COSE222: Computer Architecture
Assignment #2

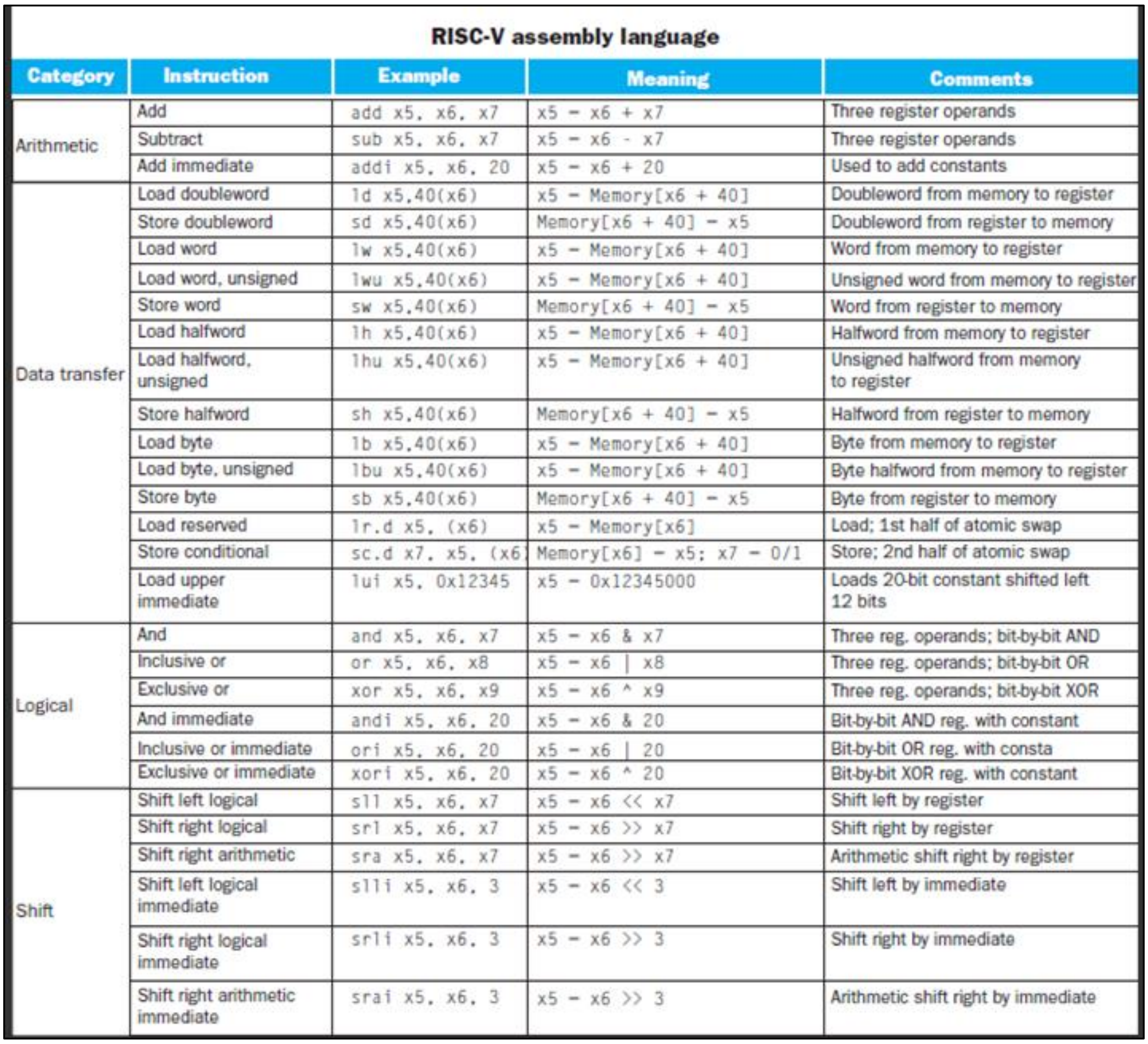
Due: Oct 18, 2020 (Sunday) 11:59pm on Blackboard

Total points: 183

Please answer for the questions. Write your student ID and name on the top of the document. Submit your homework with "**PDF**" format only. (You can easily generate the pdf files from Microsoft Word or HWP. You can also handwrite your answers to scan the handwritten documents with "**PDF**" format. You may use the document capture applications such as "Office Lens" for scanning your documents with your smartphones.)

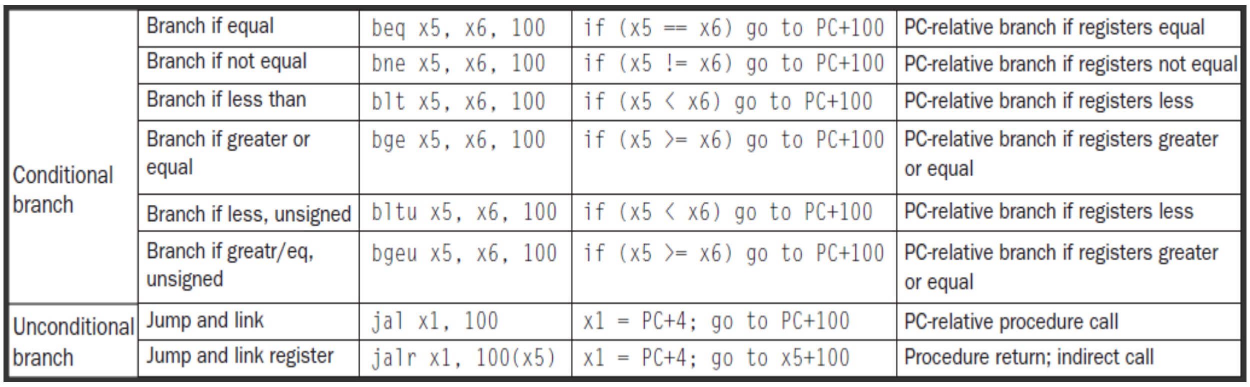
The answer rules:
(1) You can write answers in both Korean and English.
(2) Please make your final answer numbers have two decimal places.
(3) Performance of A is improved by *NN* % compared to performance B if PerfA / PerfB = 1.*NN*.

Please refer the following RISC-V assembly instructions for your answers.



RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load doubleword	`ld x5,40(x6)`	`x5 = Memory[x6 + 40]`	Doubleword from memory to register
	Store doubleword	`sd x5,40(x6)`	`Memory[x6 + 40] = x5`	Doubleword from register to memory
	Load word	`lw x5,40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5,40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5,40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5,40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5,40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5,40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5,40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5,40(x6)`	`x5 = Memory[x6 + 40]`	Byte halfword from memory to register
	Store byte	`sb x5,40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with consta
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate



Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greatr/eq, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

1. For the following C statement, write the corresponding RISC-V assembly code. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5`, `x6`, `x7`, `x28`, and `x29`, respectively. Assume that the base address of the array `A` and `B` are in registers `x10` and `x11`, respectively. The size of a single element of the array `A` and `B` is 8 bytes. (Use only a register `x30` for storing temporary values. You should minimize the lines of your code. Do not use `mul`) [10]

```
B[4] = A[4*i-j];
```

2. For the RISC-V assembly instructions below, what is the corresponding C statement? Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5`, `x6`, `x7`, `x28`, and `x29`, respectively. Assume that the base address of the array `A` and `B` are in registers `x10` and `x11`, respectively. The size of a single element of the array `A` and `B` is 8 bytes. [10]

```
slli  x30, x5, 3
add   x30, x10, x30
slli  x31, x6, 4
add   x31, x11, x31
ld    x5, 0(x30)
addi  x12, x30, 8
ld    x30, 0(x12)
sub   x30, x30, x5
sd    x30, 0(x31)
```

3. Answer for the following questions

(a) Translate the following C code to RISC-V assembly code straightforwardly. Assume that the variables `f` and `i` are assigned to registers `x5` and `x6`, respectively. Assume that the base address of the array `A` and `B` are in registers `x10` and `x11`, respectively. The size of a single element of the array `A` and `B` is 8 bytes. (Use x28, x29, x30, and x31 as registers for temporary data. You should minimize the lines of your code.) [15]

```
for (i = 9; i >= 0; i--) {
    f = A[i];
    A[i] = B[i];
    B[i] = f;
}
```

(b) The above C code shown in Questions 3-(a) can be rewritten using pointers `pA` and `pB` as follows. Translate the following C code to RISC-V assembly code. Assume that the variables `f` and `i` are assigned to registers `x5` and `x6`, respectively. Assume that the base address of the array `A` and `B` are in registers `x10` and `x11`, respectively. The size of a single element of the array `A` and `B` is 8 bytes. (Use x28, x29, x30, and x31 as registers for temporary data. You should minimize the lines of your code.) [15]

```
pB = &B[9];
for (pA = &A[9]; pA >= &A[0]; pA = pA - 1) {
    f = *pA
    *pA = *pB;
    *pB = f;
    pB = pB - 1;
}
```

(c) Assuming the CPI of every instruction is the same, compare the performance of the above two codes. [5]

4. Translate the following RISC-V code to C. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5`, `x6`, `x7`, `x28`, and `x29`, respectively. Assume that the base address of the array `A` and `B` are in registers `x10` and `x11`, respectively. The size of a single element of the array `A` and `B` is 8 bytes. [10]

```
addi  x30, x10, 16
addi  x31, x10, 8
ld    x5, 0(x30)
ld    x30, 8(x31)
add   x30, x30, x5
subi  x5, x30, 16
```

5. Implement the following C code using the minimum number of the **base integer** RISC-V instruction (RV64I) we learned. You cannot use multiplication and division instructions here. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers `x5`, `x6`, `x7`, `x28`, and `x29`, respectively. Use only a register `x30` for storing temporary values. [12]

(a) `g = -f;`

(b) `g = 7*f;`

(c) `g = 2*(f % 16);`

6. Assume the following register contents initially:

`x5 = 0x0000000000000000`, `x6 = 0x000000000013FF0F`

(a) Figure out the values in the registers `x5`, `x6`, and `x7` respectively when the following code completes its execution. Represent your answer in a hexadecimal format. [15]

```
LOOP:    beq     x6, x0, DONE
         srli    x7, x6, 1
         and     x6, x6, x7
         addi    x5, x5, 1
         jal     x0, LOOP
DONE:
```

(b) Figure out how many times the **beq** instruction is executed. [5]

7. Suppose the program counter (PC) is set to `0x30000000`.

(a) What range of addresses can be reached using the RISC-V *jump-and-link* (jal) instruction? (In other words, what is the set of possible values for the PC after jal executes?) [4]

(b) What range of addresses can be reached using the RISC-V *branch-if-equal* (beq) instruction? (In other words, what is the set of possible values for the PC after beq executes?) [4]

8. Consider the following RISC-V loop:

```
        addi    x5, x0, 0
        addi    x6, x0, 100
LOOP:   blt     x6, x0, DONE
        addi    x6, x6, -1
        addi    x5, x5, 4
        jal     x0, LOOP
DONE:
```

(a) What is the final value in register x5? Represent your answer in a hexadecimal format. [5]

(b) For the loop above, write the equivalent C code. Assume that the registers x5 and x6 are integers i and j, respectively. [10]

9. Let us assume you are to translate the following C code to RISC-V assembly code. Assume that the values of a, b, i, and j are in registers x5, x6, x7, and x29, respectively. Also, assume that register x10 holds the base address of the array D. You can use x30 and x31 as temporary registers. The size of a single element of array D is 8 bytes.

```
for (i=0; i < a; i++)
    for (j=0; j < b; j+=2)
        D[2*j] = i + j;
```

Complete the following RISC-V assembly code. [20]

```
          addi    x7, x0, 0
LOOPI:    bge
          addi
          addi    x29, x0, 0
LOOPJ:    bge
          add
          sd
          addi
          addi
          jal
ENDJ:     addi
          jal
ENDI:
```

10. Write the RISC-V assembly code that creates the 64-bit constant `0x1122334455667788` and stores that value to register `x10`. (*hint: you should use lui twice.*) [8]

11. Assume for a given processor the CPI of arithmetic instructions is 1, the CPI of load/store instructions is 10, and the CPI of branch instructions is 3. Assume a program has following instructions breakdown: 800 million arithmetic instructions, 300 million load/store instructions, 200 million branch instructions.

(a) Suppose that new, more powerful arithmetic instructions are added to the instruction set. On average, through the use of these more powerful arithmetic instructions, we can reduce the number of arithmetic instructions needed to execute a program by 25%, while increasing the clock cycle time by only 10%. Is this a good design choice? Why? [10]

(b) Suppose that we find a way to double the performance of arithmetic instructions. What is the overall speedup of our machine? What if we find a way to improve the performance of arithmetic instructions by 10 times? [10]

12. Let us assume that a given processor does not provide dedicated multiplier hardware. In this case we need to convert a multiplication into an equivalent function using other instructions. Assume you are requested to design a multiplier function for two 32-bit unsigned integer numbers. In the following C code, `A` and `B` are 32-bit unsigned integer types and operands of the multiplier function. Translate the following C code to the RISC-V assembly code. The multiplication result is held in `res`. Assume that `A`, `B`, `i`, and `res` holds the registers `x5`, `x6`, `x7`, and `x28` respectively. You can use `x29` and `x30` for temporary registers. [15]

```
int i;
int res = 0;
for (i = 0; i < 32; i++) {
    if (B & 0x01)
        res += (A << i);
    B = B >> 1;
}
```