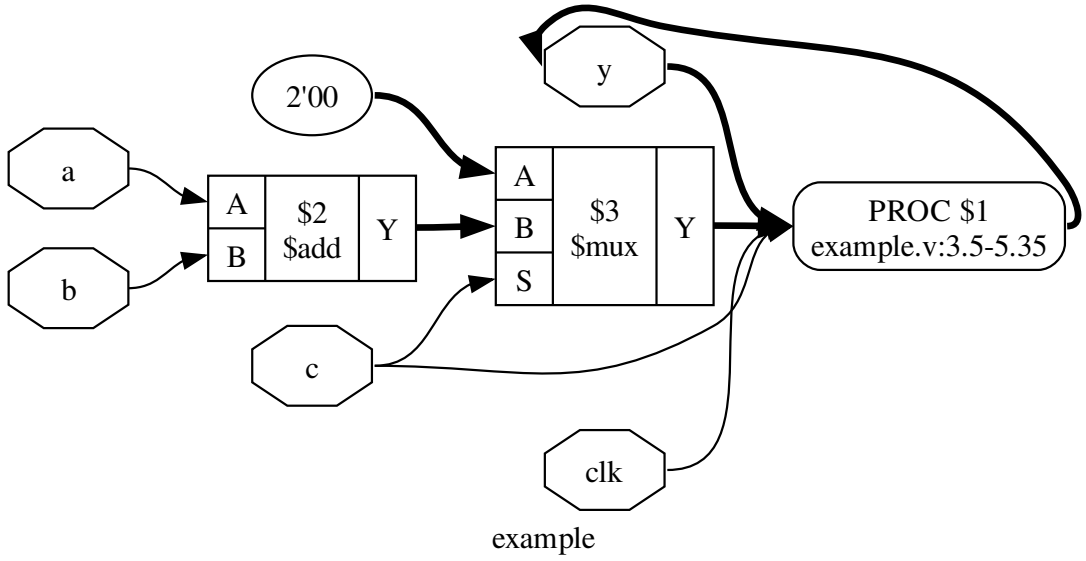


Fig. 3.24: Output of the first *show* command in Listing 3.49

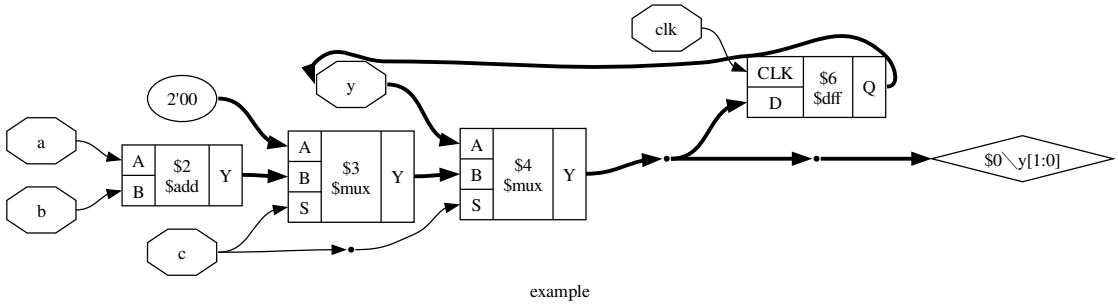


Fig. 3.25: Output of the second *show* command in Listing 3.49

only care about doing the transformation in the least complicated way, not about cleaning up after them. The next call to *clean* (or *opt*, which includes *clean* as one of its operations) will clean up these artefacts. This operation is so common in Yosys scripts that it can simply be abbreviated with the ;; token, which doubles as separator for commands. Unless one wants to specifically analyze this artefacts left behind some operations, it is therefore recommended to always call *clean* before calling *show*.

In this script we directly call *opt* as the next step, which finally leads us to the third diagram:

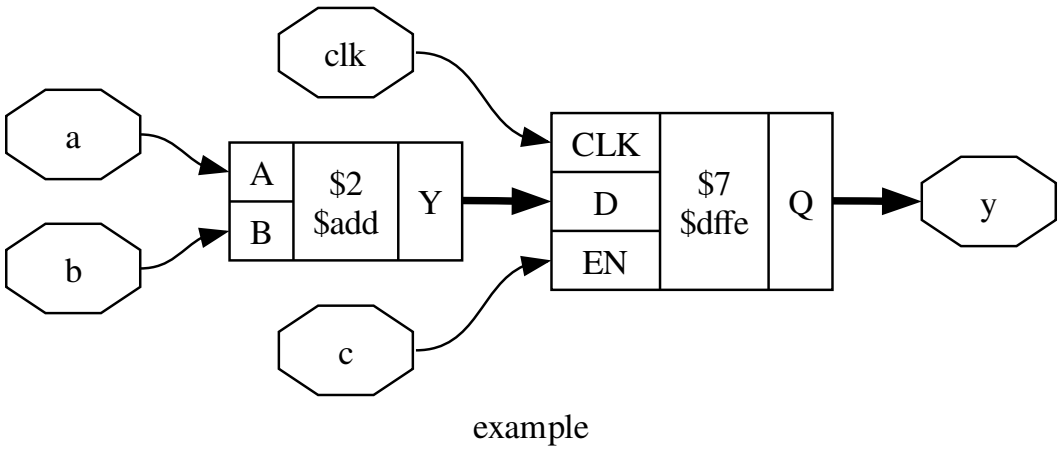


Fig. 3.26: Output of the third *show* command in *example_show.ys*

Here we see that the *opt* command not only has removed the artifacts left behind by *proc*, but also determined correctly that it can remove the first *$mux* cell without changing the behavior of the circuit.

Break-out boxes for signal vectors

The code listing below shows a simple circuit which uses a lot of spliced signal accesses.

Listing 3.50: splice.v

```
module splice_demo(a, b, c, d, e, f, x, y);

input [1:0] a, b, c, d, e, f;
output [1:0] x;
assign x = {a[0], a[1]};

output [11:0] y;
assign {y[11:4], y[1:0], y[3:2]} =
                {a, b, -{c, d}, ~{e, f}};

endmodule
```

Notice how the output for this circuit from the *show* command (Fig. 3.27) appears quite complex. This is an unfortunate side effect of the way Yosys handles signal vectors (aka. multi-bit wires or buses) as native

objects. While this provides great advantages when analyzing circuits that operate on wide integers, it also introduces some additional complexity when the individual bits of of a signal vector are accessed.

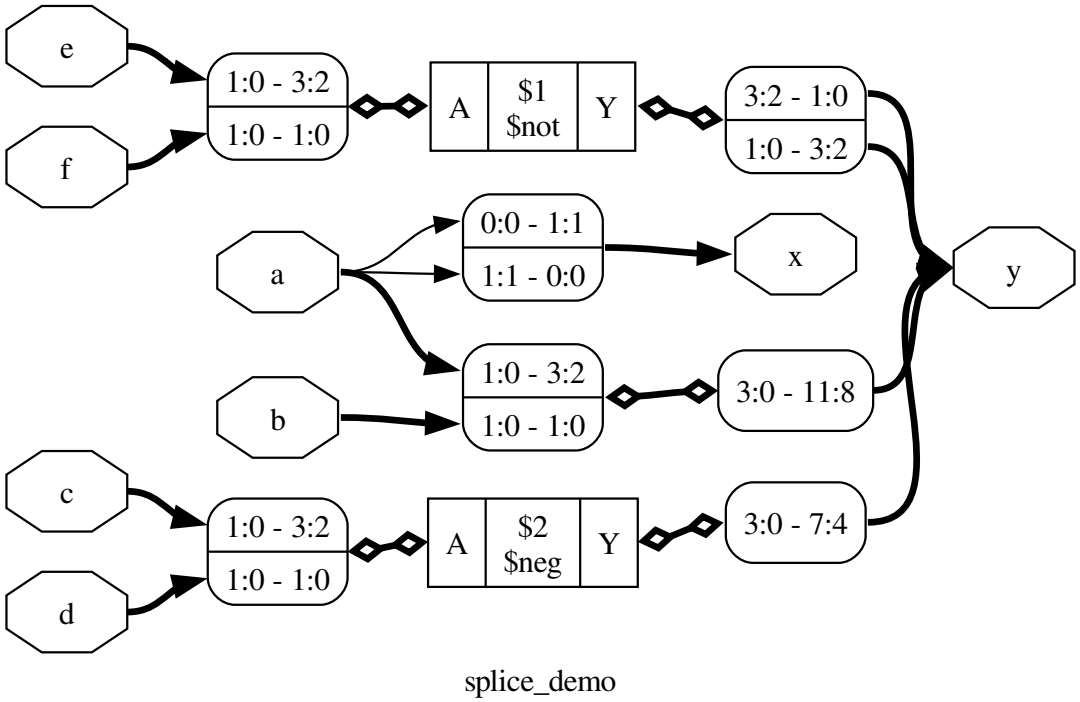


Fig. 3.27: Output of `yosys -p 'prep -top splice_demo; show' splice.v`

The key elements in understanding this circuit diagram are of course the boxes with round corners and rows labeled `<MSB_LEFT>:<LSB_LEFT> - <MSB_RIGHT>:<LSB_RIGHT>`. Each of these boxes have one signal per row on one side and a common signal for all rows on the other side. The `<MSB>:<LSB>` tuples specify which bits of the signals are broken out and connected. So the top row of the box connecting the signals `a` and `x` indicates that the bit 0 (i.e. the range 0:0) from signal `a` is connected to bit 1 (i.e. the range 1:1) of signal `x`.

Lines connecting such boxes together and lines connecting such boxes to cell ports have a slightly different look to emphasise that they are not actual signal wires but a necessity of the graphical representation. This distinction seems like a technicality, until one wants to debug a problem related to the way Yosys internally represents signal vectors, for example when writing custom Yosys commands.

Gate level netlists

Fig. 3.28 shows two common pitfalls when working with designs mapped to a cell library:

Listing 3.51: Generating Fig. 3.28

```
read_verilog cmos.v
prep -top cmos_demo
techmap
```

(continues on next page)

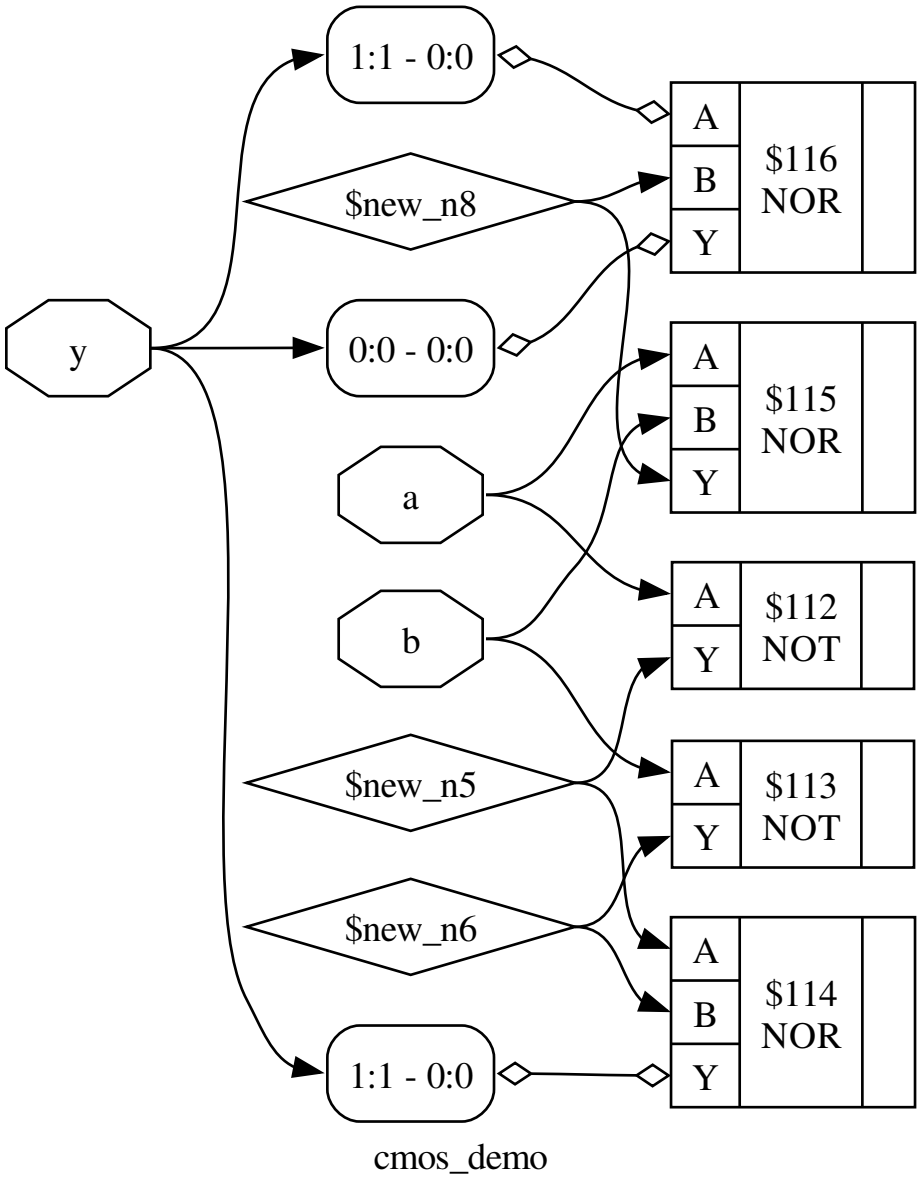


Fig. 3.28: A half-adder built from simple CMOS gates, demonstrating common pitfalls when using *show*

(continued from previous page)

```
abc -liberty ../intro/mycells.lib;;
show -format dot -prefix cmos_00
```

First, Yosys did not have access to the cell library when this diagram was generated, resulting in all cell ports defaulting to being inputs. This is why all ports are drawn on the left side the cells are awkwardly arranged in a large column. Secondly the two-bit vector `y` requires breakout-boxes for its individual bits, resulting in an unnecessary complex diagram.

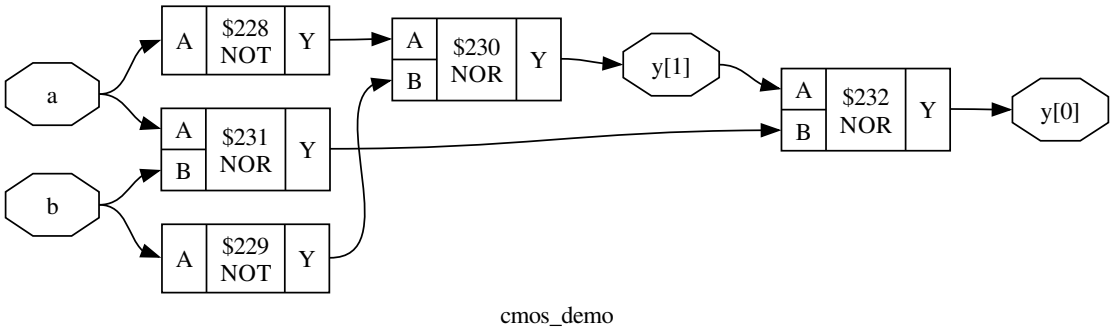


Fig. 3.29: Effects of `splitnets` command and of providing a cell library on design in Fig. 3.28

Listing 3.52: Generating Fig. 3.29

```
read_verilog cmos.v
prep -top cmos_demo
techmap
splitnets -ports
abc -liberty ../intro/mycells.lib;;
show -lib ../intro/mycells.v -format dot -prefix cmos_01
```

For Fig. 3.29, Yosys has been given a description of the cell library as Verilog file containing blackbox modules. There are two ways to load cell descriptions into Yosys: First the Verilog file for the cell library can be passed directly to the *show* command using the `-lib <filename>` option. Secondly it is possible to load cell libraries into the design with the `read_verilog -lib <filename>` command. The second method has the great advantage that the library only needs to be loaded once and can then be used in all subsequent calls to the *show* command.

In addition to that, Fig. 3.29 was generated after `splitnet -ports` was run on the design. This command splits all signal vectors into individual signal bits, which is often desirable when looking at gate-level circuits. The `-ports` option is required to also split module ports. Per default the command only operates on interior signals.

Miscellaneous notes

Per default the *show* command outputs a temporary dot file and launches `xdot` to display it. The options `-format`, `-viewer` and `-prefix` can be used to change format, viewer and filename prefix. Note that the `pdf` and `ps` format are the only formats that support plotting multiple modules in one run. The `dot` format can be used to output multiple modules, however `xdot` will raise an error when trying to read them.

In densely connected circuits it is sometimes hard to keep track of the individual signal wires. For these cases it can be useful to call *show* with the `-colors <integer>` argument, which randomly assigns colors to the nets. The integer (> 0) is used as seed value for the random color assignments. Sometimes it is necessary it try some values to find an assignment of colors that looks good.

The command `help show` prints a complete listing of all options supported by the *show* command.

Navigating the design

Plotting circuit diagrams for entire modules in the design brings us only helps in simple cases. For complex modules the generated circuit diagrams are just stupidly big and are no help at all. In such cases one first has to select the relevant portions of the circuit.

In addition to *what* to display one also needs to carefully decide *when* to display it, with respect to the synthesis flow. In general it is a good idea to troubleshoot a circuit in the earliest state in which a problem can be reproduced. So if, for example, the internal state before calling the `techmap` command already fails to verify, it is better to troubleshoot the coarse-grain version of the circuit before `techmap` than the gate-level circuit after `techmap`.

Note

It is generally recommended to verify the internal state of a design by writing it to a Verilog file using `write_verilog -noexpr` and using the simulation models from `simlib.v` and `simcells.v` from the Yosys data directory (as printed by `yosys-config --datdir`).

Interactive navigation

Once the right state within the synthesis flow for debugging the circuit has been identified, it is recommended to simply add the `shell` command to the matching place in the synthesis script. This command will stop the synthesis at the specified moment and go to shell mode, where the user can interactively enter commands.

For most cases, the shell will start with the whole design selected (i.e. when the synthesis script does not already narrow the selection). The command *ls* can now be used to create a list of all modules. The command *cd* can be used to switch to one of the modules (type `cd ..` to switch back). Now the *ls* command lists the objects within that module. This is demonstrated below using `example.v` from *A simple circuit*:

Listing 3.53: Output of *ls* and *cd* after running `yosys example.v`

```
yosys> ls

1 modules:
  example

yosys> cd example

yosys [example]> ls

8 wires:
  $0\y[1:0]
  $add$example.v:5$2_Y
  $ternary$example.v:5$3_Y
  a
  b
  c
```

(continues on next page)

(continued from previous page)

```
  clk
  y

2 cells:
  $add$example.v:5$2
  $ternary$example.v:5$3

1 processes:
  $proc$example.v:3$1
```

When a module is selected using the *cd* command, all commands (with a few exceptions, such as the `read_` and `write_` commands) operate only on the selected module. This can also be useful for synthesis scripts where different synthesis strategies should be applied to different modules in the design.

We can see that the cell names from Fig. 3.26 are just abbreviations of the actual cell names, namely the part after the last dollar-sign. Most auto-generated names (the ones starting with a dollar sign) are rather long and contains some additional information on the origin of the named object. But in most cases those names can simply be abbreviated using the last part.

Usually all interactive work is done with one module selected using the *cd* command. But it is also possible to work from the design-context (`cd ..`). In this case all object names must be prefixed with `<module_name>/`. For example `a*/b*` would refer to all objects whose names start with `b` from all modules whose names start with `a`.

The `dump` command can be used to print all information about an object. For example, calling `dump $2` after the `cd example` above:

Listing 3.54: Output of `dump $2` after Listing 3.53

```
attribute \src "example.v:5.22-5.27"
cell $add $add$example.v:5$2
  parameter \A_SIGNED 0
  parameter \B_SIGNED 0
  parameter \A_WIDTH 1
  parameter \B_WIDTH 1
  parameter \Y_WIDTH 2
  connect \A \a
  connect \B \b
  connect \Y $add$example.v:5$2_Y
end
```

This can for example be useful to determine the names of nets connected to cells, as the net-names are usually suppressed in the circuit diagram if they are auto-generated. Note that the output is in the RTLIL representation, described in *The RTL Intermediate Language (RTLIL)*.

Design Investigation

Yosys can also be used to investigate designs (or netlists created from other tools).

- The selection mechanism, especially patterns such as `%ci` and `%co`, can be used to figure out how parts of the design are connected.
- Commands such as `submod`, *expose*, and `splice` can be used to transform the design into an equivalent design that is easier to analyse.
- Commands such as `eval` and *sat* can be used to investigate the behavior of the circuit.

- *show - generate schematics using graphviz*.
- `dump`.
- `add` and `delete` can be used to modify and reorganize a design dynamically.

The code used is included in the Yosys code base under `docs/source/code_examples/scrambler`.

Changing design hierarchy

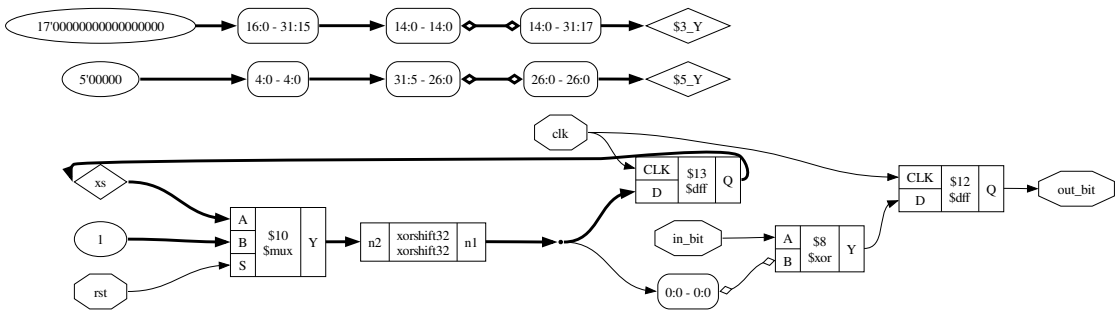
Commands such as `flatten` and `submod` can be used to change the design hierarchy, i.e. flatten the hierarchy or moving parts of a module to a submodule. This has applications in synthesis scripts as well as in reverse engineering and analysis. An example using `submod` is shown below for reorganizing a module in Yosys and checking the resulting circuit.

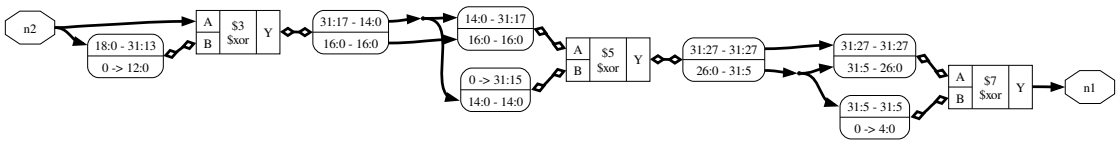
Listing 3.55: `scrambler.v`

```
module scrambler(
        input clk, rst, in_bit,
        output reg out_bit
);
    reg [31:0] xs;
    always @(posedge clk) begin
            if (rst)
            xs = 1;
        xs = xs ^ (xs << 13);
        xs = xs ^ (xs >> 17);
        xs = xs ^ (xs << 5);
        out_bit <= in_bit ^ xs[0];
    end
endmodule
```

Listing 3.56: `scrambler.ys`

```
read_verilog scrambler.v

hierarchy; proc;;

cd scrambler
submod -name xorshift32 xs %c %ci %D %c %ci:+[D] %D %ci*:-$dff xs %co %ci %d
```





Analyzing the resulting circuit with `eval`:

Todo

replace inline code

```
> cd xorshift32
> rename n2 in
> rename n1 out

> eval -set in 1 -show out
Eval result: \out = 270369.

> eval -set in 270369 -show out
Eval result: \out = 67634689.

> sat -set out 632435482
Signal Name                  Dec        Hex                                  Bin
-------------------- ---------- ---------- ----------------------------------
\in                   745495504   2c6f5bd0      00101100011011110101101111010000
\out                  632435482   25b2331a      00100101101100100011001100011010
```

Behavioral changes

Commands such as `techmap` can be used to make behavioral changes to the design, for example changing asynchronous resets to synchronous resets. This has applications in design space exploration (evaluation of various architectures for one circuit).

The following techmap map file replaces all positive-edge async reset flip-flops with positive-edge sync reset flip-flops. The code is taken from the example Yosys script for ASIC synthesis of the Amber ARMv2 CPU.

Todo

replace inline code

```
(* techmap_celltype = "$adff" *)
module adff2dff (CLK, ARST, D, Q);

    parameter WIDTH = 1;
    parameter CLK_POLARITY = 1;
```

(continues on next page)

(continued from previous page)

```
    parameter ARST_POLARITY = 1;
    parameter ARST_VALUE = 0;

    input CLK, ARST;
    input [WIDTH-1:0] D;
    output reg [WIDTH-1:0] Q;

    wire [1023:0] _TECHMAP_DO_ = "proc";

    wire _TECHMAP_FAIL_ = !CLK_POLARITY || !ARST_POLARITY;

    always @(posedge CLK)
        if (ARST)
            Q <= ARST_VALUE;
        else
            Q <= D;

endmodule
```

For more on the `techmap` command, see the page on *Techmap by example*.

Advanced investigation techniques

When working with very large modules, it is often not enough to just select the interesting part of the module. Instead it can be useful to extract the interesting part of the circuit into a separate module. This can for example be useful if one wants to run a series of synthesis commands on the critical part of the module and wants to carefully read all the debug output created by the commands in order to spot a problem. This kind of troubleshooting is much easier if the circuit under investigation is encapsulated in a separate module.

Recall the `memdemo` design from *Advanced logic cone selection*:

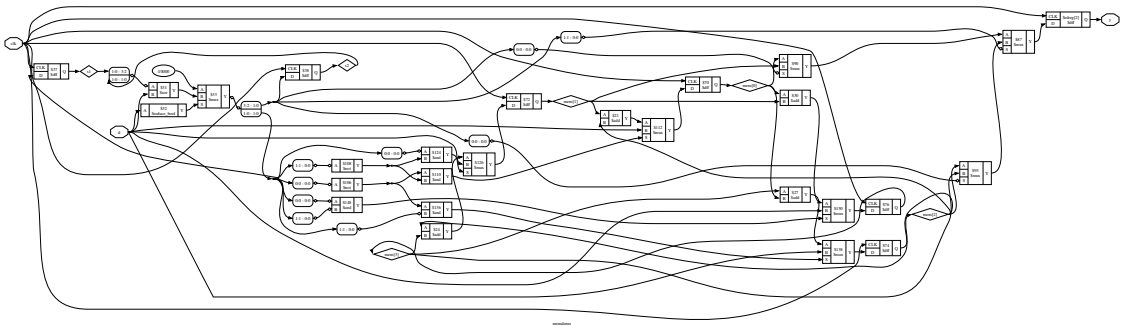


Fig. 3.30: `memdemo`

Because this produces a rather large circuit, it can be useful to split it into smaller parts for viewing and working with. Listing 3.57 does exactly that, utilising the `submod` command to split the circuit into three sections: `outstage`, `selstage`, and `scramble`.

Listing 3.57: Using `submod` to break up the circuit from `memdemo.v`

```
select -set outstage y %ci2:+$dff[Q,D] %ci*:-$mux[S]:-$dff
select -set selstage y %ci2:+$dff[Q,D] %ci*:-$dff @outstage %d
select -set scramble mem* %ci2 %ci*:-$dff mem* %d @selstage %d
submod -name scramble @scramble
submod -name outstage @outstage
submod -name selstage @selstage
```

The `-name` option is used to specify the name of the new module and also the name of the new cell in the current module. The resulting circuits are shown below.

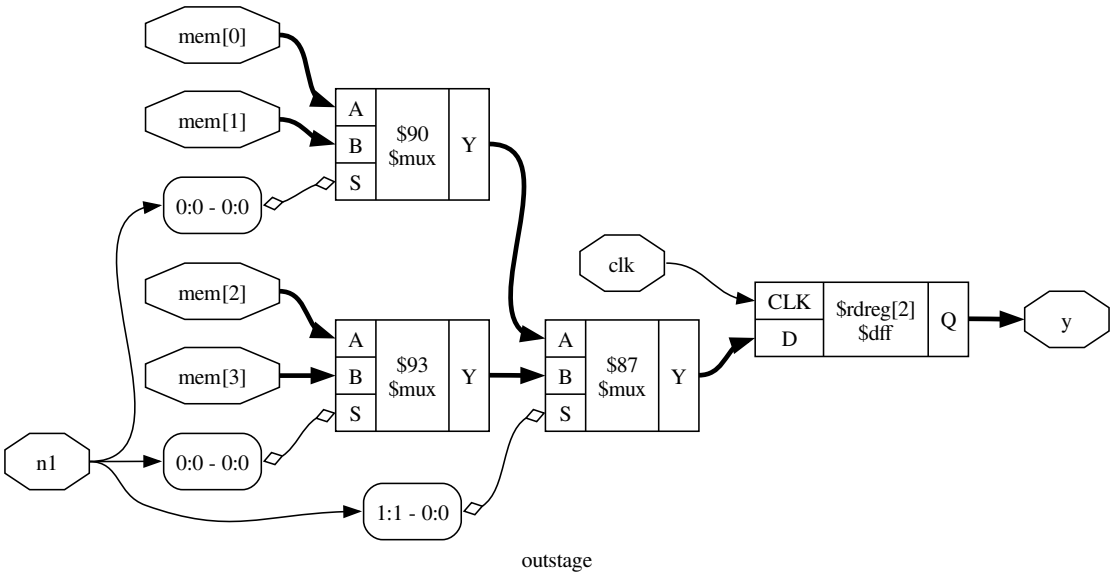


Fig. 3.31: `outstage`

Evaluation of combinatorial circuits

The `eval` command can be used to evaluate combinatorial circuits. As an example, we will use the `selstage` subnet of `memdemo` which we found above and is shown in Fig. 3.32.

Todo

replace inline code

```
yosys [selstage]> eval -set s2,s1 4'b1001 -set d 4'hc -show n2 -show n1

1. Executing EVAL pass (evaluate the circuit given an input).
Full command line: eval -set s2,s1 4'b1001 -set d 4'hc -show n2 -show n1
```

(continues on next page)

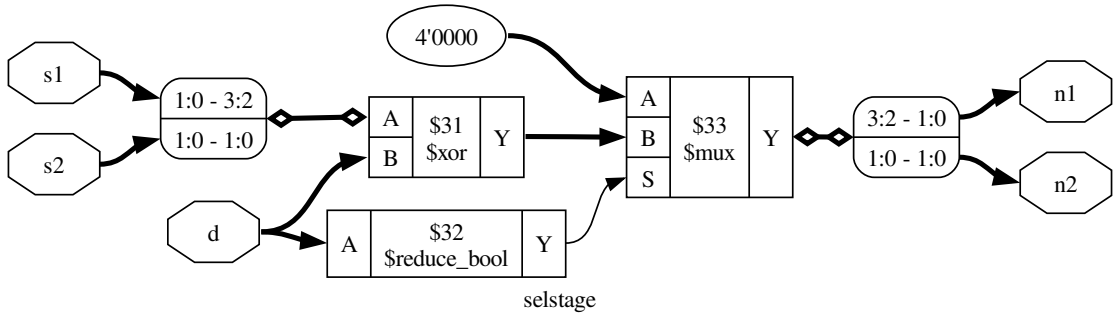


Fig. 3.32: `selstage`

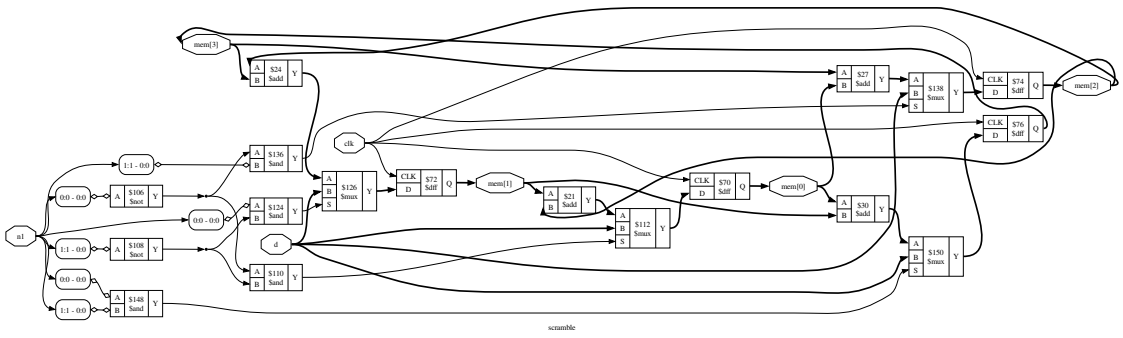


Fig. 3.33: `scramble`

(continued from previous page)

```
Eval result: \n2 = 2'10.
Eval result: \n1 = 2'10.
```

So the `-set` option is used to set input values and the `-show` option is used to specify the nets to evaluate. If no `-show` option is specified, all selected output ports are used per default.

If a necessary input value is not given, an error is produced. The option `-set-undef` can be used to instead set all unspecified input nets to undef (`x`).

The `-table` option can be used to create a truth table. For example:

```
yosys [selstage]> eval -set-undef -set d[3:1] 0 -table s1,d[0]

10. Executing EVAL pass (evaluate the circuit given an input).
Full command line: eval -set-undef -set d[3:1] 0 -table s1,d[0]

  \s1 \d [0] |  \n1  \n2
 ---- ------ | ---- ----
 2'00    1'0 | 2'00 2'00
 2'00    1'1 | 2'xx 2'00
 2'01    1'0 | 2'00 2'00
 2'01    1'1 | 2'xx 2'01
 2'10    1'0 | 2'00 2'00
 2'10    1'1 | 2'xx 2'10
 2'11    1'0 | 2'00 2'00
 2'11    1'1 | 2'xx 2'11

Assumed undef (x) value for the following signals: \s2
```

Note that the `eval` command (as well as the *sat* command discussed in the next sections) does only operate on flattened modules. It can not analyze signals that are passed through design hierarchy levels. So the `flatten` command must be used on modules that instantiate other modules before these commands can be applied.

Solving combinatorial SAT problems

Often the opposite of the `eval` command is needed, i.e. the circuits output is given and we want to find the matching input signals. For small circuits with only a few input bits this can be accomplished by trying all possible input combinations, as it is done by the `eval -table` command. For larger circuits however, Yosys provides the *sat* command that uses a SAT solver, MiniSAT, to solve this kind of problems.

> **Note**
>
> While it is possible to perform model checking directly in Yosys, it is highly recommended to use SBY or EQY for formal hardware verification.

The *sat* command works very similar to the `eval` command. The main difference is that it is now also possible to set output values and find the corresponding input values. For Example:

Todo

replace inline code

```
yosys [selstage]> sat -show s1,s2,d -set s1 s2 -set n2,n1 4'b1001

11. Executing SAT pass (solving SAT problems in the circuit).
Full command line: sat -show s1,s2,d -set s1 s2 -set n2,n1 4'b1001

Setting up SAT problem:
Import set-constraint: \s1 = \s2
Import set-constraint: { \n2 \n1 } = 4'1001
Final constraint equation: { \n2 \n1 \s1 } = { 4'1001 \s2 }
Imported 3 cells to SAT database.
Import show expression: { \s1 \s2 \d }

Solving problem with 81 variables and 207 clauses..
SAT solving finished - model found:

  Signal Name                 Dec        Hex             Bin
  -------------------- ---------- ---------- ---------------
  \d                            9          9            1001
  \s1                           0          0              00
  \s2                           0          0              00
```

Note that the *sat* command supports signal names in both arguments to the `-set` option. In the above example we used `-set s1 s2` to constraint `s1` and `s2` to be equal. When more complex constraints are needed, a wrapper circuit must be constructed that checks the constraints and signals if the constraint was met using an extra output port, which then can be forced to a value using the `-set` option. (Such a circuit that contains the circuit under test plus additional constraint checking circuitry is called a `miter` circuit.)

Listing 3.58: `primetest.v`, a simple miter circuit for testing if a number is prime. But it has a problem.

```
module primetest(p, a, b, ok);
input [15:0] p, a, b;
output ok = p != a*b || a == 1 || b == 1;
endmodule
```

Listing 3.58 shows a miter circuit that is supposed to be used as a prime number test. If `ok` is 1 for all input values `a` and `b` for a given `p`, then `p` is prime, or at least that is the idea.

Todo

replace inline code

Listing 3.59: Experiments with the miter circuit from `primetest.v`.

```
yosys [primetest]> sat -prove ok 1 -set p 31

```

(continues on next page)

(continued from previous page)

```
1. Executing SAT pass (solving SAT problems in the circuit).
Full command line: sat -prove ok 1 -set p 31

Setting up SAT problem:
Import set-constraint: \p = 16'0000000000011111
Final constraint equation: \p = 16'0000000000011111
Imported 6 cells to SAT database.
Import proof-constraint: \ok = 1'1
Final proof equation: \ok = 1'1

Solving problem with 2790 variables and 8241 clauses..
SAT proof finished - model found: FAIL!

  ______                    ___       ___       _ _            _ _
 (_____ \                  / __)     / __)     (_) |          | | |
  _____) )___ ___   ___ _| |__    _| |__ _____ _| | _____  __| | |
 |  ____/ ___) _ \ / _ (_   __)  (_   __|____ | | || ___ |/ _  |_|
 | |   | |  | |_| | |_| || |       | |  / ___ | | || ____( (_| |_
 |_|   |_|   \___/ \___/ |_|       |_|  \_____|_|\_)_____)\____|_|


  Signal Name                 Dec        Hex                   Bin
  -------------------- ---------- ---------- ----------------------
  \a                         15029       3ab5      0011101010110101
  \b                          4099       1003      0001000000000011
  \ok                            0          0                     0
  \p                            31         1f      0000000000011111
```

The Yosys shell session shown in Listing 3.59 demonstrates that SAT solvers can even find the unexpected solutions to a problem: Using integer overflow there actually is a way of "factorizing" 31. The clean solution would of course be to perform the test in 32 bits, for example by replacing `p != a*b` in the miter with `p != {16'd0,a}b`, or by using a temporary variable for the 32 bit product `a*b`. But as 31 fits well into 8 bits (and as the purpose of this document is to show off Yosys features) we can also simply force the upper 8 bits of `a` and `b` to zero for the *sat* call, as is done below.

Todo

replace inline code

Listing 3.60: Miter circuit from `primetest.v`, with the upper 8 bits of `a` and `b` constrained to prevent overflow.

```
yosys [primetest]> sat -prove ok 1 -set p 31 -set a[15:8],b[15:8] 0

1. Executing SAT pass (solving SAT problems in the circuit).
Full command line: sat -prove ok 1 -set p 31 -set a[15:8],b[15:8] 0

Setting up SAT problem:
Import set-constraint: \p = 16'0000000000011111
Import set-constraint: { \a [15:8] \b [15:8] } = 16'0000000000000000
Final constraint equation: { \a [15:8] \b [15:8] \p } = { 16'0000000000000000 16
```

(continues on next page)

(continued from previous page)

```
↪'0000000000011111 }
Imported 6 cells to SAT database.
Import proof-constraint: \ok = 1'1
Final proof equation: \ok = 1'1

Solving problem with 2790 variables and 8257 clauses..
SAT proof finished - no model found: SUCCESS!

                  /$$$$$$      /$$$$$$$$     /$$$$$$$
                 /$$__  $$    | $$_____/    | $$__  $$
                | $$  \ $$    | $$          | $$  \ $$
                | $$  | $$    | $$$$$       | $$  | $$
                | $$  | $$    | $$__/       | $$  | $$
                | $$/$$ $$    | $$          | $$  | $$
                |  $$$$$$/ /$$| $$$$$$$$ /$$| $$$$$$$//$$
                 \____ $$$|__/|________/|__/|_______/|__/
                      \__/
```

The `-prove` option used in Listing 3.60 works similar to `-set`, but tries to find a case in which the two arguments are not equal. If such a case is not found, the property is proven to hold for all inputs that satisfy the other constraints.

It might be worth noting, that SAT solvers are not particularly efficient at factorizing large numbers. But if a small factorization problem occurs as part of a larger circuit problem, the Yosys SAT solver is perfectly capable of solving it.

Solving sequential SAT problems

The SAT solver functionality in Yosys can not only be used to solve combinatorial problems, but can also solve sequential problems. Let's consider the `memdemo` design from *Advanced logic cone selection* again, and suppose we want to know which sequence of input values for `d` will cause the output y to produce the sequence 1, 2, 3 from any initial state. Let's use the following command:

Todo

replace inline code?

```
sat -seq 6 -show y -show d -set-init-undef \
  -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3
```

The `-seq 6` option instructs the *sat* command to solve a sequential problem in 6 time steps. (Experiments with lower number of steps have show that at least 3 cycles are necessary to bring the circuit in a state from which the sequence 1, 2, 3 can be produced.)

The `-set-init-undef` option tells the *sat* command to initialize all registers to the undef (`x`) state. The way the `x` state is treated in Verilog will ensure that the solution will work for any initial state.

The `-max_undef` option instructs the *sat* command to find a solution with a maximum number of undefs. This way we can see clearly which inputs bits are relevant to the solution.

Finally the three `-set-at` options add constraints for the `y` signal to play the 1, 2, 3 sequence, starting with time step 4.

This produces the following output:

Todo

replace inline code

Listing 3.61: Solving a sequential SAT problem in the `memdemo` module.

```
yosys [memdemo]> sat -seq 6 -show y -show d -set-init-undef \
    -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3

1. Executing SAT pass (solving SAT problems in the circuit).
Full command line: sat -seq 6 -show y -show d -set-init-undef
    -max_undef -set-at 4 y 1 -set-at 5 y 2 -set-at 6 y 3

Setting up time step 1:
Final constraint equation: { } = { }
Imported 29 cells to SAT database.

Setting up time step 2:
Final constraint equation: { } = { }
Imported 29 cells to SAT database.

Setting up time step 3:
Final constraint equation: { } = { }
Imported 29 cells to SAT database.

Setting up time step 4:
Import set-constraint for timestep: \y = 4'0001
Final constraint equation: \y = 4'0001
Imported 29 cells to SAT database.

Setting up time step 5:
Import set-constraint for timestep: \y = 4'0010
Final constraint equation: \y = 4'0010
Imported 29 cells to SAT database.

Setting up time step 6:
Import set-constraint for timestep: \y = 4'0011
Final constraint equation: \y = 4'0011
Imported 29 cells to SAT database.

Setting up initial state:
Final constraint equation: { \y \s2 \s1 \mem[3] \mem[2] \mem[1]
            \mem[0] } = 24'xxxxxxxxxxxxxxxxxxxxxxxx

Import show expression: \y
Import show expression: \d

Solving problem with 10322 variables and 27881 clauses..
SAT model found. maximizing number of undefs.
SAT solving finished - model found:
```

(continues on next page)

(continued from previous page)

```
Time Signal Name                 Dec        Hex              Bin
---- -------------------- ---------- ---------- ---------------
init \mem[0]                      --         --             xxxx
init \mem[1]                      --         --             xxxx
init \mem[2]                      --         --             xxxx
init \mem[3]                      --         --             xxxx
init \s1                          --         --               xx
init \s2                          --         --               xx
init \y                           --         --             xxxx
---- -------------------- ---------- ---------- ---------------
   1 \d                            0          0             0000
   1 \y                           --         --             xxxx
---- -------------------- ---------- ---------- ---------------
   2 \d                            1          1             0001
   2 \y                           --         --             xxxx
---- -------------------- ---------- ---------- ---------------
   3 \d                            2          2             0010
   3 \y                            0          0             0000
---- -------------------- ---------- ---------- ---------------
   4 \d                            3          3             0011
   4 \y                            1          1             0001
---- -------------------- ---------- ---------- ---------------
   5 \d                           --         --             001x
   5 \y                            2          2             0010
---- -------------------- ---------- ---------- ---------------
   6 \d                           --         --             xxxx
   6 \y                            3          3             0011
```

It is not surprising that the solution sets `d = 0` in the first step, as this is the only way of setting the `s1` and `s2` registers to a known value. The input values for the other steps are a bit harder to work out manually, but the SAT solver finds the correct solution in an instant.

There is much more to write about the *`sat`* command. For example, there is a set of options that can be used to performs sequential proofs using temporal induction [EenSorensson03]. The command `help sat` can be used to print a list of all options with short descriptions of their functions.

3.2.4 Symbolic model checking

Todo

check text context

Note

While it is possible to perform model checking directly in Yosys, it is highly recommended to use SBY or EQY for formal hardware verification.

Symbolic Model Checking (SMC) is used to formally prove that a circuit has (or has not) a given property.

One application is Formal Equivalence Checking: Proving that two circuits are identical. For example this is a very useful feature when debugging custom passes in Yosys.

Other applications include checking if a module conforms to interface standards.

The *sat* command in Yosys can be used to perform Symbolic Model Checking.

Checking techmap

Todo

add/expand supporting text

Let's take a look at an example included in the Yosys code base under docs/source/code_examples/synth_flow:

Listing 3.62: techmap_01_map.v

```
module \$add (A, B, Y);

parameter A_SIGNED = 0;
parameter B_SIGNED = 0;
parameter A_WIDTH = 1;
parameter B_WIDTH = 1;
parameter Y_WIDTH = 1;

input [A_WIDTH-1:0] A;
input [B_WIDTH-1:0] B;
output [Y_WIDTH-1:0] Y;

generate
  if ((A_WIDTH == 32) && (B_WIDTH == 32))
    begin
      wire [16:0] S1 = A[15:0] + B[15:0];
      wire [15:0] S2 = A[31:16] + B[31:16] + S1[16];
      assign Y = {S2[15:0], S1[15:0]};
    end
  else
    wire _TECHMAP_FAIL_ = 1;
endgenerate

endmodule
```

Listing 3.63: techmap_01.v

```
module test(input [31:0]  a, b,
            output [31:0] y);
assign y = a + b;
endmodule
```

Listing 3.64: techmap_01.ys

```
read_verilog techmap_01.v
hierarchy -check -top test
techmap -map techmap_01_map.v;;
```

To see if it is correct we can use the following code:

Todo

replace inline code

```
# read test design
read_verilog techmap_01.v
hierarchy -top test

# create two version of the design: test_orig and test_mapped
copy test test_orig
rename test test_mapped

# apply the techmap only to test_mapped
techmap -map techmap_01_map.v test_mapped

# create a miter circuit to test equivalence
miter -equiv -make_assert -make_outputs test_orig test_mapped miter
flatten miter

# run equivalence check
sat -verify -prove-asserts -show-inputs -show-outputs miter
```

Result:

```
Solving problem with 945 variables and 2505 clauses..
SAT proof finished - no model found: SUCCESS!
```

AXI4 Stream Master

The code used in this section is included in the Yosys code base under `docs/source/code_examples/axis`.

The following AXI4 Stream Master has a bug. But the bug is not exposed if the slave keeps `tready` asserted all the time. (Something a test bench might do.)

Symbolic Model Checking can be used to expose the bug and find a sequence of values for `tready` that yield the incorrect behavior.

Todo

add/expand supporting text

Listing 3.65: axis_master.v

```
module axis_master(aclk, aresetn, tvalid, tready, tdata);
    input aclk, aresetn, tready;
    output reg tvalid;
    output reg [7:0] tdata;

    reg [31:0] state;
    always @(posedge aclk) begin
        if (!aresetn) begin
            state <= 314159265;
            tvalid <= 0;
            tdata <= 'bx;
        end else begin
            if (tvalid && tready)
                    tvalid <= 0;
            if (!tvalid || !tready) begin
            //              ^- should not be inverted!
                state = state ^ state << 13;
                state = state ^ state >> 7;
                state = state ^ state << 17;
                if (state[9:8] == 0) begin
                    tvalid <= 1;
                    tdata <= state;
                end
            end
        end
    end
endmodule
```

Listing 3.66: axis_test.v

```
module axis_test(aclk, tready);
    input aclk, tready;
    wire aresetn, tvalid;
    wire [7:0] tdata;

    integer counter = 0;
    reg aresetn = 0;

    axis_master uut (aclk, aresetn, tvalid, tready, tdata);

    always @(posedge aclk) begin
            if (aresetn && tready && tvalid) begin
            if (counter == 0) assert(tdata ==  19);
            if (counter == 1) assert(tdata ==  99);
            if (counter == 2) assert(tdata ==   1);
            if (counter == 3) assert(tdata == 244);
            if (counter == 4) assert(tdata == 133);
            if (counter == 5) assert(tdata == 209);
            if (counter == 6) assert(tdata == 241);
            if (counter == 7) assert(tdata == 137);
```

(continues on next page)

(continued from previous page)

```
            if (counter == 8) assert(tdata == 176);
            if (counter == 9) assert(tdata ==   6);
            counter <= counter + 1;
        end
        aresetn <= 1;
    end
endmodule
```

Listing 3.67: `test.ys`

```
read_verilog -sv axis_master.v axis_test.v
hierarchy -top axis_test

proc; flatten;;
sat -seq 50 -prove-asserts
```

Result with unmodified `axis_master.v`:

Todo

replace inline code

```
Solving problem with 159344 variables and 442126 clauses..
SAT proof finished - model found: FAIL!
```

Result with fixed `axis_master.v`:

```
Solving problem with 159144 variables and 441626 clauses..
SAT proof finished - no model found: SUCCESS!
```

Witness framework and per-property tracking

When using AIGER-based formal verification flows (such as the `abc` engine in SBY), Yosys provides infrastructure for tracking individual formal properties through the verification pipeline.

The `rename -witness` pass assigns public names to all cells that participate in the witness framework:

- Witness signal cells: *`$anyconst`*, *`$anyseq`*, *`$anyinit`*, *`$allconst`*, *`$allseq`*
- Formal property cells: *`$assert`*, *`$assume`*, *`$cover`*, *`$live`*, *`$fair`*, *`$check`*

These public names allow downstream tools to refer to individual properties by their hierarchical path rather than by anonymous internal identifiers.

The `write_aiger -ywmap` option generates a map file for conversion to and from Yosys witness traces, and also allows for mapping AIGER bad-state properties and invariant constraints back to individual formal properties by name. This enables features like per-property pass/fail reporting (e.g. `abc pdr` with `--keep-going` mode).

The *`write_smt2`* backend similarly uses the public witness names when emitting SMT2 comments. Cells whose `hdlname` attribute contains the `_witness_` marker are treated as having private names for comment purposes, keeping solver output clean.

3.2.5 Dataflow tracking

Yosys can be used to answer questions such as "can this signal affect this other signal?" via its *dataflow tracking* support. For this, four special cells, *$get_tag*, *$set_tag*, *$overwrite_tag* and *$original_tag* are inserted into the design (e.g. by a custom Yosys pass) and then the *dft_tag* is run, which converts these cells into ordinary logic. Typically, one would then use SBY to prove assertions involving these cells.

Ordinarily in Yosys, the state of a bit is simply `0` or `1` (or one of the special values, `z` and `x`). During dataflow tracking they are augmented with a set of tags. For example, the state of a bit could be `0` and the set of tags `"KEY"` and `"OVERFLOW"`.

In addition to their usual operations on the logical bits, Yosys operations must now also process the status of the tags. For this, tags are simply *forwarded* or *propagated* (i.e. copied) from inputs to outputs, according to the following general rule:

> A tag is forwarded from an input to an output if the input can affect the output, for that particular state of all other inputs.

For example, XOR, AND and OR cells propagate tags as follows:

1. XOR simply forwards all tags from its inputs to its output, because inputs to XOR can always affect the output.
2. AND forwards tags on a given input only if the other input is `1`. Because if one input is `0`, the other input can never affect the output.
3. Similarly, OR forwards tags only if the other input is `0`.

There are two exceptions to this rule:

1. In general, propagation is only determined approximately. For example, unless the `dft_tag` code knows about a cell, it simply assumes the worst-case behaviour that all inputs can affect all outputs. Further, the code also does not consider that, when a signal affects multiple inputs of a cell, the resulting simultaneous changes of the inputs can cancel each other out, for example `A ^ A` or `A ^ (B ^ A)` is independent of `A`, but its tags would be propagated nonetheless.
2. If tag groups are used, the rules are modified (see below).

Because of this propagation behaviour, we can answer questions about what signals are affected by a certain signal, by injecting a tag at that point in the circuit, and observing where the tag is visible.

Example use cases

As an example use case, consider a cryptographic processor which is not supposed to expose its secret keys to the outside world. We can tag all key bits with the `"KEY"` tag and use SBY to formally verify that no external signal ever carries the `"KEY"` tag, meaning that key information is not visible to the outside. As a caveat, we have to manually clear the `"KEY"` tag during cryptographic operations, as proving that the cryptographic operations themselves do not leak key information is beyond the ability of Yosys. However we can still easily detect, if e.g. an engineer forgot to remove debugging code that allows reading back key data.

As a different use case, we can modify all adders in the design to set the `"OVERFLOW"` tag on their output bits, if the addition overflowed, and then add asserts to all flip-flop inputs and output signals that they do not carry the `"OVERFLOW"` tag, i.e. that the results of overflowed additions never affect system state. Note that in this particular example we use the ability of tag insertion to be conditional on logic, in this case the overflow condition of an adder.

Semantics of dataflow tracking cells

`$set_tag` has inputs `A`, `SET`, `CLR`, an output `Y` and a string parameter `TAG`. The logic value of `A` and all tags other than the one named by the `TAG` parameter are simply copied to `Y`. If `SET` is `1`, then the named tag is added to `Y`. Otherwise, if `CLR` is `1`, then the named tag is removed. Otherwise, the tag is unchanged, i.e. it is present in `Y` if it is present in `A`.

`$get_tag` has an input `A` and an output `Y` and a string parameter `TAG`. `$get_tag` inspects `A` for the presence or absence of a tag of the given name and sets `Y` to `1` if the tag is present. The logical value of `A` is completely ignored.

`$overwrite_tag` functions like `$set_tag`, but lacks the `Y` output. Instead of providing a modified version of the input signal, it modifies the signal `A` "in-place", i.e. if a signal is input to `$overwrite_tag`, that is equivalent to interposing a `$set_tag` between its driver and all cells it is connected to. The main purpose of `$overwrite_tag` is adding tags to signals produced within a module that cannot or should not be modified itself.

`$original_tag` functions identically to `$get_tag`, but ignores `$overwrite_tag`, i.e. when converting the `$overwrite_tag` to `$set_tag` as described above, it is equivalent to inserting the `$get_tag` *before* the `$set_tag`.

Tag groups

Tag groups are an advanced feature that modify the propagation rule discussed above. To use tag groups, simply name tags according to the schema `"group:name"`. For example, `"key:0"`, `"key:a"`, `"key:b"` would be three tags in the `"key"` group.

The propagation rule is then amended by

> Inputs cannot block the propagation of each other's tags for tags of the same group.

For example, an AND gate will propagate a given tag on one input, if the other input is either 1 or carries a tag of the same group. So if one input is `0, "key:a"` and the other is `0, "key:b"` the result would be `0, "key:a", "key:b"`, rather than simply `0`. Note that if we add an unrelated `"overflow"` tag to the first input, it would still not be propagated.

3.3 Minimizing failing (or bugged) designs

Todo

pending merge of https://github.com/YosysHQ/yosys/pull/5068

This document is a how-to guide for reducing problematic designs to the bare minimum needed for reproducing the issue. This is a Yosys specific alternative to the Stack Overflow article: How to create a Minimal, Reproducible Example, and is intended to help when there's something wrong with your design, or with Yosys itself.

> **Note**
>
> This guide assumes a moderate degree of familiarity with Yosys and requires some amount of problem solving ability.

3.3.1 Before you start

The first (and often overlooked) step, is to check for and *read* any error messages or warnings. Passing the `-q` flag when running Yosys will make it so that only warnings and error messages are written to the console. Don't just read the last message either, there may be warnings that indicate a problem before it happens. While some things may only be regarded as warnings, such as multiple drivers for the same signal or logic loops, these can cause problems in some synthesis flows but not others.

A Yosys error (one that starts with `ERROR:`) may give you a line number from your design, or the name of the object causing issues. If so, you may already have enough information to resolve the problem, or at least understand why it's happening.

> **Note**
>
> If you're not already, try using the latest version from the Yosys GitHub. You may find that your issue has already been fixed! And even if it isn't, testing with two different versions is a good way to ensure reproducibility.

Another thing to be aware of is that Yosys generally doesn't perform rigorous checking of input designs to ensure they are valid. This is especially true for the *`read_verilog`* frontend. It is instead recommended that you try load it with iverilog or verilator first, as an invalid design can often lead to unexpected issues.

If you're using a custom synthesis script, try take a bit of time to figure out which command is failing. Calling `echo on` at the start of your script will `echo` each command executed; the last echo before the error should then be where the error has come from. Check the help message for the failing command; does it indicate limited support, or mention some other command that needs to be run first? You can also try to call *`check`* and/or `hierarchy -check` before the failure to see if they report and errors or warnings.

3.3.2 Minimizing RTLIL designs with bugpoint

Yosys provides the `bugpoint` command for reducing a failing design to the smallest portion of that design which still results in failure. While initially developed for Yosys crashes, `bugpoint` can also be used for designs that lead to non-fatal errors, or even failures in other tools that use the output of a Yosys script.

> **Note**
>
> Make sure to back up your code (design source and yosys script(s)) before making any modifications. Even if the code itself isn't important, this can help avoid "losing" the error while trying to debug it.

Can I use bugpoint?

The first thing to be aware of is that `bugpoint` is not available in every build of Yosys. Because the command works by invoking external processes, it requires that Yosys can spawn executables. Notably this means `bugpoint` is not able to be used in WebAssembly builds such as that available via YoWASP. The easiest way to check your build of Yosys is by running `yosys -h bugpoint`. If Yosys displays the help text for `bugpoint` then it is available for use.

Listing 3.68: `bugpoint` is unavailable

```
$ yosys -h bugpoint

-- Running command `help bugpoint' --
No such command or cell type: bugpoint
```

Next you need to separate loading the design from the failure point; you should be aiming to reproduce the failure by running `yosys -s <load.ys> -s <failure.ys>`. If the failure occurs while loading the design, such as during *read_verilog* you will instead have to minimize the input design yourself. Check out the instructions for *Minimizing Verilog designs* below.

Note

You should also be able to run the two scripts separately, calling first `yosys -s <load.ys> -p 'write_rtlil design.il'` and then `yosys -s <failure.ys> design.il`. If this doesn't work then it may mean that the failure isn't reproducible from RTLIL and `bugpoint` won't work either.

When we talk about failure points here, it doesn't just mean crashes or errors in Yosys. The `<failure.ys>` script can also be a user-defined failure such as the *select* command with one of the `-assert-*` options; an example where this might be useful is when a pass is supposed to remove a certain kind of cell, but there is some edge case where the cell is not removed. Another use-case would be minimizing a design which fails with the *equiv_opt* command, suggesting that the optimization in question alters the circuit in some way.

It is even possible to use `bugpoint` with failures *external* to Yosys, by making use of the *exec* command in `<failure.ys>`. This is especially useful when Yosys is outputting an invalid design, or when some other tool is incompatible with the design. Be sure to use the `exec -expect-*` options so that the pass/fail can be detected correctly. Multiple calls to *exec* can be made, or even entire shell scripts:

```
exec -expect-return 1 --bash <script.sh>
```

Our final failure we can use with `bugpoint` is one returned by a wrapper process, such as `valgrind` or `timeout`. In this case you will be calling something like `<wrapper> yosys -s <failure.ys> design.il`. Here, Yosys is run under a wrapper process which checks for some failure state, like a memory leak or excessive runtime.

How do I use bugpoint?

At this point you should have:

1. either an RTLIL file containing the design to minimize (referred to here as `design.il`), or a Yosys script, `<load.ys>`, which loads it; and
2. a Yosys script, `<failure.ys>`, which produces the failure and returns a non-zero return status.

Now call `yosys -qq -s <failure.ys> design.il` and take note of the error(s) that get printed. A template script, `<bugpoint.ys>`, is provided here which you can use. Make sure to configure it with the correct filenames and use only one of the methods to load the design. Fill in the `-grep` option with the error message printed just before. If you are using a wrapper process for your failure state, add the `-runner "<wrapper>"` option to the `bugpoint` call.

Listing 3.69: <bugpoint.ys> template script

```
# Load design
read_rtlil design.il
## OR
script <load.ys>

# Call bugpoint with failure
bugpoint -script <failure.ys> -grep "<string>"

# Save minimized design
write_rtlil min.il
```

The -grep option is used to search the log file generated by the Yosys under test. If the error message is generated by something else, such as a wrapper process or compiler sanitizer, then you should instead use -err_grep. For an OS error, like a SEGFAULT, you can also use -expect-return to check the error code returned.

Note

Checking the error message or return status is optional, but highly recommended. bugpoint can quite easily introduce bugs by creating malformed designs that commands were not intended to handle. By having some way to check the error, bugpoint can ensure that it is the *right* error being reproduced. This is even more important when <failure.ys> contains more than one command.

By default, bugpoint is able to remove any part of the design. In order to keep certain parts, for instance because you already know they are related to the failure, you can use the bugpoint_keep attribute. This can be done with (* bugpoint_keep *) in Verilog, attribute \bugpoint_keep 1 in RTLIL, or setattr -set bugpoint_keep 1 [selection] from a Yosys script. It is also possible to limit bugpoint to only removing certain *kinds* of objects, such as only removing entire modules or cells (instances of modules). For more about the options available, check help bugpoint or bugpoint.

In some situations, it may also be helpful to use *setenv* before bugpoint to set environment variables for the spawned processes. An example of this is setenv UBSAN_OPTIONS halt_on_error=1 for where you are trying to raise an error on undefined behaviour but only want the child process to halt on error.

Note

Using *setenv* in this way may or may not affect the current process. For instance the UBSAN_OPTIONS halt_on_error here only affects child processes, as does the *Yosys environment variable* ABC because they are only read on start-up. While others, such as YOSYS_NOVERIFIC and HOME, are evaluated each time they are used.

Once you have finished configuration, you can now run yosys <bugpoint.ys>. The first thing bugpoint will do is test the input design fails. If it doesn't, make sure you are using the right yosys executable; unless the -yosys option is provided, it will use whatever the shell defaults to, *not* the current yosys. If you are using the -runner option, try replacing the bugpoint command with write_rtlil test.il and then on a new line, !<wrapper> yosys -s <failure.ys> test.il to check it works as expected and returns a non-zero status.

See also

For more on script parsing and the use of `!`, check out *Script parsing*.

Depending on the size of your design, and the length of your `<failure.ys>`, `bugpoint` may take some time; remember, it will run `yosys -s <failure.ys>` on each iteration of the design. The bigger the design, the more iterations. The longer the `<failure.ys>`, the longer each iteration will take. As the design shrinks and `bugpoint` converges, each iteration should take less and less time. Once all simplifications are exhausted and there are no more objects that can be removed, the script will continue and the minimized design can be saved.

What do I do with the minimized design?

First off, check the minimized design still fails. This is especially important if you're not using *`write_rtlil`* to output the minimized design. For example, if you ran *example bugpoint minimizer* below, then calling `yosys -s <failure.ys> min.v` should still fail in the same way.

Listing 3.70: example `bugpoint` minimizer

```
read_verilog design.v
bugpoint -script <failure.ys>
write_verilog min.v
```

The *`write_rtlil`* command is generally more reliable, since `bugpoint` will have run that exact code through the failing script. Other `write_*` commands convert from the RTLIL and then back again during the `read_*` which can result in differences which mean the design no longer fails.

Note

Simply calling Yosys with the output of `write_*`, as in `yosys -s <failure.ys> min.v`, does not guarantee that the corresponding `read_*` will be used. For more about this, refer to *Loading a design*, or load the design explicitly with `yosys -p 'read_verilog min.v' -s <failure.ys>`.

Once you've verified the failure still happens, check out *Identifying issues* for more on what to do next.

3.3.3 Minimizing Verilog designs

See also

This section is not specific to Yosys, so feel free to use another guide such as Stack Overflow's How to create a Minimal, Reproducible Example.

Be sure to check any errors or warnings for messages that might identify source lines or object names that might be causing the failure, and back up your source code before modifying it. If you have multiple source files, you should start by reducing them down to a single file. If a specific file is failing to read, try removing everything else and just focus on that one. If your source uses the `include` directive, replace it with the contents of the file referenced.

Unlike RTLIL designs where we can use `bugpoint`, Yosys does not provide any tools for minimizing Verilog designs. Instead, you should use an external tool like C-Reduce (with the `--not-c` flag) or sv-bugpoint.

C-Reduce

As a very brief overview for using C-Reduce, you want your failing source design (`test.v`), and some shell script which checks for the error being investigated (`test.sh`). Below is an *Example test.sh for C-Reduce* which uses *`logger`* and the `-expect error "<string>" 1` option to perform a similar role to `bugpoint -grep`, along with `verilator` to lint the code and make sure it is still valid.

Listing 3.71: Example test.sh for C-Reduce

```
#!/usr/bin/env bash
verilator --lint-only test.v &&/
yosys -p 'logger -expect error "unsupported" 1; read_verilog test.v'
```

Listing 3.72: input test.v

```
module top(input clk, a, b, c, output x, y, z);
   always @(posedge clk) begin
      if (a == 1'b1)
         $stop;
   end
   assign x = a;
   assign y = a ^ b;
   assign z = c;
endmodule
```

In this example `read_verilog test.v` is giving an error message that contains the string "unsupported" because the `$stop` system task is only supported in `initial` blocks. By calling `creduce ./test.sh test.v --not-c` we can minimize the design to just the failing code, while still being valid Verilog.

Listing 3.73: output test.v

```
module a;
always begin $stop;
end endmodule
```

sv-bugpoint

sv-bugpoint works quite similarly to C-Reduce, except it requires an output directory to be provided and the check script needs to accept the target file as an input argument: `sv-bugpoint outDir/ test.sh test.v`

Listing 3.74: Example test.sh for sv-bugpoint

```
#!/usr/bin/env bash
verilator --lint-only $1 &&/
yosys -p "logger -expect error \"unsupported\" 1; read_verilog $1"
```

Notice that the commands for `yosys -p` are now in double quotes (`"`), and the quotes around the error string are escaped (`\"`). This is necessary for the `$1` argument subsitution to work correctly.

Doing it manually

If for some reason you are unable to use a tool to minimize your code, you can still do it manually. But it can be a time consuming process and requires a lot of iteration. At any point in the process, you can check for anything that is unused or totally disconnected (ports, wires, etc) and remove them. If a specific module is causing the problem, try to set that as the top module instead. Any parameters should have their default values changed to match the failing usage.

As a rule of thumb, try to split things roughly in half at each step; similar to a "binary search". If you have 10 cells (instances of modules) in your top module, and have no idea what is causing the issue, split them into two groups of 5 cells. For each group of cells, try remove them and see if the failure still happens. If the error still occurs with the first group removed, but disappears when the second group is removed, then the first group can be safely removed. If a module has no more instances, remove it entirely. Repeat this for each remaining group of cells until each group only has 1 cell in it and no more cells can be removed without making the error disappear. You can also repeat this for each module still in your design.

After minimizing the number of cells, do the same for the process blocks in your top module. And again for any generate blocks and combinational blocks. Remember to check for any ports or signals which are no longer used and remove those too. Any signals which are written but never read can also be removed.

Note

Depending on where the design is failing, there are some commands which may help in identifying unused objects in the design. `hierarchy` will identify which modules are used and which are not, but check for `$paramod` modules before removing unused ones. `debug clean` will list all unused wires in each module, as well as unused cells which were automatically generated (giving the line number of the source that generated them). Adding the `-purge` flag will also include named wires that would normally be ignored by *clean*. Though when there are large numbers of unused wires it is often easier to just delete sections of the code and see what happens.

Next, try to remove or reduce assignments (`a = b`) and operations (`a + b`). A good place to start is by checking for any wires/registers which are read but never written. Try removing the signal declaration and replacing references to it with `'0` or `'x`. Do this with any constants too. Try to replace strings with numeric values, and wide signals with smaller ones, then see if the error persists.

Check if there are any operations that you can simplify, like replacing `a & '0` with `'0`. If you have enable or reset logic, try removing it and see if the error still occurs. Try reducing `if .. else` and `case` blocks to a single case. Even if that doesn't work, you may still be able to remove some paths; start with cases that appear to be unreachable and go from there.

Note

When sharing code on the Yosys GitHub, please try to keep things in English. Declarations and strings should stick to the letters a-z and numbers 0-9, unless the error is arising because of the names/characters used.

3.3.4 Identifying issues

When identifying issues, it is quite useful to understand the conditions under which the issue is occurring. While there are occasionally bugs that affect a significant number of designs, Yosys changes are tested on a variety of designs and operating systems which typically catch any such issues before they make it into the main branch. So what is is it about your situation that makes it unusual?

Note

If you have access to a different platform you could also check if your issue is reproducible there. Some issues may be specific to the platform or build of Yosys.

Try to match the minimized design back to its original context. Could you achieve the same thing a different way, and if so, does this other method have the same issue? Try to change the design in small ways and see

what happens; while `bugpoint` can reduce and simplify a design, it doesn't *change* much. What happens if you change operators, for example a left shift (or *`$shl`*) to a right shift (or *`$shr`*)? Try to see if the issue is tied to specific parameters, widths, or values.

Search the existing issues and see if someone has already made a bug report. This is where changing the design and finding the limits of what causes the failure really comes in handy. If you're more familiar with how the problem can arise, you may be able to find a related issue more easily. If an issue already exists for one case of the problem but you've found other cases, you can comment on the issue and help get it solved. If there are no existing or related issues already, then check out the steps for *Reporting bugs*.

Warning

If you are using a fuzzer to find bugs, follow the instructions for *Identifying the root cause of bugs*. **Do not** open more than one fuzzer generated issue at a time if you can not identify the root cause. If you are found to be doing this, your issues may be closed without further investigation.

3.4 Notes on Verilog support in Yosys

3.4.1 Unsupported Verilog-2005 Features

The following Verilog-2005 features are not supported by Yosys and there are currently no plans to add support for them:

- Non-synthesizable language features as defined in IEC 62142(E):2005 / IEEE Std. 1364.1(E):2002
- The `tri`, `triand` and `trior` net types
- The `config` and `disable` keywords and library map files

3.4.2 Verilog Attributes and non-standard features

- The `full_case` attribute on case statements is supported (also the non-standard `// synopsys full_case` directive)
- The `parallel_case` attribute on case statements is supported (also the non-standard `// synopsys parallel_case` directive)
- The `// synopsys translate_off` and `// synopsys translate_on` directives are also supported (but the use of `` `ifdef .. `endif `` is strongly recommended instead).
- The `nomem2reg` attribute on modules or arrays prohibits the automatic early conversion of arrays to separate registers. This is potentially dangerous. Usually the front-end has good reasons for converting an array to a list of registers. Prohibiting this step will likely result in incorrect synthesis results.
- The `mem2reg` attribute on modules or arrays forces the early conversion of arrays to separate registers.
- The `nomeminit` attribute on modules or arrays prohibits the creation of initialized memories. This effectively puts `mem2reg` on all memories that are written to in an `initial` block and are not ROMs.
- The `nolatches` attribute on modules or always-blocks prohibits the generation of logic-loops for latches. Instead all not explicitly assigned values default to x-bits. This does not affect clocked storage elements such as flip-flops.
- The `nosync` attribute on registers prohibits the generation of a storage element. The register itself will always have all bits set to 'x' (undefined). The variable may only be used as blocking assigned temporary variable within an always block. This is mostly used internally by Yosys to synthesize Verilog functions and access arrays.

- The `nowrshmsk` attribute on a register prohibits the generation of shift-and-mask type circuits for writing to bit slices of that register.
- The `onehot` attribute on wires mark them as one-hot state register. This is used for example for memory port sharing and set by the fsm_map pass.
- The `blackbox` attribute on modules is used to mark empty stub modules that have the same ports as the real thing but do not contain information on the internal configuration. This modules are only used by the synthesis passes to identify input and output ports of cells. The Verilog backend also does not output blackbox modules on default. *read_verilog*, unless called with `-noblackbox` will automatically set the blackbox attribute on any empty module it reads.
- The `noblackbox` attribute set on an empty module prevents *read_verilog* from automatically setting the blackbox attribute on the module.
- The `whitebox` attribute on modules triggers the same behavior as `blackbox`, but is for whitebox modules, i.e. library modules that contain a behavioral model of the cell type.
- The `lib_whitebox` attribute overwrites `whitebox` when *read_verilog* is run in `-lib` mode. Otherwise it's automatically removed.
- The `dynports` attribute is used by the Verilog front-end to mark modules that have ports with a width that depends on a parameter.
- The `hdlname` attribute is used by some passes to document the original (HDL) name of a module when renaming a module. It should contain a single name, or, when describing a hierarchical name in a flattened design, multiple names separated by a single space character.
- The `keep` attribute on cells and wires is used to mark objects that should never be removed by the optimizer. This is used for example for cells that have hidden connections that are not part of the netlist, such as IO pads. Setting the `keep` attribute on a module has the same effect as setting it on all instances of the module.
- The `keep_hierarchy` attribute on cells and modules keeps the `flatten` command from flattening the indicated cells and modules.
- The `gate_cost_equivalent` attribute on a module can be used to specify the estimated cost of the module as a number of basic gate instances. See the help message of command `keep_hierarchy` which interprets this attribute.
- The `init` attribute on wires is set by the frontend when a register is initialized "FPGA-style" with `reg foo = val`. It can be used during synthesis to add the necessary reset logic.
- The `top` attribute on a module marks this module as the top of the design hierarchy. The `hierarchy` command sets this attribute when called with `-top`. Other commands, such as `flatten` and various backends use this attribute to determine the top module.
- The `src` attribute is set on cells and wires created by to the string `<hdl-file-name>:<line-number>` by the HDL front-end and is then carried through the synthesis. When entities are combined, a new |-separated string is created that contains all the strings from the original entities.
- The `defaultvalue` attribute is used to store default values for module inputs. The attribute is attached to the input wire by the HDL front-end when the input is declared with a default value.
- The `parameter` and `localparam` attributes are used to mark wires that represent module parameters or localparams (when the HDL front-end is run in `-pwires` mode).
- Wires marked with the `hierconn` attribute are connected to wires with the same name (format `cell_name.identifier`) when they are imported from sub-modules by `flatten`.

- The `clkbuf_driver` attribute can be set on an output port of a blackbox module to mark it as a clock buffer output, and thus prevent `clkbufmap` from inserting another clock buffer on a net driven by such output.
- The `clkbuf_sink` attribute can be set on an input port of a module to request clock buffer insertion by the `clkbufmap` pass.
- The `clkbuf_inv` attribute can be set on an output port of a module with the value set to the name of an input port of that module. When the `clkbufmap` would otherwise insert a clock buffer on this output, it will instead try inserting the clock buffer on the input port (this is used to implement clock inverter cells that clock buffer insertion will "see through").
- The `clkbuf_inhibit` is the default attribute to set on a wire to prevent automatic clock buffer insertion by `clkbufmap`. This behaviour can be overridden by providing a custom selection to `clkbufmap`.
- The `invertible_pin` attribute can be set on a port to mark it as invertible via a cell parameter. The name of the inversion parameter is specified as the value of this attribute. The value of the inversion parameter must be of the same width as the port, with 1 indicating an inverted bit and 0 indicating a non-inverted bit.
- The `iopad_external_pin` attribute on a blackbox module's port marks it as the external-facing pin of an I/O pad, and prevents `iopadmap` from inserting another pad cell on it.
- The module attribute `abc9_lut` is an integer attribute indicating to `abc9` that this module describes a LUT with an area cost of this value, and propagation delays described using `specify` statements.
- The module attribute `abc9_box` is a boolean specifying a black/white-box definition, with propagation delays described using `specify` statements, for use by `abc9`.
- The port attribute `abc9_carry` marks the carry-in (if an input port) and carry-out (if output port) ports of a box. This information is necessary for `abc9` to preserve the integrity of carry-chains. Specifying this attribute onto a bus port will affect only its most significant bit.
- The module attribute `abc9_flop` is a boolean marking the module as a flip-flop. This allows `abc9` to analyse its contents in order to perform sequential synthesis.
- The frontend sets attributes `always_comb`, `always_latch` and `always_ff` on processes derived from SystemVerilog style always blocks according to the type of the always. These are checked for correctness in `proc_dlatch`.
- The cell attribute `wildcard_port_conns` represents wildcard port connections (SystemVerilog `.*`). These are resolved to concrete connections to matching wires in `hierarchy`.
- In addition to the `(* ... *)` attribute syntax, Yosys supports the non-standard `{* ... *}` attribute syntax to set default attributes for everything that comes after the `{* ... *}` statement. (Reset by adding an empty `{* *}` statement.)
- In module parameter and port declarations, and cell port and parameter lists, a trailing comma is ignored. This simplifies writing Verilog code generators a bit in some cases.
- Modules can be declared with `module mod_name(...);` (with three dots instead of a list of module ports). With this syntax it is sufficient to simply declare a module port as 'input' or 'output' in the module body.
- When defining a macro with `\`define`, all text between triple double quotes is interpreted as macro body, even if it contains unescaped newlines. The triple double quotes are removed from the macro body. For example:

```
`define MY_MACRO(a, b) """
  assign a = 23;
```

(continues on next page)

(continued from previous page)

```
   assign b = 42;
"""
```

- The attribute `via_celltype` can be used to implement a Verilog task or function by instantiating the specified cell type. The value is the name of the cell type to use. For functions the name of the output port can be specified by appending it to the cell type separated by a whitespace. The body of the task or function is unused in this case and can be used to specify a behavioral model of the cell type for simulation. For example:

```
module my_add3(A, B, C, Y);
  parameter WIDTH = 8;
  input [WIDTH-1:0] A, B, C;
  output [WIDTH-1:0] Y;
  ...
endmodule

module top;
  ...
  (* via_celltype = "my_add3 Y" *)
  (* via_celltype_defparam_WIDTH = 32 *)
  function [31:0] add3;
    input [31:0] A, B, C;
    begin
      add3 = A + B + C;
    end
  endfunction
  ...
endmodule
```

- The `wiretype` attribute is added by the verilog parser for wires of a typedef'd type to indicate the type identifier.
- Various `enum_value_{value}` attributes are added to wires of an enumerated type to give a map of possible enum items to their values.
- The `enum_base_type` attribute is added to enum items to indicate which enum they belong to (enums – anonymous and otherwise – are automatically named with an auto-incrementing counter). Note that enums are currently not strongly typed.
- A limited subset of DPI-C functions is supported. The plugin mechanism (see `help plugin`) can be used to load .so files with implementations of DPI-C routines. As a non-standard extension it is possible to specify a plugin alias using the `<alias>:` syntax. For example:

```
module dpitest;
  import "DPI-C" function foo:round = real my_round (real);
  parameter real r = my_round(12.345);
endmodule
```

```
$ yosys -p 'plugin -a foo -i /lib/libm.so; read_verilog dpitest.v'
```

- Sized constants (the syntax `<size>'s?[bodh]<value>`) support constant expressions as `<size>`. If the expression is not a simple identifier, it must be put in parentheses. Examples: `WIDTH'd42`, `(4+2)'b101010`
- The system tasks `$finish`, `$stop` and `$display` are supported in initial blocks in an unconditional

context (only if/case statements on expressions over parameters and constant values are allowed). The intended use for this is synthesis-time DRC.

- There is limited support for converting `specify .. endspecify` statements to special `$specify2`, `$specify3`, and `$specrule` cells, for use in blackboxes and whiteboxes. Use `read_verilog -specify` to enable this functionality. (By default these blocks are ignored.)
- The `reprocess_after` internal attribute is used by the Verilog frontend to mark cells with bindings which might depend on the specified instantiated module. Modules with such cells will be reprocessed during the `hierarchy` pass once the referenced module definition(s) become available.
- The `smtlib2_module` attribute can be set on a blackbox module to specify a formal model directly using SMT-LIB 2. For such a module, the `smtlib2_comb_expr` attribute can be used on output ports to define their value using an SMT-LIB 2 expression. For example:

```
(* blackbox *)
(* smtlib2_module *)
module submod(a, b);
  input [7:0] a;
  (* smtlib2_comb_expr = "(bvnot a)" *)
  output [7:0] b;
endmodule
```

3.4.3 Non-standard or SystemVerilog features for formal verification

- Support for `assert`, `assume`, `restrict`, and `cover` is enabled when *read_verilog* is called with `-formal`.
- The system task `$initstate` evaluates to 1 in the initial state and to 0 otherwise.
- The system function `$anyconst` evaluates to any constant value. This is equivalent to declaring a reg as `rand const`, but also works outside of checkers. (Yosys also supports `rand const` outside checkers.)
- The system function `$anyseq` evaluates to any value, possibly a different value in each cycle. This is equivalent to declaring a reg as `rand`, but also works outside of checkers. (Yosys also supports `rand` variables outside checkers.)
- The system functions `$allconst` and `$allseq` can be used to construct formal exist-forall problems. Assumptions only hold if the trace satisfies the assumption for all `$allconst/$allseq` values. For assertions and cover statements it is sufficient if just one `$allconst/$allseq` value triggers the property (similar to `$anyconst/$anyseq`).
- Wires/registers declared using the `anyconst/anyseq/allconst/allseq` attribute (for example `(* anyconst *) reg [7:0] foobar;`) will behave as if driven by a `$anyconst/$anyseq/$allconst/$allseq` function.
- The SystemVerilog tasks `$past`, `$stable`, `$rose` and `$fell` are supported in any clocked block.
- The syntax `@($global_clock)` can be used to create FFs that have no explicit clock input (`$ff` cells). The same can be achieved by using `@(posedge <netname>)` or `@(negedge <netname>)` when `<netname>` is marked with the `(* gclk *)` Verilog attribute.

3.4.4 Supported features from SystemVerilog

When *read_verilog* is called with `-sv`, it accepts some language features from SystemVerilog:

- The `assert` statement from SystemVerilog is supported in its most basic form. In module context: `assert property (<expression>);` and within an always block: `assert(<expression>);`. It is transformed to an `$assert` cell.

- The `assume`, `restrict`, and `cover` statements from SystemVerilog are also supported. The same limitations as with the `assert` statement apply.
- The keywords `always_comb`, `always_ff` and `always_latch`, `logic` and `bit` are supported.
- Declaring free variables with `rand` and `rand const` is supported.
- Checkers without a port list that do not need to be instantiated (but instead behave like a named block) are supported.
- SystemVerilog packages are supported. Once a SystemVerilog file is read into a design with *`read_verilog`*, all its packages are available to SystemVerilog files being read into the same design afterwards.
  - nested packages are currently not supported (i.e. calling `import` inside a `package` .. `endpackage` block)
- typedefs are supported (including inside packages)
  - type casts are currently not supported
- enums are supported (including inside packages)
  - but are currently not strongly typed
- packed structs and unions are supported
  - arrays of packed structs/unions are currently not supported
  - structure literals are currently not supported
- multidimensional arrays are supported
  - array assignment of unpacked arrays is currently not supported
  - array literals are currently not supported
- SystemVerilog interfaces (SVIs), including modports for specifying whether ports are inputs or outputs, are partially supported.
  - interfaces must be provided as *named* arguments, not positional arguments. i.e. `foo bar(.intf(intf0), .x(x));` is supported but `foo bar(intf0, x);` is not.
- Assignments within expressions are supported.
- The `unique`, `unique0`, and `priority` SystemVerilog keywords are supported on `if` and `case` conditionals. (The Verilog frontend will process conditionals using these keywords by annotating their representation with the appropriate `full_case` and/or `parallel_case` attributes, which are described above.)
- SystemVerilog string literals are supported (triple-quoted strings and escape sequences such as line continuations and hex escapes).

3.5 Scripting with Pyosys

Pyosys is a limited subset of the Yosys C++ API (aka "libyosys") made available using the Python programming language.

Like `.ys` and `.tcl` scripts, Pyosys provides an interface to write Yosys scripts in the Python programming language, giving you the benefits of a type system, control flow, object-oriented programming, and more; especially that the other options lack a type system and control flow/OOP in Tcl is limited.

Though unlike these two, Pyosys goes a bit further, allowing you to use the Yosys API to implement advanced functionality that would otherwise require custom passes written in C++.

3.5.1 Getting Pyosys

Pyosys supports CPython 3.8 or higher. You can access Pyosys using one of two methods:

1. Compiling Yosys with the Makefile flag `ENABLE_PYOSYS=1`

   This adds the flag `-y` to the Yosys binary, which allows you to execute Python scripts using an interpreter embedded in Yosys itself:

   ```
   yosys -y ./my_pyosys_script.py
   ```

   Do note this requires some build-time dependencies to be available to Python, namely, `pybind11` and `cxxheaderparser`. By default, the required `uv` package will be used to create an ephemeral environment with the correct versions of the tools installed.

   You can force use of your current Python environment by passing the Makefile flag `PYOSYS_USE_UV=0`.

2. Installing the Pyosys wheels

   On macOS and GNU/Linux you can install pre-built wheels of Yosys using `pip`:

   ```
   python3 -m pip install pyosys
   ```

   Which then allows you to run your scripts as follows:

   ```
   python3 ./my_pyosys_script.py
   ```

3.5.2 Scripting and Database Inspection

To start with, you have to import libyosys as follows:

```
from pyosys import libyosys
```

As a reminder, Python allows you to alias imported modules and objects, so this import may be preferable for terseness:

```
from pyosys import libyosys as ys
```

Now, scripting is actually quite similar to `.ys` and `.tcl` script in that you can provide mostly text commands. Albeit, you can construct your scripts to use Python's amenities like conditional execution, loops, and functions:

```
do_flatten = True

ys.run_pass("read_verilog tests/simple/fiedler-cooley.v")
ys.run_pass("hierarchy -check -auto-top")
if do_flatten:
   ys.run_pass("flatten")
```

. . . but this does not strictly provide anything that Tcl scripts do not provide you with. The real power of using Pyosys comes from the fact you can manually instantiate, manage, and interact with the design database.

As an example, here is the same script with a manually instantiated design.

```
design = ys.Design()

ys.run_pass("read_verilog tests/simple/fiedler-cooley.v", design)
ys.run_pass("hierarchy -check -auto-top", design)
```

What's new here is that you can manually inspect the design's database. This gives you access to a huge chunk of the design database API as declared in the `kernel/rtlil.h` header.

For example, here's how to list the input and output ports of the top module of your design:

```
top_module = design.top_module()

for id, wire in top_module.wires_.items():
        if not wire.port_input and not wire.port_output:
                continue
        description = "input" if wire.port_input else "output"
        description += " " + wire.name.str()
        if wire.width != 1:
                frm = wire.start_offset
                to = wire.start_offset + wire.width
                if wire.upto:
                        to, frm = frm, to
                description += f" [{to}:{frm}]"
        print(description)
```

Tip

C++ data structures in Yosys are bridged to Python such that they have a pretty similar API to Python objects, for example:

- `std::vector` supports the same methods as iterables in Python.
- `std::set` and hashlib `pool` support the same methods as `set`s in Python. While `set` is ordered, `pool` is not and modifications may cause a complete reordering of the set.
- `dict` supports the same methods as `dict`s in Python, albeit it is unordered, and modifications may cause a complete reordering of the dictionary.
- `idict` uses a custom set of methods because it doesn't map very cleanly to an existing Python data structure. See `pyosys/hashlib.h` for more info.

For most operations, the Python equivalents are also supported as arguments where they will automatically be cast to the right type, so you do not have to manually instantiate the right underlying C++ object(s) yourself.

3.5.3 Modifying the Database

Warning

Any modifications to the database may invalidate previous references held by Python, just as if you were writing C++. Pyosys does not currently attempt to keep deleted objects alive if a reference is held by Python.

You are not restricted to inspecting the database either: you have the ability to modify it, and introduce new elements and/or changes to your design.

As a demonstrative example, let's assume we want to add an enable line to all flip-flops in our fiedler-cooley design.

First of all, we will run `synth` to convert all of the logic to Yosys's internal cell structure (see *Gate-level cells*):

```
ys.run_pass("synth", design)
```

Next, we need to add the new port. The method for this is `Module::addWire`.

Tip

IdString is Yosys's internal representation of strings used as identifiers within Verilog designs. They are efficient as only integers are stored and passed around, but they can be translated to and from normal strings at will.

Pyosys will automatically cast Python strings to IdStrings for you, but the rules around IdStrings apply, namely that *broadly*:

- Identifiers for internal cells must start with `$`.
- All other identifiers must start with `\`.

```
enable_line = top_module.addWire("\\enable")
enable_line.port_input = True
top_module.fixup_ports()
```

Notice how we modified the wire then called a method to make Yosys re-process the ports.

Next, we can iterate over all constituent cells, and if they are of the type `$_DFF_P_`, we do two things:

1. Change their type to `$_DFFE_PP_` to enable hooking up an enable signal.
2. Hooking up the enable signal.

```
for cell in top_module.cells_.values():
        if cell.type != "$_DFF_P_":
                continue
        cell.type = "$_DFFE_PP_"
        cell.setPort("\\E", ys.SigSpec(enable_line))
```

To verify that you did everything correctly, it is prudent to call `.check()` on the module you're manipulating as follows after you're done with a set of changes:

```
top_module.check()
ys.run_pass("stat", design)
```

And then finally, write your outputs. Here, I choose an intermediate Verilog file and `synth_ice40` to map it to the iCE40 architecture.

```
ys.run_pass("write_verilog out.v", design)
ys.run_pass("synth_ice40 -json out.json", design)
```

And voilà, you will note that in the intermediate output, all `always @` statements should have an `if (enable)`.

3.5.4 Encapsulating as Passes

Just like when writing C++, you can encapsulate routines in terms of "passes", which adds your Pass to a global registry of commands accessible using `run_pass`.

```
from pyosys import libyosys as ys

class AllEnablePass(ys.Pass):
        def __init__(self):
                super().__init__(
                        "all_enable",
                        "makes all _DFF_P_ registers require an enable signal"
                )

        def execute(self, args, design):
                ys.log_header(design, "Adding enable signals\n")
                ys.log_push()
                top_module = design.top_module()

                if "\\enable" not in top_module.wires_:
                        enable_line = top_module.addWire("\\enable")
                        enable_line.port_input = True
                        top_module.fixup_ports()

                for cell in top_module.cells_.values():
                        if cell.type != "$_DFF_P_":
                                continue
                        cell.type = "$_DFFE_PP_"
                        cell.setPort("\\E", ys.SigSpec(enable_line))
                ys.log_pop()

p = AllEnablePass() # register the pass

# using the pass

design = ys.Design()
ys.run_pass("read_verilog tests/simple/fiedler-cooley.v", design)
ys.run_pass("hierarchy -check -auto-top", design)
ys.run_pass("synth", design)
ys.run_pass("all_enable", design)
ys.run_pass("write_verilog out.v", design)
ys.run_pass("synth_ice40 -json out.json", design)
```

In general, abstract classes and virtual methods are not really supported by Pyosys due to their complexity, but there are two exceptions which are:

- `Pass` in `kernel/register.h`

- `Monitor` in `kernel/rtlil.h`

CHAPTER

FOUR

YOSYS INTERNALS

Todo

less academic

Yosys is an extensible open source hardware synthesis tool. It is aimed at designers who are looking for an easily accessible, universal, and vendor-independent synthesis tool, as well as scientists who do research in electronic design automation (EDA) and are looking for an open synthesis framework that can be used to test algorithms on complex real-world designs.

Yosys can synthesize a large subset of Verilog 2005 and has been tested with a wide range of real-world designs, including the OpenRISC 1200 CPU, the openMSP430 CPU, the OpenCores I2C master, and the k68 CPU.

Todo

add RISC-V core example

Yosys is written in C++, targeting C++20 at minimum. This chapter describes some of the fundamental Yosys data structures. For the sake of simplicity the C++ type names used in the Yosys implementation are used in this chapter, even though the chapter only explains the conceptual idea behind it and can be used as reference to implement a similar system in any language.

4.1 Internal flow

A (usually short) synthesis script controls Yosys.

These scripts contain three types of commands:

- **Frontends**, that read input files (usually Verilog);
- **Passes**, that perform transformations on the design in memory;
- **Backends**, that write the design in memory to a file (various formats are available: Verilog, BLIF, EDIF, SPICE, BTOR, ...).

4.1.1 Flow overview

Todo

less academic

Figure 4.1 shows the simplified data flow within Yosys. Rectangles in the figure represent program modules and ellipses internal data structures that are used to exchange design data between the program modules.

Design data is read in using one of the frontend modules. The high-level HDL frontends for Verilog and VHDL code generate an abstract syntax tree (AST) that is then passed to the AST frontend. Note that both HDL frontends use the same AST representation that is powerful enough to cover the Verilog HDL and VHDL language.

The AST Frontend then compiles the AST to Yosys's main internal data format, the RTL Intermediate Language (RTLIL). A more detailed description of this format is given in *The RTL Intermediate Language (RTLIL)*.

There is also a text representation of the RTLIL data structure that can be parsed using the RTLIL Frontend which is described in *RTLIL text representation*.

The design data may then be transformed using a series of passes that all operate on the RTLIL representation of the design.

Finally the design in RTLIL representation is converted back to text by one of the backends, namely the Verilog Backend for generating Verilog netlists and the RTLIL Backend for writing the RTLIL data in the same format that is understood by the RTLIL Frontend.

With the exception of the AST Frontend, which is called by the high-level HDL frontends and can't be called directly by the user, all program modules are called by the user (usually using a synthesis script that contains text commands for Yosys).

By combining passes in different ways and/or adding additional passes to Yosys it is possible to adapt Yosys to a wide range of applications. For this to be possible it is key that (1) all passes operate on the same data structure (RTLIL) and (2) that this data structure is powerful enough to represent the design in different stages of the synthesis.

4.1.2 Control and data flow

Todo

less academic

The data- and control-flow of a typical synthesis tool is very similar to the data- and control-flow of a typical compiler: different subsystems are called in a predetermined order, each consuming the data generated by the last subsystem and generating the data for the next subsystem (see Fig. 4.2).

The first subsystem to be called is usually called a frontend. It does not process the data generated by another subsystem but instead reads the user input—in the case of a HDL synthesis tool, the behavioural HDL code.

The subsystems that consume data from previous subsystems and produce data for the next subsystems (usually in the same or a similar format) are called passes.

The last subsystem that is executed transforms the data generated by the last pass into a suitable output format and writes it to a disk file. This subsystem is usually called the backend.

In Yosys all frontends, passes and backends are directly available as commands in the synthesis script. Thus the user can easily create a custom synthesis flow just by calling passes in the right order in a synthesis

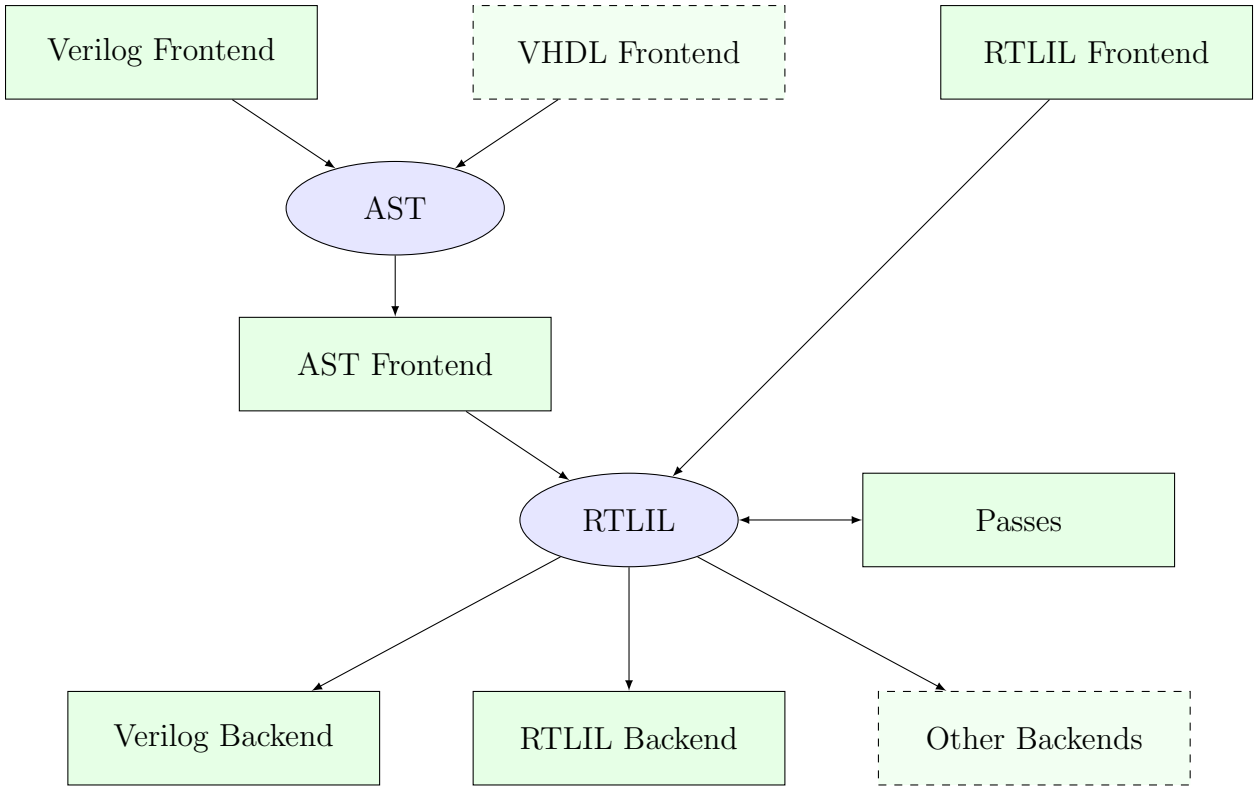


Fig. 4.1: Yosys simplified data flow (ellipses: data structures, rectangles: program modules)

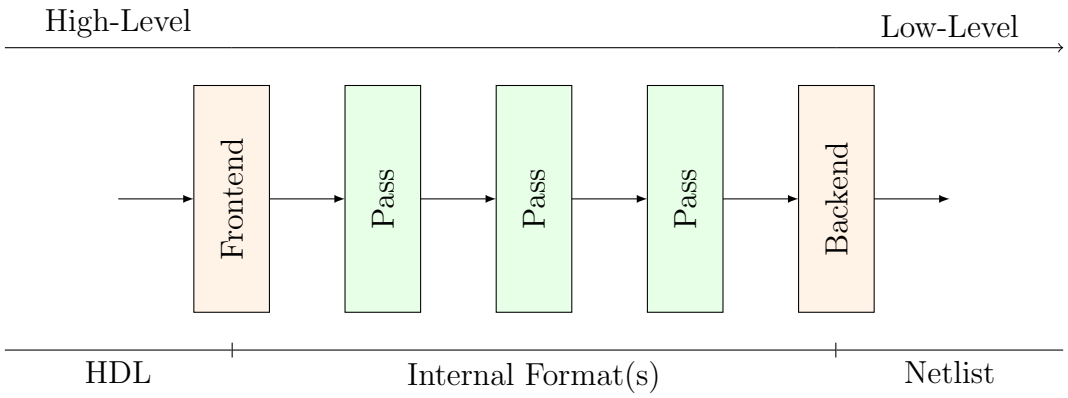


Fig. 4.2: General data- and control-flow of a synthesis tool

script.

4.1.3 The Verilog and AST frontends

This chapter provides an overview of the implementation of the Yosys Verilog and AST frontends. The Verilog frontend reads Verilog-2005 code and creates an abstract syntax tree (AST) representation of the input. This AST representation is then passed to the AST frontend that converts it to RTLIL data, as illustrated in Fig. 4.3.

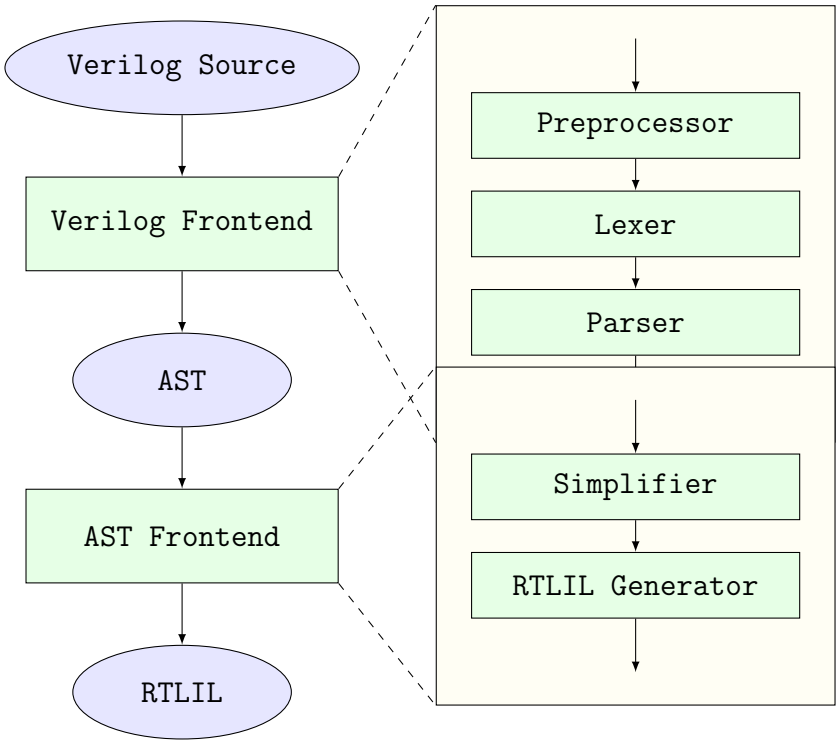


Fig. 4.3: Simplified Verilog to RTLIL data flow

Transforming Verilog to AST

The Verilog frontend converts the Verilog sources to an internal AST representation that closely resembles the structure of the original Verilog code. The Verilog frontend consists of three components, the Preprocessor, the Lexer and the Parser.

The source code to the Verilog frontend can be found in `frontends/verilog/` in the Yosys source tree.

The Verilog preprocessor

The Verilog preprocessor scans over the Verilog source code and interprets some of the Verilog compiler directives such as `` `include ``, `` `define `` and `` `ifdef ``.

It is implemented as a C++ function that is passed a file descriptor as input and returns the pre-processed Verilog code as a `std::string`.

The source code to the Verilog Preprocessor can be found in `frontends/verilog/preproc.cc` in the Yosys source tree.

The Verilog lexer

The Verilog Lexer is written using the lexer generator flex. Its source code can be found in `frontends/verilog/verilog_lexer.l` in the Yosys source tree. The lexer does little more than identifying all keywords and literals recognised by the Yosys Verilog frontend.

The lexer keeps track of the current location in the Verilog source code with a `VerilogLexer::out_loc` and uses it to construct parser-defined symbol objects.

Finally the lexer identifies and handles special comments such as "`// synopsys translate_off`" and "`// synopsys full_case`". (It is recommended to use `` `ifdef `` constructs instead of the Synsopsys translate_on/off comments and attributes such as `(* full_case *)` over "`// synopsys full_case`" whenever possible.)

The Verilog parser

The Verilog Parser is written using the parser generator bison. Its source code can be found in `frontends/verilog/verilog_parser.y` in the Yosys source tree.

It generates an AST using the `AST::AstNode` data structure defined in `frontends/ast/ast.h`. An `AST::AstNode` object has the following properties:

Table 4.1: AST node types with their corresponding Verilog constructs.

AST Node Type	Corresponding Verilog Construct
AST_NONE	This Node type should never be used.
AST_DESIGN	This node type is used for the top node of the AST tree. It has no corresponding Verilog construct.
AST_MODULE, AST_TASK, AST_FUNCTION	`module`, `task` and `function`
AST_WIRE	`input`, `output`, `wire`, `reg` and `integer`
AST_MEMORY	Verilog Arrays
AST_AUTOWIRE	Created by the simplifier when an undeclared signal name is used.
AST_PARAMETER, AST_LOCALPARAM	`parameter` and `localparam`
AST_PARASET	Parameter set in cell instantiation
AST_ARGUMENT	Port connection in cell instantiation
AST_RANGE	Bit-Index in a signal or element index in array
AST_CONSTANT	A literal value
AST_CELLTYPE	The type of cell in cell instantiation
AST_IDENTIFIER	An Identifier (signal name in expression or cell/task/etc. name in other contexts)
AST_PREFIX	Construct an identifier in the form <prefix>[<index>].<suffix> (used only in advanced generate constructs)
AST_FCALL, AST_TCALL	Call to function or task
AST_TO_SIGNED, AST_TO_UNSIGNED	The `$signed()` and `$unsigned()` functions
AST_CONCAT, AST_REPLICATE	The `{...}` and `{...{...}}` operators
AST_BIT_NOT, AST_BIT_AND, AST_BIT_OR, AST_BIT_XOR, AST_BIT_XNOR	The bitwise operators `~`, `&`, `|`, `^` and `~^`
AST_REDUCE_AND, AST_REDUCE_OR, AST_REDUCE_XOR, AST_REDUCE_XNOR	The unary reduction operators `~`, `&`, `|`, `^` and `~^`
AST_REDUCE_BOOL	Conversion from multi-bit value to boolean value (equivalent to AST_REDUCE_OR)

continues on next page

Table 4.1 – continued from previous page

AST_SHIFT_LEFT, AST_SHIFT_RIGHT, AST_SHIFT_SLEFT, AST_SHIFT_SRIGHT	The shift operators `<<`, `>>`, `<<<` and `>>>`
AST_LT, AST_LE, AST_EQ, AST_NE, AST_GE, AST_GT	The relational operators `<`, `<=`, `==`, `!=`, `>=` and `>`
AST_ADD, AST_SUB, AST_MUL, AST_DIV, AST_MOD, AST_POW	The binary operators `+`, `-`, `*`, `/`, `%` and `**`
AST_POS, AST_NEG	The prefix operators `+` and `-`
AST_LOGIC_AND, AST_LOGIC_OR, AST_LOGIC_NOT	The logic operators `&&`, `||` and `!`
AST_TERNARY	The ternary `?:`-operator
AST_MEMRD AST_MEMWR	Read and write memories. These nodes are generated by the AST simplifier for writes/reads to/from Verilog arrays.
AST_ASSIGN	An `assign` statement
AST_CELL	A cell instantiation
AST_PRIMITIVE	A primitive cell (`and`, `nand`, `or`, etc.)
AST_ALWAYS, AST_INITIAL	Verilog `always`- and `initial`-blocks
AST_BLOCK	A `begin`-`end`-block
AST_ASSIGN_EQ. AST_ASSIGN_LE	Blocking (`=`) and nonblocking (`<=`) assignments within an `always`- or `initial`-block
AST_CASE. AST_COND, AST_DEFAULT	The `case` (`if`) statements, conditions within a case and the default case respectively
AST_FOR	A `for`-loop with an `always`- or `initial`-block
AST_GENVAR, AST_GENBLOCK, AST_GENFOR, AST_GENIF	The `genvar` and `generate` keywords and `for` and `if` within a generate block.
AST_POSEDGE, AST_NEGEDGE, AST_EDGE	Event conditions for `always` blocks.

- The node type
  This enum (`AST::AstNodeType`) specifies the role of the node. Table 4.1 contains a list of all node types.
- The child nodes
  This is a list of pointers to all children in the abstract syntax tree.
- Attributes
  As almost every AST node might have Verilog attributes assigned to it, the `AST::AstNode` has direct support for attributes. Note that the attribute values are again AST nodes.
- Node content
  Each node might have additional content data. A series of member variables exist to hold such data. For example the member `std::string str` can hold a string value and is used e.g. in the `AST_IDENTIFIER` node type to store the identifier name.
- Source code location
  Each `AST::AstNode` is automatically annotated with the current source code location by the `AST::AstNode` constructor. The `location` type is a manual reimplementation of the bison-provided location type. This type is defined at `frontends/verilog/verilog_location.h`.

The `AST::AstNode` constructor can be called with up to 4 child nodes. This simplifies the creation of AST nodes for simple expressions a bit. For example the bison code for parsing multiplications:

```
1 basic_expr TOK_ASTER attr basic_expr {
2   $$ = std::make_unique<AstNode>(AST_MUL, std::move($1), std::move($4));
```

(continues on next page)

(continued from previous page)

```
  SET_AST_NODE_LOC($$.get(), @1, @4);
  append_attr($$.get(), $3);
} |
```

The generated AST data structure is then passed directly to the AST frontend that performs the actual conversion to RTLIL.

Note that the Yosys command `read_verilog` provides the options `-yydebug` and `-dump_ast` that can be used to print the parse tree or abstract syntax tree respectively.

Transforming AST to RTLIL

The AST frontend converts a set of modules in AST representation to modules in RTLIL representation and adds them to the current design. This is done in two steps: simplification and RTLIL generation.

The source code to the AST frontend can be found in `frontends/ast/` in the Yosys source tree.

AST simplification

A full-featured AST is too complex to be transformed into RTLIL directly. Therefore it must first be brought into a simpler form. This is done by calling the `AST::AstNode::simplify()` method of all `AST_MODULE` nodes in the AST. This initiates a recursive process that performs the following transformations on the AST data structure:

- Inline all task and function calls.
- Evaluate all `generate`-statements and unroll all `for`-loops.
- Perform const folding where it is necessary (e.g. in the value part of `AST_PARAMETER`, `AST_LOCALPARAM`, `AST_PARASET` and `AST_RANGE` nodes).
- Replace `AST_PRIMITIVE` nodes with appropriate `AST_ASSIGN` nodes.
- Replace dynamic bit ranges in the left-hand-side of assignments with `AST_CASE` nodes with `AST_COND` children for each possible case.
- Detect array access patterns that are too complicated for the `RTLIL::Memory` abstraction and replace them with a set of signals and cases for all reads and/or writes.
- Otherwise replace array accesses with `AST_MEMRD` and `AST_MEMWR` nodes.

In addition to these transformations, the simplifier also annotates the AST with additional information that is needed for the RTLIL generator, namely:

- All ranges (width of signals and bit selections) are not only const folded but (when a constant value is found) are also written to member variables in the AST_RANGE node.
- All identifiers are resolved and all `AST_IDENTIFIER` nodes are annotated with a pointer to the AST node that contains the declaration of the identifier. If no declaration has been found, an `AST_AUTOWIRE` node is created and used for the annotation.

This produces an AST that is fairly easy to convert to the RTLIL format.

Generating RTLIL

After AST simplification, the `AST::AstNode::genRTLIL()` method of each `AST_MODULE` node in the AST is called. This initiates a recursive process that generates equivalent RTLIL data for the AST data.

The `AST::AstNode::genRTLIL()` method returns an `RTLIL::SigSpec` structure. For nodes that represent expressions (operators, constants, signals, etc.), the cells needed to implement the calculation described by

the expression are created and the resulting signal is returned. That way it is easy to generate the circuits for large expressions using depth-first recursion. For nodes that do not represent an expression (such as `AST_CELL`), the corresponding circuit is generated and an empty `RTLIL::SigSpec` is returned.

Synthesizing Verilog always blocks

For behavioural Verilog code (code utilizing `always`- and `initial`-blocks) it is necessary to also generate `RTLIL::Process` objects. This is done in the following way:

Whenever `AST::AstNode::genRTLIL()` encounters an `always`- or `initial`-block, it creates an instance of `AST_INTERNAL::ProcessGenerator`. This object then generates the `RTLIL::Process` object for the block. It also calls `AST::AstNode::genRTLIL()` for all right-hand-side expressions contained within the block.

First the `AST_INTERNAL::ProcessGenerator` creates a list of all signals assigned within the block. It then creates a set of temporary signals using the naming scheme `$ <number> \ <original_name>` for each of the assigned signals.

Then an `RTLIL::Process` is created that assigns all intermediate values for each left-hand-side signal to the temporary signal in its `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree.

Finally a `RTLIL::SyncRule` is created for the `RTLIL::Process` that assigns the temporary signals for the final values to the actual signals.

A process may also contain memory writes. A `RTLIL::MemWriteAction` is created for each of them.

Calls to `AST::AstNode::genRTLIL()` are generated for right hand sides as needed. When blocking assignments are used, `AST::AstNode::genRTLIL()` is configured using global variables to use the temporary signals that hold the correct intermediate values whenever one of the previously assigned signals is used in an expression.

Unfortunately the generation of a correct `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree for behavioural code is a non-trivial task. The AST frontend solves the problem using the approach described on the following pages. The following example illustrates what the algorithm is supposed to do. Consider the following Verilog code:

```
always @(posedge clock) begin
    out1 = in1;
    if (in2)
        out1 = !out1;
    out2 <= out1;
    if (in3)
        out2 <= out2;
    if (in4)
        if (in5)
            out3 <= in6;
        else
            out3 <= in7;
    out1 = out1 ^ out2;
end
```

This is translated by the Verilog and AST frontends into the following RTLIL code (attributes, cell parameters and wire declarations not included):

```
cell $logic_not $logic_not$<input>:4$2
  connect \A \in1
  connect \Y $logic_not$<input>:4$2_Y
end
```

(continues on next page)

(continued from previous page)

```
cell $xor $xor$<input>:13$3
  connect \A $1\out1[0:0]
  connect \B \out2
  connect \Y $xor$<input>:13$3_Y
end
process $proc$<input>:1$1
  assign $0\out3[0:0] \out3
  assign $0\out2[0:0] $1\out1[0:0]
  assign $0\out1[0:0] $xor$<input>:13$3_Y
  switch \in2
    case 1'1
      assign $1\out1[0:0] $logic_not$<input>:4$2_Y
    case
      assign $1\out1[0:0] \in1
  end
  switch \in3
    case 1'1
      assign $0\out2[0:0] \out2
    case
  end
  switch \in4
    case 1'1
      switch \in5
        case 1'1
          assign $0\out3[0:0] \in6
        case
          assign $0\out3[0:0] \in7
      end
    case
  end
  sync posedge \clock
    update \out1 $0\out1[0:0]
    update \out2 $0\out2[0:0]
    update \out3 $0\out3[0:0]
end
```

Note that the two operators are translated into separate cells outside the generated process. The signal `out1` is assigned using blocking assignments and therefore `out1` has been replaced with a different signal in all expressions after the initial assignment. The signal `out2` is assigned using nonblocking assignments and therefore is not substituted on the right-hand-side expressions.

The `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree must be interpreted the following way:

- On each case level (the body of the process is the root case), first the actions on this level are evaluated and then the switches within the case are evaluated. (Note that the last assignment on line 13 of the Verilog code has been moved to the beginning of the RTLIL process to line 13 of the RTLIL listing.)

  I.e. the special cases deeper in the switch hierarchy override the defaults on the upper levels. The assignments in lines 12 and 22 of the RTLIL code serve as an example for this.

  Note that in contrast to this, the order within the `RTLIL::SwitchRule` objects within a `RTLIL::CaseRule` is preserved with respect to the original AST and Verilog code.

- The whole `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree describes an asynchronous circuit. I.e. the decision tree formed by the switches can be seen independently for each assigned signal. Whenever one

assigned signal changes, all signals that depend on the changed signals are to be updated. For example the assignments in lines 16 and 18 in the RTLIL code in fact influence the assignment in line 12, even though they are in the "wrong order".

The only synchronous part of the process is in the `RTLIL::SyncRule` object generated at line 35 in the RTLIL code. The sync rule is the only part of the process where the original signals are assigned. The synchronization event from the original Verilog code has been translated into the synchronization type (posedge) and signal (`\clock`) for the `RTLIL::SyncRule` object. In the case of this simple example the `RTLIL::SyncRule` object is later simply transformed into a set of d-type flip-flops and the `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree to a decision tree using multiplexers.

In more complex examples (e.g. asynchronous resets) the part of the `RTLIL::CaseRule`/`RTLIL::SwitchRule` tree that describes the asynchronous reset must first be transformed to the correct `RTLIL::SyncRule` objects. This is done by the `proc_arst` pass.

The ProcessGenerator algorithm

The `AST_INTERNAL::ProcessGenerator` uses the following internal state variables:

- `subst_rvalue_from` and `subst_rvalue_to`
  These two variables hold the replacement pattern that should be used by `AST::AstNode::genRTLIL()` for signals with blocking assignments. After initialization of `AST_INTERNAL::ProcessGenerator` these two variables are empty.
- `subst_lvalue_from` and `subst_lvalue_to`
  These two variables contain the mapping from left-hand-side signals (`\ <name>`) to the current temporary signal for the same thing (initially `$0\ <name>`).
- `current_case`
  A pointer to a `RTLIL::CaseRule` object. Initially this is the root case of the generated `RTLIL::Process`.

As the algorithm runs these variables are continuously modified as well as pushed to the stack and later restored to their earlier values by popping from the stack.

On startup the ProcessGenerator generates a new `RTLIL::Process` object with an empty root case and initializes its state variables as described above. Then the `RTLIL::SyncRule` objects are created using the synchronization events from the AST_ALWAYS node and the initial values of `subst_lvalue_from` and `subst_lvalue_to`. Then the AST for this process is evaluated recursively.

During this recursive evaluation, three different relevant types of AST nodes can be discovered: `AST_ASSIGN_LE` (nonblocking assignments), `AST_ASSIGN_EQ` (blocking assignments) and `AST_CASE` (`if` or `case` statement).

Handling of nonblocking assignments

When an `AST_ASSIGN_LE` node is discovered, the following actions are performed by the ProcessGenerator:

- The left-hand-side is evaluated using `AST::AstNode::genRTLIL()` and mapped to a temporary signal name using `subst_lvalue_from` and `subst_lvalue_to`.
- The right-hand-side is evaluated using `AST::AstNode::genRTLIL()`. For this call, the values of `subst_rvalue_from` and `subst_rvalue_to` are used to map blocking-assigned signals correctly.
- Remove all assignments to the same left-hand-side as this assignment from the `current_case` and all cases within it.
- Add the new assignment to the `current_case`.

Handling of blocking assignments

When an AST_ASSIGN_EQ node is discovered, the following actions are performed by the ProcessGenerator:

- Perform all the steps that would be performed for a nonblocking assignment (see above).
- Remove the found left-hand-side (before lvalue mapping) from subst_rvalue_from and also remove the respective bits from subst_rvalue_to.
- Append the found left-hand-side (before lvalue mapping) to subst_rvalue_from and append the found right-hand-side to subst_rvalue_to.

Handling of cases and if-statements

When an AST_CASE node is discovered, the following actions are performed by the ProcessGenerator:

- The values of subst_rvalue_from, subst_rvalue_to, subst_lvalue_from and subst_lvalue_to are pushed to the stack.
- A new RTLIL::SwitchRule object is generated, the selection expression is evaluated using AST::AstNode::genRTLIL() (with the use of subst_rvalue_from and subst_rvalue_to) and added to the RTLIL::SwitchRule object and the object is added to the current_case.
- All lvalues assigned to within the AST_CASE node using blocking assignments are collected and saved in the local variable this_case_eq_lvalue.
- New temporary signals are generated for all signals in this_case_eq_lvalue and stored in this_case_eq_ltemp.
- The signals in this_case_eq_lvalue are mapped using subst_rvalue_from and subst_rvalue_to and the resulting set of signals is stored in this_case_eq_rvalue.

Then the following steps are performed for each AST_COND node within the AST_CASE node:

- Set subst_rvalue_from, subst_rvalue_to, subst_lvalue_from and subst_lvalue_to to the values that have been pushed to the stack.
- Remove this_case_eq_lvalue from subst_lvalue_from/subst_lvalue_to.
- Append this_case_eq_lvalue to subst_lvalue_from and append this_case_eq_ltemp to subst_lvalue_to.
- Push the value of current_case.
- Create a new RTLIL::CaseRule. Set current_case to the new object and add the new object to the RTLIL::SwitchRule created above.
- Add an assignment from this_case_eq_rvalue to this_case_eq_ltemp to the new current_case.
- Evaluate the compare value for this case using AST::AstNode::genRTLIL() (with the use of subst_rvalue_from and subst_rvalue_to) modify the new current_case accordingly.
- Recursion into the children of the AST_COND node.
- Restore current_case by popping the old value from the stack.

Finally the following steps are performed:

- The values of subst_rvalue_from, subst_rvalue_to, subst_lvalue_from and subst_lvalue_to are popped from the stack.
- The signals from this_case_eq_lvalue are removed from the subst_rvalue_from/subst_rvalue_to-pair.

- The value of `this_case_eq_lvalue` is appended to `subst_rvalue_from` and the value of `this_case_eq_ltemp` is appended to `subst_rvalue_to`.
- Map the signals in `this_case_eq_lvalue` using `subst_lvalue_from`/`subst_lvalue_to`.
- Remove all assignments to signals in `this_case_eq_lvalue` in `current_case` and all cases within it.
- Add an assignment from `this_case_eq_ltemp` to `this_case_eq_lvalue` to `current_case`.

Further analysis of the algorithm for cases and if-statements

With respect to nonblocking assignments the algorithm is easy: later assignments invalidate earlier assignments. For each signal assigned using nonblocking assignments exactly one temporary variable is generated (with the `$0`-prefix) and this variable is used for all assignments of the variable.

Note how all the `_eq_`-variables become empty when no blocking assignments are used and many of the steps in the algorithm can then be ignored as a result of this.

For a variable with blocking assignments the algorithm shows the following behaviour: First a new temporary variable is created. This new temporary variable is then registered as the assignment target for all assignments for this variable within the cases for this `AST_CASE` node. Then for each case the new temporary variable is first assigned the old temporary variable. This assignment is overwritten if the variable is actually assigned in this case and is kept as a default value otherwise.

This yields an `RTLIL::CaseRule` that assigns the new temporary variable in all branches. So when all cases have been processed a final assignment is added to the containing block that assigns the new temporary variable to the old one. Note how this step always overrides a previous assignment to the old temporary variable. Other than nonblocking assignments, the old assignment could still have an effect somewhere in the design, as there have been calls to `AST::AstNode::genRTLIL()` with a `subst_rvalue_from`/`subst_rvalue_to`-tuple that contained the right-hand-side of the old assignment.

The proc pass

The ProcessGenerator converts a behavioural model in AST representation to a behavioural model in `RTLIL::Process` representation. The actual conversion from a behavioural model to an RTL representation is performed by the *proc* pass and the passes it launches:

- *proc_clean* and *proc_rmdead*
  These two passes just clean up the `RTLIL::Process` structure. The *proc_clean* pass removes empty parts (eg. empty assignments) from the process and *proc_rmdead* detects and removes unreachable branches from the process's decision trees.
- *proc_arst*
  This pass detects processes that describe d-type flip-flops with asynchronous resets and rewrites the process to better reflect what they are modelling: Before this pass, an asynchronous reset has two edge-sensitive sync rules and one top-level `RTLIL::SwitchRule` for the reset path. After this pass the sync rule for the reset is level-sensitive and the top-level `RTLIL::SwitchRule` has been removed.
- *proc_mux*
  This pass converts the `RTLIL::CaseRule`/`RTLIL::SwitchRule`-tree to a tree of multiplexers per written signal. After this, the `RTLIL::Process` structure only contains the `RTLIL::SyncRule` s that describe the output registers.
- *proc_dff*
  This pass replaces the `RTLIL::SyncRule`s to d-type flip-flops (with asynchronous resets if necessary).
- *proc_memwr*
  This pass replaces the `RTLIL::MemWriteAction`s with *$memwr* cells.

- *proc_clean*
  A final call to *proc_clean* removes the now empty `RTLIL::Process` objects.

Performing these last processing steps in passes instead of in the Verilog frontend has two important benefits:

First it improves the transparency of the process. Everything that happens in a separate pass is easier to debug, as the RTLIL data structures can be easily investigated before and after each of the steps.

Second it improves flexibility. This scheme can easily be extended to support other types of storage-elements, such as sr-latches or d-latches, without having to extend the actual Verilog frontend.

Todo

Synthesizing Verilog arrays

Add some information on the generation of *$memrd* and *$memwr* cells and how they are processed in the memory pass.

Todo

Synthesizing parametric designs

Add some information on the `RTLIL::Module::derive()` method and how it is used to synthesize parametric modules via the hierarchy pass.

4.2 Internal formats

Yosys uses two different internal formats. The first is used to store an abstract syntax tree (AST) of a Verilog input file. This format is simply called AST and is generated by the Verilog Frontend. This data structure is consumed by a subsystem called AST Frontend[1]. This AST Frontend then generates a design in Yosys' main internal format, the Register-Transfer-Level-Intermediate-Language (RTLIL) representation. It does that by first performing a number of simplifications within the AST representation and then generating RTLIL from the simplified AST data structure.

The RTLIL representation is used by all passes as input and outputs. This has the following advantages over using different representational formats between different passes:

- The passes can be rearranged in a different order and passes can be removed or inserted.
- Passes can simply pass-thru the parts of the design they don't change without the need to convert between formats. In fact Yosys passes output the same data structure they received as input and performs all changes in place.
- All passes use the same interface, thus reducing the effort required to understand a pass when reading the Yosys source code, e.g. when adding additional features.

The RTLIL representation is basically a netlist representation with the following additional features:

- An internal cell library with fixed-function cells to represent RTL datapath and register cells as well as logical gate-level cells (single-bit gates and registers).
- Support for multi-bit values that can use individual bits from wires as well as constant bits to represent coarse-grain netlists.
- Support for basic behavioural constructs (if-then-else structures and multi-case switches with a sensitivity list for updating the outputs).

[1] In Yosys the term pass is only used to refer to commands that operate on the RTLIL data structure.

- Support for multi-port memories.

The use of RTLIL also has the disadvantage of having a very powerful format between all passes, even when doing gate-level synthesis where the more advanced features are not needed. In order to reduce complexity for passes that operate on a low-level representation, these passes check the features used in the input RTLIL and fail to run when unsupported high-level constructs are used. In such cases a pass that transforms the higher-level constructs to lower-level constructs must be called from the synthesis script first.

4.2.1 The RTL Intermediate Language (RTLIL)

All frontends, passes and backends in Yosys operate on a design in RTLIL representation. The only exception are the high-level frontends that use the AST representation as an intermediate step before generating RTLIL data.

In order to avoid reinventing names for the RTLIL classes, they are simply referred to by their full C++ name, i.e. including the `RTLIL::` namespace prefix, in this document.

Figure 4.4 shows a simplified Entity-Relationship Diagram (ER Diagram) of RTLIL. In $1 : N$ relationships the arrow points from the N side to the 1. For example one `RTLIL::Design` contains N (zero to many) instances of `RTLIL::Module` . A two-pointed arrow indicates a $1 : 1$ relationship.

The `RTLIL::Design` is the root object of the RTLIL data structure. There is always one "current design" in memory which passes operate on, frontends add data to and backends convert to exportable formats. But in some cases passes internally generate additional `RTLIL::Design` objects. For example when a pass is reading an auxiliary Verilog file such as a cell library, it might create an additional `RTLIL::Design` object and call the Verilog frontend with this other object to parse the cell library.

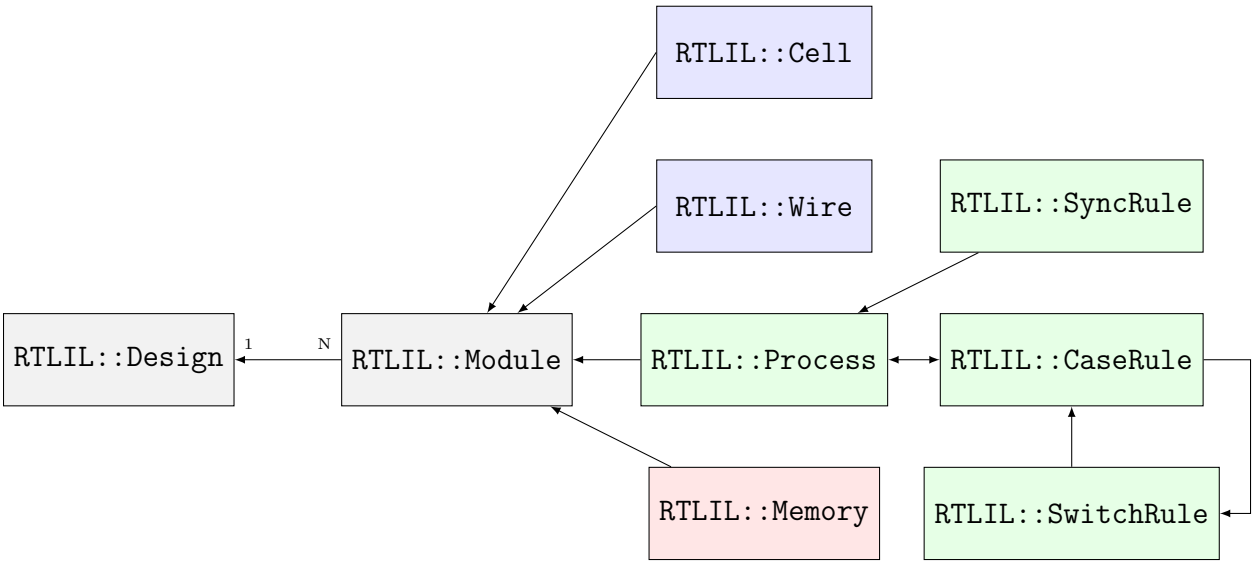


Fig. 4.4: Simplified RTLIL Entity-Relationship Diagram

There is only one active `RTLIL::Design` object that is used by all frontends, passes and backends called by the user, e.g. using a synthesis script. The `RTLIL::Design` then contains zero to many `RTLIL::Module` objects. This corresponds to modules in Verilog or entities in VHDL. Each module in turn contains objects from three different categories:

- `RTLIL::Cell` and `RTLIL::Wire` objects represent classical netlist data.
- `RTLIL::Process` objects represent the decision trees (if-then-else statements, etc.) and synchronization declarations (clock signals and sensitivity) from Verilog always and VHDL process blocks.

- `RTLIL::Memory` objects represent addressable memories (arrays).

Usually the output of the synthesis procedure is a netlist, i.e. all `RTLIL::Process` and `RTLIL::Memory` objects must be replaced by `RTLIL::Cell` and `RTLIL::Wire` objects by synthesis passes.

All features of the HDL that cannot be mapped directly to these RTLIL classes must be transformed to an RTLIL-compatible representation by the HDL frontend. This includes Verilog-features such as generate-blocks, loops and parameters.

The following sections contain a more detailed description of the different parts of RTLIL and rationale behind some of the design decisions.

RTLIL identifiers

All identifiers in RTLIL (such as module names, port names, signal names, cell types, etc.) follow the following naming convention: they must either start with a backslash (\) or a dollar sign (`$`).

Identifiers starting with a backslash are public visible identifiers. Usually they originate from one of the HDL input files. For example the signal name `\sig42` is most likely a signal that was declared using the name `sig42` in an HDL input file. On the other hand the signal name `$sig42` is an auto-generated signal name. The backends convert all identifiers that start with a dollar sign to identifiers that do not collide with identifiers that start with a backslash.

This has three advantages:

- First, it is impossible that an auto-generated identifier collides with an identifier that was provided by the user.
- Second, the information about which identifiers were originally provided by the user is always available which can help guide some optimizations. For example, *`opt_clean`* tries to preserve signals with a user-provided name but doesn't hesitate to delete signals that have auto-generated names when they just duplicate other signals. Note that this can be overridden with the `-purge` option to also delete internal nets with user-provided names.
- Third, the delicate job of finding suitable auto-generated public visible names is deferred to one central location. Internally auto-generated names that may hold important information for Yosys developers can be used without disturbing external tools. For example the Verilog backend assigns names in the form `_123_`.

Whitespace and control characters (any character with an ASCII code 32 or less) are not allowed in RTLIL identifiers; most frontends and backends cannot support these characters in identifiers.

In order to avoid programming errors, the RTLIL data structures check if all identifiers start with either a backslash or a dollar sign, and contain no whitespace or control characters. Violating these rules results in a runtime error.

All RTLIL identifiers are case sensitive.

Some transformations, such as flattening, may have to change identifiers provided by the user to avoid name collisions. When that happens, attribute `hdlname` is attached to the object with the changed identifier. This attribute contains one name (if emitted directly by the frontend, or is a result of disambiguation) or multiple names separated by spaces (if a result of flattening). All names specified in the `hdlname` attribute are public and do not include the leading \.

RTLIL::Design and RTLIL::Module

The `RTLIL::Design` object is basically just a container for `RTLIL::Module` objects. In addition to a list of `RTLIL::Module` objects the `RTLIL::Design` also keeps a list of selected objects, i.e. the objects that passes should operate on. In most cases the whole design is selected and therefore passes operate on the whole design. But this mechanism can be useful for more complex synthesis jobs in which only parts of the design should be affected by certain passes.

Besides the objects shown in the *ER diagram* above, an `RTLIL::Module` object contains the following additional properties:

- The module name
- A list of attributes
- A list of connections between wires
- An optional frontend callback used to derive parametrized variations of the module

The attributes can be Verilog attributes imported by the Verilog frontend or attributes assigned by passes. They can be used to store additional metadata about modules or just mark them to be used by certain part of the synthesis script but not by others.

Verilog and VHDL both support parametric modules (known as "generic entities" in VHDL). The RTLIL format does not support parametric modules itself. Instead each module contains a callback function into the AST frontend to generate a parametrized variation of the `RTLIL::Module` as needed. This callback then returns the auto-generated name of the parametrized variation of the module. (A hash over the parameters and the module name is used to prohibit the same parametrized variation from being generated twice. For modules with only a few parameters, a name directly containing all parameters is generated instead of a hash string.)

RTLIL::Cell and RTLIL::Wire

A module contains zero to many `RTLIL::Cell` and `RTLIL::Wire` objects. Objects of these types are used to model netlists. Usually the goal of all synthesis efforts is to convert all modules to a state where the functionality of the module is implemented only by cells from a given cell library and wires to connect these cells with each other. Note that module ports are just wires with a special property.

An `RTLIL::Wire` object has the following properties:

- The wire name
- A list of attributes
- A width (buses are just wires with a width more than 1)
- Bus direction (MSB to LSB or vice versa)
- Lowest valid bit index (LSB or MSB depending on bus direction)
- If the wire is a port: port number and direction (input/output/inout)

As with modules, the attributes can be Verilog attributes imported by the Verilog frontend or attributes assigned by passes.

In Yosys, busses (signal vectors) are represented using a single wire object with a width more than 1. So Yosys does not convert signal vectors to individual signals. This makes some aspects of RTLIL more complex but enables Yosys to be used for coarse grain synthesis where the cells of the target architecture operate on entire signal vectors instead of single bit wires.

In Verilog and VHDL, busses may have arbitrary bounds, and LSB can have either the lowest or the highest bit index. In RTLIL, bit 0 always corresponds to LSB; however, information from the HDL frontend is preserved so that the bus will be correctly indexed in error messages, backend output, constraint files, etc.

An `RTLIL::Cell` object has the following properties:

- The cell name and type
- A list of attributes
- A list of parameters (for parametric cells)
- Cell ports and the connections of ports to wires and constants

The connections of ports to wires are coded by assigning an `RTLIL::SigSpec` to each cell port. The `RTLIL::SigSpec` data type is described in the next section.

RTLIL::SigSpec

A "signal" is everything that can be applied to a cell port. I.e.

- A bit from a wire (`RTLIL::SigBit`)
  1em For example: `mywire[24]`
- A range of bits from a wire (wire variant of `RTLIL::SigChunk`)
  1em For example: `mywire, mywire[15:8]`
- Any constant value of arbitrary bit-width (`std::vector<RTLIL::State>` variant of `RTLIL::SigChunk`)
  1em For example: `1337, 16'b0000010100111001, 1'b1, 1'bx`

The `RTLIL::SigSpec` data type is used to represent signals. It contains a single `RTLIL::SigChunk` or a vector of `RTLIL::SigBit`. The `RTLIL::Cell` object contains one `RTLIL::SigSpec` for each cell port.

In addition, connections between wires are represented using a pair of `RTLIL::SigSpec` objects. Such pairs are needed in different locations. Therefore the type name `RTLIL::SigSig` was defined for such a pair.

RTLIL::Process

When a high-level HDL frontend processes behavioural code it splits it up into data path logic (e.g. the expression `a + b` is replaced by the output of an adder that takes a and b as inputs) and an `RTLIL::Process` that models the control logic of the behavioural code. Let's consider a simple example:

```
module ff_with_en_and_async_reset(clock, reset, enable, d, q);
input clock, reset, enable, d;
output reg q;
always @(posedge clock, posedge reset)
    if (reset)
        q <= 0;
    else if (enable)
        q <= d;
endmodule
```

In this example there is no data path and therefore the `RTLIL::Module` generated by the frontend only contains a few `RTLIL::Wire` objects and an `RTLIL::Process`. The `RTLIL::Process` in RTLIL syntax:

```
process $proc$ff_with_en_and_async_reset.v:4$1
    assign $0\q[0:0] \q
    switch \reset
        case 1'1
            assign $0\q[0:0] 1'0
        case
            switch \enable
                case 1'1
                    assign $0\q[0:0] \d
                case
            end
    end
    sync posedge \clock
        update \q $0\q[0:0]
```

(continues on next page)

(continued from previous page)

```
15      sync posedge \reset
16          update \q $0\q[0:0]
17  end
```

This `RTLIL::Process` contains two `RTLIL::SyncRule` objects, two `RTLIL::SwitchRule` objects and five `RTLIL::CaseRule` objects. The wire `$0\q[0:0]` is an automatically created wire that holds the next value of `\q`. The lines 2..12 describe how `$0\q[0:0]` should be calculated. The lines 13..16 describe how the value of `$0\q[0:0]` is used to update `\q`.

An `RTLIL::Process` is a container for zero or more `RTLIL::SyncRule` objects and exactly one `RTLIL::CaseRule` object, which is called the root case.

An `RTLIL::SyncRule` object contains an (optional) synchronization condition (signal and edge-type), zero or more assignments (`RTLIL::SigSig`), and zero or more memory writes (`RTLIL::MemWriteAction`). The always synchronization condition is used to break combinatorial loops when a latch should be inferred instead.

An `RTLIL::CaseRule` is a container for zero or more assignments (`RTLIL::SigSig`) and zero or more `RTLIL::SwitchRule` objects. An `RTLIL::SwitchRule` objects is a container for zero or more `RTLIL::CaseRule` objects.

In the above example the lines 2..12 are the root case. Here `$0\q[0:0]` is first assigned the old value `\q` as default value (line 2). The root case also contains an `RTLIL::SwitchRule` object (lines 3..12). Such an object is very similar to the C switch statement as it uses a control signal (`\reset` in this case) to determine which of its cases should be active. The `RTLIL::SwitchRule` object then contains one `RTLIL::CaseRule` object per case. In this example there is a case[1] for `\reset == 1` that causes `$0\q[0:0]` to be set (lines 4 and 5) and a default case that in turn contains a switch that sets `$0\q[0:0]` to the value of `\d` if `\enable` is active (lines 6..11).

A case can specify zero or more compare values that will determine whether it matches. Each of the compare values must be the exact same width as the control signal. When more than one compare value is specified, the case matches if any of them matches the control signal; when zero compare values are specified, the case always matches (i.e. it is the default case).

A switch prioritizes cases from first to last: multiple cases can match, but only the first matched case becomes active. This normally synthesizes to a priority encoder. The parallel_case attribute allows passes to assume that no more than one case will match, and full_case attribute allows passes to assume that exactly one case will match; if these invariants are ever dynamically violated, the behavior is undefined. These attributes are useful when an invariant invisible to the synthesizer causes the control signal to never take certain bit patterns.

The lines 13..16 then cause `\q` to be updated whenever there is a positive clock edge on `\clock` or `\reset`.

In order to generate such a representation, the language frontend must be able to handle blocking and nonblocking assignments correctly. However, the language frontend does not need to identify the correct type of storage element for the output signal or generate multiplexers for the decision tree. This is done by passes that work on the RTLIL representation. Therefore it is relatively easy to substitute these steps with other algorithms that target different target architectures or perform optimizations or other transformations on the decision trees before further processing them.

One of the first actions performed on a design in RTLIL representation in most synthesis scripts is identifying asynchronous resets. This is usually done using the *proc_arst* pass. This pass transforms the above example to the following `RTLIL::Process`:

[1] The syntax `1'1` in the RTLIL code specifies a constant with a length of one bit (the first `1`), and this bit is a one (the second `1`).

```
process $proc$ff_with_en_and_async_reset.v:4$1
    assign $0\q[0:0] \q
    switch \enable
        case 1'1
            assign $0\q[0:0] \d
        case
    end
    sync posedge \clock
        update \q $0\q[0:0]
    sync high \reset
        update \q 1'0
end
```

This pass has transformed the outer `RTLIL::SwitchRule` into a modified `RTLIL::SyncRule` object for the `\reset` signal. Further processing converts the `RTLIL::Process` into e.g. a d-type flip-flop with asynchronous reset and a multiplexer for the enable signal:

```
cell $adff $procdff$6
    parameter \ARST_POLARITY 1'1
    parameter \ARST_VALUE 1'0
    parameter \CLK_POLARITY 1'1
    parameter \WIDTH 1
    connect \ARST \reset
    connect \CLK \clock
    connect \D $0\q[0:0]
    connect \Q \q
end
cell $mux $procmux$3
    parameter \WIDTH 1
    connect \A \q
    connect \B \d
    connect \S \enable
    connect \Y $0\q[0:0]
end
```

Different combinations of passes may yield different results. Note that *$adff* and *$mux* are internal cell types that still need to be mapped to cell types from the target cell library.

Some passes refuse to operate on modules that still contain `RTLIL::Process` objects as the presence of these objects in a module increases the complexity. Therefore the passes to translate processes to a netlist of cells are usually called early in a synthesis script. The proc pass calls a series of other passes that together perform this conversion in a way that is suitable for most synthesis tasks.

RTLIL::Memory

For every array (memory) in the HDL code an `RTLIL::Memory` object is created. A memory object has the following properties:

- The memory name
- A list of attributes
- The width of an addressable word
- The size of the memory in number of words

All read accesses to the memory are transformed to *$memrd* cells and all write accesses to *$memwr* cells by the language frontend. These cells consist of independent read- and write-ports to the memory. Memory initialization is transformed to *$meminit* cells by the language frontend. The `\MEMID` parameter on these cells is used to link them together and to the `RTLIL::Memory` object they belong to.

The rationale behind using separate cells for the individual ports versus creating a large multiport memory cell right in the language frontend is that the separate *$memrd* and *$memwr* cells can be consolidated using resource sharing. As resource sharing is a non-trivial optimization problem where different synthesis tasks can have different requirements it lends itself to do the optimisation in separate passes and merge the `RTLIL::Memory` objects and *$memrd* and *$memwr* cells to multiport memory blocks after resource sharing is completed.

The memory pass performs this conversion and can (depending on the options passed to it) transform the memories directly to d-type flip-flops and address logic or yield multiport memory blocks (represented using *$mem* cells).

See *Memories* for details about the memory cell types.

4.3 Working with the Yosys codebase

This section goes into additional detail on the Yosys source code and git repository. This information is not needed for simply using Yosys, but may be of interest for developers looking to customise Yosys builds.

4.3.1 Writing extensions

Todo

check text is coherent

Todo

update to use `/code_examples/extensions/test*.log`

This chapter contains some bits and pieces of information about programming yosys extensions. Don't be afraid to ask questions on the YosysHQ Discourse.

Todo

mention coding guide

Quick guide

Code examples from this section are included in the `docs/source/code_examples/extensions` directory of the Yosys source code.

Program components and data formats

See *The RTL Intermediate Language (RTLIL)* document for more information about the internal data storage format used in Yosys and the classes that it provides.

This document will focus on the much simpler version of RTLIL left after the commands *proc* and *memory* (or `memory -nomap`):

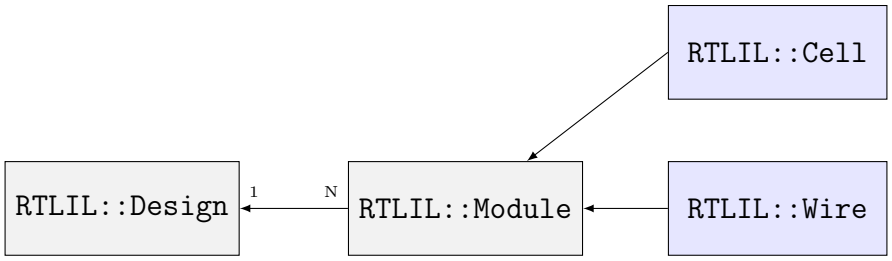


Fig. 4.5: Simplified RTLIL entity-relationship diagram without memories and processes

It is possible to only work on this simpler version:

Todo

consider replacing inline code

```
for (RTLIL::Module *module : design->selected_modules() {
    if (module->has_memories_warn() || module->has_processes_warn())
        continue;
    ....
}
```

When trying to understand what a command does, creating a small test case to look at the output of `dump` and *`show`* before and after the command has been executed can be helpful. *Selections* has more information on using these commands.

Creating a command

Todo

add/expand supporting text

Let's create a very simple test command which prints the arguments we called it with, and lists off the current design's modules.

Listing 4.1: Example command `my_cmd` from `my_cmd.cc`

```
#include <kernel/yosys.h>
USING_YOSYS_NAMESPACE

struct MyPass : public Pass {
    MyPass() : Pass("my_cmd", "just a simple test") { }
    void execute(std::vector<std::string> args, RTLIL::Design *design) override
    {
        log("Arguments to my_cmd:\n");
        for (auto &arg : args)
            log("  %s\n", arg);

        log("Modules in current design:\n");
```

(continues on next page)

(continued from previous page)

```
        for (auto mod : design->modules())
            log("  %s (%d wires, %d cells)\n", mod,
                    GetSize(mod->wires()), GetSize(mod->cells()));
    }
} MyPass;
```

Note that we are making a global instance of a class derived from `Yosys::Pass`, which we get by including `kernel/yosys.h`.

Compiling to a plugin

Yosys can be extended by adding additional C++ code to the Yosys code base, or by loading plugins into Yosys. For maintainability it is generally recommended to create plugins.

The following command compiles our example `my_cmd` to a Yosys plugin:

Todo

replace inline code

```
yosys-config --exec --cxx --cxxflags --ldflags \
-o my_cmd.so -shared my_cmd.cc --ldlibs
```

Or shorter:

```
yosys-config --build my_cmd.so my_cmd.cc
```

Running Yosys with the `-m` option allows the plugin to be used. Here's a quick example that also uses the `-p` option to run `my_cmd foo bar`.

```
$ yosys -m ./my_cmd.so -p 'my_cmd foo bar'

-- Running command `my_cmd foo bar' --
Arguments to my_cmd:
  my_cmd
  foo
  bar
Modules in current design:
```

Creating modules from scratch

Let's create the following module using the RTLIL API:

Listing 4.2: absval_ref.v

```
module absval_ref(input signed [3:0] a, output [3:0] y);
        assign y = a[3] ? -a : a;
endmodule
```

We'll do the same as before and format it as a a `Yosys::Pass`.

Listing 4.3: `test1` - creating the absval module, from `my_cmd.cc`

```
struct Test1Pass : public Pass {
    Test1Pass() : Pass("test1", "creating the absval module") { }
    void execute(std::vector<std::string>, RTLIL::Design *design) override
    {
        if (design->has("\\absval") != 0)
            log_error("A module with the name absval already exists!\n");

        RTLIL::Module *module = design->addModule("\\absval");
        log("Name of this module: %s\n", module);

        RTLIL::Wire *a = module->addWire("\\a", 4);
        a->port_input = true;
        a->port_id = 1;

        RTLIL::Wire *y = module->addWire("\\y", 4);
        y->port_output = true;
        y->port_id = 2;

        RTLIL::Wire *a_inv = module->addWire(NEW_ID, 4);
        module->addNeg(NEW_ID, a, a_inv, true);
        module->addMux(NEW_ID, a, a_inv, RTLIL::SigSpec(a, 3), y);

        module->fixup_ports();
    }
} Test1Pass;
```

```
$ yosys -m ./my_cmd.so -p 'test1' -Q

-- Running command `test1' --
Name of this module: absval
```

And if we look at the schematic for this new module we see the following:

Modifying modules

Most commands modify existing modules, not create new ones.

When modifying existing modules, stick to the following DOs and DON'Ts:

- Do not remove wires. Simply disconnect them and let a successive *`clean`* command worry about removing it.
- Use `module->fixup_ports()` after changing the `port_*` properties of wires.
- You can safely remove cells or change the `connections` property of a cell, but be careful when changing the size of the `SigSpec` connected to a cell port.
- Use the `SigMap` helper class (see next section) when you need a unique handle for each signal bit.

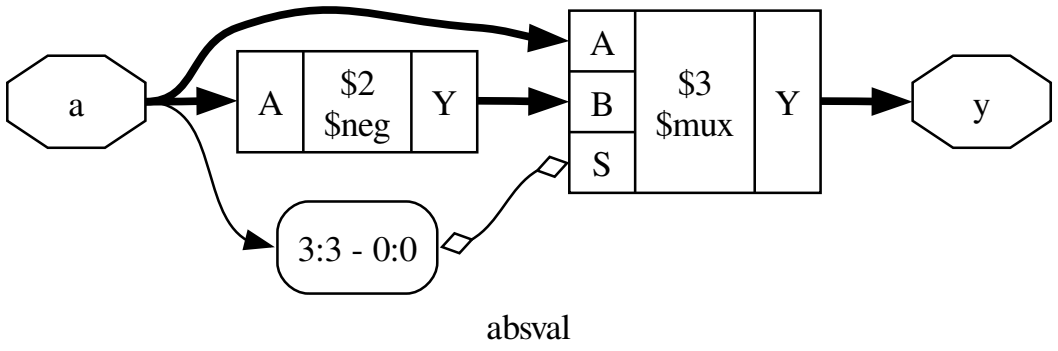


Fig. 4.6: Output of `yosys -m ./my_cmd.so -p 'test1; show'`

Using the SigMap helper class

Consider the following module:

Listing 4.4: `sigmap_test.v`

```
module test(input a, output x, y);
        assign x = a, y = a;
endmodule
```

In this case `a`, `x`, and `y` are all different names for the same signal. However:

Todo

use my_cmd.cc literalincludes

```
RTLIL::SigSpec a(module->wire("\\a")), x(module->wire("\\x")),
                                       y(module->wire("\\y"));
log("%d %d %d\n", a == x, x == y, y == a); // will print "0 0 0"
```

The `SigMap` helper class can be used to map all such aliasing signals to a unique signal from the group (usually the wire that is directly driven by a cell or port).

```
SigMap sigmap(module);
log("%d %d %d\n", sigmap(a) == sigmap(x), sigmap(x) == sigmap(y),
                  sigmap(y) == sigmap(a)); // will print "1 1 1"
```

Printing log messages

The `log()` function is a `printf()`-like function that can be used to create log messages.

Use `log_signal()` to create a C-string for a SigSpec object:

```
log("Mapped signal x: %s\n", log_signal(sigmap(x)));
```

The pointer returned by `log_signal()` is automatically freed by the log framework at a later time.

Use `log_id()` to create a C-string for an `RTLIL::IdString`:

```
log("Name of this module: %s\n", log_id(module->name));
```

Use `log_header()` and `log_push()`/`log_pop()` to structure log messages:

Todo

replace inline code

```
log_header(design, "Doing important stuff!\n");
log_push();
for (int i = 0; i < 10; i++)
    log("Log message #%d.\n", i);
log_pop();
```

Error handling

Use `log_error()` to report a non-recoverable error:

Todo

replace inline code

```
if (design->modules.count(module->name) != 0)
    log_error("A module with the name %s already exists!\n", module);
```

Use `log_cmd_error()` to report a recoverable error:

```
if (design->selection().empty())
    log_cmd_error("This command can't operator on an empty selection!\n");
```

Use `log_assert()` and `log_abort()` instead of `assert()` and `abort()`.

The "stubnets" example module

The following is the complete code of the "stubnets" example module. It is included in the Yosys source distribution under `docs/source/code_examples/stubnets`.

Listing 4.5: stubnets.cc

```
// This is free and unencumbered software released into the public domain.
//
// Anyone is free to copy, modify, publish, use, compile, sell, or
// distribute this software, either in source code form or as a compiled
// binary, for any purpose, commercial or non-commercial, and by any
// means.

#include <kernel/yosys.h>
#include <kernel/sigtools.h>

#include <string>
#include <map>
#include <set>

USING_YOSYS_NAMESPACE
PRIVATE_NAMESPACE_BEGIN

// this function is called for each module in the design
static void find_stub_nets(RTLIL::Design *design, RTLIL::Module *module, bool report_
↪bits)
{
        // use a SigMap to convert nets to a unique representation
        SigMap sigmap(module);

        // count how many times a single-bit signal is used
        std::map<RTLIL::SigBit, int> bit_usage_count;

        // count output lines for this module (needed only for summary output at the end)
        int line_count = 0;

        log("Looking for stub wires in module %s:\n", module);

        // For all ports on all cells
        for (auto &cell_iter : module->cells_)
        for (auto &conn : cell_iter.second->connections())
        {
                // Get the signals on the port
                // (use sigmap to get a uniqe signal name)
                RTLIL::SigSpec sig = sigmap(conn.second);

                // add each bit to bit_usage_count, unless it is a constant
                for (auto &bit : sig)
                        if (bit.wire != NULL)
                                bit_usage_count[bit]++;
        }

        // for each wire in the module
        for (auto &wire_iter : module->wires_)
        {
                RTLIL::Wire *wire = wire_iter.second;

```

(continues on next page)

(continued from previous page)

```
51                  // .. but only selected wires
52                  if (!design->selected(module, wire))
53                          continue;
54
55                  // add +1 usage if this wire actually is a port
56                  int usage_offset = wire->port_id > 0 ? 1 : 0;
57
58                  // we will record which bits of the (possibly multi-bit) wire are stub␣
    ↪signals
59                  std::set<int> stub_bits;
60
61                  // get a signal description for this wire and split it into separate bits
62                  RTLIL::SigSpec sig = sigmap(wire);
63
64                  // for each bit (unless it is a constant):
65                  // check if it is used at least two times and add to stub_bits otherwise
66                  for (int i = 0; i < GetSize(sig); i++)
67                          if (sig[i].wire != NULL && (bit_usage_count[sig[i]] + usage_
    ↪offset) < 2)
68                                  stub_bits.insert(i);
69
70                  // continue if no stub bits found
71                  if (stub_bits.size() == 0)
72                          continue;
73
74                  // report stub bits and/or stub wires, don't report single bits
75                  // if called with report_bits set to false.
76                  if (GetSize(stub_bits) == GetSize(sig)) {
77                          log("  found stub wire: %s\n", wire);
78                  } else {
79                          if (!report_bits)
80                                  continue;
81                          log("  found wire with stub bits: %s [", wire);
82                          for (int bit : stub_bits)
83                                  log("%s%d", bit == *stub_bits.begin() ? "" : ", ", bit);
84                          log("]\n");
85                  }
86
87                  // we have outputted a line, increment summary counter
88                  line_count++;
89          }
90
91          // report summary
92          if (report_bits)
93                  log("  found %d stub wires or wires with stub bits.\n", line_count);
94          else
95                  log("  found %d stub wires.\n", line_count);
96  }
97
98  // each pass contains a singleton object that is derived from Pass
99  struct StubnetsPass : public Pass {
100         StubnetsPass() : Pass("stubnets") { }
```

(continues on next page)

(continued from previous page)

```
	void execute(std::vector<std::string> args, RTLIL::Design *design) override
	{
		// variables to mirror information from passed options
		bool report_bits = 0;

		log_header(design, "Executing STUBNETS pass (find stub nets).\n");

		// parse options
		size_t argidx;
		for (argidx = 1; argidx < args.size(); argidx++) {
			std::string arg = args[argidx];
			if (arg == "-report_bits") {
				report_bits = true;
				continue;
			}
			break;
		}

		// handle extra options (e.g. selection)
		extra_args(args, argidx, design);

		// call find_stub_nets() for each module that is either
		// selected as a whole or contains selected objects.
		for (auto &it : design->modules_)
			if (design->selected_module(it.first))
				find_stub_nets(design, it.second, report_bits);
	}
} StubnetsPass;

PRIVATE_NAMESPACE_END
```

Listing 4.6: `Makefile`

```
.PHONY: all dots examples
all: dots examples
dots:
examples:

.PHONY: test
test: stubnets.so
	yosys -ql test1.log -m ./stubnets.so test.v -p "stubnets"
	yosys -ql test2.log -m ./stubnets.so test.v -p "opt; stubnets"
	yosys -ql test3.log -m ./stubnets.so test.v -p "techmap; opt; stubnets -report_
↪bits"
	tail test1.log test2.log test3.log

stubnets.so: stubnets.cc
	yosys-config --exec --cxx --cxxflags -I../../../../ --ldflags -o $@ -shared $^ --
↪ldlibs

.PHONY: clean
clean:
```

(continues on next page)

(continued from previous page)

```
18         rm -f test1.log test2.log test3.log
19         rm -f stubnets.so stubnets.d
```

Listing 4.7: `test.v`

```
1  module uut(in1, in2, in3, out1, out2);
2
3  input [8:0] in1, in2, in3;
4  output [8:0] out1, out2;
5
6  assign out1 = in1 + in2 + (in3 >> 4);
7
8  endmodule
```

4.3.2 Compiling with Verific library

The easiest way to get Yosys with Verific support is to contact YosysHQ for a Tabby CAD Suite evaluation license and download link. The TabbyCAD Suite includes additional patches and a custom extensions library in order to get the most out of the Verific parser when using Yosys.

If you already have a license for the Verific parser, in either source or binary form, you may be able to compile Yosys with partial Verific support yourself.

The Yosys-Verific patch

YosysHQ maintains and develops a patch for Verific in order to better integrate with Yosys and to provide features required by some of the formal verification front-end tools. To license this patch for your own Yosys builds, contact YosysHQ.

> **Warning**
>
> While synthesis from RTL may be possible without this patch, YosysHQ provides no guarantees of correctness and is unable to provide support.

We recommend against using unpatched Yosys+Verific builds in conjunction with the formal verification front-end tools unless you are familiar with their inner workings. There are cases where the tools will appear to work, while producing incorrect results.

> **Note**
>
> Some of the formal verification front-end tools may not be fully supported without the full TabbyCAD suite. If you want to use these tools, including SBY, make sure to ask us if the Yosys-Verific patch is right for you.

Compile options

To enable Verific support `ENABLE_VERIFIC` has to be set to `1` and `VERIFIC_DIR` needs to point to the location where the library is located.

Parameter	Default	Description
ENABLE_VERIFIC	0	Enable compilation with Verific
VERIFIC_DIR	/usr/local/src/verific_lib	Library and headers location

Since there are multiple Verific library builds and they can have different features, there are compile options to select them.

Parameter	Default	Description
ENABLE_VERIFIC_SYSTEMVERILOG	1	SystemVerilog support
ENABLE_VERIFIC_VHDL	1	VHDL support
ENABLE_VERIFIC_HIER_TREE	1	Hierarchy tree support
ENABLE_VERIFIC_YOSYSHQ_EXTENSIONS	0	YosysHQ specific extensions support
ENABLE_VERIFIC_EDIF	0	EDIF support
ENABLE_VERIFIC_LIBERTY	0	Liberty file support

To find the compile options used for a given Yosys build, call `yosys-config --cxxflags`. This documentation was built with the following compile options:

```
  --cxxflags    -O2 -flto=auto -ffat-lto-objects -fexceptions -g \
                    -grecord-gcc-switches -pipe -Wall -Werror=format-security \
                    -Wp,-U_FORTIFY_SOURCE,-D_FORTIFY_SOURCE=3 \
                    -Wp,-D_GLIBCXX_ASSERTIONS \
                    -specs=/usr/lib/rpm/redhat/redhat-hardened-cc1 \
                    -fstack-protector-strong \
                    -specs=/usr/lib/rpm/redhat/redhat-annobin-cc1 -m64 \
                    -march=x86-64 -mtune=generic -fasynchronous-unwind-tables \
                    -fstack-clash-protection -fcf-protection -mtls-dialect=gnu2 \
                    -fno-omit-frame-pointer -mno-omit-leaf-frame-pointer -Wall \
                    -Wextra -Werror=unused -ggdb -I"/usr/include/yosys" -MD \
                    -MP -D_YOSYS_ -fPIC -I/usr/include -DYOSYS_VER=\"0.66\" \
                    -DYOSYS_MAJOR=0 -DYOSYS_MINOR=66 -DYOSYS_COMMIT=0.66 \
                    -std=c++20 -O3 -DYOSYS_ENABLE_READLINE \
                    -DYOSYS_ENABLE_PLUGINS -DYOSYS_ENABLE_GLOB \
                    -DYOSYS_ENABLE_ZLIB -DTCL_WITH_EXTERNAL_TOMMATH \
                    -DWITH_GZFILEOP -DYOSYS_ENABLE_TCL -DYOSYS_ENABLE_THREADS \
                    -DYOSYS_ENABLE_ABC
```

Note

The YosysHQ specific extensions are only available with the TabbyCAD suite.

Required Verific features

The following features, along with their corresponding Yosys build parameters, are required for the Yosys-Verific patch:

- RTL elaboration with
  - SystemVerilog with `ENABLE_VERIFIC_SYSTEMVERILOG`, and/or
  - VHDL support with `ENABLE_VERIFIC_VHDL`.

- Hierarchy tree support and static elaboration with `ENABLE_VERIFIC_HIER_TREE`.

Please be aware that the following Verific configuration build parameter needs to be enabled in order to create the fully supported build:

```
database/DBCompileFlags.h:
    DB_PRESERVE_INITIAL_VALUE
```

Note

Yosys+Verific builds may compile without these features, but we provide no guarantees and cannot offer support if they are disabled or the Yosys-Verific patch is not used.

Optional Verific features

The following Verific features are available with TabbyCAD and can be enabled in Yosys builds:

- EDIF support with `ENABLE_VERIFIC_EDIF`, and
- Liberty file support with `ENABLE_VERIFIC_LIBERTY`.

Partially supported builds

This section describes Yosys+Verific configurations which we have confirmed as working in the past, however they are not a part of our regular tests so we cannot guarantee they are still functional.

To be able to compile Yosys with Verific, the Verific library must have support for at least one HDL language with RTL elaboration enabled. The following table lists a series of build configurations which are possible, but only provide a limited subset of features. Please note that support is limited without YosysHQ specific extensions of Verific library.

Configuration values:

a. `ENABLE_VERIFIC_SYSTEMVERILOG`

b. `ENABLE_VERIFIC_VHDL`

c. `ENABLE_VERIFIC_HIER_TREE`

d. `ENABLE_VERIFIC_YOSYSHQ_EXTENSIONS`

Features	Configuration values a	b	c	d
SystemVerilog + RTL elaboration	1	0	0	0
VHDL + RTL elaboration	0	1	0	0
SystemVerilog + VHDL + RTL elaboration	1	1	0	0
SystemVerilog + RTL elaboration + Static elaboration + Hier tree	1	0	1	0
VHDL + RTL elaboration + Static elaboration + Hier tree	0	1	1	0
SystemVerilog + VHDL + RTL elaboration + Static elaboration + Hier tree	1	1	1	0

Note

In case your Verific build has EDIF and/or Liberty support, you can enable those options. These are not mentioned above for simplification and since they are disabled by default.

4.3.3 Writing a new backend using FunctionalIR

What is FunctionalIR

To simplify the writing of backends for functional languages or similar targets, Yosys provides an alternative intermediate representation called FunctionalIR which maps more directly on those targets.

FunctionalIR represents the design as a function `(inputs, current_state) -> (outputs, next_state)`. This function is broken down into a series of assignments to variables. Each assignment is a simple operation, such as an addition. Complex operations are broken up into multiple steps. For example, an RTLIL addition will be translated into a sign/zero extension of the inputs, followed by an addition.

Like SSA form, each variable is assigned to exactly once. We can thus treat variables and assignments as equivalent and, since this is a graph-like representation, those variables are also called "nodes". Unlike RTLIL's cells and wires representation, this representation is strictly ordered (topologically sorted) with definitions preceding their use.

Every node has a "sort" (the FunctionalIR term for what might otherwise be called a "type"). The sorts available are

- `bit[n]` for an `n`-bit bitvector, and
- `memory[n,m]` for an immutable array of `2**n` values of sort `bit[m]`.

In terms of actual code, Yosys provides a class `Functional::IR` that represents a design in FunctionalIR. `Functional::IR::from_module` generates an instance from an RTLIL module. The entire design is stored as a whole in an internal data structure. To access the design, the `Functional::Node` class provides a reference to a particular node in the design. The `Functional::IR` class supports the syntax `for(auto node : ir)` to iterate over every node.

`Functional::IR` also keeps track of inputs, outputs and states. By a "state" we mean a pair of a "current state" input and a "next state" output. One such pair is created for every register and for every memory. Every input, output and state has a name (equal to their name in RTLIL), a sort and a kind. The kind field usually remains as the default value `$input`, `$output` or `$state`, however some RTLIL cells such as `$assert` or `$anyseq` generate auxiliary inputs/outputs/states that are given a different kind to distinguish them from ordinary RTLIL inputs/outputs/states.

- To access an individual input/output/state, use `ir.input(name, kind)`, `ir.output(name, kind)` or `ir.state(name, kind)`. `kind` defaults to the default kind.
- To iterate over all inputs/outputs/states of a certain kind, methods `ir.inputs`, `ir.outputs`, `ir.states` are provided. Their argument defaults to the default kinds mentioned.
- To iterate over inputs/outputs/states of any kind, use `ir.all_inputs`, `ir.all_outputs` and `ir.all_states`.
- Outputs have a node that indicate the value of the output, this can be retrieved via `output.value()`.
- States have a node that indicate the next value of the state, this can be retrieved via `state.next_value()`. They also have an initial value that is accessed as either `state.initial_value_signal()` or `state.initial_value_memory()`, depending on their sort.

Each node has a "function", which defines its operation (for a complete list of functions and a specification of their operation, see `functional.h`). Functions are represented as an enum `Functional::Fn` and the function field can be accessed as `node.fn()`. Since the most common operation is a switch over the function that also accesses the arguments, the `Node` class provides a method `visit` that implements the visitor pattern. For example, for an addition node `node` with arguments `n1` and `n2`, `node.visit(visitor)`

would call `visitor.add(node, n1, n2)`. Thus typically one would implement a class with a method for every function. Visitors should inherit from either `Functional::AbstractVisitor<ReturnType>` or `Functional::DefaultVisitor<ReturnType>`. The former will produce a compiler error if a case is unhandled, the latter will call `default_handler(node)` instead. Visitor methods should be marked as `override` to provide compiler errors if the arguments are wrong.

Utility classes

`functional.h` also provides utility classes that are independent of the main FunctionalIR representation but are likely to be useful for backends.

`Functional::Writer` provides a simple formatting class that wraps a `std::ostream` and provides the following methods:

- `writer << value` wraps `os << value`.
- `writer.print(fmt, value0, value1, value2, ...)` replaces `{0}`, `{1}`, `{2}`, etc in the string `fmt` with `value0`, `value1`, `value2`, resp. Each value is formatted using `os << value`. It is also possible to write `{}` to refer to one past the last index, i.e. `{1} {} {} {7} {}` is equivalent to `{1} {2} {3} {7} {8}`.
- `writer.print_with(fn, fmt, value0, value1, value2, ...)` functions much the same as `print` but it uses `os << fn(value)` to print each value and falls back to `os << value` if `fn(value)` is not legal.

`Functional::Scope` keeps track of variable names in a target language. It is used to translate between different sets of legal characters and to avoid accidentally re-defining identifiers. Users should derive a class from `Scope` and supply the following:

- `Scope<Id>` takes a template argument that specifies a type that's used to uniquely distinguish variables. Typically this would be `int` (if variables are used for `Functional::IR` nodes) or `IdString`.
- The derived class should provide a constructor that calls `reserve` for every reserved word in the target language.
- A method `bool is_character_legal(char c, int index)` has to be provided that returns `true` iff `c` is legal in an identifier at position `index`.

Given an instance `scope` of the derived class, the following methods are then available:

- `scope.reserve(std::string name)` marks the given name as being in-use
- `scope.unique_name(IdString suggestion)` generates a previously unused name and attempts to make it similar to `suggestion`.
- `scope(Id id, IdString suggestion)` functions similar to `unique_name`, except that multiple calls with the same `id` are guaranteed to retrieve the same name (independent of `suggestion`).

`sexpr.h` provides classes that represent and pretty-print s-expressions. S-expressions can be constructed with `SExpr::list`, for example `SExpr expr = SExpr::list("add", "x", SExpr::list("mul", "y", "z"))` represents `(add x (mul y z))` (by adding `using SExprUtil::list` to the top of the file, `list` can be used as shorthand for `SExpr::list`). For prettyprinting, `SExprWriter` wraps an `std::ostream` and provides the following methods:

- `writer << sexpr` writes the provided expression to the output, breaking long lines and adding appropriate indentation.
- `writer.open(sexpr)` is similar to `writer << sexpr` but will omit the last closing parenthesis. Further arguments can then be added separately with `<<` or `open`. This allows for printing large s-expressions without needing to construct the whole expression in memory first.

- `writer.open(sexpr, false)` is similar to `writer.open(sexpr)` but further arguments will not be indented. This is used to avoid unlimited indentation on structures with unlimited nesting.
- `writer.close(n = 1)` closes the last `n` open s-expressions.
- `writer.push()` and `writer.pop()` are used to automatically close s-expressions. `writer.pop()` closes all s-expressions opened since the last call to `writer.push()`.
- `writer.comment(string)` writes a comment on a separate-line. `writer.comment(string, true)` appends a comment to the last printed s-expression.
- `writer.flush()` flushes any buffering and should be called before any direct access to the underlying `std::ostream`. It does not close unclosed parentheses.
- The destructor calls `flush` but also closes all unclosed parentheses.

Example: A minimal functional backend

At its most basic, there are three steps we need to accomplish for a minimal functional backend. First, we need to convert our design into FunctionalIR. This is most easily done by calling the `Functional::IR::from_module()` static method with our top-level module, or iterating over and converting each of the modules in our design. Second, we need to handle each of the `Functional::Nodes` in our design. Iterating over the `Functional::IR` includes reading the module inputs and current state, but not writing the results. So our final step is to handle the outputs and next state.

In order to add an output command to Yosys, we implement the `Yosys::Backend` class and provide an instance of it:

Listing 4.8: Example source code for a minimal functional backend, `dummy.cc`

```
#include "kernel/functional.h"
#include "kernel/yosys.h"

USING_YOSYS_NAMESPACE
PRIVATE_NAMESPACE_BEGIN

struct FunctionalDummyBackend : public Backend {
        FunctionalDummyBackend() : Backend("functional_dummy", "dump generated␣
↪Functional IR") {}
        void execute(std::ostream *&f, std::string filename, std::vector<std::string>␣
↪args, RTLIL::Design *design) override
        {
                // backend pass boiler plate
                log_header(design, "Executing dummy functional backend.\n");

                size_t argidx = 1;
                extra_args(f, filename, args, argidx, design);

                for (auto module : design->selected_modules())
                {
                        log("Processing module `%s`.\n", module->name);

                        // convert module to FunctionalIR
                        auto ir = Functional::IR::from_module(module);
                        *f << "module " << module->name.c_str() << "\n";
```

(continues on next page)

(continued from previous page)

```
                        // write node functions
                        for (auto node : ir)
                                *f << "  assign " << node.name().unescape()
                                   << " = " << node.to_string() << "\n";
                        *f << "\n";

                        // write outputs and next state
                        for (auto output : ir.outputs())
                                *f << " " << output->kind.unescape()
                                   << " " << output->name.unescape()
                                   << " = " << output->value().name().unescape() << "\n";
                        for (auto state : ir.states())
                                *f << " " << state->kind.unescape()
                                   << " " << state->name.unescape()
                                   << " = " << state->next_value().name().unescape() <<
↪"\n";
                }
        }
} FunctionalDummyBackend;

PRIVATE_NAMESPACE_END
```

Because we are using the `Backend` class, our `"functional_dummy"` is registered as the `write_functional_dummy` command. The `execute` method is the part that runs when the user calls the command, handling any options, preparing the output file for writing, and iterating over selected modules in the design. Since we don't have any options here, we set `argidx = 1` and call the `extra_args()` method. This method will read the command arguments, raising an error if there are any unexpected ones. It will also assign the pointer `f` to the output file, or stdout if none is given.

> **Note**
>
> For more on adding new commands to Yosys and how they work, refer to *Writing extensions*.

For this minimal example all we are doing is printing out each node. The `node.name()` method returns an `RTLIL::IdString`, which we convert for printing with `unescape()`. Then, to print the function of the node, we use `node.to_string()` which gives us a string of the form `function(args)`. The `function` part is the result of `Functional::IR::fn_to_string(node.fn())`; while `args` is the zero or more arguments passed to the function, most commonly the name of another node. Behind the scenes, the `node.to_string()` method actually wraps `node.visit(visitor)` with a private visitor whose return type is `std::string`.

Finally we iterate over the module's outputs and states, using `Functional::IROutput::value()` and `Functional::IRState::next_value()` respectively in order to get the results of the transfer function.

Example: Adapting SMT-LIB backend for Rosette

This section will introduce the SMT-LIB functional backend (*write_functional_smt2*) and what changes are needed to work with another s-expression target, Rosette (*write_functional_rosette*).

Overview

> Rosette is a solver-aided programming language that extends Racket with language constructs for program synthesis, verification, and more. To verify or synthesize code, Rosette compiles it to logical constraints solved with off-the-shelf SMT solvers.
>
> —https://emina.github.io/rosette/

Rosette, being backed by SMT solvers and written with s-expressions, uses code very similar to the *write_functional_smt2* output. As a result, the SMT-LIB functional backend can be used as a starting point for implementing a Rosette backend.

Full code listings for the initial SMT-LIB backend and the converted Rosette backend are included in the Yosys source repository under `backends/functional` as `smtlib.cc` and `smtlib_rosette.cc` respectively. Note that the Rosette language is an extension of the Racket language; this guide tends to refer to Racket when talking about the underlying semantics/syntax of the language.

The major changes from the SMT-LIB backend are as follows:

- all of the `Smt` prefixes in names are replaced with `Smtr` to mean `smtlib_rosette`;
- syntax is adjusted for Racket;
- data structures for input/output/state are changed from using `declare-datatype` with statically typed fields, to using `struct` with no static typing;
- the transfer function also loses its static typing;
- sign/zero extension in Rosette use the output width instead of the number of extra bits, gaining static typing;
- the single scope is traded for a global scope with local scope for each struct;
- initial state is provided as a constant value instead of a set of assertions;
- and the `-provides` option is introduced to more easily use generated code within Rosette based applications.

Scope

Our first addition to the *minimal backend* above is that for both SMT-LIB and Rosette backends, we are now targetting real languages which bring with them their own sets of constraints with what we can use as identifiers. This is where the `Functional::Scope` class described above comes in; by using this class we can safely rename our identifiers in the generated output without worrying about collisions or illegal names/characters.

In the SMT-LIB version, the `SmtScope` class implements `Scope<int>`; provides a constructor that iterates over a list of reserved keywords, calling `reserve` on each; and defines the `is_character_legal` method to reject any characters which are not allowed in SMT-LIB variable names to then be replaced with underscores in the output. To use this scope we create an instance of it, and call the `Scope::unique_name()` method to generate a unique and legal name for each of our identifiers.

In the Rosette version we update the list of legal ascii characters in the `is_character_legal` method to only those allowed in Racket variable names.

Listing 4.9: diff of `Scope` class

```
-struct SmtScope : public Functional::Scope<int> {
-        SmtScope() {
+struct SmtrScope : public Functional::Scope<int> {
+        SmtrScope() {
```

(continues on next page)

(continued from previous page)

```
                for(const char **p = reserved_keywords; *p != nullptr; p++)
                        reserve(*p);
        }
        bool is_character_legal(char c, int index) override {
-               return isascii(c) && (isalpha(c) || (isdigit(c) && index > 0) || strchr(
↪"~!@$%^&*_-+=<>.?/", c));
+               return isascii(c) && (isalpha(c) || (isdigit(c) && index > 0) || strchr(
↪"@$%^&_+=.", c));
        }
};
```

For the reserved keywords we trade the SMT-LIB specification for Racket to prevent parts of our design from accidentally being treated as Racket code. We also no longer need to reserve `pair`, `first`, and `second`. In *write_functional_smt2* these are used for combining the `(inputs, current_state)` and `(outputs, next_state)` into a single variable. Racket provides this functionality natively with `cons`, which we will see later.

Listing 4.10: diff of `reserved_keywords` list

```
 const char *reserved_keywords[] = {
-  // reserved keywords from the smtlib spec
-  ...
+  // reserved keywords from the racket spec
+  ...

   // reserved for our own purposes
-  "pair", "Pair", "first", "second",
-  "inputs", "state",
+  "inputs", "state", "name",
   nullptr
 };
```

Note

We skip over the actual list of reserved keywords from both the smtlib and racket specifications to save on space in this document.

Sort

Next up in *write_functional_smt2* we see the `Sort` class. This is a wrapper for the `Functional::Sort` class, providing the additional functionality of mapping variable declarations to s-expressions with the `to_sexpr()` method. The main change from `SmtSort` to `SmtrSort` is a syntactical one with signals represented as `bitvectors`, and memories as `lists` of signals.

Listing 4.11: diff of `Sort` wrapper

```
        SExpr to_sexpr() const {
                if(sort.is_memory()) {
-                       return list("Array", list("_", "BitVec", sort.addr_width()),␣
↪list("_", "BitVec", sort.data_width()));
+                       return list("list", list("bitvector", sort.addr_width()), list(
```

(continues on next page)

(continued from previous page)

```
↪"bitvector", sort.data_width()));
                } else if(sort.is_signal()) {
-                       return list("_", "BitVec", sort.width());
+                       return list("bitvector", sort.width());
                } else {
                        log_error("unknown sort");
                }
        }
```

Struct

As we saw in the *minimal backend* above, the `Functional::IR` class tracks the set of inputs, the set of outputs, and the set of "state" variables. The SMT-LIB backend maps each of these sets into its own `SmtStruct`, with each variable getting a corresponding field in the struct and a specified *Sort*. *`write_functional_smt2`* then defines each of these structs as a new `datatype`, with each element being strongly-typed.

In Rosette, rather than defining new datatypes for our structs, we use the native `struct`. We also only declare each field by name because Racket provides less static typing. For ease of use, we provide the expected type for each field as comments.

Listing 4.12: diff of `write_definition` method

```
        void write_definition(SExprWriter &w) {
-               w.open(list("declare-datatype", name));
-               w.open(list());
-               w.open(list(name));
-               for(const auto &field : fields)
-                       w << list(field.accessor, field.sort.to_sexpr());
-               w.close(3);
+               vector<SExpr> field_list;
+               for(const auto &field : fields) {
+                       field_list.emplace_back(field.name);
+               }
+               w.push();
+               w.open(list("struct", name, field_list, "#:transparent"));
+               if (field_names.size()) {
+                       for (const auto &field : fields) {
+                               auto bv_type = field.sort.to_sexpr();
+                               w.comment(field.name + " " + bv_type.to_string());
+                       }
+               }
+               w.pop();
        }
```

Each field is added to the `SmtStruct` with the `insert` method, which also reserves a unique name (or accessor) within the *Scope*. These accessors combine the struct name and field name and are globally unique, being used in the `access` method for reading values from the input/current state.

Listing 4.13: `Struct::access()` method

```
        SExpr access(SExpr record, IdString name) {
                size_t i = field_names.at(name);
                return list(fields[i].accessor, std::move(record));
```

(continues on next page)

(continued from previous page)

```
        }
```

In Rosette, struct fields are accessed as `<struct_name>-<field_name>` so including the struct name in the field name would be redundant. For *write_functional_rosette* we instead choose to make field names unique only within the struct, while accessors are unique across the whole module. We thus modify the class constructor and `insert` method to support this; providing one scope that is local to the struct (`local_scope`) and one which is shared across the whole module (`global_scope`), leaving the `access` method unchanged.

Listing 4.14: diff of struct constructor

```
-       SmtStruct(std::string name, SmtScope &scope) : scope(scope), name(name) {}
-       void insert(IdString field_name, SmtSort sort) {
+       SmtrStruct(std::string name, SmtrScope &scope) : global_scope(scope), local_
↪scope(), name(name) {}
+       void insert(IdString field_name, SmtrSort sort) {
                field_names(field_name);
-               auto accessor = scope.unique_name("\\" + name + "_" + field_name.
↪unescape());
-               fields.emplace_back(Field{sort, accessor});
+               auto base_name = local_scope.unique_name(field_name);
+               auto accessor = name + "-" + base_name;
+               global_scope.reserve(accessor);
+               fields.emplace_back(Field{sort, accessor, base_name});
        }
```

Finally, `SmtStruct` also provides a `write_value` template method which calls a provided function on each element in the struct. This is used later for assigning values to the output/next state pair. The only change here is to remove the check for zero-argument constructors since this is not necessary with Rosette `structs`.

Listing 4.15: diff of `write_value` method

```
        template<typename Fn> void write_value(SExprWriter &w, Fn fn) {
-               if(field_names.empty()) {
-                       // Zero-argument constructors in SMTLIB must not be called as␣
↪functions.
-                       w << name;
-               } else {
-                       w.open(list(name));
-                       for(auto field_name : field_names) {
-                               w << fn(field_name);
-                               w.comment(field_name.unescape(), true);
-                       }
-                       w.close();
+               w.open(list(name));
+               for(auto field_name : field_names) {
+                       w << fn(field_name);
+                       w.comment(field_name.unescape(), true);
                }
+               w.close();
        }
```

PrintVisitor

Remember in the *minimal backend* we converted nodes into strings for writing using the `node.to_string()` method, which wrapped `node.visit()` with a private visitor. We now want a custom visitor which can convert nodes into s-expressions. This is where the `PrintVisitor` comes in, implementing the abstract `Functional::AbstractVisitor` class with a return type of `SExpr`. For most functions, the Rosette output is very similar to the corresponding SMT-LIB function with minor adjustments for syntax.

Listing 4.16: portion of `Functional::AbstractVisitor` implementation diff showing similarities

```
        SExpr logical_shift_left(Node, Node a, Node b) override { return list("bvshl",␣
↪n(a), extend(n(b), b.width(), a.width())); }
        SExpr logical_shift_right(Node, Node a, Node b) override { return list("bvlshr",␣
↪n(a), extend(n(b), b.width(), a.width())); }
        SExpr arithmetic_shift_right(Node, Node a, Node b) override { return list("bvashr
↪", n(a), extend(n(b), b.width(), a.width())); }
-       SExpr mux(Node, Node a, Node b, Node s) override { return list("ite", to_
↪bool(n(s)), n(b), n(a)); }
-       SExpr constant(Node, RTLIL::Const const &value) override { return smt_
↪const(value); }
-       SExpr memory_read(Node, Node mem, Node addr) override { return list("select",␣
↪n(mem), n(addr)); }
-       SExpr memory_write(Node, Node mem, Node addr, Node data) override { return list(
↪"store", n(mem), n(addr), n(data)); }
+       SExpr mux(Node, Node a, Node b, Node s) override { return list("if", to_
↪bool(n(s)), n(b), n(a)); }
+       SExpr constant(Node, RTLIL::Const const& value) override { return list("bv",␣
↪smt_const(value), value.size()); }
+       SExpr memory_read(Node, Node mem, Node addr) override { return list("list-ref-bv
↪", n(mem), n(addr)); }
+       SExpr memory_write(Node, Node mem, Node addr, Node data) override { return list(
↪"list-set-bv", n(mem), n(addr), n(data)); }
```

However there are some differences in the two formats with regards to how booleans are handled, with Rosette providing built-in functions for conversion.

Listing 4.17: portion of `Functional::AbstractVisitor` implementation diff showing differences

```
        SExpr from_bool(SExpr &&arg) {
-               return list("ite", std::move(arg), "#b1", "#b0");
+               return list("bool->bitvector", std::move(arg));
        }
        SExpr to_bool(SExpr &&arg) {
-               return list("=", std::move(arg), "#b1");
+               return list("bitvector->bool", std::move(arg));
        }
```

Of note here is the rare instance of the Rosette implementation *gaining* static typing rather than losing it. Where SMT_LIB calls zero/sign extension with the number of extra bits needed (given by `out_width - a.width()`), Rosette instead specifies the type of the output (given by `list("bitvector", out_width)`).

Listing 4.18: zero/sign extension implementation diff

```
-       SExpr zero_extend(Node, Node a, int out_width) override { return list(list("_",
↪"zero_extend", out_width - a.width()), n(a)); }
-       SExpr sign_extend(Node, Node a, int out_width) override { return list(list("_",
↪"sign_extend", out_width - a.width()), n(a)); }
+       SExpr zero_extend(Node, Node a, int out_width) override { return list("zero-
↪extend", n(a), list("bitvector", out_width)); }
+       SExpr sign_extend(Node, Node a, int out_width) override { return list("sign-
↪extend", n(a), list("bitvector", out_width)); }
```

Note

Be sure to check the source code for the full list of differences here.

Module

With most of the supporting classes out of the way, we now reach our three main steps from the *minimal backend*. These are all handled by the `SmtModule` class, with the mapping from RTLIL module to FunctionalIR happening in the constructor. Each of the three `SmtStruct`s; inputs, outputs, and state; are also created in the constructor, with each value in the corresponding lists in the IR being `insert`ed.

Listing 4.19: `SmtModule` constructor

```
        SmtModule(Module *module)
                : ir(Functional::IR::from_module(module))
                , scope()
                , name(scope.unique_name(module->name))
                , input_struct(scope.unique_name(module->name.str() + "_Inputs"), scope)
                , output_struct(scope.unique_name(module->name.str() + "_Outputs"),␣
↪scope)
                , state_struct(scope.unique_name(module->name.str() + "_State"), scope)
        {
                scope.reserve(name + "-initial");
                for (auto input : ir.inputs())
                        input_struct.insert(input->name, input->sort);
                for (auto output : ir.outputs())
                        output_struct.insert(output->name, output->sort);
                for (auto state : ir.states())
                        state_struct.insert(state->name, state->sort);
        }
```

Since Racket uses the `-` to access struct fields, the `SmtrModule` instead uses an underscore for the name of the initial state.

Listing 4.20: diff of `Module` constructor

```
-               scope.reserve(name + "-initial");
+               scope.reserve(name + "_initial");
+               if (assoc_list_helpers) {
+                       input_helper_name = scope.unique_name(module->name.str() + "_
↪inputs_helper");
```

(continues on next page)

(continued from previous page)

```
+                       scope.reserve(*input_helper_name);
+                       output_helper_name = scope.unique_name(module->name.str() + "_
↪outputs_helper");
+                       scope.reserve(*output_helper_name);
+               }
```

The `write` method is then responsible for writing the FunctionalIR to the output file, formatted for the corresponding backend. `SmtModule::write()` breaks the output file down into four parts: defining the three structs, declaring the `pair` datatype, defining the transfer function `(inputs, current_state) -> (outputs, next_state)` with `write_eval`, and declaring the initial state with `write_initial`. The only change for the `SmtrModule` is that the `pair` declaration isn't needed.

Listing 4.21: diff of `Module::write()` method

```
        void write(std::ostream &out)
        {
                SExprWriter w(out);

                input_struct.write_definition(w);
                output_struct.write_definition(w);
                state_struct.write_definition(w);

-               w << list("declare-datatypes",
-                       list(list("Pair", 2)),
-                       list(list("par", list("X", "Y"), list(list("pair", list("first",
↪ "X"), list("second", "Y"))))));
-
+               if (use_assoc_list_helpers) {
+                       write_assoc_list_helpers(w);
+               }
```

The `write_eval` method is where the FunctionalIR nodes, outputs, and next state are handled. Just as with the *minimal backend*, we iterate over the nodes with `for(auto n : ir)`, and then use the `Struct::write_value()` method for the `output_struct` and `state_struct` to iterate over the outputs and next state respectively.

Listing 4.22: iterating over FunctionalIR nodes in `SmtModule::write_eval()`

```
                for(auto n : ir)
                        if(!inlined(n)) {
                                w.open(list("let", list(list(node_to_sexpr(n), n.
↪visit(visitor)))), false);
                                w.comment(SmtSort(n.sort()).to_sexpr().to_string(),␣
↪true);
                        }
```

The main differences between our two backends here are syntactical. First we change the `define-fun` for the Racket style `define` which drops the explicitly typed inputs/outputs. And then we change the final result from a `pair` to the native `cons` which acts in much the same way, returning both the `outputs` and the `next_state` in a single variable.

Listing 4.23: diff of `Module::write_eval()` transfer function declaration

```
-                w.open(list("define-fun", name,
-                        list(list("inputs", input_struct.name),
-                             list("state", state_struct.name)),
-                        list("Pair", output_struct.name, state_struct.name)));
+                w.open(list("define", list(name, "inputs", "state")));
```

Listing 4.24: diff of output/next state handling `Module::write_eval()`

```
-                w.open(list("pair"));
+                w.open(list("cons"));
                output_struct.write_value(w, [&](IdString name) { return node_to_
↪sexpr(ir.output(name).value()); });
                state_struct.write_value(w, [&](IdString name) { return node_to_sexpr(ir.
↪state(name).next_value()); });
                w.pop();
```

For the `write_initial` method, the SMT-LIB backend uses `declare-const` and `asserts` which must always hold true. For Rosette we instead define the initial state as any other variable that can be used by external code. This variable, `[name]_initial`, can then be used in the `[name]` function call; allowing the Rosette code to be used in the generation of the `next_state`, whereas the SMT-LIB code can only verify that a given `next_state` is correct.

Listing 4.25: diff of `Module::write_initial()` method

```
        void write_initial(SExprWriter &w)
        {
-                std::string initial = name + "-initial";
-                w << list("declare-const", initial, state_struct.name);
+                w.push();
+                auto initial = name + "_initial";
+                w.open(list("define", initial));
+                w.open(list(state_struct.name));
                for (auto state : ir.states()) {
-                        if(state->sort.is_signal())
-                                w << list("assert", list("=", state_struct.
↪access(initial, state->name), smt_const(state->initial_value_signal())));
-                        else if(state->sort.is_memory()) {
+                        if (state->sort.is_signal())
+                                w << list("bv", smt_const(state->initial_value_
↪signal()), state->sort.width());
+                        else if (state->sort.is_memory()) {
                                const auto &contents = state->initial_value_memory();
+                                w.open(list("list"));
                                for(int i = 0; i < 1<<state->sort.addr_width(); i++) {
-                                        auto addr = smt_const(RTLIL::Const(i, state->
↪sort.addr_width()));
-                                        w << list("assert", list("=", list("select",␣
↪state_struct.access(initial, state->name), addr), smt_const(contents[i])));
+                                        w << list("bv", smt_const(contents[i]), state->
```

(continues on next page)

(continued from previous page)

```
↪sort.data_width());
                                }
+                                w.close();
                        }
                }
+                w.pop();
+        }
+
```

Backend

The final part is the `Backend` itself, with much of the same boiler plate as the *minimal backend*. The main difference is that we use the *Module* to perform the actual processing.

Listing 4.26: The `FunctionalSmtBackend`

```
struct FunctionalSmtBackend : public Backend {
        FunctionalSmtBackend() : Backend("functional_smt2", "Generate SMT-LIB from␣
↪Functional IR") {}

        void help() override { log("\nFunctional SMT Backend.\n\n"); }

        void execute(std::ostream *&f, std::string filename, std::vector<std::string>␣
↪args, RTLIL::Design *design) override
        {
                log_header(design, "Executing Functional SMT Backend.\n");

                size_t argidx = 1;
                extra_args(f, filename, args, argidx, design);

                for (auto module : design->selected_modules()) {
                        log("Processing module `%s`.\n", module->name);
                        SmtModule smt(module);
                        smt.write(*f);
                }
        }
} FunctionalSmtBackend;
```

There are two additions here for Rosette. The first is that the output file needs to start with the `#lang` definition which tells the compiler/interpreter that we want to use the Rosette language module. The second is that the *`write_functional_rosette`* command takes an optional argument, `-provides`. If this argument is given, then the output file gets an additional line declaring that everything in the file should be exported for use; allowing the file to be treated as a Racket package with structs and mapping function available for use externally.

Listing 4.27: relevant portion of diff of `Backend::execute()` method

```
+                *f << "#lang rosette/safe\n";
+                if (provides) {
+                        *f << "(provide (all-defined-out))\n";
+                }
```

(continues on next page)

(continued from previous page)

4.3.4 Identifying the root cause of bugs

This document references Yosys internals and is intended for people interested in solving or investigating Yosys bugs. This also applies if you are using a fuzzing tool; fuzzers have a tendency to find many variations of the same bug, so identifying the root cause is important for avoiding issue spam.

If you're familiar with C/C++, you might try to have a look at the source code of the command that's failing. Even if you can't fix the problem yourself, it can be very helpful for anyone else investigating if you're able to identify where the issue is arising.

Finding the failing command

Using the `-L` flag can help here, allowing you to specify a file to log to, such as `yosys -L out.log -s script.ys`. Most commands will print a header message when they begin; something like `2.48. Executing HIERARCHY pass (managing design hierarchy).` The last header message will usually be the failing command. There are some commands which don't print a header message, so you may want to add `echo on` to the start of your script. The `echo` command echoes each command executed, along with any arguments given to it. For the `hierarchy` example above this might be `yosys> hierarchy -check`.

Note

It may also be helpful to use the *`log`* command to add messages which you can then search for either in the terminal or the logfile. This can be quite useful if your script contains script-passes, like the *Synth commands*, which call many sub-commands and you're not sure exactly which script-pass is calling the failing command.

Minimizing scripts

Warning

This section is intended as **advanced usage**, and generally not necessary for normal bug reports.

If you're using a command line prompt, such as `yosys -p 'synth_xilinx' -o design.json design.v`, consider converting it to a script. It's generally much easier to iterate over changes to a script in a file rather than one on the command line, as well as being better for sharing with others.

Listing 4.28: example script, `script.ys`, for prompt `yosys -p 'synth_xilinx' -o design.json design.v`

```
read_verilog design.v
synth_xilinx
write_json design.json
```

Next up you want to remove everything *after* the error occurs. If your final command calls sub-commands, replace it with its contents first. In the case of the *Synth commands*, as well as certain other script-passes, you can use the `-run` option to simplify this. For example we can replace `synth -top <top> -lut` with the *example replacement script for synth command*. The options `-top <top> -lut` can be provided to each `synth` step, or to just the step(s) where it is relevant, as done here.

Listing 4.29: example replacement script for `synth` command

```
synth -top <top> -run :coarse
synth -lut -run coarse:fine
synth -lut -run fine:check
synth -run check:
```

Say we ran *example replacement script for synth command* and were able to remove the `synth -run check:` and still got our error, then we check the log and we see the last thing before the error was `7.2. Executing MEMORY_MAP pass (converting memories to logic and flip-flops)`. By checking the output of `yosys -h synth` (or the `synth` help page) we can see that the *`memory_map`* pass is called in the `fine` step. We can then update our script to the following:

Listing 4.30: example replacement script for `synth` when *`memory_map`* is failing

```
synth -top <top> -run :fine
opt -fast -full
memory_map
```

By giving `synth` the option `-run :fine`, we are telling it to run from the beginning of the script until the `fine` step, where we then give it the exact commands to run. There are some cases where the commands given in the help output are not an exact match for what is being run, but are instead a simplification. If you find that replacing the script-pass with its contents causes the error to disappear, or change, try calling the script-pass with `echo on` to see exactly what commands are being called and what options are used.

Warning

Before continuing further, *back up your code*. The following steps can remove context and lead to over-minimizing scripts, hiding underlying issues. Check out *Why context matters* to learn more.

When a problem is occurring many steps into a script, minimizing the design at the start of the script isn't always enough to identify the cause of the issue. Each extra step of the script can lead to larger sections of the input design being needed for the specific problem to be preserved until it causes a crash. So to find the smallest possible reproducer it can sometimes be helpful to remove commands prior to the failure point.

The simplest way to do this is by writing out the design, resetting the current state, and reading back the design:

```
write_rtlil <design.il>; design -reset; read_rtlil <design.il>;
```

In most cases, this can be inserted immediately before the failing command while still producing the error, allowing you to *minimize your RTLIL* with the `<design.il>` output. For our previous example with *`memory_map`*, if *resetting the design immediately before failure* still gives the same error, then we should now be able to call `yosys design.il -p 'memory_map'` to reproduce it.

Listing 4.31: resetting the design immediately before failure

```
synth -top <top> -run :fine
opt -fast -full
write_rtlil design.il; design -reset; read_rtlil design.il;
memory_map
```

If that doesn't give the error (or doesn't give the same error), then you should try to move the write/reset/read

earlier in the script until it does. If you have no idea where exactly you should put the reset, the best way is to use a "binary search" type approach, reducing the possible options by half after each attempt.

> **Note**
>
> By default, *write_rtlil* doesn't include platform specific IP blocks and other primitive cell models which are typically loaded with a `read_verilog -lib` command at the start of the synthesis script. You may have to duplicate these commands *after* the call to `design -reset`. It is also possible to write out *everything* with `select =*; write_rtlil -selected <design.il>`.

As an example, your script has 16 commands in it before failing on the 17th. If resetting immediately before the 17th doesn't reproduce the error, try between the 8th and 9th (8 is half of the total 16). If that produces the error then you can remove everything before the *read_rtlil* and try reset again in the middle of what's left, making sure to use a different name for the output file so that you don't overwrite what you've already got. If the error isn't produced then you need to go earlier still, so in this case you would do between the 4th and 5th (4 is half of the previous 8). Repeat this until you can't reduce the remaining commands any further.

A more conservative, but more involved, method is to remove or comment out commands prior to the failing command. Each command, or group of commands, can be disabled one at a time while checking if the error still occurs, eventually giving the smallest subset of commands needed to take the original input through to the error. The difficulty with this method is that depending on your design, some commands may be necessary even if they aren't needed to reproduce the error. For example, if your design includes `process` blocks, many commands will fail unless you run the *proc* command. While this approach can do a better job of maintaining context, it is often easier to *recover* the context after the design has been minimized for producing the error. For more on recovering context, checkout *Why context matters*.

Why context matters

Sometimes when a command is raising an error, you're seeing a symptom rather than the underlying issue. It's possible that an earlier command may be putting the design in an invalid state, which isn't picked up until the error is raised. This is particularly true for the pre-packaged *Synth commands*, which rely on a combination of generic and architecture specific passes. As new features are added to Yosys and more designs are supported, the types of cells output by a pass can grow and change; and sometimes this leads to a mismatch in what a pass is intended to handle.

If you minimized your script, and removed commands prior to the failure to get a smaller reproducer, try to work backwards and find which commands may have contributed to the design failing. From the minimized design you should have some understanding of the cell or cells which are producing the error; but where did those cells come from? The name and/or type of the cell can often point you in the right direction:

```
# internal cell types start with a $
# lowercase for word-level, uppercase for bit-level
$and
$_AND_

# cell types with $__ are typically intermediate cells used in techmapping
$__MUL16X16

# cell types without a $ are either user-defined or architecture specific
my_module
SB_MAC16

# object names might give you the name of the pass that created them
```

(continues on next page)

(continued from previous page)

```
$procdff$1204
$memory\rom$rdmux[0][0][0]$a$1550

# or even the line number in the Yosys source
$auto$muxcover.cc:557:implement_best_cover$2152
$auto$alumacc.cc:495:replace_alu$1209
```

Try running the unminimized script and search the log for the names of the objects in your minimized design. In the case of cells you can also search for the type of the cell. Remember that calling *stat* will list all the types of cells currently used in the design, and `select -list =*` will list the names of of all the current objects. You can add these commands to your script, or use an interactive terminal to run each command individually. Adding them to the script can be more repeatable, but if it takes a long time to run to the point you're interested in then an interactive shell session can give you more flexibility once you reach that point. You can also add a call to the `shell` command at any point in a script to start an interactive session at a given point; allowing you to script any preparation steps, then come back once it's done.

The `--dump-design` option

Yosys provides the `--dump-design` option (or `-P` for short) for dumping the design at specific steps of the script based on the log header. If the last step before an error is `7.2. Executing MEMORY_MAP pass (converting memories to logic and flip-flops)`, then calling Yosys with `--dump-design 7.2:bad.il` will save the design *before* this command runs, in the file `bad.il`.

It is also possible to use this option multiple times, e.g. `-P2:hierarchy.il -P7 -P7.2:bad.il`, to get multiple dumps in the same run. This can make it easier to follow the design through each step to find where certain cells or connections are coming from. `--dump-design ALL` is also allowed, writing out the design at each log header.

A worked example

Say you did all the minimization and found that an error in `synth_xilinx` occurs when a call to `techmap -map +/xilinx/cells_map.v` with `MIN_MUX_INPUTS` defined parses a *$_MUX16_* with all inputs set to `1'x`. You could fix the bug in `+/xilinx/cells_map.v`, but that might only solve this one case while leaving other problems that haven't been found yet. So you step through the original script, calling *stat* after each step to find when the *$_MUX16_* is added.

You find that the *$_MUX16_* is introduced by a call to `muxcover`, but all the inputs are defined, so calling `techmap` now works as expected. From running `bugpoint` with the failing techmap you know that the cell with index `2297` will fail, so you call `select top/*$2297` to limit to just that cell. This can then be saved with `design -save pre_bug` or `write_rtlil -selected pre_bug.il`, so that you don't have to re-run all the earlier steps to get back here.

Next you step through the remaining commands and call `dump` after each to find when the inputs are disconnected. You find that `opt -full` has optimized away portions of the circuit, leading to *opt_expr* setting the undriven mux inputs to `x`, but failing to remove the now unnecessary *$_MUX16_*. Now you've identified a problem in *opt_expr* that affects all of the wide muxes, and could happen in any synthesis flow, not just `synth_xilinx`.

See also

This example is taken from YosysHQ/yosys#4590 and can be reproduced with a version of Yosys between 0.45 and 0.51.

4.3.5 Contributing to Yosys

Reporting bugs

We fix well-reported bugs the fastest. A good bug report is an issue on the issue tracker and includes the following information:

Title

briefly describe the issue, for example:

> techmap of wide mux with undefined inputs raises error during synth_xilinx

- tells us what's happening ("raises error")
- gives the command affected (`techmap`)
- an overview of the input design ("wide mux with undefined inputs")
- and some context where it was found ("during `synth_xilinx`")

Reproduction Steps

The reproduction steps should be a minimal, complete and verifiable example MVCE. Providing an MVCE with your bug report drastically increases the likelihood that someone will be able to help resolve your issue. Make sure that your report input is free of any problems as reported by the *`check`* command. One way to minimize a design is to use the `bugpoint` command. You can learn more in the *how-to guide for bugpoint*.

The reproduction steps are ideally a code-block (starting and ending with triple backquotes) containing the minimized design (Verilog or RTLIL), followed by a code-block containing the minimized yosys script OR a command line call to yosys with code-formatting (starting and ending with single backquotes).

````
min.v
```verilog
// minimized Verilog design
```

min.ys
```
read_verilog min.v
# minimum sequence of commands to reproduce error
```

OR

`yosys -p ': minimum sequence of commands;' min.v`
````

Alternatively, you can provide a single code-block which includes the minimized design as a "here document" followed by the sequence of commands which reproduce the error

- see *Loading a design* for more on heredocs.

````
```
read_rtlil <<EOF
# minimized RTLIL design
EOF
# minimum sequence of commands
```
````

Don't forget to mention:

- any important environment variables or command line options
- if the problem occurs for a range of values/designs, what is that range
- if you're using an external tool, such as `valgrind`, to detect the issue, what version of that tool are you using and what options are you giving it

Warning

Please try to avoid the use of any external plugins/tools in the reproduction steps if they are not directly related to the issue being raised. This includes frontend plugins such as GHDL or slang; use *write_rtlil* on the minimized design instead. This also includes tools which provide a wrapper around Yosys such as OpenLane; you should instead minimize your input and reproduction steps to just the Yosys part.

Expected Behaviour

Describe what you'd expect to happen when we follow the reproduction steps if the bug was fixed.

If you have a similar design/script that doesn't give the error, include it here as a reference. If the bug is that an error *should* be raised but isn't, note if there are any other commands with similar error messages.

Actual Behaviour

Describe what you actually see when you follow the reproduction steps.

This can include:

- any error messages
- any details relevant to the crash that were found with `--trace` or `--debug` flags
- the part of the source code that triggers the bug
  - if possible, use a permalink to the source on GitHub
  - you can browse the source repository for a certain commit with the failure and open the source file, select the relevant lines (click on the line number for the first relevant line, then while holding shift click on the line number for the last relevant line), click on the `...` that appears and select "Copy permalink"
  - should look something like `https://github.com/YosysHQ/yosys/blob/<commit_hash>/path/to/file#L139-L147`
  - clicking on "Preview" should reveal a code block containing the lines of source specified, with a link to the source file at the given commit

Additional Details

Anything else you think might be helpful or relevant when verifying or fixing the bug.

Once you have created the issue, any additional details can be added as a comment on that issue. You can include any additional context as to what you were doing when you first encountered the bug.

If this issue discovered through the use of a fuzzer, ALWAYS declare that. If you've minimized the script, consider including the `bugpoint` script you used, or the original script, for example:

```
Minimized with
```
read_verilog design.v
original sequence of commands prior to error
bugpoint -script <failure.ys> -grep "<string>"
write_rtlil min.il
```

OR

Minimized from
`yosys -p ': original sequence of commands to produce error;' design.v`
```

If possible, it may also help to share the original un-minimized design. If the design is too big for a comment, consider turning it into a Gist

Contributing code

Code that matters

If you're adding complex functionality, or modifying core parts of yosys, we highly recommend discussing your motivation and approach ahead of time on the Discourse forum. Please, be as explicit and concrete as possible when explaining the motivation for what you're building. Additionally, if you do so on the forum first before you starting hacking away at C++, you might solve your problem without writing a single line of code!

PRs are considered for relevance, priority, and quality based on their descriptions first, code second.

Before you build or fix something, also search for existing issues.

We have open developer 'jour fixe' (Dev JF) meetings where developers from YosysHQ and the community come together to discuss open issues and PRs. This is also a good place to talk to us about how to implement larger PRs.

Making sense

Given enough effort, the behavior of any code can be figured out to any desired extent. However, the author of the code is by far in the best position to make this as easy as possible.

Yosys is a long-standing project and has accumulated a lot of C-style code that's not written to be read, just written to run. We improve this bit by bit when opportunities arise, but it is what it is. New additions are expected to be a lot cleaner.

The purpose and behavior of the code changed should be described clearly. Your change should contain exactly what it needs to match that description. This means:

- nothing more than that - no dead code, no undocumented features
- nothing missing - if something is partially built, that's fine, but you have to make that clear. For example, some passes only support some types of cells

Here are some software engineering approaches that help:

- Use abstraction to model the problem and hide details
  - Maximize the usage of types (structs over loose variables), not necessarily in an object-oriented way
  - Use functions, scopes, type aliases

- In new passes, make sure the logic behind how and why it works is actually provided in coherent comments, and that variable and type naming is consistent with the terms you use in the description.
- The logic of the implementation should be described in mathematical or algorithm theory terms. Correctness, termination, computational complexity. Make it clear if you're re-implementing a classic data structure for logic synthesis or graph traversal etc.
- There's various ways of traversing the design with use-def indices (for getting drivers and driven signals) available in Yosys. They have advantages and sometimes disadvantages. Prefer not re-implementing these
- Prefer references over pointers, and smart pointers over raw pointers
- Aggressively deduplicate code. Within functions, within passes, across passes, even against existing code
- Prefer declaring things `const`
- Prefer range-based for loops over C-style

Common mistakes

- Deleting design objects invalidates iterators. Defer deletions or hold a copy of the list of pointers to design objects
- Deleting wires can get sketchy and is intended to be done solely by the `opt_clean` pass so just don't do it
- Iterating over an entire design and checking if things are selected is more inefficient than using the `selected_*` methods
- Remember to call `fixup_ports` at the end if you're modifying module interfaces

Testing your change

Untested code can't be maintained. Inevitable codebase-wide changes are likely to break anything untested. Tests also help reviewers understand the purpose of the code change in practice.

Your code needs to come with tests. If it's a feature, a test that covers representative examples of the added behavior. If it's a bug fix, it should reproduce the original isolated bug. But in some situations, adding a test isn't viable. If you can't provide a test, explain this decision.

Prefer writing unit tests (`tests/unit`) for isolated tests to the internals of more serious code changes, like those to the core of yosys, or more algorithmic ones.

The rest of the test suite is mostly based on running Yosys on various Yosys and Tcl scripts that manually call Yosys commands. See *Testing Yosys* for more information about how our test suite is structured. The basic test writing approach is checking for the presence of some kind of object or pattern with `-assert-count` in *Selections*.

It's often best to use equivalence checking with `equiv_opt -assert` or similar to prove that the changes done to the design by a modified pass preserve equivalence. But some code isn't meant to preserve equivalence. Sometimes proving equivalence takes an impractically long time for larger inputs. Also beware, the `equiv_` passes are a bit quirky and might even have incorrect results in unusual situations.

Coding style

Yosys is written in C++20.

In general Yosys uses `int` instead of `size_t`. To avoid compiler warnings for implicit type casts, always use `GetSize(foobar)` instead of `foobar.size()`. (`GetSize()` is defined in `kernel/yosys.h`)

For auto formatting code, a `.clang-format` file is present top-level. Yosys code is using tabs for indentation. A tab is 8 characters wide, but prefer not relying on it. A continuation of a statement in the following line is indented by two additional tabs. Lines are as long as you want them to be. A good rule of thumb is to break lines at about column 150. Opening braces can be put on the same or next line as the statement opening the block (if, switch, for, while, do). Put the opening brace on its own line for larger blocks, especially blocks that contains blank lines. Remove trailing whitespace on sight.

Otherwise stick to the Linux Kernel Coding Style.

Pull requests (PRs)

If you are working on something to add to Yosys, or fix something that isn't working quite right, make a pull request (PR).

An open PR, even as a draft, tells everyone that you're working on it and they don't have to. It can also be a useful way to solicit feedback on in-progress changes.

We use labels to help categorise issues and PRs. If a label seems relevant to your work, please do add it; this also includes the labels beginning with 'status-'. The 'merge-' labels are used by maintainers for tracking and communicating which PRs are ready and pending merge; please do not use these labels if you are not a maintainer.

Git style

We don't have a strict commit message style.

Some style hints:

- Refactor and document existing code if you touch it, but in separate commits from your functional changes
- Prefer smaller commits organized by good chunks. Git has a lot of features like fixup commits, interactive rebase with autosquash

Reviewing PRs

Reviewing PRs is a totally valid form of external contributing to the project!

Who's the reviewer?

YosysHQ GmbH is a company with a mandate to make decisions for the good of YosysHQ open source software. It was co-founded by Claire Xenia Wolf, the original author of Yosys. Within it, we allocate reviews based on expertise with the topic at hand as well as member time constraints. However, decisions including reviews are also contributed by people external to the company.

If you're intimately acquainted with a part of the codebase, we will be happy to defer to your experience and have you review PRs. The official way we like is our CODEOWNERS file in the git repository. What we're looking for in code owners is activity and trust. For activity, if you're only interested in a yosys pass for example for the time you spend writing a thesis, it might be better to focus on writing good tests and docs in the PRs you submit rather than to commit to code ownership and therefore to be responsible for fixing things and reviewing other people's PRs at various unexpected points later. If you're prolific in some part of the codebase and not a code owner, we still value your experience and may tag you in PRs.

As a matter of fact, the purpose of code ownership is to avoid maintainer burnout by removing orphaned parts of the codebase. If you become a code owner and stop being responsive, in the future, we might decide to remove such code if convenient and costly to maintain. It's simply more respectful of the users' time to explicitly cut something out than let it "bitrot". Larger projects like LLVM or linux could not survive

without such things, but Yosys is far smaller, and there are implicit expectations of stability we aim to respect within reason.

Sometimes, multiple maintainers may add review comments. This is considered healthy collaborative even if it might create disagreement at times. If somebody is already reviewing a PR, others, even non-maintainers are free to leave comments with extra observations and alternate perspectives in a collaborative spirit.

How to review

First, read everything above about contributing. Those are the values you should gently enforce as a reviewer. They're ordered by importance, but explicitly, descriptions are more important than code, long-form comments describing the design are more important than piecemeal comments, etc.

If a PR is poorly described, incomplete, tests are broken, or if the author is not responding, please don't feel pressured to take over their role by reverse engineering the code or fixing things for them, unless there are good reasons to do so.

If a PR author submits LLM outputs they haven't understood themselves, they will not be able to implement feedback. Take this into consideration as well. We do not ban LLM code from the codebase, we ban bad code.

Reviewers may have diverse styles of communication while reviewing - one may do one thorough review, another may prefer a back and forth with the basics out the way before digging into the code. Generally, PRs may have several requests for modifications and long discussions, but often they just are good enough to merge as-is.

The CI is required to go green for merging. New contributors need a CI run to be triggered by a maintainer before their PRs take up computing resources. It's a single click from the github web interface. We test on various platforms and compilers. Sanitizer builds are only tested on the main branch.

4.3.6 Testing Yosys

Todo

adding tests (makefile-tests vs seed-tests)

Running the included test suite

The Yosys source comes with a test suite to avoid regressions and keep everything working as expected. Tests can be run by calling `make test` from the root Yosys directory. By default, this runs vanilla and unit tests.

Vanilla tests

These make up the majority of our testing coverage. They can be run with `make vanilla-test` and are based on calls to make subcommands (`make makefile-tests`) and shell scripts (`make seed-tests` and `make abcopt-tests`). Both use `run-test.sh` files, but make-based tests only call `tests/gen-tests-makefile.sh` to generate a makefile appropriate for the given directory, so only afterwards when make is invoked do the tests actually run.

Usually their structure looks something like this: you write a .ys file that gets automatically run, which runs a frontend like `read_verilog` or `read_rtlil` with a relative path or a heredoc, then runs some commands including the command under test, and then uses *Selections* with `-assert-count`. Usually it's unnecessary to "register" the test anywhere as if it's being added to an existing directory, depending on how the `run-test.sh` in that directory works.

Unit tests

Running the unit tests requires the following additional packages:

```
sudo apt-get install libgtest-dev
```

No additional requirements.

Unit tests can be run with `make unit-test`.

Functional tests

Testing functional backends (see *Writing a new backend using FunctionalIR*) has a few requirements in addition to those listed in *Build prerequisites*:

```
sudo apt-get install racket
raco pkg install rosette
pip install pytest-xdist pytest-xdist-gnumake
```

```
brew install racket
raco pkg install rosette
pip install pytest-xdist pytest-xdist-gnumake
```

If you don't have one of the *CAD suite(s)* installed, you should also install Z3 following their instructions.

Then, set the `ENABLE_FUNCTIONAL_TESTS` make variable when calling `make test` and the functional tests will be run as well.

Docs tests

There are some additional tests for checking examples included in the documentation, which can be run by calling `make test` from the `yosys/docs` sub-directory (or `make -C docs test` from the root). This also includes checking some macro commands to ensure that descriptions of them are kept up to date, and is mostly intended for CI.

Automatic testing

The Yosys Git repo has automatic testing of builds and running of the included test suite on both Ubuntu and macOS, as well as across range of compiler versions. For up to date information, including OS versions, refer to the git actions page.

4.4 Techmap by example

As a quick recap, the `techmap` command replaces cells in the design with implementations given as Verilog code (called "map files"). It can replace Yosys' internal cell types (such as *`$or`*) as well as user-defined cell types.

- Verilog parameters are used extensively to customize the internal cell types.
- Additional special parameters are used by techmap to communicate meta-data to the map files.
- Special wires are used to instruct techmap how to handle a module in the map file.
- Generate blocks and recursion are powerful tools for writing map files.

Code examples used in this document are included in the Yosys code base under `docs/source/code_examples/techmap`.

4.4.1 Mapping OR3X1

Todo

add/expand supporting text

Note

This is a simple example for demonstration only. Techmap shouldn't be used to implement basic logic optimization.

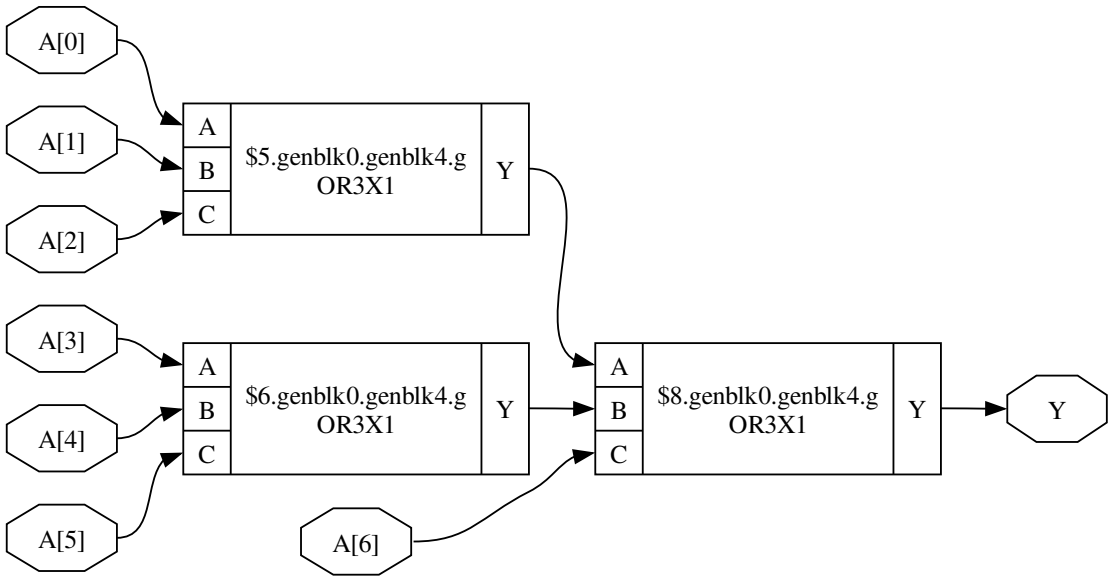
Listing 4.32: `red_or3x1_map.v`

```
module \$reduce_or (A, Y);

    parameter A_SIGNED = 0;
    parameter A_WIDTH = 0;
    parameter Y_WIDTH = 0;

    input [A_WIDTH-1:0] A;
    output [Y_WIDTH-1:0] Y;

    function integer min;
        input integer a, b;
        begin
            if (a < b)
                min = a;
            else
                min = b;
        end
    endfunction

    genvar i;
    generate begin
        if (A_WIDTH == 0) begin
            assign Y = 0;
        end
        if (A_WIDTH == 1) begin
            assign Y = A;
        end
        if (A_WIDTH == 2) begin
            wire ybuf;
            OR3X1 g (.A(A[0]), .B(A[1]), .C(1'b0), .Y(ybuf));
            assign Y = ybuf;
        end
        if (A_WIDTH == 3) begin
            wire ybuf;
            OR3X1 g (.A(A[0]), .B(A[1]), .C(A[2]), .Y(ybuf));
            assign Y = ybuf;
        end
```

(continues on next page)

(continued from previous page)

```
        if (A_WIDTH > 3) begin
            localparam next_stage_sz = (A_WIDTH+2) / 3;
            wire [next_stage_sz-1:0] next_stage;
            for (i = 0; i < next_stage_sz; i = i+1) begin
                localparam bits = min(A_WIDTH - 3*i, 3);
                assign next_stage[i] = |A[3*i +: bits];
            end
            assign Y = |next_stage;
        end
    end endgenerate
endmodule
```



Listing 4.33: `red_or3x1_test.ys`

```
read_verilog red_or3x1_test.v
hierarchy -check -top test

techmap -map red_or3x1_map.v;;

splitnets -ports
show -prefix red_or3x1 -format dot -notitle -lib red_or3x1_cells.v
```

Listing 4.34: `red_or3x1_test.v`

```
module test (A, Y);
    input [6:0] A;
    output Y;
    assign Y = |A;
```

(continues on next page)

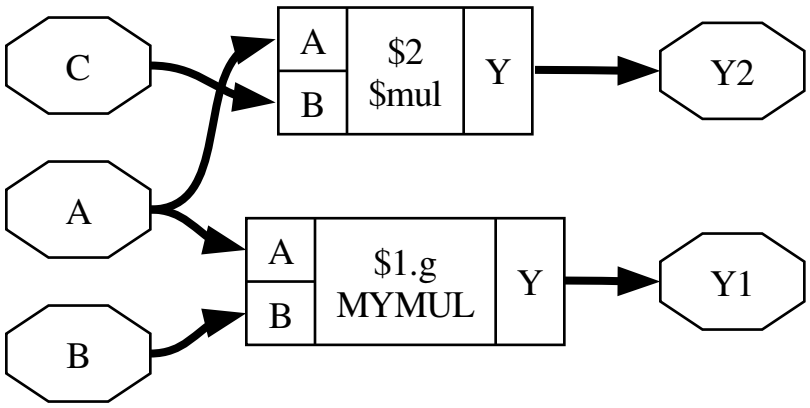
(continued from previous page)

```
endmodule
```

4.4.2 Conditional techmap

- In some cases only cells with certain properties should be substituted.
- The special wire `_TECHMAP_FAIL_` can be used to disable a module in the map file for a certain set of parameters.
- The wire `_TECHMAP_FAIL_` must be set to a constant value. If it is non-zero then the module is disabled for this set of parameters.
- Example use-cases:
  - coarse-grain cell types that only operate on certain bit widths
  - memory resources for different memory geometries (width, depth, ports, etc.)

Example:



Listing 4.35: `sym_mul_map.v`

```
module \$mul (A, B, Y);
    parameter A_SIGNED = 0;
    parameter B_SIGNED = 0;
    parameter A_WIDTH = 1;
    parameter B_WIDTH = 1;
    parameter Y_WIDTH = 1;

    input [A_WIDTH-1:0] A;
    input [B_WIDTH-1:0] B;
    output [Y_WIDTH-1:0] Y;

    wire _TECHMAP_FAIL_ = A_WIDTH != B_WIDTH || B_WIDTH != Y_WIDTH;
```

(continues on next page)

(continued from previous page)

```
    MYMUL #( .WIDTH(Y_WIDTH) ) g ( .A(A), .B(B), .Y(Y) );
endmodule
```

Listing 4.36: `sym_mul_test.v`

```
module test(A, B, C, Y1, Y2);
    input   [7:0] A, B, C;
    output  [7:0] Y1 = A * B;
    output [15:0] Y2 = A * C;
endmodule
```

Listing 4.37: `sym_mul_test.ys`

```
read_verilog sym_mul_test.v
hierarchy -check -top test

techmap -map sym_mul_map.v;;

show -prefix sym_mul -format dot -notitle -lib sym_mul_cells.v
```

4.4.3 Scripting in map modules

- The special wires `_TECHMAP_DO_*` can be used to run Yosys scripts in the context of the replacement module.
- The wire that comes first in alphabetical oder is interpreted as string (must be connected to constants) that is executed as script. Then the wire is removed. Repeat.
- You can even call techmap recursively!
- Example use-cases:
  - Using always blocks in map module: call *proc*
  - Perform expensive optimizations (such as *freduce*) on cells where this is known to work well.
  - Interacting with custom commands.

Note

PROTIP:

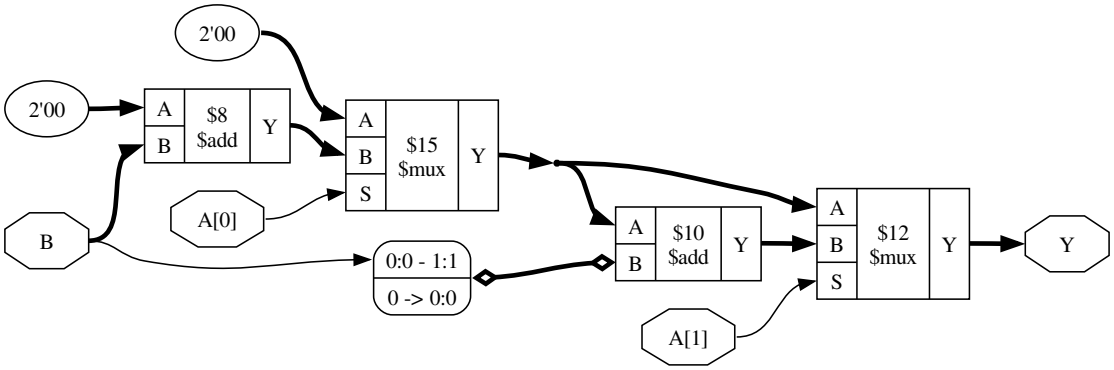
Commands such as `shell`, `show -pause`, and `dump` can be used in the `_TECHMAP_DO_*` scripts for debugging map modules.

Example:

Listing 4.38: `mymul_map.v`

```
module MYMUL(A, B, Y);
    parameter WIDTH = 1;
    input [WIDTH-1:0] A, B;
    output reg [WIDTH-1:0] Y;
```

(continues on next page)



(continued from previous page)

```
    wire [1023:0] _TECHMAP_DO_ = "proc; clean";

    integer i;
    always @* begin
        Y = 0;
        for (i = 0; i < WIDTH; i=i+1)
            if (A[i])
                Y = Y + (B << i);
    end
endmodule
```

Listing 4.39: `mymul_test.v`

```
module test(A, B, Y);
    input  [1:0] A, B;
    output [1:0] Y = A * B;
endmodule
```

Listing 4.40: `mymul_test.ys`

```
read_verilog mymul_test.v
hierarchy -check -top test

techmap -map sym_mul_map.v \
        -map mymul_map.v;;

rename test test_mapped
read_verilog mymul_test.v
miter -equiv test test_mapped miter
flatten miter

sat -verify -prove trigger 0 miter
```

(continues on next page)

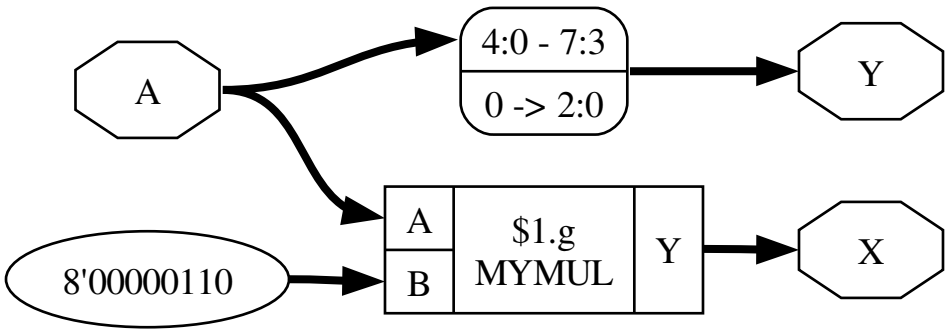
(continued from previous page)

```
splitnets -ports test_mapped/A
show -prefix mymul -format dot -notitle test_mapped
```

4.4.4 Handling constant inputs

- The special parameters `_TECHMAP_CONSTMSK_<port-name>_` and `_TECHMAP_CONSTVAL_<port-name>_` can be used to handle constant input values to cells.
- The former contains 1-bits for all constant input bits on the port.
- The latter contains the constant bits or undef (x) for non-constant bits.
- Example use-cases:
  - Converting arithmetic (for example multiply to shift).
  - Identify constant addresses or enable bits in memory interfaces.

Example:



Listing 4.41: `mulshift_map.v`

```
module MYMUL(A, B, Y);
    parameter WIDTH = 1;
    input [WIDTH-1:0] A, B;
    output reg [WIDTH-1:0] Y;

    parameter _TECHMAP_CONSTVAL_A_ = WIDTH'bx;
    parameter _TECHMAP_CONSTVAL_B_ = WIDTH'bx;

    reg _TECHMAP_FAIL_;
    wire [1023:0] _TECHMAP_DO_ = "proc; clean";

    integer i;
    always @* begin
            _TECHMAP_FAIL_ <= 1;
        for (i = 0; i < WIDTH; i=i+1) begin
```

(continues on next page)

(continued from previous page)

```
            if (_TECHMAP_CONSTVAL_A_ === WIDTH'd1 << i) begin
                _TECHMAP_FAIL_ <= 0;
                Y <= B << i;
            end
            if (_TECHMAP_CONSTVAL_B_ === WIDTH'd1 << i) begin
                _TECHMAP_FAIL_ <= 0;
                Y <= A << i;
            end
        end
    end
endmodule
```

Listing 4.42: `mulshift_test.v`

```
module test (A, X, Y);
input [7:0] A;
output [7:0] X = A * 8'd 6;
output [7:0] Y = A * 8'd 8;
endmodule
```

Listing 4.43: `mulshift_test.ys`

```
read_verilog mulshift_test.v
hierarchy -check -top test

techmap -map sym_mul_map.v \
        -map mulshift_map.v;;

show -prefix mulshift -format dot -notitle -lib sym_mul_cells.v
```

4.4.5 Handling shorted inputs

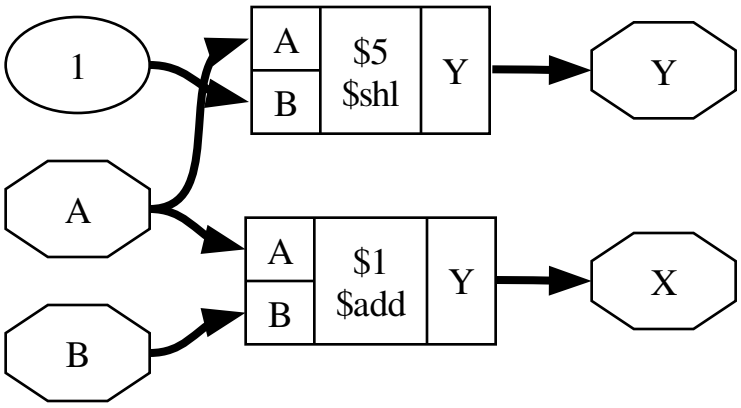
- The special parameters `_TECHMAP_BITS_CONNMAP_` and `_TECHMAP_CONNMAP_<port-name>_` can be used to handle shorted inputs.
- Each bit of the port correlates to an `_TECHMAP_BITS_CONNMAP_` bits wide number in `_TECHMAP_CONNMAP_<port-name>_`.
- Each unique signal bit is assigned its own number. Identical fields in the `_TECHMAP_CONNMAP_<port-name>_` parameters mean shorted signal bits.
- The numbers 0-3 are reserved for `0`, `1`, `x`, and `z` respectively.
- Example use-cases:
  - Detecting shared clock or control signals in memory interfaces.
  - In some cases this can be used for for optimization.

Example:

Listing 4.44: `addshift_map.v`

```
module \$add (A, B, Y);
  parameter A_SIGNED = 0;
```

(continues on next page)



(continued from previous page)

```
  parameter B_SIGNED = 0;
  parameter A_WIDTH = 1;
  parameter B_WIDTH = 1;
  parameter Y_WIDTH = 1;

  input [A_WIDTH-1:0] A;
  input [B_WIDTH-1:0] B;
  output [Y_WIDTH-1:0] Y;

  parameter _TECHMAP_BITS_CONNMAP_ = 0;
  parameter _TECHMAP_CONNMAP_A_ = 0;
  parameter _TECHMAP_CONNMAP_B_ = 0;

  wire _TECHMAP_FAIL_ = A_WIDTH != B_WIDTH || B_WIDTH < Y_WIDTH ||
                        _TECHMAP_CONNMAP_A_ != _TECHMAP_CONNMAP_B_;

  assign Y = A << 1;
endmodule
```

Listing 4.45: `addshift_test.v`

```
module test (A, B, X, Y);
input [7:0] A, B;
output [7:0] X = A + B;
output [7:0] Y = A + A;
endmodule
```

Listing 4.46: `addshift_test.ys`

```
read_verilog addshift_test.v
hierarchy -check -top test
```

(continues on next page)

(continued from previous page)

```
techmap -map addshift_map.v;;

show -prefix addshift -format dot -notitle
```

4.4.6 Notes on using techmap

- Don't use positional cell parameters in map modules.
- You can use the `$__`-prefix for internal cell types to avoid collisions with the user-namespace. But always use two underscores or the internal consistency checker will trigger on these cells.
- Techmap has two major use cases:
  - Creating good logic-level representation of arithmetic functions. This also means using dedicated hardware resources such as half- and full-adder cells in ASICS or dedicated carry logic in FPGAs.
  - Mapping of coarse-grain resources such as block memory or DSP cells.

4.5 Hashing and associative data structures in Yosys

4.5.1 Container classes based on hashing

Yosys uses `dict<K, T>` and `pool<T>` as main container classes. `dict<K, T>` is essentially a replacement for `std::unordered_map<K, T>` and `pool<T>` is a replacement for `std::unordered_set<T>`. The main characteristics are:

- **`dict<K, T>` and `pool<T>` are about 2x faster than the std containers**
  (though this claim hasn't been verified for over 10 years)
- **references to elements in a `dict<K, T>` or `pool<T>` are invalidated by**
  insert and remove operations (similar to `std::vector<T>` on `push_back()`).
- **some iterators are invalidated by `erase()`. specifically, iterators**
  that have not passed the erased element yet are invalidated. (`erase()` itself returns valid iterator to the next element.)
- **no iterators are invalidated by `insert()`. elements are inserted at**
  `begin()`. i.e. only a new iterator that starts at `begin()` will see the inserted elements.
- **the method `.count(key, iterator)` is like `.count(key)` but only**
  considers elements that can be reached via the iterator.
- **iterators can be compared. `it1 < it2` means that the position of `t2`**
  can be reached via `t1` but not vice versa.
- **the method `.sort()` can be used to sort the elements in the container**
  the container stays sorted until elements are added or removed.
- **`dict<K, T>` and `pool<T>` will have the same order of iteration across**
  all compilers, standard libraries and architectures.

In addition to `dict<K, T>` and `pool<T>` there is also an `idict<K>` that creates a bijective map from `K` to incrementing integers. For example:

```
idict<string, 42> si;
log("%d\n", si("hello"));      // will print 42
log("%d\n", si("world"));      // will print 43
```

(continues on next page)

(continued from previous page)

```
log("%d\n", si.at("world"));   // will print 43
log("%d\n", si.at("dummy"));   // will throw exception
log("%s\n", si[42]));  // will print hello
log("%s\n", si[43]));  // will print world
log("%s\n", si[44]));  // will throw exception
```

It is not possible to remove elements from an idict.

Finally `mfp<K>` implements a merge-find set data structure (aka. disjoint-set or union-find) over the type `K` ("mfp" = merge-find-promote).

4.5.2 The hash function

The hash function generally used in Yosys is the XOR version of DJB2:

```
state = ((state << 5) + state) ^ value
```

This is an old-school hash designed to hash ASCII characters. Yosys doesn't hash a lot of ASCII text, but it still happens to be a local optimum due to factors described later.

Hash function quality is multi-faceted and highly dependent on what is being hashed. Yosys isn't concerned by any cryptographic qualities, instead the goal is minimizing total hashing collision risk given the data patterns within Yosys. In general, a good hash function typically folds values into a state accumulator with a mathematical function that is fast to compute and has some beneficial properties. One of these is the avalanche property, which demands that a small change such as flipping a bit or incrementing by one in the input produces a large, unpredictable change in the output. Additionally, the bit independence criterion states that any pair of output bits should change independently when any single input bit is inverted. These properties are important for avoiding hash collision on data patterns like the hash of a sequence not colliding with its permutation, not losing from the state the information added by hashing preceding elements, etc.

DJB2 lacks these properties. Instead, since Yosys hashes large numbers of data structures composed of incrementing integer IDs, Yosys abuses the predictability of DJB2 to get lower hash collisions, with regular nature of the hashes surviving through the interaction with the "modulo prime" operations in the associative data structures. For example, some most common objects in Yosys are interned `IdString`s of incrementing indices or `SigBit`s with bit offsets into wire (represented by its unique `IdString` name) as the typical case. This is what makes DJB2 a local optimum. Additionally, the ADD version of DJB2 (like above but with addition instead of XOR) is used to this end for some types, abandoning the general pattern of folding values into a state value.

4.5.3 Making a type hashable

Let's first take a look at the external interface on a simplified level. Generally, to get the hash for `T obj`, you would call the utility function `run_hash<T>(const T& obj)`, corresponding to `hash_ops<T>::hash(obj)`, the default implementation of which uses `hash_ops<T>::hash_into(Hasher(), obj)`. `Hasher` is the class actually implementing the hash function, hiding its initialized internal state, and passing it out on `hash_t yield()` with perhaps some finalization steps.

`hash_ops<T>` is the star of the show. By default it pulls the `Hasher h` through a `Hasher T::hash_into(Hasher h)` method. That's the method you have to implement to make a record (class or struct) type easily hashable with Yosys hashlib associative data structures.

`hash_ops<T>` is specialized for built-in types like `int` or `bool` and treats pointers the same as integers, so it doesn't dereference pointers. Since many RTLIL data structures like `RTLIL::Wire` carry their own unique index `Hasher::hash_t hashidx_;`, there are specializations for `hash_ops<Wire*>` and others in `kernel/hashlib.h` that actually dereference the pointers and call `hash_into` on the instances pointed to.

`hash_ops<T>` is also specialized for simple compound types like `std::pair<U>` by calling hash_into in sequence on its members. For flexible size containers like `std::vector<U>` the size of the container is hashed first. That is also how implementing hashing for a custom record data type should be - unless there is strong reason to do otherwise, call `h.eat(m)` on the `Hasher h` you have received for each member in sequence and `return h;`.

The `hash_ops<T>::hash(obj)` method is not indended to be called when context of implementing the hashing for a record or other compound type. When writing it, you should connect it to `hash_ops<T>::hash_into(Hasher h)` as shown below. If you have a strong reason to do so, and you have to create a special implementation for top-level hashing, look at how `hash_ops<RTLIL::SigBit>::hash(...)` is implemented in `kernel/rtlil.h`.

4.5.4 Porting plugins from the legacy interface

Previously, the interface to implement hashing on custom types was just `unsigned int T::hash() const`. This meant hashes for members were computed independently and then ad-hoc combined with the hash function with some xorshift operations thrown in to mix bits together somewhat. A plugin can stay compatible with both versions prior and after the break by implementing both interfaces based on the existance and value of `YS_HASHING_VERSION`.

Listing 4.47: Example hash compatibility wrapper

```
#ifndef YS_HASHING_VERSION
unsigned int T::hash() const {
   return mkhash(a, b);
}
#elif YS_HASHING_VERSION == 1
Hasher T::hash_into(Hasher h) const {
   h.eat(a);
   h.eat(b);
   return h;
}
Hasher T::hash() const {
   Hasher h;
   h.eat(*this);
   return h;
}
#else
#error "Unsupported hashing interface"
#endif
```

Feel free to contact Yosys maintainers with related issues.

CHAPTER

FIVE

A PRIMER ON DIGITAL CIRCUIT SYNTHESIS

This chapter contains a short introduction to the basic principles of digital circuit synthesis.

5.1 Levels of abstraction

Digital circuits can be represented at different levels of abstraction. During the design process a circuit is usually first specified using a higher level abstraction. Implementation can then be understood as finding a functionally equivalent representation at a lower abstraction level. When this is done automatically using software, the term synthesis is used.

So synthesis is the automatic conversion of a high-level representation of a circuit to a functionally equivalent low-level representation of a circuit. Figure 5.1 lists the different levels of abstraction and how they relate to different kinds of synthesis.

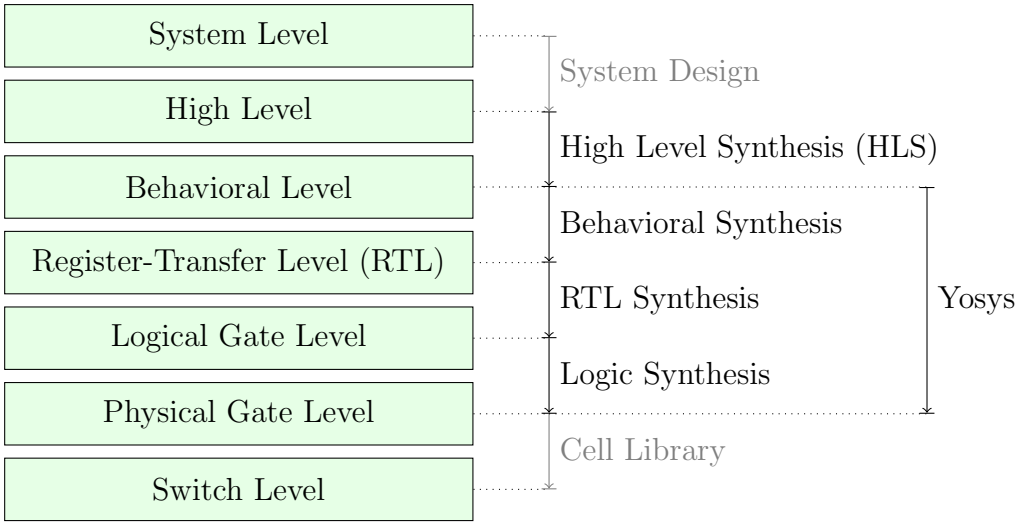


Fig. 5.1: Different levels of abstraction and synthesis.

Regardless of the way a lower level representation of a circuit is obtained (synthesis or manual design), the lower level representation is usually verified by comparing simulation results of the lower level and the higher level representation[1]. Therefore even if no synthesis is used, there must still be a simulatable representation of the circuit in all levels to allow for verification of the design.

Note: The exact meaning of terminology such as "High-Level" is of course not fixed over time. For example the HDL "ABEL" was first introduced in 1985 as "A High-Level Design Language for Programmable Logic

[1] In recent years formal equivalence checking also became an important verification method for validating RTL and lower abstraction representation of the design.

Devices" [LHBB85], but would not be considered a "High-Level Language" today.

5.1.1 System level

The System Level abstraction of a system only looks at its biggest building blocks like CPUs and computing cores. At this level the circuit is usually described using traditional programming languages like C/C++ or Matlab. Sometimes special software libraries are used that are aimed at simulation circuits on the system level, such as SystemC.

Usually no synthesis tools are used to automatically transform a system level representation of a circuit to a lower-level representation. But system level design tools exist that can be used to connect system level building blocks.

The IEEE 1685-2009 standard defines the IP-XACT file format that can be used to represent designs on the system level and building blocks that can be used in such system level designs. [A+10]

5.1.2 High level

The high-level abstraction of a system (sometimes referred to as algorithmic level) is also often represented using traditional programming languages, but with a reduced feature set. For example when representing a design at the high level abstraction in C, pointers can only be used to mimic concepts that can be found in hardware, such as memory interfaces. Full featured dynamic memory management is not allowed as it has no corresponding concept in digital circuits.

Tools exist to synthesize high level code (usually in the form of C/C++/SystemC code with additional metadata) to behavioural HDL code (usually in the form of Verilog or VHDL code). Aside from the many commercial tools for high level synthesis there are also a number of FOSS tools for high level synthesis.

5.1.3 Behavioural level

At the behavioural abstraction level a language aimed at hardware description such as Verilog or VHDL is used to describe the circuit, but so-called behavioural modelling is used in at least part of the circuit description. In behavioural modelling there must be a language feature that allows for imperative programming to be used to describe data paths and registers. This is the always-block in Verilog and the process-block in VHDL.

In behavioural modelling, code fragments are provided together with a sensitivity list; a list of signals and conditions. In simulation, the code fragment is executed whenever a signal in the sensitivity list changes its value or a condition in the sensitivity list is triggered. A synthesis tool must be able to transfer this representation into an appropriate datapath followed by the appropriate types of register.

For example consider the following Verilog code fragment:

```
1 always @(posedge clk)
2     y <= a + b;
```

In simulation the statement `y <= a + b` is executed whenever a positive edge on the signal `clk` is detected. The synthesis result however will contain an adder that calculates the sum `a + b` all the time, followed by a d-type flip-flop with the adder output on its D-input and the signal `y` on its Q-output.

Usually the imperative code fragments used in behavioural modelling can contain statements for conditional execution (`if`- and `case`-statements in Verilog) as well as loops, as long as those loops can be completely unrolled.

Interestingly there seems to be no other FOSS Tool that is capable of performing Verilog or VHDL behavioural syntheses besides Yosys.

5.1.4 Register-Transfer Level (RTL)

On the Register-Transfer Level the design is represented by combinatorial data paths and registers (usually d-type flip flops). The following Verilog code fragment is equivalent to the previous Verilog example, but is in RTL representation:

```
assign tmp = a + b;       // combinatorial data path

always @(posedge clk)     // register
    y <= tmp;
```

A design in RTL representation is usually stored using HDLs like Verilog and VHDL. But only a very limited subset of features is used, namely minimalistic always-blocks (Verilog) or process-blocks (VHDL) that model the register type used and unconditional assignments for the datapath logic. The use of HDLs on this level simplifies simulation as no additional tools are required to simulate a design in RTL representation.

Many optimizations and analyses can be performed best at the RTL level. Examples include FSM detection and optimization, identification of memories or other larger building blocks and identification of shareable resources.

Note that RTL is the first abstraction level in which the circuit is represented as a graph of circuit elements (registers and combinatorial cells) and signals. Such a graph, when encoded as list of cells and connections, is called a netlist.

RTL synthesis is easy as each circuit node element in the netlist can simply be replaced with an equivalent gate-level circuit. However, usually the term RTL synthesis does not only refer to synthesizing an RTL netlist to a gate level netlist but also to performing a number of highly sophisticated optimizations within the RTL representation, such as the examples listed above.

A number of FOSS tools exist that can perform isolated tasks within the domain of RTL synthesis steps. But there seems to be no FOSS tool that covers a wide range of RTL synthesis operations.

5.1.5 Logical gate level

At the logical gate level the design is represented by a netlist that uses only cells from a small number of single-bit cells, such as basic logic gates (AND, OR, NOT, XOR, etc.) and registers (usually D-Type Flip-flops).

A number of netlist formats exists that can be used on this level, e.g. the Electronic Design Interchange Format (EDIF), but for ease of simulation often a HDL netlist is used. The latter is a HDL file (Verilog or VHDL) that only uses the most basic language constructs for instantiation and connecting of cells.

There are two challenges in logic synthesis: First finding opportunities for optimizations within the gate level netlist and second the optimal (or at least good) mapping of the logic gate netlist to an equivalent netlist of physically available gate types.

The simplest approach to logic synthesis is two-level logic synthesis, where a logic function is converted into a sum-of-products representation, e.g. using a Karnaugh map. This is a simple approach, but has exponential worst-case effort and cannot make efficient use of physical gates other than AND/NAND-, OR/NOR- and NOT-Gates.

Therefore modern logic synthesis tools utilize much more complicated multi-level logic synthesis algorithms [BHSV90]. Most of these algorithms convert the logic function to a Binary-Decision-Diagram (BDD) or And-Inverter-Graph (AIG) and work from that representation. The former has the advantage that it has a unique normalized form. The latter has much better worst case performance and is therefore better suited for the synthesis of large logic functions.

Good FOSS tools exists for multi-level logic synthesis.

Yosys contains basic logic synthesis functionality but can also use ABC for the logic synthesis step. Using ABC is recommended.

5.1.6 Physical gate level

On the physical gate level only gates are used that are physically available on the target architecture. In some cases this may only be NAND, NOR and NOT gates as well as D-Type registers. In other cases this might include cells that are more complex than the cells used at the logical gate level (e.g. complete half-adders). In the case of an FPGA-based design the physical gate level representation is a netlist of LUTs with optional output registers, as these are the basic building blocks of FPGA logic cells.

For the synthesis tool chain this abstraction is usually the lowest level. In case of an ASIC-based design the cell library might contain further information on how the physical cells map to individual switches (transistors).

5.1.7 Switch level

A switch level representation of a circuit is a netlist utilizing single transistors as cells. Switch level modelling is possible in Verilog and VHDL, but is seldom used in modern designs, as in modern digital ASIC or FPGA flows the physical gates are considered the atomic build blocks of the logic circuit.

5.1.8 Yosys

Yosys is a Verilog HDL synthesis tool. This means that it takes a behavioural design description as input and generates an RTL, logical gate or physical gate level description of the design as output. Yosys' main strengths are behavioural and RTL synthesis. A wide range of commands (synthesis passes) exist within Yosys that can be used to perform a wide range of synthesis tasks within the domain of behavioural, RTL and logic synthesis. Yosys is designed to be extensible and therefore is a good basis for implementing custom synthesis tools for specialised tasks.

5.2 Features of synthesizable Verilog

The subset of Verilog [A+06] that is synthesizable is specified in a separate IEEE standards document, the IEEE standard 1364.1-2002 [A+02]. This standard also describes how certain language constructs are to be interpreted in the scope of synthesis.

This section provides a quick overview of the most important features of synthesizable Verilog, structured in order of increasing complexity.

5.2.1 Structural Verilog

Structural Verilog (also known as Verilog Netlists) is a Netlist in Verilog syntax. Only the following language constructs are used in this case:

- Constant values
- Wire and port declarations
- Static assignments of signals to other signals
- Cell instantiations

Many tools (especially at the back end of the synthesis chain) only support structural Verilog as input. ABC is an example of such a tool. Unfortunately there is no standard specifying what Structural Verilog actually is, leading to some confusion about what syntax constructs are supported in structural Verilog when it comes to features such as attributes or multi-bit signals.

5.2.2 Expressions in Verilog

In all situations where Verilog accepts a constant value or signal name, expressions using arithmetic operations such as `+`, `-` and `*`, boolean operations such as `&` (AND), `|` (OR) and `^` (XOR) and many others (comparison operations, unary operator, etc.) can also be used.

During synthesis these operators are replaced by cells that implement the respective function.

Many FOSS tools that claim to be able to process Verilog in fact only support basic structural Verilog and simple expressions. Yosys can be used to convert full featured synthesizable Verilog to this simpler subset, thus enabling such applications to be used with a richer set of Verilog features.

5.2.3 Behavioural modelling

Code that utilizes the Verilog always statement is using Behavioural Modelling. In behavioural modelling, a circuit is described by means of imperative program code that is executed on certain events, namely any change, a rising edge, or a falling edge of a signal. This is a very flexible construct during simulation but is only synthesizable when one of the following is modelled:

- **Asynchronous or latched logic**
  In this case the sensitivity list must contain all expressions that are used within the always block. The syntax `@*` can be used for these cases. Examples of this kind include:

```
// asynchronous
always @* begin
    if (add_mode)
        y <= a + b;
    else
        y <= a - b;
end

// latched
always @* begin
    if (!hold)
        y <= a + b;
end
```

  Note that latched logic is often considered bad style and in many cases just the result of sloppy HDL design. Therefore many synthesis tools generate warnings whenever latched logic is generated.

- **Synchronous logic (with optional synchronous reset)**
  This is logic with d-type flip-flops on the output. In this case the sensitivity list must only contain the respective clock edge. Example:

```
// counter with synchronous reset
always @(posedge clk) begin
    if (reset)
        y <= 0;
    else
        y <= y + 1;
end
```

- **Synchronous logic with asynchronous reset**
  This is logic with d-type flip-flops with asynchronous resets on the output. In this case the sensitivity list must only contain the respective clock and reset edges. The values assigned in the reset branch must be constant. Example:

```
// counter with asynchronous reset
always @(posedge clk, posedge reset) begin
    if (reset)
        y <= 0;
    else
        y <= y + 1;
end
```

Many synthesis tools support a wider subset of flip-flops that can be modelled using always-statements (including Yosys). But only the ones listed above are covered by the Verilog synthesis standard and when writing new designs one should limit herself or himself to these cases.

In behavioural modelling, blocking assignments (=) and non-blocking assignments (<=) can be used. The concept of blocking vs. non-blocking assignment is one of the most misunderstood constructs in Verilog [CI00].

The blocking assignment behaves exactly like an assignment in any imperative programming language, while with the non-blocking assignment the right hand side of the assignment is evaluated immediately but the actual update of the left hand side register is delayed until the end of the time-step. For example the Verilog code `a <= b; b <= a;` exchanges the values of the two registers.

5.2.4 Functions and tasks

Verilog supports Functions and Tasks to bundle statements that are used in multiple places (similar to Procedures in imperative programming). Both constructs can be implemented easily by substituting the function/task-call with the body of the function or task.

5.2.5 Conditionals, loops and generate-statements

Verilog supports `if-else`-statements and `for`-loops inside `always`-statements.

It also supports both features in `generate`-statements on the module level. This can be used to selectively enable or disable parts of the module based on the module parameters (`if-else`) or to generate a set of similar subcircuits (`for`).

While the `if-else`-statement inside an always-block is part of behavioural modelling, the three other cases are (at least for a synthesis tool) part of a built-in macro processor. Therefore it must be possible for the synthesis tool to completely unroll all loops and evaluate the condition in all `if-else`-statement in `generate`-statements using const-folding..

5.2.6 Arrays and memories

Verilog supports arrays. This is in general a synthesizable language feature. In most cases arrays can be synthesized by generating addressable memories. However, when complex or asynchronous access patterns are used, it is not possible to model an array as memory. In these cases the array must be modelled using individual signals for each word and all accesses to the array must be implemented using large multiplexers.

In some cases it would be possible to model an array using memories, but it is not desired. Consider the following delay circuit:

```
module (clk, in_data, out_data);

parameter BITS = 8;
parameter STAGES = 4;

input clk;
```

(continues on next page)

(continued from previous page)

```
input [BITS-1:0] in_data;
output [BITS-1:0] out_data;
reg [BITS-1:0] ffs [STAGES-1:0];

integer i;
always @(posedge clk) begin
    ffs[0] <= in_data;
    for (i = 1; i < STAGES; i = i+1)
        ffs[i] <= ffs[i-1];
end

assign out_data = ffs[STAGES-1];

endmodule
```

This could be implemented using an addressable memory with STAGES input and output ports. A better implementation would be to use a simple chain of flip-flops (a so-called shift register). This better implementation can either be obtained by first creating a memory-based implementation and then optimizing it based on the static address signals for all ports or directly identifying such situations in the language front end and converting all memory accesses to direct accesses to the correct signals.

5.3 Challenges in digital circuit synthesis

This section summarizes the most important challenges in digital circuit synthesis. Tools can be characterized by how well they address these topics.

5.3.1 Standards compliance

The most important challenge is compliance with the HDL standards in question (in case of Verilog the IEEE Standards 1364.1-2002 and 1364-2005). This can be broken down in two items:

- Completeness of implementation of the standard
- Correctness of implementation of the standard

Completeness is mostly important to guarantee compatibility with existing HDL code. Once a design has been verified and tested, HDL designers are very reluctant regarding changes to the design, even if it is only about a few minor changes to work around a missing feature in a new synthesis tool.

Correctness is crucial. In some areas this is obvious (such as correct synthesis of basic behavioural models). But it is also crucial for the areas that concern minor details of the standard, such as the exact rules for handling signed expressions, even when the HDL code does not target different synthesis tools. This is because (unlike software source code that is only processed by compilers), in most design flows HDL code is not only processed by the synthesis tool but also by one or more simulators and sometimes even a formal verification tool. It is key for this verification process that all these tools use the same interpretation for the HDL code.

5.3.2 Optimizations

Generally it is hard to give a one-dimensional description of how well a synthesis tool optimizes the design. First of all because not all optimizations are applicable to all designs and all synthesis tasks. Some optimizations work (best) on a coarse-grained level (with complex cells such as adders or multipliers) and others work (best) on a fine-grained level (single bit gates). Some optimizations target area and others target speed. Some work well on large designs while others don't scale well and can only be applied to small designs.

A good tool is capable of applying a wide range of optimizations at different levels of abstraction and gives the designer control over which optimizations are performed (or skipped) and what the optimization goals are.

5.3.3 Technology mapping

Technology mapping is the process of converting the design into a netlist of cells that are available in the target architecture. In an ASIC flow this might be the process-specific cell library provided by the fab. In an FPGA flow this might be LUT cells as well as special function units such as dedicated multipliers. In a coarse-grain flow this might even be more complex special function units.

An open and vendor independent tool is especially of interest if it supports a wide range of different types of target architectures.

5.4 Script-based synthesis flows

A digital design is usually started by implementing a high-level or system-level simulation of the desired function. This description is then manually transformed (or re-implemented) into a synthesizable lower-level description (usually at the behavioural level) and the equivalence of the two representations is verified by simulating both and comparing the simulation results.

Then the synthesizable description is transformed to lower-level representations using a series of tools and the results are again verified using simulation. This process is illustrated in Fig. 5.2.

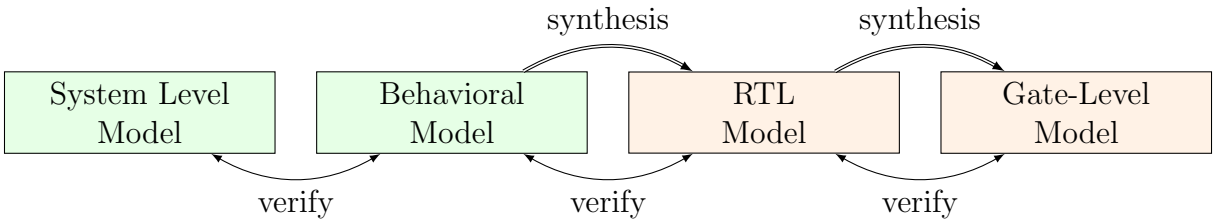


Fig. 5.2: Typical design flow. Green boxes represent manually created models. Orange boxes represent models generated by synthesis tools.

In this example the System Level Model and the Behavioural Model are both manually written design files. After the equivalence of system level model and behavioural model has been verified, the lower level representations of the design can be generated using synthesis tools. Finally the RTL Model and the Gate-Level Model are verified and the design process is finished.

However, in any real-world design effort there will be multiple iterations for this design process. The reason for this can be the late change of a design requirement or the fact that the analysis of a low-abstraction model (e.g. gate-level timing analysis) revealed that a design change is required in order to meet the design requirements (e.g. maximum possible clock speed).

Whenever the behavioural model or the system level model is changed their equivalence must be re-verified by re-running the simulations and comparing the results. Whenever the behavioural model is changed the synthesis must be re-run and the synthesis results must be re-verified.

In order to guarantee reproducibility it is important to be able to re-run all automatic steps in a design project with a fixed set of settings easily. Because of this, usually all programs used in a synthesis flow can be controlled using scripts. This means that all functions are available via text commands. When such a tool provides a GUI, this is complementary to, and not instead of, a command line interface.

Usually a synthesis flow in an UNIX/Linux environment would be controlled by a shell script that calls all required tools (synthesis and simulation/verification in this example) in the correct order. Each of these

tools would be called with a script file containing commands for the respective tool. All settings required for the tool would be provided by these script files so that no manual interaction would be necessary. These script files are considered design sources and should be kept under version control just like the source code of the system level and the behavioural model.

5.5 Methods from compiler design

Some parts of synthesis tools involve problem domains that are traditionally known from compiler design. This section addresses some of these domains.

5.5.1 Lexing and parsing

The best known concepts from compiler design are probably lexing and parsing. These are two methods that together can be used to process complex computer languages easily. [ASU86]

A lexer consumes single characters from the input and generates a stream of lexical tokens that consist of a type and a value. For example the Verilog input `assign foo = bar + 42;` might be translated by the lexer to the list of lexical tokens given in Tab. 5.1.

Table 5.1: Exemplary token list for the statement `assign foo = bar + 42;`

Token-Type	Token-Value
TOK_ASSIGN	-
TOK_IDENTIFIER	“foo”
TOK_EQ	-
TOK_IDENTIFIER	“bar”
TOK_PLUS	-
TOK_NUMBER	42
TOK_SEMICOLON	-

The lexer is usually generated by a lexer generator (e.g. flex) from a description file that is using regular expressions to specify the text pattern that should match the individual tokens.

The lexer is also responsible for skipping ignored characters (such as whitespace outside string constants and comments in the case of Verilog) and converting the original text snippet to a token value.

Note that individual keywords use different token types (instead of a keyword type with different token values). This is because the parser usually can only use the Token-Type to make a decision on the grammatical role of a token.

The parser then transforms the list of tokens into a parse tree that closely resembles the productions from the computer languages grammar. As the lexer, the parser is also typically generated by a code generator (e.g. bison) from a grammar description in Backus-Naur Form (BNF).

Let’s consider the following BNF (in Bison syntax):

```
1 assign_stmt: TOK_ASSIGN TOK_IDENTIFIER TOK_EQ expr TOK_SEMICOLON;
2 expr: TOK_IDENTIFIER | TOK_NUMBER | expr TOK_PLUS expr;
```

The parser converts the token list to the parse tree in Fig. 5.3. Note that the parse tree never actually exists as a whole as data structure in memory. Instead the parser calls user-specified code snippets (so-called reduce-functions) for all inner nodes of the parse tree in depth-first order.

In some very simple applications (e.g. code generation for stack machines) it is possible to perform the task at hand directly in the reduce functions. But usually the reduce functions are only used to build an

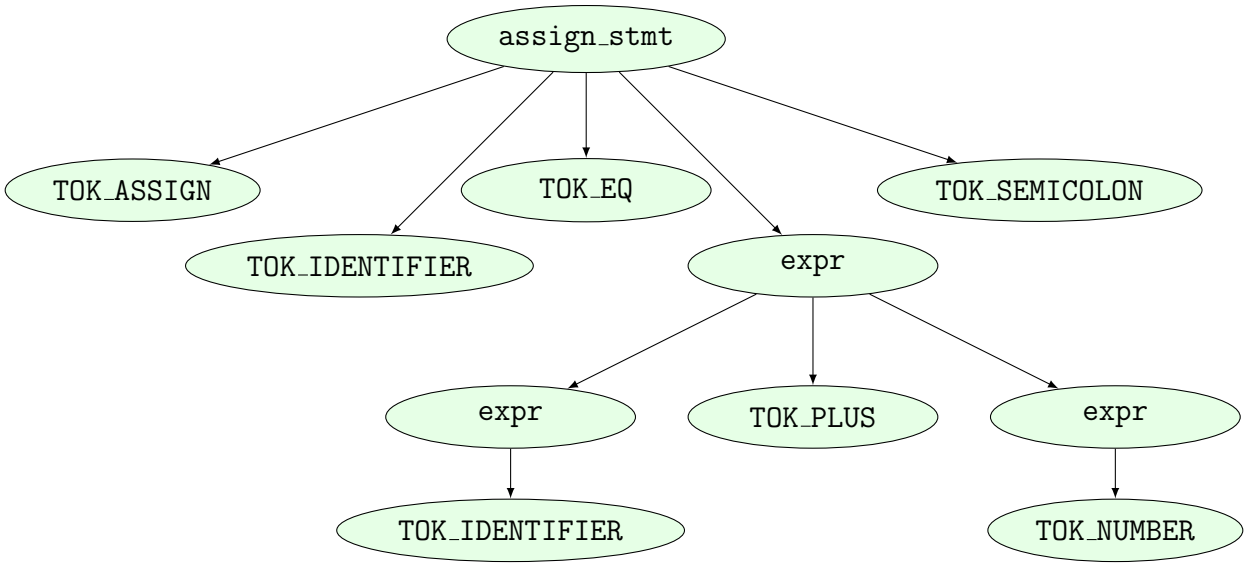


Fig. 5.3: Example parse tree for the Verilog expression `assign foo = bar + 42;`

in-memory data structure with the relevant information from the parse tree. This data structure is called an abstract syntax tree (AST).

The exact format for the abstract syntax tree is application specific (while the format of the parse tree and token list are mostly dictated by the grammar of the language at hand). Figure 5.4 illustrates what an AST for the parse tree in Fig. 5.3 could look like.

Usually the AST is then converted into yet another representation that is more suitable for further processing. In compilers this is often an assembler-like three-address-code intermediate representation. [ASU86]

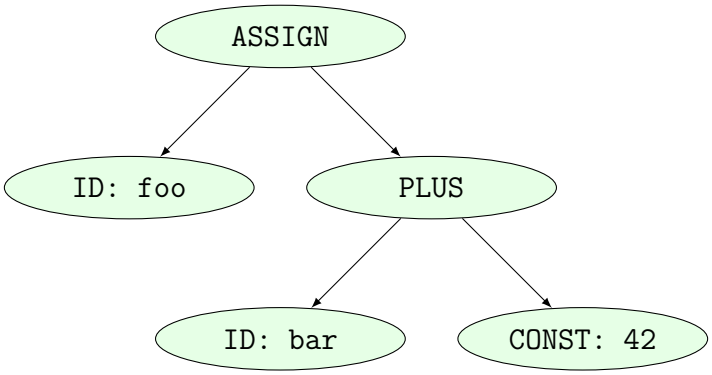


Fig. 5.4: Example abstract syntax tree for the Verilog expression `assign foo = bar + 42;`

5.5.2 Multi-pass compilation

Complex problems are often best solved when split up into smaller problems. This is certainly true for compilers as well as for synthesis tools. The components responsible for solving the smaller problems can be connected in two different ways: through Single-Pass Pipelining and by using Multiple Passes.

Traditionally a parser and lexer are connected using the pipelined approach: The lexer provides a function that is called by the parser. This function reads data from the input until a complete lexical token has been read. Then this token is returned to the parser. So the lexer does not first generate a complete list of lexical

tokens and then pass it to the parser. Instead they run concurrently and the parser can consume tokens as the lexer produces them.

The single-pass pipelining approach has the advantage of lower memory footprint (at no time must the complete design be kept in memory) but has the disadvantage of tighter coupling between the interacting components.

Therefore single-pass pipelining should only be used when the lower memory footprint is required or the components are also conceptually tightly coupled. The latter certainly is the case for a parser and its lexer. But when data is passed between two conceptually loosely coupled components it is often beneficial to use a multi-pass approach.

In the multi-pass approach the first component processes all the data and the result is stored in a in-memory data structure. Then the second component is called with this data. This reduces complexity, as only one component is running at a time. It also improves flexibility as components can be exchanged easier.

Most modern compilers are multi-pass compilers.

5.5.3 Static Single Assignment (SSA) form

In imperative programming (and behavioural HDL design) it is possible to assign the same variable multiple times. This can either mean that the variable is independently used in two different contexts or that the final value of the variable depends on a condition.

The following examples show C code in which one variable is used independently in two different contexts:

```
void demo1()
{
    int a = 1;
    printf("%d\n", a);

    a = 2;
    printf("%d\n", a);
}
```

```
void demo1()
{
    int a = 1;
    printf("%d\n", a);

    int b = 2;
    printf("%d\n", b);
}
```

```
void demo2(bool foo)
{
    int a;
    if (foo) {
        a = 23;
        printf("%d\n", a);
    } else {
        a = 42;
        printf("%d\n", a);
    }
}
```

```
void demo2(bool foo)
{
    int a, b;
    if (foo) {
        a = 23;
        printf("%d\n", a);
    } else {
        b = 42;
        printf("%d\n", b);
    }
}
```

In both examples the left version (only variable `a`) and the right version (variables `a` and `b`) are equivalent. Therefore it is desired for further processing to bring the code in an equivalent form for both cases.

In the following example the variable is assigned twice but it cannot be easily replaced by two variables:

```
void demo3(bool foo)
{
    int a = 23
    if (foo)
        a = 42;
    printf("%d\n", a);
}
```

Static single assignment (SSA) form is a representation of imperative code that uses identical representations for the left and right version of demos 1 and 2, but can still represent demo 3. In SSA form each assignment assigns a new variable (usually written with an index). But it also introduces a special Φ-function to merge the different instances of a variable when needed. In C-pseudo-code the demo 3 would be written as follows using SSA from:

```
void demo3(bool foo)
{
    int a_1, a_2, a_3;
    a_1 = 23
    if (foo)
        a_2 = 42;
    a_3 = phi(a_1, a_2);
    printf("%d\n", a_3);
}
```

The Φ-function is usually interpreted as "these variables must be stored in the same memory location" during code generation. Most modern compilers for imperative languages such as C/C++ use SSA form for at least some of its passes as it is very easy to manipulate and analyse.

CHAPTER

SIX

RTLIL TEXT REPRESENTATION

This appendix documents the text representation of RTLIL in extended Backus-Naur form (EBNF).

The grammar is not meant to represent semantic limitations. That is, the grammar is "permissive", and later stages of processing perform more rigorous checks.

The grammar is also not meant to represent the exact grammar used in the RTLIL frontend, since that grammar is specific to processing by lex and yacc, is even more permissive, and is somewhat less understandable than simple EBNF notation.

Finally, note that all statements (rules ending in `-stmt`) terminate in an end-of-line. Because of this, a statement cannot be broken into multiple lines.

6.1 Lexical elements

6.1.1 Characters

An RTLIL file is a stream of bytes. Strictly speaking, a "character" in an RTLIL file is a single byte. The lexer treats multi-byte encoded characters as consecutive single-byte characters. While other encodings *may* work, UTF-8 is known to be safe to use. Byte order marks at the beginning of the file will cause an error.

ASCII spaces (32) and tabs (9) separate lexer tokens.

A `nonws` character, used in identifiers, is any character whose encoding consists solely of bytes above ASCII space (32).

An `eol` is one or more consecutive ASCII newlines (10) and carriage returns (13).

6.1.2 Identifiers

There are two types of identifiers in RTLIL:

- Publically visible identifiers
- Auto-generated identifiers

```
<id>            ::= <public-id> | <autogen-id>
<public-id>     ::= \ <nonws>+
<autogen-id>    ::= $ <nonws>+
```

6.1.3 Values

A *value* consists of a width in bits and a bit representation, most significant bit first. Bits may be any of:

- `0`: A logic zero value
- `1`: A logic one value

- `x`: An unknown logic value (or don't care in case patterns)
- `z`: A high-impedance value (or don't care in case patterns)
- `m`: A marked bit (internal use only)
- `-`: A don't care value

When the bit representation has fewer bits than the width, it is padded to the width with the most significant explicit bit, or `0` if the most significant explicit bit is `1`, or `x` if there are no explicit bits.

An *integer* is simply a signed integer value in decimal format. **Warning:** Integer constants are limited to 32 bits. That is, they may only be in the range $[-2147483648, 2147483648)$. Integers outside this range will result in an error.

```
<value>         ::= <decimal-digit>+ ' <binary-digit>*
<decimal-digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<binary-digit>  ::= 0 | 1 | x | z | m | -
<integer>       ::= -? <decimal-digit>+
```

6.1.4 Strings

A string is a series of characters delimited by double-quote characters. Within a string, any character except ASCII NUL (0) may be used. In addition, certain escapes can be used:

- `\n`: A newline
- `\t`: A tab
- `\ooo`: A character specified as a one, two, or three digit octal value

All other characters may be escaped by a backslash, and become the following character. Thus:

- `\\`: A backslash
- `\"`: A double-quote
- `\r`: An 'r' character

6.1.5 Comments

A comment starts with a `#` character and proceeds to the end of the line. All comments are ignored.

6.2 File

A file consists of an optional autoindex statement followed by zero or more modules.

```
<file> ::= <autoidx-stmt>? <module>*
```

6.2.1 Autoindex statements

The autoindex statement sets the global autoindex value used by Yosys when it needs to generate a unique name, e.g. `flattenN`. The N part is filled with the value of the global autoindex value, which is subsequently incremented. This global has to be dumped into RTLIL, otherwise e.g. dumping and running a pass would have different properties than just running a pass on a warm design.

```
<autoidx-stmt> ::= autoidx <integer> <eol>
```

6.2.2 Modules

Declares a module, with zero or more attributes, consisting of zero or more wires, memories, cells, processes, and connections.

```
<module>            ::= <attr-stmt>* <module-stmt> <module-body> <module-end-stmt>
<module-stmt>       ::= module <id> <eol>
<module-body>       ::= (<param-stmt>
                      |  <conn-stmt>
                      |  <wire>
                      |  <memory>
                      |  <cell>
                      |  <process>)*
<param-stmt>        ::= parameter <id> <constant>? <eol>
<constant>          ::= <value> | <integer> | <string>
<module-end-stmt>   ::= end <eol>
```

6.2.3 Attribute statements

Declares an attribute with the given identifier and value.

```
<attr-stmt> ::= attribute <id> <constant> <eol>
```

6.2.4 Signal specifications

A signal is anything that can be applied to a cell port, i.e. a constant value, all bits or a selection of bits from a wire, or concatenations of those.

Warning: When an integer constant is a sigspec, it is always 32 bits wide, 2's complement. For example, a constant of -1 is the same as `32'11111111111111111111111111111111`, while a constant of 1 is the same as `32'1`.

See *RTLIL::SigSpec* for an overview of signal specifications.

```
<sigspec> ::= <constant>
            |  <wire-id>
            |  <sigspec> [ <integer> (:<integer>)? ]
            |  { <sigspec>* }
```

When a `<wire-id>` is specified, the wire must have been previously declared.

When a signal slice is specified, the left-hand integer must be greather than or equal to the right-hand integer.

6.2.5 Connections

Declares a connection between the given signals.

```
<conn-stmt> ::= connect <sigspec> <sigspec> <eol>
```

6.2.6 Wires

Declares a wire, with zero or more attributes, with the given identifier and options in the enclosing module.

See *RTLIL::Cell and RTLIL::Wire* for an overview of wires.

```
<wire>          ::= <attr-stmt>* <wire-stmt>
<wire-stmt>     ::= wire <wire-option>* <wire-id> <eol>
<wire-id>       ::= <id>
<wire-option>   ::= width <integer>
                  | offset <integer>
                  | input <integer>
                  | output <integer>
                  | inout <integer>
                  | upto
                  | signed
```

6.2.7 Memories

Declares a memory, with zero or more attributes, with the given identifier and options in the enclosing module.

See *RTLIL::Memory* for an overview of memory cells, and *Memories* for details about memory cell types.

```
<memory>        ::= <attr-stmt>* <memory-stmt>
<memory-stmt>   ::= memory <memory-option>* <id> <eol>
<memory-option> ::= width <integer>
                  | size <integer>
                  | offset <integer>
```

6.2.8 Cells

Declares a cell, with zero or more attributes, with the given identifier and type in the enclosing module.

Cells perform functions on input signals. See *Internal cell library* for a detailed list of cell types.

```
<cell>              ::= <attr-stmt>* <cell-stmt> <cell-body-stmt>* <cell-end-stmt>
<cell-stmt>         ::= cell <cell-type> <cell-id> <eol>
<cell-id>           ::= <id>
<cell-type>         ::= <id>
<cell-body-stmt>    ::= parameter (signed | real)? <id> <constant> <eol>
                      | connect <id> <sigspec> <eol>
<cell-end-stmt>     ::= end <eol>
```

6.2.9 Processes

Declares a process, with zero or more attributes, with the given identifier in the enclosing module. The body of a process consists of zero or more assignments followed by zero or more switches and zero or more syncs.

See *RTLIL::Process* for an overview of processes.

```
<process>       ::= <attr-stmt>* <proc-stmt> <process-body> <proc-end-stmt>
<proc-stmt>     ::= process <id> <eol>
<process-body>  ::= <assign-stmt>* <switch>* <sync>*
<assign-stmt>   ::= assign <dest-sigspec> <src-sigspec> <eol>
<dest-sigspec>  ::= <sigspec>
<src-sigspec>   ::= <sigspec>
<proc-end-stmt> ::= end <eol>
```

6.2.10 Switches

Switches test a signal for equality against a list of cases. Each case specifies a comma-separated list of signals to check against. If there are no signals in the list, then the case is the default case. The body of a case consists of zero or more assignments followed by zero or more switches. Both switches and cases may have zero or more attributes.

```
<switch>            ::= <switch-stmt> <case>* <switch-end-stmt>
<switch-stmt>       ::= <attr-stmt>* switch <sigspec> <eol>
<case>              ::= <attr-stmt>* <case-stmt> <case-body>
<case-stmt>         ::= case <compare>? <eol>
<compare>           ::= <sigspec> (, <sigspec>)*
<case-body>         ::= <assign-stmt>* <switch>*
<switch-end-stmt>   ::= end <eol>
```

6.2.11 Syncs

Syncs update signals with other signals when an event happens. Such an event may be:

- An edge or level on a signal
- Global clock ticks
- Initialization
- Always

```
<sync>          ::= <sync-stmt> <update-stmt>*
<sync-stmt>     ::= sync <sync-type> <sigspec> <eol>
                  |  sync global <eol>
                  |  sync init <eol>
                  |  sync always <eol>
<sync-type>     ::= low | high | posedge | negedge | edge
<update-stmt>   ::= update <dest-sigspec> <src-sigspec> <eol>
                  |  <attr-stmt>* memwr <id> <sigspec> <sigspec> <sigspec> <constant>
↪<eol>
```

CHAPTER

SEVEN

AUXILIARY LIBRARIES

The Yosys source distribution contains some auxiliary libraries that are compiled into Yosys and can be used in plugins.

7.1 BigInt

The files in `libs/bigint/` provide a library for performing arithmetic with arbitrary length integers. It is written by Matt McCutchen.

The BigInt library is used for evaluating constant expressions, e.g. using the ConstEval class provided in kernel/consteval.h.

See also: http://mattmccutchen.net/bigint/

7.2 dlfcn-win32

The `dlfcn` library enables runtime loading of plugins without requiring recompilation of Yosys. The files in `libs/dlfcn-win32` provide an implementation of `dlfcn` for Windows.

See also: https://github.com/dlfcn-win32/dlfcn-win32

7.3 ezSAT

The files in `libs/ezsat` provide a library for simplifying generating CNF formulas for SAT solvers. It also contains bindings of MiniSAT. The ezSAT library is written by C. Wolf. It is used by the *sat* pass.

7.4 fst

`libfst` files from gtkwave are included in `libs/fst` to support reading/writing signal traces from/to the GTKWave developed FST format. This is primarily used in the `sim` command.

7.5 json11

For reading/writing designs from/to JSON, *read_json* and *write_json* should be used. For everything else there is the json11 library:

> json11 is a tiny JSON library for C++11, providing JSON parsing and serialization.

This library is used for outputting machine-readable statistics (*stat* with `-json` flag), using the RPC frontend (`connect_rpc`), and the yosys-witness `yw` format.

7.6 MiniSAT

The files in `libs/minisat` provide a high-performance SAT solver, used by the *sat* command.

7.7 SHA1

The files in `libs/sha1/` provide a public domain SHA1 implementation written by Steve Reid, Bruce Guenter, and Volker Grabsch. It is used for generating unique names when specializing parameterized modules.

7.8 SubCircuit

The files in `libs/subcircuit` provide a library for solving the subcircuit isomorphism problem. It is written by C. Wolf and based on the Ullmann Subgraph Isomorphism Algorithm [Ull76]. It is used by the `extract` pass.

CHAPTER

EIGHT

AUXILIARY PROGRAMS

Besides the main yosys executable, the Yosys distribution contains a set of additional helper programs.

8.1 yosys-config

The `yosys-config` tool (an auto-generated shell-script) can be used to query compiler options and other information needed for building loadable modules for Yosys. See *Writing extensions* for details.

```
Usage: ./yosys-config [--exec] [--prefix pf] args..
       ./yosys-config --build modname.so cppsources..

Replacement args:
    --cxx         g++
    --cxxflags    -O2 -flto=auto -ffat-lto-objects -fexceptions -g \
                      -grecord-gcc-switches -pipe -Wall -Werror=format-security \
                      -Wp,-U_FORTIFY_SOURCE,-D_FORTIFY_SOURCE=3 \
                      -Wp,-D_GLIBCXX_ASSERTIONS \
                      -specs=/usr/lib/rpm/redhat/redhat-hardened-cc1 \
                      -fstack-protector-strong \
                      -specs=/usr/lib/rpm/redhat/redhat-annobin-cc1 -m64 \
                      -march=x86-64 -mtune=generic -fasynchronous-unwind-tables \
                      -fstack-clash-protection -fcf-protection -mtls-dialect=gnu2 \
                      -fno-omit-frame-pointer -mno-omit-leaf-frame-pointer -Wall \
                      -Wextra -Werror=unused -ggdb -I"/usr/include/yosys" -MD \
                      -MP -D_YOSYS_ -fPIC -I/usr/include -DYOSYS_VER=\"0.66\" \
                      -DYOSYS_MAJOR=0 -DYOSYS_MINOR=66 -DYOSYS_COMMIT=0.66 \
                      -std=c++20 -O3 -DYOSYS_ENABLE_READLINE \
                      -DYOSYS_ENABLE_PLUGINS -DYOSYS_ENABLE_GLOB \
                      -DYOSYS_ENABLE_ZLIB -DTCL_WITH_EXTERNAL_TOMMATH \
                      -DWITH_GZFILEOP -DYOSYS_ENABLE_TCL -DYOSYS_ENABLE_THREADS \
                      -DYOSYS_ENABLE_ABC
    --linkflags   -rdynamic
    --ldflags     (alias of --linkflags)
    --libs        -lstdc++ -lm -lrt -lreadline -lffi -ldl -lz -ltcl9.0 -ltclstub -
↪ltommath -lpthread
    --ldlibs      (alias of --libs)
    --bindir      /usr/bin
    --datdir      /usr/share/yosys

All other args are passed through as they are.
```

(continues on next page)

(continued from previous page)

```
Use --exec to call a command instead of generating output. Example usage:

  ./yosys-config --exec --cxx --cxxflags --ldflags -o plugin.so -shared plugin.cc --libs

The above command can be abbreviated as:

  ./yosys-config --build plugin.so plugin.cc

Use --prefix to change the prefix for the special args from '--' to
something else. Example:

  ./yosys-config --prefix @ bindir: @bindir

The args --bindir and --datdir can be directly followed by a slash and
additional text. Example:

  ./yosys-config --datdir/simlib.v
```

8.2 yosys-filterlib

Todo

how does a filterlib rules-file work?

The `yosys-filterlib` tool is a small utility that can be used to strip or extract information from a Liberty file. This can be useful for removing sensitive or proprietary information such as timing or other trade secrets.

```
Usage: filterlib [rules-file [liberty-file]]
   or: filterlib -verilogsim [liberty-file]
```

8.3 yosys-abc

This is a fork of ABC with a small set of custom modifications that have not yet been accepted upstream. Not all versions of Yosys work with all versions of ABC. So Yosys comes with its own yosys-abc to avoid compatibility issues between the two.

8.4 yosys-smtbmc

The `yosys-smtbmc` tool is a utility used by SBY for interacting with smt solvers.

```
yosys-smtbmc [options] <yosys_smt2_output>

    -h, --help
        show this message

    -t <num_steps>
```

(continues on next page)

(continued from previous page)

```
    -t <skip_steps>:<num_steps>
    -t <skip_steps>:<step_size>:<num_steps>
        default: skip_steps=0, step_size=1, num_steps=20

    -g
        generate an arbitrary trace that satisfies
        all assertions and assumptions.

    -i
        instead of BMC run temporal induction

    -c
        instead of regular BMC run cover analysis

    -m <module_name>
        name of the top module

    --smtc <constr_filename>
        read constraints file

    --cex <cex_filename>
        read cex file as written by ABC's "write_cex -n"

    --aig <prefix>
        read AIGER map file (as written by Yosys' "write_aiger -map")
        and AIGER witness file. The file names are <prefix>.aim for
        the map file and <prefix>.aiw for the witness file.

    --aig <aim_filename>:<aiw_filename>
        like above, but for map files and witness files that do not
        share a filename prefix (or use different file extensions).

    --aig-noheader
        the AIGER witness file does not include the status and
        properties lines.

    --yw <yosys_witness_filename>
        read a Yosys witness.

    --btorwit <btor_witness_filename>
        read a BTOR witness.

    --noinfo
        only run the core proof, do not collect and print any
        additional information (e.g. which assert failed)

    --presat
        check if the design with assumptions but without assertions
        is SAT before checking if assertions are UNSAT. This will
        detect if there are contradicting assumptions. In some cases
        this will also help to "warm up" the solver, potentially
        yielding a speedup.
```

(continues on next page)

(continued from previous page)

```
    --final-only
        only check final constraints, assume base case

    --assume-skipped <start_step>
        assume asserts in skipped steps in BMC.
        no assumptions are created for skipped steps
        before <start_step>.

    --dump-vcd <vcd_filename>
        write trace to this VCD file
        (hint: use 'write_smt2 -wires' for maximum
        coverage of signals in generated VCD file)

    --dump-yw <yw_filename>
        write trace as a Yosys witness trace

    --dump-vlogtb <verilog_filename>
        write trace as Verilog test bench

    --vlogtb-top <hierarchical_name>
        use the given entity as top module for the generated
        Verilog test bench. The <hierarchical_name> is relative
        to the design top module without the top module name.

    --dump-smtc <constr_filename>
        write trace as constraints file

    --smtc-init
        write just the last state as initial constraint to smtc file

    --smtc-top <old>[:<new>]
        replace <old> with <new> in constraints dumped to smtc
        file and only dump object below <old> in design hierarchy.

    --noinit
        do not assume initial conditions in state 0

    --dump-all
        when using -g or -i, create a dump file for each
        step. The character '%' is replaced in all dump
        filenames with the step number.

    --append <num_steps>
        add <num_steps> time steps at the end of the trace
        when creating a counter example (this additional time
        steps will still be constrained by assumptions)

    --binary
        dump anyconst values as raw bit strings

    --keep-going
```

(continues on next page)

(continued from previous page)

```
    continue BMC after the first failed assertion and report
    further failed assertions. To output multiple traces
    covering all found failed assertions, the character '%' is
    replaced in all dump filenames with an increasing number.
    In cover mode, don't stop when a cover trace contains a failed
    assertion.

--check-witness
    check that the used witness file contains sufficient
    constraints to force an assertion failure.

--detect-loops
    check if states are unique in temporal induction counter examples
    (this feature is experimental and incomplete)

--incremental
    run in incremental mode (experimental)

--track-assumes
    track individual assumptions and report a subset of used
    assumptions that are sufficient for the reported outcome. This
    can be used to debug PREUNSAT failures as well as to find a
    smaller set of sufficient assumptions.

--minimize-assumes
    when using --track-assumes, solve for a minimal set of sufficient assumptions.

-s <solver>
    set SMT solver: z3, yices, boolector, bitwuzla, cvc4, cvc5, mathsat, dummy
    default: yices

-S <opt>
    pass <opt> as command line argument to the solver

--timeout <value>
    set the solver timeout to the specified value (in seconds).

--logic <smt2_logic>
    use the specified SMT2 logic (e.g. QF_AUFBV)

--dummy <filename>
    if solver is "dummy", read solver output from that file
    otherwise: write solver output to that file

--smt2-option <option>=<value>
    enable an SMT-LIBv2 option.

-v
    enable debug output

--unroll
    unroll uninterpreted functions
```

(continues on next page)

(continued from previous page)

```
    --noincr
        don't use incremental solving, instead restart solver for
        each (check-sat). This also avoids (push) and (pop).

    --noprogress
        disable timer display during solving
        (this option is set implicitly on Windows)

    --dump-smt2 <filename>
        write smt2 statements to file

    --info <smt2-info-stmt>
        include the specified smt2 info statement in the smt2 output

    --nocomments
        strip all comments from the generated smt2 code
```

8.5 yosys-witness

`yosys-witness` is a new tool to inspect and convert yosys witness traces. This is used in SBY and SCY for producing traces in a consistent format independent of the solver.

```
Usage: yosys-witness [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  aiw2yw   Convert an AIGER witness trace into a Yosys witness trace.
  display  Display a Yosys witness trace in a human readable format.
  stats    Display statistics of a Yosys witness trace.
  wit2yw   Convert a BTOR witness trace into a Yosys witness trace.
  yw2aiw   Convert a Yosys witness trace into an AIGER witness trace.
  yw2yw    Transform a Yosys witness trace.
```

Note

`yosys-witness` requires click Python package for use.

Todo

see if we can get the two hanging appnotes as lit references

CHAPTER

NINE

INTERNAL CELL LIBRARY

The intermediate language used by Yosys (RTLIL) represents logic and memory with a series of cells. This section provides details for those cells, breaking them down into two major categories: coarse-grain word-level cells; and fine-grain gate-level cells. An additional section contains a list of properties which may be shared across multiple cells.

9.1 Word-level cells

Most of the RTL cells closely resemble the operators available in HDLs such as Verilog or VHDL. Therefore Verilog operators are used in the following sections to define the behaviour of the RTL cells.

Note that all RTL cells have parameters indicating the size of inputs and outputs. When passes modify RTL cells they must always keep the values of these parameters in sync with the size of the signals connected to the inputs and outputs.

Simulation models for the RTL cells can be found in the file `techlibs/common/simlib.v` in the Yosys source tree.

9.1.1 Unary operators

All unary RTL cells have one input port `A` and one output port `Y`. They also have the following parameters:

`A_SIGNED`
: Set to a non-zero value if the input `A` is signed and therefore should be sign-extended when needed.

`A_WIDTH`
: The width of the input port `A`.

`Y_WIDTH`
: The width of the output port `Y`.

Table 9.1: Cell types for unary operators with their corresponding Verilog expressions.

Verilog	Cell Type
`Y = ~A`	*`$not`*
`Y = +A`	*`$pos`*
`Y = -A`	*`$neg`*
`Y = &A`	*`$reduce_and`*
`Y = |A`	*`$reduce_or`*
`Y = ^A`	*`$reduce_xor`*
`Y = ~^A`	*`$reduce_xnor`*
`Y = |A`	*`$reduce_bool`*
`Y = !A`	*`$logic_not`*

For the unary cells that output a logical value (*$reduce_and*, *$reduce_or*, *$reduce_xor*, *$reduce_xnor*, *$reduce_bool*, *$logic_not*), when the Y_WIDTH parameter is greater than 1, the output is zero-extended, and only the least significant bit varies.

Note that *$reduce_or* and *$reduce_bool* generally represent the same logic function. But the *read_verilog* frontend will generate them in different situations. A *$reduce_or* cell is generated when the prefix | operator is being used. A *$reduce_bool* cell is generated when a bit vector is used as a condition in an if-statement or ?:-expression.

yosys> help $buf

A simple coarse-grain buffer cell type for the experimental buffered-normalized mode. Note this cell does't get removed by 'opt_clean' and is not recommended for general use.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.1: simlib.v

```
106 module \$buf (A, Y);
107
108     parameter WIDTH = 0;
109
110     input [WIDTH-1:0] A;
111     output [WIDTH-1:0] Y;
112
113     assign Y = A;
114
115 endmodule
```

yosys> help $logic_not

A logical inverter. This corresponds to the Verilog unary prefix '!' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.2: simlib.v

```
1521 module \$logic_not (A, Y);
1522
1523     parameter A_SIGNED = 0;
1524     parameter A_WIDTH = 0;
1525     parameter Y_WIDTH = 0;
1526
1527     input [A_WIDTH-1:0] A;
1528     output [Y_WIDTH-1:0] Y;
1529
1530     generate
1531         if (A_SIGNED) begin:BLOCK1
1532             assign Y = !$signed(A);
1533         end else begin:BLOCK2
```

(continues on next page)

(continued from previous page)

```
1534            assign Y = !A;
1535        end
1536    endgenerate
1537
1538 endmodule
```

yosys> help $neg

An arithmetic inverter. This corresponds to the Verilog unary prefix '-' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.3: simlib.v

```
126 module \$neg (A, Y);
127
128     parameter A_SIGNED = 0;
129     parameter A_WIDTH = 0;
130     parameter Y_WIDTH = 0;
131
132     input [A_WIDTH-1:0] A;
133     output [Y_WIDTH-1:0] Y;
134
135     generate
136         if (A_SIGNED) begin:BLOCK1
137             assign Y = -$signed(A);
138         end else begin:BLOCK2
139             assign Y = -A;
140         end
141     endgenerate
142
143 endmodule
```

yosys> help $not

Bit-wise inverter

This corresponds to the Verilog unary prefix '~' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.4: simlib.v

```
48 module \$not (A, Y);
49
50     parameter A_SIGNED = 0;
51     parameter A_WIDTH = 0;
52     parameter Y_WIDTH = 0;
53
54     input [A_WIDTH-1:0] A;
```

(continues on next page)

(continued from previous page)

```
55      output [Y_WIDTH-1:0] Y;
56
57      generate
58          if (A_SIGNED) begin:BLOCK1
59              assign Y = ~$signed(A);
60          end else begin:BLOCK2
61              assign Y = ~A;
62          end
63      endgenerate
64
65  endmodule
```

yosys> help $pos

A buffer. This corresponds to the Verilog unary prefix '+' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.5: simlib.v

```
76  module \$pos (A, Y);
77
78      parameter A_SIGNED = 0;
79      parameter A_WIDTH = 0;
80      parameter Y_WIDTH = 0;
81
82      input [A_WIDTH-1:0] A;
83      output [Y_WIDTH-1:0] Y;
84
85      generate
86          if (A_SIGNED) begin:BLOCK1
87              assign Y = $signed(A);
88          end else begin:BLOCK2
89              assign Y = A;
90          end
91      endgenerate
92
93  endmodule
```

yosys> help $reduce_and

An AND reduction. This corresponds to the Verilog unary prefix '&' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.6: simlib.v

```
278 module \$reduce_and (A, Y);
279
280     parameter A_SIGNED = 0;
```

(continues on next page)

(continued from previous page)

```
281     parameter A_WIDTH = 0;
282     parameter Y_WIDTH = 0;
283
284     input [A_WIDTH-1:0] A;
285     output [Y_WIDTH-1:0] Y;
286
287     generate
288         if (A_SIGNED) begin:BLOCK1
289             assign Y = &$signed(A);
290         end else begin:BLOCK2
291             assign Y = &A;
292         end
293     endgenerate
294
295 endmodule
```

yosys> help $reduce_bool

An OR reduction. This cell type is used instead of $reduce_or when a signal is implicitly converted to a boolean signal, e.g. for operands of '&&' and '||'.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.7: simlib.v

```
391 module \$reduce_bool (A, Y);
392
393     parameter A_SIGNED = 0;
394     parameter A_WIDTH = 0;
395     parameter Y_WIDTH = 0;
396
397     input [A_WIDTH-1:0] A;
398     output [Y_WIDTH-1:0] Y;
399
400     generate
401         if (A_SIGNED) begin:BLOCK1
402             assign Y = !(!$signed(A));
403         end else begin:BLOCK2
404             assign Y = !(!A);
405         end
406     endgenerate
407
408 endmodule
```

yosys> help $reduce_or

An OR reduction. This corresponds to the Verilog unary prefix '|' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.8: simlib.v

```
306 module \$reduce_or (A, Y);
307
308     parameter A_SIGNED = 0;
309     parameter A_WIDTH = 0;
310     parameter Y_WIDTH = 0;
311
312     input [A_WIDTH-1:0] A;
313     output [Y_WIDTH-1:0] Y;
314
315     generate
316         if (A_SIGNED) begin:BLOCK1
317             assign Y = |$signed(A);
318         end else begin:BLOCK2
319             assign Y = |A;
320         end
321     endgenerate
322
323 endmodule
```

yosys> help $reduce_xnor

A XNOR reduction. This corresponds to the Verilog unary prefix '~^' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.9: simlib.v

```
362 module \$reduce_xnor (A, Y);
363
364     parameter A_SIGNED = 0;
365     parameter A_WIDTH = 0;
366     parameter Y_WIDTH = 0;
367
368     input [A_WIDTH-1:0] A;
369     output [Y_WIDTH-1:0] Y;
370
371     generate
372         if (A_SIGNED) begin:BLOCK1
373             assign Y = ~^$signed(A);
374         end else begin:BLOCK2
375             assign Y = ~^A;
376         end
377     endgenerate
378
379 endmodule
```

yosys> help $reduce_xor

A XOR reduction. This corresponds to the Verilog unary prefix '^' operator.

Properties

is_evaluable

`Simulation model` (verilog)

Listing 9.10: simlib.v

```
334 module \$reduce_xor (A, Y);
335
336     parameter A_SIGNED = 0;
337     parameter A_WIDTH = 0;
338     parameter Y_WIDTH = 0;
339
340     input [A_WIDTH-1:0] A;
341     output [Y_WIDTH-1:0] Y;
342
343     generate
344         if (A_SIGNED) begin:BLOCK1
345             assign Y = ^$signed(A);
346         end else begin:BLOCK2
347             assign Y = ^A;
348         end
349     endgenerate
350
351 endmodule
```

9.1.2 Binary operators

All binary RTL cells have two input ports `A` and `B` and one output port `Y`. They also have the following parameters:

`A_SIGNED`
: Set to a non-zero value if the input `A` is signed and therefore should be sign-extended when needed.

`A_WIDTH`
: The width of the input port `A`.

`B_SIGNED`
: Set to a non-zero value if the input `B` is signed and therefore should be sign-extended when needed.

`B_WIDTH`
: The width of the input port `B`.

`Y_WIDTH`
: The width of the output port `Y`.

Table 9.2: Cell types for binary operators with their corresponding Verilog expressions.

Verilog	Cell Type	Verilog	Cell Type
Y = A & B	*$and*	Y = A ** B	*$pow*
Y = A | B	*$or*	Y = A < B	*$lt*
Y = A ^ B	*$xor*	Y = A <= B	*$le*
Y = A ~^ B	*$xnor*	Y = A == B	*$eq*
Y = A << B	*$shl*	Y = A != B	*$ne*
Y = A >> B	*$shr*	Y = A >= B	*$ge*
Y = A <<< B	*$sshl*	Y = A > B	*$gt*
Y = A >>> B	*$sshr*	Y = A + B	*$add*
Y = A && B	*$logic_and*	Y = A - B	*$sub*
Y = A || B	*$logic_or*	Y = A * B	*$mul*
Y = A === B	*$eqx*	Y = A / B	*$div*
Y = A !== B	*$nex*	Y = A % B	*$mod*
N/A	*$shift*	N/A	*$divfloor*
N/A	*$shiftx*	N/A	*$modfloor*

The *$shl* and *$shr* cells implement logical shifts, whereas the *$sshl* and *$sshr* cells implement arithmetic shifts. The *$shl* and *$sshl* cells implement the same operation. All four of these cells interpret the second operand as unsigned, and require `B_SIGNED` to be zero.

Two additional shift operator cells are available that do not directly correspond to any operator in Verilog, *$shift* and *$shiftx*. The *$shift* cell performs a right logical shift if the second operand is positive (or unsigned), and a left logical shift if it is negative. The *$shiftx* cell performs the same operation as the *$shift* cell, but the vacated bit positions are filled with undef (x) bits, and corresponds to the Verilog indexed part-select expression.

For the binary cells that output a logical value (*$logic_and*, *$logic_or*, *$eqx*, *$nex*, *$lt*, *$le*, *$eq*, *$ne*, *$ge*, *$gt*), when the `Y_WIDTH` parameter is greater than 1, the output is zero-extended, and only the least significant bit varies.

Division and modulo cells are available in two rounding modes. The original *$div* and *$mod* cells are based on truncating division, and correspond to the semantics of the verilog `/` and `%` operators. The *$divfloor* and *$modfloor* cells represent flooring division and flooring modulo, the latter of which corresponds to the `%` operator in Python. See the following table for a side-by-side comparison between the different semantics.

Table 9.3: Comparison between different rounding modes for division and modulo cells.

Division	Result	Truncating		Flooring	
		$div	$mod	$divfloor	$modfloor
-10 / 3	-3.3	-3	-1	-4	2
10 / -3	-3.3	-3	1	-4	-2
-10 / -3	3.3	3	-1	3	-1
10 / 3	3.3	3	1	3	1

```
yosys> help $add
```

Addition of inputs 'A' and 'B'. This corresponds to the Verilog '+' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.11: simlib.v

```
module \$add (A, B, Y);

    parameter A_SIGNED = 0;
    parameter B_SIGNED = 0;
    parameter A_WIDTH = 0;
    parameter B_WIDTH = 0;
    parameter Y_WIDTH = 0;

    input [A_WIDTH-1:0] A;
    input [B_WIDTH-1:0] B;
    output [Y_WIDTH-1:0] Y;

    generate
        if (A_SIGNED && B_SIGNED) begin:BLOCK1
            assign Y = $signed(A) + $signed(B);
        end else begin:BLOCK2
            assign Y = A + B;
        end
    endgenerate

endmodule
```

yosys> help $and

A bit-wise AND. This corresponds to the Verilog '&' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.12: simlib.v

```
module \$and (A, B, Y);

    parameter A_SIGNED = 0;
    parameter B_SIGNED = 0;
    parameter A_WIDTH = 0;
    parameter B_WIDTH = 0;
    parameter Y_WIDTH = 0;

    input [A_WIDTH-1:0] A;
    input [B_WIDTH-1:0] B;
    output [Y_WIDTH-1:0] Y;

    generate
        if (A_SIGNED && B_SIGNED) begin:BLOCK1
            assign Y = $signed(A) & $signed(B);
        end else begin:BLOCK2
            assign Y = A & B;
        end
```

(continues on next page)

(continued from previous page)

```
172     endgenerate
173
174 endmodule
```

yosys> help $bweqx

Bit-wise case equality

A bit-wise version of *$eqx*.

Properties

- *is_evaluable*
- *x-aware*

Simulation model (verilog)

Listing 9.13: simlib.v

```
2013 module \$bweqx (A, B, Y);
2014
2015     parameter WIDTH = 0;
2016
2017     input [WIDTH-1:0] A, B;
2018     output [WIDTH-1:0] Y;
2019
2020     genvar i;
2021     generate
2022         for (i = 0; i < WIDTH; i = i + 1) begin:slices
2023             assign Y[i] = A[i] === B[i];
2024         end
2025     endgenerate
2026
2027 endmodule
```

yosys> help $div

Divider

This corresponds to the Verilog '/' operator, performing division and truncating the result (rounding towards 0).

Properties

- *is_evaluable*
- *x-output*

Simulation model (verilog)

Listing 9.14: simlib.v

```
1340 module \$div (A, B, Y);
1341
1342     parameter A_SIGNED = 0;
1343     parameter B_SIGNED = 0;
1344     parameter A_WIDTH = 0;
1345     parameter B_WIDTH = 0;
```

(continues on next page)

(continued from previous page)

```
1346     parameter Y_WIDTH = 0;
1347
1348     input [A_WIDTH-1:0] A;
1349     input [B_WIDTH-1:0] B;
1350     output [Y_WIDTH-1:0] Y;
1351
1352     generate
1353         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1354             assign Y = $signed(A) / $signed(B);
1355         end else begin:BLOCK2
1356             assign Y = A / B;
1357         end
1358     endgenerate
1359
1360 endmodule
```

yosys> help **$divfloor**

Division with floored result (rounded towards negative infinity).

Properties

is_evaluable

Simulation model (verilog)

Listing 9.15: simlib.v

```
1403 module \$divfloor (A, B, Y);
1404
1405     parameter A_SIGNED = 0;
1406     parameter B_SIGNED = 0;
1407     parameter A_WIDTH = 0;
1408     parameter B_WIDTH = 0;
1409     parameter Y_WIDTH = 0;
1410
1411     input [A_WIDTH-1:0] A;
1412     input [B_WIDTH-1:0] B;
1413     output [Y_WIDTH-1:0] Y;
1414
1415     generate
1416         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1417             localparam WIDTH =
1418                     A_WIDTH >= B_WIDTH && A_WIDTH >= Y_WIDTH ? A_WIDTH :
1419                     B_WIDTH >= A_WIDTH && B_WIDTH >= Y_WIDTH ? B_WIDTH : Y_WIDTH;
1420             wire [WIDTH:0] A_buf, B_buf, N_buf;
1421             assign A_buf = $signed(A);
1422             assign B_buf = $signed(B);
1423             assign N_buf = (A[A_WIDTH-1] == B[B_WIDTH-1]) || A == 0 ? A_buf :
↪$signed(A_buf - (B[B_WIDTH-1] ? B_buf+1 : B_buf-1));
1424             assign Y = $signed(N_buf) / $signed(B_buf);
1425         end else begin:BLOCK2
1426             assign Y = A / B;
1427         end
1428     endgenerate
```

(continues on next page)

(continued from previous page)

```
1429
1430 endmodule
```

yosys> help $eq

An equality comparison between inputs 'A' and 'B'. This corresponds to the Verilog '==' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.16: simlib.v

```
790 module \$eq (A, B, Y);
791
792     parameter A_SIGNED = 0;
793     parameter B_SIGNED = 0;
794     parameter A_WIDTH = 0;
795     parameter B_WIDTH = 0;
796     parameter Y_WIDTH = 0;
797
798     input [A_WIDTH-1:0] A;
799     input [B_WIDTH-1:0] B;
800     output [Y_WIDTH-1:0] Y;
801
802     generate
803         if (A_SIGNED && B_SIGNED) begin:BLOCK1
804             assign Y = $signed(A) == $signed(B);
805         end else begin:BLOCK2
806             assign Y = A == B;
807         end
808     endgenerate
809
810 endmodule
```

yosys> help $eqx

Case equality

An exact equality comparison between inputs 'A' and 'B'. Also known as the case equality operator. This corresponds to the Verilog '===' operator. Unlike equality comparison that can give 'x' as output, an exact equality comparison will strictly give '0' or '1' as output, even if input includes 'x' or 'z' values.

Properties

- *is_evaluable*
- *x-aware*

Simulation model (verilog)

Listing 9.17: simlib.v

```
855 module \$eqx (A, B, Y);
856
```

(continues on next page)

(continued from previous page)

```
857     parameter A_SIGNED = 0;
858     parameter B_SIGNED = 0;
859     parameter A_WIDTH = 0;
860     parameter B_WIDTH = 0;
861     parameter Y_WIDTH = 0;
862
863     input [A_WIDTH-1:0] A;
864     input [B_WIDTH-1:0] B;
865     output [Y_WIDTH-1:0] Y;
866
867     generate
868         if (A_SIGNED && B_SIGNED) begin:BLOCK1
869             assign Y = $signed(A) === $signed(B);
870         end else begin:BLOCK2
871             assign Y = A === B;
872         end
873     endgenerate
874
875 endmodule
```

yosys> help $ge

A greater-than-or-equal-to comparison between inputs ‘A’ and ‘B’. This corresponds to the Verilog ‘>=’ operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.18: simlib.v

```
918 module \$ge (A, B, Y);
919
920     parameter A_SIGNED = 0;
921     parameter B_SIGNED = 0;
922     parameter A_WIDTH = 0;
923     parameter B_WIDTH = 0;
924     parameter Y_WIDTH = 0;
925
926     input [A_WIDTH-1:0] A;
927     input [B_WIDTH-1:0] B;
928     output [Y_WIDTH-1:0] Y;
929
930     generate
931         if (A_SIGNED && B_SIGNED) begin:BLOCK1
932             assign Y = $signed(A) >= $signed(B);
933         end else begin:BLOCK2
934             assign Y = A >= B;
935         end
936     endgenerate
937
938 endmodule
```

yosys> help $gt

A greater-than comparison between inputs 'A' and 'B'. This corresponds to the Verilog '>' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.19: simlib.v

```
950 module \$gt (A, B, Y);
951
952     parameter A_SIGNED = 0;
953     parameter B_SIGNED = 0;
954     parameter A_WIDTH = 0;
955     parameter B_WIDTH = 0;
956     parameter Y_WIDTH = 0;
957
958     input [A_WIDTH-1:0] A;
959     input [B_WIDTH-1:0] B;
960     output [Y_WIDTH-1:0] Y;
961
962     generate
963         if (A_SIGNED && B_SIGNED) begin:BLOCK1
964             assign Y = $signed(A) > $signed(B);
965         end else begin:BLOCK2
966             assign Y = A > B;
967         end
968     endgenerate
969
970 endmodule
```

yosys> help $le

A less-than-or-equal-to comparison between inputs 'A' and 'B'. This corresponds to the Verilog '<=' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.20: simlib.v

```
758 module \$le (A, B, Y);
759
760     parameter A_SIGNED = 0;
761     parameter B_SIGNED = 0;
762     parameter A_WIDTH = 0;
763     parameter B_WIDTH = 0;
764     parameter Y_WIDTH = 0;
765
766     input [A_WIDTH-1:0] A;
767     input [B_WIDTH-1:0] B;
768     output [Y_WIDTH-1:0] Y;
769
```

(continues on next page)

(continued from previous page)

```
770     generate
771         if (A_SIGNED && B_SIGNED) begin:BLOCK1
772             assign Y = $signed(A) <= $signed(B);
773         end else begin:BLOCK2
774             assign Y = A <= B;
775         end
776     endgenerate
777
778 endmodule
```

yosys> help $logic_and

A logical AND. This corresponds to the Verilog '&&' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.21: simlib.v

```
1549 module \$logic_and (A, B, Y);
1550
1551     parameter A_SIGNED = 0;
1552     parameter B_SIGNED = 0;
1553     parameter A_WIDTH = 0;
1554     parameter B_WIDTH = 0;
1555     parameter Y_WIDTH = 0;
1556
1557     input [A_WIDTH-1:0] A;
1558     input [B_WIDTH-1:0] B;
1559     output [Y_WIDTH-1:0] Y;
1560
1561     generate
1562         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1563             assign Y = $signed(A) && $signed(B);
1564         end else begin:BLOCK2
1565             assign Y = A && B;
1566         end
1567     endgenerate
1568
1569 endmodule
```

yosys> help $logic_or

A logical OR. This corresponds to the Verilog '||' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.22: simlib.v

```
1580 module \$logic_or (A, B, Y);
1581
```

(continues on next page)

(continued from previous page)

```
1582     parameter A_SIGNED = 0;
1583     parameter B_SIGNED = 0;
1584     parameter A_WIDTH = 0;
1585     parameter B_WIDTH = 0;
1586     parameter Y_WIDTH = 0;
1587
1588     input [A_WIDTH-1:0] A;
1589     input [B_WIDTH-1:0] B;
1590     output [Y_WIDTH-1:0] Y;
1591
1592     generate
1593         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1594             assign Y = $signed(A) || $signed(B);
1595         end else begin:BLOCK2
1596             assign Y = A || B;
1597         end
1598     endgenerate
1599
1600 endmodule
```

yosys> help $lt

A less-than comparison between inputs ‘A’ and ‘B’. This corresponds to the Verilog ‘<’ operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.23: simlib.v

```
726 module \$lt (A, B, Y);
727
728     parameter A_SIGNED = 0;
729     parameter B_SIGNED = 0;
730     parameter A_WIDTH = 0;
731     parameter B_WIDTH = 0;
732     parameter Y_WIDTH = 0;
733
734     input [A_WIDTH-1:0] A;
735     input [B_WIDTH-1:0] B;
736     output [Y_WIDTH-1:0] Y;
737
738     generate
739         if (A_SIGNED && B_SIGNED) begin:BLOCK1
740             assign Y = $signed(A) < $signed(B);
741         end else begin:BLOCK2
742             assign Y = A < B;
743         end
744     endgenerate
745
746 endmodule
```

yosys> help $mod

Modulo

This corresponds to the Verilog '%' operator, giving the module (or remainder) of division and truncating the result (rounding towards 0).

Invariant: $div(A, B) * B + $mod(A, B) == A

Properties

- *is_evaluable*
- *x-output*

Simulation model (verilog)

Listing 9.24: simlib.v

```
module \$mod (A, B, Y);

    parameter A_SIGNED = 0;
    parameter B_SIGNED = 0;
    parameter A_WIDTH = 0;
    parameter B_WIDTH = 0;
    parameter Y_WIDTH = 0;

    input [A_WIDTH-1:0] A;
    input [B_WIDTH-1:0] B;
    output [Y_WIDTH-1:0] Y;

    generate
        if (A_SIGNED && B_SIGNED) begin:BLOCK1
            assign Y = $signed(A) % $signed(B);
        end else begin:BLOCK2
            assign Y = A % B;
        end
    endgenerate

endmodule
```

yosys> help $modfloor

Modulo/remainder of division with floored result (rounded towards negative infinity).

Invariant: $divfloor(A, B) * B + $modfloor(A, B) == A

Properties

is_evaluable

Simulation model (verilog)

Listing 9.25: simlib.v

```
module \$modfloor (A, B, Y);

    parameter A_SIGNED = 0;
    parameter B_SIGNED = 0;
    parameter A_WIDTH = 0;
    parameter B_WIDTH = 0;
    parameter Y_WIDTH = 0;
```

(continues on next page)

(continued from previous page)

```
1450
1451        input [A_WIDTH-1:0] A;
1452        input [B_WIDTH-1:0] B;
1453        output [Y_WIDTH-1:0] Y;
1454
1455        generate
1456            if (A_SIGNED && B_SIGNED) begin:BLOCK1
1457                localparam WIDTH = B_WIDTH >= Y_WIDTH ? B_WIDTH : Y_WIDTH;
1458                wire [WIDTH-1:0] B_buf, Y_trunc;
1459                assign B_buf = $signed(B);
1460                assign Y_trunc = $signed(A) % $signed(B);
1461                // flooring mod is the same as truncating mod for positive division␣
    ↪results (A and B have
1462                // the same sign), as well as when there's no remainder.
1463                // For all other cases, they behave as `floor - trunc = B`
1464                assign Y = (A[A_WIDTH-1] == B[B_WIDTH-1]) || Y_trunc == 0 ? Y_trunc :
    ↪$signed(B_buf) + $signed(Y_trunc);
1465            end else begin:BLOCK2
1466                // no difference between truncating and flooring for unsigned
1467                assign Y = A % B;
1468            end
1469        endgenerate
1470
1471    endmodule
```

yosys> help $mul

Multiplication of inputs ‘A’ and ‘B’. This corresponds to the Verilog ‘*’ operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.26: simlib.v

```
1045    module \$mul (A, B, Y);
1046
1047        parameter A_SIGNED = 0;
1048        parameter B_SIGNED = 0;
1049        parameter A_WIDTH = 0;
1050        parameter B_WIDTH = 0;
1051        parameter Y_WIDTH = 0;
1052
1053        input [A_WIDTH-1:0] A;
1054        input [B_WIDTH-1:0] B;
1055        output [Y_WIDTH-1:0] Y;
1056
1057        generate
1058            if (A_SIGNED && B_SIGNED) begin:BLOCK1
1059                assign Y = $signed(A) * $signed(B);
1060            end else begin:BLOCK2
1061                assign Y = A * B;
1062            end
```

(continues on next page)

(continued from previous page)

```
1063     endgenerate
1064
1065 endmodule
```

yosys> help $ne

An inequality comparison between inputs 'A' and 'B'. This corresponds to the Verilog '!=' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.27: simlib.v

```
822 module \$ne (A, B, Y);
823
824     parameter A_SIGNED = 0;
825     parameter B_SIGNED = 0;
826     parameter A_WIDTH = 0;
827     parameter B_WIDTH = 0;
828     parameter Y_WIDTH = 0;
829
830     input [A_WIDTH-1:0] A;
831     input [B_WIDTH-1:0] B;
832     output [Y_WIDTH-1:0] Y;
833
834     generate
835         if (A_SIGNED && B_SIGNED) begin:BLOCK1
836             assign Y = $signed(A) != $signed(B);
837         end else begin:BLOCK2
838             assign Y = A != B;
839         end
840     endgenerate
841
842 endmodule
```

yosys> help $nex

Case inequality

This corresponds to the Verilog '!==' operator.

Refer to *$eqx* for more details.

Properties

- *is_evaluable*
- *x-aware*

Simulation model (verilog)

Listing 9.28: simlib.v

```
886 module \$nex (A, B, Y);
887
888     parameter A_SIGNED = 0;
```

(continues on next page)

(continued from previous page)

```
889     parameter B_SIGNED = 0;
890     parameter A_WIDTH = 0;
891     parameter B_WIDTH = 0;
892     parameter Y_WIDTH = 0;
893
894     input [A_WIDTH-1:0] A;
895     input [B_WIDTH-1:0] B;
896     output [Y_WIDTH-1:0] Y;
897
898     generate
899         if (A_SIGNED && B_SIGNED) begin:BLOCK1
900             assign Y = $signed(A) !== $signed(B);
901         end else begin:BLOCK2
902             assign Y = A !== B;
903         end
904     endgenerate
905
906 endmodule
```

yosys> help $or

A bit-wise OR. This corresponds to the Verilog '|' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.29: simlib.v

```
185 module \$or (A, B, Y);
186
187     parameter A_SIGNED = 0;
188     parameter B_SIGNED = 0;
189     parameter A_WIDTH = 0;
190     parameter B_WIDTH = 0;
191     parameter Y_WIDTH = 0;
192
193     input [A_WIDTH-1:0] A;
194     input [B_WIDTH-1:0] B;
195     output [Y_WIDTH-1:0] Y;
196
197     generate
198         if (A_SIGNED && B_SIGNED) begin:BLOCK1
199             assign Y = $signed(A) | $signed(B);
200         end else begin:BLOCK2
201             assign Y = A | B;
202         end
203     endgenerate
204
205 endmodule
```

yosys> help $pow

Exponentiation of an input (Y = A ** B). This corresponds to the Verilog '**' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.30: simlib.v

```
1485 module \$pow (A, B, Y);
1486
1487     parameter A_SIGNED = 0;
1488     parameter B_SIGNED = 0;
1489     parameter A_WIDTH = 0;
1490     parameter B_WIDTH = 0;
1491     parameter Y_WIDTH = 0;
1492
1493     input [A_WIDTH-1:0] A;
1494     input [B_WIDTH-1:0] B;
1495     output [Y_WIDTH-1:0] Y;
1496
1497     generate
1498         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1499             assign Y = $signed(A) ** $signed(B);
1500         end else if (A_SIGNED) begin:BLOCK2
1501             assign Y = $signed(A) ** B;
1502         end else if (B_SIGNED) begin:BLOCK3
1503             assign Y = A ** $signed(B);
1504         end else begin:BLOCK4
1505             assign Y = A ** B;
1506         end
1507     endgenerate
1508
1509 endmodule
```

yosys> help **$shift**

Variable shifter

Performs a right logical shift if the second operand is positive (or unsigned), and a left logical shift if it is negative.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.31: simlib.v

```
543 module \$shift (A, B, Y);
544
545     parameter A_SIGNED = 0;
546     parameter B_SIGNED = 0;
547     parameter A_WIDTH = 0;
548     parameter B_WIDTH = 0;
549     parameter Y_WIDTH = 0;
550
551     input [A_WIDTH-1:0] A;
```

(continues on next page)

(continued from previous page)

```
552     input [B_WIDTH-1:0] B;
553     output [Y_WIDTH-1:0] Y;
554
555     generate
556         if (A_SIGNED) begin:BLOCK1
557             if (B_SIGNED) begin:BLOCK2
558                 assign Y = $signed(B) < 0 ? $signed(A) << -B : $signed(A) >> B;
559             end else begin:BLOCK3
560                 assign Y = $signed(A) >> B;
561             end
562         end else begin:BLOCK4
563             if (B_SIGNED) begin:BLOCK5
564                 assign Y = $signed(B) < 0 ? A << -B : A >> B;
565             end else begin:BLOCK6
566                 assign Y = A >> B;
567             end
568         end
569     endgenerate
570
571 endmodule
```

yosys> help **$shiftx**

Indexed part-select

Same as the *$shift* cell, but fills with 'x'.

Properties

- *is_evaluable*
- *x-output*

Simulation model (verilog)

Listing 9.32: simlib.v

```
580 module \$shiftx (A, B, Y);
581
582     parameter A_SIGNED = 0;
583     parameter B_SIGNED = 0;
584     parameter A_WIDTH = 0;
585     parameter B_WIDTH = 0;
586     parameter Y_WIDTH = 0;
587
588     input [A_WIDTH-1:0] A;
589     input [B_WIDTH-1:0] B;
590     output [Y_WIDTH-1:0] Y;
591
592     generate
593         if (Y_WIDTH > 0)
594             if (B_SIGNED) begin:BLOCK1
595                 assign Y = A[$signed(B) +: Y_WIDTH];
596             end else begin:BLOCK2
597                 assign Y = A[B +: Y_WIDTH];
```

(continues on next page)

(continued from previous page)

```
598             end
599     endgenerate
600
601 endmodule
```

yosys> help $shl

A logical shift-left operation. This corresponds to the Verilog '<<' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.33: simlib.v

```
419 module \$shl (A, B, Y);
420
421     parameter A_SIGNED = 0;
422     parameter B_SIGNED = 0;
423     parameter A_WIDTH = 0;
424     parameter B_WIDTH = 0;
425     parameter Y_WIDTH = 0;
426
427     input [A_WIDTH-1:0] A;
428     input [B_WIDTH-1:0] B;
429     output [Y_WIDTH-1:0] Y;
430
431     generate
432         if (A_SIGNED) begin:BLOCK1
433             assign Y = $signed(A) << B;
434         end else begin:BLOCK2
435             assign Y = A << B;
436         end
437     endgenerate
438
439 endmodule
```

yosys> help $shr

A logical shift-right operation. This corresponds to the Verilog '>>' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.34: simlib.v

```
450 module \$shr (A, B, Y);
451
452     parameter A_SIGNED = 0;
453     parameter B_SIGNED = 0;
454     parameter A_WIDTH = 0;
455     parameter B_WIDTH = 0;
456     parameter Y_WIDTH = 0;
```

(continues on next page)

(continued from previous page)

```
457
458     input [A_WIDTH-1:0] A;
459     input [B_WIDTH-1:0] B;
460     output [Y_WIDTH-1:0] Y;
461
462     generate
463         if (A_SIGNED) begin:BLOCK1
464             assign Y = $signed(A) >> B;
465         end else begin:BLOCK2
466             assign Y = A >> B;
467         end
468     endgenerate
469
470 endmodule
```

yosys> help $sshl

An arithmatic shift-left operation. This corresponds to the Verilog '<<<' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.35: simlib.v

```
482 module \$sshl (A, B, Y);
483
484     parameter A_SIGNED = 0;
485     parameter B_SIGNED = 0;
486     parameter A_WIDTH = 0;
487     parameter B_WIDTH = 0;
488     parameter Y_WIDTH = 0;
489
490     input [A_WIDTH-1:0] A;
491     input [B_WIDTH-1:0] B;
492     output [Y_WIDTH-1:0] Y;
493
494     generate
495         if (A_SIGNED) begin:BLOCK1
496             assign Y = $signed(A) <<< B;
497         end else begin:BLOCK2
498             assign Y = A <<< B;
499         end
500     endgenerate
501
502 endmodule
```

yosys> help $sshr

An arithmatic shift-right operation. This corresponds to the Verilog '>>>' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.36: simlib.v

```
514 module \$sshr (A, B, Y);
515
516     parameter A_SIGNED = 0;
517     parameter B_SIGNED = 0;
518     parameter A_WIDTH = 0;
519     parameter B_WIDTH = 0;
520     parameter Y_WIDTH = 0;
521
522     input [A_WIDTH-1:0] A;
523     input [B_WIDTH-1:0] B;
524     output [Y_WIDTH-1:0] Y;
525
526     generate
527         if (A_SIGNED) begin:BLOCK1
528             assign Y = $signed(A) >>> B;
529         end else begin:BLOCK2
530             assign Y = A >>> B;
531         end
532     endgenerate
533
534 endmodule
```

yosys> help $sub

Subtraction between inputs 'A' and 'B'. This corresponds to the Verilog '-' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.37: simlib.v

```
1013 module \$sub (A, B, Y);
1014
1015     parameter A_SIGNED = 0;
1016     parameter B_SIGNED = 0;
1017     parameter A_WIDTH = 0;
1018     parameter B_WIDTH = 0;
1019     parameter Y_WIDTH = 0;
1020
1021     input [A_WIDTH-1:0] A;
1022     input [B_WIDTH-1:0] B;
1023     output [Y_WIDTH-1:0] Y;
1024
1025     generate
1026         if (A_SIGNED && B_SIGNED) begin:BLOCK1
1027             assign Y = $signed(A) - $signed(B);
1028         end else begin:BLOCK2
1029             assign Y = A - B;
1030         end
1031     endgenerate
1032
```

(continues on next page)

(continued from previous page)

```
1033 endmodule
```

yosys> help $xnor

A bit-wise XNOR. This corresponds to the Verilog '~^' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.38: simlib.v

```
247 module \$xnor (A, B, Y);
248
249     parameter A_SIGNED = 0;
250     parameter B_SIGNED = 0;
251     parameter A_WIDTH = 0;
252     parameter B_WIDTH = 0;
253     parameter Y_WIDTH = 0;
254
255     input [A_WIDTH-1:0] A;
256     input [B_WIDTH-1:0] B;
257     output [Y_WIDTH-1:0] Y;
258
259     generate
260         if (A_SIGNED && B_SIGNED) begin:BLOCK1
261             assign Y = $signed(A) ~^ $signed(B);
262         end else begin:BLOCK2
263             assign Y = A ~^ B;
264         end
265     endgenerate
266
267 endmodule
```

yosys> help $xor

A bit-wise XOR. This corresponds to the Verilog '^' operator.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.39: simlib.v

```
216 module \$xor (A, B, Y);
217
218     parameter A_SIGNED = 0;
219     parameter B_SIGNED = 0;
220     parameter A_WIDTH = 0;
221     parameter B_WIDTH = 0;
222     parameter Y_WIDTH = 0;
223
224     input [A_WIDTH-1:0] A;
225     input [B_WIDTH-1:0] B;
```

(continues on next page)

(continued from previous page)

```
226     output [Y_WIDTH-1:0] Y;
227
228     generate
229         if (A_SIGNED && B_SIGNED) begin:BLOCK1
230             assign Y = $signed(A) ^ $signed(B);
231         end else begin:BLOCK2
232             assign Y = A ^ B;
233         end
234     endgenerate
235
236 endmodule
```

9.1.3 Multiplexers

Multiplexers are generated by the Verilog HDL frontend for `?:`-expressions. Multiplexers are also generated by the proc pass to map the decision trees from RTLIL::Process objects to logic.

The simplest multiplexer cell type is *`$mux`*. Cells of this type have a `WITDH` parameter and data inputs `A` and `B` and a data output `Y`, all of the specified width. This cell also has a single bit control input `S`. If `S` is 0 the value from the input `A` is sent to the output, if it is 1 the value from the `B` input is sent to the output. So the *`$mux`* cell implements the function `Y = S ? B : A`.

The *`$pmux`* cell is used to multiplex between many inputs using a one-hot select signal. Cells of this type have a `WIDTH` and a `S_WIDTH` parameter and inputs `A`, `B`, and `S` and an output `Y`. The `S` input is `S_WIDTH` bits wide. The `A` input and the output are both `WIDTH` bits wide and the `B` input is `WIDTH*S_WIDTH` bits wide. When all bits of `S` are zero, the value from `A` input is sent to the output. If the n'th bit from `S` is set, the value n'th `WIDTH` bits wide slice of the `B` input is sent to the output. When more than one bit from `S` is set the output is undefined. Cells of this type are used to model "parallel cases" (defined by using the `parallel_case` attribute, the `unique` or `unique0` SystemVerilog keywords, or detected by an optimization).

The *`$tribuf`* cell is used to implement tristate logic. Cells of this type have a `WIDTH` parameter and inputs `A` and `EN` and an output `Y`. The `A` input and `Y` output are `WIDTH` bits wide, and the `EN` input is one bit wide. When `EN` is 0, the output is not driven. When `EN` is 1, the value from `A` input is sent to the `Y` output. Therefore, the *`$tribuf`* cell implements the function `Y = EN ? A : 'bz`.

Behavioural code with cascaded if-then-else- and case-statements usually results in trees of multiplexer cells. Many passes (from various optimizations to FSM extraction) heavily depend on these multiplexer trees to understand dependencies between signals. Therefore optimizations should not break these multiplexer trees (e.g. by replacing a multiplexer between a calculated signal and a constant zero with an *`$and`* gate).

```
yosys> help $bmux
```

Binary-encoded multiplexer

Selects between 'slices' of A where each value of S corresponds to a unique slice.

Properties

`is_evaluable`

Simulation model (verilog)

Listing 9.40: simlib.v

```
1666 module \$bmux (A, S, Y);
1667
```

(continues on next page)

(continued from previous page)

```
1668     parameter WIDTH = 0;
1669     parameter S_WIDTH = 0;
1670
1671     input [(WIDTH << S_WIDTH)-1:0] A;
1672     input [S_WIDTH-1:0] S;
1673     output [WIDTH-1:0] Y;
1674
1675     wire [WIDTH-1:0] bm0_out, bm1_out;
1676
1677     generate
1678         if (S_WIDTH > 1) begin:muxlogic
1679             \$bmux #(.WIDTH(WIDTH), .S_WIDTH(S_WIDTH-1)) bm0 (.A(A[(WIDTH << (S_
    ↪WIDTH - 1))-1:0]), .S(S[S_WIDTH-2:0]), .Y(bm0_out));
1680             \$bmux #(.WIDTH(WIDTH), .S_WIDTH(S_WIDTH-1)) bm1 (.A(A[(WIDTH << S_
    ↪WIDTH)-1:WIDTH << (S_WIDTH - 1)]), .S(S[S_WIDTH-2:0]), .Y(bm1_out));
1681             assign Y = S[S_WIDTH-1] ? bm1_out : bm0_out;
1682         end else if (S_WIDTH == 1) begin:simple
1683             assign Y = S ? A[2*WIDTH-1:WIDTH] : A[WIDTH-1:0];
1684         end else begin:passthru
1685             assign Y = A;
1686         end
1687     endgenerate
1688
1689 endmodule
```

yosys> help $bwmux

Bit-wise multiplexer

Equivalent to a series of 1-bit wide *$mux* cells.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.41: simlib.v

```
2035 module \$bwmux (A, B, S, Y);
2036
2037     parameter WIDTH = 0;
2038
2039     input [WIDTH-1:0] A, B;
2040     input [WIDTH-1:0] S;
2041     output [WIDTH-1:0] Y;
2042
2043     genvar i;
2044     generate
2045         for (i = 0; i < WIDTH; i = i + 1) begin:slices
2046             assign Y[i] = S[i] ? B[i] : A[i];
2047         end
2048     endgenerate
2049
2050 endmodule
```

yosys> help $demux

Demultiplexer i.e routing single input to several outputs based on select signal. Unselected outputs are driven to zero.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.42: simlib.v

```
1741 module \$demux (A, S, Y);
1742
1743     parameter WIDTH = 1;
1744     parameter S_WIDTH = 1;
1745
1746     input [WIDTH-1:0] A;
1747     input [S_WIDTH-1:0] S;
1748     output [(WIDTH << S_WIDTH)-1:0] Y;
1749
1750     genvar i;
1751     generate
1752         for (i = 0; i < (1 << S_WIDTH); i = i + 1) begin:slices
1753             assign Y[i*WIDTH+:WIDTH] = (S == i) ? A : 0;
1754         end
1755     endgenerate
1756
1757 endmodule
```

yosys> help $mux

Multiplexer i.e selecting between two inputs based on select signal.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.43: simlib.v

```
1647 module \$mux (A, B, S, Y);
1648
1649     parameter WIDTH = 0;
1650
1651     input [WIDTH-1:0] A, B;
1652     input S;
1653     output [WIDTH-1:0] Y;
1654
1655     assign Y = S ? B : A;
1656
1657 endmodule
```

yosys> help $pmux

Priority-encoded multiplexer

Selects between 'slices' of B where each slice corresponds to a single bit of S. Outputs A when all bits of S are low.

Properties

- *is_evaluable*
- *x-output*

Simulation model (verilog)

Listing 9.44: simlib.v

```
1699 module \$pmux (A, B, S, Y);
1700
1701     parameter WIDTH = 0;
1702     parameter S_WIDTH = 0;
1703
1704     input [WIDTH-1:0] A;
1705     input [WIDTH*S_WIDTH-1:0] B;
1706     input [S_WIDTH-1:0] S;
1707     output reg [WIDTH-1:0] Y;
1708
1709     integer i;
1710     reg found_active_sel_bit;
1711
1712     always @* begin
1713         Y = A;
1714         found_active_sel_bit = 0;
1715         for (i = 0; i < S_WIDTH; i = i+1)
1716             case (S[i])
1717                 1'b1: begin
1718                     Y = found_active_sel_bit ? 'bx : B >> (WIDTH*i);
1719                     found_active_sel_bit = 1;
1720                 end
1721                 1'b0: ;
1722                 1'bx: begin
1723                     Y = 'bx;
1724                     found_active_sel_bit = 'bx;
1725                 end
1726             endcase
1727     end
1728
1729 endmodule
```

yosys> help $tribuf

A tri-state buffer. This buffer conditionally drives the output with the value of the input based on the enable signal.

Simulation model (verilog)

Listing 9.45: simlib.v

```
1816 module \$tribuf (A, EN, Y);
1817
1818     parameter WIDTH = 0;
1819
1820     input [WIDTH-1:0] A;
```

(continues on next page)

(continued from previous page)

```
1821      input EN;
1822      output [WIDTH-1:0] Y;
1823
1824      assign Y = EN ? A : 'bz;
1825
1826  endmodule
```

9.1.4 Registers

SR-type latches are represented by *$sr* cells. These cells have input ports SET and CLR and an output port Q. They have the following parameters:

WIDTH
: The width of inputs SET and CLR and output Q.

SET_POLARITY
: The set input bits are active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

CLR_POLARITY
: The reset input bits are active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

Both set and reset inputs have separate bits for every output bit. When both the set and reset inputs of an *$sr* cell are active for a given bit index, the reset input takes precedence.

D-type flip-flops are represented by *$dff* cells. These cells have a clock port CLK, an input port D and an output port Q. The following parameters are available for *$dff* cells:

WIDTH
: The width of input D and output Q.

CLK_POLARITY
: Clock is active on the positive edge if this parameter has the value 1'b1 and on the negative edge if this parameter is 1'b0.

D-type flip-flops with asynchronous reset are represented by *$adff* cells. As the *$dff* cells they have CLK, D and Q ports. In addition they also have a single-bit ARST input port for the reset pin and the following additional two parameters:

ARST_POLARITY
: The asynchronous reset is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

ARST_VALUE
: The state of Q will be set to this value when the reset is active.

Usually these cells are generated by the *proc* pass using the information in the designs RTLIL::Process objects.

D-type flip-flops with synchronous reset are represented by *$sdff* cells. As the *$dff* cells they have CLK, D and Q ports. In addition they also have a single-bit SRST input port for the reset pin and the following additional two parameters:

SRST_POLARITY
: The synchronous reset is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

SRST_VALUE
: The state of Q will be set to this value when the reset is active.

Note that the *$adff* and *$sdff* cells can only be used when the reset value is constant.

D-type flip-flops with asynchronous load are represented by *$aldff* cells. As the *$dff* cells they have CLK, D and Q ports. In addition they also have a single-bit ALOAD input port for the async load enable pin, a AD input port with the same width as data for the async load data, and the following additional parameter:

ALOAD_POLARITY
: The asynchronous load is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

D-type flip-flops with asynchronous set and reset are represented by *$dffsr* cells. As the *$dff* cells they have CLK, D and Q ports. In addition they also have multi-bit SET and CLR input ports and the corresponding polarity parameters, like *$sr* cells. When both set and reset are active, the state and output is undefined.

D-type flip-flops with enable are represented by *$dffe*, *$adffe*, *$aldffe*, *$dffsre*, *$sdffe*, and *$sdffce* cells, which are enhanced variants of *$dff*, *$adff*, *$aldff*, *$dffsr*, *$sdff* (with reset over enable) and *$sdff* (with enable over reset) cells, respectively. They have the same ports and parameters as their base cell. In addition they also have a single-bit EN input port for the enable pin and the following parameter:

EN_POLARITY
: The enable input is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

D-type latches are represented by *$dlatch* cells. These cells have an enable port EN, an input port D, and an output port Q. The following parameters are available for *$dlatch* cells:

WIDTH
: The width of input D and output Q.

EN_POLARITY
: The enable input is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

The latch is transparent when the EN input is active.

D-type latches with reset are represented by *$adlatch* cells. In addition to *$dlatch* ports and parameters, they also have a single-bit ARST input port for the reset pin and the following additional parameters:

ARST_POLARITY
: The asynchronous reset is active-high if this parameter has the value 1'b1 and active-low if this parameter is 1'b0.

ARST_VALUE
: The state of Q will be set to this value when the reset is active.

D-type latches with set and reset are represented by *$dlatchsr* cells. In addition to *$dlatch* ports and parameters, they also have multi-bit SET and CLR input ports and the corresponding polarity parameters, like *$sr* cells.

```
yosys> help $adff

    Simulation model (verilog)
```

Listing 9.46: simlib.v

```
module \$adff (CLK, ARST, D, Q);

    parameter WIDTH = 0;
    parameter CLK_POLARITY = 1'b1;
    parameter ARST_POLARITY = 1'b1;
    parameter ARST_VALUE = 0;

    input CLK, ARST;
    input [WIDTH-1:0] D;
    output reg [WIDTH-1:0] Q;
    wire pos_clk = CLK == CLK_POLARITY;
    wire pos_arst = ARST == ARST_POLARITY;

    always @(posedge pos_clk, posedge pos_arst) begin
        if (pos_arst)
            Q <= ARST_VALUE;
        else
            Q <= D;
    end

endmodule
```

```
yosys> help $adffe
```

Simulation model (verilog)

Listing 9.47: simlib.v

```
module \$adffe (CLK, ARST, EN, D, Q);

    parameter WIDTH = 0;
    parameter CLK_POLARITY = 1'b1;
    parameter EN_POLARITY = 1'b1;
    parameter ARST_POLARITY = 1'b1;
    parameter ARST_VALUE = 0;

    input CLK, ARST, EN;
    input [WIDTH-1:0] D;
    output reg [WIDTH-1:0] Q;
    wire pos_clk = CLK == CLK_POLARITY;
    wire pos_arst = ARST == ARST_POLARITY;

    always @(posedge pos_clk, posedge pos_arst) begin
        if (pos_arst)
            Q <= ARST_VALUE;
        else if (EN == EN_POLARITY)
            Q <= D;
    end

endmodule
```

```
yosys> help $adlatch
```

Simulation model (verilog)

Listing 9.48: simlib.v

```
module \$adlatch (EN, ARST, D, Q);

    parameter WIDTH = 0;
    parameter EN_POLARITY = 1'b1;
    parameter ARST_POLARITY = 1'b1;
    parameter ARST_VALUE = 0;

    input EN, ARST;
    input [WIDTH-1:0] D;
    output reg [WIDTH-1:0] Q;

    always @* begin
        if (ARST == ARST_POLARITY)
            Q = ARST_VALUE;
        else if (EN == EN_POLARITY)
            Q = D;
    end

endmodule
```

yosys> help $aldff

Simulation model (verilog)

Listing 9.49: simlib.v

```
module \$aldff (CLK, ALOAD, AD, D, Q);

    parameter WIDTH = 0;
    parameter CLK_POLARITY = 1'b1;
    parameter ALOAD_POLARITY = 1'b1;

    input CLK, ALOAD;
    input [WIDTH-1:0] AD;
    input [WIDTH-1:0] D;
    output reg [WIDTH-1:0] Q;
    wire pos_clk = CLK == CLK_POLARITY;
    wire pos_aload = ALOAD == ALOAD_POLARITY;

    always @(posedge pos_clk, posedge pos_aload) begin
        if (pos_aload)
            Q <= AD;
        else
            Q <= D;
    end

endmodule
```

yosys> help $aldffe

Simulation model (verilog)

Listing 9.50: simlib.v

```
2532 module \$aldffe (CLK, ALOAD, AD, EN, D, Q);
2533
2534     parameter WIDTH = 0;
2535     parameter CLK_POLARITY = 1'b1;
2536     parameter EN_POLARITY = 1'b1;
2537     parameter ALOAD_POLARITY = 1'b1;
2538
2539     input CLK, ALOAD, EN;
2540     input [WIDTH-1:0] D;
2541     input [WIDTH-1:0] AD;
2542     output reg [WIDTH-1:0] Q;
2543     wire pos_clk = CLK == CLK_POLARITY;
2544     wire pos_aload = ALOAD == ALOAD_POLARITY;
2545
2546     always @(posedge pos_clk, posedge pos_aload) begin
2547         if (pos_aload)
2548             Q <= AD;
2549         else if (EN == EN_POLARITY)
2550             Q <= D;
2551     end
2552
2553 endmodule
```

```
yosys> help $dff

    Simulation model (verilog)
```

Listing 9.51: simlib.v

```
2323 module \$dff (CLK, D, Q);
2324
2325     parameter WIDTH = 0;
2326     parameter CLK_POLARITY = 1'b1;
2327
2328     input CLK;
2329     input [WIDTH-1:0] D;
2330     output reg [WIDTH-1:0] Q;
2331     wire pos_clk = CLK == CLK_POLARITY;
2332
2333     always @(posedge pos_clk) begin
2334         Q <= D;
2335     end
2336
2337 endmodule
```

```
yosys> help $dffe

    Simulation model (verilog)
```

Listing 9.52: simlib.v

```
module \$dffe (CLK, EN, D, Q);

    parameter WIDTH = 0;
    parameter CLK_POLARITY = 1'b1;
    parameter EN_POLARITY = 1'b1;

    input CLK, EN;
    input [WIDTH-1:0] D;
    output reg [WIDTH-1:0] Q;
    wire pos_clk = CLK == CLK_POLARITY;

    always @(posedge pos_clk) begin
        if (EN == EN_POLARITY) Q <= D;
    end

endmodule
```

yosys> help $dffsr

Simulation model (verilog)

Listing 9.53: simlib.v

```
module \$dffsr (CLK, SET, CLR, D, Q);

    parameter WIDTH = 0;
    parameter CLK_POLARITY = 1'b1;
    parameter SET_POLARITY = 1'b1;
    parameter CLR_POLARITY = 1'b1;

    input CLK;
    input [WIDTH-1:0] SET, CLR, D;
    output reg [WIDTH-1:0] Q;

    wire pos_clk = CLK == CLK_POLARITY;
    wire [WIDTH-1:0] pos_set = SET_POLARITY ? SET : ~SET;
    wire [WIDTH-1:0] pos_clr = CLR_POLARITY ? CLR : ~CLR;

    genvar i;
    generate
        for (i = 0; i < WIDTH; i = i+1) begin:bitslices
            always @(posedge pos_set[i], posedge pos_clr[i], posedge pos_clk)
                if (pos_clr[i])
                    Q[i] <= 0;
                else if (pos_set[i])
                    Q[i] <= 1;
                else
                    Q[i] <= D[i];
        end
    endgenerate

```

(continues on next page)

(continued from previous page)

```
endmodule
```

```
yosys> help $dffsre
```

Simulation model (verilog)

Listing 9.54: simlib.v

```
module \$dffsre (CLK, SET, CLR, EN, D, Q);

    parameter WIDTH = 0;
    parameter CLK_POLARITY = 1'b1;
    parameter SET_POLARITY = 1'b1;
    parameter CLR_POLARITY = 1'b1;
    parameter EN_POLARITY = 1'b1;

    input CLK, EN;
    input [WIDTH-1:0] SET, CLR, D;
    output reg [WIDTH-1:0] Q;

    wire pos_clk = CLK == CLK_POLARITY;
    wire [WIDTH-1:0] pos_set = SET_POLARITY ? SET : ~SET;
    wire [WIDTH-1:0] pos_clr = CLR_POLARITY ? CLR : ~CLR;

    genvar i;
    generate
        for (i = 0; i < WIDTH; i = i+1) begin:bitslices
            always @(posedge pos_set[i], posedge pos_clr[i], posedge pos_clk)
                if (pos_clr[i])
                    Q[i] <= 0;
                else if (pos_set[i])
                    Q[i] <= 1;
                else if (EN == EN_POLARITY)
                    Q[i] <= D[i];
        end
    endgenerate

endmodule
```

```
yosys> help $dlatch
```

Simulation model (verilog)

Listing 9.55: simlib.v

```
module \$dlatch (EN, D, Q);

    parameter WIDTH = 0;
    parameter EN_POLARITY = 1'b1;

    input EN;
    input [WIDTH-1:0] D;
    output reg [WIDTH-1:0] Q;
```

(continues on next page)

(continued from previous page)

```
2620
2621        always @* begin
2622            if (EN == EN_POLARITY)
2623                Q = D;
2624        end
2625
2626    endmodule
```

```
yosys> help $dlatchsr
```

Simulation model (verilog)

Listing 9.56: simlib.v

```
2655    module \$dlatchsr (EN, SET, CLR, D, Q);
2656
2657        parameter WIDTH = 0;
2658        parameter EN_POLARITY = 1'b1;
2659        parameter SET_POLARITY = 1'b1;
2660        parameter CLR_POLARITY = 1'b1;
2661
2662        input EN;
2663        input [WIDTH-1:0] SET, CLR, D;
2664        output reg [WIDTH-1:0] Q;
2665
2666        wire pos_en = EN == EN_POLARITY;
2667        wire [WIDTH-1:0] pos_set = SET_POLARITY ? SET : ~SET;
2668        wire [WIDTH-1:0] pos_clr = CLR_POLARITY ? CLR : ~CLR;
2669
2670        genvar i;
2671        generate
2672            for (i = 0; i < WIDTH; i = i+1) begin:bitslices
2673                always @*
2674                    if (pos_clr[i])
2675                        Q[i] = 0;
2676                    else if (pos_set[i])
2677                        Q[i] = 1;
2678                    else if (pos_en)
2679                        Q[i] = D[i];
2680            end
2681        endgenerate
2682
2683    endmodule
```

```
yosys> help $sdff
```

Simulation model (verilog)

Listing 9.57: simlib.v

```
2481    module \$sdff (CLK, SRST, D, Q);
2482
2483        parameter WIDTH = 0;
```

(continues on next page)

(continued from previous page)

```
2484     parameter CLK_POLARITY = 1'b1;
2485     parameter SRST_POLARITY = 1'b1;
2486     parameter SRST_VALUE = 0;
2487
2488     input CLK, SRST;
2489     input [WIDTH-1:0] D;
2490     output reg [WIDTH-1:0] Q;
2491     wire pos_clk = CLK == CLK_POLARITY;
2492     wire pos_srst = SRST == SRST_POLARITY;
2493
2494     always @(posedge pos_clk) begin
2495         if (pos_srst)
2496             Q <= SRST_VALUE;
2497         else
2498             Q <= D;
2499     end
2500
2501 endmodule
```

```
yosys> help $sdffce

    Simulation model (verilog)
```

Listing 9.58: simlib.v

```
2584 module \$sdffce (CLK, SRST, EN, D, Q);
2585
2586     parameter WIDTH = 0;
2587     parameter CLK_POLARITY = 1'b1;
2588     parameter EN_POLARITY = 1'b1;
2589     parameter SRST_POLARITY = 1'b1;
2590     parameter SRST_VALUE = 0;
2591
2592     input CLK, SRST, EN;
2593     input [WIDTH-1:0] D;
2594     output reg [WIDTH-1:0] Q;
2595     wire pos_clk = CLK == CLK_POLARITY;
2596     wire pos_srst = SRST == SRST_POLARITY;
2597
2598     always @(posedge pos_clk) begin
2599         if (EN == EN_POLARITY) begin
2600             if (pos_srst)
2601                 Q <= SRST_VALUE;
2602             else
2603                 Q <= D;
2604         end
2605     end
2606
2607 endmodule
```

```
yosys> help $sdffe

    Simulation model (verilog)
```

Listing 9.59: simlib.v

```
2558 module \$sdffe (CLK, SRST, EN, D, Q);
2559
2560     parameter WIDTH = 0;
2561     parameter CLK_POLARITY = 1'b1;
2562     parameter EN_POLARITY = 1'b1;
2563     parameter SRST_POLARITY = 1'b1;
2564     parameter SRST_VALUE = 0;
2565
2566     input CLK, SRST, EN;
2567     input [WIDTH-1:0] D;
2568     output reg [WIDTH-1:0] Q;
2569     wire pos_clk = CLK == CLK_POLARITY;
2570     wire pos_srst = SRST == SRST_POLARITY;
2571
2572     always @(posedge pos_clk) begin
2573         if (pos_srst)
2574             Q <= SRST_VALUE;
2575         else if (EN == EN_POLARITY)
2576             Q <= D;
2577     end
2578
2579 endmodule
```

```
yosys> help $sr

    Simulation model (verilog)
```

Listing 9.60: simlib.v

```
2273 module \$sr (SET, CLR, Q);
2274
2275     parameter WIDTH = 0;
2276     parameter SET_POLARITY = 1'b1;
2277     parameter CLR_POLARITY = 1'b1;
2278
2279     input [WIDTH-1:0] SET, CLR;
2280     output reg [WIDTH-1:0] Q;
2281
2282     wire [WIDTH-1:0] pos_set = SET_POLARITY ? SET : ~SET;
2283     wire [WIDTH-1:0] pos_clr = CLR_POLARITY ? CLR : ~CLR;
2284
2285     genvar i;
2286     generate
2287         for (i = 0; i < WIDTH; i = i+1) begin:bitslices
2288             always @*
2289                 if (pos_clr[i])
2290                     Q[i] <= 0;
2291                 else if (pos_set[i])
2292                     Q[i] <= 1;
2293         end
2294     endgenerate
```

(continues on next page)

(continued from previous page)

```
2295
2296 endmodule
```

9.1.5 Memories

Memories are either represented using `RTLIL::Memory` objects, *`$memrd_v2`*, *`$memwr_v2`*, and *`$meminit_v2`* cells, or by *`$mem_v2`* cells alone.

In the first alternative the `RTLIL::Memory` objects hold the general metadata for the memory (bit width, size in number of words, etc.) and for each port a *`$memrd_v2`* (read port) or *`$memwr_v2`* (write port) cell is created. Having individual cells for read and write ports has the advantage that they can be consolidated using resource sharing passes. In some cases this drastically reduces the number of required ports on the memory cell. In this alternative, memory initialization data is represented by *`$meminit_v2`* cells, which allow delaying constant folding for initialization addresses and data until after the frontend finishes.

The *`$memrd_v2`* cells have a clock input `CLK`, an enable input `EN`, an address input `ADDR`, a data output `DATA`, an asynchronous reset input `ARST`, and a synchronous reset input `SRST`. They also have the following parameters:

`MEMID`
: The name of the `RTLIL::Memory` object that is associated with this read port.

`ABITS`
: The number of address bits (width of the `ADDR` input port).

`WIDTH`
: The number of data bits (width of the `DATA` output port). Note that this may be a power-of-two multiple of the underlying memory's width – such ports are called wide ports and access an aligned group of cells at once. In this case, the corresponding low bits of `ADDR` must be tied to 0.

`CLK_ENABLE`
: When this parameter is non-zero, the clock is used. Otherwise this read port is asynchronous and the `CLK` input is not used.

`CLK_POLARITY`
: Clock is active on the positive edge if this parameter has the value `1'b1` and on the negative edge if this parameter is `1'b0`.

`TRANSPARENCY_MASK`
: This parameter is a bitmask of write ports that this read port is transparent with. The bits of this parameter are indexed by the write port's `PORTID` parameter. Transparency can only be enabled between synchronous ports sharing a clock domain. When transparency is enabled for a given port pair, a read and write to the same address in the same cycle will return the new value. Otherwise the old value is returned.

`COLLISION_X_MASK`
: This parameter is a bitmask of write ports that have undefined collision behavior with this port. The bits of this parameter are indexed by the write port's `PORTID` parameter. This behavior can only be enabled between synchronous ports sharing a clock domain. When undefined collision is enabled for a given port pair, a read and write to the same address in the same cycle will return the undefined (all-X) value.This option is exclusive (for a given port pair) with the transparency option.

`ARST_VALUE`
: Whenever the `ARST` input is asserted, the data output will be reset to this value. Only used for synchronous ports.

`SRST_VALUE`
: Whenever the `SRST` input is synchronously asserted, the data output will be reset to this value. Only

used for synchronous ports.

`INIT_VALUE`
: The initial value of the data output, for synchronous ports.

`CE_OVER_SRST`
: If this parameter is non-zero, the `SRST` input is only recognized when `EN` is true. Otherwise, `SRST` is recognized regardless of `EN`.

The *$memwr_v2* cells have a clock input `CLK`, an enable input `EN` (one enable bit for each data bit), an address input `ADDR` and a data input `DATA`. They also have the following parameters:

`MEMID`
: The name of the `RTLIL::Memory` object that is associated with this write port.

`ABITS`
: The number of address bits (width of the `ADDR` input port).

`WIDTH`
: The number of data bits (width of the `DATA` output port). Like with *$memrd_v2* cells, the width is allowed to be any power-of-two multiple of memory width, with the corresponding restriction on address.

`CLK_ENABLE`
: When this parameter is non-zero, the clock is used. Otherwise this write port is asynchronous and the `CLK` input is not used.

`CLK_POLARITY`
: Clock is active on positive edge if this parameter has the value `1'b1` and on the negative edge if this parameter is `1'b0`.

`PORTID`
: An identifier for this write port, used to index write port bit mask parameters.

`PRIORITY_MASK`
: This parameter is a bitmask of write ports that this write port has priority over in case of writing to the same address. The bits of this parameter are indexed by the other write port's `PORTID` parameter. Write ports can only have priority over write ports with lower port ID. When two ports write to the same address and neither has priority over the other, the result is undefined. Priority can only be set between two synchronous ports sharing the same clock domain.

The *$meminit_v2* cells have an address input `ADDR`, a data input `DATA`, with the width of the `DATA` port equal to `WIDTH` parameter times `WORDS` parameter, and a bit enable mask input `EN` with width equal to `WIDTH` parameter. All three of the inputs must resolve to a constant for synthesis to succeed.

`MEMID`
: The name of the `RTLIL::Memory` object that is associated with this initialization cell.

`ABITS`
: The number of address bits (width of the `ADDR` input port).

`WIDTH`
: The number of data bits per memory location.

`WORDS`
: The number of consecutive memory locations initialized by this cell.

`PRIORITY`
: The cell with the higher integer value in this parameter wins an initialization conflict.

The HDL frontend models a memory using `RTLIL::Memory` objects and asynchronous *$memrd_v2* and *$memwr_v2* cells. The *memory* pass (i.e. its various sub-passes) migrates *$dff* cells into the *$memrd_v2*

and *$memwr_v2* cells making them synchronous, then converts them to a single *$mem_v2* cell and (optionally) maps this cell type to *$dff* cells for the individual words and multiplexer-based address decoders for the read and write interfaces. When the last step is disabled or not possible, a *$mem_v2* cell is left in the design.

The *$mem_v2* cell provides the following parameters:

`MEMID`
: The name of the original `RTLIL::Memory` object that became this *$mem_v2* cell.

`SIZE`
: The number of words in the memory.

`ABITS`
: The number of address bits.

`WIDTH`
: The number of data bits per word.

`INIT`
: The initial memory contents.

`RD_PORTS`
: The number of read ports on this memory cell.

`RD_WIDE_CONTINUATION`
: This parameter is `RD_PORTS` bits wide, containing a bitmask of "wide continuation" read ports. Such ports are used to represent the extra data bits of wide ports in the combined cell, and must have all control signals identical with the preceding port, except for address, which must have the proper sub-cell address encoded in the low bits.

`RD_CLK_ENABLE`
: This parameter is `RD_PORTS` bits wide, containing a clock enable bit for each read port.

`RD_CLK_POLARITY`
: This parameter is `RD_PORTS` bits wide, containing a clock polarity bit for each read port.

`RD_TRANSPARENCY_MASK`
: This parameter is `RD_PORTS*WR_PORTS` bits wide, containing a concatenation of all `TRANSPARENCY_MASK` values of the original *$memrd_v2* cells.

`RD_COLLISION_X_MASK`
: This parameter is `RD_PORTS*WR_PORTS` bits wide, containing a concatenation of all `COLLISION_X_MASK` values of the original *$memrd_v2* cells.

`RD_CE_OVER_SRST`
: This parameter is `RD_PORTS` bits wide, determining relative synchronous reset and enable priority for each read port.

`RD_INIT_VALUE`
: This parameter is `RD_PORTS*WIDTH` bits wide, containing the initial value for each synchronous read port.

`RD_ARST_VALUE`
: This parameter is `RD_PORTS*WIDTH` bits wide, containing the asynchronous reset value for each synchronous read port.

`RD_SRST_VALUE`
: This parameter is `RD_PORTS*WIDTH` bits wide, containing the synchronous reset value for each synchronous read port.

`WR_PORTS`
: The number of write ports on this memory cell.

`WR_WIDE_CONTINUATION`
: This parameter is `WR_PORTS` bits wide, containing a bitmask of "wide continuation" write ports.

`WR_CLK_ENABLE`
: This parameter is `WR_PORTS` bits wide, containing a clock enable bit for each write port.

`WR_CLK_POLARITY`
: This parameter is `WR_PORTS` bits wide, containing a clock polarity bit for each write port.

`WR_PRIORITY_MASK`
: This parameter is `WR_PORTS*WR_PORTS` bits wide, containing a concatenation of all `PRIORITY_MASK` values of the original *$memwr_v2* cells.

The *$mem_v2* cell has the following ports:

`RD_CLK`
: This input is `RD_PORTS` bits wide, containing all clock signals for the read ports.

`RD_EN`
: This input is `RD_PORTS` bits wide, containing all enable signals for the read ports.

`RD_ADDR`
: This input is `RD_PORTS*ABITS` bits wide, containing all address signals for the read ports.

`RD_DATA`
: This output is `RD_PORTS*WIDTH` bits wide, containing all data signals for the read ports.

`RD_ARST`
: This input is `RD_PORTS` bits wide, containing all asynchronous reset signals for the read ports.

`RD_SRST`
: This input is `RD_PORTS` bits wide, containing all synchronous reset signals for the read ports.

`WR_CLK`
: This input is `WR_PORTS` bits wide, containing all clock signals for the write ports.

`WR_EN`
: This input is `WR_PORTS*WIDTH` bits wide, containing all enable signals for the write ports.

`WR_ADDR`
: This input is `WR_PORTS*ABITS` bits wide, containing all address signals for the write ports.

`WR_DATA`
: This input is `WR_PORTS*WIDTH` bits wide, containing all data signals for the write ports.

The *memory_collect* pass can be used to convert discrete *$memrd_v2*, *$memwr_v2*, and *$meminit_v2* cells belonging to the same memory to a single *$mem_v2* cell, whereas the *memory_unpack* pass performs the inverse operation. The *memory_dff* pass can combine asynchronous memory ports that are fed by or feeding registers into synchronous memory ports. The *memory_bram* pass can be used to recognize *$mem_v2* cells that can be implemented with a block RAM resource on an FPGA. The *memory_map* pass can be used to implement *$mem_v2* cells as basic logic: word-wide DFFs and address decoders.

```
yosys> help $mem

    Simulation model (verilog)
```

Listing 9.61: simlib.v

```
module \$mem (RD_CLK, RD_EN, RD_ADDR, RD_DATA, WR_CLK, WR_EN, WR_ADDR, WR_DATA);

    parameter MEMID = "";
    parameter signed SIZE = 4;
    parameter signed OFFSET = 0;
    parameter signed ABITS = 2;
    parameter signed WIDTH = 8;
    parameter signed INIT = 1'bx;

    parameter signed RD_PORTS = 1;
    parameter RD_CLK_ENABLE = 1'b1;
    parameter RD_CLK_POLARITY = 1'b1;
    parameter RD_TRANSPARENT = 1'b1;

    parameter signed WR_PORTS = 1;
    parameter WR_CLK_ENABLE = 1'b1;
    parameter WR_CLK_POLARITY = 1'b1;

    input [RD_PORTS-1:0] RD_CLK;
    input [RD_PORTS-1:0] RD_EN;
    input [RD_PORTS*ABITS-1:0] RD_ADDR;
    output reg [RD_PORTS*WIDTH-1:0] RD_DATA;

    input [WR_PORTS-1:0] WR_CLK;
    input [WR_PORTS*WIDTH-1:0] WR_EN;
    input [WR_PORTS*ABITS-1:0] WR_ADDR;
    input [WR_PORTS*WIDTH-1:0] WR_DATA;

    reg [WIDTH-1:0] memory [SIZE-1:0];

    integer i, j;
    reg [WR_PORTS-1:0] LAST_WR_CLK;
    reg [RD_PORTS-1:0] LAST_RD_CLK;

    function port_active;
        input clk_enable;
        input clk_polarity;
        input last_clk;
        input this_clk;
        begin
            casez ({clk_enable, clk_polarity, last_clk, this_clk})
                4'b0???: port_active = 1;
                4'b1101: port_active = 1;
                4'b1010: port_active = 1;
                default: port_active = 0;
            endcase
        end
    endfunction

    initial begin
        for (i = 0; i < SIZE; i = i+1)
```

(continues on next page)

(continued from previous page)

```
2993            memory[i] = INIT >>> (i*WIDTH);
2994    end
2995
2996    always @(RD_CLK, RD_ADDR, RD_DATA, WR_CLK, WR_EN, WR_ADDR, WR_DATA) begin
2997    `ifdef SIMLIB_MEMDELAY
2998        #`SIMLIB_MEMDELAY;
2999    `endif
3000        for (i = 0; i < RD_PORTS; i = i+1) begin
3001            if (!RD_TRANSPARENT[i] && RD_CLK_ENABLE[i] && RD_EN[i] && port_
    ↪active(RD_CLK_ENABLE[i], RD_CLK_POLARITY[i], LAST_RD_CLK[i], RD_CLK[i])) begin
3002                // $display("Read from %s: addr=%b data=%b", MEMID, RD_ADDR[i*ABITS␣
    ↪+: ABITS],  memory[RD_ADDR[i*ABITS +: ABITS] - OFFSET]);
3003                RD_DATA[i*WIDTH +: WIDTH] <= memory[RD_ADDR[i*ABITS +: ABITS] -␣
    ↪OFFSET];
3004            end
3005        end
3006
3007        for (i = 0; i < WR_PORTS; i = i+1) begin
3008            if (port_active(WR_CLK_ENABLE[i], WR_CLK_POLARITY[i], LAST_WR_CLK[i],␣
    ↪WR_CLK[i]))
3009                for (j = 0; j < WIDTH; j = j+1)
3010                    if (WR_EN[i*WIDTH+j]) begin
3011                        // $display("Write to %s: addr=%b data=%b", MEMID, WR_
    ↪ADDR[i*ABITS +: ABITS], WR_DATA[i*WIDTH+j]);
3012                        memory[WR_ADDR[i*ABITS +: ABITS] - OFFSET][j] = WR_
    ↪DATA[i*WIDTH+j];
3013                    end
3014        end
3015
3016        for (i = 0; i < RD_PORTS; i = i+1) begin
3017            if ((RD_TRANSPARENT[i] || !RD_CLK_ENABLE[i]) && port_active(RD_CLK_
    ↪ENABLE[i], RD_CLK_POLARITY[i], LAST_RD_CLK[i], RD_CLK[i])) begin
3018                // $display("Transparent read from %s: addr=%b data=%b", MEMID, RD_
    ↪ADDR[i*ABITS +: ABITS],  memory[RD_ADDR[i*ABITS +: ABITS] - OFFSET]);
3019                RD_DATA[i*WIDTH +: WIDTH] <= memory[RD_ADDR[i*ABITS +: ABITS] -␣
    ↪OFFSET];
3020            end
3021        end
3022
3023        LAST_RD_CLK <= RD_CLK;
3024        LAST_WR_CLK <= WR_CLK;
3025    end
3026
3027 endmodule
```

```
yosys> help $mem_v2

    Simulation model (verilog)
```

Listing 9.62: simlib.v

```
module \$mem_v2 (RD_CLK, RD_EN, RD_ARST, RD_SRST, RD_ADDR, RD_DATA, WR_CLK, WR_EN,␣
↪WR_ADDR, WR_DATA);

    parameter MEMID = "";
    parameter signed SIZE = 4;
    parameter signed OFFSET = 0;
    parameter signed ABITS = 2;
    parameter signed WIDTH = 8;
    parameter signed INIT = 1'bx;

    parameter signed RD_PORTS = 1;
    parameter RD_CLK_ENABLE = 1'b1;
    parameter RD_CLK_POLARITY = 1'b1;
    parameter RD_TRANSPARENCY_MASK = 1'b0;
    parameter RD_COLLISION_X_MASK = 1'b0;
    parameter RD_WIDE_CONTINUATION = 1'b0;
    parameter RD_CE_OVER_SRST = 1'b0;
    parameter RD_ARST_VALUE = 1'b0;
    parameter RD_SRST_VALUE = 1'b0;
    parameter RD_INIT_VALUE = 1'b0;

    parameter signed WR_PORTS = 1;
    parameter WR_CLK_ENABLE = 1'b1;
    parameter WR_CLK_POLARITY = 1'b1;
    parameter WR_PRIORITY_MASK = 1'b0;
    parameter WR_WIDE_CONTINUATION = 1'b0;

    input [RD_PORTS-1:0] RD_CLK;
    input [RD_PORTS-1:0] RD_EN;
    input [RD_PORTS-1:0] RD_ARST;
    input [RD_PORTS-1:0] RD_SRST;
    input [RD_PORTS*ABITS-1:0] RD_ADDR;
    output reg [RD_PORTS*WIDTH-1:0] RD_DATA;

    input [WR_PORTS-1:0] WR_CLK;
    input [WR_PORTS*WIDTH-1:0] WR_EN;
    input [WR_PORTS*ABITS-1:0] WR_ADDR;
    input [WR_PORTS*WIDTH-1:0] WR_DATA;

    reg [WIDTH-1:0] memory [SIZE-1:0];

    integer i, j, k;
    reg [WR_PORTS-1:0] LAST_WR_CLK;
    reg [RD_PORTS-1:0] LAST_RD_CLK;

    function port_active;
        input clk_enable;
        input clk_polarity;
        input last_clk;
        input this_clk;
        begin
```

(continues on next page)

(continued from previous page)

```
3081            casez ({clk_enable, clk_polarity, last_clk, this_clk})
3082                4'b0???: port_active = 1;
3083                4'b1101: port_active = 1;
3084                4'b1010: port_active = 1;
3085                default: port_active = 0;
3086            endcase
3087        end
3088    endfunction
3089
3090    initial begin
3091        for (i = 0; i < SIZE; i = i+1)
3092            memory[i] = INIT >>> (i*WIDTH);
3093        RD_DATA = RD_INIT_VALUE;
3094    end
3095
3096    always @(RD_CLK, RD_ARST, RD_ADDR, RD_DATA, WR_CLK, WR_EN, WR_ADDR, WR_DATA)␣
↪begin
3097    `ifdef SIMLIB_MEMDELAY
3098        #`SIMLIB_MEMDELAY;
3099    `endif
3100        for (i = 0; i < RD_PORTS; i = i+1) begin
3101            if (RD_CLK_ENABLE[i] && RD_EN[i] && port_active(RD_CLK_ENABLE[i], RD_
↪CLK_POLARITY[i], LAST_RD_CLK[i], RD_CLK[i])) begin
3102                // $display("Read from %s: addr=%b data=%b", MEMID, RD_ADDR[i*ABITS␣
↪+: ABITS],  memory[RD_ADDR[i*ABITS +: ABITS] - OFFSET]);
3103                RD_DATA[i*WIDTH +: WIDTH] <= memory[RD_ADDR[i*ABITS +: ABITS] -␣
↪OFFSET];
3104
3105                for (j = 0; j < WR_PORTS; j = j+1) begin
3106                    if (RD_TRANSPARENCY_MASK[i*WR_PORTS + j] && port_active(WR_CLK_
↪ENABLE[j], WR_CLK_POLARITY[j], LAST_WR_CLK[j], WR_CLK[j]) && RD_ADDR[i*ABITS +:␣
↪ABITS] == WR_ADDR[j*ABITS +: ABITS])
3107                        for (k = 0; k < WIDTH; k = k+1)
3108                            if (WR_EN[j*WIDTH+k])
3109                                RD_DATA[i*WIDTH+k] <= WR_DATA[j*WIDTH+k];
3110                    if (RD_COLLISION_X_MASK[i*WR_PORTS + j] && port_active(WR_CLK_
↪ENABLE[j], WR_CLK_POLARITY[j], LAST_WR_CLK[j], WR_CLK[j]) && RD_ADDR[i*ABITS +:␣
↪ABITS] == WR_ADDR[j*ABITS +: ABITS])
3111                        for (k = 0; k < WIDTH; k = k+1)
3112                            if (WR_EN[j*WIDTH+k])
3113                                RD_DATA[i*WIDTH+k] <= 1'bx;
3114                end
3115            end
3116        end
3117
3118        for (i = 0; i < WR_PORTS; i = i+1) begin
3119            if (port_active(WR_CLK_ENABLE[i], WR_CLK_POLARITY[i], LAST_WR_CLK[i],␣
↪WR_CLK[i]))
3120                for (j = 0; j < WIDTH; j = j+1)
3121                    if (WR_EN[i*WIDTH+j]) begin
3122                        // $display("Write to %s: addr=%b data=%b", MEMID, WR_
↪ADDR[i*ABITS +: ABITS], WR_DATA[i*WIDTH+j]);
```

(continues on next page)

(continued from previous page)

```
3123                        memory[WR_ADDR[i*ABITS +: ABITS] - OFFSET][j] = WR_
    ↪DATA[i*WIDTH+j];
3124                    end
3125        end
3126
3127        for (i = 0; i < RD_PORTS; i = i+1) begin
3128            if (!RD_CLK_ENABLE[i]) begin
3129                // $display("Combinatorial read from %s: addr=%b data=%b", MEMID,␣
    ↪RD_ADDR[i*ABITS +: ABITS],  memory[RD_ADDR[i*ABITS +: ABITS] - OFFSET]);
3130                RD_DATA[i*WIDTH +: WIDTH] <= memory[RD_ADDR[i*ABITS +: ABITS] -␣
    ↪OFFSET];
3131            end
3132        end
3133
3134        for (i = 0; i < RD_PORTS; i = i+1) begin
3135            if (RD_SRST[i] && port_active(RD_CLK_ENABLE[i], RD_CLK_POLARITY[i],␣
    ↪LAST_RD_CLK[i], RD_CLK[i]) && (RD_EN[i] || !RD_CE_OVER_SRST[i]))
3136                RD_DATA[i*WIDTH +: WIDTH] <= RD_SRST_VALUE[i*WIDTH +: WIDTH];
3137            if (RD_ARST[i])
3138                RD_DATA[i*WIDTH +: WIDTH] <= RD_ARST_VALUE[i*WIDTH +: WIDTH];
3139        end
3140
3141        LAST_RD_CLK <= RD_CLK;
3142        LAST_WR_CLK <= WR_CLK;
3143    end
3144
3145 endmodule
```

```
yosys> help $meminit

    Simulation model (verilog)
```

Listing 9.63: simlib.v

```
2893 module \$meminit (ADDR, DATA);
2894
2895     parameter MEMID = "";
2896     parameter ABITS = 8;
2897     parameter WIDTH = 8;
2898     parameter WORDS = 1;
2899
2900     parameter PRIORITY = 0;
2901
2902     input [ABITS-1:0] ADDR;
2903     input [WORDS*WIDTH-1:0] DATA;
2904
2905     initial begin
2906         if (MEMID != "") begin
2907             $display("ERROR: Found non-simulatable instance of $meminit!");
2908             $finish;
2909         end
2910     end
```

(continues on next page)

(continued from previous page)

```
2911
2912 endmodule
```

```
yosys> help $meminit_v2
```

```
    Simulation model (verilog)
```

Listing 9.64: simlib.v

```
2917 module \$meminit_v2 (ADDR, DATA, EN);
2918
2919     parameter MEMID = "";
2920     parameter ABITS = 8;
2921     parameter WIDTH = 8;
2922     parameter WORDS = 1;
2923
2924     parameter PRIORITY = 0;
2925
2926     input [ABITS-1:0] ADDR;
2927     input [WORDS*WIDTH-1:0] DATA;
2928     input [WIDTH-1:0] EN;
2929
2930     initial begin
2931         if (MEMID != "") begin
2932             $display("ERROR: Found non-simulatable instance of $meminit_v2!");
2933             $finish;
2934         end
2935     end
2936
2937 endmodule
```

```
yosys> help $memrd
```

```
    Simulation model (verilog)
```

Listing 9.65: simlib.v

```
2784 module \$memrd (CLK, EN, ADDR, DATA);
2785
2786     parameter MEMID = "";
2787     parameter ABITS = 8;
2788     parameter WIDTH = 8;
2789
2790     parameter CLK_ENABLE = 0;
2791     parameter CLK_POLARITY = 0;
2792     parameter TRANSPARENT = 0;
2793
2794     input CLK, EN;
2795     input [ABITS-1:0] ADDR;
2796     output [WIDTH-1:0] DATA;
2797
2798     initial begin
2799         if (MEMID != "") begin
```

(continues on next page)

(continued from previous page)

```
            $display("ERROR: Found non-simulatable instance of $memrd!");
            $finish;
        end
    end

endmodule
```

yosys> help $memrd_v2

Simulation model (verilog)

Listing 9.66: simlib.v

```
module \$memrd_v2 (CLK, EN, ARST, SRST, ADDR, DATA);

    parameter MEMID = "";
    parameter ABITS = 8;
    parameter WIDTH = 8;

    parameter CLK_ENABLE = 0;
    parameter CLK_POLARITY = 0;
    parameter TRANSPARENCY_MASK = 0;
    parameter COLLISION_X_MASK = 0;
    parameter ARST_VALUE = 0;
    parameter SRST_VALUE = 0;
    parameter INIT_VALUE = 0;
    parameter CE_OVER_SRST = 0;

    input CLK, EN, ARST, SRST;
    input [ABITS-1:0] ADDR;
    output [WIDTH-1:0] DATA;

    initial begin
        if (MEMID != "") begin
            $display("ERROR: Found non-simulatable instance of $memrd_v2!");
            $finish;
        end
    end

endmodule
```

yosys> help $memwr

Simulation model (verilog)

Listing 9.67: simlib.v

```
module \$memwr (CLK, EN, ADDR, DATA);

    parameter MEMID = "";
    parameter ABITS = 8;
    parameter WIDTH = 8;

```

(continues on next page)

(continued from previous page)

```
2846     parameter CLK_ENABLE = 0;
2847     parameter CLK_POLARITY = 0;
2848     parameter PRIORITY = 0;
2849
2850     input CLK;
2851     input [WIDTH-1:0] EN;
2852     input [ABITS-1:0] ADDR;
2853     input [WIDTH-1:0] DATA;
2854
2855     initial begin
2856         if (MEMID != "") begin
2857             $display("ERROR: Found non-simulatable instance of $memwr!");
2858             $finish;
2859         end
2860     end
2861
2862 endmodule
```

```
yosys> help $memwr_v2

    Simulation model (verilog)
```

Listing 9.68: simlib.v

```
2865 module \$memwr_v2 (CLK, EN, ADDR, DATA);
2866
2867     parameter MEMID = "";
2868     parameter ABITS = 8;
2869     parameter WIDTH = 8;
2870
2871     parameter CLK_ENABLE = 0;
2872     parameter CLK_POLARITY = 0;
2873     parameter PORTID = 0;
2874     parameter PRIORITY_MASK = 0;
2875
2876     input CLK;
2877     input [WIDTH-1:0] EN;
2878     input [ABITS-1:0] ADDR;
2879     input [WIDTH-1:0] DATA;
2880
2881     initial begin
2882         if (MEMID != "") begin
2883             $display("ERROR: Found non-simulatable instance of $memwr_v2!");
2884             $finish;
2885         end
2886     end
2887
2888 endmodule
```

9.1.6 Finite state machines

Todo

Describe *$fsm* cell

```
yosys> help $fsm

    Simulation model (verilog)
```

Listing 9.69: simlib.v

```
2689 module \$fsm (CLK, ARST, CTRL_IN, CTRL_OUT);
2690
2691     parameter NAME = "";
2692
2693     parameter CLK_POLARITY = 1'b1;
2694     parameter ARST_POLARITY = 1'b1;
2695
2696     parameter CTRL_IN_WIDTH = 1;
2697     parameter CTRL_OUT_WIDTH = 1;
2698
2699     parameter STATE_BITS = 1;
2700     parameter STATE_NUM = 1;
2701     parameter STATE_NUM_LOG2 = 1;
2702     parameter STATE_RST = 0;
2703     parameter STATE_TABLE = 1'b0;
2704
2705     parameter TRANS_NUM = 1;
2706     parameter TRANS_TABLE = 4'b0x0x;
2707
2708     input CLK, ARST;
2709     input [CTRL_IN_WIDTH-1:0] CTRL_IN;
2710     output reg [CTRL_OUT_WIDTH-1:0] CTRL_OUT;
2711
2712     wire pos_clk = CLK == CLK_POLARITY;
2713     wire pos_arst = ARST == ARST_POLARITY;
2714
2715     reg [STATE_BITS-1:0] state;
2716     reg [STATE_BITS-1:0] state_tmp;
2717     reg [STATE_BITS-1:0] next_state;
2718
2719     reg [STATE_BITS-1:0] tr_state_in;
2720     reg [STATE_BITS-1:0] tr_state_out;
2721     reg [CTRL_IN_WIDTH-1:0] tr_ctrl_in;
2722     reg [CTRL_OUT_WIDTH-1:0] tr_ctrl_out;
2723
2724     integer i;
2725
2726     task tr_fetch;
2727         input [31:0] tr_num;
```

(continues on next page)

(continued from previous page)

```
2728        reg [31:0] tr_pos;
2729        reg [STATE_NUM_LOG2-1:0] state_num;
2730        begin
2731            tr_pos = (2*STATE_NUM_LOG2+CTRL_IN_WIDTH+CTRL_OUT_WIDTH)*tr_num;
2732            tr_ctrl_out = TRANS_TABLE >> tr_pos;
2733            tr_pos = tr_pos + CTRL_OUT_WIDTH;
2734            state_num = TRANS_TABLE >> tr_pos;
2735            tr_state_out = STATE_TABLE >> (STATE_BITS*state_num);
2736            tr_pos = tr_pos + STATE_NUM_LOG2;
2737            tr_ctrl_in = TRANS_TABLE >> tr_pos;
2738            tr_pos = tr_pos + CTRL_IN_WIDTH;
2739            state_num = TRANS_TABLE >> tr_pos;
2740            tr_state_in = STATE_TABLE >> (STATE_BITS*state_num);
2741            tr_pos = tr_pos + STATE_NUM_LOG2;
2742        end
2743    endtask
2744
2745    always @(posedge pos_clk, posedge pos_arst) begin
2746        if (pos_arst) begin
2747            state_tmp = STATE_TABLE[STATE_BITS*(STATE_RST+1)-1:STATE_BITS*STATE_
    ↪RST];
2748            for (i = 0; i < STATE_BITS; i = i+1)
2749                if (state_tmp[i] === 1'bz)
2750                    state_tmp[i] = 0;
2751            state <= state_tmp;
2752        end else begin
2753            state_tmp = next_state;
2754            for (i = 0; i < STATE_BITS; i = i+1)
2755                if (state_tmp[i] === 1'bz)
2756                    state_tmp[i] = 0;
2757            state <= state_tmp;
2758        end
2759    end
2760
2761    always @(state, CTRL_IN) begin
2762        next_state <= STATE_TABLE[STATE_BITS*(STATE_RST+1)-1:STATE_BITS*STATE_RST];
2763        CTRL_OUT <= 'bx;
2764        // $display("---");
2765        // $display("Q: %b %b", state, CTRL_IN);
2766        for (i = 0; i < TRANS_NUM; i = i+1) begin
2767            tr_fetch(i);
2768            // $display("T: %b %b -> %b %b [%d]", tr_state_in, tr_ctrl_in, tr_state_
    ↪out, tr_ctrl_out, i);
2769            casez ({state, CTRL_IN})
2770                {tr_state_in, tr_ctrl_in}: begin
2771                    // $display("-> %b %b <-   MATCH", state, CTRL_IN);
2772                    {next_state, CTRL_OUT} <= {tr_state_out, tr_ctrl_out};
2773                end
2774            endcase
2775        end
2776    end
2777
```

(continues on next page)

(continued from previous page)

```
endmodule
```

9.1.7 Coarse arithmetics

Todo

Add information about *$alu*, *$fa*, *$macc_v2*, and *$lcu* cells.

The *$macc* cell type represents a generalized multiply and accumulate operation. The cell is purely combinational. It outputs the result of summing up a sequence of products and other injected summands.

```
Y = 0 +- a0factor1 * a0factor2 +- a1factor1 * a1factor2 +- ...
     + B[0] + B[1] + ...
```

The A port consists of concatenated pairs of multiplier inputs ("factors"). A zero length factor2 acts as a constant 1, turning factor1 into a simple summand.

In this pseudocode, `u(foo)` means an unsigned int that's foo bits long.

```
struct A {
   u(CONFIG.mul_info[0].factor1_len) a0factor1;
   u(CONFIG.mul_info[0].factor2_len) a0factor2;
   u(CONFIG.mul_info[1].factor1_len) a1factor1;
   u(CONFIG.mul_info[1].factor2_len) a1factor2;
   ...
};
```

The cell's `CONFIG` parameter determines the layout of cell port `A`. The CONFIG parameter carries the following information:

```
struct CONFIG {
   u4 num_bits;
   struct mul_info {
      bool is_signed;
      bool is_subtract;
      u(num_bits) factor1_len;
      u(num_bits) factor2_len;
   }[num_ports];
};
```

B is an array of concatenated 1-bit-wide unsigned integers to also be summed up.

```
yosys> help $alu
```

Arithmetic logic unit

A building block supporting both binary addition/subtraction operations, and indirectly, comparison operations. Typically created by the `alumacc` pass, which transforms: *$add*, *$sub*, *$lt*, *$le*, *$ge*, *$gt*, *$eq*, *$eqx*, *$ne*, *$nex* cells into this *$alu* cell.

Properties

is_evaluable

`Simulation model (verilog)`

Listing 9.70: simlib.v

```
665 module \$alu (A, B, CI, BI, X, Y, CO);
666
667     parameter A_SIGNED = 0;
668     parameter B_SIGNED = 0;
669     parameter A_WIDTH = 1;
670     parameter B_WIDTH = 1;
671     parameter Y_WIDTH = 1;
672
673     input [A_WIDTH-1:0] A;      // Input operand
674     input [B_WIDTH-1:0] B;      // Input operand
675     output [Y_WIDTH-1:0] X;     // A xor B (sign-extended, optional B inversion,
676                                 //          used in combination with
677                                 //          reduction-AND for $eq/$ne ops)
678     output [Y_WIDTH-1:0] Y;     // Sum
679
680     input CI;                   // Carry-in (set for $sub)
681     input BI;                   // Invert-B (set for $sub)
682     output [Y_WIDTH-1:0] CO;    // Carry-out
683
684     wire [Y_WIDTH-1:0] AA, BB;
685
686     generate
687         if (A_SIGNED && B_SIGNED) begin:BLOCK1
688             assign AA = $signed(A), BB = BI ? ~$signed(B) : $signed(B);
689         end else begin:BLOCK2
690             assign AA = $unsigned(A), BB = BI ? ~$unsigned(B) : $unsigned(B);
691         end
692     endgenerate
693
694     // this is 'x' if Y and CO should be all 'x', and '0' otherwise
695     wire y_co_undef = ^{A, A, B, B, CI, CI, BI, BI};
696
697     assign X = AA ^ BB;
698     // Full adder
699     assign Y = (AA + BB + CI) ^ {Y_WIDTH{y_co_undef}};
700
701     function get_carry;
702         input a, b, c;
703         get_carry = (a&b) | (a&c) | (b&c);
704     endfunction
705
706     genvar i;
707     generate
708         assign CO[0] = get_carry(AA[0], BB[0], CI) ^ y_co_undef;
709         for (i = 1; i < Y_WIDTH; i = i+1) begin:BLOCK3
710             assign CO[i] = get_carry(AA[i], BB[i], CO[i-1]) ^ y_co_undef;
711         end
712     endgenerate
713
714 endmodule
```

```
yosys> help $fa
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.71: simlib.v

```
module \$fa (A, B, C, X, Y);

    parameter WIDTH = 1;

    input [WIDTH-1:0] A, B, C;
    output [WIDTH-1:0] X, Y;

    wire [WIDTH-1:0] t1, t2, t3;

    assign t1 = A ^ B, t2 = A & B, t3 = C & t1;
    assign Y = t1 ^ C, X = (t2 | t3) ^ (Y ^ Y);

endmodule
```

```
yosys> help $lcu
```

Lookahead carry unit A building block dedicated to fast computation of carry-bits used in binary arithmetic operations. By replacing the ripple carry structure used in full-adder blocks, the more significant bits of the sum can be expected to be computed more quickly. Typically created during `techmap` of $alu cells (see the "_90_alu" rule in +/techmap.v).

Properties

is_evaluable

Simulation model (verilog)

Listing 9.72: simlib.v

```
module \$lcu (P, G, CI, CO);

    parameter WIDTH = 1;

    input [WIDTH-1:0] P;     // Propagate
    input [WIDTH-1:0] G;     // Generate
    input CI;                // Carry-in

    output reg [WIDTH-1:0] CO; // Carry-out

    integer i;
    always @* begin
        CO = 'bx;
        if (^{P, G, CI} !== 1'bx) begin
            CO[0] = G[0] || (P[0] && CI);
            for (i = 1; i < WIDTH; i = i+1)
                CO[i] = G[i] || (P[i] && CO[i-1]);
        end
    end
```

(continues on next page)

(continued from previous page)

```
652
653 endmodule
```

```
yosys> help $macc
```

Multiply and accumulate. A building block for summing any number of negated and unnegated signals and arithmetic products of pairs of signals. Cell port A concatenates pairs of signals to be multiplied together. When the second signal in a pair is zero length, a constant 1 is used instead as the second factor. Cell port B concatenates 1-bit-wide signals to also be summed, such as "carry in" in adders. Typically created by the `alumacc` pass, which transforms $add and $mul into $macc cells.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.73: simlib.v

```
1081 module \$macc (A, B, Y);
1082
1083     parameter A_WIDTH = 0;
1084     parameter B_WIDTH = 0;
1085     parameter Y_WIDTH = 0;
1086     // CONFIG determines the layout of A, as explained below
1087     parameter CONFIG = 4'b0000;
1088     parameter CONFIG_WIDTH = 4;
1089
1090     // In the terms used for this cell, there's mixed meanings for the term "port".␣
     ↪To disambiguate:
1091     // A cell port is for example the A input (it is constructed in C++ as cell->
     ↪setPort(ID::A, ...))
1092     // Multiplier ports are pairs of multiplier inputs ("factors").
1093     // If the second signal in such a pair is zero length, no multiplication is␣
     ↪necessary, and the first signal is just added to the sum.
1094     input [A_WIDTH-1:0] A; // Cell port A is the concatenation of all arithmetic␣
     ↪ports
1095     input [B_WIDTH-1:0] B; // Cell port B is the concatenation of single-bit␣
     ↪unsigned signals to be also added to the sum
1096     output reg [Y_WIDTH-1:0] Y; // Output sum
1097
1098     // Xilinx XSIM does not like $clog2() below..
1099     function integer my_clog2;
1100         input integer v;
1101         begin
1102             if (v > 0)
1103                 v = v - 1;
1104             my_clog2 = 0;
1105             while (v) begin
1106                 v = v >> 1;
1107                 my_clog2 = my_clog2 + 1;
1108             end
1109         end
1110     endfunction
1111
```

(continues on next page)

(continued from previous page)

```
    // Bits that a factor's length field in CONFIG per factor in cell port A
    localparam integer num_bits = CONFIG[3:0] > 0 ? CONFIG[3:0] : 1;
    // Number of multiplier ports
    localparam integer num_ports = (CONFIG_WIDTH-4) / (2 + 2*num_bits);
    // Minium bit width of an induction variable to iterate over all bits of cell port A
    localparam integer num_abits = my_clog2(A_WIDTH) > 0 ? my_clog2(A_WIDTH) : 1;

    // In this pseudocode, u(foo) means an unsigned int that's foo bits long.
    // The CONFIG parameter carries the following information:
    //    struct CONFIG {
    //        u4 num_bits;
    //        struct port_field {
    //            bool is_signed;
    //            bool is_subtract;
    //            u(num_bits) factor1_len;
    //            u(num_bits) factor2_len;
    //        }[num_ports];
    //    };

    // The A cell port carries the following information:
    //    struct A {
    //        u(CONFIG.port_field[0].factor1_len) port0factor1;
    //        u(CONFIG.port_field[0].factor2_len) port0factor2;
    //        u(CONFIG.port_field[1].factor1_len) port1factor1;
    //        u(CONFIG.port_field[1].factor2_len) port1factor2;
    //        ...
    //    };
    // and log(sizeof(A)) is num_abits.
    // No factor1 may have a zero length.
    // A factor2 having a zero length implies factor2 is replaced with a constant 1.

    // Additionally, B is an array of 1-bit-wide unsigned integers to also be summed up.
    // Finally, we have:
    // Y = port0factor1 * port0factor2 + port1factor1 * port1factor2 + ...
    //     * B[0] + B[1] + ...

    function [2*num_ports*num_abits-1:0] get_port_offsets;
        input [CONFIG_WIDTH-1:0] cfg;
        integer i, cursor;
        begin
            cursor = 0;
            get_port_offsets = 0;
            for (i = 0; i < num_ports; i = i+1) begin
                get_port_offsets[(2*i + 0)*num_abits +: num_abits] = cursor;
                cursor = cursor + cfg[4 + i*(2 + 2*num_bits) + 2 +: num_bits];
                get_port_offsets[(2*i + 1)*num_abits +: num_abits] = cursor;
                cursor = cursor + cfg[4 + i*(2 + 2*num_bits) + 2 + num_bits +: num_bits];
            end
        end
```

(continues on next page)

(continued from previous page)

```
1161    endfunction
1162
1163    localparam [2*num_ports*num_abits-1:0] port_offsets = get_port_offsets(CONFIG);
1164
1165    `define PORT_IS_SIGNED   (0 + CONFIG[4 + i*(2 + 2*num_bits)])
1166    `define PORT_DO_SUBTRACT (0 + CONFIG[4 + i*(2 + 2*num_bits) + 1])
1167    `define PORT_SIZE_A      (0 + CONFIG[4 + i*(2 + 2*num_bits) + 2 +: num_bits])
1168    `define PORT_SIZE_B      (0 + CONFIG[4 + i*(2 + 2*num_bits) + 2 + num_bits +:␣
    ↪num_bits])
1169    `define PORT_OFFSET_A    (0 + port_offsets[2*i*num_abits +: num_abits])
1170    `define PORT_OFFSET_B    (0 + port_offsets[2*i*num_abits + num_abits +: num_
    ↪abits])
1171
1172    integer i, j;
1173    reg [Y_WIDTH-1:0] tmp_a, tmp_b;
1174
1175    always @* begin
1176        Y = 0;
1177        for (i = 0; i < num_ports; i = i+1)
1178        begin
1179            tmp_a = 0;
1180            tmp_b = 0;
1181
1182            for (j = 0; j < `PORT_SIZE_A; j = j+1)
1183                tmp_a[j] = A[`PORT_OFFSET_A + j];
1184
1185            if (`PORT_IS_SIGNED && `PORT_SIZE_A > 0)
1186                for (j = `PORT_SIZE_A; j < Y_WIDTH; j = j+1)
1187                    tmp_a[j] = tmp_a[`PORT_SIZE_A-1];
1188
1189            for (j = 0; j < `PORT_SIZE_B; j = j+1)
1190                tmp_b[j] = A[`PORT_OFFSET_B + j];
1191
1192            if (`PORT_IS_SIGNED && `PORT_SIZE_B > 0)
1193                for (j = `PORT_SIZE_B; j < Y_WIDTH; j = j+1)
1194                    tmp_b[j] = tmp_b[`PORT_SIZE_B-1];
1195
1196            if (`PORT_SIZE_B > 0)
1197                tmp_a = tmp_a * tmp_b;
1198
1199            if (`PORT_DO_SUBTRACT)
1200                Y = Y - tmp_a;
1201            else
1202                Y = Y + tmp_a;
1203        end
1204        for (i = 0; i < B_WIDTH; i = i+1) begin
1205            Y = Y + B[i];
1206        end
1207    end
1208
1209    `undef PORT_IS_SIGNED
1210    `undef PORT_DO_SUBTRACT
```

(continues on next page)

(continued from previous page)

```
    `undef PORT_SIZE_A
    `undef PORT_SIZE_B
    `undef PORT_OFFSET_A
    `undef PORT_OFFSET_B

endmodule
```

yosys> help $macc_v2

Multiply and add. This cell represents a generic fused multiply-add operation, it supersedes the earlier $macc cell.

Properties

is_evaluable

Simulation model (verilog)

Listing 9.74: simlib.v

```
module \$macc_v2 (A, B, C, Y);

    parameter NPRODUCTS = 0;
    parameter NADDENDS = 0;
    parameter A_WIDTHS = 16'h0000;
    parameter B_WIDTHS = 16'h0000;
    parameter C_WIDTHS = 16'h0000;
    parameter Y_WIDTH = 0;

    parameter PRODUCT_NEGATED = 1'bx;
    parameter ADDEND_NEGATED = 1'bx;
    parameter A_SIGNED = 1'bx;
    parameter B_SIGNED = 1'bx;
    parameter C_SIGNED = 1'bx;

    function integer sum_widths1;
        input [(16*NPRODUCTS)-1:0] widths;
        integer i;
        begin
            sum_widths1 = 0;
            for (i = 0; i < NPRODUCTS; i++) begin
                sum_widths1 = sum_widths1 + widths[16*i+:16];
            end
        end
    endfunction

    function integer sum_widths2;
        input [(16*NADDENDS)-1:0] widths;
        integer i;
        begin
            sum_widths2 = 0;
            for (i = 0; i < NADDENDS; i++) begin
                sum_widths2 = sum_widths2 + widths[16*i+:16];
            end
        end
```

(continues on next page)

(continued from previous page)

```
1263    endfunction
1264
1265    input [sum_widths1(A_WIDTHS)-1:0] A; // concatenation of LHS factors
1266    input [sum_widths1(B_WIDTHS)-1:0] B; // concatenation of RHS factors
1267    input [sum_widths2(C_WIDTHS)-1:0] C; // concatenation of summands
1268    output reg [Y_WIDTH-1:0] Y; // output sum
1269
1270    integer i, j, ai, bi, ci, aw, bw, cw;
1271    reg [Y_WIDTH-1:0] product;
1272    reg [Y_WIDTH-1:0] addend, oper_a, oper_b;
1273
1274    always @* begin
1275        Y = 0;
1276        ai = 0;
1277        bi = 0;
1278        for (i = 0; i < NPRODUCTS; i = i+1)
1279        begin
1280            aw = A_WIDTHS[16*i+:16];
1281            bw = B_WIDTHS[16*i+:16];
1282
1283            oper_a = 0;
1284            oper_b = 0;
1285            for (j = 0; j < Y_WIDTH && j < aw; j = j + 1)
1286                oper_a[j] = A[ai + j];
1287            for (j = 0; j < Y_WIDTH && j < bw; j = j + 1)
1288                oper_b[j] = B[bi + j];
1289            // A_SIGNED[i] == B_SIGNED[i] as RTLIL invariant
1290            if (A_SIGNED[i] && B_SIGNED[i]) begin
1291                for (j = aw; j > 0 && j < Y_WIDTH; j = j + 1)
1292                    oper_a[j] = oper_a[j - 1];
1293                for (j = bw; j > 0 && j < Y_WIDTH; j = j + 1)
1294                    oper_b[j] = oper_b[j - 1];
1295            end
1296
1297            product = oper_a * oper_b;
1298
1299            if (PRODUCT_NEGATED[i])
1300                Y = Y - product;
1301            else
1302                Y = Y + product;
1303
1304            ai = ai + aw;
1305            bi = bi + bw;
1306        end
1307
1308        ci = 0;
1309        for (i = 0; i < NADDENDS; i = i+1)
1310        begin
1311            cw = C_WIDTHS[16*i+:16];
1312
1313            addend = 0;
1314            for (j = 0; j < Y_WIDTH && j < cw; j = j + 1)
```

(continues on next page)

(continued from previous page)

```
                addend[j] = C[ci + j];
            if (C_SIGNED[i]) begin
                for (j = cw; j > 0 && j < Y_WIDTH; j = j + 1)
                    addend[j] = addend[j - 1];
            end

            if (ADDEND_NEGATED[i])
                Y = Y - addend;
            else
                Y = Y + addend;

            ci = ci + cw;
        end
    end

endmodule
```

9.1.8 Arbitrary logic functions

The *$lut* cell type implements a single-output LUT (lookup table). It implements an arbitrary logic function with its \LUT parameter to map input port \A to values of \Y output port values. In psuedocode: `Y = \LUT[A]`. \A has width set by parameter \WIDTH and \Y has a width of 1. Every logic function with a single bit output has a unique *$lut* representation.

The *$sop* cell type implements a sum-of-products expression, also known as disjunctive normal form (DNF). It implements an arbitrary logic function. Its structure mimics a programmable logic array (PLA). Output port \Y is the sum of products of the bits of the input port \A as defined by parameter \TABLE. \A is \WIDTH bits wide. The number of products in the sum is set by parameter \DEPTH, and each product has two bits for each input bit - for the presence of the unnegated and negated version of said input bit in the product. Therefore the \TABLE parameter holds `2 * \WIDTH * \DEPTH` bits.

For example:

Let \WIDTH be 3. We would like to represent `\Y =~\A[0] + \A[1]~\A[2]`. There are 2 products to be summed, so \DEPTH shall be 2.

```
~A[2]-----+
 A[2]----+|
~A[1]---+||
 A[1]--+|||
~A[0]-+||||
 A[0]+|||||
     |||||| product formula
     010000 ~\A[0]
     001001 \A[1]~\A[2]
```

So the value of \TABLE will become 010000001001.

Any logic function with a single bit output can be represented with $sop but may have variously minimized or ordered summands represented in the \TABLE values.

```
yosys> help $lut
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.75: simlib.v

```
module \$lut (A, Y);

    parameter WIDTH = 0;
    parameter LUT = 0;

    input [WIDTH-1:0] A;
    output Y;

    \$bmux #(.WIDTH(1), .S_WIDTH(WIDTH)) mux(.A(LUT[(1<<WIDTH)-1:0]), .S(A), .Y(Y));

endmodule
```

yosys> help $sop

Properties

is_evaluable

Simulation model (verilog)

Listing 9.76: simlib.v

```
module \$sop (A, Y);

    parameter WIDTH = 0;
    parameter DEPTH = 0;
    parameter TABLE = 0;

    input [WIDTH-1:0] A;
    output reg Y;

    integer i, j;
    reg match;

    always @* begin
        Y = 0;
        for (i = 0; i < DEPTH; i=i+1) begin
            match = 1;
            for (j = 0; j < WIDTH; j=j+1) begin
                if (TABLE[2*WIDTH*i + 2*j + 0] && A[j]) match = 0;
                if (TABLE[2*WIDTH*i + 2*j + 1] && !A[j]) match = 0;
            end
            if (match) Y = 1;
        end
    end

endmodule
```

9.1.9 Specify rules

Todo

$specify2, *$specify3*, and *$specrule* cells.

```
yosys> help $specify2

    Simulation model (verilog)
```

Listing 9.77: simlib.v

```
1831 module \$specify2 (EN, SRC, DST);
1832
1833     parameter FULL = 0;
1834     parameter SRC_WIDTH = 1;
1835     parameter DST_WIDTH = 1;
1836
1837     parameter SRC_DST_PEN = 0;
1838     parameter SRC_DST_POL = 0;
1839
1840     parameter T_RISE_MIN = 0;
1841     parameter T_RISE_TYP = 0;
1842     parameter T_RISE_MAX = 0;
1843
1844     parameter T_FALL_MIN = 0;
1845     parameter T_FALL_TYP = 0;
1846     parameter T_FALL_MAX = 0;
1847
1848     input EN;
1849     input [SRC_WIDTH-1:0] SRC;
1850     input [DST_WIDTH-1:0] DST;
1851
1852     localparam SD = SRC_DST_PEN ? (SRC_DST_POL ? 1 : 2) : 0;
1853
1854     `ifdef SIMLIB_SPECIFY
1855     specify
1856         if (EN && SD==0 && !FULL) (SRC  => DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
    ↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1857         if (EN && SD==0 &&  FULL) (SRC  *> DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
    ↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1858         if (EN && SD==1 && !FULL) (SRC +=> DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
    ↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1859         if (EN && SD==1 &&  FULL) (SRC +*> DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
    ↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1860         if (EN && SD==2 && !FULL) (SRC -=> DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
    ↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1861         if (EN && SD==2 &&  FULL) (SRC -*> DST) = (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX,
    ↪ T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1862     endspecify
1863     `endif
```

(continues on next page)

(continued from previous page)

```
1864
1865 endmodule
```

```
yosys> help $specify3

    Simulation model (verilog)
```

Listing 9.78: simlib.v

```
1870 module \$specify3 (EN, SRC, DST, DAT);
1871
1872     parameter FULL = 0;
1873     parameter SRC_WIDTH = 1;
1874     parameter DST_WIDTH = 1;
1875
1876     parameter EDGE_EN = 0;
1877     parameter EDGE_POL = 0;
1878
1879     parameter SRC_DST_PEN = 0;
1880     parameter SRC_DST_POL = 0;
1881
1882     parameter DAT_DST_PEN = 0;
1883     parameter DAT_DST_POL = 0;
1884
1885     parameter T_RISE_MIN = 0;
1886     parameter T_RISE_TYP = 0;
1887     parameter T_RISE_MAX = 0;
1888
1889     parameter T_FALL_MIN = 0;
1890     parameter T_FALL_TYP = 0;
1891     parameter T_FALL_MAX = 0;
1892
1893     input EN;
1894     input [SRC_WIDTH-1:0] SRC;
1895     input [DST_WIDTH-1:0] DST, DAT;
1896
1897     localparam ED = EDGE_EN     ? (EDGE_POL    ? 1 : 2) : 0;
1898     localparam SD = SRC_DST_PEN ? (SRC_DST_POL ? 1 : 2) : 0;
1899     localparam DD = DAT_DST_PEN ? (DAT_DST_POL ? 1 : 2) : 0;
1900
1901     `ifdef SIMLIB_SPECIFY
1902     specify
1903         // DD=0
1904
1905         if (EN && DD==0 && SD==0 && ED==0 && !FULL) (        SRC  => (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1906         if (EN && DD==0 && SD==0 && ED==0 &&  FULL) (        SRC  *> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1907         if (EN && DD==0 && SD==0 && ED==1 && !FULL) (posedge SRC  => (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1908         if (EN && DD==0 && SD==0 && ED==1 &&  FULL) (posedge SRC  *> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
```

(continues on next page)

(continued from previous page)

```
1909         if (EN && DD==0 && SD==0 && ED==2 && !FULL) (negedge SRC  => (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1910         if (EN && DD==0 && SD==0 && ED==2 &&  FULL) (negedge SRC  *> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1911
1912         if (EN && DD==0 && SD==1 && ED==0 && !FULL) (        SRC +=> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1913         if (EN && DD==0 && SD==1 && ED==0 &&  FULL) (        SRC +*> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1914         if (EN && DD==0 && SD==1 && ED==1 && !FULL) (posedge SRC +=> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1915         if (EN && DD==0 && SD==1 && ED==1 &&  FULL) (posedge SRC +*> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1916         if (EN && DD==0 && SD==1 && ED==2 && !FULL) (negedge SRC +=> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1917         if (EN && DD==0 && SD==1 && ED==2 &&  FULL) (negedge SRC +*> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1918
1919         if (EN && DD==0 && SD==2 && ED==0 && !FULL) (        SRC -=> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1920         if (EN && DD==0 && SD==2 && ED==0 &&  FULL) (        SRC -*> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1921         if (EN && DD==0 && SD==2 && ED==1 && !FULL) (posedge SRC -=> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1922         if (EN && DD==0 && SD==2 && ED==1 &&  FULL) (posedge SRC -*> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1923         if (EN && DD==0 && SD==2 && ED==2 && !FULL) (negedge SRC -=> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1924         if (EN && DD==0 && SD==2 && ED==2 &&  FULL) (negedge SRC -*> (DST  : DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1925
1926         // DD=1
1927
1928         if (EN && DD==1 && SD==0 && ED==0 && !FULL) (        SRC  => (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1929         if (EN && DD==1 && SD==0 && ED==0 &&  FULL) (        SRC  *> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1930         if (EN && DD==1 && SD==0 && ED==1 && !FULL) (posedge SRC  => (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1931         if (EN && DD==1 && SD==0 && ED==1 &&  FULL) (posedge SRC  *> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1932         if (EN && DD==1 && SD==0 && ED==2 && !FULL) (negedge SRC  => (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1933         if (EN && DD==1 && SD==0 && ED==2 &&  FULL) (negedge SRC  *> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1934
1935         if (EN && DD==1 && SD==1 && ED==0 && !FULL) (        SRC +=> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1936         if (EN && DD==1 && SD==1 && ED==0 &&  FULL) (        SRC +*> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1937         if (EN && DD==1 && SD==1 && ED==1 && !FULL) (posedge SRC +=> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
```

(continues on next page)

(continued from previous page)

```
1938         if (EN && DD==1 && SD==1 && ED==1 &&  FULL) (posedge SRC +*> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1939         if (EN && DD==1 && SD==1 && ED==2 && !FULL) (negedge SRC +=> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1940         if (EN && DD==1 && SD==1 && ED==2 &&  FULL) (negedge SRC +*> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1941
1942         if (EN && DD==1 && SD==2 && ED==0 && !FULL) (        SRC -=> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1943         if (EN && DD==1 && SD==2 && ED==0 &&  FULL) (        SRC -*> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1944         if (EN && DD==1 && SD==2 && ED==1 && !FULL) (posedge SRC -=> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1945         if (EN && DD==1 && SD==2 && ED==1 &&  FULL) (posedge SRC -*> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1946         if (EN && DD==1 && SD==2 && ED==2 && !FULL) (negedge SRC -=> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1947         if (EN && DD==1 && SD==2 && ED==2 &&  FULL) (negedge SRC -*> (DST +: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1948
1949         // DD=2
1950
1951         if (EN && DD==2 && SD==0 && ED==0 && !FULL) (        SRC  => (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1952         if (EN && DD==2 && SD==0 && ED==0 &&  FULL) (        SRC  *> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1953         if (EN && DD==2 && SD==0 && ED==1 && !FULL) (posedge SRC  => (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1954         if (EN && DD==2 && SD==0 && ED==1 &&  FULL) (posedge SRC  *> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1955         if (EN && DD==2 && SD==0 && ED==2 && !FULL) (negedge SRC  => (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1956         if (EN && DD==2 && SD==0 && ED==2 &&  FULL) (negedge SRC  *> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1957
1958         if (EN && DD==2 && SD==1 && ED==0 && !FULL) (        SRC +=> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1959         if (EN && DD==2 && SD==1 && ED==0 &&  FULL) (        SRC +*> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1960         if (EN && DD==2 && SD==1 && ED==1 && !FULL) (posedge SRC +=> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1961         if (EN && DD==2 && SD==1 && ED==1 &&  FULL) (posedge SRC +*> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1962         if (EN && DD==2 && SD==1 && ED==2 && !FULL) (negedge SRC +=> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1963         if (EN && DD==2 && SD==1 && ED==2 &&  FULL) (negedge SRC +*> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1964
1965         if (EN && DD==2 && SD==2 && ED==0 && !FULL) (        SRC -=> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1966         if (EN && DD==2 && SD==2 && ED==0 &&  FULL) (        SRC -*> (DST -: DAT))␣
     ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
```

(continues on next page)

(continued from previous page)

```
1967        if (EN && DD==2 && SD==2 && ED==1 && !FULL) (posedge SRC -=> (DST -: DAT))␣
    ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1968        if (EN && DD==2 && SD==2 && ED==1 &&  FULL) (posedge SRC -*> (DST -: DAT))␣
    ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1969        if (EN && DD==2 && SD==2 && ED==2 && !FULL) (negedge SRC -=> (DST -: DAT))␣
    ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1970        if (EN && DD==2 && SD==2 && ED==2 &&  FULL) (negedge SRC -*> (DST -: DAT))␣
    ↪= (T_RISE_MIN:T_RISE_TYP:T_RISE_MAX, T_FALL_MIN:T_FALL_TYP:T_FALL_MAX);
1971    endspecify
1972    `endif
1973
1974 endmodule
```

```
yosys> help $specrule

    Simulation model (verilog)
```

Listing 9.79: simlib.v

```
1979 module \$specrule (SRC_EN, DST_EN, SRC, DST);
1980
1981     parameter TYPE = "";
1982     parameter T_LIMIT = 0;
1983     parameter T_LIMIT2 = 0;
1984
1985     parameter SRC_WIDTH = 1;
1986     parameter DST_WIDTH = 1;
1987
1988     parameter SRC_PEN = 0;
1989     parameter SRC_POL = 0;
1990
1991     parameter DST_PEN = 0;
1992     parameter DST_POL = 0;
1993
1994     input SRC_EN, DST_EN;
1995     input [SRC_WIDTH-1:0] SRC;
1996     input [DST_WIDTH-1:0] DST;
1997
1998     `ifdef SIMLIB_SPECIFY
1999     specify
2000         // TBD
2001     endspecify
2002     `endif
2003
2004 endmodule
```

9.1.10 Formal verification cells

Note

Some front-ends may not support the generic *$check* cell, in such cases calling `chformal -lower` will

convert each *$check* cell into it's equivalent. See *chformal* for more.

Todo

Describe formal cells

$check, *$assert*, *$assume*, *$live*, *$fair*, *$cover*, *$equiv*, *$initstate*, *$anyconst*, *$anyseq*, *$anyinit*, *$allconst*, and *$allseq*.

Also *$ff* and *$_FF_* cells.

```
yosys> help $allconst

    Simulation model (verilog)
```

Listing 9.80: simlib.v

```
2183 module \$allconst (Y);
2184
2185     parameter WIDTH = 0;
2186
2187     output [WIDTH-1:0] Y;
2188
2189     assign Y = 'bx;
2190
2191 endmodule
```

```
yosys> help $allseq

    Simulation model (verilog)
```

Listing 9.81: simlib.v

```
2196 module \$allseq (Y);
2197
2198     parameter WIDTH = 0;
2199
2200     output [WIDTH-1:0] Y;
2201
2202     assign Y = 'bx;
2203
2204 endmodule
```

```
yosys> help $anyconst

    Simulation model (verilog)
```

Listing 9.82: simlib.v

```
2136 module \$anyconst (Y);
2137
```

(continues on next page)

(continued from previous page)

```
2138    parameter WIDTH = 0;
2139
2140    output [WIDTH-1:0] Y;
2141
2142    assign Y = 'bx;
2143
2144  endmodule
```

```
yosys> help $anyinit
```

Simulation model (verilog)

Listing 9.83: simlib.v

```
2165  module \$anyinit (D, Q);
2166
2167      parameter WIDTH = 0;
2168
2169      input [WIDTH-1:0] D;
2170      output reg [WIDTH-1:0] Q;
2171
2172      initial Q <= 'bx;
2173
2174      always @(`SIMLIB_GLOBAL_CLOCK) begin
2175          Q <= D;
2176      end
2177
2178  endmodule
```

```
yosys> help $anyseq
```

Simulation model (verilog)

Listing 9.84: simlib.v

```
2149  module \$anyseq (Y);
2150
2151      parameter WIDTH = 0;
2152
2153      output [WIDTH-1:0] Y;
2154
2155      assign Y = 'bx;
2156
2157  endmodule
```

```
yosys> help $assert
```

Simulation model (verilog)

Listing 9.85: simlib.v

```
2055  module \$assert (A, EN);
2056
```

(continues on next page)

(continued from previous page)

```
2057     input A, EN;
2058
2059     `ifndef SIMLIB_NOCHECKS
2060     always @* begin
2061         if (A !== 1'b1 && EN === 1'b1) begin
2062             $display("Assertion %m failed!");
2063             $stop;
2064         end
2065     end
2066     `endif
2067
2068 endmodule
```

```
yosys> help $assume

    Simulation model (verilog)
```

Listing 9.86: simlib.v

```
2073 module \$assume (A, EN);
2074
2075     input A, EN;
2076
2077     `ifndef SIMLIB_NOCHECKS
2078     always @* begin
2079         if (A !== 1'b1 && EN === 1'b1) begin
2080             $display("Assumption %m failed!");
2081             $stop;
2082         end
2083     end
2084     `endif
2085
2086 endmodule
```

```
yosys> help $cover

    Simulation model (verilog)
```

Listing 9.87: simlib.v

```
2109 module \$cover (A, EN);
2110
2111     input A, EN;
2112
2113 endmodule
```

```
yosys> help $equiv

    Simulation model (verilog)
```

Listing 9.88: simlib.v

```
module \$equiv (A, B, Y);

    input A, B;
    output Y;

    assign Y = (A !== 1'bx && A !== B) ? 1'bx : A;

    `ifndef SIMLIB_NOCHECKS
    always @* begin
        if (A !== 1'bx && A !== B) begin
            $display("Equivalence failed!");
            $stop;
        end
    end
    `endif

endmodule
```

```
yosys> help $fair
```

```
    Simulation model (verilog)
```

Listing 9.89: simlib.v

```
module \$fair (A, EN);

    input A, EN;

endmodule
```

```
yosys> help $ff
```

```
    Simulation model (verilog)
```

Listing 9.90: simlib.v

```
module \$ff (D, Q);

    parameter WIDTH = 0;

    input [WIDTH-1:0] D;
    output reg [WIDTH-1:0] Q;

    always @(`SIMLIB_GLOBAL_CLOCK) begin
        Q <= D;
    end

endmodule
```

```
yosys> help $initstate
```

```
    Simulation model (verilog)
```

Listing 9.91: simlib.v

```
2118 module \$initstate (Y);
2119
2120     output reg Y = 1;
2121     reg [3:0] cnt = 1;
2122     reg trig = 0;
2123
2124     initial trig <= 1;
2125
2126     always @(cnt, trig) begin
2127         Y <= |cnt;
2128         cnt <= cnt + |cnt;
2129     end
2130
2131 endmodule
```

yosys> help $live

Simulation model (verilog)

Listing 9.92: simlib.v

```
2091 module \$live (A, EN);
2092
2093     input A, EN;
2094
2095 endmodule
```

Formal support cells

yosys> help $future_ff

Simulation model (verilog)

Listing 9.93: simlib.v

```
3208 module \$future_ff (A, Y);
3209
3210     parameter WIDTH = 0;
3211
3212     input [WIDTH-1:0] A;
3213     output [WIDTH-1:0] Y;
3214
3215     assign Y = A;
3216
3217 endmodule
```

yosys> help $get_tag

Simulation model (verilog)

Listing 9.94: simlib.v

```
3167 module \$get_tag (A, Y);
3168
3169     parameter TAG = "";
3170     parameter WIDTH = 0;
3171
3172     input [WIDTH-1:0] A;
3173     output [WIDTH-1:0] Y;
3174
3175     assign Y = A;
3176
3177 endmodule
```

```
yosys> help $original_tag
```

```
Simulation model (verilog)
```

Listing 9.95: simlib.v

```
3194 module \$original_tag (A, Y);
3195
3196     parameter TAG = "";
3197     parameter WIDTH = 0;
3198
3199     input [WIDTH-1:0] A;
3200     output [WIDTH-1:0] Y;
3201
3202     assign Y = A;
3203
3204 endmodule
```

```
yosys> help $overwrite_tag
```

```
Simulation model (verilog)
```

Listing 9.96: simlib.v

```
3181 module \$overwrite_tag (A, SET, CLR);
3182
3183     parameter TAG = "";
3184     parameter WIDTH = 0;
3185
3186     input [WIDTH-1:0] A;
3187     input [WIDTH-1:0] SET;
3188     input [WIDTH-1:0] CLR;
3189
3190 endmodule
```

```
yosys> help $set_tag
```

```
Simulation model (verilog)
```

Listing 9.97: simlib.v

```
3151 module \$set_tag (A, SET, CLR, Y);
3152
3153     parameter TAG = "";
3154     parameter WIDTH = 0;
3155
3156     input [WIDTH-1:0] A;
3157     input [WIDTH-1:0] SET;
3158     input [WIDTH-1:0] CLR;
3159     output [WIDTH-1:0] Y;
3160
3161     assign Y = A;
3162
3163 endmodule
```

9.1.11 Debugging cells

The *$print* cell is used to log the values of signals, akin to (and translatable to) the `$display` and `$write` family of tasks in Verilog. It has the following parameters:

`FORMAT`
: The internal format string. The syntax is described below.

`ARGS_WIDTH`
: The width (in bits) of the signal on the `ARGS` port.

`TRG_ENABLE`
: True if triggered on specific signals defined in `TRG`; false if triggered whenever `ARGS` or `EN` change and `EN` is 1.

If `TRG_ENABLE` is true, the following parameters also apply:

`TRG_WIDTH`
: The number of bits in the `TRG` port.

`TRG_POLARITY`
: For each bit in `TRG`, 1 if that signal is positive-edge triggered, 0 if negative-edge triggered.

`PRIORITY`
: When multiple *$print* or *$check* cells fire on the same trigger, they execute in descending priority order.

Ports:

`TRG`
: The signals that control when this *$print* cell is triggered.

  If the width of this port is zero and `TRG_ENABLE` is true, the cell is triggered during initial evaluation (time zero) only.

`EN`
: Enable signal for the whole cell.

`ARGS`
: The values to be displayed, in format string order.

```
yosys> help $check
```

Simulation model (verilog)

Listing 9.98: simlib.v

```
2250 module \$check (A, EN, TRG, ARGS);
2251
2252     parameter FLAVOR = "";
2253     parameter PRIORITY = 0;
2254
2255     parameter FORMAT = "";
2256     parameter ARGS_WIDTH = 0;
2257
2258     parameter TRG_ENABLE = 1;
2259     parameter TRG_WIDTH = 0;
2260     parameter TRG_POLARITY = 0;
2261
2262     input A;
2263     input EN;
2264     input [TRG_WIDTH-1:0] TRG;
2265     input [ARGS_WIDTH-1:0] ARGS;
2266
2267 endmodule
```

```
yosys> help $print
```

Simulation model (verilog)

Listing 9.99: simlib.v

```
2230 module \$print (EN, TRG, ARGS);
2231
2232     parameter PRIORITY = 0;
2233
2234     parameter FORMAT = "";
2235     parameter signed ARGS_WIDTH = 0;
2236
2237     parameter TRG_ENABLE = 1;
2238     parameter signed TRG_WIDTH = 0;
2239     parameter TRG_POLARITY = 0;
2240
2241     input EN;
2242     input [TRG_WIDTH-1:0] TRG;
2243     input [ARGS_WIDTH-1:0] ARGS;
2244
2245 endmodule
```

```
yosys> help $scopeinfo
```

Simulation model (verilog)

Listing 9.100: simlib.v

```
3222 module \$scopeinfo ();
3223
3224     parameter TYPE = "";
3225
3226 endmodule
```

Format string syntax

The format string syntax resembles Python f-strings. Regular text is passed through unchanged until a format specifier is reached, starting with a `{`.

Format specifiers have the following syntax. Unless noted, all items are required:

`{`
: Denotes the start of the format specifier.

size
: Signal size in bits; this many bits are consumed from the `ARGS` port by this specifier.

`:`
: Separates the size from the remaining items.

justify
: `>` for right-justified, `<` for left-justified.

padding
: `0` for zero-padding, or a space for space-padding.

width*?*
: (optional) The number of characters wide to pad to.

base

- `b` for base-2 integers (binary)
- `o` for base-8 integers (octal)
- `d` for base-10 integers (decimal)
- `h` for base-16 integers (hexadecimal)
- `c` for ASCII characters/strings
- `t` and `r` for simulation time (corresponding to `$time` and `$realtime`)

For integers, this item may follow:

`+`*?*
: (optional, decimals only) Include a leading plus for non-negative numbers. This can assist with symmetry with negatives in tabulated output.

signedness
: `u` for unsigned, `s` for signed. This distinction is only respected when rendering decimals.

ASCII characters/strings have no special options, but the signal size must be divisible by 8.

For simulation time, the signal size must be zero.

Finally:

`}`
: Denotes the end of the format specifier.

Some example format specifiers:

- `{8:>02hu}` - 8-bit unsigned integer rendered as hexadecimal, right-justified, zero-padded to 2 characters wide.
- `{32:< 15d+s}` - 32-bit signed integer rendered as decimal, left-justified, space-padded to 15 characters wide, positive values prefixed with `+`.
- `{16:< 10hu}` - 16-bit unsigned integer rendered as hexadecimal, left-justified, space-padded to 10 characters wide.
- `{0:>010t}` - simulation time, right-justified, zero-padded to 10 characters wide.

To include literal `{` and `}` characters in your format string, use `{{` and `}}` respectively.

It is an error for a format string to consume more or less bits from `ARGS` than the port width.

Values are never truncated, regardless of the specified width.

Note that further restrictions on allowable combinations of options may apply depending on the backend used.

For example, Verilog does not have a format specifier that allows zero-padding a string (i.e. more than 1 ASCII character), though zero-padding a single character is permitted.

Thus, while the RTLIL format specifier `{8:>02c}` translates to `%02c`, `{16:>02c}` cannot be represented in Verilog and will fail to emit. In this case, `{16:> 02c}` must be used, which translates to `%2s`.

9.1.12 Wire cells

Todo

Add information about *$slice* and *$concat* cells.

```
yosys> help $concat
```

Concatenation of inputs into a single output (Y = {B, A}).

Properties

is_evaluable

Simulation model (verilog)

Listing 9.101: simlib.v

```
1625 module \$concat (A, B, Y);
1626
1627     parameter A_WIDTH = 0;
1628     parameter B_WIDTH = 0;
1629
1630     input [A_WIDTH-1:0] A;
1631     input [B_WIDTH-1:0] B;
1632     output [A_WIDTH+B_WIDTH-1:0] Y;
1633
1634     assign Y = {B, A};
1635
1636 endmodule
```

```
yosys> help $connect
```

Simulation model (verilog)

Listing 9.102: simlib.v

```
3232 module \$connect (A, B);
3233
3234     parameter WIDTH = 0;
3235
3236     inout [WIDTH-1:0] A;
3237     inout [WIDTH-1:0] B;
3238
3239     tran connect[WIDTH-1:0] (A, B);
3240
3241 endmodule
```

```
yosys> help $input_port
```

Simulation model (verilog)

Listing 9.103: simlib.v

```
3246 module \$input_port (Y);
3247
3248     parameter WIDTH = 0;
3249
3250     inout [WIDTH-1:0] Y;
3251
3252 endmodule
```

```
yosys> help $slice
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.104: simlib.v

```
1604 module \$slice (A, Y);
1605
1606     parameter OFFSET = 0;
1607     parameter A_WIDTH = 0;
1608     parameter Y_WIDTH = 0;
1609
1610     input [A_WIDTH-1:0] A;
1611     output [Y_WIDTH-1:0] Y;
1612
1613     assign Y = A >> OFFSET;
1614
1615 endmodule
```

9.2 Gate-level cells

For gate level logic networks, fixed function single bit cells are used that do not provide any parameters.

Simulation models for these cells can be found in the file `techlibs/common/simcells.v` in the Yosys source tree.

In most cases gate level logic networks are created from RTL networks using the techmap pass. The flip-flop cells from the gate level logic network can be mapped to physical flip-flop cells from a Liberty file using the dfflibmap pass. The combinatorial logic cells can be mapped to physical cells from a Liberty file via ABC using the abc pass.

9.2.1 Combinatorial cells (simple)

Table 9.4: Cell types for gate level combinatorial cells (simple)

Verilog	Cell Type
`Y = A`	*$_BUF_*
`Y = ~A`	*$_NOT_*
`Y = A & B`	*$_AND_*
`Y = ~(A & B)`	*$_NAND_*
`Y = A | B`	*$_OR_*
`Y = ~(A | B)`	*$_NOR_*
`Y = A ^ B`	*$_XOR_*
`Y = ~(A ^ B)`	*$_XNOR_*
`Y = S ? B : A`	*$_MUX_*

`yosys> help $_AND_`

A 2-input AND gate.

```
Truth table:    A B | Y
               -----+---
                0 0 | 0
                0 1 | 0
                1 0 | 0
                1 1 | 1
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.105: simcells.v

```
78 module \$_AND_ (A, B, Y);
79     input A, B;
80     output Y;
81     assign Y = A & B;
82 endmodule
```

`yosys> help $_BUF_`

A buffer. This cell type is always optimized away by the opt_clean pass.

```
Truth table:    A | Y
               ---+---
                0 | 0
                1 | 1
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.106: simcells.v

```
40 module \$_BUF_ (A, Y);
41     input A;
42     output Y;
43     assign Y = A;
44 endmodule
```

yosys> help **$_MUX_**

A 2-input MUX gate.

```
Truth table:    A B S | Y
               -------+---
                a - 0 | a
                - b 1 | b
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.107: simcells.v

```
236 module \$_MUX_ (A, B, S, Y);
237     input A, B, S;
238     output Y;
239     assign Y = S ? B : A;
240 endmodule
```

yosys> help **$_NAND_**

A 2-input NAND gate.

```
Truth table:    A B | Y
               -----+---
                0 0 | 1
                0 1 | 1
                1 0 | 1
                1 1 | 0
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.108: simcells.v

```
98  module \$_NAND_ (A, B, Y);
99      input A, B;
100     output Y;
101     assign Y = ~(A & B);
102 endmodule
```

yosys> help $_NOR_

A 2-input NOR gate.

```
Truth table:    A B | Y
               -----+---
                0 0 | 1
                0 1 | 0
                1 0 | 0
                1 1 | 0
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.109: simcells.v

```
138 module \$_NOR_ (A, B, Y);
139     input A, B;
140     output Y;
141     assign Y = ~(A | B);
142 endmodule
```

yosys> help $_NOT_

An inverter gate.

```
Truth table:    A | Y
               ---+---
                0 | 1
                1 | 0
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.110: simcells.v

```
58  module \$_NOT_ (A, Y);
59      input A;
60      output Y;
61      assign Y = ~A;
62  endmodule
```

yosys> help $_OR_

A 2-input OR gate.

```
Truth table:    A B | Y
               -----+---
                0 0 | 0
                0 1 | 1
                1 0 | 1
                1 1 | 1
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.111: simcells.v

```
118 module \$_OR_ (A, B, Y);
119     input A, B;
120     output Y;
121     assign Y = A | B;
122 endmodule
```

yosys> help $_XNOR_

A 2-input XNOR gate.

```
Truth table:    A B | Y
               -----+---
                0 0 | 1
                0 1 | 0
                1 0 | 0
                1 1 | 1
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.112: simcells.v

```
178 module \$_XNOR_ (A, B, Y);
179     input A, B;
180     output Y;
181     assign Y = ~(A ^ B);
182 endmodule
```

yosys> help $_XOR_

A 2-input XOR gate.

```
Truth table:    A B | Y
               -----+---
                0 0 | 0
```

(continues on next page)

(continued from previous page)

```
                0 1 | 1
                1 0 | 1
                1 1 | 0
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.113: simcells.v

```
158 module \$_XOR_ (A, B, Y);
159     input A, B;
160     output Y;
161     assign Y = A ^ B;
162 endmodule
```

9.2.2 Combinatorial cells (combined)

These cells combine two or more combinatorial cells (simple) into a single cell.

Table 9.5: Cell types for gate level combinatorial cells (combined)

Verilog	Cell Type
Y = A & ~B	*$_ANDNOT_*
Y = A | ~B	*$_ORNOT_*
Y = ~((A & B) | C)	*$_AOI3_*
Y = ~((A | B) & C)	*$_OAI3_*
Y = ~((A & B) | (C & D))	*$_AOI4_*
Y = ~((A | B) & (C | D))	*$_OAI4_*
Y = ~(S ? B : A)	*$_NMUX_*
(see below)	*$_MUX4_*
(see below)	*$_MUX8_*
(see below)	*$_MUX16_*

The *$_MUX4_* , *$_MUX8_* and *$_MUX16_* cells are used to model wide muxes, and correspond to the following Verilog code:

```
// $_MUX4_
assign Y = T ? (S ? D : C) :
               (S ? B : A);
// $_MUX8_
assign Y = U ? T ? (S ? H : G) :
                   (S ? F : E) :
               T ? (S ? D : C) :
                   (S ? B : A);
// $_MUX16_
assign Y = V ? U ? T ? (S ? P : O) :
                       (S ? N : M) :
                   T ? (S ? L : K) :
                       (S ? J : I) :
```

(continues on next page)

(continued from previous page)

```
              U ? T ? (S ? H : G) :
                      (S ? F : E) :
                  T ? (S ? D : C) :
                      (S ? B : A);
```

yosys> help $_ANDNOT_

A 2-input AND-NOT gate.

```
Truth table:    A B | Y
               -----+---
                0 0 | 0
                0 1 | 0
                1 0 | 1
                1 1 | 0
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.114: simcells.v

```
198 module \$_ANDNOT_ (A, B, Y);
199     input A, B;
200     output Y;
201     assign Y = A & (~B);
202 endmodule
```

yosys> help $_AOI3_

A 3-input And-Or-Invert gate.

```
Truth table:    A B C | Y
               -------+---
                0 0 0 | 1
                0 0 1 | 0
                0 1 0 | 1
                0 1 1 | 0
                1 0 0 | 1
                1 0 1 | 0
                1 1 0 | 0
                1 1 1 | 0
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.115: simcells.v

```
367 module \$_AOI3_ (A, B, C, Y);
368     input A, B, C;
369     output Y;
370     assign Y = ~((A & B) | C);
371 endmodule
```

yosys> help $_AOI4_

A 4-input And-Or-Invert gate.

```
Truth table:    A B C D | Y
                ---------+---
                0 0 0 0 | 1
                0 0 0 1 | 1
                0 0 1 0 | 1
                0 0 1 1 | 0
                0 1 0 0 | 1
                0 1 0 1 | 1
                0 1 1 0 | 1
                0 1 1 1 | 0
                1 0 0 0 | 1
                1 0 0 1 | 1
                1 0 1 0 | 1
                1 0 1 1 | 0
                1 1 0 0 | 0
                1 1 0 1 | 0
                1 1 1 0 | 0
                1 1 1 1 | 0
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.116: simcells.v

```
423 module \$_AOI4_ (A, B, C, D, Y);
424     input A, B, C, D;
425     output Y;
426     assign Y = ~((A & B) | (C & D));
427 endmodule
```

yosys> help $_MUX16_

A 16-input MUX gate.

```
Truth table:    A B C D E F G H I J K L M N O P S T U V | Y
                ---------------------------------------+---
                a - - - - - - - - - - - - - - - - 0 0 0 0 | a
                - b - - - - - - - - - - - - - - - 1 0 0 0 | b
                - - c - - - - - - - - - - - - - - 0 1 0 0 | c
                - - - d - - - - - - - - - - - - - 1 1 0 0 | d
```

(continues on next page)

(continued from previous page)

```
                - - - - e - - - - - - - - - - - 0 0 1 0 | e
                - - - - - f - - - - - - - - - - 1 0 1 0 | f
                - - - - - - g - - - - - - - - - 0 1 1 0 | g
                - - - - - - - h - - - - - - - - 1 1 1 0 | h
                - - - - - - - - i - - - - - - - 0 0 0 1 | i
                - - - - - - - - - j - - - - - - 1 0 0 1 | j
                - - - - - - - - - - k - - - - - 0 1 0 1 | k
                - - - - - - - - - - - l - - - - 1 1 0 1 | l
                - - - - - - - - - - - - m - - - 0 0 1 1 | m
                - - - - - - - - - - - - - n - - 1 0 1 1 | n
                - - - - - - - - - - - - - - o - 0 1 1 1 | o
                - - - - - - - - - - - - - - - p 1 1 1 1 | p
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.117: simcells.v

```
336 module \$_MUX16_ (A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, S, T, U, V, Y);
337     input A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, S, T, U, V;
338     output Y;
339     assign Y = V ? U ? T ? (S ? P : O) :
340                            (S ? N : M) :
341                        T ? (S ? L : K) :
342                            (S ? J : I) :
343                    U ? T ? (S ? H : G) :
344                            (S ? F : E) :
345                        T ? (S ? D : C) :
346                            (S ? B : A);
347 endmodule
```

yosys> help $_MUX4_

A 4-input MUX gate.

```
Truth table:    A B C D S T | Y
                ------------+---
                a - - - 0 0 | a
                - b - - 1 0 | b
                - - c - 0 1 | c
                - - - d 1 1 | d
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.118: simcells.v

```
276 module \$_MUX4_ (A, B, C, D, S, T, Y);
277     input A, B, C, D, S, T;
278     output Y;
279     assign Y = T ? (S ? D : C) :
280                    (S ? B : A);
281 endmodule
```

yosys> help $_MUX8_

An 8-input MUX gate.

```
Truth table:    A B C D E F G H S T U | Y
               ----------------------+---
                a - - - - - - - 0 0 0 | a
                - b - - - - - - 1 0 0 | b
                - - c - - - - - 0 1 0 | c
                - - - d - - - - 1 1 0 | d
                - - - - e - - - 0 0 1 | e
                - - - - - f - - 1 0 1 | f
                - - - - - - g - 0 1 1 | g
                - - - - - - - h 1 1 1 | h
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.119: simcells.v

```
301 module \$_MUX8_ (A, B, C, D, E, F, G, H, S, T, U, Y);
302     input A, B, C, D, E, F, G, H, S, T, U;
303     output Y;
304     assign Y = U ? T ? (S ? H : G) :
305                        (S ? F : E) :
306                    T ? (S ? D : C) :
307                        (S ? B : A);
308 endmodule
```

yosys> help $_NMUX_

A 2-input inverting MUX gate.

```
Truth table:    A B S | Y
               -------+---
                0 - 0 | 1
                1 - 0 | 0
                - 0 1 | 1
                - 1 1 | 0
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.120: simcells.v

```
256 module \$_NMUX_ (A, B, S, Y);
257     input A, B, S;
258     output Y;
259     assign Y = S ? !B : !A;
260 endmodule
```

yosys> help **$_OAI3_**

A 3-input Or-And-Invert gate.

```
Truth table:    A B C | Y
               -------+---
                0 0 0 | 1
                0 0 1 | 1
                0 1 0 | 1
                0 1 1 | 0
                1 0 0 | 1
                1 0 1 | 0
                1 1 0 | 1
                1 1 1 | 0
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.121: simcells.v

```
391 module \$_OAI3_ (A, B, C, Y);
392     input A, B, C;
393     output Y;
394     assign Y = ~((A | B) & C);
395 endmodule
```

yosys> help **$_OAI4_**

A 4-input Or-And-Invert gate.

```
Truth table:    A B C D | Y
               ---------+---
                0 0 0 0 | 1
                0 0 0 1 | 1
                0 0 1 0 | 1
                0 0 1 1 | 1
                0 1 0 0 | 1
                0 1 0 1 | 0
                0 1 1 0 | 0
                0 1 1 1 | 0
                1 0 0 0 | 1
                1 0 0 1 | 0
                1 0 1 0 | 0
```

(continues on next page)

(continued from previous page)

```
                1 0 1 1 | 0
                1 1 0 0 | 1
                1 1 0 1 | 0
                1 1 1 0 | 0
                1 1 1 1 | 0
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.122: simcells.v

```
455 module \$_OAI4_ (A, B, C, D, Y);
456     input A, B, C, D;
457     output Y;
458     assign Y = ~((A | B) & (C | D));
459 endmodule
```

yosys> help $_ORNOT_

A 2-input OR-NOT gate.

```
Truth table:    A B | Y
                ----+---
                0 0 | 1
                0 1 | 0
                1 0 | 1
                1 1 | 1
```

Properties

is_evaluable

Simulation model (verilog)

Listing 9.123: simcells.v

```
218 module \$_ORNOT_ (A, B, Y);
219     input A, B;
220     output Y;
221     assign Y = A | (~B);
222 endmodule
```

9.2.3 Flip-flop cells

The cell types *$_DFF_N_* and *$_DFF_P_* represent d-type flip-flops.

Table 9.6: Cell types for basic flip-flops

Verilog	Cell Type
`always @(negedge C) Q <= D`	*$_DFF_N_*
`always @(posedge C) Q <= D`	*$_DFF_P_*

The cell types `$_DFFE_[NP][NP]_` implement d-type flip-flops with enable. The values in the table for these cell types relate to the following Verilog code template.

```
always @(CLK_EDGE C)
   if (EN == EN_LVL)
      Q <= D;
```

Table 9.7: Cell types for gate level logic networks (FFs with enable)

ClkEdge	*EnLvl*	Cell Type
negedge	0	*$_DFFE_NN_*
negedge	1	*$_DFFE_NP_*
posedge	0	*$_DFFE_PN_*
posedge	1	*$_DFFE_PP_*

The cell types `$_DFF_[NP][NP][01]_` implement d-type flip-flops with asynchronous reset. The values in the table for these cell types relate to the following Verilog code template, where `RST_EDGE` is `posedge` if `RST_LVL` if `1`, and `negedge` otherwise.

```
always @(CLK_EDGE C, RST_EDGE R)
   if (R == RST_LVL)
      Q <= RST_VAL;
   else
      Q <= D;
```

The cell types `$_SDFF_[NP][NP][01]_` implement d-type flip-flops with synchronous reset. The values in the table for these cell types relate to the following Verilog code template:

```
always @(CLK_EDGE C)
   if (R == RST_LVL)
      Q <= RST_VAL;
   else
      Q <= D;
```

Table 9.8: Cell types for gate level logic networks (FFs with reset)

ClkEdge	*RstLvl*	*RstVal*	Cell Type
negedge	0	0	*$_DFF_NN0_* , *$_SDFF_NN0_*
negedge	0	1	*$_DFF_NN1_* , *$_SDFF_NN1_*
negedge	1	0	*$_DFF_NP0_* , *$_SDFF_NP0_*
negedge	1	1	*$_DFF_NP1_* , *$_SDFF_NP1_*
posedge	0	0	*$_DFF_PN0_* , *$_SDFF_PN0_*
posedge	0	1	*$_DFF_PN1_* , *$_SDFF_PN1_*
posedge	1	0	*$_DFF_PP0_* , *$_SDFF_PP0_*
posedge	1	1	*$_DFF_PP1_* , *$_SDFF_PP1_*

The cell types `$_DFFE_[NP][NP][01][NP]_` implement d-type flip-flops with asynchronous reset and enable. The values in the table for these cell types relate to the following Verilog code template, where `RST_EDGE` is `posedge` if `RST_LVL` if `1`, and `negedge` otherwise.

```
always @(CLK_EDGE C, RST_EDGE R)
   if (R == RST_LVL)
```

(continues on next page)

(continued from previous page)

```
    Q <= RST_VAL;
  else if (EN == EN_LVL)
    Q <= D;
```

The cell types `$_SDFFE_[NP][NP][01][NP]_` implement d-type flip-flops with synchronous reset and enable, with reset having priority over enable. The values in the table for these cell types relate to the following Verilog code template:

```
always @(CLK_EDGE C)
  if (R == RST_LVL)
    Q <= RST_VAL;
  else if (EN == EN_LVL)
    Q <= D;
```

The cell types `$_SDFFCE_[NP][NP][01][NP]_` implement d-type flip-flops with synchronous reset and enable, with enable having priority over reset. The values in the table for these cell types relate to the following Verilog code template:

```
always @(CLK_EDGE C)
  if (EN == EN_LVL)
    if (R == RST_LVL)
      Q <= RST_VAL;
    else
      Q <= D;
```

Table 9.9: Cell types for gate level logic networks (FFs with reset and enable)

ClkEdge	*RstLvl*	*RstVal*	*EnLvl*	Cell Type
negedge	0	0	0	*$_DFFE_NN0N_* , *$_SDFFE_NN0N_* , *$_SDFFCE_NN0N_*
negedge	0	0	1	*$_DFFE_NN0P_* , *$_SDFFE_NN0P_* , *$_SDFFCE_NN0P_*
negedge	0	1	0	*$_DFFE_NN1N_* , *$_SDFFE_NN1N_* , *$_SDFFCE_NN1N_*
negedge	0	1	1	*$_DFFE_NN1P_* , *$_SDFFE_NN1P_* , *$_SDFFCE_NN1P_*
negedge	1	0	0	*$_DFFE_NP0N_* , *$_SDFFE_NP0N_* , *$_SDFFCE_NP0N_*
negedge	1	0	1	*$_DFFE_NP0P_* , *$_SDFFE_NP0P_* , *$_SDFFCE_NP0P_*
negedge	1	1	0	*$_DFFE_NP1N_* , *$_SDFFE_NP1N_* , *$_SDFFCE_NP1N_*
negedge	1	1	1	*$_DFFE_NP1P_* , *$_SDFFE_NP1P_* , *$_SDFFCE_NP1P_*
posedge	0	0	0	*$_DFFE_PN0N_* , *$_SDFFE_PN0N_* , *$_SDFFCE_PN0N_*
posedge	0	0	1	*$_DFFE_PN0P_* , *$_SDFFE_PN0P_* , *$_SDFFCE_PN0P_*
posedge	0	1	0	*$_DFFE_PN1N_* , *$_SDFFE_PN1N_* , *$_SDFFCE_PN1N_*
posedge	0	1	1	*$_DFFE_PN1P_* , *$_SDFFE_PN1P_* , *$_SDFFCE_PN1P_*
posedge	1	0	0	*$_DFFE_PP0N_* , *$_SDFFE_PP0N_* , *$_SDFFCE_PP0N_*
posedge	1	0	1	*$_DFFE_PP0P_* , *$_SDFFE_PP0P_* , *$_SDFFCE_PP0P_*
posedge	1	1	0	*$_DFFE_PP1N_* , *$_SDFFE_PP1N_* , *$_SDFFCE_PP1N_*
posedge	1	1	1	*$_DFFE_PP1P_* , *$_SDFFE_PP1P_* , *$_SDFFCE_PP1P_*

The cell types `$_DFFSR_[NP][NP][NP]_` implement d-type flip-flops with asynchronous set and reset. The values in the table for these cell types relate to the following Verilog code template, where `RST_EDGE` is `posedge` if `RST_LVL` if `1`, `negedge` otherwise, and `SET_EDGE` is `posedge` if `SET_LVL` if `1`, `negedge` otherwise.

When both set and reset are active, the state and output is undefined. The Verilog code model does not correspond to this due to limitations of synthesizable Verilog.

```
always @(CLK_EDGE C, RST_EDGE R, SET_EDGE S)
   if (R == RST_LVL)
      Q <= 0;
   else if (S == SET_LVL)
      Q <= 1;
   else
      Q <= D;
```

Table 9.10: Cell types for gate level logic networks (FFs with set and reset)

ClkEdge	*SetLvl*	*RstLvl*	Cell Type
negedge	0	0	*$_DFFSR_NNN_*
negedge	0	1	*$_DFFSR_NNP_*
negedge	1	0	*$_DFFSR_NPN_*
negedge	1	1	*$_DFFSR_NPP_*
posedge	0	0	*$_DFFSR_PNN_*
posedge	0	1	*$_DFFSR_PNP_*
posedge	1	0	*$_DFFSR_PPN_*
posedge	1	1	*$_DFFSR_PPP_*

The cell types `$_DFFSRE_[NP][NP][NP][NP]_` implement d-type flip-flops with asynchronous set and reset and enable. The values in the table for these cell types relate to the following Verilog code template, where `RST_EDGE` is `posedge` if `RST_LVL` if `1`, `negedge` otherwise, and `SET_EDGE` is `posedge` if `SET_LVL` if `1`, `negedge` otherwise.

When both set and reset are active, the state and output is undefined. The Verilog code model does not correspond to this due to limitations of synthesizable Verilog.

```
always @(CLK_EDGE C, RST_EDGE R, SET_EDGE S)
   if (R == RST_LVL)
      Q <= 0;
   else if (S == SET_LVL)
      Q <= 1;
   else if (E == EN_LVL)
      Q <= D;
```

Table 9.11: Cell types for gate level logic networks (FFs with set and reset and enable)

ClkEdge	*SetLvl*	*RstLvl*	*EnLvl*	Cell Type
negedge	0	0	0	*$_DFFSRE_NNNN_*
negedge	0	0	1	*$_DFFSRE_NNNP_*
negedge	0	1	0	*$_DFFSRE_NNPN_*
negedge	0	1	1	*$_DFFSRE_NNPP_*
negedge	1	0	0	*$_DFFSRE_NPNN_*
negedge	1	0	1	*$_DFFSRE_NPNP_*
negedge	1	1	0	*$_DFFSRE_NPPN_*
negedge	1	1	1	*$_DFFSRE_NPPP_*
posedge	0	0	0	*$_DFFSRE_PNNN_*
posedge	0	0	1	*$_DFFSRE_PNNP_*
posedge	0	1	0	*$_DFFSRE_PNPN_*
posedge	0	1	1	*$_DFFSRE_PNPP_*
posedge	1	0	0	*$_DFFSRE_PPNN_*
posedge	1	0	1	*$_DFFSRE_PPNP_*
posedge	1	1	0	*$_DFFSRE_PPPN_*
posedge	1	1	1	*$_DFFSRE_PPPP_*

Todo

flip-flops with async load, `$_ALDFFE?_[NP]{2,3}_`

```
yosys> help $_ALDFFE_NNN_
```

A negative edge D-type flip-flop with negative polarity async load and negative polarity clock enable.

```
Truth table:    D C L AD E | Q
               ------------+---
                - - 0 a  - | a
                d \ - -  0 | d
                - - - -  - | q
```

Simulation model (verilog)

Listing 9.124: simcells.v

```
1420 module \$_ALDFFE_NNN_ (D, C, L, AD, E, Q);
1421     input D, C, L, AD, E;
1422     output reg Q;
1423     always @(negedge C or negedge L) begin
1424         if (L == 0)
1425             Q <= AD;
1426         else if (E == 0)
1427             Q <= D;
1428     end
1429 endmodule
```

```
yosys> help $_ALDFFE_NNP_
```

A negative edge D-type flip-flop with negative polarity async load and positive polarity clock enable.

```
Truth table:    D C L AD E | Q
               ------------+---
                - - 0 a  - | a
                d \ - -  1 | d
                - - - -  - | q
```

Simulation model (verilog)

Listing 9.125: simcells.v

```
module \$_ALDFFE_NNP_ (D, C, L, AD, E, Q);
    input D, C, L, AD, E;
    output reg Q;
    always @(negedge C or negedge L) begin
        if (L == 0)
            Q <= AD;
        else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help **$_ALDFFE_NPN_**

A negative edge D-type flip-flop with positive polarity async load and negative polarity clock enable.

```
Truth table:    D C L AD E | Q
               ------------+---
                - - 1 a  - | a
                d \ - -  0 | d
                - - - -  - | q
```

Simulation model (verilog)

Listing 9.126: simcells.v

```
module \$_ALDFFE_NPN_ (D, C, L, AD, E, Q);
    input D, C, L, AD, E;
    output reg Q;
    always @(negedge C or posedge L) begin
        if (L == 1)
            Q <= AD;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help **$_ALDFFE_NPP_**

A negative edge D-type flip-flop with positive polarity async load and positive polarity clock enable.

```
Truth table:    D C L AD E | Q
               ------------+---
                - - 1 a  - | a
```

(continues on next page)

(continued from previous page)

```
               d \ - -  1 | d
               - - - -  - | q
```

Simulation model (verilog)

Listing 9.127: simcells.v

```
1495 module \$_ALDFFE_NPP_ (D, C, L, AD, E, Q);
1496     input D, C, L, AD, E;
1497     output reg Q;
1498     always @(negedge C or posedge L) begin
1499         if (L == 1)
1500             Q <= AD;
1501         else if (E == 1)
1502             Q <= D;
1503     end
1504 endmodule
```

yosys> help **$_ALDFFE_PNN_**

A positive edge D-type flip-flop with negative polarity async load and negative polarity clock enable.

```
Truth table:    D C L AD E | Q
               ------------+---
                - - 0 a  - | a
                d / - -  0 | d
                - - - -  - | q
```

Simulation model (verilog)

Listing 9.128: simcells.v

```
1520 module \$_ALDFFE_PNN_ (D, C, L, AD, E, Q);
1521     input D, C, L, AD, E;
1522     output reg Q;
1523     always @(posedge C or negedge L) begin
1524         if (L == 0)
1525             Q <= AD;
1526         else if (E == 0)
1527             Q <= D;
1528     end
1529 endmodule
```

yosys> help **$_ALDFFE_PNP_**

A positive edge D-type flip-flop with negative polarity async load and positive polarity clock enable.

```
Truth table:    D C L AD E | Q
               ------------+---
                - - 0 a  - | a
                d / - -  1 | d
                - - - -  - | q
```

Simulation model (verilog)

Listing 9.129: simcells.v

```
module \$_ALDFFE_PNP_ (D, C, L, AD, E, Q);
    input D, C, L, AD, E;
    output reg Q;
    always @(posedge C or negedge L) begin
        if (L == 0)
            Q <= AD;
        else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_ALDFFE_PPN_

A positive edge D-type flip-flop with positive polarity async load and negative polarity clock enable.

```
Truth table:    D C L AD E | Q
               ------------+---
                - - 1 a  - | a
                d / - -  0 | d
                - - - -  - | q
```

Simulation model (verilog)

Listing 9.130: simcells.v

```
module \$_ALDFFE_PPN_ (D, C, L, AD, E, Q);
    input D, C, L, AD, E;
    output reg Q;
    always @(posedge C or posedge L) begin
        if (L == 1)
            Q <= AD;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_ALDFFE_PPP_

A positive edge D-type flip-flop with positive polarity async load and positive polarity clock enable.

```
Truth table:    D C L AD E | Q
               ------------+---
                - - 1 a  - | a
                d / - -  1 | d
                - - - -  - | q
```

Simulation model (verilog)

Listing 9.131: simcells.v

```
module \$_ALDFFE_PPP_ (D, C, L, AD, E, Q);
    input D, C, L, AD, E;
    output reg Q;
```

(continues on next page)

(continued from previous page)

```
1598     always @(posedge C or posedge L) begin
1599         if (L == 1)
1600             Q <= AD;
1601         else if (E == 1)
1602             Q <= D;
1603     end
1604 endmodule
```

yosys> help $_ALDFF_NN_

A negative edge D-type flip-flop with negative polarity async load.

```
Truth table:    D C L AD | Q
               ----------+---
                - - 0 a  | a
                d \ - -  | d
                - - - -  | q
```

Simulation model (verilog)

Listing 9.132: simcells.v

```
1323 module \$_ALDFF_NN_ (D, C, L, AD, Q);
1324     input D, C, L, AD;
1325     output reg Q;
1326     always @(negedge C or negedge L) begin
1327         if (L == 0)
1328             Q <= AD;
1329         else
1330             Q <= D;
1331     end
1332 endmodule
```

yosys> help $_ALDFF_NP_

A negative edge D-type flip-flop with positive polarity async load.

```
Truth table:    D C L AD | Q
               ----------+---
                - - 1 a  | a
                d \ - -  | d
                - - - -  | q
```

Simulation model (verilog)

Listing 9.133: simcells.v

```
1347 module \$_ALDFF_NP_ (D, C, L, AD, Q);
1348     input D, C, L, AD;
1349     output reg Q;
1350     always @(negedge C or posedge L) begin
1351         if (L == 1)
1352             Q <= AD;
1353         else
```

(continues on next page)

(continued from previous page)

```
            Q <= D;
    end
endmodule
```

yosys> help $_ALDFF_PN_

A positive edge D-type flip-flop with negative polarity async load.

```
Truth table:    D C L AD | Q
               ----------+---
                - - 0 a  | a
                d / - -  | d
                - - - -  | q
```

Simulation model (verilog)

Listing 9.134: simcells.v

```
module \$_ALDFF_PN_ (D, C, L, AD, Q);
    input D, C, L, AD;
    output reg Q;
    always @(posedge C or negedge L) begin
        if (L == 0)
            Q <= AD;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_ALDFF_PP_

A positive edge D-type flip-flop with positive polarity async load.

```
Truth table:    D C L AD | Q
               ----------+---
                - - 1 a  | a
                d / - -  | d
                - - - -  | q
```

Simulation model (verilog)

Listing 9.135: simcells.v

```
module \$_ALDFF_PP_ (D, C, L, AD, Q);
    input D, C, L, AD;
    output reg Q;
    always @(posedge C or posedge L) begin
        if (L == 1)
            Q <= AD;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_DFFE_NNON_

A negative edge D-type flip-flop with negative polarity reset and negative polarity clock enable.

```
Truth table:    D C R E | Q
                ---------+---
                - - 0 - | 0
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.136: simcells.v

```
module \$_DFFE_NNON_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(negedge C or negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFE_NNOP_

A negative edge D-type flip-flop with negative polarity reset and positive polarity clock enable.

```
Truth table:    D C R E | Q
                ---------+---
                - - 0 - | 0
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.137: simcells.v

```
module \$_DFFE_NNOP_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(negedge C or negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFE_NN1N_

A negative edge D-type flip-flop with negative polarity set and negative polarity clock enable.

```
Truth table:    D C R E | Q
                ---------+---
```

(continues on next page)

(continued from previous page)

```
                - - 0 - | 1
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.138: simcells.v

```
974 module \$_DFFE_NN1N_ (D, C, R, E, Q);
975     input D, C, R, E;
976     output reg Q;
977     always @(negedge C or negedge R) begin
978         if (R == 0)
979             Q <= 1;
980         else if (E == 0)
981             Q <= D;
982     end
983 endmodule
```

yosys> help $_DFFE_NN1P_

A negative edge D-type flip-flop with negative polarity set and positive polarity clock enable.

```
Truth table:    D C R E | Q
                ---------+---
                - - 0 - | 1
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.139: simcells.v

```
999  module \$_DFFE_NN1P_ (D, C, R, E, Q);
1000     input D, C, R, E;
1001     output reg Q;
1002     always @(negedge C or negedge R) begin
1003         if (R == 0)
1004             Q <= 1;
1005         else if (E == 1)
1006             Q <= D;
1007     end
1008 endmodule
```

yosys> help $_DFFE_NN_

A negative edge D-type flip-flop with negative polarity enable.

```
Truth table:    D C E | Q
                -------+---
                d \ 0 | d
                - - - | q
```

Simulation model (verilog)

Listing 9.140: simcells.v

```
650 module \$_DFFE_NN_ (D, C, E, Q);
651     input D, C, E;
652     output reg Q;
653     always @(negedge C) begin
654         if (!E) Q <= D;
655     end
656 endmodule
```

yosys> help $_DFFE_NPON_

A negative edge D-type flip-flop with positive polarity reset and negative polarity clock enable.

```
Truth table:    D C R E | Q
                ---------+---
                - - 1 - | 0
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.141: simcells.v

```
1024 module \$_DFFE_NPON_ (D, C, R, E, Q);
1025     input D, C, R, E;
1026     output reg Q;
1027     always @(negedge C or posedge R) begin
1028         if (R == 1)
1029             Q <= 0;
1030         else if (E == 0)
1031             Q <= D;
1032     end
1033 endmodule
```

yosys> help $_DFFE_NPOP_

A negative edge D-type flip-flop with positive polarity reset and positive polarity clock enable.

```
Truth table:    D C R E | Q
                ---------+---
                - - 1 - | 0
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.142: simcells.v

```
1049 module \$_DFFE_NPOP_ (D, C, R, E, Q);
1050     input D, C, R, E;
1051     output reg Q;
1052     always @(negedge C or posedge R) begin
1053         if (R == 1)
1054             Q <= 0;
```

(continues on next page)

(continued from previous page)

```
1055        else if (E == 1)
1056            Q <= D;
1057    end
1058 endmodule
```

yosys> help $_DFFE_NP1N_

A negative edge D-type flip-flop with positive polarity set and negative polarity clock enable.

```
Truth table:    D C R E | Q
               ---------+---
                - - 1 - | 1
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.143: simcells.v

```
1074 module \$_DFFE_NP1N_ (D, C, R, E, Q);
1075     input D, C, R, E;
1076     output reg Q;
1077     always @(negedge C or posedge R) begin
1078         if (R == 1)
1079             Q <= 1;
1080         else if (E == 0)
1081             Q <= D;
1082     end
1083 endmodule
```

yosys> help $_DFFE_NP1P_

A negative edge D-type flip-flop with positive polarity set and positive polarity clock enable.

```
Truth table:    D C R E | Q
               ---------+---
                - - 1 - | 1
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.144: simcells.v

```
1099 module \$_DFFE_NP1P_ (D, C, R, E, Q);
1100     input D, C, R, E;
1101     output reg Q;
1102     always @(negedge C or posedge R) begin
1103         if (R == 1)
1104             Q <= 1;
1105         else if (E == 1)
1106             Q <= D;
1107     end
1108 endmodule
```

yosys> help $_DFFE_NP_

A negative edge D-type flip-flop with positive polarity enable.

```
Truth table:    D C E | Q
               -------+---
                d \ 1 | d
                - - - | q
```

Simulation model (verilog)

Listing 9.145: simcells.v

```
module \$_DFFE_NP_ (D, C, E, Q);
    input D, C, E;
    output reg Q;
    always @(negedge C) begin
        if (E) Q <= D;
    end
endmodule
```

yosys> help $_DFFE_PNON_

A positive edge D-type flip-flop with negative polarity reset and negative polarity clock enable.

```
Truth table:    D C R E | Q
               ---------+---
                - - 0 - | 0
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.146: simcells.v

```
module \$_DFFE_PNON_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(posedge C or negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFE_PNOP_

A positive edge D-type flip-flop with negative polarity reset and positive polarity clock enable.

```
Truth table:    D C R E | Q
               ---------+---
                - - 0 - | 0
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.147: simcells.v

```
module \$_DFFE_PNOP_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(posedge C or negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFE_PN1N_

A positive edge D-type flip-flop with negative polarity set and negative polarity clock enable.

```
Truth table:    D C R E | Q
               ---------+---
                - - 0 - | 1
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.148: simcells.v

```
module \$_DFFE_PN1N_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(posedge C or negedge R) begin
        if (R == 0)
            Q <= 1;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFE_PN1P_

A positive edge D-type flip-flop with negative polarity set and positive polarity clock enable.

```
Truth table:    D C R E | Q
               ---------+---
                - - 0 - | 1
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.149: simcells.v

```
module \$_DFFE_PN1P_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
```

(continues on next page)

(continued from previous page)

```
    always @(posedge C or negedge R) begin
        if (R == 0)
            Q <= 1;
        else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFE_PN_

A positive edge D-type flip-flop with negative polarity enable.

```
Truth table:    D C E | Q
               -------+---
                d / 0 | d
                - - - | q
```

Simulation model (verilog)

Listing 9.150: simcells.v

```
module \$_DFFE_PN_ (D, C, E, Q);
    input D, C, E;
    output reg Q;
    always @(posedge C) begin
        if (!E) Q <= D;
    end
endmodule
```

yosys> help $_DFFE_PPON_

A positive edge D-type flip-flop with positive polarity reset and negative polarity clock enable.

```
Truth table:    D C R E | Q
               ---------+---
                - - 1 - | 0
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.151: simcells.v

```
module \$_DFFE_PPON_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(posedge C or posedge R) begin
        if (R == 1)
            Q <= 0;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFE_PP0P_

A positive edge D-type flip-flop with positive polarity reset and positive polarity clock enable.

```
Truth table:    D C R E | Q
                ---------+---
                - - 1 - | 0
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.152: simcells.v

```
1249 module \$_DFFE_PP0P_ (D, C, R, E, Q);
1250     input D, C, R, E;
1251     output reg Q;
1252     always @(posedge C or posedge R) begin
1253         if (R == 1)
1254             Q <= 0;
1255         else if (E == 1)
1256             Q <= D;
1257     end
1258 endmodule
```

yosys> help $_DFFE_PP1N_

A positive edge D-type flip-flop with positive polarity set and negative polarity clock enable.

```
Truth table:    D C R E | Q
                ---------+---
                - - 1 - | 1
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.153: simcells.v

```
1274 module \$_DFFE_PP1N_ (D, C, R, E, Q);
1275     input D, C, R, E;
1276     output reg Q;
1277     always @(posedge C or posedge R) begin
1278         if (R == 1)
1279             Q <= 1;
1280         else if (E == 0)
1281             Q <= D;
1282     end
1283 endmodule
```

yosys> help $_DFFE_PP1P_

A positive edge D-type flip-flop with positive polarity set and positive polarity clock enable.

```
Truth table:    D C R E | Q
                ---------+---
```

(continues on next page)

(continued from previous page)

```
                - - 1 - | 1
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.154: simcells.v

```
1299 module \$_DFFE_PP1P_ (D, C, R, E, Q);
1300     input D, C, R, E;
1301     output reg Q;
1302     always @(posedge C or posedge R) begin
1303         if (R == 1)
1304             Q <= 1;
1305         else if (E == 1)
1306             Q <= D;
1307     end
1308 endmodule
```

yosys> help $_DFFE_PP_

A positive edge D-type flip-flop with positive polarity enable.

```
Truth table:    D C E | Q
                -------+---
                d / 1 | d
                - - - | q
```

Simulation model (verilog)

Listing 9.155: simcells.v

```
710 module \$_DFFE_PP_ (D, C, E, Q);
711     input D, C, E;
712     output reg Q;
713     always @(posedge C) begin
714         if (E) Q <= D;
715     end
716 endmodule
```

yosys> help $_DFFSRE_NNNN_

A negative edge D-type flip-flop with negative polarity set, negative polarity reset and negative polarity clock enable.

```
Truth table:    C S R E D | Q
                -----------+---
                - 0 0 - - | x
                - - 0 - - | 0
                - 0 - - - | 1
                \ - - 0 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.156: simcells.v

```
module \$_DFFSRE_NNNN_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
    output reg Q;
    always @(negedge C, negedge S, negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
            else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_NNNP_

A negative edge D-type flip-flop with negative polarity set, negative polarity reset and positive polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 0 0 - - | x
                - - 0 - - | 0
                - 0 - - - | 1
                \ - - 1 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.157: simcells.v

```
module \$_DFFSRE_NNNP_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
    output reg Q;
    always @(negedge C, negedge S, negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
            else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_NNPN_

A negative edge D-type flip-flop with negative polarity set, positive polarity reset and negative polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 0 1 - - | x
                - - 1 - - | 0
                - 0 - - - | 1
```

(continues on next page)

(continued from previous page)

```
                \ - - 0 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.158: simcells.v

```
module \$_DFFSRE_NNPN_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
    output reg Q;
    always @(negedge C, negedge S, posedge R) begin
        if (R == 1)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
            else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_NNPP_

A negative edge D-type flip-flop with negative polarity set, positive polarity reset and positive polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 0 1 - - | x
                - - 1 - - | 0
                - 0 - - - | 1
                \ - - 1 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.159: simcells.v

```
module \$_DFFSRE_NNPP_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
    output reg Q;
    always @(negedge C, negedge S, posedge R) begin
        if (R == 1)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
            else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_NPNN_

A negative edge D-type flip-flop with positive polarity set, negative polarity reset and negative polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 1 0 - - | x
                - - 0 - - | 0
                - 1 - - - | 1
                \ - - 0 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.160: simcells.v

```
1970 module \$_DFFSRE_NPNN_ (C, S, R, E, D, Q);
1971     input C, S, R, E, D;
1972     output reg Q;
1973     always @(negedge C, posedge S, negedge R) begin
1974         if (R == 0)
1975             Q <= 0;
1976         else if (S == 1)
1977             Q <= 1;
1978             else if (E == 0)
1979             Q <= D;
1980     end
1981 endmodule
```

yosys> help $_DFFSRE_NPNP_

A negative edge D-type flip-flop with positive polarity set, negative polarity reset and positive polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 1 0 - - | x
                - - 0 - - | 0
                - 1 - - - | 1
                \ - - 1 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.161: simcells.v

```
1999 module \$_DFFSRE_NPNP_ (C, S, R, E, D, Q);
2000     input C, S, R, E, D;
2001     output reg Q;
2002     always @(negedge C, posedge S, negedge R) begin
2003         if (R == 0)
2004             Q <= 0;
2005         else if (S == 1)
2006             Q <= 1;
2007             else if (E == 1)
2008             Q <= D;
2009     end
2010 endmodule
```

yosys> help $_DFFSRE_NPPN_

A negative edge D-type flip-flop with positive polarity set, positive polarity reset and negative polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 1 1 - - | x
                - - 1 - - | 0
                - 1 - - - | 1
                \ - - 0 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.162: simcells.v

```
module \$_DFFSRE_NPPN_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
    output reg Q;
    always @(negedge C, posedge S, posedge R) begin
        if (R == 1)
            Q <= 0;
        else if (S == 1)
            Q <= 1;
            else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_NPPP_

A negative edge D-type flip-flop with positive polarity set, positive polarity reset and positive polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 1 1 - - | x
                - - 1 - - | 0
                - 1 - - - | 1
                \ - - 1 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.163: simcells.v

```
module \$_DFFSRE_NPPP_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
    output reg Q;
    always @(negedge C, posedge S, posedge R) begin
        if (R == 1)
            Q <= 0;
        else if (S == 1)
            Q <= 1;
```

(continues on next page)

(continued from previous page)

```
            else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_PNNN_

A positive edge D-type flip-flop with negative polarity set, negative polarity reset and negative polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 0 0 - - | x
                - - 0 - - | 0
                - 0 - - - | 1
                / - - 0 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.164: simcells.v

```
module \$_DFFSRE_PNNN_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
    output reg Q;
    always @(posedge C, negedge S, negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
            else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_PNNP_

A positive edge D-type flip-flop with negative polarity set, negative polarity reset and positive polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 0 0 - - | x
                - - 0 - - | 0
                - 0 - - - | 1
                / - - 1 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.165: simcells.v

```
module \$_DFFSRE_PNNP_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
```

(continues on next page)

(continued from previous page)

```
    output reg Q;
    always @(posedge C, negedge S, negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
            else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_PNPN_

A positive edge D-type flip-flop with negative polarity set, positive polarity reset and negative polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 0 1 - - | x
                - - 1 - - | 0
                - 0 - - - | 1
                / - - 0 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.166: simcells.v

```
module \$_DFFSRE_PNPN_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
    output reg Q;
    always @(posedge C, negedge S, posedge R) begin
        if (R == 1)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
            else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_PNPP_

A positive edge D-type flip-flop with negative polarity set, positive polarity reset and positive polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 0 1 - - | x
                - - 1 - - | 0
                - 0 - - - | 1
                / - - 1 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.167: simcells.v

```
2173 module \$_DFFSRE_PNPP_ (C, S, R, E, D, Q);
2174     input C, S, R, E, D;
2175     output reg Q;
2176     always @(posedge C, negedge S, posedge R) begin
2177         if (R == 1)
2178             Q <= 0;
2179         else if (S == 0)
2180             Q <= 1;
2181             else if (E == 1)
2182             Q <= D;
2183     end
2184 endmodule
```

yosys> help $_DFFSRE_PPNN_

A positive edge D-type flip-flop with positive polarity set, negative polarity reset and negative polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 1 0 - - | x
                - - 0 - - | 0
                - 1 - - - | 1
                / - - 0 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.168: simcells.v

```
2202 module \$_DFFSRE_PPNN_ (C, S, R, E, D, Q);
2203     input C, S, R, E, D;
2204     output reg Q;
2205     always @(posedge C, posedge S, negedge R) begin
2206         if (R == 0)
2207             Q <= 0;
2208         else if (S == 1)
2209             Q <= 1;
2210             else if (E == 0)
2211             Q <= D;
2212     end
2213 endmodule
```

yosys> help $_DFFSRE_PPNP_

A positive edge D-type flip-flop with positive polarity set, negative polarity reset and positive polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 1 0 - - | x
                - - 0 - - | 0
                - 1 - - - | 1
```

(continues on next page)

(continued from previous page)

```
               / - - 1 d | d
               - - - - - | q
```

Simulation model (verilog)

Listing 9.169: simcells.v

```
module \$_DFFSRE_PPNP_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
    output reg Q;
    always @(posedge C, posedge S, negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (S == 1)
            Q <= 1;
            else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_PPPN_

A positive edge D-type flip-flop with positive polarity set, positive polarity reset and negative polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 1 1 - - | x
                - - 1 - - | 0
                - 1 - - - | 1
                / - - 0 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.170: simcells.v

```
module \$_DFFSRE_PPPN_ (C, S, R, E, D, Q);
    input C, S, R, E, D;
    output reg Q;
    always @(posedge C, posedge S, posedge R) begin
        if (R == 1)
            Q <= 0;
        else if (S == 1)
            Q <= 1;
            else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSRE_PPPP_

A positive edge D-type flip-flop with positive polarity set, positive polarity reset and positive polarity clock enable.

```
Truth table:    C S R E D | Q
               -----------+---
                - 1 1 - - | x
                - - 1 - - | 0
                - 1 - - - | 1
                / - - 1 d | d
                - - - - - | q
```

Simulation model (verilog)

Listing 9.171: simcells.v

```
2289 module \$_DFFSRE_PPPP_ (C, S, R, E, D, Q);
2290     input C, S, R, E, D;
2291     output reg Q;
2292     always @(posedge C, posedge S, posedge R) begin
2293         if (R == 1)
2294             Q <= 0;
2295         else if (S == 1)
2296             Q <= 1;
2297             else if (E == 1)
2298             Q <= D;
2299     end
2300 endmodule
```

yosys> help $_DFFSR_NNN_

A negative edge D-type flip-flop with negative polarity set and negative polarity reset.

```
Truth table:    C S R D | Q
               ---------+---
                - 0 0 - | x
                - - 0 - | 0
                - 0 - - | 1
                \ - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.172: simcells.v

```
1622 module \$_DFFSR_NNN_ (C, S, R, D, Q);
1623     input C, S, R, D;
1624     output reg Q;
1625     always @(negedge C, negedge S, negedge R) begin
1626         if (R == 0)
1627             Q <= 0;
1628         else if (S == 0)
1629             Q <= 1;
1630         else
1631             Q <= D;
1632     end
1633 endmodule
```

yosys> help $_DFFSR_NNP_

A negative edge D-type flip-flop with negative polarity set and positive polarity reset.

```
Truth table:    C S R D | Q
               ---------+---
                - 0 1 - | x
                - - 1 - | 0
                - 0 - - | 1
                \ - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.173: simcells.v

```
module \$_DFFSR_NNP_ (C, S, R, D, Q);
    input C, S, R, D;
    output reg Q;
    always @(negedge C, negedge S, posedge R) begin
        if (R == 1)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSR_NPN_

A negative edge D-type flip-flop with positive polarity set and negative polarity reset.

```
Truth table:    C S R D | Q
               ---------+---
                - 1 0 - | x
                - - 0 - | 0
                - 1 - - | 1
                \ - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.174: simcells.v

```
module \$_DFFSR_NPN_ (C, S, R, D, Q);
    input C, S, R, D;
    output reg Q;
    always @(negedge C, posedge S, negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (S == 1)
            Q <= 1;
        else
            Q <= D;
```

(continues on next page)

(continued from previous page)

```
1690     end
1691 endmodule
```

yosys> help $_DFFSR_NPP_

A negative edge D-type flip-flop with positive polarity set and positive polarity reset.

```
Truth table:    C S R D | Q
                ---------+---
                - 1 1 - | x
                - - 1 - | 0
                - 1 - - | 1
                \ - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.175: simcells.v

```
1709 module \$_DFFSR_NPP_ (C, S, R, D, Q);
1710     input C, S, R, D;
1711     output reg Q;
1712     always @(negedge C, posedge S, posedge R) begin
1713         if (R == 1)
1714             Q <= 0;
1715         else if (S == 1)
1716             Q <= 1;
1717         else
1718             Q <= D;
1719     end
1720 endmodule
```

yosys> help $_DFFSR_PNN_

A positive edge D-type flip-flop with negative polarity set and negative polarity reset.

```
Truth table:    C S R D | Q
                ---------+---
                - 0 0 - | x
                - - 0 - | 0
                - 0 - - | 1
                / - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.176: simcells.v

```
1738 module \$_DFFSR_PNN_ (C, S, R, D, Q);
1739     input C, S, R, D;
1740     output reg Q;
1741     always @(posedge C, negedge S, negedge R) begin
1742         if (R == 0)
1743             Q <= 0;
```

(continues on next page)

(continued from previous page)

```
        else if (S == 0)
            Q <= 1;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSR_PNP_

A positive edge D-type flip-flop with negative polarity set and positive polarity reset.

```
Truth table:    C S R D | Q
               ---------+---
                - 0 1 - | x
                - - 1 - | 0
                - 0 - - | 1
                / - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.177: simcells.v

```
module \$_DFFSR_PNP_ (C, S, R, D, Q);
    input C, S, R, D;
    output reg Q;
    always @(posedge C, negedge S, posedge R) begin
        if (R == 1)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSR_PPN_

A positive edge D-type flip-flop with positive polarity set and negative polarity reset.

```
Truth table:    C S R D | Q
               ---------+---
                - 1 0 - | x
                - - 0 - | 0
                - 1 - - | 1
                / - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.178: simcells.v

```
module \$_DFFSR_PPN_ (C, S, R, D, Q);
    input C, S, R, D;
```

(continues on next page)

(continued from previous page)

```
    output reg Q;
    always @(posedge C, posedge S, negedge R) begin
        if (R == 0)
            Q <= 0;
        else if (S == 1)
            Q <= 1;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_DFFSR_PPP_

A positive edge D-type flip-flop with positive polarity set and positive polarity reset.

```
Truth table:    C S R D | Q
               ---------+---
                - 1 1 - | x
                - - 1 - | 0
                - 1 - - | 1
                / - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.179: simcells.v

```
module \$_DFFSR_PPP_ (C, S, R, D, Q);
    input C, S, R, D;
    output reg Q;
    always @(posedge C, posedge S, posedge R) begin
        if (R == 1)
            Q <= 0;
        else if (S == 1)
            Q <= 1;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_DFF_NN0_

A negative edge D-type flip-flop with negative polarity reset.

```
Truth table:    D C R | Q
               -------+---
                - - 0 | 0
                d \ - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.180: simcells.v

```
731 module \$_DFF_NN0_ (D, C, R, Q);
732     input D, C, R;
733     output reg Q;
734     always @(negedge C or negedge R) begin
735         if (R == 0)
736             Q <= 0;
737         else
738             Q <= D;
739     end
740 endmodule
```

yosys> help $_DFF_NN1_

A negative edge D-type flip-flop with negative polarity set.

```
Truth table:    D C R | Q
               -------+---
                - - 0 | 1
                d \ - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.181: simcells.v

```
755 module \$_DFF_NN1_ (D, C, R, Q);
756     input D, C, R;
757     output reg Q;
758     always @(negedge C or negedge R) begin
759         if (R == 0)
760             Q <= 1;
761         else
762             Q <= D;
763     end
764 endmodule
```

yosys> help $_DFF_NP0_

A negative edge D-type flip-flop with positive polarity reset.

```
Truth table:    D C R | Q
               -------+---
                - - 1 | 0
                d \ - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.182: simcells.v

```
779 module \$_DFF_NP0_ (D, C, R, Q);
780     input D, C, R;
781     output reg Q;
```

(continues on next page)

(continued from previous page)

```
    always @(negedge C or posedge R) begin
        if (R == 1)
            Q <= 0;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_DFF_NP1_

A negative edge D-type flip-flop with positive polarity set.

```
Truth table:    D C R | Q
               -------+---
                - - 1 | 1
                d \ - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.183: simcells.v

```
module \$_DFF_NP1_ (D, C, R, Q);
    input D, C, R;
    output reg Q;
    always @(negedge C or posedge R) begin
        if (R == 1)
            Q <= 1;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_DFF_N_

A negative edge D-type flip-flop.

```
Truth table:    D C | Q
               -----+---
                d \ | d
                - - | q
```

Simulation model (verilog)

Listing 9.184: simcells.v

```
module \$_DFF_N_ (D, C, Q);
    input D, C;
    output reg Q;
    always @(negedge C) begin
        Q <= D;
    end
endmodule
```

yosys> help $_DFF_PN0_

A positive edge D-type flip-flop with negative polarity reset.

```
Truth table:    D C R | Q
               -------+---
                - - 0 | 0
                d / - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.185: simcells.v

```
827 module \$_DFF_PN0_ (D, C, R, Q);
828     input D, C, R;
829     output reg Q;
830     always @(posedge C or negedge R) begin
831         if (R == 0)
832             Q <= 0;
833         else
834             Q <= D;
835     end
836 endmodule
```

yosys> help $_DFF_PN1_

A positive edge D-type flip-flop with negative polarity set.

```
Truth table:    D C R | Q
               -------+---
                - - 0 | 1
                d / - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.186: simcells.v

```
851 module \$_DFF_PN1_ (D, C, R, Q);
852     input D, C, R;
853     output reg Q;
854     always @(posedge C or negedge R) begin
855         if (R == 0)
856             Q <= 1;
857         else
858             Q <= D;
859     end
860 endmodule
```

yosys> help $_DFF_PP0_

A positive edge D-type flip-flop with positive polarity reset.

```
Truth table:    D C R | Q
               -------+---
```

(continues on next page)

(continued from previous page)

```
                - - 1 | 0
                d / - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.187: simcells.v

```
module \$_DFF_PP0_ (D, C, R, Q);
    input D, C, R;
    output reg Q;
    always @(posedge C or posedge R) begin
        if (R == 1)
            Q <= 0;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_DFF_PP1_

A positive edge D-type flip-flop with positive polarity set.

```
Truth table:    D C R | Q
               -------+---
                - - 1 | 1
                d / - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.188: simcells.v

```
module \$_DFF_PP1_ (D, C, R, Q);
    input D, C, R;
    output reg Q;
    always @(posedge C or posedge R) begin
        if (R == 1)
            Q <= 1;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_DFF_P_

A positive edge D-type flip-flop.

```
Truth table:    D C | Q
               -----+---
                d / | d
                - - | q
```

Simulation model (verilog)

Listing 9.189: simcells.v

```
630 module \$_DFF_P_ (D, C, Q);
631     input D, C;
632     output reg Q;
633     always @(posedge C) begin
634         Q <= D;
635     end
636 endmodule
```

yosys> help $_FF_

A D-type flip-flop that is clocked from the implicit global clock. (This cell type is usually only used in netlists for formal verification.)

Simulation model (verilog)

Listing 9.190: simcells.v

```
589 module \$_FF_ (D, Q);
590     input D;
591     output reg Q;
592     always @($global_clock) begin
593         Q <= D;
594     end
595 endmodule
```

yosys> help $_SDFFCE_NNON_

A negative edge D-type flip-flop with negative polarity synchronous reset and negative polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 0 0 | 0
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.191: simcells.v

```
2908 module \$_SDFFCE_NNON_ (D, C, R, E, Q);
2909     input D, C, R, E;
2910     output reg Q;
2911     always @(negedge C) begin
2912         if (E == 0) begin
2913             if (R == 0)
2914                 Q <= 0;
2915             else
2916                 Q <= D;
2917         end
2918     end
2919 endmodule
```

yosys> help $_SDFFCE_NNOP_

A negative edge D-type flip-flop with negative polarity synchronous reset and positive polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 0 1 | 0
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.192: simcells.v

```
2935 module \$_SDFFCE_NN0P_ (D, C, R, E, Q);
2936     input D, C, R, E;
2937     output reg Q;
2938     always @(negedge C) begin
2939         if (E == 1) begin
2940             if (R == 0)
2941                 Q <= 0;
2942             else
2943                 Q <= D;
2944         end
2945     end
2946 endmodule
```

yosys> help $_SDFFCE_NN1N_

A negative edge D-type flip-flop with negative polarity synchronous set and negative polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 0 0 | 1
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.193: simcells.v

```
2962 module \$_SDFFCE_NN1N_ (D, C, R, E, Q);
2963     input D, C, R, E;
2964     output reg Q;
2965     always @(negedge C) begin
2966         if (E == 0) begin
2967             if (R == 0)
2968                 Q <= 1;
2969             else
2970                 Q <= D;
2971         end
2972     end
2973 endmodule
```

yosys> help $_SDFFCE_NN1P_

A negative edge D-type flip-flop with negative polarity synchronous set and positive polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 0 1 | 1
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.194: simcells.v

```
2989 module \$_SDFFCE_NN1P_ (D, C, R, E, Q);
2990     input D, C, R, E;
2991     output reg Q;
2992     always @(negedge C) begin
2993         if (E == 1) begin
2994             if (R == 0)
2995                 Q <= 1;
2996             else
2997                 Q <= D;
2998         end
2999     end
3000 endmodule
```

yosys> help $_SDFFCE_NPON_

A negative edge D-type flip-flop with positive polarity synchronous reset and negative polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 1 0 | 0
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.195: simcells.v

```
3016 module \$_SDFFCE_NPON_ (D, C, R, E, Q);
3017     input D, C, R, E;
3018     output reg Q;
3019     always @(negedge C) begin
3020         if (E == 0) begin
3021             if (R == 1)
3022                 Q <= 0;
3023             else
3024                 Q <= D;
3025         end
3026     end
3027 endmodule
```

yosys> help $_SDFFCE_NP0P_

A negative edge D-type flip-flop with positive polarity synchronous reset and positive polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 1 1 | 0
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.196: simcells.v

```
3043 module \$_SDFFCE_NP0P_ (D, C, R, E, Q);
3044     input D, C, R, E;
3045     output reg Q;
3046     always @(negedge C) begin
3047         if (E == 1) begin
3048             if (R == 1)
3049                 Q <= 0;
3050             else
3051                 Q <= D;
3052         end
3053     end
3054 endmodule
```

yosys> help $_SDFFCE_NP1N_

A negative edge D-type flip-flop with positive polarity synchronous set and negative polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 1 0 | 1
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.197: simcells.v

```
3070 module \$_SDFFCE_NP1N_ (D, C, R, E, Q);
3071     input D, C, R, E;
3072     output reg Q;
3073     always @(negedge C) begin
3074         if (E == 0) begin
3075             if (R == 1)
3076                 Q <= 1;
3077             else
3078                 Q <= D;
3079         end
3080     end
3081 endmodule
```

yosys> help $_SDFFCE_NP1P_

A negative edge D-type flip-flop with positive polarity synchronous set and positive polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 1 1 | 1
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.198: simcells.v

```
3097 module \$_SDFFCE_NP1P_ (D, C, R, E, Q);
3098     input D, C, R, E;
3099     output reg Q;
3100     always @(negedge C) begin
3101         if (E == 1) begin
3102             if (R == 1)
3103                 Q <= 1;
3104             else
3105                 Q <= D;
3106         end
3107     end
3108 endmodule
```

yosys> help $_SDFFCE_PNON_

A positive edge D-type flip-flop with negative polarity synchronous reset and negative polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 0 0 | 0
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.199: simcells.v

```
3124 module \$_SDFFCE_PNON_ (D, C, R, E, Q);
3125     input D, C, R, E;
3126     output reg Q;
3127     always @(posedge C) begin
3128         if (E == 0) begin
3129             if (R == 0)
3130                 Q <= 0;
3131             else
3132                 Q <= D;
3133         end
3134     end
3135 endmodule
```

yosys> help $_SDFFCE_PN0P_

A positive edge D-type flip-flop with negative polarity synchronous reset and positive polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 0 1 | 0
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.200: simcells.v

```
3151 module \$_SDFFCE_PN0P_ (D, C, R, E, Q);
3152     input D, C, R, E;
3153     output reg Q;
3154     always @(posedge C) begin
3155         if (E == 1) begin
3156             if (R == 0)
3157                 Q <= 0;
3158             else
3159                 Q <= D;
3160         end
3161     end
3162 endmodule
```

yosys> help $_SDFFCE_PN1N_

A positive edge D-type flip-flop with negative polarity synchronous set and negative polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 0 0 | 1
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.201: simcells.v

```
3178 module \$_SDFFCE_PN1N_ (D, C, R, E, Q);
3179     input D, C, R, E;
3180     output reg Q;
3181     always @(posedge C) begin
3182         if (E == 0) begin
3183             if (R == 0)
3184                 Q <= 1;
3185             else
3186                 Q <= D;
3187         end
3188     end
3189 endmodule
```

yosys> help $_SDFFCE_PN1P_

A positive edge D-type flip-flop with negative polarity synchronous set and positive polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
               ---------+---
                - / 0 1 | 1
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.202: simcells.v

```
3205 module \$_SDFFCE_PN1P_ (D, C, R, E, Q);
3206     input D, C, R, E;
3207     output reg Q;
3208     always @(posedge C) begin
3209         if (E == 1) begin
3210             if (R == 0)
3211                 Q <= 1;
3212             else
3213                 Q <= D;
3214         end
3215     end
3216 endmodule
```

yosys> help $_SDFFCE_PP0N_

A positive edge D-type flip-flop with positive polarity synchronous reset and negative polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
               ---------+---
                - / 1 0 | 0
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.203: simcells.v

```
3232 module \$_SDFFCE_PP0N_ (D, C, R, E, Q);
3233     input D, C, R, E;
3234     output reg Q;
3235     always @(posedge C) begin
3236         if (E == 0) begin
3237             if (R == 1)
3238                 Q <= 0;
3239             else
3240                 Q <= D;
3241         end
3242     end
3243 endmodule
```

yosys> help $_SDFFCE_PP0P_

A positive edge D-type flip-flop with positive polarity synchronous reset and positive polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 1 1 | 0
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.204: simcells.v

```
3259 module \$_SDFFCE_PP0P_ (D, C, R, E, Q);
3260     input D, C, R, E;
3261     output reg Q;
3262     always @(posedge C) begin
3263         if (E == 1) begin
3264             if (R == 1)
3265                 Q <= 0;
3266             else
3267                 Q <= D;
3268         end
3269     end
3270 endmodule
```

yosys> help $_SDFFCE_PP1N_

A positive edge D-type flip-flop with positive polarity synchronous set and negative polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 1 0 | 1
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.205: simcells.v

```
3286 module \$_SDFFCE_PP1N_ (D, C, R, E, Q);
3287     input D, C, R, E;
3288     output reg Q;
3289     always @(posedge C) begin
3290         if (E == 0) begin
3291             if (R == 1)
3292                 Q <= 1;
3293             else
3294                 Q <= D;
3295         end
3296     end
3297 endmodule
```

yosys> help $_SDFFCE_PP1P_

A positive edge D-type flip-flop with positive polarity synchronous set and positive polarity clock enable (with clock enable having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 1 1 | 1
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.206: simcells.v

```
3313 module \$_SDFFCE_PP1P_ (D, C, R, E, Q);
3314     input D, C, R, E;
3315     output reg Q;
3316     always @(posedge C) begin
3317         if (E == 1) begin
3318             if (R == 1)
3319                 Q <= 1;
3320             else
3321                 Q <= D;
3322         end
3323     end
3324 endmodule
```

yosys> help $_SDFFE_NNON_

A negative edge D-type flip-flop with negative polarity synchronous reset and negative polarity clock enable (with reset having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 0 - | 0
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.207: simcells.v

```
2508 module \$_SDFFE_NNON_ (D, C, R, E, Q);
2509     input D, C, R, E;
2510     output reg Q;
2511     always @(negedge C) begin
2512         if (R == 0)
2513             Q <= 0;
2514         else if (E == 0)
2515             Q <= D;
2516     end
2517 endmodule
```

yosys> help $_SDFFE_NNOP_

A negative edge D-type flip-flop with negative polarity synchronous reset and positive polarity clock

enable (with reset having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 0 - | 0
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.208: simcells.v

```
2533 module \$_SDFFE_NN0P_ (D, C, R, E, Q);
2534     input D, C, R, E;
2535     output reg Q;
2536     always @(negedge C) begin
2537         if (R == 0)
2538             Q <= 0;
2539         else if (E == 1)
2540             Q <= D;
2541     end
2542 endmodule
```

yosys> help $_SDFFE_NN1N_

A negative edge D-type flip-flop with negative polarity synchronous set and negative polarity clock enable (with set having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 0 - | 1
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.209: simcells.v

```
2558 module \$_SDFFE_NN1N_ (D, C, R, E, Q);
2559     input D, C, R, E;
2560     output reg Q;
2561     always @(negedge C) begin
2562         if (R == 0)
2563             Q <= 1;
2564         else if (E == 0)
2565             Q <= D;
2566     end
2567 endmodule
```

yosys> help $_SDFFE_NN1P_

A negative edge D-type flip-flop with negative polarity synchronous set and positive polarity clock enable (with set having priority).

```
Truth table:    D C R E | Q
                ---------+---
```

(continues on next page)

(continued from previous page)

```
                - \ 0 - | 1
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.210: simcells.v

```
module \$_SDFFE_NN1P_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(negedge C) begin
        if (R == 0)
            Q <= 1;
        else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_SDFFE_NP0N_

A negative edge D-type flip-flop with positive polarity synchronous reset and negative polarity clock enable (with reset having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 1 - | 0
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.211: simcells.v

```
module \$_SDFFE_NP0N_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(negedge C) begin
        if (R == 1)
            Q <= 0;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_SDFFE_NP0P_

A negative edge D-type flip-flop with positive polarity synchronous reset and positive polarity clock enable (with reset having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 1 - | 0
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.212: simcells.v

```
module \$_SDFFE_NP0P_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(negedge C) begin
        if (R == 1)
            Q <= 0;
        else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_SDFFE_NP1N_

A negative edge D-type flip-flop with positive polarity synchronous set and negative polarity clock enable (with set having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 1 - | 1
                d \ - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.213: simcells.v

```
module \$_SDFFE_NP1N_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(negedge C) begin
        if (R == 1)
            Q <= 1;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_SDFFE_NP1P_

A negative edge D-type flip-flop with positive polarity synchronous set and positive polarity clock enable (with set having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - \ 1 - | 1
                d \ - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.214: simcells.v

```
2683 module \$_SDFFE_NP1P_ (D, C, R, E, Q);
2684     input D, C, R, E;
2685     output reg Q;
2686     always @(negedge C) begin
2687         if (R == 1)
2688             Q <= 1;
2689         else if (E == 1)
2690             Q <= D;
2691     end
2692 endmodule
```

yosys> help $_SDFFE_PN0N_

A positive edge D-type flip-flop with negative polarity synchronous reset and negative polarity clock enable (with reset having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 0 - | 0
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.215: simcells.v

```
2708 module \$_SDFFE_PN0N_ (D, C, R, E, Q);
2709     input D, C, R, E;
2710     output reg Q;
2711     always @(posedge C) begin
2712         if (R == 0)
2713             Q <= 0;
2714         else if (E == 0)
2715             Q <= D;
2716     end
2717 endmodule
```

yosys> help $_SDFFE_PN0P_

A positive edge D-type flip-flop with negative polarity synchronous reset and positive polarity clock enable (with reset having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 0 - | 0
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.216: simcells.v

```
module \$_SDFFE_PN0P_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(posedge C) begin
        if (R == 0)
            Q <= 0;
        else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_SDFFE_PN1N_

A positive edge D-type flip-flop with negative polarity synchronous set and negative polarity clock enable (with set having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 0 - | 1
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.217: simcells.v

```
module \$_SDFFE_PN1N_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(posedge C) begin
        if (R == 0)
            Q <= 1;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_SDFFE_PN1P_

A positive edge D-type flip-flop with negative polarity synchronous set and positive polarity clock enable (with set having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 0 - | 1
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.218: simcells.v

```
2783 module \$_SDFFE_PN1P_ (D, C, R, E, Q);
2784     input D, C, R, E;
2785     output reg Q;
2786     always @(posedge C) begin
2787         if (R == 0)
2788             Q <= 1;
2789         else if (E == 1)
2790             Q <= D;
2791     end
2792 endmodule
```

yosys> help $_SDFFE_PP0N_

A positive edge D-type flip-flop with positive polarity synchronous reset and negative polarity clock enable (with reset having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 1 - | 0
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.219: simcells.v

```
2808 module \$_SDFFE_PP0N_ (D, C, R, E, Q);
2809     input D, C, R, E;
2810     output reg Q;
2811     always @(posedge C) begin
2812         if (R == 1)
2813             Q <= 0;
2814         else if (E == 0)
2815             Q <= D;
2816     end
2817 endmodule
```

yosys> help $_SDFFE_PP0P_

A positive edge D-type flip-flop with positive polarity synchronous reset and positive polarity clock enable (with reset having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 1 - | 0
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.220: simcells.v

```
module \$_SDFFE_PP0P_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(posedge C) begin
        if (R == 1)
            Q <= 0;
        else if (E == 1)
            Q <= D;
    end
endmodule
```

yosys> help $_SDFFE_PP1N_

A positive edge D-type flip-flop with positive polarity synchronous set and negative polarity clock enable (with set having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 1 - | 1
                d / - 0 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.221: simcells.v

```
module \$_SDFFE_PP1N_ (D, C, R, E, Q);
    input D, C, R, E;
    output reg Q;
    always @(posedge C) begin
        if (R == 1)
            Q <= 1;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help $_SDFFE_PP1P_

A positive edge D-type flip-flop with positive polarity synchronous set and positive polarity clock enable (with set having priority).

```
Truth table:    D C R E | Q
                ---------+---
                - / 1 - | 1
                d / - 1 | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.222: simcells.v

```
2883 module \$_SDFFE_PP1P_ (D, C, R, E, Q);
2884     input D, C, R, E;
2885     output reg Q;
2886     always @(posedge C) begin
2887         if (R == 1)
2888             Q <= 1;
2889         else if (E == 1)
2890             Q <= D;
2891     end
2892 endmodule
```

yosys> help $_SDFF_NN0_

A negative edge D-type flip-flop with negative polarity synchronous reset.

```
Truth table:    D C R | Q
               -------+---
                - \ 0 | 0
                d \ - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.223: simcells.v

```
2315 module \$_SDFF_NN0_ (D, C, R, Q);
2316     input D, C, R;
2317     output reg Q;
2318     always @(negedge C) begin
2319         if (R == 0)
2320             Q <= 0;
2321         else
2322             Q <= D;
2323     end
2324 endmodule
```

yosys> help $_SDFF_NN1_

A negative edge D-type flip-flop with negative polarity synchronous set.

```
Truth table:    D C R | Q
               -------+---
                - \ 0 | 1
                d \ - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.224: simcells.v

```
2339 module \$_SDFF_NN1_ (D, C, R, Q);
2340     input D, C, R;
2341     output reg Q;
```

(continues on next page)

(continued from previous page)

```
    always @(negedge C) begin
        if (R == 0)
            Q <= 1;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_SDFF_NP0_

A negative edge D-type flip-flop with positive polarity synchronous reset.

```
Truth table:    D C R | Q
               -------+---
                - \ 1 | 0
                d \ - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.225: simcells.v

```
module \$_SDFF_NP0_ (D, C, R, Q);
    input D, C, R;
    output reg Q;
    always @(negedge C) begin
        if (R == 1)
            Q <= 0;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_SDFF_NP1_

A negative edge D-type flip-flop with positive polarity synchronous set.

```
Truth table:    D C R | Q
               -------+---
                - \ 1 | 1
                d \ - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.226: simcells.v

```
module \$_SDFF_NP1_ (D, C, R, Q);
    input D, C, R;
    output reg Q;
    always @(negedge C) begin
        if (R == 1)
            Q <= 1;
        else
```

(continues on next page)

(continued from previous page)

```
            Q <= D;
    end
endmodule
```

yosys> help $_SDFF_PN0_

A positive edge D-type flip-flop with negative polarity synchronous reset.

```
Truth table:    D C R | Q
               -------+---
                - / 0 | 0
                d / - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.227: simcells.v

```
module \$_SDFF_PN0_ (D, C, R, Q);
    input D, C, R;
    output reg Q;
    always @(posedge C) begin
        if (R == 0)
            Q <= 0;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_SDFF_PN1_

A positive edge D-type flip-flop with negative polarity synchronous set.

```
Truth table:    D C R | Q
               -------+---
                - / 0 | 1
                d / - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.228: simcells.v

```
module \$_SDFF_PN1_ (D, C, R, Q);
    input D, C, R;
    output reg Q;
    always @(posedge C) begin
        if (R == 0)
            Q <= 1;
        else
            Q <= D;
    end
endmodule
```

yosys> help $_SDFF_PP0_

A positive edge D-type flip-flop with positive polarity synchronous reset.

```
Truth table:    D C R | Q
               -------+---
                - / 1 | 0
                d / - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.229: simcells.v

```
2459 module \$_SDFF_PP0_ (D, C, R, Q);
2460     input D, C, R;
2461     output reg Q;
2462     always @(posedge C) begin
2463         if (R == 1)
2464             Q <= 0;
2465         else
2466             Q <= D;
2467     end
2468 endmodule
```

yosys> help $_SDFF_PP1_

A positive edge D-type flip-flop with positive polarity synchronous set.

```
Truth table:    D C R | Q
               -------+---
                - / 1 | 1
                d / - | d
                - - - | q
```

Simulation model (verilog)

Listing 9.230: simcells.v

```
2483 module \$_SDFF_PP1_ (D, C, R, Q);
2484     input D, C, R;
2485     output reg Q;
2486     always @(posedge C) begin
2487         if (R == 1)
2488             Q <= 1;
2489         else
2490             Q <= D;
2491     end
2492 endmodule
```

9.2.4 Latch cells

The cell types *$_DLATCH_N_* and *$_DLATCH_P_* represent d-type latches.

Table 9.12: Cell types for basic latches

Verilog	Cell Type
`always @* if (!E) Q <= D`	*$_DLATCH_N_*
`always @* if (E) Q <= D`	*$_DLATCH_P_*

The cell types `$_DLATCH_[NP][NP][01]_` implement d-type latches with reset. The values in the table for these cell types relate to the following Verilog code template:

```
always @*
   if (R == RST_LVL)
      Q <= RST_VAL;
   else if (E == EN_LVL)
      Q <= D;
```

Table 9.13: Cell types for gate level logic networks (latches with reset)

EnLvl	*RstLvl*	*RstVal*	Cell Type
0	0	0	*$_DLATCH_NN0_*
0	0	1	*$_DLATCH_NN1_*
0	1	0	*$_DLATCH_NP0_*
0	1	1	*$_DLATCH_NP1_*
1	0	0	*$_DLATCH_PN0_*
1	0	1	*$_DLATCH_PN1_*
1	1	0	*$_DLATCH_PP0_*
1	1	1	*$_DLATCH_PP1_*

The cell types `$_DLATCHSR_[NP][NP][NP]_` implement d-type latches with set and reset. The values in the table for these cell types relate to the following Verilog code template:

```
always @*
   if (R == RST_LVL)
      Q <= 0;
   else if (S == SET_LVL)
      Q <= 1;
   else if (E == EN_LVL)
      Q <= D;
```

Table 9.14: Cell types for gate level logic networks (latches with set and reset)

EnLvl	*SetLvl*	*RstLvl*	Cell Type
0	0	0	*$_DLATCHSR_NNN_*
0	0	1	*$_DLATCHSR_NNP_*
0	1	0	*$_DLATCHSR_NPN_*
0	1	1	*$_DLATCHSR_NPP_*
1	0	0	*$_DLATCHSR_PNN_*
1	0	1	*$_DLATCHSR_PNP_*
1	1	0	*$_DLATCHSR_PPN_*
1	1	1	*$_DLATCHSR_PPP_*

The cell types `$_SR_[NP][NP]_` implement sr-type latches. The values in the table for these cell types relate to the following Verilog code template:

```
always @*
   if (R == RST_LVL)
      Q <= 0;
   else if (S == SET_LVL)
      Q <= 1;
```

Table 9.15: Cell types for gate level logic networks (SR latches)

SetLvl	*RstLvl*	Cell Type
0	0	*$_SR_NN_*
0	1	*$_SR_NP_*
1	0	*$_SR_PN_*
1	1	*$_SR_PP_*

yosys> help $_DLATCHSR_NNN_

A negative enable D-type latch with negative polarity set and negative polarity reset.

```
Truth table:    E S R D | Q
               ---------+---
                - - 0 - | 0
                - 0 - - | 1
                0 - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.231: simcells.v

```
3575 module \$_DLATCHSR_NNN_ (E, S, R, D, Q);
3576     input E, S, R, D;
3577     output reg Q;
3578     always @* begin
3579         if (R == 0)
3580             Q <= 0;
3581         else if (S == 0)
3582             Q <= 1;
3583         else if (E == 0)
3584             Q <= D;
3585     end
3586 endmodule
```

yosys> help $_DLATCHSR_NNP_

A negative enable D-type latch with negative polarity set and positive polarity reset.

```
Truth table:    E S R D | Q
               ---------+---
                - - 1 - | 0
                - 0 - - | 1
```

(continues on next page)

(continued from previous page)

```
                0 - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.232: simcells.v

```
3603 module \$_DLATCHSR_NNP_ (E, S, R, D, Q);
3604     input E, S, R, D;
3605     output reg Q;
3606     always @* begin
3607         if (R == 1)
3608             Q <= 0;
3609         else if (S == 0)
3610             Q <= 1;
3611         else if (E == 0)
3612             Q <= D;
3613     end
3614 endmodule
```

yosys> help $_DLATCHSR_NPN_

A negative enable D-type latch with positive polarity set and negative polarity reset.

```
Truth table:    E S R D | Q
                ---------+---
                - - 0 - | 0
                - 1 - - | 1
                0 - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.233: simcells.v

```
3631 module \$_DLATCHSR_NPN_ (E, S, R, D, Q);
3632     input E, S, R, D;
3633     output reg Q;
3634     always @* begin
3635         if (R == 0)
3636             Q <= 0;
3637         else if (S == 1)
3638             Q <= 1;
3639         else if (E == 0)
3640             Q <= D;
3641     end
3642 endmodule
```

yosys> help $_DLATCHSR_NPP_

A negative enable D-type latch with positive polarity set and positive polarity reset.

```
Truth table:    E S R D | Q
                ---------+---
```

(continues on next page)

(continued from previous page)

```
                - - 1 - | 0
                - 1 - - | 1
                0 - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.234: simcells.v

```
3659 module \$_DLATCHSR_NPP_ (E, S, R, D, Q);
3660     input E, S, R, D;
3661     output reg Q;
3662     always @* begin
3663         if (R == 1)
3664             Q <= 0;
3665         else if (S == 1)
3666             Q <= 1;
3667         else if (E == 0)
3668             Q <= D;
3669     end
3670 endmodule
```

yosys> help $_DLATCHSR_PNN_

A positive enable D-type latch with negative polarity set and negative polarity reset.

```
Truth table:    E S R D | Q
                ---------+---
                - - 0 - | 0
                - 0 - - | 1
                1 - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.235: simcells.v

```
3687 module \$_DLATCHSR_PNN_ (E, S, R, D, Q);
3688     input E, S, R, D;
3689     output reg Q;
3690     always @* begin
3691         if (R == 0)
3692             Q <= 0;
3693         else if (S == 0)
3694             Q <= 1;
3695         else if (E == 1)
3696             Q <= D;
3697     end
3698 endmodule
```

yosys> help $_DLATCHSR_PNP_

A positive enable D-type latch with negative polarity set and positive polarity reset.

```
Truth table:    E S R D | Q
                ---------+---
                - - 1 - | 0
                - 0 - - | 1
                1 - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.236: simcells.v

```
3715 module \$_DLATCHSR_PNP_ (E, S, R, D, Q);
3716     input E, S, R, D;
3717     output reg Q;
3718     always @* begin
3719         if (R == 1)
3720             Q <= 0;
3721         else if (S == 0)
3722             Q <= 1;
3723         else if (E == 1)
3724             Q <= D;
3725     end
3726 endmodule
```

yosys> help $_DLATCHSR_PPN_

A positive enable D-type latch with positive polarity set and negative polarity reset.

```
Truth table:    E S R D | Q
                ---------+---
                - - 0 - | 0
                - 1 - - | 1
                1 - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.237: simcells.v

```
3743 module \$_DLATCHSR_PPN_ (E, S, R, D, Q);
3744     input E, S, R, D;
3745     output reg Q;
3746     always @* begin
3747         if (R == 0)
3748             Q <= 0;
3749         else if (S == 1)
3750             Q <= 1;
3751         else if (E == 1)
3752             Q <= D;
3753     end
3754 endmodule
```

yosys> help $_DLATCHSR_PPP_

A positive enable D-type latch with positive polarity set and positive polarity reset.

```
Truth table:    E S R D | Q
               ---------+---
                - - 1 - | 0
                - 1 - - | 1
                1 - - d | d
                - - - - | q
```

Simulation model (verilog)

Listing 9.238: simcells.v

```
3771 module \$_DLATCHSR_PPP_ (E, S, R, D, Q);
3772     input E, S, R, D;
3773     output reg Q;
3774     always @* begin
3775         if (R == 1)
3776             Q <= 0;
3777         else if (S == 1)
3778             Q <= 1;
3779         else if (E == 1)
3780             Q <= D;
3781     end
3782 endmodule
```

yosys> help $_DLATCH_NNO_

A negative enable D-type latch with negative polarity reset.

```
Truth table:    E R D | Q
               -------+---
                - 0 - | 0
                0 - d | d
                - - - | q
```

Simulation model (verilog)

Listing 9.239: simcells.v

```
3381 module \$_DLATCH_NNO_ (E, R, D, Q);
3382     input E, R, D;
3383     output reg Q;
3384     always @* begin
3385         if (R == 0)
3386                     Q <= 0;
3387         else if (E == 0)
3388             Q <= D;
3389     end
3390 endmodule
```

yosys> help $_DLATCH_NN1_

A negative enable D-type latch with negative polarity set.

```
Truth table:    E R D | Q
               -------+---
```

(continues on next page)

(continued from previous page)

```
                - 0 - | 1
                0 - d | d
                - - - | q
```

Simulation model (verilog)

Listing 9.240: simcells.v

```
module \$_DLATCH_NN1_ (E, R, D, Q);
    input E, R, D;
    output reg Q;
    always @* begin
        if (R == 0)
                    Q <= 1;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help **$_DLATCH_NP0_**

A negative enable D-type latch with positive polarity reset.

```
Truth table:    E R D | Q
               -------+---
                - 1 - | 0
                0 - d | d
                - - - | q
```

Simulation model (verilog)

Listing 9.241: simcells.v

```
module \$_DLATCH_NP0_ (E, R, D, Q);
    input E, R, D;
    output reg Q;
    always @* begin
        if (R == 1)
                    Q <= 0;
        else if (E == 0)
            Q <= D;
    end
endmodule
```

yosys> help **$_DLATCH_NP1_**

A negative enable D-type latch with positive polarity set.

```
Truth table:    E R D | Q
               -------+---
                - 1 - | 1
                0 - d | d
                - - - | q
```

Simulation model (verilog)

Listing 9.242: simcells.v

```
3453 module \$_DLATCH_NP1_ (E, R, D, Q);
3454     input E, R, D;
3455     output reg Q;
3456     always @* begin
3457         if (R == 1)
3458                     Q <= 1;
3459         else if (E == 0)
3460             Q <= D;
3461     end
3462 endmodule
```

yosys> help $_DLATCH_N_

A negative enable D-type latch.

```
Truth table:    E D | Q
               -----+---
                0 d | d
                - - | q
```

Simulation model (verilog)

Listing 9.243: simcells.v

```
3338 module \$_DLATCH_N_ (E, D, Q);
3339     input E, D;
3340     output reg Q;
3341     always @* begin
3342         if (E == 0)
3343             Q <= D;
3344     end
3345 endmodule
```

yosys> help $_DLATCH_PN0_

A positive enable D-type latch with negative polarity reset.

```
Truth table:    E R D | Q
               -------+---
                - 0 - | 0
                1 - d | d
                - - - | q
```

Simulation model (verilog)

Listing 9.244: simcells.v

```
3477 module \$_DLATCH_PN0_ (E, R, D, Q);
3478     input E, R, D;
3479     output reg Q;
3480     always @* begin
3481         if (R == 0)
3482                     Q <= 0;
```

(continues on next page)

(continued from previous page)

```
3483        else if (E == 1)
3484            Q <= D;
3485    end
3486 endmodule
```

yosys> help $_DLATCH_PN1_

A positive enable D-type latch with negative polarity set.

```
Truth table:    E R D | Q
               -------+---
                - 0 - | 1
                1 - d | d
                - - - | q
```

Simulation model (verilog)

Listing 9.245: simcells.v

```
3501 module \$_DLATCH_PN1_ (E, R, D, Q);
3502     input E, R, D;
3503     output reg Q;
3504     always @* begin
3505         if (R == 0)
3506                     Q <= 1;
3507         else if (E == 1)
3508             Q <= D;
3509     end
3510 endmodule
```

yosys> help $_DLATCH_PP0_

A positive enable D-type latch with positive polarity reset.

```
Truth table:    E R D | Q
               -------+---
                - 1 - | 0
                1 - d | d
                - - - | q
```

Simulation model (verilog)

Listing 9.246: simcells.v

```
3525 module \$_DLATCH_PP0_ (E, R, D, Q);
3526     input E, R, D;
3527     output reg Q;
3528     always @* begin
3529         if (R == 1)
3530                     Q <= 0;
3531         else if (E == 1)
3532             Q <= D;
3533     end
3534 endmodule
```

yosys> help $_DLATCH_PP1_

A positive enable D-type latch with positive polarity set.

```
Truth table:    E R D | Q
               -------+---
                - 1 - | 1
                1 - d | d
                - - - | q
```

Simulation model (verilog)

Listing 9.247: simcells.v

```
3549 module \$_DLATCH_PP1_ (E, R, D, Q);
3550     input E, R, D;
3551     output reg Q;
3552     always @* begin
3553         if (R == 1)
3554                     Q <= 1;
3555         else if (E == 1)
3556             Q <= D;
3557     end
3558 endmodule
```

yosys> help $_DLATCH_P_

A positive enable D-type latch.

```
Truth table:    E D | Q
               -----+---
                1 d | d
                - - | q
```

Simulation model (verilog)

Listing 9.248: simcells.v

```
3359 module \$_DLATCH_P_ (E, D, Q);
3360     input E, D;
3361     output reg Q;
3362     always @* begin
3363         if (E == 1)
3364             Q <= D;
3365     end
3366 endmodule
```

yosys> help $_SR_NN_

A set-reset latch with negative polarity SET and negative polarity RESET.

```
Truth table:    S R | Q
               -----+---
                - 0 | 0
                0 - | 1
                - - | q
```

Simulation model (verilog)

Listing 9.249: simcells.v

```
module \$_SR_NN_ (S, R, Q);
    input S, R;
    output reg Q;
    always @* begin
        if (R == 0)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
    end
endmodule
```

yosys> help $_SR_NP_

A set-reset latch with negative polarity SET and positive polarity RESET.

```
Truth table:    S R | Q
               -----+---
                - 1 | 0
                0 - | 1
                - - | q
```

Simulation model (verilog)

Listing 9.250: simcells.v

```
module \$_SR_NP_ (S, R, Q);
    input S, R;
    output reg Q;
    always @* begin
        if (R == 1)
            Q <= 0;
        else if (S == 0)
            Q <= 1;
    end
endmodule
```

yosys> help $_SR_PN_

A set-reset latch with positive polarity SET and negative polarity RESET.

```
Truth table:    S R | Q
               -----+---
                - 0 | 0
                1 - | 1
                - - | q
```

Simulation model (verilog)

Listing 9.251: simcells.v

```
module \$_SR_PN_ (S, R, Q);
    input S, R;
```

(continues on next page)

(continued from previous page)

```
547     output reg Q;
548     always @* begin
549         if (R == 0)
550             Q <= 0;
551         else if (S == 1)
552             Q <= 1;
553     end
554 endmodule
```

yosys> help $_SR_PP_

A set-reset latch with positive polarity SET and positive polarity RESET.

```
Truth table:    S R | Q
               -----+---
                - 1 | 0
                1 - | 1
                - - | q
```

Simulation model (verilog)

Listing 9.252: simcells.v

```
569 module \$_SR_PP_ (S, R, Q);
570     input S, R;
571     output reg Q;
572     always @* begin
573         if (R == 1)
574             Q <= 0;
575         else if (S == 1)
576             Q <= 1;
577     end
578 endmodule
```

9.2.5 Other gate-level cells

Other gate-level cells

yosys> help $_TBUF_

A tri-state buffer.

```
Truth table:    A E | Y
               -----+---
                a 1 | a
                - 0 | z
```

Simulation model (verilog)

Listing 9.253: simcells.v

```
473 module \$_TBUF_ (A, E, Y);
474     input A, E;
475     output Y;
```

(continues on next page)

(continued from previous page)

```
476     assign Y = E ? A : 1'bz;
477 endmodule
```

9.3 Cell properties

is_evaluable

These cells are able to be used in conjunction with the `eval` command. Some passes, such as *opt_expr*, may also be able to perform additional optimizations on cells which are evaluable.

x-aware

Some passes will treat these cells as the non 'x' aware cell. For example, during synthesis *$eqx* will typically be treated as *$eq*.

x-output

These cells can produce 'x' output even if all inputs are defined. For example, a *$div* cell with divisor (`B`) equal to zero has undefined output.

Refer to the propindex for the list of cells with a given property.

CHAPTER

TEN

COMMAND LINE REFERENCE

```
Usage:
  ./yosys [OPTION...] [<infile> [..]]

 operation options:
  -b, --backend <backend>       use <backend> for the output file specified on the␣
↪command line
  -f, --frontend <frontend>     use <frontend> for the input files on the command line
  -s, --scriptfile <scriptfile>
                                execute the commands in <scriptfile>
  -c, --tcl-scriptfile <tcl_scriptfile>
                                execute the commands in the TCL <tcl_scriptfile> (see
↪'help tcl' for details)
  -C, --tcl-interactive         enters TCL interactive shell mode
  -p, --commands <commands>     execute <commands> (to chain commands, separate them␣
↪with semicolon + whitespace: 'cmd1; cmd2')
  -r, --top <top>               elaborate the specified HDL <top> module
  -m, --plugin <plugin>         load the specified <plugin> module
  -D, --define <define>[=<value>]
                                set the specified Verilog define to <value> if supplied␣
↪via command "read -define"
  -S, --synth                   shortcut for calling the "synth" command, a default␣
↪script for transforming the Verilog input to a gate-level netlist. For example: yosys -
↪o output.blif -S input.v For more complex synthesis jobs it is recommended to use the␣
↪read_* and write_* commands in a script file instead of specifying input and output␣
↪files on the command line.
  -H                            print the command list
  -h, --help [<command>]        print this help message. If given, print help for
↪<command>.
  -V, --version                 print version information and exit
      --git-hash                print git commit hash and exit

 logging options:
  -Q                            suppress printing of banner (copyright, disclaimer,␣
↪version)
  -T                            suppress printing of footer (log hash, version, timing␣
↪statistics)
      --no-version              suppress writing out Yosys version anywhere excluding -V,
↪ --version
  -q, --quiet                   quiet operation. Only write warnings and error messages␣
↪to console. Use this option twice to also quiet warning messages
```

(continues on next page)

(continued from previous page)

```
  -v, --verbose <level>         print log headers up to <level> to the console. Implies -
↪q for everything except the 'End of script.' message.
  -t, --timestamp               annotate all log messages with a time stamp
  -d, --detailed-timing         print more detailed timing stats at exit
  -l, --logfile <logfile>       write log messages to <logfile>
  -L, --line-buffered-logfile <logfile>
                                like -l but open <logfile> in line buffered mode
  -o, --outfile <outfile>       write the design to <outfile> on exit
  -P, --dump-design <header_id>[:<filename>]
                                dump the design when printing the specified log header␣
↪to a file. yosys_dump_<header_id>.il is used as filename if none is specified. Use 'ALL
↪' as <header_id> to dump at every header.
  -W, --warning-as-warning <regex>
                                print a warning for all log messages matching <regex>
  -w, --warning-as-message <regex>
                                if a warning message matches <regex>, it is printed as␣
↪regular message instead
  -e, --warning-as-error <regex>
                                if a warning message matches <regex>, it is printed as␣
↪error message instead
  -E, --deps-file <depsfile>    write a Makefile dependencies file <depsfile> with input␣
↪and output file names

 developer options:
  -X, --trace                   enable tracing of core data structure changes. for␣
↪debugging
  -M, --randomize-pointers      will slightly randomize allocated pointer addresses. for␣
↪debugging
      --autoidx <idx>           start counting autoidx up from <seed>, similar effect to␣
↪--hash-seed
      --hash-seed <seed>        mix up hashing values with <seed>, for extreme␣
↪optimization and testing
  -A, --abort                   will call abort() at the end of the script. for debugging
  -x, --experimental <feature>  do not print warnings for the experimental <feature>
  -g, --debug                   globally enable debug log messages
      --perffile <perffile>     write a JSON performance log to <perffile>
```

10.1 Command reference

Todo

Can we warn on command groups that aren't included anywhere?

List of all commands

10.1.1 Yosys environment variables

HOME
: Yosys command history is stored in `$HOME/.yosys_history`. Graphics (from *show* and *viz* commands) will output to this directory by default. This environment variable is also used in some cases for resolving filenames with `~`.

PATH
: May be used in OpenBSD builds for finding the location of Yosys executable.

TMPDIR
: Used for storing temporary files.

ABC
: When compiling Yosys with out-of-tree ABC using **ABCEXTERNAL**, this variable can be used to override the external ABC executable.

YOSYS_NOVERIFIC
: If Yosys was built with Verific, this environment variable can be used to temporarily disable Verific support.

YOSYS_COVER_DIR and YOSYS_COVER_FILE
: When using code coverage, these environment variables control the output file name/location.

YOSYS_ABORT_ON_LOG_ERROR
: Can be used for debugging Yosys internals. Setting it to 1 causes abort() to be called when Yosys terminates with an error message.

10.1.2 Reading input files

write_aiger - write design to AIGER file

```
yosys> help write_aiger
```

write_aiger [options] [filename]

```
Write the current design to an AIGER file. The design must be flattened and
must not contain any cell types except $_AND_, $_NOT_, simple FF types,
$assert and $assume cells, and $initstate cells.

$assert and $assume cells are converted to AIGER bad state properties and
invariant constraints.
```

-ascii	write ASCII version of AIGER format
-zinit	convert FFs to zero-initialized FFs, adding additional inputs for uninitialized FFs.

-miter	design outputs are AIGER bad state properties
-symbols	include a symbol table in the generated AIGER file
-no-sort	don't sort input/output ports
-map <filename>	write an extra file with port and latch symbols
-vmap <filename>	like -map, but more verbose
-no-startoffset	make indexes zero based, enable using map files with␣ ↪smt solvers.
-ywmap <filename>	write a map file for conversion to and from yosys␣ ↪witness traces, also allows for mapping AIGER bad-state properties and␣ ↪invariant constraints back to individual formal properties by␣ ↪name.
-I, -O, -B, -L	If the design contains no input/output/assert/flip-flop␣ ↪then create one dummy input/output/bad_state-pin or latch to make the␣ ↪tools reading the AIGER file happy.

write_aiger2 - (experimental) write design to AIGER file

```
yosys> help write_aiger2
```

> **Warning**
>
> This command is experimental

```
write_aiger2 [options] [filename]
```

```
Write the selected module to an AIGER file.
```

-strash	perform structural hashing while writing
-flatten	allow descending into submodules and write a flattened␣ ↪view of the design hierarchy starting at the selected top

```
This command is able to ingest all combinational cells except for:

    $neg, $slice, $lut, $sop, $shl, $shr, $sshl, $sshr, $shift, $shiftx,
```

(continues on next page)

(continued from previous page)

```
    $eqx, $nex, $add, $sub, $mul, $div, $mod, $divfloor, $modfloor, $pow,
    $concat, $macc, $bweqx, $demux, $lcu, $alu, $macc_v2,

And all combinational gates except for:

    $_MUX4_, $_MUX8_, $_MUX16_,
```

write_blif - write design to BLIF file

```
yosys> help write_blif
```

```
write_blif [options] [filename]
```

```
Write the current design to an BLIF file.
```

```
-top top_module
    set the specified module as design top module

-buf <cell-type> <in-port> <out-port>
    use cells of type <cell-type> with the specified port names for buffers

-unbuf <cell-type> <in-port> <out-port>
    insert buffer cells with the specified name and port names with
    a .names statement that models a buffer

-true <cell-type> <out-port>

-false <cell-type> <out-port>

-undef <cell-type> <out-port>
    use the specified cell types to drive nets that are constant 1, 0, or
    undefined. when '-' is used as <cell-type>, then <out-port> specifies
    the wire name to be used for the constant signal and no cell driving
    that wire is generated. when '+' is used as <cell-type>, then <out-port>
    specifies the wire name to be used for the constant signal and a .names
    statement is generated to drive the wire.

-noalias
    if a net name is aliasing another net name, then by default a net
    without fanout is created that is driven by the other net. This option
    suppresses the generation of this nets without fanout.
```

```
The following options can be useful when the generated file is not going to be
read by a BLIF parser but a custom tool. It is recommended not to name the
output file *.blif when any of these options are used.
```

-icells	do not translate Yosys's internal gates to generic BLIF logic functions. Instead create .subckt or .gate lines for all cells.
-gates	print .gate instead of .subckt lines for all cells that are not instantiations of other modules from this design.
-conn	do not generate buffers for connected wires. instead use the non-standard .conn statement.
-attr	use the non-standard .attr statement to write cell attributes
-param	use the non-standard .param statement to write cell parameters
-cname	use the non-standard .cname statement to write cell names
-iname, -iattr	enable -cname and -attr functionality for .names statements (the .cname and .attr statements will be included in the BLIF output after the truth table for the .names statement)
-blackbox	write blackbox cells with .blackbox statement.
-impltf	do not write definitions for the $true, $false and $undef wires.
-gatesi	write initial bit(s) with .gateinit for gates that needs to be initialized.

write_btor - write design to BTOR file

yosys> help **write_btor**

write_btor [options] [filename]

Write a BTOR description of the current design.

-v	Add comments and indentation to BTOR output file
-s	Output only a single bad property for all asserts

```
-c                  Output cover properties using 'bad' statements instead␣
                    ↪of asserts

-i <filename>       Create additional info file with auxiliary information

-x                  Output symbols for internal netnames (starting with '$')

-ywmap <filename>   Create a map file for conversion to and from Yosys␣
                    ↪witness traces
```

write_cxxrtl - convert design to C++ RTL simulation

yosys> help **write_cxxrtl**

write_cxxrtl [options] [filename]

```
Write C++ code that simulates the design. The generated code requires a driver
that instantiates the design, toggles its clock, and interacts with its ports.

The following driver may be used as an example for a design with a single clock
driving rising edge triggered flip-flops:

    #include "top.cc"

    int main() {
      cxxrtl_design::p_top top;
      top.step();
      while (1) {
        /* user logic */
        top.p_clk.set(false);
        top.step();
        top.p_clk.set(true);
        top.step();
      }
    }

Note that CXXRTL simulations, just like the hardware they are simulating, are
subject to race conditions. If, in the example above, the user logic would run
simultaneously with the rising edge of the clock, the design would malfunction.

This backend supports replacing parts of the design with black boxes implemented
in C++. If a module marked as a CXXRTL black box, its implementation is ignored,
and the generated code consists only of an interface and a factory function.
The driver must implement the factory function that creates an implementation of
the black box, taking into account the parameters it is instantiated with.

For example, the following Verilog code defines a CXXRTL black box interface for
a synchronous debug sink:

    (* cxxrtl_blackbox *)
    module debug(...);
      (* cxxrtl_edge = "p" *) input clk;
```

(continues on next page)

(continued from previous page)

```
      input en;
      input [7:0] i_data;
      (* cxxrtl_sync *) output [7:0] o_data;
    endmodule

For this HDL interface, this backend will generate the following C++ interface:

    struct bb_p_debug : public module {
      value<1> p_clk;
      bool posedge_p_clk() const { /* ... */ }
      value<1> p_en;
      value<8> p_i_data;
      wire<8> p_o_data;

      bool eval(performer *performer) override;
      virtual bool commit(observer &observer);
      bool commit() override;

      static std::unique_ptr<bb_p_debug>
      create(std::string name, metadata_map parameters, metadata_map attributes);
    };

The `create' function must be implemented by the driver. For example, it could
always provide an implementation logging the values to standard error stream:

    namespace cxxrtl_design {

    struct stderr_debug : public bb_p_debug {
      bool eval(performer *performer) override {
        if (posedge_p_clk() && p_en)
          fprintf(stderr, "debug: %02x\n", p_i_data.data[0]);
        p_o_data.next = p_i_data;
        return bb_p_debug::eval(performer);
      }
    };

    std::unique_ptr<bb_p_debug>
    bb_p_debug::create(std::string name, cxxrtl::metadata_map parameters,
                       cxxrtl::metadata_map attributes) {
      return std::make_unique<stderr_debug>();
    }

    }

For complex applications of black boxes, it is possible to parameterize their
port widths. For example, the following Verilog code defines a CXXRTL black box
interface for a configurable width debug sink:

    (* cxxrtl_blackbox, cxxrtl_template = "WIDTH" *)
    module debug(...);
      parameter WIDTH = 8;
      (* cxxrtl_edge = "p" *) input clk;
```

(continues on next page)

(continued from previous page)

```
      input en;
      (* cxxrtl_width = "WIDTH" *) input [WIDTH - 1:0] i_data;
      (* cxxrtl_width = "WIDTH" *) output [WIDTH - 1:0] o_data;
    endmodule

For this parametric HDL interface, this backend will generate the following C++
interface (only the differences are shown):

    template<size_t WIDTH>
    struct bb_p_debug : public module {
      // ...
      value<WIDTH> p_i_data;
      wire<WIDTH> p_o_data;
      // ...
      static std::unique_ptr<bb_p_debug<WIDTH>>
      create(std::string name, metadata_map parameters, metadata_map attributes);
    };

The `create' function must be implemented by the driver, specialized for every
possible combination of template parameters. (Specialization is necessary to
enable separate compilation of generated code and black box implementations.)

    template<size_t SIZE>
    struct stderr_debug : public bb_p_debug<SIZE> {
      // ...
    };

    template<>
    std::unique_ptr<bb_p_debug<8>>
    bb_p_debug<8>::create(std::string name, cxxrtl::metadata_map parameters,
                          cxxrtl::metadata_map attributes) {
      return std::make_unique<stderr_debug<8>>();
    }

The following attributes are recognized by this backend:

    cxxrtl_blackbox
        only valid on modules. if specified, the module contents are ignored,
        and the generated code includes only the module interface and a factory
        function, which will be called to instantiate the module.

    cxxrtl_edge
        only valid on inputs of black boxes. must be one of "p", "n", "a".
        if specified on signal `clk`, the generated code includes edge detectors
        `posedge_p_clk()` (if "p"), `negedge_p_clk()` (if "n"), or both (if
        "a"), simplifying implementation of clocked black boxes.

    cxxrtl_template
        only valid on black boxes. must contain a space separated sequence of
        identifiers that have a corresponding black box parameters. for each
        of them, the generated code includes a `size_t` template parameter.
```

(continues on next page)

(continued from previous page)

```
    cxxrtl_width
        only valid on ports of black boxes. must be a constant expression, which
        is directly inserted into generated code.

    cxxrtl_comb, cxxrtl_sync
        only valid on outputs of black boxes. if specified, indicates that every
        bit of the output port is driven, correspondingly, by combinatorial or
        synchronous logic. this knowledge is used for scheduling optimizations.
        if neither is specified, the output will be pessimistically treated as
        driven by both combinatorial and synchronous logic.

The following options are supported by this backend:
```

-print-wire-types enable debug logging, for pass developers.

-header

```
generate separate interface (.h) and implementation (.
↪cc) files.
if specified, the backend must be called with a␣
↪filename, and filename
of the interface is derived from filename of the␣
↪implementation.
otherwise, interface and implementation are generated␣
↪together.
```

-namespace <ns-name>

```
place the generated code into namespace <ns-name>. if␣
↪not specified,
"cxxrtl_design" is used.
```

-print-output <stream>

```
print cells in the generated code direct their output␣
↪to <stream>.
must be one of "std::cout", "std::cerr". if not␣
↪specified,
"std::cout" is used. explicitly provided performer␣
↪overrides this.
```

-nohierarchy

```
use design hierarchy as-is. in most designs, a top␣
↪module should be
present as it is exposed through the C API and has␣
↪unbuffered outputs
for improved performance; it will be determined␣
↪automatically if absent.
```

-noflatten

```
don't flatten the design. fully flattened designs can␣
↪evaluate within
one delta cycle if they have no combinatorial feedback.
note that the debug interface and waveform dumps use␣
↪full hierarchical
names for all wires even in flattened designs.
```

-noproc	don't convert processes to netlists. in most designs, converting processes significantly improves evaluation performance at the cost of slight increase in compilation time.
-O <level>	set the optimization level. the default is -O6. higher optimization levels dramatically decrease compile and run time, and highest level possible for a design should be used.
-O0	no optimization.
-O1	unbuffer internal wires if possible.
-O2	like -O1, and localize internal wires if possible.
-O3	like -O2, and inline internal wires if possible.
-O4	like -O3, and unbuffer public wires not marked (*keep*) if possible.
-O5	like -O4, and localize public wires not marked (*keep*) if possible.
-O6	like -O5, and inline public wires not marked (*keep*) if possible.
-g <level>	set the debug level. the default is -g4. higher debug levels provide more visibility and generate more code, but do not pessimize evaluation.
-g0	no debug information. the C API is disabled.
-g1	include bare minimum of debug information necessary to access all design state. the C API is enabled.
-g2	like -g1, but include debug information for all public wires that are directly accessible through the C++ interface.
-g3	like -g2, and include debug information for public wires that are tied to a constant or another public wire.

-g4	like -g3, and compute debug information on demand for all public wires that were optimized out.

write_edif - write design to EDIF netlist file

```
yosys> help write_edif
```

```
write_edif [options] [filename]
```

Write the current design to an EDIF netlist file.

-top top_module	set the specified module as design top module
-nogndvcc	do not create "GND" and "VCC" cells. (this will produce an error if the design contains constant nets. use "hilomap" to map to custom constant drivers first)
-gndvccy	create "GND" and "VCC" cells with "Y" outputs. (the default is "G" for "GND" and "P" for "VCC".)
-attrprop	create EDIF properties for cell attributes
-keep	create extra KEEP nets by allowing a cell to drive multiple nets.
-pvector {par|bra|ang}	sets the delimiting character for module port rename clauses to parentheses, square brackets, or angle brackets.
-lsbidx	use index 0 for the LSB bit of a net or port instead of MSB.

Unfortunately there are different "flavors" of the EDIF file format. This command generates EDIF files for the Xilinx place&route tools. It might be necessary to make small modifications to this command when a different tool is targeted.

write_firrtl - write design to a FIRRTL file

```
yosys> help write_firrtl
```

```
write_firrtl [options] [filename]
```

Write a FIRRTL netlist of the current design.
The following commands are executed by this command:

(continues on next page)

(continued from previous page)

```
        pmuxtree
        bmuxmap
        demuxmap
        bwmuxmap
```

write_functional_cxx - convert design to C++ using the functional backend

```
yosys> help write_functional_cxx
```

```
TODO: add help message
```

write_functional_rosette - Generate Rosette compatible Racket from Functional IR

```
yosys> help write_functional_rosette
```

```
write_functional_rosette [options] [filename]
```

```
Functional Rosette Backend.
```

```
    -provides           include 'provide' statement(s) for loading output as a␣
                        ↪module

    -assoc-list-helpers provide helper functions which convert inputs/outputs␣
                        ↪from/to association lists
```

write_functional_smt2 - Generate SMT-LIB from Functional IR

```
yosys> help write_functional_smt2
```

```
Functional SMT Backend.
```

write_intersynth - write design to InterSynth netlist file

```
yosys> help write_intersynth
```

```
write_intersynth [options] [filename]
```

```
Write the current design to an 'intersynth' netlist file. InterSynth is
a tool for Coarse-Grain Example-Driven Interconnect Synthesis.
```

```
    -notypes            do not generate celltypes and conntypes commands. i.e.␣
                        ↪just output
                        the netlists. this is used for postsilicon synthesis.

    -lib <verilog_or_liberty_file>
                        use the specified library file for determining whether␣
                        ↪cell ports are
                        inputs or outputs. This option can be used multiple␣
                        ↪times to specify
                        more than one library.
```

-selected	only write selected modules. modules must be selected entirely or not at all.

http://bygone.clairexen.net/intersynth/

write_jny - generate design metadata

yosys> help write_jny

```
jny [options] [selection]

Write JSON netlist metadata for the current design
```

-no-connections	Don't include connection information in the netlist output.
-no-attributes	Don't include attributed information in the netlist output.
-no-properties	Don't include property information in the netlist output.

```
The JSON schema for JNY output files is located in the "jny.schema.json" file
which is located at "https://raw.githubusercontent.com/YosysHQ/yosys/main/misc/jny.
↪schema.json"
```

write_json - write design to a JSON file

yosys> help write_json

write_json [options] [filename]

```
Write a JSON netlist of the current design.
```

-aig	include AIG models for the different gate types
-compat-int	emit 32-bit or smaller fully-defined parameter values directly as JSON numbers (for compatibility with old parsers)
-selected	output only select module
-noscopeinfo	don't include $scopeinfo cells in the output

```
The general syntax of the JSON output created by this command is as follows:

    {
      "creator": "Yosys <version info>",
      "modules": {
        <module_name>: {
          "attributes": {
            <attribute_name>: <attribute_value>,
            ...
          },
          "parameter_default_values": {
            <parameter_name>: <parameter_value>,
            ...
          },
          "ports": {
            <port_name>: <port_details>,
            ...
          },
          "cells": {
            <cell_name>: <cell_details>,
            ...
          },
          "memories": {
            <memory_name>: <memory_details>,
            ...
          },
          "netnames": {
            <net_name>: <net_details>,
            ...
          }
        }
      },
      "models": {
        ...
      },
    }

Where <port_details> is:

    {
      "direction": <"input" | "output" | "inout">,
      "bits": <bit_vector>
      "offset": <the lowest bit index in use, if non-0>
      "upto": <1 if the port bit indexing is MSB-first>
      "signed": <1 if the port is signed>
    }

The "offset" and "upto" fields are skipped if their value would be 0.
They don't affect connection semantics, and are only used to preserve original
HDL bit indexing.
And <cell_details> is:
```

(continues on next page)

(continued from previous page)

```
    {
      "hide_name": <1 | 0>,
      "type": <cell_type>,
      "model": <AIG model name, if -aig option used>,
      "parameters": {
        <parameter_name>: <parameter_value>,
        ...
      },
      "attributes": {
        <attribute_name>: <attribute_value>,
        ...
      },
      "port_directions": {
        <port_name>: <"input" | "output" | "inout">,
        ...
      },
      "connections": {
        <port_name>: <bit_vector>,
        ...
      },
    }

And <memory_details> is:

    {
      "hide_name": <1 | 0>,
      "attributes": {
        <attribute_name>: <attribute_value>,
        ...
      },
      "width": <memory width>
      "start_offset": <the lowest valid memory address>
      "size": <memory size>
    }

And <net_details> is:

    {
      "hide_name": <1 | 0>,
      "bits": <bit_vector>
      "offset": <the lowest bit index in use, if non-0>
      "upto": <1 if the port bit indexing is MSB-first>
      "signed": <1 if the port is signed>
    }

The "hide_name" fields are set to 1 when the name of this cell or net is
automatically created and is likely not of interest for a regular user.

The "port_directions" section is only included for cells for which the
interface is known.

Module and cell ports and nets can be single bit wide or vectors of multiple
```

(continues on next page)

(continued from previous page)

```
bits. Each individual signal bit is assigned a unique integer. The <bit_vector>
values referenced above are vectors of this integers. Signal bits that are
connected to a constant driver are denoted as string "0", "1", "x", or
"z" instead of a number.

Bit vectors (including integers) are written as string holding the binary
representation of the value. Strings are written as strings, with an appended
blank in cases of strings of the form /[01xz]* */.

For example the following Verilog code:

    module test(input x, y);
      (* keep *) foo #(.P(42), .Q(1337))
          foo_inst (.A({x, y}), .B({y, x}), .C({4'd10, {4{x}}}));
    endmodule

Translates to the following JSON output:

    {
      "creator": "Yosys 0.9+2406 (git sha1 fb1168d8, clang 9.0.1 -fPIC -Os)",
      "modules": {
        "test": {
          "attributes": {
            "cells_not_processed": "00000000000000000000000000000001",
            "src": "test.v:1.1-4.10"
          },
          "ports": {
            "x": {
              "direction": "input",
              "bits": [ 2 ]
            },
            "y": {
              "direction": "input",
              "bits": [ 3 ]
            }
          },
          "cells": {
            "foo_inst": {
              "hide_name": 0,
              "type": "foo",
              "parameters": {
                "P": "00000000000000000000000000101010",
                "Q": "00000000000000000000010100111001"
              },
              "attributes": {
                "keep": "00000000000000000000000000000001",
                "module_not_derived": "00000000000000000000000000000001",
                "src": "test.v:3.1-3.55"
              },
              "connections": {
                "A": [ 3, 2 ],
                "B": [ 2, 3 ],
```

(continues on next page)

(continued from previous page)

```
                "C": [ 2, 2, 2, 2, "0", "1", "0", "1" ]
              }
            }
          },
          "netnames": {
            "x": {
              "hide_name": 0,
              "bits": [ 2 ],
              "attributes": {
                "src": "test.v:1.19-1.20"
              }
            },
            "y": {
              "hide_name": 0,
              "bits": [ 3 ],
              "attributes": {
                "src": "test.v:1.22-1.23"
              }
            }
          }
        }
      }
    }

The models are given as And-Inverter-Graphs (AIGs) in the following form:

    "models": {
      <model_name>: [
        /*   0 */ [ <node-spec> ],
        /*   1 */ [ <node-spec> ],
        /*   2 */ [ <node-spec> ],
        ...
      ],
      ...
    },

The following node-types may be used:

    [ "port", <portname>, <bitindex>, <out-list> ]
      - the value of the specified input port bit

    [ "nport", <portname>, <bitindex>, <out-list> ]
      - the inverted value of the specified input port bit

    [ "and", <node-index>, <node-index>, <out-list> ]
      - the ANDed value of the specified nodes

    [ "nand", <node-index>, <node-index>, <out-list> ]
      - the inverted ANDed value of the specified nodes

    [ "true", <out-list> ]
      - the constant value 1
```

(continues on next page)

(continued from previous page)

```
    [ "false", <out-list> ]
      - the constant value 0

All nodes appear in topological order. I.e. only nodes with smaller indices
are referenced by "and" and "nand" nodes.

The optional <out-list> at the end of a node specification is a list of
output portname and bitindex pairs, specifying the outputs driven by this node.

For example, the following is the model for a 3-input 3-output $reduce_and cell
inferred by the following code:

    module test(input [2:0] in, output [2:0] out);
      assign in = &out;
    endmodule

    "$reduce_and:3U:3": [
      /*   0 */ [ "port", "A", 0 ],
      /*   1 */ [ "port", "A", 1 ],
      /*   2 */ [ "and", 0, 1 ],
      /*   3 */ [ "port", "A", 2 ],
      /*   4 */ [ "and", 2, 3, "Y", 0 ],
      /*   5 */ [ "false", "Y", 1, "Y", 2 ]
    ]

Future version of Yosys might add support for additional fields in the JSON
format. A program processing this format must ignore all unknown fields.
```

write_rtlil - write design to RTLIL file

```
yosys> help write_rtlil
```

```
write_rtlil [filename]
```

```
Write the current design to an RTLIL file. (RTLIL is a text representation
of a design in yosys's internal format.)
```

`-selected`	only write selected parts of the design.
`-sort`	sort design in-place (used to be default).

write_simplec - convert design to simple C code

```
yosys> help write_simplec
```

```
write_simplec [options] [filename]
```

```
Write simple C code for simulating the design. The C code written can be used to
simulate the design in a C environment, but the purpose of this command is to
generate code that works well with C-based formal verification.
```

```
-verbose          this will print the recursive walk used to export the␣
                  ↪modules.

-i8, -i16, -i32, -i64 the maximum integer bit width to use in the␣
                  ↪generated code.
```

```
THIS COMMAND IS UNDER CONSTRUCTION
```

write_smt2 - write design to SMT-LIBv2 file

```
yosys> help write_smt2
```

```
write_smt2 [options] [filename]
```

```
Write a SMT-LIBv2 [1] description of the current design. For a module with name
'<mod>' this will declare the sort '<mod>_s' (state of the module) and will
define and declare functions operating on that state.

The following SMT2 functions are generated for a module with name '<mod>'.
Some declarations/definitions are printed with a special comment. A prover
using the SMT2 files can use those comments to collect all relevant metadata
about the design.

    ; yosys-smt2-module <mod>
    (declare-sort |<mod>_s| 0)
        The sort representing a state of module <mod>.

    (define-fun |<mod>_h| ((state |<mod>_s|)) Bool (...))
        This function must be asserted for each state to establish the
        design hierarchy.

    ; yosys-smt2-input <wirename> <width>
    ; yosys-smt2-output <wirename> <width>
    ; yosys-smt2-register <wirename> <width>
    ; yosys-smt2-wire <wirename> <width>
    (define-fun |<mod>_n <wirename>| (|<mod>_s|) (_ BitVec <width>))
    (define-fun |<mod>_n <wirename>| (|<mod>_s|) Bool)
        For each port, register, and wire with the 'keep' attribute set an
        accessor function is generated. Single-bit wires are returned as Bool,
        multi-bit wires as BitVec.

    ; yosys-smt2-cell <submod> <instancename>
    (declare-fun |<mod>_h <instancename>| (|<mod>_s|) |<submod>_s|)
        There is a function like that for each hierarchical instance. It
        returns the sort that represents the state of the sub-module that
        implements the instance.

    (declare-fun |<mod>_is| (|<mod>_s|) Bool)
        This function must be asserted 'true' for initial states, and 'false'
        otherwise.

    (define-fun |<mod>_i| ((state |<mod>_s|)) Bool (...))
```

(continues on next page)

(continued from previous page)

```
        This function must be asserted 'true' for initial states. For
        non-initial states it must be left unconstrained.

    (define-fun |<mod>_t| ((state |<mod>_s|) (next_state |<mod>_s|)) Bool (...))
        This function evaluates to 'true' if the states 'state' and
        'next_state' form a valid state transition.

    (define-fun |<mod>_a| ((state |<mod>_s|)) Bool (...))
        This function evaluates to 'true' if all assertions hold in the state.

    (define-fun |<mod>_u| ((state |<mod>_s|)) Bool (...))
        This function evaluates to 'true' if all assumptions hold in the state.

    ; yosys-smt2-assert <id> <filename:linenum>
    (define-fun |<mod>_a <id>| ((state |<mod>_s|)) Bool (...))
        Each $assert cell is converted into one of this functions. The function
        evaluates to 'true' if the assert statement holds in the state.

    ; yosys-smt2-assume <id> <filename:linenum>
    (define-fun |<mod>_u <id>| ((state |<mod>_s|)) Bool (...))
        Each $assume cell is converted into one of this functions. The function
        evaluates to 'true' if the assume statement holds in the state.

    ; yosys-smt2-cover <id> <filename:linenum>
    (define-fun |<mod>_c <id>| ((state |<mod>_s|)) Bool (...))
        Each $cover cell is converted into one of this functions. The function
        evaluates to 'true' if the cover statement is activated in the state.

Options:
```

-verbose

```
this will print the recursive walk used to export the␣
↪modules.
```

-stbv

```
Use a BitVec sort to represent a state instead of an␣
↪uninterpreted
sort. As a side-effect this will prevent use of arrays␣
↪to model
memories.
```

-stdt

```
Use SMT-LIB 2.6 style datatypes to represent a state␣
↪instead of an
uninterpreted sort.
```

-nobv

```
disable support for BitVec (FixedSizeBitVectors theory).
↪ without this
option multi-bit wires are represented using the BitVec␣
↪sort and
support for coarse grain cells (incl. arithmetic) is␣
↪enabled.
```

```
-nomem                  disable support for memories (via ArraysEx theory).␣
                        ↪this option is
                        implied by -nobv. only $mem cells without merged␣
                        ↪registers in
                        read ports are supported. call "memory" with -nordff to␣
                        ↪make sure
                        that no registers are merged into $mem read ports. '
                        ↪<mod>_m' functions
                        will be generated for accessing the arrays that are␣
                        ↪used to represent
                        memories.

-wires                  create '<mod>_n' functions for all public wires. by␣
                        ↪default only ports,
                        registers, and wires with the 'keep' attribute are␣
                        ↪exported.

-tpl <template_file>    use the given template file. the line containing only␣
                        ↪the token '%%'
                        is replaced with the regular output of this command.

-solver-option <option> <value>
                        ↪`yosys-smt2-solver-option` directive for yosys-
                        ↪smtbmc to write
                        the given option as a `(set-option ...)` command in the␣
                        ↪SMT-LIBv2.

[1] For more information on SMT-LIBv2 visit http://smt-lib.org/ or read David
R. Cok's tutorial: https://smtlib.github.io/jSMTLIB/SMTLIBTutorial.pdf

---------------------Example:-------------------------------------------------

                        Consider the following module (test.v). We want to␣
                        ↪prove that the output can
                        never transition from a non-zero value to a zero value.

                        module test(input clk, output reg [3:0] y);
                        always @(posedge clk)
                        y <= (y << 1) | ^y;
                        endmodule

                        For this proof we create the following template (test.
                        ↪tpl).

                        ; we need QF_UFBV for this proof
                        (set-logic QF_UFBV)

                        ; insert the auto-generated code here
                        %%

                        ; declare two state variables s1 and s2
```

(continues on next page)

(continued from previous page)

```
(declare-fun s1 () test_s)
(declare-fun s2 () test_s)

; state s2 is the successor of state s1
(assert (test_t s1 s2))

; we are looking for a model with y non-zero in s1
(assert (distinct (|test_n y| s1) #b0000))

; we are looking for a model with y zero in s2
(assert (= (|test_n y| s2) #b0000))

; is there such a model?
(check-sat)

The following yosys script will create a 'test.smt2'␣
↪file for our proof:

read_verilog test.v
hierarchy -check; proc; opt; check -assert
write_smt2 -bv -tpl test.tpl test.smt2

Running 'cvc4 test.smt2' will print 'unsat' because y␣
↪can never transition
from non-zero to zero in the test design.
```

write_smv - write design to SMV file

```
yosys> help write_smv
```

```
write_smv [options] [filename]
```

```
Write an SMV description of the current design.
```

```
-verbose
    this will print the recursive walk used to export the␣
    ↪modules.

-tpl <template_file>
    use the given template file. the line containing only␣
    ↪the token '%%'
    is replaced with the regular output of this command.
```

```
THIS COMMAND IS UNDER CONSTRUCTION
```

write_spice - write design to SPICE netlist file

```
yosys> help write_spice
```

```
write_spice [options] [filename]
```

```
Write the current design to an SPICE netlist file.
```

-big_endian	generate multi-bit ports in MSB first order (default is LSB first)
-neg net_name	set the net name for constant 0 (default: Vss)
-pos net_name	set the net name for constant 1 (default: Vdd)
-buf DC|subckt_name	set the name for jumper element (default: DC) (used to connect different nets)
-nc_prefix	prefix for not-connected nets (default: _NC)
-inames	include names of internal ($-prefixed) nets in outputs (default is to use net numbers instead)
-top top_module	set the specified module as design top module

write_table - write design as connectivity table

```
yosys> help write_table
```

```
write_table [options] [filename]
```

```
Write the current design as connectivity table. The output is a tab-separated
ASCII table with the following columns:

  module name
  cell name
  cell type
  cell port
  direction
  signal

module inputs and outputs are output using cell type and port '-' and with
'pi' (primary input) or 'po' (primary output) or 'pio' as direction.
```

write_verilog - write design to Verilog file

```
yosys> help write_verilog
```

```
write_verilog [options] [filename]
```

```
Write the current design to a Verilog file.
```

-sv	with this option, SystemVerilog constructs like always_comb are used
-norename	without this option all internal object names (the ones with a dollar instead of a backslash prefix) are changed to short

(continues on next page)

(continued from previous page)

```
                    ↪names in the
                    format '_<number>_'.
-renameprefix <prefix>  insert this prefix in front of auto-generated instance␣
                    ↪names
-noattr             with this option no attributes are included in the␣
                    ↪output
-attr2comment       with this option attributes are included as comments in␣
                    ↪the output
-noexpr             without this option all internal cells are converted to␣
                    ↪Verilog
                    expressions.
-noparallelcase     With this option no parallel_case attributes are used.␣
                    ↪Instead, a case
                    statement that assigns don't-care values for priority␣
                    ↪dependent inputs
                    is generated.
-siminit            add initial statements with hierarchical refs to␣
                    ↪initialize FFs when
                    in -noexpr mode.
-nodec              32-bit constant values are by default dumped as decimal␣
                    ↪numbers,
                    not bit pattern. This option deactivates this feature␣
                    ↪and instead
                    will write out all constants in binary.
-decimal            dump 32-bit constants in decimal and without size and␣
                    ↪radix
-nohex              constant values that are compatible with hex output are␣
                    ↪usually
                    dumped as hex values. This option deactivates this␣
                    ↪feature and
                    instead will write out all constants in binary.
-nostr              Parameters and attributes that are specified as strings␣
                    ↪in the
                    original input will be output as strings by this back-
                    ↪end. This
                    deactivates this feature and instead will write string␣
                    ↪constants
                    as binary numbers.
```

-simple-lhs	Connection assignments with simple left hand side without concatenations.
-extmem	instead of initializing memories using assignments to individual elements, use the '$readmemh' function to read initialization data from a file. This data is written to a file named by appending a sequential index to the Verilog filename and replacing the extension with '.mem', e.g. 'write_verilog -extmem foo.v' writes 'foo-1.mem', 'foo-2.mem' and so on.
-defparam	use 'defparam' statements instead of the Verilog-2001 syntax for cell parameters.
-default_params	emit module parameter declarations from parameter_default_values.
-blackboxes	usually modules with the 'blackbox' attribute are ignored. with this option set only the modules with the 'blackbox' attribute are written to the output file.
-selected	only write selected modules. modules must be selected entirely or not at all.
-v	verbose output (print new names of all renamed wires and cells)

Note that RTLIL processes can't always be mapped directly to Verilog always blocks. This frontend should only be used to export an RTLIL netlist, i.e. after the "proc" pass has been used to convert all processes to logic networks and registers. A warning is generated when this command is called on a design with RTLIL processes.

write_xaiger - write design to XAIGER file

yosys> help **write_xaiger**

write_xaiger [options] [filename]

Write the top module (according to the (* top *) attribute or if only one module is currently selected) to an XAIGER file. Any non $_NOT_, $_AND_, (optionally

(continues on next page)

(continued from previous page)

```
$_DFF_N_, $_DFF_P_), or non (* abc9_box *) cells will be converted into psuedo-
inputs and pseudo-outputs. Whitebox contents will be taken from the equivalent
module in the '$abc9_holes' design, if it exists.
```

`-ascii`	write ASCII version of AIGER format
`-map <filename>`	write an extra file with port and box symbols
`-dff`	write $_DFF_[NP]_ cells

write_xaiger2 - (experimental) write module to XAIGER file

```
yosys> help write_xaiger2
```

> **Warning**
>
> This command is experimental

```
write_xaiger2 [options] [filename]
```

```
Write the selected module to a XAIGER file including the 'h' and 'a' extensions
with box information for ABC.
```

`-strash`	perform structural hashing while writing
`-flatten`	allow descending into submodules and write a flattened view of the design hierarchy starting at the selected top
`-mapping_prep`	after the file is written, prepare the module for reintegration of a mapping in a subsequent command. all cells which are not blackboxed nor whiteboxed are removed from the design as well as all wires which only connect to removed cells (conflicts with -flatten)
`-map2 <file>`	write a map2 file which 'read_xaiger2 -sc_mapping' can read to reintegrate a mapping (conflicts with -flatten)

10.1.3 Writing output files

read_aiger - read AIGER file

```
yosys> help read_aiger
```

```
read_aiger [options] [filename]
```

```
Load module from an AIGER file into the current design.
```

-module_name <module_name>	name of module to be created (default: <filename>)
-clk_name <wire_name>	if specified, AIGER latches to be transformed into $_DFF_P_ cells clocked by wire of this name. otherwise, $_FF_ cells will be used
-map <filename>	read file with port and latch symbols
-wideports	merge ports that match the pattern 'name[int]' into a single multi-bit port 'name'
-xaiger	read XAIGER extensions

read_blif - read BLIF file

```
yosys> help read_blif
```

```
read_blif [options] [filename]
```

```
Load modules from a BLIF file into the current design.
```

-sop	Create $sop cells instead of $lut cells
-wideports	Merge ports that match the pattern 'name[int]' into a single multi-bit port 'name'.

read_json - read JSON file

```
yosys> help read_json
```

```
read_json [filename]
```

```
Load modules from a JSON file into the current design See "help write_json"
for a description of the file format.
```

read_liberty - read cells from liberty file

```
yosys> help read_liberty
```

```
read_liberty [filename]
```

```
Read cells from liberty file as modules into current design.
```

-lib	only create empty blackbox modules
-wb	mark imported cells as whiteboxes
-nooverwrite	ignore re-definitions of modules. (the default behavior is to create an error message if the existing module is not a blackbox module, and overwrite the existing module if it is a blackbox module.)
-overwrite	overwrite existing modules with the same name
-ignore_miss_func	ignore cells with missing function specification of outputs
-ignore_miss_dir	ignore cells with a missing or invalid direction specification on a pin
-ignore_miss_data_latch	ignore latches with missing data and/or enable pins
-ignore_buses	ignore cells with bus interfaces (wide ports)
-setattr <attribute_name>	set the specified attribute (to the value 1) on all loaded modules
-unit_delay	import combinational timing arcs under the unit delay model

read_rtlil - read modules from RTLIL file

```
yosys> help read_rtlil
```

```
read_rtlil [filename]
```

```
Load modules from an RTLIL file to the current design. (RTLIL is a text
representation of a design in yosys's internal format.)
```

-nooverwrite	ignore re-definitions of modules. (the default behavior is to create an error message if the existing module is not a blackbox module, and overwrite the existing module if it is a blackbox module.)
-overwrite	overwrite existing modules with the same name

-lib	only create empty blackbox modules
-legalize	prevent semantic errors (e.g. reference to unknown wire, ↪ redefinition of wire/cell) by deterministically rewriting the input into something␣ ↪valid. Useful when using fuzzing to generate random but valid RTLIL.

read_verilog - read modules from Verilog file

```
yosys> help read_verilog
```

```
read_verilog [options] [filename]
```

```
Load modules from a Verilog file to the current design. A large subset of
Verilog-2005 is supported.
```

-sv	enable support for SystemVerilog features. (only a␣ ↪small subset of SystemVerilog is supported)
-formal	enable support for SystemVerilog assertions and some␣ ↪Yosys extensions replace the implicit -D SYNTHESIS with -D FORMAL
-nosynthesis	don't add implicit -D SYNTHESIS
-noassert	ignore assert() statements
-noassume	ignore assume() statements
-norestrict	ignore restrict() statements
-assume-asserts	treat all assert() statements like assume() statements
-assert-assumes	treat all assume() statements like assert() statements
-nodisplay	suppress output from display system tasks ($display et.␣ ↪al). This does not affect the output from a later 'sim'␣ ↪command.
-debug	alias for -dump_ast1 -dump_ast2 -dump_vlog1 -dump_vlog2␣ ↪-yydebug
-dump_ast1	dump abstract syntax tree (before simplification)
-dump_ast2	dump abstract syntax tree (after simplification)

```
-no_dump_ptr
    do not include hex memory addresses in dump (easier to␣
    ↪diff dumps)

-dump_vlog1
    dump ast as Verilog code (before simplification)

-dump_vlog2
    dump ast as Verilog code (after simplification)

-dump_rtlil
    dump generated RTLIL netlist

-yydebug
    enable parser debug output

-nolatches
    usually latches are synthesized into logic loops
    this option prohibits this and sets the output to 'x'
    in what would be the latches hold condition

    this behavior can also be achieved by setting the
    'nolatches' attribute on the respective module or
    always block.

-nomem2reg
    under certain conditions memories are converted to␣
    ↪registers
    early during simplification to ensure correct handling␣
    ↪of
    complex corner cases. this option disables this␣
    ↪behavior.

    this can also be achieved by setting the 'nomem2reg'
    attribute on the respective module or register.

    This is potentially dangerous. Usually the front-end␣
    ↪has good
    reasons for converting an array to a list of registers.
    Prohibiting this step will likely result in incorrect␣
    ↪synthesis
    results.

-mem2reg
    always convert memories to registers. this can also be
    achieved by setting the 'mem2reg' attribute on the␣
    ↪respective
    module or register.

-nomeminit
    do not infer $meminit cells and instead convert␣
    ↪initialized
    memories to registers directly in the front-end.

-ppdump
    dump Verilog code after pre-processor

-nopp
    do not run the pre-processor
```

-nodpi	disable DPI-C support
-noblackbox	do not automatically add a (* blackbox *) attribute to an empty module.
-lib	only create empty blackbox modules. This implies -DBLACKBOX. modules with the (* whitebox *) attribute will be preserved. (* lib_whitebox *) will be treated like (* whitebox *).
-nowb	delete (* whitebox *) and (* lib_whitebox *) attributes from all modules.
-specify	parse and import specify blocks
-noopt	don't perform basic optimizations (such as const folding) in the high-level front-end.
-icells	interpret cell types starting with '$' as internal cell types
-pwires	add a wire for each module parameter
-nooverwrite	ignore re-definitions of modules. (the default behavior is to create an error message if the existing module is not a black box module, and overwrite the existing module otherwise.)
-overwrite	overwrite existing modules with the same name
-defer	only read the abstract syntax tree and defer actual compilation to a later 'hierarchy' command. Useful in cases where the default parameters of modules yield invalid or not synthesizable code.
-noautowire	make the default of `default_nettype be "none" instead of "wire".
-setattr <attribute_name>	set the specified attribute (to the value 1) on all loaded modules

```
-Dname[=definition]      define the preprocessor symbol 'name' and set its␣
                        ↪optional value
                        'definition'

-Idir                   add 'dir' to the directories which are used when␣
                        ↪searching include
                        files

-relativeshare          use paths relative to share directory for source␣
                        ↪locations
                        where possible (experimental).
```

```
The command 'verilog_defaults' can be used to register default options for
subsequent calls to 'read_verilog'.

Note that the Verilog frontend does a pretty good job of processing valid
verilog input, but has not very good error reporting. It generally is
recommended to use a simulator (for example Icarus Verilog) for checking
the syntax of the code, rather than to rely on read_verilog for that.

Depending on if read_verilog is run in -formal mode, either the macro
SYNTHESIS or FORMAL is defined automatically, unless -nosynthesis is used.
In addition, read_verilog always defines the macro YOSYS.

See the Yosys README file for a list of non-standard Verilog features
supported by the Yosys Verilog front-end.
```

read_verilog_file_list - parse a Verilog file list

```
yosys> help read_verilog_file_list
```

```
read_verilog_file_list [options]
```

```
Parse a Verilog file list, and pass the list of Verilog files to read_verilog
command
```

```
-F file_list_path       File list file contains list of Verilog files to be␣
                        ↪parsed, any path is
                        treated relative to the file list file

-f file_list_path       File list file contains list of Verilog files to be␣
                        ↪parsed, any path is
                        treated relative to current working directroy
```

read_xaiger2 - (experimental) read XAIGER file

```
yosys> help read_xaiger2
```

> **Warning**
>
> This command is experimental

```
read_xaiger2 -sc_mapping [options] <filename>
```

```
Read a standard cell mapping from a XAIGER file into an existing module.
```

`-module_name <name>`	name of the target module
`-map2 <filename>`	read file with symbol information

10.1.4 Yosys kernel commands

> **Warning**
>
> No commands found for group 'kernel'
>
> Documentation may have been built without `source_location` support. Try check /cmd/index_other.

10.1.5 Formal verification

assertpmux - adds asserts for parallel muxes

```
yosys> help assertpmux
```

```
assertpmux [options] [selection]
```

```
This command adds asserts to the design that assert that all parallel muxes
($pmux cells) have a maximum of one of their inputs enable at any time.
```

`-noinit`	do not enforce the pmux condition during the init state
`-always`	usually the $pmux condition is only checked when the $pmux output is used by the mux tree it drives. this option will deactivate this additional constraint and check the $pmux condition always.

async2sync - convert async FF inputs to sync circuits

```
yosys> help async2sync
```

```
async2sync [options] [selection]
```

```
This command replaces async FF inputs with sync circuits emulating the same
behavior for when the async signals are actually synchronized to the clock.
```

(continues on next page)

(continued from previous page)

```
This pass assumes negative hold time for the async FF inputs. For example when
a reset deasserts with the clock edge, then the FF output will still drive the
reset value in the next cycle regardless of the data-in value at the time of
the clock edge.
```

-nolower

```
Do not automatically run 'chformal -lower' to lower
↪$check cells.
```

chformal - change formal constraints of the design

```
yosys> help chformal
```

chformal [types] [mode] [options] [selection]

Make changes to the formal constraints of the design. The [types] options the type of constraint to operate on. If none of the following options are given, the command will operate on all constraint types:

-assert	*$assert* cells, representing `assert(...)` constraints
-assume	*$assume* cells, representing `assume(...)` constraints
-live	*$live* cells, representing `assert(s_eventually ...)`
-fair	*$fair* cells, representing `assume(s_eventually ...)`
-cover	*$cover* cells, representing `cover()` statements

Additionally chformal will operate on *$check* cells corresponding to the selected constraint types.

Exactly one of the following modes must be specified:

-remove	remove the cells and thus constraints from the design
-early	bypass FFs that only delay the activation of a constraint. When inputs of the bypassed FFs do not remain stable between clock edges, this may result in unexpected behavior.
-delay <N>	delay activation of the constraint by <N> clock cycles
-skip <N>	ignore activation of the constraint in the first <N> clock cycles
-coverenable	add cover statements for the enable signals of the constraints
-assert2assume	
-assert2cover	
-assume2assert	
-live2fair	
-fair2live	change the roles of cells as indicated. these options can be combined
-lower	convert each $check cell into an $assert, $assume, $live, $fair or $cover cell. If the $check cell contains a message, also produce a $print cell.

clk2fflogic - convert clocked FFs to generic $ff cells

```
yosys> help clk2fflogic
```

```
clk2fflogic [options] [selection]
```

```
This command replaces clocked flip-flops with generic $ff cells that use the
implicit global clock. This is useful for formal verification of designs with
multiple clocks.

This pass assumes negative hold time for the async FF inputs. For example when
a reset deasserts with the clock edge, then the FF output will still drive the
reset value in the next cycle regardless of the data-in value at the time of
the clock edge.
```

```
-nolower        Do not automatically run 'chformal -lower' to lower
                ↪$check cells.

-nopeepopt      Do not automatically run 'peepopt -formalclk' to␣
                ↪rewrite clock patterns
                to more formal friendly forms.
```

cutpoint - adds formal cut points to the design

```
yosys> help cutpoint
```

```
cutpoint [options] [selection]
```

```
This command adds formal cut points to the design.
```

```
-undef          set cutpoint nets to undef (x). the default behavior is␣
                ↪to create
                an $anyseq cell and drive the cutpoint net from that

-noscopeinfo    do not create '$scopeinfo' cells that preserve␣
                ↪attributes of cells that
                were removed by this pass

                cutpoint -blackbox [options]
```

```
Replace all instances of blackboxes in the design with a formal cut point.
```

dft_tag - create tagging logic for data flow tracking

```
yosys> help dft_tag
```

```
dft_tag [options] [selection]
```

```
This pass... TODO
```

```
-overwrite-only     Only process $overwrite_tag and $original_tag cells.

-tag-public         For each public wire that may carry tagged data, create␣
                    ↪a new public
                    wire (named <wirename>:<tagname>) that carries the tag␣
                    ↪bits. Note
                    that without this, tagging logic will only be emitted␣
                    ↪as required
                    for uses of $get_tag.
```

fmcombine - combine two instances of a cell into one

```
yosys> help fmcombine
```

```
fmcombine [options] module_name gold_cell gate_cell
```

```
This pass takes two cells, which are instances of the same module, and replaces
them with one instance of a special 'combined' module, that effectively
contains two copies of the original module, plus some formal properties.

This is useful for formal test benches that check what differences in behavior
a slight difference in input causes in a module.

-initeq             Insert assumptions that initially all FFs in both␣
                    ↪circuits have the
                    same initial values.

-anyeq              Do not duplicate $anyseq/$anyconst cells.

-fwd                Insert forward hint assumptions into the combined␣
                    ↪module.

-bwd                Insert backward hint assumptions into the combined␣
                    ↪module.
                    (Backward hints are logically equivalend to fordward␣
                    ↪hits, but
                    some solvers are faster with bwd hints, or even both -
                    ↪bwd and -fwd.)

-nop                Don't insert hint assumptions into the combined module.
                    (This should not provide any speedup over the original␣
                    ↪design, but
                    strangely sometimes it does.)

If none of -fwd, -bwd, and -nop is given, then -fwd is used as default.
```

fminit - set init values/sequences for formal

```
yosys> help fminit
```

```
fminit [options] <selection>
```

```
This pass creates init constraints (for example for reset sequences) in a formal
model.
```

```
-seq <signal> <sequence>
        Set sequence using comma-separated list of values, use 'z for
        unconstrained bits. The last value is used for the remainder of the
        trace.

-set <signal> <value>
        Add constant value constraint

-posedge <signal>

-negedge <signal>
        Set clock for init sequences
```

formalff - prepare FFs for formal

```
yosys> help formalff
```

```
formalff [options] [selection]
```

```
This pass transforms clocked flip-flops to prepare a design for formal
verification. If a design contains latches and/or multiple different clocks run
the async2sync or clk2fflogic passes before using this pass.
```

```
-clk2ff
        Replace all clocked flip-flops with $ff cells that use the implicit
        global clock. This assumes, without checking, that the design uses a
        single global clock. If that is not the case, the clk2fflogic pass
        should be used instead.

-ff2anyinit
        Replace uninitialized bits of $ff cells with $anyinit cells. An
        $anyinit cell behaves exactly like an $ff cell with an undefined
        initialization value. The difference is that $anyinit inhibits
        don't-care optimizations and is used to track solver-provided values
        in witness traces.

        If combined with -clk2ff this also affects newly created $ff cells.
```

```
-anyinit2ff      Replaces $anyinit cells with uninitialized $ff cells.␣
                 ↪This performs the
                 reverse of -ff2anyinit and can be used, before running␣
                 ↪a backend pass
                 (or similar) that is not yet aware of $anyinit cells.

                 Note that after running -anyinit2ff, in general,␣
                 ↪performing don't-care
                 optimizations is not sound in a formal verification␣
                 ↪setting.

-fine            Emit fine-grained $_FF_ cells instead of coarse-grained
                 ↪$ff cells for
```

-anyinit2ff. Cannot be combined with -clk2ff or -ff2anyinit.

```
-setundef        Find FFs with undefined initialization values for which␣
                 ↪changing the
                 initialization does not change the observable behavior␣
                 ↪and initialize
                 them. For -ff2anyinit, this reduces the number of␣
                 ↪generated $anyinit
                 cells that drive wires with private names.

-hierarchy       Propagates the 'replaced_by_gclk' attribute set by␣
                 ↪clk2ff upwards
                 through the design hierarchy towards the toplevel␣
                 ↪inputs. This option
                 works on the whole design and ignores the selection.

-assume          Add assumptions that constrain wires with the 'replaced_
                 ↪by_gclk'
                 attribute to the value they would have before an active␣
                 ↪clock edge.

-declockgate     Detect clock-gating patterns and modify any FFs clocked␣
                 ↪by the gated
                 clock to use the ungated clock with the gate signal as␣
                 ↪clock enable.
                 This doesn't affect the design's behavior during FV but␣
                 ↪can enable the
                 use of formal verification methods that only support a␣
                 ↪single global
                 clock.
```

freduce - perform functional reduction

```
yosys> help freduce

    freduce [options] [selection]
```

```
This pass performs functional reduction in the circuit. I.e. if two nodes are
equivalent, they are merged to one node and one of the redundant drivers is
disconnected. A subsequent call to 'clean' will remove the redundant drivers.
```

`-v, -vv`	enable verbose or very verbose output
`-inv`	enable explicit handling of inverted signals
`-stop <n>`	stop after <n> reduction operations. this is mostly used for debugging the freduce command itself.
`-dump <prefix>`	dump the design to <prefix>_<module>_<num>.il after each reduction operation. this is mostly used for debugging the freduce command.

```
This pass is undef-aware, i.e. it considers don't-care values for detecting
equivalent nodes.

All selected wires are considered for rewiring. The selected cells cover the
circuit that is analyzed.
```

future - resolve future sampled value functions

```
yosys> help future
```

```
future [options] [selection]
```

glift - create GLIFT models and optimization problems

```
yosys> help glift
```

```
glift <command> [options] [selection]
```

```
Augments the current or specified module with gate-level information flow
tracking (GLIFT) logic using the "constructive mapping" approach. Also can set
up QBF-SAT optimization problems in order to optimize GLIFT models or trade off
precision and complexity.

Commands:
```

`-create-precise-model`	Replaces the current or specified module with one that has corresponding "taint" inputs, outputs, and internal nets along with precise taint tracking logic. For example, precise taint tracking logic for an AND gate is: `y_t = a & b_t | b & a_t | a_t & b_t`

```
-create-imprecise-model    Replaces the current or specified module with one that
                           ↪has corresponding
                           "taint" inputs, outputs, and internal nets along with
                           ↪imprecise "All OR"
                           taint tracking logic:

                           y_t = a_t | b_t

-create-instrumented-model Replaces the current or specified module with one that
                           ↪has corresponding
                           "taint" inputs, outputs, and internal nets along with 4
                           ↪varying-precision
                           versions of taint tracking logic. Which version of
                           ↪taint tracking logic is
                           used for a given gate is determined by a MUX selected
                           ↪by an $anyconst cell.
                           By default, unless the `-no-cost-model` option is
                           ↪provided, an additional
                           wire named `__glift_weight` with the `keep` and
                           ↪`minimize` attributes is
                           added to the module along with pmuxes and adders to
                           ↪calculate a rough
                           estimate of the number of logic gates in the GLIFT
                           ↪model given an assignment
                           for the $anyconst cells. The four versions of taint
                           ↪tracking logic for an
                           AND gate are:
                           y_t = a & b_t | b & a_t | a_t & b_t       (like `-
                           ↪create-precise-model`)
                           y_t = a_t | a & b_t
                           y_t = b_t | b & a_t
                           y_t = a_t | b_t                          (like `-
                           ↪create-imprecise-model`)
```

```
Options:
```

```
-taint-constants           Constant values in the design are labeled as tainted.
                           (default: label constants as un-tainted)

-keep-outputs              Do not remove module outputs. Taint tracking outputs
                           ↪will appear in the
                           module ports alongside the orignal outputs.
                           (default: original module outputs are removed)

-simple-cost-model         Do not model logic area. Instead model the number of
                           ↪non-zero assignments to
                           $anyconsts. Taint tracking logic versions vary in their
                           ↪size, but all
                           reduced-precision versions are significantly smaller
                           ↪than the fully-precise
```

(continues on next page)

(continued from previous page)

```
version. A non-zero $anyconst assignment means that␣
↪reduced-precision taint
tracking logic was chosen for some gate. Only␣
↪applicable in combination with
`-create-instrumented-model`. (default: use a complex␣
↪model and give that
wire the "keep" and "minimize" attributes)
```

-no-cost-model

```
Do not model taint tracking logic area and do not␣
↪create a `__glift_weight`
wire. Only applicable in combination with `-create-
↪instrumented-model`.
(default: model area and give that wire the "keep" and
↪"minimize"
attributes)
```

-instrument-more

```
Allow choice from more versions of (even simpler) taint␣
↪tracking logic. A
total of 8 versions of taint tracking logic will be␣
↪added per gate,
including the 4 versions from `-create-instrumented-
↪model` and these
additional versions:

y_t = a_t
y_t = b_t
y_t = 1
y_t = 0

Only applicable in combination with `-create-
↪instrumented-model`.
(default: do not add more versions of taint tracking␣
↪logic.
```

miter - automatically create a miter circuit

```
yosys> help miter
```

miter -equiv [options] gold_name gate_name miter_name

```
Creates a miter circuit for equivalence checking. The gold- and gate- modules
must have the same interfaces. The miter circuit will have all inputs of the
two source modules, prefixed with 'in_'. The miter circuit has a 'trigger'
output that goes high if an output mismatch between the two source modules is
detected.
```

-ignore_gold_x

```
a undef (x) bit in the gold module output will match␣
↪any value in
the gate module output.
```

-make_outputs	also route the gold- and gate-outputs to 'gold_*' and 'gate_*' outputs on the miter circuit.
-make_outcmp	also create a cmp_* output for each gold/gate output pair.
-make_assert	also create an 'assert' cell that checks if trigger is always low.
-make_cover	also create a 'cover' cell for each gold/gate output pair.
-flatten	call 'flatten -wb; opt_expr -keepdc -undriven;;' on the miter circuit.
-cross	allow output ports on the gold module to match input ports on the gate module. This is useful when the gold module contains additional logic to drive some of the gate module inputs.

miter -assert [options] module [miter_name]

Creates a miter circuit for property checking. All input ports are kept, output ports are discarded. An additional output 'trigger' is created that goes high when an assert is violated. Without a miter_name, the existing module is modified.

-make_outputs	keep module output ports.
-flatten	call 'flatten -wb; opt_expr -keepdc -undriven;;' on the miter circuit.

mutate - generate or apply design mutations

yosys> help **mutate**

mutate -list N [options] [selection]

Create a list of N mutations using an even sampling.

-o filename	Write list to this file instead of console output
-s filename	Write a list of all src tags found in the design to the specified file
-seed N	RNG seed for selecting mutations

-none	Include a "none" mutation in the output
-ctrl name width value	Add -ctrl options to the output. Use 'value' for first mutation, then simply count up from there.
-mode name	
-module name	
-cell name	
-port name	
-portbit int	
-ctrlbit int	
-wire name	
-wirebit int	
-src string	Filter list of mutation candidates to those matching the given parameters.
-cfg option int	Set a configuration option. Options available: weight_pq_w weight_pq_b weight_pq_c weight_pq_s weight_pq_mw weight_pq_mb weight_pq_mc weight_pq_ms weight_cover pick_cover_prcnt

mutate -mode MODE [options]

Apply the given mutation.

-ctrl name width value	Add a control signal with the given name and width. The mutation is activated if the control signal equals the given value.
-module name	
-cell name	
-port name	
-portbit int	
-ctrlbit int	Mutation parameters, as generated by 'mutate -list N'.
-wire name	
-wirebit int	
-src string	Ignored. (They are generated by -list for documentation purposes.)

qbfsat - solve a 2QBF-SAT problem in the circuit

```
yosys> help qbfsat
```

```
qbfsat [options] [selection]
```

```
This command solves an "exists-forall" 2QBF-SAT problem defined over the
currently selected module. Existentially-quantified variables are declared by
assigning a wire "$anyconst". Universally-quantified variables may be
explicitly declared by assigning a wire "$allconst", but module inputs will be
treated as universally-quantified variables by default.
```

```
-nocleanup
    Do not delete temporary files and directories. Useful for debugging.

-dump-final-smt2 <file>
    Pass the --dump-smt2 option to yosys-smtbmc.

-assume-outputs
    Add an "$assume" cell for the conjunction of all one-bit module output
    wires.

-assume-negative-polarity
    When adding $assume cells for one-bit module output wires, assume they
    are negative polarity signals and should always be low, for example like
    the miters created with the `miter` command.

-nooptimize
    Ignore "\minimize" and "\maximize" attributes, do not emit
    "(maximize)" or "(minimize)" in the SMT-LIBv2, and generally make no
    attempt to optimize anything.

-nobisection
    If a wire is marked with the "\minimize" or "\maximize" attribute,
    do not attempt to optimize that value with the default iterated solving
    and threshold bisection approach. Instead, have yosys-smtbmc emit a
    "(minimize)" or "(maximize)" command in the SMT-LIBv2 output and
    hope that the solver supports optimizing quantified bitvector problems.

-solver <solver>
    Use a particular solver. Choose one of: "z3", "yices", "cvc4"
    and "cvc5". (default: yices)

-solver-option <name> <value>
    Set the specified solver option in the SMT-LIBv2 problem file.
```

```
-timeout <value>        Set the per-iteration timeout in seconds.
                        (default: no timeout)

-O0, -O1, -O2           Control the use of ABC to simplify the QBF-SAT problem␣
                        ↪before solving.

-sat                    Generate an error if the solver does not return "sat".

-unsat                  Generate an error if the solver does not return "unsat".

-show-smtbmc            Print the output from yosys-smtbmc.

-specialize             If the problem is satisfiable, replace each "$anyconst"␣
                        ↪cell with its
                        corresponding constant value from the model produced by␣
                        ↪the solver.

-specialize-from-file <solution file>
                        Do not run the solver, but instead only attempt to␣
                        ↪replace each
                        "$anyconst" cell in the current module with a constant␣
                        ↪value provided
                        by the specified file.

-write-solution <file>  If the problem is satisfiable, write the corresponding␣
                        ↪constant value
                        for each "$anyconst" cell from the model produced by␣
                        ↪the solver to the
                        specified file.
```

sat - solve a SAT problem in the circuit

```
yosys> help sat
```

```
sat [options] [selection]

This command solves a SAT problem defined over the currently selected circuit
and additional constraints passed as parameters.

-all                    show all solutions to the problem (this can grow␣
                        ↪exponentially, use

-max <N> instead to get <N> solutions)

-max <N>                like -all, but limit number of solutions to <N>

-enable_undef           enable modeling of undef value (aka 'x-bits')
                        this option is implied by -set-def, -set-undef et.␣
                        ↪cetera
```

```
    -max_undef                  maximize the number of undef bits in solutions, giving␣
                                ↪a better
                                picture of which input bits are actually vital to the␣
                                ↪solution.

    -set <signal> <value>       set the specified signal to the specified value.

    -set-def <signal>           add a constraint that all bits of the given signal must␣
                                ↪be defined

    -set-any-undef <signal>     add a constraint that at least one bit of the given␣
                                ↪signal is undefined

    -set-all-undef <signal>     add a constraint that all bits of the given signal are␣
                                ↪undefined

    -set-def-inputs             add -set-def constraints for all module inputs

    -set-def-formal             add -set-def constraints for formal $anyinit, $anyconst,
                                ↪ $anyseq cells

    -show <signal>              show the model for the specified signal. if no -show␣
                                ↪option is
                                passed then a set of signals to be shown is␣
                                ↪automatically selected.

    -show-inputs, -show-outputs add all module (input/output) ports to the list of␣
                                ↪shown signals

    -show-regs, -show-public, -show-all
                                show all registers, show signals with 'public' names,␣
                                ↪show all signals

    -ignore-div-by-zero         ignore all solutions that involve a division by zero

    -ignore-unknown-cells       ignore all cells that can not be matched to a SAT model

The following options can be used to set up a sequential problem:

    -seq <N>                    set up a sequential problem with <N> time steps. The␣
                                ↪steps will
                                be numbered from 1 to N.

                                note: for large <N> it can be significantly faster to␣
                                ↪use

    -tempinduct-baseonly -maxsteps <N> instead of -seq <N>.

    -set-at <N> <signal> <value>
```

```
-unset-at <N> <signal>    for unset the specified signal to the specified␣
                          ↪value in the
                          given timestep. this has priority over a -set for the␣
                          ↪same signal.

-set-assumes              set all assumptions provided via $assume cells

-set-def-at <N> <signal>

-set-any-undef-at <N> <signal>

-set-all-undef-at <N>     set all constraints in the given timestep.

-set-init <signal> <value>  the initial value for the register driving the␣
                          ↪signal to the value

-set-init-undef           set all initial states (not set using -set-init) to␣
                          ↪undef

-set-init-def             do not force a value for the initial state but do not␣
                          ↪allow undef

-set-init-zero            set all initial states (not set using -set-init) to zero

-dump_vcd <vcd-filename>  dump SAT model (counter example in proof) to VCD file

-dump_json <json-filename>  SAT model (counter example in proof) to a WaveJSON␣
                          ↪file.

-dump_cnf <cnf-filename>  CNF of SAT problem (in DIMACS format). in temporal␣
                          ↪induction
                          proofs this is the CNF of the first induction step.
```

```
The following additional options can be used to set up a proof. If also -seq
is passed, a temporal induction proof is performed.
```

```
-tempinduct               Perform a temporal induction proof. In a temporal␣
                          ↪induction proof it is
                          proven that the condition holds forever after the␣
                          ↪number of time steps
                          specified using -seq.

-tempinduct-def           Perform a temporal induction proof. Assume an initial␣
                          ↪state with all
                          registers set to defined values for the induction step.

-tempinduct-baseonly      Only only the basecase half of temporal induction␣
                          ↪(requires -maxsteps)
```

-tempinduct-induction-only	Run only the induction half of temporal induction
-tempinduct-skip <N>	Skip the first <N> steps of the induction proof. note: this will assume that the base case holds for <N>␣ ↪steps. this must be proven independently with "-tempinduct- ↪baseonly
-maxsteps <N>". Use -initsteps if you just want to set a	Use initial induction length
-prove <signal> <value>	Attempt to proof that <signal> is always <value>.
-prove-x <signal> <value>	Like -prove, but an undef (x) bit in the lhs matches␣ ↪any value on the right hand side. Useful for equivalence checking.
-prove-asserts	Prove that all asserts in the design hold.
-prove-skip <N>	Do not enforce the prove-condition for the first <N>␣ ↪time steps.
-maxsteps <N>	Set a maximum length for the induction.
-initsteps <N>	Set initial length for the induction. This will speed up the search of the right induction␣ ↪length for deep induction proofs.
-stepsize <N>	Increase the size of the induction proof in steps of <N> ↪. This will speed up the search of the right induction␣ ↪length for deep induction proofs.
-timeout <N>	Maximum number of seconds a single SAT instance may␣ ↪take.
-select-solver <name>	Select SAT solver implementation for this invocation. If not given, uses scratchpad key 'sat.solver' if set,␣ ↪otherwise default.
-verify	Return an error and stop the synthesis script if the␣ ↪proof fails.
-verify-no-timeout	Like -verify but do not return an error for timeouts.
-falsify	Return an error and stop the synthesis script if the␣ ↪proof succeeds.

```
-falsify-no-timeout    like -falsify but do not return an error for timeouts.
```

supercover - add hi/lo cover cells for each wire bit

```
yosys> help supercover
```

```
supercover [options] [selection]
```

```
This command adds two cover cells for each bit of each selected wire, one
checking for a hi signal level and one checking for lo level.
```

synthprop - synthesize SVA properties

```
yosys> help synthprop
```

```
synthprop [options]
```

```
This creates synthesizable properties for the selected module.
```

```
-name <portname>       name of the output port for assertions (default:␣
                       ↪assertions).

-map <filename>        write the port mapping for synthesizable properties␣
                       ↪into the given file.

-or_outputs            Or all outputs together to create a single output that␣
                       ↪goes high when
                       any property is violated, instead of generating␣
                       ↪individual output bits.

-reset <portname>      name of the top-level reset input. Latch a high state␣
                       ↪on the generated
                       outputs until an asynchronous top-level reset input is␣
                       ↪activated.

-resetn <portname>     like above but with inverse polarity
```

xprop - formal x propagation

```
yosys> help xprop
```

```
xprop [options] [selection]
```

```
This pass transforms the circuit into an equivalent circuit that explicitly
encodes the propagation of x values using purely 2-valued logic. On the
interface between xprop-transformed and non-transformed parts of the design,
appropriate conversions are inserted automatically.
```

```
-split-inputs

-split-outputs
```

```
-split-ports
    Replace each input/output/port with two new ports, one carrying the
    defined values (named <portname>_d) and one carrying the mask of which
    bits are x (named <portname>_x). When a bit in the <portname>_x is set
    the corresponding bit in <portname>_d is ignored for inputs and
    guaranteed to be 0 for outputs.

-split-public
    Replace each public non-port wire with two new wires, one carrying the
    defined values (named <wirename>_d) and one carrying the mask of which
    bits are x (named <wirename>_x). When a bit in the <portname>_x is set
    the corresponding bit in <wirename>_d is guaranteed to be 0 for
    outputs.

-assume-encoding
    Add encoding invariants as assumptions. This can speed up formal
    verification tasks.

-assert-encoding
    Add encoding invariants as assertions. Used for testing the xprop
    pass itself.

-assume-def-inputs
    Assume all inputs are fully defined. This adds corresponding
    assumptions to the design and uses these assumptions to optimize the
    xprop encoding.

-required
    Produce a runtime error if any encountered cell could not be encoded.

-formal
    Produce a runtime error if any encoded cell uses a signal that is
neither known to be non-x nor driven by another encoded cell.

-debug-asserts
    Add assertions checking that the encoding used by this pass never
    produces x values within the encoded signals.
```

10.1.6 Passes

Working with hierarchy

Warning

No commands found for group 'passes/hierarchy'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Converting process blocks

proc - translate processes to netlists

```
yosys> help proc
```

```
proc [options] [selection]
```

```
This pass calls all the other proc_* passes in the most common order.

    proc_clean
    proc_rmdead
    proc_prune
    proc_init
    proc_arst
    proc_rom
    proc_mux
    proc_dlatch
    proc_dff
    proc_memwr
    proc_clean
    opt_expr -keepdc

This replaces the processes in the design with multiplexers,
flip-flops and latches.

The following options are supported:
```

```
-nomux                  Will omit the proc_mux pass.

-norom                  Will omit the proc_rom pass.

-global_arst [!]<netname>   This option is passed through to proc_arst.

-ifx                    This option is passed through to proc_mux. proc_rmdead␣
                        ↪is not
                        executed in -ifx mode.

-noopt                  Will omit the opt_expr pass.
```

proc_arst - detect asynchronous resets

yosys> help **proc_arst**

proc_arst [-global_arst [!]<netname>] [selection]

```
This pass identifies asynchronous resets in the processes and converts them
to a different internal representation that is suitable for generating
flip-flop cells with asynchronous resets.
```

-global_arst [!]<netname>

```
Instances that have a net with the given name, use␣
↪this net as async
reset for registers that have been assign initial␣
↪values in their
declaration ('reg foobar = constant_value;'). Use the '!
↪' modifier for
active low reset signals. Note: the frontend stores the␣
↪default value
in the 'init' attribute on the net.
```

proc_clean - remove empty parts of processes

yosys> help **proc_clean**

proc_clean [options] [selection]

-quiet

```
do not print any messages.
```

```
This pass removes empty parts of processes and ultimately removes a process
if it contains only empty structures.
```

proc_dff - extract flip-flops from processes

yosys> help **proc_dff**

proc_dff [selection]

```
This pass identifies flip-flops in the processes and converts them to
d-type flip-flop cells.
```

proc_dlatch - extract latches from processes

yosys> help **proc_dlatch**

proc_dlatch [selection]

```
This pass identifies latches in the processes and converts them to
d-type latches.
```

proc_init - convert initial block to init attributes

```
yosys> help proc_init
```

```
proc_init [selection]
```

```
This pass extracts the 'init' actions from processes (generated from Verilog
'initial' blocks) and sets the initial value to the 'init' attribute on the
respective wire.
```

proc_memwr - extract memory writes from processes

```
yosys> help proc_memwr
```

```
proc_memwr [selection]
```

```
This pass converts memory writes in processes into $memwr cells.
```

proc_mux - convert decision trees to multiplexers

```
yosys> help proc_mux
```

```
proc_mux [options] [selection]
```

```
This pass converts the decision trees in processes (originating from if-else
and case statements) to trees of multiplexer cells.
```

```
    -ifx            Use Verilog simulation behavior with respect to undef␣
                    ↪values in
                    'case' expressions and 'if' conditions.
```

proc_prune - remove redundant assignments

```
yosys> help proc_prune
```

```
proc_prune [selection]
```

```
This pass identifies assignments in processes that are always overwritten by
a later assignment to the same signal and removes them.
```

proc_rmdead - eliminate dead trees in decision trees

```
yosys> help proc_rmdead
```

```
proc_rmdead [selection]
```

```
This pass identifies unreachable branches in decision trees and removes them.
```

proc_rom - convert switches to ROMs

```
yosys> help proc_rom
```

```
proc_rom [selection]
```

```
This pass converts switches into read-only memories when appropriate.
```

FSM handling

fsm - extract and optimize finite state machines

```
yosys> help fsm
```

```
fsm [options] [selection]
```

```
This pass calls all the other fsm_* passes in a useful order. This performs
FSM extraction and optimization. It also calls opt_clean as needed:

    fsm_detect          unless got option -nodetect
    fsm_extract

    fsm_opt
    opt_clean
    fsm_opt

    fsm_expand          if got option -expand
    opt_clean           if got option -expand
    fsm_opt             if got option -expand

    fsm_recode          unless got option -norecode

    fsm_info

    fsm_export          if got option -export
    fsm_map             unless got option -nomap

Options:
```

```
-expand, -norecode, -export, -nomap    enable/disable passes as indicated above

-fullexpand                            call expand with -full option

-encoding type

-fm_set_fsm_file file

-encfile file                          passed through to fsm_recode pass
```

```
This pass uses a subset of FF types to detect FSMs. Run 'opt -nosdff -nodffe'
before this pass to prepare the design.
```

fsm_detect - finding FSMs in design

```
yosys> help fsm_detect
```

```
fsm_detect [options] [selection]
```

```
This pass detects finite state machines by identifying the state signal.
The state signal is then marked by setting the attribute 'fsm_encoding'
on the state signal to "auto".
```

```
-ignore-self-reset    Mark FSMs even if they are self-resetting
```

```
Existing 'fsm_encoding' attributes are not changed by this pass.

Signals can be protected from being detected by this pass by setting the
'fsm_encoding' attribute to "none".

This pass uses a subset of FF types to detect FSMs. Run 'opt -nosdff -nodffe'
before this pass to prepare the design for fsm_detect.
```

fsm_expand - expand FSM cells by merging logic into it

```
yosys> help fsm_expand
```

```
fsm_expand [-full] [selection]
```

```
The fsm_extract pass is conservative about the cells that belong to a finite
state machine. This pass can be used to merge additional auxiliary gates into
the finite state machine.

By default, fsm_expand is still a bit conservative regarding merging larger
word-wide cells. Call with -full to consider all cells for merging.
```

fsm_export - exporting FSMs to KISS2 files

```
yosys> help fsm_export
```

```
fsm_export [-noauto] [-o filename] [-origenc] [selection]
```

```
This pass creates a KISS2 file for every selected FSM. For FSMs with the
'fsm_export' attribute set, the attribute value is used as filename, otherwise
the module and cell name is used as filename. If the parameter '-o' is given,
the first exported FSM is written to the specified filename. This overwrites
the setting as specified with the 'fsm_export' attribute. All other FSMs are
exported to the default name as mentioned above.
```

```
-noauto           only export FSMs that have the 'fsm_export' attribute␣
                  ↪set

-o filename       filename of the first exported FSM
```

-origenc	use binary state encoding as state names instead of s0, s1, ...

fsm_extract - extracting FSMs in design

```
yosys> help fsm_extract
```

fsm_extract [selection]

```
This pass operates on all signals marked as FSM state signals using the
'fsm_encoding' attribute. It consumes the logic that creates the state signal
and uses the state signal to generate control signal and replaces it with an
FSM cell.

The generated FSM cell still generates the original state signal with its
original encoding. The 'fsm_opt' pass can be used in combination with the
'opt_clean' pass to eliminate this signal.
```

fsm_info - print information on finite state machines

```
yosys> help fsm_info
```

fsm_info [selection]

```
This pass dumps all internal information on FSM cells. It can be useful for
analyzing the synthesis process and is called automatically by the 'fsm'
pass so that this information is included in the synthesis log file.
```

fsm_map - mapping FSMs to basic logic

```
yosys> help fsm_map
```

fsm_map [selection]

```
This pass translates FSM cells to flip-flops and logic.
```

fsm_opt - optimize finite state machines

```
yosys> help fsm_opt
```

fsm_opt [selection]

```
This pass optimizes FSM cells. It detects which output signals are actually
not used and removes them from the FSM. This pass is usually used in
combination with the 'opt_clean' pass (see also 'help fsm').
```

fsm_recode - recoding finite state machines

```
yosys> help fsm_recode
```

```
fsm_recode [options] [selection]
```

```
This pass reassign the state encodings for FSM cells. At the moment only
one-hot encoding and binary encoding is supported.
```

```
-encoding <type>      specify the encoding scheme used for FSMs without the
                      'fsm_encoding' attribute or with the attribute set to␣
                      ↪`auto'.

-fm_set_fsm_file <file>  generate a file containing the mapping from old to new␣
                      ↪FSM encoding
                      in form of Synopsys Formality set_fsm_* commands.

-encfile <file>       write the mappings from old to new FSM encoding to a␣
                      ↪file in the
                      following format:

                      .fsm <module_name> <state_signal>
                      .map <old_bitpattern> <new_bitpattern>
```

Memory handling

memory - translate memories to basic cells

```
yosys> help memory
```

```
memory [-norom] [-nomap] [-nordff] [-nowiden] [-nosat] [-memx] [-no-rw-check]␣
[-bram <bram_rules>] [selection]
```

```
This pass calls all the other memory_* passes in a useful order:

    opt_mem
    opt_mem_priority
    opt_mem_feedback
    memory_bmux2rom                     (skipped if called with -norom)
    memory_dff [-no-rw-check]           (skipped if called with -nordff or -memx)
    opt_clean
    memory_share [-nowiden] [-nosat]
    opt_mem_widen
    memory_memx                         (when called with -memx)
    opt_clean
    memory_collect
    memory_bram -rules <bram_rules>     (when called with -bram)
    memory_map                          (skipped if called with -nomap)

This converts memories to word-wide DFFs and address decoders
or multiport memory blocks if called with the -nomap option.
```

memory_bmux2rom - convert muxes to ROMs

```
yosys> help memory_bmux2rom
```

```
memory_bmux2rom [options] [selection]
```

```
This pass converts $bmux cells with constant A input to ROMs.
```

memory_bram - map memories to block rams

```
yosys> help memory_bram
```

```
memory_bram -rules <rule_file> [selection]
```

```
This pass converts the multi-port $mem memory cells into block ram instances.
The given rules file describes the available resources and how they should be
used.

The rules file contains configuration options, a set of block ram description
and a sequence of match rules.

The option 'attr_icase' configures how attribute values are matched. The value 0
means case-sensitive, 1 means case-insensitive.

A block ram description looks like this:

    bram RAMB1024X32     # name of BRAM cell
      init 1             # set to '1' if BRAM can be initialized
      abits 10           # number of address bits
      dbits 32           # number of data bits
      groups 2           # number of port groups
      ports  1 1         # number of ports in each group
      wrmode 1 0         # set to '1' if this groups is write ports
      enable 4 1         # number of enable bits
      transp 0 2         # transparent (for read ports)
      clocks 1 2         # clock configuration
      clkpol 2 2         # clock polarity configuration
    endbram

For the option 'transp' the value 0 means non-transparent, 1 means transparent
and a value greater than 1 means configurable. All groups with the same
value greater than 1 share the same configuration bit.

For the option 'clocks' the value 0 means non-clocked, and a value greater
than 0 means clocked. All groups with the same value share the same clock
signal.

For the option 'clkpol' the value 0 means negative edge, 1 means positive edge
and a value greater than 1 means configurable. All groups with the same value
greater than 1 share the same configuration bit.

Using the same bram name in different bram blocks will create different variants
of the bram. Verilog configuration parameters for the bram are created as
```

(continues on next page)

(continued from previous page)

```
needed.

It is also possible to create variants by repeating statements in the bram block
and appending '@<label>' to the individual statements.

A match rule looks like this:

    match RAMB1024X32
      max waste 16384    # only use this bram if <= 16k ram bits are unused
      min efficiency 80  # only use this bram if efficiency is at least 80%
    endmatch

It is possible to match against the following values with min/max rules:

    words  ........  number of words in memory in design
    abits  ........  number of address bits on memory in design
    dbits  ........  number of data bits on memory in design
    wports  .......  number of write ports on memory in design
    rports  .......  number of read ports on memory in design
    ports  ........  number of ports on memory in design
    bits  .........  number of bits in memory in design
    dups ..........  number of duplications for more read ports

    awaste  .......  number of unused address slots for this match
    dwaste  .......  number of unused data bits for this match
    bwaste  .......  number of unused bram bits for this match
    waste  ........  total number of unused bram bits (bwaste*dups)
    efficiency  ...  total percentage of used and non-duplicated bits

    acells  .......  number of cells in 'address-direction'
    dcells  .......  number of cells in 'data-direction'
    cells  ........  total number of cells (acells*dcells*dups)

A match containing the command 'attribute' followed by a list of space
separated 'name[=string_value]' values requires that the memory contains any
one of the given attribute name and string values (where specified), or name
and integer 1 value (if no string_value given, since Verilog will interpret
'(* attr *)' as '(* attr=1 *)').
A name prefixed with '!' indicates that the attribute must not exist.

The interface for the created bram instances is derived from the bram
description. Use 'techmap' to convert the created bram instances into
instances of the actual bram cells of your target architecture.

A match containing the command 'or_next_if_better' is only used if it
has a higher efficiency than the next match (and the one after that if
the next also has 'or_next_if_better' set, and so forth).

A match containing the command 'make_transp' will add external circuitry
to simulate 'transparent read', if necessary.

A match containing the command 'make_outreg' will add external flip-flops
```

(continues on next page)

(continued from previous page)

```
to implement synchronous read ports, if necessary.

A match containing the command 'shuffle_enable A' will re-organize
the data bits to accommodate the enable pattern of port A.
```

memory_collect - creating multi-port memory cells

```
yosys> help memory_collect
```

```
memory_collect [selection]
```

```
This pass collects memories and memory ports and creates generic multiport
memory cells.
```

memory_dff - merge input/output DFFs into memory read ports

```
yosys> help memory_dff
```

```
memory_dff [-no-rw-check] [selection]
```

```
This pass detects DFFs at memory read ports and merges them into the memory
port. I.e. it consumes an asynchronous memory port and the flip-flops at its
interface and yields a synchronous memory port.
```

`-no-rw-check`	marks all recognized read ports as "return don't-care value on read/write collision" (same result as setting the no_rw_check attribute on all memories).

memory_libmap - map memories to cells

```
yosys> help memory_libmap
```

```
memory_libmap -lib <library_file> [-D <condition>] [selection]
```

```
This pass takes a description of available RAM cell types and maps
all selected memories to one of them, or leaves them to be mapped to FFs.
```

`-lib <library_file>`	selects a library file containing RAM cell definitions. This option can be passed more than once to select multiple libraries. See passes/memory/memlib.md for description of the library format.
`-D <condition>`	Enables a condition that can be checked within the library file to eg. select between slightly different hardware

(continues on next page)

(continued from previous page)

```
                    ↪variants.
                    This option can be passed any number of times.
```

-logic-cost-rom <num>

-logic-cost-ram <num>

```
                    Sets the cost of a single bit for memory lowered to␣
                    ↪soft logic.
```

-no-auto-distributed

-no-auto-block

-no-auto-huge

```
                    Disables automatic mapping of given kind of RAMs. ␣
                    ↪Manual mapping
                    (using ram_style or other attributes) is still␣
                    ↪supported.
```

-force-params

```
                    Always generate memories with WIDTH and ABITS␣
                    ↪parameters.
```

memory_map - translate multiport memories to basic cells

```
yosys> help memory_map
```

memory_map [options] [selection]

```
This pass converts multiport memory cells as generated by the memory_collect
pass to word-wide DFFs and address decoders.
```

-attr !<name>

```
                    do not map memories that have attribute <name> set.
```

-attr <name>[=<value>]

```
                    for memories that have attribute <name> set, only map␣
                    ↪them if its value
                    is a string <value> (if specified), or an integer 1␣
                    ↪(otherwise). if this
                    option is specified multiple times, map the memory if␣
                    ↪the attribute is
                    to any of the values.
```

-iattr

```
                    for -attr, suppress case sensitivity in matching of
                    ↪<value>.
```

-rom-only

```
                    only perform conversion for ROMs (memories with no␣
                    ↪write ports).
```

-keepdc

```
                    when mapping ROMs, keep x-bits shared across read ports.
```

-formal

```
                    map memories for a global clock based formal␣
                    ↪verification flow.
                    This implies -keepdc, uses $ff cells for ROMs and sets␣
```

(continues on next page)

(continued from previous page)

```
↪hdlname
attributes. It also has limited support for async write␣
↪ports
as generated by clk2fflogic.
```

memory_memx - emulate vlog sim behavior for mem ports

```
yosys> help memory_memx
```

```
memory_memx [selection]
```

```
This pass adds additional circuitry that emulates the Verilog simulation
behavior for out-of-bounds memory reads and writes.
```

memory_narrow - split up wide memory ports

```
yosys> help memory_narrow
```

```
memory_narrow [options] [selection]
```

```
This pass splits up wide memory ports into several narrow ports.
```

memory_nordff - extract read port FFs from memories

```
yosys> help memory_nordff
```

```
memory_nordff [options] [selection]
```

```
This pass extracts FFs from memory read ports. This results in a netlist
similar to what one would get from not calling memory_dff.
```

memory_share - consolidate memory ports

```
yosys> help memory_share
```

```
memory_share [-nosat] [-nowiden] [selection]
```

```
This pass merges share-able memory ports into single memory ports.

The following methods are used to consolidate the number of memory ports:

  - When multiple write ports access the same address then this is converted
    to a single write port with a more complex data and/or enable logic path.

  - When multiple read or write ports access adjacent aligned addresses, they
    are merged to a single wide read or write port.  This transformation can be
    disabled with the "-nowiden" option.

  - When multiple write ports are never accessed at the same time (a SAT
    solver is used to determine this), then the ports are merged into a single
    write port.  This transformation can be disabled with the "-nosat" option.
```

(continues on next page)

(continued from previous page)

```
Note that in addition to the algorithms implemented in this pass, the $memrd
and $memwr cells are also subject to generic resource sharing passes (and other
optimizations) such as "share" and "opt_merge".
```

memory_unpack - unpack multi-port memory cells

```
yosys> help memory_unpack
```

```
memory_unpack [selection]
```

```
This pass converts the multi-port $mem memory cells into individual $memrd and
$memwr cells. It is the counterpart to the memory_collect pass.
```

Optimization passes

clean - remove unused cells and wires

```
yosys> help clean
```

```
clean [options] [selection]
```

```
This is identical to 'opt_clean', but less verbose.

When commands are separated using the ';;' token, this command will be executed
between the commands.

When commands are separated using the ';;;' token, this command will be executed
in -purge mode between the commands.
```

opt - perform simple optimizations

```
yosys> help opt
```

```
opt [options] [selection]
```

```
This pass calls all the other opt_* passes in a useful order. This performs
a series of trivial optimizations and cleanups. This pass executes the other
passes in the following order:

    opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-
↪keepdc]
    opt_merge [-share_all] -nomux

    do
        opt_muxtree
        opt_reduce [-fine] [-full]
        opt_merge [-share_all]
        opt_share  (-full only)
        opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat]  (except when called with -
↪noff)
        opt_hier (-hier only)
```

(continues on next page)

(continued from previous page)

```
        opt_clean [-purge]
        opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-
↪keepdc]
    while <changed design>

When called with -fast the following script is used instead:

    do
        opt_expr [-mux_undef] [-mux_bool] [-undriven] [-noclkinv] [-fine] [-full] [-
↪keepdc]
        opt_merge [-share_all]
        opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat]  (except when called with -
↪noff)
        opt_hier (-hier only)
        opt_clean [-purge]
    while <changed design in opt_dff>

Note: Options in square brackets (such as [-keepdc]) are passed through to
the opt_* commands when given to 'opt'.
```

opt_balance_tree - $and/$or/$xor/$add/$mul cascades to trees

```
yosys> help opt_balance_tree
```

opt_balance_tree [options] [selection]

```
This pass converts cascaded chains of $and/$or/$xor/$add/$mul cells into
trees of cells to improve timing.
```

-arith	only convert arithmetic cells.
-logic	only convert logic cells.

opt_clean - remove unused cells and wires

```
yosys> help opt_clean
```

opt_clean [options] [selection]

```
This pass identifies wires and cells that are unused and removes them. Other
passes often remove cells but leave the wires in the design or reconnect the
wires but leave the old cells in the design. This pass can be used to clean up
after the passes that do the actual work.

This pass only operates on completely selected modules without processes.
```

-purge	also remove internal nets if they have a public name

opt_demorgan - Optimize reductions with DeMorgan equivalents

```
yosys> help opt_demorgan
```

```
opt_demorgan [selection]
```

```
This pass pushes inverters through $reduce_* cells if this will reduce the
overall gate count of the circuit
```

opt_dff - perform DFF optimizations

```
yosys> help opt_dff
```

```
opt_dff [-nodffe] [-nosdff] [-keepdc] [-sat] [selection]
```

```
This pass converts flip-flops to a more suitable type by merging clock enables
and synchronous reset multiplexers, removing unused control inputs, or
potentially removes the flip-flop altogether, converting it to a constant
driver.
```

```
-nodffe          disables dff -> dffe conversion, and other transforms␣
                 ↪recognizing clock
                 enable

-nosdff          disables dff -> sdff conversion, and other transforms␣
                 ↪recognizing sync
                 resets

-simple-dffe     only enables clock enable recognition transform for␣
                 ↪obvious cases

-sat             additionally invoke SAT solver to detect and remove␣
                 ↪flip-flops (with
                 non-constant inputs) that can also be replaced with a␣
                 ↪constant driver

-keepdc          some optimizations change the behavior of the circuit␣
                 ↪with respect to
                 don't-care bits. for example in 'a+0' a single x-bit in
                 ↪'a' will cause
                 all result bits to be set to x. this behavior changes␣
                 ↪when 'a+0' is
                 replaced by 'a'. the -keepdc option disables all such␣
                 ↪optimizations.
```

opt_expr - perform const folding and simple expression rewriting

```
yosys> help opt_expr
```

```
opt_expr [options] [selection]
```

```
This pass performs const folding on internal cell types with constant inputs.
It also performs some simple expression rewriting.
```

-mux_undef	remove 'undef' inputs from $mux, $pmux and $_MUX_ cells
-mux_bool	replace $mux cells with inverters or buffers when possible
-undriven	replace undriven nets with undef (x) constants
-noclkinv	do not optimize clock inverters by changing FF types
-fine	perform fine-grain optimizations
-full	alias for -mux_undef -mux_bool -undriven -fine
-keepdc	some optimizations change the behavior of the circuit with respect to don't-care bits. for example in 'a+0' a single x-bit in 'a' will cause all result bits to be set to x. this behavior changes when 'a+0' is replaced by 'a'. the -keepdc option disables all such optimizations.

opt_ffinv - push inverters through FFs

```
yosys> help opt_ffinv
```

```
opt_ffinv [selection]
```

```
This pass pushes inverters to the other side of a FF when they can be merged
into LUTs on the other side.
```

opt_hier - perform cross-boundary optimization

```
yosys> help opt_hier
```

```
opt_hier [selection]
```

```
This pass considers the design hierarchy and propagates unused signals, constant
signals, and tied-together signals across module boundaries to facilitate
optimization. Only the selected modules are affected.

Note this pass changes port semantics on modules which are not the top.
```

opt_lut - optimize LUT cells

```
yosys> help opt_lut
```

```
opt_lut [options] [selection]
```

```
This pass combines cascaded $lut cells with unused inputs.
```

```
-tech ice40        treat the design as a LUT-mapped circuit for the iCE40␣
                   ↪architecture
                   and preserve connections to SB_CARRY as appropriate

-limit N           only perform the first N combines, then stop. useful␣
                   ↪for debugging.
```

opt_lut_ins - discard unused LUT inputs

```
yosys> help opt_lut_ins
```

```
opt_lut_ins [options] [selection]
```

```
This pass removes unused inputs from LUT cells (that is, inputs that can not
influence the output signal given this LUT's value).  While such LUTs cannot
be directly emitted by ABC, they can be a result of various post-ABC
transformations, such as mapping wide LUTs (not all sub-LUTs will use the
full set of inputs) or optimizations such as xilinx_dffopt.
```

```
-tech <technology>  instead of generic $lut cells, operate on LUT cells␣
                    ↪specific
                    to the given technology.  Valid values are: xilinx,␣
                    ↪lattice,
                    gowin, analogdevices.
```

opt_mem - optimize memories

```
yosys> help opt_mem
```

```
opt_mem [options] [selection]
```

```
This pass performs various optimizations on memories in the design.
```

opt_mem_feedback - convert memory read-to-write port feedback paths to write enables

```
yosys> help opt_mem_feedback
```

```
opt_mem_feedback [selection]
```

```
This pass detects cases where an asynchronous read port is only connected via
a mux tree to a write port with the same address.  When such a connection is
found, it is replaced with a new condition on an enable signal, allowing
for removal of the read port.
```

opt_mem_priority - remove priority relations between write ports that can never collide

```
yosys> help opt_mem_priority
```

```
opt_mem_priority [selection]
```

```
This pass detects cases where one memory write port has priority over another
even though they can never collide with each other -- ie. there can never be
a situation where a given memory bit is written by both ports at the same
time, for example because of always-different addresses, or mutually exclusive
enable signals. In such cases, the priority relation is removed.
```

opt_mem_widen - optimize memories where all ports are wide

```
yosys> help opt_mem_widen
```

```
opt_mem_widen [options] [selection]
```

```
This pass looks for memories where all ports are wide and adjusts the base
memory width up until that stops being the case.
```

opt_merge - consolidate identical cells

```
yosys> help opt_merge
```

```
opt_merge [options] [selection]
```

```
This pass identifies cells with identical type and input signals. Such cells
are then merged to one cell.
```

-nomux	Do not merge MUX cells.
-share_all	Operate on all cell types, not just built-in types.
-keepdc	Do not merge flipflops with don't-care bits in their␣ ↪initial value.

opt_muxtree - eliminate dead trees in multiplexer trees

```
yosys> help opt_muxtree
```

```
opt_muxtree [selection]
```

```
This pass analyzes the control signals for the multiplexer trees in the design
and identifies inputs that can never be active. It then removes this dead
branches from the multiplexer trees.

This pass only operates on completely selected modules without processes.
```

opt_reduce - simplify large MUXes and AND/OR gates

```
yosys> help opt_reduce
```

```
opt_reduce [options] [selection]
```

```
This pass performs two interlinked optimizations:

1. it consolidates trees of large AND gates or OR gates and eliminates
duplicated inputs.

2. it identifies duplicated inputs to MUXes and replaces them with a single
input with the original control signals OR'ed together.
```

-fine	perform fine-grain optimizations
-full	alias for -fine

opt_share - merge mutually exclusive cells of the same type that share an input signal

```
yosys> help opt_share
```

```
opt_share [selection]
```

```
This pass identifies mutually exclusive cells of the same type that:
    (a) share an input signal,
    (b) drive the same $mux, $_MUX_, or $pmux multiplexing cell,

allowing the cell to be merged and the multiplexer to be moved from
multiplexing its output to multiplexing the non-shared input signals.
```

recover_names - Execute a lossy mapping command and recover original netnames

```
yosys> help recover_names
```

```
recover_names [command]
```

```
This pass executes a lossy mapping command and uses a combination of simulation
 to find candidate equivalences and SAT to recover exact original net names.
```

Technology mapping

See also

Technology libraries

Warning

No commands found for group 'passes/techmap'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Design modification

expose - convert internal signals to module ports

```
yosys> help expose
```

```
expose [options] [selection]
```

```
This command exposes all selected internal signals of a module as additional
outputs.
```

`-dff`	only consider wires that are directly driven by␣ ↪register cell.
`-cut`	when exposing a wire, create an input/output pair and␣ ↪cut the internal signal path at that wire.
`-input`	when exposing a wire, create an input port and␣ ↪disconnect the internal driver.
`-shared`	only expose those signals that are shared among the␣ ↪selected modules. this is useful for preparing modules for equivalence␣ ↪checking.
`-evert`	also turn connections to instances of other modules to␣ ↪additional inputs and outputs and remove the module instances.
`-evert-dff`	turn flip-flops to sets of inputs and outputs.
`-sep <separator>`	when creating new wire/port names, the original object␣ ↪name is suffixed with this separator (default: '.') and the port name or␣ ↪a type designator for the exposed signal.

Equivalence checking

equiv_add - add a $equiv cell

```
yosys> help equiv_add
```

```
equiv_add [-try] gold_sig gate_sig
```

```
This command adds an $equiv cell for the specified signals.
```

```
equiv_add [-try] -cell gold_cell gate_cell
```

```
This command adds $equiv cells for the ports of the specified cells.
```

equiv_induct - proving $equiv cells using temporal induction

```
yosys> help equiv_induct
```

```
equiv_induct [options] [selection]
```

```
Uses a version of temporal induction to prove $equiv cells.

Only selected $equiv cells are proven and only selected cells are used to
perform the proof.
```

-undef	enable modelling of undef states
-seq <N>	the max. number of time steps to be considered (default = 4)
-set-assumes	set all assumptions provided via $assume cells
-ignore-unknown-cells	ignore all cells that can not be matched to a SAT model

```
This command is very effective in proving complex sequential circuits, when
the internal state of the circuit quickly propagates to $equiv cells.

However, this command uses a weak definition of 'equivalence': This command
proves that the two circuits will not diverge after they produce equal
outputs (observable points via $equiv) for at least <N> cycles (the <N>
specified via -seq).

Combined with simulation this is very powerful because simulation can give
you confidence that the circuits start out synced for at least <N> cycles
after reset.
```

equiv_make - prepare a circuit for equivalence checking

```
yosys> help equiv_make
```

```
equiv_make [options] gold_module gate_module equiv_module
```

```
This creates a module annotated with $equiv cells from two presumably
equivalent modules. Use commands such as 'equiv_simple' and 'equiv_status'
to work with the created equivalent checking module.
```

-inames	Also match cells and wires with $... names.
-blacklist <file>	Do not match cells or signals that match the names in the file.

-encfile <file>	Match FSM encodings using the description from the file. See 'help fsm_recode' for details.
-make_assert	Check equivalence with $assert cells instead of $equiv. $eqx (===) is used to compare signals.

Note: The circuit created by this command is not a miter (with something like a trigger output), but instead uses $equiv cells to encode the equivalence checking problem. Use 'miter -equiv' if you want to create a miter circuit.

equiv_mark - mark equivalence checking regions

yosys> help **equiv_mark**

equiv_mark [options] [selection]

This command marks the regions in an equivalence checking module. Region 0 is the proven part of the circuit. Regions with higher numbers are connected unproven subcricuits. The integer attribute 'equiv_region' is set on all wires and cells.

equiv_miter - extract miter from equiv circuit

yosys> help **equiv_miter**

equiv_miter [options] miter_module [selection]

This creates a miter module for further analysis of the selected $equiv cells.

-trigger	Create a trigger output
-cmp	Create cmp_* outputs for individual unproven $equiv cells
-assert	Create a $assert cell for each unproven $equiv cell
-undef	Create compare logic that handles undefs correctly

equiv_opt - prove equivalence for optimized circuit

yosys> help **equiv_opt**

equiv_opt [options] [command]

This command uses temporal induction to check circuit equivalence before and after an optimization pass.

-run <from_label>:<to_label>	only run the commands between the labels (see below). an empty from label is synonymous to the start of the command list, and empty to label is synonymous to the end of the command list.
-map <filename>	expand the modules in this file before proving equivalence. this is useful for handling architecture-specific primitives.
-blacklist <file>	Do not match cells or signals that match the names in the file (passed to equiv_make).
-assert	produce an error if the circuits are not equivalent.
-multiclock	run clk2fflogic before equivalence checking.
-async2sync	run async2sync before equivalence checking.
-undef	enable modelling of undef states during equiv_induct.
-nocheck	disable running check before and after the command under test.

```
The following commands are executed by this verification command:

    run_pass:
        hierarchy -auto-top
        design -save preopt
        check -assert    (unless -nocheck)
        [command]
        check -assert    (unless -nocheck)
        design -stash postopt

    prepare:
        design -copy-from preopt  -as gold A:top
        design -copy-from postopt -as gate A:top

    techmap:    (only with -map)
        techmap -wb -D EQUIV -autoproc -map <filename> ...

    prove:
        clk2fflogic    (only with -multiclock)
        async2sync     (only with -async2sync)
        equiv_make -blacklist <filename> ... gold gate equiv
        equiv_induct [-undef] equiv
        equiv_status [-assert] equiv
```

(continues on next page)

(continued from previous page)

```
    restore:
        design -load preopt
```

equiv_purge - purge equivalence checking module

```
yosys> help equiv_purge
```

```
equiv_purge [options] [selection]
```

```
This command removes the proven part of an equivalence checking module, leaving
only the unproven segments in the design. This will also remove and add module
ports as needed.
```

equiv_remove - remove $equiv cells

```
yosys> help equiv_remove
```

```
equiv_remove [options] [selection]
```

```
This command removes the selected $equiv cells. If neither -gold nor -gate is
used then only proven cells are removed.
```

-gold	keep gold circuit
-gate	keep gate circuit

equiv_simple - try proving simple $equiv instances

```
yosys> help equiv_simple
```

```
equiv_simple [options] [selection]
```

```
This command tries to prove $equiv cells using a simple direct SAT approach.
```

-undef	enable modelling of undef states
-seq <N>	the max. number of time steps to be considered (default = 1)
-set-assumes	set all assumptions provided via $assume cells
-ignore-unknown-cells	ignore all cells that can not be matched to a SAT model
-v	verbose output
-short	create shorter input cones that stop at shared nodes. This yields simpler SAT problems but sometimes fails to prove equivalence.

-nogroup	disabling grouping of $equiv cells by output wire

equiv_status - print status of equivalent checking module

```
yosys> help equiv_status
```

```
equiv_status [options] [selection]
```

```
This command prints status information for all selected $equiv cells.
```

-assert	produce an error if any unproven $equiv cell is found

equiv_struct - structural equivalence checking

```
yosys> help equiv_struct
```

```
equiv_struct [options] [selection]
```

```
This command adds additional $equiv cells based on the assumption that the
gold and gate circuit are structurally equivalent. Note that this can introduce
bad $equiv cells in cases where the netlists are not structurally equivalent,
for example when analyzing circuits with cells with commutative inputs. This
command will also de-duplicate gates.
```

-fwd	by default this command performans forward sweeps until␣ ↪nothing can be merged by forwards sweeps, then backward sweeps␣ ↪until forward sweeps are effective again. with this option set only␣ ↪forward sweeps are performed.
-fwonly <cell_type>	add the specified cell type to the list of cell types␣ ↪that are only merged in forward sweeps and never in backward sweeps. ↪$equiv is in this list automatically.
-icells	by default, the internal RTL and gate cell types are␣ ↪ignored. add this option to also process those cell types with this␣ ↪command.
-maxiter <N>	maximum number of iterations to run before aborting

Simulating circuits

Warning

No commands found for group 'passes/sat'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Design status

cd - a shortcut for 'select -module <name>'

```
yosys> help cd
```

cd <modname>

```
This is just a shortcut for 'select -module <modname>'.
```

cd <cellname>

```
When no module with the specified name is found, but there is a cell
with the specified name in the current module, then this is equivalent
to 'cd <celltype>'.
```

cd ..

```
Remove trailing substrings that start with '.' in current module name until
the name of a module in the current design is generated, then switch to that
module. Otherwise clear the current selection.
```

cd

```
This is just a shortcut for 'select -clear'.
```

check - check for obvious problems in the design

```
yosys> help check
```

check [options] [selection]

```
This pass identifies the following problems in the current design:

  - combinatorial loops
  - two or more conflicting drivers for one wire
  - used wires that do not have a driver

Options:
```

-noinit	also check for wires which have the 'init' attribute set
-initdrv	also check for wires that have the 'init' attribute set and are not driven by an FF cell type

-mapped	also check for internal cells that have not been mapped to cells of the target architecture
-allow-tbuf	modify the -mapped behavior to still allow $_TBUF_ cells
-assert	produce a runtime error if any problems are found in the current design
-force-detailed-loop-check	for the detection of combinatorial loops, use a detailed connectivity model for all internal cells for which it is available. This disables falling back to a simpler overapproximating model for those cells for which the detailed model is expected costly.

debug - run command with debug log messages enabled

yosys> help **debug**

debug cmd

```
Execute the specified command with debug log messages enabled

    debug -on
    debug -off

Enable or disable debug log messages globally
```

edgetypes - list all types of edges in selection

yosys> help **edgetypes**

edgetypes [options] [selection]

```
This command lists all unique types of 'edges' found in the selection. An 'edge'
is a 4-tuple of source and sink cell type and port name.
```

exec - execute commands in the operating system shell

yosys> help **exec**

exec [options] -- [command]

```
Execute a command in the operating system shell.  All supplied arguments are
concatenated and passed as a command to popen(3).  Whitespace is not guaranteed
to be preserved, even if quoted.  stdin and stderr are not connected, while
stdout is logged unless the "-q" option is specified.
```

```
-q                      Suppress stdout and stderr from subprocess

-expect-return <int>    Generate an error if popen() does not return specified␣
                        ↪value.
                        May only be specified once; the final specified value␣
                        ↪is controlling
                        if specified multiple times.

-expect-stdout <regex>  Generate an error if the specified regex does not match␣
                        ↪any line
                        in subprocess's stdout.  May be specified multiple␣
                        ↪times.

-not-expect-stdout <regex>  Generate an error if the specified regex matches any␣
                        ↪line
                        in subprocess's stdout.  May be specified multiple␣
                        ↪times.

                        Example: exec -q -expect-return 0 -- echo "bananapie" |␣
                        ↪grep "nana"
```

log - print text and log files

```
yosys> help log
```

```
log [options] string

log [ -push | -pop ]
```

```
Print the given string to the screen and/or the log file. This is useful for TCL
scripts, because the TCL command "puts" only goes to stdout but not to
logfiles.
```

```
-stdout                 Print the output to stdout too. This is useful when all␣
                        ↪Yosys is
                        executed with a script and the -q (quiet operation)␣
                        ↪argument to notify
                        the user.

-stderr                 Print the output to stderr too.

-nolog                  Don't use the internal log() command. Use either -
                        ↪stdout or -stderr,
                        otherwise no output will be generated at all.

-n                      do not append a newline

-header                 log a pass header
```

-push	push a new level on the pass counter
-pop	pop from the pass counter

logger - set logger properties

```
yosys> help logger
```

```
logger [options]
```

This command sets global logger properties, also available using command line options.

-[no]time	enable/disable display of timestamp in log output.
-[no]stderr	enable/disable logging errors to stderr.
-warn regex	print a warning for all log messages matching the regex.
-nowarn regex	if a warning message matches the regex, it is printed as regular message instead.
-werror regex	if a warning message matches the regex, it is printed as error message instead and the tool terminates with a nonzero return code.
-[no]debug	globally enable/disable debug log messages.
-experimental <feature>	print warnings for the specified experimental feature
-expect <type> <regex> <expected_count>	expect a log, warning or error to appear. matched errors will terminate with exit code 0. Types prefix-log, prefix-warning and prefix-error match the entire logged string, including filename if present.
-expect-no-warnings	gives error in case there is at least one warning that is not expected.
-check-expected	verifies that the patterns previously set up by -expect have actually been met, then clears the expected log list. If this is not called manually, the check will happen at yosys exit time instead.

ls - list modules or objects in modules

```
yosys> help ls
```

```
ls [selection]
```

```
When no active module is selected, this prints a list of modules.

When an active module is selected, this prints a list of objects in the module.
```

ltp - print longest topological path

```
yosys> help ltp
```

```
ltp [options] [selection]
```

```
This command prints the longest topological path in the design. (Only considers
paths within a single module, so the design must be flattened.)
```

-noff	automatically exclude FF cell types

plugin - load and list loaded plugins

```
yosys> help plugin
```

```
plugin [options]
```

```
Load and list loaded plugins.
```

-i <plugin_filename>	load (install) the specified plugin.
-a <alias_name>	Register the specified alias name for the loaded plugin
-l	List loaded plugins

portarcs - derive port arcs for propagation delay

```
yosys> help portarcs
```

```
portarcs [options] [selection]
```

```
This command characterizes the combinational content of selected modules and
derives timing arcs going from module inputs to module outputs representing the
propagation delay of the module.
```

-draw	plot the computed delay table to the terminal
-icells	assign unit delay to gates from the internal Yosys cell library

```
-write          write the computed arcs back into the module as
                ↪$specify2 instances
```

portlist - list (top-level) ports

```
yosys> help portlist
```

```
portlist [options] [selection]
```

```
This command lists all module ports found in the selected modules.

If no selection is provided then it lists the ports on the top module.
```

```
-m              print verilog blackbox module definitions instead of␣
                ↪port lists
```

printattrs - print attributes of selected objects

```
yosys> help printattrs
```

```
printattrs [selection]
```

```
Print all attributes of the selected objects.
```

scc - detect strongly connected components (logic loops)

```
yosys> help scc
```

```
scc [options] [selection]
```

```
This command identifies strongly connected components (aka logic loops) in the
design.
```

```
-expect <num>       expect to find exactly <num> SCCs. A different number␣
                    ↪of SCCs will
                    produce an error.

-max_depth <num>    limit to loops not longer than the specified number of␣
                    ↪cells. This
                    can e.g. be useful in identifying small local loops in␣
                    ↪a module that
                    implements one large SCC.

-nofeedback         do not count cells that have their output fed back into␣
                    ↪one of their
                    inputs as single-cell scc.

-all_cell_types     Usually this command only considers internal non-memory␣
                    ↪cells. With
```

(continues on next page)

(continued from previous page)

```
this option set, all cells are considered. For unknown␣
↪cells all ports
are assumed to be bidirectional 'inout' ports.
```

-set_attr <name> <value>

```
set the specified attribute on all cells that are part␣
↪of a logic
loop. the special token {} in the value is replaced␣
↪with a unique
identifier for the logic loop.
```

-select

```
replace the current selection with a selection of all␣
↪cells and wires
that are part of a found logic loop
```

-specify

```
examine specify rules to detect logic loops in whitebox/
↪blackbox cells
```

scratchpad - get/set values in the scratchpad

yosys> help **scratchpad**

scratchpad [options]

```
This pass allows reading and modifying values from the scratchpad of the current
design. Options:
```

-get <identifier>

```
print the value saved in the scratchpad under the given␣
↪identifier.
```

-set <identifier> <value>

```
set the given value in the scratchpad under the given␣
↪identifier.
```

-unset <identifier>

```
remove the entry for the given identifier from the␣
↪scratchpad.
```

-copy <identifier_from> <identifier_to>

```
copy the value for the first identifier to the second␣
↪identifier.
```

-assert <identifier> <value>

```
assert that the entry for the given identifier is set␣
↪to the given
value.
```

-assert-set <identifier>

```
assert that the entry for the given identifier exists.
```

-assert-unset <identifier>

```
assert that the entry for the given identifier does not␣
↪exist.
```

```
The identifier may not contain whitespace. By convention, it is usually prefixed
by the name of the pass that uses it, e.g. 'opt.did_something'. If the value
contains whitespace, it must be enclosed in double quotes.
```

select - modify and view the list of selected objects

```
yosys> help select
```

```
select [ -add | -del | -set <name> ] {-read <filename> | <selection>}

select [ -unset <name> ]

select [ <assert_option> ] {-read <filename> | <selection>}

select [ -list | -list-mod | -write <filename> | -count | -clear ]

select -module <modname>
```

```
Most commands use the list of currently selected objects to determine which part
of the design to operate on. This command can be used to modify and view this
list of selected objects.

Note that many commands support an optional [selection] argument that can be
used to override the global selection for the command. The syntax of this
optional argument is identical to the syntax of the <selection> argument
described here.
```

`-add, -del`	add or remove the given objects to the current selection. without this options the current selection is replaced.
`-set <name>`	do not modify the current selection. instead save the new selection under the given name (see @<name> below). to save the current selection, use "select -set <name> %"
`-unset <name>`	do not modify the current selection. instead remove a previously saved selection under the given name (see @<name> below).
`-assert-none`	do not modify the current selection. instead assert that the given selection is empty. i.e. produce an error if any object or module matching the selection is found.
`-assert-any`	do not modify the current selection. instead assert that the given selection is non-empty. i.e. produce an error if no object or module matching the selection is found.

-assert-mod-count N	do not modify the current selection. instead assert that the given selection contains exactly N modules (partially or fully selected).
-assert-count N	do not modify the current selection. instead assert that the given selection contains exactly N objects.
-assert-max N	do not modify the current selection. instead assert that the given selection contains less than or exactly N objects.
-assert-min N	do not modify the current selection. instead assert that the given selection contains at least N objects.
-list	list all objects in the current selection
-write <filename>	like -list but write the output to the specified file
-read <filename>	read the specified file (written by -write)
-count	count all objects in the current selection
-clear	clear the current selection. this effectively selects the whole design. it also resets the selected module (see -module). use the command 'select *' to select everything but stay in the current module.
-none	create an empty selection. the current module is unchanged.
-module <modname>	limit the current scope to the specified module. the difference between this and simply selecting the module is that all object names are interpreted relative to this module after this command until the selection is cleared again.

```
When this command is called without an argument, the current selection
is displayed in a compact form (i.e. only the module name when a whole module
is selected).

The <selection> argument itself is a series of commands for a simple stack
machine. Each element on the stack represents a set of selected objects.
```

(continues on next page)

(continued from previous page)

```
After this commands have been executed, the union of all remaining sets
on the stack is computed and used as selection for the command.

Pushing (selecting) object when not in -module mode:

    <mod_pattern>
        select the specified module(s)

    <mod_pattern>/<obj_pattern>
        select the specified object(s) from the module(s)

Pushing (selecting) object when in -module mode:

    <obj_pattern>
        select the specified object(s) from the current module

By default, patterns will not match black/white-box modules or their
contents. To include such objects, prefix the pattern with '='.

A <mod_pattern> can be a module name, wildcard expression (*, ?, [..])
matching module names, or one of the following:

    A:<pattern>, A:<pattern>=<pattern>
        all modules with an attribute matching the given pattern
        in addition to = also <, <=, >=, and > are supported

    N:<pattern>
        all modules with a name matching the given pattern
        (i.e. 'N:' is optional as it is the default matching rule)

An <obj_pattern> can be an object name, wildcard expression, or one of
the following:

    w:<pattern>
        all wires with a name matching the given wildcard pattern

    i:<pattern>, o:<pattern>, x:<pattern>
        all inputs (i:), outputs (o:) or any ports (x:) with matching names

    s:<size>, s:<min>:<max>
        all wires with a matching width

    m:<pattern>
        all memories with a name matching the given pattern

    c:<pattern>
        all cells with a name matching the given pattern

    t:<pattern>
        all cells with a type matching the given pattern

    t:@<name>
```

(continues on next page)

(continued from previous page)

```
        all cells with a type matching a module in the saved selection <name>

    p:<pattern>
        all processes with a name matching the given pattern

    a:<pattern>
        all objects with an attribute name matching the given pattern

    a:<pattern>=<pattern>
        all objects with a matching attribute name-value-pair.
        in addition to = also <, <=, >=, and > are supported

    r:<pattern>, r:<pattern>=<pattern>
        cells with matching parameters. also with <, <=, >= and >.

    n:<pattern>
        all objects with a name matching the given pattern
        (i.e. 'n:' is optional as it is the default matching rule)

    @<name>
        push the selection saved prior with 'select -set <name> ...'

The following actions can be performed on the top sets on the stack:

    %
        push a copy of the current selection to the stack

    %%
        replace the stack with a union of all elements on it

    %n
        replace top set with its invert

    %u
        replace the two top sets on the stack with their union

    %i
        replace the two top sets on the stack with their intersection

    %d
        pop the top set from the stack and subtract it from the new top

    %D
        like %d but swap the roles of two top sets on the stack

    %c
        create a copy of the top set from the stack and push it

    %x[<num1>|*][.<num2>][:<rule>[:<rule>..]]
        expand top set <num1> num times according to the specified rules.
        (i.e. select all cells connected to selected wires and select all
        wires connected to selected cells) The rules specify which cell
```

(continues on next page)

(continued from previous page)

```
        ports to use for this. the syntax for a rule is a '-' for exclusion
        and a '+' for inclusion, followed by an optional comma separated
        list of cell types followed by an optional comma separated list of
        cell ports in square brackets. a rule can also be just a cell or wire
        name that limits the expansion (is included but does not go beyond).
        select at most <num2> objects. a warning message is printed when this
        limit is reached. When '*' is used instead of <num1> then the process
        is repeated until no further object are selected.

    %ci[<num1>|*][.<num2>][:<rule>[:<rule>..]]
    %co[<num1>|*][.<num2>][:<rule>[:<rule>..]]
        similar to %x, but only select input (%ci) or output cones (%co)

    %xe[...] %cie[...] %coe
        like %x, %ci, and %co but only consider combinatorial cells

    %a
        expand top set by selecting all wires that are (at least in part)
        aliases for selected wires.

    %s
        expand top set by adding all modules that implement cells in selected
        modules

    %m
        expand top set by selecting all modules that contain selected objects

    %M
        select modules that implement selected cells

    %C
        select cells that implement selected modules

    %R[<num>]
        select <num> random objects from top selection (default 1)

Example: the following command selects all wires that are connected to a
'GATE' input of a 'SWITCH' cell:

    select */t:SWITCH %x:+[GATE] */t:SWITCH %d
```

setenv - set an environment variable

yosys> help **setenv**

setenv name value

```
Set the given environment variable on the current process. Values containing
whitespace must be passed in double quotes (").
```

show - generate schematics using graphviz

```
yosys> help show
```

```
show [options] [selection]
```

```
Create a graphviz DOT file for the selected part of the design and compile it
to a graphics file (usually SVG or PostScript).
```

-viewer <viewer>

```
Run the specified command with the graphics file as parameter.
On Windows, this pauses yosys until the viewer exits.
Use "-viewer none" to not run any command.
```

-format <format>

```
Generate a graphics file in the specified format. Use 'dot' to just
generate a .dot file, or other <format> strings such as 'svg' or 'ps'
to generate files in other formats (this calls the 'dot' command).
```

-lib <verilog_or_liberty_file>

```
Use the specified library file for determining whether cell ports are
inputs or outputs. This option can be used multiple times to specify
more than one library.

note: in most cases it is better to load the library before calling
show with 'read_verilog -lib <filename>'. it is also possible to
load liberty files with 'read_liberty -lib <filename>'.
```

-prefix <prefix>

```
generate <prefix>.* instead of ~/.yosys_show.*
```

-color <color> <object>

```
assign the specified color to the specified object. The object can be
a single selection wildcard expressions or a saved set of objects in
the @<name> syntax (see "help select" for details).
```

-label <text> <object>

```
assign the specified label text to the specified object. The object can
be a single selection wildcard expressions or a saved set of objects in
the @<name> syntax (see "help select" for details).
```

-colors <seed>

```
Randomly assign colors to the wires. The integer argument is the seed
for the random number generator. Change the seed value if the colored
```

(continues on next page)

(continued from previous page)

```
                        graph still is ambiguous. A seed of zero deactivates␣
                        ↪the coloring.

-colorattr <attribute_name>
                        use the specified attribute to assign colors. A unique␣
                        ↪color is
                        assigned to each unique value of this attribute.

-wireshape <graphviz_shape>
                        use the specified shape for wire nodes. E.g. plaintext.

-width                  annotate buses with a label indicating the width of the␣
                        ↪bus.

-signed                 mark ports (A, B) that are declared as signed (using␣
                        ↪the [AB]_SIGNED
                        cell parameter) with an asterisk next to the port name.

-stretch                stretch the graph so all inputs are on the left side␣
                        ↪and all outputs
                        (including inout ports) are on the right side.

-pause                  wait for the user to press enter to before returning

-enum                   enumerate objects with internal ($-prefixed) names

-long                   do not abbreviate objects with internal ($-prefixed)␣
                        ↪names

-notitle                do not add the module name as graph title to the dot␣
                        ↪file

-nobg                   don't run viewer in the background, IE wait for the␣
                        ↪viewer tool to
                        exit before returning

-href                   adds href attribute to all items representing cells and␣
                        ↪wires, using
                        src attribute of origin
```

```
When no <format> is specified, 'dot' is used. When no <format> and <viewer> is
specified, 'xdot' is used to display the schematic (POSIX systems only).

The generated output files are '~/.yosys_show.dot' and '~/.yosys_show.<format>',
unless another prefix is specified using -prefix <prefix>.

Yosys on Windows and YosysJS use different defaults: The output is written
to 'show.dot' in the current directory and new viewer is launched each time
the 'show' command is executed.
```

sta - perform static timing analysis

```
yosys> help sta
```

```
sta [options] [selection]
```

```
This command performs static timing analysis on the design. (Only considers
paths within a single module, so the design must be flattened.)
```

stat - print some statistics

```
yosys> help stat
```

```
stat [options] [selection]
```

```
Print some statistics (number of objects) on the selected portion of the
design.
Extracts the area of cells from a liberty file, if provided.
```

-top <module>	print design hierarchy with this module as top. if the design is fully selected and a module has the 'top' attribute set, this module is used default value for this option.
-liberty <liberty_file>	use cell area information from the provided liberty file
-tech <technology>	print area estimate for the specified technology. Currently supported values for <technology>: xilinx, analogdevices, cmos
-width	annotate internal cell types with their word width. e.g. $add_8 for an 8 bit wide $add cell.
-json	output the statistics in a machine-readable JSON format. this is output to the console; use "tee" to output to a file.
-hierarchy	print hierarchical statistics, i.e. The area and number of cells include submodules. this changes the format of the json output.

tee - redirect command output to file

```
yosys> help tee
```

```
tee [-q] [-o logfile|-a logfile] cmd
```

```
Execute the specified command, optionally writing the commands output to the
specified logfile(s).
```

-q	Do not print output to the normal destination (console and/or log file).
-o logfile	Write output to this file, truncate if exists.
-a logfile	Write output to this file, append if exists.
-s scratchpad	Write output to this scratchpad value, truncate if it exists. +INT, -INT Add/subtract INT from the -v setting for this command.

torder - print cells in topological order

```
yosys> help torder
```

```
torder [options] [selection]
```

```
This command prints the selected cells in topological order.
```

-stop <cell_type> <cell_port>	stop at the specified cell port in topological sorting
-noautostop	by default Q outputs of internal FF cells and memory read port outputs are not used in topological sorting. this option deactivates that.

trace - redirect command output to file

```
yosys> help trace
```

```
trace cmd
```

```
Execute the specified command, logging all changes the command performs on
the design in real time.
```

viz - visualize data flow graph

```
yosys> help viz
```

```
viz [options] [selection]
```

```
Create a graphviz DOT file for the selected part of the design, showing the
relationships between the selected wires, and compile it to a graphics
file (usually SVG or PostScript).
```

```
-viewer <viewer>        Run the specified command with the graphics file as␣
                        ↪parameter.
                        On Windows, this pauses yosys until the viewer exits.

-format <format>        Generate a graphics file in the specified format. Use
                        ↪'dot' to just
                        generate a .dot file, or other <format> strings such as
                        ↪'svg' or 'ps'
                        to generate files in other formats (this calls the 'dot
                        ↪' command).

-prefix <prefix>        generate <prefix>.* instead of ~/.yosys_viz.*

-pause                  wait for the user to press enter to before returning

-nobg                   don't run viewer in the background, IE wait for the␣
                        ↪viewer tool to
                        exit before returning

-set-vg-attr            set their group index as 'vg' attribute on cells and␣
                        ↪wires

-g <selection>          manually define a group of terminal signals. this group␣
                        ↪is not being
                        merged with other terminal groups.

-u <selection>          manually define a unique group for each wire in the␣
                        ↪selection.

-x <selection>          manually exclude wires from being considered. (usually␣
                        ↪this is
                        used for global signals, such as reset.)

-s <selection>          like -g, but mark group as 'special', changing the␣
                        ↪algorithm to
                        preserve as much info about this groups connectivity as␣
                        ↪possible.

-G <selection_expr> .

-U <selection_expr> .

-X <selection_expr> .

-S <selection_expr>     like -u, -g, -x, and -s, but parse all arguments up to␣
                        ↪a terminating .
                        as a single select expression. (see 'help select' for␣
                        ↪details)

-0, -1, -2, -3, -4, -5, -6, -7, -8, -9  each level corresponds to an␣
                        ↪incresingly more
```

(continues on next page)

(continued from previous page)

```
aggressive sequence of strategies for merging nodes of␣
↪the data flow
graph. (default: 9)
```

```
When no <format> is specified, 'dot' is used. When no <format> and <viewer> is
specified, 'xdot' is used to display the schematic (POSIX systems only).

The generated output files are '~/.yosys_viz.dot' and '~/.yosys_viz.<format>',
unless another prefix is specified using -prefix <prefix>.

Yosys on Windows and YosysJS use different defaults: The output is written
to 'show.dot' in the current directory and new viewer is launched each time
the 'show' command is executed.
```

write_file - write a text to a file

```
yosys> help write_file
```

```
write_file [options] output_file [input_file]
```

```
Write the text from the input file to the output file.
```

```
-a                  Append to output file (instead of overwriting)
```

```
Inside a script the input file can also can a here-document:

    write_file hello.txt <<EOT
    Hello World!
    EOT
```

10.1.7 Technology libraries

Listed in alphabetical order.

Generic

Warning

No commands found for group 'techlibs/common'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Achronix

Warning

No commands found for group 'techlibs/achronix'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Anlogic

Warning

No commands found for group 'techlibs/anlogic'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

CoolRunner-II

Warning

No commands found for group 'techlibs/coolrunner2'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

eASIC

Warning

No commands found for group 'techlibs/easic'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

FABulous

Warning

No commands found for group 'techlibs/fabulous'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Gatemate

Warning

No commands found for group 'techlibs/gatemate'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Gowin

Warning

No commands found for group 'techlibs/gowin'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

GreenPAK4

rmports - remove module ports with no connections

```
yosys> help rmports
```

```
rmports [selection]
```

```
This pass identifies ports in the selected modules which are not used or
driven and removes them.
```

iCE40

Warning

No commands found for group 'techlibs/ice40'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Intel (MAX10, Cyclone IV)

Warning

No commands found for group 'techlibs/intel'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Intel ALM (Cyclone V, Arria V, Cyclone 10 GX)

Warning

No commands found for group 'techlibs/intel_alm'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Lattice

Warning

No commands found for group 'techlibs/lattice'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Microchip

Warning

No commands found for group 'techlibs/microchip'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Microchip - SmartFusion2/IGLOO2

Warning

No commands found for group 'techlibs/sf2'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

NanoXplore

Warning

No commands found for group 'techlibs/nanoxplore'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

QuickLogic

Warning

No commands found for group 'techlibs/quicklogic'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

Xilinx

Warning

No commands found for group 'techlibs/xilinx'

Documentation may have been built without `source_location` support. Try check /cmd/index_other.

10.1.8 Internal commands for developers

example_dt - drivertools example

```
yosys> help example_dt
```

> **Warning**
>
> This command is intended for internal developer use only

```
TODO: add help message
```

internal_stats - print internal statistics

```
yosys> help internal_stats
```

> **Warning**
>
> This command is experimental

> **Warning**
>
> This command is intended for internal developer use only

```
Print internal statistics for developers (experimental)
```

linux_perf - turn linux perf recording off or on

```
yosys> help linux_perf
```

> **Warning**
>
> This command is intended for internal developer use only

```
linux_perf [on|off]
```

This pass turns Linux 'perf' profiling on or off, when it has been configured to use control FIFOs.YOSYS_PERF_CTL and YOSYS_PERF_ACK must point to Linux perf control FIFOs.

Example shell command line:

```
mkfifo /tmp/perf.fifo /tmp/perf-ack.fifo
YOSYS_PERF_CTL=/tmp/perf.fifo YOSYS_PERF_ACK=/tmp/perf-ack.fifo \
  perf record --latency --delay=-1 \
  --control=fifo:/tmp/perf.fifo,/tmp/perf-ack.fifo --call-graph=dwarf ./yosys \
  -dt -p "read_rtlil design.rtlil; linux_perf on; opt_clean; linux_perf off"
```

test_abcloop - automatically test handling of loops in abc command

```
yosys> help test_abcloop
```

> **Warning**
>
> This command is intended for internal developer use only

test_abcloop [options]

```
Test handling of logic loops in ABC.
```

-n {integer}	create this number of circuits and test them (default = 100).
-s {positive_integer}	use this value as rng seed value (default = unix time).

test_autotb - generate simple test benches

```
yosys> help test_autotb
```

> **Warning**
>
> This command is intended for internal developer use only

test_autotb [options] [filename]

```
Automatically create primitive Verilog test benches for all modules in the
design. The generated testbenches toggle the input pins of the module in
a semi-random manner and dumps the resulting output signals.

This can be used to check the synthesis results for simple circuits by
comparing the testbench output for the input files and the synthesis results.

The backend automatically detects clock signals. Additionally a signal can
be forced to be interpreted as clock signal by setting the attribute
'gentb_clock' on the signal.

The attribute 'gentb_constant' can be used to force a signal to a constant
value after initialization. This can e.g. be used to force a reset signal
low in order to explore more inner states in a state machine.

The attribute 'gentb_skip' can be attached to modules to suppress testbench
generation.
```

-n <int>	number of iterations the test bench should run (default = 1000)
-seed <int>	seed used for pseudo-random number generation (default = 0). a value of 0 will cause an arbitrary seed to be chosen, based on the current system time.

test_cell - automatically test the implementation of a cell type

```
yosys> help test_cell
```

> **Warning**
>
> This command is intended for internal developer use only

```
test_cell [options] {cell-types}
```

```
Tests the internal implementation of the given cell type (for example '$add')
by comparing SAT solver, EVAL and TECHMAP implementations of the cell types..

Run with 'all' instead of a cell type to run the test on all supported
cell types. Use for example 'all /$add' for all cell types except $add.
```

-n {integer}	create this number of cell instances and test them (default = 100).
-s {positive_integer}	use this value as rng seed value (default = unix time).
-f {rtlil_file}	don't generate circuits. instead load the specified RTLIL file.
-w {filename_prefix}	don't test anything. just generate the circuits and write them to RTLIL files with the specified prefix
-map {filename}	pass this option to techmap.
-simlib	use "techmap -D SIMLIB_NOCHECKS -map +/simlib.v -max_iter 2 -autoproc"
-aigmap	instead of calling "techmap", call "aigmap"
-muxdiv	when creating test benches with dividers, create an additional mux to mask out the division-by-zero case
-script {script_file}	instead of calling "techmap", call "script {script_file}".
-const	set some input bits to random constant values
-nosat	do not check SAT model or run SAT equivalence checking
-noeval	do not check const-eval models

-noopt	do not opt tecchmapped design
-edges	test cell edges db creator against sat-based implementation
-v	print additional debug information to the console
-vlog {filename}	create a Verilog test bench to test simlib and write_verilog
-bloat {factor}	increase cell size limits b{factor} times where possible
-check_cost	check if the estimated cell cost is a valid upper bound for the techmapped cell count

test_generic - test the generic compute graph

yosys> help test_generic

> **Warning**
>
> This command is intended for internal developer use only

TODO: add help message

test_pmgen - test pass for pmgen

yosys> help test_pmgen

> **Warning**
>
> This command is intended for internal developer use only

test_pmgen -reduce_chain [options] [selection]

Demo for recursive pmgen patterns. Map chains of AND/OR/XOR to $reduce_*.

test_pmgen -reduce_tree [options] [selection]

Demo for recursive pmgen patterns. Map trees of AND/OR/XOR to $reduce_*.

test_pmgen -eqpmux [options] [selection]

Demo for recursive pmgen patterns. Optimize EQ/NE/PMUX circuits.

test_pmgen -generate [options] <pattern_name>

```
Create modules that match the specified pattern.
```

10.1.9 Writing command help

- use *chformal* as an example
- generated help content below

chformal - change formal constraints of the design

```
yosys> help chformal
```

chformal [types] [mode] [options] [selection]

Make changes to the formal constraints of the design. The [types] options the type of constraint to operate on. If none of the following options are given, the command will operate on all constraint types:

-assert	*$assert* cells, representing `assert(...)` constraints
-assume	*$assume* cells, representing `assume(...)` constraints
-live	*$live* cells, representing `assert(s_eventually ...)`
-fair	*$fair* cells, representing `assume(s_eventually ...)`
-cover	*$cover* cells, representing `cover()` statements

Additionally chformal will operate on *$check* cells corresponding to the selected constraint types.

Exactly one of the following modes must be specified:

-remove	remove the cells and thus constraints from the design
-early	bypass FFs that only delay the activation of a constraint. When inputs of the bypassed FFs do not remain stable between clock edges, this may result in unexpected behavior.
-delay <N>	delay activation of the constraint by <N> clock cycles
-skip <N>	ignore activation of the constraint in the first <N> clock cycles
-coverenable	add cover statements for the enable signals of the constraints
-assert2assume	
-assert2cover	
-assume2assert	
-live2fair	
-fair2live	change the roles of cells as indicated. these options can be combined
-lower	convert each $check cell into an $assert, $assume, $live, $fair or $cover cell. If the $check cell contains a message, also produce a $print cell.

The `formatted_help()` method

- `PrettyHelp::get_current()`
- `PrettyHelp::set_group()`
  - used with `.. autocmdgroup:: <group>`

  - can assign group and return false
  - if no group is set, will try to use `source_location` and assign group from path to source file
- return value
  - true means help content added to current `PrettyHelp`
  - false to use `Pass::help()`
- adding content
  - help content is a list of `ContentListing` nodes, each one having a type, body, and its own list of children `ContentListing`s
  - `PrettyHelp::get_root()` returns the root `ContentListing` (`type="root"`)
  - `ContentListing::{usage, option, codeblock, paragraph}` each add a `ContentListing` to the current node, with type the same as the method
    - the first argument is the body of the new node
    - `usage` shows how to call the command (i.e. its "signature")
    - `paragraph` content is formatted as a paragraph of text with line breaks added automatically
    - `codeblock` content is displayed verbatim, use line breaks as desired; takes an optional `language` argument for assigning the language in RST output for code syntax highlighting (use `yoscrypt` for yosys script syntax highlighting)
    - `option` lists a single option for the command, usually starting with a dash (`-`); takes an optional second argument which adds a paragraph node as a means of description
  - `ContentListing::open_usage` creates and returns a new usage node, can be used to e.g. add text/options specific to a given usage of the command
  - `ContentListing::open_option` creates and returns a new option node, can be used to e.g. add multiple paragraphs to an option's description
  - paragraphs are treated as raw RST, allowing for inline formatting and references as if it were written in the RST file itself

Listing 10.1: `ChformalPass::formatted_help()` from `passes/cmds/chformal.cc`

```
	bool formatted_help() override {
		auto *help = PrettyHelp::get_current();
		help->set_group("formal");

		auto content_root = help->get_root();

		content_root->usage("chformal [types] [mode] [options] [selection]");
		content_root->paragraph(
			"Make changes to the formal constraints of the design. The [types] options "
			"the type of constraint to operate on. If none of the following options are "
			"given, the command will operate on all constraint types:"
		);

		content_root->option("-assert", "`$assert` cells, representing ``assert(.
```

(continues on next page)

(continued from previous page)

```
↪..)`` constraints");
                content_root->option("-assume", "`$assume` cells, representing ``assume(.
↪..)`` constraints");
                content_root->option("-live", "`$live` cells, representing ``assert(s_
↪eventually ...)``");
                content_root->option("-fair", "`$fair` cells, representing ``assume(s_
↪eventually ...)``");
                content_root->option("-cover", "`$cover` cells, representing ``cover()``␣
↪statements");
                content_root->paragraph(
                        "Additionally chformal will operate on `$check` cells␣
↪corresponding to the "
                        "selected constraint types."
                );

                content_root->paragraph("Exactly one of the following modes must be␣
↪specified:");

                content_root->option("-remove", "remove the cells and thus constraints␣
↪from the design");
                content_root->option("-early",
                        "bypass FFs that only delay the activation of a constraint. When␣
↪inputs "
                        "of the bypassed FFs do not remain stable between clock edges,␣
↪this may "
                        "result in unexpected behavior."
                );
                content_root->option("-delay <N>", "delay activation of the constraint␣
↪by <N> clock cycles");
                content_root->option("-skip <N>", "ignore activation of the constraint␣
↪in the first <N> clock cycles");
                auto cover_option = content_root->open_option("-coverenable");
                cover_option->paragraph(
                        "add cover statements for the enable signals of the constraints"
                );
#ifdef YOSYS_ENABLE_VERIFIC
                cover_option->paragraph(
                        "Note: For the Verific frontend it is currently not guaranteed␣
↪that a "
                        "reachable SVA statement corresponds to an active enable signal."
                );
#endif
                content_root->option("-assert2assume");
                content_root->option("-assert2cover");
                content_root->option("-assume2assert");
                content_root->option("-live2fair");
                content_root->option("-fair2live", "change the roles of cells as␣
↪indicated. these options can be combined");
                content_root->option("-lower",
                        "convert each $check cell into an $assert, $assume, $live, $fair␣
↪or "
                        "$cover cell. If the $check cell contains a message, also␣
```

(continues on next page)

(continued from previous page)

```
↪produce a "
                        "$print cell."
                );
                return true;
        }
```

Dumping command help to json

- `help -dump-cmds-json cmds.json`
  - generates a `ContentListing` for each command registered in Yosys
  - tries to parse unformatted `Pass::help()` output if `Pass::formatted_help()` is unimplemented or returns false
    - if a line starts with four spaces followed by the name of the command then a space, it is parsed as a signature (usage node)
    - if a line is indented and starts with a dash (`-`), it is parsed as an option
    - anything else is parsed as a codeblock and added to either the root node or the current option depending on the indentation
  - dictionary of command name to `ContentListing`
    - uses `ContentListing::to_json()` recursively for each node in root
    - root node used for source location of class definition
    - includes flags set during pass constructor (e.g. `experimental_flag` set by `Pass::experimental()`)
    - also title (`short_help` argument in `Pass::Pass`), group, and class name
  - dictionary of group name to list of commands in that group
- used by sphinx autodoc to generate help content

Listing 10.2: *chformal* in generated `cmds.json`

```
    "chformal": {
      "title": "change formal constraints of the design",
      "content": [
        {"body": "chformal [types] [mode] [options] [selection]", "content": [], "options
↪": {}, "type": "usage"},
        {"body": "Make changes to the formal constraints of the design. The [types]␣
↪options the type of constraint to operate on. If none of the following options are␣
↪given, the command will operate on all constraint types:", "content": [], "options": {}
↪, "type": "text"},
        {"body": "-assert", "content": [{"body": "`$assert` cells, representing␣
↪``assert(...)`` constraints", "content": [], "options": {}, "type": "text"}], "options
↪": {}, "type": "option"},
        {"body": "-assume", "content": [{"body": "`$assume` cells, representing␣
↪``assume(...)`` constraints", "content": [], "options": {}, "type": "text"}], "options
↪": {}, "type": "option"},
        {"body": "-live", "content": [{"body": "`$live` cells, representing ``assert(s_
↪eventually ...)``", "content": [], "options": {}, "type": "text"}], "options": {},
↪"type": "option"},
```

(continues on next page)

(continued from previous page)


```
        {"body": "-fair", "content": [{"body": "`$fair` cells, representing ``assume(s_
↪eventually ...)``", "content": [], "options": {}, "type": "text"}], "options": {},
↪"type": "option"},
        {"body": "-cover", "content": [{"body": "`$cover` cells, representing␣
↪``cover()`` statements", "content": [], "options": {}, "type": "text"}], "options": {},
↪ "type": "option"},
        {"body": "Additionally chformal will operate on `$check` cells corresponding to␣
↪the selected constraint types.", "content": [], "options": {}, "type": "text"},
        {"body": "Exactly one of the following modes must be specified:", "content": [],
↪"options": {}, "type": "text"},
        {"body": "-remove", "content": [{"body": "remove the cells and thus constraints␣
↪from the design", "content": [], "options": {}, "type": "text"}], "options": {}, "type
↪": "option"},
        {"body": "-early", "content": [{"body": "bypass FFs that only delay the␣
↪activation of a constraint. When inputs of the bypassed FFs do not remain stable␣
↪between clock edges, this may result in unexpected behavior.", "content": [], "options
↪": {}, "type": "text"}], "options": {}, "type": "option"},
        {"body": "-delay <N>", "content": [{"body": "delay activation of the constraint␣
↪by <N> clock cycles", "content": [], "options": {}, "type": "text"}], "options": {},
↪"type": "option"},
        {"body": "-skip <N>", "content": [{"body": "ignore activation of the constraint␣
↪in the first <N> clock cycles", "content": [], "options": {}, "type": "text"}],
↪"options": {}, "type": "option"},
        {"body": "-coverenable", "content": [{"body": "add cover statements for the␣
↪enable signals of the constraints", "content": [], "options": {}, "type": "text"}],
↪"options": {}, "type": "option"},
        {"body": "-assert2assume", "content": [], "options": {}, "type": "option"},
        {"body": "-assert2cover", "content": [], "options": {}, "type": "option"},
        {"body": "-assume2assert", "content": [], "options": {}, "type": "option"},
        {"body": "-live2fair", "content": [], "options": {}, "type": "option"},
        {"body": "-fair2live", "content": [{"body": "change the roles of cells as␣
↪indicated. these options can be combined", "content": [], "options": {}, "type": "text
↪"}], "options": {}, "type": "option"},
        {"body": "-lower", "content": [{"body": "convert each $check cell into an
↪$assert, $assume, $live, $fair or $cover cell. If the $check cell contains a message,␣
↪also produce a $print cell.", "content": [], "options": {}, "type": "text"}], "options
↪": {}, "type": "option"}
      ],
      "group": "formal",
      "source_file": "unknown",
      "source_line": 0,
      "source_func": "unknown",
      "experimental_flag": false,
      "internal_flag": false
    },
```


Note

Synthesis command scripts are special cased

If the final block of help output starts with the string `"The following commands are executed by this synthesis command:n"`, then the rest of the code block is formatted as `yoscrypt` (e.g.

synth_ice40). The caveat here is that if the `script()` calls `run()` on any commands *prior* to the first `check_label` then the auto detection will break and revert to unformatted code (e.g. `synth_fabulous`).

Command line rendering

- if `Pass::formatted_help()` returns true, will call `PrettyHelp::log_help()`
  - traverse over the children of the root node and render as plain text
  - effectively the reverse of converting unformatted `Pass::help()` text
  - lines are broken at 80 characters while maintaining indentation (controlled by `MAX_LINE_LEN` in `kernel/log_help.cc`)
  - each line is broken into words separated by spaces, if a given word starts and ends with backticks they will be stripped
- if it returns false it will call `Pass::help()` which should call `log()` directly to print and format help text
  - if `Pass::help()` is not overridden then a default message about missing help will be displayed

```
-- Running command `help chformal' --

    chformal [types] [mode] [options] [selection]

Make changes to the formal constraints of the design. The [types] options the
type of constraint to operate on. If none of the following options are given,
the command will operate on all constraint types:

    -assert
        $assert cells, representing assert(...) constraints

    -assume
        $assume cells, representing assume(...) constraints

    -live
        $live cells, representing ``assert(s_eventually ...)``

    -fair
        $fair cells, representing ``assume(s_eventually ...)``

    -cover
        $cover cells, representing cover() statements

Additionally chformal will operate on $check cells corresponding to the
selected constraint types.

Exactly one of the following modes must be specified:

    -remove
        remove the cells and thus constraints from the design

    -early
        bypass FFs that only delay the activation of a constraint. When inputs
```

(continues on next page)

(continued from previous page)

```
        of the bypassed FFs do not remain stable between clock edges, this may result
        in unexpected behavior.

    -delay <N>
        delay activation of the constraint by <N> clock cycles

    -skip <N>
        ignore activation of the constraint in the first <N> clock cycles

    -coverenable
        add cover statements for the enable signals of the constraints

    -assert2assume
    -assert2cover
    -assume2assert
    -live2fair
    -fair2live
        change the roles of cells as indicated. these options can be combined

    -lower
        convert each $check cell into an $assert, $assume, $live, $fair or
        $cover cell. If the $check cell contains a message, also produce a $print
        cell.
```

RST generated from autocmd

- below is the raw RST output from `autocmd` (`YosysCmdDocumenter` class in `docs/util/cmd_documenter.py`) for *`chformal`* command
- heading will be rendered as a subheading of the most recent heading (see *chformal autocmd* above rendered under *Writing command help*)
- `.. cmd:def:: <cmd>` line is indexed for cross references with `:cmd:ref:` directive (*chformal autocmd* above uses `:noindex:` option so that *`chformal`* still links to the correct location)
  - `:title:` option controls text that appears when hovering over the *`chformal`* link
- commands with warning flags (experimental or internal) add a `.. warning` block before any of the help content
- if a command has no `source_location` the `.. note` at the bottom will instead link to /cmd/index_other

Listing 10.3: Generated rst for `.. autocmd:: chformal`

```
chformal - change formal constraints of the design
##################################################
.. cmd:def:: chformal
   :title: change formal constraints of the design

   .. cmd:usage:: chformal::chformal [types] [mode] [options] [selection]

   Make changes to the formal constraints of the design. The [types] options the type of␣
```

(continues on next page)

(continued from previous page)

```
↪constraint to operate on. If none of the following options are given, the command will␣
↪operate on all constraint types:

   :option -assert:
      `$assert` cells, representing ``assert(...)`` constraints

   :option -assume:
      `$assume` cells, representing ``assume(...)`` constraints

   :option -live:
      `$live` cells, representing ``assert(s_eventually ...)``

   :option -fair:
      `$fair` cells, representing ``assume(s_eventually ...)``

   :option -cover:
      `$cover` cells, representing ``cover()`` statements

   Additionally chformal will operate on `$check` cells corresponding to the selected␣
↪constraint types.

   Exactly one of the following modes must be specified:

   :option -remove:
      remove the cells and thus constraints from the design

   :option -early:
      bypass FFs that only delay the activation of a constraint. When inputs of the␣
↪bypassed FFs do not remain stable between clock edges, this may result in unexpected␣
↪behavior.

   :option -delay <N>:
      delay activation of the constraint by <N> clock cycles

   :option -skip <N>:
      ignore activation of the constraint in the first <N> clock cycles

   :option -coverenable:
      add cover statements for the enable signals of the constraints

   :option -assert2assume:
   :option -assert2cover:
   :option -assume2assert:
   :option -live2fair:
   :option -fair2live:
      change the roles of cells as indicated. these options can be combined

   :option -lower:
      convert each $check cell into an $assert, $assume, $live, $fair or $cover cell. If␣
↪the $check cell contains a message, also produce a $print cell.
```

BIBLIOGRAPHY

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[A+02] IEEE Standards Association and others. Ieee standard for verilog register transfer level synthesis. *IEEE Std 1364.1-2002*, 2002. doi:10.1109/IEEESTD.2002.94220.

[A+04] IEEE Standards Association and others. Ieee standard for vhdl register transfer level (rtl) synthesis. *IEEE Std 1076.6-2004 (Revision of IEEE Std 1076.6-1999)*, 2004. doi:10.1109/IEEESTD.2004.94802.

[A+06] IEEE Standards Association and others. Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006. doi:10.1109/IEEESTD.2006.99495.

[A+09] IEEE Standards Association and others. Ieee standard vhdl language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, 26 2009. doi:10.1109/IEEESTD.2009.4772740.

[A+10] IEEE Standards Association and others. Ieee standard for ip-xact, standard structure for packaging, integrating, and reusing ip within tools flows. *IEEE Std 1685-2009*, pages C1–360, 2010. doi:10.1109/IEEESTD.2010.5417309.

[BHSV90] R.K. Brayton, G.D. Hachtel, and A.L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, 1990. doi:10.1109/5.52213.

[CI00] Clifford E. Cummings and Sunburst Design Inc. Nonblocking assignments in verilog synthesis, coding styles that kill. In *SNUG (Synopsys Users Group) 2000 User Papers, section-MC1 (1 st paper*. 2000.

[EenSorensson03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[GW13] Johann Glaser and C. Wolf. Methodology and example-driven interconnect synthesis for designing heterogeneous coarse-grain reconfigurable architectures. In Jan Haase, editor, *Advances in Models, Methods, and Tools for Complex Chip Design — Selected contributions from FDL'12*. Springer, 2013.

[HS96] G D Hachtel and F Somenzi. Logic synthesis and verification algorithms. 1996.

[LHBB85] Kyu Y. Lee, Michael Holley, Mary Bailey, and Walter Bright. A high-level design language for programmable logic devices. *VLSI Design (Manhasset NY: CPM Publications)*, pages 50–62, June 1985.

[STGR10] Yiqiong Shi, Chan Wai Ting, Bah-Hwee Gwee, and Ye Ren. A highly efficient method for extracting fsms from flattened gate-level netlist. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, 2610–2613. 2010. doi:10.1109/ISCAS.2010.5537093.

[Ull76] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976. doi:10.1145/321921.321925.

[Wol13] C. Wolf. Design and implementation of the yosys open synthesis suite. Bachelor Thesis, Vienna University of Technology, 2013.

[WGS+12] C. Wolf, Johann Glaser, Florian Schupfer, Jan Haase, and Christoph Grimm. Example-driven interconnect synthesis for heterogeneous coarse-grain reconfigurable logic. In *FDL Proceeding of the 2012 Forum on Specification and Design Languages*, 194–201. 2012.

PROPERTY INDEX

is_

is_evaluable, 421
$alu, 317
$fa, 318
$lcu, 319
$macc, 320
$macc_v2, 323
$add, 270
$and, 271
$bweqx, 272
$div, 272
$divfloor, 273
$eq, 274
$eqx, 274
$ge, 275
$gt, 275
$le, 276
$logic_and, 277
$logic_or, 277
$lt, 278
$mod, 278
$modfloor, 279
$mul, 280
$ne, 281
$nex, 281
$or, 282
$pow, 282
$shift, 283
$shiftx, 284
$shl, 285
$shr, 285
$sshl, 286
$sshr, 286
$sub, 287
$xnor, 288
$xor, 288
$_ANDNOT_, 348
$_AOI3_, 348
$_AOI4_, 349
$_MUX16_, 349
$_MUX4_, 350
$_MUX8_, 351
$_NMUX_, 351
$_OAI3_, 352
$_OAI4_, 352
$_ORNOT_, 353
$_AND_, 343
$_BUF_, 343
$_MUX_, 344
$_NAND_, 344
$_NOR_, 345
$_NOT_, 345
$_OR_, 345
$_XNOR_, 346
$_XOR_, 346
$lut, 325
$sop, 326
$bmux, 289
$bwmux, 290
$demux, 290
$mux, 291
$pmux, 291
$buf, 264
$logic_not, 264
$neg, 265
$not, 265
$pos, 266
$reduce_and, 266
$reduce_bool, 267
$reduce_or, 267
$reduce_xnor, 268
$reduce_xor, 268
$concat, 341
$slice, 342

X-

x-aware, 421
$bweqx, 272
$eqx, 274
$nex, 281
x-output, 421
$div, 272
$mod, 278
$shiftx, 284

$pmux, 291

INTERNAL CELL REFERENCE

Internal cell

$alu, 317
$fa, 318
$lcu, 319
$macc, 320
$macc_v2, 323
$add, 270
$and, 271
$bweqx, 272
$div, 272
$divfloor, 273
$eq, 274
$eqx, 274
$ge, 275
$gt, 275
$le, 276
$logic_and, 277
$logic_or, 277
$lt, 278
$mod, 278
$modfloor, 279
$mul, 280
$ne, 281
$nex, 281
$or, 282
$pow, 282
$shift, 283
$shiftx, 284
$shl, 285
$shr, 285
$sshl, 286
$sshr, 286
$sub, 287
$xnor, 288
$xor, 288
$_ANDNOT_, 348
$_AOI3_, 348
$_AOI4_, 349
$_MUX16_, 349
$_MUX4_, 350
$_MUX8_, 351
$_NMUX_, 351
$_OAI3_, 352
$_OAI4_, 352
$_ORNOT_, 353
$_AND_, 343
$_BUF_, 343
$_MUX_, 344
$_NAND_, 344
$_NOR_, 345
$_NOT_, 345
$_OR_, 345
$_XNOR_, 346
$_XOR_, 346
$check, 338
$print, 339
$scopeinfo, 339
$allconst, 332
$allseq, 332
$anyconst, 332
$anyinit, 333
$anyseq, 333
$assert, 333
$assume, 334
$cover, 334
$equiv, 334
$fair, 335
$ff, 335
$initstate, 335
$live, 336
$future_ff, 336
$get_tag, 336
$original_tag, 337
$overwrite_tag, 337
$set_tag, 337
$fsm, 315
$_TBUF_, 420
$lut, 325
$sop, 326
$mem, 306
$mem_v2, 308
$meminit, 311
$meminit_v2, 312
$memrd, 312

$memrd_v2, 313
$memwr, 313
$memwr_v2, 314
$bmux, 289
$bwmux, 290
$demux, 290
$mux, 291
$pmux, 291
$tribuf, 292
$adff, 294
$adffe, 295
$adlatch, 295
$aldff, 296
$aldffe, 296
$dff, 297
$dffe, 297
$dffsr, 298
$dffsre, 299
$dlatch, 299
$dlatchsr, 300
$sdff, 300
$sdffce, 301
$sdffe, 301
$sr, 302
$_ALDFFE_NNN_, 357
$_ALDFFE_NNP_, 357
$_ALDFFE_NPN_, 358
$_ALDFFE_NPP_, 358
$_ALDFFE_PNN_, 359
$_ALDFFE_PNP_, 359
$_ALDFFE_PPN_, 360
$_ALDFFE_PPP_, 360
$_ALDFF_NN_, 361
$_ALDFF_NP_, 361
$_ALDFF_PN_, 362
$_ALDFF_PP_, 362
$_DFFE_NNON_, 362
$_DFFE_NNOP_, 363
$_DFFE_NN1N_, 363
$_DFFE_NN1P_, 364
$_DFFE_NN_, 364
$_DFFE_NPON_, 365
$_DFFE_NPOP_, 365
$_DFFE_NP1N_, 366
$_DFFE_NP1P_, 366
$_DFFE_NP_, 366
$_DFFE_PNON_, 367
$_DFFE_PNOP_, 367
$_DFFE_PN1N_, 368
$_DFFE_PN1P_, 368
$_DFFE_PN_, 369
$_DFFE_PPON_, 369
$_DFFE_PPOP_, 369
$_DFFE_PP1N_, 370
$_DFFE_PP1P_, 370
$_DFFE_PP_, 371
$_DFFSRE_NNNN_, 371
$_DFFSRE_NNNP_, 372
$_DFFSRE_NNPN_, 372
$_DFFSRE_NNPP_, 373
$_DFFSRE_NPNN_, 373
$_DFFSRE_NPNP_, 374
$_DFFSRE_NPPN_, 374
$_DFFSRE_NPPP_, 375
$_DFFSRE_PNNN_, 376
$_DFFSRE_PNNP_, 376
$_DFFSRE_PNPN_, 377
$_DFFSRE_PNPP_, 377
$_DFFSRE_PPNN_, 378
$_DFFSRE_PPNP_, 378
$_DFFSRE_PPPN_, 379
$_DFFSRE_PPPP_, 379
$_DFFSR_NNN_, 380
$_DFFSR_NNP_, 380
$_DFFSR_NPN_, 381
$_DFFSR_NPP_, 382
$_DFFSR_PNN_, 382
$_DFFSR_PNP_, 383
$_DFFSR_PPN_, 383
$_DFFSR_PPP_, 384
$_DFF_NNO_, 384
$_DFF_NN1_, 385
$_DFF_NPO_, 385
$_DFF_NP1_, 386
$_DFF_N_, 386
$_DFF_PNO_, 386
$_DFF_PN1_, 387
$_DFF_PPO_, 387
$_DFF_PP1_, 388
$_DFF_P_, 388
$_FF_, 389
$_SDFFCE_NNON_, 389
$_SDFFCE_NNOP_, 389
$_SDFFCE_NN1N_, 390
$_SDFFCE_NN1P_, 390
$_SDFFCE_NPON_, 391
$_SDFFCE_NPOP_, 391
$_SDFFCE_NP1N_, 392
$_SDFFCE_NP1P_, 392
$_SDFFCE_PNON_, 393
$_SDFFCE_PNOP_, 393
$_SDFFCE_PN1N_, 394
$_SDFFCE_PN1P_, 394
$_SDFFCE_PPON_, 395
$_SDFFCE_PPOP_, 395
$_SDFFCE_PP1N_, 396
$_SDFFCE_PP1P_, 396
$_SDFFE_NNON_, 397

$_SDFFE_NN0P_, 397
$_SDFFE_NN1N_, 398
$_SDFFE_NN1P_, 398
$_SDFFE_NP0N_, 399
$_SDFFE_NP0P_, 399
$_SDFFE_NP1N_, 400
$_SDFFE_NP1P_, 400
$_SDFFE_PN0N_, 401
$_SDFFE_PN0P_, 401
$_SDFFE_PN1N_, 402
$_SDFFE_PN1P_, 402
$_SDFFE_PP0N_, 403
$_SDFFE_PP0P_, 403
$_SDFFE_PP1N_, 404
$_SDFFE_PP1P_, 404
$_SDFF_NN0_, 405
$_SDFF_NN1_, 405
$_SDFF_NP0_, 406
$_SDFF_NP1_, 406
$_SDFF_PN0_, 407
$_SDFF_PN1_, 407
$_SDFF_PP0_, 407
$_SDFF_PP1_, 408
$_DLATCHSR_NNN_, 410
$_DLATCHSR_NNP_, 410
$_DLATCHSR_NPN_, 411
$_DLATCHSR_NPP_, 411
$_DLATCHSR_PNN_, 412
$_DLATCHSR_PNP_, 412
$_DLATCHSR_PPN_, 413
$_DLATCHSR_PPP_, 413
$_DLATCH_NN0_, 414
$_DLATCH_NN1_, 414
$_DLATCH_NP0_, 415
$_DLATCH_NP1_, 415
$_DLATCH_N_, 416
$_DLATCH_PN0_, 416
$_DLATCH_PN1_, 417
$_DLATCH_PP0_, 417
$_DLATCH_PP1_, 417
$_DLATCH_P_, 418
$_SR_NN_, 418
$_SR_NP_, 419
$_SR_PN_, 419
$_SR_PP_, 420
$specify2, 327
$specify3, 328
$specrule, 331
$buf, 264
$logic_not, 264
$neg, 265
$not, 265
$pos, 266
$reduce_and, 266
$reduce_bool, 267
$reduce_or, 267
$reduce_xnor, 268
$reduce_xor, 268
$concat, 341
$connect, 341
$input_port, 342
$slice, 342

TAG INDEX

read_aiger, 449
read_blif, 450
read_json, 450
read_liberty, 450
read_rtlil, 451
read_verilog, 452
read_verilog_file_list, 455
read_xaiger2, 455
assertpmux, 456
async2sync, 456
chformal, 457
clk2fflogic, 458
cutpoint, 458
dft_tag, 458
fmcombine, 459
fminit, 460
formalff, 460
freduce, 461
future, 462
glift, 462
miter, 464
mutate, 465
qbfsat, 467
sat, 468
supercover, 472
synthprop, 472
xprop, 472
write_aiger, 425
write_aiger2, 426
write_blif, 427
write_btor, 428
write_cxxrtl, 429
write_edif, 434
write_firrtl, 434
write_functional_cxx, 435
write_functional_rosette, 435
write_functional_smt2, 435
write_intersynth, 435
write_jny, 436
write_json, 436
write_rtlil, 441
write_simplec, 441
write_smt2, 442
write_smv, 445
write_spice, 445
write_table, 446
write_verilog, 446
write_xaiger, 448
write_xaiger2, 449
example_dt, 519
internal_stats, 520
linux_perf, 520
test_abcloop, 520
test_autotb, 521
test_cell, 522
test_generic, 523
test_pmgen, 523
abc, ??
abc9, ??
abc9_exe, ??
abc9_ops, ??
abc_new, ??
abstract, ??
add, ??
aigmap, ??
alumacc, ??
anlogic_eqn, ??
anlogic_fixcarry, ??
arith_tree, ??
attrmap, ??
attrmvcp, ??
autoname, ??
blackbox, ??
bmuxmap, ??
booth, ??
box_derive, ??
bufnorm, ??
bugpoint, ??
bwmuxmap, ??
cellmatch, ??
chparam, ??
chtype, ??
clean_zerowidth, ??
clkbufmap, ??

clockgate, ??
connect, ??
connect_rpc, ??
connwrappers, ??
constmap, ??
coolrunner2_fixup, ??
coolrunner2_sop, ??
copy, ??
delete, ??
deminout, ??
demuxmap, ??
design, ??
design_equal, ??
dffinit, ??
dfflegalize, ??
dfflibmap, ??
dffunmap, ??
dump, ??
echo, ??
efinix_fixcarry, ??
eval, ??
extract, ??
extract_counter, ??
extract_fa, ??
extract_reduce, ??
extractinv, ??
flatten, ??
flowmap, ??
fst2tb, ??
gatemate_foldinv, ??
greenpak4_dffinv, ??
help, ??
hierarchy, ??
hilomap, ??
history, ??
ice40_braminit, ??
ice40_dsp, ??
ice40_opt, ??
ice40_wrapcarry, ??
icell_liberty, ??
insbuf, ??
iopadmap, ??
jny, ??
json, ??
keep_hierarchy, ??
lattice_dsp_nexus, ??
lattice_gsr, ??
libcache, ??
license, ??
linecoverage, ??
lut2bmux, ??
lut2mux, ??
maccmap, ??
microchip_dffopt, ??
microchip_dsp, ??
muxcover, ??
muxpack, ??
nlutmap, ??
nx_carry, ??
onehot, ??
opensta, ??
paramap, ??
peepopt, ??
pmux2shiftx, ??
pmuxtree, ??
prep, ??
ql_bram_merge, ??
ql_bram_types, ??
ql_dsp_io_regs, ??
ql_dsp_macc, ??
ql_dsp_simd, ??
ql_ioff, ??
raise_error, ??
read, ??
rename, ??
scatter, ??
script, ??
sdc, ??
sdc_expand, ??
setattr, ??
setparam, ??
setundef, ??
share, ??
shell, ??
shregmap, ??
sim, ??
simplemap, ??
sort, ??
splice, ??
splitcells, ??
splitnets, ??
submod, ??
synth, ??
synth_achronix, ??
synth_analogdevices, ??
synth_anlogic, ??
synth_coolrunner2, ??
synth_easic, ??
synth_ecp5, ??
synth_efinix, ??
synth_fabulous, ??
synth_gatemate, ??
synth_gowin, ??
synth_greenpak4, ??
synth_ice40, ??
synth_intel, ??
synth_intel_alm, ??
synth_lattice, ??

synth_microchip, ??
synth_nanoxplore, ??
synth_nexus, ??
synth_quicklogic, ??
synth_sf2, ??
synth_xilinx, ??
tcl, ??
techmap, ??
test_select, ??
timeest, ??
tribuf, ??
uniquify, ??
verific, ??
verilog_defaults, ??
verilog_defines, ??
wbflip, ??
wrapcell, ??
wreduce, ??
xilinx_dffopt, ??
xilinx_dsp, ??
xilinx_srl, ??
zinit, ??
expose, 493
equiv_add, 493
equiv_induct, 494
equiv_make, 494
equiv_mark, 495
equiv_miter, 495
equiv_opt, 495
equiv_purge, 497
equiv_remove, 497
equiv_simple, 497
equiv_status, 498
equiv_struct, 498
fsm, 477
fsm_detect, 478
fsm_expand, 478
fsm_export, 478
fsm_extract, 479
fsm_info, 479
fsm_map, 479
fsm_opt, 479
fsm_recode, 479
memory, 480
memory_bmux2rom, 481
memory_bram, 481
memory_collect, 483
memory_dff, 483
memory_libmap, 483
memory_map, 484
memory_memx, 485
memory_narrow, 485
memory_nordff, 485
memory_share, 485
memory_unpack, 486
clean, 486
opt, 486
opt_balance_tree, 487
opt_clean, 487
opt_demorgan, 488
opt_dff, 488
opt_expr, 488
opt_ffinv, 489
opt_hier, 489
opt_lut, 490
opt_lut_ins, 490
opt_mem, 490
opt_mem_feedback, 490
opt_mem_priority, 491
opt_mem_widen, 491
opt_merge, 491
opt_muxtree, 491
opt_reduce, 492
opt_share, 492
recover_names, 492
proc, 474
proc_arst, 475
proc_clean, 475
proc_dff, 475
proc_dlatch, 475
proc_init, 476
proc_memwr, 476
proc_mux, 476
proc_prune, 476
proc_rmdead, 476
proc_rom, 476
cd, 499
check, 499
debug, 500
edgetypes, 500
exec, 500
log, 501
logger, 502
ls, 503
ltp, 503
plugin, 503
portarcs, 503
portlist, 504
printattrs, 504
scc, 504
scratchpad, 505
select, 506
setenv, 510
show, 511
sta, 513
stat, 513
tee, 513
torder, 514

trace, 514
viz, 514
write_file, 516
rmports, 518

COMMAND REFERENCE

Command

abc, ??
abc9, ??
abc9_exe, ??
abc9_ops, ??
abc_new, ??
abstract, ??
add, ??
aigmap, ??
alumacc, ??
anlogic_eqn, ??
anlogic_fixcarry, ??
arith_tree, ??
assertpmux, 456
async2sync, 456
attrmap, ??
attrmvcp, ??
autoname, ??
blackbox, ??
bmuxmap, ??
booth, ??
box_derive, ??
bufnorm, ??
bugpoint, ??
bwmuxmap, ??
cd, 499
cellmatch, ??
check, 499
chformal, 457
chparam, ??
chtype, ??
clean, 486
clean_zerowidth, ??
clk2fflogic, 458
clkbufmap, ??
clockgate, ??
connect, ??
connect_rpc, ??
connwrappers, ??
constmap, ??
coolrunner2_fixup, ??
coolrunner2_sop, ??
copy, ??
cutpoint, 458
debug, 500
delete, ??
deminout, ??
demuxmap, ??
design, ??
design_equal, ??
dffinit, ??
dfflegalize, ??
dfflibmap, ??
dffunmap, ??
dft_tag, 458
dump, ??
echo, ??
edgetypes, 500
efinix_fixcarry, ??
equiv_add, 493
equiv_induct, 494
equiv_make, 494
equiv_mark, 495
equiv_miter, 495
equiv_opt, 495
equiv_purge, 497
equiv_remove, 497
equiv_simple, 497
equiv_status, 498
equiv_struct, 498
eval, ??
example_dt, 519
exec, 500
expose, 493
extract, ??
extract_counter, ??
extract_fa, ??
extract_reduce, ??
extractinv, ??
flatten, ??
flowmap, ??
fmcombine, 459
fminit, 460
formalff, 460

freduce, 461
fsm, 477
fsm_detect, 478
fsm_expand, 478
fsm_export, 478
fsm_extract, 479
fsm_info, 479
fsm_map, 479
fsm_opt, 479
fsm_recode, 479
fst2tb, ??
future, 462
gatemate_foldinv, ??
glift, 462
greenpak4_dffinv, ??
help, ??
hierarchy, ??
hilomap, ??
history, ??
ice40_braminit, ??
ice40_dsp, ??
ice40_opt, ??
ice40_wrapcarry, ??
icell_liberty, ??
insbuf, ??
internal_stats, 520
iopadmap, ??
jny, ??
json, ??
keep_hierarchy, ??
lattice_dsp_nexus, ??
lattice_gsr, ??
libcache, ??
license, ??
linecoverage, ??
linux_perf, 520
log, 501
logger, 502
ls, 503
ltp, 503
lut2bmux, ??
lut2mux, ??
maccmap, ??
memory, 480
memory_bmux2rom, 481
memory_bram, 481
memory_collect, 483
memory_dff, 483
memory_libmap, 483
memory_map, 484
memory_memx, 485
memory_narrow, 485
memory_nordff, 485
memory_share, 485
memory_unpack, 486
microchip_dffopt, ??
microchip_dsp, ??
miter, 464
mutate, 465
muxcover, ??
muxpack, ??
nlutmap, ??
nx_carry, ??
onehot, ??
opensta, ??
opt, 486
opt_balance_tree, 487
opt_clean, 487
opt_demorgan, 488
opt_dff, 488
opt_expr, 488
opt_ffinv, 489
opt_hier, 489
opt_lut, 490
opt_lut_ins, 490
opt_mem, 490
opt_mem_feedback, 490
opt_mem_priority, 491
opt_mem_widen, 491
opt_merge, 491
opt_muxtree, 491
opt_reduce, 492
opt_share, 492
paramap, ??
peepopt, ??
plugin, 503
pmux2shiftx, ??
pmuxtree, ??
portarcs, 503
portlist, 504
prep, ??
printattrs, 504
proc, 474
proc_arst, 475
proc_clean, 475
proc_dff, 475
proc_dlatch, 475
proc_init, 476
proc_memwr, 476
proc_mux, 476
proc_prune, 476
proc_rmdead, 476
proc_rom, 476
qbfsat, 467
ql_bram_merge, ??
ql_bram_types, ??
ql_dsp_io_regs, ??
ql_dsp_macc, ??

ql_dsp_simd, ??
ql_ioff, ??
raise_error, ??
read, ??
read_aiger, 449
read_blif, 450
read_json, 450
read_liberty, 450
read_rtlil, 451
read_verilog, 452
read_verilog_file_list, 455
read_xaiger2, 455
recover_names, 492
rename, ??
rmports, 518
sat, 468
scatter, ??
scc, 504
scratchpad, 505
script, ??
sdc, ??
sdc_expand, ??
select, 506
setattr, ??
setenv, 510
setparam, ??
setundef, ??
share, ??
shell, ??
show, 511
shregmap, ??
sim, ??
simplemap, ??
sort, ??
splice, ??
splitcells, ??
splitnets, ??
sta, 513
stat, 513
submod, ??
supercover, 472
synth, ??
synth_achronix, ??
synth_analogdevices, ??
synth_anlogic, ??
synth_coolrunner2, ??
synth_easic, ??
synth_ecp5, ??
synth_efinix, ??
synth_fabulous, ??
synth_gatemate, ??
synth_gowin, ??
synth_greenpak4, ??
synth_ice40, ??
synth_intel, ??
synth_intel_alm, ??
synth_lattice, ??
synth_microchip, ??
synth_nanoxplore, ??
synth_nexus, ??
synth_quicklogic, ??
synth_sf2, ??
synth_xilinx, ??
synthprop, 472
tcl, ??
techmap, ??
tee, 513
test_abcloop, 520
test_autotb, 521
test_cell, 522
test_generic, 523
test_pmgen, 523
test_select, ??
timeest, ??
torder, 514
trace, 514
tribuf, ??
uniquify, ??
verific, ??
verilog_defaults, ??
verilog_defines, ??
viz, 514
wbflip, ??
wrapcell, ??
wreduce, ??
write_aiger, 425
write_aiger2, 426
write_blif, 427
write_btor, 428
write_cxxrtl, 429
write_edif, 434
write_file, 516
write_firrtl, 434
write_functional_cxx, 435
write_functional_rosette, 435
write_functional_smt2, 435
write_intersynth, 435
write_jny, 436
write_json, 436
write_rtlil, 441
write_simplec, 441
write_smt2, 442
write_smv, 445
write_spice, 445
write_table, 446
write_verilog, 446
write_xaiger, 448
write_xaiger2, 449

xilinx_dffopt, ??
xilinx_dsp, ??
xilinx_srl, ??
xprop, 472
zinit, ??