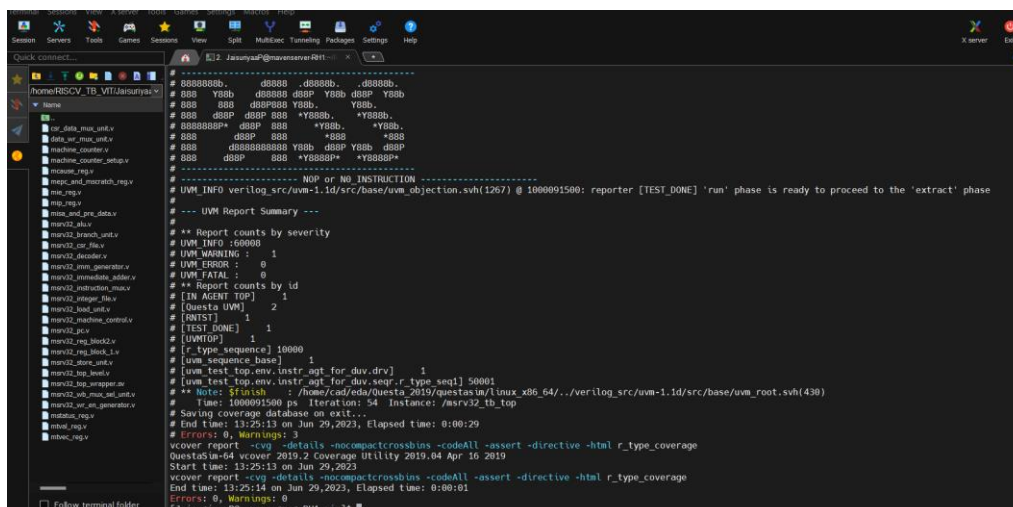


In RISV processor we will be executing all the instructions parallel which means we can execute most of the instructions at the same time. From Program Counter we will find the address of the instruction to be executed now. Now that current instruction which is to be executed is given to the instruction memory. From there the instruction is given to the control memory which helps to find what type of instruction is it, what operation to be executed based on funct3, funct7 and opcode and decoder which finds the address of the source register 1 and 2 and the destination register. Now the address of rs1, rs2 and

destination register(rd) is given to the reg block where the value of rs1,rs2 and rd will be stored based on the values passed which is of 5 bit value. After passing through that block we will have rs1,rs2 and rd values as 32 bits. Now the value of rs1 is passed to the ALU unit and rs2 value is passed to the Instruction mux. Now to that instruction mux rs2, $IMMI = sxt(INSTR[31:20])$, $IMMS = \{INST[31:25], INST[11:7]\}$ is passed. Now based on the type of instruction and operation to be performed which was generated by the control logic the value will be send to ALU unit from the instruction mux . If the instruction is of R-type, then the rs2 register value will be passed to ALU unit from the instruction mux and then arithmetic and logical operations will be performed based on funct 3 and funct7 binary values between the rs1 and rs2 and then the result will be passed to the Write back mux and then the write mux will select that instruction and then write the value to the destination register in the reg file block by making write enable as 1. If the instruction is of I type, then the constant value which is of $IMMI = sxt(INSTR[31:20])$ is passed to the ALU unit from Instruction mux and arithmetic and logical operations will be performed based on funct3 and imm[11:5] between the rs1 and IMMI and then the result will be passed to the Write back mux and then the write mux will select that instruction and then write the value to the destination register in the reg file block by making write enable as 1. If the instruction is of load type which is a I type instruction then the constant value which is of $IMMI = sxt(INSTR[31:20])$ is passed to the ALU unit from Instruction mux and then addition operation will be performed between rs1 and IMMI and that result will given to the data memory which enables the read operation and calculates the address of the memory where the values are stored. Now that values will be passed to the write back mux and then the write back mux will enable that operation and then the result will be stored in the destination register which is present inside the reg file by making write enable operation as 1. If the instruction is of store type then the value $IMMS = \{INST[31:25], INST[11:7]\}$ will be passed to the ALU unit from the instruction mux and then the addition operation will be performed between rs1 and IMMS and that result will be given to the data memory which enables the write operation. Here rs2 value will be passed to the data memory. The value which was present in the rs2 is given to the rs1+IMMS address and finally that value is stored in that address. Here we will not enable write back mux. Here write enable is equal to 0 so that we will not write the value to the destination register. If the instruction is of branch type then the rs2 value will be passed to the ALU unit from the instruction mux and the various operations will be performed between rs1 and rs2 based on funct3 and INST[30] and the result will be stored in BT. If the value of BT=0, then pc is equal to pc+4 or else pc=pc+IMMB. These operations will be

performed with the help of control logic. $BT=1$ if the given condition of operations between $rs1$ and $rs2$ is satisfied or else $BT=0$. Here the write back mux value is not enabled and write enable value=0 so that no value is written to the destination register. If the instruction is JAL then the value $rd=pc+4$ and that value will be stored in the destination register by making write enable=1 and the value of pc will be incremented by IMMJ. The value of $IMMJ=\{SXT(imm[20:1]), 1'b0\}$. Here the jump will be happen upto 2^{20} address since it is of 20 bits. Here we can do only short jump and long jump is not possible. If we want to do long jump then the instruction must be JALR. Here the value of $rd=pc+4$ and then that value will be written to the reg file by making write enable as 1. Here the value of $pc=\{(rs1+IMMI), 1'b0\}$. Here the value of $IMMI=SXT(imm[11:0])$. Here the value of pc will be incremented as above mentioned and then value of rd will be loaded to reg file at the same time. If the instruction is LUI then the value $rd=IMMU$ will be written into the reg block by making write enable as 1. If the instruction is AUIPC then the value $rd=pc+IMMU$ will be written into the reg block by making write enable as 1. The above mentioned operations will be controlled by the control unit. The top module handles exceptions and interrupts that may occur during program execution. It monitors external events and initiates the necessary actions to handle exceptions, such as interrupting the current instruction flow and transferring control to the appropriate exception handling routines. If the RISC processor implements a pipelined architecture, the top module manages the pipeline stages and ensures the proper flow of instructions through the pipeline. It handles hazards and stalls to maintain the correct execution order and data dependencies. Overall, the top module of a RISC processor plays a crucial role in coordinating the different functional units and control signals to execute instructions and manage the processor's operation effectively.

Output:



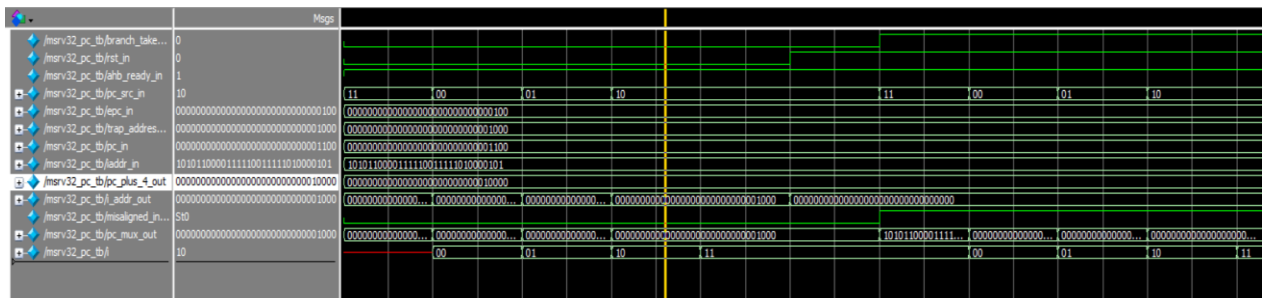
```

/home/RISCV_TB_VT/jaisurya@
# 888  Y88b  d8888  d8888b  d8888b
# 888  888  d88P888  Y88b  Y88b
# 888  d88P  d88P 888  *Y88b  *Y88b
# 888888P* d88P 888  *Y88b  *Y88b
# 888  d88P 888  *888  *888
# 888  d8888888888  d88P Y88b  d88P
# 888  d88P 888  *Y888P* *Y888P*

--- NOP or NO_INSTRUCTION ---
UVM INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 10000091500: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase

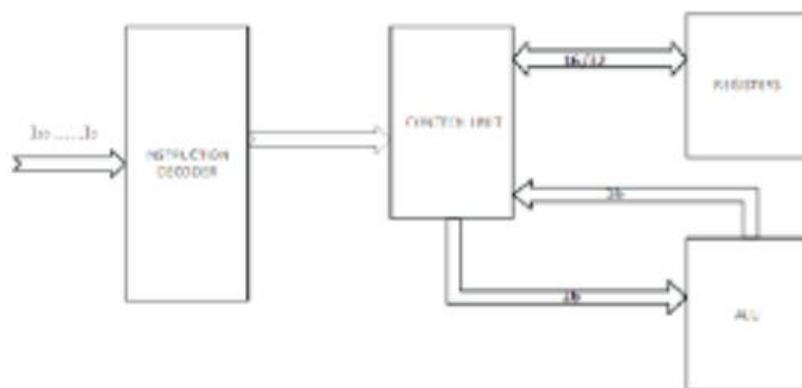
--- UVM Report Summary ---
** Report counts by severity
UVM INFO 160000
UVM WARNING 1
UVM ERROR 0
UVM FATAL 0
** Report counts by id
[IN AGENT TOP] 1
[Questa UVM] 2
[RNTST] 1
[TEST_DONE] 1
[UWNTOP] 1
[r_type_sequence] 10000
[uvm_sequence_base] 1
UvmTestTopEnvUnstrAgntForDivDrv 1
UvmTestTopEnvUnstrAgntForDivSeqRTypeSeq 50001
** Note: $finish : /home/cad/eda/questa_2019/questasim/linux_x86_64/.../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
time: 1000091500 ps iteration: 34 Instance: /asrv32_tb_top
Saving coverage database on exit...
End time: 13:25:13 on Jun 29, 2023, Elapsed time: 0:00:29
Errors: 0, Warnings: 3
vcover report -cvg -details -nocompactcrossbins -codeAll -assert -directive -html r_type.coverage
Questasim-64 vcover 2019.2 Coverage Utility 2019.04 Apr 16 2019
Start time: 13:25:13 on Jun 29, 2023
vcover report -cvg -details -nocompactcrossbins -codeAll -assert -directive -html r_type.coverage
End time: 13:25:14 on Jun 29, 2023, Elapsed time: 0:00:01
Errors: 0, Warnings: 0
[jaisurya@navenserver-RH1 ~]$

```

REG Block:

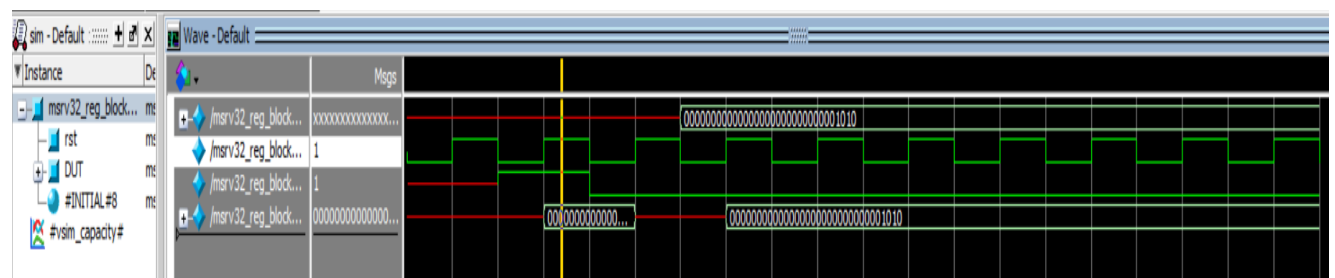
Block Diagram:



Functionality:

If the reset is high ,the output is zero else the output will based on the present value of pc_mux_in which is based on the posedge of the clock.

Output waveform:

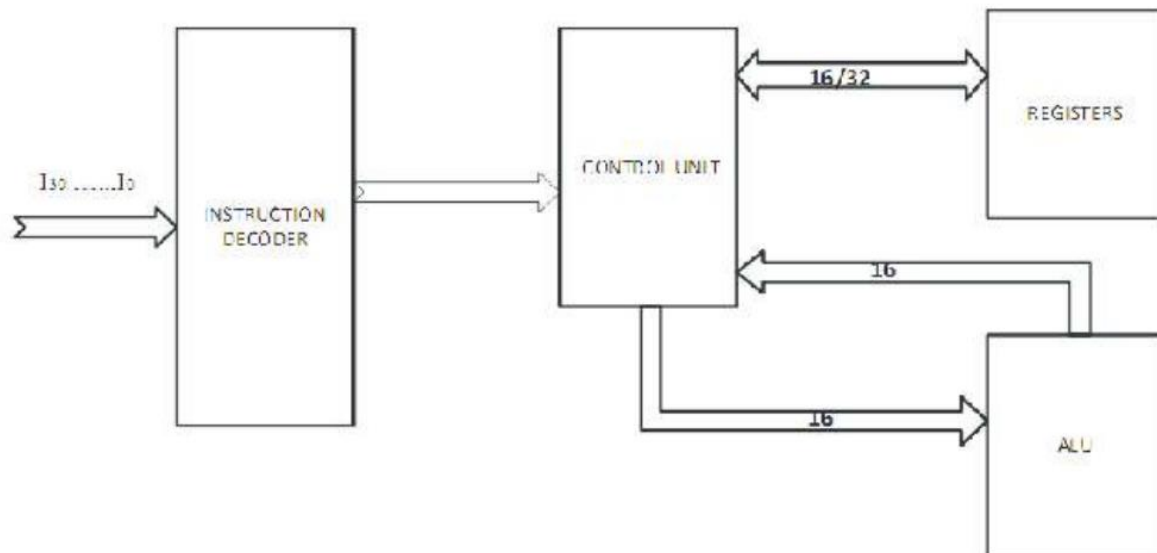


```

pc_in=      x,pc_out=      x
pc_in=      x,pc_out=      0
pc_in=      x,pc_out=      x
pc_in=      10,pc_out=      x
pc_in=      10,pc_out=      10

```

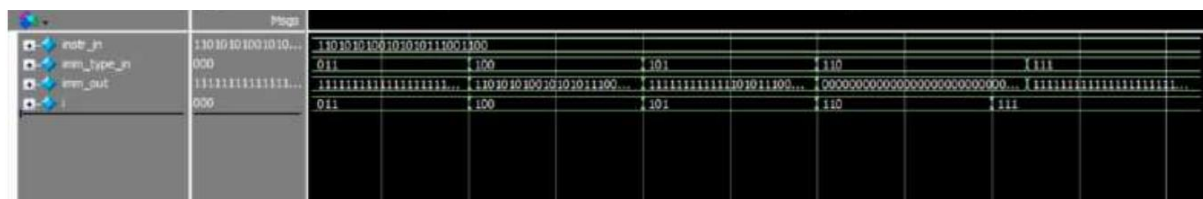
Block Diagram:



Functionality:

It rearranges the immediate bit contained in the instruction and if necessary sign-extend it to a 32-bit value. By using opcode [6:4] we can distinguish different types of instruction and find the type of instruction to be executed. The remaining opcode bit [3:0] is useless.

Output Waveform:



```

SDM 7> run -all
Instruction without opcode [31:7] = 1101010100101010111001100
opcode place = xxxxxxxx

Complete Instruction = 11010101001010101110011000000000
opcode = 000 -- immediate_address = 1111111111111111110101010010

Complete Instruction = 11010101001010101110011000000001
opcode = 001 -- immediate_address = 111111111111111111110101010010

Complete Instruction = 11010101001010101110011000000010
opcode = 010 -- immediate_address = 11111111111111111111110101001100

Complete Instruction = 11010101001010101110011000000011
opcode = 011 -- immediate_address = 11111111111111111111010101001100

Complete Instruction = 110101010010101011100110000000100
opcode = 100 -- immediate_address = 11010101001010101110000000000000

Complete Instruction = 110101010010101011100110000000101
opcode = 101 -- immediate_address = 1111111111110101010100101010010

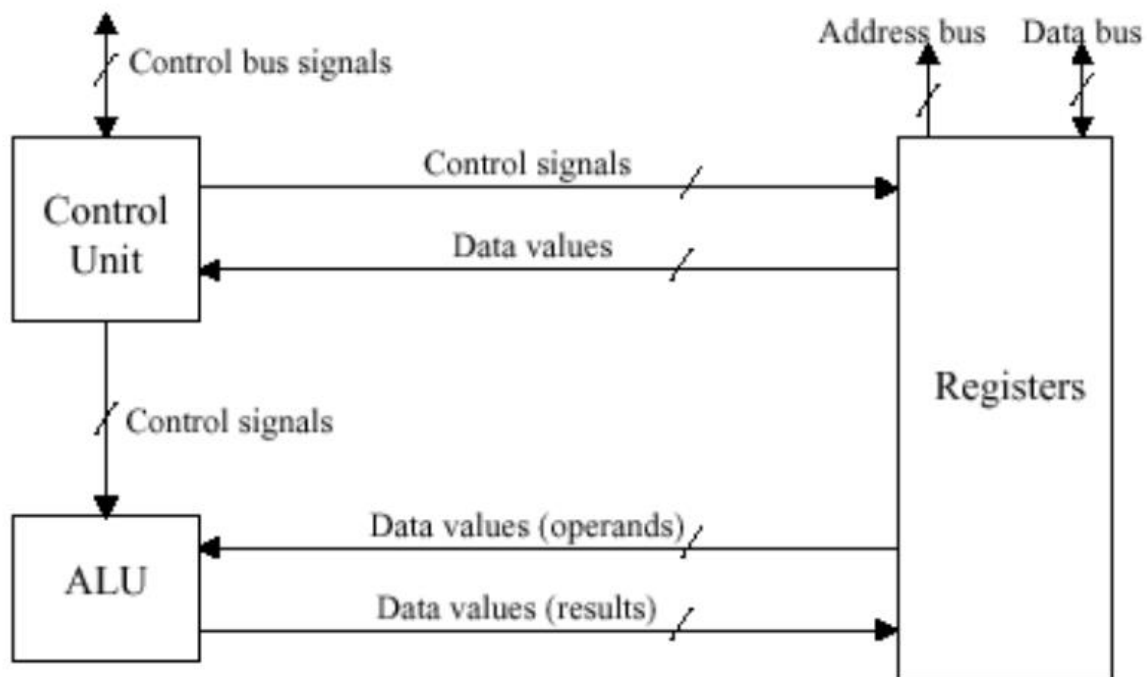
Complete Instruction = 110101010010101011100110000000110
opcode = 110 -- immediate_address = 000000000000000000000000000010101

opcode = 111 -- immediate_address = 111111111111111111111101010100101

```


Immediate Adder:

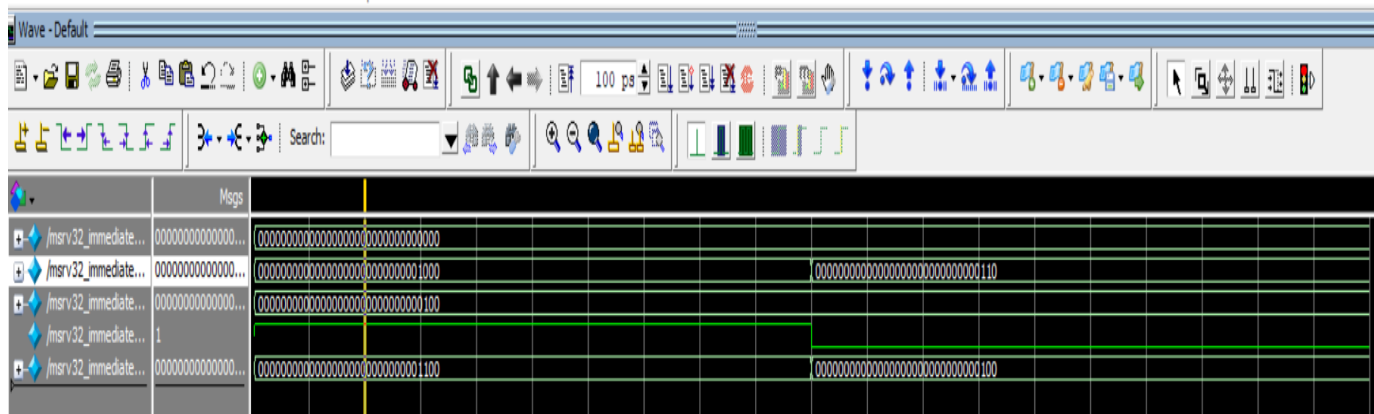
Block Diagram:



Functionality:

If `iadder_src_in` is high then `iadder_out` is the sum of source register and `imm_in` else `iadder_out` is the sum of `pc_in` and `imm_in`. Here the output `iadder_out` and the input `pc_in` and `rs_1_in` are of 32 bits

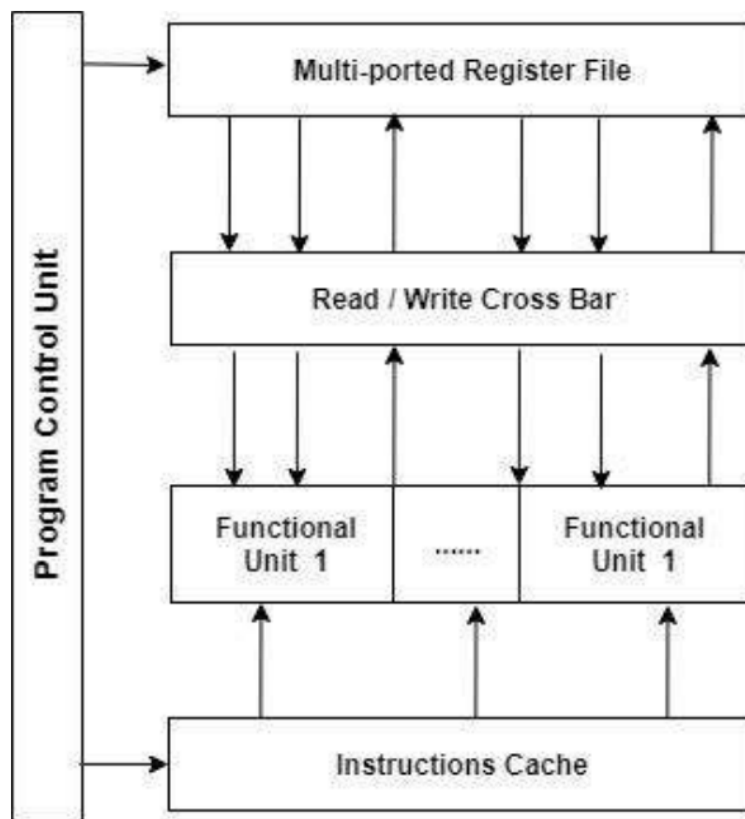
Output Waveform:



```
pc_in=00000000000000000000000000000000,rs_1_in=00000000000000000000000000000000,imm_in=00000000000000000000000000000000,iadder_out=00000000000000000000000000000000
pc_in=00000000000000000000000000000000,rs_1_in=00000000000000000000000000000000,imm_in=00000000000000000000000000000000,iadder_out=00000000000000000000000000000000
```

Integer File:

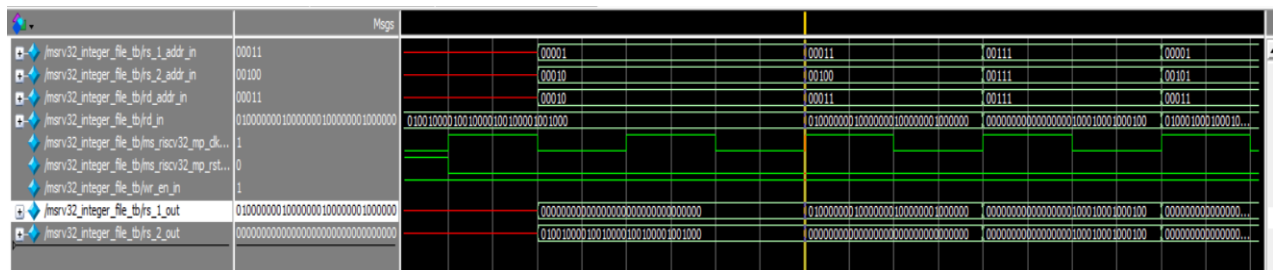
Block Diagram:



Functionality:

The msrv32 integer file has 32 general-purpose registers and supports read and write operations. Initially all registers values are 0. At R0 no write operation is performed because it is hardwired to 0. 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. 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. If `rs_1_addr_in` or `rs_2_addr_in` is equal to `rd_addr_in` and `wr_en_in` is high then `rs_1_out` is equal to `rd_addr_in` or `rs_2_out` is equal to `rd_addr_in` else `rs_1_out` is equal to the corresponding rs1 register address value or `rs_2_out` is equal to the corresponding rs2 register address value.

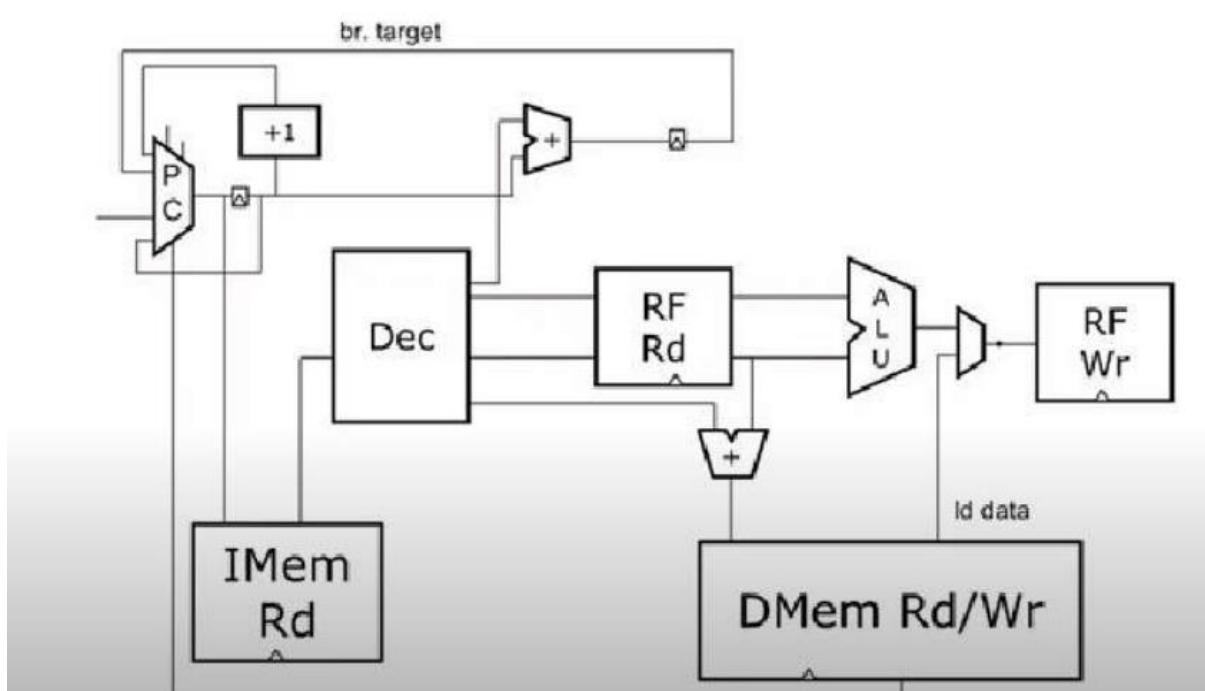
Output Waveform:



```
# The rs1 out is = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, The rs2 out is = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# The rs1 out is = 00000000000000000000000000000000, The rs2 out is = 01001000010010000100100001001000
# The rs1 out is = 0100000001000000001000000010000000, The rs2 out is = 00000000000000000000000000000000
# The rs1 out is = 0000000000000000000100010001000100, The rs2 out is = 0000000000000000000100010001000100
# The rs1 out is = 0000000000000000000000000000000000, The rs2 out is = 0000000000000000000000000000000000
# ** Note: $finish      : C:/intelFPGA/17.1/Jaimaven/msrv32_integer_file_tb.v(58)
# Time: 110 ps  Iteration: 0  Instance: /msrv32_integer_file_tb
# 1
```

Write Enable block:

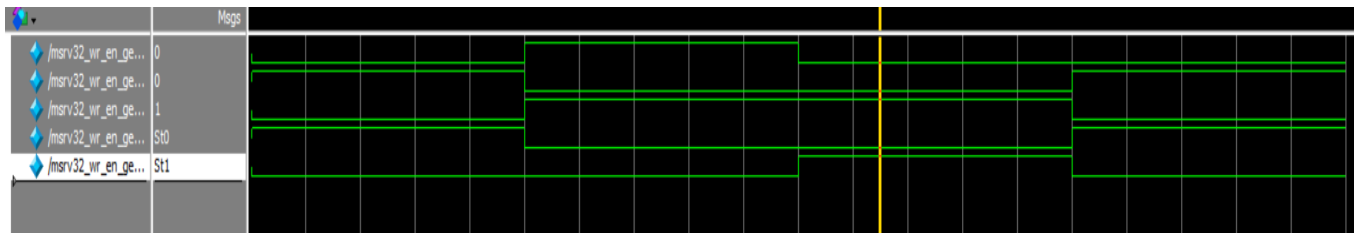
Block Diagram:



Functionality:

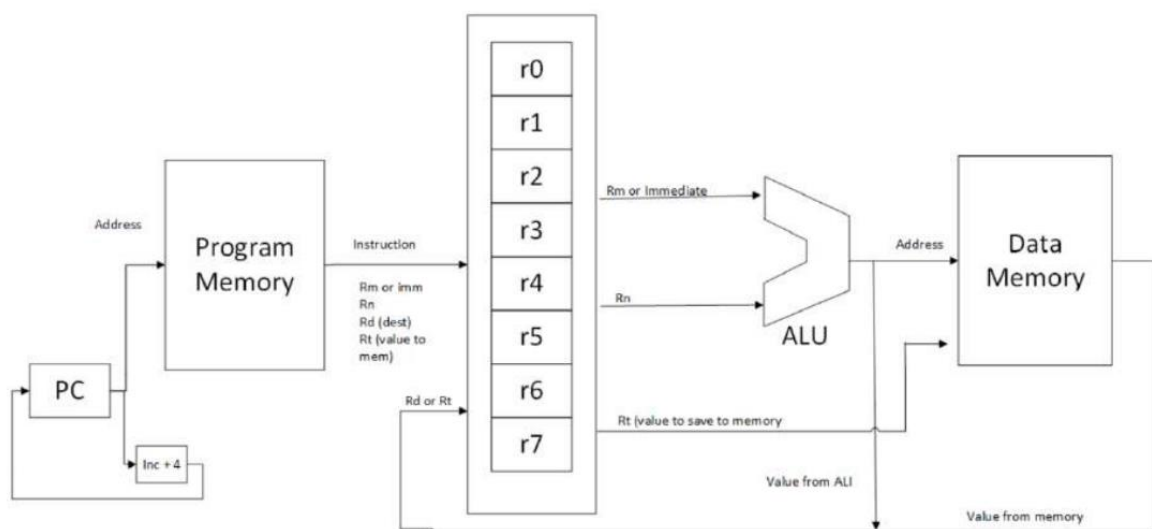
It is important for any instruction to write the value to the register. If flush_in is high then wr_en_csr_file_out and wr_en_integer_file_out is equal to 0 and no values will be written to the destination register else wr_en_csr_file_out = csr_wr_en_reg_in and wr_en_integer_file_out = rf_wr_en_reg_in and the corresponding values will be written into the destination register.

Output Waveform:



Instruction Mux:

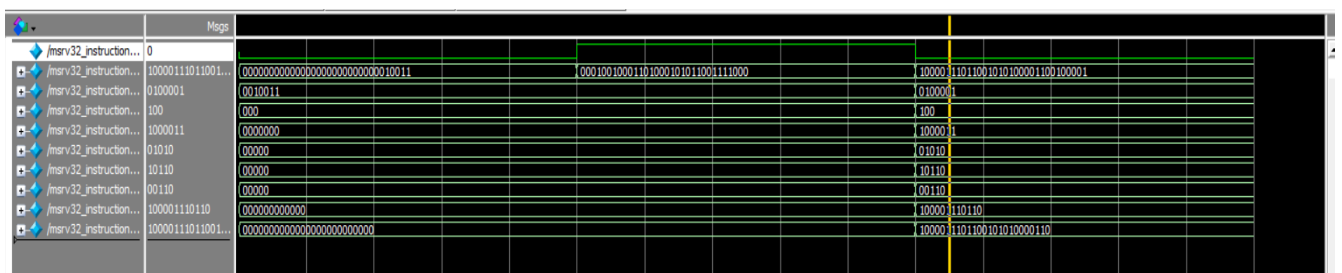
Block Diagram:



Functionality:

It takes the instruction and provides the field for the other modules to perform operations. If flush is high then 32'h00000013 is provided to the field and the slicing of bits will happen based on what operation and instruction to be performed else the input instruction will be given to the field and slicing will happen based on what operation and instruction to be performed.

Output Waveform:



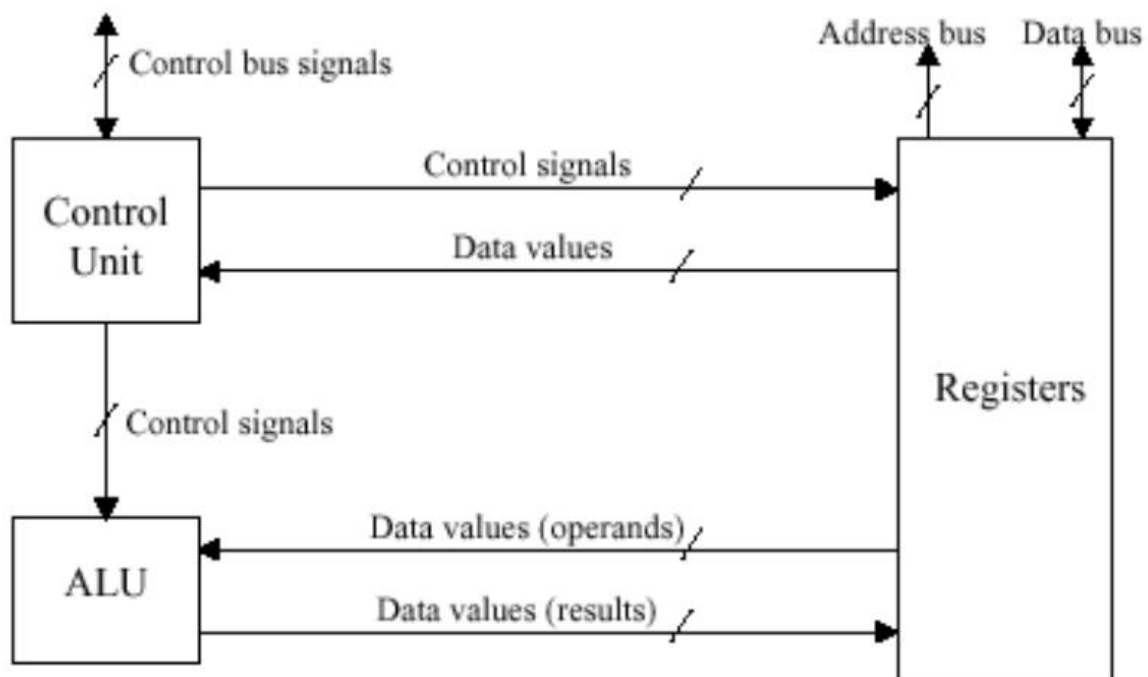
```

# opcode_out = 13
# funct3_out = 0
# funct7_out = 00
# rsladdr_out = 00
# rs2addr_out = 00
# rdaddr_out = 00
# csr_addr_out = 000
# instr_out = 0000000
# opcode_out = 21
# funct3_out = 4
# funct7_out = 43
# rsladdr_out = 0a
# rs2addr_out = 16
# rdaddr_out = 06
# csr_addr_out = 876
# instr_out = 10eca86
# ** Note: $finish      : C:/intelFPGA/17.1/Jaimaven/msrv32_instruction_mux_tb.v(71)
#   Time: 30 ps  Iteration: 0  Instance: /msrv32_instruction_mux_tb

```

Branch Unit:

Block Diagram:

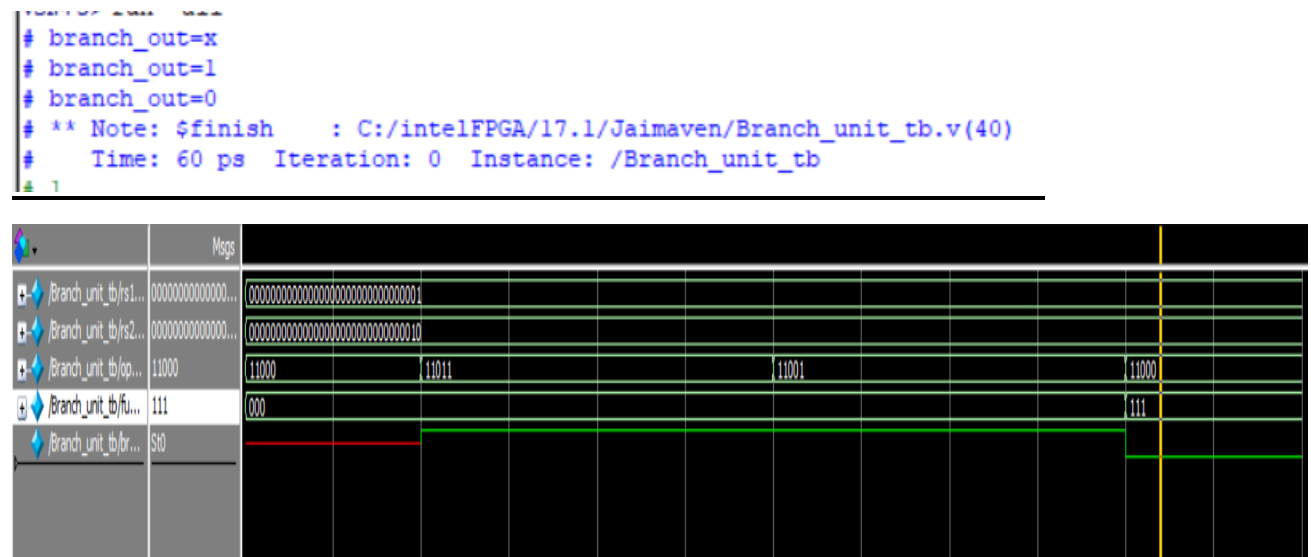


Functionality:

It is used to check whether the branch operation is to be executed or not based on opcode[6:2]. Using source register 1 and 2 different types of operations of branch will be executed by using funct3 and opcode[6:2] and result will be stored in branch_taken_out. If other instructions like JAL and

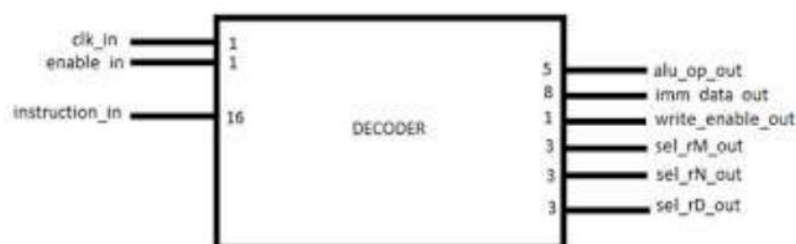
JALR to be executed the value of branch_taken_out is equal to 1.If it is not branch, JAL and JALR type instruction then the value of branch_taken_in=0.

Output Waveform:



Decoder:

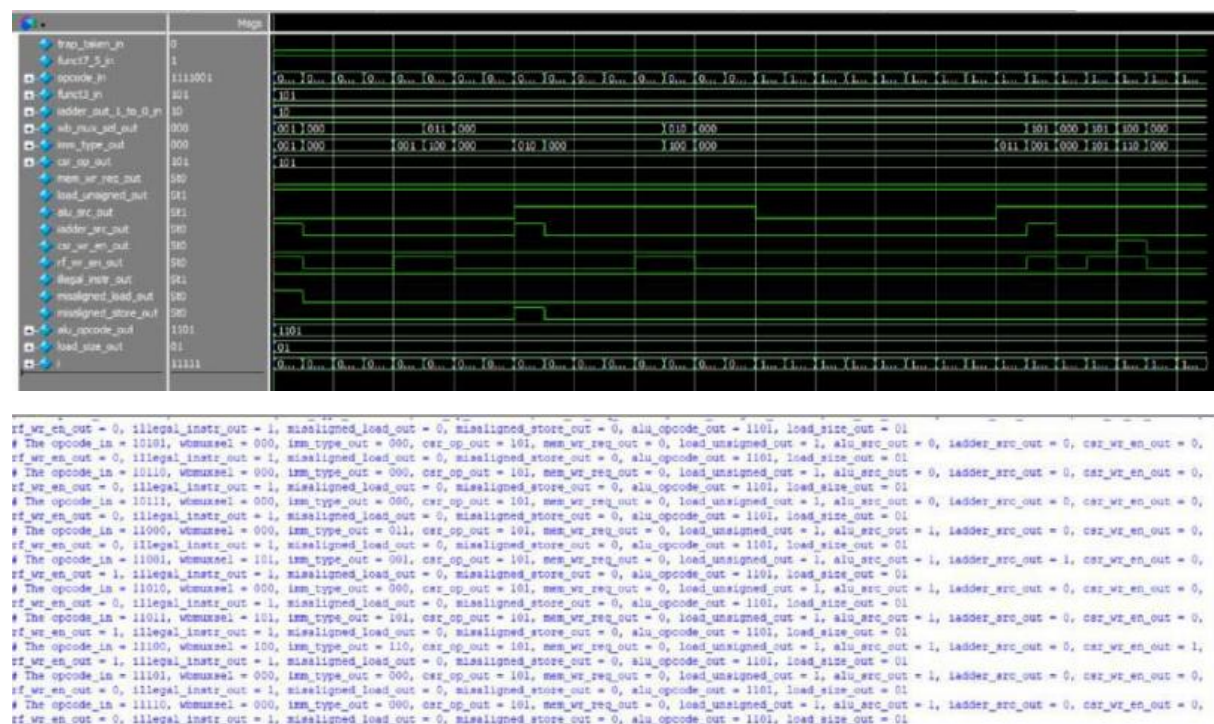
Block Diagram:



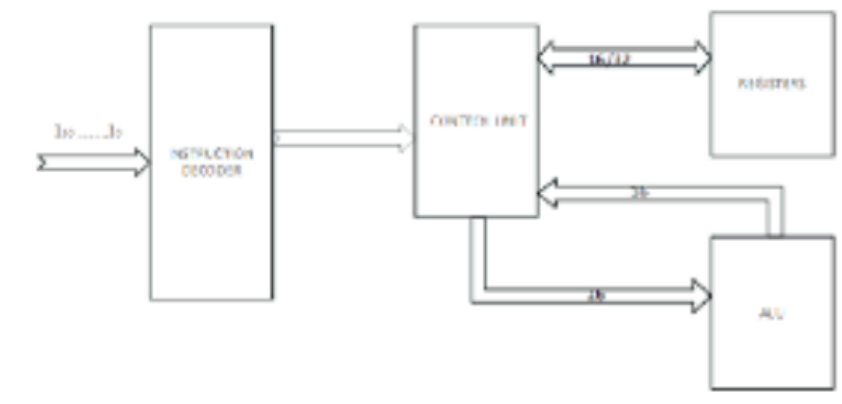
Functionality:

Decoder decodes the instruction and generate the signal output which controls memory unit,load unit, store unit , Arithmetic and Logical operations , 2 register files(Integer and CSR), the immediate generator and the Write back Multiplexer. 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.

`Alu_opcode_out[2:0]` is assigned to `funct3`,`alu_opcode_out[3]` = `funct7_5_in&~(is_addi| is_sltiu | is_andi | is_ori | is_xori)` `load_size_out` is asserted for `funct3_in[1:0]` and `load_unsigned_out` is assted for `funct3_in[2]`. `Alu_src_out` is asserted for 5th bit of `opcode_in`. `iadder_src_out` is enabled if either `is_load`,`is_store` or `is_jalr` occurs. `Csr_wr_en_out` is enabled if `is_csr` occurs.



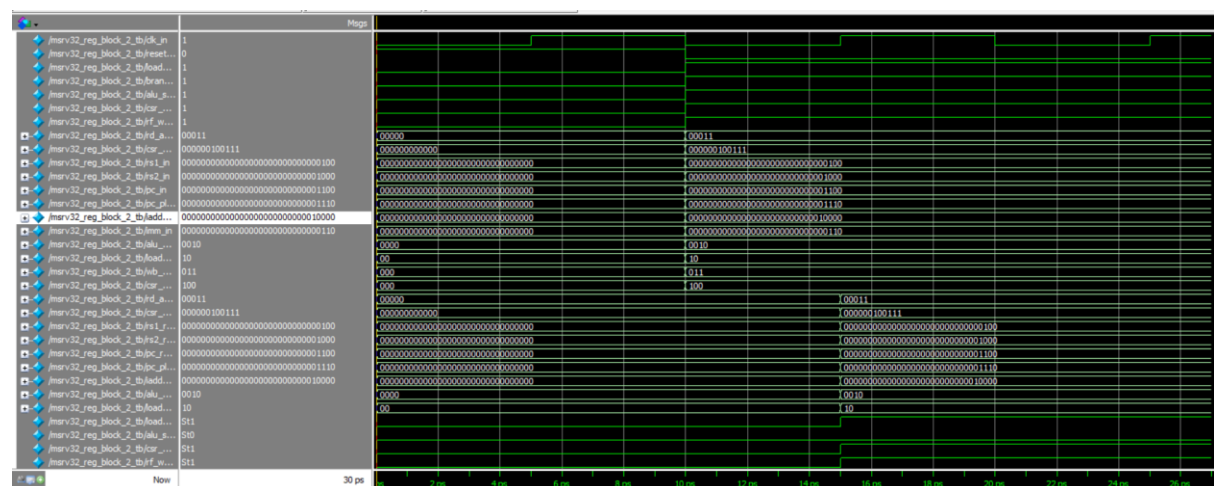
Block Diagram:



Functionality:

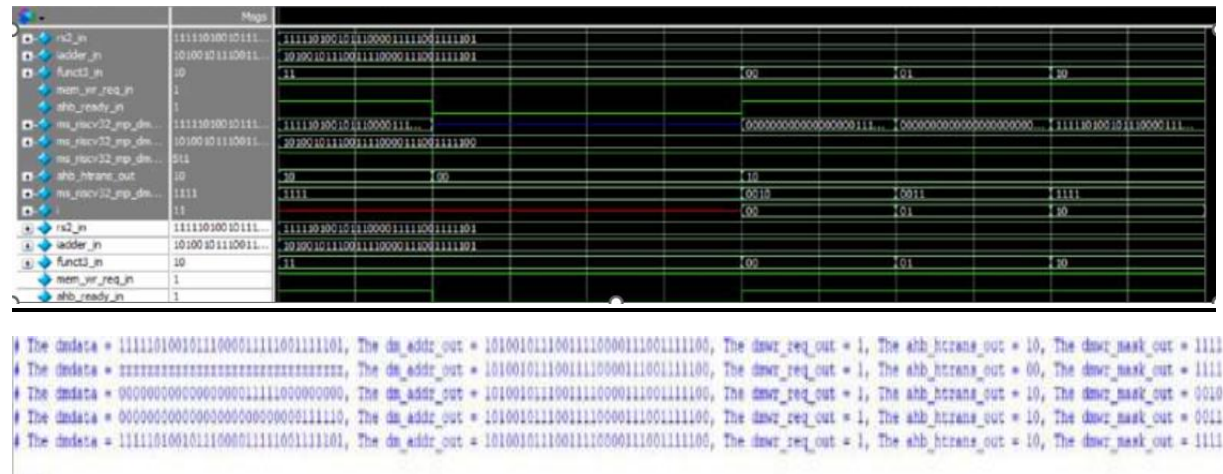
If reset is high all the outputs present in the block will be 0 else it produces output at the posedge of the clock based on the input values. It also integrates a 2:1 MUX with select-lines as branch_taken_in. If branch_taken_in is high then the value of imm_reg_out[0] = 0 else the value of imm_reg_out[0] = imm_in[0]

Output Waveform:



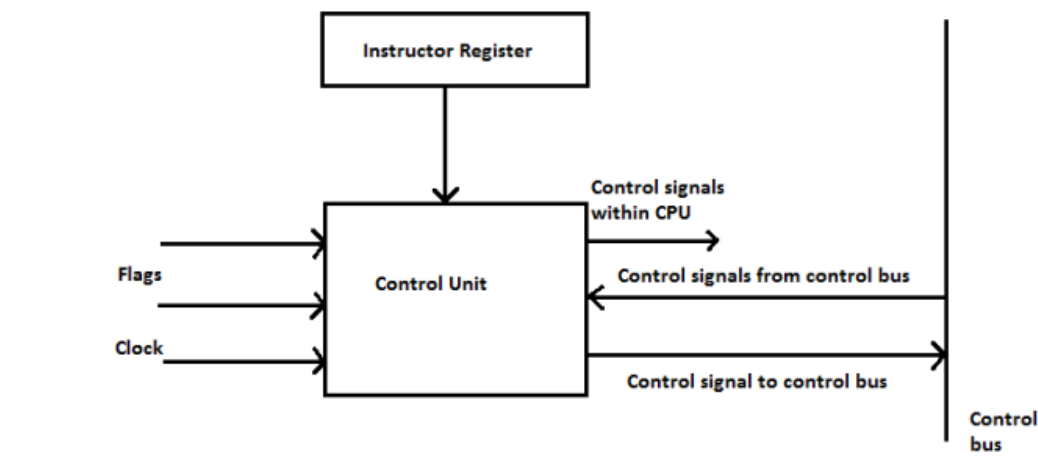
ahb_htrans_out is high and it has the value 2'b10 during a valid store instruction and it is 2'b00 when the transfer is completed and nothing is there to store and ahb_htrans_out is low. Here the value which was present in the register is given to the corresponding address of the memory.

Output Waveform:



Load Unit:

Block Diagram:

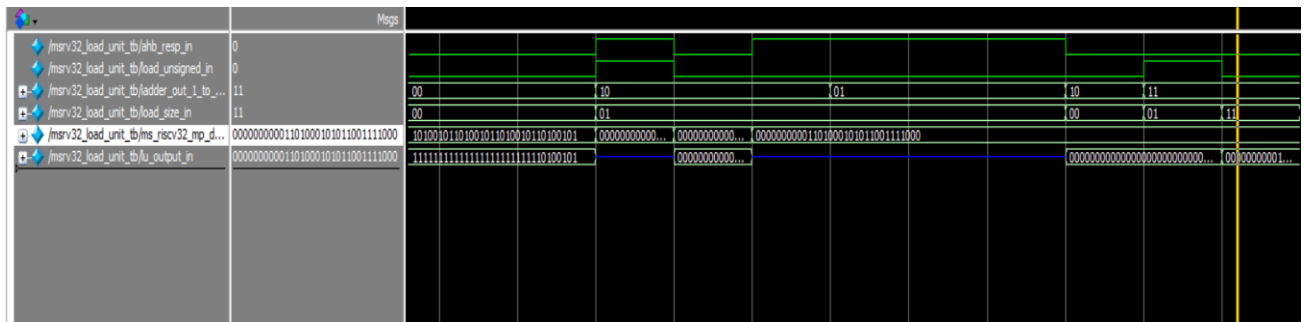


Functionality:

It reads the data_in_input signal and forms a 32 bit value based on the load instruction type which is based on the binary code in funct 3,load_unsigned and iaddr_out_1_to_0_in. if load_size_in is 2'b00 then the byte value will be stored on the least significant byte based on iaddr_out_1_to_0_in. Remaining 24 bits will be filled based on load_unsigned_in. If load_unsigned_in =1 then the upper 24 bits will be filled with 0 else the upper 24 bit value will be filled based on the sign bit . if load_size_in is 2'b01 then the half word value will be

stored on the least significant 16 bit based on iadder_out_1_to_0_in[1]. If load_unsigned_in = 1 then the upper 16 bits will be filled with 0 else the upper 16 bit value will be filled based on the sign bit. If load_size_in is 2'b10, 2'b11 then the full word will be loaded into the memory. If ahb_resp_in is 0 then the value will be loaded to the register or else we will get high impedance as the output.

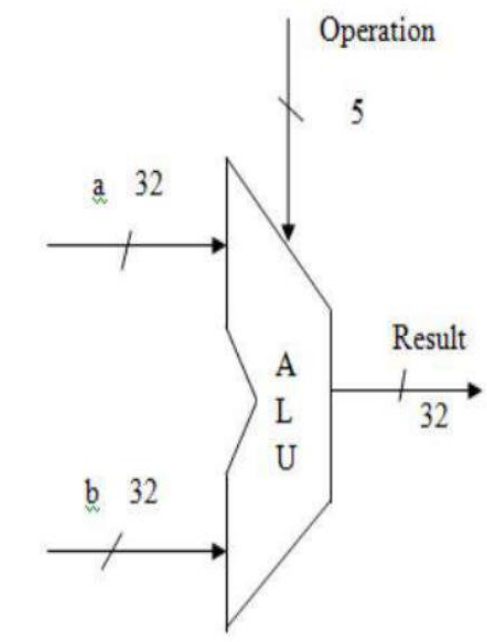
Output Waveform:



```
VSIM 3> run -all
# Initial values:
# ahb_resp_in = 0
# load_unsigned_in = 0
# iadder_out_1_to_0_in = 00
# load_size_in = 00
# ms_riscv32_mp_dmdata_in = a5a5a5a5
# lu_output_in = ffffffffa5
# ** Note: $finish : C:/intelFPGA/17.1/Jaimaven/msrv32_load_unit_tb.v(110)
# Time: 60 ps Iteration: 0 Instance: /msrv32_load_unit_tb
```

ALU Unit:

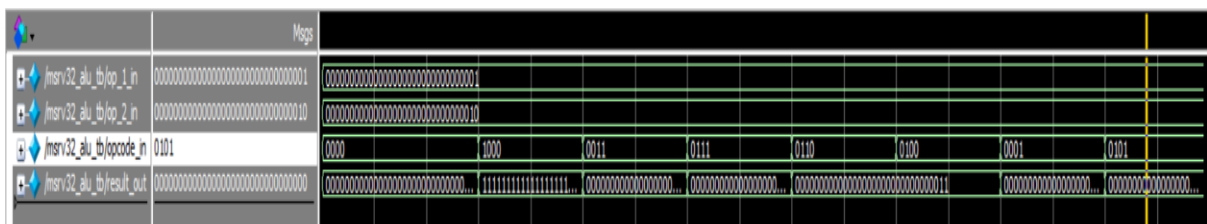
Block Diagram:



Functionality:

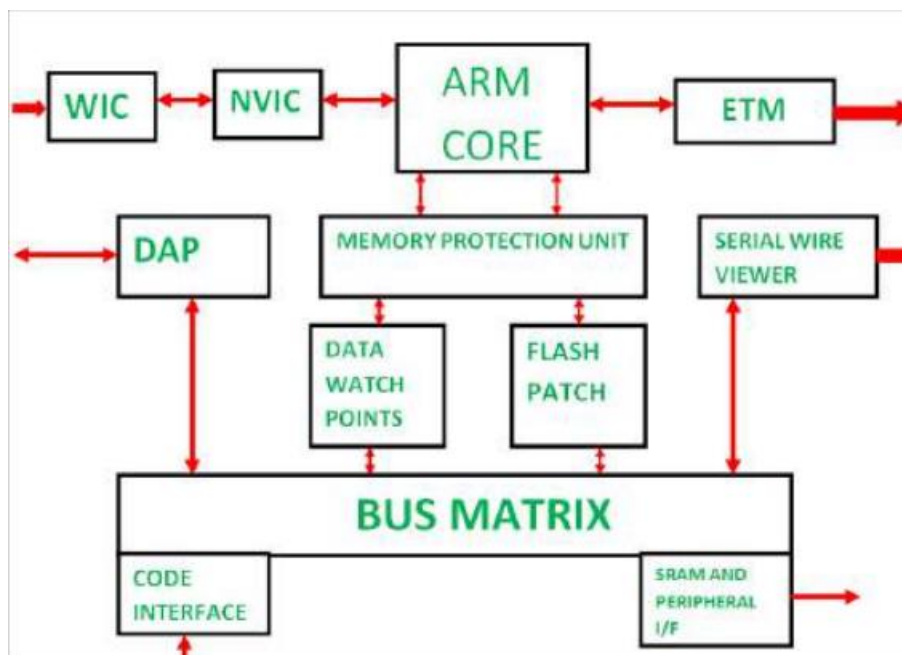
The ALU applies ten logical and arithmetic operations in parallel to two 32-bit operands, outputting the result selected by opcode_in. The opcode values were assigned to facilitate instruction decoding. The most significant bit of opcode in matches with the second most significant bit in the instruction funct7 field. The remaining three bits match with the instruction funct3 field. Based on the opcode different types of operations will be selected and performed between 2 operands.

Output Waveform:



Write Back Mux unit:

Block Diagram:



Functionality:

If alu_src_reg_in' is high then 'alu_2nd_src_mux out is equal to rs2_reg_in. alu_2nd_src_mux out is equal to imm_reg_in. Based on the wb_mux_sel_reg_in output signal wb_mux our will be assigned. If wh_mux_sel_reg in is equal to 000 which is WB_ALU then wb_mux out is equal to alu_result_in.

If wb_mux_sel_reg_in is equal to 001 which is WB LU then wh_mux_out is equal to lu_output_in. If wh_mux_sel_reg_in is equal to 010 which is WB_IMM then wb_mux_out is equal to imm_reg_in.

If 'wb_mux_sel reg_in' is equal to 100 which is WB_CSR then wb_mux_out is equal to csr_data_in.

Output Waveform:

```
# wb_mux_out = 12345678
# alu_2nd_src_mux_out = 00000007
# wb_mux_out = 87654321
# alu_2nd_src_mux_out = 00000002
# ** Note: $finish      : C:/intelFPGA/17.1/msrv32_wb_mux_sel_unit_tb.v(85)
# Time: 30 ps Iteration: 0 Instance: /msrv32_wb_mux_sel_unit_tb
```