RISC-V

Study and Implementation of RISC-V Architecture

Trainee - Soumyadip Manna

Summer,2025

Mentor - Ashish Jindal - SC-D

SOLID STATE PHYSICS LABORATORY, DRDO,DELHI

INTRODUCTION

At its core, a digital computer has at least one Central Processing Unit (CPU). A CPU executes a continuous stream of instructions called a program. These program instructions are expressed in what is called machine language. Each machine language instruction is a binary value.

- **The Digital Computer**

There are different types of computers. A digital computer is the type that most people think of when they hear the word computer. Other varieties of computers include analog and quantum. A digital computer is one that processes data represented using numeric values (digits), most commonly expressed in binary (ones and zeros) form.

A typical digital computer is composed of storage systems (memory, disc drives, USB drives, etc.),a CPU (with one or more cores), input peripherals (a keyboard and mouse) and output peripherals (display, printer or speakers.)

CPU

The CPU is a collection of registers and circuitry designed to manipulate the register data and to exchange data and instructions with the main memory. The instructions that are read from the main memory tell the CPU to perform various mathematical and logical operations on the data in its registers and where to save the results of those operations.

- **Execution Unit**

  The part of a CPU that coordinates all aspects of the operations of each instruction is called the execution unit. It is what performs the transfers of instructions and data between the CPU and the main memory and tells the registers when they are supposed to either store or recall data being transferred. The execution unit also controls the ALU (Arithmetic and Logic Unit)

- **Arithmetic and Logic Unit**

  When an instruction manipulates data by performing things like an addition, subtraction, comparison or other similar operations , the ALU is what will calculate the sum, difference, and so on. . . under the control of the execution unit.

COMPUTER ARCHITECTURE

It is the *design and organization* of a computer's functional components, It defines how a computer system is structured, how it operates, and how different components (like CPU, memory, and I/O devices) interact with each other.

It includes the instruction set, hardware components, data formats, and techniques for memory addressing and control.

Example Of different Computer Architecture

> **Von Neumann**

- **Single memory** shared by both instructions and data.
- **Sequential execution**: One instruction at a time.
- **Von Neumann bottleneck**: Limited data transfer rate between CPU and memory.

> **Harvard**

- **Separate memory** for instructions and data.
- Can **fetch instructions and data simultaneously** — higher throughput.
- Often used in **DSPs (Digital Signal Processors)** and **microcontrollers**.

> RISC

- Simple, **fixed-length instructions**.
- Emphasizes **speed** and **efficiency** with a **load/store architecture**.
- Examples: **RISC-V**, **ARM**, **MIPS**.

  - **Features**:
    - One instruction per clock cycle
    - Fewer instructions, but faster
    - Easy pipelining

> CISC

- Large, **complex instructions** that can do multiple operations.
- Example: **x86 architecture**.

  - **Features**:
    - Variable-length instructions
    - Designed to reduce number of instructions per program
    - More hardware complexity

ISA [Instruction Set Architecture]

The catalog of rules that describes the details of the instructions and features that a given CPU provides is called an **Instruction Set Architecture (ISA)**. An ISA is typically expressed in terms of the specific meaning of each binary instruction that a CPU can recognize and how it will process each one.

The RISC-V ISA is dened as a set of modules. The purpose of dividing the ISA into modules is to allow an implementer to select which features to incorporate into a CPU design.

Any given RISC-V implementation must provide one of the base modules and zero or more of the extension modules.

- **RV Base Module**

The base modules are **RV32I (32-bit general purpose)**, **RV32E (32-bit embedded)**, **RV64I (64-bit general purpose)** and **RV128I (128-bit general purpose)**.

These base modules provide the minimal functional set of integer operations needed to execute a useful application

- **RV Extension Module**

RISC-V extension modules may be included by an implementer interested in optimizing a design for one or more purposes.

Available extension modules include **M (integer math)**, **A (atomic)**, **F (32-bit floating point)**, **D (64-bit floating point)**, **Q (128-bit floating point)**, **C (compressed size instructions)** and others.

RISC

RISC stands for **Reduced Instruction Set Computer**. It is a computer architecture design philosophy that uses a small, highly optimized set of instructions, allowing for faster and more efficient execution.

- Why Choosing RISC Over CISC ?

Feature	RISC	CISC
Instruction Set	Small, simple	Large, complex
Instruction Size	Fixed	Variable
Execution Time	One instruction per cycle	May take multiple cycles
Examples	RISC-V, ARM, MIPS	x86, Intel 8086, AMD

RISC ARCHITECTURE

Year	Event
1960s	IBM 801 project began — the roots of RISC thinking.
1974	John Cocke at IBM introduced the **IBM 801**, the **first RISC processor prototype**.
1981	**UC Berkeley** RISC project led by David Patterson released **RISC I**, later **RISC II**.
1984	**Stanford University** released the **MIPS** project led by John Hennessy (became MIPS Technologies).
1985	**ARM (Acorn RISC Machine)** developed in the UK, evolved into one of the most successful RISC families.
1990s	RISC used in **workstations and embedded systems**; Apple, Sun (SPARC), SGI, etc., adopted it.
2000s–Present	**ARM dominates mobile devices**; **RISC-V** emerges as **open-source** RISC architecture, gaining popularity in academia and industry.

Greatness About RISC-V

RISC-V offers a modern, open-standard Instruction Set Architecture (ISA) that addresses key challenges in hardware design, scalability, and ecosystem support. Its adoption is growing rapidly due to the following core strengths:

1. Open and Royalty-Free Standard

RISC-V is governed by an open license, allowing unrestricted use for academic, commercial, and industrial purposes. This eliminates proprietary licensing costs and promotes transparent hardware development.

2. Modular and Extensible Design

The base ISA (e.g., RV32I) is intentionally minimal, with optional standard extensions (e.g., M for multiplication/division, F/D for floating-point operations, C for compressed instructions). This modularity allows implementers to tailor designs to application-specific requirements, from microcontrollers to high-performance processors.

3. Simplicity and Implementability

RISC-V maintains a clean, orthogonal instruction set with fixed instruction lengths, enabling easier implementation, verification, and optimization. This makes it highly suitable for both educational and industrial processor development.

Greatness About RISC-V

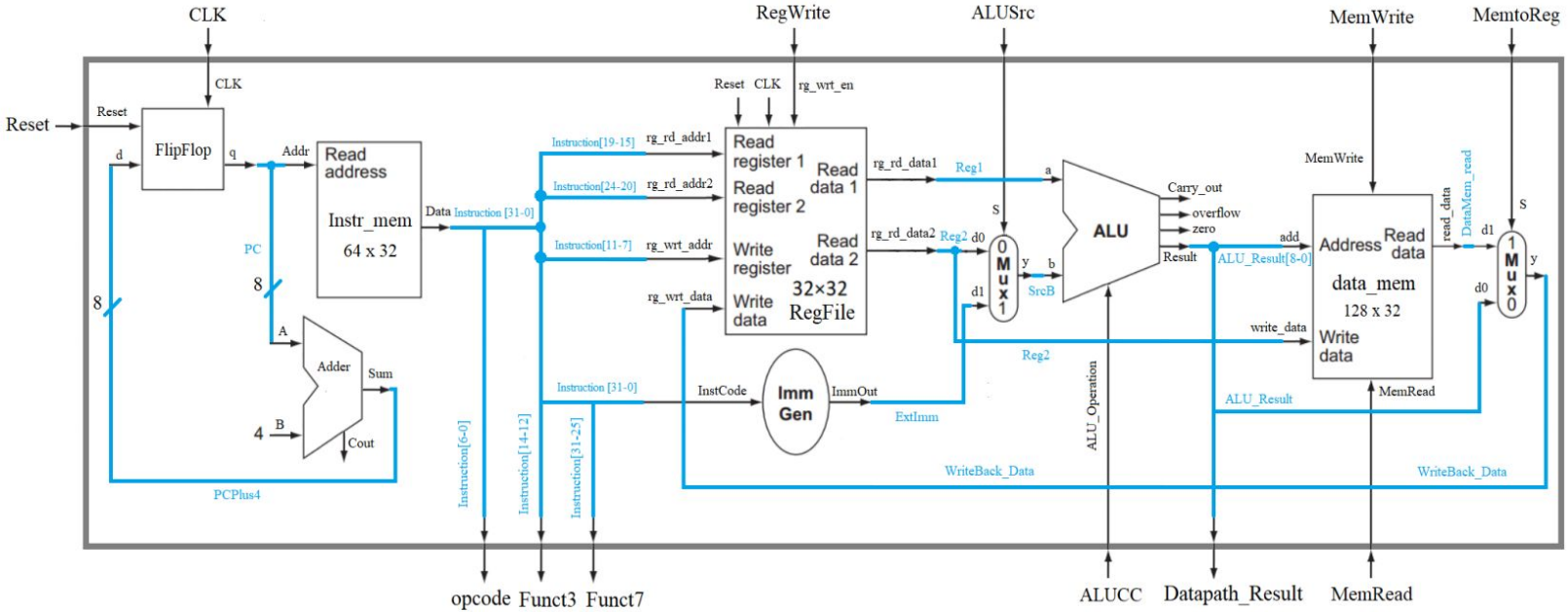
4. Scalability Across Platforms

The ISA supports 32-bit, 64-bit, and 128-bit address spaces (RV32, RV64, RV128), facilitating use in a wide range of domains—from embedded systems to high-performance computing.

5. Efficient Performance Architecture

RISC-V adheres to load/store principles and supports pipelining and parallelism efficiently. It enables high instruction throughput and deterministic performance characteristics, especially beneficial for real-time and low-power designs.

SINGLE CYCLE RISC-V 32 bit PROCESSOR



HART

Analogous to a core in other types of CPUs, a **hart (hardware thread)** in a RISC-V CPU refers to the collection of **32 registers, instruction execution unit and ALU**.

When more than one hart is present in a CPU, a different stream of instructions can be execute done each hart all at the same time. Programs that are written to take advantage of this are called multithreaded.

Single-threaded is like a chef doing everything alone

Imagine one chef in a kitchen:

- He chops, cooks, serves, and cleans — one task at a time.
- That's single-threading — simple, but slow under heavy load.

Single-Threaded

Multi-threading is like a team of chefs

Now imagine many chefs working in parallel:

- One chops veggies, another boils water, another serves dishes.
- This is multi-threading — more efficient, but needs coordination (locks, semaphores, etc.).

Multi-Threading

CPU REGISTERS AND SP REGISTERS

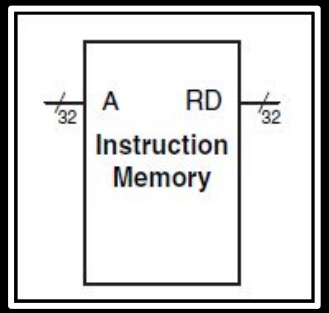
In the RV32 CPU there are **31 general purpose registers** that **each contain 32 bits** (where each bit is one binary digit value of one or zero) and a number of special-purpose registers. Each of the general purpose registers is given a name such as **x1, x2, . . . on up to x31** (general purpose refers to the fact that the CPU itself does not prescribe any particular function to any of these registers.) **Two important special-purpose registers are x0 and pc**.

Reg	ABI/Alias	Description	Saved
x0	zero	Hard-wired zero	
x1	ra	Return address	
x2	sp	Stack pointer	yes
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary/alternate link register	
x6-7	t1-2	Temporaries	
x8	s0/fp	Saved register/frame pointer	yes
x9	s1	Saved register	yes
x10-11	a0-1	Function arguments/return value	
x12-17	a2-7	Function arguments	
x18-27	s2-11	Saved registers	yes
x28-31	t3-6	Temporaries	

Register x0 will always represent the value zero or logical false no matter what. If any instruction tries to change the value in x0 the operation will fail. The need for zero is so common that, other than the fact that it is **hard-wired to zero**, the x0 register is made available as if it were otherwise a general purpose register.

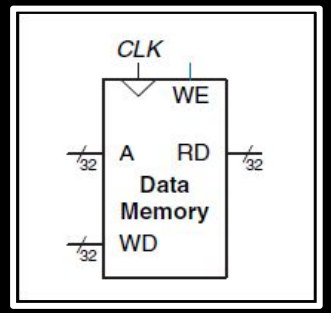
The pc register is called the **program counter**. The CPU uses it to remember the memory address where its program instructions are located.

MEMORY



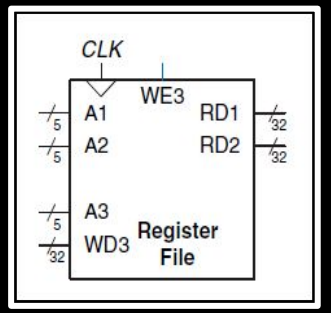
The instruction memory has a single read port. It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD.

Instruction Memory



The data memory has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock.
If the write enable is 0, it reads address A onto RD

Data Memory



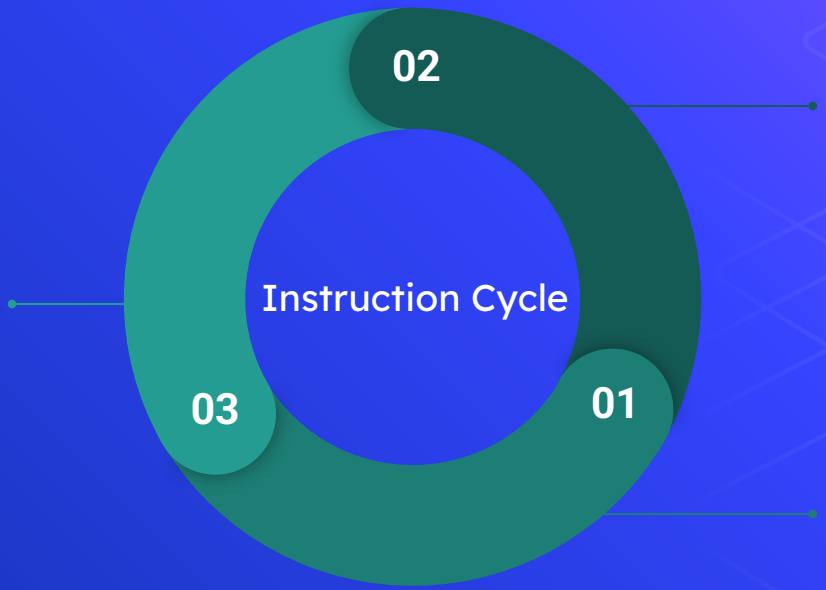
The 32-element × 32-bit register file has two read ports (A1 and A2)and one write port(A3) with 5 bits address input. They read the 32-bit register values onto read data outputs RD1 and RD2, respectively. A 32-bit write data input, WD; a write enable input, WE3; and a clock.

Register File

PROGRAM EXECUTION

Instruction Execute

During the execute stage, the CPU performs the operation specified by the instruction, such as arithmetic computation or data transfer. If the instruction involves computation, the result is typically calculated using values from registers. In the case of memory operations, the CPU may store data to or load data from a specific memory address. For control flow instructions like jump or branch, the CPU may modify the program counter (PC) to alter the sequence of execution based on conditions. Once the execution is complete, the updated PC determines the next instruction to be fetched in the subsequent cycle.



Instruction Decode

Once an instruction has been fetched, it must be inspected to determine what operation(s) are to be performed. This means inspecting the portions of the instruction that dictate which registers are involved and what that, if anything, ALU should do.

Instruction Fetch

In order to fetch an instruction from the main memory the CPU will update the address in the pc register and then request that the main memory return the value of the data stored at that address.

NB: In the RISC-V ISA the pc register points to the current instruction where in most other designs, the pc register points to the next instruction.

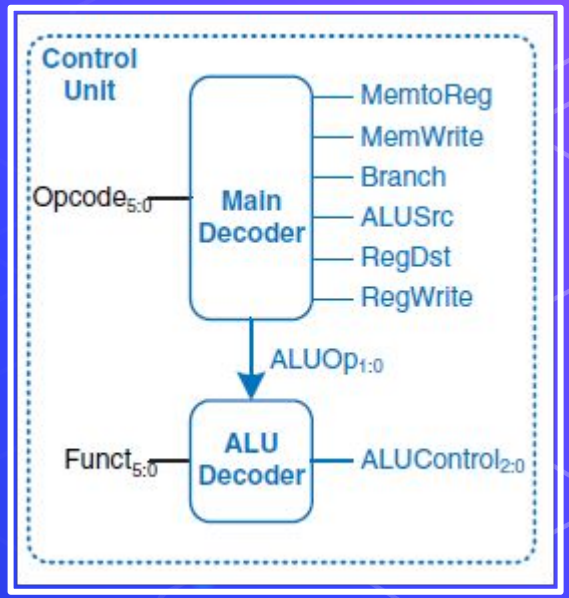
ALU

An Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit. For example, a typical ALU might perform addition, subtraction, magnitude comparison, AND, and OR operations. The ALU forms the heart of most computer systems.
N-bit ALU with N-bit inputs and outputs. The ALU receives a control signal F that specifies which function to perform.

ALUControl	Operation	Related Mnemonic
4'h0	add	ADD, ADDI
4'h1	sub	SUB
4'h2	shift left	SLL, SLLI
4'h3	signed less than	SLT, BLT
4'h4	unsigned less than	SLTU, BLTU
4'h5	xor	XOR, XORI
4'h6	signed shift right	SRA, SRAI
4'h7	logical shift right	SRL, SRLI
4'h8	or	OR, ORI
4'h9	and	AND, ANDI
4'hA	equal comparison	BEQ
4'hB	not equal	BNE
4'hC	greater/equal (signed)	BGE
4'hD	greater/equal (unsigned)	BGEU

CONTROL UNIT

The control unit computes the control signals based on the opcode and funct fields of the instruction. Most of the control information comes from the opcode, but R-type instructions also use the funct field to determine the ALU operation. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic.



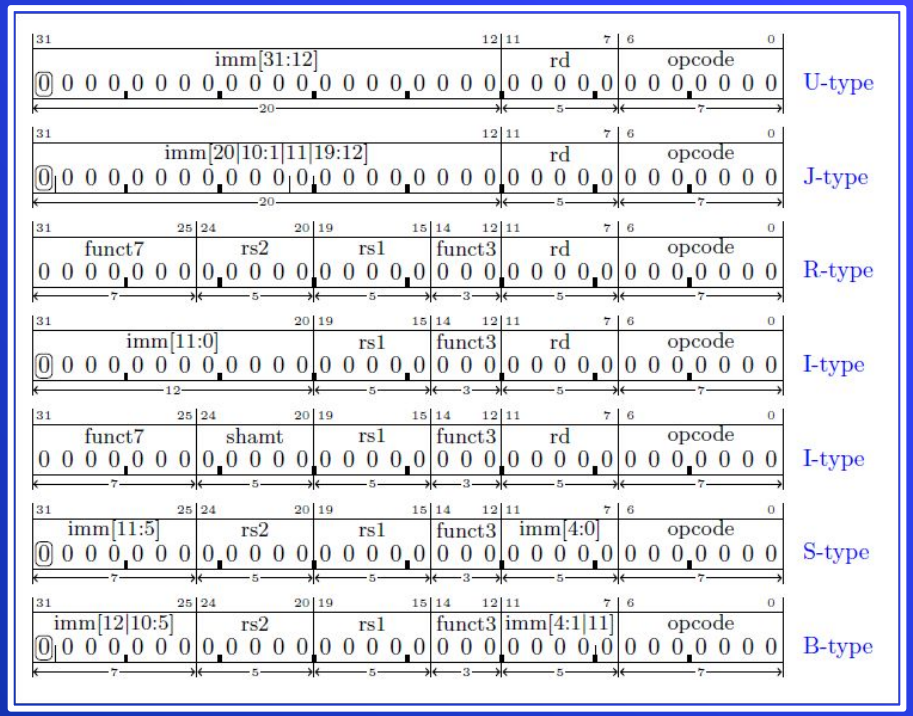
Main Decoder

The main decoder computes most of the outputs from the opcode. It also determines a ALUOp signal

ALU Decoder

The ALU decoder uses this ALUOp signal in conjunction with the funct field to compute ALUControl.

RISC-V INSTRUCTION FORMATS



** Addressing Modes immediate, register, base-displacement, pc-relative

Conventions and Terminology

Term	Meaning
XLEN	Register width (e.g., 32, 64 ı ard 128 bits)
sx(val)	Sign-extend value to the left
zx(val)	Zero-extend value to the left
zr(val)	8-bit memory at address *addr*
m1(addr)	16-bit little-endian memory at *addr*
m16(addr)	32-bit little-endian memory at *addr*
m32(addr)	64-bit little-endian memory at *addr*
m64(addr)	128-bit little-endian memory at *addr*
.+offset	Current instruction address ± offset
.−offset	Current instruction address − offset
pcrel13	PC-relative address within ±4KB
pcrel21	PC-relative address within ±1MB
pc	Program Counter (current instruction address)
rd	Destination register
rs1	Source register 1
rs2	Souree register 2 used in instructions

RISC-V INSTRUCTION SET

Usage Template		Type	Description	Detailed Description
add	rd, rs1, rs2	R	Add	rd ← rs1 + rs2, pc ← pc+4
addi	rd, rs1, imm	I	Add Immediate	rd ← rs1 + imm_i, pc ← pc+4
and	rd, rs1, rs2	R	And	rd ← rs1 ∧ rs2, pc ← pc+4
andi	rd, rs1, imm	I	And Immediate	rd ← rs1 ∧ imm_i, pc ← pc+4
auipc	rd, imm	U	Add Upper Immediate to PC	rd ← pc + imm_u, pc ← pc+4
beq	rs1, rs2, pcrel_13	B	Branch Equal	pc ← pc + ((rs1==rs2) ? imm_b : 4)
bge	rs1, rs2, pcrel_13	B	Branch Greater or Equal	pc ← pc + ((rs1>=rs2) ? imm_b : 4)
bgeu	rs1, rs2, pcrel_13	B	Branch Greater or Equal Unsigned	pc ← pc + ((rs1>=rs2) ? imm_b : 4)
blt	rs1, rs2, pcrel_13	B	Branch Less Than	pc ← pc + ((rs1<rs2) ? imm_b : 4)
bltu	rs1, rs2, pcrel_13	B	Branch Less Than Unsigned	pc ← pc + ((rs1<rs2) ? imm_b : 4)
bne	rs1, rs2, pcrel_13	B	Branch Not Equal	pc ← pc + ((rs1!=rs2) ? imm_b : 4)
jal	rd, pcrel_21	J	Jump And Link	rd ← pc+4, pc ← pc+imm_j
jalr	rd, imm(rs1)	I	Jump And Link Register	rd ← pc+4, pc ← (rs1+imm_i) & ∼1
lb	rd, imm(rs1)	I	Load Byte	rd ← sx(m8(rs1+imm_i)), pc ← pc+4
lbu	rd, imm(rs1)	I	Load Byte Unsigned	rd ← zx(m8(rs1+imm_i)), pc ← pc+4
lh	rd, imm(rs1)	I	Load Halfword	rd ← sx(m16(rs1+imm_i)), pc ← pc+4
lhu	rd, imm(rs1)	I	Load Halfword Unsigned	rd ← zx(m16(rs1+imm_i)), pc ← pc+4
lui	rd, imm	U	Load Upper Immediate	rd ← imm_u, pc ← pc+4
lw	rd, imm(rs1)	I	Load Word	rd ← sx(m32(rs1+imm_i)), pc ← pc+4
or	rd, rs1, rs2	R	Or	rd ← rs1 ∨ rs2, pc ← pc+4
ori	rd, rs1, imm	I	Or Immediate	rd ← rs1 ∨ imm_i, pc ← pc+4
sb	rs2, imm(rs1)	S	Store Byte	m8(rs1+imm_s) ← rs2[7:0], pc ← pc+4
sh	rs2, imm(rs1)	S	Store Halfword	m16(rs1+imm_s) ← rs2[15:0], pc ← pc+4
sll	rd, rs1, rs2	R	Shift Left Logical	rd ← rs1 << (rs2%XLEN), pc ← pc+4
slli	rd, rs1, shamt	I	Shift Left Logical Immediate	rd ← rs1 << shamt_i, pc ← pc+4
slt	rd, rs1, rs2	R	Set Less Than	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4

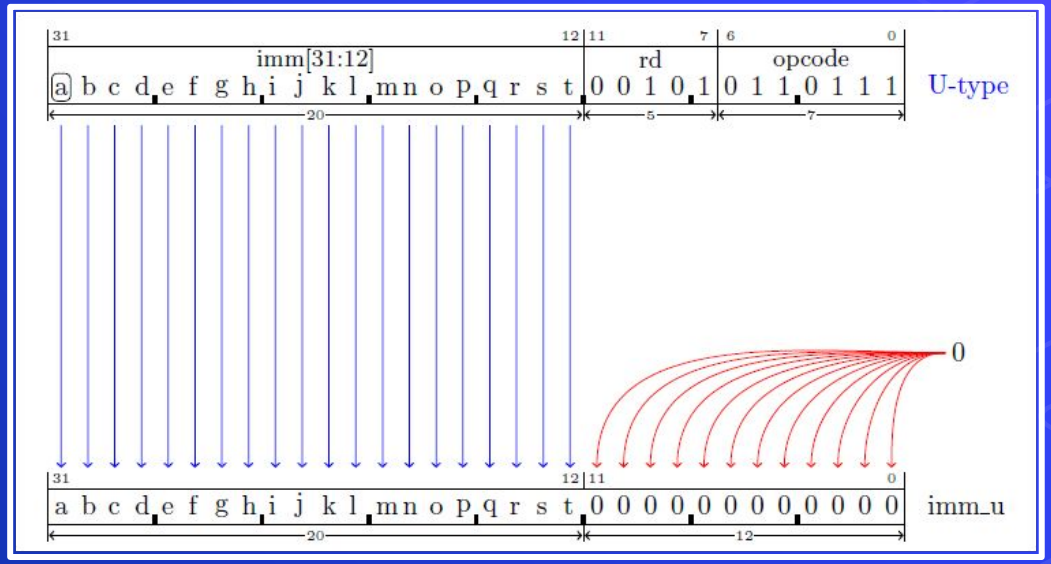
RISC-V INSTRUCTION SET

Usage Template		Type	Description	Detailed Description
slti	rd, rs1, imm	I	Set Less Than Immediate	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4
sltiu	rd, rs1, imm	I	Set Less Than Immediate Unsigned	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4
sltu	rd, rs1, rs2	R	Set Less Than Unsigned	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4
sra	rd, rs1, rs2	R	Shift Right Arithmetic	rd ← rs1 >> (rs2%XLEN), pc ← pc+4
srai	rd, rs1, shamt	I	Shift Right Arithmetic Immediate	rd ← rs1 >> shamt_i, pc ← pc+4
srl	rd, rs1, rs2	R	Shift Right Logical	rd ← rs1 >> (rs2%XLEN), pc ← pc+4
srli	rd, rs1, shamt	I	Shift Right Logical Immediate	rd ← rs1 >> shamt_i, pc ← pc+4
sub	rd, rs1, rs2	R	Subtract	rd ← rs1 - rs2, pc ← pc+4
sw	rs2, imm(rs1)	S	Store Word	m32(rs1+imm_s) ← rs2[31:0], pc ← pc+4
xor	rd, rs1, rs2	R	Exclusive Or	rd ← rs1 ⊕ rs2, pc ← pc+4
xori	rd, rs1, imm	I	Exclusive Or Immediate	rd ← rs1 ⊕ imm_i, pc ← pc+4

31 25	24 20	19 15	14 12	11 7	6 0			
imm[31:12]				rd	0 1 1 0 1 1 1	U-type	lui	rd,imm
imm[31:12]				rd	0 0 1 0 1 1 1	U-type	auipc	rd,imm
imm[20|10:1|11|19:12]				rd	1 1 0 1 1 1 1	J-type	jal	rd,pcrel_21
imm[11:0]		rs1	0 0 0	rd	1 1 0 0 1 1 1	I-type	jalr	rd,imm(rs1)
imm[12|10:5]	rs2	rs1	0 0 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	beq	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	0 0 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bne	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 0 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	blt	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 0 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bge	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 1 0	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bltu	rs1,rs2,pcrel_13
imm[12|10:5]	rs2	rs1	1 1 1	imm[4:1|11]	1 1 0 0 0 1 1	B-type	bgeu	rs1,rs2,pcrel_13
imm[11:0]		rs1	0 0 0	rd	0 0 0 0 0 1 1	I-type	lb	rd,imm(rs1)
imm[11:0]		rs1	0 0 1	rd	0 0 0 0 0 1 1	I-type	lh	rd,imm(rs1)
imm[11:0]		rs1	0 1 0	rd	0 0 0 0 0 1 1	I-type	lw	rd,imm(rs1)
imm[11:0]		rs1	1 0 0	rd	0 0 0 0 0 1 1	I-type	lbu	rd,imm(rs1)
imm[11:0]		rs1	1 0 1	rd	0 0 0 0 0 1 1	I-type	lhu	rd,imm(rs1)
imm[11:5]	rs2	rs1	0 0 0	imm[4:0]	0 1 0 0 0 1 1	S-type	sb	rs2,imm(rs1)
imm[11:5]	rs2	rs1	0 0 1	imm[4:0]	0 1 0 0 0 1 1	S-type	sh	rs2,imm(rs1)
imm[11:5]	rs2	rs1	0 1 0	imm[4:0]	0 1 0 0 0 1 1	S-type	sw	rs2,imm(rs1)
imm[11:0]		rs1	0 0 0	rd	0 0 1 0 0 1 1	I-type	addi	rd,rs1,imm
imm[11:0]		rs1	0 1 0	rd	0 0 1 0 0 1 1	I-type	slti	rd,rs1,imm
imm[11:0]		rs1	0 1 1	rd	0 0 1 0 0 1 1	I-type	sltiu	rd,rs1,imm
imm[11:0]		rs1	1 0 0	rd	0 0 1 0 0 1 1	I-type	xori	rd,rs1,imm
imm[11:0]		rs1	1 1 0	rd	0 0 1 0 0 1 1	I-type	ori	rd,rs1,imm
imm[11:0]		rs1	1 1 1	rd	0 0 1 0 0 1 1	I-type	andi	rd,rs1,imm
0 0 0 0 0 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1	I-type	slli	rd,rs1,shamt
0 0 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srli	rd,rs1,shamt
0 1 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1	I-type	srai	rd,rs1,shamt
0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type	add	rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1	R-type	sub	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1	R-type	sll	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1	R-type	slt	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1	R-type	sltu	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1	R-type	xor	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type	srl	rd,rs1,rs2
0 1 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1	R-type	sra	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1	R-type	or	rd,rs1,rs2
0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1	R-type	and	rd,rs1,rs2

U-TYPE

The U-Type format is used for instructions that use a 20-bit immediate operand and an rd destination register.
The rd eld contains an x register number to be set to a value that depends on the instruction.
If XLEN=32 then the 20 bit imm value will extracted from the instruction and converted to form the imm_u value.



- **lui rd,imm**

  Set register rd to the imm_u value.

  For example: lui x23,0x12345 will result in setting register x23 to the value 0x12345000.

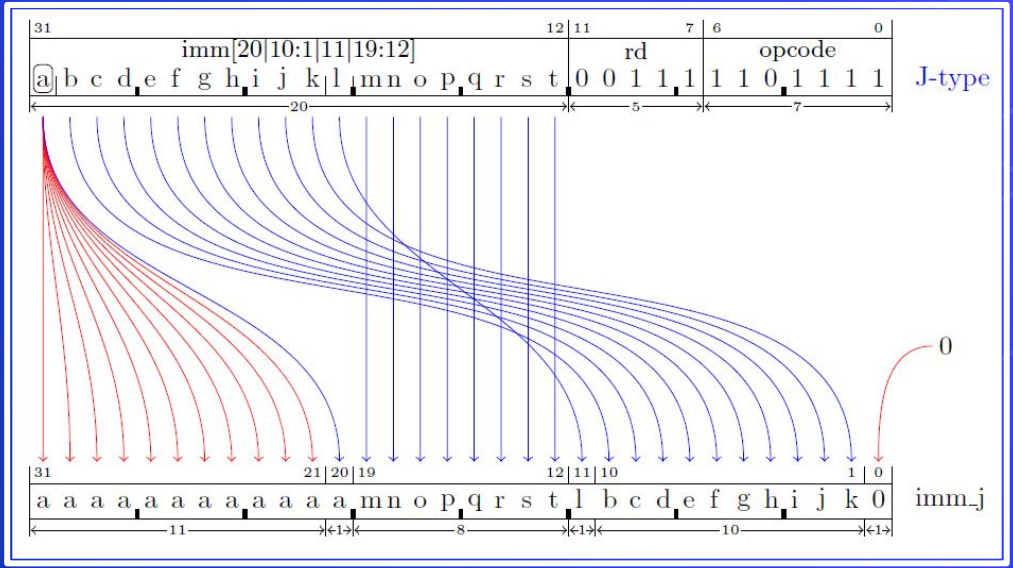
- **auipc rd,imm**

  Add the address of the instruction to the imm_u value and store the result in register rd.

  For example, if the instruction auipc x22, 0x10001 is executed from memory address 0x800012f4 then register x22 will be set to 0x900022f4.

J-TYPE

The J-type instruction format is used to encode the jal instruction with an immediate value that determines the jump target address.
It is similar to the U-type, but the bits in the immediate operand are arranged in a different order.
If XLEN=32 then the 20 bit imm value will extracted from the instruction and converted to form the imm_j value.



- **jal rd,pcrel 21**

  Set register rd to the address of the next instruction that would otherwise be executed (the address of the jal instruction + 4) and then jump to the address given by the sum of the pc register and the imm_j value.
  Note that pcrel_21 is expressed in the instruction as a target address or label that is converted to a 21-bit value representing a pc-relative offset to the target address.

  For example, consider the jal instructions in the following code:

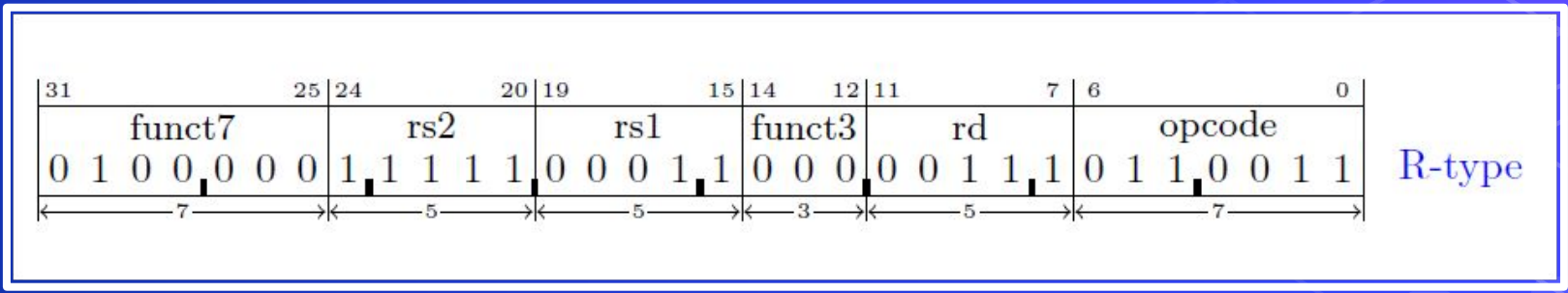
```
00000010: 000002ef jal x5,0x10     # jump to self (address 0x10)
00000014: 008002ef jal x5,0x1c     # jump to address 0x1c
00000018: 00100073 ebreak
0000001c: 00100073 ebreak
```

  The instruction at address 0x10 has a target address of 0x10 and the imm_j is zero because offset from the \current instruction" to the target is zero.
  The instruction at address 0x14 has a target address of 0x1c and the imm_j is 0x08 because 0x1c - 0x14 = 0x08.

R-TYPE

The R-type instructions are used for operations that set a destination register rd to the result of an arithmetic, logical or shift operation applied to source registers rs1 and rs2.

Note that instruction bit 30 (part of the the funct7 field) is used to select between the add and sub instructions as well as to select between srl and sra.



- **add rd,rs1,rs2**

  Set register rd to rs1 + rs2.
  Note that the value of funct7 must be zero for this instruction.

- **and rd,rs1,rs2**

  Set register rd to the bitwise and of rs1 and rs2. For example, if x17 = 0x55551111 and x18 = 0xff00ff00 then the instruction and x12,x17,x18 will set x12 to the value 0x55001100.

- **or rd,rs1,rs2**

  Set register rd to the bitwise or of rs1 and rs2.
  For example, if x17 = 0x55551111 and x18 = 0xff00ff00 then the instruction or x12,x17,x18 will set x12 to the value 0xff55ff11.

- **sll rd,rs1,rs2**

  Shift rs1 left by the number of bits specified in the least significant 5 bits of rs2 and store the result in rd1.
  For example, if x17 = 0x12345678 and x18 = 0x08 then the instruction sll x12,x17,x18 will set x12 to the value 0x34567800.

- **slt rd,rs1,rs2**

  If the signed integer value in rs1 is less than the signed integer value in rs2 then set rd to 1. Otherwise, set rd to 0.
  For example, if x17 = 0x12345678 and x18 = 0x0000ffff then the instruction slt x12,x17,x18 will set x12 to the value 0x00000000.

- **sltu rd,rs1,rs2**

  If the unsigned integer value in rs1 is less than the unsigned integer value in rs2 then set rd to1. Otherwise, set rd to 0.
  For example, if x17 = 0x12345678 and x18 = 0x0000ffff then the instruction sltu x12,x17,x18 will set x12 to the value 0x00000000.

- **sra rd,rs1,rs2**

  Arithmetic-shift rs1 right by the number of bits given in the least-significant 5 bits of the rs2 register and store the result in rd1
  For example, if x17 = 0x87654321 and x18 = 0x08 then the instruction sra x12,x17,x18 will set x12 to the value 0xff876543. Note that the value of funct7 must be zero for this instruction.

- **srl rd,rs1,rs2**

  Logic-shift rs1 right by the number of bits given in the least-significant 5 bits of the rs2 register and store the result in rd1.
  For example, if x17 = 0x87654321 and x18 = 0x08 then the instruction srl x12,x17,x18 will set x12 to the value 0x00876543. Note that the value of funct7 must be 0b0100000 for this instruction

- **sub rd,rs1,rs2**

  Set register rd to the bitwise xor of rs1 and rs2.
  Note that the value of funct7 must be 0b0100000 for this instruction. The value of funct7 is how the sub instruction is differentiated from the add instruction.

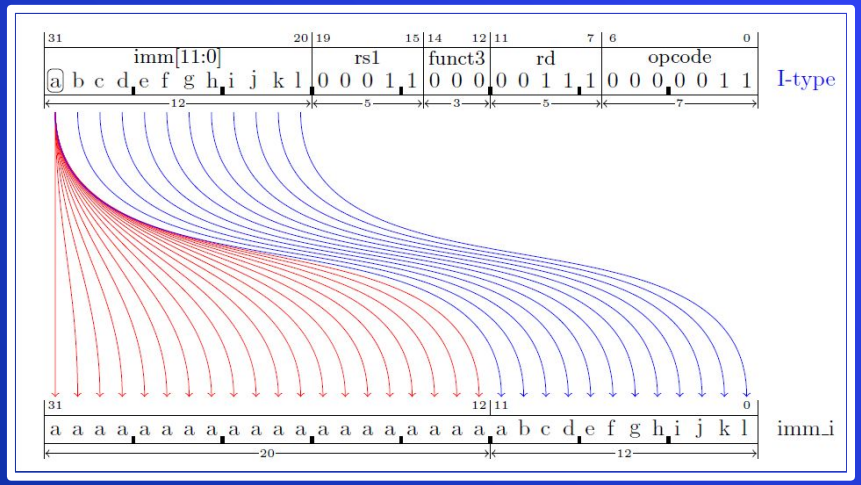
- **xor rd,rs1,rs2**

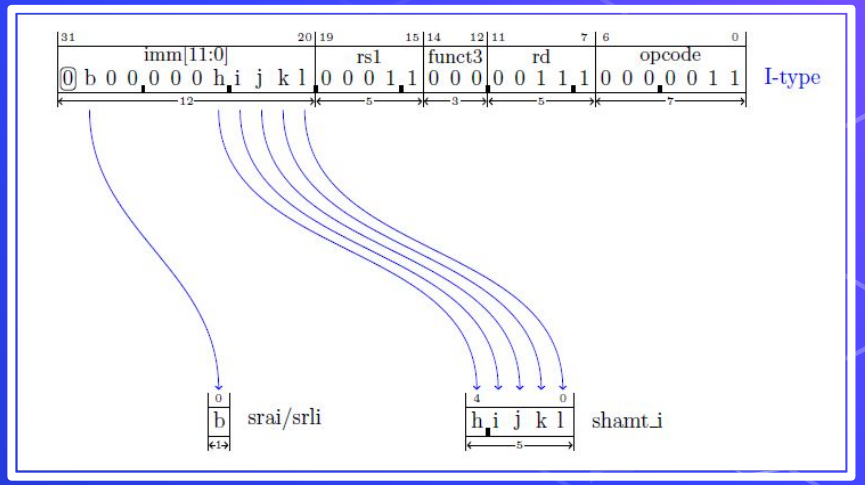
  Set register rd to the bitwise xor of rs1 and rs2.
  For example, if x17 = 0x55551111 and x18 = 0xff00ff00 then the instruction xor x12,x17,x18 will set x12 to the value 0xaa55ee11.

I-TYPE

The I-type instruction format is used to encode instructions with a signed 12-bit immediate operand with a range of [-2048....2047] an rd register, and an rs1 register.

If XLEN=32 then the 12-bit imm value will extracted from the instruction and converted to form the imm_i value.





** Decoding an I-type Shift Instruction.

- **addi rd, rs1, imm**

  Set register rd to the value of rs1 plus the sign-extended immediate imm.
  Example: If x17 = 0x00000005 and imm = 3, then addi x12, x17, 3 will set x12 to 0x00000008.

- **andi rd, rs1, imm**

  Set register rd to the bitwise AND of rs1 and the sign-extended immediate imm.
  Example: If x17 = 0x55551111, then andi x12, x17, 0x0ff will set x12 to 0x00000011.

- **jalr rd, imm(rs1)**

  Set register rd to the address of the next instruction (address of jalr + 4), then jump to the address rs1 + imm with the least significant bit of the result forced to 0.

- **lb rd, imm(rs1)**

  Load a sign-extended byte from the address rs1 + imm and store it in rd.
  Example: If x13 = 0x00002650, then lb x12, 1(x13) will set x12 to 0xffffff80.

- **lbu rd, imm(rs1)**

  Load a zero-extended byte from the address rs1 + imm and store it in rd.
  Example: If x13 = 0x00002650, then lbu x12, 1(x13) will set x12 to 0x00000080.

- **lh rd, imm(rs1)**
  Load a sign-extended 16-bit half-word from the little-endian memory at rs1 + imm and store it in rd.
  If x13 = 0x00002650, then lh x12, -2(x13) will set x12 to 0x00004307.

- **lhu rd, imm(rs1)**
  Load a zero-extended 16-bit half-word from the little-endian memory at rs1 + imm and store it in rd.
  Example: If x13 = 0x00002650, then lhu x12, -2(x13) will set x12 to 0x00004307.

- **lw rd, imm(rs1)**
  Load a sign-extended 32-bit word from the little-endian memory at rs1 + imm and store it in rd.
  Example: If x13 = 0x00002650, then lw x12, -4(x13) will set x12 to 0x4307a503.

- **ori rd, rs1, imm**
  Set register rd to the bitwise OR of rs1 and the sign-extended immediate imm.
  Example: If x17 = 0x55551111, then ori x12, x17, 0x0ff will set x12 to 0x555511ff.

- **slli rd, rs1, imm**

  Shift the value in rs1 left by the number of bits specified in shamt (shift amount), and store the result in rd.
  Example: If x17 = 0x12345678, then slli x12, x17, 4 will set x12 to 0x23456780.

- **slti rd, rs1, imm**

  If the signed integer in rs1 is less than the signed immediate imm, set rd to 1. Otherwise, set rd to 0.

- **sltiu rd, rs1, imm**

  If the unsigned integer in rs1 is less than the unsigned immediate imm, set rd to 1. Otherwise, set rd to 0.
  Note: Although imm is sign-extended during decoding, it is treated as unsigned for the comparison.

- **srai rd, rs1, imm**

  Perform an arithmetic (sign-preserving) right shift on rs1 by the number of bits in shamt and store the result in rd.
  Example: If x17 = 0x87654321, then srai x12, x17, 4 will set x12 to 0xf8765432.
  Note: This instruction requires bit 30 = 1.

- **srli rd, rs1, imm**

  Perform a logical (zero-fill) right shift on rs1 by the number of bits in shamt and store the result in rd.
  Example: If x17 = 0x87654321, then srli x12, x17, 4 will set x12 to 0x08765432.
  Note: This instruction requires bit 30 = 0.

- **xori rd, rs1, imm**

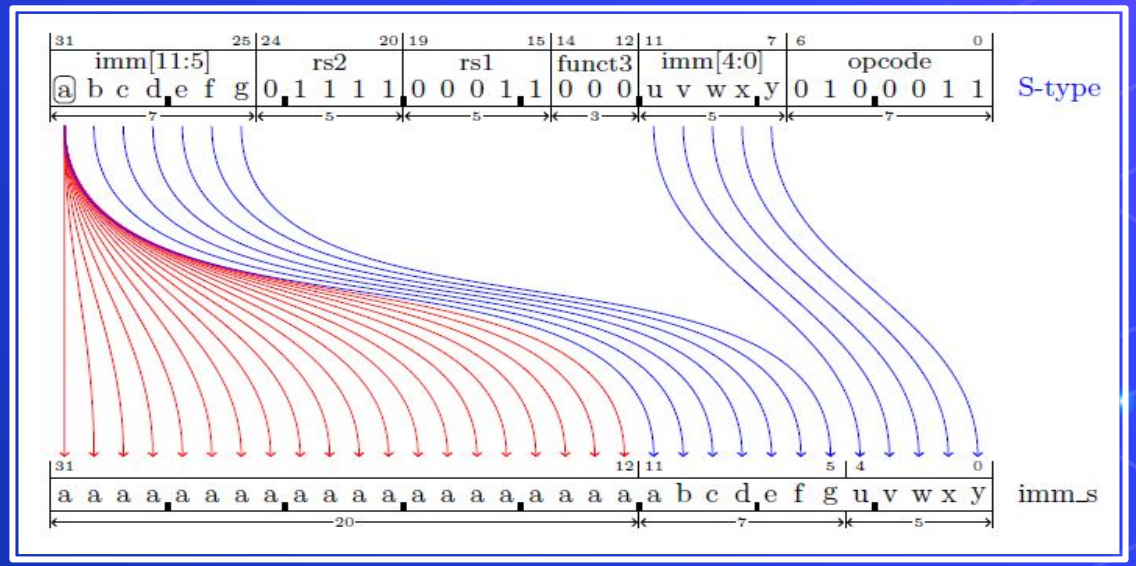
  Set rd to the bitwise XOR of rs1 and the sign-extended immediate imm.

  For example, If x17 = 0x55551111, then xori x12, x17, 0x0ff will set x12 to 0x555511ee.

S-TYPE

The S-type instruction format is used to encode instructions with a signed 12-bit immediate operand with a range of [-2048...2047], an rs1 register, and an rs2 register.

If XLEN=32 then the 12 bit imm value will extracted from the instruction and converted to form the imm_s value.



- **sb rs2,imm(rs1)**
  Set the byte of memory at the address given by the sum of rs1 and imm_s to the 8 LSBs of rs2.

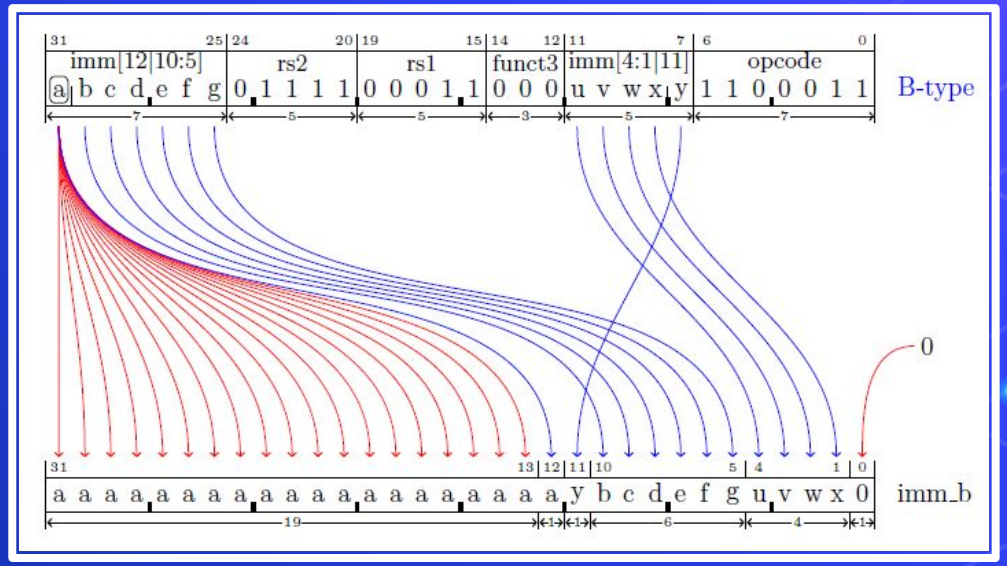
- **sh rs2,imm(rs1)**
  Set the 16-bit half-word of memory at the address given by the sum of rs1 and imm_s to the 16 LSBs of rs2.

- **sw rs2,imm(rs1)**
  Store the 32-bit value in rs2 into the memory at the address given by the sum of rs1 and imm_s.

B-TYPE

The B-type instruction format is used for branch instructions that require an even immediate value that is used to determine the branch target address as an o
set from the current instruction address.

If XLEN=32 then the 12 bit imm value will extracted from the instruction and converted to form the imm_b value.



- **beq rs1,rs2,pcrel 13**
  If rs1 is equal to rs2 then add imm_b to the pc register.
- **bge rs1,rs2,pcrel 13**
  If the signed value in rs1 is greater than or equal to the signed value in rs2 then add imm_b to the pc register.
- **bgeu rs1,rs2,pcrel 13**
  If the unsigned value in rs1 is greater than or equal to the unsigned value in rs2 then add imm_b to the pc register.
- **blt rs1,rs2,pcrel 13**
  If the signed value in rs1 is less than the signed value in rs2 then add imm_b to the pc register.
- **bltu rs1,rs2,pcrel 13**
  If the unsigned value in rs1 is less than the unsigned value in rs2 then add imm_b to the pc register.
- **bne rs1,rs2,pcrel 13**
  If rs1 is not equal to rs2 then add imm_b to the pc register.

Instruction		Description	Operation	Type	31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
					funct7			funct3		opcode
lui	rd,imm	Load Upper Immediate	rd ← imm_u, pc ← pc+4	U	imm[31:12]				rd	0 1 1 0 1 1 1
auipc	rd,imm	Add Upper Immediate to PC	rd ← pc + imm_u, pc ← pc+4	U	imm[31:12]				rd	0 0 1 0 1 1 1
jal	rd,pcrel_21	Jump And Link	rd ← pc+4, pc ← pc+imm_j	J	imm[20|10:1|11|19:12]				rd	1 1 0 1 1 1 1
jalr	rd,imm(rs1)	Jump And Link Register	rd ← pc+4, pc ← (rs1+imm_i) ∧ ~1	I	imm[11:0]		rs1	0 0 0	rd	1 1 0 0 1 1 1
beq	rs1,rs2,pcrel_13	Branch Equal	pc ← pc + ((rs1==rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	0 0 0	imm[4:1|11]	1 1 0 0 0 1 1
bne	rs1,rs2,pcrel_13	Branch Not Equal	pc ← pc + ((rs1!=rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	0 0 1	imm[4:1|11]	1 1 0 0 0 1 1
blt	rs1,rs2,pcrel_13	Branch Less Than	pc ← pc + ((rs1<rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	1 0 0	imm[4:1|11]	1 1 0 0 0 1 1
bge	rs1,rs2,pcrel_13	Branch Greater or Equal	pc ← pc + ((rs1>=rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	1 0 1	imm[4:1|11]	1 1 0 0 0 1 1
bltu	rs1,rs2,pcrel_13	Branch Less Than Unsigned	pc ← pc + ((rs1<rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	1 1 0	imm[4:1|11]	1 1 0 0 0 1 1
bgeu	rs1,rs2,pcrel_13	Branch Greater or Equal Unsigned	pc ← pc + ((rs1>=rs2) ? imm_b : 4)	B	imm[12|10:5]	rs2	rs1	1 1 1	imm[4:1|11]	1 1 0 0 0 1 1
lb	rd,imm(rs1)	Load Byte	rd ← sx(m8(rs1+imm_i)), pc ← pc+4	I	imm[11:0]		rs1	0 0 0	rd	0 0 0 0 0 1 1
lh	rd,imm(rs1)	Load Halfword	rd ← sx(m16(rs1+imm_i)), pc ← pc+4	I	imm[11:0]		rs1	0 0 1	rd	0 0 0 0 0 1 1
lw	rd,imm(rs1)	Load Word	rd ← sx(m32(rs1+imm_i)), pc ← pc+4	I	imm[11:0]		rs1	0 1 0	rd	0 0 0 0 0 1 1
lbu	rd,imm(rs1)	Load Byte Unsigned	rd ← zx(m8(rs1+imm_i)), pc ← pc+4	I	imm[11:0]		rs1	1 0 0	rd	0 0 0 0 0 1 1
lhu	rd,imm(rs1)	Load Halfword Unsigned	rd ← zx(m16(rs1+imm_i)), pc ← pc+4	I	imm[11:0]		rs1	1 0 1	rd	0 0 0 0 0 1 1
sb	rs2,imm(rs1)	Store Byte	m8(rs1+imm_s) ← rs2[7:0], pc ← pc+4	S	imm[11:5]	rs2	rs1	0 0 0	imm[4:0]	0 1 0 0 0 1 1
sh	rs2,imm(rs1)	Store Halfword	m16(rs1+imm_s) ← rs2[15:0], pc ← pc+4	S	imm[11:5]	rs2	rs1	0 0 1	imm[4:0]	0 1 0 0 0 1 1
sw	rs2,imm(rs1)	Store Word	m32(rs1+imm_s) ← rs2[31:0], pc ← pc+4	S	imm[11:5]	rs2	rs1	0 1 0	imm[4:0]	0 1 0 0 0 1 1
addi	rd,rs1,imm	Add Immediate	rd ← rs1 + imm_i, pc ← pc+4	I	imm[11:0]		rs1	0 0 0	rd	0 0 1 0 0 1 1
slti	rd,rs1,imm	Set Less Than Immediate	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4	I	imm[11:0]		rs1	0 1 0	rd	0 0 1 0 0 1 1
sltiu	rd,rs1,imm	Set Less Than Immediate Unsigned	rd ← (rs1 < imm_i) ? 1 : 0, pc ← pc+4	I	imm[11:0]		rs1	0 1 1	rd	0 0 1 0 0 1 1
xori	rd,rs1,imm	Exclusive Or Immediate	rd ← rs1 ⊕ imm_i, pc ← pc+4	I	imm[11:0]		rs1	1 0 0	rd	0 0 1 0 0 1 1
ori	rd,rs1,imm	Or Immediate	rd ← rs1 ∨ imm_i, pc ← pc+4	I	imm[11:0]		rs1	1 1 0	rd	0 0 1 0 0 1 1
andi	rd,rs1,imm	And Immediate	rd ← rs1 ∧ imm_i, pc ← pc+4	I	imm[11:0]		rs1	1 1 1	rd	0 0 1 0 0 1 1
slli	rd,rs1,shamt	Shift Left Logical Immediate	rd ← rs1 << shamt_i, pc ← pc+4	I	0 0 0 0 0 0 0	shamt	rs1	0 0 1	rd	0 0 1 0 0 1 1
srli	rd,rs1,shamt	Shift Right Logical Immediate	rd ← rs1 >> shamt_i, pc ← pc+4	I	0 0 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1
srai	rd,rs1,shamt	Shift Right Arithmetic Immediate	rd ← rs1 >> shamt_i, pc ← pc+4	I	0 1 0 0 0 0 0	shamt	rs1	1 0 1	rd	0 0 1 0 0 1 1
add	rd,rs1,rs2	Add	rd ← rs1 + rs2, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1
sub	rd,rs1,rs2	Subtract	rd ← rs1 - rs2, pc ← pc+4	R	0 1 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1
sll	rd,rs1,rs2	Shift Left Logical	rd ← rs1 << (rs2%XLEN), pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	0 0 1	rd	0 1 1 0 0 1 1
slt	rd,rs1,rs2	Set Less Than	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	0 1 0	rd	0 1 1 0 0 1 1
sltu	rd,rs1,rs2	Set Less Than Unsigned	rd ← (rs1 < rs2) ? 1 : 0, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	0 1 1	rd	0 1 1 0 0 1 1
xor	rd,rs1,rs2	Exclusive Or	rd ← rs1 ⊕ rs2, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1
srl	rd,rs1,rs2	Shift Right Logical	rd ← rs1 >> (rs2%XLEN), pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1
sra	rd,rs1,rs2	Shift Right Arithmetic	rd ← rs1 >> (rs2%XLEN), pc ← pc+4	R	0 1 0 0 0 0 0	rs2	rs1	1 0 1	rd	0 1 1 0 0 1 1
or	rd,rs1,rs2	Or	rd ← rs1 ∨ rs2, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1
and	rd,rs1,rs2	And	rd ← rs1 ∧ rs2, pc ← pc+4	R	0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1
ecall		Trap to Debugger		I	0 0 0 0 0 0 0 0 0 0 0 0		0 0 0 0 0	0 0 0	0 0 0 0 0	1 1 1 0 0 1 1
ebreak		Trap to Operating System		I	0 0 0 0 0 0 0 0 0 0 0 1		0 0 0 0 0	0 0 0	0 0 0 0 0	1 1 1 0 0 1 1
csrrw	rd,csr,rs1	Atomic Read/Write	rd ← csr, csr ← rs1, pc ← pc+4	I	csr[11:0]		rs1	0 0 1	rd	1 1 1 0 0 1 1
csrrs	rd,csr,rs1	Atomic Read and Set	rd ← csr, csr ← csr ∨ rs1, pc ← pc+4	I	csr[11:0]		rs1	0 1 0	rd	1 1 1 0 0 1 1
csrrc	rd,csr,rs1	Atomic Read and Clear	rd ← csr, csr ← csr ∧ ~rs1, pc ← pc+4	I	csr[11:0]		rs1	0 1 1	rd	1 1 1 0 0 1 1
csrrwi	rd,csr,zimm	Atomic Read/Write Immediate	rd ← csr, csr ← zimm, pc ← pc+4	I	csr[11:0]		zimm[4:0]	1 0 1	rd	1 1 1 0 0 1 1
csrrsi	rd,csr,zimm	Atomic Read and Set Immediate	rd ← csr, csr ← csr ∨ zimm, pc ← pc+4	I	csr[11:0]		zimm[4:0]	1 1 0	rd	1 1 1 0 0 1 1
csrrci	rd,csr,zimm	Atomic Read and Clear Immediate	rd ← csr, csr ← csr ∧ ~zimm, pc ← pc+4	I	csr[11:0]		zimm[4:0]	1 1 1	rd	1 1 1 0 0 1 1

IMPLEMENTATION OF SINGLE CYCLE RISC-V PROCESSOR

To implement a single-cycle RISC-V processor, I utilize Hardware Description Language (HDL), specifically SystemVerilog, due to its robust capabilities and flexibility for designing complex digital systems.

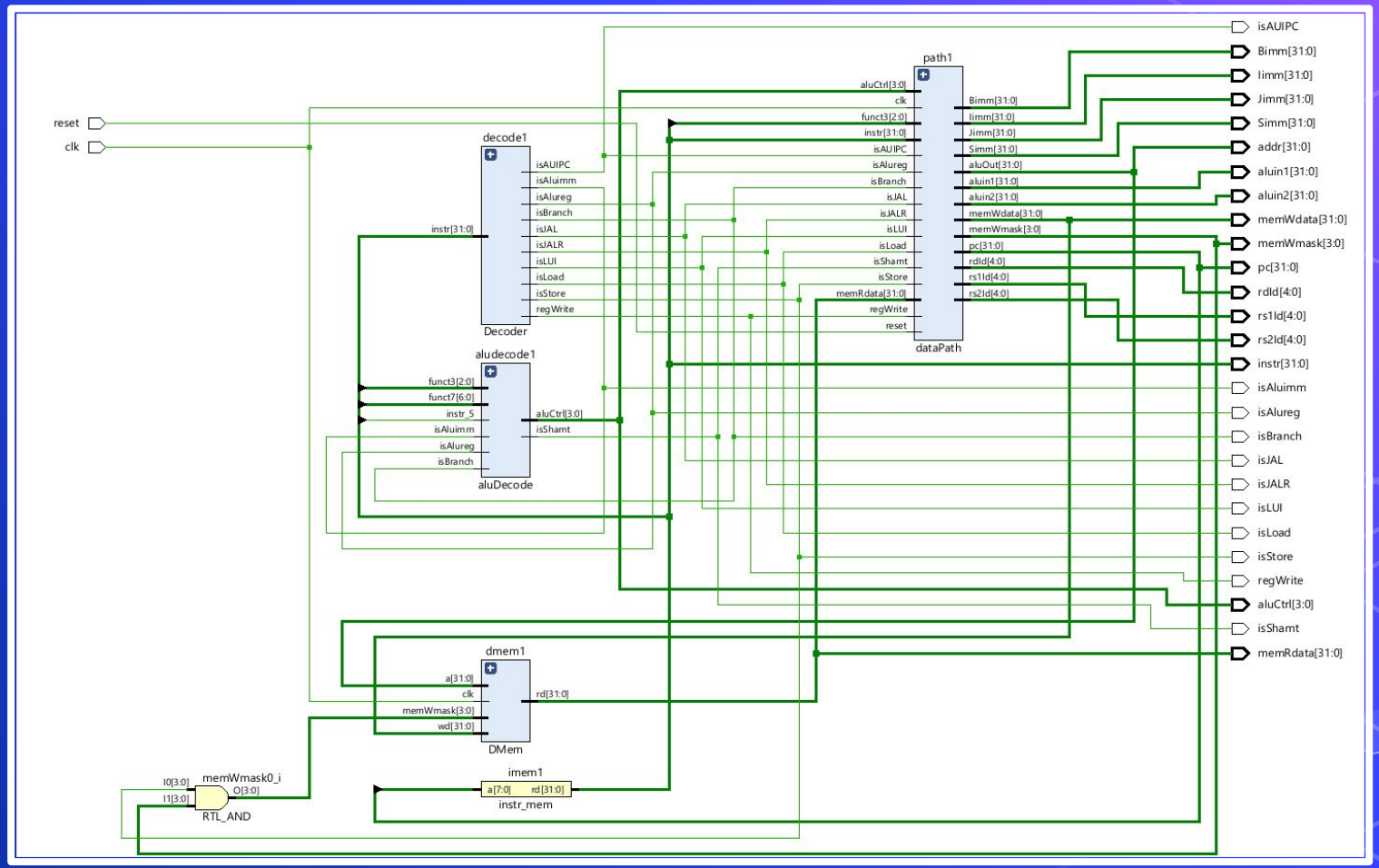
For simulation and hardware implementation, I rely on **Xilinx Vivado**, a comprehensive design suite that supports synthesis, simulation, and bitstream generation.

To develop and test assembly programs, I use the **RISC-V Toolchain** along with **rvddt (RISC-V Data Dump Tool)**. These tools enable me to write RISC-V assembly code and generate the corresponding machine code, which is essential for simulating and validating the processor's functionality.

VERILOG CODE OF RISC–V TOP MODULE

```
module RISCV(
  input logic clk, reset,
  output logic [31:0] pc ,instr, memWdata, addr, aluin1, aluin2, Simm, Jimm, Bimm, Iimm, memRdata,
  output logic [4:0] rs1Id, rs2Id, rdId,
  output logic [3:0] memWmask, aluCtrl,
  output logic isAlureg, regWrite, isJAL, isJALR, isBranch, isLUI, isAUIPC, isAluimm,isLoad, isStore, isShamt
   );
   logic [2:0] funct3;
   logic [6:0] funct7;
   logic isZero;
Decoder decode1 (instr,isAlureg, isAluimm, regWrite, isJAL, isJALR, isBranch, isLUI, isAUIPC,isLoad,
isStore);
aluDecode aludecode1 (funct3,funct7,instr[5], isBranch, isAlureg, isAluimm,aluCtrl,isShamt);
dataPath path1 (clk, reset,isAlureg, regWrite, isJAL, isJALR, isBranch, isLUI, isAUIPC, isLoad, isStore,
isShamt,funct3,aluCtrl,instr, memRdata,pc, addr, memWdata, aluin1, aluin2, Simm, Bimm, Jimm,
Iimm,rs1Id, rs2Id, rdId, memWmask,isZero );
dist_mem_gen_0 imem1( .a(pc[9:2]),.spo(instr) );
 DMem dmem1 (clk,{{4{isStore}} & memWmask}, addr, memWdata,memRdata);
assign funct3 = instr[14:12];
assign funct7 = instr[31:25];
endmodule
```

SCHEMATIC OF SINGLE CYCLE RISC-V



VERILOG CODE OF RISC–V DATAPATH MODULE

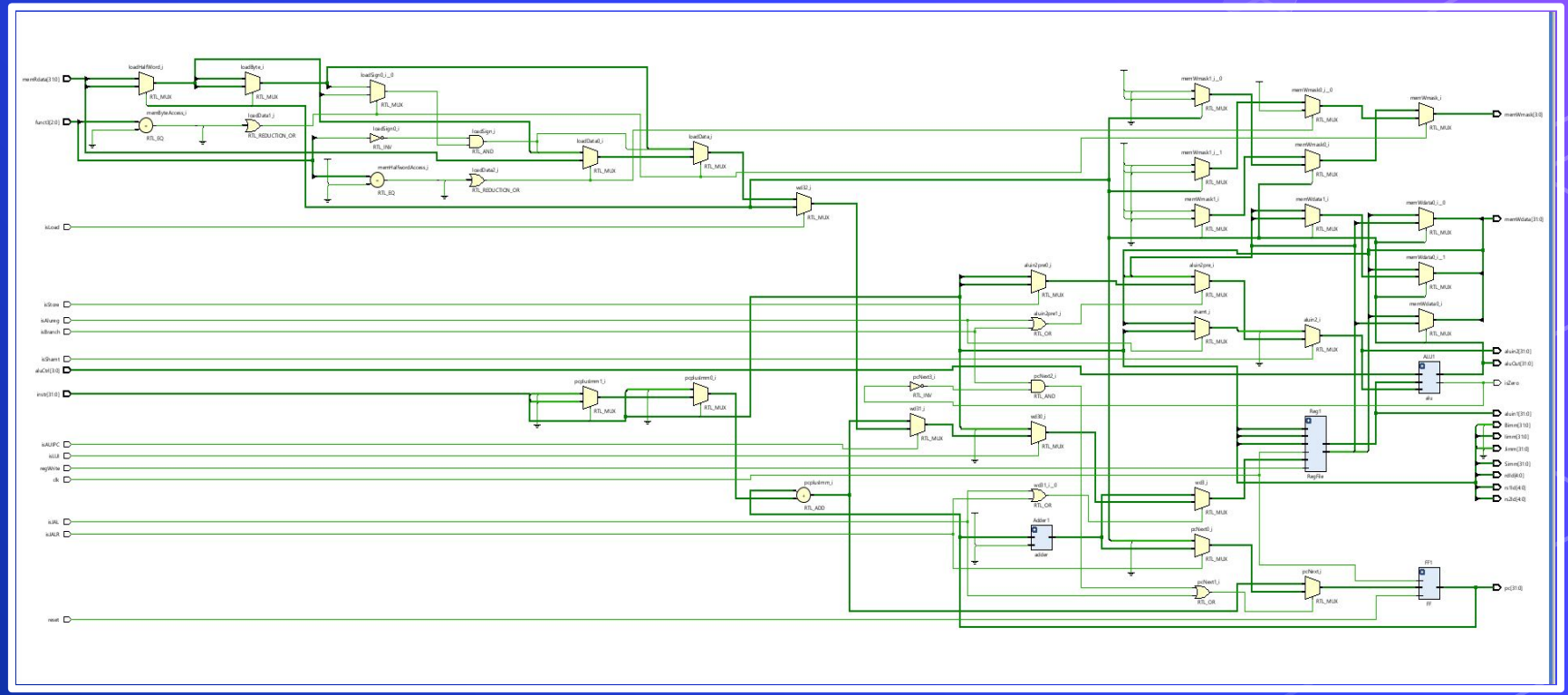
```
module dataPath(
 input logic clk, reset,
 input logic isAlureg, regWrite, isJAL, isJALR, isBranch, isLUI, isAUIPC, isLoad, isStore, isShamt,
 input logic [2:0] funct3,
 input logic [3:0] aluCtrl,
 input logic [31:0] instr, memRdata,
 output logic [31:0] pc, aluOut, memWdata, aluin1, aluin2, Simm, Bimm, Jimm, Iimm,
 output logic [4:0] rs1Id, rs2Id, rdId,
 output logic [3:0] memWmask,
 output logic isZero
   );
   logic [31:0] pcNext, pcplus4,rd2,aluin2pre,pcplusImm,wd3,loadData;
   logic[ 1:0] memByteAccess, memHalfwordAccess;
   logic [15:0] loadHalfWord;
   logic [7:0] loadByte;
   logic loadSign;
   logic [31:0] Uimm;
   logic [4:0] shamt;
   FF FF1 (.clk(clk),.reset(reset),.d(pcNext),.q(pc));
   adder Adder1 ( .ip1(pc), .ip2(32'b100), .op(pcplus4));
   RegFile Reg1(.clk(clk),.we3(regWrite),.a1(rs1Id),.a2(rs2Id),.a3(rdId),.wd3(wd3),.rd1(aluin1),.rd2(rd2));
   alu ALU1(.aluCtrl(aluCtrl),.op1(aluin1), .op2(aluin2),.aluOut(aluOut),.isZero(isZero));
 assign memByteAccess = funct3[1:0] == 2'b00;
 assign memHalfwordAccess = funct3[1:0] == 2'b01;
```

VERILOG CODE OF RISC–V DATAPATH MODULE

```
assign loadHalfWord = aluOut[1] ? memRdata[31:16] : memRdata[15:0];
 assign loadByte = aluOut[0] ? loadHalfWord[15:8] : loadHalfWord[7:0];
 assign loadData = memByteAccess ? {{24{loadSign}}, loadByte} : memHalfwordAccess ? {{16{loadSign}},
                                                                                    loadHalfWord} : memRdata;
 assign memWdata[ 7: 0] = rd2[7:0];
 assign memWdata[15: 8] = aluOut[0] ? rd2[7:0]  : rd2[15: 8];
 assign memWdata[23:16] = aluOut[1] ? rd2[7:0]  : rd2[23:16];
 assign memWdata[31:24] = aluOut[0] ? rd2[7:0]  : aluOut[1] ? rd2[15:8] : rd2[31:24];
 assign memWmask = memByteAccess ? (aluOut[1] ? (aluOut[0] ? 4'b1000 : 4'b0100) : (aluOut[0] ? 4'b0010 : 4'b0001)) :
memHalfwordAccess ?(aluOut[1] ? 4'b1100 : 4'b0011) : 4'b1111;
 assign pcplusImm = pc + (instr[3] ? Jimm[31:0] : instr[4] ? Uimm[31:0] : Bimm[31:0]);
 assign aluin2pre =  (isAlureg | isBranch) ? rd2 :isStore ? Simm :Iimm;
 assign aluin2 = isShamt ?{27'b0 , shamt} : aluin2pre;
 assign pcNext = (isBranch && !isZero || isJAL) ? pcplusImm :isJALR ? {aluOut[31:1],1'b0}:pcplus4;
 assign wd3 = (isJAL || isJALR) ? pcplus4 :isLUI ? Uimm :isAUIPC ? pcplusImm :isLoad ? loadData :aluOut;
assign rs1Id = instr[19:15];
  assign rs2Id = instr[24:20];
  assign rdId  = instr[11:7];
assign Uimm = {instr[31], instr[30:12], {12{1'b0}}};
  assign Iimm = {{21{instr[31]}}, instr[30:20]};
  assign Simm = {{21{instr[31]}}, instr[30:25],instr[11:7]};
  assign Bimm = {{20{instr[31]}}, instr[7],instr[30:25],instr[11:8],1'b0};
  assign Jimm = {{12{instr[31]}}, instr[19:12],instr[20],instr[30:21],1'b0};
assign shamt = isAlureg ? rd2[4:0] : instr[24:20];
  assign loadSign = !funct3[2] & (memByteAccess ? loadByte[7] : loadHalfWord[15]);
endmodule
```

** Cont

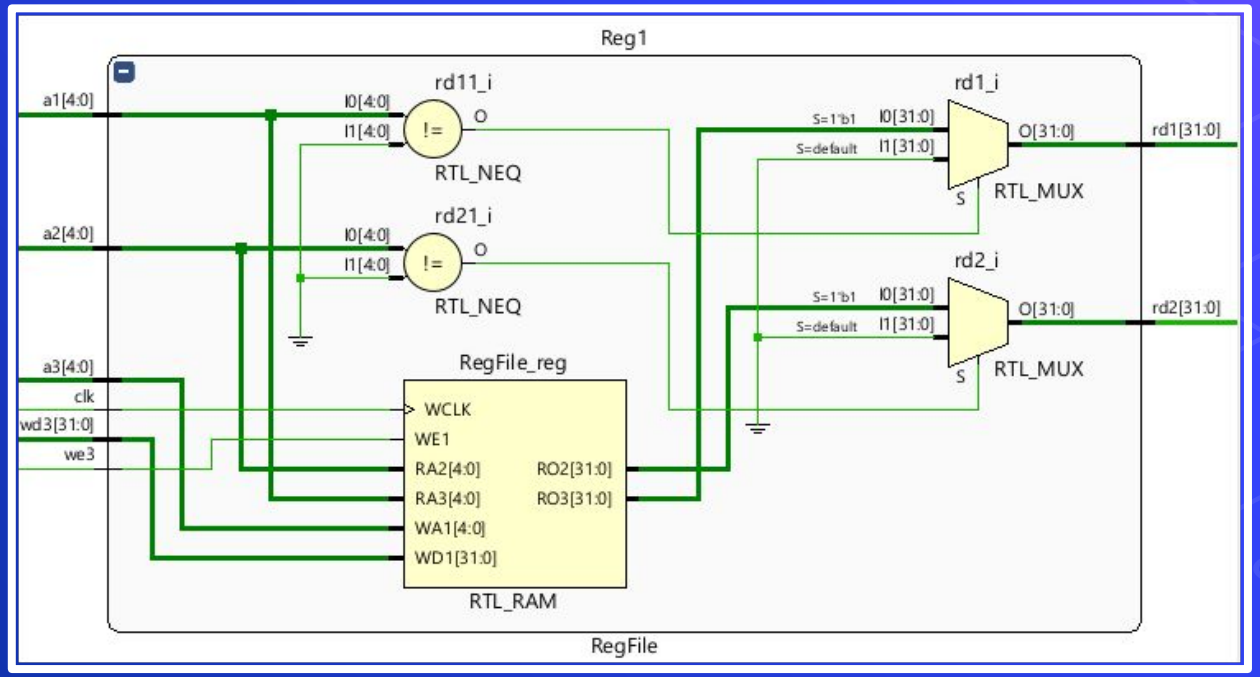
SCHEMATIC OF DATAPATH OF SINGLE CYCLE RISC-V

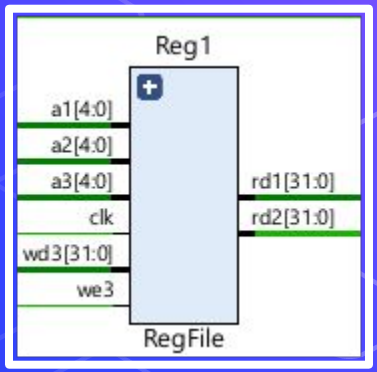


VERILOG CODE OF RISC–V REGISTER FILE MODULE

```
module RegFile(

input logic clk,
input logic we3,
input logic [4:0] a1,a2,a3,
input logic [31:0] wd3,
output logic [31:0] rd1,rd2
  );

  logic [31:0] RegFile [31:0];

  always_ff @ (posedge clk)
  if(we3) RegFile[a3] <= wd3 ; // as x0 register is hardwired to zero

 assign rd1 = (a1 != 32'b0) ? RegFile[a1] : 32'b0;
 assign rd2 = (a2 != 32'b0) ? RegFile[a2] : 32'b0;

endmodule
```

SCHEMATIC OF REGISTER FILE OF SINGLE CYCLE RISC-V

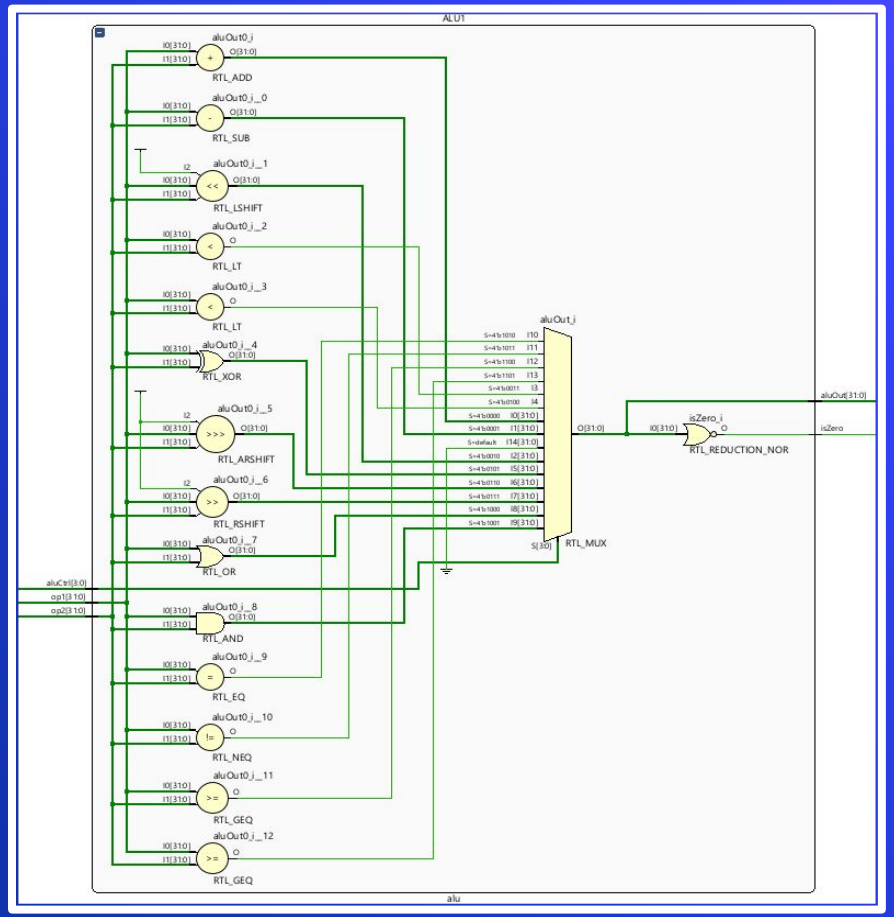


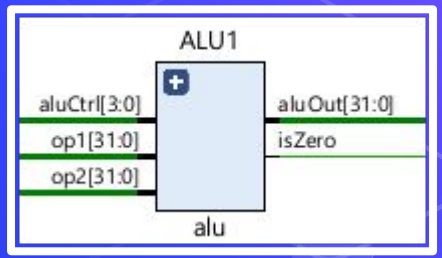


VERILOG CODE OF RISC–V ALU MODULE

```
module alu(
  input logic [3:0] aluCtrl,
  input logic [31:0] op1, op2,
  output logic [31:0] aluOut,
  output logic isZero
   );
   always_comb
     case(aluCtrl)
          4'h0: aluOut <= op1 + op2;
          4'h1: aluOut <= op1 - op2;
          4'h2: aluOut <= op1 << op2;
          4'h3: aluOut <= $signed(op1) < $signed(op2);
          4'h4: aluOut <= op1 < op2;
          4'h5: aluOut <= op1 ^ op2;
          4'h6: aluOut <= $signed(op1) >>> op2;
          4'h7: aluOut <= op1 >> op2;
          4'h8: aluOut <= op1 | op2;
          4'h9: aluOut <= op1 & op2;
          4'ha: aluOut <= op1 == op2;
          4'hb: aluOut <= op1 != op2;
          4'hc: aluOut <= $signed(op1) >= $signed(op2);
          4'hd: aluOut <= op1 >= op2;
          default: aluOut <= 31'b0;
       endcase
    assign isZero = ~|aluOut;
endmodule
```

SCHEMATIC OF ALU OF SINGLE CYCLE RISC-V





VERILOG CODE OF RISC–V ALU DECODER MODULE

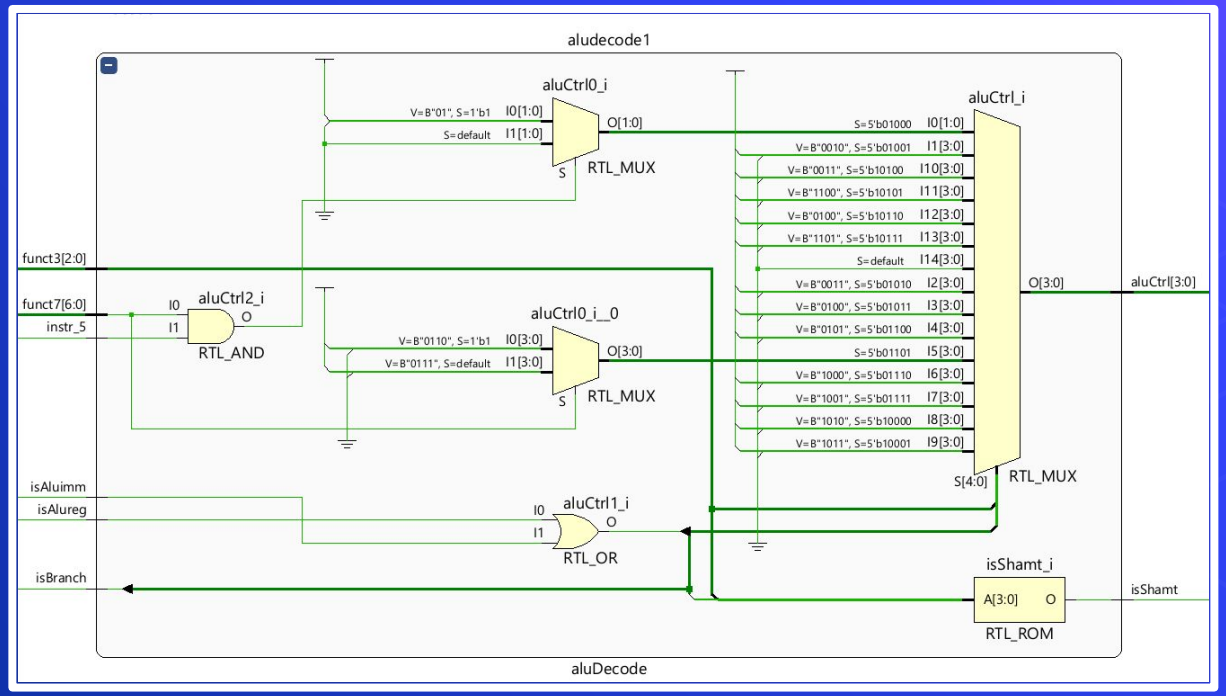
```
module aluDecode(
  input logic [2:0] funct3,
  input logic [6:0] funct7,
  input logic instr_5, isBranch, isAlureg, isAluimm,
  output logic [3:0] aluCtrl,
  output logic isShamt
   );
   always_comb begin
   case({isBranch, isAlureg || isAluimm, funct3})
          5'b01000: aluCtrl <= (funct7[5] & instr_5) ? (4'h1) : (4'h0);
          5'b01001: aluCtrl <= 4'h2;
          5'b01010: aluCtrl <= 4'h3;
          5'b01011: aluCtrl <= 4'h4;
          5'b01100: aluCtrl <= 4'h5;
          5'b01101: aluCtrl <= funct7[5]? 4'h6 : 4'h7;
          5'b01110: aluCtrl <= 4'h8;
          5'b01111: aluCtrl <= 4'h9;
          5'b10000: aluCtrl <= 4'ha;
          5'b10001: aluCtrl <= 4'hb;
          5'b10100: aluCtrl <= 4'h3;
          5'b10101: aluCtrl <= 4'hc;
```

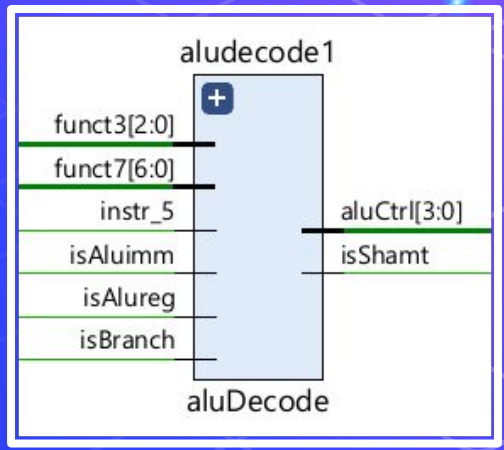
VERILOG CODE OF RISC–V ALU DECODER MODULE

```
        5'b10110: aluCtrl <= 4'h4;
        5'b10111: aluCtrl <= 4'hd;
        default: aluCtrl <= 4'h0;  // adder by default
      endcase
  case({isAlureg || isAluimm, funct3})
  4'b1001, 4'b1101 : isShamt <= 1'b1;
  default: isShamt <= 1'b0;
  endcase

    end

endmodule
```

** Cont

SCHEMATIC OF ALU DECODER OF SINGLE CYCLE RISC-V

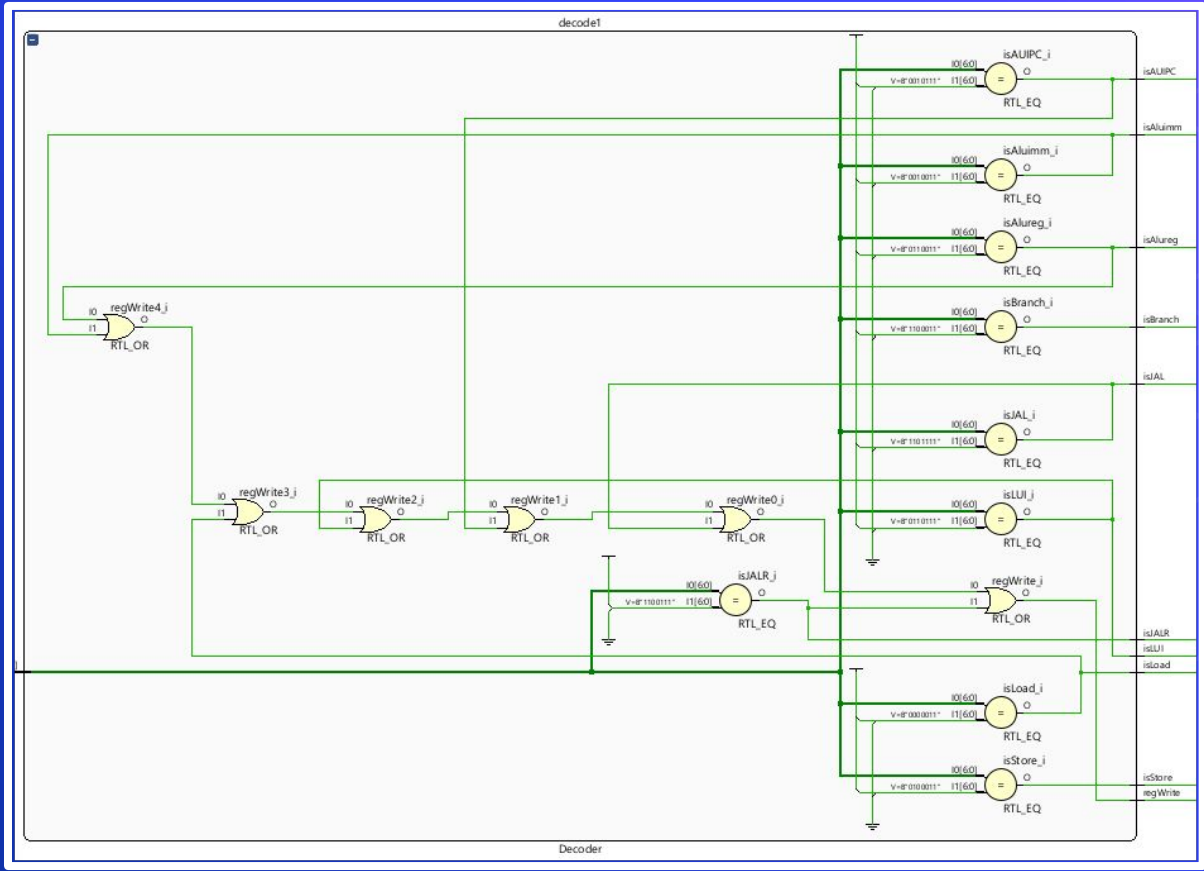


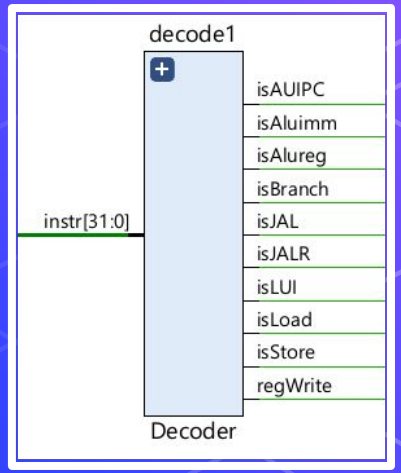


VERILOG CODE OF RISC–V MAIN DECODER MODULE

```
module Decoder(
input logic [31:0] instr,
output logic isAlureg, isAluimm, regWrite, isJAL, isJALR, isBranch, isLUI, isAUIPC,
output logic isLoad, isStore

    );

    assign isAlureg =  (instr[6:0] == 7'b0110011); // rd <- rs1 OP rs2
    assign isAluimm =  (instr[6:0] == 7'b0010011); // rd <- rs1 OP Iimm
    assign isBranch =  (instr[6:0] == 7'b1100011); // if(rs1 OP rs2) PC<-PC+Bimm
    assign isJALR   =  (instr[6:0] == 7'b1100111); // rd <- PC+4; PC<-rs1+Iimm
    assign isJAL    =  (instr[6:0] == 7'b1101111); // rd <- PC+4; PC<-PC+Jimm
    assign isAUIPC  =  (instr[6:0] == 7'b0010111); // rd <- PC + Uimm
    assign isLUI    =  (instr[6:0] == 7'b0110111); // rd <- Uimm
    assign isLoad   =  (instr[6:0] == 7'b0000011); // rd <- mem[rs1+Iimm]
    assign isStore  =  (instr[6:0] == 7'b0100011); // mem[rs1+Simm] <- rs2
    assign regWrite = isAlureg || isAluimm || isLoad || isLUI || isAUIPC || isJAL || isJALR;
endmodule
```

SCHEMATIC OF MAIN DECODER OF SINGLE CYCLE RISC-V

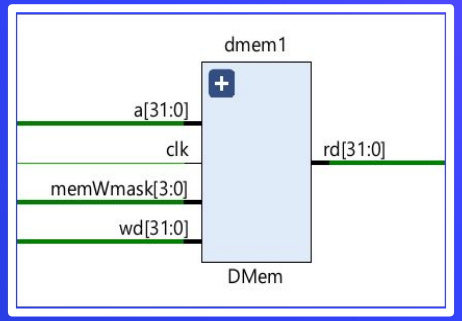


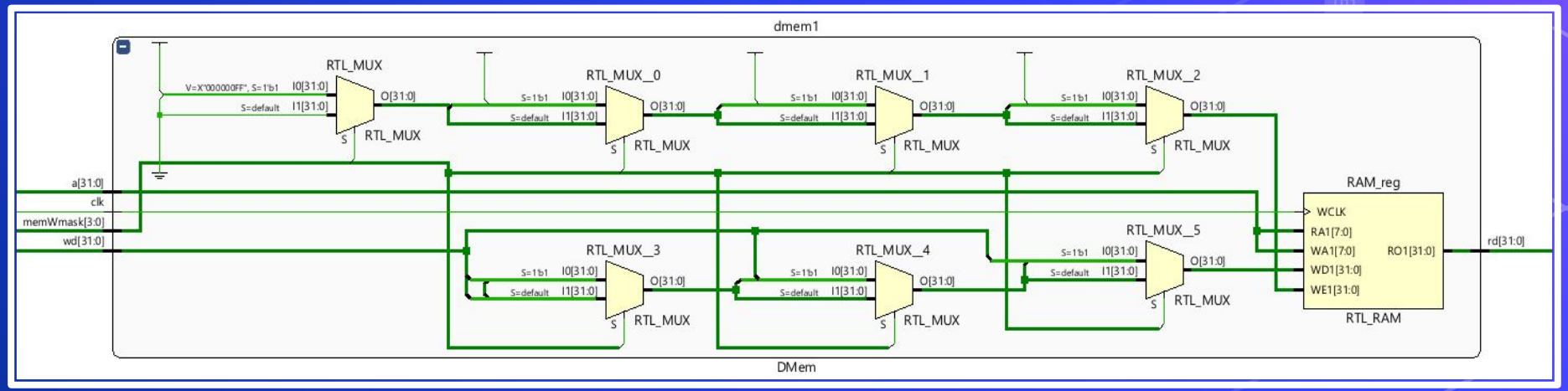


VERILOG CODE OF RISC–V DATA MEMORY MODULE

```
module DMem(
 input logic clk,
 input logic [3:0] memWmask,
 input logic [31:0] a, wd,
 output logic [31:0] rd
    );
    logic [31:0] RAM [255:0] ;
    always_ff @(posedge clk) begin

 if(memWmask[0]) RAM[a[31:2]][7:0] <= wd[7:0];
 if(memWmask[1]) RAM[a[31:2]][15:8] <= wd[15:8];
 if(memWmask[2]) RAM[a[31:2]][23:16] <= wd[23:16];
 if(memWmask[3]) RAM[a[31:2]][31:24] <= wd[31:24];

end
assign rd = RAM[a[31:2]];
endmodule
```

SCHEMATIC OF DATA MEMORY OF SINGLE CYCLE RISC-V





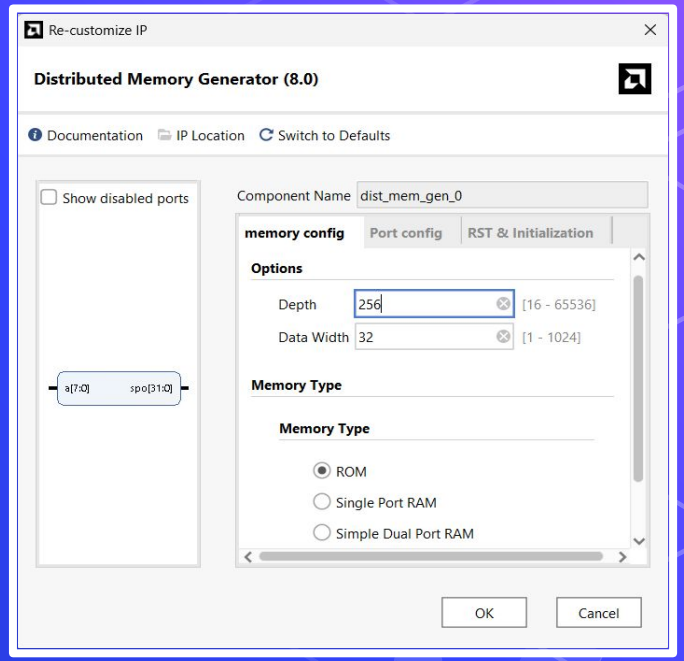
VERILOG CODE OF RISC–V INSTRUCTION MEMORY MODULE

For the Instruction Memory I utilise Vivado IP Generator to generate a Distributed Memory IP (ROM with the specification of Depth 256 and Data Width of 32 bits).

Installation Template

```
dist_mem_gen_0 imem(
  .a(a),      // input wire [7 : 0] a
  .spo(spo)  // output wire [31 : 0] spo
);
```

- Now For initialize ROM/BRAM with Machine Code of the Assembly Program using imem.coe (COEfficient) File .

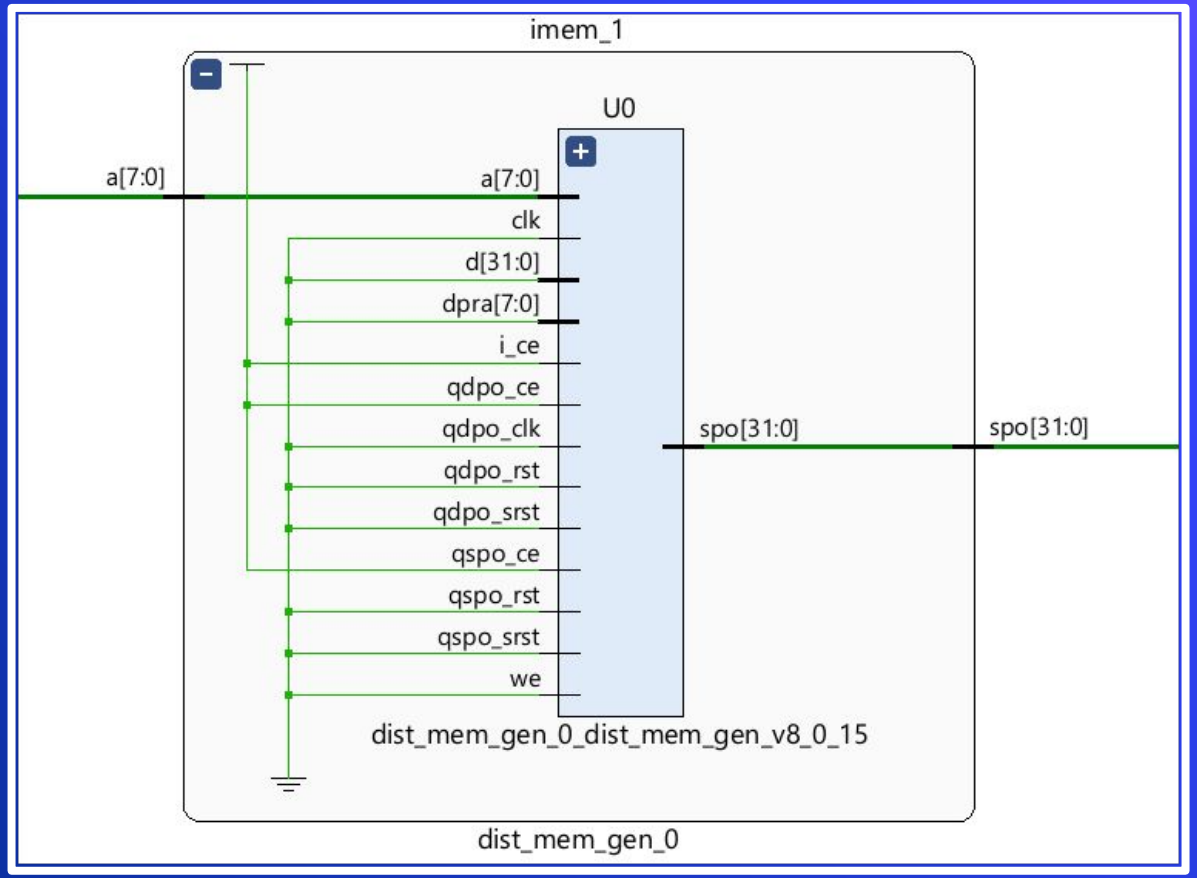


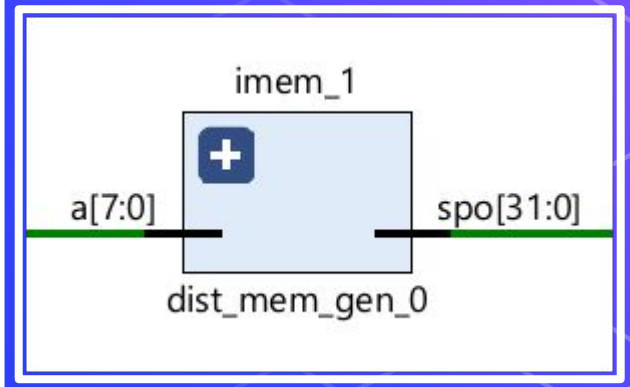
Format of .coe File

```
memory_initialization_radix=16; // means hex values.
memory_initialization_vector=
004001b7,
20000113,
00e00293,
...,
00008067;
```

** 32-bit instructions, one per line, **comma-separated**, **last ends with semicolon**.

SCHEMATIC OF INSTRUCTION MEMORY OF SINGLE CYCLE RISC-V





SIMULATION AND TESTING OF THE PROCESSOR

All of the software presented was assembled/compiled using the **RISC-V GNU toolchain** and executed using the **rvddt simulator** on a Linux (Ubuntu 24.04.1 LTS) operating system.

Steps Of Installing GNU toolchain and rvddt

- In order to install custom code in a location that will not cause interference with other applications these will install the toolchain under a private directory : **~/projects/riscv/install**
- Before building the toolchain, a number of utilities must be present on your system. The following will install those that are needed:

```
sudo apt install autoconf automake autotools - dev curl python3 python - dev libmpc -dev \
libmpfr - dev libgmp -dev gawk build - essential bison flex texinfo gperf \
 libtool patchutils bc zlib1g - dev libexpat - dev
```

- To download, compile and install the toolchain:

```
mkdir -p ~/ projects / riscv
cd ~/ projects / riscv
 git clone https :// github . com / riscv /riscv -gnu - toolchain
 cd riscv -gnu - toolchain
 INS_DIR =~/ projects / riscv / install / rv32i
 ./ configure -- prefix = $INS_DIR \
 --with - multilib - generator ="rv32i -ilp32 --; rv32imafd -ilp32 --; rv32ima -ilp32 --"
make
```

SIMULATION AND TESTING OF THE PROCESSOR

- After building the toolchain, make it available 1855 by putting it into your PATH by adding the following to the end of your .bashrc file: export PATH = $PATH : $INS_DIR

- Download and install the rvddt simulator by executing the following commands. Building the rvddt example programs will verify that the GNU toolchain has been built and installed properly.

```
cd ~/ projects / riscv
git clone https :// github . com / johnwinans / rvddt .git
cd rvddt / src
make world
cd ../ examples
 make world
```

- After building rvddt, make it available by putting it into your PATH by adding the following to the end of your .bashrc file: export PATH = $PATH :~/ projects / riscv / rvddt / src

** Cont

WRITING THE ASSEMBLY PROGRAM

For Testing Purpose I have Written a Assembly Language Code here:

```
.section .text
.globl _start

_start:
    li    t1, 10          # t1 = 10
    li    t2, 5           # t2 = 5
    add   t3, t1, t2      # t3 = 10 + 5 = 15

    sw    t3, 0x20(zero)  # store t3 at memory[0x20]
    sw    t2, 0x24(zero)  # store t2 at memory[0x24]
    sw    t1, 0x28(zero)  # store t1 at memory[0x28]

    lw    t5, 0x20(zero)  # t5 = memory[0x20] = 15
    lw    t6, 0x24(zero)  # t6 = memory[0x24] = 5

    sub   t3, t5, t6      # t3 = 15 - 5 = 10
    beq   t3, t1, match   # if t3 == t1, jump to match
    j     fail            # else go to fail

match:
    addi  t1, t1, 1       # t1 = t1 + 1 = 11
    addi  t2, t2, 2       # t2 = t2 + 2 = 7
    add   t3, t1, t2      # t3 = 11 + 7 = 18
    j     done

fail:
    li    t3, 0           # set t3 to 0

done:
    nop
    j done                # infinite loop to freeze
```

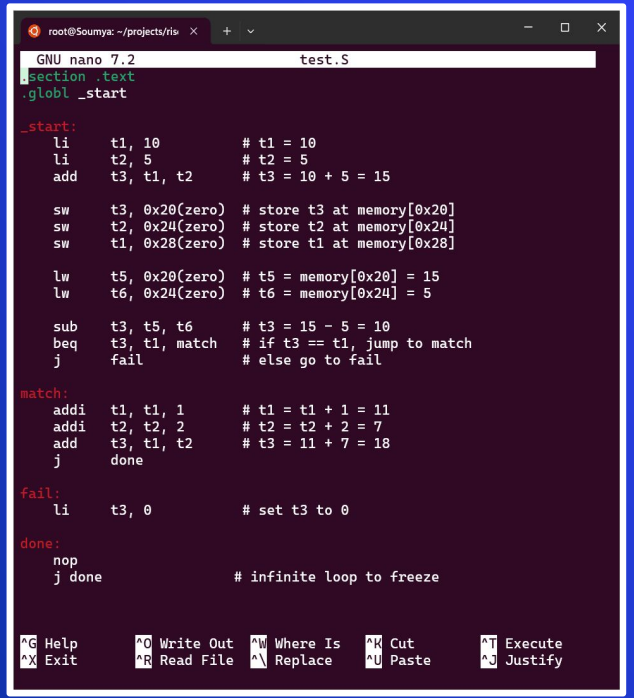
GENERATION OF MACHINE CODE

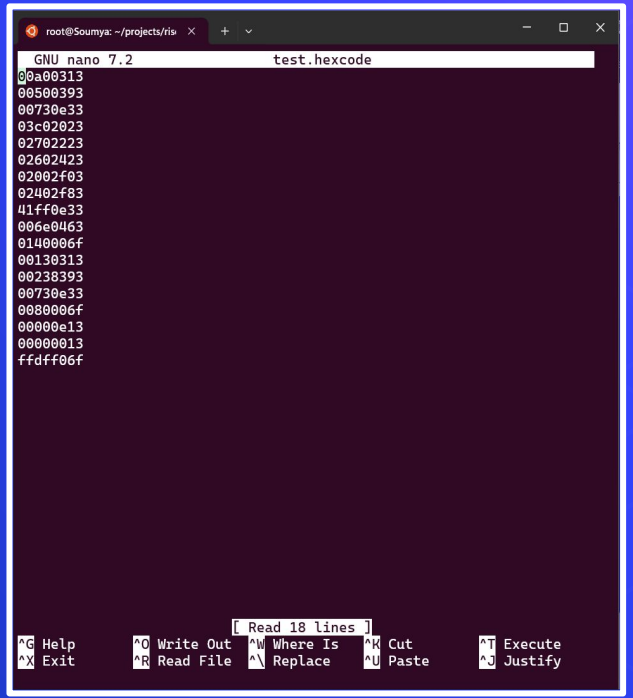
For getting the machine code from the assembly program required following Steps

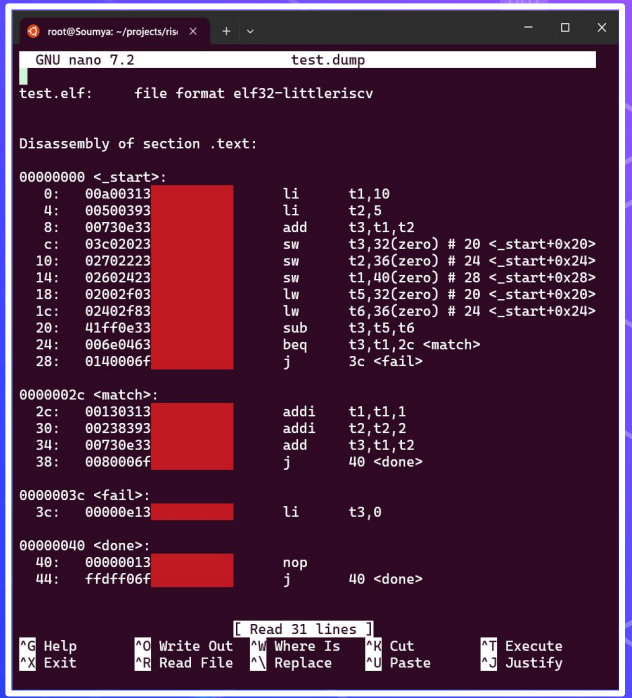
```
root@Soumya: ~/projects/ris
root@Soumya:~/projects/riscv/rvddt/examples# mkdir rv32test
root@Soumya:~/projects/riscv/rvddt/examples# cd rv32test
root@Soumya:~/projects/riscv/rvddt/examples/rv32test# nano test.S
root@Soumya:~/projects/riscv/rvddt/examples/rv32test# riscv32-unknown-elf-gcc -march=rv32i -mabi=ilp32 -nostdlib -Ttext=0x0 -o test.elf test
.S
root@Soumya:~/projects/riscv/rvddt/examples/rv32test# riscv32-unknown-elf-gcc -march=rv32i -mabi=ilp32 -nostdlib -Ttext=0x0 -o test.elf test
.S
riscv32-unknown-elf-objdump -d test.elf > test.dump
riscv32-unknown-elf-objcopy -j .text -O binary test.elf test.bin
riscv32-unknown-elf-objcopy -j .text -O ihex test.elf test.hex
root@Soumya:~/projects/riscv/rvddt/examples/rv32test# nano test.dump
root@Soumya:~/projects/riscv/rvddt/examples/rv32test# riscv32-unknown-elf-objdump -d test.elf | grep "^[[:space:]]*[0-9a-f]\+:" | \
  sed -E 's/^[[:space:]]*([0-9a-f]+):\s*([0-9a-f]+).*/\2/' > test.hexcode
root@Soumya:~/projects/riscv/rvddt/examples/rv32test# nano test.hexcode
```

- Make a new sub directory under the rvddt/example .
- Open a RISC-V assembly source file (.S) file as shown above.
- Write your Assembly Program in shown format, then Save and exit the file.
- Now Compile the Program using the command : riscv32-unknown-elf-gcc -march=rv32i -mabi=ilp32 -nostdlib -Ttext=0x0 -o test.elf test.S
- After compiling to generate machine codes using this command : riscv32-unknown-elf-objdump -d test.elf | grep "^[[:space:]]*[0-9a-f]\+:" | \ sed -E 's/^[[:space:]]*([0-9a-f]+):\s*([0-9a-f]+).*/\2/' > test.hexcode
- The test.hexcode file contains the machine code.
- Copy the contains of the test.hexcode file and generate a .coe file in the shown format.
- Now initialize the instruction rom with the .coe file and start the simulation in the Xilinx Vivado.

GENERATION OF MACHINE CODE







.dump, .hex and .bin files can be generated using the following command :

```
riscv32-unknown-elf-objdump -d test.elf > test.dump
riscv32-unknown-elf-objcopy -j .text -O binary test.elf test.bin
riscv32-unknown-elf-objcopy -j .text -O ihex test.elf test.hex
```

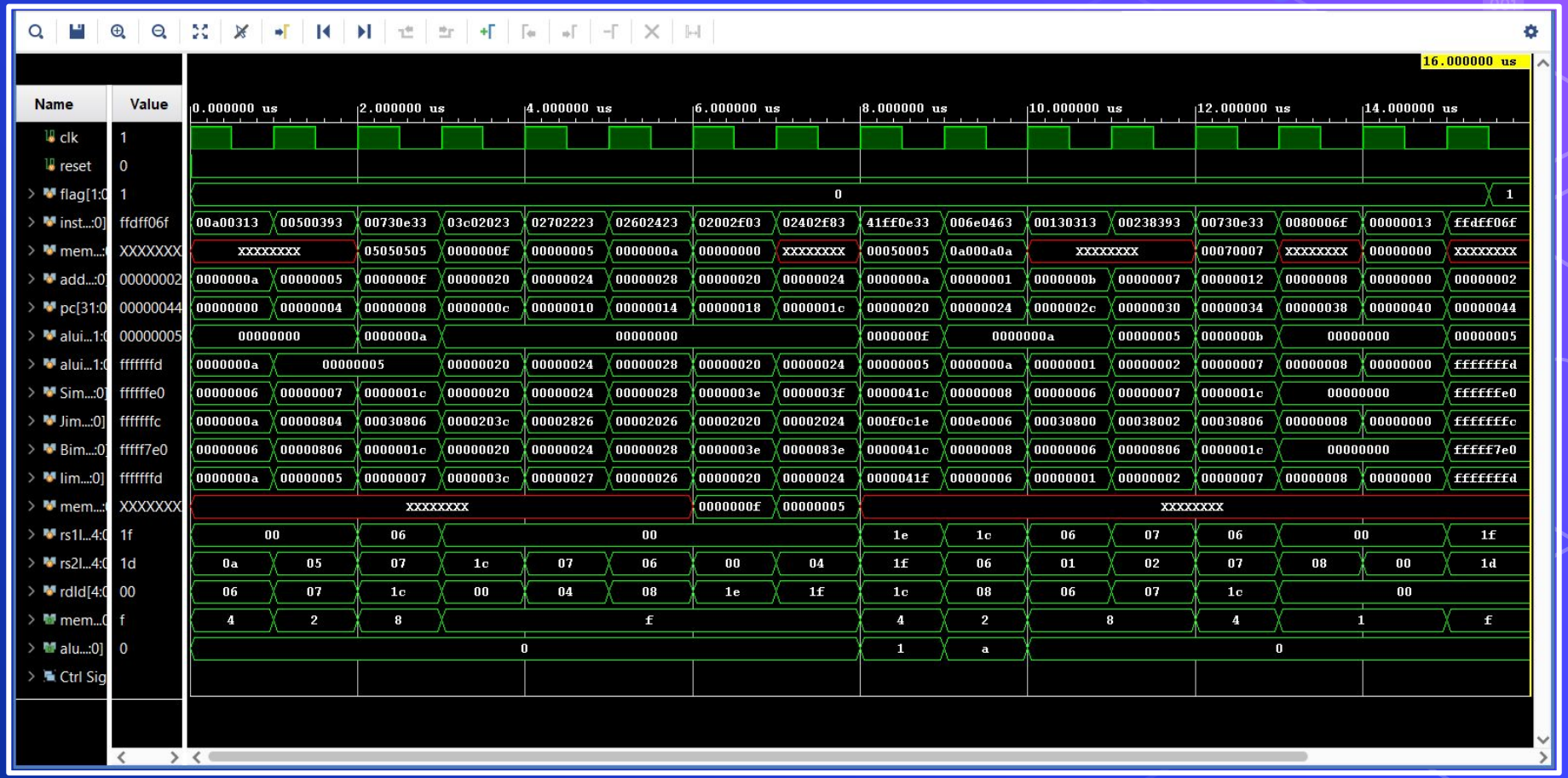
**

- **.hex**: Text format of binary code for flashing devices.
- **.bin**: Raw binary machine code for direct memory loading.
- **.dump**: Disassembled human-readable assembly code for debugging.

VIVADO SIMULATION TESTBENCH

```
module RISCV_TB;
 logic clk, reset;
    logic [1:0] flag;
    logic [31:0] instr, memWdata, addr, pc, aluin1, aluin2, Simm, Jimm, Bimm, Iimm, memRdata;
    logic [4:0] rs1Id, rs2Id, rdId ;
    logic [3:0] memWMask, aluCtrll;
    logic isAlureg, regWrite,  isJAL, isJALR, isBranch, isLUI, isAUIPC, isALUimm,  isLoad,   isStore,  isShamt;

RISCV dut (clk, reset, pc ,instr, memWdata, addr, aluin1, aluin2, Simm, Jimm, Bimm, Iimm, memRdata,rs1Id, rs2Id, rdId,
      memWmask, aluCtrl,isAlureg, regWrite, isJAL, isJALR, isBranch, isLUI, isAUIPC, isAluimm,isLoad, isStore, isShamt );

initial begin
        flag = 0;
        reset = 1; #15;
        reset = 0; #2;
    end

    always begin
        clk <= 1; #100;
        clk <= 0; #100;
    end

    always @(negedge clk) begin
        if (pc == 8'h44)
            flag = 1;
        if (flag == 2'b01)
        repeat(3) @(posedge clk)  $finish;
    end
endmodule
```

VIVADO SIMULATION RESULT



SIMULATION RESULT VERIFICATION

According to the assembly program our expected result should be:

Final **Expected** Registers and Memory Values:

- t1 (x6) = 11 (Store to memory[0x28])
- t2 (x7) = 7 (Store to memory[0x24])
- t3 (x28) = 18 (Store to memory[0x20])
- t5 (x30) = 15 (loaded from memory[0x20])
- t6 (x31) = 5 (loaded from memory[0x24])
- DMem[8] = 15
- DMem[9] = 5
- DMem[10] = 10

All other registers remain unchanged (or zero if they were never written).

Final **Received** Registers and Memory Values:

- t1 (x6) = 11 (Store to memory[0x28])
- t2 (x7) = 7 (Store to memory[0x24])
- t3 (x28) = 18 (Store to memory[0x20])
- t5 (x30) = 15 (loaded from memory[0x20])
- t6 (x31) = 5 (loaded from memory[0x24])
- DMem[8] = 15
- DMem[9] = 5
- DMem[10] = 10

All other registers and memory remain unchanged .

Memory Address Calculation of DMem:

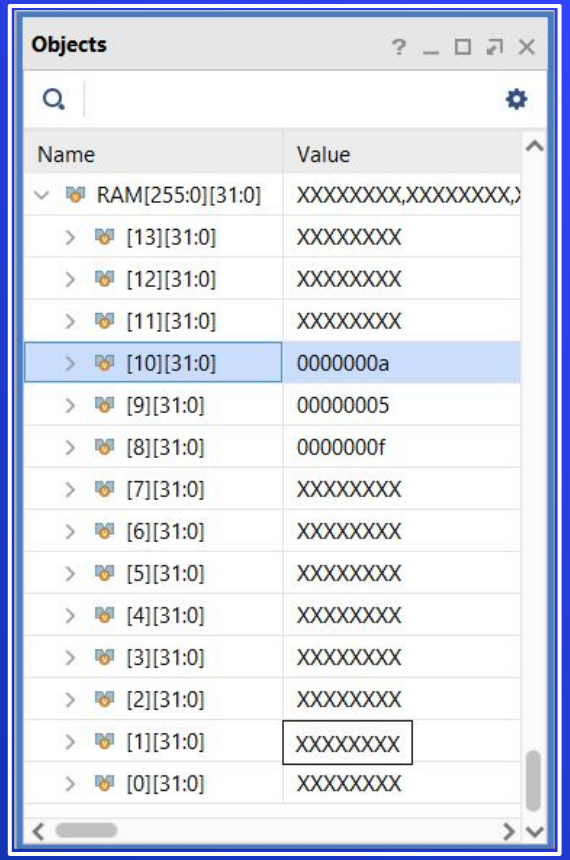
Storing at 0x20 :
0x20 = 32
Since RISC-V instructions are 32 bits wide (4 bytes), instructions must be aligned on 4-byte boundaries. This means the address of an instruction or data is always a **multiple of 4**.

So the memory position in decimal is 32/4 = 8.

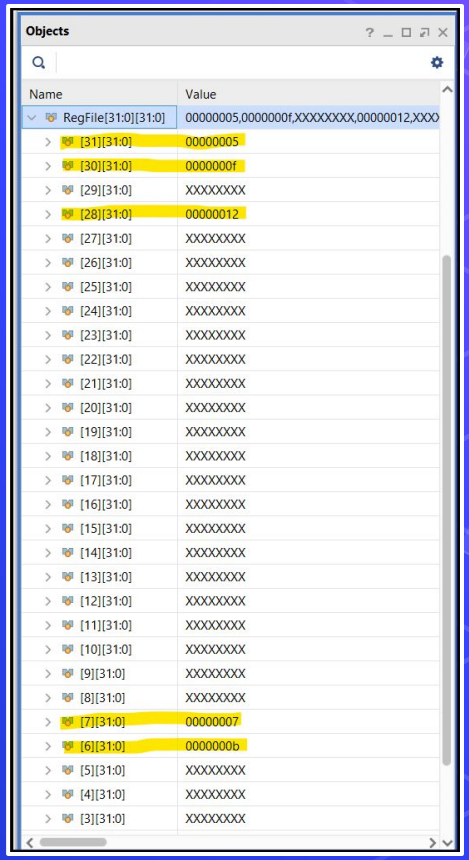
So storing at 0x20 = DMem[8].

**** All Result In Decimal Format**

SIMULATION RESULT VERIFICATION



RegFile Content



DMem Content

BIBLIOGRAPHY

- The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191213 Editors: Andrew Waterman, Krste Asanovic, SiFive Inc., CS Division, EECS Department, University of California, Berkeley , andrew@sifive.com, krste@berkeley.edu, December 13, 2019.

- Digital Design and Computer Architecture , Second Edition , Authors: David Money Harris, Sarah L. Harris.

- RISC-V Assembly Language Programming ,(Draft v0.18.3-0-g8a08bae) John Winans , jwinans@niu.edu, April 23, 2024.

- Computer Architecture A Quantitative Approach (5th edition),Authors : John L. Hennessy (Stanford University), David A. Patterson (University of California, Berkeley)