VLSI DESIGN INTERNSHIP

RISC-V ISA & RV32I RTL CODING

A.NARMATHA
22BEC1280

aceic
Accelerate Collaborate Excel

MAVEN SILICON
Centre of Excellence in VLSI

CERTIFICATE OF COMPLETION

This is to certify that Mr./Ms. **Narmatha A (22BEC1280)** *has successfully completed VLSI Design Internship Program from* **05-June-2024** *to* **13-July-2024**

During the internship program with us, he/she worked on **RISC-V ISA & RV32I RTL Design** *project.*

Date: 02/08/2024
Place: Bengaluru

MSUID: MS/V-DI/2024-25/487

SIVAKUMAR P R
FOUNDER & CEO, MAVEN SILICON

MAVEN SILICON
Centre of Excellence in VLSI

MAVEN SILICON GRADE REPORT

Grading Index

>90%	>70%	>50%	>30%	>10%	<10%
"O"	"A+"	"A"	"B+"	"B"	"C"

Name
Narmatha A

College Registration Number
22BEC1280

Course Name
VLSI Design Internship Program

Project Name
RISC-V ISA & RV32I RTL Design

Date
July 2024

Project Test Score (in %)	70
Project Submission Report Score (in %)	60
Grade	A

ABOUT MAVEN SILICON

01

Maven Silicon is a distinguished company in the domain of VLSI (Very Large Scale Integration) design and semiconductor technology. It has established itself as a leading institution that offers a wide array of services and training programs related to VLSI design and verification.

02

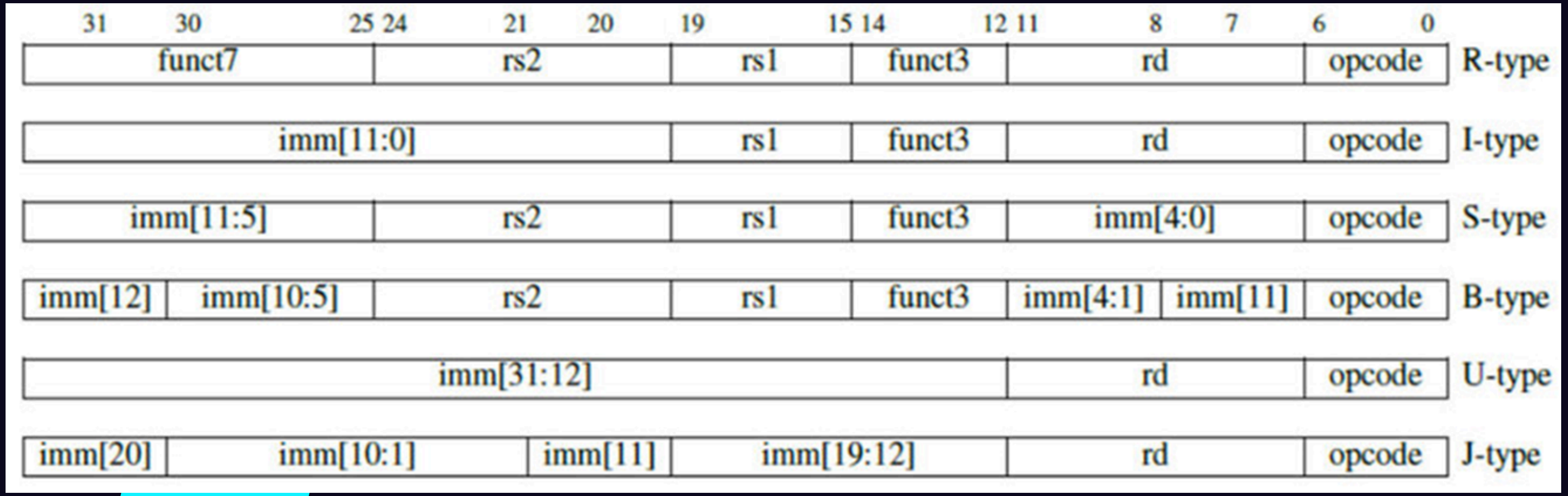
•One of the standout features of Maven Silicon is its comprehensive VLSI training programs. These programs cover a wide spectrum of topics, from the fundamentals of digital design to advanced concepts such as ASIC and FPGA design. This ensures that students are not only taught theoretical concepts but also gain insights into the latest industry trends and best practices

OBJECTIVE OF THE TRAINING

The VLSI Design Methodologies Course by MAVEN SILICON provided an overview of VLSI and explains various VLSI technology.Following project assignment, each student received project-specific training. In this case, detailed concepts of

- Instruction Set Architecture (ISA): Understand the fundamental principles of how instructions are encoded, decoded, and executed.
- Pipeline Design: Learn how to design efficient pipelines to improve processor performance.
- Memory Hierarchy: Explore the concepts of cache memory, virtual memory, and memory management units.
- Control Unit Design: Grasp the design of control units to generate control signals for the data path.
- Data Path Design: Understand the components of the data path, such as registers, ALU, and memory interface.

RISC V ISA INSTRUCTION FORMAT



RISC V R-TYPE I TYPE

name	31:25	24:20	19:15	14:12	11:7	opcode	Description
add	0000000	rs2	rs1	000	rd	0110011	(rd) = (rs1) + (rs2)
sub	0100000	rs2	rs1	000	rd	0110011	(rd) = (rs1) - (rs2)
slt	0000000	rs2	rs1	010	rd	0110011	(rd) = signed(rs1) < signed(rs2) ? 1 : 0
sltu	0000000	rs2	rs1	011	rd	0110011	(rd) = unsigned(rs1) < unsigned(rs2) ? 1 : 0
xor	0000000	rs2	rs1	100	rd	0110011	(rd) = (rs1) ^ (rs2)
or	0000000	rs2	rs1	110	rd	0110011	(rd) = (rs1) | (rs2)
and	0000000	rs2	rs1	111	rd	0110011	(rd) = (rs1) & (rs2)
sll	0000000	rs2	rs1	001	rd	0110011	(rd) = (rs1) << unsigned(rs2[4:0])
srl	0000000	rs2	rs1	101	rd	0110011	(rd) = (rs1) >> unsigned(rs2[4:0])
sra	0100000	rs2	rs1	101	rd	0110011	(rd) = signed(rs1) >>> unsigned(rs2[4:0])

name	31:20	24:20	19:15	14:12	11:7	opcode	Description
addi	imm[11:0]		rs1	000	rd	0010011	(rd) = (rs1) + signed(imm)
slti	imm[11:0]		rs1	010	rd	0010011	(rd) = signed(rs1) < signed(imm) ? 1 : 0
stliu	imm[11:0]		rs1	011	rd	0010011	(rd) = unsigned(rs1) < unsigned(imm) ? 1 : 0
xori	imm[11:0]		rs1	100	rd	0010011	(rd) = (rs1) ^ signed(imm)
ori	imm[11:0]		rs1	110	rd	0010011	(rd) = (rs1) | signed(imm)
andi	imm[11:0]		rs1	111	rd	0010011	(rd) = (rs1) & signed(imm)
slli	0000000	imm[4:0]	rs1	001	rd	0010011	(rd) = (rs1) << unsigned(imm)
srli	0000000	imm[4:0]	rs1	101	rd	0010011	(rd) = (rs1) >> unsigned(imm)
srai	0100000	imm[4:0]	rs1	101	rd	0010011	(rd) = signed(rs1) >>> unsigned(imm)

RISC V L-TYPE S-TYPE

Load Instructions (I-Type)

name	31:20	24:20	19:15	14:12	11:7	opcode	Description
lb	imm[11:0]		rs1	000	rd	0000011	(rd) = signed(mem_byte[(rs1) + signed(imm)])
lh	imm[11:0]		rs1	001	rd	0000011	(rd) = signed(mem_hw[(rs1) + signed(imm)])
lw	imm[11:0]		rs1	010	rd	0000011	(rd) = mem[(rs1) + signed(imm)]
lbu	imm[11:0]		rs1	100	rd	0000011	(rd) = unsigned(mem_byte[(rs1) + signed(imm)])
lhu	imm[11:0]		rs1	101	rd	0000011	(rd) = signed(mem_hw[(rs1) + signed(imm)])

Store Instructions (S-Type)

name	31:20	24:20	19:15	14:12	11:7	opcode	Description
sb	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	mem_byte[(rs1) + signed(imm)] = (rs2)[7:0]
sh	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	mem_hw[(rs1) + signed(imm)] = (rs2)[15:0]
sw	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	mem[(rs1) + signed(imm)] = (rs2)

RISC V B-TYPE J-TYPE

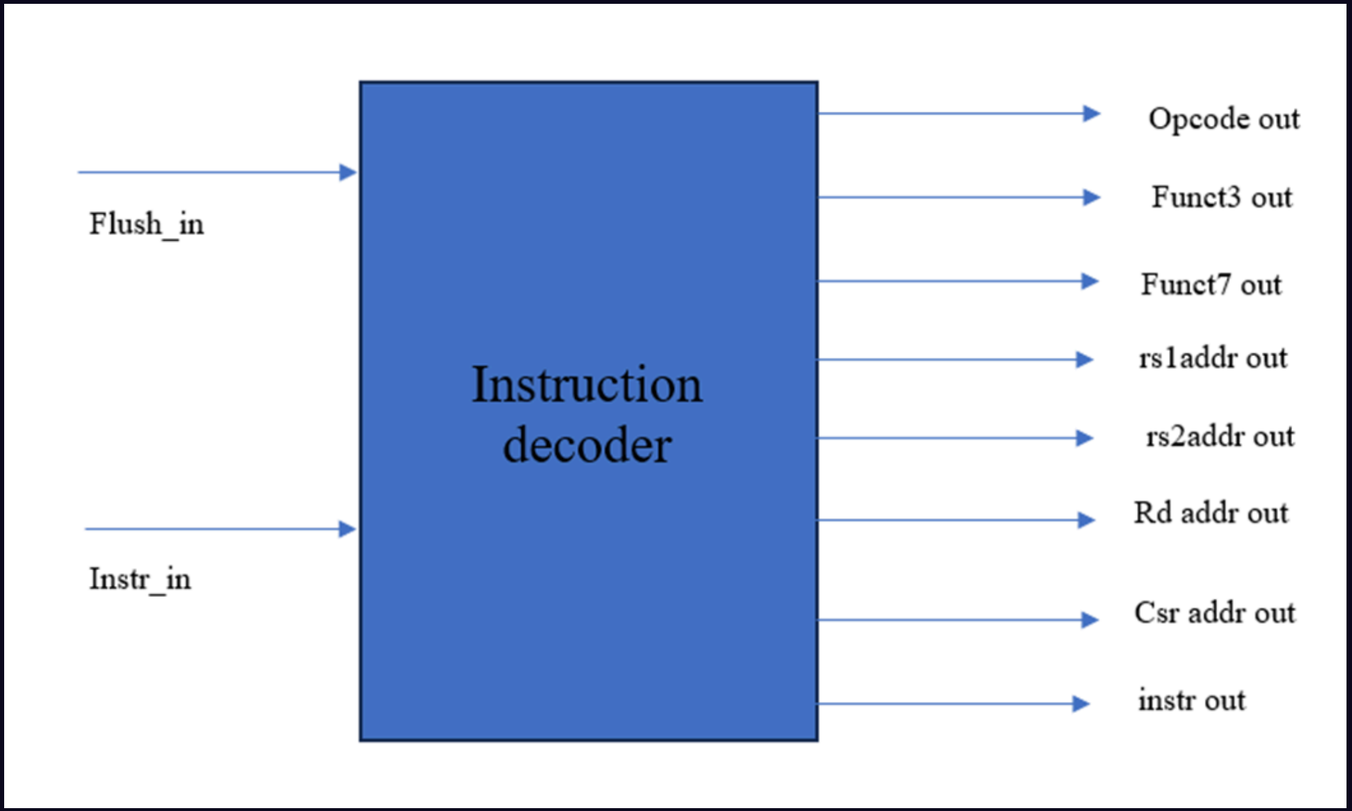
Branch Instructions (B-Type)

name	31:25	24:20	19:15	14:12	11:7	opcode	Description
beq	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	pc = pc + ((rs1) == (rs2) ? signed(imm) : 4)
bne	imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	pc = pc + ((rs1) != (rs2) ? signed(imm) : 4)
blt	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	pc = pc + ((rs1) < (rs2) ? signed(imm) : 4)
bge	imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	pc = pc + ((rs1) >= (rs2) ? signed(imm) : 4)
bltu	imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	pc = pc + (unsigned((rs1) < (rs2)) ? signed(imm) : 4)
bgeu	imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	pc = pc + (unsigned((rs1) >= (rs2)) ? signed(imm) : 4)

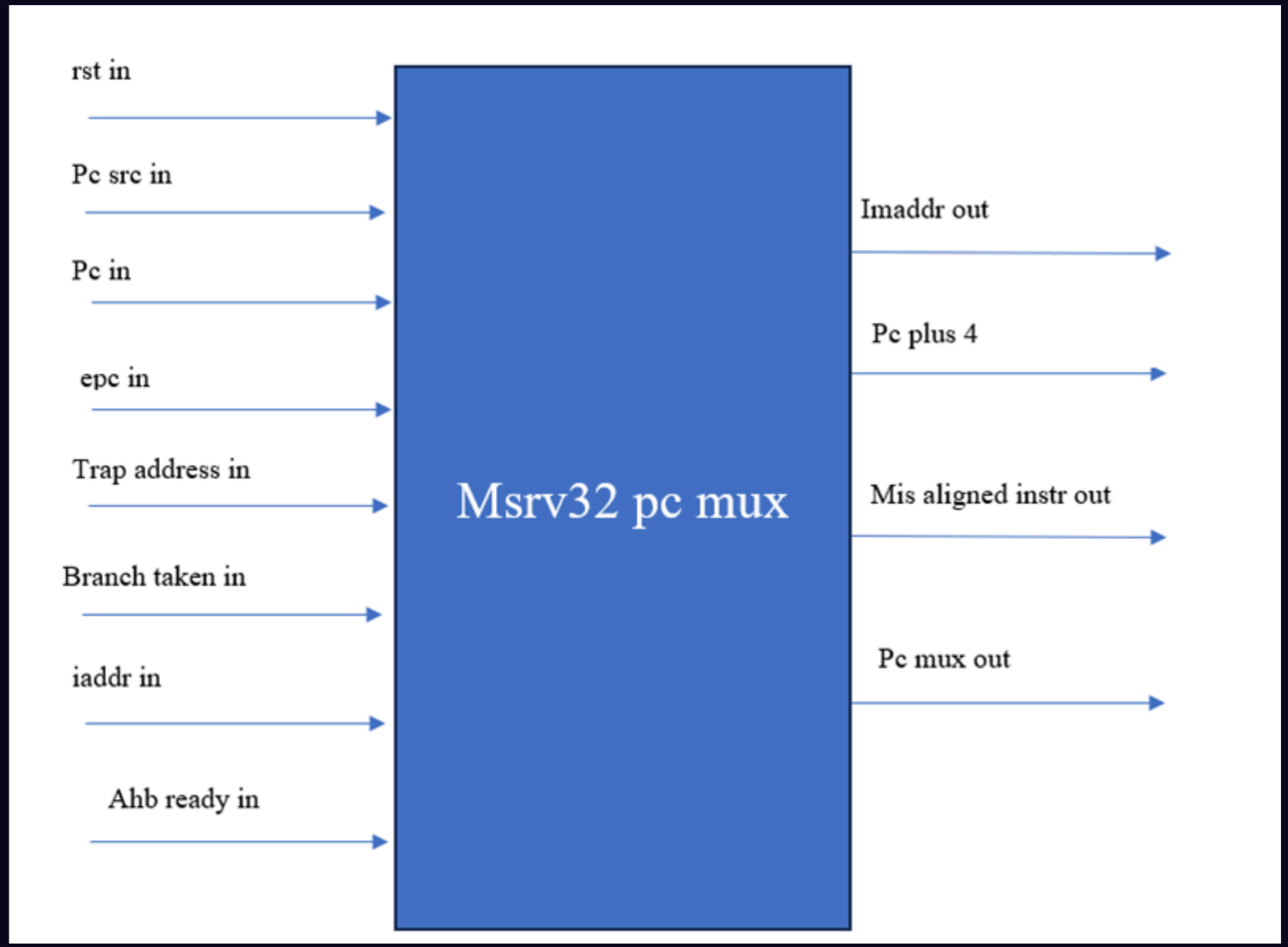
Jump and Link Instructions (J-Type, I-Type)

name	31:25	19:15	14:12	11:7	opcode	Description
jal	imm[20|10:1|11|19:12]			rd	1101111	(rd) = pc + 4, pc = pc + signed(imm)
jalr	imm[11:0]	rs1	000	rd	1100111	(rd) = pc + 4, pc = ((rs1)+ signed (imm)) & ~32'h1

INSTRUCTION DECODER



RISC V PC MUX


rst in
Pc src in
Pc in
epc in
Trap address in
Branch taken in
iaddr in
Ahb ready in
Msrv32 pc mux
Imaddr out
Pc plus 4
Mis aligned instr out
Pc mux out

PC MUX

Internal Signals

- **i_addr** (reg, 32 bits): The internal register for the instruction address.
- **next_pc** (wire, 32 bits): The calculated next PC value based on branching.

Misaligned Instruction Detection:

- omisaligned_instr_out is set if next_pc is misaligned (i.e., if the least significant bit is 1) and branch_taken_in is high.

PC Increment:

- opc_plus_4_out is calculated as the current pc_in plus 4, which is the address of the next sequential instruction.

Next PC Calculation:

- onext_pc is calculated as:
- {iaddr_in, 1'b0} if branch_taken_in is high (indicating a branch),
- pc_plus_4_out otherwise.

INSTRUCTION DECODER

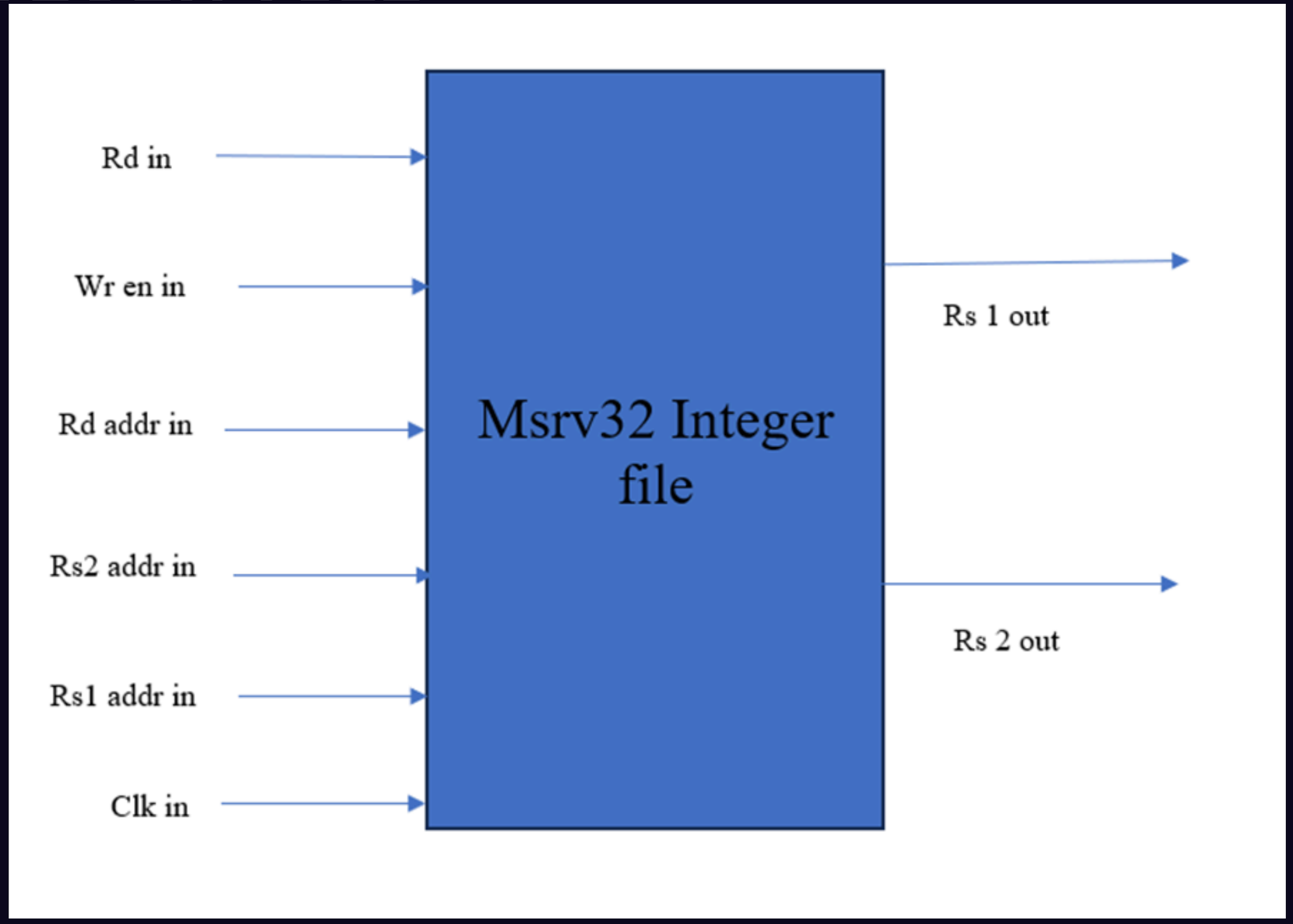
PC Multiplexer:

- The pc_mux_out value is selected based on pc_src_in:
- 2'b00: pc_mux_out is set to BOOT_ADDRESS.
- 2'b01: pc_mux_out is set to epc_in.
- 2'b10: pc_mux_out is set to trap_address_in.
- 2'b11: pc_mux_out is set to next_pc.

Instruction Address Update:

- The i_addr register is updated:
- To BOOT_ADDRESS if rst_in is high (indicating a reset).
- To pc_mux_out if ahb_ready_in is high (indicating that the AHB is ready).

RISC V INTEGER FILE



INTEGER FILE

Register File:

- The module uses an array of 32 registers (reg_file) to store 32-bit integer values. Each register is identified by its address (0 to 31).

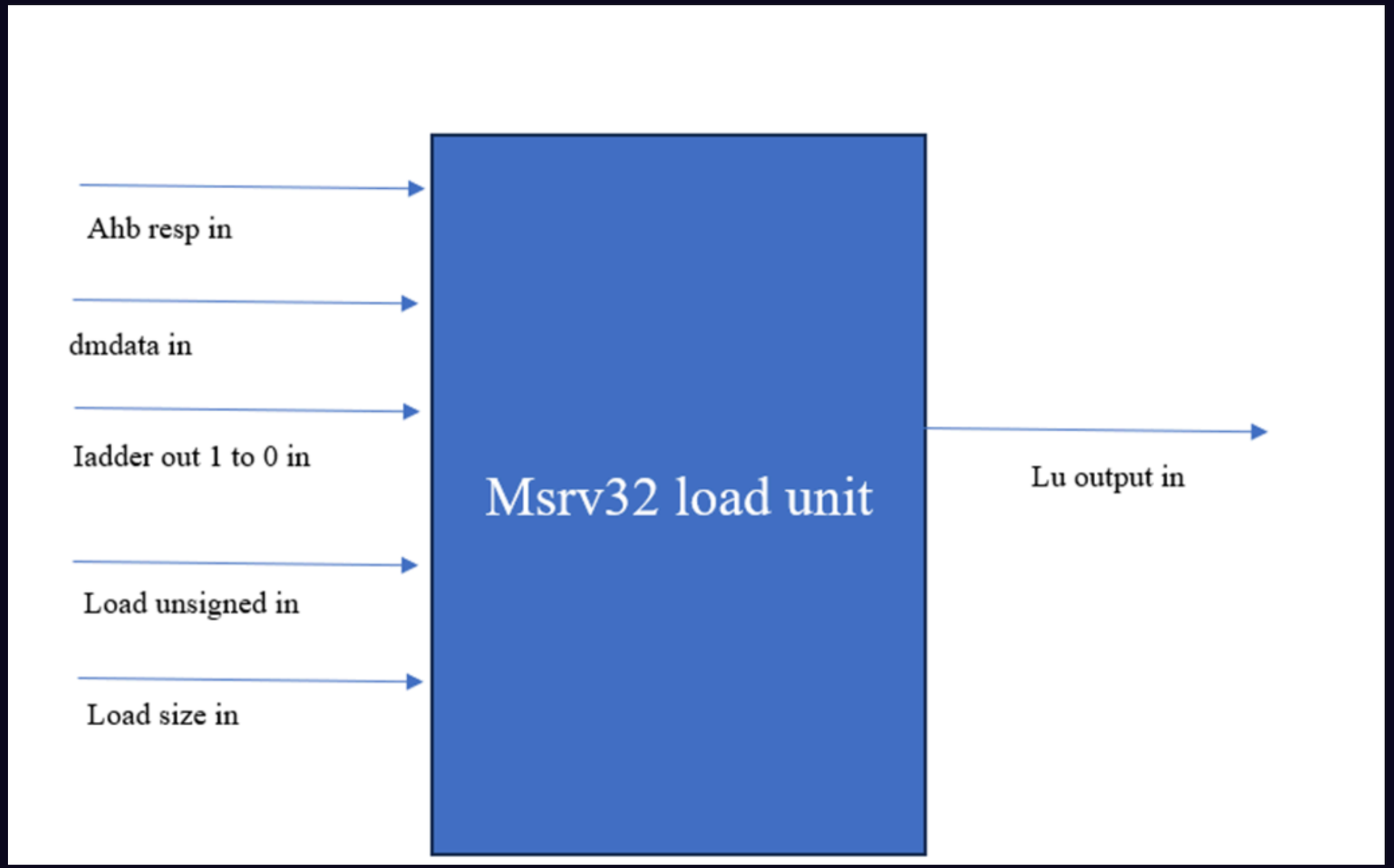
Read Operations:

- The rs_1_addr_in and rs_2_addr_in signals specify the addresses of the registers to be read for the first and second source operands (RS1 and RS2), respectively.
- The module uses these addresses to access the corresponding registers in reg_file and assign the read data to the rs_1_out and rs_2_out outputs.

Write Operations:

- The wr_en_in signal controls write operations to the register file.
- If wr_en_in is high (active) and rd_addr_in is valid (not zero), the data provided by rd_in is written to the register specified by the address rd_addr_in.

LOAD UNIT


Ahb resp in
dmdata in
Iadder out 1 to 0 in
Load unsigned in
Load size in
Msrv32 load unit
Lu output in

LOAD UNIT

Load Size:

- The load_size_in signal specifies the size of the data to be loaded.
- Based on this value, the module selects the appropriate portion of the data_in for the output.

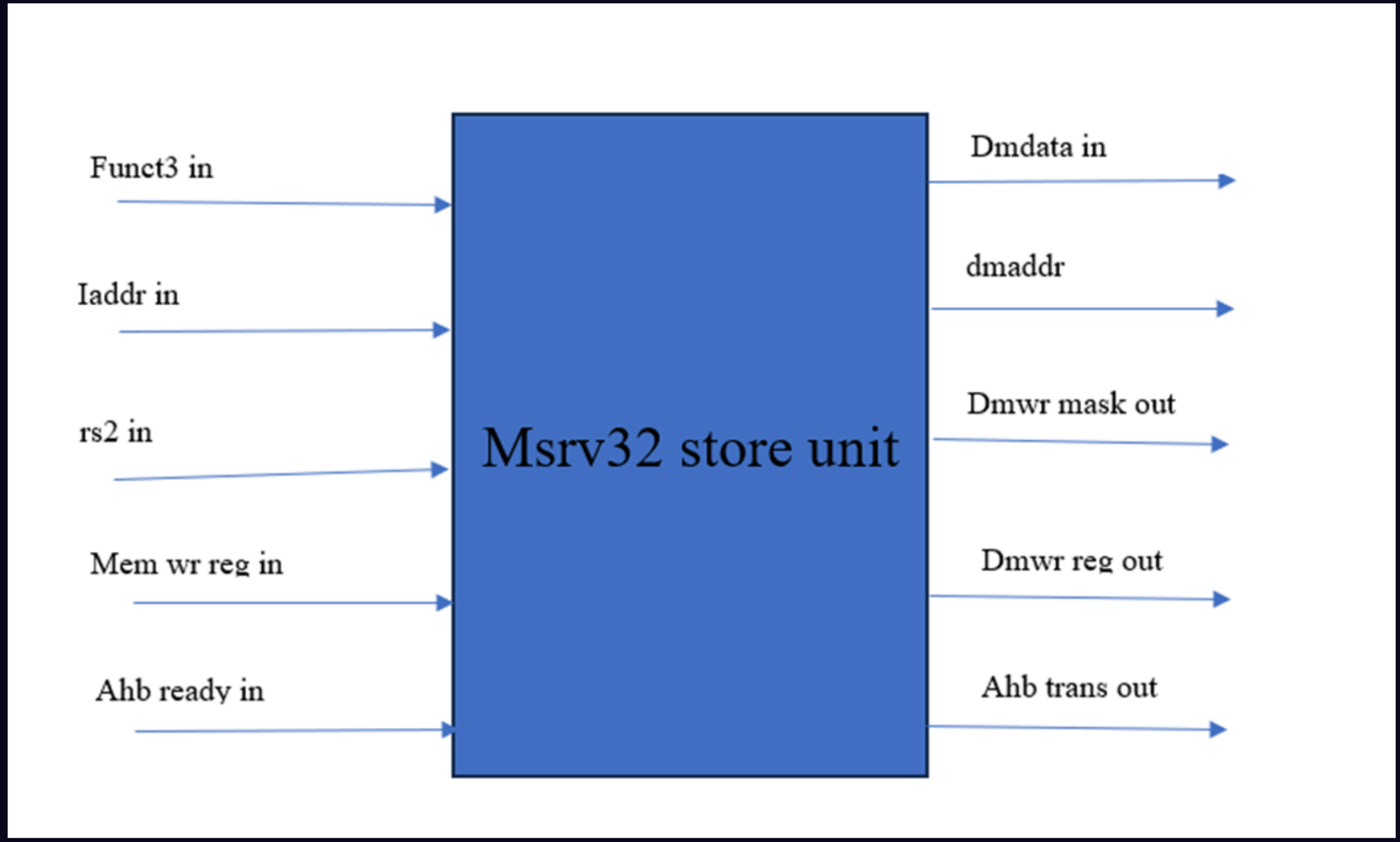
Byte Address Offset:

- The iadder_1_to_0_in signal indicates the byte offset within the loaded word (data_in) based on the base address used for the load instruction. This helps select the specific byte or half-word to be extracted.

Sign Extension:

- The load_unsigned_in signal determines whether sign extension is performed during the load operation.
- If load_unsigned_in is high (active), the module performs zero extension for byte and half-word loads. This means the sign bit (bit 7 for bytes and bit 15 for half-words) is replicated to fill the higher-order bits of the output.
  If load_unsigned_in is low (inactive), the sign bit from the loaded data is replicated to extend the sign for signed byte or half-word loads

STORE UINT


Funct3 in
Iaddr in
rs2 in
Mem wr reg in
Ahb ready in
Msrv32 store unit
Dmdata in
dmaddr
Dmwr mask out
Dmwr reg out
Ahb trans out

STORE UNIT

Address Calculation:

- The d_addr_out is assigned the base address from iadder_in with the two least significant bits cleared to ensure alignment for byte or halfword stores.

Data Formatting (always blocks):

- byte_dout: This register stores the rs2_in value shifted and padded with zeros depending on the byte address within the word
- halfword_dout: This register stores the rs2_in value shifted and padded with zeros depending on the halfword address within the word (

Output Selection and Bus Signals (always blocks):

- If ahb_ready_in is high (bus ready), the block selects the appropriate data (byte_dout, halfword_dout, or rs2_in) based on funct3_in for writing. If ahb_ready_in is low (bus not ready), the block sets ahb_htrans_out to 0 (no transfer).

Memory Write Request:

- The wr_req_out is simply assigned the same value as mem_wr_req_in. This module doesn't modify the memory write request itself.

WRITE BACK MUX

WRITE BACK MUX

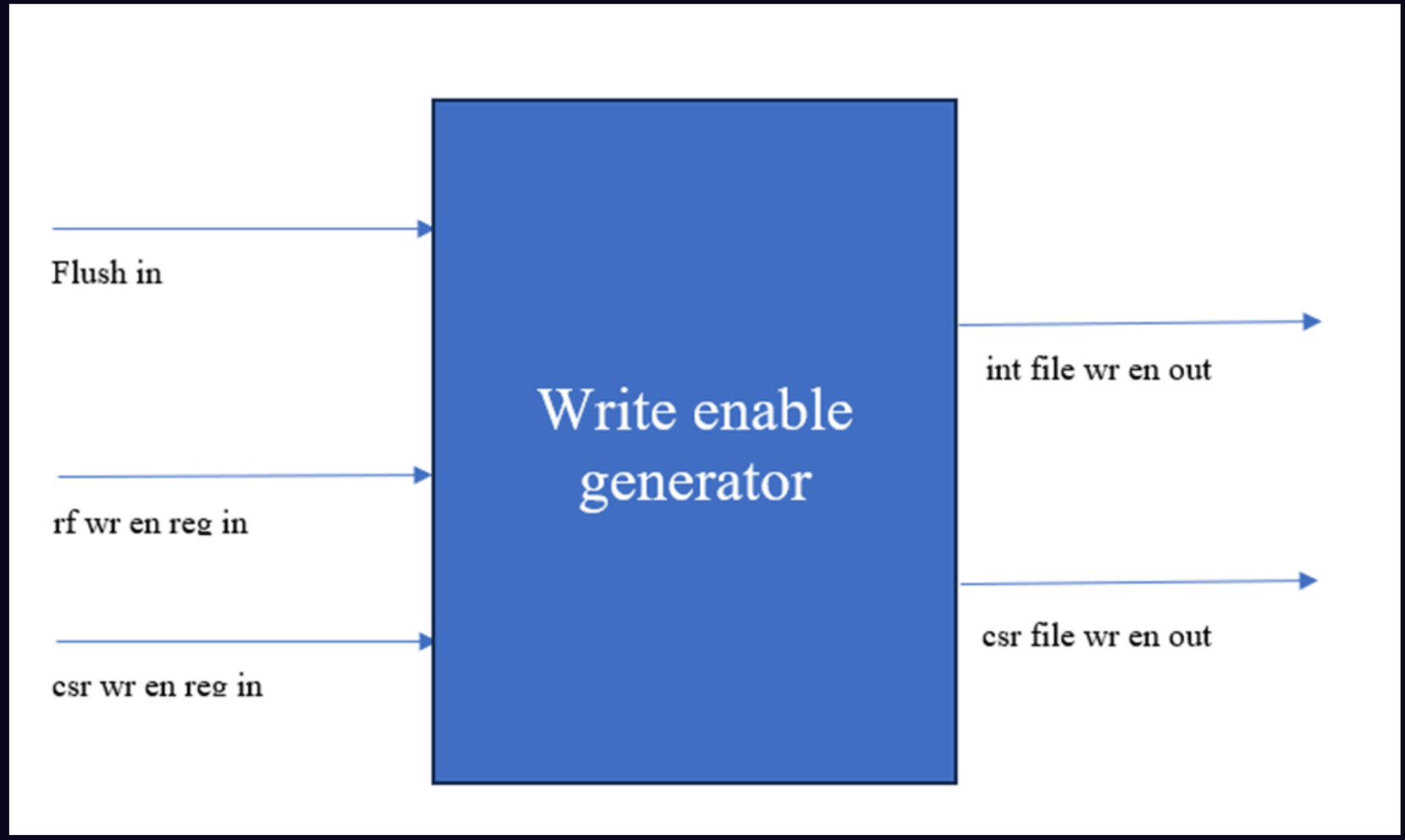
ALU Second Source Mux (always block assignment):

- This assignment selects the second operand for the ALU based on the alu_source_reg_in signal. It's done outside the main always block for efficiency.
- If alu_source_reg_in is high, it selects the value from the second source register (rs2_reg_in).
- If alu_source_reg_in is low, it selects the immediate value (imm_reg_in).

Write-back Mux Selection (always block):

- The always@* block selects the data source for writing back to the register file based on the wb_mux_sel_reg_in control signal.
- It uses a case statement to compare the control signal with predefined constants (WB_ALU, WB_LU, etc.). These constants represent different stages or results within the pipeline that can provide data for writing back.
- Depending on the matched case, the block assigns the corresponding data input (e.g., alu_result_in for ALU output, lu_output_in for Load Unit output, etc.) to the wb_mux_out register.

WRITE ENABLE GENERATOR



WRITE ENABLE GENERATOR

This module uses a simple assignment strategy to generate the write enable signals based on the flush_in signal and the corresponding register file write enable signals from the pipeline (rf_wr_en_reg_in and csr_wr_en_reg_in).

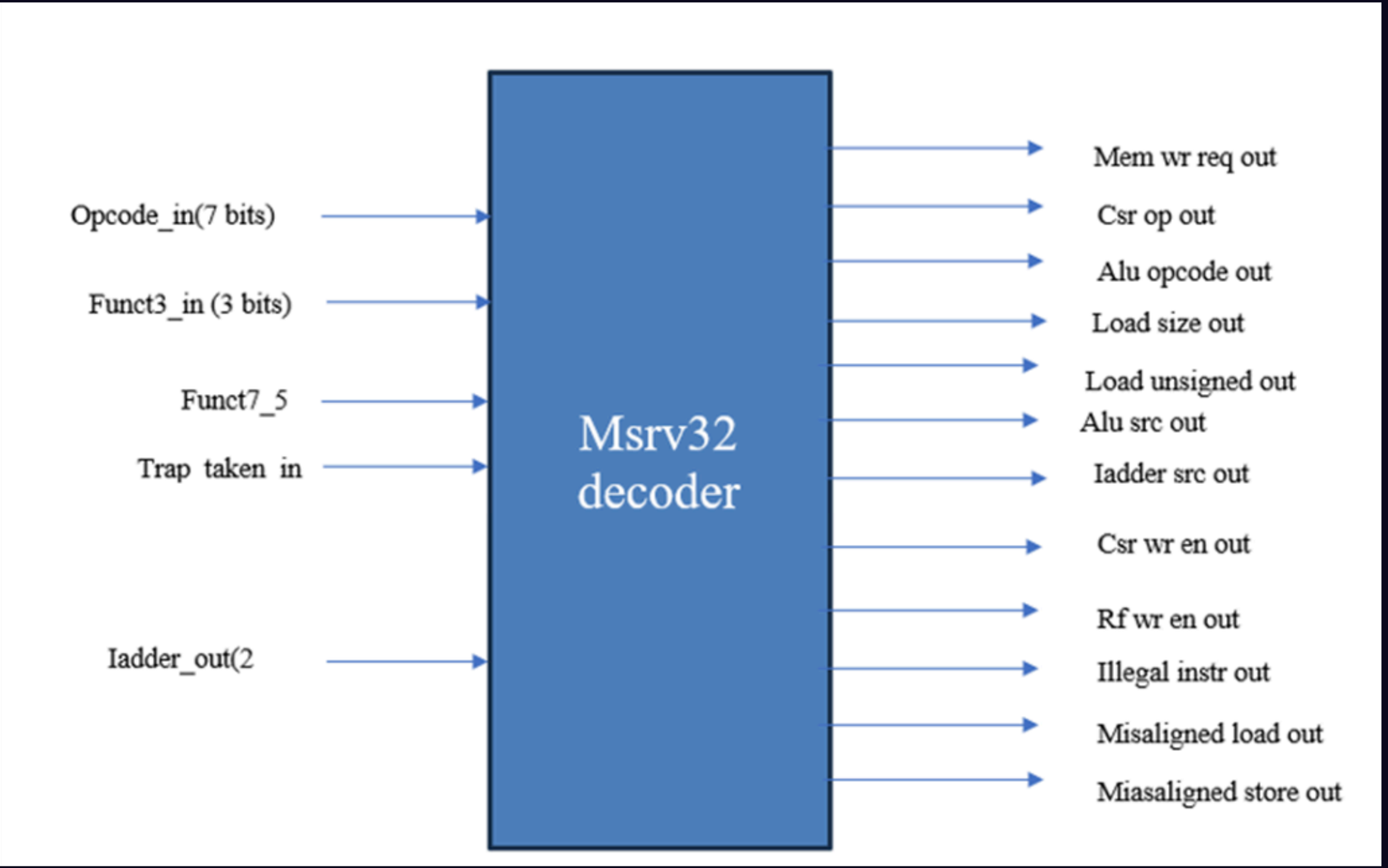
Flush Signal Priority:

- The flush_in signal is an active high signal, likely used for pipeline flushes or other events where writes need to be disabled.
- The assignments prioritize the flush_in signal. If flush_in is high, both output write enables (wr_en_integer_file_out and wr_en_csr_file_out) are set to low (writes disabled), regardless of the original write enable signals from the pipeline.

Normal Operation:

- If flush_in is low (inactive), the module assigns the respective register file write enable signals (rf_wr_en_reg_in and csr_wr_en_reg_in) to the outputs (wr_en_integer_file_out and wr_en_csr_file_out). This allows writes to proceed based on the pipeline's write enable signals when there's no flush condition.

DECODER


Opcode_in(7 bits)
Funct3_in (3 bits)
Funct7_5
Trap taken in
Iadder_out(2
Msrv32 decoder
Mem wr req out
Csr op out
Alu opcode out
Load size out
Load unsigned out
Alu src out
Iadder src out
Csr wr en out
Rf wr en out
Illegal instr out
Misaligned load out
Miasaligned store out

DECODER

Opcode Decoding:

- Decodes opcode_in[6:2] to determine the type of instruction and sets corresponding flags (e.g., is_branch, is_jal, is_jalr).

Function Code Evaluation:

- Evaluates the funct3_in field to determine specific operations for immediate ALU instructions and sets corresponding flags (is_addi, is_slti, is_sltiu, is_andi, is_ori, is_xori)..

Memory Load/Store Handling:

- load_size_out and load_unsigned_out are set based on funct3_in.
- mem_wr_req_out is set for store instructions unless there's a misalignment or a trap.
- Immediate Address Handling:
- Sets iadder_src_out for load, store, and jump register instructions.
- Determines misaligned load/store conditions based on funct3_in and iadder_1_to_0_in.

INSTRUCTION DECODER

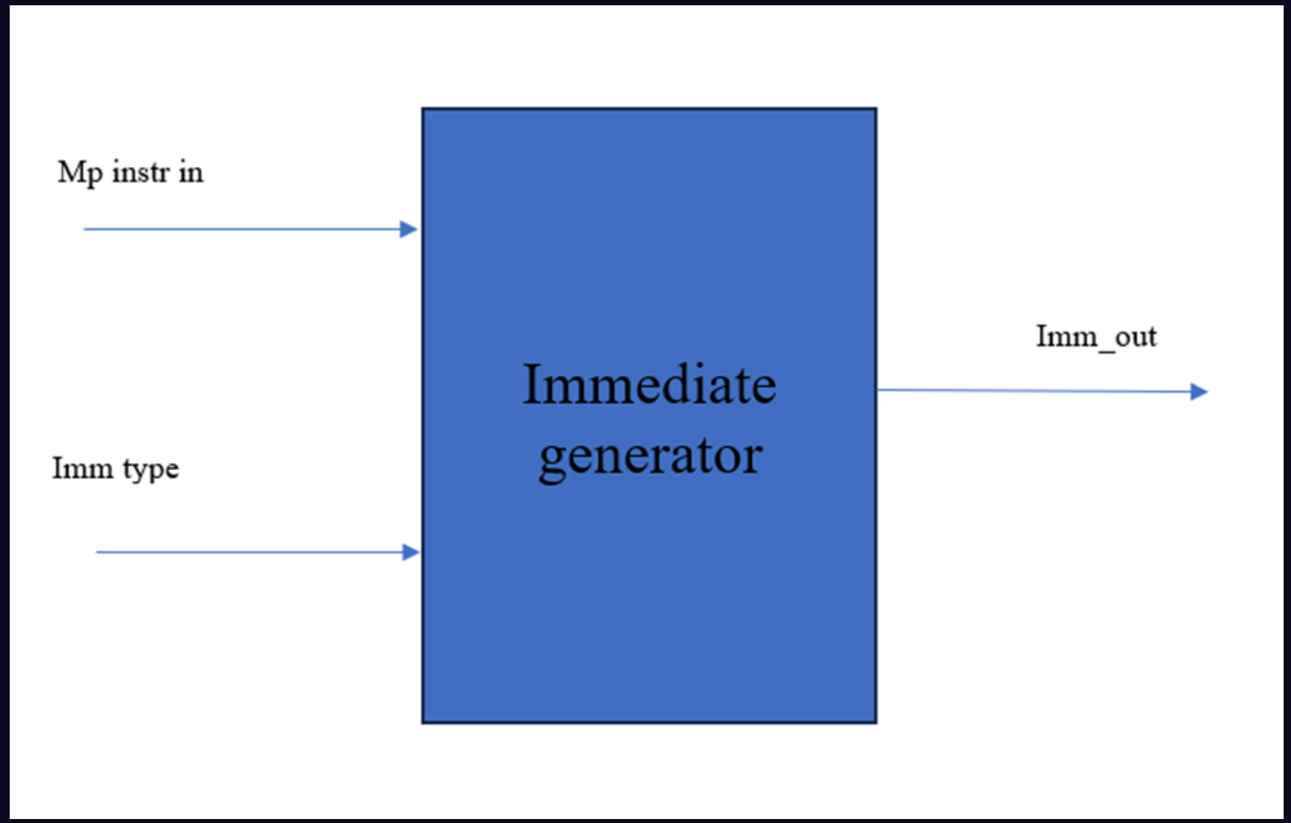
Control Signal Generation:

- csr_wr_en_out and csr_op_out are set for CSR instructions.
- rf_wr_en_out is set for instructions that require writing back to the register file.
- wb_mux_sel_out and imm_type_out are set based on the type of instruction.
- illegal_instr_out is set if the instruction is not implemented or has an invalid opcode.

Trap Handling:

- Misalignment checks (misaligned_load_out, misaligned_store_out) ensure proper handling of memory alignment issues

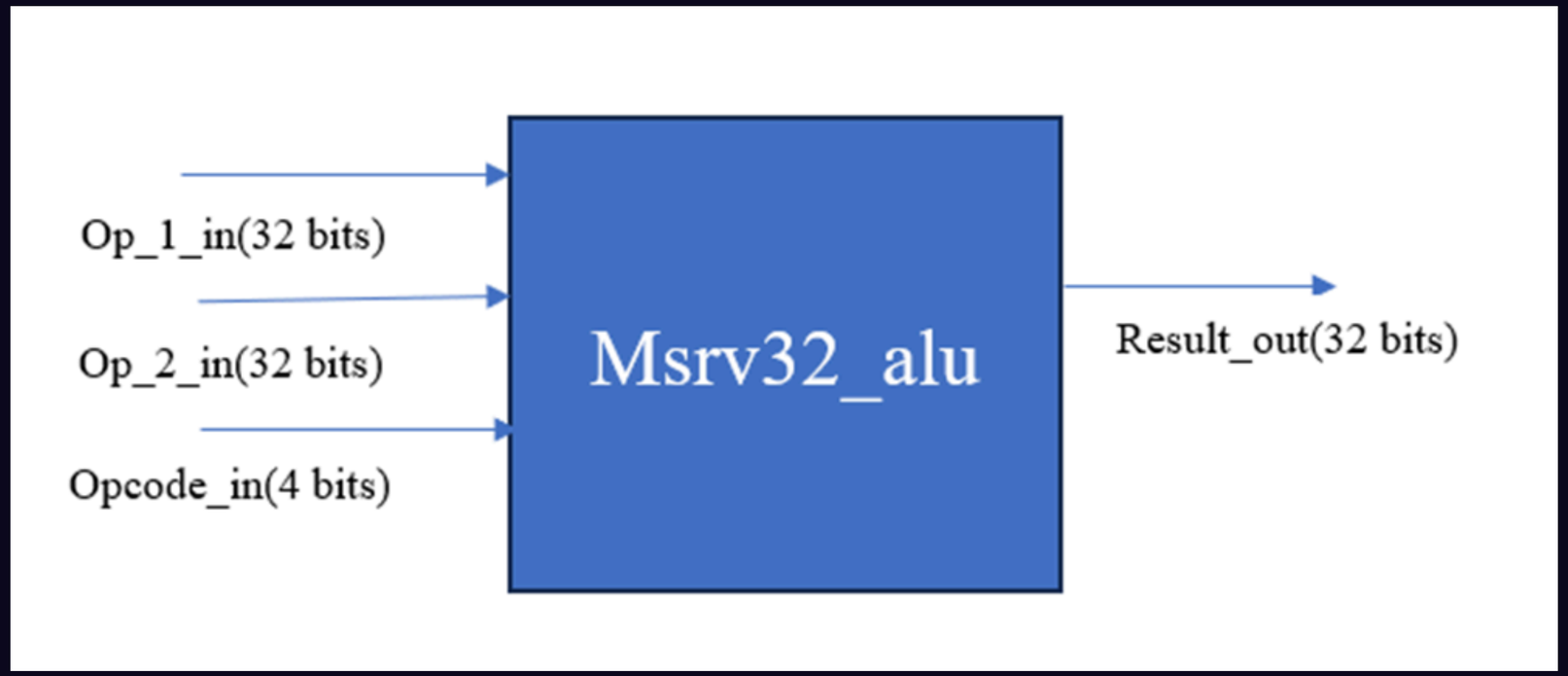
IMMEDIATE GENERATOR



IMMEDIATE GENERATOR

- ·**R-Type (imm_type_in = 000):** These instructions (e.g., add, sub) don't use an immediate value. The module sets imm_out to zero by replicating the sign bit (bit 31) of the instruction 20 times and concatenating it with the instruction bits 31:20.
- ·**I-Type (imm_type_in = 001):** These instructions (e.g., addi, lw, sw) use a sign-extended immediate value in the range of -2^12 to 2^12-1..
- ·**S-Type (imm_type_in = 010):** These instructions (e.g., sw, addi) use a sign-extended immediate value in the range of -2^12 to 2^12-1 for memory access..
- ·**B-Type (imm_type_in = 011):** These instructions (e.g., beq, jal) use a branch offset. The module creates a 32-bit immediate value by replicating the sign bit (bit 31) 19 times, concatenating it with bits 31, 7, 30:25, 11:8, and a 0 bit at the least significant position. This sign-extends the branch offset based on the branch target location.
- ·**U-Type (imm_type_in = 100):** These instructions (e.g., lui) use a 20-bit unsigned immediate value. The module takes instruction bits 31:12 and concatenates them with 12 zero bits, forming a 32-bit unsigned immediate value.
- ·**J-Type (imm_type_in = 101):** These instructions (e.g., jal) use a jump offset. The module replicates the sign bit (bit 31) 11 times, concatenates it with bits 31, 19:12, 20, 30:21, and a 0 bit, forming a 32-bit immediate value for the jump target address.

ALU BLOCK


Op_1_in(32 bits)
Op_2_in(32 bits)
Opcode_in(4 bits)
Msrv32_alu
Result_out(32 bits)

ALU BLOCK

Arithmetic:

- Addition: op_1_in + op_2_in (or op_1_in - op_2_in if the most significant bit of opcode_in is 1)

Logical:

- AND: op_1_in & op_2_in
- OR: op_1_in | op_2_in
- XOR: op_1_in ^ op_2_in

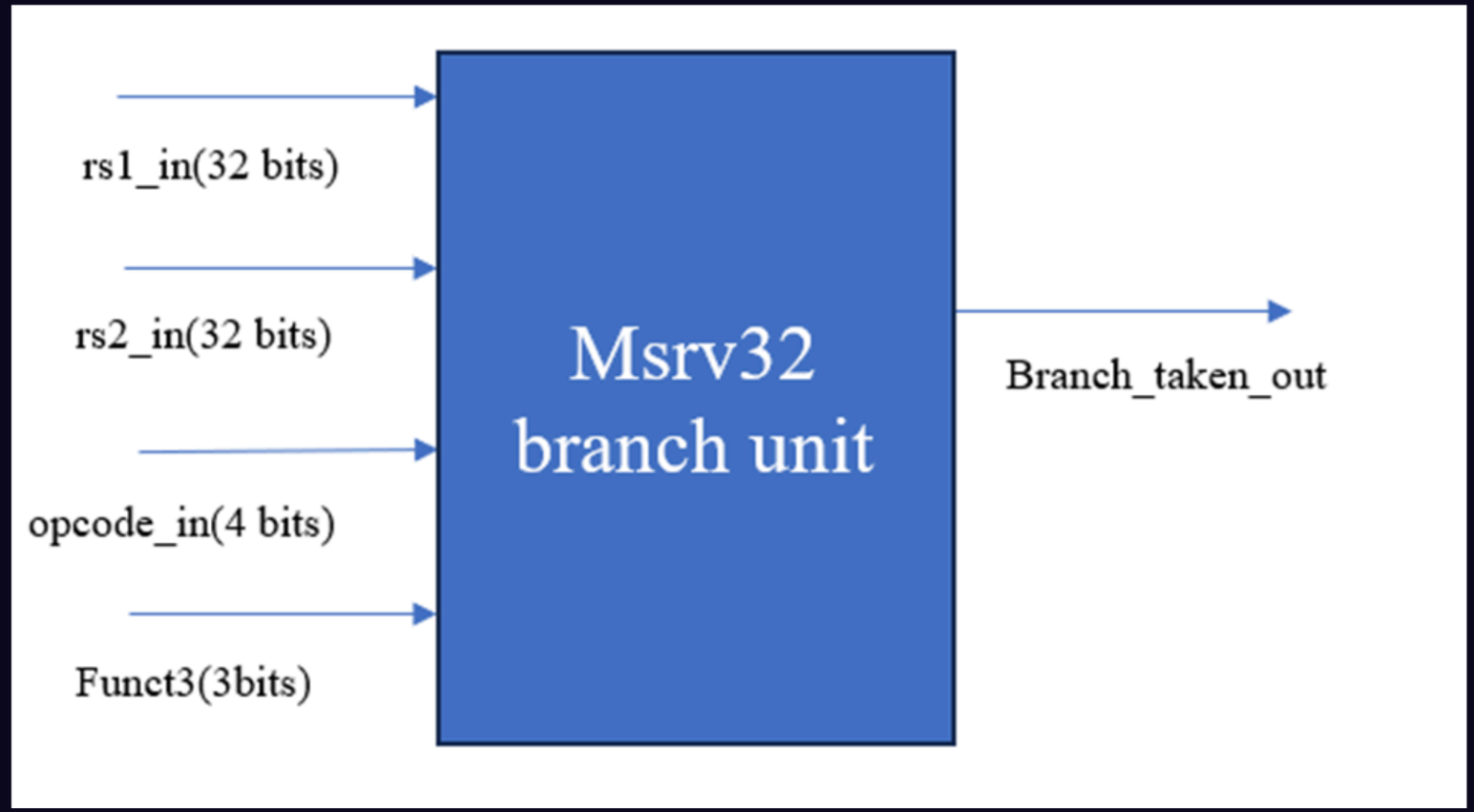
Shift:

- Left Shift Logical: op_1_in << op_2_in[4:0]
- Right Shift Logical: op_1_in >> op_2_in[4:0]
- Right Shift Arithmetic: op_1_in >>> op_2_in[4:0] (sign-extended shift)

Comparison:

- Set Less Than: op_1_in < op_2_in (sets the least significant bit of the result to 1 if op_1_in is less than op_2_in, otherwise 0)
- Set Less Than Unsigned: op_1_in < op_2_in (unsigned comparison)

BRANCH UNIT


rs1_in(32 bits)
rs2_in(32 bits)
opcode_in(4 bits)
Funct3(3bits)
Msrv32
branch unit
Branch_taken_out

BRANCH UNIT

Opcode Decoding:

- Determines the instruction type (branch, jump) based on the opcode_6_to_2_in input.
- Sets corresponding flags is_jal, is_jalr, or is_branch accordingly.

Branch Condition Evaluation:

- For branch instructions, evaluates the branch condition based on funct3_in and rs1_in, rs2_in values.
- Sets the take flag to indicate whether the branch should be taken.

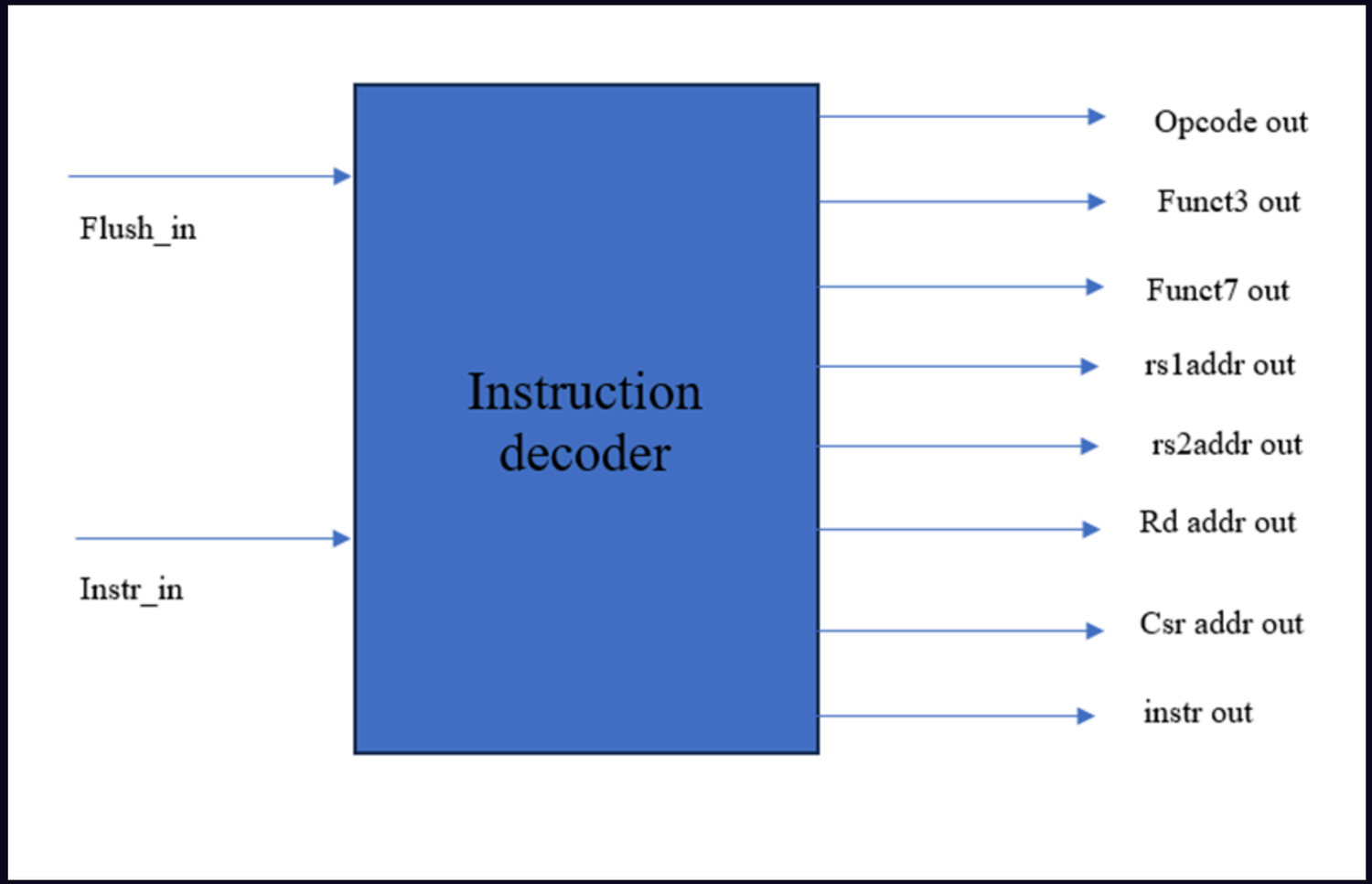
Jump Condition:

- For jump instructions, sets the pc_mux_sel flag to indicate that the PC should be updated with the jump target.

Branch Taken Output:

- Combines the pc_mux_sel_en and pc_mux_sel signals to determine the final branch_taken_out value.

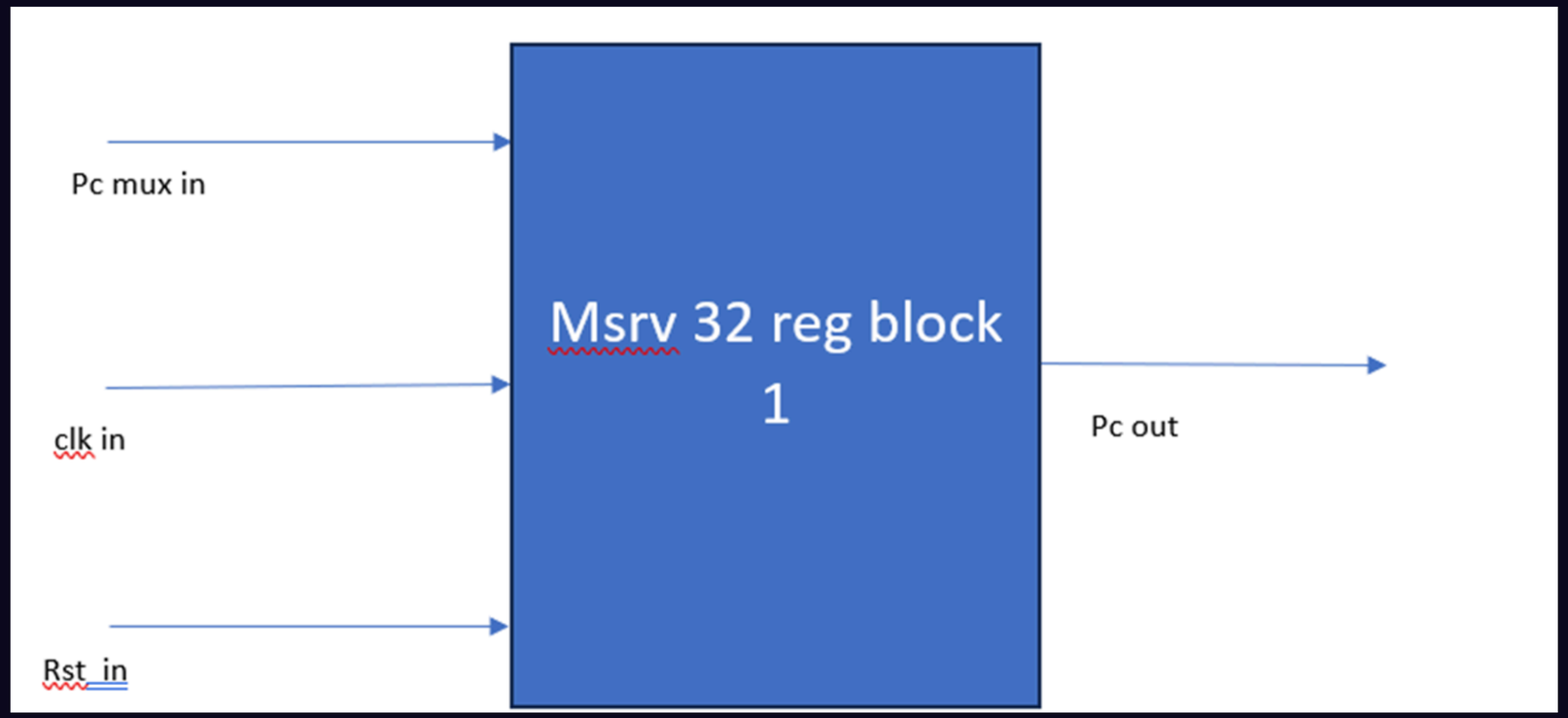
INSTRUCTION DECODER



INSTRUCTION DECODER

- opcode_out: Captures the opcode bits (bits 6:0) of the instruction.
- funct7_out: Captures the funct7 bits (bits 31:25) of the instruction. (Might be unused depending on the implementation)
- funct3_out: Captures the funct3 bits (bits 14:12) of the instruction.
- csr_addr_out: Captures the CSR address bits (bits 31:20) of the instruction. (Only used for CSR instructions)
- rs1_addr_out: Captures the register address bits for the first source operand (RS1) (bits 19:15).
- rs2_addr_out: Captures the register address bits for the second source operand (RS2) (bits 24:20).
- rd_addr_out: Captures the register address bits for the destination register (RD) (bits 11:7).
- instr_31_7_out: Captures the remaining bits of the instruction from bit 31 to bit 7 (25 bits). This might be used for immediate value extraction or other instruction-specific purposes.

REG BLOCK 1



REG BLOCK 1

- The module uses a positive edge-triggered clock (posedge clk_in) to synchronize its operation with the system clock.
- It includes an always block that executes on every positive clock edge.
- Inside the always block, there's an if-else statement to control the value written to the pc_out register.
- If rst_in is high (active reset), the pc_out is reset to the BOOT_ADDRESS. This ensures the program starts from the beginning when the system is reset.
- If rst_in is low (reset inactive), the pc_out is updated with the value provided by the pc_mux_in input. This allows the program counter to change based on the program execution flow.

REG BLOCK 2

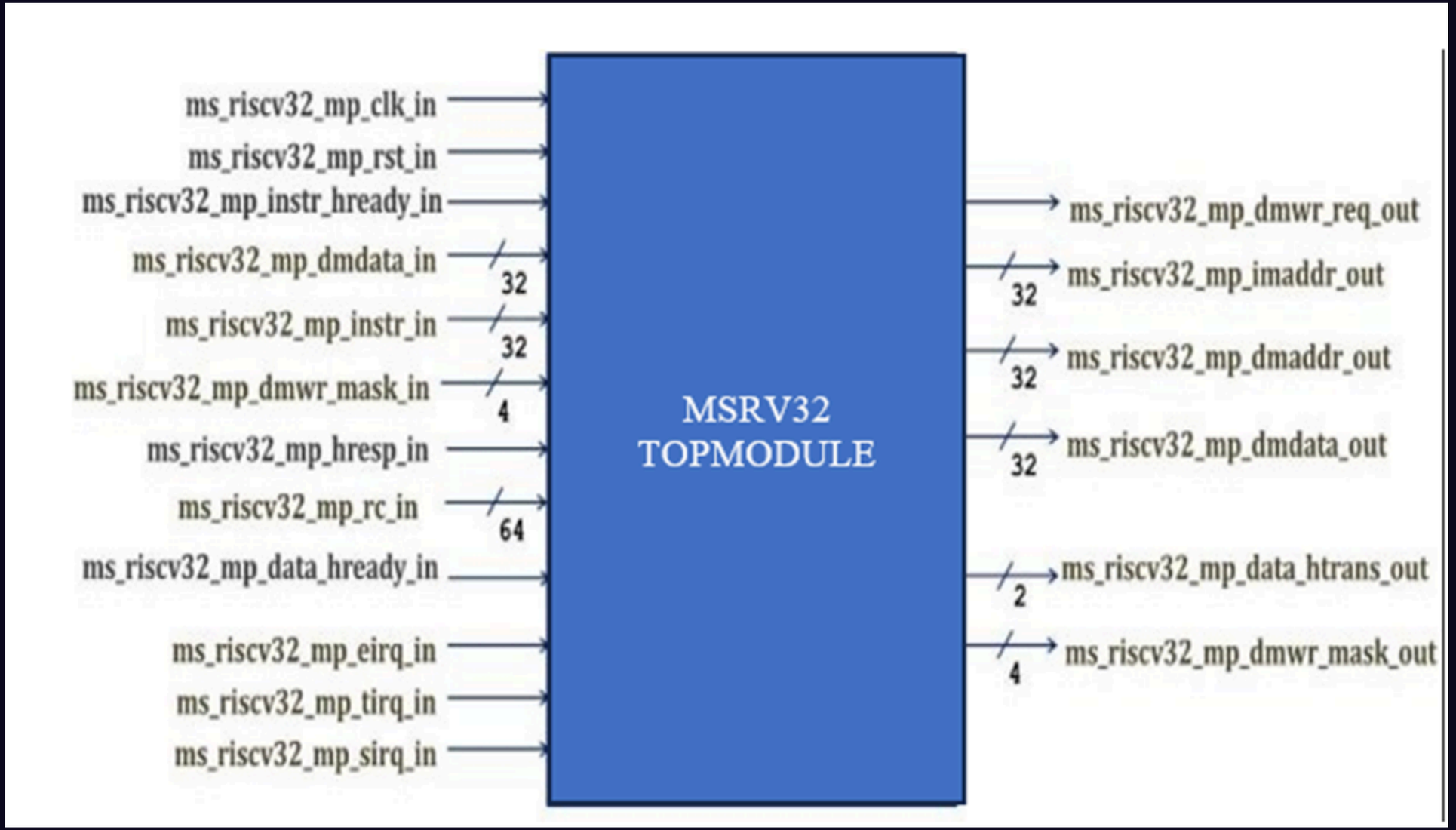

Branch taken in
Rd addr in
csr addr in
Rs1 in
Rs2 in
pc in
Pc plus 4 out
Alu opcode in
Load size in
Load unsigned in
Abh src in
Csr w en in
Rf w en in
Wb mux sel in
Csr op on
Rst in
Clk in
Imm in
Iaddr in
Msrv32 reg block 2
Rd addr reg
csr addr reg
Rs1 reg out
Rs2 reg out
Pc mux out
Pc plus 4 reg out
Alu opcode reg out
Load size reg out
Load unsigned reg out
Abh src reg out
Csr w ren reg out
Rf w ren reg out
Wb mux sel reg out
Csr op reg out
Imm reg out
Iaddr reg out

REG BLOCK 2

The module uses a positive edge-triggered clock (posedge clk_in) to synchronize its operation with the system clock.

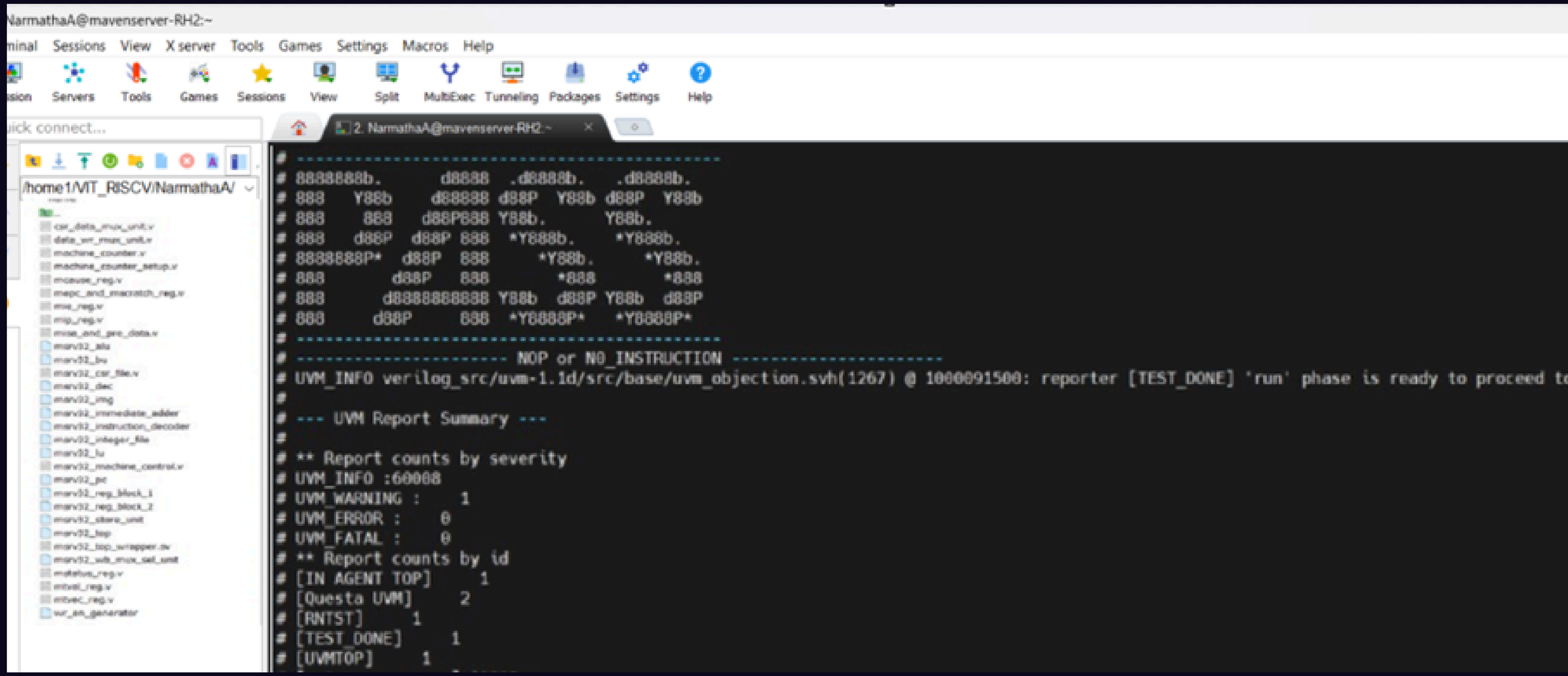
- It includes an always block that executes on every positive clock edge.
- Inside the always block, there's an if-else statement to control the value written to the pc_out register.
- If rst_in is high (active reset), the pc_out is reset to the BOOT_ADDRESS. This ensures the program starts from the beginning when the system is reset.
- If rst_in is low (reset inactive), the pc_out is updated with the value provided by the pc_mux_in input. This allows the program counter to change based on the program execution flow.

TOP MODULE


ms_riscv32_mp_clk_in
ms_riscv32_mp_rst_in
ms_riscv32_mp_instr_hready_in
ms_riscv32_mp_dmdata_in
32
ms_riscv32_mp_instr_in
32
ms_riscv32_mp_dmwr_mask_in
4
ms_riscv32_mp_hresp_in
ms_riscv32_mp_rc_in
64
ms_riscv32_mp_data_hready_in
ms_riscv32_mp_eirq_in
ms_riscv32_mp_tirq_in
ms_riscv32_mp_sirq_in
MSRV32
TOPMODULE
ms_riscv32_mp_dmwr_req_out
ms_riscv32_mp_imaddr_out
32
ms_riscv32_mp_dmaddr_out
32
ms_riscv32_mp_dmdata_out
32
ms_riscv32_mp_data_htrans_out
2
ms_riscv32_mp_dmwr_mask_out
4

TOP MODULE

- I**ntegration**: It integrates and connects all the lower-level modules of the design, establishing the overall system's architecture.
- Interface Definition: It defines the external interface of the design, including input and output ports, clock signals, and reset signals.
- **Configuration:** It includes parameters and settings that configure the entire design, such as clock frequencies, memory sizes, and other system-level parameters.
- **Control Logic:** It may contain control logic that orchestrates the interactions between different modules, ensuring proper sequencing and data flow.
- **Testbench Connection:** In simulation, it connects to the testbench, allowing the design to be verified and validated.


NarmathaA@mavenserver-RH2:~
minal Sessions View X server Tools Games Settings Macros Help
sion Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
uick connect...
2. NarmathaA@mavenserver-RH2:~
/home1/VIT_RISCV/NarmathaA/
8888888b. d8888 .d8888b. .d8888b.
888 Y88b d88888 d88P Y88b d88P Y88b
888 888 d88P888 Y88b. Y88b.
888 d88P d88P 888 *Y888b. *Y888b.
8888888P* d88P 888 *Y88b. *Y88b.
888 d88P 888 *888 *888
888 d8888888888 Y88b d88P Y88b d88P
888 d88P 888 *Y8888P* *Y8888P*
---------- NOP or NO_INSTRUCTION ----------
UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 1000091500: reporter [TEST_DONE] 'run' phase is ready to proceed to
--- UVM Report Summary ---
** Report counts by severity
UVM_INFO :60008
UVM_WARNING : 1
UVM_ERROR : 0
UVM_FATAL : 0
** Report counts by id
[IN AGENT TOP] 1
[Questa UVM] 2
[RNTST] 1
[TEST_DONE] 1
[UVMTOP] 1

CONCLUSION

The combination of advanced VLSI technology, the RISC-V ISA, and efficient pipeline designs makes it possible to develop high-performance, cost-effective processors suitable for a wide range of applications. The careful management of hazards and design considerations ensures that these processors meet the stringent demands of modern computing environments.

LEARNING OUTCOME

- **Digital Design:** A deep understanding of digital design principles, including logic gates, Boolean algebra, and sequential logic.
- **Verilog HDL:** Proficiency in writing and simulating Verilog code, a crucial skill for hardware design.
- **Computer Architecture:** Knowledge of processor architecture, instruction set design, pipelining, and memory systems.
- **VLSI Design Flow:** Familiarity with the complete VLSI design flow, from RTL design to physical synthesis and layout.
- **Testing and Verification:** Experience in writing testbenches, simulating designs, and debugging issues.

FUTURE SCOPE

- **AI and Machine Learning:** Custom RISC-V cores can be designed to accelerate specific AI and ML workloads, improving performance and efficiency.
- **Automotive and Aerospace:** RISC-V's open-source nature and potential for customization make it attractive for safety-critical applications in these industries.
- **Advanced Microarchitectures:** Continued research and development in microarchitecture techniques, such as out-of-order execution and speculative execution, will enhance the performance of RISC-V processors.
- The future of RISC-V RV32 processors looks bright. Its flexibility, open-source nature, and growing ecosystem make it a compelling choice for a wide range of applications. As the technology continues to mature, we can expect to see even more innovative and powerful RISC-V-based products in the years to come.