Yosys Open SYnthesis Suite

Claire Xenia Wolf

https://yosyshq.net/yosys/

October 4, 2024

Abstract

Yosys is the first full-featured open source software for Verilog HDL synthesis. It supports most of Verilog-2005 and is well tested with real-world designs from the ASIC and FPGA world.

Learn how to use Yosys to create your own custom synthesis flows and discover why open source HDL synthesis is important for researchers, hobbyists, educators and engineers alike.

This presentation covers basic concepts of Yosys, writing synthesis scripts for a wide range of applications, creating Yosys scripts for various non-synthesis applications (such as formal equivalence checking) and writing extensions to Yosys using the C++ API.

About me

Hi! I'm Claire Xenia Wolf.

I like writing open source software. For example:

- Yosys
- OpenSCAD (now maintained by Marius Kintel)
- SPL (a not very popular scripting language)
- EmbedVM (a very simple compiler+vm for 8 bit micros)
- Lib(X)SVF (a library to play SVF/XSVF files over JTAG)
- ROCK Linux (discontinued since 2010)

Outline

Yosys is an Open Source Verilog synthesis tool, and more.

Outline of this presentation:

- Introduction to the field and Yosys
- Yosys by example: synthesis
- Yosys by example: advanced synthesis
- Yosys by example: beyond synthesis
- Writing Yosys extensions in C++

Section 1

Introduction to Yosys

Levels of Abstraction for Digital Circuits

- **System Level**
- High Level
- Behavioral Level
- Register-Transfer Level (RTL)
- Logical Gate Level
- Physical Gate Level
- Switch Level

Definition: System Level

Overall view of the circuit. E.g. block-diagrams or instruction-set architecture descriptions.

Levels of Abstraction for Digital Circuits

- System Level
- **High Level**
- Behavioral Level
- Register-Transfer Level (RTL)
- Logical Gate Level
- Physical Gate Level
- Switch Level

Definition: High Level

Functional implementation of circuit in high-level programming language (C, C++, SystemC, Matlab, Python, etc.).

Levels of Abstraction for Digital Circuits

- System Level
- High Level
- **Behavioral Level**
- Register-Transfer Level (RTL)
- Logical Gate Level
- Physical Gate Level
- Switch Level

Definition: Behavioral Level

Cycle-accurate description of circuit in hardware description language (Verilog, VHDL, etc.).

Levels of Abstraction for Digital Circuits

- System Level
- High Level
- Behavioral Level
- **Register-Transfer Level (RTL)**
- Logical Gate Level
- Physical Gate Level
- Switch Level

Definition: Register-Transfer Level (RTL)

List of registers (flip-flops) and logic functions that calculate the next state from the previous one. Usually a netlist utilizing high-level cells such as adders, multipliers, multiplexer, etc.

Levels of Abstraction for Digital Circuits

- System Level
- High Level
- Behavioral Level
- Register-Transfer Level (RTL)
- **Logical Gate Level**
- Physical Gate Level
- Switch Level

Definition: Logical Gate Level

Netlist of single-bit registers and basic logic gates (such as AND, OR, NOT, etc.). Popular form: And-Inverter-Graphs (AIGs) with pairs of primary inputs and outputs for each register bit.

Levels of Abstraction for Digital Circuits

- System Level
- High Level
- Behavioral Level
- Register-Transfer Level (RTL)
- Logical Gate Level
- **Physical Gate Level**
- Switch Level

Definition: Physical Gate Level

Netlist of cells that actually are available on the target architecture (such as CMOS gates in an ASIC or LUTs in an FPGA). Optimized for area, power, and/or speed (static timing or number of logic levels).

Levels of Abstraction for Digital Circuits

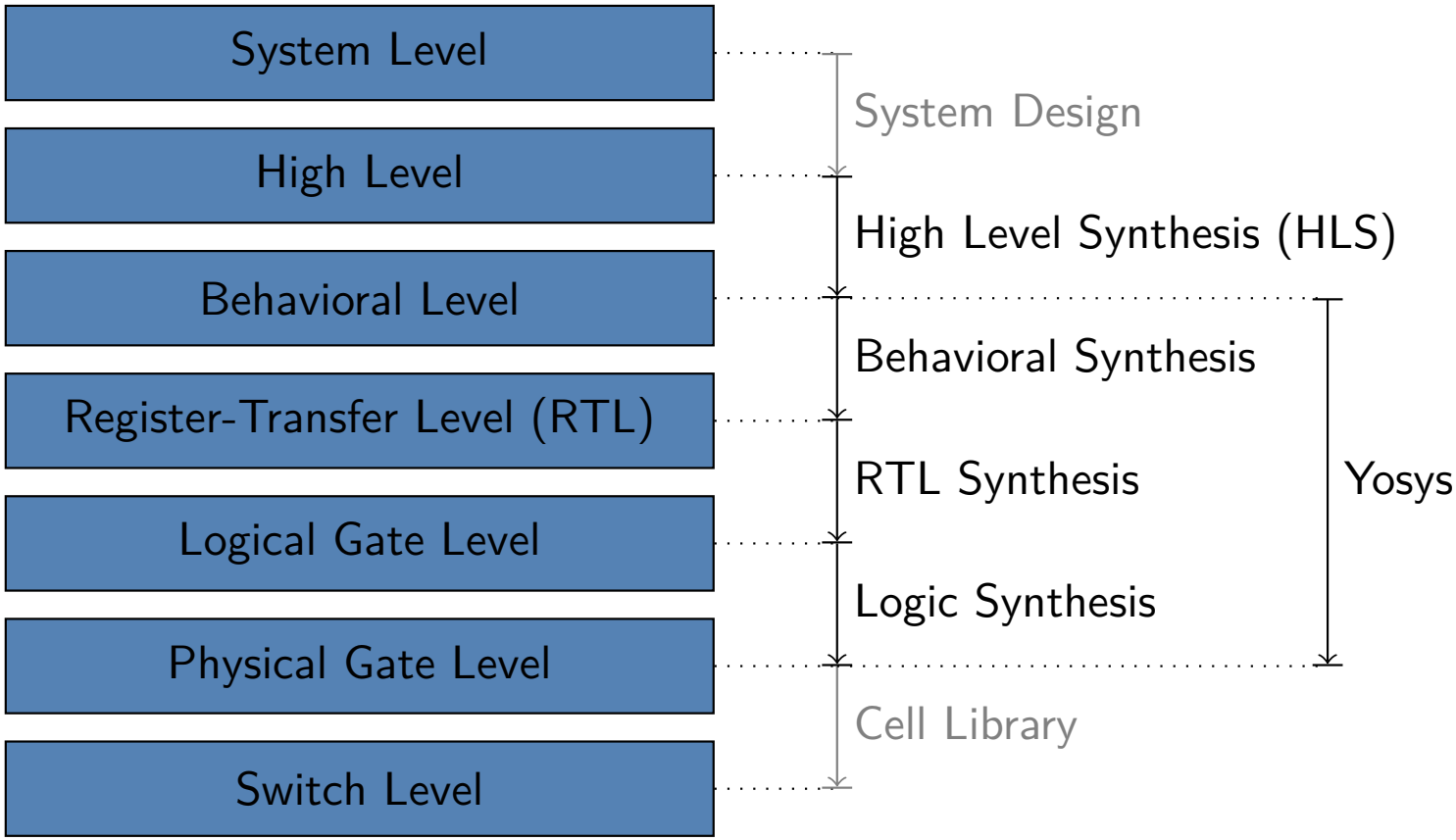
- System Level
- High Level
- Behavioral Level
- Register-Transfer Level (RTL)
- Logical Gate Level
- Physical Gate Level
- **Switch Level**

Definition: Switch Level

Netlist of individual transistors.

Digital Circuit Synthesis

Synthesis Tools (such as Yosys) can transform HDL code to circuits:



What Yosys can and can't do

Things Yosys can do:

- 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].

Things Yosys can't do:

- Process high-level languages such as C/C++/SystemC.
- Create physical layouts (place&route).

A typical flow combines Yosys with with a low-level implementation tool, such as Qflow[2] for ASIC designs.

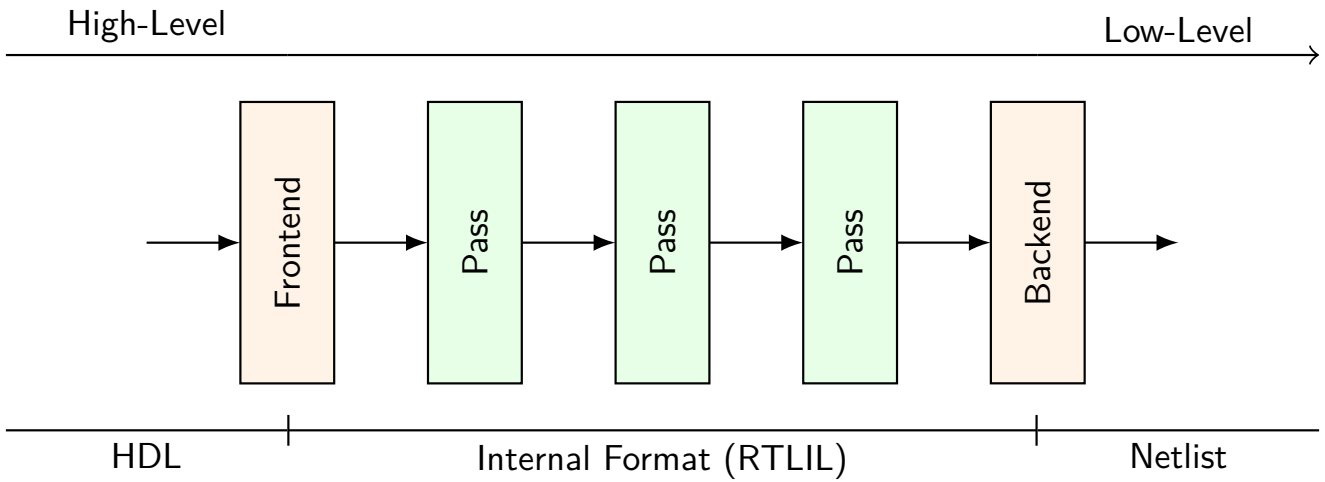
[1] `http://www.eecs.berkeley.edu/~alanmi/abc/`

[2] `http://opencircuitdesign.com/qflow/`

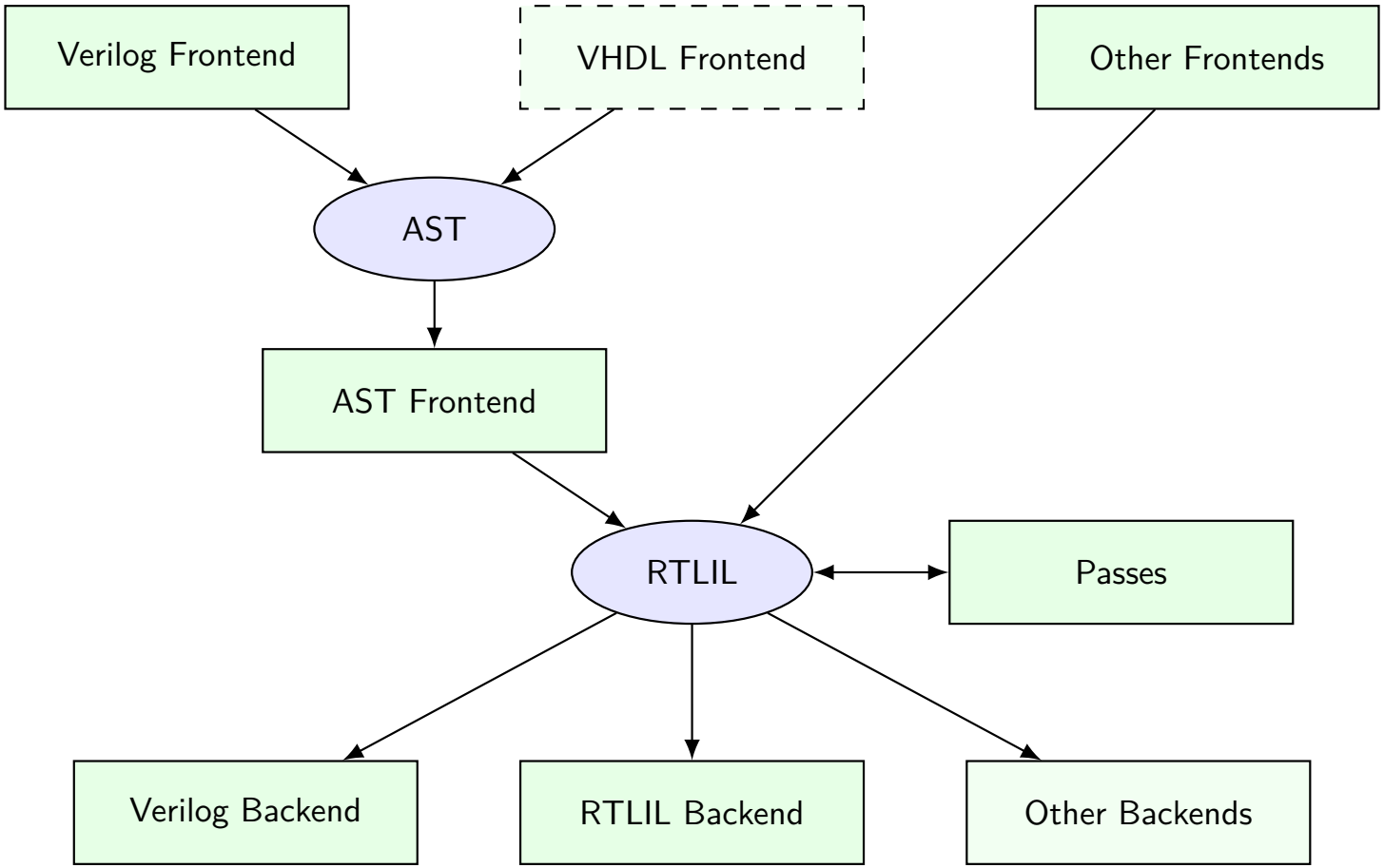
Yosys Data- and Control-Flow

A (usually short) synthesis script controls Yosys.
This 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, ...).



Program Components and Data Formats



Example Project

The following slides cover an example project. This project contains three files:

- A simple ASIC synthesis script
- A digital design written in Verilog
- A simple CMOS cell library

Direct link to the files:

`https://github.com/YosysHQ/yosys/tree/master/manual/PRESENTATION_Intro`

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: `read_verilog counter.v`

Read Verilog source file and convert to internal representation.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: `hierarchy -check -top counter`

Elaborate the design hierarchy. Should always be the first command after reading the design. Can re-run AST front-end.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: `proc`

Convert "processes" (the internal representation of behavioral Verilog code) into multiplexers and registers.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: opt

Perform some basic optimizations and cleanups.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: `fsm`

Analyze and optimize finite state machines.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: opt

Perform some basic optimizations and cleanups.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: `memory`

Analyze memories and create circuits to implement them.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: opt

Perform some basic optimizations and cleanups.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: techmap

Map coarse-grain RTL cells (adders, etc.) to fine-grain logic gates (AND, OR, NOT, etc.).

Example Project – Synthesis Script

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

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

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

```
# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: opt

Perform some basic optimizations and cleanups.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: `dfflibmap -liberty mycells.lib`

Map registers to available hardware flip-flops.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: `abc -liberty mycells.lib`

Map logic to available hardware gates.

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: `clean`

Clean up the design (just the last step of `opt`).

Example Project – Synthesis Script

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

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

# mapping to internal cell library
techmap; opt

# mapping flip-flops to mycells.lib
dfflibmap -liberty mycells.lib

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

# cleanup
clean

# write synthesized design
write_verilog synth.v
```

Command: `write_verilog synth.v`

Write final synthesis result to output file.

Example Project – Verilog Source: counter.v

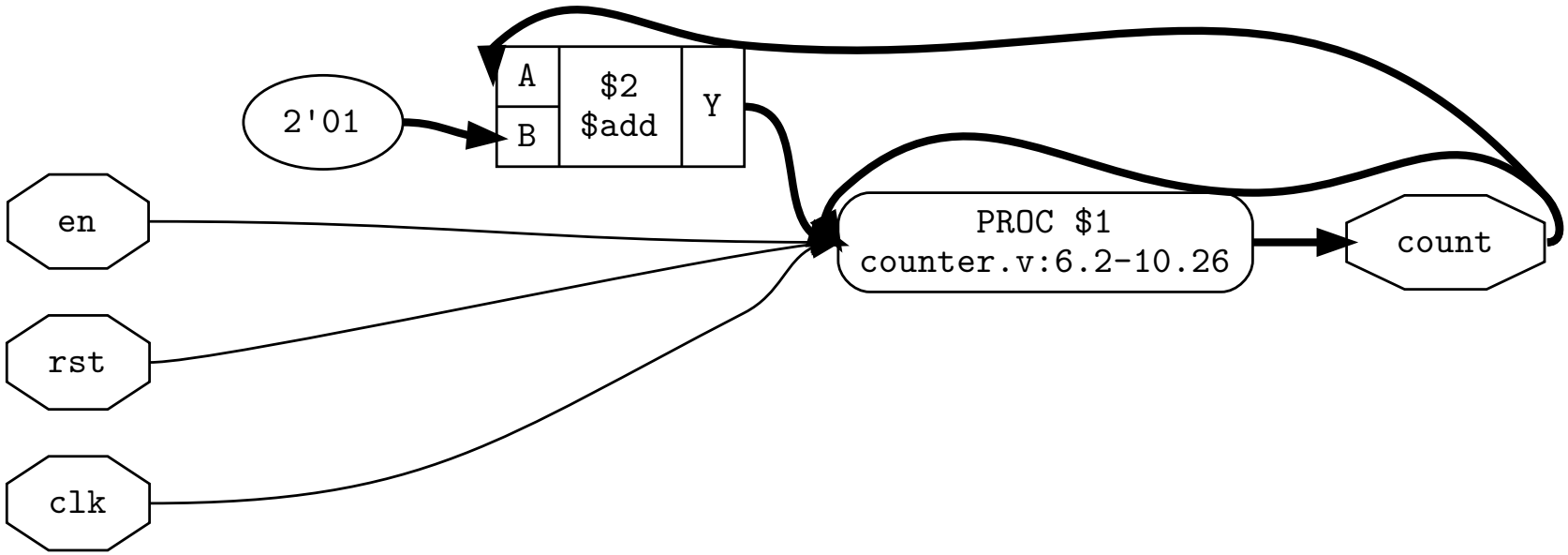
```
module counter (clk, rst, en, count);

        input clk, rst, en;
        output reg [1:0] count;

        always @(posedge clk)
                if (rst)
                        count <= 2'd0;
                else if (en)
                        count <= count + 2'd1;

endmodule
```

Example Project – Cell Library: `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;
             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"; }
  }
}
```

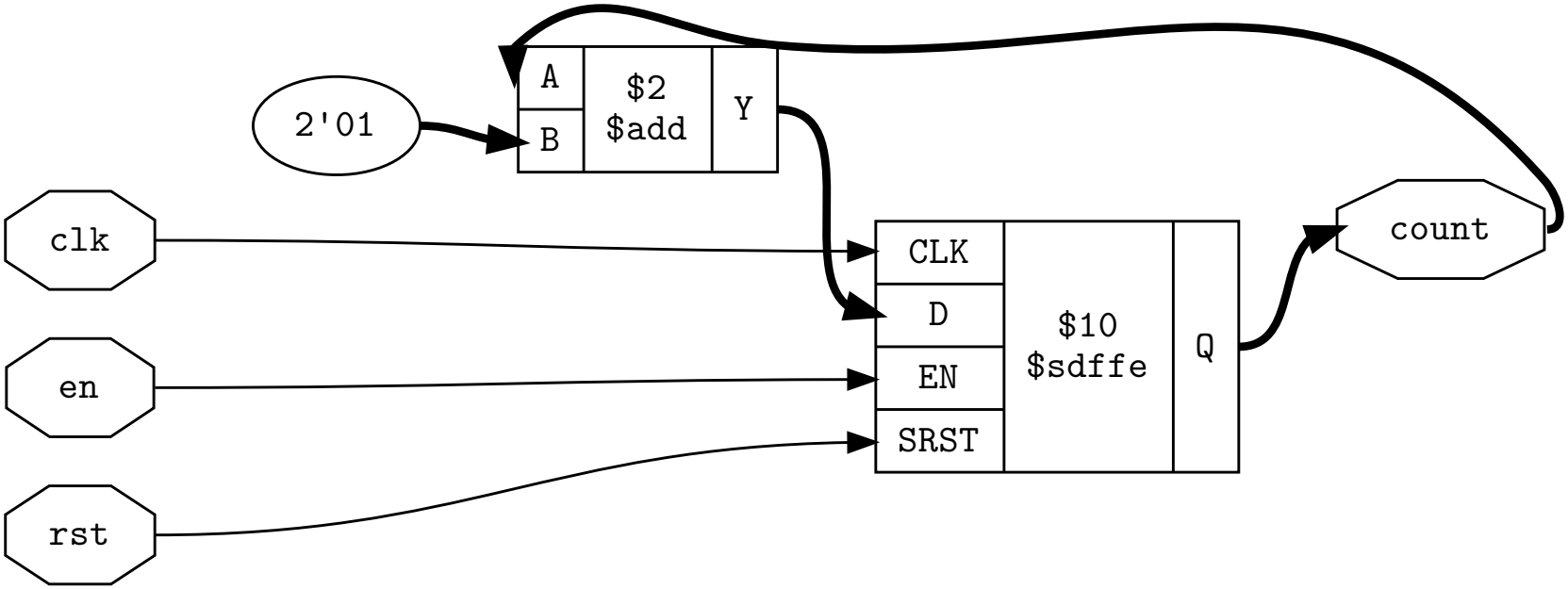
Running the Synthesis Script – Step 1/4

```
read_verilog counter.v
hierarchy -check -top counter
```



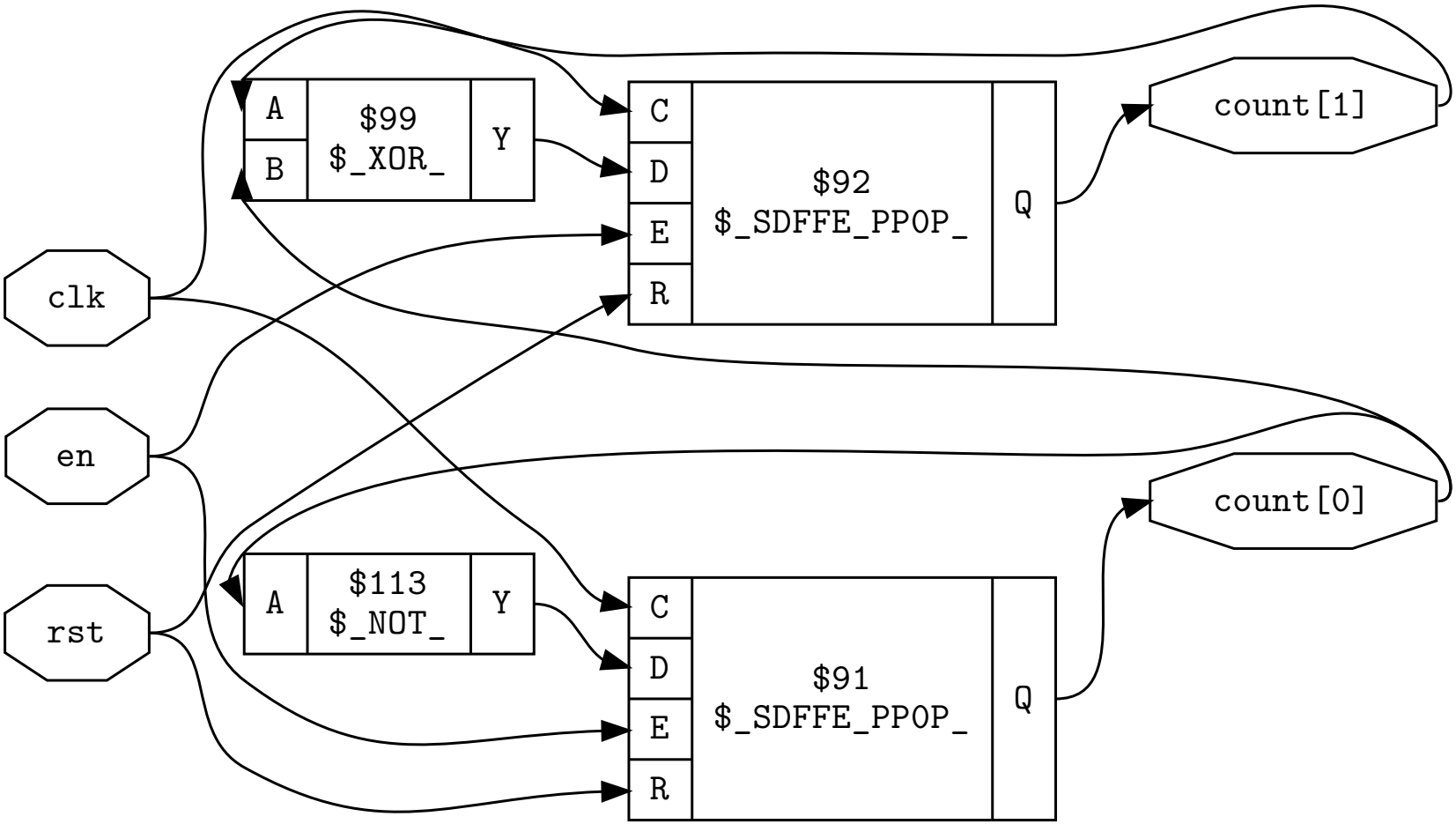
Running the Synthesis Script – Step 2/4

```
proc; opt; fsm; opt; memory; opt
```



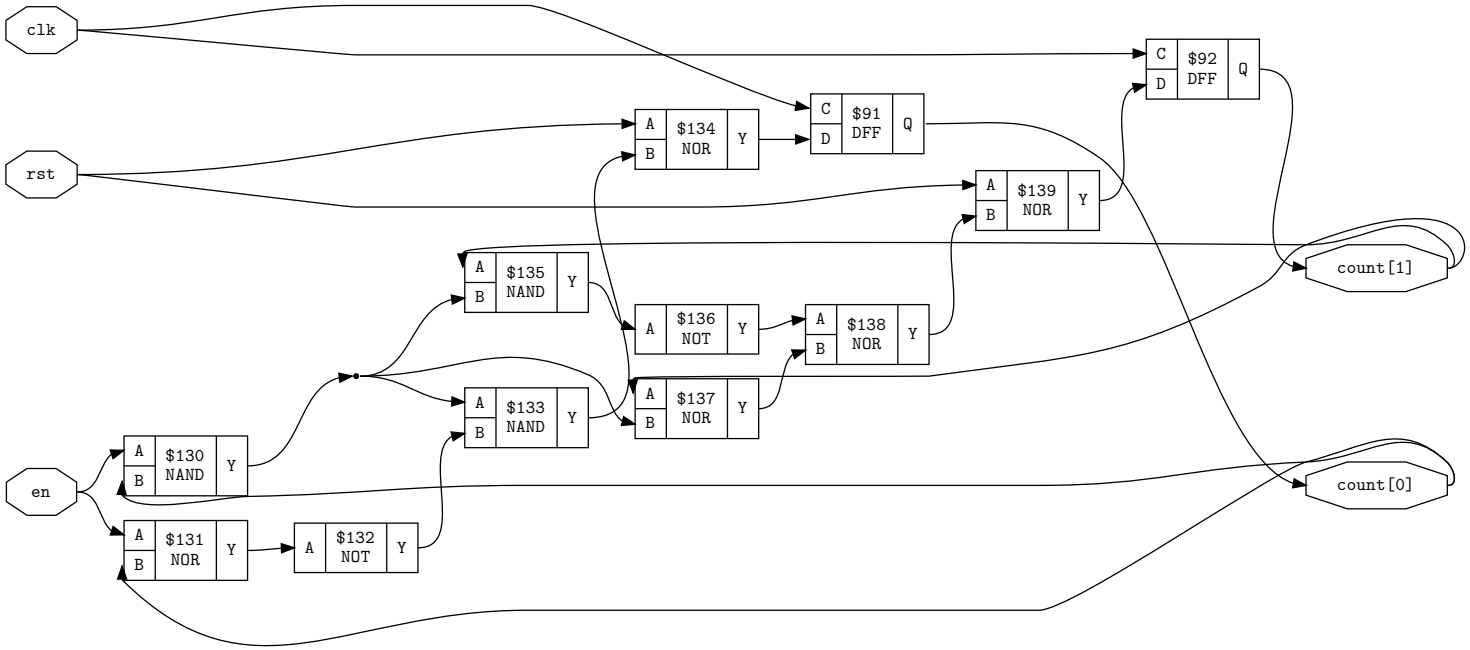
Running the Synthesis Script – Step 3/4

```
techmap; opt
```



Running the Synthesis Script – Step 4/4

```
dfflibmap -liberty mycells.lib
abc -liberty mycells.lib
clean
```



The synth command

Yosys contains a default (recommended example) synthesis script in form of the `synth` command. The following commands are executed by this synthesis command:

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

coarse:
    proc
    opt
    wreduce
    alumacc
    share
    opt
    fsm
    opt -fast
    memory -nomap
    opt_clean
```

```
fine:
    opt -fast -full
    memory_map
    opt -full
    techmap
    opt -fast

abc:
    abc -fast
    opt -fast
```

Yosys Commands 1/3 (excerpt)

Command reference:

- Use "`help`" for a command list and "`help` *command*" for details.
- Or run "`yosys -H`" or "`yosys -h` *command*".
- Or go to `https://yosyshq.net/yosys/documentation.html`.

Commands for design navigation and investigation:

```
cd                  # a shortcut for 'select -module <name>'
ls                  # list modules or objects in modules
dump                # print parts of the design in RTLIL format
show                # generate schematics using graphviz
select              # modify and view the list of selected objects
```

Commands for executing scripts or entering interactive mode:

```
shell               # enter interactive command mode
history             # show last interactive commands
script              # execute commands from script file
tcl                 # execute a TCL script file
```

Yosys Commands 2/3 (excerpt)

Commands for reading and elaborating the design:

```
read_rtlil          # read modules from RTLIL file
read_verilog        # read modules from Verilog file
hierarchy           # check, expand and clean up design hierarchy
```

Commands for high-level synthesis:

```
proc                # translate processes to netlists
fsm                 # extract and optimize finite state machines
memory              # translate memories to basic cells
opt                 # perform simple optimizations
```

Commands for technology mapping:

```
techmap             # generic technology mapper
abc                 # use ABC for technology mapping
dfflibmap           # technology mapping of flip-flops
hilomap             # technology mapping of constant hi- and/or lo-drivers
iopadmap            # technology mapping of i/o pads (or buffers)
flatten             # flatten design
```

Yosys Commands 3/3 (excerpt)

Commands for writing the results:

```
write_blif          # write design to BLIF file
write_btor          # write design to BTOR file
write_edif          # write design to EDIF netlist file
write_rtlil         # write design to RTLIL file
write_spice         # write design to SPICE netlist file
write_verilog       # write design to Verilog file
```

Script-Commands for standard synthesis tasks:

```
synth               # generic synthesis script
synth_xilinx        # synthesis for Xilinx FPGAs
```

Commands for model checking:

```
sat                 # solve a SAT problem in the circuit
miter               # automatically create a miter circuit
scc                 # detect strongly connected components (logic loops)
```

... and many many more.

More Verilog Examples 1/3

```
module detectprime(a, y);
    input [4:0] a;
    output y;

    integer i, j;
    reg [31:0] lut;

    initial begin
        for (i = 0; i < 32; i = i+1) begin
            lut[i] = i > 1;
            for (j = 2; j*j <= i; j = j+1)
                if (i % j == 0)
                    lut[i] = 0;
        end
    end

    assign y = lut[a];
endmodule
```

More Verilog Examples 2/3

```
module carryadd(a, b, y);
    parameter WIDTH = 8;
    input [WIDTH-1:0] a, b;
    output [WIDTH-1:0] y;

    genvar i;
    generate
        for (i = 0; i < WIDTH; i = i+1) begin:STAGE
            wire IN1 = a[i], IN2 = b[i];
            wire C, Y;
            if (i == 0)
                assign C = IN1 & IN2, Y = IN1 ^ IN2;
            else
                assign C = (IN1 & IN2) | ((IN1 | IN2) & STAGE[i-1].C),
                       Y = IN1 ^ IN2 ^ STAGE[i-1].C;
            assign y[i] = Y;
        end
    endgenerate
endmodule
```

More Verilog Examples 3/3

```
module cam(clk, wr_enable, wr_addr, wr_data, rd_data, rd_addr, rd_match);
    parameter WIDTH = 8;
    parameter DEPTH = 16;
    localparam ADDR_BITS = $clog2(DEPTH-1);

    input clk, wr_enable;
    input [ADDR_BITS-1:0] wr_addr;
    input [WIDTH-1:0] wr_data, rd_data;
    output reg [ADDR_BITS-1:0] rd_addr;
    output reg rd_match;

    integer i;
    reg [WIDTH-1:0] mem [0:DEPTH-1];

    always @(posedge clk) begin
        rd_addr <= 'bx;
        rd_match <= 0;
        for (i = 0; i < DEPTH; i = i+1)
            if (mem[i] == rd_data) begin
                rd_addr <= i;
                rd_match <= 1;
            end
        if (wr_enable)
            mem[wr_addr] <= wr_data;
    end
endmodule
```

Currently unsupported Verilog-2005 language features

- Tri-state logic
- The wor/wand wire types (maybe for 0.5)
- Latched logic (is synthesized as logic with feedback loops)
- Some non-synthesizable features that should be ignored in synthesis are not supported by the parser and cause a parser error (file a bug report if you encounter this problem)

Verification of Yosys

Continuously checking the correctness of Yosys and making sure that new features do not break old ones is a high priority in Yosys.

Two external test suites have been built for Yosys: VlogHammer and yosys-bigsim (see next slides)

In addition to that, yosys comes with ≈200 test cases used in "`make test`".

A debug build of Yosys also contains a lot of asserts and checks the integrity of the internal state after each command.

Verification of Yosys – VlogHammer

VlogHammer is a Verilog regression test suite developed to test the different subsystems in Yosys by comparing them to each other and to the output created by some other tools (Xilinx Vivado, Xilinx XST, Altera Quartus II, ...).

Yosys Subsystems tested: Verilog frontend, const folding, const eval, technology mapping, simulation models, SAT models.

Thousands of auto-generated test cases containing code such as:

```
assign y9 = $signed(((+$signed((^(6'd2 ** a2))))<$unsigned($unsigned(((+a3))))));
assign y10 = (-((+((+{2{(~^p13)}}))))^~(!{{b5,b1,a0},(a1&p12),(a4+a3)}));
assign y11 = (~&(-{(-3'sd3),($unsigned($signed($unsigned({p0,b4,b1}))))}));
```

Some bugs in Yosys where found and fixed thanks to VlogHammer. Over 50 bugs in the other tools used as external reference where found and reported so far.

Verification of Yosys – yosys-bigsim

yosys-bigsim is a collection of real-world open-source Verilog designs and test benches. yosys-bigsim compares the testbench outputs of simulations of the original Verilog code and synthesis results.

The following designs are included in yosys-bigsim (excerpt):

- `openmsp430` – an MSP430 compatible 16 bit CPU
- `aes_5cycle_2stage` – an AES encryption core
- `softusb_navre` – an AVR compatible 8 bit CPU
- `amber23` – an ARMv2 compatible 32 bit CPU
- `lm32` – another 32 bit CPU from Lattice Semiconductor
- `verilog-pong` – a hardware pong game with VGA output
- `elliptic_curve_group` – ECG point-add and point-scalar-mul core
- `reed_solomon_decoder` – a Reed-Solomon Error Correction Decoder

Benefits of Open Source HDL Synthesis

- Cost (also applies to “free as in free beer” solutions)
- Availability and Reproducibility
- Framework- and all-in-one-aspects
- Educational Tool

Yosys is open source under the ISC license.

Benefits of Open Source HDL Synthesis – 1/3

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

  Today the cost for a mask set in 180 nm 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.

Benefits of Open Source HDL Synthesis – 2/3

- 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.

Benefits of Open Source HDL Synthesis – 3/3

- 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.

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[3]

[3] Not limited to synthesis but also formal verification, reverse engineering, ...

Projects (that I know of) using Yosys – (1/2)

- Ongoing PhD project on coarse grain synthesis

  Johann Glaser and C. Wolf. Methodology and Example-Driven Interconnect Synthesis for Designing Heterogeneous Coarse-Grain Reconfigurable Architectures. In Jan Haase, editor, *Models, Methods, and Tools for Complex Chip Design. Lecture Notes in Electrical Engineering. Volume 265, 2014, pp 201-221. Springer, 2013.*

- I know several people that use Yosys simply as Verilog frontend for other flows (using either the BLIF and BTOR backends).

- I know some analog chip designers that use Yosys for small digital control logic because it is simpler than setting up a commercial flow.

Projects (that I know of) using Yosys – (2/2)

- Efabless
  - Not much information on the website (`http://efabless.com`) yet.
  - Very cheap 180nm prototyping process (partnering with various fabs)
  - A semiconductor company, NOT an EDA company
  - Web-based design environment
  - HDL Synthesis using Yosys
  - Custom place&route tool

  - efabless is building an Open Source IC as reference design.
    (to be announced soon: `http://www.openic.io`)

Supported Platforms

- Main development OS: Kubuntu 14.04
- There is a PPA for ubuntu (not maintained by me)
- Any current Debian-based system should work out of the box
- When building on other Linux distributions:
  - Needs compiler with some C++11 support
  - See README file for build instructions
  - Post to the subreddit if you get stuck
- Ported to OS X (Darwin) and OpenBSD
- Native win32 build with VisualStudio
- Cross win32 build with MXE

Other Open Source Tools

- Icarus Verilog

  Verilog Simulation (and also a good syntax checker)

  `http://iverilog.icarus.com/`

- Qflow (incl. TimberWolf, qrouter and Magic)

  A complete ASIC synthesis flow, using Yosys and ABC

  `http://opencircuitdesign.com/qflow/`

- ABC

  Logic optimization, technology mapping, and more

  `http://www.eecs.berkeley.edu/~alanmi/abc/`

Yosys needs you

. . . as an active user:

- Use Yosys for on your own projects
- .. even if you are not using it as final synthesis tool
- Join the discussion on the Subreddit
- Report bugs and send in feature requests

. . . as a developer:

- Use Yosys as environment for your (research) work
- .. you might also want to look into ABC for logic-level stuff
- Fork the project on github or create loadable plugins
- We need a VHDL frontend or a good VHDL-to-Verilog converter

Documentation, Downloads, Contacts

- Website:

  `https://yosyshq.net/yosys/`

- Manual, Command Reference, Application Notes:

  `https://yosyshq.net/yosys/documentation.html`

- Instead of a mailing list we have a SubReddit:

  `http://www.reddit.com/r/yosys/`

- Direct link to the source code:

  `https://github.com/YosysHQ/yosys`

Summary

- Yosys is a powerful tool and framework for Verilog synthesis.
- It uses a command-based interface and can be controlled by scripts.
- By combining existing commands and implementing new commands Yosys can be used in a wide range of application far beyond simple synthesis.

Questions?

`https://yosyshq.net/yosys/`

Section 2

Yosys by example – Synthesis

Typical Phases of a Synthesis Flow

- Reading and elaborating the design
- Higher-level synthesis and optimization
  - Converting `always`-blocks to logic and registers
  - Perform coarse-grain optimizations (resource sharing, const folding, ...)
  - Handling of memories and other coarse-grain blocks
  - Extracting and optimizing finite state machines
- Convert remaining logic to bit-level logic functions
- Perform optimizations on bit-level logic functions
- Map bit-level logic gates and registers to cell library
- Write results to output file

Reading the design

```
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
```

Design elaboration

During design elaboration Yosys figures out how the modules are hierarchically connected. It also re-runs the AST parts of the Verilog frontend to create all needed variations of parametric modules.

```
# simplest form. at least this version should be used after reading all input files
#
hierarchy

# recommended form. fails if parts of the design hierarchy are missing, removes
# everything that is unreachable from the top module, and marks the top module.
#
hierarchy -check -top top_module
```

The proc command

The Verilog frontend converts always-blocks to RTL netlists for the expressions and "processes" for the control- and memory elements.

The proc command transforms this "processes" to netlists of RTL multiplexer and register cells.

The proc command is actually a macro-command that calls the following other commands:

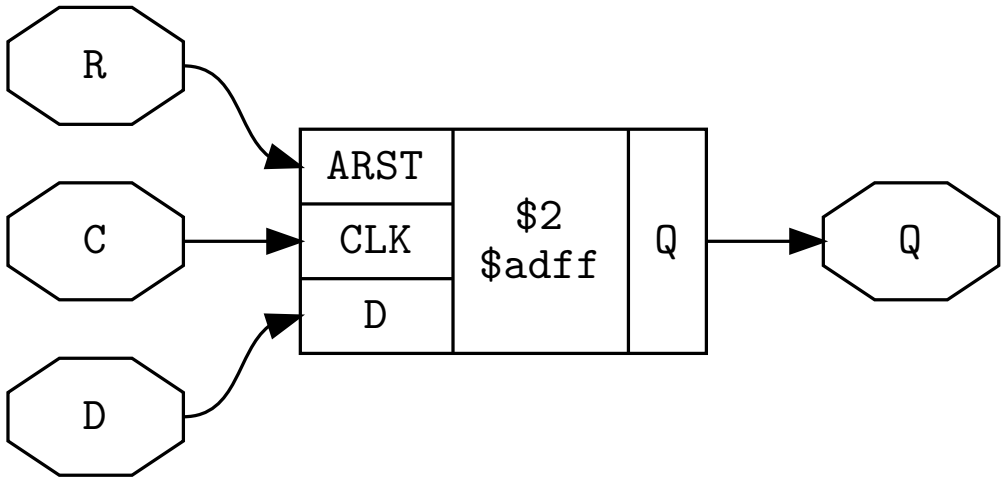
```
proc_clean      # remove empty branches and processes
proc_rmdead     # remove unreachable branches
proc_init       # special handling of "initial" blocks
proc_arst       # identify modeling of async resets
proc_mux        # convert decision trees to multiplexer networks
proc_dff        # extract registers from processes
proc_clean      # if all went fine, this should remove all the processes
```

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

The proc command – Example 1/3

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

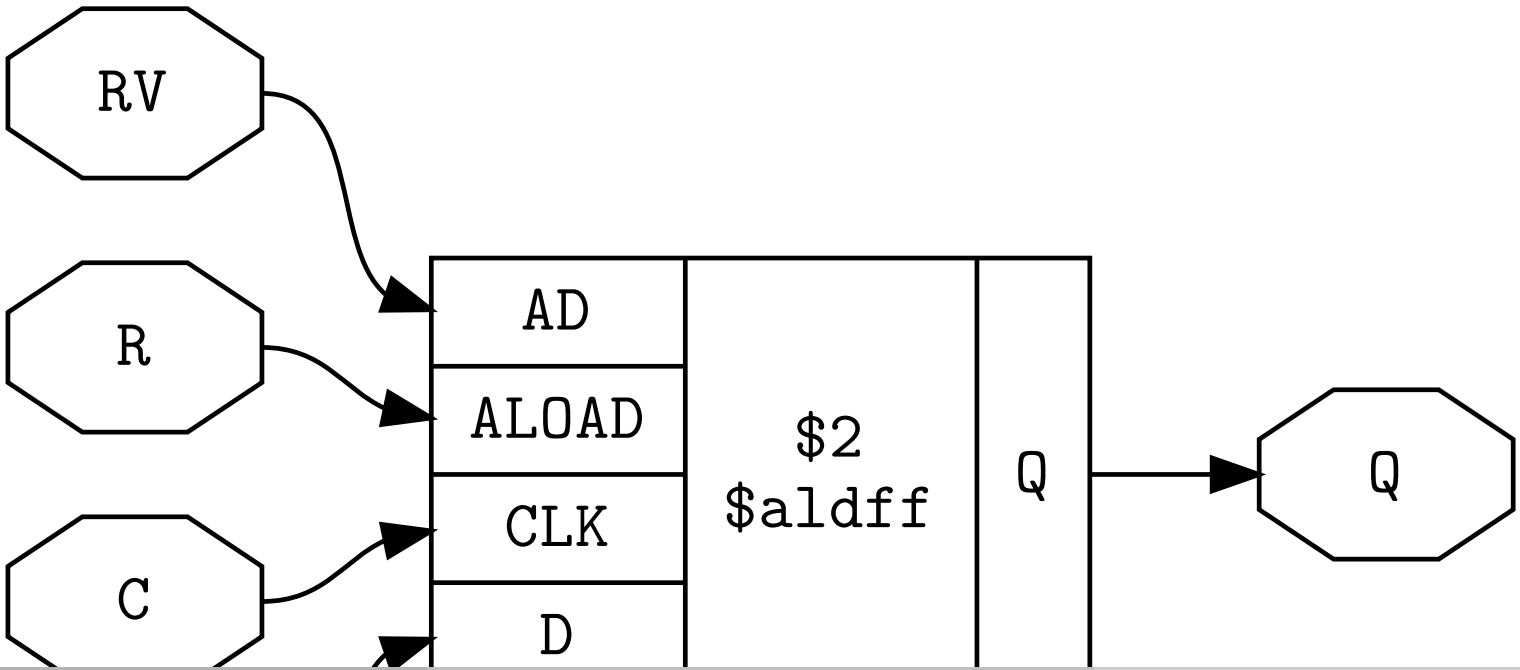
```
read_verilog proc_01.v
hierarchy -check -top test
proc;;
```



The proc command – Example 2/3

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

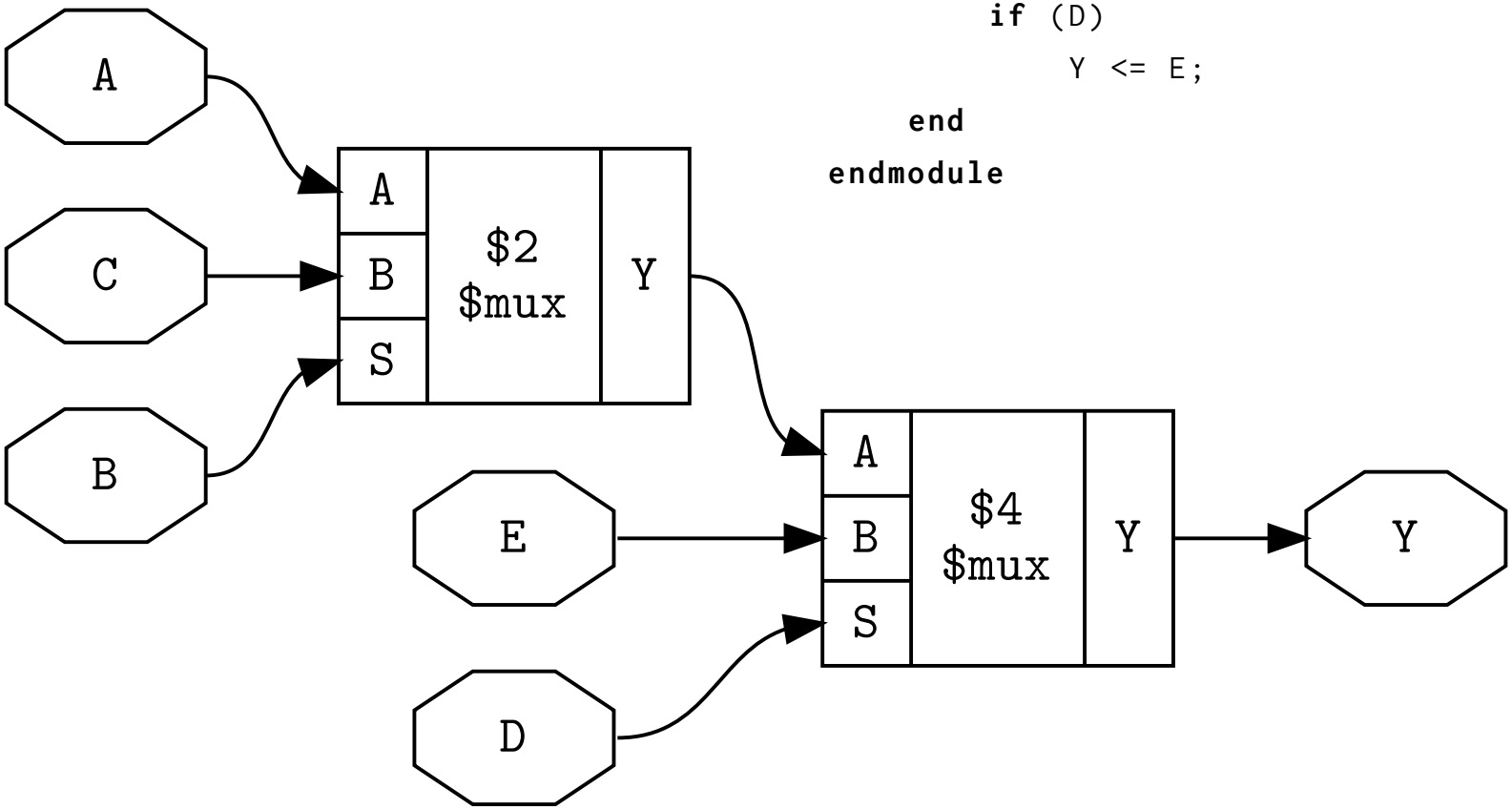
```
read_verilog proc_02.v
hierarchy -check -top test
proc;;
```



The proc command – Example 3/3

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

```
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
```



The opt command

The opt command implements a series of simple optimizations. It also is a macro command that calls other commands:

```
opt_expr                # const folding and simple expression rewriting
opt_merge -nomux        # merging identical cells

do
    opt_muxtree         # remove never-active branches from multiplexer tree
    opt_reduce          # consolidate trees of boolean ops to reduce functions
    opt_merge           # merging identical cells
    opt_rmdff           # remove/simplify registers with constant inputs
    opt_clean           # remove unused objects (cells, wires) from design
    opt_expr            # const folding and simple expression rewriting
while [changed design]
```

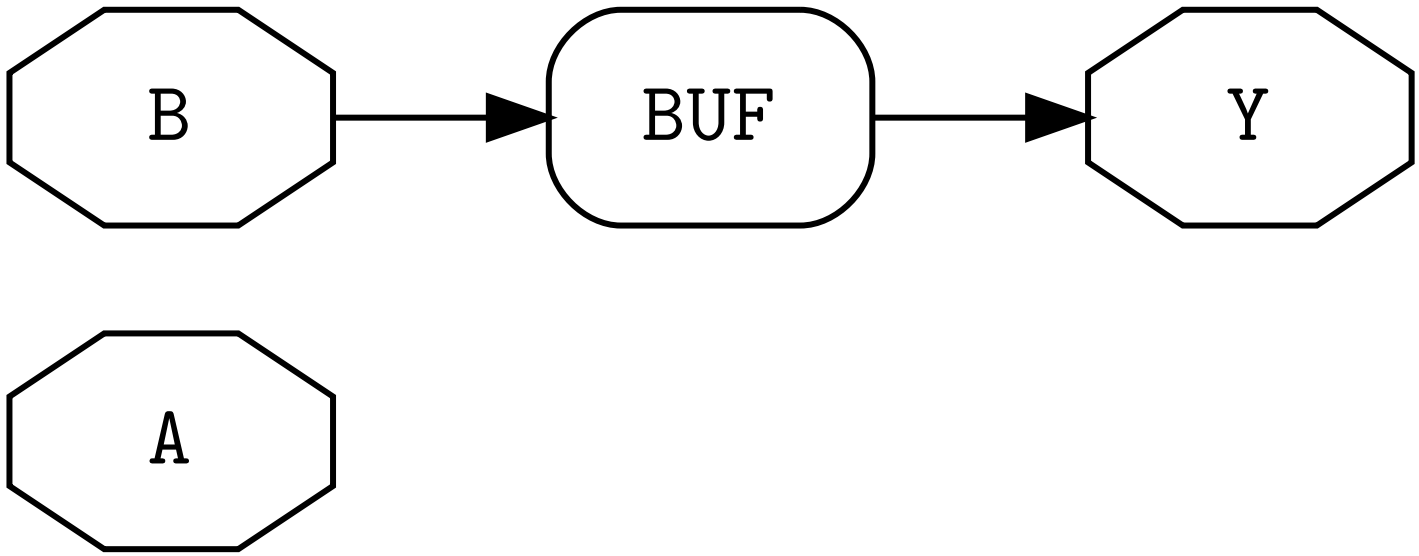
The command clean can be used as alias for opt_clean. And ;; can be used as shortcut for clean. For example:

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

The opt command – Example 1/4

```
read_verilog opt_01.v
hierarchy -check -top test
opt
```

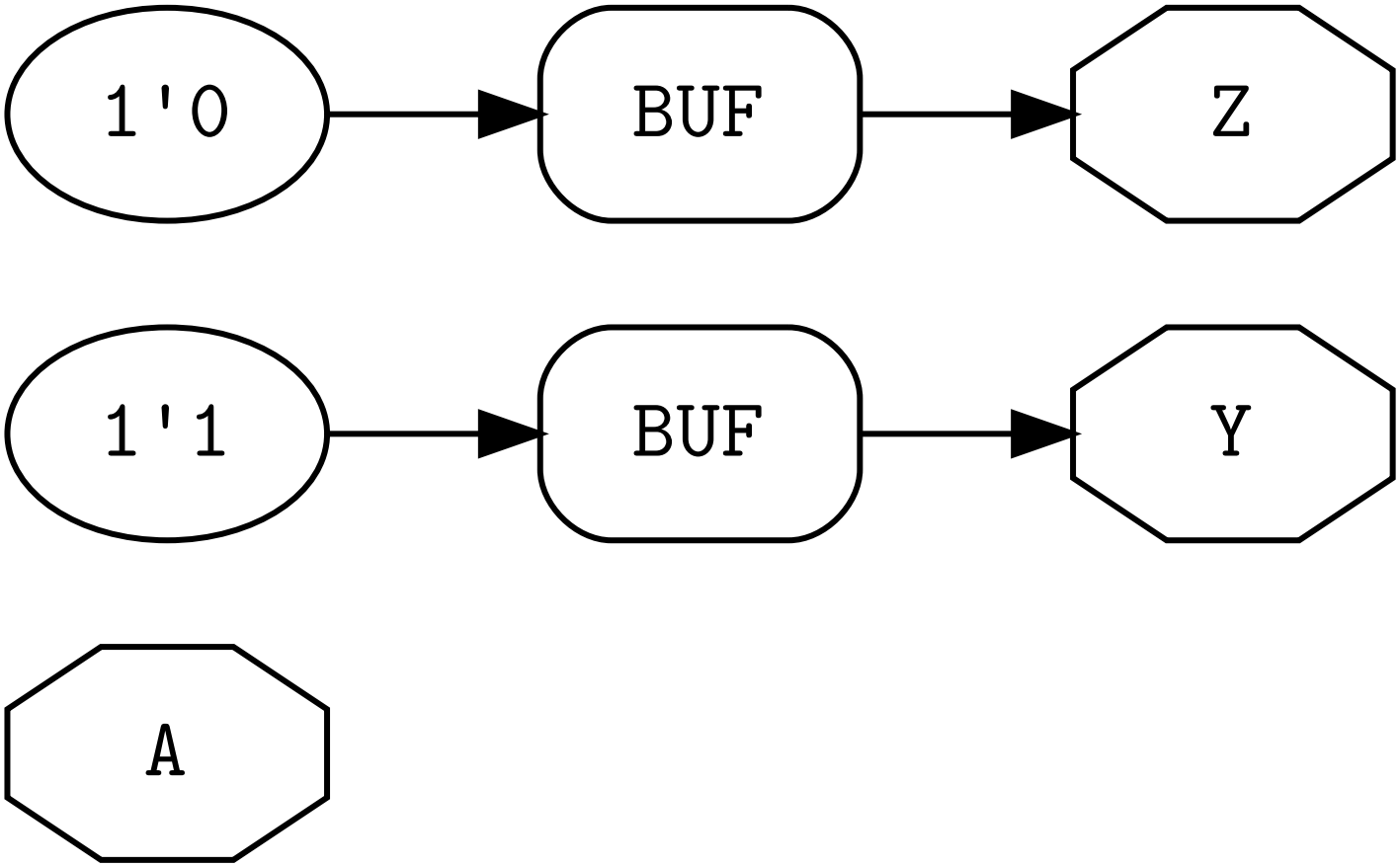
```
module test(input A, B, output Y);
assign Y = A ? A ? B : 1'b1 : B;
endmodule
```



The opt command – Example 2/4

```
read_verilog opt_02.v
hierarchy -check -top test
opt
```

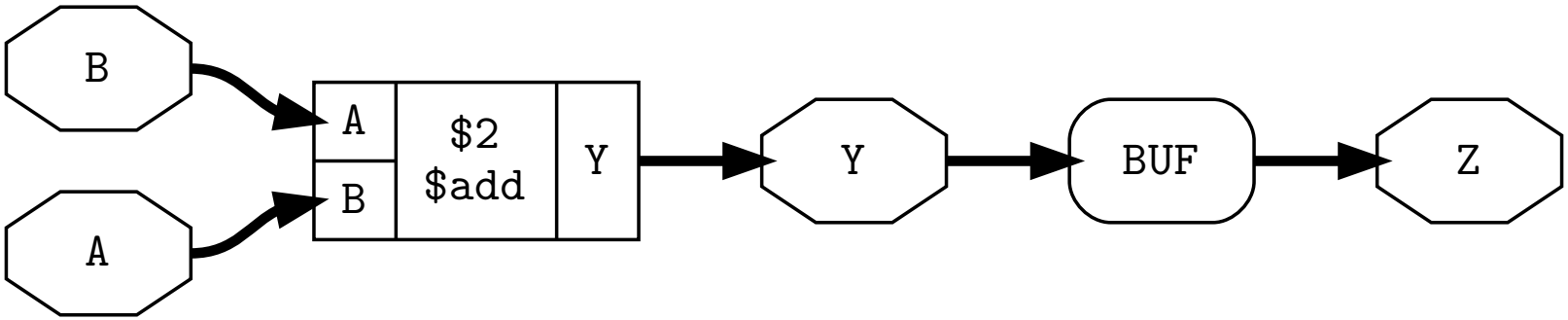
```
module test(input A, output Y, Z);
assign Y = A == A, Z = A != A;
endmodule
```



The opt command – Example 3/4

```
read_verilog opt_03.v
hierarchy -check -top test
opt
```

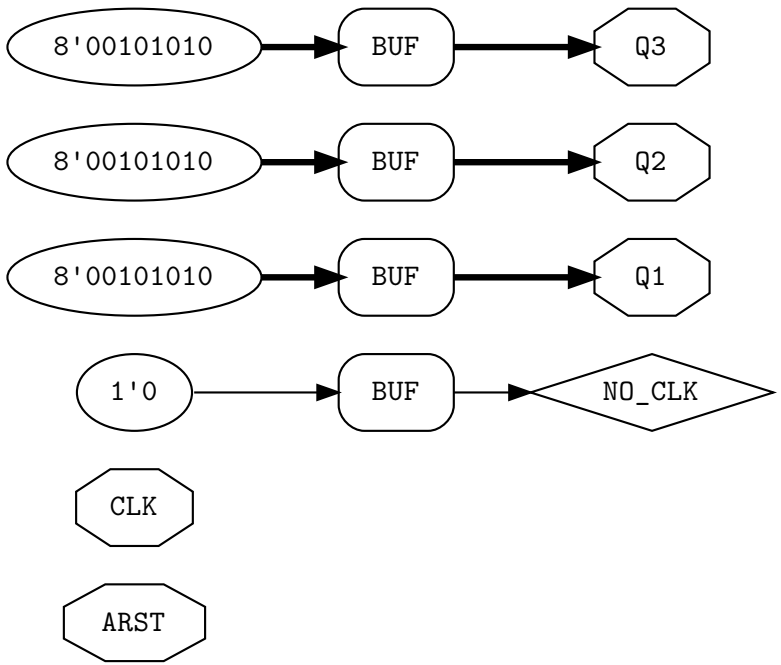
```
module test(input  [3:0] A, B,
            output [3:0] Y, Z);
assign Y = A + B, Z = B + A;
endmodule
```



The opt command – Example 4/4

```
module test(input CLK, ARST,
            output [7:0] Q1, Q2, Q3);

wire NO_CLK = 0;

always @(posedge CLK, posedge ARST)
        if (ARST)
                Q1 <= 42;

always @(posedge NO_CLK, posedge ARST)
        if (ARST)
                Q2 <= 42;
        else
                Q2 <= 23;

always @(posedge CLK)
        Q3 <= 42;

endmodule
```

```
read_verilog opt_04.v
hierarchy -check -top test
proc; opt
```



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.

The designs in `yosys-bigsim` are a good playground for experimenting with the effects of calling `opt` in various places of the flow.

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

The `clean` command 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 `;;` is a good idea in every synthesis script.

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` transforms memories to an implementation. Per default that is logic for address decoders and registers. It also is a macro command that calls other commands:

```
# this merges registers into the memory read- and write cells.
memory_dff

# this collects all read and write cells for a memory and transforms them
# into one multi-port memory cell.
memory_collect

# this takes the multi-port memory cell and transforms it to address decoder
# logic and registers. This step is skipped if "memory" is called with -nomap.
memory_map
```

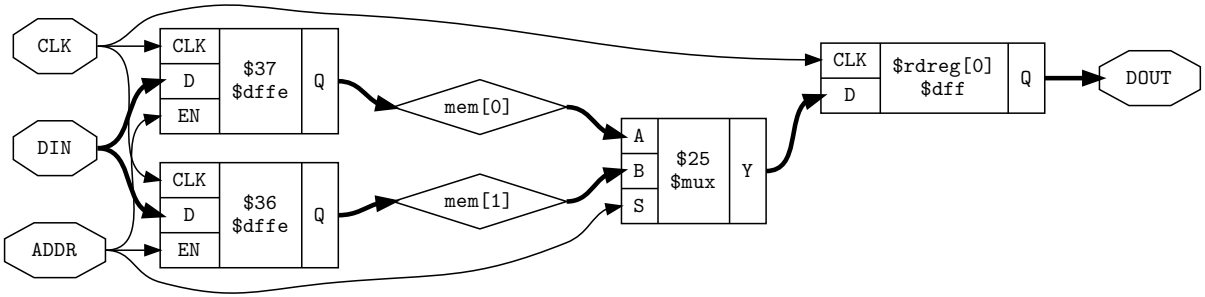
Usually it is preferred to use architecture-specific RAM resources for memory. For example:

```
memory -nomap; techmap -map my_memory_map.v; memory_map
```

The memory command – Example 1/2

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

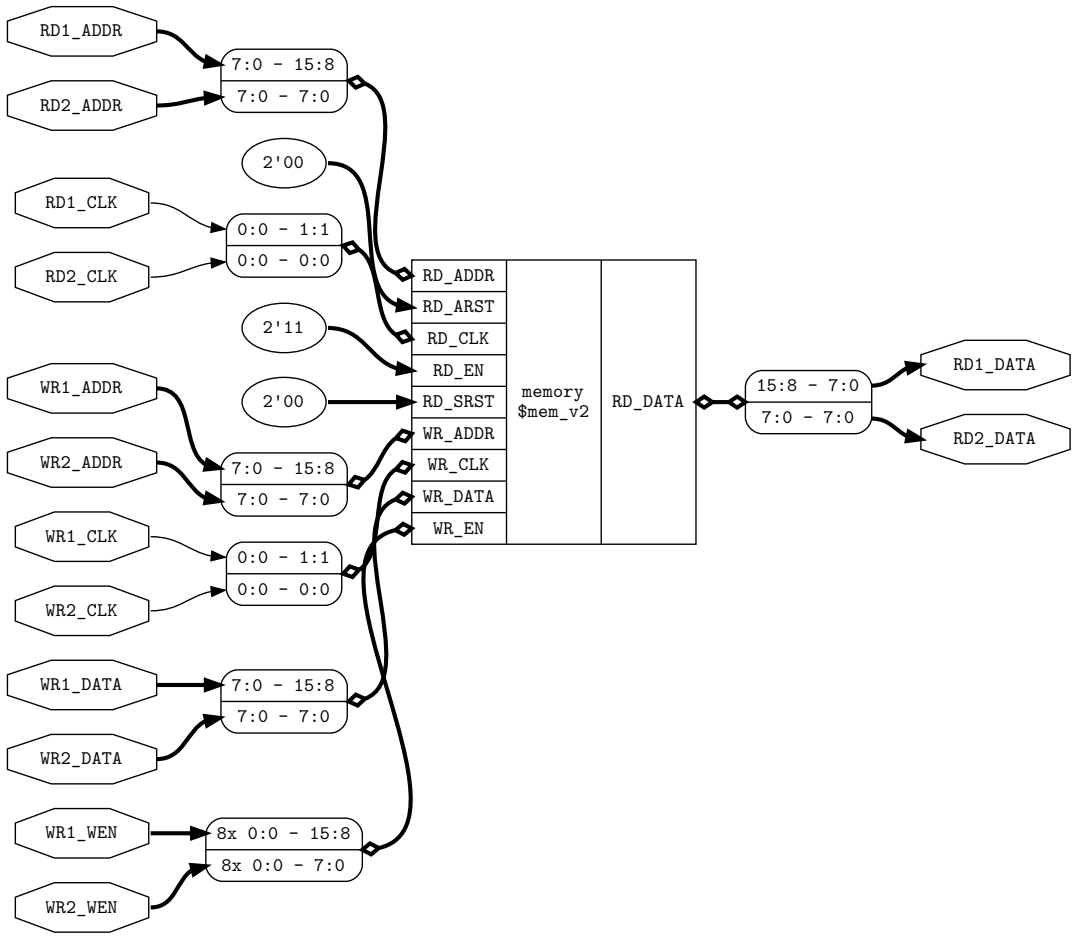
```
module test(input       CLK, ADDR,
            input       [7:0] DIN,
            output reg  [7:0] DOUT);
    reg [7:0] mem [0:1];
    always @(posedge CLK) begin
        mem[ADDR] <= DIN;
        DOUT <= mem[ADDR];
    end
endmodule
```



The memory command – Example 2/2

```
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
);

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
```

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



The `fsm` command

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:

```
fsm_detect          # unless got option -nodetect
fsm_extract

fsm_opt
clean
fsm_opt

fsm_expand          # if got option -expand
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
```

The `fsm` command – details

Some details on the most important commands from the `fsm_*` group:

The `fsm_detect` command identifies FSM state registers and marks them with the `(* fsm_encoding = "auto" *)` attribute, if they do not have the `fsm_encoding` set already. Mark registers with `(* fsm_encoding = "none" *)` to disable FSM optimization for a register.

The `fsm_extract` command replaces the entire FSM (logic and state registers) with a `$fsm` cell.

The commands `fsm_opt` and `fsm_recode` can be used to optimize the FSM.

Finally the `fsm_map` command can be used to convert the (optimized) `$fsm` cell back to logic and registers.

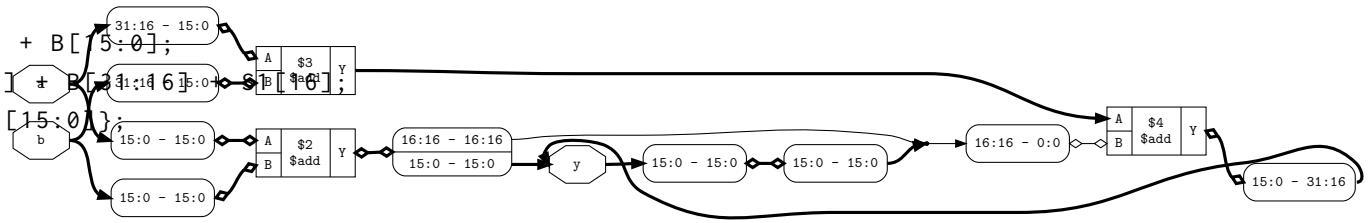
The techmap command

The techmap command replaces cells with implementations given as verilog source. For example implementing a 32 bit adder using 16 bit adders:

```
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
```

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

```
read_verilog techmap_01.v
hierarchy -check -top test
techmap -map techmap_01_map.v;;
```



The techmap command – stdcell mapping

When techmap is used without a map file, it uses a built-in map file to map all RTL cell types to a generic library of built-in logic gates and registers.

The built-in logic gate types are:

```
$_NOT_ $_AND_ $_OR_ $_XOR_ $_MUX_
```

The register types are:

```
$_SR_NN_ $_SR_NP_ $_SR_PN_ $_SR_PP_
$_DFF_N_ $_DFF_P_
$_DFF_NN0_ $_DFF_NN1_ $_DFF_NP0_ $_DFF_NP1_
$_DFF_PN0_ $_DFF_PN1_ $_DFF_PP0_ $_DFF_PP1_
$_DFFSR_NNN_ $_DFFSR_NNP_ $_DFFSR_NPN_ $_DFFSR_NPP_
$_DFFSR_PNN_ $_DFFSR_PNP_ $_DFFSR_PPN_ $_DFFSR_PPP_
$_DLATCH_N_ $_DLATCH_P_
```

The abc command

The abc command provides an interface to ABC[4], an open source tool for low-level logic synthesis.

The abc command processes a netlist of internal gate types and can perform:

- logic minimization (optimization)
- mapping of logic to standard cell library (liberty format)
- mapping of logic to k-LUTs (for FPGA synthesis)

Optionally abc can process registers from one clock domain and perform sequential optimization (such as register balancing).

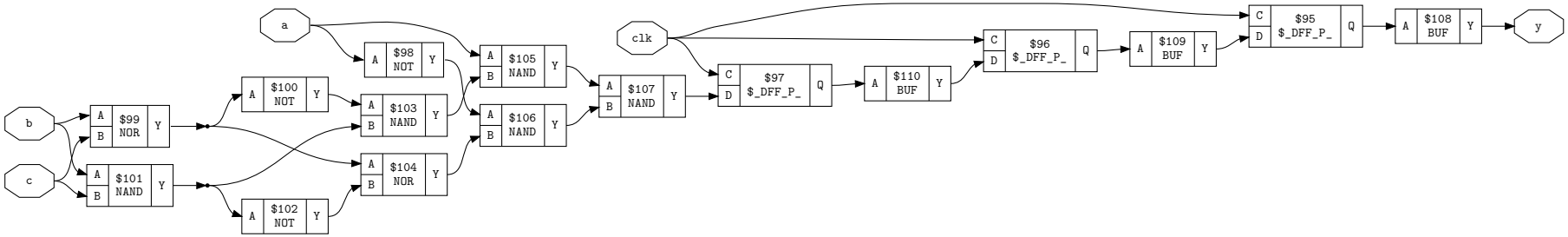
ABC is also controlled using scripts. An ABC script can be specified to use more advanced ABC features. It is also possible to write the design with `write_blif` and load the output file into ABC outside of Yosys.

[4] `http://www.eecs.berkeley.edu/~alanmi/abc/`

The abc command – Example

```
module test(input clk, a, b, c,
            output reg y);

        reg [2:0] q1, q2;
        always @(posedge clk) begin
                q1 <= { a, b, c };
                q2 <= q1;
                y <= ^q2;
        end
endmodule
```

```
read_verilog abc_01.v
read_verilog -lib abc_01_cells.v
hierarchy -check -top test
proc; opt; techmap
abc -dff -liberty abc_01_cells.lib;;
```



Other special-purpose mapping commands

`dfflibmap`

This command maps the internal register cell types to the register types described in a liberty file.

`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 this cells to the design.

Example Synthesis Script

```
# read and elaborate design
read_verilog cpu_top.v cpu_ctrl.v cpu_regs.v
read_verilog -D WITH_MULT cpu_alu.v
hierarchy -check -top cpu_top

# high-level synthesis
proc; opt; fsm;; memory -nomap; opt

# substitute block rams
techmap -map map_rams.v

# map remaining memories
memory_map

# low-level synthesis
techmap; opt; flatten;; abc -lut6
techmap -map map_xl_cells.v

# add clock buffers
select -set xl_clocks t:FDRE %x:+FDRE[C] t:FDRE %d
iopadmap -inpad BUFGP O:I @xl_clocks

# add io buffers
select -set xl_nonclocks w:* t:BUFGP %x:+BUFGP[I] %d
iopadmap -outpad OBUF I:O -inpad IBUF O:I @xl_nonclocks

# write synthesis results
write_edif synth.edif
```

Teaser / Outlook

The weird `select` expressions at the end of this script are discussed in the next part (Section 3, “Advanced Synthesis”) of this presentation.

Summary

- Yosys provides commands for each phase of the synthesis.
- Each command solves a (more or less) simple problem.
- Complex commands are often only front-ends to simple commands.
- `proc; opt; fsm; opt; memory; opt; techmap; opt; abc;;`

Questions?

`https://yosyshq.net/yosys/`

Section 3

Yosys by example – Advanced Synthesis

Overview

This section contains 4 subsections:

- Using selections
- Advanced uses of techmap
- Coarse-grain synthesis
- Automatic design changes

Subsection 1

Using selections

of Section 3

Yosys by example – Advanced Synthesis

Simple selections

Most Yosys commands make use of the "selection framework" of Yosys. It can be used to apply commands only to part of the design. For example:

```
delete                  # will delete the whole design, but

delete foobar           # will only delete the module foobar.
```

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
select -clear           # reset selection (select whole design)
```

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 for interactive design investigation.

Selecting by object property or type

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

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

A complete list of this pattern expressions can be found in the command reference to the `select` command.

Combining selection

When more than one selection expression is used in one statement, then they are pushed on a stack. The final elements on the stack are combined into a union:

```
select t:$dff r:WIDTH>1     # all cells of type $dff and/or with a parameter WIDTH > 1
```

Special %-commands can be used to combine the elements on the stack:

```
select t:$dff r:WIDTH>1 %i  # all cells of type $dff *AND* with a parameter WIDTH > 1
```

Examples for %-codes (see `help select` for full list)

%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

Expanding selections

Selections of cells and wires can be expanded along connections using %-codes for selecting input cones (%ci), output cones (%co), or both (%x).

```
# select all wires that are inputs to $add cells
select t:$add %ci w:* %i
```

Additional constraints such as port names can be specified.

```
# select all wires that connect a "Q" output with a "D" input
select c:* %co:+[Q] w:* %i c:* %ci:+[D] w:* %i %i

# select the multiplexer tree that drives the signal 'state'
select state %ci*:+$mux,$pmux[A,B,Y]
```

See `help select` for full documentation of this expressions.

Incremental selection

Sometimes a selection can most easily be described by a series of add/delete operations. The commands `select -add` and `select -del` respectively add or remove objects from the current selection instead of overwriting it.

```
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 mor 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
```

Creating selection variables

Selections can be stored under a name with the `select -set <name>` command. The stored selections can be used in later select expressions using the syntax `@<name>`.

```
select -set cone_a state_a %ci*:-$dff  # set @cone_a to the input cone of state_a
select -set cone_b state_b %ci*:-$dff  # set @cone_b to the input cone of state_b
select @cone_a @cone_b %i              # select the objects that are in both cones
```

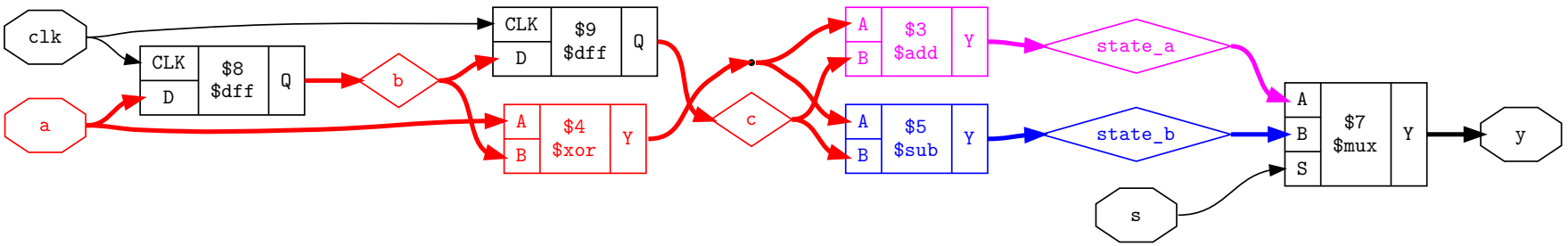
Remember that select expressions can also be used directly as arguments to most commands. Some commands also except a single select argument to some options. In those cases selection variables must be used to capture more complex selections.

```
dump @cone_a @cone_b

select -set cone_ab @cone_a @cone_b %i
show -color red @cone_ab -color magenta @cone_a -color blue @cone_b
```

Creating selection variables – Example

```
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
```

```
read_verilog select.v
hierarchy -check -top test
proc; opt
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 pdf -notitle \
     -color red @cone_ab -color magenta @cone_a \
     -color blue @cone_b
```



Subsection 2

Advanced uses of techmap

of Section 3

Yosys by example – Advanced Synthesis

Introduction to techmap

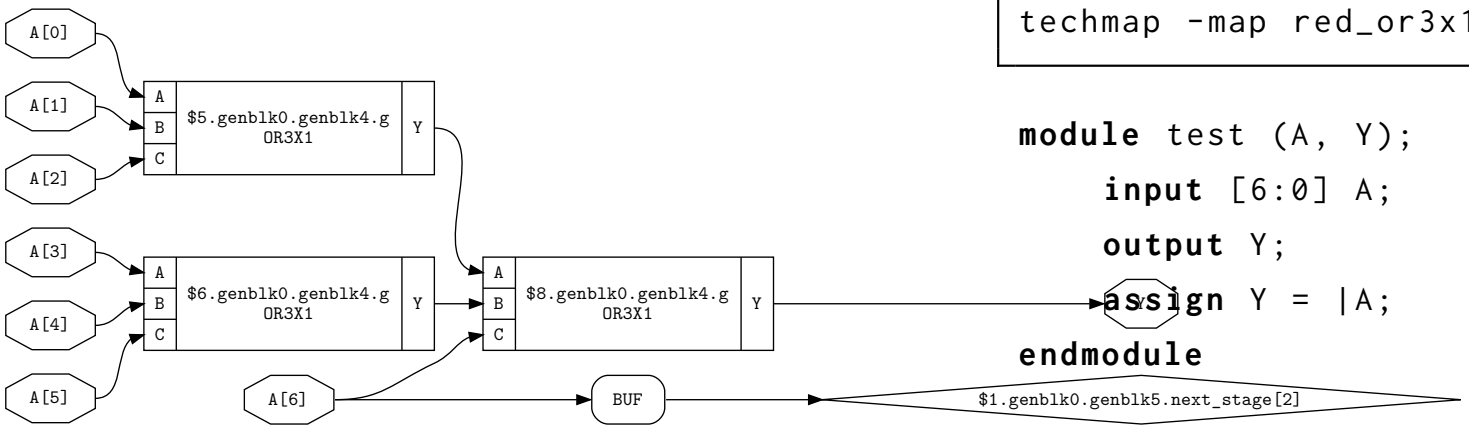
- The `techmap` command replaces cells in the design with implementations given as Verilog code (called "map files"). It can replace Yosys' internal cell types (such as `$or`) as well as user-defined cell types.
- Verilog parameters are used extensively to customize the internal cell types.
- Additional 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.

Introduction to techmap – Example 1/2

To map the Verilog OR-reduction operator to 3-input OR gates:

```
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
        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
```

Introduction to techmap – Example 2/2




```
techmap -map red_or3x1_map.v;;
```

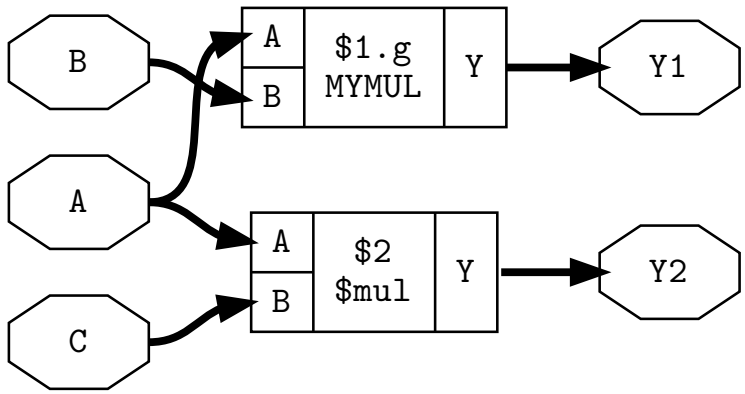
```
module test (A, Y);
    input [6:0] A;
    output Y;
    assign Y = |A;
endmodule
```

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.)

Conditional techmap – Example

```
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;

    MYMUL #( .WIDTH(Y_WIDTH) ) g ( .A(A), .B(B), .Y(Y) );
endmodule
```




```
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
```

```
read_verilog sym_mul_test.v
hierarchy -check -top test

techmap -map sym_mul_map.v;;
```

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.

PROTIP: Commands such as `shell`, `show -pause`, and `dump` can be use in the `_TECHMAP_DO_*` scripts for debugging map modules.

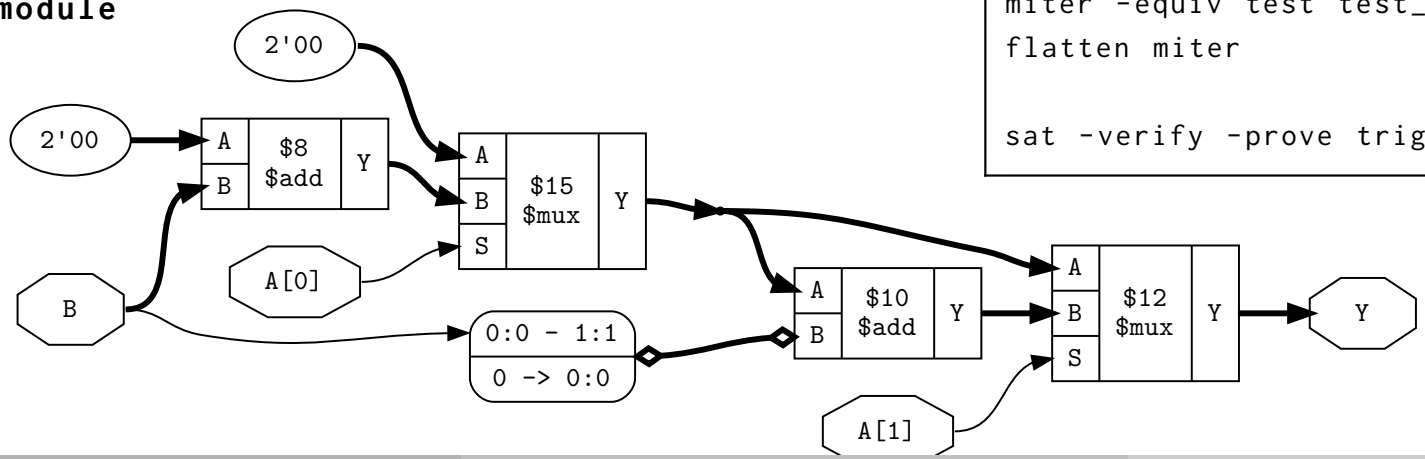
Scripting in map modules – Example

```
module MYMUL(A, B, Y);
    parameter WIDTH = 1;
    input [WIDTH-1:0] A, B;
    output reg [WIDTH-1:0] Y;

    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
```

```
module test(A, B, Y);
    input  [1:0] A, B;
    output [1:0] Y = A * B;
endmodule
```

```
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
```



Handling constant inputs

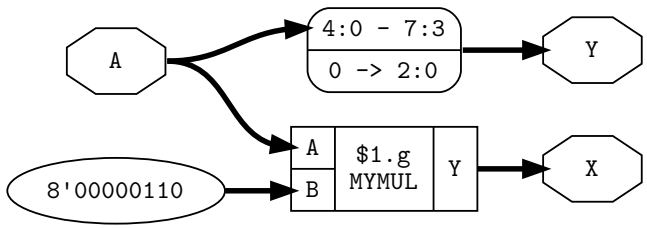
- The special parameters `_TECHMAP_CONSTMSK_`<*port-name*>`_` and `_TECHMAP_CONSTVAL_`<*port-name*>`_` can be used to handle constant input values to cells.
- The former contains 1-bits for all constant input bits on the port.
- The latter contains the constant bits or undef (x) for non-constant bits.
- Example use-cases:
  - Converting arithmetic (for example multiply to shift)
  - Identify constant addresses or enable bits in memory interfaces.

Handling constant inputs – Example

```
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
            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
```

```
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
```

```
read_verilog mulshift_test.v
hierarchy -check -top test

techmap -map sym_mul_map.v \
        -map mulshift_map.v;;
```



Handling shorted inputs

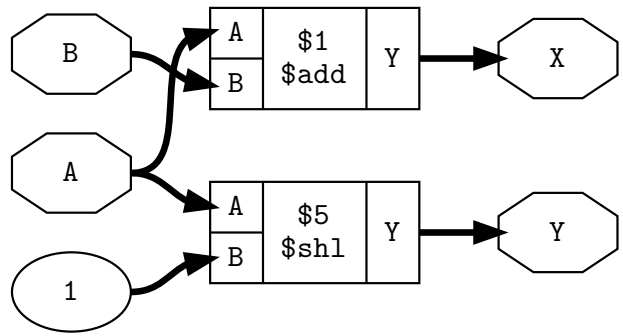
- The special parameters `_TECHMAP_BITS_CONNMAP_` and `_TECHMAP_CONNMAP_`<*port-name*>`_` can be used to handle shorted inputs.
- Each bit of the port correlates to an `_TECHMAP_BITS_CONNMAP_` bits wide number in `_TECHMAP_CONNMAP_`<*port-name*>`_`.
- Each unique signal bit is assigned its own number. 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.

Handling shorted inputs – Example

```
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;

  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
```

```
module test (A, B, X, Y);
input [7:0] A, B;
output [7:0] X = A + B;
output [7:0] Y = A + A;
endmodule
```

```
read_verilog addshift_test.v
hierarchy -check -top test

techmap -map addshift_map.v;;
```



Notes on using techmap

- Don't use positional cell parameters in map modules.
- Don't try to implement basic logic optimization with techmap.
  (So the OR-reduce using OR3X1 cells map was actually a bad example.)
- 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 this 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.

Subsection 3

Coarse-grain synthesis

of Section 3

Yosys by example – Advanced Synthesis

Intro to coarse-grain synthesis

In coarse-grain synthesis the target architecture has cells of the same complexity or larger complexity than the internal RTL representation. For example:

```
wire [15:0] a, b;
wire [31:0] c, y;
assign y = a * b + c;
```

This circuit contains two cells in the RTL representation: one multiplier and one adder. In some architectures this circuit can be implemented using a single circuit element, for example an FPGA DSP core. Coarse grain synthesis is this mapping of groups of circuit elements to larger components.

Fine-grain synthesis would be matching the circuit elements to smaller components, such as LUTs, gates, or half- and full-adders.

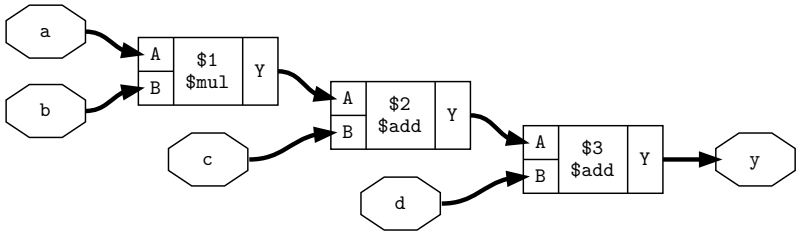
The extract pass

- 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.

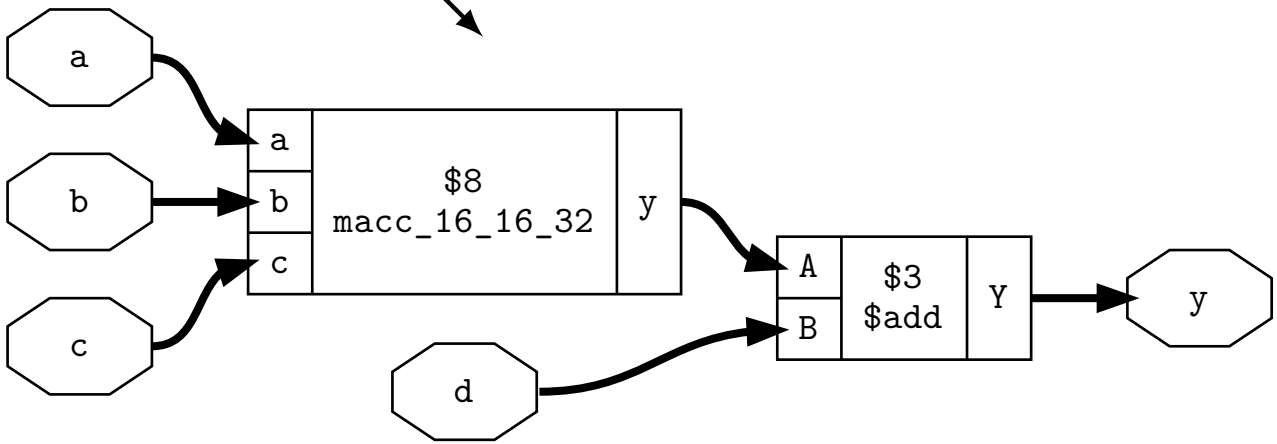
The extract pass – Example 1/2

```
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
```

```
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
```

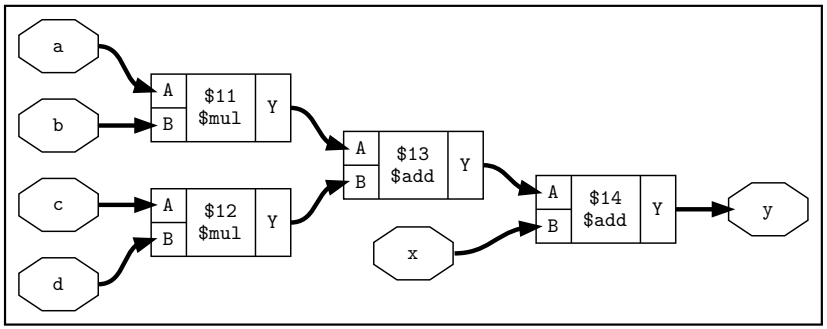


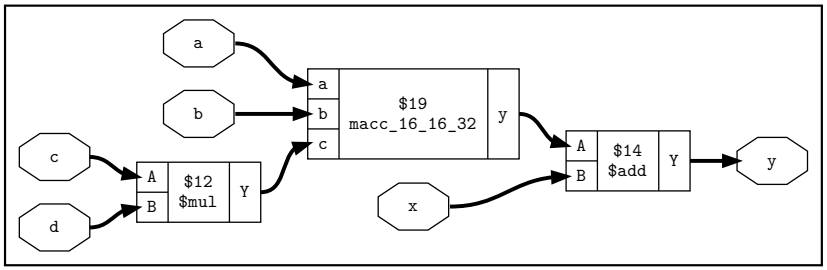

```
read_verilog macc_simple_test.v
hierarchy -check -top test

extract -map macc_simple_xmap.v;;
```



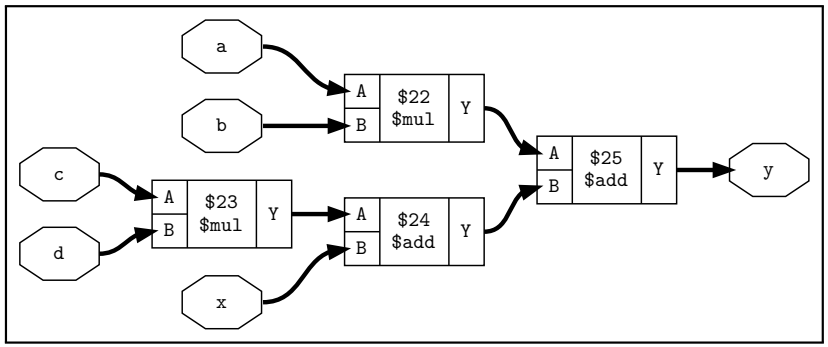
The extract pass – Example 2/2

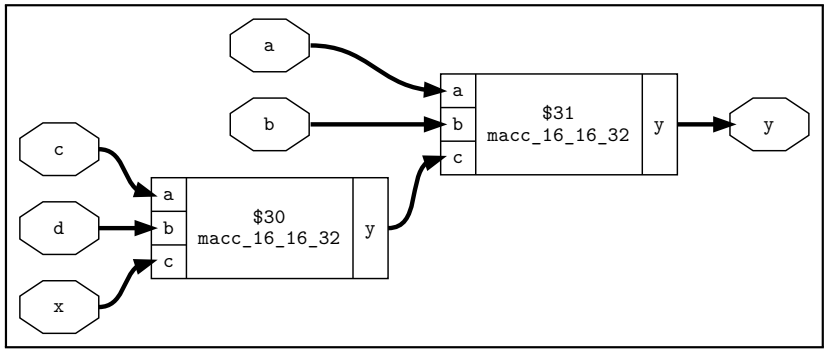
```
module test(a, b, c, d, x, y);
input [15:0] a, b, c, d;
input [31:0] x;
output [31:0] y;
assign y = a*b + c*d + x;
endmodule
```






```
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
```





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`.

The following sides detail 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).

Example: DSP48_MACC – 1/13

Preconditioning: `macc_xilinx_swap_map.v`

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

```
(* 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;

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
```

Example: DSP48_MACC – 2/13

Wrapping multipliers: `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

\$__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
```

Example: DSP48_MACC – 3/13

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

endmodule
```

Example: DSP48_MACC – 4/13

Extract: `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.

Example: DSP48_MACC – 5/13

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

endmodule
```

Example: DSP48_MACC – 6/13

Unwrapping adders: `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
```

Example: DSP48_MACC – 7/13

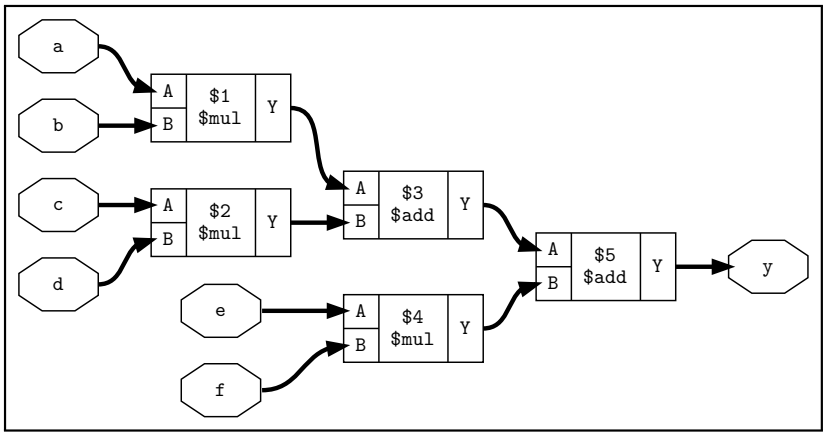
test1

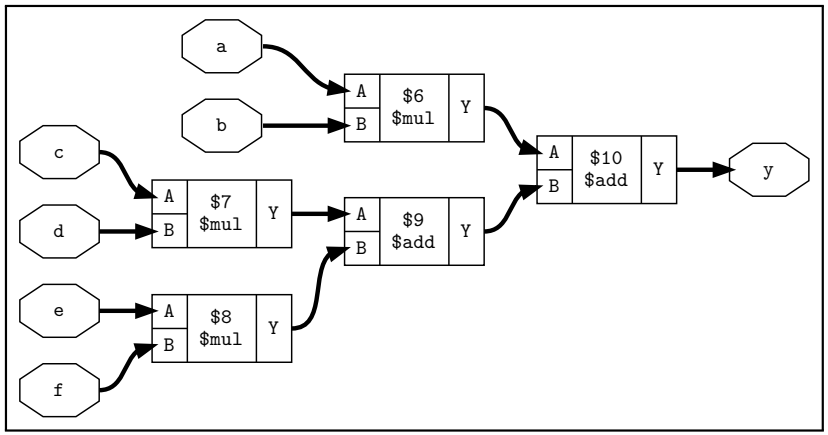
```
module test1(a, b, c, d, e, f, y);
    input [19:0] a, b, c;
    input [15:0] d, e, f;
    output [41:0] y;
    assign y = a*b + c*d + e*f;
endmodule
```

test2

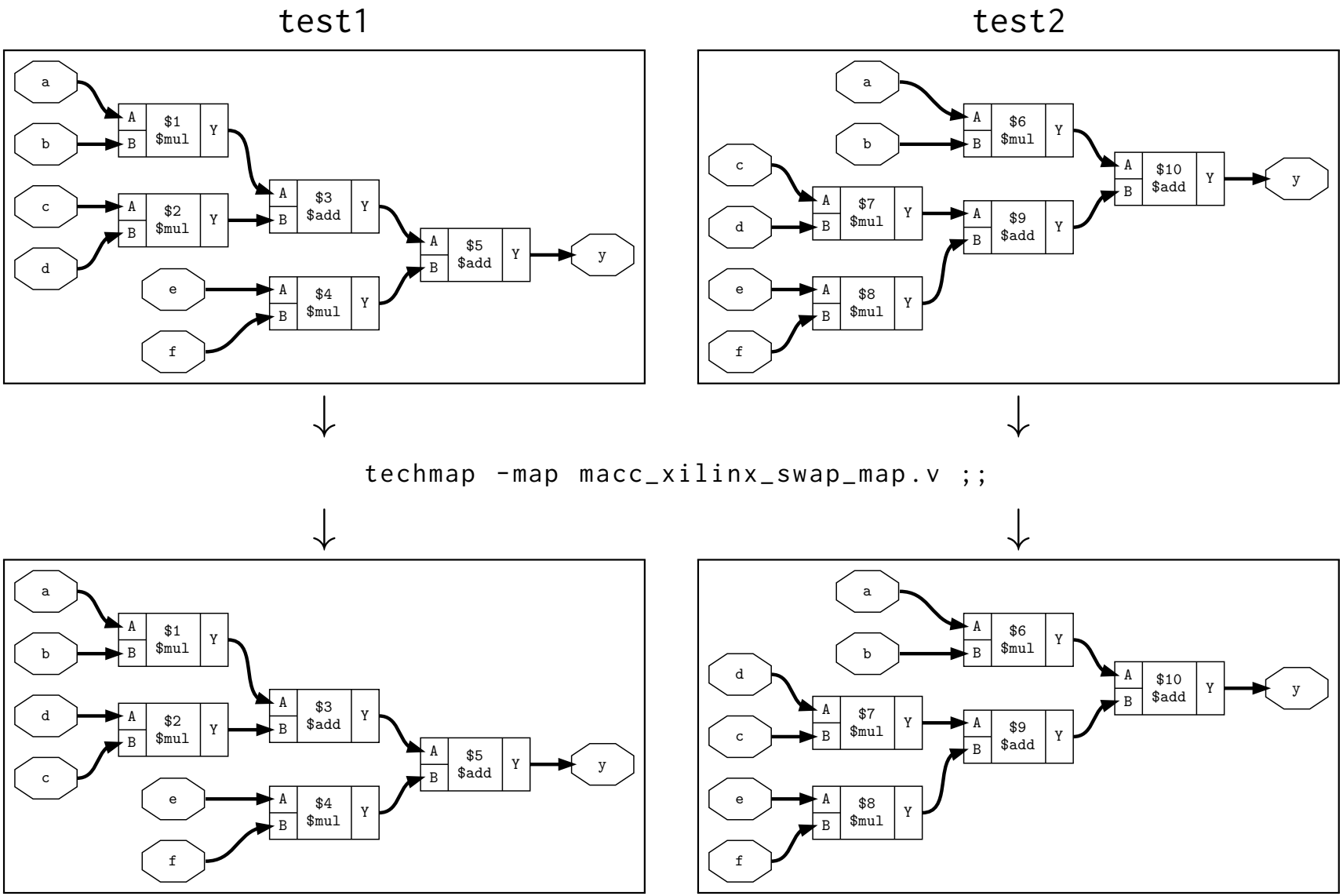
```
module test2(a, b, c, d, e, f, y);
    input [19:0] a, b, c;
    input [15:0] d, e, f;
    output [41:0] y;
    assign y = a*b + (c*d + e*f);
endmodule
```

```
read_verilog macc_xilinx_test.v
hierarchy -check
```



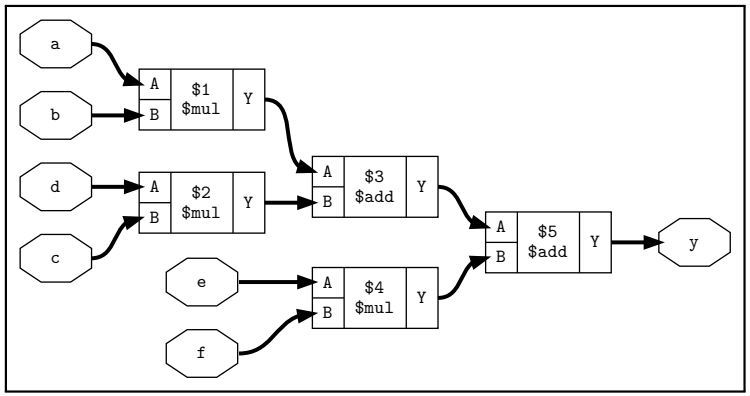


Example: DSP48_MACC – 8/13

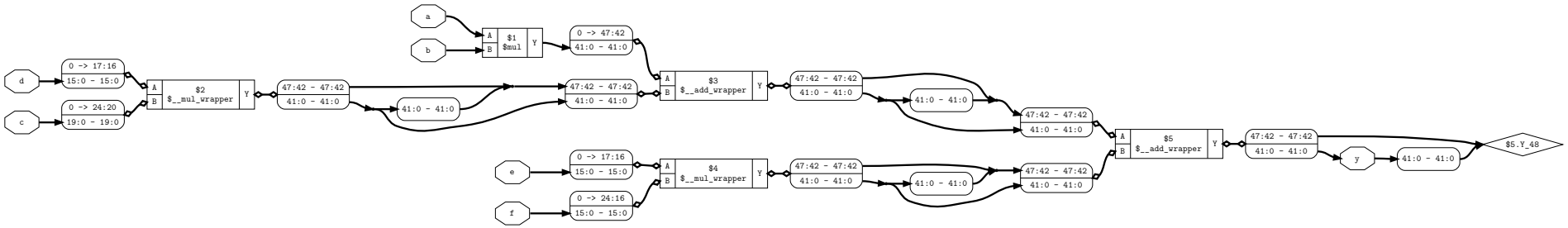


Example: DSP48_MACC – 9/13

Wrapping in `test1`:

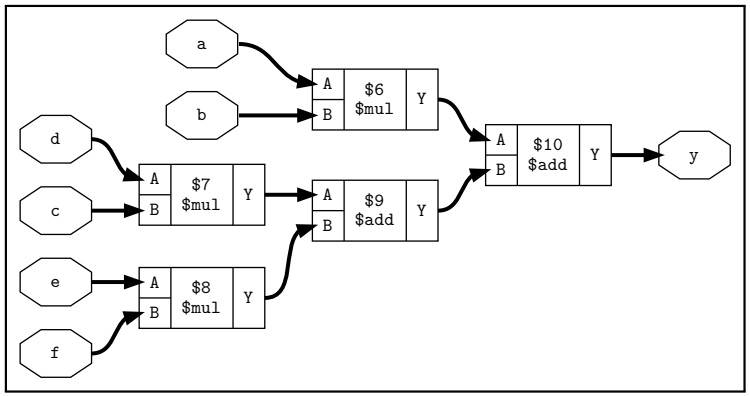



```
techmap -map macc_xilinx_wrap_map.v

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

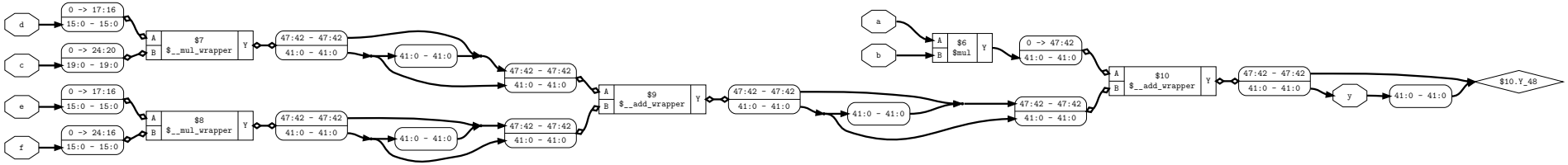


Example: DSP48_MACC – 10/13

Wrapping in `test2`:




```
techmap -map macc_xilinx_wrap_map.v

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

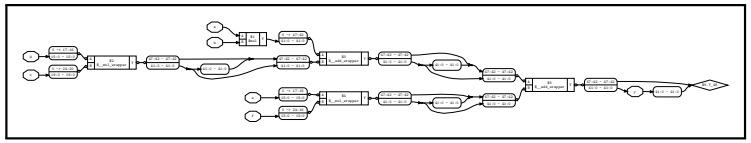


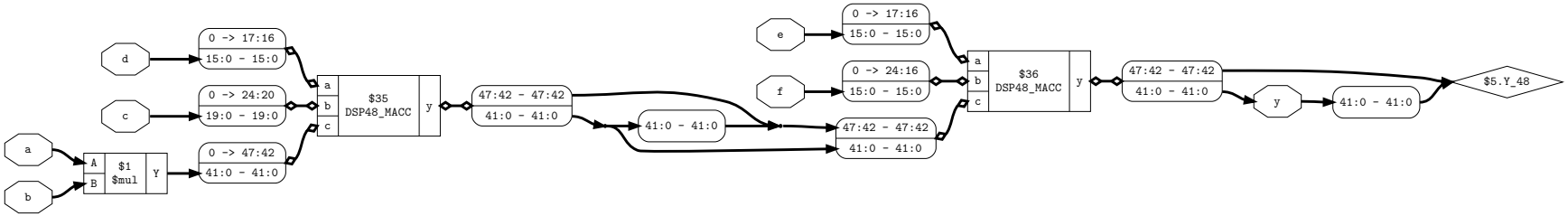
Example: DSP48_MACC – 11/13

Extract in `test1`:

```
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 ;;
```



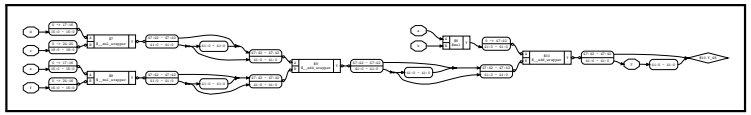


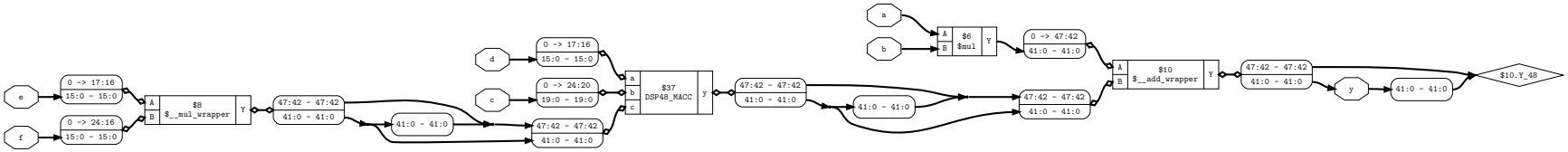
Example: DSP48_MACC – 12/13

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
```

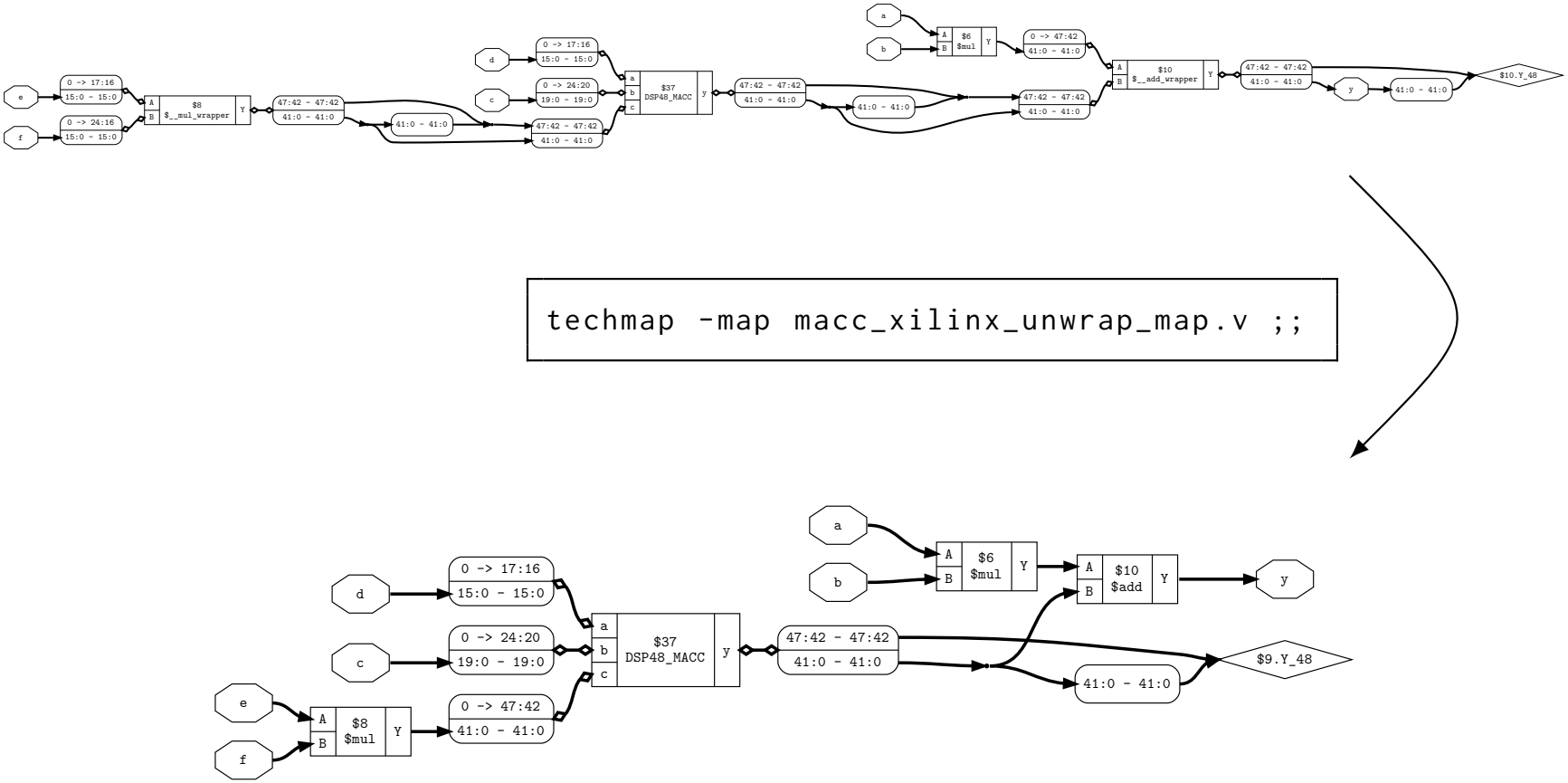
```
extract -constports -ignore_parameters \
        -map %__macc_xilinx_xmap       \
        -swap $__add_wrapper A,B ;;
```





Example: DSP48_MACC – 13/13

Unwrap in `test2`:



Subsection 4

Automatic design changes

of Section 3

Yosys by example – Advanced Synthesis

Changing the design from Yosys

Yosys commands can be used to change the design in memory. Examples of this are:

- **Changes in 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.
- **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).

Example: Async reset to sync reset

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.

```
(* techmap_celltype = "$adff" *)
module adff2dff (CLK, ARST, D, Q);

    parameter WIDTH = 1;
    parameter CLK_POLARITY = 1;
    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;
```

```
// ..continued..

    always @(posedge CLK)
        if (ARST)
            Q <= ARST_VALUE;
        else
             <= D;

endmodule
```

Summary

- A lot can be achieved in Yosys just with the standard set of commands.
- The commands `techmap` and `extract` can be used to prototype many complex synthesis tasks.

Questions?

`https://yosyshq.net/yosys/`

Section 4

Yosys by example – Beyond Synthesis

Overview

This section contains 2 subsections:

- Interactive Design Investigation
- Symbolic Model Checking

Subsection 1

Interactive Design Investigation

of Section 4

Yosys by example – Beyond Synthesis

Interactive Design Investigation

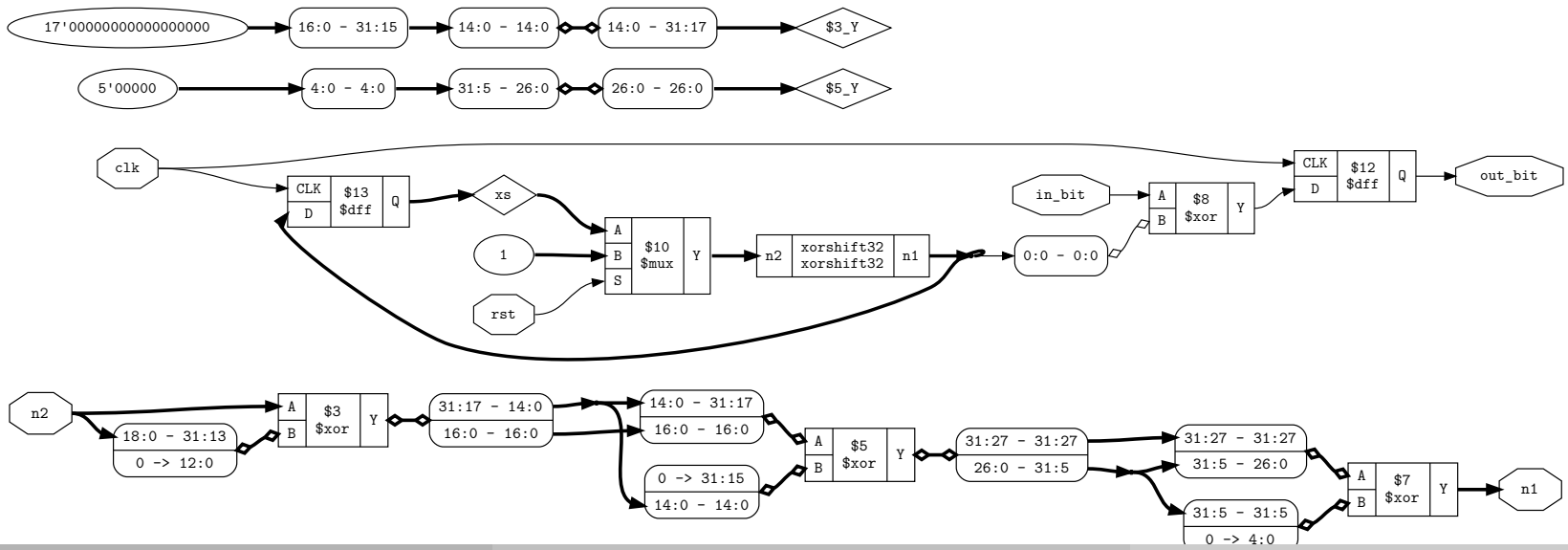
Yosys can also be used to investigate designs (or netlists created from other tools).

- The selection mechanism (see slides "Using Selections"), 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`, `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.

Example: Reorganizing a module

```
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
```

```
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
```



Example: Analysis of circuit behavior

```
> 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

> 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
```

Subsection 2

Symbolic Model Checking

of Section 4

Yosys by example – Beyond Synthesis

Symbolic Model Checking

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.

Example: Formal Equivalence Checking (1/2)

Remember the following example?

```
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
```

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

```
read_verilog techmap_01.v
hierarchy -check -top test
techmap -map techmap_01_map.v;;
```

Lets see if it is correct..

Example: Formal Equivalence Checking (2/2)

```
# 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
```

```
...

Solving problem with 945 variables and 2505 clauses..
SAT proof finished - no model found: SUCCESS!
```

Example: Symbolic Model Checking (1/2)

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.

```
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
```

```
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);
            if (counter == 8) assert(tdata == 176);
            if (counter == 9) assert(tdata ==   6);
            counter <= counter + 1;
        end
        aresetn <= 1;
    end
endmodule
```

Example: Symbolic Model Checking (2/2)

```
read_verilog -sv axis_master.v axis_test.v
hierarchy -top axis_test

proc; flatten;;
sat -seq 50 -prove-asserts
```

. . . with unmodified `axis_master.v`:

```
Solving problem with 159344 variables and 442126 clauses..
SAT proof finished - model found: FAIL!
```

. . . with fixed `axis_master.v`:

```
Solving problem with 159144 variables and 441626 clauses..
SAT proof finished - no model found: SUCCESS!
```

Summary

- Yosys provides useful features beyond synthesis.
- The commands `sat` and `eval` can be used to analyse the behavior of a circuit.
- The `sat` command can also be used for symbolic model checking.
- This can be useful for debugging and testing designs and Yosys extensions alike.

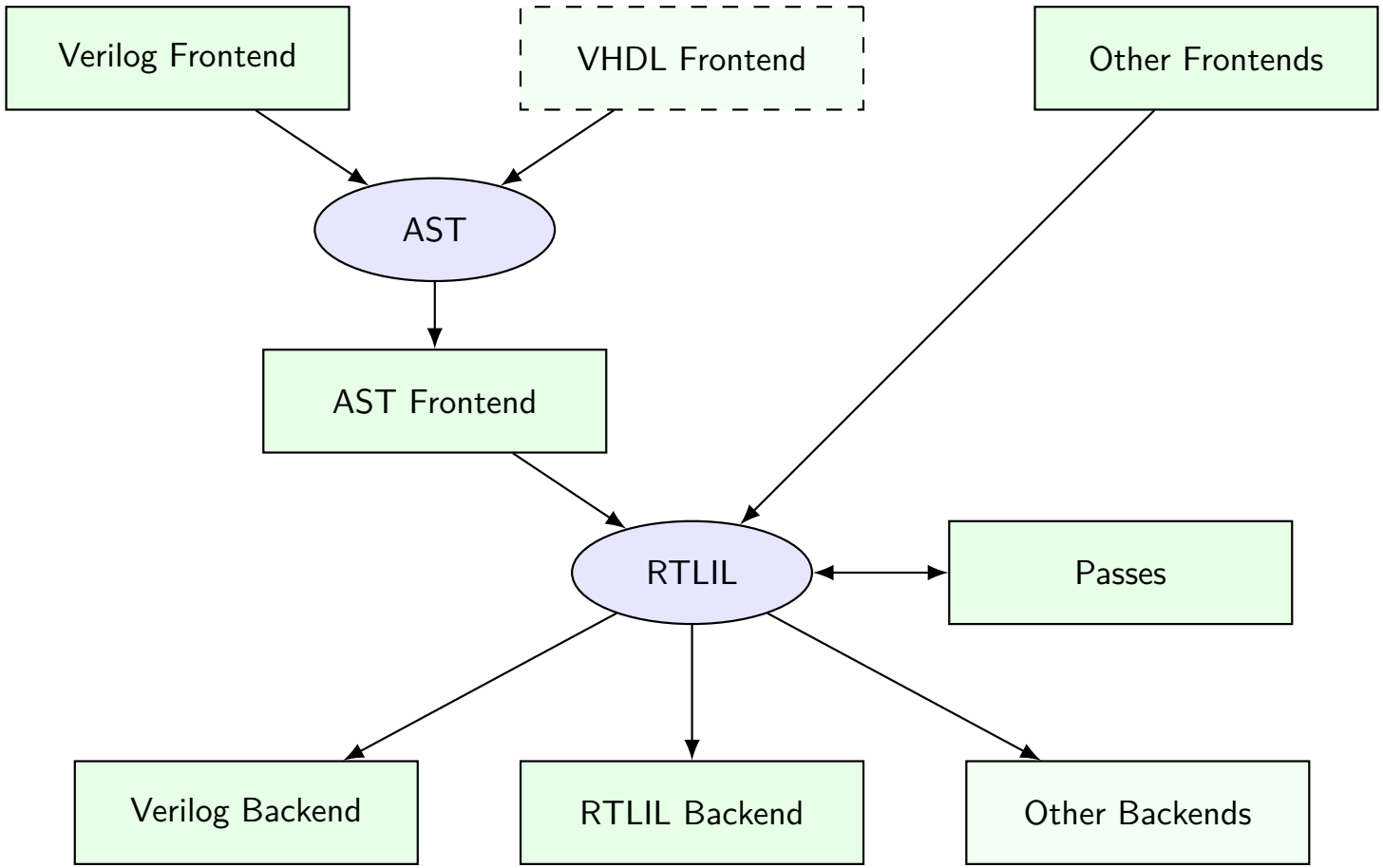
Questions?

`https://yosyshq.net/yosys/`

Section 5

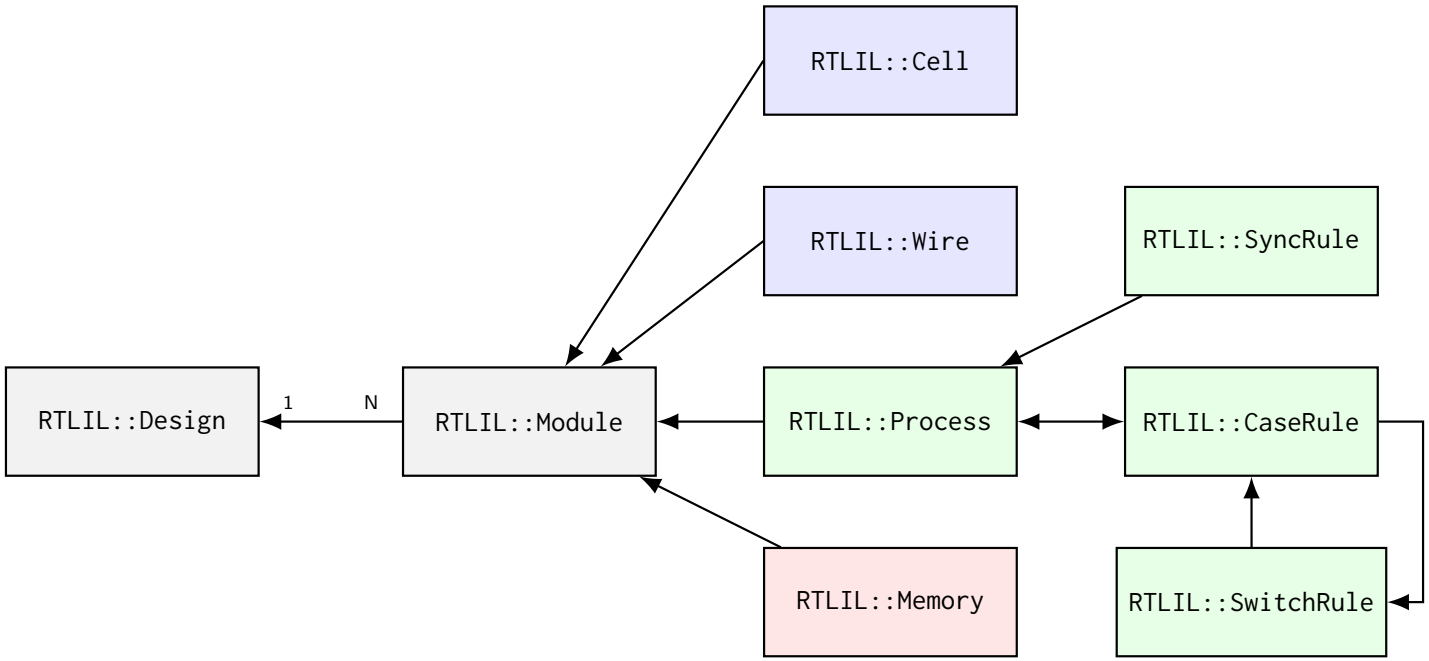
Writing Yosys extensions in C++

Program Components and Data Formats



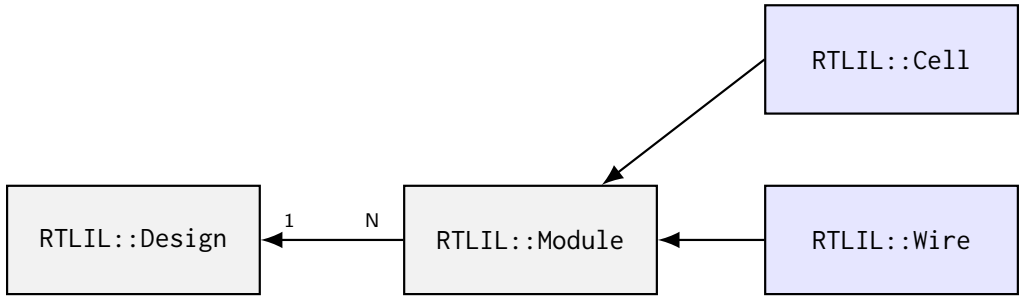
Simplified RTLIL Entity-Relationship Diagram

Between passses and frontends/backends the design is stored in Yosys' internal RTLIL (RTL Intermediate Language) format. For writing Yosys extensions it is key to understand this format.



RTLIL without memories and processes

After the commands `proc` and `memory` (or `memory -nomap`), we are left with a much simpler version of RTLIL:



Many commands simply choose to only work on this simpler version:

```
for (RTLIL::Module *module : design->selected_modules() {
    if (module->has_memories_warn() || module->has_processes_warn())
        continue;
    ....
}
```

For simplicity we only discuss this version of RTLIL in this presentation.

Using dump and show commands

- The `dump` command prints the design (or parts of it) in the text representation of RTLIL.
- The `show` command visualizes how the components in the design are connected.

When trying to understand what a command does, create a small test case and look at the output of `dump` and `show` before and after the command has been executed.

The RTLIL Data Structures

The RTLIL data structures are simple structs utilizing `pool<>` and `dict<>` containers (drop-in replacements for `std::unordered_set<>` and `std::unordered_map<>`).

- Most operations are performed directly on the RTLIL structs without setter or getter functions.
- In debug builds a consistency checker is run over the in-memory design between commands to make sure that the RTLIL representation is intact.
- Most RTLIL structs have helper methods that perform the most common operations.

See `yosys/kernel/rtlil.h` for details.

RTLIL::IdString

`RTLIL::IdString` in many ways behave like a `std::string`. It is used for names of RTLIL objects. Internally a RTLIL::IdString object is only a single integer.

The first character of a `RTLIL::IdString` specifies if the name is *public* or *private*:

- `RTLIL::IdString[0] == '\\'`:
  This is a public name. Usually this means it is a name that was declared in a Verilog file.

- `RTLIL::IdString[0] == '$'`:
  This is a private name. It was assigned by Yosys.

Use the `NEW_ID` macro to create a new unique private name.

RTLIL::Design and RTLIL::Module

The `RTLIL::Design` and `RTLIL::Module` structs are the top-level RTLIL data structures. Yosys always operates on one active design, but can hold many designs in memory.

```
struct RTLIL::Design {
    dict<RTLIL::IdString, RTLIL::Module*> modules_;
    ...
};

struct RTLIL::Module {
    RTLIL::IdString name;
    dict<RTLIL::IdString, RTLIL::Wire*> wires_;
    dict<RTLIL::IdString, RTLIL::Cell*> cells_;
    std::vector<RTLIL::SigSig> connections_;
    ...
};
```

(Use the various accessor functions instead of directly working with the `*_` members.)

The RTLIL::Wire Structure

Each wire in the design is represented by a `RTLIL::Wire` struct:

```
struct RTLIL::Wire {
    RTLIL::IdString name;
    int width, start_offset, port_id;
    bool port_input, port_output;
    ...
};
```

`width`	The total number of bits. E.g. 10 for `[9:0]`.
`start_offset`	The lowest bit index. E.g. 3 for `[5:3]`.
`port_id`	Zero for non-ports. Positive index for ports.
`port_input`	True for `input` and `inout` ports.
`port_output`	True for `output` and `inout` ports.

RTLIL::State and RTLIL::Const

The `RTLIL::State` enum represents a simple 1-bit logic level:

```
enum RTLIL::State {
    S0 = 0,
    S1 = 1,
    Sx = 2, // undefined value or conflict
    Sz = 3, // high-impedance / not-connected
    Sa = 4, // don't care (used only in cases)
    Sm = 5  // marker (used internally by some passes)
};
```

The `RTLIL::Const` struct represents a constant multi-bit value:

```
struct RTLIL::Const {
    std::vector<RTLIL::State> bits;
    ...
};
```

Notice that Yosys is not using special `VCC` or `GND` driver cells to represent constants. Instead constants are part of the RTLIL representation itself.

The RTLIL::SigSpec Structure

The `RTLIL::SigSpec` struct represents a signal vector. Each bit can either be a bit from a wire or a constant value.

```
struct RTLIL::SigBit
{
    RTLIL::Wire *wire;
    union {
        RTLIL::State data; // used if wire == NULL
        int offset;        // used if wire != NULL
    };
    ...
};

struct RTLIL::SigSpec {
    std::vector<RTLIL::SigBit> bits_; // LSB at index 0
    ...
};
```

The `RTLIL::SigSpec` struct has a ton of additional helper methods to compare, analyze, and manipulate instances of `RTLIL::SigSpec`.

The RTLIL::Cell Structure(1/2)

The `RTLIL::Cell` struct represents an instance of a module or library cell.

The ports of the cell are associated with `RTLIL::SigSpec` instances and the parameters are associated with `RTLIL::Const` instances:

```
struct RTLIL::Cell {
    RTLIL::IdString name, type;
    dict<RTLIL::IdString, RTLIL::SigSpec> connections_;
    dict<RTLIL::IdString, RTLIL::Const> parameters;
    ...
};
```

The `type` may refer to another module in the same design, a cell name from a cell library, or a cell name from the internal cell library:

```
$not $pos $neg $and $or $xor $xnor $reduce_and $reduce_or $reduce_xor $reduce_xnor
$reduce_bool $shl $shr $sshl $sshr $lt $le $eq $ne $eqx $nex $ge $gt $add $sub $mul $div $mod
$divfloor $modfloor $pow $logic_not $logic_and $logic_or $mux $pmux $slice $concat $lut $assert $sr $dff
$dffsr $adff $dlatch $dlatchsr $memrd $memwr $mem $fsm $_NOT_ $_AND_ $_OR_ $_XOR_ $_MUX_ $_SR_NN_
$_SR_NP_ $_SR_PN_ $_SR_PP_ $_DFF_N_ $_DFF_P_ $_DFF_NN0_ $_DFF_NN1_ $_DFF_NP0_ $_DFF_NP1_ $_DFF_PN0_
$_DFF_PN1_ $_DFF_PP0_ $_DFF_PP1_ $_DFFSR_NNN_ $_DFFSR_NNP_ $_DFFSR_NPN_ $_DFFSR_NPP_ $_DFFSR_PNN_
$_DFFSR_PNP_ $_DFFSR_PPN_ $_DFFSR_PPP_ $_DLATCH_N_ $_DLATCH_P_ $_DLATCHSR_NNN_ $_DLATCHSR_NNP_
$_DLATCHSR_NPN_ $_DLATCHSR_NPP_ $_DLATCHSR_PNN_ $_DLATCHSR_PNP_ $_DLATCHSR_PPN_ $_DLATCHSR_PPP_
```

The RTLIL::Cell Structure(2/2)

Simulation models (i.e. *documentation* :-) for the internal cell library:

`yosys/techlibs/common/simlib.v` and
`yosys/techlibs/common/simcells.v`

The lower-case cell types (such as `$and`) are parameterized cells of variable width. This so-called *RTL Cells* are the cells described in `simlib.v`.

The upper-case cell types (such as `$_AND_`) are single-bit cells that are not parameterized. This so-called *Internal Logic Gates* are the cells described in `simcells.v`.

The consistency checker also checks the interfaces to the internal cell library. If you want to use private cell types for your own purposes, use the `$__`-prefix to avoid name collisions.

Connecting wires or constant drivers

Additional connections between wires or between wires and constants are modelled using `RTLIL::Module::connections`:

```
typedef std::pair<RTLIL::SigSpec, RTLIL::SigSpec> RTLIL::SigSig;

struct RTLIL::Module {
    ...
    std::vector<RTLIL::SigSig> connections_;
    ...
};
```

`RTLIL::SigSig::first` is the driven signal and `RTLIL::SigSig::second` is the driving signal. Example usage (setting wire `foo` to value `42`):

```
module->connect(module->wire("\\foo"),
                RTLIL::SigSpec(42, module->wire("\\foo")->width));
```

Creating modules from scratch

Let's create the following module using the RTLIL API:

```
module absval(input signed [3:0] a, output [3:0] y);
    assign y = a[3] ? -a : a;
endmodule

RTLIL::Module *module = new RTLIL::Module;
module->name = "\\absval";

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, 1, 3), y);

module->fixup_ports();
```

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 slide) when you need a unique handle for each signal bit.

Using the SigMap helper class

Consider the following module:

```
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:

```
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[5]:

```
log("Mapped signal x: %s\n", log_signal(sigmap(x)));
```

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:

```
log_header(design, "Doing important stuff!\n");
log_push();
for (int i = 0; i < 10; i++)
    log("Log message #%d.\n", i);
log_pop();
```

[5] The pointer returned by `log_signal()` is automatically freed by the log framework at a later time.

Error handling

Use `log_error()` to report a non-recoverable error:

```
if (design->modules.count(module->name) != 0)
    log_error("A␣module␣with␣the␣name␣%s␣already␣exists!\n",
            RTLIL::id2cstr(module->name));
```

Use `log_cmd_error()` to report a recoverable error:

```
if (design->selection_stack.back().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()`.

Creating a command

Simply create a global instance of a class derived from `Pass` to create a new yosys command:

```
#include "kernel/yosys.h"
USING_YOSYS_NAMESPACE

struct MyPass : public Pass {
    MyPass() : Pass("my_cmd", "just a simple test") { }
    virtual void execute(std::vector<std::string> args, RTLIL::Design *design)
    {
        log("Arguments to my_cmd:\n");
        for (auto &arg : args)
            log("  %s\n", arg.c_str());

        log("Modules in current design:\n");
        for (auto mod : design->modules())
            log("  %s (%d wires, %d cells)\n", log_id(mod),
                    GetSize(mod->wires()), GetSize(mod->cells()));
    }
} MyPass;
```

Creating a plugin

Yosys can be extended by adding additional C++ code to the Yosys code base, or by loading plugins into Yosys.

Use the following command to compile a Yosys plugin:

```
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
```

Load the plugin using the yosys `-m` option:

```
yosys -m ./my_cmd.so -p 'my_cmd foo bar'
```

Summary

- Writing Yosys extensions is very straight-forward.
- . . . and even simpler if you don't need RTLIL::Memory or RTLIL::Process objects.

- Writing synthesis software? Consider learning the Yosys API and make your work part of the Yosys framework.

Questions?

`https://yosyshq.net/yosys/`