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.

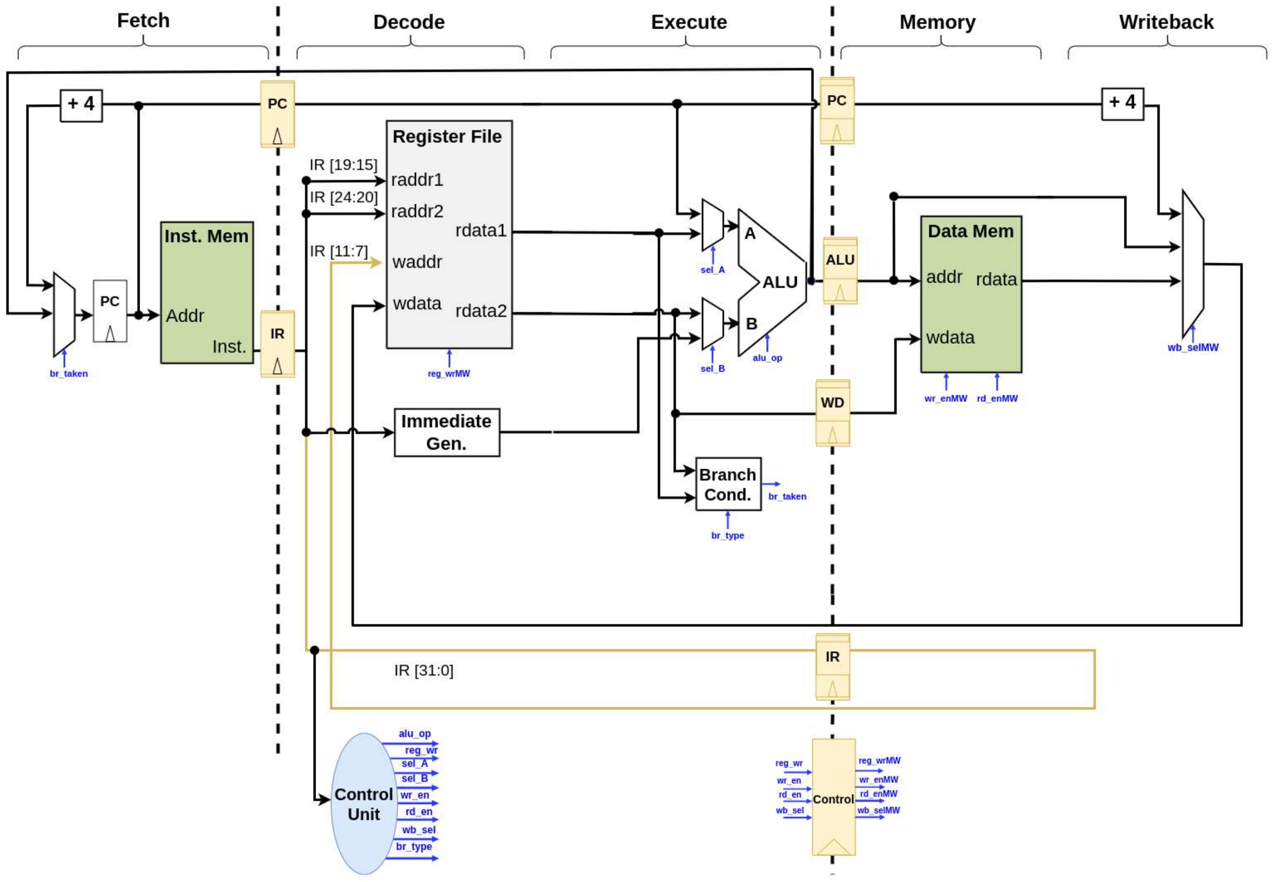

Figure 7.1. Pipelined processor microarchitecture.

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.

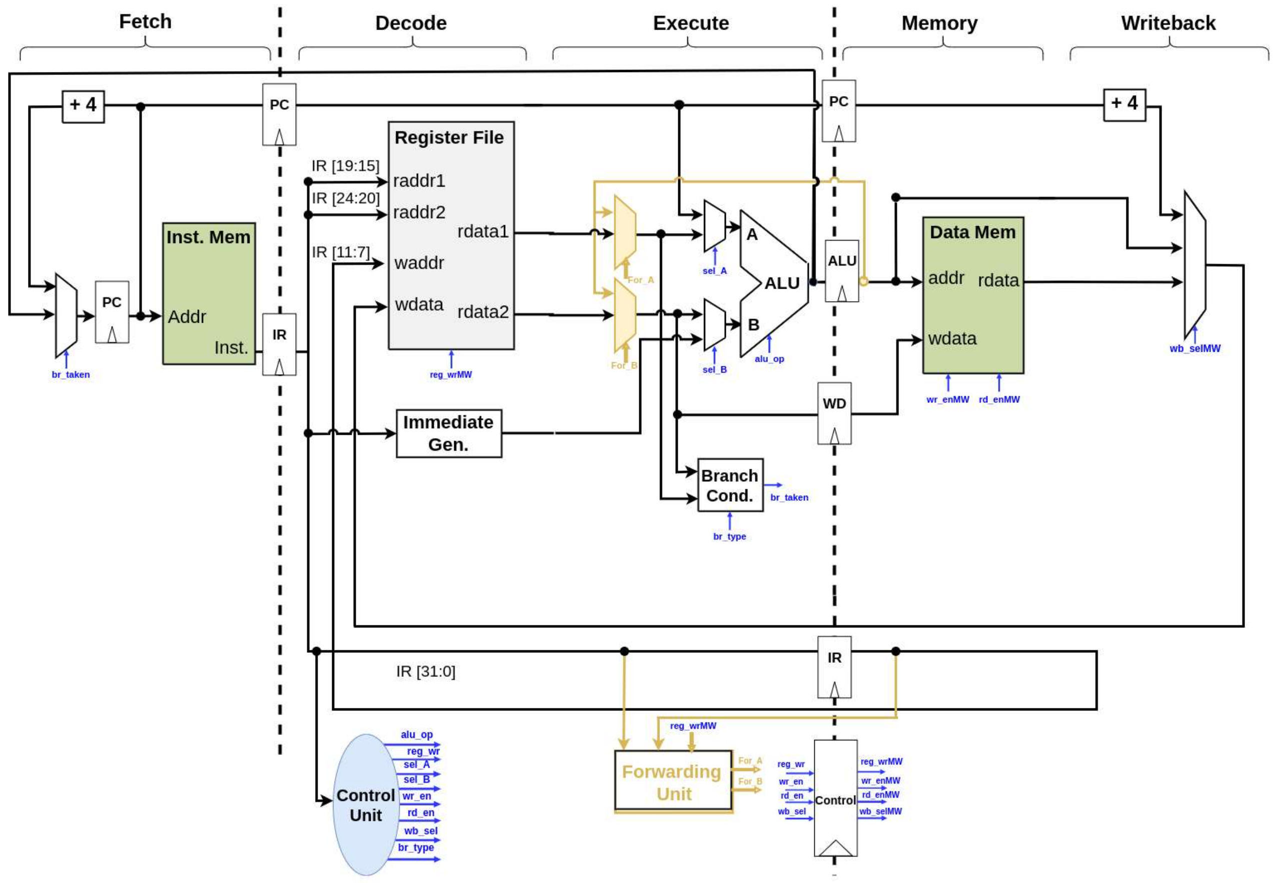

Figure 8.1. Pipelined processor microarchitecture with forwarding.

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.

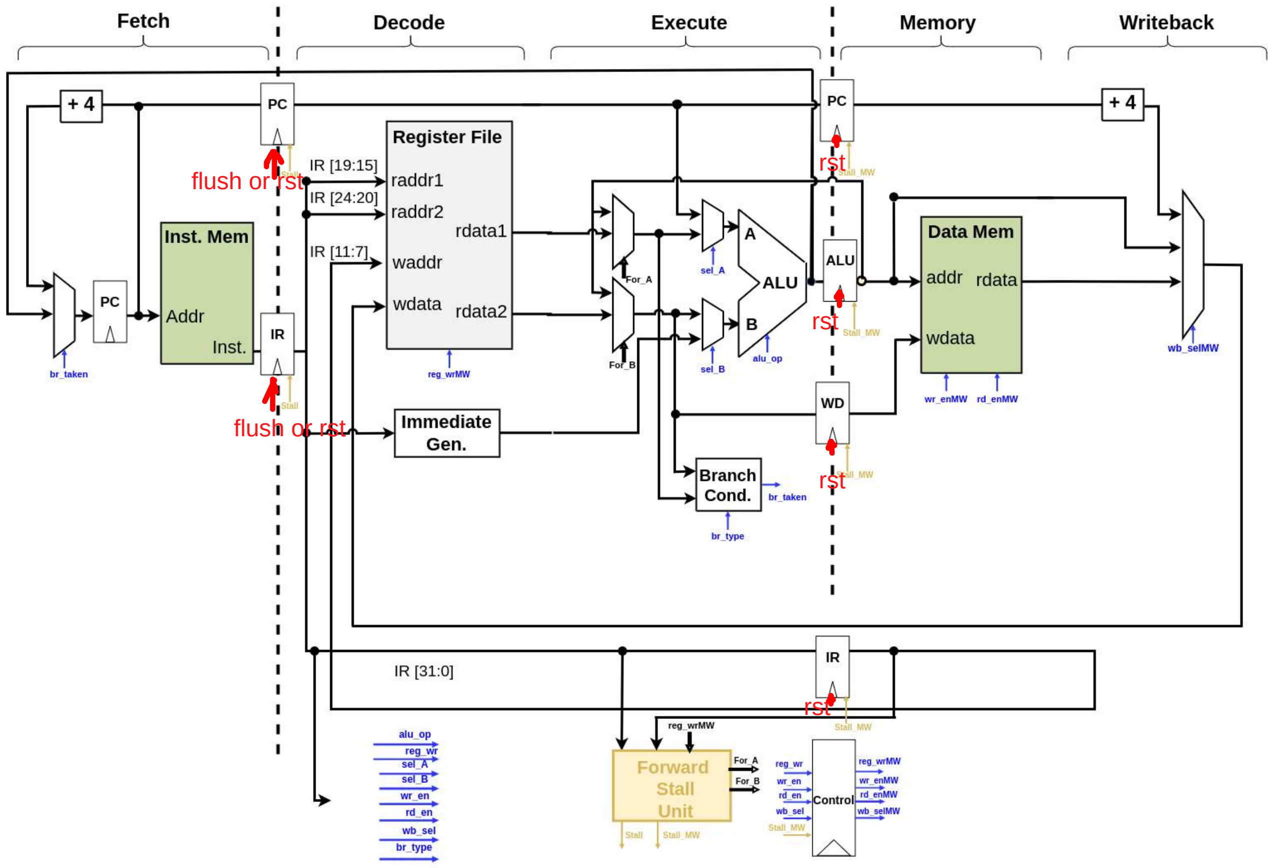

Figure 8.2. Pipelined processor microarchitecture with stalling.

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.

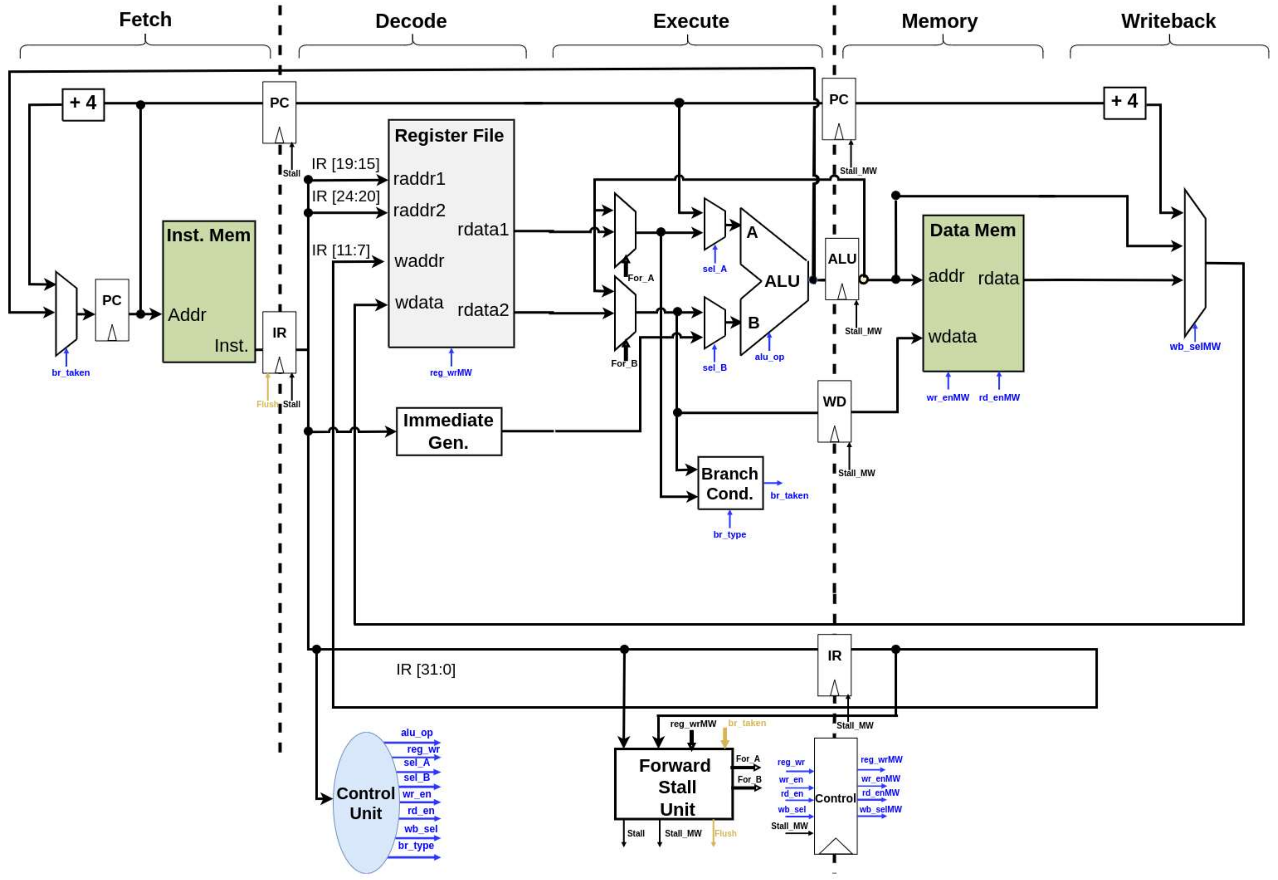

Figure 8.3. Pipelined processor microarchitecture with flushing.

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.

Lab 9 – CSR Support

In this lab, we are going to support privileged architecture in our pipelined processor. For this purpose, we are going to partially support the machine mode of the RISC-V specification in our processor. This will involve adding support for some new instructions in our datapath. We are also going to create a new register file which is going to contain the machine mode CSR registers which can be accessed by these new instructions.

CSR Register File

First, we are going to create a new registerfile which will contain our CSR registers. For this lab, we are only going to implement **`mip, mie, mstatus, mcause, mtvec`** and **`mepc`** in our register file. These registers have their 12-bit address defined as part of the RISC-V specification. The register file will have the address of the CSR register, the value of PC during the Memory-Writeback stage, the CSR register write control, the CSR register read control, the interrupt/exception pins and the CSR register write data at its input. The register file will have the CSR read data and the exception PC at its output. This can be illustrated by Figure 9.1.

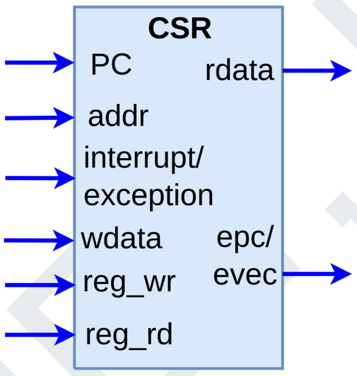

Figure 9.1. CSR Register File.

Listing 9.1 illustrates the implementation of the CSR Register file read and write operations. In the illustrated code we can see that the read and write operation will depend on the 12-bit address of the CSR registers. For checking the values address of the CSR registers refer to Table 2.5 of the RISC-V privileged specifications manual.

```
// CSR read operation
always_comb begin
   csr_rdata  = '0;
   if(exe2csr_ctrl.csr_reg_rd) begin
       case (exe2csr_data.csr_addr)
           CSR_ADDR_MSTATUS        : csr_rdata    = csr_mstatus_ff;
           CSR_ADDR_MIE            : csr_rdata    = csr_mie_ff;
           ...
           CSR_ADDR_MEPC           : csr_rdata    = csr_mepc_ff;
       endcase // exu2csr_data.csr_addr
   end
end
```

```
// CSR write operation
always_comb begin
   csr_mstatus_wr_flag          = 1'b0;
   csr_mie_wr_flag              = 1'b0;
   ...
   csr_mepc_wr_flag             = 1'b0;
   if (exe2csr_ctrl.csr_wr_req) begin
       case (exe2csr_data.csr_addr)
           CSR_ADDR_MSTATUS         : csr_mstatus_wr_flag  = 1'b1;
           CSR_ADDR_MIE             : csr_mie_wr_flag      = 1'b1;
           ...
           CSR_ADDR_MEPC            : csr_mepc_wr_flag     = 1'b1;
       endcase // exu2csr_data.csr_addr
   end // exe2csr_ctrl.csr_wr_req
end

// Update the mip (machine interrupt pending) CSR
always_ff @(negedge rst_n, posedge clk) begin
   if (~rst_n) begin
       csr_mip_ff <= {`XLEN{1'b0}};
   end else if (csr_mip_wr_flag) begin
       csr_mip_ff <= csr_wdata;
   end
end
...
// Update the mtvec CSR
always_ff @(negedge rst_n, posedge clk) begin
   if (~rst_n) begin
       csr_mtvec_ff <= {`XLEN{1'b0}};
   end else if (csr_mtvec_wr_flag) begin
       csr_mtvec_ff <= csr_wdata;
   end
end
```

Listing 9.1. CSR Register File Read and Write operations.

Tasks

- Implement the CSR register file while complying with the specifications manual and simulate it to check the read and write operation.

Implementing the CSR Instructions

In this lab, we are going to implement the ***`CSRRW`*** and ***`mret`*** instructions in our pipeline. The `CSRRW` is the CSR Read Write instruction which reads the old value of the CSR register and saves it in the destination register. Then the value of the source register is written to the CSR register. Figure 9.2 illustrates the instruction format for the CSR instruction.

31 20	15	12	7	0
csr	rs1	func3	rd	opcode

Figure 9.2. CSRRW Instruction Format.

Implementation of CSRRW

For implementing these instructions, we will be required to integrate the CSR register file in our datapath. We will also modify our controller for adding two new control signals for the read and write operation of the CSR register file. The address of the CSR register file will be given by the immediate value generator and the data to be written to the CSR register will be given by the output of the rs1 forwarding mux. The read data output of the CSR register file will be connected to the writeback mux. These changes have been illustrated in Figure 9.3.

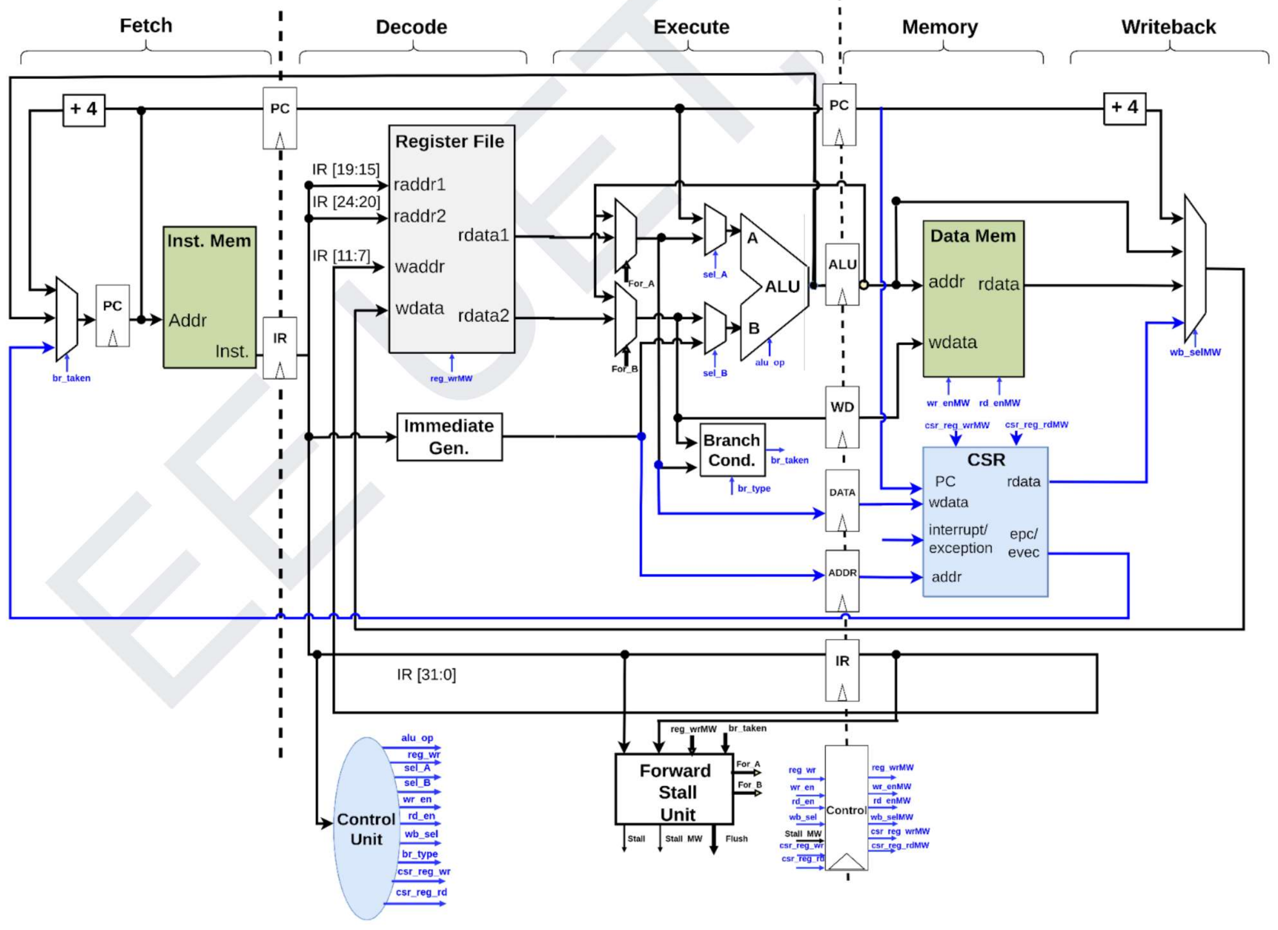

Figure 9.3. Integration of CSRRW instruction in the Datapath.

Tasks

- Integrate the CSR register file and implement the ***CSRRW*** instruction in your processor.
- Write a simple assembly code to test whether the CSR instruction is working correctly or not. A sample code has been provided in Listing 9.2.

```
li x10,1
li x12,10
csrrw x11,mtvec,x10
csrrw x13,mtvec,x12
```

Listing 9.2. Sample CSR Instruction Test assembly

Implementation of MRET

Now we will also modify our controller for adding a new multiplexor and new control signals for detecting the ***mret*** instruction. The CSR register file receives this control signal from the pipeline register. In case the ***mret*** instruction is received by the CSR register file the value of **mepc** register will be loaded in the **epc/evec** output of the register file and **epc_taken** flag of the mux will be asserted.

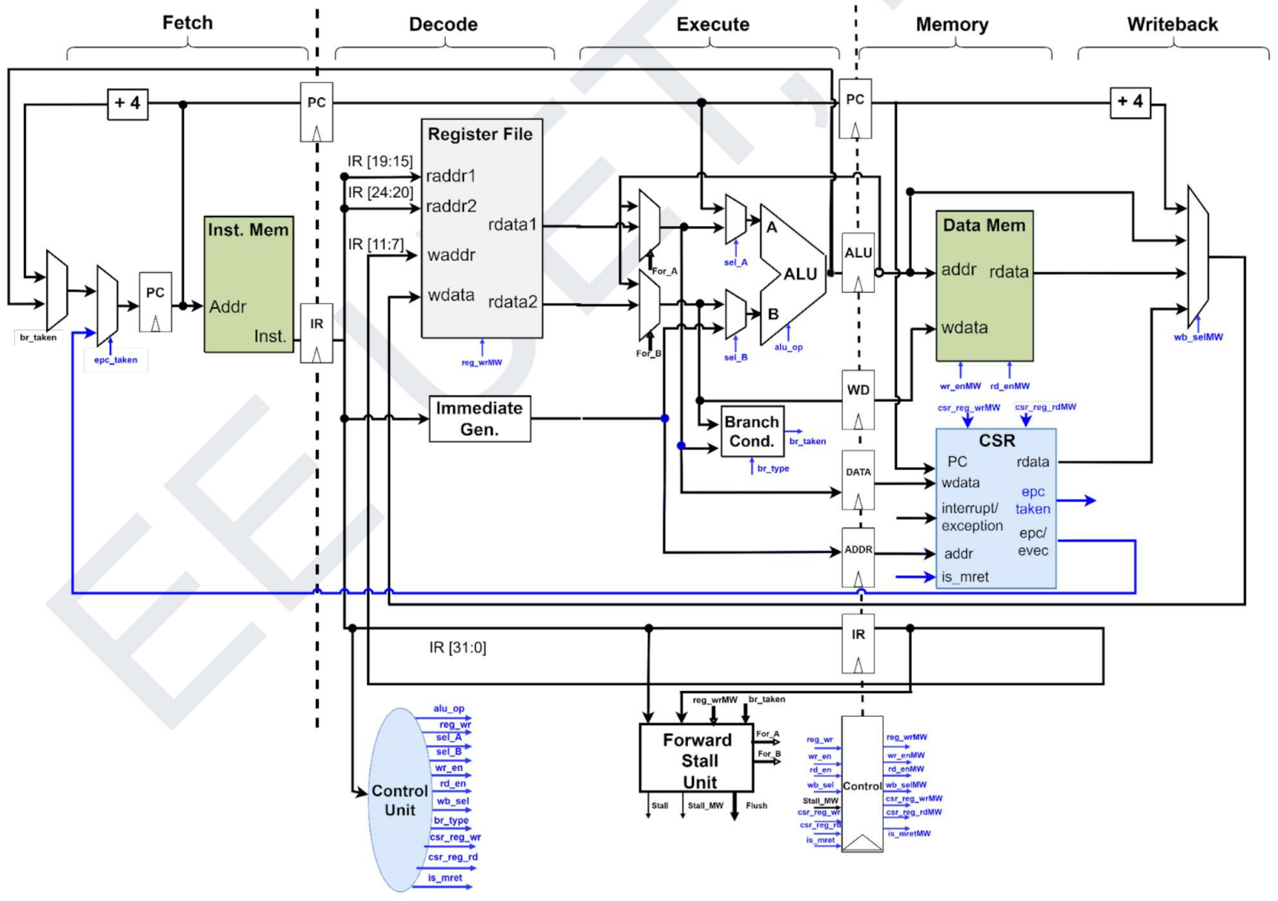

Figure 9.4. Integration of MRET instruction in the Datapath.

Tasks

- Implement the `MRET` instruction in your processor.

Interrupt Handling

Whenever an interrupt arrives, the interrupt will be registered at the positive edge of the clock in the corresponding bit of the **mip** register based on the source of the interrupt. If the corresponding bit of the **mie** register is high and the MIE bit of the **mstatus** register is high then we say that the Exception/interrupt has occurred and the value of PC in Memory-Writeback stage will be saved in the **mepc** register.

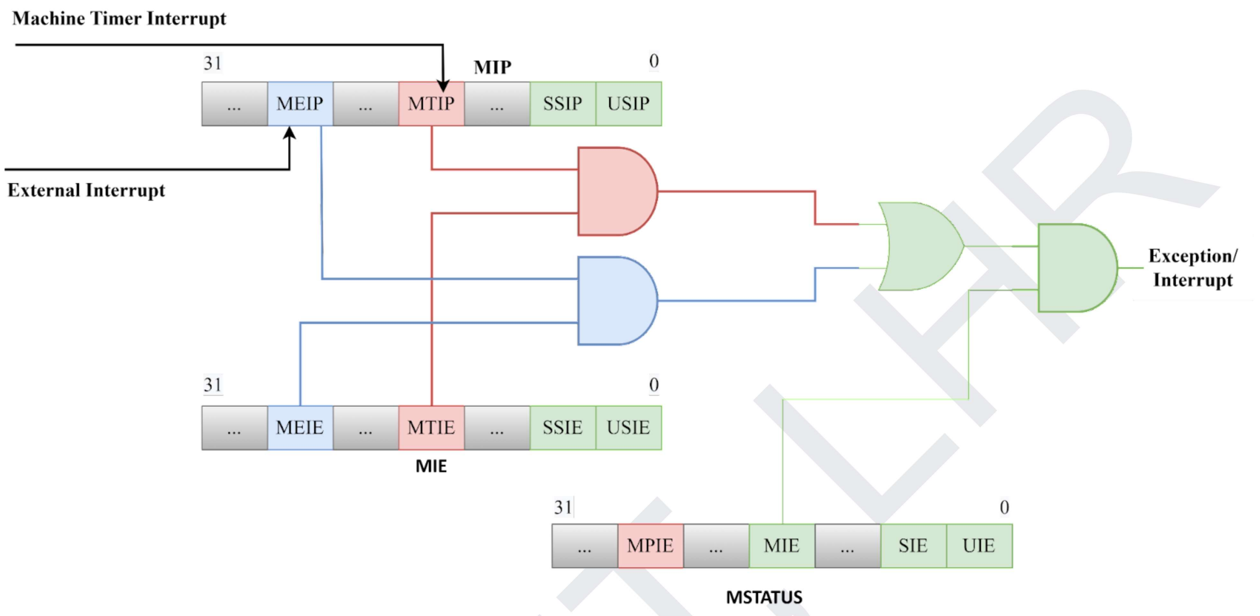

Figure 9.5. Interrupt Handling in CSR Register File.

When the Exception/interrupt has occurred **epc_taken** flag in datapath will be asserted and the cause of the interrupt will be saved in the **mcause** register by the hardware. Based on the mode of the **mtvec** register, the **epc/evec** address will be calculated as the ***base_address*** and ***base_address+cause*4*** in non-vector mode and vector mode respectively. Please note that the **mie, mstatus** and **mtvec** register will be configured by software using ***csrrw*** instruction.

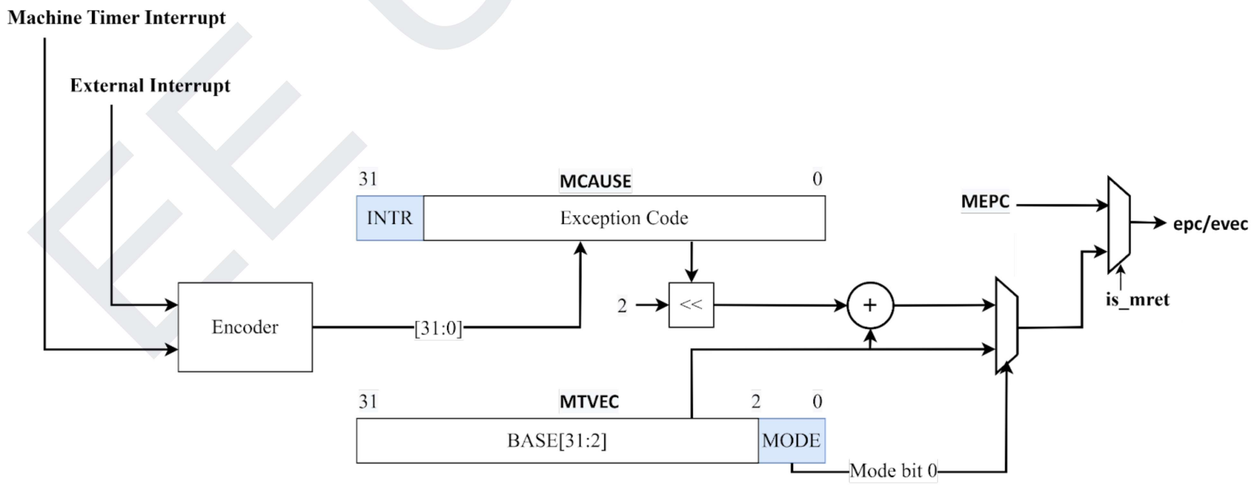

Figure 9.6. Address Calculation in CSR Register File.

Tasks

- Modify the CSR register file for handling single interrupt. Generate the interrupts using testbenches.
- Write a simple assembly code to test whether the processor is handling interrupt. Consult the RISC-V specifications for configuring the CSR registers. A sample code is shown in Listing 9.3.

```
j main
j interrupt_handler

main:
    li x11,(calculate your own value)
    li x12,(calculate your own value)
    li x13,(calculate your own value)
    csrrw x0,mie,x11
    csrrw x0,mstatus,x12
    csrrw x0,mtvec,x13

loop:
    addi x14,x14,1
    j loop

interrupt_handler:
    csrrw x0,mie,x0
    li x2,0xFFFFFFFF
    xor x3,x3,x2
    csrrw x0,mie,x11
    mret
```

Listing 9.3. Sample Interrupt Handling Test assembly