University of California, Davis
Department of Electrical and Computer Engineering

EEC180 DIGITAL SYSTEMS II Winter Quarter 2025

LAB 5: Sequential System Design

Objective: The purpose of this lab is to utilize the M9K memory blocks available on the DE10-Lite FPGA. You will learn to utilize the RAM in a digital system design.

Prelab

Read the lab carefully and show the steps associated with calculating the square root of 16 and 34 using the hardware shown in Part II of the lab (See Figure 4). Draw a state diagram for the control circuit in II. The control circuit should output a done signal when the square root is complete and wait until *St* = 0 before resetting. The N register should be loaded when *St* = 1.

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.

For Quartus to correctly infer your memory to M9K blocks, you must follow a specific Verilog implementation, using a total of five I/O signals: *clk* (clock), *addr* (address), *mwr* (memory write enable), *mdi* (memory data in), and *mdo* (memory data out).

Ensure that both memory read and memory write are synchronous to the clock.

```
reg [3:0] memory [15:0] /* synthesis ramstyle = "M9K" */;

always @ (posedge clk) begin
        if (mwr) memory[addr] <= mdi;           //write mem
        mdo <= memory[addr];                    // read mem
end
```

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.

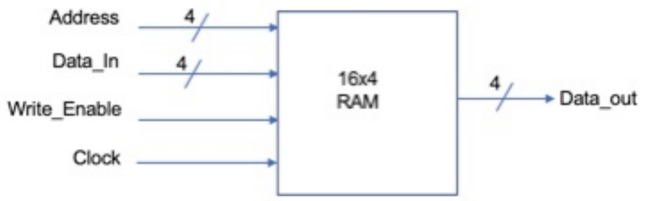


Figure 1. A 16x4 RAM module

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

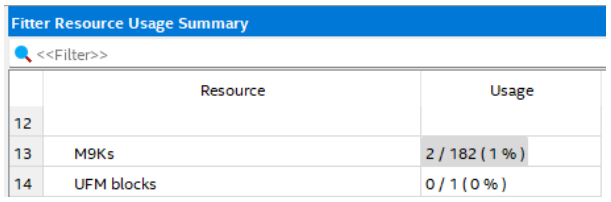
Your memory modules must operate as follows:

1. Only one RAM module (RAM1 or RAM0) can be written into at a time.
2. The Select switch (SW[8]) determines which memory module will be written.
3. A write operation will occur to the selected RAM module on a positive Clock transition when the Write Enable signal is active (high).
4. Both RAM modules can be read to their respective 7-segment displays simultaneously.
5. The RAM output data must be clocked (synchronous) for Quartus to use M9K blocks.

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* as shown above and in the link below https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/hdl/vlog/vlog_file_dir_ram.htm

2. Compile your program. **Demonstrate that your code infers M9K memory blocks by examining the Compilation Report.** Navigate to Fitter > Resource Section > Resource Usage Summary to view the number of M9Ks used by your design.



Fitter Resource Usage Summary

<<Filter>>

	Resource	Usage
12		
13	M9Ks	2 / 182 (1 %)
14	UFM blocks	0 / 1 (0 %)

Figure 2. M9K Usage in Fitter Resource Usage Summary

3. You can also go to Tools -> Chip Planner to see mapping of the resources on the actual chip and in this case, one of the M9K blocks (Yellow color) must be enabled as shown below.

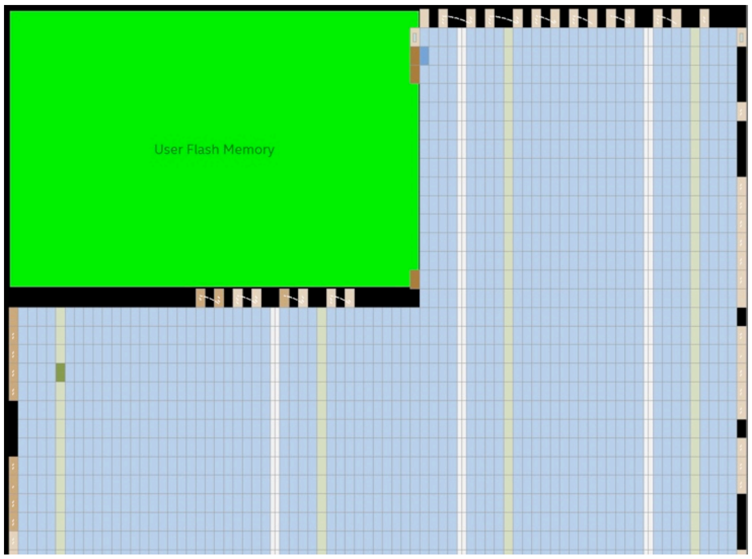


Figure 3. Instantiation of M9K as shown on Chip Planner

3. Modify your Verilog design to specify the initial contents of your RAM modules. To enable initialization features, navigate to Assignments > Settings. Then click the "Device/Board" button at the top right. Then click the "Device and Pin Options…" button. Finally, in the configuration pane, change the configuration mode to "Single Compressed Image with Memory Initialization (512kbits UFM)."

   Note the following from the MAX10 manual: https://www.intel.com/programmable/technical-pdfs/683865.pdf

   Only the following Intel MAX 10 configuration modes support memory initialization:
   - *Single Compressed Image with Memory Initialization*
   - *Single Uncompressed Image with Memory Initialization*

4. In your design, you can initialize the memory 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, then specify the initial content source using a synthesis pragma. These files may be created by navigating File > New > Memory Initialization File. Finally, in your code, add *ram_init_file* to your instantiation after the *ramstyle* to inform Quartus where to look for the initialization contents: https://www.intel.com/content/www/us/en/docs/programmable/683283/18-1/ram-initialization-file-for-inferred-memory.html

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

Memory IPs & In-System Memory Content Editor (Optional. Ungraded)

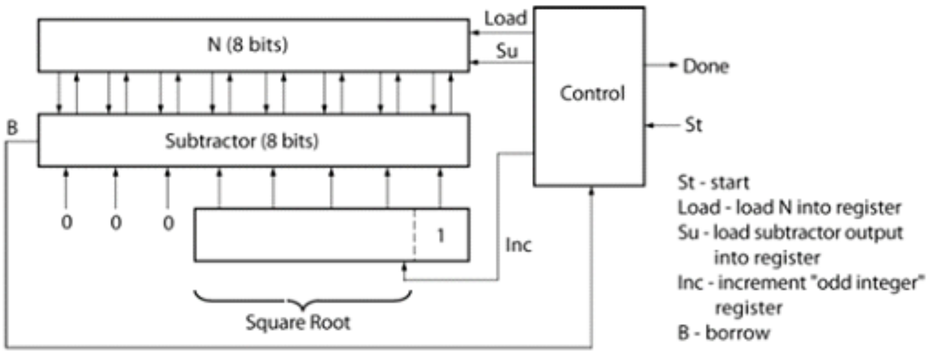
Complete the tutorial for using Memory IPs and the In-System Memory Content Editor tool located in Canvas under Lab 5 > Extra Credit > Extra_Credit_Part_I.pdf

II. Integer Square Root Computation

You will design a circuit that finds the square root of an 8-bit unsigned binary number *N*, using the method of subtracting out odd integers. To find the square root of *N*, we subtract 1, then 3, then 5, and so on, until we can no longer subtract without the result going negative. The number of times we subtract is equal to the square root of *N*. For example, to find the square root of 27:

27 – 1 = 26; 26 – 3 = 23; 23 – 5 = 18; 18 – 7 = 11; 11 – 9 = 2; 2 – 11 (can't subtract).

Since we subtracted five times, the square root of 27 is 5 (rounded down). Note that the final odd integer that was supposed to be subtracted was $11_{10} = 1011_2$, and this consists of the final square root result ($101_2 = 5_{10}$) followed by a 1.



N (8 bits)
Load
Su
Control
Done
St
B
Subtractor (8 bits)
0
0
0
1
Inc
Square Root
St - start
Load - load N into register
Su - load subtractor output into register
Inc - increment "odd integer" register
B - borrow

Figure 4. Square Root Module Block Diagram

Assume you have a 16x8 RAM (expand your RAM from Part I from 4 bits per word to 8) that holds the inputs. Connect the *St* signal to a switch on the DE-10 board. Each time the switch is toggled, you can display a new result until we are done with all 16 words. After the 16 values, the memory will wrap around back to address 0. Your design should read each element from the RAM, perform the square root computation, and output the input *N* and the square root result *Sqrt* on the HEX display of the board.

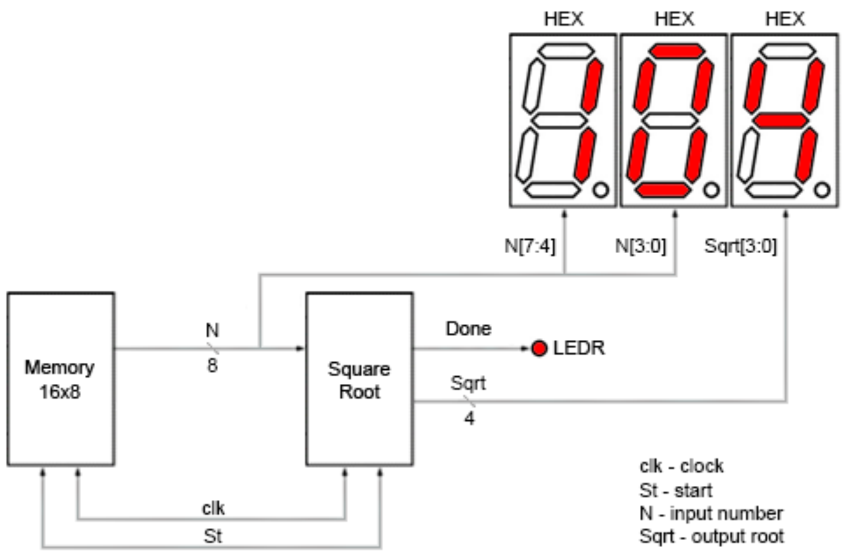


Figure 5. Top-Level Block Diagram for FPGA Synthesis

Perform the following steps:

1. Write the Verilog code for an 8-bit Square Root unit, as shown in Figure 4. Your Verilog module should have the following inputs and outputs:
   Inputs: *Clock*, *ResetN*, *St*, *N*
   Outputs: *Done*, *Sqrt*

2. Simulate your Verilog design with a test bench using the following test cases.

00000001 (1_{10})	00000000 (0_{10})
00000100 (4_{10})	00000110 (6_{10})
00001001 (9_{10})	00001101 (13_{10})
00010000 (16_{10})	00010101 (21_{10})
00011001 (25_{10})	00011011 (27_{10})
00100100 (36_{10})	00101100 (44_{10})
00110001 (49_{10})	11100001 (225_{10})
01000000 (64_{10})	11111111 (255_{10})

The test bench code is provided to you at the end of this write-up. Utilize the code to ensure your module is working correctly. Study the code so that you understand how it works. Make modifications as needed to make the test bench work with your module.

3. **Demonstrate your simulation to your TA** and have them sign a verification sheet. Print your simulation waveforms for a single square root computation.

4. Use your Square Root module to create a top-level Verilog file that will be used to program the FPGA. Initialize your modified 16x8 RAM to store the test cases. Refer to Figure 5.

5. Synthesize your square root circuit and verify that it compiles without errors. **Demonstrate your FPGA to your TA showing all test cases**.

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 all functional simulations.
- Resource report indicating how many FPGA resources were required for each of the designs in Parts I and II.

IV. Grading Guidelines

• Prelab	15 points
• Part I Demonstration	50 points
• Part II Functional Simulation	75 points
• Part II Synthesis	35 points
• Lab Report	25 points

V. Appendix

Test Bench Code

```
//------------------------- tb_sqrt.v -------------------------------
module tb_sqrt;

parameter num_vectors=16;            // 16 words
reg Clock, Resetn, Start;
wire Done;
reg [7:0] InputVal;
wire [3:0] OuputSqrt;
reg [7:0] vectors [0:num_vectors-1];
integer i;

squareroot  UUT  (.Clock(Clock),  .Resetn(Resetn),  .Start(Start),  .InputVal(InputVal),
.Done(Done), .OutputSqrt(OutputSqrt));

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
     InputVal = vectors[i];          // load input value
     #20 Start = 1'b1;               // Start = ‘true’
     wait (Done==1);
     $display("Input=0x%h, SqRt=0x%h", InputVal, OutputSqrt);
     #20 Start = 1'b0;               // After data is captured, reset Start
     wait (Done==0);
  end
  $finish;
 end

endmodule
```

Test Vectors File

```
// File: testvecs

00000001
00000100
00001001
00010000
00011001
00100100
00110001
01000000
00000000
00000110
00001101
00010101
00011011
00101100
11100001
11111111
```