6.004 Worksheet Questions
L02 – RISC-V Assembly

Computational Instructions

R-type: Register-register instructions: opcode = OP = 0110011

Arithmetic	Comparisons	Logical	Shifts
ADD, SUB	SLT, SLTU	AND, OR, XOR	SLL, SRL, SRA

Assembly instr: **oper rd, rs1, rs2**
Behavior: **reg[rd] <= reg[rs1] oper reg[rs2]**

- Rd = destination register (where result is saved)
- Rs1, rs2 = operand registers (their contents are used in the calculation)
- Example: "add x1, x2, x3" means "set x1 equal to x2 + x3"

SLT – Set less than
SLTU – Set less than unsigned
SLL – Shift left logical
SRL – Shift right logical
SRA – Shift right arithmetic

I-type: Register-immediate instructions: with opcode = OP-IMM = 0010011

Arithmetic	Comparisons	Logical	Shifts
ADDI	SLTI, SLTIU	ANDI, ORI, XORI	SLLI, SRLI, SRAI

Assembly instr: **oper rd, rs1, immI**

Behavior: **imm = signExtend(immI)** (sign extend to 32 bits)
reg[rd] <= reg[rs1] oper imm

- "Immediate" just means constant; immI is a 12-bit constant.
- Same functions as R-type except SUBI is not needed because immediate can be negative.
- Function is encoded in funct3 bits plus instr[30]. Instr[30] = 1 for SRAI. So SRLI and SRAI use same funct3 encoding.
- Example: "addi x1, x2, 0x4A7" means "set x1 equal to x2 + 1191"

U-type: opcode = LUI = 0110111

LUI – load upper immediate

Assembly instr: **lui rd, immU**

Behavior: **imm = {immU, 12'b0}** (immU concat. with 12 bits of 0)
Reg[rd] <= imm

- LUI is used to set a register equal to a number that is greater than 12 bits. A register can contain up to 32 bits, but "addi" only works with 12; LUI is used for the remaining 20 (32 – 12 = 20).
- immU = a 20 bit constant
- Example: "lui x2, 0x12345" would load register x2 with 0x12345000.
- In practice, it is simpler to use the pseudo-instruction "li" for loads of any size; see below for more details on pseudo-instructions.

Load Store Instructions

I-type: Load: with opcode = LOAD = 0000011

LW – load word

Assembly instr: **lw rd, immI(rs1)**

Behavior: **imm = signExtend(immI)** (to 32 bits)
Reg[rd] <= Mem[R[rs1] + imm]

- immI is a 12-bit constant known as the "offset;" this instruction will load the value located at an address given by the contents of rs1 + the offset. This is useful for accessing contiguous memory locations within the same program. The offset should be a multiple of 4 since a word (32 bits) in memory takes up 4 bytes and memory is byte-addressed.
- Example: If register x1 contains 0x1000, then "lw x2, 8(x1) will find the memory address 0x1008 and copy its contents into register x2.

S-type: Store: opcode = STORE = 0100011

SW – store word

Assembly instr: **sw rs2, immS(rs1)**

Behavior: **imm = signExtend(immS)**
Mem[R[rs1] + imm] <= R[rs2]

- immS is a 12-bit constant "offset" which works the same way as the offset described above for LW.
- Example: If register x3 contains 0x2000 and register x4 contains 0x3, the instruction "sw x4, 4(x3)" will store the value 0x3 into the memory location 0x2004.

Control Instructions

B-type: Conditional Branches: opcode = 1100011

Assembly instr: **oper rs1, rs2, label**

Behavior:
imm = distance to label in bytes =
signExtend({immB[12:1],0})
pc <= (R[rs1] comp R[rs2]) ? pc + imm : pc + 4

Compares register rs1 to rs2. If comparison is true, then the program counter (PC) jumps to the instruction following the label specified; in other words, PC is updated with PC + imm. If the comparison is false, PC becomes PC + 4, aka the next instruction (no jumping). Comparison type is defined by operation.

- BEQ – branch if equal (==)
- BNE – branch if not equal (!=)
- BLT – branch if less than (<)
- BGE – branch if greater than or equal (>=)
- BLTU – branch if less than using unsigned numbers (< unsigned)
- BGEU – branch if greater than or equal using unsigned numbers (>= unsigned)

J-type: Unconditional Jump: opcode = JAL = 1101111

Assembly instr: **JAL rd, label**

Behavior:
imm = distance to label in bytes =
signExtend({immJ{20:1},0})
Reg[rd] <= pc + 4; pc <= pc + imm

- JAL = "jump and link"
- Saves PC+4 (the return address) into rd
- Sets PC = PC + jump distance (to label specified)
- immJ is a 20 bit constant; therefore, the jump distance must be <= 20 bits, aka within 2^{18} instructions of the PC (because there are 4 addresses per instruction)
- Use it for functions: "jal ra, FuncName" (will be discussed in more detail later)

I-type: Unconditional Jump: opcode = JALR = 1100111

Assembly instr: **JALR rd, rs1, immI**

Behavior:
imm = signExtend(immI)
Reg[rd] <= pc + 4; pc <= (R[rs1]+imm) & ~0x00000001
(zero out the bottom bit of pc)

- JALR = "jump and link register"
- Writes PC+4 (return address) to rd and sets PC = rs1 + immI
- immI is a 12-bit constant

Common pseudoinstructions:

Jump:
j label = jal x0, label (ignore return address)

Load Immediate:
li x1, 0x1000 = lui x1, 1
li x1, 0x1100 = lui x1, 1; addi x1, x1, 0x100
li x4, 3 = addi x4, x0, 3

Move:
mv x3, x2 = addi x3, x2, 0

Branch if equal to zero:
beqz x1, target = beq x1, x0, target

Branch if not equal to zero:
bnez x1, target = bneq x1, x0, target

See the Reference Card for more.

Note: A small subset of essential problems are marked with a red star (★). We especially encourage you to try these out before recitation.

Problem 1

Compile the following expressions to RISC-V assembly. Assume a is stored at address 0x1000, b is stored at 0x1004, and c is stored at 0x1008. Assume that all values are 32-bit signed integers.

1. `a = b + 3*c;` ★

With a, b, and c being stored at addresses 0x1000, 0x1004, and 0x1008, each of these solutions are loosely structured in the following way:

1) Load a,b,c with LW
2) Perform operation
3) Store result with SW

Note that we do not have a multiplication instruction. We compute 3c with c << 1 + c. A left bit-shift by 1 (slli) is equivalent to multiplication by 2. Additionally, when loading, we use the offset field of the LW instruction to read the correct address. 8(x1) = 0x1000+8 = 0x1008, 4(x1) = 0x1004

```
// 1. Load values a,b,c
li x1, 0x1000     // actually lui x1, 1
lw x2, 8(x1)      // x2 = c, use offset to get 0x1008
lw x3, 4(x1)      // x3 = b, use offset to get 0x1004
// 2. Calculate a = b + 3c
slli x4, x2, 1    // x4 = x2 << 1 = 2c
add x4, x4, x2    // x4 = 2c + c = 3c
add x4, x4, x3    // x4 = 3c + b
// 3. Store value into a
sw x4, 0(x1)      // store x4 into a
```

2. `if (a > b) { c = 17; }` ★

We use branching to implement the IF statement, where the load for c=17 is skipped if the condition a > b is not satisfied.

```
li x1, 0x1000     // actually lui x1, 1
lw x2, 0(x1)      // x2 = a
lw x3, 4(x1)      // x3 = b
// branch to end if a <=b (or b >=a)
bge x3, x2, end
li x4, 17         // actually just addi x4, x0, 17
sw x4, 8(x1)      // c = 17
end:
```

3.
```
sum = 0;
for (i = 0; i < 10; i = i+1) { sum += i; }
```

Registers:

- x1: sum – cumulative sum
- x2: i – index
- x3: 10 – condition for FOR loop (i < 10).

We loop by checking for the condition (i < 10), and branching to the loop body beginning while the condition is met. There are no branch instructions that take an immediate, so we need to first store value 10 into a register, and then do a branch instruction comparing to the register.

```
addi x1, x0, 0    // x1 = 0 (sum)
addi x2, x0, 0    // x2 = 0 (i)
addi x3, x0, 10   // x3 = 10
loop:
add x1, x1, x2    // x1 = x1 + x2 or sum = sum + i
addi x2, x2, 1    // i = i+1
// if i < 10, branch to beginning of loop body
blt x2, x3, loop
```

Problem 2 ★

Compile the following expression assuming that a is stored at address 0x1100, and b is stored at 0x1200, and c is stored at 0x2000. Assume a, b, and c are arrays whose elements are stored in consecutive memory locations. Assume that all values are 32-bit signed integers.

```
for (i = 0; i < 10; i = i+1) { c[i] = a[i] + b[i]; }
```

Registers:

- x1: address of a[0]
- x2: address of c[0]
- x3: i – index
- x4: 4i – because of the length of a word, we multiply the i by 4 to get the right offset
  - RISC-V memory is indexed by byte and each word is four bytes long
- x5: address of a[i]
- x6: address of c[i]
- x7: 1) value of a[i], 2) a[i] + b[i]
- x8: value of b[i]
- x9: 10 – condition for FOR loop (i < 10)

The loop is implemented identically to above in Problem 1-3. We must first obtain the address given index i, which is 0x1100 + 4i for a[i], 0x1200 + 4i for b[i], and 0x2000 for c[i]

```
li x1, 0x1100     // x1 = address of a[0] (lui x1, 1; addi x1,
                     x1, 0x100)
li x2, 0x2000     // x2 = address of c[0] (lui x2, 2)
li x3, 0          // x3 = 0 (i)           (addi x3, x0, 0)
li x9, 10
loop:
slli x4, x3, 2    // x4 = 4 * i
add x5, x1, x4    // x5 = address of a[i]
add x6, x2, x4    // x6 = address of c[i]
lw x7, 0(x5)      // x7 = a[i]
lw x8, 0x100(x5)  // x8 = b[i]; b is offset from a by 0x100
add x7, x7, x8    // x7 = a[i] + b[i]
sw x7, 0(x6)      // c[i] = a[i] + b[i]
addi x3, x3, 1    // i = i + 1
blt x3, x9, loop  // branch back to loop if i < 10
```

Problem 3 ★

Hand assemble the following sequence of instructions into its equivalent binary encoding.
Hint: use the ISA Reference Card (posted under "Resources" on the website) to parse and encode the instruction.

```
addi x1, x1, -1
```

addi x1, x1, -1
-1 encoded as 12 bits is 0xfff
x1 in 5 bits is 0b00001
func3 for addi = 000
op = 0010011 (since addi is a register-immediate instruction)

addi: imm[11:0],rs1,func3,rd,op = 0xfff08093 =
0b111111111111_00001_000_00001_0010011

Problem 4

A) Assume that the registers are initialized to: x1=8, x2=10, x3=12, x4=0x1234, x5=24 before execution of each of the following assembly instructions. For each instruction, provide the value of the specified register or memory location. **If your answers are in hexadecimal, make sure to prepend them with the prefix 0x.**

1. `SLL x6, x4, x5` **Value of x6:** __0x34000000_____ ★

We shift left 0x1234 (x4) by 24 (x5) into x6:
0x1234 << 24 = 0x1234000000
However, since we are working in 32bits, we truncate correspondingly to get: 0x34000000

2. `ADD x7, x3, x2` **Value of x7:** ___22_____

We add 12 (x3) by x2 (10) into x7: 12 + 10 = 22

3. `ADDI x8, x1, 2` **Value of x8:** ___10_____

We add 8 (x1) by constant 2 into x8: 8 + 2 = 10

4. `SW x2, 4(x4)` **Value stored:** __10____ **at address:** ___0x1238_____ ★

x2 is the value we are writing into the address at x4 + 4
x2 = 10 (value stored)
x4 + 4 = 0x1234 + 4 = 0x1238

B) Assume X is at address 0x1CE8

```
        li x1, 0x1CE8
        lw x4, 0(x1)
        blt x4, x0, L1
        addi x2, x0, 17
        beq x0, x0, L2
L1:     srai x2, x4, 4
L2:

X:   .word 0x87654321
```

Value left in x4? 0x87654321_____

Value left in x2? 0xF8765432_____

Line by line decomposition:
1. x1 = 0x1CE8 – load value 0x1CE8 into x1
2. x4 = 0x87654321 – load word at address x1 + 0 = 0x1CE8 into x4
3. Branch into L1 – if (0x87654321 < 0), then jump to L1
4. x2 = 0xF876432 – 0x87654321 >> 4 into x2 (right shift arithmetic)

Problem 5

Compile the following Fibonacci implementation to RISC-V assembly.

```
# Reference Fibonacci implementation in Python
def fibonacci_iterative(n):
    if n == 0:
        return 0
    n = n - 1
    x, y = 0, 1
    while n > 0:
        # Parallel assignment of x and y
        # The new values for x and y are computed at the same time, and
        # then the values of x and y are updated afterwards
        x, y = y, x + y
        n = n - 1
    return y
```

Registers:

- x1: n
- x2: y (final result)
- x3: x
- x5: x + y

```
// x1 = n
// x2 = final result
bne x1, x0, start // branch if n!=0
li x2, 0
j end             // pseudo instruction for jal x0, end
start:
addi x1, x1, -1   // n = n - 1
li x3, 0          // x = 0
li x2, 1          // y = 1 (you're returning y at the end, so use
                      x2 to hold y)
loop:
bge x0, x1, end   // stop loop if 0 >= n
addi x5, x3, x2   // tmp = x + y
mv x3, x2         // x = y (pseudo instruction for addi x3, x2, 0)
mv x2, x5         // y = tmp (pseudo instruction for addi x2, x5, 0)
addi x1, x1, -1   // n = n - 1
j loop            // pseudo instruction for jal x0, loop
end:
```