**HW**

**Introduction:**

Digital circuits can be represented at different levels of abstraction. During the design process, a circuit is usually first specified using a higher-level abstraction. Implementation can then be understood as finding a functionally equivalent representation at a lower abstraction level. When this is done automatically using software, the term synthesis is used. So synthesis is the automatic conversion of a high-level representation of a circuit to a functionally equivalent low-level representation of a circuit.

Regardless of how a lower-level representation of a circuit is obtained (synthesis or manual design), the lower-level representation is usually verified. In recent years formal equivalence checking has become a vital verification technique for validating RTL and lower abstraction representations of the design.

Once you have verified the functional equivalence of the synthesized netlist, the timing requirements of the design have to be checked. For example, the adder may add, but does it add fast enough? At the behavioral level, clock cycle time is an abstract notion, but at the structural level, an actual cycle time has to be met by a particular set of gates. A timing analyzer is used to verify the timing.

Power analysis has materialized as a principal theme in today’s semiconductor industries. A power analyzer is a tool that determines how much power is utilized by a circuit.

When the target is a low-power application, the search for the optimal solution must include at each level of abstraction a design improvement loop In such a loop, a power analyzer ranks the various design synthesis and optimization options and thus helps in selecting the one that is potentially more effective from the power standpoint.

This tutorial describes the RTL to gate-level design flow using open-source tools. The intention behind this exercise is to give students a broad idea of the steps involved in these flows.

This tutorial will consist of three steps:

1. RTL Synthesis
2. Functional Equivalence checking
3. Timing and Power Analysis

We will use the following tools during this lab:

1. [Yosys](https://yosyshq.net/yosys/): Open synthesis suite
2. [OpenSta](https://github.com/The-OpenROAD-Project/OpenSTA): Gate-level static timing and power analyzer
3. [FreePDK-45nm](https://github.com/mflowgen/freepdk-45nm): Process design kit
4. Verilog design files for an open-source [64-bit RISC-V CPU](https://gitlab.com/shaktiproject/cores/c-class).

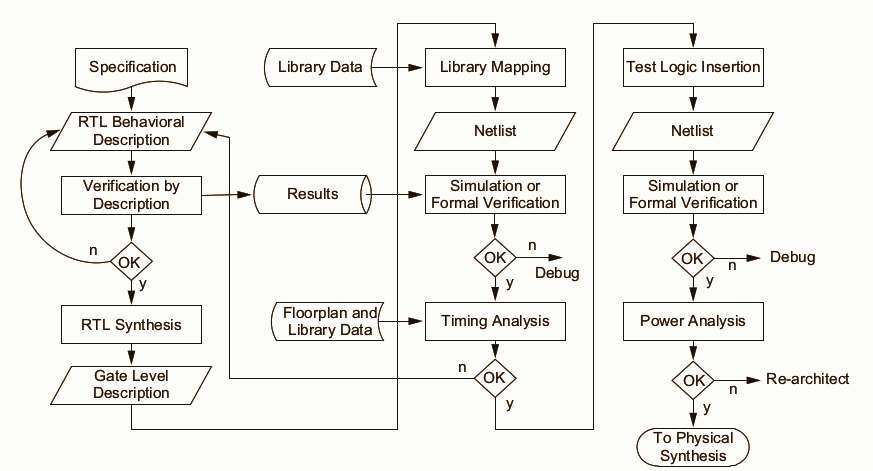


Fig: Synthesis Flow

**Follow the commands marked in the boxes.**

Run the following commands:

| **\*\*Connect to your** [**Thinlinc account**](https://engineering.purdue.edu/ECN/Support/KB/Docs/ECEThinlinc) **on desktop.eceprog.ecn.purdue.edu\*\***  git clone https://github.com/sujay-pandit/ece-51216-tutorial.git  cd ece-51216-tutorial; |
| --- |

**Part - I: Synthesis**

It is the process of transforming RTL to a Gate level netlist. Synthesis takes multiple steps:

* Converting RTL into simple logic gates.
* Mapping those gates to actual technology-dependent logic gates available in the technology libraries (standard cell libraries).

To achieve this, we will run Yosys with the following script:

Input: Design RTL files

Script:

// Read all Verilog design files

read\_verilog \*.v;

// Set the top module and check for any problems in the design

hierarchy -check -top mkccore\_axi4;

// Transform always\_blocks to netlists of RTL multiplexer and

// register cells

proc;

// Perform some logic optimizations and clean ups

opt;

// Perform FSM optimizations

fsm; opt;

// Optimize memory read/write cells

memory; opt;

// Map all RTL cells to a generic library of gates and

// registers

techmap; opt;

// Map internal register cell types to the register types

// described in the liberty file

dfflibmap -liberty $PDK\_NAME

// Optimize and Map logic cells to the specified standard cell

// library

abc -dff -liberty $PDK\_NAME

// Remove unused cells and wires

clean;

// Write synthesized netlist to a specified file

write\_verilog -noattr synth\_core.yv

Output: Synthesized Netlist (synth\_core.yv)

Run the following command: (This can take a few minutes)

| **./script\_synth.ys** |
| --- |

**Part - II: Logic Equivalence Checking**

LEC is done to ensure that the synthesis optimizations do not alter the designer's intent. Since it can take a long time to perform a LEC check on large designs, for our understanding, we will perform LEC check on a simple combinational circuit, the single-cycle integer ALU of the core.

We will use the following script along with Yosys to perform LEC:

Input: Module RTL (module\_fn\_alu.v) and Synthesized Netlist( synth\_alu.yv)

Script:

// Read golden RTL file

read\_verilog module\_fn\_alu.v

// Prepare for Verification flow

prep -flatten -top module\_fn\_alu

design -stash gold

// Read synthesized netlist

read\_verilog synth\_alu.yv

// Read the PDK used for synthesis

read\_liberty -ignore\_miss\_func $PDK\_NAME

// Prepare for Verification flow (Replace gates with their

// definition)

prep -flatten -top module\_fn\_alu

design -stash gate

design -copy-from gold -as gold module\_fn\_alu

design -copy-from gate -as gate module\_fn\_alu

// Make equivalence circuit

equiv\_make gold gate equiv

hierarchy -top equiv

// Show the equivalence circuit (Commented Out)

#show

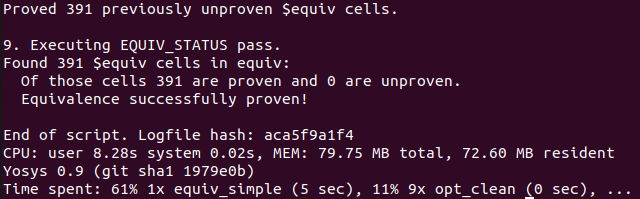
// Perform equivalence check

equiv\_simple -v

// Report if any errors are found

equiv\_status -assert

Output: Proven or Fails (Below is a sample output)



Run the following command:

| **./script\_equivalence.ys** |
| --- |

**Part-III: Timing and Power Analysis**

Once the synthesized logic is verified, we perform static timing and power analysis to check if the synthesized netlist can meet desired performance and power constraints. For this purpose, we will use OpenSta with the following script:

Input: Library file (PDK) and Synthesized Netlist (synth\_core.yv)

Script:

// Read PDK used for synthesis

read\_liberty $PDK\_NAME

// Read the synthesized netlist

read\_verilog synth\_core.yv

// Link top-level module

link\_design mkccore\_axi4

// Create clock and link it to the clock port in the design (

create\_clock -name clk -period 5000 {CLK}

// Generate timing report

report\_checks

// Generate power report

Report\_power

Output: Timing and Power report (timingpower.log)

Run the following command:

| **./sta timingpower.tcl** |
| --- |

**Questions:**

1. Even though the ALU is a small design, it is complex enough that it will be hard to make sense of the circuit generated for checking equivalence. For simplicity, we will do this exercise again for the multiplier.

Modify “script\_synth.ys” to perform synthesis on “signedmul.v” and then modify “script\_equivalence.ys” to perform equivalence on it. In the equivalence script, uncomment the command “show”. Now re-run the script with your changes and see the generated equivalence circuit (After running the equivalence script, Wait for a few seconds for the Dot Viewer to appear). Describe the idea behind it in your own words.

Note: “signedmul.v” is a parametrized module; for this exercise, modify the Verilog file and set the width parameter to 2 for both inputs.

1. You are given multiple skywater\_130nm “.lib” files in your directory. As you saw in your timing report, for the given clock, the generated netlist doesn’t meet timing (slack VIOLATED). Identify the “.lib” file for which the design meets timing and has the least power consumption.

Hint: Running scripts repeatedly with the changed library will take a long time. Modify the scripts to run on a subset of the design (a small sequential circuit) and see which library performs better. You can use mkregisterfile.v and RegFile.v as the only design file for your test design.

(Bonus: Prove equivalence for the register file)

**Recommended readings and References:**

* [Yosys manual](https://yosyshq.net/yosys/files/yosys_manual.pdf) (Chapter-2)
* CMOS VLSI Design - Neil H Weste and David Money Harris