EEE 498/591: Project Part 3

February 28, 2017

[40 Points] Due March 14th at midnight
Summary: This assignment continues from project part 1 in asking you to become more familiar with the the course tools and Verilog design.
Pair Programming: This assignment may be completed in pairs. Our expectation is that you will complete the assignment together: discussing concepts, debugging errors, and learning from each-other. You may submit your assignment individually, though you will be evaluated with the same criteria. Only one assignment needs to be submitted for each programming pair. You may select a new programming partner for each part of the project.
Study Group: If you haven't already it is a good idea to form a study group of 5-7 students. You will find your study group to be more effective if you meet at a consistent time to discuss the course lecture, reading, and assignmentes.

Design Process: If you have had some trouble quickly completing the assignments, I would strongly suggest utilizing the following design process.

1. **Practice:** Think about the underlying algorithm in the hardware. Practice executing it to see if you understand it.

2. **Pre-design:** You should spend some time thinking about your design. You should draw a block diagram of your design. A good block diagram will generally translate a single block to a single line of code, a module, or a mux. It is also a very good habit to label the wires in your block diagram with their signal names.

3. **Thought-experiment:** Once you have a block diagram you should think about how it might operate. You might write out a table of all of the wire values for a number of cycles to get a sense of what you think should happen. This is an opportunity to realize you have missed an important component or missed something critical about the functionality of your design.

4. **Write HDL:** If you are happy with your design you should write out the HDL. A good strategy is to focus on implementing the driver for each wire.

5. **Write a Testbench:** Once you are happy with the design you should write a test bench. A test bench should have three components: randomized stimulus to drive your block, a check that the output values are correct, and a check on some of the expected properties for the timing of your outputs.

6. **Debug:** Your design will likely be incorrect in the first pass. You should spend some time using a combination of print statements and waveform viewer to identify how the behavior of your hardware has diverged from its expected behavior. If you find a bug it is a good idea to write a simple test to quickly detect that specific error. A high-level debug flow would hypothesize the source of the bug, confirm/reject the hypothesis, and then correct the bug if the hypothesis is true.

7. **Synthesize:** Once your design is functional, you should run synthesis on it. You should read the logs to make sure that your design completed synthesis correctly. You will also run the gate level netlist to confirm that your design is synthesizing correctly.

1. [**0 Points**] **Prepare your submission:** You should create a directory that you will submit through email. We will refer to that directory as "$DIR". It is up to you where you create your directory, but a good place might be " /eee591/ProjPart3"[1] Your directory should contain a file to help us resolve your identity. Your file should be "$DIR/submit.txt" with a line indicating your name and posting id. It should contain your last name (as it appears in blackboard), followed by your first name, following by your posting id. There should be a single line for each of you. For example:

```
Hamm, John, 1234569
Douglas, Michael, 135711
```

2. [**0 Points**] **Project Directory** For most assignments in the class you will be provided a starter directory that includes a setup script for the environment, a Makefile for the flow, and starting code for the assignment. While the theme of the projects is generally an incrementally more complex design, we will often provide the solutions from the last assignment as the starting point for the current assignment. In this assignment no additional components will be provided over ProjPart2. You should start from there. Solutions will be posted at a future time.

3. [**10 Points**] **Reflection:** Reflect on your prior projects in 500-1000 words. Thinking about the design process, how did your experience reflect or diverge from what you expected? Did you find additional steps or re-ordering the steps helpful? Also, think about anytime that you were stuck. What was the sticking point? How did you come unstuck? How might you become unstuck more quickly in the future, or how might you avoid some stickiness? The context for this reflection might be this summary by Carol Dweck on her research on mindset [2]:

   "In a fixed mindset students believe their basic abilities, their intelligence, their talents, are just fixed traits. They have a certain amount and that's that, and then their goal becomes to look smart all the time and never look dumb. In a growth mindset students understand that their talents and abilities can be developed through effort, good teaching and persistence. They don't necessarily think everyone's the same or anyone can be Einstein, but they believe everyone can get smarter if they work at it."

   You should submit your typed reflection at the start of class after the assignment due date, individually. Points will be awarded based on clarity, style, topic, and percieved effort.

[1]You can use "mkdir -p /eee591/ProjPart3" to create all of the directories in one go
[2]here

4. **[10 Points] Register File** Make a synthesizable register file (RF). It should have two read and one write port. The read ports should be combinational logic, but the write of course must be synchronous. The register file should have 32 registers and the 0 address should always return 0. Make the RF parameterizable. We may change the width and number of stored locations in the autograder.

```
// RF.vp
//; my $bW = parameter( name=>"RFBitWidth", val=>16, doc=>"RF word width");
//; my $mS = parameter( name=>"RFSize", val=>16, doc=>"Number of RF entries");
//; my $aW = parameter( name=>"AddressWidth", val=>clog2($mS), doc=>"RF address wid

module ‘mname‘ (
          input  logic   [‘$bW-1‘:0]  wdat,
          input  logic   [‘$aW-1‘:0]  rpa_ad, rpb_ad, wp_ad,
          input  logic               wena,
          input logic                clk,

          output  logic  [‘$bW-1‘:0]  rpadat, rpbdat
);
// Empty module
endmodule: ‘mname‘
```

- **Submit:**
  - $DIR/sub1/rtl/RF.vp
  - $DIR/sub1/rtl/*.vp –
  - $DIR/sub1/char/char.csv
- **Rubric:** We will run your RF.vp and its children using our testbench, Makefile, and syn.tcl. We will procedurally parse the reports to determine the quality of your submission.

Submitted a tarball with all required files	0.6
Genesis2 runs to completion	0.6
VCS compilation runs to completion	0.6
VCS simulation runs to completion	0.6
Synthesis runs to completion	0.6
Design passes 90% verification tests	2.0
Design passes all verification tests	2.0
Gate Netlist passes all verification tests	3.0
Total	10.0

5. **[10 Points] Synchronous Read Memory** A standard SRAM memory reads and writes synchronously. The write looks like the RF. The read should appear on the output after the clock edge (this is the tCLK2Q for a sense amplifier and the latch after it). There is only one address, since the input data is used if it is a write operation and the output is used if it is a read. The read and write should thus be mutually exclusive (be careful to test for this). If neither or both are asserted, the memory should do nothing. Make the memory 256 words with 32 bits. Make it parameterized. We may test with other values.

```
// memsram.vp
//; my $bW = parameter( name=>"BitWidth", val=>16, doc=>"memory word width in bits"
//; my $mS = parameter( name=>"Size", val=>16, doc=>"Number of memory entries");
//; my $aW = parameter( name=>"AddressWidth", val=>clog2($mS), doc=>"Memory address

module ‘mname‘ (
          input  logic   [‘$bW-1‘:0]  wdat,
          input  logic   [‘$aW-1‘:0]  add,
          input  logic             wena, rdena,
          input logic              clk,

          output  logic  [‘$bW-1‘:0]  rddat
);
// Empty module
endmodule: ‘mname‘
```

6. **Submit:**
   - $DIR/sub2/rtl/memsram.vp
   - $DIR/sub2/rtl/*.vp – if you created additional verilog files for your hardware design you should include those here so that we can evaluate them as well. Note that you should not include your verification code, we will use our own testbench.
   - $DIR/sub2/char/char.csv

7. **Rubric:** We will run your memsram.vp and its children using our testbench, Makefile, and syn.tcl. If you found that you needed to make changes to the Makefile or synthesis script, these will not be propagated over. We will procedurally parse the reports to determine the quality of your submission.

Submitted a tarball with all required files	0.6
Genesis2 runs to completion	0.6
VCS compilation runs to completion	0.6
VCS simulation runs to completion	0.6
Synthesis runs to completion	0.6
Design passes all verification tests	2.0
Gate Netlist passes all verification tests	2.0
Design has 2ns clock period	3.0
Total	10.0

8. [**10 Points**] **Sequential Divider** For the second part of the course project we will implement a sequential divider. A sequential divider uses a single adder module to iteratively subtract the denominator from the numerator to calculate the quotient and remainder. A simple approach would right shift the denominator, while using the sign of the resulting subtraction to determine the quotient. You should implement your design in "seqDiv.vp" with the following interface.

```
// seqDiv.vp
//; my $bW = parameter( name=>"bitWidth", val=>16, doc=>"Width of input");
module `mname` (
          input  logic [`$bW-1`:0] num,
          input  logic [`$bW-1`:0] den,
          input  logic             nd_valid,
          output logic             nd_ready,

          input logic              clk,
          input logic              rst,

          output logic             qr_valid,
          output logic [`$bW-1`:0] quo,
          output logic [`$bW-1`:0] rem
                );
// Empty module
endmodule: `mname`
```

As the seqDiv is busy, some control flow is need to indicate when a result is available (qr_valid) and when new inputs can be provided (qr_ready). It is possible that there are no new inputs for the module, in which case the inputs will not be valid (nd_valid). rst is active low, clk is clock, num is numerator, den is denominator, quo is quotient, and rem is remainder.

- **Practice:** Think about long division, how would the binary version differ from the decimal version.
- **Pre-design:** You should spend some time thinking about your design. You should draw a block diagram of your design including flip flops, shifters, adders, and state machine module. It is also a very good habit to label the wires in your block diagram with their signal names.
- **Though-experiment:** Once you have a block diagram you should think about how it might operate. You might write out a table of all of the wire values for a number of cycles to get a sense of what you think should happen. This is an opportunity to realize you have missed an important component or missed something critical about the functionality of your design.
- **Write HDL:** If you are happy with your design you should write out the HDL. A good strategy is to focus on implementing the driver for each wire.

- **Write a Testbench:** Once you are happy with the design you should write a test bench. In this case you will need to make sure that you are dealing with control flow correctly and comparing the correct output. Your test bench should measure the average number of cycles that it takes to complete a division. A simple method would be to divide the number of cycles by the count of cycles that the output is valid. Your test bench should be "top_seqDiv.vp" and the instance of seqMult should be named "my_seqDiv"
- **Debug:** Your design will likely be incorrect in the first pass. You should spend some time using a combination of print statements and waveform viewer to identify how the behavior of your hardware has diverged from its expected behavior.
- **Synthesize:** Once your design is functional, you should run synthesis on it. You should read the logs to make sure that your design completed synthesis correctly. For example, it would be a good idea to discuss the various warnings and errors you see on Piazza to determine which are benign and which are not.

```
make gate_sim MODULE_NAME=seqDiv CLK_PERIOD=2.0
```

9. **Submit:**
   - $DIR/sub3/rtl/seqDiv.vp
   - $DIR/sub3/rtl/*.vp – if you created additional verilog files for your hardware design you should include those here so that we can evaluate them as well. Note that you should not include your verification code, we will use our own testbench.
   - $DIR/sub3/char/char.csv which should contain the characterization of your design as a comma delimited file, for example:

```
divRate_cyc, 16.0, # Average Cycles per Division
area_umsq, 10000.0, # Cell Area of your design
power_mw, .001, # Power of the design
delay_ns, .5, # The minimum achievable timing in the design
target_ns, .4, # The desired clock frequency
```

10. Rubric: We will run your seqDiv.vp and its children using our testbench, Makefile, and syn.tcl. If you found that you needed to make changes to the Makefile or synthesis script, these will not be propagated over. We will procedurally parse the reports to determine the quality of your submission.

Submitted a tarball with all required files	0.6
Genesis2 runs to completion	0.6
VCS compilation runs to completion	0.6
VCS simulation runs to completion	0.6
Synthesis runs to completion	0.6
Design passes 90% verification tests	2.0
Design passes all verification tests	2.0
Gate Netlist passes all verification tests	3.0
Total	10.0

11. **Submission** Once you have completed your submission directory, you should create a "tarball" and email it to the submission email. You should create your tarball with the following command:

    ```
    tar -czvf ProjPart3.tar.gz ProjPart3
    ```

    You should email your tarball to:

    ```
    asu.eee.591ca.spring17@gmail.com
    ```

    You should use the subject line that helps you distinguish your submissions.

    If everything is working well you should receive a response within a few hours regarding your score. Feel free to submit as many times as you would like. You should assume that the grading script is down until announced otherwise. As always, we appreciate your help debugging issues as they come up. We will be much crueler if you hang the autograder this time–carefully check your design before submitting.