*Implementation of 5-stage pipelined*

*RISC-V processor (RV32I) in Verilog*

Achutha Aswin Naick

*College of Engineering, Trivandrum*

**ABSTRACT**

**IMPLEMENTATION OF 5-STAGE PIPELINED**

**RISC-V PROCESSOR (RV32I) IN VERILOG**

*This project involves the design and simulation of a RISC-V processor using Verilog HDL. The RISC-V instruction set architecture (ISA) offers a free and open alternative to proprietary ISAs, making it an attractive choice for both hardware and software design in modern computing systems.*

*A 5-stage pipelined architecture—comprising Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB)—was implemented to enhance the processor’s performance through instruction-level parallelism.*

*The processor design includes several key modules:*

* ***Core components:*** *alu, alu\_decoder, main\_decoder, instruction\_memory, data\_memory, reg\_file*
* ***Control and data path support:*** *pc\_mux, wb\_mux, forwarding units (forward\_Amux, forward\_Bmux)*
* ***Pipeline registers:*** *if\_id\_reg, id\_ex\_reg, ex\_mem\_reg, mem\_wb\_reg, if\_id\_reg\_ctrl, id\_ex\_reg\_ctrl, ex\_mem\_reg\_ctrl, mem\_wb\_reg\_ctrl*
* ***Hazard detection unit:*** *To handle data hazards and ensure correct instruction execution*

*Each module was developed, simulated, and functionally verified using* ***AMD Xilinx Vivado 2023.2*** *software. After individual testing, all modules were integrated into a top-level module named “riscv\_integrated”. A simple testbench was written to validate the correct operation of the processor in a pipelined environment.*

*This project demonstrates the full design cycle of a pipelined RISC-V CPU from module-level construction to system-level integration and verification, highlighting its suitability for educational and prototyping purposes.*

**LIST OF ABBREVIATIONS**

|  |  |
| --- | --- |
| adder………………………... | 32-bit Adder |
| alu…………………………... | Arithmetic Logic Unit |
| alu\_decoder………………… | Arithmetic Logic Unit Decoder |
| controller…………………… | Control Unit |
| data\_memory………………. | Data Memory Unit |
| datapath………………...…... | Datapath |
| ex\_mem\_reg………………... | Data Pipeline Register between the Execute (EX) and Memory Access (MEM) stages. |
| ex\_mem\_reg\_ctrl…………... | Control Pipeline Register between the Execute (EX) and Memory Access (MEM) stages. |
| forward\_Amux……………... | Forwarding Unit A |
| forward\_Bmux……………... | Forwarding Unit B |
| hazard\_detection\_unit……… | Hazard Detection Unit |
| id\_ex\_reg…………………... | Data Pipeline Register between the Instruction Decode (ID) and Execute (EX) stages. |
| id\_ex\_reg\_ctrl……………… | Control Pipeline Register between the Instruction Decode (ID) and Execute (EX) stages. |
| if\_id\_reg……………………. | Data Pipeline Register between the Instruction Fetch (IF) and Instruction Decode (ID) stages. |
| immediate\_extend………….. | Immediate Extend |
| instruction\_memory………... | Instruction Memory |
| main\_decoder………………. | Main Decoder |
| mem\_wb\_reg……………….. | Data Pipeline Register between Memory Access (MEM) and Write Back (WB) stages. |
| mem\_wb\_reg\_ctrl…………... | Control Pipeline Register between Memory Access (MEM) and Write Back (WB) stages. |
| mux\_2\_1……………………. | 2 x 1 Multiplexer |
| mux\_3\_1……………………. | 3 x 1 Multiplexer |
| mux\_4\_1……………………. | 4 x 1 Multiplexer |
| pc\_mux……………………... | Program Counter Multplexer |
|  |  |
| reg\_file……………………... | Register File |
| reset\_ff……………………... | Reset Flip-Flop |
| riscv\_integrated……………. | RISC-V Integrated Module |
| riscv\_top…………………… | RISC-V top Module |
| wb\_mux……………………. | Write Back Multiplexer |

**INTRODUCTION**

RISC-V is a free and open Instruction Set Architecture (ISA) rooted in the principles of the Reduced Instruction Set Computing (RISC) architecture. It was developed in 2010 at the University of California, Berkeley, with the vision of creating a simple, extensible, and openly available ISA for academic and industrial use.

One of the key advantages of RISC-V is its open-source licensing model, which eliminates the licensing fees typically associated with proprietary ISAs. This significantly lowers the entry barrier for innovation in the semiconductor industry, fostering rapid development and widespread adoption.

RISC-V is inherently modular and flexible. The base ISA consists of only 47 core instructions, and additional functionalities can be added through standard or custom extensions to suit specific design requirements. This modularity allows designers to tailor their implementations for a wide range of applications—from low-power embedded systems to high-performance computing.

Importantly, the RISC-V specification defines the ISA but does not prescribe the microarchitectural implementation or mandate the inclusion of specific subsets. As a result, various RISC-V-based processors may incorporate different sets of extensions, such as the compressed (C) extension for reducing code size, power consumption, and memory footprint—making RISC-V ideal for resource-constrained systems.

**PURPOSE OF DOCUMENTATION**

This documentation presents a detailed overview of the design, implementation, verification, and potential future improvements of theRISC-V processor. It offers valuable insights into the architectural choices made during development, the functionality of individual modules, and the processor’s overall performance. The document is intended for engineers, developers, educators, and researchers with an interest in RISC architecture, processor design concepts, and hardware implementation strategies.

The objectives of this documentation include:

* Promoting effective knowledge sharing among team members and stakeholders.
* Serving as a reliable reference for future upgrades or modifications.
* Acting as a learning resource for individuals exploring processor architecture and design.

**PROBLEM STATEMENT**

* The Instruction Set Architecture (ISA) forms the core abstraction of a computer system, acting as the interface between hardware and software. It defines essential aspects such as supported data types, registers, memory access mechanisms, executable instructions, and processor features. Major companies like Intel, IBM, and ARM rely on proprietary ISAs for their processors. However, these ISAs are typically closed-source, protected by licensing agreements that restrict use by third parties. Obtaining usage rights involves long negotiations and high costs, making it inaccessible to small-scale organizations, research groups, and the broader community.
* To address this challenge, the adoption of an open and royalty-free ISA becomes crucial. Such an approach fosters innovation and democratizes hardware design, enabling small organizations and academic institutions to participate in processor development. RISC-V, developed at the University of California, Berkeley, in 2010, is a free and open ISA that meets this need. RISC-V’s flexibility and openness make it an ideal candidate for processor design in educational, research, and industrial settings.
* Another critical concern is processor efficiency. Traditional single-cycle processors execute one instruction per clock cycle, leading to performance degradation when handling complex instructions. Moreover, the clock cycle must accommodate the longest instruction, increasing latency. This inefficiency can be significantly mitigated using pipelining. In a pipelined architecture, multiple instructions are processed simultaneously in overlapping stages (fetch, decode, execute, memory access, write back), greatly improving instruction throughput and overall performance.

Therefore, this project focuses on designing and simulating a 5-stage pipelined RISC-V processor using Verilog, leveraging the advantages of both open ISA and pipelined execution to demonstrate a high-performance, customizable processor architecture.

**OBJECTIVE**

1. To implement 5-stage pipelined design of the RISC-V processor in Verilog

2. To verify the functionality of the design by performing testbench and simulation

**OVERVIEW OF RISC ARCHITECTURE**

**RISC (Reduced Instruction Set Computer) Architecture** emphasizes streamlined instruction execution by using a minimal, highly optimized set of operations. Unlike CISC (Complex Instruction Set Computing) architectures, RISC focuses on simplicity and speed, leading to improved performance and reduced design complexity. The key attributes of RISC include:

* **Simple Instruction Set**: By incorporating fewer and simpler instructions, the processor's control unit becomes less complex, enabling quicker decoding and execution.
* **Single-Cycle Execution Time**: Most instructions are designed to execute in a single clock cycle, ensuring predictable behaviour and facilitating straightforward timing analysis and pipeline integration.
* **Load/Store Design**: Memory is accessed only through dedicated instructions like load and store, while all other operations occur between registers. This reduces dependency on slower memory accesses and improves overall execution efficiency.
* **Efficient Pipelining**: RISC is inherently suited for pipelined execution, where instructions are processed in parallel through sequential stages (such as fetch, decode, execute, memory, and writeback). This overlap in processing boosts throughput significantly.
* **Enough Register Availability**: RISC processors commonly include a larger set of general-purpose registers, minimizing the need for frequent memory access and enabling faster computation through efficient data storage and retrieval.

**ADVANTAGES OF RISC ARCHITECTURE**

The benefits of RISC-based designs go well beyond just high-speed execution:

* **Power Efficiency**: Due to their simplified instruction set and streamlined control logic, RISC processors consume less energy per operation. This makes them especially suitable for portable and low-power devices like smartphones, wearables, and tablets.
* **Scalable Architecture**: RISC processors offer a modular framework that can be easily scaled for higher performance or integrated into larger systems-on-chip (SoCs), supporting a broad range of applications from embedded systems to high-performance computing.
* **Simplified Development and Integration**: The reduced complexity of the instruction set and hardware design makes RISC processors easier to implement and optimize in both hardware and software, accelerating development cycles and improving maintainability.

**APPLICATIONS OF RISC ARCHITECTURE**

RISC-based processors have found widespread adoption across multiple fields, thanks to their streamlined design and performance benefits:

* **Embedded Systems**: RISC architectures are a popular choice in embedded devices where efficient power usage and reliable performance are essential, such as in automotive control units, IoT devices, and industrial automation.
* **Mobile Computing**: Energy-efficient RISC designs, particularly those based on the ARM architecture, are prevalent in smartphones and tablets, enabling extended battery life without compromising speed.

**SOME BASIC BRAINSTORMING QUESTIONS**

**Q.** **What is the need for 5-stage pipelining?**

**A.** Traditional single-cycle processor architectures execute each instruction entirely within one clock cycle. While this approach simplifies control logic, it leads to significant inefficiencies. Each instruction must wait until the current instruction completes before it begins execution. Additionally, since different instructions require varying amounts of time to execute, the clock period must be set according to the slowest instruction. This results in wasted clock cycles and reduced overall performance. Thus, pipelining is introduced as an effective architectural technique to enhance processor throughput and efficiency.

**Q.** **What is pipelining?**

**A.** Pipelining is a performance optimization strategy in processor design that divides the execution of an instruction into multiple distinct stages. These stages are executed in a concurrent and overlapped manner, allowing multiple instructions to be processed simultaneously at different phases of execution. This technique significantly improves instruction throughput, as a new instruction can be fetched in every clock cycle, while previous instructions are concurrently processed in later stages. The processor's overall speed increases without reducing the execution time of individual instructions.

**Q. What is present in this 5-Stage Pipeline Architecture?**

**A.** 5-stage pipelining breaks the instruction execution into the following stages:

1. **Instruction Fetch** (IF):

The processor retrieves the instruction from memory based on the current value of the Program Counter (PC). The instruction is loaded into the instruction cache or instruction register, and the PC is incremented to point to the next instruction.

2. **Instruction Decode** (ID):

In this stage, the fetched instruction is decoded to determine the operation type and the source/destination registers. The required operand values are read from the register file. Immediate values are extracted and sign-extended values are also extracted.

3. **Execute** (EX):

The main operations specified by the instruction is performed during this stage. This may include arithmetic and logical operations in the Arithmetic Logic Unit (ALU), effective address computation for memory instructions, or calculation of Branch Target addresses.

4. **Memory Access** (MEM):

If the instruction requires interaction with memory (e.g., load or store), this stage handles reading from or writing to data memory. Instructions such as lw and sw use this stage to access main memory. For instructions that do not involve memory (e.g., add, sub), this stage is effectively bypassed.

5. **Writeback** (WB):

The final stage involves writing the result of the computation or memory operation back to the destination register in the register file. This ensures that subsequent instructions can access updated values.

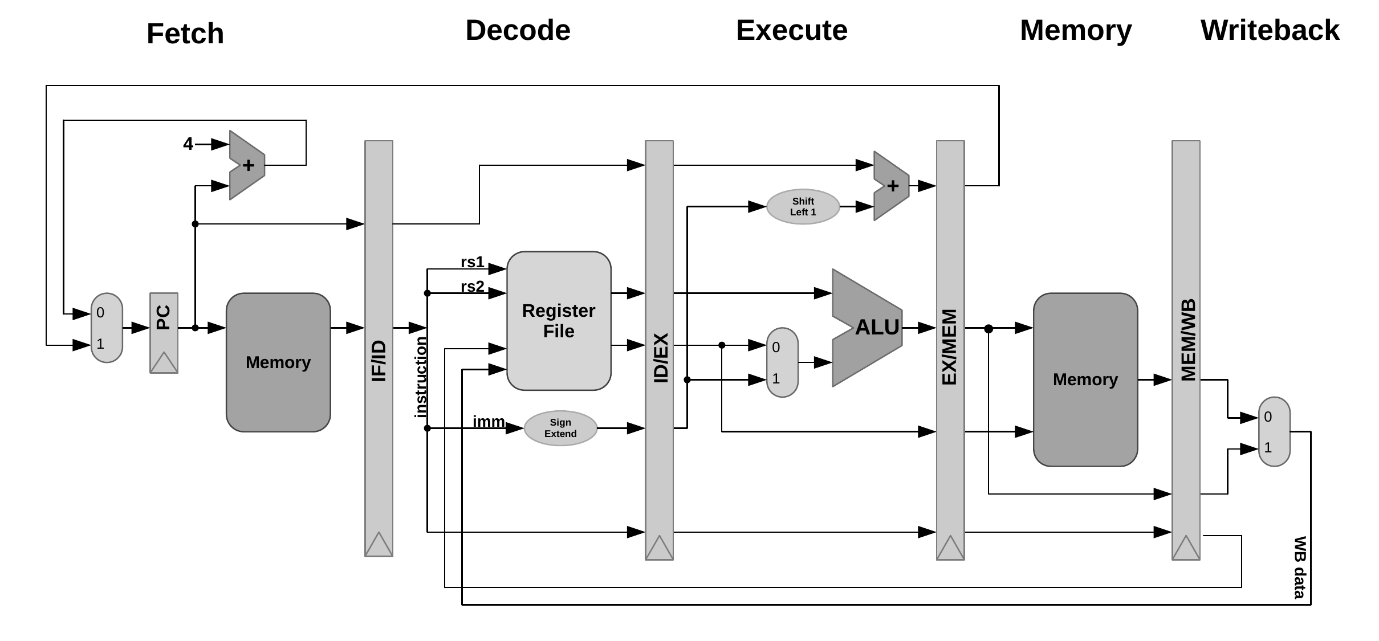


Figure: A Rough Basic 5-stage Pipelining diagram

(source: [*https://www.alrj.org/astorisc-architecture-overview-pipeline.html*](https://www.alrj.org/astorisc-architecture-overview-pipeline.html))

**Software Used: AMD Xilinx Vivado 2023.2**

Vivado is an FPGA design and simulation tool by Xilinx, widely used for implementing hardware designs in Verilog.

For RISC-V processor implementation, Vivado provides features like RTL synthesis, timing analysis, IP integration, and simulation, enabling verification and deployment on FPGA boards such as the Basys3, Zynq 7000 ZedBoard etc.

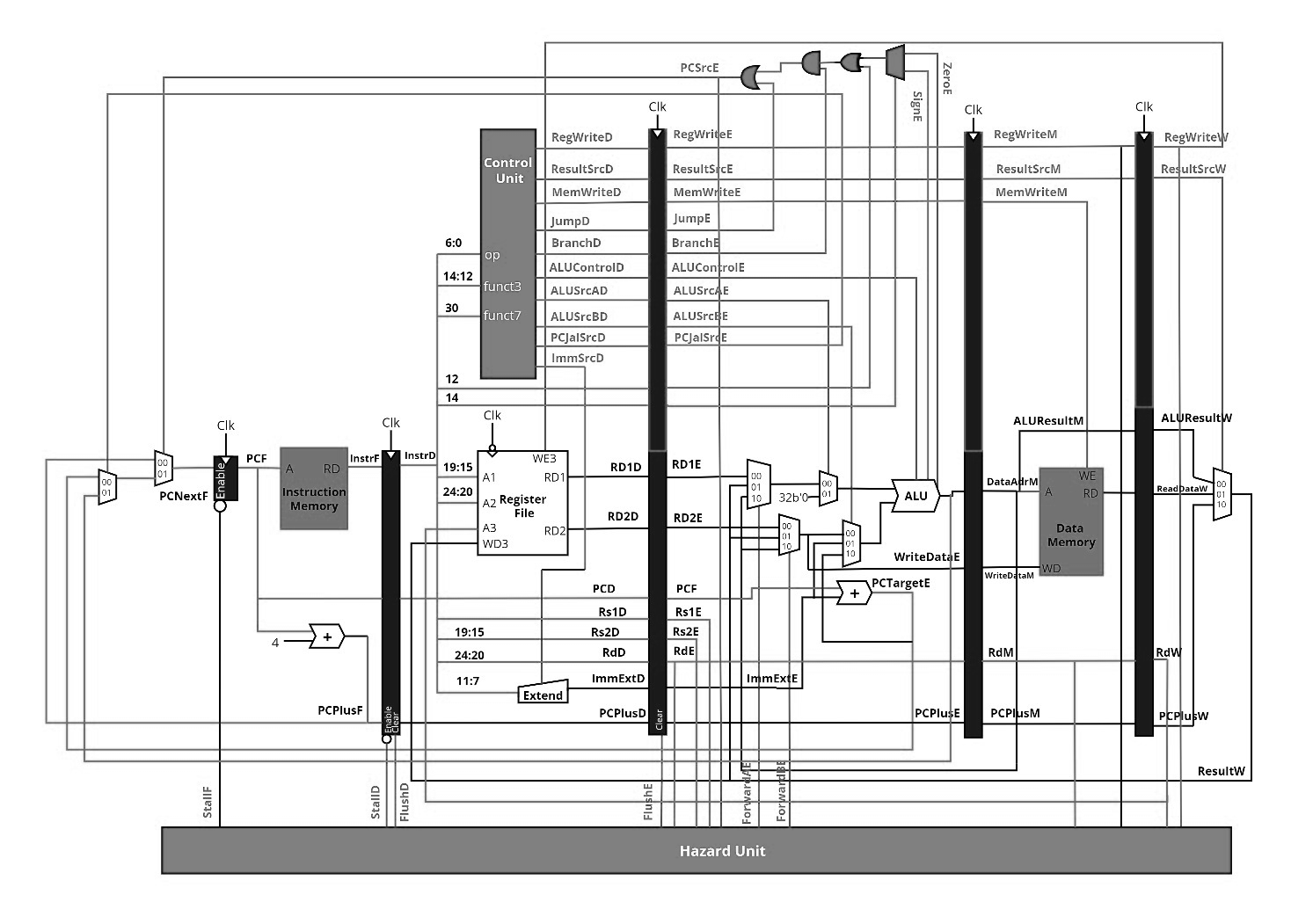
In the project, Vivado will be the main software used to design and simulation the RISC-V processor. Various module such as controller, datapath, hazard unit, pipeline registers, instruction memory, adder, register, data memory, alu and alu decoder etc will be coded in Vivado using Verilog. The functionality for each element will also be tested and verified by simulation using Vivado. Finally, each element will be integrated in a main module to form the top design of a 32-bit 5-stage pipeline RISC-V processor.

**Roadmap for Designing each Module** **to finally Build our RISC-V processor!**

**(my recommendation)**

alu => alu\_decoder => main\_decoder => data\_memory => instruction\_memory => immediate\_extend => if\_id\_reg => id\_ex\_reg => id\_ex\_reg\_ctrl => ex\_mem\_reg => ex\_mem\_reg\_ctrl => mem\_wb\_reg => mem\_wb\_reg\_ctrl => wb\_mux => pc\_mux => reg\_file => forward\_Amux => forward\_Bmux => mux\_2\_1 => mux\_3\_1 => mux\_4\_1 => controller => datapath => hazard\_detection\_unit => riscv\_integrated => riscv\_top

**LET’S START BUILDING OUR RISC-V!**



We will be discussing what the module is responsible for and what are the main highlights of the module.

Additionally, for the first time readers, who are new to Computer Organization & Architecture but have some previous knowledge on Verilog HDL, I have included the majorly raised questions which readers and students face while designing the modules.

These are the same questions which I had asked to myself when I designed my RISC-V Processor for the first time. I am happy to ask questions to myself, because I consider this to be a medium through which I can improve my knowledge and dive deeper into the architecture concepts.

As per my roadmap we will be designing our **ALU (Arithmetic Logic Unit)** first, and will move on.

**A1. ALU (Arithmetic Logic Unit) *(alu.v)***

Arithmetic Logic Unit (alu) is a fundamental component of a CPU. It is responsible for performing arithmetic and logical operations on binary data. It reads the data from pipeline register id\_ex\_reg as an input and perform various arithmetic operation based on the signals from alu\_decoder. The output is stored as ALUResult. In this RISC-V processor design, 10 instructions are implemented in the ALU

The 10 instructions are:

1. Addition

2. Subtraction

3. AND operation

4. OR operation

5. SLL/SLLI (shift left logical/shift left logical immediate)

6. SLT/SLTI (set less than/set less than immediate)

7. XOR

8. SRL

9. SLTU/SLTIU

10. SRA

The ALU unit will have:

Inputs: 32-bit SourceA (first operand value)

32-bit SourceB (second oeprand value)

3-bit ALUControl (to select an ALU Operation)

Outputs: 32-bit ALUResult (result of the ALU operation)

1-bit Zero (Stores 1 if ALUResult =0)

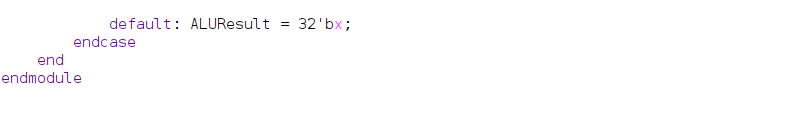
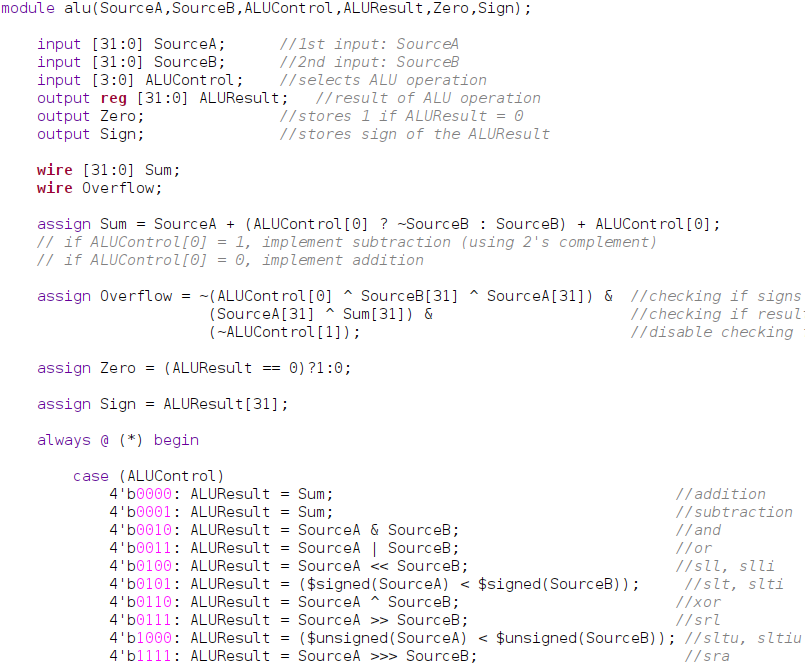
1-bit Sign (Stores the sign of the ALUResult)

wire-types: 32-bit Sum (store the value of either Add/ Sub)

Overflow (checking the sign of operands and ALUResult)

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of alu in Verilog code

**Commonly asked Questions regarding alu module:**

**Q1. Why ALUResult is of reg-type?**

A. ALUResult is decalred as reg-type because it is assigned inside an always block, and signal when assigned inside a procedural block, like always and initial, they should be of reg-type.

**Q2. Explain the Sum formation.**

A. I have designed the add and sub operations such that when ALUControl is 4’b0000, add operation is performed and when ALUControl is 4’b0001, sub operation is performed.

So, when LSB of ALUControl is 1, i.e ALUControl[0] = 1, SourceA will be added to negation of SourceB and 1 (subtraction by 2’s complement). Similarly when LSB of ALUControl is 0, i.e ALUControl[0] = 0, SourceA will simply be added to SourceB and 0 (addition).

**Q3. Explain the Overflow assignment.**

A. ~(ALUControl[0] ^ SourceB[31] ^ SourceA[31]) : this checks whether the sign of operands are same for addition, and opposite for subtraction.

(SourceA[31] ^ Sum[31]): this checks whether the Sum and operandA has the same sign.

(~ALUControl[1]) : since these checking is only required for addition and subtraction operations, all the other operations shall be neglected from calculating Overflow.

We have performed AND operation on these three logics as all the conditions should be satisfied at a time.

**Q4. Why default case has ALUResult = 32’bx?**

A. To avoid synthesis of unintended latches.

**A2. ALU Decoder *(alu\_decoder.v)***

Arithmetic Logic Unit Decoder (alu\_decoder) is a unit inside the control unit and is used to decode the instructions. It received the signal from the Main Decoder Unit (main\_decoder) and determine the type of operation that had to be performed by the alu.

Inputs: 1-bit opcodebit5 (5th bit of the opcode: instruction[6:0])

3-bit funct3 (instruction[14:12])

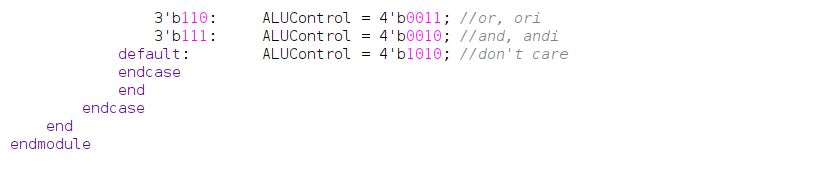
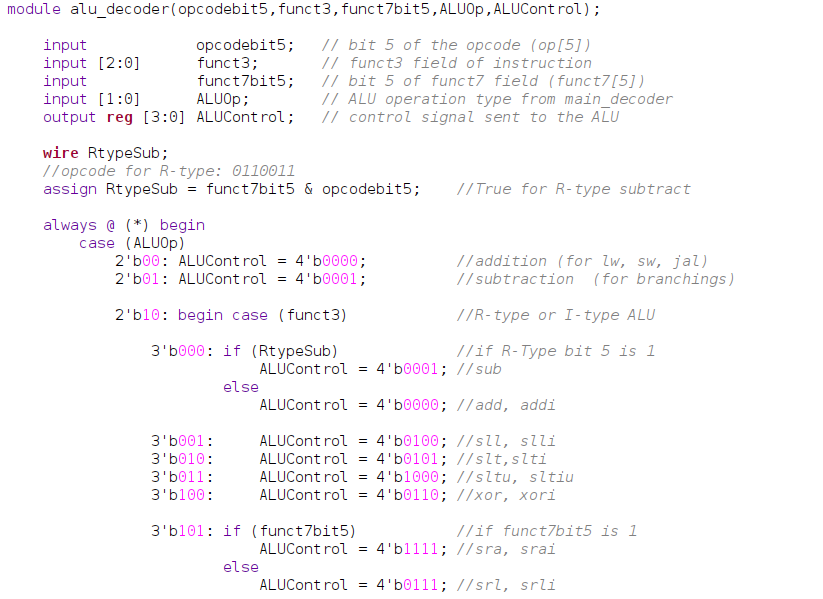
1-bit funct7bit5 (5th bit of the funct7(instruction[31:25]))

2-bit ALUOp (control signal from main\_decoder)

Outputs: 4-bit ALUControl (control signal to ALU)

wire-types: 1-bit RtypeSub (decides whether R-type subtraction should take place or not)

reg types: -

 Implementation of alu\_decoder in Verilog

**Commonly asked Questions regarding alu\_decoder module:**

**Q1. Explain RtypeSub.**

A. RtypeSub is obtained by performing AND operation on funct7bit5 and opcodebit5. This is used to differentiate the R-type ADD and SUB instruction.

\* (opcode for R-type: 0110011)

**Q2. Table for simplified understanding of ALUControl?**

A.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ALUOp | RtypeSub | funct3 | ALUControl | Operation |
| 2’b00 | 0 | - | 4’b0000 | Add  (lw,sw,jal) |
| 2’b01 | 0 | - | 4’b0001 | Sub (branch) |
| 2’b10 | 0 | 3’b000 | 4’b0000 | add/addi |
| 2’b10 | 1 | 3’b000 | 4’b0001 | sub |
| 2’b10 | 0 | 3’b001 | 4’b0100 | sll/slli |
| 2’b10 | 0 | 3’b010 | 4’b0101 | slt/sltu |
| 2’b10 | 0 | 3’b011 | 4’b1000 | slti/sltiu |
| 2’b10 | 0 | 3’b100 | 4’b0110 | xor/xori |
| 2’b10 | 0 | 3’b101 | 4’b1111 | sra/srai |
| 2’b10 | 0 | 3’b101 | 4’b0111 | srl/srli |
| 2’b10 | 0 | 3’b110 | 4’b0011 | or/ori |
| 2’b11 | 0 | 3’b111 | 4’b0010 | and/andi |

**Q3. Why default case has ALUControl = 4’b1010?**

A. I have assigned any 4-bit value that is not used for any ALU operation, so the simulation will simply neglect it.

**Q4. Why SRL and SRA have different logics?**

A. As per the RISC-V referrence card, funct7 of SRL and SRA are different. For SRL, funct7bit 5 is 0, and for SRA, funct7bit5 is 1. Thus, we frame different logics for both.

**A3. Main Decoder** ***(main\_decoder.v)***

Main Decoder (main\_decoder) is a unit present inside the control unit and is used to generate the control signals from the 7 bits opcode (instruction[6:0]) to determine the types of instruction. The control signals are RegWrite, ImmSrc, ALUSrcA, ALUSrcB, MemWrite, ResultSrc, Branch, ALUOp, and Jump. Each of these control signals control the multiplexer for making decisions in the datapath to allow the data flow accordingly to the instructions.

Inputs: 7-bit opcode (opcode: instruction[6:0])

Outputs: 2-bit ALUOp (control signal to alu\_decoder)

1-bit ALUSrcA (select source of operandA (rs1 or imm))

2-bit ALUSrcB (select source of operandA (rs2 or imm or PC+4 or anyother source))

1-bit RegWrite (enables writing to register file (reg\_file))

1-bit MemWrite (enables writing to data memory)

2-bit ResultSrc (selects Result source to register file from ALUResult or Data memory or PC+4)

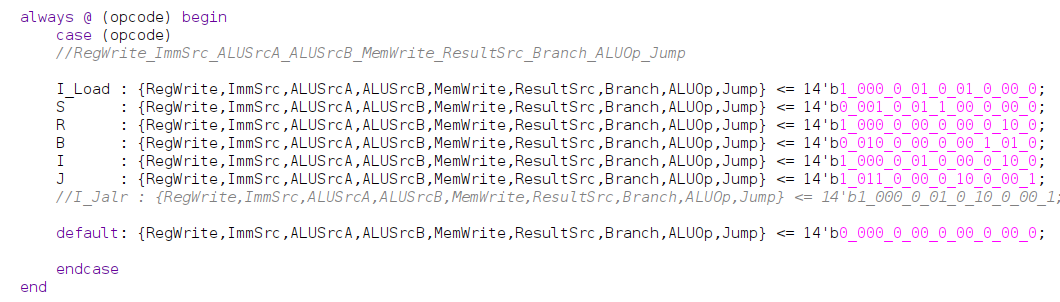
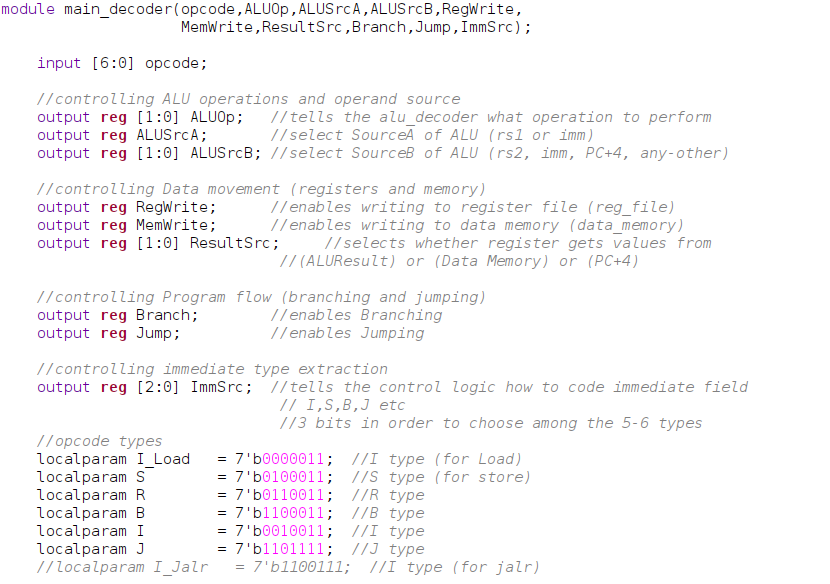
1-bit Branch (trigger branching)

1-bit Jump (trigger jumping)

3-bit ImmSrc (controls the immediate extend module to determine the type of instructions)

wire-types: -

reg types: -

Implementation of main\_decoder in Verilog

**Commonly asked Questions regarding main\_decoder module:**

**Q1. Why such particular signals are used in inputs and outputs?**

1. Since most of the signals are control signals, all the signals are generally the main and important to control different other components present in the architecture.

**Q2. Why is a case statement used on opcode?**

A. The opcode uniquely identifies the instruction type (e.g., R, I, S, B, J types). The case statement maps each opcode to the specific set of control signals needed for that instruction's execution.

**Q3. What is the significance of combining all control signals in a single binary literal like 14'b1\_000\_0\_01\_0\_01\_0\_00\_0?**

A. It groups related control signals into one assignment to simplify code and make each instruction's behaviour easy to visualize. The bit slicing corresponds to the control signal vector:

*RegWrite, ImmSrc[2:0], ALUSrcA, ALUSrcB[1:0], MemWrite, ResultSrc[1:0], Branch, ALUOp[1:0], Jump*

**Q4. What does the ImmSrc signal do? Why is it 3 bits?**

A . ImmSrc selects how to extract and sign/zero extend the immediate value from the instruction based on instruction format (I, S, B, J types).

**Q5. What does ALUSrcA control?**

A. ALUSrcA selects the first input to the ALU:

* 0 → register value rs1
* 1 → PC (used for auipc, branches, etc.)

**Q6.** **What does ALUSrcB control?**  
A. ALUSrcB is a 2-bit signal that selects the second ALU operand:

* 00 → rs2 (register value)
* 01 → Immediate value
* 10 → PC + 4 (e.g., for jal)
* 11 → Reserved/future use

**Q7. What is the purpose of ResultSrc?**

A. ResultSrc determines what value is written back to the register file:

* 00 → ALU result
* 01 → Data read from memory
* 10 → PC + 4 (for jal, jalr)
* 11 → Reserved/future use (NOT USED HERE)

**Q8.** **Why is Jump separated from Branch?**  
A. Because jump instructions (e.g., jal, jalr) always update the PC, while branch instructions conditionally update the PC based on ALU flags. Keeping them separate allows better control logic.

**Q9. How does this decoder help in pipeline control?**

A It provides control signals in the Decode (ID) stage, which are passed to subsequent pipeline stages through pipeline registers (like id\_ex\_reg\_ctrl) (discussed later) to ensure correct timing and usage.

**A4. Data Memory** ***(data\_memory.v)***

In computer architecture, data memory is a component of a computer system that is responsible for storing and retrieving data. Data memory is typically used to store data that is actively being processed by alu. Usually there are 3 inputs in this module, which are write\_enable, data\_address, and write\_data. The module takes the memory address from the results of alu (data\_address) and data from register file (write\_data). Write enable (write\_enable) is used to control the write permission of the data to the data memory.

Inputs: clock (input clock signal)

1-bit write\_enable (to control whether data is to be written)

32-bit data\_address (specifies the memory address to read from or write to)

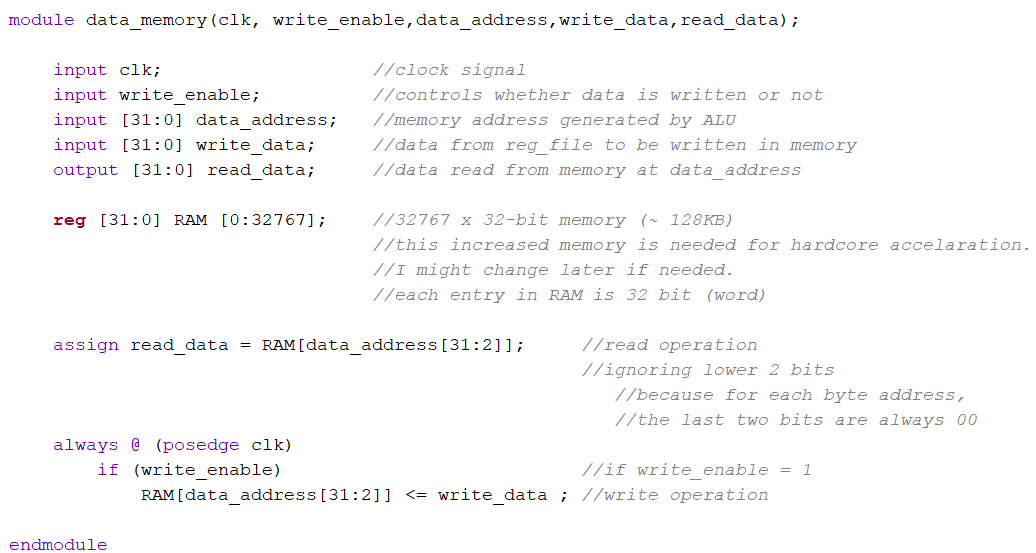
32-bit write\_data (data to be written in memory from register file)

Outputs: 32-bit read\_data (read data from memory)

wire-types: -

reg types: RAM (128KB: needed for my case, can be changed according to use)

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of data\_memory in Verilog

**Commonly asked Questions regarding data\_memory module:**

**Q1. Why is data\_address[31:2] used instead of the full 32-bit address?**

A. Since memory is word-addressed (32-bit/4-byte), the last 2 bits of a valid aligned address are always 0 (e.g., 0x00000008 ends in 00).

* Ignoring data\_address[1:0] ensures access to aligned 4-byte words.
* data\_address[31:2] gives the word index into the RAM array.

**Q2. Is this a synchronous or asynchronous memory?**

A Writes are synchronous: triggered on the positive edge of the clock. Reads are asynchronous: read happens immediately via the assign statement.

**Q3.** **Why is read done asynchronously and write synchronously?**

A.Asynchronous reads are faster and common in simple designs where timing isn't critical. Synchronous writes ensure data stability and predictability during write operations.

**Q4. Why is RAM declared as reg [31:0] RAM [0:32767]; and not wire?**

A. reg is needed because memory must hold state. In Verilog, wire is only used for combinational connections; not for storage.

**A5. Instruction Memory** ***(instruction\_memory.v)***

In computer architecture, instruction memory is a component of a computer system that stores the instruction of a program. It is responsible for holding the sequence of instruction that the CPU fetches, decodes, and executes during the program execution. In this module, 32-bit instruction set is generated and stored in the RAM array. The instruction to be fetch is based on the program counter fetch (PCF) input. As each instruction is 4 bytes, the value of PCF will be incremented by 4 to fetch the next instruction.

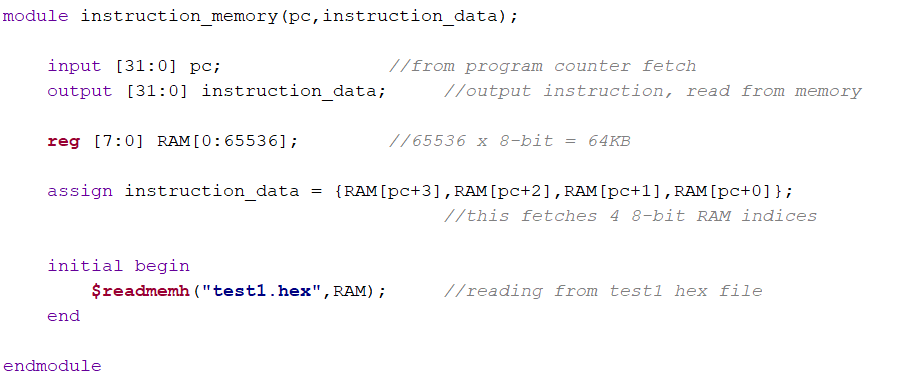
Inputs: 32-bit pc (program counter value from PCF)

Outputs: 32-bit instruction\_data (read instruction data from memory)

wire-types: -

reg types: RAM (64KB: needed for my case, can be changed according to use)

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of Instruction Memory in Verilog

**Commonly asked Questions regarding data\_memory module:**

**Q1. Why is RAM defined as 8-bit wide?**

A Because memory is byte-addressable, and RISC-V instructions are composed of 4 bytes (32 bits). To fetch a full instruction, we combine 4 sequential 8-bit memory locations

**Q2. Why is instruction assembled as {RAM[pc+3], RAM[pc+2], RAM[pc+1], RAM[pc+0]}?**

A. This is to follow little-endian format, where:

* LSB (least significant byte) is stored at the lowest address,
* MSB (most significant byte) is stored at the highest address.

This is standard for RISC-V instruction encoding.

**Q3.** **Why no clock (clk) or write\_enable?**

A. Because:

* This is read-only memory (ROM-like) for instructions only.
* No write operations are performed during execution.
* Only initialized once at startup.

**A6. Immediate Extend** ***(immediate\_extend.v)***

The immediate\_extend module plays a crucial role in a RISC-V processor by generating properly sign-extended immediate values based on instruction encoding formats. Different RISC-V instruction types—such as I-type, S-type, B-type, and J-type—encode their immediate fields differently across various bit positions so that it can be used in arithmetic, memory, and control operations. This module takes the relevant bits from the instruction (instr[31:7]) and a 3-bit imm\_src control signal that specifies the instruction type.

Inputs: 25-bit instr (instruction)

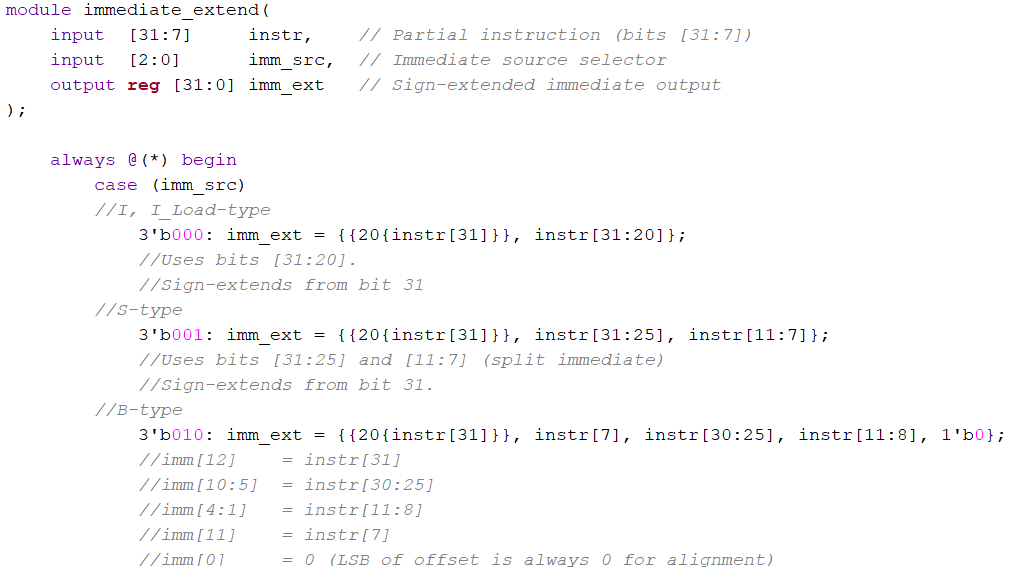
3-bit imm\_src (immediate type source selector)

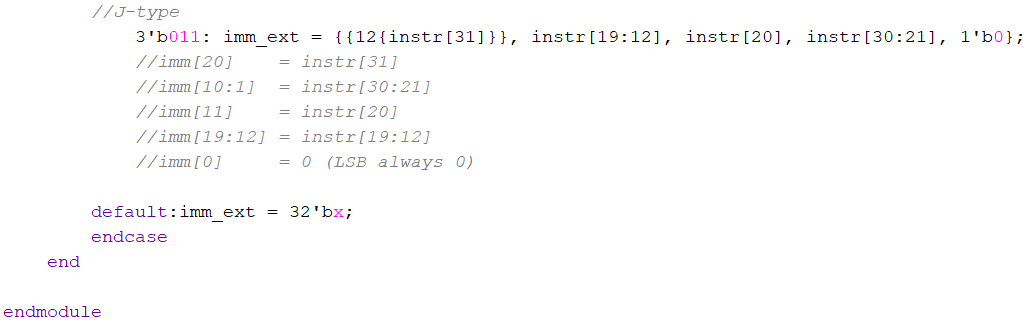
Outputs: 32-bit imm\_ext (immediate output with sign extension)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



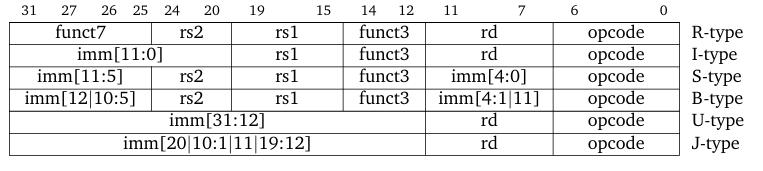


Implementation of immediate\_extend in Verilog

**Commonly asked Questions regarding immediate\_extend module:**

**Q1. Table for instruction formats**

A.

****

**Q2. Why is the input instr only from bits [31:7] and not the full [31:0]?**

A. The lower 7 bits ([6:0]) represent the opcode, which is not part of the immediate value. Only [31:7] contains the bits needed to form the immediate field in various instruction types.

**Q3. What does the imm\_src signal do?**

A. imm\_src is a 3-bit control signal from the controller that tells the module which type of instruction is being decoded (I, S, B, J), so the immediate bits can be properly extracted and sign-extended.

**Q4. How is the I-type immediate extended?**

A. This uses bits [31:20] and extends the sign bit (instr[31]) to fill the upper 20 bits.

**Q5. Why are S-type immediate split across two fields?**

A.In S-type instructions (sw, etc.), the immediate is stored in bits:

* [31:25] (upper)
* [11:7] (lower)

**Q6. How is immediate extend found the B-type immediate?**

A. B-type (branch) immediate are spread across non-contiguous fields and include a zero at the LSB for alignment, The LSB is set to 0 because branch offsets are always even

**Q7. How is immediate extend found the J-type immediate?**

A RISC-V J-type encodes the 20-bit immediate in a dispersed format to optimize instruction encoding.

**PIPELINE REGISTERS**

In computer architecture, pipeline registers are a temporary storage element used in a processor’s pipeline to hold data between different stages of instruction execution process. It serves as a synchronization point between adjacent stages, allowing instruction to flow through the pipeline in a controlled manner. In the five-stage pipelined RISC-V processor, four primary pipeline registers—if\_id\_reg, id\_ex\_reg, ex\_mem\_reg, and mem\_wb\_reg—are implemented to handle the sequential flow of data through the datapath. To maintain clarity and modularity in signal management, control signals are handled separately using dedicated controller pipeline registers: id\_ex\_reg\_ctrl, ex\_mem\_reg\_ctrl, and mem\_wb\_reg\_ctrl. This separation facilitates better organization, simplifies debugging, and enhances the scalability of the design. The registers are named for the two stages separated by that register. For example, the first pipeline register is if\_id\_reg because it separates the instruction fetch and instruction decode stages.

**A7. Instruction Fetch / Instruction Decode Register (datapath)**  ***(if\_id\_reg.v)***

if\_id\_reg register as it names called, it separates the instruction fetch and instruction decode stages. It used to store data such as instruction fetch from instruction memory and ready to be released to decode stage on the next clock cycle. Besides, the current PC and next incremented PC address (PCplus4F) is also saved in the if\_id\_reg register in case it needed later for an instruction, such as beq.

Inputs: 1-bit clk (input clock)

1-bit reset (active high reset)

1-bit clear (remove misplace conditions)

1-bit enable (to enable the register for functioning)

32-bit InstrF (instruction fetched in Fetch stage)

32-bit PCF (Program Counter value in Fetch Stage)

32-bit PCplus4F (Program Counter +4 value in Fetch Stage)

Outputs:32-bit InstrD (instruction fetched in Decode stage)

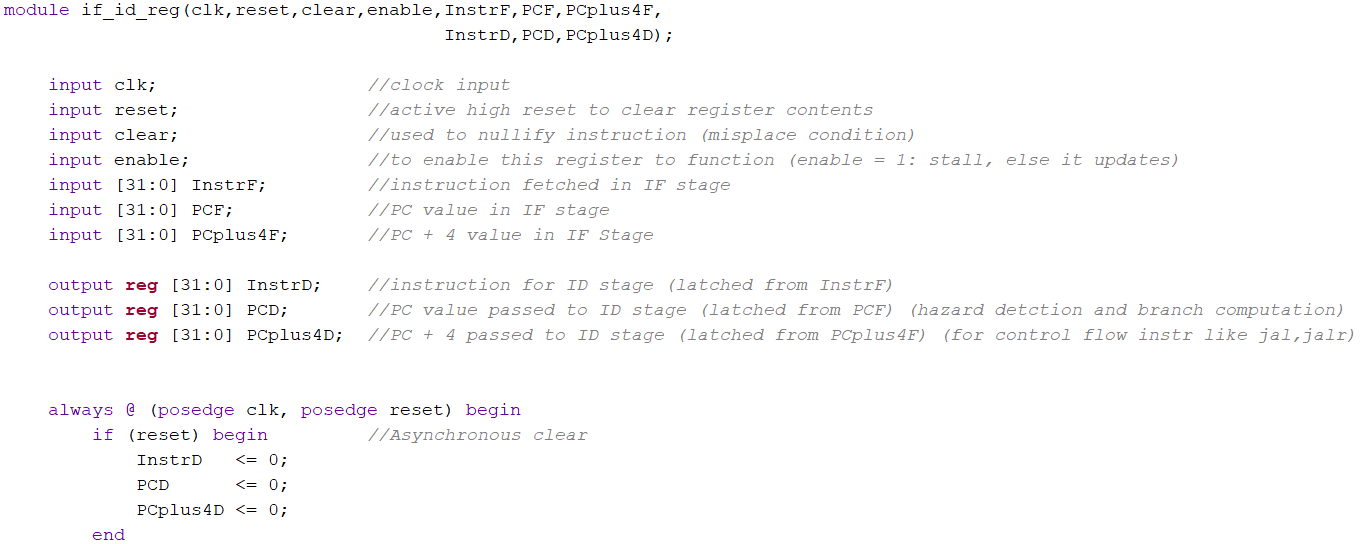
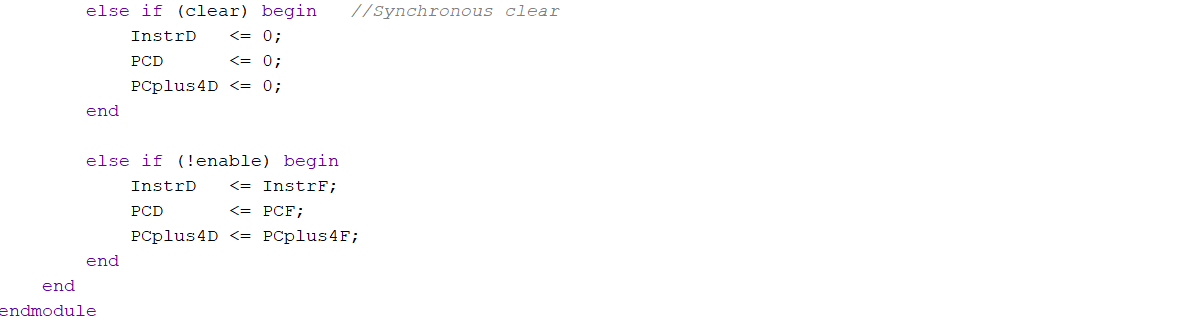
32-bit PCD (Program Counter value in Decode Stage)

32-bit PCplus4D (Program Counter +4 value in Decode Stage)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.

Implementation of if\_id\_reg in Verilog

**Commonly asked Questions regarding if\_id\_reg module:**

**Q1. Why do we need the enable signal in if\_id\_reg?**

A The enable signal is used to control stalling. When enable is high, the register does not update, effectively stalling the pipeline stage. When enable is low, it allows the new instruction and PC values to be latched. This is essential for hazard handling.

**Q2.** **What does the clear signal do in this module?**

A The clear signal synchronously resets the contents of the register to zero on the next clock edge. It is typically used to insert a bubble (NOP) into the pipeline—for instance, after a branch is taken or when there's a misprediction.

**Q3. Why is the reset signal handled asynchronously?**

A The reset signal clears the register contents immediately, regardless of the clock, ensuring that the pipeline starts in a known state upon reset. It is usually used during system startup or hard reset conditions.

**Q4.** **What happens if both reset and clear are asserted simultaneously?**

A Since reset is asynchronous and has higher priority in the always block, it will take precedence and zero out all outputs immediately.

**A8. Instruction Decode / Instruction Execute Register (datapath)**  ***(id\_ex\_reg.v)***

id\_ex\_reg register as it names called, it separates the instruction decode and execute stages. It used to store information such as read data (RD1\_D, RD2\_D) from the register file and extended immediate value (immediate\_extend\_D). Besides, it carries forward the data of PC and PCplus4F from if\_id\_reg pipeline register. Instruction[11:7] (rd\_D), Instruction[19:15] (rs1\_D), and Instruction[24:20] (rs2\_D) will also be stored to id\_ex\_reg register and send to Hazard Unit in execute stage for hazard handling.

Inputs: 1-bit clk (input clock)

1-bit reset (active high reset)

1-bit clear (remove misplace conditions)

32-bit RD1\_D, RD2\_D (operands from reg\_file)

32-bit PCD (Program Counter value in Decode Stage)

32-bit PCplus4D (Program Counter +4 value in Decode Stage)

32-bit immediate\_extend\_D (immediate\_extend in Decode)

5-bit rs1\_D, rs2\_D (source registers index value in Decde)

5-bit rd\_D (destination register index value in Decode)

Outputs: 32-bit RD1\_E, RD2\_E (operands from reg\_file)

32-bit PCE (Program Counter value in Execute Stage)

32-bit PCplus4E (Program Counter +4 value in Execute Stage)

32-bit immediate\_extend\_E (immediate\_extend in Execute)

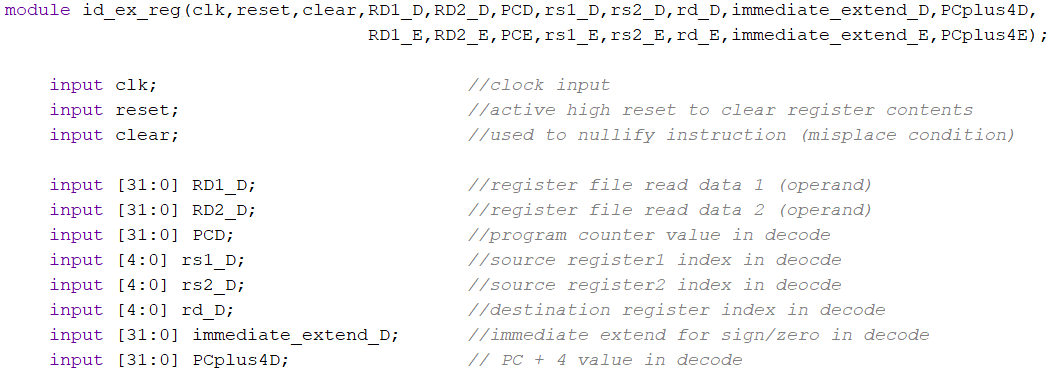
5-bit rs1\_E, rs2\_E (source registers index value in Execute)

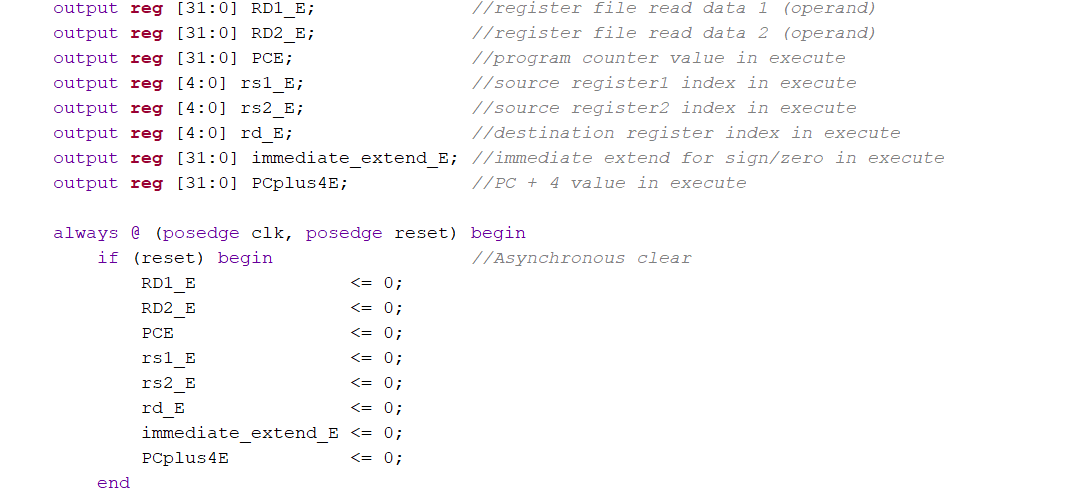
5-bit rd\_E (destination register index value in Execute)

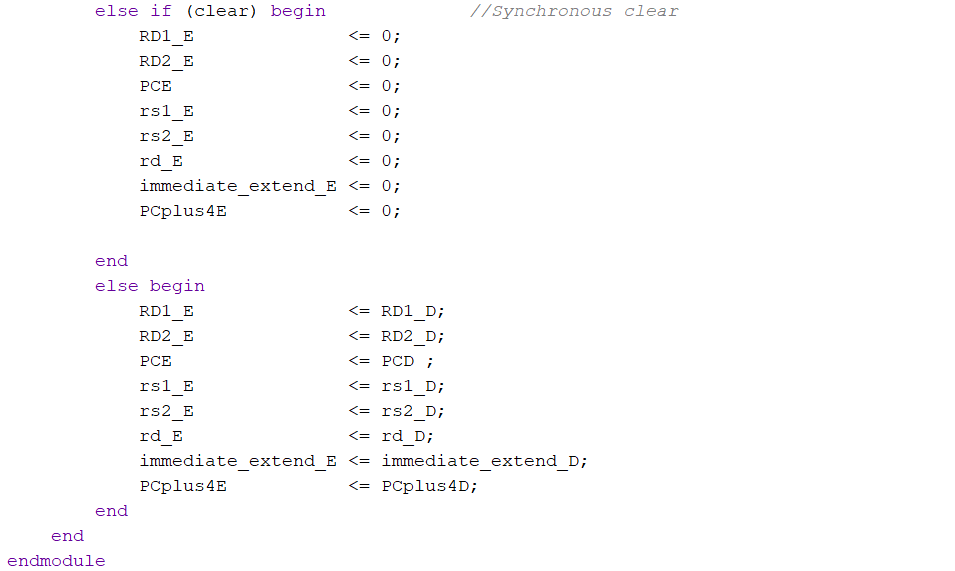
wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.







Implementation of id\_ex\_reg (datapath) in Verilog

**Commonly asked Questions regarding id\_ex\_reg module:**

**Q1.** **Why are both reset and clear used, and what's the difference?**

A Reset is an asynchronous signal that clears all registers immediately when activated, regardless of the clock. Clear is a synchronous control signal used to flush this stage (insert NOP) during control hazard scenarios such as mis predicted branches.

**Q2. Why are register indices (rs1\_D, rs2\_D, rd\_D) passed to the EX stage?**

A These indices are critical for forwarding logic and hazard detection. They help the processor determine if a future instruction depends on the result of a previous one.

**A9. Instruction Execute / Memory Access Register (datapath)**  ***(ex\_mem\_reg.v)***

ex\_mem register as it names called, it separates the execute and memory stages. It is used to store the ALU results (ALUResult) and write data (write\_data). At the same time, Instruction[11:7] (rd) and PCplus4F are also carried forward from previous pipeline registers and stored in ex\_mem register.

Inputs: 1-bit clk (input clock)

1-bit reset (active high reset)

32-bit ALUResult\_E (result obtained from ALU)

32-bit write\_data\_E (data written from reg\_file in Execute)

32-bit PCplus4E (Program Counter +4 value in Execute)

5-bit rd\_E (destination register index value in Execute)

Outputs: 32-bit ALUResult\_M (result obtained from ALU)

32-bit write\_data\_M (data written from reg\_file in Memory Access)

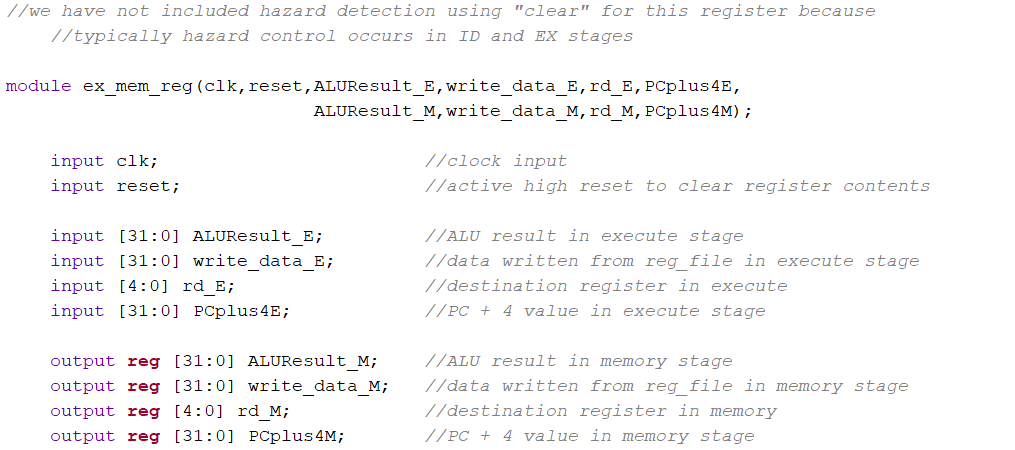
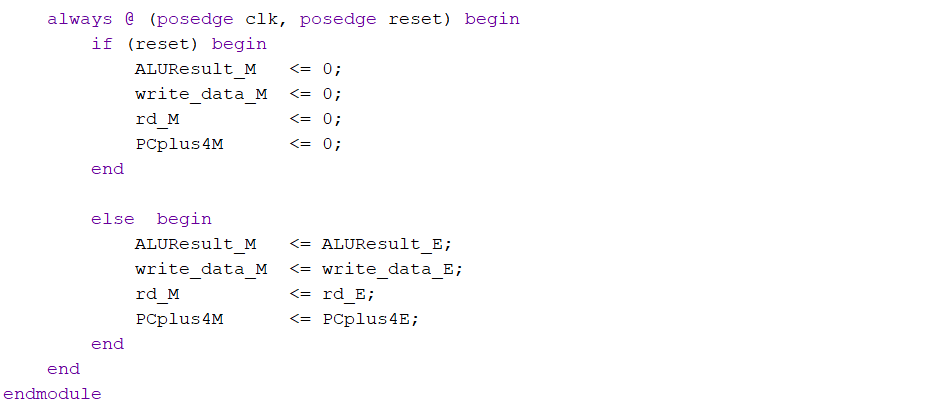
32-bit PCplus4M (Program Counter +4 value in Memory Access)

5-bit rd\_M (destination register index value in Memory Access)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.

Implementation of ex\_mem\_reg (datapath) in Verilog

**Commonly asked Questions regarding ex\_mem\_reg** **module:**

**Q1. Why is hazard detection not included in this register?**

A Hazard detection typically occurs in the ID and EX stages. Once the instruction reaches the ex\_mem register, it's assumed safe to proceed. Therefore, this register doesn't include flush or stall logic. That logic is applied earlier in the pipeline.

**A10. Memory Access / Write Back Register (datapath)**  ***(mem\_wb\_reg.v)***

mem\_wb\_reg register as it names called, it separates the memory and writeback stages. It used to store ALU results (ALUResult) and read data (read\_data) from data memory. Instruction[11:7] (rd) and PCplus4F are also carried forward from previous pipeline registers and store in mem\_wb register.

Inputs: 1-bit clk (input clock)

1-bit reset (active high reset)

32-bit ALUResult\_M (result obtained from ALU)

32-bit read\_data\_M (data read from memory in Memory Access)

32-bit PCplus4M (Program Counter +4 value in Memory Access)

5-bit rd\_M (destination register index value in Memory Access)

Outputs: 32-bit ALUResult\_W (result obtained from ALU)

32-bit read\_data\_W (data read from memory in Write Back)

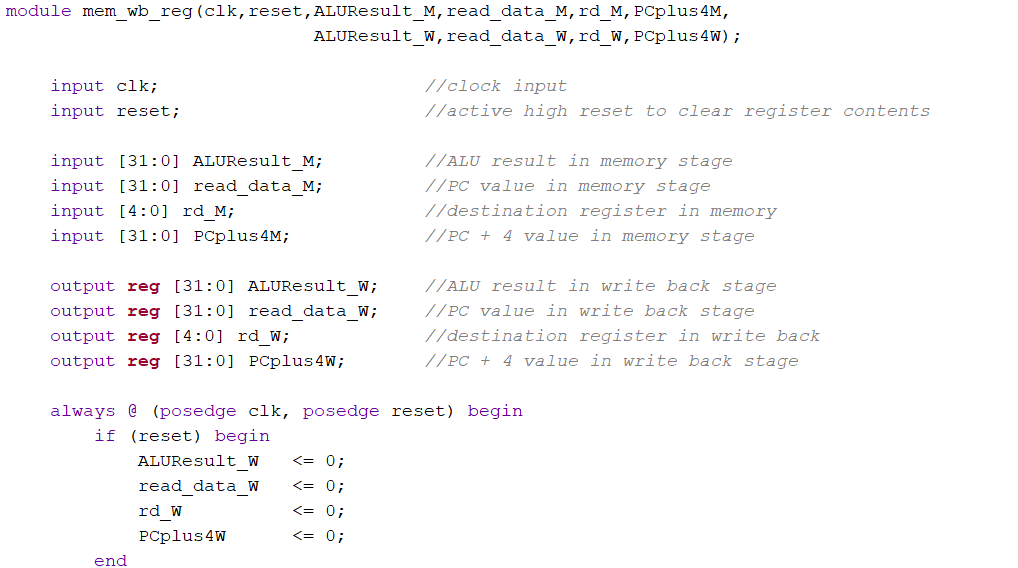
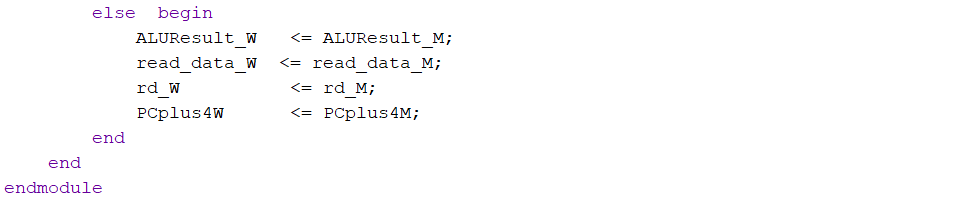
32-bit PCplus4W (Program Counter +4 value in Write Back)

5-bit rd\_W (destination register index value in Write Back)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.

Implementation of mem\_wb\_reg (datapath) in Verilog

**Commonly asked Questions regarding mem\_wb\_reg** **module:**

**Q1. Why is read\_data\_M included here?**

A This signal carries the output of a memory read operation (e.g., lw). It needs to be available in the WB stage so the value can be written back into the register file.

\*\*\**I did not create separate module for Instruction Fetch/ Instruction Decode pipeline register for Control Signals because it stores only instruction and PC+4, which are purely data neutral values. There are no decoded control signals at this point yet because, decoding happens in the ID stage, right after this register.*

**A11. Instruction Decode / Instruction Execute Register (controller)*(id\_ex\_reg\_ctrl.v)***

The id\_ex\_reg\_ctrl module is a pipeline control register that separates the decode stage from the execute stage in a pipelined RISC-V processor.

id\_ex\_reg\_ctrl specifically handles the control signals generated during the decode stage, and passes them forward into the execute stage.

Inputs: 1-bit clk (Clock input)

1-bit reset (clears all control signals)

1-bit clear (used to clear control signals during misprediction or hazard)

1-bit RegWriteD (Register file write enable from Decode stage)

1-bit MemWriteD (Memory write enable from Decode stage)

1-bit JumpD (Indicates jump instruction in Decode stage)

1-bit BranchD (Indicates branch instruction in Decode stage)

1-bit ALUSrcAD (ALU operand A source select (e.g., PC or register))

2-bit ALUSrcBD (ALU operand B source select (e.g., immediate, register, constant))

2-bit ResultSrcD (Mux selector for the result source (e.g., ALU result, memory, PC+4))

4-bit ALUControlD (ALU operation control signal generated in decode)

Outputs1-bit RegWriteE (Register file write enable from Execute stage)

1-bit MemWriteE (Memory write enable from Execute stage)

1-bit JumpE (Indicates jump instruction in Execute stage)

1-bit BranchE (Indicates branch instruction in Execute stage)

1-bit ALUSrcAE (ALU operand A source select (e.g., PC or register))

2-bit ALUSrcBE (ALU operand B source select (e.g., immediate, register, constant))

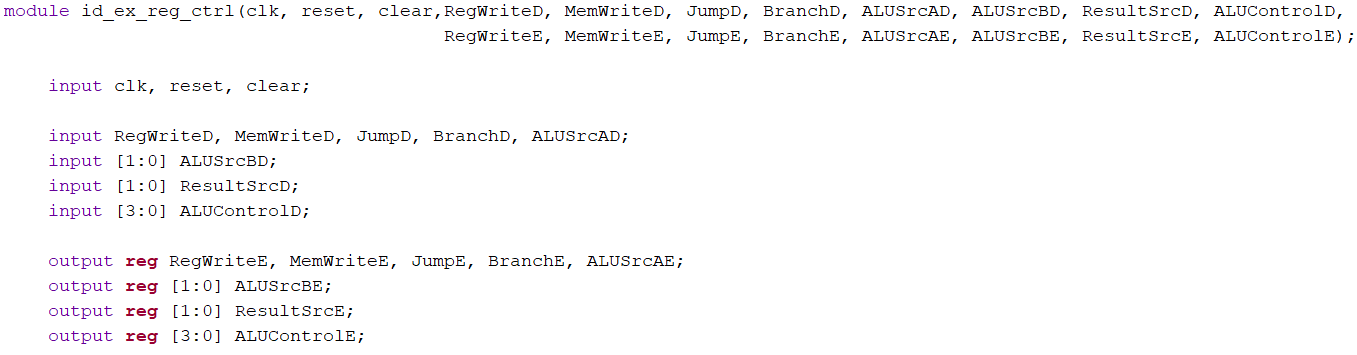
2-bit ResultSrcE (Mux selector for the result source (e.g., ALU result, memory, PC+4))

4-bit ALUControlE (ALU operation control signal generated in execute)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of id\_ex\_reg\_ctrl (controller) in Verilog

**Commonly asked Questions regarding id\_ex\_reg\_ctrl module:**

**Q1.What happens when the clear signal is asserted during execution?**

A When clear is asserted, all output control signals from this module (RegWriteE, MemWriteE, ALUControlE, etc.) are set to zero. This effectively invalidates the instruction in the EX stage, preventing erroneous operations.

**Q2.** **How does this module help in hazard handling or control?**

**A** The module allows the control unit or hazard detection unit to flush the pipeline selectively using the clear signal. This prevents incorrect operations due to hazards by disabling control signals in the EX stage when needed.

**Q3.** **What will be the output of the module if both reset and clear are asserted?**

A Since the reset condition is checked before clear, the module will prioritize reset. Thus, all outputs will be set to zero due to reset.

**Q4. Why is id\_ex\_reg\_ctrl separated from id\_ex\_reg (which handles data signals)?**

A. id\_ex\_reg handles data path signals like register values (RD1, RD2), immediate, PC, etc. id\_ex\_reg\_ctrl handles control signals like RegWrite, MemWrite, ALUSrc, Branch, etc. This improves debugging of the entire processor and reduced critical path delay.

**A12. Instruction Execute /Memory Access Register (controller)*(ex\_mem\_reg\_ctrl.v)***

The ex\_mem\_reg\_ctrl module is a pipeline control register that separates the execute stage from the memory stage in a pipelined RISC-V processor.

ex\_mem\_reg\_ctrl specifically handles the control signals generated during the execute stage, and passes them forward into the memory stage.

Inputs: 1-bit clk (Clock input)

1-bit reset (clears all control signals)

1-bit RegWriteE (Register file write enable from Execute)

1-bit MemWriteE (Memory write enable from Execute)

2-bit ResultSrcE (Selects the source of the result (e.g., ALU result, memory output, PC+4))

Outputs: 1-bit RegWriteM (Register file write enable from Memory Access)

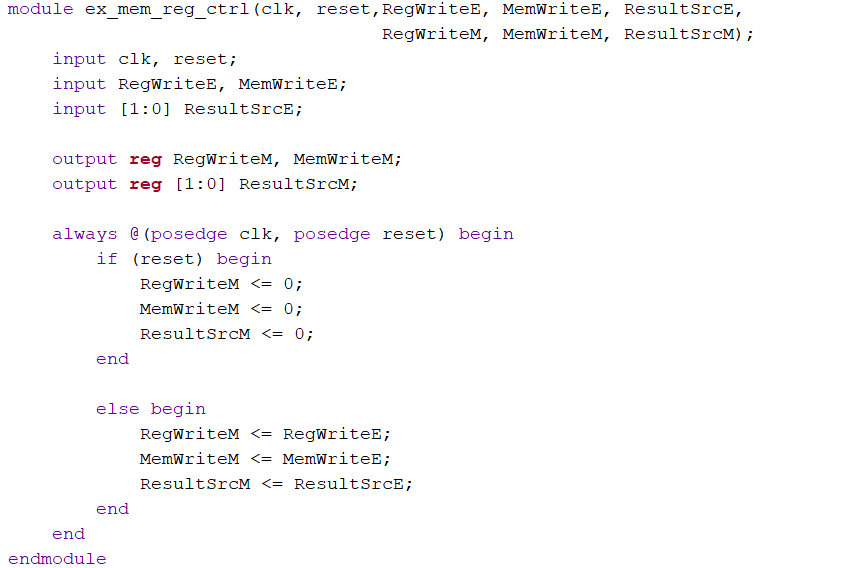
1-bit MemWriteM (Memory write enable from Memory Access)

2-bit ResultSrcM (Selects the source of the result (e.g., ALU result, memory output, PC+4))

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of ex\_mem\_reg\_ctrl (controller) in Verilog

**Commonly asked Questions regarding ex\_mem\_reg\_ctrl module:**

**Q1. Is there a need for a clear signal in this module (like in id\_ex\_reg\_ctrl)?**

A. Generally, a clear signal is not included here because:

* Hazards (like control/data hazards) are usually resolved in earlier stages (ID or EX).
* The MEM stage is closer to write-back, so flushing control is less common unless a full reset occurs.

**A13. Memory Access / Write Back Register (controller)*(mem\_wb\_reg\_ctrl.v)***

The mem\_wb\_reg\_ctrl module is a pipeline control register that separates the memory stage from the writeback stage in a pipelined RISC-V processor.

mem\_wb\_reg\_ctrl specifically handles the control signals generated during the memory stage, and passes them forward into the writeback stage.

Inputs: 1-bit clk (Clock input)

1-bit reset (clears all control signals)

1-bit RegWriteM (Register file write enable from Memory Access)

1-bit MemWriteM (Memory write enable from Memory Access)

2-bit ResultSrcM (Selects the source of the result (e.g., ALU result, memory output, PC+4))

Outputs: 1-bit RegWriteW (Register file write enable from Write Back)

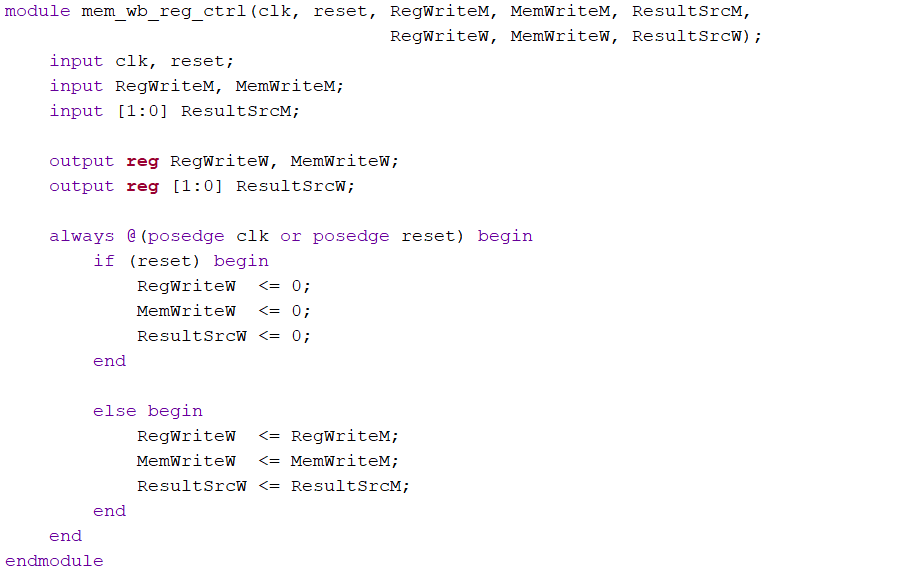
1-bit MemWriteW (Memory write enable from Write Back)

2-bit ResultSrcW (Selects the source of the result (e.g., ALU result, memory output, PC+4))

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of mem\_wb\_reg\_ctrl (controller) in Verilog

**Commonly asked Questions regarding mem\_wb\_reg\_ctrl module:**

**Q1. Is there a need for a clear signal in this module (like in id\_ex\_reg\_ctrl)?**

A. Generally, a clear signal is not included here because:

* Hazards (like control/data hazards) are usually resolved in earlier stages (ID or EX).
* The MEM stage is closer to write-back, so flushing control is less common unless a full reset occurs.

**Q2. What would happen if RegWriteW is not passed correctly?**

A If RegWriteW is incorrect, the processor might fail to write back valid results to registers.

**A14. Write Back MUX** ***(wb\_mux.v)***

The ALU has the capability to carry out arithmetic operations like addition (A+B) or logical operation like equality comparison (A=B). Depending on the specific instructions being executed, the output of the ALU could be either a memory address or the result obtained from the ALU operation. To handle this situation, a MUX is needed to make decision between selecting the data address or the ALU output value for writing back to the register file. The MUX acts as a switch that selects one of the two inputs based on ResultSrc control signal. This allows flexibility and efficient data handling in the processor’s pipeline.

Inputs: 32-bit ALUResultW (ALU result)

32-bit read\_data\_W (reading data from mem\_wb\_reg)

32-bit PCplus4W (PC + 4 value obtained from mem\_wb\_reg)

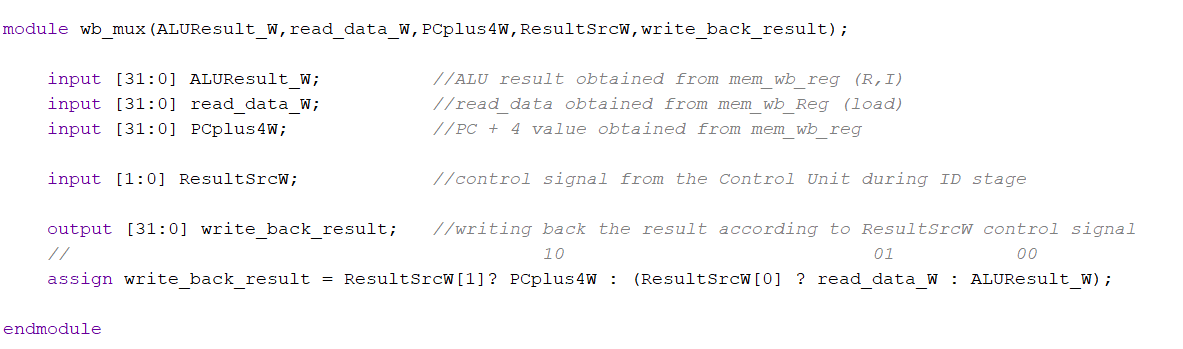
32-bit ResultSrcW (control signal from Control Signal)

Outputs: 32-bit write\_back (writing back to reg\_file)

wire-types: -

reg types: - register (32 registers each having size 32-bit)

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of wb\_mux in Verilog

**Commonly asked Questions regarding pc\_mux** **module:**

**Q1. What happens if the instruction is jal?**

A For jal, we write back the return address (PC+4) to the register. So, ResultSrcW = 2'b10. The MUX selects PCplus4W.

**A15. Program Counter MUX** ***(pc\_mux.v)***

The Program Counter (PC) is a vital component used by the CPU to maintain the current instruction being executed. Under normal circumstances, the program counter increments by a fixed value, typically 4 (corresponding to a 32-bit instruction) for each clock cycle. This ensures that the program counter always points to the memory address of the next instruction to be executed. However, the program counter can be interrupted or modified by a jump signal (jump) from the control unit. When the predetermined conditions are met, the control unit instructs the program counter to deviate from its regular incrementation and instead updated its value to the jump address. Therefore, pc\_mux is used to select the incremented instruction address (PCplus4F) or the jump address (JumpTargetE). The pc\_mux is controlled by the PCSrcE signal. If PCSrcE signal is high, pc\_mux will JumpTargetE on next clock cycle, else PCplus4F will be selected.

Inputs: 32-bit PCplus4F (PC + 4 valued address)

32-bit JumpTargetE (Jumping to another address)

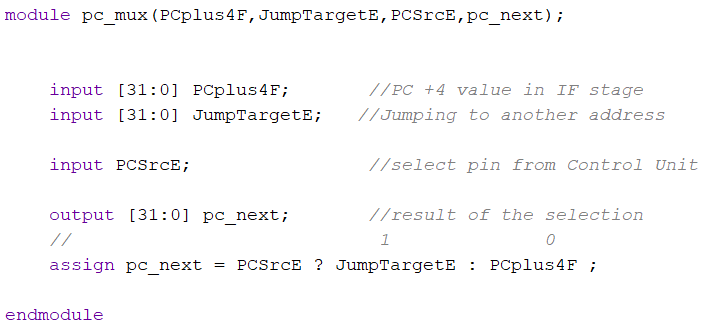
1-bit PCSrcE (program counter selector)

Outputs: 32-bit pc\_next (next pc address)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of pc\_mux in Verilog

**Commonly asked Questions regarding pc\_mux** **module:**

**Q1. What kind of hazard does this help resolve?**

A This helps with control hazards, where the processor must decide whether to continue sequentially or take a branch based on control logic in the execute stage.

**Q2. What is JumpTargetE?**

A It is the computed target address for jump (jal, jalr) or branch (beq, bne, etc.) instructions. It generally comes from the Execute stage (hence the E in the name).

**A16. Register File** ***(reg\_file.v)***

The register file (reg\_file) in a CPU is a crucial component responsible for temporarily storing and providing operands during program execution. It functions as a high-speed memory block containing a set of general-purpose registers, each capable of holding a fixed-width data word (typically 32 bits). The register file typically interfaces with four key inputs: RegWrite, rs1, rs2, and write\_data. The RegWrite control signal enables or disables write operations. When asserted, the value in write\_data is written to the register specified by rd\_w. The rs1 and rs2 fields, derived from Instruction[19:15] and Instruction[24:20] in the IF/ID pipeline register, specify the source registers whose values are read and forwarded to the ID/EX pipeline register for ALU operations in the Execute stage. This structure ensures seamless data access and modification within the pipelined architecture.

Inputs: 1-bit clk (clock input)

1-bit RegWrite (control signal to write to reg\_file or not)

5-bit rs1, rs2 (source registers index value from if\_id\_reg)

5-bit rd\_w (destinationregister indexvalue from mem\_wb\_reg)

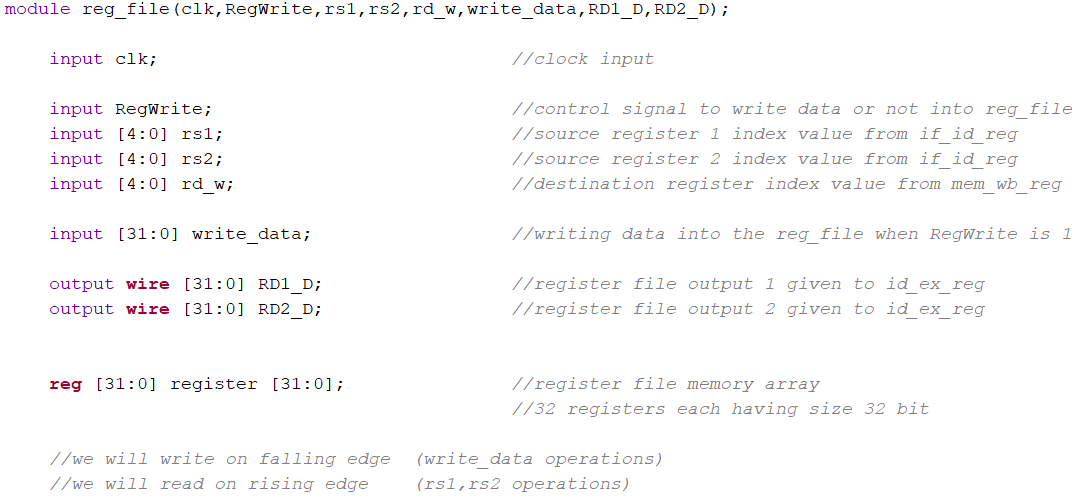
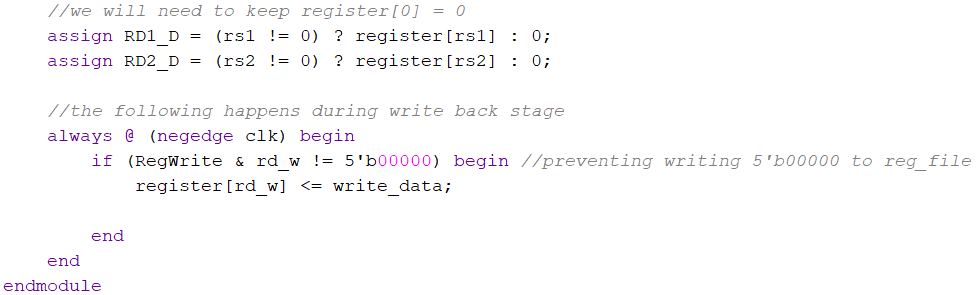
32-bit write\_data (writing data to reg\_file when RegWrite =1)

Outputs: 32-bit RD1\_D, RD2\_D (reg\_file outputs given to id\_ex\_reg)

wire-types: -

reg types: - register (32 registers each having size 32-bit)

Refer to the complete diagram IN PAGE 19 for clarity.

Implementation of reg\_file in Verilog

**Commonly asked Questions regarding reg\_file** **module:**

**Q1. When does write happen in this register file?**

A. On the falling edge of the clock (negedge clk) during the Write-Back stage of the pipeline. Using negedge here models the time gap between reading and writing.

**Q2**. **When does read happen?**

A. Reads are combinational (i.e., immediate) and continuous — no clock needed. The values of RD1\_D and RD2\_D reflect the register contents for rs1 and rs2.

**Q3. Why are outputs RD1\_D and RD2\_D declared as wire?**

A Because they are assigned using assign (combinational logic), not inside always blocks.

**Q4. Why register[0] is hardwired to 0?**

A Avoids Unintended Overwrites: Writing to x0 has no effect; this helps avoid bugs from accidentally overwriting a critical value.

**A17. 3x1 Multiplexer** ***(mux\_3\_1.v)***

A 3x1 multiplexer is a fundamental combinational circuit that selects one of four input signals and forwards it to the output based on a single select line. Here, we considered the 4th input signal to be 0. It acts like a digitally controlled switch.

Inputs: 32-bit A (input A)

32-bit B (input B)

32-bit C (input C)

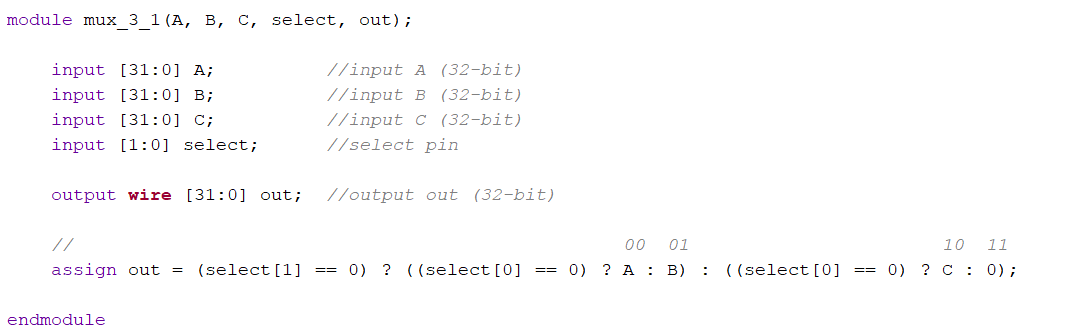
2-bit select (select pin)

Outputs: 32-bit out (output from 3x1 MUX)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of mux\_3\_1 in Verilog

**A18. 2x1 Multiplexer** ***(mux\_2\_1.v)***

A 2x1 multiplexer is a fundamental combinational circuit that selects one of two input signals and forwards it to the output based on a single select line.

Inputs: 32-bit A (input A)

32-bit B (input B)

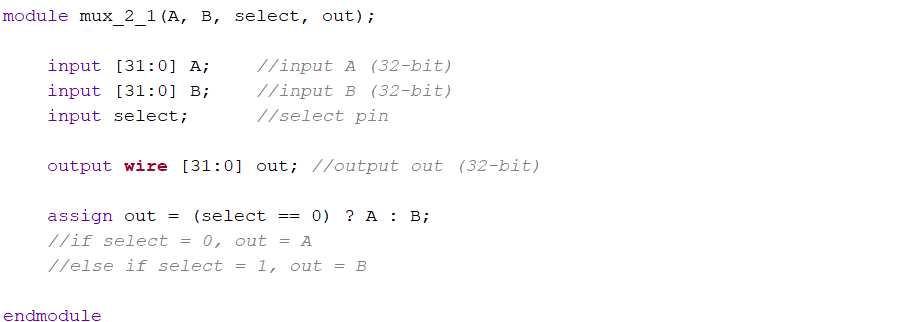
1-bit select (select pin)

Outputs: 32-bit out (output from 2x1 MUX)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of mux\_2\_1 in Verilog

**A19. Reset Flip-Flop** ***(reset\_ff.v)***

The reset\_ff module is a 32-bit register with an asynchronous reset and a stall control input. It is typically used in pipelined processor designs to store and propagate data between pipeline stages while supporting pipeline control mechanisms like stalling and flushing.

Inputs: 1-bit clk (input clock)

1-bit reset (active high reset)

32-bit d\_in (data in pin)

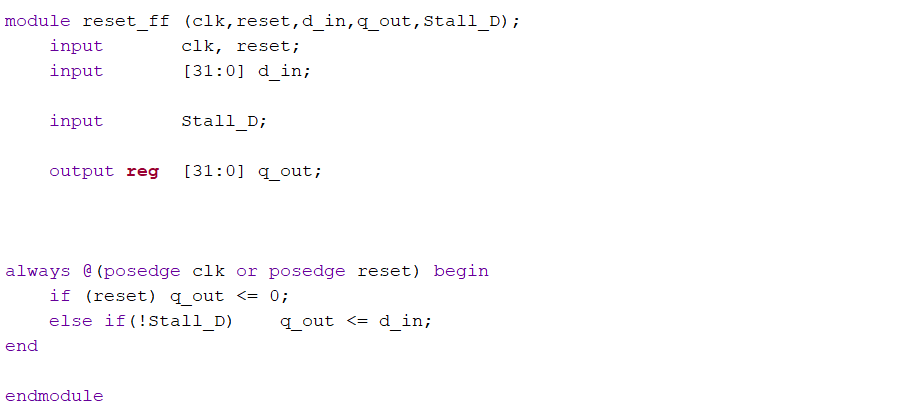
1-bit Stall\_D (control signal which determines whether to hold the value or latch it up in next clk cycle)

Outputs: 32-bit q\_out (output from the flip-flop)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of reset\_ff in Verilog

**Commonly asked Questions regarding reset\_ff** **module:**

**Q1. Describe Asynchronous Reset**

A. Asynchronous Reset (reset): When reset is high, the output q\_out is immediately cleared to 0, regardless of the clock or other inputs.

**Q2. Describe Stall Control**

A Stall Control (Stall\_D):

* When Stall\_D is high, the register holds its current value (i.e., stalling the data flow).
* When Stall\_D is low, the input d\_in is latched into q\_out on the rising clock edge.

**A20. Hazard Detection Unit** ***(hazard\_detection\_unit.v)***

The Hazard Unit is an essential part of a pipelined CPU that identifies and manages situations where instruction execution could go wrong due to data dependencies or conflicts. These scenarios, known as hazards, occur when instructions rely on data that hasn't yet been fully processed or written back. To maintain correct program execution and avoid unnecessary delays, the hazard unit detects these dependencies and applies solutions like stalling or forwarding. Instead of waiting for the Write Back (WB) stage to update the register file, the CPU retrieves the needed data directly from intermediate pipeline registers. This approach ensures that subsequent instructions receive their operands in time, preserving the efficiency and accuracy of the instruction pipeline.

There are 2 solutions to handle hazards namely forwarding and stalling.

A. *Forwarding Unit*

Forwarding is a technique used by hazard unit to handle data dependencies between instruction and mitigate hazards. It allows the CPU to forward data from the producer instruction to the consumer instruction, bypassing intermediate pipeline stages and avoid the need for stalls or bubbles.

Hazard conditions can be handled by 2 forwarding control signals

namely forward\_Amux and forward\_Bmux. forward\_Amux is used to control Forward MuxA and forward\_Bmux is used to control Forward MuxB.

**A20.A.1 Forward Multipliexer A*(forward\_Amux.v)***

Forward Multiplexer A takes the input of RD1\_E, ALUResult\_M, and write\_back\_result. RD1\_E is referred as register value. ALUResult\_M is referred to the alu result in memory stage whereas write\_back\_result is referred to the results in writeback stage. Both ALUResult\_M and write\_back\_result are forwarded values. ForwardA control signal is used to select either the register file value or the forwarded values.

Inputs: 32-bit RD1\_E (register value from reg\_file)

32-bit ALUResult\_M (forwarded value from ex\_mem\_reg)

32-bit write\_back\_result(result from wb\_mux)

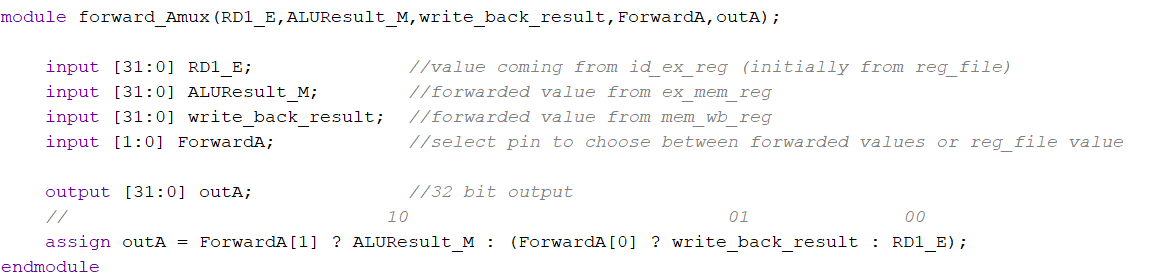
2-bit ForwardA (select pin to choose from ALUResult\_M and RD1\_E)

Outputs: 32-bit outA (32-bit output)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of forward\_Amux in Verilog

**Commonly asked Questions regarding forward\_Amux** **module:**

**Q1. Why is ForwardA[1] given priority over ForwardA[0]?**

A This is a deliberate prioritization in hardware. The EX/MEM forwarding path (ALUResult\_M) typically contains newer data than MEM/WB (write\_back\_result), so it must be checked first to ensure the most recent value is used.

**Q2.** **What would happen if forwarding logic was not used?**

A Without forwarding, the pipeline would have to stall (insert NOPs) whenever an instruction depends on the result of a previous instruction still being processed. This reduces performance due to idle cycles.

**A20.A.2 Forward Multipliexer B*(forward\_Bmux.v)***

Forward Multiplexer A takes the input of RD2\_E, ALUResult\_M, and write\_back\_result. RD2\_E is referred as register value. ALUResult\_M is referred to the alu result in memory stage whereas write\_back\_result is referred to the results in writeback stage. Both ALUResult\_M and write\_back\_result are forwarded values. ForwardB control signal is used to select either the register file value or the forwarded values.

Inputs: 32-bit RD2\_E (register value from reg\_file)

32-bit ALUResult\_M (forwarded value from ex\_mem\_reg)

32-bit write\_back\_result(result from wb\_mux)

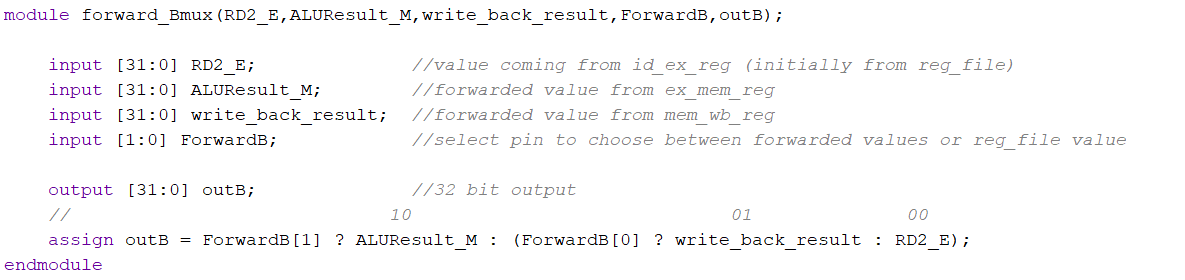
2-bit ForwardB (select pin to choose from ALUResult\_M and RD2\_E)

Outputs: 32-bit outB (32-bit output)

wire-types: -

reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.



Implementation of forward\_Bmux in Verilog

B. *Stalling*

This solution to handle hazards will be discussed later.

**INTEGRATING OUR RISC-V PROCESSOR**

The RISC-V processor was developed by integrating all the individually verified modules into a cohesive design. The integration focused on three main units: the controller unit, the hazard detection unit, and the datapath unit.

* The **controller unit** generates the necessary control signals based on the instruction currently being executed.
* The **hazard unit** identifies potential data or control hazards during execution and takes appropriate steps to handle them, ensuring smooth pipeline operation.
* The **datapath unit** facilitates the flow of data across all five pipeline stages, enabling efficient instruction processing.

**B.1 Controller Unit *(controller.v)***

The controller unit is responsible for generating control signals that drive the operation of the RISC-V processor's datapath. Based on the instruction's opcode, funct3, and funct7 fields, it decodes the instruction type and determines how data flows through the pipeline.

It internally connects a main decoder, ALU decoder, and pipeline control registers (id\_ex\_reg\_ctrl, ex\_mem\_reg\_ctrl, mem\_wb\_reg\_ctrl) to produce stage-specific control signals. Additionally, it handles branch decision logic, including special conditions for bne, blt, and bge using the ALU's Zero and Sign outputs.  
Through effective instruction decoding, hazard flushing, and PC control signal generation, the controller ensures correct execution across all five pipeline stages.

Inputs: 1-bit clk (clock input)

1-bit reset (active high reset pin)

7-bit opcode (7 bit opcode)

3-bit funct3 (3 bit funct3)

1-bit funct7bit5 (5th bit of funct7)

1-bit Zero\_E (Zero result of ALU in execute stage)

1-bit Sign\_E (Sign result of ALU in execute stage)

1-bit Flush\_E (Used to flush the pipeline)

Outputs: 1-bit ResultSrcE0 (LSB bit of ResultSrcE control signal)

2-bit ResultSrcW (control signal to wb\_mux)

1-bit MemWriteM (sends signal to data\_memory as write\_enable)

1-bit PCJalSrcE (sends signals to PC Jal mux)

1-bit PCSrcE (sends signals to PC mux)

1-bit ALUSrcAE (select operand 1 source (rs1 or PC))

2-bitALUSrcBE (select operand 2 source (rs2 or imm or +4 or any-other))

1-bit RegWriteM (control signal to control write operation in reg\_file (M))

1-bit RegWriteW (control signal to control write operation in reg\_file (W))

3-bit ImmSrcD (immediate source selector)

4-bit ALUControlE (controls which ALU operation to perform)

wire-types: - 2-bit ALUOpD (ALU operation type from main\_decoder to alu\_decoder)

2-bit ResultSrcD (control signal to wb\_mux)

2-bit ResultSrcE (control signal to wb\_mux)

2-bit ResultSrcM (control signal to wb\_mux)

1-bit RegWriteD (control signals to control write operation in reg\_file (D))

1-bit RegWriteE (control signals to control write operation in reg\_file (D))

4-bit ALUControlD (controls which ALU operation to perform)

1-bit ALUSrcAD (select operand 1 source (rs1 or PC))

1-bit BranchD, BranchE (branch control conditions based on funct3)

1-bit MemWriteD, MemWriteE (sends signal to data\_memory as write\_enable)

1-bit JumpD, JumpE (used in pc\_mux for jump interrupts)

2-bit ALUSrcBD (select operand 2 source (rs2 or imm or +4 or any-other))

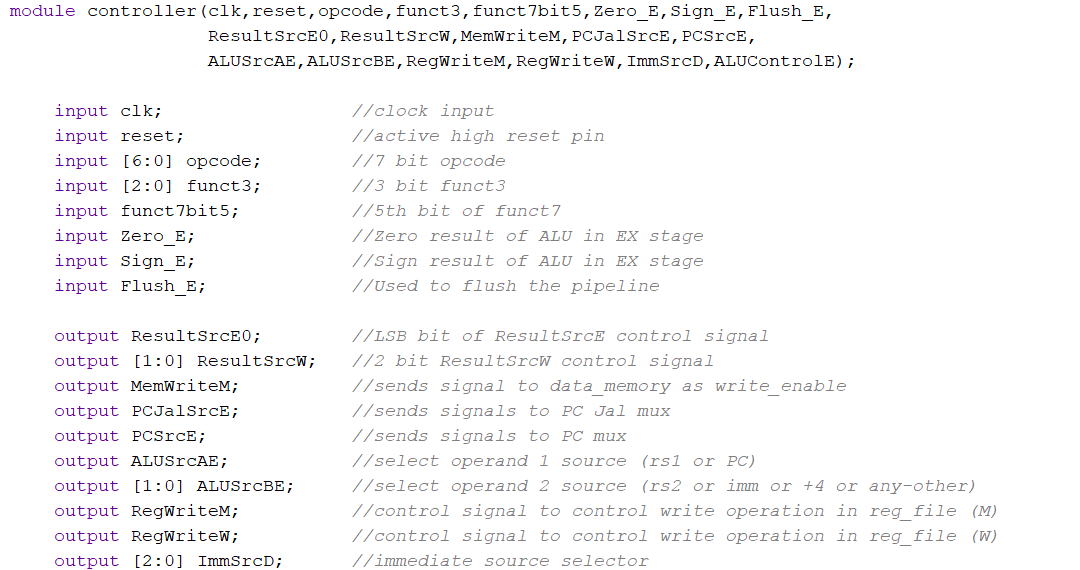
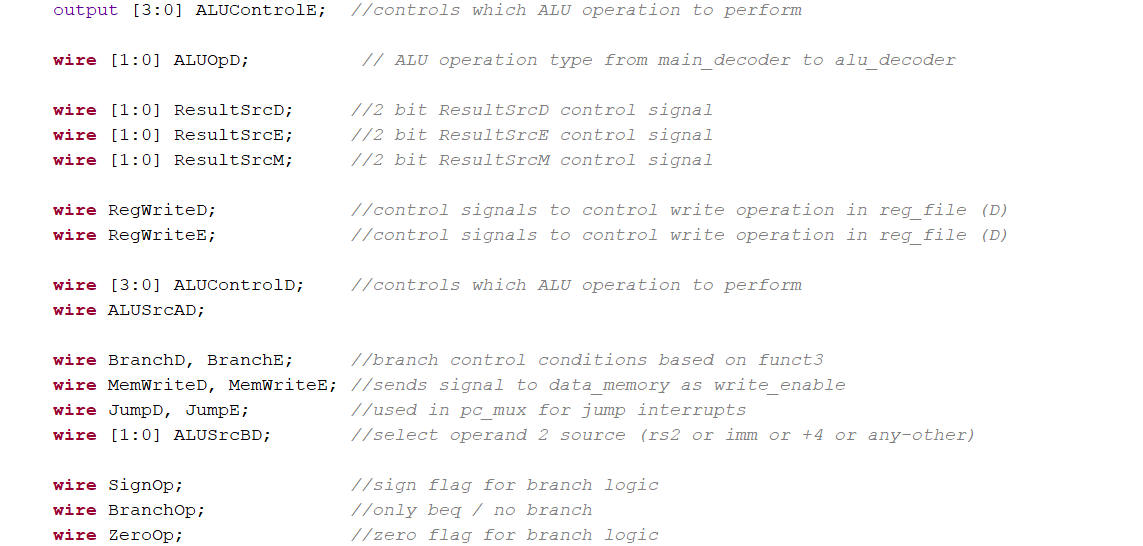
1-bit SignOp (sign flag for branch logic)

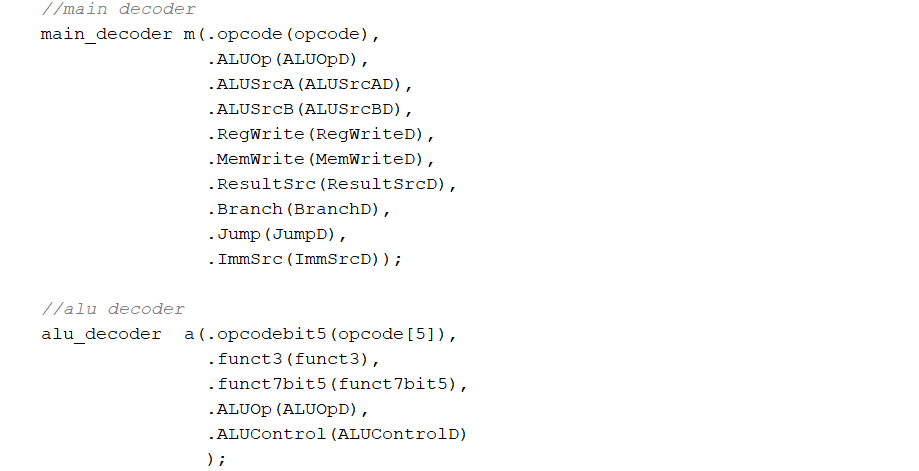
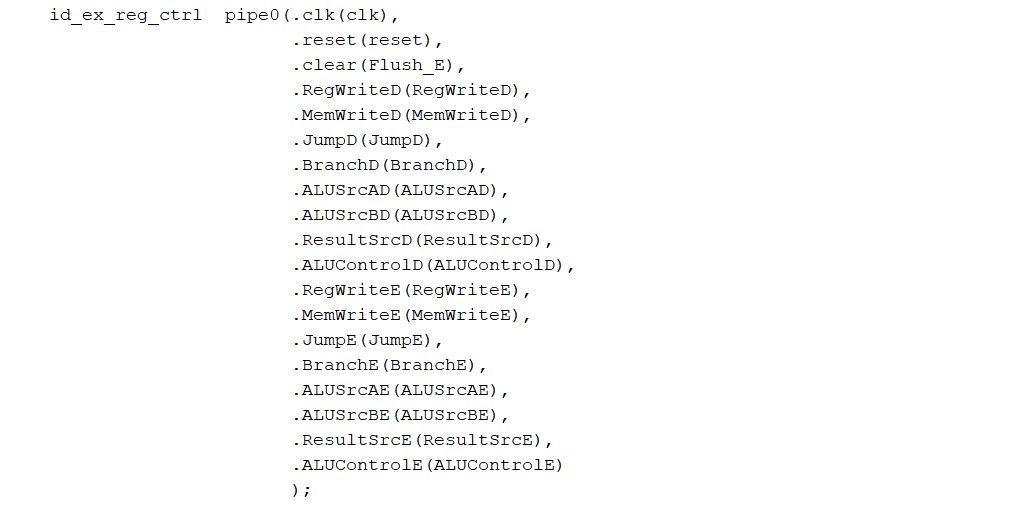
1-bit BranchOp (only beq / no branch)

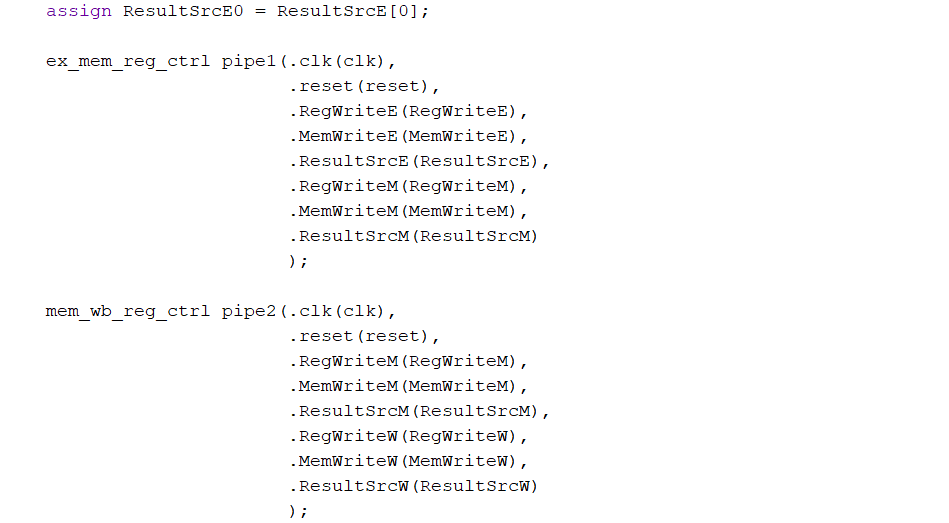
1-bit ZeroOp; (zero flag for blanch logic)

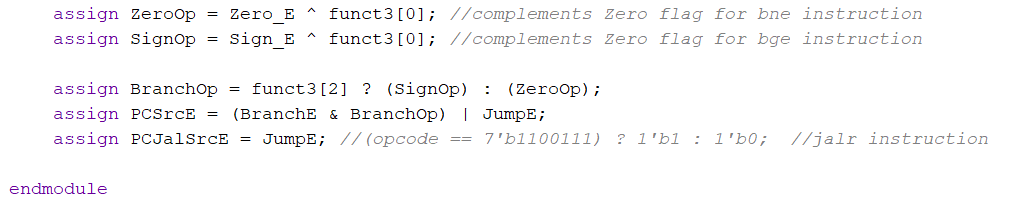
reg types: -

Refer to the complete diagram IN PAGE 19 for clarity.





Implementation of controller in Verilog

**Commonly asked Questions regarding forward\_Amux** **module:**

**Q1.** **Why are there separate pipeline register modules like id\_ex\_reg\_ctrl, ex\_mem\_reg\_ctrl, and mem\_wb\_reg\_ctrl used here?**

A. These modules isolate and propagate control signals through the pipeline stages (ID→EX, EX→MEM, MEM→WB) ensuring that each stage has the correct set of control signals, even with instruction overlap in a pipelined processor.

**Q2. What does the signal Flush\_E do in this context?**

A. Flush\_E clears the ID/EX control pipeline register (e.g., during a mispredicted branch), preventing unintended instruction execution by nullifying control signals.

**Q3. What is the purpose of the ResultSrcE0 output?**

A. ResultSrcE0 is the least significant bit of the ResultSrcE signal, often used as a simple binary selector (e.g., in muxes deciding between ALU output and memory read value).

**Q4. How does the controller determine whether to branch?**

A. Branch decision is based on the instruction's funct3 and ALU status flags (Zero\_E, Sign\_E). The logic computes BranchOp, which is combined with the BranchE signal to generate PCSrcE.

**Q5. Why is funct3[0] used in XOR with Zero\_E and Sign\_E?**

A. funct3[0] differentiates between branch instructions like beq (==) and bne (!=), or blt (<) and bge (>=), by toggling the interpretation of ALU comparison flags.

**Q6. What is the difference between PCSrcE and PCJalSrcE?**

A. PCSrcE decides whether the next PC should be the jump/branch target or PC+4. PCJalSrcE is used to decide between jal and jalr targets specifically, which may be needed for further mux control.

**Q7. How is ALUControlE generated?**

A. It is generated through the alu\_decoder, which takes opcode[5], funct3, funct7bit5, and ALUOpD as inputs and produces a 4-bit ALU control signal (ALUControlD), which is then pipelined to execute stage as ALUControlE.

**Q8. What is the function of main\_decoder and alu\_decoder?**

A. main\_decoder: Decodes the opcode into high-level control signals (e.g., branch, jump, MemWrite).

alu\_decoder: Further refines ALU control based on opcode, funct3, funct7 to select specific ALU operations.

**Q9. Why are RegWrite and MemWrite pipelined through multiple stages?**

A. These control signals must reach the corresponding stages (MEM and WB) at the correct time. For instance, write-back (RegWriteW) only occurs in the WB stage, so it must be passed through each pipeline register stage accordingly.

**Q10. What is the purpose of the ALUSrcAE, ALUSrcBE signals?**

A. These control which inputs are fed into the ALU:

* ALUSrcAE: Selects between register rs1 and PC for the first ALU operand.
* ALUSrcBE: Selects between register rs2, immediate, or +4 for the second ALU operand.

**Q11. What does the JumpE signal represent?**

A. The JumpE signal indicates whether the current instruction in the Execute stage is a jump (jal or jalr). It is used in conjunction with BranchE to control program flow redirection.

**Q12. What happens when both BranchE and JumpE are active?**

A. The logic in PCSrcE = (BranchE & BranchOp) | JumpE ensures that jumps have higher priority. If both are active, a jump will override the branch.

**Q13. Why is Flush\_E not used globally to flush all stages?**

A. Flush\_E only affects the ID/EX stage because it's typically used for control hazards (e.g., wrong branch prediction). Flushing other stages (e.g., MEM or WB) could result in data loss or incorrect write-backs.

**Q14. Why are so many intermediate control signals declared as wire and not reg?**

A. In synthesizable Verilog, control signals that are purely combinational logic (not stored over time) are best declared as wire. Registers (reg) are used only when storing values across clock cycles.

**Q15. How does the controller support pipelining without causing data hazards?**

A. The controller does not handle hazards directly. Instead, it works with the Hazard Detection Unit and Forwarding Unit, which generate stall, flush, and forward signals based on pipeline register contents.

**B.2 Datapath *(datapath.v)***

The datapath module forms the core of the pipelined RISC-V processor, facilitating the sequential flow and execution of instructions across all five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). It orchestrates the movement of instructions and data through pipeline registers (if\_id\_reg, id\_ex\_reg, ex\_mem\_reg, mem\_wb\_reg) while handling ALU operations, register file interactions, forwarding, and hazard control. This module connects major components like the ALU, multiplexers, register file, immediate extender, and memory interfaces, ensuring proper data computation and control signal propagation. It is designed to be stall- and flush-aware to accommodate hazard resolution and maintain instruction correctness in the pipelined architecture.

Inputs: 1-bit clk (clock input)

1-bit reset (active high reset pin)

2-bit ResultSrcW (select pin for wb\_mux)

1-bit PCJalSrcE ()

1-bit PCSrcE ()

1-bit ALUSrcAE (5th bit of funct7)

2-bit ALUSrcBE (Zero result of ALU in execute stage)

1-bit RegWriteW (Sign result of ALU in execute stage)

2-bit ImmSrcD (Used to flush the pipeline)

4-bit ALUControlE (5th bit of funct7)

32-bit Instr\_F (Zero result of ALU in execute stage)

32-bit ReadDataM (Sign result of ALU in execute stage)

2-bit ForwardAE (Used to flush the pipeline)

2-bit ForwardBE (Used to flush the pipeline)

1-bit Stall\_D ()

1-bit Stall\_F ()

1-bit Flush\_D ()

1-bit Flush\_E ()

Outputs: 1-bit Zero\_E (LSB bit of ResultSrcE control signal)

1-bit Sign\_E (control signal to wb\_mux)

32-bit PCF (sends signal to data\_memory as write\_enable)

32-bit Instr\_D (sends signals to PC Jal mux)

32-bit ALUResultM (sends signals to PC mux)

32-bit WriteDataM (select operand 1 source (rs1 or PC))

5-bit rs1\_D,rs2\_D,rs1\_E,rs2\_E (select operand 2 source (rs2 or imm or +4 or any-other))

5-bit rd\_E, rd\_M, rd\_W (control signal to control write operation in reg\_file (M))

32-bit ResultW (control signal to control write operation in reg\_file (W))

wire-types: - 32-bit PCD,PCE ()

32-bit ALUResultE,ALUResultM ()

32-bit ReadDataM,ReadDataW ()

32-bit PCNext4F ()

32-bit PCplus4F, PCplus4D, PCplus4E, PCplus4M, PCplus4W ()

32-bit PCTargetE ()

32-bit JumpTargetE ()

32-bit WriteDataE ()

32-bit ImmExtD,ImmExtE ()

32-bit SrcAEfor ()

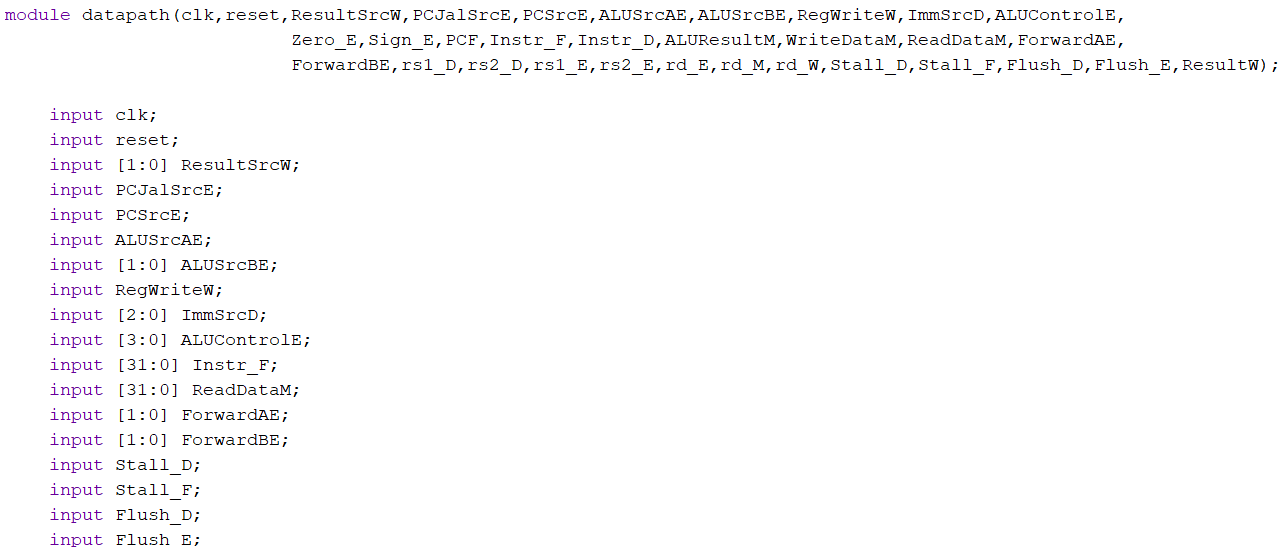
32-bit SrcAE,SrcBE ()

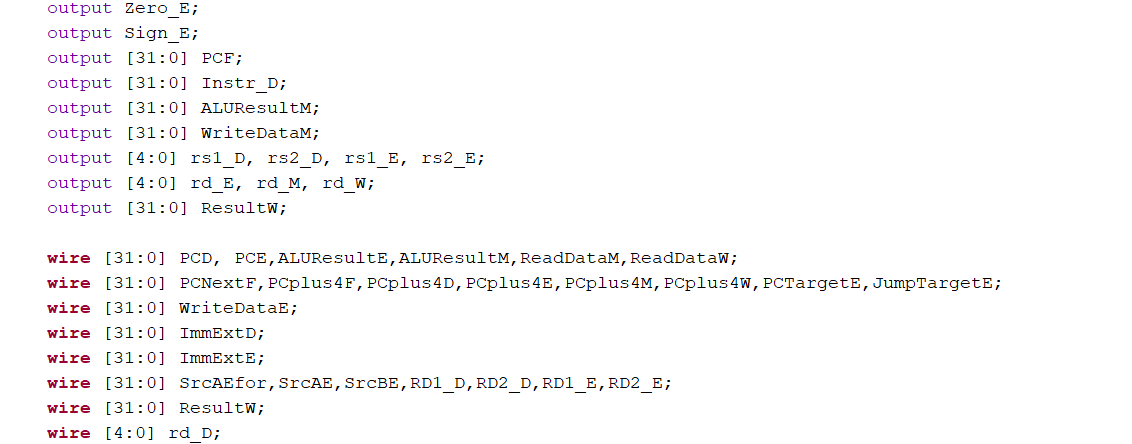
32-bit RD1\_D, RD2\_D, RD1\_E, RD2\_E ()

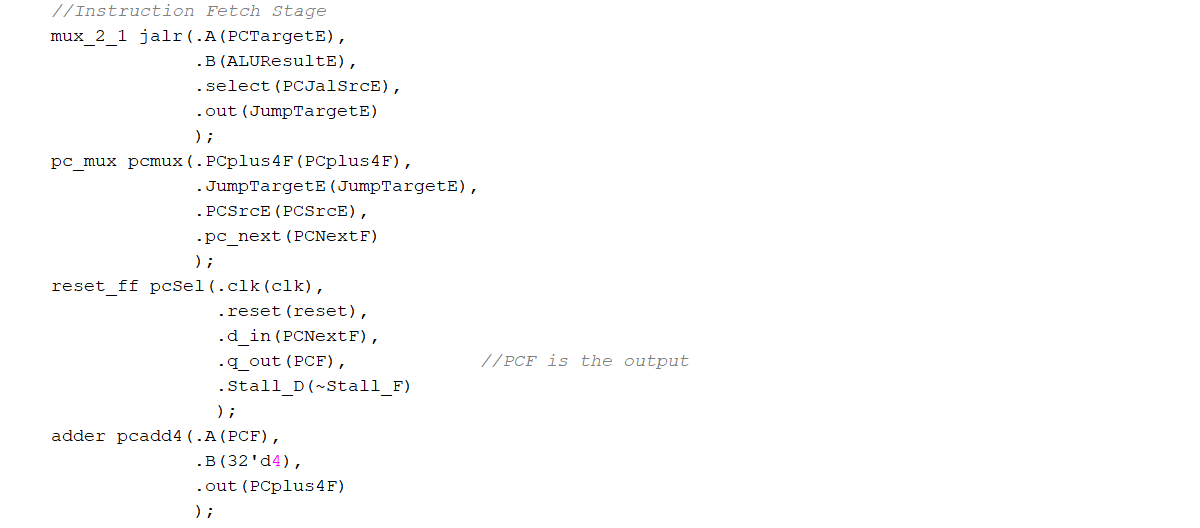
32-bit ResultW ()

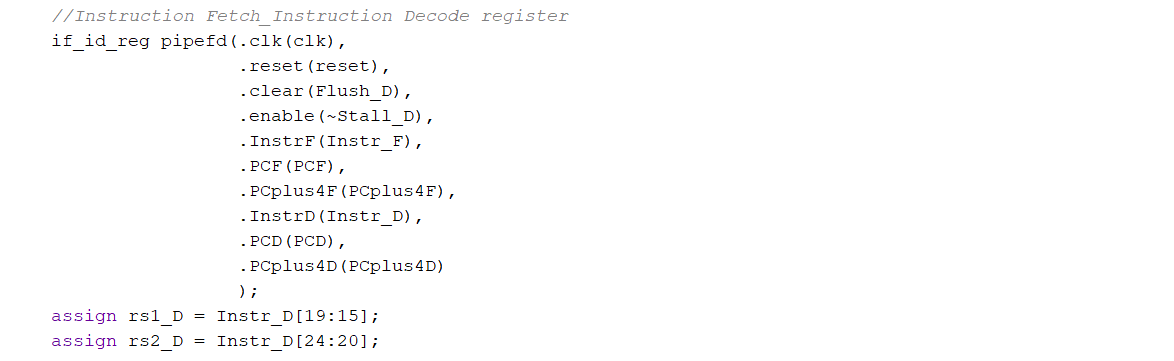
32-bit rd\_D ()

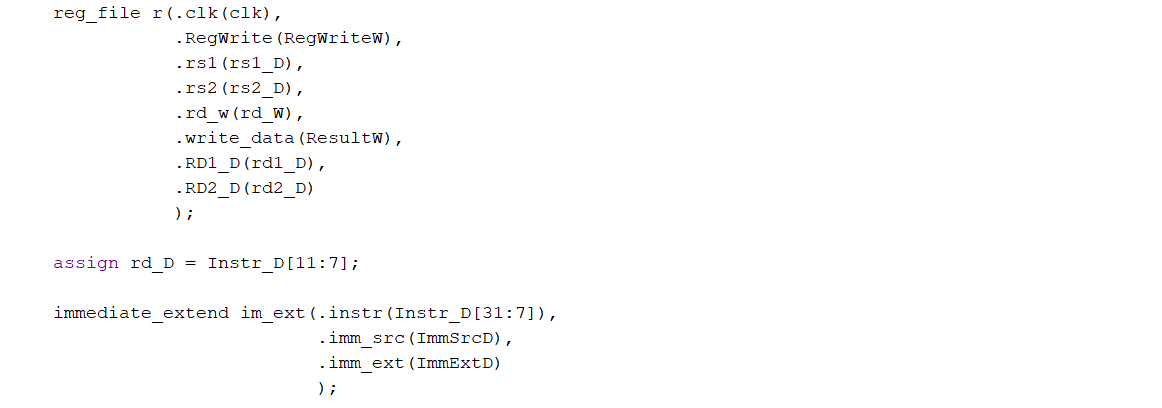
reg types: -

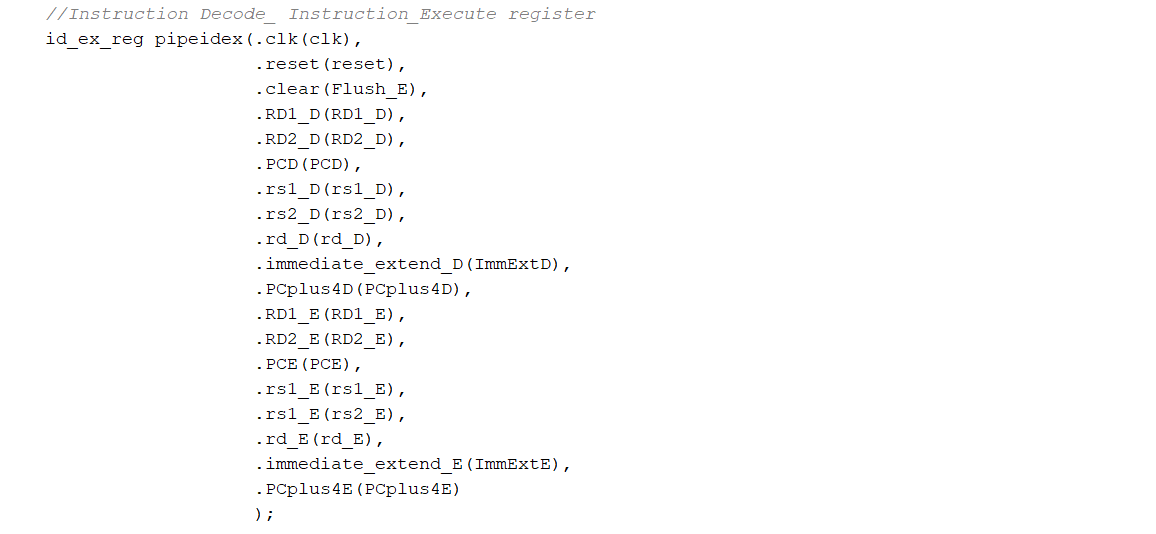


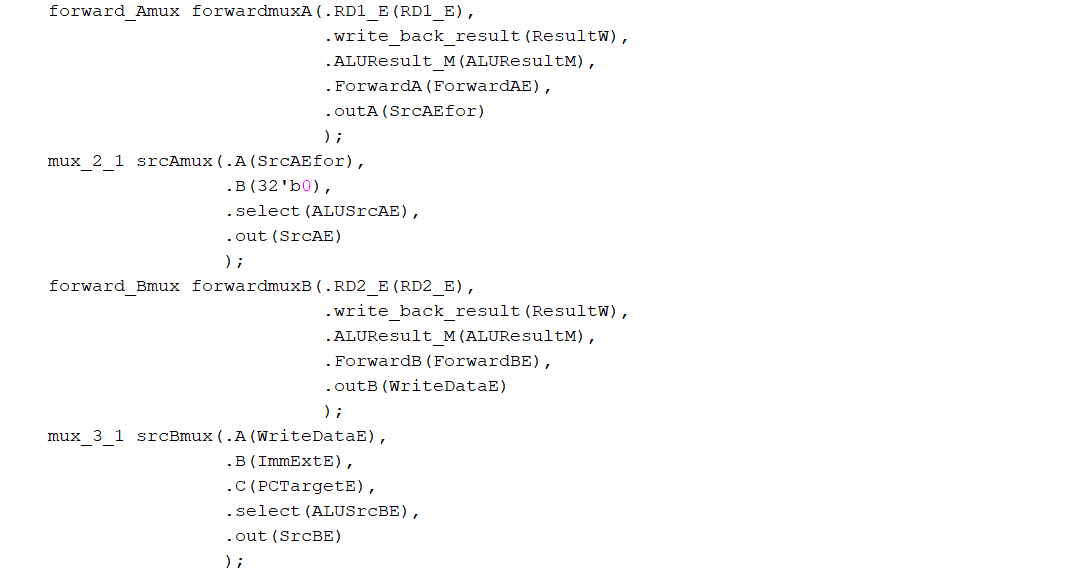


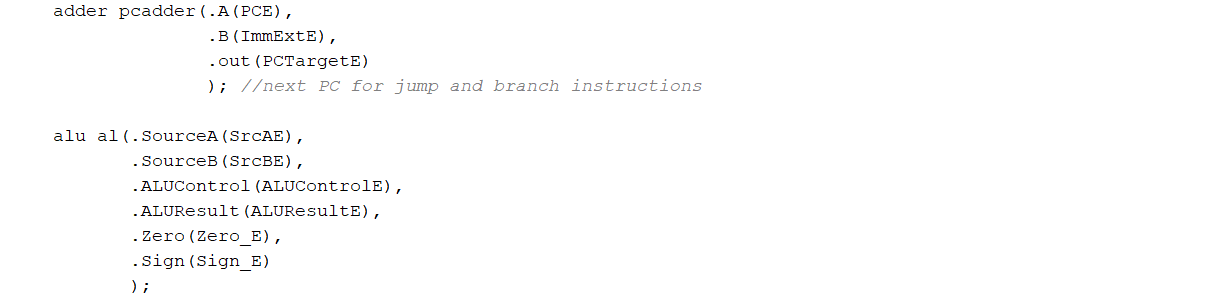


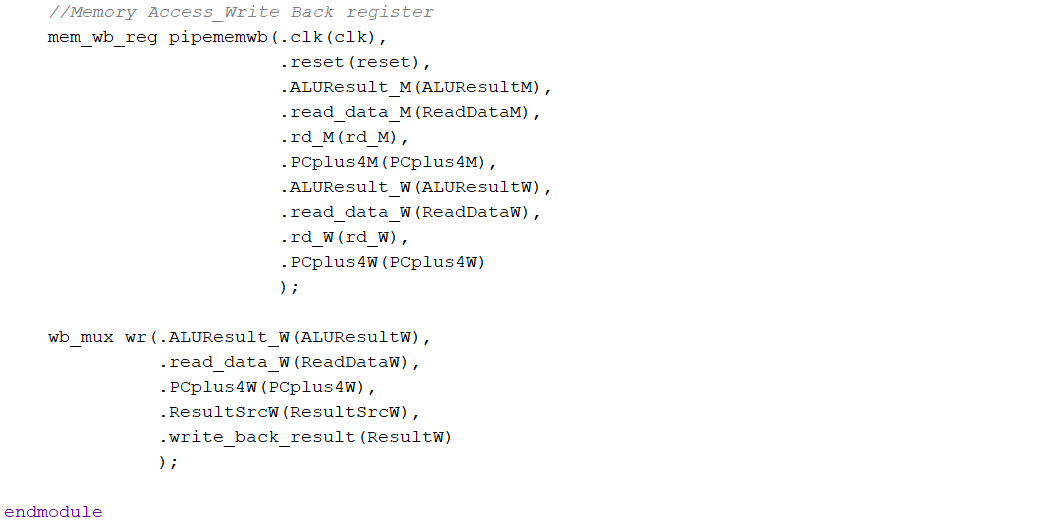
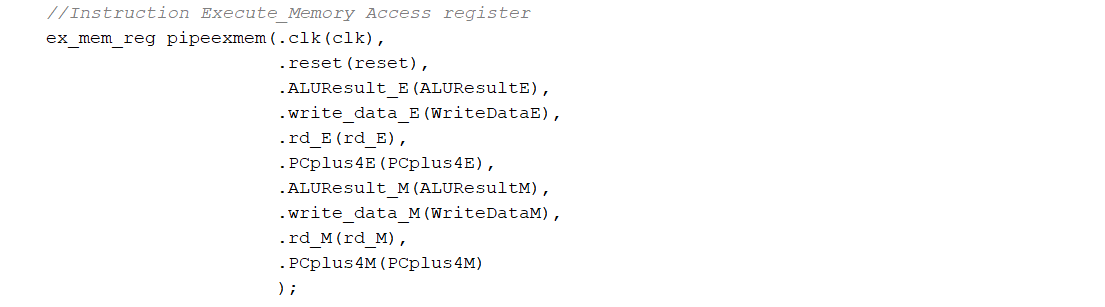












Implementation of datapath in Verilog