Lab 7 – Pipelined Architecture

Pipelining does not reduce ***time-to-completion*** for an instruction, rather it increases the ***throughput*** of the processor. Multiple different operations (for different instructions) are performed simultaneously, using different hardware resources. Using pipelining, allows us to run the entire hardware at higher operating frequency, which effectively improves the system throughput.

From Single Cycle to Pipelined Architecture

We modify our single cycle implementation to 3-stage pipelined architecture. For that purpose we decompose the single cycle processor to the following three stages.

1) Fetch
2) Decode and Execute
3) Memory and Writeback

Specifically, the pipelined datapath is formed by splitting the single-cycle datapath into three stages, where each pair of consecutive stages is separated by pipeline registers. For instance, to introduce the pipeline stage between fetch phase and decode & execute phase, two registers (namely PC register and Instruction register) are required as can be seen from the following Figure 7.1.

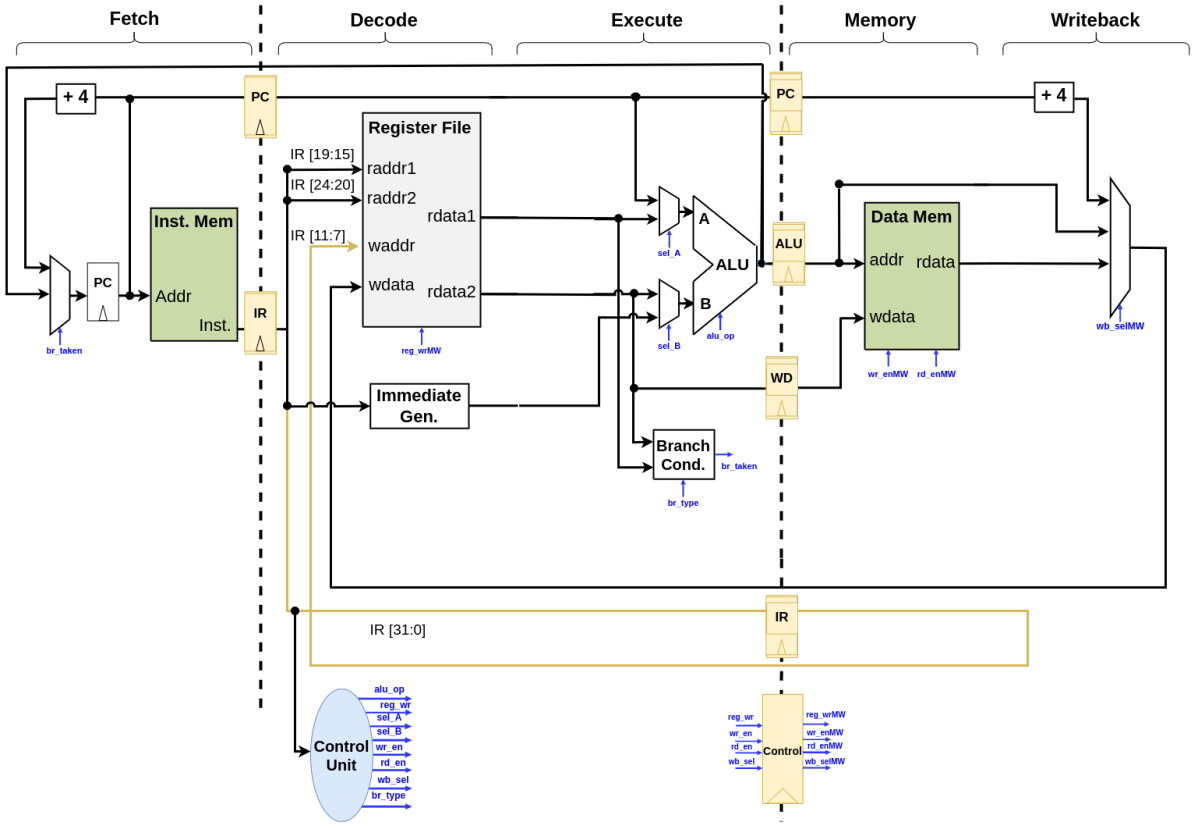

Figure 7.1. Pipelined processor microarchitecture.

Listing 7.1 illustrates the configurable pipeline stage implementation at the top level.

```
// Fetch <-----> Decode pipeline/nopipeline
`ifdef IF2ID_PIPELINE_STAGE
  type_if2id_data_s                    if2id_data_pipe_ff;

  always_ff @(posedge clk) begin
      if (rst_n) begin
          if2id_data_pipe_ff <= '0;
      end else begin
          if2id_data_pipe_ff <= if2id_data;
      end
  end
`endif // IF2ID_PIPELINE_STAGE

// Instruction Decode module instantiation
decode decode_module (
    .rst_n              (rst_n),
    .clk                (clk),

    // ID module interface signals
`ifdef IF2ID_PIPELINE_STAGE
    .if2id_data_i       (if2id_data_pipe_ff),
`else
    .if2id_data_i       (if2id_data),
`endif
    .id2if_fb_rdy_i     (id2if_rdy ),
    .id2exe_ctrl_o      (id2exe_ctrl),
    .id2exe_data_o      (id2exe_data),
    .wb2id_fb_i         (wb2id_fb)
);
```

Listing 7.1. Configurable pipeline stage implementation for fetch and decode phase (Code segment from pipeline top module).

Tasks

- Implement a pipeline stage between execute and memory phases.
- Test a simple assembly program which has no data dependency between its instructions and verify your implementation.

Lab 8 – Pipelined Architecture (Resolving Hazards)

In the previous lab, the single cycle RISC-V processor was converted to a pipelined processor by the help of pipelining registers. The pipelined processor is going to handle multiple instructions concurrently and due to dependency of the result of an instruction on another. These hazards can be classified as data or control hazards. Data hazards take place when an instruction tries to read a register that has not been updated by the previous instructions. On the other hand, the control hazards take place when the decision of fetching the next instruction has not been during the decode stages. The control hazards in case of jumps and taken branches. This is due to the fact that when jump/branch is resolved in the execution phase, the subsequent instruction is being fetched simultaneously.

Resolving Data Hazards

In the case of the three stage pipeline, some data hazards can be resolved by forwarding the result of the Memory-Writeback stage to the Decode-Execute stage which is performed by adding forwarding multiplexers. Forwarding is used when the destination register in the Memory-Writeback stage matches either of the source registers in the Decode-Execute stage. This leads to the addition of two forwarding multiplexers and a forwarding unit which takes the whole instruction in the two pipeline stages as the inputs and the selection of the two muxes becomes the outputs. This can be illustrated by Figure 8.1.

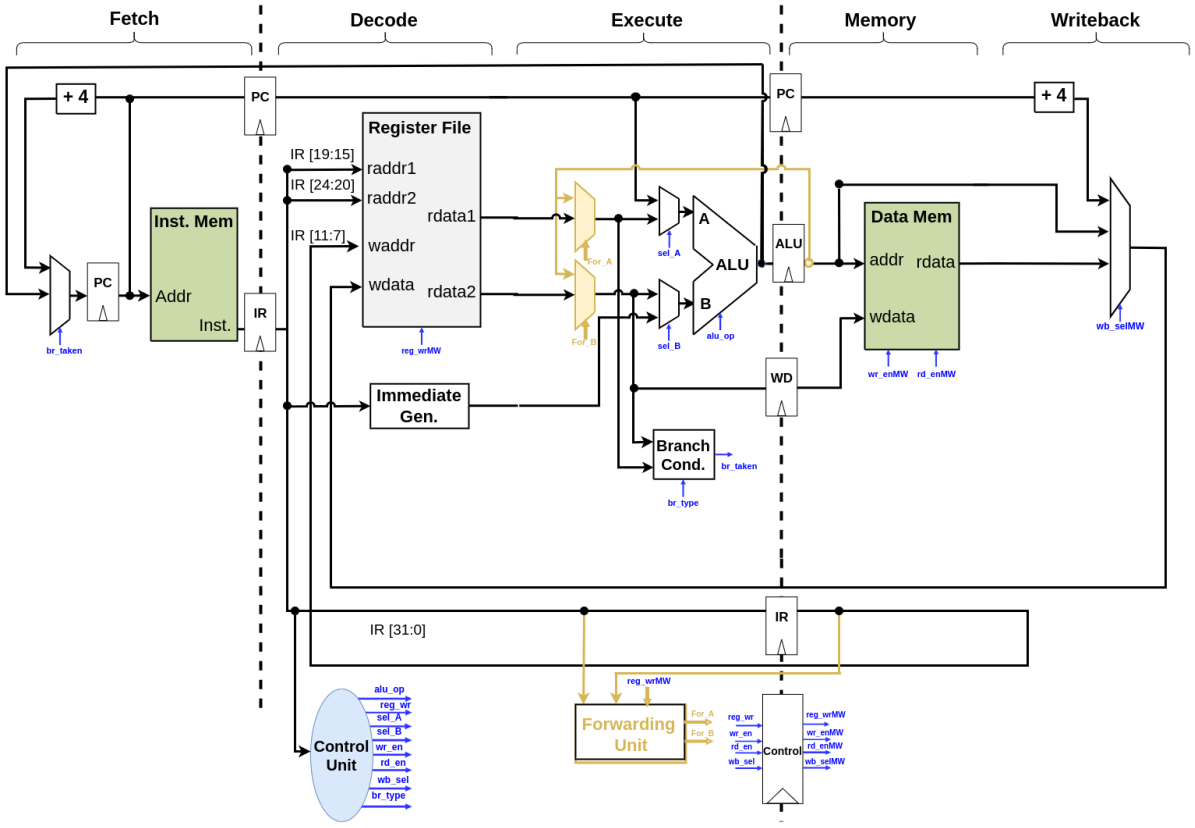

Figure 8.1. Pipelined processor microarchitecture with forwarding.

Listing 8.1 illustrates the implementation of the forwarding module.

```
// Check the validity of the source operands from EXE stage
assign rs1_valid = |exe2fwd.rs1_addr;
assign rs2_valid = |exe2fwd.rs2_addr;

// Hazard detection
assign lsu2rs1_hazard = ((exe2fwd.rs1_addr == lsu2fwd.rd_addr) & lsu2fwd.rd_wr_req) &
rs1_valid;
assign lsu2rs2_hazard = ((exe2fwd.rs2_addr == lsu2fwd.rd_addr) & lsu2fwd.rd_wr_req) &
rs2_valid;

// Generate the forwarding signals
assign fwd2exe.fwd_lsu_rs1 = lsu2rs1_hazard;
assign fwd2exe.fwd_lsu_rs2 = lsu2rs2_hazard;
```

Listing 8.1. Hazard Detection and Forwarding.

Forwarding is not sufficient in case of load instructions which can have multi-cycle latency due to which the results can not be forwarded. The only solution left would be to stall the pipeline until the result has been written to the register file. When a stage is stalled, all the previous stages must also be stalled in order to avoid instruction loss. For this purpose, we add the stalling capability to the forwarding to make it the forward stall unit. This adds the stall signals to all the pipeline registers which has been illustrated in Figure 8.2.

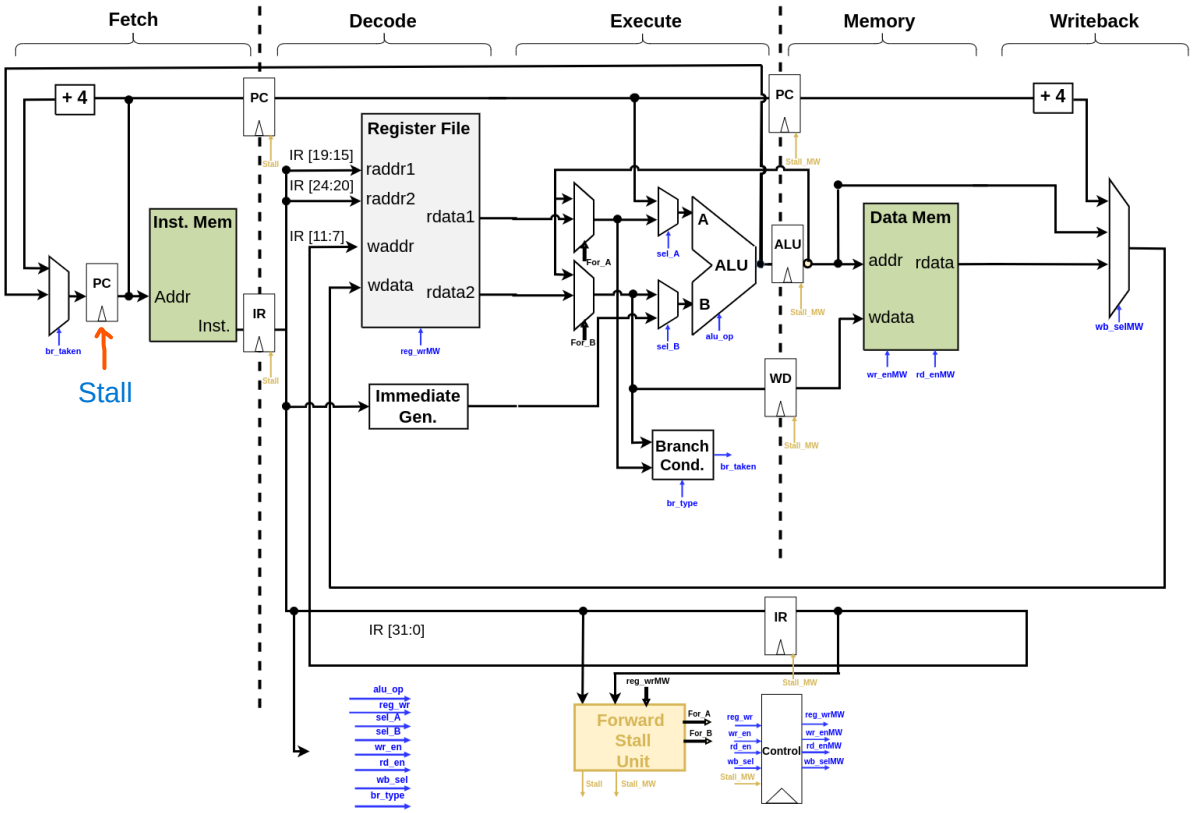

Figure 8.2. Pipelined processor microarchitecture with stalling.

Resolving Control Hazards

For taken branches as well as jumps the following instruction (which has been fetched) should not be executed. Rather it should be flushed from the pipeline, while the program counter is updated to the new address. For this purpose we need to flush the Decode-Execute stage which is done by setting the instruction pipeline register between the Fetch stage and the Decode-Execute to **nop**. For this purpose, we need to modify our forward stall module to add the br_taken flag as its input and the flush signal as the output. These changes can be observed in Figure 8.3.

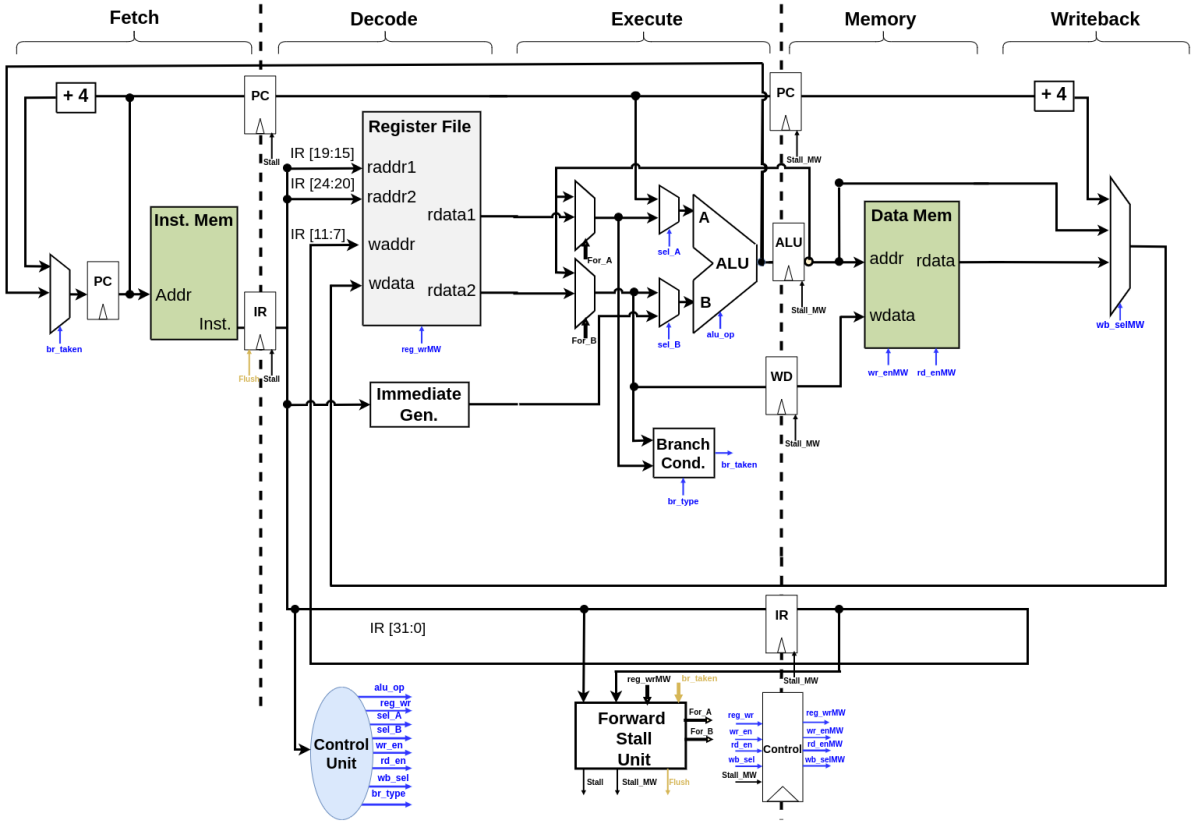

Figure 8.3. Pipelined processor microarchitecture with flushing.

Listings 8.2 and 8.3 illustrate the implementation for PC updating and fetch stage flushing.

```
// PC update state machine
always_ff @(posedge clk) begin
    if (rst_n) begin
        pc_ff <= '0;
    end else begin
        pc_ff <= pc_next;
    end
end

assign pc_next = exe2if_fb.jump_br_taken ? exe2if_fb.alu_pc
               : id2if_fb_rdy            ? (pc_ff + 32'd4)
               : pc_ff;
```

Listing 8.2. PC update state machine incorporating jump/branch related PC updating.

```
`ifdef IF2ID_PIPELINE_STAGE
    assign if2id_data.instr = exe2if_fb.jump_br_taken
                            ? `INSTR_NOP       // Insert NOP for jump or branch taken
                            : imem2if_rdata_i;
`else
    assign if2id_data.instr = imem2if_rdata_i;
`endif
```

Listing 8.3. Instruction flushing during fetch phase for jump/branch related control hazard.

Tasks

- Implement the proposed stall/forwarding/flush strategy to resolve as many hazards as possible and implement the proposed strategy.
- Write an assembly program to test some of these hazards and verify the implementation.