Lab6 Dynamically Scheduled Pipelines using Scoreboarding

3220106039 李瀚轩

1. Code Design

The `normal_stall` function is employed to identify whether a structural conflict or a Write-After-Write (WAW) conflict exists. In the case of structural conflicts, the function determines if the current instruction requires a specific functional unit and whether that functional unit is already in use. For WAW conflicts, the function checks if the destination register (`dst`) that the current instruction intends to write to is also being written to by another functional unit.

```
assign normal_stall = (use_FU != `FU_BLANK && FUS[use_FU][`BUSY]) |
(|RRS[dst]);
```

Next we need to ensure write after read. This code guarantees that before writing back to a register, all operations that depend on the value of that register have already read it. Specifically, when the `ALU_WAR` signal is active (true), it signifies that the ALU functional unit can write back to its destination register without causing conflicts with any other functional unit's read operations. Each condition evaluates whether the source register of another functional unit matches the ALU's destination register and whether that source register is ready to be accessed for reading.

```
wire ALU_WAR = (
        (FUS[`FU_MEM][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L] |
~FUS[`FU_MEM][`RDY1])  &    //fill sth. here
        (FUS[`FU_MEM][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L] |
~FUS[`FU_MEM][`RDY2])  &    //fill sth. here
        (FUS[`FU_MUL][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L] |
~FUS[`FU_MUL][`RDY1])  &    //fill sth. here
        (FUS[`FU_MUL][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L] |
~FUS[`FU_MUL][`RDY2])  &    //fill sth. here
        (FUS[`FU_DIV][`SRC1_H:`SRC1_L]  != FUS[`FU_ALU][`DST_H:`DST_L] |
~FUS[`FU_DIV][`RDY1])  &    //fill sth. here
        (FUS[`FU_DIV][`SRC2_H:`SRC2_L]  != FUS[`FU_ALU][`DST_H:`DST_L] |
~FUS[`FU_DIV][`RDY2])  &    //fill sth. here
        (FUS[`FU_JUMP][`SRC1_H:`SRC1_L] != FUS[`FU_ALU][`DST_H:`DST_L] |
~FUS[`FU_JUMP][`RDY1])  &    //fill sth. here
        (FUS[`FU_JUMP][`SRC2_H:`SRC2_L] != FUS[`FU_ALU][`DST_H:`DST_L] |
~FUS[`FU_JUMP][`RDY2])       //fill sth. here
    );
```

Next we need to maintain the scoreboard table.

```
always @ (posedge clk or posedge rst) begin
        if (rst) begin
        // reset the scoreboard
        for (i = 0; i < 32; i = i + 1) begin
            RRS[i] <= 3'b0;
        end

        for (i = 1; i <= 5; i = i + 1) begin
            FUS[i] <= 32'b0;
            IMM[i] <= 32'b0;
        end
    end

    else begin
        // IS
        if (RO_en) begin
            // not busy, no WAW, write info to FUS and RRS
            if (|dst) RRS[dst] <= use_FU;
            FUS[use_FU][`BUSY] <= 1'b1;
            FUS[use_FU][`OP_H:`OP_L] <= op;
            FUS[use_FU][`SRC1_H:`SRC1_L] <= src1;
            FUS[use_FU][`SRC2_H:`SRC2_L] <= src2;
            FUS[use_FU][`DST_H:`DST_L] <= dst;
            FUS[use_FU][`RDY1] <= rdy1;
            FUS[use_FU][`RDY2] <= rdy2;
            FUS[use_FU][`FU1_H:`FU1_L] <= fu1;
            FUS[use_FU][`FU2_H:`FU2_L] <= fu2;
            FUS[use_FU][`FU_DONE] <= 1'b0;

            IMM[use_FU] <= imm;
            PCR[use_FU] <= PC;
        end

        // RO
        if (FUS[`FU_JUMP][`RDY1] & FUS[`FU_JUMP][`RDY2]) begin
            // JUMP
            FUS[`FU_JUMP][`RDY1] <= 1'b0;
            FUS[`FU_JUMP][`RDY2] <= 1'b0;
        end
        else if (FUS[`FU_ALU][`RDY1] & FUS[`FU_ALU][`RDY2]) begin
//fill sth. here.
            // ALU
            FUS[`FU_ALU][`RDY1] <= 1'b0;
            FUS[`FU_ALU][`RDY2] <= 1'b0;
        end
        else if (FUS[`FU_MEM][`RDY1] & FUS[`FU_MEM][`RDY2]) begin
//fill sth. here.
            // MEM
            FUS[`FU_MEM][`RDY1] <= 1'b0;
            FUS[`FU_MEM][`RDY2] <= 1'b0;
        end
        else if (FUS[`FU_MUL][`RDY1] & FUS[`FU_MUL][`RDY2]) begin
//fill sth. here.
            // MUL
            FUS[`FU_MUL][`RDY1] <= 1'b0;
            FUS[`FU_MUL][`RDY2] <= 1'b0;
            end
            else if (FUS[`FU_DIV][`RDY1] & FUS[`FU_DIV][`RDY2]) begin
    //fill sth. here.
                // DIV
                FUS[`FU_DIV][`RDY1] <= 1'b0;
                FUS[`FU_DIV][`RDY2] <= 1'b0;
            end

            // EX
            if(ALU_done) FUS[`FU_ALU][`FU_DONE] <= 1'b1;
            if(DIV_done) FUS[`FU_DIV][`FU_DONE] <= 1'b1;
            if(MUL_done) FUS[`FU_MUL][`FU_DONE] <= 1'b1;
            if(MEM_done) FUS[`FU_MEM][`FU_DONE] <= 1'b1;
            if(JUMP_done) FUS[`FU_JUMP][`FU_DONE] <= 1'b1;

            // WB
            if (FUS[`FU_JUMP][`FU_DONE] & JUMP_WAR) begin
                FUS[`FU_JUMP] <= 32'b0;
                RRS[FUS[`FU_JUMP][`DST_H:`DST_L]] <= 3'b0;

                // ensure RAW
                if (FUS[`FU_ALU][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_ALU]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_MEM]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_MUL]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_JUMP) FUS[`FU_DIV]
[`RDY1] <= 1'b1;           //fill sth. here

                if (FUS[`FU_ALU][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_ALU]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_MEM]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_MUL]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_DIV][`FU2_H:`FU2_L] == `FU_JUMP) FUS[`FU_DIV]
[`RDY2] <= 1'b1;           //fill sth. here
            end
            // ALU
            else if (FUS[`FU_ALU][`FU_DONE] & ALU_WAR) begin
                FUS[`FU_ALU] <= 32'b0;
                RRS[FUS[`FU_ALU][`DST_H:`DST_L]] <= 3'b0;

                // ensure RAW
                if (FUS[`FU_JUMP][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_JUMP]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_MEM]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_MUL]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_ALU) FUS[`FU_DIV]
[`RDY1] <= 1'b1;           //fill sth. here
```

```
                if (FUS[`FU_JUMP][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_JUMP]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_MEM]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_MUL]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_DIV][`FU2_H:`FU2_L] == `FU_ALU) FUS[`FU_DIV]
[`RDY2] <= 1'b1;           //fill sth. here
            end
            // MEM
            else if (FUS[`FU_MEM][`FU_DONE] & MEM_WAR) begin
                FUS[`FU_MEM] <= 32'b0;
                RRS[FUS[`FU_MEM][`DST_H:`DST_L]] <= 3'b0;

                // ensure RAW
                if (FUS[`FU_JUMP][`FU1_H:`FU1_L] == `FU_MEM) FUS[`FU_JUMP]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_ALU][`FU1_H:`FU1_L] == `FU_MEM) FUS[`FU_ALU]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_MEM) FUS[`FU_MUL]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_MEM) FUS[`FU_DIV]
[`RDY1] <= 1'b1;           //fill sth. here

                if (FUS[`FU_JUMP][`FU2_H:`FU2_L] == `FU_MEM) FUS[`FU_JUMP]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_ALU][`FU2_H:`FU2_L] == `FU_MEM) FUS[`FU_ALU]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_MEM) FUS[`FU_MUL]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_DIV][`FU2_H:`FU2_L] == `FU_MEM) FUS[`FU_DIV]
[`RDY2] <= 1'b1;           //fill sth. here
            end
            // MUL
            else if (FUS[`FU_MUL][`FU_DONE] & MUL_WAR) begin
                FUS[`FU_MUL] <= 32'b0;
                RRS[FUS[`FU_MUL][`DST_H:`DST_L]] <= 3'b0;

                // ensure RAW
                if (FUS[`FU_JUMP][`FU1_H:`FU1_L] == `FU_MUL) FUS[`FU_JUMP]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_ALU][`FU1_H:`FU1_L] == `FU_MUL) FUS[`FU_ALU]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_MUL) FUS[`FU_MEM]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_DIV][`FU1_H:`FU1_L] == `FU_MUL) FUS[`FU_DIV]
[`RDY1] <= 1'b1;           //fill sth. here

                if (FUS[`FU_JUMP][`FU2_H:`FU2_L] == `FU_MUL) FUS[`FU_JUMP]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_ALU][`FU2_H:`FU2_L] == `FU_MUL) FUS[`FU_ALU]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_MUL) FUS[`FU_MEM]
[`RDY2] <= 1'b1;           //fill sth. here
```

```
                if (FUS[`FU_DIV][`FU2_H:`FU2_L] == `FU_MUL) FUS[`FU_DIV]
[`RDY2] <= 1'b1;           //fill sth. here
            end
            // DIV
            else if (FUS[`FU_DIV][`FU_DONE] & DIV_WAR) begin
                FUS[`FU_DIV] <= 32'b0;
                RRS[FUS[`FU_DIV][`DST_H:`DST_L]] <= 3'b0;

                // ensure RAW
                if (FUS[`FU_JUMP][`FU1_H:`FU1_L] == `FU_DIV) FUS[`FU_JUMP]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_ALU][`FU1_H:`FU1_L] == `FU_DIV) FUS[`FU_ALU]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MEM][`FU1_H:`FU1_L] == `FU_DIV) FUS[`FU_MEM]
[`RDY1] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MUL][`FU1_H:`FU1_L] == `FU_DIV) FUS[`FU_MUL]
[`RDY1] <= 1'b1;           //fill sth. here

                if (FUS[`FU_JUMP][`FU2_H:`FU2_L] == `FU_DIV) FUS[`FU_JUMP]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_ALU][`FU2_H:`FU2_L] == `FU_DIV) FUS[`FU_ALU]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MEM][`FU2_H:`FU2_L] == `FU_DIV) FUS[`FU_MEM]
[`RDY2] <= 1'b1;           //fill sth. here
                if (FUS[`FU_MUL][`FU2_H:`FU2_L] == `FU_DIV) FUS[`FU_MUL]
[`RDY2] <= 1'b1;           //fill sth. here
            end
        end
    end
```

The primary purpose of this code is to manage the scoreboard state, ensuring the proper execution and write-back of instructions. The detailed steps are as follows:

1. **Reset Stage**: When the reset signal `rst` is active (high), all registers and functional unit states in the scoreboard are reset to their initial conditions.
2. **Instruction Issue (IS) Stage**: If the `RO_en` signal is high, it indicates that the current instruction can be issued. The scoreboard updates the Functional Unit Status (FUS) and Register Read Status (RRS) tables, recording the current instruction's destination register, source registers, opcode, and other relevant information.
3. **Register Operand (RO) Stage**: The code checks whether the source registers of each functional unit are ready to be read. If they are, the `RDY1` and `RDY2` signals in the FUS table are updated, and the `FU1` and `FU2` flags are cleared.
4. **Execution (EX) Stage**: Based on the `done` signal from each functional unit, the `FU_DONE` signal in the FUS table is updated, indicating that the functional unit has completed its execution.
5. **Write-Back (WB) Stage**: When a functional unit has finished execution and there is no Write-After-Read (WAR) hazard, the write-back operation is performed, and the FUS and RRS tables are updated accordingly. Additionally, the code checks if any other functional units are waiting for the result of the current

functional unit. If so, the waiting state is cleared, and the ready signal for the operand is set to 1.

Lastly, we will generate control signals.

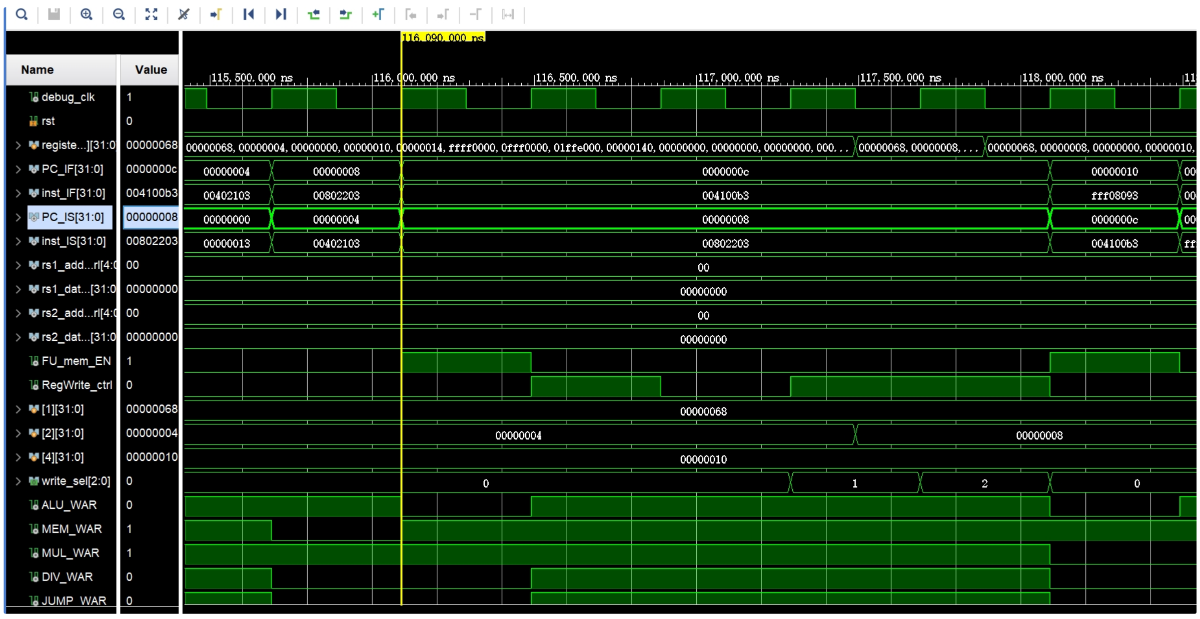
```
always @ (*) begin
        ALU_en = 0;
        MEM_en = 0;
        MUL_en = 0;
        DIV_en = 0;
        JUMP_en = 0;

        rs1_ctrl = 0;
        rs2_ctrl = 0;
        PC_ctrl = 0;
        imm_ctrl = 0;
        JUMP_op = 0;
        ALU_op = 0;
        ALU_use_PC = 0;
        ALU_use_imm = 0;
        MEM_we = 0;
        MEM_bhw = 0;
        MUL_op = 0;
        DIV_op = 0;

        // JUMP
        if (FUS[`FU_JUMP][`RDY1] & FUS[`FU_JUMP][`RDY2]) begin
            ALU_en = 1'b0;
            MEM_en = 1'b0;
            MUL_en = 1'b0;
            DIV_en = 1'b0;
            JUMP_en = 1'b1;

            JUMP_op = FUS[`FU_JUMP][`OP_H:`OP_L];
            rs1_ctrl = FUS[`FU_JUMP][`SRC1_H:`SRC1_L];
            rs2_ctrl = FUS[`FU_JUMP][`SRC2_H:`SRC2_L];
            PC_ctrl = PCR[`FU_JUMP];
            imm_ctrl = IMM[`FU_JUMP];
        end
        // ALU
        else if (FUS[`FU_ALU][`RDY1] & FUS[`FU_ALU][`RDY2]) begin
            ALU_en = 1'b1;
            MEM_en = 1'b0;
            MUL_en = 1'b0;
            DIV_en = 1'b0;
            JUMP_en = 1'b0;

            ALU_op = FUS[`FU_ALU][`OP_H:`OP_L] == `ALU_AUIPC ?
                      4'b0001 : FUS[`FU_ALU][`OP_L+3:`OP_L];
            rs1_ctrl = FUS[`FU_ALU][`SRC1_H:`SRC1_L];
            rs2_ctrl = FUS[`FU_ALU][`SRC2_H:`SRC2_L];
            ALU_use_PC = FUS[`FU_ALU][`OP_H:`OP_L] == `ALU_AUIPC;
            ALU_use_imm = FUS[`FU_ALU][`OP_H];
            PC_ctrl = PCR[`FU_ALU];
            imm_ctrl = IMM[`FU_ALU];
        end
        // MEM
        else if (FUS[`FU_MEM][`RDY1] & FUS[`FU_MEM][`RDY2]) begin
            ALU_en = 1'b0;
            MEM_en = 1'b1;
            MUL_en = 1'b0;
            DIV_en = 1'b0;
            JUMP_en = 1'b0;

            MEM_we = FUS[`FU_MEM][`OP_L];
            MEM_bhw = FUS[`FU_MEM][`OP_L+3:`OP_L+1];
            rs1_ctrl = FUS[`FU_MEM][`SRC1_H:`SRC1_L];
            rs2_ctrl = FUS[`FU_MEM][`SRC2_H:`SRC2_L];   // if store
            imm_ctrl = IMM[`FU_MEM];
        end
        // MUL
        else if (FUS[`FU_MUL][`RDY1] & FUS[`FU_MUL][`RDY2]) begin
            ALU_en = 1'b0;
            MEM_en = 1'b0;
            MUL_en = 1'b1;
            DIV_en = 1'b0;
            JUMP_en = 1'b0;

            MUL_op = FUS[`FU_MUL][`OP_L+2:`OP_L];
            rs1_ctrl = FUS[`FU_MUL][`SRC1_H:`SRC1_L];
            rs2_ctrl = FUS[`FU_MUL][`SRC2_H:`SRC2_L];
        end
        else if (FUS[`FU_DIV][`RDY1] & FUS[`FU_DIV][`RDY2]) begin
            ALU_en = 1'b0;
            MEM_en = 1'b0;
            MUL_en = 1'b0;
            DIV_en = 1'b1;
            JUMP_en = 1'b0;

            DIV_op = FUS[`FU_DIV][`OP_L+1:`OP_L];
            rs1_ctrl = FUS[`FU_DIV][`SRC1_H:`SRC1_L];
            rs2_ctrl = FUS[`FU_DIV][`SRC2_H:`SRC2_L];
        end
    end

    // WB
    always @ (*) begin
        write_sel = 0;
        reg_write = 0;
        rd_ctrl = 0;

        if (FUS[`FU_JUMP][`FU_DONE] & JUMP_WAR) begin
            write_sel = 3'd4;
            reg_write = 1'b1;
            rd_ctrl = FUS[`FU_JUMP][`DST_H:`DST_L];
        end
        else if (FUS[`FU_ALU][`FU_DONE] & ALU_WAR) begin
            write_sel = 3'd0;
```

```
                reg_write = 1'b1;
                rd_ctrl = FUS[`FU_ALU][`DST_H:`DST_L];
            end
            else if (FUS[`FU_MEM][`FU_DONE] & MEM_WAR) begin
                write_sel = 3'd1;
                reg_write = 1'b1;
                rd_ctrl = FUS[`FU_MEM][`DST_H:`DST_L];
            end
            else if (FUS[`FU_MUL][`FU_DONE] & MUL_WAR) begin
                write_sel = 3'd2;
                reg_write = 1'b1;
                rd_ctrl = FUS[`FU_MUL][`DST_H:`DST_L];
            end
            else if (FUS[`FU_DIV][`FU_DONE] & DIV_WAR) begin
                write_sel = 3'd3;
                reg_write = 1'b1;
                rd_ctrl = FUS[`FU_DIV][`DST_H:`DST_L];
            end
        end
endmodule
```

The primary purpose of this code is to generate control signals that ensure instructions are executed and written back at the appropriate stages. The detailed steps are as follows:

1. **Register Operand (RO) Stage:** Based on the state information in the scoreboard, the code generates enable signals for each functional unit (`ALU_en` , `MEM_en` , `MUL_en` , `DIV_en` , `JUMP_en`). Additionally, it sets the corresponding opcode, source registers, immediate values, and other necessary parameters for each functional unit.
2. **Write-Back (WB) Stage:** Based on the state in the scoreboard, the code generates write-back control signals (`write_sel` , `reg_write` , `rd_ctrl`) to ensure that the results are correctly written back to the register file when the functional unit has completed execution and there is no Write-After-Read (WAR) hazard.

2. Simulation



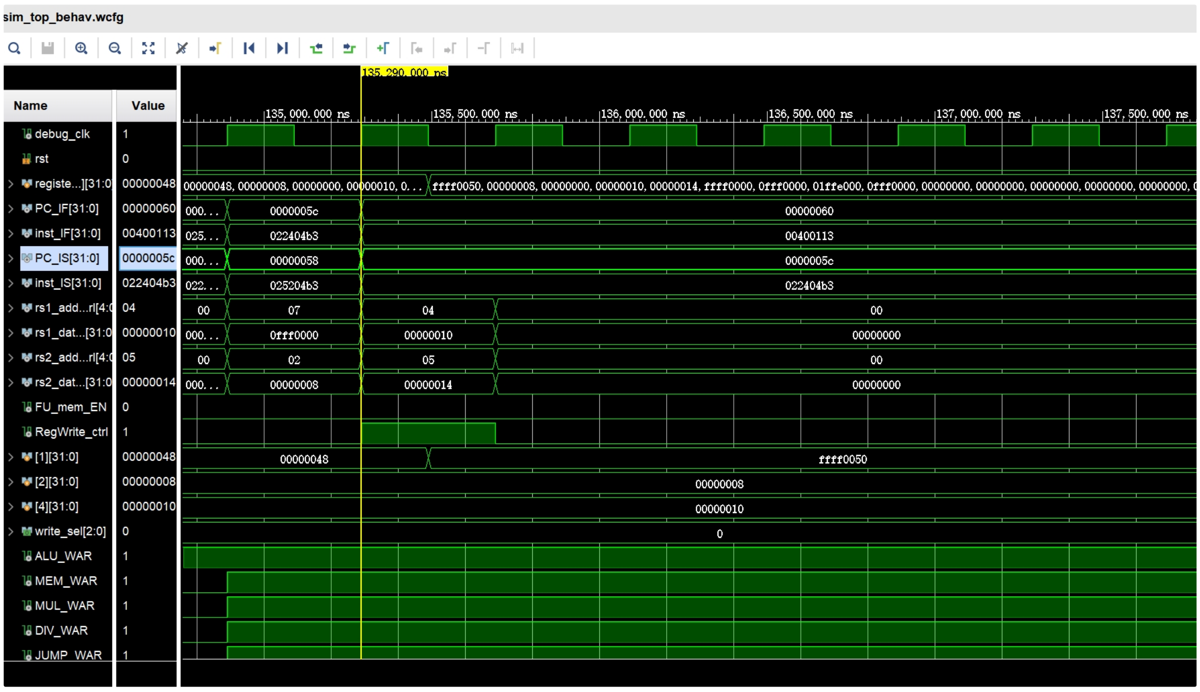
In this scenario, we observe the following:

- The instruction `lw x2, 4(x0)` is currently in the **Execution (EX)** stage, meaning it is actively using the **Memory (MEM)** functional unit.
- The instruction `lw x4, 8(x0)` is ready to be issued, but it also requires the **MEM** functional unit for execution.
- Since the **MEM** functional unit is already in use by the first instruction, a **structural conflict** occurs, leading to a stall. This is indicated by `normal_stall` being set to `1`.

After four cycles, the first instruction (`lw x2, 4(x0)`) completes its write-back stage, freeing up the **MEM** functional unit. At this point:

- `normal_stall` is set to `0`, indicating that the structural conflict has been resolved.
- `IS_en` (Instruction Issue Enable) becomes `1`, allowing the second instruction (`lw x4, 8(x0)`) to be issued and proceed to the next stage.

This demonstrates how the system ensures proper resource management and avoids conflicts by stalling when necessary and resuming execution once resources become available.



At the specified addresses, the following sequence of events occurs due to structural contention and data dependencies:

1. **Structural Contention at 0x58 and 0x5C:**
   - The instruction at address **0x58** is currently using a shared resource (e.g., a functional unit like the ALU or MEM).
   - The instruction at address **0x5C** is ready to be issued but cannot proceed because the shared resource is still in use by the instruction at **0x58**.
   - As a result, the instruction at **0x5C** is stalled until the instruction at **0x58** completes its execution and releases the resource.
2. **Data Dependency at 0x60:**
   - The instruction at address **0x60** depends on the value of register `x2`, which is being read by the instruction at **0x5C**.
   - The instruction at **0x60** must wait for the instruction at **0x5C** to read the value of `x2` before it can perform its write-back operation.
   - This ensures that the correct value of `x2` is available for the instruction at **0x60** to use, avoiding data hazards.

In summary:

- Structural contention at **0x58** and **0x5C** causes a stall until the resource is freed.
- The data dependency at **0x60** ensures proper sequencing of operations, preventing incorrect results due to out-of-order execution.