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, XOR</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: \texttt{lui rd, immU}

Behavior: \begin{align*}
\text{imm} &= \{\text{immU,12'b0}\} \\
\text{Reg}[rd] &= \text{imm}
\end{align*}

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

\textbf{Load Store Instructions}

**I-type: Load** with opcode = \texttt{LOAD} = \texttt{0000111}

\texttt{LW} – load word

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

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

**S-type: Store** opcode = \texttt{STORE} = \texttt{0100111}

\texttt{SW} – store word

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

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

\texttt{immS} is a 12 bit constant.

\textbf{Control Instructions}

**B-type: Conditional Branches** opcode = \texttt{1100011}

Assembly instr: \texttt{oper rs1, rs2, label}

Behavior: \begin{align*}
\text{imm} &= \text{distance to label in bytes = signExtend}\{\text{immB[12:1]},0\} \\
\text{pc} &= \text{pc} + \text{imm} \quad \text{if } \text{comp} \text{ } R[rs2] \quad \text{or otherwise } \text{pc} = \text{pc} + 4
\end{align*}

Compares register \texttt{rs1} to \texttt{rs2}. If comparison is true then \texttt{pc} is updated with \texttt{pc} + \texttt{imm}, otherwise \texttt{pc} becomes \texttt{pc} + 4. 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 Jumps: opcode = JAL = 1101111

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

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

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

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

Behavior: \( \text{imm} = \text{signExtend(immI)} \)
\( \text{pc}[rd] \leq \text{pc} + 4; \text{pc} \leq (\text{R}[rs1]+imm) \& \sim 0x00000001 \) 
(\text{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:

\[ \text{j label } = \text{jal } x0, \text{ label} \ (\text{ignore return address}) \]
\[ \text{li } x1, 0x1000 = \text{lui } x1, 1 \]
\[ \text{li } x1, 0x1100 = \text{lui } x1, 1; \text{addi } x1, x1, 0x100 \]
\[ \text{li } x4, 3 = \text{addi } x4, x0, 3 \]

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

\[ \text{beqz } x1, \text{ target } = \text{beq } x1, x0, \text{ target} \]
\[ \text{bnez } x1, \text{ target } = \text{bneq } x1, x0, \text{ target} \]
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 RISCV 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. \( a = b + 3c; \)

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 \( c \ll 1 + c \). 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.

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

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]

\[
\begin{align*}
\text{li} \ x1, \ 0x1100 & \quad (\text{lui} \ x1, 1; \text{addi} \ x1, \ x1, 0x100) \\
\text{li} \ x2, \ 0x2000 & \quad (\text{lui} \ x2, 2) \\
\text{li} \ x3, \ 0 & \quad (\text{addi} \ x3, \ x0, 0) \\
\text{li} \ x9, \ 10 & \\
\text{loop:} \\
\text{sll} \ x4, \ x3, \ 2 & \quad (x4 = 4 \times i) \\
\text{add} \ x5, \ x1, \ x4 & \quad (x5 = \text{address of a[i]}) \\
\text{add} \ x6, \ x2, \ x4 & \quad (x6 = \text{address of c[i]}) \\
\text{lw} \ x7, \ 0(x5) & \quad (x7 = a[i]) \\
\text{lw} \ x8, \ 0x100(x5) & \quad (x8 = b[i]; \ b \text{ is offset from a by } 0x100) \\
\text{add} \ x7, \ x7, \ x8 & \quad (x7 = a[i] + b[i]) \\
\text{sw} \ x7, \ 0(x6) & \quad (c[i] = a[i] + b[i]) \\
\text{addi} \ x3, \ x3, \ 1 & \quad (i = i + 1) \\
\text{blt} \ x3, \ x9, \ \text{loop} & \quad (\text{branch back to loop if } i < 10)
\end{align*}
\]
**Problem 3.**

Hand assemble the following sequence of instructions into its equivalent binary encoding. (Hint: use the ISA Reference Card at the end of this worksheet 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: \(x_1=8, x_2=10, x_3=12, x_4=0x1234, x_5=24\) 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 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_F876543__________

Line by line decomposition:
1. \(x_1 = 0x1CE8\) – load value 0x1CE8 into x1
2. \(x_4 = 0x87654321\) – load word at address \(x_1 + 0 = 0x1CE8\) into x4
3. Branch into L1 – if \((0x87654321 < 0)\), then jump to L1
4. \(x_2 = 0xF876432\) – \(0x8765432 >> 4\) 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 = 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:
```
### 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>LUI</td>
<td>lui rd, luiConstant</td>
<td>Load Upper Immediate</td>
<td>reg[rd] &lt;= luiConstant * 12</td>
</tr>
<tr>
<td>JAL</td>
<td>jal rd, label</td>
<td>Jump and Link</td>
<td>reg[rd] &lt;= pc + 4</td>
</tr>
<tr>
<td>JALR</td>
<td>jalr rd, offset(rs1)</td>
<td>Jump and Link Register</td>
<td>reg[rd] &lt;= pc + 4</td>
</tr>
<tr>
<td>BEQ</td>
<td>beq rs1, rs2, label</td>
<td>Branch if =</td>
<td>pc &lt;= (reg[rs1] == reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>BNE</td>
<td>bne rs1, rs2, label</td>
<td>Branch if #</td>
<td>pc &lt;= (reg[rs1] != reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>BLT</td>
<td>blt rs1, rs2, label</td>
<td>Branch if &lt; (Signed)</td>
<td>pc &lt;= (reg[rs1] &lt; reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>BGE</td>
<td>bge rs1, rs2, label</td>
<td>Branch if ≥ (Signed)</td>
<td>pc &lt;= (reg[rs1] &gt;= reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>BLTU</td>
<td>bltu rs1, rs2, label</td>
<td>Branch if &lt; (Unsigned)</td>
<td>pc &lt;= (reg[rs1] &lt; reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>BGEU</td>
<td>bgeu rs1, rs2, label</td>
<td>Branch if ≥ (Unsigned)</td>
<td>pc &lt;= (reg[rs1] &gt;= reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>LW</td>
<td>lw rd, offset(rs1)</td>
<td>Load Word</td>
<td>reg[rd] &lt;= mem[reg[rs1] + offset]</td>
</tr>
<tr>
<td>SW</td>
<td>sw rs2, offset(rs1)</td>
<td>Store Word</td>
<td>mem[reg[rs1] + offset] &lt;= reg[rs2]</td>
</tr>
<tr>
<td>ADDI</td>
<td>addi rd, rs1, constant</td>
<td>Add Immediate</td>
<td>reg[rd] &lt;= reg[rs1] + constant</td>
</tr>
<tr>
<td>SLTI</td>
<td>slti rd, rs1, constant</td>
<td>Compare &lt; Immediate (Signed)</td>
<td>reg[rd] &lt;= (reg[rs1] &lt; constant) ? 1 : 0</td>
</tr>
<tr>
<td>SLTIU</td>
<td>sltiu rd, rs1, constant</td>
<td>Compare &lt; Immediate (Unsigned)</td>
<td>reg[rd] &lt;= (reg[rs1] &lt; constant) ? 1 : 0</td>
</tr>
<tr>
<td>XORI</td>
<td>xorI rd, rs1, constant</td>
<td>XOR Immediate</td>
<td>reg[rd] &lt;= (reg[rs1] ^ reg[rs2]) constant</td>
</tr>
<tr>
<td>ORI</td>
<td>ori rd, rs1, constant</td>
<td>OR Immediate</td>
<td>reg[rd] &lt;= reg[rs1]</td>
</tr>
<tr>
<td>ANDI</td>
<td>andi rd, rs1, constant</td>
<td>AND Immediate</td>
<td>reg[rd] &lt;= reg[rs1]</td>
</tr>
<tr>
<td>SLLI</td>
<td>slli rd, rs1, constant</td>
<td>Shift Left Logical Immediate</td>
<td>reg[rd] &lt;= reg[rs1] + constant</td>
</tr>
<tr>
<td>SLLI</td>
<td>slli rd, rs1, constant</td>
<td>Shift Left Logical Immediate</td>
<td>reg[rd] &lt;= reg[rs1] + constant</td>
</tr>
<tr>
<td>SRLI</td>
<td>srrl rd, rs1, constant</td>
<td>Shift Right Logical</td>
<td>reg[rd] &lt;= reg[rs1] + reg[rs2]</td>
</tr>
<tr>
<td>SRAI</td>
<td>sral rd, rs1, rs2</td>
<td>Shift Right Arithmetic</td>
<td>reg[rd] &lt;= reg[rs1] + reg[rs2]</td>
</tr>
<tr>
<td>ORI</td>
<td>orI rd, rs1, rs2</td>
<td>OR</td>
<td>reg[rd] &lt;= reg[rs1] + reg[rs2]</td>
</tr>
<tr>
<td>ANDI</td>
<td>andi rd, rs1, rs2</td>
<td>AND</td>
<td>reg[rd] &lt;= reg[rs1] + reg[rs2]</td>
</tr>
</tbody>
</table>

Note: luiConstant is a 20-bit value. offset and constant are signed 12-bit values that are sign-extended to 32-bit values. label is a 32-bit memory address or its alias name.

### MIT 6.004 ISA Reference Card: Pseudoinstructions

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>li rd, constant</td>
<td>Load Immediate</td>
<td>reg[rd] &lt;= constant</td>
</tr>
<tr>
<td>mv rd, rs1</td>
<td>Move</td>
<td>reg[rd] &lt;= reg[rs1] + 0</td>
</tr>
<tr>
<td>nand rd, rs1</td>
<td>Logical Not</td>
<td>reg[rd] &lt;= reg[rs1] *</td>
</tr>
<tr>
<td>neg rd, rs1</td>
<td>Arithmetic Negation</td>
<td>reg[rd] &lt;= 0 - reg[rs1]</td>
</tr>
<tr>
<td>j label</td>
<td>Jump</td>
<td>pc &lt;= label</td>
</tr>
<tr>
<td>jal label</td>
<td>Jump and Link (with ra)</td>
<td>reg[ra] &lt;= pc + 4</td>
</tr>
<tr>
<td>jr rs</td>
<td>Jump Register</td>
<td>pc &lt;= reg[rs1] &amp; ~1</td>
</tr>
<tr>
<td>jalr rs</td>
<td>Jump and Link Register (with ra)</td>
<td>reg[ra] &lt;= pc + 4</td>
</tr>
<tr>
<td>ret</td>
<td>Return from Subroutine</td>
<td>pc &lt;= reg[ra]</td>
</tr>
<tr>
<td>bgt rs1, rs2, label</td>
<td>Branch &gt; (Signed)</td>
<td>pc &lt;= (reg[rs1] &gt; reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>ble rs1, rs2, label</td>
<td>Branch ≤ (Signed)</td>
<td>pc &lt;= (reg[rs1] &lt;= reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>bgtu rs1, rs2, label</td>
<td>Branch &gt; (Unsigned)</td>
<td>pc &lt;= (reg[rs1] &gt; reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>bleu rs1, rs2, label</td>
<td>Branch ≤ (Unsigned)</td>
<td>pc &lt;= (reg[rs1] &lt;= reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>beqz rs1, label</td>
<td>Branch = 0</td>
<td>pc &lt;= (reg[rs1] == 0) ? label : pc + 4</td>
</tr>
<tr>
<td>bnez rs1, label</td>
<td>Branch ≠ 0</td>
<td>pc &lt;= (reg[rs1] != 0) ? label : pc + 4</td>
</tr>
<tr>
<td>bltz rs1, label</td>
<td>Branch &lt; 0 (Signed)</td>
<td>pc &lt;= (reg[rs1] &lt; 0) ? label : pc + 4</td>
</tr>
<tr>
<td>bnez rs1, label</td>
<td>Branch ≥ 0 (Signed)</td>
<td>pc &lt;= (reg[rs1] &gt;= 0) ? label : pc + 4</td>
</tr>
<tr>
<td>bgtz rs1, label</td>
<td>Branch &gt; 0 (Signed)</td>
<td>pc &lt;= (reg[rs1] &gt; 0) ? label : pc + 4</td>
</tr>
<tr>
<td>blez rs1, label</td>
<td>Branch ≤ 0 (Signed)</td>
<td>pc &lt;= (reg[rs1] &lt;= 0) ? label : pc + 4</td>
</tr>
</tbody>
</table>
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>x8</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>t8–t2</td>
<td>Temporary registers</td>
<td>Caller</td>
</tr>
<tr>
<td>x8–x9</td>
<td>s8–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>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RV32I Base Instruction Set (MIT 6.004 subset)

| imm[11:0] | rs1 | 010 | rd | 0000001 | LW |
| imm[11:3] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |
| imm[11:0] | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | rs1 | 011 | rd | 0010011 | SLTU |
| 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 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 00000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 01000000 | shamt | 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 | SLTU |
| 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 |

- For JAL and branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU), the immediate encodes the target address as an offset from the current pc (i.e., pc + imm = label).
- Not all immediate bits are encoded. Missing lower bits are filled with zeros and missing upper bits are sign-extended.