### 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]`

- 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

<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)` (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:**

\[
\text{lui\ rd, immU}
\]

**Behavior:**

\[
\text{imm = } \{\text{immU, 12'b0}\}\ (\text{immU concat. with 12 bits of 0}) \\
\text{Reg[rd]} \leftarrow \text{imm}
\]

- 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** is 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:**

\[
\text{lw\ rd, immI(rs1)}
\]

**Behavior:**

\[
\text{imm = signExtend(immI) (to 32 bits)} \\
\text{Reg[rd]} \leftarrow \text{Mem[R[rs1] + imm]}
\]

- **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:**

\[
\text{sw\ rs2, immS(rs1)}
\]

**Behavior:**

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

- **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: \texttt{oper rs1, rs2, label}

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

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.

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

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

Assembly instr: \texttt{JAL rd, label}

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

- \texttt{JAL} = “jump and link”
- Saves PC+4 (the return address) into rd
- Sets PC = PC + jump distance (to label specified)
- \texttt{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: \texttt{JALR rd, rs1, immI}

Behavior:
\begin{itemize}
  \item \texttt{imm = signExtend(immI)}
  \item \texttt{Reg[rd] <= pc + 4; pc <= (R[rs1]+imm) \& \sim 0x00000001}
\end{itemize}

- \texttt{JALR} = “jump and link register”
- Writes PC+4 (return address) to rd and sets PC = rs1 + immI
- \texttt{immI} is a 12-bit constant
**Common pseudoinstructions:**

**Jump:**
\[ j \text{ label} = \text{jal} \ x0, \ \text{label} \] (ignore return address)

**Load Immediate:**
\[ \text{li} \ x1, 0x1000 = \text{lui} \ x1, 1 \]
\[ \text{li} \ x1, 0x1100 = \text{lui} \ x1, 1; \text{addi} \ x1, x1, 0x100 \]
\[ \text{li} \ x4, 3 = \text{addi} \ x4, x0, 3 \]

**Move:**
\[ \text{mv} \ x3, x2 = \text{addi} \ x3, x2, 0 \]

**Branch if equal to zero:**
\[ \text{beqz} \ x1, \text{target} = \text{beq} \ x1, x0, \text{target} \]

**Branch if not equal to zero:**
\[ \text{bnez} \ x1, \text{target} = \text{bneq} \ x1, x0, \text{target} \]

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. \[ a = b + 3c; \] ★

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

3. \[ \text{sum} = 0; \]
   \[ \text{for} \ (i = 0; i < 10; i = i+1) \ { \ 	ext{sum} += i; } \]
**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]; }
```
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.*

\texttt{addi \ x1, \ x1, \ -1}
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: ____________ ★

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

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

4. SW x2, 4(x4)  
   Value stored: _________ at address: __________ ★

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: Value left in x2? ____________
X: .word 0x87654321
```

Value left in x4? ____________

Value left in x2? ____________
Problem 5.

Compile the following Fibonacci implementation to RISC-V assembly.

```python
# 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
```