



NAME: SAPPARAPU DEVI SRI PRASAD

REGD.NO:21BEC7245

VIT-AP UNIVERSITY

MAVEN SILICON RISC-V DI INTERNSHIP

AUG-2023

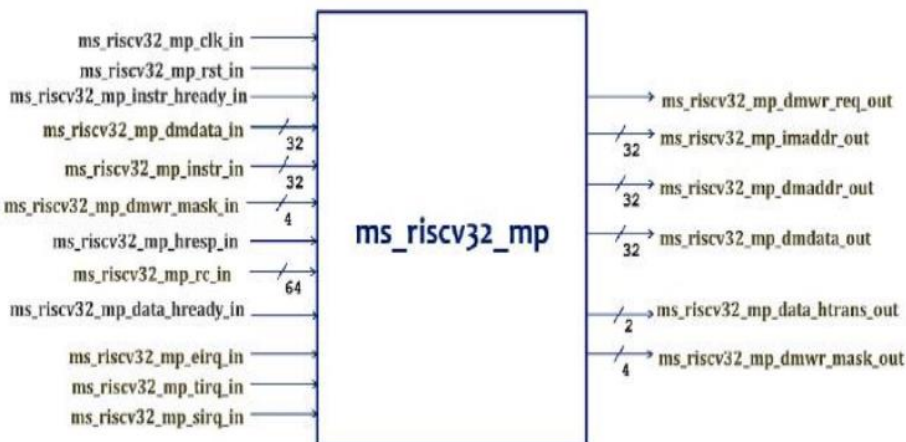
TABLE OF CONTENTS

S.NO	SUB-BLOCK NAMES
1	PC_MUX
2	REG_BLOCK_1
3	IMMEDIATE_GENERATOR
4	IMMEDIATE_ADDER
5	INTEGER_FILE
6	WRITE_ENABLE_GENERATOR
7	INSTRUCTION_MUX
8	BRANCH_UNIT
9	DECODER
10	REGISTER_BLOCK_2
11	STORE_UNIT
12	LOAD_UNIT
13	ALU
14	WB_MUX

ABOUT RISC-V RV32I RISC-V 32I

one of the core variants of the open-source RISC-V instruction set architecture, is designed as a 32-bit architecture with a strong emphasis on simplicity and modularity. Renowned for its open nature, it provides a minimal yet effective set of instructions for tasks such as arithmetic, logical operations, and data manipulation. Featuring 32 general purpose registers and a pipeline-friendly structure, RISC-V 32I serves as a versatile foundation for custom processor designs. Its open accessibility and potential for expansion beyond the basic instruction set make it an excellent choice for a wide range of applications, from embedded systems to Internet of Things (IoT) devices. It enables the creation of customized, efficient computing solutions. RISC-V, originating at UC Berkeley in 2010, was conceived to address the demand for an open, adaptable computing architecture. Its open-source nature and adherence to RISC principles have given rise to RISC-V 32I, a fundamental 32-bit variant that focuses on crucial integer-based instructions.

Top Module Block diagram

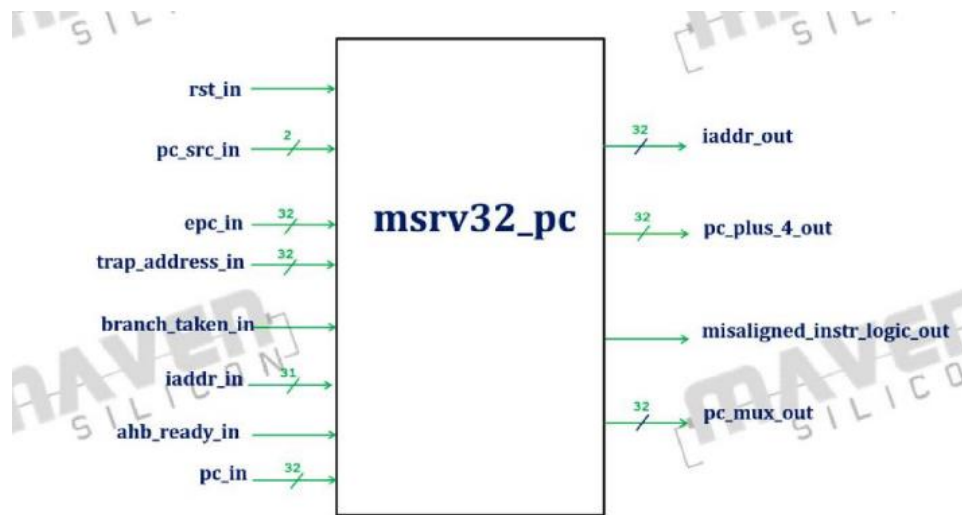


UVM TEST:

[illegible]

1.PC_MUX:

BLOCK DIAGRAM:



FUNCTIONALITY:

In a processor, the PC (Program Counter) is a register that holds the address of the next instruction to be executed. The PC_MUX is responsible for choosing the next value for the PC based on various conditions, such as branch instructions, jump instructions, or normal sequential execution.

Input Selection: The PC_MUX will have multiple inputs, each corresponding to a potential next address. For example, one input might be for the next sequential address, one for a branch target address, and others for different types of jumps or exceptions.

Control Signals: There will be control signals that determine which input is selected. These signals are typically derived from the instruction being executed and other control logic in the processor.

Output: The selected input becomes the output of the PC_MUX and is used as the next value for the PC.

Clock and Synchronization: The PC_MUX will operate in sync with the clock signal of the processor to ensure proper timing.

SIMULATION OF PC_MUX:

PROJECT MANAGER - PC_MUX

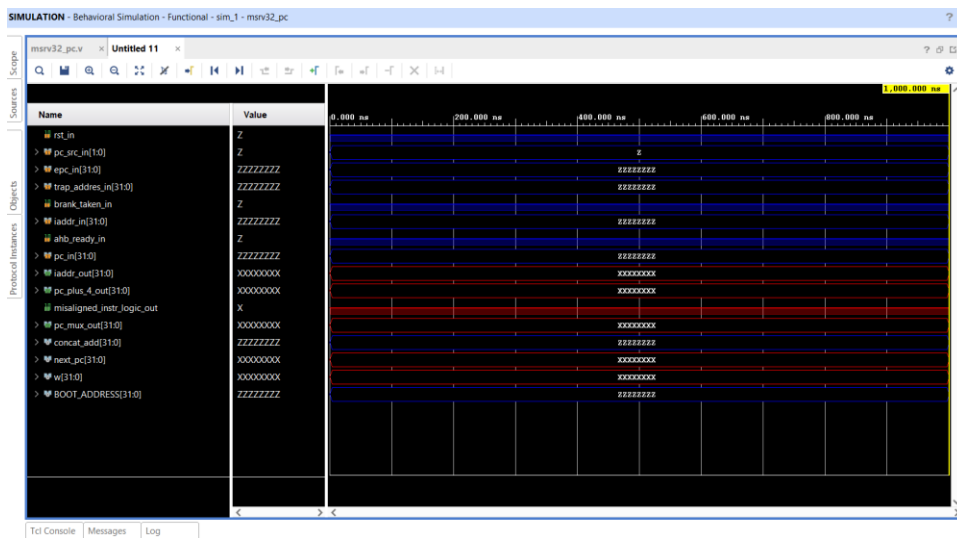
Project Summary x msv32_pc.v x

D:\projects\PC_MUX\PC_MUX\srcs\new\msv32_pc.v

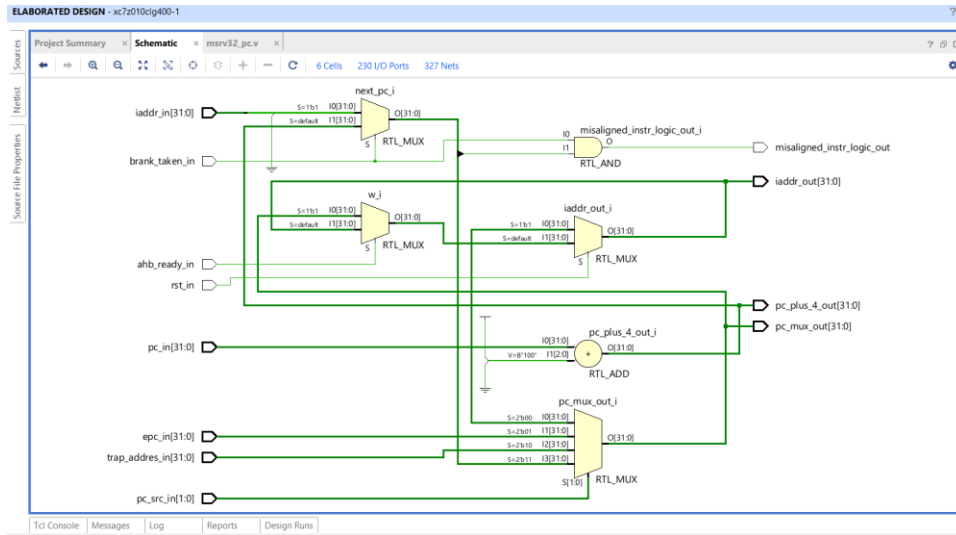
```
23 module msv32_pc(rst_in,pc_src_in,epc_in,trap_address_in,brank_taken_in,iaddr_in,ahb_ready_in,pc_in,iaddr_out,
24 pc_plus_4_out,misaligned_instr_logic_out,pc_mux_out);
25 input rst_in;
26 input [1:0] pc_src_in;
27 input [31:0] epc_in;
28 input [31:0] trap_address_in;
29 input brank_taken_in;
30 input [31:0] iaddr_in;
31 input ahb_ready_in;
32 input [31:0] pc_in;
33 output [31:0] iaddr_out;
34 output [31:0] pc_plus_4_out;
35 output misaligned_instr_logic_out;
36 output reg [31:0] pc_mux_out;
37 wire [31:0] concat_addr_wire [31:0];
38 wire [31:0] BOOT_ADDRESS;
39 assign concat_addr=(iaddr_in,1'b1);
40 assign pc_plus_4_out=pc_in+32'h4;
41 assign next_pc=brank_taken_in?concat_addr:pc_plus_4_out;
42 assign misaligned_instr_logic_out=(brank_taken_in) && (next_pc[1]);
43 assign w=ahb_ready_in?pc_mux_out:iaddr_out;
44 assign iaddr_out=rst_in?BOOT_ADDRESS:w;
45 always @(pc_src_in) begin
46 case (pc_src_in)
47 2'b00:pc_mux_out=BOOT_ADDRESS;
48 2'b01:pc_mux_out=pc_in;
49 2'b10:pc_mux_out=trap_address_in;
50 2'b11:pc_mux_out=next_pc;
51 default:;
52 endcase
53 end
```

Tcl Console | Messages | Log | Reports | Design Runs

WAVE FORM:



NETLIST:

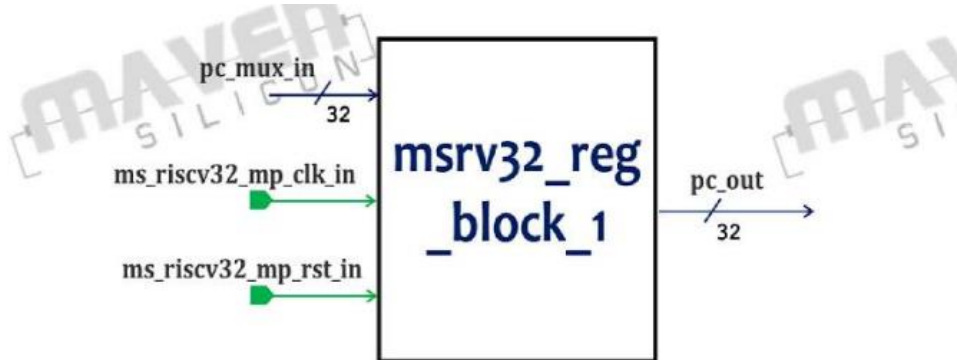


Description:

In RISC-V's 32-bit architecture, the PC MUX (Program Counter Multiplexer) is a vital component responsible for choosing the next instruction address. It takes inputs from various sources, like branch instructions and the next sequential instruction, to decide which address becomes the program counter for the next cycle. This process is essential for managing control flow and ensuring instructions are executed in the correct sequence, thus determining the program's execution path within the CPU.

2.REG_BLOCK_1:

BLOCK DIAGRAM:



FUNCTIONALITY:

It registers the **pc_mux_in** and produces the output at the posedge clock if there is no reset.

Output Production: After registering **pc_mux_in**, Register Block 1 produces an output. This output could be made available to other parts of the digital system for further processing or use.

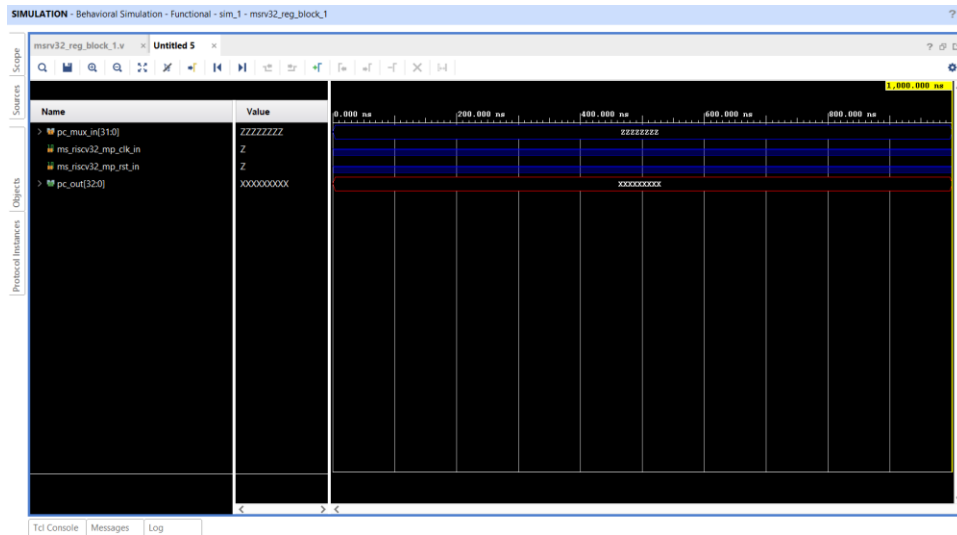
Clock Edge Sensitivity: The register within Register Block 1 responds to the rising edge (posedge) of the clock signal. This is a common behavior for synchronous digital designs, where operations are synchronized to the clock signal.

SIMULATION OF REG_BLOCK_1:

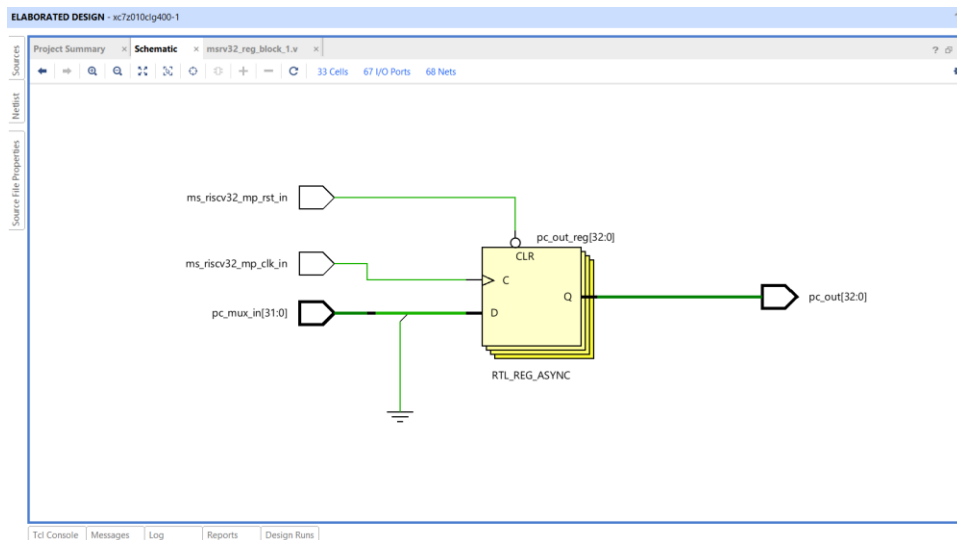
```
PROJECT MANAGER - reg_block1
Project Summary  x  msrv32_reg_block_1.v
D:/projects/reg_block1/reg_block1/srcs/sources_1/new/msrv32_reg_block_1.v

2 // Company:
3 // Engineer:
4 //
5 // Create Date: 28.10.2023 17:06:54
6 // Design Name:
7 // Module Name: msrv32_reg_block_1
8 // Project Name:
9 // Target Device:
10 // Tool Versions:
11 // Description:
12 //
13 // Dependencies:
14 //
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 //
21
22 module msrv32_reg_block_1 (pc_mux_in, ms_riscv32_mp_clk_in, ms_riscv32_mp_rst_in, pc_out);
23 input [31:0] pc_mux_in;
24 input ms_riscv32_mp_clk_in;
25 input ms_riscv32_mp_rst_in;
26 output reg [31:0] pc_out;
27 always @(posedge ms_riscv32_mp_clk_in or negedge ms_riscv32_mp_rst_in) begin
28   if (~ms_riscv32_mp_rst_in) pc_out<=32'h0;
29   else pc_out<=pc_mux_in;
30 end
31 endmodule
32
```


WAVE FORM:



NETLIST:

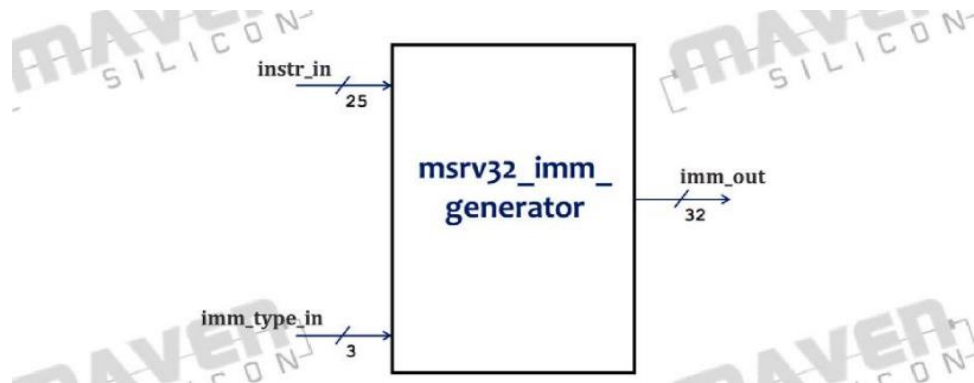


Description:

In the RISC-V 32-bit architecture, "reg block 1" refers to a set of general-purpose registers designated as x1 through x31. These registers serve as temporary data storage for program operations. Notably, x0 is a special register always holding the value zero. In practice, x1 typically stores return addresses in function calls, while the remaining registers, x2 through x31, are available for various programmer-defined uses. Often, x31 is utilized as the stack pointer (sp) to manage function call frames and local variables effectively.

3.IMMEDIATE GENERATOR:

BLOCKDIAGRAM:



FUNCTIONALITY:

An "immediate generator" in the context of computer architecture typically refers to a component that generates immediate values or constants used in instructions. Immediate values are constants or literals that are part of an instruction and are used as operands in operations.

Generating Constants: The primary function of an immediate generator is to produce constant values that can be used in instructions. These constants can be integers, floating-point numbers, or other data types depending on the architecture.

Providing Operand Values: The immediate generator supplies these constant values directly to the instruction execution units within a processor. These values are used as operands for arithmetic, logic, or other operations.

Encoding and Formatting: The immediate generator encodes the constant value in the format required by the instruction set architecture. This may involve converting the constant to a specific binary representation.

Supporting Operations: Immediate values generated by the immediate generator can be used in a wide range of operations, including arithmetic operations (addition, subtraction, etc.), logical operations (AND, OR, XOR), comparisons, and more.

Timing and Synchronization: The immediate generator operates in sync with the processor's clock and timing requirements. It ensures that the generated constant is available at the correct time for instruction execution.

SIMULATION OF IMMEDIATE GENERATOR:

PROJECT MANAGER - immediate_generator

Project Summary x **msrv32_imm_generator.v** x

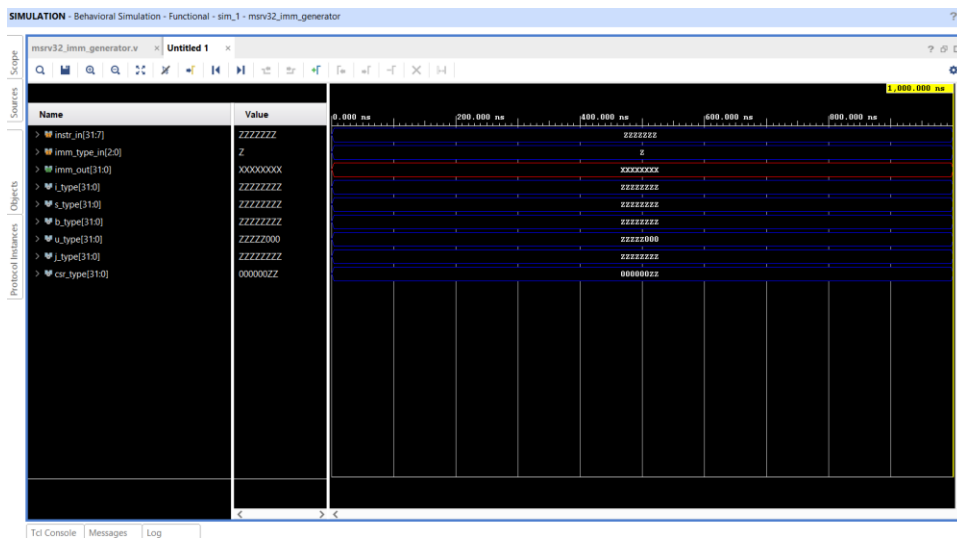
D:/projects/immediate_generator/immediate_generator/srcs/sources_1/new/msrv32_imm_generator.v

```
15 //  
16 // Revision:  
17 // Revision 0.01 - File Created  
18 // Additional Comments:  
19 //  
20 //////////////////////////////////////  
21  
22 module msrv32_imm_generator (instr_in, imm_type_in, imm_out);  
23   input [31:7] instr_in;  
24   input [2:0] imm_type_in;  
25   output reg [31:0] imm_out;  
26   wire [31:0] i_type, s_type, b_type, u_type, j_type, ccr_type;  
27   assign i_type = ((20(instr_in[31]))), instr_in[31:20];  
28   assign s_type = ((20(instr_in[31])), instr_in[31:25], instr_in[11:7]);  
29   assign b_type = ((20(instr_in[31])), instr_in[7], instr_in[30:25], instr_in[11:8], 1'b0);  
30   assign u_type = (instr_in[31:12], 12'h0);  
31   assign j_type = ((12(instr_in[31])), instr_in[19:12], instr_in[20], instr_in[30:21], 1'b0);  
32   assign ccr_type = (27'b0, instr_in[19:15]);  
33   always @ (imm_type_in) begin  
34     case (imm_type_in)  
35       3'dimm_out=i_type;  
36       3'dimm_out=s_type;  
37       3'dimm_out=b_type;  
38       3'dimm_out=u_type;  
39       3'dimm_out=j_type;  
40       3'dimm_out=ccr_type;  
41       3'dimm_out=i_type;  
42     endcase  
43   end  
44 end  
45 endmodule
```

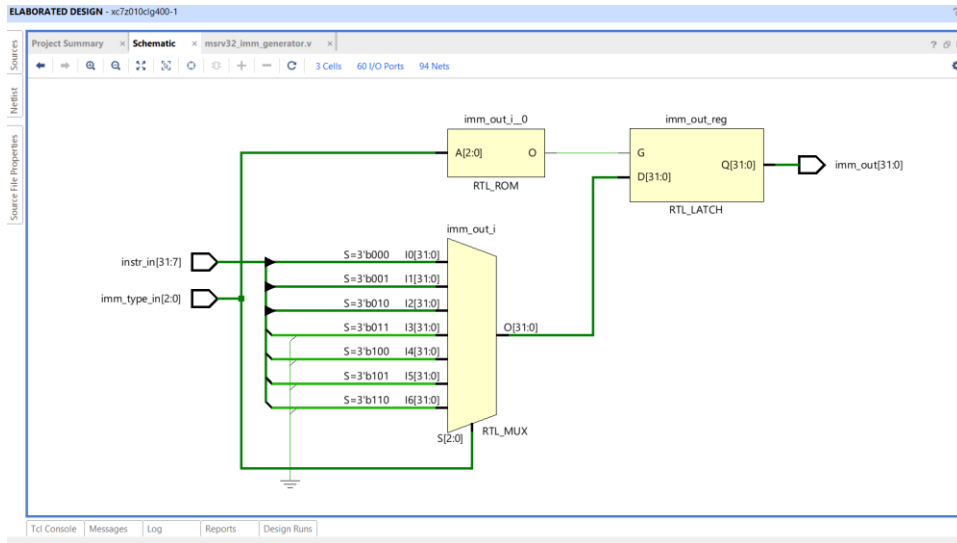
Tcl Console Messages Log Reports Design Runs

22:1 Insert Verilog

WAVE FORM :

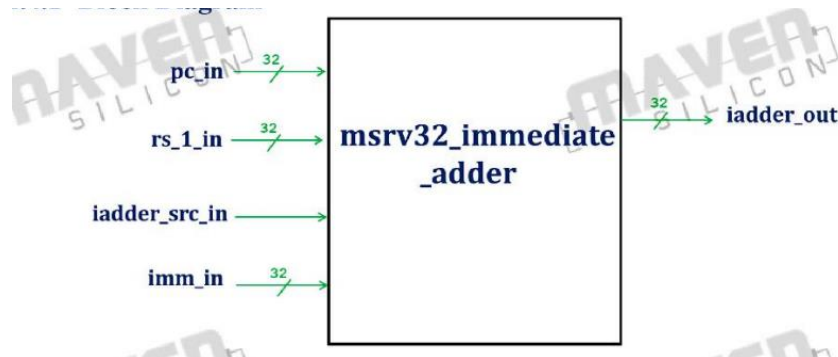


NETLIST:



4.IMMEDIATE ADDER:

BLOCK DIAGRAM:



FUNCTIONALITY:

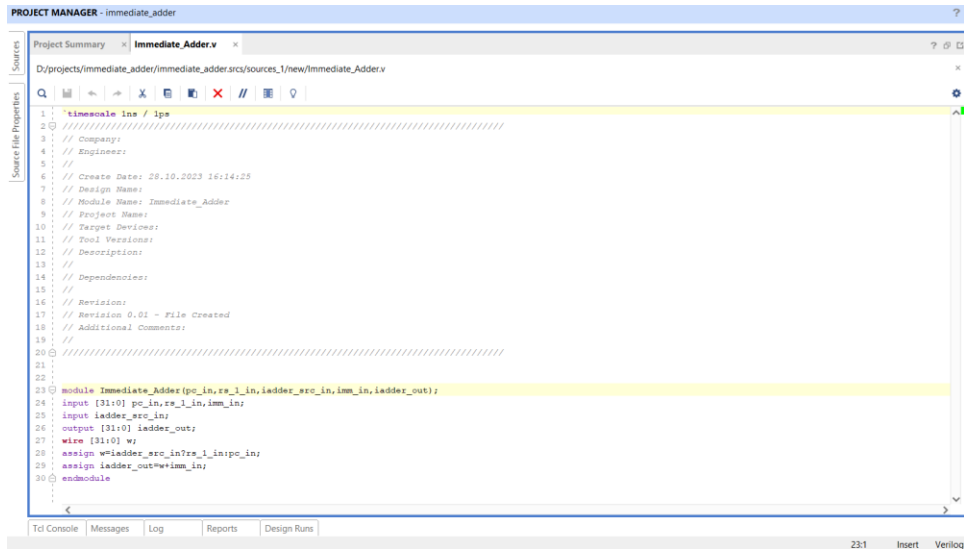
An immediate adder is a specific component within a processor's execution unit designed to perform addition operations involving an immediate value. Here's a breakdown of its functionality:

Addition Operation: The primary function of an immediate adder is to perform addition operations. It takes an immediate value and adds it to another operand (usually a register value) provided as input.

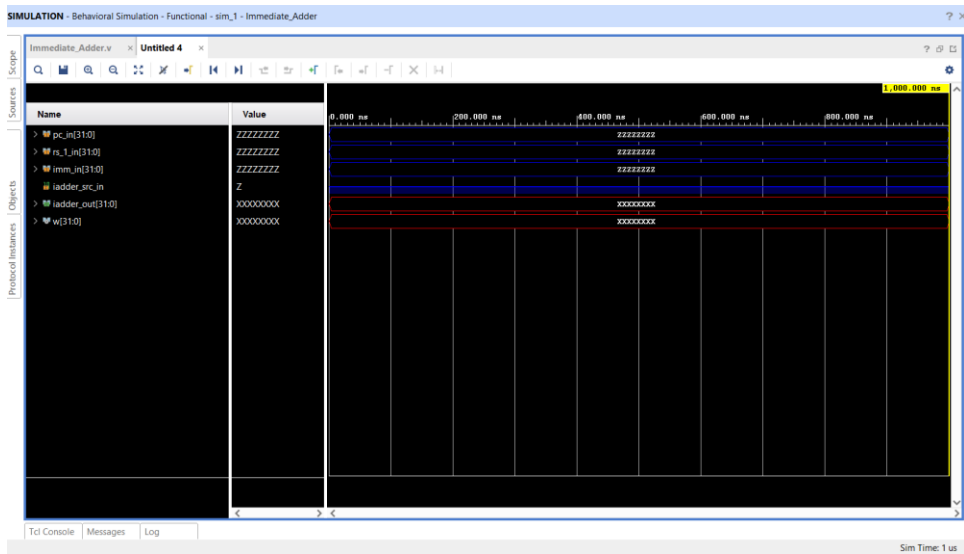
Arithmetic Logic Unit (ALU) Operation: The immediate adder is a part of the Arithmetic Logic Unit (ALU) in a processor. It performs the addition operation, and in some cases, it may also handle other arithmetic operations.

Overflow Detection: It may include circuitry to detect overflow conditions that occur when the addition operation produces a result that is too large to be represented with the given number of bits.

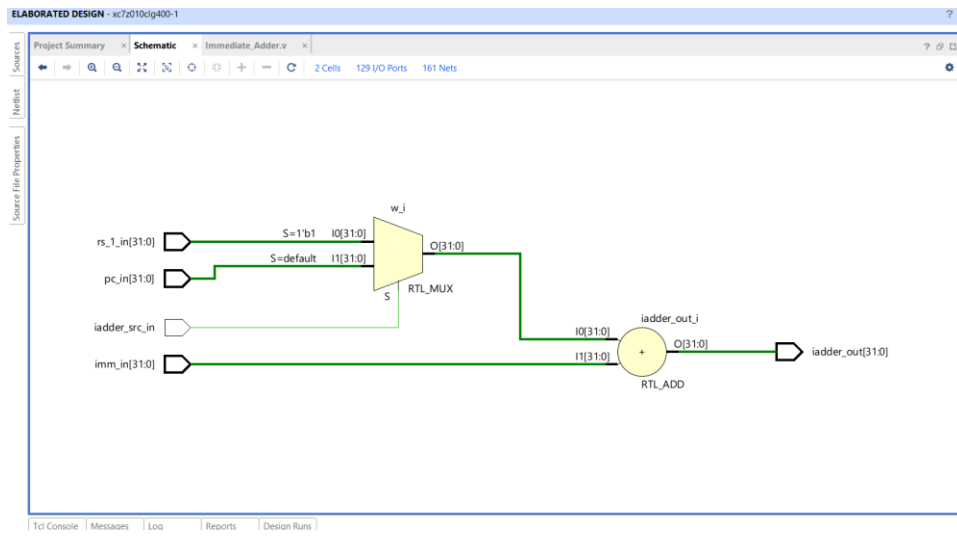
SIMULATION OF IMMEDIATE ADDER:



WAVE FORM:

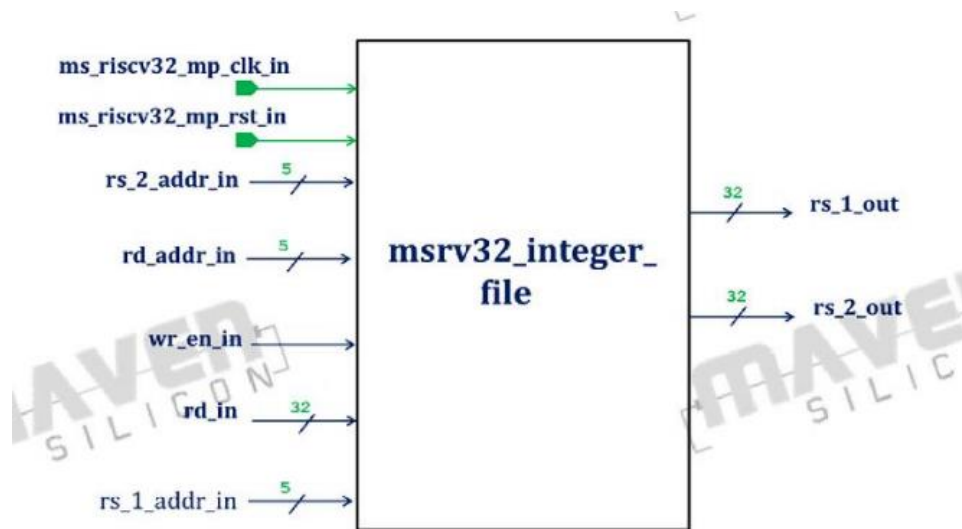


NETLIST:



5.INTEGER FILE:

BLOCK DIAGRAM:



FUNCTIONALITY:

An "integer file" in the context of computer architecture typically refers to a collection of registers or storage elements specifically designed to hold integer data. Here's a breakdown of its functionality:

Data Storage: The primary function of an integer file is to store integer data. Each element within the file (which may be a register or memory location) is capable of holding an integer value.

Read and Write Operations: The integer file supports both read and write operations. Data can be written into an element to update its content, and it can be read from an element to retrieve the stored information.

Arithmetic and Logic Operations: Elements in an integer file can be used as operands for arithmetic (addition, subtraction, multiplication, division, etc.) and logic (AND, OR, XOR, etc.) operations.

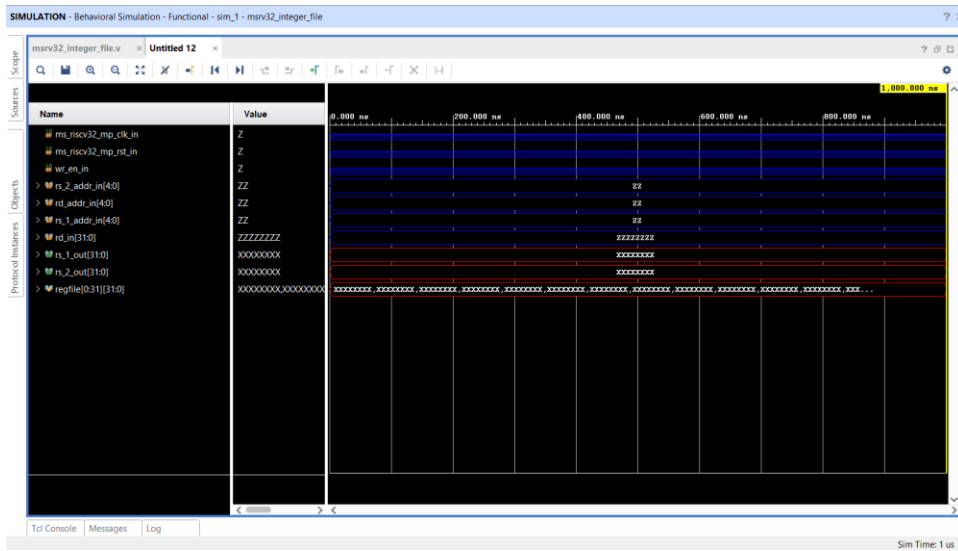
Data Movement: Data can be moved between elements within the integer file or between the file and other components of the processor, such as the ALU (Arithmetic Logic Unit) or memory.

Indexing and Addressing: The elements in the integer file are typically addressed by an index or address. This allows specific elements to be accessed or modified based on their position within the file.

SIMULATION OF INTEGER_FILE:

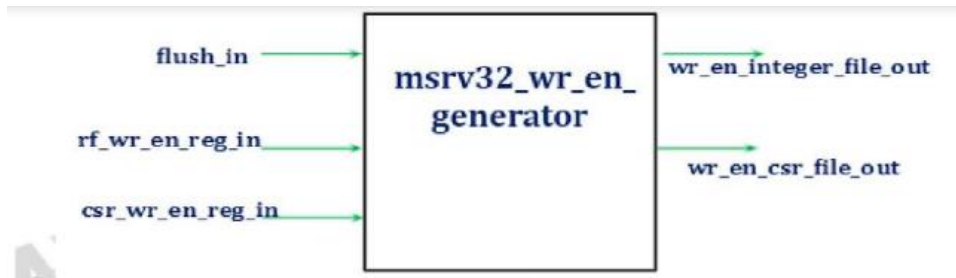
[illegible]

WAVE FORM:



6.WRITE ENABLE GENERATOR:

BLOCK DIAGRAM:



FUNCTIONALITY:

A "Write Enable Generator" in digital design refers to a component responsible for generating control signals that determine when a write operation is allowed to occur in a register or memory element. Here's a breakdown of its functionality:

Control Signal Generation:The primary function of a Write Enable Generator is to produce control signals that indicate whether a write operation should be allowed to occur. These signals are often referred to as "write enables" or "write signals".

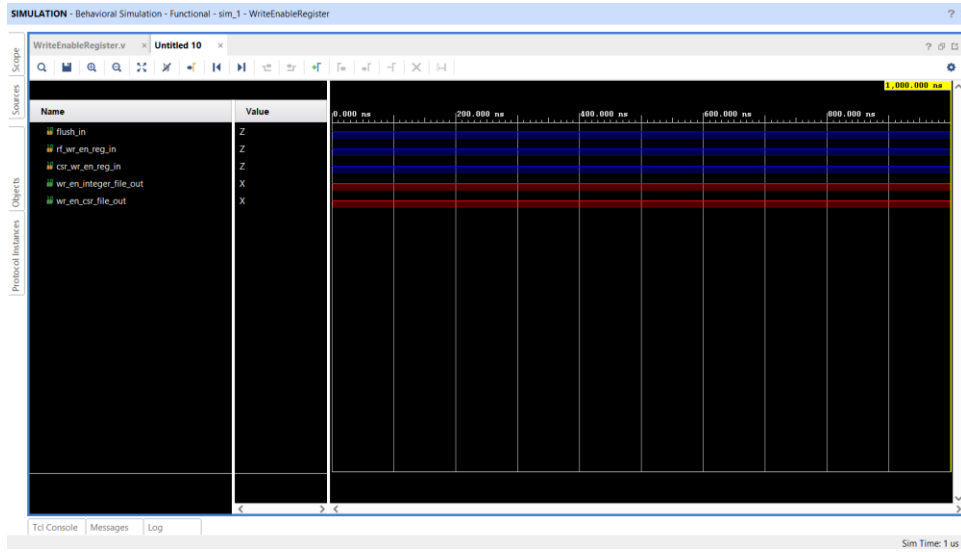
Conditional Write Operation:The generator evaluates certain conditions or criteria to determine whether a write operation should be permitted. These conditions may be based on inputs from the system, status flags, or other logic signals.

Write Strobe Signal Generation:In some cases, the Write Enable Generator may produce a write strobe signal. This signal serves as an additional indicator to trigger the actual writing of data.

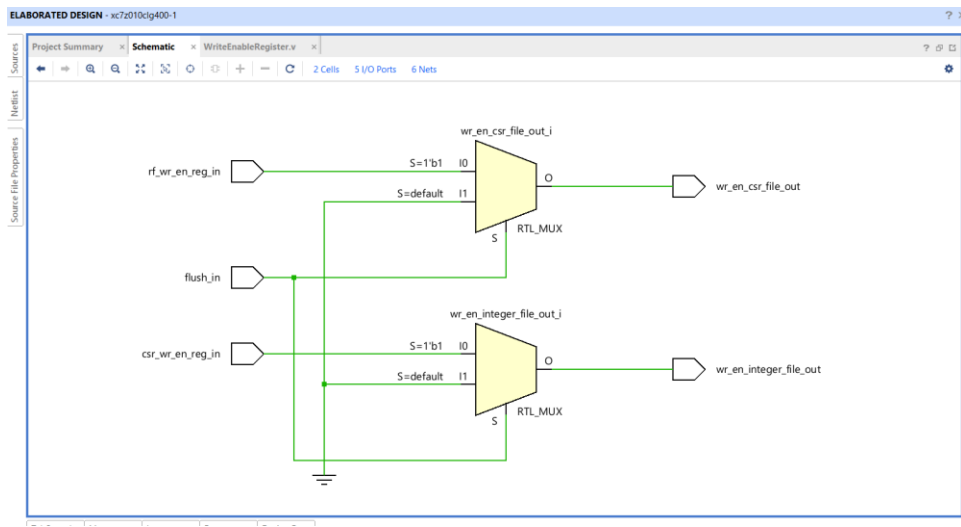
SIMULATION OF WRITE ENABLE GENERATOR:

```
1 timescale 1ns / 1ps
2 // Company:
3 // Engineer:
4 // Create Date: 28.10.2023 17:13:54
5 // Design Name:
6 // Module Name: WriteEnableRegister
7 // Project Name:
8 // Target Device:
9 // Tool Versions:
10 // Description:
11 //
12 // Dependencies:
13 //
14 // Revision:
15 // Revision 0.01 - File Created
16 // Additional Comments:
17 //
18 //
19 //
20 //
21 module WriteEnableRegister(flush_in, rf_wr_en_reg_in, csr_wr_en_reg_in, wr_en_integer_file_out, wr_en_csr_file_out);
22 input flush_in, rf_wr_en_reg_in, csr_wr_en_reg_in;
23 output wr_en_integer_file_out, wr_en_csr_file_out;
24 assign wr_en_integer_file_out = flush_in & rf_wr_en_reg_in;
25 assign wr_en_csr_file_out = flush_in & csr_wr_en_reg_in;
26 endmodule
```

WAVE FORM:

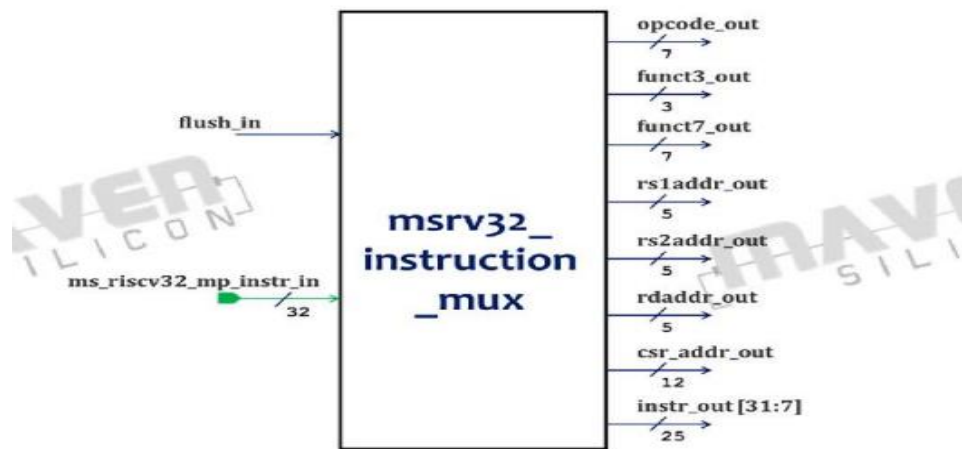


NETLIST:



7.INSTRUCTION MUX:

BLOCK DIAGRAM:



FUNCTIONALITY:

An instruction multiplexer, often referred to as an "instruction mux" or "opcode mux," is a critical component within a processor's instruction fetch stage. Its primary functionality is to select the appropriate instruction from a set of possible sources and pass it along for further processing. Here's a breakdown of its functionality:

Multiple Instruction Sources: The instruction mux takes inputs from various sources. These sources may include the instruction memory, branch prediction unit, or other components involved in instruction fetching.

Selection Logic: The mux has logic to select one instruction from among the available sources based on control signals or conditions provided by the processor's control unit.

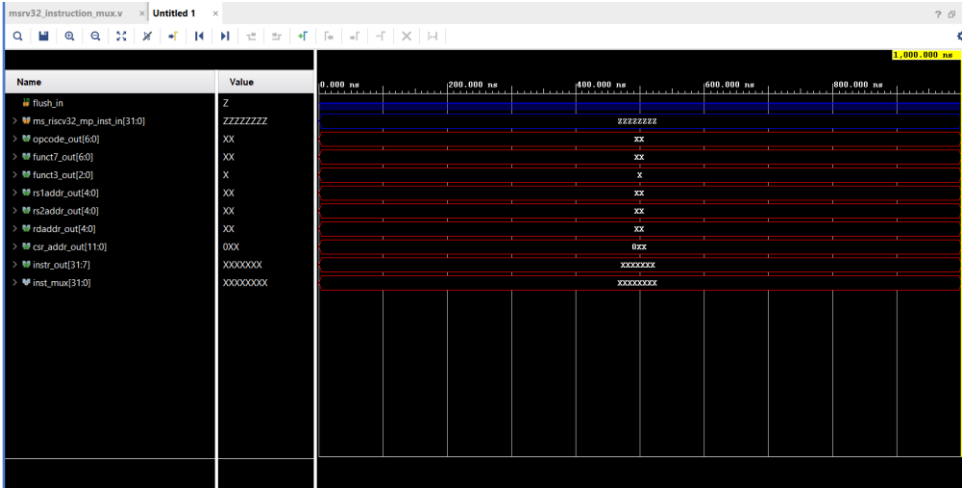
Conditional Branch Handling: In many cases, the instruction mux is responsible for handling conditional branches. It selects the next instruction based on the outcome of a branch prediction or condition evaluation.

Pipeline Flushing (if applicable): In cases where a pipeline needs to be flushed (e.g., due to a mispredicted branch), the instruction mux plays a role in ensuring that the correct instruction is selected for the next fetch cycle.

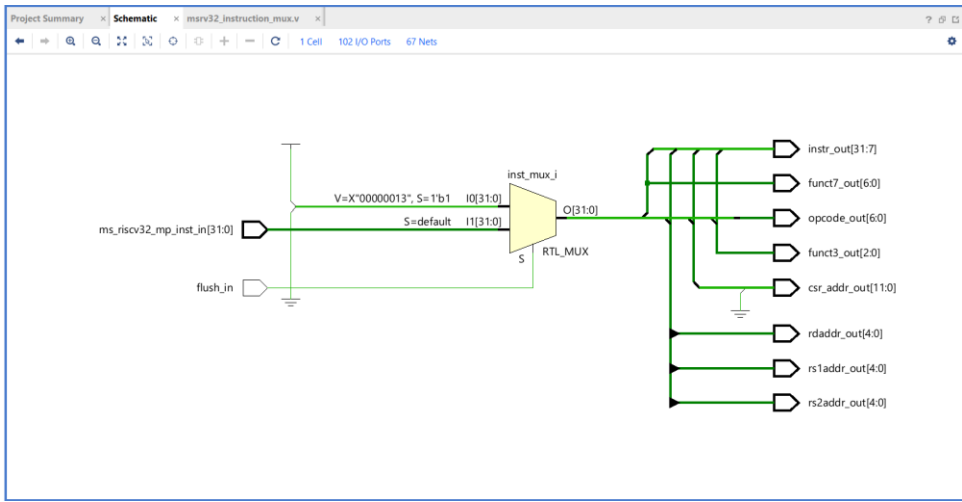
Integration in the Overall Control Logic: The instruction mux is an integral part of the control logic within a processor. It works in conjunction with other components to ensure the correct flow of instructions through the pipeline.

SIMULATION OF INSTRUCTION_MUX:

WAVE FORM:

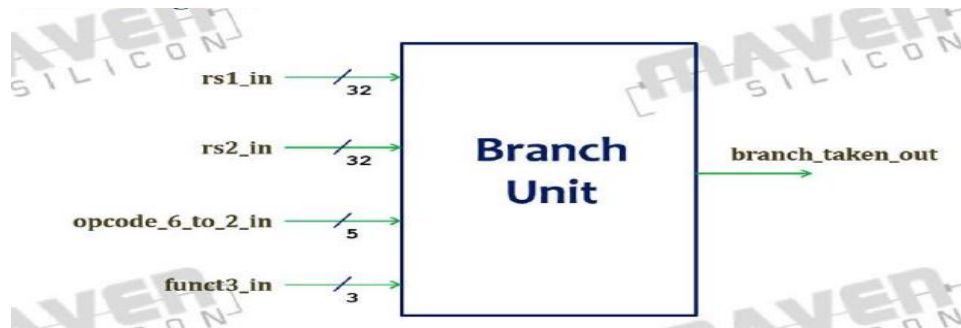


NETLIST:



8. BRANCH UNIT:

BLOCK DIAGRAM:



FUNCTIONALITY:

The branch unit is a component responsible for handling branch instructions. Branch instructions are used to alter the program flow by conditionally changing the address of the next instruction to be executed. Here's a breakdown of the functionality of the branch unit in RISC-V:

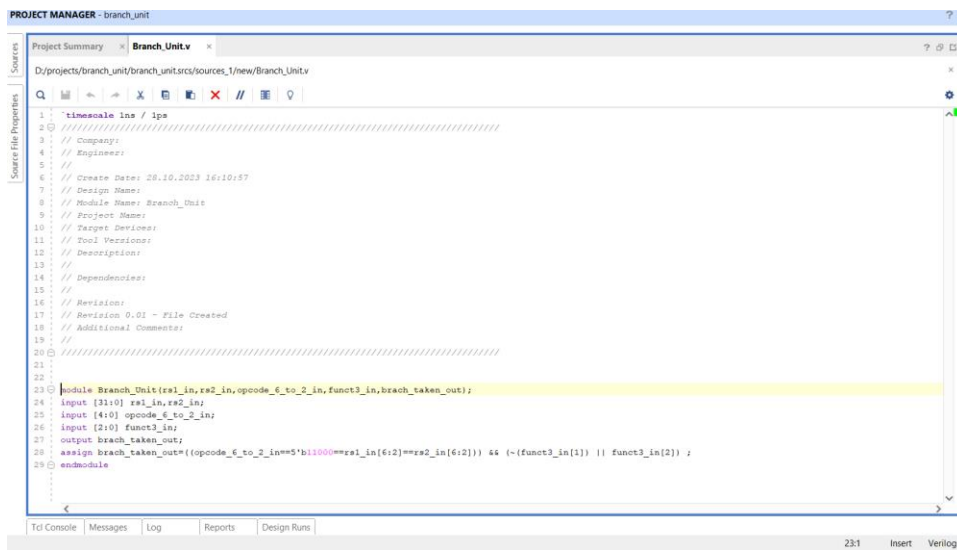
Branch Instruction Decoding: The branch unit decodes branch instructions from the instruction stream. It identifies whether the current instruction is a branch instruction and extracts relevant fields like the branch condition, target address, and offset.

Condition Evaluation: The branch unit evaluates the condition specified in the branch instruction. This condition determines whether the branch should be taken or not. Common conditions include equality, inequality, greater-than, less-than, etc.

Pipeline Control: The branch unit interacts with the processor's pipeline control logic. It determines whether to stall the pipeline, forward data, or take other actions based on the outcome of the branch condition.

Conditional Branch Handling: If the branch condition is true, the branch unit redirects the program flow to the target address. If false, it allows the program to continue with the next sequential instruction.

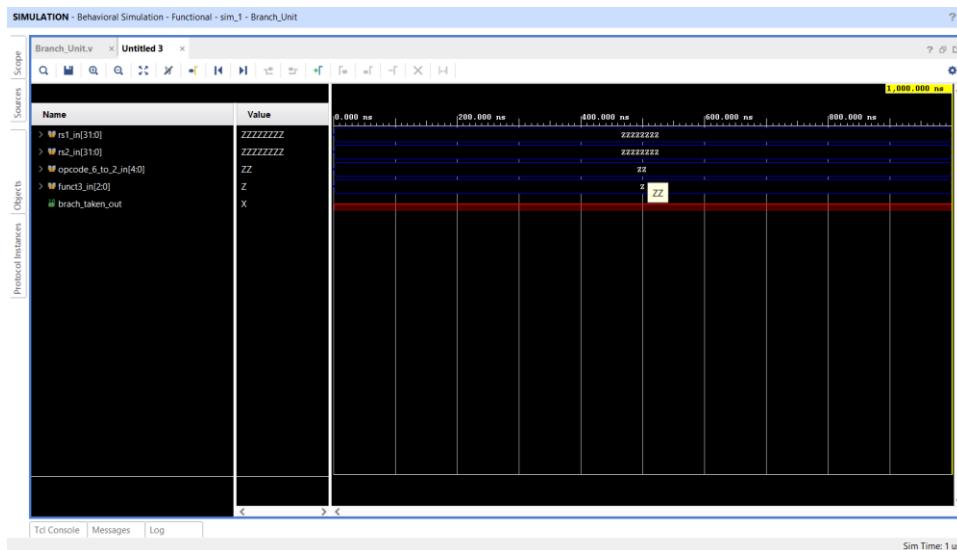
SIMULATION OF BRANCH_UNIT:



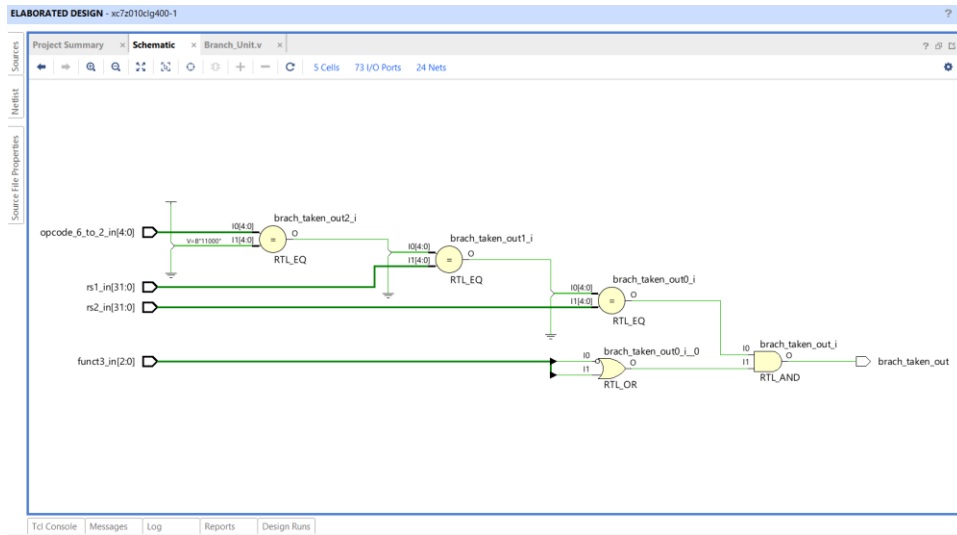
The screenshot shows the Project Manager window for a project named 'branch_unit'. The 'Branch_Unit.v' file is open, displaying Verilog code. The code includes a module definition for 'Branch_Unit' with inputs 'rs1_in', 'rs2_in', 'opcode_6_to_2_in', 'funct3_in', and 'branch_taken_out'. The code implements a branch unit logic, including a conditional branch based on the opcode and the funct3 field. The code is as follows:

```
1: `timescale 1ns / 1ps
2: //////////////////////////////////////
3: // Company:
4: // Engineer:
5: //
6: // Create Date: 28.10.2023 16:10:57
7: // Design Name:
8: // Module Name: Branch_Unit
9: // Project Name:
10: // Target Devices:
11: // Tool Versions:
12: // Descriptions:
13: //
14: // Dependencies:
15: //
16: // Revisions:
17: // Revision 0.01 - File Created
18: // Additional Comments:
19: //
20: //////////////////////////////////////
21:
22:
23: module Branch_Unit(rs1_in,rs2_in,opcode_6_to_2_in,funct3_in,branch_taken_out);
24: input [31:0] rs1_in,rs2_in;
25: input [4:0] opcode_6_to_2_in;
26: input [2:0] funct3_in;
27: output branch_taken_out;
28: assign branch_taken_out=((opcode_6_to_2_in==5'b1000==rs1_in[6:2]==rs2_in[6:2])) && (~funct3_in[1] || funct3_in[2]);
29: endmodule
```

WAVE FORM:

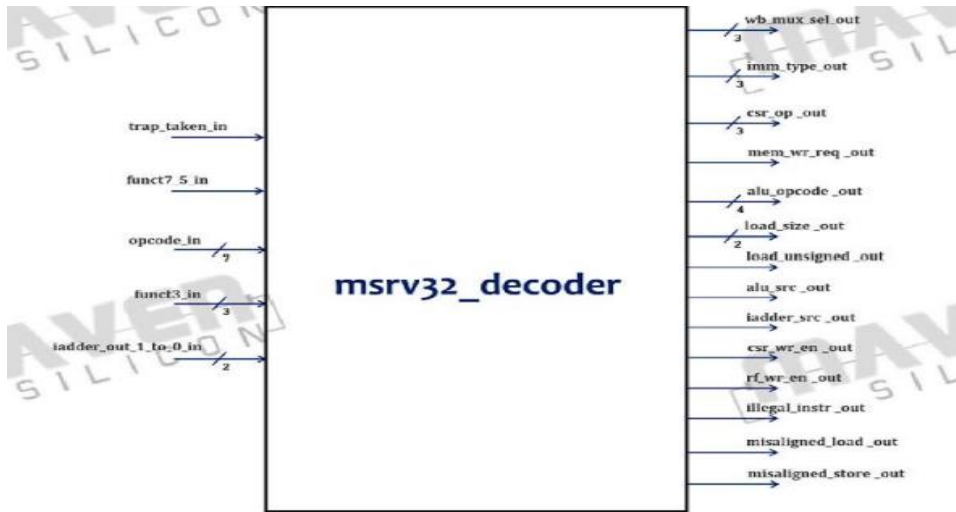


NETLIST:



9.DECODER

BLOCK DIAGRAM:



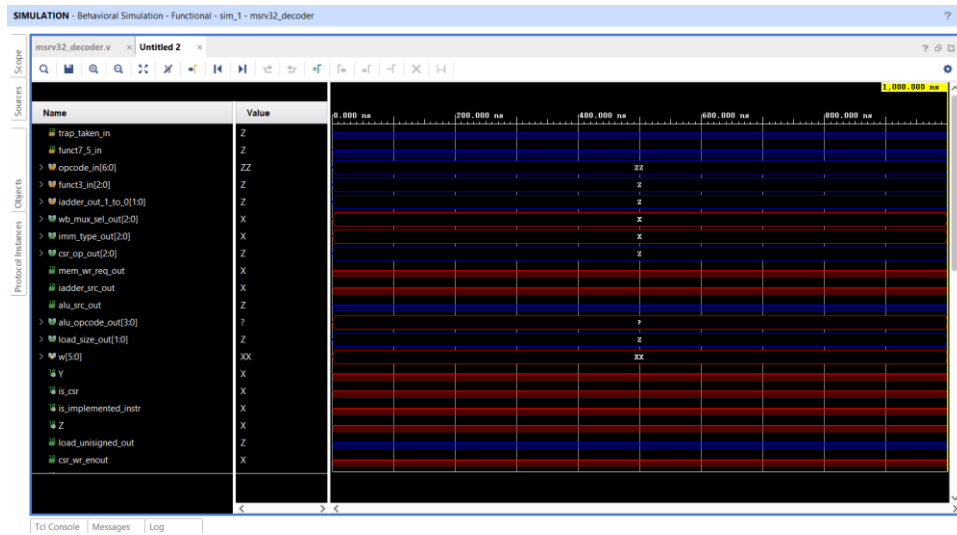
FUNCTIONALITY:

The decoder decodes the instructions and generates the signals that control the memory, the load unit, the store unit, the ALU, 2 register files, the immediate generator and write back multiplexer.

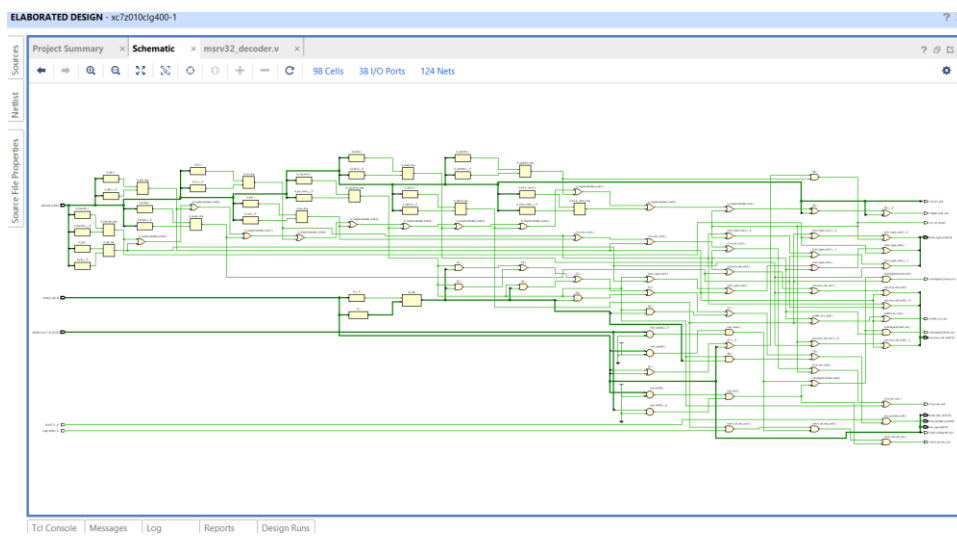
SIMULATION OF DECODER:

```
PROJECT MANAGER - Decoder
Project Summary - msrv32_decoder.v
D:/projects/Decoder/Decoder/srcs/sources_1/new/msrv32_decoder.v
// Revision 0.01 - File Created
// Additional Comments:
//
//
module msrv32_decoder(trap_taken_in,funct7_5_in,opcode_in,funct3_in,iadder_out_1_to_0,wb_mux_sel_out,imm_type_out,csr_op_out,mem_wr_req_out,alu_opcode_out,load_size_out,load_unsigned_out,csr_wr_en_out,rf_wr_en_out,illegal_instr_out,misaligned_load_out,misaligned_store_out);
input trap_taken_in,funct7_5_in;
input [6:0] opcode_in;
input [2:0] funct3_in;
input [1:0] iadder_out_1_to_0;
output [2:0] wb_mux_sel_out,imm_type_out,csr_op_out;
output mem_wr_req_out,iadder_src_out,alu_src_out;
output [3:0] alu_opcode_out;
output [1:0] load_size_out;
reg [5:0] w;
wire Y,is_csr,is_implemented_instr,2;
output load_unsigned_out,csr_wr_en_out,rf_wr_en_out,illegal_instr_out,misaligned_load_out,misaligned_store_out;
reg is_branch,is_jal,is_salr,is_alupc,is_lui,is_op,is_op_imm,is_load,is_store,is_system,is_misc_mem;
and G1(is_add,is_op_imm,w[0]);
and G2(is_alui,is_op_imm,w[1]);
and G3(is_alui,is_op_imm,w[2]);
and G4(is_andi,is_op_imm,w[3]);
and G5(is_ori,is_op_imm,w[4]);
and G6(is_xori,is_op_imm,w[5]);
or G7(Y,funct3_in[0],funct3_in[0],funct3_in[0]);
and G8(is_csr,Y,is_system);
assign csr_wr_en_out=is_csr;
assign rf_wr_en_out=is_lui || is_op || is_salr || is_load || is_csr || is_op_imm;
assign wb_mux_sel_out[0]=is_load||is_alupc||is_jal||is_salr;
system wb_mux_sel_out[1:2]=1'b11;
endmodule
```

WAVE FORM:

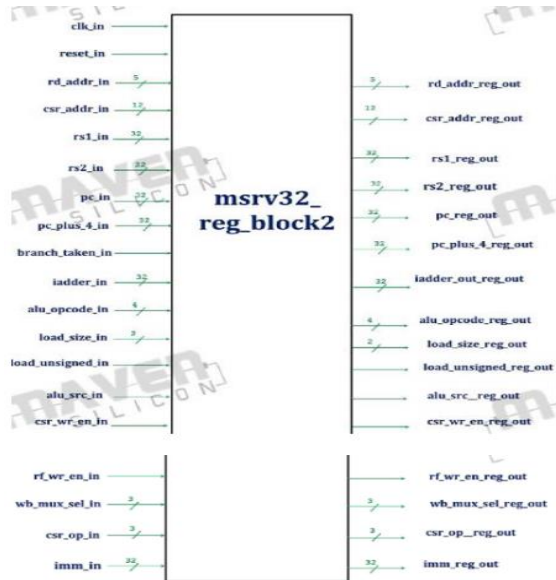


NETLIST:



10.REGISTER BLOCK_2:

BLOCK DIAGRAM:



FUNCTIONALITY:

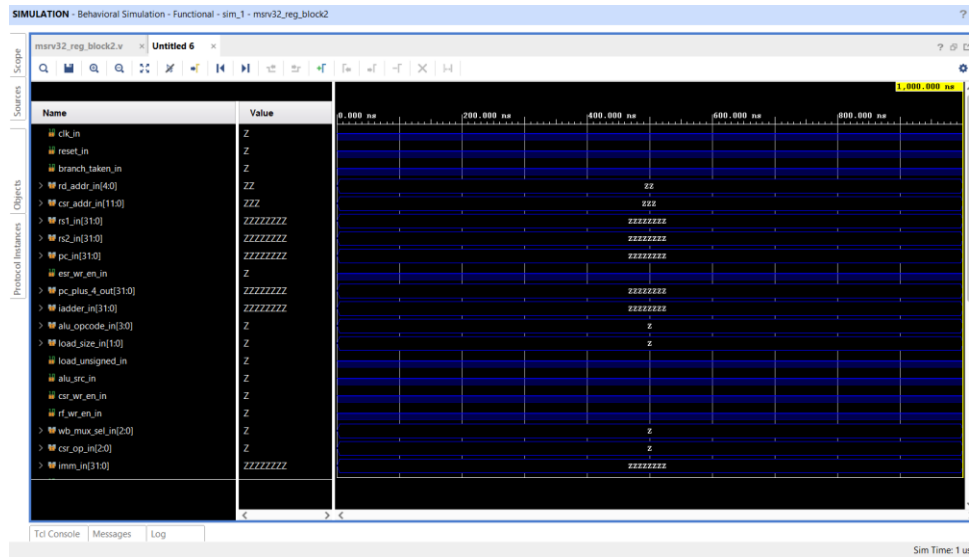
It registers all the inputs and produces the outputs the outputs at the posedge of `clk` if there is no reset. The block also integrates a 2:1 MUX with the select line as `branch_taken_in`. The LSB of `iadder_out_reg_out` is assigned with 0 if `branch_taken_in` is 1 else `iadder_out_reg_out[0]` is assigned with register value of `iadder_in[0]`.

SIMULATION OF REGISTER BLOCK_2:

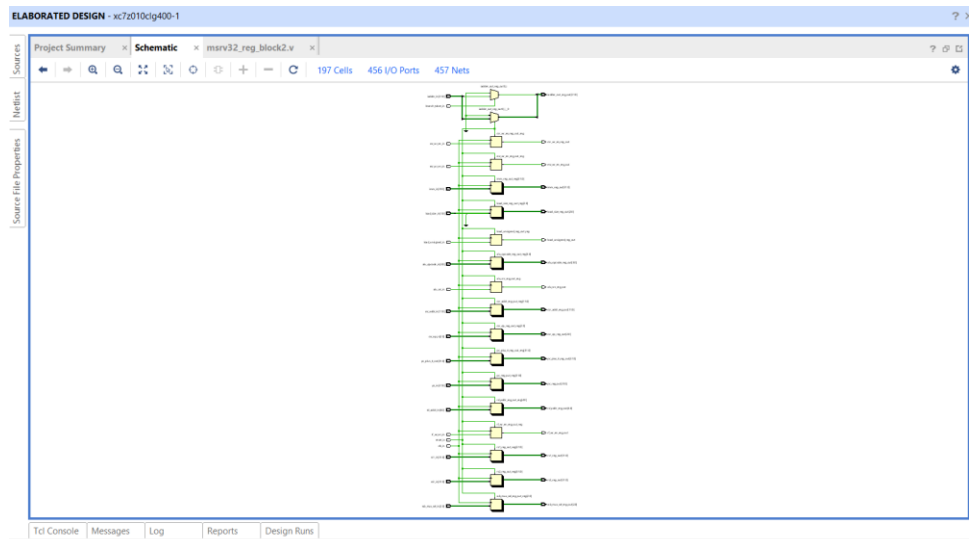
The screenshot shows the Verilog code for the `msrv32_reg_block2` module. The code is as follows:

```
15 //  
16 // Revision:  
17 // Revision 0.01 - File Created  
18 // Additional Comments:  
19 //  
20 ///////////////////////////////////////////////////////////////////  
21  
22 module msrv32_reg_block2 (  
23     input clk_in,  
24     input reset_in,branch_taken_in,  
25     input [4:0] rd_addr_in,  
26     input [11:0] csr_addr_in,  
27     input [31:0] rs1_in,  
28     input [31:0] rs2_in,  
29     input [31:0] pc_in,  
30     input csr_wr_en_in,  
31     input [31:0] pc_plus_4_out,  
32     input [31:0] iadder_in,  
33     input [3:0] alu_opcode_in,  
34     input [1:0] load_size_in,  
35     input load_unsigned_in,alu_src_in,csr_wr_en_in,rf_wr_en_in,  
36     input [2:0] wb_mux_sel_in,csr_op_in,  
37     input [31:0] imm_in,  
38     output reg csr_wr_en_reg_out,  
39     output reg [4:0] rd_addr_reg_out,  
40     output reg [11:0] csr_addr_reg_out,  
41     output reg [31:0] rs1_reg_out,  
42     output reg [31:0] rs2_reg_out,  
43     output reg [31:0] pc_reg_out,  
44     output reg [31:0] pc_plus_4_reg_out,  
45     output reg [31:0] iadder_out_reg_out,  
46     output reg [3:0] alu_opcode_reg_out  
47 )  
48  
49     // Register declarations  
50     reg [31:0] csr_wr_en_reg;  
51     reg [4:0] rd_addr_reg;  
52     reg [11:0] csr_addr_reg;  
53     reg [31:0] rs1_reg;  
54     reg [31:0] rs2_reg;  
55     reg [31:0] pc_reg;  
56     reg [31:0] pc_plus_4_reg;  
57     reg [31:0] iadder_reg;  
58     reg [3:0] alu_opcode_reg;  
59  
60     // Initial values  
61     csr_wr_en_reg <- 0;  
62     rd_addr_reg <- 0;  
63     csr_addr_reg <- 0;  
64     rs1_reg <- 0;  
65     rs2_reg <- 0;  
66     pc_reg <- 0;  
67     pc_plus_4_reg <- 0;  
68     iadder_reg <- 0;  
69     alu_opcode_reg <- 0;  
70  
71     // Register update logic  
72     always @(posedge clk_in) begin  
73         if (reset_in) begin  
74             csr_wr_en_reg <- 0;  
75             rd_addr_reg <- 0;  
76             csr_addr_reg <- 0;  
77             rs1_reg <- 0;  
78             rs2_reg <- 0;  
79             pc_reg <- 0;  
80             pc_plus_4_reg <- 0;  
81             iadder_reg <- 0;  
82             alu_opcode_reg <- 0;  
83         end else begin  
84             csr_wr_en_reg <- csr_wr_en_in;  
85             rd_addr_reg <- rd_addr_in;  
86             csr_addr_reg <- csr_addr_in;  
87             rs1_reg <- rs1_in;  
88             rs2_reg <- rs2_in;  
89             pc_reg <- pc_in;  
90             pc_plus_4_reg <- pc_plus_4_out;  
91             iadder_reg <- iadder_in;  
92             alu_opcode_reg <- alu_opcode_in;  
93         end  
94     end  
95  
96     // Output assignments  
97     csr_wr_en_reg_out <- csr_wr_en_reg;  
98     rd_addr_reg_out <- rd_addr_reg;  
99     csr_addr_reg_out <- csr_addr_reg;  
100    rs1_reg_out <- rs1_reg;  
101    rs2_reg_out <- rs2_reg;  
102    pc_reg_out <- pc_reg;  
103    pc_plus_4_reg_out <- pc_plus_4_reg;  
104    iadder_out_reg_out <- iadder_reg;  
105    alu_opcode_reg_out <- alu_opcode_reg;  
106 end
```

WAVE FORM:

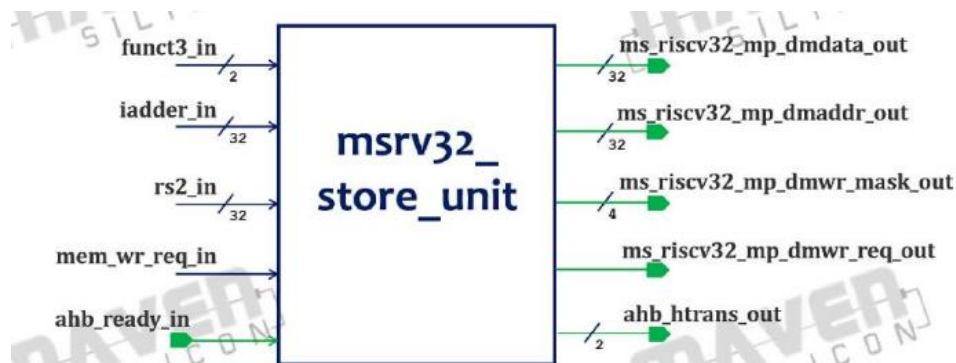


NETLIST:



11.STORE_UNIT

BLOCK DIAGRAM:



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=2'600 a byte of data will be stored in data out depending on the iaddr_in [1:0]. For ex. If iaddr_in=2'500 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=2'601, 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.

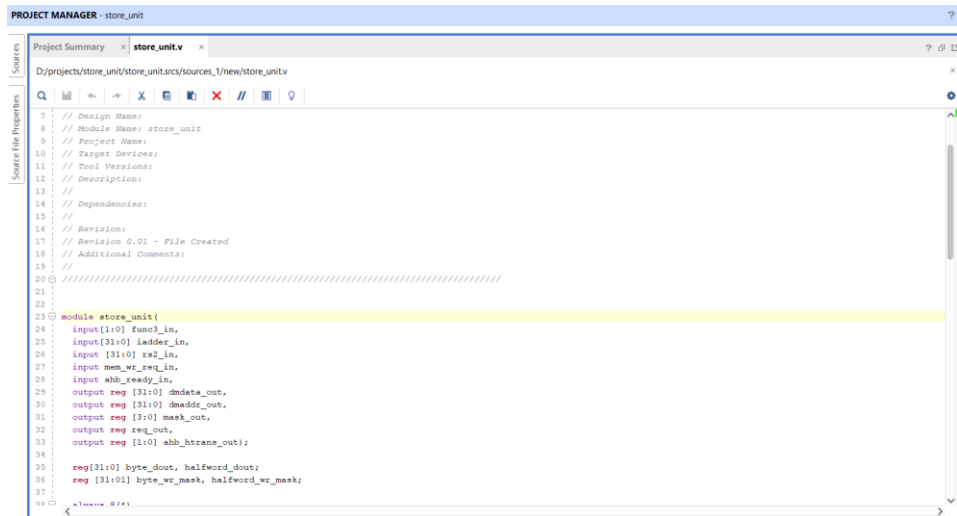
2. Depending on the funct3 in signal wr_mask_out signal will get values. If funct3_in=2'b00, depending on the iaddr in [1:0], ie. if iaddr_in=2'b01 then wr_mask_out (2'b0, mem_wr_req_in, 1'b0) If funct3_in=2'601, depending on the iaddr_in [1] ie. if iaddr_in [1] =1'b1 then wr_mask_out=((2'mem_wr_req_in), 2'b0) and for other combinations word of data ([4'mem_wr_req_in]) will be stored.

3. The dm_addr out should be aligned with respect to (iaddr_in [31:2], 2'600).

4. The wr_req out should be aligned with respect to mem_wr_req_in.

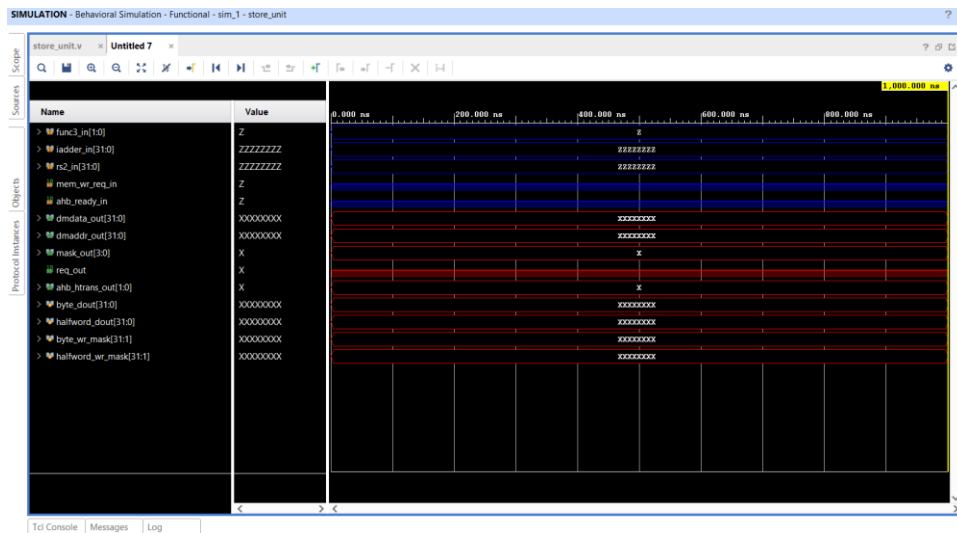
5. The ahb_htrans_out is 2'610 during a valid store instruction and is 2'600 when the transfer is completed.

SIMULATION OF STORE_UNIT:

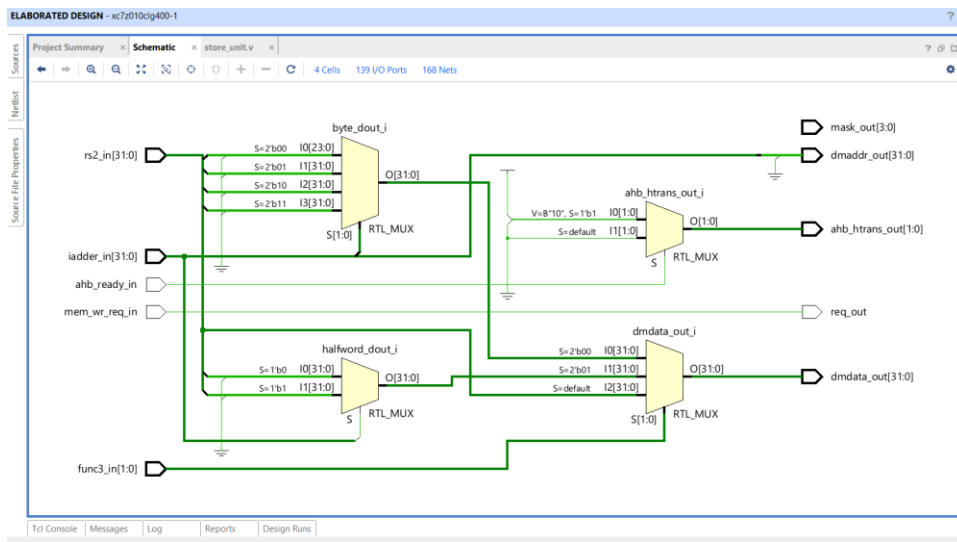


```
7 // Design Name:
8 // Module Name: store_unit
9 // Project Name:
10 // Target Device:
11 // Tool Versions:
12 // Description:
13 //
14 // Dependencies:
15 //
16 // Revision:
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///////////////////////////////////////////////////////////////////
21
22
23 module store_unit(
24     input [1:0] func3_in,
25     input [31:0] iaddr_in,
26     input [31:0] rs2_in,
27     input mem_wr_req_in,
28     input ahb_ready_in,
29     output reg [31:0] dmdata_out,
30     output reg [31:0] dmdadr_out,
31     output reg [3:0] mask_out,
32     output reg req_out,
33     output reg [1:0] ahb_htrans_out);
34
35     reg [31:0] byte_dout, halfword_dout;
36     reg [31:0] byte_wr_mask, halfword_wr_mask;
37
38     //1. store. 0.1.1.
```

WAVEFORM :

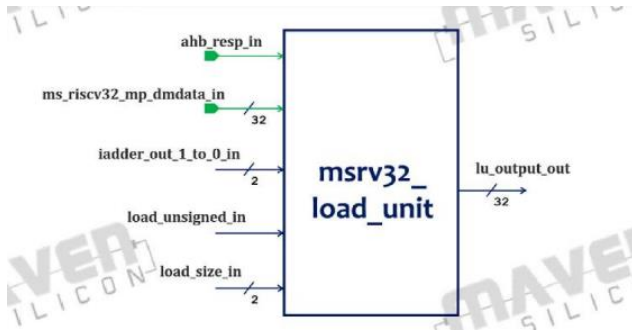


NETLIST:



12.LOAD_UNIT:

BLOCK DIAGRAM



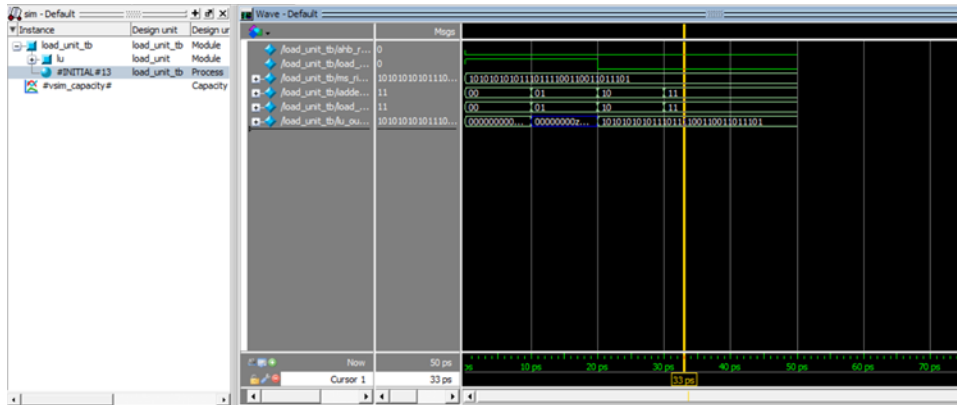
FUNCTIONALITY:

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.

SIMULATION OF LOAD_UNIT:

```
1 module load_unit (  
2     input ahb_resp_in, load_unsigned_in,  
3     input [31:0] ms_riscv32_mp_dm_data_in,  
4     input [1:0] iadder_out_1to0_in, load_size_in,  
5     output reg [31:0] lu_output_out;  
6  
7     reg [7:0] data_byte;  
8     reg [15:0] data_half;  
9     wire [23:0] byte_ext;  
10    wire [15:0] half_ext;  
11  
12    always @(*) begin  
13        if (ahb_resp_in)  
14            case (load_size_in)  
15                2'b00: lu_output_out = {byte_ext, data_byte};  
16                2'b01: lu_output_out = {half_ext, data_half};  
17                2'b10: lu_output_out = ms_riscv32_mp_dm_data_in;  
18                2'b11: lu_output_out = ms_riscv32_mp_dm_data_in;  
19            endcase  
20        else  
21            lu_output_out = 32'd0;  
22        end  
23    end  
24  
25    always @(*) begin  
26        case (iadder_out_1to0_in)  
27            2'b00: data_byte = ms_riscv32_mp_dm_data_in[7:0];  
28            2'b01: data_byte = ms_riscv32_mp_dm_data_in[15:8];  
29            2'b10: data_byte = ms_riscv32_mp_dm_data_in[23:16];  
30        end  
31    end  
32 end
```


WAVE FORM:



13.ARUTHMETIC LOGIC UNIT(ALU):

BLOCK DIAGRAM:



FUNCTIONALITY:

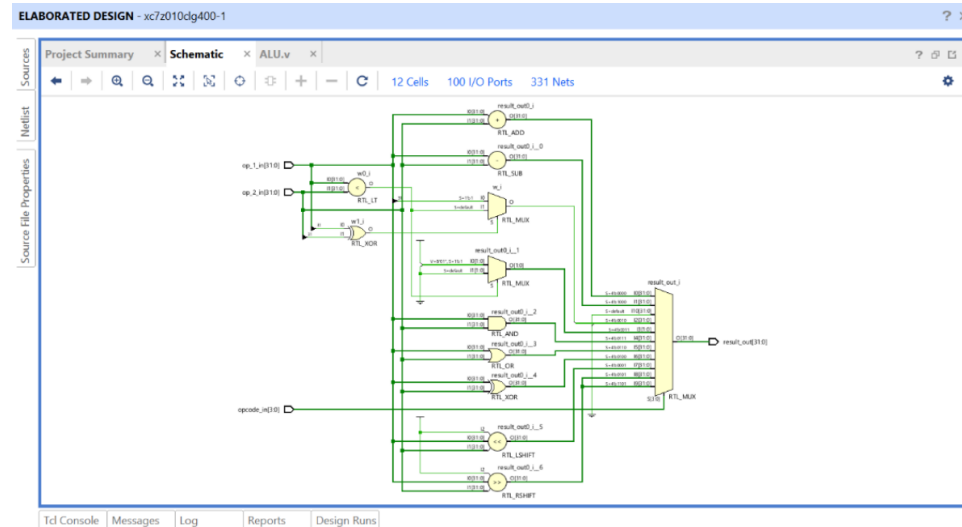
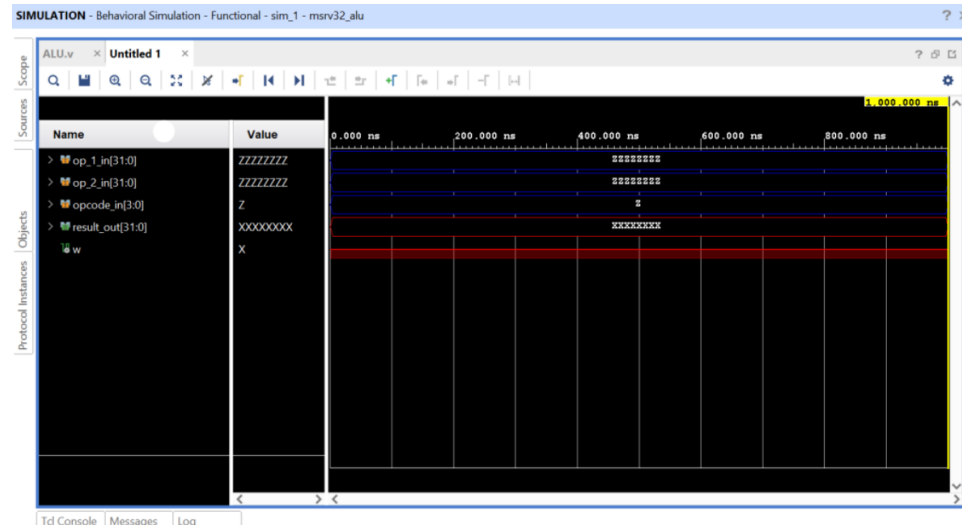
The ALU applies ten distinct logical and arithmetic operations in parallel to two operations in 32-bit operands, outputting the result selected by opcode_in. The ALU input/output signals and the opcodes are shown in tables below.

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.

SIMULATION OF ALU:

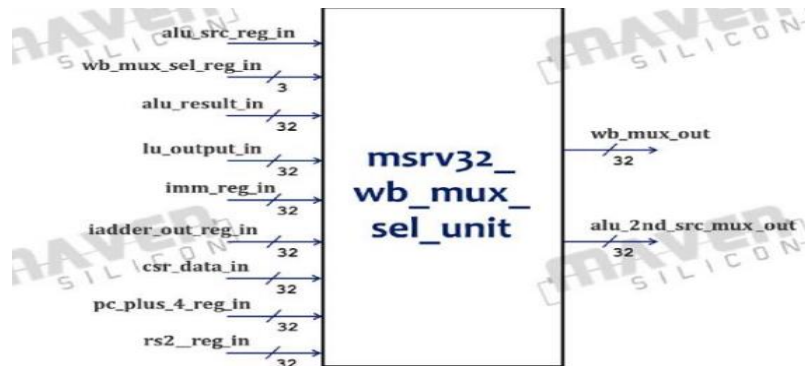
```
sim - Default | C:/altera/13.0sp1/alu.v (alu_tb/alu_unit) - Default
Instance | Design unit | Desi
alu_tb | alu_tb | Mod
alu_unit | alu | Mod
#INITIAL#10 | alu_tb | Proo
#vsim_capacity# | | Cap

Ln# |
1 | module alu (
2 | input [31:0] op1_in,op2_in,
3 | input [3:0] opcode_in,
4 | output reg [31:0] result_out);
5 | wire w;
6 |
7 | assign w = (op1_in[31]~op2_in[31]) ? (op1_in[31]):(op1_in~op2_in);
8 |
9 | always @(*) begin
10 | case (opcode_in)
11 | 4'b0000: result_out = op1_in+op2_in;
12 | 4'b1000: result_out = op1_in~op2_in;
13 | 4'b0010: result_out = (([31:1'b0]),w);
14 | 4'b0011: result_out = (op1_in~op2_in)?1'b1:1'b0;
15 | 4'b0111: result_out = op1_in&op2_in;
16 | 4'b0110: result_out = op1_in|op2_in;
17 | 4'b0100: result_out = op1_in*op2_in;
18 | 4'b0001: result_out = op1_in<<op2_in;
19 | 4'b0101: result_out = op1_in>>op2_in;
20 | 4'b1101: result_out = op1_in>>>op2_in;
21 | default: result_out = 32'h00000000;
22 | endcase
23 | end
24 |
25 | endmodule
```



14.WB_MUX:

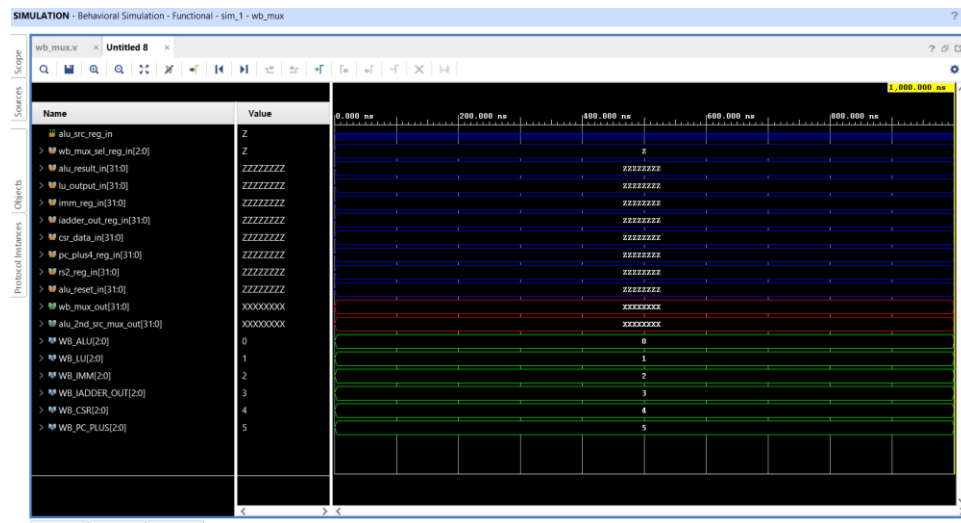
BLOCK DIAGRAM:



SIMULATION OF WB_MUX:

```
PROJECT MANAGER - wb_mux
Project Summary | wb_mux.v
D:\projects\wb_mux\wb_mux\srcs\sources_1\new\wb_mux.v
Sources
Source File Properties
22
23 module wb_mux (
24   input alu_src_reg_in,
25   input [2:0] wb_mux_sel_reg_in,
26   input [31:0] alu_result_in, lu_output_in, imm_reg_in, iadder_out_reg_in, csr_data_in, pc_plus4_reg_in, rs2_reg_in, alu_reset_in,
27   output reg [31:0] wb_mux_out,
28   output [31:0] alu_2nd_src_mux_out
29 );
30
31 parameter WB_ALU = 3'b000,
32 WB_LD = 3'b001,
33 WB_STOR = 3'b010,
34 WB_IADDR_OUT = 3'b011,
35 WB_CSR = 3'b100,
36 WB_PC_PLUS = 3'b101;
37
38 assign alu_2nd_src_mux_out = (alu_src_reg_in) ? rs2_reg_in : imm_reg_in;
39
40 always @(*) begin
41   case (wb_mux_sel_reg_in)
42     WB_ALU: wb_mux_out <= alu_result_in;
43     WB_LD: wb_mux_out <= lu_output_in;
44     WB_STOR: wb_mux_out <= imm_reg_in;
45     WB_IADDR_OUT: wb_mux_out <= iadder_out_reg_in;
46     WB_CSR: wb_mux_out <= csr_data_in;
47     WB_PC_PLUS: wb_mux_out <= pc_plus4_reg_in;
48     default: wb_mux_out <= alu_reset_in;
49   endcase
50 end
51
52 endmodule
23:1 Insert Verilog
```

WAVE FORM:



NETLIST:

