Advanced computer architecture

GROUP 6: E/20/280, E/20/279, E/20/254

Activity 1

Instructions

All the instructions in the RISC-V architecture can be divided into 6 sections

Base Integer Instruction Set (I)

- **I**: The mandatory core of RISC-V, providing essential instructions for:
    - Integer arithmetic and logic operations
    - Control flow (e.g., branches and jumps)
    - Memory access (e.g., loads and stores)

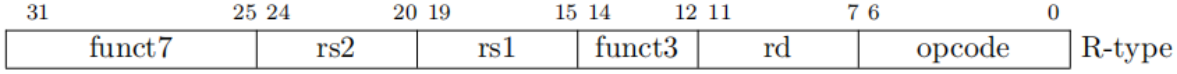
Optional Extensions

- **M (Integer Multiplication and Division Extension)**
    - Adds instructions for efficient integer multiplication and division.
    - Commonly used in applications that require arithmetic-intensive computations.
- **A (Atomic Instructions Extension)**
    - Introduces atomic memory operations for synchronization in multi-threaded or parallel environments.
    - Key instructions include **Load-Reserved (LR)** and **Store-Conditional (SC)**.
- **F (Single-Precision Floating-Point Extension)**
    - Provides support for 32-bit single-precision floating-point operations (IEEE 754 standard).
    - Used in applications requiring floating-point arithmetic.
- **D (Double-Precision Floating-Point Extension)**
    - Adds support for 64-bit double-precision floating-point operations (IEEE 754 standard).
    - Requires the **F** extension as a prerequisite.
- **C (Compressed Instructions Extension)**
    - Introduces 16-bit instructions alongside the standard 32-bit instructions.
    - Reduces program size, saving memory and improving performance in memory-constrained environments.

Here we only implement the IM set only.

Instructions (divided by format)

1. R-type 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

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

for basic I instructions ;

rs1 – source 1

rs2 – source 2

for M ;

rs1 – Multiplicity

rs2 – Multiplier

For any;

rd – destination

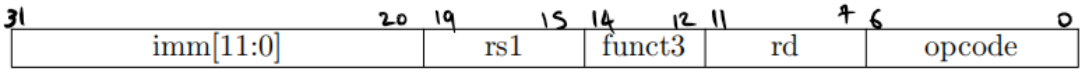
ADD: rd=rs1+rs2

SUB: rd=rs1−rs2

SLL: rd=rs1<<rs2[shamt] shift
SLT: rd=(rs1<rs2)(signed) store
SLTU: rd=(rs1<rs2) (unsigned) store
XOR: rd=rs1⊕rs2
SRL: rd=rs1>>rs2[shamt] (logical)
SRA: rd=rs1>>rs2[shamt] (arithmetic)(sign extend)
OR: rd=rs1∨rs2
AND: rd=rs1∧rs2

MUL: rd=(rs1×rs2)[31:0]
MULH: Upper 32 bits of signed rs1×rs2
MULHSU: Upper 32 bits of signed rs1×unsigned rs2
MULHU: Upper 32 bits of unsigned rs1×rs2
DIV: Signed rd=rs1/rs2
DIVU: Unsigned rd=rs1/rs2
REM: Signed remainder rd=rs1%rs2
REMU: Unsigned remainder rd=rs1%rs2

2. I-type instructions



imm[11:0]	rs1	funct3	rd	opcode	I-type

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

imm[11:0]	rs1	000	rd	1100111	JALR

- Imm – offset
- Rs1 – Base

Add rs1 register value to **sign extended** (from 12 bits to 32 bits) Imm field to get the mem address

LB - load 8 bit (byte) from memory and sign extend to 32 bits and store in **rd**
LH - Load 16 bit(half word) mem, sign extend to 32 bit and store in **rd**
LW - load 32 word to **rd**
LBU - load unsigned 8 bit (byte) value and **Zero extend to 32** and store in **rd**
LHU - load 16 bit (half word) value and **Zero extend to 32** and store in **rd**

JALR - (jump and link register) get target address by - sign extended Imm + rs1
Set least significand bit to **Zero** (from calculated target)
Write PC+4 (before jump PC value) to **rd**

Use x0(reg 0) as rd if no need to store

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

- Imm – immediate
- rs1 – source

ADDI - add **rs1** and Sign extended **IMM** , and store in **rd**
SLTI - (set less than immediate) set **rd** = 1 if **rs1 < imm**(signed extended) **(both treated as signed values)**

If not, rd = 0

SLTIU - (set less than immediate **unsigned**) same as above but values treated as unsigned.
XORI - perform XOR(bitwise) on **rs1 and imm**(sign extended) and store result in **rd**
ORI - perform OR(bitwise) on **rs1 and imm**(sign extended) and store result in **rd**
ANDI - perform AND(bitwise) on **rs1 and imm**(sign extended) and store result in **rd**

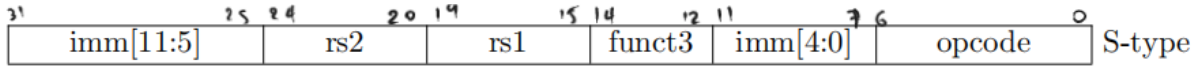
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

- rs1 – source
- Imm divided into two. (func7 and shamt)
- Shamt – shift amount

SLLI - logical shift left of the value in **rs1** by **shamt** (5 least significant bits from the **imm)** new left bits are set to **zeros.**

SRLI - logical shift right of the value in **rs1** by **shamt** (5 least significant bits from the **imm)** filling the vacated bits on the left with **zeros**
SRAI - arithmetic shifts right, of the value in **rs1** by **shamt** (5 least significant bits from the **imm) maintaining the sign bit** (arithmetic shift)

3. S-type instructions



imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

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

- Imm – offset
- rs1 – source
- rs2 - base

Take rs1 + imm([31:25]+[11:7])(sign extended) (offset) as the target address, and **store the value in rs2 to the target**

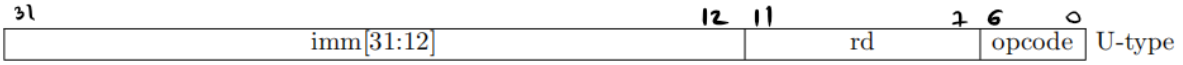
SB - store 8 low bits(byte) from rs2

SH - store 16 low bits(half word) from rs2

SW - store 32 low bits(word) from rs2

Note: when storing Byte or half word, the rest should stay unchanged(no zero or sign extension) (therefore need a control signal to MEM.)

4. U-type instructions



imm[31:12]	rd	opcode	U-type

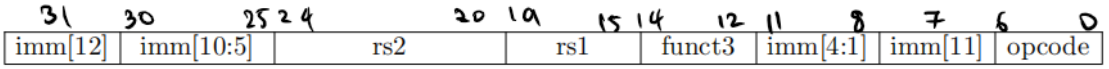
imm[31:12]	rd	0110111	LUI
imm[31:12]	rd	0010111	AUIPC

LUI - (load upper immediate) store the 20 bits from the **imm** to the **most significant 20 bits** of the 32 bits i**n rd(remaining** least significand **12 bits** are **filled with zero)**

AUIPC - (add upper immediate to **PC**) store the result of;

PC + imm**(remaining** least significand **12 bits** are **filled with zero)**

5. B-type instructions



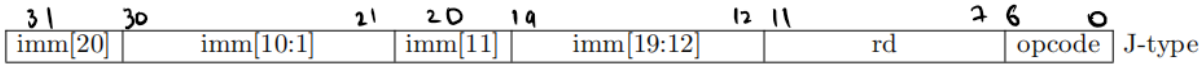
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

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

- Imm – offset
- rs1 – source1
- rs2 – source2

Immediate calculate by combining;
Imm[12:0] = [31] + [7] + [30:25] + [11:8] + 0 (word align)
Offset = sign extend imm[12:0]
BEQ - branch to PC+offset if **rs1 = rs2**
BNE - branch to PC+offset if **rs1 != rs2**
BLT - branch to PC+offset if **rs1 < rs2 (Signed comparison)**
BGE - branch to PC+offset if **rs1 >= rs2 (Signed comparison)**
BLTU - branch to PC+offset if **rs1 < rs2 (unSigned comparison)**
BGEU - branch to PC+offset if **rs1 >= rs2 (unSigned comparison)**

6. J-type instructions



imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	J-type

imm[20|10:1|11|19:12]	rd	1101111	JAL

Immidiate calculated by combining;

Imm[20:0] - [31] + [19:12] + [20] + [30:21] + 0 (word align)
Offset = sign extend imm[20:0]
JAL - jump to PC+offset , store PC + 4 to **rd**

Main units and it's functions

1. Program Counter (PC) - The Program Counter (PC) holds the address of the current instruction being executed. It is updated to point to the next instruction after every clock cycle unless modified by a branch or jump instruction.
    - **Inputs**:
        i. **RESET:** Resets the PC to an initial value, typically 0.
        ii. **CLK:** Clock signal to synchronize updates.
        iii. **MUX Output:** Determines the next PC value (either incremented, branch target, or jump target).
    - **Outputs** :
        i. **PC Address:** 32-bit address used to fetch the instruction from instruction memory.
    - **Example:**If PC = 0x00000004, the next instruction fetched from memory will be at address 0x00000004.

2. Instruction Memory - Stores the program's instructions and provides the current instruction to the CPU based on the PC value.
    - **Inputs**:
        i. **PC Address:** Specifies the memory location of the instruction to be fetched.
    - **Outputs**:
        i. **Instruction:** The 32-bit instruction fetched from the specified memory location.
    - **Example:**If the instruction at address 0x00000004 is `ADD x5, x6, x7`, the instruction memory outputs this binary encoding to be processed.

3. PC Adder - Calculates the address of the next sequential instruction by adding 4 to the current PC value (assuming 4-byte instructions).
    - **Inputs:**
        i. **PC:** Current instruction address.
    - **Outputs:**
        i. **PC + 4:** Address of the next sequential instruction.
    - **Example:** If PC = 0x00000008, PC Adder outputs 0x0000000C as the next address.
4. Control Unit - Decodes the fetched instruction and generates control signals to direct the flow of data and operations in the CPU.
    - **Inputs:**
        i. **Instruction[31:0]:** The fetched 32-bit instruction.

- **Outputs:**
    i. Various control signals like `ALU_OP`, `WRITE_REG`, `MEM_READ`, `MEM_WRITE`, `BRANCH`, and `JUMP`.
- **Example:**For an `ADD` instruction, the control unit sets `ALU_OP` to "Add" and enables the register write signal.

5. .Register File - Stores the CPU's general-purpose registers. Provides two source operands to the ALU and allows writing back the result.
    - **Inputs:**
        i. **RS1, RS2 (Register Addresses):** Specifies source registers.
        ii. **WRITE_ADDR:** Address of the destination register.
        iii. **WRITE_EN:** Enables write operation.
        iv. **WRITE_DATA:** Data to be written to the destination register.
    - **Outputs:**
        i. **OUT1, OUT2:** Data read from the source registers.
    - **Example:**For the instruction `ADD x1, x2, x3`, registers x2 and x3 are read, and their sum is written to x1.

6. ALU (Arithmetic Logic Unit) - Performs arithmetic and logical operations on the input operands.
    - **Inputs:**
        i. **ALU_OP:** Specifies the operation (e.g., Add, Subtract, AND, OR).
        ii. **Operand 1, Operand 2:** Data inputs for the operation.
    - **Outputs**:
        i. **Result:** Computed result of the operation.
        ii. **Zero Flag:** Indicates whether the result is zero (used for branch conditions).
    - **Example**:For an `ADD` instruction, if Operand 1 = 5 and Operand 2 = 3, the ALU outputs 8.

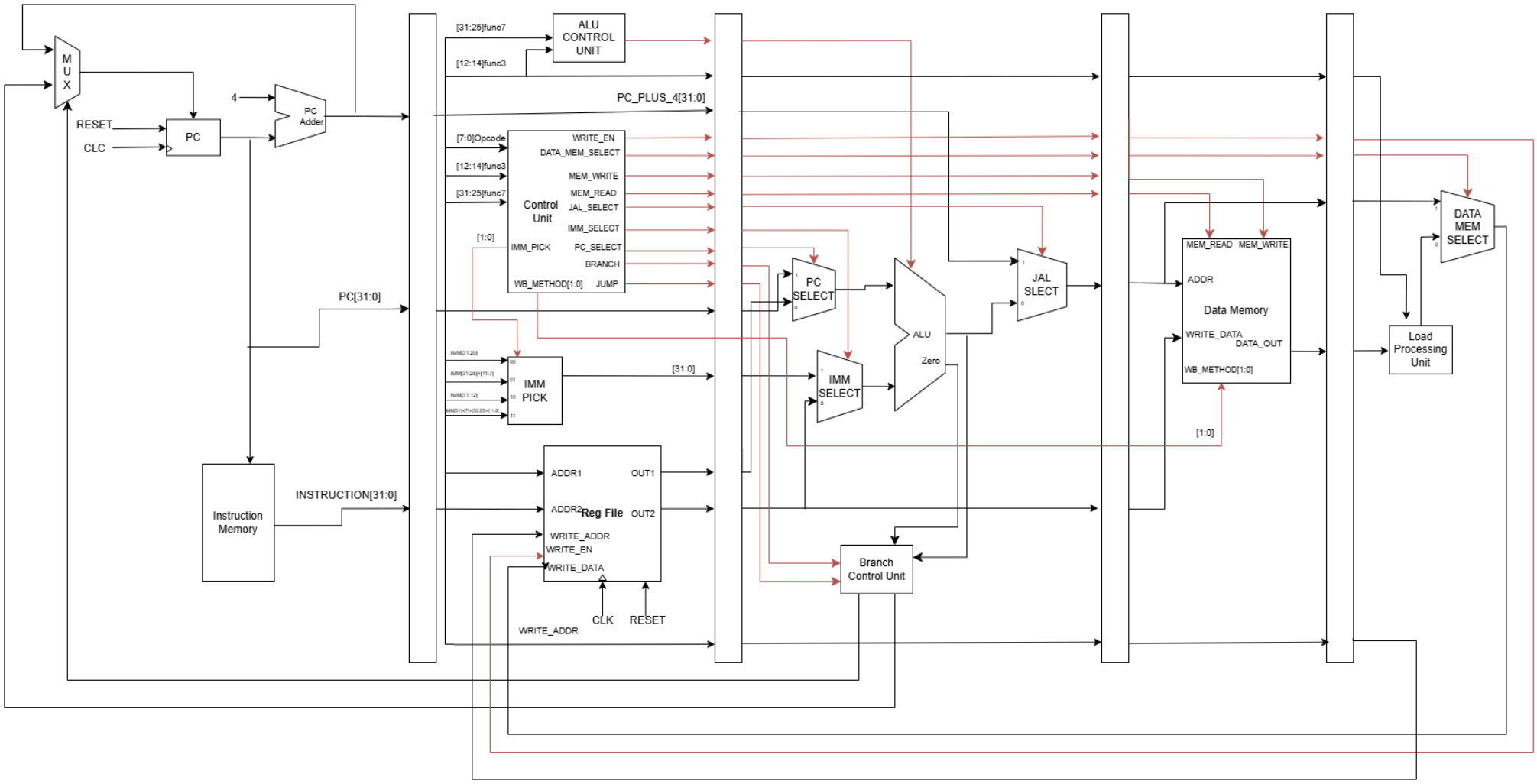
7. Data Memory - Stores data values and provides data read/write functionality for load/store instructions.
    - **Inputs:**
        i. **ADDR:** Address to read/write data.
        ii. **WRITE_DATA:** Data to be written to memory.
        iii. **MEM_READ, MEM_WRITE:** Control signals to enable reading or writing.
    - **Outputs:**

    i. **DATA_OUT:** Data read from memory.
  - **Example:** For a `LOAD` instruction at address 0x00000010, Data Memory outputs the value stored at that address.
8. Branch Control Unit - Determines the next PC value for branch instructions based on the branch condition and the zero flag from the ALU.
  - **Inputs:**
    i. **BRANCH:** Control signal indicating a branch instruction.
    ii. **Zero Flag:** Output from the ALU.
    iii. **Branch Target Address:** Calculated branch target.
  - **Outputs:**
    i. **PC (Updated):** Address of the branch target or next sequential instruction.
  - **Example:** For a `BEQ` (branch if equal) instruction, if Zero Flag = 1, the PC is updated to the branch target address.

9. Multiplexers (MUX) - Used throughout the architecture to select between multiple inputs based on control signals.
  - **Inputs:**
    i. **Data Inputs:** Multiple inputs to choose from.
    ii. **Control Signal:** Determines the selected input.
  - **Outputs:**
    i. **Selected Input:** Output based on the control signal.
  - **Example:** For a branch instruction, the MUX selects between `PC + 4` and the branch target address based on the branch condition.

10. Clock (CLK) - Provides a timing signal to synchronize operations across all components.
  - **Example:** The PC is updated at every rising edge of the clock.

11. Load Processing Unit(LPU) - when writing back to the register from MEM, select between LB, LH, LW, LBU, LHU.


M
U
X
RESET
CLC
PC
4
PC
Adder
PC[31:0]
Instruction
Memory
INSTRUCTION[31:0]
[31:25]func7
[12:14]func3
ALU
CONTROL
UNIT
PC_PLUS_4[31:0]
[7:0]Opcode
[12:14]func3
[31:25]func7
Control
Unit
WRITE_EN
DATA_MEM_SELECT
MEM_WRITE
MEM_READ
JAL_SELECT
IMM_SELECT
PC_SELECT
BRANCH
JUMP
IMM_PICK
WB_METHOD[1:0]
[1:0]
IMM
PICK
[31:0]
PC
SELECT
IMM
SELECT
ALU
Zero
JAL
SLECT
MEM_READ
MEM_WRITE
ADDR
Data Memory
WRITE_DATA
DATA_OUT
WB_METHOD[1:0]
[1:0]
DATA
MEM
SELECT
Load
Processing
Unit
ADDR1
ADDR2
Reg File
OUT1
OUT2
WRITE_ADDR
WRITE_EN
WRITE_DATA
CLK
RESET
WRITE_ADDR
Branch
Control Unit