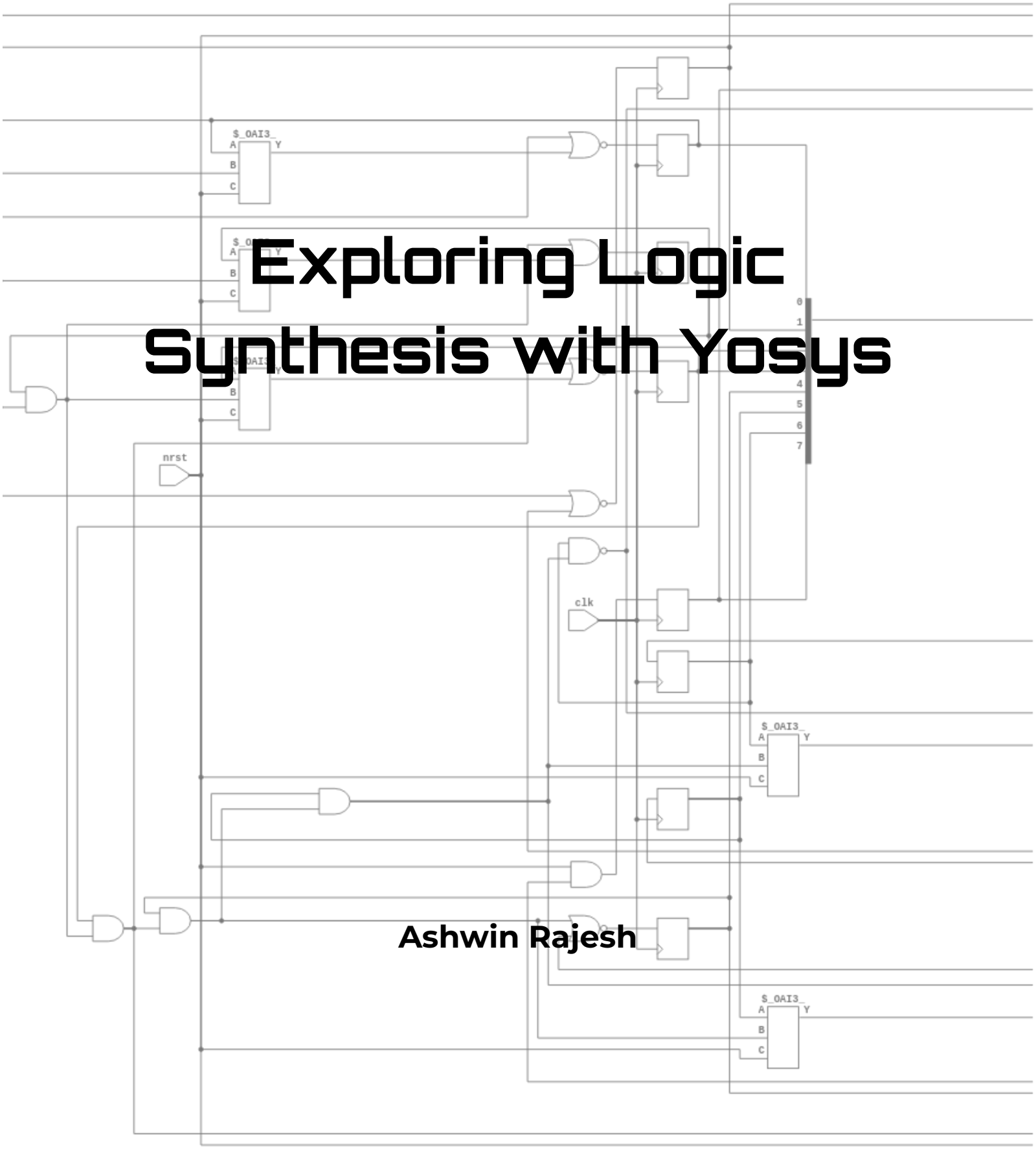


Exploring Logic Synthesis with Yosys

Ashwin Rajesh

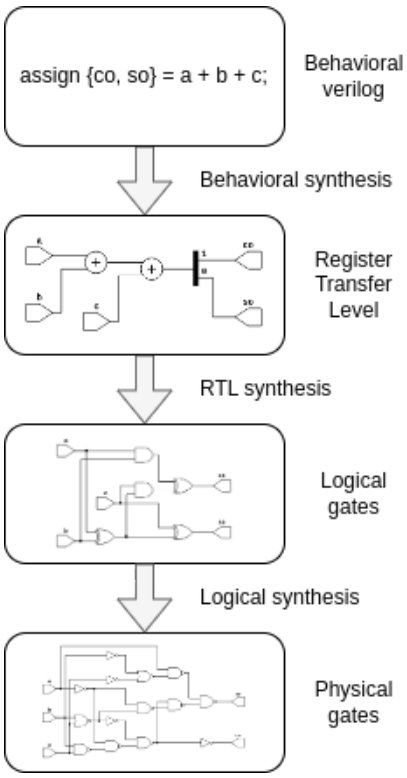
List of Contents

Section I : Introduction

A large part of digital design now is done using languages like verilog or VHDL. The process of converting lines of code into digital circuits is often treated like a mysterious black box while learning HDL coding. In this book, I will demystify this using Yosys, an open-source logic synthesis tool where you can get hands-on experience of every step of synthesizing digital logic.

What is Yosys?

Modern digital designs are made using languages like Verilog or VHDL, which describe logic behaviorally using high-level constructs like if-else, case, etc. This is then synthesized into gates and other structures that can be placed in a chip or FPGA. Later, physical synthesis tools place and connect these cells on the chip or map them to FPGA resources.



Logic synthesis flow
(full adder to NAND, NOR)

Logic synthesis involves multiple stages, as shown in the diagram. First, the behavioral verilog program is converted to a **Register Transfer Level** netlist, or what's commonly referred to as a **dataflow style** model, which consists of registers, gates, and other high-level blocks like multipliers, adders, multiplexers, etc. Then, we map this to a **generic set** of logic gates and finally to gates available in the **target technology** (LUTs in FPGAs or nand, nor, inv, etc, in CMOS).

What can we do with it?

Its primary use is to **synthesize verilog** (no VHDL support) into any subset of hardware blocks, be it LUTs or standard cells. We can also convert designs from, say, LUTs to multiplexers, and so on, or explore what our design looks like, such as using NAND gates or multiplexers, etc. In this article, I will be exploring how it works to synthesize logic and how you can write scripts to

synthesize the way you want. It is also quite useful for working with logic in general and has been used to build other tools, such as **formal verification**.

How does Yosys work?

It uses an internal format called **RTLIL** to represent a circuit at any level of abstraction (behavioral, RTL, etc.). The main components of Yosys are called "**passes**," which transform the circuit in some way, be it by optimizing it or mapping it to some elements. Passes work on the RTLIL representation. Different **frontends** can be used to read into the RTLIL representation. This includes verlog code and netlists in JSON or BLIF format. The **backends** can export the RTLIL at any stage into any of these formats or even as a spice netlist or a C program to simulate the circuit. This **modular design** makes Yosys very versatile and extensible.

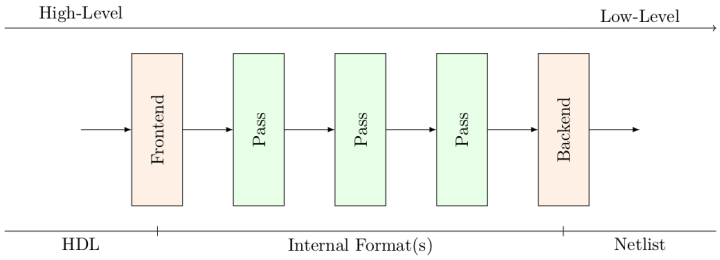


Image source : Yosys manual [1]

We can use Yosys by entering commands in a file or using a real-time command line interface. These commands can be used to call these frontends, backends, or passes or analyze the circuit. Essentially, these commands are the language of yosys, and that is what we need to learn to use it.

Installation & Setup

The easiest way to install, if you are on linux is by using a package manager like **apt** for ubuntu for example. If you use ubuntu, enter this command and you should have it! For other linux distros, try the same with your package manager

```
sudo apt-get install yosys
```

Another way to install Yosys is as part of the OSS CAD Suite from YosysHQ, the maintainers of Yosys. Head over to the releases page and download the version for your OS (look at the file names). Follow the instructions in the README, and you should have it running! This would also install their formal verification tool, symbiyosys (i may write an article on that later if time permits!).

Once you have it installed, launch Yosys using the command line. You can invoke it in interactive mode, where you can type each command in a command line one-by-one and view its outputs or write the commands in a script file and pass it to yosys. Yosys scripts usually use the .ys extension, but it is a simple ASCII text file.

```
yosys
```
(interactive mode)

```
yosys <script.ys>
```
(runs the commands in the file)

The version of Yosys from apt is a bit old and lacks some features, so I have built it from the github source. I have used the latest version, which, as of writing, is version 0.43.

Section II : Typical yosys flow & common commands

First, let us go through a simple Yosys synthesis script to understand the language, common commands, and the general flow. We shall go deeper into these commands later.

A simple script and verilog design

This is the Yosys script we will be using. It has some of the most useful and popular commands in Yosys and shows the general flow of logic synthesis. We can enter each of these commands in the interactive mode and view the output using the show command at any stage.

```
# Read input file to internal representation
read_verilog counter.v

# Convert high-level behavioral parts to DFFs and muxes
proc

# Perform some simple optimizations
opt

# Convert design to gate-level netlists from an internal library
techmap

# Perform some simple optimizations
opt

# Use ABC to map cells to the target technology
abc

# Cleanup
opt

# Write results to output file
write_verilog counter_out.v
```

The verilog code we will try to synthesize, named “counter.v” is a simple 4-bit counter with synchronous reset and enable as shown below.

```
module counter(
    input clk, nrst, en,

    output[3:0] count
);

    reg[3:0] count;

    always @(posedge clk) begin
        if(~nrst)
            count <= 0;
        else if(en)
            count <= count + 1;
    end

endmodule
```

Let us now try to understand what happens in the Yosys script, line by line. Firstly, let us see how the command language works.

Understanding the script language

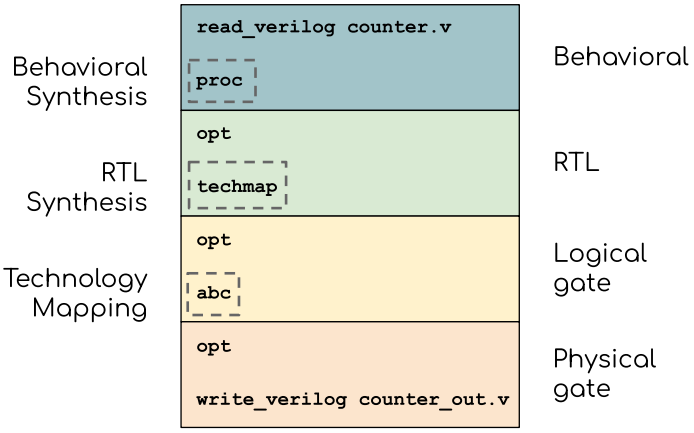
The language is similar to a command line language (like bash) in that we start with a command name and then follow with options and inputs. Options are always preceded with a hyphen (-) and can be succeeded by an input to the option. The command inputs come at the end of the command. For example in

```
dfflibmap -liberty cells.lib top_module
```

“`dfflibmap`” is the command, “`liberty`” is an option and “`cells.lib`” is the input to the “`liberty`” option. “`top_module`” is the input to the command. You can find all the commands and usage well documented in the yosys manual commandline reference. Otherwise, just use the “help” command in the interactive view.

Understanding the flow

As discussed in the introduction, the process of synthesis from behavioral verilog code involves multiple steps.



Decoding the yosys script

Firstly, we import the verilog design using the **read_verilog** command. Then, we convert behavioral RTL code into “RTL code” in behavioral synthesis. Here, RTL refers to “register transfer level”, which means all logic is explained as registers and logic between them. There are no behavioral constructs like case or if statements. The **proc** command performs this translation.

Next, we have RTL synthesis, where the logic is described by logical gates. The **techmap** command performs this step. Finally, we have logical synthesis where the gates are mapped to gates available in our technology. This is done by the **abc** command. This step also considers timing constraints. We use the **opt** command to perform simple optimizations after each step. In the end, we export the netlist as verilog using **write_verilog**.

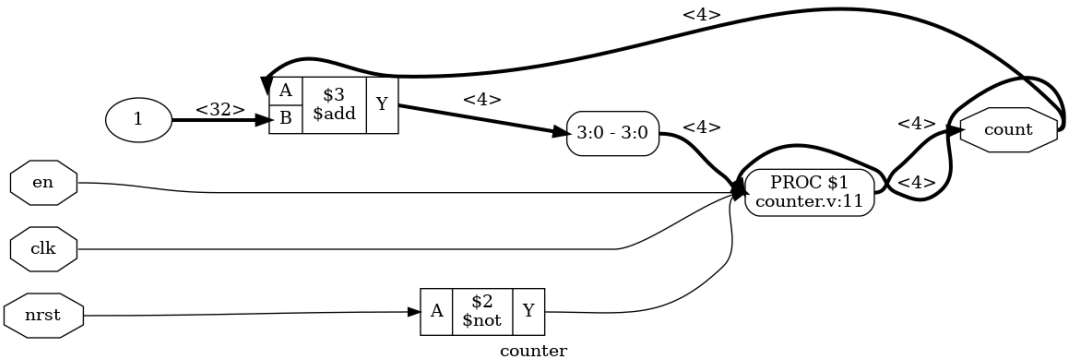
Now, let us look at some very useful commands for using with Yosys.

help : Get info about yosys commands

If you need to know what any command does and its options, just use the help command with the command name as the input. Typing help without any input will list all the commands.

show : Visualize the design

This is one of the most useful commands for debugging the design at any stage. It may not be as pretty as Vivado with component symbols, but it is good enough for debugging small circuits. The output immediately after importing the verilog file is shown below.



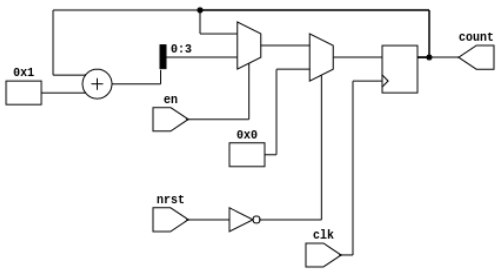
Here, the PROC block corresponds to the always block in the code. The RHS expressions used in the block, i.e **`~nrst`**, **`clk`**, **`en`**, and **`count + 1`** are inputs to the block, and the variables being set (count in our case) are the outputs of the PROC block. All logic except the behaviorally defined logic in the always block is now in a netlist form.

proc : Behavioral synthesis

The proc command synthesizes the PROC blocks in a design. That is, it performs behavioral synthesis by converting the design to an RTL format.

The output after the proc command is shown below. I use another tool called netlistsvg to generate these diagrams. I cover how to generate these neater diagrams in another section. You can use the show command to view the circuit for now. It may be messy, but it's convenient.

Comparing the code and the circuit, we can understand that all it did was translate the if condition to multiplexers and infer a d flip-flop for the count variable. The output is a proper RTL netlist at this point. We will see how the proc command decodes an always or initial block later in another section.



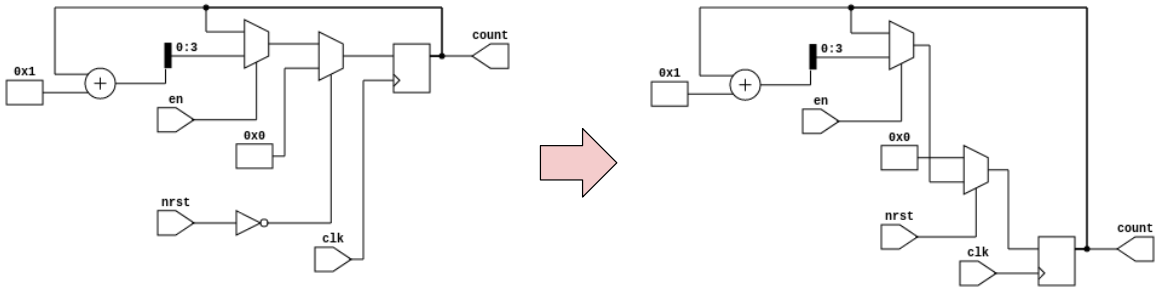
opt : Simple optimizations

There are several common optimizations that can reduce unnecessary computations. These are often “common sense” for digital designers. For example,

1. If all inputs to a block are constant, precompute the output.
2. If an input to an OR gate is always 1, replace the output with 1
3. If an input to an AND gate is always 1, remove that input.

We will go deeper into these optimizations in the next section but for now, let us just take for granted that this performs several such simple optimizations iteratively to simplify the circuit. We call the opt pass after most other passes to remove redundant components.

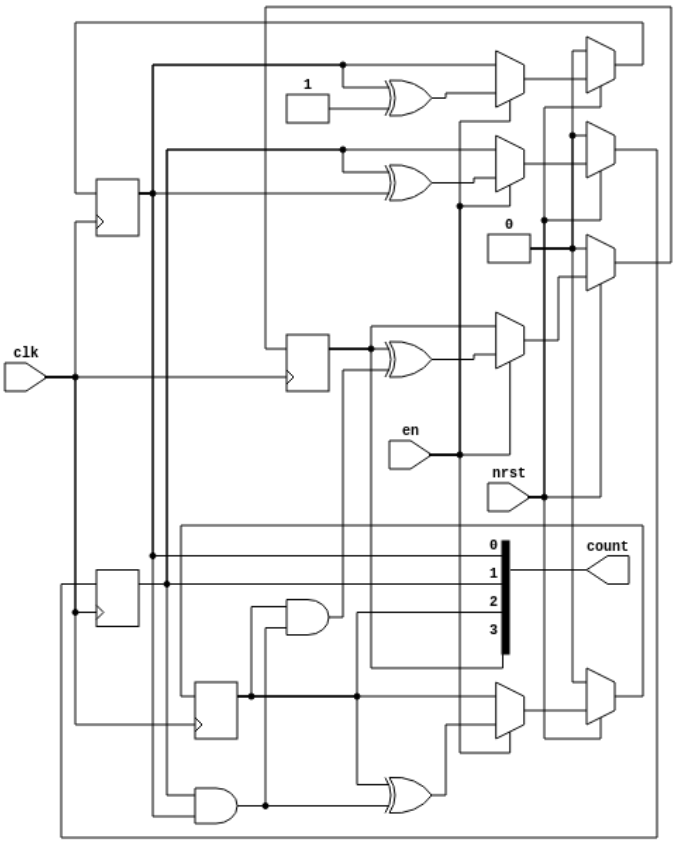
In this case, the opt pass optimized the multiplexer responsible for reset and swapped the inputs of the multiplexer to eliminate the inverter. Again, a common-sense optimization that we can write a rule for.



techmap : RTL synthesis

This command performs **RTL synthesis**, by converting higher level elements like adders, subtractors, shifters and multipliers to individual gates. It simply substitutes RTL blocks into gates from a library using a “map” for each block. It can also extract blocks like custom DSP and ALU units.

The gate-level circuit of the counter is shown below. You can try to find the adders and other components in this circuit. You may be able to trace the signals and confirm that it uses a register, an incrementor (AND, XOR gates) and multiplexers for enable and reset.

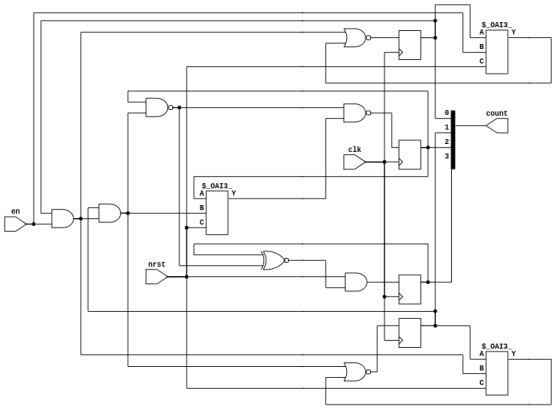


abc : Boolean optimization & Technology mapping

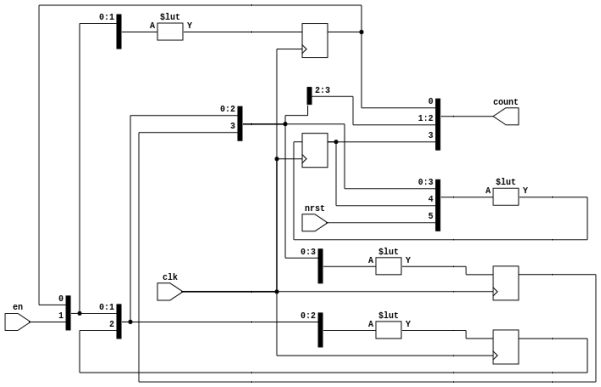
The final step is optimally mapping these gates into those we have available in our technology with timing constraints. The ABC tool is used for this. It is a **logic synthesis** engine, another open-source project developed at Berkeley.

Without any arguments, it converts it into a default cell library. The output after abc and opt is shown below. We can see the ripple adder much more clearly here. The ripple adder is implemented using OAI (Or-And-Invert) gates AND gates and a NOR gate. We can also set other sets of gates as the target. For example, if we want to see how the above design can be implemented using FPGAs with 6 input LUTs, we can use the “abc -lut 6” command.

Default result



With LUTs



stat : Get statistics of the design

This is a handy command that prints simple statistics like the number of different components and the sum of cell areas if a liberty file is used. The output for our design after all the steps is:

```
=== counter ===

   Number of wires:                 12
   Number of wire bits:             18
   Number of public wires:           4
   Number of public wire bits:       7
   Number of memories:               0
   Number of memory bits:            0
   Number of processes:              0
   Number of cells:                 15
     $_ANDNOT_                       1
     $_AND_                          3
     $_DFF_P_                        4
     $_NAND_                         1
     $_NOR_                          2
     $_OAI3_                         3
     $_XNOR_                         1
```

write_verilog : Write output to a verilog file

This is a “backend” for the flow, exporting the RTLIL to a verilog file. The output is not very readable and is meant for use by other tools.

Section III : Behavioral Synthesis

Behavioral synthesis is the process of converting behavioral logic into an RTL netlist. Practically, this involves converting the behavioral logic we write in always and initial blocks into implementable hardware blocks like multiplexers and registers. Let us take a look at how this works.

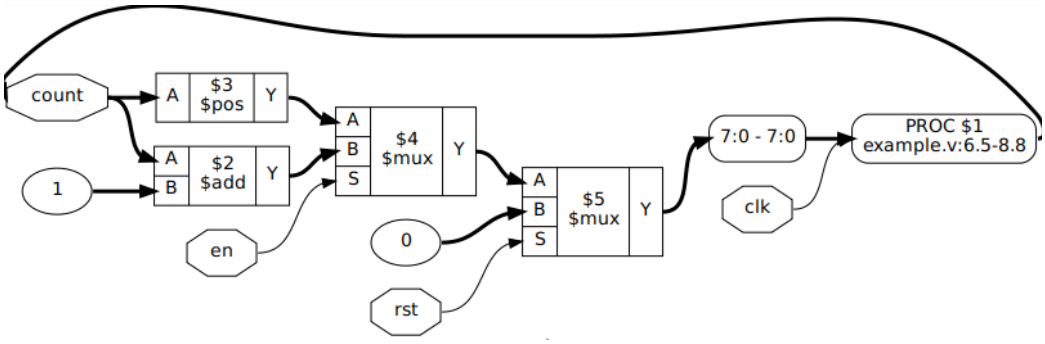
Parsing of behavioral code

When reading verilog code, Yosys splits a behavioral (always / initial) block into two parts. The first part is the **datapath** which consists of the expressions used on the right-hand side of each statement/assignment in the block. This can be directly converted into a structural representation by converting the operators in the expression into a graph. Even structures defined in generate blocks are elaborated at this stage.

The other part is the **control path** which defines which of these statements are executed and in what order. This is done using dynamic constructs like if, case, and for. Essentially, the control path defines when and what assignments are made to the variables on the left-hand side of the expressions.

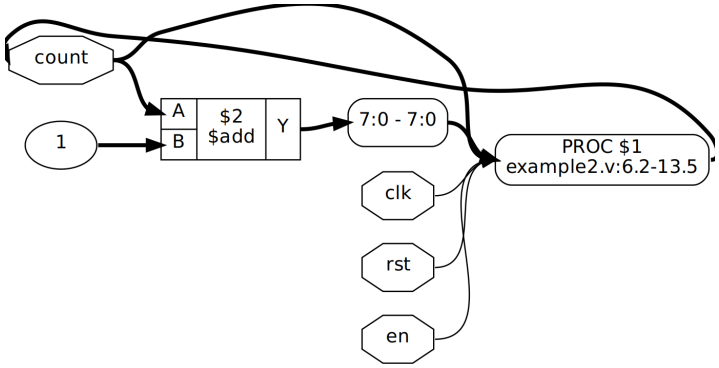
Yosys generates a **PROC** block for the control path when reading a verilog file using read_verilog. The datapath is directly mapped to a circuit. Let us consider the example of an up counter as given below.

```
always @(posedge clk) begin
     count <= rst ? 0 : (en ? (count + 1) : count);
end
```



Here, there is not much control apart from the “`posedge clk`” trigger, which indicates that the count variable is assigned on the posedge of “`clk`”. The datapath consists of “`rst ? 0 : (en ? (count + 1) : count)`”. The same code can be written using if-else conditions, as shown below

```
always @(posedge clk) begin
     if(rst)
          count <= 0;
     else if(en)
          count <= count + 1;
     else
          count <= count;
end
```



Here, the multiplexers have moved into the PROC control block, and only the adder is part of the datapath. Both these will give the same circuit after the proc pass, but the proc pass has to extract the multiplexers in the second case. The PROC block can be thought of as taking every right-hand expression from the block as the input and assigning it to the left-hand side variables.

Most Yosys passes cannot operate on behavioral-level circuits with the PROC block, so the first step in synthesis is always running the proc pass.

The proc pass

The **proc** pass transforms the PROC blocks into RTL structures like registers and multiplexers.

The proc pass is broken down into multiple other passes, each of which performs a part of the translation. The proc pass is equivalent to calling the following passes sequentially.

```
# Cleaning up the PROC block decision trees
proc_clean      # Remove empty branches and processes
proc_rmdead     # Remove unreachable branches
proc_prune      # Remove redundant branches

# Identify special patterns (resets, initial, etc)
proc_init       # Set initial values from initial blocks
proc_arst       # Identify async resets

# Extract components of the PROC block
proc_rom        # Extract ROMs
proc_mux        # Extract multiplexers
proc_dlatch     # Extract latches
proc_dff        # Extract flip flops
proc_memwr      # Extract memory write ports

# Clean up
proc_clean      # Remove the now redundant PROC blocks
opt_expr -keepdc    # Const optimization & cleanup
```

The first three commands, i.e. **proc_clean**, **proc_rmdead**, and **proc_prune**, optimize the internal control flow of the PROC block. The **proc_init** block parses the initial block code and sets the initial values of variables.

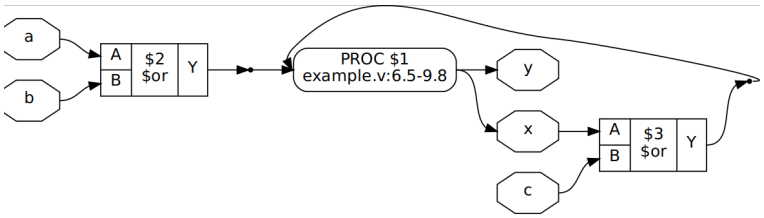
The **proc_arst** block detects asynchronous sets/resets (from the sensitivity list) and flags it. This is required because asynchronous resets need dedicated flip-flops. The rest of the passes extract different components from the PROC block, like multiplexers, flip flops, latches, and memory writes.

Parsing blocking / non-blocking statements

First, let us see how blocking and nonblocking statements are parsed in the datapath. The following are three examples. The first two are combinational always blocks, and the last one is a synchronized always block.

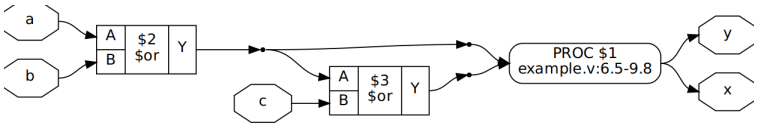
Example 1 : combinational nonblocking

```
always @(*) begin
     x <= a | b;
     y <= x | c;
end
```



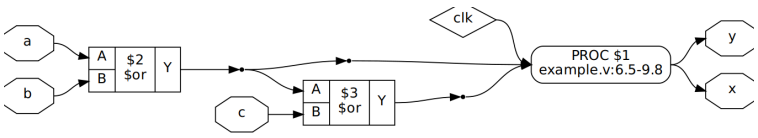
Example 2 : combinational blocking

```
always @(*) begin
     x = a | b;
     y = x | c;
end
```



Example 3 : clocked blocking

```
always @(posedge clk)
begin
     x = a | b;
     y = x | c;
end
```



For blocking assignments, the value for Y is computed directly using the expression for X (a | b). However, for nonblocking assignments, the value for Y is computed from the output of the PROC block. This makes sense if we think of the PROC block as **performing the assignments** of the two statements, so the value of Y is computed from the previously assigned value of X.

The only difference between the combinational example (2nd) and the clocked example (3rd) is that the clocked PROC block also has a clk input. In this case, the PROC block also models the posedge flip flop involved because the assignment is done on the posedge of the clock. If we had written Example 3 using nonblocking assignments, the result would be similar to Example 2 but with the CLK input as well.

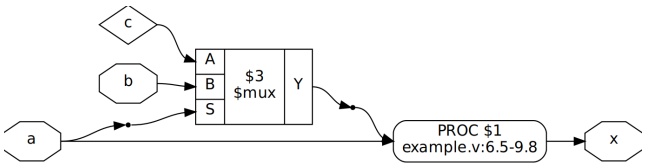
Ultimately, the 1st and 2nd circuits must be mapped to be identical. This will happen because the PROC block will synthesize to be a passthrough in both cases since assignment is done anytime the right-hand side changes.

proc_mux : Extracting multiplexer trees

The if/else and case statements in always blocks are extracted to multiplexers. This is done in this pass. Let us look at some examples and their circuit after proc_mux. proc_mux effectively **merges expressions** (x = c, x = b) into a single expression (x = a ? b : c) using multiplexers, **considering the control path**.

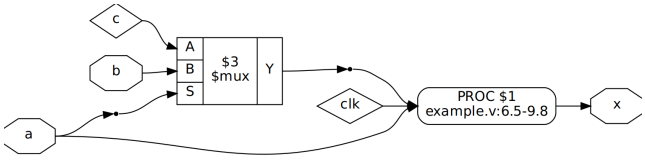
Example 1 : combinational if/else

```
always @(*) begin
     if(a)     x <= b;
     else      x <= c;
end
```



Example 2 : clocked if/else

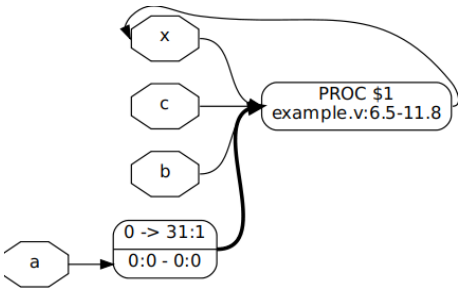
```
always @(posedge clk) begin
     if(a)     x <= b;
     else      x <= c;
end
```



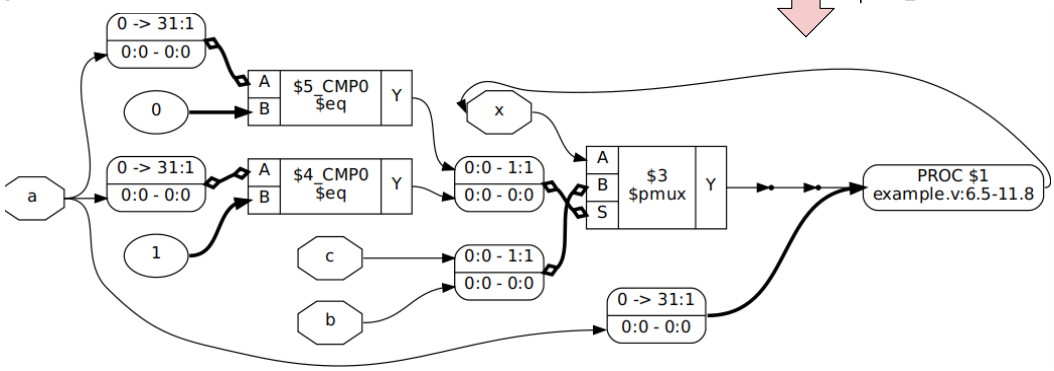
Example 3 : combinational case

```
always @(*) begin
     case(a)
          0: x = c;
          1: x = b;
     endcase
end
```

read_verilog



proc_mux



Even at this stage, the only visible difference between the clocked and combinational always blocks (1st, 2nd examples) is that the clocked block has a CLK input. The 3rd example was synthesized very differently than the 1st example despite having identical functionality because it uses the **case statement**. It compares the input with each key using the $eq block and selects the desired expression using a PMUX (priority mux).

A PMUX is a cascade of multiplexers, equivalent to cascaded if-else statements. PMUX is used here since case statements are comparable to an if-else ladder in verilog, with the first case having the highest priority. Examples 3 and 1 will be synthesized identically after further optimizations.

proc_dlatch : Extracting D Latches

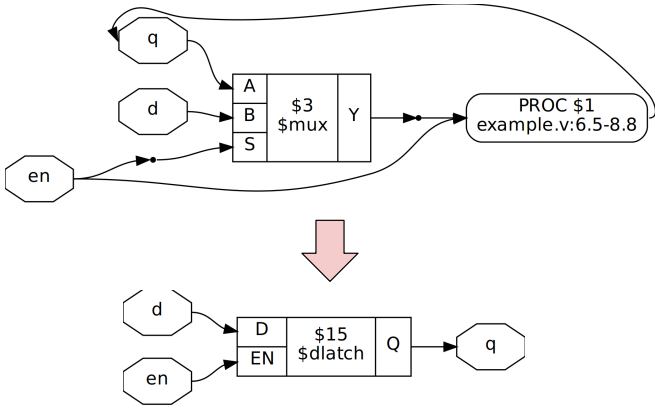
Extracts D latches from combinational PROC blocks if required. If a latch is not required, it directly connects the reg to the expression value (RHS) using a wire. Consider the following examples. The circuits shown are after proc_mux and then after the rest of the proc script

Example 1 : explicit latch

```
always @(*) begin
     if(en)    q = d;
     else      q = q;
end
```

Example 2 : implicit latch

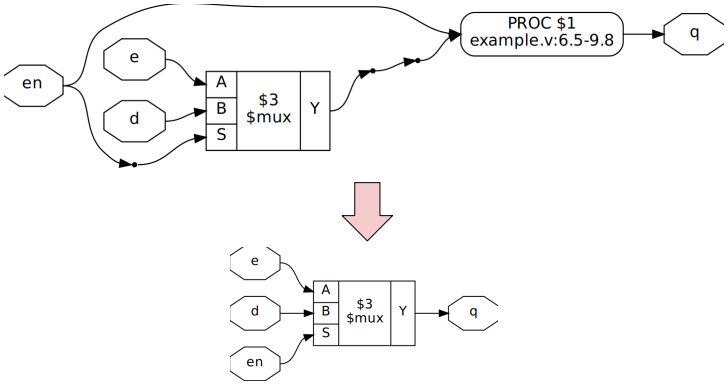
```
always @(*) begin
     if(en)    q = d;
end
```



Example 1 & Example 2 (equivalent)

Example 3 : no latch

```
always @(*) begin
     if(en)    q = d;
     else      q = e;
end
```



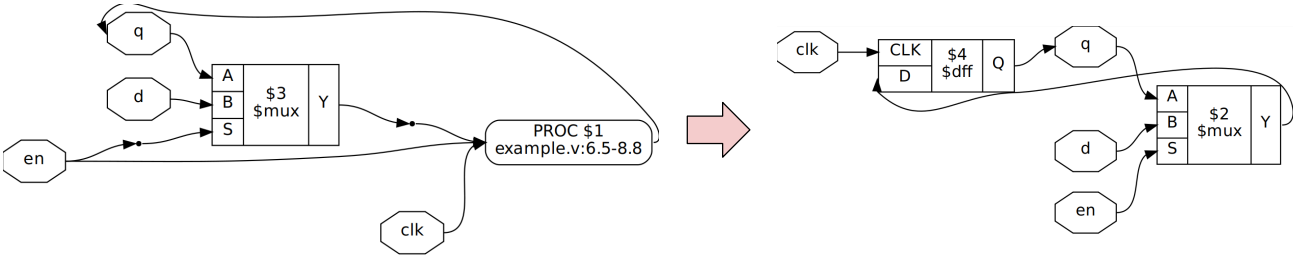
Example 3

proc_dff : Extracting D flip-flops

Extracts D flip-flops from clocked PROC blocks. In general, all LHS regs in a clocked always block are passed through a flip-flop. If an async set/reset was detected, an appropriate reg variant is used. proc_mux extracts the multiplexers that implement synchronous reset and enable, and proc_dff extracts the flip flops.

Example 1 : implicit enable FF

```
always @(posedge clk) begin
     if(en)    q <= d;
end
```

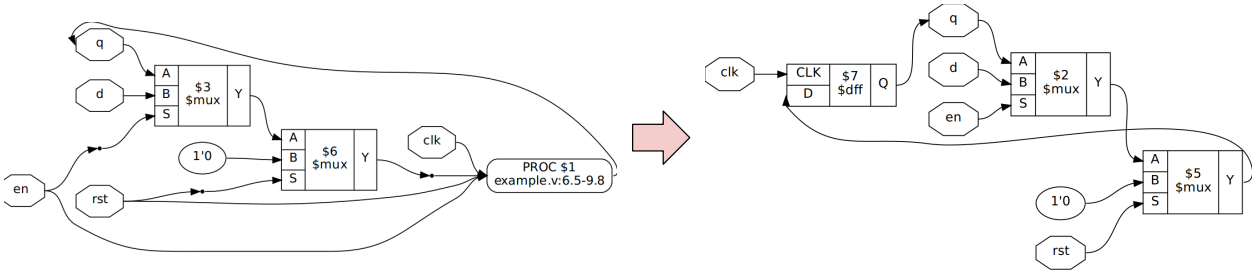


Example 2 : implicit enable FF

```
always @(posedge clk) begin
     if(en)    q <= d;
     else      q <= q;
end
```

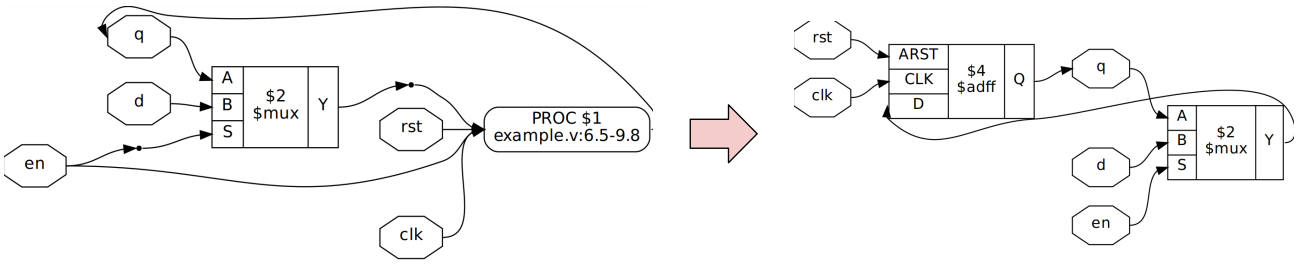
Example 3 : Synchronous reset

```
always @(posedge clk) begin
     if(rst)        q <= 0;
     else if(en)    q <= d;
end
```



Example 4 : Asynchronous reset

```
always @(posedge clk, posedge rst) begin
     if(rst)        q <= 0;
     else if(en)    q <= d;
end
```



Examples 1 and 2 result in the same circuit from the start (after read_verilog). Comparing Example 3 and Example 4, it should be noted that the if condition for the asynchronous reset was not extracted by proc_mux since it was detected as an asynchronous reset by proc_arst. An asynchronous reset flip-flop ($adff) was synthesized instead of a synchronous flip-flop ($dff).

proc_memwr : Extracting memory writes

Memory writes in Yosys are separate blocks. This pass extracts such blocks if a memory address is written inside a PROC block. We will look at how memory accesses are synthesized in another section.

Summary

To summarize, every behavioral block is synthesized in mainly 3 stages :

1) Extract the datapaths, i.e, the RHS of the expressions (read_verilog)
2) Extract the multiplexer trees for conditional execution (proc_mux)
3) Extract latches or flip-flops (proc_dlatch, proc_dff)

Asynchronous resets and memory writes are extracted separately.

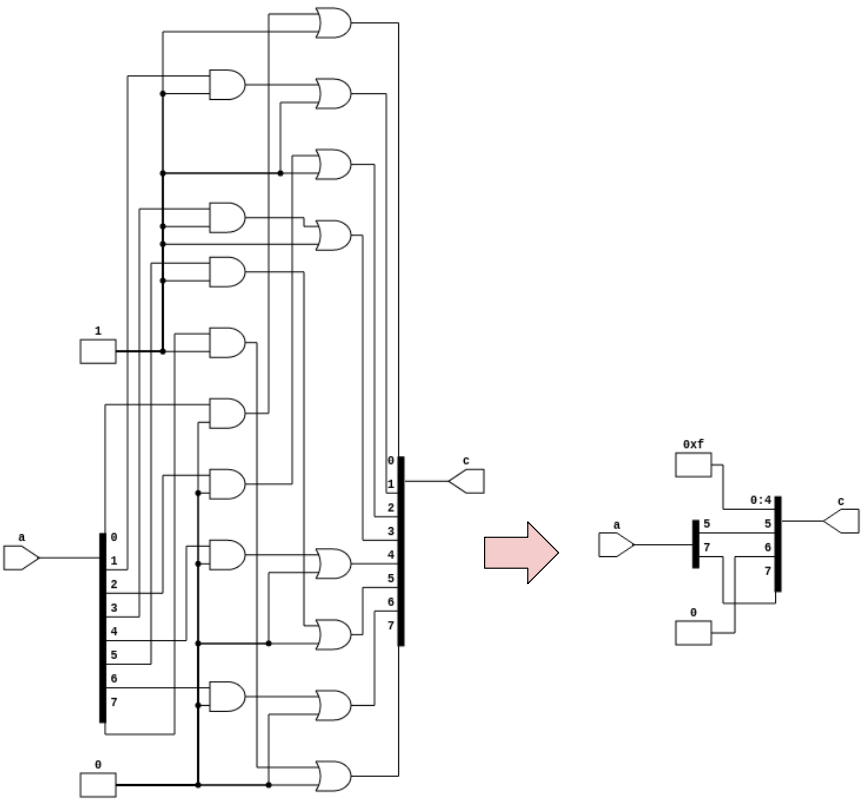
Section IV : Circuit Optimizations

Yosys utilizes a number of optimizations to clean up the circuit, remove redundant operations and generate better results. These are simple rule-based optimizations, unlike what a boolean optimization engine like ABC would do. This section covers some such optimization passes

opt_expr : Expression optimization

Optimizes expressions and removes operators. For example, “a | 0 = a”, “a & 0 = 0” and so on. Yosys has a list of such simple logic reduction rules, which we all learn in our basic digital design courses, as tables for each RTL block. Below is one example which uses a bit mask with AND and OR operations. This will be converted to wire connections.

```
assign c = (a & 8'b10101010) | 8'b00001111;
```

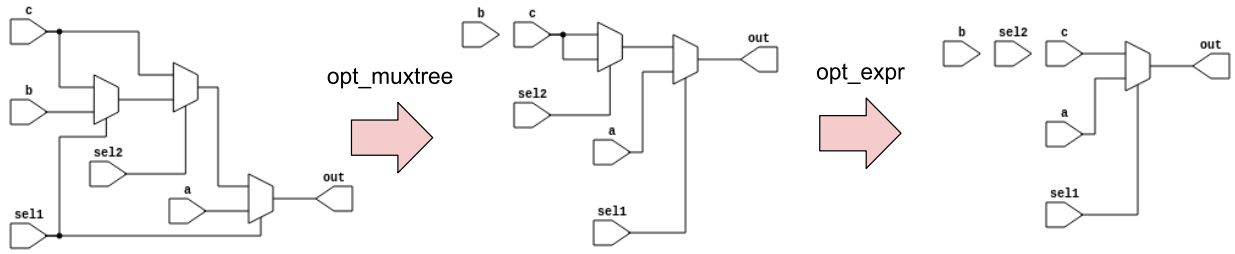


Circuit before and after opt_expr

opt_muxtree : Multiplexer tree optimizations

Optimizes trees of multiplexers by analyzing the select inputs. Some nodes in the tree could be eliminated altogether because their combination of select signals would not be triggered. Let us look at the conditional expression below. We can reason that if the output of the sel2 multiplexer is to be routed to the output, sel1 should be 0, and hence the last MUX is useless. opt_muxtree figures out such patterns in trees of multiplexers. We do need to use opt_expr to eliminate a mux with both inputs being the same.

```
assign out = (sel1 ? a : (sel2 ? (sel1 ? b : c) : c));
```



Simplifying the circuit using opt_muxtree and then opt_expr

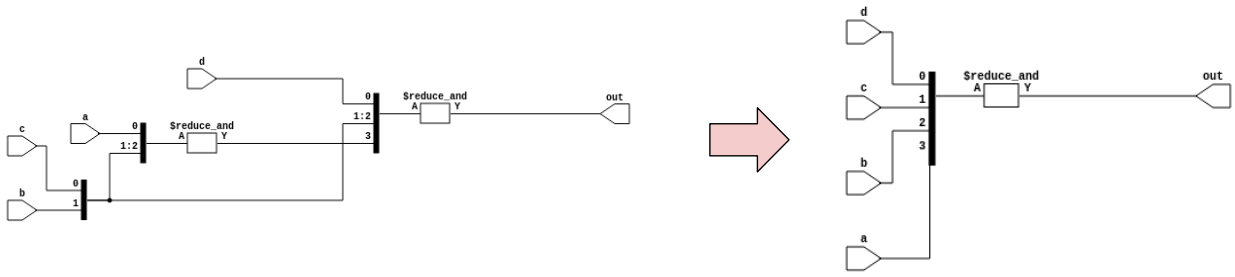
opt_reduce : Optimize trees of reduction operations

Analyze reduction AND and OR blocks and check if their inputs are identical to remove them. For example,

```
assign out = &{&{a, b, c, a}, b, c, d};
```

is equivalent to

```
assign out = &{a, b, c, d};
```

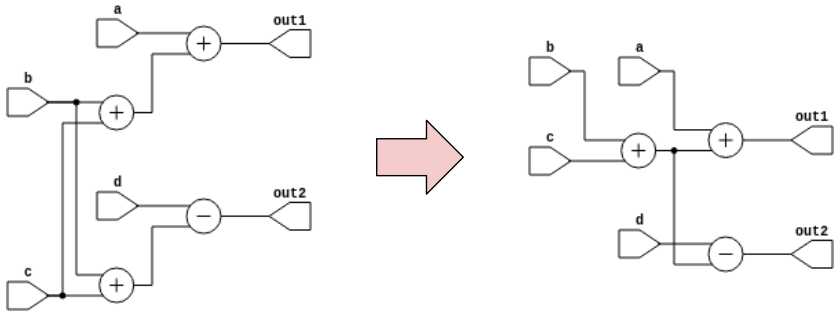


Circuit optimized after opt_reduce

opt_merge : Merge identical cells

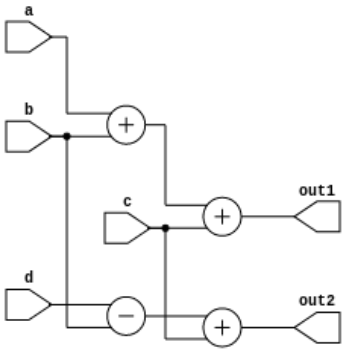
Merges redundant bocks by identifying identical blocks with the same inputs. For example,

```
assign out1 = a + (b + c);
assign out2 = d - (b + c);
```



Resource sharing by opt_merge

However, it should be noted that this does not happen if the operations are ordered differently because opt_merge does not have the logic to move RTL blocks. For example, if the brackets are removed, the initial circuit is inferred with the subtraction done first for out2, and opt_merge can not do anything. It only identifies identical blocks with identical inputs. ABC is more powerful but opt_merge does its job as part of a simple clean-up optimization run

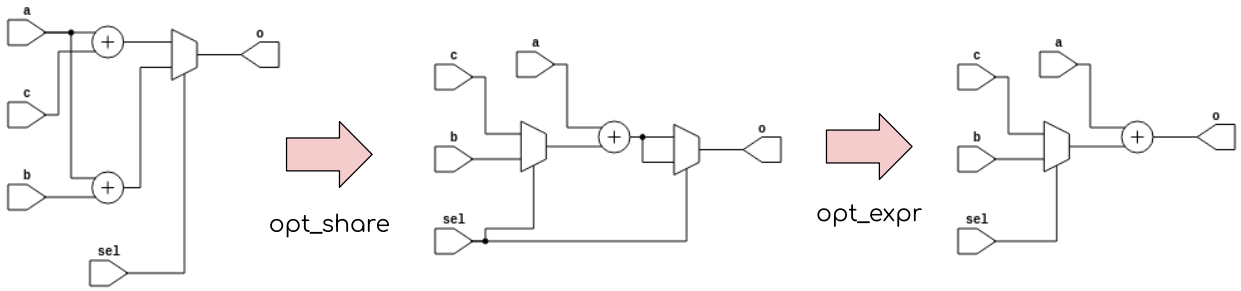


opt_merge does not change the order of evaluation

opt_share : Share resources

Identifies identical cells with at least one common input and whose output is multiplexed. This multiplexer can be moved before the cell to share the cell. For example, consider the design below.

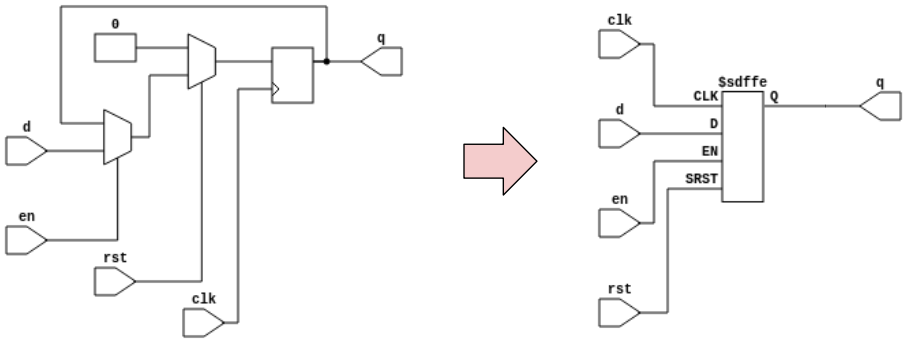
```
assign o = sel ? (a + b) : (a + c);
```



Resource sharing using opt_share and opt_expr

opt_dff : Merge and remove flip-flops

Optimizes away constant value flip flops and merges reset and enable logic into the flip flop. For example, in the logic below, since enable and reset are implemented as multiplexers before a DFF, these will be merged to form a different flip-flop variant in this opt pass. The below example shows the merging of en and rst signals into the flip flop.



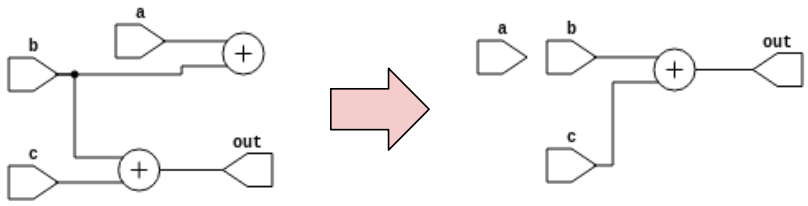
Merging reset and enable logic using opt_dff

opt_clean : Remove unused signals and cells

Some passes may leave unconnected logic or wires which are not used anywhere. Such cells and wires are removed by opt_clean. For example,

consider the code below, where out is the only output of the module, and temp is not used.

```
wire[7:0] temp;

assign temp = a + b;

assign out  = b + c;
```



Removing unused logic with opt_clean

opt : Iteratively optimize

Runs all the other optimizations in a sequence and repeats them till the circuit does not change in the loop.

```
# Run only once
opt_expr
opt_merge -nomux

# Run till no more optimizations are possible
do
    opt_muxtree
    opt_reduce
    opt_merge
    opt_share
    opt_dff    # Except when called with -noff
    opt_clean
    opt_expr
while <changed design>
```

Section V : Technology Mapping

Technology mapping involves transforming the RTL netlist into the target library. This is done in two steps. First, the circuit is converted to a standard set of gates by simply substituting the RTL blocks. Then, another stage transforms this netlist into the target technology (LUTs in FPGAs or cells in ASIC design).

In Yosys, the first transformation is done to a standard internal gate library using the **techmap** command. Then an optimizer called **ABC** is used to transform this into the target library and optimize the circuit.

techmap : Substituting RTL blocks

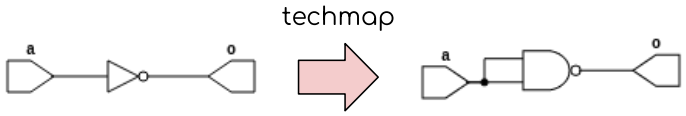
The first step i.e. RTL synthesis, involves generating individual gates for higher-level RTL blocks like vectorized gates, adders, multipliers, etc. The techmap pass is used for this. Techmap simply maps a block into another structure made of simpler blocks that can be further mapped down.

```
techmap -map <map_file>
```

The option passed is a “technology map” file with the rules for mapping the structures. By default, it uses the file techlibs/common/techmap.v. The techmap command reference explains the format of a techmap file.

The map file is a verilog file where each module is a “rule”. The module attributes specify which blocks (according to name and type) will match the rule, and these blocks are replaced with the contents of the verilog module. For example, the rule below translates an inverter into a NAND gate.

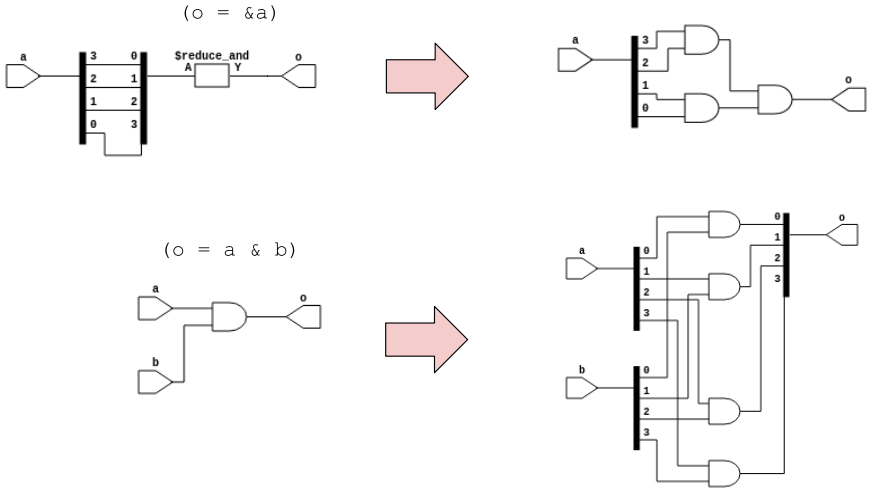
```
(* techmap_celltype = "$_NOT_" *)
module nand_not (input A, output Y);
    $_NAND_ nand_inst(.A(A), .B(A), .Y(Y));
endmodule
```



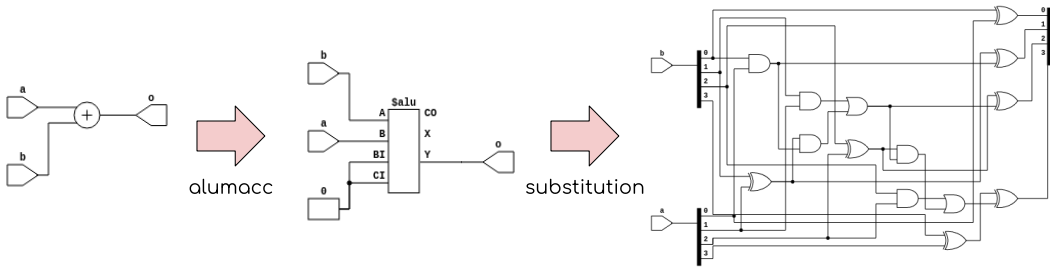
The rules are often parameterized and generate statements are used inside the module to make the rule work for different port widths and even different RTL block types. The rules can be nested, and techmap will iteratively substitute blocks until no more blocks can be substituted.

Some common mappings are implemented as Yosys passes.

The **simplemap** pass transforms simple RTL operations like boolean operators and buffers. The transformation of 4-bit reduce AND and bitwise AND is shown below. It can be seen how the 4 AND gates were split up from the vectorized bitwise AND gate. This pass performs such simple mappings.

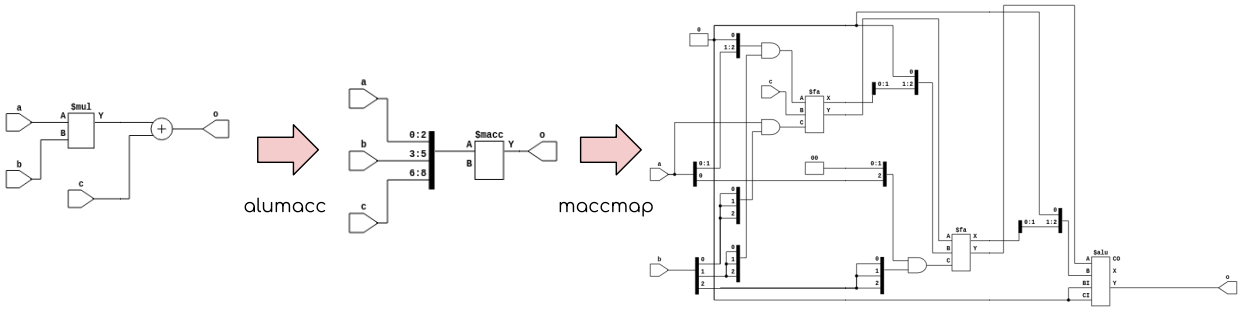


The **alumacc** pass converts adders, subtractors, comparators, etc, to ALUs and multipliers to MACC blocks. The example below shows the transformation of a 4-bit adder to an ALU block and then to gates using the technology map. The ALU block is a combinational block with two data inputs and a carry and borrow input. The outputs are the output and the propagate and carry bits. Subtractors and comparators are also transformed to the ALU unit when passed through this.



The technology map file describes how the ALU block will be implemented. By default, it uses a **brent-kung tree adder**. For FPGAs it would use the carry tree or DSP block.

The **maccmap** pass converts MACC units to adders and ALU blocks. In FPGAs, MAC (multiply-accumulate) units can be implemented using DSP slices. If such dedicated blocks are not available, we would need to run this pass. A multiplier (or MAC unit) is implemented in 3 stages: **partial product generation**, **adder tree**, and the final **ripple adder**. The example below is for a 3-bit MAC unit where the bitwise AND generates the partial products, the full adders ($fa) for the adder tree, and the ALU for the ripple addition.



By using these two passes, any structure that can utilize an ALU, like comparators, adders, subtractors, or multipliers, will be converted to ALU blocks which can either be implemented using custom hardware (FPGA carry chains or DSP units) or be synthesized to adders using mapping rules.

In the default techmap, the ALU is mapped to a Brent-kung tree adder. Take a look at the default techmap to see how this rule is defined.

Synthesis scripts typically use multiple technology mapping files to map different types of blocks and optimize the circuit for their technology. For

example, techlibs/common/gate2lut.v defines how standard gates will be substituted to Look-up tables.

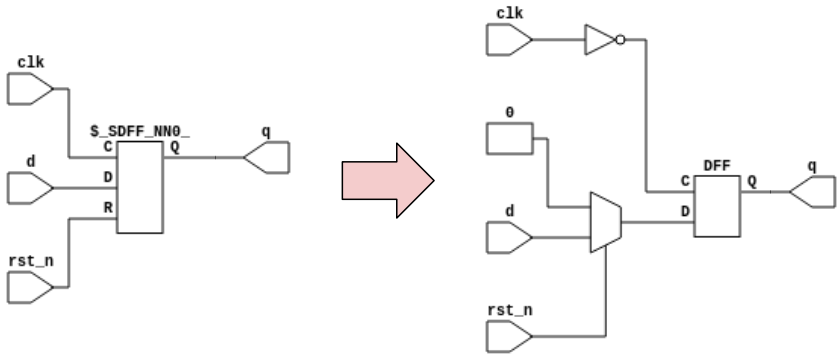
dfflibmap : Register technology mapping

The internal library has all combinations of flip-flops with different set, reset, and clock attributes. However, the final library may only have a small subset of these. The dfflibmap pass transforms the flip-flops from the internal library to the final tech library. The final technology library is specified using a “liberty” format file.

```
dfflibmap -liberty library.lib
```

dfflibmap renames flip flops with identical functionality and maps the other flip flops in the design by adding additional logic to create equivalent versions that can be implemented. For example, the code below that models a negedge flip flop with negative synchronous reset is transformed to a normal DFF when we give a library with only a simple DFF (refer appendix to see the library used).

```
always @(negedge clk)
     if(~rst_n)     q <= 0;
     else           q <= d;
```



The reset and clock logic were merged to form a specialized flip-flop by running opt_diff. This is reversed by dfflibmap since an equivalent synchronous reset flip-flop was not available.

abc : Combinational technology mapping

ABC is a standalone logic synthesis engine that Yosys uses for technology mapping. It is a collection of algorithms that can manipulate logic with area and delay optimization. It is an incredible open-source project that can have a separate article dedicated to it on its own. You can learn more about ABC here.

ABC is integrated into Yosys as a pass. It reads the technology library from a ".lib" (liberty) file, which describes the cells available for implementation and their timing, power, and area. This is passed using the "-liberty <lib file>" option.

Timing constraints are given with max delay with the "-D <picosecs>" option and a constraint file with input driver and output load information with the "-constr <constraint file>" option.

```
abc -liberty <lib_file> -D <delay> -constr <constr_file>
```

The ABC pass has some options for playing around with standard sets of gates or LUTs, which we can use to explore synthesis without an actual tech library.

```
abc -lut <w1>:<w2>
```

Use LUTs of widths w1 to w2 (inclusive). All LUTs with width <= <w1> have constant area cost. For LUTs larger than <w1>, the area cost doubles with each additional input bit. the delay cost is constant for all lut widths.

```
abc -g type1,type2,...
```

With this option, we can list the individual gates that can be used. The supported gates are AND, NAND, OR, NOR, XOR, XNOR, ANDNOT, ORNOT, MUX, NMUX, AOI3, OAI3, AOI4, and OAI4. We can also specify a set of gates using shortforms as shown below. By default, all these gates are used. We can use this to see how our code translates to all NAND gates, for example.

We can run techmap and then abc to remap circuits between libraries, from say LUTs to CMOS or vice-versa

Name	Gates used
simple	AND OR XOR MUX
cmos2	NAND NOR
cmos3	NAND NOR AOI3 OAI3
cmos4	NAND NOR AOI3 OAI3 AOI4 OAI4
cmos	NAND NOR AOI3 OAI3 AOI4 OAI4 NMUX MUX XOR XNOR
gates	AND NAND OR NOR XOR XNOR ANDNOT ORNOT
aig	AND NAND OR NOR ANDNOT ORNOT

extract : Subcircuit substitution

If the technology has more complicated cells available, the extract command can be used to detect and replace such subcircuits that match the available cell.

For example, if we have a hard IP that finds the maximum of two numbers, and we build a circuit that computes the maximum of 4 numbers as follows, the extract tool can find the subcircuits that can be replaced with the IP. For most cases, however, the MACC and ALU extraction from techmap should work well enough.

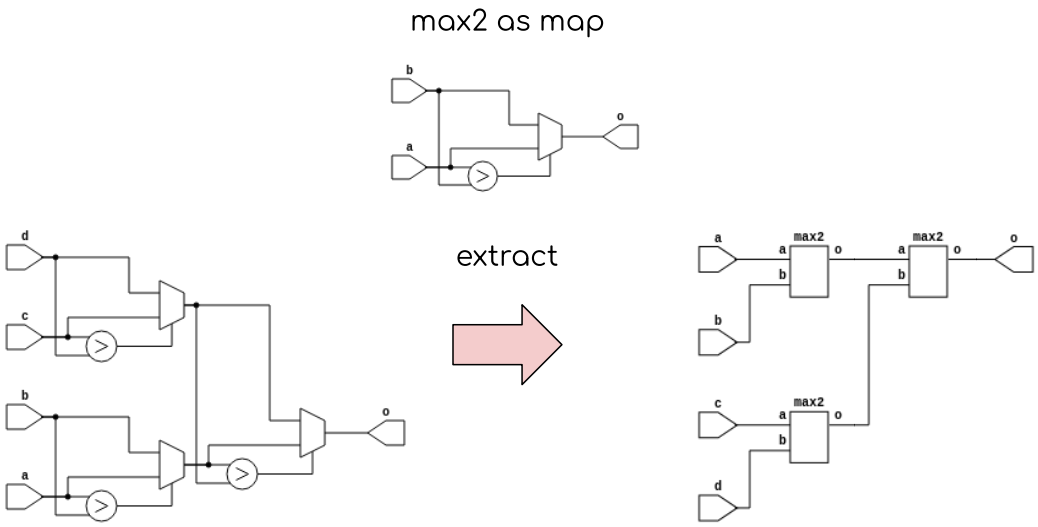
```
module max4(
    input[3:0]      a, b, c, d,
    output[3:0]     o
);

    wire[3:0] t1, t2;

    assign t1 = a > b    ? a  : b;
    assign t2 = c > d    ? c  : d;
    assign o  = t1 > t2  ? t1 : t2;

endmodule
```

The logic to find the maximum of 2 values that can be written inside another module called mux2 as:

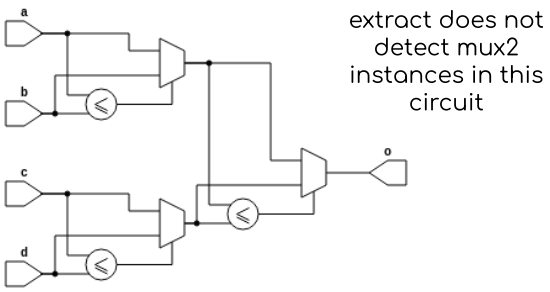
```
assign o = a > b ? a : b;
```

Then, if we run the following command to extract instances of the template, it instantiates three such modules.

```
extract -map max2.v
```



However, this does not work if the map design does not exactly match the structure in the target design. For example, if we write the mux4 module using <= instead of > and swap the mux inputs, the functionality remains the same, but the extract pass will not be able to extract the subcircuits.



Section VI : Memory Inference

One topic we have avoided till now is the inference of memory blocks. While writing HDL, memory blocks like SRAM and ROM blocks can either be **structurally instantiated** as modules which are then compiled, or **'inferred'** by the tool where we write behavioral code that models the desired type of memory, and the synthesis tool instantiates the memory of the required size.

Now, let us see how Yosys synthesizes such memory. Since memory inference is typically used in FPGA flows, we will look at this from an FPGA synthesis perspective. For ASIC designs, structural instantiation is preferred, and a dedicated "**memory compiler**" is used to generate such memory blocks individually.

Types of memory resources and inference rules

In FPGAs, there are four types of memories available. Each of these have their own uses and are intended to be used in specific situations.

LUT RAM

- Write ports : One
- Read ports : Any number, asynchronous
- Implemented using LUT resources as memory
- Custom initialization and reset values

Flip-flop RAM

- Write ports : Any number in the same clock domain
- Read ports : Any number, asynchronous
- Implemented using FFs, decoders, multiplexers
- Custom initialization and reset values

Block RAM

- Two ports with independent clocks that can both read / write
- Synchronous reads
- Implemented as dedicated IP from an SRAM array
- Custom initialization values, but contents cannot be reset

Huge RAM

- Single port that can be used for read / write (exclusive read/write)
- Synchronous reads
- Implemented as dedicated IP from an SRAM array
- All values are 0 at initialization, and no reset

Each of these is intended to be inferred in specific situations. In general, the automatic selection process follows this :

- If any read port is asynchronous, only LUT RAM (or FF RAM) can be used.
- If there is more than one write port, only block RAM can be used
    - FF-RAM can also be used for multiple writes but only in the same clock domain, but is more expensive
- Otherwise, either FF RAM, LUT RAM, or block RAM will be used, depending on memory size

This process can be overridden by attaching a ram_style attribute to the memory. If the synthesis program cannot realize the intended type of logic, it will throw an error.

Flip-flop RAM : `(* ram_style = “logic” *)`

LUT RAM : `(* ram_style = “distributed” *)`

Block RAM : `(* ram_style = “block” *)`

Huge RAM : `(* ram_style = “huge” *)`

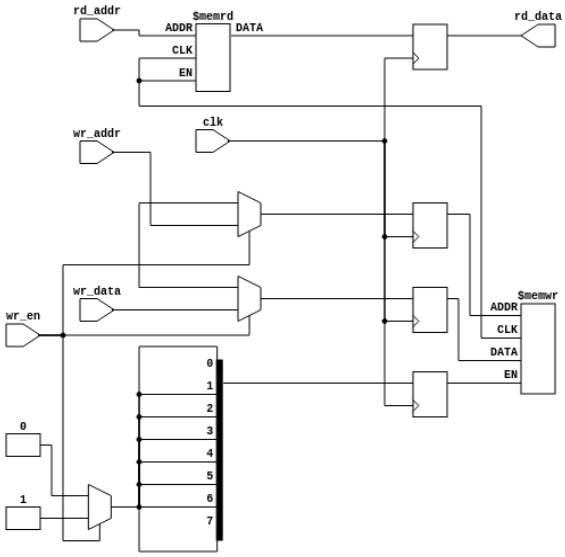
Memory access blocks

Memory read / write accesses and initialization (using $readmemb, etc) are converted to separate blocks in the behavioral synthesis stage. Consider a synchronous read RAM as given below

```
reg[7:0]  mem[255:0];
     assign rd_data = mem[rd_addr];
```

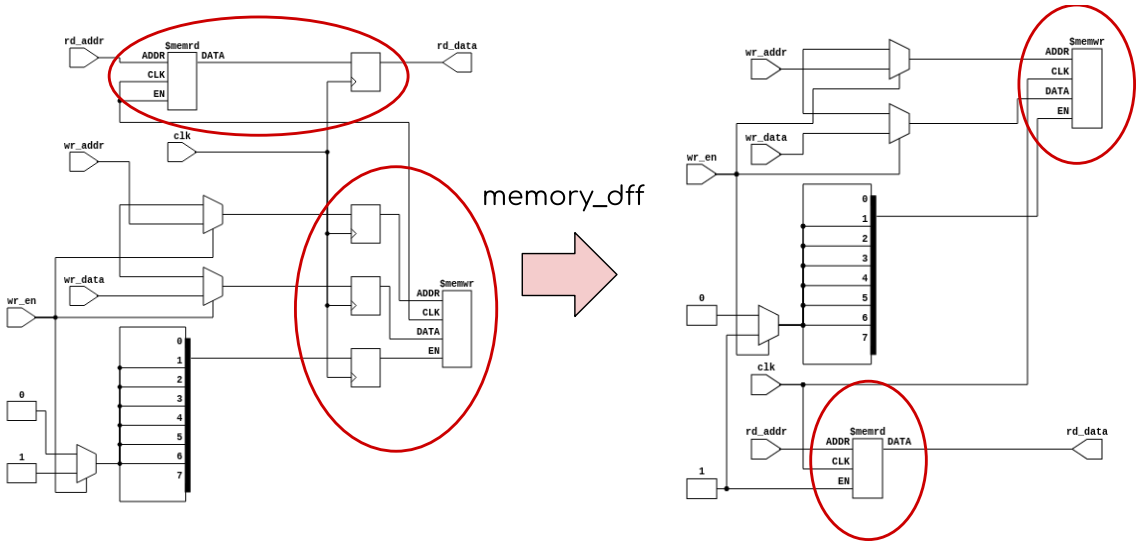
```
always @(posedge clk) begin
   rd_data <= mem[rd_addr];
   if(wr_en)
         mem[wr_addr] <= wr_data;
end
```

The output of this code after proc is shown below. Each read and write has been synthesized to registers and memory blocks ($memrd and $memwr). The clock signals of these are unknown (x) to indicate asynchronous nature. Note that wr_addr and wr_data are also multiplexed by wr_en, which selects unknown (x) when wr_en is low. Unknown signals show up as unconnected in the diagram. The EN signal to $memwr is a bitwise write enable.



memory_dff : Merging DFFs into synchronous ports

Synchronous reads and writes are synthesized to D flip flops by the proc pas. The memory_dff pass merges these D flip flops into the read and write blocks by making the ports synchronous. The CLK signal of the read and write ports are set to the CLK input of the D flip flops.



memory_share : Merge memory ports

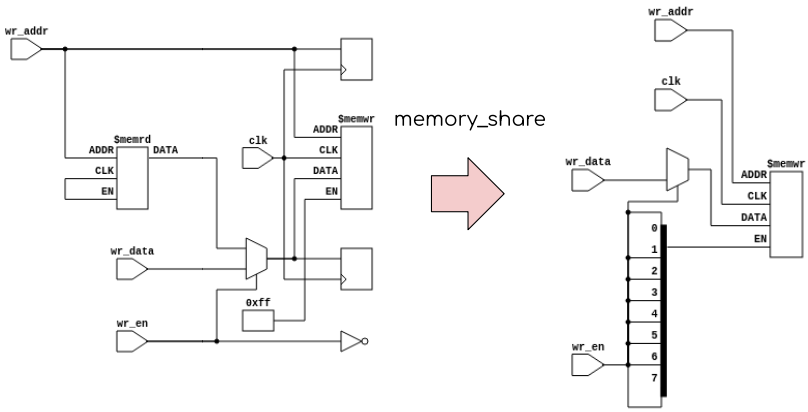
Some reads and writes can be merged to a single port in some conditions. This pass identifies such cases and merges them. These include trivial optimizations like merging blocks with the same exact inputs and accessing the same memory. There are other non-trivial patterns as well, like:

1) Feedback from async read to write port for implementing enable

When write ports are connected to async read ports accessing the same address, then this feedback path is converted to a write port with byte/part enable signals. We write this sort of code to mean “write back the same data we read if enable is low”. We usually write this for byte-wise enable signals.

The feedback path must contain only multiplexers. Writing multiplexer using other logic doesn't work!

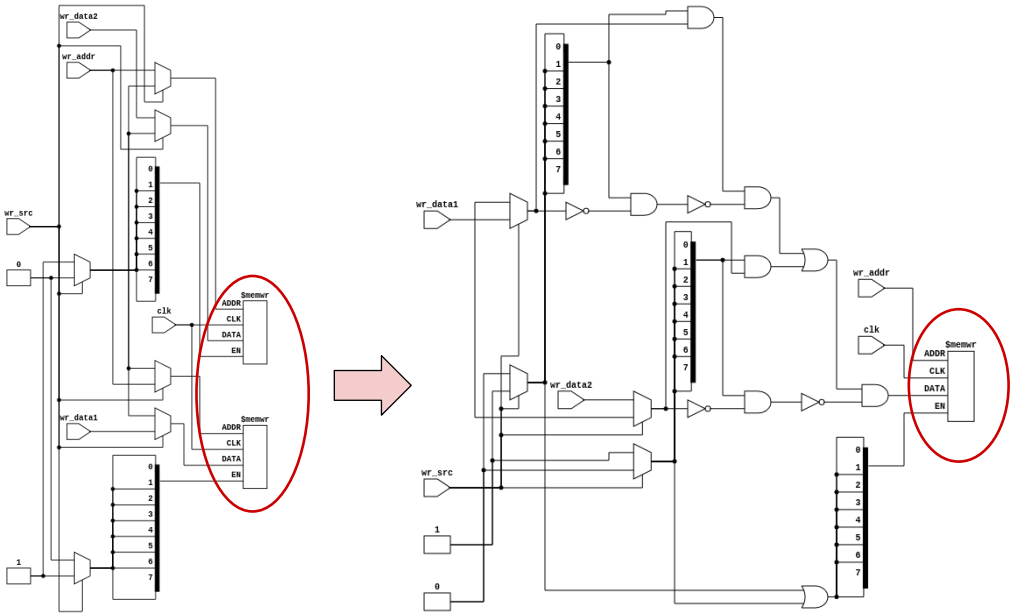
```
wire[7:0] rd_temp = mem[wr_addr];
wire[7:0] wr_temp = wr_en ? wr_data : rd_temp;

always @(posedge clk) begin
     mem[wr_addr] <= wr_temp;
end
```



2) Writes to the same address

When multiple write ports access the same address, typically, only one of them does it at a time (exclusive enable signals). Such ports are converted to a single write port with more complex data and enabled logic paths. For example, if we had code, as shown below, for writing from one of two separate sources, these writes could be merged using a multiplexer.

```
always @(posedge clk) begin
     if(wr_src)
          mem[wr_addr] <= wr_data1;
     else
          mem[wr_addr] <= wr_data2;
end
```

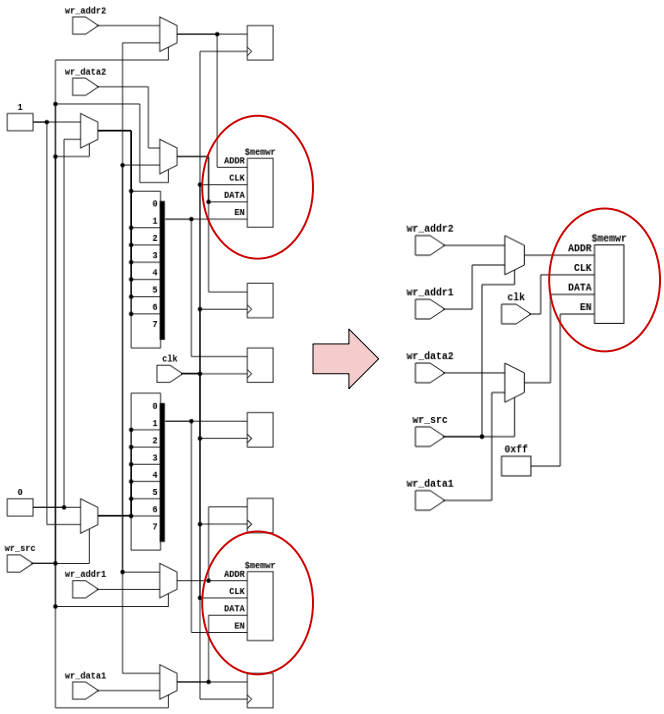


Sure, we could've written it as a MUX in the first place, but in some cases (like inside a case statement), this might be more natural. The final circuit also forms a multiplexer but in an unusually long-winded way. In general, this is not recommended, but Yosys is able to detect that these two writes can (and should) be merged since they are to be the same address and are exclusive (not enabled at the same time).

3) Independent writes

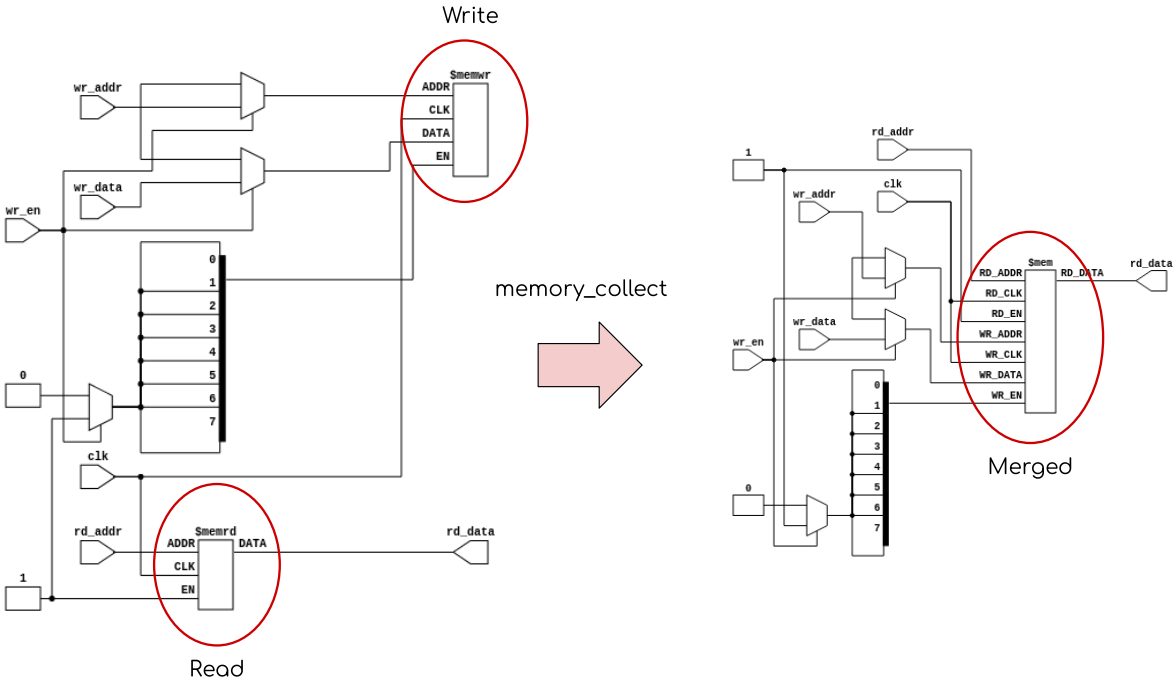
When multiple write ports are never accessed at the same time, then the ports are merged into a single write port. A boolean SAT solver mathematically proves that the two enable signals are exclusive. We can consider the same example as before but with two different address writing styles.

```
always @(posedge clk) begin
     if(wr_src)
          mem[wr_addr1] <= wr_data1;
     else
          mem[wr_addr2] <= wr_data2;
end
```



memory_collect : Merge ports to the same memory

Till now, we had independent ports for each read and write. This pass merges all accesses to the same memory into a single MEM block with all the memory accesses to that memory.



memory_bram : Detect BRAM instances

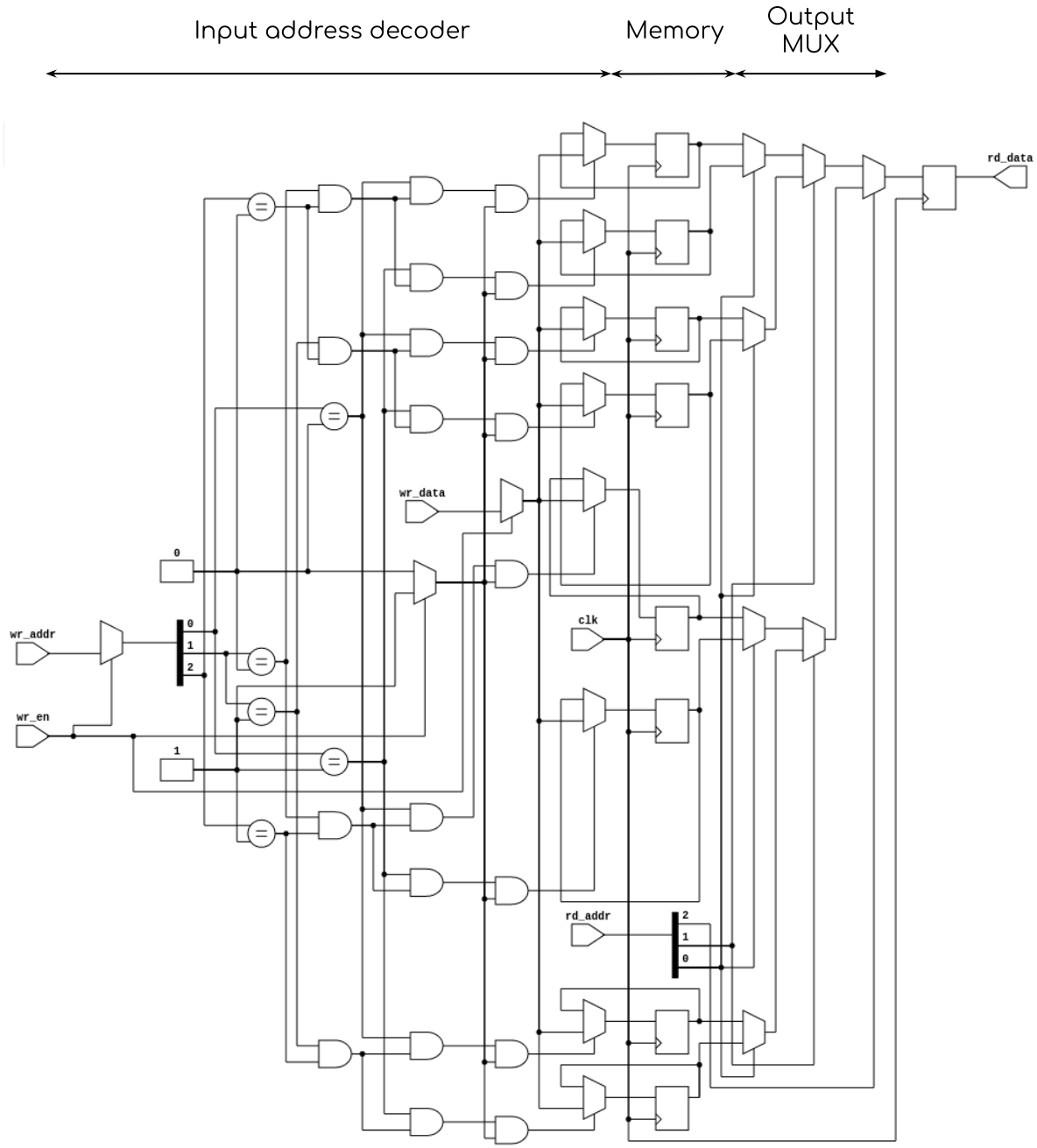
This pass checks if the MEM blocks can be mapped to block RAMs. This is done using a rules file that describes the properties of the BRAM resources available and in what conditions they can be matched to MEM blocks. Such conditions include address and data port widths, number of instances (for multiple read ports, BRAMs are duplicated), number of used data bits, etc, to ensure that BRAM mapping happens only when the MEM block is large enough.

memory_libmap : Map MEM blocks to available templates

This pass also maps suitable MEM blocks to those from a library. However, it can also map to distributed RAM and is used frequently in FPGA flows.

memory_map : Map MEM blocks to flip flops and logic

This pass is run after memory_bram or memory_libmap to generate memory using flip flops, decoders, and multiplexers. This is the most expensive and hence, least preferred type of memory to be implemented and therefore is run in the end to clean up memory instances that could not be inferred into any other kind. For example, the following is the output for an 8-word 8-bit synchronous read memory.



memory : Run all memory mapping passes in order

This pass runs all the other memory mapping passes one by one in an appropriate order. It runs the following commands. There are options to omit memory_map so that memory_libmap can be run instead and map RAM blocks to distributed RAM or other types of RAM instead of FF RAM where possible.

```
# Some optimizations
opt_mem
opt_mem_priority
opt_mem_feedback

# Merge sync read/write registers and ports
memory_bmux2rom     # Detects ROM blocks from multiplexers
memory_dff          # Merge DFFs to synchronous ports
opt_clean
memory_share        # Share ports where possible

# More optimizations
opt_mem_widen       # Optimize the memory
opt_clean           # Remove unused wires and blocks

# Merge the ports and map to BRAM / FF RAM
memory_collect      # Merge ports into MEM block
memory_bram -rules <bram_rules>     # Extract BRAM
memory_map          # Convert rest to FFs and logic
```

Section VII : Finite State Machines

Most digital circuit classes would teach optimizing FSMs using state reduction and state assignments. I for sure did not enjoy this process much. To make this easier, all modern synthesis tools can detect and optimize FSMs. Yosys, is a bit limited here and can only recode states to a one-hot encoding but nevertheless, let us look at how Yosys detects and optimizes FSMs.

fsm_detect : Detect FSM state registers

This pass detects and flags registers corresponding to FSM states. This should be done **at the RTL stage**, i.e. after proc and before techmap. This is done using a straightforward rule stated as follows:

"The input to the D flip flop is driven by a multiplexer tree with only the previous state or constants as inputs."

This is quite effective because this is usually how we write FSMs in HDLs. The next-state logic is coded using if or case statements (which synthesize to multiplexers), which choose between the previous state and another state. There are additional rules so that state recoding won't impact the remaining circuit.

1. The state wire must not be an output of the module (cant recode)
2. The state wire must only be used in the MUX tree for next-state logic, or by simple relational cells like equals

The command outputs a text message when a state variable is detected. For example, we will use the verilog code below as a reference throughout this section.

It should be noted that FSM states with initial values cannot be detected or optimized! It seems like some bugs caused this feature to be removed.

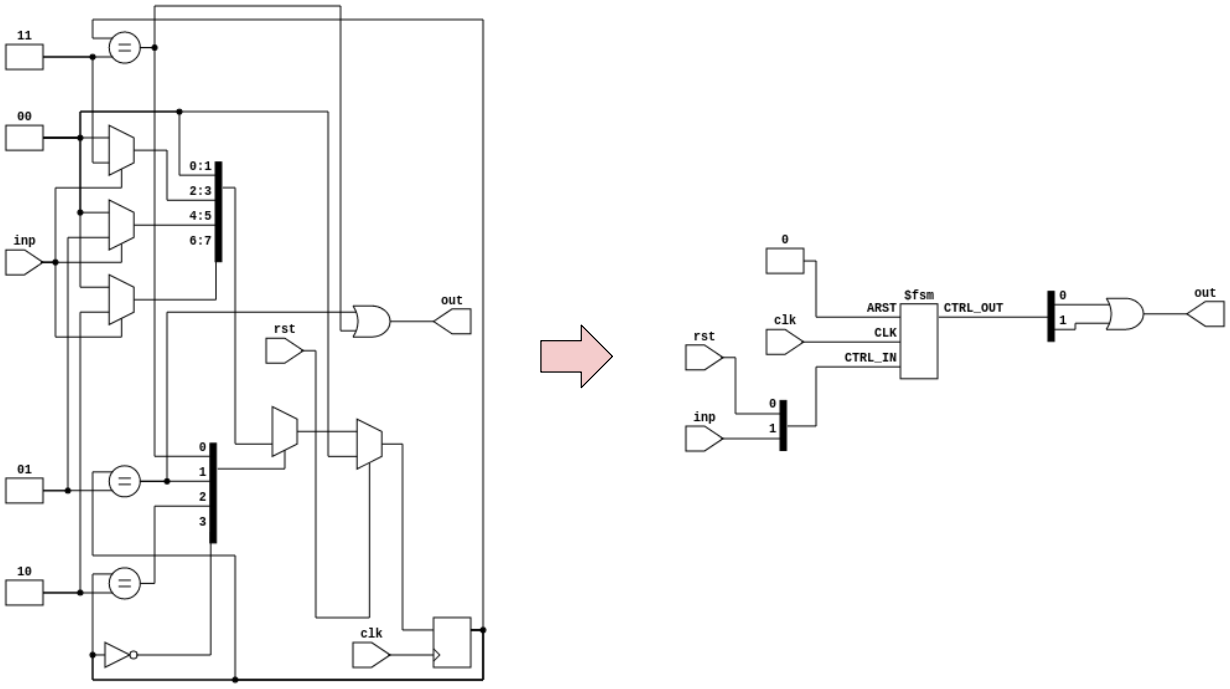
fsm_extract : Extract FSM state, next state logic and outputs

This pass extracts the state registers detected by fsm_detect and the multiplexer tree logic into an FSM block. This basically separates the FSM from the rest of the circuit so that it can be optimized independently.

For example, let us consider the code for a simple state machine as in the following code:

```
module test(
    input clk,
    input rst,

    input inp,
    output out
);

    localparam s0 = 2'b00;
    localparam s1 = 2'b10;
    localparam s2 = 2'b01;
    localparam s3 = 2'b11;

    reg[1:0] state;

    assign out = (state == s2 | state == s3);

    always @(posedge clk) begin
        if(rst)
            state <= s0;
        else begin
            case(state)
                s0: state <= inp ? s1 : s0;
                s1: state <= inp ? s2 : s0;
                s2: state <= inp ? s3 : s0;
                s3: state <= s0;
            endcase
        end
    end

endmodule
```

The circuits after proc and then after **fsm_detect** and **fsm_extract** (and opt) are shown below. We can see that the registers, the next state logic, and part of the output logic have been absorbed into a **$fsm** block.



fsm_info : Print fsm info

```
FSM `$fsm$\state$17' from module `\test':
-----------------------------------

  Information on FSM $fsm$\state$17 (\state):

  Number of input signals:    2
  Number of output signals:   6
  Number of state bits:       2

  Input signals:
    0: \rst
    1: \inp

  Output signals:
    0: $eq$example.v:16$1_Y
```

```
    1: $eq$example.v:16$2_Y
    2: $0\state[1:0] [0]
    3: $0\state[1:0] [1]
    4: $procmux$11_CMP
    5: $procmux$12_CMP

  State encoding:
    0:       2'00  <RESET STATE>
    1:       2'10
    2:       2'01
    3:       2'11

   Transition  Table  (state_in,  ctrl_in,  state_out,
ctrl_out):
      0:     0 2'00   ->     0 6'100000
      1:     0 2'10   ->     1 6'101000
      2:     0 2'-1   ->     0 6'100000
      3:     1 2'00   ->     0 6'010000
      4:     1 2'10   ->     2 6'010100
      5:     1 2'-1   ->     0 6'010000
      6:     2 2'00   ->     0 6'000001
      7:     2 2'10   ->     3 6'001101
      8:     2 2'-1   ->     0 6'000001
      9:     3 2'-0   ->     0 6'000010
     10:     3 2'-1   ->     0 6'000010

-------------------------------------
```

fsm_opt : Remove unnecessary outputs from FSM block

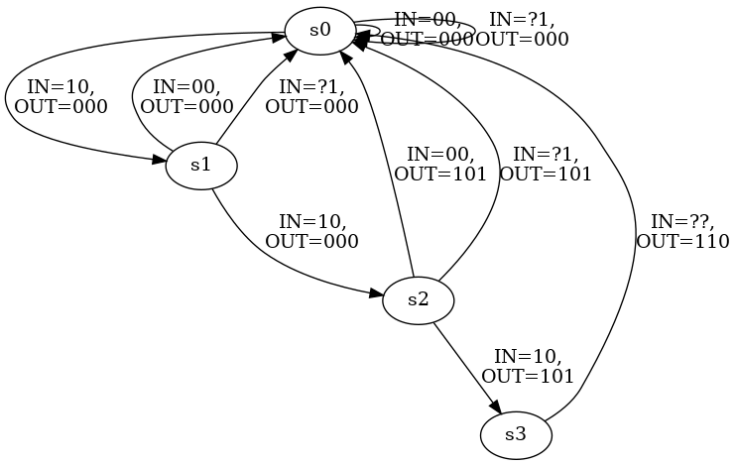
This pass performs basic cleanup of the FSM block. The FSM block from the fsm_extract pass also gives the FSM states as an output port, which is unused (and hence don't show up in the diagram). This port is removed using the fsm_opt pass and then the opt_clean pass to clean up unused wires and cells.

fsm_export : Export FSM info into KISS2

Exports the FSM and its states and transitions into a file format called “KISS2”, which is an ASCII representation of a state transition table. This can be

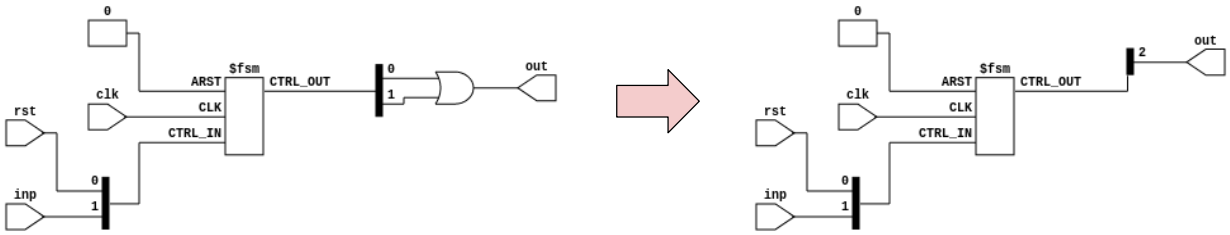
used to generate a state diagram of the extracted FSM, for example. We will see how to generate such a diagram in a later section.

```
fsm_export -o out.kiss2
```



fsm_expand : Expand FSM block to other outputs

This pass expands the FSM block to include more logic. In our case, the OR gate is also included in the $fsm block.



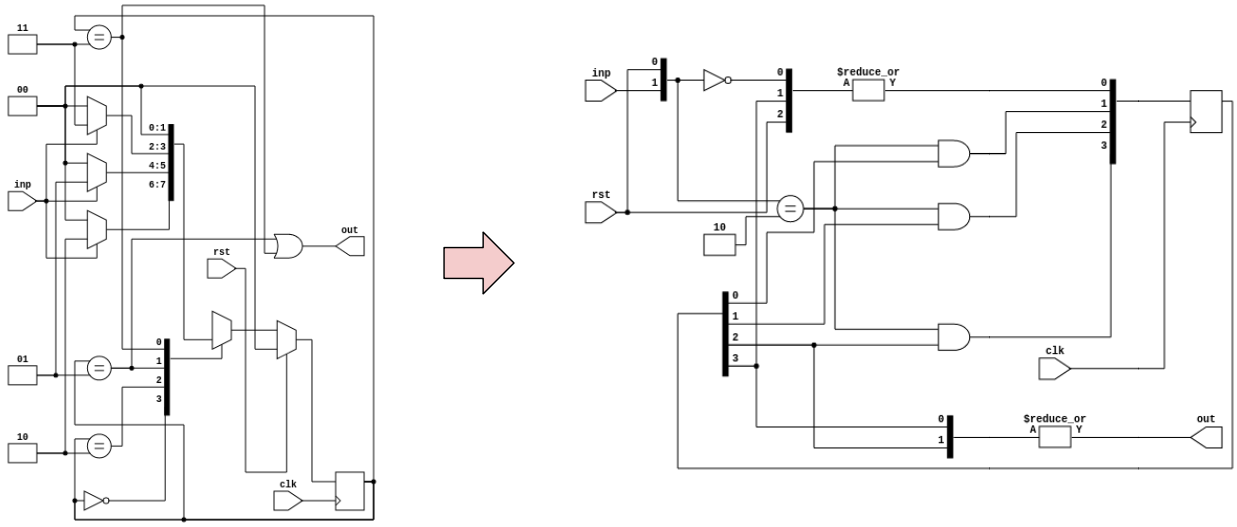
fsm_recode : Recode FSM states

This pass recodes the FSM states. It can perform one-hot encoding or binary encoding. By default, it converts everything to one-hot encoding. After fsm_recode, if we run fsm_info we see:

```
State encoding:
  0:     4'---1  <RESET STATE>
  1:     4'--1-
  2:     4'-1--
  3:     4'1---
```

fsm_map : Map FSM back into RTL

This maps the FSM block back into RTL blocks. It does the inverse process of fsm_extract. The diagram compares the circuit before and after one-hot recoding. We can see how much simpler the re-coded version is, demonstrating why one-hot state encoding is useful.



fsm : Optimize FSM

Runs all the previous FSM optimizations and transformations sequentially. It is equivalent to calling the following commands one by one.

```
# Identify and extract FSMs:
fsm_detect
fsm_extract

# Basic optimizations:
fsm_opt
opt_clean
fsm_opt

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

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

# Print information about FSMs:
fsm_info

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

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

Section VIII : Third-party visualization tools

In the process of writing this, I went searching for better visualization tools and methods and I came across netlistsvg for circuit diagrams. Further, I also came across a thread on stack overflow when visualizing FSM state diagrams.

Better circuit diagrams

We can use the "**netlistsvg**" tool that creates much better circuit diagrams than the show tool gives us. I use the show command for debugging and netlistsvg for clean diagrams for documentation. It is a standalone open-source project, and you can install it and learn more about it here.

The tool is not integrated into Yosys, so we need to export the netlist in json format using the "write_json" command in Yosys

```
write_json circuit.json
```
(Run in Yosys to Write circuit netlist in .json format)

Then, run netlistsvg from the command line to generate the diagram

```
netlistsvg circuit.json -o circuit.svg
```
(Run in CLI to generate diagram in .svg format)

The output is in .svg format. I used the svgexport tool to automate conversion to .png format, but there are plenty of other ways if you don't mind doing it manually.

```
svgexport circuit.svg circuit.png -x1
```
(Run in CLI to convert .svg diagram to .png format)

Please note that netlistsvg does generate diagrams with unnecessary blank spaces. I have edited this later to make it look neater, so don't be surprised if the diagrams look different!

FSM state transition diagrams

We have seen how the fsm_export can export the FSM into a KISS2 format file.

```
fsm_export -o fsm.kiss2
```

(Run in Yosys after extracting FSM to export FSM info)

First, this file has to be converted to a state transition graph. The ".dot" format used by Graphviz is one of the most common ways to store graphs. This code to covert KISS2 to dot format is taken from a stack overflow answer.

```
#!/usr/bin/env python3

import fileinput

print("digraph fsm {")

for line in fileinput.input():
    if not line.startswith("."):
        in_bits, from_state, to_state, out_bits = line.split()
        print(
            '%s -> %s [label="IN=%s,\\nOUT=%s"];'
            % (
                from_state,
                to_state,
                in_bits.replace("-", "?"),
                out_bits.replace("-", "?"),
            )
        )

print("}")
```

Feed the kiss2 file as text input to this Python file and save the text output from Python to a file using this command.

```
cat fsm.kiss2 | python3 kiss2dot.py > fsm.dot
```

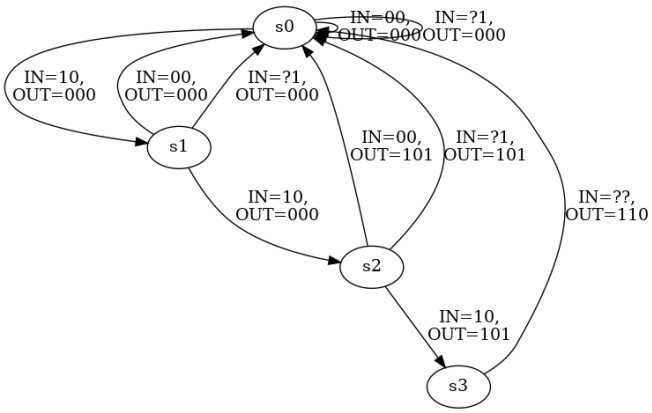
(Run in CLI to generate a dot file)

The .dot file is a format used by graphviz to represent graph structures. It can be converted to an image using the dot command which you get after installing graphviz.

```
dot -Tpng fsm.dot > fsm.png
```

(Run in CLI to convert to png)

The output looks as shown below. It's not the prettiest, but it is something!



References & Links

1. **Yosys Manual**, Clifford Wolf, YosysHQ
   https://yosyshq.readthedocs.io/projects/yosys/en/latest/CHAPTER_Basics.html
2. **ABC: A System for Sequential Synthesis and Verification**, Berkeley Verification and Synthesis Research Center
   https://people.eecs.berkeley.edu/~alanmi/abc/abc.htm
3. The **netlistsvg** tool, Neil Turley
   https://github.com/nturley/netlistsvg
4. **FSM export using Yosys** on stack overflow, Clifford Wolf
   https://stackoverflow.com/questions/32645965/fsm-export-using-yosys
5. **Graphviz**
   https://graphviz.org/doc/info/command.html

Appendix

The internal gate library

The cells available in the default library used for technology mapping is shown below.

Further, the library also has flip flops with set, reset, and enable signals. The library has multiple combinations of **clock sensitivity** (negedge/posedge), **reset level** (active high/low reset), **set level**, **enable level** (enabled on high/low), **reset value** (initial value after reset), and **reset type** (synchronous/asynchronous). Some combinations like synchronous set & reset & enable flips flops are not part of the library, but most combinations are present.

The same goes for latches with different cells at set levels, reset levels, and enable levels.

Name	Verilog	Description
$_BUF_	Y = A	Buffer
$_NOT_	Y = ~A	Inverter
$_AND_	Y = A & B	AND
$_NAND_	Y = ~(A & B)	NAND
$_ANDNOT_	Y = A & ~B	AND with one inv input
$_OR_	Y = A | B	OR
$_NOR_	Y = ~(A | B)	NOR
$_ORNOT_	Y = A | ~B	OR with one inv input
$_XOR_	Y = A ^ B	XOR
$_XNOR_	Y = ~(A ^ B)	XNOR
$_AOI3_	Y = ~((A & B) | C)	3-input And-Or-Invert
$_OAI3_	Y = ~((A | B) & C)	3-input Or-And-Invert
$_AOI4_	Y = ~((A & B) | (C & D))	4-input And-Or-Invert
$_OAI4_	Y = ~((A | B) & (C | D))	4-input Or-And-Invert
$_MUX_	Y = S ? B : A	2-input multiplexer
$_NMUX_	Y = ~(S ? B : A)	2-MUX with inv output
$_MUX4_		4-input multiplexer
$_MUX8_		8-input multiplexer
$_MUX16_		16-input multiplexer
$_TBUF_	Y = EN ? A : 1'bz	Tristate buffer
$_DFF_N_	always @(negedge C) Q <= D	Negegdge D flip flop
$_DFF_P_	always @(posedge C) Q <= D	Posedge D flip flop
$_DLATCH_N_	always @* if (!E) Q <= D	Negedge level latch
$_DLATCH_P_	always @* if (E) Q <= D	Positive level latch

Example liberty file

The following file from the Yosys examples is used as the library for technology mapping. It is a very simple library with a buffer, inverter, nand gate, nor gate, and 2 D flip flop, one with asynchronous set and reset.

```
// test comment
/* test comment */
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"; }
  }
  cell(DFFSR) {
    area: 18;
    ff("IQ", "IQN") { clocked_on: C;
                      next_state: D;
                          preset: S;
                           clear: R; }
    pin(C) { direction: input;
                clock: true; }
    pin(D) { direction: input; }
    pin(Q) { direction: output;
             function: "IQ"; }
    pin(S) { direction: input; }
    pin(R) { direction: input; }
    ; // empty statement
  }
}
```