University of California, Davis
Department of Electrical and Computer Engineering

EEC180 DIGITAL SYSTEMS II Winter Quarter 2021

LAB 4: Memory System Design / Booth Multiplier Design

Objective: In the first part of the lab, you will use Verilog to model RAM modules. In the second part, you will design a multiplier for signed binary numbers using Booth's algorithm and verify using the testbench given to you.

Prelab

Read the lab carefully and show the steps associated with the multiplication of -10 and 9 using a Booth multiplier as shown in II.

I. RAM

There are several ways to implement a memory component in an Intel FPGA. One method is to instantiate a RAM module from the Quartus Prime Parameterized Modules. The MegaWizard Plug-in Manager enables you to configure the RAM module to fit your desired specifications. Another way to implement a memory component is to model it behaviorally in Verilog. In this lab, you will implement memory components using behavioral modeling.
Memory can be specified in Verilog as a two-dimensional array. For example, a memory with 32 words and 4-bits per word can be declared with the statement:

```
reg [3:0] memory [31:0];
```

In the MAX 10 FPGA, memory can be implemented either using flip-flops or by using dedicated memory resources within the FPGA known as M9K blocks. Each M9K block contains 8192 memory bits that can be configured to implement various memory modules. There are 182 such blocks available on MAX10 chip. Depending on how you write your Verilog code, the Quartus Prime compiler will either infer flip-flops or M9K memory blocks to implement your memory device.

In this part, you will design **two** 16x4 RAM modules and implement them in the DE10 board. The block diagram for a 16x4 RAM is shown in Figure 1.

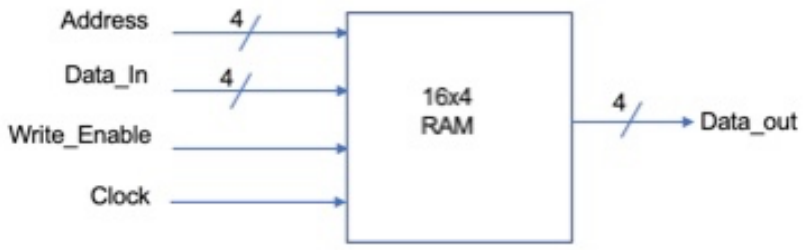

Figure 1. A 16x4 RAM module

You will test your memory modules using the switches, LEDs and 7-segment displays on the DE10 board. The I/O device assignments are as follows:

I/O signal	DE10 device
Write Enable (for both RAM modules)	SW[9]
Select RAM module for writing	SW[8]
Clock	KEY[0]
Address (for both RAM modules)	SW[7:4]
Data_In (for both RAM modules)	SW[3:0]
Address Display	HEX3
Data_In Display	HEX2
RAM1 Data_Out Display	HEX1
RAM0 Data_Out Display	HEX0

Your memory modules must operate as follows:

- Only one RAM module (RAM1 or RAM0) can be written into at a time.
- The Select switch (SW[8]) determines which memory module will be written.
- A write operation will occur to the selected RAM module on a positive Clock transition when the Write Enable signal is active (high).
- Both RAM modules can be read to their respective 7-segment displays simultaneously.
- The RAM output data can be either clocked (synchronous) or unclocked (asynchronous).

Perform the following steps:

1. Write the Verilog code to implement the two 16x4 RAM modules on the Altera DE10 board in M9K blocks. Use *synthesis ramstyle pragma* as shown in the link, (https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/hdl/vlog/vlog_file_dir_ram.htm).
   Compile your program. Verify that your code infers M9K memory blocks by examining the Compilation Report. Ideally, you should also determine how to infer flip-flops for your memory devices instead of M9K blocks. You can also go to Tools -> Chip Planner to see mapping of the resources on actual chip and in this case, one of the M9K blocks (Yellow color) must be enabled as shown below.

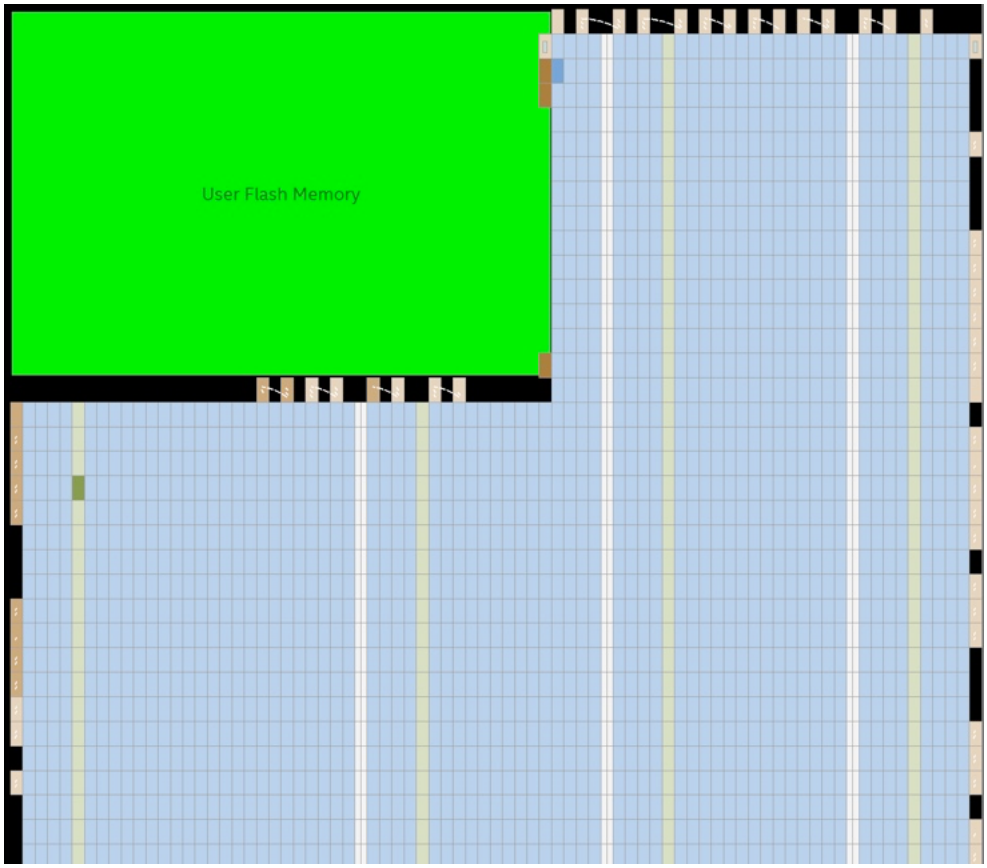

Figure 2. Instantiation of M9K as shown on Chip Planner

2. Modify your Verilog design to specify the initial contents of your RAM modules. The easiest way to do this is by using an **initial** construct. For example, you could use a for-loop within an **initial** block to initialize the RAM contents. Another option is to use a MIF (Memory Initialization File) to assign initial values. (See the Quartus Prime documentation, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/archives/qts-qpp-5v1-17-0.pdf for information on both methods of memory initialization.)

   In addition, the Intel® Quartus® Prime software offers several IP cores to implement memory modes. See the Quartus Prime documentation, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_ram_rom.pdf for more information.

   The **initial** construct is probably more straightforward for our application.

3. Download and test your design. Demonstrate to your TA that you can read out the initial contents of your RAM modules and that you can write new values to the RAMs.

II. Multiplication of Signed Numbers

In this part, you will use Verilog to design and simulate a multiplier for twos complement, signed binary numbers using Booth's algorithm.

Booth's algorithm works as follows, assuming each number is n bits including sign: Use a (n+1)-bit register for the accumulator (A) so the sign bit will not be lost if an overflow occurs. Also, use an (n+1)-bit register (B) to hold the multiplier and an n-bit register (C) to hold the multiplicand.

1. Clear A (the accumulator), load the multiplier into the upper n bits of B, clear B_0, and load the multiplicand into C.
2. Test the lower two bits of B (B_1B_0).
   If B_1B_0 = 01, then add C to A (C should be sign-extended to n+1 bits and added to A using an (n+1)-bit adder).
   If B_1B_0 = 10, then add the 2's complement of C to A.
   If B_1B_0 = 00 or 11, skip this step.
3. Shift A and B together right one place with sign extended.
4. Repeat steps 2 and 3, n-1 more times.
5. The product will be in A and B, except ignore B_0.

Example for n=5: Multiply -9 by -13.

#	**Action**	**A**	**B**	**B_1B_0**	
1.	Load registers.	000000	100110	10	C=10111
2.	Add 2's complement of C to A.	001001			
		001001	100110		
3.	Shift A&B.	000100	110011	11	
3.	Shift A&B.	000010	011001	01	
2.	Add C to A.	110111			
		111001	011001		
3.	Shift A&B.	111100	101100	00	
3.	Shift A&B.	111110	010110	10	
2.	Add 2's comp of C to A.	001001			
		000111	010110		
3.	Shift A&B.	000011	101011		

The result is: 0001110101 = 117

The controller for the multiplier can be implemented as a Mealy finite state machine (FSM) with only three states as shown below in Figure 2.

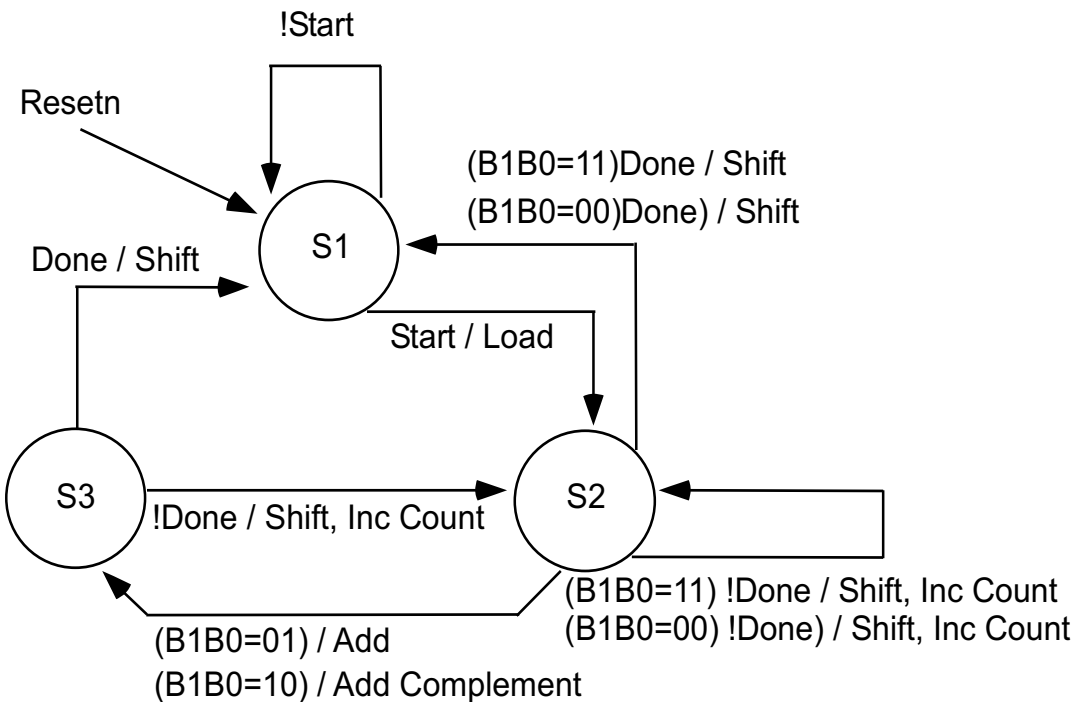

Figure 3. State Diagram for Booth Multiplier

In the reset state, S1, the controller waits for the Start signal. When the controller receives the Start signal, it should generate the Load signal to load registers B and C, clear bit B_0 and register A, and go into state S2. Note that this is a synchronous state machine so all state transitions must be synchronized to the system clock. In state S2, the controller will do the following:

- If B_1B_0 = 00 or 11, shift A and B. If the termination count has been reached, go to S1. Otherwise, increment the count and remain in S2.
- If B_1B_0 = 01, add C to A and go to S3.
- If B_1B_0 = 10, add the 2's complement of C to A and go to S3.

In state S3, registers A and B are shifted. If the termination count has been reached, the controller will go to S1; otherwise it will increment the count and go back to S2.

You may have noticed that the FSM operations in each state do not exactly coincide with the algorithm description given earlier. However, if you carefully compare the flows of the FSM and the algorithm, you will see that the FSM accomplishes the same operations with fewer state transitions.

Perform the following steps:

1. Write the Verilog code for an 8-bit Booth multiplier (n=8). Your Verilog module should have the following inputs and outputs:
   Inputs: Clock, Resetn, Start, Multiplier (Mplier), Multiplicand (Mcand)
   Outputs: Done, Product

2. Simulate your Verilog design with a test bench using the following test cases. The numbers are in twos-complement format.

   01100110 x 00110011
   10100110 x 01100110
   01101011 x 10001110
   11001100 x 10011001
   10000000 x 10000000
   11111111 x 11111111
   00000000 x 01010101
   01111111 x 01111111

   The test bench code is provided to you at the end of this write-up. Study the code so that you understand how it works.
3. Demonstrate your simulation to your TA and have him sign a verification sheet. Print your simulation waveforms for a single multiplication.
4. Synthesize your multiplier circuit and verify that it compiles without errors.

III. Lab Report

For your lab report, include the following:

- Lab Cover Sheet with signed TA verification for successful download of Part I, simulation of Part II, and synthesis of Part II.
- Complete Verilog source code for your Parts I and II.
- Simulation waveforms of your functional simulations.
- Resource report indicating how many FPGA resources were required for each the designs in Parts I and II.

IV. Grading Guidelines

- Prelab 10 points
- Part I Demonstration 35 points
- Part II Functional Simulation 50 points
- Part II Synthesis 30 points
- Lab Report 25 points

V. Appendix – Test Bench Code

```
//-------------------- tb_booth.v --------------------------------

module tb_booth;

parameter n=8;                          // n-bit Booth multiplier
parameter num_vectors=8;
reg Clock, Resetn, Start;
wire Done;
reg [n-1:0] Mplier, Mcand;
wire [n+n-1:0] Product;
reg [n+n-1:0] vectors [0:num_vectors-1];
integer i;


booth UUT (.Clock(Clock), .Resetn(Resetn), .Start(Start), .Mplier(Mplier), .Mcand(Mcand),
.Done(Done), .Product(Product));

initial                                 // Clock generator
 begin
   Clock = 1'b0;
   forever #20 Clock = ~Clock;    // Clock period = 40 ns
 end

initial                                 // Test stimulus
 begin
   Resetn = 1'b0;                       // synchronous reset of state machine
   Start = 1'b0;                        // set Start to 'false'
   #80 Resetn = 1'b1;                   // reset low for 2 Clock periods
   $readmemb ("testvecs", vectors);             // read testvecs file
   for (i=0; i<num_vectors; i=i+1) begin
          {Mplier, Mcand} = vectors[i];         // load Mplier, Mcand
          #20 Start = 1'b1;                             // Start = 'true'
          #80 Start = 1'b0;             // After 2 clock cycles, reset Start
          wait (Done==1);
          wait (Done==0);
          $display("Mplier=%h, Mcand=%h, Product=%h",Mplier,Mcand,Product);
   end
 end

endmodule


// File: testvecs
//
01100110_00110011
10100110_01100110
01101011_10001110
11001100_10011001
10000000_10000000
11111111_11111111
00000000_01010101
01111111_00000000
```