CME341 Assignment 7

```
Revised Oct 27, 2014: Added a note to the question on testing the
                      program memory that explains the address signal
                      must stay valid for 2 ns after the clock.
Revised Oct 25, 2017: Revised instructions on creating program memory
                      to be consistent with Quartus Prime.
Revised Nov. 2, 2017: Added list of libraries that must be include in
                      the compilation of the testbench for Question 4.
Revised Oct 26, 2020: Clarified the instructions on building the program
                      memory and on using the .hex editor.
```

1. NOTE: This question has been placed in this assignment because it solidifies the student's understanding of how the microprocessor works. The result will be useful for testing circuits that will be designed in later assignments. While a student may think it prudent to defer doing this question, it will be of most benefit to do this question at this time.

   The assembly program below could well be part of a suite of test programs that tests the operation of the microprocessor. The program below is designed to test the "mov" instruction.

   (a) Explain in words the flow of (the logic used in) the test program. In other words "What does the test program do?".

   (b) Determine a good input sequence for *i_pins*. (In determining a good sequence it should be noticed that *i_pins* is only read twice and that the two values read should check to see that no bit is stuck at 1 or 0 in any register.) List the values of the *i_pins* on the positive edges of the clock starting with the second edge after *reset* goes low. (At the time of the first clock edge after "reset" goes low the "sync_reset" is still high. The second positive edge of the clock is the first positive edge after "sync_reset" goes low.) Using the *i_pins* waveform you specified, determine the sequence of values for the output register "o_reg".

   (c) Hand assemble the program into machine code. Remember that a "mov" instruction becomes a move *i_pins*, to the "dst" register instruction when the "dst" and "src" registers are the same. The

exception is when the "src" and "dst" are both 4H. In this case the "src" register is r and the "dst" register is o_reg and the instruction is interpreted as "move contents of r to o_reg", i.e., o_reg = r.

```
*********************************************
*
*  PROGRAM THAT TESTS THE MOV INSTRUCTION
*
*********************************************
Column 1 is the address in hex.
Column 2, which starts after the colon, is the assembly instruction
The comment field begins after a double slash, i.e. //

00: x0 = 0; // load x0 with the constant 0
01: x1 = 0;
02: y0 = 0;
03: y1 = 0;
04: o_reg = 0;
05: m = 0;
06: i = 0;
07: x0 = i_pins; // move the value of i_pins to x0
                 // value of i_pins comes from FPGA pins.
                 // Of course in this class i_pins is simulated
                 //  in Modelsim-Altera with ‘‘initial’’ procedure
08: x1 = x0;
09: x0 = i_pins; // move the value of i_pins to x0
                 // NOTE: should change value of i_pins after
                 // 8th rising edge of clock so that a different
                 // value gets moved to x0 on this instruction
0A: y0 = x1;
0B: x1 = x0;
0C: y1 = y0;
0D: y0 = x1;
0E: m = y1;
0F: y1 = y0;
10: i = m;
11: m = y1;
```

```
12: o_reg = i;
13: i = m;
14: o_reg = i;
15: r = -x0;
16: o_reg = r;
17: x0 = r;
18: r = -x0;
19: o_reg = r;
1A: o_reg = F;
1B: jump 00H;
```

The answer to the hand assembly into machine code is given at the end of the assignment.

2. Create an Intel-Format hexadecimal file for the "move instruction" test program. The program memory is a 256 word by 8 bits/word ROM. The procedure for creating an Intel-Format hex file in Quartus II is given below.

   (a) Start the Intel-Format hexadecimal file editor by:

   (b) Select File → new. This brings up a window with a table of contents.

   (c) Select the "memory files" heading and then subheading "Hexadecimal (Intel-Format) File". Clicking OK brings up a miniwindow where the word size and the length of the RAM/ROM is specified. Specify 256 words by 8 bits/word and click OK. This opens the Intel-Format hexadecimal file editor.

   (d) The hexadecimal file editor presents a spread sheet like interface with the left column being "address" cells and the remaining columns being "memory word" cells. Each memory word cell represents a word in memory. The memory address for a memory word cell is obtained by adding the address offset given at the top of the corresponding column to the address cell of the corresponding row.

   It is probably easiest to detach the memory editor page from the Quartus window (`window` → `detach` and then explore the options

under the menu items at the top of the page. The most useful options are under the menu item "view".

The radix of the address field and the memory word field should be changed to hexadecimal. It can be done by pulling down the items under "view" and selecting appropriately or it can be done as follows: Right click on one of the address offsets (say +0) at the top of a memory word column. Set the address radix, which is the radix for the first column, to hexadecimal. Then set the radix for the memory word field (referred to as memory radix) to hexadecimal.

(e) Enter the machine code program into the memory word cells and then save the file as rom.hex.

3. Design the program memory module, which is just a ROM that has it address lines buffered, using the megawizard. The ROM should be 256 words by 8 bits/word. The address lines have to be buffered as there is no option to do otherwise on Cyclone IV FPGAs. Make sure the data outputs are not registered. The procedure for creating a ROM in Quartus II is given below:

   (a) Select Tools → IP Catalog. This will bring up a window titled "IP Catelog"

   (b) Expand the "basic functions" menu, which is listed under "Library". Then expand the menu for "On Chip Memory". Doing this will provide a drop-down list of different memory options.

   (c) Select "ROM 1-port". This will bring up a window asking for the name of the IP variation file. The IP variation file is the file that will contain the Verilog prototype of the ROM. Name the IP variation file `program_memory`.

   Make sure the IP variation type is Verilog and then click OK. This will bring up a progression of windows that asks a series of questions.

   (d) Fill out the first page making the output bus 8 bits, the address bus 8 bits (256 words), the memory block type "auto" and the clocking "a single clock". Then click next.

   (e) Progress through the sequence of windows answering the questions appropriately. Make sure the output of the ROM is not registered.

Make sure you specify the initialization file, which is the file used to initialize the ROM at compile time. This file must be of type .hex, which is hexadecimal (Intel format) or type .mif, which is short for memory initialization file.

This is the file you created in the previous question.

(f) A list of files that can be generated by Quartus will be presented the end of the sequence of question pages.

You have the option of generating them or not by whether or not you check the box. You need only the .v file. Uncheck all other boxes.

4. Create a test bench in Modelsim-Altera to test your program memory module, which is basically is to verify that the compiler has burned the contents of your rom.hex file into the ROM. In the test bench make a counter to generate a sequence that goes from 8'H00 to 8'HFF for the address input.

Note: When the .vo file generated by Quartus includes an IP variation file (like program_memory.v) then the testbench needs the library `aletra_mf_ver` in addition to libraries `cycloneive_ver` and `altera_ver`. In sequencing through the IP core generator to generate the IP variation file there is a page near the end that states the library `altera_mf` is needed in the simulator. However, since Verilog HDL is used, the file that is actually needed is `altera_mf_ver` and not `alerta_mf`.

Note: The .vo file for the program memory is an accurate model of the hardware. This means it will not work properly if the set up or hold time for the address input to the ROM is not met. By definition of set up and hold times the address input must not change in the time interval from T_{su} prior to the clock to T_h after the clock, where T_{su} is the set up time and T_h is the hold time. The set up and hold times are dependent on the FPGA family and the speed grade of the device. For this assignment assume that T_{su} is 5 ns and T_h is 2 ns.

The focus of this class is synthesis. To that end it would be nice if the generation of signals in the test bench are modelled on hardware

concepts. For example the Verilog HDL for the address signal could model a hardware counter. This could be done as follows:

```
initial
    address = 8'd0;

always @ (posedge clk)
    #2 address = address +8'd1; // address changes 2 ns after the clk
```

where it is assumed the time scale is set to 1 ns.

A second way to introduce a 2 ns hold time, which is not the preferred method for this class, is:

```
initial
begin
        address = 8'd0;
     #2 address = 8'd0; // make an offset of 2 ns
forever
  #1000 address = address + 8'd1; // increment address every micro second
end
```

5. This question has three objectives: get the student some experience using the hex file editor, gain more understanding of the operation of the microprocessor, especially with respect to the data memory and to get a feel for the contents of the program ROM, which is machine code.

   Write a program for your EE431 microprocessor that tests the data memory (not the program memory). The program you create should be a sequence of pseudo code instructions, like "move X0 to DM".

   Your test program should write something to every memory location in the data RAM and then read every memory location and perform some sort of check to verify the values read were the same the values written.

   After creating the program in pseudo code, hand assemble the program into machine code and create an Intel-Format hexadecimal file (Name the file data_mem_test_prog.hex). When the time comes to test your data memory, this hex file can be "burned" into your program memory ROM and then executed.

Answers

```
*********************************************
*
*      MACHINE CODES FOR THE PROGRAM THAT TESTS
*               THE MOV INSTRUCTION
*
*********************************************

Column 1 is the address in hex.
Column 2, which starts after the first colon, is the machine code
Column 3, which starts after the second colon, is the assembly instruction
The comment field begins after a double slash, i.e. //

00: 00: x0 = 0; //load instruction
01: 10: x1 = 0;
02: 20: y0 = 0;
03: 30: y1 = 0;
04: 40: o_reg = 0;
05: 50: m = 0;
06: 60: i = 0;
07: 80: x0 = i_pins; // mov instruction
08: 88: x1 = x0;
09: 80: x0 = i_pins;
0A: 91: y0 = x1;
0B: 88: x1 = x0;
0C: 9A: y1 = y0;
0D: 91: y0 = x1;
0E: AB: m = y1;
0F: 9A: y1 = y0;
10: B5: i = m;
11: AB: m = y1;
12: A6: o_reg = i;
13: B5: i = m;
14: A6: o_reg = i;
```

```
15: C0: r = -x0;  // ALU instruction
16: A4: o_reg = r;
17: 84: x0 = r;
18: C0: r = -x0;
19: A4: o_reg = r;
1A: 4F: o_reg = F;
1B: E0: jump 00H;
```