

Task2: RISC-V Instruction set

===== by N Karthik Reddy

Objective: understand various types of RISC-V instructions and where it can be used.

RISC-V stands for **Reduced Instruction set Computer**.

- it is a **free & open ISA**
 - **ISA** stands for **Instruction Set Architecture**
- RISC-V is a one of the ISA's family
 - RISC-V processors have following 5 different **Instruction Cycles (IC)**
 - 1.Fetch 2.decode 3.execute 4.Memory 5.writeback
 - **instruction cycle (IC)** consists of several steps, each performs a specific function in the execution of the instruction. The major steps in the instruction cycle are:
 - 1. Instruction Fetch (IF):** Read instruction from instruction memory.
 - In the instruction fetch cycle, the CPU retrieves the instruction from memory. The instruction then stored at the address specified by the program counter (PC). For every clock cycle, PC incremented by 4 (Byte aligned) to point next instruction in memory.
 - 2. Instruction Decode (ID):** Read program registers.
 - In the decode cycle, the CPU interprets the instruction and determines what operation needs to be performed. This involves identifying the opcode and any operands that are needed to execute the instruction. It basically decodes the instruction given by the Program Counter current address.
 - 3. Execute:** Compute value or address.

- In the execute cycle, the CPU **performs the operation specified by the instruction**. This may involve reading or writing data from or to memory, **performing arithmetic or logic operations on data**, or manipulating the control flow of the program.
- ex: Read **rs1 ([19:15]bits)** and **rs2 ([24:20]bits)** source registers (5 bits each) and write result into **rd ([11:7]bits)** target register. The [31:25] funct7 and [14:12] funct3 fields select the type of operation, finally the opcode for R-type instruction is specified at [6:0]bits.

4. Memory (Memory access): Read or write back data

- load and store instructions transfer a value between the registers and memory. (register ---> memory). LOAD is encoded in I type format where as the STORE is encoded in S type instruction format.

5. Write Back (WB): Write program registers.

- It is the final stage in execution of the instruction in RISC-V pipeline. During this stage, the result of instruction will be written back to the register file, that will be available for the future instructions.
- The control signals ensure that the correct register (destination register) is updated with the correct data.
- Example: ADD rd, rs1, rs2 In the WB stage, the result from the ALU is written back to the destination register rd.
- Example: LW rd, offset(rs1) In the WB stage, the data read from memory is written to the destination register rd.

6. PC: Update the program counter **address by 4** for the next instruction to be executed .

○ currently 4 base ISA's

- Each **base integer instruction set** is characterized by the **width of the integer registers** and the corresponding **size of the address space** and by the **number of integer registers**
- Two primary base integer variants - **1. RV32I** and **2. RV64I** - XLEN - refers width of an integer register in bits (either 32 or 64).
- 6 types of instruction formats (R/I/S/U/SB/UJ)
 - **R-format I-format S-format U-format SB-Format UJ-format**

- R stands for **Register**
- I stands for **immediate, loads**
- S refers to **store**
- U refers to **Upper immediate**
- B refers to **branch** type
- J refers to **Jump**

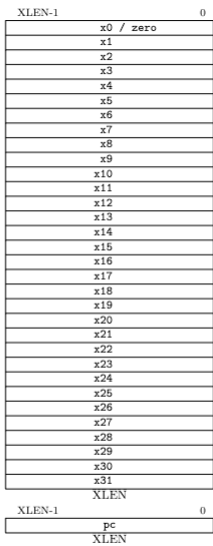
The 6 Instruction Formats

- **R-Format:** instructions using 3 register inputs
– `add, xor, mul` —arithmetic/logical ops
- **I-Format:** instructions with immediates, loads
– `addi, lw, jalr, slli`
- **S-Format:** store instructions: `sw, sb`
- **SB-Format:** branch instructions: `beq, bge`
- **U-Format:** instructions with upper immediates
– `lui, auipc` —upper immediate is 20-bits
- **UJ-Format:** jump instructions: `jal`

1. **User visible Base integer registers** - General-Purpose Register and PC

- there are **31 general-purpose registers x1–x31**, which hold integer values.
- Register x0 is hardwired to the constant 0.
- For **RV32 the x registers are 32 bits wide**, and for **RV64 they are 64 bits wide**
 - each RISC-V instruction = 32 bits = 1 Word = 4 Bytes

- we uses the term **XLEN** to refer to the current **width of an x register in bits** (either **32 or 64**).



- Figure 2.1: RISC-V user-level base integer register state.
- There is 1 additional user-visible register: **PC - program counter**
 - pc holds the **address of the current instruction**.

2. Base Instruction Format

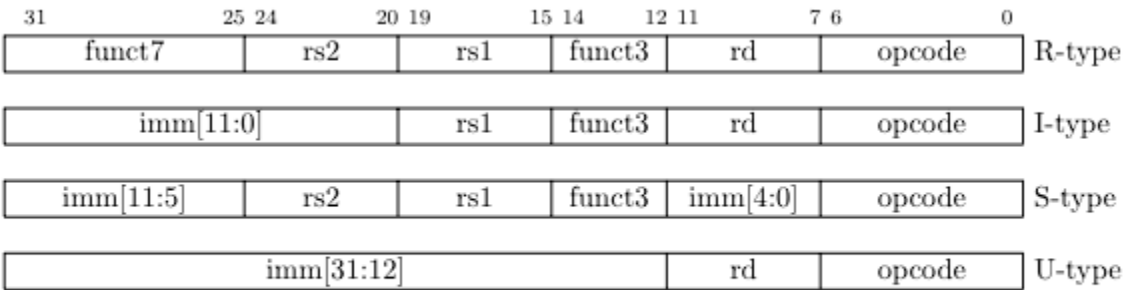


Figure 2.2: RISC-V base instruction formats.

32-bit RISC-V instruction formats

| Format | Bit | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--|------------|-----------|----|----|----|----|----|-----|-----|----|----|----|------|------------|----|----|----|--------|--------|----|----------|----------|---|--------|------|--------|---|---|---|---|---|---|
| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Register/register | funct7 | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | rd | | | opcode | | | | | | | | |
| Immediate | imm[11:0] | | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | opcode | | | | | | | | |
| Store | imm[11:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:0] | | | opcode | | | | | | | | |
| Branch | [12] | imm[10:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:1] | | | [11] | opcode | | | | | | |
| Upper immediate | imm[31:12] | | | | | | | | | | | | | | | | | | | | rd | | | opcode | | | | | | | | |
| Jump | [20] | imm[10:1] | | | | | | | | | | | [11] | imm[19:12] | | | | | | | rd | | | opcode | | | | | | | | |
| <ul style="list-style-type: none">• <i>opcode</i> (7 bits): Partially specifies one of the 6 types of <i>instruction formats</i>.• <i>funct7</i> (7 bits) and <i>funct3</i> (3 bits): These two fields extend the <i>opcode</i> field to specify the operation to be performed.• <i>rs1</i> (5 bits) and <i>rs2</i> (5 bits): Specify, by index, the first and second operand registers respectively (i.e., source registers).• <i>rd</i> (5 bits): Specifies, by index, the destination register to which the computation result will be directed. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

- All are fixed **32 bits in length** and must **4-byte boundary aligned** in memory
- The RISC-V ISA keeps the **source (rs1 and rs2) and destination (rd) registers** at the same position in all formats to simplify decoding.
- **Immediates** are packed towards the leftmost available bits in the instruction
- **sign bit** for all immediates always at the 31-bit of the instruction
- There are a further two variants of the instruction formats (SB/UJ) based on the handling of immediates, as shown in Figure 2.3.

| | 31 | 27 | 26 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 | |
|----|-----------------------|----|----|----|----|-----|-----|-----|--------|--------|----|-------------|--------|--------|--|
| R | funct7 | | | | | rs2 | | rs1 | | funct3 | | rd | | Opcode | |
| I | imm[11:0] | | | | | | rs1 | | funct3 | | rd | | Opcode | | |
| S | imm[11:5] | | | | | rs2 | | rs1 | | funct3 | | imm[4:0] | | opcode | |
| SB | imm[12 10:5] | | | | | rs2 | | rs1 | | funct3 | | imm[4:1 11] | | opcode | |
| U | imm[31:12] | | | | | | | | | | rd | | opcode | | |
| UJ | imm[20 10:1 11 19:12] | | | | | | | | | | rd | | opcode | | |

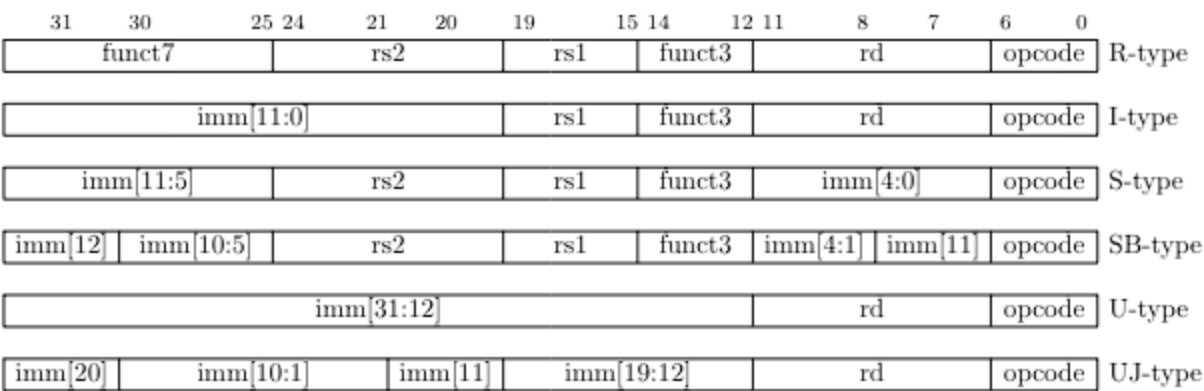


Figure 2.3: RISC-V base instruction formats showing immediate variants.

- In Figure 2.3 each immediate subfield is labeled with the bit position (**imm[x]**) in the immediate value being produced.
- The only difference between the S and SB formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the SB format.
 - Instead of shifting all bits in the instruction encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1])

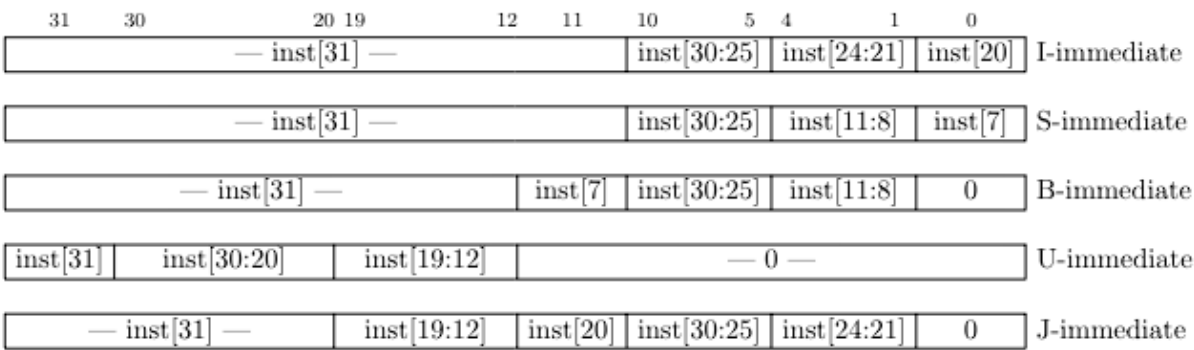


Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

- 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].
- and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in SB format.

- Similarly, the only difference between the U and UJ formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates.
 - The location of instruction bits in the U and UJ format immediates is chosen to maximize overlap with the other formats and with each other.

Now lets discuss each instruction formats (R/I/S/U/SB/UJ) in details below

1. R-format (Register) - Arithmetic and logical operations

| 31 | 25 24 | 20 19 | 15 14 | 12 11 | 7 6 | 0 |
|---------|-------|-------|--------------|-------|--------|---|
| funct7 | rs2 | rs1 | funct3 | rd | opcode | |
| 7 | 5 | 5 | 3 | 5 | 7 | |
| 0000000 | src2 | src1 | ADD/SLT/SLTU | dest | OP | |
| 0000000 | src2 | src1 | AND/OR/XOR | dest | OP | |
| 0000000 | src2 | src1 | SLL/SRL | dest | OP | |
| 0100000 | src2 | src1 | SUB/SRA | dest | OP | |

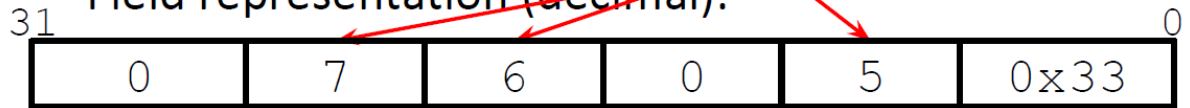
Figure5-1 R-type instruction machine code format

- Each field is viewed as its own unsigned int – 5-bit fields can represent any number 0-31, while 7-bit fields can represent any number 0-128, etc.
- **opcode [6:0]bits** (7 bits): partially specifies operation
 - e.g. R-types have opcode = 0b0110011=0x33, SB (branch) types have opcode = 0b1100011=0x63.
- funct7 [31:25]bits (7) + funct3 [14:12] (3) : total 10 bits. combined with opcode, these two fields describe what operation to perform.
 - How many R-instructions can we encode? Ans: with opcode fixed at 0x33, just funct varies: $(2^7) \times (2^3) = (2^{10}) = 1024$
- rs1 (5): 1st operand ("source register 1")
- rs2 (5): 2nd operand (second source register)
- rd (5): "destination register" — receives the result of computation
- lets take an instruction example **add x5,x6,x7** - as simple as **add rd,r1,r2**.
Instruction Code:

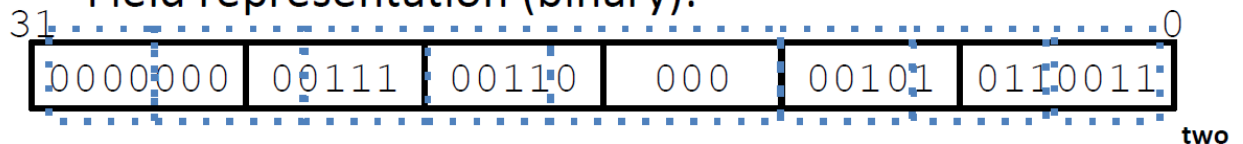
0x0073_02B3

- RISC-V Instruction: `add x5, x6, x7`

Field representation (decimal):



Field representation (binary):



hex representation: 0x 0073 02B3

decimal representation: 7,537,331

examples:-

1. `ADD r6, r2, r1` addition

- Opcode: 0110011
- Funct3: 000
- Funct7: 0000000
- rd: 00110 // represent as 5'b6
- rs1: 00001 // represent as 5'b1
- rs2: 00010 // represent as 5'b2
- Instruction Code: 0000000 00010 00001 000 00110 0110011

2. `SUB r7, r1, r2` subtraction

- Opcode: 0110011
- Funct3: 000
- Funct7: 0100000
- rd: 00111
- rs1: 00001
- rs2: 00010
- Instruction Code: 0100000 00010 00001 000 00111 0110011

3. `AND r8, r1, r3` AND operation between r1 & r3

- Opcode: 0110011
- Funct3: 111
- Funct7: 0000000
- rd: 01000
- rs1: 00001
- rs2: 00011
- Instruction Code: 0000000 00011 00001 111 01000 0110011

4. `OR r9, r2, r5` OR operation between r2 and r5

- Opcode: 0110011
- Funct3: 110
- Funct7: 0000000
- rd: 01001
- rs1: 00010
- rs2: 00101
- Instruction Code: 0000000 00101 00010 110 01001 0110011

5. XOR r10, r1, r4 XOR operation between r1 and r4

- Opcode: 0110011
- Funct3: 100
- Funct7: 0000000
- rd: 01010
- rs1: 00001
- rs2: 00100
- Instruction Code: 0000000 00100 00001 100 01010 0110011

6. SLT r11, r2, r4 r11=1 if x is negative; r11=0 if 0 or positive

- Opcode: 0110011
- Funct3: 010
- Funct7: 0000000
- rd: 01011
- rs1: 00010
- rs2: 00100
- Instruction Code: 0000000 00100 00010 010 01011 0110011

7. SRL r16, r14, r2

- Opcode: 0110011
- Funct3: 101
- Funct7: 0000000
- rd: 10000
- rs1: 01110
- rs2: 00010
- Instruction Code: `0000000 00010 01110 101 10000 0110011

8. SLL r15, r1, r2

- Opcode: 0110011
- Funct3: 001
- Funct7: 0000000
- rd: 01111
- rs1: 00001
- rs2: 00010
- Instruction Code: 0000000 00010 00001 001 01111 0110011

All RV32 R-format instructions

| | | | | | | |
|---------|-----|-----|-----|----|---------|------|
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |

Different encoding in funct7 + funct3 selects different operations

Arithmetic Operation

| Mnemonic | Instruction | Type | Description |
|----------------------|----------------------------------|------|-------------------------------------|
| ADD rd, rs1, rs2 | Add | R | $rd \leftarrow rs1 + rs2$ |
| SUB rd, rs1, rs2 | Subtract | R | $rd \leftarrow rs1 - rs2$ |
| ADDI rd, rs1, imm12 | Add immediate | I | $rd \leftarrow rs1 + imm12$ |
| SLT rd, rs1, rs2 | Set less than | R | $rd \leftarrow rs1 < rs2 ? 1 : 0$ |
| SLTI rd, rs1, imm12 | Set less than immediate | I | $rd \leftarrow rs1 < imm12 ? 1 : 0$ |
| SLTU rd, rs1, rs2 | Set less than unsigned | R | $rd \leftarrow rs1 < rs2 ? 1 : 0$ |
| SLTIU rd, rs1, imm12 | Set less than immediate unsigned | I | $rd \leftarrow rs1 < imm12 ? 1 : 0$ |
| LUI rd, imm20 | Load upper immediate | U | $rd \leftarrow imm20 \ll 12$ |
| AUIP rd, imm20 | Add upper immediate to PC | U | $rd \leftarrow PC + imm20 \ll 12$ |

Logical Operations

| Mnemonic | Instruction | Type | Description |
|---------------------|----------------------------------|------|--------------------------------------|
| AND rd, rs1, rs2 | AND | R | $rd \leftarrow rs1 \ \& \ rs2$ |
| OR rd, rs1, rs2 | OR | R | $rd \leftarrow rs1 \ \ rs2$ |
| XOR rd, rs1, rs2 | XOR | R | $rd \leftarrow rs1 \ \wedge \ rs2$ |
| ANDI rd, rs1, imm12 | AND immediate | I | $rd \leftarrow rs1 \ \& \ imm12$ |
| ORI rd, rs1, imm12 | OR immediate | I | $rd \leftarrow rs1 \ \ imm12$ |
| XORI rd, rs1, imm12 | XOR immediate | I | $rd \leftarrow rs1 \ \wedge \ imm12$ |
| SLL rd, rs1, rs2 | Shift left logical | R | $rd \leftarrow rs1 \ \ll \ rs2$ |
| SRL rd, rs1, rs2 | Shift right logical | R | $rd \leftarrow rs1 \ \gg \ rs2$ |
| SRA rd, rs1, rs2 | Shift right arithmetic | R | $rd \leftarrow rs1 \ \gg \ rs2$ |
| SLLI rd, rs1, shamt | Shift left logical immediate | I | $rd \leftarrow rs1 \ \ll \ shamt$ |
| SRLI rd, rs1, shamt | Shift right logical imm. | I | $rd \leftarrow rs1 \ \gg \ shamt$ |
| SRAI rd, rs1, shamt | Shift right arithmetic immediate | I | $rd \leftarrow rs1 \ \gg \ shamt$ |

2. I-format - (immediate, loads)

| 32-bit RISC-V instruction formats | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|-----------------------------------|------------|-----------|----|----|----|----|----|-----|-----|----|----|----|------|------------|----|----|----|--------|--------|----|----------|----------|---|---|--------|--------|---|---|---|---|---|---|
| Format | Bit | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Register/register | funct7 | | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | rd | | | | opcode | | | | | | |
| Immediate | imm[11:0] | | | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | | opcode | | | | | | |
| Upper immediate | imm[31:12] | | | | | | | | | | | | | | | | | | | | rd | | | | opcode | | | | | | | |
| Store | imm[11:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:0] | | | | opcode | | | | | | | |
| Branch | [12] | imm[10:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:1] | | | [11] | opcode | | | | | | |
| Jump | [20] | imm[10:1] | | | | | | | | | | | [11] | imm[19:12] | | | | | | | | rd | | | | opcode | | | | | | |

- **opcode (7 bits):** Partially specifies which of the 6 types of *instruction formats*.
- **funct7, and funct3 (10 bits):** These two fields, further than the *opcode* field, specify the operation to be performed.
- **rs1 (5 bits):** Specifies, by index, the register containing first operand (i.e., source register).
- **rs2 (5 bits):** Specifies the second operand register.
- **rd (5 bits):** Specifies the destination register to which the computation result will be directed.

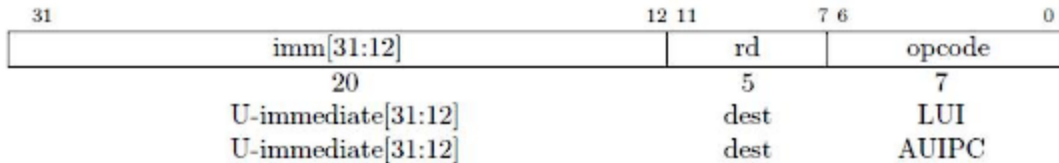


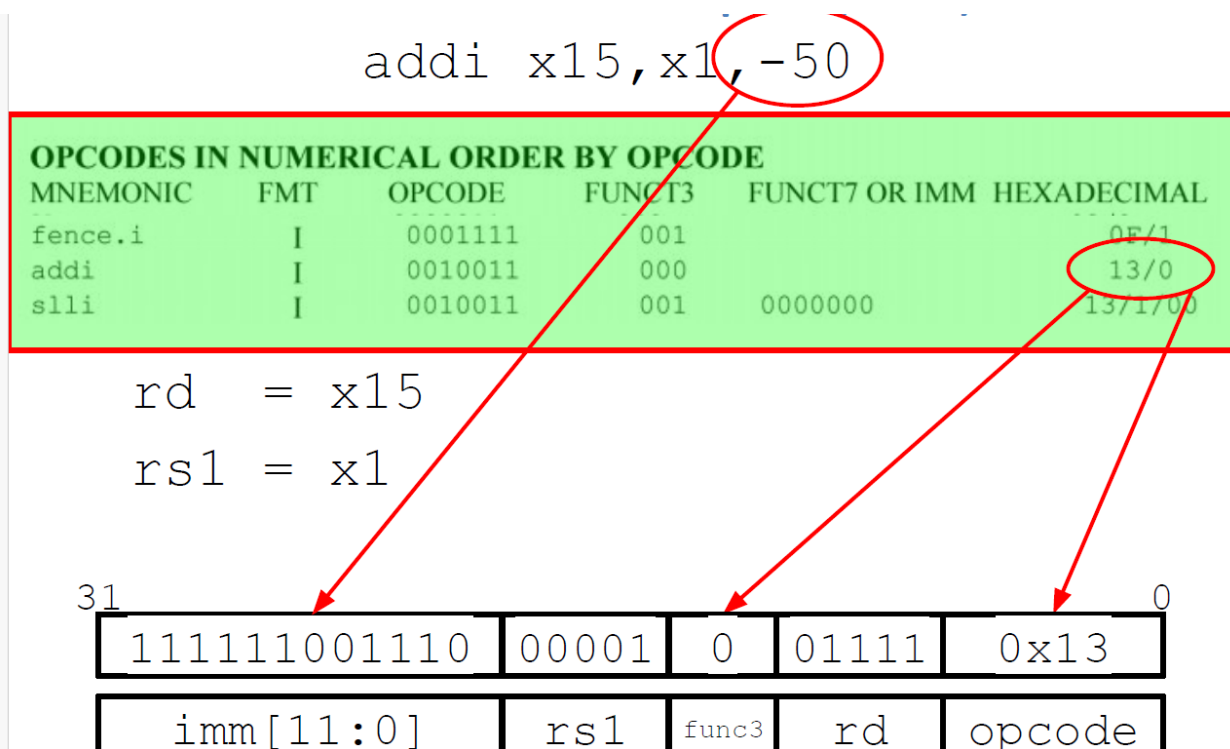
Figure4-1 U-type instruction machine code format

U-type instructions include opcode, rd, 20-bit immediate. As shown in Figure 4-1.

The upper 12 bits of I-type is an immediate number. The opcode is different from other instruction formats because the corresponding specific operations are different, and other parts are very similar to R-type

- In a RISC processor, access to memory is only done through special load and store instructions.
- These instructions come in a number of variants to be able to load values of different bit-size.
- The LD (Load Doubleword) is only supported on RV64I as it loads a 64-bit value.
- A word refers to a 32-bit value, so LW (Load Word) could be used to load a regular 32-bit integer. While working with strings you may want to load individual bytes by using LB.
- First notice that, if instruction has immediate, then it uses at most 2 registers (1 src, 1 dst).
- Key difference: Only **imm** field is different from R-format: rs2 and funct7 replaced by 12-bit signed immediate, mm[11:0]

- opcode (7): uniquely specifies the instruction
- rs1 (5): specifies a register operand
- rd (5): specifies destination register that receives result of computation
- immediate (12): 12 bit number - All computations done in words, so 12-bit immediate must be extended to 32 bits - always sign-extended to 32-bits before use in an arithmetic operation
- Can represent 2^{12} different immediates i.e., imm[11:0] can hold values in range $[-2^{11}, +2^{11})$
- example: RISC-V Instruction: **addi x15,x1,-50**



Field representation (binary):

31

111111001110

00001

000

01111

0010011

0

hex representation: 0x FCE0 8793

decimal representation: 4,242,573,203

two

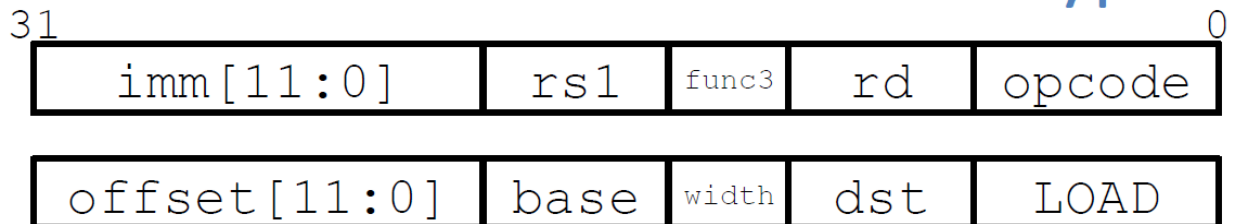
All RISC-V I-Type Arithmetic Instructions

| | | | | | | |
|-----------|-------|-----|-----|----|---------|-------|
| imm[11:0] | | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | ANDI |
| 0000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 | SRAI |

One of the higher-order immediate bits is used to distinguish “shift right logical” (SRLI) from “shift right arithmetic” (SRAI)

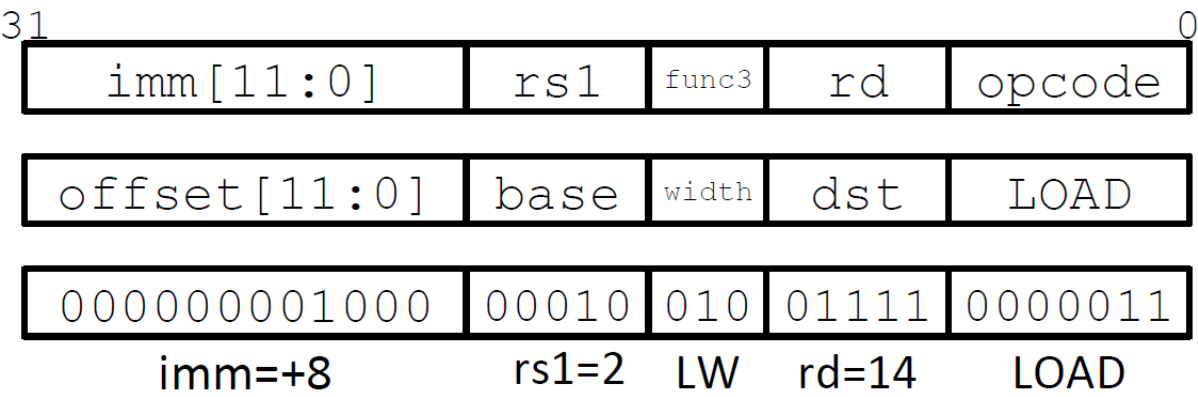
“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

- Load Instructions are also I-Type



- The 12-bit signed immediate is added to the base address in register **rs1** to form the memory address – This is very similar to the add-immediate operation but **used to create address, not to create final result** - Value loaded from memory is stored in rd - Example: **lw x14, 8(x2)**

- `lw x14, 8(x2)`



All RV32 Load Instructions

| | | | | | |
|-----------|-----|-----|----|---------|-----|
| imm[11:0] | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | rs1 | 001 | rd | 0000011 | LH |
| imm[11:0] | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:0] | rs1 | 101 | rd | 0000011 | LHU |

↑
func3 field encodes size and signedness of load data

- LBU is “load unsigned byte”
- LH is “load halfword”, which loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register
- LHU is “load unsigned halfword”, which zero-extends 16 bits to fill destination 32-bit register
- There is no LWU in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register

Load / Store Operations

| Mnemonic | Instruction | Type | Description |
|--------------------|------------------------|------|--|
| LD rd, imm12(rs1) | Load doubleword | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LW rd, imm12(rs1) | Load word | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LH rd, imm12(rs1) | Load halfword | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LB rd, imm12(rs1) | Load byte | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LWU rd, imm12(rs1) | Load word unsigned | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LHU rd, imm12(rs1) | Load halfword unsigned | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LBU rd, imm12(rs1) | Load byte unsigned | I | $rd \leftarrow mem[rs1 + imm12]$ |
| SD rs2, imm12(rs1) | Store doubleword | S | $rs2 \rightarrow mem[rs1 + imm12]$ |
| SW rs2, imm12(rs1) | Store word | S | $rs2(31:0) \rightarrow mem[rs1 + imm12]$ |
| SH rs2, imm12(rs1) | Store halfword | S | $rs2(15:0) \rightarrow mem[rs1 + imm12]$ |
| SB rs2, imm12(rs1) | Store byte | S | $rs2(7:0) \rightarrow em[rs1 + imm12]$ |

3. S-format (store)

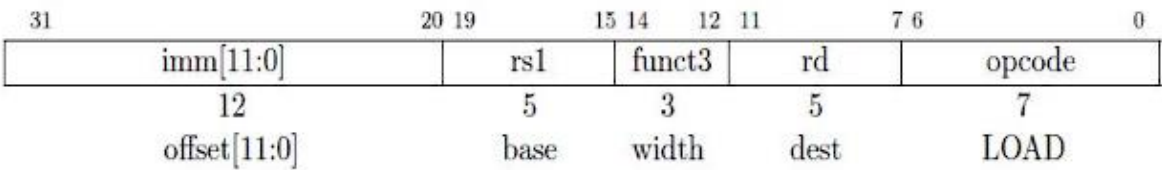


Figure8-1 LOAD instruction machine code format (I-type)

The instructions that mentioned before are all register instructions, and LOAD & STORE are instructions of the memory. In the load part, the value must be obtained from the memory, such as ROM, EPROM, EEROM, FLASH, DDR, SRAM, and other memory.

STORE

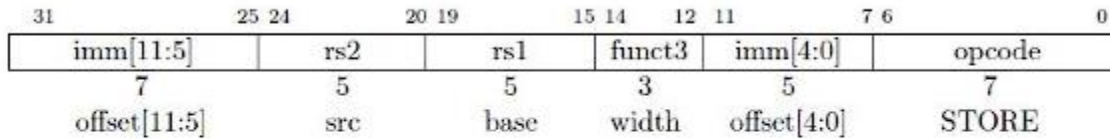


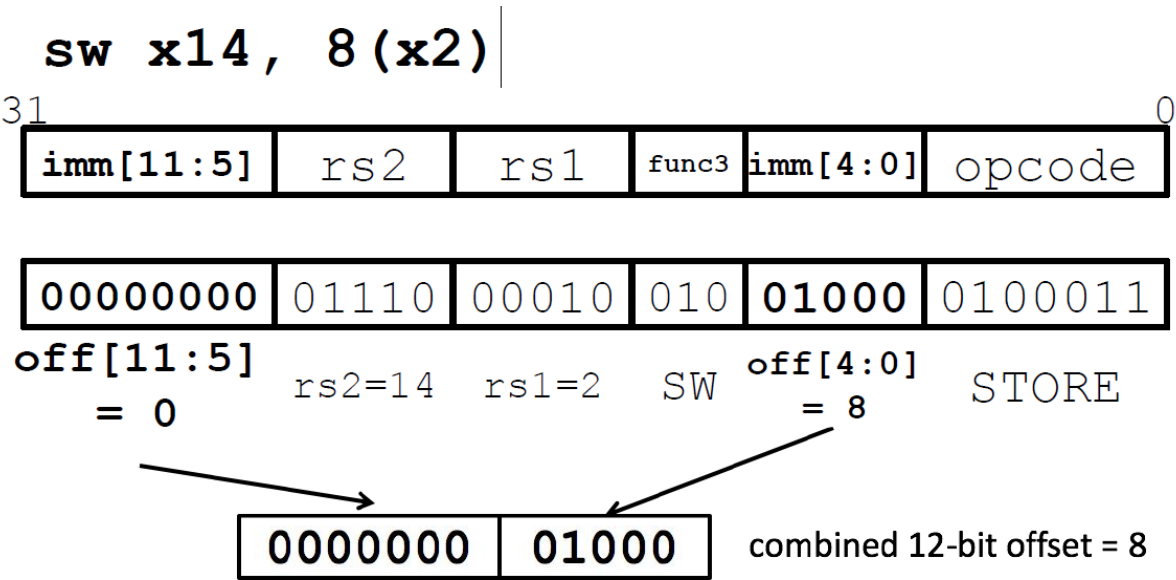
Figure8-2 S-type command format

The store instructions use S-type format. The store instruction writes the value to the memory, s-type does not have rd.

32-bit RISC-V instruction formats

| Format | Bit | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|------------|-----------|----|----|----|----|----|-----|-----|----|----|----|------|------------|----|----|----|--------|--------|----|----------|----------|---|----|--------|--------|---|--------|---|---|---|---|--|
| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Register/register | funct7 | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | rd | | | | opcode | | | | | | | | |
| Immediate | imm[11:0] | | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | | opcode | | | | | | | | |
| Upper immediate | imm[31:12] | | | | | | | | | | | | | | | | | | | | rd | | | | opcode | | | | | | | | |
| Store | imm[11:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:0] | | | | opcode | | | | | | | | |
| Branch | [12] | imm[10:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:1] | | | [11] | opcode | | | | | | | |
| Jump | [20] | imm[10:1] | | | | | | | | | | | [11] | imm[19:12] | | | | | | | | | | rd | | | | opcode | | | | | |
| <ul style="list-style-type: none">• opcode (7 bits): Partially specifies which of the 6 types of <i>instruction formats</i>.• funct7, and funct3 (10 bits): These two fields, further than the <i>opcode</i> field, specify the operation to be performed.• rs1 (5 bits): Specifies, by index, the register containing first operand (i.e., source register).• rs2 (5 bits): Specifies the second operand register.• rd (5 bits): Specifies the destination register to which the computation result will be directed. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

- The characteristic of S-type instruction is that **there is no rd register**.
- Store needs to read two registers, **rs1** for base memory address, and **rs2** for data to be stored, as well as need immediate offset!.
- Can't have both rs2 and immediate in same place as other instructions!
- Note: stores don't write a value to the register file, no rd!
- RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – keep rs1/rs2 fields in same place
- register names more critical than immediate bits in hardware design.
- Example: sw x14, 8(x2)



All RV32 Store Instructions

| | | | | | | |
|-----------|-----|-----|-----|----------|---------|----|
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |

Load / Store Operations

| Mnemonic | Instruction | Type | Description |
|--------------------|------------------------|------|--|
| LD rd, imm12(rs1) | Load doubleword | I | $rd \leftarrow \text{mem}[rs1 + \text{imm12}]$ |
| LW rd, imm12(rs1) | Load word | I | $rd \leftarrow \text{mem}[rs1 + \text{imm12}]$ |
| LH rd, imm12(rs1) | Load halfword | I | $rd \leftarrow \text{mem}[rs1 + \text{imm12}]$ |
| LB rd, imm12(rs1) | Load byte | I | $rd \leftarrow \text{mem}[rs1 + \text{imm12}]$ |
| LWU rd, imm12(rs1) | Load word unsigned | I | $rd \leftarrow \text{mem}[rs1 + \text{imm12}]$ |
| LHU rd, imm12(rs1) | Load halfword unsigned | I | $rd \leftarrow \text{mem}[rs1 + \text{imm12}]$ |
| LBU rd, imm12(rs1) | Load byte unsigned | I | $rd \leftarrow \text{mem}[rs1 + \text{imm12}]$ |
| SD rs2, imm12(rs1) | Store doubleword | S | $rs2 \rightarrow \text{mem}[rs1 + \text{imm12}]$ |
| SW rs2, imm12(rs1) | Store word | S | $rs2(31:0) \rightarrow \text{mem}[rs1 + \text{imm12}]$ |
| SH rs2, imm12(rs1) | Store halfword | S | $rs2(15:0) \rightarrow \text{mem}[rs1 + \text{imm12}]$ |
| SB rs2, imm12(rs1) | Store byte | S | $rs2(7:0) \rightarrow \text{em}[rs1 + \text{imm12}]$ |

4. U-format (Upper immediate)

Many of the arithmetic operations do sign extension, rd is destination register, rd and rs1 are original registers

Arithmetic Operation

| Mnemonic | Instruction | Type | Description |
|----------------------|----------------------------------|------|-------------------------------------|
| ADD rd, rs1, rs2 | Add | R | $rd \leftarrow rs1 + rs2$ |
| SUB rd, rs1, rs2 | Subtract | R | $rd \leftarrow rs1 - rs2$ |
| ADDI rd, rs1, imm12 | Add immediate | I | $rd \leftarrow rs1 + imm12$ |
| SLT rd, rs1, rs2 | Set less than | R | $rd \leftarrow rs1 < rs2 ? 1 : 0$ |
| SLTI rd, rs1, imm12 | Set less than immediate | I | $rd \leftarrow rs1 < imm12 ? 1 : 0$ |
| SLTU rd, rs1, rs2 | Set less than unsigned | R | $rd \leftarrow rs1 < rs2 ? 1 : 0$ |
| SLTIU rd, rs1, imm12 | Set less than immediate unsigned | I | $rd \leftarrow rs1 < imm12 ? 1 : 0$ |
| LUI rd, imm20 | Load upper immediate | U | $rd \leftarrow imm20 \ll 12$ |
| AUIP rd, imm20 | Add upper immediate to PC | U | $rd \leftarrow PC + imm20 \ll 12$ |

- **How do we deal with 32-bit immediates?**
 - Our I-type instructions only give us 12 bits
- **Solution:** Need a new instruction format for dealing with the rest of the 20 bits.
- This instruction should deal with:
 - a destination register to put the 20 bits into
 - the immediate of 20 bits
 - the instruction opcode

Upper Immediate Instructions format:

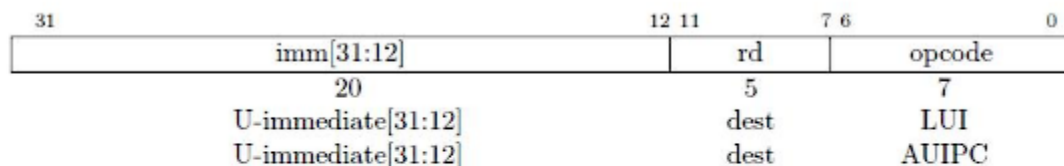


Figure4-1 U-type instruction machine code format

U-type instructions include opcode, rd, 20-bit immediate. As shown in Figure 4-1.

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
 - **LUI** – Load Upper Immediate
 - **AUIPC** – Add Upper Immediate to PC
- LUI to create long immediates:
 - lui writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
 - Together with an addi to set low 12 bits, can create any 32-bit value in a register using two instructions (lui/addi).
 - lui x10, 0x87654 # x10 = 0x87654000
 - addi x10, x10, 0x321 # x10 = 0x87654321
- **Corner Case:**
- How to set 0xDEADBEEF?
 - lui x10, 0xDEADB # x10 = 0xDEADB000
 - addi x10, x10, 0xEEF # x10 = 0xDEADAEFF
- addi 12-bit immediate is always sign-extended!
 - if top bit of the 12-bit immediate is a 1, it will subtract -1 from upper 20 bits

Solution

- How to set 0xDEADBEEF?

```
lui x10, 0xDEADC      # x10 = 0xDEADC000
addi x10, x10, 0xEEF   # x10 = 0xDEADBEEF
```

Pre-increment value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

Assembler pseudo-op handles all of this:

```
li x10, 0xDEADBEEF # Creates two instructions
```

- **AUIPC**: Add upper immediate value to PC
 - Adds upper immediate value to PC and places result in destination register
- sed for PC-relative addressing
 - **Label: auipc x10, 0**
 - Puts address of label into x10

SB-format (Branch)

Branching

| Mnemonic | Instruction | Type | Description |
|----------------------|---------------------------------------|------|---|
| BEQ rs1, rs2, imm12 | Branch equal | SB | if $rs1 == rs2$ $pc \leftarrow pc + imm12$ |
| BNE rs1, rs2, imm12 | Branch not equal | SB | if $rs1 != rs2$ $pc \leftarrow pc + imm12$ |
| BGE rs1, rs2, imm12 | Branch greater than or equal | SB | if $rs1 \geq rs2$ $pc \leftarrow pc + imm12$ |
| BGEU rs1, rs2, imm12 | Branch greater than or equal unsigned | SB | if $rs1 \geq rs2$ $pc \leftarrow pc + imm12$ |
| BLT rs1, rs2, imm12 | Branch less than | SB | if $rs1 < rs2$ $pc \leftarrow pc + imm12$ |
| BLTU rs1, rs2, imm12 | Branch less than unsigned | SB | if $rs1 < rs2$ $pc \leftarrow pc + imm12 \ll 1$ |
| JAL rd, imm20 | Jump and link | UJ | $rd \leftarrow pc + 4$ $pc \leftarrow pc + imm20$ |
| JALR rd, imm12(rs1) | Jump and link register | I | $rd \leftarrow pc + 4$ $pc \leftarrow rs1 + imm12$ |

| | | | | | | | | | |
|-----------------|-----------|-------|-------|---------|----------------|---------|--------|---|---|
| 31 | 30 | 25 24 | 20 19 | 15 14 | 12 11 | 8 | 7 | 6 | 0 |
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | | |
| 1 | 6 | 5 | 5 | 3 | 4 | 1 | 7 | | |
| offset[12,10:5] | | src2 | src1 | BEQ/BNE | offset[11,4:1] | | BRANCH | | |
| offset[12,10:5] | | src2 | src1 | BLT[U] | offset[11,4:1] | | BRANCH | | |
| offset[12,10:5] | | src2 | src1 | BGE[U] | offset[11,4:1] | | BRANCH | | |

Figure7-1 B-type Instruction machine code format

B-type instruction, B: branch instruction. These jump instructions are conditional.

There are 6 conditional jump instructions, as shown in Figure 7-1

Opcode==7'b110_0011

BEQ rs1,rs2,label //(imm)

funct3==3'b000

Imm is a 12-bit immediate, which is an integer multiple of 2, sign extended to 32bit

Branching Instructions:

- *beq, bne, bge, blt*
 - Need to specify an **address** to go to
 - Also take **two registers** to compare
 - Doesn't write into a register (similar to stores)
- How to encode label, i.e., where to branch to?

Branching Instruction Usage:

- Branches typically used for loops (if-else, while, for)
 - Loops are generally small (< 50 instructions)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
 - Largest branch distance limited by size of code
 - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing:

- PC-Relative Addressing: Use the immediate field as a two's complement offset to PC
 - Branches generally change the PC by a small amount
 - Can specify $\pm 2^{11}$ addresses from the PC
- Why not use byte address offset from PC as the immediate?

Branching Reach:

- Recall: RISC-V uses 32-bit addresses, and memory is byte-addressed
- Instructions are "word-aligned": Address is always a multiple of 4 (in bytes).
- PC ALWAYS points to an instruction
 - PC is typed as a pointer to a word
 - can do C-like pointer arithmetic
- **Let immediate specify #words instead of #bytes**
 - Instead of specifying $\pm 2^{11}$ bytes from the PC, we will now specify $\pm 2^{11}$ words = $\pm 2^{13}$ byte addresses around PC

Branch Calculation:

- If we **don't** take the branch:
 $PC = PC + 4$ = next instruction
- If we **do** take the branch:
 $PC = PC + (\text{immediate} * 4)$
- **Observations:**
 - *immediate* is number of instructions to move (remember, specifies words) either forward (+) or backwards (-)

RISC-V Feature, $n \times 16$ -bit instructions:

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length
- 16-bit = half-word
- **To enable this, RISC-V scales the branch offset to be half-words even when there are no 16-bit instructions.**
- Reduces branch reach by half and means that $\frac{1}{2}$ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class)
- RISC-V conditional branches can only reach $\pm 2^{10} \times 32$ -bit instructions either side of PC.

RISC-V B-Format for Branches:

- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents values -2^{12} to $+2^{12}-2$ in 2-byte increments
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

| | | | | | | | | | |
|-----------------|-----------|-------|-------|---------|----------------|---------|--------|---|---|
| 31 | 30 | 25 24 | 20 19 | 15 14 | 12 11 | 8 | 7 | 6 | 0 |
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | | |
| 1 | 6 | 5 | 5 | 3 | 4 | 1 | 7 | | |
| offset[12,10:5] | | src2 | src1 | BEQ/BNE | offset[11,4:1] | | BRANCH | | |
| offset[12,10:5] | | src2 | src1 | BLT[U] | offset[11,4:1] | | BRANCH | | |
| offset[12,10:5] | | src2 | src1 | BGE[U] | offset[11,4:1] | | BRANCH | | |

Figure7-1 B-type Instruction machine code format

Branch Example (1/2)

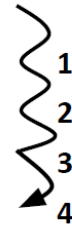
- RISCV Code:

```

Loop: beq  x19,x10,End
      add  x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:   <target instr>

```

Start counting from
instruction AFTER the
branch



- Branch offset = $4 \times 32\text{-bit instructions} = 16 \text{ bytes}$
- (Branch with offset of 0, branches to itself)

Branch Example (1/2)

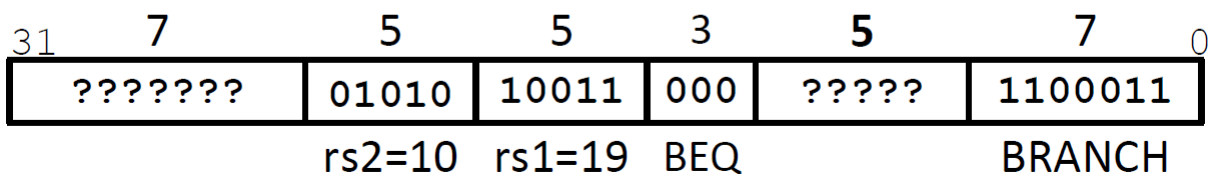
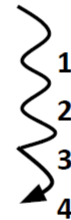
- RISCV Code:

```

Loop: beq  x19,x10,End
      add  x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:   <target instr>

```

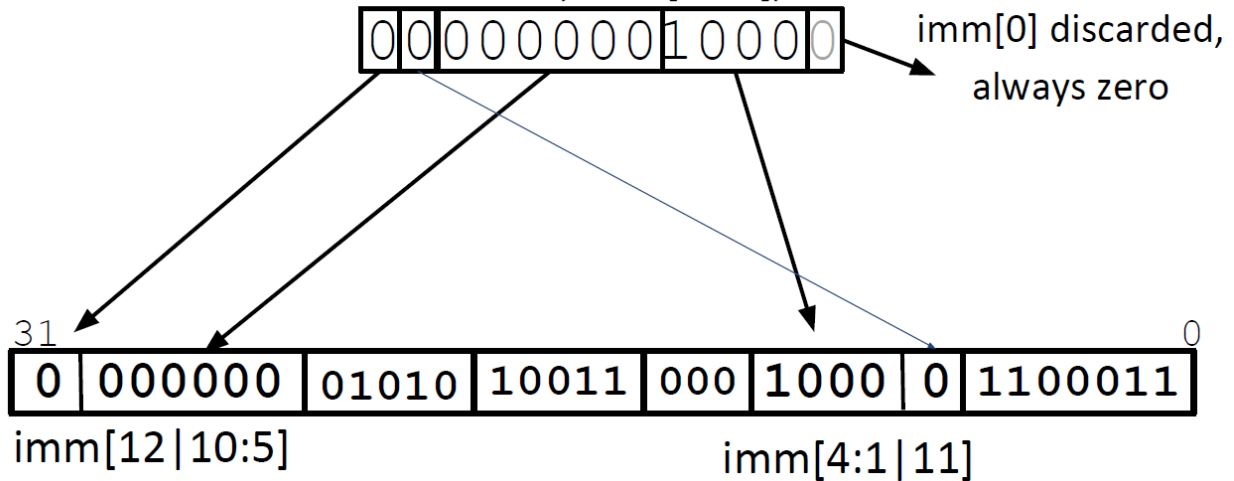
Start counting from
instruction AFTER the
branch



Branch Example (1/2)

beq x19,x10,offset = 16 bytes

13-bit immediate, imm[12:0], with value 16



RISC-V Immediate Encoding

- Why is it so confusing?!?!

| Instruction Encodings, inst[31:0] | | | | | | | | | | | | | | | | | | |
|-----------------------------------|----|-----------|----|-----|-----|-----|-----|--------|--------|--------|----------|----------|--------|---------|--------|--------|--|--------|
| 31 | 30 | 25 | 24 | 21 | 20 | 19 | 15 | 14 | 12 | 11 | 8 | 7 | 6 | 0 | | | | |
| funct7 | | | | rs2 | | | rs1 | | funct3 | | rd | | | opcode | | R-type | | |
| imm[11:0] | | | | | | rs1 | | funct3 | | rd | | | opcode | | I-type | | | |
| imm[11:5] | | | | rs2 | | | rs1 | | funct3 | | imm[4:0] | | | opcode | | S-type | | |
| imm[12] | | imm[10:5] | | | rs2 | | | rs1 | | funct3 | | imm[4:1] | | imm[11] | | opcode | | B-type |

| 32-bit immediates produced, imm[31:0] | | | | | | | | | | | |
|---------------------------------------|----|----|----|----|----|-------------|---|---|-------------|---|--|
| 31 | 30 | 20 | 19 | 12 | 11 | 10 | 5 | 4 | 1 | 0 | |
| — inst[31] — | | | | | | inst[30:25] | | | inst[24:21] | | |
| | | | | | | | | | inst[20] | | |
| | | | | | | | | | | | |
| — inst[31] — | | | | | | inst[30:25] | | | inst[11:8] | | |
| | | | | | | | | | inst[7] | | |
| | | | | | | | | | | | |
| — inst[31] — | | | | | | inst[30:25] | | | inst[11:8] | | |
| | | | | | | | | | 0 | | |
| | | | | | | | | | | | |

Upper bits sign-extended from inst[31] always

Only bit 7 of instruction changes role in immediate between S and B

All RISC-V Branch Instructions

| | | | | | | |
|--------------|-----|-----|-----|-------------|---------|------|
| imm[12 10:5] | rs2 | rs1 | 000 | imm[4:1 11] | 1100011 | BEQ |
| imm[12 10:5] | rs2 | rs1 | 001 | imm[4:1 11] | 1100011 | BNE |
| imm[12 10:5] | rs2 | rs1 | 100 | imm[4:1 11] | 1100011 | BLT |
| imm[12 10:5] | rs2 | rs1 | 101 | imm[4:1 11] | 1100011 | BGE |
| imm[12 10:5] | rs2 | rs1 | 110 | imm[4:1 11] | 1100011 | BLTU |
| imm[12 10:5] | rs2 | rs1 | 111 | imm[4:1 11] | 1100011 | BGEU |

Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
 - If moving individual lines of code, then yes
 - If moving all of code, then no (why?)
- What do we do if destination is $> 2^{10}$ instructions away from branch?
 - Other instructions save us:

```

beq x10,x0,far          bne x10,x0,next
# next instr           →   j    far
                        next: # next instr

```

6. UJ-format (jump)

Subroutine calls, jumps (UJ), and branches (SB)

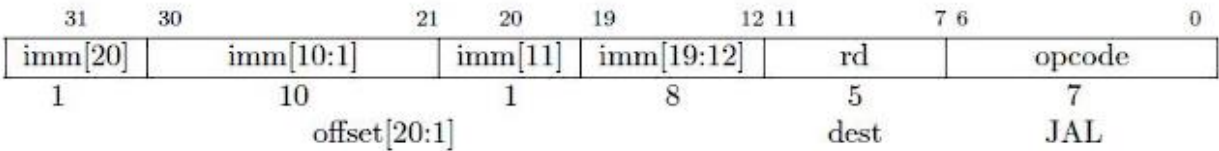


Figure6-1 J-type instruction machine code format

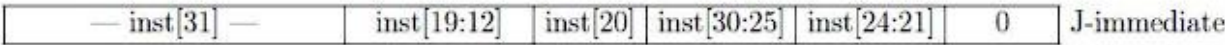
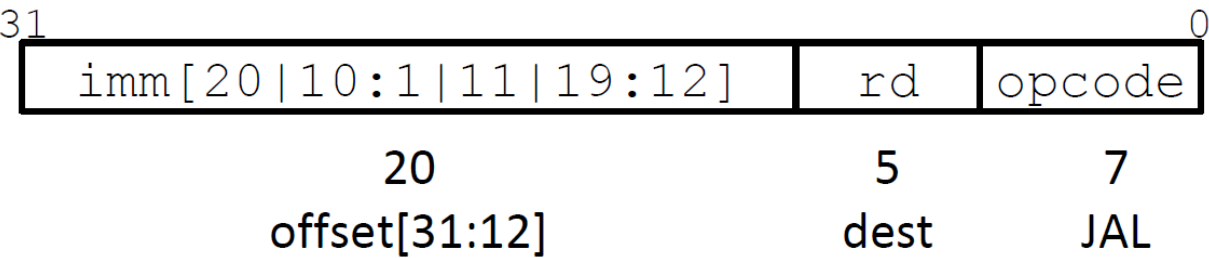


Figure6-2 J-type diagram of immediate recovery

JAL jump and link
JAL rd, label
opcode==7' b110_1111
Unconditional jump and link instructions
Use J-type format

JJ-Format Instructions:

- For branches, we assumed that we won't want to branch too far, so we can specify a **change** in the PC
- For general jumps (**jal**), we may jump to **anywhere** in code memory
 - Ideally, we would specify a 32-bit memory address to jump to
 - Unfortunately, we can't fit both a 7-bit *opcode* and a 32-bit address into a single 32-bit word
 - Also, when linking we must write to an **rd** register



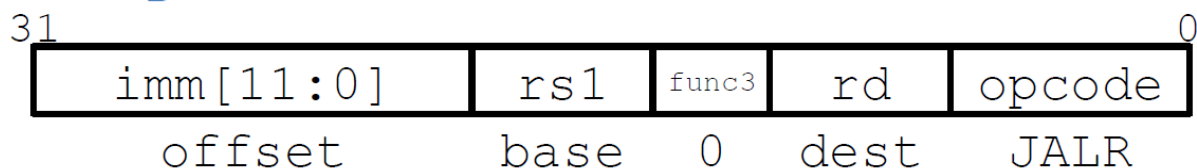
- jal saves **PC+4** in register **rd** (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
- $\pm 2^{18}$ 32-bit instructions

- **Reminder:** "j" jump is a pseudo-instruction—the assembler will instead use jal but sets rd=x0 to discard return address.
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost
- **j pseudo-instruction**
- j Label = jal x0, Label # Discard return address
- **Call function within 2¹⁸ instructions of PC**
- **jal ra, FuncName**
- Why is the immediate so funky?
 - Similar reasoning as for branch immediates

Branching

| Mnemonic | Instruction | Type | Description |
|----------------------|---------------------------------------|------|--------------------------------------|
| BEQ rs1, rs2, imm12 | Branch equal | SB | if rs1 == rs2 PC ← PC + imm12 |
| BNE rs1, rs2, imm12 | Branch not equal | SB | if rs1 != rs2 PC ← PC + imm12 |
| BGE rs1, rs2, imm12 | Branch greater than or equal | SB | if rs1 >= rs2 PC ← PC + imm12 |
| BGEU rs1, rs2, imm12 | Branch greater than or equal unsigned | SB | if rs1 >= rs2 PC ← PC + imm12 |
| BLT rs1, rs2, imm12 | Branch less than | SB | if rs1 < rs2 PC ← PC + imm12 |
| BLTU rs1, rs2, imm12 | Branch less than unsigned | SB | if rs1 < rs2 PC ← PC + imm12 << 1 |
| JAL rd, imm20 | Jump and link | UJ | rd ← PC + 4 PC ← PC + imm20 |
| JALR rd, imm12(rs1) | Jump and link register | I | rd ← PC + 4 PC ← rs1 + imm12 |

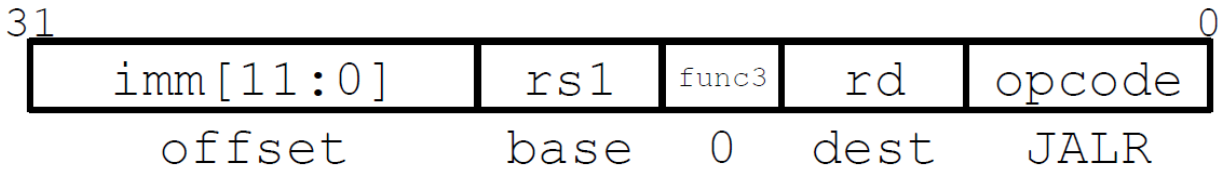
jalr Instruction (I-Format):



- **jalr rd, rs1, offset**
- Writes PC+4 to rd (return address)

- Sets PC = rs1 + offset
- Uses same immediates as arithmetic & loads
 - **no** multiplication by 2 bytes

Uses of jalr:



- ret and jr psuedo-instructions
 - ret = jr ra = jalr x0, ra, 0
- Call function at any 32-bit absolute address
 - lui x1, <hi 20 bits>
 - jalr ra, x1, <lo 12 bits>
- Jump PC-relative with 32-bit offset
 - auipc x1, <hi 20 bits> jalr x0, x1, <lo 12 bits>

Question: When combining two C files into one executable, we can compile them independently and then merge them

together.

Question: When combining two C files into one executable, we can compile them independently and then merge them together.

When merging two or more binaries:

- 1) **Jump** instructions don't require any changes
- 2) **Branch** instructions don't require any changes

| | 1 | 2 |
|-----|---|---|
| (A) | F | F |
| (B) | F | T |
| (C) | T | F |
| (D) | T | T |

RISC-V ISA cheatsheet

RISC-V Instruction-Set

Erik Enghelm <erik.enghelm@gmail.com>

Arithmetic Operation

| Mnemonic | Instruction | Type | Description |
|----------------------|----------------------------------|------|-------------------------------------|
| ADD rd, rs1, rs2 | Add | R | $rd \leftarrow rs1 + rs2$ |
| SUB rd, rs1, rs2 | Subtract | R | $rd \leftarrow rs1 - rs2$ |
| ADDI rd, rs1, imm12 | Add Immediate | I | $rd \leftarrow rs1 + imm12$ |
| SLT rd, rs1, rs2 | Set less than | R | $rd \leftarrow rs1 < rs2 ? 1 : 0$ |
| SLTI rd, rs1, imm12 | Set less than immediate | I | $rd \leftarrow rs1 < imm12 ? 1 : 0$ |
| SLTU rd, rs1, rs2 | Set less than unsigned | R | $rd \leftarrow rs1 < rs2 ? 1 : 0$ |
| SLTIU rd, rs1, imm12 | Set less than immediate unsigned | I | $rd \leftarrow rs1 < imm12 ? 1 : 0$ |
| LUI rd, imm20 | Load upper immediate | U | $rd \leftarrow imm20 \ll 12$ |
| AUIP rd, imm20 | Add upper immediate to PC | U | $rd \leftarrow PC + imm20 \ll 12$ |

Logical Operations

| Mnemonic | Instruction | Type | Description |
|---------------------|----------------------------------|------|----------------------------------|
| AND rd, rs1, rs2 | AND | R | $rd \leftarrow rs1 \& rs2$ |
| OR rd, rs1, rs2 | OR | R | $rd \leftarrow rs1 rs2$ |
| XOR rd, rs1, rs2 | XOR | R | $rd \leftarrow rs1 \wedge rs2$ |
| ANDI rd, rs1, imm12 | AND Immediate | I | $rd \leftarrow rs1 \& imm12$ |
| ORI rd, rs1, imm12 | OR Immediate | I | $rd \leftarrow rs1 imm12$ |
| XORI rd, rs1, imm12 | XOR Immediate | I | $rd \leftarrow rs1 \wedge imm12$ |
| SLL rd, rs1, rs2 | Shift left logical | R | $rd \leftarrow rs1 \ll rs2$ |
| SRL rd, rs1, rs2 | Shift right logical | R | $rd \leftarrow rs1 \gg rs2$ |
| SRA rd, rs1, rs2 | Shift right arithmetic | R | $rd \leftarrow rs1 \gg rs2$ |
| SLLI rd, rs1, shamt | Shift left logical immediate | I | $rd \leftarrow rs1 \ll shamt$ |
| SRLI rd, rs1, shamt | Shift right logical imm. | I | $rd \leftarrow rs1 \gg shamt$ |
| SRAI rd, rs1, shamt | Shift right arithmetic immediate | I | $rd \leftarrow rs1 \gg shamt$ |

Load / Store Operations

| Mnemonic | Instruction | Type | Description |
|--------------------|------------------------|------|---|
| LD rd, imm12(rs1) | Load doubleword | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LD rd, imm12(rs1) | Load word | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LH rd, imm12(rs1) | Load halfword | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LB rd, imm12(rs1) | Load byte | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LDU rd, imm12(rs1) | Load word unsigned | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LHU rd, imm12(rs1) | Load halfword unsigned | I | $rd \leftarrow mem[rs1 + imm12]$ |
| LBU rd, imm12(rs1) | Load byte unsigned | I | $rd \leftarrow mem[rs1 + imm12]$ |
| SD rs2, imm12(rs1) | Store doubleword | S | $rs2 \leftarrow mem[rs1 + imm12]$ |
| SW rs2, imm12(rs1) | Store word | S | $rs2(31:0) \leftarrow mem[rs1 + imm12]$ |
| SH rs2, imm12(rs1) | Store halfword | S | $rs2(15:0) \leftarrow mem[rs1 + imm12]$ |
| SB rs2, imm12(rs1) | Store byte | S | $rs2(7:0) \leftarrow mem[rs1 + imm12]$ |

Pseudo Instructions

| Mnemonic | Instruction | Base Instruction(s) |
|-----------------------|--------------------------------|---|
| LI rd, imm12 | Load Immediate (near) | ADDI rd, zero, imm12 |
| LI rd, imm | Load Immediate (far) | LUI rd, imm[31:12] ADDI rd, rd, imm[11:0] |
| LA rd, sym | Load address (far) | AUIP rd, sym[31:12] ADDI rd, rd, sym[11:0] |
| MV rd, rs | Copy register | ADDI rd, rs, 0 |
| NOT rd, rs | One's complement | XORI rd, rs, -1 |
| NEG rd, rs | Two's complement | SUB rd, zero, rs |
| BGT rs1, rs2, offset | Branch if rs1 > rs2 | BLT rs2, rs1, offset |
| BLE rs1, rs2, offset | Branch if rs1 ≤ rs2 | BGE rs2, rs1, offset |
| BGTU rs1, rs2, offset | Branch if rs1 > rs2 (unsigned) | BLTU rs2, rs1, offset |
| BLEU rs1, rs2, offset | Branch if rs1 ≤ rs2 (unsigned) | BGEU rs2, rs1, offset |
| BEQ rs1, offset | Branch if rs1 = 0 | BEQ rs1, zero, offset |
| BNE rs1, offset | Branch if rs1 ≠ 0 | BNE rs1, zero, offset |
| BGEZ rs1, offset | Branch if rs1 ≥ 0 | BGE rs1, zero, offset |
| BLEZ rs1, offset | Branch if rs1 ≤ 0 | BGE zero, rs1, offset |
| BGTZ rs1, offset | Branch if rs1 > 0 | BLT zero, rs1, offset |
| J offset | Unconditional jump | JAL zero, offset |
| CALL offset12 | Call subroutine (near) | JALR ra, ra, offset12 |
| CALL offset | Call subroutine (far) | AUIP ra, offset[31:12] JALR ra, ra, offset[11:0] |
| RET | Return from subroutine | JALR zero, 0(ra) |
| NOP | No operation | ADDI zero, zero, 0 |

Branching

| Mnemonic | Instruction | Type | Description |
|----------------------|---------------------------------------|------|---|
| BEQ rs1, rs2, imm12 | Branch equal | SB | if $rs1 = rs2$ $PC \leftarrow PC + imm12$ |
| BNE rs1, rs2, imm12 | Branch not equal | SB | if $rs1 \neq rs2$ $PC \leftarrow PC + imm12$ |
| BGE rs1, rs2, imm12 | Branch greater than or equal | SB | if $rs1 \geq rs2$ $PC \leftarrow PC + imm12$ |
| BGEU rs1, rs2, imm12 | Branch greater than or equal unsigned | SB | if $rs1 \geq rs2$ $PC \leftarrow PC + imm12$ |
| BLT rs1, rs2, imm12 | Branch less than | SB | if $rs1 < rs2$ $PC \leftarrow PC + imm12$ |
| BLTU rs1, rs2, imm12 | Branch less than unsigned | SB | if $rs1 < rs2$ $PC \leftarrow PC + imm12 \ll 1$ |
| JAL rd, imm20 | Jump and link | UJ | $rd \leftarrow PC + 4$ $PC \leftarrow PC + imm20$ |
| JALR rd, imm12(rs1) | Jump and link register | I | $rd \leftarrow PC + 4$ $PC \leftarrow rs1 + imm12$ |

Register File

| | | | |
|-----|-----|-----|-----|
| r0 | r1 | r2 | r3 |
| r4 | r5 | r6 | r7 |
| r8 | r9 | r10 | r11 |
| r12 | r13 | r14 | r15 |
| r16 | r17 | r18 | r19 |
| r20 | r21 | r22 | r23 |
| r24 | r25 | r26 | r27 |
| r28 | r29 | r30 | r31 |

Register Aliases

| | | | |
|-------|----|-----|-----|
| zero | ra | sp | gp |
| tp | t0 | t1 | t2 |
| s0/fp | s1 | a0 | a1 |
| a2 | a3 | a4 | a5 |
| a6 | a7 | s2 | s3 |
| s4 | s5 | s6 | s7 |
| s8 | s9 | s10 | s11 |
| t3 | t4 | t5 | t6 |

ra - return address
sp - stack pointer
gp - global pointer
tp - thread pointer

t0 - t6 - Temporary registers
s0 - s11 - Saved by callee
a0 - a7 - Function arguments
a0 - a1 - Return value(s)

32-bit instruction format

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|-----------|----|----|----|----|----|----|-----|-----|----|----|-----|-----|----|----|------|------|----|----|----|-----------|----|---|--------|--------|---|---|---|---|---|---|---|
| R | func | | | | | | | rs2 | | | | rs1 | | | | func | | | | rd | | | | opcode | | | | | | | | |
| | immediate | | | | | | | | | | | | rs1 | | | | func | | | | rd | | | | opcode | | | | | | | |
| SB | immediate | | | | | | | | rs2 | | | | rs1 | | | | func | | | | immediate | | | | opcode | | | | | | | |
| UJ | immediate | | | | | | | | | | | | | | | | | | | | rd | | | | opcode | | | | | | | |

Another useful thing to point out about this sheet is the pseudo instructions. They are not real instructions supported by RISC-V processors. Instead, they are just convenient ways of writing other instructions. For instance, if I want to move the value of one register say x3 to x4 then it would be nice if RISC-V had a move instruction. MV x4, x3 would accomplish this, except it doesn't really exist. Why? Because the same can be accomplished with:

- **ADDI x4, x3, 0** $\# x4 \leftarrow x3 + 0$

That means you can avoid adding encoding for an MV instruction to the instruction-set architecture (ISA).

One great example of the benefits of pseudo instructions is the LI and LA instructions. Because all RISC-V instructions must be 32-bit wide, they cannot contain a full 32-bit address. Thus loading a 32-

bit address into a register has to be done as a two-step process. First, we load the top 20 bits with either LUI or AUIP and then we add the remaining 12 bits with ADDI.

```
.section .text                # Mark code section
    LUI  a1,    %hi(msg)      # Load upper 20 bits of msg address
    ADDI a1, a1, %lo(msg)      # Load lower 12 bits of msg address
    CALL puts                 # Call puts function to show string
loop:
    J loop                    # Jump to loop - Infinite loop
.section .data                # Mark section for R/W data storage
msg: .string "Hello World\n"
```

To create code that can be loaded into any memory address (position independent code) we use the LA instruction which translated into AUIP and ADDI.

By using pseudo-instructions we greatly simplify this code:

```
.section .text                # Mark code section
    LI  a1, msg                # Load immediate. Julia expands to
                                # multiple instructions as needed.
    CALL puts                 # Call puts function to show string
loop:
    J loop                    # Jump to loop - Infinite loop
.section .data                # Mark section for R/W data storage
msg: .string "Hello World\n"
```

| Mnemonic | Instruction | Base instruction(s) |
|-----------------------|-------------------------------------|--|
| LI rd, imm12 | Load immediate (near) | ADDI rd, zero, imm12 |
| LI rd, imm | Load immediate (far) | LUI rd, imm[31:12] ADDI rd, rd, imm[11:0] |
| LA rd, sym | Load address (far) | AUIPC rd, sym[31:12] ADDI rd, rd, sym[11:0] |
| MV rd, rs | Copy register | ADDI rd, rs, 0 |
| NOT rd, rs | One's complement | XORI rd, rs, -1 |
| NEG rd, rs | Two's complement | SUB rd, zero, rs |
| BGT rs1, rs2, offset | Branch if $rs1 > rs2$ | BLT rs2, rs1, offset |
| BLE rs1, rs2, offset | Branch if $rs1 \leq rs2$ | BGE rs2, rs1, offset |
| BGTU rs1, rs2, offset | Branch if $rs1 > rs2$ (unsigned) | BLTU rs2, rs1, offset |
| BLEU rs1, rs2, offset | Branch if $rs1 \leq rs2$ (unsigned) | BGEU rs2, rs1, offset |
| BEQZ rs1, offset | Branch if $rs1 = 0$ | BEQ rs1, zero, offset |
| BNEZ rs1, offset | Branch if $rs1 \neq 0$ | BNE rs1, zero, offset |
| BGEZ rs1, offset | Branch if $rs1 \geq 0$ | BGE rs1, zero, offset |
| BLEZ rs1, offset | Branch if $rs1 \leq 0$ | BGE zero, rs1, offset |
| BGTZ rs1, offset | Branch if $rs1 > 0$ | BLT zero, rs1, offset |
| J offset | Unconditional jump | JAL zero, offset |
| CALL offset12 | Call subroutine (near) | JALR ra, ra, offset12 |
| CALL offset | Call subroutine (far) | AUIPC ra, offset[31:12] JALR ra, ra, offset[11:0] |
| RET | Return from subroutine | JALR zero, 0(ra) |
| NOP | No operation | ADDI zero, zero, 0 |