“More on Verilog: State machines using square integer root.”

Due: Sep. 24 at 6pm. You must submit your designs electronically and hand in your printed report. You may hand in reports in lecture, or at my office by 6pm. No late submissions.

Points: 2% of class grade

Goals: State machine design, synthesis experience, design trade-offs, familiarity with functional unit behavior.

In this project you will use a multiplier (supplied by us) to perform the following tasks:

1. Synthesize multiplier using a number of different configurations. (~20% of score)
2. Using the multiplier design an “integer square root” (ISR) functional unit. When you turn it in, the filename should be `my_isr.v`, and the testbench should be named `isr_test.v` (~60% of score)
3. Synthesize the ISR. (~20% of score)

Each of the three parts is described below. At the end of this assignment is a list of questions that you are to answer as well as directions for what to turn in. We strongly recommend you read the whole assignment before starting. You will be graded for style (following the rules we’ve given you) in addition to correctness.

Part I

On the course website we have provided a 64-bit pipelined multiplier in a zip file. Examine the various files, including the Makefile and the testbench. Synthesize the multiplier (this may take 10-15 minutes). Using the supplied multiplier as a template, make two otherwise identical multipliers that have 2 and 4 pipeline stages each (i.e., increase the number of bits multiplied in each stage while reducing the number of stages). Synthesize each of these, keeping around the files needed to answer the questions found at the end of this assignment.

Part II

You are now to create a module that uses the multiplier we supplied (above) to compute the square root of an integer. It will take a 64-bit number as an input and generate a 32-bit number that is the largest integer that is not larger than the square root of the input. It is to function as follows:

- If reset is asserted during a rising clock-edge, the input value is to be stored. The module will then find the integer square root of that value.
- When the module has computed the answer, the output is placed into “result” and “done” is driven high.
- It must never take more than 600 clocks from the last clock reset is asserted to the rising edge of done.
- The module declaration must be:

```plaintext
module ISR(reset, value, clock, result, done);
input reset;
input [63:0] value;
input clock;
output [31:0] result;
output done;
```

Note that there is no requirement (or even suggestion) that this module be pipelined.
You will almost certainly need to perform something of a binary search in order to accomplish this task. A simple algorithm is:

```verbatim
for(i=31 to 0){
    set bit i of proposed_solution
    if(proposed_solution squared is greater than value)
        clear bit i of proposed solution
}
```

Note that you will not be using a loop in your Verilog code. This is just the algorithm.

**Part III**

You are to synthesize the module you created in Part II. This may take a fair amount of CPU time…

**Questions**

Answer the following questions:

1. What is the cycle time achieved when you synthesized our multiplier? What does this mean the total latency is for a multiplication?
2. Answer question 1 for the two multipliers you created.
3. Consider the relative values of the answers you found to questions 1 and 2. Do these seem reasonable? Why or why not?
4. What was the cycle time found in Part III?
5. How long would it take for your module to compute the square root of 1001 given the cycle time of question 4? Would you expect a performance gain or penalty if you used your 2-stage multiplier?

In addition to providing your answers to the above questions, you are to attach printouts of the relevant pages of the reports generated by Design Compiler that provide the cycle-time information. This should be no more than one page per question and the relevant result should be clearly highlighted. Those printouts should be stapled to the back of your answers to these questions.

**Turn-in**

You are to turn in the answers to the Questions section and the design compiler printouts in class (or at my office by 6pm). You are to electronically turn in your ISR module and supporting code (you do not need to submit your 2-stage or 4-stage multipliers electronically). You should use the same submission script as you used for assignment 1. Use the multiplier we supplied as a basis for your ISR. Your Makefile should behave identically to the multiplier when “make”, “make syn” and “make clean” are used. You are to supply a testbench that does a reasonable job of testing your ISR module. The only syntax we care about is that you print either “@@@Passed” or “@@@Failed” in your output, all other output doesn’t matter as long as you keep your total output to less than 10,000 lines. We will run your testbench on several working/non-working ISR’s and find out how well you detect errors and that will be factored into your grade.

Be sure you used the file names we asked you to use (see the first section of this assignment).
Helpful Hints

1. If you are having trouble with synthesis try adding the following before your always @ posedge clock blocks:
   
   ```
   //synopsys sync_set_reset "reset"
   ```
   And the following to your tcl file:
   
   ```
   set compile_segmap_synchronous_extraction "true"
   ```
2. You may need to modify your mult_test.v file to test the pipelined versions of the multiplier (look at ~line 48).
3. It is not necessary to use the pipelined feature of the multiplier to write your ISR, you should still be able to meet the 600 cycle limit.
4. You should use only 2 always blocks in each module, one for your registers (@posedge clock) and one for the calculation of the next state (@*) see the example template below:

```vhd
module ISR(reset, value, clock, result, done);
input reset;
input [31:0] value;
input clock;
output [15:0] result;
output done;
reg [3:0] counter;
reg [3:0] next_counter;
....
always @*
begin
  next_counter = counter;
  ....
  if (some event happens)
  begin
    next_counter = counter + 1;
    ....
  end
end
always @(posedge clock)
begin
  if (reset)
  begin
    counter <= #1 0;
    ....
  end
  else
  begin
    counter <= #1 next_counter;
    ....
  end
end
endmodule
```