

RISC-V ISA & RV32I RTL Design

Adithiyha R

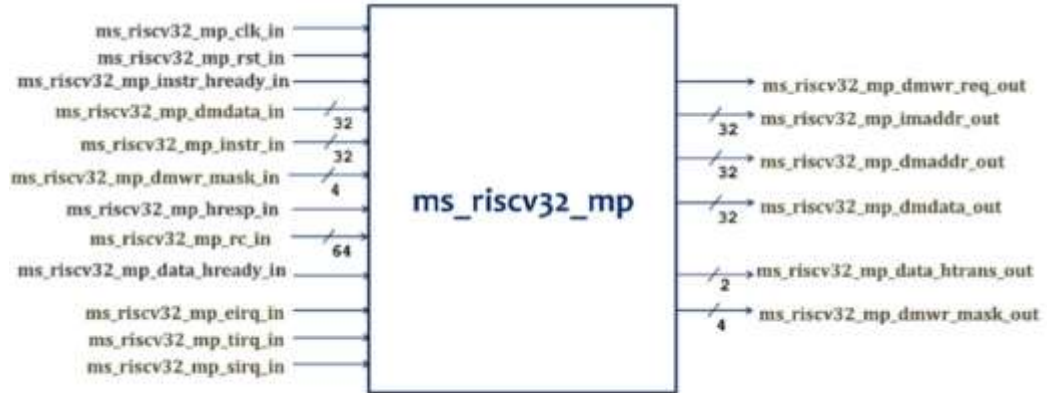
21BEE1181

Table of Contents

- 1. RISC-V Block Diagram & Micro Architecture**
 - 1.1 Block Diagram**
 - 1.2 Micro Architecture with 3 stage pipelining (without inter connections between various blocks)**
 - 1.3 Micro Architecture with 3 stage pipelining (with inter connections between various blocks)**
 - 1.4 Micro Architecture with connection in Pipeline Stage 1**
 - 1.5 Micro Architecture with connection in pipeline Stage 2**
 - 1.6 Micro Architecture with connection in pipeline Stage 3**
- 2. Various Blocks in RISC-V Micro Architecture**
 - 2.1 PC MUX**
 - 2.2 Reg Block 1**
 - 2.3 Immediate generator**
 - 2.4 Immediate adder**
 - 2.5 Integer File**
 - 2.6 Write Enable Generator**
 - 2.7 Instruction Mux**
 - 2.8 Branch Unit**
 - 2.9 Decoder**
 - 2.10 Machine Control**
 - 2.11 CSR File**
 - 2.12 Reg Block 2**
 - 2.13 Store Unit**
 - 2.14 Load Unit**
 - 2.15 ALU**
 - 2.16 WB Mux Selection Unit**

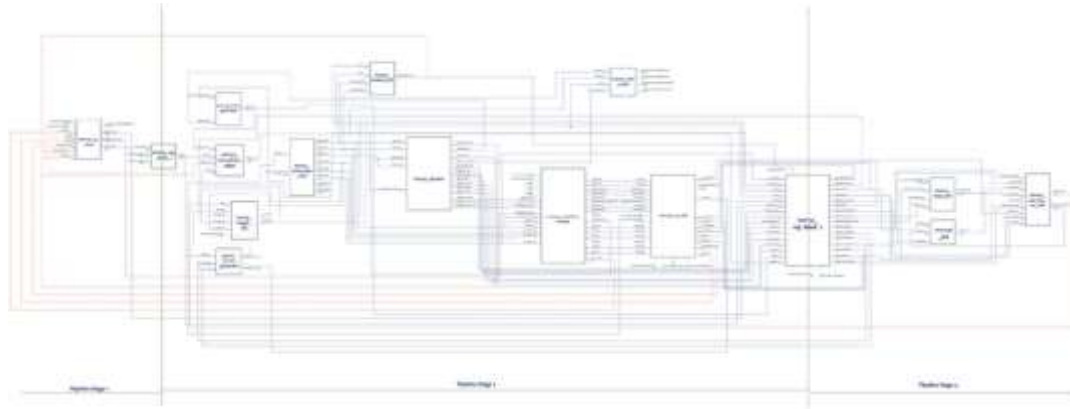
1. RISC-V Block Diagram and Micro Architecture

1.1 Block Diagram

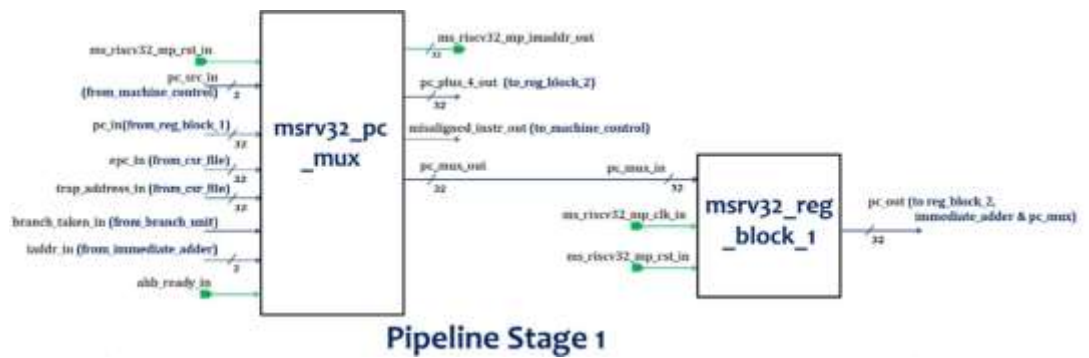


Signal name	Description
ms_riscv32_mp_clk_in	System clock
ms_riscv32_mp_rst_in	System Reset (Power on Reset), active high asynchronous
ms_riscv32_mp_dmdata_in	Contains the data fetched from the external memory.
ms_riscv32_mp_instr_hready_in	Active high signal which indicates that the AHB slave (Instruction Memory) is ready to sample the instruction address & start driving the instruction.
ms_riscv32_mp_data_hready_in	Active high signal which indicates that the AHB slave (data memory) is ready to sample data from the processor during write operation.
ms_riscv32_mp_hresp_in	Active low signal which indicates that the AHB slave (data memory) is ready to drive data to the processor during read operation.
ms_riscv32_mp_data_htrans_out	AHB master generates htrans_out to indicate whether the transfer is a Non-sequential transfer or an IDLE transfer.
ms_riscv32_mp_instr_in	Contains the instruction fetched from the external memory.

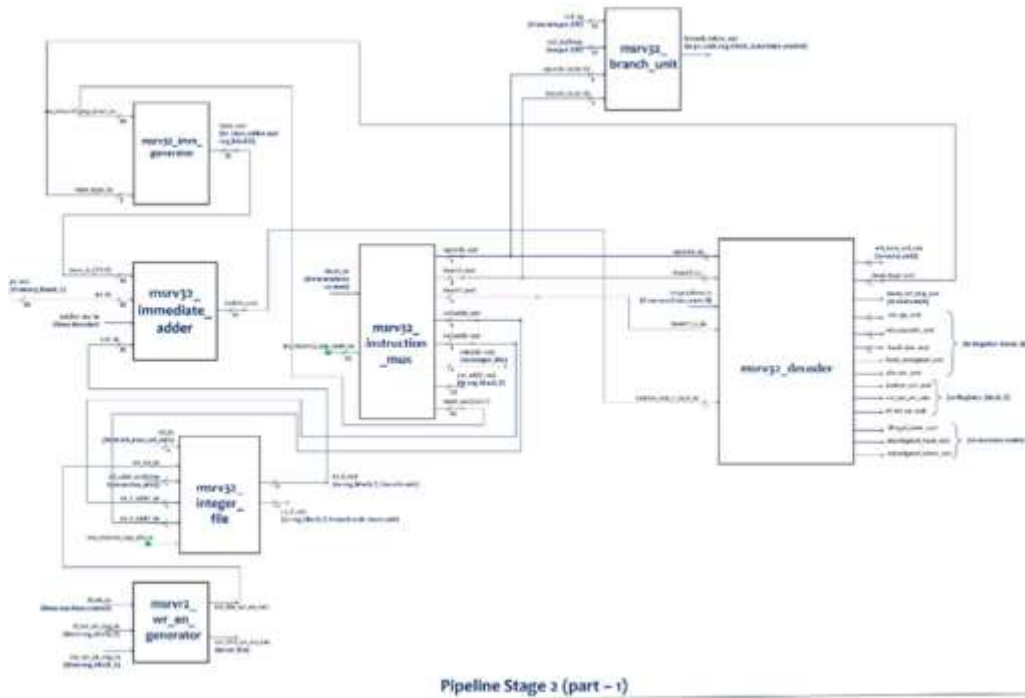
1.3 Micro Architecture with 3 stage pipelining (with inter connections between various blocks)



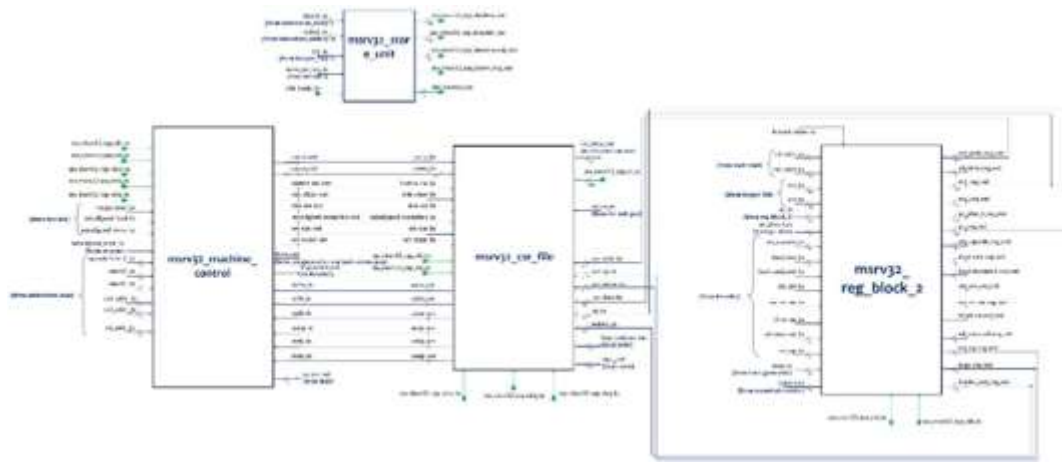
1.4 Micro Architecture with connections in pipeline Stage 1



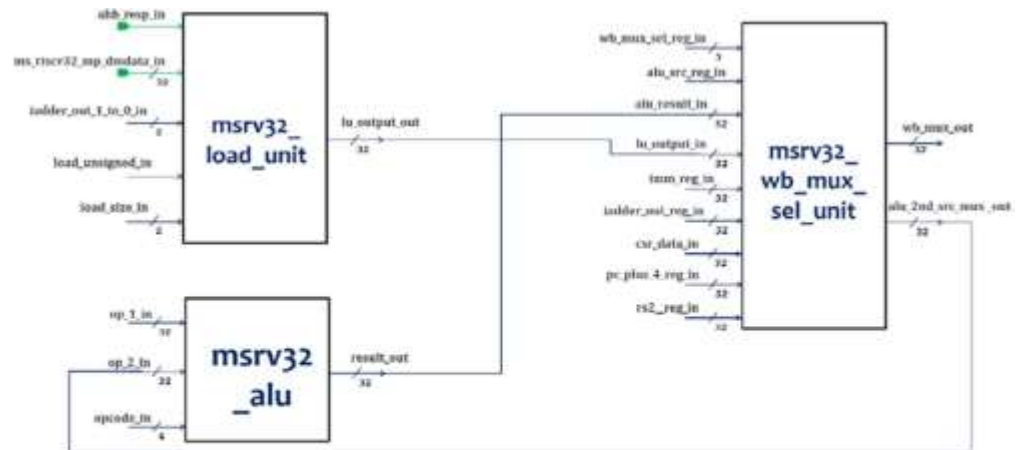
1.5 Micro Architecture with connections in Pipeline Stage 2 (Part 1)



Micro Architecture with connections in Pipeline Stage 2 (Part 2)



1.5 Micro Architecture with connections in Pipeline Stage 3

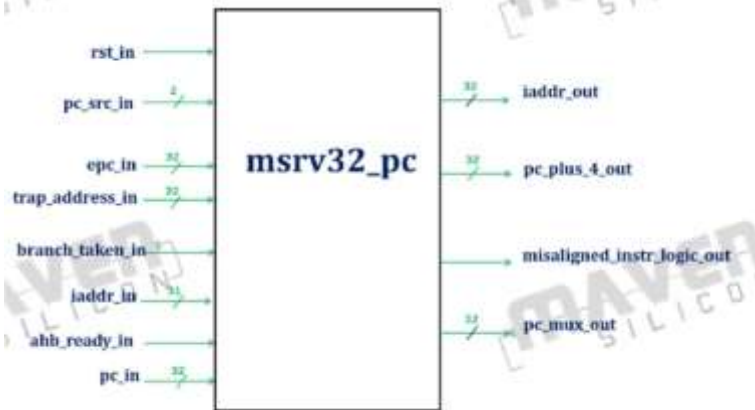


Pipeline Stage 3

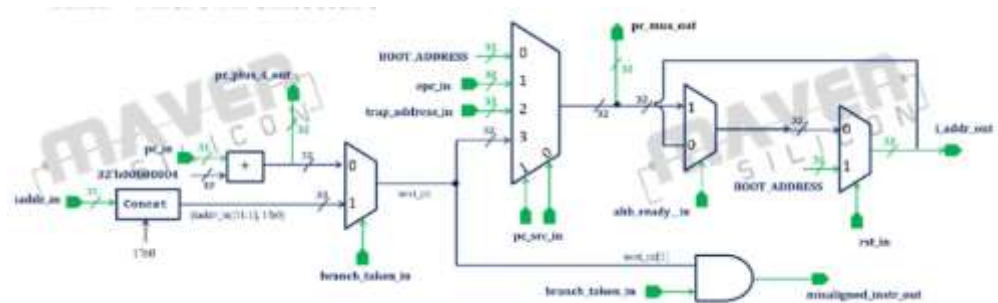
2. Various Blocks in RISC – V Micro Architecture

2.1 PC MUC

2.1.1 BLOCK DIAGRAM



2.1.2 Micro Architecture



2.1.3 Signal Description

Signal name	Description
rst_in	Power on reset (from core)
pc_src_in	Current state of the core. (from machine control)
ahb_ready_in	Active high signal used to update the i_addr_out
epc_in	Trap_return address. (from csr file)
trap_address_in	Address when interrupt occurs. (from csr file)
branch_taken_in	Indication when branch instruction occurs. (from branch unit)
iaddr_in	Address with addition of immediate value. (from immediate adder)
pc_plus_4_out	Registered for stage_2 operation. (to reg_block_2)
pc_mux_out	Similar to pc_out but not registered. (to reg_block_1)
misaligned_instr_logic_out	Indicates misaligned instruction. (to machine control)
iaddr_out	Output of the Core (instruction_address)
pc_in	Program Counter input (from reg_block_1)

2.1.4 Functionality

This module will be used to hold the address of next instruction to be executed.

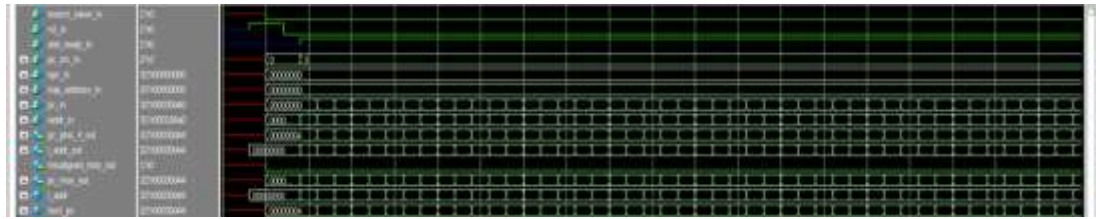
1. 'i_adder_out' is used to interface with instruction memory.
2. Based on the current state of the core, the program counter will be loaded.
 - When the interrupt happens then PC should go to trap_address_in.
 - After completion of interrupt PC should go back to epc_in.
 - For normal operation PC should go to new address ({iaddr_in[31:1],1'b0}) if there is a branch instruction to be executed or else 'pc_in + 32'h00000004'.
 - For reset state PC should get 'BOOT_ADDRESS'.
(BOOT_ADDRESS is a parameter of 32'h00000000 value).
3. 'pc_plus_out' will be assigned with 'pc_in + 32'h00000004'.
4. 'misaligned_instr_logic_out' will be generated by doing 'AND' operation between 'branch_taken_in' and first bit of next_pc.

5. If there is a reset then iaddr_out should get 'BOOT_ADDRESS' else if it checks if the 'ahb_ready_in' high, then iaddr_out gets pc_mux_out.

'pc_src_in' signals combinations and their description

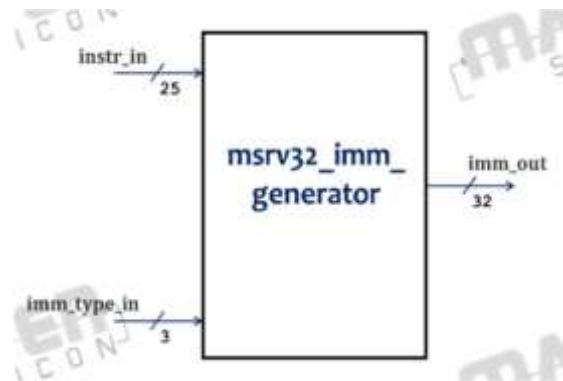
Combination Value	Description
2'b00	Reset State
2'b01	Trap_return (completion of Interrupts)
2'b10	Trap_taken (Interrupts)
2'b11	Operating State
Default	Operating State

2.1.5 Output Waveform



2.3 Immediate generator

2.3.1 Block Diagram



2.3.2 Functionality

The Immediate Generator rearranges the immediate bits contained in the instruction and, if necessary sign-extends it to form a 32-bit value. The unit is controlled by the `imm_type_out` signal, generated by the Control Unit. Table shows the unit input and output signals. The `imm_type_in` is generated by `msrv32_decoder` module.

The below table depicts the region of immediate data for different types of instructions

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12:10:5]				rs2		rs1		funct3		imm[4:1:11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20:10:1:11:19:12]										rd		opcode		J-type

From the instruction set we can conclude that opcode [6:4] is used to distinguish between different types of instruction. Assign `imm_out` according to the below table.

3'b000	R_TYPE
3'b001	I_TYPE
3'b010	S_TYPE
3'b011	B_TYPE
3'b100	U_TYPE
3'b101	J_TYPE
3'b110	CSR_TYPE
3'b111	I_TYPE (default)

2.3.3 OUTPUT WAVEFORM

	Msgs								
sim:/msrv32_img/in...	00000000000000...		0...	0...	0...	0...	0...	0...	0...
sim:/msrv32_img/im...	000		000	000	000	000	000	000	000
sim:/msrv32_img/im...	00000000000000...		0...	0...	0...	0...	0...	0...	0...

2.4 Immediate adder

2.4.1 BLOCK DIAGRAM



2.4.2 Functionality

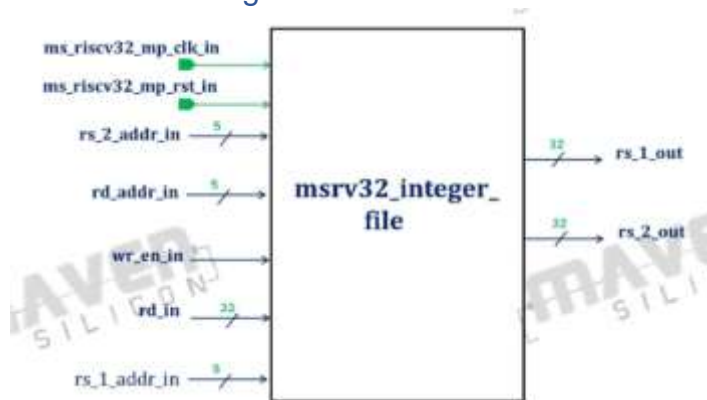
The output of immediate adder module will be 32 bit address depending on the instruction format (instruction to the core). This module will add the 'immediate' value with 'rs1' when core wants to perform load, store, jump and link reg (jalr) instructions. The core is performing 'branch' instruction and jump and link (jal) then 'immediate' value will be added in a 'PC'. So that core will jump on new address with the help of program counter. When core 'iadder_src_in' is high then 'iadder_out' = $rs_1_in + imm_in$ else 'iadder_out' = $pc_in + imm_in$.

OUTPUT WAVEFORM

[illegible]

2.5 Integer File

2.5.1 Block Diagram



2.5.2 Functionality

The msrv32_integer_file has 32 general-purpose registers and supports read and write operations. Reads are requested by pipeline stage 2 and provide data from one or two registers. Writes are requested by stage 3 and put the data coming from the Write back Multiplexer into the selected register.

Initially all registers can be initialized with 0. The first register is hardwired with 0. No write operations are permitted for reg_file [0] location.

If stage 3 requests to write to a register being read by stage 2, the data to be written is immediately forwarded to stage 2 to avoid data hazard.

This can be done by forwarding the rd_in directly to rs_1_out and rs_2_out instead of reading from reg_file.

If rs_1_addr_in is equal to rd_addr_in and wr_en_in is asserted then drive rs_1_out with rd_in else reg_file [rs_1_addr_in]

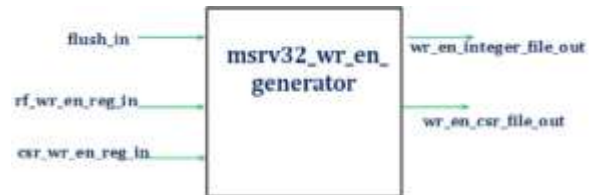
If rs_2_addr_in is equal to rd_addr_in and wr_en_in is asserted then drive rs_2_out with rd_in else reg_file [rs_2_addr_in]

2.5.3 OUTPUT WAVEFORM

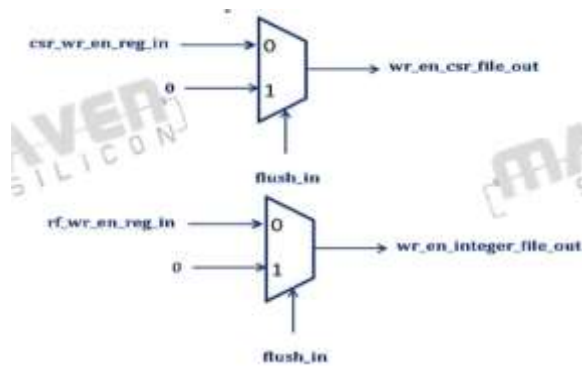


2.6 Write Enable Generator

2.6.1 Block Diagram



2.6.2 Functionality

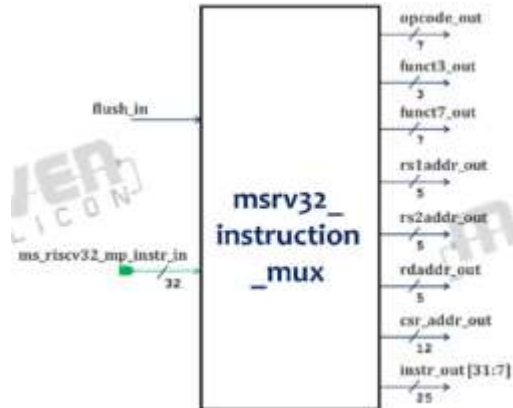


2.6.3 OUTPUT Waveform



2.7 Instruction MUX

2.7.1 BLOCK DIAGRAM

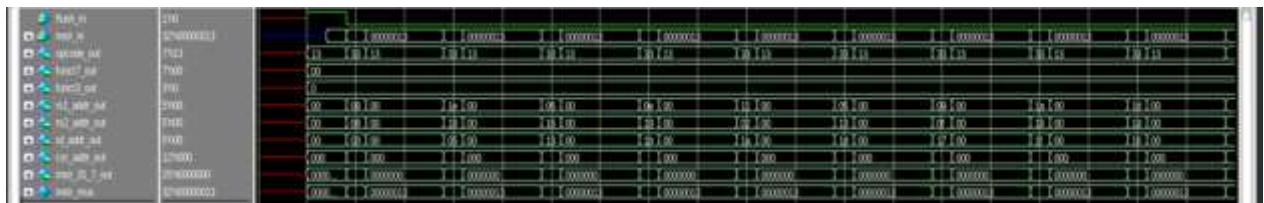


2.7.2 Functionality

This module takes the instruction (input to core) and provides the fields which is used by the other modules to perform there functionality.

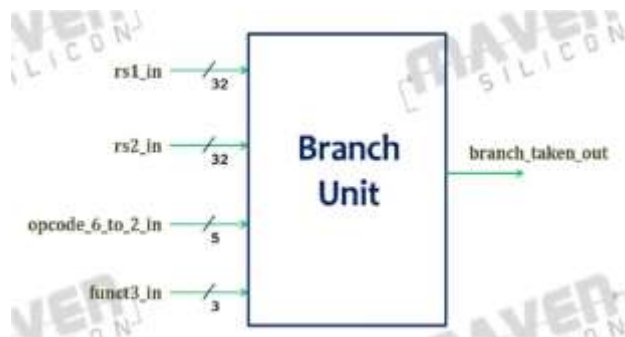
1. When 'flush' is high use 32'h00000013 to provide the fields.
2. When 'flush' is low use core instruction (ms_riscv32_mp_instr_in) to provide the fields.
3. opcode_out = instr_mux [6:0]
 funct3_out = instr_mux [14:12]
 funct7_out = instr_mux [31:25]
 csr_addr_out = instr_mux [31:20]
 rs1addr_out = instr_mux [19:15]
 rs2addr_out = instr_mux [24:20]
 rdaddr_out = instr_mux [11:7]
 instr_out = instr_mux [31:7]

2.7.3 Output Waveform



2.8 Branch Unit

2.8.1 Block Diagram

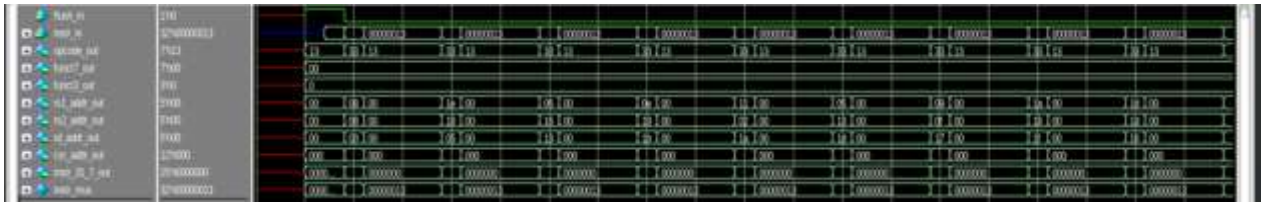


2.8.2 Functionality

The Branch Unit decides if a branch instruction must be taken or not. It receives two operands from the Integer Register File and, based on the value of opcode and funct3 instruction fields, it decides the branch. Branch conditions are conditional jumps. Jump instructions are interpreted as unconditional jumps that must always be taken. Internally, the unit realizes just two comparisons, deriving other four from them.

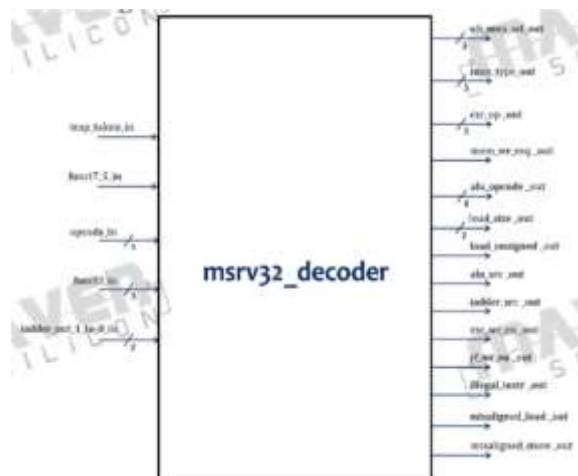
Note: Refer Instruction set format to understand the functionality

2.8.3 Output Waveform



2.9 Decoder

2.9.1 Block Diagram



2.9.2 Functionality

The Decoder decodes the instruction and generates the signals that control the memory, the Load Unit, the Store Unit, the ALU, the two register files (Integer and CSR), the Immediate Generator and the Write back Multiplexer.

Note: The output `imm_type_out` should be generated as per the `imm_generator` module & `wb_mux_sel_out` as per `wb_mux_sel_unit` module.

The output logic for decode is given below.

- a) `alu_opcode_out[2:0]` is assigned to `funct3_in`
- b) `alu_opcode_out[3] = funct7_5_in & ~(is_addi | is_slti | is_sltiu | is_andi | is_ori | is_xori)`
- c) `load_size_out` is asserted for `funct3_in[1:0]` and `load_unsigned_out` is asserted for `funct3_in[2]`
- d) `alu_src_out` is asserted for 5th bit of `opcode_in`
- e) `iadder_src_out` is enabled if either `is_load`, `is_store` or `is_jalr` occurs.
- f) `csr_wr_en_out` is enabled if `is_csr` occurs.
- g) `rf_wr_en_out = is_lui | is_auipec | is_jalr | is_jal | is_op | is_load | is_csr | is_op_imm`
- h) `wb_mux_sel_out`
`wb_mux_sel_out [0] = is_load | is_auipec | is_jal | is_jalr`
`wb_mux_sel_out [1] = is_lui | is_auipec`
`wb_mux_sel_out [2] = is_csr | is_jal | is_jalr`
- i) `imm_type_out`
`imm_type_out [0] = is_op_imm | is_load | is_jalr | is_branch | is_jal`
`imm_type_out [1] = is_store | is_branch | is_csr`
`imm_type_out [2] = is_lui | is_auipec | is_jal | is_csr`
- j) `is_implemented_instr` is enabled if any of the valid opcode occurs.
- k) `csr_op_out` is asserted by `funct3_in`.
- l) `misaligned_load_out` is asserted if `mal_word` or `mal_half` occurs along with `is_load`.
- m) `misaligned_store_out` is asserted if `mal_word` or `mal_half` occurs along with `is_store`.
- n) `mem_wr_req_out` is enabled if `is_store` occurs and `trap_taken_in`, `mal_word` and `mal_half` are low.

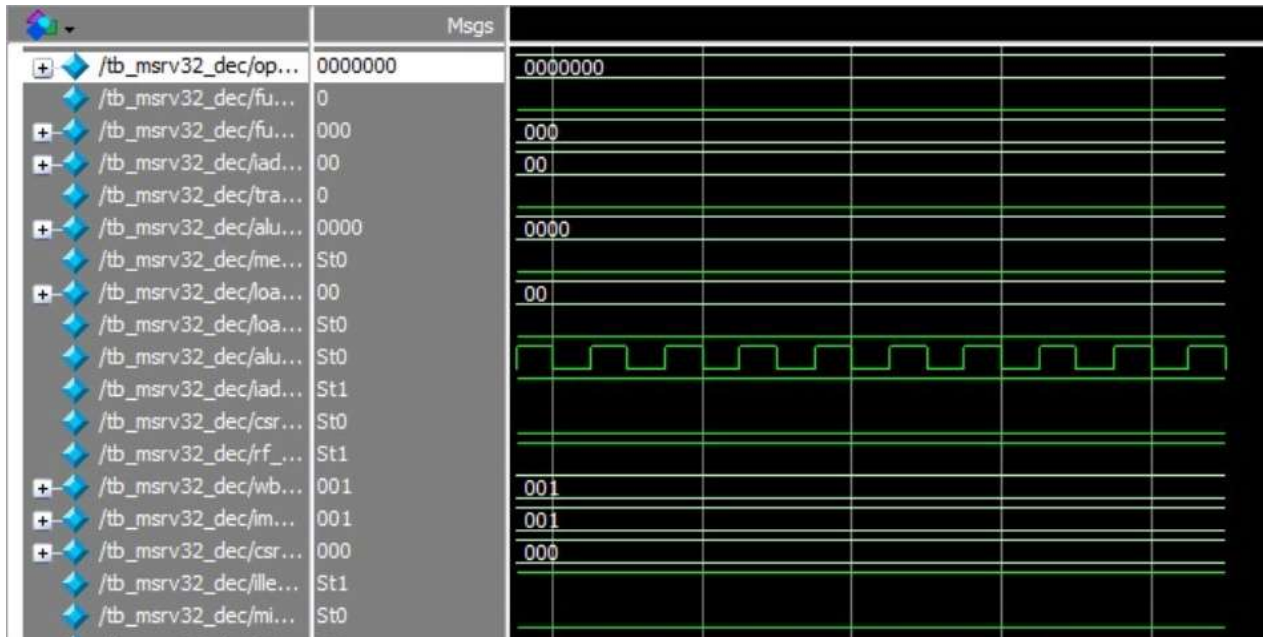
The logic for internal wires are given below.

Internal wire `mal_word` is asserted if `funct3_in` refers to word and if `iadder_1_to_0_in` is not zero.

Internal wire `mal_half` is asserted if `funct3_in` refers to half-word and if `iadder_1_to_0_in [0]` is not zero.

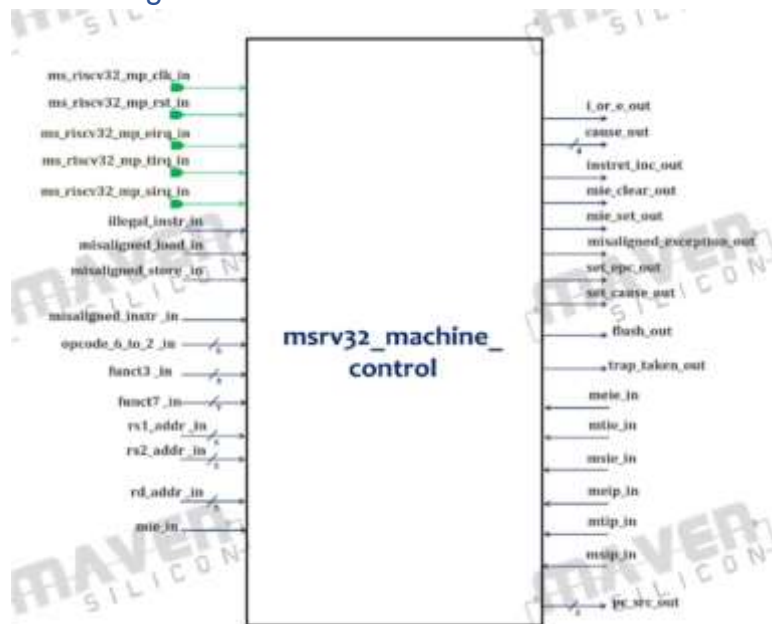
`is_csr = is_system & (funct3_in [2] | funct3_in [1] | funct3_in [0])`

2.9.2 Output Waveform



2.10 Machine Control

2.10.1 Block Diagram



2.10.2 Functionality

The outputs of internal control logic are asserted as per the table

Input	The output to be asserted
Funct7_in = 7'b0001000	funct7_wfi
Funct7_in = 7'b0011000	funct7_mret
opcode_6_to_2_in = 5'b11100	is_system
Funct3_in = 3'b000	funct3_zero
Funct7_in = 3'b0000000	funct7_zero
rs1_addr_in = 5'b00000	rs1_addr_zero
rs2_addr_in = 5'b00000	rs2_addr_zero
rd_addr_in = 5'b00000	rd_zero
rs2_addr_in = 5'b00101	rs2_addr_wfi
rs2_addr_in = 5'b00010	rs2_addr_mret
rs2_addr_in = 5'b00001	rs2_addr_ebreak

Some more outputs logics are given below.

1. ip is asserted when any one of the signal eip, tip, sip is high
2. trap_taken is asserted when either mie & ip are high or any one of the signal exception, ecall, ebreak is high.
3. eip is asserted when meie_in is 1 and e_irq_in or meip_in is 1
4. tip is asserted when mtie_in is 1 and t_irq_in or mtip_in is 1
5. sip is asserted when msie_in is 1 and s_irq_in or msip_in is 1
6. exception is asserted when any one of the signal illegal_instr_in, misaligned_instr_in, misaligned_load_in, misaligned_store_in is high.

is_system	funct7_mret	rs2_addr_mret	rs1_addr_zero	funct3_zero	rd_zero	funct7_zero	rs2_addr_zero	rs2_addr_ebreak	output to be asserted
1	1	1	1	1	X	X	X	X	mret
1	X	X	1	1	1	1	1	X	ecall
1	X	X	1	1	1	1	X	1	ebreak

The Output Synchronous Logic for cause_out & i_or_e_out

These outputs depends on the interrupts as per the below logic under reset condition

cause = 4'b0 & i_or_e = 1'b0

cause & i_or_e in "state_operating" state

if m-mode external interrupt is occurred (eip), then

cause = 4'b1011 & i_or_e = 1'b1

else if m-mode software interrupt is occurred (sip), then

cause = 4'b0011 & i_or_e = 1'b1

else if m-mode timer interrupt is occurred (tip = 1), then

cause = 4'b0111; i_or_e = 1'b1;

else if illegal instruction is occurred (illegal_instr), then

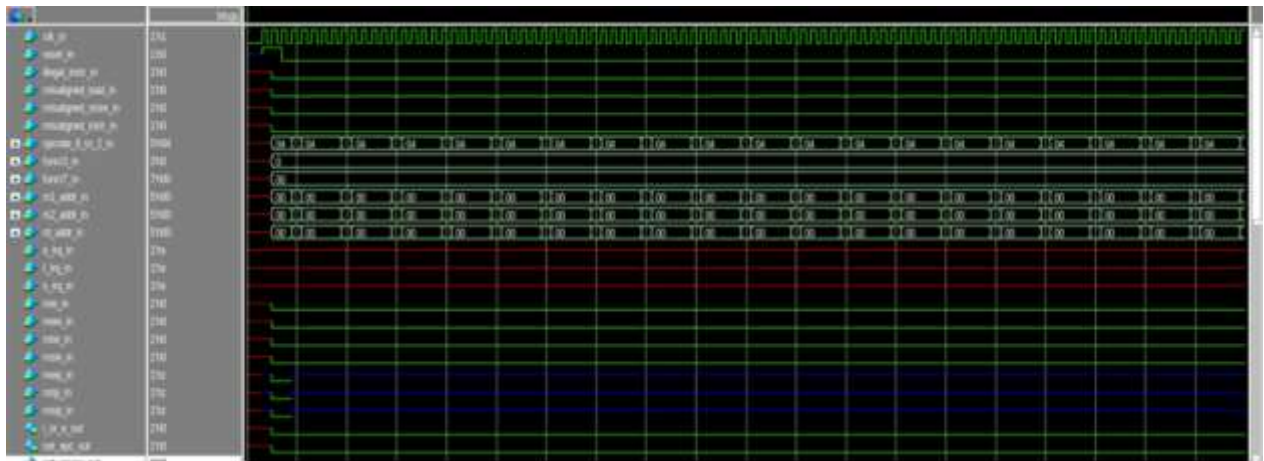
cause = 4'b0010 & i_or_e = 1'b0

else if the instruction address is misaligned (misaligned_instr)

cause = 4'b0000 & i_or_e = 1'b0;

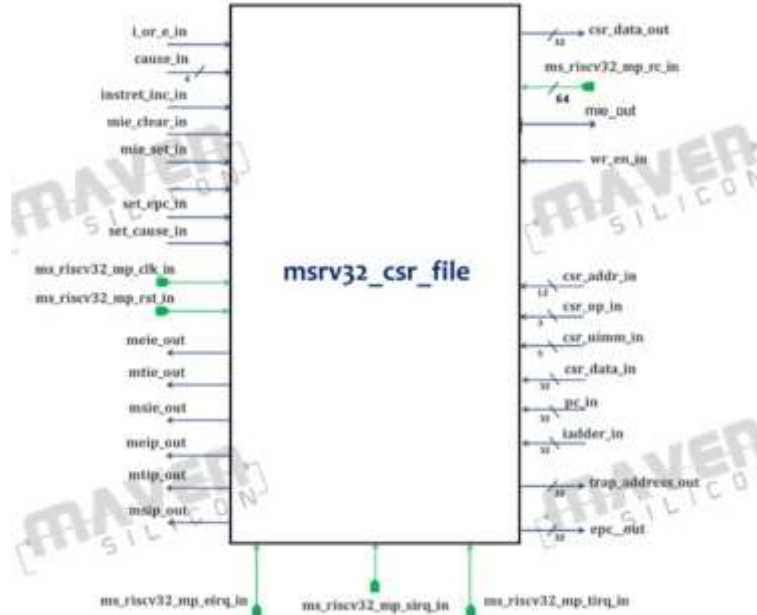
else if environment call from m-mode (ecall), then

The `misaligned_exception_out` is asserted in synchronous with the clock only when any of `misaligned_instr`, `misaligned_load`, `misaligned_store` inputs are 1.



2.11 CSR File

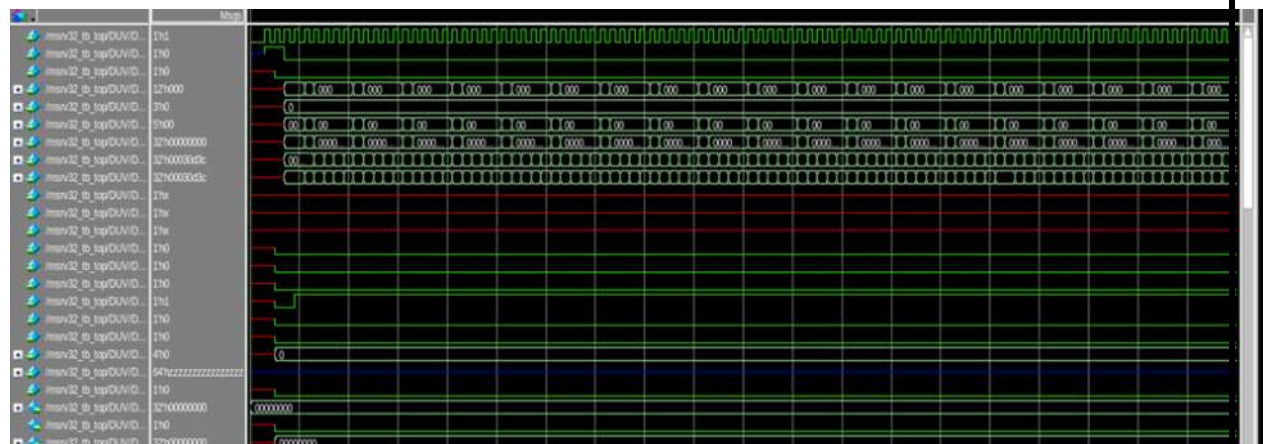
2.11.1 Block Diagram



2.11.2 Functionality

The CSR Register File has the control and status registers required for the implementation of M-mode. Read/Write, set and clear operations can be applied to the registers.

2.11.3 Output Waveform



2.12 Reg Block 2

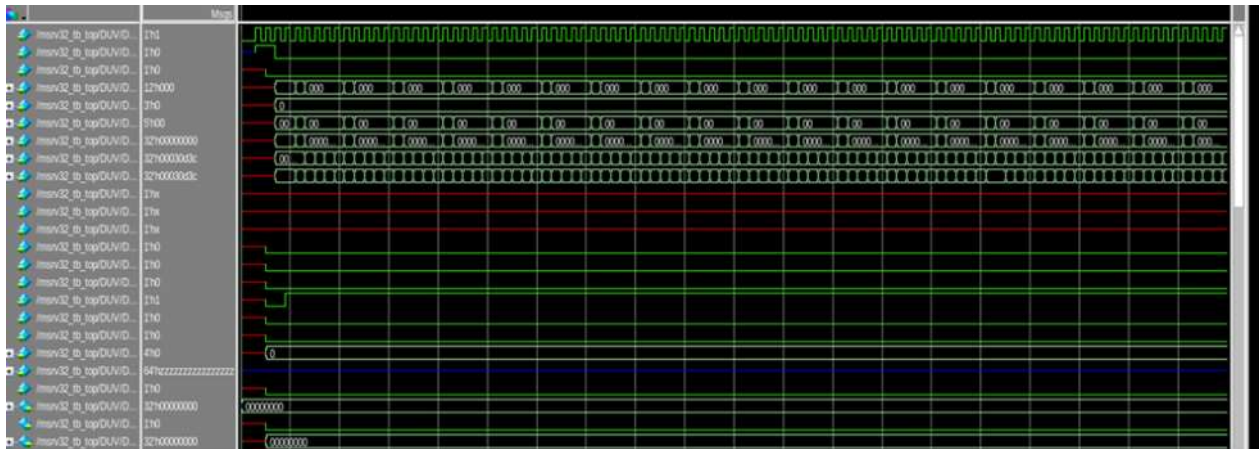
2.12.1 Block Diagram



2.12.2 Functionality

It registers all the inputs and produces the outputs at the posedge of clk if there is no reset. The block also integrates a 2:1 MUX with select-line as **branch_taken_in**. The LSB of iaddr_out_reg_out is assigned with 0 if branch_taken_in is 1, else iaddr_out_reg_out [0] is assigned with registered value of iaddr_in [0].

2.12.3 Output Waveforms:



2.13 Store Unit

2.13.1 Block Diagram



2.13.2 Functionality

The Store Unit drives the signals that interface with the external data memory.

It places the data to be written (which can be a byte, half word or word) in the right position in `data_out` and sets the value of `wr_mask_out` in an appropriate way.

1. Depending on the funct3_in signal & ahb_ready_in, data_out signal will get values.

If data_hready_in is high, then if funct3_in=2'b00 a byte of data will be stored in data_out depending on the iaddr_in [1:0].

For ex. If iaddr_in=2'b00 then data_out = {8'b0, 8'b0, 8'b0, rs2_in[7:0]}.

iaddr_in = 2'b01 then data_out = {8'b0, 8'b0, rs2_in[15:8], 8'b0},

For funct3_in=2'b01, a half word of data will be stored in data_out (depending on the iaddr_in [1]. For ex. iaddr_in [1] = 1'b1 then data_out = {rs2_in [31:16], 16'b0} and for other combinations rs2_in will be stored in data_out.

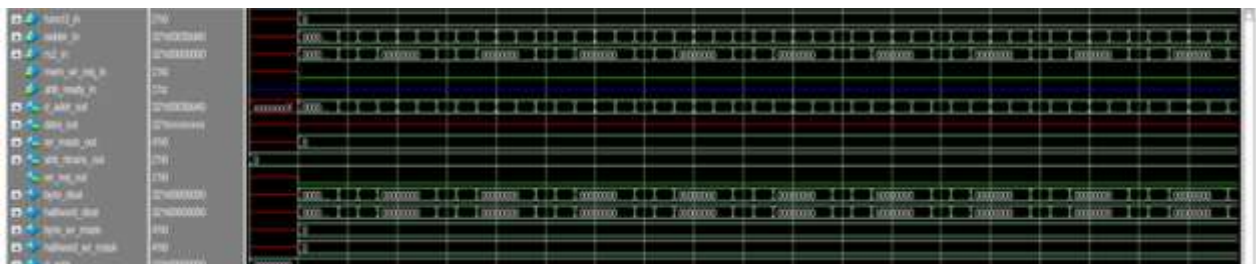
Depending on the funect3_in signal wr_mask_out signal will get values.

If funct3_in=2'b00, depending on the iaddr_in [1:0], i.e. if iaddr_in=2'b01 then wr_mask_out={2'b0, mem_wr_req_in, 1'b0}

If `func3_in=2'b01`, depending on the `iaddr_in [1]` i.e. if `iaddr_in [1] =1'b1` then `wr_mask_out= ((2{mem_wr_req_in}), 2'b0)` and for other combinations word of data (`(4{mem_wr_req_in})`) will be stored.

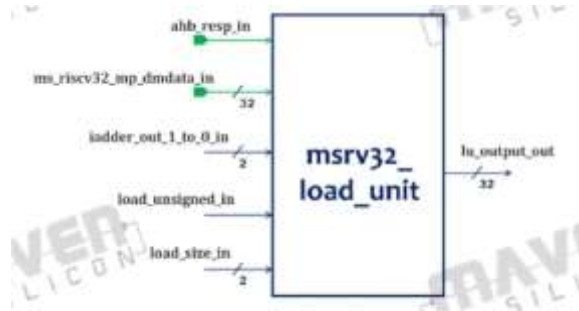
3. The `dm_adder_out` should be aligned with respect to `{iaddr_in [31:2], 2'b00}`.
4. The `wr_req_out` should be aligned with respect to `mem_wr_req_in`.
5. The `ahb_htrans_out` is `2'b10` during a valid store instruction and is `2'b00` when the transfer is completed.

2.13.3 Output Waveform



2.14 Load Unit

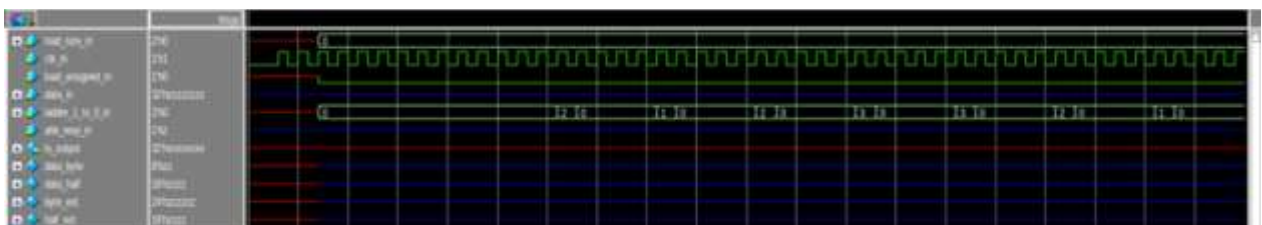
2.14.1 Block Diagram



2.14.2 Functionality

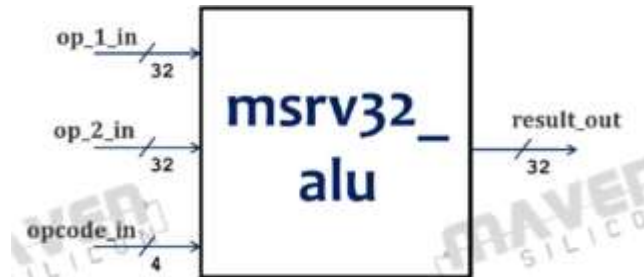
The Load Unit reads the **data_in** input signal and forms a 32-bit value based on the load instruction type (encoded in the **funct3** field). The formed value (placed on output) can then be written in the Integer Register File. The module input and output signals are shown in the above table. The value of output is formed as shown in the above table.

2.14.3 Output Waveform



2.15 ALU

2.15.1 Block Diagram



2.15.2 Functionality

The ALU applies ten distinct logical and arithmetic operations in parallel to two 32-bit operands, outputting the result selected by opcode_in. The ALU input/output signals and the opcodes are shown in tables below.

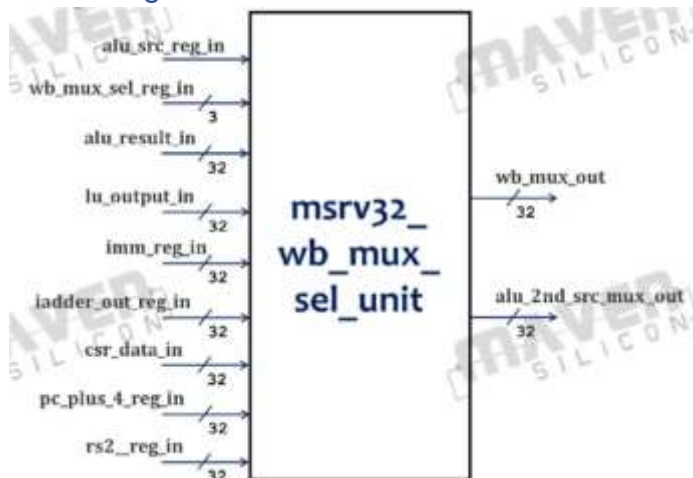
The opcode values were assigned to facilitate instruction decoding. The most significant bit of opcode_in matches with the second most significant bit in the instruction funct7 field. The remaining three bits match with the instruction funct3 field.

2.15.3 Output Waveform



2.16 WB Mux Selection Unit

2.16.1 Block Diagram



2.16.2 Functionality

If 'alu_src_reg_in' is high then 'alu_2nd_src_mux_out' = 'rs2_reg_in' else 'alu_2nd_src_mux_out' = 'imm_reg_in'.

Based on the 'wb_mux_sel_reg_in' output signal 'wb_mux_out' will be assigned

- If 'wb_mux_sel_reg_in' = WB_ALU then 'wb_mux_out' = 'alu_result_in'.
- If 'wb_mux_sel_reg_in' = WB_LU then 'wb_mux_out' = 'lu_output_in'.
- If 'wb_mux_sel_reg_in' = WB_IMM then 'wb_mux_out' = 'imm_reg_in'.
- If 'wb_mux_sel_reg_in' = WB_IADDER_OUT then 'wb_mux_out' = 'iadder_out_reg_in'.
- If 'wb_mux_sel_reg_in' = WB_CSR then 'wb_mux_out' = 'csr_data_in'.
- If 'wb_mux_sel_reg_in' = WB_PC_PLUS then 'wb_mux_out' = 'pc_plus_4_reg_in'.
- For remaining combination 'wb_mux_out' = 'alu_result_in'.

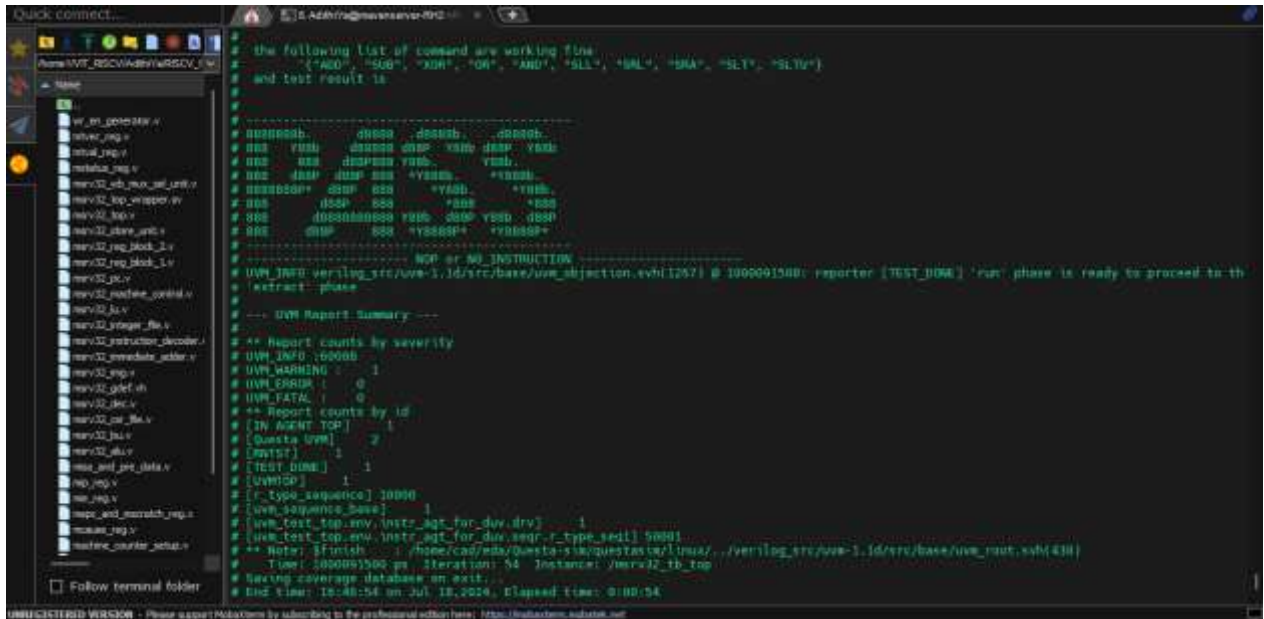
wb_mux_sel_reg_in value , description and the instruction type is given below.

wb_mux_sel_reg_in Values	Description	Instruction type
000	WB_MUX	ALU_output(R_type, I_type)
001	WB_LU	LOAD_UNIT
010	WB_IMM	LU(immediate value)
011	WB_IADDER_OUT	AUIPC(RS1+IMM/PC+IMM)
100	WB_CSR	CSR
101	WB_PC_PLUS	PC+4(for jal, jalr)

2.16.3 Output Waveform



UVM TEST RESULT:



```
Quick connect...
/home/UT_RSQVIA/ut_rsqv/ut_rsqv.v
# the following list of command are working fine
# ("ADD", "SUB", "XOR", "OR", "AND", "SL", "SRL", "SRA", "SLT", "SLTU")
# and test result is
#
# -----
# 0000000b, 0000, 00000b, 00000b,
# 000 000b 00000b 000b 000b 000b
# 000 000 00000b 000b, 000b,
# 000 000b 000b 000b 00000b, 00000b,
# 0000000b* 000b 000b 000b, 000b,
# 000 000b 000b 000b 000b 000b
# 000 000000000b 000b 000b 000b 000b
# 000 000b 000b 000b 00000b* 00000b*
#
# -----
# UVM INFO: Verilog src/uvw-1.1d/src/base/uvw_objection.vvh(1257) @ 1000001500: reporter [TEST_DONE] 'run' phase is ready to proceed to th
# extract phase
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM INFO: 0000b
# UVM WARNING: 1
# UVM ERROR: 0
# UVM FATAL: 0
#
# ** Report counts by id
# [IN AGENT TOP] 1
# [Queue UVM] 2
# [XVIST] 1
# [TEST_DONE] 1
# [UVMTOP] 1
# [r_type_sequence] 1000b
# [uvw_sequence_base] 1
# [uvw_test_top.env.instr_sgt_for_duv.drv] 1
# [uvw_test_top.env.instr_sgt_for_duv.sqr.r_type_sedi] 00001
#
# ** Note: $finish ... /home/cad/eda/Questasim/questasim/linux/.../verilog_src/uvw-1.1d/src/base/uvw_rout.vvh(438)
# Time: 1000001500 ps, Iteration: 54, Instance: /uvw32_tb_top
# Saving coverage database on exit...
# End time: 15:48:54 on Jul 15, 2024, Elapsed time: 0:00:54
```