32-bit RISC-V Processor

Design Specification

Author

Marco

Development Environment

HDL	Verilog HDL
EDA Tool	Vivado 2025.1
ISA	RV32I + RV32M
Target	Xilinx ZCU104

Version 1.0 • February 2026

Contents

1 Abstract & Introduction **3**

2 RV32I & RV32M **4**
- 2.1 The RV32I & RV32M Instruction Table supported by this processor . . . 5

3 RISC-V Core **7**
- 3.1 Processor Core . . . 8
  - 3.1.1 Datapath Descriptions for All Instructions . . . 8
- 3.2 Program Counter (PC) . . . 14
- 3.3 Register File / General-Purpose Register (GPR) . . . 14
- 3.4 Immediate Generator (ImmGen) . . . 15
- 3.5 Processor Controller . . . 15
- 3.6 Arithmetic Logic Unit (ALU) . . . 16
- 3.7 ALU Controller . . . 16
- 3.8 Branch Processing Unit (BPU) . . . 16
- 3.9 Load Data Unit (LDU) . . . 17
- 3.10 Data Memory . . . 17
- 3.11 Five-Stage Pipelined System . . . 17
- 3.12 Forwarding, Hazard and Flush Detection Unit . . . 18
  - 3.12.1 Forwarding Detection . . . 18
  - 3.12.2 Hazard Detection . . . 19
  - 3.12.3 Flush Detection . . . 19
- 3.13 Branch History Table (BHT) & Branch Tag Buffer (BTB) . . . 20
  - 3.13.1 BHT (Branch History Table) . . . 20
  - 3.13.2 BTB (Branch Target Buffer) . . . 21
- 3.14 Control State Register (CSR) . . . 22
  - 3.14.1 Supported CSR Registers . . . 22
- 3.15 Verification . . . 23

4 AXI-4 Lite Bus, BRAM and Cache **24**
- 4.1 AXI-4 Lite Interface with BRAM . . . 25
  - 4.1.1 Required signals on an AXI4-Lite interface . . . 25
  - 4.1.2 AXI4-Lite interface to BRAM . . . 26
- 4.2 Two-way set-associative Cache with AXI-4 Lite to BRAM . . . 27
  - 4.2.1 Instruction Cache (Only Read Cache) . . . 27
  - 4.2.2 Data Cache . . . 28

5 RISC-V Processor **29**
5.1 Full System Integration 30
5.2 System-Level Verification 31
5.3 Synthesis Results 32

Chapter 1
Abstract & Introduction

This project is based on Computer Organization Course and UC Berkeley CS 61C. The goal was to design a 32-bit pipelined RISC-V CPU supporting RV32I and RV32M, including Forwarding, Hazard Detection, Flush Detection, Dynamic Branch Prediction, and two Caches with AXI4-Lite Bus to BRAM. The final design was implemented and verified on FPGA.

Our development process follows a modular approach, beginning with the core execution unit and progressively expanding into a complete system-on-chip (SoC) architecture. The project is structured into four sequential phases :

1. **Core Processor Design:** Implement a 5-stage pipelined RISC-V CPU supporting both the RV32I (Base Integer) and RV32M (Integer Multiplication and Division) instruction sets, including Forwarding, Hazard Detection, Flush Detection, and Dynamic Branch Prediction.

2. **Bus Infrastructure:** Design an AXI4-Lite bus interface compliant with the ARM AXI protocol, followed by comprehensive system-level testing by interfacing the bus with BRAM (Block RAM).

3. **Memory Hierarchy:** Develop independent Instruction and Data Caches, and integrate them with the AXI4-Lite Bus and BRAM to verify cache coherency and proper timing behaviour.

4. **Full System Integration:** Integrate both caches into the pipelined RISC-V CPU, forming a complete processor connected to BRAM via the AXI4-Lite interconnect, and deploy the final design onto an FPGA.

The complete source code, RTL design files, testbenches, and verification scripts for this project are publicly available on GitHub:

`https://github.com/akira2963753/Pipelined-RV32-SoC`

Chapter 2
RV32I & RV32M

2.1 The RV32I & RV32M Instruction Table supported by this processor

Table 2.1: R-Type Instructions

R-Type	Description	Funct7	Rs2	Rs1	Funct3	Rd	Opcode
ADD	Rd = Rs1 + Rs2	0000000	Rs2	Rs1	000	Rd	0110011
SUB	Rd = Rs1 - Rs2	0100000	Rs2	Rs1	000	Rd	0110011
SLL	Rd = Rs1 << Rs2	0000000	Rs2	Rs1	001	Rd	0110011
SLT	Rd = (Rs1 < Rs2)	0000000	Rs2	Rs1	010	Rd	0110011
SLTU	Rd = (Rs1 < Rs2)	0000000	Rs2	Rs1	011	Rd	0110011
XOR	Rd = Rs1 ⊕ Rs2	0000000	Rs2	Rs1	100	Rd	0110011
SRL	Rd = Rs1 >> Rs2	0000000	Rs2	Rs1	101	Rd	0110011
SRA	Rd = Rs1 >>> Rs2	0100000	Rs2	Rs1	101	Rd	0110011
OR	Rd = Rs1 | Rs2	0000000	Rs2	Rs1	110	Rd	0110011
AND	Rd = Rs1 & Rs2	0000000	Rs2	Rs1	111	Rd	0110011
MUL	Rd = (Rs1 × Rs2)[31:0]	0000001	Rs2	Rs1	000	Rd	0110011
MULH	Rd = (Rs1 × Rs2)[63:32]	0000001	Rs2	Rs1	001	Rd	0110011
MULHSU	Rd = (Rs1 × Rs2[U])[63:32]	0000001	Rs2	Rs1	010	Rd	0110011
MULHU	Rd = (Rs1[U] × Rs2[U])[63:32]	0000001	Rs2	Rs1	011	Rd	0110011
DIV	Rd = Rs1 / Rs2	0000001	Rs2	Rs1	100	Rd	0110011
DIVU	Rd = Rs1[U] / Rs2[U]	0000001	Rs2	Rs1	101	Rd	0110011
REM	Rd = Rs1 % Rs2	0000001	Rs2	Rs1	110	Rd	0110011
REMU	Rd = Rs1[U] % Rs2[U]	0000001	Rs2	Rs1	111	Rd	0110011

Table 2.2: I-Type Instructions

I-Type	Description	Imm[11:0]	-	Rs1	Funct3	Rd	Opcode
ADDI	Rd = Rs1 + Imm	Imm[11:0]	-	Rs1	000	Rd	0010011
SUBI	Rd = Rs1 - Imm	Imm[11:0]	-	Rs1	001	Rd	0010011
SLTI	Rd = (Rs1 < Imm)	Imm[11:0]	-	Rs1	010	Rd	0010011
SLTIU	Rd = (Rs1 < Imm)	Imm[11:0]	-	Rs1	011	Rd	0010011
XORI	Rd = Rs1 ⊕ Imm	Imm[11:0]	-	Rs1	100	Rd	0010011
ORI	Rd = Rs1 | Imm	Imm[11:0]	-	Rs1	110	Rd	0010011
ANDI	Rd = Rs1 & Imm	Imm[11:0]	-	Rs1	111	Rd	0010011
JALR	Rd = PC + 4, PC = Rs1 + Imm & (-1)	Imm[11:0]	-	Rs1	000	Rd	1100111
SLLI	Rd = Rs1 << shamt	0000000	Shamt	Rs1	001	Rd	0010011
SRLI	Rd = Rs1 >> shamt	0000000	Shamt	Rs1	101	Rd	0010011
SRAI	Rd = Rs1 >>> shamt	0100000	Shamt	Rs1	101	Rd	0010011
LB	Rd[7:0] = M[Rs1+Imm]	Imm[11:0]	-	Rs1	000	Rd	0000011
LH	Rd[15:0] = M[Rs1+Imm]	Imm[11:0]	-	Rs1	001	Rd	0000011
LW	Rd = M[Rs1+Imm]	Imm[11:0]	-	Rs1	010	Rd	0000011
LBU	Rd[7:0] = M[Rs1+Imm]	Imm[11:0]	-	Rs1	011	Rd	0000011
LHU	Rd[15:0] = M[Rs1+Imm]	Imm[11:0]	-	Rs1	100	Rd	0000011
CSRRW	Rd = CSR[csr]; CSR[csr] = Rs1	CSR[11:0]	-	Rs1	001	Rd	1110011

Continued on next page

Table 2.2 – *Continued from previous page*

I-Type	**Description**	**Imm[11:0]**	**-**	**Rs1**	**Funct3**	**Rd**	**Opcode**
CSRRS	Rd = CSR[csr]; CSR[csr] | = Rs1	CSR[11:0]	-	Rs1	010	Rd	1110011
CSRRC	Rd = CSR[csr]; CSR[csr] &= ~Rs1	CSR[11:0]	-	Rs1	011	Rd	1110011

Table 2.3: S-Type Instructions

S-Type	**Description**	**Imm[11:5]**	**Rs2**	**Rs1**	**Funct3**	**Imm[4:0]**	**Opcode**
SB	M[Rs1 + Imm][7:0] = Rs2[7:0]	Imm[11:5]	Rs2	Rs1	000	Imm[4:0]	0100011
SH	M[Rs1 + Imm][15:0] = Rs2[15:0]	Imm[11:5]	Rs2	Rs1	001	Imm[4:0]	0100011
SW	M[Rs1 + Imm] = Rs2	Imm[11:5]	Rs2	Rs1	010	Imm[4:0]	0100011

Table 2.4: B-Type Instructions

B-Type	**Description**	**Imm[12|10:5]**	**Rs2**	**Rs1**	**Funct3**	**Imm[4:1|11]**	**Opcode**
BEQ	if (Rs1 == Rs2) PC = PC + Imm	Imm[12|10:5]	Rs2	Rs1	000	Imm[4:1|11]	1100011
BNE	if (Rs1 != Rs2) PC = PC + Imm	Imm[12|10:5]	Rs2	Rs1	001	Imm[4:1|11]	1100011
BLT	if (Rs1 < Rs2) PC = PC + Imm	Imm[12|10:5]	Rs2	Rs1	100	Imm[4:1|11]	1100011
BGE	if (Rs1 ≥ Rs2) PC = PC + Imm	Imm[12|10:5]	Rs2	Rs1	101	Imm[4:1|11]	1100011
BLTU	if (Rs1[U] < Rs2[U]) PC = PC + Imm	Imm[12|10:5]	Rs2	Rs1	110	Imm[4:1|11]	1100011
BGEU	if (Rs1[U] ≥ Rs2[U]) PC = PC + Imm	Imm[12|10:5]	Rs2	Rs1	111	Imm[4:1|11]	1100011

Table 2.5: U-Type Instructions

U-Type	**Description**	**Imm[31:12]**	**Rd**	**Opcode**
LUI	Rd = Imm << 12	Imm[31:12]	Rd	0110111
AUIPC	Rd = PC + (Imm << 12)	Imm[31:12]	Rd	0010111

Table 2.6: J-Type Instructions

J-Type	**Description**	**Imm[20|10:1|11|19:12]**	**-**	**-**	**-**	**Rd**	**Opcode**
JAL	Rd = PC + 4, PC = PC + Imm	Imm[20|10:1|11|19:12]	-	-	-	Rd	1101111

Chapter 3
RISC-V Core

3.1 Processor Core

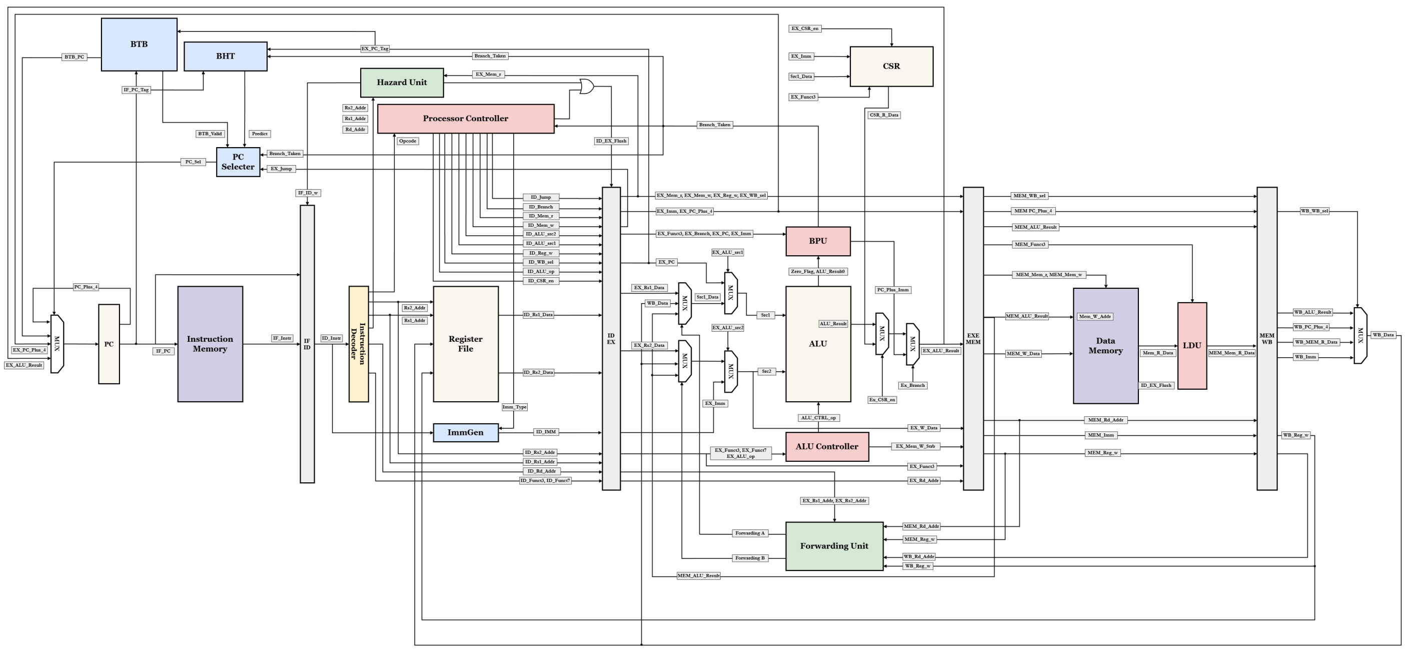


Figure 3.1: RISC-V Core System Framework Diagram Supporting RV32I and RV32M

3.1.1 Datapath Descriptions for All Instructions

Due to the large size of the datapath table, it will be presented across multiple pages.

Table 3.1: Instruction Operation Datapath Description

Instruction	Operation Datapath Description
ADD Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully add the two data values. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SUB Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully subtract the two data values. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SLL Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 << Rs2. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.

Continued on next page

Table 3.1 – *Continued from previous page*

Instruction	**Operation Datapath Description**
SLT Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to determine whether Rs1 is less than Rs2. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SLTU Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to determine whether Rs1 [U] is less than Rs2 [U]. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
XOR Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 ⊕ Rs2. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SRL Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 >> Rs2. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SRA Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 >>> Rs2. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
OR Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 | Rs2. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
AND Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 & Rs2. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
MUL Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 × Rs2 and output the lower 32 bits. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
MULH Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 × Rs2 and output the upper 32 bits. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
MULHSU Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 × Rs2 [U] and output the upper 32 bits. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.

Continued on next page

Table 3.1 – *Continued from previous page*

Instruction	**Operation Datapath Description**
MULHU Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 [U] × Rs2 [U] and output the upper 32 bits. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
DIV Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 / Rs2. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
DIVU Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 [U] / Rs2 [U]. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
REM Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 % Rs2. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
REMU Rd, Rs1, Rs2	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 and EX_ALU_src2 = 0, allowing the ALU to successfully perform Rs1 [U] % Rs2 [U]. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
ADDI Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 + Imm. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SUBI Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 - Imm. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SLTI Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to determine whether Rs1 is less than Imm. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SLTIU Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to determine whether Rs1 [U] is less than Imm [U] (though Imm is still sign-extended first). Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
XORI Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 ⊕ Imm. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.

Continued on next page

Table 3.1 – *Continued from previous page*

Instruction	**Operation Datapath Description**
ORI Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 | Imm. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
ANDI Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 & Imm. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
JALR Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 + Imm & (-1). Then, with EX_Jump = 1, PC_sel = 3, setting PC = EX_ALU_Result. In the WB stage, WB_WB_sel = 1, resulting in Rd = PC + 4.
SLLI Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 << Imm. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SRLI Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 >> Imm. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SRAI Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 >>> Imm. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
SB Rs1, Rs2, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 + Imm to calculate the memory write address. Then, W_STRB controls the byte write operation, writing Rs2[7:0].
SH Rs1, Rs2, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 + Imm to calculate the memory write address. Then, W_STRB controls the half-word write operation, writing Rs2[15:0].
SW Rs1, Rs2, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 + Imm to calculate the memory write address. Then, W_STRB controls the word write operation, writing Rs2[31:0].
LB Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 + Imm to calculate the memory read address. Then, the LDU controls and adjusts the output data, storing it into Rs2[31:0].

Continued on next page

Table 3.1 – *Continued from previous page*

Instruction	**Operation Datapath Description**
LH Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 + Imm to calculate the memory read address. Then, the LDU controls and adjusts the output data, storing it into Rs2[31:0].
LW Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 + Imm to calculate the memory read address. Then, the LDU controls and adjusts the output data, storing it into Rs2[31:0].
LBU Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 + Imm to calculate the memory read address. Then, the LDU controls and adjusts the output data, storing it into Rs2[31:0].
LHU Rd, Rs1, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 + Imm to calculate the memory read address. Then, the LDU controls and adjusts the output data, storing it into Rs2[31:0].
BEQ Rs1, Rs2, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 1, EX_ALU_src2 = 1, allowing the ALU to successfully perform PC + Imm, which is also sent to the BPU to determine whether the branch is taken. If the branch is taken, it checks whether a prediction was made earlier. If the previous prediction was branch taken and the predicted address is correct, no jump is needed. Otherwise, PC_sel = 3, selecting EX_ALU_Result.
BNE Rs1, Rs2, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 1, EX_ALU_src2 = 1, allowing the ALU to successfully perform PC + Imm, which is also sent to the BPU to determine whether the branch is taken. If the branch is taken, it checks whether a prediction was made earlier. If the previous prediction was branch taken and the predicted address is correct, no jump is needed. Otherwise, PC_sel = 3, selecting EX_ALU_Result.
BLT Rs1, Rs2, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 1, EX_ALU_src2 = 1, allowing the ALU to successfully perform PC + Imm, which is also sent to the BPU to determine whether the branch is taken. If the branch is taken, it checks whether a prediction was made earlier. If the previous prediction was branch taken and the predicted address is correct, no jump is needed. Otherwise, PC_sel = 3, selecting EX_ALU_Result.

Continued on next page

Table 3.1 – *Continued from previous page*

Instruction	**Operation Datapath Description**
BGE Rs1, Rs2, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 1, EX_ALU_src2 = 1, allowing the ALU to successfully perform PC + Imm, which is also sent to the BPU to determine whether the branch is taken. If the branch is taken, it checks whether a prediction was made earlier. If the previous prediction was branch taken and the predicted address is correct, no jump is needed. Otherwise, PC_sel = 3, selecting EX_ALU_Result.
BLTU Rs1, Rs2, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 1, EX_ALU_src2 = 1, allowing the ALU to successfully perform PC + Imm, which is also sent to the BPU to determine whether the branch is taken. If the branch is taken, it checks whether a prediction was made earlier. If the previous prediction was branch taken and the predicted address is correct, no jump is needed. Otherwise, PC_sel = 3, selecting EX_ALU_Result.
BGEU Rs1, Rs2, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 1, EX_ALU_src2 = 1, allowing the ALU to successfully perform PC + Imm, which is also sent to the BPU to determine whether the branch is taken. If the branch is taken, it checks whether a prediction was made earlier. If the previous prediction was branch taken and the predicted address is correct, no jump is needed. Otherwise, PC_sel = 3, selecting EX_ALU_Result.
LUI Rd, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 0, EX_ALU_src2 = 1, allowing the ALU to successfully perform Rs1 = Imm << 12. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
AUIPC Rd, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 1, EX_ALU_src2 = 1, allowing the ALU to successfully perform PC + Imm << 12. Then in the WB stage, with WB_WB_sel = 0, WB_ALU_Result is written back.
JAL Rd, Imm	After decoding in the ID stage, Forwarding A & B = 0, selecting EX_Rd_Data as the data for Rs1 and Rs2. EX_ALU_src1 = 1, EX_ALU_src2 = 1, allowing the ALU to successfully perform PC + Imm. Then, with PC_sel = 3, PC = PC + Imm. In the WB stage, WB_WB_sel = 1, resulting in Rd = PC + 4.

3.2 Program Counter (PC)

The PC Module is responsible for managing the program counter updates and control, tracking the address of the instruction currently being executed by the CPU.

Table 3.2: PC Module I/O Ports

I/O	Width	Port
Input	1	CLK
Input	1	RST_N
Input	2	PC_sel
Input	‘PC_WIDTH	EX_ALU_Result
Input	‘PC_WIDTH	IF_PC_Plus_4
Input	‘PC_WIDTH	BTB_PC
Input	‘PC_WIDTH	EX_PC_Plus_4
Output	‘PC_WIDTH	IF_PC

3.3 Register File / General-Purpose Register (GPR)

The Register File provides read and write functionality for 32 general-purpose registers (x0-x31), serving as data storage during CPU operations. The Register File features two read ports with direct read access, and one write port that writes on the rising edge of the clock. Additionally, the x0 register is hard-wired to zero.

Table 3.3: Register File I/O Ports

I/O	Width	Port
Input	1	CLK
Input	1	RST_N
Input	1	WB_Reg_w
Input	‘ADDR_WIDTH	ID_Rs1_Addr
Input	‘ADDR_WIDTH	ID_Rs2_Addr
Input	‘ADDR_WIDTH	WB_Rd_Addr
Input	‘DATA_WIDTH	WB_Rd_Data
Output	‘DATA_WIDTH	ID_Rs1_Data
Output	‘DATA_WIDTH	ID_Rs2_Data

3.4 Immediate Generator (ImmGen)

The ImmGen module is responsible for extracting the immediate value from instructions and performing sign-extension based on the instruction type, generating a 32-bit immediate value.

Table 3.4: ImmGen I/O Ports

I/O	Width	Port
Input	‘INSTR_WIDTH	ID_Instr
Input	3	Imm_Type
Output	‘DATA_WIDTH	ID_Imm

3.5 Processor Controller

The Processor Controller is the CPU’s control unit, responsible for decoding instructions and generating all control signals to coordinate the operation of each module.

Table 3.5: Processor Controller I/O Ports

I/O	Width	Port
Input	‘OPCODE_WIDTH	Opcode
Input	1	Branch_Taken
Input	1	ID_EX_Jump
Input	1	EX_Predict_Taken
Input	‘PC_WIDTH	ID_PC
Input	‘PC_WIDTH	Branch_PC
Output	3	Imm_Type
Output	2	ALU_op
Output	2	WB_sel
Output	1	Reg_w
Output	1	ALU_src1
Output	1	ALU_src2
Output	1	Mem_w
Output	1	Mem_r
Output	1	Branch
Output	1	Jump
Output	1	CSR_en
Output	1	IF_ID_Flush
Output	1	ID_EX_Flush_1

3.6 Arithmetic Logic Unit (ALU)

The ALU performs all arithmetic and logic operations, serving as the CPU's core computational unit.

Table 3.6: ALU I/O Ports

I/O	Width	Port
Input	‘DATA_WIDTH	Src1
Input	‘DATA_WIDTH	Src2
Input	5	ALU_CTRL_op
Output	‘DATA_WIDTH	ALU_Result
Output	1	Zero_Flag

3.7 ALU Controller

The ALU Controller is responsible for combining the instruction's funct3 and funct7 fields with the Processor Controller's ALU_op signal to generate the actual operation control signals for the ALU.

Table 3.7: ALU Controller I/O Ports

I/O	Width	Port
Input	2	ALU_op
Input	3	Funct3
Input	7	Funct7
Output	4	Mem_W_Strb
Output	5	ALU_CTRL_op

3.8 Branch Processing Unit (BPU)

The BPU is responsible for handling the decision logic for all branch and jump instructions, determining whether to take a branch or jump.

Table 3.8: BPU I/O Ports

I/O	Width	Port
Input	1	ALU_Result[0]
Input	1	Zero_Flag
Input	7	Funct3
Input	1	EX_Branch
Input	‘PC_WIDTH	EX_PC
Input	‘DATA_WIDTH	EX_Imm
Output	‘PC_WIDTH	PC_Plus_Imm
Output	1	Branch_Taken

3.9 Load Data Unit (LDU)

The LDU is responsible for processing data read from memory by Load instructions, performing appropriate data extraction and sign-extension based on different Load instruction types.

Table 3.9: LDU I/O Ports

I/O	Width	Port
Input	3	Funct3
Input	‘DATA_WIDTH	Mem_R_Data
Output	‘DATA_WIDTH	LDU_Result

3.10 Data Memory

The Data Memory module provides byte-addressable read and write access for the MEM stage. It supports byte, half-word, and word granularity through write strobe control, and adopts a write-first policy for simultaneous read/write access to the same address.

Table 3.10: Data Memory I/O Ports

I/O	Width	Port
Input	1	CLK
Input	1	Mem_r
Input	1	Mem_w
Input	‘DATA_MEM_ADDR_WIDTH	Mem_Addr
Input	‘DATA_MEM_WIDTH	Mem_W_Data
Input	4	Mem_W_Strb
Output	‘DATA_MEM_WIDTH	Mem_R_Data

3.11 Five-Stage Pipelined System

This project utilizes a Five-Stage Pipelined System to partition the entire Processor into five stages: IF, ID, EX, MEM, and WB, improving instruction execution efficiency. If you want to see more simple implementation about Five-Stage Pipelined System, you can go to my Github Repo of 2025 NTUST Computer Organization Course. The Repo implement Five-Stage Pipelined MIPS CPU step by step.

3.12 Forwarding, Hazard and Flush Detection Unit

This module is responsible for detecting and resolving various hazard conditions in the pipeline, ensuring correct pipeline execution and maintaining data consistency.

3.12.1 Forwarding Detection

Detects **Data Hazards** in the EX and MEM stages, generating forwarding signals to deliver correct data to the ALU in advance, avoiding unnecessary stalls.

Table 3.11: Forwarding Detection I/O Ports

I/O	Width	Port
Input	‘ADDR_WIDTH	MEM_Rd_Addr
Input	1	MEM_Reg_w
Input	‘ADDR_WIDTH	WB_Rd_Addr
Input	1	WB_Reg_w
Input	‘ADDR_WIDTH	EX_Rs1_Addr
Input	‘ADDR_WIDTH	EX_Rs2_Addr
Output	2	Forward_A
Output	2	Forward_B

```verilog
wire load_EX_MEM = (MEM_Reg_w&&MEM_Rd_Addr!=0);
wire load_MEM_WB = (WB_Reg_w&&WB_Rd_Addr!=0);

always @(*) begin
    if(load_EX_MEM && (MEM_Rd_Addr==EX_Rs1_Addr)) Forward_A = 2'b10; // ALU_Result
    else if(load_MEM_WB && (WB_Rd_Addr==EX_Rs1_Addr)) Forward_A = 2'b01; // WB_DATA
    else Forward_A = 2'b00;

    if(load_EX_MEM && (MEM_Rd_Addr==EX_Rs2_Addr)) Forward_B = 2'b10; // ALU_Result
    else if(load_MEM_WB && (WB_Rd_Addr==EX_Rs2_Addr)) Forward_B = 2'b01; // WB_DATA
    else Forward_B = 2'b00;
end
```

Listing 3.1: Forwarding Detection Logic

3.12.2 Hazard Detection

Detects **Load-Use Hazards**. When a Load instruction is immediately followed by an instruction that uses the loaded data, it generates a stall signal to pause the pipeline for one cycle.

Table 3.12: Hazard Detection I/O Ports

I/O	Width	Port
Input	‘ADDR_WIDTH	Rs1_Addr
Input	‘ADDR_WIDTH	Rs2_Addr
Input	‘ADDR_WIDTH	Rd_Addr
Input	1	EX_Mem_r
Output	1	IF_ID_w
Output	1	ID_EX_Flush_0

```
always @(*) begin
    if(EX_Mem_r&&((RdAddr==Rs1Addr)||(RdAddr==Rs2Addr))) begin
        IF_ID_w = 1'b0;
        ID_EX_Flush_0 = 1'b1;
    end
    else begin
        IF_ID_w = 1'b1;
        ID_EX_Flush_0 = 1'b0;
    end
end
```

Listing 3.2: Hazard Detection Logic

3.12.3 Flush Detection

Detects branch/jump and control flow changes (**Control Hazards**), generating flush signals to convert invalid instructions in the IF and ID stages into bubbles (NOPs).

```
always @(*) begin
    if(ID_PC == Branch_PC && (Branch_Taken||ID_EX_Jump)) IF_ID_Flush = 0;
    else begin
        if(Branch_Taken||ID_EX_Jump) IF_ID_Flush = 1;
        else if(EX_Predict_Taken) IF_ID_Flush = 1;
        else IF_ID_Flush = 0;
    end
end

assign ID_EX_Flush_1 = (ID_PC == Branch_PC && (Branch_Taken || ID_EX_Jump))? 0 :
                       ((Branch_Taken || ID_EX_Jump) || (EX_Predict_Taken&&~(
                          Branch_Taken||ID_EX_Jump))) ;
```

Listing 3.3: Flush Detection Logic

3.13 Branch History Table (BHT) & Branch Tag Buffer (BTB)

The BHT and BTB together implement a dynamic branch prediction mechanism. By recording historical branch behavior to predict branch outcomes, they reduce pipeline stalls caused by control hazards.

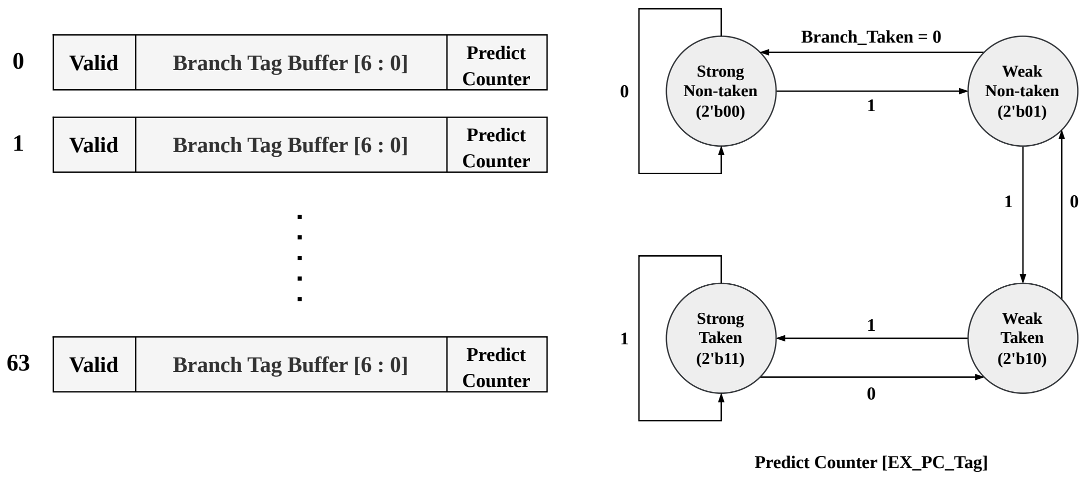


Figure 3.2: BHT & BTB Dynamic Branch Prediction Architecture

3.13.1 BHT (Branch History Table)

Uses 2-bit saturating predict counters to record the historical behavior of branch instructions and predict whether branches will be taken.

- Employs four states: Strong Non-Taken, Weak Non-Taken, Weak Taken, Strong Taken
- The prediction counter is updated at the **EX stage**, once the actual branch outcome (`Branch_Taken`) is resolved, ensuring the state machine reflects the most recent branch behaviour
- Uses lower bits of the PC as an inde (`PC Tag`) to access the prediction table

Table 3.13: BHT I/O Ports

I/O	Width	Port
Input	1	CLK
Input	1	RST_N
Input	‘BHT_PC_WIDTH	PC_Tag
Input	1	Branch_Taken
Input	‘BHT_PC_WIDTH	EX_PC_Tag
Output	1	Predict

3.13.2 BTB (Branch Target Buffer)

Stores the target addresses of branch instructions, providing the jump target when a branch is predicted taken.

- Records the PC tag of branch instructions and their corresponding target addresses
- Supports fast lookup to avoid waiting for branch target calculation
- Works in conjunction with BHT prediction results to update the PC in the IF stage

Table 3.14: BTB I/O Ports

I/O	Width	Port
Input	1	CLK
Input	1	RST_N
Input	‘BHT_PC_WIDTH	PC_Tag
Input	1	Branch_Taken
Input	‘BHT_PC_WIDTH	EX_PC_Tag
Input	‘PC_WIDTH	Branch_PC
Output	‘PC_WIDTH	BTB_PC
Output	1	BTB_Valid

3.14 Control State Register (CSR)

The CSR module implements RISC-V Control and Status Registers, providing system-level state management, exception handling, and privilege mode control functionality.

Table 3.15: CSR I/O Ports

I/O	Width	Port
Input	1	CLK
Input	1	RST_N
Input	1	CSR_en
Input	12	CSR_Addr
Input	‘DATA_WIDTH	CSR_W_Data
Input	3	Funct3
Output	‘DATA_WIDTH	CSR_R_Data

3.14.1 Supported CSR Registers

Table 3.16: Supported CSR Registers

Name	Address	Description	Access
mstatus	0x300	Machine Status Register	CSRRW / CSRRS / CSRRC
mtvec	0x305	Machine Trap Vector Base Address	CSRRW / CSRRS / CSRRC
mepc	0x341	Machine Exception Program Counter	CSRRW / CSRRS / CSRRC
mcause	0x342	Machine Exception Cause	CSRRW / CSRRS / CSRRC
rdcycle	0xC00	Cycle Counter	Read-only

3.15 Verification

The verification approach for this project does not use the Spike RISC-V ISA Simulator. Instead, a faster solution is adopted. Using Claude Code, a RISC-V CPU behavioral model Golden_Result.py is rapidly built in Python to simulate instruction execution and generate expected outputs for the Register File and Data Memory as the Golden Result.

The Golden Result is then automatically compared with RF.out and DM.out generated by the Verilog Testbench to verify the correctness of the RTL design.

This project includes 12 TestCases designed to verify that all RV32I and RV32M instructions are correctly implemented. These test cases cover different types of operations, including arithmetic and logic operations, memory access, branches and jumps, as well as multiply-divide extension instructions, ensuring the CPU executes correctly under various instruction combinations and boundary conditions.

Each TestCase is written in assembly language and then converted to machine code via Instr_Transfer.py to be written into IM.dat. Users can complete the workflow of selecting a Test Case → conversion → result verification through the automated Verify_Script.py script.

Chapter 4
AXI-4 Lite Bus, BRAM and Cache

4.1 AXI-4 Lite Interface with BRAM

AXI (Advanced eXtensible Interface) **[AXI Protocol Specification]** is a high-performance, high-bandwidth, and low-latency on-chip interconnect protocol within the AMBA standard. Its architectural core lies in the decoupling of address/control and data phases, enabling maximum transmission efficiency and optimized bus utilization. In this implementation, an AXI4-Lite bus will be implemented to interface with the BRAM for data storage and retrieval.

4.1.1 Required signals on an AXI4-Lite interface

Table 4.1: Required signals on an AXI4-Lite interface

Name	**Width**	**Description**
AWVALID	1	Asserted high to indicate that the signals on the AW channel are valid.
AWREADY	1	Asserted high to indicate that a transfer on the AW channel can be accepted.
WVALID	1	Asserted high to indicate that the signals on the W channel are valid.
WREADY	1	Asserted high to indicate that a transfer on the W channel can be accepted.
WSTRB	4	Byte strobe signals indicating which byte lanes of WDATA contain valid data.
BVALID	1	Asserted high to indicate that the signals on the B channel are valid.
BREADY	1	Asserted high to indicate that a transfer on the B channel can be accepted.
BRESP	2	Write response status indicating the result of the write transaction.
ARVALID	1	Asserted high to indicate that the signals on the AR channel are valid.
ARREADY	1	Asserted high to indicate that a transfer on the AR channel can be accepted.
RVALID	1	Asserted high to indicate that the signals on the R channel are valid.
RREADY	1	Asserted high to indicate that a transfer on the R channel can be accepted.
RRESP	2	Read response status indicating the result of the read transaction.

Note: BRESP and RRESP are fixed to `2'b00` (OKAY) in this implementation.

4.1.2 AXI4-Lite interface to BRAM

In this implementation, For Single-Port RAM with simultaneous read and write to the same address, a **write-first mode** is adopted and reading from BRAM requires 1 cycle.

Tested.v is the top-level module that connects AXI4_Lite_Bus.v with the Single-Port BRAM IP generated by Vivado. Pattern.v serves as the testbench, testing four different cases:

- **Test Case 1 :** Sequential Write/Read with AW/W separate (AW and W channels arrive in separate cycles)
- **Test Case 2 :** Parallel Write/Read with AW/W together (AW and W channels arrive in same cycle)
- **Test Case 3 :** Byte-Enable Write (W arrives before AW, W_STRB = 4'b0011)
- **Test Case 4 :** Sequential Pattern Write then Read (Write sequential pattern, then read back all data)

4.2 Two-way set-associative Cache with AXI-4 Lite to BRAM

4.2.1 Instruction Cache (Only Read Cache)

In this implementation, the Instruction Memory is designated as **read-only**. Consequently, the Instruction Cache is also read-only and does not support data write-backs or write-throughs. The AXI4-Lite bus interface remains unchanged, as write functionality is disabled by simply masking all write-related signals.

When the CPU requires an instruction, it issues CPU_REQ along with CPU_REQ_ADDR. Once the instruction is ready, the system asserts CPU_REQ_VALID, at which point the CPU retrieves the instruction from the CPU_REQ_DATA bus. The full RTL design and verification for this module is available here.

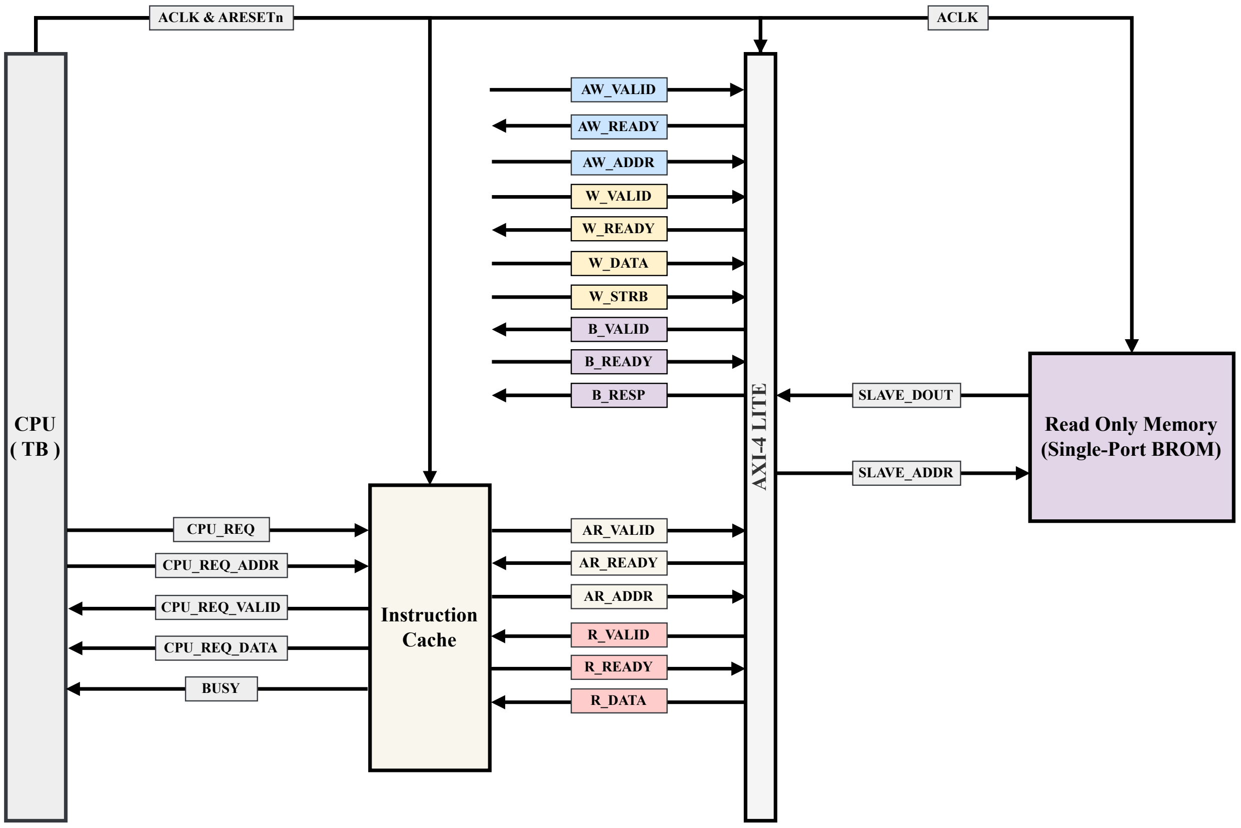


Figure 4.1: Two-way set-associative Cache with AXI4-Lite to BRAM

4.2.2 Data Cache

In this implementation, the Data Cache employs a Write-Through policy instead of Write-Back with a Write Buffer, which presents an opportunity for future optimization. The Data Cache read operation follows the same logic as the Instruction Cache described in Section 4.2.1. It is important to note that the write and read operations share the same ADDR PORT, and the system follows a Write-First approach similar to Memory. The full RTL design and verification for this module is available here.

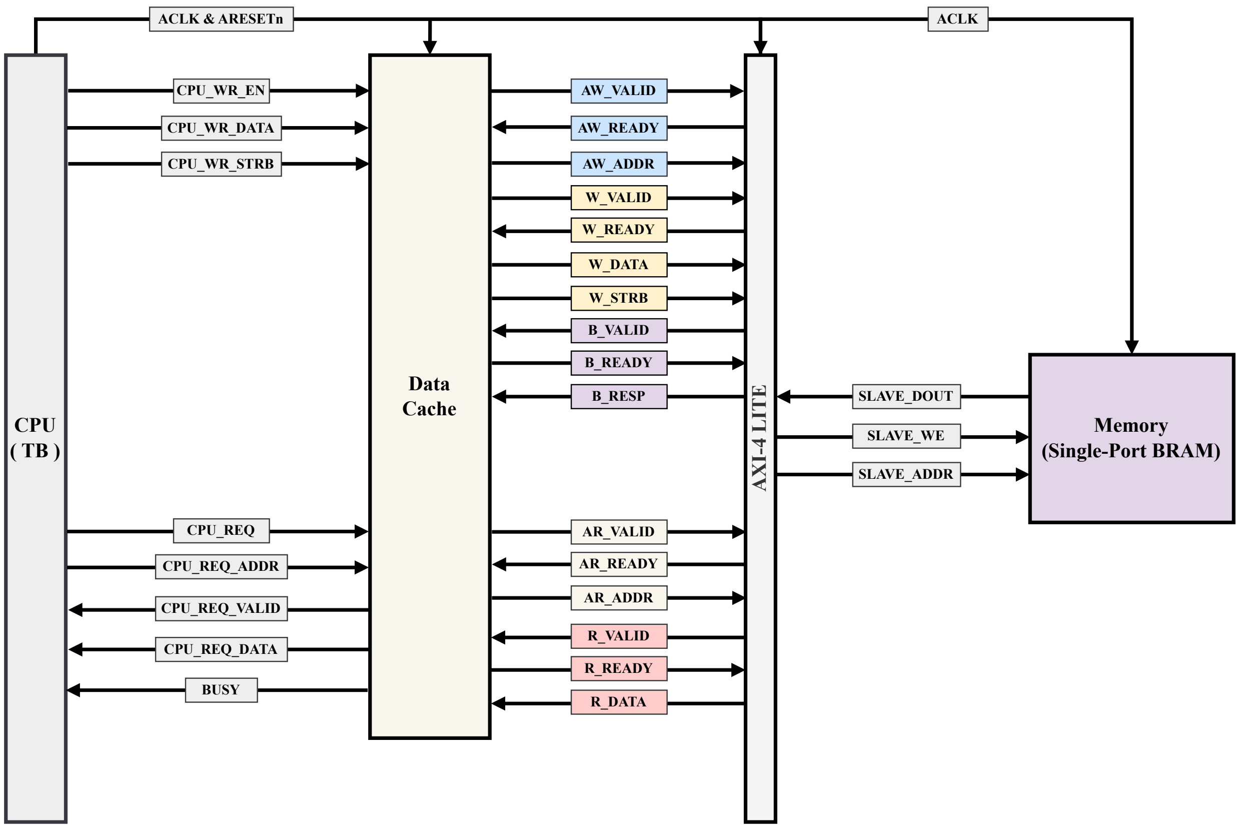


Figure 4.2: Data Cache with AXI4-Lite to BRAM

Chapter 5
RISC-V Processor

5.1 Full System Integration

In previous chapters, we successfully implemented a 5-stage pipelined RISC-V CPU supporting the RV32I and RV32M instruction sets, an AXI4-Lite Bus Interface with BRAM, an I-Cache (AXI4-Lite to BROM), and a D-Cache (AXI4-Lite to BRAM). In this chapter, we will integrate both caches into the CPU to complete the **Final RISC-V Processor**, as you can see the Figure 5.1.

It is worth noting that in this chapter, some of the modules introduced in Chapter 3 will be updated with additional control signals or logic to accommodate the cache integration. However, the overall architecture remains largely unchanged.

The integration of the I-Cache is relatively straightforward : while waiting for a valid instruction read (a cache miss or pending fetch), we simply **disable write-enable for the IF/ID pipeline registers** and **stall the Program Counter (PC)** to prevent it from updating. This effectively freezes the front-end until the instruction is ready.

Integrating the D-Cache is considerably more complex. Because memory read and write operations spend multiple clock cycles, the entire processor must stall until the transaction completes successfully. To achieve this, **stall signals must be asserted across all pipeline stages.**

Furthermore, extra care must be taken to ensure that a **Branch_Taken signal is not triggered during a stall**, which prevents illegal state transitions or incorrect instruction execution. We can determine the completion of a write operation by monitoring the AXI-4 Lite response via **BVALID && BREADY**. For read operations, we track the status through **CPU_REQ_VALID**. Ultimately, whenever the D-Cache is in a **busy state**, the entire pipeline must remain stalled.

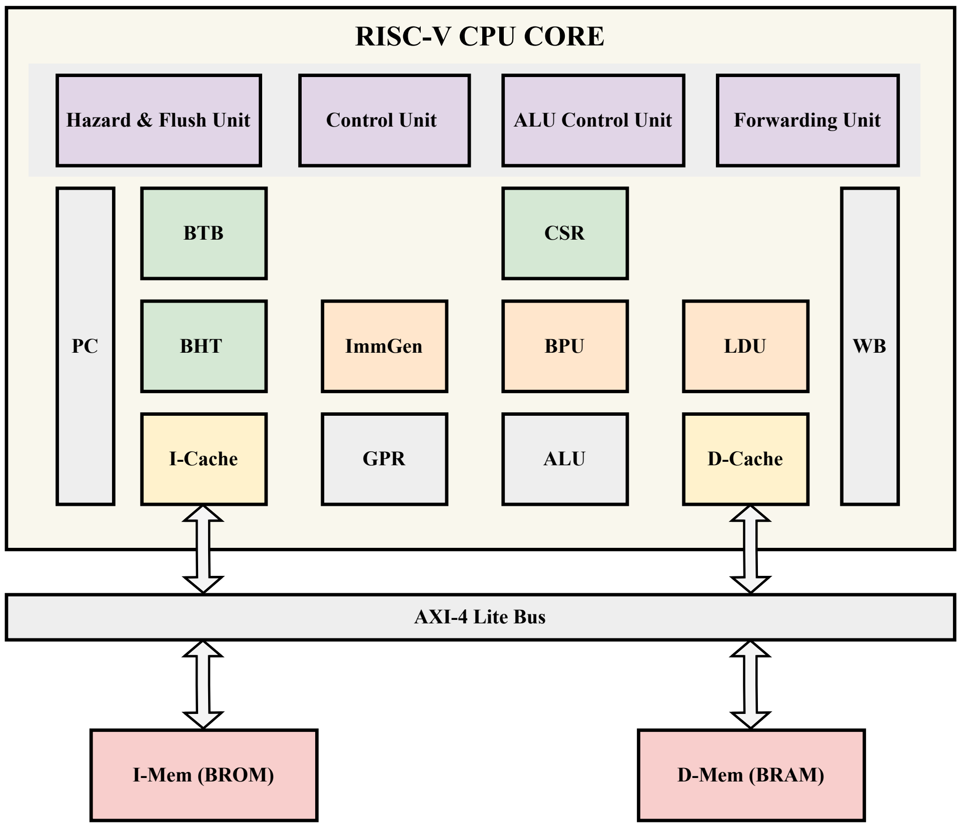


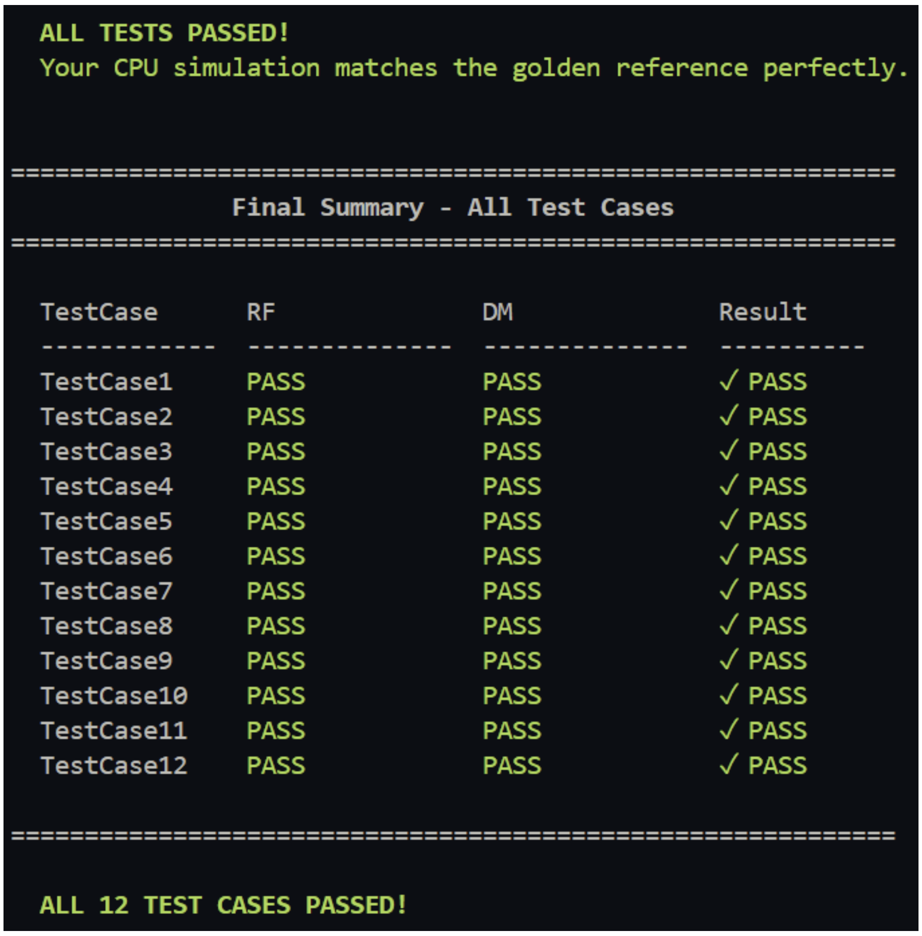
Figure 5.1: RISC-V Processor Overview

5.2 System-Level Verification

The verification process remains consistent with the flow established in Section 3.15, with a few key additions to the automation suite. We have updated Verify_Script.py to include a script that converts IM.dat into a .coe format, specifically for the Instruction Memory (BROM IP).

Furthermore, to facilitate the rapid export of Data Memory content via the TestBench, we opted not to use the Vivado-generated BRAM IP. Instead, we implemented a custom module, D_BRAM.v, which replicates the IP's functionality while allowing for easier file I/O operations and data extraction.

More importantly, to accelerate the verification process, I have written the Vivado simulation TCL commands into a Script.tcl file and integrated it with the Verify_Script.py script to enable automated RTL simulation and verification. Users can choose to run a single Test Case or execute all Test Cases at once.



```
  ALL TESTS PASSED!
  Your CPU simulation matches the golden reference perfectly.


============================================================
               Final Summary - All Test Cases
============================================================

  TestCase      RF              DM              Result
  ------------  --------------  --------------  ----------
  TestCase1     PASS            PASS            ✓ PASS
  TestCase2     PASS            PASS            ✓ PASS
  TestCase3     PASS            PASS            ✓ PASS
  TestCase4     PASS            PASS            ✓ PASS
  TestCase5     PASS            PASS            ✓ PASS
  TestCase6     PASS            PASS            ✓ PASS
  TestCase7     PASS            PASS            ✓ PASS
  TestCase8     PASS            PASS            ✓ PASS
  TestCase9     PASS            PASS            ✓ PASS
  TestCase10    PASS            PASS            ✓ PASS
  TestCase11    PASS            PASS            ✓ PASS
  TestCase12    PASS            PASS            ✓ PASS

============================================================

  ALL 12 TEST CASES PASSED!
```

Figure 5.2: Verification Results

5.3 Synthesis Results

The design was synthesised targeting the Xilinx ZCU104 (UltraScale+) platform at a clock frequency of **250 MHz**. The table below summarises the key post-synthesis resource utilisation, followed by the complete cell-level breakdown reported by Vivado.

Table 5.1: Resource Utilisation Summary

Resource	**Count**
LUT (total)	55,911
Flip-Flop (total)	76,491
DSP48E2	12
BRAM	2
Clock Frequency	**250 MHz**

Table 5.2: Vivado Report Cell Usage

Cell	**Count**	**Cell**	**Count**
Clock / Carry		*LUT*	
BUFG	1	LUT1	225
CARRY8	717	LUT2	664
DSP		LUT3	2,677
DSP_ALU	12	LUT4	1,646
DSP_A_B_DATA	12	LUT5	6,316
DSP_C_DATA	12	LUT6	44,383
DSP_MULTIPLIER	12	*MUX*	
DSP_M_DATA	12	MUXF7	18,036
DSP_OUTPUT	12	MUXF8	8,343
DSP_PREADD	12	*Memory*	
DSP_PREADD_DATA	12	RAMB18E2	1
Flip-Flop		RAMB36E2	1
FDCE	10,942	*I/O Buffer*	
FDPE	5	IBUF	2
FDRE	65,544	OBUF	32