I. System overview

This design implements a 5-stage pipelined processor based on the RISC-V RV32I instruction set, with a strong emphasis on stability, simplicity, and efficiency in both area and power. The architecture is highly modular, making it well-suited for FPGA prototyping and future extensions. The processor pipeline consists of five main stages:

- **IF (Instruction Fetch):** Fetches the next instruction from instruction memory.
- **ID (Instruction Decode):** Decodes the instruction and reads operands from the register file.
- **EX (Execute):** Performs arithmetic/logic operations and address calculations.
- **MEM (Memory Access):** Accesses data memory for load/store instructions.
- **WB (Write Back):** Writes computation or memory results back to the register file.

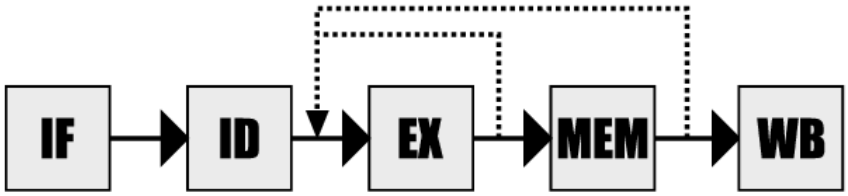

Figure 1. Five-Stage RISC Pipeline

Each stage consists of separate hardware modules, communicating via stage buffers that preserve control and data signals.

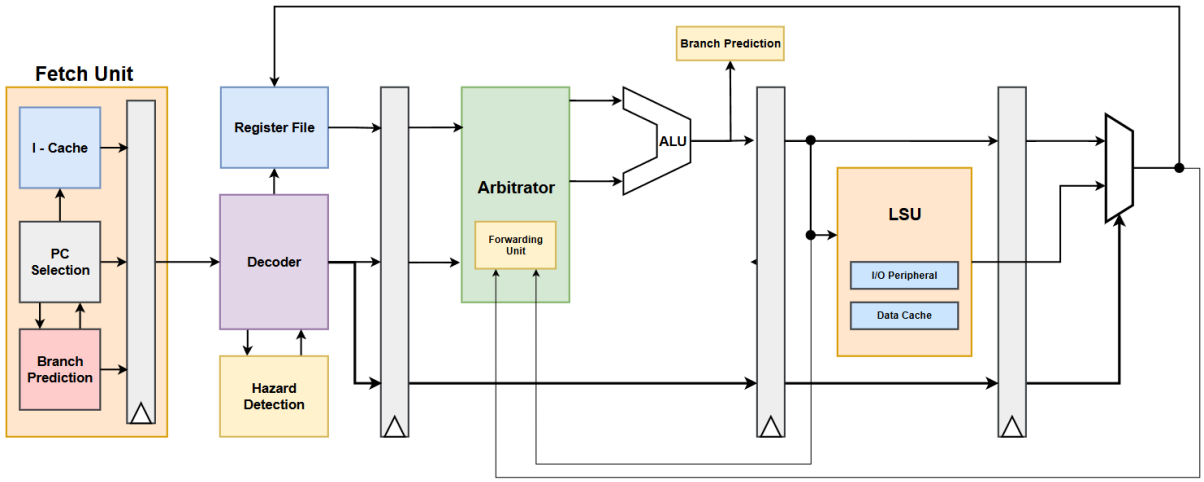

Figure 2. System Overview of the RV32I 5-Stage Pipeline Processor Design

The Fetch stage is encapsulated entirely within a dedicated Fetch Unit, which includes branch prediction logic, a program counter selector and register, and a small instruction cache. It outputs to a stage buffer that separates it from the Decode stage, enhancing modularity and simplifying future upgrades.

In the Execute stage, the Arbitrator plays a central role in operand selection and functional unit management. It routes instructions to available execution units such as the ALU and is designed with future support for modules like the FPU, multiplier, or divider. Integrated within the Arbitrator is a Forwarding Unit, responsible for handling data hazards by forwarding results to dependent instructions without pipeline stalls when possible.

The processor also implements a hazard detection unit to manage cases that cannot be resolved by forwarding alone, introducing stalls only when necessary to preserve correctness. This approach ensures the pipeline remains efficient and responsive under typical workloads.

The Memory stage is managed by a Load/Store Unit (LSU), which handles memory accesses and provides a basic memory-mapped I/O interface. In this simplified version, the LSU includes a small local data memory and a few I/O control registers. Future versions can be extended to include full support for external bus protocols and more complex peripheral communication.

I/O interfacing is handled entirely through the LSU and memory-mapped registers. The processor exposes two 32-bit input ports (for switches and general input) and four 32-bit output ports (labeled A through D), enabling flexible integration with testbenches, peripherals, and embedded applications.

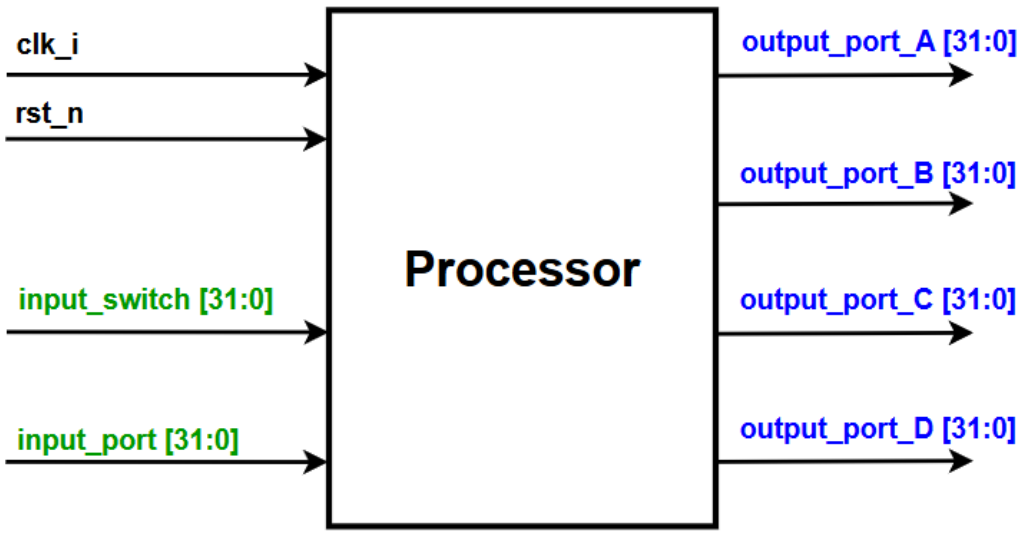

Figure 3. Processor Top-Level Input and Output Pins

Overall, this RV32I processor provides a clean, extensible foundation for future development in RISC-V-based embedded systems, combining modularity with efficiency and support for continued architectural growth.

II. Design Specifications

1. Fetch Unit

The Fetch Unit manages all tasks in the fetch stage, including branch prediction, selecting the next program counter (PC), and reading instructions from the instruction cache. It decides the next PC based on pipeline status and prediction results to fetch the correct instruction each cycle.

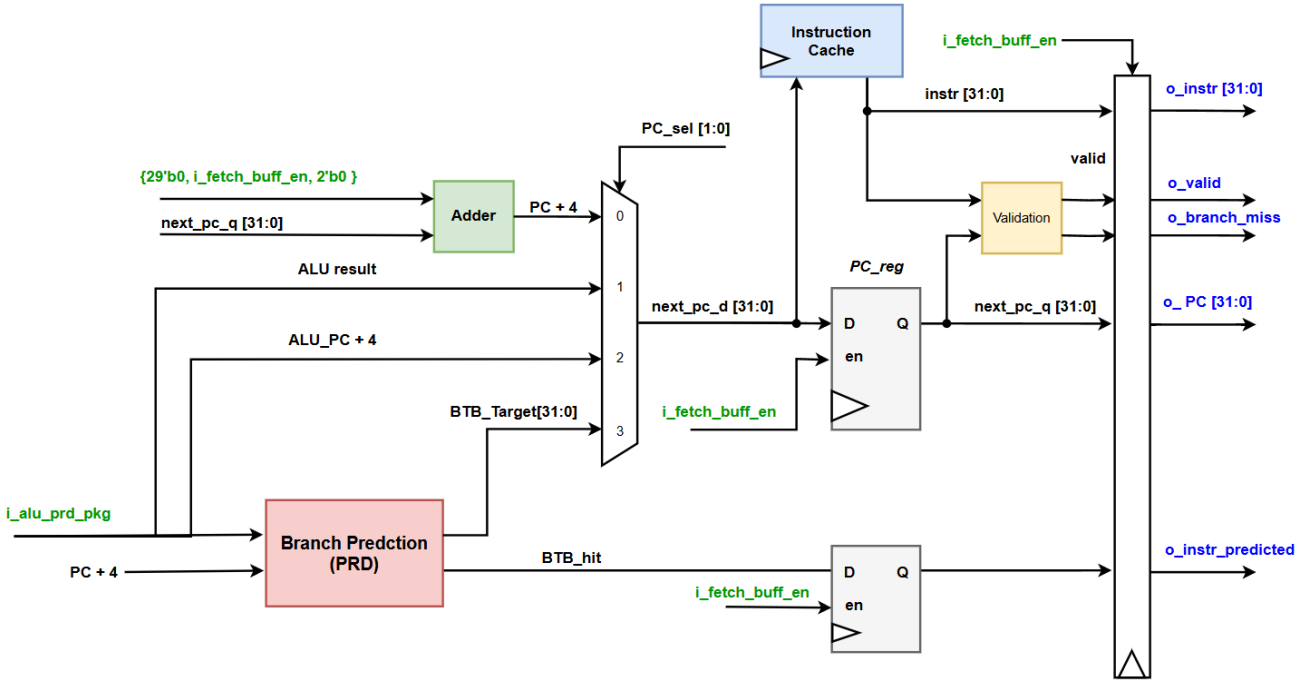

Figure 4. Block diagram of the Fetch Unit

The PC register and stage buffer both use input enable signals to support stalling when needed. Because the instruction cache is accessed using an unregistered PC from the PC multiplexer, the buffer's input enable signal is also used to control the Adder that computes PC + 4. In the event of a pipeline stall, this signal effectively acts as bit 2 of the Adder's 32’d4 input, preventing unnecessary PC updates.

The PC multiplexer supports four PC sources, selected using the PC_sel signal. These sources are listed in the following table:

Table 1. PC selection based on PC_sel signals

PC_sel *(2-bit)*	**Selected Data** *(32-bit)*	**Description**
2'b00	pc_plus4	Default PC increment: used when no branching occurs (sequential instruction flow).
2'b01	alu_update_target	Target address from ALU, used to correct "not taken" branch mispredictions.
2'b10	alu_pc_plus4	PPC + 4 from branch instruction in EX stage, used to restore PC when a "taken" branch was mispredicted.
2'b11	prd_br_target	Predicted branch target from PRD (BTB hit and branch predicted as taken).

The PC_sel signal, which controls the PC multiplexer, is determined based on the branch prediction outcome and whether a misprediction has occurred. It relies on the following three signals:

- **BTB_hit**: Indicates that the current instruction was predicted as a branch (a prediction was made using the Branch Target Buffer).
- **br_miss_taken**: Indicates a misprediction where the branch was predicted as taken, but it turned out to be not taken during execution.
- **br_miss_not_taken**: Indicates a misprediction where no branch was predicted (not taken), but the instruction was actually resulted in taken during execution.

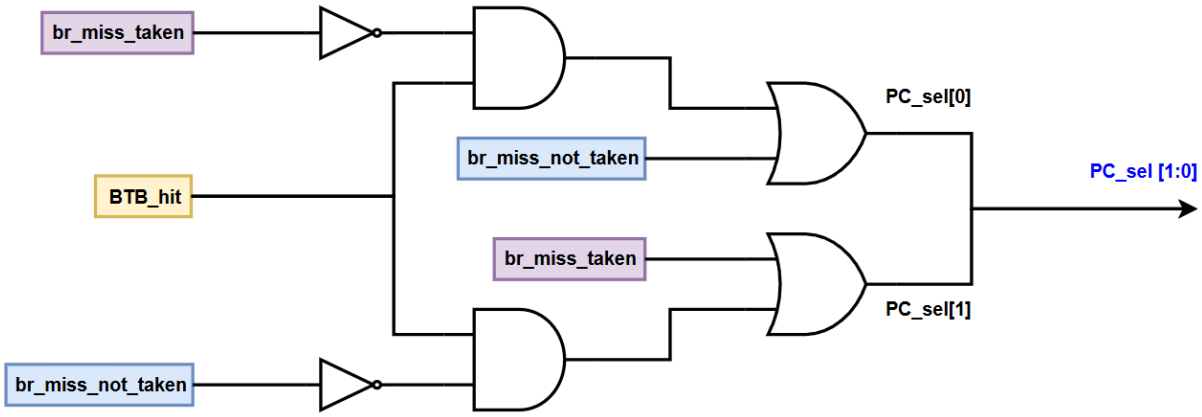

Figure 5. Logic Diagram for PC_sel[1:0] Generation Based on Branch Prediction Outcome

Branch prediction is implemented within the Fetch Unit by maintaining a Branch Target Buffer (BTB) that stores the target addresses and history of previously encountered branch instructions. When a known branch is encountered again, the unit predicts its behavior and provides the stored

target address. As branch instructions execute in the ALU, they send back information to update the branch history or insert new entries into the BTB if the branch is newly encountered.

When a prediction is made, a flag is set to indicate that the current instruction is the result of branch prediction. This flag is passed along with the instruction through the pipeline and is later used to check for misprediction. Based on the actual outcome in the execute stage, the system can detect two types of mispredictions: a "taken" misprediction (when the branch was not predicted but actually taken) and a "not taken" misprediction (when the branch was predicted as taken but wasn't).

Each instruction in the pipeline carries a valid bit that stays with it through all stages of execution. This signal is essential for controlling instruction flow, as it allows the pipeline to flush or cancel an instruction simply by clearing its valid bit. Every stage checks this bit to decide whether to continue processing the instruction, and any stage can invalidate it if a flush is needed. The valid bit is first generated in the Fetch stage, based on whether the instruction fetched from the cache is valid and the PC is properly aligned.

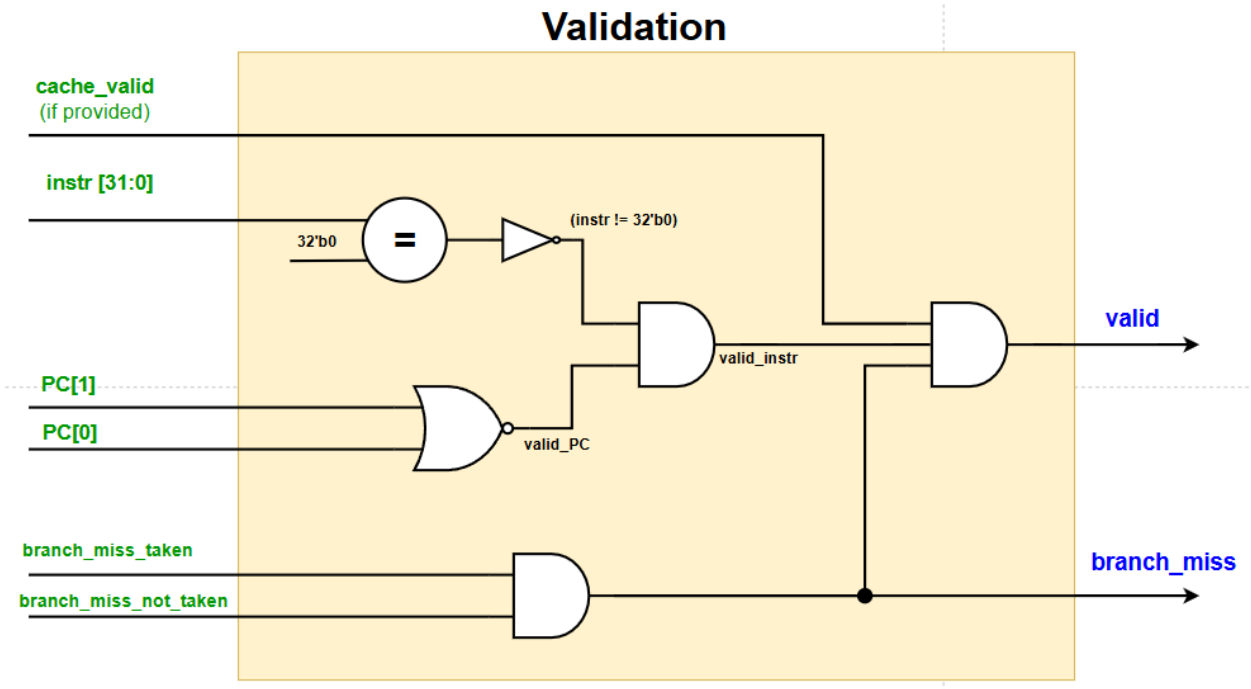

Figure 6. Instruction Valid bit generation

The input and output signals of the Fetch Unit are specified as follows:

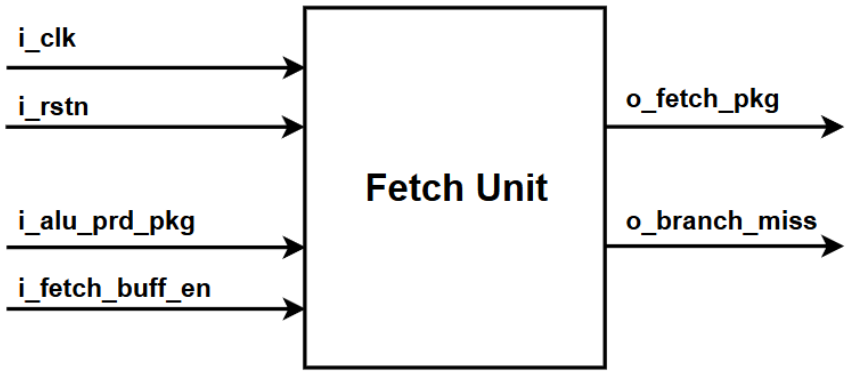

Figure 7. Fetch Unit Top-Level Input and Output Pins

Table 2. Input and Output signals description of the Fetch Unit

No.	Signal Name	Width	Type	Description
1	i_clk	1	Input	System clock pin
2	i_rstn	1	Input	Active-Low reset pin
3	i_fetch_buff_en	1	Input	Enable signal for registers and Fetch stage buffer
4	i_alu_prd_pkg	100	Input	Data package from ALU to Branch Prediction
5	o_fetch_pkg	66	Output	Data package to Decode Stage
6	o_branch_miss	1	Output	Branch misprediction flag

- **Content of *o_fetch_pkg* data package:**

Table 3. Content of o_fetch_pkg data package from Fetch Unit

No.	Signal Name	Width	Description
1	instr	32	Fetched instruction from Instruction Cache
2	pc	32	Program counter value corresponding to the fetched instruction.
3	prd_taken	1	Indicates the fetched instruction was predicted as taken by the branch prediction (PRD).
4	valid	1	Indicates whether the fetched instruction is valid and should proceed in the pipeline.

- **Content of *i_alu_prd_pkg* data package:**

Table 4. Content of i_alu_prd_pkg data package from ALU to Fetch Unit

No.	Signal Name	Width	Description
1	br_update_pc	32	PC of the currently executing Branch instruction in ALU
2	br_pc_plus4	32	PC + 4 of the currently executing branch instruction, used when correcting a taken misprediction
3	br_target	1	Computed target address of the executing branch instruction in the ALU
4	br_update_en	1	Indicates whether a branch instruction is currently executing in the ALU
5	br_taken	1	Indicates whether the branch was taken based on the ALU evaluation result
6	br_valid	1	Indicates the branch instruction in the ALU is valid and its result should be used
7	br_already_predicted	1	Indicates whether the branch instruction was previously predicted, used to detect mispredictions

2. Decode Stage

The Decode Stage (ID) is responsible for interpreting the fetched instruction and generating the necessary control signals and operand values for the Execute and later stages. In this design, the Decode stage performs instruction decoding, reads source operands from the Register File, and includes a Hazard Detection Unit to identify data hazards that cannot be resolved by the Forwarding Unit in the Execute stage. These hazards particularly those involving dependencies on data not yet written back that must be detected early in the ID stage to ensure correct pipeline behavior.

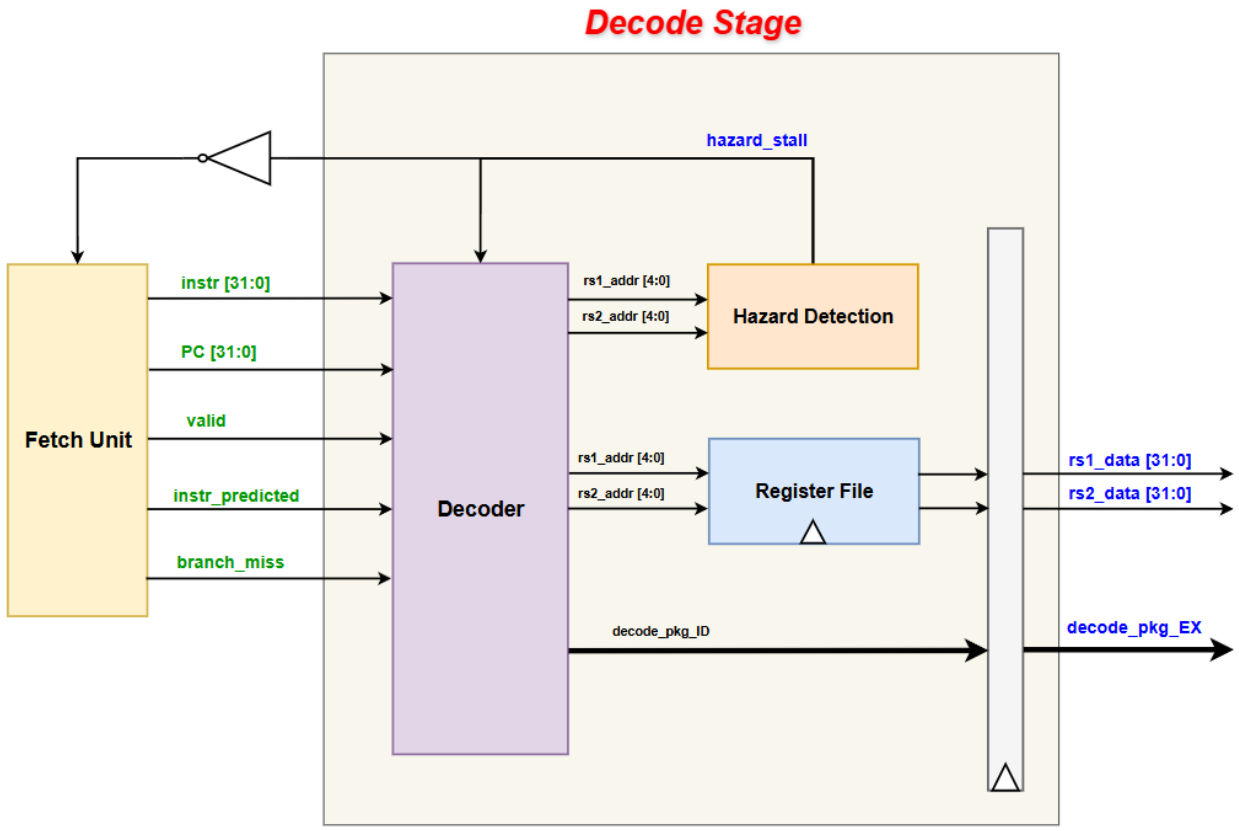

Figure 8. Decode Stage Block Diagram

2.1. Decoder

The Decoder is responsible for translating the raw 32-bit instruction (instr[31:0]) into structured control signals and operand information required by downstream pipeline stages. It interprets the instruction format based on the RISC-V ISA and extracts critical fields such as opcode, register addresses, immediate values, and function codes.

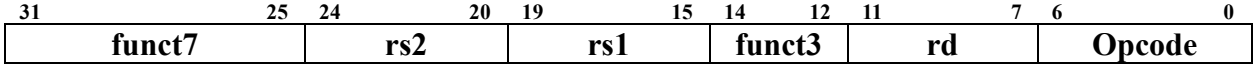

Figure 9. Basic RISC-V Instruction Set Encoding

As shown in Figure 9, the instruction word basically composed of 6 fields, including opcode, rd, funct3, rs1, rs2, and funct7. The first step in decoding is to parse these fields. The source register addresses rs1 and rs2 are used to fetch operands from the Register File, while rd indicates the destination register for the result.

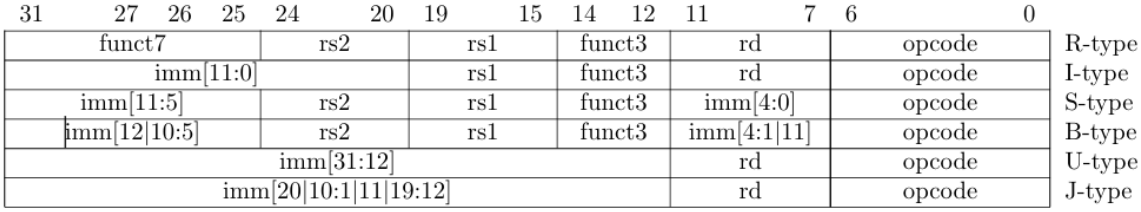

Figure 10. Instruction Types

Instructions with similar behavior, such as arithmetic, load, store, and branch operations, share common opcodes. Their specific operation is further identified using the funct3 and funct7 fields. These function fields, in conjunction with the opcode, determine the instruction type and are used to generate the appropriate immediate values and control signals required by subsequent stages of the pipeline.

In design, the decoder has it own classification of the instruction types and not based on common RISC-V instruction type classification, this is due to a more clear and convenience control signal generation, however this not a big deal as the instruction classification only matter and useful for the decoder internally. The instruction type is specied by the decoder in this design are:

Table 5. Instruction Clasification

Instr_type *(3-bit)*	**Instruction Type**	**Opcode** *(7-bit)*	**Description**
3'b000	NONE	7'bXXX_XXXX	Invalid or undefined instruction
3'b001	R_TYPE	7'b011_0011	Register-to-register operations
3'b010	I_TYPE	7'b001_0011	Immediate operations
3'b011	L_TYPE	7'b000_0011	Load instructions
3'b100	S_TYPE	7'b010_0011	Store instructions
3'b101	B_TYPE	7'b110_0011	Branch instrucitons
3'b110	SYS_TYPE	7'b111_0011	System/environment instructions
3'b111	OTHER		Other instructions with unique opcodes

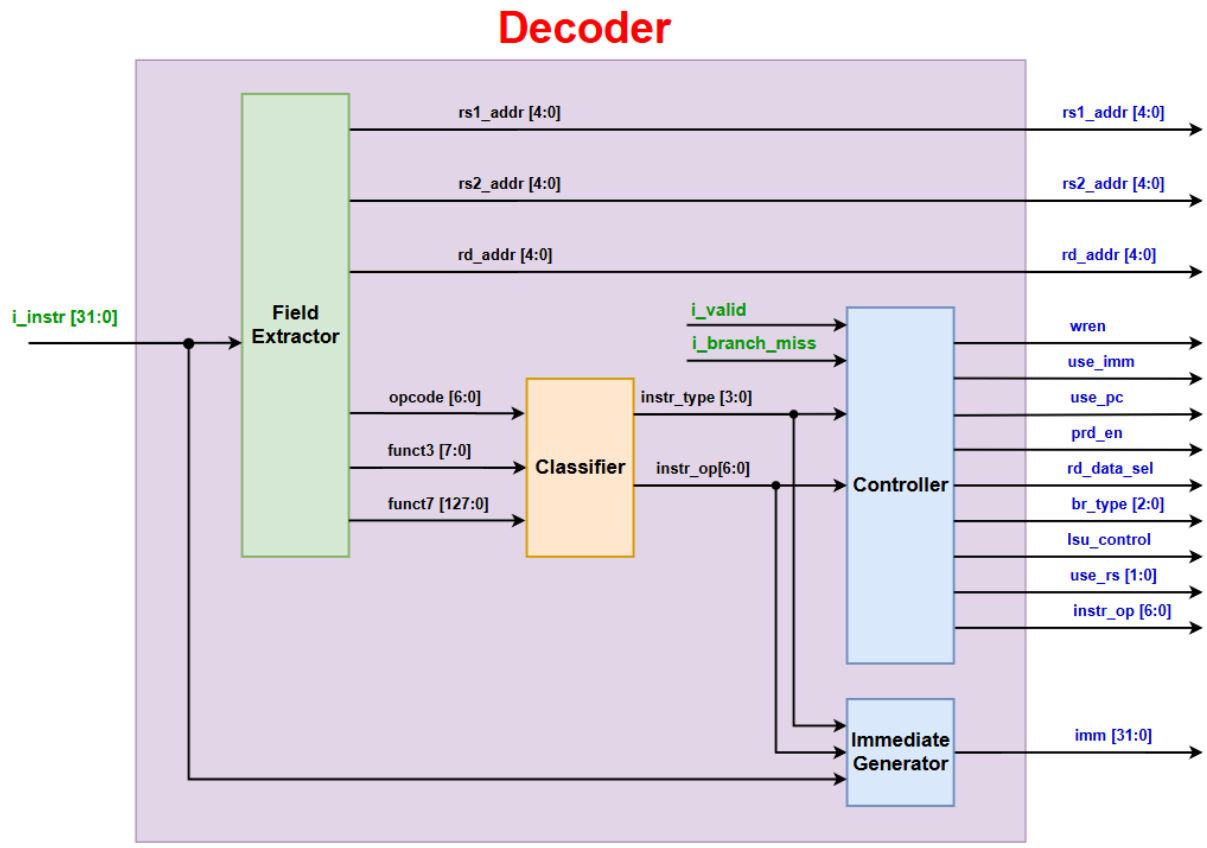

Figure 11. Decoder Block Diagram

(Each block in the diagram represents a functional section within the decoder, not separate modules. Each section handles a specific step in the decoding process)

The Immediate Generator takes the raw 32-bit instruction (instr[31:0]) as input and extracts the immediate value based on the instruction's format defined by the ISA. After that it uses the decoded instruction type and operation information to determine the correct immediate field and sign-extension logic required for execution.

The controller generates control signals based on the decoded instruction type and operation, enabling proper execution in later pipeline stages. Since it has direct access to the decoded fields and operations within the Decode stage, it is both efficient and logical to generate all necessary control signals for the ALU, memory access, and writeback operations at this point. Additionally, the controller produces operand selection signals to assist the arbitrator in routing data correctly to the Execute stage.

The controller receives two critical signals from the pipeline: ***stall*** and ***branch misprediction***. If either signal is asserted, the controller invalidates the current instruction by setting all control signals to inactive. This is particularly important during a stall, as the instruction currently in the Decode stage will remain in the IF/ID buffer and be re-decoded once the stall is cleared. Without invalidation, this would cause the same instruction to be executed twice. By clearing the control

signals during a stall, the pipeline ensures that invalid or duplicate instructions are safely bypassed and do not affect downstream stages.

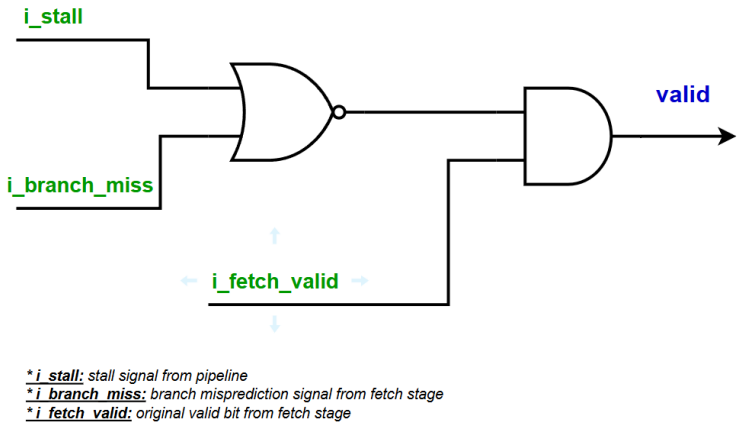

Figure 12. Valid bit generation by Decode stage

The input and output signals of the Decoder are specified as follows:

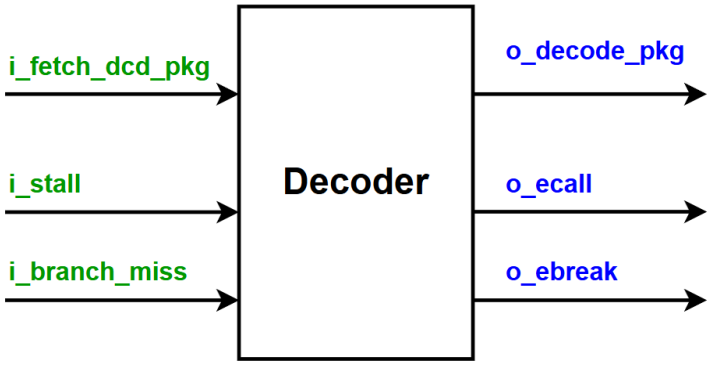

Figure 13, Input and output signals of the Decoder

Table 6. Description of Input and output signals of the Decoder

No.	Signal Name	Width	Type	Description
1	i_stall	1	Input	Indicates a stall request from the pipeline to halt decode processing
2	i_branch_miss	1	Input	Indicates a branch misprediction occurred, requiring instruction flush
3	i_fetch_dcd_pkg	66	Iutput	Data package received from the Fetch stage for decoding
4	o_decode_pkg	103	Output	Decoded instruction package sent to later stages
5	o_ecall	1	Output	High when a valid ECALL instruction is detected during decoding.
6	o_ebreak	2	Output	High when a valid EBREAK instruction is detected during decoding.

- ***Content of the "o_decode_pkg" data package:***

Table 7. Description of the o_decode_pkg data package

No.	Signal Name	Width	Description
1	instr_op	6	Encoded operation type of the decoded instruction (hold up to 64 types).
2	rs1_addr	5	Address of the first source register (rs1)
3	rs2_addr	5	Address of the second source register (rs2)
4	rd_addr	5	Address of the destination register (rd)
5	imm	32	Immediate value generated for instructions that use immediates.
6	pc	32	The PC of the decoded instuction
7	op_a_use_pc	1	High if the instruction uses PC as operand A.
8	op_b_use_im	1	High if the instruction uses the immediate value as operand B.
9	use_rs1	1	High if the instruction uses the value from the first Source Register
10	use_rs2	1	High if the instruction uses the value from the second Source Register
11	wren	1	Write-enable signal for writing back to the register file
12	rd_data_sel	2	Select signal for choosing the data source for write-back to the destination register
13	prd_en	1	High if the instruction is a branch and should update the Branch Target Buffer (BTB).
14	branch_type	3	Encoded type of branch instruction (BEQ, BNE, BLT, BGE, BGEU, BLTU).
15	predicted_instr	1	High if the instruction was predicted by the Branch Prediction Unit.
16	load_en	1	High if the instruction is a Load operation.
17	store_en	1	High if the instruction is a Store operation.
18	lsu_byte	1	High if the memory access is a byte operation.
19	lsu_haldword	1	High if the memory access is a halfword operation.
20	lsu_signed	1	High if the load operation is signed.
21	valid	1	High if the decoded instruction is valid and should proceed to execution.

The decoder needs to generate specific control signals to indicate whether an instruction uses the PC, an immediate value, or source register values from the Register File. This is essential

because the Arbitrator must determine the correct operand sources. For example, branch instructions use the PC as an operand, while others may use an immediate as operand B. Instructions like LUI or AUIPC do not use source registers or the PC as operands, so additional signals are required to inform the Arbitrator when source register values are not needed. In such cases, the Arbitrator can safely assign zero to the unused operands.

2.2. Hazard Detection

The Hazard Detection Unit identifies data dependencies that cannot be resolved by the Forwarding Unit or any pipeline stage beyond Decode. When such hazards are detected, it generates a stall signal to pause the pipeline, allowing time for the hazard to be resolved before continuing execution. Two key scenarios are currently handled:

- **Load-Use Hazard:** When a load instruction is in the Execute (EX) stage, the data it retrieves won't be available until two cycles later: one cycle to enter the Memory (MEM) stage and another to complete the memory read. If the next instruction depends on this data, it cannot proceed immediately, as forwarding is not yet possible. The pipeline must stall for one cycle to allow the data to become available and be forwarded correctly.

- **Read-After-Write Hazard (RAW):** Because the Forwarding Unit is located in the Execute stage instead of ID stage, it cannot resolve hazards where an instruction in the Decode (ID) stage requires data that is still being written back in the Writeback (WB) stage. In this case, the pipeline must stall for one cycle to allow the WB stage to complete the write to the Register File. Once the data is written, the stalled instruction can retrieve the correct value directly from the Register File.

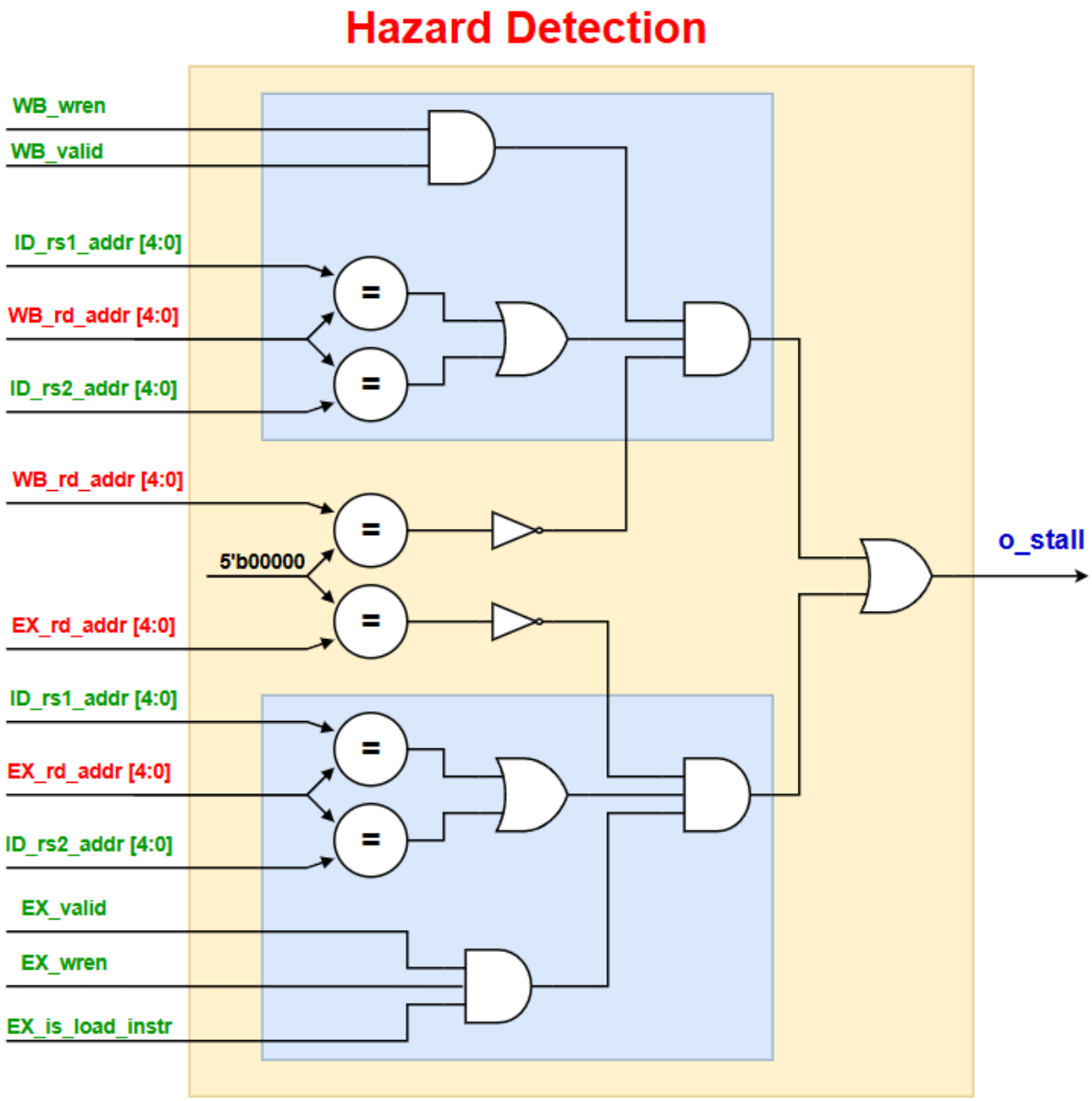

Figure 14. Stall signal generation from Hazard Detection Unit

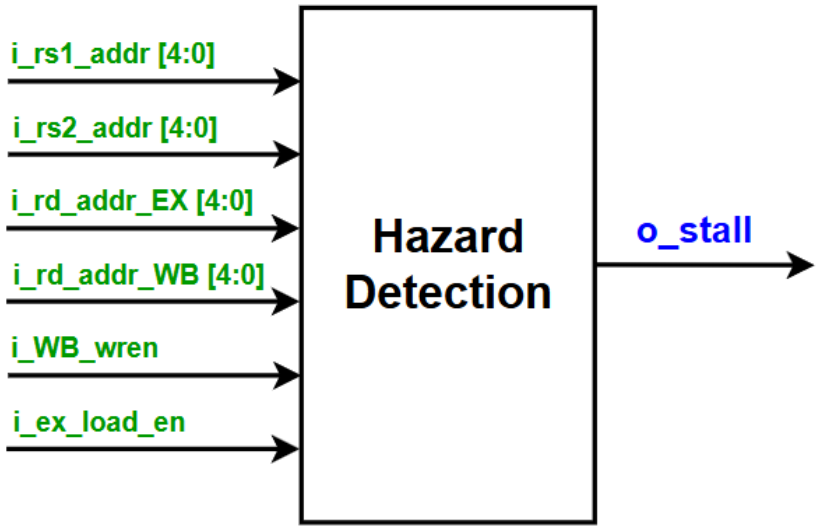

Figure 15. Input and Output signals of the Hazard Detection module

Table 8. Input and Output signals Description of Hazard Detection module

No.	Signal Name	Width	Type	Description
1	i_rs1_addr	5	Input	Source register 1 address from the Decoder.
2	i_rs2_addr	5	Input	Source register 2 address from the Decoder.
3	i_rd_addr_EX	5	Input	Destination register address of the instruction in EX stage.
4	i_rd_addr_WB	5	Input	Destination register address of the instruction in WB stage
5	i_WB_wren	1	Input	Write enable signal from the WB stage (AND-ed with the valid bit).
6	i_ex_load_en	1	Input	High to indicate the instruction in the EX stage is a Load instruction (AND-ed with the valid bit).
7	o_stall	1	Output	Stall signal generated to the pipeline when a hazard is detected.

3. Execute Stage

3.1. Forwarding Unit

The Forwarding Unit addresses basic Read-After-Write (RAW) hazards by comparing the destination registers of instructions in later pipeline stages with the source registers of the instruction currently in the Execute (EX) stage. If a match is detected, the unit generates control signals that inform the Arbitrator which data needs to be forwarded to the corresponding source register.

To efficiently handle this comparison, the Forwarding Unit is structured as a set of Forwarding Cells. Each cell is responsible for comparing the destination register of a specific pipeline stage (MEM, WB or any other location in the future development) with the current source registers (rs1 and rs2) in the EX stage. Based on the comparison results, each cell outputs a 2-bit signal indicating whether one or both source operands require forwarding.

This modular design makes the Forwarding Unit simple and highly scalable. As more stages or locations are added that may introduce data hazards (such as additional pipeline stages or multi-issue designs), new Forwarding Cells can be incorporated without redesigning the entire unit. Each cell independently handles comparisons for its respective stage, ensuring clean and extensible hazard resolution.

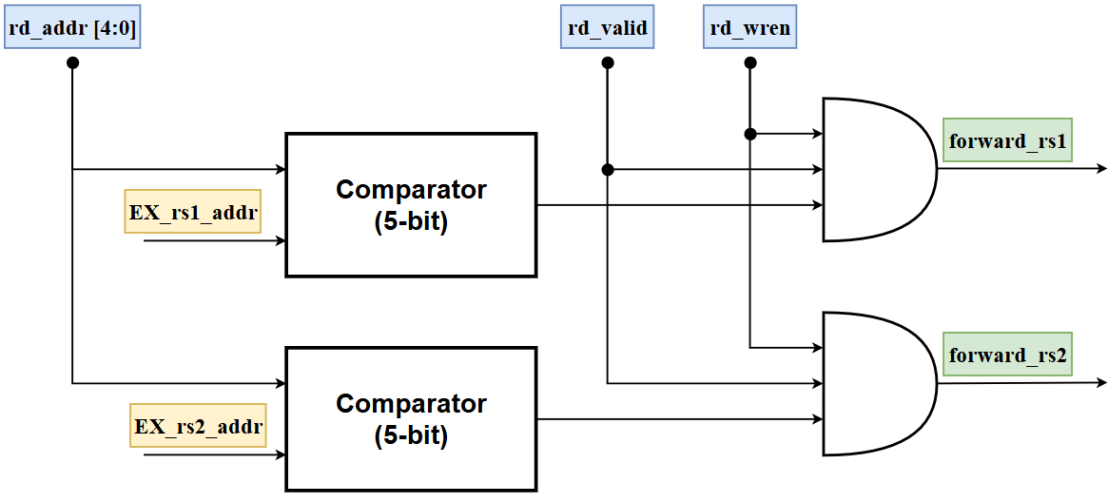

Figure 16. Forwarding Cell for compare Destination Register of a location with current Source Register

The Forwarding Unit prioritizes data from the pipeline stage closest to the Execute (EX) stage. When multiple destination registers in different stages match the same source register, the unit selects the nearest match and suppresses the others. This priority is enforced by using simple logic, such as AND gates, to mask the match signals from Forwarding Cells associated with later stages once an earlier match has been found.

3.2. Arbitrator

The Arbitrator is responsible for managing the execution stage by assigning operands to the appropriate functional unit. While the current RV32I processor only includes a single combinational ALU, the design anticipates future expansion to support additional functional units such as multipliers, dividers, or other multi-cycle execution units. In such cases, the Arbitrator would also need to monitor the status of each unit and determine when it is ready to accept a new instruction.

In the current simplified design for RV32I, the Arbitrator's main role is to select the correct operands for the ALU based on the decoded control signals. It uses the ***use_rs1*** and ***use_rs2*** signals from the decoder to check if the operands should come from the source registers. If instead the instruction uses the PC (use_pc) or an immediate value (use_imm), the Arbitrator selects those accordingly. If none of these control signals are set, it defaults the operand to zero.

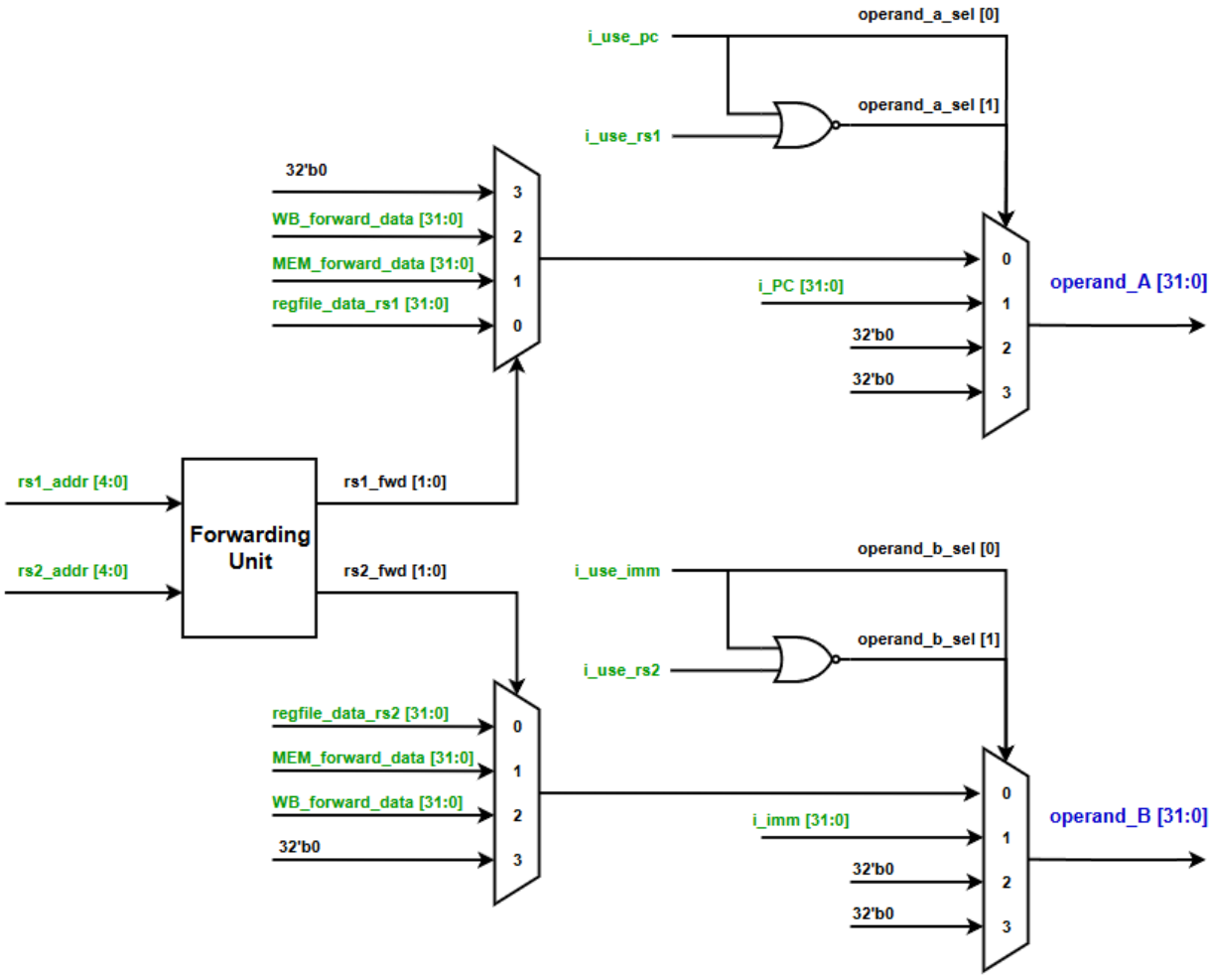

Figure 17. Arbitrating Logic for RV32I pipeline to ALU

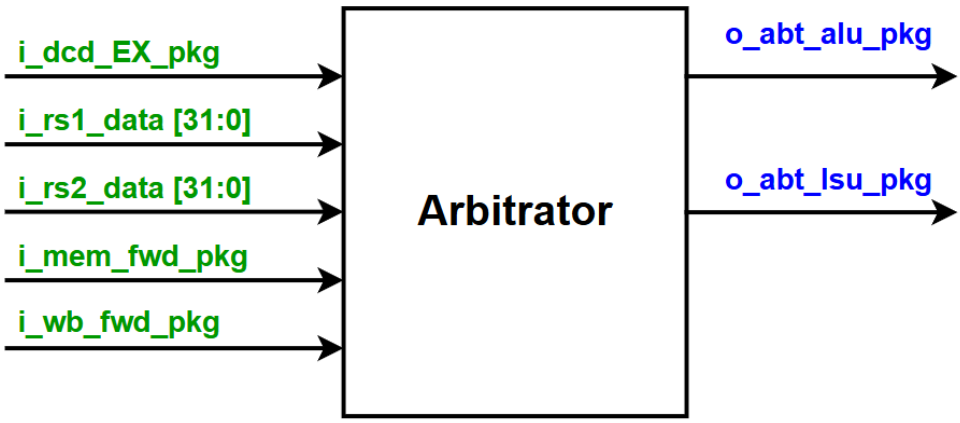

Figure 18. Input and Output signals of Arbitrator

Table 9. Input and Output signals description of Arbitrator

No.	Signal Name	Width	Type	Description
1	i_dcd_EX_pkg		Input	Data package from ID stage to EX stage
2	i_rs1_data	32	Input	Source register 1 data from theRegfile in ID stage
3	i_rs2_data	32	Input	Source register 2 data from theRegfile in ID stage
4	i_mem_fwd_pkg		Input	Forwarding data package from MEM stage
5	i_wb_fwd_pkg		Input	Forwarding data package from WB stage
6	o_abt_alu_pkg		Output	Data package from Aribtrator to ALU
7	o_abt_lsu_pkg		Output	Control signals passed from Abitrator to LSU

- ***Content of "i_mem_fwd_pkg" or "i_wb_fwd_pkg" data package:***

No.	Signal Name	Width	Description
1	Rd_data	32	Data to writeback to the Destination Register
2	Rd_addr	5	Address of the Destination Register
3	Fwd_allow	1	Indicates whether the current instruction allows data forwarding. This signal is high only when both the write-enable (wren) and valid signals of the instruction are asserted.

- ***Content of "o_abt_alu_pkg" data package:***

No.	Signal Name	Width	Description
1	instr_op	6	Encoded operation type of the instruction, used to determine ALU behavior.
2	oprand_a	32	Value of operand A, selected and forwarded by the Arbitrator.
3	operand_b	32	Value of operand B, selected and forwarded by the Arbitrator.
4	rs1_data	32	Data read from source register 1 (rs1) in Regfile
5	rs2_data	32	Data read from source register 2 (rs1) in Regfile
6	pc	32	PC of the current executing instruction
7	rd_addr	5	Address of the destination register where the result should be written.
8	wren	1	Write-enable signal for writing result back to the register file.
9	prd_en	1	Indicates that the instruction is a branch and allows the Branch Prediction Unit to update the BTB.
10	rd_data_sel	2	Select signal to choose the correct data source for write-back in the WB stage.
11	valid	1	Valid bit indicating whether the current instruction is active and should be executed.

- ***Content of "o_abt_lsu_pkg" data package:***

No.	Signal Name	Width	Description
1	Store_data	32	Data to be written to memory (the value from register rs2).
2	Load_en	1	High if the current instruction is a load operation.
3	Store_en	1	High if the current instruction is a store operation.
4	Lsu_byte	1	Indicates that the memory access is a Byte operation.
5	Lsu_halfword	1	Indicates that the memory access is a Halfword operation.
6	Lsu_sgined	1	Indicates that the load operation is signed.
7	valid	1	High if the memory access instruction is valid and should be executed.

3.3. Arithmetic and Logical Unit (ALU)

The ALU in this design performs two key functions: executing arithmetic and logical operations, and evaluating branch conditions to determine whether a branch should be taken.

For arithmetic and logical instructions, the required operands are selected by the Arbitrator and forwarded to the ALU. Inside the ALU, these operands are processed by dedicated sub-blocks, each responsible for a specific operation such as addition, subtraction, bitwise logic, comparisons, or shifting. The outputs from these sub-blocks are then routed to a multiplexer.

This multiplexer selects the appropriate result based on the op_sel control signal, which is generated by the Arbitration Unit. The op_sel signal reflects the operation type decoded from the instruction in the previous pipeline stage, ensuring that the correct computation is performed for each instruction. The Adder/Subtractor unit within the ALU is also responsible for calculating the target address of branch instructions and the effective address for memory access operations. The operands required for these calculations are selected and provided by the Arbitrator.

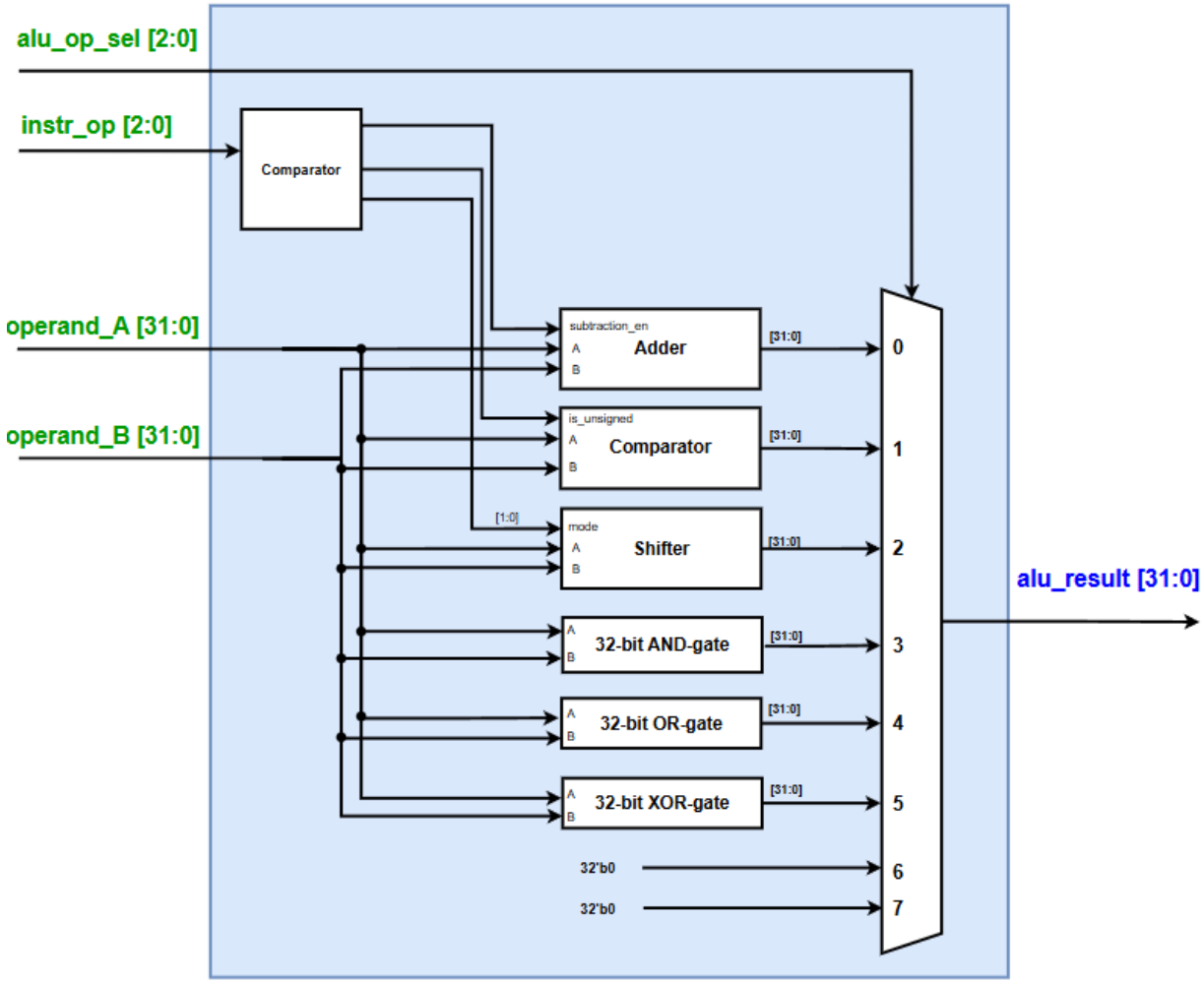

Figure 19. Block diagram of ALU's Arithmetic and Logical instruction execution

Table 10. ALU operation selection based on alu_op_sel [2:0] signals

Alu_op_sel	Selected functional block	Description
3'b000	Adder	Performs addition and subtraction operations. *Supports: ADD, SUB, ADDI*
3'b001	Comparator	32-bit comparator. Outputs boolean result extended to 32 bits. *Supports: SLT, SLTU, SLTI, SLTIU*
3'b010	Shifter	Executes logical and arithmetic shift operations (left/right). *Supports: SRA, SRAI, SRL, SRLI, SLL, SLLI*
3'b011	32-bit AND gate	Performs bitwise logical AND. *Supports: AND, ANDI*
3'b100	32-bit OR gate	Performs bitwise logical OR. *Supports: OR, ORI*
3'b101	32-bit XOR gate	Performs bitwise logical XOR. *Supports: XOR, XORI*
3'b110	Reserved	Not used. Output is fixed to 0.
3'b111	Reserved	Not used. Output is fixed to 0.

- **Shifter Module**

The right shift operation is controlled by a 5-bit signal, which refered as "shift_amount". Each bit in this shift_amount signal corresponds to a specific shift amount: 1, 2, 4, 8, and 16. This means that we can shift the data by any combination of these amounts by processing each bit of the shift_amount signal.

The right shifting process is divided into 5 stages, each responsible for shifting the data according to one bit of the "shift_amount" signal:

- ***First Stage:*** *Checks the least significant bit of shift_amount (bit 0). If it is 1, the data is right-shifted by 1 position.*
- ***Second Stage:*** *Checks the next bit of shift_amount (bit 1). If it is 1, the data is right-shifted by 2 positions.*
- ***Third Stage:*** *Checks the next bit of shift_amount (bit 2). If it is 1, the data is right-shifted by 4 positions.*
- ***Fourth Stage:*** *Checks the next bit of shift_amount (bit 3). If it is 1, the data is right-shifted by 8 positions.*
- ***Fifth Stage:*** *Checks MSB of shift_amount (bit 4). If it is 1, the data is right-shifted by 16 positions.*

Each stage uses a 2x1 multiplexer to determine whether to pass the data through unchanged or to shift it by the specified amount for that stage. The output of each stage is then fed into the next stage. The output of the last stage is the output of the data with right shift amount specified by the shift_amount signal.

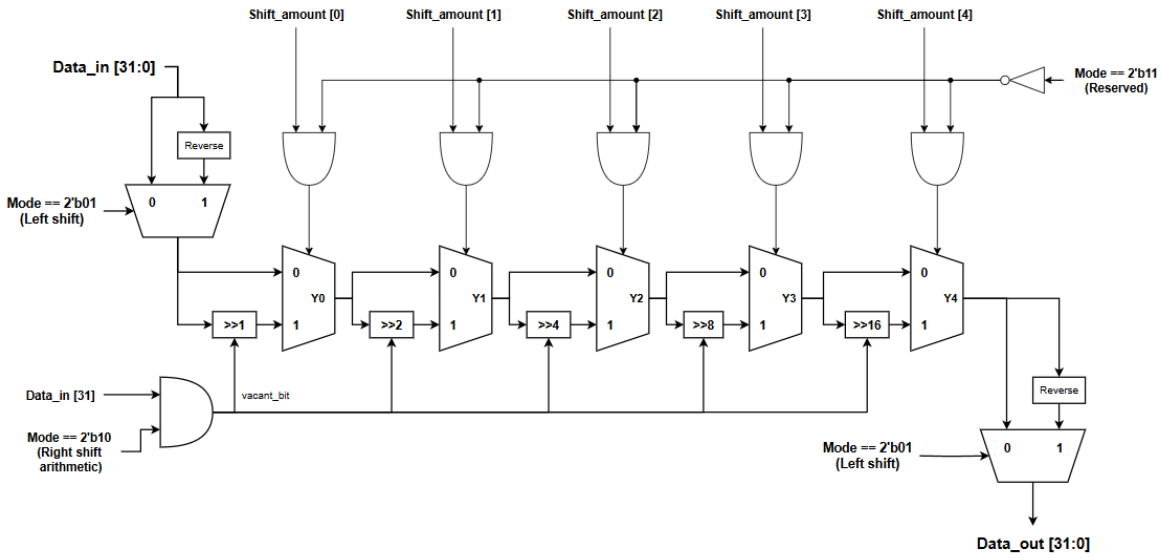

Figure 20. Diagram of shifter

The small shifter module in each stage can be implemented by directly connecting each bit of the input data to its new position, corresponding to the specified shift amount. This method involves simple wiring and does not require any logic gates or other electrical components.

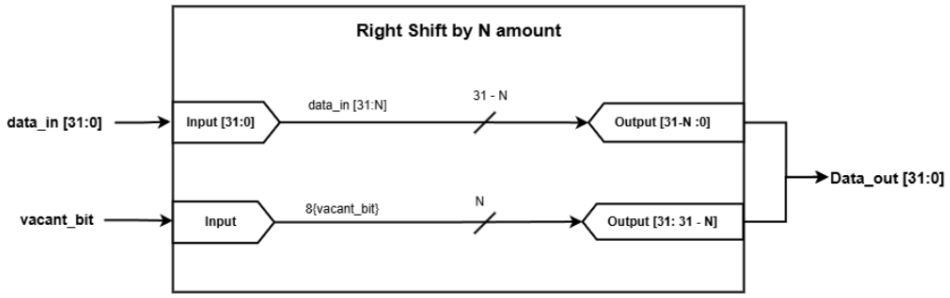

Figure 21. Right shifter module by an defined N amount

The vacant bit is determined externally to perform a right shift arithmetic operation. The right shift arithmetic mode is selected, the vacant bit is set to the sign bit of the binary number represented by data_in, which preserves the sign of the original number by filling the leftmost positions with the sign bit.

$$Vacant\_bit \; = \; Right\_shift\_arithmetic \quad \boldsymbol{AND} \quad sign\_bit$$

When the left shift logic mode is selected, we first reversing the input data, performing a right shift logic operation, and then reversing the data back to its original order.

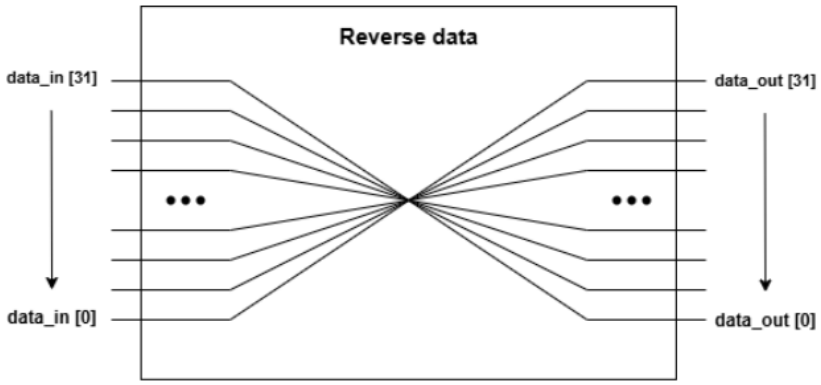

Figure 22. Reverse data modulr for left shifting

The reverse data circuit also involves simple wiring and no logic gates or other electrical components. We utilize a 32-bit multiplexer to determine whether data should be reversed in left shifting mode

A 2-bit control signal is used for selecting the shift operation mode, as the design supports three functional modes: logical right shift, logical left shift, and arithmetic right shift. The remaining mode, represented by mode == 2'b11, is reserved. When this mode is selected, the circuit performs no shift operation, effectively passing the input data unchanged. This behavior can be implemented by gating the shift_amount input with AND gates controlled by the mode select logic, as shown in the figure below.

$$Sel_N = shift\_amount \quad \boldsymbol{AND} \quad (mode == 2'b11)$$

- **Comparator Module**

The 32-bit comparator in the ALU is a magnitude comparator capable of determining whether one 32-bit operand is greater than, equal to, or less than another. This comparator is primarily used for branch condition evaluation, helping to decide whether a branch instruction should be taken based on the comparison result.

The 32-bit magnitude comparator is implemented by cascading smaller 4-bit magnitude comparator modules, as illustrated in the block diagram below. This modular approach simplifies design and enables scalable, bitwise comparison logic.

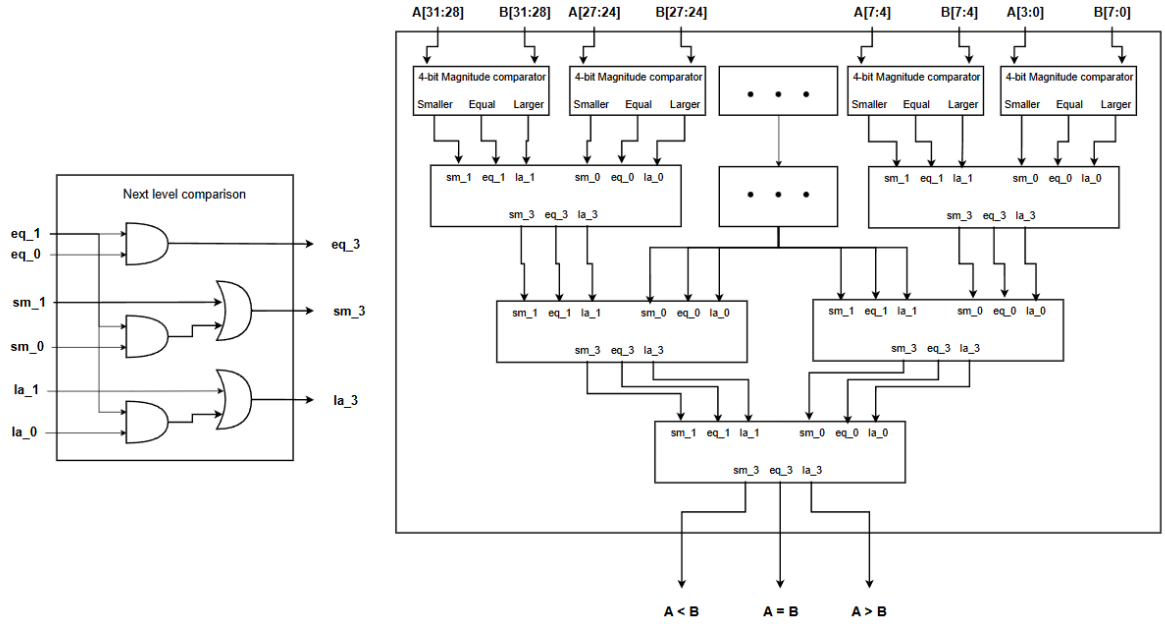

Figure 23. Diagram of 32-bit magnitude comparator

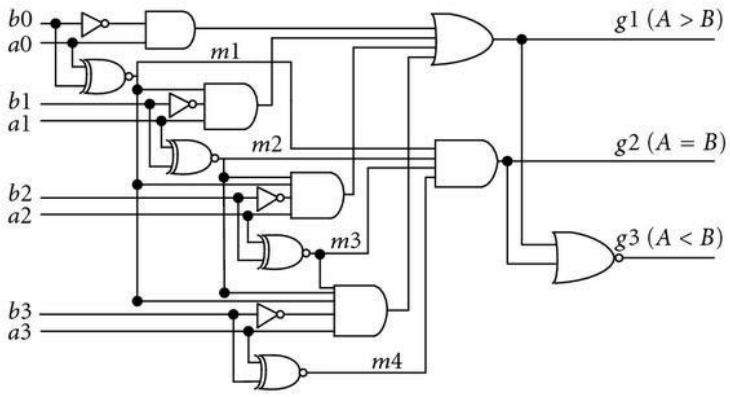

Figure 24. Diagram of a 4-bit magnitude comparator

The comparator configuration shown above supports only unsigned comparisons. However, the RV32I instruction set includes instructions that require signed comparisons. To handle this, we need to adapt the unsigned comparator to perform signed comparisons as well.

For signed comparisons, if both operands have the same sign (both positive or both negative), we can compare them as if they were unsigned. However, if the operands have different signs, the comparison result is straightforward: the operand with the positive sign is always greater than the one with the negative sign.

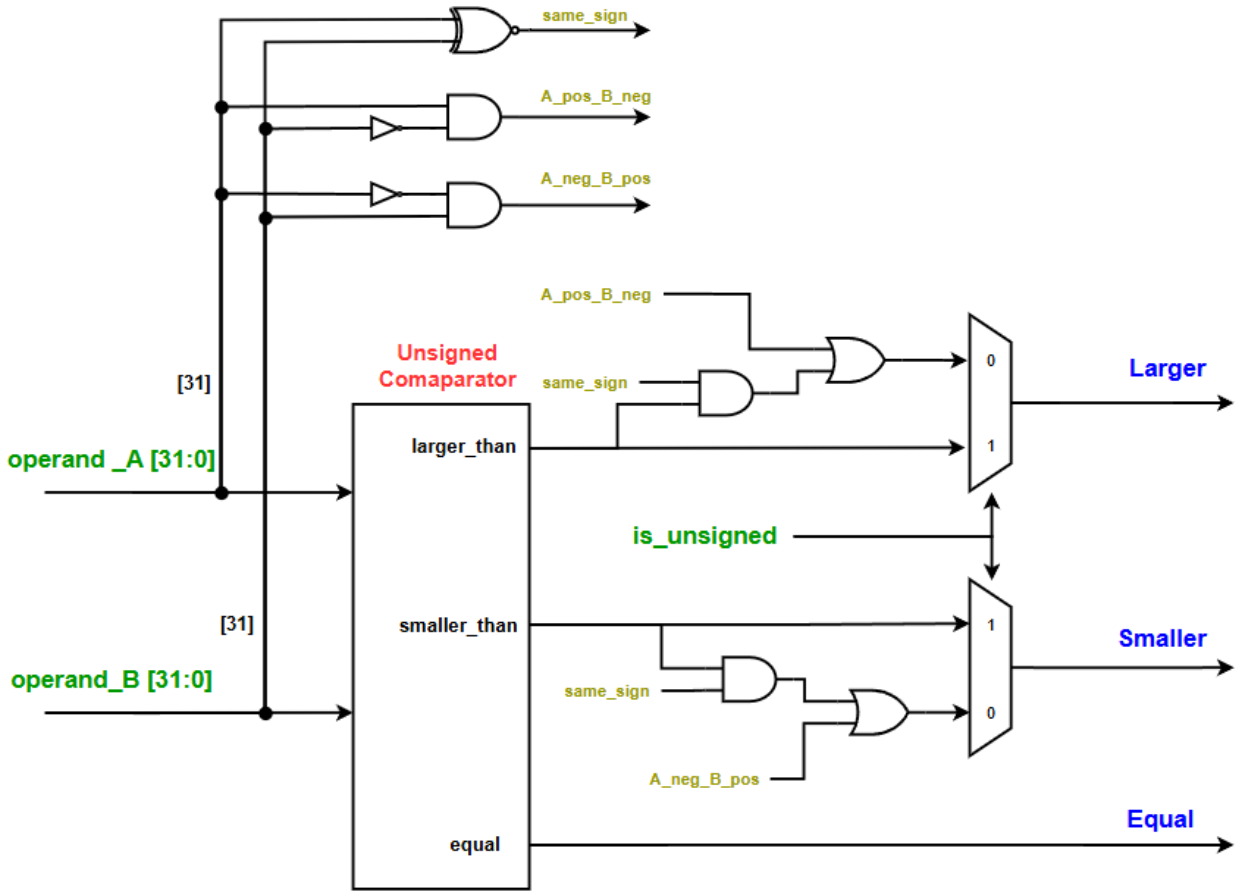

Figure 25. Signed comparator constrcuted from Unsigned comparator

- **Branch Instruction Handling:**

To handle branch instructions, the processor compares the values of the two source registers to determine whether the branch condition is met, based on the specific branch type (e.g., BEQ, BNE, BLT). If the condition is satisfied, the branch is taken. In contrast, for jump instructions like JAL and JALR, the control transfer is unconditional—the jump is always taken.

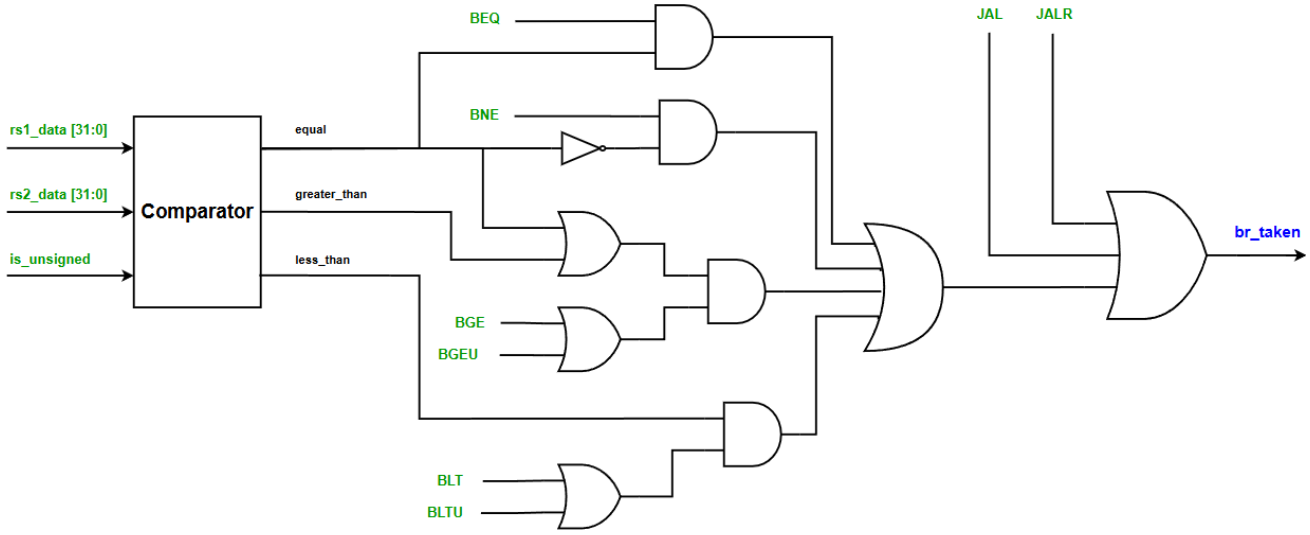

Figure 26. Branch instruction execution in ALU

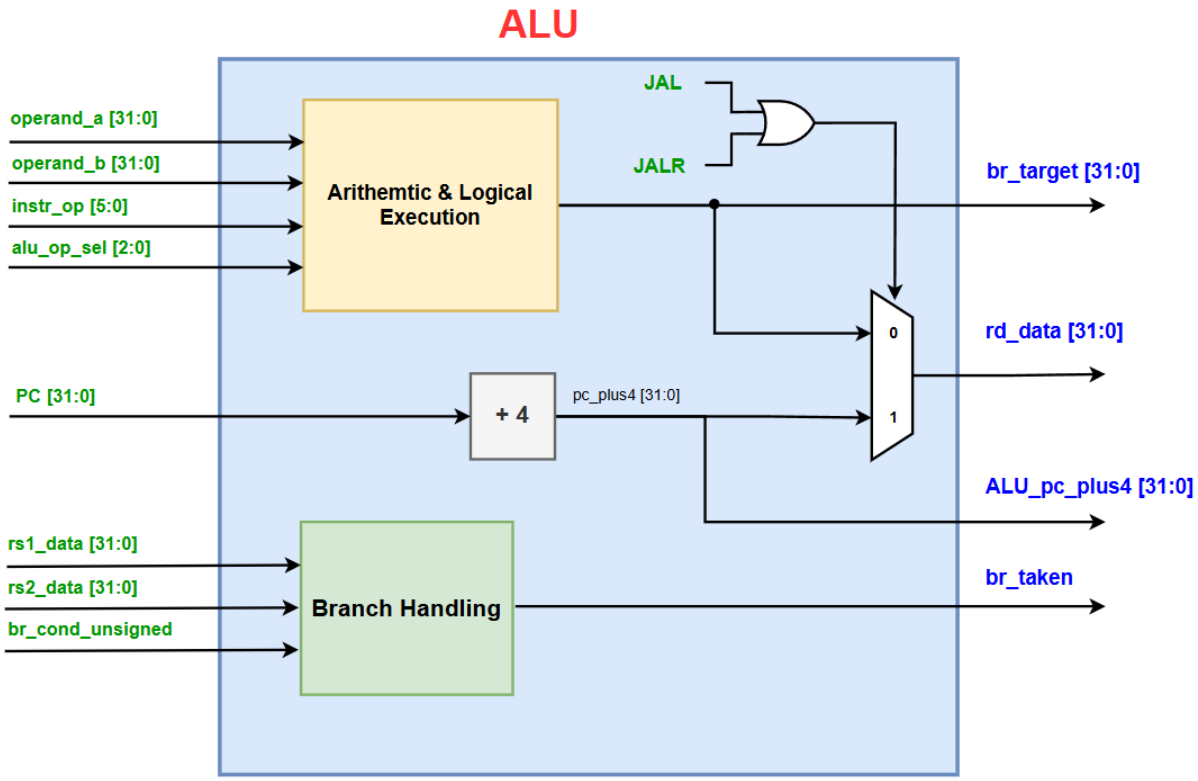

Figure 27. Block Diagram of ALU in EX stage

- **The input and output signals of the ALU:**

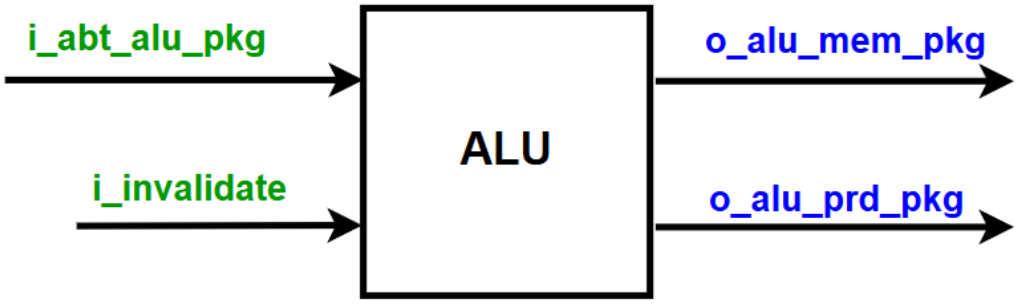

Figure 28. The input and output signals of ALU

Table 11. Input and Output signals description of ALU

No.	Signal Name	Width	Type	Description
1	i_invalidate	1	Input	Signal from the pipeline used to invalidate the instruction currently executing in the EX stage.
2	i_abt_alu_pkg		Input	Data package sent from the Arbitrator to the ALU containing operands and control signals.
3	o_alu_mem_pkg		Output	Data package from the ALU forwarded to the MEM stage for further processing.
4	o_alu_prd_pkg		Output	Data package sent from the ALU to the Branch Prediction Unit in the Fetch stage for updating prediction information.

- ***Content of "o_alu_mem_pkg" data package:***

No.	Signal Name	Width	Description
1	rd_data	32	The result data to be written back to the destination register (also holds the effective address for Memory Access stage).
2	rd_addr	5	Address of the destination register for the current instruction.
3	rd_data_sel	2	Select signal used to determine the source of data to be written back during the Writeback (WB) stage.
4	wren	1	Write-enable signal that authorizes writing to the register file.
5	valid	1	Indicates whether the current instruction is valid and should proceed to execution and writeback.

- ***Content of "o_alu_prd_pkg" data package:***

No.	Signal Name	Width	Description
1	br_update_en	1	Indicates that a branch instruction is currently being executed in the ALU
2	br_update_pc	32	PC of the executed Branch instruction
3	br_pc_plus4	32	PC + 4 of the executed Branch instruction
4	br_target	32	Target address computed by the executed branch instruction.
5	br_taken	1	Indicates whether the branch is reuslt in taken.
6	br_valid	1	The executed Branch instruction is Valid
7	br_already_predicted	1	High if the branch instruction was previously predicted, used for misprediction detection.s

4. Memory Access Stage

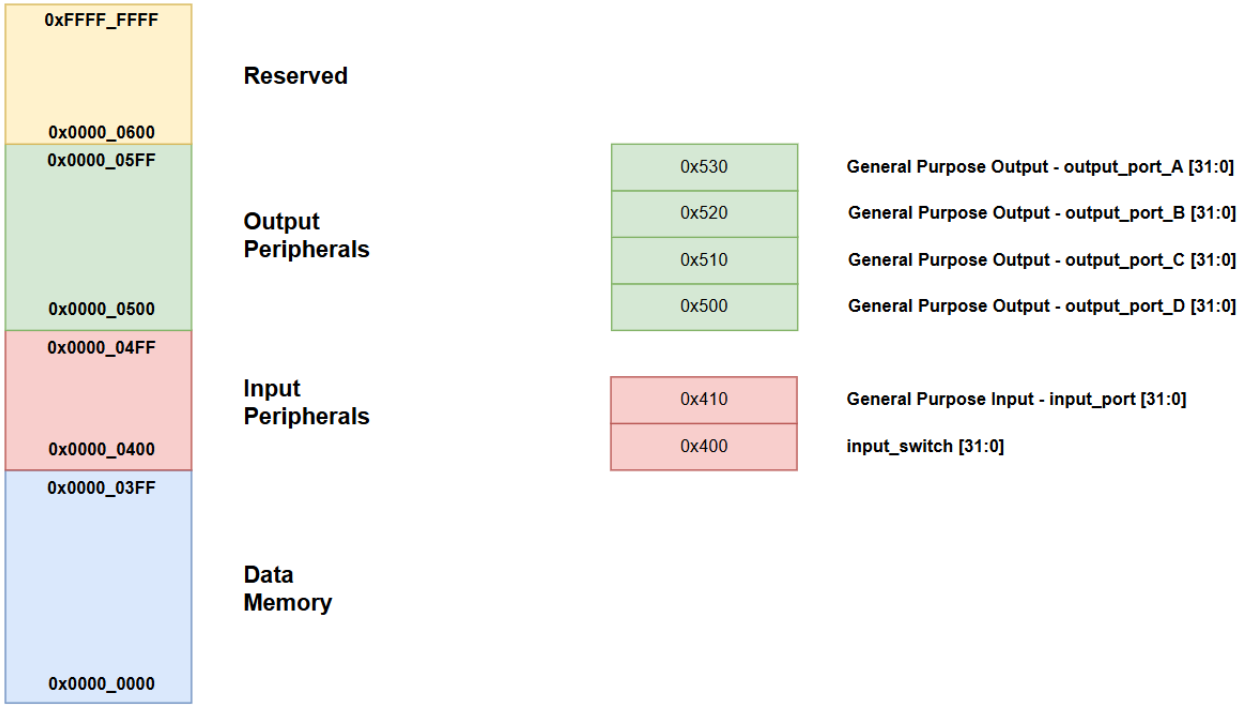

Figure 29. Simple Memory Map for the currentl Rv32I processor (Unofficial)

5. Writeback stage