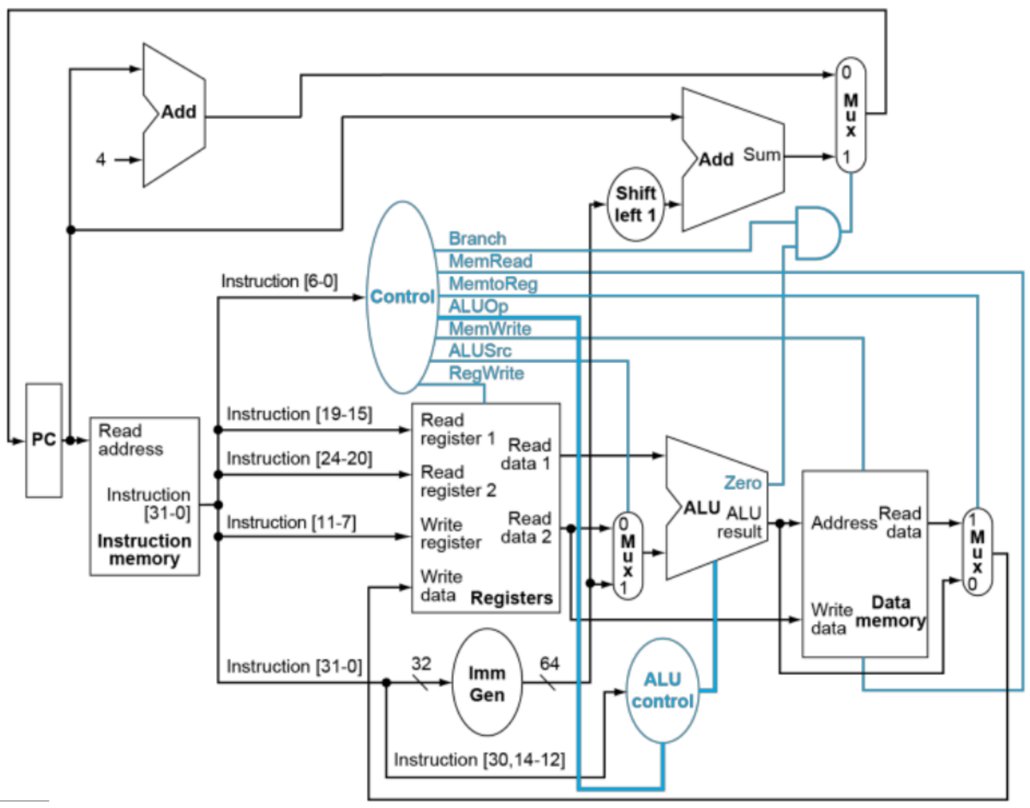
PROJECT REPORT

Saarthak Sabharwal - 2023102055
Aanchal Amit Mundhada - 2023112016
Krishna Goel - 2023112009

Datapath With Control



Program Counter (PC) and Instruction Fetch Process

1. Introduction

The **Program Counter (PC) and Instruction Fetch process** form the foundation of the processor’s execution cycle. The **PC** holds the address of the next instruction to be executed, while the **Instruction Fetch** mechanism retrieves this instruction from memory and prepares it for execution. This process ensures the correct sequencing of program execution.

2. Components Involved

The key components in the **PC and Instruction Fetch** process include:

1. **Program Counter (PC)**: Holds the address of the next instruction to be executed.
2. **Instruction Memory**: Stores machine instructions and provides them based on the address received from the PC.
3. **Adder (`PC + 4`)**: Computes the address of the next sequential instruction.
4. **Multiplexer (MUX)**: Selects between normal sequential execution (`PC + 4`) or a branch/jump target.
5. **Control Unit**: Determines whether the PC should update sequentially or jump to a new location.

3. Process Flow of PC and Instruction Fetch

Fetching the Instruction Address from PC

- At the beginning of each cycle, the **Program Counter (PC)** holds the address of the instruction to be fetched.
- This address is sent to **Instruction Memory**, which retrieves the corresponding **32-bit instruction**.

Reading the Instruction from Memory

- The **Instruction Memory** receives the PC address and outputs the instruction stored at that location.

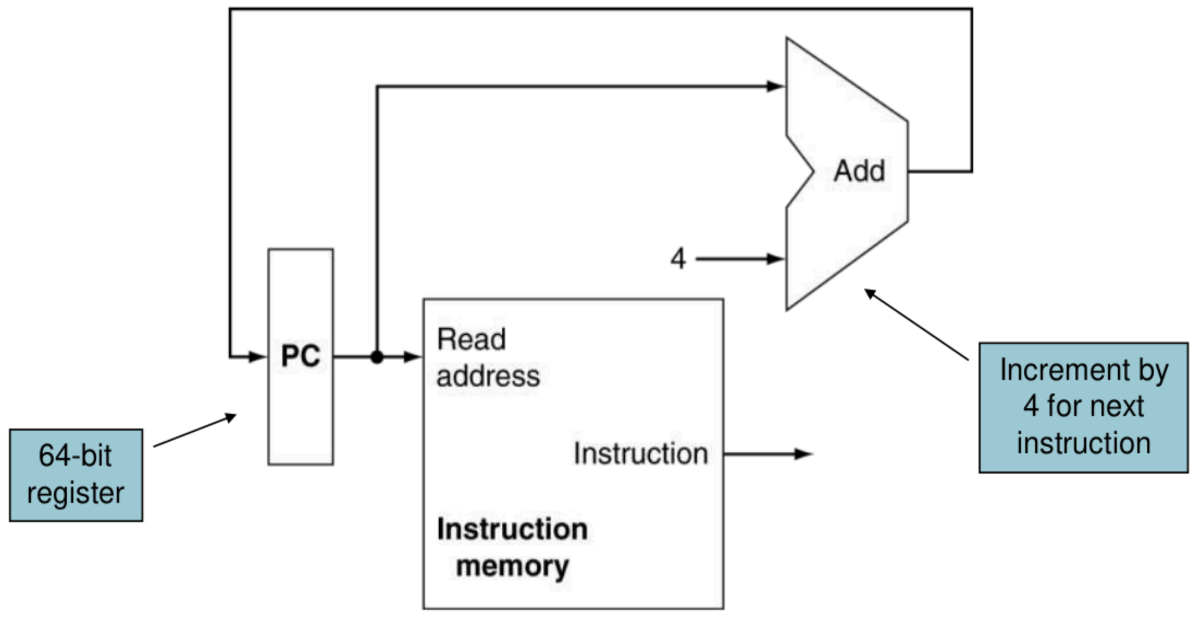
- This instruction is sent to:
  - **Control Unit** (to generate control signals).
  - **Register File** (to determine register operands).
  - **Immediate Generator** (to extract immediate values, if present).

Computing the Next Instruction Address (`PC + 4`)

- Since RISC-V instructions are **fixed-length (32-bit or 4 bytes)**, the next sequential instruction is located at: Next PC=PC+4\text{Next PC} = PC + 4Next PC=PC+4
- This addition is performed by an **Adder**, ensuring smooth sequential execution.

Handling Branches and Jumps

- If the fetched instruction is a **branch (`BEQ`, `BNE`)** or **jump (`JAL`, `JALR`)**, an alternative target address is computed using:
  - The **Immediate Generator** (to obtain offset values).
  - A **Shift Left Unit** (to adjust branch/jump offsets).
  - An **Adder** (to compute the final branch/jump target).
- A **Multiplexer (MUX)** selects between:
  - `PC + 4` (sequential execution).
  - **Branch Target Address** (if a conditional branch is taken).
  - **Jump Target Address** (for direct jumps).
- The selection is controlled by signals from the **Control Unit** and the **ALU Zero Flag**.

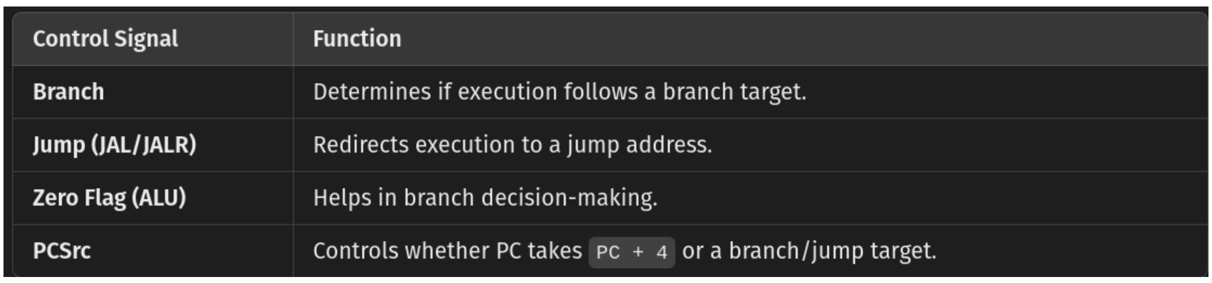


INSTRUCTION FETCH AND PC UPDATE

Updating the Program Counter (PC)

- The selected **next PC value** (from the MUX) is written back into the **Program Counter (PC)** at the end of the cycle.
- This updated value becomes the address for the next instruction fetch, ensuring continuous execution.

4. Control Signals Affecting PC and Instruction Fetch



Control Signal	Function
Branch	Determines if execution follows a branch target.
Jump (JAL/JALR)	Redirects execution to a jump address.
Zero Flag (ALU)	Helps in branch decision-making.
PCSrc	Controls whether PC takes `PC + 4` or a branch/jump target.

The **Program Counter (PC) and Instruction Fetch** process ensures that the processor retrieves and executes instructions in the correct sequence. The **PC** updates dynamically based on sequential execution, branching, or jumping, while the **Instruction Fetch unit** retrieves and prepares the next instruction for execution. This process is fundamental to the smooth functioning of a processor's execution cycle.

Branch Instruction Overview

Branch instructions (e.g., BEQ, BNE) dictate the flow of execution based on a comparison between two register values. If the condition is satisfied (e.g., the two register values are equal), the **Program Counter (PC)** is rewritten to the target address of the branch. If the condition is not satisfied, the program continues with the next instruction in sequence.

Components Involved

1. Register File

The **Register File** saves the processor's general-purpose registers and provides high-speed access to them. Two registers are required in the case of branch instructions to perform the comparison.

Working of the Register File:

- **Read Register 1 and Read Register 2:**
  The instruction provides the register addresses (usually **rs1** and **rs2** in RISC-V). These addresses are used to read the contents of two registers.

- **Read Data 1 and Read Data 2:**
  The **Register File** outputs the data contained in the registers specified by the instruction. These values are used for comparison

within the **ALU**.

In essence, the **Register File** retrieves the operand values needed to calculate the branch condition.

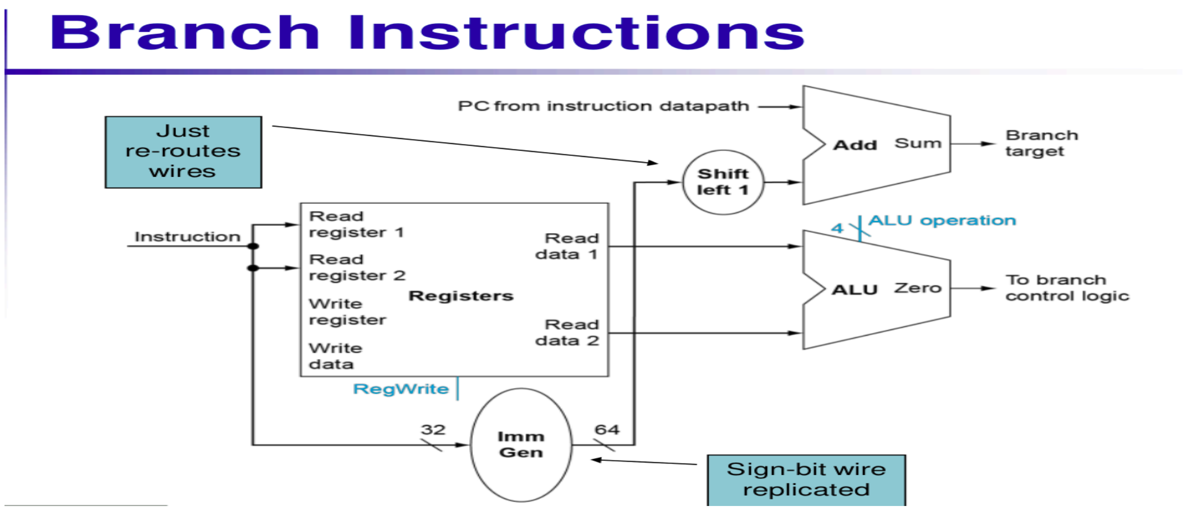
2. Immediate Generator (Imm Gen)

The **Immediate Generator (Imm Gen)** extracts the **immediate field** from the instruction, which typically denotes the offset for the branch target address. For branch instructions, this immediate value is used to compute the target address relative to the current instruction.

Working of the Immediate Generator (Imm Gen):

- The **Immediate Field** is extracted from the instruction (usually bits [31:0] in the instruction).
- This immediate value is **sign-extended** to 64 bits (in 64-bit processors) to handle negative offsets properly.
- The immediate value is then **shifted left by 1 bit** because the instructions are word-aligned. Shifting aligns the address with the instruction boundary.

The **Imm Gen** thus produces the extended and shifted immediate value, which is used to calculate the branch target address.



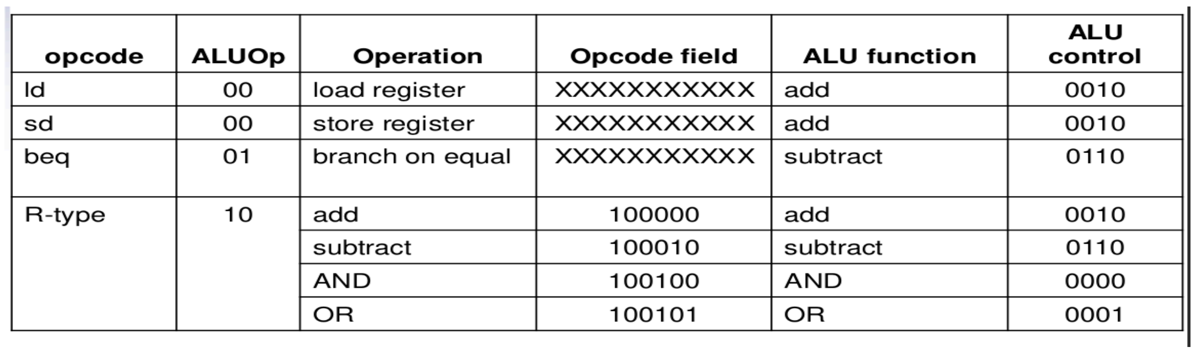
3. ALU (Arithmetic Logic Unit)

The **ALU** is the core unit responsible for performing arithmetic and logical operations. In branch instructions, the **ALU** compares the two register values to check whether the branch condition is met.

Operation of the ALU:

- The **ALU** receives the two register values (**Read Data 1** and **Read Data 2**) from the **Register File**.
- It then performs the comparison operation depending on the type of branch instruction:
    - **BEQ (Branch if Equal):** Checks if the two register values are the same.
    - **BNE (Branch if Not Equal):** Checks if the two register values are different.
    - **BLT (Branch if Less Than):** Checks if the first register value is less than the second.
    - **BGE (Branch if Greater Than or Equal):** Checks if the first register value is greater than or equal to the second.
- The **ALU** generates an output:
    - **ALU Zero Flag:** When the comparison condition is met (e.g., values are equal for BEQ), the **Zero flag** is set to 1. This flag determines whether the branch should be taken.

The **ALU** essentially determines whether the branch condition is met by comparing the two register values.



opcode	ALUOp	Operation	Opcode field	ALU function	ALU control
ld	00	load register	XXXXXXXXXXX	add	0010
sd	00	store register	XXXXXXXXXXX	add	0010
beq	01	branch on equal	XXXXXXXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001

ALU CONTROL

4. Adder

The **Adder** is responsible for calculating the branch target address and also helps in computing the next instruction's sequential address (`PC + 4`).

Operation of the Adder:

- The **Adder** performs two functions:
    1. **PC + 4:** It computes the next instruction address (sequential execution). This is simply the current **PC value** incremented by 4.
    2. **Branch Target Calculation:** The adder takes the **shifted immediate** from the **Imm Gen** and adds it to the current **PC + 4**. This produces the **branch target address** when the branch condition is satisfied.

The output of the adder (either **PC + 4** or the **branch target address**) is sent to a **MUX**, which determines the new **PC value**.

Final Step - PC Update

Depending on the **ALU Zero flag** (i.e., whether the branch condition is met), the **MUX** selects either the **next sequential PC** (`PC + 4`) or the **branch target address** computed by the adder. The selected value is then written back to the **PC register**, which either resumes sequential execution or branches to the target address

Load/Store Datapath

Load/Store Datapath is employed to process load (e.g., LW - Load Word) and store (e.g., SW - Store Word) instructions. These instructions enable memory access, enabling data to be loaded from memory into registers or stored from registers into memory.

Operation of the Load/Store Datapath

1. ALU (Arithmetic Logic Unit)

The ALU is responsible for calculating the effective memory address needed for load and store operations.

It requires two inputs:

- **Base Register Value:** It stores the initial memory address.
- **Offset (Immediate Value):** It is given in the instruction to identify the precise memory location.

The ALU executes the following calculation:

$$\text{Effective Address} = \text{Base Address} + \text{Offset}$$

The result from the ALU is then fed to the Data Memory Unit for memory access.

2. Data Memory

The Data Memory Unit stores and loads data according to the calculated memory address.

The calculated address from the ALU is an input to the memory unit.
Two primary operations occur:

- **Memory Read (Load Operation):**
    - If the instruction is a load instruction (for example, LW), data is read from memory at the calculated address.
    - The **MemRead signal** is asserted to bring in the needed data.
- **Memory Write (Store Operation):**

  - If the instruction is a store instruction (e.g., SW), the register value is stored in memory at the calculated address.
  - The **MemWrite signal** is asserted to write the data.

3. Multiplexer (MUX)

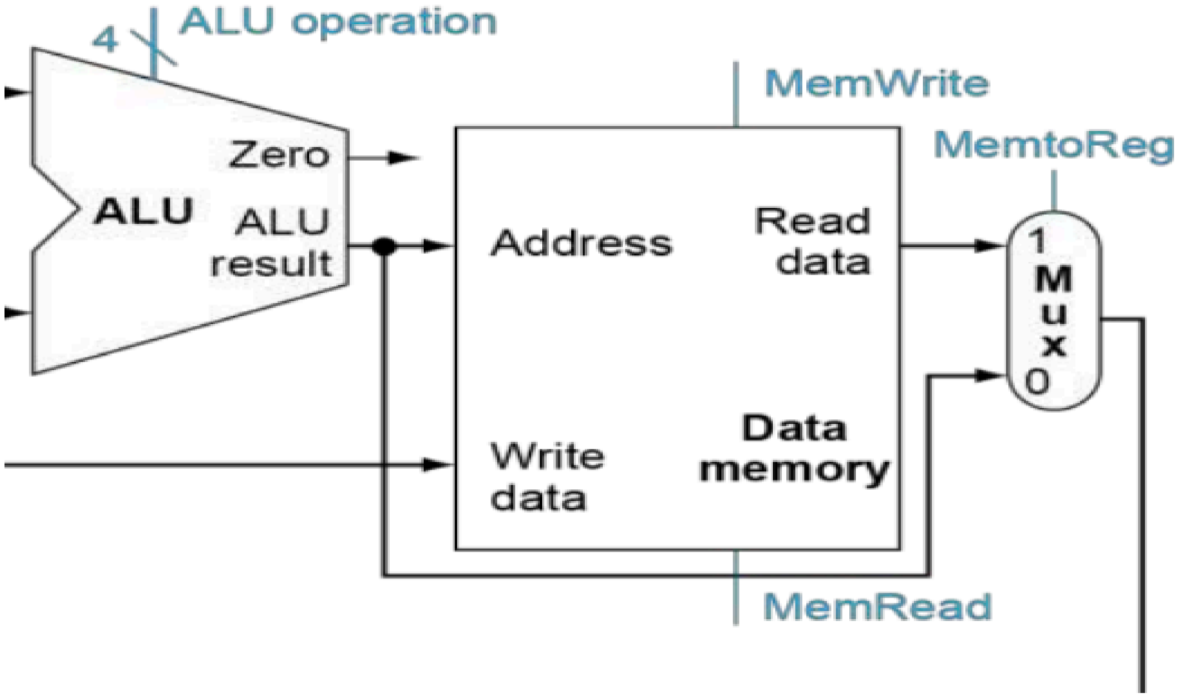
The Multiplexer (MUX) determines the data that is written back to the registers.

The control signal **MemtoReg** selects between:

- The **ALU result** (for arithmetic/logical operations).
- The **Memory Read Data** (for load instructions).

The selection is determined by the value of **MemtoReg**:

- **0:** ALU result is chosen (arithmetic and logical operations are done).
- **1:** Memory Read Data is chosen (load instruction).



Step-by-Step Execution of Load and Store Instructions

For Load Instructions (e.g., LW - Load Word)

1. The **ALU calculates the memory address** by performing the sum of the base register value and offset.
2. The **memory is referenced**, and information is read from the calculated address.
3. The **MUX chooses the Memory Read Data** (MemtoReg = 1).
4. Information is **written into the destination register**.

For Store Instructions (i.e., SW - Store Word)

1. The ALU calculates the memory address by adding the base register value and offset.
2. The **information to be stored** is retrieved from a register.
3. The **memory is written at the calculated address** (MemWrite is on).
4. No writing back of data is made into registers.

Summary of Components and Their Roles

1. **ALU** – Computes the effective memory address.
2. **Data Memory** – Reads or writes data based on the computed address.
3. **MUX** – Selects between ALU result (arithmetic operations) and memory output (load instruction).
4. **MemRead** – Enables memory read operation (used in load instructions).
5. **MemWrite** – Enables memory write operation (used in store instructions).
6. **MemtoReg** – Determines whether the register receives ALU result or memory data.

The **Load/Store Datapath** is essential for memory operations in a processor. It ensures that data is efficiently transferred between registers and memory while maintaining proper address calculations and control signals. By using the ALU, data memory, and MUX effectively, the datapath executes load and store instructions seamlessly.

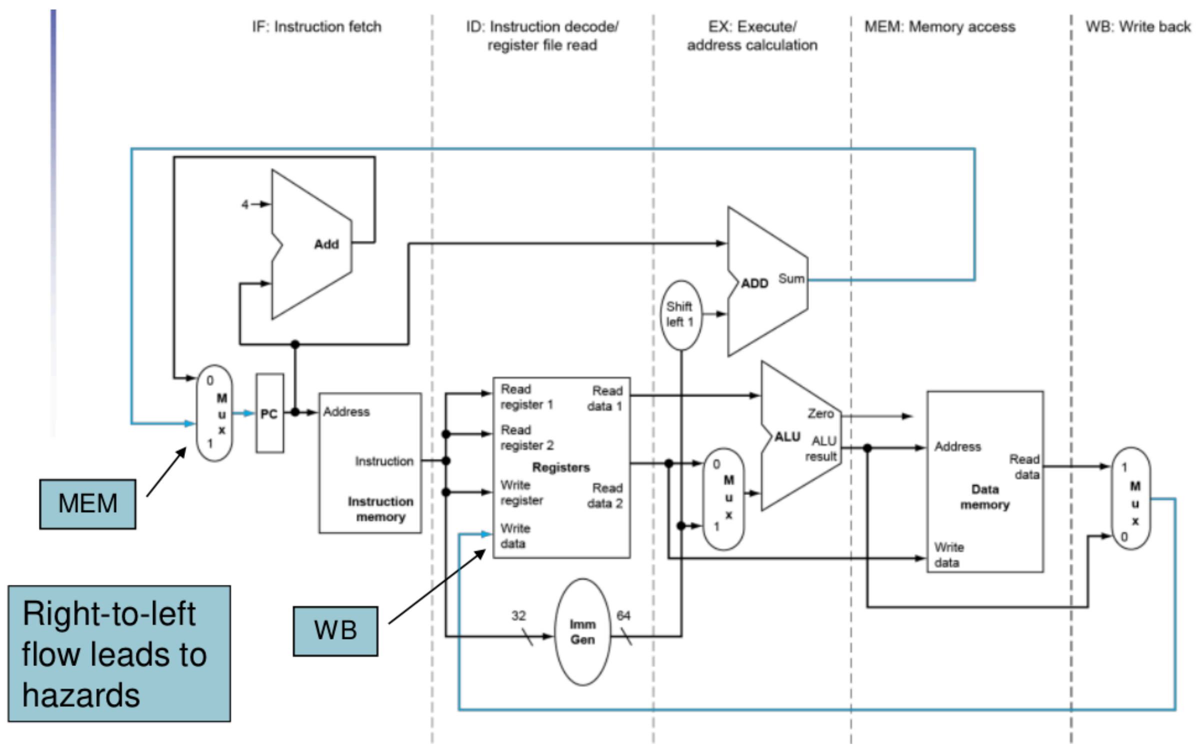
RISC-V Pipeline

A pipeline in a processor enables several instructions to be executed at once by dividing the process of execution into stages. The RISC-V architecture uses a five-stage pipeline, where one instruction proceeds through five stages before finishing execution.

The five stages of the RISC-V pipeline are:

1. Instruction Fetch (IF)
2. Instruction Decode (ID) & Register Read
3. Execute (EX)
4. Memory Access (MEM)
5. Write Back (WB)

Each phase serves a specific purpose, and as a result, various instructions can be in various phases simultaneously, enhancing performance and efficiency. Let's consider each phase individually.



PIPELINE DATAPATH

1. Instruction Fetch (IF)

It is the initial phase of the pipeline where the CPU retrieves the instruction from memory.

Parts involved in this phase:

- **Program Counter (PC):** Stores the address of the next instruction to be executed.
- **Instruction Memory:** A memory unit that stores all program instructions.
- **Adder:** Employed to compute the address of the next instruction.

Steps in Instruction Fetch:

1. The PC supplies the address of the instruction to be fetched from memory.
2. The instruction is fetched from memory and held in the instruction register.

3. The PC is modified to point to the next instruction (typically by incrementing 4, as each instruction in RISC-V is 4 bytes long).

Analogy: It is similar to reading a recipe from a cookbook prior to cooking.

2. Instruction Decode (ID) & Register Read

Once the instruction is retrieved, the CPU must know what to execute. This is done in the Instruction Decode stage.

Components of this stage:

- **Control Unit:** Identifies instruction type and provides control signals.
- **Register File:** Holds all registers and facilitates reading values.
- **Immediate Generator:** Derives immediate values from instructions.

Instructions in Instruction Decode:

- The instruction is categorized as to which type of instruction it is (e.g., arithmetic, memory, or branch).
- The values of the registers are read (if necessary).
    - In case of R-type instructions, both source registers are read.
    - In case of I-type and S-type instructions, one register is read only.
- If the instruction contains an immediate value (e.g., addi or lw), it is extracted from the instruction.

Analogy: This is like gathering all ingredients after understanding the recipe.

3. Execute (EX)

In this stage, the actual computation happens. This is where the Arithmetic Logic Unit (ALU) performs calculations or determines memory addresses.

Components involved in this stage:

- **Arithmetic Logic Unit (ALU):** Does arithmetic (addition, subtraction) and logical (AND, OR) operations.
- **ALU Control Unit:** Determines what operation the ALU needs to do.
- **Multiplexer (MUX):** Chooses between register values and immediate values for ALU operations.

Steps in Execution:

- If it's an arithmetic instruction (e.g., add, sub), the ALU does the operation using values from registers.
- If it's a memory instruction (e.g., lw, sw), the ALU computes the memory address by adding the value of the base register and the immediate offset.
- If it's a branch instruction (e.g., beq), the ALU checks two registers to find out if it should take the branch.

Analogy: This is similar to chopping vegetables or combining ingredients when cooking.

4. Memory Access (MEM)

This phase interacts with memory in case the instruction is load or store data.

Modules engaged in this phase:

- **Data Memory:** Holds and recalls data for load (lw) and store (sw) instructions.
- **Control Signals:** MemRead and MemWrite decide to read from memory or write to memory.

Steps involved in Memory Access:

- If it is a load instruction (lw), memory is accessed at the calculated address, and the data read out is saved in a temporary buffer.

- If it is a store instruction (sw), the register value is saved to memory at the calculated address.
- If it is an arithmetic or branch instruction, this phase is bypassed.

Hazards in RISC-V Pipeline

A **hazard** occurs when the smooth execution of the pipeline is disrupted. Hazards can cause stalls (delays) or require additional logic to resolve conflicts. There are three main types of hazards:

1. Structural Hazards

- Occur when two or more instructions require the same hardware resource at the same time.
- Example: If the instruction memory and data memory are not separate, a fetch and a memory access operation may conflict.
- **Solution:** Use separate memory for instructions and data (Harvard Architecture) or introduce pipeline stalls.

2. Data Hazards

- Occur when an instruction depends on the result of a previous instruction that has not yet completed.

- Example:

```
add  x19, x0, x1
sub  x2, x19, x3
```

  Here x19 is being calculated in d=first instruction but we need it in next instruction and it is not available.

- Types of Data Hazards**:**

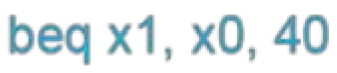
  - **RAW (Read After Write)**: Occurs when an instruction needs a value that has not been written yet.

- **WAR (Write After Read)**: Happens if a later instruction writes to a register before an earlier instruction reads from it (rare in RISC pipelines).
- **WAW (Write After Write)**: When two instructions write to the same register, causing incorrect values.

- Solution:
  - **Forwarding (Bypassing)**: Data is sent directly from one stage to another instead of waiting.
  - **Pipeline Stalls (NOP insertion)**: The processor inserts delay cycles until data is available.

3. Control Hazards (Branch Hazards)

- Occur when the pipeline does not know which instruction to fetch next due to a branch (e.g., `beq`, `bne`).
- Example:



```
beq x1, x0, 40
```

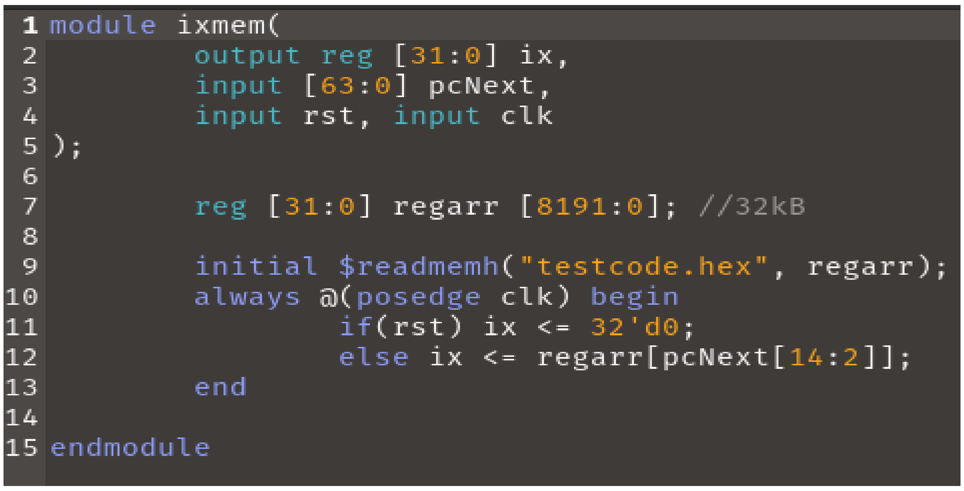
Solution:

- **Branch Prediction**: Guess whether the branch will be taken or not.
- **Delayed Branching**: Execute a useful instruction in the delay slot before the branch is resolved.
- **Flushing (Pipeline Stall)**: Stop fetching instructions until the branch decision is made.

EXPLANATION OF EACH BLOCK

Instruction Memory

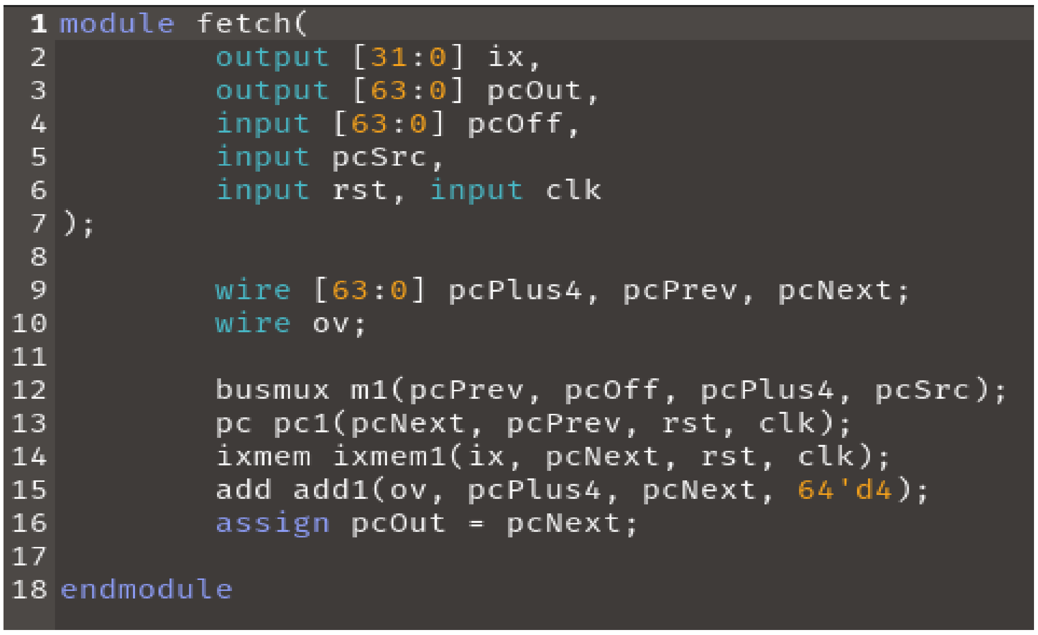
The `ixmem` module serves as an instruction memory, storing 32-bit instructions in an array (`regarr`) of size 8,192 words (32KB). The memory is initialized with instructions from the file `"testcode.hex"`. On each clock cycle, if `rst` is asserted, the instruction output (`ix`) is reset to zero. Otherwise, it fetches the instruction at the address indexed by `pcNext[14:2]`, effectively using `pcNext` as a word-aligned instruction pointer. This module ensures proper instruction retrieval during program execution.



```
module ixmem(
        output reg [31:0] ix,
        input [63:0] pcNext,
        input rst, input clk
);

        reg [31:0] regarr [8191:0]; //32kB

        initial $readmemh("testcode.hex", regarr);
        always @(posedge clk) begin
                if(rst) ix <= 32'd0;
                else ix <= regarr[pcNext[14:2]];
        end

endmodule
```

Fetch

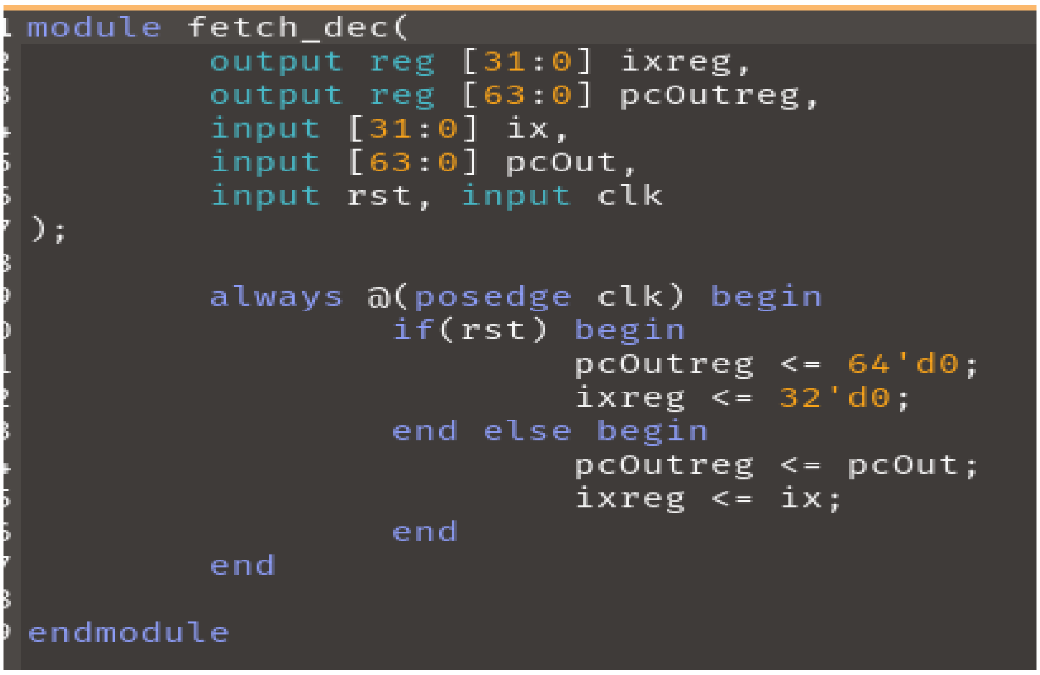
The `fetch` module is responsible for fetching instructions from memory and updating the program counter (PC). It computes the next PC value by either adding 4 (normal execution) or using a branch offset (`pcOff`) based on the `pcSrc` signal. A multiplexer (`busmux`) selects between these two values, and the updated PC is stored in a register (`pc`). The fetched instruction is obtained from instruction memory (`ixmem`) using this PC value.



```verilog
module fetch(
        output [31:0] ix,
        output [63:0] pcOut,
        input [63:0] pcOff,
        input pcSrc,
        input rst, input clk
);

        wire [63:0] pcPlus4, pcPrev, pcNext;
        wire ov;

        busmux m1(pcPrev, pcOff, pcPlus4, pcSrc);
        pc pc1(pcNext, pcPrev, rst, clk);
        ixmem ixmem1(ix, pcNext, rst, clk);
        add add1(ov, pcPlus4, pcNext, 64'd4);
        assign pcOut = pcNext;

endmodule
```

Fetch decode

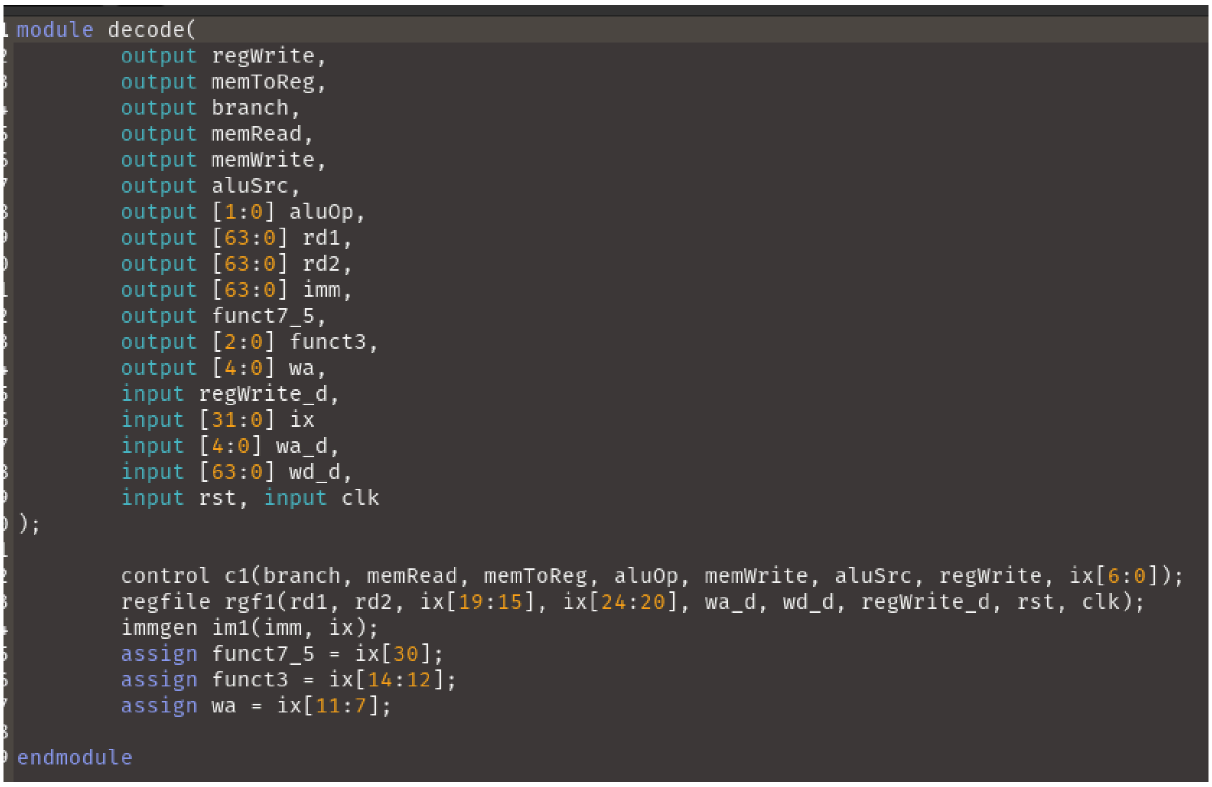
The `fetch_dec` module acts as a pipeline register between the fetch and decode stages. It latches the instruction (`ix`) and program counter (`pcOut`) on every clock cycle. If the reset (`rst`) signal is high, both registers are cleared to zero. Otherwise, the fetched instruction and PC value are stored and passed to the next stage of the pipeline.



```verilog
module fetch_dec(
        output reg [31:0] ixreg,
        output reg [63:0] pcOutreg,
        input [31:0] ix,
        input [63:0] pcOut,
        input rst, input clk
);

        always @(posedge clk) begin
                if(rst) begin
                        pcOutreg <= 64'd0;
                        ixreg <= 32'd0;
                end else begin
                        pcOutreg <= pcOut;
                        ixreg <= ix;
                end
        end

endmodule
```

Decode

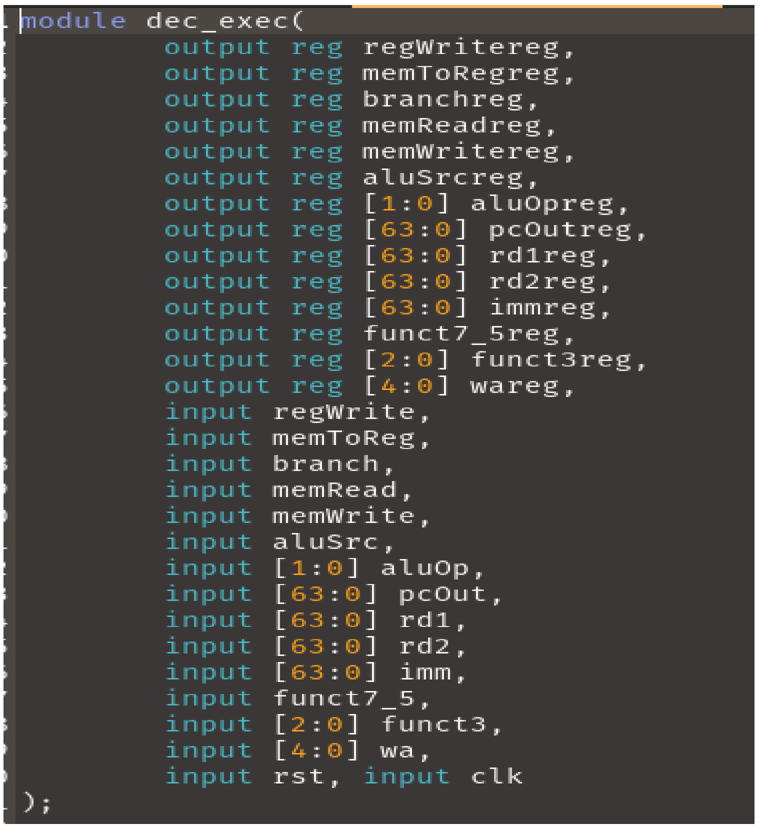
The decode module is responsible for decoding an instruction and generating control signals, register values, and immediate values for execution. It extracts opcode bits (ix[6:0]) to generate control signals using the control unit, which determines operations like aluSrc, memRead, and branch. The regfile (register file) reads source register values (rd1, rd2) based on instruction fields (ix[19:15], ix[24:20]) and writes back data (wd_d) when enabled. The immgen module extracts the immediate value from the instruction. Additionally, the module extracts function codes (funct7_5, funct3) and destination register address (wa) from the instruction for further processing in the pipeline.



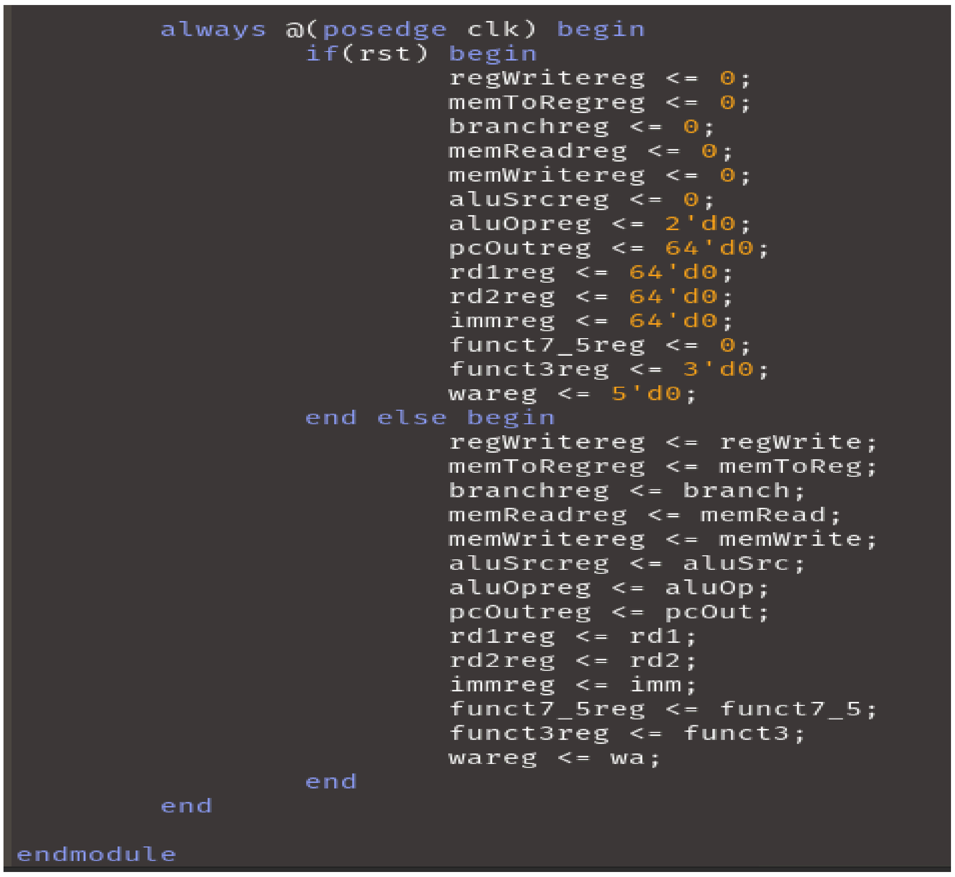
```
module decode(
        output regWrite,
        output memToReg,
        output branch,
        output memRead,
        output memWrite,
        output aluSrc,
        output [1:0] aluOp,
        output [63:0] rd1,
        output [63:0] rd2,
        output [63:0] imm,
        output funct7_5,
        output [2:0] funct3,
        output [4:0] wa,
        input regWrite_d,
        input [31:0] ix
        input [4:0] wa_d,
        input [63:0] wd_d,
        input rst, input clk
);

        control c1(branch, memRead, memToReg, aluOp, memWrite, aluSrc, regWrite, ix[6:0]);
        regfile rgf1(rd1, rd2, ix[19:15], ix[24:20], wa_d, wd_d, regWrite_d, rst, clk);
        immgen im1(imm, ix);
        assign funct7_5 = ix[30];
        assign funct3 = ix[14:12];
        assign wa = ix[11:7];

endmodule
```

Decode execute

The `dec_exec` module serves as a pipeline register between the decode and execute stages in a processor. It stores various control signals (`regWrite`, `memRead`, `aluSrc`, etc.), register values (`rd1`, `rd2`), the immediate value (`imm`), and instruction fields (`funct7_5`, `funct3`, `wa`) on every clock cycle. If `rst` is asserted, all stored values reset to zero; otherwise, it latches the input values, ensuring proper data forwarding to the execution stage. This module helps maintain data consistency and enables smooth pipelined execution.



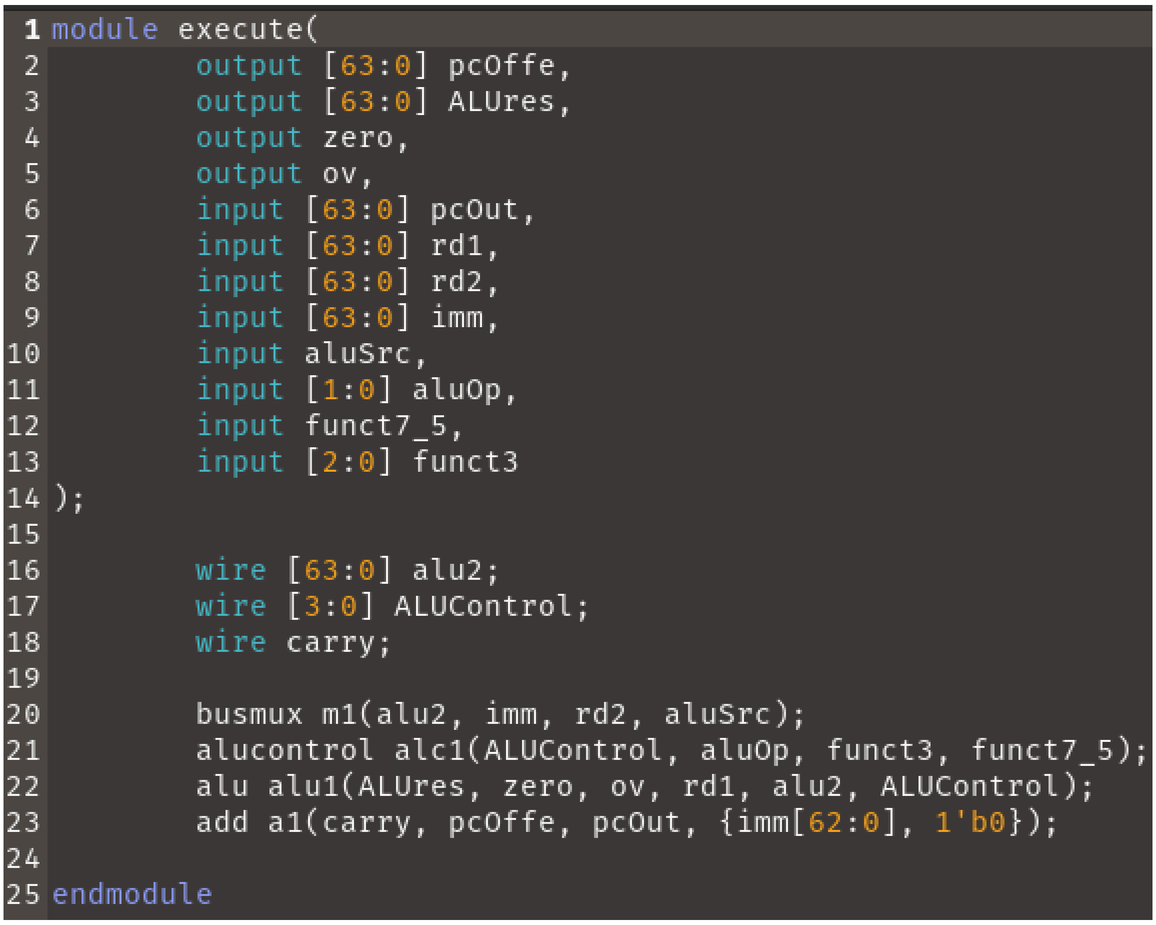
```
module dec_exec(
        output reg regWritereg,
        output reg memToRegreg,
        output reg branchreg,
        output reg memReadreg,
        output reg memWritereg,
        output reg aluSrcreg,
        output reg [1:0] aluOpreg,
        output reg [63:0] pcOutreg,
        output reg [63:0] rd1reg,
        output reg [63:0] rd2reg,
        output reg [63:0] immreg,
        output reg funct7_5reg,
        output reg [2:0] funct3reg,
        output reg [4:0] wareg,
        input regWrite,
        input memToReg,
        input branch,
        input memRead,
        input memWrite,
        input aluSrc,
        input [1:0] aluOp,
        input [63:0] pcOut,
        input [63:0] rd1,
        input [63:0] rd2,
        input [63:0] imm,
        input funct7_5,
        input [2:0] funct3,
        input [4:0] wa,
        input rst, input clk
);
```



```
        always @(posedge clk) begin
                if(rst) begin
                        regWritereg <= 0;
                        memToRegreg <= 0;
                        branchreg <= 0;
                        memReadreg <= 0;
                        memWritereg <= 0;
                        aluSrcreg <= 0;
                        aluOpreg <= 2'd0;
                        pcOutreg <= 64'd0;
                        rd1reg <= 64'd0;
                        rd2reg <= 64'd0;
                        immreg <= 64'd0;
                        funct7_5reg <= 0;
                        funct3reg <= 3'd0;
                        wareg <= 5'd0;
                end else begin
                        regWritereg <= regWrite;
                        memToRegreg <= memToReg;
                        branchreg <= branch;
                        memReadreg <= memRead;
                        memWritereg <= memWrite;
                        aluSrcreg <= aluSrc;
                        aluOpreg <= aluOp;
                        pcOutreg <= pcOut;
                        rd1reg <= rd1;
                        rd2reg <= rd2;
                        immreg <= imm;
                        funct7_5reg <= funct7_5;
                        funct3reg <= funct3;
                        wareg <= wa;
                end
        end

endmodule
```

Execute

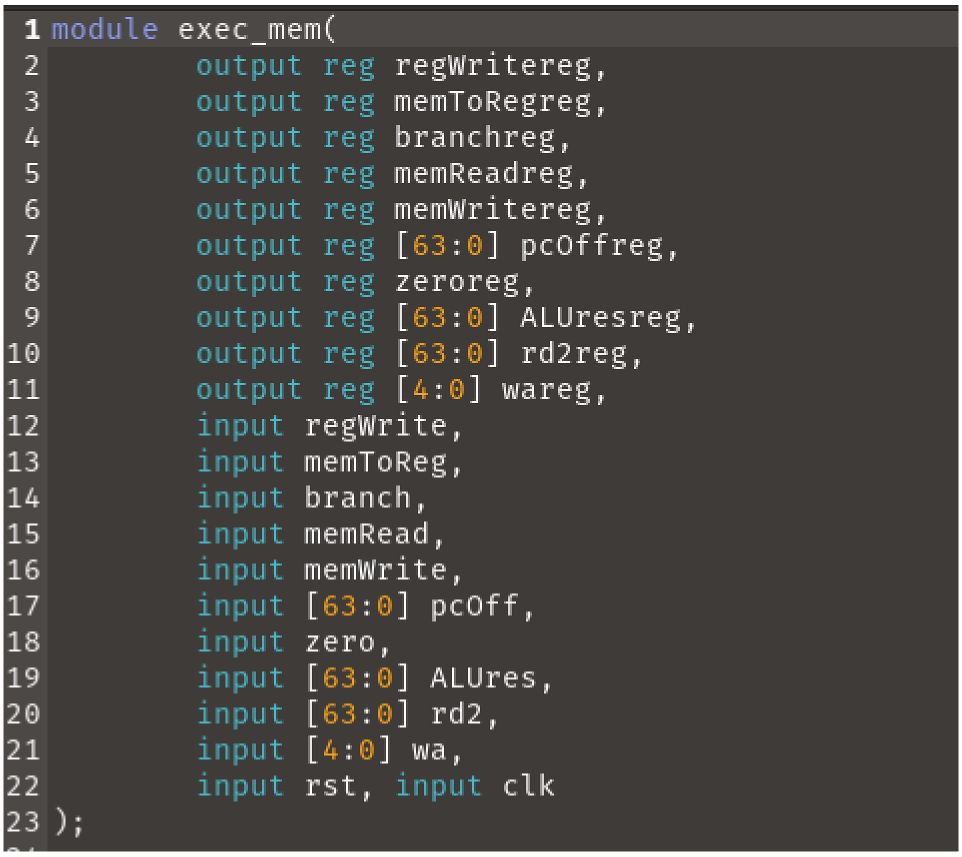
The `execute` module performs arithmetic and logical operations using the ALU and computes the branch target address. It selects the second ALU operand (`alu2`) using a multiplexer (`busmux`), choosing between `rd2` (register value) or `imm` (immediate value) based on `aluSrc`. The `alu control` module generates the ALU control signals based on `aluOp`, `funct3`, and `funct7_5`, which determine the operation type. The ALU (`alu`) then executes the operation and produces the result (`ALUres`), along with flags like `zero` and overflow (`ov`). Additionally, a separate adder (`a1`) computes the branch target address (`pcOffe`) by adding the shifted immediate value to `pcOut`, aiding in branch decision-making.



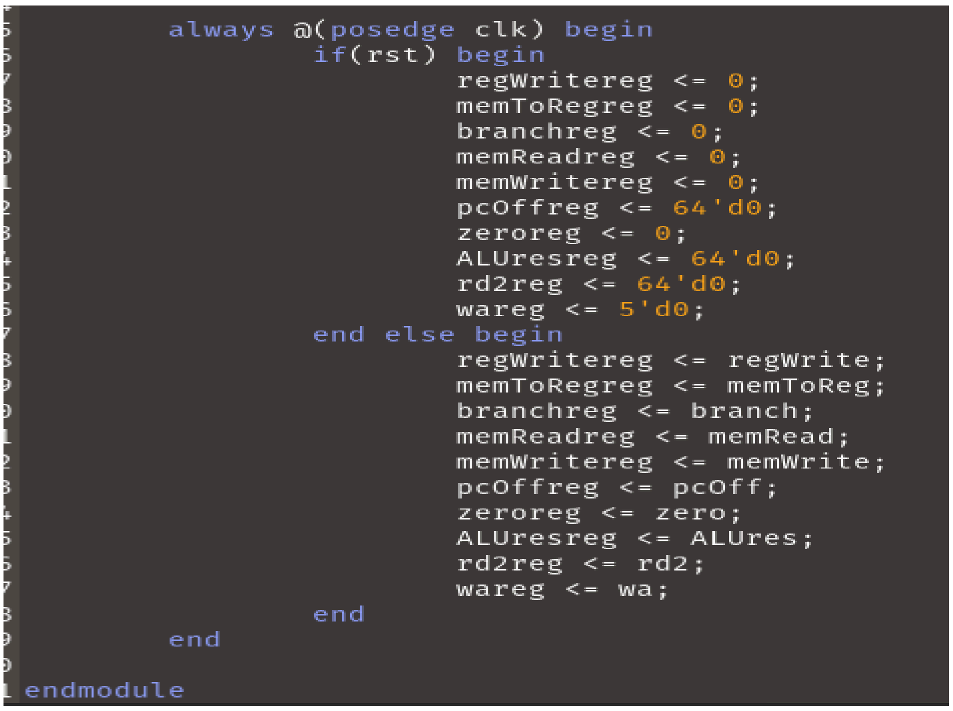
```
module execute(
        output [63:0] pcOffe,
        output [63:0] ALUres,
        output zero,
        output ov,
        input [63:0] pcOut,
        input [63:0] rd1,
        input [63:0] rd2,
        input [63:0] imm,
        input aluSrc,
        input [1:0] aluOp,
        input funct7_5,
        input [2:0] funct3
);

        wire [63:0] alu2;
        wire [3:0] ALUControl;
        wire carry;

        busmux m1(alu2, imm, rd2, aluSrc);
        alucontrol alc1(ALUControl, aluOp, funct3, funct7_5);
        alu alu1(ALUres, zero, ov, rd1, alu2, ALUControl);
        add a1(carry, pcOffe, pcOut, {imm[62:0], 1'b0});

endmodule
```

Execute Memory

The `exec_mem` module acts as a pipeline register between the execute and memory stages, storing control signals and execution results for the next stage. It latches values like ALU result (`ALUres`), branch target (`pcOff`), zero flag (`zero`), and the second register operand (`rd2`) on every clock cycle. If `rst` is asserted, all values reset to zero; otherwise, it updates the stored values with new inputs. This ensures that data and control signals are correctly passed to the memory stage while maintaining pipeline synchronization.



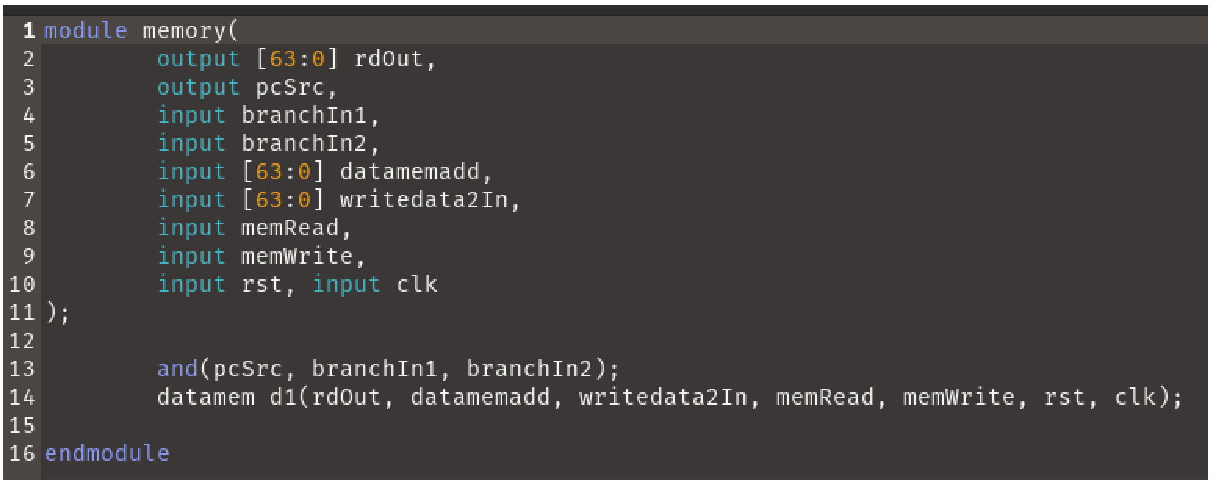
```verilog
module exec_mem(
        output reg regWritereg,
        output reg memToRegreg,
        output reg branchreg,
        output reg memReadreg,
        output reg memWritereg,
        output reg [63:0] pcOffreg,
        output reg zeroreg,
        output reg [63:0] ALUresreg,
        output reg [63:0] rd2reg,
        output reg [4:0] wareg,
        input regWrite,
        input memToReg,
        input branch,
        input memRead,
        input memWrite,
        input [63:0] pcOff,
        input zero,
        input [63:0] ALUres,
        input [63:0] rd2,
        input [4:0] wa,
        input rst, input clk
);
```



```verilog
        always @(posedge clk) begin
                if(rst) begin
                        regWritereg <= 0;
                        memToRegreg <= 0;
                        branchreg <= 0;
                        memReadreg <= 0;
                        memWritereg <= 0;
                        pcOffreg <= 64'd0;
                        zeroreg <= 0;
                        ALUresreg <= 64'd0;
                        rd2reg <= 64'd0;
                        wareg <= 5'd0;
                end else begin
                        regWritereg <= regWrite;
                        memToRegreg <= memToReg;
                        branchreg <= branch;
                        memReadreg <= memRead;
                        memWritereg <= memWrite;
                        pcOffreg <= pcOff;
                        zeroreg <= zero;
                        ALUresreg <= ALUres;
                        rd2reg <= rd2;
                        wareg <= wa;
                end
        end

endmodule
```

Memory

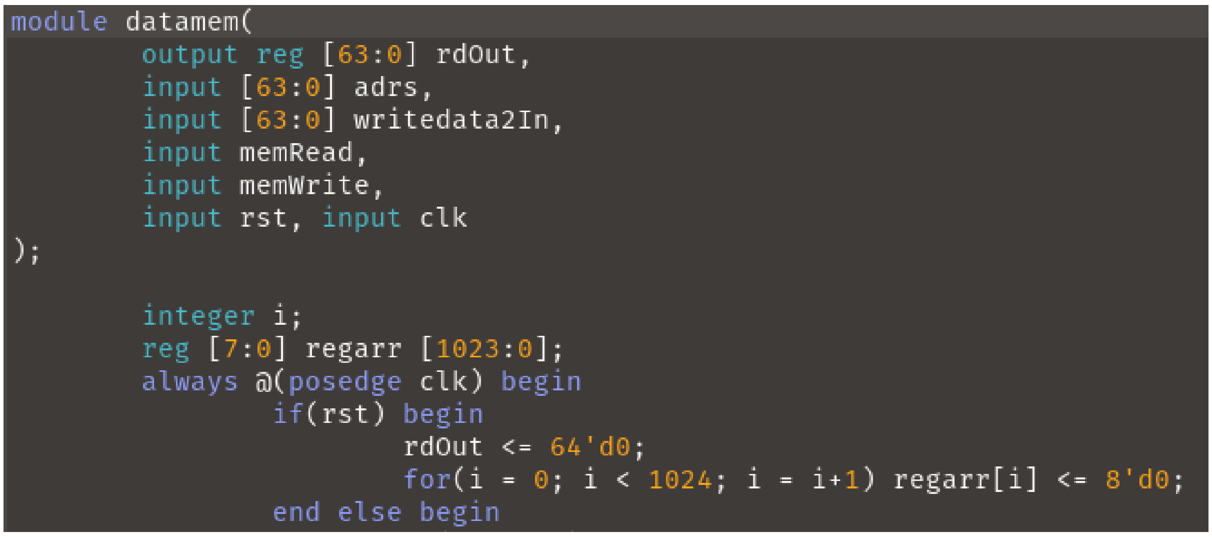
The `memory` module handles memory operations and branch decision-making. It determines whether to take a branch by performing an AND operation on branchIn1 and branchIn2, setting pcSrc accordingly. The datamem module reads from or writes to memory at the address datamemadd, using writedata2In as input data when memWrite is enabled. If memRead is active, it outputs the read data (rdOut). The module ensures correct memory access and branch control within the pipeline.



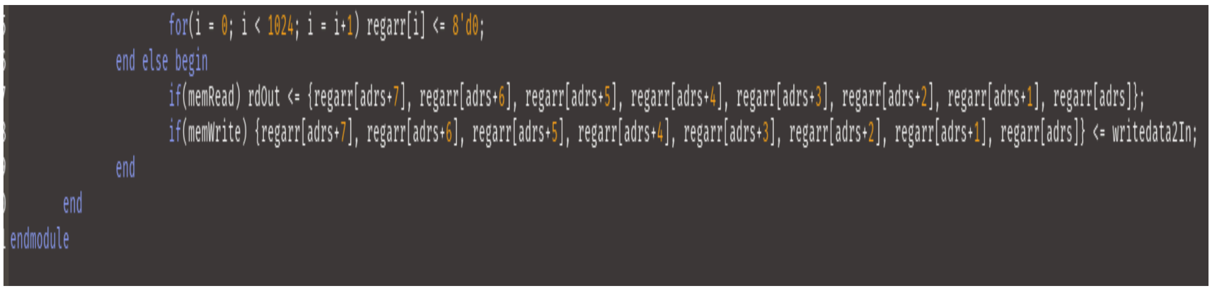
```
module memory(
        output [63:0] rdOut,
        output pcSrc,
        input branchIn1,
        input branchIn2,
        input [63:0] datamemadd,
        input [63:0] writedata2In,
        input memRead,
        input memWrite,
        input rst, input clk
);

        and(pcSrc, branchIn1, branchIn2);
        datamem d1(rdOut, datamemadd, writedata2In, memRead, memWrite, rst, clk);

endmodule
```

Data Memory

The datamem module implements a simple data memory using an array of 1024 bytes (regarr). On reset (rst), all memory locations are initialized to zero, and rdOut is cleared. When memRead is asserted, the module reads 64 bits (8 consecutive bytes) from the specified address (adrs) and outputs it as rdOut. If memWrite is active, it writes the 64-bit input (writedata2In) to the corresponding memory location. The module ensures proper memory read and write operations synchronized with the clock (clk).



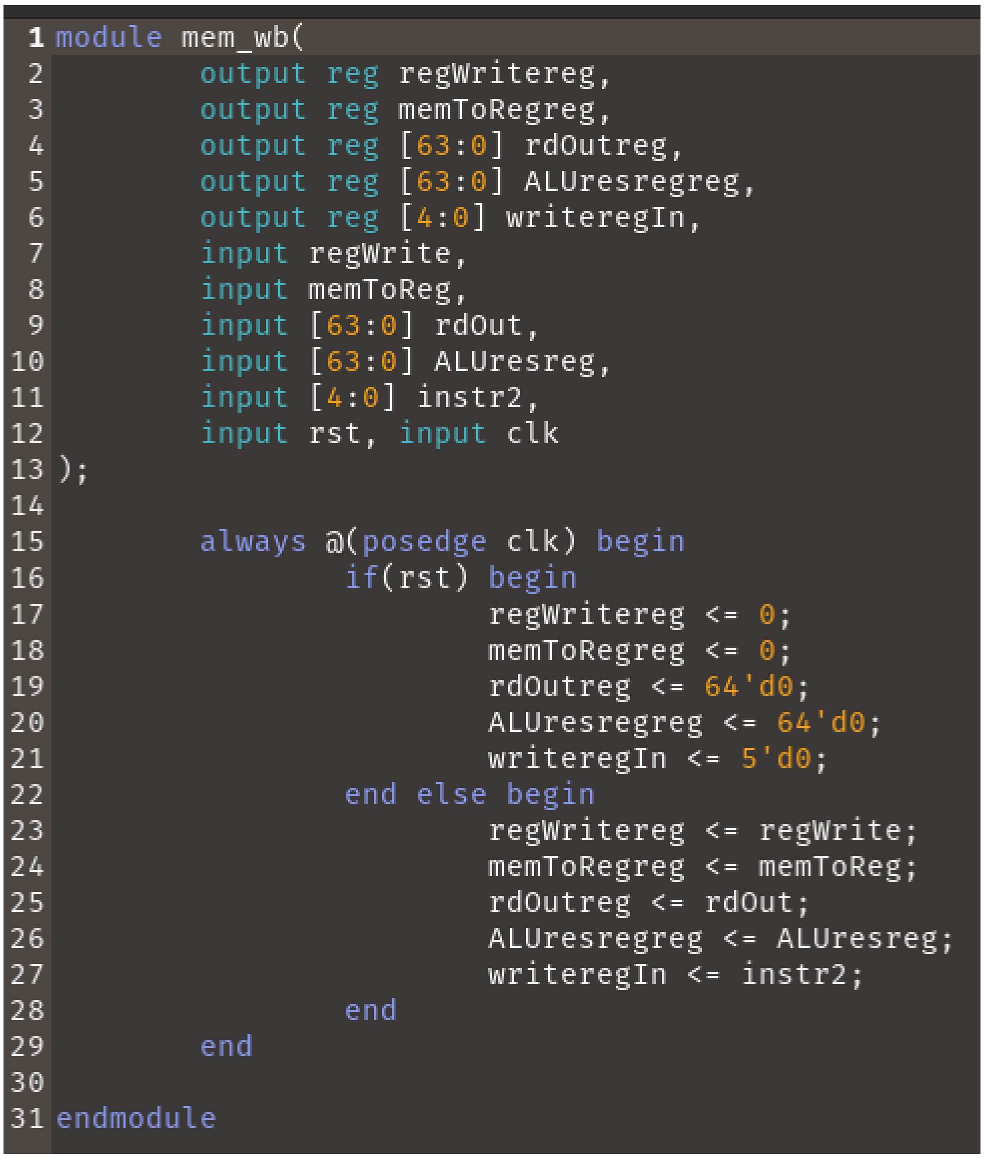
```
module datamem(
        output reg [63:0] rdOut,
        input [63:0] adrs,
        input [63:0] writedata2In,
        input memRead,
        input memWrite,
        input rst, input clk
);

        integer i;
        reg [7:0] regarr [1023:0];
        always @(posedge clk) begin
                if(rst) begin
                        rdOut <= 64'd0;
                        for(i = 0; i < 1024; i = i+1) regarr[i] <= 8'd0;
                end else begin
```



```
                        for(i = 0; i < 1024; i = i+1) regarr[i] <= 8'd0;
                end else begin
                        if(memRead) rdOut <= {regarr[adrs+7], regarr[adrs+6], regarr[adrs+5], regarr[adrs+4], regarr[adrs+3], regarr[adrs+2], regarr[adrs+1], regarr[adrs]};
                        if(memWrite) {regarr[adrs+7], regarr[adrs+6], regarr[adrs+5], regarr[adrs+4], regarr[adrs+3], regarr[adrs+2], regarr[adrs+1], regarr[adrs]} <= writedata2In;
                end
        end
endmodule
```

<u>Memory Write Back</u>

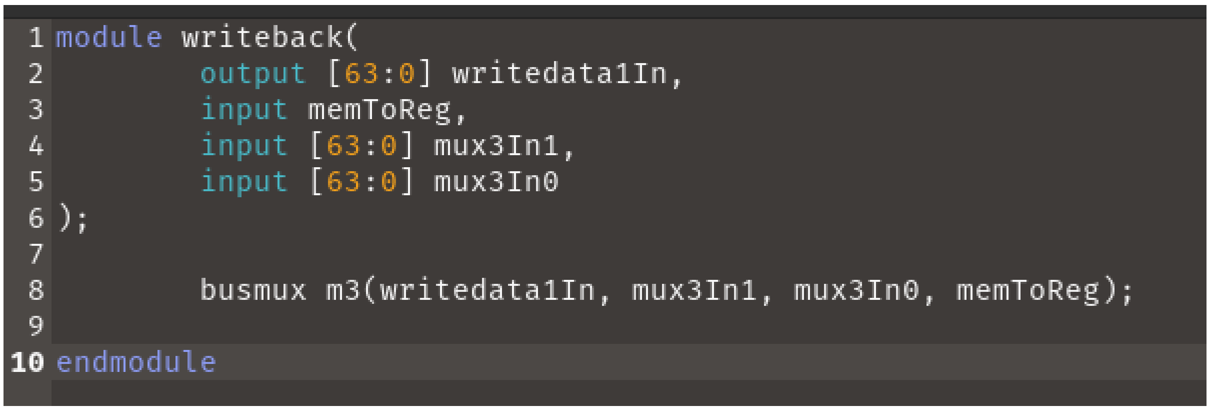
The `mem_wb` module acts as a pipeline register between the memory and write-back stages, storing values needed for writing results back to registers. It latches control signals (`regWrite`, `memToReg`), memory output (`rdOut`), ALU result (`ALUresreg`), and destination register (`instr2`) on each clock cycle. If `rst` is asserted, all stored values reset to zero; otherwise, they update with new inputs. This ensures that the write-back stage receives the correct data for register updates, maintaining smooth pipeline execution.



```
module mem_wb(
        output reg regWritereg,
        output reg memToRegreg,
        output reg [63:0] rdOutreg,
        output reg [63:0] ALUresregreg,
        output reg [4:0] writeregIn,
        input regWrite,
        input memToReg,
        input [63:0] rdOut,
        input [63:0] ALUresreg,
        input [4:0] instr2,
        input rst, input clk
);

        always @(posedge clk) begin
                if(rst) begin
                        regWritereg <= 0;
                        memToRegreg <= 0;
                        rdOutreg <= 64'd0;
                        ALUresregreg <= 64'd0;
                        writeregIn <= 5'd0;
                end else begin
                        regWritereg <= regWrite;
                        memToRegreg <= memToReg;
                        rdOutreg <= rdOut;
                        ALUresregreg <= ALUresreg;
                        writeregIn <= instr2;
                end
        end

endmodule
```

Write Back

The `writeback` module selects the final data to be written back to the register file. It uses a multiplexer (`busmux`) to choose between the memory output (`mux3In1`) and the ALU result (`mux3In0`), based on the `memToReg` control signal. If `memToReg` is high, the memory data is written back; otherwise, the ALU result is used. This ensures that the correct value is stored in the destination register during the write-back stage of the pipeline.



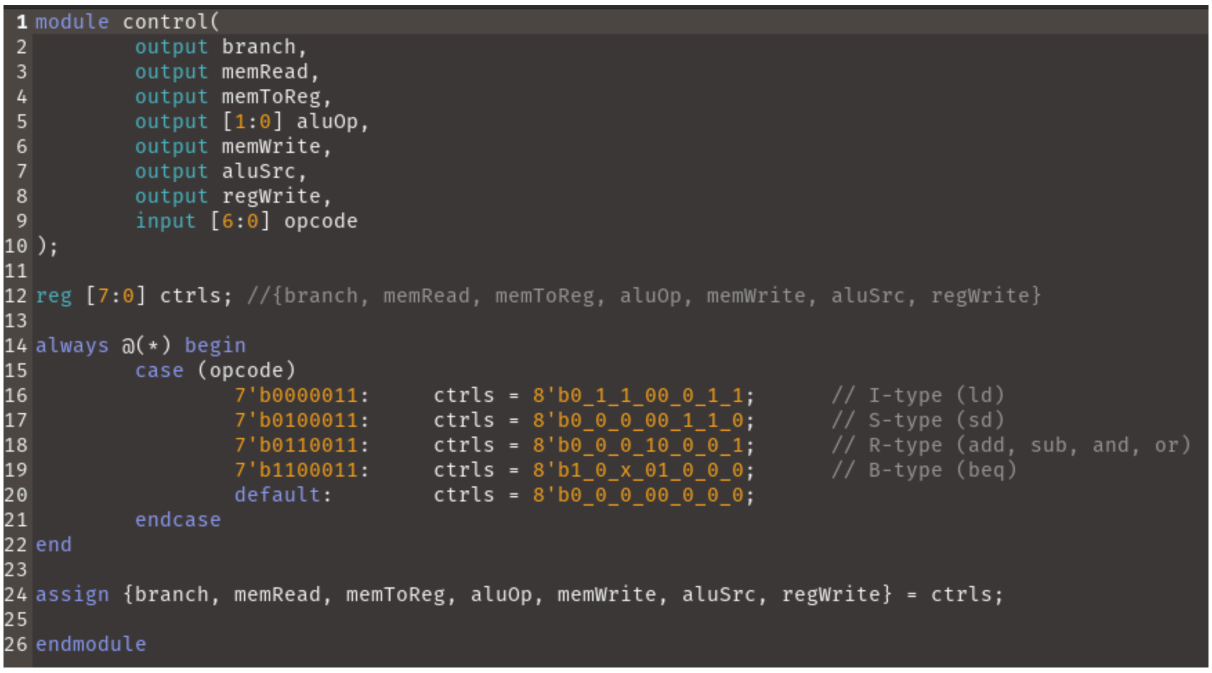
```
module writeback(
        output [63:0] writedata1In,
        input memToReg,
        input [63:0] mux3In1,
        input [63:0] mux3In0
);

        busmux m3(writedata1In, mux3In1, mux3In0, memToReg);

endmodule
```

Control

The `control` module is responsible for generating control signals based on the `opcode` of an instruction. It determines how different components of the processor should behave during execution. The module assigns values to control signals such as `branch`, `memRead`, `memToReg`, `aluOp`, `memWrite`, `aluSrc`, and `regWrite` depending on the instruction type:

- **Load (`ld`, I-type)**: Enables memory read (`memRead`), memory-to-register (`memToReg`), and register write (`regWrite`).
- **Store (`sd`, S-type)**: Enables memory write (`memWrite`) but disables `memToReg` and `regWrite`.
- **R-type (`add`, `sub`, `and`, `or`)**: Uses ALU (`aluOp = 10`), without memory operations.
- **Branch (`beq`, B-type)**: Enables branching (`branch = 1`), allowing conditional jumps.
- **Default**: Sets all control signals to `0`, preventing unintended behavior.

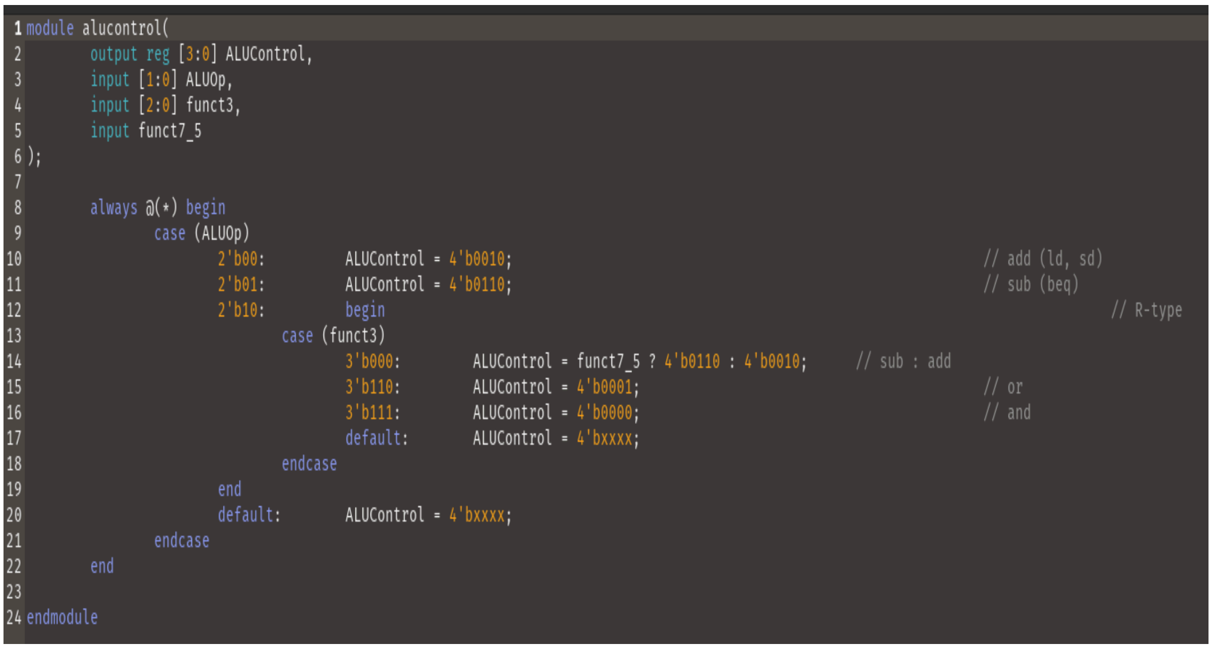
This module plays a crucial role in directing the datapath of the processor based on the type of instruction being executed.



```verilog
module control(
        output branch,
        output memRead,
        output memToReg,
        output [1:0] aluOp,
        output memWrite,
        output aluSrc,
        output regWrite,
        input [6:0] opcode
);

reg [7:0] ctrls; //{branch, memRead, memToReg, aluOp, memWrite, aluSrc, regWrite}

always @(*) begin
        case (opcode)
                7'b0000011:     ctrls = 8'b0_1_1_00_0_1_1;      // I-type (ld)
                7'b0100011:     ctrls = 8'b0_0_0_00_1_1_0;      // S-type (sd)
                7'b0110011:     ctrls = 8'b0_0_0_10_0_0_1;      // R-type (add, sub, and, or)
                7'b1100011:     ctrls = 8'b1_0_x_01_0_0_0;      // B-type (beq)
                default:        ctrls = 8'b0_0_0_00_0_0_0;
        endcase
end

assign {branch, memRead, memToReg, aluOp, memWrite, aluSrc, regWrite} = ctrls;

endmodule
```

ALU CONTROL

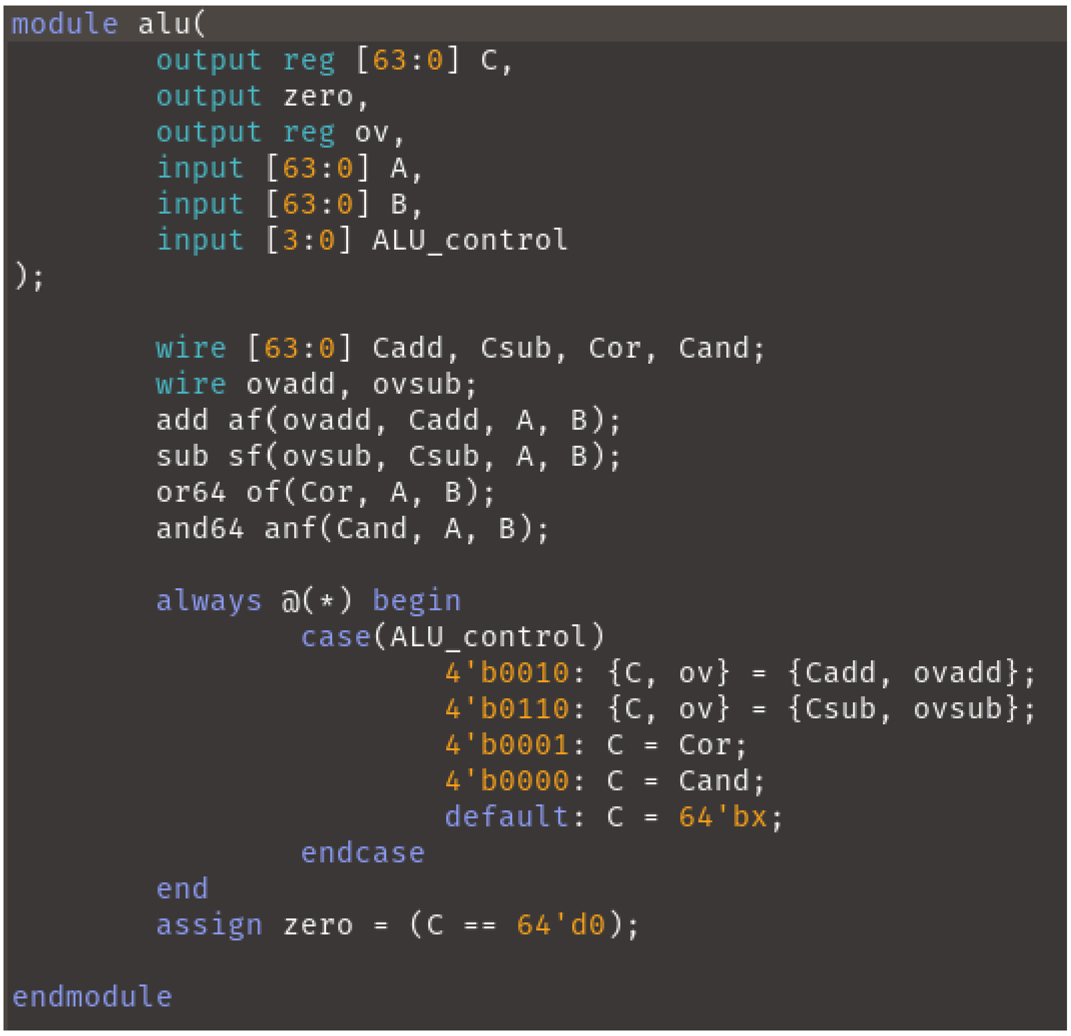
The `alu control` module generates the control signals for the ALU based on the `ALUOp` signal and instruction function codes (`funct3`, `funct7_5`). For load (`ld`) and store (`sd`) instructions (`ALUOp = 00`), it sets the ALU to perform addition (`0010`). For branch (`beq`, `ALUOp = 01`), it sets subtraction (`0110`). For R-type instructions (`ALUOp = 10`), it checks `funct3` to determine the specific ALU operation: addition or subtraction (`funct7_5`), OR (`0001`), or AND (`0000`). If the instruction is invalid, it assigns an undefined value (`xxxx`). This module ensures the ALU executes the correct operation based on the instruction type.



```
module alucontrol(
        output reg [3:0] ALUControl,
        input [1:0] ALUOp,
        input [2:0] funct3,
        input funct7_5
);

        always @(*) begin
                case (ALUOp)
                        2'b00:          ALUControl = 4'b0010;                                                           // add (ld, sd)
                        2'b01:          ALUControl = 4'b0110;                                                           // sub (beq)
                        2'b10:          begin                                                                                           // R-type
                                case (funct3)
                                        3'b000:         ALUControl = funct7_5 ? 4'b0110 : 4'b0010;      // sub : add
                                        3'b110:         ALUControl = 4'b0001;                                           // or
                                        3'b111:         ALUControl = 4'b0000;                                           // and
                                        default:        ALUControl = 4'bxxxx;
                                endcase
                        end
                        default:        ALUControl = 4'bxxxx;
                endcase
        end

endmodule
```

ALU

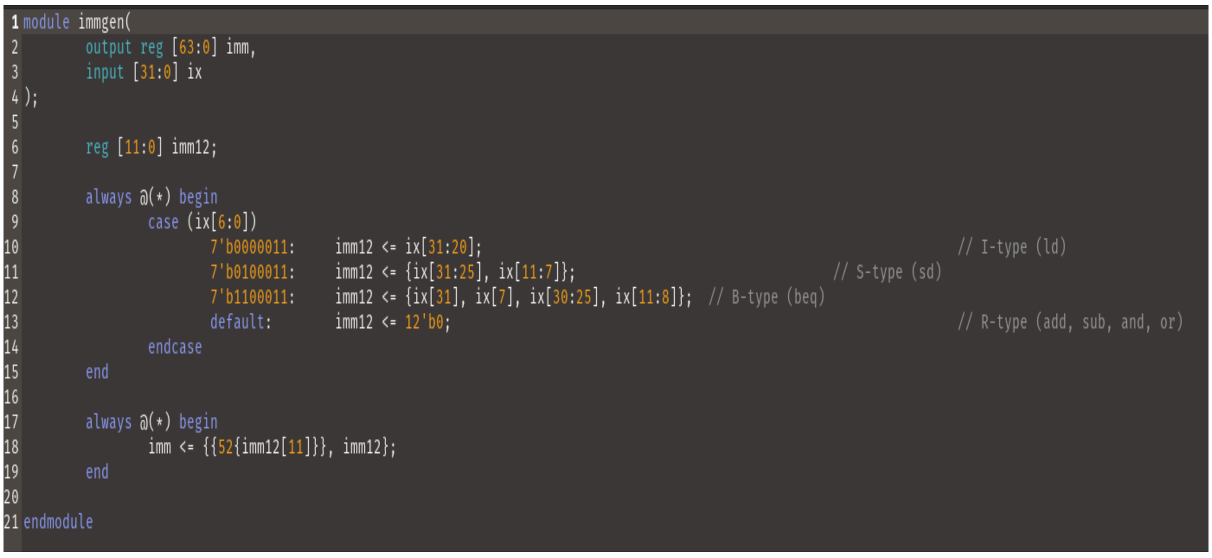
The alu module performs arithmetic and logical operations based on the ALU_control signal. It supports addition (add), subtraction (sub), bitwise OR (or64), and bitwise AND (and64). Overflow flags (ovadd, ovsub) are generated for addition and subtraction operations. The output C holds the result of the selected operation, while zero is asserted when C is zero. The module ensures correct execution of ALU operations, handling overflow for arithmetic instructions.



```
module alu(
        output reg [63:0] C,
        output zero,
        output reg ov,
        input [63:0] A,
        input [63:0] B,
        input [3:0] ALU_control
);

        wire [63:0] Cadd, Csub, Cor, Cand;
        wire ovadd, ovsub;
        add af(ovadd, Cadd, A, B);
        sub sf(ovsub, Csub, A, B);
        or64 of(Cor, A, B);
        and64 anf(Cand, A, B);

        always @(*) begin
                case(ALU_control)
                        4'b0010: {C, ov} = {Cadd, ovadd};
                        4'b0110: {C, ov} = {Csub, ovsub};
                        4'b0001: C = Cor;
                        4'b0000: C = Cand;
                        default: C = 64'bx;
                endcase
        end
        assign zero = (C == 64'd0);

endmodule
```

Immediate Generator

The `immgen` module extracts and sign-extends the immediate value from the instruction (`ix`) based on its format. For **I-type (ld)**, it takes bits `[31:20]`; for **S-type (sd)**, it combines `[31:25]` and `[11:7]`; and for **B-type (beq)**, it rearranges bits `[31]`, `[7]`, `[30:25]`, and `[11:8]`. R-type instructions, which do not use an immediate, default to zero. The extracted 12-bit immediate is then sign-extended to 64 bits to preserve the correct value for arithmetic operations.



```verilog
module immgen(
        output reg [63:0] imm,
        input [31:0] ix
);

        reg [11:0] imm12;

        always @(*) begin
                case (ix[6:0])
                        7'b0000011:     imm12 <= ix[31:20];                                              // I-type (ld)
                        7'b0100011:     imm12 <= {ix[31:25], ix[11:7]};                          // S-type (sd)
                        7'b1100011:     imm12 <= {ix[31], ix[7], ix[30:25], ix[11:8]};  // B-type (beq)
                        default:        imm12 <= 12'b0;                                                  // R-type (add, sub, and, or)
                endcase
        end

        always @(*) begin
                imm <= {{52{imm12[11]}}, imm12};
        end

endmodule
```

Test Case for Implementation

SEQUENTIAL

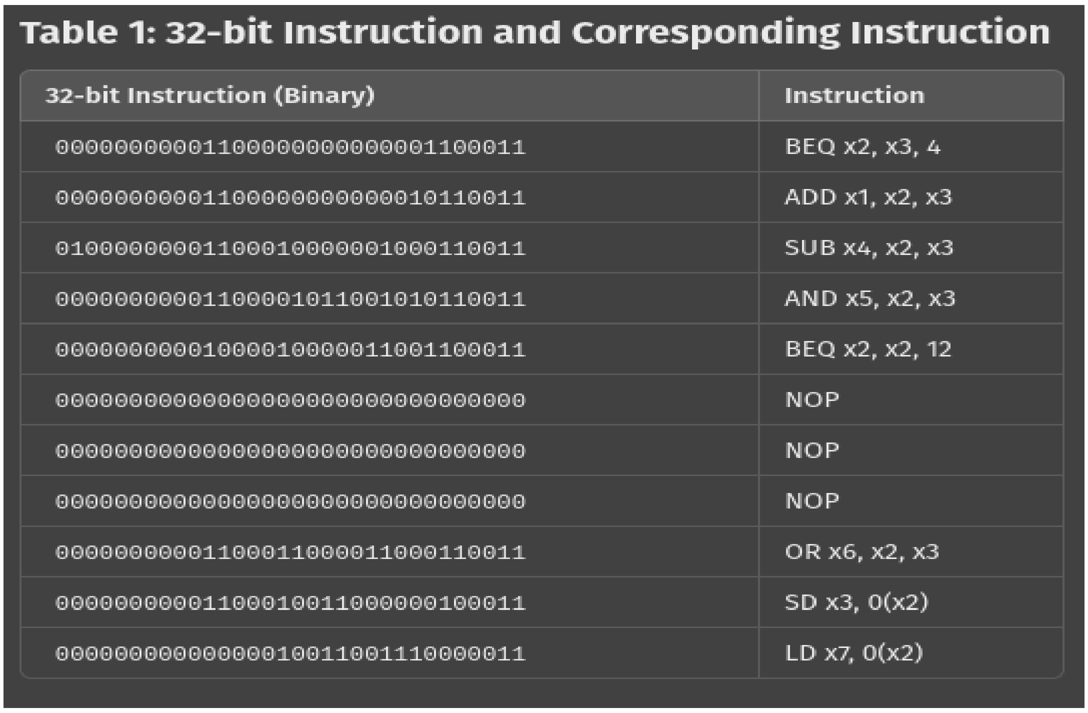


Table 1: 32-bit Instruction and Corresponding Instruction

32-bit Instruction (Binary)	Instruction
00000000001100000000000001100011	BEQ x2, x3, 4
00000000001100000000000010110011	ADD x1, x2, x3
01000000001100010000001000110011	SUB x4, x2, x3
00000000001100001011001010110011	AND x5, x2, x3
00000000001000010000011001100011	BEQ x2, x2, 12
00000000000000000000000000000000	NOP
00000000000000000000000000000000	NOP
00000000000000000000000000000000	NOP
00000000001100011000011000110011	OR x6, x2, x3
00000000001100010011000000100011	SD x3, 0(x2)
00000000000000010011001110000011	LD x7, 0(x2)

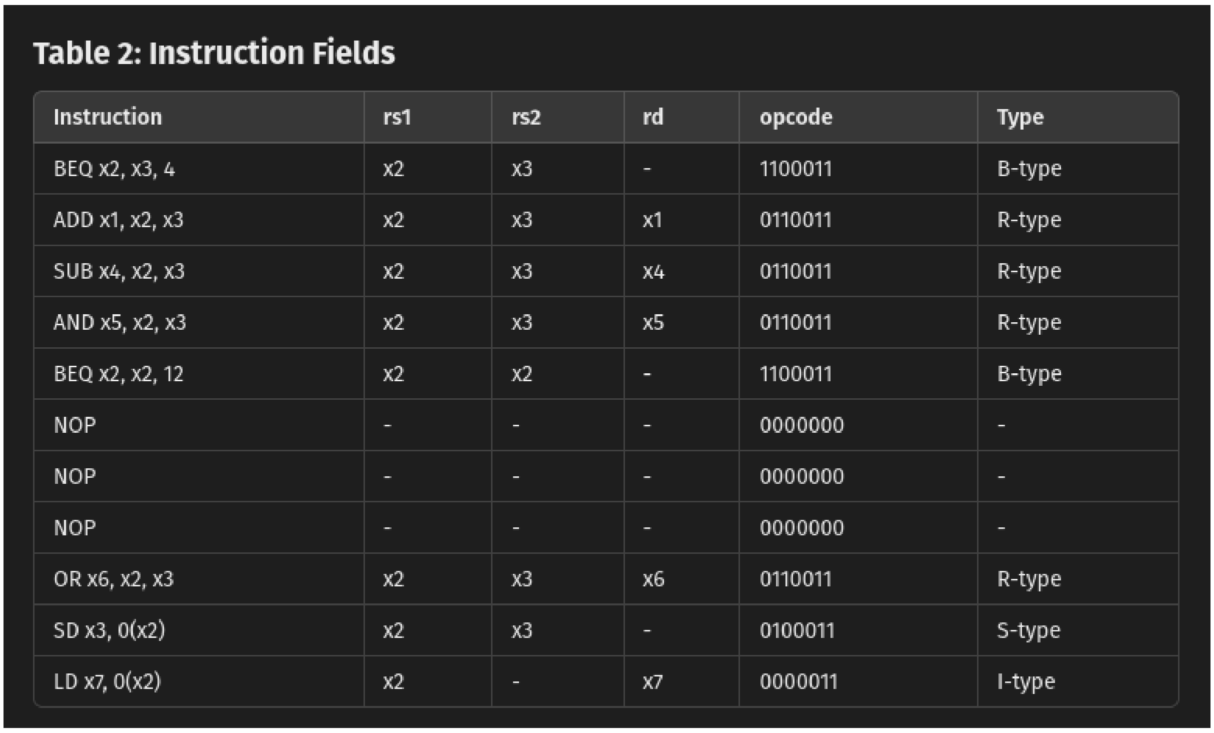
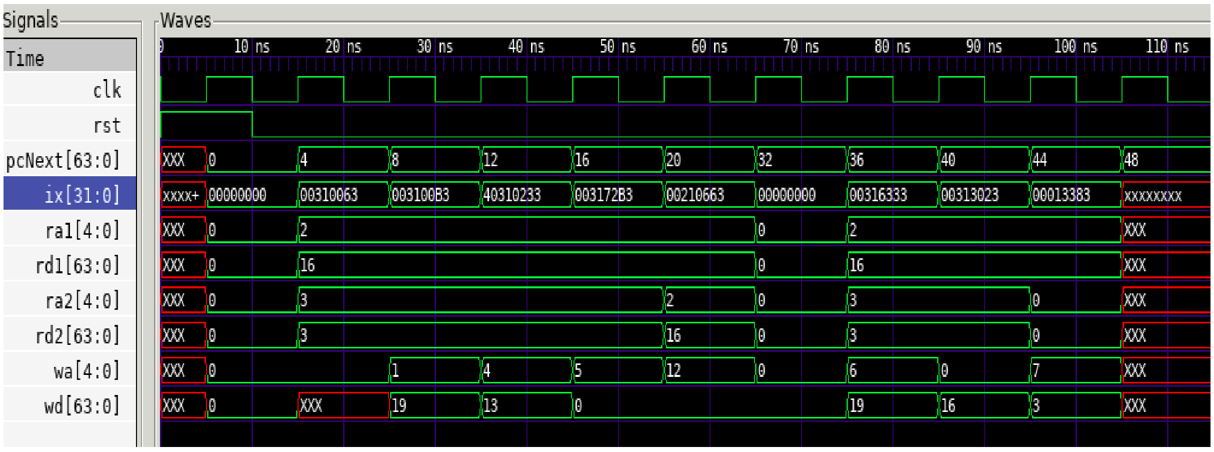


Table 2: Instruction Fields

Instruction	rs1	rs2	rd	opcode	Type
BEQ x2, x3, 4	x2	x3	-	1100011	B-type
ADD x1, x2, x3	x2	x3	x1	0110011	R-type
SUB x4, x2, x3	x2	x3	x4	0110011	R-type
AND x5, x2, x3	x2	x3	x5	0110011	R-type
BEQ x2, x2, 12	x2	x2	-	1100011	B-type
NOP	-	-	-	0000000	-
NOP	-	-	-	0000000	-
NOP	-	-	-	0000000	-
OR x6, x2, x3	x2	x3	x6	0110011	R-type
SD x3, 0(x2)	x2	x3	-	0100011	S-type
LD x7, 0(x2)	x2	-	x7	0000011	I-type



```
pcNext[63:0]  XXX 0 4 8 12 16 20 32 36 40 44 48
ix[31:0]      xxxx+ 00000000 00310063 003100B3 40310233 003172B3 00210663 00000000 00316333 00313023 00013383 xxxxxxxx
ra1[4:0]      XXX 0 2 0 2 XXX
rd1[63:0]     XXX 0 16 0 16 XXX
ra2[4:0]      XXX 0 3 2 0 3 0 XXX
rd2[63:0]     XXX 0 3 16 0 3 0 XXX
wa[4:0]       XXX 0 1 4 5 12 0 6 0 7 XXX
wd[63:0]      XXX 0 XXX 19 13 0 19 16 3 XXX
```

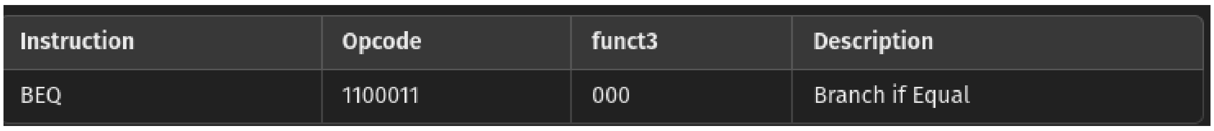
GTK WAVE REPRESENTATION

1. <u>**BEQ (Branch if Equal)**</u>

Format (B-Type Instruction)
BEQ rs1, rs2, offset

- Compares the values in registers rs1 and rs2.
- If they are equal, the program counter (PC) branches to the address determined by the offset.
- Otherwise, execution continues sequentially.

Opcode & Function Codes



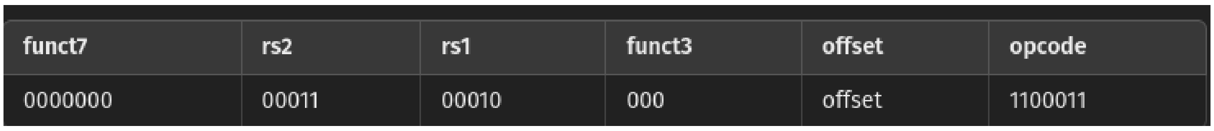
Instruction	Opcode	funct3	Description
BEQ	1100011	000	Branch if Equal

Execution Steps

1. Instruction Fetch:
    - PC provides the instruction address to instruction memory.
    - Instruction memory fetches the BEQ instruction.
    - PC increments by 4 (PC = PC + 4).
2. Instruction Decode:
    - The Control Unit decodes the instruction.
    - Determines it is a BEQ operation.
    - Sets control signals: Branch = 1, ALUOp = 01 (comparison).
3. Register Read:
    - Register file reads values from rs1 and rs2.
4. ALU Execution:
    - ALU performs comparison:
      Zero = (rs1 == rs2) ? 1 : 0
5. Branch Decision:
    - If Zero == 1, PC is updated with PC + offset.
    - If Zero == 0, execution continues sequentially.

Instruction Breakdown

32-bit Instruction: 00000000001100010000000001100011



funct7	rs2	rs1	funct3	offset	opcode
0000000	00011	00010	000	offset	1100011

Assembly Instruction:
BEQ x2, x3, 4

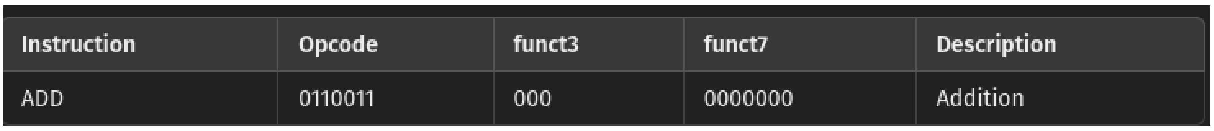
- If x2 == x3, execution jumps 4 instructions ahead.

2. **ADD (Addition)**

Format (R-Type Instruction)
ADD rd, rs1, rs2

- Adds the values in registers rs1 and rs2, storing the result in rd.

Opcode & Function Codes:



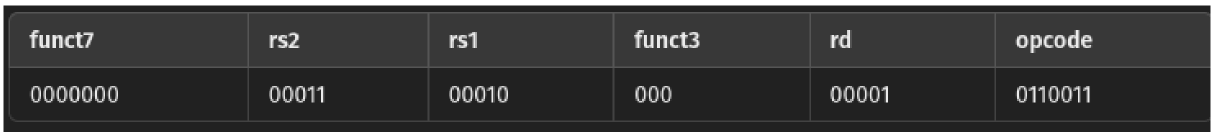
Instruction	Opcode	funct3	funct7	Description
ADD	0110011	000	0000000	Addition

Execution Steps

1. Instruction Fetch:
   - PC fetches the ADD instruction.
   - PC increments by 4 (PC = PC + 4).
2. Instruction Decode:
   - Control Unit decodes the instruction.
   - Identifies an ADD operation.
   - Sets control signals: RegWrite = 1, ALUOp = 10 (R-type operation).
3. Register Read:
   - Register file reads values from rs1 and rs2.
4. ALU Execution:
   - ALU performs addition:
     Result = rs1 + rs2
5. Write Back:
   - Result is written back to register rd.

Instruction Breakdown

32-bit Instruction: 00000000001100010000000010110011



funct7	rs2	rs1	funct3	rd	opcode
0000000	00011	00010	000	00001	0110011

Assembly Instruction:
ADD x1, x2, x3

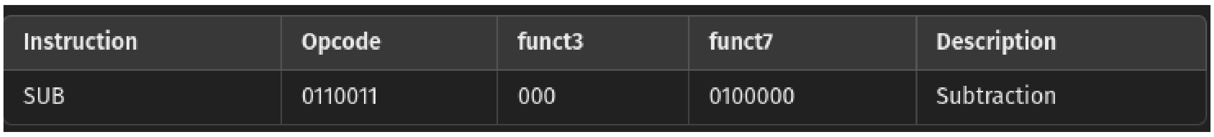
- Adds the values in x2 and x3, storing the result in x1.

3. **SUB (Subtraction)**

Format (R-Type Instruction)
SUB rd, rs1, rs2

- Subtracts rs2 from rs1 and stores the result in rd.

Opcode & Function Codes:



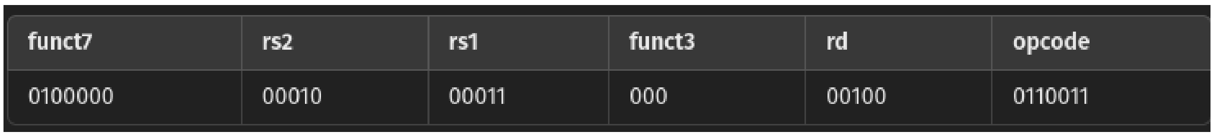
Instruction	Opcode	funct3	funct7	Description
SUB	0110011	000	0100000	Subtraction

Execution Steps

1. Instruction Fetch:
   - PC fetches the SUB instruction.
   - PC increments by 4 (PC = PC + 4).
2. Instruction Decode:
   - Control Unit decodes the instruction.
   - Identifies a SUB operation.
   - Sets control signals: RegWrite = 1, ALUOp = 10 (R-type operation).
3. Register Read:
   - Register file reads values from rs1 and rs2.
4. ALU Execution:
   - ALU performs subtraction:
     Result = rs1 - rs2
5. Write Back:
   - Result is written back to register rd.

Instruction Breakdown

32-bit Instruction: 01000000010000011000001000110011



funct7	rs2	rs1	funct3	rd	opcode
0100000	00010	00011	000	00100	0110011

Assembly Instruction:
SUB x4, x2, x3

- Subtract x3 from x2, storing the result in x4.

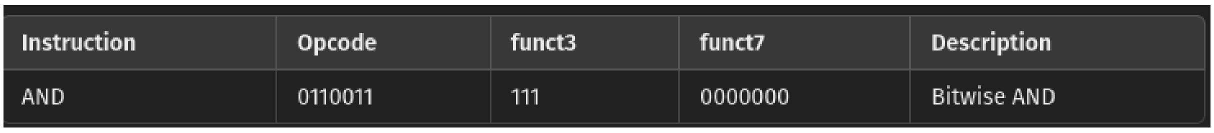
---

4. **AND (Bitwise AND)**

Format (R-Type Instruction)
AND rd, rs1, rs2

- Performs a bitwise AND operation between rs1 and rs2, storing the result in rd.

Opcode & Function Codes



Instruction	Opcode	funct3	funct7	Description
AND	0110011	111	0000000	Bitwise AND

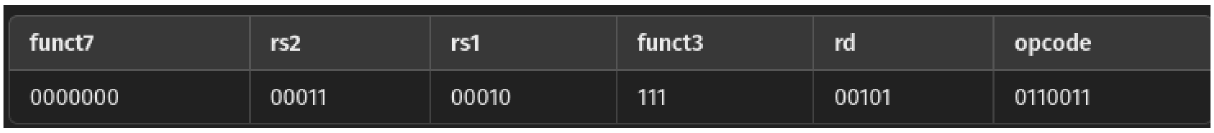
Execution Steps

1. Instruction Fetch:
    - PC fetches the AND instruction.
    - PC increments by 4 (PC = PC + 4).
2. Instruction Decode:
    - Control Unit decodes the instruction.
    - Identifies an AND operation.
    - Sets control signals: RegWrite = 1, ALUOp = 10 (R-type operation).
3. Register Read:
    - Register file reads values from rs1 and rs2.
4. ALU Execution:

    - ALU performs bitwise AND:
      Result = rs1 & rs2
5. Write Back:
    - Result is written back to register rd.

Instruction Breakdown

32-bit Instruction: 00000000001100010111001010110011



funct7	rs2	rs1	funct3	rd	opcode
0000000	00011	00010	111	00101	0110011

Assembly Instruction:
 AND x5, x2, x3

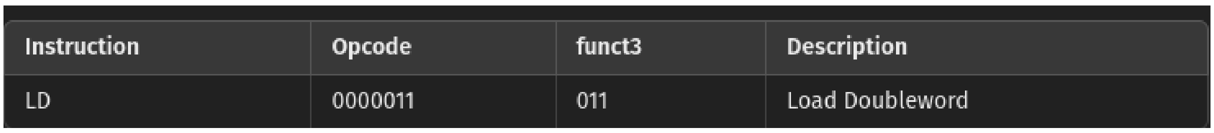
- Performs x2 & x3, storing the result in x5.

5. **LD (Load Doubleword)**

Format (I-Type Instruction)
 LD rd, offset(rs1)

- Loads a 64-bit value from memory at address rs1 + offset and stores it in register rd.

Opcode & Function Codes:



Instruction	Opcode	funct3	Description
LD	0000011	011	Load Doubleword

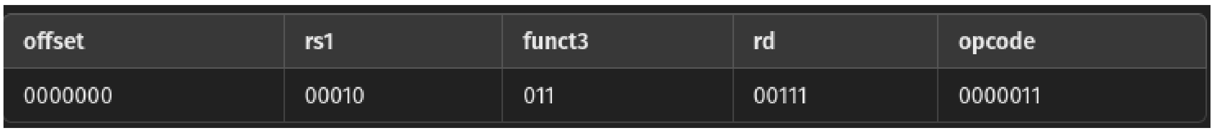
Execution Steps

1. Instruction Fetch:
    - PC fetches the LD instruction.
    - PC increments by 4 (PC = PC + 4).
2. Instruction Decode:
    - The Control Unit decodes the instruction.
    - Identifies it as a Load operation.

    - Sets control signals: MemRead = 1, RegWrite = 1, ALUOp = 00 (address calculation).
3. Register Read:
    - Register file reads value from rs1 (base address).
4. Address Calculation (ALU Execution):
    - ALU computes memory address:
      Memory Address = rs1 + offset
5. Memory Access:
    - Data is fetched from the computed memory address.
6. Write Back:
    - Loaded data is written into rd.

Instruction Breakdown

32-bit Instruction: 00000000000000011011000110000011



offset	rs1	funct3	rd	opcode
0000000	00010	011	00111	0000011

Assembly Instruction:
LD x7, 0(x2)

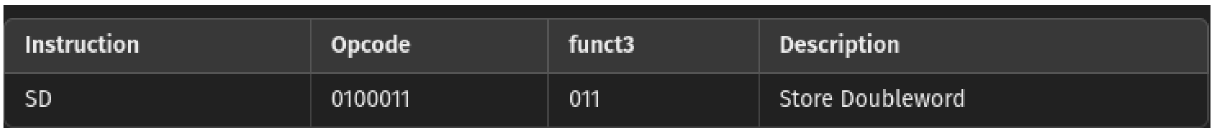
- Loads a 64-bit value from memory address x2 + 0 into x7.

2. SD (Store Doubleword)

Format (S-Type Instruction)
SD rs2, offset(rs1)

- Stores a 64-bit value from register rs2 into memory at address rs1 + offset.

Opcode & Function Codes



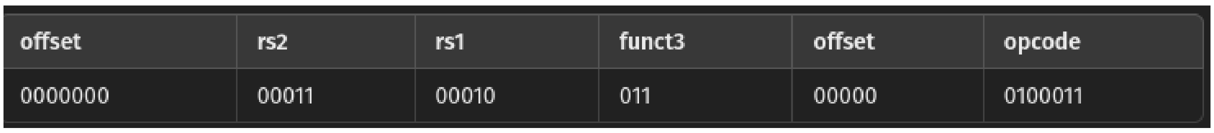
Instruction	Opcode	funct3	Description
SD	0100011	011	Store Doubleword

Execution Steps

1. Instruction Fetch:
   - PC fetches the SD instruction.
   - PC increments by 4 (PC = PC + 4).
2. Instruction Decode:
   - Control Unit decodes the instruction.
   - Identifies it as a Store operation.
   - Sets control signals: MemWrite = 1, ALUOp = 00 (address calculation).
3. Register Read:
   - Register file reads value from rs1 (base address).
   - Register file reads data from rs2 (to be stored).
4. Address Calculation (ALU Execution):
   - ALU computer memory address:
     Memory Address = rs1 + offset
5. Memory Access:
   - Data from rs2 is stored at the computed memory address.

Instruction Breakdown

32-bit Instruction: 00000000001100010011000000100011



offset	rs2	rs1	funct3	offset	opcode
0000000	00011	00010	011	00000	0100011

Assembly Instruction:
SD x3, 0(x2)

- Stores the 64-bit value from x3 into memory at address x2 + 0.

1. `beq x2, x3, 4`

- **Branch if Equal (BEQ)** compares x2 and x3.
- If they are equal, it jumps to the instruction **4 words (16 bytes) ahead**; otherwise, it continues sequentially.

2. `add x1, x2, x3`

- **Addition**: `x1 = x2 + x3`.
- This executes only if the previous `beq` did not branch.

3. **`sub x4, x2, x3`**

- **Subtraction**: `x4 = x2 - x3`.
- Executes normally if `beq` didn't branch.

4. **`and x5, x2, x3`**

- **Bitwise AND**: `x5 = x2 & x3`.
- Executes if `beq` didn’t branch.

5. **`beq x2, x2, 2`**

- Always true (`x2 == x2`), so this instruction **jumps 2 words (8 bytes) forward**.
- Skips the next two instructions (`nop, nop`).

6. **`nop` (No Operation)**

- Does nothing, but is skipped due to the `beq x2, x2, 2`.

7. **`nop`**

- Again, does nothing and is also skipped.

8. **`nop`**

- Third `nop`, also skipped due to the previous `beq`.

9. **`or x6, x2, x3`**

- **Bitwise OR**: `x6 = x2 | x3`.
- Executes after the branch delay.

10. **`ld x7, 0(x2)`**

- **Load Doubleword**: Loads 64-bit value from memory at address `x2` into `x7`.

11. `sd x3, 0(x2)`

- **Store Doubleword**: Stores `x3` into memory at address `x2`.

Execution Summary

- The first `beq` may cause a branch (skipping the next three instructions).
- The second `beq` **always branches**, skipping three `nop`s.
- After the branch, execution continues with `or`, `ld`, and `sd`.
- **Memory operations (`ld` and `sd`) interact with data stored at address `x2`.**

Test Case for Implementation

Pipeline with Hazrads

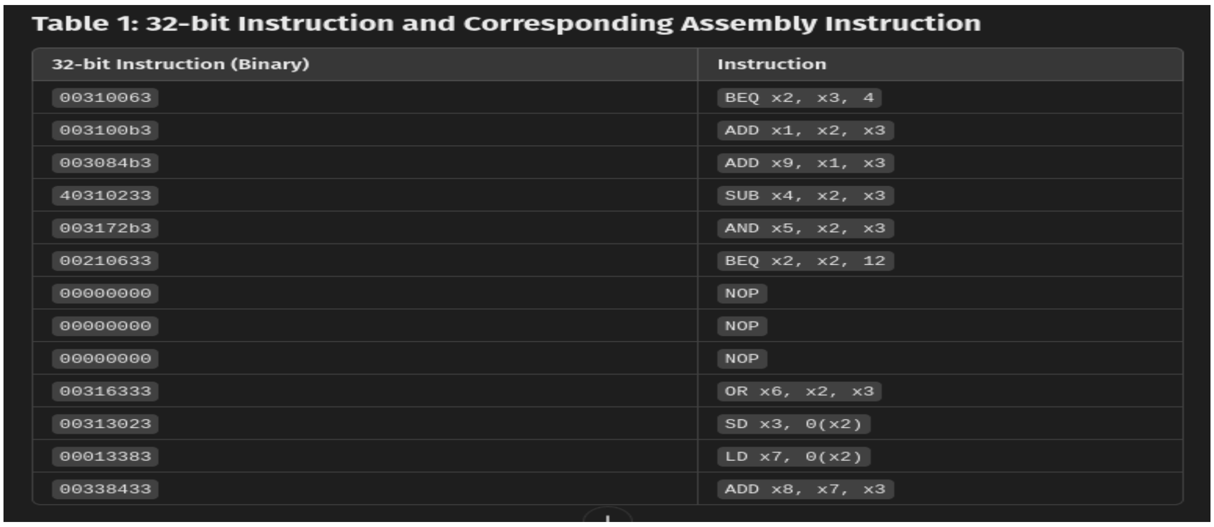


Table 1: 32-bit Instruction and Corresponding Assembly Instruction

32-bit Instruction (Binary)	Instruction
00310063	BEQ x2, x3, 4
003100b3	ADD x1, x2, x3
003084b3	ADD x9, x1, x3
40310233	SUB x4, x2, x3
003172b3	AND x5, x2, x3
00210633	BEQ x2, x2, 12
00000000	NOP
00000000	NOP
00000000	NOP
00316333	OR x6, x2, x3
00313023	SD x3, 0(x2)
00013383	LD x7, 0(x2)
00338433	ADD x8, x7, x3

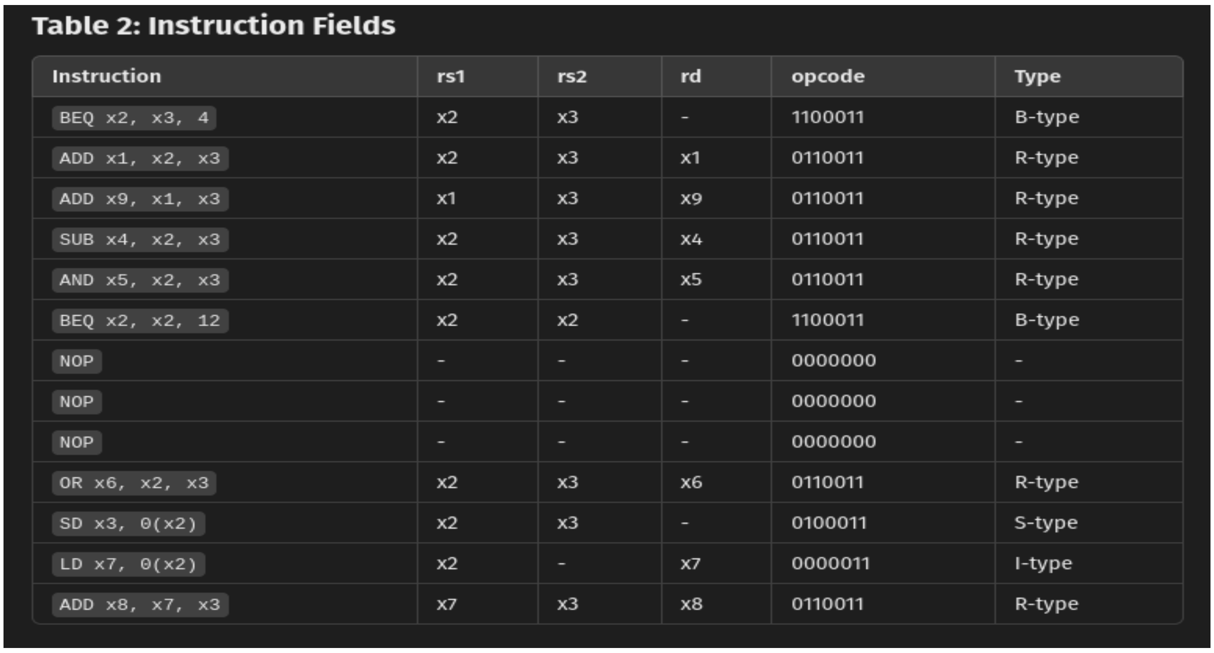
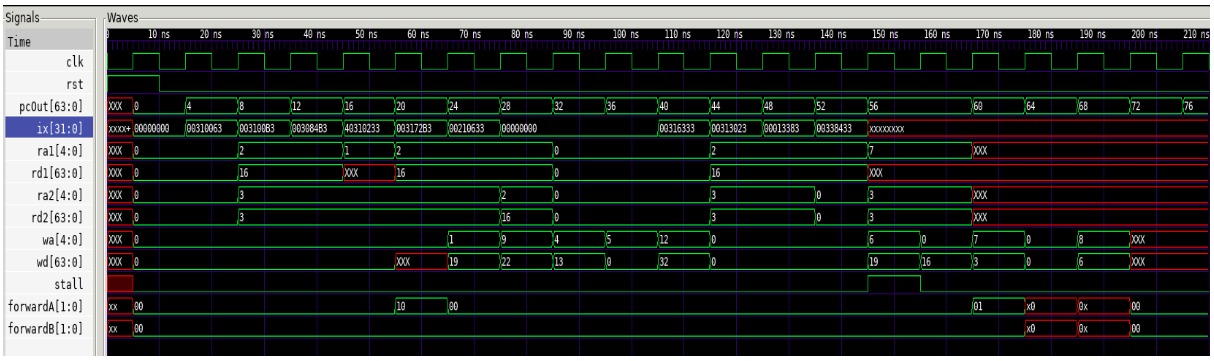


Table 2: Instruction Fields

Instruction	rs1	rs2	rd	opcode	Type
`BEQ x2, x3, 4`	x2	x3	-	1100011	B-type
`ADD x1, x2, x3`	x2	x3	x1	0110011	R-type
`ADD x9, x1, x3`	x1	x3	x9	0110011	R-type
`SUB x4, x2, x3`	x2	x3	x4	0110011	R-type
`AND x5, x2, x3`	x2	x3	x5	0110011	R-type
`BEQ x2, x2, 12`	x2	x2	-	1100011	B-type
`NOP`	-	-	-	0000000	-
`NOP`	-	-	-	0000000	-
`NOP`	-	-	-	0000000	-
`OR x6, x2, x3`	x2	x3	x6	0110011	R-type
`SD x3, 0(x2)`	x2	x3	-	0100011	S-type
`LD x7, 0(x2)`	x2	-	x7	0000011	I-type
`ADD x8, x7, x3`	x7	x3	x8	0110011	R-type



GTK WAVE

Pipeline Execution for Each Instruction

Each instruction follows **five pipeline stages**:

1. **Instruction Fetch (IF)** – The instruction is fetched from memory.
2. **Instruction Decode (ID)** – The instruction is decoded, and registers are read.
3. **Execute (EX)** – The ALU performs computations or calculates memory addresses.
4. **Memory Access (MEM)** – The instruction accesses memory if necessary.
5. **Write Back (WB)** – The result is written back to the register file if required.

1. BEQ x2, x3, 4 (Branch if Equal)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.
In the **ID** stage, the instruction is decoded, and registers x2 and x3 are read.
In the **EX** stage, the ALU compares the values in x2 and x3 and calculates the branch target address.
In the **MEM** stage, there is no memory access; if the branch is taken, the PC is updated to the target address.
In the **WB** stage, there is no write-back as this is a control instruction that only affects PCs.

2. ADD x1, x2, x3 (Addition)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.
In the **ID** stage, the instruction is decoded, and registers x2 and x3 are read.
In the **EX** stage, the ALU performs the addition operation x2 + x3.
In the **MEM** stage, there is no memory access required.
In the **WB** stage, the result is stored in register x1.

3. ADD x9, x1, x3 (Addition)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.
In the **ID** stage, the instruction is decoded, and registers x1 and x3 are read.
In the **EX** stage, the ALU performs the addition operation x1 + x3.
In the **MEM** stage, there is no memory access required.
In the **WB** stage, the result is stored in register x9.

4. SUB x4, x2, x3 (Subtraction)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.
In the **ID** stage, the instruction is decoded, and registers x2 and x3 are read.
In the **EX** stage, the ALU performs the subtraction operation x2 - x3.
In the **MEM** stage, there is no memory access required.
In the **WB** stage, the result is stored in register x4.

5. AND x5, x2, x3 (Bitwise AND)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.
In the **ID** stage, the instruction is decoded, and registers x2 and x3 are read.
In the **EX** stage, the ALU performs a bitwise AND operation between x2 and x3.
In the **MEM** stage, there is no memory access required.
In the **WB** stage, the result is stored in register x5.

6. BEQ x2, x2, 12 (Branch Always)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.
In the **ID** stage, the instruction is decoded, and registers x2 and x2 are read.
In the **EX** stage, since x2 is always equal to x2, the ALU calculates the branch target address.
In the **MEM** stage, if the branch is taken, the PC is updated.
In the **WB** stage, there is no write-back as this is a control instruction that only affects PCs.

7. NOP (No Operation)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.
In the **ID** stage, the instruction is decoded, but no registers are read.
In the **EX** stage, no operation is performed.
In the **MEM** stage, no memory access is required.
In the **WB** stage, no result is stored.

8. OR x6, x2, x3 (Bitwise OR)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.
In the **ID** stage, the instruction is decoded, and registers x2 and x3 are read.
In the **EX** stage, the ALU performs a bitwise OR operation between x2 and x3.
In the **MEM** stage, there is no memory access required.
In the **WB** stage, the result is stored in register x6.

9. SD x3, 0(x2) (Store Doubleword)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.
In the **ID** stage, the instruction is decoded, and registers x2 and x3 are read.
In the **EX** stage, the ALU calculates the memory address by adding the immediate offset (0) to x2.
In the **MEM** stage, the value of x3 is stored in memory at the calculated address.
In the **WB** stage, there is no write-back since this is a store instruction.

10. LD x7, 0(x2) (Load Doubleword)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.

In the **ID** stage, the instruction is decoded, and register x2 is read.
In the **EX** stage, the ALU calculates the memory address by adding the immediate offset (0) to x2.
In the **MEM** stage, the value at the calculated memory address is read.
In the **WB** stage, the loaded value is stored in register x7.

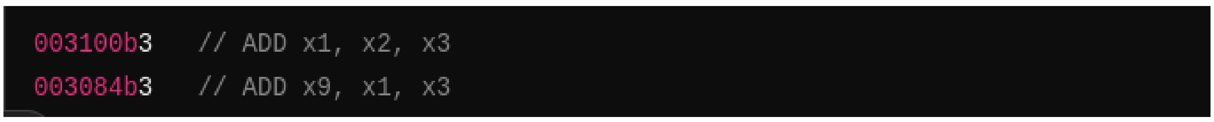
11. ADD x8, x7, x3 (Addition)

In the **IF** stage, the instruction is fetched from memory, and the PC is incremented.
In the **ID** stage, the instruction is decoded, and registers x7 and x3 are read.
In the **EX** stage, the ALU performs the addition operation x7 + x3.
In the **MEM** stage, there is no memory access required.
In the **WB** stage, the result is stored in register x8.

Data Hazard in Our Test Case Instructions

Example 1: Data Hazard

Instructions:



```
003100b3   // ADD x1, x2, x3
003084b3   // ADD x9, x1, x3
```

Issue:

- The second instruction reads x1 before the first instruction writes its updated value (x1 = x2 + x3).

- This creates a **data hazard** because `ADD x9, x1, x3` requires `x1`, but it hasn't been updated yet.

Resolution:

- **Data Forwarding**: The computed value (`x2 + x3`) is forwarded directly from the **EX/MEM register** instead of reading from the register file.

Example 2: Data Hazard

Instructions:

```
003100b3   // ADD x1, x2, x3
003084b3   // ADD x9, x1, x3
40310233   // SUB x4, x2, x3
```

Issue:

- The second instruction uses `x1` before its updated value is written back.
- This is a **data hazard** because `ADD x1, x2, x3` writes to `x1`, but `ADD x9, x1, x3` needs `x1` immediately after.

Resolution:

- **Data Forwarding**: The result from `ADD x1, x2, x3` is forwarded from the **MEM/WB register** to the **EX stage** of `ADD x9, x1, x3`.

Double Data Hazard Example

Instructions:

```
00316333   // OR x6, x2, x3
00313023   // SD x3, 0(x2)
00013383   // LD x7, 0(x2)
00338433   // ADD x8, x7, x3
```

Issue:

- `LD x7, 0(x2)` loads data, but `ADD x8, x7, x3` immediately uses `x7`, creating a **Load-Use Hazard**.
- The value of `x7` is **not yet available** when `ADD x8, x7, x3` is executed.

Resolution:

- **Pipeline Stalling**: Introduce a one-cycle delay (NOP) to allow `LD x7, 0(x2)` to complete.
- **Forwarding**: If possible, forward the value from `LD` as soon as it is available.

Control Hazard Example

Instructions:

```
00310063   // BEQ x2, x3, 4
003100b3   // ADD x1, x2, x3
003084b3   // ADD x9, x1, x3
```

Issue:

- The branch instruction (`BEQ x2, x3, 4`) depends on the **ALU Zero flag**, which is not computed until the **EX stage**.
- The processor might **fetch incorrect instructions** if the branch is taken.

Resolution:

- **Static Branch Prediction**: Assume the branch is **not taken**.
- **Pipeline Flush**: If the branch is taken, the incorrectly fetched instructions are removed.

Load Data Hazard Example

Instructions:

```
00013383   // LD x7, 0(x2)
00338433   // ADD x8, x7, x3
```

Issue:

- `ADD x8, x7, x3` tries to use `x7` before `LD x7, 0(x2)` has written it back to the register.
- This causes a **load data hazard**.

Resolution:

- **Introduce a Stall (NOP)** to wait for `LD x7, 0(x2)` to complete.
- **Forwarding from MEM/WB stage**: If the processor supports forwarding, use the value before writing to the register.

CONTRIBUTION

Saarthak: Data Hazard Detection, forwarding, and pipelining registers,Wiring pf Pipeline registers, Test case preparation for sequential, ALU Code block implementation and module syncing.

Aanchal: Sequential implementation, Wiring of pipelined registers, stalling in pipelining, and testing of pipelined implementation, Decoder Logic Implementation.

Krishna: Sequential implementation, report preparation, and sequential testing, testing of pipelined implementation, test case preparation for Pipelined, Control Unit Implementation.