Project Report –1

VLSI Design Lab

Title: Control Status Registers (CSR)

Submitted By:

Manish Ranjan(24M1176)

Anubhav Bhura(24M1219)

Razul N Haris(24M1182)

Supervisor:

Prof. Laxmeesha Somappa

February 21,2025

CSR

Basic Theory of Control and Status Registers (CSRs) in Processor Architectures

Control and Status Registers (CSRs) are critical components in modern processor designs, particularly within the RISC-V architecture. They serve as a central mechanism to control processor operations, manage state, and facilitate exception and interrupt handling. CSRs encapsulate both configuration and status information, enabling the system to respond dynamically to various events and maintain proper execution flow.

At the heart of CSR functionality lies a set of key registers that monitor and control the processor's behavior. Among these are the privilege register, status register, trap vector register, exception program counter, cause register, interrupt pending and enable registers, and additional auxiliary registers.

1. **Privilege and Status Management**

The privilege register maintains the current operating level of the processor, ensuring that only authorized code executes critical operations. The status register holds vital flags such as interrupt enable bits and other state indicators that reflect the current condition of the processor. When an exceptional event occurs, portions of the status register are updated to record the active state, allowing for proper restoration after the event is handled.

2. **Exception Handling and Trap Processing**

Exception handling is a cornerstone of CSR utility. When the processor detects an exception—be it due to an environment call, a breakpoint, or misaligned instruction fetch—it records the event's cause in a dedicated register. The exception program counter saves the address of the instruction that triggered the exception, ensuring that execution can resume correctly after the exception is serviced. The trap vector register provides the base address for the exception handler, which, when combined with the cause information, directs the processor to the correct service routine.

3. **Interrupt Management**

Interrupts, whether triggered externally or by internal conditions, are managed through dedicated registers that track both pending and enabled interrupt signals. By comparing these registers, the system determines when to service an interrupt. This controlled mechanism prevents spurious or unauthorized interrupts from disrupting normal execution, thereby maintaining system stability.

4. **CSR Access Operations**

Instructions specifically designed for CSR access allow for reading and modifying these registers. Operations may include writing a new value, setting specific bits, or clearing bits within the register. A masking mechanism ensures that only the writable parts of a register are modified, protecting reserved bits from unintended changes. Furthermore, access control enforced by privilege levels prevents unauthorized modifications, and any attempt to alter a CSR without sufficient rights results in an exception.

5. **System Robustness and Context Switching**

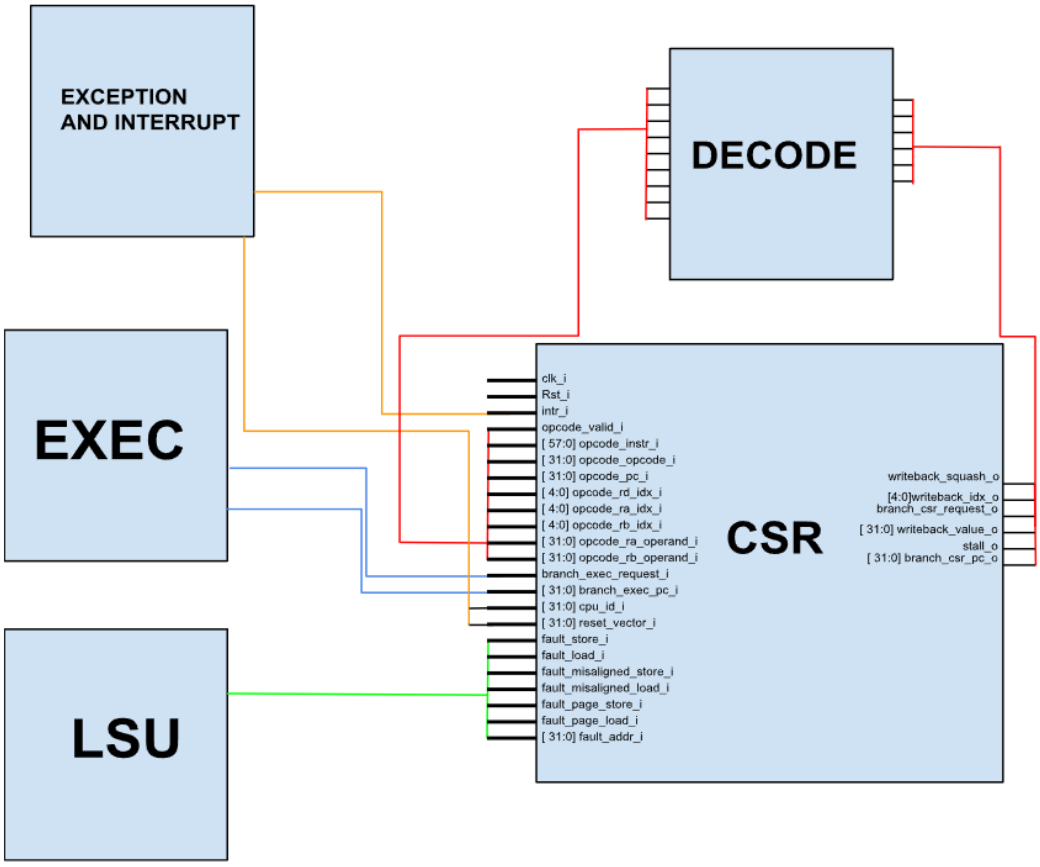
In scenarios where an exception or interrupt is triggered, the processor follows a well-defined sequence: the cause is recorded, the current state is saved (including privilege and status information), and the control is transferred to an appropriate handler. Once the exception is handled, a return mechanism restores the saved state, allowing normal execution to continue. This sequence not only supports reliable error recovery but also facilitates efficient context switching between different execution states.

Implementation

I. Inputs and Output Ports :

```
input  wire         clk_i,
input  wire         rst_i,
input  wire         intr_i,       //External interrupt input from Exception and Interrupt
input  wire         opcode_valid_i,  //Indicates if the current opcode is valid from Decode
input  wire [57:0]  opcode_instr_i,  //instruction input from Decode
input  wire [31:0]  opcode_opcode_i,  //opcode input from Decode
input  wire [31:0]  opcode_pc_i,    //Program counter for the current instruction from Decode
input  wire [4:0]   opcode_rd_idx_i,  // Destination register index from Decode
input  wire [4:0]   opcode_ra_idx_i,  // First source register index from Decode
input  wire [4:0]   opcode_rb_idx_i,  //Second source register index from Decode
input  wire [31:0]  opcode_ra_operand_i,  //Value of the first source register from Decode
input  wire [31:0]  opcode_rb_operand_i,   //Value of the second source register from Decode
input  wire         branch_exec_request_i,   // Indicates a branch execution request from Exec
input  wire [31:0]  branch_exec_pc_i,       // Target PC for the branch execution from Exec
input  wire [31:0]  cpu_id_i,               //CPU identification number from Exception and Interrupt
input  wire [31:0]  reset_vector_i,         //Reset vector address from Exception and Interrupt
input  wire         fault_store_i,          //Indicates a store fault from LSU
input  wire         fault_load_i,           // Indicates a load fault from LSU
input  wire         fault_misaligned_store_i, //Indicates a misaligned store fault from LSU
input  wire         fault_misaligned_load_i,  //Indicates a misaligned load fault from LSU
input  wire         fault_page_store_i,       //Indicates a page fault during store from LSU
input  wire         fault_page_load_i,        //Indicates a page fault during load from LSU
input  wire [31:0]  fault_addr_i,              // Address where a fault occurred from LSU

output reg  [4:0]   writeback_idx_o,            //Index of the register to be written back to Decode
output reg          writeback_squash_o,         //Indicates if the writeback should be squashed to Decode
output reg  [31:0]  writeback_value_o,          // Value to be written back to Decode
output reg          stall_o,                    //Indicates if the pipeline should stall to Decode
output reg          branch_csr_request_o,       //Indicates a branch request due to CSR operation to Decode
output reg  [31:0]  branch_csr_pc_o             // Target PC for the CSR-induced branch to Decode
```

II. RISC-V CSR Module Integration :



The CSR module is a central hub that exchanges control, status, and exception-related information between various parts of your processor. Here's a breakdown of its communication:

1. **Interaction with the Decode Unit:**
    a. **Instruction and Operand Data:**
    The module receives opcode-related signals from the decode stage (e.g., `opcode_valid_i`, `opcode_instr_i`, `opcode_opcode_i`, `opcode_pc_i`) along with source (`opcode_ra_operand_i`, `opcode_rb_operand_i`) and destination register indices (`opcode_rd_idx_i`, `opcode_ra_idx_i`, `opcode_rb_idx_i`). These inputs tell the CSR module what operation to perform and provide the data it needs to work on.
    b. **Writeback to Decode:**
    After processing, the CSR module sends the result back to the decode unit using outputs like `writeback_idx_o` (indicating which register to update),

`writeback_value_o` (the computed value), and `writeback_squash_o` (to indicate if the writeback should be ignored due to, for example, branch mispredictions or exceptions).

2. **Connection with the Execution Unit:**
    a. **Branch Control:**
    The execution unit signals branch-related events via `branch_exec_request_i` and provides the target program counter (`branch_exec_pc_i`). If a CSR operation necessitates a branch (for instance, due to an exception or interrupt), the CSR module issues its own branch request (`branch_csr_request_o`) along with the target address (`branch_csr_pc_o`) to help steer the control flow accordingly.
3. **Interface with Exception and Interrupt Logic:**
    a. **Interrupt and Reset Handling:**
    The external interrupt signal (`intr_i`) comes from the exception/interrupt management module, letting the CSR module know when an external event needs attention. Additionally, signals such as `cpu_id_i` and `reset_vector_i` provide identification and initialization information to properly manage interrupts and system resets.
4. **Feedback from the Load-Store Unit (LSU):**
    a. **Fault Detection:**
    The CSR module monitors several fault signals from the LSU: `fault_store_i`, `fault_load_i`, `fault_misaligned_store_i`, `fault_misaligned_load_i`, `fault_page_store_i`, and `fault_page_load_i`. The `fault_addr_i` indicates the memory address where a fault occurred. These inputs help the CSR module update status registers and trigger appropriate exception handling routines when memory-related errors are detected.
5. **Clock and Reset Synchronization:**
    a. **System Coordination:**
    The `clk_i` and `rst_i` inputs ensure that all operations within the CSR module are synchronized with the overall processor clock and that the module can be properly initialized or reset when needed.

III. Results: Post Synthesis Timing Simulation.

1. ECALL and CSRRW Instruction

Test Case Details:

```
#10 rst = 0;

#100
// Ecall
opcode_valid  = 1;
opcode_opcode = 32'h00000073;  // ECALL opcode
opcode_pc     = 32'h1004;      // Example PC that will be captured in mepc

#10;
 opcode_valid  = 0;

 #20  // Reading Mcause for Ecall through CSRRW Instruction
 opcode_valid = 1;
 opcode_opcode = {12'h342, 5'b00001, 3'b001, 5'b00010, 7'b1110011};  // CSRRW instruction
 opcode_ra_operand = 32'h11111100;

  #10;
 opcode_valid  = 0;

 #20
```

2. MRET

(We have different name in code: ERET)

Test Case Details:

```
// Allow time for exception processing (e.g., updating mepc, mcause, etc.)
// Return from Exception via ERET ---

 opcode_valid  = 1;
 opcode_opcode = 32'h10200073;  // ERET/MRET opcode
                                // For ERET, opcode_pc_i is typically not used; the return target is taken from mepc.
 #10;
 opcode_valid  = 0;
```

3. EBREAK

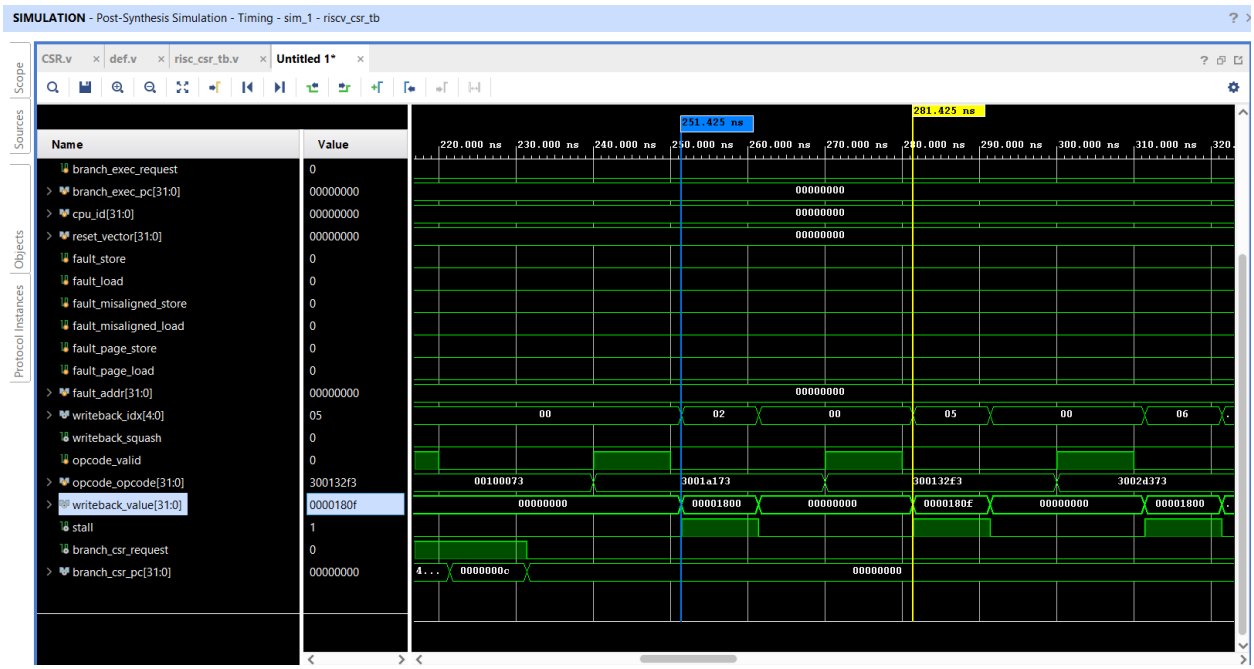
Test Case Details:

```
                                                    // For EBREAK, opcode_pc_i is typically not
#10;
opcode_valid  = 0;

#20   //EBREAK

opcode_valid  = 1;
opcode_opcode = 32'h00100073;

#10
opcode_valid = 0;

#20 //Reading Mcause of MRET/ERET
  opcode_valid = 1;
  opcode_opcode = {12'h342, 5'b00101, 3'b111, 5'b00111, 7'b1110011};
```

4. CSRRS Instruction

Test Case Details:

```
#20    // CSRRS Instruction Reading Mstatus
  opcode_valid = 1;
  opcode_opcode = {12'h300, 5'b00011, 3'b010, 5'b00010, 7'b1110011};  // CSRRS x3, mstatus, x1
  opcode_ra_operand = 32'h0000000F;  // rs1 = 0x0000000F //set the bits in those  specified by this Reg.(last 4 digit in this case)

#10
 opcode_valid = 0;

#20  //1111111F
  opcode_valid = 1;
  opcode_opcode = {12'h300, 5'b00010, 3'b011, 5'b00101, 7'b1110011};  // CSRRC x3, mstatus, x1 //Reading CSRRS
```

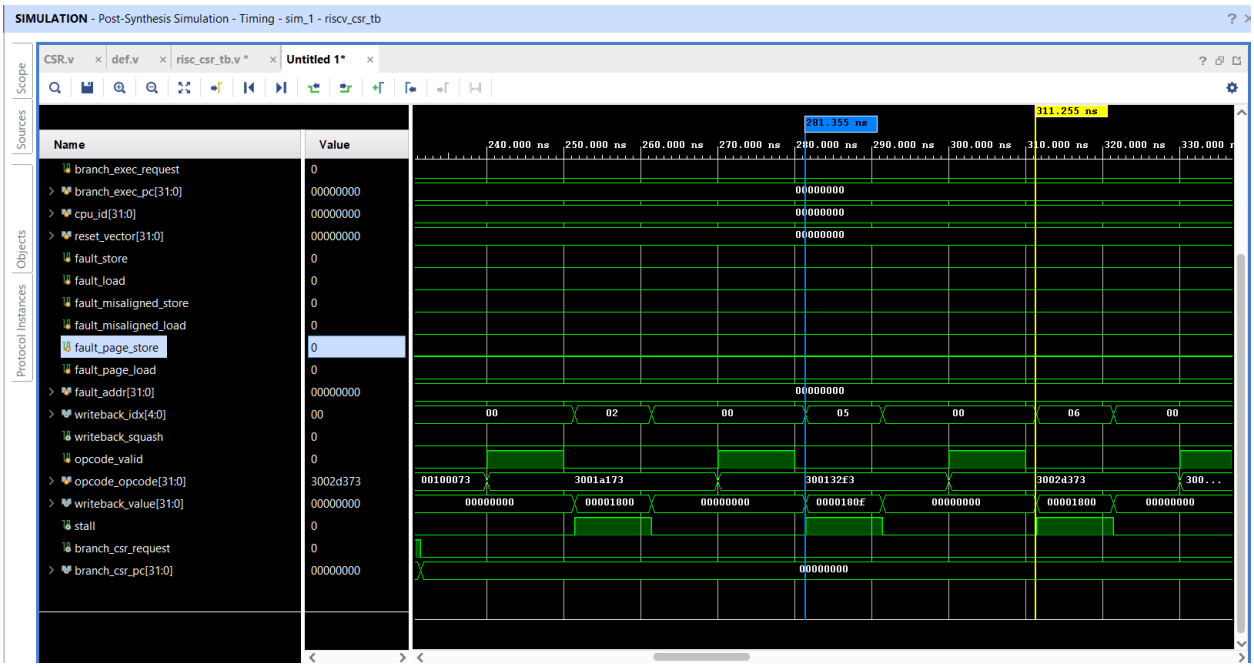


Note: `mstatus = 1800` occurs because an exception stores the previous privilege level (`mpriv`) in `mstatus[12:11]`, setting the MPRV bit

5. CSRRC Instruction

Test Case Details:

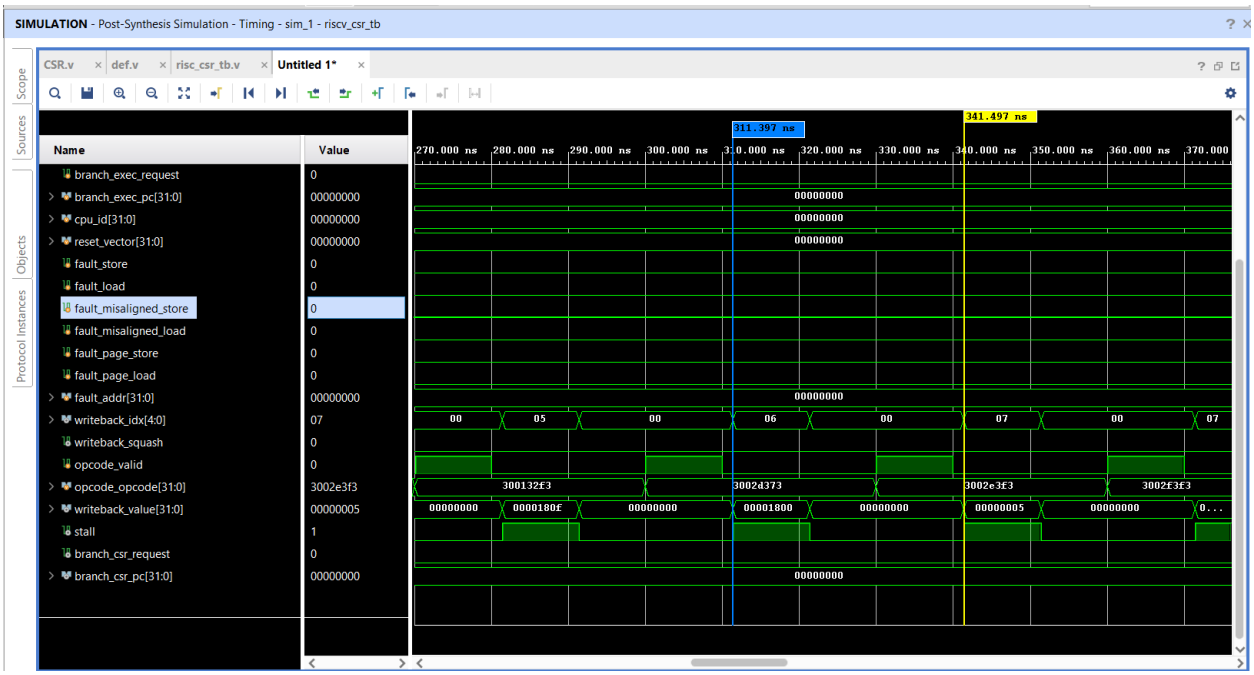
```
#10
 opcode_valid = 0;

#20 // CSRRC Instruction
  opcode_valid = 1;
  opcode_opcode = {12'h300, 5'b00010, 3'b011, 5'b00101, 7'b1110011};  // CSRRC x3, mstatus, x1 //Reading CSRRS
  //opcode_ra_operand = 32'h000000F0;  // rs1 = 0x0000000F
  opcode_ra_operand = 32'h0000000F;//clear the bits in those  specified by this Reg.(last 4 digit in this case)

#10
 opcode_valid = 0;

#20 //CSSRRWI
  opcode_valid = 1;
  opcode_opcode = {12'h300, 5'b00101, 3'b101, 5'b00110, 7'b1110011};  // CSRRWI x3, mstatus, x1  //Reading CSRRC
```



6. CSRRWI Instruction

Test Case Details:

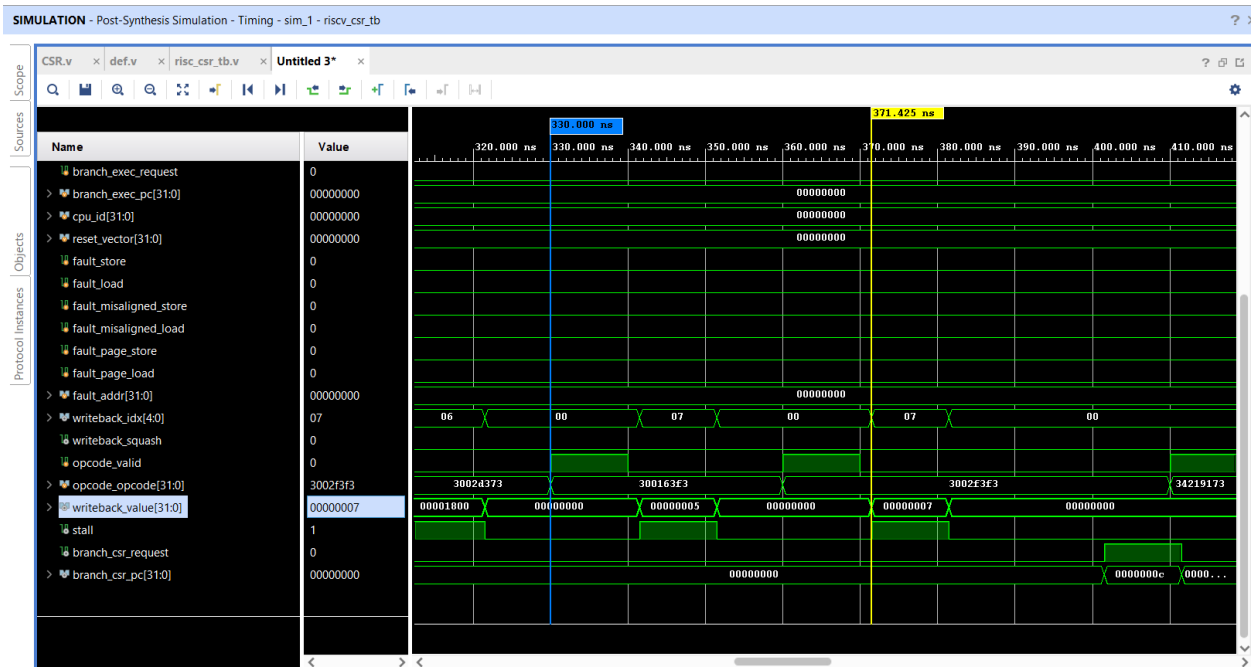
```
#10
 opcode_valid = 0;

#20 //CSSRRWI
  opcode_valid = 1;
  opcode_opcode = {12'h300, 5'b00101, 3'b101, 5'b00110, 7'b1110011};  // CSRRWI x3, mstatus, x1  //Reading CSRRC

#10
 opcode_valid = 0;

#20 //CSRRSI
  opcode_valid = 1;
  opcode_opcode = {12'h300, 5'b00101, 3'b110, 5'b00111, 7'b1110011};  // CSRRSI x3, mstatus, x1 //Reading CSSRWI
```



7. CSRRSI Instruction

Test Case Details:

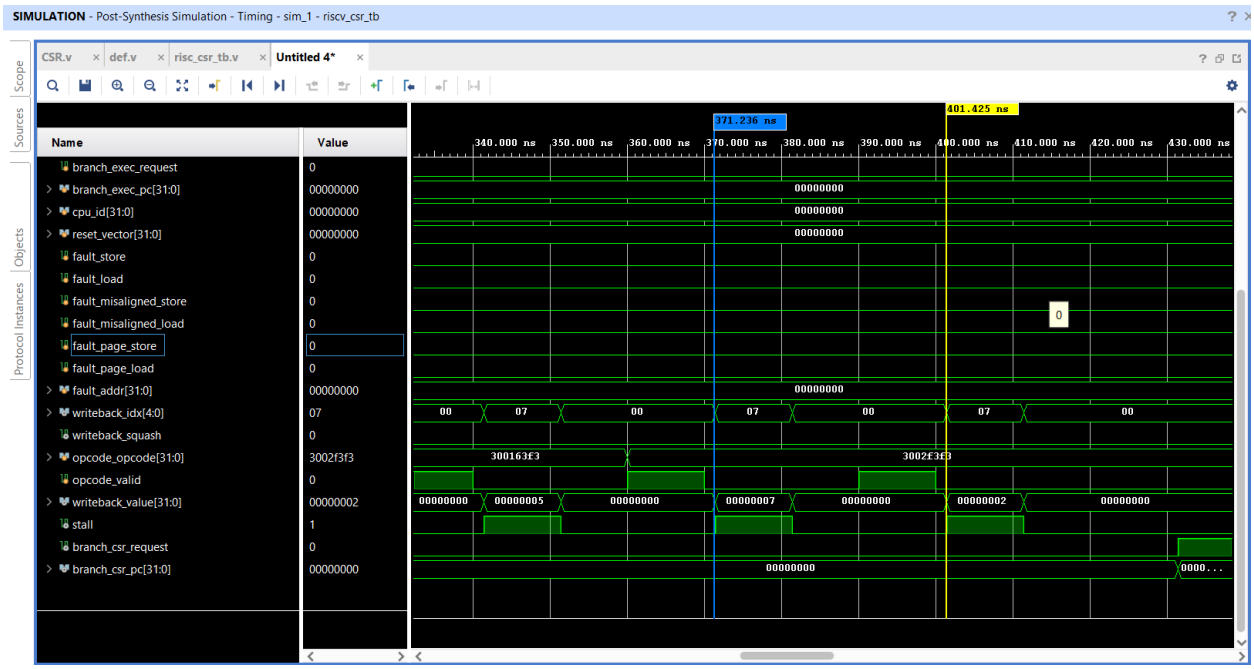
```
#20 //CSRRSI
  opcode_valid = 1;
  opcode_opcode = {12'h300, 5'b00101, 3'b110, 5'b00111, 7'b1110011};  // CSRRSI x3, mstatus, x1 //Reading CSSRWI

#10
 opcode_valid = 0;

#20 //CSRRCI
  opcode_valid = 1;
  opcode_opcode = {12'h300, 5'b00010, 3'b111, 5'b00111, 7'b1110011};  // CSRRC1 x3, mstatus, x1 //set the bits specified by rs1 for CSRRSI
```



8. CSRRCI Instruction

Test Case Details:

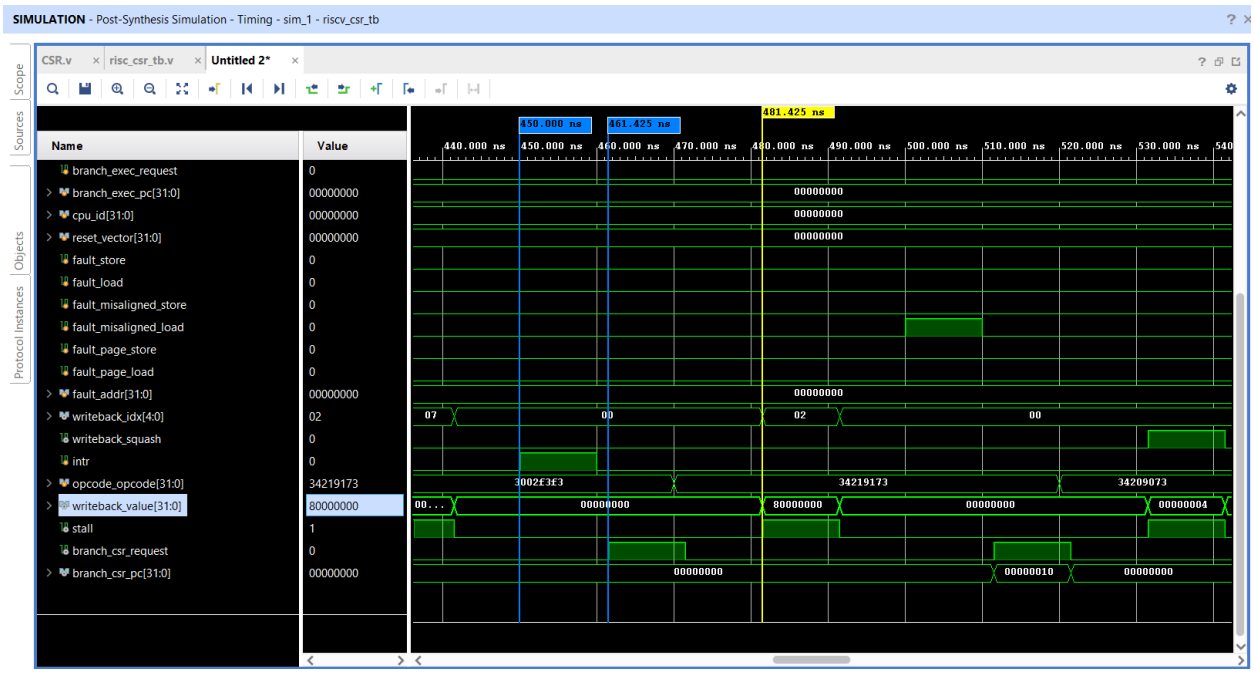
```
        #10
         opcode_valid = 0;

        #20 //CSRRCI
          opcode_valid = 1;
          opcode_opcode = {12'h300, 5'b00101, 3'b111, 5'b00111, 7'b1110011};  // CSRRC1 x3, mstatus, x1

        #10
         opcode_valid = 0;
```



10. Interrupt:

Test Case Details:

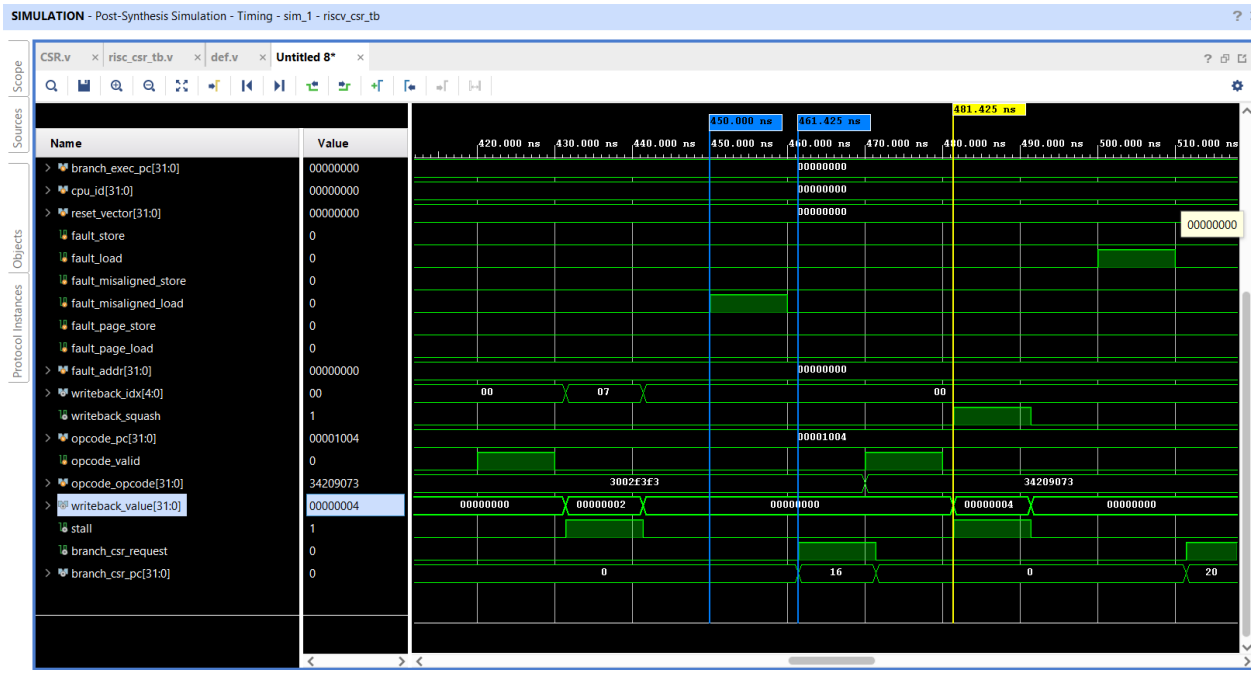
```
 // Step 2: Trigger an interrupt.
// When intr is asserted, the module will set temp_mip[11] = 1.
// Since mie now has bit 11 enabled, the condition (mie & mip != 0)
// will be true, and the module should set mcause to MCAUSE_INTERRUPT.
// It should then request a branch to mtvec + (MCAUSE_INTERRUPT << 2).
#20
intr = 1;
#10;
intr = 0;
#10

  opcode_valid = 1;
  opcode_opcode = {12'h342, 5'b00011, 3'b001, 5'b00010, 7'b1110011};  // CSRRW x3, mstatus, x1
  opcode_ra_operand = 32'h0000000F;  // rs1 = 0x0000000F
```



Exceptions:

1. Fault Misaligned Load

`Mcause=4`

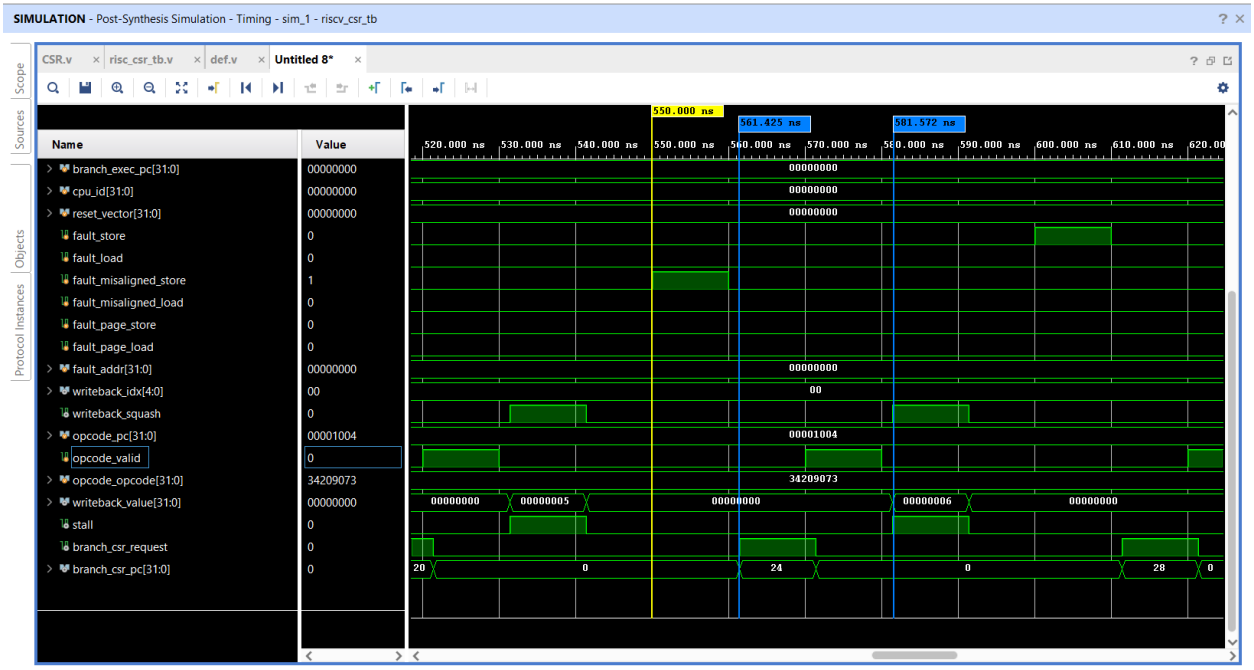


2. Fault Load

Mcause=5

3. Fault Misaligned Store

Mcause=6

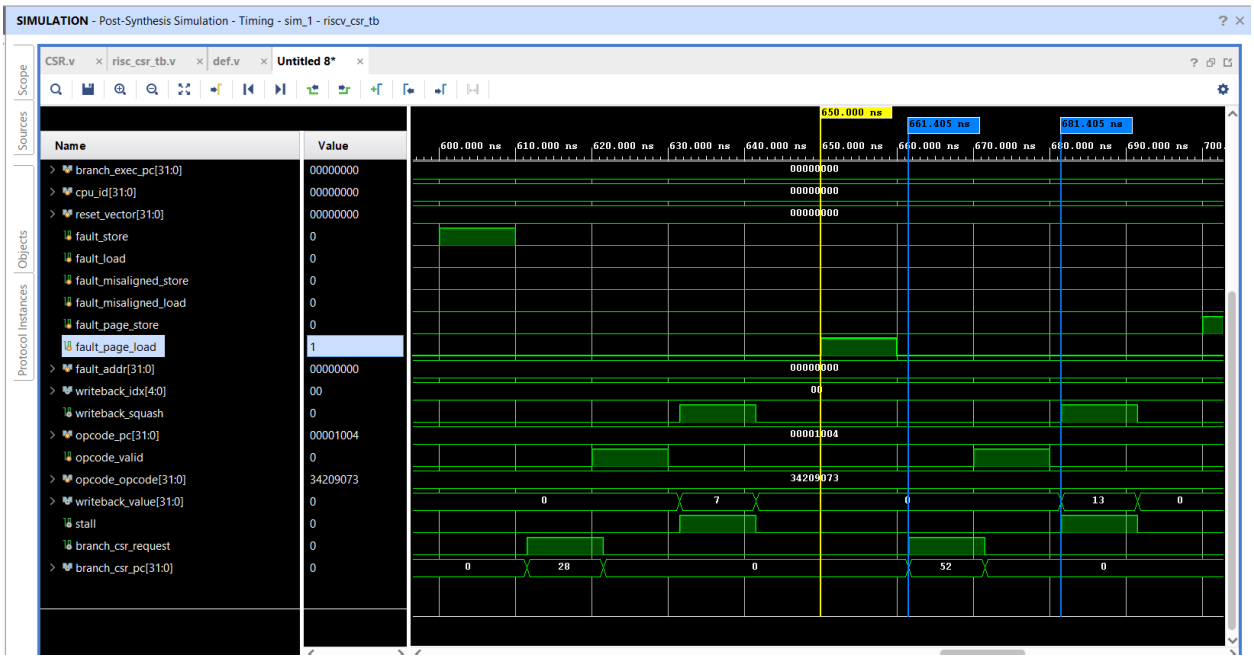


4. Fault Store

Mcause=7

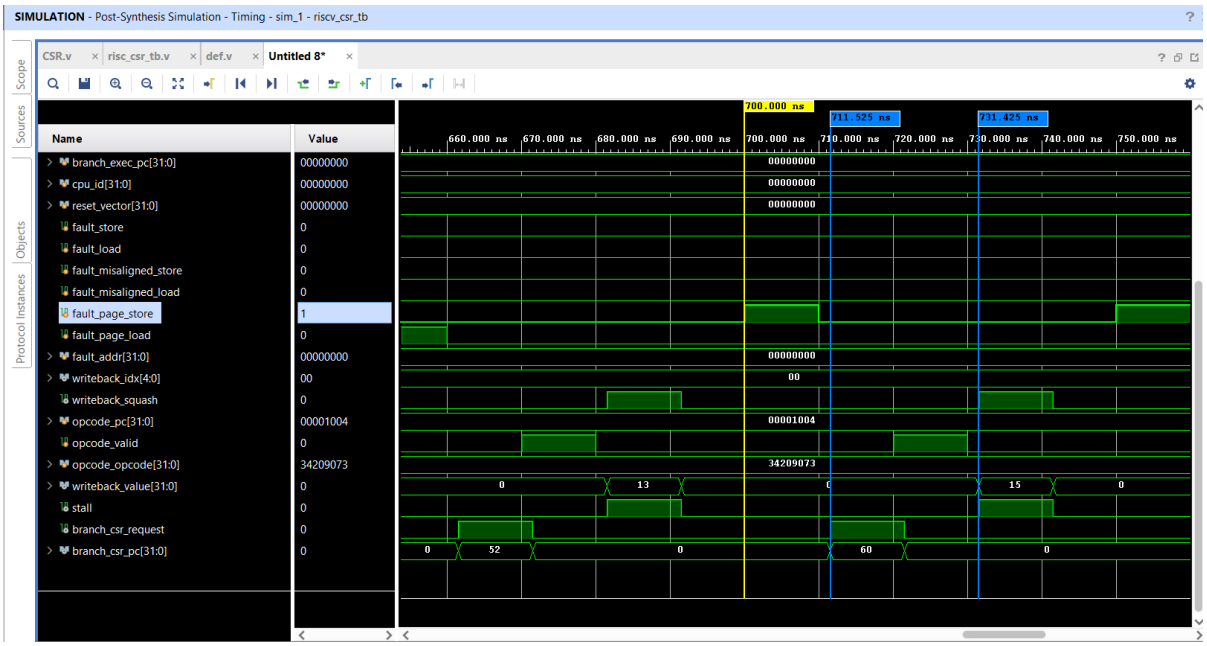
5. Fault Page Load

Mcause=13

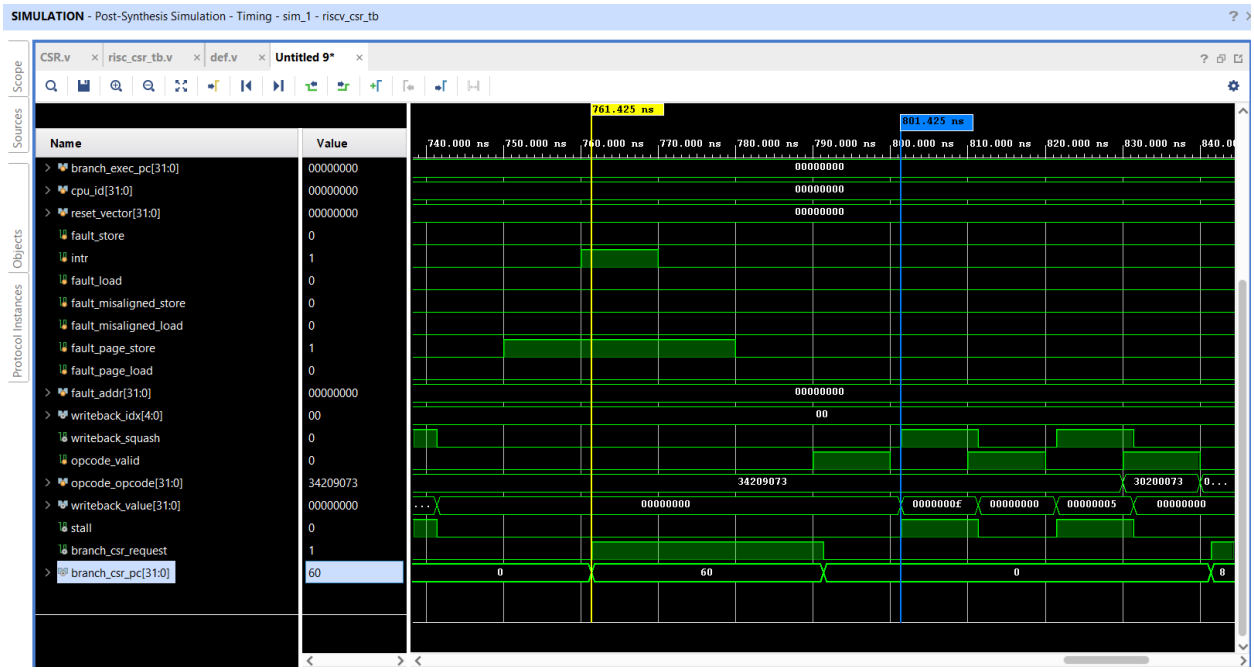


6. *Fault Page Store*

Mcause=15



7. Scenario: where both Fault(exception) and Interrupts occur simultaneously but exception has the priority



8. Illegal Instruction

Mcause=2

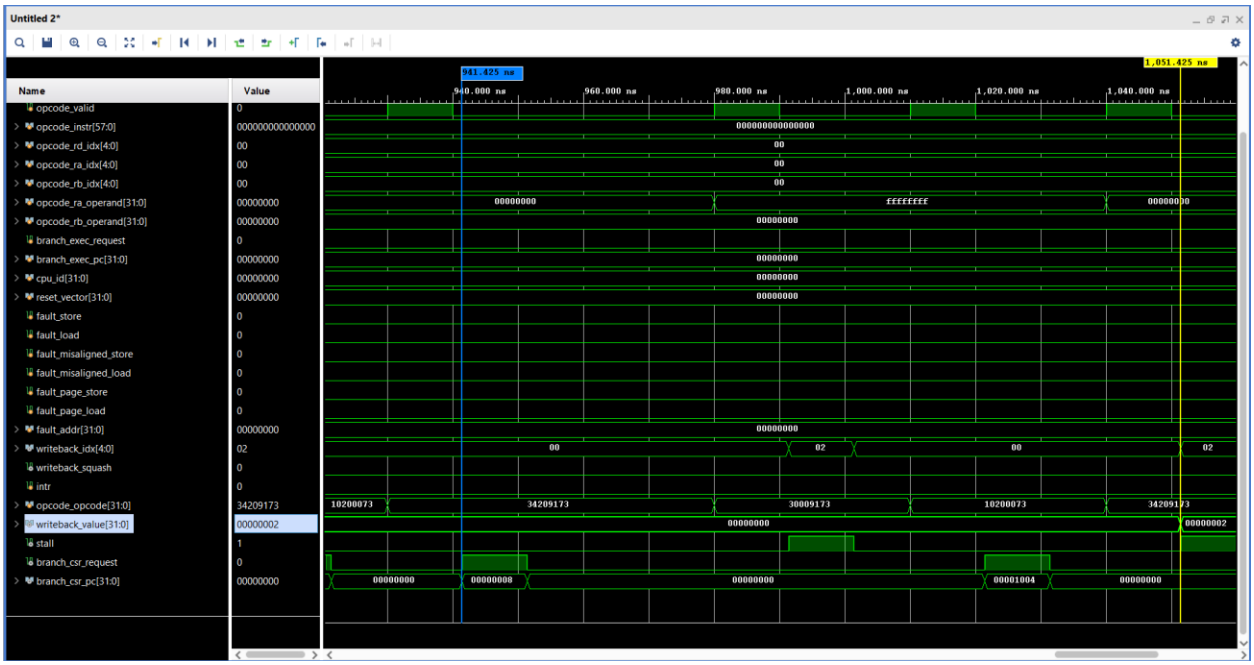
Test Case detail:

risc_csr_tb.v *

C:/M.tech/VLSI_design_lab/project_1/Project_1.srcs/sim_1/new/risc_csr_tb.v

```
319      #20//illegal instruction
320      opcode_valid = 1;
321      opcode_opcode = {12'h300, 5'b00001, 3'b001, 5'b00010, 7'b1110011};  // CSRRW instruction
322      opcode_ra_operand = 32'h00000000;
323
324      #10;
325      opcode_valid  = 0;
326      #20
327       opcode_valid  = 1;
328          opcode_opcode = 32'h10200073;  // ERET/MRET opcode
329
330      #10;
331      opcode_valid  = 0;
332
333      #20
334      opcode_valid = 1;
335      opcode_opcode = {12'h342, 5'b00001, 3'b001, 5'b00010, 7'b1110011};  // CSRRW instruction
336      opcode_ra_operand = 32'h00000000;
337      #10;
338      opcode_valid  = 0;
339       #20;
340
341       #20
342      opcode_valid = 1;
343      opcode_opcode = {12'h300, 5'b00001, 3'b001, 5'b00010, 7'b1110011};  // CSRRW instruction
344      opcode_ra_operand = 32'hFFFFFFFF;
345
346      #10;
347      opcode_valid  = 0;
348      #20
349       opcode_valid  = 1;
350          opcode_opcode = 32'h10200073;  // ERET/MRET opcode
351
352      #10;
353      opcode_valid  = 0;
354      #20
355      opcode_valid = 1;
356      opcode_opcode = {12'h342, 5'b00001, 3'b001, 5'b00010, 7'b1110011};  // CSRRW instruction
357      opcode_ra_operand = 32'h00000000;
358      #10;
359      opcode_valid  = 0;
```

Note: First, we set the privilege level to **user** by setting `mstatus[12:11]` to `00`. Then, we use **ERET** to set `mpriv` to `00` (user privilege). Finally, we execute a **CSRRW** instruction, which is not allowed in user mode, causing an **illegal instruction exception**. We then read the **mcause** value using **CSRRW** on the **mcause** register.



9. *Misaligned Fetch*

```
Mcause=0;
```