CprE 381 – Computer Organization and Assembly Level Programming, Fall 2018

Lab #6

In this lab you will continue to learn VHDL and Quartus skills and get practice in combining the two. Specifically, you will learn how to use synchronous VHDL processes to make memory units in the form of clocked registers. A register can be thought of as vector version of a 1-bit D flip-flop. This week we will be implementing a design for a sequential multiplier. The design can be found in your textbook in section 3.3 which discusses how multiplication works in hardware.

The book Free Range VHDL is provided on the class website and is a good tool to learn VHDL. Chapter 5 explains how to use process statements which is one focus of this lab.

0) Before working on the lab:

(a) Create a new folder for lab 6 <user home folder>/cpre381/lab6. Use this directory to save VHDL and Quartus files.

(b) Unzip the provided lab6.zip in the lab6 folder. The zip contains five example VHDL files:

a. 1-bit DFF which you can modify to make an N-bit register.

b. An example use of generics, for-generate and testbench design in VHDL in the form of a one's complementer

Of course, more examples can be found in the Free Range VHDL book.

(c) Read the first few pages of Section 3.3 in the class textbook, "Computer Organization and Design 5th Edition". This will remind you how multiplication works in binary.

1) A **Sequential Multiplier** mimics the simple multiplication algorithm we learned in elementary school. Start with the least significant place value (1-bit) of the *Multiplier* and multiply that single place value by the entire *Multiplicand*. The result is the first row of the product. In binary, multiplication of vector by a single bit is as simple as ANDing the single bit with each place value in the vector. So the multiplication will be either all zeros (when the single bit = '0') or the original vector (when the single bit = '1'). Continuing, add a new row to the result and append a zero before taking the next place value of the *Multiplier* and multiplying it by the *Multiplicand*. After you've gone through all place values of the *Multiplier,* sum the resulting rows to achieve the final product. A visual of the result of this process is given on the first page of section 3.3 in the textbook.

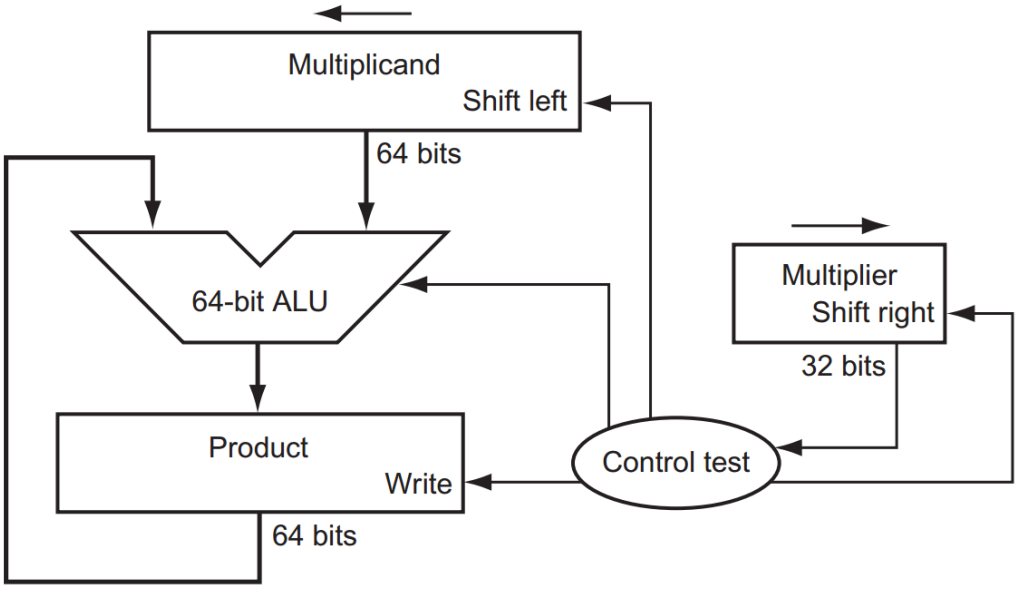

FIGURE 3.3 First version of the multiplication hardware. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (Appendix B describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

We will be implementing the above circuit using a combination of VHDL and the schematic tool in Quartus.

2) The ***Multiplicand*** block is a 64-bit left-shift register. The register should be controlled by load signal (not pictured) which allows a 32-bit value to be loaded into the bottom half of the 64-bit register with the upper half set to all 0. Using VHDL create the *Multiplicand* 64-bit left-shift register w/load and reset signals. You can refer to the provided dff.vhd file for how a 1-bit register is implemented in VHDL using an input clock signal. You can modify this file to build the *Multiplicand* register, if you'd like.

(a) In your submission provide your code for this component. Name the VHDL file *multiplicand.vhd*.

(b) In your report provide simulation waveform screenshots of the component working as described here. Remember, VHDL can be simulated using the ModelSim program. Provide test cases for loading, reset, and left-shifting. Explain how you implemented the left-shift operation in your report.

3) The ***Multiplier*** block is a 32-bit right-shift register. The register should be controlled by load signal which allows a new 32-bit value to be loaded into the register. Using VHDL create the *Multiplier* 32-bit right-shift register w/load and reset signals. You can refer to the provided dff.vhd file for how a 1-bit register is implemented in VHDL. You can modify this file to build the *Multiplier* register, if you'd like.

(a) In your submission provide your code for this component. Name the VHDL file *multiplier.vhd*.

(b) In your report provide simulation waveform screenshots of the component working as described here. Provide test cases for loading, reset, and right-shifting. Explain how you implemented the right-shift operation in your report.

4) The **64-bit ALU** component adds two 64-bit values together. The difference between this Arithmetic Logic Unit (ALU) and a 64-bit adder is the enable signal provided by the Control Test component. For this lab we will simply use a 64-bit full-adder in place of the 64-bit ALU with enable control. We will also replace the Control Test unit with a simple multiplexor. Pick one way to build a 64-bit adder and implement it:

(a) Use eight of the 8-bit full-adder you created in lab5 to create a 64-bit adder. This process will be the same as using 1-bit full-adders to create an 8-bit full-adder as you did in lab5.

(b) Using VHDL create a 64-bit adder using the dataflow architecture style. You can ask your TA if you are having trouble with this.

(c) Alternatively, create an N-bit adder using VHDL generics and dataflow architecture style. This would be fairly similar to choosing **(b)** except you will now have an N-bit adder for any other designs you may need it for.

In your report describe which method you chose to implement a 64-bit adder. Also give screenshots of simulation results of multiple test cases. Submit your code/files for the 64-bit adder.

5) We will replace the Control Test unit shown in the sequential multiplier diagram with a simple 2:1 multiplexor (w/64-bit input and output vectors). The output of the multiplexor should be connected to the 64-bit adder input where the M*ultiplicand* register was originally connected. The '0' input of the 2:1 mux should be connected to a 64-bit vector constant of "00…000". The '1' input of the 2:1 mux should be connected to the 64-bit output of the *Multiplicand* register. The select signal for this 2:1 mux is the least significant bit of the *Multiplier* register.

(a) Create this 2:1 mux using VHDL. It should have two 64-bit inputs and one 64-bit output. The select signal should be 1-bit.

(b) Create a 64-bit register using VHDL. This register will serve as the product register in the circuit diagram above and represents the output of the sequential multiplier. The *Product* register is very similar to the *Muliplicand* register except that it does not shift. It simply stores the input into the register on the rising edge of a clock cycle.

(c) Wire all the components together. If you want, you can use Quartus to match the provided design for the sequential multiplier with the Control Test unit replaced by a 2:1 mux. Refer to lab 5 if you've forgotten how to create block files from VHDL files. Attach a screenshot of your finished Quartus schematic. If you choose not to use Quartus for this, sketch the overall diagram, and attach it on the lab report (individual component can be drawn as a black box, but make sure to mark all inputs & outputs and how they are wired).

(d) Simulate the sequential multiplier using multiple test cases. Provide screenshots of your simulation in your report. Include all your VHDL code (and Quartus files if any) in your submission.

SUBMISSION:

- Create a zip file *Lab-6-submit.zip*, including the completed code and screenshots from the lab.
- The lab report which answers all questions from this document. You can include your screenshots in the report if you'd like.
- The file names in your submission should be self-explanatory.
- Submit the report as a pdf and the zip on Canvas, before your lab time next week.