ECE 51216, Spring 2023
Digital Systems Design Automation

Homework #5 – Running a complete RTL-to-logic design flow

Due Wednesday, April 12th , 2023 (11:59PM EST)

- It will help if you read through the entire document before you start working on this homework.
- It is OK for you to discuss these problems with other students in the class, but you should understand the tools and run the exercises step-by-step on your own to generate the solutions. Also, feel free to use the class mailing list for general discussions related to the topics covered in this homework, but please do not post solutions.

Introduction

In this homework, you will run a full RTL-to-logic design flow involving synthesis, timing and power analysis and verification. We will use open-source electronic design automation (EDA) tools and an open-source RISC-V microprocessor for this purpose.
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. In other words, synthesis is the automatic conversion of a high-level representation of a circuit to a functionally equivalent low-level representation of a circuit. In this course we are especially concerned about synthesis from the Register-transfer level (RTL) to a netlist at the logic level (Gates).
Regardless of how a lower-level representation of a circuit is obtained (synthesis or manual design), the lower-level representation is usually verified for functional correctness. While simulation is widely used for verification, in recent years, formal equivalence checking has become a vital verification technique.
Once the functional equivalence of the synthesized netlist is established, the timing requirements of the design must be checked. For example, an adder may add, but does it add fast enough? At the logic level the speed metric we are specifically concerned about is the clock period or cycle time which is inversely related to the clock frequency. A timing analyzer is used to verify that a given circuit meets a specific timing constraint, i.e., a target clock period.
Power consumption has become a very important metric in virtually all integrated circuits designed today. Power analysis or estimation is the process of determining how much power is consumed by a given circuit.
It is common to run synthesis, verification, timing, and power analysis in a loop to create an optimized implementation of the design that meets the desired constraints.

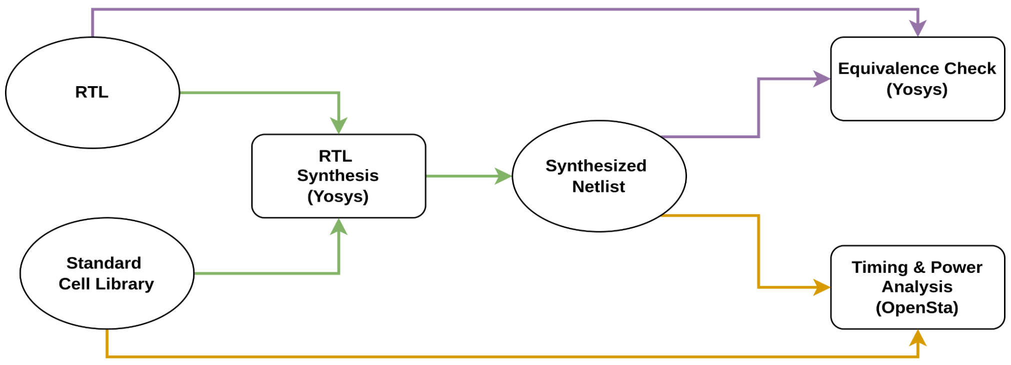
This homework will consist of three steps:

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

We will use the following open-source materials:

1. Yosys: Open synthesis suite
2. OpenSta: Gate-level static timing and power analyzer
3. Skywater 130nm: Standard cell library
4. Verilog design files for an open-source 64-bit RISC-V CPU.

The design flow you will step through is illustrated in the following figure.



The commands you need to run are outlined in boxes

****Connect to your Thinlinc account on desktop.eceprog.ecn.purdue.edu and then run the following commands****
Run the following command on your terminal**:**

```
git clone https://github.com/sujay-pandit/ece-51216-tutorial.git
cd ece-51216-tutorial;
```

Note: Before proceeding further with this tutorial, make sure your "gcc" is picked from "/bin/gcc". You can check this by executing "which gcc" on your terminal and modifying your PATH variable as necessary to make sure "/bin/gcc" appears first in your PATH.

Part - I: Logic Synthesis

Logic synthesis refers to the process of transforming RTL to a gate-level netlist. As discussed in class, it consists of two important steps:

- Converting RTL into an optimized technology-independent network of gates.
- Mapping those gates to actual technology-dependent logic gates contained in a technology library (standard cell library).

To achieve this, we will run Yosys with the following script: (The script has already been provided to you in the homework directory)

Input: Design RTL files
Synthesis 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 standard cell library
dfflibmap -liberty $LIB_NAME
// Optimize and Map logic cells to the specified standard cell
// library
abc -dff -liberty $LIB_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 on your terminal: (This may take a few minutes)

```
./script_synth.ys
```

Part - II: Logic Equivalence Checking (LEC)

LEC is done to ensure that the synthesis optimizations do not alter the functionality of the design. Since it can take a long time to perform a LEC check on large designs, for this homework, 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: 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 Standard cell library used for synthesis
read_liberty  -ignore_miss_func $LIB_NAME
// Prepare for Verification flow (Replace standard cells with their
// logic function)
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 for now)
#show -pause
// 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 on your terminal:

```
./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: Standard cell library file and Synthesized Netlist (synth_core.yv)
Script:

```
// Read Standard cell library used for synthesis
read_liberty $LIB_NAME
// Read the synthesized netlist
read_verilog synth_core.yv
// Link top-level module
link_design mkccore_axi4
// Create clock, link it to the clock port in the design and specify the
// clock period constraint (4000ps = 250MHz)
create_clock -name clk -period 4000 {CLK}
// Generate timing report
report_checks
// Generate power report
report_power
```

Output: Timing and Power report (timingpower.log)

Run the following command on your terminal:

```
./sta timingpower.tcl
```

Questions: (Your HW submission should just be a document that contains answers to these questions)

1. For each step of the tutorial, copy and paste the last ten lines of the output from the terminal console as text. [Do not paste screenshots for this question].
2. You are given multiple skywater_130nm ".lib" files in your directory. As you saw in your timing report generated from Part-III, for the given clock, the generated netlist for the core doesn't meet timing (slack VIOLATED).
   Identify the ".lib" file for which the design meets timing and has the least power consumption. Specify the critical path, slack, and the total power consumed for this new netlist.
3. 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 equivalence checking. 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. For this exercise, modify "signedmul.v" and set the width parameter to 2 for both inputs. Also, In the modified equivalence script, uncomment the command "show -pause". 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 circuit in Dot Viewer to load).

   Write a few sentences describing in your own words what the equivalence circuit represents and include the picture of the generated Dot viewer output.

4. **[BONUS]**
So far in the homework, you have proven equivalence for a combinational ALU circuit and a multiplier circuit. In this part, you are required to prove equivalence for a multi-cycle restoring divider.
Modify the synthesis script and equivalence checking script to make "mkrestoring_div.v" the top module.

Go through the Yosys manual and identify the commands you will need to prove equivalence.

If you are successful, you will see a similar terminal output showing the message as follows

```
153. Executing EQUIV_STATUS pass.
Found [redacted] $equiv cells in equiv:
  Of those cells [redacted] are proven and 0 are unproven.
  Equivalence successfully proven!

End of script. Logfile hash: 84be3a9e96
CPU: user 18.24s system 0.04s, MEM: 210.88 MB total, 201.65 MB resident
Yosys 0.9 (git sha1 1979e0b)
Time spent: 63% 1x [redacted] (11 sec), 11% 147x read_verilog (2 sec), ...
```

Paste your modified equivalence script and the number of $equiv cells.

NOTE: Given the large size of this design, generating Dot viewer output may take really long. It is recommended to run it with the "show -pause" command commented.

References

- Yosys manual (Chapter-2)
- OpenSta manual