EE/CE/TE 1202– Introduction to Electrical Engineering II

Dr. Amir Khoobroo

Lab 6: Toy Processor with Memory on the Basys2 Board

Assigned: 11/30/2015 **Due: 12/07/2015**

In the last lab, you will be using the Basys2 board. Hopefully you have tested your toy processor with the memory unit with *ISim* several times. Now we are going to synthesize our design and upload it onto the Basys2 board. We will also write 2 programs in machine code and execute them on the Basys2 board—with your processor.

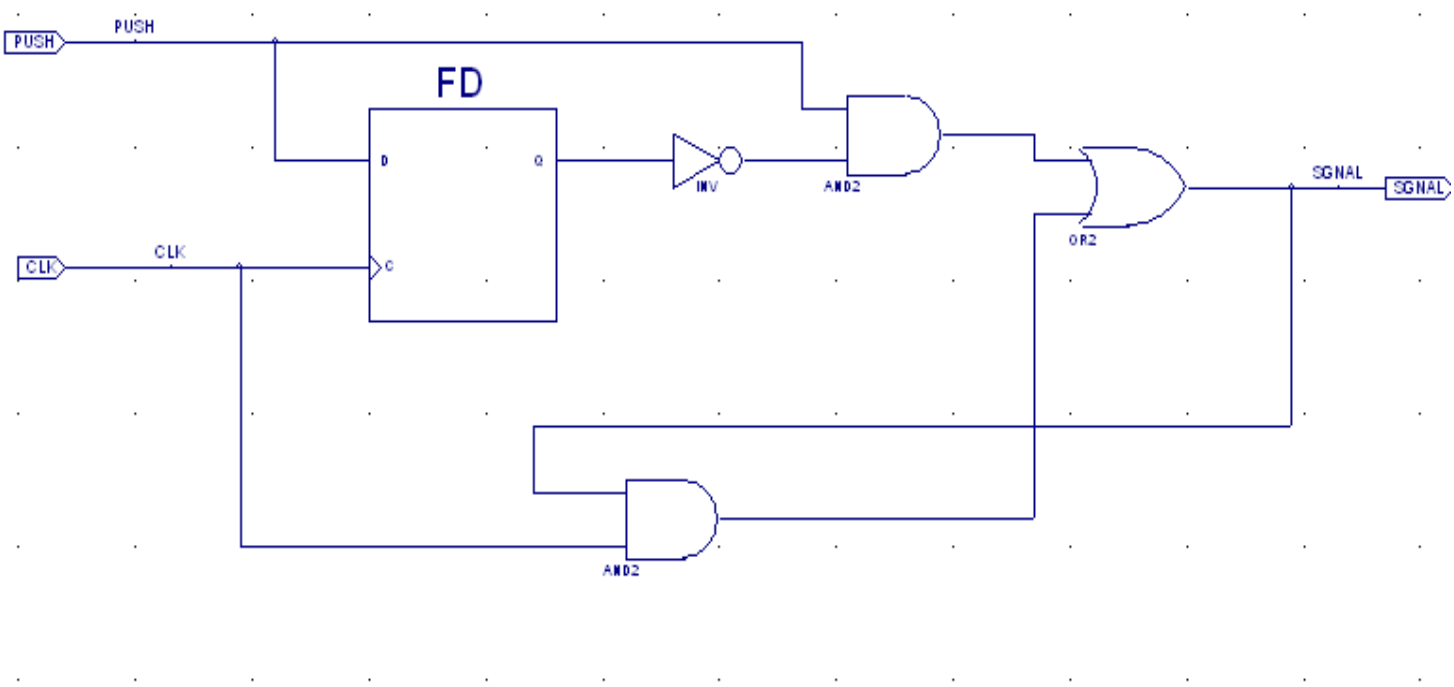
This lab is divided into 2 parts. In the first part, you will finish the last bit of circuit design to adapt your toy processor to the Basys2 board. In the second part, you will write and load the program onto the Basys2 board.

In this lab, it is important to make sure your project before this point is correct. And it actually saves you time if you test each component you build with *ISim* before you go on to the next step.

Part 1:

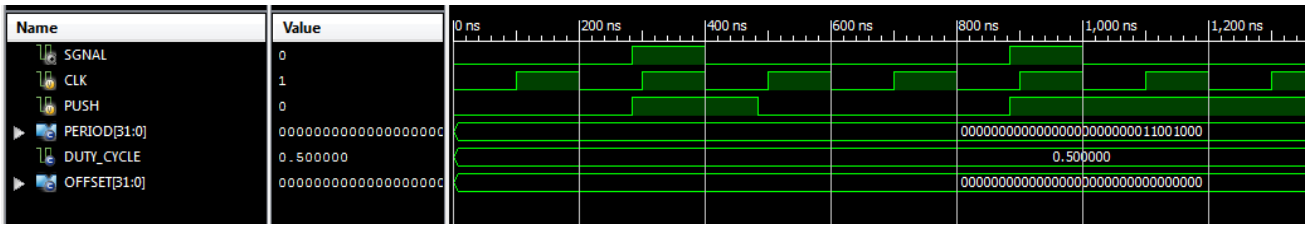
The clock we will use for our toy processor will be provided by the FPGA. Because this clock is very fast, a program would run in a couple of milliseconds, and we would not be able to see anything but the final result. To get by this, we can use a push-button to generate a clock edge to allow us to manually step through the simulation. Xilinx will not allow the clock itself to be connected to a push-button, so we will create a finite state machine (FSM) for this purpose. This also accounts for the fact that one push of the button will probably be longer than the normal 100ns clock time, so it is only the initial downward push of the button that is accounted for.

1. Create a schematic “clk_signal_sch.sch”.
2. Draw your schematic as follows.



Symbol *clk_signal_sch*

3. Simulate the schematic. Your results will look like:



With the simulation file "clk_signal_tb.v"

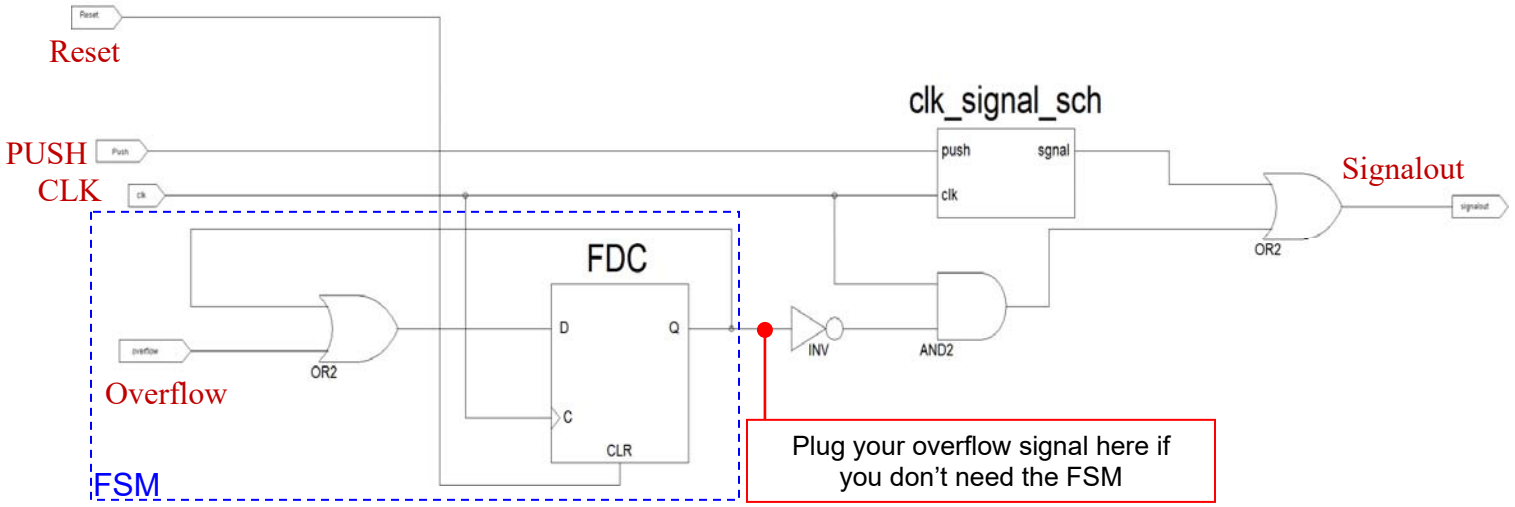
```
// clk_signal_tb.v
        `timescale 1ns/1ps

        module clk_signal_tb;
          reg CLK = 1'b0;
          reg PUSH = 1'b0;
          wire SGNAL;

          parameter PERIOD = 200;
          parameter real DUTY_CYCLE = 0.5;
          parameter OFFSET = 0;

          initial    // Clock process for CLK
          begin
            #OFFSET;
```

```
    forever
    begin
      CLK = 1'b0;
      #(PERIOD-(PERIOD*DUTY_CYCLE)) CLK = 1'b1;
      #(PERIOD*DUTY_CYCLE);
    end
  end

  clk_signal_sch UUT (
    .CLK(CLK),
    .PUSH(PUSH),
    .SGNAL(SGNAL));

  initial begin
    // ------------  Current Time:  285ns
    #285;
    PUSH = 1'b1;
    // -------------------------------------
    // ------------  Current Time:  485ns
    #200;
    PUSH = 1'b0;
    // -------------------------------------
    // ------------  Current Time:  885ns
    #400;
    PUSH = 1'b1;
    // -------------------------------------
    // ------------  Current Time:  1485ns
    #600;
    PUSH = 1'b0;
    // -------------------------------------
  end
endmodule
```

4. Generate a symbol for clk_signal_sch.sch

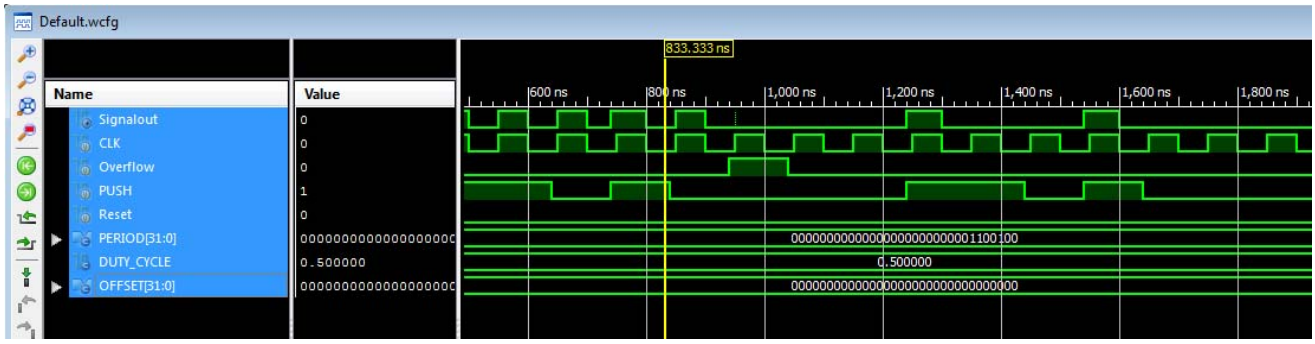
Initially when we are in state zero of the toy processor, it takes 256 clock cycles to transition to the next state. Certainly you don't want to push a button for 256 times just to get out of state zero. We would like the clock to be automatic when the memory is bootstrapping and to be the manual push-button after that. The easy way to do it is to take the overflow signal from the address counter in the memory circuit and connect it to a FSM that keeps it high once it goes high. Then we can create a circuit that ignores the actual clock stepping signal and receives the clock itself as long as the overflow signal is low. Once it goes high, the circuit reverts to the normal clock stepping FSM mentioned above. Of course, if your overflow signal from last lab already shows this behavior (that is, if it stays up forever after it goes high the first time), you don't need the FSM, just the small combinatory circuit that follows it.

1. Create a schematic “BypassClk.sch”.
2. Draw your schematic as follows.



Circuit to Bypass Clock Step during Initialization

3. Simulate the schematic. Your results will look like:



With simulation testbench “BypassClk_tb.v”

```
//BypassClk_tb.v
`timescale 1ns/1ps

module BypassClk_tb;
   reg CLK = 1'b0;
   reg Overflow = 1'b0;
   reg PUSH = 1'b0;
   reg Reset = 1'b0;
   wire Signalout;

   parameter PERIOD = 100;
   parameter real DUTY_CYCLE = 0.5;
   parameter OFFSET = 0;
```

```
initial    // Clock process for CLK
begin
   #OFFSET;
   forever
   begin
      CLK = 1'b0;
      #(PERIOD-(PERIOD*DUTY_CYCLE)) CLK = 1'b1;
      #(PERIOD*DUTY_CYCLE);
   end
end

BypassClk UUT (
   .CLK(CLK),
   .Overflow(Overflow),
   .PUSH(PUSH),
   .Reset(Reset),
   .Signalout(Signalout));

initial begin
   // ------------  Current Time:  140ns
   #140;
   Reset = 1'b1;
   // ------------------------------------
   // ------------  Current Time:  340ns
   #200;
   Reset = 1'b0;
   // ------------------------------------
   // ------------  Current Time:  440ns
   #100;
   PUSH = 1'b1;
   // ------------------------------------
   // ------------  Current Time:  640ns
   #200;
   PUSH = 1'b0;
   // ------------------------------------
   // ------------  Current Time:  740ns
   #100;
   PUSH = 1'b1;
   // ------------------------------------
   // ------------  Current Time:  840ns
   #100;
   PUSH = 1'b0;
   // ------------------------------------
   // ------------  Current Time:  940ns
   #100;
```

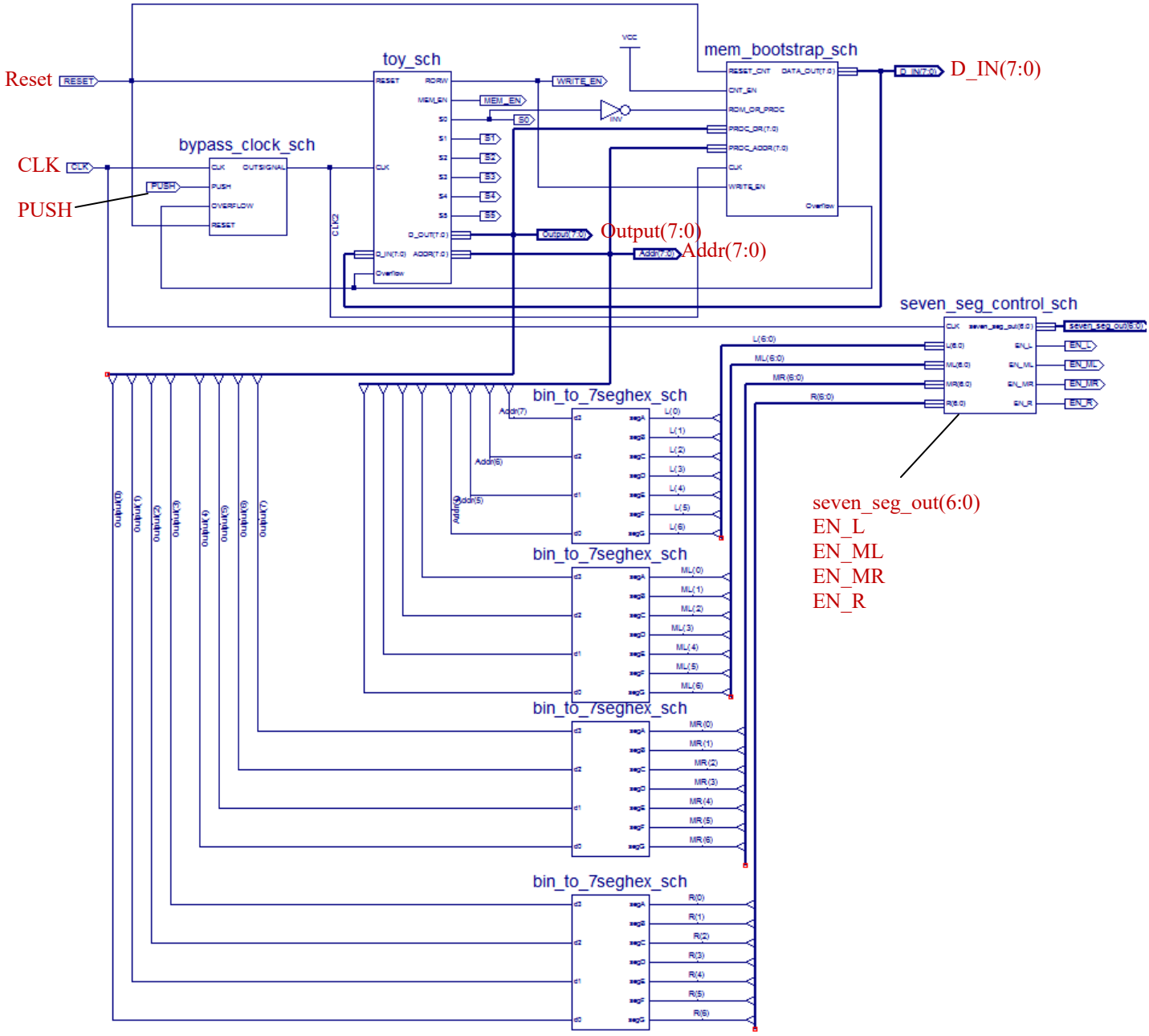
```
        Overflow = 1'b1;
        // ------------------------------------
        // -------------  Current Time:  1040ns
        #100;
        Overflow = 1'b0;
        // ------------------------------------
        // -------------  Current Time:  2140ns
        #200;
        PUSH = 1'b1;
        // ------------------------------------
        // -------------  Current Time:  2340ns
        #200;
        PUSH = 1'b0;
                #100;
                PUSH = 1'b1;
                #100;
                PUSH = 1'b0;
        // ------------------------------------
    end

endmodule
```

4. Generate a symbol for BypassClk.sch

Part 2: Programming the Basys2 Board

Before you move on, you need to include **bin_to_7seghex_sch** and **seven_seg_control_sch** from the 4-bit ALU from Lab 2. If they are already in your project, there is no need to re-include them (it wouldn't make a difference either way). Also, you will need to replace your ROM array with the one in the Lab6 zip file. This is the same as the ROM array that is already in your project, but each ROM 32x1 instance has been labeled manually (a lengthy manual process of which we are sparing you).

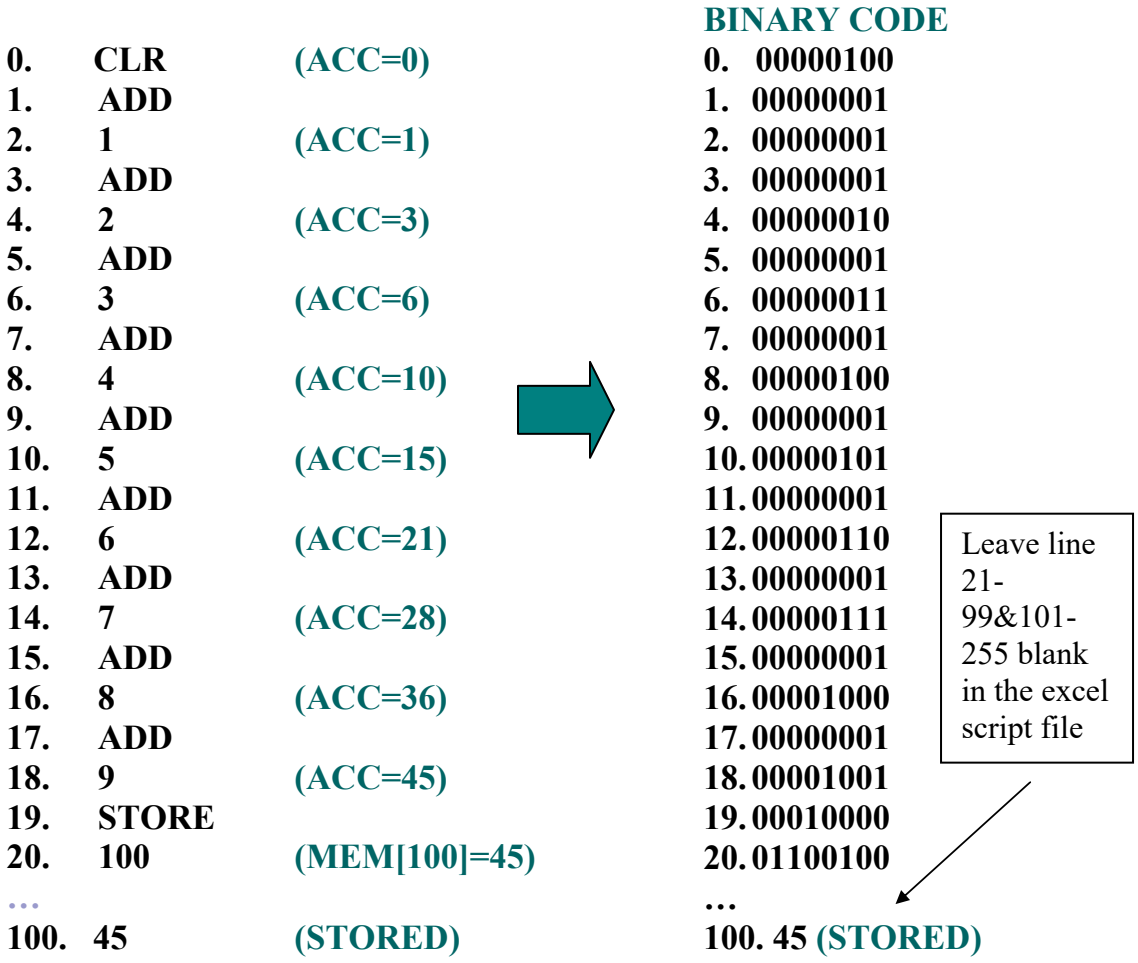
Now you are ready to make the final schematic of your ToyProcessor. Open the schematics from the previous lab and change it as follows:



Programming the first Serial Addition Program

Understanding how a program is run on the board is an important part of the final processor. We will illustrate the process using a program that adds up the digits {0,…,9}.

Writing the program entails simply laying out the desired instructions and their accompanying operands/addresses and then converting everything into binary. Remember the convention that the operand/address always follows directly after the instruction in memory. Also, a good programming practice is to start off with a clear instruction at memory location zero. For the summation program, there is the clear and then nine add instructions, each followed by one of the digits {1,…,9}. Finally, the program ends by storing the resulting sum in memory location 100. As you can see below, each line is then written in binary, for the convenience of later initializing it into the ROM array.

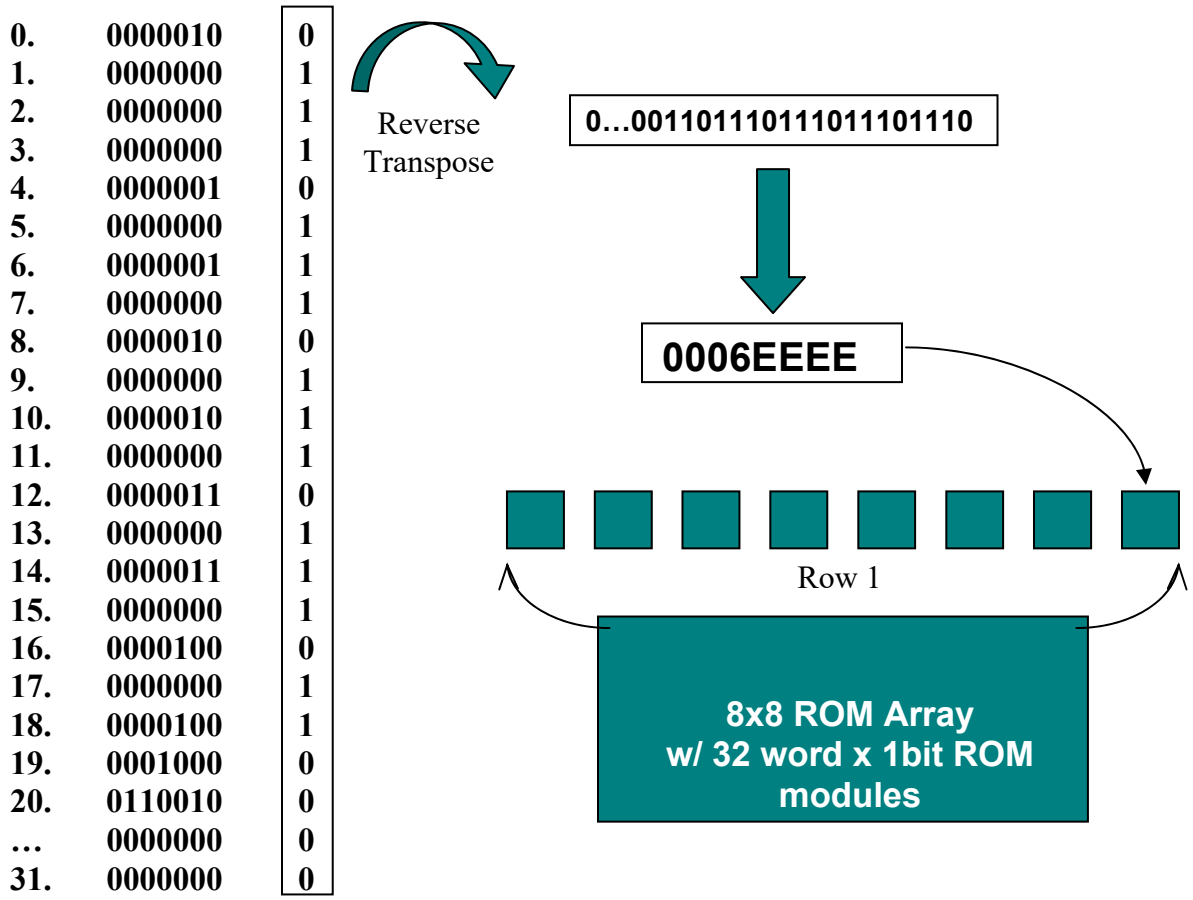


Writing the Program

Luckily, we have an Excel script to help us initializing the ROM array. The script lets you insert a program in binary and it will tell you the appropriate 8-hexadecimal-digit code to use when initializing each of the 64 ROM components. However, it is also important to understand what is going on behind the script. Each 32 word x 1 bit ROM module is initialized by taking the reverse transpose of the 32x1 bit matrix and converting

it into hexadecimal. The result is plugged into the appropriate ROM component as shown in the diagram. For implementation to the board, a User Constraint File (UCF) must be created that contains init statements, such as:

INST "BOOTSTRAP/ROMARRAY/ROM11" INIT = 00000000;



Programming the ROM

To use the Excel script to initialize the ROM, open the Excel file named ROM_Initialization_Script. In the second column to the left, enter your code instruction one line for each row. If this is your first time using Excel (for Excel with version earlier than 2003), you may need to install the analysis tool pack of Excel. To do that, select **Tools → Add-Ins → Analysis ToolPak** in Excel, and click **OK**. A corresponding UCF code will generate in the highlighted fields. After you are done you excel should look like as follows.

	Address	Instruction				
1	Address	Instruction				
2	0	100				
3	1	1	Copy and paste the content of the box below			
4	2	1				
5	3	1	INST BOOTSTRAP/ROMARRAY/ROM11 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM12 INIT = 00100000;	INST BOOTSTRAP/ROMARRAY/ROM	
6	4	10	INST BOOTSTRAP/ROMARRAY/ROM21 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM22 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM	
7	5	1	INST BOOTSTRAP/ROMARRAY/ROM31 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM32 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM	
8	6	11	INST BOOTSTRAP/ROMARRAY/ROM41 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM42 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM	
9	7	1	INST BOOTSTRAP/ROMARRAY/ROM51 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM52 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM	
10	8	100	INST BOOTSTRAP/ROMARRAY/ROM61 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM62 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM	
11	9	1	INST BOOTSTRAP/ROMARRAY/ROM71 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM72 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM	
12	10	101	INST BOOTSTRAP/ROMARRAY/ROM81 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM82 INIT = 00000000;	INST BOOTSTRAP/ROMARRAY/ROM	
13	11	1				
14	12	110				
15	13	1				
16	14	111				

Copy the whole highlighted section to the right. Then you can paste that whole section at the end of the UCF file of the overall project as shown in the attached Appendix. Or you can just input the INIT value by hand in the schematic with values corresponding to each ROM block generated by the excel script.

The name BOOTSTRAP/ROMARRAY/ROMxx is essentially the path from the top level schematics to each 32x1 ROM. This means you should rename your mem_bootstrap_sch and the ROM array inside it to match, like this:

1) Open your final toyprocessor schematics, then double-click on the mem_bootstrap_sch block.
2) Under InstName, change the value to BOOTSTRAP then press OK.
3) Now open you mem_bootstrap_sch schematics and double-click on the ROM_array block.
4) Under InstName, change the value to ROMARRAY and press OK.

When generating the UCF you should think about which LEDs, switches and buttons you would like to use for your inputs and outputs. Choose wisely, as it can make debugging and operating your toy processor much easier. Make sure you show all the important outputs on LEDs (such as the value of the accumulator), and that you connect all the important inputs to buttons/switches (such as the manual clock). Make sure your toy processor schematic is set as top module and that there are no other UCF files in your project. Now add your new UCF, then load that on Basys2 board as before.

Congratulations!!! Your Basys2 board is programmed. You need to test it by yourself.

Final Task: Programming the Self-Modifying Program

Write a second UCF file where the ROM is initialized to the second program that we saw in class (self-modifying code), which also performs addition of digits 0 through 9. A self-modified program will be.

100	CLR (ACC=0)
1	ADD
1010	1 0 (ACC=10, becomes 9) (Running Index)
10	SUB
1	1 (ACC=9) (Current Number)
10000	STORE
10	2 (Update Running Index)
1000	BNZ
10010	18 (Done if Current Index is 0)
1	ADD
0	0 (Running Sum, becomes 9)
10000	STORE
1010	10 (Update Running Sum)
10000	STORE
1100100	100 (Keep a Copy in Final Location)
100	CLR (ACC=0)
1000	BNZ
1	1 (Forced Loop to Location 1)

Using ROM Excel file, generate code again and paste to UCF file then program it onto board.

Appendix

Here is an example UCF file for the first program. Yours may look different if you used other names for the signals and for the ROMS.

###Auto generated UCF file for BASYS2 Board

NET S0 LOC=M5; # Discrete LED 0 (active high)
NET S1 LOC=M11; # Discrete LED 1 (active high)
NET S2 LOC=P7; # Discrete LED 2 (active high)
NET S3 LOC=P6; # Discrete LED 3 (active high)
NET S4 LOC=N5; # Discrete LED 4 (active high)
NET S5 LOC=N4; # Discrete LED 5 (active high)
NET MEM_EN LOC=P4; # Discrete LED 6 (active high)
NET Overflow LOC=G1; # Discrete LED 7 (active high)

BASYS2 7-Segment LED:
BASYS2   digit enables:
NET EN_L LOC=K14; # left digit enable (active low)
NET EN_ML LOC=M13; # middle left digit enable (active low)
NET EN_MR LOC=J12; # middle right digit enable (active low)
NET EN_R LOC=F12; # right digit enable (active low)

BASYS2   segment enables:
NET seven_seg_out<0> LOC=L14; # LED display segment a (active low)
NET seven_seg_out<1> LOC=H12; # LED display segment b (active low)
NET seven_seg_out<2> LOC=N14; # LED display segment c (active low)
NET seven_seg_out<3> LOC=N11; # LED display segment d (active low)
NET seven_seg_out<4> LOC=P12; # LED display segment e (active low)
NET seven_seg_out<5> LOC=L13; # LED display segment f (active low)
NET seven_seg_out<6> LOC=M12; # LED display segment g (active low)

NET Reset LOC=M4; # Pushbutton 2 (active high)
NET PUSH LOC=A7; # Pushbutton 3 (active high)

NET CLK LOC=B8; # clock

INST BOOTSTRAP/ROMARRAY/ROM11 INIT = 00000000;
INST BOOTSTRAP/ROMARRAY/ROM12 INIT = 00100000;
INST BOOTSTRAP/ROMARRAY/ROM13 INIT = 00100000; …….