**Computational Instructions**

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

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD, SUB</td>
<td>SLT, SLTU</td>
<td>AND, OR, XOR</td>
<td>SLL, SRL, SRA</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparisons</th>
<th>Logical</th>
<th>Shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td>SLTI, SLTIU</td>
<td>ANDI, ORI, XORI</td>
<td>SLLI, SRLI, SRAI</td>
</tr>
</tbody>
</table>

**Assembly instr:** oper rd, rs1, immI  
**Behavior:** imm = signExtend(immI)  
reg[rd] <= reg[rs1] oper imm

Same functions as R-type except SUBI is not needed.  
Function is encoded in funct3 bits plus instr[30]. Instr[30] = 1 for SRAI. So SRLI and SRAI use same funct3 encoding. immI is a 12 bit constant.

**U-type**: opcode = LUI or AUIPC = (01|00)10111

LUI – load upper immediate  
AUIPC – add upper immediate to PC
Assembly instr: \( \text{lui } \text{rd, immU} \)

Behavior: \( \text{imm} = \{\text{immU,12'b0}\} \)
\( \text{Reg[rd]} \leq \text{imm} \)

For example \( \text{lui } x2, 2 \) would load register \( x2 \) with 0x2000. \( \text{immU} \) is a 20 bit constant.

**Load Store Instructions**

**I-type: Load** with opcode = \( \text{LOAD} = 0000011 \)

LW – load word

Assembly instr: \( \text{lw } \text{rd, immI(rs1)} \)

Behavior: \( \text{imm} = \text{signExtend(immI)} \)
\( \text{Reg[rd]} \leq \text{Mem[R[rs1] + imm]} \)

**S-type: Store** opcode = \( \text{STORE} = 0100111 \)

SW – store word

Assembly instr: \( \text{sw } \text{rs2, immS(rs1)} \)

Behavior: \( \text{imm} = \text{signExtend(immS)} \)
\( \text{Mem[R[rs1] + imm]} \leq \text{R[rs2]} \)

\( \text{immS} \) is a 12 bit constant.

**Control Instructions**

**SB-type: Conditional Branches** opcode = \( 1100011 \)

Assembly instr: \( \text{oper } \text{rs1, rs2, label} \)

Behavior: \( \text{imm} = \text{distance to label in bytes} = \{\text{immS[12:1],0}\} \)
\( \text{pc} \leq (\text{R[rs1] comp R[rs2]}) ? \text{pc + imm : pc + 4} \)

Compares register \( \text{rs1} \) to \( \text{rs2} \). If comparison is true then \( \text{pc} \) is updated with \( \text{pc + imm} \), otherwise \( \text{pc} \) becomes \( \text{pc + 4} \). 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)
UJ-type: Unconditional Jumps: opcode = JAL = 110111

Assembly instr: JAL rd, label

Behavior: imm = distance to label in bytes = \{immU\{20:1\},0\}
          pc[rd] <= pc + 4; pc <= pc + imm

I-type: Unconditional Jump: opcode = JALR = 110011

Assembly instr: JALR rd, rs1, immI

Behavior: imm = signExtend(immI)
          pc[rd] <= pc + 4; pc <= (R[rs1]+imm) & ~0x01
          (zero out the bottom bit of pc)

JAL – jump and link
JALR – jump and link register

immJ is a 20 bit constant (used by JAL)
immI is a 12 bit constant (used by JALR)

Common pseudoinstructions:

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

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

mv x3, x2 = addi x3, x2, 0

beqz x1, target = beq x1, x0, target
bneqz x1, target = bneq x1, x0, target
Figure 2.2: RISC-V base instruction formats.

```
  31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  --------------------------
  funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
  --------------------------
  imm[11:0] | rs1 | funct3 | rd | opcode | I-type
  --------------------------
  --------------------------
  imm[31:12] | rd | opcode | U-type
```

Figure 2.3: RISC-V base instruction formats showing immediate variants.

```
  31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  --------------------------
  funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
  --------------------------
  imm[11:0] | rs1 | funct3 | rd | opcode | I-type
  --------------------------
  --------------------------
  --------------------------
```

Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

```
  31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  --------------------------
  --------------------------
  --------------------------
  --------------------------
  --------------------------
```

Table 9.1: RISC-V base opcode map, inst[1:0]=11

```
<table>
<thead>
<tr>
<th>inst[4:2]</th>
<th>inst[3:0]</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111 (&gt; 325)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LOAD</td>
<td>LOAD-FP</td>
<td>custom-0</td>
<td>MISC-MEM</td>
<td>OP-IMM</td>
<td>AUIPC</td>
<td>OP-IMM-32</td>
<td>489</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>STORE</td>
<td>STORE-FP</td>
<td>custom-0</td>
<td>AMO</td>
<td>OP</td>
<td>LUI</td>
<td>OP-32</td>
<td>648</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>MADD</td>
<td>MSUB</td>
<td>NMSUB</td>
<td>NMADD</td>
<td>OP-FP</td>
<td>reserved</td>
<td>custom-3/f=128</td>
<td>489</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>BRANCH</td>
<td>JALR</td>
<td>reserved</td>
<td>JAL</td>
<td>SYSTEM</td>
<td>reserved</td>
<td>custom-3/f=128</td>
<td>≥ 938</td>
<td></td>
</tr>
</tbody>
</table>
```
| imm[31:12] | rd  | 0110111  | LUI  |
| imm[20:11:19:12] | rd  | 1101111  | JAL  |
| imm[11:0] | rs1 | 000 | rd | 1100111  | JALR  |
| imm[11:0] | rs1 | 010 | rd | 0000011  | LW  |
| imm[11:0] | rs1 | 000 | rd | 0010011  | ADDI |
| imm[11:0] | rs1 | 010 | rd | 0010011  | SLTI |
| imm[11:0] | rs1 | 011 | rd | 0010011  | SLTIU |
| imm[11:0] | rs1 | 100 | rd | 0010011  | XOR |
| imm[11:0] | rs1 | 110 | rd | 0010011  | ORI |
| imm[11:0] | rs1 | 111 | rd | 0010011  | ANDI |
| 00000000 | shammt | rs1 | 001 | rd | 0010011  | SLLJ |
| 00000000 | shammt | rs1 | 101 | rd | 0010011  | SRJ |
| 01000000 | shammt | rs1 | 101 | rd | 0010011  | SRAI |
| 00000000 | rs2 | rs1 | 000 | rd | 0110011  | ADD |
| 01000000 | rs2 | rs1 | 000 | rd | 0110011  | SUB |
| 00000000 | rs2 | rs1 | 001 | rd | 0110011  | SLL |
| 00000000 | rs2 | rs1 | 010 | rd | 0110011  | SLT |
| 00000000 | rs2 | rs1 | 011 | rd | 0110011  | SJTU |
| 00000000 | rs2 | rs1 | 100 | rd | 0110011  | XOR |
| 00000000 | rs2 | rs1 | 101 | rd | 0110011  | SRL |
| 01000000 | rs2 | rs1 | 101 | rd | 0110011  | SRA |
| 00000000 | rs2 | rs1 | 110 | rd | 0110011  | OR |
| 00000000 | rs2 | rs1 | 111 | rd | 0110011  | AND |
Problem 1.

Compile the following expressions to RISCV assembly. Assume a is stored at address 0x1000, b is stored at 0x1004, and c is stored at 0x1008.

1. \( a = b + 3c; \)

```assembly
li x1, 0x1000  // actually lui x1, 1
lw x2, 8(x1)   // x2 = c
lw x3, 4(x1)   // x3 = b
slli x4, x2, 1 // x4 = x2 << 1 = 2c
add x4, x4, x2 // x4 = 2c + c = 3c
add x4, x4, x3 // x4 = 3c + b
sw x4, 0(x1)   // store x4 into a
```

2. \[ \text{if (a > b) c = 17;} \]

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

```assembly
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.

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

```
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:
sll 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]
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.

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_______

2. ADD x7, x3, x2
   Value of x7: ___22________

3. ADDI x8, x1, 2
   Value of x8: ___10________

4. SW x2, 4(x4)
   Value stored: __10_______ at address: ___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________________
Problem 4.

Compile the following Fibonacci implementation to RISCV assembly.

# Reference Fibonacci implementation in Python

def fibonacci_iterative(n):
    if n == 0:
        return 0
    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 -= 1
    return y

// x1 = n
// x2 = final result
bne x1, x0, start
li x2, 0
j end  // (pseudo instruction for jal x0, end)

code:

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

end: