CSCI 2021 Machine Architecture and Organization Fall 2017 Homework Assignment # 3

Instructions:

- This homework must be done individually.
- Posted Tuesday, October 31 and **due on moodle at 11:55 pm on Tuesday, November 14**
- Please **submit as a PDF** containing solutions, any other format will result in a deduction
- There are six problems; we will go over them in discussion section 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 (HCL - similar to Practice Problem 4.11 in the book) (10 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 (Data Dependences) (20 points)

The listing below shows a sequence of Y86-64 instructions from a time-critical part of a program. There are six register data dependencies between instructions in this sequence. Fill in the blanks on the right with details about the 6 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. Instruction #______ depends on instruction #______ via register __________

g. 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?

h. 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 3 (Pipelining) (30 points)

In this exercise, we examine how pipelining affects the clock cycle time of the processor. Assume that individual stages of the data path have the following latencies:

Fetch	Decode	Execute	Memory	Writeback
250 ps	500 ps	150 ps	250 ps	200 ps

Storing the results in an output register costs 20 ps. Also, assume when pipelining, each pipeline stage costs 20 ps extra for the registers between pipeline stages.

You are working with the following assembly code:

```
.L1
1     movq      $3, %rdx
2     leaq      -24(%rbp), %rbx
3     leaq      (%rbx,%rdx,4), %rbx
4     movq      (%rbx), %rdi
5     addq      %rdi, %rsi
6     movq      %rsi, %rax
7     cmpb      %al, $1
8     je        .L1
```

A. Suppose you have a *sequential* processor with the following rules:
   - Subsequent instructions are not fetched until the preceding ones have finished executing entirely
   - Registers do not separate the different stages in the datapath (i.e., data flows from one end to the other without being stored in a register to be synchronized with the clock edge)
   - Instructions are not required to undergo every stage (i.e., an instruction that sums values in two registers would use only the Fetch, Decode, Execute, and Writeback stages)

   i. **Given this setup, how many picoseconds would two iterations (i.e., conditional jump is taken) of the above code require?**

   ii. Recall that *latency* is the total time required to perform an instruction, and that *throughput* is the number of of instructions that can be executed per unit of time. **With these definitions, calculate the average latency and throughput of the above code for our sequential processor.**

B. Suppose you are now upgrading yourself to a pipelined processor similar to the one described in Figure 4.41 (p. 424) in the textbook. (You are still too plebian to use fancy techniques, such as branch prediction, memory prefetching, or data forwarding.)

   i. **Draw the pipeline stages for each instruction for the first two iterations (i.e., conditional jump is taken) in the following table.** The first two have been done for you. (Hint: You will need to insert “bubbles” to avoid certain hazards.)

Instruction	Pipeline stages												
1	F	D	E	M	W								
2		F	D	E	M	W							
3													
4													
5													
6													
7													
8													
1													
2													
3													
4													
5													
6													
7													
8													

   ii. Based on your work above, **how many picoseconds will this new processor require to execute the assembly code featured above?**

   iii. **Calculate the average latency and throughput of the above code for our basic pipelined processor.**

C. You have finally decided to treat yourself to a state-of-the-art processor that has 100% branch prediction (don’t ask how) and all the data forwarding paths described in Figure 4.2 (p. 440) in the textbook.

i. **Draw the stages for each instruction for the first two iterations (i.e., conditional jump is taken) in the following table.**

Instruction	Pipeline stages													
1														
2														
3														
4														
5														
6														
7														
8														
1														
2														
3														
4														
5														
6														
7														
8														

ii. Based on your work above, **how many picoseconds will this state-of-the-art processor require to execute the assembly code featured above?**

iii. **Calculate the average latency and throughput of the above code for this processor.**

D. Examine the performance calculations (execution time, latency, and throughput) you did for parts A, B, and C. **Summarize and *briefly* justify these results (i.e., the winner/loser for each category).**

Problem 4 (Disassembling Code) (15 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: 40120f0f00000000000063221000
B. 0x400: a05f204530f0feffffffffffffff6060b05f90
C. 0x500: 104065fcffffffffffffff10106067741905000000000000000000201200

Problem 5 (x86 to Y86 Conversion) (10 points)

Given the c program:

```
int sum(int x) {
    if (x == 0 || x ==1) {
        return 1;
    } else {
        return x + mult(x-1);
    }
}
```

The following x86-64 assembly code is generated:

```
sum:
     cmpl      $1, %rdi
     ja        .L8
     movl      $1, %eax
     ret
.L8:
     pushq     %rbx
     movl      %edi, %ebx
     leal      -1(%rdi), %edi
     call      sum
     addl      %ebx, %eax
     popq      %rbx
     ret
```

Convert this to Y86-64 assembly code that does the same thing. An example in the textbook can be found on page 365.

Problem 6 (Y86 Stages) (15 points)

This question describes a new instruction we might consider adding to the Y86-64 processor. Like the textbook does in the 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, you should make a table with 6 rows labeled Fetch, Decode, Execute, Memory, Writeback, 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 accesses to the register file and memory. 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:

1. `movq %rbp, %rsp`
2. `popq %rbp`

	Leave
Fetch	
Decode	
Execute	
Memory	
Write Back	
PC Update	