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

Moti Begna, begna002, sec: 007

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.

```
int secondLargest = [
    (A > B && A < C) || (A < B && A > C) : A;
    (B > A && B < C) || (B < A && B > C)  : B;
    1                                    : C;
];
```

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

```
0x300:   30F6EFFFFFFFFFFFFFF |  irmovq  $-33, %rsi
0x30a:   6367                |  xorq    %rsi, %rdi
0x30c:   10                  |  nop
0x30d:   90                  |  ret
```

B. 0x400: A06F6277739448560100000000B07F (NEEDS REVISING)

```
0x400:   A06F                |  pushq   %rsi
0x402:   6277                |  andq    %rdi, %rdi
0x404:   739448560100000000  |  je   loop
0x40d:   B07F                |  pop     %rdi
```

C. 0x500: 2006606680864856010000000 (NEEDS REVISING)

```
0x500:   2006                |  rrmovq  %rax, %rsi
0x502:   6066                |  addq    %rsi, %rsi
0x504:   808648560100000000  |  call proc
```

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	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] ValP ← [PC + 2]	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] ValP ← [PC + 2]
Decode	ValA ← R[rA] //%rbp ValB ← R[rB] //%rsp	ValA ← R[rA]
Execute	ValE ← ValA + 8	ValE ← ValA + D
Memory	ValM ← ValA	ValM ← ValA
Write Back	R[rB] ← ValE R[rA] ←ValM	R[rA] ← ValE
PC Update	PC ← ValP	PC ← ValP

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 #3 depends on instruction #2 via register %rcx

b. Instruction #4 depends on instruction #1 via register %r8

c. Instruction #4 depends on instruction #2 via register %rcx

d. Instruction #5 depends on instruction #4 via register %rcx

e. Instruction #6 depends on instruction #5 via register %rbx

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?

   Instruction 2 which depends on the register %rax

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.

//Switch original Instruction 1 and 2

```
mrmovq 8(%rax), %rcx       # Instruction 1
```

```
iaddq $8, %r8              # Instruction 2
addq %rcx, %rbx            # Instruction 3
subq %r8, %rcx             # Instruction 4
rrmovq %rcx, %rbx    # Instruction 5
rmmovq %rbx, 24(%rax)      # Instruction 6
```

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
```

Instruction	Stages															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0x100	F	D	E	M	W											
0x10a		F	D	E	M	W										
bubble					***E***	***M***	***W***									
0x114			F	D	D	E	M	W								
0x116				F	F	D	E	M	W							
0x12a						F	D									
bubble								**E**	**M**	**W**						
0x12c							F	D								

bubble									**D**	**E**	**M**	**W**				
0x12e								F	F	D	E	M	W			
0x137										F	D	E	M	W		
0x11f											F	D	E	M	W	
0x129												F	D	E	M	W