RISC-V RV32I RTL DESIGN USING VERILOG HDL

PROJECT REPORT

UNDER GUIDELINES

Of

MAVEN SILICON

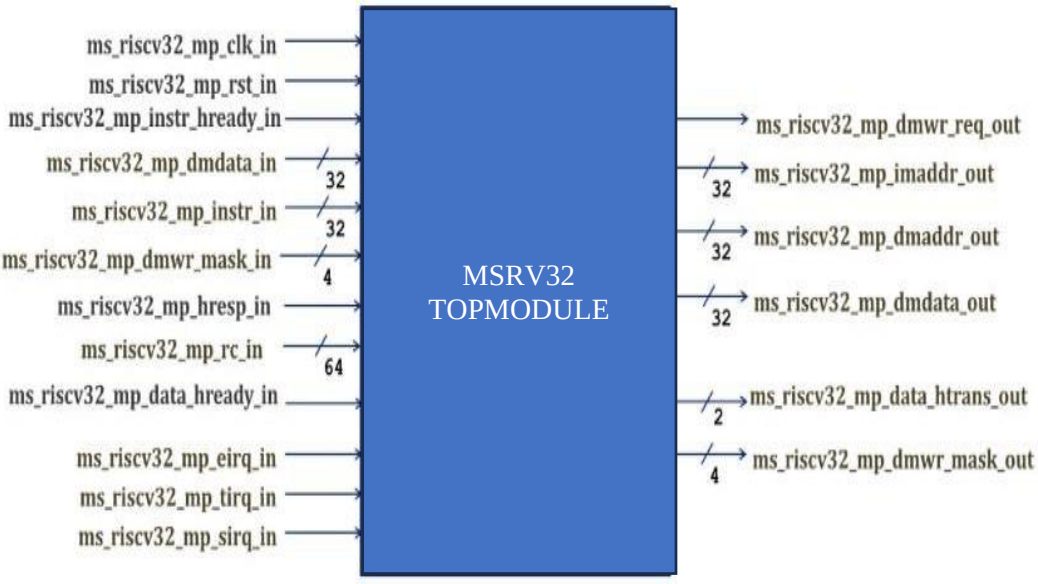
by

NARMATHA A

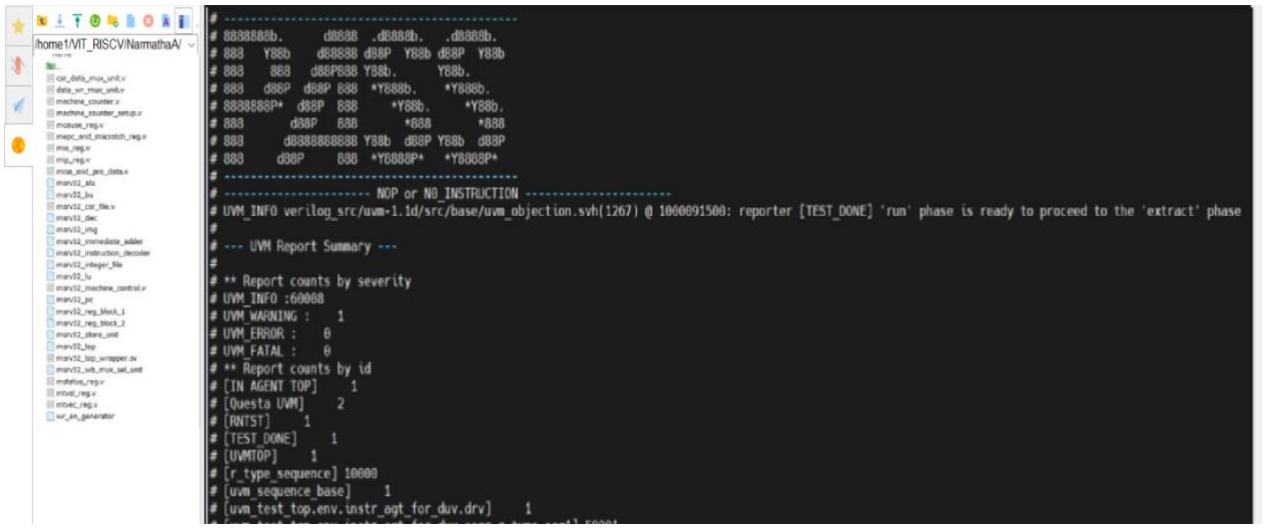
Project Overview:

The RISC-V RV32I RTL Design project aims to design a 32-bit RISC-V processor using the RV32I instruction set architecture.

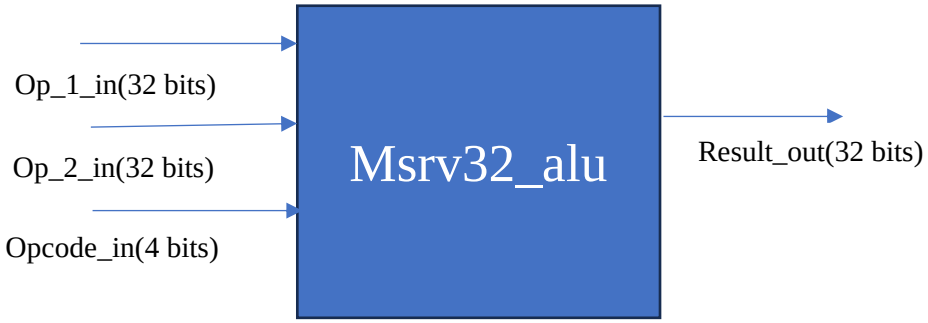
Top Module Block Diagram and Functionality:



FINAL UVM VERIFICATION FOR RISC-V RV32I RTL



ALU BLOCK:



FUNCTIONALITIES:

- **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)

Output: Produces a 32-bit result, result_out, based on the selected operation.

RTL CODE:

C:/intelFPGA/18.1/msrv32_alu.v - Default

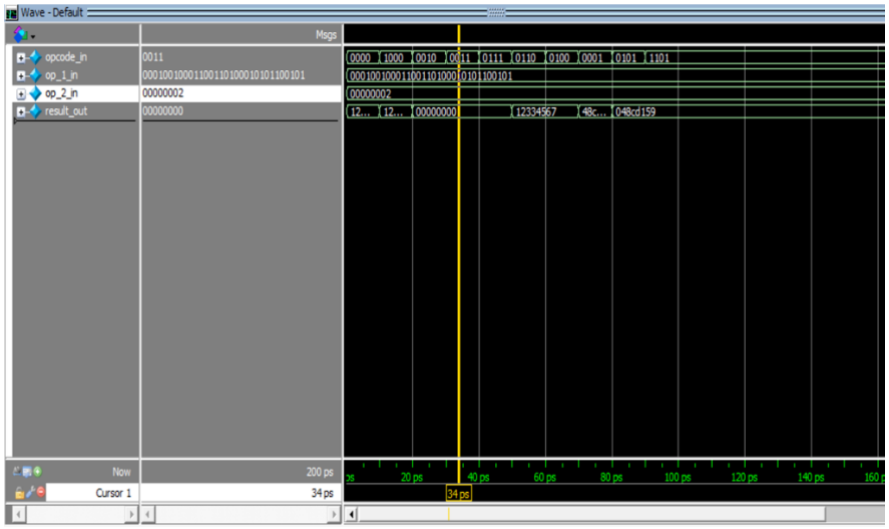
```
Ln#
 1  module alu(input [31:0] op_1_in,
 2                  input [31:0] op_2_in,
 3                  input [3:0] opcode_in,
 4                  output reg [31:0] result_out
 5                 );
 6
 7
 8     parameter  FUNCT3_ADD      =    3'b000;
 9     parameter  FUNCT3_SLT      =    3'b010;
10     parameter  FUNCT3_SLTU     =    3'b011;
11     parameter  FUNCT3_AND      =    3'b111;
12     parameter  FUNCT3_OR       =    3'b110;
13     parameter  FUNCT3_XOR      =    3'b100;
14     parameter  FUNCT3_SLL      =    3'b001;
15     parameter  FUNCT3_SRL      =    3'b101;
16
17
18     wire signed [31:0] signed_op1;
19     wire signed [31:0] adder_op2;
20     wire [31:0] minus_op2;
21     wire [31:0] sra_result;
22     wire [31:0] srl_result;
23     wire [31:0] shr_result;
24     wire slt_result;
25     wire sltu_result;
26
27     reg [31:0] pre_result;
28
29     assign signed_op1 = op_1_in;
30     assign minus_op2 = -op_2_in;
31     assign adder_op2 = opcode_in[3] == 1'b1 ? minus_op2 : op_2_in;
32     assign sra_result = signed_op1 >>> op_2_in[4:0];
33     assign srl_result = op_1_in >> op_2_in[4:0];
34     assign shr_result = opcode_in[3] == 1'b1 ? sra_result : srl_result;
35     assign sltu_result = op_1_in < op_2_in;
36     assign slt_result = op_1_in[31] ^ op_2_in[31] ? op_1_in[31] : sltu_result;
37
38
39     always @*
40     begin
```

```
   begin

      case(opcode_in[2:0])

         FUNCT3_ADD  : result_out = op_1_in + adder_op2;
         FUNCT3_SRL  : result_out = shr_result;
         FUNCT3_OR   : result_out = op_1_in | op_2_in;
         FUNCT3_AND  : result_out = op_1_in & op_2_in;
         FUNCT3_XOR  : result_out = op_1_in ^ op_2_in;
         FUNCT3_SLT  : result_out = {31'b0, slt_result};
         FUNCT3_SLTU : result_out = {31'b0, sltu_result};
         FUNCT3_SLL  : result_out = op_1_in << op_2_in[4:0];
         default     : result_out = 32'b0;

      endcase

   end

endmodule
```

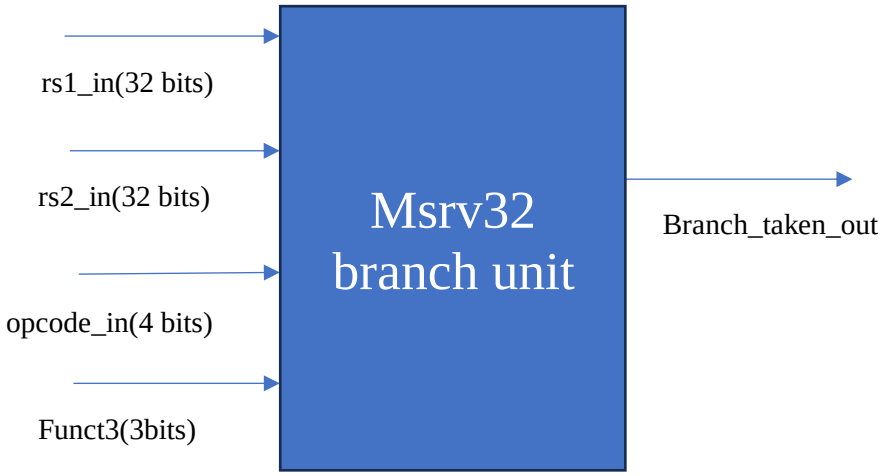
TEST BENCH:

```
module alu_tb();
reg [3:0] opcode_in;
reg [31:0] op_1_in,op_2_in;
wire [31:0] result_out;

msrv32_alu DUT(.opcode_in(opcode_in),.op_1_in(op_1_in),.op_2_in(op_2_in),.result_out(result_out));

initial
begin
opcode_in=4'b0000;op_1_in=32'h12334565;op_2_in=32'h00000002;#10
opcode_in=4'b1000;op_1_in=32'h12334565;op_2_in=32'h00000002;#10
opcode_in=4'b0010;op_1_in=32'h12334565;op_2_in=32'h00000002;#10
opcode_in=4'b0011;op_1_in=32'h12334565;op_2_in=32'h00000002;#10
opcode_in=4'b0111;op_1_in=32'h12334565;op_2_in=32'h00000002;#10
opcode_in=4'b0110;op_1_in=32'h12334565;op_2_in=32'h00000002;#10
opcode_in=4'b0100;op_1_in=32'h12334565;op_2_in=32'h00000002;#10
opcode_in=4'b0001;op_1_in=32'h12334565;op_2_in=32'h00000002;#10
opcode_in=4'b0101;op_1_in=32'h12334565;op_2_in=32'h00000002;#10
opcode_in=4'b1101;op_1_in=32'h12334565;op_2_in=32'h00000002;#10;
end
endmodule
```

WAVE FORM:



BRANCH UNIT



FUNCTIONALITIES:

Internal Logic

- **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.

Branch Conditions

The take flag is calculated based on the funct3_in value and the comparison of rs1_in and rs2_in:

- funct3_in == 3'b000: Branch if equal (BEQ)
- funct3_in == 3'b001: Branch if not equal (BNE)
- funct3_in == 3'b100: Branch if less than or equal (BLEZ)
- funct3_in == 3'b101: Branch if greater than or equal (BGEZ)
- funct3_in == 3'b110: Branch if less than (BLT)
- funct3_in == 3'b111: Branch if greater than or equal (BGE)

RTL CODE:

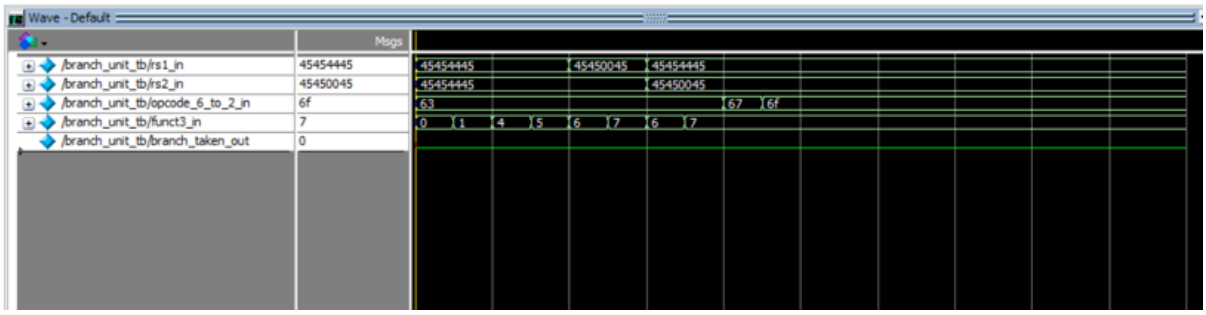
C:/intelFPGA/18.1/msrv32_bu.v - Default

```
Ln#
 1  module msrv32_bu(
 2                   input  [6:2] opcode_6_to_2_in,
 3                   input  [2:0] funct3_in,
 4                   input  [31:0] rs1_in,rs2_in,
 5                   output branch_taken_out
 6                  );
 7
 8     parameter  OPCODE_BRANCH   =    5'b11000;
 9     parameter  OPCODE_JAL      =    5'b11011;
10     parameter  OPCODE_JALR     =    5'b11001;
11
12
13     wire pc_mux_sel;
14     wire pc_mux_sel_en;
15     reg is_branch;
16     reg is_jal;
17     reg is_jalr;
18     wire is_jump;
19     reg take;
20
21     assign is_jump = is_jal | is_jalr;
22     assign pc_mux_sel_en = is_branch | is_jal | is_jalr;
23     assign pc_mux_sel = (is_jump == 1'b1) ? 1'b1 : take;
24     assign branch_taken_out = pc_mux_sel_en & pc_mux_sel;
25
26
27     always @(*)
28     begin
29        case (funct3_in)
30
31           3'b000  : take = (rs1_in == rs2_in);
32           3'b001  : take = !(rs1_in == rs2_in);
33           3'b100  : take = rs1_in[31] ^ rs2_in[31] ? rs1_in[31] : (rs1_in < rs2_in);
34           3'b101  : take = rs1_in[31] ^ rs2_in[31] ? ~rs1_in[31] : !(rs1_in < rs2_in);
35           3'b110  : take = (rs1_in < rs2_in);
36           3'b111  : take = !(rs1_in < rs2_in);
37           default : take = 1'b0;
38
39        endcase
40     end

   endmodule
```

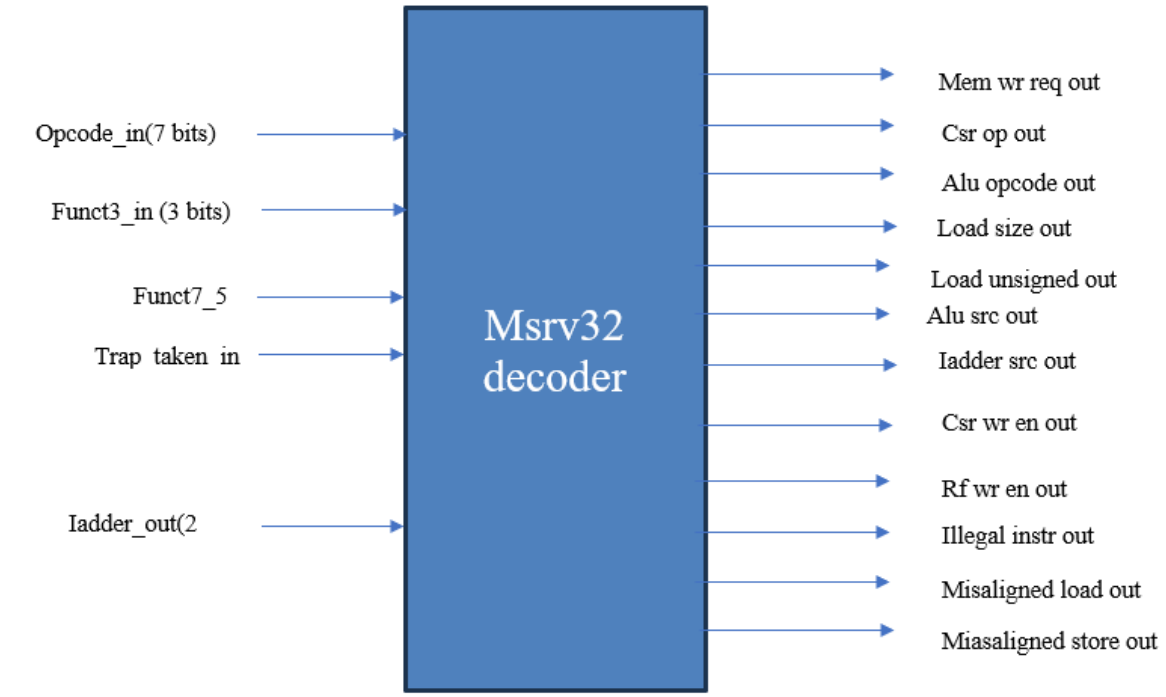
TEST BENCH

```
module branch_unit_tb();
reg[31:0] rs1_in,rs2_in;
reg[6:0] opcode_6_to_2_in;
reg[2:0] funct3_in;
wire branch_taken_out;

rsrv32_bu DUT(.rs1_in(rs1_in),.rs2_in(rs2_in),.opcode_6_to_2_in(opcode_6_to_2_in),.funct3_in(funct3_in),.branch_taken_out(branch_taken_out));

initial
begin
opcode_6_to_2_in=7'b1100011;funct3_in=3'b000;rs2_in=32'h45454445;rs1_in=32'h45454445;#10
opcode_6_to_2_in=7'b1100011;funct3_in=3'b001;rs2_in=32'h45454445;rs1_in=32'h45454445;#10
opcode_6_to_2_in=7'b1100011;funct3_in=3'b100;rs2_in=32'h45454445;rs1_in=32'h45454445;#10
opcode_6_to_2_in=7'b1100011;funct3_in=3'b101;rs2_in=32'h45454445;rs1_in=32'h45454445;#10
opcode_6_to_2_in=7'b1100011;funct3_in=3'b110;rs2_in=32'h45454445;rs1_in=32'h45450045;#10
opcode_6_to_2_in=7'b1100011;funct3_in=3'b111;rs2_in=32'h45454445;rs1_in=32'h45450045;#10
opcode_6_to_2_in=7'b1100011;funct3_in=3'b110;rs2_in=32'h45450045;rs1_in=32'h45454445;#10
opcode_6_to_2_in=7'b1100011;funct3_in=3'b111;rs2_in=32'h45450045;rs1_in=32'h45454445;#10
opcode_6_to_2_in=7'b1100111;funct3_in=3'b111;rs2_in=32'h45450045;rs1_in=32'h45454445;#10
opcode_6_to_2_in=7'b1101111;funct3_in=3'b111;rs2_in=32'h45450045;rs1_in=32'h45454445;#10;
end
endmodule
```

WAVE FORM:



DECODER



FUNCTIONALITES:

Internal Logic

1. **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).
    - Sets flags for instruction types such as is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, and is_system.
2. **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).
    - Sets alu_opcode_out[2:0] based on funct3_in.
    - Sets alu_opcode_out[3] based on funct7_5_in and whether the instruction is an immediate type ALU operation.
3. **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.

4. **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.
5. **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.
6. **Trap Handling:**
   - Misalignment checks (misaligned_load_out, misaligned_store_out) ensure proper handling of memory alignment issues.

RTL CODE

C:/intelFPGA/18.1/msrv32_dec.v - Default *

```
Ln#
 1  module msrv32_dec(
 2                    input [6:0] opcode_in,
 3                    input funct7_5_in,
 4                    input [2:0] funct3_in,
 5                    input [1:0] iadder_1_to_0_in,
 6                    input trap_taken_in,
 7
 8                    output [3:0] alu_opcode_out,
 9                    output mem_wr_req_out,
10                    output [1:0] load_size_out,
11                    output load_unsigned_out,
12                    output alu_src_out,
13                    output iadder_src_out,
14                    output csr_wr_en_out,
15                    output rf_wr_en_out,
16                    output [2:0] wb_mux_sel_out,
17                    output [2:0] imm_type_out,
18                    output [2:0] csr_op_out,
19                    output illegal_instr_out,
20                    output misaligned_load_out,
21                    output misaligned_store_out
22                   );
23
24
25      parameter  OPCODE_OP       =    5'b01100;
26      parameter  OPCODE_OP_IMM   =    5'b00100;
27      parameter  OPCODE_LOAD     =    5'b00000;
28      parameter  OPCODE_STORE    =    5'b01000;
29      parameter  OPCODE_BRANCH   =    5'b11000;
30      parameter  OPCODE_JAL      =    5'b11011;
31      parameter  OPCODE_JALR     =    5'b11001;
32      parameter  OPCODE_LUI      =    5'b01101;
33      parameter  OPCODE_AUIPC    =    5'b00101;
34      parameter  OPCODE_MISC_MEM =    5'b00011;
35      parameter  OPCODE_SYSTEM   =    5'b11100;
36
37
38      parameter  FUNCT3_ADD      =    3'b000;
39      parameter  FUNCT3_SUB      =    3'b000;
40      parameter  FUNCT3_SLT      =    3'b010;
```

Library | Project | alu.v | msrv32_alu.v | msrv32_bu.v | msrv32_dec.v

```
parameter  FUNCT3_SLT      =    3'b010;
parameter  FUNCT3_SLTU     =    3'b011;
parameter  FUNCT3_AND      =    3'b111;
parameter  FUNCT3_OR       =    3'b110;
parameter  FUNCT3_XOR      =    3'b100;
parameter  FUNCT3_SLL      =    3'b001;
parameter  FUNCT3_SRL      =    3'b101;
parameter  FUNCT3_SRA      =    3'b101;

reg is_branch;
reg is_jal;
reg is_jalr;
reg is_auipc;
reg is_lui;
reg is_load;
reg is_store;
reg is_system;
wire is_csr;
reg is_op;
reg is_op_imm;
reg is_misc_mem;
reg is_addi;
reg is_slti;
reg is_sltiu;
reg is_andi;
reg is_ori;
reg is_xori;
wire is_implemented_instr;
wire mal_word;
wire mal_half;
wire misaligned;


always @*
begin
   case(opcode_in [6:2])
      OPCODE_OP       : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b10000000000;
      OPCODE_OP_IMM   : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b01000000000;
      OPCODE_LOAD     : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00100000000;
      OPCODE_STORE    : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00010000000;
```

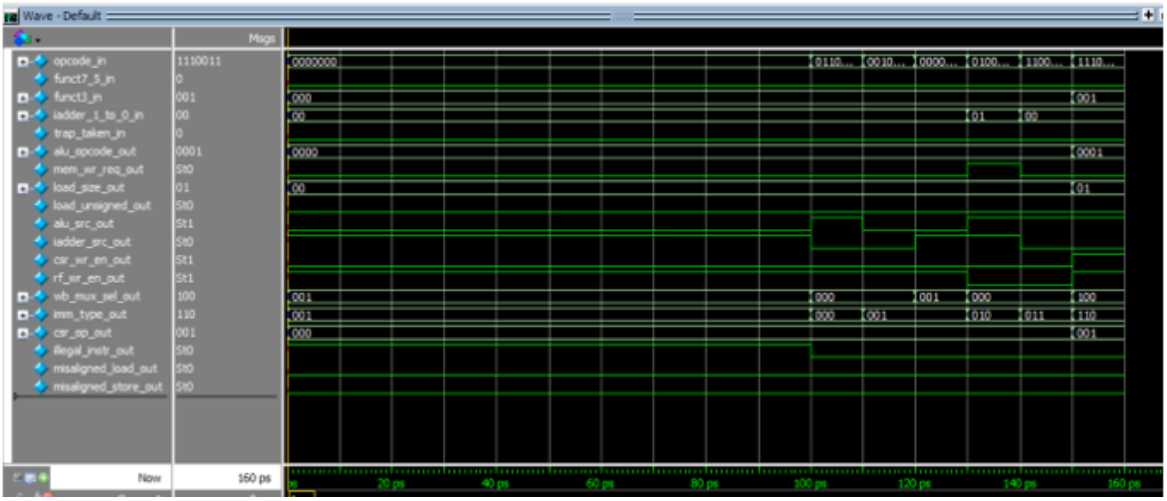
```
        OPCODE_OP         : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b10000000000;
        OPCODE_OP_IMM     : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b01000000000;
        OPCODE_LOAD       : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00100000000;
        OPCODE_STORE      : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00010000000;
        OPCODE_BRANCH     : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00001000000;
        OPCODE_JAL        : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00000100000;
        OPCODE_JALR       : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00000010000;
        OPCODE_LUI        : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00000001000;
        OPCODE_AUIPC      : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00000000100;
        OPCODE_MISC_MEM   : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00000000010;
        OPCODE_SYSTEM     : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00000000001;
        default           : {is_op, is_op_imm, is_load, is_store, is_branch, is_jal, is_jalr, is_lui, is_auipc, is_misc_mem, is_system} = 11'b00000000000;
    endcase
end

always @*
begin
    case(funct3_in)
        FUNCT3_ADD       : {is_addi,is_slti,is_sltiu,is_andi,is_ori,is_xori} = {is_op_imm, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0};
        FUNCT3_SLT       : {is_addi,is_slti,is_sltiu,is_andi,is_ori,is_xori} = {1'b0, is_op_imm, 1'b0, 1'b0, 1'b0, 1'b0};
        FUNCT3_SLTU      : {is_addi,is_slti,is_sltiu,is_andi,is_ori,is_xori} = {1'b0, 1'b0, is_op_imm, 1'b0, 1'b0, 1'b0};
        FUNCT3_AND       : {is_addi,is_slti,is_sltiu,is_andi,is_ori,is_xori} = {1'b0, 1'b0, 1'b0, is_op_imm, 1'b0, 1'b0};
        FUNCT3_OR        : {is_addi,is_slti,is_sltiu,is_andi,is_ori,is_xori} = {1'b0, 1'b0, 1'b0, 1'b0, is_op_imm, 1'b0};
        FUNCT3_XOR       : {is_addi,is_slti,is_sltiu,is_andi,is_ori,is_xori} = {1'b0, 1'b0, 1'b0, 1'b0, 1'b0, is_op_imm};
        default          : {is_addi,is_slti,is_sltiu,is_andi,is_ori,is_xori} = 6'b000000;
    endcase
end

assign load_size_out = funct3_in[1:0];
assign load_unsigned_out = funct3_in[2];

assign alu_src_out = opcode_in[5];

assign is_csr = is_system & (funct3_in[2] | funct3_in[1] | funct3_in[0]);
assign csr_wr_en_out = is_csr;
assign csr_op_out = funct3_in;

assign is_csr = is_system & (funct3_in[2] | funct3_in[1] | funct3_in[0]);
assign csr_wr_en_out = is_csr;
assign csr_op_out = funct3_in;

assign iadder_src_out = is_load | is_store | is_jalr;
assign rf_wr_en_out = is_lui | is_auipc | is_jalr | is_jal | is_op | is_load | is_csr | is_op_imm;//these operations require the register file
assign alu_opcode_out[2:0] = funct3_in;
assign alu_opcode_out[3] = funct7_5_in & ~(is_addi | is_slti | is_sltiu | is_andi | is_ori | is_xori);//select between identical funct3 operations

assign wb_mux_sel_out[0] = is_load | is_auipc | is_jal | is_jalr;
assign wb_mux_sel_out[1] = is_lui | is_auipc;
assign wb_mux_sel_out[2] = is_csr | is_jal | is_jalr;

assign imm_type_out[0] = is_op_imm | is_load | is_jalr | is_branch | is_jal;
assign imm_type_out[1] = is_store | is_branch | is_csr;
assign imm_type_out[2] = is_lui | is_auipc | is_jal | is_csr;

assign is_implemented_instr = is_op | is_op_imm | is_branch | is_jal | is_jalr | is_auipc | is_lui | is_system | is_misc_mem | is_load | is_store;

assign illegal_instr_out = ~opcode_in[1] | ~opcode_in[0] | ~is_implemented_instr;

assign mal_word = funct3_in[1] & ~funct3_in[0] & (iadder_1_to_0_in[1] | iadder_1_to_0_in[0]);
assign mal_half = ~funct3_in[1] & funct3_in[0] & iadder_1_to_0_in[0];
assign misaligned = mal_word | mal_half;
assign misaligned_store_out = is_store & misaligned;
assign misaligned_load_out = is_load & misaligned;

assign mem_wr_req_out = is_store & ~misaligned & ~trap_taken_in;

endmodule
```

TESTBENCH:

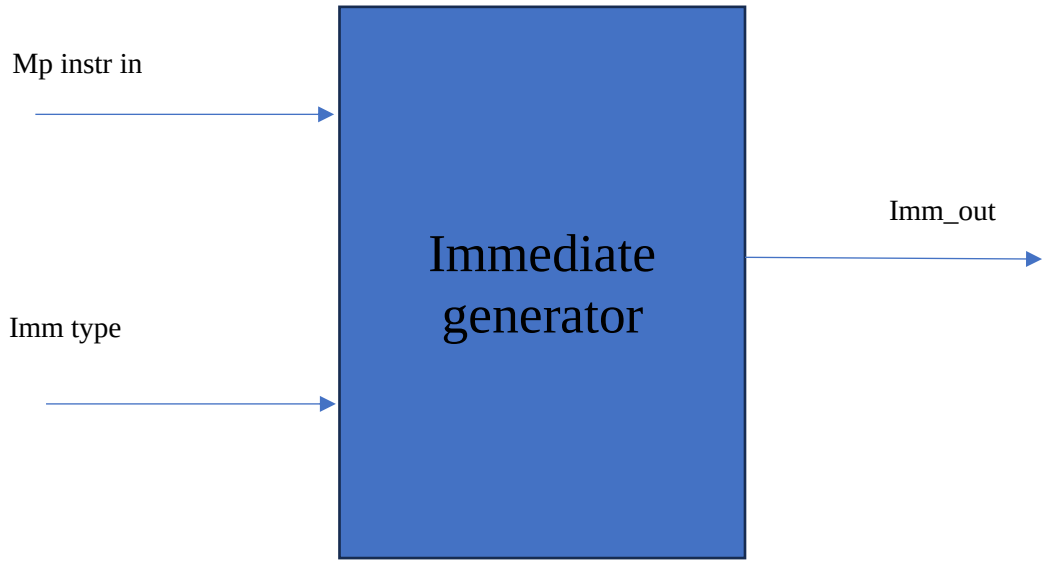
```
module tb_msrv32_dec;

  // Inputs
  reg [6:0] opcode_in;
  reg funct7_5_in;
  reg [2:0] funct3_in;
  reg [1:0] iadder_1_to_0_in;
  reg trap_taken_in;

  // Outputs
  wire [3:0] alu_opcode_out;
  wire mem_wr_req_out;
  wire [1:0] load_size_out;
  wire load_unsigned_out;
  wire alu_src_out;
  wire iadder_src_out;
  wire csr_wr_en_out;
  wire rf_wr_en_out;
  wire [2:0] wb_mux_sel_out;
  wire [2:0] imm_type_out;
  wire [2:0] csr_op_out;
  wire illegal_instr_out;
  wire misaligned_load_out;
  wire misaligned_store_out;

  // Instantiate the Unit Under Test (UUT)
  msrv32_dec uut (
    .opcode_in(opcode_in),
    .funct7_5_in(funct7_5_in),
    .funct3_in(funct3_in),
    .iadder_1_to_0_in(iadder_1_to_0_in),
    .trap_taken_in(trap_taken_in),
    .alu_opcode_out(alu_opcode_out),
    .mem_wr_req_out(mem_wr_req_out),
    .load_size_out(load_size_out),
    .load_unsigned_out(load_unsigned_out),
    .alu_src_out(alu_src_out),
    .iadder_src_out(iadder_src_out),
    .csr_wr_en_out(csr_wr_en_out),
    .rf_wr_en_out(rf_wr_en_out),
    .wb_mux_sel_out(wb_mux_sel_out),
    .imm_type_out(imm_type_out),
    .csr_op_out(csr_op_out),
    .illegal_instr_out(illegal_instr_out),
    .misaligned_load_out(misaligned_load_out),
    .misaligned_store_out(misaligned_store_out)
  );

  initial begin
    // Initialize Inputs
    opcode_in = 0;
    funct7_5_in = 0;
    funct3_in = 0;
    iadder_1_to_0_in = 0;
    trap_taken_in = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here
    $monitor("At time %t, opcode_in = %h, funct3_in = %h, funct7_5_in = %h, iadder_1_to_0_in = %h, trap_taken_in = %h, alu_opcode_out = %h, mem_wr_req_out = %h, load_size_out = %h, load_unsigned_out = %h, al
          $time, opcode_in, funct3_in, funct7_5_in, iadder_1_to_0_in, trap_taken_in, alu_opcode_out, mem_wr_req_out, load_size_out, load_unsigned_out, alu_src_out, iadder_src_out, csr_wr_en_out, rf_wr_en_
    // Test Vector 1: OPCODE_OP
    opcode_in = 7'b0110011;
    funct3_in = 3'b000;
    funct7_5_in = 0;
    iadder_1_to_0_in = 2'b00;
    trap_taken_in = 0;
    #10;

    // Test Vector 2: OPCODE_OP_IMM (ADDI)
    opcode_in = 7'b0010011;
    funct3_in = 3'b000;
    funct7_5_in = 0;
    iadder_1_to_0_in = 2'b00;
    trap_taken_in = 0;
    #10;

    // Test Vector 3: OPCODE_LOAD (LB)
    opcode_in = 7'b0000011;
    funct3_in = 3'b000;
    funct7_5_in = 0;
    iadder_1_to_0_in = 2'b00;
    trap_taken_in = 0;
    #10;
```

```
    // Test Vector 1: OPCODE_OP
    opcode_in = 7'b0110011;
    funct3_in = 3'b000;
    funct7_5_in = 0;
    iadder_1_to_0_in = 2'b00;
    trap_taken_in = 0;
    #10;

    // Test Vector 2: OPCODE_OP_IMM (ADDI)
    opcode_in = 7'b0010011;
    funct3_in = 3'b000;
    funct7_5_in = 0;
    iadder_1_to_0_in = 2'b00;
    trap_taken_in = 0;
    #10;

    // Test Vector 3: OPCODE_LOAD (LB)
    opcode_in = 7'b0000011;
    funct3_in = 3'b000;
    funct7_5_in = 0;
    iadder_1_to_0_in = 2'b00;
    trap_taken_in = 0;
    funct3_in = 3'b000;
    funct7_5_in = 0;
    iadder_1_to_0_in = 2'b00;
    trap_taken_in = 0;
    #10;

    // Test Vector 6: OPCODE_SYSTEM (CSR)
    opcode_in = 7'b1110011;
    funct3_in = 3'b001;
    funct7_5_in = 0;
    iadder_1_to_0_in = 2'b00;
    trap_taken_in = 0;
    #10;

    // Finish the test
    $finish;
  end

endmodule
```

WAVE FORM:



IMMEDIATE GENERATOR:



FUNCTIONALITIES:

Instruction Types and Immediate Extraction:

- **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. This essentially ignores the immediate field in the instruction for R-type.
- **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. The module replicates the sign bit (bit 31) 20 times and concatenates it with instruction bits 31:20 to create the immediate value.
- **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. The module replicates the sign bit (bit 31) 20 times, concatenates it with instruction bits 31:25 and 11:7, forming the immediate value.
- **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.
- **CSR-Type (imm_type_in = 110):** These instructions (e.g., csrrw) use a 5-bit CSR register specifier. The module sets the first 27 bits of imm_out to zero and concatenates them with instruction bits 19:15, forming a 32-bit value with the CSR register number in the lower 5 bits.
- **Invalid (imm_type_in = 111):** This is likely an unused case or an error condition. The module replicates the sign bit (bit 31) 20 times and concatenates it with instruction bits 31:20, similar to R-type behavior.

RTL CODE:

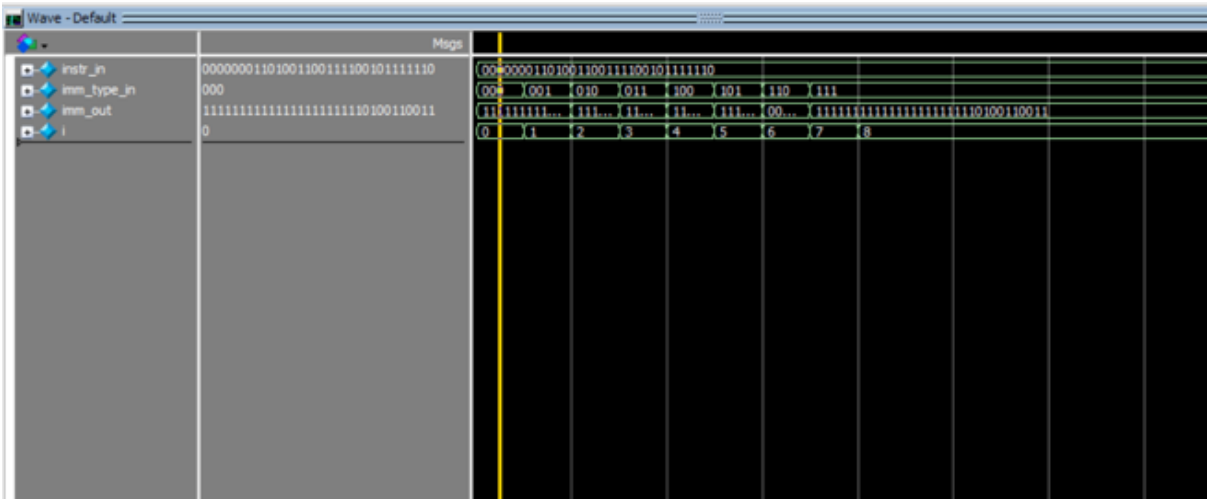
C:/intelFPGA/18.1/msrv32_immediate_generator.v - Default

```
module msrv32_immediate_generator(
                  input  [31:7] instr_in,
                  input  [2:0] imm_type_in,
                  output reg [31:0] imm_out
                 );


   parameter R_TYPE          =    3'b000;
   parameter I_TYPE          =    3'b001;
   parameter S_TYPE          =    3'b010;
   parameter B_TYPE          =    3'b011;
   parameter U_TYPE          =    3'b100;
   parameter J_TYPE          =    3'b101;
   parameter CSR_TYPE        =    3'b110;


   always @(*)
   begin
      case (imm_type_in)
         3'b000   : imm_out = { {20{instr_in[31]}}, instr_in[31:20] };
         I_TYPE   : imm_out = { {20{instr_in[31]}}, instr_in[31:20] };
         S_TYPE   : imm_out = { {20{instr_in[31]}}, instr_in[31:25], instr_in[11:7] };
         B_TYPE   : imm_out = { {19{instr_in[31]}}, instr_in[31], instr_in[7], instr_in[30:25], instr_in[11:8], 1'b0 };
         U_TYPE   : imm_out = { instr_in[31:12], 12'h000 };
         J_TYPE   : imm_out = { {11{instr_in[31]}}, instr_in[31], instr_in[19:12], instr_in[20], instr_in[30:21], 1'b0 };
         CSR_TYPE : imm_out = { 27'b0, instr_in[19:15] };
         3'b111   : imm_out = { {20{instr_in[31]}}, instr_in[31:20] };
      endcase
   end

endmodule
```

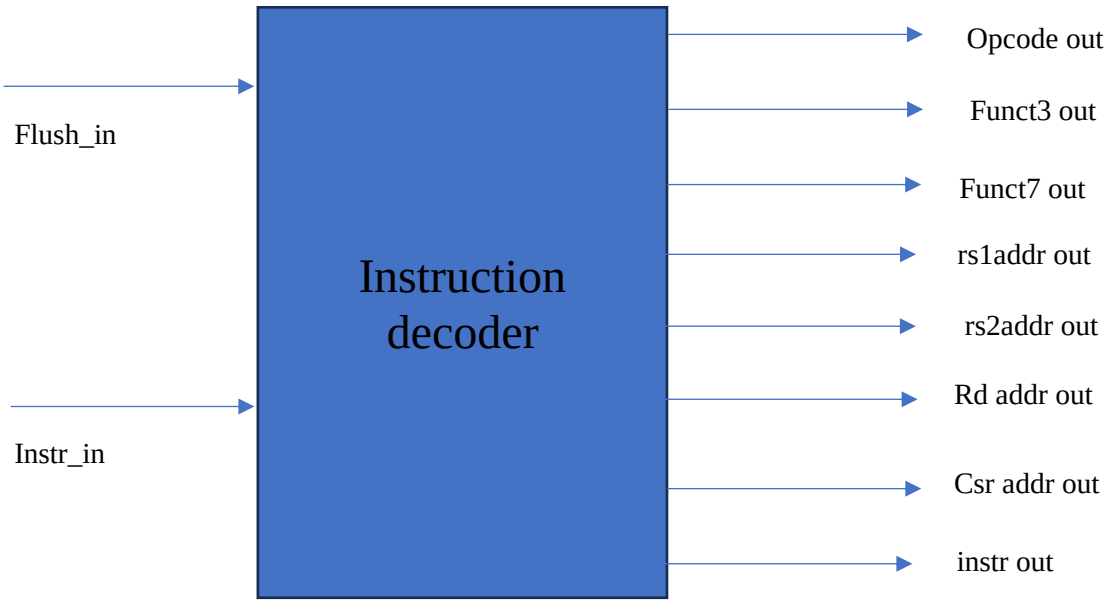
TESTBENCH:

```
module immediate_generator_tb();
reg [31:0]instr_in;
reg [2:0]imm_type_in;
wire [31:0]imm_out;

integer i;

msrv32_imd DUT(.instr_in(instr_in),.imm_type_in(imm_type_in),.imm_out(imm_out));

task initialize;
begin
instr_in=25'b1101001100111100101111110;
imm_type_in=2'b0;
end
endtask

task select(input [2:0]s);
begin
imm_type_in=s;
end
endtask

initial
begin
initialize;
        for(i=0;i<8;i=i+1)
        begin
        select(i);
        #10;
        end
end
endmodule
```

WAVE FORM:



INSTRUCTION DECODER:



FUNCTIONALITIES:

Inputs:

- flush_in: A control signal (1 bit) to reset the instruction decoder.
- instr_in: The 32-bit instruction word to be decoded.
- **Outputs:**
- opcode_out: The opcode field of the instruction (7 bits).
- funct7_out: The funct7 field of the instruction (7 bits). (Might be unused depending on the implementation)
- funct3_out: The funct3 field of the instruction (3 bits).
- rs1_addr_out: The register file address for the first source operand (RS1) (5 bits).
- rs2_addr_out: The register file address for the second source operand (RS2) (5 bits).
- rd_addr_out: The register file address for the destination register (RD) (5 bits).
- csr_addr_out: The CSR (Control and Status Register) address field (12 bits). (Only used for CSR instructions)
- instr_31_7_out: The remaining bits of the instruction from bit 31 to bit 7 (25 bits).

Functionality:

This module acts as a decoder unit for MIPS RV32 instructions. It separates the instruction word (instr_in) into its constituent fields based on their bit positions in the instruction format:

1. **Flush Signal:**
    - The flush_in signal acts as a reset mechanism. If it's high (active), the decoder outputs a constant value of 32'h00000013 (likely an invalid instruction) on all output signals. This might be used to stall the pipeline or signal an error condition.
2. **Instruction Multiplexing:**
    - A wire instr_mux is used to hold the instruction after potential flushing.

- If flush_in is low (inactive), instr_mux is simply assigned the input instruction instr_in.
- If flush_in is high (active), instr_mux is assigned a constant value, effectively replacing the actual instruction with an invalid one.

3. **Field Extraction:**
   - The decoder then extracts specific bit fields from the instr_mux wire based on their known locations within the MIPS RV32 instruction format and assigns them to the corresponding output registers:
     - 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.

RTL CODE:

C:/intelFPGA/18.1/msrv32_instruction_decoder.v - Default *

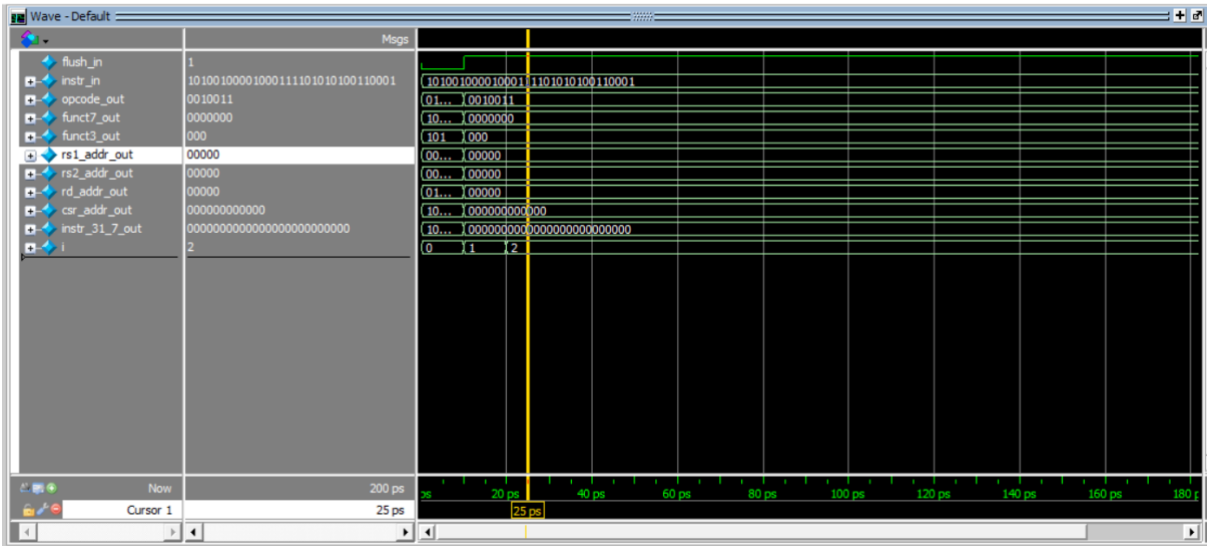
```
Ln#
 1  module msrv32_instruction_decoder(input          flush_in,
 2                                    input  [31:0] instr_in,
 3                                    output [6:0 ] opcode_out,funct7_out,
 4                                    output [2:0 ] funct3_out,
 5                                    output [4:0 ] rs1_addr_out,rs2_addr_out,rd_addr_out,
 6                                    output [11:0] csr_addr_out,
 7                                    output [24:0] instr_31_7_out
 8  );
 9
10     wire [31:0] instr_mux;
11
12
13     assign instr_mux = flush_in ? 32'h00000013 : instr_in;
14
15
16     assign opcode_out       = instr_mux[6:0];
17     assign funct3_out       = instr_mux [14:12];
18     assign funct7_out       = instr_mux [31:25];
19     assign csr_addr_out     = instr_mux[31:20];
20     assign rs1_addr_out     = instr_mux[19:15];
21     assign rs2_addr_out     = instr_mux[24:20];
22     assign rd_addr_out      = instr_mux[11:7];
23     assign instr_31_7_out = instr_mux[31:7];
24
25  endmodule
26
```

TESTBENCH:

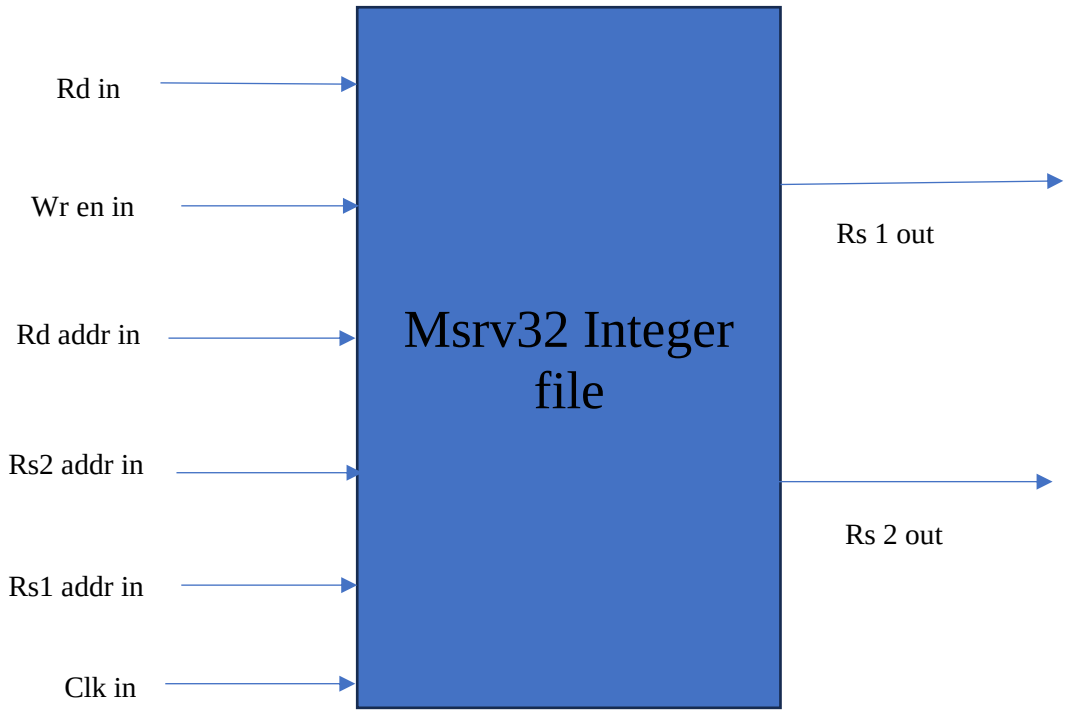
```
Ln#
 1  module instruction_mux_tb();
 2   reg flush_in;
 3   reg  [31:0] instr_in;
 4   wire [6:0]opcode_out,funct7_out;
 5   wire [2:0]funct3_out;
 6   wire [4:0]rsladdr_out,rs2addr_out,rdaddr_out;
 7   wire [11:0]csr_addr_out;
 8   wire [24:0]instr_out;
 9   integer i;
10
11
12   instruction_mux DUT(.flush_in(flush_in),.instr_in(instr_in),.opcode_out(opcode_out),.funct3_out(funct3_out),.funct7_out(funct7_out),.rsladdr_out(rsl
13
14   task initialize;
15   begin
16    instr_in=32'hA423D531;
17   end
18   endtask
19
20   task flu(input s);
21   begin
22    flush_in=s;
23   end
24   endtask
25
26   initial
27   begin
28    initialize;
29    for(i=0;i<2;i=i+1)
```

```
 8   wire [24:0]instr_out;
 9   integer i;
10
11
12   instruction_mux DUT(.flush_in(flush_in),.instr_in(instr_in),.opcode_out(opcode_out),.funct3_out(funct3_out),.funct7_out(funct7_out),.rsladdr_o
13
14   task initialize;
15   begin
16    instr_in=32'hA423D531;
17   end
18   endtask
19
20   task flu(input s);
21   begin
22    flush_in=s;
23   end
24   endtask
25
26   initial
27   begin
28    initialize;
29    for(i=0;i<2;i=i+1)
30   begin
31    flu(i);
32    #10;
33   end
34   end
35   endmodule
36
```

WAVE FORM:



INTEGER FILE



FUNCTIONALITIES:

1. **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).
2. **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.
3. **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.
4. **Forwarding Logic (Datapath Optimization):**
    - The module implements forwarding logic to optimize the datapath.
        - Two signals, fwd_op1_enable and fwd_op2_enable, are calculated to determine if forwarding is necessary.
        - Forwarding occurs when the destination register (RD) of a write operation is the same address as the source register (RS1 or RS2) of a subsequent read operation in the same clock cycle. In this case, the data being written is forwarded directly to the read output, bypassing the register file. This avoids potential delays caused by writing to the register file and then reading from it in the same cycle.
        - The fwd_op1_enable signal is high if rs_1_addr_in matches rd_addr_in and there's a write operation (wr_en_in is high). This enables forwarding for the first operand (RS1).
        - The fwd_op2_enable signal is similar but checks for rs_2_addr_in and potential forwarding for the second operand (RS2).

5. **Reset:**
    - The reset_in signal initializes the register file. If reset is active, all registers in reg_file are set to zero.

RTL CODE:

C:/intelFPGA/18.1/msrv32_integer_file.v - Default

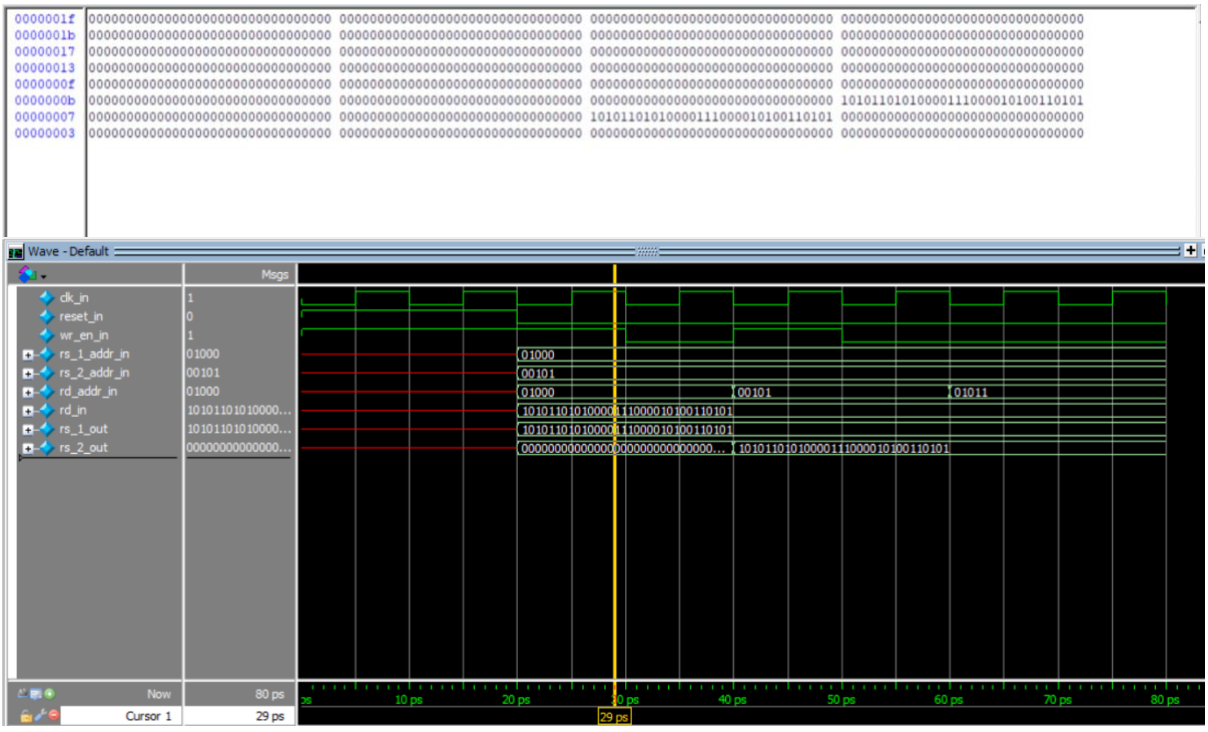
```
Ln#
 1  module msrv32_integer_file( input clk_in, reset_in,
 2
 3                              input  [4:0]  rs_1_addr_in,rs_2_addr_in,
 4                              output [31:0] rs_1_out,rs_2_out,
 5
 6                              input  [4:0]  rd_addr_in,
 7                              input         wr_en_in,
 8                              input  [31:0] rd_in
 9                             );
10
11
12
13      reg [31:0] reg_file [31:0];
14
15
16      wire fwd_op1_enable, fwd_op2_enable;
17      integer i;
18
19
20      assign fwd_op1_enable = (rs_1_addr_in == rd_addr_in && wr_en_in == 1'b1) ? 1'b1 : 1'b0;
21      assign fwd_op2_enable = (rs_2_addr_in == rd_addr_in && wr_en_in == 1'b1) ? 1'b1 : 1'b0;
22
23
24
25     always@(posedge clk_in or posedge reset_in)
26     begin
27        if(reset_in)
28        begin
29        for(i = 0; i < 32; i = i+1)
30           reg_file[i]  = 32'b0;
31        end
32
33        else if(wr_en_in && rd_addr_in)
34        begin
35           reg_file[rd_addr_in] <= rd_in;
36        end
37     end
38
39     assign rs_1_out = fwd_op1_enable == 1'b1 ? rd_in : reg_file[rs_1_addr_in];
40     assign rs_2_out = fwd_op2_enable == 1'b1 ? rd_in : reg_file[rs_2_addr_in];
```

TESTBENCH:

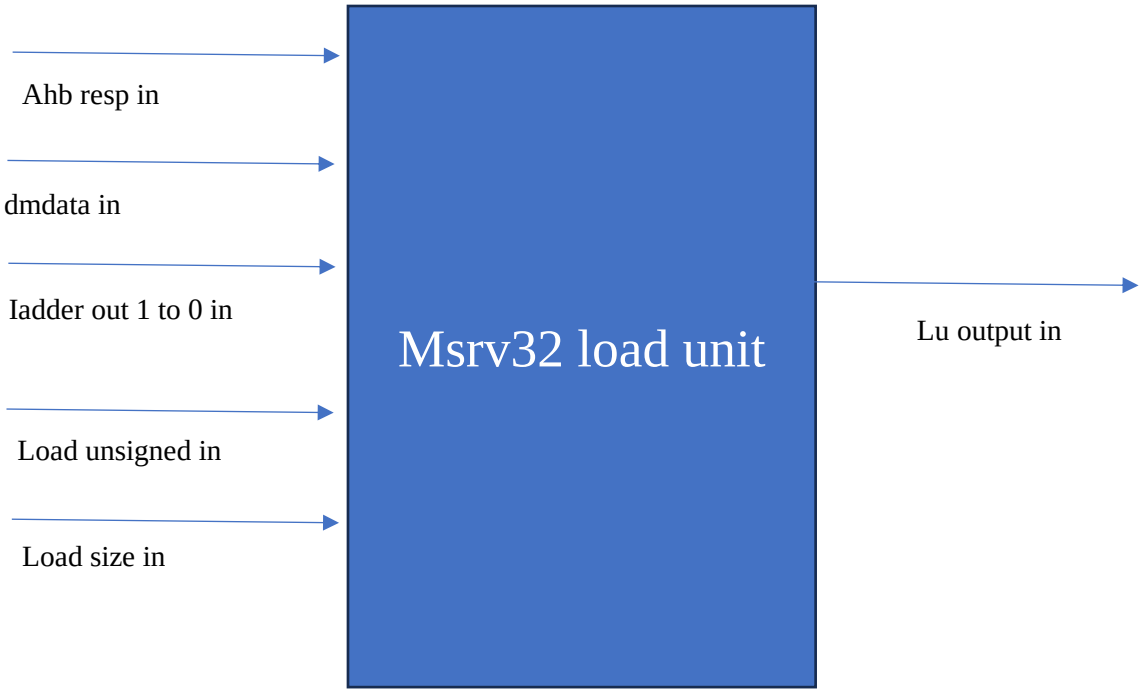
```
module integer_file_tb;
reg clk,rst,wr_en_in;
reg [4:0] rs_1_addr_in,rs_2_addr_in,rd_addr_in;
reg [31:0] rd_in;
wire [31:0]rs_1_out,rs_2_out;

msrv32_integer_file DUT(.clk(clk),.rst(rst),.rs_1_addr_in(rs_1_addr_in),.rs_2_addr_in(rs_2_addr_in),.rd_addr_in(rd_addr_in),.wr_e

always #5 clk=~clk;

initial
begin
clk=0;
rst=1;
wr_en_in=1;
#20
rst=0;
wr_en_in=1;
rd_in=32'hAD438535;
rs_1_addr_in=5'b01000;
rs_2_addr_in=5'b00101;
rd_addr_in=5'b01000;
```

```
#10
wr_en_in=0;
#10
wr_en_in=0;
rd_in=32'hAD438535;
rs_1_addr_in=5'b01000;
rs_2_addr_in=5'b00101;
rd_addr_in=5'b01011;

#10

rst=0;
wr_en_in=0;
rd_in=32'hAD438535;
rs_1_addr_in=5'b01000;
rs_2_addr_in=5'b00101;
rd_addr_in=5'b01011;

#10;
end
endmodule
```

WAVE AND MEMEORY:



LOAD UNIT



FUNCTIONALITIES:

1. **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.
2. **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.
3. **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.

4. **Output Construction:**
    - The module uses three always combinational blocks to handle data selection and sign extension:
        - One block selects the appropriate byte based on iadder_1_to_0_in.
        - Another block selects the appropriate half-word based on iadder_1_to_0_in[1].
        - The third block performs zero extension (if load_unsigned_in is high) or sign extension (if load_unsigned_in is low) for byte and half-word loads.
    - Finally, a case statement within the main always block assembles the final output (lu_output) based on the load_size_in value.
        - For byte loads, the selected byte is concatenated with a zero- or sign-extended upper portion (24 bits) depending on the load_unsigned_in signal.
        - For half-word loads, the selected half-word is concatenated with a zero- or sign-extended upper portion (16 bits).
        - For word loads, the entire data_in is used as the output.
5. **AHB Response (Optional):**
    - The ahb_resp_in signal might be used for error handling or flow control related to the memory access operation. It's not directly involved in the core data selection or extension logic but could be used to stall the load operation or signal an error condition if ahb_resp_in is low (indicating a failed memory access).

RTL CODE:

C:/intelFPGA/18.1/msrv32_lu.v - Default

```
Ln#
 1  module msrv32_lu(
 2                   input  [1:0] load_size_in,
 3                   input  clk_in,
 4                   input  load_unsigned_in,
 5                   input  [31:0] data_in,
 6                   input  [1:0] iadder_1_to_0_in,
 7                   input  ahb_resp_in,
 8                   output reg [31:0] lu_output
 9                   );
10
11
12      reg [7:0] data_byte;
13      reg [15:0] data_half;
14      wire [23:0] byte_ext;
15      wire [15:0] half_ext;
16
17      always @(*)
18      begin
19         begin
20         if(!ahb_resp_in)
21         case(load_size_in)
22
23                2'b00: lu_output = {byte_ext, data_byte};
24                2'b01: lu_output = {half_ext, data_half};
25                2'b10: lu_output = data_in;
26                2'b11: lu_output = data_in;
27
28         endcase
29         end
30      end
31
32      always @*
33      begin
34
35         case(iadder_1_to_0_in)
36
37            2'b00: data_byte = data_in[7:0];
38            2'b01: data_byte = data_in[15:8];
39            2'b10: data_byte = data_in[23:16];
40            2'b11: data_byte = data_in[31:24];
```

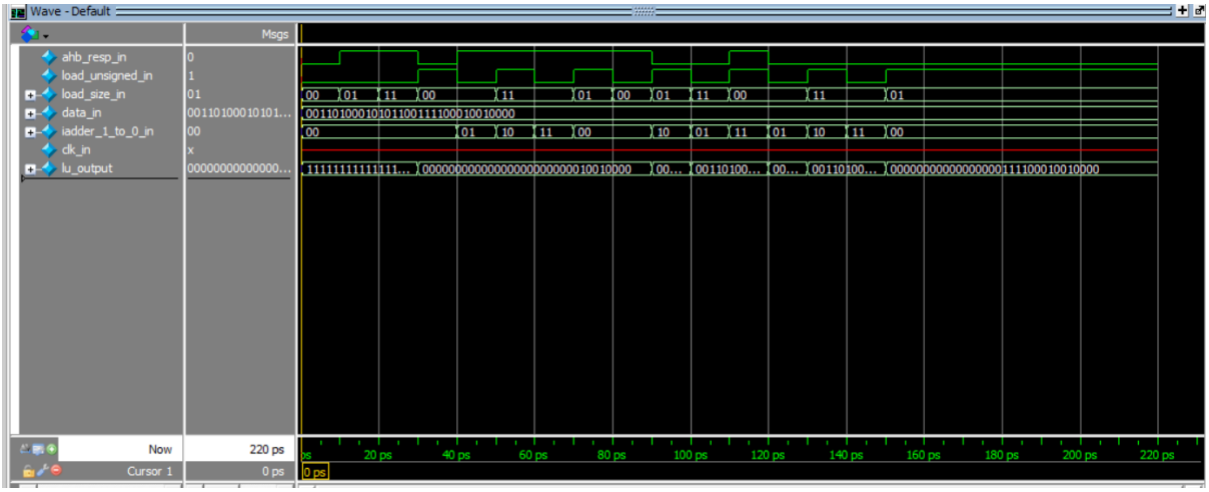
Library | Project | alu.v | msrv32_alu.v | msrv32_lu.v | msrv32_dec.v *

```
Ln#
20      if(!ahb_resp_in)
21      case(load_size_in)
22
23           2'b00: lu_output = {byte_ext, data_byte};
24           2'b01: lu_output = {half_ext, data_half};
25           2'b10: lu_output = data_in;
26           2'b11: lu_output = data_in;
27
28      endcase
29      end
30   end
31
32   always @*
33   begin
34
35      case(iadder_1_to_0_in)
36
37         2'b00: data_byte = data_in[7:0];
38         2'b01: data_byte = data_in[15:8];
39         2'b10: data_byte = data_in[23:16];
40         2'b11: data_byte = data_in[31:24];
41
42      endcase
43
44   end
45
46   always @*
47   begin
48
49      case(iadder_1_to_0_in[1])
50
51         1'b0: data_half = data_in[15:0];
52         1'b1: data_half = data_in[31:16];
53
54      endcase
55
56   end
57
58
59   assign byte_ext = load_unsigned_in == 1'b1 ? 24'b0 : {24{data_byte[7]}};
60   assign half_ext = load_unsigned_in == 1'b1 ? 16'b0 : {16{data_half[15]}};
61
62endmodule
63
```

TESTBENCH:

```
Ln#
 1  module load_unit_tb;
 2  reg ahb_resp_in,load_unsigned_in;
 3  reg [1:0]load_size_in;
 4  reg [31:0]data_in;
 5  reg [1:0]iadder_1_to_0_in;
 6  reg  clk_in;
 7  wire [31:0]lu_output;
 8
 9  msrv32_lu DUT(.clk_in(clk_in),.ahb_resp_in(ahb_resp_in),.data_in(data_in),.iadder_1_to_0_in(iadder_1_to_0_in),.load_unsigned_in(load_unsi
10
11  initial
12  begin
13  data_in=32'h34567890;
14  iadder_1_to_0_in=2'b00;load_unsigned_in=0;load_size_in=2'b00;ahb_resp_in=0;#10
15  iadder_1_to_0_in=2'b00;load_unsigned_in=0;load_size_in=2'b01;ahb_resp_in=1;#10
16  iadder_1_to_0_in=2'b00;load_unsigned_in=0;load_size_in=2'b11;ahb_resp_in=1;#10
17  iadder_1_to_0_in=2'b00;load_unsigned_in=1;load_size_in=2'b00;ahb_resp_in=0;#10
18  iadder_1_to_0_in=2'b01;load_unsigned_in=0;load_size_in=2'b00;ahb_resp_in=1;#10
19  iadder_1_to_0_in=2'b10;load_unsigned_in=1;load_size_in=2'b11;ahb_resp_in=1;#10
20  iadder_1_to_0_in=2'b11;load_unsigned_in=0;load_size_in=2'b11;ahb_resp_in=1;#10
21  iadder_1_to_0_in=2'b00;load_unsigned_in=1;load_size_in=2'b01;ahb_resp_in=1;#10
22
23  iadder_1_to_0_in=2'b00;load_unsigned_in=0;load_size_in=2'b00;ahb_resp_in=1;#10
24  iadder_1_to_0_in=2'b10;load_unsigned_in=1;load_size_in=2'b01;ahb_resp_in=0;#10
25  iadder_1_to_0_in=2'b01;load_unsigned_in=0;load_size_in=2'b11;ahb_resp_in=0;#10
26  iadder_1_to_0_in=2'b11;load_unsigned_in=1;load_size_in=2'b00;ahb_resp_in=1;#10
27  iadder_1_to_0_in=2'b01;load_unsigned_in=0;load_size_in=2'b00;ahb_resp_in=0;#10
28  iadder_1_to_0_in=2'b10;load_unsigned_in=1;load_size_in=2'b11;ahb_resp_in=0;#10
29  iadder_1_to_0_in=2'b11;load_unsigned_in=0;load_size_in=2'b11;ahb_resp_in=0;#10
30  iadder_1_to_0_in=2'b00;load_unsigned_in=1;load_size_in=2'b01;ahb_resp_in=0;#10;
31  end
32  endmodule
33
```

WAVE FORM:



PC MUX:

FUNCTIONALITIES:

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.

Parameter

- **BOOT_ADDRESS**: The address to which the PC is set upon reset.

Internal Logic

1. **Misaligned Instruction Detection:**
    - misaligned_instr_out is set if next_pc is misaligned (i.e., if the least significant bit is 1) and branch_taken_in is high.

2. **PC Increment:**
    - pc_plus_4_out is calculated as the current pc_in plus 4, which is the address of the next sequential instruction.
3. **Next PC Calculation:**
    - next_pc is calculated as:
        - {iaddr_in, 1'b0} if branch_taken_in is high (indicating a branch),
        - pc_plus_4_out otherwise.
4. **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.
5. **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).

RTL CODE:

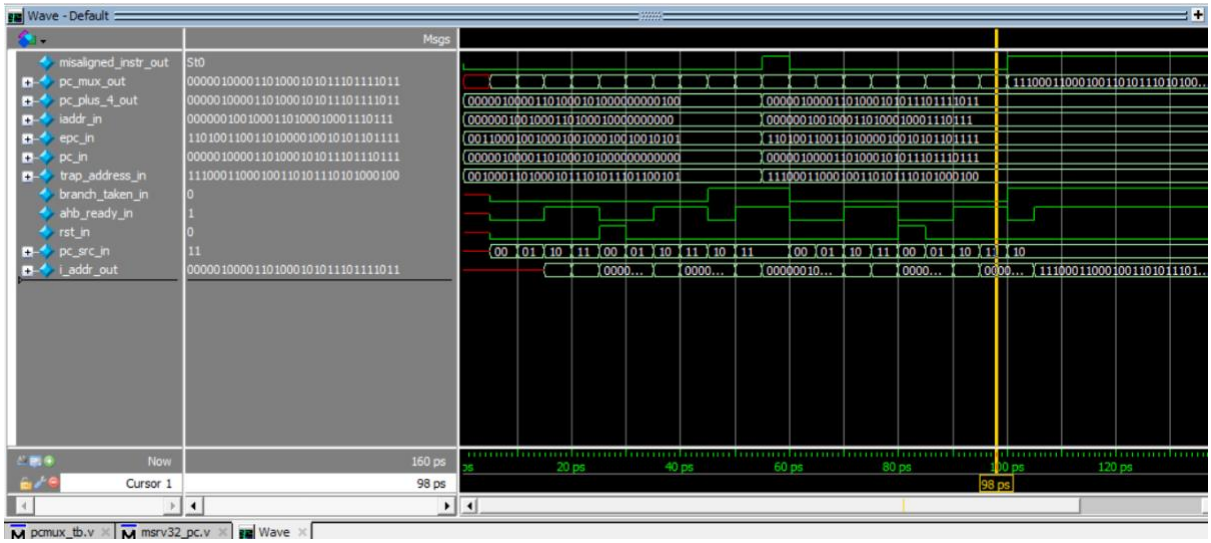
C:/intelFPGA/18.1/msrv32_pc.v - Default

```
Ln#
 1  module msrv32_pc (input              branch_taken_in,rst_in,
 2                    input              ahb_ready_in,
 3                    input       [1:0 ] pc_src_in,
 4                    input       [31:0] epc_in,trap_address_in,pc_in,
 5                    input       [31:1] iaddr_in,
 6                    output      [31:0] pc_plus_4_out,i_addr_out,
 7                    output             misaligned_instr_out,
 8                    output  reg [31:0] pc_mux_out
 9                   );
10     reg [31:0] i_addr;
11     parameter BOOT_ADDRESS = 0;
12
13
14     wire [31:0] next_pc;
15
16     assign misaligned_instr_out=next_pc[1] & branch_taken_in;
17     assign pc_plus_4_out = pc_in + 32'h00000004;
18
19
20     assign next_pc=branch_taken_in ? {iaddr_in,1'b0} : pc_plus_4_out;
21
22
23     assign i_addr_out = i_addr;
24
25
26     always@(*)
27     begin
28        case(pc_src_in)
29           2'b00  : pc_mux_out = BOOT_ADDRESS;
30           2'b01  : pc_mux_out = epc_in;
31           2'b10  : pc_mux_out = trap_address_in;
32           2'b11  : pc_mux_out = next_pc;
33           default: pc_mux_out = next_pc;
34        endcase
35      end
36
37     always @(*)
38     begin
39        if(rst_in)
40        i_addr = BOOT_ADDRESS;
37
38     always @(*)
39     begin
40        if(rst_in)
41        i_addr = BOOT_ADDRESS;
42        else if(ahb_ready_in)
43        i_addr =  pc_mux_out;
44     end
45
46  endmodule
```

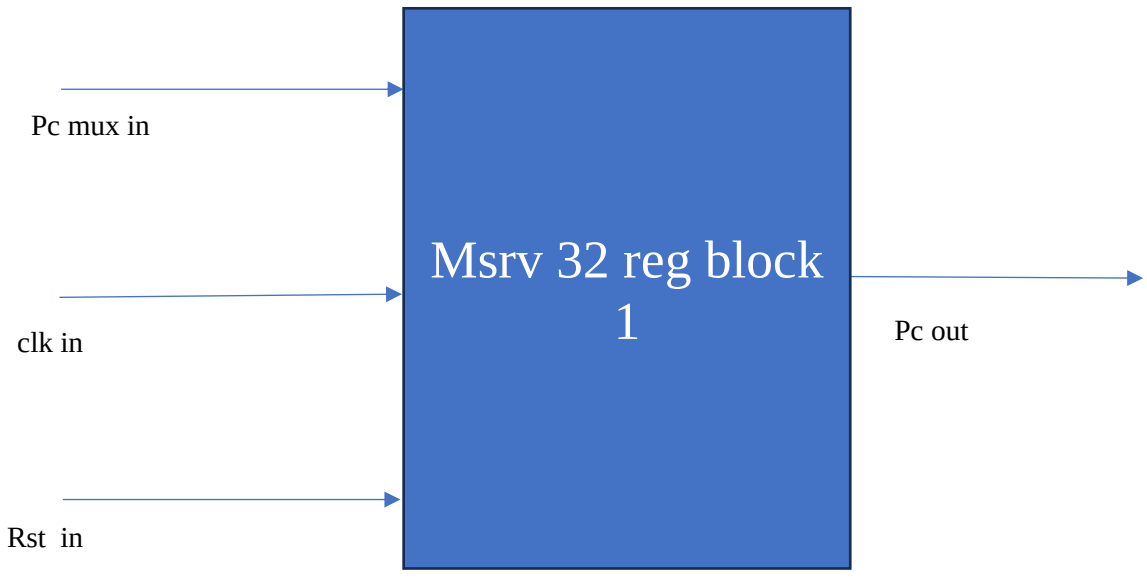
TESTBENCH:

```
Ln#
 1  module pcmux_tb();
 2  wire misaligned_instr_out;
 3  wire [31:0] pc_mux_out,pc_plus_4_out;
 4  reg [31:1] iaddr_in;
 5  reg [31:0] epc_in,pc_in,trap_address_in;
 6  reg branch_taken_in,ahb_ready_in,rst_in;
 7  reg [1:0] pc_src_in;
 8  wire[31:0] i_addr_out;
 9  msrv32_pc DUT(.pc_mux_out(pc_mux_out),.i_addr_out(i_addr_out),.pc_plus_4_out(pc_plus_4_out),.misaligned_instr_out(misaligned_instr_out),
10  .pc_in(pc_in),.iaddr_in(iaddr_in),.epc_in(epc_in),.branch_taken_in(branch_taken_in),.trap_address_in(trap_address_in),.pc_src_in(pc_src_in),
11  .rst_in(rst_in),.ahb_ready_in(ahb_ready_in));
12  initial
13  begin
14  pc_in=31'h43450000;//random number
15  iaddr_in=31'h1234000;//random number
16  epc_in=32'h31224495;
17  trap_address_in=32'h2345D765;
18  #5
19  branch_taken_in=1'b0;ahb_ready_in=1'b0;rst_in=1'b0;pc_src_in=2'b00;#5
20  branch_taken_in=1'b0;ahb_ready_in=1'b0;rst_in=1'b0;pc_src_in=2'b01;#5
21  branch_taken_in=1'b0;ahb_ready_in=1'b1;rst_in=1'b0;pc_src_in=2'b10;#5
22  branch_taken_in=1'b0;ahb_ready_in=1'b1;rst_in=1'b0;pc_src_in=2'b11;#5
23  branch_taken_in=1'b0;ahb_ready_in=1'b0;rst_in=1'b1;pc_src_in=2'b00;#5
24  branch_taken_in=1'b0;ahb_ready_in=1'b0;rst_in=1'b0;pc_src_in=2'b01;#5
25  branch_taken_in=1'b0;ahb_ready_in=1'b1;rst_in=1'b0;pc_src_in=2'b10;#5
26  branch_taken_in=1'b0;ahb_ready_in=1'b1;rst_in=1'b0;pc_src_in=2'b11;#5
27  branch_taken_in=1'b1;ahb_ready_in=1'b0;rst_in=1'b0;pc_src_in=2'b10;#5
28  branch_taken_in=1'b1;ahb_ready_in=1'b1;rst_in=1'b0;pc_src_in=2'b11;#5
29  pc_in=31'h04345777;//random number
30  iaddr_in=31'h01234477;//random number
31  epc_in=32'hD334256F;
32  trap_address_in=32'hE3135D44;
33  #5
34  branch_taken_in=1'b0;ahb_ready_in=1'b0;rst_in=1'b0;pc_src_in=2'b00;#5
35  branch_taken_in=1'b0;ahb_ready_in=1'b0;rst_in=1'b0;pc_src_in=2'b01;#5
36  branch_taken_in=1'b0;ahb_ready_in=1'b1;rst_in=1'b0;pc_src_in=2'b10;#5
37  branch_taken_in=1'b0;ahb_ready_in=1'b1;rst_in=1'b0;pc_src_in=2'b11;#5
38  branch_taken_in=1'b0;ahb_ready_in=1'b0;rst_in=1'b1;pc_src_in=2'b00;#5
39  branch_taken_in=1'b0;ahb_ready_in=1'b0;rst_in=1'b0;pc_src_in=2'b01;#5
40  branch_taken_in=1'b0;ahb_ready_in=1'b1;rst_in=1'b0;pc_src_in=2'b10;#5
41  branch_taken_in=1'b0;ahb_ready_in=1'b1;rst_in=1'b0;pc_src_in=2'b11;#5
42  branch_taken_in=1'b1;ahb_ready_in=1'b0;rst_in=1'b0;pc_src_in=2'b10;#5
43  branch_taken_in=1'b1;ahb_ready_in=1'b1;rst_in=1'b0;pc_src_in=2'b10;#5;
44  end
45  endmodule
```

WAVE FORM:



REG BLOCK 1



FUNCTIONALITIES:

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.

RTL CODE:

C:/intelFPGA/18.1/msrv32_reg_block_1.v - Default *

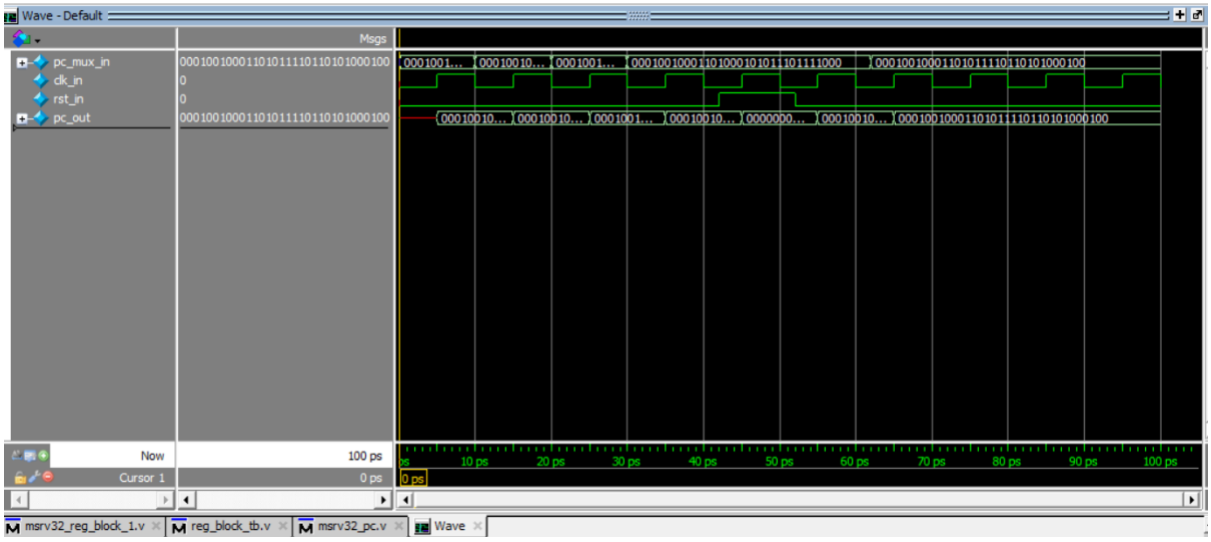
```
Ln#
 1  module msrv32_reg_block_1(input             clk_in,rst_in,
 2                            input      [31:0] pc_mux_in,
 3                            output reg [31:0] pc_out
 4                           );
 5
 6     parameter BOOT_ADDRESS = 0;
 7     always@(posedge clk_in)
 8     begin
 9        if(rst_in)
10           pc_out <= BOOT_ADDRESS;
11        else
12           pc_out <= pc_mux_in;
13     end
14  endmodule
15
```

TESTBENCH:

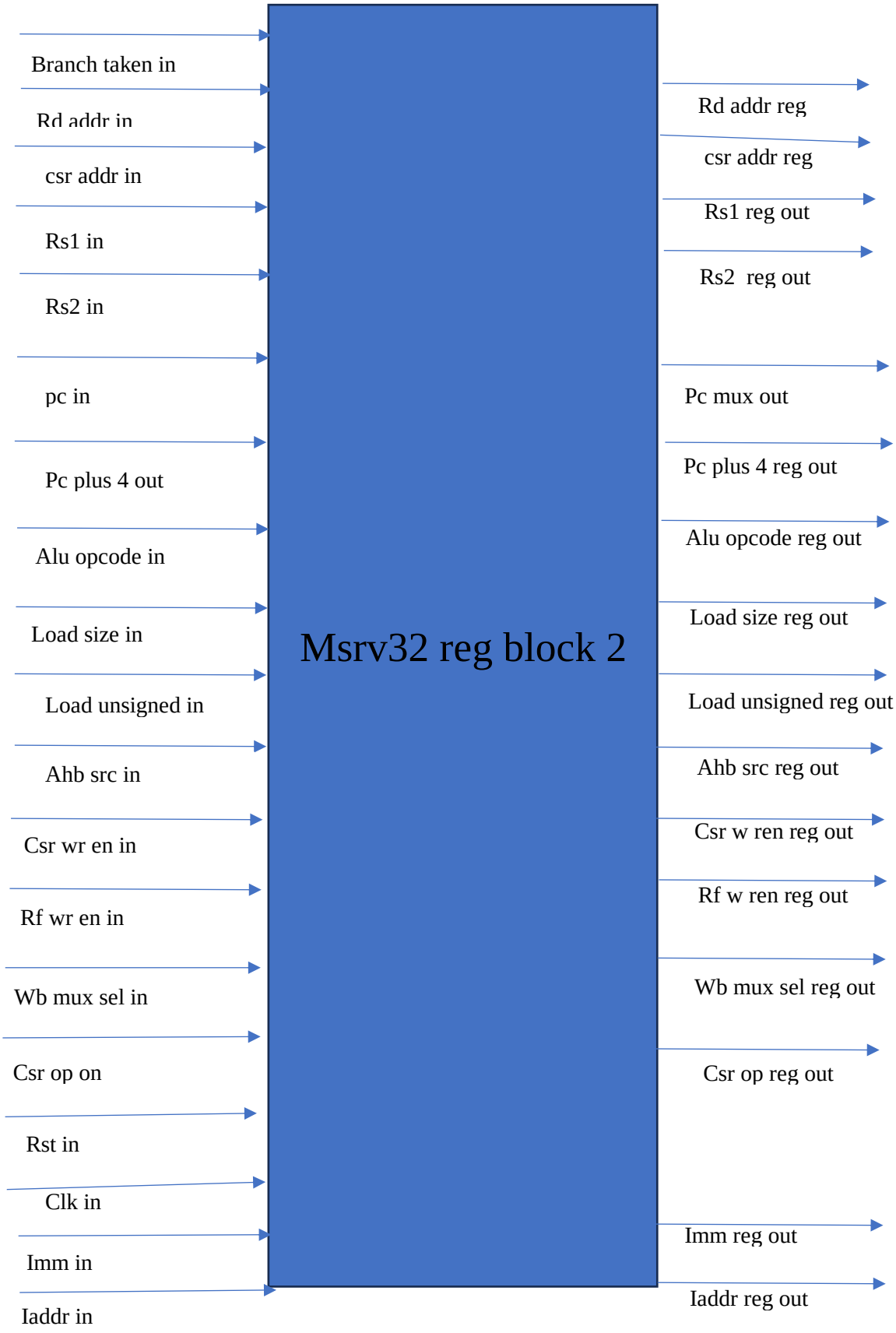
```
Ln#
1   module reg_block_tb();
2   reg [31:0] pc_mux_in;
3   reg clk_in,rst_in;
4   wire [31:0] pc_out;
5
6   msrv32_reg_block_1 DUT(.pc_mux_in(pc_mux_in),.clk_in(clk_in),.rst_in(rst_in),.pc_out(pc_out));
7
8   always #5 clk_in=~clk_in;
9
10  initial
11  begin
12  clk_in=0;
13  pc_mux_in=32'h12345778;rst_in=0;#10
14  pc_mux_in=32'h12869495;rst_in=0;#10
15  pc_mux_in=32'h12345455;rst_in=0;#10
16  pc_mux_in=32'h12345778;rst_in=0;#12
17  rst_in=1;#10
18  rst_in=0;#10
19  pc_mux_in=32'h1235ED44;rst_in=0;#10
20  pc_mux_in=32'h1235ED44;rst_in=0;#10
21  pc_mux_in=32'h1235ED44;rst_in=0;#10
22  pc_mux_in=32'h1235ED44;rst_in=0;#12;
23  end
24  endmodule
25
```

msrv32_reg_block_1.v | reg_block_tb.v | msrv32_pc.v | Wave

WAVE FORM:



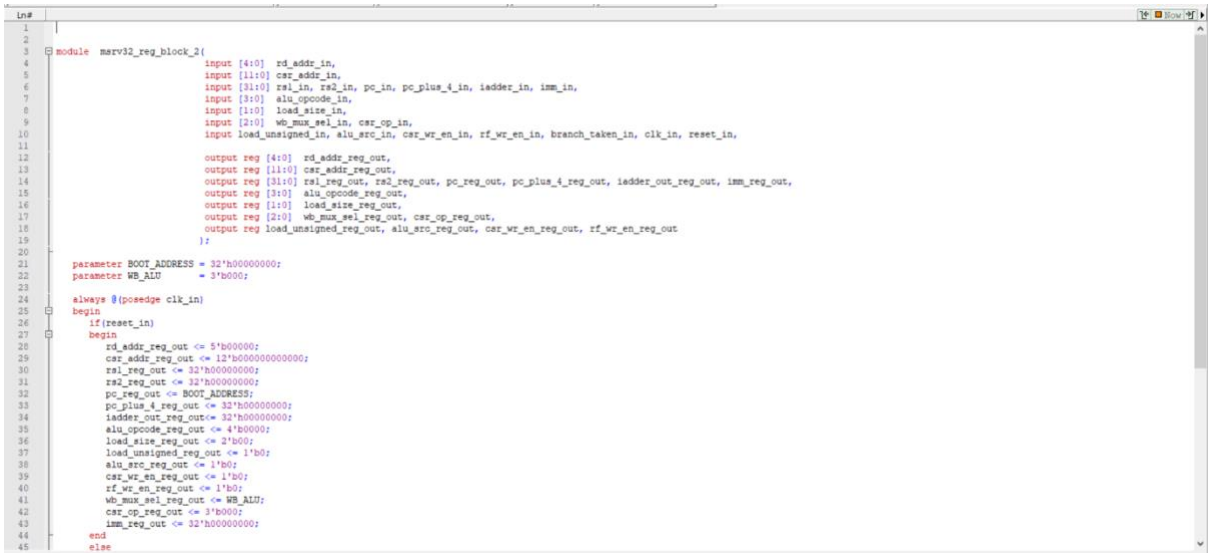
REG BLOCK 2


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
Ahb src in
Csr wr en in
Rf wr 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
Ahb 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

FUNCTIONALITIES:

- 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 handle the reset functionality.
    - If reset_in is high (active reset), all register outputs are reset to specific values (often 0 or default values). This initializes the pipeline registers during processor startup.
    - If reset_in is low (reset inactive), the module latches the various input signals into their respective output registers. This allows the pipeline to operate with the latest values during normal program execution.

RTL CODE:



```
module  msrv32_reg_block_2(
                           input [4:0]  rd_addr_in,
                           input [11:0] csr_addr_in,
                           input [31:0] rs1_in, rs2_in, pc_in, pc_plus_4_in, iadder_in, imm_in,
                           input [3:0]  alu_opcode_in,
                           input [1:0]  load_size_in,
                           input [2:0]  wb_mux_sel_in, csr_op_in,
                           input load_unsigned_in, alu_src_in, csr_wr_en_in, rf_wr_en_in, branch_taken_in, clk_in, reset_in,

                           output reg [4:0]  rd_addr_reg_out,
                           output reg [11:0] csr_addr_reg_out,
                           output reg [31:0] rs1_reg_out, rs2_reg_out, pc_reg_out, pc_plus_4_reg_out, iadder_out_reg_out, imm_reg_out,
                           output reg [3:0]  alu_opcode_reg_out,
                           output reg [1:0]  load_size_reg_out,
                           output reg [2:0]  wb_mux_sel_reg_out, csr_op_reg_out,
                           output reg load_unsigned_reg_out, alu_src_reg_out, csr_wr_en_reg_out, rf_wr_en_reg_out
                           );

   parameter BOOT_ADDRESS = 32'h00000000;
   parameter WB_ALU       = 3'b000;

   always @(posedge clk_in)
   begin
      if(reset_in)
      begin
         rd_addr_reg_out <= 5'b00000;
         csr_addr_reg_out <= 12'b000000000000;
         rs1_reg_out <= 32'h00000000;
         rs2_reg_out <= 32'h00000000;
         pc_reg_out <= BOOT_ADDRESS;
         pc_plus_4_reg_out <= 32'h00000000;
         iadder_out_reg_out<= 32'h00000000;
         alu_opcode_reg_out <= 4'b0000;
         load_size_reg_out <= 2'b00;
         load_unsigned_reg_out <= 1'b0;
         alu_src_reg_out <= 1'b0;
         csr_wr_en_reg_out <= 1'b0;
         rf_wr_en_reg_out <= 1'b0;
         wb_mux_sel_reg_out <= WB_ALU;
         csr_op_reg_out <= 3'b000;
         imm_reg_out <= 32'h00000000;
      end
      else
```

```
    always @(posedge clk_in)
    begin
      if(reset_in)
      begin
         rd_addr_reg_out <= 5'b00000;
         csr_addr_reg_out <= 12'b000000000000;
         rs1_reg_out <= 32'h00000000;
         rs2_reg_out <= 32'h00000000;
         pc_reg_out <= BOOT_ADDRESS;
         pc_plus_4_reg_out <= 32'h00000000;
         iadder_out_reg_out<= 32'h00000000;
         alu_opcode_reg_out <= 4'b0000;
         load_size_reg_out <= 2'b00;
         load_unsigned_reg_out <= 1'b0;
         alu_src_reg_out <= 1'b0;
         csr_wr_en_reg_out <= 1'b0;
         rf_wr_en_reg_out <= 1'b0;
         wb_mux_sel_reg_out <= WB_ALU;
         csr_op_reg_out <= 3'b000;
         imm_reg_out <= 32'h00000000;
      end
      else
      begin
         rd_addr_reg_out <= rd_addr_in;
         csr_addr_reg_out <= csr_addr_in;
         rs1_reg_out <= rs1_in;
         rs2_reg_out <= rs2_in;
         pc_reg_out <= pc_in;
         pc_plus_4_reg_out <= pc_plus_4_in;
         iadder_out_reg_out[31:1] <= iadder_in[31:1];
         iadder_out_reg_out[0] <= branch_taken_in ? 1'b0 : iadder_in[0];
         alu_opcode_reg_out <= alu_opcode_in;
         load_size_reg_out <= load_size_in;
         load_unsigned_reg_out <= load_unsigned_in;
         alu_src_reg_out <= alu_src_in;
         csr_wr_en_reg_out <= csr_wr_en_in;
         rf_wr_en_reg_out <= rf_wr_en_in;
         wb_mux_sel_reg_out<= wb_mux_sel_in;
         csr_op_reg_out <= csr_op_in;
         imm_reg_out <= imm_in;
      end
    end

endmodule
```

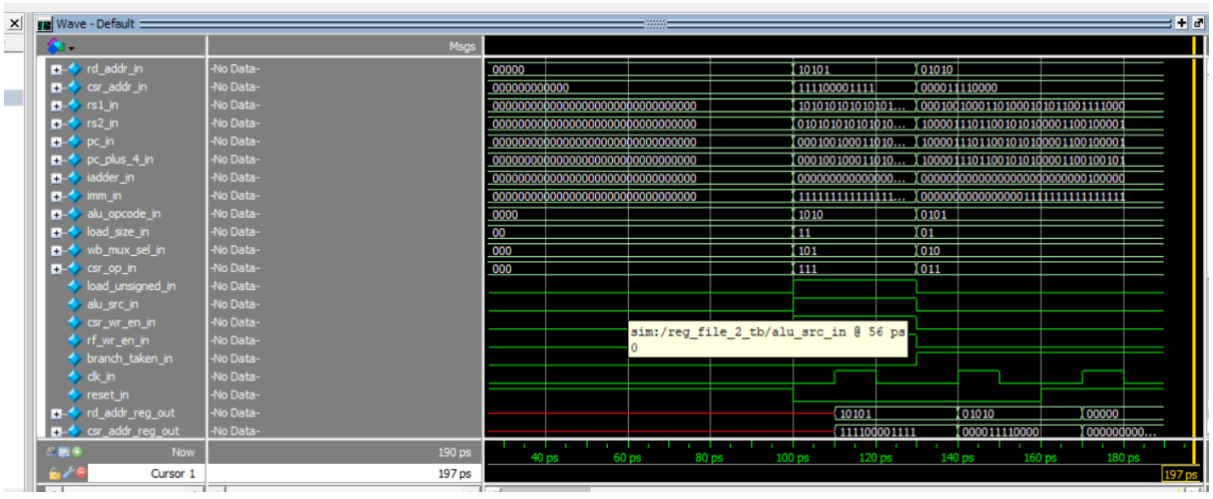
TESTBENCH:

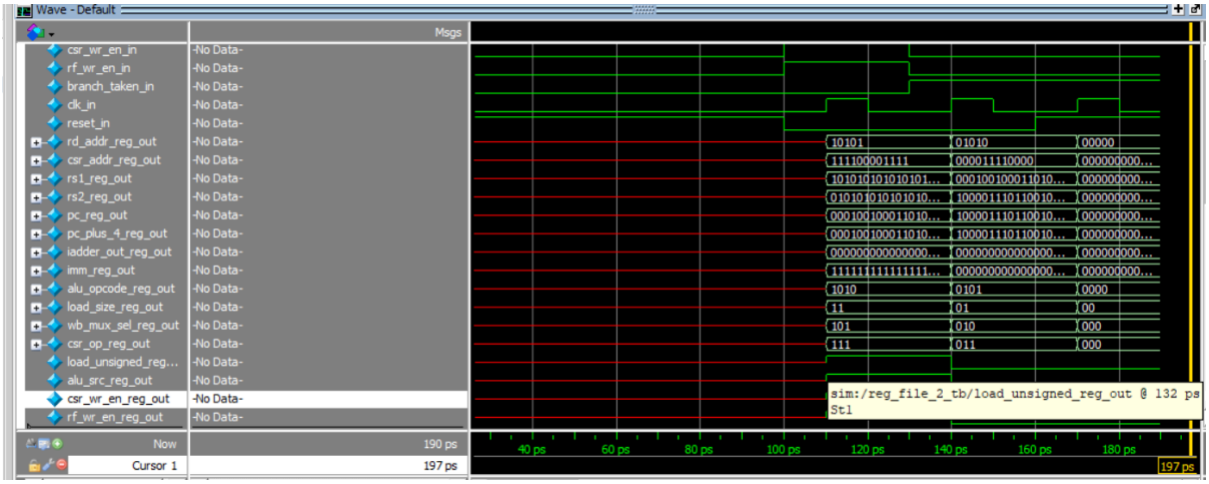
```
module reg_file_2_tb;

  // Inputs
  reg [4:0] rd_addr_in;
  reg [11:0] csr_addr_in;
  reg [31:0] rs1_in, rs2_in, pc_in, pc_plus_4_in, iadder_in, imm_in;
  reg [3:0] alu_opcode_in;
  reg [1:0] load_size_in;
  reg [2:0] wb_mux_sel_in, csr_op_in;
  reg load_unsigned_in, alu_src_in, csr_wr_en_in, rf_wr_en_in, branch_taken_in, clk_in, reset_in;

  // Outputs
  wire [4:0] rd_addr_reg_out;
  wire [11:0] csr_addr_reg_out;
  wire [31:0] rs1_reg_out, rs2_reg_out, pc_reg_out, pc_plus_4_reg_out, iadder_out_reg_out, imm_reg_out;
  wire [3:0] alu_opcode_reg_out;
  wire [1:0] load_size_reg_out;
  wire [2:0] wb_mux_sel_reg_out, csr_op_reg_out;
  wire load_unsigned_reg_out, alu_src_reg_out, csr_wr_en_reg_out, rf_wr_en_reg_out;

  // Instantiate the Unit Under Test (UUT)
  msrv32_reg_block_2 uut (
    .rd_addr_in(rd_addr_in),
    .csr_addr_in(csr_addr_in),
    .rs1_in(rs1_in),
    .rs2_in(rs2_in),
    .pc_in(pc_in),
    .pc_plus_4_in(pc_plus_4_in),
    .iadder_in(iadder_in),
    .imm_in(imm_in),
    .alu_opcode_in(alu_opcode_in),
    .load_size_in(load_size_in),
    .wb_mux_sel_in(wb_mux_sel_in),
    .csr_op_in(csr_op_in),
    .load_unsigned_in(load_unsigned_in),
    .alu_src_in(alu_src_in),
    .csr_wr_en_in(csr_wr_en_in),
    .rf_wr_en_in(rf_wr_en_in),
    .branch_taken_in(branch_taken_in),
    .clk_in(clk_in),
    .reset_in(reset_in),
    .rd_addr_reg_out(rd_addr_reg_out),
    .csr_addr_reg_out(csr_addr_reg_out),
    .rs1_reg_out(rs1_reg_out),
    .rs2_reg_out(rs2_reg_out),
```

```
// Apply a clock pulse
clk_in = 1;
#10;
clk_in = 0;
#10;

// Test Vector 2: Change some values
rd_addr_in = 5'b01010;
csr_addr_in = 12'b000011110000;
rs1_in = 32'h12345678;
rs2_in = 32'h87654321;
pc_in = 32'h87654321;
pc_plus_4_in = 32'h87654325;
iadder_in = 32'h00000020;
imm_in = 32'h0000FFFF;
alu_opcode_in = 4'b0101;
load_size_in = 2'b01;
wb_mux_sel_in = 3'b010;
csr_op_in = 3'b011;
load_unsigned_in = 0;
alu_src_in = 0;
csr_wr_en_in = 0;
rf_wr_en_in = 0;
branch_taken_in = 1;
#10;

// Apply a clock pulse
clk_in = 1;
#10;
clk_in = 0;
#10;

// Test Vector 3: Reset the block
reset_in = 1;
#10;
```

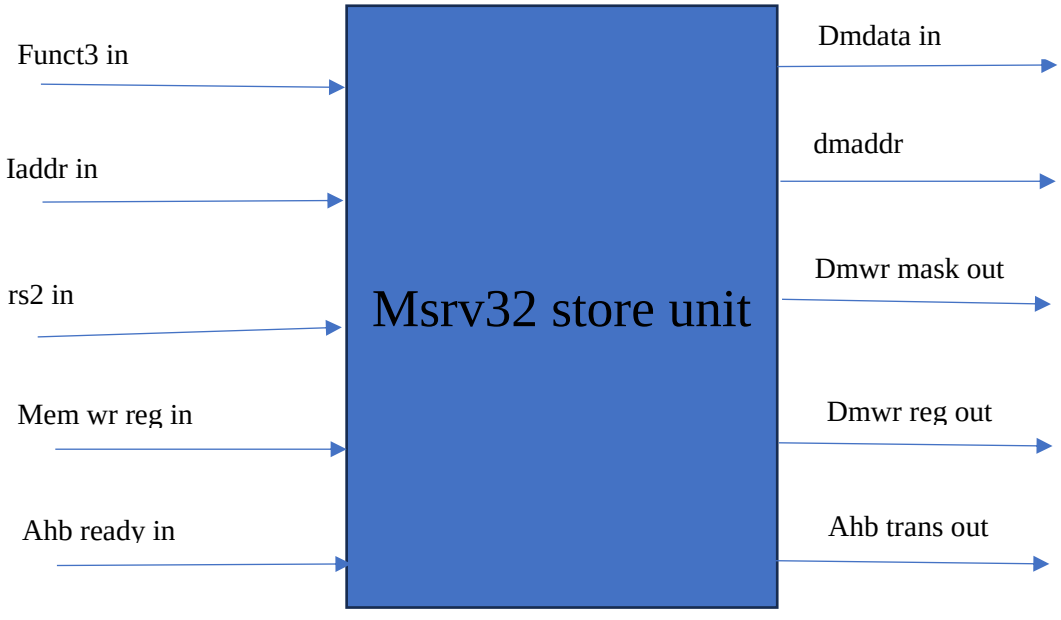
```
  rs2_in = 32'h87654321;
  pc_in = 32'h87654321;
  pc_plus_4_in = 32'h87654325;
  iadder_in = 32'h00000020;
  imm_in = 32'h0000FFFF;
  alu_opcode_in = 4'b0101;
  load_size_in = 2'b01;
  wb_mux_sel_in = 3'b010;
  csr_op_in = 3'b011;
  load_unsigned_in = 0;
  alu_src_in = 0;
  csr_wr_en_in = 0;
  rf_wr_en_in = 0;
  branch_taken_in = 1;
  #10;

  // Apply a clock pulse
  clk_in = 1;
  #10;
  clk_in = 0;
  #10;

  // Test Vector 3: Reset the block
  reset_in = 1;
  #10;

  // Apply a clock pulse
  clk_in = 1;
  #10;
  clk_in = 0;
  #10;

  // Finish the test
  $finish;
end

initial begin
  $monitor("At time %t, rd_addr_reg_out = %b, csr_addr_reg_out = %b, rs1_reg_out = %h, rs2_reg_out = %h, pc_reg_out = %h, pc_plus_4_reg_out = %h, iadder_out_reg_out = %h, imm_reg_out = %h, alu_opcode_reg_o
           $time, rd_addr_reg_out, csr_addr_reg_out, rs1_reg_out, rs2_reg_out, pc_reg_out, pc_plus_4_reg_out, iadder_out_reg_out, imm_reg_out, alu_opcode_reg_out, load_size_reg_out, wb_mux_sel_reg_out, csr
end

endmodule
```

WAVE FORM:





STORAGE UNIT



FUNCTIONALITIES:

1. **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.

2. **Data Formatting (always blocks):**
   - Two separate always@(*) blocks handle data formatting based on funct3_in and iadder_in[1:0]:
     - byte_dout: This register stores the rs2_in value shifted and padded with zeros depending on the byte address within the word (e.g., for byte address 0, only the lower 8 bits of rs2_in are used).
     - halfword_dout: This register stores the rs2_in value shifted and padded with zeros depending on the halfword address within the word (e.g., for halfword address 0, only the lower 16 bits of rs2_in are used).
   - Similar always@(*) blocks define byte_wr_mask and halfword_wr_mask which are write masks used to enable writes to specific bytes within a word during byte or halfword stores.
3. **Output Selection and Bus Signals (always blocks):**
   - An always@(*) block controls the data_out, ahb_htrans_out, and wr_mask_out based on funct3_in, mem_wr_req_in, and ahb_ready_in:
     - 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. It sets ahb_htrans_out to indicate a non-sequential (scattered) access and sets wr_mask_out to the corresponding byte or halfword write mask.
     - If ahb_ready_in is low (bus not ready), the block sets ahb_htrans_out to 0 (no transfer).
4. **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.

RTL CODE:

C:/intelFPGA/18.1/msrv32_store_unit.v - Default

```
Ln#
 1  module msrv32_store_unit(input      [1:0 ] funct3_in,
 2                           input      [31:0] iadder_in,rs2_in,
 3                           input             mem_wr_req_in,ahb_ready_in,
 4                           output     [31:0] d_addr_out,
 5                           output reg [31:0] data_out,
 6                           output reg [3 :0] wr_mask_out,
 7                           output reg [1:0]  ahb_htrans_out,
 8                           output            wr_req_out
 9  );
10
11
12     reg [31:0] byte_dout,halfword_dout;
13     reg [3:0 ] byte_wr_mask,halfword_wr_mask;
14     reg [31:0] d_addr=0;
15
16     assign d_addr_out= {iadder_in[31:2],2'b00};
17
18     assign wr_req_out=mem_wr_req_in;
19
20     always@(*)
21     begin
22        case(iadder_in[1:0])
23           2'b00   : byte_dout = {8'b0,8'b0,8'b0,rs2_in[7:0]};
24           2'b01   : byte_dout = {8'b0,8'b0,rs2_in[7:0],8'b0};
25           2'b10   : byte_dout = {8'b0,rs2_in[7:0],8'b0,8'b0};
26           2'b11   : byte_dout = {rs2_in[7:0],8'b0,8'b0,8'b0};
27           default : byte_dout = 32'b0;
28        endcase
29     end
30
31
32     always@(*)
33     begin
34        case(iadder_in[1])
35           1'b0    : halfword_dout = {16'b0,rs2_in[15:0]};
36           1'b1    : halfword_dout = {rs2_in[15:0],16'b0};
37           default : halfword_dout = 32'b0;
38        endcase
39     end
40
```

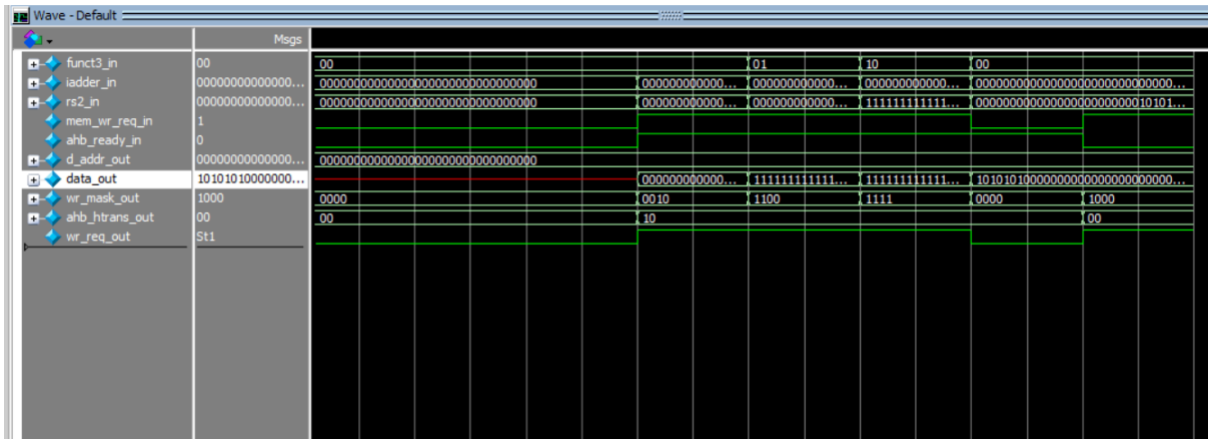
```
40
41     always@(*)
42     begin
43       if(ahb_ready_in)
44        begin
45           case(funct3_in)
46              2'b00   : data_out = byte_dout;
47              2'b01   : data_out = halfword_dout;
48              default : data_out = rs2_in;
49           endcase
50           ahb_htrans_out = 2'b10;
51        end
52        else
53           ahb_htrans_out = 2'b00;
54     end
55
56
57     always@(*)
58     begin
59        case(funct3_in)
60           2'b00   : wr_mask_out = byte_wr_mask;
61           2'b01   : wr_mask_out = halfword_wr_mask;
62           default : wr_mask_out = {4{mem_wr_req_in}};
63        endcase
64     end
65
66
67     always@(*)
68     begin
69        case(iadder_in[1:0])
70           2'b00   : byte_wr_mask = {3'b0,mem_wr_req_in};
71           2'b01   : byte_wr_mask = {2'b0,mem_wr_req_in,1'b0};
72           2'b10   : byte_wr_mask = {1'b0,mem_wr_req_in,2'b0};
73           2'b11   : byte_wr_mask = {mem_wr_req_in,3'b0};
74           default : byte_wr_mask = {4{mem_wr_req_in}};
75        endcase
76     end
77
78     always@(*)
79     begin
```

```
        case(funct3_in)
           2'b00   : data_out = byte_dout;
           2'b01   : data_out = halfword_dout;
           default : data_out = rs2_in;
        endcase
        ahb_htrans_out = 2'b10;
     end
     else
        ahb_htrans_out = 2'b00;
  end


  always@(*)
  begin
     case(funct3_in)
        2'b00   : wr_mask_out = byte_wr_mask;
        2'b01   : wr_mask_out = halfword_wr_mask;
        default : wr_mask_out = {4{mem_wr_req_in}};
     endcase
  end


  always@(*)
  begin
     case(iadder_in[1:0])
        2'b00   : byte_wr_mask = {3'b0,mem_wr_req_in};
        2'b01   : byte_wr_mask = {2'b0,mem_wr_req_in,1'b0};
        2'b10   : byte_wr_mask = {1'b0,mem_wr_req_in,2'b0};
        2'b11   : byte_wr_mask = {mem_wr_req_in,3'b0};
        default : byte_wr_mask = {4{mem_wr_req_in}};
     endcase
  end

  always@(*)
  begin
     case(iadder_in[1])
        1'b0    : halfword_wr_mask = {2'b0,{2{mem_wr_req_in}}};
        1'b1    : halfword_wr_mask = {{2{mem_wr_req_in}},2'b0};
        default : halfword_wr_mask = {4{mem_wr_req_in}};
     endcase
end
endmodule
```

TESTBENCH :

```
Ln#
 1  module store_unit_tb;
 2
 3     // Inputs
 4     reg [1:0] funct3_in;
 5     reg [31:0] iadder_in;
 6     reg [31:0] rs2_in;
 7     reg mem_wr_req_in;
 8     reg ahb_ready_in;
 9
10     // Outputs
11     wire [31:0] d_addr_out;
12     wire [31:0] data_out;
13     wire [3:0] wr_mask_out;
14     wire [1:0] ahb_htrans_out;
15     wire wr_req_out;
16
17     // Instantiate the Unit Under Test (UUT)
18     msrv32_store_unit uut (
19       .funct3_in(funct3_in),
20       .iadder_in(iadder_in),
21       .rs2_in(rs2_in),
22       .mem_wr_req_in(mem_wr_req_in),
23       .ahb_ready_in(ahb_ready_in),
24       .d_addr_out(d_addr_out),
25       .data_out(data_out),
26       .wr_mask_out(wr_mask_out),
27       .ahb_htrans_out(ahb_htrans_out),
28       .wr_req_out(wr_req_out)
29     );
30
31     initial begin
32       // Initialize Inputs
33       funct3_in = 0;
34       iadder_in = 0;
35       rs2_in = 0;
36       mem_wr_req_in = 0;
37       ahb_ready_in = 0;
38
39       // Wait for global reset to finish
40       #100;
41
42       // Test case 1: Byte write, unaligned address
43       funct3_in = 2'b00; // Byte
44       iadder_in = 32'h00000001; // Address with offset 1
45       rs2_in = 32'h000000FF; // Data to write
```

WAVE FORM:



WB MUX

FUNCTIONALITIES:

1. **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).
    - The output of this mux (alu_2nd_src_mux_out) is not directly an output of the module but might be used internally within the processor.
2. **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.
    - If no case is matched, the default behavior assigns the ALU output (alu_result_in) to the write-back mux output.

RTL CODE:

```
Ln#
 1  module msrv32_wb_mux_sel_unit(input [2:0] wb_mux_sel_reg_in,
 2                                input [31:0] alu_result_in,lu_output_in,imm_reg_in,
 3                                iadder_out_reg_in,csr_data_in,pc_plus_4_reg_in,rs2_reg_in,
 4                                input alu_source_reg_in,
 5                                output reg [31:0] wb_mux_out,
 6                                output [31:0] alu_2nd_src_mux_out);
 7
 8     parameter WB_ALU = 3'b000,WB_LU = 3'b001, WB_IMM = 3'b010,WB_IADDER_OUT = 3'b011,
 9               WB_CSR = 3'b100, WB_PC_PLUS = 3'b101;
10     assign alu_2nd_src_mux_out = alu_source_reg_in ? rs2_reg_in : imm_reg_in;
11      always@*
12      begin
13         case(wb_mux_sel_reg_in)
14            WB_ALU        : wb_mux_out = alu_result_in;
15            WB_LU         : wb_mux_out = lu_output_in;
16            WB_IMM        : wb_mux_out = imm_reg_in;
17            WB_IADDER_OUT : wb_mux_out = iadder_out_reg_in;
18            WB_CSR        : wb_mux_out = csr_data_in;
19            WB_PC_PLUS    : wb_mux_out = pc_plus_4_reg_in;
20            default       : wb_mux_out = alu_result_in;
21          endcase
22      end
23
24  endmodule
25
```

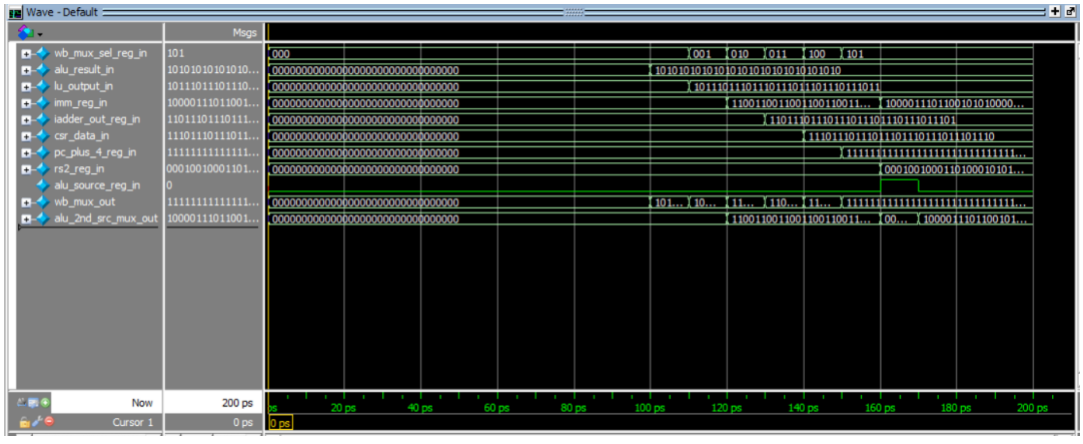
TESTBENCH:

```
Ln#
 1  module tb_msrv32_wb_mux_sel_unit;
 2
 3    // Inputs
 4    reg [2:0] wb_mux_sel_reg_in;
 5    reg [31:0] alu_result_in;
 6    reg [31:0] lu_output_in;
 7    reg [31:0] imm_reg_in;
 8    reg [31:0] iadder_out_reg_in;
 9    reg [31:0] csr_data_in;
10    reg [31:0] pc_plus_4_reg_in;
11    reg [31:0] rs2_reg_in;
12    reg alu_source_reg_in;
13
14    // Outputs
15    wire [31:0] wb_mux_out;
16    wire [31:0] alu_2nd_src_mux_out;
17
18    // Instantiate the Unit Under Test (UUT)
19    msrv32_wb_mux_sel_unit uut (
20      .wb_mux_sel_reg_in(wb_mux_sel_reg_in),
21      .alu_result_in(alu_result_in),
22      .lu_output_in(lu_output_in),
23      .imm_reg_in(imm_reg_in),
24      .iadder_out_reg_in(iadder_out_reg_in),
25      .csr_data_in(csr_data_in),
26      .pc_plus_4_reg_in(pc_plus_4_reg_in),
27      .rs2_reg_in(rs2_reg_in),
28      .alu_source_reg_in(alu_source_reg_in),
29      .wb_mux_out(wb_mux_out),
30      .alu_2nd_src_mux_out(alu_2nd_src_mux_out)
31    );
32
33    initial begin
34      // Initialize Inputs
35      wb_mux_sel_reg_in = 0;
36      alu_result_in = 0;
37      lu_output_in = 0;
38      imm_reg_in = 0;
39      iadder_out_reg_in = 0;
40      csr_data_in = 0;
41      pc_plus_4_reg_in = 0;
42      rs2_reg_in = 0;
43      alu_source_reg_in = 0;
44
```

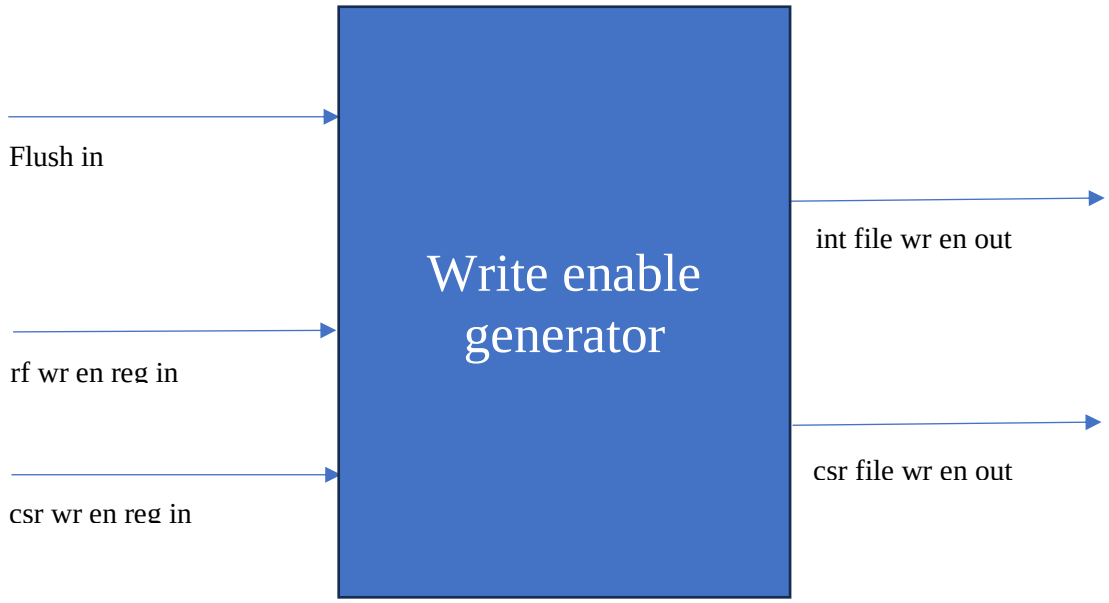
```
Ln#
30      .alu_2nd_src_mux_out(alu_2nd_src_mux_out)
31    );
32
33    initial begin
34      // Initialize Inputs
35      wb_mux_sel_reg_in = 0;
36      alu_result_in = 0;
37      lu_output_in = 0;
38      imm_reg_in = 0;
39      iadder_out_reg_in = 0;
40      csr_data_in = 0;
41      pc_plus_4_reg_in = 0;
42      rs2_reg_in = 0;
43      alu_source_reg_in = 0;
44
45      // Wait for global reset to finish
46      #100;
47
48      // Test case 1: ALU result selection
49      wb_mux_sel_reg_in = 3'b000;
50      alu_result_in = 32'hAAAAAAAA;
51      #10;
52
53      // Test case 2: LU output selection
54      wb_mux_sel_reg_in = 3'b001;
55      lu_output_in = 32'hBBBBBBBB;
56      #10;
57
58      // Test case 3: Immediate value selection
59      wb_mux_sel_reg_in = 3'b010;
60      imm_reg_in = 32'hCCCCCCCC;
61      #10;
62
63      // Test case 4: IADDER output selection
64      wb_mux_sel_reg_in = 3'b011;
65      iadder_out_reg_in = 32'hDDDDDDDD;
66      #10;
67
68      // Test case 5: CSR data selection
69      wb_mux_sel_reg_in = 3'b100;
70      csr_data_in = 32'hEEEEEEEE;
71      #10;
72
73      // Test case 6: PC plus 4 selection
74      wb_mux_sel_reg_in = 3'b101;
```

```
    // Test case 3: Immediate value selection
    wb_mux_sel_reg_in = 3'b010;
    imm_reg_in = 32'hCCCCCCCC;
    #10;

    // Test case 4: IADDER output selection
    wb_mux_sel_reg_in = 3'b011;
    iadder_out_reg_in = 32'hDDDDDDDD;
    #10;

    // Test case 5: CSR data selection
    wb_mux_sel_reg_in = 3'b100;
    csr_data_in = 32'hEEEEEEEE;
    #10;

    // Test case 6: PC plus 4 selection
    wb_mux_sel_reg_in = 3'b101;
    pc_plus_4_reg_in = 32'hFFFFFFFF;
    #10;

    // Test case 7: ALU second source MUX selection - use RS2
    alu_source_reg_in = 1;
    rs2_reg_in = 32'h12345678;
    imm_reg_in = 32'h87654321;
    #10;

    // Test case 8: ALU second source MUX selection - use immediate
    alu_source_reg_in = 0;
    rs2_reg_in = 32'h12345678;
    imm_reg_in = 32'h87654321;
    #10;

    // Finish the test
    $finish;
  end

  initial begin
    $monitor("At time %t, wb_mux_sel_reg_in = %b, alu_result_in = %h, lu_output_in = %h, imm_reg_in = %h, iadder_out_reg_in = %h, csr_data_in = %h, pc_plus_4_reg_in = %h, rs2_reg_in = %h, alu_source_reg_in =
             $time, wb_mux_sel_reg_in, alu_result_in, lu_output_in, imm_reg_in, iadder_out_reg_in, csr_data_in, pc_plus_4_reg_in, rs2_reg_in, alu_source_reg_in, wb_mux_out, alu_2nd_src_mux_out);
  end

endmodule
```

WAVE FORM:



WRITE ENABLE GENERATOR



FUNCTIONALITIES:

Inputs:

- flush_in: Flush signal (active high).
- rf_wr_en_reg_in: Register file write enable signal (from pipeline).
- csr_wr_en_reg_in: CSR file write enable signal (from pipeline).

Outputs:

- wr_en_integer_file_out: Write enable signal for the integer register file (output).
- wr_en_csr_file_out: Write enable signal for the CSR file (output).

Functionality:

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.

RTL CODE:

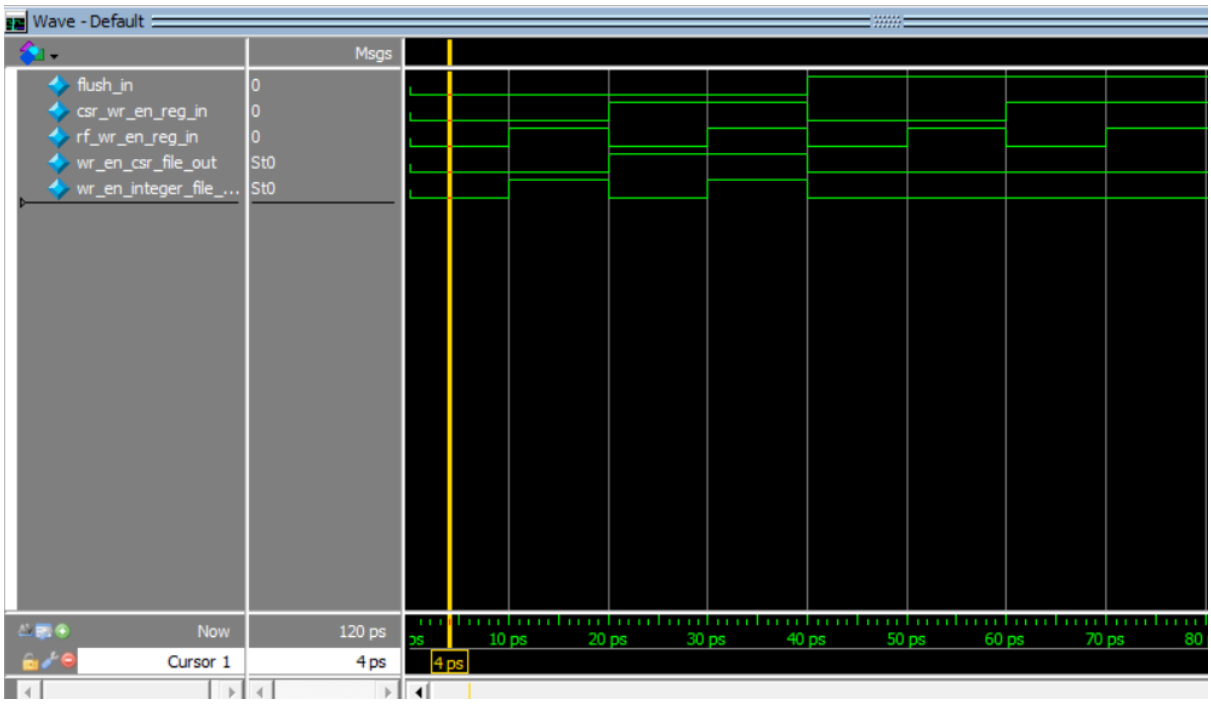
C:/intelFPGA/18.1/msrv32_wr_en_generator.v - Default *

```
Ln#
1  module msrv32_wr_en_generator(input flush_in,rf_wr_en_reg_in,csr_wr_en_reg_in,
2                                output wr_en_integer_file_out,wr_en_csr_file_out
3                               );
4     assign wr_en_integer_file_out = flush_in ? 1'b0 : rf_wr_en_reg_in;
5     assign wr_en_csr_file_out = flush_in ? 1'b0 : csr_wr_en_reg_in;
6
7  endmodule
8
```

TESTBENCH:

```
Ln#
1  module wr_en_generator_tb();
2  reg flush_in,csr_wr_en_reg_in,rf_wr_en_reg_in;
3  wire wr_en_csr_file_out,wr_en_integer_file_out;
4
5  msrv32_wr_en_generator DUT(.flush_in(flush_in),.csr_wr_en_reg_in(csr_wr_en_reg_in),.rf_wr_en_reg_in(rf_wr_en_reg_in),.wr_en_csr_file_out(wr_en_csr_
6
7
8
9  initial
10 begin
11 flush_in=1'b0;csr_wr_en_reg_in=1'b0;rf_wr_en_reg_in=1'b0;#10
12 flush_in=1'b0;csr_wr_en_reg_in=1'b0;rf_wr_en_reg_in=1'b1;#10
13 flush_in=1'b0;csr_wr_en_reg_in=1'b1;rf_wr_en_reg_in=1'b0;#10
14 flush_in=1'b0;csr_wr_en_reg_in=1'b1;rf_wr_en_reg_in=1'b1;#10
15 flush_in=1'b1;csr_wr_en_reg_in=1'b0;rf_wr_en_reg_in=1'b0;#10
16 flush_in=1'b1;csr_wr_en_reg_in=1'b0;rf_wr_en_reg_in=1'b1;#10
17 flush_in=1'b1;csr_wr_en_reg_in=1'b1;rf_wr_en_reg_in=1'b0;#10
18 flush_in=1'b1;csr_wr_en_reg_in=1'b1;rf_wr_en_reg_in=1'b1;#10;
19 end
20 endmodule
21
22
```

WAVE FORM:


Wave - Default
Msgs
flush_in
csr_wr_en_reg_in
rf_wr_en_reg_in
wr_en_csr_file_out
wr_en_integer_file_...
0
0
0
St0
St0
Now
120 ps
Cursor 1
4 ps
4 ps
10 ps
20 ps
30 ps
40 ps
50 ps
60 ps
70 ps