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

**Behavior:**
\[
\text{imm} = \{\text{immU, 12'b0}\}
\]
\[
\text{Reg[rd]} <= \text{imm}
\]

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 = LOAD = 0000011

**LW – load word**

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

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

**S-type: Store:** opcode = STORE = 0100011

**SW – store word**

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

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

immS is a 12 bit constant.

**Control Instructions**

**SB-type: Conditional Branches:** opcode = 1100011

**Assembly instr:**
\[\text{oper } \text{rs1}, \text{rs2}, \text{label}\]

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

Compares register rs1 to rs2. If comparison is true then pc is updated with pc + imm, otherwise pc becomes 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 = 1100111

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
## MIT 6.004 ISA Reference Card: Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JALR</strong></td>
<td>jalr rd, rs1, imm8</td>
<td>Jump and Link Register</td>
<td>reg[rd] := pc + 4</td>
</tr>
<tr>
<td><strong>BEQ</strong></td>
<td>beq rs1, rs2, imm8</td>
<td>Branch if =</td>
<td>reg[rd] := pc + 4</td>
</tr>
<tr>
<td><strong>BNE</strong></td>
<td>bne rs1, rs2, imm8</td>
<td>Branch if ≠</td>
<td>reg[rd] := pc + 4</td>
</tr>
<tr>
<td><strong>BLT</strong></td>
<td>blt rs1, rs2, imm8</td>
<td>Branch if &lt; (Signed)</td>
<td>reg[rd] := pc + 4</td>
</tr>
<tr>
<td><strong>BLE</strong></td>
<td>ble rs1, rs2, imm8</td>
<td>Branch if ≤ (Signed)</td>
<td>reg[rd] := pc + 4</td>
</tr>
<tr>
<td><strong>BLTU</strong></td>
<td>bltu rs1, rs2, imm8</td>
<td>Branch if &lt; (Unsigned)</td>
<td>reg[rd] := pc + 4</td>
</tr>
<tr>
<td><strong>BLEU</strong></td>
<td>bleu rs1, rs2, imm8</td>
<td>Branch if ≤ (Unsigned)</td>
<td>reg[rd] := pc + 4</td>
</tr>
<tr>
<td><strong>LW</strong></td>
<td>lw rd, imm(rs1)</td>
<td>Load Word</td>
<td>reg[rd] := mem[reg[rs1] + imm]</td>
</tr>
<tr>
<td><strong>SW</strong></td>
<td>sw rs2, imm(rs1)</td>
<td>Store Word</td>
<td>mem[reg[rs1] + imm] := reg[rs2]</td>
</tr>
<tr>
<td><strong>ADDI</strong></td>
<td>addi rd, rs1, imm8</td>
<td>Add Immediate</td>
<td>reg[rd] := reg[rs1] + imm8</td>
</tr>
<tr>
<td><strong>SLT</strong></td>
<td>slt rs1, rs2, imm1</td>
<td>Compare &lt; Immediate</td>
<td>reg[rd] := reg[rs1] &lt; reg[rs2] ? 1 : 0</td>
</tr>
<tr>
<td><strong>SLTU</strong></td>
<td>sltu rs1, rs2, imm1</td>
<td>Compare &lt; Immediate (Unsigned)</td>
<td>reg[rd] := reg[rs1] &lt; reg[rs2] ? 1 : 0</td>
</tr>
<tr>
<td><strong>XOR</strong></td>
<td>xor rs1, rs2, imm1</td>
<td>XOR Immediate</td>
<td>reg[rd] := reg[rs1] ^ reg[rs2]</td>
</tr>
<tr>
<td><strong>AND</strong></td>
<td>and rs1, rs2, imm1</td>
<td>And Immediate</td>
<td>reg[rd] := reg[rs1] &amp; reg[rs2]</td>
</tr>
<tr>
<td><strong>SLL</strong></td>
<td>sll rs1, rs2, imm1</td>
<td>Shift Left Logical Immediate</td>
<td>reg[rd] := reg[rs1] &lt;&lt; imm1</td>
</tr>
<tr>
<td><strong>SRL</strong></td>
<td>srl rs1, rs2, imm1</td>
<td>Shift Right Logical Immediate</td>
<td>reg[rd] := reg[rs1] &gt;&gt; imm1</td>
</tr>
<tr>
<td><strong>SRA</strong></td>
<td>sra rs1, rs2, imm1</td>
<td>Shift Right Arithmetic Immediate</td>
<td>reg[rd] := reg[rs1] &gt;&gt; imm1</td>
</tr>
<tr>
<td><strong>SUB</strong></td>
<td>sub rs1, rs2, imm1</td>
<td>Subtract</td>
<td>reg[rd] := reg[rs1] - reg[rs2]</td>
</tr>
<tr>
<td><strong>SHL</strong></td>
<td>shll rs1, rs2, imm1</td>
<td>Shift Left Logical</td>
<td>reg[rd] := reg[rs1] &lt;&lt; imm1</td>
</tr>
<tr>
<td><strong>SLTU</strong></td>
<td>sltui rs1, rs2, imm1</td>
<td>Compare &lt; (Signed)</td>
<td>reg[rd] := reg[rs1] &lt; reg[rs2] ? 1 : 0</td>
</tr>
<tr>
<td><strong>XORI</strong></td>
<td>xor rs1, rs2, imm1</td>
<td>XOR</td>
<td>reg[rd] := reg[rs1] ^ reg[rs2]</td>
</tr>
<tr>
<td><strong>ANDI</strong></td>
<td>and rs1, rs2, imm1</td>
<td>And</td>
<td>reg[rd] := reg[rs1] &amp; reg[rs2]</td>
</tr>
<tr>
<td><strong>SLLI</strong></td>
<td>slli rs1, rs2, imm1</td>
<td>Shift Left Logical</td>
<td>reg[rd] := reg[rs1] &lt;&lt; imm1</td>
</tr>
<tr>
<td><strong>SRLI</strong></td>
<td>sral rs1, rs2, imm1</td>
<td>Shift Right Logical</td>
<td>reg[rd] := reg[rs1] &gt;&gt; imm1</td>
</tr>
<tr>
<td><strong>SRAI</strong></td>
<td>srai rs1, rs2, imm1</td>
<td>Shift Right Arithmetic</td>
<td>reg[rd] := reg[rs1] &gt;&gt; imm1</td>
</tr>
<tr>
<td><strong>SUBI</strong></td>
<td>subi rs1, rs2, imm1</td>
<td>Subtract</td>
<td>reg[rd] := reg[rs1] - reg[rs2]</td>
</tr>
<tr>
<td><strong>J</strong></td>
<td>jal ra</td>
<td>Jump</td>
<td>pc := reg[ra]</td>
</tr>
<tr>
<td><strong>JAL</strong></td>
<td>jalr ra</td>
<td>Jump and Link (with ra)</td>
<td>reg[ra] := pc + 4</td>
</tr>
<tr>
<td><strong>JR</strong></td>
<td>jr ra</td>
<td>Jump Register</td>
<td>pc := reg[ra] &amp; 1</td>
</tr>
<tr>
<td><strong>JALR</strong></td>
<td>jalr ra</td>
<td>Jump and Link Register (with ra)</td>
<td>reg[ra] := pc + 4</td>
</tr>
<tr>
<td><strong>BHI</strong></td>
<td>bhi rs1, ra, label</td>
<td>Branch &gt; (Signed)</td>
<td>pc := (reg[rs1] &gt; reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>BLT</strong></td>
<td>blt rs1, ra, label</td>
<td>Branch &lt; (Signed)</td>
<td>pc := (reg[rs1] &lt; reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>BLE</strong></td>
<td>ble rs1, ra, label</td>
<td>Branch ≤ (Signed)</td>
<td>pc := (reg[rs1] ≤ reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>BNE</strong></td>
<td>bne rs1, ra, label</td>
<td>Branch ≠ (Signed)</td>
<td>pc := (reg[rs1] ≠ reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>BEQ</strong></td>
<td>beq rs1, ra, label</td>
<td>Branch = (Signed)</td>
<td>pc := (reg[rs1] = reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>BNE</strong></td>
<td>bne rs1, ra, label</td>
<td>Branch ≠ (Signed)</td>
<td>pc := (reg[rs1] ≠ reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>JAL</strong></td>
<td>jalr ra</td>
<td>Jump and Link Register (with ra)</td>
<td>reg[ra] := pc + 4</td>
</tr>
<tr>
<td><strong>JAL</strong></td>
<td>jalr ra</td>
<td>Jump and Link Register (with ra)</td>
<td>reg[ra] := pc + 4</td>
</tr>
<tr>
<td><strong>BHI</strong></td>
<td>bhi rs1, ra, label</td>
<td>Branch &gt; (Signed)</td>
<td>pc := (reg[rs1] &gt; reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>BLT</strong></td>
<td>blt rs1, ra, label</td>
<td>Branch &lt; (Signed)</td>
<td>pc := (reg[rs1] &lt; reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>BLE</strong></td>
<td>ble rs1, ra, label</td>
<td>Branch ≤ (Signed)</td>
<td>pc := (reg[rs1] ≤ reg[ra]) ? label : pc + 4</td>
</tr>
</tbody>
</table>

**NOTE:** All immediate values (immU, immI, immL, immB, and immS) are sign-extended to 32-bits.

## MIT 6.004 ISA Reference Card: Pseudoinstructions

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>li</strong></td>
<td>Load Immediate</td>
<td>reg[rd] := constant</td>
</tr>
<tr>
<td><strong>mv</strong></td>
<td>Move</td>
<td>reg[rd] := reg[rs1] + 0</td>
</tr>
<tr>
<td><strong>not</strong></td>
<td>Logical Not</td>
<td>reg[rd] := reg[rs1]’ - 1</td>
</tr>
<tr>
<td><strong>neg</strong></td>
<td>Arithmetic Negation</td>
<td>reg[rd] := reg[rs1] - reg[rs1]</td>
</tr>
<tr>
<td><strong>j</strong></td>
<td>Jump</td>
<td>pc := label</td>
</tr>
<tr>
<td><strong>jal</strong></td>
<td>Jump and Link (with ra)</td>
<td>reg[ra] := pc + 4</td>
</tr>
<tr>
<td><strong>jr</strong></td>
<td>Jump Register</td>
<td>pc := reg[ra] &amp; 1</td>
</tr>
<tr>
<td><strong>jalr</strong></td>
<td>Jump and Link Register (with ra)</td>
<td>reg[ra] := pc + 4</td>
</tr>
<tr>
<td><strong>ret</strong></td>
<td>Return from Subroutine</td>
<td>pc := reg[rs1]</td>
</tr>
<tr>
<td><strong>bgt</strong></td>
<td>Branch &gt; ( Signed)</td>
<td>pc := (reg[rs1] &gt; reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>ble</strong></td>
<td>Branch &lt; ( Signed)</td>
<td>pc := (reg[rs1] &lt; reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>bge</strong></td>
<td>Branch ≥ ( Signed)</td>
<td>pc := (reg[rs1] ≥ reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>bne</strong></td>
<td>Branch ≠ ( Signed)</td>
<td>pc := (reg[rs1] ≠ reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>beq</strong></td>
<td>Branch = ( Signed)</td>
<td>pc := (reg[rs1] = reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>bgt</strong></td>
<td>Branch &gt; ( Signed)</td>
<td>pc := (reg[rs1] &gt; reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>ble</strong></td>
<td>Branch &lt; ( Signed)</td>
<td>pc := (reg[rs1] &lt; reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>bge</strong></td>
<td>Branch ≥ ( Signed)</td>
<td>pc := (reg[rs1] ≥ reg[ra]) ? label : pc + 4</td>
</tr>
<tr>
<td><strong>bne</strong></td>
<td>Branch ≠ ( Signed)</td>
<td>pc := (reg[rs1] ≠ reg[ra]) ? label : pc + 4</td>
</tr>
</tbody>
</table>

1

**6.004 Worksheet** - 4 of 11 - L02 – RISC-V Assembly
### MIT 6.004 ISA Reference Card: Calling Convention

<table>
<thead>
<tr>
<th>Registers</th>
<th>Symbolic names</th>
<th>Description</th>
<th>Saver</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Hardwired zero</td>
<td>—</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return address</td>
<td>Caller</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global pointer</td>
<td>—</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread pointer</td>
<td>—</td>
</tr>
<tr>
<td>x5-x7</td>
<td>t0-t2</td>
<td>Temporary registers</td>
<td>Caller</td>
</tr>
<tr>
<td>x8-x9</td>
<td>s0-s1</td>
<td>Saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>x10-x11</td>
<td>a0-a1</td>
<td>Function arguments and return values</td>
<td>Caller</td>
</tr>
<tr>
<td>x12-x17</td>
<td>a2-a7</td>
<td>Function arguments</td>
<td>Caller</td>
</tr>
<tr>
<td>x18-x27</td>
<td>s2-s11</td>
<td>Saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>x28-x31</td>
<td>t3-t6</td>
<td>Temporary registers</td>
<td>Caller</td>
</tr>
</tbody>
</table>

### MIT 6.004 ISA Reference Card: Instruction Encodings

<table>
<thead>
<tr>
<th>funct7</th>
<th>imm[11:0]</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
<th>B-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opode</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>imm[11:5]</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>imm(4)</td>
<td>opode</td>
<td>S-type</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RV32I Base Instruction Set (MIT 6.004 subset)</th>
<th>rd</th>
<th>0100111</th>
<th>LUI</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>1101111</td>
<td>JAL</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
</tr>
<tr>
<td>0000000</td>
<td>shamt</td>
<td>rs1</td>
<td>001</td>
</tr>
<tr>
<td>0000000</td>
<td>shamt</td>
<td>rs1</td>
<td>001</td>
</tr>
<tr>
<td>0100000</td>
<td>shamt</td>
<td>rs1</td>
<td>001</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
</tr>
</tbody>
</table>
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.

1. \[ a = b + 3c; \] ⭐

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 ∙ 2. 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

```riscv
// 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.

```riscv
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, 0(x1)      // c = 17
end:
```

6.004 Worksheet - 6 of 11 - L02 – RISC-V Assembly
3. \[
\text{sum} = 0; \\
\text{for (i = 0; i < 10; i = i+1) sum += i;}
\]

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.

\[
\begin{align*}
\text{addi} & \ x1, x0, 0 \quad // x1 = 0 \text{ (sum)} \\
\text{addi} & \ x2, x0, 0 \quad // x2 = 0 \text{ (i)} \\
\text{addi} & \ x3, x0, 10 \quad // x3 = 10 \\
\text{loop:} & \\
\text{add} & \ x1, x1, x2 \quad // x1 = x1 + x2 \text{ or sum = sum + i} \\
\text{addi} & \ x2, x2, 1 \quad // i = i+1 \\
\text{blt} & \ x2, x3, \text{loop}
\end{align*}
\]
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.

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:
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]; 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.

loop:
addi x1, x1, -1
bnez x1, loop

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

bnez x1, loop = bne x1, x0, loop
x1 in 5 bits 0b00001 = rs1
x0 in 5 bits is 0b00000 = rs2
func3 for bne = 001
op = 1100011

We store the offset to the label, which is -4 (0b100), into the immediate value. Since the least significant bit (bottom bit) is always 0 with the offset, we can store bits 12:1 of the immediate value into the instruction. Using bits 12:1 doubles the max offset of branches as compared to 11:1.

imm[12:1] = distance to label in bytes / 2 = -2 = 0xffff
imm[12] = 1
imm[10:5] = 0b11111
imm[4:1] = 0b1110

bnez: imm[12],imm[10:5],rs2,rs1,func3,imm[4:1],imm[11],op = 0xfe009ee3 =
0b1_111111_00000_00001_001_1110_1_1100011
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? 0x_87654321
   Value left in x2? 0x_F876432

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 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

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: