RISC-V

Trainee - Divyanshi Soni
Mentor - Ashish Jindal - SC-D

Softwares used: Xilinx Vivado, Iverilog, GTKwave

Introduction

RISC-V is an open-source, modular **Instruction Set Architecture (ISA)** developed at UC Berkeley. It is royalty-free and designed for a wide range of systems — from microcontrollers to high-performance servers.

🔑 Key Features

- **Open & Royalty-Free**: Encourages innovation and custom hardware.
- **Modular & Scalable**: Small base ISA with optional extensions.
- **Simplicity**: Easy to implement and verify.

🔒 Privilege Levels

- **U (User)**: For applications and embedded code (e.g., microcontrollers).
- **S (Supervisor)**: OS-level control (memory, I/O, etc.).
- **M (Machine)**: Highest privilege; hardware and boot control.

📏 ISA Variants

- **RV32** – 32-bit: Embedded/low-power.
- **RV64** – 64-bit: General-purpose processors.
- **RV128** – 128-bit: Experimental, ultra-high performance.

Overview: RISC-V Instruction Set Manual, Vol.1: User Level ISA, Version 2.1

RISC-V ISA – Introduction & Motivation

Overview:

- **RISC-V** (pronounced “risk-five”) is an open-source Instruction Set Architecture developed at **UC Berkeley**, intended for both academic research and commercial use.
- Designed as a **modular and extensible architecture**, RISC-V decouples the ISA from any specific microarchitecture or implementation technology (e.g., in-order, out-of-order, FPGA, ASIC).

Key Goals:

- Fully open and **free for anyone to implement**.
- Minimal **base ISA** (RV32I, RV64I) to simplify design and foster innovation.
- Modular: allows **optional standard extensions** (e.g., M, A, F, D, C) and **custom non-standard extensions**.
- Supports **32-, 64-, and 128-bit address spaces** (RV32, RV64, RV128).
- Flexible encoding to support **variable-length instructions** (16-bit compressed, 48+, etc.).

Why Not Existing ISAs?

- Commercial ISAs (e.g., x86, ARM) are proprietary, complex, and not well-suited for customization or academic use.
- RISC-V offers a **clean slate**, optimized for clarity, efficiency, and long-term stability.

Standard Extensions – M, A, F, D, C

M

M – Integer Multiply/Divide

- Adds MUL, DIV, REM, etc.
- Essential for general-purpose computing and efficient math operations.

A

A – Atomic Memory Operations

- Provides atomic Read-Modify-Write (AMO) instructions and Load-Reserved/Store-Conditional (LR/SC) pairs.
- Enables multi-threaded synchronization.

F

F – Single-Precision Floating-Point (IEEE 754-2008)

- Adds 32-bit floating-point registers, arithmetic, conversions, and load/store operations.

D

D – Double-Precision Floating-Point

- Extends F with support for 64-bit IEEE 754-2008 operations.

C

C – Compressed Instructions

- 16-bit encodings for frequently used instructions.
- Reduces **code size**, **instruction fetch bandwidth**, and **energy consumption**.

Abbreviated Notation:
RV32G = RV32IMAFD
RV64G = RV64IMAFD
Additional Extensions (Planned/Experimental):
Q: Quad-precision FP
V: Vector instructions
B: Bit manipulation
P: Packed SIMD
T: Transactional memory

More on Extensions of RISC-V

Code	Feature
I	Base Integer (mandatory)
M	Multiply/Divide
A	Atomic Instructions
F/D	Single/Double-Precision Floating Point
C	Compressed Instructions
Zicsr	Control/Status Register Access
Zifencei	Instruction Fence (sync)

⚙️ **Applications**

- Embedded Systems (U-mode)
- Operating Systems (S-mode)
- Boot and Hardware Control (M-mode)

I – Base Integer Extension
Core instruction set including basic arithmetic, logic, control flow, and memory operations using 32 general-purpose registers.

M – Multiply/Divide Extension
Adds hardware support for integer multiplication and division, crucial for performance in mathematical and DSP-intensive applications.

A – Atomic Instructions
Enables atomic read-modify-write operations for multi-threading and synchronization in multiprocessor or shared-memory systems.

F – Single-Precision Floating Point
Adds IEEE 754-compliant 32-bit floating-point operations, used in scientific computing, AI, and signal processing.

D – Double-Precision Floating Point
Extends F with 64-bit floating-point operations, providing higher precision for complex mathematical computations.

C – Compressed Instructions
Provides 16-bit instruction encoding for reduced code size and improved performance in memory-constrained embedded systems.

Zicsr – Control and Status Register Access
Supports instructions to access and modify control/status registers, essential for exception handling, interrupts, and system state management.

Zifencei – Instruction-Fence for Synchronization
Ensures correct ordering of instruction fetches, useful in self-modifying code and instruction memory updates in multi-core systems.

Base Integer ISAs – RV32I, RV64I, RV32E

RV32I

Core set of **47 simple, orthogonal instructions** for integer computation, memory access, and control flow.

Uses **32 general-purpose registers (x0–x31)**; x0 is hardwired to zero.

Provides the **minimum instruction set** needed for modern compilers and operating systems.

All instructions are **32-bit fixed-length** with **aligned memory accesses** (4-byte boundary).

RV64I

Extends RV32I to support **64-bit registers and address space**.

Adds instructions for 64-bit operations and maintains backward compatibility with RV32I.

RV32E

Variant of RV32I with only **16 general-purpose registers** to reduce hardware area and power.

Targeted at **small microcontrollers** and resource-constrained embedded systems.

Registers & PC:

x0–x31: General-purpose.
pc (Program Counter): Holds the address of the current instruction.

Memory Model, Control Flow & Philosophy

Memory Model:

- **Byte-addressed**, **little-endian**, and **flat address space**.
- Only **Load/Store** instructions access memory (load-store architecture).
- Supports **aligned and misaligned** memory accesses (though aligned are more efficient).
- Uses **FENCE** instructions to define memory ordering for multi-threading.

Control Transfer:

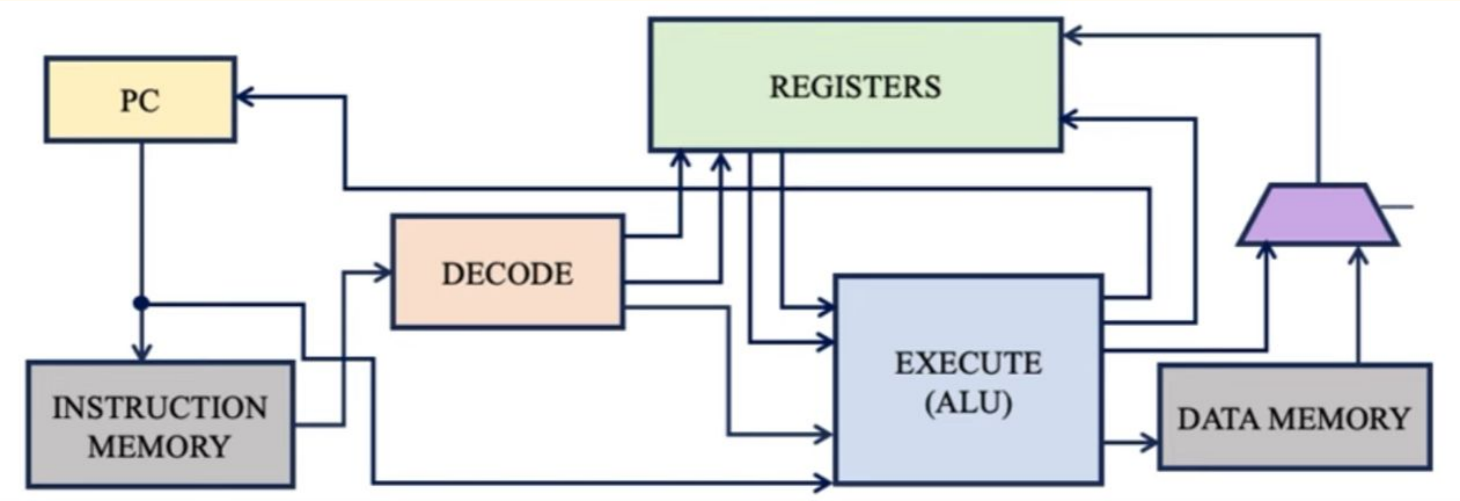
- **Unconditional jumps**: JAL, JALR (also used for function calls and returns).
- **Conditional branches**: BEQ, BNE, BLT, BGE, etc., with **PC-relative** addressing.
- No **delay slots** (unlike MIPS); instructions execute in strict sequential order.

Design Philosophy:

- **Minimal base ISA** for simplicity and flexibility.
- Avoids legacy baggage of older ISAs.
- Separates **user-level and privileged ISA specs**.
- ISA is **frozen** at version 2.0 for compatibility.
- Enables long-term software and hardware ecosystem stability.

Processor

This is the basic skeleton of a RISC processor.
Examples of RISC: MIPS,RISC-V



Execution Stages

The RISC-V processor follows a classic 5-stage pipeline: **IF, ID, EX, MEM, and WB**. In the **Fetch (IF)** stage, the instruction is retrieved from instruction memory. The **Decode (ID)** stage interprets the instruction and reads the necessary registers. Next, the **Execute (EX)** stage performs arithmetic or branch computations using the ALU. In the **Memory (MEM)** stage, load or store operations are performed using data memory. Finally, the **Writeback (WB)** stage stores the result back into the register file. This pipelining improves instruction throughput and overall processor efficiency.

- **ID (Decode):** Decode & read registers

- **MEM (Memory):** Load/Store operation

IF (Fetch):

Get instruction from memory

- **EX (Execute):** ALU or Branch logic

- **WB (Writeback):** Result to register file

Single Cycle Processor Architecture

🧩 **Definition:**

A **single-cycle processor** executes every instruction — regardless of type — in **exactly one clock cycle**.

⚙️ **Key Characteristics:**

- Each instruction completes the full 5 stages (IF, ID, EX, MEM, WB) **within one clock period**.
- The clock cycle must be **long enough to accommodate the slowest instruction** (e.g., lw, jal).
- All functional units (ALU, register file, memory) are **combinationally connected** in one large datapath.

Advantages:

- **Simplicity:** Easy to design and debug.
- **Ideal for learning:** Clear instruction flow and control.
- **No state between stages:** Everything flows combinationally.

⚠️ **Disadvantages:**

- **Inefficient clock cycle:** Wasted time for simple instructions (e.g., add still waits for MEM stage).
- **Slow clock:** Worst-case execution time defines the clock period.
- **No resource reuse:** ALU and memory can't be reused across instructions.

✅ **Use Case:**

- Educational designs
- FPGA prototypes for core testing

Multi-Cycle Processor Architecture

🧩 Definition:

A **multi-cycle processor** splits instruction execution across **multiple cycles**, with **one stage per clock** (e.g., IF, ID, EX...).

⚙️ Key Characteristics:

- The datapath is reused — same ALU is used for address computation and arithmetic in different cycles.
- Instructions take **different numbers of clock cycles** based on complexity.
- A **finite state machine (FSM)** or controller governs the current stage.

📌 Advantages:

- **Shorter clock cycle:** Clock is based on the **fastest stage**, not entire instruction.
- **Resource optimization:** Hardware units are reused across cycles.
- **Energy-efficient:** Only active units toggle.

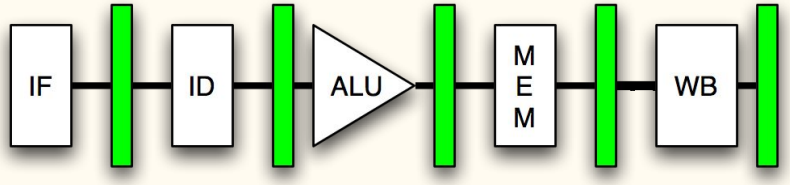
⚠️ Disadvantages:

- **Slower throughput:** One instruction takes 3–6 cycles on average.
- **Complex controller:** FSM must manage many transitions and signals.
- **Not ideal for superscalar or out-of-order designs.**

✅ Use Case:

- Low-power microcontrollers
- Older RISC/CISC processors
- Some embedded SoCs

Pipelined Processor Architecture



🧩 Definition:

A **pipelined processor** overlaps the execution of multiple instructions by dividing execution into **discrete stages**, just like an assembly line.

⚙️ Key Characteristics:

- Execution stages:
  - **IF → ID → EX → MEM → WB**
- One instruction starts every clock cycle, while previous ones are in later stages.
- Uses **pipeline registers** between stages to hold intermediate data and control signals.

📌 Advantages:

- **High throughput:** One instruction completes every cycle (after filling pipeline).
- **Efficient hardware use:** All parts of the datapath are active on every cycle.
- **Better performance with same clock speed.**

✅ Use Case:

- Modern CPUs (ARM Cortex, RISC-V cores, Intel)
- High-performance embedded systems
- Any system that requires high **Instruction-Per-Cycle (IPC)**

⚠️ Disadvantages:

- **Pipeline hazards:**
  - *Data Hazards:* Instructions depend on each other (e.g., read-after-write)
  - *Control Hazards:* Caused by branches and jumps
  - *Structural Hazards:* Conflicting hardware usage
- **Need for hazard detection units, forwarding, stalls, branch prediction**
- **Complexity increases significantly**

Comparison: Single-Cycle vs Multi-Cycle vs Pipelined Processors

Feature	🟢 Single-Cycle	🟡 Multi-Cycle	🔵 Pipelined
⏱️ Clock Period	Long (worst-case)	Short (fastest stage)	Short (fastest stage)
🔄 Instruction Latency	1 cycle	Varies (3–6 cycles)	~5 cycles (ideal)
🚀 Throughput	Low	Medium	High
🔁 Hardware Reuse	❌ Not reused	✅ Reused	✅ Reused (with stalls)
📊 Utilization	Poor	Good	Excellent
⚙️ Control Complexity	Low	Medium (FSM)	High (hazards, stalls)
🧠 Design Difficulty	Easy	Moderate	Complex

CPU performance

Processor Architecture vs. Microarchitecture

- A single processor architecture can have multiple microarchitectures.
- Microarchitectures vary in cost, performance, and hardware complexity.
- Cost increases with more gates and memory.

📈 Trends in Technology

- CMOS advancements allow more transistors on a chip each year.
- More transistors enable higher performance without increasing cost.

⚖️ Measuring Performance

🚫 Misleading Metrics

- Marketing often uses clock frequency to advertise speed.
- Example: Intel promoted higher clock speeds, while AMD offered better real-world performance at lower frequencies.

✅ Accurate Measurement

- The **best way** to evaluate performance is to measure how long a program takes to run.
- If your program isn't available, use a collection of similar programs (benchmarks).

🧮 Execution Time Factors

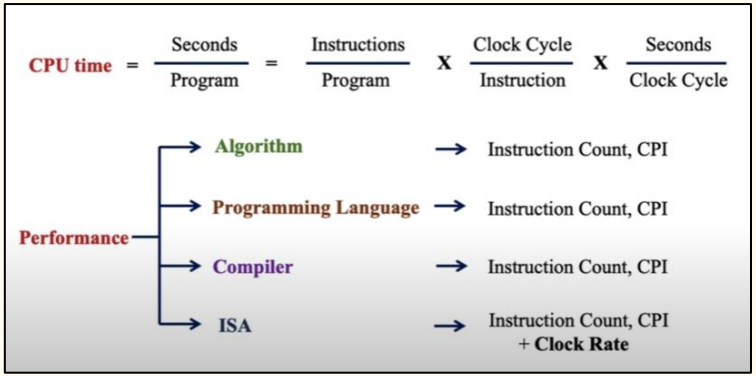
- **Instruction Count**:
  - Depends on architecture and code quality.
  - Complex instructions do more work but can be slower.
- **Cycles Per Instruction (CPI)**:
  - Average number of cycles needed to execute an instruction.
  - Varies with microarchitecture.
- **Clock Period**:
  - Time for one cycle; determined by critical path logic.
  - Influenced by logic design (e.g., carry-lookahead vs ripple-carry adders) and manufacturing.

🎯 Microarchitectural Goal

- Minimize **execution time** while staying within cost and power limits.
- Decisions impact both CPI and clock period.

🧩 Other System Bottlenecks

- Performance can also be limited by memory, disk, graphics, or network.
- Fast processors don't help if other components are slow (e.g., using dial-up Internet).



RV32I Base Integer Instruction Set, Version 2.0

—

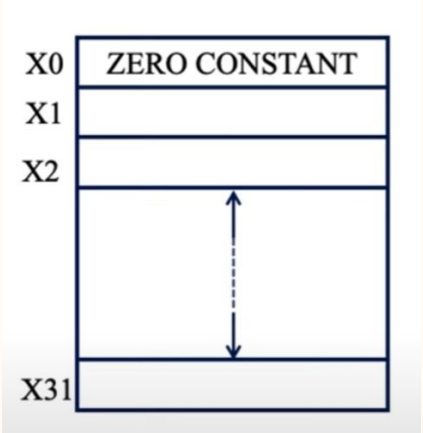
Chapter 2 from RISC-V Instruction Set Manual, Vol.1: User Level ISA, Version 2.1

Introduction

RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA was also designed to reduce the hardware required in a minimal implementation. RV32I contains 47 unique instructions, though a simple implementation might cover the eight SCALL/SBREAK/CSRR* instructions with a single SYSTEM hardware instruction that always traps and might be able to implement the FENCE and FENCE.I instructions as NOPs, reducing hardware instruction count to 38 total. RV32I can emulate almost any other ISA extension (except the A extension, which requires additional hardware support for atomicity).

This project models a subset of the RV32I base integer instruction set in Verilog HDL using Xilinx Vivado. The processor is built using a clean, modular datapath and controller structure, simulates correctly using Icarus Verilog + GTKWave

Register File



XLEN-1 0

x0 / zero
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31

XLEN

XLEN-1 0

pc

XLEN

Figure 2.1: RISC-V user-level base integer register state.

All registers are denoted as x0 to x31, and also have **conventional ABI names** (like zero, ra, sp, t0, s0, etc.).

- **Usage in Execution:**
  - Register file supports **two read ports and one write port**, enabling efficient **R-type instruction** execution.
  - Used for storing operands, function return values, and intermediate results during instruction execution.
- **Architectural Note:**
  - **x8–x9** and **x18–x27** are **callee-saved**, meaning their values are preserved across function calls.
  - **x10–x17** are **caller-saved**, often used for function arguments and return values.

Key Points About General Purpose Registers in RISC-V:

- **RISC-V provides 32 general-purpose registers (x0–x31)**, each 32 bits wide in RV32I. These are used for data storage, computation, and control flow during program execution.
- Each register has a **conventional name (ABI name)** for readability and function-specific use in higher-level languages and calling conventions.
- **x0 (zero)** is always hardwired to 0 and cannot be modified. Useful for clearing values or comparisons.
- **x1 (ra)** is the **return address** used during function calls. It is saved by the **caller**, meaning the function that calls another must save this value if needed later.
- **x2 (sp)** acts as the **stack pointer**, holding the top address of the current stack frame.
- **x3 (gp)** and **x4 (tp)** are used for **global and thread pointers**, respectively. tp is preserved by the called function (**callee**).
- **Temporary registers (t0–t6)** are used for intermediate calculations and are caller-saved.
- **Saved registers (s0–s11)** are callee-saved, meaning if a function uses them, it must preserve their original values.
- **Argument/return registers (a0–a7)** carry arguments to functions and return values from them. These are volatile and caller-saved.

Register	ABI Name	Description	Saver
x0	zero	Zero constant	-
x1	ra	Return address	caller
x2	sp	Stack Pointer	-
x3	gp	Global Pointer	-
x4	tp	Thread Pointer	callee
X5-X7	t0 - t2	Temporaries	caller
x8	s0 / fp	Saved/frame pointer	callee
x9	s1	Saved register	callee
x10-x11	a0-a1	Fn args/return values	caller
x12-x17	a2-a7	Fn args	caller
x18-x27	s2-s11	Saved registers	callee
X28-x31	t3-t6	Temporaries	caller

Type of Instructions

R-Type: Register to Register
I-Type: Register Immediate,Load,JLR, Ecall and Ebreak
S-Type: Store
B-Type: Branch
J-Type: Jump and Link
U-Type: Load/Add upper immediate

Instructions Format

31	[30 ——— 25]	[24 – 21]	20	[19 – 15]	[14 — 12]	[11 —— 8]	7	[6 ——— 0]	
funct7		rs2		rs1	funct3	rd		opcode	R - type
imm [11:0]				rs1	funct3	rd		opcode	I - type
imm [11:5]		rs2		rs1	funct3	imm [4:0]		opcode	S - type
imm [12]	imm [10:5]	rs2		rs1	funct3	imm [4:1]	imm [11]	opcode	B - type
imm [31:12]						rd		opcode	U - type
imm [20]	imm [10:1]		imm [11]	imm [19:12]		rd		opcode	J - type

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description	Note
add	ADD	R	011_0011	000	000_0000	rd = rs1 + rs2	
sub	SUB	R	011_0011	000	010_0000	rd = rs1 - rs2	
xor	XOR	R	011_0011	100	000_0000	rd = rs1 ^ rs2	
or	OR	R	011_0011	110	000_0000	rd = rs1 | rs2	
and	AND	R	011_0011	111	000_0000	rd = rs1 & rs2	
sll	Shift Left Logical	R	011_0011	001	000_0000	rd = rs1 << rs2	
srl	Shift Right Logical	R	011_0011	101	000_0000	rd = rs1 >>u rs2	
sra	Shift Right Arith*	R	011_0011	101	010_0000	rd = rs1 >>s rs2	
slt	Set Less Than	R	011_0011	010	000_0000	rd = (rs1 s< rs2) ? 1 : 0	
sltu	Set Less Than (U)	R	011_0011	011	000_0000	rd = (rs1 u< rs2) ? 1 : 0	
addi	ADD Immediate	I	001_0011	000		rd = rs1 + IMMI	IMMI = SXT (imm[11: 0])
xori	XOR Immediate	I	001_0011	100		rd = rs1 ^ IMMI	
ori	OR Immediate	I	001_0011	110		rd = rs1 | IMMI	
andi	AND Immediate	I	001_0011	111		rd = rs1 & IMMI	
slli	Shift Left Logical Imm	I	001_0011	001	imm[11:5] = 000_0000	rd = rs1 << imm[4:0]	
srli	Shift Right Logical Imm	I	001_0011	101	imm[11:5] = 000_0000	rd = rs1 >>u imm[4:0]	
srai	Shift Right Arith Imm	I	001_0011	101	imm[11:5] = 010_0000	rd = rs1 >>s imm[4:0]	
slti	Set Less Than Imm	I	001_0011	010		rd = (rs1 s< IMMI) ? 1 : 0	
sltiu	Set Less Than Imm (U)	I	001_0011	011		rd = (rs1 u< IMMI) ? 1 : 0	

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	Description	Note
lb	Load Byte	I	000_0011	000	rd = M[rs1 + IMMI]	IMMI = SXT (imm[11:0])
lh	Load Half	I	000_0011	001	rd = M[rs1 + IMMI]	
lw	Load Word	I	000_0011	010	rd = M[rs1 + IMMI]	
lbu	Load Byte (U)	I	000_0011	100	rd = M[rs1 + IMMI]	
lhu	Load Half (U)	I	000_0011	101	rd = M[rs1 + IMMI]	
sb	Store Byte	S	010_0011	000	M[rs1 + IMMI] = rs2	IMMI = SXT (imm[11:0])
sh	Store Half Store	S	010_0011	001	M[rs1 + IMMI] = rs2	
sw	Word	S	010_0011	010	M[rs1 + IMMI] = rs2	
beq	Branch ==	B	110_0011	000	if (rs1 == rs2) PC += IMMB	IMMB = SXT ({imm[12:1],1'b0})
bne	Branch !=	B	110_0011	001	if(rs1 != rs2) PC += IMMB	
blt	Branch <	B	110_0011	100	if(rs1 s< rs2) PC += IMMB	
bge	Branch ≥	B	110_0011	101	if(rs1 >=s rs2) PC += IMMB	
bltu	Branch < (U)	B	110_0011	110	if(rs1 u< rs2) PC +=IMMB	
bgeu	Branch ≥ (U)	B	110_0011	111	if(rs1 >=u rs2) PC += IMMB	
jal	Jump And Link	J	110_1111		rd= PC+4; PC += IMMJ	IMMJ = SXT ({imm[20:1],1'b0})
jalr	Jump And Link Reg	I	110_0111	000	rd= PC+4; PC={(rs1 + IMMI),1'b0}	IMMI = SXT (imm[11:0])
lui	Load Upper Imm	U	011_0111		rd = IMMU	IMMU = { imm[31:12],12'b0 }
auipc	Add Upper Imm to PC	U	001_0111		rd = PC + IMMU	
ecall	Environment Call	I	111_0011	000	Transfer Control to OS	imm[11:0] = 0000_0000_0000
ebreak	Environment Break	I	111_0011	000	Transfer Control to debugger	imm[11:0] = 0000_0000_0001

Immediate Variants and their Types

In the RISC-V instruction set architecture, many instructions use immediate values—constants that are directly embedded within the instruction. Since all RISC-V instructions are 32 bits wide, different instruction formats handle immediates in various ways depending on the operation being performed.

The **R-type** format does not include any immediate. It is used for register-to-register operations like add or sub, using fields such as rs1, rs2, rd, and control bits like funct3, funct7, and opcode.

The **I-type** format is used in instructions like addi, lw, and jalr. It includes a 12-bit immediate in the form imm[11:0], placed alongside source register rs1, destination register rd, and control fields.

The **S-type** format is used for store instructions like sw. The 12-bit immediate is split between imm[11:5] and imm[4:0], and combined during execution to calculate memory addresses.

The **SB-type** format handles branch instructions like beq and bne. It uses a scattered 13-bit immediate formed from bits imm[12], imm[10:5], imm[4:1], and imm[11], which is shifted left for alignment.

In **U-type** format, used for lui and auipc, the immediate is a 20-bit value from bits imm[31:12]. This is positioned in the upper bits of a 32-bit value, with the lower 12 bits set to zero.

The **UJ-type** format is used in jump instructions like jal. It builds a 21-bit immediate from imm[20], imm[10:1], imm[11], and imm[19:12]. This value is also left-shifted to maintain word alignment.

The lower half of the image shows how these immediate fields are reconstructed from the raw instruction bits. Each immediate type (I, S, B, U, J) has a specific reconstruction rule, ensuring the correct immediate value is formed during instruction decoding.

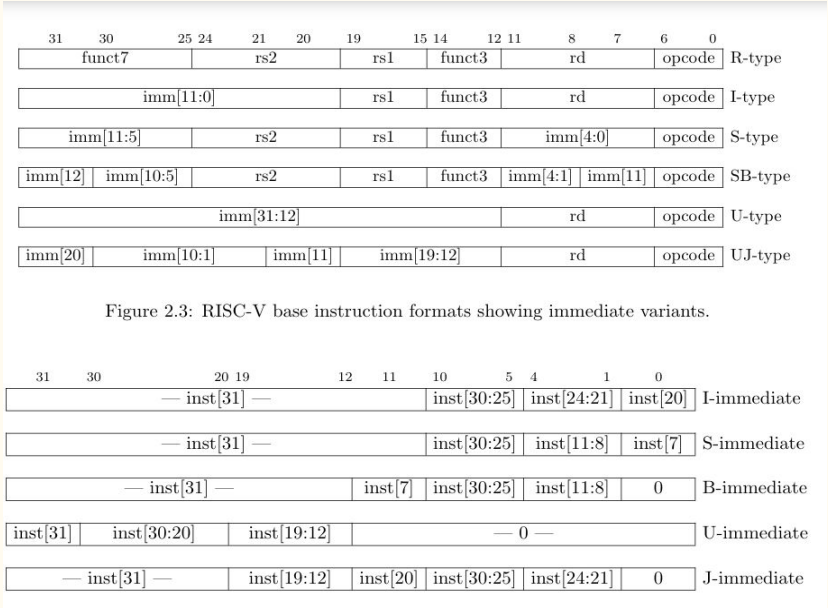


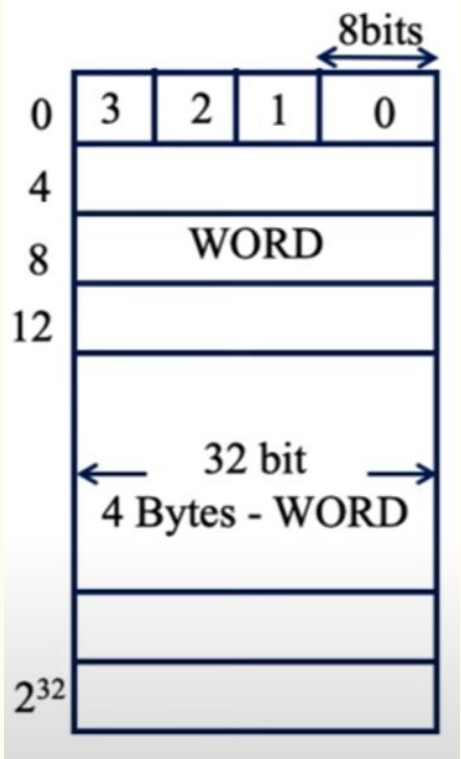
Figure 2.3: RISC-V base instruction formats showing immediate variants.

Memory

- Each element - 32 bits
- Instruction format; Instruction+Data
- Byte Addressing: 0,4,8,12
- Address Bus: 32 bits
- Max. Address: 2^32
- One word: 4 bytes : 32 bits

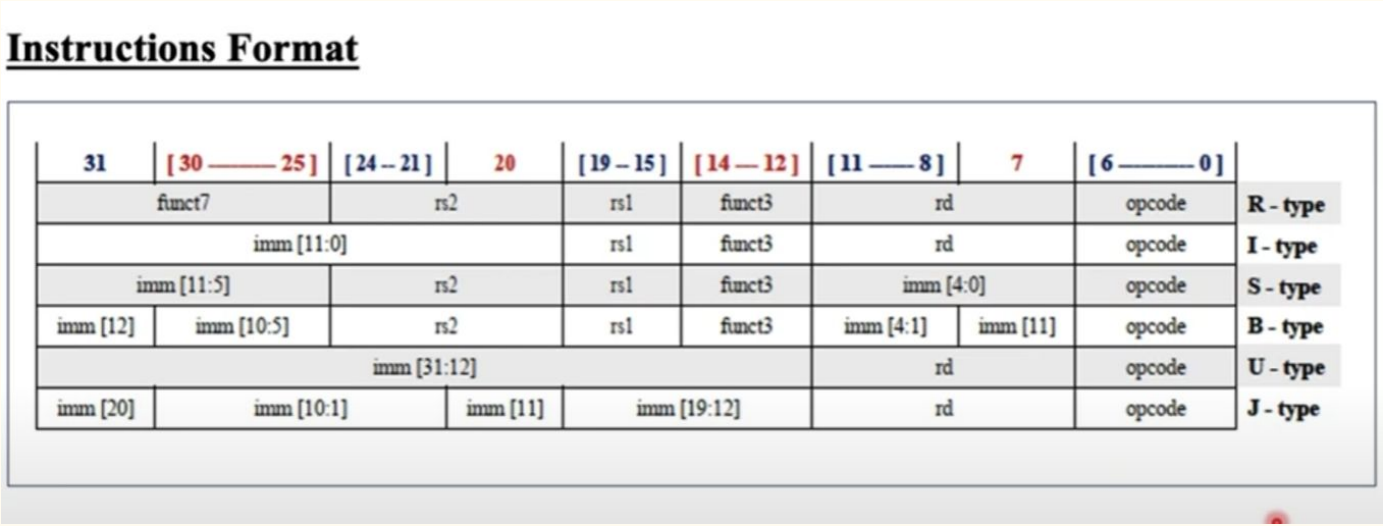
In RISC-V, memory is typically organized as a byte-addressable space where both instructions and data can reside. While the ISA supports a unified memory model (Von Neumann architecture), it is flexible and allows for separated instruction and data memories in actual hardware designs.

In this implementation, instruction and data memories have been separated following a Harvard-style architecture. This separation enables parallel access to instructions and data, improving performance and simplifying memory control. Immediate values in load and store instructions help calculate effective memory addresses, allowing efficient access to nearby data.

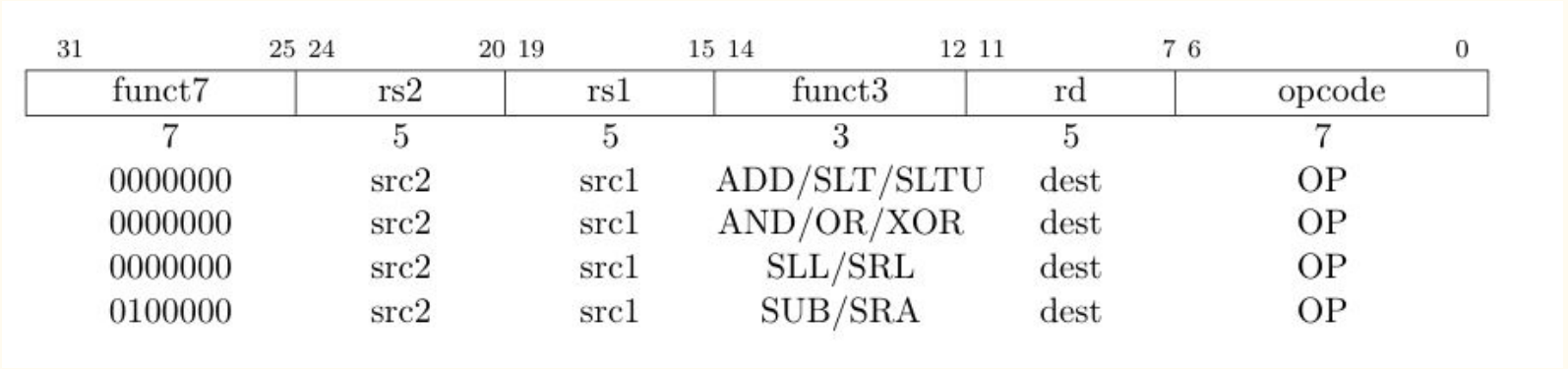


ISA Instruction classification

- Arithmetic and Logical
- Load and Store: Data Transfer b/w register and Memory
- Conditional: Statements and loops, non sequential order execution



R -Type instructions



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

Integer Register-Register Operations

All operations read the rs1 and rs2 Registers as source operands and write the result into register rd. The funct7 and funct3 fields select the type of operation.

Description of R type Instructions

- ADD and SUB perform **addition and subtraction**, respectively.
    - **Overflow is ignored**, and only the **lower XLEN bits** of the result are written to the destination register rd.
- SLT and SLTU perform **signed and unsigned comparisons**, respectively:
    - If rs1 $<$ rs2, then rd $=$ 1; otherwise, rd $=$ 0.
    - For example, SLTU rd, x0, rs2 sets rd to 1 **if rs2 $\neq$ 0**, else sets rd to 0.
        - (This acts like the assembler pseudo-op SNEZ rd, rs2 – "Set if Not Equal to Zero".)
- AND, OR, and XOR perform **bitwise logical operations** between rs1 and rs2.
- SLL, SRL, and SRA are **shift operations**:
    - SLL: Logical Left Shift (fills lower bits with 0)
    - SRL: Logical Right Shift (fills upper bits with 0)
    - SRA: Arithmetic Right Shift (fills upper bits with **sign bit**)

The shift amount is taken from the **lower 5 bits of register rs2** (for RV32).

Description of r type instructions

U: Unsigned

Inst	Name	FMT	Opcode	funct3	funct7	Description
add	ADD	R	011_0011	000	000_0000	rd = rs1 + rs2
sub	SUB	R	011_0011	000	010_0000	rd = rs1 - rs2
xor	XOR	R	011_0011	100	000_0000	rd = rs1 ^ rs2
or	OR	R	011_0011	110	000_0000	rd = rs1 | rs2
and	AND	R	011_0011	111	000_0000	rd = rs1 & rs2
sll	Shift Left Logical	R	011_0011	001	000_0000	rd = rs1 << rs2
srl	Shift Right Logical	R	011_0011	101	000_0000	rd = rs1 >>u rs2
sra	Shift Right Arith*	R	011_0011	101	010_0000	rd = rs1 >>s rs2
slt	Set Less Than	R	011_0011	010	000_0000	rd = (rs1 s < rs2) ? 1 : 0
sltu	Set Less Than (U)	R	011_0011	011	000_0000	rd = (rs1 u < rs2) ? 1 : 0

Classification

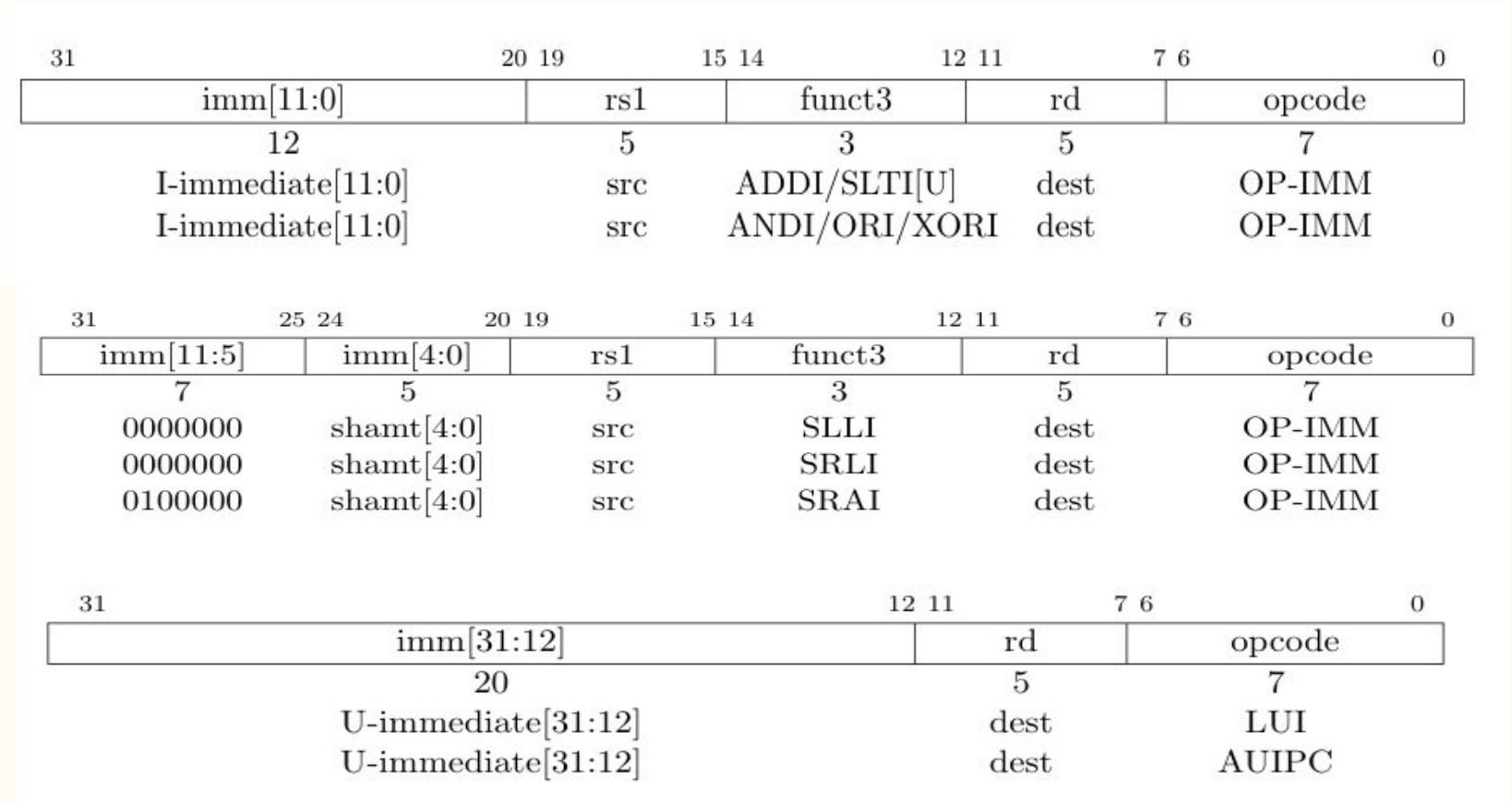
R-Type

Arithmetic	Logical	Shift	Comparison
add, sub	and, or, xor	sll, srl, sra	slt, sltu

I-Type

Arithmetic	Logical	Shift	Comparison
addi	andi, ori, xori	slli, srli, srai	slti, sltiu

I - type instructions: Integer Register-Immediate Instructions



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

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 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

Description of I type Instructions

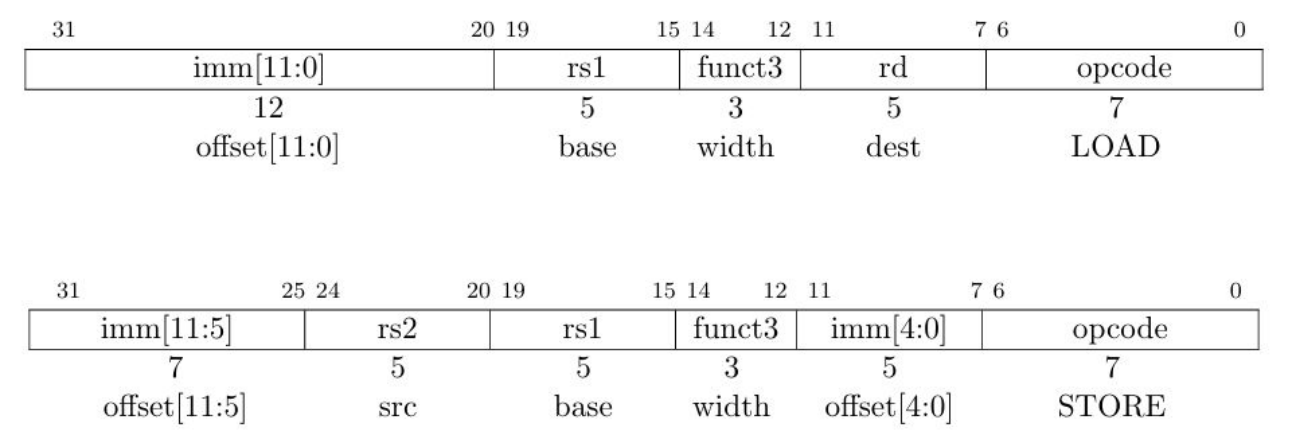
- ADDI adds the sign-extended 12-bit immediate to register rs1. Arithmetic over owisignored and the result is simply the low XLEN bits of the result. ADDI rd,rs1,0 is used to implement the MV rd,rs1 assembler pseudo-instruction.
- SLTI(set less than immediate) places the value 1 in register rd if register rs1 is less than the sign extended immediate when both are treated assigned numbers, else 0 is written to rd.
- SLTIU is similar but compares the values as unsigned numbers(i.e., the immediate is rst sign-extended to XLEN bits then treated as an unsigned number).Note,SLTIU rd,rs1, 1 sets rd to 1 if rs 1 equals zero,otherwise sets rd to 0(assembler pseudo-op SEQZ rd,rs).
- ANDI,ORI,XORI are logical operations that perform bitwise AND, OR, and XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd. Note,XOR I rd,rs1, -1 performs a bitwise logical inversion of register rs1 (assembler pseudo-instruction NOT rd,rs)
- Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in rs1, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right shift type is encoded in a high bit of the I-immediate.
- SLLI is a logical left shift (zeros are shifted into the lower bits);
  SRLI is a logical right shift (zeros are shifted into the upper bits); and
  SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

Description of I type Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description
addi	ADD Immediate	I	001_0011	000		rd = rs1 + IMMI
xori	XOR Immediate	I	001_0011	100		rd = rs1 ^ IMMI
ori	OR Immediate	I	001_0011	110		rd = rs1 | IMMI
andi	AND Immediate	I	001_0011	111		rd = rs1 & IMMI
slli	Shift Left Logical Imm	I	001_0011	001	imm[11:5] = 000_0000	rd = rs1 << imm[4:0]
srli	Shift Right Logical Imm	I	001_0011	101	imm[11:5] = 000_0000	rd = rs1 >>u imm[4:0]
srai	Shift Right Arith Imm	I	001_0011	101	imm[11:5] = 010_0000	rd = rs1 >>s imm[4:0]
slti	Set Less Than Imm	I	001_0011	010		rd = (rs1 s < IMMI) ? 1 : 0
sltiu	Set Less Than Imm (U)	I	001_0011	011		rd = (rs1 u < IMMI) ? 1 : 0

IMMI = SXT (imm[11: 0])

Load-I and Store-S type instructions



Address= rs1+SXT(imm[11:0])

Load: rd=M[Address];

Store: M[Address]=rs2;

Format : `lw rd, offset(rs1)`

`sw rs2, offset(rs1)`

rd =M[rs1 + offset]

M[rs1 + offset] = rs2

Description of Load-I and Store-S-type instructions

RV32I is a **load-store architecture**, meaning **only load and store instructions access memory**, while **arithmetic instructions operate only on CPU registers**.

- RV32I provides a **32-bit user address space**, byte-addressed and **little-endian**.
- The execution environment defines which portions of the address space are legal to access.

Load Instructions (I-type format):
Transfer a value **from memory to register rd**. The **effective byte address** is computed as:
rs1 + sign-extended 12-bit offset.

- LW: Loads 32-bit value from memory into rd.
- LH: Loads 16-bit value and **sign-extends** to 32 bits before storing in rd.
- LHU: Loads 16-bit value and **zero-extends** to 32 bits.
- LB, LBU: Same as above, but for 8-bit values.

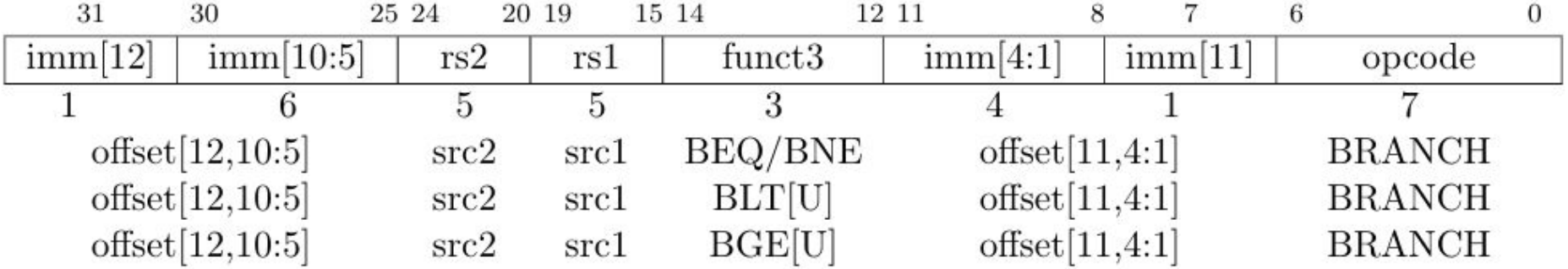
Store Instructions (S-type format):
Transfer a value **from register rs2 to memory**, at the effective address:
rs1 + sign-extended 12-bit offset.

- SW: Stores the lower 32 bits of rs2.
- SH: Stores the lower 16 bits.
- SB: Stores the lower 8 bits.

Description of Load-I and Store-S-type instructions

Inst	Name	FMT	Opcode	funct3	Description	Note
lb	Load Byte	I	000_0011	000	rd = M[rs1 + IMMI]	IMMI = SXT (imm[11:0])
lh	Load Half	I	000_0011	001	rd = M[rs1 + IMMI]	
lw	Load Word	I	000_0011	010	rd = M[rs1 + IMMI]	
lbu	Load Byte (U)	I	000_0011	100	rd = M[rs1 + IMMI]	
lhu	Load Half (U)	I	000_0011	101	rd = M[rs1 + IMMI]	
sb	Store Byte	S	010_0011	000	M[rs1 + IMMI] = rs2	IMMI = SXT (imm[11:0])
sh	Store Half Word	S	010_0011	001	M[rs1 + IMMI] = rs2	
sw	Word	S	010_0011	010	M[rs1 + IMMI] = rs2	

B type instructions



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

IMMB={SXT[12:0],1'b0}

True: PC+IMMB

False: PC+4

Description of B type Instructions

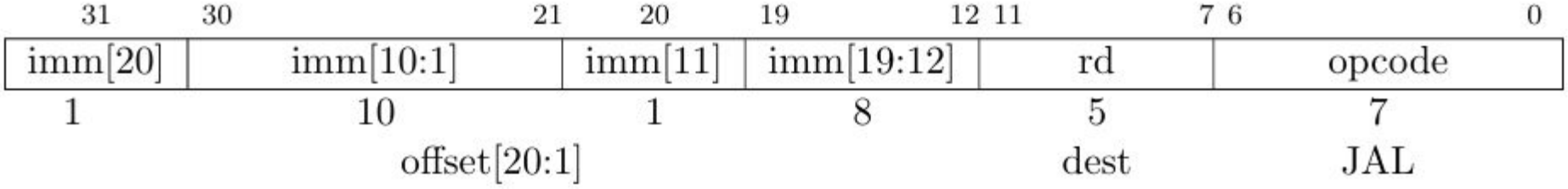
Inst	Name	FMT	Opcode	funct3	Description	Note
beq	Branch ==	B	110_0011	000	if (rs1 == rs2) PC += IMMB	IMMB = SXT ({imm[12:1],1'b0})
bne	Branch !=	B	110_0011	001	if(rs1 != rs2) PC += IMMB	
blt	Branch <	B	110_0011	100	if(rs1 s< rs2) PC += IMMB	
bge	Branch ≥	B	110_0011	101	if(rs1 >=s rs2) PC += IMMB	
bltu	Branch < (U)	B	110_0011	110	if(rs1 u< rs2) PC +=IMMB	
bgeu	Branch ≥ (U)	B	110_0011	111	if(rs1 >=u rs2) PC += IMMB	

Branch instructions compare two registers:

- BEQ and BNE take the branch if registers rs1 and rs2 are equal or unequal, respectively.
- BLT and BLTU take the branch if rs1 is less than rs2, using signed and unsigned comparisons, respectively.
- BGE and BGEU take the branch if rs1 is greater than or equal to rs2, again using signed and unsigned comparisons.

Note: BGT, BGTU, BLE, and BLEU can be synthesized by **reversing the operands** to BLT, BLTU, BGE, and BGEU, respectively.

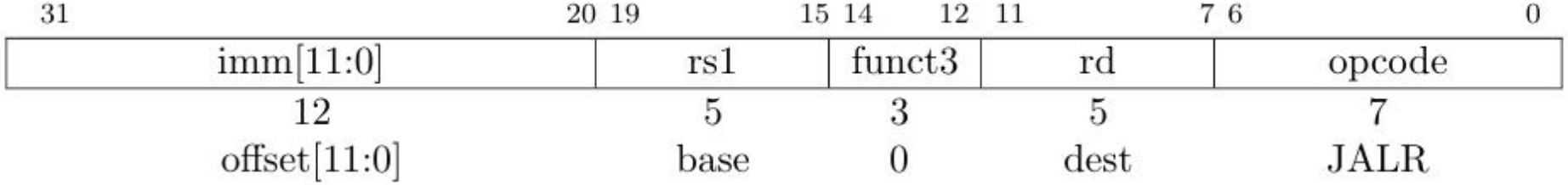
Jump and Link type instructions



IMMJ=SXT{[20:1],1b'0}

Rd: PC=PC+4, PC=PC+IMMJ

Jump and LinkR Instructions- I type



- **J-Type: jal**
  - Unconditional jump and link

```
jal rd, label
rd = PC + 4
PC = + imm
```

  - Jump target specified as label
  - Label encoded as an offset from the current address.
  - Only 20 bits allocated for IMM[Can address only 2^20]

- **I-Type: jalr**
  - Unconditional jump to end link register

```
jalr rd, imm(rs)
rd = PC + 4
PC= rs + imm
```

  - **Example:** `jalr x20,4(x12)` #Jump to x12+ 4
  - Supports by jump-Register holds 32-bit value(address)
  - Long jump : 2^32 locations

U type instructions

LUI: rd= IMMU

AUIPC: rd=PC+IMMU

IMMU={SXT[31:12],12'b0}

Description of J,U and I type Instructions

- The jump and link (JAL) instruction uses the UJ-type format, where the J-immediate encodes a signed o set in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Jumps can therefore target a 1MiB range.
- JAL stores the address of the instruction following the jump (pc+4) into register rd. The standard software calling convention uses x1 as the return address register. Plain unconditional jumps (assembler pseudo-op J) are encoded as a JAL with rd=x0
- The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the 12-bit signed I-immediate to the register rs1, then setting the least significant bit of the result to zero. The address of the instruction following the jump (pc+4)is written to register rd. Register x0 can be used as the destination if the result is not required.
- The JAL and JALR instructions can generate a misaligned instruction fetch exception if the target address is not aligned to a four-byte boundary.
- LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bit with zeros.
- AUIPC(add upper immediate pc) isusedtobuildpc-relative addresses and uses the U-type format.AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd.

Description of J,U and I type Instructions

Inst	Name	FMT	Opcode	funct3	Description	Note
jal	Jump And Link	J	110_1111		rd= PC+4; PC += IMMJ	IMMJ = SXT ({imm[20:1],1'b0})
jalr	Jump And Link Reg	I	110_0111	000	rd= PC+4; PC= { (rs1 + IMMI), 1'b0 }	IMMI = SXT(imm[11:0])
lui	Load Upper Imm	U	011_0111		rd = IMMU	IMMU = { imm[31:12],12'b0 }
auipc	Add Upper Imm to PC	U	001_0111		rd = PC + IMMU	
ecall	Environment Call	I	111_0011	000	Transfer Control to OS	imm[11:0] = 0000_0000_0000
ebreak	Environment Break	I	111_0011	000	Transfer Control to debugger	imm[11:0] = 0000_0000_0001

The complete RV32I Instruction Set

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
0000	pred | succ	00000	000	00000	0001111	FENCE
0000	0000 | 0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

RV32I (Core Subset) Modeling in Verilog using Xilinx Vivado
Single Cycle Processor

Chapter 2 from RISC-V Instruction Set Manual, Vol.1: User Level ISA, Version 2.1

IMPLEMENTING IN VERILOG

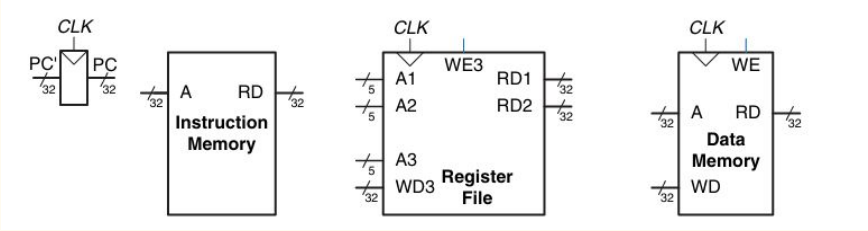
A subset of RV32I instructions is modelled. 21 out of 47 total instructions are supported by the current design of RV32I.
Here is the list of Supported and Unsupported Instructions:

The Designed uP is Single-Cycle Processor.

Supported instructions	Unsupported instruction
R-Type: add,sub,and,or,xor	**R-Type (extras):** `sll`, `srl`, `sra`, `slt`, `sltu`
I-Type: addi,andi,ori,xori	**I-Type (extras):** `slli`, `srli`, `srai`, `slti`, `sltiu`
Load-Store Type: lw,sw	**System instructions:** `ecall`, `ebreak`, `fence`, `csrrw`, etc.
B-Type: beq,bne,blt,bge,bltu,bgeu	
U-Type: lui,auipc	
J-Type: jal,jalr	

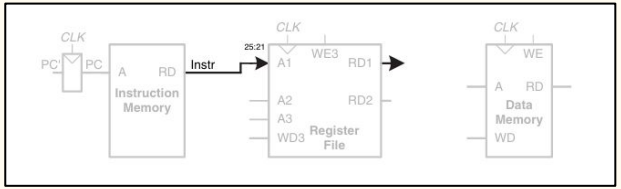
Datapath Design

We first design a RISC-V microarchitecture that executes instructions in a single cycle. We begin constructing the datapath by connecting the state elements with combinational logic that can execute the various instructions. Control signals determine which specific instruction is carried out by the datapath at any given time. The controller contains combinational logic that generates the appropriate control signals based on the current instruction.

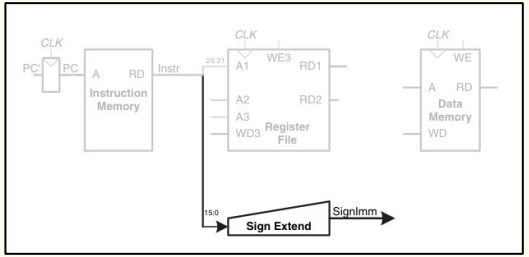


STATE ELEMENTS

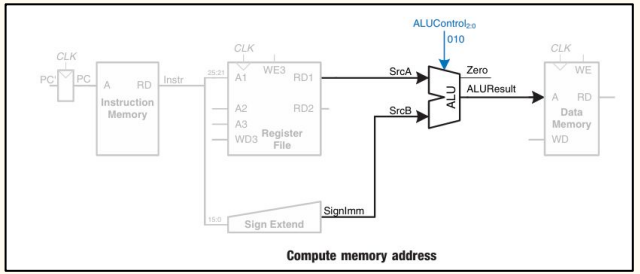
Step-by-Step Datapath Design:



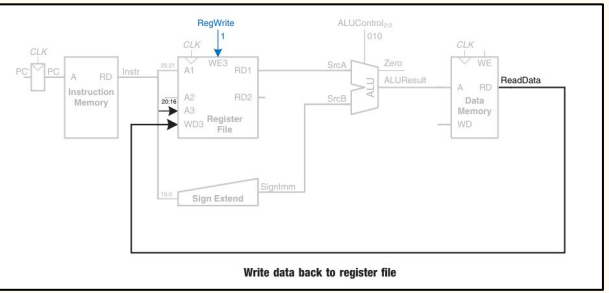
Read Source Operand from register File



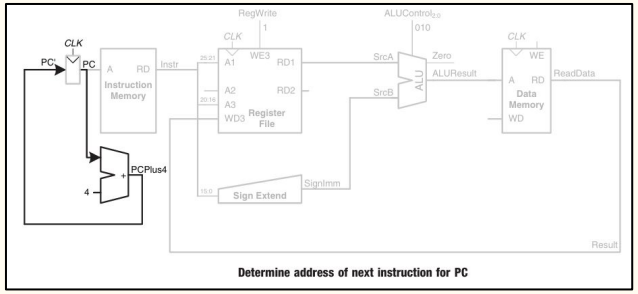
Sign Extend the Immediate



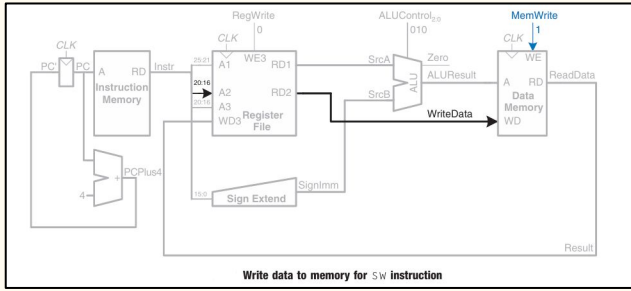
Compute Memory Address



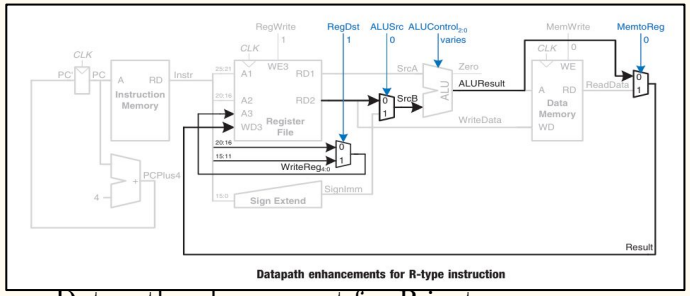
Write Data Back to register File



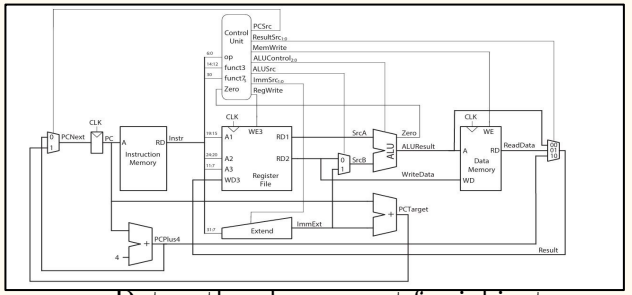
Determine Address of next instruction for PC



Write Data to memory for sw instruction

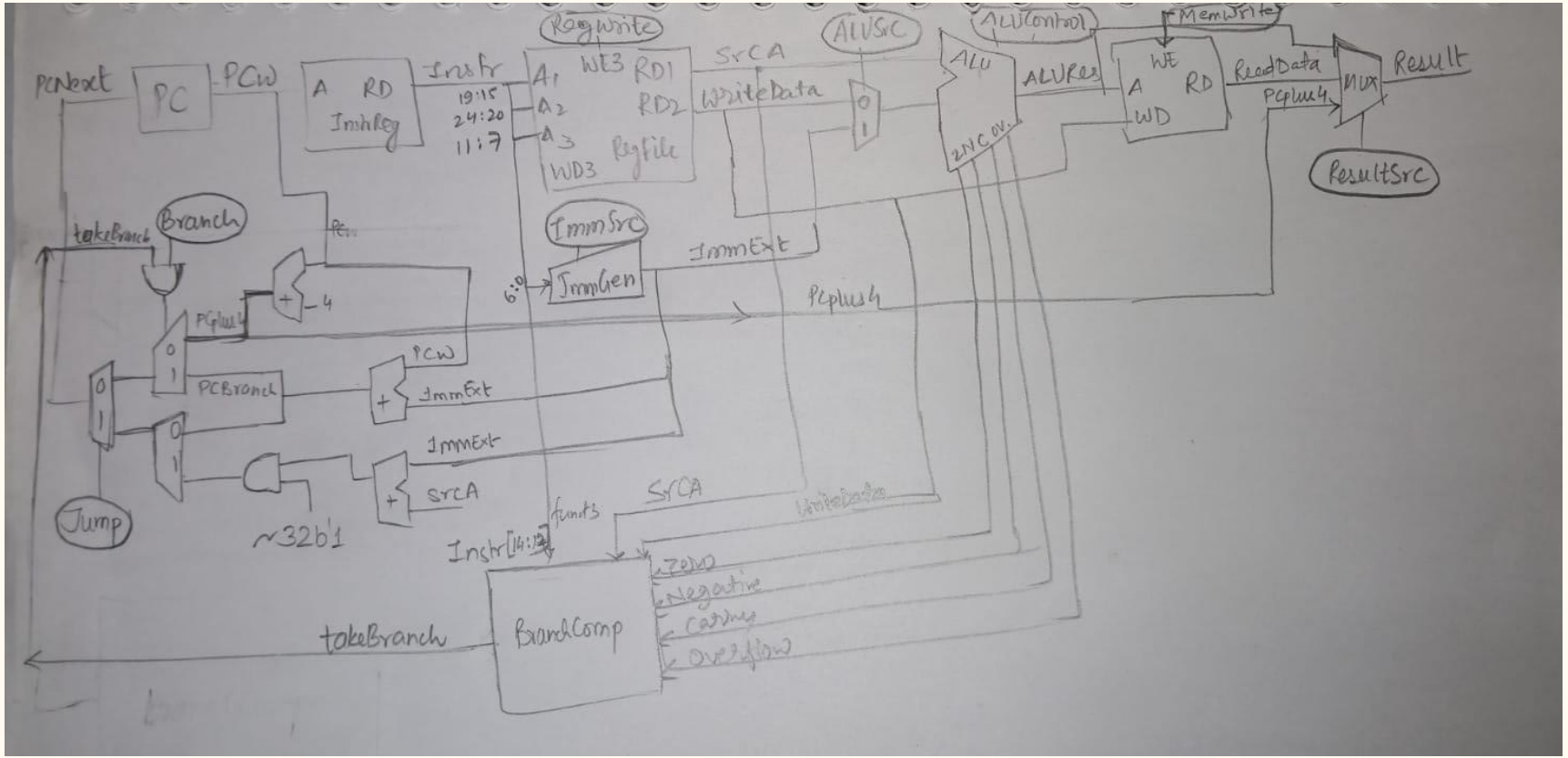


Datapath enhancement for R instr.



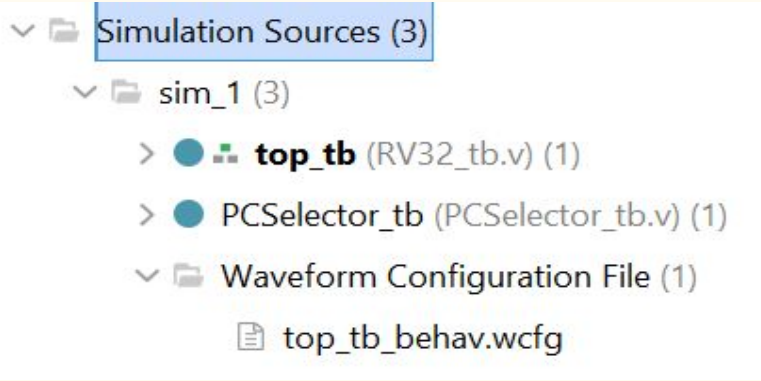
Datapath enhancement for jal instr.

Complete datapath

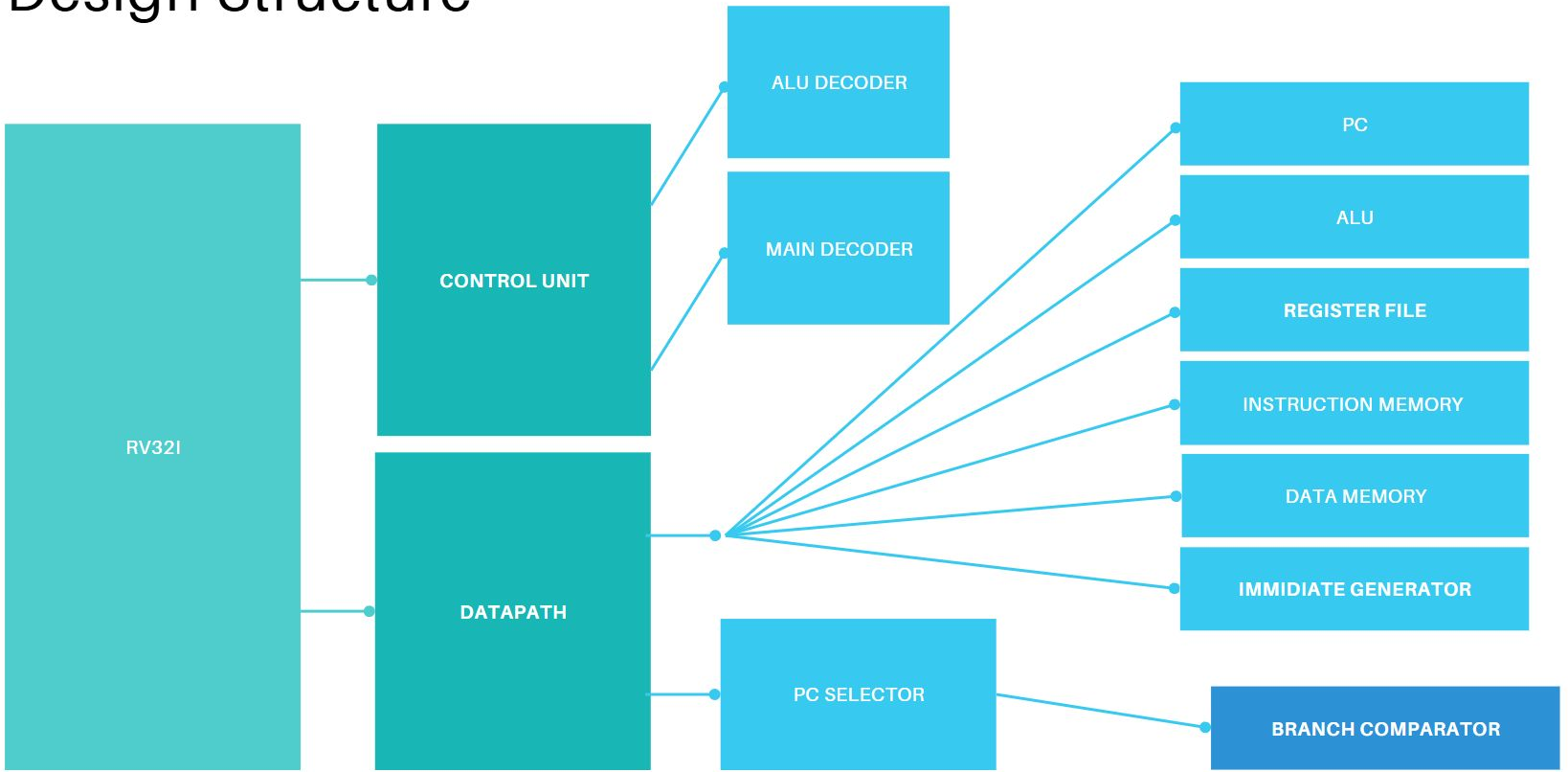


Design Structure

Here is a modular structure for the design of RV32I.Each module is placed in a different .v file for better readability and flexibility.



Design Structure



PC Module

Purpose:

- The Program Counter (PC) holds the **address of the current instruction** to be fetched.
- It updates at each rising clock edge to point to the **next instruction**.
- If reset is active, it resets to address 0x00000000

Key Features:

- **Input:**
  clk: Clock signal
  rst: Reset signal
  PCNext: Next PC value (from PCSelector)
- **Output:**
  - PC: Current instruction address

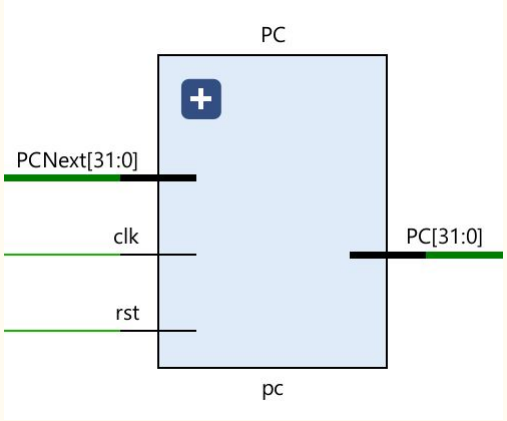
🧠 **Explanation:**

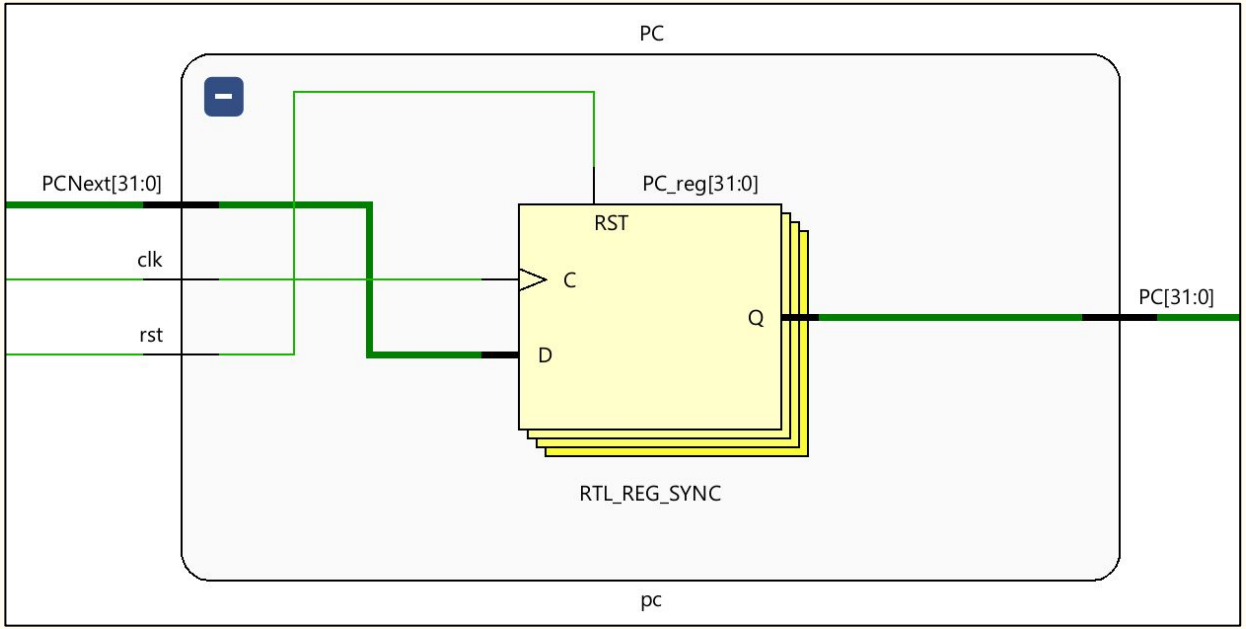
- On reset (rst = 1), PC is set to 0 — this ensures the processor starts executing from the first instruction.
- On every positive edge of the clock, if not reset, the PC is updated to the next instruction address (PCNext).
- This module ensures **sequential execution** unless PCNext is calculated differently due to jumps/branches.

```
module pc(
    input clk, rst,
    input [31:0] PCNext,
    output reg [31:0] PC

);
    always @ (posedge clk)
    begin
        if(rst)
         begin
            PC <= 32'h00000000;
         end
        else
        begin
            PC <= PCNext;
        end
    end
endmodule
```

Schematic of PC

Outer Schematic showing inputs and outputs:





PC Selector Module

Purpose:

- Computes the next PC value depending on:
  - Normal execution (PC+4)
  - Conditional branches (beq, bne)
  - Unconditional jumps (jal, jalr)

📌 **Outputs:**

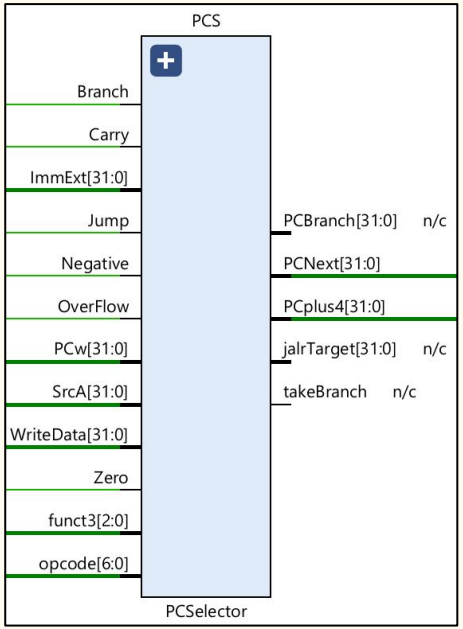
- PCNext: Final address to send to PC
- PCplus4: Default sequential address
- PCBranch: PC + branch offset
- jalrTarget: For jalr (SrcA + offset)

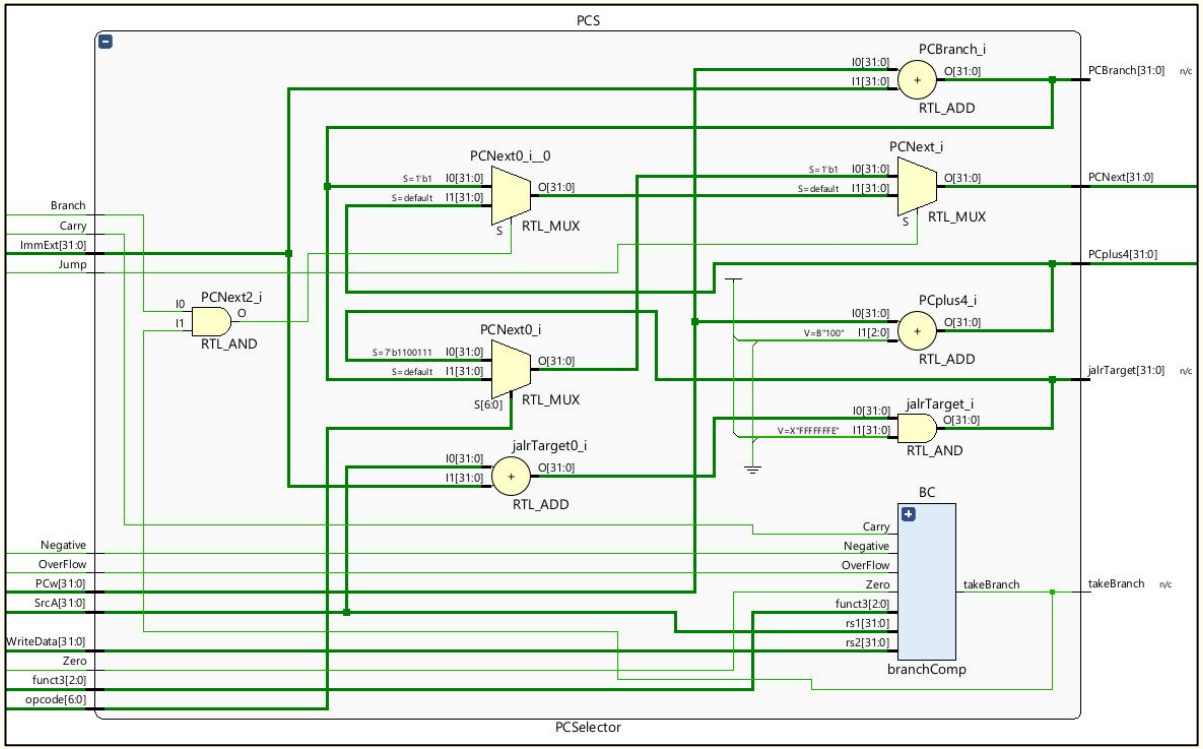
Key module that ties control and datapath together.

```
module PCSelector(
   input [31:0] PCw,ImmExt,SrcA,WriteData,
   input [6:0] opcode,
   input Branch,Jump,Zero,Negative,OverFlow,Carry,
   input [2:0] funct3,
   output [31:0] PCplus4,PCNext,PCBranch,jalrTarget,
   output takeBranch
);

   assign PCplus4=PCw + 32'd4;
   assign PCBranch= PCw + ImmExt;
   assign jalrTarget = (SrcA+ImmExt) & ~32'b1;
//    assign PCJumpTarget = PCw + ImmExt;


   branchComp BC(
      .funct3(funct3),
      .rs1(SrcA),
      .rs2(WriteData),
      .Zero(Zero),
      .Negative(Negative),
      .OverFlow(OverFlow),
      .Carry(Carry),
      .takeBranch(takeBranch)
   );

   assign PCNext = Jump ?
            ((opcode == 7'b1100111) ? jalrTarget : PCBranch) :
            ((Branch && takeBranch) ? PCBranch : PCplus4);

endmodule
```

PC Selector Module Schematic

Outer Schematic showing inputs and outputs:





Branch Comparator Module

```
module branchComp(
    input [2:0] funct3,
    input [31:0] rs1, rs2,
    input Zero, Negative,OverFlow,Carry,
    output reg takeBranch
);

    always @(*) begin
        case (funct3)
            3'b000: takeBranch = Zero;                  // BEQ
            3'b001: takeBranch = ~Zero;                 // BNE
            3'b100: takeBranch = (Negative != OverFlow);    // BLT
            3'b101: takeBranch = (Negative == OverFlow);    // BGE
            3'b110: takeBranch = ~Carry;                // BLTU
            3'b111: takeBranch = Carry;                 // BGEU
            default: takeBranch = 1'b0;

        endcase
    end

endmodule
```

Purpose:

- Determines if a branch condition is met.

Inputs:

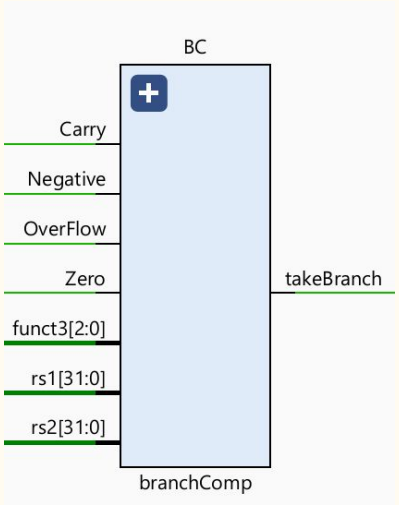
- funct3: Determines type of branch (beq, bne, etc.)
- rs1, rs2: Values from register file
- Flags: Zero, Negative, OverFlow, Carry

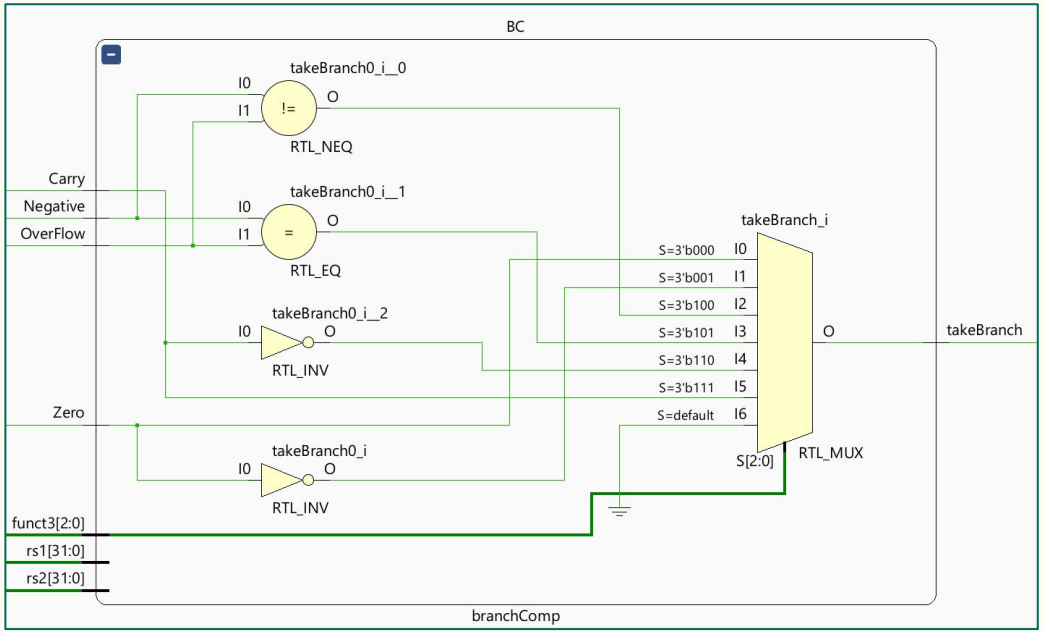
📌 **Output:**

- takeBranch: Boolean output for PCSelector.
- IT IS =1 WHEN BRANCH CONDITION IS MET.

Branch Comparator Module Schematic

Outer Schematic showing inputs and outputs:





Register File Module

Purpose:

- Holds the **32 general-purpose registers** (x0–x31), each 32-bits wide.
- Supports two simultaneous reads and one write.

Key Features:

- **Inputs:**
  - A1, A2: Source register addresses
  - A3: Destination register address
  - WD3: Write data
  - WE3: Write enable
  - clk, rst
- **Outputs:**
  - RD1, RD2: Read data from source registers

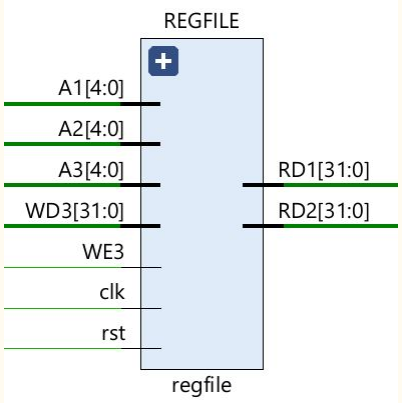
🧠 **Explanation:**

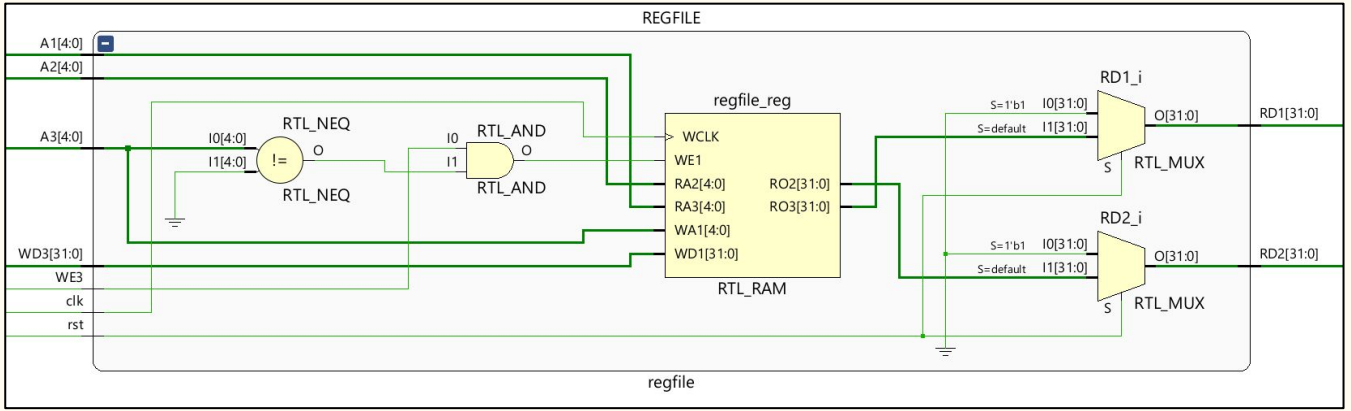
- **Two reads:** The values in registers A1 and A2 are read in parallel and output as RD1 and RD2.
- **One write:** On positive clock edge, if WE3 = 1 and A3 ≠ 0, WD3 is written into regfile[A3].
- **x0 register:** Hardwired to zero; regfile[0] never changes.
- Implements the **Instruction Decode (ID)** and **Write Back (WB)** stages.

```
module regfile(
   input [4:0] A1,A2,A3,
   input WE3, clk, rst,
   input [31:0] WD3,
   output [31:0] RD1,RD2
);

   reg [31:0] regfile [31:0];

// read // mux implemented
   assign RD1= (rst) ? 32'h0: regfile[A1];
   assign RD2=  (rst) ? 32'h0 : regfile[A2];


// write
   always @(posedge clk)
   begin
    if (WE3 && A3 != 5'd0)
      begin
         regfile[A3]<= WD3;
      end
   end

   initial begin
      regfile[0]=32'h00000000;
   end
endmodule
```

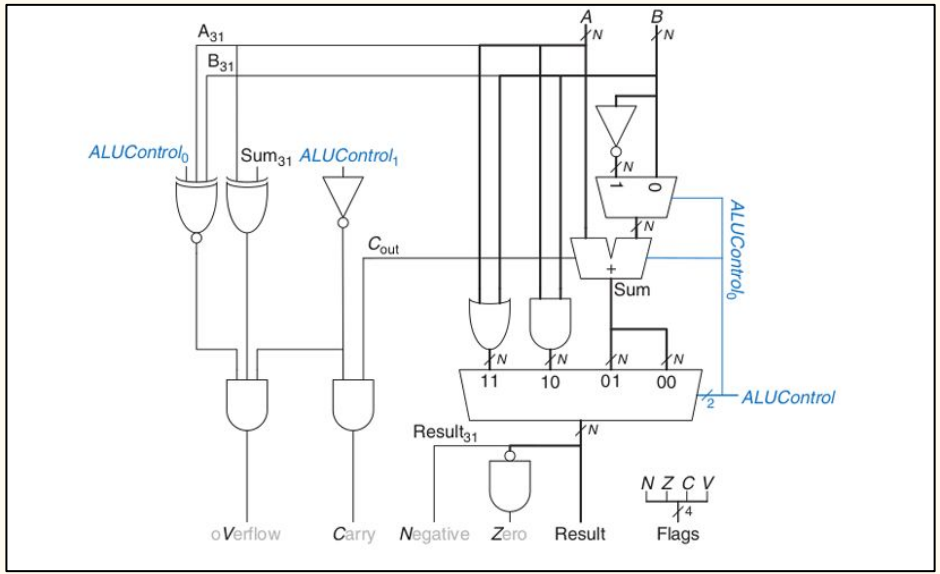
Schematic of Register File Module

Outer Schematic showing inputs and outputs





ALU DESIGN



Based on the ALU control the result is obtained for different instructions such as add,sub,and,or,etc. The flags computed using the logic shown are used for comparing values for branch instructions such as beq,bne,etc.

Comparison	Signed	Unsigned
=	Z	Z
≠	$\overline{Z}$	$\overline{Z}$
<	N⊕V	$\overline{C}$
≤	Z + (N⊕V)	Z + $\overline{C}$
>	$\overline{Z}$ • ($\overline{N \oplus V}$)	$\overline{Z}$ • C
≥	($\overline{N \oplus V}$)	C

The Result is assigned based on the ALUControl Signal. Below is the table for it:

ALUControl	operation
000	ADD
001	SUB
010	AND
011	OR
100	XOR
101	SLT

```
module alu(
    input [31:0] A, B,
    input [2:0] ALUControl,
    output Carry, OverFlow, Zero, Negative,
    output reg [31:0] Result
);

    wire Cout;
    wire [31:0] Sum;

    assign {Cout, Sum} = (ALUControl == 3'b001) ? (A - B) : (A + B);

    always @(*) begin
        case (ALUControl)
            3'b000: Result = A + B;         // ADD
            3'b001: Result = A - B;         // SUB
            3'b010: Result = A & B;         // AND
            3'b011: Result = A | B;         // OR
            3'b100: Result = A ^ B;         // XOR
            3'b101: Result = (A < B) ? 32'b1 : 32'b0; // SLT
            default: Result = 32'b0;
        endcase
    end

    assign OverFlow = (ALUControl == 3'b000 || ALUControl == 3'b001) ?
                      ((A[31] == B[31]) && (Result[31] != A[31])) : 1'b0;

    assign Carry = (ALUControl == 3'b000) ? Cout :
                   (ALUControl == 3'b001) ? ~Cout : 1'b0;

    assign Zero = (Result == 32'b0);
    assign Negative = Result[31];

endmodule
```

ALU

Purpose:

- Performs all arithmetic and logical operations needed by instructions.

Inputs and Outputs:

- **Inputs:**
  - A, B: Operands
  - ALUControl: 3-bit operation selector
- **Outputs:**
  - Result: Computed result
  - Zero: Whether result is zero
  - Negative: MSB of result
  - OverFlow, Carry: Optional flags (used in comparisons)

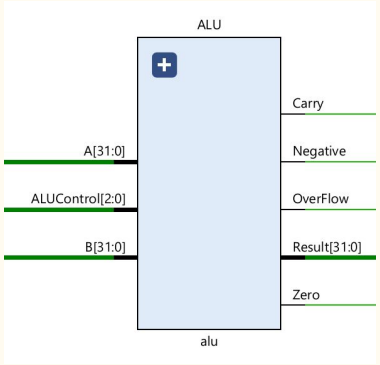
🧠 **Explanation:**

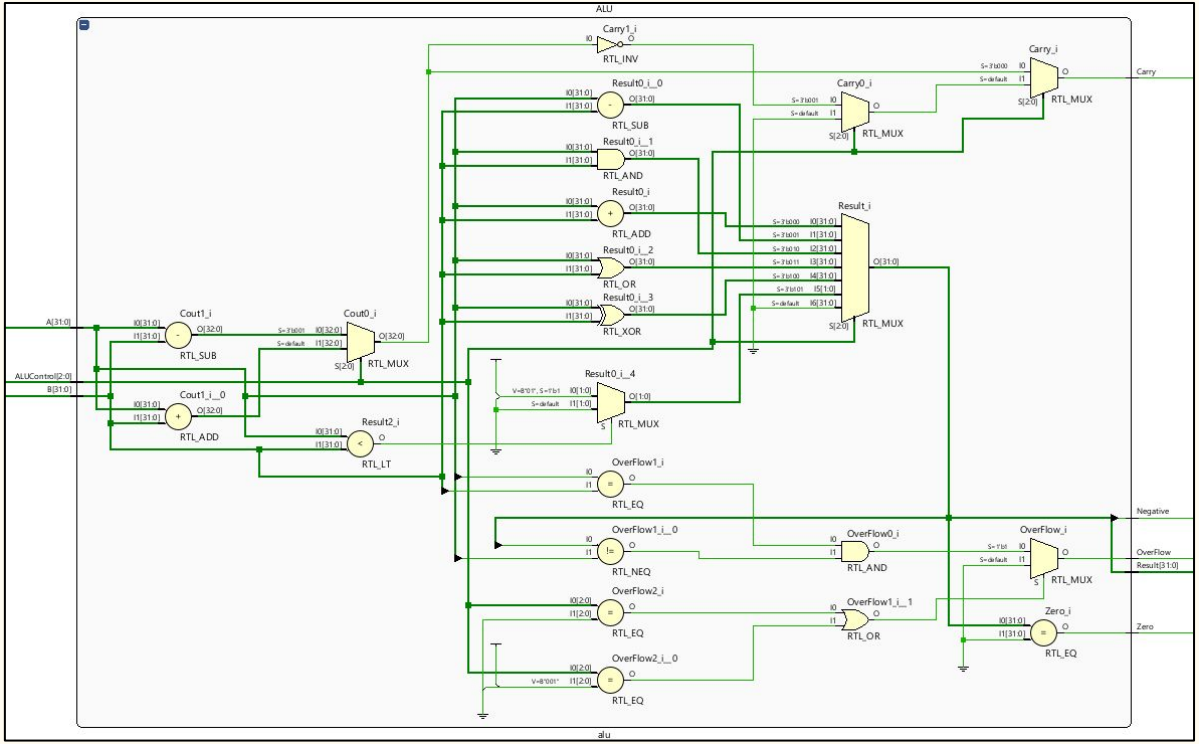
- The control logic decides which operation to perform.
- Flags like Zero, Negative, etc., are useful for branch decisions.
- Example: For add x3, x1, x2, ALUControl = 000, Result = x1 + x2

Part of the **Execution (EX)** stage.

Schematic of ALU module

Outer Schematic showing inputs and outputs:





Instruction Memory Module

```
module instr_mem(
    input [31:0] A,
    input rst,
    output [31:0] RD
);
    /// creating memory
    reg [31:0] Mem [1023:0];

    assign RD = (rst) ? 32'b0: Mem[A[31:2]];// doubt

    // initial begin
        // $readmemh("memfile.hex", Mem);
    // end

    initial begin
    // --- I-type ---
    Mem[0]  = 32'h00A00093; // addi x1, x0, 10      ; x1 = 10
    Mem[1]  = 32'h01400113; // addi x2, x0, 20      ; x2 = 20

    // --- R-type ---
    Mem[2]  = 32'h002081B3; // add x3, x1, x2       ; x3 = x1 + x2 = 30
    Mem[3]  = 32'h40110233; // sub x4, x2, x1       ; x4 = x2 - x1 = 10
    Mem[4]  = 32'h0020F2B3; // and x5, x1, x2       ; x5 = x1 & x2 = 0
    Mem[5]  = 32'h0020E333; // or  x6, x1, x2       ; x6 = x1 | x2 = 30

    // --- I-type logic ---
    Mem[6]  = 32'h00F0C393; // xori x7, x1, 15      ; x7 = x1 ^ 15
    Mem[7]  = 32'h00517413; // andi x8, x2, 5       ; x8 = x2 & 5
    Mem[8]  = 32'h00516493; // ori  x9, x2, 5       ; x9 = x2 | 5
```

```
  // --- Memory access ---
    Mem[9]  = 32'h00302023; // sw x3, 0(x0)         ; MEM[0] = x3
    Mem[10] = 32'h00002503; // lw x10, 0(x0)        ; x10 = MEM[0] = 30

        // --- U-type ---
    Mem[11] = 32'h123458B7; // lui x17, 0x12345     ; x17 = 0x12345000
    Mem[12] = 32'h00001917; // auipc x18, 0x1       ; x18 = PC + 0x1000

    // --- Branching ---
    Mem[13] = 32'h00350463; // beq x10, x3, +8      ; skip next
    Mem[14] = 32'h06F00593; // addi x11, x0, 111    ; should be skipped

    Mem[15] = 32'h00209463; // bne x1, x2, +8       ; skip next
    Mem[16] = 32'h0DE00613; // addi x12, x0, 222    ; should be skipped

    // --- Jumps ---//

    Mem[17] = 32'h04C00793; // addi x15, x0, 76     ; x15 = 76
    Mem[18] = 32'h00078867; // jalr x16, x15, 0     ; jump to x15 + 0

    Mem[19] = 32'h008006EF; // jal x13, +8          ; jump 2 instructions forward
    Mem[20] = 32'h06300713; // addi x14, x0, 99     ; should be skipped
end

endmodule
```

Instruction Memory Module

Purpose:

- Stores the **program instructions**.
- Read-only during execution.
- Indexed using the Program Counter address.

Key Features:

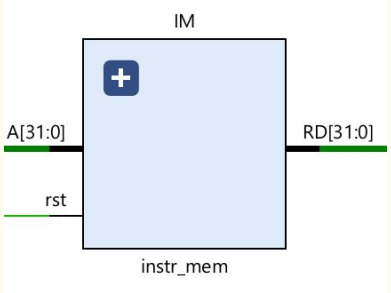
- **Input:**
  - A: Address (from PC)
  - rst: Reset (optional clearing)
- **Output:**
  - RD: 32-bit instruction

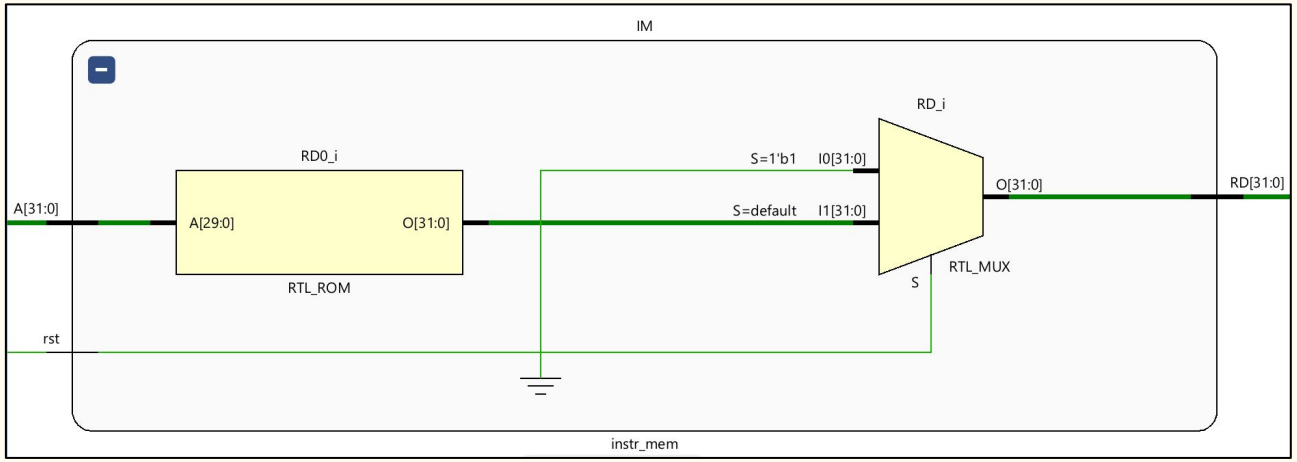
🧠 **Explanation:**

- Instructions are stored as **32-bit words** in an array Mem[1023:0].
- Since words are **4 bytes each**, the address indexing is done using A[31:2] (ignoring lower 2 bits).
- If rst = 1, outputs 0 (no instruction).

Schematic of Instruction Memory

Outer Schematic showing inputs and outputs:





Data Memory Module

Purpose:

- Stores data for lw and sw instructions.

📌 **Inputs:**

- clk: Clock
- WE: Write enable (MemWrite)
- A: Address (from ALU)
- WD: Write data
- rst: Reset

📌 **Output:**

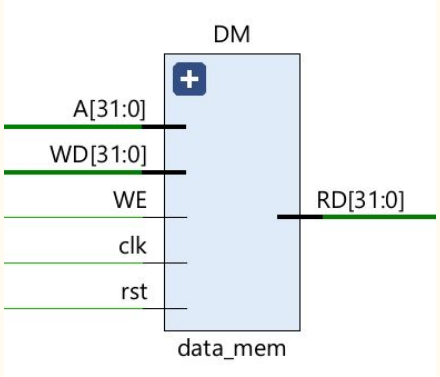
- RD: Read data

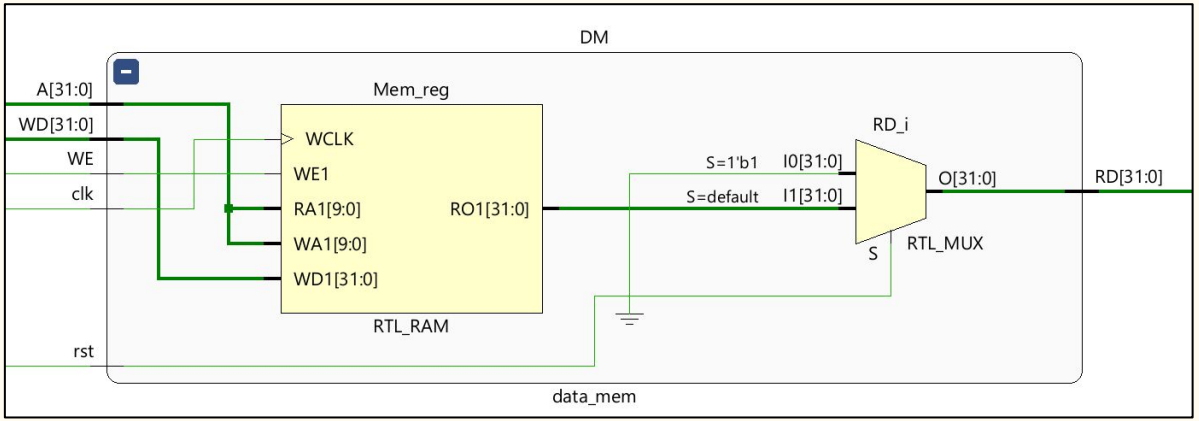
Data is accessed word-aligned using A[31:2].

```
module data_mem(
    input clk,WE,rst,
    input [31:0] A,
    input [31:0] WD,
    output [31:0] RD
);

    reg [31:0] Mem [1023:0];
// write
    always @(posedge clk)
    begin
        if (WE)
            begin
                Mem[A[31:2]] <= WD;
            end
    end

// read

assign RD = (rst) ? 32'd0 : Mem[A[31:2]];

endmodule
```

Schematic of Data Memory

Outer Schematic showing inputs and outputs:

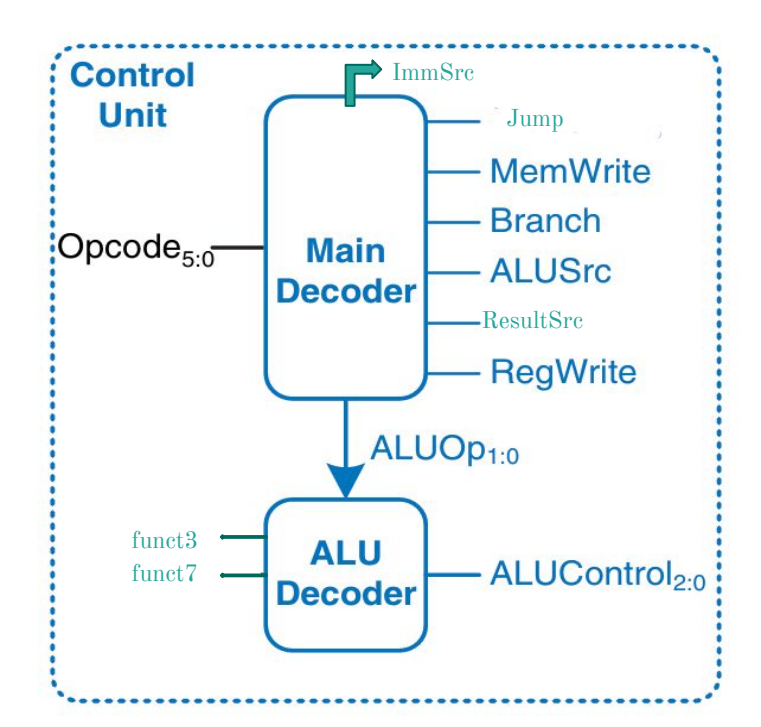




Control unit Module

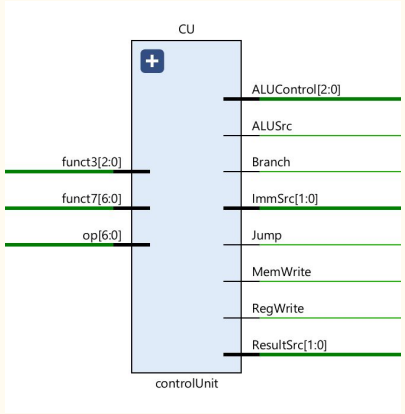
```
module controlUnit(op,RegWrite,ImmSrc,ALUSrc,MemWrite,ResultSrc,
                   Branch,funct3,funct7,ALUControl,Jump);

    input [6:0]op,funct7;
    input [2:0]funct3;
    output RegWrite,ALUSrc,MemWrite,Branch,Jump;
    output [1:0] ResultSrc;
    output [1:0] ImmSrc;
    output [2:0] ALUControl;

    wire [1:0] ALUOp;

    mainDecoder MD(
            .op(op),
            .RegWrite(RegWrite),
            .ImmSrc(ImmSrc),
            .MemWrite(MemWrite),
            .ResultSrc(ResultSrc),
            .Branch(Branch),
            .Jump(Jump),
            .ALUSrc(ALUSrc),
            .ALUOp(ALUOp)
    );
    aluDecoder ALUD(
                    .ALUOp(ALUOp),
                    .funct3(funct3),
                    .funct7(funct7),
                    .op(op),
                    .ALUControl(ALUControl)
    );


endmodule
```

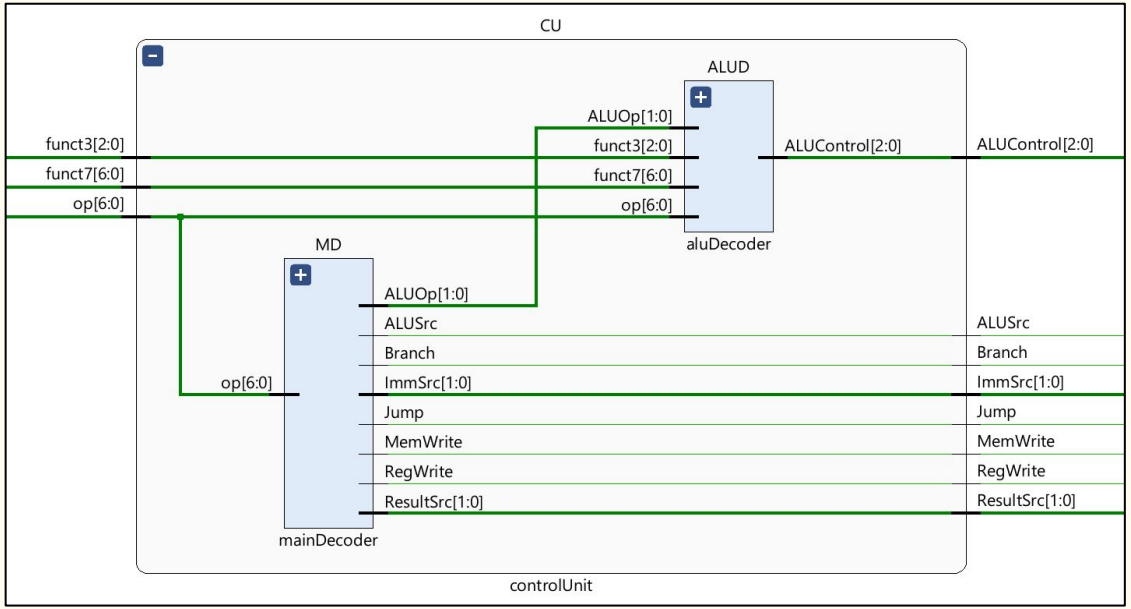
The **control unit** in this RV32I processor design generates control signals based on the opcode and funct fields of the instruction. It coordinates data flow by enabling modules like the ALU, register file, and memory, and selects appropriate paths via multiplexers. This ensures correct execution for each instruction type (R, I, S, SB, U, UJ).



Schematic of Control Unit

Outer Schematic showing inputs and outputs:





ALU DECODER

Purpose:

- Generates the ALUControl signal based on:
  - ALUOp: From main decoder
  - funct3, funct7: From instruction
    If ALUOp = 10, it's an R-type or I-type logic operation.

    The combination of funct3 and funct7 determines the exact operation.

    For example:

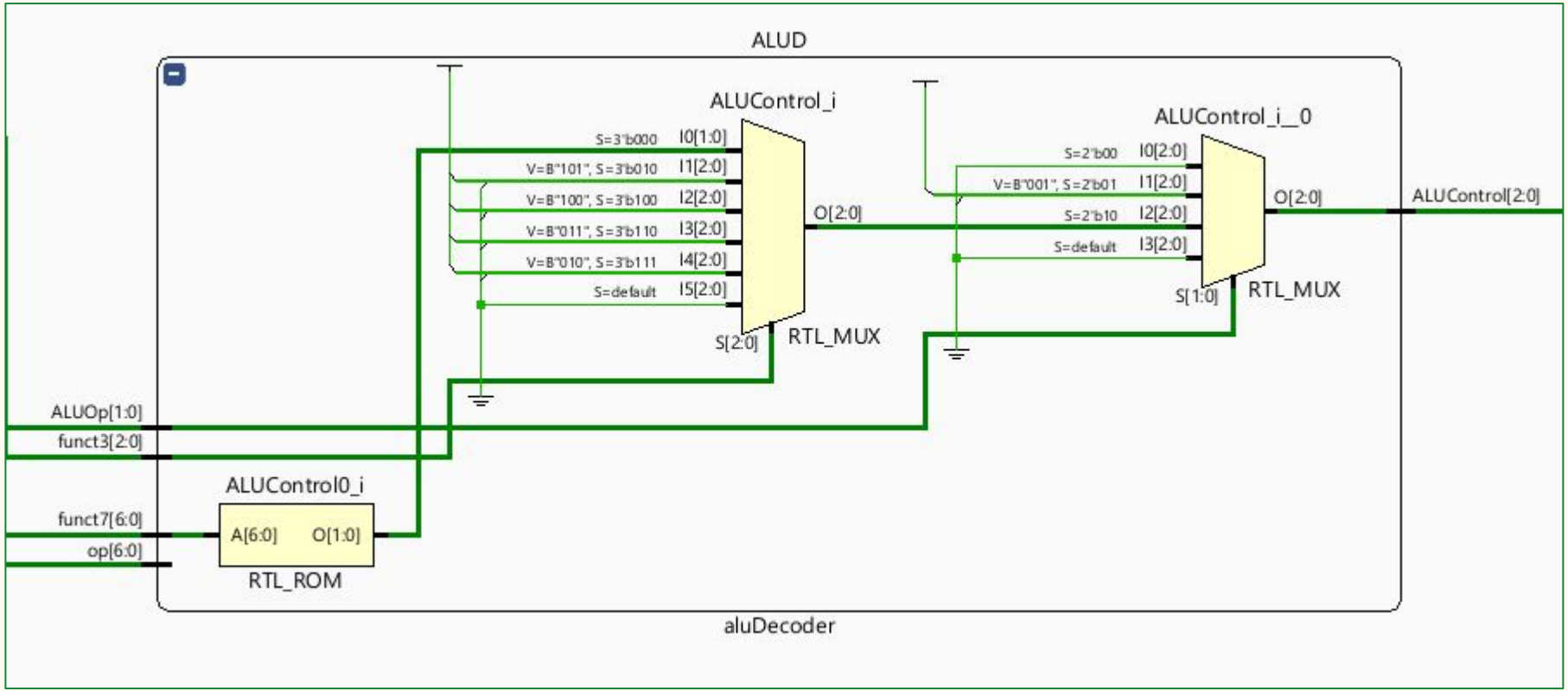
- funct3 = 000, funct7 = 0 → ADD
- funct3 = 000, funct7 = 0x20 → SUB
  - opcode: To distinguish R/I types

ALUOp	funct3	funct7	ALUControl	operation
00	xxx	xxxxxxx	000	lw/sw add
01	xxx	xxxxxxx	001	beq
10	000	0100000	001	SUB
		otherwise	000	ADD
	010	xxxxxxx	101	SLT
	100	xxxxxxx	100	XOR
	110	xxxxxxx	011	OR
	11	xxxxxxx	010	AND
default			000	default

ALU Decoder Module

```
module aluDecoder(
    input [1:0] ALUOp,
    input [6:0] op,
    input [6:0] funct7,
    input [2:0] funct3,
    output reg [2:0] ALUControl

);

always @(*) begin
        case (ALUOp)
            2'b00: ALUControl = 3'b000; // For lw/sw -> ADD
            2'b01: ALUControl = 3'b001; // For beq -> SUB
            2'b10: begin
                case (funct3)
                    3'b000: ALUControl = (funct7 == 7'b0100000) ? 3'b001 : 3'b000; // SUB : ADD
                    3'b010: ALUControl = 3'b101; // SLT
                    3'b100: ALUControl = 3'b100; // XOR
                    3'b110: ALUControl = 3'b011; // OR
                    3'b111: ALUControl = 3'b010; // AND
                    default: ALUControl = 3'b000;
                endcase
            end
            default: ALUControl = 3'b000;
        endcase
    end
endmodule
```

Schematic of ALU Decoder



Main Decoder:

X is considered 0 in the code

opcode	Operation	RegWrite	MemWrite	ResultSrc	ALUSrc	Branch	Jump	ImmSrc	ALUOp
0110011	R-type	1	x	xx	0	0	0	xx	10
0010011	I-type	1	x	xx	1	0	0	00	10
0000011	lw	1	x	01	1	0	0	xx	00
0100011	sw	x	1	xx	1	0	0	01	00
1100011	B-type	x	x	xx	0	1	0	10	01
0110111	lui	1	x	xx	1	0	0	11	00
0010111	auipc	1	x	xx	1	0	0	11	00
1101111	jal	1	x	xx	x	0	1	11	10
1100111	jalr	1	x	xx	1	0	1	00	xx

Main Decoder Module

```
module mainDecoder(
    input [6:0] op,
    output reg RegWrite,
    output reg MemWrite,
    output reg [1:0] ResultSrc,
    output reg ALUSrc,
    output reg Branch,
    output reg Jump,
    output reg [1:0] ImmSrc,
    output reg [1:0] ALUOp
);

    always @(*) begin
        // Default values
        RegWrite   = 0;
        MemWrite   = 0;
        ResultSrc  = 2'b00;
        ALUSrc     = 0;
        Branch     = 0;
        Jump       = 0;
        ALUOp      = 2'b00;
        ImmSrc     = 2'b00;

        case (op)
            7'b0110011: begin // R-type
                RegWrite = 1;
                ALUSrc   = 0;
                ALUOp    = 2'b10;
            end

            7'b0010011: begin // I-type ALU (addi, xori, etc.)
                RegWrite = 1;
                ALUSrc   = 1;
                ALUOp    = 2'b10;
                ImmSrc   = 2'b00;
            end
```

```
            7'b0000011: begin // lw
                RegWrite  = 1;
                ALUSrc    = 1;
                ALUOp     = 2'b00;
                ImmSrc    = 2'b00;
                ResultSrc= 2'b01;
            end

            7'b0100011: begin // sw
                MemWrite = 1;
                ALUSrc   = 1;
                ALUOp    = 2'b00;
                ImmSrc   = 2'b01;
            end

            7'b1100011: begin // B-type: beq, bne
                Branch  = 1;
                ALUSrc  = 0;
                ALUOp   = 2'b01;
                ImmSrc  = 2'b10;
            end

            7'b0110111: begin // lui
                RegWrite = 1;
                ALUSrc   = 1;
                ALUOp    = 2'b00;
                ImmSrc   = 2'b11;
            end

            7'b0010111: begin // auipc
                RegWrite = 1;
                ALUSrc   = 1;
                ALUOp    = 2'b00;
                ImmSrc   = 2'b11;
            end
```

```
            7'b1101111: begin // jal
                RegWrite = 1;
                Jump     = 1;
                ImmSrc   = 2'b11; // reuse B/J format
                ResultSrc = 2'b10; // PC+4
            end

            7'b1100111: begin // jalr
                RegWrite = 1;
                Jump     = 1;
                ALUSrc   = 1;
                ImmSrc   = 2'b00; // I-type format
                ResultSrc = 2'b10; // PC+4
            end

            default: ; // keep defaults
        endcase
    end
endmodule
```

Schematic of Main Decoder

Purpose:

- Generates **high-level control signals** from the opcode.

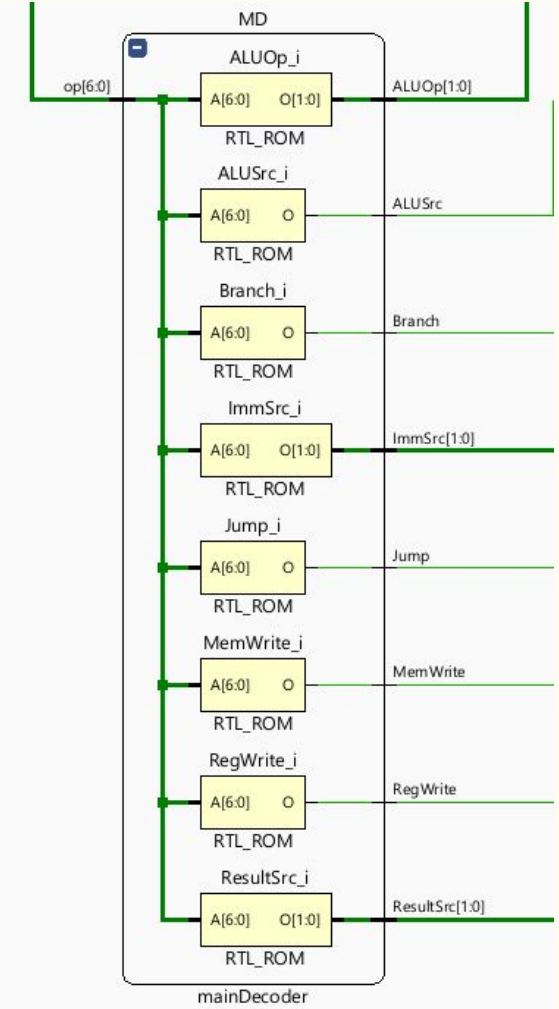
📌 **Outputs:**

- RegWrite: Enable register write
- MemWrite: Enable data memory write
- ALUSrc: Selects ALU input (register or immediate)
- ImmSrc: Chooses immediate format
- ResultSrc: Selects result source (ALU, memory, PC+4)
- Branch, Jump: For PCSelector
- ALUOp: To guide aluDecoder

🧠 **Explanation:**

- Uses case (opcode) to decode:
  - 0110011: R-Type → RegWrite=1, ALUSrc=0
  - 0010011: I-Type → ALUSrc=1
  - 0000011: LW → ResultSrc = memory
  - 1100011: Branch
  - 1101111: JAL

Central to processor control; guides rest of the datapath.



```
module ImmGen(
    input [31:0] instr,
    input [6:0] opcode,
    input [1:0] ImmSrc,
    output reg [31:0] ImmExt
);

    always @(*) begin
        case (ImmSrc)
            2'b00: // I-type (lw, addi, etc.)
                ImmExt = {{20{instr[31]}}, instr[31:20]};

            2'b01: // S-type (sw)
                ImmExt = {{20{instr[31]}}, instr[31:25], instr[11:7]};

            2'b10: // B-type (beq, bne, etc.)
                ImmExt = {{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0};

            2'b11: begin // U-type (lui, auipc) or J-type (jal)
                if (opcode == 7'b1101111) begin
                    // J-type (jal)
                    ImmExt = {{11{instr[31]}}, instr[31], instr[19:12], instr[20], instr[30:21], 1'b0};
                end else begin
                    // U-type (lui, auipc)
                    ImmExt = {instr[31:12], 12'b0};
                end
            end

            default:
                ImmExt = 32'b0;
        endcase
    enD
endmodule
```

Immediate Generator Module

Purpose:

- Extracts and sign-extends the immediate field from an instruction based on its format.

📌 Inputs:

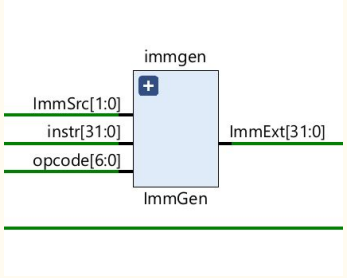
- instr: The 32-bit instruction
- opcode, ImmSrc: Instruction type selector

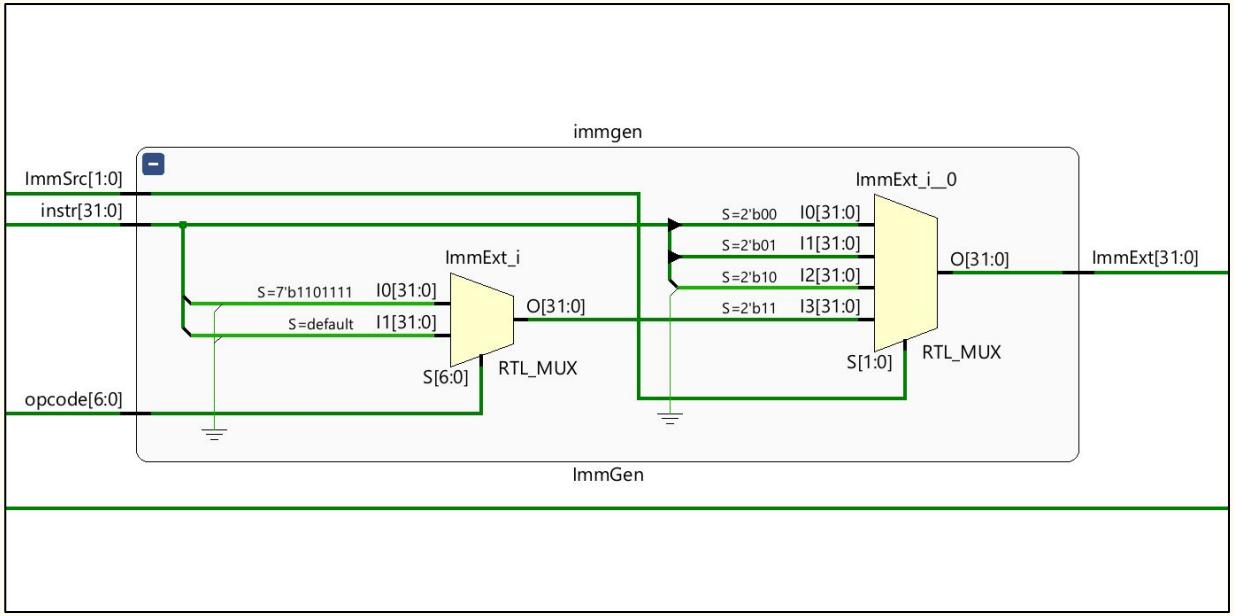
📌 Output:

- ImmExt: 32-bit sign-extended immediate

Schematic of Immediate generator

Outer Schematic showing inputs and outputs:





Top Module (Complete Processor)

```
module top(
    input clk,rst
);
    wire [31:0] PCw,ImmExt,Instr,ALUResult,
                SrcB,SrcA,ReadData,WriteData,
                PCplus4,Result,PCBranch,
                PCNext,jalrTarget;
    wire [2:0] funct3;
    wire [2:0] ALUControl;
    wire [6:0] funct7, opcode;
    wire [1:0] ImmSrc,ResultSrc;
    wire RegWrite,MemWrite,ALUSrc,Branch,
                Zero,takeBranch,
                Carry,OverFlow,Jump;

     // === Instruction Fields ===
    assign funct3 = Instr[14:12];
    assign funct7 = Instr[31:25];
    assign opcode = Instr[6:0];

    pc PC(
        .clk(clk),.rst(rst),
        .PC(PCw),.PCNext(PCNext)
    );
    PCSelector PCS(
        .PCw(PCw),.ImmExt(ImmExt),
        .SrcA(SrcA),.Branch(Branch),
        .opcode(opcode),.WriteData(WriteData),
        .Jump(Jump),.Zero(Zero),
        .Negative(Negative),
        .OverFlow(OverFlow),
        .Carry(Carry),.funct3(funct3),
        .PCNext(PCNext),
        .PCplus4(PCplus4),
        .takeBranch(takeBranch),
        .PCBranch(PCBranch),
        .jalrTarget(jalrTarget)
    );
```

```
instr_mem IM(
        .rst(rst),
        .A(PCw),
        .RD(Instr)
    );

    regfile REGFILE (
        .rst(rst),
        .clk(clk),
        .A1(Instr[19:15]),
        .A2(Instr[24:20]),
        .A3(Instr[11:7]),
        .WE3(RegWrite),
        .WD3(Result),
        .RD1(SrcA),
        .RD2(WriteData)
    );

    ImmGen immgen (
    .instr(Instr),
    .opcode(opcode),
    .ImmSrc(ImmSrc),
    .ImmExt(ImmExt)
);

    assign SrcB = ALUSrc ? ImmExt :
WriteData;

    alu ALU(
        .A(SrcA),
        .B(SrcB),
        .Result(ALUResult),
        .ALUControl(ALUControl),
        .OverFlow(OverFlow),
        .Carry(Carry),
        .Zero(Zero),
        .Negative(Negative)

    );
```

```
data_mem DM(
        .clk(clk),
        .WE(MemWrite),
        .rst(rst),
        .WD(WriteData),
        .RD(ReadData),
        .A(ALUResult)
    );


    controlUnit CU (
        .op(opcode),
        .RegWrite(RegWrite),
        .ImmSrc(ImmSrc),
        .ALUSrc(ALUSrc),
        .MemWrite(MemWrite),
        .ResultSrc(ResultSrc),
        .Branch(Branch),
        .Jump(Jump),
        .funct3(funct3),
        .funct7(funct7),
        .ALUControl(ALUControl)
        );


    // === Result MUX: ALU, Mem, PC+4
===

    mux3 MUX_Result (
        .a(ALUResult),
        .b(ReadData),
        .c(PCplus4),
        .sel(ResultSrc), // Jump=1 → PC+4
        .out(Result)
    );

endmodule
```

```
module mux3(
    input [31:0] a, b, c,
    input [1:0] sel,
    output reg [31:0] out
);
    always @(*) begin
        case(sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10,
            2'b11: out = c; // PC+4
        endcase
    end
endmodule
```

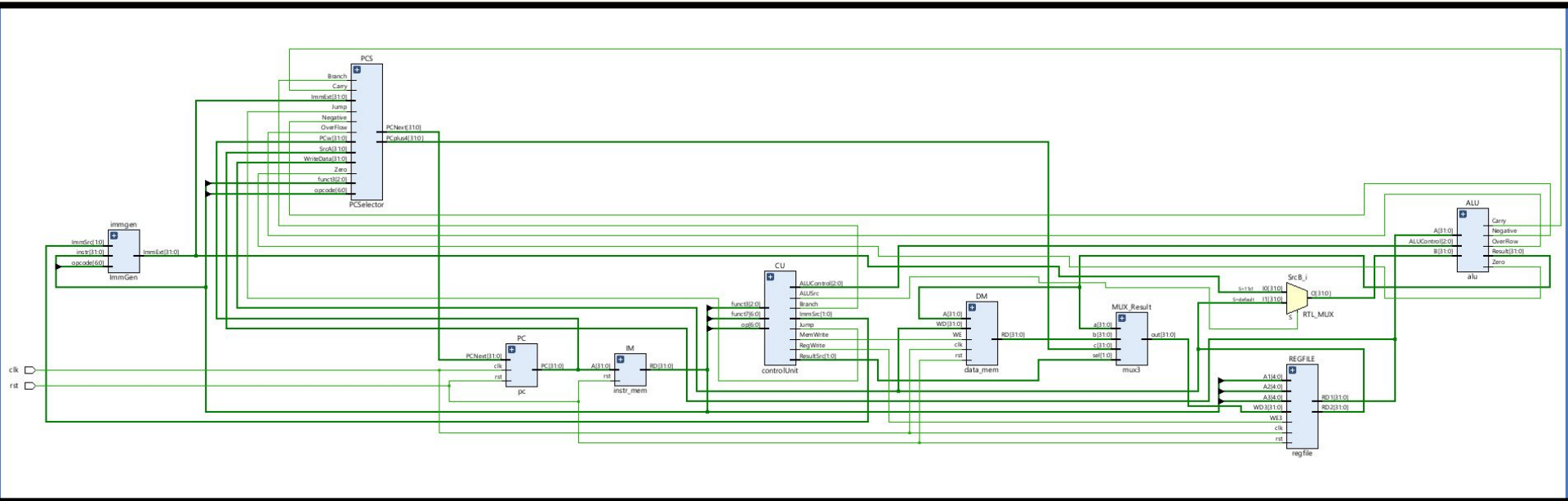
Schematic of The Complete Processor

Purpose:

- Integrates all submodules.
- Controls instruction execution flow.
- Central module to simulate the processor.

Includes:

- Instantiations of PC, PCSelector, ALU, RegFile, ControlUnit, ImmGen, etc.
- Connects outputs from one module as inputs to another.



Testbench

This testbench has been developed to validate the design and functionality of the custom RV32I-based integer microarchitecture. It simulates the sequential execution of instructions stored in the instruction memory, operating at a rate of **one instruction per clock cycle**. The simulation is performed using **Vivado**, and the output is verified through **waveform analysis**.

Purpose

The primary objective of this testbench is to:

- Apply a controlled clock and reset sequence.
- Initialize the processor state.
- Simulate program execution.
- Record waveform data for functional verification.

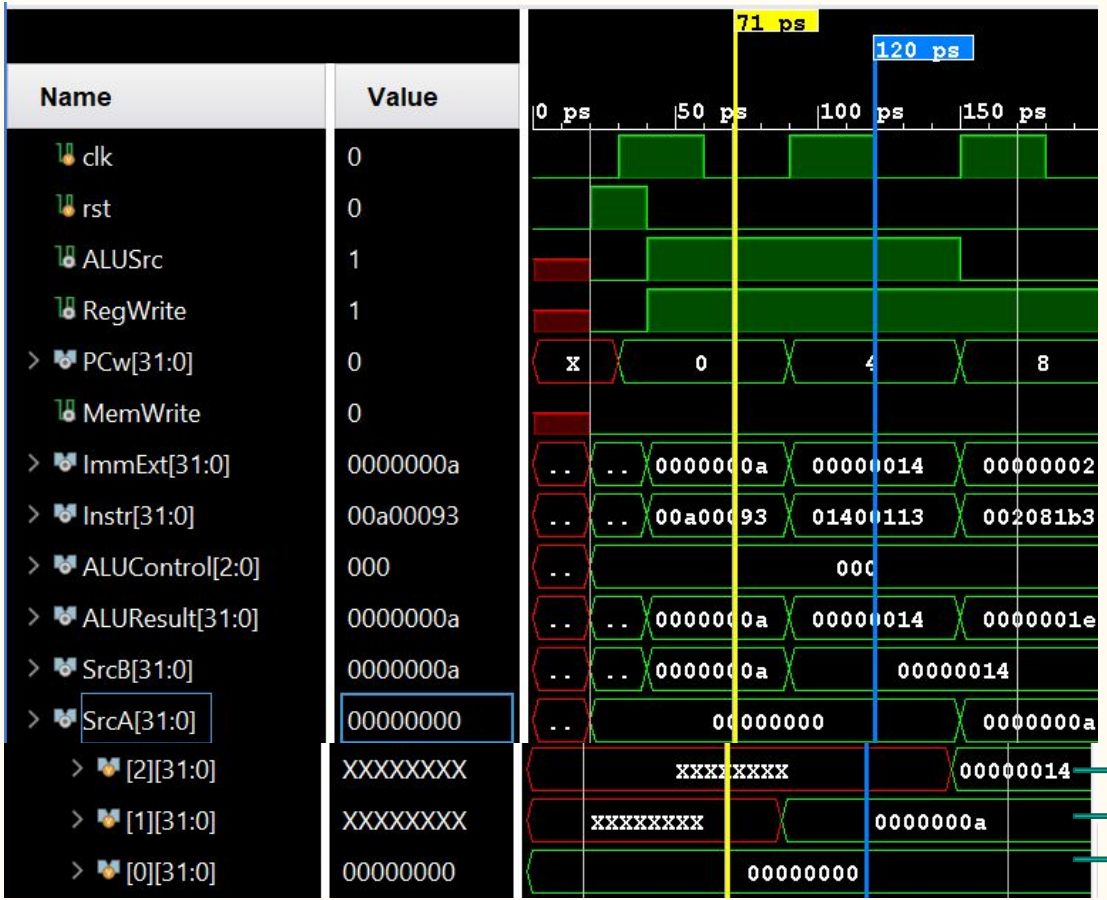
Key Features

- **Clock Generation**: A periodic clock signal is generated with a time period of 60 ps (30 ps high, 30 ps low) to drive the processor.
- **Reset Sequencing**: The testbench applies a reset pulse to ensure the processor starts in a known state.
- **Waveform Dumping**: Using $dumpfile and $dumpvars, simulation data is exported in .vcd format for visualization in waveform viewers like GTKWave.
- **Execution Time Frame**: The test runs for a fixed number of cycles (1500 time units post-reset) and then terminates using $finish.

```
`timescale 1ps/1ps
module top_tb();

    reg clk=1'b1,rst;

    top TOP(
        .clk(clk),
        .rst(rst)
    );

    initial begin
        $dumpfile("SingleTop.vcd");
        $dumpvars(0);

    end

    always
    begin
        clk = ~ clk;
        #30;

    end
```

```
    initial
    begin
        rst<=0;
        #20
        rst <= 1'b1;
        #20;

        rst <=1'b0;
        #1500;
        $finish;
    end
endmodule
```

Simulation Results:Initialization Instructions



0: addi x1, x0, 10 | 1: addi x2, x0, 20

Both instructions write into the RegFile so RegWrite=1

0:

ImmExt=A, wich is connected to SrcB

ALU: SrcA=0 SrcB=A, ALUResult=A

And for Write back, Result=A

1:

ImmExt=14H, wich is connected to SrcB

ALU: SrcA=0H SrcB=14H, ALUResult=14H

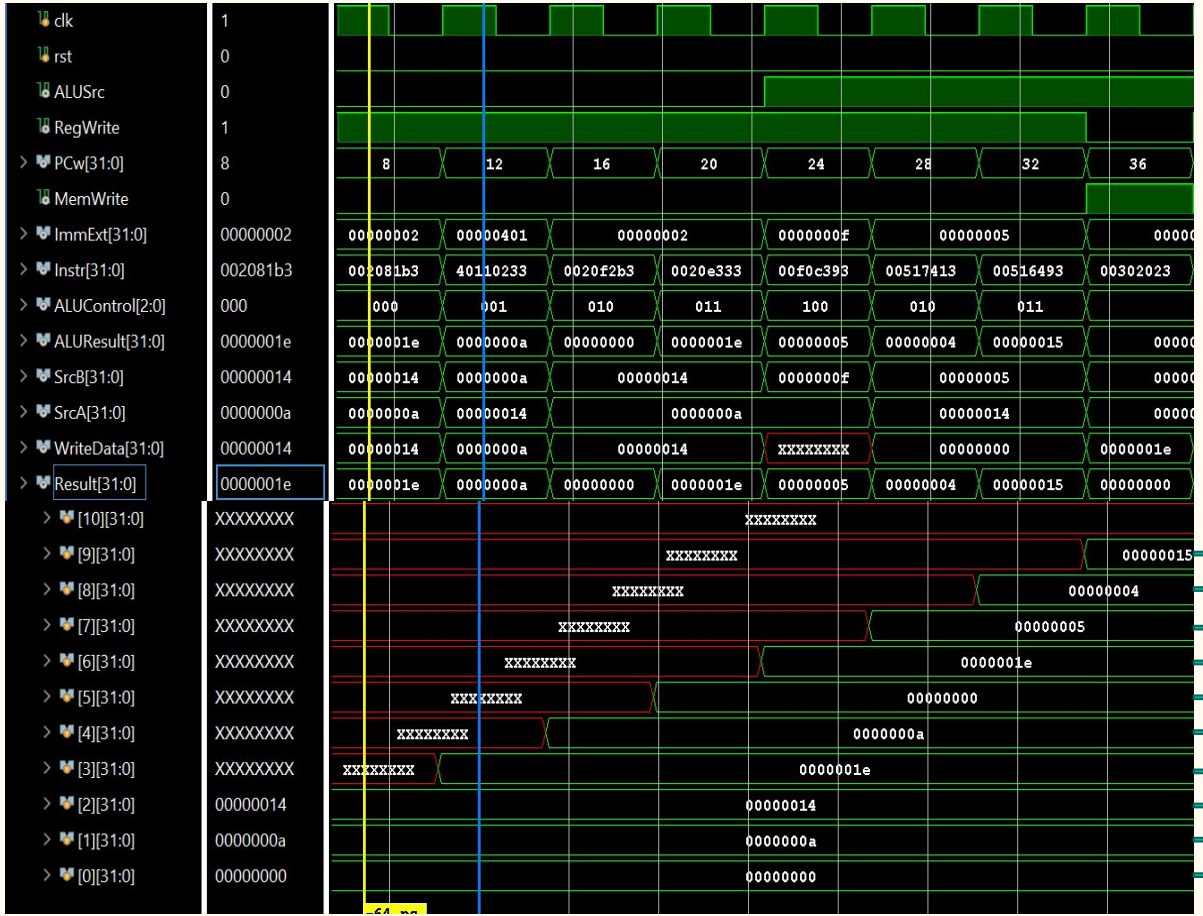
And for Write back, Result=14H

x2=14H=20

x1=AH=10

x0=always 0

Simulation Results:Arithmetic & Logic Ops



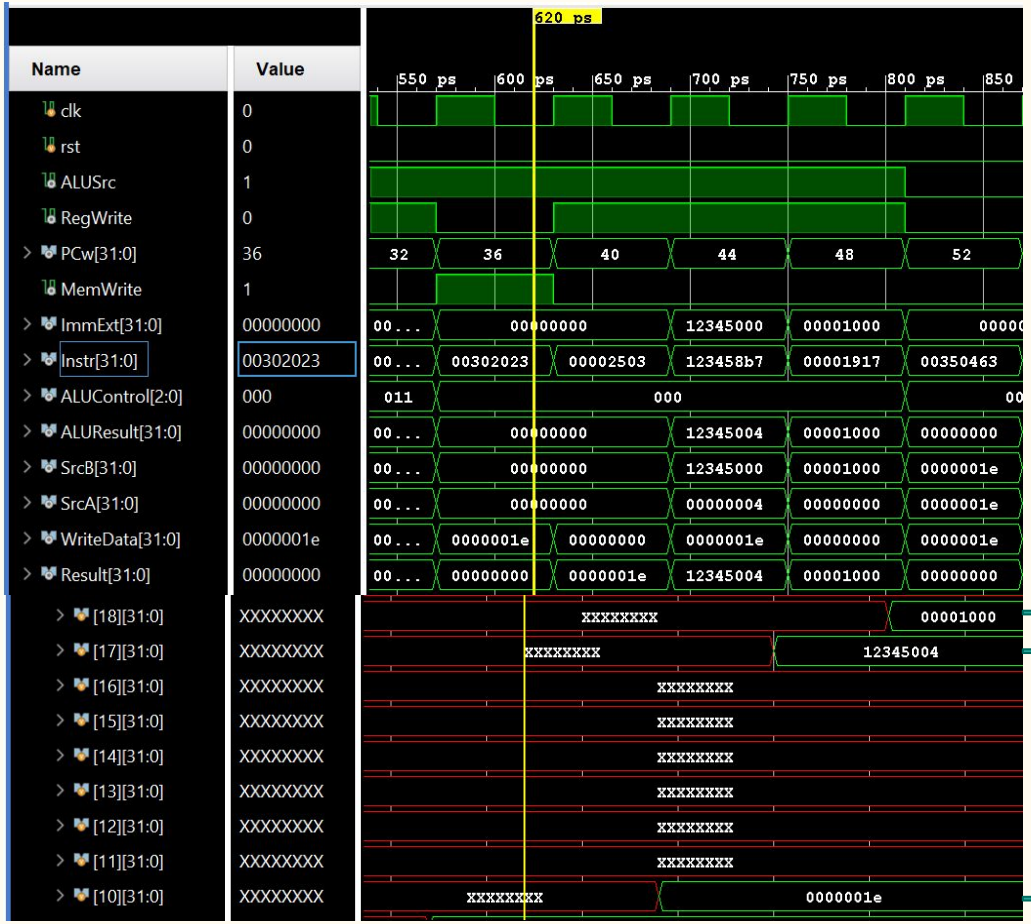
2: add x3, x1, x2
3: sub x4, x2, x1
4: and x5, x1, x2
5: or x6, x1, x2
6: xori x7, x1, 15
7: andi x8, x2, 5
8: ori x9, x2, 5

All the instruction write in to Reg. so RegWrite=1.
ALUControl also matches with the instruction, acc. to table

Lets calculate the results:
2:[x3]=[x1]+[x2]=10+20=30=1e
3:[x4]=[x2]-[x1]=20-10=10
4:[x5]=[x1]&[x2]=01010 & 10100=00000
5:[x6]=[x1] | [x2]=01010 | 10100=11110=30=1e
6:[x7]=[x1]^15=1010 ^ 1111 = 0101=5
7:[x8]=[x2]&5=10100&00101=00100=4
8:.[x9]=[x2] | 5=10100 | 00101=10101=21=15H

x9=21=15H
x8=4
x7=5
x6=30=1EA
x5=0=0H
x4=10=AH
x3=1EH=30
x2=14H=20
x1=AH=10
x0=always 0

Simulation Results:Memory Access + Immediate



9: sw x3, 0(x0) **10: lw x10, 0(x0)**
11: lui x17, 0x12345 **12: auipc x18, 0x1**

Only sw writesinto Mem,so MemWrite=1 @9
Rest all write into register, so RegWrite=1
ALUControl for sw,lw,lui,lupc=000(add)

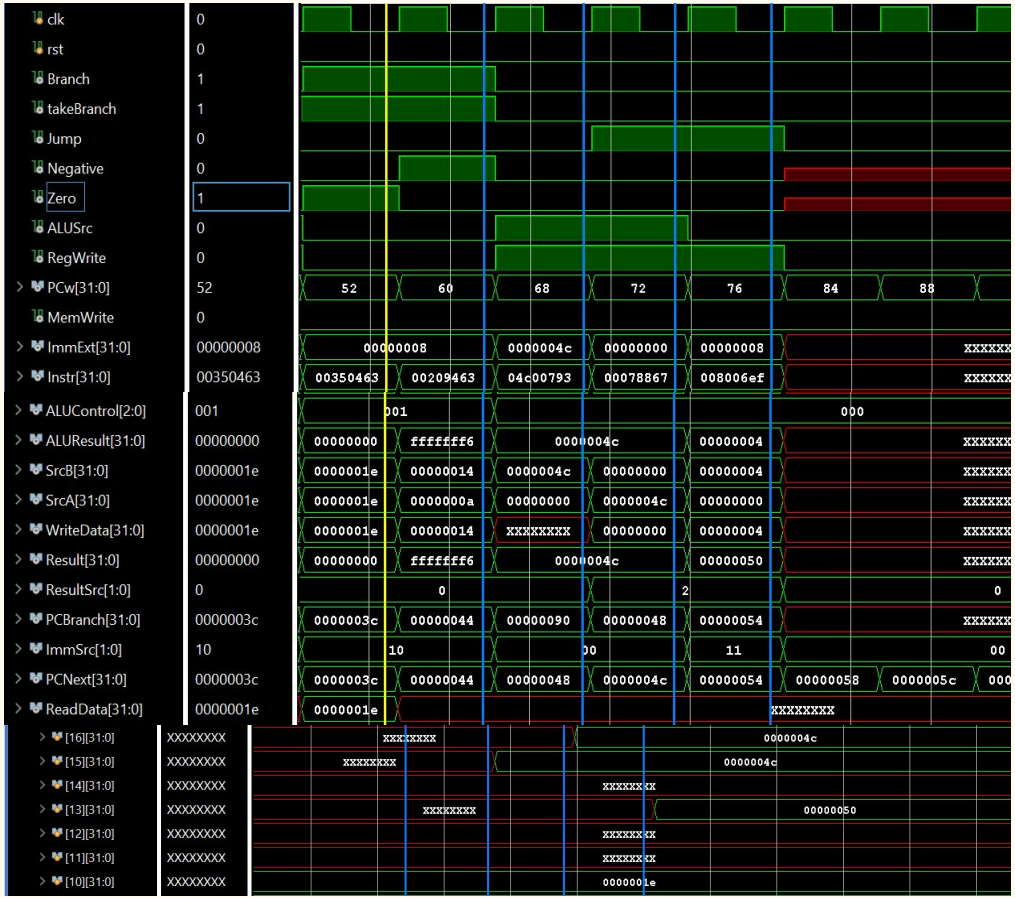
sw:stores [x3]=30=1e into Data memory(x0); Writedata=1e and no data in/out from alu.

lw:loads regfile from data memory.
regfile[x10]=datamem[x0]=30=1e

lui:[x17]=

auipc:[x18]=

Simulation Results: Branch + Jump



13: beq x10, x3, +8 // branch taken
14: addi x11, x0, 111 // skipped
15: bne x1, x2, +8 // branch taken
16: addi x12, x0, 222 // skipped
17: addi x15, x0, 76
18: jalr x16, x15, 0
19: jal x13, +8
20: addi x14, x0, 99 // skipped

Branch:ALUControl=001(sub to compare)

13: Branch=1; [x10]=[x3]-->30=30, so takeBranch=1=Zero
PC+8=52+8=60

14: skipped

15:[x1]!=[x2]--> 10!=20 takenBranch=1 PC+8=60+8=68

16:skipped

17: [x15]=76

18: Jump=1; PCNext=76; rd=x16=PC+4=76

19:Jump=1; PCNext=PC+8=76+8=84;
No instruction at 84, so end.

Why RISC-V Matters in Defence Systems

RISC-V is an open-source ISA, allowing full visibility and modification of instruction sets.

It offers **modularity**, enabling creation of minimal or highly custom processors—ideal for different types of defence systems.

No dependence on foreign vendors like ARM or Intel, eliminating backdoors or supply chain risks.

India is increasingly pushing for **indigenous chip design**—RISC-V plays a key role in achieving **AtmaNirbhar Bharat** in strategic electronics.
Ideal for low-power, high-performance embedded applications common in **missiles, UAVs, communication systems, and radar**.

Global defence leaders like **DARPA** and **ISRO** have already invested in RISC-V projects

India's RISC-V Revolution: Toward Semiconductor Self-Reliance

Pune Times Mirror Epaper Pune PCMC Entertainment News Sports Fab Champ

HOME ▸ NEWS ▸ INDIA ▸ INDIA LAUNCHES VEGA MICROPROCESSORS, PAVING THE WAY FOR TECHNOLOGICAL INDEPEI

India Launches VEGA Microprocessors, Paving the Way for Technological Independence

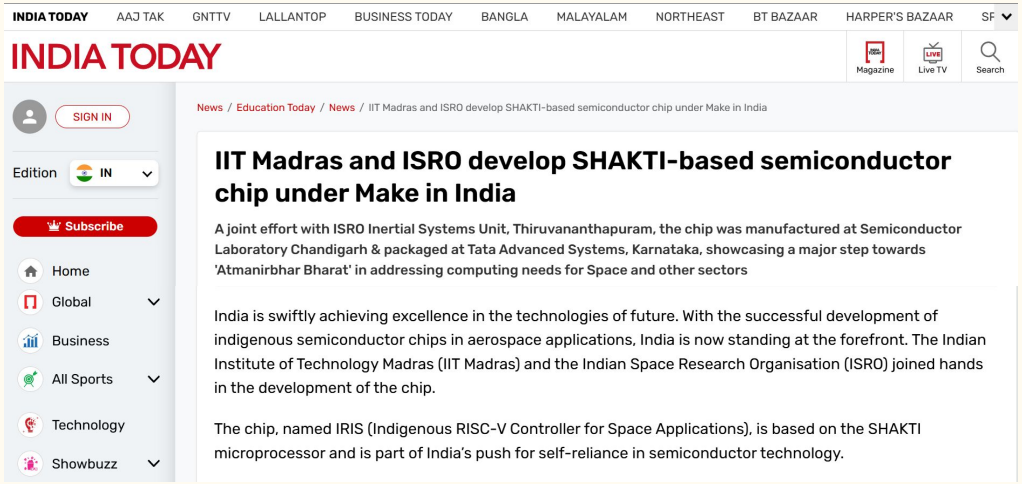
This underscores India's drive towards a secure and self-reliant technological ecosystem

By PuneMirror Bureau | Reported By Prince Chaudhuri | Mon, 13 Jan 2025 | 02:52 pm

India Launches VEGA Microprocessors, Paving the Way for Technological Independence

As India positions itself to lead in the global electronics market, the Government's DIR-V program has taken a significant step forward with C-DAC's successful design and development of the VEGA series of microprocessors. This series includes India's first indigenous 64-bit multi-core RISC-V based Superscalar Out-of-Order Processor, setting a new benchmark in highperformance computing.

The VEGA microprocessors are based on the RISC-V Instruction Set Architecture and feature 32/64-bit Single/Dual/Quad Core superscalar Out-of-Order high-performance processor cores. During a recent event, Ashwini Vaishnaw, Minister o f Railways , M inistery of Information and Broadcasting, and Minister of Electronics and Information Technology, along with senior officials from MeitY, launched a VEGA-based SoC ASIC and two DIR-V VEGA processorbased development boards, marking a milestone in India's technological journey.



INDIA TODAY AAJ TAK GNTTV LALLANTOP BUSINESS TODAY BANGLA MALAYALAM NORTHEAST BT BAZAAR HARPER'S BAZAAR SF

INDIA TODAY

Magazine Live TV Search

SIGN IN

Edition IN

Subscribe

Home
Global
Business
All Sports
Technology
Showbuzz

News / Education Today / News / IIT Madras and ISRO develop SHAKTI-based semiconductor chip under Make in India

IIT Madras and ISRO develop SHAKTI-based semiconductor chip under Make in India

A joint effort with ISRO Inertial Systems Unit, Thiruvananthapuram, the chip was manufactured at Semiconductor Laboratory Chandigarh & packaged at Tata Advanced Systems, Karnataka, showcasing a major step towards 'Atmanirbhar Bharat' in addressing computing needs for Space and other sectors

India is swiftly achieving excellence in the technologies of future. With the successful development of indigenous semiconductor chips in aerospace applications, India is now standing at the forefront. The Indian Institute of Technology Madras (IIT Madras) and the Indian Space Research Organisation (ISRO) joined hands in the development of the chip.

The chip, named IRIS (Indigenous RISC-V Controller for Space Applications), is based on the SHAKTI microprocessor and is part of India's push for self-reliance in semiconductor technology.

India is making significant strides in achieving technological independence with indigenous semiconductor developments. The successful launch of VEGA microprocessors and the SHAKTI-based IRIS chip by IIT Madras and ISRO showcase the country's growing capabilities in RISC-V architecture. These initiatives mark critical milestones under the 'Make in India' and 'Atmanirbhar Bharat' missions, paving the way for a secure, self-reliant electronics ecosystem.

RISC-V in Critical Military Applications

- **Missiles & Tactical Control Units**
  - RISC-V cores used in embedded guidance processors.
  - Allows custom ALUs for telemetry, trajectory computation.
  - Redundant dual-core designs with hardware lockstep for fail-safe operation.

- **UAVs & Autonomous Defence Vehicles**
  - RISC-V cores used in real-time flight control and onboard decision-making.
  - Integrates with AI accelerators for obstacle detection and target tracking.
  - Secure firmware updates through bootROM with hashing.

- **Secure Military Communication Devices**
  - Hardware cryptographic engines implemented using custom RISC-V extensions (e.g., AES, RSA, SHA).
  - Instructions for constant-time execution to prevent timing attacks.
  - Configurable logic blocks to add public key security.

- **Edge Sensor Networks & Radar Signal Units**
  - Lightweight RISC-V cores (like RV32E) for edge signal processing.
  - Enables mesh networking in battlefield with integrated MAC/PHY layers.
  - Data encryption + anti-jamming protocols using hardware modules.

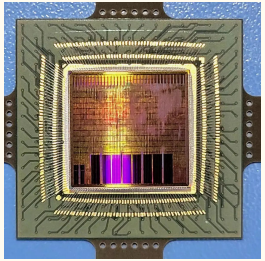
Securing Military Hardware with RISC-V

🧠 **SHAKTI (IIT Madras)**

- India's first open-source RISC-V processor family.
- **T-Class**: Secure core for encryption, authentication, space applications (used by ISRO's IRNSS).
- All source code publicly available → full auditability by DRDO and others.

🛰️ **IRIS (ISRO + IITM, 2025)**

- Designed for **inertial navigation systems**.
- Radiation-hardened 64-bit RISC-V for spaceborne and missile-grade applications.
- Co-developed with a focus on low jitter and high timing accuracy.



🔧 **VEGA Processors (C-DAC)**

- 32-bit & 64-bit RISC-V cores targeting **defence-grade IoT and command systems**.
- Support for real-time processing, lightweight security mechanisms, and low-power modes.

🏛️ **DIR-V Program (MeitY)**

- Central government initiative to boost RISC-V core design for domestic industry.
- Supports startups and academic labs building processors for Indian armed forces.

What’s Next in RISC-V for Defence

- **Secure Multi-Core for Radar Signal Processing**

Multi-core RISC-V processors with custom DSP and crypto extensions enable real-time encrypted radar data processing. Ideal for airborne surveillance and mobile battlefield radar systems requiring high-throughput + tamper-resistant compute.

- **Crypto-Enhanced Pipelined Cores for UAVs**

Custom pipelined RISC-V cores with AES/SHA instructions power secure flight control, telemetry, and AI inference on drones. Hardware crypto + low-latency pipeline = optimal performance and airspace security.

- **Memory-Isolated SoCs for Defence Data Centers**

RISC-V SoCs with capability-based memory protection and secure boot ensure robust, tamper-proof computing in command centers. Prevents malware spread and safeguards critical military data.

✅ RISC-V = Secure, Custom, Indigenous Computing for Strategic Defence

HECTOR-V: A Secure RISC-V TEE Architecture

Overview:
HECTOR-V proposes a heterogeneous CPU design for building secure Trusted Execution Environments (TEEs) by integrating a dedicated secure processor (RVSCP) into a RISC-V SoC. Unlike traditional TEEs like Intel SGX and ARM TrustZone, HECTOR-V ensures stronger isolation, secure I/O, and resilience against side-channel attacks.

Key Features:

- **Dedicated Secure Core:** RVSCP enforces control-flow integrity and handles trustlets with minimal software attack surface.
- **Hardware Security Monitor:** Manages fine-grained access control to peripherals using identifier-based access and ownership transfer.
- **Secure I/O Paths:** Enforces per-core and per-process access to devices via hardware firewalls.
- **Flexible Ownership:** Security monitor ownership can be dynamically transferred to adapt use cases like secure boot or trustlet execution.

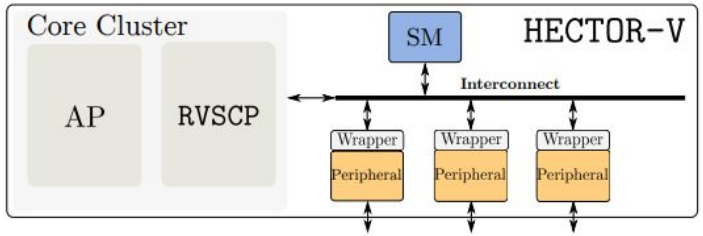


Figure 1: Overall HECTOR-V design.

Outcome:
HECTOR-V enhances TEE security by reducing the Trusted Computing Base, securing peripheral access, and leveraging hardware-enforced domain separation within a RISC-V platform.

References

1. RISC-V Instruction Set Manual, Vol.1: User Level ISA, Version 2.1
2. Digital Design and Computer Architecture- David Money Harris & Sarah L. Harris
3. https://riscv.org/
4. NPTEL Lectures by Prof. Indranil Sengupta