RV32IM PIPELINED PROCESSOR

FINAL REPORT

Group 04
E/20/032 – A.M.N.C. Bandara
E/20/034 – G.M.M.R. Bandara
E/20/157 – S.M.B.G. Janakantha

Department of Computer Engineering
University of Peradeniya
Sri Lanka

PART 1 – Planning

Instruction Types, Encoding Formats, and Their Opcodes

Based on handling the immediate, RISC-V architecture has 6 types of instructions. They are:

1. R – Type (Register Type)
2. I – Type (Immediate Type)
3. S – Type (Store Type)
4. B – Type (Branch Type)
5. U – Type (Upper Immediate)
6. J – Type (Jump Type)

Let's consider the details one by one.

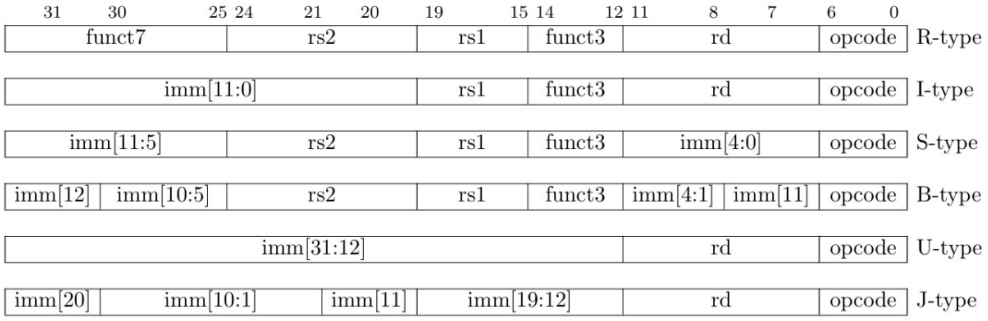


Figure 1: RISC-V base instruction formats showing immediate variants.

1. R – Type (Register Type) Instructions

R-Type (Register Type) instructions perform arithmetic and logical operations on registers. These instructions take three register operands: two source registers (rs1, rs2) and one destination register (rd). The result of the operation is stored in the destination register.

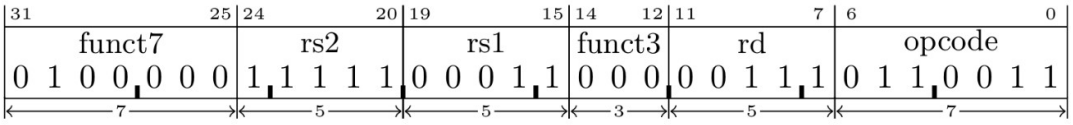


Figure 2: R-Type Instruction Encoding Format

- Bits(6-0): The OP-CODE field identifies the instruction type (for R-Type instructions, this is typically **0110011**). The control logic should use this to interpret the remaining fields and derive the control signals.
- Bits(11-7): The rd field identifies the destination register. After each R-type operation, the result is written into this register.
- Bits(14-12): The funct3 field further specifies the operation (e.g., for distinguishing between **add** and **sub**).
- Bits(19-15): The rs1 field identifies the source register 1.
- Bits(24-20): The rs2 field identifies the source register 2.
- Bits(31-25): The funct7 field specifies the operation (e.g., for distinguishing between **add** and **sub**).

Table 1: OP-CODEs of R-Type Instruction Set

Instruction	Description	Opcode	func3	func7
add	ADD the value in rs1 and rs2 value and put it into rd. Syntax: add rd, rs1, rs2 C code: rd = rs1 + rs2	0110011	000	0000000
sub	SUBSTRACT the value in rs1 by rs2 value and put it into rd. Syntax: sub rd, rs1, rs2 C code: rd = rs1 - rs2	0110011	000	0100000
sll	LOGICALLY LEFT SHIFT the bits of rs1 by rs2 amount and put it into rd. Syntax: sll rd, rs1, rs2 C code: rd = rs1 << rs2	0110011	001	0000000
slt	Sets the rd to 1 if the sign integer value of rs1 is less than the sign integer value rs2; otherwise, it sets it to 0. Syntax: slt rd, rs1, rs2 C code: rd = (rs1<rs2) ? 1 : 0	0110011	010	0000000
sltu	Sets the rd to 1 if the unsigned integer value of rs1 is less than the unsigned integer value rs2; otherwise, it sets it to 0. Syntax: sltu rd, rs1, rs2 C code: rd = (rs1<rs2) ? 1 : 0	0110011	011	0000000
xor	Perform the bitwise XOR operation on the values in rs1 and rs2 and put it into rd. Syntax: xor rd, rs1, rs2 C code: rd = rs1 ^ rs2	0110011	100	0000000
srl	LOGICALLY RIGHT SHIFT the bits of rs1 by rs2 amount and put it into rd. Syntax: srl rd, rs1, rs2 C code: rd = rs1 >> rs2	0110011	101	0000000
sra	ARITHMETICALLY RIGHT SHIFT bits of rs1 by rs2 amount and put it into rd. Syntax: sra rd, rs1, rs2 C code: rd = rs1 >> rs2 (sign-extended)	0110011	101	0100000
Or	Perform the bitwise OR operation on the values in rs1 and rs2 and put it into rd. Syntax: or rd, rs1, rs2 C code: rd = rs1 | rs2	0110011	110	0000000
and	Perform the bitwise AND operation on the values in rs1 and rs2 and put it into rd. Syntax: and rd, rs1, rs2 C code: rd = rs1 & rs2	0110011	111	0000000

mul	Multiplies the values in rs1 and rs2 and stores the lower 32 bits of the result in rd. Syntax: mul rd, rs1, rs2 C code: rd = (rs1*rs2)[31:0]	0110011	000	0000001
mulh	Multiply High Signed, Multiplies the values in rs1 and rs2 as signed integers and stores the upper 32 bits of the result in rd. Syntax: mulh rd, rs1, rs2 C code: rd = (rs1*rs2)[63:32]	0110011	001	0000001
mulhsu	Multiply High Signed-Unsigned, Multiplies rs1 as a signed integer and rs2 as an unsigned integer. The upper 32 bits of the result are stored in rd. Syntax: mulhsu rd, rs1, rs2 C code: rd = (rs1*rs2)[63:32]	0110011	010	0000001
mulhu	Multiply High Unsigned, Multiplies rs1 and rs2 as unsigned integers. The upper 32 bits of the result are stored in rd. Syntax: mulhsu rd, rs1, rs2 C code: rd = (rs1*rs2)[63:32]	0110011	011	0000001
div	Divides rs1 by rs2 as signed integers and stores the quotient in rd. Division by zero results in an undefined value in rd. Syntax: div rd, rs1, rs2 C code: rd = rs1 / rs2	0110011	100	0000001
divu	Divide Unsigned, divides rs1 by rs2 as unsigned integers and stores the quotient in rd. Division by zero results in an undefined value in rd. Syntax: div rd, rs1, rs2 C code: rd = rs1 / rs2	0110011	101	0000001
rem	Remainder Signed, computes the remainder of rs1 divided by rs2 as signed integers and stores the result in rd. Division by zero results in an undefined value in rd. Syntax: rem rd, rs1, rs2 C code: rd = rs1 % rs2	0110011	110	0000001
remu	Remainder Unsigned, computes the remainder of rs1 divided by rs2 as unsigned integers and stores the result in rd. Division by zero results in an undefined value in rd. Syntax: remu rd, rs1, rs2 C code: rd = rs1 % rs2	0110011	111	0000001

2. I – Type (Immediate Type) Instructions

The I-Type (Immediate Type) instruction format is used to encode instruction with a signed 12-bit immediate operand (there are some special cases where this number changes) with a range of –2048 to 2047, a rd register, and a rs1 register.

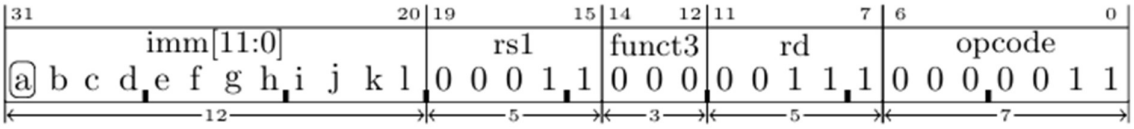


Figure 3: I-Type Instruction Encoding Format

- Bits(6-0): OP-CODE of Instructions. These bits are used to derive control signals by the control logic. For I-type instructions, there are 3 opcodes that specify different types of operations.
    - 0000011 – Load Immediate instructions
    - 0010011 – Arithmetic immediate instructions
    - 1100111 – Jump instruction

- Bits(11-7): The rd field identifies the destination register. Where the results will be saved for all immediate arithmetic and load instructions.

- Bits(14-12): The func3 field. Which specifies the type of operation based on opcode. Which is used to generate control signals in **ALUOP** and how load works

- Bits(19-15): The rs1 , The source register address.

- Bits(31-20): 12 bit immediate value.

- Bits[30]: This bit is specifically used in **srli/srai** instructions to differentiate from one other.

❖ For all the I-type instructions (other than shift) the decoding of instructions as follows,

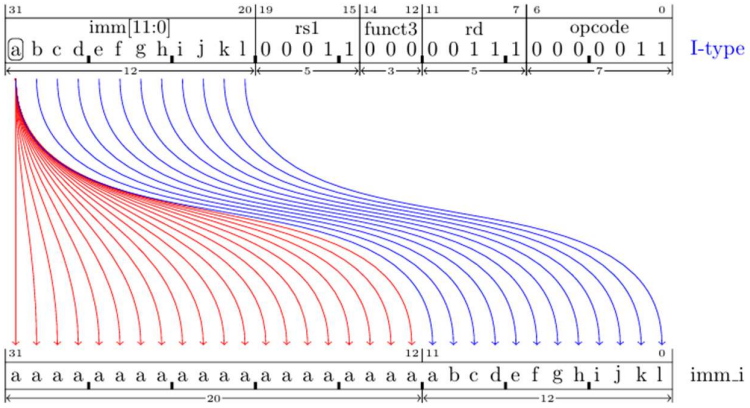


Figure 4: I-Type Instruction Immediate Decoding

❖ For shifting instructions,

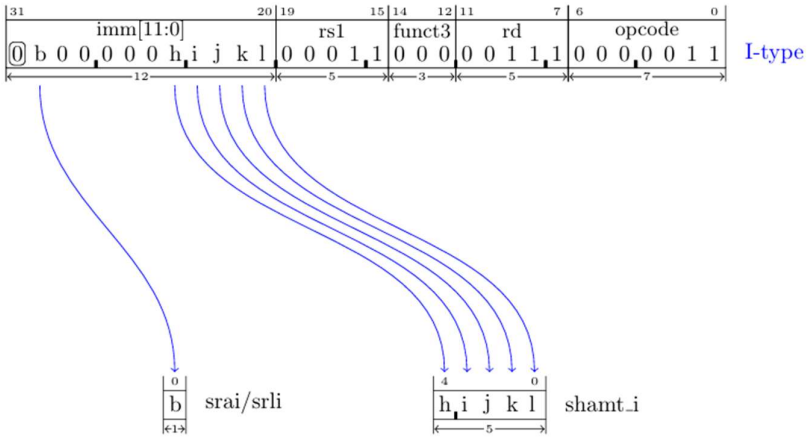


Figure 5: I-Type Shift Instruction Immediate Decoding

Table 2: OP-CODEs of I-Type Instruction Set

Instruction	Description	Opcode	func3	func7
lb	Load Byte, set register rd to the value of the sign-extended byte fetched from the memory address given by the sum of rs1 and imm_i. Syntax: lb rd, imm(rs1) C code: rd = mem[rs1+imm_i]	0000011	000	N/A
lh	Load Halfword, set register rd to the value of the sign-extended 16-bit little-endian half-word value fetched from the memory address given by the sum of rs1 and imm_i. Syntax: lh rd, imm(rs1) C code: rd = mem[rs1+imm_i]	0000011	001	N/A
lw	Load Word, set register rd to the value of the sign-extended 32-bit little-endian word value fetched from the memory address given by the sum of rs1 and imm_i. Syntax: lw rd, imm(rs1) C code: rd = mem[rs1+imm_i]	0000011	010	N/A
lbu	Load Byte Unsigned, set register rd to the value of the zero-extended byte fetched from the memory address given by sum of rs1 and imm_i. Syntax: lbu rd, imm(rs1) C code: rd = zeroExtend(mem[rs1+imm_i])	0000011	100	N/A

lhu	Load Halfword Unsigned, set register rd to the value of the sign-extended 16-bit little-endian half-word value fetched from the memory address given by the sum of rs1 and imm_i. Syntax: lhu rd, imm(rs1) C code: rd = zeroExtend(mem[rs1+imm_i])	0000011	101	N/A
addi	Add the rs1 value and imm_i and store the result in register rd. Syntax: addi rd, rs1, imm C code: rd = rs1 + imm_i	0010011	000	N/A
slli	LOGICALLY LEFT SHIFT the bits of rs1 by shamt_i amount and put it into rd. (0's will be added to the to the shifted bits in *LSB* side) Syntax: slli rd, rs1, shamt C code: rd = rs1 << shamt_i	0010011	001	000000
slti	Sets the rd to 1 if the sign integer value of rs1 is less than the sign integer value imm_i; otherwise, it sets it to 0. Syntax: slti rd, rs1, imm C code: rd = (rs1 < imm_i) ? 1 : 0	0010011	010	N/A
sltiu	Sets the rd to 1 if the unsigned integer value of rs1 is less than the unsigned integer value imm_i; otherwise, it sets it to 0. Syntax: sltui rd, rs1, rs2 C code: rd = (rs1 < imm_i) ? 1 : 0	0010011	011	N/A
xori	Bitwise XOR rs1 and imm_i and store the result in rd. (imm will be sign extended) Syntax: xori rd, rs1, imm C code: rd = rs1 ^ imm_i	0010011	100	N/A
srli	LOGICALLY RIGHT SHIFT the bits of rs1 by shamt_i amount and put it into rd. (0's will be added to the shifted bits in *MSB* side) Syntax: srl rd, rs1, imm C code: rd = rs1 >> shamt_i	0010011	101	000000
srai	ARITHMETICALLY RIGHT SHIFT bits of rs1 by shamt_i amount and put it into rd. Syntax: srai rd, rs1, imm C code: rd = rs1 >> shamt_i (sign-extended)	0010011	101	010000

ori	Bitwise OR rs1 and imm_i and store the result in rd. Syntax: ori rd, rs1, imm C code: rd = rs1 | imm_i	0010011	110	N/A
andi	Bitwise AND rs1 and imm_i and store the result in rd. Syntax: ori rd, rs1, imm C code: rd = rs1 & imm_i	0010011	111	N/A
jalr	Jumps to an address specified by the rs1 and the imm_i value, and saves the return address in the rd. Syntax: jalr rd, imm(rs1) C code: rd = PC + 4 (address of the next instruction) PC = (rs1 + imm_i) & ~1 (target address)	1100111	000	N/A

3. S – Type (Store Type) Instructions

S-Type (Store Type) is used for store operations, which store data from a register to memory. They include two register operands and a 12-bit immediate value for the memory address offset.

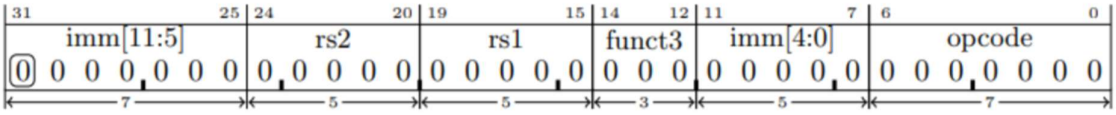


Figure 6: S-Type Instruction Encoding Format

- Bits(6-0): The OP-CODE field identifies the instruction type (for S-Type instructions, this is typically **0100011**). The control logic should use this to interpret the remaining fields and derive the control signals.

- Bits(14-12): The funct3 part is used to encode the type

- Bits(19-15): The rs1 field identifies the source register 1 that is added to the 12-bit immediate field to form the memory address.

- Bits(24-20): The rs2 field identifies the source register 2. That is the source register whose value should be stored into memory.

- Bits(31-25)(11-7): The 12-bit immediate offset is split into two parts:
  - imm[11:5]: using the Bits(31-25) field of the instruction
  - imm[4:0]: using the Bits(11-7) field of the instruction

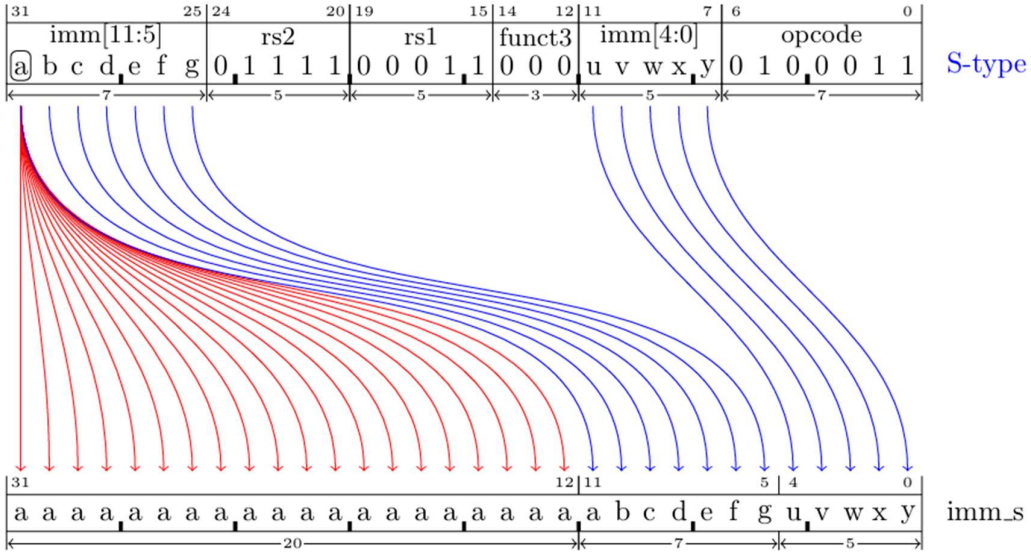


Figure 7: Decoding an S-type Instruction.

Table 3: OP-CODEs of S-Type Instruction Set

Instruction	Description	Opcode	func3	func7
sb	Store byte takes a byte from the least significant 8 bits of the rs2 and writes it to the memory address given by imm_s + rs1 Syntax: sb rs2, imm_s(rs1) C code: mem[rs1+imm_s] = rs2[7:0]	0100011	000	N/A
sh	Store half takes a halfword from the least significant 16 bits of the rs2 and writes it to the memory address given by imm_s + rs1 Syntax: sh rs2, imm_s(rs1) C code: mem[rs1+imm_s] = rs2[15:0]	0100011	001	N/A
sw	Store word takes a word from the least significant 32 bits of the rs2 and stores it to the memory address given by imm_s + rs1 Syntax: sw rs2, imm_s(rs1) C code: mem[rs1+imm_s] = rs2[31:0]	0100011	111	N/A

4. B – Type (Branch Type) Instructions

B-Type (Branch Type) instructions perform conditional branches based on the components of two registers (rs1, rs2) values. Here we compare the values of the rs1 and rs2. If the comparison is true, the program counter (pc) is updated to the target address computed from the relative branch offset. If the comparison is false, the program counter is incremented to the next sequential instruction.

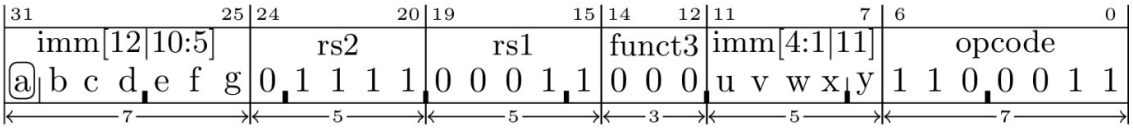


Figure 8: B-Type Instruction Encoding Format

- Bits(6-0): The OP-CODE field identifies the instruction type (for B-Type instructions, this is typically **1100011**). The control logic should use this to interpret the remaining fields and derive the control signals.

- Bits(14-12): The funct3 field further determines the specific branch type.

- Bits(19-15): The rs1 field identifies the source register 1.

- Bits(24-20): The rs2 field identifies the source register 2..

- Bits(31-25)(11-7): The 13-bit imm field split into 4 parts:
  - imm[12]: using the Bits(31) field of the instruction
  - imm[11]: using the Bits(7) field of the instruction
  - imm[10:5]: using the Bits(30-25) field of the instruction
  - imm[4:1]: using the Bits(11-8) field of the instruction

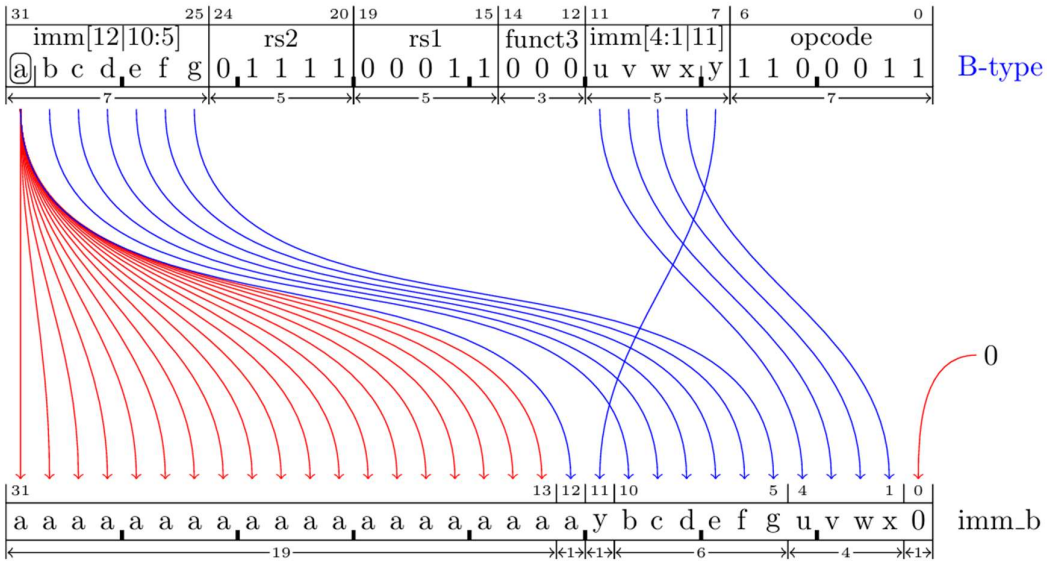


Figure 9: Decoding a B-type Instruction.

Table 4: OP-CODEs of B-Type Instruction Set

Instruction	Description	Opcode	func3	func7
beq	If the values in rs1 and rs2 are equal, add imm_b to the pc register. Otherwise, go to the next instruction. Syntax: beq rs1, rs2, imm_b C code: pc = pc + (rs1==rs2 ? imm_b : 4)	1100011	000	N/A
bne	If the values in rs1 and rs2 are not equal, add imm_b to the pc register. Otherwise, go to the next instruction. Syntax: bne rs1, rs2, imm_b C code: pc = pc + (rs1!=rs2 ? imm_b : 4)	1100011	001	N/A
blt	If the signed value in rs1 is less than the signed value in rs2, add imm_b to the pc register. Otherwise, go to the next instruction. Syntax: blt rs1, rs2, imm_b C code: pc = pc + (rs1<rs2 ? imm_b : 4)	1100011	100	N/A
bge	If the signed value in rs1 is greater than or equal to the signed value in rs2, add imm_b to the pc register. Otherwise, go to the next instruction. Syntax: bge rs1, rs2, imm_b C code: pc = pc + (rs1>=rs2 ? imm_b : 4)	1100011	101	N/A
bltu	If the unsigned value in rs1 is less than the unsigned value in rs2, add imm_b to the pc register. Otherwise, go to the next instruction. Syntax: bltu rs1, rs2, imm_b C code: pc = pc + (rs1<rs2 ? imm_b : 4)	1100011	110	N/A
bequ	If the unsigned value in rs1 is greater than or equal to the unsigned value in rs2, add imm_b to the pc register. Otherwise, go to the next instruction. Syntax: bgeu rs1, rs2, imm_b C code: pc = pc + (rs1>=rs2 ? imm_b : 4)	1100011	111	N/A

5. U – Type (Upper Immediate) Instructions

The U-type (Upper Immediate) format is used for instructions that use a 20-bit immediate operand and destination register (rd). The **rd** field contains a register address that would be set to a value depending on the instruction.

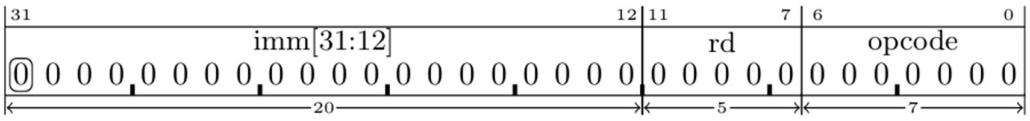


Figure 10: U-Type Instruction Encoding Format

- Bits(6-0): The OP-CODE field of Instructions. These bits are used to derive control signals by the control logic. For J-type instructions, there are 2 variants:
  - 0010111 – auipc instruction
  - 0110111 – lui instruction

- Bits(11-7): The rd field identifies the destination register. Where the relevant value will be stored after operation.

- Bits(31-12): 20-bit immediate value.

- Bits[31]: Used to sign extend the 21-bit immediate value.

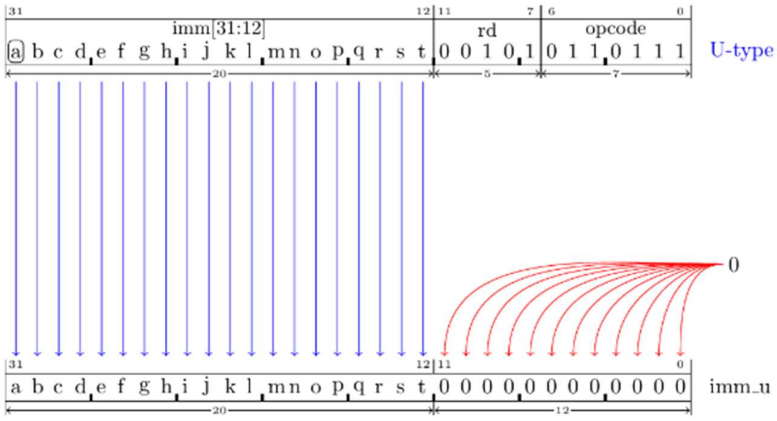


Figure 11: U-Type Instruction Immediate Decoding

Table 5: OP-CODEs of U-Type Instruction Set

Instruction	Description	Opcode	func3	func7
auipc	Add the address of the instruction to the imm_u value and store the result in register rd Syntax: auipc rd, imm_u C code: rd = pc + imm_u	0010111	N/A	N/A
lui	Set the register rd to the imm_u value. Syntax: lui rd, imm C code: rd = imm_u	0110111	N/A	N/A

6. J – Type (Jump Type) Instructions

The J-type (Jump Type) instructions perform unconditional jumps to the target address. It is similar to the U-type, but bits in the immediate operand are arranged in a different order.

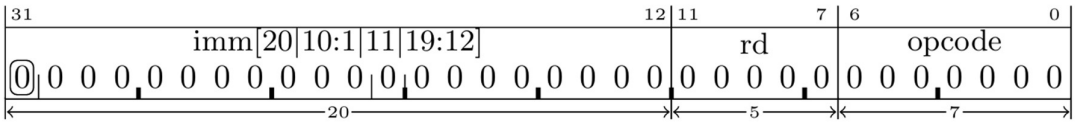


Figure 12: J-Type Instruction Encoding Format

- Bits(6-0): The OP-CODE of Instructions. These bits are used to derive control signals by the control logic. For J-type instructions, it's **1101111**.

- Bits(11-7): The rd field identifies the destination register (link register address). Where the return address (pc + 4) will be stored.

- Bits(31-12): 20-bit immediate value.

- Bits[31]: Used to sign extend the 21-bit immediate value.

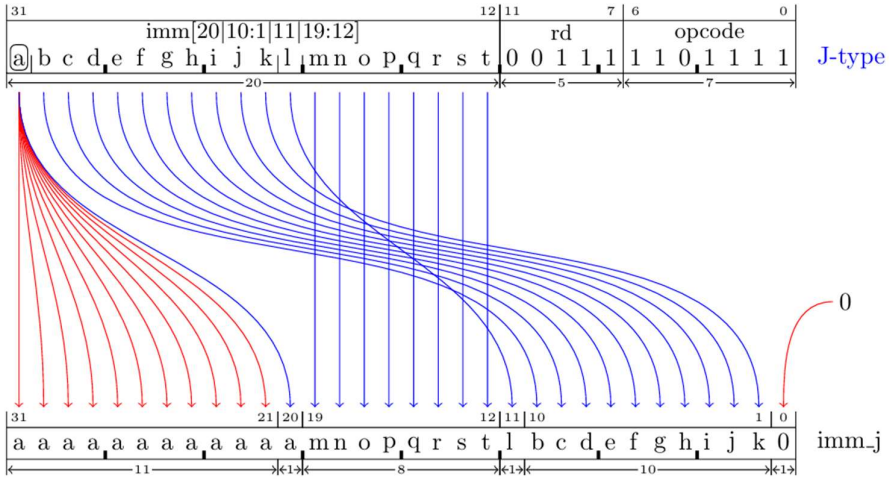


Figure 13: J-Type Instruction Immediate Decoding

Table 6: OP-CODEs of J-Type Instruction Set

Instruction	Description	Opcode	func3	func7
jal	Set rd to the address of the next instruction (address of the jal + 4) and then jump to the address given by the pc + imm_j value. Syntax: jal rd, pcrel21 Description: rd = PC + 4 (address of the next instruction) PC = pc + imm_j (target address)	1101111	N/A	N/A

Planned Data Path

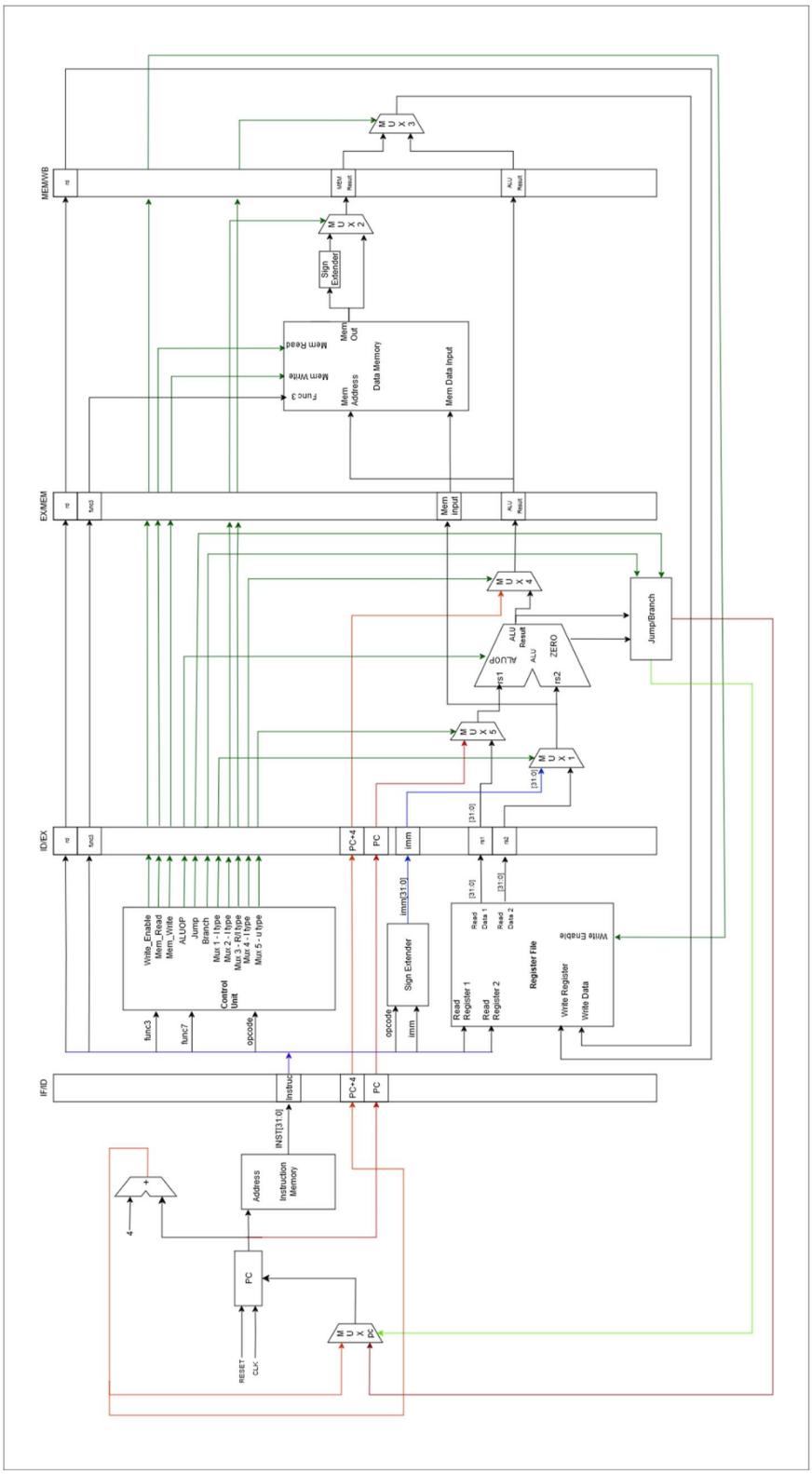


Figure 14: RISC-V 32IM Pipelined Data Path

Source File: datapath.png, draw.io

Testing

Here we hope to test the code with the following cases.

1. Correctness of individual instructions
2. Pipelined behaviour
3. Sequential Instruction Testing
4. Edge Cases

Individual Instruction Testing

Arithmetic Instructions (ADD, SUB, MUL, etc.)

- Input: add x1, x2, x3 *(x2 = 5, x3 = 3).*
- Expected Output: x1 = 8.

Load/Store Instructions (LW, SW)

- Input: lw x1, 0(x2) *(x2 points to memory holding 0x12345678)*.
- Expected Output: x1 = 0x12345678.

Branch Instructions (BEQ, BNE, BLT, etc.)

- Input: beq x1, x2, label *(x1 = x2)*.
- Expected Output: PC jumps to label.

Immediate Instructions (ADDI, ANDI, etc.)

- Input: addi x1, x2, 10 *(x2 = 5)*.
- Expected Output: x1 = 15.

Pipeline Behaviour Testing

Data Hazard Without Forwarding

- Input:
```
add x1, x2, x3    // x1 = x2 + x3
sub x4, x1, x5    // Use x1 immediately
```
- Expected Output: Stall in the pipeline (bubble inserted).

Data Hazard with Forwarding

- Input:
```
add x1, x2, x3    // x1 = x2 + x3
sub x4, x1, x5    // Forward x1 to EX stage
```
- Expected Output: No stalls; result forwarded correctly.

Control Hazard

- Input:
```
beq x1, x2, label      // Branch condition
add x3, x4, x5         // Next instruction
```

- Expected Output: Flush pipeline if branch is taken; PC updated correctly.

Structural Hazard

- Input: Test simultaneous memory access (e.g., instruction fetch and data access).
- Expected Output: No conflict if the pipeline handles it; stall otherwise.

Sequential Instruction Testing

Basic Arithmetic Sequence

- Input:
```
add x1, x0, x0         // x1 = 0
addi x2, x0, 5         // x2 = 5
add x3, x1, x2         // x3 = x1 + x2
sub x4, x3, x2         // x4 = x3 - x2
```

- Expected Output: x3 = 5, x4 = 0.

Load/Store Sequence

- Input:
```
sw x2, 0(x3)           // Store x2 into memory
lw x4, 0(x3)           // Load from memory into x4
```

- Expected Output: x4 = x2.

Branch and Loop

- Input:
```
addi x1, x0, 0                 // Initialize x1 = 0
addi x2, x0, 5                 // Set x2 = 5
Loop:   add x1, x1, x2         // x1 += x2
        subi x3, x3, 1         // Decrement counter
        bne x3, x0, Loop       // Branch if not zero
```

- Expected Output: x1 = 5 * iterations

Edge Case Testing

Branch Target Alignment

- Input: Branch to misaligned address.
- Expected Output: Raise exception or align address.

Memory Alignment

- Input: Load/Store with unaligned address (e.g., lw x1, 3(x2)).
- Expected Output: Correct exception handling.

Division by Zero

- Input: DIV x1, x2, x3 (x3 = 0).
- Expected Output: Raise exception or return specific value.

Instruction Fetch Beyond Memory

- Input: Fetch instruction from invalid memory address.
- Expected Output: Raise exception.

PART 2 – Hardware Units

ALU

The ALU implemented follows the RISC-V 32IM specification, supporting arithmetic, logical, and shift operations. The design is fully combinational computing the result in a single clock cycle without requiring registers.

The ALU has the following inputs and outputs:

Inputs:
- A (32-bit): First operand.
- B (32-bit): Second operand.
- ALU SELECT (4-bit): Selects the operation to perform.

Outputs:
- ALU RESULT (32-bit): The computed result of the operation.

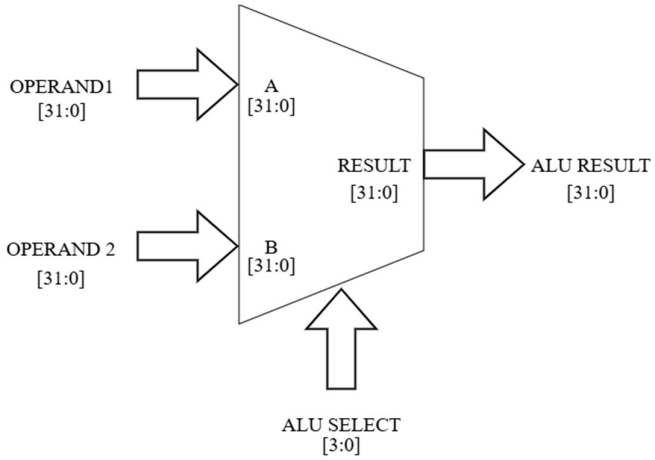


Figure 15: Interfaces of the 32-bit ALU

The Table below shows the functions that the 32-bit ALU is capable of performing.

ALU SELECT	Function	Description
0000	ADD	Signed addition A + B → RESULT
0001	SUB	Signed subtraction A - B → RESULT
0010	AND	Bitwise AND A & B → RESULT
0011	OR	Bitwise OR A | B → RESULT
0100	XOR	Bitwise XOR A ⊕ B → RESULT
0101	SLL	Logical Left Shift A << B → RESULT
0110	SRL	Logical Right Shift A >> B → RESULT
0111	SRA	Arithmetic Shift Right A >>> B → RESULT
1000	SLT	Set if less than (signed) If A<B: 1 → RESULT Else: 0 → RESULT
1001	MUL	Lower 32 bits of signed multiplication (A(signed)*B(signed)) [31:0] → RESULT
1010	MULH	Upper 32 bits of signed multiplication (A(signed)*B(signed)) [63:32] → RESULT
1011	MULHU	Upper 32 bits of unsigned multiplication (A(unsigned)*B(unsigned)) [63:32] → RESULT
1100	FORWARD	Forward OPERAND 2 into RESULT B → RESULT
1101	DIV	Signed division A(signed) / B(signed) → RESULT
1110	DIVU	Unsigned division A(unsigned) / B(unsigned) → RESULT
1111	REM	Reminder of signed division A % B → RESULT

Figure 16: ALU Function

These functional units are implemented as separate modules, and instantiated inside the ALU module. The ALU uses a 16x1 MUX to pick one of the functional unit's outputs and send it to RESULT based on the ALU SELECT value.

Register File

The register file in our implementation is a 32x32-bit register bank containing 32 registers, each 32-bits wide. It is used to store and retrieve data for the processor's execution units. The design supports asynchronous reads and synchronous writes, ensuring efficient operation while preventing unintended modifications.

The register file has the following inputs and outputs:

Inputs:

- CLK (1-bit): Clock signal, used for write operations
- WRITEENABLE (1-bit): Enables writing to a register when high
- DATAW (32-bit): Data to be written to the register
- RSR1 (5-bit): Address of the first register to read
- RSR2 (5-bit): Address of the second register to read
- RSW (5-bit): Address of the register to write to
- RST (1-bit): Reset all register values to 0

Outputs:

- DATAR1 (32-bit): Data read from register RSR1.
- DATAR2 (32-bit): Data read from register RSR2.

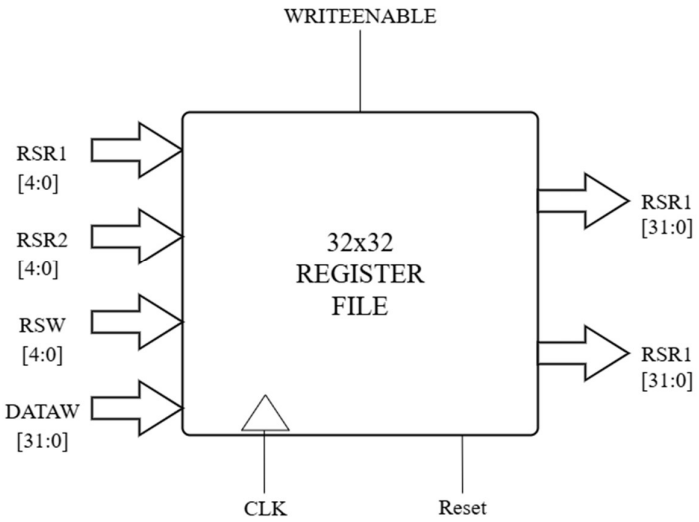


Figure 17: Interfaces of the 32x32 Register File

Register file allows two register values to be fetched in the same cycle. Writes occur only on the rising edge of the clock when WRITEENABLE is high. Register 0 is hardwired to zero which ensures that x0 always holds the constant value 0, as required by RISC-V.

Control Unit

The Control Unit is responsible for decoding instructions and generating the necessary control signals for different processor components. It ensures that each instruction is executed correctly by enabling the appropriate data paths, selecting ALU operations, and managing memory and register accesses.

The control unit has the following inputs and outputs:

Inputs:

- INST (32-bit): The instruction fetched from instruction memory
- BREQ (1-bit): Indicates if two register values are equal
- BRLT (1-bit): Indicates if one register value is less than the other

Outputs:

- REGWEN (1-bit): Enables writing to the register file
- ALUSEL (4-bit): Selects the ALU operation
- BSEL (1-bit): Chooses between a register value or an immediate as the second ALU operand
- WBSEL (2-bit): Determines the source of data for writing back to the register file
- MEMRW (1-bit): Controls memory read (0) or write (1) operations
- IMMSEL (3-bit): Selects the type of immediate value (I, S, B, U, J)
- PCSEL (1-bit): Determines the next program counter source (branch/jump)
- BRUN (1-bit): Determines if the branch comparison should be unsigned
- ASEL (1-bit): Selects the first ALU operand (register value or PC)

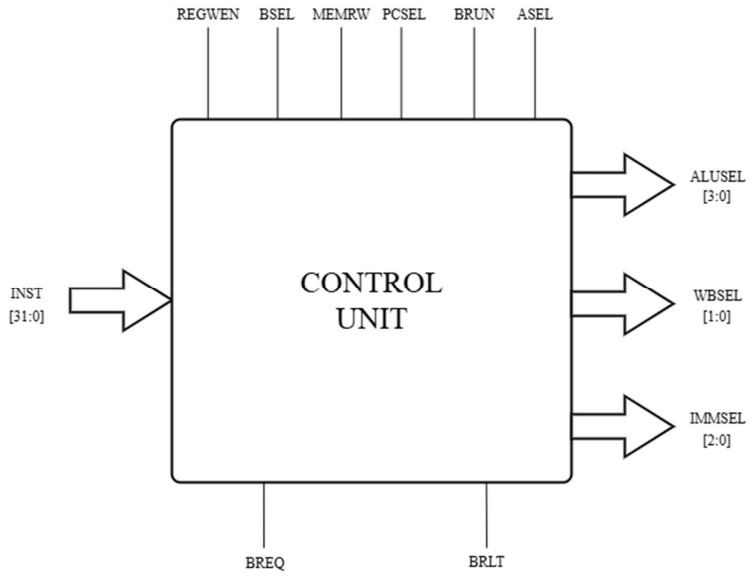


Figure 18: Interfaces of the Control Unit

Data Memory

The Data Memory module with Memory array size of 8KB (8192 bytes) addressing lower 13 bits of the input address. is responsible for handling load (read) and store (write) operations for a RISC-V processor. It is byte addressable and implements both signed and unsigned load and store operations.

The Data Memory has the following inputs and outputs:

Inputs:

- CLK (1-bit): Clock signal (synchronous write operations)
- ADDR (32-bit): Byte address for memory access
- DATAW (32-bit): Data to be written into memory
- MEMRW (1-bit): Read/Write control
- FUNC3 (3-bit): Encodes the load/store operation type (lb, lbu, lh, lhu, lw, sb, sh, sw)

Outputs:

- DATAR (32-bit): Data read from memory

Memory is byte addressable. The module correctly aligns data for halfword and word operations.

FUNC3	Function	Description
000	LB	Load Byte (signed) → Sign-extends 8-bit value to 32-bit
100	LBU	Load Byte Unsigned → Zero-extends 8-bit value to 32-bit
001	LH	Load Halfword (signed) → Sign-extends 16-bit value to 32-bit
101	LHU	Load Halfword Unsigned → Zero-extends 16-bit value to 32-bit
010	LW	Load Word (32-bit) → Reads full 32-bit word

Figure 19: Load Operations

FUNC3	Function	Description
000	SB	Store Byte → Writes only the least significant byte of DATAW
001	SH	Store Halfword → Writes the lower 16 bits of DATAW
010	SW	Store Word → Writes the entire 32-bit word to memory

Figure 20: Store Operations

Immediate Generator

The Immediate Generator module extracts immediate values from RISC-V instructions based on the instruction format. It correctly handles sign extension and bit manipulation to generate proper immediate values for different instruction types.

The Immediate Generator has the following inputs and outputs:

Inputs:

- INST (32-bit): The instruction from which to extract the immediate
- IMMSEL (3-bit): Selects the type of immediate to generate.

Outputs:

- IMM (32-bit): The sign-extended immediate output.

IMMSEL	Used Operations	Description
000	Used for immediate arithmetic, loads, and shift operations (e.g., addi, lw).	• Extracts bits 31-20 and sign-extends to 32-bit. • IMM = sign_extend(INST [31:20])
001	Used for store instructions (e.g., sw, sh, sb)	• Extracts bits 31-25 & 11-7 and sign-extends to 32-bit. • IMM = sign_extend(INST[31:25] & inst[11:7])
010	Used for branch instructions (e.g., beq, bne)	• Extracts bits 31, 7, 30-25, 11-8, adds a trailing 0 (word-aligned) • IMM = sign_extend(INST[31] & INST[7] & INST[30:25] & INST[11:8] & '0')
011	Used for jump instructions (e.g., jal)	• Extracts bits 31, 19-12, 11, 20, 30-21 and sign extends to 32-bit • IMM = sign_extend(INST[31] & INST[19:12] & INST[11] & INST[20] & INST[30:21])
100	Used for upper immediate instructions (e.g., lui, auipc).	• Extracts bits 31-12 and shifts left by 12 (zero-padded) • IMM = INST[31:12] & 12'b0

Figure 19: Load Operations

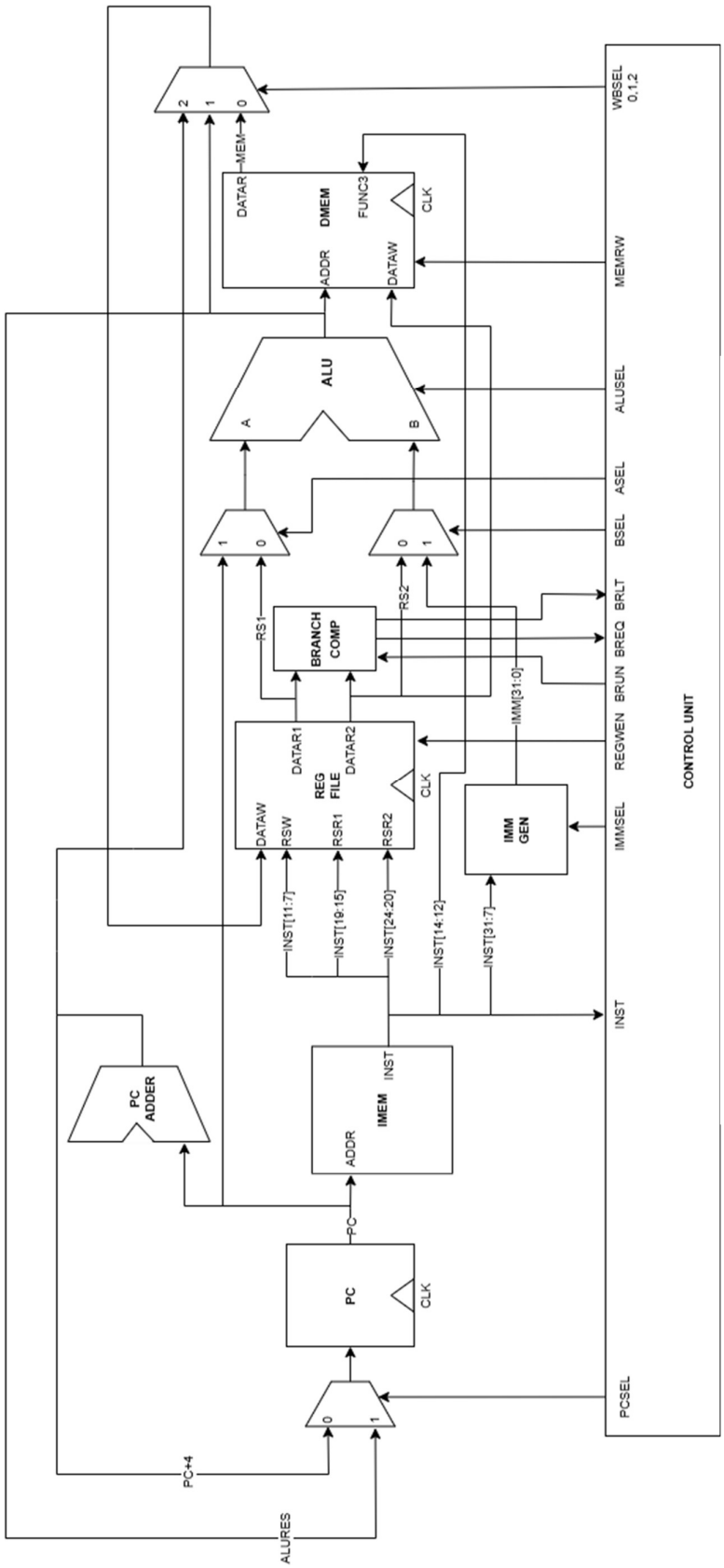


Figure 20: Fully connected datapath without pipeline implementation

PART 3 – Integration

Pipelining is a technique used in processor design to improve instruction throughput by overlapping execution stages. Instead of executing one instruction at a time, multiple instructions are processed simultaneously in different pipeline stages.

To implement pipelining efficiently, the processor is divided into five stages, each responsible for a specific part of instruction execution.

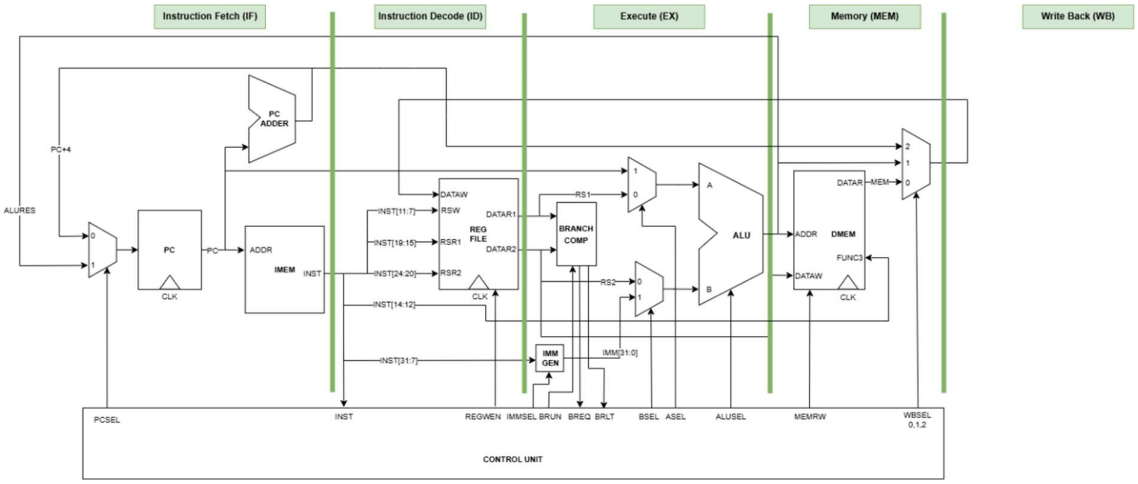


Figure 21: Pipeline stages

To implement an efficient pipelined processor, registers are placed between each stage to hold signals until the next clock cycle, ensuring proper data flow. The program counter (PC) is recalculated as PC+4 in the Memory stage to avoid unnecessary storage and transmission. The design follows a one control unit per stage approach, where each stage independently computes relevant signals, reducing complexity and dependencies. Instructions are continuously passed through the pipeline to ensure control logic operates correctly in each stage. Additionally, the RSW (Write Register Address) is modified to use the Rd field from the Write Back stage instruction rather than the Instruction Decode stage, preventing incorrect register writes due to instruction overlap. Furthermore, PC Select value found at execution stage (EX_PCSEL) along with the ALU Result are sent to the Program Counter Selector to fetch the correct instructions according to the branch condition calculated in the execution stage.

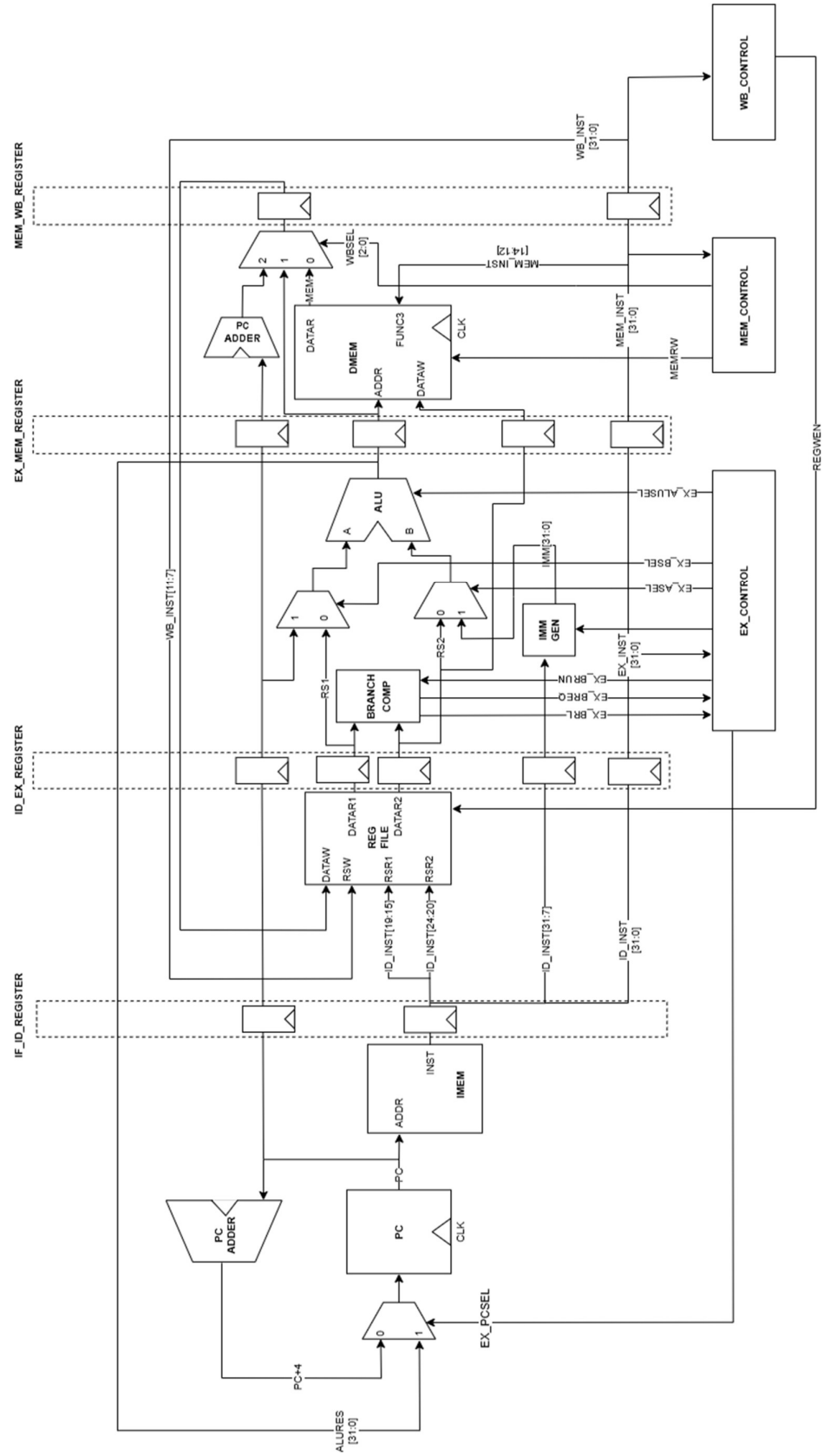


Figure 22: Datapath of the pipelined implementation

Execution Stage Control (EX CONTROL)

The EX_CONTROL unit is responsible for generating control signals required during the Execute (EX) stage of a pipelined processor. It takes the instruction from the EX stage, decodes its opcode and function fields, and sets various control signals that guide the Arithmetic Logic Unit, Immediate Generator and Branch Compare unit operations. The design follows a combinational logic approach, that outputs are directly determined by inputs without intermediate storage. When the reset signal is active, all control signals are set to their default values to prevent incorrect execution.

Memory Stage Control (MEM CONTROL)

The MEM_CONTROL unit is responsible for handling memory-related operations in the Memory (MEM) stage of a pipelined processor. It generates control signals that determine whether data should be read from or written to memory and selects the correct value to be forwarded to the Write Back (WB) stage. Control signals are set to their default values (MEMRW = '0', WBSEL = "00") to avoid unintentional memory writes.

Write Back Stage Control (WB CONTROL)

The WB_CONTROL unit manages the register write-back operations in the pipeline by controlling the REGWEN signal. For R-Type and I-Type instructions, which perform arithmetic or logical operations, the result must be written back to a register, so REGWEN is set to '1'. Similarly, Load instructions fetch data from memory and store it in a register, requiring REGWEN = '1'. However, Store and Branch instructions do not modify registers, so REGWEN is set to '0'. The JAL and JALR instructions, which save the return address (PC + 4) in a register, also need REGWEN = '1'. Additionally, LUI and AUIPC instructions, which manipulate immediate values and update registers, require register writes, so REGWEN is set to '1'. This ensures that only the appropriate instructions modify the register file, maintaining correct data flow in the pipeline.

A separate ID_Control unit is not required because the necessary control signals for instruction decoding and register access are already determined by other control units in later pipeline stages. In the Instruction Decode (ID) stage, the primary task is reading the register file.

Register File Asynchronous Read and Write

Note that in the pipeline implementation the Register File is no connected to the clock. The decision to make the register file as a module to be asynchronous for writes is based on the fact that the Write Back (WB) stage register (WB_Register) is already synchronous, ensuring that the write data is stable when it reaches the register file. Since the write enable signal (REGWEN) is generated synchronously in the WB stage, there is no need to introduce an additional clock cycle delay for writing into the register file. However, Register Writes overall being synchronous is preserved.

By allowing asynchronous writes, the register file can immediately update the destination register as soon as valid data is available from the WB stage, rather than waiting for the next clock edge. This ensures that the register file is updated without unnecessary delays, optimizing performance. Additionally, since reads from the register file are combinational, the updated values can be used in the next instruction cycle without requiring extra forwarding mechanisms.

This design choice keeps the pipeline efficient by eliminating unnecessary clock cycle delays, ensuring that instructions complete their execution without introducing artificial bottlenecks in the Write Back stage.

Test Programs

Following set of instructions were loaded to the instruction memory

```
0  => "00000000101000000000000001010010011",  -- addi r5, r0, 10    r5 = r0 + 10 = 10
1  => "11111100111000000000000000010010011",  -- addi r1, r0, -50   r1 = r0 + (-50) = -50
2  => "00000000111100000000000000010010011",  -- addi r2, r0, 15    r2 = r0 + 15 = 15
3  => "00000000010100001000000110110011",  -- add r7, r1, r5     r7 = r1 + r5 = -40
4  => "00000010010100001000000110110011",  -- mul r3, r1, r5     r3 = r1 * r5 = -500
5  => "00000010010100001010000110110011",  -- div r3, r2, r5     r3 = r2 / r5 = 1
6  => "00000010010100001011000110110011",  -- rem r3, r2, r5     r3 = r2 mod r5 = 5
7  => "00000000000100000010000000100011",  -- sw r1, 0(r0) store r1 first 4 bytes on 0x00, 0x01, 0x02, 0x03
8  => "00000000000000000000000001110000011",  -- lb r7, 0(r0) store first byte of 0x00 on r7
9  => "11111110000000000000011101100011",  -- beq r0, r0, -2 compare r0 and r0 and branch 2 half words back if true
```

Figure 23: Test instruction set for the pipeline processor

The first three instructions perform immediate addition (ADDI) to initialize registers with values. The processor should correctly update r5, r1, and r2 with 10, -50, and 15, respectively on subsequent clock cycles.

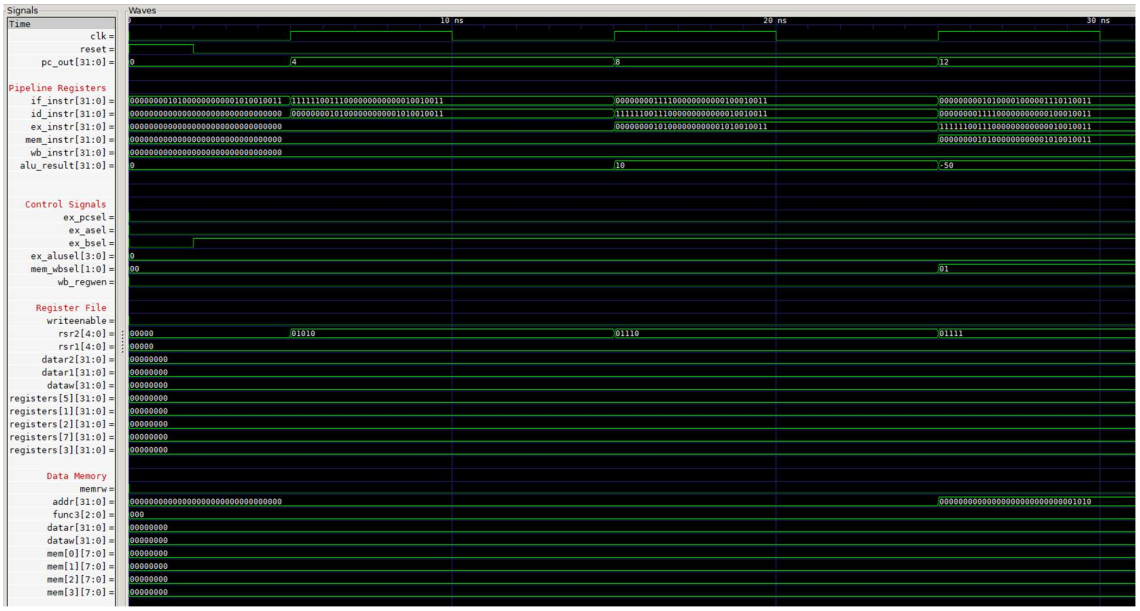


Figure 24: Waveform of the test instruction set

In the first few clock cycles instructions can be seen entering IF, ID, EX and MEM Stages. EX_ASEL signal ensures that the immediate value generated from the immediate value generator is sent to the ALU for the add operation.

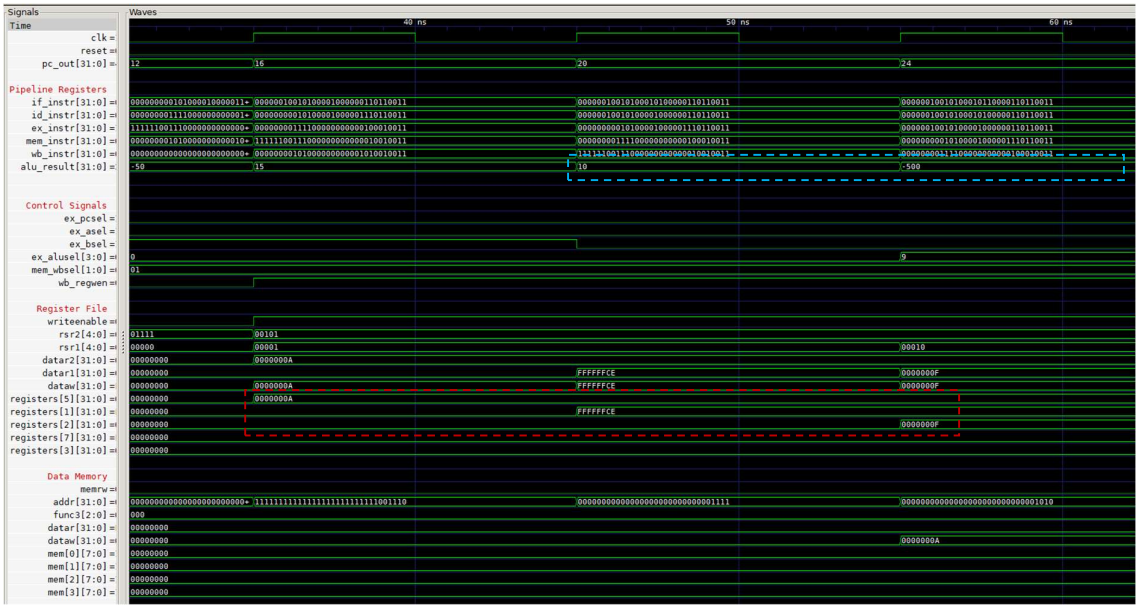


Figure 25: Waveform of the test instruction set continued

In the next few clock cycles it can be seen that the register values of the given register addresses being updated in the positive clock edge of each subsequent clock cycle (annotated in Red).

In the 4th, 5th, 6th and 7th instruction it tests if the processor can correctly execute a register-register arithmetic operation. MUL, DIV, and REM testing the multiplication, division, and remainder operations. It can be observed that the ALU outputs the almost all of the expected result when each instruction reaches the execution stage (annotated in Blue). Note that the expected result is -40 but the ALU result is 10. This is because value of r1 has not been written to the register file during the time where the Instruction Decode of the 4th instruction is done. Therefore, 0 is fetched as the r2 value resulting in r7 being stored as 10. This data hazard will be later resolved during hazard handling phase.

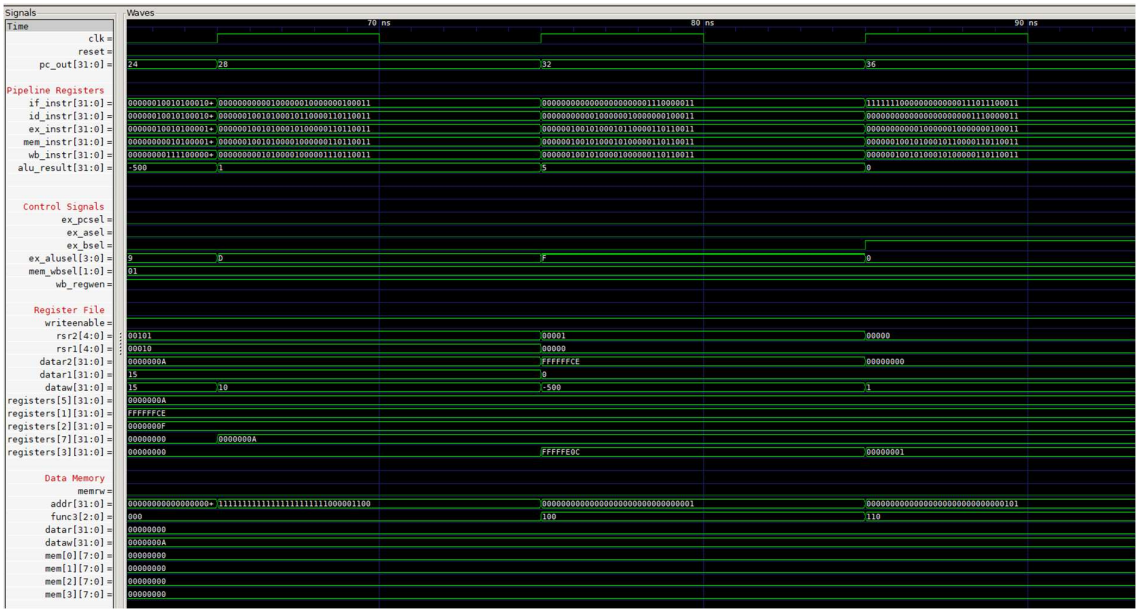


Figure 26: Waveform of the test instruction set continued

In the 8th and 9th instruction SW writes r1 (-50) into memory at address 0x00.arithmetic operation. LB loads a single byte from memory into r7. This tests byte addressing and memory access (annotated in Yellow).

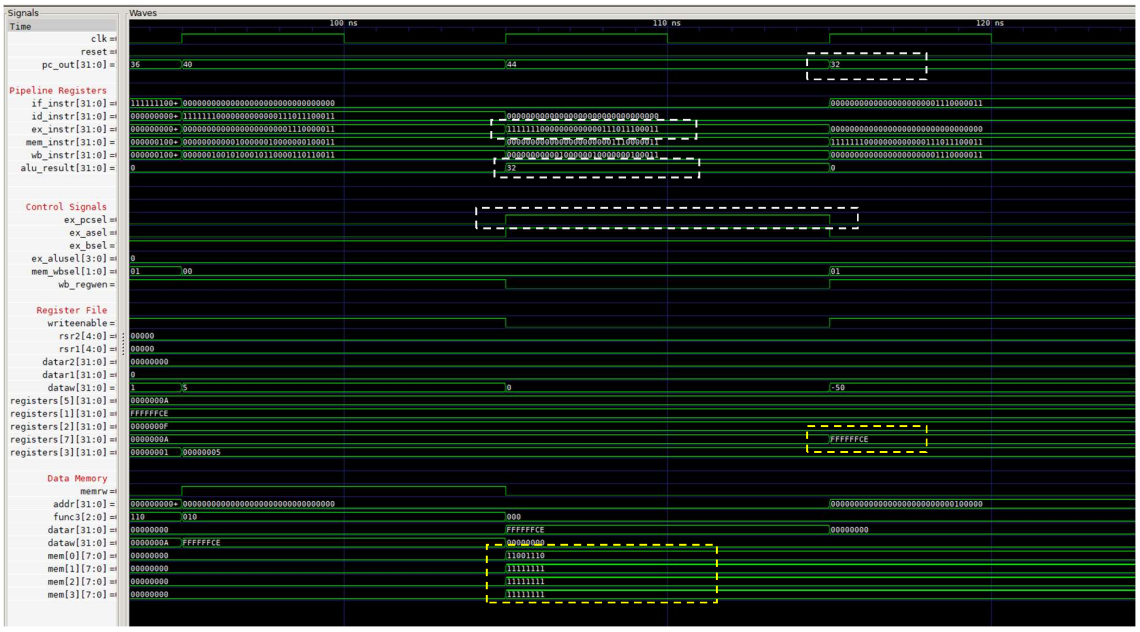


Figure 27: Waveform of the test instruction set continued

In the 10th instruction there is an infinite loop (BEQ r0, r0, -2). Since r0 is always 0, the condition is always true, and execution jumps back 1 instruction (two half words). When the branch instruction in in the execution stage the conditions are evaluated and decides whether or not to branch. This will assert the PCSEL signal and PC SELECTOR will send the ALU RESULT in the execution stage as the next PC value.

Since Control Hazards are not yet handled there will be 2 subsequent instructions after the Branch instruction inside the pipeline whatever the branch result is (can be observed as PC=40 and PC=44 in the above figure). These hazards can be handled for the time being by replacing them as NOPs.

PART 3 – Hazard Handling

Pipeline hazards occur when instructions in a pipeline interfere with each other, leading to stalls, incorrect execution, or performance degradation. There are Three main types of such hazards.

1. Data Hazards
2. Control Hazards
3. Structural Hazards

Data Hazards

A data hazard occurs when an instruction depends on the result of a previous instruction that has not yet completed.

One solution that can be implemented in hardware level is to integrate forwarding hardware into the pipeline processor. Instead of waiting for register values to be written to the register file, forward the result directly from the EX_MEM_REGISTER or MEM_WB_REGISTER pipeline register which avoids the need for stalling in most cases.

Hardware modifications done to handle data hazards are as follows,

In our pipeline processor, we introduced modifications to ASEL_MUX and BSEL_MUX to handle data hazards effectively using forwarding. Additionally, we updated the control unit logic to compare EX_INST, MEM_INST, and WB_INST to detect dependencies and apply the correct forwarding strategy.

The ASEL_MUX and BSEL_MUX are responsible for selecting the correct source operands for the ALU. The modifications ensure that if a required register value is not yet written back, the latest computed value is forwarded from a later pipeline stage. This allows immediate forwarding instead of waiting for the WB stage, reducing stalls and improving execution speed. To accommodate WB_REGWD (write back stage register value) and ALUR_MEM (ALU result in MEM stage). Both ASEL_MUX and BSEL mux inputs are required to be expanded up to 4 inputs. The control unit was modified to identify data hazards by comparing destination and source registers (RSR1, RSR2) across pipeline stages and it detects if an instruction in EX_INST needs data from MEM_INST or WB_INST and sets the selector signals accordingly.

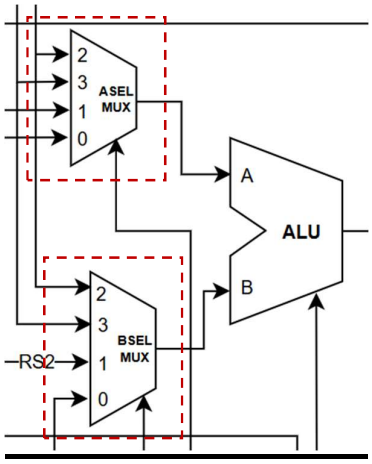


Figure 28: Modified ALU input selectors

To address load-use hazards, we introduced a unit called LOAD_USE that compares the instructions in the IF and ID stages. If a load-use hazard is detected, it generates a signal to notify both PC_ADDER and IMEM. In response, IMEM replaces the current instruction in the IF stage with a NOP. At the same time, PC_ADDER is reset to its previous value, ensuring that after the NOP, the next fetched instruction is the one that was initially replaced.

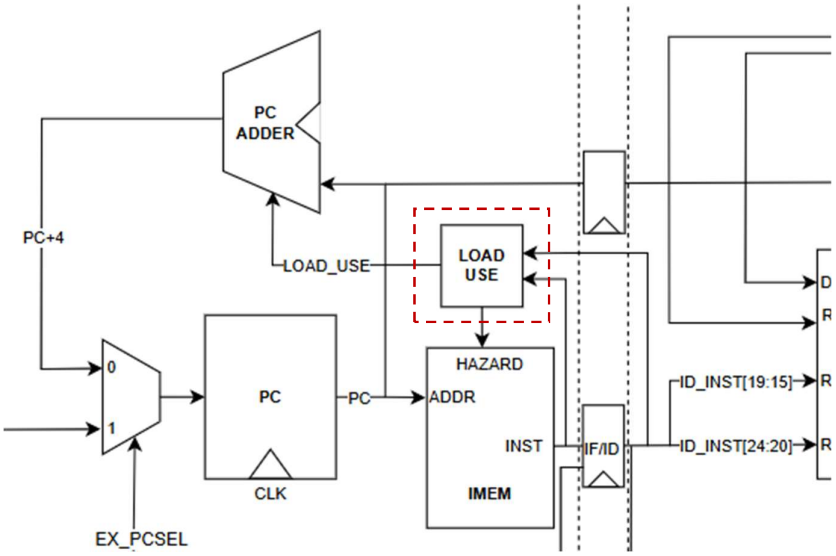


Figure 29: Load use hazard handling hardware modifications

Control Hazards

Control hazards occur when the processor encounters a branch instruction and does not know the outcome until a later stage in the pipeline. To minimize their impact, we implemented the technique of flushing the pipeline when a branch is taken, we insert a NOP into the ID and EX stages, effectively clearing incorrect instructions from the pipeline and then PC_ADDER updates the program counter to the target address. Here, the EX_PCSEL signal acts as the Control Hazard detection signal.

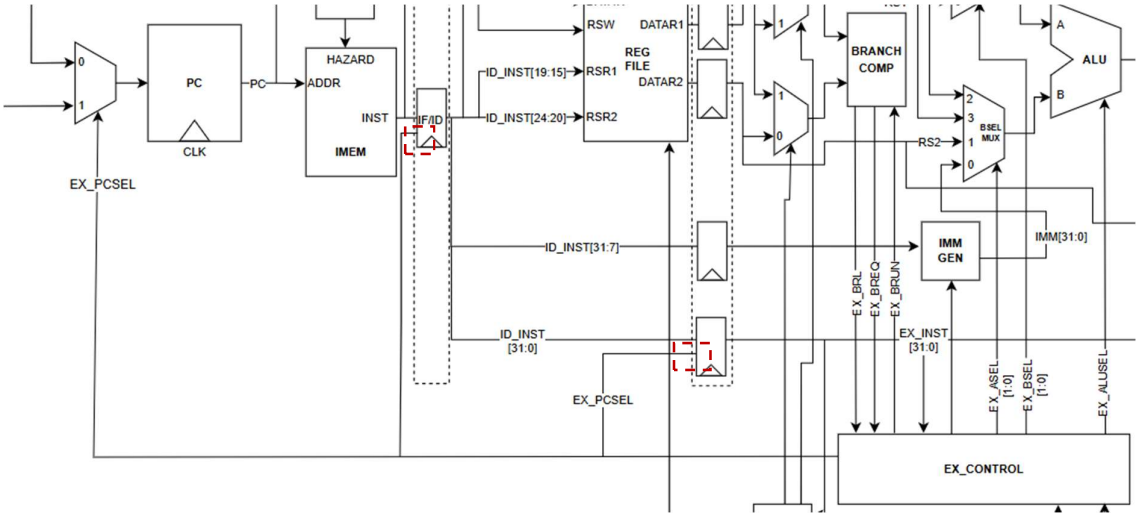


Figure 30: Control hazard handling hardware modifications

Structural Hazards

A structural hazard occurs when multiple instructions compete for the same hardware resource. Separate Instruction and Data Memory (Harvard Architecture), allowing multiple reads/writes to the register file simultaneously in different pipeline stages which are already implemented in the RISC V ISA prevents such hazards. No additional hardware modifications were required.

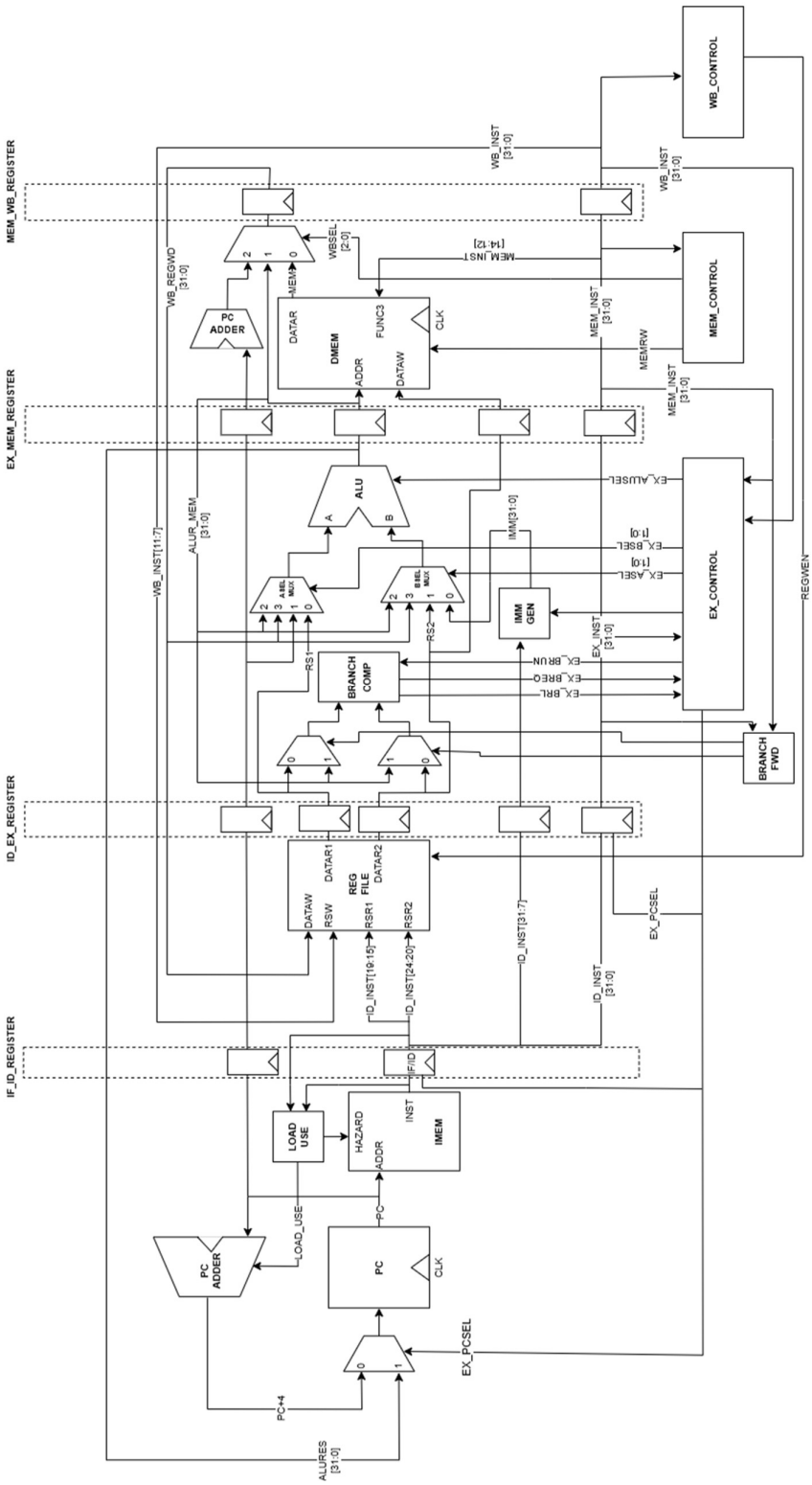


Figure 31: Datapath of the pipelined implementation with hazard handling hardware

Testing for Hazard Handling

Test for Data Hazard Handling (load use)

Before starting the program we store the value 15 in the address 0 of the data memory.

Test code:

lw r1, 0(r0)
add r2, r1, r1 – load use hazard
add r3, r1, r2 – forwarding

```
0  => "00000000000000000010000010000011",  -- lw r1, 0(r0)    ; Load from address 0 into r1
1  => "00000000000100001000000100110011",  -- add r2, r1, r1  ; Hazard! r1 is used before it's available
2  => "00000000001000001000000110110011",  -- add r3, r1, r2  ; Hazard! r1 & r2 is used before it's available
```

Figure 32: Test code for data hazard handling

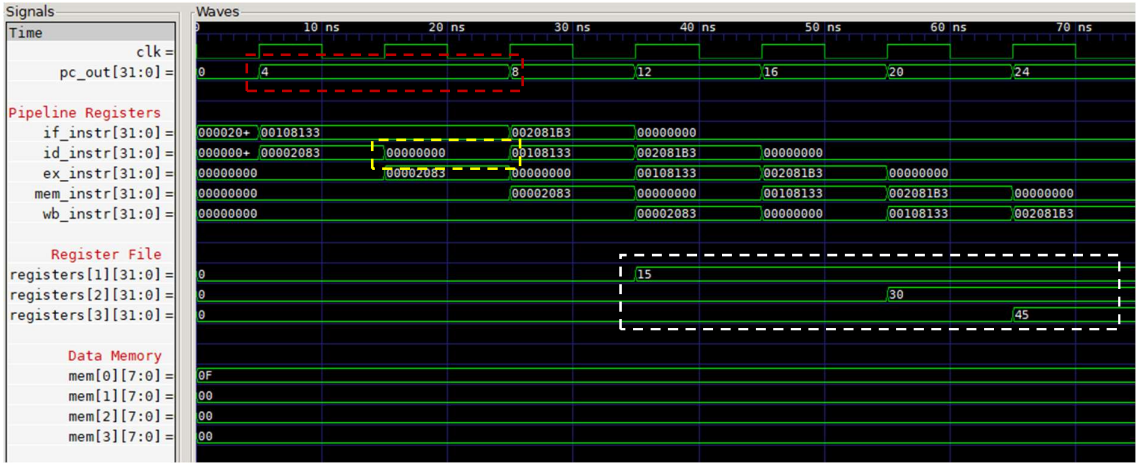


Figure 33: Waveform during data hazard handling test code

It can be observed that the program counter value is not updated in the next clock cycle once a load use hazard is identified (Red) and NOP instruction has been injected into the ID stage (Yellow). Finally, in the WB stage, the registers have been updated with the correct values (White).

Test for Data Hazard Handling (Forwarding)

Before starting the program, we store the value 11 and 10 in register r2 and r5.

Test code:

add r1, r0, r5
add r3, r1, r1
add r4, r3, r2

```
0  => "00000000010100000000000010110011",  -- add r1, r0, r5   ; r1 = r5
1  => "00000000000100010000000110110011",  -- add r3, r1, r2   ; Should use r1 without a stall
2  => "00000000001000011000001000110011",  -- add r4, r3, r2   ; Should use r3's result immediately
```

Figure 34: Test code for forwarding testing

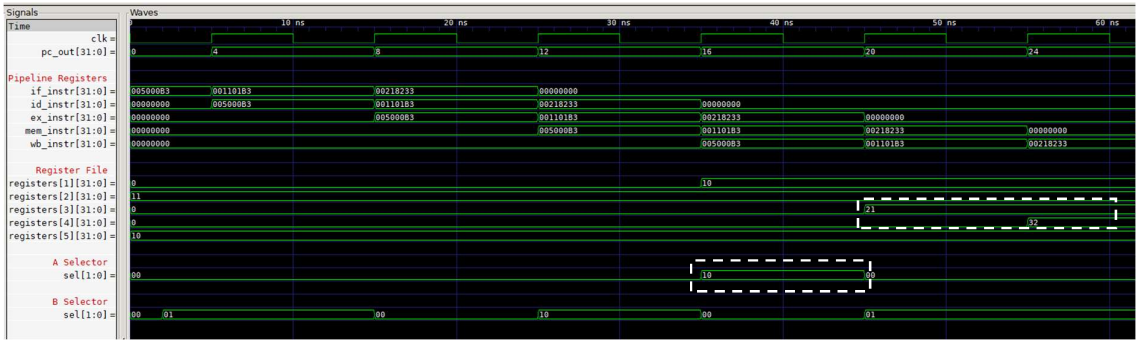


Figure 35: Waveform during forwarding test code

Here, the processor has used forwarding to allow r1 and r3 to be used immediately without inserting a stall. ASEL_MUX select signal can be observed to be set to select MEM_ALURES (ALU Result in MEM stage).

Test for Control Hazard Handling (Branch Hazard)

This test checks if the pipeline properly handles control hazards by flushing incorrect instructions after a branch.

Test code:

```
0  => "00000000101000000000001010010011", -- addi r5, r0, 10    r5 = r0 + 10 = 10
1  => "11111100111000101000000010010011", -- addi r1, r5, -50   r1 = r5 + (-50) = -40
2  => "01000000101000010000000110110011", -- sub r5, r1, r3     r3 = r1 - r5
3  => "00000000000100000010000000100011", -- SW r1, 0(r0)
4  => "01000000101000000000000110110011", -- sub r5, r0, r3     r3 = r0 - r5
5  => "11111110000000000000111011100011", -- beq r0, r0, -2 compare r0 and r0 and branch 2 half words back if true
6  => "00000000000000000001000111000011", -- lw r7, 0(r0)
7  => "00000000000000011100010000110011", -- add r16, r7, r0
```

Figure 36: Test code for pipeline flushing

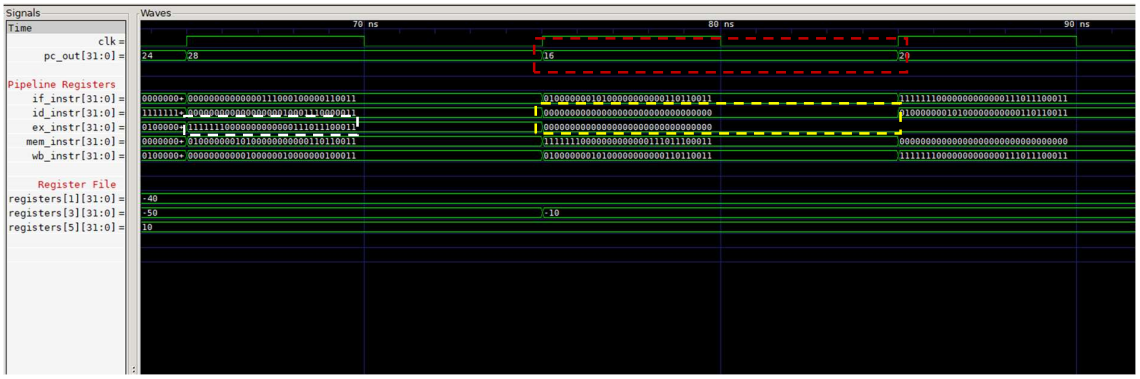


Figure 37: Waveform during pipeline flushing test code

It can be observed that when the BEQ instruction reaches the execution (White) stage and branch is taken following ID and EX instructions in the next clock cycle are flushed and replaced with NOPs (Yellow) additionally, the PC value is set to the branch target (Red). Note that PC_OUT in the figure is referring the PC value at the IF stage.

PART 4 – Using Synopsis Tools

Synopsis Design Vision

Synopsys Design Vision is a logic synthesis tool. It will take HDL designs and synthesize them to gate level HDL netlists. It can synthesize to generic gates or to other design libraries such as the vtvt_tsmc libraries or OSU standard cell libraries. The tool exists in a gui and command line version. The gui version is referred to as design vision and the command line version is known as design compiler (dc).

The basic steps:

Analyse

- This step checks the design files for syntax. This step also saves modules (verilog) and entities (vhdl) into a local folder in an intermediate format.

Elaborate

- This step builds a design from the intermediate format files created in the Elaborate step.

Compile

- This is the synthesizing step. Here the design is mapped to a gate library or cell library.

Save

- After compiling a design one can save the synthesized design into HDL or other formats.

After finishing the synthesis process, it is required to analyse the results of the synthesis to determine if there is a need to change code to achieve certain goals, or verify that goals have been met. These reports can also be used to offer comparisons of several synthesis attempts.

```
design_vision> analyze -format vhdl -library WORK ALU.vhdl
Running PRESTO HDLC
Compiling Entity Declaration ALU
Compiling Architecture ALU_ARCH of ALU
Presto compilation completed successfully.
1
design_vision> elaborate ALU -architecture ALU_ARCH
Running PRESTO HDLC
Presto compilation completed successfully. (alu)
Elaborated 1 design.
Current design is now 'alu'.
Information: Building the design 'adder_unit'. (HDL-193)
Presto compilation completed successfully. (adder_unit)
Information: Building the design 'subtract_slt_unit'. (HDL-193)
Presto compilation completed successfully. (subtract_slt_unit)
Information: Building the design 'and_unit'. (HDL-193)
Presto compilation completed successfully. (and_unit)
Information: Building the design 'or_unit'. (HDL-193)
Presto compilation completed successfully. (or_unit)
Information: Building the design 'sll_unit'. (HDL-193)
Presto compilation completed successfully. (sll_unit)
Information: Building the design 'SRL_unit'. (HDL-193)
Presto compilation completed successfully. (SRL_unit)
Information: Building the design 'arithmetic_shift_unit'. (HDL-193)
Presto compilation completed successfully. (arithmetic_shift_unit)
Information: Building the design 'mul_unit'. (HDL-193)
Presto compilation completed successfully. (mul_unit)
Information: Building the design 'div_unit'. (HDL-193)
Presto compilation completed successfully. (div_unit)
Information: Building the design 'mux16x32'. (HDL-193)
Warning:  ./mux16x32.vhdl:34: DEFAULT branch of CASE statement cannot be reached. (ELAB-311)

Statistics for case statements in always block at line 15 in file
        './mux16x32.vhdl'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            17            |    auto/auto     |
===============================================
Presto compilation completed successfully. (mux16x32)
Warning: Design 'xor_unit.db:xor_unit' comes before design 'xor_unit.ddc:xor_unit' in the li
nk_library; 'xor_unit.ddc:xor_unit' will be ignored. (UIO-92)
Information: Design 'xor_unit' is referenced in design 'alu.db:alu'. (UIO-93)
```

Figure 38: Analyse and elaborate the pipelined processor ALU

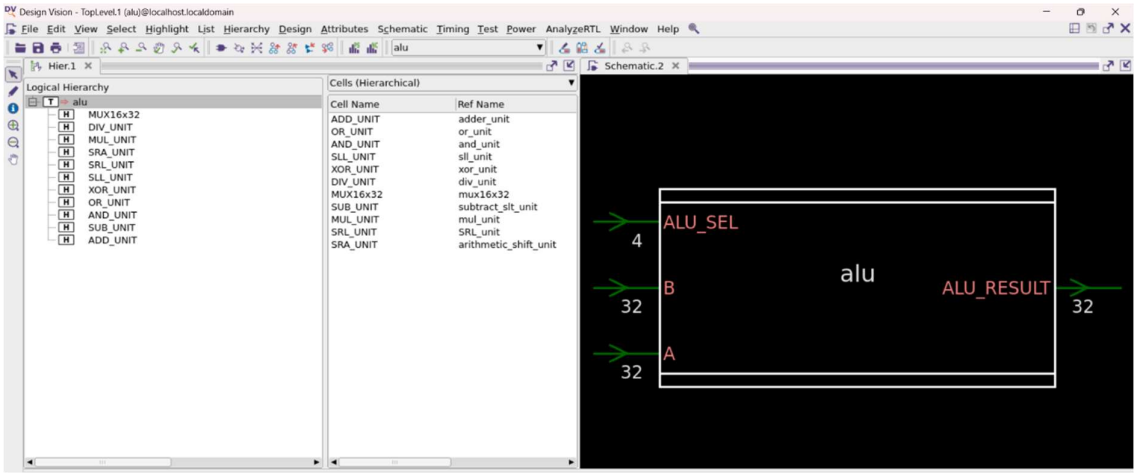


Figure 39: View of high-level schematic of pipelined processor ALU

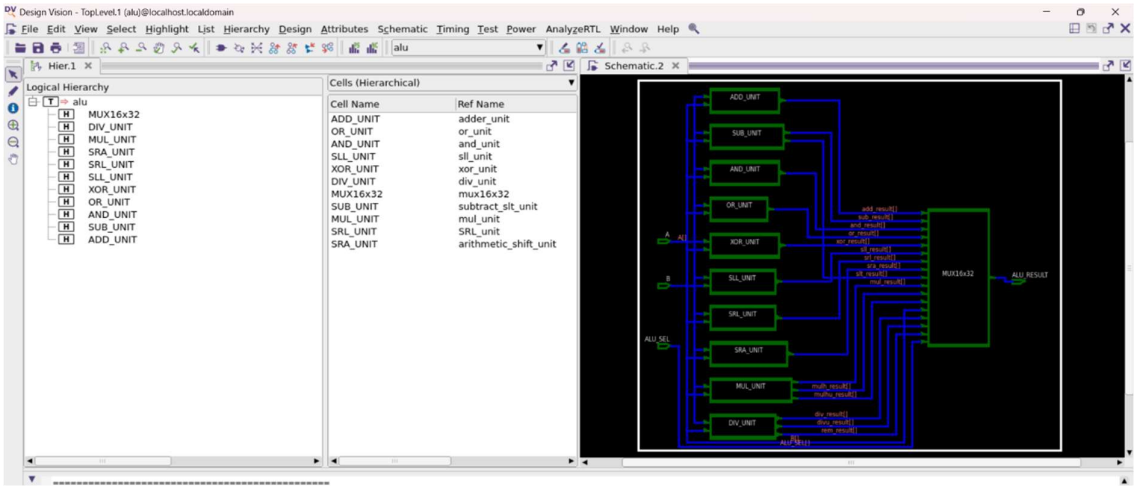


Figure 40: Expanded view of high-level schematic of pipelined processor ALU

report_design command displays attributes of the current design, such as technology library, flip-flop types, operation conditions.

```
Operating Conditions:


    Operating Condition Name : nom_pvt
    Library : class
    Process :   1.00
    Temperature :  25.00
    Voltage :   5.00
    Interconnect Model : balanced_tree

Wire Loading Model:

    No wire loading specified.


Wire Loading Model Mode: top.

Timing Ranges:

    No timing ranges specified.

Pin Input Delays:

    None specified.

Pin Output Delays:

    None specified.

Disabled Timing Arcs:

    No arcs disabled.

Required Licenses:

    None Required

Design Parameters:

    None specified.

Information: This design contains unmapped logic. (RPT-7)
1
design_vision>
```

Figure 41: Design report of pipelined processor ALU

The key process that Design Compiler does is RTL synthesis. This means, converting a gate level logic Verilog file to transistor level Verilog with the help of Technology library provided by the foundry and does area and delay optimizations on the current design.

```
design_vision> compile -map_effort medium
CPU Load: 1%, Ram Free: 26 GB, Swap Free: 15 GB, Work Disk Free: 62 GB, Tmp Disk Free: 7 GB
============================================================================================
| Flow Information                                                                         |
--------------------------------------------------------------------------------------------
| Flow         | Design Compiler                                                           |
============================================================================================
| Design Information                                    | Value                            |
============================================================================================
| Number of Scenarios                                   | 0                                |
| Leaf Cell Count                                       | 256                              |
| Number of User Hierarchies                            | 11                               |
| Sequential Cell Count                                 | 0                                |
| Macro Count                                           | 0                                |
| Number of Power Domains                               | 0                                |
| Number of Path Groups                                 | 1                                |
| Number of VT Class                                    | 0                                |
| Number of Clocks                                      | 0                                |
| Number of Dont Touch Cells                            | 41                               |
| Number of Dont Touch Nets                             | 0                                |
| Number of Size Only Cells                             | 0                                |
| Design with UPF Data                                  | false                            |
============================================================================================
```

Figure 42: Synthesis of pipelined processor ALU

```
  Beginning Pass 1 Mapping
  -----------------------
  Processing 'mux16x32'
  Processing 'div_unit'
S^H  Processing 'mul_unit'
  Processing 'arithmetic_shift_unit'
  Processing 'SRL_unit'
  Processing 'sll_unit'
  Processing 'xor_unit'
  Processing 'or_unit'
  Processing 'and_unit'
  Processing 'subtract_slt_unit'
  Processing 'adder_unit'
  Processing 'alu'
Warning: Design 'xor_unit.db:xor_unit' comes before design 'xor_unit.ddc:xor_unit' in the link_library; 'xor_unit.ddc:xor_unit' will be ignored. (UIO-92)
Information: Design 'xor_unit' is referenced in design 'alu.db:alu'. (UIO-93)

  Updating timing information
Information: Updating design information... (UID-85)
Information: Design 'alu' has no optimization constraints set. (OPT-108)

  Beginning Implementation Selection
  ----------------------------------
  Processing 'div_unit_DW_div_uns_0'
  Processing 'div_unit_DW_mod_tc_0'
  Processing 'div_unit_DW_div_tc_0'
  Processing 'mul_unit_DW02_mult_0'
  Processing 'mul_unit_DW01_add_0'
  Processing 'mul_unit_DW02_mult_1'
  Processing 'mul_unit_DW01_add_1'
  Processing 'arithmetic_shift_unit_DW01_ash_0'
  Processing 'arithmetic_shift_unit_DW01_sub_0'
  Processing 'arithmetic_shift_unit_DW_rash_0'
  Processing 'SRL_unit_DW_rash_0_DW_rash_1'
  Processing 'sll_unit_DW01_ash_0_DW01_ash_1'
  Processing 'subtract_slt_unit_DW01_cmp2_0'
  Processing 'subtract_slt_unit_DW01_sub_0_DW01_sub_1'
  Processing 'adder_unit_DW01_add_0_DW01_add_2'

  Beginning Mapping Optimizations  (Medium effort)
  -------------------------------
  Structuring 'mux16x32'
  Mapping 'mux16x32'
```

Figure 43: Synthesis of pipelined processor ALU (Mapping)

```
Processing 'adder_unit_DW01_add_0_DW01_add_2'

Beginning Mapping Optimizations  (Medium effort)
------------------------------
Structuring 'mux16x32'
Mapping 'mux16x32'
Structuring 'div_unit'
Mapping 'div_unit'
Structuring 'arithmetic_shift_unit'
Mapping 'arithmetic_shift_unit'
Structuring 'xor_unit'
Mapping 'xor_unit'
Structuring 'or_unit'
Mapping 'or_unit'
Structuring 'and_unit'
Mapping 'and_unit'

                                TOTAL
  ELAPSED            WORST NEG  SETUP    DESIGN
    TIME      AREA     SLACK     COST    RULE COST         ENDPOINT
---------- --------- --------- --------- --------- -------------------------
    0:01:27   46856.0      0.00       0.0       0.0
    0:01:27   46856.0      0.00       0.0       0.0
    0:01:27   46856.0      0.00       0.0       0.0
    0:01:27   46856.0      0.00       0.0       0.0
    0:01:28   46856.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0
```

Figure 44: Synthesis of pipelined processor ALU (Mapping)

```
Beginning Delay Optimization Phase
----------------------------------

                                TOTAL
  ELAPSED            WORST NEG  SETUP    DESIGN
    TIME      AREA     SLACK     COST    RULE COST         ENDPOINT
---------- --------- --------- --------- --------- -------------------------
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0


Beginning Area-Recovery Phase  (cleanup)
----------------------------

                                TOTAL
  ELAPSED            WORST NEG  SETUP    DESIGN
    TIME      AREA     SLACK     COST    RULE COST         ENDPOINT
---------- --------- --------- --------- --------- -------------------------
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:28   46237.0      0.00       0.0       0.0
    0:01:29   46235.0      0.00       0.0       0.0
    0:01:29   46235.0      0.00       0.0       0.0
    0:01:29   46235.0      0.00       0.0       0.0
    0:01:29   46235.0      0.00       0.0       0.0
    0:01:29   46235.0      0.00       0.0       0.0
    0:01:29   46235.0      0.00       0.0       0.0
    0:01:29   46235.0      0.00       0.0       0.0
    0:01:29   46235.0      0.00       0.0       0.0
    0:01:29   46235.0      0.00       0.0       0.0
Loading db file '/storage/synopsys/syn/V-2023.12-SP5-3/libraries/syn/class.db'


Note: Symbol # after min delay cost means estimated hold TNS across all active scenarios
```

Figure 45: Synthesis of pipelined processor ALU (Mapping)

Due to complete ALU schematic being too large to capture as a image, shown here is the schematic of the OR_UNIT in the ALU after synthesis.

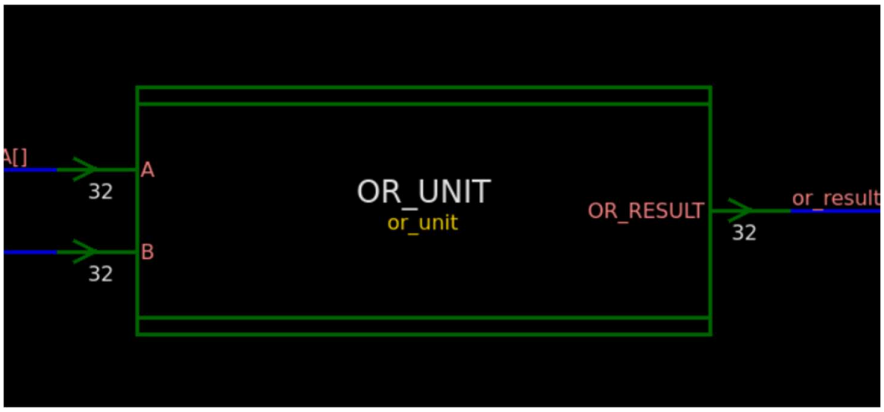


Figure 45: High-level view of OR_UNIT

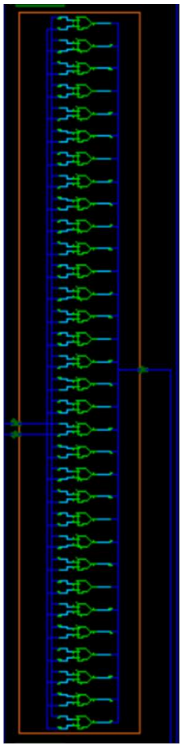


Figure 46: Gate-level view of OR_UNIT after synthesis

report_cell displays information about the ASIC logic cells in the current design.

```
design_vision> report_cell

****************************************
Report : cell
Design : alu
Version: V-2023.12-SP5-3
Date   : Sun Mar  9 00:02:45 2025
****************************************

Attributes:
    b - black box (unknown)
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic

Cell                    Reference       Library            Area  Attributes
--------------------------------------------------------------------------------
ADD_UNIT                adder_unit                    340.000000
                                                                h
AND_UNIT                and_unit                      64.000000 h
DIV_UNIT                div_unit                      20968.000000
                                                                h
MUL_UNIT                mul_unit                      21972.000000
                                                                h
MUX16x32                mux16x32                      748.000000
                                                                h
OR_UNIT                 or_unit                       64.000000 h
SLL_UNIT                sll_unit                      597.000000
                                                                h
SRA_UNIT                arithmetic_shift_unit         692.000000
                                                                h
SRL_UNIT                SRL_unit                      372.000000
                                                                h
SUB_UNIT                subtract_slt_unit             322.000000
                                                                h
XOR_UNIT                xor_unit                      96.000000 h
--------------------------------------------------------------------------------
Total 11 cells                                        46235.000000
1
```

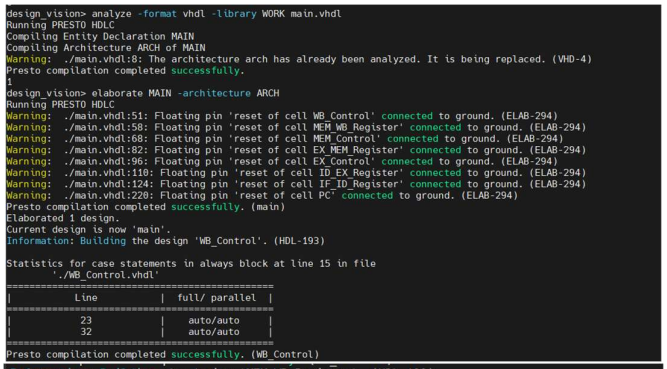
Figure 47: Cell report of pipelined processor ALU

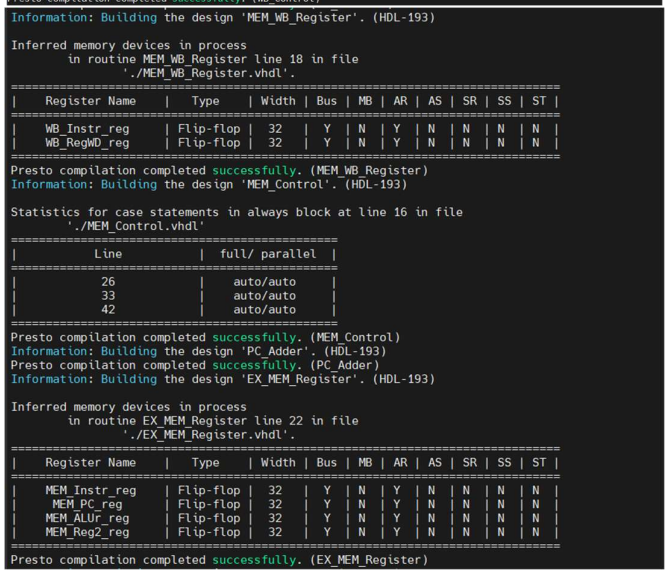
report _timing displays maximum delay timing in the current design.

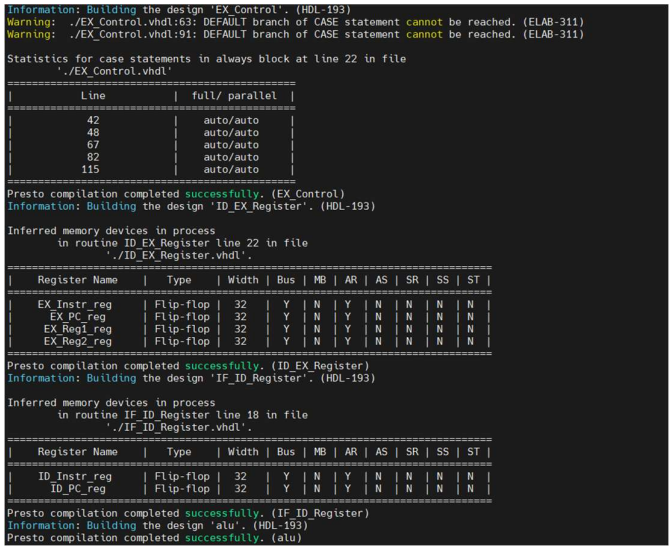
```
design_vision> report_timing
Information: Updating design information... (UID-85)

****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : alu
Version: V-2023.12-SP5-3
Date   : Sun Mar  9 00:03:30 2025
****************************************

Operating Conditions: nom_pvt   Library: class
Wire Load Model Mode: top

  Startpoint: B[31] (input port)
  Endpoint: ALU_RESULT[31]
            (output port)
  Path Group: (none)
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  alu                20x20                 class

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  DIV_UNIT/U19/Z (AN2)                                    0.87    1434.97 f
  DIV_UNIT/REM_RESULT[31] (div_unit)                      0.00    1434.97 f
  MUX16x32/IN15[31] (mux16x32)                            0.00    1434.97 f
  MUX16x32/U65/Z (AO2)                                    1.31    1436.28 r
  MUX16x32/U64/Z (ND8)                                    1.50    1437.78 f
  MUX16x32/DATA_OUT[31] (mux16x32)                        0.00    1437.78 f
  ALU_RESULT[31] (out)                                    0.00    1437.78 f
  data arrival time                                               1437.78
  --------------------------------------------------------------------------
  (Path is unconstrained)
```

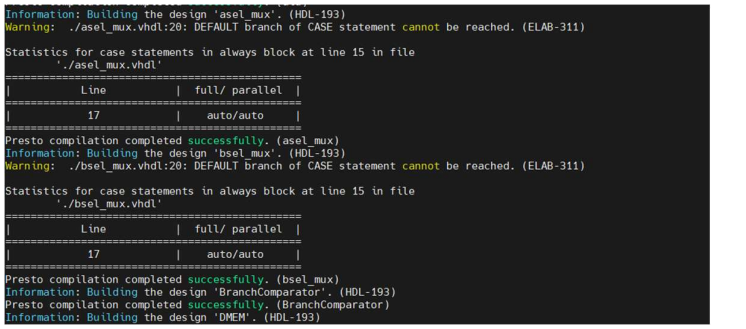
Figure 47: timing report of pipelined processor ALU

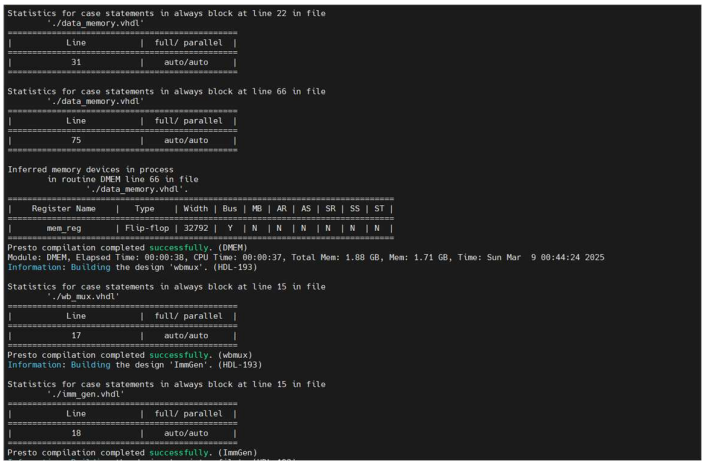
Then the complete processor was compiled, elaborated and synthesized. Given below are the results.

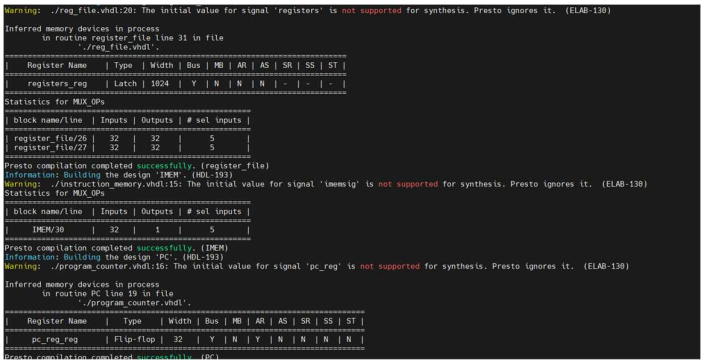












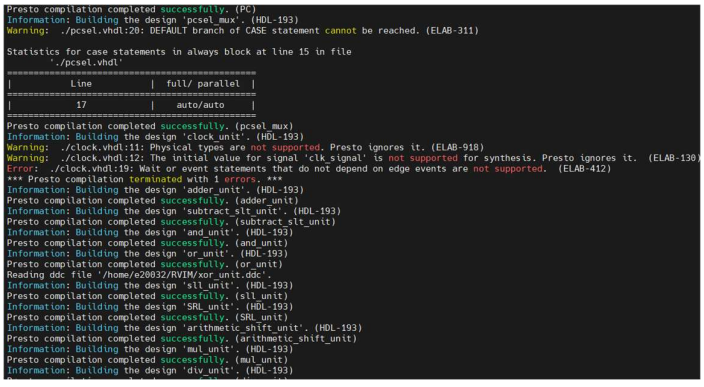


Figure 48: Elaboration results of top module (main.vhdl)

Note that the warnings in the process are due to artificial timing delays and signal variable assignments. These are default ignored during synthesis and causes no issue other than the warning.

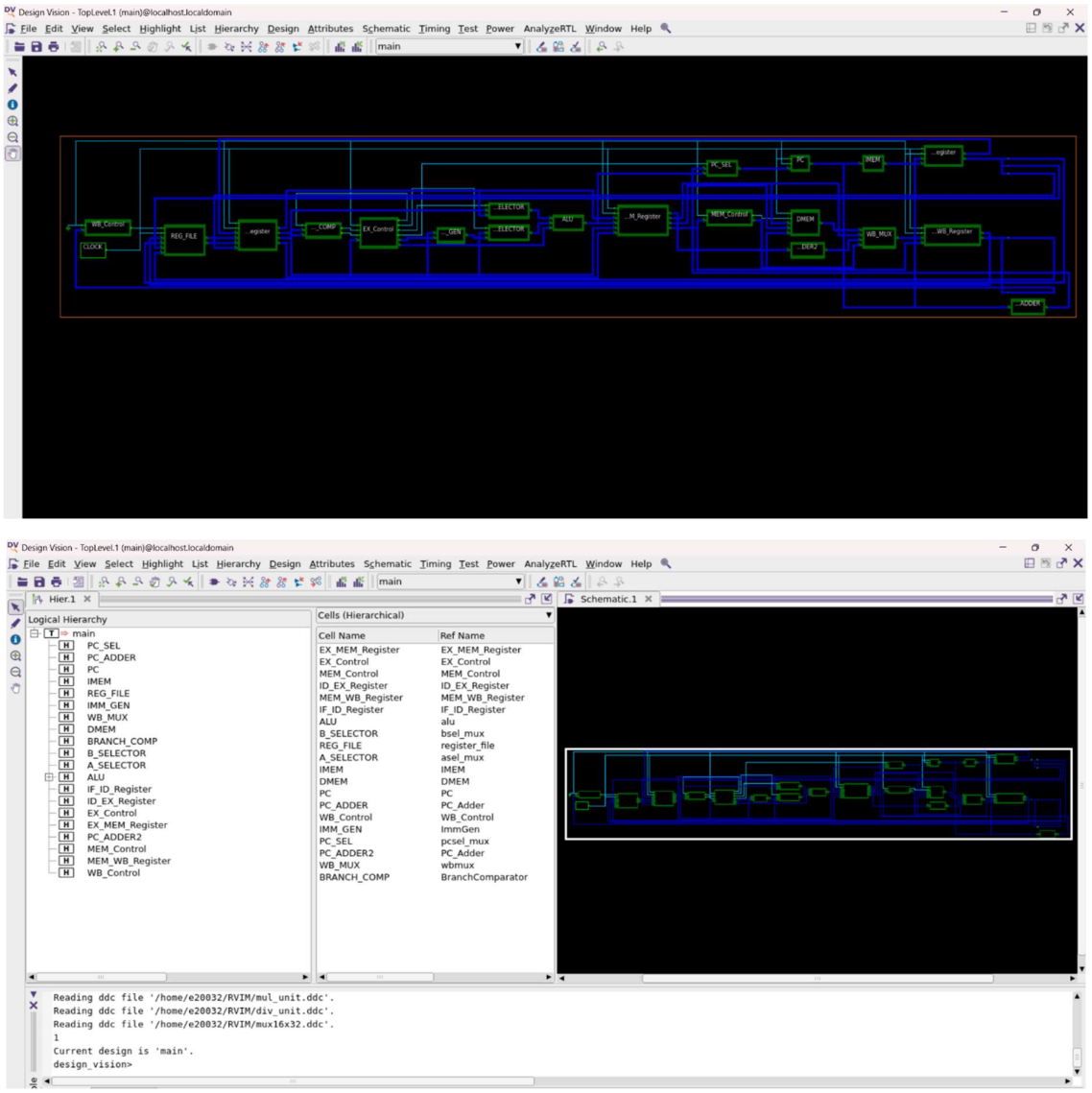


Figure 49: Schematic view after synthesis

```
design_vision> design_vision> report_area

****************************************
Report : area
Design : main
Version: V-2023.12-SP5-3
Date   : Sun Mar  9 01:55:56 2025
****************************************

Library(s) Used:

    gtech (File: /storage/synopsys/syn/V-2023.12-SP5-3/libraries/syn/gtech.db)
    class (File: /storage/synopsys/syn/V-2023.12-SP5-3/libraries/syn/class.db)

Number of ports:                         6905
Number of nets:                        523482
Number of cells:                       352716
Number of combinational cells:         285456
Number of sequential cells:             67124
Number of macros/black boxes:               0
Number of buf/inv:                      70283
Number of references:                      20

Combinational area:                96.000000
Buf/Inv area:                       0.000000
Noncombinational area:              0.000000
Macro/Black Box area:               0.000000
Net Interconnect area:      undefined  (No wire load specified)

Total cell area:                   96.000000
Total area:                 undefined

Information: This design contains unmapped logic. (RPT-7)
1
```

Figure 50: Area report after synthesis

```
****************************************
Report : power
         -analysis_effort low
Design : main
Version: V-2023.12-SP5-3
Date   : Sun Mar  9 02:18:08 2025
****************************************


Library(s) Used:

    gtech (File: /storage/synopsys/syn/V-2023.12-SP5-3/libraries/syn/gtech.db)
    class (File: /storage/synopsys/syn/V-2023.12-SP5-3/libraries/syn/class.db)

Information: The cells in your design are not characterized for internal power. (PWR-229)

Operating Conditions: nom_pvt   Library: class
Wire Load Model Mode: top


Global Operating Voltage = 5
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 0.100000ff
    Time Units = 1ns
    Dynamic Power Units = 100nW    (derived from V,C,T units)
    Leakage Power Units = Unitless


Attributes
----------
i - Including register clock pin internal power


  Cell Internal Power  =   0.0000 nW    (0%)
  Net Switching Power  =  15.0707  W  (100%)
                         ---------
Total Dynamic Power    =  15.0707  W  (100%)

Cell Leakage Power     =   0.0000
```

Figure 51: Power analysis report after synthesis

Synopsis VC Static

Lint_RTL

1. First, make a file where you need to do the Lint Test.

    mkdir myTest

    cd myTest

2. Then, make the vhdl/Verilog/systemVerilog file in there.

    vim test.vhdl

3. Then, make the **run.tcl** file

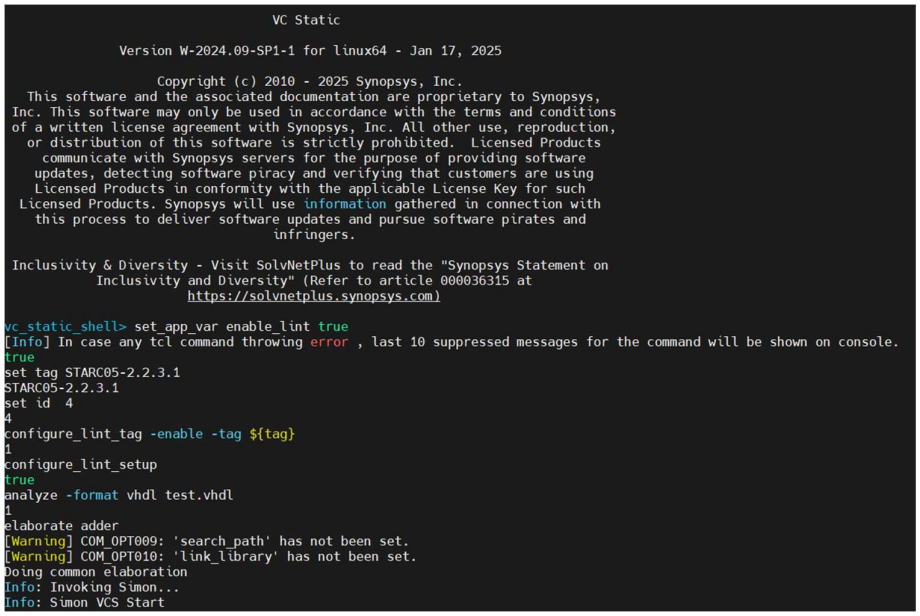
    vim run.tcl

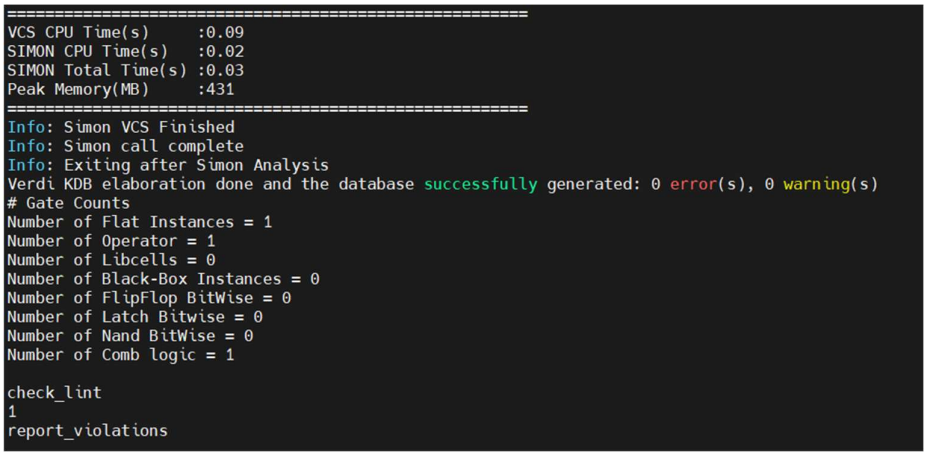
```
set_app_var enable_lint true

set tag STARC05-2.2.3.1
set id  4

configure_lint_tag -enable -tag ${tag}
configure_lint_setup

analyze -format vhdl test.vhdl

elaborate adder

check_lint
report_violations
report_violations -app {lint design} -limit 0 -verbose -file report_lint_before_fix_lint.txt
```

Note: Make sure to change the file name (test.vhdl) , file type (vhdl/sverilog/verilog), and the top module name (adder) in the run.t

4. Then run the run.tcl file

vc_static_shell -f run.tcl





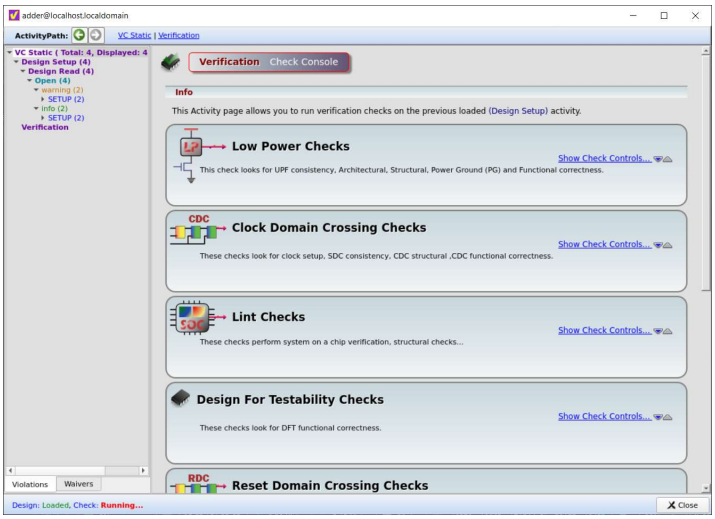
Note: You can review both created files using the cat <filename with extension>

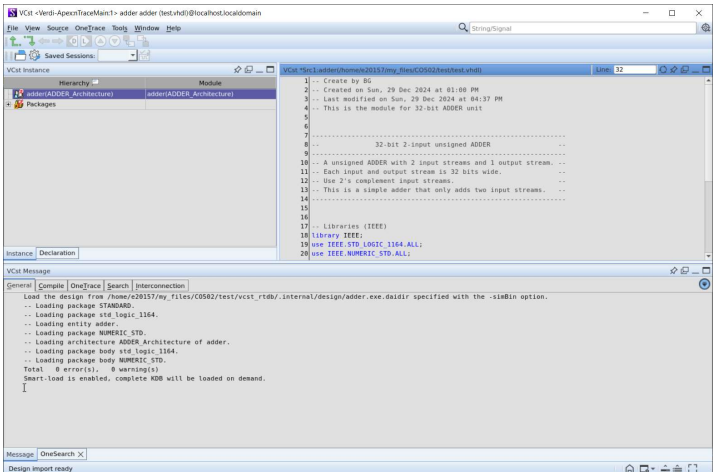
We can use a separate file directory for testing tools, and then we can import the .vhdl file and test it.

```
set_app_var enable_lint true

set tag STARC05-2.2.3.1
set id  4
set WORK_DIR "/home/e20157/my_files/CO502/test/"

configure_lint_tag -enable -tag ${tag}
configure_lint_setup

analyze -format vhdl "/home/e20157/my_files/CO502/test/test.vhdl"

elaborate adder

check_lint
report_violations
report_violations -app {lint design} -limit 0 -verbose -file report_lint_before_fix_lint.txt
```

We can do other VC Static test using the GUI

vc_static_shell> start_gui





Examples:

Component file: *adder.vhdl*

```
-- Create by BG
-- Created on Sun, 29 Dec 2024 at 01:00 PM
-- Last modified on Sun, 29 Dec 2024 at 04:37 PM
-- This is the module for 32-bit ADDER unit


----------------------------------------------------------------
--             32-bit 2-input unsigned ADDER                  --
----------------------------------------------------------------
-- A unsigned ADDER with 2 input streams and 1 output stream. --
-- Each input and output stream is 32 bits wide.              --
-- Use 2's complement input streams.                          --
-- This is a simple adder that only adds two input streams.   --
----------------------------------------------------------------


-- Libraries (IEEE)
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Entity (module)
entity adder is
    port(
        input_1  : in  std_logic_vector(31 downto 0);  -- First 32-bit signed input
        input_2  : in  std_logic_vector(31 downto 0);  -- Second 32-bit signed input
        output_1 : out std_logic_vector(31 downto 0)   -- 32-bit signed output
    );
end adder;

-- Architecture of the entity (module) - Defines its behavior
architecture ADDER_Architecture of adder is
begin
    output_1 <= std_logic_vector(unsigned(input_1) + unsigned(input_2));
end ADDER_Architecture;
```

Testbench file: *adder_tb.vhdl*

```
entity adder_tb is
end adder_tb;

-- Architecture for the Testbench
architecture TB_ARCH of adder_tb is

    -- Component Declaration (DUT: Device Under Test)
    component adder
        port(
            input_1  : in  std_logic_vector(31 downto 0);
            input_2  : in  std_logic_vector(31 downto 0);
            output_1 : out std_logic_vector(31 downto 0)
        );
    end component;
```

```
    -- Test Signals
    signal input_1_tb  : std_logic_vector(31 downto 0) := (others => '0');
    signal input_2_tb  : std_logic_vector(31 downto 0) := (others => '0');
    signal output_1_tb : std_logic_vector(31 downto 0);

begin
    -- Instantiate the Adder Module
    UUT: adder port map(
        input_1  => input_1_tb,
        input_2  => input_2_tb,
        output_1 => output_1_tb
    );

    -- Stimulus Process
    stimulus: process
    begin
        -- Test Case 1: 0 + 0
        input_1_tb <= std_logic_vector(to_unsigned(0, 32));
        input_2_tb <= std_logic_vector(to_unsigned(0, 32));
        wait for 10 ns;

        -- Test Case 2: 5 + 10
        input_1_tb <= std_logic_vector(to_unsigned(5, 32));
        input_2_tb <= std_logic_vector(to_unsigned(10, 32));
        wait for 10 ns;

        -- Test Case 3: 12345 + 67890
        input_1_tb <= std_logic_vector(to_unsigned(12345, 32));
        input_2_tb <= std_logic_vector(to_unsigned(67890, 32));
        wait for 10 ns;

        -- Test Case 5: 2's complement negative values (simulate signed addition)
        input_1_tb <= std_logic_vector(to_signed(-10, 32));
        input_2_tb <= std_logic_vector(to_signed(20, 32));
        wait for 10 ns;

        -- Stop Simulation
        wait;
    end process;
```

```
end TB_ARCH;
```

TCL file: *run.tcl*

```
 set_app_var enable_lint true

set tag STARC05-2.2.3.1
set id  4
set WORK_DIR "/home/e20157/my_files/CO502/test/"

 configure_lint_tag -enable -tag ${tag}
 configure_lint_setup

 analyze -format vhdl ${WORK_DIR}/test_tb.vhdl

 elaborate adder_tb

 check_lint
 report_violations
 report_violations -app {lint design} -limit 0 -verbose -file report_lint_before_fix_lint.txt
```

Result: vc_static_shell -f run.tcl

```
                                VC Static

              Version W-2024.09-SP1-1 for linux64 - Jan 17, 2025

                    Copyright (c) 2010 - 2025 Synopsys, Inc.
   This software and the associated documentation are proprietary to Synopsys,
 Inc. This software may only be used in accordance with the terms and conditions
 of a written license agreement with Synopsys, Inc. All other use, reproduction,
   or distribution of this software is strictly prohibited.  Licensed Products
     communicate with Synopsys servers for the purpose of providing software
    updates, detecting software piracy and verifying that customers are using
    Licensed Products in conformity with the applicable License Key for such
  Licensed Products. Synopsys will use information gathered in connection with
    this process to deliver software updates and pursue software pirates and
                                  infringers.

 Inclusivity & Diversity - Visit SolvNetPlus to read the "Synopsys Statement on
            Inclusivity and Diversity" (Refer to article 000036315 at
                        https://solvnetplus.synopsys.com)

vc_static_shell> set_app_var enable_lint true
[Info] In case any tcl command throwing error , last 10 suppressed messages for the command will be shown on console.
true
set tag STARC05-2.2.3.1
STARC05-2.2.3.1
set id  4
4
set WORK_DIR "/home/e20157/my_files/CO502/test/"
/home/e20157/my_files/CO502/test/
configure_lint_tag -enable -tag ${tag}
1
configure_lint_setup
true
analyze -format vhdl ${WORK_DIR}/test_tb.vhdl
1
```

```
elaborate test_tb
[Warning] COM_OPT009: 'search_path' has not been set.
[Warning] COM_OPT010: 'link_library' has not been set.
Doing common elaboration
Info: Invoking Simon...
Info: Simon VCS Start

=====================================================
VCS CPU Time(s)     :0.08
SIMON CPU Time(s)   :0.03
SIMON Total Time(s) :0.02
Peak Memory(MB)     :431
=====================================================
Info: Simon VCS Finished
Info: Simon call complete
Info: Exiting after Simon Analysis
Verdi KDB elaboration done and the database successfully generated: 0 error(s), 0 warning(s)
# Gate Counts
Number of Flat Instances = 65
Number of Operator = 64
Number of Libcells = 0
Number of Black-Box Instances = 1
Number of FlipFlop BitWise = 0
Number of Latch Bitwise = 0
Number of Nand BitWise = 0
Number of Comb logic = 4

check_lint
1
report_violations
```

```
  ------------------------------------------------------------------------------
  Management Summary
  ------------------------------------------------------------------------------
  Stage        Family                Fatals    Errors  Warnings      Infos
  -----        ------------------  --------  --------  --------  --------
  DESIGN_READ  DESIGN                     0         0         3         0
  DESIGN_READ  ErrorAnalyzeBBox           0         1         0         0
  TCL          SETUP                      0         0         2         0
  -----        ------------------  --------  --------  --------  --------
  Total                                   0         1         5         0


  ------------------------------------------------------------------------------
  Tree Summary
  ------------------------------------------------------------------------------
  Severity  Stage        Tag                 Count
  --------  -----        ------------------  -----
  error     DESIGN_READ  ErrorAnalyzeBBox       1
  warning   DESIGN_READ  SM_IGN_IV              2
  warning   DESIGN_READ  SM_URMI                1
  warning   TCL          COM_OPT009             1
  warning   TCL          COM_OPT010             1
  --------  -----        ------------------  -----
  Total                                         6


1
report_violations -app {lint design} -limit 0 -verbose -file report_lint_before_fix_lint.txt
1

1
vc_static_shell> quit
=====================================================
Total Time(S)  :207.92
CPU Time(S)    :7.44
Peak Memory(MB):1982
=====================================================
```

Example 02: Cloning a git repo and doing the RTL_Lint test

01. Make a directory and go to it

```
[e20157@localhost CO502]$ pwd
/home/e20157/my_files/CO502
[e20157@localhost CO502]$ mkdir RISC_V_32IM
[e20157@localhost CO502]$ cd RISC_V_32IM/
```

02. Clone the git repo

```
[e20157@localhost RISC_V_32IM]$ git clone https://github.com/Bimsara-Janakantha/RV32IM-Pipelined-Processor.git
Cloning into 'RV32IM-Pipelined-Processor'...
remote: Enumerating objects: 1229, done.
remote: Counting objects: 100% (174/174), done.
remote: Compressing objects: 100% (107/107), done.
remote: Total 1229 (delta 118), reused 102 (delta 66), pack-reused 1055 (from 2)
Receiving objects: 100% (1229/1229), 19.38 MiB | 3.39 MiB/s, done.
Resolving deltas: 100% (859/859), done.
```

03. Go to the directory that would be tested

```
[e20157@localhost my_files]$ cd CO502/
RISC_V_32IM/ test/
[e20157@localhost my_files]$ cd CO502/RISC_V_32IM/RV32IM-Pipelined-Processor/04_Hardware_Implementation/
[e20157@localhost 04_Hardware_Implementation]$ ls
2sCOMPLEMENTER.vhdl  Assembler.c        FORWARD.vhdl       instr_mem.mem  OR.vhdl              Readme.md          REG_IF_ID.vhdl   SLTU.vhdl
ADDER.vhdl           CONTROL_UNIT.vhdl  ghdl_run.sh        MASKTB.vhdl    PCTB.vhdl            REG_EX_MEM.vhdl    REG_MEM_WB.vhdl  SLT.vhdl
ALUTB.vhdl           CPUTB.vhdl         IMEM_TB.vhdl       MASK.vhdl      PC.vhdl              Reg_File_tb.vhdl   SHIFT.vhdl       testing
ALU.vhdl             CPU.vhdl           IMEM.vhdl          MUX2_1.vhdl    Pipeline_Reg_TB.vhdl REG_FILE.vhdl      Skelton.PNG      XOR.vhdl
AND.vhdl             DMEM.vhdl          IMM_DECORDER.vhdl  MUX4_1.vhdl    program.s            REG_ID_EX.vhdl     Skelton.vhdl
```

04. Make a directory called **testing**

```
[e20157@localhost 04. Hardware Implementation]$ mkdir testing
[e20157@localhost 04. Hardware Implementation]$ cd testing/
```

05. Make a run.tcl file with given template

```
set_app_var enable_lint true

set tag STARC05-2.2.3.1
set id  4
set WORK_DIR "/home/e20157/my_files/CO502/RISC_V_32IM/RV32IM-Pipelined-Processor/04_Hardware_Implementation/"

configure_lint_tag -enable -tag ${tag}
configure_lint_setup

analyze -format vhdl "${WORK_DIR}/CPUTB.vhdl"

elaborate CPUTB

check_lint
report_violations
report_violations -app {lint design} -limit 0 -verbose -file report_lint_before_fix_lint.txt
```

06. Run the run.tcl file

```
[e20157@localhost testing]$ vc_static_shell -f run.tcl

                                VC Static

              Version W-2024.09-SP1-1 for linux64 - Jan 17, 2025

                   Copyright (c) 2010 - 2025 Synopsys, Inc.
   This software and the associated documentation are proprietary to Synopsys,
 Inc. This software may only be used in accordance with the terms and conditions
 of a written license agreement with Synopsys, Inc. All other use, reproduction,
   or distribution of this software is strictly prohibited.  Licensed Products
     communicate with Synopsys servers for the purpose of providing software
    updates, detecting software piracy and verifying that customers are using
    Licensed Products in conformity with the applicable License Key for such
  Licensed Products. Synopsys will use information gathered in connection with
    this process to deliver software updates and pursue software pirates and
                                  infringers.

 Inclusivity & Diversity - Visit SolvNetPlus to read the "Synopsys Statement on
            Inclusivity and Diversity" (Refer to article 000036315 at
                        https://solvnetplus.synopsys.com)

vc_static_shell> set_app_var enable_lint true
[Info] In case any tcl command throwing error , last 10 suppressed messages for the command will be shown on console.
true
set tag STARC05-2.2.3.1
STARC05-2.2.3.1
set id  4
4
set WORK_DIR "/home/e20157/my_files/CO502/RISC_V_32IM/RV32IM-Pipelined-Processor/04_Hardware_Implementation/"
```

```
/home/e20157/my_files/CO502/RISC_V_32IM/RV32IM-Pipelined-Processor/04_Hardware_Implementation/
configure_lint_tag -enable -tag ${tag}
1
configure_lint_setup
true
analyze -format vhdl "${WORK_DIR}/CPUTB.vhdl"
1
elaborate CPUTB
[Warning] COM_OPT009: 'search_path' has not been set.
[Warning] COM_OPT010: 'link_library' has not been set.
Doing common elaboration
Info: Invoking Simon...
Info: Simon VCS Start
```

```
Warning: Ignoring Report Statement @ /home/e20157/my_files/CO502/RISC_V_32IM/RV32IM-Pipelined-Processor/04_Hardware_Implementation/CPUTB.vhdl:94
```

```
============================================================
VCS CPU Time(s)     :0.08
SIMON CPU Time(s)   :0.02
SIMON Total Time(s) :0.02
Peak Memory(MB)     :450
============================================================
Info: Simon VCS Finished
Info: Simon call complete
Info: Exiting after Simon Analysis
Verdi KDB elaboration done and the database successfully generated: 0 error(s), 0 warning(s)
# Gate Counts
Number of Flat Instances = 5
Number of Operator = 2
Number of Libcells = 0
Number of Black-Box Instances = 3
Number of FlipFlop BitWise = 0
Number of Latch Bitwise = 0
Number of Nand BitWise = 0
Number of Comb logic = 4

check_lint
1
report_violations
```

```
  ------------------------------------------------------------------------------
  Management Summary
  ------------------------------------------------------------------------------
  Stage        Family               Fatals    Errors  Warnings     Infos
  -----        ------------------  --------  --------  --------  --------
  DESIGN_READ  DESIGN                    0         0         4         0
  DESIGN_READ  ErrorAnalyzeBBox          0         3         0         0
  TCL          SETUP                     0         0         2         0
  -----        ------------------  --------  --------  --------  --------
  Total                                  0         3         6         0


  ------------------------------------------------------------------------------
  Tree Summary
  ------------------------------------------------------------------------------
  Severity  Stage        Tag                Count
  --------  -----        ------------------  -----
  error     DESIGN_READ  ErrorAnalyzeBBox        3
  warning   DESIGN_READ  SM_IGN_IV               1
  warning   DESIGN_READ  SM_URMI                 3
  warning   TCL          COM_OPT009              1
  warning   TCL          COM_OPT010              1
  --------  -----        ------------------  -----
  Total                                          9


1
report_violations -app {lint design} -limit 0 -verbose -file report_lint_before_fix_lint.txt
1

1
vc_static_shell> quit
============================================================
Total Time(S)  :1033.25
CPU Time(S)    :7.78
Peak Memory(MB):1982
============================================================
```

Limitations of the Processor

1. Lacks complex instructions such as floating-point operations and vector processing.
2. No cache memory, which causes frequent memory access delays.
3. No support for virtual memory or paging.
4. Follows a single core, single threaded execution model, limiting performance for modern applications.
5. Lacks system level features like interrupts, system calls, and a privileged execution mode.
6. Design doesn't optimize power consumption. It might be inefficient for embedded systems.

The pipelined processor with hazard handling faces limitations in performance, memory, parallel execution, and OS support. Upgrading with advanced pipelining, caching, multi-threading, and power optimizations can improve it.

Documentation

- Github Repository
  https://github.com/cepdnaclk/e20-co502-RV32IM_Pipelined_Processor_Group-04

- Project Site
  https://cepdnaclk.github.io/e20-co502-RV32IM_Pipelined_Processor_Group-04/

References

John Winans, “Release spelling and phrasing improvements · johnwinans/rvalp,” GitHub. Available: https://github.com/johnwinans/rvalp/releases/tag/v0.18.3.

A. Waterman and K. Asanović, The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

GitHub, “RISC-V.” Available: https://github.com/riscv

Cs61c.org, “Home | CS 61C Fall 2024.” Available: https://cs61c.org/fa24/

J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 6th ed. Morgan Kaufmann, 2020.

B. Kim and H. Lee, “Optimizing pipeline hazard handling techniques for high-performance processors,” in 2020 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1446-1450.

Francis G. Wolff, “EECS 317: Digital Logic Design,” Available: http://bear.ces.cwru.edu/eecs_316/eecs_317_20010216.pdf