Brown University
School of Engineering
EN164 Design of Computing Systems
Professor Sherief Reda
LAB 05 (200 points)
Final report due on Friday April 25 2014 (Milestone on April 18)

In this lab you are required to substantially improve your original single-cycle processor design of lab04 by using pipelining. You will need to submit only one lab report on the due date of April 25th. However, you will need to demonstrate progress to the TAs on April 18th as outlined in this assignment.

Guidelines:

- Similar to the single-cycle processor, in order to verify that your processor works correctly, you need to store the execution results in the RAM data memory and then read the memory contents with the Quartus II tool after you run and are done with the program.
- **The design metric for this lab is execution runtime, which is equal to the number of actual clock cycles taken by the program multiplied by your clock period. Twenty points of this lab are allocated based on your processor runtime on the factorial program.**
- To count the number of clock cycles, you should keep a running count of the number of executed cycles in the $k0 register. That is, you need to increment this register every clock cycle, and store the value of this register in memory before you halt.
- It is probable that the EX more than 50% of the delay of the critical path in your single-cycle design. Since the frequency of your pipelined-processor is determined by the stage has the highest delay, the EX stage will naturally determine the maximum clock frequency of your pipeline processor. Thus, there will be perhaps no benefit of having separate MEM and WB stages, and it might be better to combine them into one stage. Thus, you might like to either design a two-stage pipeline processor (IF-ID-EX / MEM-WB) or a three-stage pipeline processor (IF-ID / EX / MEM-WB). Both designs are acceptable, and it is up to you to choose. The two-stage design will simplify hazard detection and forwarding. [If you want to do advanced stuff, you can pipeline your multiplier (check Megawizard) effectively splitting the bottleneck EX stage into two or more stages (e.g., EX1 and EX2).]
- You need to think of interesting ways to clock your data memory in your pipeline design to avoid unnecessary penalties.
- Please verify that your design works using the same program (factorial and your program of choice) as you did in lab 4.

Good luck!

(a) Due Date April 18th. Implement the pipelined processor assuming no hazards exist in code. To run the factorial program correctly, you need to insert **nop** manually in your code to eliminate hazards. Please see code below for a two-stage pipeline; you will have to modify for a three-stage design.

```
            addi $a0, $0, 6
            jal factorial
            sw $v0, 0($0)
            sw $k0, 4($0)
            halt
factorial:  addi $sp, $sp, -8
            nop
            sw   $a0, 4($sp)
            sw   $ra, 0($sp)
            addi $t0, $0, 2
            nop
            slt  $t0, $a0, $t0
            nop
            beq  $t0, $0, else
            addi $v0, $0, 1
            addi $sp, $sp, 8
            jr   $ra
else:       addi $a0, $a0, -1
            jal  factorial
            lw   $ra, 0($sp)
            lw   $a0, 4($sp)
            addi $sp, $sp, 8
            mul  $v0, $a0, $v0
            jr   $ra
```

(b) Due Date April 18th. In this case you need to modify your pipeline design so that it stalls whenever it detects a RAW hazard. Verify that you execute the same number of cycles as in part (a). Since you stall in hardware, you can remove the inserted **nop**s.

(c) Due Date April 25th. You need to demonstrate the final version of your design where it forwards operands whenever possible but stalls otherwise. Depending on your design, you may have to stall for the following RAW example to work correctly:

```
lw    $s1, 0(4)
addi  $s2, $s1, 4
```

You can certainly hide RAW compute-use hazards by forwarding

```
addi  $s1, $0, 10
addi  $s2, $s1, 4
```

With forwarding implemented, you can remove the nop instructions in the factorial code. What is the new number of cycles that the program takes to finish executing?

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

1. Include the assembly and machine code of the factorial program and your program of choice in the documentation. **Make sure that your program of choice can detect both RAW load-use and compute-use hazards.** Print screenshots that shows the memory contents after executing the factorial program and your program of choice. **Analyze your programs and prove that the reported numbers of cycles are correct.**

2. Use the TimingQuest tool to analyze the timing in your design. Report the critical path delay in your design and predicted Fmax. Use the tool to (1) locate the critical path in your floorplan and print the annotated critical in the floorplan view, and (2) estimate the delay breakdown among the different stages of your pipeline. **Explain how did you use the timing path information to optimize your design so that it can run at higher speed. Explore the possible optimization settings of the Quartus II tool and report the impact of relevant different settings on the timing and area of your design.**

3. Using the actual board, find the maximum frequency that your processor can sustain without producing incorrect results. You need to keep on incrementing the frequency of the design (by increasing the PLL multipliers – please read guidelines), and re-running your experiments. Once the processors fails in booting, report the lowest frequency in which such failure occurs. Contrast the actual maximum frequency to the estimated Fmax from the TimingQuest tool.

4. Explain your branch resolving strategy.

5. 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 additions.