CSCI 2021 Machine Architecture and Organization, Fall 2018, Written Assignment #3

Instructions:

- This assignment must be done individually.
- Posted Thursday November 8 and due on Friday November 16
- This assignment must be submitted as a PDF to Canvas by 11:55PM on the due date, there is not a late option.
- You may type your assignment or you may hand write your assignment and submit a scanned copy to Canvas. If you do not have access to a scanner, use an app such as CamScanner on your phone.
- Your assignment must be legible. If you turn in an assignment that we cannot clearly read, we are not obligated to grade it and can give it a 0. If you are concerned about the legibility of your handwriting, please type your assignment.
- Along with your name, include your student ID number, x500 (internet ID), and discussion section at the top of your assignment.
- There are **five** problems; we will go over them in lab after the due date.
- The textbook in this context is: R. Bryant, D. O'Hallaron, Computer Systems: A Programmer's Perspective (3rd Edition)

Problem 1 (20 points)

Design a logic circuit that finds the second largest value among the set of 3 words A, B, and C using an HCL case expression.

Problem 2 (20 points)

For each byte sequence listed, determine the Y86 instruction sequence it encodes. There is no invalid byte in any sequence. Your answer should be in the style that is used in the solution of Practice Problem 4.2 (page 360), meaning that you should include the instruction and the memory address of each instruction.

A. 0x300: 30F6EFFFFFFFFFFFFFF63671090
B. 0x400: A06F6277739448560100000000B07F
C. 0x500: 2006606680864856010000000

Problem 3 (20 points)

This question describes two new instructions we might consider adding to the Y86-64 processor. Like the textbook does in section 4.3 and we did in lecture for the existing Y86-64 instructions, give the actions that a SEQ-style Y86-64 implementation would take in each stage to execute the instruction. For each instruction, fill out the table below, with a row for Fetch, Decode, Execute, Memory, Write Back, and PC update (similar to Figure 4.18). Use the same notations as we did

in class, with signal names like valA, and R[...] and M[...] to represent access to the register file and memory respectively.

The **leave** instruction is used to clean up a stack frame when a frame pointer is in use (it's mentioned in section 3.10.5 of the textbook, p. 292). It first copies the frame pointer %rbp into the stack pointer %rsp and then it pops the top entry of the stack into %rbp. In other words, it is equivalent to the two-instruction sequence:

```
movq %rbp, %rsp
popq %rbp
```

The **indirect jump** instruction `jmp *D(rA)` jumps to a code address stored at the location `D + rA`, like the x86-64 instruction used to implement a jump table.

	Leave	**Indirect Jump**
Fetch		
Decode		
Execute		
Memory		
Write Back		
PC Update		

Problem 4 (20 points)

The listing below shows a sequence of Y86-64 instructions from a time-critical part of a program. There are five register data dependencies between instructions in this sequence. Fill in the blanks below with details about the 5 dependencies: give the number of which instruction (higher numbered) depends on which previous instruction (lower numbered) via which Y86-64 register (that they both access).

```
iaddq $8, %r8            # Instruction 1
mrmovq 8(%rax), %rcx     # Instruction 2
addq %rcx, %rbx          # Instruction 3
subq %r8, %rcx           # Instruction 4
```

```
rrmovq %rcx, %rbx         # Instruction 5
rmmovq %rbx, 24(%rax)     # Instruction 6
```

a. Instruction #_____ depends on instruction #_____ via register _____

b. Instruction #_____ depends on instruction #_____ via register _____

c. Instruction #_____ depends on instruction #_____ via register _____

d. Instruction #_____ depends on instruction #_____ via register _____

e. Instruction #_____ depends on instruction #_____ via register _____

f. Only one of these dependencies is a "load-use" hazard that will hurt the performance of the code when running on our pipelined Y86-64 implementation. Which one is that?

g. You can make the code run faster without changing its behavior by rearranging the instructions so that the load-use hazard does not require stalling. Show your improved program after rearranging the instructions.

Problem 5 (20 points)

This question is about control hazards, and how they are handled by the PIPE architecture presented on in Figure 4.52 (p. 440) of the textbook. As presented in the book's implementation, assume that the processor's branch predictor always takes conditional jumps. Draw the pipeline stages for each instruction of the following code. These diagrams should look like those presented in Figures 4.53 - 56 (pgs. 441-444).

```
0x100:     irmovq    $3, %rdx
0x10a:     irmovq    $2, %rcx
0x114:     subq      %rdx, %rcx
0x116:     je        target
0x11f:     mrmovq    200(%rdx), %rcx
0x129:     halt
0x12a: target:
0x12a:     mulq      %rcx, %rdx
```

```
0x12c:    addq      %rdx, %rdx
0x12e:    jge       target
0x137:    halt
```