# RV32IM

# PIPELINED PROCESSOR

## (CO502 – Advanced Computer Architecture)

## (PHASE 01 – PART 01)

Group 04
E/20/032 – A.M.N.C. Bandara
E/20/034 – G.M.M.R. Bandara
E/20/157 – S.M.B.G. Janakantha

Department of Computer Engineering
University of Peradeniya
Sri Lanka

# ACTIVITY 01: Instruction Types, Encoding Formats, and Their Opcodes

Based on handling the immediate, RISC-V architecture has 6 types of instructions. They are:

1. R – Type (Register Type)
2. I – Type (Immediate Type)
3. S – Type (Store Type)
4. B – Type (Branch Type)
5. U – Type (Upper Immediate)
6. J – Type (Jump Type)

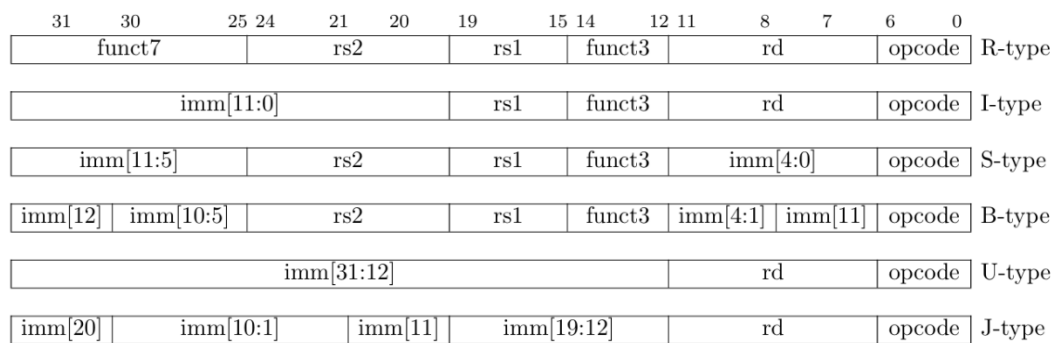Let's consider the details one by one.



*Figure 1: RISC-V base instruction formats showing immediate variants.*

## 1. R – Type (Register Type) Instructions

R-Type (Register Type) instructions perform arithmetic and logical operations on registers. These instructions take three register operands: two source registers (rs1, rs2) and one destination register (rd). The result of the operation is stored in the destination register.
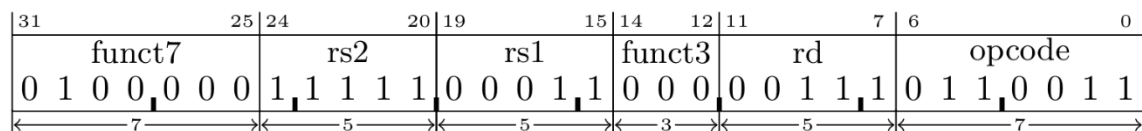


*Figure 2: R-Type Instruction Encoding Format*

- `Bits(6-0):` The `OP-CODE` field identifies the instruction type (for R-Type instructions, this is typically **0110011**). The control logic should use this to interpret the remaining fields and derive the control signals.

- `Bits(11-7):` The `rd` field identifies the destination register. After each R-type operation, the result is written into this register.

- `Bits(14-12):` The `funct3` field further specifies the operation (e.g., for distinguishing between **add** and **sub**).

- `Bits(19-15):` The `rs1` field identifies the source register 1.

- `Bits(24-20):` The `rs2` field identifies the source register 2.

- `Bits(31-25):` The `funct7` field specifies the operation (e.g., for distinguishing between **add** and **sub**).

*Table 1: OP-CODEs of R-Type Instruction Set*

| Instruction | Description | Opcode | func3 | func7 |
|---|---|---|---|---|
| add | ADD the value in rs1 and rs2 value and put it into rd.<br>Syntax: add rd, rs1, rs2<br>C code: rd = rs1 + rs2 | 0110011 | 000 | 0000000 |
| sub | SUBSTRACT the value in rs1 by rs2 value and put it into rd.<br>Syntax: sub rd, rs1, rs2<br>C code: rd = rs1 - rs2 | 0110011 | 000 | 0100000 |
| sll | LOGICALLY LEFT SHIFT the bits of rs1 by rs2 amount and put it into rd.<br>Syntax: sll rd, rs1, rs2<br>C code: rd = rs1 << rs2 | 0110011 | 001 | 0000000 |
| slt | Sets the rd to 1 if the sign integer value of rs1 is less than the sign integer value rs2; otherwise, it sets it to 0.<br>Syntax: slt rd, rs1, rs2<br>C code: rd = (rs1<rs2) ? 1 : 0 | 0110011 | 010 | 0000000 |
| sltu | Sets the rd to 1 if the unsigned integer value of rs1 is less than the unsigned integer value rs2; otherwise, it sets it to 0.<br>Syntax: sltu rd, rs1, rs2<br>C code: rd = (rs1<rs2) ? 1 : 0 | 0110011 | 011 | 0000000 |
| xor | Perform the bitwise XOR operation on the values in rs1 and rs2 and put it into rd.<br>Syntax: xor rd, rs1, rs2<br>C code: rd = rs1 ^ rs2 | 0110011 | 100 | 0000000 |
| srl | LOGICALLY RIGHT SHIFT the bits of rs1 by rs2 amount and put it into rd.<br>Syntax: srl rd, rs1, rs2<br>C code: rd = rs1 >> rs2 | 0110011 | 101 | 0000000 |
| sra | ARITHMETICALLY RIGHT SHIFT bits of rs1 by rs2 amount and put it into rd.<br>Syntax: sra rd, rs1, rs2<br>C code: rd = rs1 >> rs2 (sign-extended) | 0110011 | 101 | 0100000 |
| Or | Perform the bitwise OR operation on the values in rs1 and rs2 and put it into rd.<br>Syntax: or rd, rs1, rs2<br>C code: rd = rs1 | rs2 | 0110011 | 110 | 0000000 |
| and | Perform the bitwise AND operation on the values in rs1 and rs2 and put it into rd.<br>Syntax: and rd, rs1, rs2<br>C code: rd = rs1 & rs2 | 0110011 | 111 | 0000000 |

| | | | | |
|---|---|---|---|---|
| mul | Multiplies the values in rs1 and rs2 and stores the lower 32 bits of the result in rd.<br>Syntax: mul rd, rs1, rs2<br>C code: rd = (rs1*rs2)[31:0] | 0110011 | 000 | 0000001 |
| mulh | Multiply High Signed, Multiplies the values in rs1 and rs2 as signed integers and stores the upper 32 bits of the result in rd.<br>Syntax: mulh rd, rs1, rs2<br>C code: rd = (rs1*rs2)[63:32] | 0110011 | 001 | 0000001 |
| mulhsu | Multiply High Signed-Unsigned, Multiplies rs1 as a signed integer and rs2 as an unsigned integer. The upper 32 bits of the result are stored in rd.<br>Syntax: mulhsu rd, rs1, rs2<br>C code: rd = (rs1*rs2)[63:32] | 0110011 | 010 | 0000001 |
| mulhu | Multiply High Unsigned, Multiplies rs1 and rs2 as unsigned integers. The upper 32 bits of the result are stored in rd.<br>Syntax: mulhsu rd, rs1, rs2<br>C code: rd = (rs1*rs2)[63:32] | 0110011 | 011 | 0000001 |
| div | Divides rs1 by rs2 as signed integers and stores the quotient in rd. Division by zero results in an undefined value in rd.<br>Syntax: div rd, rs1, rs2<br>C code: rd = rs1 / rs2 | 0110011 | 100 | 0000001 |
| divu | Divide Unsigned, divides rs1 by rs2 as unsigned integers and stores the quotient in rd. Division by zero results in an undefined value in rd.<br>Syntax: div rd, rs1, rs2<br>C code: rd = rs1 / rs2 | 0110011 | 101 | 0000001 |
| rem | Remainder Signed, computes the remainder of rs1 divided by rs2 as signed integers and stores the result in rd. Division by zero results in an undefined value in rd.<br>Syntax: rem rd, rs1, rs2<br>C code: rd = rs1 % rs2 | 0110011 | 110 | 0000001 |
| remu | Remainder Unsigned, computes the remainder of rs1 divided by rs2 as unsigned integers and stores the result in rd. Division by zero results in an undefined value in rd.<br>Syntax: remu rd, rs1, rs2<br>C code: rd = rs1 % rs2 | 0110011 | 111 | 0000001 |

## 2. I – Type (Immediate Type) Instructions

The I-Type (Immediate Type) instruction format is used to encode instruction with a signed 12-bit immediate operand (there are some special cases where this number changes) with a range of –2048 to 2047, a rd register, and a rs1 register.
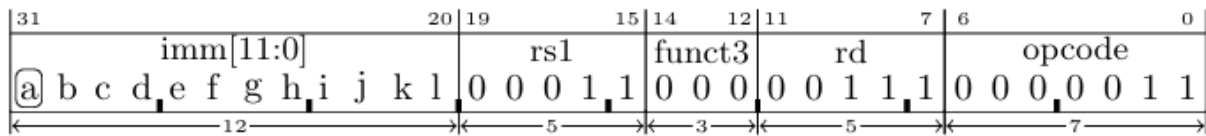


*Figure 3: I-Type Instruction Encoding Format*

- **Bits(6-0):** OP-CODE of Instructions. These bits are used to derive control signals by the control logic. For I-type instructions, there are 3 opcodes that specify different types of operations.
    - o 0000011 – Load Immediate instructions
    - o 0010011 – Arithmetic immediate instructions
    - o 1100111 – Jump instruction

- **Bits(11-7):** The rd field identifies the destination register. Where the results will be saved for all immediate arithmetic and load instructions.

- **Bits(14-12):** The func3 field. Which specifies the type of operation based on opcode. Which is used to generate control signals in **ALUOP** and how load works

- **Bits(19-15):** The rs1, The source register address.

- **Bits(31-20):** 12 bit immediate value.

- **Bits[30]:** This bit is specifically used in **addi** and **srli/srai** instructions to differentiate from one other. (For addi when ins[30] = 0, which implies this is an **add** instruction, if ins[30] = 1, then it's a **sub** instruction.)

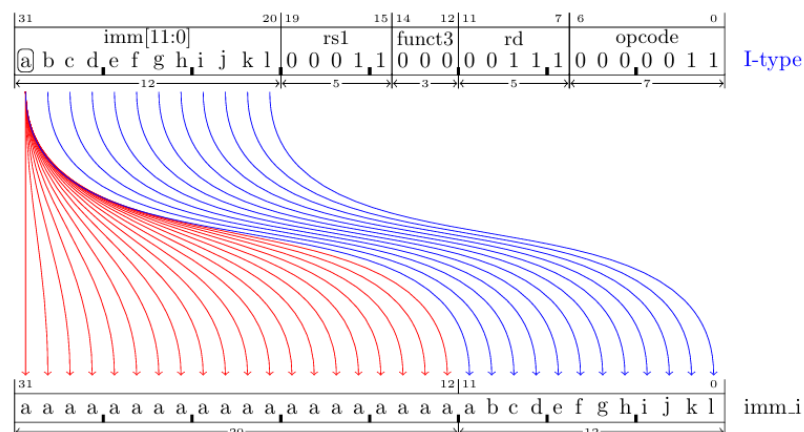❖ For all the I-type instructions (other than shift) the decoding of instructions as follows,



*Figure 4: I-Type Instruction Immediate Decoding*
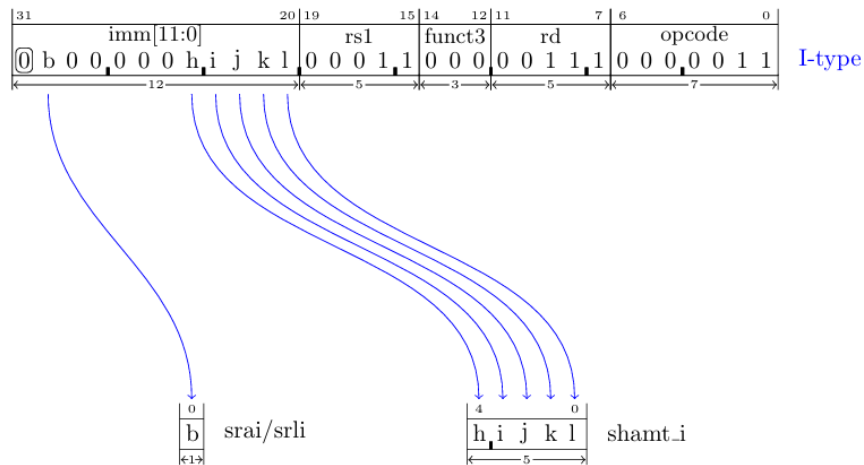
❖ For shifting instructions,



*Figure 5: I-Type Shift Instruction Immediate Decoding*

*Table 2: OP-CODEs of I-Type Instruction Set*

| Instruction | Description | Opcode | func3 | func7 |
|---|---|---|---|---|
| lb | Load Byte, set register rd to the value of the sign-extended byte fetched from the memory address given by the sum of rs1 and imm_i.<br>Syntax: lb rd, imm(rs1)<br>C code: rd = mem[rs1+imm_i] | 0000011 | 000 | N/A |
| lh | Load Halfword, set register rd to the value of the sign-extended 16-bit little-endian half-word value fetched from the memory address given by the sum of rs1 and imm_i.<br>Syntax: lh rd, imm(rs1)<br>C code: rd = mem[rs1+imm_i] | 0000011 | 001 | N/A |
| lw | Load Word, set register rd to the value of the sign-extended 32-bit little-endian word value fetched from the memory address given by the sum of rs1 and imm_i.<br>Syntax: lw rd, imm(rs1)<br>C code: rd = mem[rs1+imm_i] | 0000011 | 010 | N/A |
| lbu | Load Byte Unsigned, set register rd to the value of the zero-extended byte fetched from the memory address given by sum of rs1 and imm_i.<br>Syntax: lbu rd, imm(rs1)<br>C code: rd = zeroExtend(mem[rs1+imm_i]) | 0000011 | 100 | N/A |

| | | | | |
|---|---|---|---|---|
| lhu | Load Halfword Unsigned, set register rd to the value of the sign-extended 16-bit little-endian half-word value fetched from the memory address given by the sum of rs1 and imm_i.<br>Syntax: lhu rd, imm(rs1)<br>C code: rd = zeroExtend(mem[rs1+imm_i]) | 0000011 | 101 | N/A |
| addi | Add the rs1 value and imm_i and store the result in register rd.<br>Syntax: addi rd, rs1, imm<br>C code: rd = rs1 + imm_i | 0010011 | 000 | N/A |
| slli | LOGICALLY LEFT SHIFT the bits of rs1 by shamt_i amount and put it into rd. (0's will be added to the to the shifted bits in *LSB* side)<br>Syntax: slli rd, rs1, shamt<br>C code: rd = rs1 << shamt_i | 0010011 | 001 | 000000 |
| slti | Sets the rd to 1 if the sign integer value of rs1 is less than the sign integer value imm_i; otherwise, it sets it to 0.<br>Syntax: slti rd, rs1, imm<br>C code: rd = (rs1 < imm_i) ? 1 : 0 | 0010011 | 010 | N/A |
| sltiu | Sets the rd to 1 if the unsigned integer value of rs1 is less than the unsigned integer value imm_i; otherwise, it sets it to 0.<br>Syntax: sltui rd, rs1, rs2<br>C code: rd = (rs1 < imm_i) ? 1 : 0 | 0010011 | 011 | N/A |
| xori | Bitwise XOR rs1 and imm_i and store the result in rd. (imm will be sign extended)<br>Syntax: xori rd, rs1, imm<br>C code: rd = rs1 ^ imm_i | 0010011 | 100 | N/A |
| srli | LOGICALLY RIGHT SHIFT the bits of rs1 by shamt_i amount and put it into rd. (0's will be added to the shifted bits in *MSB* side)<br>Syntax: srl rd, rs1, imm<br>C code: rd = rs1 >> shamt_i | 0010011 | 101 | 000000 |
| srai | ARITHMETICALLY RIGHT SHIFT bits of rs1 by shamt_i amount and put it into rd.<br>Syntax: srai rd, rs1, imm<br>C code: rd = rs1 >> shamt_i (sign-extended) | 0010011 | 101 | 010000 |

| | | | | |
|---|---|---|---|---|
| ori | Bitwise OR `rs1` and `imm_i` and store the result in `rd`.<br>Syntax: `ori rd, rs1, imm`<br>C code: `rd = rs1 | imm_i` | 0010011 | 110 | N/A |
| andi | Bitwise AND `rs1` and `imm_i` and store the result in `rd`.<br>Syntax: `ori rd, rs1, imm`<br>C code: `rd = rs1 & imm_i` | 0010011 | 111 | N/A |
| jalr | Jumps to an address specified by the `rs1` and the `imm_i` value, and saves the return address in the `rd`.<br>Syntax: `jalr rd, imm(rs1)`<br>C code:<br>   `rd = PC + 4` (address of the next instruction)<br>   `PC = (rs1 + imm_i) & ~1` (target address) | 1100111 | 000 | N/A |

## 3. S – Type (Store Type) Instructions

S-Type (Store Type) is used for store operations, which store data from a register to memory. They include two register operands and a 12-bit immediate value for the memory address offset.
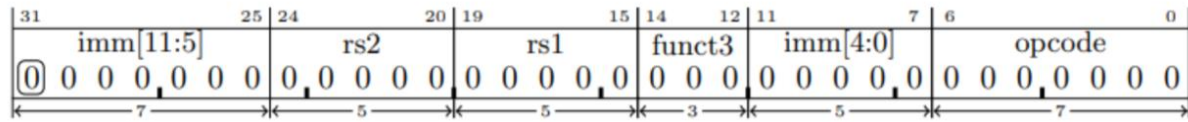


*Figure 6: S-Type Instruction Encoding Format*

- `Bits(6-0):` The `OP-CODE` field identifies the instruction type (for S-Type instructions, this is typically **0100011**). The control logic should use this to interpret the remaining fields and derive the control signals.

- `Bits(14-12):` The `funct3` part is used to encode the type

- `Bits(19-15):` The `rs1` field identifies the source register 1 that is added to the 12-bit immediate field to form the memory address.

- `Bits(24-20):` The `rs2` field identifies the source register 2. That is the source register whose value should be stored into memory.

- `Bits(31-25)(11-7):` The 12-bit `immediate` offset is split into two parts:
    - `imm[11:5]:` using the `Bits(31-25)` field of the instruction
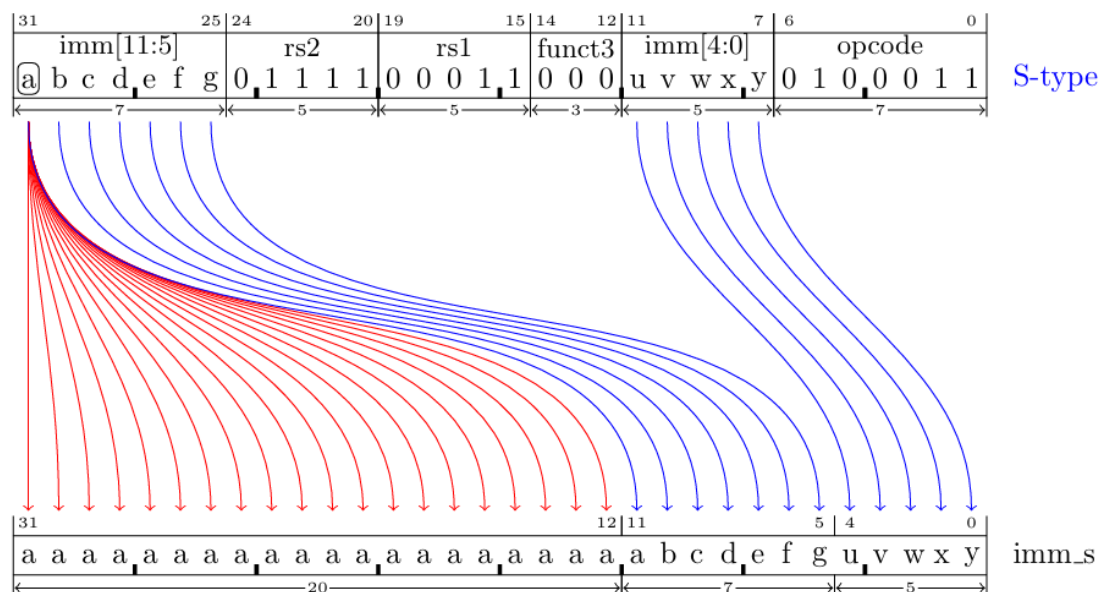    - `imm[4:0]:` using the `Bits(11-7)` field of the instruction



*Figure 7: Decoding an S-type Instruction.*

*Table 3: OP-CODEs of S-Type Instruction Set*

| Instruction | Description | Opcode | func3 | func7 |
|:---:|---|:---:|:---:|:---:|
| sb | `Store  byte` takes a byte from the least significant 8 bits of the `rs2` and writes it to the memory address given by `imm_s + rs1`<br>Syntax: `sb rs2, imm_s(rs1)`<br>C code: `mem[rs1+imm_s] = rs2[7:0]` | 0100011 | 000 | N/A |
| sh | `Store  half` takes a halfword from the least significant 16 bits of the `rs2` and writes it to the memory address given by `imm_s + rs1`<br>Syntax: `sh rs2, imm_s(rs1)`<br>C code: `mem[rs1+imm_s] = rs2[15:0]` | 0100011 | 001 | N/A |
| sw | `Store  word` takes a word from the least significant 32 bits of the `rs2` and stores it to the memory address given by `imm_s + rs1`<br>Syntax: `sw rs2, imm_s(rs1)`<br>C code: `mem[rs1+imm_s] = rs2[31:0]` | 0100011 | 111 | N/A |

## 4. B – Type (Branch Type) Instructions

B-Type (Branch Type) instructions perform conditional branches based on the components of two registers (rs1, rs2) values. Here we compare the values of the rs1 and rs2. If the comparison is true, the program counter (pc) is updated to the target address computed from the relative branch offset. If the comparison is false, the program counter is incremented to the next sequential instruction.
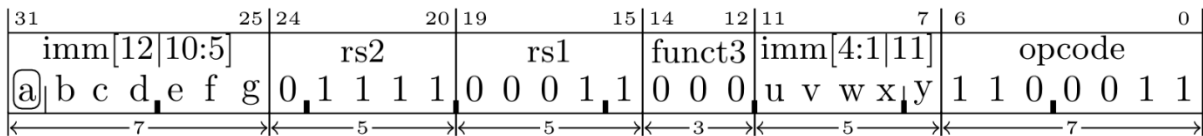


*Figure 8: B-Type Instruction Encoding Format*

- `Bits(6-0):` The `OP-CODE` field identifies the instruction type (for B-Type instructions, this is typically **1100111**). The control logic should use this to interpret the remaining fields and derive the control signals.

- `Bits(14-12):` The `funct3` field further determines the specific branch type.

- `Bits(19-15):` The `rs1` field identifies the source register 1.

- `Bits(24-20):` The `rs2` field identifies the source register 2..

- `Bits(31-25)(11-7):` The 13-bit `imm` field split into 4 parts:
    - `imm[12]:` using the `Bits(31)` field of the instruction
    - `imm[11]:` using the `Bits(7)` field of the instruction
    - `imm[10:5]:` using the `Bits(30-25)` field of the instruction
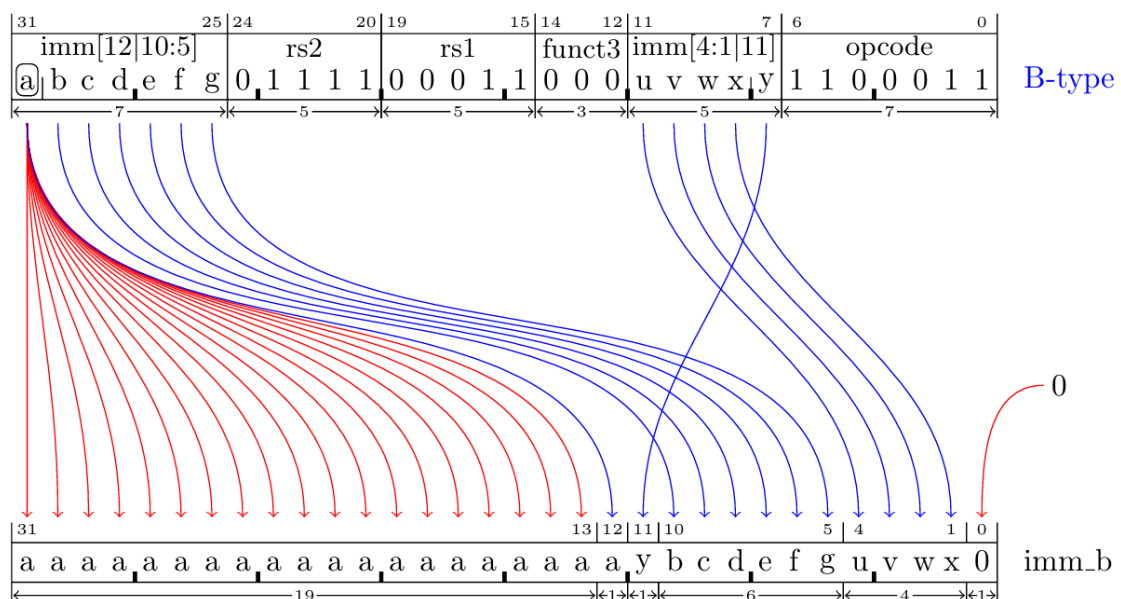    - `imm[4:1]:` using the `Bits(11-8)` field of the instruction



*Figure 9: Decoding a B-type Instruction.*

*Table 4: OP-CODEs of B-Type Instruction Set*

| Instruction | Description | Opcode | func3 | func7 |
|---|---|---|---|---|
| beq | If the values in `rs1` and `rs2` are `equal`, add `imm_b` to the `pc` register. Otherwise, go to the next instruction.<br>Syntax: `beq rs1, rs2, imm_b`<br>C code: `pc = pc + (rs1==rs2 ? imm_b : 4)` | 0100011 | 000 | N/A |
| bne | If the values in `rs1` and `rs2` are `not equal`, add `imm_b` to the `pc` register. Otherwise, go to the next instruction.<br>Syntax: `bne rs1, rs2, imm_b`<br>C code: `pc = pc + (rs1!=rs2 ? imm_b : 4)` | 0100011 | 001 | N/A |
| blt | If the signed value in `rs1` is `less than` the signed value in `rs2`, add `imm_b` to the `pc` register. Otherwise, go to the next instruction.<br>Syntax: `blt rs1, rs2, imm_b`<br>C code: `pc = pc + (rs1<rs2 ? imm_b : 4)` | 0100011 | 100 | N/A |
| bge | If the signed value in `rs1` is `greater than or equal to` the signed value in `rs2`, add `imm_b` to the `pc` register. Otherwise, go to the next instruction.<br>Syntax: `bge rs1, rs2, imm_b`<br>C code: `pc = pc + (rs1>=rs2 ? imm_b : 4)` | 0100011 | 101 | N/A |
| bltu | If the unsigned value in `rs1` is `less than` the unsigned value in `rs2`, add `imm_b` to the `pc` register. Otherwise, go to the next instruction.<br>Syntax: `bltu rs1, rs2, imm_b`<br>C code: `pc = pc + (rs1<rs2 ? imm_b : 4)` | 0100011 | 110 | N/A |
| bequ | If the unsigned value in `rs1` is `greater than or equal to` the unsigned value in `rs2`, add `imm_b` to the `pc` register. Otherwise, go to the next instruction.<br>Syntax: `bgeu rs1, rs2, imm_b`<br>C code: `pc = pc + (rs1>=rs2 ? imm_b : 4)` | 0100011 | 111 | N/A |

## 5. U – Type (Upper Immediate) Instructions

The U-type (Upper Immediate) format is used for instructions that use a 20-bit immediate operand and destination register (rd). The **rd** field contains a register address that would be set to a value depending on the instruction.
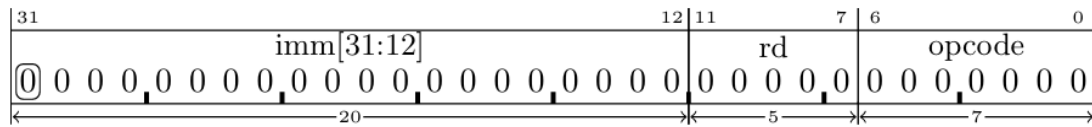


*Figure 10: U-Type Instruction Encoding Format*

- `Bits(6-0):` The `OP-CODE` field of Instructions. These bits are used to derive control signals by the control logic. For J-type instructions, there are 2 variants:
    - `0010111` – `auipc` instruction
    - `0110111` – `lui` instruction

- `Bits(11-7):` The `rd` field identifies the destination register. Where the relevant value will be stored after operation.

- `Bits(31-12):` 20-bit `immediate` value.

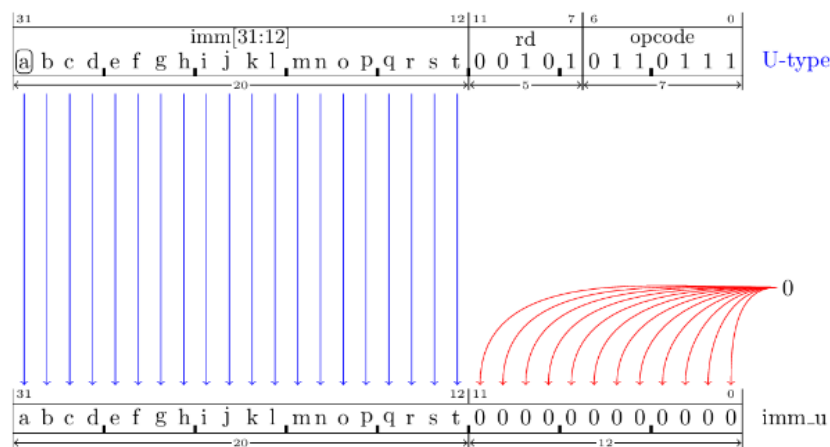- `Bits[31]:` Used to sign extend the 21-bit `immediate` value.



*Figure 11: U-Type Instruction Immediate Decoding*

*Table 5: OP-CODEs of U-Type Instruction Set*

| Instruction | Description | Opcode | func3 | func7 |
|---|---|---|---|---|
| auipc | Add the address of the instruction to the `imm_u` value and store the result in register `rd`<br>Syntax: `auipc rd, imm_u`<br>C code: `rd = pc + imm_u` | 0010111 | N/A | N/A |
| lui | Set the register `rd` to the `imm_u` value.<br>Syntax: `lui rd, imm`<br>C code: `rd = imm_u` | 0110111 | N/A | N/A |

## 6. J – Type (Jump Type) Instructions

The J-type (Jump Type) instructions perform unconditional jumps to the target address. It is similar to the U-type, but bits in the immediate operand are arranged in a different order.
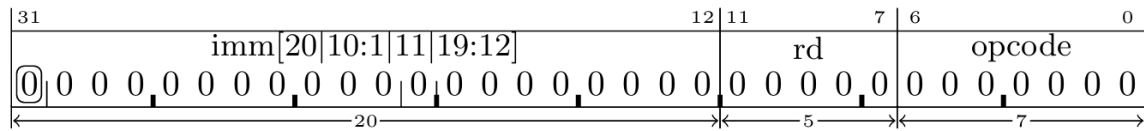


*Figure 12: J-Type Instruction Encoding Format*

- `Bits(6-0):`    The `OP-CODE` of Instructions. These bits are used to derive control signals by the control logic. For J-type instructions, it's **1101111**.

- `Bits(11-7):`    The `rd` field identifies the destination register (link register address). Where the return address (`pc + 4`) will be stored.

- `Bits(31-12):` 20-bit `immediate` value.

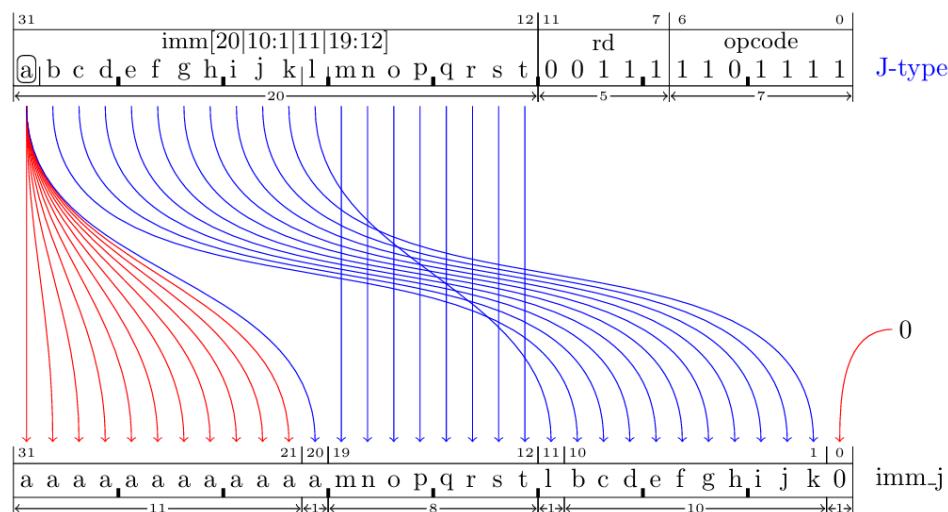- `Bits[31]:`    Used to sign extend the 21-bit immediate value.



*Figure 13: J-Type Instruction Immediate Decoding*

*Table 6: OP-CODEs of J-Type Instruction Set*

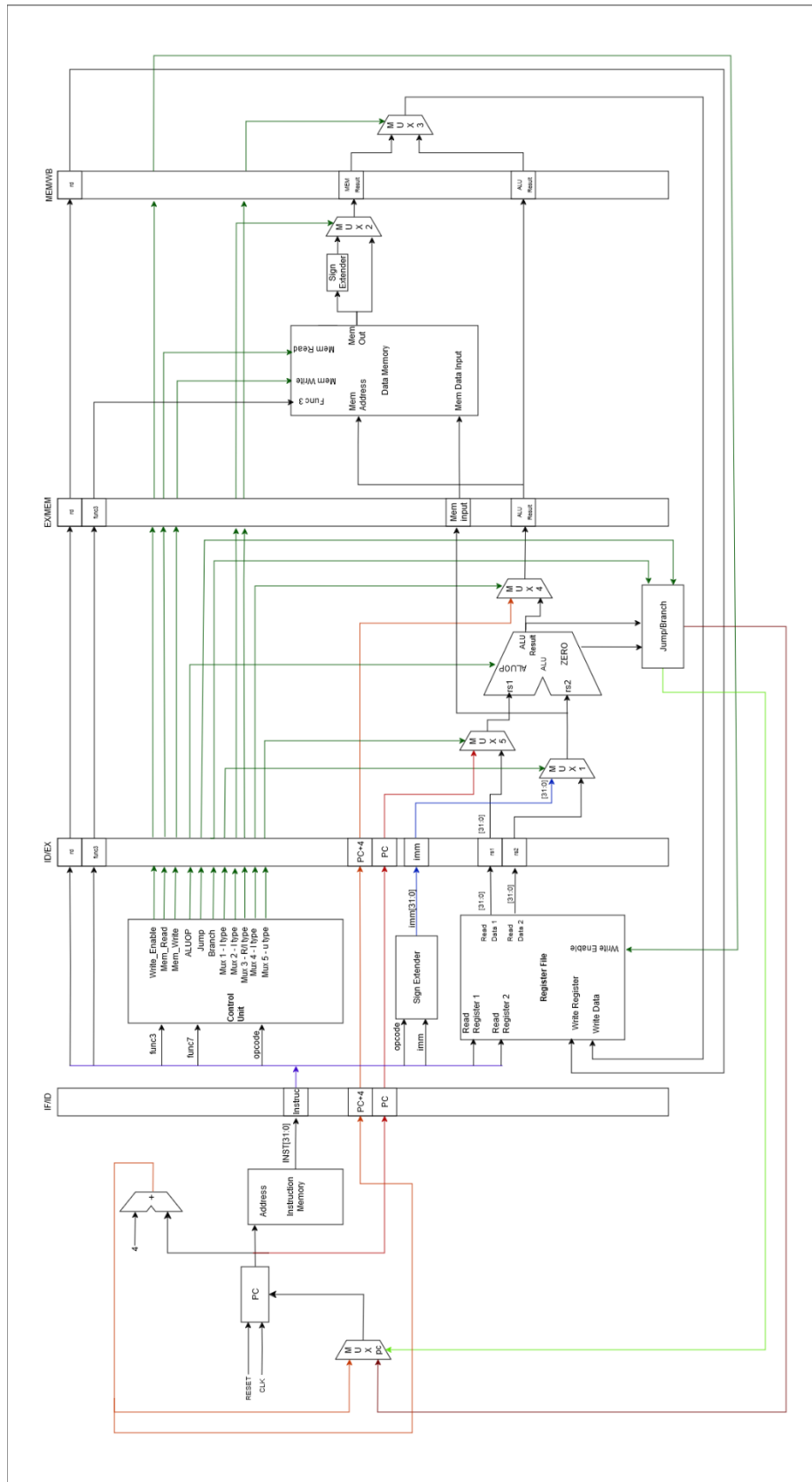| Instruction | Description | Opcode | func3 | func7 |
|---|---|---|---|---|
| jal | Set `rd` to the address of the next instruction (address of the `jal + 4`) and then jump to the address given by the `pc + imm_j` value.<br>Syntax: `jal rd, pcrel21`<br>Description:<br>    rd = PC + 4  (address of the next instruction)<br>    PC = pc + imm_j  (target address) | 1101111 | N/A | N/A |

# ACTIVITY 02: Data Path



*Figure 14: RISC-V 32IM Pipelined Data Path*

Source File: datapath.png, draw.io

# ACTIVITY 03: Testing

Here we hope to test the code with the following cases.

1. Correctness of individual instructions
2. Pipelined behaviour
3. Sequential Instruction Testing
4. Edge Cases

## Individual Instruction Testing

Arithmetic Instructions (ADD, SUB, MUL, etc.)

- Input: `add x1, x2, x3`        *(x2 = 5, x3 = 3).*
- Expected Output: x1 = 8.

Load/Store Instructions (LW, SW)

- Input: `lw x1, 0(x2)`        *(x2 points to memory holding 0x12345678).*
- Expected Output: x1 = 0x12345678.

Branch Instructions (BEQ, BNE, BLT, etc.)

- Input: `beq x1, x2, label`        *(x1 = x2).*
- Expected Output: PC jumps to label.

Immediate Instructions (ADDI, ANDI, etc.)

- Input: `addi x1, x2, 10`        *(x2 = 5).*
- Expected Output: x1 = 15.

## Pipeline Behaviour Testing

Data Hazard Without Forwarding

- Input:
```
add x1, x2, x3      // x1 = x2 + x3
sub x4, x1, x5      // Use x1 immediately
```

- Expected Output: Stall in the pipeline (bubble inserted).

Data Hazard with Forwarding

- Input:
```
add x1, x2, x3      // x1 = x2 + x3
sub x4, x1, x5      // Forward x1 to EX stage
```

- Expected Output: No stalls; result forwarded correctly.

Control Hazard

- Input: `beq x1, x2, label    // Branch condition`
  `      add x3, x4, x5      // Next instruction`

- Expected Output: Flush pipeline if branch is taken; PC updated correctly.

Structural Hazard

- Input: Test simultaneous memory access (e.g., instruction fetch and data access).
- Expected Output: No conflict if the pipeline handles it; stall otherwise.

## Sequential Instruction Testing

Basic Arithmetic Sequence

- Input: `add x1, x0, x0       // x1 = 0`
  `      addi x2, x0, 5      // x2 = 5`
  `      add x3, x1, x2      // x3 = x1 + x2`
  `      sub x4, x3, x2      // x4 = x3 - x2`

- Expected Output: x3 = 5, x4 = 0.

Load/Store Sequence

- Input: `sw x2, 0(x3)         // Store x2 into memory`
  `      lw x4, 0(x3)        // Load from memory into x4`

- Expected Output: x4 = x2.

Branch and Loop

- Input: `addi x1, x0, 0           // Initialize x1 = 0`
  `      addi x2, x0, 5          // Set x2 = 5`
  `      Loop: add x1, x1, x2    // x1 += x2`
  `            subi x3, x3, 1    // Decrement counter`
  `            bne x3, x0, Loop  // Branch if not zero`

- Expected Output: x1 = 5 * iterations

## **Edge Case Testing**

Branch Target Alignment

- Input: Branch to misaligned address.
- Expected Output: Raise exception or align address.

Memory Alignment

- Input: Load/Store with unaligned address (e.g., `lw x1, 3(x2)`).
- Expected Output: Correct exception handling.

Division by Zero

- Input: DIV x1, x2, x3 (x3 = 0).
- Expected Output: Raise exception or return specific value.

Instruction Fetch Beyond Memory

- Input: Fetch instruction from invalid memory address.
- Expected Output: Raise exception.

# Reference:

johnwinans (2024). *Release spelling and phrasing improvements · johnwinans/rvalp*. [online] GitHub. Available at: https://github.com/johnwinans/rvalp/releases/tag/v0.18.3 [Accessed 1 Dec. 2024].

John's Basement (2020). *RISC-V RV32I Instruction Encoding*. [online] YouTube. Available at: https://www.youtube.com/watch?v=VNy-J0u7-jY&list=PL3by7evD3F53Dz2RiB47Ztp9l_piGVuus&index=2 [Accessed 1 Dec. 2024].

Waterman, A. and Asanović, K. (2017). *The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2*. [online] Available at: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf. [Accessed 1 Dec. 2024].

GitHub. (n.d.). *RISC-V*. [online] Available at: https://github.com/riscv. [Accessed 1 Dec. 2024]

Cs61c.org. (2024). *Home | CS 61C Fall 2024*. [online] Available at: https://cs61c.org/fa24/ [Accessed 1 Dec. 2024].