Building a Pipelined Processor
RISC V

Part 01

RV32IM Instructions
And

Pipeline Datapath

Group members:

E/18/180 - Kodituwakku M.K.N.M
E/18/266 - Premathilaka K.N.I
E/18/368 - Uduwanage H.U

1. Instruction Encoding Formats

RV32IM is a specific variant of the RISC-V ISA that includes the base 32-bit instruction set architecture along with the integer extension and the multiplication and division extension.

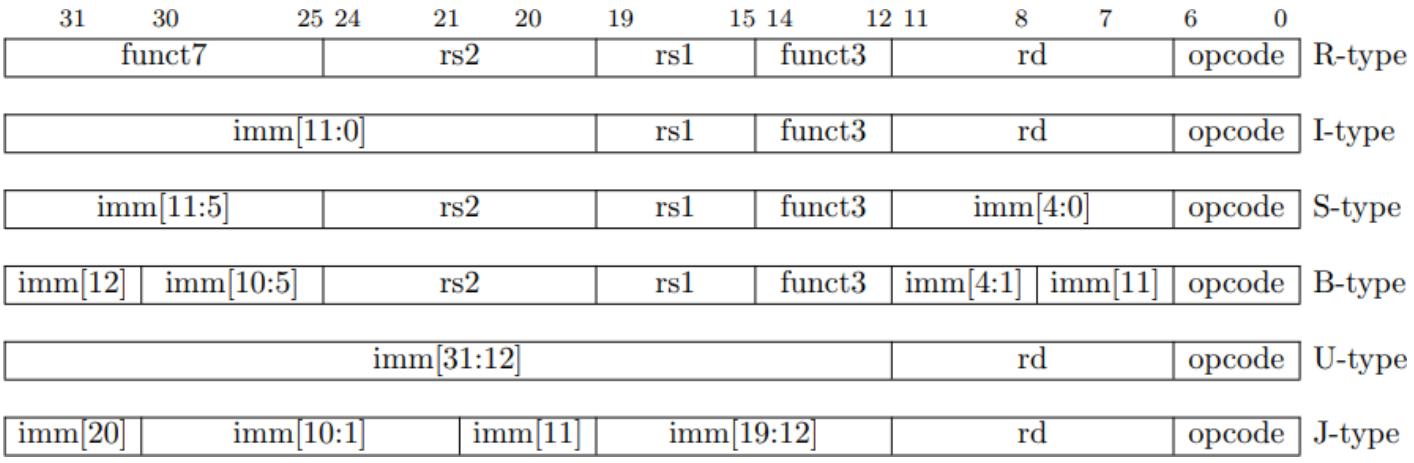
The RISC-V instruction set includes a range of instruction types, including:

1. R-Type: This instruction type is used for arithmetic and logical operations, such as add, subtract, AND, OR, etc. The R-Type instruction in RISC-V has a fixed format that includes the opcode field, three register fields for source operands, and a register field for the destination operand.
2. I-Type: This instruction type is used for immediate operations, such as addi (add immediate), lw (load word), sw (store word), etc. The I-Type instruction in RISC-V has a fixed format that includes the opcode field, a register field for the destination operand, a register field for the source operand, and a 12-bit immediate value field.
3. S-Type: This instruction type is used for store operations, such as sb (store byte), sh (store halfword), and sw (store word). The S-Type instruction in RISC-V has a fixed format that includes the opcode field, two register fields for the source operands, and a 12-bit immediate value field.
4. U-Type: This instruction type is used for unconditional jump operations, such as jal (jump and link). The U-Type instruction in RISC-V has a fixed format that includes the opcode field and a 20-bit immediate value field.

In summary, while the instruction types R-Type, I-Type, S-Type, and U-Type are commonly used in computer architecture, their exact formats and meanings may vary between different ISAs, including RISC-V.

S-Type and U-Type instructions are further classified based on how the immediate value is handled. Figure 1 depicts the RISC-V ISA instruction forms.

Encoding Formats



2. Description of instruction & Encoding formats

2.1 Integer Computational Instructions

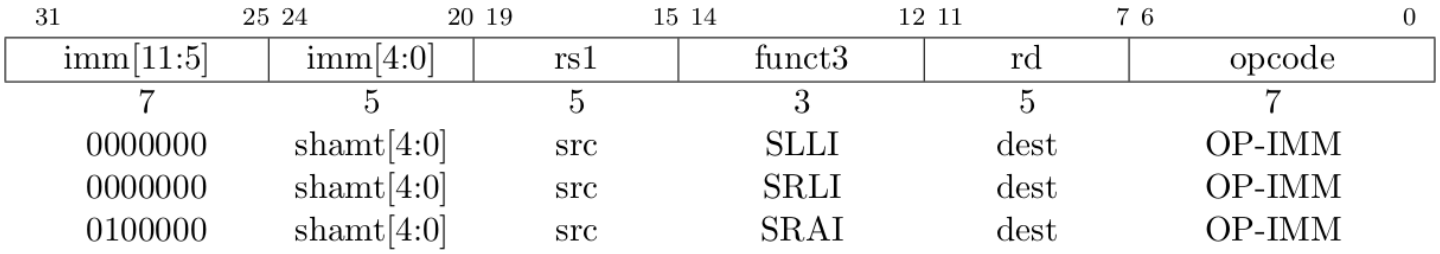
2.1.1. Register - Immediate Instructions

The register-immediate instructions are a type of instruction that operate on an immediate value and a register. These instructions allow data to be loaded into a register or arithmetic operations to be performed on a register using an immediate value.

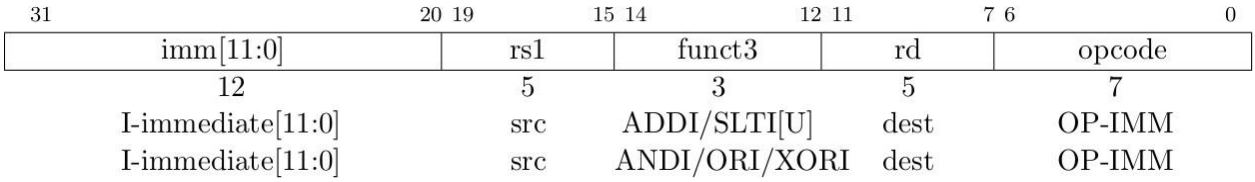
Instruction	Instruction Type	Instruction Name	Example	Description
ADDI	I - Type	Add immediate Unsigned	ADDI x5, x7, 7	Reg[x5] <- Reg[x7] + 7
SLTI	I - Type	Set less than Immediate	SLTI x1, x2, 3	if (Regs[x2] > 3): Regs[x1] <- 1 else: Regs[x1] <- 0 (Signed Comparison)
SLTIU	I - Type	Set less than Immediate Unsigned	SLTI x1, x2, 3	if (Regs[x2] > 3): Regs[x1] <- 1 else: Regs[x1] <- 0 (Unsigned Comparison)
ANDI	I - Type	Logical AND immediate	ANDI x1, x2, 3	Reg[x1] <- Reg[x2] & 3
ORI	I - Type	Logical OR immediate	ORI x1, x2, 3	Reg[x1] <- Reg[x2] | 3
XORI	I - Type	Logical XOR immediate	XORI x1, x2, 3	Reg[x1] <- Reg[x2] ^ 3
SLLI	I - Type	Logical Left shift Immediate	SLLI x1, x2, 3	Reg[x1] <- Reg[x2] << 3

SRLI	I - Type	Logical right Shift Immediate	SRLI x1, x2, 3	Reg[x1] <- Reg[x2] >> 3 (zeros are shifted into the upper bits)
SRAI	I - Type	Arithmetic Right Shift	SRAI x1, x2, 3	Reg[x1] <- Reg[x2] >> 3 (Original Sign bit is copied into the vacated upper bits)
LUI	U - Type	Load Upper Immediate	LUI x1, 3	Reg[x1] <- $[3]_{20\ bits}$ $[0]_{12\ bits}$ immediate value in the top 20 bits & lowest 12 bits with zeros
AUIPC	U - Type	Add upper immediate to PC	ALUPC x1, 3	Reg[x1]<-PC+$[[3]_{20\ bits}[0]_{12}]$

Encoding Formats



31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

2.1.2. Register - Register Instructions

The register-register instructions are a type of instruction that operate on two registers. These instructions allow data to be transferred between registers or arithmetic operations to be performed on two registers.

Instruction	Instruction Type	Instruction Name	Example	Description
ADD	R - Type	Addition	ADD x1, x2, x3	Reg[x1] <- Reg[x2] + Reg[x3]
SLT	R - Type	Set less than	SLT x1, x2, x3	if (Regs[x2]>Regs[x3]): Regs[x1]<-1 else: Regs[x1] <- 0 (Signed Comparison)
SLTU	R - Type	Set less than unsigned	SLTU x1, x2, x3	if (Regs[x2] > Regs[x3]): Regs[x1] <- 1 else: Regs[x1] <- 0 (Signed Comparison)
AND	R - Type	Bitwise logical AND	AND x1, x2, x3	Reg[x1] <- Reg[x2] & Regs[x3]
OR	R - Type	Bitwise logical OR	OR x1, x2, x3	Reg[x1] <- Reg[x2] | Regs[x3]
XOR	R - Type	Bitwise logical XOR	XOR x1, x2, x3	Reg[x1] <- Reg[x2] ^ Regs[x3]
SLL	R - Type	Logical Left Shift	SLL x1, x2, x3	Reg[x1] <- Reg[x2] << Regs[x3]
SRL	R - Type	Logical Right Shift	SRL x1, x2, x3	Reg[x1] <- Reg[x2] >> 3 (zeros are shifted into the upper bits)
SUB	R - Type	Subtraction	SUB x1, x2, x3	Reg[x1] <- Reg[x2] - Reg[x3]
SRA	R - Type	Arithmetic Right shift	SRA x1,x2,x3	Reg[x1] <- Reg[x2] >> Reg[x3] (Original Sign bit is copied into the vacated upper bits)

Encoding Formats

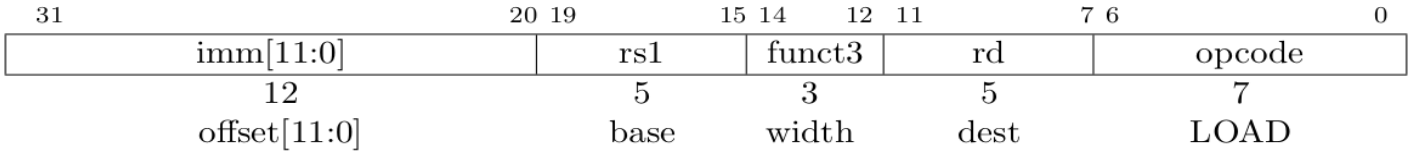
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

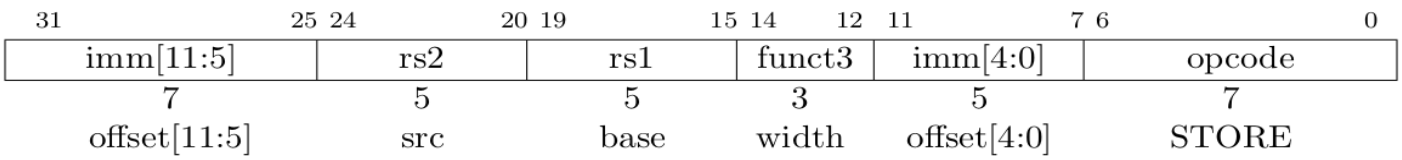
2.2 Load and Store Instructions

Load and store instructions are used to move data between the processor's registers and memory. These instructions are fundamental for accessing data stored in memory, and they are used in almost all programs.

Instruction	Instruction Type	Instruction Name	Example	Description
LW	I - Type	Load Word(32bit)	Lw x1, 60(x2)	Regs[x1]<- Mem[60+Regs[x2]]$_{32\ bit}$
LH	I - Type	Load Half word (16 bit)	LH x1, 60(x2)	Regs[x1]<- [Mem[60+Regs[x2]]]$_{16\ bit}$ Sign extend to 32 bits
LHU	I - Type	Load half word Unsigned	LHU x1, 60(x2)	Regs[x1]<- [Mem[60+Regs[x2]]]$_{16\ bit}$ Zero extend to 32 bits
LB	I - Type	Load Byte (8 bit)	LB x1, 60(x2)	Regs[x1]<- [Mem[60+Regs[x2]]]$_{8\ bit}$ Sign extend to 32 bits
LBU	I - Type	Load Byte Unsigned	LBU x1, 60(x2)	Regs[x1]<- [Mem[60+Regs[x2]]]$_{8\ bit}$ Zero extend to 32 bits
SW	S - Type	Store Word	SW x1, 60(x2)	[Mem[60+Regs[x2]]] <- Reg[x1]$_{32\ bit}$
SH	S - Type	Store Half word	SH x1, 60(x2)	[Mem[60+Regs[x2]]] <- Reg[x1]$_{16\ bit}$
SB	S - Type	Store Byte	SB x1, 60(x2)	[Mem[60+Regs[x2]]] <- Reg[x1]$_{8\ bit}$

Encoding Formats



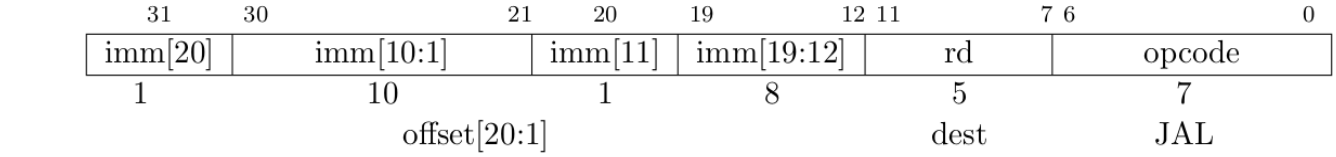


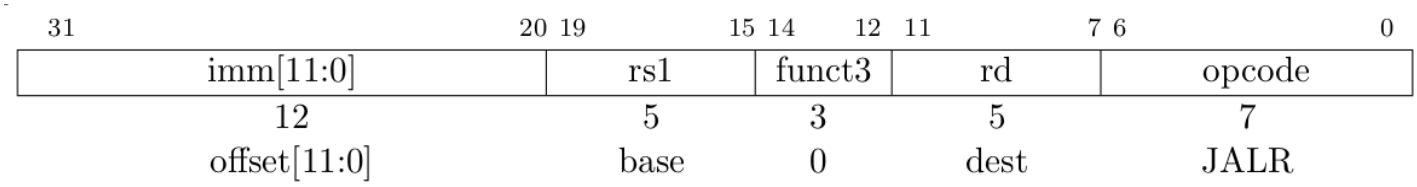
2.3 Control Transfer Instructions

These instructions can be used to perform conditional and unconditional jumps, call and return from subroutines, and handle exceptions.In the RV32IM instruction set architecture (ISA), control transfer instructions are used to transfer control from one part of a program to another.

Instruction	Instruction Type	Instruction Name	Example	Description
JAL	J - Type	Jump and Link	JAL x1, offset	Regs[x1] <- PC+4; PC <- PC + (offset << 1)
JALR	I - Type	Jump and link register	JALR x1, x2,offset	Regs[x1] <- PC+4; PC<-Regs[x2]+(offset<<1)
BEQ	B - Type	Branch Equal	BEQ x1, x2,offset	if(Regs[x1]== Regs[x2]): PC<-PC+(offset<< 1)
BNE	B - Type	Branch Not Equal	BNE x1, x2,offset	if(Regs[x1] != Regs[x2]): PC <- PC+(offset<< 1)
BLT	B - Type	Branch Less Than	BLT x1, x2,offset	if(Regs[x1] < Regs[x2]): PC <- PC+(offset<< 1)
BLTU	B - Type	Branch Less Than unsigned	BLTU x1, x2,offset	if(Regs[x1] < Regs[x2]): PC <- PC+(offset<< 1) (Unsigned comparison)
BGE	B - Type	Branch Greater than or equal	BGE x1, x2,offset	if(Regs[x1] > Regs[x2]): PC <- PC+ (offset<< 1)
BGEU	B - Type	Branch Greater than or Equal Unsigned	BGEU x1, x2,offset	if(Regs[x1] > Regs[x2]): PC <- PC+ (offset<< 1) (Unsigned comparison)

Encoding Formats



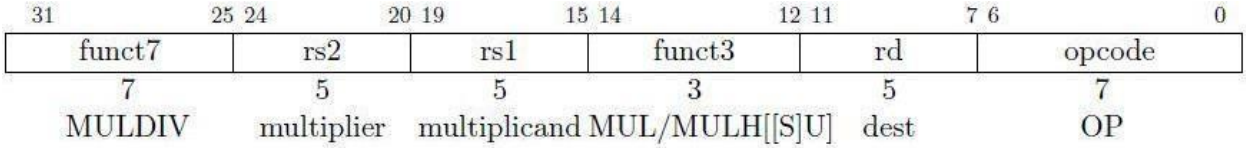


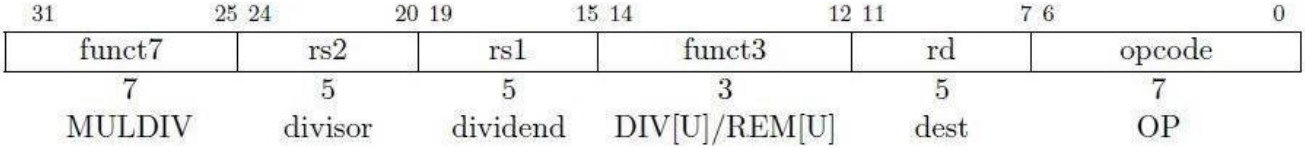
2.4 Multiplication / Division Instructions

Multiplication and division instructions are used to perform arithmetic operations on integer values.

Instruction	Instruction Type	Instruction Name	Example	Description
MUL	R - Type	Multiplicati on	MUL x1,x2,x3	Reg[x1]<-[Reg[x2]*Reg[x3]]$_{\text{lower 32bit}}$
MULH	R - Type	Multiplicati on High	MULH x1,x2,x3	Reg[x1]<-[Reg[x2]*Reg[x3]]$_{\text{upper32bit}}$ Signed x Signed
MULHU	R - Type	Multiplicati on High Unsigned	MULHU x1,x2,x3	Reg[x1]<-[Reg[x2]*Reg[x3]]$_{\text{upper32bit}}$ (Unsigned x Unsigned)
MULHSU	R - Type	Multiplicati on High Signed UnSigned	MULHSU x1,x2,x3	Reg[x1]<-[Reg[x2]*Reg[x3]]$_{\text{upper32bit}}$ (Signed x Unsigned)
DIV	R - Type	Division	DIV x1,x2,x3	Reg[x1]<-[Reg[x2] / Reg[x3]] (Signed Division) Round towards Zero
DIVU	R - Type	Division Unsigned	DIVU x1,x2,x3	Reg[x1]<-[Reg[x2] / Reg[x3]] (Unsigned Division) Round towards Zero
REM	R - Type	Remainder	REM x1,x2,x3	Reg[x1]<-[Reg[x2] % Reg[x3]] Sign of the result = Sign of the dividend
REMU	R - Type	Remainder Unsigned	REMUx1,x2,x3	Reg[x1]<-[Reg[x2] % Reg[x3]] (Unsigned Division)

Encoding Formats



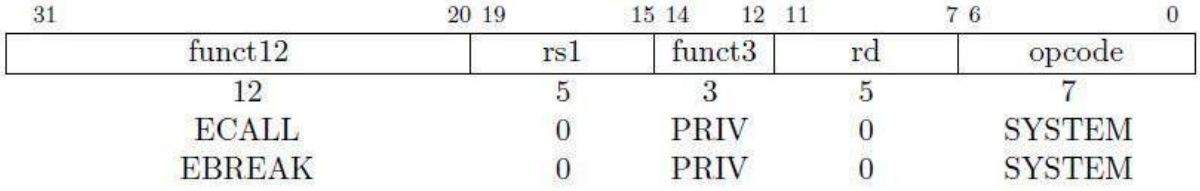


2.5 ECALL and EBREAK Instructions

Environment call (ECALL) and environment breakpoints (EBREAK) instructions are used to provide a way for programs to interact with the operating system and debugging tools.

The ECALL instruction is used to make a system call to the operating system. This instruction triggers an exception, and the operating system responds by performing the requested action, such as opening a file or allocating memory.

Encoding Formats

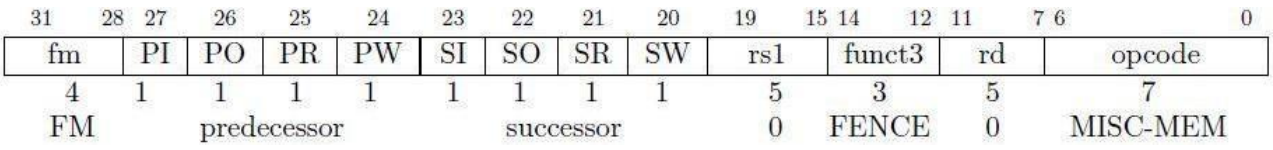


31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

2.6 FENCE Instruction

FENCE instructions are used to ensure that memory operations are performed in a specific order. FENCE stands for "memory fence," and these instructions are used to create a memory barrier that ensures that memory operations are completed before other operations can proceed.

Encoding Formats



31 — 28	27	26	25	24	23	22	21	20	19 — 15	14 — 12	11 — 7	6 — 0
fm	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
FM	predecessor				successor				0	FENCE	0	MISC-MEM

3. Opcodes

Each opcode in a specific architecture, represents a specific operation that the processor can perform, such as arithmetic operations, logic operations, data transfer operations, and control flow operations.RISC-V provides a large number of opcodes that cover a wide range of operations, including arithmetic operations (such as addition, subtraction, multiplication, and division), logical operations (such as AND, OR, and XOR), data transfer operations (such as load and store), and control flow operations (such as branch and jump).

Here Are some basic opcodes which are used in RISC V arcitecture.

- **ADD**: This opcode is used to add two values together and store the result in a register.
- **SUB**: This opcode is used to subtract one value from another and store the result in a register.
- **LOAD**: This opcode is used to load a value from memory into a register.
- **STORE**: This opcode is used to store a value from a register into memory.
- **BRANCH**: This opcode is used to branch to a different part of the program based on a certain condition.
- **JUMP**: This opcode is used to jump to a different part of the program.
- **AND**: This opcode is used to perform a logical AND operation on two values and store the result in a register.
- **OR**: This opcode is used to perform a logical OR operation on two values and store the result in a register.
- **XOR**: This opcode is used to perform a logical XOR operation on two values and store the result in a register.
- **SLT**: This opcode is used to set a register to 1 if the first value is less than the second value, or 0 otherwise.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
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[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[11:5]	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[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
0100000	shamt	rs1	101	rd	0010011	SRAI
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
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

FIGURE 1 : RV32I BASE INSTRUCTION SET FORMAT WITH OPCODES

Considering the OPCODES with RV32M standard extensions.

The RV32M standard extensions refer to a set of standard extensions for the RISC-V 32-bit instruction set architecture (ISA) that are designed to support integer multiplication and division, as well as hardware support for multiplication with accumulation and division with remainder. These extensions are commonly used in embedded systems, signal processing, and other applications that require efficient multiplication and division operations.

The RV32M extensions include the following:

- **M (Integer Multiplication and Division)**: This extension adds support for integer multiplication and division instructions, including multiply (MUL), multiply high (MULH), multiply high unsigned (MULHU), multiply-add (MADD), multiply-add unsigned (MADDU), multiply-subtract (MSUB), and multiply-subtract unsigned.

- **A (Atomic Instructions)**: This extension adds support for atomic memory access instructions, including load-reserved (LR), store-conditional (SC), atomic memory operations (AMOs), and fence instructions.

- **F (Single-Precision Floating-Point)**: This extension adds support for single-precision floating-point arithmetic instructions, including floating-point add (FADD.S), floating-point subtract (FSUB.S), floating-point multiply (FMUL.S), and floating-point divide (FDIV.S).

- **D (Double-Precision Floating-Point)**: This extension adds support for double-precision floating-point arithmetic instructions, including floating-point add (FADD.D), floating-point subtract (FSUB.D), floating-point multiply (FMUL.D), and floating-point divide (FDIV.D).

- **Q (Quad-Precision Floating-Point)**: This extension adds support for quad-precision floating-point arithmetic instructions, including floating-point add (FADD.Q), floating-point subtract (FSUB.Q), floating-point multiply (FMUL.Q), and floating-point divide (FDIV.Q).

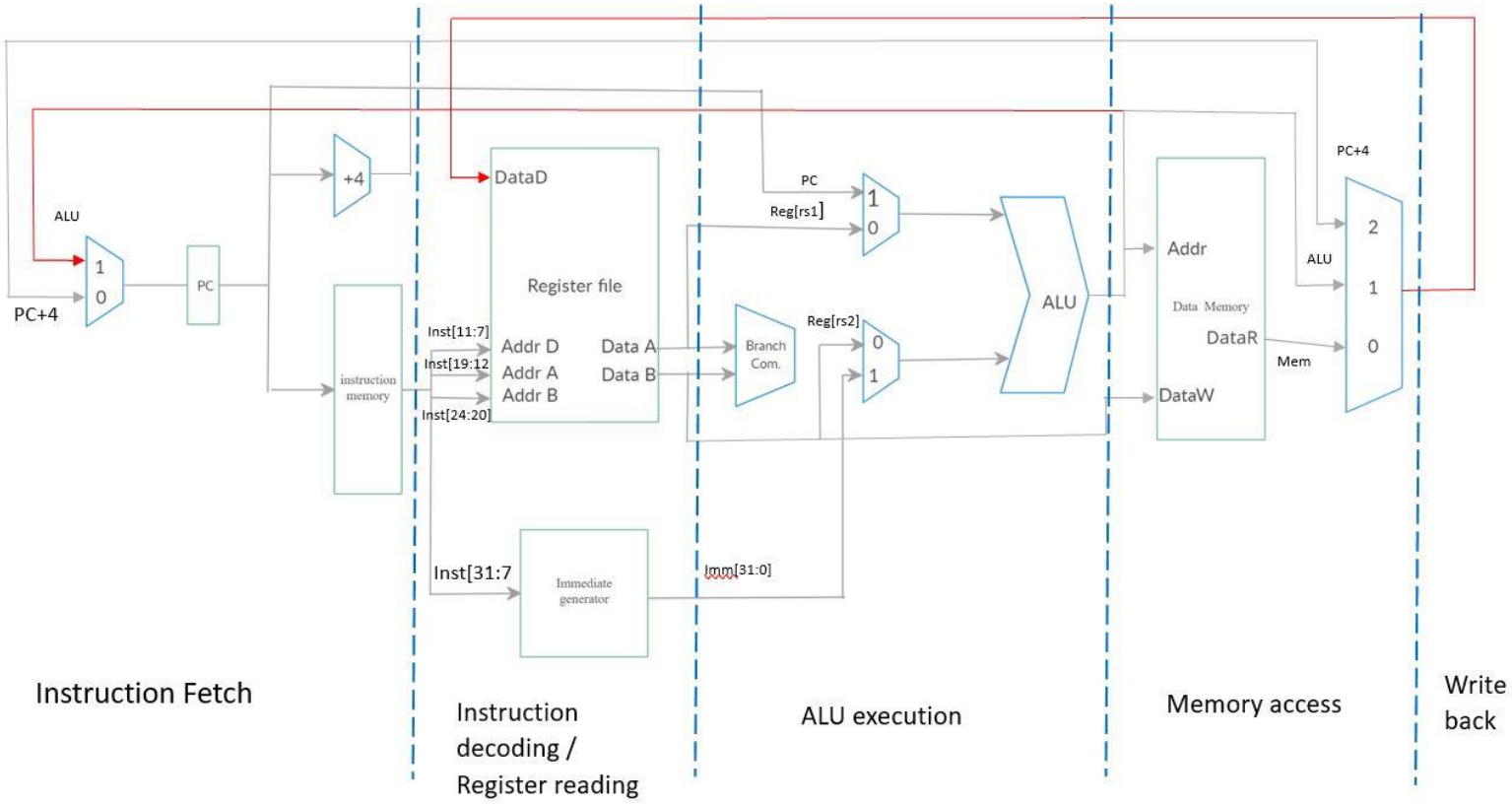
- **L (Integer Multiply-Add with Carry)**: This extension adds support for integer multiplication with accumulation instructions, including multiply-add with carry (MAC), multiply-add with carry unsigned (MACU), and multiply-subtract with carry (MSAC).

RV32M Standard Extension

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

FIGURE 2 : RV32M BASE INSTRUCTION SET FORMAT WITH OPCODES

4. Pipeline Diagram



5. Hardware

5.1 Hardware Units

●Instruction Memory
●Register File (32 x 32 bit registers)
●ALU (Multiplication and shifting hardware included)
●Data Memory
●Control Unit
●Branch logic (For comparing and generating the control signal related to branch and jump instruction)
●Immediate selecting hardware (Hardware to select the correct immediate format and for sign extending)

5.2 Control Signals

- ALUOP - Selecting ALU operation
- REG_WRITE_EN - Enable the writing to the Register File
- PC_SEL - Select the source to the PC register
- IMM_SEL - Select the correct format of the immediate
- OP1SEL - Select the ALU operand 1 source
- OP2SEL - Select the ALU operand 2 source
- MEM_WRITE - Enable writing to the data memory
- MEM_READ - Enable reading from the data memory
- WB_SEL - Select the write back value source
- BRANCH_JUMP - Select the branch comparing type and the jump

INSTRUCTION	PC_SEL	IMM_SEL	OP1_SEL	OP2_SEL	ALUOP	MEM_WRITE	MEM_READ	REG_WRITE_EN	WB_SEL	BRANCH_JUMP
LUI	PC + 4	U	*	IMM	FORWARD	0	0	1	ALU	000
AUIPC	PC + 4	U	PC	IMM	ADD	0	0	1	ALU	000
JAL	ALU	J	PC	IMM	ADD	0	0	1	PC + 4	111
JALR	ALU	I	DATA1	IMM	ADD	0	0	1	PC + 4	111
BEQ	PC + 4 / ALU	B	PC	IMM	ADD	0	0	0	*	001
BNE	PC + 4 / ALU	B	PC	IMM	ADD	0	0	0	*	010
BLT	PC + 4 / ALU	B	PC	IMM	ADD	0	0	0	*	011
BGE	PC + 4 / ALU	B	PC	IMM	ADD	0	0	0	*	100
BLTU	PC + 4 / ALU	B	PC	IMM	ADD	0	0	0	*	101
BGEU	PC + 4 / ALU	B	PC	IMM	ADD	0	0	0	*	110
LB	PC + 4	I	DATA1	IMM	ADD	0	1	1	MEM	000
LH	PC + 4	I	DATA2	IMM	ADD	0	1	1	MEM	000
LW	PC + 4	I	DATA3	IMM	ADD	0	1	1	MEM	000
LBU	PC + 4	I	DATA4	IMM	ADD	0	1	1	MEM	000
LHU	PC + 4	I	DATA5	IMM	ADD	0	1	1	MEM	000
SB	PC + 4	S	DATA6	IMM	ADD	1	0	0	*	000
SH	PC + 4	S	DATA7	IMM	ADD	1	0	0	*	000
SW	PC + 4	S	DATA8	IMM	ADD	1	0	0	*	000
ADDI	PC + 4	I	DATA1	IMM	ADD	0	0	1	ALU	000

INSTRUCTION	PC_SEL	IMM_SEL	OP1_SEL	OP2_SEL	ALUOP	MEM_WRITE	MEM_READ	REG_WRITE_EN	WB_SEL	BRANCH_JUMP
SLTI	PC + 4	I	DATA1	IMM	SLT	0	0	1	ALU	000
SLTIU	PC + 4	IU	DATA1	IMM	SLTU	0	0	1	ALU	000
XORI	PC + 4	I	DATA1	IMM	XOR	0	0	1	ALU	000
ORI	PC + 4	I	DATA1	IMM	OR	0	0	1	ALU	000
ANDI	PC + 4	I	DATA1	IMM	AND	0	0	1	ALU	000
SLLI	PC + 4	SFT	DATA1	IMM	SLL	0	0	1	ALU	000
SRLI	PC + 4	SFT	DATA1	IMM	SRL	0	0	1	ALU	000
SRAI	PC + 4	SFT	DATA1	IMM	SRA	0	0	5	ALU	000
ADD	PC + 4	*	DATA1	DATA2	ADD	0	0	1	ALU	000
SUB	PC + 4	*	DATA1	DATA2	SUB	0	0	1	ALU	000
SLL	PC + 4	*	DATA1	DATA2	SLL	0	0	1	ALU	000
SLT	PC + 4	*	DATA1	DATA2	SLT	0	0	1	ALU	000
SLTU	PC + 4	*	DATA1	DATA2	SLTU	0	0	1	ALU	000
XOR	PC + 4	*	DATA1	DATA2	XOR	0	0	1	ALU	000
SRL	PC + 4	*	DATA1	DATA2	SRL	0	0	1	ALU	000
SRA	PC + 4	*	DATA1	DATA2	SRA	0	0	1	ALU	000
OR	PC + 4	*	DATA1	DATA2	OR	0	0	1	ALU	000
AND	PC + 4	*	DATA1	DATA2	AND	0	0	1	ALU	000
MUL	PC + 4	*	DATA1	DATA2	MUL	0	0	1	ALU	000
MULH	PC + 4	*	DATA1	DATA2	MULH	0	0	1	ALU	000
MULHSU	PC + 4	*	DATA1	DATA2	MULHSU	0	0	1	ALU	000
MULHU	PC + 4	*	DATA1	DATA2	MULHU	0	0	1	ALU	000
DIV	PC + 4	*	DATA1	DATA2	DIV	0	0	1	ALU	000

INSTRUCTION	PC_SEL	IMM_SEL	OP1_SEL	OP2_SEL	ALUOP	MEM_WRITE	MEM_READ	REG_WRITE_EN	WB_SEL	BRANCH_JUMP
DIVU	PC + 4	*	DATA1	DATA2	DIVU	0	0	1	ALU	000
REM	PC + 4	*	DATA1	DATA2	REM	0	0	1	ALU	000
REMU	PC + 4	*	DATA1	DATA2	REMU	0	0	1	ALU	000