"More on Verilog: State machines using integer square-root."

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. (~65% of score)
3. Synthesize the ISR. (~15% of score)

Each of the three parts are 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.

Part I

On the course website we have provided a 32-bit pipelined multiplier in a zip file. Examine the various files, including the Makefile and the two testbenches. Synthesize the multiplier (this may take 10-15 minutes). Using the supplied multiplier as a template make three otherwise identical multipliers that have 1, 2, and 4 pipeline 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 32-bit number as an input and generate a 16-bit number that is the largest integer that is not larger than the square root of the input. It is to function as follows:

- The value of "`input`" must be held constant for at least one full clock period before reset goes low and may not change while reset is low. 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. These values are held until reset is again asserted.
- It must never take more than 400 clocks from the falling edge of `reset` to the rising edge of `done`.
- The module declaration must be:

```
module ISR(reset, input, clock, result, done);
input reset;
input [31:0] input;
input clock;
output [15:0] 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:

```
for(i=15 to 0){
       set bit i of proposed_solution
       if(proposed_solution squared is greater than input)
              clear bit i of proposed solution
```

```
}
```

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 three 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 turn in printouts of the portions of the reports generated by vcs which provide the cycle-time information. This should be no more than one page per question and the relevant portions 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 (and printouts) to the Questions section into the 470 box. You are to electronically turn in your ISR module and supporting code. We strongly suggest you use the multiplier as a basis for this. Your Makefile should behave identically to the multiplier when “make”, “make syn”, “make testsyn” and “make clean” are used. You are to supply a testbench that does a reasonable job of testing your ISR module.