Tutorial 5
CSE 112 Computer Organization

1. ISA description:

The instructions supported by an ISA are mentioned in the table below. The ISA has 32 general-purpose registers: x0 to x31. And, the register x0 is hard-wired to zero.

Name	Semantics	Syntax
Add	Performs reg1 = reg2 + reg3	add reg1, reg2, reg3
Sub	Performs reg1 = reg2 - reg3	sub reg1, reg2, reg3
Branch if equal	Branch or PC=PC+sext({imm[12:1],1'b0}) if sext(rs1) == sext(rs2)	beq reg1, reg2, imm[12:1]
Branch if not equal	Branch or PC=PC+sext({imm[12:1],1'b0}) if sext(rs1) != sext(rs2)	bne reg1, reg2, imm[12:1]
Add Immediate	Perform reg1 = reg2 + sext(imm[11:0])	addi reg1, reg2, imm[11:0]

Programing example: suppose there is a need to compute 1 + 2 using the ISA described above, then, this might be done as follows:

```
addi x1, x0, #1
addi x2, x0, #2
add x2, x2, x1
```

Using the ISA described above, write the assembly code for the following problems:

A. **Store 10 and 5 in registers x1 and x2. Check whether they are equal or not. If they are equal, then write 1 to register x3, else write 0 to register x3.**
B. **Store 10 and 5 in registers x1 and x2. Multiply them. Write the product to register x3.**
C. **Store X to register x1. Find the sum of all whole numbers till the value store in register x1. Store the sum in register x2. Given that X is a constant positive integer, X = 2, then x1 = 2, x2 = 1+2 = 3**

You may also use labels in your code to refer to instruction locations.

[Hint: First, try to write the code as an algorithm instead of directly writing in the assembly]

Ans:

Note to TAs: Solve this first question to explain to students how to write assembly code with a focus on how to deal with branches using labels. Explain briefly how to write assembly code for branching and loops using labels.

A.

```
x1 = 10;
x2 = 5;
x2 = x2-x1;
if(!x2)
{
        x3 = 1;
}
else
```

```
        {
                x3 = 0;
        }
        end
```

```
        addi x1, x0, #10
        addi x2, x0, #5
        sub x2, x2, x1
        beq x2, x0, EQUAL
        bne x2, x0, EXIT
EQUAL:          -------> label name
        addi x3, x0, #1
        beq x0, x0, #0
EXIT:           ------> label name
        addi x3, x0, #0
        beq x0, x0, #0
```

B.

```
        x1 = 10
        x2 = 5
        x4 = x2
        x5 = 1
        If (!x1 || !x2)  //a logical OR operation between the negation of r0 and the negation of r1. It evaluates to true if either
                         r0 or r1 is equal to zero.
                x3 =0
        Jump to end
         while(x4):
                x3 = x3 + x1
                x4 = x4 - x5
        end
```

```
        addi x1, x0, #10
        addi x2, x0, #5
        addi x4, x2, #0
        addi x5, x0, #1
        addi x3, x0, #0
        beq x1, x0, arg_zero
        beq x2, x0, arg_zero
        loop:
                add x3, x3, x1
                sub x4, x4, x5
                beq x4, x0, loop
                beq x0, x0, #0
arg_zero:
        beq x0, x0, #0
```

C.

```
        x1 = 2;
        x2 = 0;
        x3 = x1;
        x4 = 1;
        while (x3)
        {
                x2= x2+x3;
                x3= x3-x4;
        }
        end
```

```
        addi x1, x0, #2           // Given X=2
        addi x2, x0, #0
        addi x3, x1, #0
        addi x4, x0, #1
        beq x3, x0, loop_exit
loop:
        add x2, x2, x3
        sub x3, x3, x4
        bne x3, x0, loop
        beq x0, x0, #0
loop_exit:
        beq x0, x0, #0
```

2.

A. Trace the execution of the following assembly program, which has been written using the instructions described in Q1. Assume every instruction can be completed in one clock cycle each. Write down the state of <u>all the registers</u> being used *after* the execution of each instruction. Also, trace the current <u>program counter</u> accordingly. Answer by filling in the table [given at the end of the question]. If the program ends, fill all the cells in the next row with END [PC increases by 1].

```
0 addi x1,x0, #2
1 addi x2,x0, #0
2 addi x3,x0, #1
3 addi x4,x0, #1
4 beq x0, x1 loop_exit
5 loop: add x2, x3, x2
6 add x3, x4,x3
7 sub x1, x1, x3
8 bneq x0, x1 loop
```

```
9 loop_exit: beq x0, x0 0
```

B. Assuming all instructions are executed in 1 clock cycle and the frequency of the processor is 1GHz, what is the total execution time of the program?

C. Suppose add, sub instructions take 1 clock cycle; beq, bneq take 2 clock cycles, and addi takes 3 clock cycles to execute, and the processor frequency is increased to 2 GHz. What is the execution time now? Is it a gain or a loss in execution time when compared to part B?

Clock Cycle	PC	x1	x2	x3	x4
0					
1					
2					
...	...	...	...	...	...

Ans:

```
0 addi x1,x0, #2
1 addi x2,x0, #0
2 addi x3,x0, #1
3 addi x4,x0, #1
4 beq x0, x1 loop_exit
5 loop: add x2, x3, x2
6 add x3, x4,x3
7 sub x1, x1, x3
8 bneq x0, x1 loop
9 loop_exit: beq x0, x0 0
```

Solution:

A.

Clock Cycle	PC	x1	x2	x3	x4
0	0	2	-	-	-
1	1	2	0	-	-
2	2	2	0	1	-
3	3	2	0	1	1
4	4	2	0	1	1
5	5	2	1	1	1
6	6	2	1	2	1
7	7	1	1	2	1
8	8	1	1	2	1
9	5	1	3	2	1
10	6	1	3	3	1
11	7	0	3	3	1
12	8	0	3	3	1
13	END	END	END	END	END

B.

13 * 1ns = 13ns [f=1/T, so 1Ghz = 1ns]

C. 3+3+3+3+2+1+1+1+2+1+1+1+2=24 cycles

Execution time = 24 x 0.5 = 12 ns

Execution time is less compared to above (B).

3. Suppose we decide to come up with a new assembly syntax for the same ISA. In the new syntax, the order of operands is different. The change is only done to the assembly syntax. The ISA is the same, i.e., the machine code syntax is the same. An example of add instruction in both the syntaxes is given below:

Old syntax assembly: add r1,r2, r3

(semantically r1 = r2+r3)

Old syntax machine code:000 01 10 11

New syntax assembly: add r3, r2, r1

(semantically r1 = r2+r3)

New syntax machine code:000 01 10 11

A. Would such a change in assembly syntax require changing the processor hardware? Why, why not?

B. Would such a change in assembly syntax require a change in the assembler? Why, why not?

C. If a compiled program (machine code) is given to you, can you tell which syntax the programmer used to write the code? If yes, how? If not, why?

Ans:

A. No, it wouldn't. The hardware/ISA decides opcode format. The specific "flavour" of the syntax is not determined by the hardware/ISA. We can have multiple assembly syntaxes for the same ISA. Example: x86 ISA has 2 widely used syntaxes: the Intel syntax and AT&T syntax.

B. Yes, the assembler needs to be modified so that it can correctly parse the new syntax and generate the correct code.

C. No, we can't. Regardless of the syntax, the assembled code will be in the form of binary instructions, whose syntax is dictated by the ISA. No matter what syntax we use, we will get the same binary representation of the program.

Note to the TAs: The objective of this question is to make students realize that the ISA and the assembly syntax are two different things.