Microarchitecture Design and Verification

Title:

Design of a 5-Stage Pipelined RISC-V RV32I Processor

Abstract:

This project implements a 5-stage pipelined processor based on the RISC-V 32-bit Integer Instruction Set (RV32I). The supported instructions are:

- **Arithmetic/Logical:** ADD, ADDI, SUB, AND, OR, XOR
- **Memory Access:** LW, SW
- **Control Flow:** BEQ, JAL

The goal is to design, implement, and verify the processor using **VHDL/Verilog**, with pipeline integration, hazard handling, and testbenches. The design will be documented and optimized for clarity and performance.

Instruction Set Subset (Supported Instructions)

Category	**Instruction**	**Description**
Arithmetic	ADD	Register + Register
	ADDI	Register + Immediate
	SUB	Register - Register
Logical	AND	Bitwise AND
	OR	Bitwise OR
	XOR	Bitwise XOR
Memory	LW	Load Word (from memory to register)
	SW	Store Word (from register to memory)
Control Flow	BEQ	Branch if Equal
	JAL	Jump and Link (PC-relative jump + link)

Instruction Formats:

- **ADD**

Encoding

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0 0 0 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1
					opcode

Format

add rd,rs1,rs2

Description

Adds the registers rs1 and rs2 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.

Implementation

x[rd] = x[rs1] + x[rs2]

- **SUB**

Encoding

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0 1 1 0 0 0 0	rs2	rs1	0 0 0	rd	0 1 1 0 0 1 1
					opcode

Format

sub rd,rs1,rs2

Description

Subs the register rs2 from rs1 and stores the result in rd. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result.

Implementation

x[rd] = x[rs1] – x[rs2]

- ADDI

Encoding

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11: 0]	rs1	0 0 0	rd	0 0 1 0 0 1 1
				opcode

Format
addi rd,rs1,imm

Description
Adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored 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.

Implementation
x[rd] = x[rs1] + sext(immediate)

- <u>**AND**</u>

Encoding

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0 0 0 0 0 0 0	rs2	rs1	1 1 1	rd	0 1 1 0 0 1 1
					opcode

Format
and rd,rs1,rs2

Description
Performs bitwise AND on registers rs1 and rs2 and place the result in rd

Implementation
x[rd] = x[rs1] & x[rs2]

- <u>**OR**</u>

Encoding

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0 0 0 0 0 0 0	rs2	rs1	1 1 0	rd	0 1 1 0 0 1 1
					opcode

Format
or rd,rs1,rs2

Description
Performs bitwise OR on registers rs1 and rs2 and place the result in rd

Implementation
x[rd] = x[rs1] | x[rs2]

- **<u>XOR</u>**

Encoding

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0
0 0 0 0 0 0 0	rs2	rs1	1 0 0	rd	0 1 1 0 0 1 1
					opcode

Format

xor rd,rs1,rs2

Description

Performs bitwise XOR on registers rs1 and rs2 and place the result in rd

Implementation

x[rd] = x[rs1] ^ x[rs2]

- **<u>LW</u>**

Encoding

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:0]	rs1	0 1 0	rd	0 0 0 0 0 1 1
				opcode

Format

lw rd,offset(rs1)

Description

Loads a 32-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

Implementation

x[rd] = sext(M[x[rs1] + sext(offset)][31:0])

- **<u>SW</u>**

Encoding

31 … 20	19 … 15	14 … 12	11 … 7	6 … 0
imm[11:0]	rs1	0 1 0	rd	0 0 0 0 0 1 1
				opcode

Format
lw rd,offset(rs1)

Description
Loads a 32-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.

Implementation
x[rd] = sext(M[x[rs1] + sext(offset)][31:0])

- **BEQ**

Encoding

<table>
<tr><td>31 25</td><td>24 20</td><td>19 15</td><td>14 12</td><td>11 7</td><td>6 0</td></tr>
<tr><td>imm[12|10:5]</td><td>rs1</td><td>rs1</td><td>0 0 0</td><td>imm[4:1|11]</td><td>1 1 0 0 0 1 1</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td>opcode</td></tr>
</table>

Format
beq rs1,rs2,offset

Description
Take the branch if registers rs1 and rs2 are equal.

Implementation
if (rs1 == rs2) pc += sext(offset)

- **JAL**

Encoding

<table>
<tr><td>31 25</td><td>24 20</td><td>19 15</td><td>14 12</td><td>11 7</td><td>6 0</td></tr>
<tr><td>imm[11:5]</td><td>rs2</td><td>rs1</td><td>0 1 0</td><td>imm[4:0]</td><td>0 1 0 0 0 1 1</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td>opcode</td></tr>
</table>

Format
jal rd,offset

Description
Jump to address and place return address in rd.

Implementation
x[rd] = pc+4; pc += sext(offset)

Register File:

- **Size: 32 general-purpose registers (x0–x31), each 32 bits wide.**
- **Special Behavior:**
  - **x0 (R0) is hardwired to 0 — any write to this register is ignored, and reads always return 0.**
- **Initialization:**
  - **At reset, all registers are set to 0.**
- **Read/Write Ports:**
  - **2 Read Ports → allows simultaneous reading of two source registers (rs1, rs2).**
  - **1 Write Port → allows writing back the result to destination register (rd) in the Write Back (WB) stage.**
- **Access Latency: One cycle (register values available in the same cycle as access).**

Pipeline Depth:

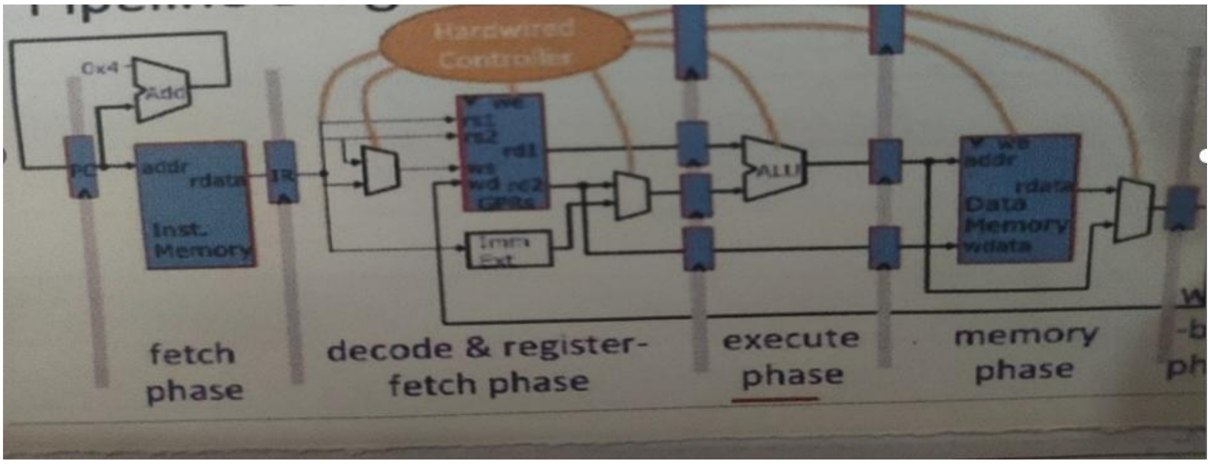
- **Chosen Pipeline: 5-stage classic RISC pipeline** (like MIPS/RISC-V textbooks).
- **Reason for Choice:**
  - Provides a **good balance** between performance and complexity.
  - Well-documented in academic literature (makes debugging + explaining easier).
  - Matches the requirements for your selected instruction subset.
- **Stages Defined:**

1. **Instruction Fetch (IF)**
   - Fetches the instruction from instruction memory.
   - Updates PC to PC + 4 (or branch/jump target if taken).
2. **Instruction Decode / Register Fetch (ID)**
   - Decodes instruction fields (opcode, rd, rs1, rs2, funct3, funct7).
   - Reads register values from the register file.
   - Generates immediate values (via sign-extension).
   - Produces control signals.
3. **Execute / ALU (EX)**
   - Performs arithmetic or logic operations.
   - Calculates branch targets and evaluates branch conditions.
   - Computes effective memory addresses for load/store.

4. **Memory Access (MEM)**
   - Accesses data memory for LW (load) or SW (store).
   - If not a memory instruction, this stage just passes ALU results forward.
5. **Write Back (WB)**
   - Writes result back to the register file (rd).
   - Source can be ALU output or data memory output.

Pipeline Stages

Stage	Name	Operations Performed
IF	Instruction Fetch	Fetch instruction from memory, increment PC
ID	Instruction Decode & Register Read	Decode opcode, read registers, sign-extend immediate
EX	Execute / ALU	Perform arithmetic/logic, compute branch target
MEM	Memory Access	Load/Store to Data Memory
WB	Write Back	Write results back to register file



- **Data Hazards:** The need for one instruction to use data from a prior, unfinished instruction.
Solution: Data Forwarding/Bypassing.

Control Hazards: The uncertainty caused by branches.

Solution: Branch Prediction (simple flush) or Branch Delay Slot (simpler to implement).

Data Forwarding Unit Logic

The Forwarding Unit resolves **Read-After-Write (RAW)** hazards by selecting the most recent value of a register from later pipeline stages (EX/MEM or MEM/WB) and forwarding it to the ALU inputs.

ForwardA (for Rs1)

Condition	Action
If (EX/MEM.RegWrite = 1) AND (EX/MEM.Rd ≠ 0) AND (EX/MEM.Rd = ID/EX.Rs1)	Forward ALU result from EX/MEM to ALU input A
Else if (MEM/WB.RegWrite = 1) AND (MEM/WB.Rd ≠ 0) AND (MEM/WB.Rd = ID/EX.Rs1)	Forward Writeback data from MEM/WB to ALU input A
Else	Use value from Register File

ForwardB (for Rs2)

Condition	Action
If (EX/MEM.RegWrite = 1) AND (EX/MEM.Rd ≠ 0) AND (EX/MEM.Rd = ID/EX.Rs2)	Forward ALU result from EX/MEM to ALU input B
Else if (MEM/WB.RegWrite = 1) AND (MEM/WB.Rd ≠ 0) AND (MEM/WB.Rd = ID/EX.Rs2)	Forward Writeback data from MEM/WB to ALU input B
Else	Use value from Register File

Hazard Detection and Stall/Flush Logic

The Hazard Detection Unit identifies **Load-Use hazards** and **Control hazards (branches, jumps)**.

Stall Logic (Load-Use Hazard)

Condition	Action
If (ID/EX.MemRead = 1) AND ((ID/EX.Rd = IF/ID.Rs1) OR (ID/EX.Rd = IF/ID.Rs2))	Stall the pipeline: - Hold PC and IF/ID register (don’t update) - Insert bubble (NOP) into ID/EX

Flush Logic (Control Hazard)

Condition	Action
If (branch is taken in EX) OR (JAL executed in EX)	Flush IF/ID pipeline register (invalidate instruction) Insert NOP into pipeline

Instruction Memory (IMEM)

- **Size:** 4 KB (example, can scale to 8 KB or 16 KB if program is larger).
- **Width:** 32-bit (one RISC-V instruction per address).
- **Access:**
    - Read-only during execution.
    - Indexed by Program Counter (PC).
    - PC increments by +4 (word-aligned).
- **IF Stage Interface:**
    - Input: PC (address)
    - Output: Instruction (32-bit word)

Data Memory (DMEM)

- **Size:** 4 KB (example, scalable to 16 KB).
- **Width:** 32-bit.
- **Access:** Read/Write.
- **Alignment:** Word-aligned (can later expand for byte/halfword if needed).
- **MEM Stage Interface:**
    - Inputs: Address, WriteData, MemWrite, MemRead
    - Output: ReadData

I/O Definition (Top-Level Module Ports)

- **Inputs:**
    - clk : System clock
    - reset : Active-high reset
    - instr_mem_data (from Instruction Memory)
    - data_mem_data (from Data Memory)
- **Outputs:**
    - instr_mem_addr : Program Counter (to IMEM)

- data_mem_addr : Data Memory address
- data_mem_write : Write enable
- data_mem_wdata : Data to be written

- **Optional status/debug outputs:**
  - halt : Processor finished execution
  - error : Illegal instruction or misalignment

1.3.d Verification Hooks (For Simulation/Debugging)

- **Expose internal signals** for waveform observation:
  - Current PC value (from IF stage)
  - Instruction register (IF/ID pipeline register)
  - Register File read/write ports
  - ALU result (EX stage output)
  - Memory access signals (MEM stage)
  - Writeback data and destination register (WB stage)
- These will help in writing testbenches and debugging pipeline hazards.