Lab5: Pipelined CPU supporting multi-cycle operations

3220106039 李瀚轩

I. Objectives and Requirements

1. Understand the principles of pipelines that support multi-cycle operations.
2. Master the design methods of pipelines supporting multi-cycle operations.
3. Master the verification methods for a pipelined CPU supporting multi-cycle operations.

II. Experiment Process

Compared to a standard 5-stage pipeline CPU, the functionality of the IF and ID stages remains unchanged, while the EXE and MEM stages are merged into a single "FU" (Functional Unit) stage. The FU stage handles ALU operations, multiplication, division, memory access, and branch/jump operations. The WB stage uses a multiplexer to select the result.

1. **ALU Module:**

The core code for the ALU module is as follows:

```
always@(posedge clk) begin
        if(EN & ~state) begin // state == 0
            Control <= ALUControl;   //to fill sth.in
            A <= ALUA;
            B <= ALUB;
            state <= 1;
        end
        else state <= 0;
    end
```

Here, we have a `state` register that indicates whether the ALU is performing a computation. The initial state is `0`, and when the enable signal (`EN`) is high and `state == 0`, the `Control`, `A`, and `B` signals are loaded with `ALUControl`, `ALUA`, and `ALUB` respectively. Once the computation is complete, the state is reset to `0`.

2. **Multiplication Unit:**

This module uses a 7-bit register `state` to control the multiplication state. `state[0]` indicates whether multiplication is complete. When `EN` is 1 and `state == 0`, the values `A` and `B` are loaded into `A_reg` and `B_reg`, and the highest bit of `state` (`state[6]`) is set to 1, indicating the start of multiplication. The `always` block is clocked on the rising edge, and the `state` register is shifted after each clock cycle to simulate the state machine.

```
always@(posedge clk) begin
    if(EN & state == 0) begin // state == 0
        A_reg <= A;
        B_reg <= B;
        state[6] <= 1;  // Start multiplication
    end
    else state <= (state >> 1);  // Shift state for next cycle
end
```

3. **Division Module:**

This module uses a 1-bit `state` register to control the division unit. The initial state is `0`, indicating no division operation is in progress. When `EN` is 1 and `state == 0`, the division operation begins. `res_valid` indicates whether the result is available. The following code shows the division control:

```
always@(posedge clk) begin
    if(EN & ~state & ~res_valid) begin // state == 0
        A_reg <= A;
        A_valid <= 1;
        B_reg <= B;
        B_valid <= 1;
        state <= 1;
    end
    else if(res_valid) begin
        A_valid <= 0;
        B_valid <= 0;
        state <= 0;
    end
end
```

When `EN == 1`, `state == 0`, and `res_valid == 0`, the division computation begins by loading the values of `A` and `B` into the registers, setting `A_valid` and `B_valid` to `1`, and updating the `state`. Once the division is complete (`res_valid == 1`), the division process ends, and the state is reset.

4. **Jump Module:**

This logic is similar to the previous ones but includes an `EN` signal and a `state` register. When `state == 1`, the jump unit is active. The initial state is `0`, indicating that the jump operation hasn't started. The `always` block controls the loading of data into the registers. When `EN == 1` and `state == 0`, the module loads data into the registers, and the state is set to `1`, indicating the start of the jump operation.

```
`timescale 1ns / 1ps

module FU_jump(
    input clk, EN, JALR,
    input[2:0] cmp_ctrl,
    input[31:0] rs1_data, rs2_data, imm, PC,
    output[31:0] PC_jump, PC_wb,
    output cmp_res, finish
);

    reg state;
    assign finish = state == 1'b1;
    initial begin
        state = 0;
    end

    reg JALR_reg;
    reg[2:0] cmp_ctrl_reg;
    reg[31:0] rs1_data_reg, rs2_data_reg, imm_reg, PC_reg;

    always@(posedge clk) begin
        if(EN & ~state) begin //state == 0
            JALR_reg <= JALR;
            cmp_ctrl_reg <= cmp_ctrl;
            rs1_data_reg <= rs1_data;
            rs2_data_reg <= rs2_data;
            imm_reg <= imm;
            PC_reg <= PC;
            state <= 1;
        end
        else begin
            state <= 0;
        end
    end

    cmp_32 cmp(.a(rs1_data_reg), .b(rs2_data_reg), .ctrl(cmp_ctrl_reg),
.c(cmp_res));
    assign PC_jump = JALR ? (rs1_data_reg + imm_reg) : (PC_reg + imm_reg);
    assign PC_wb = PC_reg + 32'd4;              //to fill sth.in

endmodule
```

The module compares `rs1_data` and `rs2_data` using the `cmp_32` module, and the result is stored in `cmp_res`. The jump address is calculated depending on the `JALR` signal. If `JALR` is 1, the jump address is `rs1_data + imm`. Otherwise, the jump address is `PC + imm`. The write-back address (`PC_wb`) is `PC + 4`, which points to the next instruction.

5. **MEM Module:**

Memory access also supports multiple cycles. We use a 2-bit `state` register similar to the multiplication module. The `finish` signal is generated when `state[0] == 1'b1`, indicating that the memory operation is complete. When `state[0] == 1`, `finish` is 1, indicating the completion of the operation. The `always` block controls the loading of registers and the state transition. When `EN == 1` and `state == 00`, the module loads input data and calculates the memory address, then begins the memory operation. The input values `rs1_data`, `rs2_data`, `imm`, `mem_w`, and `bhw` are stored in the registers, and the memory address is calculated as `addr = imm + rs1_data`.

```
always@(posedge clk) begin
    if(EN & state == 0) begin // state == 0
```

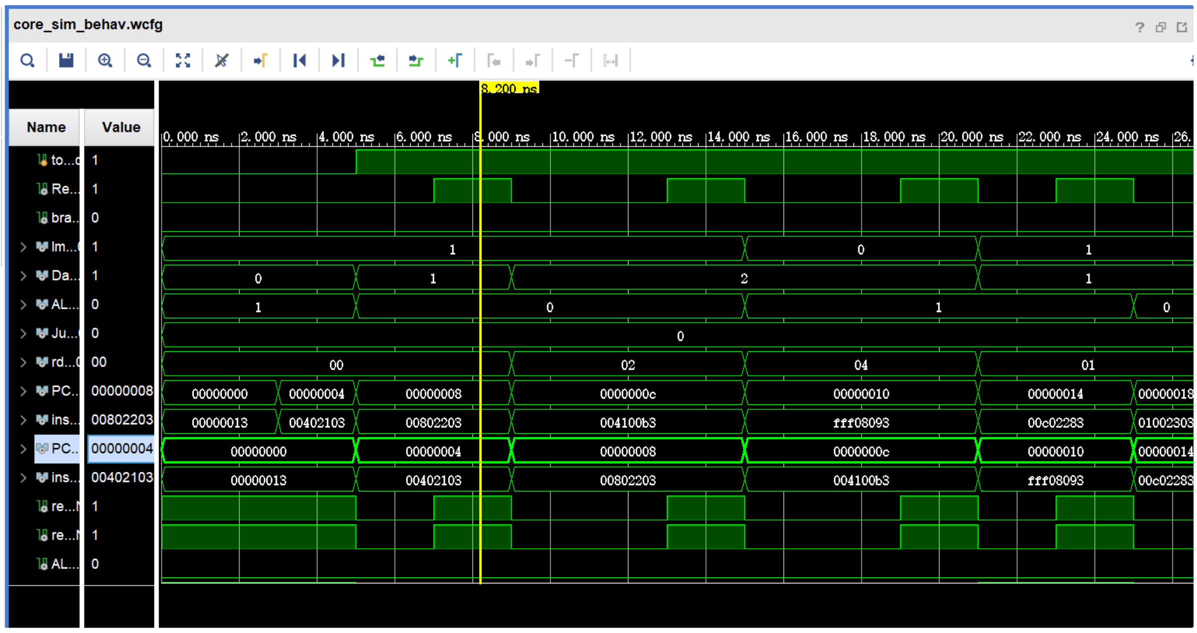
```
        rs1_data_reg <= rs1_data;
        rs2_data_reg <= rs2_data;
        imm_reg <= imm;
        mem_w_reg <= mem_w;
        bhw_reg <= bhw;
        addr <= imm + rs1_data;
        state[1] <= 1;
    end
    else begin
        state <= (state >> 1);
    end
end
```

6. **RV32Core:**

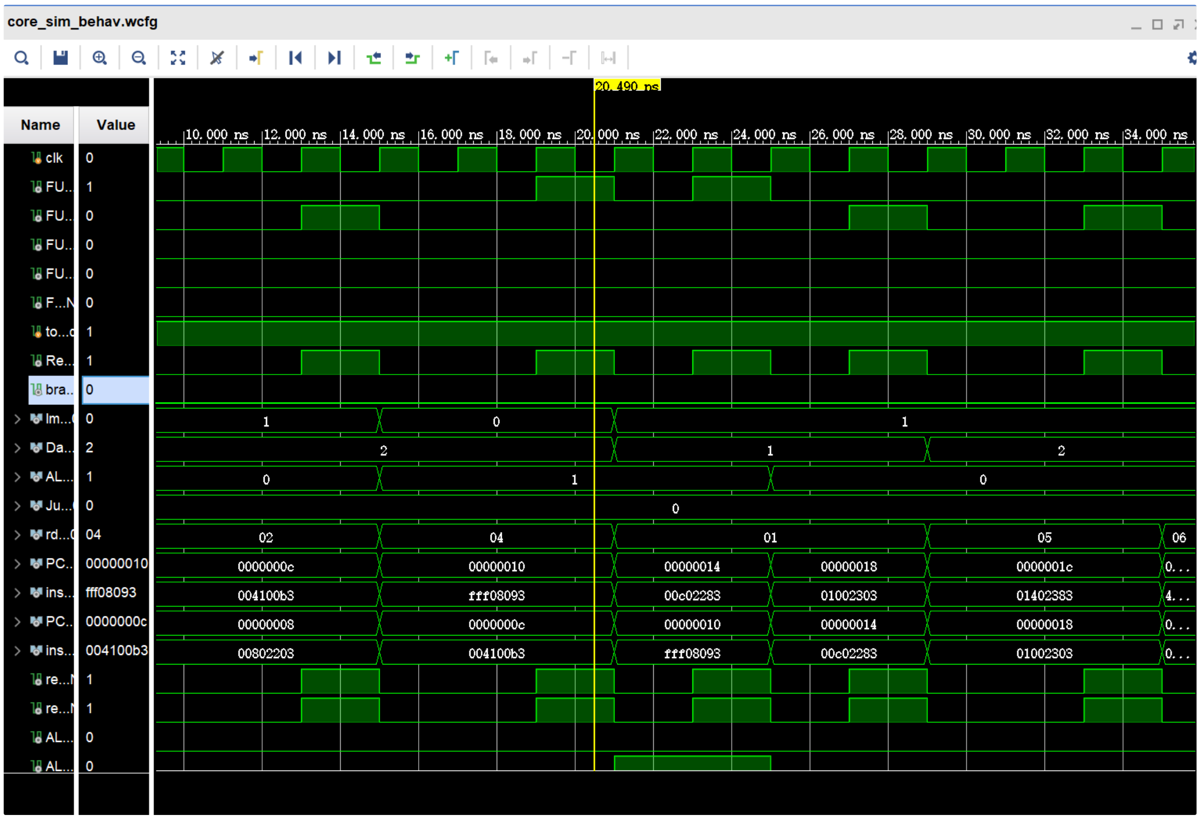
We need to complete the `ImmGen` and ALU data input multiplexers, as well as the final WB stage multiplexer, based on the starter code. These connections follow the architecture diagram.

```
ImmGen
imm_gen(.ImmSel(ImmSel_ctrl),.inst_field(inst_ID),.Imm_out(Imm_out_ID));
     //to fill sth.in
MUX2T1_32
mux_imm_ALU_ID_A(.I0(rs1_data_ID),.I1(PC_ID),.s(ALUSrcA_ctrl),.o(ALUA_ID));



    MUX2T1_32
mux_imm_ALU_ID_B(.I0(rs2_data_ID),.I1(Imm_out_ID),.s(ALUSrcB_ctrl),.o(ALUB_ID)
);

MUX8T1_32
mux_DtR(.s(DatatoReg_ctrl),.I0(32'd0),.I1(ALUout_WB),.I2(mem_data_WB),.I3(mulr
es_WB),
        .I4(divres_WB),.I5(PC_wb_WB),.I6(32'd0),.I7(32'd0),.o(wt_data_WB));
```

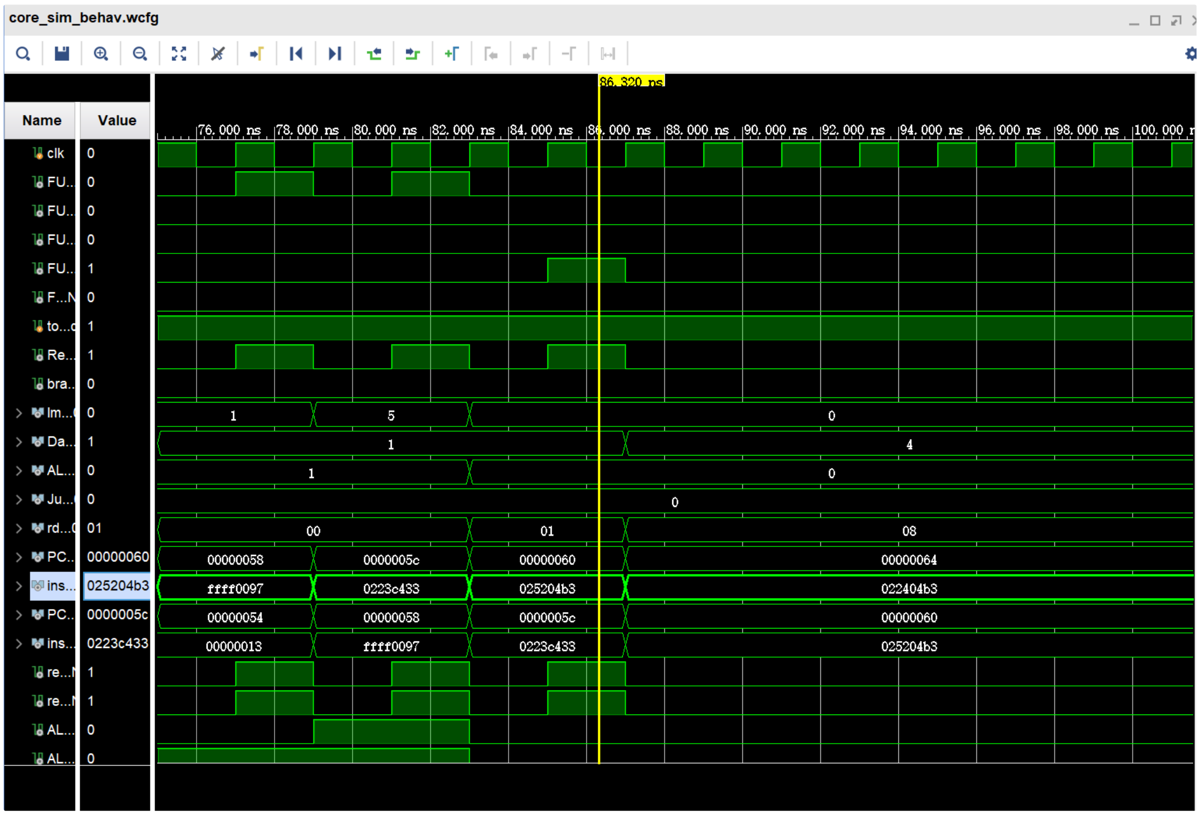
III. Simulation and Analysis



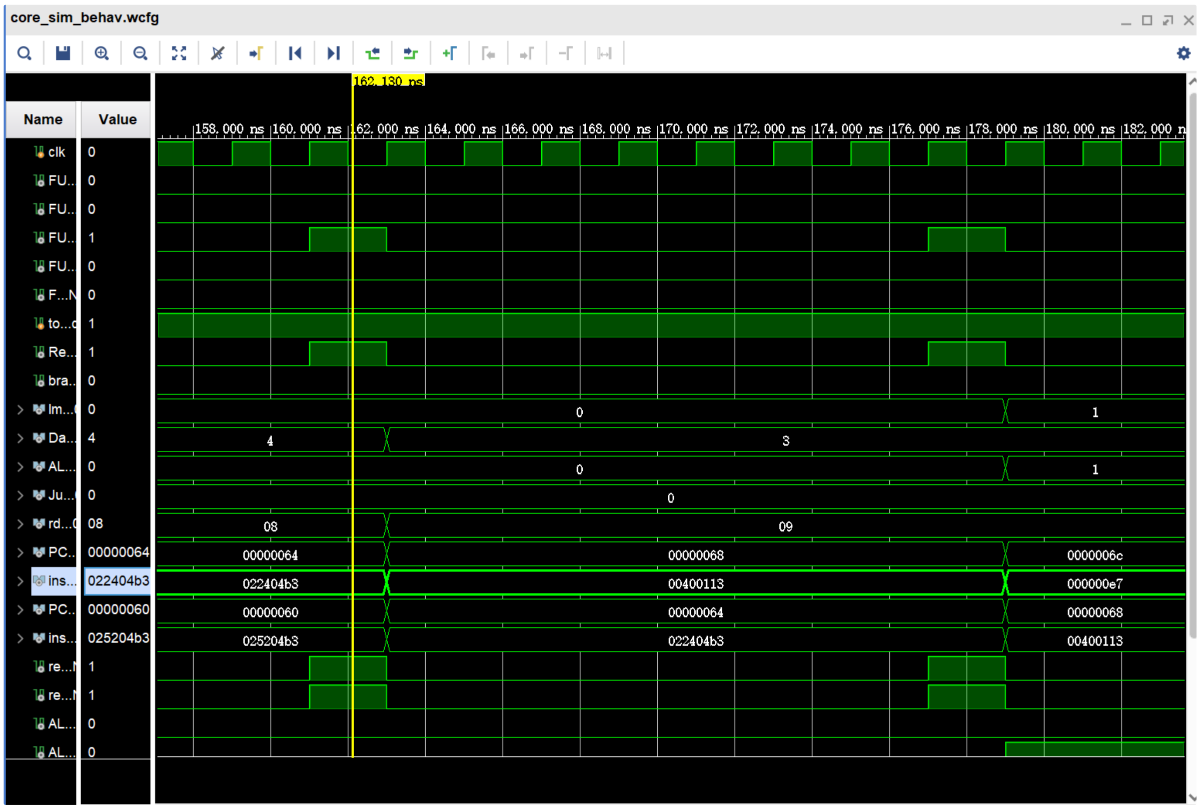
The figure below shows the simulation results for the `load` instruction. We can see that the operation stalls for two clock cycles and continues the pipeline after the third clock cycle, correctly writing the result back.



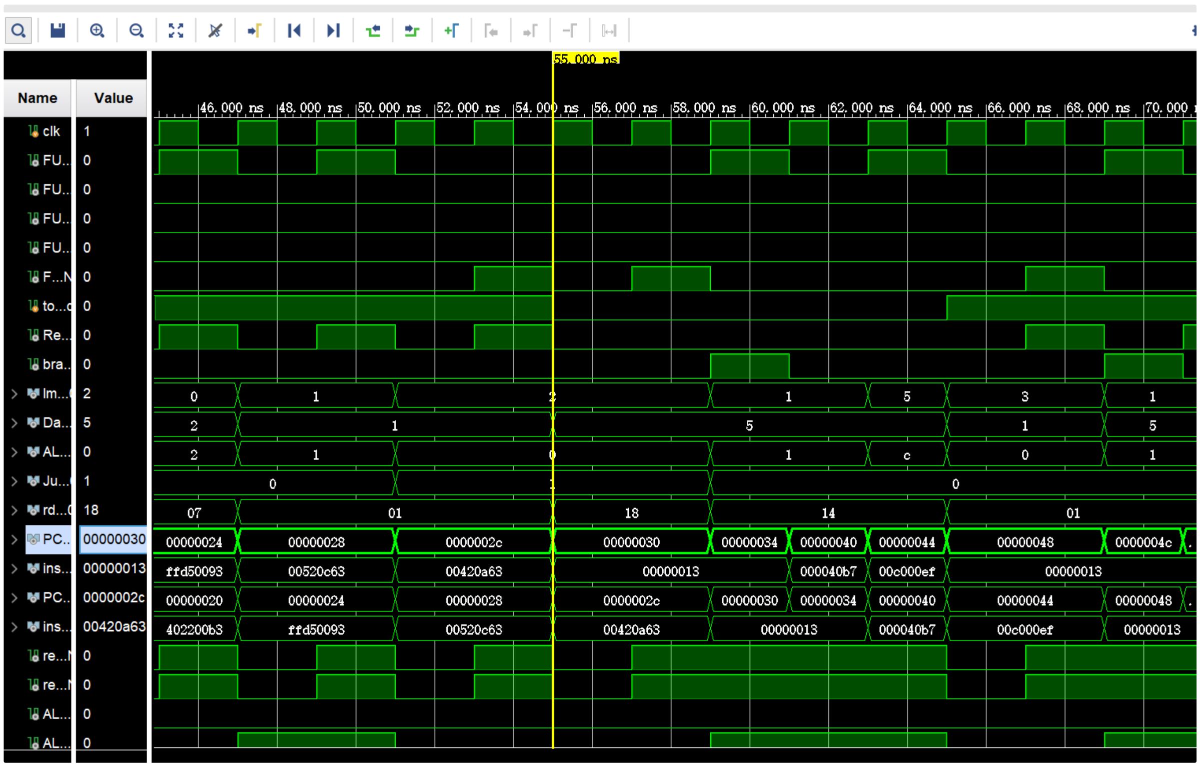
During the ALU operation, the `FU_ALU_EN` signal is first set to 1. The execution takes one clock cycle. Afterward, it is set to 0, and the `RegWrite` signal is set to 1 to write the result back to the register. At this point, `reg_IF_EN` and `reg_ID_EN` are set to 1, indicating that the pipeline continues to execute.



The number of stall during the division instruction is correct.



The number of stall during the multiplication instruction is correct.



Name	Value
clk	1
FU...	0
FU...	0
FU...	0
FU...	0
F...N	0
to...d	0
Re...	0
bra...	0
Im...	2
Da...	5
AL...	0
Ju...	1
rd...	18
PC...	00000030
ins...	00000013
PC...	0000002c
ins...	00420a63
re...I	0
re...I	0
AL...	0
AL...	0

During the jump operation, the `FU_jump_EN` signal is first set to 1 to perform the jump comparison. If a jump is needed, the `branch_ctrl` signal is set to 1, and the PC is selected accordingly. When both `Jump` and `finish` signals become 1, the jump is completed.