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: \( \text{oper rd, rs1, rs2} \)

Behavior: \( \text{reg[rd]} \leq \text{reg[rs1]} \text{ oper reg[rs2]} \)

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: \( \text{oper rd, rs1, immI} \)

Behavior: \( \text{imm} = \text{signExtend(immI)} \)
\[ \text{reg[rd]} \leq \text{reg[rs1]} \text{ oper imm} \]

Same functions as R-type except SUBI is not needed.
Function is encoded in funct3 bits plus instr[30]. Inst[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:  lui rd, immU

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

For example lui x2, 2 would load register x2 with 0x2000.  
immU is a 20 bit constant.

Load Store Instructions

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

LW – load word

Assembly instr:  lw rd, immI(rs1)

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

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.

Control Instructions

SB-type: Conditional Branches: opcode = 1100011

Assembly instr:  oper rs1, rs2, label

Behavior:  
imm = distance to label in bytes = \{immS[12:1],0\}  
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 Jump: opcode = JAL = 110111

Assembly instr: \[ \text{JAL } \text{rd, label} \]

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

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

Assembly instr: \[ \text{JALR } \text{rd, rs1, immI} \]

Behavior: \[ \text{imm = signExtend(immI)} \]
\[ \text{pc[rd]} <= \text{pc} + 4; \text{pc} <= (\text{R[rs1]}+\text{imm}) \& \sim\text{0x01} \]
(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:

\[ j \text{ label = jal } x0, \text{ label} \] (ignore return address)
\[ \text{li } x1, 0x1000 = \text{lui } x1, 1 \]
\[ \text{li } x1, 0x1100 = \text{lui } x1, 1; \text{addi } x1, x1, 0x100 \]
\[ \text{li } x4, 3 = \text{addi } x4, x0, 3 \]
\[ \text{mv } x3, x2 = \text{addi } x3, x2, 0 \]
\[ \text{beqz } x1, \text{ target = beq } x1, x0, \text{ target} \]
\[ \text{bneqz } x1, \text{ target = bneq } x1, x0, \text{ target} \]
Figure 2.2: RISC-V base instruction formats.

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

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].

Table 9.1: RISC-V base opcode map, inst[1:0] = 11
<table>
<thead>
<tr>
<th>RV32I Base Instruction Set (MIT 6.5084 subset)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>imm[31:12]</strong></td>
</tr>
<tr>
<td><strong>imm[20:10:11:9:12]</strong></td>
</tr>
<tr>
<td><strong>imm[11:0]</strong></td>
</tr>
<tr>
<td><strong>imm[12:10:5]</strong></td>
</tr>
<tr>
<td><strong>imm[12:10:5]</strong></td>
</tr>
<tr>
<td><strong>imm[12:10:5]</strong></td>
</tr>
<tr>
<td><strong>imm[12:10:5]</strong></td>
</tr>
<tr>
<td><strong>imm[12:10:5]</strong></td>
</tr>
<tr>
<td><strong>imm[12:10:5]</strong></td>
</tr>
<tr>
<td><strong>imm[11:0]</strong></td>
</tr>
<tr>
<td><strong>imm[11:5]</strong></td>
</tr>
<tr>
<td><strong>imm[11:0]</strong></td>
</tr>
<tr>
<td><strong>imm[11:0]</strong></td>
</tr>
<tr>
<td><strong>imm[11:0]</strong></td>
</tr>
<tr>
<td><strong>imm[11:0]</strong></td>
</tr>
<tr>
<td><strong>imm[11:0]</strong></td>
</tr>
<tr>
<td><strong>imm[11:0]</strong></td>
</tr>
<tr>
<td>0000000</td>
</tr>
<tr>
<td>00000000</td>
</tr>
<tr>
<td>0100000</td>
</tr>
<tr>
<td>0000000</td>
</tr>
<tr>
<td>0100000</td>
</tr>
<tr>
<td>0000000</td>
</tr>
<tr>
<td>0000000</td>
</tr>
<tr>
<td>0000000</td>
</tr>
<tr>
<td>0000000</td>
</tr>
<tr>
<td>0000000</td>
</tr>
<tr>
<td>0000000</td>
</tr>
</tbody>
</table>
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. \[ a = b + 3c; \]

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

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

for (i = 0; i < 10; i = i+1) c[i] = a[i] + b[i];
Problem 3.

A) Assume that the registers are initialized to: $x_1=8$, $x_2=10$, $x_3=12$, $x_4=0x1234$, $x_5=24$ 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 $x_6$, $x_4$, $x_5$  
   Value of $x_6$: __________

2. ADD $x_7$, $x_3$, $x_2$  
   Value of $x_7$: __________

3. ADDI $x_8$, $x_1$, 2  
   Value of $x_8$: __________

4. SW $x_2$, 4($x_4$)  
   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:  
X: .word 0x87654321
```

Value left in $x4$? __________

Value left in $x2$? __________
Problem 4.

Compile the following Fibonacci implementation to RISCV assembly.

```python
# 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
```