THE UNIVERSITY OF BRITISH COLUMBIA
Computer Science 313
Quiz 1
Practice Quiz 1

Warning this is a sample quiz and may include items not covered in this version of the course. Similarly, things covered in this version of the course may not be covered in this quiz.

A reference sheet consisting of register numbering, instruction to byte encoding and the CPU stage schematic diagrams will be provide. They will be in the same format as in the text.

Marks	
1.	/9
2.	/5
3.	/4
4.	/7
Total	/25

1) The questions in this section refer to the code below. In the left column is a small C function, and on the right is the assembler output produced by the gcc C compiler from that code. The standard X86-64 (IA64) type calling conventions are used. This means parameters are passed in registers.

```
1) long long  sum2(long long a[],
                   long long cnt)
2) {
3)   long long  i;
4)   long long  tot = 0;
5)   for (i = 0; i < cnt; i++) {
6)    if (a[i] > 0) {
7)      tot += a[i];
8)    }
9)   }
10)    return tot;
11)  }
```

Note: The instruction leaq has the following form:

leaq (%rA, %rB , val) %rC

%rc ← %rB * val + %rA

If val is omitted it is treated as 1.

Recall that move instructions that start with a c are conditional moves and that, in an assembler listing, numbers with a leading $ are in hex, otherwise they are in decimal. Also the test instruction is like a subtract except the result of the subtract is not stored back into a register.

```
1) .sum2:
2)   testq   %rsi, %rsi
3)   jle .L4
4)   leaq    (%rdi,%rsi,8), %rsi
5)   xorl    %eax, %eax
6) .L3:
7)   movq    (%rdi), %rdx
8)   leaq    (%rax,%rdx), %rcx
9)   testq   %rdx, %rdx
10)  cmovg   %rcx, %rax
11)  addq    $8, %rdi
12)  cmpq    %rsi, %rdi
13)  jne .L3
14)  ret
15)  .L4:
16)      xorl    %eax, %eax
17)      ret
```

9 a) In this code, parameters are passed in registers. What register or registers are used to pass parameters and what argument (i.e. the name of variable from the C code) is assigned to each register used for passing parameters? Clearly justify your choice and reference the line numbers in the listing.

%rsi is cnt and we can tell that because it is tested for 0 and if it is 0 the function returns 0, which is what the C code specifies. %rdi is a (i.e. the address of where the array is stored in memory we can tell this from line 7 where we must be retrieving the value of an array element. (1.5 mark for each register and reason for a total of 3)

b) In the assembly listing what is the purpose of line 7? Explain what it is doing at a high level and relate it back to the C code.

It is retrieving the value of a[i] and storing the value of a[i] in a register for subsequent usage.

(2 marks)

c) Suppose the program were stopped at line 12 of the assembler listing. Is it possible to determine the value of i at that point? (i.e. what register or memory location is i stored in or how could you compute the value of i based on the contents of memory locations or registers that are accessible at this point.) If it is possible to determine i explain how and if not explain why not

I is directly stored in any register or memory location so printing its value is tricky since i isn't stored in any register. Instead a register holds the address of element a[i] and compares it to the ending location as a termination condition as opposed to incrementing i and and checking it. Consequently to determine the current value of i we would have to know the array starting location. That was passwd in %rdi, but %rdi is not saved and its value is changed in line 11 so we have know way of knowing or determining this so we can't determine I.

4 marks.

5 **2)** In the following table, beside each Y86-64 assembly instruction, list, in hex, the byte sequence produced for that instruction. To aid marking break sequences of bytes up into groups of 4. For example 12345678 is to be written as 1234 5678. You will be penalized if you do not do this.

pushq %rbp	*a05f*
irmovq 1, %rax	*30f0 0100 0000 0000 0000*
subq %rax, %r9	*6109*
rmmovq %r9, 0x98(%rdi)	*4097 9800 0000 0000 0000*
popq %rbp	*b05f*
ret	*90*

4 **3)** There are some X86-64 assembler instructions that don't have a coresponding Y86-64 instruction so multiple &86-64 instructions have to be used. Below are a couple of such instructions. First describe in a high level what the instruction is doing and then translate those instructions into the corresponding Y86-64 instructions.

a) addq 8(%rbp), %rax

This instruction takes the value in rbp adds 8 to and, retrieves the data at that value and adds it to the contents of rax. In Y86 this would be

rmmovq 8(%rbp), %r10

addq %r10, %rax.

b) subq 8, %rbp

Subtracts 4=8 from the value in rbp.

irmovq 8, %rax

aubq %rax, %rbp

7 4) Your task is to implement a new Y86 instruction ***call rA***. This instruction calls the function whose address is stored in rA and pushes the return address onto the stack. At the high level we can describe what the instruction does as follows:

PC = PC + 2; %rsp = %rsp + 8; M_8[%rsp] = PC; PC = %rA;

The format of the instruction in memory is as follows:

8	9	rA	F		

Use the circuit diagram to determine the variables you will need to use. If needed **make sure to set dstM and dstE** in the proper stage. Use the notation R[x] for access to register x and M_b[a] for access to *b* bytes of memory starting at address ***a.*** You are to fill in the second column of the following table with the function for each of the indicated stages. If a stage doesn't need to do anything indicate that, *do not leave it blank*. You do not need to specify srcA and srcB. Keep in mind that since this is a new instruction slight modifications to the circuit diagram might be needed to route signals (i.e. variables) to where they are now needed.

Stage	Computation
Fetch	*icode:ifun ← M_1[PC]* *rA:rB ← M_1[PC + 1]* *valP ← PC + 2* *(1 mark for each stage decode gets 2)*
Decode	*valA = R[%rA]* *valB ← R[%rsp]* *dstE ←%rsp*
Execute	*valE ← valB + 8*
Memory	*M_8[valE] ← valP*
Write-back	*R[dstE] ← valE (or R[%rsp] - valE)*
PC-update	*PC ← valA*