Lab 2: Multipliers

Introduction

In this lab you will build multiplier circuits and test them on the Altera DE1-SoC board. You will use Verilog to describe the circuits, and then simulate them using *ModelSim* software with your own testbench. Please read the accompanying documents *Testbench Tutorial* and `Creating Generic Hardware`. The latter is optional, but may save you effort in writing your code.

Part I: Signed Array Multiplier

The Multiplier Module

Create an **8x8 signed array multiplier that uses Booth encoding**. Each row of the multiplier should consist of a number of Multiplier Cells, as shown on the left hand side of Figure 1. Furthermore, each row should use one instance of the Booth Encoder Cell (shown on the right hand side of the figure). A simple example of an array-based multiplier can be seen in Chapter 10 of the course textbook (*Digital System Design*). Note that while this multiplier has similar structure as the one you are supposed to design, it does not use Booth encoding, and therefore supports only unsigned values.

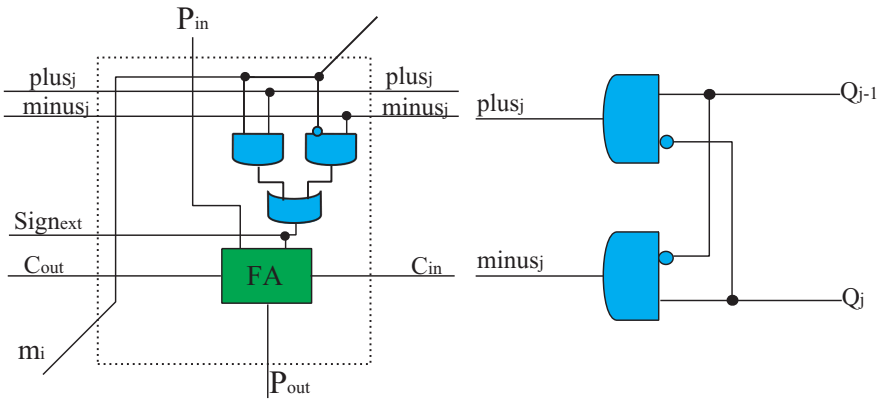


Figure 1: An array multiplier cell (left) and a Booth Encoder cell (right)

Your circuit should be implemented in its own separate Verilog or SystemVerilog module, and not directly as the top-level module of a DE1-SoC Quartus project. It should accept two 8-bit inputs, representing the two numbers to be multiplied in signed twos-complement representation. It produces a single 16-bit output. **You must not use the * and + Verilog operators in the design of your circuit.** Structural coding, by instantiating rows of Multiplier and Booth cells, is encouraged. You may find the `generate` statement helpful for instantiating many copies of your sub-modules. See the accompanying document *Creating Generic Hardware* for a tutorial.

The Testbench

After you've created your 8x8 signed array multiplier module, the next thing you should do is to simulate it to make sure it is correct. For this, you will write a testbench to test every possible combination of inputs (all 65536 of them), and compare the result of your multiplier to the value returned by the Verilog multiplication operator *. Please see the accompanying document *Testbench Tutorial*, which describes a very similar testbench to the one required for this lab, except for testing an adder rather than a multiplier.

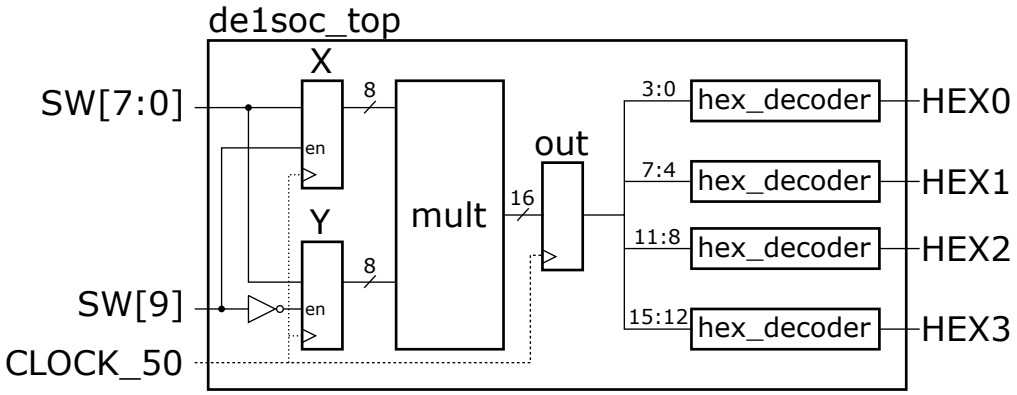
If a mismatch occurs between the expected product and your module's output, you should print a message to the ModelSim console using the `$display` command indicating which test case failed. This will help you locate the bug. If all test cases pass, print a message indicating this as well. Note: you can represent signed values in Verilog like so: `-8'd128`. This corresponds to the 8-bit value `8'b10000000`, as expected by twos complement representation of -128 in 8 bits.

Run your testbench in ModelSim to verify that your multiplier is correct. If the testbench finds errors in some test cases, debug and fix your multiplier until all results pass. Don't forget that you can set the radix of the displayed signals to *Decimal* in the wave window.

The Quartus Project

Finally, once your testbench reports that your multiplier works correctly in all cases, it's time to implement it as real hardware on the DE1-SoC board. To do this, create a top-level module that connects to the board's pins. An empty one is provided for you in the starter kit, along with the project files and pin assignments.

You need 16 inputs to drive your multiplier module, but there are not that many switches on the board. You will need to build a small circuit surrounding your multiplier module that will let you input 8 bits at a time using only 8 switches, using registers to remember the last-entered value. The 16-bit output will be displayed on four of the HEX displays, using 4 instances of the hex decoder module provided in the starter kit. Here is a full block diagram of the system:



The `mult` module is your multiplier circuit. Both inputs and the output are registered, with the enables of the X and Y registers controlled by `SW[9]` which selects which register is being loaded. The out register is always enabled and feeds the hex decoders. All registers are clocked by the board's `CLOCK_50` input. The presence of the registers will allow Quartus to properly measure the clock frequency of your multiplier.

Part II: Carry Save Multiplier

It is possible to design more advanced multipliers that can compute a product of two numbers more quickly. For Part II you will design an 8x8 *Carry Save Multiplier* (CSM). You may make it either unsigned, or signed using Booth encoding, if you find it easier to re-use part of your design from Part I. However, unsigned is simpler to create, and is all that is required for Part II. You will need to perform the same three steps as for Part I:

1. Design the 8x8 CSM module itself.
2. Create and run a testbench to verify that the CSM works correctly for all possible cases.
3. Create a Quartus project, with the same block diagram as Part I, to implement the multiplier on the DE1-SoC board.

Steps 2 and 3 should be straightforward and require little to no modification from Part I. Be mindful of your testbench code when switching from signed to unsigned inputs.

Part III (BONUS): Wallace Tree Multiplier

There exists an even faster multiplier design than the CSM. Should you choose to accept this challenge, do your own research on the Wallace Tree Multiplier and perform the same 3 steps as for Parts I and II.

Instructions

Complete these preparation steps for Parts I, II, and optionally, III, and have them ready to be examined by your TA at the start of the lab:

- Draw a diagram of the 8x8 multiplier circuit.
- Write the (System)Verilog code for the multiplier circuit.
- Write the (System)Verilog testbench and use it to debug your multiplier.
- Create the Quartus project and design the top-level module.
- Record the FPGA area utilization and clock frequency of the system. These can be found after compiling in Quartus in the compilation report (press *Ctrl-R*). *Fitter→Summary* shows the number of used ALMs and Registers. *TimeQuest Timing Analyzer→Slow 1100mV 85C Model→Fmax Summary* shows the achieved clock frequency in MHz.[1]

During the lab, demonstrate the following steps for Parts I, II, and optionally, III to your TA:

- Run your testbench in ModelSim to show that all tests pass.
- Program the FPGA and demonstrate the working multiplier on the DE1-SoC board.

Marking Scheme

- Part I preparation (2 marks)
- Part II preparation (2 marks)
- Part III preparation (2 marks) - bonus
- Part I in-lab (2 marks)
- Part II in-lab (2 marks)
- Part III in-lab (2 marks) - bonus

For the Part I and II items, you will receive the full 2 preparation marks if it is complete, and the full 2 in-lab marks if it's fully working. Partially-working or partially-complete items receive 1 mark, and incomplete or not-at-all working items receive a 0. Half marks are assigned at the TA's discretion. For Part III, it's all-or-nothing – you must present a fully working Wallace Tree Multiplier with a complete preparation. If you are unable to answer questions about your work, you will not receive full marks.

[1] The CLOCK_50 signal in the starter kit has been over-specified as 1 GHz to make Quartus try its hardest.