CO502 – Advanced Computer Architecture
Group 04
E/17/083, E/19/455

Lab Part 1 – Planning

Activity 01

RV32I Base Instruction Set

1. U-Type Instructions (LUI, AUIPC)

I. LUI (Load Upper Immediate)

Opcode – `0110111`
Format – `imm[31:12] | rd | opcode`
Purpose – Load a 20-bit constant into the upper 20 bits of a register
Operation - `rd = imm << 12`
Use Case - Efficiently loads large constants into a register.
Example - `lui x5, 0x12345` loads `0x12345000` into `x5`

II. AUIPC (Add Upper Immediate to PC)

Opcode - `0010111`
Format - `imm[31:12] | rd | opcode`
Purpose - Generate PC-relative addresses.
Operation - `rd = PC + (imm << 12)`
Use Case: Useful for position-independent code, loading addresses.
Example - `auipc x1, 0x2` at PC `0x1000` → `x1 = 0x1000 + 0x2000 = 0x3000`

2. J-Type Instructions

I. JAL (Jump and Link)

Opcode - `1101111`
Format - `imm[20|10:1|11|19:12] | rd | opcode`
Purpose - Unconditional jump with return address.

Operation - `rd = PC + 4; PC += imm`
Use Case: Function calls, long-distance jumps.
Note - Immediate is signed and aligned (LSB is 0).
Example - `jal x1, 256` saves return address and jumps ahead by 256 bytes.

3. I-Type Instructions (LUI, AUIPC)

I. JALR (Jump and Link Register)

Opcode – `1100111`
Format – `imm[11:0] | rs1 | 000 | rd | opcode`
Purpose – Jump to address in register + offset.
Operation - `rd = PC + 4; PC = (rs1 + imm) & ~1`
Use Case - Function returns, indirect jumps.
Example - `jalr x0, 0(x1)` → `PC = x1` (returns to address in x1)

II. Load Instructions

Opcode – `0000011`

Instruction	funct3	Description
LB	`000`	Load signed byte
LH	`001`	Load signed half-word
LW	`010`	Load word
LBU	`100`	Load unsigned byte
LHU	`101`	Load unsigned half word

Format – `imm[11:0] | rs1 | funct3 | rd | opcode`
Purpose – Read from memory into a register.
Operation - `rd = Mem[rs1 + imm]` (with sign or zero extension depending on instruction)

III. Immediate Instructions

Opcode – `0010011`

Instruction	funct3	Operation	Description
ADDI	`000`	`rd = rs1 + imm`	Adds a signed 12-bit immediate to the value in register `rs1` and stores the result in `rd`
SLTI	`010`	`rd = (rs1 < imm) ? 1 : 0`	Compares `rs1` and the immediate value as signed integers
SLTIU	`011`	`rd = (unsigned(rs1) < unsigned(imm)) ? 1 : 0`	Like **SLTI**, but treats both operands as unsigned integers.
XORI	`100`	`rd = rs1 ^ imm`	Performs a **bitwise XOR** between `rs1` and the immediate.
ORI	`110`	`rd = rs1 | imm`	Performs a **bitwise OR** between `rs1` and the immediate.
ANDI	`111`	`rd = rs1 & imm`	Performs a **bitwise AND** between `rs1` and the immediate.

Format – `imm[11:0] | rs1 | funct3 | rd | opcode`

Shift Immediate (same opcode `0010011`, funct7, funct3 are used)

Instruction	Funct3	Funct7	Operation	Description
SLLI	`001`	`0000000`	`rd = rs1 << shamt`	Shifts `rs1` **left** by `shamt` bits, filling with zeros.
SRLI	`101`	`0000000`	`rd = rs1 >> shamt` (logical, zero-fill)	Shifts `rs1` **right** by `shamt` bits, filling the left with **zeros** (logical shift).

SRAI	`101`	`0100000`	`rd = rs1 >> shamt` (arithmetic, sign-extend)	Shifts `rs1` **right** by `shamt` bits, **preserving the sign** (copying the sign bit).

Format – `funct7 | shamt | rs1 | funct3 | rd | opcode`

4. B-type (Branch Instruction)

Opcode – `1100011`

Instruction	funct3	Description
BEQ	`000`	Branch if equal
BNE	`001`	Branch if not equal
BLT	`100`	Branch if less than (signed)
BGE	`101`	Branch if greater or equal
BLTU	`110`	Branch if less (unsigned)
BGEU	`111`	Branch if greater (unsigned)

Format - `imm[12|10:5] | rs2 | rs1 | funct3 | imm[4:1|11] | opcode`
Operation - If condition is true, `PC = PC + imm`

5. S-type (Store Instructions)

Opcode - `0100011`

Instruction	funct3	Description
SB	`000`	Store byte
SH	`001`	Store half-word
SW	`010`	Store word

Format - `imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode`
Operation - `Mem[rs1 + imm] = rs2`

6. R-type (Register-Register ALU)

Opcode: `0110011`

Instruction	funct3	funct7	Operation
ADD	`000`	`0000000`	`rd = rs1 + rs2`
SUB	`000`	`0100000`	`rd = rs1 - rs2`
SLL	`001`	`0000000`	`rd = rs1 << rs2[4:0]`
SLT	`010`	`0000000`	`rd = (rs1 < rs2)` (signed)
SLTU	`011`	`0000000`	`rd = (rs1 < rs2)` (unsigned)
XOR	`100`	`0000000`	`rd = rs1 ^ rs2`
SRL	`101`	`0000000`	Logical right shift
SRA	`101`	`0100000`	Arithmetic right shift
OR	`110`	`0000000`	`` `rd = rs1 ``
AND	`111`	`0000000`	`rd = rs1 & rs2`

Format - `funct7 | rs2 | rs1 | funct3 | rd | opcode`

7. FENCE, ECALL, EBREAK (Make them NOP or Illegal Instructions)

I. FENCE

Opcode – `0001111`
Format – `fm | pred | succ | rs1=0 | funct3=000 | rd=0 | opcode`
Purpose – Memory and I/O ordering.
Operation - `rd = imm << 12`
Used in multi-core or I/O situations to enforce memory operation order.

II. ECALL / EBREAK

Opcode - `1110011`
Format - `000000000000 | 00000 | 000 | 00000 | opcode`
ECALL - Environment call. Used for syscall interface.
EBREAK - Causes a breakpoint trap.

RV32M Multiply and Divide Instructions

All the M instructions have R type instruction format

`funct7 | rs2 | rs1 | funct3 | rd | opcode`

1. MUL - Multiply (Low 32 bits, signed)

Opcode – `0110011`
funct7 – `0000001`
funct3 – `000`
Purpose – Signed × Signed, return **lower 32 bits**
Operation - `rd = (int32_t)rs1 * (int32_t)rs2`
Use Case - General fast signed multiplication
Example - `MUL x5, x2, x3` → `x5 = x2 * x3` (low 32 bits only)

2. MULH - Multiply High (signed × signed)

Opcode – `0110011`
funct7 – `0000001`
funct3 – `001`

Purpose – Signed × Signed, return **lower 32 bits**
Operation - `rd = ((int64_t)rs1 * (int64_t)rs2) >> 32`
Use Case - When full 64-bit result is needed.
Example - `MULH x5, x2, x3`

3. MULHSU - Multiply High (signed × unsigned)

Opcode – `0110011`
funct7 – `0000001`
funct3 – `010`
Purpose – Signed × Signed, return **lower 32 bits**
Operation - `rd = ((int64_t)rs1 * (int64_t)rs2) >> 32`
Use Case - When full 64-bit result is needed.
Example - `MULH x5, x2, x3`

4. MULHU - Multiply High (unsigned × unsigned)

Opcode – `0110011`
funct7 – `0000001`
funct3 – `011`
Purpose – Unsigned × Unsigned, return **upper 32 bits**
Operation - `rd = ((uint64_t)rs1 * (uint64_t)rs2) >> 32`
Use Case - Needed in big integer arithmetic or cryptography
Example - `MULHU x5, x2, x3`

5. DIV - Divide (signed)

Opcode – `0110011`
funct7 – `0000001`
funct3 – `100`
Purpose – Signed division
Operation - `rd = (int32_t)rs1 / (int32_t)rs2`
Use Case - Integer arithmetic
Example - `DIV x5, x2, x3`

6. DIVU - Divide (Unsigned)

Opcode – `0110011`
funct7 – `0000001`
funct3 – `101`

Purpose – Unsigned division
Operation - `rd = (uint32_t)rs1 / (uint32_t)rs2`
Use Case - Pointer arithmetic, address calculation
Example - `DIVU x5, x2, x3`

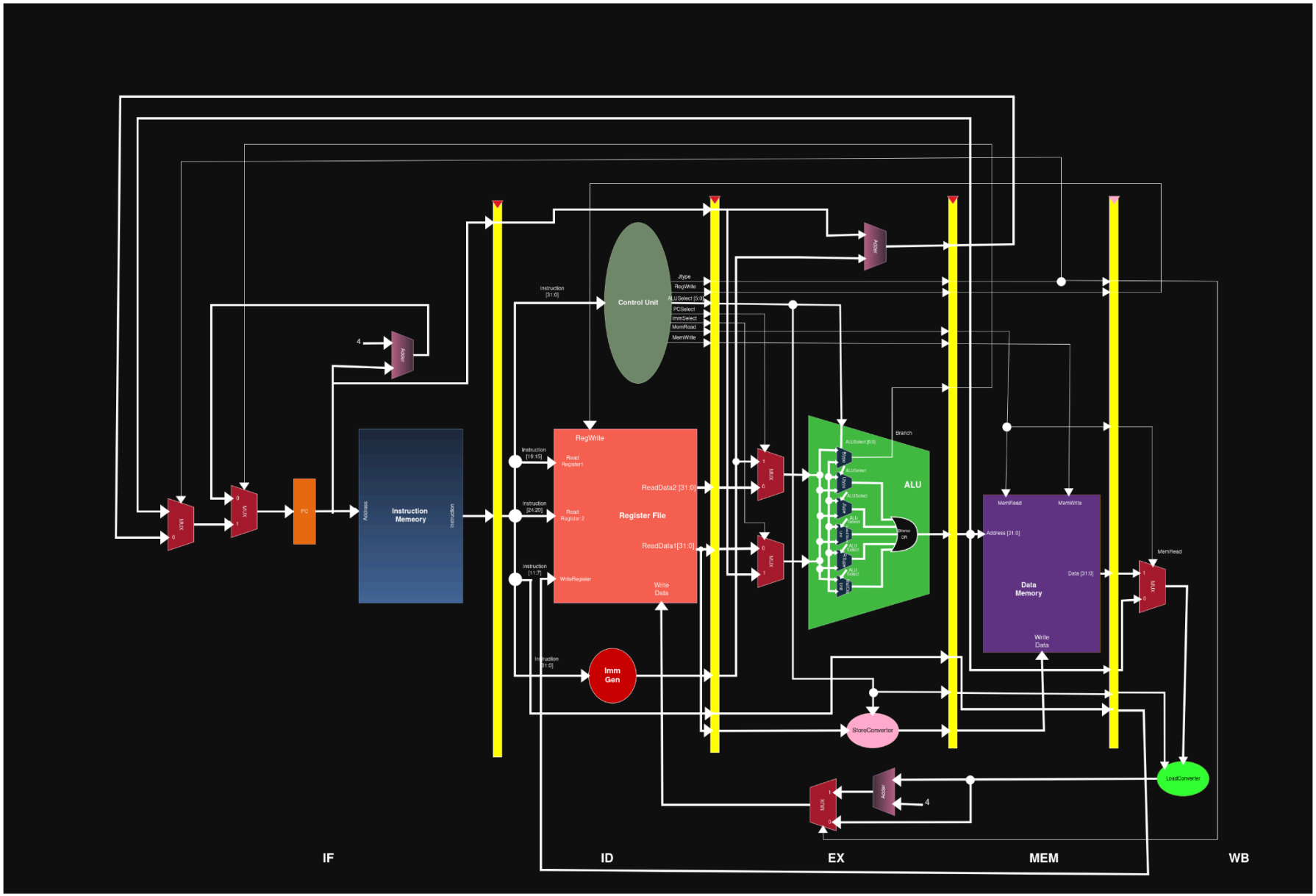
7. REM - Remainder (Signed)

Opcode – `0110011`
funct7 – `0000001`
funct3 – `110`
Purpose – Signed division
Operation - `rd = (int32_t)rs1 % (int32_t)rs2`
Use Case - Looping logic, modulo operations
Example - `REM x5, x2, x3`

8. REMU - Remainder (Unsigned)

Opcode – `0110011`
funct7 – `0000001`
funct3 – `111`
Purpose – Unsigned remainder
Operation - `rd = (uint32_t)rs1 % (uint32_t)rs2`
Use Case - Unsigned modulo arithmetic
Example - `REMU x5, x2, x3`

Activity 2


Control Unit
Jtype
RegWrite
ALUSelect [5:0]
PCSelect
ImmSelect
MemRead
MemWrite
Instruction [31:0]
Adder
4
MUX
PC
Instruction Memeory
Address
Instruction
RegWrite
Instruction [19:15]
Read Register1
Instruction [24:20]
Read Register 2
Instruction [11:7]
WriteRegister
ReadData2 [31:0]
Register File
ReadData1[31:0]
Write Data
Imm Gen
Branch
ALUSelect [5:0]
ALU
Shrxd OR
MemRead
MemWrite
Address [31:0]
Data Memory
Data [31:0]
Write Data
StoreConverter
LoadConverter
IF
ID
EX
MEM
WB

Hardware Units

- Arithmetic and Logical Unit
- Control Unit
- Register File
- Instruction Memory
- Data Memory
- Immediate Generator
- Program Counter
- Multiplexers
- Adders
- Shifters
- StoreConverter
- LoadConverter

Control Signals

- ALUSelect - Signal to do selections inside ALU when operating
- ImmSelect - Select Whether Immediate value or Register output
- PCSelect - Select Whether Program Counter Value or Register Output
- Jtype - 1 if a Jtype instruction is operated
- MemRead - Whether data is read from data memory
- MemWrite - Whether data is written to Data Memory
- RegWrite - Whether Data is written to register file

Activity 3

- Tests to verify functionality of components
- Test fetching instructions from memory based on the program counter
- Test updating the program counter according to branch and jump instructions
- Test proper decoding of different instruction types (R-type, I-type, S-type, B-type, U-type, J-type).
- Test correct reading of operands from the register file.

- Test correct results of arithmetic and logical operations.
- Test proper loading and storing of data from/to memory.
- Tests writing results to the register file.

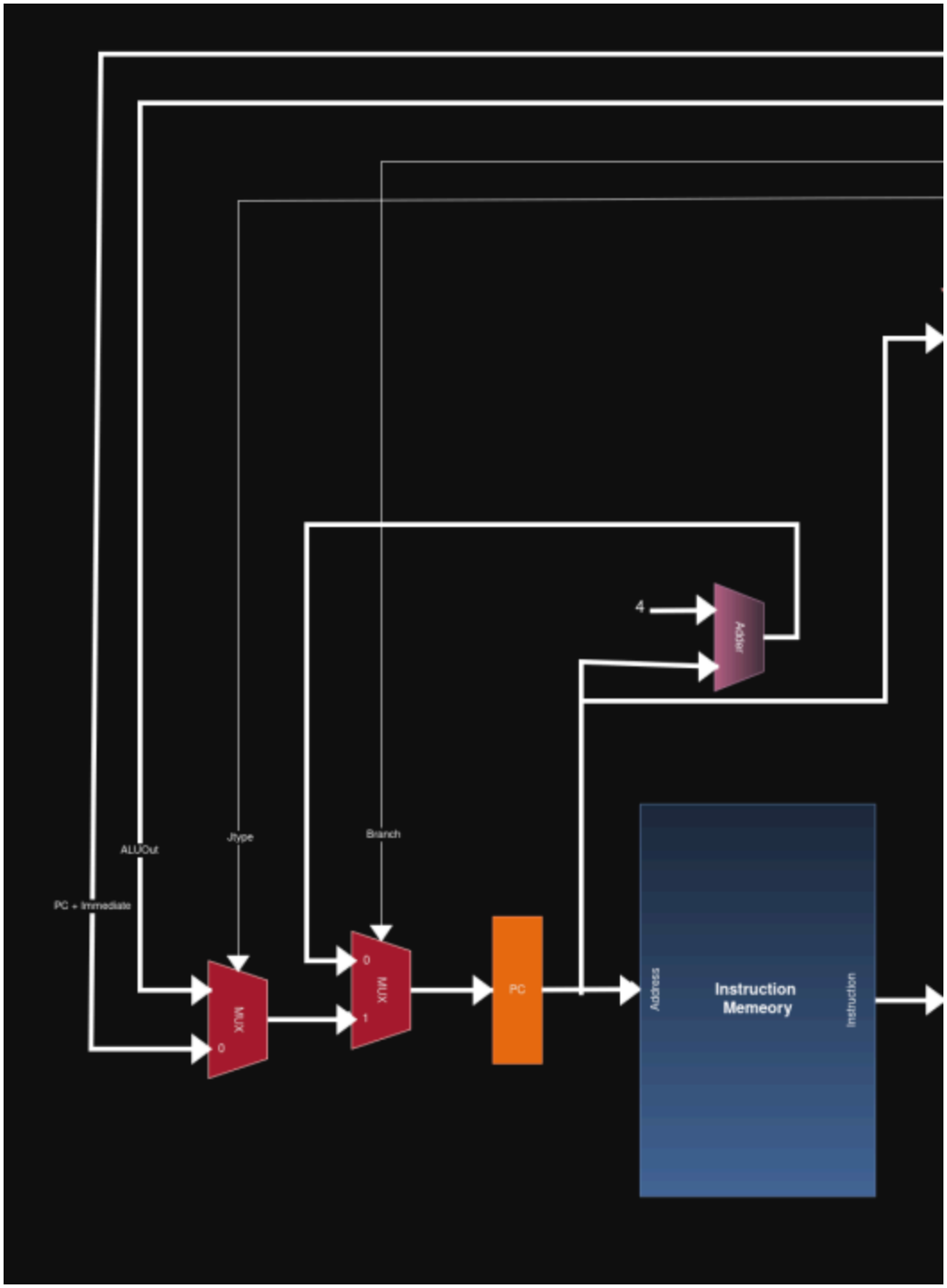
Lab Part 02 - Hardware Units

The RISC-V pipelined processor is organized into five stages: Fetch, Decode, Execute, Memory, and Writeback, each handling a specific part of instruction processing. In the **Fetch** stage, the instruction is retrieved from memory using the program counter. The **Decode** stage interprets the instruction and reads necessary registers. Execution, Memory access, and Writeback stages perform ALU operations, handle data memory, and update register values respectively, with dedicated modules handling each function in parallel for improved performance.

Fetch Cycle

Fetch Cycle consist of following modules

- Program Counter
- Instruction Memory
- Multiplexer
- Adder


4
Adder
ALUOut
Jtype
Branch
PC + Immediate
MUX
0
MUX
0
1
PC
Address
Instruction Memeory
Instruction

Program Counter

The Program Counter (PC) holds the address of the current instruction being fetched from memory. It ensures the processor executes instructions in the correct sequence. After each instruction, the PC is updated to point to the next instruction, or to a new address in case of branches or jumps. It is essential for tracking and controlling the flow of execution in a program.

Design Choices

1. **Synchronous Update with Clock:**
    - The `PC_out` is updated on the **positive edge** of the clock, which aligns with typical synchronous design principles used in pipeline architectures.

2. **Asynchronous Reset:**
    - The `reset` signal is handled asynchronously (i.e., checked along with the clock in the sensitivity list). This allows the PC to reset immediately, regardless of the clock, which is useful for global initialization.

3. **Simple and Minimal Design:**
    - The module has only essential functionality: receive a new PC input and output it on the next clock cycle, or reset to 0.
    - No enable or stall logic is present, keeping it simple for a basic pipeline design.

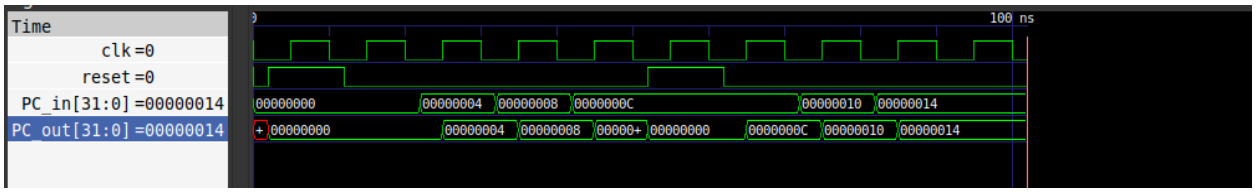
4. **32-bit Address Width:**
    - The PC is 32 bits wide, which is standard for RISC-V RV32-based designs.

Limitations

1. **No Stall or Freeze Capability:**
    - There's no stall signal to hold the PC value (i.e., prevent it from updating). This is critical in pipelined processors to handle control hazards (e.g., branches, load-use hazards).

2. **No Branch/Jump Handling Logic:**
    - The PC simply takes the input as-is; it doesn't include logic for incrementing PC or resolving branch/jump targets. That logic must exist externally and supply the correct `PC_in`.
3. **No Instruction Memory Boundary Checking:**
    - There's no logic to check if the `PC_in` is within the bounds of instruction memory. This could lead to invalid fetches unless managed externally.
4. **No Support for Delayed PC Update:**
    - Some advanced designs require delayed or speculative PC updates (e.g., branch prediction); this PC design doesn't support that.
5. **Asynchronous Reset May Lead to Glitches:**
    - While asynchronous reset allows fast initialization, it may lead to metastability or glitches in some FPGAs or ASICs if not synchronized properly with the clock domain.

Sample Waveform



Instruction Memory

Instruction Memory stores the program's machine code instructions that the processor needs to execute. During the **Fetch** stage, the instruction at the address specified by the Program Counter is read from this memory. It is typically read-only and separate from data memory to simplify access. This module ensures the CPU has timely access to the instructions needed for processing.

Design Choices

Word-Aligned Addressing (`read_address[7:2]`)

- Used bits `[7:2]` to index 64 words (4-byte aligned instructions), enabling access to 256 bytes of instruction memory.

- Ensures proper word-aligned addressing for 32-bit instructions in RISC-V.

Instruction Memory Size (`I_Mem[0:63]`)

- Memory is statically defined to hold **64 instructions**, which is sufficient for small to mid-sized programs or testbenches.
- Keeps resource usage low for simulations.

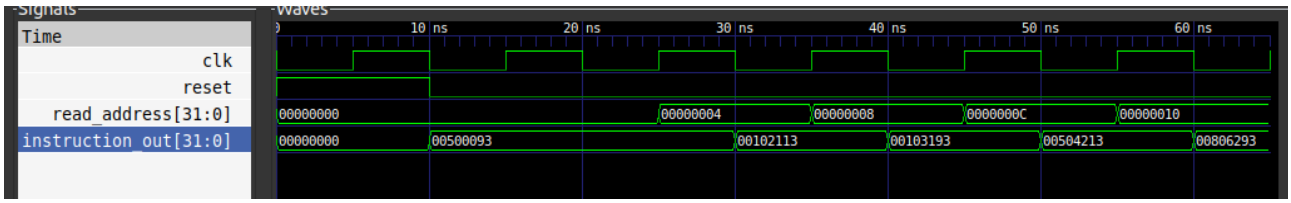
Hex File Initialization (`$readmemh`)

- The module loads instructions from an external **hex file** at simulation start, simplifying testing and allowing easy instruction updates without recompiling code.
- This decouples the processor logic from the program content.

Negative Edge Clocking (`@(negedge clk)`)

- Instructions are fetched on the **falling edge** of the clock.
- This is an uncommon choice and aimed at synchronizing with a **positive-edge sampling Fetch/Decode register**, ensuring the instruction is ready before the rising edge.

Sample Waveform



Limitations of Instruction Memory

The current implementation of the Instruction Memory module is functional for basic testing and simulation purposes but has several limitations that must be considered when scaling or adapting it to more advanced or real-world applications:

1. Fixed Size

- The memory is hardcoded to store only **64 instructions** (`I_Mem[0:63]`), which restricts the program length.
- This makes the system unsuitable for larger programs or complex software that exceeds this instruction limit.

2. No Write Support

- The module supports only **read operations**; there is **no mechanism to write new instructions** into memory during runtime.
- This means the instruction memory is **read-only after initialization**, making it unsuitable for self-modifying code or systems requiring runtime instruction updates (e.g., JIT compilation).

3. No Byte-Addressing

- The design uses the **read address directly as an index** into the instruction array (`I_Mem[read_address]`).
- This assumes that the address is **already aligned and scaled** (i.e., PC is divided by 4), which may not be handled elsewhere in the pipeline.
- It lacks logic to perform **byte-addressed alignment** or error checking for misaligned instruction fetches.

4. Lack of Timing Control

- The current design assumes **combinational read** from memory (`instruction_out` is driven directly from the array).
- It **does not model realistic memory latency**, such as access delays in real synchronous RAM.
- This simplification may cause mismatches in timing when integrating with pipelined architectures or external memory interfaces.

5. No Support for Caching or Memory Hierarchy

- The module does not integrate with any **cache system** or **instruction prefetching** mechanisms.

- As a result, it does not model realistic access patterns or performance optimizations seen in modern processors.

6. No Error Handling or Protection

- There is no error handling for **out-of-bound reads**, invalid addresses, or memory protection violations.
- This could lead to undefined behavior or data corruption if incorrect addresses are issued.

7. Not Synthesizable (in current form)

- The use of `$readmemh` inside the `initial` block is typically allowed only in **simulation**, not in synthesis for FPGA or ASIC targets.
- For synthesis, a **ROM block or memory IP** with initialization support must replace this implementation.

Use of Two Multiplexers in Fetch Stage

To select between:

- **PC + 4** (default next instruction)
- **Branch target address**, and
- **Jump target address (e.g., JAL, JALR)**

Design Choices

1. **Combinational Logic:**
    - The Mux is implemented using a simple `assign` statement, making it a purely combinational circuit. This ensures zero clock latency and instant propagation based on input changes.

2. **2-to-1 Multiplexer:**
    - It chooses between two 32-bit inputs `A` and `B` based on a 1-bit control signal `select`. This structure is widely used in processor datapaths, particularly for:

        - Selecting between ALU output and memory output.
        - PC update paths (e.g., PC+4 vs branch target).
        - Write-back stage (e.g., data vs address).

3. **Compact and Readable Syntax:**
    - The ternary operator (`? :`) provides a concise way to implement the selection logic, minimizing code complexity.

Limitations

1. **No Timing or Enable Control:**
    - The module does not support any clock or enable signal. It is always active, which is acceptable for simple datapath muxing but limits more complex timing control.

2. **No Default or Undefined Handling:**
    - If `select` is driven by an unknown or undefined value (e.g., `'x'`), the output may also become undefined. No assertion or check is implemented to handle this.

32-bit Adder Module

Module Overview

Usage of adder in the Fetch Stage - Increment the program counter to point to the next instruction (default behavior when there's no branch or jump).

Design Choices

Combinational Logic

- The module operates entirely as a combinational logic block, producing immediate outputs in response to changes in the input operands.
- This is suitable for high-speed datapaths where delay should be minimized.

Fixed Operand Width

- The inputs and output are fixed to 32-bit, which aligns with standard word lengths in architectures like RISC-V RV32I.
- This allows direct integration into 32-bit processor designs without additional conversion logic.

Simplicity and Focus

- The design intentionally avoids unnecessary complexity—there are no flags, carry-out bits, or clocking mechanisms.
- This simplicity improves reusability and makes it easier to test and verify.

Stateless Operation

- The adder does not contain any internal state, clocks, or resets.
- This choice enables its use in purely combinational paths like address calculation or ALU execution.

Limitations

No Signed Arithmetic Support

- The design assumes unsigned data unless specifically modified.
- This limits its direct applicability in operations requiring signed two's complement behavior.

No Overflow or Carry Detection

- The module does not provide flags for overflow, carry-out, or zero-detection, which are typically required for conditional branches or comparisons.

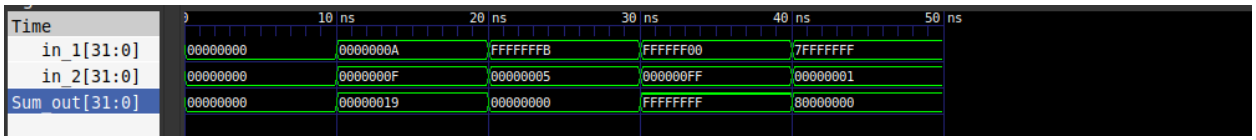
Fixed Bit Width

- The adder is limited to 32-bit inputs and output.
- It is not scalable without manual code modification, making it less flexible for designs using 8-bit, 16-bit, or 64-bit word lengths.

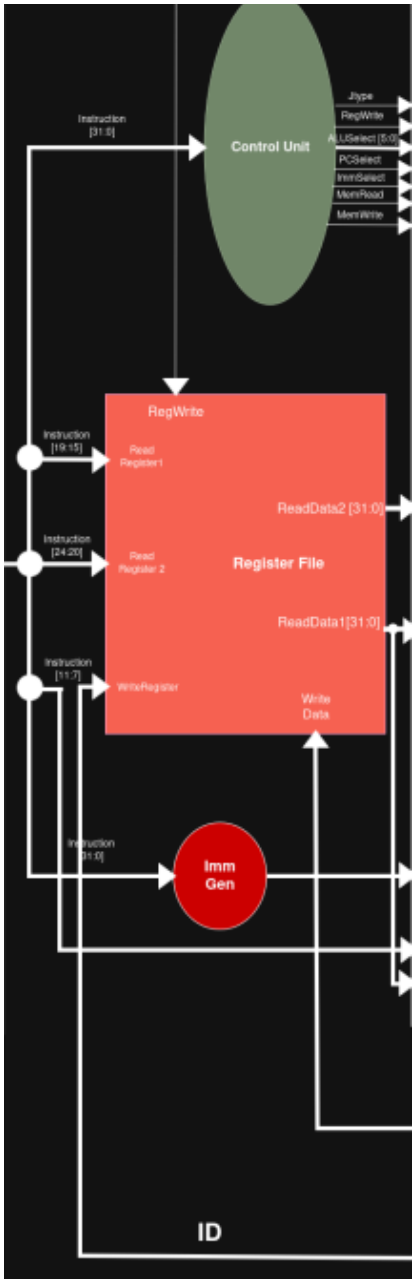
No Timing or Delay Considerations

- The module does not account for propagation delays, which may be critical in physical design or timing analysis.

Sample Waveform



Decode Stage



The Decode stage of the pipelined RISC-V processor is responsible for interpreting instructions and preparing operands and control signals for the Execute stage. It consists of three main components: the **Control Unit**, **Register File**, and **Immediate Generator**. The Control Unit decodes the instruction opcode and generates necessary control signals such as ALU operation type, memory access, register write enables, and jump/branch decisions. The Register File reads source operands based on the instruction fields and supports **write-back of data from the Writeback stage** using `RegWriteW`, `WriteAddressW`, and `writeDataW`, enabling data to be updated even while pipelining is active. The Immediate Generator extracts and properly formats the immediate values based on instruction type. Together, these modules ensure that the processor correctly decodes instructions and passes all required data and control signals to the Execute stage.

Control Unit

1. **Purpose**:
   The Control Unit decodes the 32-bit RISC-V instruction and generates all the necessary control signals for the Execute, Memory, and Writeback stages of the pipeline.

2. **Inputs**:

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

3. **Outputs**:
   - `aluSelect[5:0]`: Specifies the ALU operation to be performed.
   - `MemWrite`: Enables write operation to data memory.
   - `MemRead`: Enables read operation from data memory.
   - `ImmSelect`: Enables selection of immediate operands.
   - `PCSelect`: Controls PC source selection (for jumps/branches).
   - `regWrite`: Enables writing to a register in the register file.
   - `Jtype`: Indicates whether the instruction is a jump-type (JAL/JALR).

4. **Opcode Decoding**:
    - The module decodes the **7-bit opcode** to determine the instruction type (R-type, I-type, S-type, B-type, J-type, U-type).
    - Additional fields like `funct3` and `funct7` are used for further classification within instruction types (e.g., ADD vs. SUB vs. MUL).

5. **ALU Operation Encoding (`aluSelect`)**:
    - A 6-bit custom ALU operation code is assigned for each instruction, providing a wide range of operations including:
        - Arithmetic (ADD, SUB, MUL, etc.)
        - Logical (AND, OR, XOR, shifts)
        - Comparison (SLT, SLTU)
        - Branch conditions (BEQ, BNE, etc.)
        - Load/store indexing
    - Separate codes are also used for memory operations and jump calculations.

6. **Control Signal Generation**:
    - For **R-type** (e.g., ADD, SUB, MUL): sets `aluSelect`, enables `regWrite`, no immediate or memory access.
    - For **I-type** (e.g., ADDI, LW, JALR): sets `ImmSelect`, `regWrite`, possibly `MemRead`, and ALU code.
    - For **S-type** (e.g., SW): sets `MemWrite`, `ImmSelect`, and `aluSelect`.
    - For **B-type** (e.g., BEQ): sets `aluSelect` for comparison, sets `PCSelect` via branch logic.
    - For **J-type** (JAL, JALR): sets `regWrite`, `ImmSelect`, `PCSelect`, and asserts `Jtype`.

    - For **U-type** (LUI, AUIPC): sets `ImmSelect`, `aluSelect`, `regWrite`.

7. **Special Instructions Handling**:
    - **NOP**: Treated as an ADDI to x0 with 0 immediate (no effect).
    - **FENCE, ECALL, EBREAK**: Treated as no-ops for safe simulation and pipelined flow.

8. **Default Behavior**:
    - At the start of each evaluation, all outputs are reset to default (zero) to avoid unintended control signal activation.
    - Only specific fields are updated based on opcode decoding, making the module robust and hazard-resistant.

9. **Modular and Extensible**:
    - The design is structured to easily accommodate more RISC-V extensions (e.g., floating-point, atomic) by adding new cases with appropriate ALU opcodes and control logic.

10. **Combinational Logic**:

- Implemented using `always @(*)`, ensuring that control outputs are updated immediately in response to any change in the instruction input.

Design Choices of the Control Unit

The **Control Unit** is a fundamental component of the processor that interprets the instruction opcode (and associated funct3/funct7 fields) to generate appropriate control signals required for datapath operation. The following design choices were made to ensure clarity, modularity, and support for the RISC-V RV32I/M instruction set:

1. Instruction Decoding Based on Opcode

- The instruction word is partitioned based on the **opcode**, and in some cases, **funct3** and **funct7**, to determine the type of instruction (R-type, I-type, S-type, B-type, U-type, J-type).

- This allows easy classification and generation of the control signals like `MemRead`, `MemWrite`, `regWrite`, etc.

2. Centralized ALU Control

- An `aluSelect` signal (typically 6-bit wide) is generated to drive the ALU operation (e.g., ADD, SUB, AND, OR, MUL, DIV).
- This encoding allows integration with an ALU that supports both integer and multiply/divide operations, supporting RV32M.

3. Support for RV32M Extension

- R-type instructions are further differentiated using `funct7` to support multiply/divide instructions (e.g., `MUL`, `DIV`, `REM`), aligning with the RISC-V modular extension structure.

4. Minimalist Signal Set

- Only essential control signals are generated:
  - `MemRead` / `MemWrite` to control the Data Memory.
  - `ImmSelect` to determine whether the ALU should use an immediate.
  - `PCSelect` and `Jtype` for controlling program flow.
  - `regWrite` to enable writing back to the register file.
- This keeps the Control Unit compact and focused on instruction behavior.

5. Combinational Logic Design

- The Control Unit is implemented as **pure combinational logic**, meaning the output signals change immediately in response to changes in the instruction input.
- This is crucial for pipelined designs to avoid extra timing delays in the decode stage.

6. Pattern Matching

- The design likely uses `case` statements or opcode pattern matching to ensure clean decoding logic. This helps with maintainability and readability.

Limitations of the Control Unit

The Control Unit is a critical component responsible for decoding instructions and generating control signals to guide data flow and operation selection across the processor datapath. However, like any module, it comes with certain limitations depending on its design complexity, flexibility, and integration. Below are the key limitations:

1. Instruction Set Dependency

- **Fixed Instruction Support**: The control unit is often hardcoded to recognize only a subset of RISC-V (e.g., RV32I/M). It cannot support additional instruction sets (e.g., RV64I, RV32F, or custom instructions) unless manually extended.
- **Poor Scalability**: Extending the control logic to support new instructions may require significant redesign or re-encoding of opcode decoding logic.

2. Limited Control Signal Granularity

- In simplified control units, the control signals may be overly coarse or directly tied to specific instruction types, limiting fine-tuned optimization of pipeline behavior.
- This can reduce the efficiency of resource sharing, forwarding, or hazard mitigation across stages.

3. Lack of Microprogramming or Reconfigurability

- Most undergraduate or basic designs use **hardwired control logic** with case statements or look-up tables.
- There is no microprogramming, dynamic reconfiguration, or firmware-based control, which are features common in more advanced or commercial CPUs.

4. No Support for Complex Instruction Behaviors

- Instructions requiring multicycle operations (like division, floating-point, or CSR access) might not be well-supported unless explicitly added.
- There's often no mechanism to coordinate stall and handshaking with multi-cycle units without introducing external control logic.

5. Hazard Detection and Stall Integration is External

- In modular designs, the control unit may only decode instructions and generate static signals.
- It does not manage pipeline hazards (data, control, or structural) internally—this task is often delegated to separate modules like the **Hazard Detection Unit (HDU)** or **Forwarding Unit**. This separation can lead to coordination challenges.

6. Lack of Exception and Interrupt Handling

- Basic control units often do not handle exceptions, traps, or interrupts.
- Support for ECALL, EBREAK, or illegal instruction detection is usually omitted in early designs, limiting compliance with the RISC-V privileged specification.

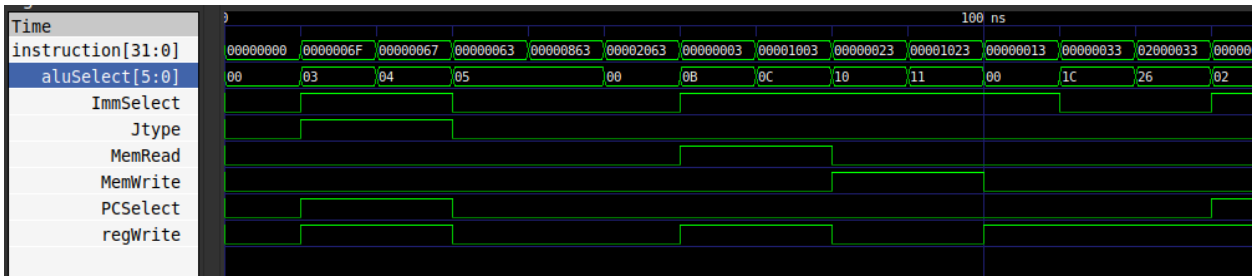
7. No Support for Dynamic Branch Prediction

- Static control logic does not include mechanisms for branch prediction or speculative execution.
- This results in frequent stalls during control hazards (e.g., branches and jumps) and reduced overall performance.

8. Synchronous Behavior Assumption

- The control unit assumes a synchronous, ideal pipeline environment.
- It may not be robust against asynchronous resets, dynamic clock domains, or irregular fetch/decode delays, making it unsuitable for complex SoC-level integration without modifications.

Sample Waveform



Register Files

1. **Purpose**:
   The Register File stores and provides fast access to the 32 general-purpose registers defined in the RISC-V architecture. It supports two simultaneous reads and one write per clock cycle, essential for pipelined processor performance.

2. **Inputs**:
   - `clk`: Clock signal to synchronize register writes.
   - `reset`: Resets all 32 registers to zero on high.
   - `RegWrite`: Enables writing to the destination register (`Rd`).
   - `Rs1`, `Rs2`: Source register addresses for reading.
   - `Rd`: Destination register address for writing.
   - `Write_data`: Data to be written into register `Rd` during the Writeback stage.

3. **Outputs**:
   - `read_data1`: Value read from register `Rs1`.
   - `read_data2`: Value read from register `Rs2`.

4. **Register Array**:
   - Implements 32 registers, each 32 bits wide: `reg_array[31:0]`.

5. **Write Logic**:
   - Synchronous with the **positive clock edge**.

- On `reset`, all 32 registers are set to `0`.
- If `RegWrite` is enabled and `Rd != 0`, `Write_data` is stored in register `Rd`.
- **Writes to register x0 are ignored**, preserving the RISC-V convention that x0 is hardwired to zero.

6. **Read Logic**:
   - Fully combinational for fast operand access.
   - If the source register is x0 (`Rs1` or `Rs2` = 0), output is always `0`.
   - Otherwise, the value from the corresponding register is output.

7. **RISC-V Compliance**:
   - Enforces the architectural rule that register `x0` always returns zero and cannot be modified.
   - Supports dual-port reads and single-port writes consistent with RISC-V pipeline needs.

8. **Timing and Pipeline Support**:
   - Data is written during the **Writeback stage** of the pipeline, and read in the **Decode stage**, supporting smooth pipelined operation.

9. **Initialization**:
   - On system reset, all registers are cleared to ensure a known state before program execution.

10. **Extensibility**:

- Easily modifiable for adding additional features like register read forwarding, register file dumps for debugging, or implementing CSR registers.

Design Choices

1. 32 General-Purpose Registers

The design includes 32 registers indexed from x0 to x31. This aligns with the RISC-V RV32I standard, ensuring compatibility with standard instruction encodings.

2. Synchronous Write Operations

Register writes occur on the rising edge of the clock to ensure stable data writes and proper timing alignment with the pipeline. Writes are only permitted when the RegWrite control signal is active and the destination register is not x0.

3. Asynchronous Read Operations

The register file supports asynchronous reads, meaning that read data becomes available immediately after selecting the source registers. This facilitates faster instruction decoding and operand fetch.

4. Enforced Zero Register

Register x0 is hardwired to zero. All writes to x0 are ignored, and any reads always return a constant zero. This feature is enforced by logic in both the write and read paths.

5. Reset Behavior

On reset, all registers in the file are cleared to zero. This ensures a known, predictable state on processor startup or system initialization.

Limitations

1. No Forwarding Support

The register file does not implement data forwarding. It cannot resolve read-after-write hazards by itself, so external hazard detection or forwarding logic is required in pipelined designs.

2. Glitch Risks from Asynchronous Reads

Asynchronous reads, while fast, can lead to timing glitches due to combinational behavior. This could affect timing closure in complex designs or high-speed implementations.

3. Single Write Port

The design supports only one write operation per clock cycle. While sufficient for in-order scalar pipelines, it is a bottleneck for superscalar or out-of-order processors that may require multiple concurrent writes.

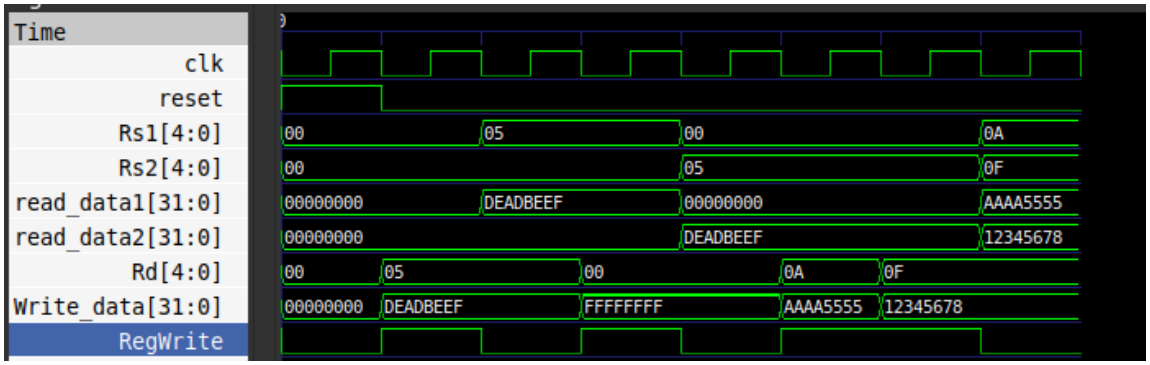
4. No Built-In Debug or Trace Capability

The register file does not include features such as register readback for debugging, test scan chains, or trace logging, which are important for large-scale verification or SoC integration.

5. Resource Utilization

Depending on synthesis tools and technology (e.g., FPGA or ASIC), the register file may consume significant logic or memory resources. Optimization strategies may be required in area-constrained designs.

Sample Waveform



Immediate Generator

The Immediate Generator module extracts the immediate operand from the instruction based on its format and sign-extends it to 32 bits to ensure correct arithmetic and address calculations. Below is a detailed description of how each immediate type is generated:

1. I-type Immediate

- **Instructions**: Arithmetic immediate (ADDI, SLTI, XORI, etc.), Load (LB, LW, etc.), and JALR.

- **Extraction**: Bits `[31:20]` of the instruction form the immediate field.
- **Sign Extension**: The most significant bit of the immediate, `instr[31]`, is replicated 20 times to fill the upper bits of the 32-bit immediate output, preserving the sign for negative values.
- **Usage**: Used as an immediate operand for ALU operations or as an offset for load instructions and jump register targets.

2. S-type Immediate

- **Instructions**: Store instructions (SB, SH, SW).
- **Extraction**: The immediate is split into two parts: the high bits `[31:25]` and the low bits `[11:7]`. These parts are concatenated as `{instr[31:25], instr[11:7]}` to form a 12-bit immediate.
- **Sign Extension**: The sign bit `instr[31]` is extended into the upper 20 bits to create a full 32-bit signed immediate.
- **Usage**: Represents the offset added to the base register to calculate the memory address for storing data.

3. B-type Immediate

- **Instructions**: Conditional branches (BEQ, BNE, BLT, BGE, etc.).
- **Extraction**: The immediate is constructed from several bits scattered across the instruction:
  - Sign bit: `instr[31]`
  - Bit 11: `instr[7]`
  - Bits 10 to 5: `instr[30:25]`
  - Bits 4 to 1: `instr[11:8]`
  - Bit 0: Always `0` (since branch offsets are multiples of 2)

- The bits are concatenated as `{instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}` forming a 13-bit immediate.
- **Sign Extension**: The sign bit `instr[31]` is extended into the upper 19 bits for a 32-bit signed immediate.

- **Usage**: Used to calculate the branch target address by adding the sign-extended immediate to the current PC.

4. U-type Immediate

- **Instructions**: LUI and AUIPC.
- **Extraction**: The immediate uses the upper 20 bits `[31:12]` of the instruction directly.
- **Sign Extension**: The lower 12 bits are set to zero, effectively shifting the immediate left by 12 bits to create a 32-bit immediate.
- **Usage**:
  - **LUI** loads the immediate into the upper 20 bits of a register.
  - **AUIPC** adds the immediate to the PC to form a PC-relative address.

5. J-type Immediate

- **Instructions**: JAL (Jump and Link).
- **Extraction**: The immediate is built from scattered bits:
  - Sign bit: `instr[31]`
  - Bits 19 to 12: `instr[19:12]`
  - Bit 11: `instr[20]`
  - Bits 10 to 1: `instr[30:21]`
  - Bit 0: Always `0` (since jump addresses are 2-byte aligned)

- The bits are concatenated as `{instr[31], instr[19:12], instr[20], instr[30:21], 1'b0}` forming a 21-bit immediate.
- **Sign Extension**: The sign bit `instr[31]` is extended into the upper 11 bits to produce a 32-bit signed immediate.
- **Usage**: Used to calculate the jump target address relative to the current PC.

6. Default Case

- If the opcode does not match any known immediate type, the output immediate defaults to zero, ensuring no unexpected operand values are produced.

Design Choices

Format Identification via Opcode

The module uses the 7 least significant bits (`instr[6:0]`) of the instruction, which represent the opcode, to identify the instruction format. This enables classification into one of the following categories:

Format-Specific Immediate Extraction

Each instruction format has a specific bit layout for the immediate field:

- **I-type (e.g., ADDI, LW):** Uses bits [31:20]. Sign-extended to 32 bits.
- **S-type (e.g., SW):** Combines bits [31:25] and [11:7], then sign-extended.
- **B-type (e.g., BEQ):** Immediate is formed from a combination of non-contiguous bits: [31], [7], [30:25], and [11:8], with a final LSB of 0.
- **U-type (e.g., LUI, AUIPC):** Takes bits [31:12] and shifts left by 12 (12 zero LSBs).
- **J-type (e.g., JAL):** Assembles immediate from [31], [19:12], [20], and [30:21], with a final LSB of 0.

All outputs are 32-bit signed values formed through appropriate bit concatenation and sign extension.

Combinational Always Block

The module is designed purely using combinational logic (`always @(*)`) to ensure immediate responsiveness to changes in the input instruction. This enables efficient integration with the decode stage in the processor pipeline.

Default Case Handling

For unsupported or unrecognized opcodes, the module defaults the immediate output to `0`.

Limitations

Lack of Detailed Sub-Decoding

Some instructions within the same opcode category require additional decoding using `funct3` or `funct7`. For example, shift instructions like `SLLI`, `SRLI`, and `SRAI` in the I-type format use a shift amount (`shamt`) instead of a sign-extended immediate. This current implementation does not differentiate between regular I-type and shift instructions.

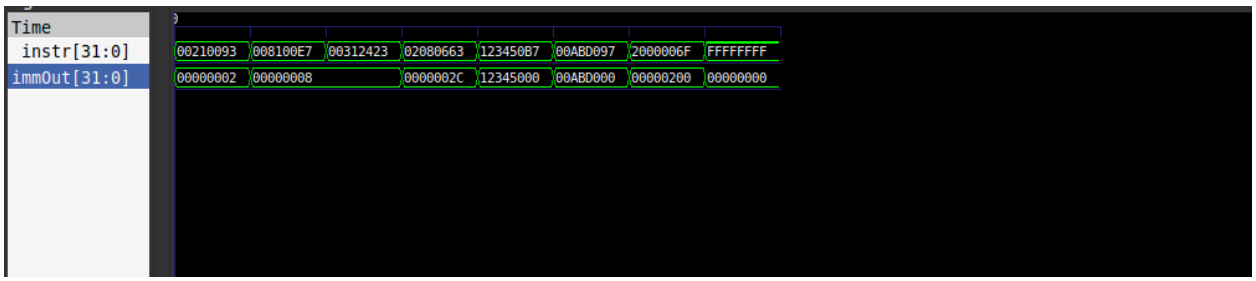
No Error Reporting for Invalid Opcodes

The module silently outputs zero for unrecognized opcodes without any error flag or exception handling. This may result in undetected issues if malformed instructions are fed into the processor.

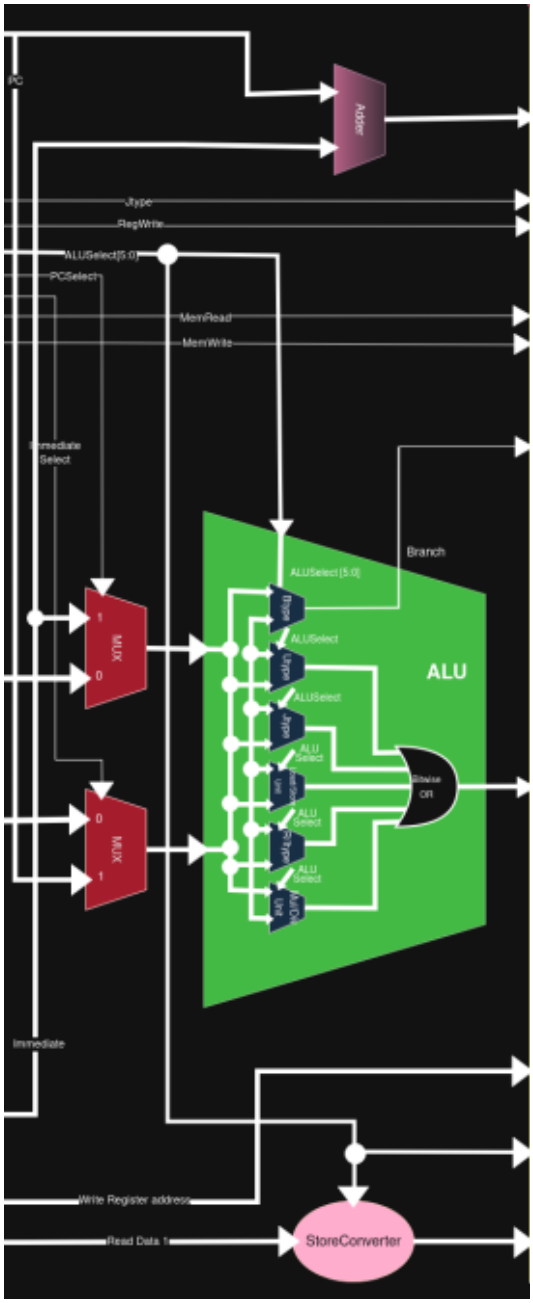
Lack of Modularity

All bit manipulation is done inline within a single `case` block. Separating immediate generation logic into reusable functions (e.g., `get_b_imm(instr)`, `get_j_imm(instr)`) would improve modularity and testability of the design.

Sample Waveform



Execution Stage



Design Overview

1. **Operand Selection via Muxes**:
    - **Mux1** selects between the PC (`PCD`) and register value (`RegOut1D`) based on `PCSelectD`. This allows PC-relative addressing (e.g., in AUIPC).

    - **Mux2** selects between an immediate (`ImmGenOutD`) and register value (`RegOut2D`) based on `ImmSelectD`. Used for ALU instructions that use immediates.

2. **Effective Address Calculation**:
    - An **Adder** computes `PC + Imm` to generate `PCPlusImmM`, the potential next PC for branches or jumps.

3. **ALU Operation**:
    - The **ALU** takes operands from the mux outputs and performs the operation specified by `ALUSelectD`.
    - It also outputs `branch_taken (BranchM)`, indicating if a branch condition is satisfied.

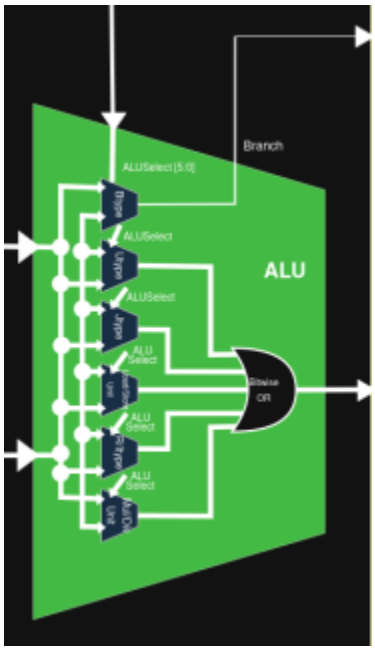
4. **Store Data Processing**:
    - A **StoreConverter** prepares the store data (`RegOut2D`) for memory access based on the store instruction type (e.g., SB, SH, SW), using `ALUSelectD`.

5. **Pipeline Signal Propagation**:
    - Control and data signals such as `WriteAddress`, `RegWrite`, `MemRead`, `MemWrite`, `ALUSelect`, and `Jtype` are forwarded to the memory stage (`M` stage) without modification.

This stage is responsible for:

- Performing arithmetic, logical, and comparison operations.
- Calculating branch/jump target addresses.
- Preparing values for memory write operations.
- Generating necessary control signals for the next stage.

ALU (Arithmetic & Logical Unit)



ALU Design Summary

1. **Purpose & Role in Pipeline:**
    - The Arithmetic Logic Unit (ALU) performs all computation-related tasks such as arithmetic operations (add, sub, mul, div), logical operations (and, or, xor), memory address calculation (for load/store), jump target generation, and branch condition checking.
    - The ALU receives two 32-bit inputs (`rs1` and `rs2`) and a 6-bit control signal (`aluSelect`) that determines the operation type.
2. **Input and Output Overview:**
    - Inputs:
        - `rs1`: First operand, typically the source register or PC.
        - `rs2`: Second operand, which can be another register or an immediate.
        - `aluSelect`: Control signal indicating the type of operation.
    - Outputs:

        - `result`: 32-bit output of the selected operation.
        - `branch_taken`: Signal indicating whether a conditional branch condition is met.

3. **Functional Submodules:**
    - **Utype**: Handles U-type instructions (`LUI`, `AUIPC`). Computes PC-relative results or immediate loading.
    - **RItype**: Handles both R-type (e.g., `ADD`, `SUB`, `AND`) and I-type arithmetic/logical instructions.
    - **muldiv**: Performs multiplication and division instructions like `MUL`, `DIV`, `REM`.
    - **LoadStoreUnit**: Computes effective memory addresses for `LW`, `SW`, etc., using base register + offset.
    - **Jtype**: Handles jump-type instructions like `JAL` and `JALR`, computing the next PC.
    - **Btype**: Evaluates branch conditions (e.g., `BEQ`, `BNE`, `BLT`) and sets the `branch_taken` signal.

4. **Operation Selection Mechanism:**
    - Each submodule is always active and computes its respective result.
    - The final `result` is computed by **bitwise OR'ing** the outputs from all submodules.
    - This assumes only one submodule will produce a meaningful (non-zero) result at any time due to exclusive `aluSelect` control decoding.

5. **Branch Handling:**
    - `Btype` module independently sets the `branch_taken` flag based on comparisons between `rs1` and `rs2`.
    - This signal is used by the control logic to determine if a branch should be taken and PC should be updated.

6. **Advantages of Modular Design:**
    - Clear separation of responsibilities per instruction type.

- Easier debugging and future extension (e.g., adding floating-point or vector operations).
- Improves design scalability and reuse in other pipeline stages.

7. **Design Limitation:**
   - All submodules are evaluated in parallel regardless of the operation type, which may lead to slightly higher power usage.
   - OR-ing all submodule outputs assumes only one will produce a non-zero value; care must be taken during testing to ensure no conflicts.

Btype - Design Summary

Overview

The `Btype` module is responsible for evaluating conditional branch instructions in the RISC-V ISA (RV32I). It determines whether a branch should be taken based on the comparison of two register operands (`rs1` and `rs2`). This decision is expressed using a single output signal: `branch_taken`.

This unit is active during the **Execute** stage of the pipeline.

Supported Branch Instructions

Instruction	ALU Select Code	Condition Checked	Description
`BEQ`	`6'b000101`	`rs1 == rs2`	Branch if equal
`BNE`	`6'b000110`	`rs1 != rs2`	Branch if not equal

`BLT`	`6'b000111`	`$signed(rs1) < $signed(rs2)`	Branch if less than
`BGE`	`6'b001000`	`$signed(rs1) >= $signed(rs2)`	Branch if greater/equal
`BLTU`	`6'b001001`	`rs1 < rs2`	Branch if less (unsigned)
`BGEU`	`6'b001010`	`rs1 >= rs2`	Branch if ge (unsigned)
`JAL`	`6'b000011`	Unconditional	Treated as always taken
`JALR`	`6'b000100`	Unconditional	Treated as always taken

JAL and JALR are not traditional B-type instructions, but they are unified under the `branch_taken` logic in this design for PC selection purposes.

Functionality

- Input:
  - `rs1`, `rs2`: Operands to be compared.
  - `aluSelect`: Control signal that selects which branch instruction is being evaluated.

- Output:

    - `branch_taken`: Boolean (1-bit) signal used in the Fetch stage to determine the next PC.

- Internally, a combinational `case` block maps the `aluSelect` code to a specific comparison.

Integration in Pipeline

- **Inputs `rs1` and `rs2`** come from the register file (via Decode stage or forwarding).
- **`aluSelect`** is provided by the control unit.
- The output `branch_taken` is used by:
    - The **PC multiplexer** to select between `PC + 4` and `PC + immediate`.
    - The **Hazard Control Unit**, which may flush or stall based on control hazards.

Design Considerations

- **Unification of JAL and JALR** under `branch_taken` simplifies PC logic but requires separate target address computation for JALR.
- **Signed vs. unsigned** comparisons are handled explicitly using `$signed(...)` in BLT/BGE logic.
- The module is purely **combinational** — no clocking or storage involved — ensuring single-cycle evaluation.

Jtype - Design Summary

Purpose

The J-type control unit is responsible for computing the next program counter (`next_pc`) in the presence of unconditional jump instructions in the RISC-V instruction set. Specifically, it supports:

- **JAL (Jump and Link)**

- **JALR (Jump and Link Register)**

These instructions alter the normal sequential flow of execution by updating the program counter based on an immediate value or a register-based target.

Inputs

- `pc` (32-bit): The current program counter, representing the address of the executing instruction.
- `imm` (32-bit): A sign-extended and possibly pre-shifted immediate value extracted from the instruction.
- `aluSelect` (6-bit): A control signal used to distinguish between the JAL and JALR instructions.

Output

- `next_pc` (32-bit): The computed target address where the control flow should jump next.

Functionality

The module computes the next PC value depending on the instruction:

- **JAL (aluSelect = `6'b000011`)**
  The next PC is calculated by adding the immediate offset to the current PC:
  ```
  next_pc = pc + imm
  ```
  This enables jumping to a position relative to the current instruction.

- **JALR (aluSelect = `6'b000100`)**
  The next PC is calculated by adding the immediate to the PC (typically a base address in a register) and clearing the least significant bit to maintain alignment:

```
next_pc = (rs1 + imm) & 32'hFFFFFFFE
```

This supports indirect jumps and function calls, where the target address is computed dynamically.

- **Default Case**
  If the control signal does not correspond to either JAL or JALR, the output `next_pc` defaults to zero. This acts as a safe fallback to prevent unintended jumps.

Design Considerations

- The immediate value used for both JAL and JALR is expected to be sign-extended and properly aligned before being fed into this module.

- The JALR path includes a bitwise AND with `0xFFFFFFFE` to enforce the RISC-V requirement that the target address be aligned to a 2-byte boundary.

Limitations

- This module assumes the immediate value has already been correctly decoded and shifted outside of this module.

- It does not handle link register updates (`rd = PC+4`), which must be handled elsewhere in the processor pipeline.

Conclusion

This J-type unit is a compact and efficient module for handling control flow changes in RISC-V via JAL and JALR instructions. Its behavior depends on the ALU selection signal and relies on correctly preprocessed immediate inputs to function accurately.

LoadStoreUnit – Design Summary

Purpose:

The `LoadStoreUnit` is responsible for computing the **effective memory address** used by load and store instructions in a RISC-V pipelined processor. It adds a base register value (`rs1`) and a 12-bit sign-extended immediate (`imm`) to determine the memory address for accessing data.

Supported Instructions:

This module handles the following **I-type (load)** and **S-type (store)** instructions:

Instruction	aluSelect	Description
`LB`	6'b001011	Load byte
`LH`	6'b001100	Load halfword
`LW`	6'b001101	Load word
`LBU`	6'b001110	Load byte unsigned
`LHU`	6'b001111	Load halfword unsigned
`SB`	6'b010000	Store byte
`SH`	6'b010001	Store halfword

`SW`	6'b010010	Store word

Inputs:

- `rs1 [31:0]`: Base address register (typically contains the starting address for memory access).
- `imm [31:0]`: Immediate value (12-bit sign-extended offset).
- `aluSelect [5:0]`: Control signal indicating the instruction type (from the control unit).

Output:

- `address [31:0]`: The final memory address computed by `rs1 + imm`.

Design Logic:

- For all supported **load and store** instructions, the effective address is calculated using:

  ```
  address = rs1 + imm
  ```

- This logic is gated by the `aluSelect` control signal. The range `6'b001011` to `6'b010010` includes all valid load/store instructions.
- For instructions outside this range, the output is reset to `32'b0` to avoid undefined memory access.

muldiv – Design Summary

Purpose:

The `muldiv` module implements the **integer multiplication and division operations** specified in the RISC-V `RV32M` extension. It performs signed and

unsigned multiplication, division, and remainder operations using two 32-bit source operands and returns the appropriate 32-bit result.

Supported Instructions:

Instruction	ALU Select (`aluSelect`)	Operation Description
`MUL`	`6'b100110`	Lower 32 bits of signed multiplication
`MULH`	`6'b100111`	Upper 32 bits of signed × signed
`MULHSU`	`6'b101000`	Upper 32 bits of signed × unsigned
`MULHU`	`6'b101001`	Upper 32 bits of unsigned × unsigned
`DIV`	`6'b101010`	Signed division (quotient)
`DIVU`	`6'b101011`	Unsigned division (quotient)
`REM`	`6'b101100`	Signed remainder
`REMU`	`6'b101101`	Unsigned remainder

Inputs:

- `rs1 [31:0]`: First source operand.
- `rs2 [31:0]`: Second source operand.
- `aluSelect [5:0]`: Operation selector from the control unit.

Output:

- `result [31:0]`: The computed result based on the selected operation.

Design Logic:

- **Multiplication Operations:**
    - `MUL`: Uses the lower 32 bits of the product:

      ```
      result = rs1 * rs2
      ```

    - `MULH`: Returns upper 32 bits of signed × signed product:

      ```
      signed_product = $signed(rs1) * $signed(rs2);
      result = signed_product[63:32];
      ```

    - `MULHSU`: Returns upper 32 bits of signed × unsigned product:

      ```
      signed_product1 = $signed(rs1) * rs2;
      result = signed_product1[63:32];
      ```

    - `MULHU`: Returns upper 32 bits of unsigned × unsigned product:

      ```
      result = unsigned_product[63:32];
      ```

- **Division and Remainder Operations:**
    - All division/remainder operations include a check for divide-by-zero, returning `0` if `rs2 == 0`.
    - `DIV`, `DIVU`: Signed and unsigned division.
    - `REM`, `REMU`: Signed and unsigned remainders.

Design Considerations:

- The implementation is **combinational** and assumes availability of a **single-cycle multiplier/divider**, which may be replaced with a **multi-cycle** or **pipelined** unit in more realistic hardware implementations.
- Division by zero returns `0`, which is consistent with typical RISC-V implementations for simplicity and safety.
- The width of intermediate products is extended to 64 bits to capture full results of multiplication.

Limitations:

- No handling for **overflow** or **division edge cases** like INT_MIN ÷ -1, which can lead to overflow in signed division.
- For synthesis on hardware (e.g., FPGA), these operations can consume significant area and may need optimization using dedicated multiplier/divider IPs or multi-cycle units.

RItype: ALU Logic for R-type and I-type Instructions

Module Overview

The `RItype` module serves as the arithmetic and logic unit (ALU) for handling both **R-type** and **I-type** instructions in a RISC-V RV32I/M processor. It operates purely as **combinational logic**, enabling seamless integration into the Execute (EX) stage of a pipelined architecture.

Functional Description

The module receives two 32-bit operands (`a` and `b`) and a 6-bit control signal `aluSelect`. Based on the value of `aluSelect`, the module performs the corresponding arithmetic or logical operation and outputs the result.

- Operand `a` typically corresponds to the contents of register `rs1`.
- Operand `b` is either the value of `rs2` (R-type) or the immediate field (I-type), decoded externally.
- The `aluSelect` signal is provided by the main control unit and identifies the specific operation to perform.

Supported Operations

The module supports the following operations from the RISC-V RV32I instruction set:

Type	Instruction	Operation Description	ALU Control (`aluSelect`)
I-type	ADDI	Addition with immediate	`010011`
I-type	ANDI	Bitwise AND with immediate	`011000`
I-type	ORI	Bitwise OR with immediate	`010111`
I-type	XORI	Bitwise XOR with immediate	`010110`
I-type	SLLI	Logical left shift	`011001`

I-type	SRLI	Logical right shift	`011010`
I-type	SRAI	Arithmetic right shift	`011011`
I-type	SLTI	Signed less-than comparison	`010100`
I-type	SLTIU	Unsigned less-than comparison	`010101`
R-type	ADD	Register addition	`011100`
R-type	SUB	Register subtraction	`100100`
R-type	AND	Bitwise AND	`100011`
R-type	OR	Bitwise OR	`100010`
R-type	XOR	Bitwise XOR	`100000`
R-type	SLL	Logical left shift	`011101`
R-type	SRL	Logical right shift	`100001`

R-type	SRA	Arithmetic right shift	`100101`
R-type	SLT	Signed less-than comparison	`011110`
R-type	SLTU	Unsigned less-than comparison	`011111`

Design Decisions and Justifications

- **Shift operations** use only the lower 5 bits of `b` (i.e., `b[4:0]`) to comply with RISC-V's 5-bit shift amount specification.
- **Signed operations** like `SRA`, `SRAI`, `SLT`, and `SLTI` use Verilog's `$signed()` operator to ensure correct behavior.
- The ALU is designed to be **modular and easily extendable**, allowing integration of future instruction formats (e.g., RV64I or RV32M operations in a separate unit).

Limitations

- The module does **not support** multiplication, division, or floating-point operations.
- It assumes that the control unit and decoder provide the correct operand type (immediate or register) in `b`.

Utype: ALU Logic for U-type Instructions (LUI and AUIPC)

Module Overview

The `Utype` module implements the arithmetic logic required for handling **U-type instructions** in the RISC-V RV32I architecture, specifically the **LUI (Load Upper Immediate)** and **AUIPC (Add Upper Immediate to PC)** instructions.

Functional Description

The module takes as inputs:

- A 32-bit **program counter (PC)** value.
- A 32-bit **immediate value** (`imm_u`), which represents the upper 20 bits of the instruction immediate, left-shifted by 12 bits as specified by the RISC-V standard.
- A 6-bit control signal `aluSelect` to specify the operation.

Based on the `aluSelect` input, the module performs the following:

- **LUI (`aluSelect = 000001`)**: Directly outputs the immediate value (`imm_u`). This instruction loads a 20-bit immediate into the upper 20 bits of the destination register, filling the lower 12 bits with zero.
- **AUIPC (`aluSelect = 000010`)**: Outputs the sum of the PC and the immediate (`pc + imm_u`). This instruction is used to compute PC-relative addresses by adding the immediate to the current PC value.

Design Decisions and Justifications

- The module uses combinational logic with an `always @(*)` block to select the output based on the control signal.
- The immediate value is expected to be pre-shifted by 12 bits externally, simplifying the internal logic.
- The use of `aluSelect` enables easy extension to other U-type instructions if needed in the future.

StoreConverter Module – Design Summary

1. Introduction

In a RISC-V processor, store instructions (`SB`, `SH`, and `SW`) are responsible for writing data from a register into memory. The `StoreConverter` module is designed to ensure correct data formatting for these store operations, particularly for instructions that write less than 32 bits (i.e., byte or halfword stores). This module performs data masking according to the instruction type so that only the relevant portions of the 32-bit data word are written to memory.

2. Purpose and Functionality

The primary function of the `StoreConverter` is to convert or mask the 32-bit input data (`inputData`) based on the `aluSelect` signal, which specifies the type of store instruction:

- For `SB` (Store Byte), only the least significant byte (bits [7:0]) is retained.
- For `SH` (Store Halfword), only the least significant halfword (bits [15:0]) is retained.
- For `SW` (Store Word) or other unsupported types, the full 32-bit word is passed unchanged.

The module's output, `outputData`, is then forwarded to the memory subsystem for the actual write operation.

3. Interface Description

- **Inputs:**
    - `inputData [31:0]`: The data value to be written to memory.
    - `aluSelect [5:0]`: A control signal that determines the type of store instruction.
- **Output:**

- `outputData [31:0]`: The correctly masked data that should be stored in memory.

4. Control Behavior

The behavior of the module is governed entirely by the `aluSelect` signal:

Instruction	aluSelect (Binary)	Action
SB	010000	Retain bits [7:0]
SH	010001	Retain bits [15:0]
SW / others	Any other	Pass full 32-bit data

Masking is achieved through simple bitwise AND operations using constant masks (`0x000000FF` for byte, `0x0000FFFF` for halfword), ensuring correctness and hardware simplicity.

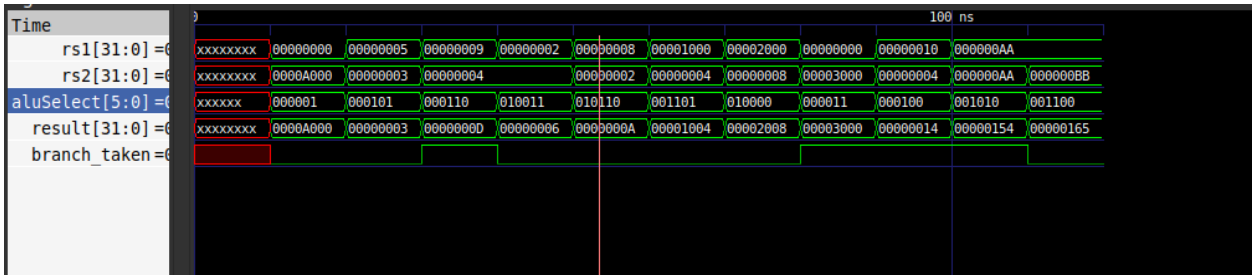
5. Design Considerations

- **Endianness:** The module assumes a little-endian architecture, which is typical for RISC-V systems.
- **Alignment:** It does not handle memory alignment checks. Alignment-related errors should be managed externally by the memory or control unit.
- **Simplicity:** The module is kept minimal and combinational to avoid unnecessary latency in the pipeline.

6. Conclusion

The `StoreConverter` is a utility module that enables correct and efficient data storage in RISC-V memory operations. By isolating the data-width formatting logic into a separate component, the processor design remains modular and easier to maintain. This module is particularly critical in systems where precise memory access behavior must be enforced to comply with the RISC-V specification.

Sample Waveform



Memory Stage

The Memory stage is the fourth stage in a standard RISC-V pipelined processor (IF → ID → EX → MEM → WB). Its core responsibilities include accessing memory for load (`LW`, `LH`, `LB`) and store (`SW`, `SH`, `SB`) instructions and passing necessary data and control signals to the Writeback stage. This design integrates a `DataMemory` unit and connects pipeline registers that carry values from the Execute stage to the next stages.


PC + Immediate
Jtype
RegWrite
Branch
MemWrite
MemRead
MemRead
MemWrite
Address [31:0]
Data [31:0]
Data
Memory
Write
Data
ALUSelect
Write Register address
Read Data 1 after StoreConverting

Data Memory

Design Summary: Data Memory Module

The Data Memory module is a synchronous memory block designed to support typical load and store operations for a 32-bit RISC-V processor pipeline. It implements a small memory array with 64 words, each 32 bits wide.

Key Features:

- **Memory Array:**
  The memory consists of 64 entries, each 32 bits wide, implemented as a register array.
- **Synchronous Write:**
  Memory write operations occur on the rising edge of the clock when the write enable (`MemWrite`) signal is asserted. The data to be written (`Write_data`) is stored at the specified address (`read_address`). This ensures predictable timing and proper synchronization with the processor clock.
- **Asynchronous Read:**
  The memory supports asynchronous read operations. When the read enable (`MemRead`) is asserted, the data at the given address is immediately output on `MemData_out`. If `MemRead` is deasserted, the output is zero, preventing invalid data usage.
- **Reset Behavior:**
  Upon an active reset signal, the memory content is cleared, setting all entries to zero. This initializes the memory to a known state, avoiding undefined behavior after reset.

Functionality Overview:

- The module supports load and store instructions by calculating the effective address and reading or writing data accordingly.
- The use of separate control signals (`MemRead`, `MemWrite`) allows precise control over memory access operations.

- Addressing is direct and limited to 64 entries, suitable for small embedded processor designs or test environments.

Design Implications:

- The design provides reliable and synchronized data storage critical for correct program execution.
- By clearing memory on reset, the module supports clean system initialization.
- The separation of read and write control enables concurrent control over memory operations in the pipeline stages.

This memory module forms the backbone for data storage during program execution in a simple pipelined RISC-V processor, facilitating load/store instructions and ensuring data integrity and synchronization with the processor clock.

Limitations of the Data Memory Module

1. **Fixed Memory Size**
    - Most implementations define a small, fixed-size memory (e.g., 64 or 256 words), which is unrealistic for real-world applications.
      This limits the addressable data space, preventing programs with larger datasets from running.
2. **No Memory-Mapped I/O**
    - If your memory is purely RAM without mapping to peripheral addresses (e.g., UART, GPIO), it cannot support I/O operations directly.
    - This limits its utility in embedded systems or SoC environments.
3. **No Support for Unaligned Access**
    - Many simplified designs do not handle unaligned memory accesses (e.g., `LW` from unaligned addresses).

- This violates the RISC-V specification, which allows trap handling or defined behavior for such cases.

4. **No Access Latency Modeling**

   - In real hardware, memory takes several cycles to respond.
   - Simple models assume 1-cycle read/write, which is overly optimistic and doesn't reflect realistic pipeline stalling.

5. **Lack of Byte/Word Granularity Control**

   - If the memory only supports 32-bit word accesses and lacks proper masking for byte (`SB`, `LB`) or halfword (`SH`, `LH`) operations, data may be incorrectly stored or read.
   - Missing support for `funct3` decoding means byte-level memory operations are not implemented accurately.

6. **No Bus Interface**

   - A standalone data memory module typically lacks standardized bus protocols like AXI, AHB, or TileLink.
   - This makes integration with larger SoC platforms or simulators (like Spike or QEMU) more difficult.

7. **No Exception Handling**

   - If the memory access is out-of-bounds, misaligned, or illegal, many simple memories do not raise exceptions or signals.
   - This prevents proper trap handling required by the RISC-V privilege architecture.

8. **No Cache Support**

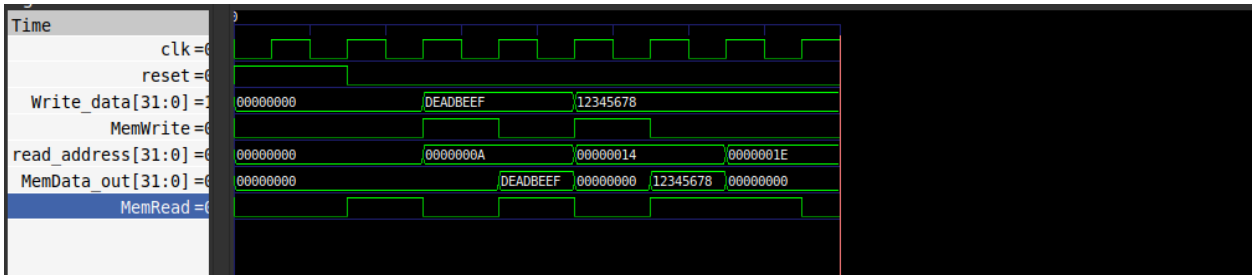
   - Without a cache layer (instruction or data cache), memory latency is not optimized.

- This affects performance when scaling to more realistic benchmarks or when implementing multi-cycle or pipelined memory.

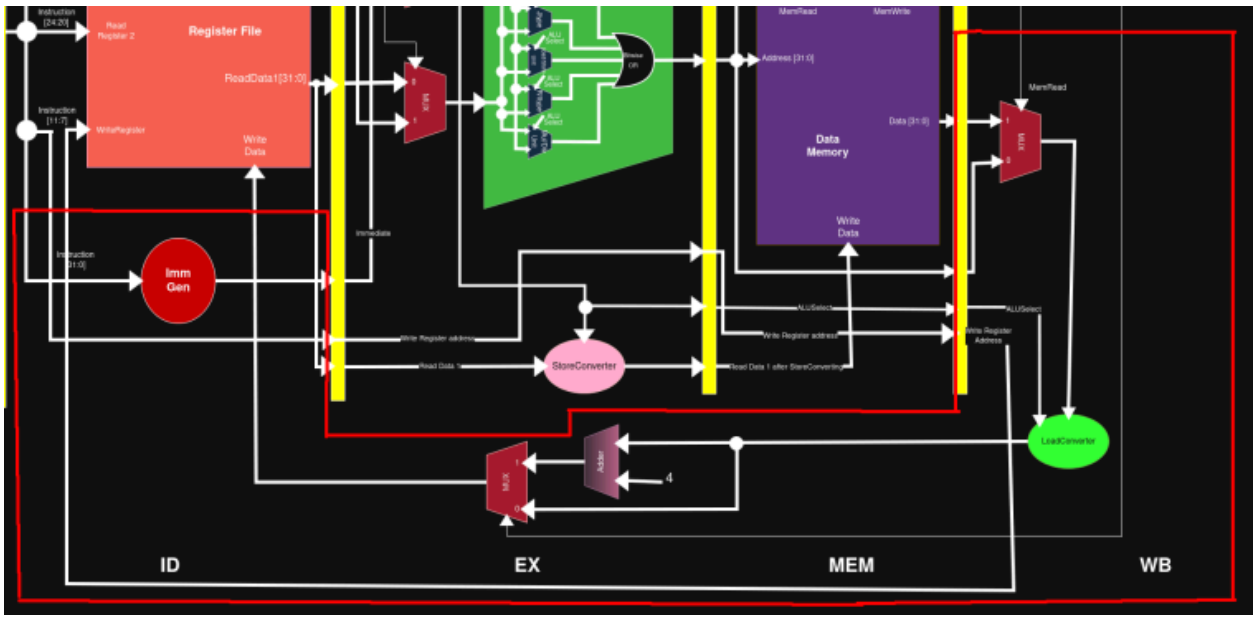
9. **No Memory Protection (MMU)**

   - The module does not support physical or virtual memory translation (i.e., no MMU or PMP).
   - It is thus only usable in M-mode (machine mode) programs and cannot support full Linux or OS-level support.

Sample Waveform



WriteBack Stage



Design Overview: WriteBack Cycle Module

The WriteBack cycle module is responsible for selecting and preparing the data that will be written back to the processor's register file during the final stage of the pipeline. This involves choosing between data from memory, ALU computation results, and handling jump instruction related address updates. The design uses several internal modules and multiplexers to achieve this functionality:

Key Modules and Components

1. **Mux1 (First Multiplexer)**
    - Selects between `DataMemOutM` (data read from memory) and `ALUOutM` (ALU result).
    - Controlled by the `MemReadM` signal, which indicates if the instruction is a load.
    - This ensures the correct source of data is passed forward depending on whether the instruction requires data from memory or ALU computation.

2. **Adder Module**
    - Adds a constant value of 4 to the output of the `LoadConverter`.
    - This operation is necessary for jump-type instructions where the return address (PC + 4) must be written back to the register.
    - This allows the processor to correctly handle function calls and jumps by storing the proper return address.

3. **Mux2 (Second Multiplexer)**
    - Selects between the incremented address from the `Adder` and the load/ALU converted data from the `LoadConverter`.
    - Controlled by the `JtypeM` signal, which identifies if the current instruction is a jump.
    - For jump instructions, it forwards the PC + 4 value; for others, it forwards the load or ALU data.

4. **Design Summary: LoadConverter Module**

The LoadConverter module is responsible for processing the data loaded from memory before it is written back to the register file. It interprets the data based on the type of load instruction and performs the necessary sign or zero extension to ensure that the data is correctly formatted as a 32-bit word.

Functional Description

- **Input Data:** The module receives raw data from the memory (`inputData`), which can represent a byte, halfword, or word depending on the load instruction.

- **Control Signal:** The `aluSelect` input determines the type of load instruction currently being executed, such as LB (load byte), LH (load halfword), LBU (load byte unsigned), or LHU (load halfword unsigned).

- **Data Conversion:**

    - For signed loads (LB, LH), the module extends the sign bits of the smaller data size to fill the upper bits of the 32-bit output.

    - For unsigned loads (LBU, LHU), the module zero-extends the smaller data size by masking out the upper bits.

    - For other instructions or default cases, it passes the input data through unchanged.

- **Output Data:** The processed and properly extended 32-bit data is sent to the write-back stage for storage in the register file.

Design Choices

1. **Case-Based Decoding**

    - The design uses the `aluSelect` signal to determine the type of load operation (`LB`, `LH`, `LBU`, `LHU`).

- Each case masks or sign-extends the data as required, ensuring compliance with RISC-V load instruction formats.

2. **Simple Combinational Logic**

   - The module is implemented using an `always @(*)` block, making it purely combinational.

   - This ensures there is no latency introduced by the Load Converter; the output updates as soon as the input changes.

3. **Pass-Through for Word Loads**

   - For standard word loads (`LW`), the module directly outputs the input data without modification, simplifying the design.

4. **Sign vs. Zero Extension**

   - For signed loads (`LB`, `LH`), the module ensures sign extension by OR-ing with predefined masks (`32'hffffff00` or `32'hffff0000`).

   - For unsigned loads (`LBU`, `LHU`), the module masks higher bits to zero.

5. **Compact and Modular Design**

   - It is a separate, standalone module, allowing easy reuse and testing independently of the memory system.

Limitations

1. **Limited to Basic Load Types**

   - The current design only supports `LB`, `LH`, `LBU`, and `LHU`. It does not include custom or future load operations that might require additional handling.

2. **Hardcoded Masks**

    - The design uses fixed masks for sign and zero extension. While sufficient for 32-bit RISC-V, this approach requires manual changes for other architectures (e.g., 64-bit systems).

3. **No Error Detection**

    - The module assumes that `aluSelect` is always valid. If an invalid control signal is provided, it defaults to a pass-through operation without generating any error or warning.

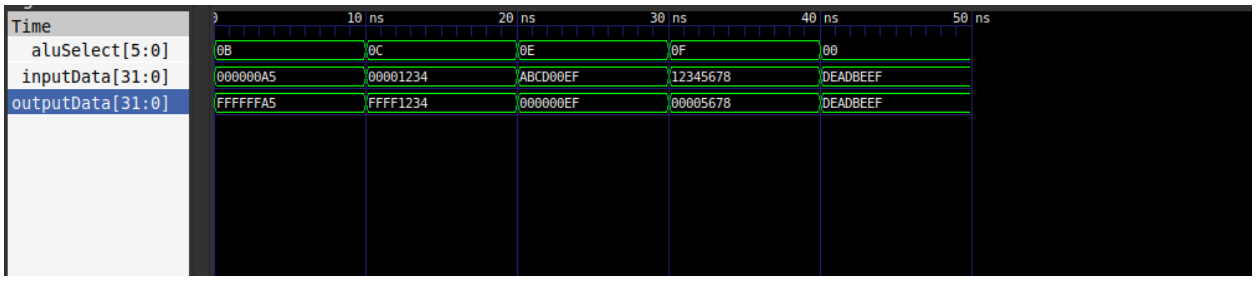
4. **No Misalignment Handling**

    - The module does not consider alignment issues (e.g., when loading bytes/halfwords from unaligned addresses). This responsibility is left to the memory or control logic.

5. **Dependence on `aluSelect` Accuracy**

    - Any incorrect `aluSelect` value from the Control Unit will directly lead to incorrect data extension or truncation, as the Load Converter does not verify its correctness.

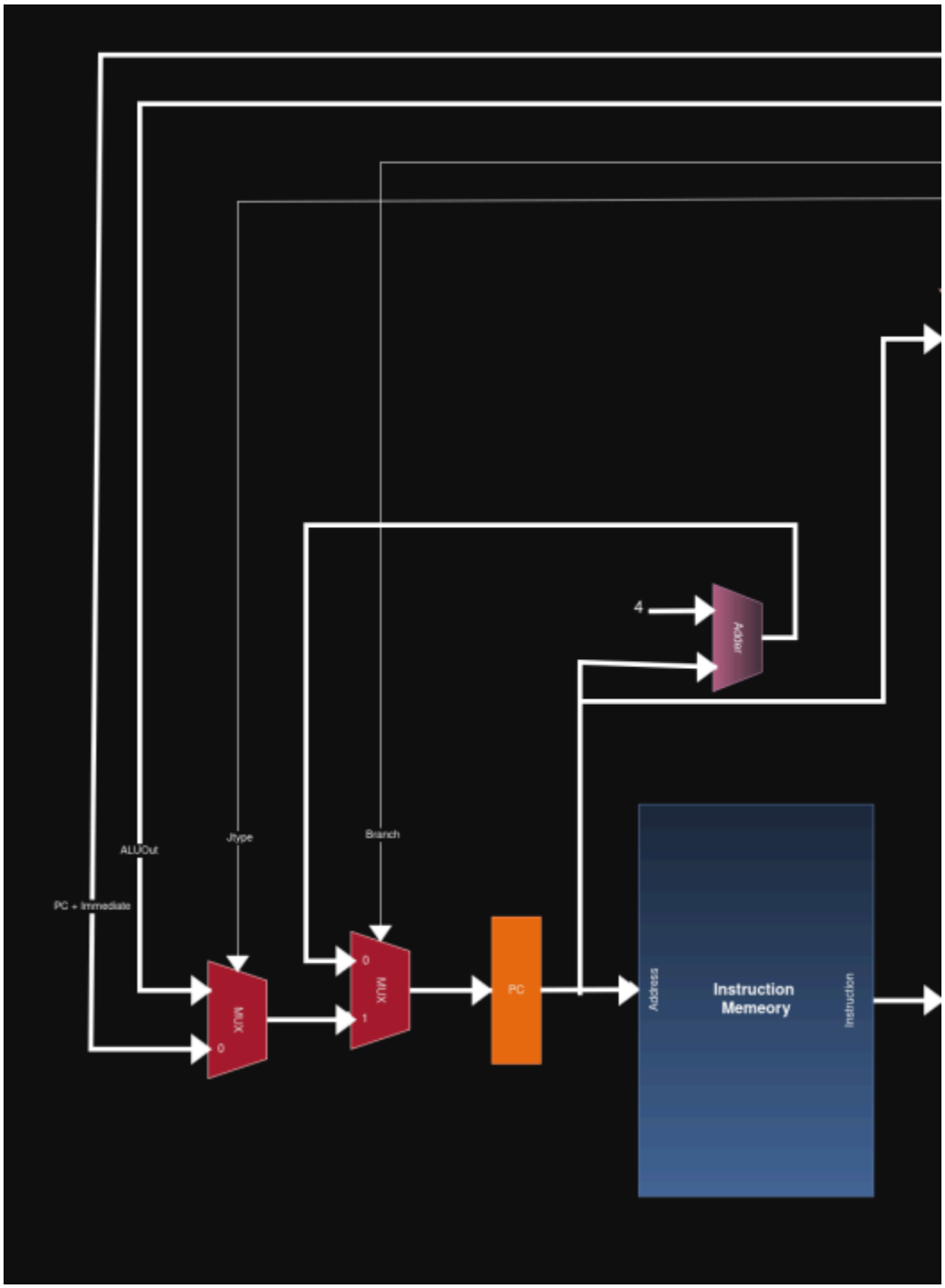
Sample Waveform



Lab Part 03 - Integration

Integration of Each Stage

Instruction Fetch Stage Integration



The **Fetch stage** is the first stage of your pipelined RISC-V processor. It is responsible for:

1. **Computing the next PC (Program Counter)**
2. **Fetching the instruction** from instruction memory using that PC
3. **Passing the PC and instruction** to the Decode (ID) stage

Functional Overview of Integration

The `Fetch_cycle` module integrates several submodules to control how the next instruction address is determined, especially during **branches** and **jumps**, and fetches the instruction from memory accordingly.

1. Program Counter (PC) Logic

- The **current PC** is stored in the `Program_Counter` module.
- The **next PC (`PCInF`)** is selected based on:
    - **Sequential execution** (PC + 4),
    - **Jump (`JAL`, `JALR`)**, or
    - **Conditional Branch** (e.g., `BEQ`, `BNE`).
- Two 2-to-1 **multiplexers (Mux)** choose the correct next PC:
    - `JtypeEMux`: selects between `ALUOutM` (JALR target) and `PCPlusImmM` (JAL target).
    - `BranchEMux`: selects between jump target and `PC + 4` depending on `BranchM`.

2. PC + 4 Calculation

- `Adder` computes `PC + 4`, which is used for sequential execution.

3. Instruction Memory Access

- `Instruction_Mem` takes the current PC and outputs the corresponding 32-bit instruction.

4. Outputs to Decode Stage

- `PCD` – current PC

- `InstrD` – current instruction
  These are forwarded to the **Decode (ID) stage** through the `IF/ID` pipeline register.

Design Choices

- **Two-stage muxing** is used to handle **jumps and branches** in a clean and modular way.
- Branch/jump control signals (`BranchM`, `JtypeM`) come from **Memory stage**, ensuring the pipeline waits for complete condition evaluation before redirecting PC.
- **Adder and Instruction_Mem** are separated to keep modularity and support reuse in testbenches.

Limitations

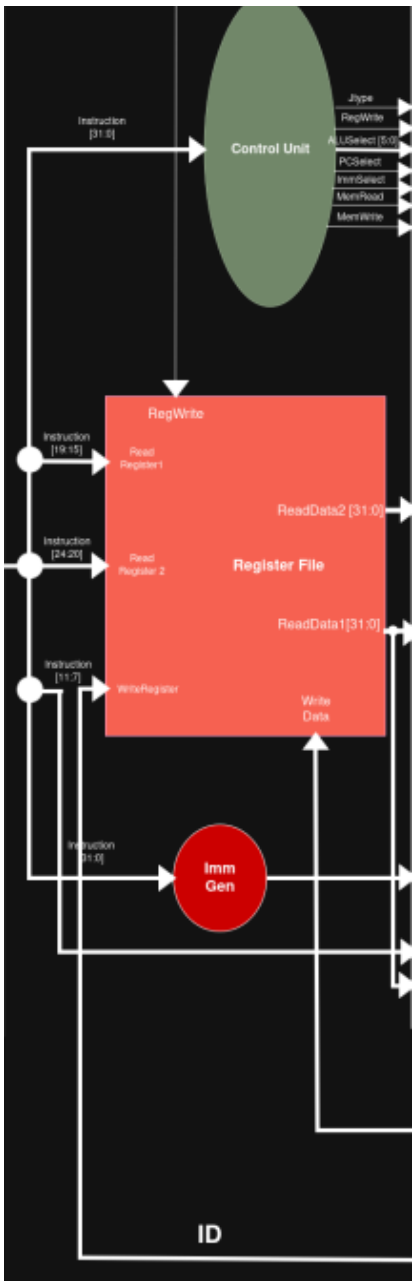
- **No stall or flush logic** is present:
    - This module assumes the rest of the pipeline handles hazards (e.g., load-use, control hazards).
    - A branch misprediction may result in **wrong instruction fetched** without flush logic.

- **Instruction memory is assumed to be synchronous** and read instantly, which may not reflect real-world memory delays.

- **No instruction cache (I-Cache)** – limits performance for real-world applications.

- **Jumps and branches resolved late (MEM stage)** – incurs a delay of several cycles for control instructions.

Instruction Decode (ID) Stage – Integration and Functionality



The **Decode Stage** is responsible for decoding the fetched instruction, reading the source registers, generating the immediate value, and preparing control

signals for the later stages in the pipeline. In your design, this is implemented in the `decode_cycle` module.

Submodule Integration and Data Flow

1. Control Unit (`ControlUnitD`)

- **Inputs**: `instructionF` (full 32-bit instruction)
- **Outputs**:
  - `aluSelect` (6-bit ALU operation selector)
  - `MemWrite`, `MemRead`
  - `ImmSelect`, `PCSelect`
  - `RegWrite`, `Jtype`
- This module decodes the opcode and funct3/funct7 fields to generate appropriate **control signals** for all downstream units (ALU, Memory, etc.).

2. Register File (`Reg_File_D`)

- **Inputs**:
  - `Rs1`, `Rs2` extracted from instruction (`instructionF[19:15]` and `instructionF[24:20]`)
  - `Rd` (write-back address) from WB stage (`WriteAddressW`)
  - `writeDataW` (data to write back)
  - `RegWriteW` signal to control write enable

- **Outputs**:
  - `read_data1` → `ReadOut1E`
  - `read_data2` → `ReadOut2E`
- Reads source registers for the current instruction and writes back results from the **Writeback stage**.

3. Immediate Generator (`ImmGen_D`)

- **Input**: full instruction
- **Output**: sign-extended immediate value (`ImmGenOutE`)

- This module decodes instruction format types (I, S, B, U, J) and produces the appropriate **sign-extended immediate** value based on `ImmSelect`.

4. Output Assignments

- `WriteAddressE = instructionF[11:7]` is the destination register for write-back.
- `PCE = PCF` forwards the current program counter to the Execute stage.

Limitations

1. **No hazard detection or stalling logic**:
    - Load-use hazards or RAW dependencies must be handled separately (likely in a dedicated Hazard Detection Unit).

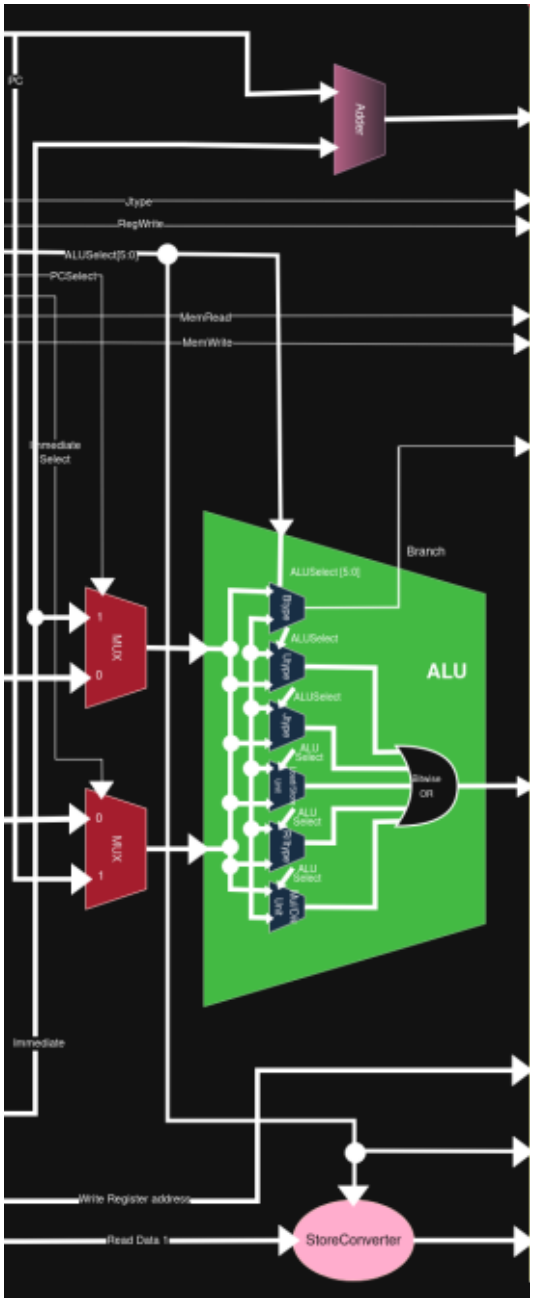
2. **No immediate multiplexer**:
    - Immediate selection logic (`ImmSelect`) is assumed to be handled implicitly by downstream logic.

3. **No forwarding logic in decode stage**:
    - RAW hazards that require forwarding are not addressed here and must be handled in Execute or by dedicated Forwarding Unit.

4. **Write-back only on positive edge**:
    - Write to register file happens synchronously with the clock, which may delay updates if data arrives late from MEM/WB.

Execution Stage Integration Summary

The **Execution Stage** in the pipelined RISC-V processor design performs core arithmetic/logic operations, address calculations, branch decisions, and data preparation for memory operations. It integrates multiple modules with clear data and control flow derived from the Decode stage and forwards results to the Memory stage.



1. Design Objectives

- Perform ALU operations with operands selected via control signals.
- Support immediate and register-based operations.
- Evaluate branch conditions.
- Prepare memory write data in correct format.
- Propagate control and data signals cleanly to the MEM stage.

2. Integrated Modules

a) Mux1 and Mux2

- **Function**: Select operands for the ALU.
    - Mux1 chooses between `PCD` and `RegOut1D` based on `PCSelectD`.
    - Mux2 selects between `ImmGenOutD` and `RegOut2D` based on `ImmSelectD`.

- **Design Choice**: These MUXes provide operand flexibility for both arithmetic and branch operations.

b) ALU

- **Function**: Performs operations like ADD, SUB, AND, OR, shifts, comparisons, etc., determined by `ALUSelectD`.
- **Outputs**:
    - `ALUOutM`: result of the computation.
    - `BranchM`: boolean output used for conditional branch decision.
- **Design Choice**: Designed to support all RV32I/M instruction operations with parameterized control via `ALUSelect`.

c) Adder

- **Function**: Computes target address for branch or jump operations as `PCD + ImmGenOutD`.

- **Design Choice**: Maintains a separate adder to prevent overloading the ALU and to ensure clean branch address calculation.

d) StoreConverter

- **Function**: Adjusts/store values based on instruction type (e.g., SB, SH, SW).
- **Inputs**: Raw register data and ALU operation type.
- **Outputs**: `StoreCounterOutM` formatted for memory storage.

- **Design Choice**: Keeps memory interface logic simple by pre-formatting data.

3. Signal Forwarding

- All relevant control and data signals (e.g., `WriteAddressD`, `RegWriteD`, `MemReadD`, etc.) are directly forwarded to the MEM stage with no transformation.
- **Purpose**: Simplifies control and ensures consistent instruction behavior.

4. Design Choices

- **Modularization**: Each subtask (e.g., operand selection, ALU operation, address calculation) is encapsulated in a separate module.
- **Clean Separation**: Avoids tight coupling between ALU and branch logic, aiding testability and debugging.
- **Forwarding Readiness**: Mux design enables future integration with forwarding logic if needed.
- **Pipeline-Friendly**: Maintains high throughput by minimizing combinational delay through distinct logic paths.

5. Limitations

- **No Hazard Mitigation**: Assumes hazard handling is done externally. Raw operand dependencies may cause incorrect behavior without forwarding.
- **Limited ALU Scope**: Only supports pre-defined ALU operations; extending instruction types requires modification of ALU and control signal design.
- **No Bypass Support for StoreConverter**: Data must be correctly forwarded before entering this stage, else stale data may be used.
- **PC-based Mux Selection**: Could be susceptible to incorrect branch prediction or delay if control hazard resolution is not correctly implemented.

Integration of Memory Stage

Purpose of Memory Stage Integration

The Memory stage is the fourth stage in a classic 5-stage RISC-V pipeline. Its key functions include:

- Performing memory read/write operations (for `lw`, `sw`, etc.)
- Forwarding data and control signals to the WriteBack stage
- Passing branch/jump target information and control signals back to the Fetch stage to aid in PC updates

Key Data and Control Signals Passed

- **From Execute Stage**:
  - ALU result (used as memory address)
  - Store data (from the Store Converter)
  - Control signals: `MemRead`, `MemWrite`, `Branch`, `Jtype`, `RegWrite`, etc.

- **To WriteBack Stage**:
  - Memory output data
  - ALU result
  - Write-back destination register address
  - Control signals for register write

- **To Fetch Stage**:
  - Branch target address
  - Branch and jump indicators

Role in Pipeline Flow

- Acts as a bridge between the Execute and WriteBack stages
- Communicates with the Data Memory module to read/write memory contents
- Supplies data to be written back into the register file
- Supplies control flow redirection data (e.g., `PCPlusImm`, `Branch`) to the Fetch stage to handle jumps and branches

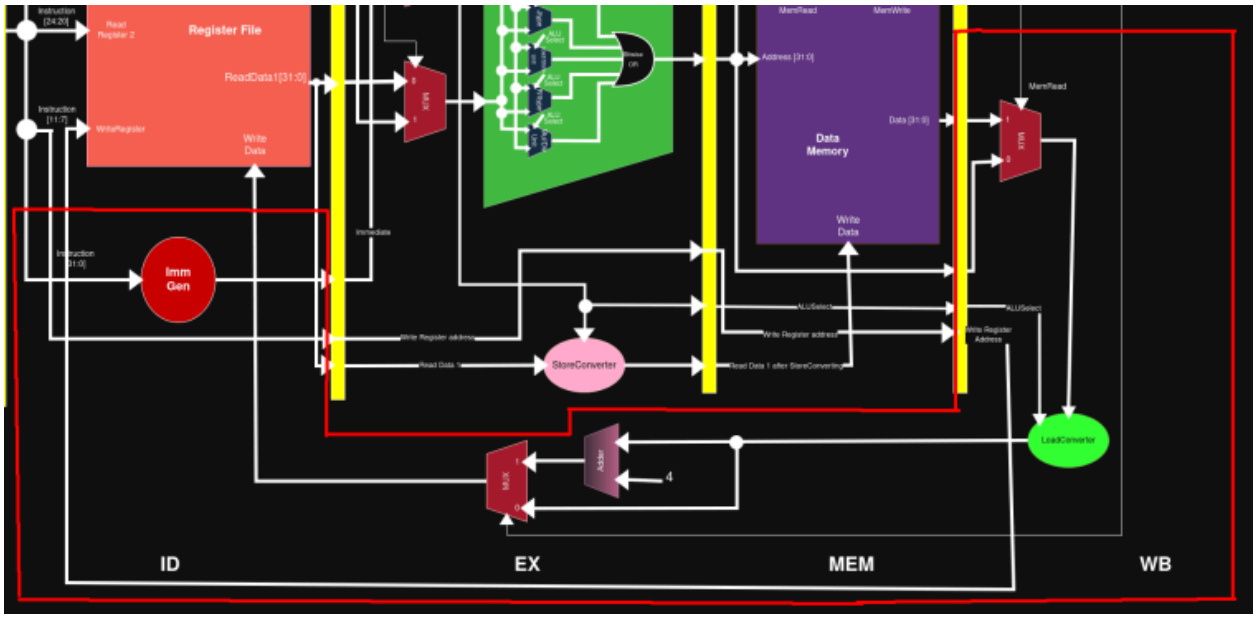
Design Decisions

- Forwarding branch/jump targets and control signals directly to Fetch to minimize latency
- Forwarding all relevant signals to WriteBack stage for accurate register file updates
- Modular encapsulation of memory and pipeline register functionality for clarity and reusability

Limitations

- No internal hazard detection or stall control; relies on upstream hazard units
- Late branch resolution increases the penalty for mispredictions
- Assumes a single-cycle memory with no wait-states or memory hierarchy
- Does not support exception handling or unaligned memory accesses

Integration Summary of the WriteBack Stage



Purpose of the WriteBack Stage

The WriteBack (WB) stage is the final step in the RISC-V pipeline. Its core responsibility is to **return computed or loaded values back to the register file**, completing the instruction's execution lifecycle.

Key Inputs

The stage receives the following from the Memory stage (MEM/WB pipeline register):

- **`DataMemOutM`**: Data loaded from memory (for `lw`, etc.)
- **`ALUOutM`**: ALU result (for arithmetic instructions)
- **`ALUSelectM`**: ALU operation selector used for load-type format decoding
- **`JtypeM`**: Indicates if the instruction is a jump (e.g., `jal`)
- **`MemReadM`**: Indicates whether the data should come from memory or ALU
- **`RegWriteM`**: Enables writing to the register file
- **`WriteAddressM`**: Destination register address

Internal Functional Units

- **Mux1**: Chooses between `DataMemOutM` and `ALUOutM` based on `MemReadM`
- **LoadConverter**: Converts raw data from memory to proper size/type (e.g., byte, halfword, signed/unsigned)
- **Adder**: Used to compute `PC + 4` for return address (`jal`, `jalr`)
- **Mux2**: Chooses between computed address (`PC + 4`) and data result based on `JtypeM`

Key Outputs

- **`RegInDataD`**: Final data to write into the register file
- **`WriteAddressD`**: Destination register number
- **`RegWriteD`**: Register write enable signal

Role in Pipeline Integration

- Completes instruction execution by updating the register file with the correct result
- Supports integration of control flow instructions (`jal`, `jalr`) by preparing `PC + 4` as the return value
- Provides proper data formatting through the `LoadConverter` to ensure correct memory value interpretation

Design Choices

- **Decoupled load data formatting** using a dedicated `LoadConverter` to modularize functionality
- **Two-stage muxing logic** to handle memory-vs-ALU selection and jump-specific redirection
- **Adder inclusion** enables accurate handling of return addresses in jump instructions

Limitations

- Assumes correct and hazard-free data has been passed from previous stages
- No built-in error handling for misaligned memory accesses
- Pipeline stalls or flushes are managed externally (not handled within this stage)
- No bypass or forwarding logic; relies on upstream stages for hazard resolution

Design Overview of Inter-Stage Pipeline Registers

Purpose

Inter-stage pipeline registers are crucial in a pipelined processor to **hold and transfer data and control signals** between pipeline stages on each clock cycle.

They **preserve intermediate results**, **maintain instruction flow**, and **support hazard management** by isolating each pipeline stage's operation.

Pipeline Registers Implemented

Your processor design includes the following key pipeline registers:

Pipeline Register	Connects	Purpose
IF/ID	Fetch → Decode	Stores fetched instruction and PC. Enables decoding in the next cycle.
ID/EX	Decode → Execute	Holds decoded control signals, operand values, immediate values, and instruction fields.
EX/MEM	Execute → Memory	Transfers ALU result, branch/jump decisions, memory operation flags, and destination register info.
MEM/WB	Memory → WriteBack	Carries memory or ALU results, register write-back info, and jump data to the register file.

Design Choices

- **Granular fields**: Control, data, and address signals are **separated and explicitly declared**, improving readability and debugging.
- **Signal grouping**: Control and data paths are grouped logically (e.g., ALU outputs with control flags), aiding clarity and modular verification.
- **Clocked behavior**: Each pipeline register module uses synchronous logic with `clk` and `reset`, ensuring **deterministic pipeline progression**.

- **Flush and Stall Handling**: Hazard detection and flushing (e.g., due to branch misprediction) are supported by zeroing or bypassing the relevant registers.

Limitations

- **Fixed-width data buses** may limit flexibility in future extensions (e.g., support for wider instructions or CSR operations).
- **No ECC or error detection**: Registers assume error-free data transfer.
- **No internal forwarding**: Forwarding logic resides in separate modules, so pipeline registers are strictly passive.

Benefits of the Design

- **Improved throughput** by enabling parallel instruction execution
- **Well-isolated stages**, allowing modular debugging and easier testing
- **Scalable structure** suitable for adding new features (e.g., CSR, interrupts, or floating-point units)

Testing for Instructions after Overall Integration

First we test for I-type instructions such that any hazard is avoided

```
00500093  // ADDI  x1, x0, 5

00102113  // SLTI  x2, x0, 1

00103193  // SLTIU x3, x0, 1

00504213  // XORI  x4, x0, 5

00806293  // ORI   x5, x0, 8

0040F313 // ANDI  x6, x1, 4

00109393  // SLLI  x7, x1, 1

0010D413 // SRLI  x8, x1, 1

4010D493 // SRAI  x9, x1, 1
```

Waveform for these instructions

After 85000 ps (85ns) onwards, these instructions start to store data on registers (after completing the pipeline)

```
=== Cycle 85000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000000
x3 = 0x00000000
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
x8 = 0x00000000
x9 = 0x00000000
x10 = 0x00000000
```

ADDI x1, x0, 5

```
=== Cycle 95000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000000
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
x8 = 0x00000000
x9 = 0x00000000
x10 = 0x00000000
```

SLTI x2, x0, 1

```
=== Cycle 105000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
x8 = 0x00000000
x9 = 0x00000000
x10 = 0x00000000
```

SLTIU x3, x0, 1

```
=== Cycle 115000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
x8 = 0x00000000
x9 = 0x00000000
x10 = 0x00000000
```

XORI x4, x0, 5

```
=== Cycle 125000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000000
x7 = 0x00000000
x8 = 0x00000000
x9 = 0x00000000
x10 = 0x00000000
```

ORI   x5, x0, 8

```
=== Cycle 135000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x00000000
x8 = 0x00000000
x9 = 0x00000000
x10 = 0x00000000
```

ANDI x6, x1, 4

```
=== Cycle 145000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000000
x9 = 0x00000000
x10 = 0x00000000
```

SLLI x7, x1, 1

```
=== Cycle 155000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000000
x10 = 0x00000000
```

SRLI x8, x1, 1

```
=== Cycle 165000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000000
```

SRAI x9, x1, 1

Next, R-type instructions

```
00208533  // ADD x10, x1, x2

405205b3  // SUB x11, x4, x5

00129633  // SLL x12, x5, x1

0020a6b3  // SLT x13, x1, x2

0020b733  // SLTU x14, x1, x2

0020c7b3  // XOR x15, x1, x2

0020d833  // SRL x16, x1, x2

4020d8b3  // SRA x17, x1, x2

0020e933  // OR  x18, x1, x2

0020f9b3  // AND x19, x1, x2
```

As in I-type Instructions, by the end of 265 ns, these R-type Instructions fill the register after calculating the values

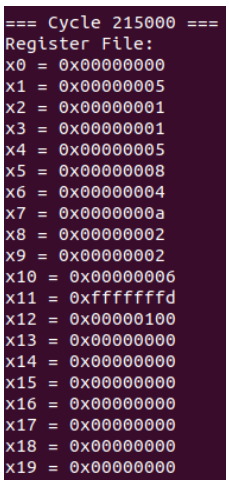
```
=== Cycle 175000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000006
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000000
x16 = 0x00000000
x17 = 0x00000000
x18 = 0x00000000
x19 = 0x00000000
```

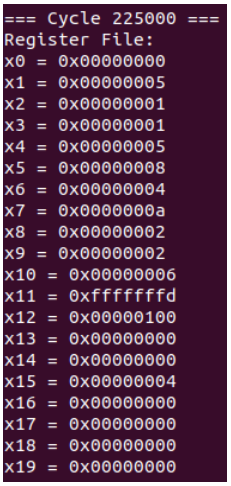
```
=== Cycle 185000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000006
x11 = 0xfffffffd
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000000
x16 = 0x00000000
x17 = 0x00000000
x18 = 0x00000000
x19 = 0x00000000
```

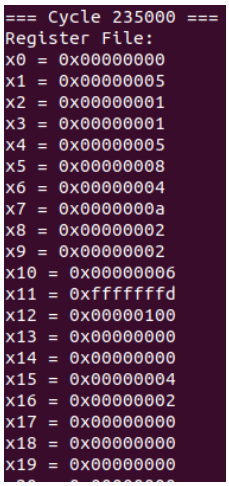
```
=== Cycle 195000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000006
x11 = 0xfffffffd
x12 = 0x00000100
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000000
x16 = 0x00000000
x17 = 0x00000000
x18 = 0x00000000
x19 = 0x00000000
```

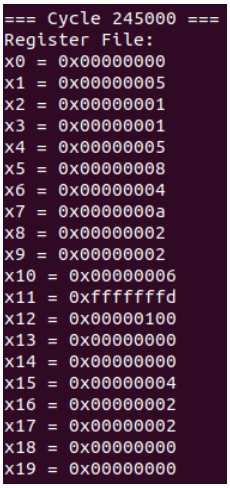
```
=== Cycle 205000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000006
x11 = 0xfffffffd
x12 = 0x00000100
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000000
x16 = 0x00000000
x17 = 0x00000000
x18 = 0x00000000
x19 = 0x00000000
```

`ADD x10, x1, x2` `SUB x11, x4, x5` `SLL x12, x5, x1` `SLT x13, x1, x2`

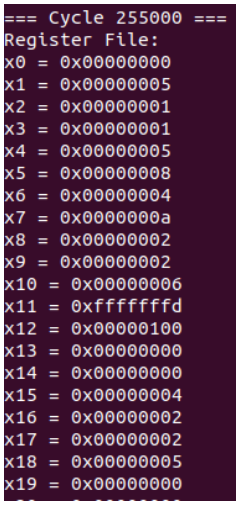


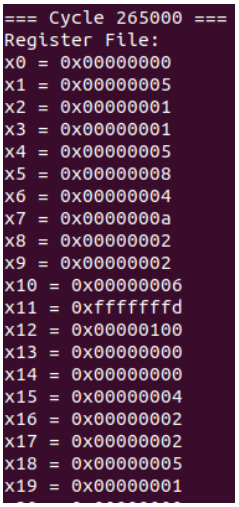






`SLTU x14, x1, x2` `XOR x15, x1, x2` `SRL x16, x1, x2` `SRA x17, x1, x2`



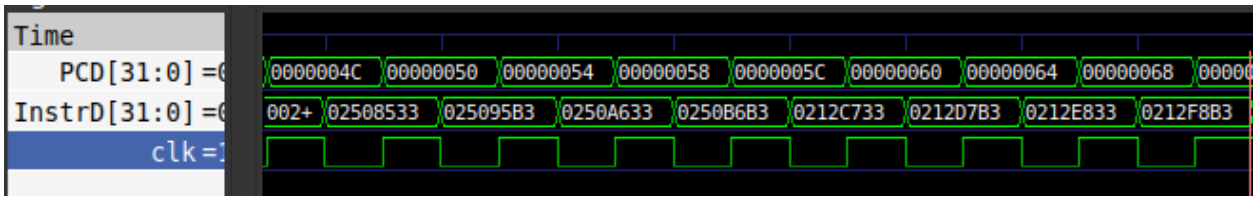


`OR  x18, x1, x2` `AND x19, x1, x2`

Next the Multiplication and Division Instructions are implemented

```
02508533  // MUL    x10, x1, x5

025095B3  // MULH   x11, x1, x5

0250A633  // MULHSU x12, x1, x5

0250B6B3  // MULHU  x13, x1, x5

0212C733  // DIV    x14, x5, x1
```

```
0212D7B3  // DIVU   x15, x5, x1

0212E833  // REM    x16, x5, x1

0212f8B3  // REMU   x17, x5, x1
```




```
=== Cycle 275000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0xfffffffd
x12 = 0x00000100
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000004
x16 = 0x00000002
x17 = 0x00000002
x18 = 0x00000005
x19 = 0x00000001
```

```
=== Cycle 285000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000100
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000004
x16 = 0x00000002
x17 = 0x00000002
x18 = 0x00000005
x19 = 0x00000001
```

```
=== Cycle 295000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000004
x16 = 0x00000002
x17 = 0x00000002
x18 = 0x00000005
x19 = 0x00000001
```

```
=== Cycle 305000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000000
x15 = 0x00000004
x16 = 0x00000002
x17 = 0x00000002
x18 = 0x00000005
x19 = 0x00000001
```

```
MUL x10, x1, x5    MULH  x11, x1, x5    MULHSU x12, x1, x5    MULHU  x13, x1, x5
```

```
=== Cycle 315000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000004
x16 = 0x00000002
x17 = 0x00000002
x18 = 0x00000005
```

```
=== Cycle 325000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000002
x17 = 0x00000002
x18 = 0x00000005
```

```
=== Cycle 335000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000002
x18 = 0x00000005
```

```
=== Cycle 345000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000003
x18 = 0x00000005
```

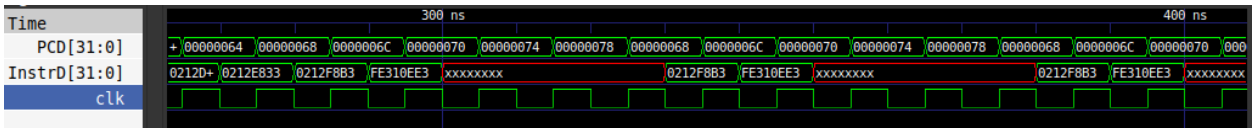
```
DIV  x14, x5, x1    DIVU x15, x5, x1    REM  x16, x5, x1    REMU x17, x5, x1
```

Then branch instructions are implemented

```
fe310ee3  // beq x2, x3, -4

//ff079ee3  // bne x15, x16, -4

//fe114ee3  // blt x2, x1, -4

//fe20dee3  // bge x1, x2, -4

//fe116ee3  // bltu x2, x1, -4

//fe20fee3 // bgeu x1, x2, -4
```

Each of these instructions are implemented while other branch instructions are commented out

```
fe310ee3  // beq x2, x3, -4
```

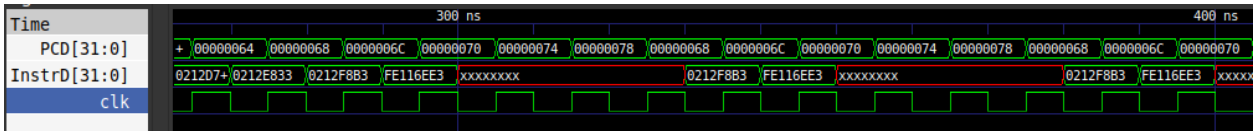



```
ff079ee3  // bne x15, x16, -4
```

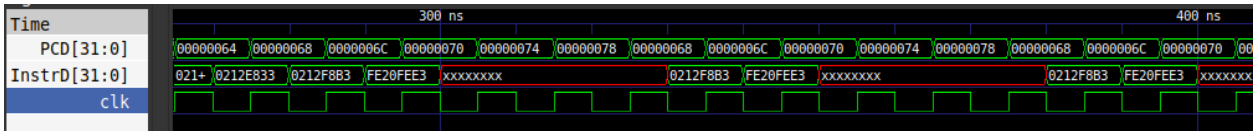

```
fe114ee3  // blt x2, x1, -4
```

```
fe20dee3  // bge x1, x2, -4
```


```
fe116ee3  // bltu x2, x1, -4
```

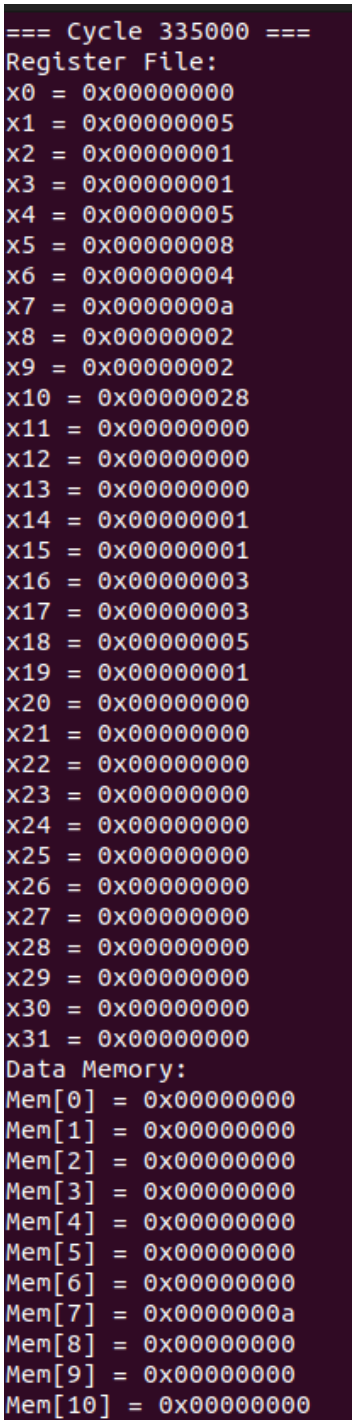



```
fe20fee3 // bgeu x1, x2, -4
```

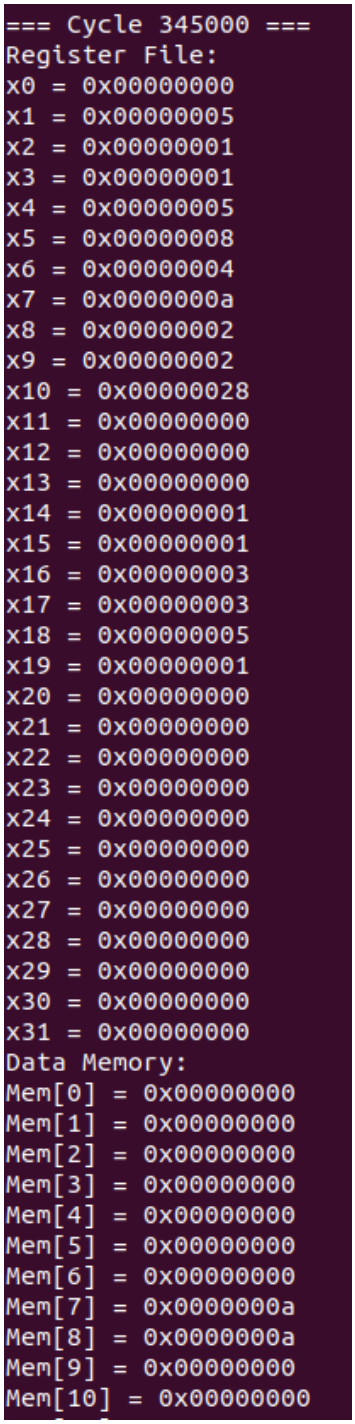


Now the store instructions are implemented

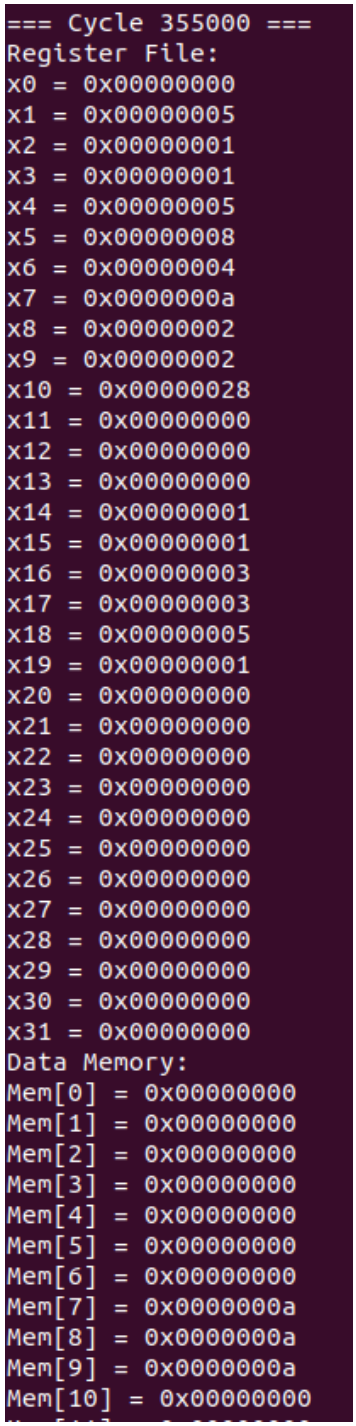
```
0x00708123 // sb x7, 2(x1)

0x007091a3 // sh x7, 3(x1)

0x0070a223 // sw x7, 4(x1)
```



`sb x7, 2(x1)`



`sh x7, 3(x1)`



`sw x7, 4(x1)`

Next Load instructions are implemented

```
0x00208a03 // lb x20, 2(x1)

0x00209a83 // lh x21, 2(x1)

0x0020ab03 // lw x22, 2(x1)

0x0020cb83 // lbu x23, 2(x1)

0x0020dc03 // lhu x24, 2(x1)
```


```
=== Cycle 375000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000003
x18 = 0x00000005
x19 = 0x00000001
x20 = 0xffffff0a
x21 = 0x00000000
x22 = 0x00000000
x23 = 0x00000000
x24 = 0x00000000
x25 = 0x00000000
x26 = 0x00000000
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

```
=== Cycle 385000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000003
x18 = 0x00000005
x19 = 0x00000001
x20 = 0xffffff0a
x21 = 0xffff000a
x22 = 0x00000000
x23 = 0x00000000
x24 = 0x00000000
x25 = 0x00000000
x26 = 0x00000000
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

```
=== Cycle 395000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000003
x18 = 0x00000005
x19 = 0x00000001
x20 = 0xffffff0a
x21 = 0xffff000a
x22 = 0x0000000a
x23 = 0x00000000
x24 = 0x00000000
x25 = 0x00000000
x26 = 0x00000000
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

```
=== Cycle 405000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000003
x18 = 0x00000005
x19 = 0x00000001
x20 = 0xffffff0a
x21 = 0xffff000a
x22 = 0x0000000a
x23 = 0x0000000a
x24 = 0x00000000
x25 = 0x00000000
x26 = 0x00000000
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

lb x20, 2(x1) lh x21, 2(x1) lw x22, 2(x1) lbu x23, 2(x1)

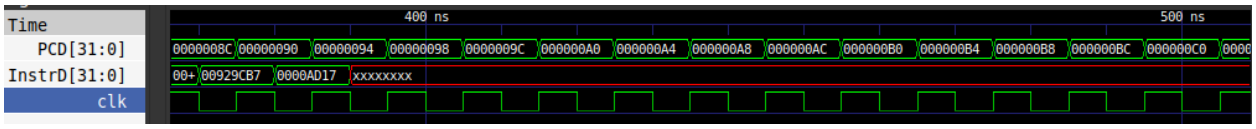
```
=== Cycle 415000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000003
x18 = 0x00000005
x19 = 0x00000001
x20 = 0xffffff0a
x21 = 0xffff000a
x22 = 0x0000000a
x23 = 0x0000000a
x24 = 0x0000000a
x25 = 0x00000000
x26 = 0x00000000
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

```
lhu x24, 2(x1)
```

Next U-type instructions are implemented

```
0x00929cb7 // lui x25, 2345
```

```
0x0000ad17 // auipc x26, 10
```



```
=== Cycle 425000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000003
x18 = 0x00000005
x19 = 0x00000001
x20 = 0xffffff0a
x21 = 0xffff000a
x22 = 0x0000000a
x23 = 0x0000000a
x24 = 0x0000000a
x25 = 0x00929000
x26 = 0x00000000
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

```
=== Cycle 435000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000003
x18 = 0x00000005
x19 = 0x00000001
x20 = 0xffffff0a
x21 = 0xffff000a
x22 = 0x0000000a
x23 = 0x0000000a
x24 = 0x0000000a
x25 = 0x00929000
x26 = 0x0000a090
x27 = 0x00000000
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

```
lui x25, 2345    auipc x26, 10
```

Finally Jump instructions are implemented

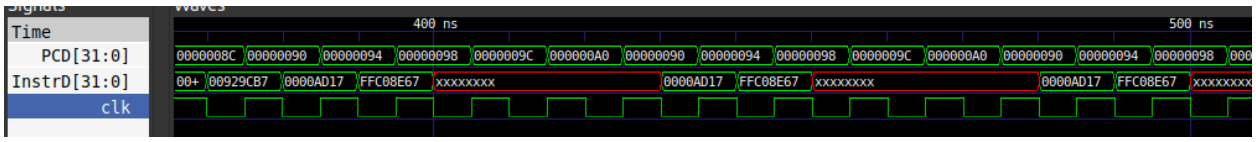
```
0xffdffdef // jal x27, -4

//0xffc08e67 // jalr x28, -4(x1)
```

(One is implemented by commenting other)

```
0xffdffdef // jal x27, -4
```

```
=== Cycle 445000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000003
x18 = 0x00000005
x19 = 0x00000001
x20 = 0xffffff0a
x21 = 0xffff000a
x22 = 0x0000000a
x23 = 0x0000000a
x24 = 0x0000000a
x25 = 0x00929000
x26 = 0x0000a090
x27 = 0x00000094
x28 = 0x00000000
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

0xffc08e67 // jalr x28, -4(x1)



```
=== Cycle 445000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000001
x3 = 0x00000001
x4 = 0x00000005
x5 = 0x00000008
x6 = 0x00000004
x7 = 0x0000000a
x8 = 0x00000002
x9 = 0x00000002
x10 = 0x00000028
x11 = 0x00000000
x12 = 0x00000000
x13 = 0x00000000
x14 = 0x00000001
x15 = 0x00000001
x16 = 0x00000003
x17 = 0x00000003
x18 = 0x00000005
x19 = 0x00000001
x20 = 0xffffff0a
x21 = 0xffff000a
x22 = 0x0000000a
x23 = 0x0000000a
x24 = 0x0000000a
x25 = 0x00929000
x26 = 0x0000a090
x27 = 0x00000000
x28 = 0x00000094
x29 = 0x00000000
x30 = 0x00000000
x31 = 0x00000000
```

Lab Part 04 - Hazard Handling

Introduction

Modern processor design often employs **pipelining** to improve instruction throughput and overall performance. In a pipelined architecture, the execution of an instruction is divided into multiple stages—typically **Fetch**, **Decode**, **Execute**, **Memory**, and **Writeback**—allowing multiple instructions to be processed simultaneously in different stages. This parallelism significantly enhances execution speed but introduces various challenges, particularly **hazards** that can compromise data correctness or cause execution delays.

To maintain the correctness of instruction execution in a pipelined processor, **hazard management** becomes essential. Hazards are situations where the next instruction in the pipeline cannot execute in the following clock cycle. They are typically categorized into three types:

- **Data Hazards**: Occur when an instruction depends on the result of a previous instruction still in the pipeline.

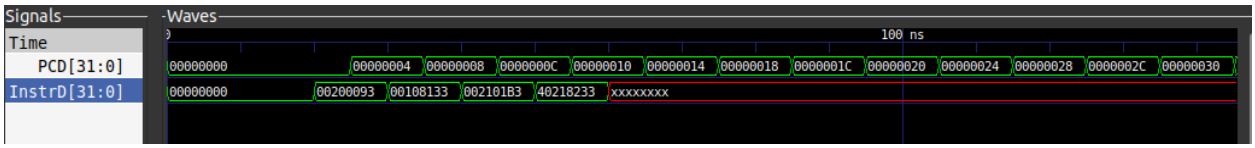
  Example 1 -

  ```
  add  x1, x0, 2     # x1 = x0 + 1

  add  x2, x1, x1    # x2 = x1 + x1 - uses result from x1

  add  x3, x2, x2    # x3 = x2 + x2 – uses result from x2

  sub  x4, x3, x2    # x4 = x3 - x2 – uses both x3 and x2
  ```

  Here is the machine code for this.

  ```
  0x00200093 // add x1, x0, 2

  0x00108133 // add x2, x1, x1

  0x002101b3 // add x3, x2, x2

  0x40218233 // sub x4, x3, x2
  ```

Here is the wave form of the instructions



Here are the results of the instructions

```
=== Cycle 75000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000002
x2 = 0x00000000
x3 = 0x00000000
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
```

```
=== Cycle 85000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000002
x2 = 0x00000000
x3 = 0x00000000
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
```

```
=== Cycle 95000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000002
x2 = 0x00000000
x3 = 0x00000000
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
```

```
=== Cycle 105000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000002
x2 = 0x00000000
x3 = 0x00000000
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
```

```
add x1, x0, 2  add x2, x1, x1  add x3, x2, x2  sub x4, x3, x2
```

X1 is updated to 2 as per 1st instruction, but successor instructions don't update the relevant registers as at the time those successor instructions fetch data from register, thoser source resitsters haven't been updated. That is a data hazard.

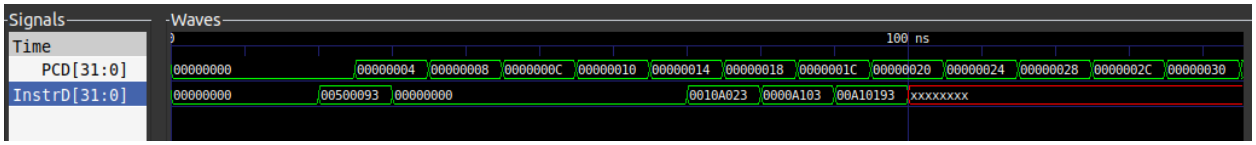
A special kind of data hazard is load use data hazard. Here is an example for that.

Example 2

```
0x00500093   // addi x1, x0, 5

0x00000000   // Nop to avoid data hazard

0x00000000   // Nop to avoid data hazard

0x00000000   // Nop to avoid data hazard

0x00000000   // Nop to avoid data hazard

0x0010a023  // sw x1, 0(x1)

0x0000a103 // lw x2, 0(x1)

0x00a10193 // addi x3, x2, 10
```

Here 7th instruction loads data from memory to the x2 register. Subsequently, x2 register is used as a source before completing the pipeline journey of 7th instruction, again happens a data hazard.

As you can see in the output given below, the x3 is not updated properly (as x3 should be 15(x0000F), not 10(0x0000A)) It is because x2 source register was not updated by the time the last instruction fetches data from register file.




```
=== Cycle 75000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000000
x3 = 0x00000000
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
```

```
Data Memory:
Mem[0] = 0x00000000
Mem[1] = 0x00000000
Mem[2] = 0x00000000
Mem[3] = 0x00000000
Mem[4] = 0x00000000
Mem[5] = 0x00000005
Mem[6] = 0x00000000
Mem[7] = 0x00000000
Mem[8] = 0x00000000
```

```
=== Cycle 135000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000005
x3 = 0x00000000
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
```

```
=== Cycle 145000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000005
x3 = 0x0000000a
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
```

cycle 11500

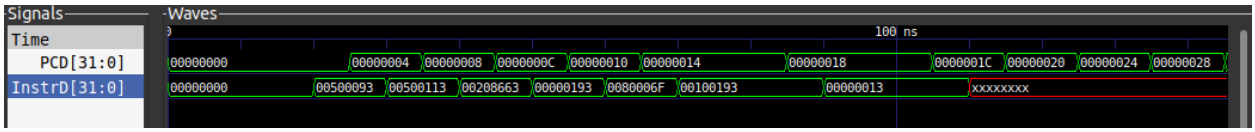
```
addi x1, x0, 5    sw x1, 0(x1)      lw x2, 0(x1)      addi x3, x2, 10
```

- **Control Hazards**: Arise from branch and jump instructions whose target is not yet known.

  Example 3-

```
00500093  // addi x1, x0, 5

00500113  // addi x2, x0, 5

00208663  // beq x1, x2, 12

00000193  // addi x3, x0, 0

0080006f  // jal x0, 8

00100193  // addi x3, x0, 1

00000013  // nop
```

If this is normally executed, after 3rd instruction, it should goes to 6th instruction straight forward.



But as it is seen in this wave form, the after 3rd instruction, it goes to 4th instruction. It is because by the time 3rd instruction changes the target address, there is no mechanism to flush or stop wrongly implemented successor instructions. This is a control hazard.

- **Structural Hazards**: Happen when hardware resources are insufficient to handle concurrent operations
  A **structural hazard** occurs in a pipelined processor when **hardware resources are insufficient** to execute multiple instructions simultaneously. This usually arises when two or more instructions need access to the **same functional unit or memory component** during the same clock cycle.

For example:

- If both the **Fetch** and **Memory** stages need access to a **single shared memory** in the same cycle, a conflict occurs.

- If there's only **one multiplier unit** and two instructions require it at the same time.

🔹 Causes of Structural Hazards

- Single memory used for both instructions and data
- Limited number of ALUs, multipliers, or FPUs
- Shared register file ports not sufficient for simultaneous reads/writes

🔹 Remedies and Solutions

1. **Hardware Duplication (Resource Duplication)**

    - Use **separate instruction and data memories** (Harvard architecture)
    - Add **multiple ALUs**, FPUs, or multipliers as needed

2. **Pipeline Scheduling**
    - Delay certain instructions intentionally to avoid resource conflicts
    - Compiler or hardware can reorder instructions to minimize hazards

3. **Stalling (Interlocking)**
    - Insert **NOPs or stalls** in the pipeline until the resource becomes available
    - Common but **reduces performance**

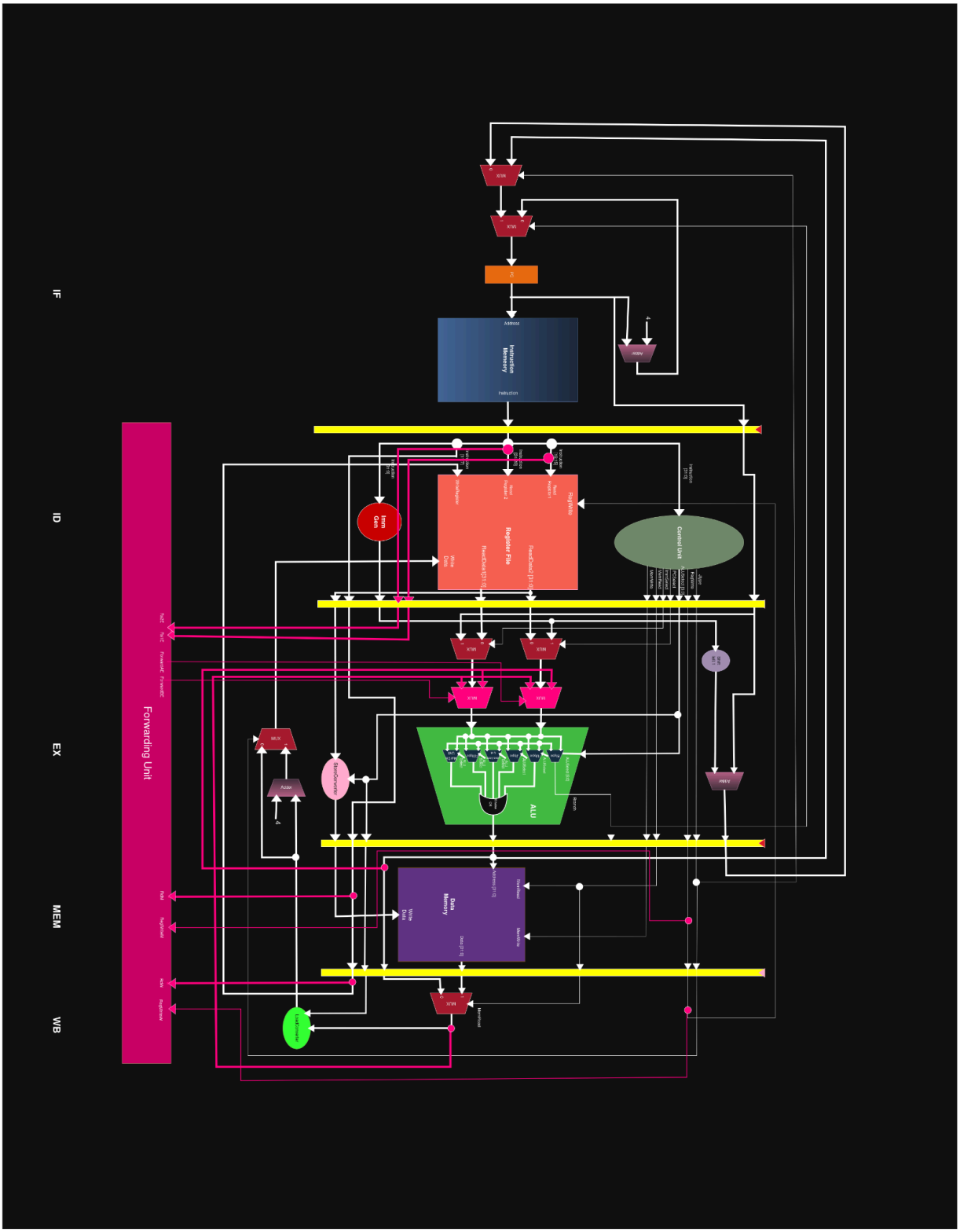
4. **Resource Arbitration Logic**
    - Design control logic to prioritize and manage access to shared resources dynamically

To resolve these hazards in our basic pipelined RISC-V processor design, three specialized hardware modules were implemented:

1. **Forwarding Unit**: Mitigates data hazards by bypassing data from later stages directly to the execution stage.
2. **Hazard Detection Unit (HDU)**: Detects load-use hazards and inserts pipeline stalls when necessary.
3. **Control Hazard Unit (CHU)**: Detects control hazards due to branches and flushes incorrect instructions to redirect program flow.

These modules work together to ensure that the processor executes instructions correctly while minimizing performance penalties caused by pipeline interruptions.

Forwarding Unit



The **Forwarding Unit** is a crucial component in a pipelined processor that helps resolve **data hazards** without stalling the pipeline.

- **What It Does:**

It **detects data dependencies** between instructions in different stages of the pipeline (especially between Execute, Memory, and WriteBack stages), and **forwards** the required operand values from a later stage back to an earlier one before the write-back is complete.

- **Why It's Needed:**

In pipelining, an instruction may need a register value that is being calculated by a previous instruction which hasn't yet written it back to the register file. Without forwarding, the processor would need to **stall** and wait.

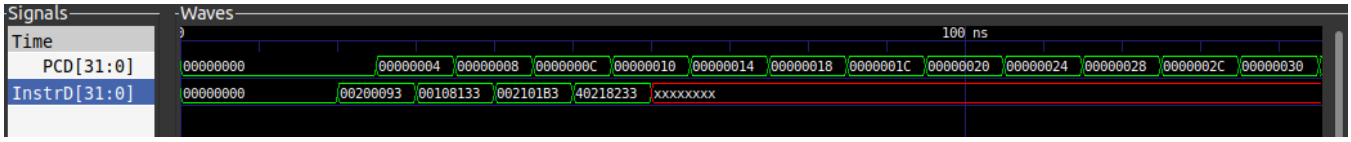
- **How It Works:**

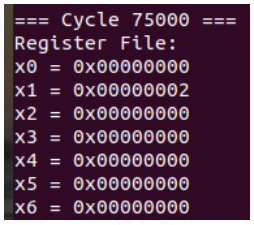
  - If the destination register of an instruction in the **EX or MEM** stage matches the source register (rs1 or rs2) of the current instruction in the **Decode or Execute** stage,

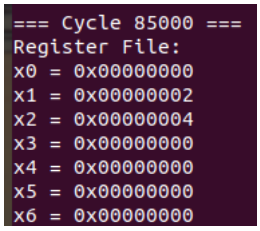
  - The unit enables **forwarding paths** to provide the correct operand directly from the pipeline register instead of waiting for it to be written back.

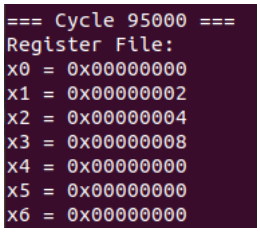
Let’s take the first example of data hazards

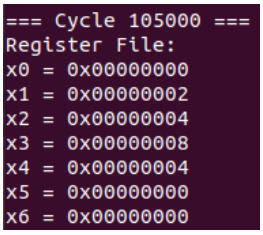
```
0x00200093 // add x1, x0, 2

0x00108133 // add x2, x1, x1

0x002101b3 // add x3, x2, x2

0x40218233 // sub x4, x3, x2
```











`add x1, x0, 2` `add x2, x1, x1` `add x3, x2, x2` `sub x4, x3, x2`

Unlike in the previous example, every instructions are able to update the register values correctly.

Hazard Detection Unit

The **Hazard Detection Unit (HDU)** in a pipelined processor detects situations where an instruction **must be delayed (stalled)** to avoid incorrect execution due to **data dependencies**.

What It Does:

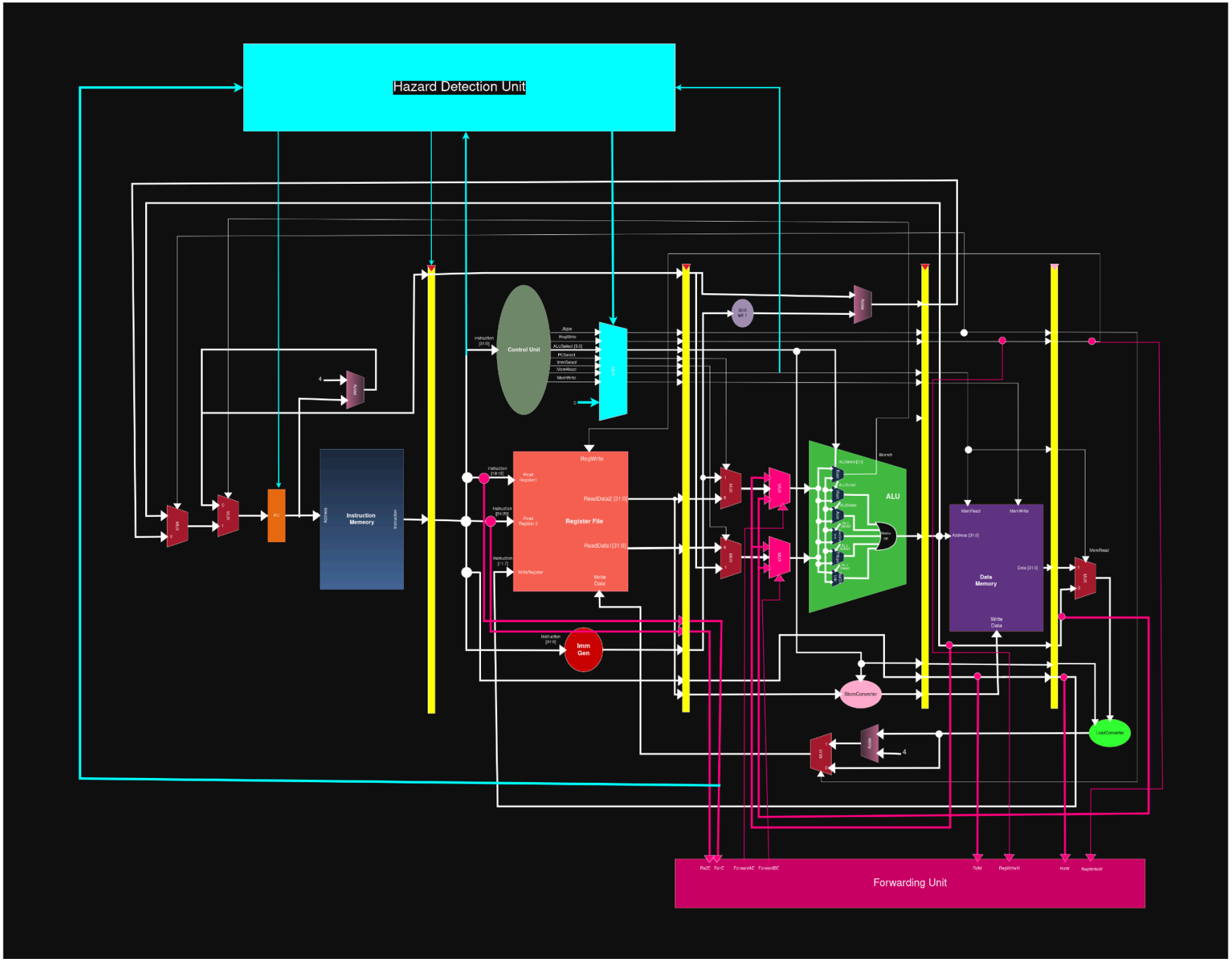
It monitors instructions in the pipeline and **inserts a stall (bubble)** if the next instruction **depends on data** that hasn't been written yet — especially from a **load instruction**.

How It Works :

- Checks if the instruction in the **Execute** stage is a `load`
- If the `load` destination (`rd`) matches the source register (`rs1` or `rs2`) of the instruction in the **Decode** stage
- Then:
  Stall the pipeline
  Insert a NOP in the Execute stage
  Freeze the PC and IF/ID register

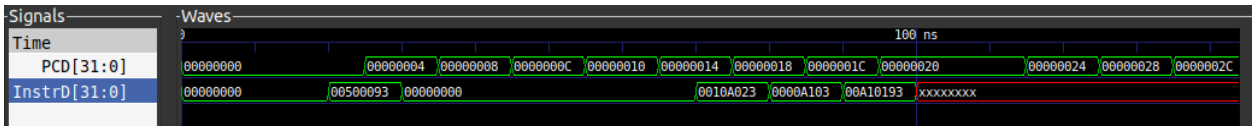
Result:

- Prevents **incorrect results**
- Ensures **data integrity**
- Maintains **correct program behavior**


Hazard Detection Unit
Control Unit
Instruction Memeory
Register File
RegWrite
ReadData2 [31:0]
ReadData1[31:0]
Write Data
Imm Gen
ALU
Data Memory
MemRead
MemWrite
Branch
StoreConverter
LoadConverter
Forwarding Unit

Let’s look at the example for load-use hazard

```
0x00500093   // addi x1, x0, 5

0x00000000   // Nop to avoid data hazard

0x00000000   // Nop to avoid data hazard

0x00000000   // Nop to avoid data hazard

0x00000000   // Nop to avoid data hazard

0x0010a023  // sw x1, 0(x1)

0x0000a103 // lw x2, 0(x1)

0x00a10193 // addi x3, x2, 10
```




```
=== Cycle 75000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000000
x3 = 0x00000000
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
```

```
Data Memory:
Mem[0] = 0x00000000
Mem[1] = 0x00000000
Mem[2] = 0x00000000
Mem[3] = 0x00000000
Mem[4] = 0x00000000
Mem[5] = 0x00000005
Mem[6] = 0x00000000
Mem[7] = 0x00000000
Mem[8] = 0x00000000
```

```
=== Cycle 135000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000005
x3 = 0x00000000
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
```

```
=== Cycle 155000 ===
Register File:
x0 = 0x00000000
x1 = 0x00000005
x2 = 0x00000005
x3 = 0x0000000f
x4 = 0x00000000
x5 = 0x00000000
x6 = 0x00000000
x7 = 0x00000000
```

cycle 11500

```
addi x1, x0, 5    sw x1, 0(x1)      lw x2, 0(x1)      addi x3, x2, 10
```

Unlike in the second example, here the x3 register is updated by the end of 155ns. X3 = x2 + 10 = 5 + 10 = 15 = x0000000f

Control Hazard Unit

A **control hazard** (or **branch hazard**) happens when the processor doesn't know the **next instruction address** because it's waiting to evaluate a **branch or jump decision**.

If it guesses wrong, it has already fetched wrong instructions — so it must **flush** and **redirect the PC** to the correct path.

Purpose of the Control Hazard Detection Unit (CHU)

The **Control Hazard Unit** solves this by:

- **Detecting** when a **branch or jump** occurs
- **Deciding** if the branch condition is true (e.g., `BEQ`, `BNE`, `BLT`, etc.)
- **Flushing** the wrong instruction (stall/freeze or discard)
- **Redirecting** the PC to the **target address** if needed

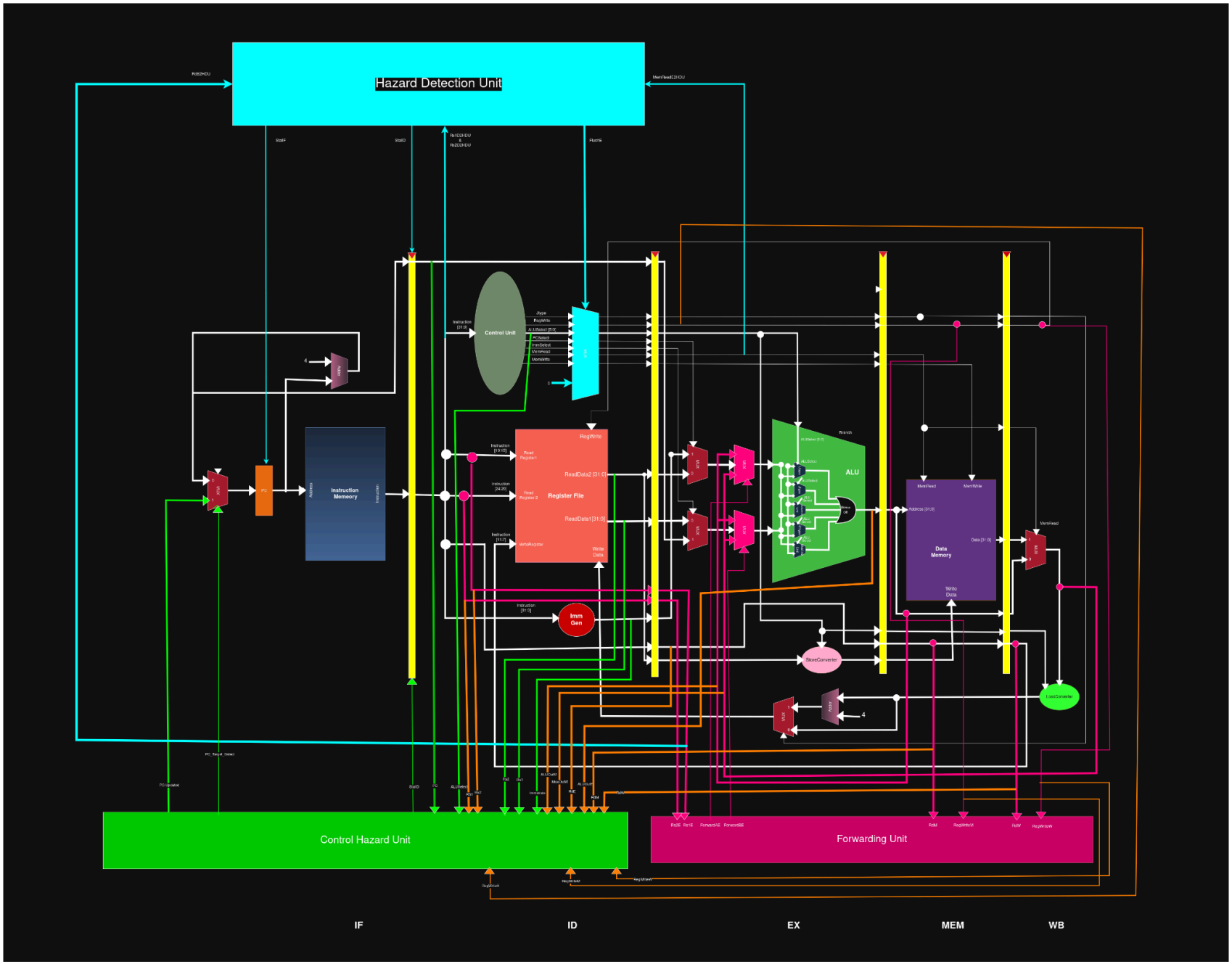
What It Does?

In the Verilog module `Control_Hazard_Unit`:

1. It receives current instruction details from the Decode stage:
    - Branch or jump type (`aluSelect`)
    - Source registers (`rs1`, `rs2`)
    - Immediate offset (`imm`)
    - Program counter (`pc`)
2. It uses **forwarding logic** to get the most up-to-date values for `rs1` and `rs2` (in case they are still being written back).
3. Based on the type of branch (`aluSelect`), it:
    - Checks the **branch condition**
    - If true:
        - **FlushD = 1** → tells the pipeline to **flush Decode stage**
        - **target_pc_valid = 1** → new branch address is ready
        - **target_pc = pc + imm** (or `rs1 + imm` for `JALR`)

Outputs:

- `FlushD` – Flushes the current instruction if the branch is taken.
- `target_pc_valid` – Indicates that a valid target PC is ready to redirect.
- `target_pc` – The address to jump to.


Hazard Detection Unit
Control Unit
Register File
ALU
Instruction Memory
Data Memory
Imm Gen
StoreConverter
LoadConverter
Control Hazard Unit
Forwarding Unit
IF
ID
EX
MEM
WB

Let’s go to 3rd example,

```
00500093  // addi x1, x0, 5

00500113  // addi x2, x0, 5

00208663  // beq x1, x2, 12

00000193  // addi x3, x0, 0

0080006f  // jal x0, 8

00100193  // addi x3, x0, 1

00000013  // nop
```

Before handling the control hazard, the 3rd instruction is succeeded by the 4th example without jumping to 6th instruction as per rules should be.

But with the control hazard unit, the branch and jump instructions can decide the path of the instruction flow after implementing a one instruction wrongly, then flushing the wrong instruction and implementing the correct instruction as seen in the following wave form.