Brown University
School of Engineering
EN1640 Design of Computing Systems
Professor Sherief Reda
Assignment 06 (150 points) – 1- or 2-person team
Final submission due on April 5 (Milestones on 3/15 and 3/22)

In this lab you are required to design and boot a single-cycle RISCV processor. To make this lab manageable, it will be split into smaller components, where you implement increasing sets of RISCV instructions by certain due dates. You will need to submit only one lab report on the due date of March 22. However, you will need to demonstrate progress by an intermediate date of March 15 by submitting the .qar of your project on that date.

- Make sure to go through all the guidelines at the end of this assignment before working on it.
- **2-person teams:** There are 30 points allocated based on your actual processor speed on the chip. You will get the full 30 points if your design achieves 80 MHz or higher, with 0.5 point deducted for every 1MHz less; for example, running at 50 MHz will lead to 15 points deducted. If interested, any speed improvements over 80 MHz will give you bonus points in the lab portion of the class, with 0.5 bonus point per MHz improvement over 80.
- **1-person teams:** You will get the full grade once your design runs at 50 MHz. If interested, any speed improvements over 50 MHz will give you bonus points in the lab portion of the class, with 0.5 bonus point per MHz improvement over 50.

a) (March 15) Implement some of the building blocks of the processor, including: (i) the register file module and test it it using Modelsim to verify correctness; (ii) instantiate the RAM (please see instructions below), and create a test in the FPGA board where you write a value into a target address in it using its address and data ports, and then read it back using the "In-System Memory Content Editor" to verify it is written in there.

b) (March 22) Implement processor with the following instructions: **`ADD, ADDI, SW, LW, ADD, SUB, AND, OR, MUL, SLLI.`** Validate the design by booting the processor and running the following two programs. Note that you can use the first program to test your processor after implementing the first three instructions (ADDI, ADD, SW). It is not a requirement; however, it is a good idea to create ModelSim testbenches and verify your code with it first before you verify it on the board. You can then continue expanding the instruction set and eventually test with the second program. Read the RAM memory contents after execution is finished. Guideline for reading the RAM is at the end of this assignment. Contrast the memory contents with the results you would get from the RISCV simulator. (Let's use the opcode 11111111 for HALT and you can get the machine code of these programs using the Venus tool). The .qar project you submit by March 22 should correct execution of Program 1 and/or Program 2 (speed is not important at this point).

Program 1:

```
addi   x1, x0, 0
addi   x2, x0, 16
addi   x3, x0, 100
addi   x4, x0, 8
add    x5, x1, x2
add    x6, x3, x4
sw     x5, 0(x1)
sw     x6, 4(x2)
HALT
```

Program 2:

```
addi   t0, x0, 8
addi   t1, x0, 15
sw     t1, 0(t0)
add    t2, t1, t0
sub    t3, t1, t0
mul    s1, t2, t3
addi   t0, t0, 4
lw     s2, -4(t0)
sub    s2, s1, s2
slli   s2, s2, 2
sw     s2, 0(t0)
halt
```

c) (April 5) Implement processor with the following instructions: **add, addi, sw, lw, add, sub, and, or, slli, mul, beq, bne, jal, jalr**

Validate the design by booting the processor and running the following code which calculates the factorial code. Make sure to initialize the SP to the end of your memory. Read the memory contents after execution is finished. Contrast the memory contents with the results you get from the RISCV simulator.

```
        addi a0, x0, 6
        jal ra, fact
        sw a0, xx
        halt
fact:
        addi sp, sp, -8
        sw   ra, 4(sp)
        sw   a0, 0(sp)
        addi t0, x0, 1
        bne  a0, t0, else
        addi a0, x0, 1
        addi sp, sp, 8
        jalr x0, 0(ra)
else:
        addi a0, a0, -1
        jal  ra, fact
        lw   t1, 0(sp)
```

```
lw   ra, 4(sp)
addi sp, sp, 8
mul  a0, a0, t1
jalr x0, 0(ra)
```

In addition to the factorial program, please validate your design using another meaningful piece of code from the your Assignment05 programs that include the direct use of a conditional branch (e.g., BNE or BEQ). You design for the full processor should be uploaded to Canvas by April 5.

Use part (c) as the reference for the following requirements of your final report:

Note: For your final working version of your design, please make sure to remove any debugging circuitry (e.g., for memory-content editor or signal tap) to reduce your footprint and improve your timing. You might also like to explore the Quartus tool settings, which can adjust the strength of circuit optimization and trade-off design area with timing.

1. Include the assembly and machine code of the factorial program and your program of choice in the documentation.

2. Your ModelSim testbenches and their outputs.

3. Print screenshots that show the memory contents after executing the factorial program and your program of choice.

4. Using the actual board and the factorial program, find the actual maximum frequency that your processor can sustain without producing incorrect results. You need to keep on incrementing the frequency with the PLL, and re-running your experiments. Once the processor fails to produce your correct result, report the highest frequency in which you had it working. This frequency will be the basis of the grading. Note that your critical path will likely involve the multiply instruction, so to crank your frequency up, you want to streamline the data path components involved in the multiply.

5. Repeat Bullet 4 but this time using factorial 12 instead of factorial 6. Report the new frequency. Why can't you clock the processor at the same high frequency in part 4?

6. Print and annotate the floorplan of your design. Report the resources being used by your processor: LEs (combinational and dedicated registers), PLL, embedded multipliers, memory blocks, and routing resources. Make sure to remove any SignalTap logic if you used it.

Guidelines for Adjusting frequency

You will need to adjust the frequency of operation either up or down. So far we used the default 50 MHz clock. To adjust the frequency, you will need to use a Phase Locked Loop (PLL) that is available in the IP Catalog MegaWizard. The PLL will take the 50 MHz clock input together with multiplier parameters and produces the target frequency clock, which you would then hook up to your design. You can also use the PLL to produce multiple clock signals with arbitrary phase shifts, which can be helpful if for your ROMs or RAMs.

Guidelines for Creation of Instruction and Data Memories:

Ideally the instruction memory should be built out of the ROM component. Unfortunately, the ROM component in the FPGA cannot be read combinationally; i.e., the output will not update its value until the positive edge of the clock comes in. Because the ROM address is supplied directly from the PC register, it will not be possible to update the PC and fetch the instruction in the same cycle. To avoid this problem, there are three possibilities:

- create a ROM directly using an array of 32-bit registers in your code and initialize the registers within your code using the `initial` statement; or
- use a ROM component and introduce a phase shift in the clock between the PC and the ROM. This shift should be ideally small; or
- Use the ROM component and use the register inside of it as your PC.

For the data memory, you should initialize the RAM blocks using the IP Catalog. Make your data memory size 1024 B or 256 words. Make sure the output port is not registered; however, there is no way to avoid that the inputs are not registered (same problem as in ROM). To fix the situation in this case, I suggest clocking the RAM with a phase shifted clock signal (>= 180 degrees). This will give the processor half a cycle (or more) to fetch and execute the instruction, and another half a cycle (or less) to access the memory and loading its contents into register. Using the PLL, you can tweak the clock settings to have an arbitrary phase shift (rather than just 180) for the RAM block. That can allow you to bump up your design frequency sometimes.

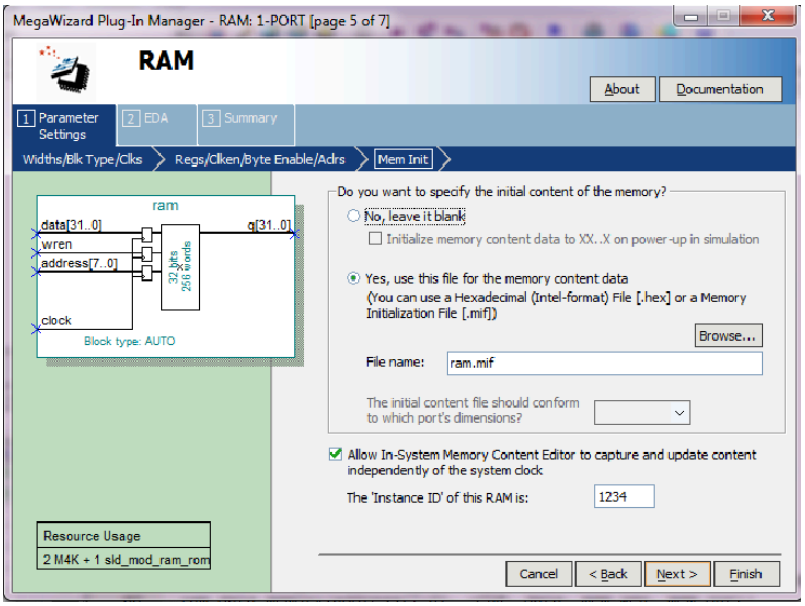
Guidelines for Debugging:

You are likely to encounter many bugs in your design. To help in debugging your code, you can use ModelSim. It is also helpful to learn how to use the Signal Tap tool. The Signal Tap tool inserts additional circuitries in your design to allow you monitor the activities of various wires during runtime through the Quartus tool. The tool is very powerful for debugging. There is a tutorial is also available on the class web page. The Signal Tap tool is very powerful and sufficient for debugging your code.

Guidelines for reading back RAM memory:

To test the correctness of your code, you will need to read back the contents of the RAM blocks after you are done with executing your code. The Quartus tool enables you to read back the contents of memory at any time after programming and during operation using the In-System Memory Content Editor tool. You need to carry out the following steps to enable such reading.

1. When you instantiate RAM, you will need to enable "Allow In-System Memory Content Editor to capture and update content independently of the system clock" as indicated in the next figure



2. After your programming your design, you need to launch the "In-System Memory Content Editor" which can be accessed from the Tools menu. Select the memory in Instance manager and click the read data from system memory button (one with a red box). You can now see the contents in the memory.