IPA PROJECT

TEAM MEMBERS DETAILS

Eswar Kumar Anantha (2023102011)

K.Roshan Lal (2023102010)

M.Radheshyam(2023102032)

RISC-V Processor Architecture Design Report

Introduction

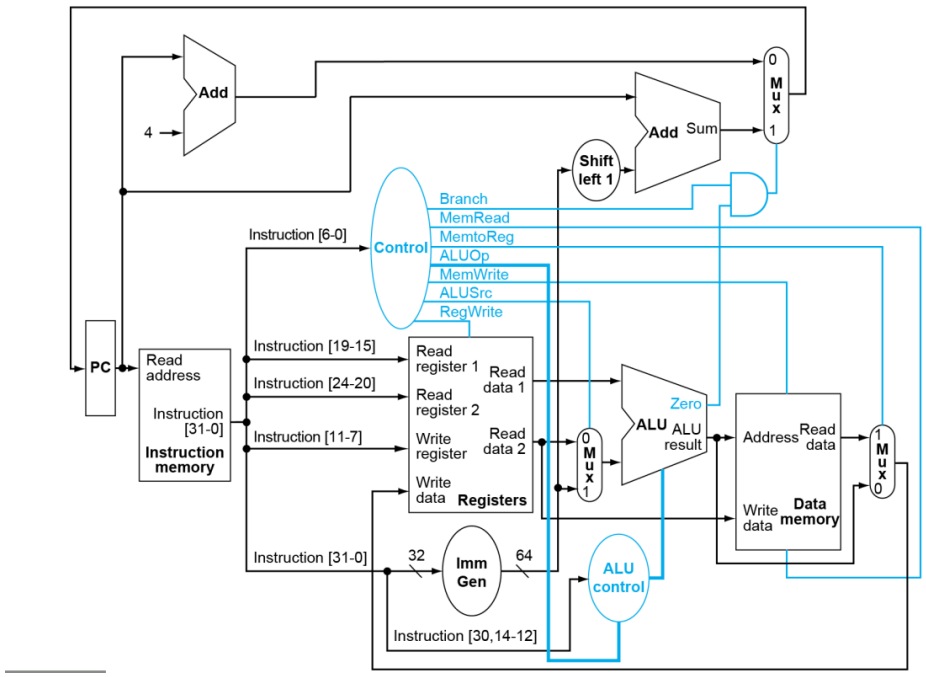
This report presents the design and implementation of a **RISC-V processor architecture** using Verilog. The design follows the **Reduced Instruction Set Computing (RISC)** principles and supports a subset of the **RISC-V ISA (Instruction Set Architecture)**. The processor is verified through simulation to ensure compliance with the specified requirements.

Processor Architecture Overview

The processor is implemented in **Verilog** and follows a **sequential-based** and **pipeline-based architecture**.

Sequential-based architecture

The **sequential-based processor architecture** follows a simple **single-cycle execution model**, where each instruction is **fetched, decoded, executed, and written back** before moving to the next instruction. This design follows **positive-edge clock updating**, meaning the processor updates the **program counter (PC)** and registers at the rising edge of the clock signal.



Unlike pipelined architectures, where multiple instructions execute in parallel across different stages, a **sequential processor completes one instruction per clock cycle** before starting the next instruction.

- At each **positive edge** of the clock, the processor fetches the next instruction from memory.

- The instruction is fully processed (fetch → decode → execute → memory → write-back).
- Only after completing the current instruction does the processor move to the next instruction in the following clock cycle.

This means that there are **no overlaps** between instructions, and each instruction takes a full clock cycle to complete.

Step-by-Step Execution and Design details in Each Clock Cycle

▼ Instruction Fetch (IF)

Design Details

The **Instruction Fetch (IF) module** is responsible for fetching instructions from the instruction memory based on the **Program Counter (PC)** value. The fetched instruction is then passed to the next stage of the processor for decoding and execution and so on .

This module is a fundamental part of the **sequential processor** and ensures that the correct instruction is read for every pc .

Module Overview

The `instruction_fetch` module contains the following key components:

- **Instruction Memory (`inst_mem`)**: A register array storing 256 instructions, each 32 bits wide.
- **Instruction Fetch Logic**: Reads the instruction based on the current **PC value**.
- **Initial Memory Setup**: The memory is preloaded with a program performing conditional branching and arithmetic operations etc .
- **Display Statements**: Provides debug information during simulation.

```
Fetching instruction at Address:          0
Instruction: 00000000001000001100001101100011
Time: 10000 | PC: 0 | Instruction: 00000000001000001100001101100011

Fetching instruction at Address:          1
Instruction: 00000000001000011000001010110011
Time: 20000 | PC: 4 | Instruction: 00000000001000011000001010110011

Fetching instruction at Address:          2
Instruction: 00000000000000000000001001100011
Time: 30000 | PC: 8 | Instruction: 00000000000000000000001001100011

Fetching instruction at Address:          3
Instruction: 01000000001000011000001010110011
Time: 40000 | PC: c | Instruction: 01000000001000011000001010110011

Fetching instruction at Address:          4
Instruction: 00000000001100010001001101100011
Time: 50000 | PC: 10 | Instruction: 00000000001100010001001101100011

Fetching instruction at Address:          5
Instruction: 00000000010000011110001100110011
Time: 60000 | PC: 14 | Instruction: 00000000010000011110001100110011

Fetching instruction at Address:          6
Instruction: 00000000000000000000001001100011
Time: 70000 | PC: 18 | Instruction: 00000000000000000000001001100011

Fetching instruction at Address:          7
Instruction: 00000000001000011111001100110011
Time: 80000 | PC: 1c | Instruction: 00000000001000011111001100110011
instruction_fetch_tb_1.v:23: $finish called at 80000 (1ps)
```

To finish the instructions we declared last instruction opcode is 1111111 then the program will be finished

CHALLENGES ENCOUNTERED :

while the branch is taken where it must go back to the previous instruction we didn't consider the negative sign so the pc is not updated correctly and instruction is not fetched correctly

INSTRUCTION DECODE

The **Instruction Decode Unit** extracts fields from a 32-bit instruction, generates control signals, and determines operand registers.

Inputs and Outputs

Inputs:

- `instruction [31:0]` : The **32-bit instruction** fetched from memory.

Outputs:

- **Instruction Fields:**

- `rs1 [4:0]` : First source register.
- `rs2 [4:0]` : Second source register (only for R-type and B-type).
- `rd [4:0]` : Destination register.
- `funct3 [2:0]` : Function code for ALU operations and branching.
- `funct7 [6:0]` : Additional function code for R-type operations.
- `opcode [6:0]` : The opcode that determines the instruction type.

- **Control Signals:**
  - `branch` : **1** if it is a branch instruction.
  - `alu_op [3:0]` : Operation code for ALU.
  - `alu_src` : Determines if one operand comes from immediate.
  - `imm [63:0]` : Sign-extended immediate value.
  - `memread` : **1** if data needs to be read from memory.
  - `memwrite` : **1** if data needs to be written to memory.
  - `memtoreg` : **1** if memory output is written to a register.
  - `regwrite` : **1** if the destination register should be written.

Design Description

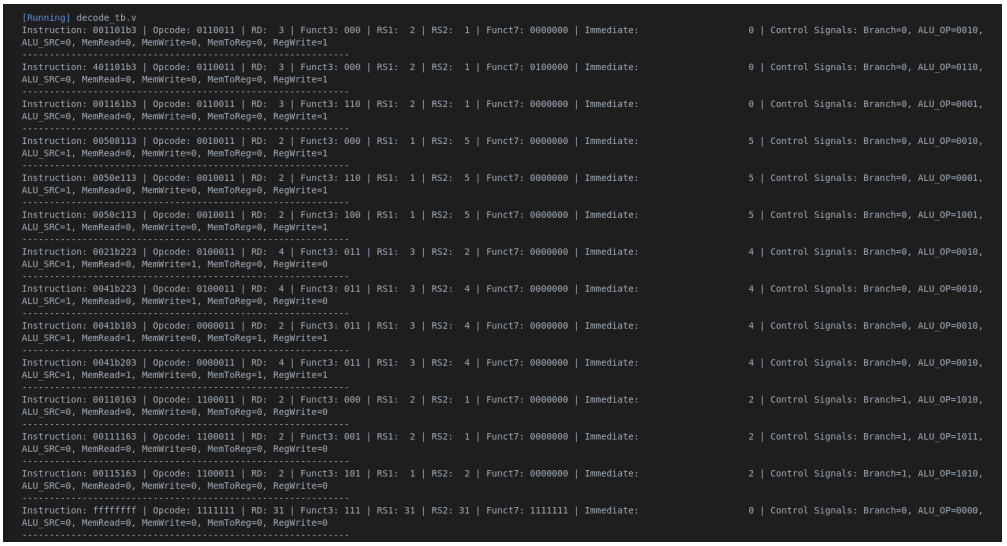
1. **Extract Fields**
   - `opcode = instruction[6:0]`
   - `rd = instruction[11:7]`
   - `funct3 = instruction[14:12]`
   - `funct7 = instruction[31:25]`
   - `rs1 = instruction[19:15]`
   - `rs2 = instruction[24:20]`
2. **Control Logic Based on `opcode`**
   - **R-type (Register-Register)** (`opcode = 0110011`):
     - Uses `rs1` , `rs2` , and `rd` .
     - ALU operation depends on `funct3` and `funct7` .
     - `regwrite = 1` , `alu_src = 0` , `memread = 0` , `memwrite = 0` .
   - **I-type (Immediate Instructions)** (`opcode = 0010011` and `0000011`):
     - Uses `rs1` and `rd` , with immediate value extracted.
     - `alu_src = 1` , `regwrite = 1` , and different `alu_op` values.
   - **S-type (Store Instructions)** (`opcode = 0100011`):
     - Uses `rs1` , `rs2` , and an immediate value.
     - `memwrite = 1` , `alu_src = 1` , and `regwrite = 0` .
   - **B-type (Branch Instructions)** (`opcode = 1100011`):
     - Uses `rs1` , `rs2` , and immediate.
     - `branch = 1` and different ALU operations (`BEQ` , `BNE` , `BGE` , etc.).

3. **Immediate Generation**
   - **I-type immediate** (for `ADDI`, `LW`, etc.):imm=SignExtend(instruction[31:20])

     imm=SignExtend(instruction[31:20])\text{imm} = \text{SignExtend}(instruction[31:20])
   - **S-type immediate** (for `SW`, `SH`, etc.):imm=SignExtend({instruction[31:25],instruction[11:7]})

     imm=SignExtend({instruction[31:25],instruction[11:7]})\text{imm} = \text{SignExtend}(\{instruction[31:25], instruction[11:7]\})
   - **B-type immediate** (for `BEQ`, `BNE`, etc.):imm=SignExtend({instruction[31],instruction[7],instruction[30:25],instruction[11:8],1'b0})

     imm=SignExtend({instruction[31],instruction[7],instruction[30:25],instruction[11:8],1'b0})\text{imm} = \text{SignExtend}(\{instruction[31], instruction[7], instruction[30:25], instruction[11:8], 1'b0\})
4. **Debugging Information**
   - `$display` prints the decoded instruction fields and control signals.



CHALLENGES ENCOUNTERED :

For the immediate we didnot consider the sign bit so error occured in the imm gen

EX_STAGE

This module is responsible for executing arithmetic, logical, and shift operations based on the **ALU control signals (`alu_op`)**. It receives operand values from the register file and an immediate value from the decode stage. The module supports different instruction types, including arithmetic, logical, shift, and comparison operations.

🔷 Inputs:

Signal Name	Width	Description
`decoder`	1-bit	Control signal indicating if the instruction is decoded correctly (not used in current design).
`rs1_val`	64-bit	Value of source register 1 (signed).

rs2_val	64-bit	Value of source register 2 (signed).
imm	64-bit	Immediate value (signed).
alu_src	1-bit	Control signal to select between rs2_val and imm as the second operand.
alu_op	4-bit	Control signal specifying the ALU operation.
branch	1-bit	Control signal for branch operations (not used in current design).

🔷 Outputs:

Signal Name	Width	Description
alu_result	64-bit	The computed result of the ALU operation.
overflow	1-bit	Indicates if an overflow occurred in addition or subtraction.

🔷 Internal Signals:

Signal Name	Width	Description
operand_b	64-bit	Selected operand (either rs2_val or imm).
add_result	64-bit	Result of addition (rs1_val + operand_b).
sub_result	64-bit	Result of subtraction (rs1_val - operand_b).
and_result	64-bit	Result of bitwise AND (rs1_val & operand_b).
or_result	64-bit	Result of bitwise OR (`rs1_val
xor_result	64-bit	Result of bitwise XOR (rs1_val ^ operand_b).
shifted_left	64-bit	Result of left shift (rs1_val << operand_b).
shifted_right_logical	64-bit	Result of logical right shift (rs1_val >> operand_b).
shifted_right_arithmetic	64-bit	Result of arithmetic right shift (rs1_val >>> operand_b).
lt_signed	1-bit	Result of signed comparison (rs1_val < operand_b).
lt_unsigned	1-bit	Result of unsigned comparison (rs1_val < operand_b using unsigned logic).
compartor	1-bit	Result of equality comparison (rs1_val == operand_b).
not_compartor	1-bit	Result of inequality comparison (rs1_val != operand_b).
add_overflow	1-bit	Overflow flag for addition.
shift_overflow	1-bit	Overflow flag for shifting (used for left shift).

🔷 Module Functionality

1. **Operand Selection (operand_b)**
   - If alu_src = 1, operand_b is imm (for immediate instructions).
   - If alu_src = 0, operand_b is rs2_val (for register-register instructions).
2. **ALU Operation Selection (alu_op)**
   - **Arithmetic Operations:**
     - 4'b0010 → Addition (add_result)
     - 4'b0110 → Subtraction (sub_result)
   - **Logical Operations:**
     - 4'b0000 → AND (and_result)
     - 4'b0001 → OR (or_result)
     - 4'b1001 → XOR (xor_result)

- **Shift Operations:**
  - `4'b0011` → Left Shift (`shifted_left`)
  - `4'b0100` → Logical Right Shift (`shifted_right_logical`)
  - `4'b0101` → Arithmetic Right Shift (`shifted_right_arithmetic`)
- **Comparison Operations:**
  - `4'b0111` → Unsigned Less Than (`lt_unsigned`)
  - `4'b1000` → Signed Less Than (`lt_signed`)
  - `4'b1010` → Equality Check (`compartor`)
  - `4'b1011` → Inequality Check (`~compartor`)

3. **Overflow Handling**
   - **Addition (`add_result`) and Subtraction (`sub_result`)** may cause overflow, which is stored in `overflow`.
   - **Shift operations** may also cause overflow (only left shift considered).

🔷 Component Instantiations

- **Arithmetic Modules:** `add` , `sub`
- **Logical Modules:** `and_gate` , `or_gate` , `xor_gate`
- **Shifting Modules:** `left_shifter` , `right_shifter_logical` , `right_shifter_arithmetic`
- **Comparison Modules:** `signed_lt_comparator` , `unsigned_lt_comparator` , `comparator`

Each module computes its respective function, and the results are **multiplexed using a case statement** inside an `always @(*)` block.

```
vvp processor_tb
VCD info: dumpfile processor_tb.vcd opened for output.
Time=0 | Instr=00000000001100010000000010110011 | Reg1= 9223372036854775807 | Reg2=                   1 | Result=-9223372036854775808 | Overflow=1
Time=10 | Instr=00000000001100010000000010110011 | Reg1=-9223372036854775808 | Reg2=                  -1 | Result= 9223372036854775807 | Overflow=1
Time=20 | Instr=00000000001100010000000010110011 | Reg1= 9223372036854775807 | Reg2=-9223372036854775808 | Result=                  -1 | Overflow=0
Time=30 | Instr=00000000001100010000000010110011 | Reg1=                 -90 | Reg2=                  40 | Result=                 -50 | Overflow=0
Time=40 | Instr=01000000001100010000000010110011 | Reg1= 9223372036854775807 | Reg2=                  -1 | Result=-9223372036854775808 | Overflow=1
Time=50 | Instr=01000000001100010000000010110011 | Reg1=-9223372036854775808 | Reg2=                   1 | Result= 9223372036854775807 | Overflow=1
Time=60 | Instr=01000000001100010000000010110011 | Reg1=-9223372036854775808 | Reg2= 9223372036854775807 | Result=                   1 | Overflow=1
Time=70 | Instr=01000000001100010000000010110011 | Reg1=                  -50 | Reg2=                 -89 | Result=                  39 | Overflow=0
Time=80 | Instr=00000000001100010111000010110011 | Reg1=    -281470681808896 | Reg2=   71777214294589695 | Result=   71776119077928960 | Overflow=0
Time=90 | Instr=00000000001100010111000010110011 | Reg1=    -281470681808896 | Reg2=   71777214294589695 | Result=    -280375465148161 | Overflow=0
Time=100 | Instr=00000000001100010100000010110011 | Reg1=    -281470681808896 | Reg2=   71777214294589695 | Result=  -72056494543077121 | Overflow=0
Time=110 | Instr=00000000001100010001000010110011 | Reg1= 9223372036854775807 | Reg2=                   1 | Result=                  -2 | Overflow=1
Time=120 | Instr=00000000001100010001000010110011 | Reg1=                   1 | Reg2=                  31 | Result=          2147483648 | Overflow=0
Time=130 | Instr=00000000001100010001000010110011 | Reg1=-9223372036854775807 | Reg2=                   3 | Result=                   8 | Overflow=1
Time=140 | Instr=00000000001100010001000010110011 | Reg1=                  -1 | Reg2=                  54 | Result=  -18014398509481984 | Overflow=0
Time=150 | Instr=00000000001100010101000010110011 | Reg1= 9223372036854775807 | Reg2=                   1 | Result= 4611686018427387903 | Overflow=0
Time=160 | Instr=00000000001100010101000010110011 | Reg1=                   1 | Reg2=                  62 | Result=                   0 | Overflow=0
Time=170 | Instr=00000000001100010101000010110011 | Reg1=-9223372036854775807 | Reg2=                  62 | Result=                   0 | Overflow=0
Time=180 | Instr=00000000001100010101000010110011 | Reg1=                  -1 | Reg2=                  45 | Result=              262143 | Overflow=0
Time=190 | Instr=01000000001100010101000010110011 | Reg1= 9223372036854775807 | Reg2=                   9 | Result=   18014398509481983 | Overflow=0
Time=200 | Instr=01000000001100010101000010110011 | Reg1=                   1 | Reg2=                   4 | Result=                   0 | Overflow=0
Time=210 | Instr=01000000001100010101000010110011 | Reg1=-9223372036854775807 | Reg2=                  34 | Result=          -536870912 | Overflow=0
Time=220 | Instr=01000000001100010101000010110011 | Reg1=                  -1 | Reg2=                   1 | Result=                  -1 | Overflow=0
Time=230 | Instr=00000000001100010010000010110011 | Reg1=                  -1 | Reg2=                 -30 | Result=                   0 | Overflow=0
Time=240 | Instr=00000000001100010010000010110011 | Reg1= 9223372036854775807 | Reg2=-9223372036854775808 | Result=                   0 | Overflow=0
Time=250 | Instr=00000000001100010010000010110011 | Reg1=                  30 | Reg2=                 -15 | Result=                   0 | Overflow=0
Time=260 | Instr=00000000001100010010000010110011 | Reg1=                 -40 | Reg2=                  50 | Result=                   1 | Overflow=0
Time=270 | Instr=00000000001100010011000010110011 | Reg1=                   1 | Reg2=                  30 | Result=                   1 | Overflow=0
Time=280 | Instr=00000000001100010011000010110011 | Reg1= 9223372036854775807 | Reg2= 9223372036854775807 | Result=                   0 | Overflow=0
Time=290 | Instr=00000000001100010011000010110011 | Reg1=                  30 | Reg2=                  15 | Result=                   0 | Overflow=0
Time=300 | Instr=00000000001100010011000010110011 | Reg1=                  40 | Reg2=                  50 | Result=                   1 | Overflow=0
src/processor_tb.v:79: $finish called at 310 (1s)
gtkwave processor_tb.vcd &
```

MEMORY STAGE

The `memory` module acts as the **data memory unit** in a processor pipeline, providing **load (memread) and store (memwrite) operations**. It interfaces with the **ALU result** for memory addressing and interacts with **register data** for storing values.

🔷 Module Functionality

1️⃣ Inputs

Signal	Width	Description
`alu_result`	64-bit	Memory address (used for both read and write)
`clk`	1-bit	Clock signal (used for write operations)
`memread`	1-bit	Control signal; **1** enables memory read
`memwrite`	1-bit	Control signal; **1** enables memory write
`rs2`	64-bit	Data to be written into memory when `memwrite = 1`

2️⃣ Outputs

Signal	Width	Description
`mem_data`	64-bit	Data read from memory if `memread = 1`

🔷 Memory Structure

- The module uses a **256-entry register array (`data_memory [0:255]`)**.
- Each memory location holds a **64-bit value**.
- **Memory addressing is done using the lower 8 bits** of `alu_result`, divided by `8` (`alu_result[7:0] >> 3`) to align **64-bit word addressing**.

🔷 Behavior

1️⃣ Memory Read (`memread = 1`)

- Reads data from `data_memory` at the address specified by `(alu_result[7:0] >> 3)`.
- The **lower 8 bits** of `alu_result` determine the **word-aligned address**.

2️⃣ Memory Write (`memwrite = 1`)

- Stores `rs2` into `data_memory` at the address specified by `(alu_result[7:0] >> 3)`.
- The write operation happens **asynchronously**, which may cause issues in actual hardware.

```
[Running] memory_tb.v
Read Data from Address 0:                    0
Read Data from Address 16:                   42
Read Data from Address 16 after Overwrite:
memory_tb.v:72: $finish called at 70000 (1ps)
Read Data from Uninitialized Address 200:
[Done] exit with code=0 in 0.06 seconds
```

Design Description of the `writeback` Module

The `writeback` module is responsible for selecting the final value that will be written back to the **register file** in a processor pipeline. It determines whether the **ALU result** or **memory data** should be written based on control signals.

🔷 Module Functionality

1️⃣ Inputs

Signal	Width	Description
`MemtoReg_reg`	1-bit	Control signal: **1** selects `mem_data` for register write-back, **0** selects `ex_alu_result`
`Regwrite_reg`	1-bit	Control signal: **1** enables writing back to the register
`ex_alu_result`	64-bit	Computed result from the ALU in the EX stage
`mem_data`	64-bit	Data read from memory in the MEM stage

2️⃣ Output

Signal	Width	Description
`registerout`	64-bit	The final value to be written back to the register file

---

🔷 Behavior

1️⃣ Memory Write-back (`MemtoReg_reg = 1`)

- If `MemtoReg_reg` is set, the module outputs `mem_data`, meaning the **data from memory** is written to the register file.

2️⃣ ALU Result Write-back (`MemtoReg_reg = 0` and `Regwrite_reg = 1`)

- If `MemtoReg_reg` is **not set**, and `Regwrite_reg = 1`, the module outputs `ex_alu_result`, meaning the **ALU result** is written to the register file.

3️⃣ Default Case (No Write-back)

- If neither condition is met, `registerout` is **set to** `0`, ensuring no unintended writes.

```
[Running] write_back_tb.v
TC1: registerout = 0000000000000000 (Expected: 0000000000000000)
TC2: registerout = 5a5a5a5a5a5a5a5a (Expected: 5A5A5A5A5A5A5A5A)
TC3: registerout = a5a5a5a5a5a5a5a5 (Expected: A5A5A5A5A5A5A5A5)
TC4: registerout = 5a5a5a5a5a5a5a5a (Expected: 5A5A5A5A5A5A5A5A)
write_back_tb.v:49: $finish called at 40000 (1ps)
[Done] exit with code=0 in 0.077 seconds
```

Design Description of the `processor` Module

The `processor` module implements a **basic 64-bit RISC-V processor sequential** , covering **instruction fetch, decode, execution, memory, and write-back stages**. It manages a **program counter (PC), register file, and control signals** to execute instructions sequentially while handling memory operations and branch conditions.

🔷 Module Overview

Component	Functionality
`instruction_fetch`	Fetches 32-bit instructions from memory
`decode`	Decodes instructions and extracts control signals

`ex_stage` (Execute)	Performs ALU operations based on decoded instruction
`memory`	Reads/writes data memory based on ALU result
`writeback`	Selects the final value to be written back to registers
Program Counter (PC)	Tracks instruction sequence and updates on branches
Register File	Stores general-purpose 64-bit registers (`x0` to `x31`)

The processor follows a **5-stage pipeline** with individual modules performing each stage's operations.

1️⃣ Instruction Fetch (`instruction_fetch`)

- Uses the **PC (`pc`)** to fetch a 32-bit instruction (`if_instruction`).
- Increments PC by **4** unless a branch occurs.
- Handles termination when encountering a **halt instruction (`opcode == 7'b1111111`)**.

2️⃣ Instruction Decode (`decode`)

- Decodes the **instruction format** and extracts:
  - **Register addresses** (`id_rs1`, `id_rs2`, `id_rd`).
  - **Control signals** (`branch`, `alu_op`, `memread`, `memwrite`, `memtoreg`, `regwrite`).
  - **Immediate values (`imm`)** for arithmetic and branching.

3️⃣ Execute (`ex_stage`)

- **Performs ALU operations** based on control signals.
- Selects between **register value (`rs2`) or immediate (`imm`)** as the second operand.
- Computes the **ALU result (`ex_alu_result`)**.
- Detects **overflow**.

4️⃣ Memory Access (`memory`)

- If `Memread = 1`, **reads memory** at `ex_alu_result` and stores it in `mem_data`.
- If `Memwrite = 1`, **writes `rs2` into memory** at `ex_alu_result`.

5️⃣ Writeback (`writeback`)

- If `MemtoReg = 1`, selects **memory data (`mem_data`)** for writing back.
- Otherwise, writes back **ALU result (`ex_alu_result`)**.
- Writes final value to **destination register (`id_rd`)**.

🔷 Register File

- **32 general-purpose 64-bit registers (`register_file [0:31]`)**.
- `id_rs1_val` and `id_rs2_val` are assigned from `register_file`.
- Register write (`id_rd = registerout1`) happens after the **writeback stage**.

🔷 Program Counter (PC) Management

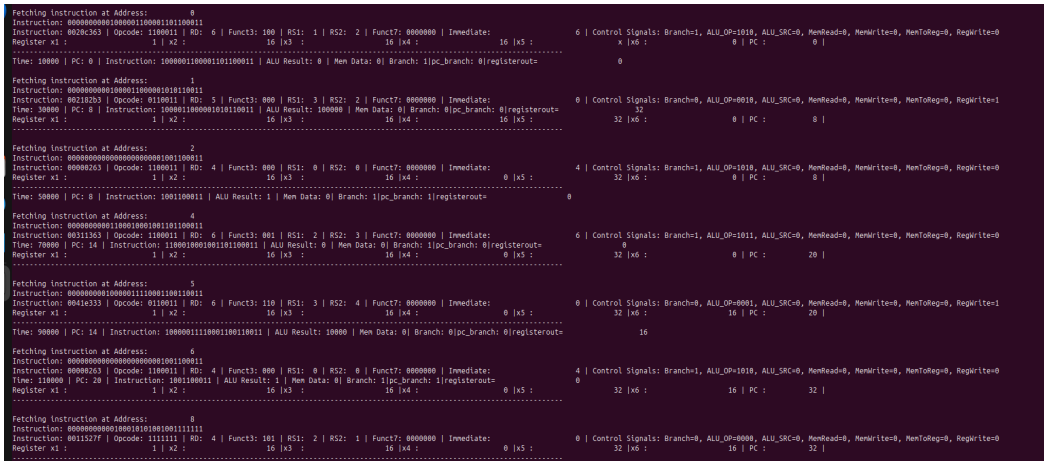
- **Default PC increment:** `pc = pc + 4` (sequential execution).
- **Branch Handling:**

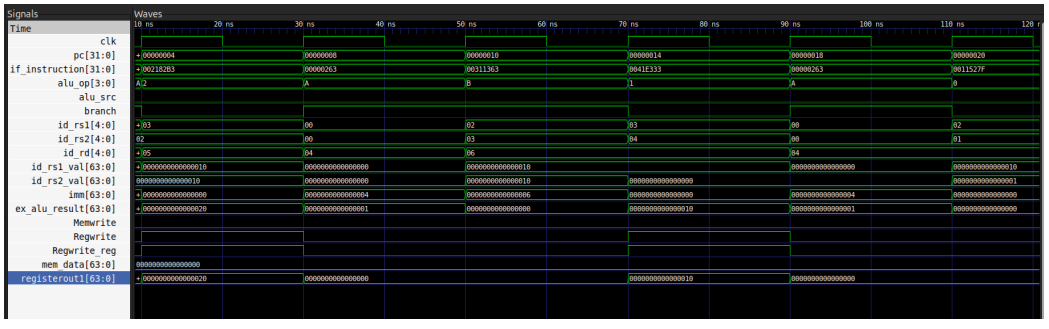
- `pc_up = branch & ex_alu_result[0]`
- If `pc_up = 1`, `pc = pc + (imm << 1)`.
- Uses the ALU result to determine branch execution.

TEST CASES:

assembly code : for simpler instruction

```
    beq x1, x2, label1    # if (x1 == x2) jump to label1 (inst_mem[3])
    add x5, x3, x2        # x5 = x3 + x2
    beq x0, x0, exit      # Unconditional jump to exit
label1:
    sub x5, x3, x2        # x5 = x3 - x2

    bne x2, x3, label2    # if (x2 != x3) jump to label2 (inst_mem[7])
    or x6, x3, x4         # x6 = x3 | x4
    beq x0, x0, exit      # Unconditional jump to exit
label2:
    and x6, x3, x2        # x6 = x3 & x2

exit:
    exit              # Infinite loop (halt execution)
```



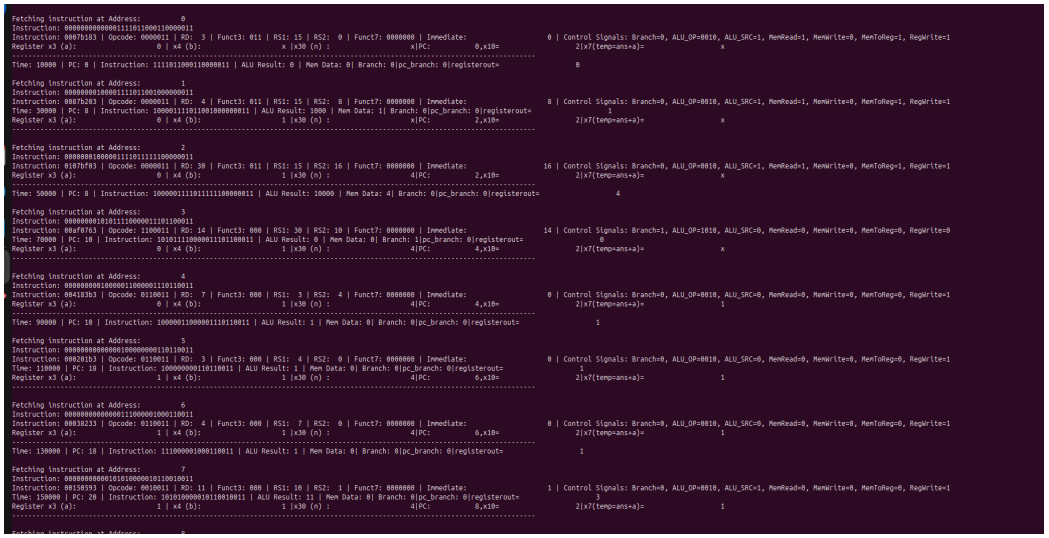


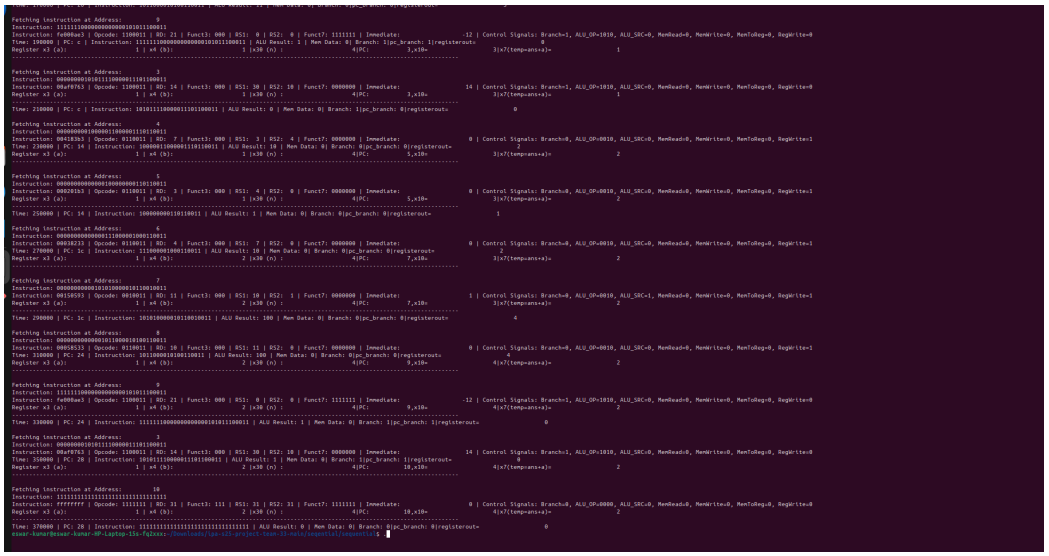
2nd Test case

Fibananaco

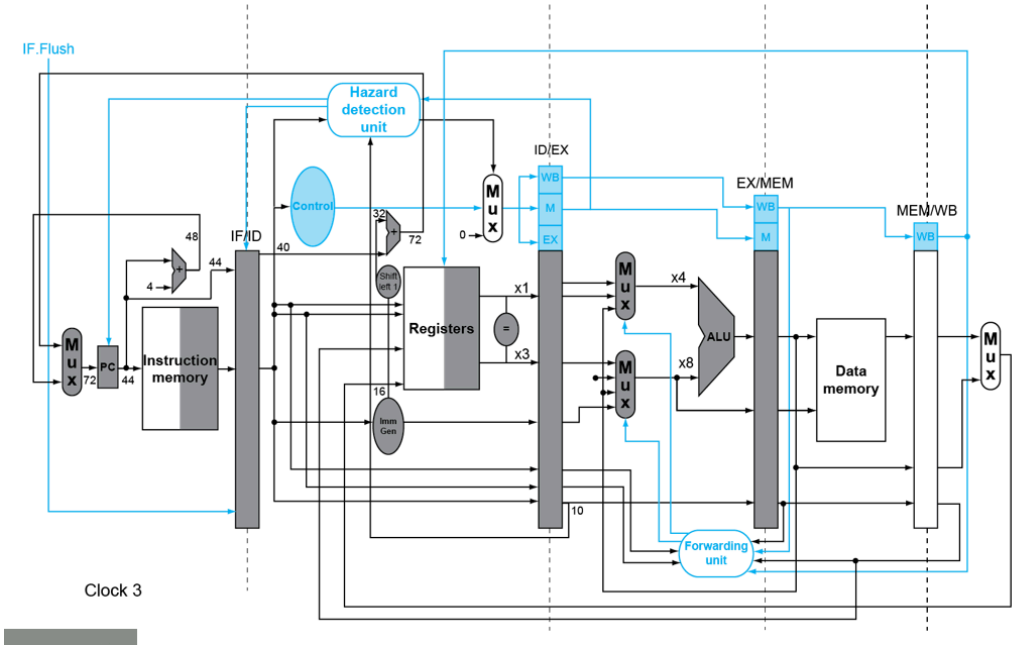
```
    ld x3, 0(x15)      # inst_mem[0]  | Load value from address x15 into x3
    ld x4, 16(x15)     # inst_mem[1]  | Load value from address (x15 + 16) into x4
```

```
    ld x30, 32(x15)      # inst_mem[2]  | Load value from address (x15 + 32) into x30

    beq x10, x30, EXIT   # inst_mem[3]  | Branch to EXIT if x10 == x30
    add x7, x3, x4       # inst_mem[4]  | x7 = x3 + x4
    add x3, x4, x0       # inst_mem[5]  | x3 = x4
    add x4, x7, x0       # inst_mem[6]  | x4 = x7
    addi x11, x10, 1     # inst_mem[7]  | x11 = x10 + 1
    add x10, x11, x0     # inst_mem[8]  | x10 = x11

    beq x0, x0, -6       # inst_mem[9]  | Unconditional jump to loop (-6 instructions back)

EXIT:  0xFFFFFFFF        # inst_mem[10] | Exit instruction (not a real RISC-V instruction)
```





Pipeline Design



Changes Made in the Pipelined Version:

Instruction Fetch Stage:

New Input Signals:

Signal Name	Type	Purpose
`pc_write`	Input	Controls whether the PC should update (hazard handling)
`clk`	Input	Synchronizes pipeline stages with clock

Added Clock Synchronization (`clk`):

- In the sequential version, the `always @(*)` block fetched instructions asynchronously.
- The pipelined version does not explicitly use `clk` in `always @(*)` , but the presence of `pc_write` suggests that control logic is used to stall or update the instruction fetch stage.

Instruction Decode Stage:

2.1 Addition of Pipeline Registers

- Introduced new outputs such as `id_ex_MemRead`, `id_ex_MemtoReg`, `id_ex_Regwrite`, `id_ex_Memwrite`, `id_ex_write`, `id_ex_rs1`, `id_ex_rs2`, and `id_ex_rd` etc to hold values for the next stage.
- These registers ensure that control signals and operands are passed correctly to the execution stage.

2.2 Hazard Detection Mechanism

- Added inputs:
  - `id_ex_MemRead`: Used to check if a load instruction in the previous stage could cause a data hazard.
  - `id_ex_Rd`, `if_id_Rs1`, `if_id_Rs2`: Used to identify read-after-write (RAW) dependencies.
  - `hazard_detected`: Control signal indicating the presence of a hazard.
- Added control signals:
  - `PCWrite`, `IF_ID_Write`: These determine whether the program counter and instruction fetch registers should be updated or stalled to prevent hazards.

2.3 Control Signal Gating Using `Control` MUX

- Introduced a control signal (`Control`) that is disabled when a hazard is detected.
- If a hazard is present, `PCWrite` and `IF_ID_Write` are disabled, preventing erroneous instruction execution.

The pipelined version of the decode stage improves performance by reducing stalls and ensuring proper data forwarding. The introduction of hazard detection and control gating mechanisms makes the pipeline more robust and efficient. These modifications are crucial for achieving higher throughput in the RISC-V processor design.

```
[Running] decode_tb.v
PCWrite=1, IF_ID_Write=1, id_ex_Memread=0, id_ex_MemtoReg=0, id_ex_Regwrite=1, id_ex_Memwrite=0, id_ex_write=0, id_ex_rs1= 2, id_ex_rs2= 1, id_ex_rd= 3,
opcode=0110011, alu_src=0, alu_op=0010, branch=0, imm=                   0
PCWrite=1, IF_ID_Write=1, id_ex_Memread=0, id_ex_MemtoReg=0, id_ex_Regwrite=1, id_ex_Memwrite=0, id_ex_write=0, id_ex_rs1= 1, id_ex_rs2= 5, id_ex_rd= 3,
opcode=0010011, alu_src=1, alu_op=0010, branch=0, imm=                   5
PCWrite=1, IF_ID_Write=1, id_ex_Memread=1, id_ex_MemtoReg=1, id_ex_Regwrite=1, id_ex_Memwrite=0, id_ex_write=0, id_ex_rs1= 1, id_ex_rs2= 4, id_ex_rd= 3,
opcode=0000011, alu_src=1, alu_op=0010, branch=0, imm=                   4
PCWrite=1, IF_ID_Write=1, id_ex_Memread=0, id_ex_MemtoReg=0, id_ex_Regwrite=0, id_ex_Memwrite=1, id_ex_write=0, id_ex_rs1= 1, id_ex_rs2= 3, id_ex_rd= 4,
opcode=0100011, alu_src=1, alu_op=0010, branch=0, imm=                   4
PCWrite=1, IF_ID_Write=1, id_ex_Memread=0, id_ex_MemtoReg=0, id_ex_Regwrite=0, id_ex_Memwrite=0, id_ex_write=0, id_ex_rs1= 2, id_ex_rs2= 1, id_ex_rd= 3,
opcode=1100011, alu_src=0, alu_op=1010, branch=1, imm=                2050
PCWrite=0, IF_ID_Write=0, id_ex_Memread=0, id_ex_MemtoReg=0, id_ex_Regwrite=0, id_ex_Memwrite=0, id_ex_write=0, id_ex_rs1= 2, id_ex_rs2= 1, id_ex_rd= 3,
opcode=1100011, alu_src=0, alu_op=0000, branch=0, imm=                   0
decode_tb.v:94: $finish called at 60000 (1ps)
PCWrite=1, IF_ID_Write=1, id_ex_Memread=0, id_ex_MemtoReg=0, id_ex_Regwrite=0, id_ex_Memwrite=0, id_ex_write=0, id_ex_rs1= 2, id_ex_rs2= 1, id_ex_rd= 3,
opcode=1100011, alu_src=0, alu_op=1010, branch=1, imm=                2050
[Done] exit with code=0 in 0.051 seconds
```

Execute Stage:

(A) Operand Handling

Sequential Code:

- Uses `rs1_val` and `rs2_val` directly from the decoder.
- Immediate value (`imm`) is selected based on `alu_src`.

Pipelined Code:

- `operandA` and `operandB` replace `rs1_val` and `rs2_val`.
- `operandB` is determined using `alu_src` in ID/EX stage to support immediate-based instructions.

Reason for Change:

- The sequential version directly uses register values.
- The pipeline introduces `ID/EX` pipeline registers that pass operands from the decode stage to ensure correct data forwarding.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

(B) Pipeline Registers for Data Transfer

Sequential Code:

- Outputs results (`alu_result`, `overflow`) immediately.

Pipelined Code:

- Introduces **EX/MEM pipeline registers** (`ex_mem_*` signals) to store computed ALU results and control signals for the memory stage.

New Signals Introduced:

Register Name	Bit-width	Description
`ex_mem_alu_result`	64 (signed)	Stores the result of the ALU operation from the Execute stage to be forwarded to the Memory stage.
`ex_mem_rd`	5	Stores the destination register address for the instruction being executed.
`ex_mem_Memwrite`	1	Control signal indicating whether the instruction writes data to memory (`1` = Write, `0` = No write).
`ex_mem_Memread`	1	Control signal indicating whether the instruction reads data from memory (`1` = Read, `0` = No read).
`ex_mem_MemtoReg`	1	Control signal determining if the data from memory should be written back to the register (`1` = Yes, `0` = No).
`ex_mem_Regwrite`	1	Control signal indicating whether the instruction writes data to a register (`1` = Write, `0` = No write).
`ex_mem_overflow`	1	Flag indicating if an overflow occurred during the ALU operation (`1` = Overflow, `0` = No overflow).

Reason for Change:

- In pipelined execution, each stage needs to pass values forward to the next stage using pipeline registers.

(C) ALU Control Handling

Sequential Code:

- Uses an **always @(*) block** to select the ALU operation based on `alu_op`.
- Directly updates `alu_result`.

Pipelined Code:

- Uses **non-blocking assignments (`<=`)** to ensure proper pipeline execution.

- Stores ALU results into `ex_mem_alu_result` instead of updating immediately.

Reason for Change:

- The pipeline needs to prevent hazards by **buffering values** in `ex_mem_*` registers before forwarding them.

(D) Memory and Writeback Control Signals

Sequential Code:

- No explicit memory-related control signals.

Pipelined Code:

- Introduces `id_ex_Memwrite`, `id_ex_Memread`, `id_ex_MemtoReg`, `id_ex_Regwrite`, and forwards them to `ex_mem_*` registers.

Reason for Change:

- Ensures that memory read/write operations happen at the correct cycle in the **Memory (MEM) stage**.

Signal Name	Purpose
`operandA, operandB`	Inputs to ALU from pipeline registers
`ex_mem_alu_result`	Stores ALU result for the MEM stage
`ex_mem_rd`	Stores destination register for WB stage
`ex_mem_Memwrite, ex_mem_Memread, ex_mem_MemtoReg, ex_mem_Regwrite`	Control signals for memory and writeback stages
`ex_mem_overflow`	Stores overflow flag from ALU

```
[Running] exstage_tb.v
Time=0 | ALU_OP=0010 | OperandA=                  10 | OperandB=                   5 | Result=                  15 | Overflow=0
Time=10000 | ALU_OP=0110 | OperandA=                10 | OperandB=                   5 | Result=                   5 | Overflow=0
Time=20000 | ALU_OP=0000 | OperandA=                10 | OperandB=                   5 | Result=                   0 | Overflow=0
Time=30000 | ALU_OP=0001 | OperandA=                10 | OperandB=                   5 | Result=                  15 | Overflow=0
Time=40000 | ALU_OP=1001 | OperandA=                10 | OperandB=                   5 | Result=                  15 | Overflow=0
Time=50000 | ALU_OP=0011 | OperandA=                10 | OperandB=                   5 | Result=                 320 | Overflow=0
Time=60000 | ALU_OP=0100 | OperandA=                10 | OperandB=                   5 | Result=                   0 | Overflow=0
Time=70000 | ALU_OP=0101 | OperandA=                10 | OperandB=                   5 | Result=                   0 | Overflow=0
Time=80000 | ALU_OP=0111 | OperandA=                10 | OperandB=                   5 | Result=                   0 | Overflow=0
Time=90000 | ALU_OP=1000 | OperandA=                10 | OperandB=                   5 | Result=                   0 | Overflow=0
Time=100000 | ALU_OP=1010 | OperandA=                10 | OperandB=                   5 | Result=                   0 | Overflow=0
Time=110000 | ALU_OP=1011 | OperandA=                10 | OperandB=                   5 | Result=                   1 | Overflow=0
Time=120000 | ALU_OP=0010 | OperandA= 9223372036854775807 | OperandB=                   5 | Result=-9223372036854775804 | Overflow=1
exstage_tb.v:123: $finish called at 140000 (1ps)
[Done] exit with code=0 in 0.659 seconds
```

Memory Stage:

Changes in Memory Read and Write Operations

- **Memory Read (`mem_wb_mem_data`)**
  - In the sequential design, `mem_data` is assigned directly based on `memread`.
  - In the pipelined design, `mem_wb_mem_data` is assigned conditionally from `data_memory[ex_mem_alu_result[7:0]]` when `ex_mem_Memread` is high.
- **Memory Write (`data_memory[ex_mem_alu_result[7:0]] <= ex_mem_rs2`)**
  - Instead of using `memwrite` from the sequential design, the pipelined version now uses `ex_mem_Memwrite` to check if memory should be written.

Reason for the Change:

- **Separation of Memory Read and Write:** In the pipeline, the memory stage must handle both operations without interfering with other stages.
- **Control Signal Dependency:** The new approach ensures that memory writes happen only when explicitly enabled by control signals passed from the EX stage.

Removal of Sequential Control and Direct Memory Access

- In the sequential design, `mem_data` was updated synchronously within an `always @(*)` block.
- The pipelined version ensures that memory operations are only carried out based on control signals from the **EX/MEM pipeline registers**.

Reason for the Change:

- **Pipeline Stages Must Operate Independently:** The memory stage should only execute memory operations when the EX stage has issued the appropriate control signals.
- **Ensuring Data Flow Continuity:** Instead of direct memory access, data is now forwarded through pipeline registers.

Summary of New Signals (Wires and Registers)

Signal Name	Type	Purpose
`mem_wb_rd`	Reg	Holds the destination register from MEM to WB stage.
`mem_wb_MemtoReg`	Reg	Control signal to determine if memory data or ALU result is written back.
`mem_wb_alu_result`	Reg	Holds ALU result for the WB stage.
`mem_wb_mem_data`	Reg	Holds memory read data for the WB stage.
`mem_wb_RegWrite`	Reg	Control signal indicating if instruction writes back to a register.
`ex_mem_rd`	Input	Destination register from EX stage.
`ex_mem_Memwrite`	Input	Control signal indicating a memory write operation.
`ex_mem_Memread`	Input	Control signal indicating a memory read operation.
`ex_mem_MemtoReg`	Input	Control signal determining if memory data or ALU result is written back.
`ex_mem_Regwrite`	Input	Control signal for register write-back.
`ex_mem_alu_result`	Input	ALU result from EX stage.
`ex_mem_rs2`	Input	Register value used for memory write operations.

```
[Running] memory_tb.v
Memory Values:                  10,                 90,                  6
Time=0 | clk=0 | MemWrite=0 | MemRead=1 | MemtoReg=0 | RegWrite=0 | rd= 0 | ALU_Result=                2 | MemData=                6
Time=5000 | clk=1 | MemWrite=0 | MemRead=1 | MemtoReg=0 | RegWrite=0 | rd= 0 | ALU_Result=                2 | MemData=                6
Memory Values:                  10,                 90,                  6
Memory Values:                  10,                 90,                 42
Time=10000 | clk=0 | MemWrite=1 | MemRead=0 | MemtoReg=0 | RegWrite=0 | rd= 0 | ALU_Result=                2 | MemData=                0
Time=15000 | clk=1 | MemWrite=1 | MemRead=0 | MemtoReg=0 | RegWrite=0 | rd= 0 | ALU_Result=                2 | MemData=                0
Memory Values:                  10,                 90,                 42
Time=20000 | clk=0 | MemWrite=0 | MemRead=0 | MemtoReg=0 | RegWrite=0 | rd= 0 | ALU_Result=                2 | MemData=                0
Time=25000 | clk=1 | MemWrite=0 | MemRead=0 | MemtoReg=0 | RegWrite=0 | rd= 0 | ALU_Result=                2 | MemData=                0
memory_tb.v:68: $finish called at 30000 (1ps)
Time=30000 | clk=0 | MemWrite=0 | MemRead=0 | MemtoReg=0 | RegWrite=0 | rd= 0 | ALU_Result=                2 | MemData=                0
[Done] exit with code=0 in 0.053 seconds
```

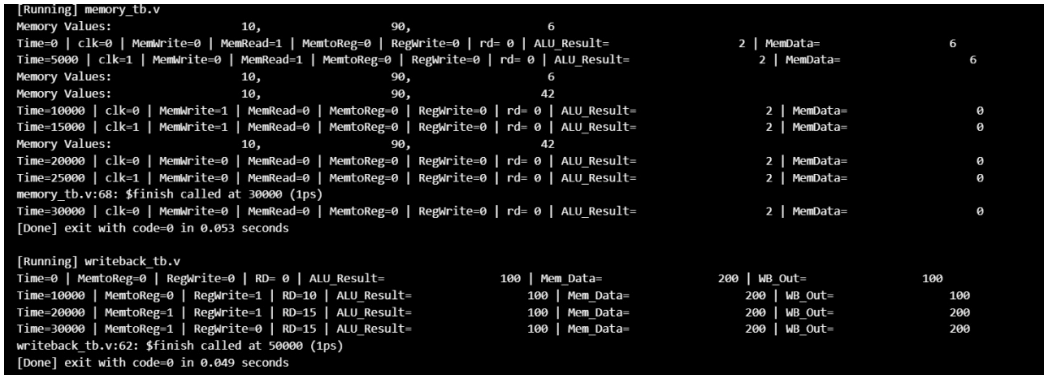
Write Back Stage:

- **Selection Between ALU Result and Memory Data**
  - In the sequential version, `registerout` was determined using `MemtoReg_reg` and `Regwrite_reg`.
  - In the pipelined version, `wb_registerout` is selected using `mem_wb_MemtoReg`, ensuring it comes from either memory (`mem_wb_mem_data`) or ALU (`mem_wb_alu_result`).
- **Register Write Enable Forwarding**
  - The `RegWrite` control signal (`wb_RegWrite`) is explicitly forwarded from the MEM/WB stage to the WB stage.

Reason for the Change:

- **Separating Control Logic from Data Flow** ensures better **pipeline efficiency** and eliminates dependency on earlier stages.

Signal Name	Type	Purpose
wb_RegWrite	Reg	Holds the write enable signal for register file.
wb_rd	Reg	Holds the destination register number for the WB stage.
wb_registerout	Reg	Final data value written to the register file.
mem_wb_MemtoReg	Input	Determines if data comes from memory or ALU.
mem_wb_RegWrite	Input	Write enable signal from MEM/WB.
mem_wb_rd	Input	Destination register from MEM/WB.
mem_wb_alu_result	Input	ALU result from MEM/WB stage.
mem_wb_mem_data	Input	Memory read data from MEM/WB stage.



Hazard Module

Design Description of the Hazard Detection Module

The **Hazard Detection Unit** is a crucial part of a pipelined processor. It is responsible for detecting **load-use hazards**, which occur when an instruction in the **ID/EX** stage is reading from memory (`MemRead = 1`), and the next instruction in the **IF/ID** stage needs the result of that memory load.

Inputs & Outputs

Signal	Width	Type	Description
ID_EX_MemRead	1-bit	Input	Indicates whether the instruction in the **ID/EX** stage is performing a memory read.
ID_EX_Rd	5-bit	Input	Destination register (`Rd`) of the instruction in the **ID/EX** stage.
IF_ID_Rs1	5-bit	Input	First source register (`Rs1`) of the instruction in the **IF/ID** stage.
IF_ID_Rs2	5-bit	Input	Second source register (`Rs2`) of the instruction in the **IF/ID** stage.
PCWrite	1-bit	Output	Controls whether the **Program Counter (PC)** should update. (`0` = Stall, `1` = Normal execution).
IF_ID_Write	1-bit	Output	Controls whether the **IF/ID** register should update. (`0` = Stall, `1` = Normal execution).
ControlMux	1-bit	Output	Determines whether the control signals for the current instruction should be stalled. (`1` = Stall, `0` = Normal execution).

Hazard Detection Logic

The hazard detection module stalls the pipeline if:

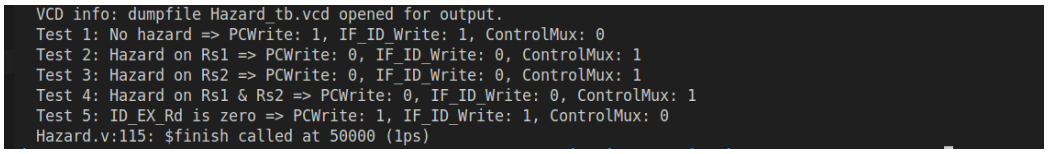
1. **A load instruction (`MemRead = 1`) is in the ID/EX stage.**
2. **The destination register (`ID_EX_Rd`) of that load instruction matches the source register (`Rs1` or `Rs2`) of the next instruction in the IF/ID stage.**

3. **The destination register is not x0 (`ID_EX_Rd ≠ 0`).**

Pipeline Stalling Example

Clock Cycle	Instruction in IF/ID Stage	Instruction in ID/EX Stage	Hazard?	Action Taken
1	`LD x2, 0(x3)`	-	No	Normal Execution
2	`ADD x5, x2, x4`	`LD x2, 0(x3)`	Yes	Stall Pipeline
3	`ADD x5, x2, x4`	`LD x2, 0(x3)` (stall)	Yes	Stall Pipeline
4	`ADD x5, x2, x4`	`LD x2, 0(x3)` (finished)	No	Resume Execution

- The pipeline is **stalled for one cycle** to allow `LD x2, 0(x3)` to complete, ensuring correct execution of `ADD x5, x2, x4`.



Design Description of the Forwarding Unit (`forwarding_unit`)

🔷 Overview

The **Forwarding Unit** is a critical component of a **pipelined processor** that helps resolve **data hazards** by forwarding data from later pipeline stages (**EX/MEM** and **MEM/WB**) back to the **EX stage**, preventing the need for stalls in certain cases.

In **RISC-V**, data hazards occur when an instruction depends on the result of a previous instruction that has not yet completed writing back to the register file. The forwarding unit detects these hazards and **bypasses the correct value** from the later pipeline stages to the ALU inputs (`ForwardA` and `ForwardB`).

🔷 Inputs & Outputs

Signal	Width	Direction	Description
`EX_MEM_RegWrite`	1-bit	Input	Indicates if the instruction in **EX/MEM** stage writes to a register.
`MEM_WB_RegWrite`	1-bit	Input	Indicates if the instruction in **MEM/WB** stage writes to a register.
`EX_MEM_Rd`	5-bit	Input	Destination register (`Rd`) of the instruction in **EX/MEM** stage.
`MEM_WB_Rd`	5-bit	Input	Destination register (`Rd`) of the instruction in **MEM/WB** stage.
`ID_EX_Rs1`	5-bit	Input	Source register 1 (`Rs1`) of the instruction in **ID/EX** stage.
`ID_EX_Rs2`	5-bit	Input	Source register 2 (`Rs2`) of the instruction in **ID/EX** stage.
`ForwardA`	2-bit	Output	Control signal for **ALU input A** (selects forwarded data from **EX/MEM** or **MEM/WB**).
`ForwardB`	2-bit	Output	Control signal for **ALU input B** (selects forwarded data from **EX/MEM** or **MEM/WB**).

🔷 Forwarding Logic

The forwarding unit checks whether the **destination register (`Rd`) of a previous instruction** matches the **source registers (`Rs1` or `Rs2`) of the current instruction** in the **ID/EX** stage.

If a match is found and **the previous instruction writes to a register**, the unit **forwards** the data to prevent stalls.

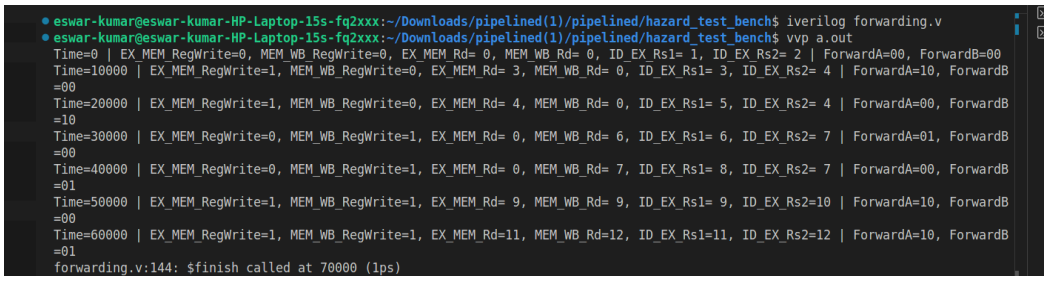
Forwarding Conditions:

1. **Forward from EX/MEM stage (`ForwardA = 10` or `ForwardB = 10`)**
   - The **EX/MEM instruction writes (`EX_MEM_RegWrite = 1`)**
   - The **destination register (`EX_MEM_Rd`) matches the source register (`ID_EX_Rs1` or `ID_EX_Rs2`)**
   - `EX_MEM_Rd` is **not register x0** (to avoid forwarding invalid data).
2. **Forward from MEM/WB stage (`ForwardA = 01` or `ForwardB = 01`)**
   - The **MEM/WB instruction writes (`MEM_WB_RegWrite = 1`)**
   - The **destination register (`MEM_WB_Rd`) matches the source register (`ID_EX_Rs1` or `ID_EX_Rs2`)**
   - `MEM_WB_Rd` is **not register x0** (to avoid forwarding invalid data).
3. **No Forwarding (`ForwardA = 00`, `ForwardB = 00`)**
   - If no match is found, the ALU receives the **original register values** from the register file.

🔷 Example Forwarding Scenarios

Cycle	Instruction in ID/EX	Instruction in EX/MEM	Instruction in MEM/WB	ForwardA	ForwardB
1	`add x5, x1, x2`	-	-	00	00
2	`sub x6, x5, x3`	`add x5, x1, x2`	-	10	00
3	`or x7, x6, x4`	`sub x6, x5, x3`	`add x5, x1, x2`	10	01

- **Cycle 2:** `x5` is needed, but the `add` instruction is in EX/MEM → `ForwardA = 10`.
- **Cycle 3:** `x6` is needed, but the `sub` instruction is in EX/MEM, while `x5` is in MEM/WB → `ForwardA = 10`, `ForwardB = 01`.



1. Stages Separated into Pipeline Registers

- In the sequential design, all operations (Fetch → Decode → Execute → Memory → Writeback) were happening in a single clock cycle.
- In the pipelined version, each stage now has its own **pipeline register** that stores intermediate results between stages.
- This separation allows overlapping of instruction execution, significantly improving performance.

---

2. Introduction of Pipeline Registers

Pipeline registers are **special registers** introduced between stages to hold the intermediate results. They are updated on the **positive edge of the clock (posedge clk)**.

Here are the pipeline registers introduced:

Pipeline Register	Purpose
IF/ID	Holds instruction fetched from memory (IF stage)
ID/EX	Holds decoded instruction data (rs1, rs2, rd, imm, control signals)
EX/MEM	Holds ALU result, memory write data, control signals
MEM/WB	Holds memory read data or ALU result to be written back to the register file

3. Pipeline Stalls and Hazard Detection

🚨 Problem: Data Hazard

- In sequential design, if **one instruction depends on the result of a previous instruction**, the processor would give the wrong result.
- Example:

```
add x1, x2, x3
sub x4, x1, x5  // x1 value from previous instruction not available yet
```

- To solve this:
  - **Hazard detection unit** was introduced to detect data hazards.
  - **Pipeline stalls** were introduced when a hazard is detected.

✅ Signals Introduced in Pipeline Processor

Here is a summary of the major signals introduced:

Signal Name	Purpose
pc_write	Controls whether the PC (Program Counter) should update or stall.
if_id_write	Controls whether the IF/ID register should update or stall.
forwardA, **forwardB**	Forwarding control signals to select the correct value for ALU inputs (prevent hazard).
hazard_detected	This signal is raised when a data hazard occurs (stall the pipeline).
MemtoReg	Determines if the value written back to the register is from memory or ALU.
RegWrite	Controls if the register should be written with a new value.
id_ex_MemRead, id_ex_MemWrite, id_ex_MemtoReg, id_ex_Regwrite	Control signals from Decode to Execute stage.
ex_mem_MemRead, ex_mem_MemWrite, ex_mem_MemtoReg, ex_mem_Regwrite	Control signals from Execute to Memory stage.
mem_wb_RegWrite, mem_wb_MemtoReg	Control signals from Memory to Writeback stage.

✅ 4. How Pipeline Registers Are Updated (Data Flow)

Let's break down how the pipeline registers are used and updated:

Stage 1: Instruction Fetch (IF)

Purpose: Fetch the instruction from instruction memory.

Pipeline Register: IF/ID

Field Name	Purpose
if_id_instruction	Holds the instruction fetched.
if_id_rs1	Holds source register 1.
if_id_rs2	Holds source register 2.
if_id_rd	Holds destination register.

How it Updates:

```
always @(posedge clk) begin
    if (if_id_write) begin
        if_id_instruction <= if_instruction;
        if_id_rs1 <= if_instruction[19:15];
        if_id_rs2 <= if_instruction[24:20];
        if_id_rd  <= if_instruction[11:7];
    end
end
```

- The instruction and register addresses are **passed to the next stage** via the IF/ID register.

Stage 2: Instruction Decode (ID)

Purpose: Decode the instruction and generate control signals.

Pipeline Register: ID/EX

Field Name	Purpose
id_ex_rs1_val	Value of register rs1.
id_ex_rs2_val	Value of register rs2.
id_ex_imm	Immediate value (sign-extended).
id_ex_rd	Destination register.
Control Signals	MemRead, MemWrite, ALUOp, RegWrite, etc.

How it Updates:

```
always @(posedge clk) begin
    id_ex_rs1_val <= register_file[id_ex_rs1];
    id_ex_rs2_val <= register_file[id_ex_rs2];
    id_ex_imm <= id_ex_imm_wire;
    id_ex_MemRead <= id_ex_MemRead_wire;
    id_ex_RegWrite <= id_ex_RegWrite_wire;
end
```

- The control signals and operands are passed to the **Execute stage.**

Stage 3: Execute (EX)

Purpose: Perform ALU operations or branch calculations.

Pipeline Register: EX/MEM

Field Name	Purpose
ex_mem_alu_result	Result from ALU.
ex_mem_rs2	Value of rs2 (for memory writes).
ex_mem_rd	Destination register.
Control Signals	MemRead, MemWrite, RegWrite, MemtoReg.

How it Updates:

```
always @(posedge clk) begin
    ex_mem_alu_result <= ex_mem_alu_result_wire;
    ex_mem_rd <= ex_mem_rd_wire;
    ex_mem_MemWrite <= ex_mem_MemWrite_wire;
end
```

- The ALU result and control signals are passed to the **Memory stage.**

Stage 4: Memory (MEM)

Purpose: Read/Write from memory.

Pipeline Register: MEM/WB

Field Name	Purpose
mem_wb_mem_data	Data read from memory (if MemRead=1).
mem_wb_alu_result	ALU result (for non-memory instructions).
mem_wb_rd	Destination register.
Control Signals	RegWrite, MemtoReg.

How it Updates:

```
always @(posedge clk) begin
    mem_wb_mem_data <= mem_wb_mem_data_wire;
    mem_wb_rd <= mem_wb_rd_wire;
    mem_wb_RegWrite <= mem_wb_RegWrite_wire;
end
```

- The data is passed to the **Writeback stage.**

Stage 5: Writeback (WB)

Purpose: Write the result back to the register file.

Write Logic:

```
always @(*) begin
    if (wb_RegWrite) begin
```

```
        register_file[wb_rd] = wb_registerout;
    end
end
```

- If `wb_RegWrite` is high, the result from ALU/memory is written back.

✅ 5. How Forwarding Works (Forwarding Unit)

🚀 Problem: Data Hazard

- Example:

```
add x1, x2, x3
sub x4, x1, x5
```

- The value of `x1` is not available in `ID` stage.

Forwarding Solution

- The **Forwarding Unit** uses:

```
assign ForwardA[1] = fwdA_EX;
assign ForwardA[0] = fwdA_MEM & ~fwdA_EX;
```

- It automatically **forwards the value** from MEM/WB or EX/MEM if needed.

✅ 6. Hazard Detection (Stall Control)

- When `MemRead` is high and `id_ex_rd` matches `if_id_rs1`, it stalls the pipeline.
- This prevents the next instruction from executing until data is ready.

Feature	Sequential Processor	Pipelined Processor
Execution Time	Slow	Faster (5x improvement)
Pipeline	No	Yes
Hazard Control	No	Yes (Forwarding + Stalling)
Control Signals	Less	More

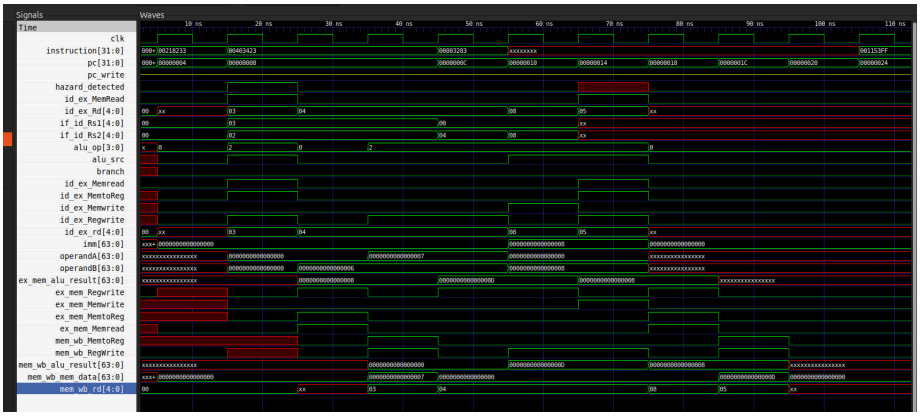
Test cases ：

```
ld x3 0(x0)
add x4 ×3 ×2
sd x4 8(x0)
ld x5  8(x0)
```

```
reg [63:0] data_memory [0:255]; // Memory array
reg  wb_write_reg;
// Initialize memory with sample values
initial begin
    data_memory[0] = 64'd7;
    data_memory[1] = 64'd34;
    data_memory[2] = 64'd6;
end
```

```
// Initialize registers manually
uut.register_file[0] = 64'd0000;
// uut.register_file[1] = 64'h0001;
uut.register_file[2] = 64'd6;
// uut.register_file[3] = 64'd14;
// uut.register_file[4] = 64'd0;
//  uut.register_file[6] = 64'd4;
// uut.register_file[7] = 64'd9;

uut.pc = 32'h0000;
```

```
 EX_MEM REGISTER DETAILS

--------------------------------------------------------------------------------------------
operandA-                x
operandB-                x
wire---operanddummy=          x ex_mem_alu_result :                  x | ex_mem_Memwrite : 0 | ex_mem_Memread : 0 | ex
_mem_MemtoReg : 0 | ex_mem_Regwrite : 0 ||
reg---ex_mem_rd :  x | ex_mem_rs2 :                  x | ex_mem_alu_result :                  x | ex_mem_Memwrite : 0 | ex_me
m_Memread : 0 | ex_mem_MemtoReg : 0 | ex_mem_Regwrite : 0 || forward A : 00 | forward B : 00
--------------------------------------------------------------------------------------------
WRITE BACK DETAILS

--------------------------------------------------------------------------------------------
wire---pc=       28|TIME : 95000  || clk : 1  || Mem_data:                  0  ||  || mem_wb_MemtoReg: 0  || mem_wb_RegWrite:
0  || rd: xxxxx
reg---pc=       28|TIME : 95000  || clk : 1  || Mem_data:                 13  ||  || mem_wb_MemtoReg: 1  || mem_wb_RegWrite: 1
  || rd: 00101
--------------------------------------------------------------------------------------------
MEM_DATA REGISTER DETAILS

--------------------------------------------------------------------------------------------
wire---mem_wb_rd :  x | mem_wb_alu_result :                  x | mem_wb_mem_data :                  0 | mem_wb_RegWrite : 0 |
 mem_wb_MemtoReg : 0
reg---mem_wb_rd :  5 | mem_wb_alu_result :                  8 | mem_wb_mem_data :                 13 | mem_wb_RegWrite : 1 |
mem_wb_MemtoReg : 1
--------------------------------------------------------------------------------------------
 TB OUTPUT---00000000000000000000000000000000000000000000000000000000000000000000000000000000
--------------------------------------------------------------------------------------------------------------------------------
-
time : 95000 ||Register x1 :                  x | x2 :                  6 | x3 :                  7 | x4 :
  13 | x5 :                 13 | x6 :                  x | x7 :                  x || PC :         28 |time=95000
--------------------------------------------------------------------------------------------------------------------------------
-
Time: 0 | clk: 1 | MemtoReg: 0 | MemRead: 0 | MemWrite: 0 | rd:  x | ALU_Result:                  x | MemData:
   0
Memory Values:                  7,                 34,                  6
Time: 0 | clk: 0 | MemtoReg: 0 | MemRead: 0 | MemWrite: 0 | rd:  x | ALU_Result:                  x | MemData:
   0
Memory Values:                  7,                 34,                  6
 TB OUTPUT---00000000000000000000000000000000000000000000000000000000000000000000000000000000
--------------------------------------------------------------------------------------------------------------------------------
```



this test case handel’s load use data hazard ,double data hazard .

ld x3 0(x0)
add x5 ×3 ×4
add x7 ×6 ×5

```
module memory (
    output reg [63:0] mem_wb_mem_data,
    output reg mem_wb_RegWrite
);

    reg [63:0] data_memory [0:255]; // Memory array
    reg  wb_write_reg;
    // Initialize memory with sample values
    initial begin
        data_memory[0] = 64'd10;
        data_memory[1] = 64'd1;
        data_memory[2] = 64'd6;
    end

    // Write operation (MEMORY WRITE)
    always @(*) begin
        //if(memwrite)begin
```

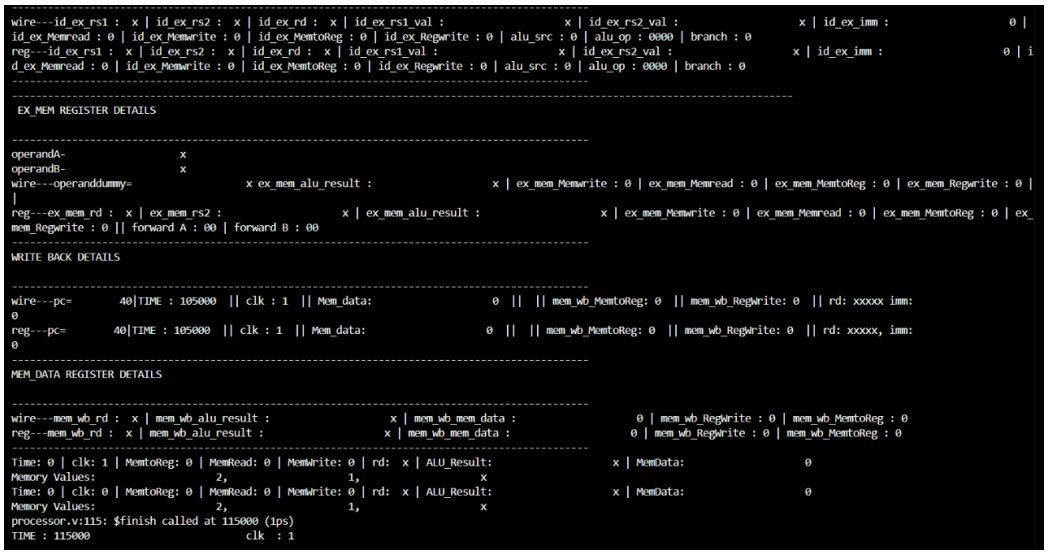
```
ocessor_tb.v > processor_tb
module processor_tb();
        clk = 1'b0;
        instr_count = 0;

        // Initialize registers manually
          // Initialize registers manually
        uut.register_file[0] = 64'd0000;
        uut.register_file[1] = 64'h0001;
        uut.register_file[6] = 64'd3;
      //  uut.register_file[3] = 64'd14;
        uut.register_file[4] = 64'd10;
        // uut.register_file[3] = 64'd14;
        // uut.register_file[4] = 64'd0;
      //  uut.register_file[6] = 64'd4;
       // uut.register_file[7] = 64'd9;

        uut.pc = 32'h0000;
```

```
Time: 0 | clk: 1 | MemtoReg: 0 | MemRead: 0 | MemWrite: 0 | rd:  7 | ALU_Result:                  23 | MemData:
   0
Memory Values:                10,                  1,                  6
Time: 0 | clk: 1 | MemtoReg: 0 | MemRead: 0 | MemWrite: 0 | rd:  x | ALU_Result:                   x | MemData:
   0
Memory Values:                10,                  1,                  6
Time: 0 | clk: 0 | MemtoReg: 0 | MemRead: 0 | MemWrite: 0 | rd:  x | ALU_Result:                   x | MemData:
   0
Memory Values:                10,                  1,                  6
 TB OUTPUT---0000000000000000000000000000000000000000000000000000000000000000000000000000
-------------------------------------------------------------------------------------------------------------------------------------------
.
time : 85000 ||Register x1 :                    1 | x2 :                    x | x3 :                   10 | x4 :
  10 | x5 :                 20 | x6 :                    3 | x7 :                   23 || PC :         24 |time=85000
-------------------------------------------------------------------------------------------------------------------------------------------
.
TIME : 85000                        clk  : 1

INSTRUCTION  FETCH DETAILS

-------------------------------------------------------------------------------------------
time=85000||PC:         24| if_instruction=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| pc_write 1
-------------------------------------------------------------------------------------------
INSTRUCTION DECODE  DETAILS

-------------------------------------------------------------------------------------------
  IF/ID REGISTER DETAILS

-------------------------------------------------------------------------------------------
register-pc=        24|Time: 85000 | if_id_instruction=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  | if_id_rs1= x | if_id_rs2= x | if_id_rd
= x | if_id_write=1
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
EXECUTION/ ALU  DETAILS

-------------------------------------------------------------------------------------------
 ID/EX  REGISTER DETAILS

-------------------------------------------------------------------------------------------
wire---id_ex_rs1 :  x | id_ex_rs2 :  x | id_ex_rd :  x | id_ex_rs1_val :                   x | id_ex_rs2_val :
  x | id_ex_imm :                  0 | id_ex_Memread : 0 | id_ex_Memwrite : 0 | id_ex_MemtoReg : 0 | id_ex_Regwrite : 0 | alu_s
rc : 0 | alu_op : 0000 | branch : 0
```

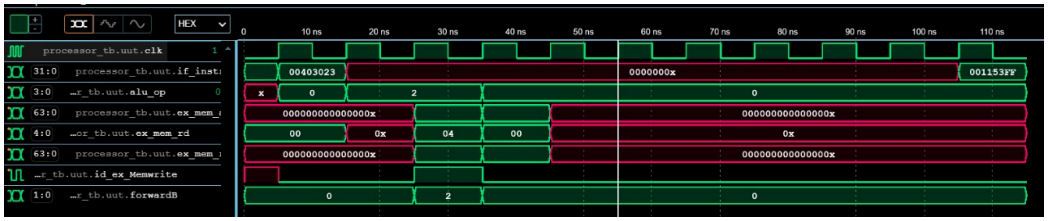
this handel’s double data hazard, load use data hazard

```
inst_mem[0] = 32'b0000000_00110_00010_000_00111_0110011; // add  x7 ×2 ×6
inst_mem[1] = 32'b000000000111_00000_011_00000_0100011; // sd x7, 0(x0)
```

```
processor_tb.v
3   module processor_tb();
30      initial begin
32          instr_count = 0;
33
34          // Initialize registers manually
35          uut.register_file[0] = 64'd0000;
36          uut.register_file[1] = 64'h0001;
37          uut.register_file[2] = 64'd0;
38          // uut.register_file[3] = 64'd14;
39          uut.register_file[5] = 64'd1;
40          uut.register_file[6] = 64'd1;
41          uut.register_file[7] = 64'd9;
42
43          uut.pc = 32'h0000;
44          uut.pc_write = 1'b1;
45          uut.if_id_write = 1'b1;
46         // uut.id_ex_write = 1'b0;
47          uut.id_ex_Memread = 1'b0;
```



```
wire---id_ex_rs1 :  x | id_ex_rs2 :  x | id_ex_rd :  x | id_ex_rs1_val :                x | id_ex_rs2_val :                x | id_ex_imm :                0 |
id_ex_Memread : 0 | id_ex_Memwrite : 0 | id_ex_MemtoReg : 0 | id_ex_Regwrite : 0 | alu_src : 0 | alu_op : 0000 | branch : 0
reg---id_ex_rs1 :  x | id_ex_rs2 :  x | id_ex_rd :  x | id_ex_rs1_val :                x | id_ex_rs2_val :                x | id_ex_imm :                0 | i
d_ex_Memread : 0 | id_ex_Memwrite : 0 | id_ex_MemtoReg : 0 | id_ex_Regwrite : 0 | alu_src : 0 | alu_op : 0000 | branch : 0
-------------------------------------------------------------------------------------------------------------------------------------------------
 EX_MEM REGISTER DETAILS

-------------------------------------------------------------------------------------------
operandA-                x
operandB-                x
wire---operanddummy=                x ex_mem_alu_result :                x | ex_mem_Memwrite : 0 | ex_mem_Memread : 0 | ex_mem_MemtoReg : 0 | ex_mem_Regwrite : 0 |
|
reg---ex_mem_rd :  x | ex_mem_rs2 :                x | ex_mem_alu_result :                x | ex_mem_Memwrite : 0 | ex_mem_Memread : 0 | ex_mem_MemtoReg : 0 | ex_
mem_Regwrite : 0 || forward A : 00 | forward B : 00
-------------------------------------------------------------------------------------------
WRITE BACK DETAILS

-------------------------------------------------------------------------------------------
wire---pc=       40|TIME : 105000  || clk : 1  || Mem_data:                0  ||  || mem_wb_MemtoReg: 0  || mem_wb_RegWrite: 0  || rd: xxxxx imm:
0
reg---pc=       40|TIME : 105000  || clk : 1  || Mem_data:                0  ||  || mem_wb_MemtoReg: 0  || mem_wb_RegWrite: 0  || rd: xxxxx, imm:
0
-------------------------------------------------------------------------------------------
MEM_DATA REGISTER DETAILS

-------------------------------------------------------------------------------------------
wire---mem_wb_rd :  x | mem_wb_alu_result :                x | mem_wb_mem_data :                0 | mem_wb_RegWrite : 0 | mem_wb_MemtoReg : 0
reg---mem_wb_rd :  x | mem_wb_alu_result :                x | mem_wb_mem_data :                0 | mem_wb_RegWrite : 0 | mem_wb_MemtoReg : 0
-------------------------------------------------------------------------------------------
Time: 0 | clk: 1 | MemtoReg: 0 | MemRead: 0 | MemWrite: 0 | rd:  x | ALU_Result:                x | MemData:                0
Memory Values:                2,                1,                x
Time: 0 | clk: 0 | MemtoReg: 0 | MemRead: 0 | MemWrite: 0 | rd:  x | ALU_Result:                x | MemData:                0
Memory Values:                2,                1,                x
processor.v:115: $finish called at 115000 (1ps)
TIME : 115000                clk  : 1
```



```
wire [1:0] forwardA ,forwardB;
forwarding_unit fwd_unit (
    .EX_MEM_RegWrite(ex_mem_RegWrite),
    .MEM_WB_RegWrite(mem_wb_RegWrite),
    .EX_MEM_Rd(ex_mem_rd),
    .MEM_WB_Rd(mem_wb_rd),
    .ID_EX_Rs1(id_ex_rs1),
    .ID_EX_Rs2(id_ex_rs2),
    .ForwardA(forwardA),
    .ForwardB(forwardB)
);
reg [63:0]operandBdummy;
always @(*) begin
    // Determine operandA based on ForwardA
    case (forwardA)
        2'b00: operandA = id_ex_rs1_val;          // No forwarding
        2'b01: operandA = mem_wb_mem_data;         // Forward from MEM/WB stage
        2'b10: operandA = ex_mem_alu_result;  // Forward from EX/MEM stage
        default: operandA = id_ex_rs1_val;        // Default to register value
    endcase

    // Determine operandB based on ForwardB
    case (forwardB)
        2'b00: operandBdummy =id_ex_rs2_val;  // Use rs2_val or immediate
        2'b01: operandBdummy = mem_wb_mem_data;         // Forward from MEM/WB stage
        2'b10: operandBdummy = ex_mem_alu_result;  // Forward from EX/MEM stage
        default: operandBdummy = id_ex_rs2_val;        // Default to register value
    endcase
    case(alu_src)
        1'b0: operandB = operandBdummy;
        1'b1: operandB = id_ex_imm;
    endcase

end
```

ld x3 0(x0)
ld x4 8(x0)
add x5 ×3 ×4

```
17
18      reg [63:0] data_memory [0:255]; // Memory array
19      reg  wb_write_reg;
20      // Initialize memory with sample values
21      initial begin
22          data_memory[0] = 64'd10;
23          data_memory[1] = 64'd12;
24          data_memory[2] = 64'd6;
25      end
26
```

```
0  || rd: xxxxx
reg---pc=         24|TIME : 75000  || clk : 1  || Mem_data:                  0  ||  || mem_wb_MemtoReg: 0  || mem_wb_RegWrite: 1
  || rd: 00101
-------------------------------------------------------------------------------------------------
MEM_DATA REGISTER DETAILS

-------------------------------------------------------------------------------------------------
wire---mem_wb_rd :  x | mem_wb_alu_result :                  x | mem_wb_mem_data :                  0 | mem_wb_RegWrite : 0 |
 mem_wb_MemtoReg : 0
reg---mem_wb_rd :  5 | mem_wb_alu_result :                 22 | mem_wb_mem_data :                  0 | mem_wb_RegWrite : 1 |
mem_wb_MemtoReg : 0
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
 HAZARD DETAILS DETAILS

id_ex_memread : 0 ||  id_ex_rd :  x || if_id_rs1 :  x || if_id_rs2 :  x || hazard_detected : 0
 TB OUTPUT---00000000000000000000000000000000000000000000000000000000000000000000000000000000
-------------------------------------------------------------------------------------------------
.
time : 75000 ||Register x1 :                  x | x2 :                  x | x3 :                 10 | x4 :
  12 | x5 :                 22 | x6 :                  x | x7 :                  9 || PC :         24 |time=75000
-------------------------------------------------------------------------------------------------
.
Time: 0 | clk: 1 | MemtoReg: 0 | MemRead: 0 | MemWrite: 0 | rd:  x | ALU_Result:                  x | MemData:
   0
Memory Values:                10,                 12,                  6
Time: 0 | clk: 0 | MemtoReg: 0 | MemRead: 0 | MemWrite: 0 | rd:  x | ALU_Result:                  x | MemData:
   0
Memory Values:                10,                 12,                  6
 TB OUTPUT---00000000000000000000000000000000000000000000000000000000000000000000000000000000
-------------------------------------------------------------------------------------------------
.
time : 85000 ||Register x1 :                  x | x2 :                  x | x3 :                 10 | x4 :
  12 | x5 :                 22 | x6 :                  x | x7 :                  9 || PC :         28 |time=85000
-------------------------------------------------------------------------------------------------
.
TIME : 85000                        clk  : 1

INSTRUCTION  FETCH DETAILS

-------------------------------------------------------------------------------------------------
time=85000|PC:         28| if_instruction=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| pc_write 1
```

This handles the load-use hazard, but for the second load and add, there will be a stall. As a stall occurs, this error is rectified.

```
ld x7 0(x0)
beq x7 ×3
```

this also gives hazard detected when

id_ex_mem_read==1&&if_id_branch==1&&(if_id_rs1==id_ex_rd||if_id_rs2=id_ex_rd) ==1

✅ Forwarding from EX/MEM Stage

In your code, you have implemented **data forwarding from EX/MEM stage** using a **forwarding unit**. The conditions for forwarding are:

Condition 1 (EX/MEM Forwarding):

If the **EX/MEM stage** has a valid result to write back (`RegWrite=1`) and the destination register (`EX/MEM.Rd`) matches the source register of the current instruction (`ID/EX.Rs1` or `ID/EX.Rs2`), then the result is **forwarded** directly from the EX stage.

Your code implementation for this:

```
case (forwardA)
    2'b00: operandA = id_ex_rs1_val;         // No forwarding
    2'b01: operandA = mem_wb_mem_data;       // Forward from MEM/WB stage
    2'b10: operandA = ex_mem_alu_result;     // Forward from EX/MEM stage
    default: operandA = id_ex_rs1_val;
endcase
```

Explanation:

- If **EX/MEM.RegWrite == 1** and **EX/MEM.Rd == ID/EX.Rs1**, then the result from the EX stage (`ex_mem_alu_result`) is forwarded to **OperandA**.
- Similarly, for **OperandB**, the EX stage result is forwarded if required.

✅ Forwarding from MEM/WB Stage

If the instruction result has already reached the **MEM/WB stage**, the value is forwarded from **MEM/WB.MemData** or **MEM/WB.ALUResult**.

Condition 2 (MEM/WB Forwarding):

```
case (forwardB)
    2'b00: operandBdummy = id_ex_rs2_val;
    2'b01: operandBdummy = mem_wb_mem_data;
    2'b10: operandBdummy = ex_mem_alu_result;
    default: operandBdummy = id_ex_rs2_val;
endcase
```

Explanation:

- If the result is available in the **MEM/WB stage**, it is forwarded to the execution stage to prevent incorrect data usage.
- This effectively eliminates the need for the instruction to wait for the write-back stage.

✅ Conditions for Forwarding

The forwarding conditions in your design are checked in the **Forwarding Unit** module:

```
if (EX_MEM_RegWrite
    && (EX_MEM_Rd != 0)
    && (EX_MEM_Rd == ID_EX_Rs1))
    ForwardA = 10;

if (MEM_WB_RegWrite
    && (MEM_WB_Rd != 0)
    && (MEM_WB_Rd == ID_EX_Rs1))
    ForwardA = 01;
```

- This prevents data hazards when `ID/EX` instruction depends on `EX/MEM` or `MEM/WB` stage results.

---

2. Load-Use Data Hazard Handling Using Stall Mechanism

A **Load-Use hazard** occurs specifically when an instruction **loads data from memory** and the very next instruction uses the loaded value.

Example:

```
ld x3, 0(x1)
add x4, x3, x5
```

In this scenario, the `add` instruction needs the value of `x3` from memory, but the `lw` instruction is still in the **MEM stage**. Without a stall, the `add` instruction would receive an invalid value.

✅ Stall Condition in Design

In your processor, you have effectively handled the **Load-Use Hazard** using a **Hazard Detection Unit**. The unit generates a `stall` signal based on the following condition:

Condition for Stall (Load-Use Hazard):

```
if (id_ex_MemRead &&
  ((id_ex_rd == if_id_rs1) || (id_ex_rd == if_id_rs2))) begin
  hazard_detected = 1;
end
```

Explanation:

- If the **ID/EX.MemRead** signal is high (`Load instruction`).
- And either **ID/EX.Rd == IF/ID.Rs1** or **ID/EX.Rd == IF/ID.Rs2** (dependency exists).
- Then, the pipeline is stalled for **one clock cycle**.

✅ Stall Control Mechanism

The stall mechanism works by freezing the **PC** and **IF/ID registers** for one cycle:

```
always @(posedge clk) begin
    if_id_write = if_id_write_wire;
```

```
    pc_write = pc_write_wire;
endPCWrite = 0 → Prevents fetching the next instruction.
```

- **IF/ID_Write = 0** → Prevents updating the IF/ID pipeline register.
- This provides time for the `lw` instruction to complete and update the register file.

✅ Effect of Stall

- One cycle is delayed for dependent instruction.
- After the `load` instruction completes, the next instruction executes normally.

---

3. Correct Write-Back Handling

✅ Handling Write-Back Hazards

A write-back hazard can occur when two instructions try to write to the same register. In your processor, you have ensured that:

- If the **MEM/WB stage** has a valid write (`RegWrite=1`) and the destination register (`Rd`) is non-zero, it writes back to the register file.
- Your code:

```
always @(*) begin
    if (wb_RegWrite) begin
        register_file[wb_rd] = wb_registerout;
    end
end
```

- This ensures **no overwrite** happens in the pipeline and maintains data consistency.

✅ 5. Conditions for Forwarding and Stalls

Hazard Type	Condition (from code)	Action Taken
EX/MEM Forwarding	`(EX/MEM.RegWrite == 1) && (EX/MEM.Rd == ID/EX.Rs1)`	Forward EX/MEM.ALU_Result to EX stage
MEM/WB Forwarding	`(MEM/WB.RegWrite == 1) && (MEM/WB.Rd == ID/EX.Rs1)`	Forward MEM/WB.Data to EX stage
Load-Use Hazard	`(ID/EX.MemRead == 1) && ((ID/EX.Rd == IF/ID.Rs1)` or `(ID/EX.Rd == IF/ID.Rs2))`	Stall Pipeline by one cycle

Work distribution :

Eswar Kumar Anantha

- **Test benches for all stages(pipeline)**
- **EX(sequential) WB(sequential)**
- Integrating(sequential)
- Hazard detection and forwarding (debugging)

K.Roshan Lal

- Test Benches for all stages (sequential)
- Debugging test cases (Pipeline,sequential)
- ID stage (sequential)
- Integration for pipeline and debugging it

M.Radheshyam

- Generating test cases (Sequential) (Pipeline)
- Report for Sequential Execution, Pipeline Execution
- generate both memory block (instruction and memory) ,compartor(external function)
- chages in pipeline all stages