Pipeline Implementation:

The single cycle RISC-V 32I is converted to a 3-staged pipeline. Following are the 3 stages:

- Fetch
- Decode & Execute
- Memory & write back

The pipelining is done to increase the throughput of the processor, but the time of completion remains same. Pipelining is instantiated by including 2 new modules that contains the required Flip Flops that act as pipeline registers. For the first module "Fetch to Decode-Execute" Flip flops for following:

- PC -> PC_DE
- Instruction -> Instruction_DE

Where, PC is coming from program counter and instruction from instruction memory. For the second module, following flip flops are inserted:

- PC_DE -> PC_MW
- Instruction_DE -> Instruction_MW
- Alu_result -> Alu_result_MW
- rdata2 -> rdata2_MW
- wdata_sel -> wdata_sel_MW
- rfwrite -> rfwrite_MW

Where, Alu_result is the output signal of ALU, rdata2 is second output of register file, wdata_sel and rfwrite are signals coming from controller.

Resolving Hazards:

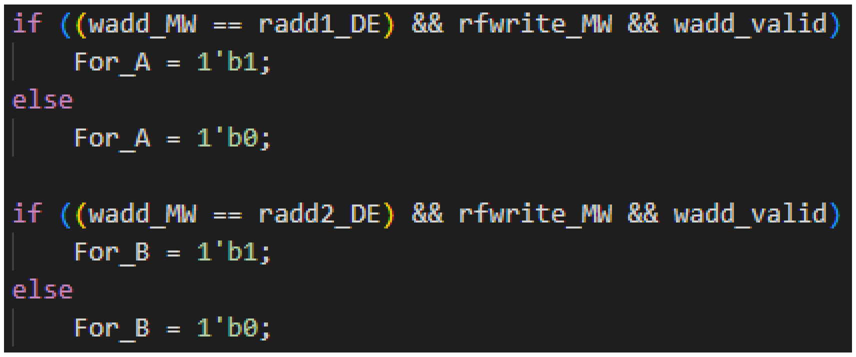
As pipeline is going to handle multiple instructions concurrently, there can be dependency of a result of an instruction on another. Following hazards can take place:

Data Hazards:

Data Hazards take place when an instruction tries to read a register that has not been updated by previous instruction. This type of hazards occurs in R, I type, especially Load instructions. Following are the solutions implemented in the model to prevent data hazards:

Forwarding:

It is implemented by forwarding the result of Memory-Writeback stage (output of Alu_result pipeline register) to Decoder-Execute stage which is performed by adding forwarding multiplexers. These muxes are instantiated in front of the output rdata1 and rdata2 of register file. Forwarding is used when the destination register in MW stage matches either of the source registers in DE stage. This also leads to the addition of a forwarding unit where the Instruction_DE and Instruction_MW are compared along with rfwrite_MW signal to see whether the forwarding is needed or not. If the condition comes true, then the high signal is sent to both forwarding multiplexers. The required condition is as follows:



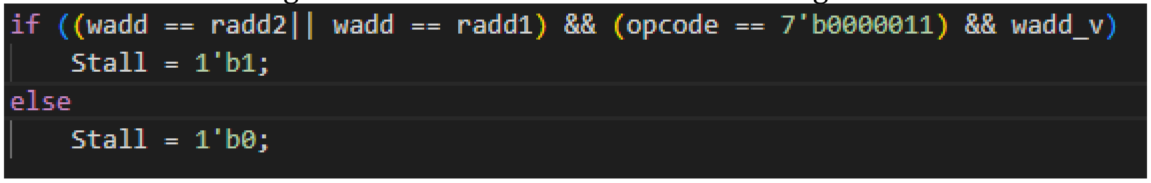
```
if ((wadd_MW == radd1_DE) && rfwrite_MW && wadd_valid)
    For_A = 1'b1;
else
    For_A = 1'b0;

if ((wadd_MW == radd2_DE) && rfwrite_MW && wadd_valid)
    For_B = 1'b1;
else
    For_B = 1'b0;
```

Figure 1: Forwarding Condition

Stalling:

Through forwarding, all the data hazards are removed, except that by Load Instruction. As the result in Load comes in MW stage, so the dependent instruction has to wait for 2 cycles to have the updated value. By forwarding, the stall for 1 cycle is reduced. For load instructions to work properly, we introduce the stalling unit that stalls the incoming instructions for 1 cycle so that load instruction could store the updated value in the required register. Note that it only happens in the case when the instructions next to load is dependent. The following condition must be satisfied for stalling:



```
if ((wadd == radd2|| wadd == radd1) && (opcode == 7'b0000011) && wadd_v)
    Stall = 1'b1;
else
    Stall = 1'b0;
```

Figure 2: Stalling Condition

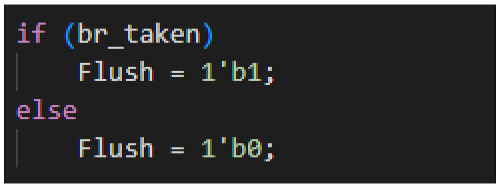
If the stalling condition is satisfied, then the stall signal is sent to Program Counter. If the signal is high, then the next PC address is set to the previous one.

Control Hazards:

Control hazards take place when decision of fetching the next instruction has not been done during decode stages. This type of hazards occurs in Branch and Jump Instructions. Following solution is implemented to deal with control hazards.

Flushing:

For branches and jumps, following instructions are flushed from the pipeline, while program counter is updated to new address. DE stage is flushed by setting the instruction pipeline register between F and DE stage to NOP. For that, the condition is checked in the forwarding stall unit. If condition becomes true, then the high signal is sent to the above register which then set the instruction at that stage to NOP. The following condition needs to be satisfied:



```
if (br_taken)
    Flush = 1'b1;
else
    Flush = 1'b0;
```

Figure 3: Flushing Condition

CSR Support:

The machine mode of RISC-V is partially supported. A new register file is created that contain CSR registers. There are total 6 of them:

- mip

- mie
- mstatus
- mcause
- mtvec
- mepc

These registers have 12-bit address defined as part of RISC-V specification. This register file have address of CSR register, value of PC during MW stage, CSR write control, CSR read control, interrupt/exception pins and CSR write data at its input. CSR read data and exception PC are regarded as outputs.

Following 2 instructions are implemented in our pipeline:

- CSRRW
- mret

CSRRW implementation:

After creating CSR register file, it is integrated in datapath. CSR read and write control signal are given to controller which decides based on opcode when to turn these signals on. Both these signals after passing through the DE to MW buffer are then given to CSR register file. The specific addresses for the 6 registers are defined as parameters in the CSR register file. Instead of getting the address of CSR register from immediate value generator, it is obtained by Instruction_MW[31:20]. Data to be written to CSR register is given by output of rdata1 forwarding mux. The read data output is connected to the writeback mux.

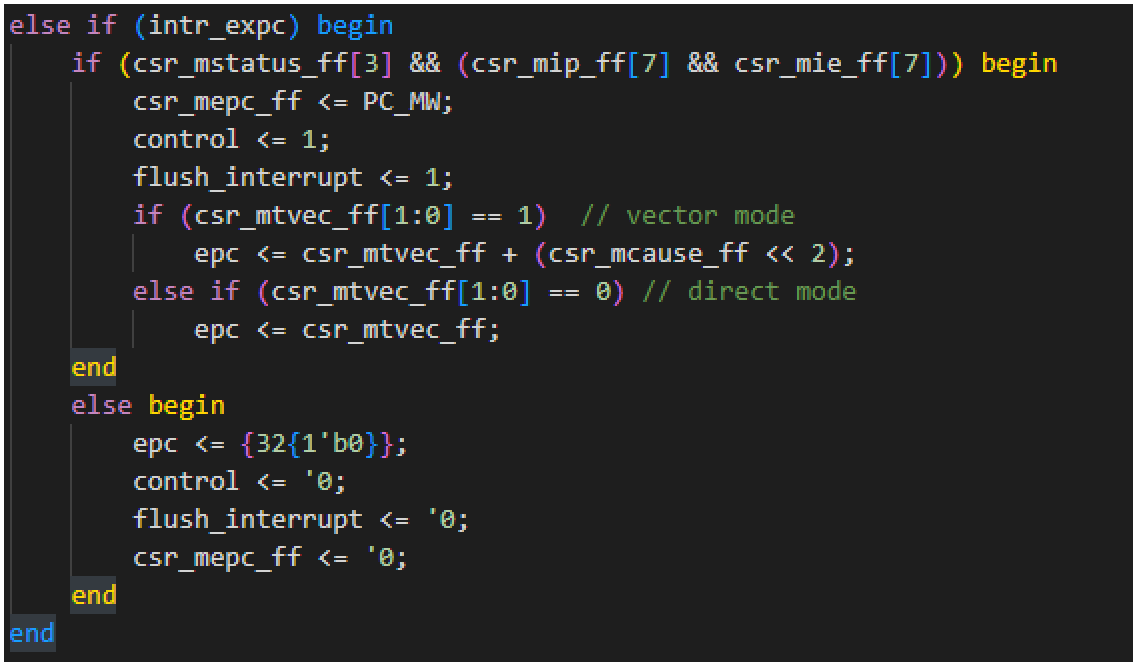
Mret implementation:

Signal for mret instruction is given by controller i.e. is_mret which decides on the basis of opcode and func3. This control signal after passing through DE to MW buffer is fed to CSR register file. When this signal is high, the value of mepc register is loaded into epc register and control flag is set to high. Both epc and control are then fed to program counter. In PC, when control is high (giving it priority) then the value of PC is set equal to epc.

Interrupt Handling:

In this module, only Timer Interrupt is being handled. For timer interrupt, a timer counter is created that causes interrupt after every few fix cycles. There is no need for an encoder as there is only one kind of interrupt. Also, for this interrupt, we will be working in direct mode, but vector mode is also implemented.

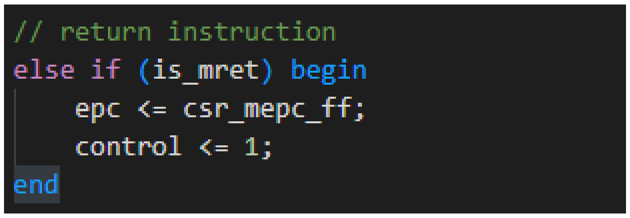
The appropriate bit in the MIP and MCAUSE register goes high whenever an interrupt occurs. To determine if the interrupt has been enabled or not, the appropriate bits of the MSTATUS and MIE registers are checked. When an interrupt occurs, MEPC stores the value of PC at the Memory-Write Back stage if the interrupt is enabled and enabled. The base address of the vector table kept in MTVEC is kept in EPC. To instruct Program counter to switch to EPC and begin processing the interrupt, CSR flag (control) is raised. CSR flush_interrupt flag goes high to flush the fetched instruction. This implementation can be seen as follows:



```
else if (intr_expc) begin
    if (csr_mstatus_ff[3] && (csr_mip_ff[7] && csr_mie_ff[7])) begin
        csr_mepc_ff <= PC_MW;
        control <= 1;
        flush_interrupt <= 1;
        if (csr_mtvec_ff[1:0] == 1)  // vector mode
            epc <= csr_mtvec_ff + (csr_mcause_ff << 2);
        else if (csr_mtvec_ff[1:0] == 0) // direct mode
            epc <= csr_mtvec_ff;
    end
    else begin
        epc <= {32{1'b0}};
        control <= '0;
        flush_interrupt <= '0;
        csr_mepc_ff <= '0;
    end
end
```

Figure 4: Interrupt Handling

Once the interruption has been satisfactorily managed, running instructions must be resumed as usual. As a result, the mret instruction is carried out. When the interrupt occurs and the CSR Flag (control) is set to high, this instruction updates epc to the value stored in mepc. Program Counter returns to the regular execution of instructions in this manner. This can be seen as follows:



```
// return instruction
else if (is_mret) begin
    epc <= csr_mepc_ff;
    control <= 1;
end
```

Figure 5; mret implementation