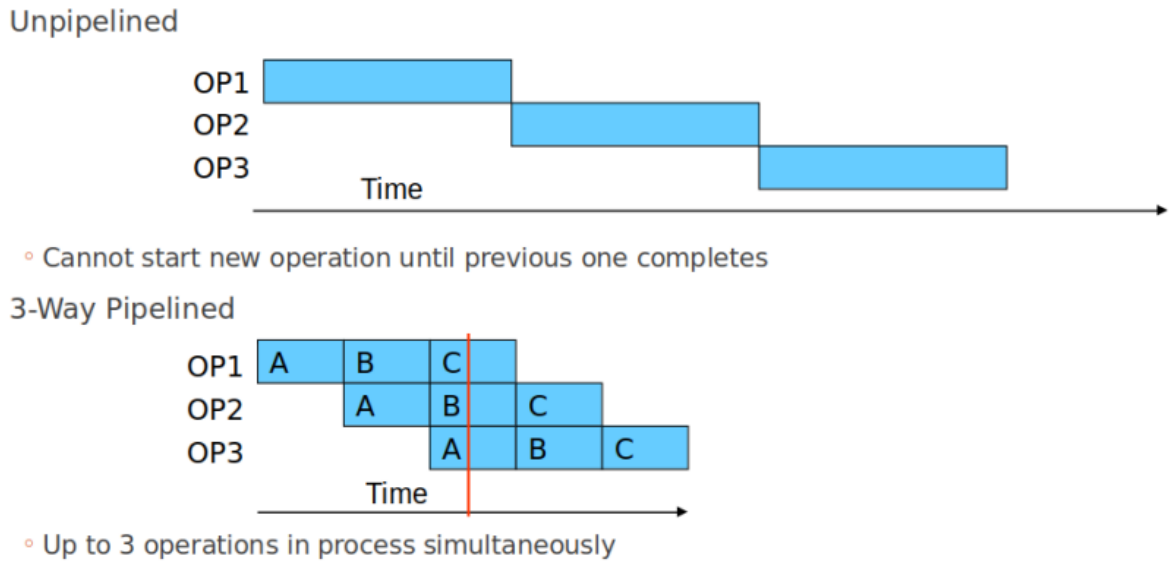
IPA Project Report

> Member 1: Soumil Gupta - 2022102035
> Member 2: Bhaskar Bhatt - 2022102003

Overview

- The aim of the project is to develop a processor architecture design based on the Y86 ISA using Verilog.
- This report describes the design details of the various stages of the processor architecture.

The following is the basic difference between a sequential vs. pipeline implementation of the processor.



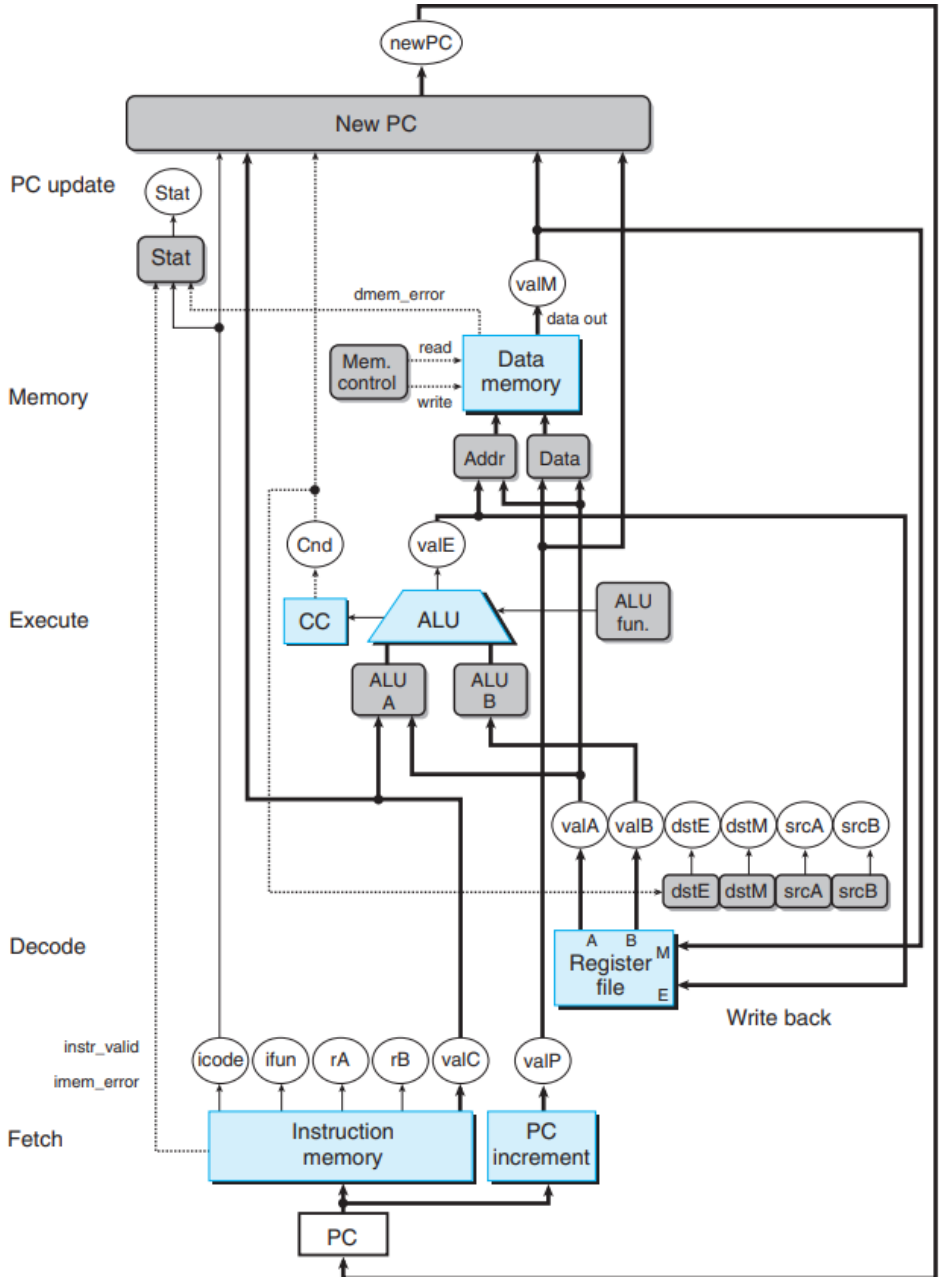
OP1, OP2, OP3 have to be implemented one after the other

The Pipelined architecture might increase the delay of the processing of one instruction, but allows for an increased throughput. With SEQ, an instruction is executed only after that which is preceding it has finished executing. In other words, the instructions follow a sequence, hence the name. On the other hand,

with PIPE, an instruction is divided into a fixed number of stages.

But there can be few problems which can be encountered while pipeline architecture is in use such as data dependencies, data hazards, etc. which needs to be handled carefully.

Sequential Processor Design (SEQ)



Hardware structure of SEQ, a sequential implementation

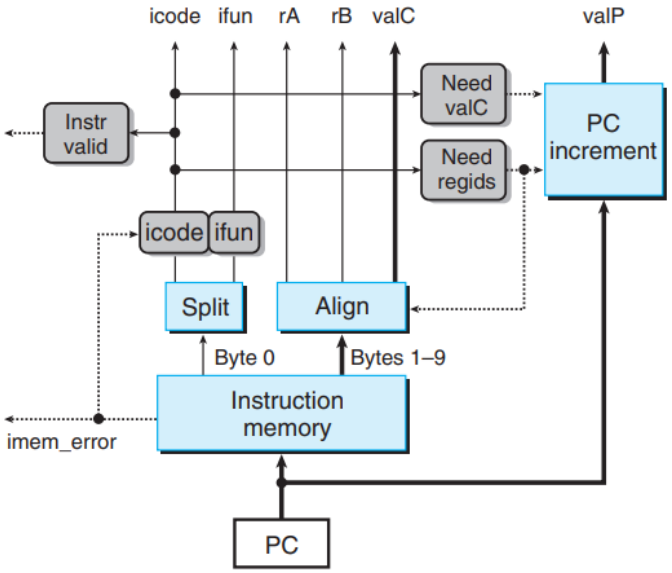
- The SEQ Design includes 6 stages:
  1. Fetch
  2. Decode
  3. Execute
  4. Memory
  5. Write Back
  6. PC Update

Stage	HALT	NOP	CMOV	IRMOVQ
Fch	icode:ifun ← M_1[PC] valP ← PC + 1	icode:ifun ← M_1[PC] valP ← PC + 1	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC + 10
Dec			valA ← R[rA]	
Exe	cpu.stat = HLT		valE ← valA Cnd ← Cond(CC,ifun)	valE ← valC
Mem				
WB			Cnd ? R[rB] ← valE	R[rB] ← valE
PC	PC ← 0	PC ← valP	PC ← valP	PC ← valP
Stage	**RMMOVQ**	**MRMOVQ**	**OPq**	**jXX**
Fch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC + 10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC + 10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC + 9
Dec	valA ← R[rA] valB ← R[rB]	 valB ← R[rB]	valA ← R[rA] valB ← R[rB]	
Exe	valE ← valB + valC	valE ← valB + valC	valE ← valB OP valA Set CC	Cnd ← Cond(CC,ifun)
Mem	M_8[valE] ← valA	valM ← M_8[valE]		
WB		R[rA] ← valM	R[rB] ← valE	
PC	PC ← valP	PC ← valP	PC ← valP	PC ← Cnd ? valC:valP
Stage	**CALL**	**RET**	**PUSHQ**	**POPQ**
Fch	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC + 9	icode:ifun ← M_1[PC] valP ← PC + 1	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2
Dec	 valB ← R[RSP]	valA ← R[RSP] valB ← R[RSP]	valA ← R[rA] valB ← R[RSP]	valA ← R[RSP] valB ← R[RSP]
Exe	valE ← valB - 8	valE ← valB + 8	valE ← valB - 8	valE ← valB + 8
Mem	M_8[valE] ← valP	valM ← M_8[valA]	M_8[valE] ← valA	valM ← M_8[valA]
WB	R[RSP] ← valE	R[RSP] ← valE	R[RSP] ← valE	R[RSP] ← valE R[rA] ← valM
PC	PC ← valC	PC ← valM	PC ← valP	PC ← valP

The entire working of above instruction stages for SEQ architecture summarized

Module Description

Fetch



SEQ fetch stage

- The fetch stage includes the instruction memory hardware unit. This unit reads 10 bytes from memory at a time, using the `PC` as the address of the first byte (byte 0).

- This byte is interpreted as the instruction byte and is split (by the unit labeled “Split”) into two 4-bit quantities. The control logic blocks labeled `icode` and `ifun` then compute the instruction and function codes as equaling either the values read from memory or, in the event that the instruction address is not valid (as indicated by the signal `imem_error`), the values corresponding to a `nop` instruction.

- Based on the value of `icode`, we can compute three 1-bit signals (shown as dashed lines in the diagram above):

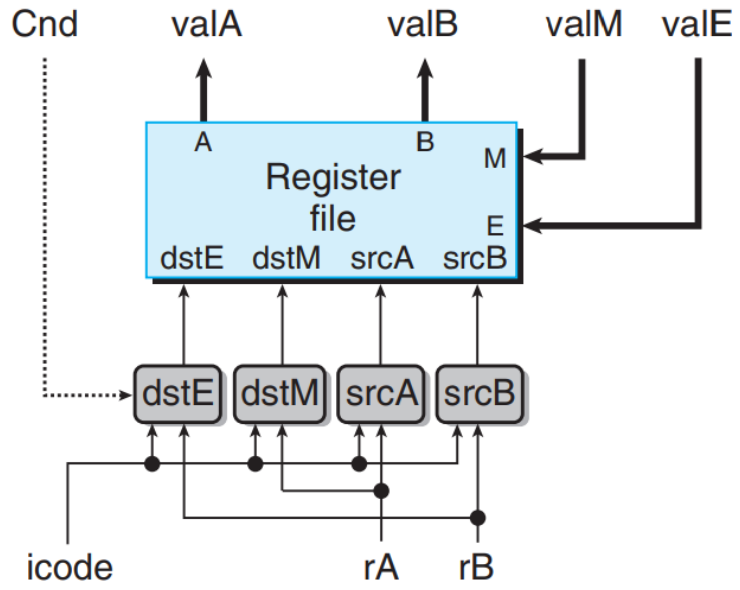
  1. `instr_valid:` Does this byte correspond to a legal Y86-64 instruction? This signal is used to detect an illegal instruction.
  2. `need_regids:` Does this instruction include a register specifier byte?
  3. `need_valC:` Does this instruction include a constant word?

- Although, in our implementation, we are only checking for `instr_valid` and `imem_error`. For `need_regids` and `need_valC` we have hardcoded for every instruction, so there is no need for these two. We set the value of `valP` directly this way.

- The signals `instr_valid` and `imem_error` (generated when the instruction address is out of bounds) are used to generate the status code in the memory stage.

- The PC incrementer hardware unit generates the signal `valP`, based on the current value of the PC, and the two signals `need_regids` and `need_valC`

Decode & Write Back Stage



SEQ decode and write-back stage

- In our implementation, since both the stages access the register file, we are combining write-back and decode stage in the same module.

- The register file has four ports. It supports up to two simultaneous reads (on ports A and B) and two simultaneous writes (on ports E and M).

- Each port has both an address connection and a data connection, where the address connection is a register ID, and the data connection is a set of 64 wires serving as either an output word (for a read port) or an input word (for a write port) of the register file.

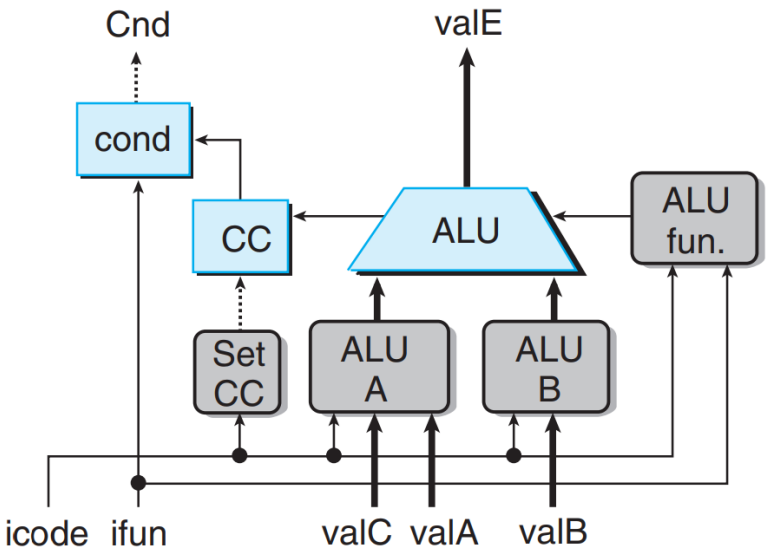
- The two read ports have address inputs `srcA` and `srcB`, while the two write ports have address inputs `dstE` and `dstM`. The special identifier `0xF` on an address port indicates that no register should be accessed.

- Register ID `srcA` indicates which register should be read to generate `valA`
- Register ID `dstE` indicates the destination register for write port E, where the computed value `valE` is stored.
- Register ID `dstM` indicates the destination register for write port M, where `valM`, the value read from memory is stored.

- From here, based upon the `icode` of the instruction we will assign values to `valA` and/or `valB`.

- **The register file:** The instruction fields are decoded to generate register identifiers for four addresses (two read and two write) used by the register file. The values read from the register file become the signals `valA` and `valB`. The two write-back values `valE` and `valM` serve as the data for the writes. We are initially assigning the registers in the register file with some value in our implementation.

Execute Stage



SEQ execute stage

- The execute stage includes the arithmetic/logic unit (ALU). This unit performs the operation add, subtract, and, or exclusive-or on inputs `aluA` and `aluB` based on the setting of the `icode` and `ifun` signal.

- Also, the condition code registers are set according to the ALU value. The condition code values are tested to determine whether a branch should be taken. Our ALU generates the three signals on which the condition codes are based—zero, sign, and overflow—every time it operates. However, we only want to set the condition codes when an `OPq` instruction is executed.

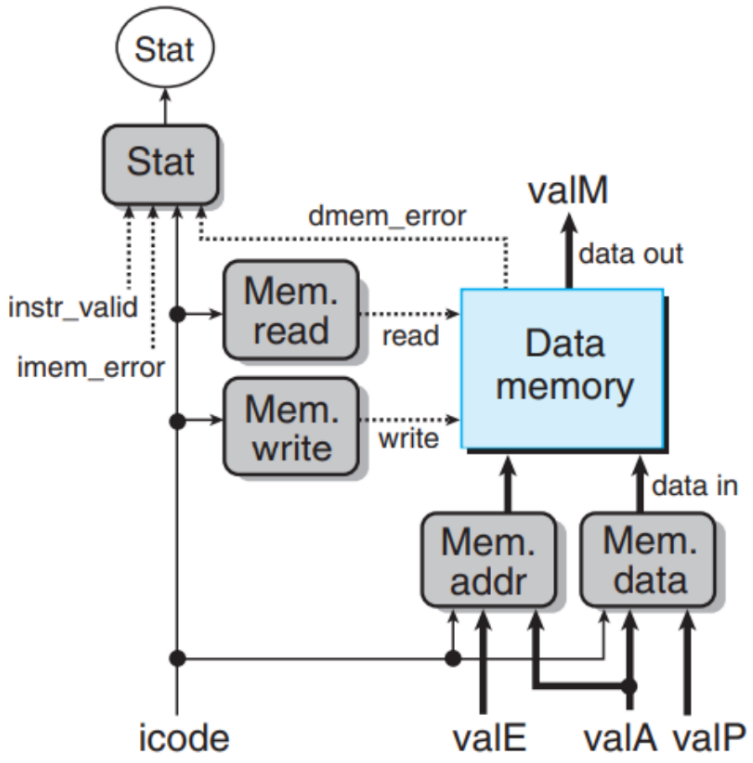
- In our implementation of the execute stage we are checking the `icode` of the instruction and based on that we are assigning values to `aluA`, `aluB` and the control which are then passed onto ALU.

- In our implementation, the control signals for the ALU are:

Control Signal	OP
00	ADD
01	SUB
10	AND
11	XOR

- For the `cmovXX` and `jXX` instruction we check the value of `ifun` to set the condition (`cnd`). The output of ALU is set to be `valE`.

Memory Stage



SEQ memory stage

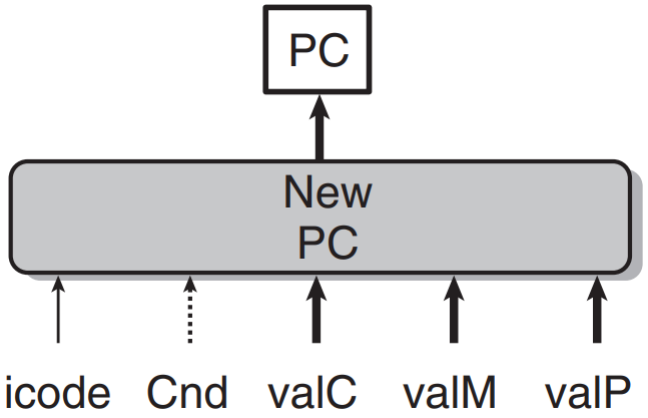
- The stage is responsible for reading and writing to memory. We declared the data memory as a register array:

```
reg [63:0] memory [0:127];
```

- Based on the `icode` we can decide whether we are required to read or write from or to memory respectively. The address for memory reads and writes is always `valE` or `valA`.

- A final function for the memory stage is to compute the status code Stat resulting from the instruction execution according to the values of `icode`, `imem_error`, and `instr_valid` generated in the fetch stage and the signal `dmem_error` generated by the data memory.

PC Update Stage



SEQ PC update stage

- The final stage in SEQ generates the new value of the program counter. This stage is only present in SEQ implementation because in PIPE implementation, we are predicting PC in the fetch stage and thus reducing the delay.

- The next value of the PC is selected from among the signals `valC`, `valM`, and `valP`, depending on the instruction code and the branch flag. The control logic for PC update is as follows:

```
word new_pc = [
        # Call.  Use instruction constant
        icode == ICALL : valC;
        # Taken branch.  Use instruction constant
        icode == IJXX && Cnd : valC;
        # Completion of RET instruction.  Use value from stack
        icode == IRET : valM;
        # Default: Use incremented PC
        1 : valP;
];
```

Logic for deciding new PC value

Supported Features for SEQ Implementation

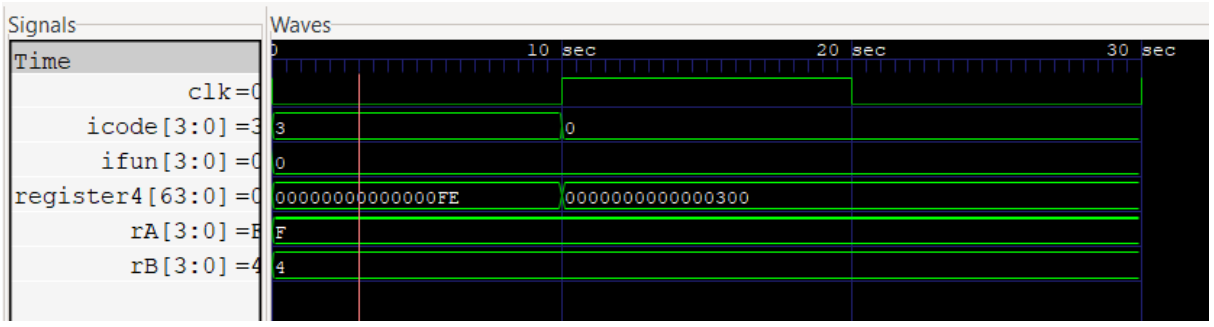
We have finished testing our processor and every instruction we have put in place. Every instruction is following through as planned, and every register, value, and status code is being updated when it should be.

We will show the waveform diagrams to demonstrate various testcases that we have prepared ourselves.

1. `irmovq:`

```
irmovq $300, %rsp
```

Result:

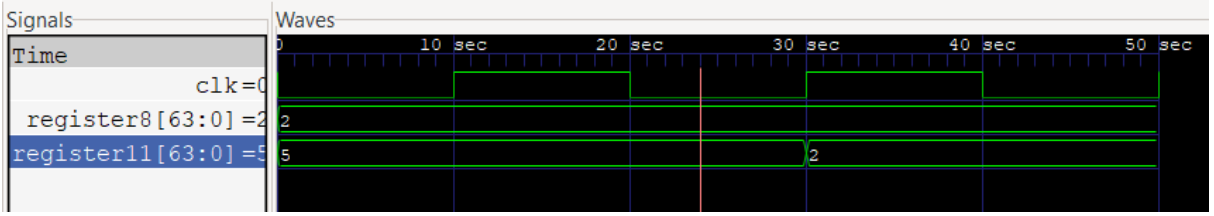


GTKWave Plot

2. `pushq & popq:` We will show the working of push and pop simultaneously. The testcase for the same is →

```
pushq %r8
popq %r11
halt
```

Result:

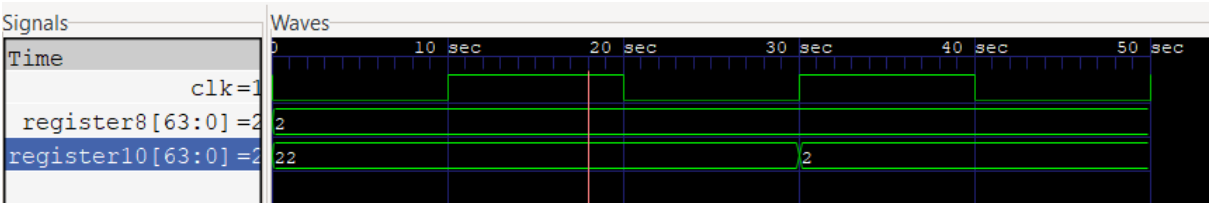


GTKWave Plot

3. `rmmovq & mrmovq:` We will show the working of rmmovq and mrmovq simultaneously. The testcase for the same is →

```
rmmovq %r8, (%r9)
mrmovq (%r9), %r10
halt
```

Result:

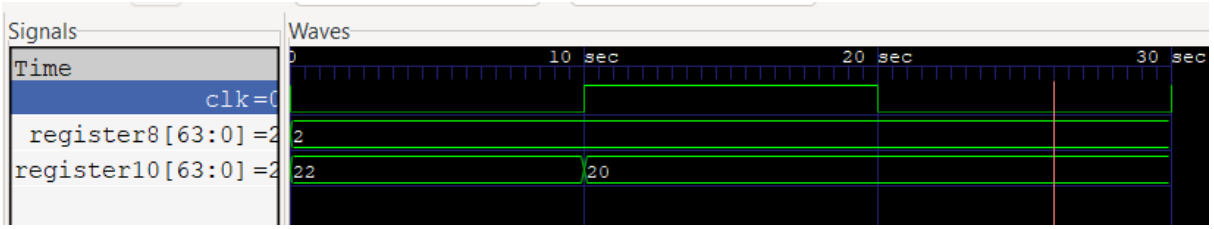


GTKWave Plot

4. `OPq:` We will demonstrate the working of the OPq using the subq operation. The testcase for the same is →

```
subq %r8, %r10
halt
```

Result:



5. `cmovXX:` We will demonstrate the working of the cmovXX instruction using cmove condition. The testcase for the same is →

```
subq %r8, %r10
cmove %r8, %r12
halt
```

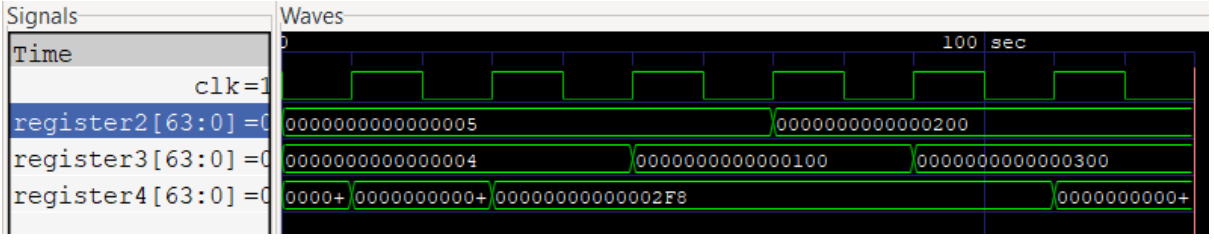
Result:

The move is not taken because the condition code criteria is not satisfied.

6. `call & ret:` We will demonstrate the working of the call and return instructions simultaneously. The testcase for the same is →

```
irmovq $0x300, %rsp
call main
halt

main: irmovq $0x100, %rbx
            irmovq $0x200, %rdx
            addq %rdx, %rbx
            ret
```

Result:



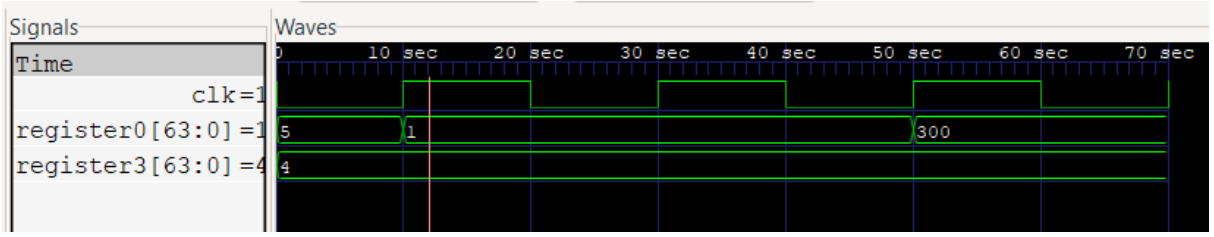
Clearly, both call and return are obeyed without any errors.

7. `jump:` We will demonstrate the working of the jXX instruction using je condition. The testcase for the same is →

```
subq %rbx, %rax
je .L1
irmovq $0x300, %rax
halt
```

```
.L1: irmovq $0x200, %rax
         halt
```

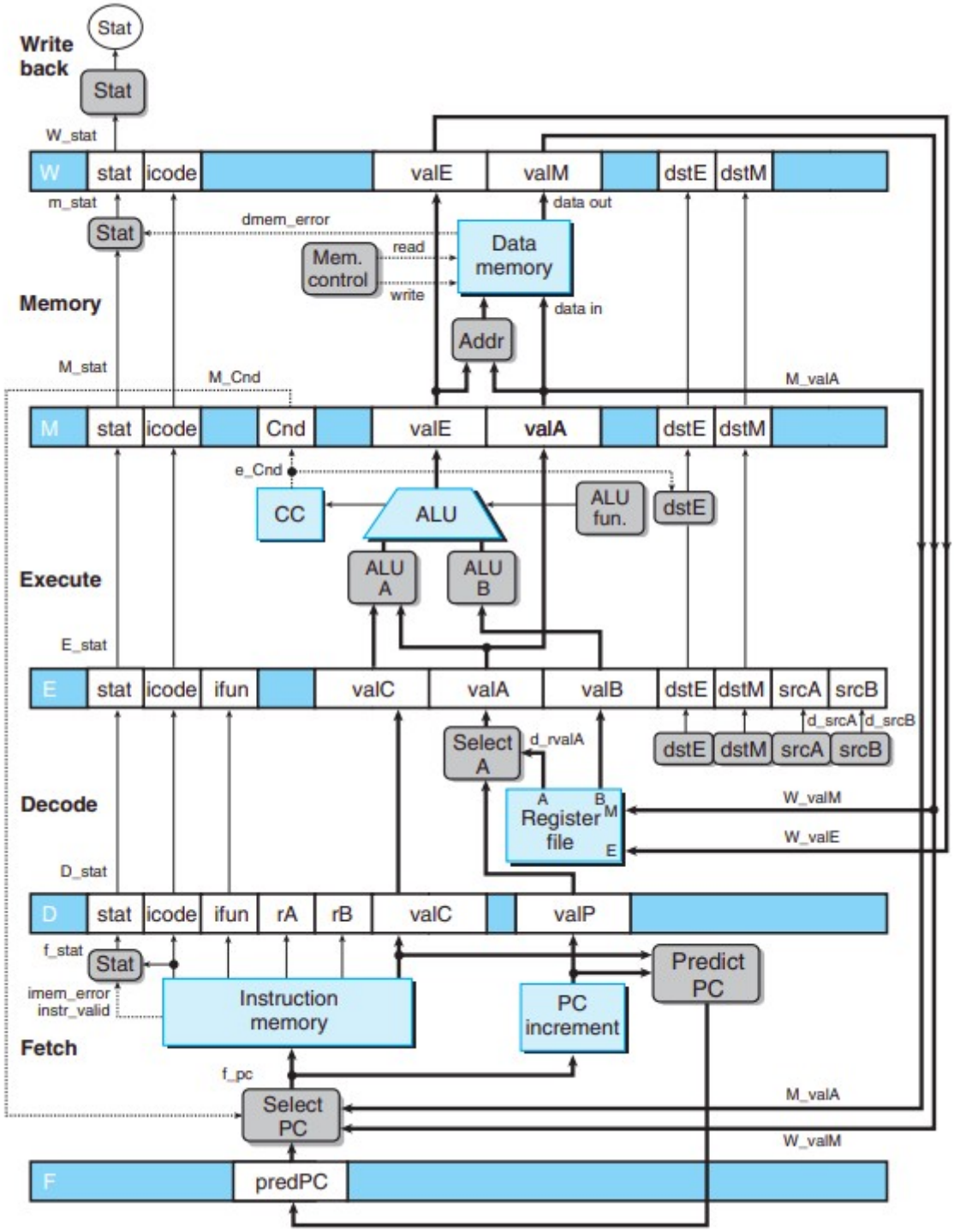
Result:



Clearly, jump is not taken because je condition is not satisfied.

Pipeline

Hardware structure of PIPE



Changes for Pipeline

Rearranging Stages:

- The PC update stage in the SEQ implementation in the last stage in the cycle of an instruction.
- For the pipelined implementation we should bring the PC update stage to the beginning of the cycle as we want to be able to continuously fetch the next instruction without having to wait for the PC update stage of the previous instruction to end had it been at the end of the cycle. This is known as circuit retiming. This changes the general sentation of the circuit without affecting its local behavior. This also allows us to balance the delays between stages in the pipelined system.
- Now the PC update stage at the beginning of the cycle can keep providing updated PC values to the fetch stage using the required values from different stages from instructions that have passed that stage.

Inserting Pipeline Registers:

- The next step to pipelining is inserting the pipeline registers.
- We know that in a pipelined implementation we rearrange some of the hardware and signals in the SEQ implementation and insert pipeline register between each stage.
- These registers stop the signals from one stage from flowing into the next stage and affecting the processing happening there.
- **F** the register inserted before the fetch stage holds a predicted value of the program counter.
- **D** sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.
- **E** sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.
- **M** sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.
- **W** sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a **ret** instruction.

Rearranging and Relabelling signals:

- In the pipelined implementation we will have all the signals of an instruction pass through every stage one by one and these will have to be names with respect to the stage it is currently in as it is not possible to

have one signal icode and have ti account for all the 5 instructions running at the same time.

- So we maintain the signal at each stage and label them with respect to the stage as f_icode,d_icode,w_icode, etc.

Processor

Defining and initialising inputs and outputs:

```
`include "fetch.v"
`include "decode.v"
`include "execute.v"
`include "memory.v"
`include "control.v"
`include "fetch_registers.v"
`include "decode_registers.v"
`include "execute_registers.v"
`include "memory_registers.v"
`include "writeback_registers.v"

module processor();

reg clk;

// Fetch I/O
reg [63:0] F_predPC = 0;
reg [2:0] Stat;
reg [3:0] W_icode = 1;
reg M_cnd = 0;

// Fetch stage output
wire [3:0] f_icode, f_ifun, f_rA, f_rB;
wire [63:0] f_valC, f_valP;
wire [63:0] f_predPC;
wire [2:0] f_stat;

// Decode I/O
reg [2:0] D_stat = 1;
reg [3:0] D_icode = 1;
reg [3:0] D_ifun = 0;
reg [3:0] D_rA = 0;
reg [3:0] D_rB = 0;
reg [63:0] D_valC = 0;
reg [63:0] D_valP = 0;
```

```
reg [3:0] W_dstM = 0;
reg [3:0] W_dstE = 0;
reg [63:0] W_valM = 0;
reg [63:0] W_valE = 0;
```

```
// Decode stage output
wire[2:0] d_stat;
wire[3:0] d_icode, d_ifun, d_srcA, d_srcB, d_dstE, d_dstM;
wire[63:0] d_valC, d_valA, d_valB;

// Execute I/O
reg [2:0] E_stat = 1;
reg [2:0] W_stat = 1;
reg [3:0] E_icode = 1;
reg [3:0] E_ifun = 0;
reg [3:0] E_dstE = 0;
reg [3:0] E_dstM = 0;
reg [3:0] E_srcA = 0;
reg [3:0] E_srcB = 0;
reg [63:0] E_valC = 0;
reg [63:0] E_valA = 0;
reg [63:0] E_valB = 0;

// Execute stage output
wire e_cnd, ZF, SF, OF;
wire [2:0] e_stat;
wire [3:0] e_icode, e_dstE, e_dstM;
wire [63:0] e_valE, e_valA;

// Memory I/O
reg [3:0] M_icode = 1;
reg [2:0] M_stat = 1;
reg [63:0] M_valA = 0;
reg [63:0] M_valE = 0;
reg [3:0] M_dstM = 0;
reg [3:0] M_dstE = 0;
```

```
// Memory stage output
wire[3:0] m_icode;
wire[2:0] m_stat;
wire[63:0] m_valE;
wire[63:0] m_valM;
wire[3:0] m_dstM;
wire[3:0] m_dstE;
```

Integration of Modules:

```
// ******** calling pipeline modules ********* //

fetch S1(
    F_predPC, M_icode, M_cnd, M_valA, W_icode, W_valM,
    f_icode, f_ifun, f_rA, f_rB, f_valC, f_valP, f_predPC, f_stat
);


decode S2(D_stat, D_icode, D_ifun, D_rA, D_rB, D_valC, D_valP, e_dstE,
        e_valE, M_dstE, M_valE, M_dstM, m_valM, W_dstM, W_valM, W_dstE, W_valE, clk,
        d_stat, d_icode, d_ifun, d_valC, d_valA, d_valB, d_dstE, d_dstM, d_srcA, d_sr
);


execute S3(
    clk, E_stat, E_icode, E_ifun, E_valC, E_valA, E_valB, E_dstE,
    E_dstM, E_srcA, E_srcB, W_stat, m_stat,
    e_stat, e_icode, e_cnd, e_valE, e_valA, e_dstE, e_dstM, ZF, SF, OF
);


memory S4( clk, M_icode, M_stat, M_valA, M_valE, M_dstM,
        M_dstE, m_icode, m_stat, m_valE, m_valM, m_dstM, m_dstE
);


// Code for Processor Status Code
// assign Stat = W_stat;


// Writing PipeLine Control Logic
wire W_stall, M_bubble, E_bubble, D_bubble, D_stall, F_stall;


control C1(W_stat, M_icode, m_stat, e_cnd, E_dstM, E_icode, d_srcA, d_srcB, D_icode,
        W_stall, M_bubble, E_bubble, D_bubble, D_stall, F_stall
        );
```

Updating the registers at positive edge of clock depending on the value of pipeline control signals of stall and bubble:

```
always @(W_stat)
begin
    Stat = W_stat;
end

// Updating F register at every positive edge of clock
always @(posedge clk)
begin
    if(!F_stall)
    begin F_predPC <= f_predPC; end
end


// Updating D register at every positive edge of clock
always @(posedge clk)
begin
    if(!D_stall)
    begin
        if (!D_bubble)
        begin
            D_stat <= f_stat;
            D_icode <= f_icode;
            D_ifun <= f_ifun;
            D_rA <= f_rA;
            D_rB <= f_rB;
            D_valC <= f_valC;
            D_valP <= f_valP;
        end

        else
        begin
            D_stat <= 1; // AOK Normal Operation
            D_icode <= 1; //basically nop
            D_ifun <= 0;
            D_rA <= 0;
            D_rB <= 0;
            D_valC <= 0;
            D_valP <= 0;
        end
    end
end
```

```
// Updating E register at every positive edge of clock
always @(posedge clk)
begin
    if(!E_bubble)
    begin
        E_stat <= d_stat;
        E_icode <= d_icode;
        E_ifun <= d_ifun;
        E_valC <= d_valC;
        E_valA <= d_valA;
        E_valB <= d_valB;
        E_dstE <= d_dstE;
        E_dstM <= d_dstM;
        E_srcA <= d_srcA;
        E_srcB <= d_srcB;
    end
    else
    begin
        E_stat <= 1; //AOK Normal Operation
        E_icode <= 1; // nop
        E_ifun <= 0;
        E_valC <= 0;
        E_valA <= 0;
        E_valB <= 0;
        E_dstE <= 0;
        E_dstM <= 0;
        E_srcA <= 0;
        E_srcB <= 0;
    end
end
```

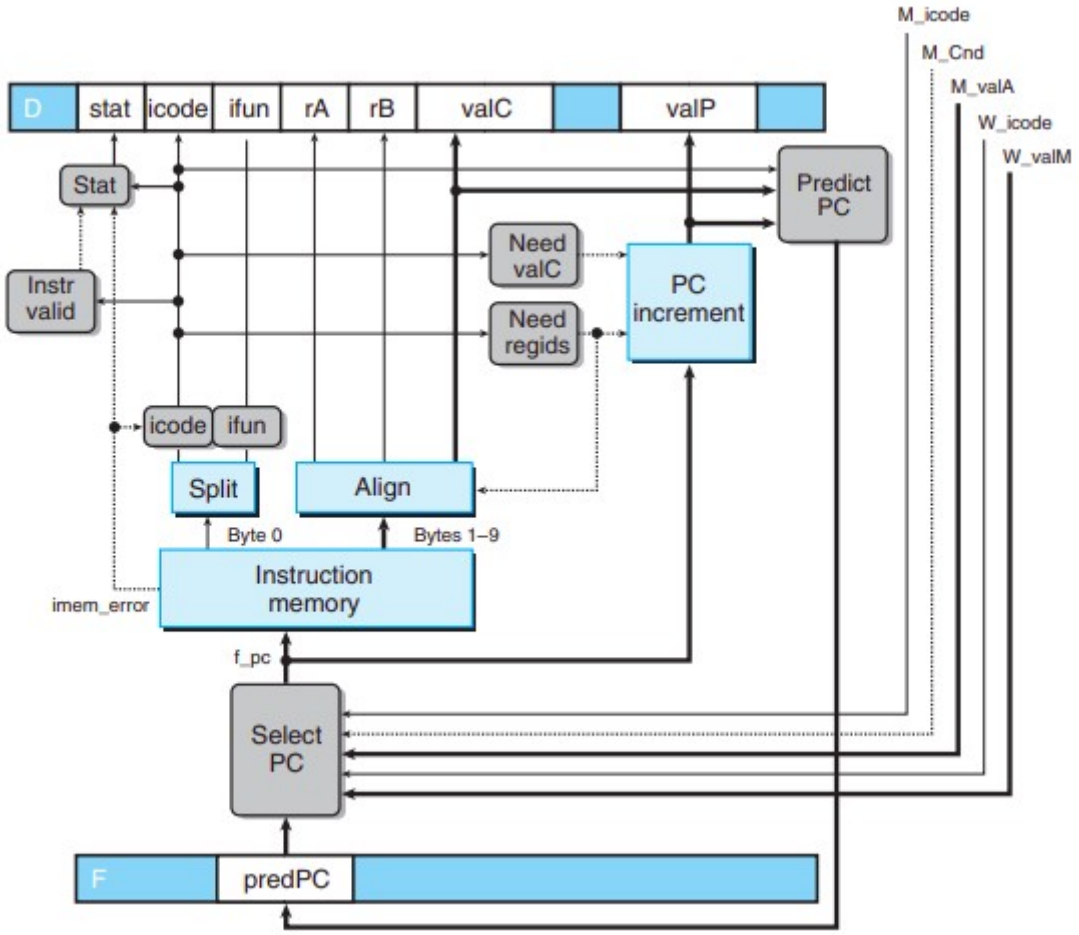
```
// Updating M register at every positive edge of clock
always @(posedge clk)
begin
    if(!M_bubble)
    begin
        M_stat <= e_stat;
        M_icode <= e_icode;
        M_valA <= e_valA;
        M_valE <= e_valE;
        M_cnd <= e_cnd;
        M_dstE <= e_dstE;
        M_dstM <= e_dstM;
    end
    else
    begin
        M_stat <= 1; //AOK Normal Operation
        M_icode <= 1; // nop
        M_valA <= 0;
        M_valE <= 0;
        M_cnd <= 0;
        M_dstE <= 0;
        M_dstM <= 0;
    end
end
```

```
// Updating W register at every positive edge of clock
always @(posedge clk)
begin
    if(!W_stall)
    begin

        W_stat <= m_stat;
        W_icode <= m_icode;
        W_dstE <= m_dstE;
        W_dstM <= m_dstM;
        W_valE <= m_valE;
        W_valM <= m_valM;
    end
end
```

```
initial
    begin
        $dumpfile("processor_tb.vcd");
        $dumpvars(0, processor);
    end

always
    begin
    clk <= 0;
    #10 clk <= ~clk;
    end


always@(*)
begin
    if(Stat != 1)
    begin
        $finish;
    end
end

endmodule
```

Fetch Module

In the fetch stage, the predicted PC from F is sent to the select PC block on the positive edge of the clock, which also has several inputs from the later stages (of an earlier instruction) to correctly calculate the correct PC value for an instruction to be fetched. If the predicted PC value is correct, it is passed to the fetch stage, or it is calculated from these later inputs. The required instruction is fetched and the appropriate values needed by the decode stage are sent to be stored in D.



Defining and initialising inputs and outputs:

```
`include "selectPC.v"
`include "predictPC.v"

module fetch(
    F_predPC, M_icode, M_cnd, M_valA, W_icode, W_valM,
    f_icode, f_ifun, f_rA, f_rB, f_valC, f_valP, f_predPC, f_stat
);

input [63:0] F_predPC, M_valA, W_valM;
input [3:0] M_icode, W_icode;
input M_cnd;

output reg [3:0] f_icode, f_ifun, f_rA, f_rB;
output reg [63:0] f_valC, f_valP;
output [63:0] f_predPC;
output reg [2:0] f_stat;
```

Instruction Set:

```
//Initalize Instruction Memory
reg [7:0] instructionMemory [0:1023];

initial begin
    //Read the testcase
    $readmemb("testcase.txt", instructionMemory);
end

wire [63:0] f_pc;

//Call the two other modules

selectPC select(.M_icode(M_icode), .M_cnd(M_cnd), .M_valA(M_valA),
.W_icode(W_icode), .W_valM(W_valM), .F_predPC(F_predPC), .f_pc(f_pc));

predictPC predict(.f_icode(f_icode), .f_valC(f_valC), .f_valP(f_valP),
 .f_predPC(f_predPC));

//Creating wires for instructionValid and imemError and others
reg instructionValid, imemError;
reg [0:7] instruction;
reg [0:7] regrArB;
```

Source Code:

```
always @(*)
begin
    if((f_icode < 4'b000) && (f_icode > 4'b1011))
    begin
        instructionValid = 1'b0;
    end

    else
    begin instructionValid = 1'b1; end
end
```

```
always @(*)
begin
    // Fetch based on f_icode

        if ((f_icode == 4'b0001) || (f_icode == 4'b1001)) //ret
        begin
            f_valP = f_pc + 1;
        end

        else if ((f_icode == 4'b0010) || (f_icode == 4'b0110) || (f_icode == 4'b1010) || (f_icode == 4'b1011))
        begin
            regrArB = {instructionMemory[f_pc+1]};
            f_rA = regrArB[0:3];
            f_rB = regrArB[4:7];
            f_valP = f_pc + 2;
            f_valC = 0;
        end

        else if (f_icode == 4'b0011 || f_icode == 4'b0100 || f_icode == 4'b0101) //irmovq, rmmovq & mrmovq
        begin
            regrArB = {instructionMemory[f_pc+1]};
            f_rA = regrArB[0:3];
            f_rB = regrArB[4:7];
            f_valC = { instructionMemory[9+f_pc],
            instructionMemory[8+f_pc], instructionMemory[7+f_pc],
            instructionMemory[6+f_pc], instructionMemory[5+f_pc], instructionMemory[4+f_pc],
            instructionMemory[3+f_pc], instructionMemory[2+f_pc]};
            f_valP = f_pc + 10;
        end

        else if (f_icode == 4'b0111 || f_icode == 4'b1000) // jXX Dest & call Dest
        begin
            f_valC = { instructionMemory[8+f_pc], instructionMemory[7+f_pc],
            instructionMemory[6+f_pc], instructionMemory[5+f_pc], instructionMemory[4+f_pc],
            instructionMemory[3+f_pc], instructionMemory[2+f_pc], instructionMemory[1+f_pc]};
            f_valP = f_pc + 9;
        end
```

```
        else
        begin
            f_rA = 15;
            f_rB = 15;
            f_valP = f_pc + 1;
        end
end
```

Predict PC Module

```
module predictPC (
    f_icode, f_valC, f_valP, f_predPC
);

input [3:0] f_icode;
input [63:0] f_valC, f_valP;

output reg [63:0] f_predPC;

// Logic : The PC prediction logic chooses valC for the fetched instruction when it is
// either a call or a jump, and valP otherwise.
always @(*)
begin
    f_predPC = f_valP; //Default Value

    if(f_icode == 4'b0111 || f_icode == 4'b1000)
    begin f_predPC = f_valC; end

end

endmodule
```

Instruction Validity and Memory Error:

```
always @(*)
begin

    if(f_pc < 0 || f_pc > 1023) // Max PC value is taken at my discretion to be 1023
    begin
        f_icode <= 1'b1;
        f_ifun <= 1'b0;
        imemError <= 1'b1;
    end

    else
    begin
        //Extracting Information about instruction
        //Default Values
        instruction <= {instructionMemory[f_pc]};
        f_icode <= instruction[0:3];
        f_ifun <= instruction[4:7];
        imemError <= 1'b0;
    end
end
```

Select Module

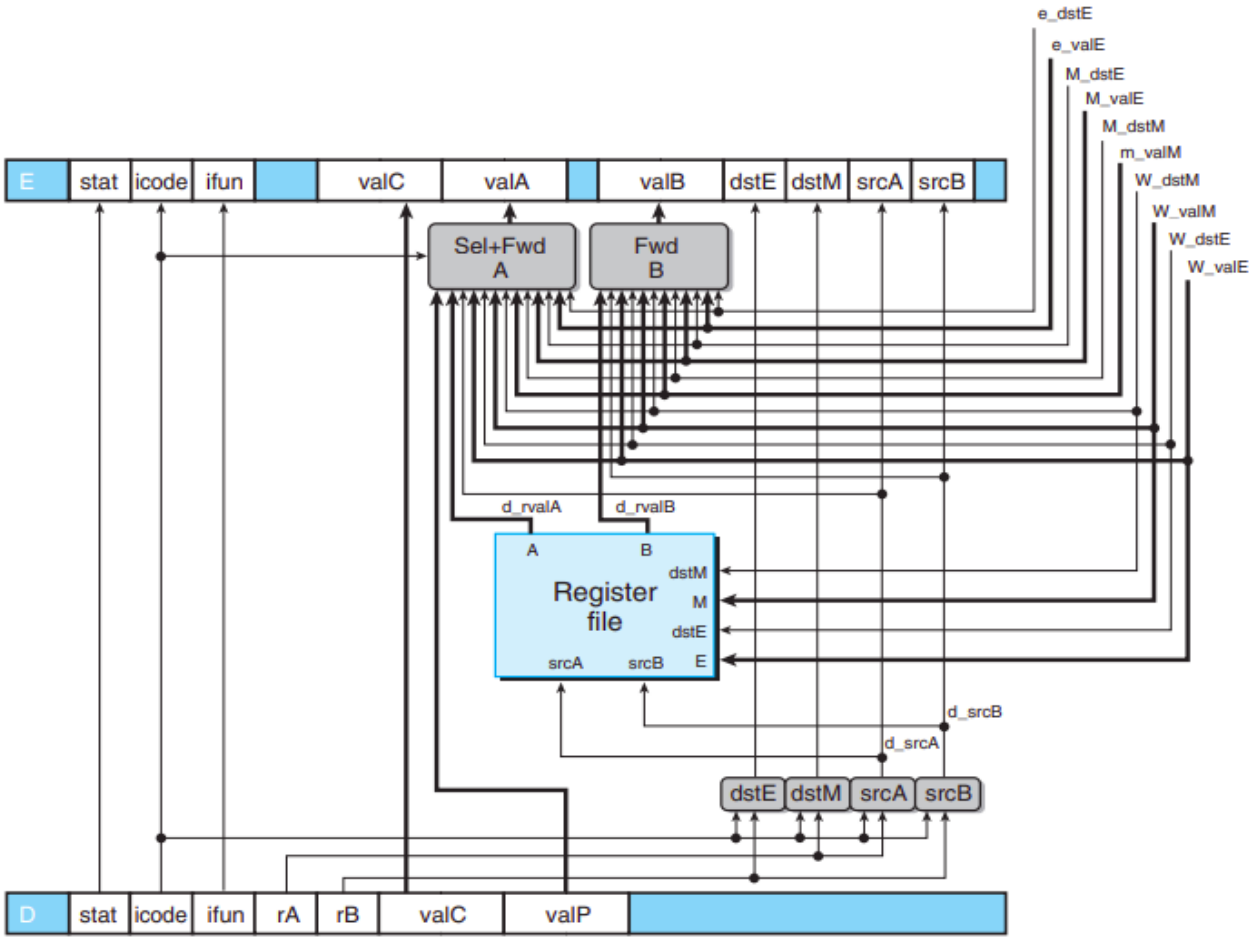
```
module selectPC (
    M_icode, M_cnd, M_valA, W_icode, W_valM, F_predPC, f_pc
);

input [3:0] M_icode, W_icode;
input M_cnd;
input [63:0] M_valA, W_valM, F_predPC;

output reg [63:0] f_pc;

always @(*)
begin

    f_pc = F_predPC; //Default Value

    if(M_icode == 4'b0111 && !M_cnd)
    begin f_pc = M_valA; end // Mispredicted branch. Fetch at incremented PC

    else if(W_icode == 4'b1001)
    begin f_pc = W_valM; end //C ompletion of RET instruction

end
endmodule
```

Status Module:

```
always @(*)
begin

    if(f_icode == 0)
    begin f_stat = 2; end // HLT

    else if(imemError == 1)
    begin f_stat = 3; end // ADR

    else if(instructionValid == 0)
    begin f_stat = 4; end // INS

    else
    begin f_stat = 1; end // AOK (Normal Operation)
end

endmodule
```

Decode

In the decode stage, the instruction is decoded and the required information is sent from D to E. This stage is the one where data forwarding is implemented, which often helps with prevention of loss of data. Since Verilog doesn't allow us to pass 2- dimensional arrays through function calls, all registers are sent separately.



Registers:

```
reg [63:0] memReg [0:15];


initial
begin
    memReg[0] = 64'd4;
    memReg[1] = 64'd2;
    memReg[2] = 64'd16;
    memReg[3] = 64'd16;
    memReg[4] = 64'd255;    // Rsp
    memReg[5] = 64'd33;
    memReg[6] = 64'd19;
    memReg[7] = 64'd7;
    memReg[8] = 64'd8;
    memReg[9] = 64'd9;
    memReg[10] = 64'd10;
    memReg[11] = 64'd11;
    memReg[12] = 64'd12;
    memReg[13] = 64'd13;
    memReg[14] = 64'd14;
    memReg[15] = 64'd0; //F register
end
```

D Block:

```
// Params with no change, wires
always @(*)
begin
    d_stat = D_stat;
    d_icode = D_icode;
    d_ifun = D_ifun;
    d_valC = D_valC;
end
```

Source Code:

```
//Update Register file
always @(posedge clk)
begin
    memReg[W_dstM] <= W_valM;
    memReg[W_dstE] <= W_valE;
end

// Setting up Destination Registers
// I Recommend seeing the slides regarding SEQ Implementation
// And then observe the destination regsiters chosen for different
// operations for decode (srcA, srcB), memory (d_dstE) and writeback (d_dest)
// Read the slides thoroughly do get an idea about how we are choosing the
// destination registers for different icodes.
// This has not been mentioned in the book directly, so you have to find your own way out.

always @(*)
begin
        case(d_icode)

        //cmovXX
        4'b0010:
        begin
            d_srcA = D_rA;
            d_srcB = 4'd15;
            d_dstE = D_rB;
            d_dstM = 4'd15;
        end

        //irmovq
        4'b0011:
        begin
            d_srcA = 4'd15;
            d_srcB = 4'd15;
            d_dstE = D_rB;
            d_dstM = 4'd15;
        end
```

```
//rmmovq
4'b0100:
begin
    d_srcA = D_rA;
    d_srcB = D_rB;
    d_dstE = 4'd15;
    d_dstM = 4'd15;
end

//mrmovq
4'b0101:
begin
    d_srcA = 4'd15;
    d_srcB = D_rB;
    d_dstE = 4'd15;
    d_dstM = D_rA;
end
```

```
//OPq
4'b0110:
begin
    d_srcA = D_rA;
    d_srcB = D_rB;
    d_dstE = D_rB;
    d_dstM = 4'd15;
end

//jXX
4'b0111:
begin
    d_srcA = 4'd15;
    d_srcB = 4'd15;
    d_dstE = 4'd15;
    d_dstM = 4'd15;
end
```

```
//call
4'b1000:
begin
    d_srcA = 4'd15;
    d_srcB = 4'd4;
    d_dstE = 4'd4;
    d_dstM = 4'd15;
end

//ret
4'b1001:
begin
    d_srcA = 4'd4;
    d_srcB = 4'd4;
    d_dstE = 4'd4;
    d_dstM = 4'd15;
end
```

```
//pushq
4'b1010:
begin
    d_srcA = D_rA;
    d_srcB = 4'd4;
    d_dstE = 4'd4;
    d_dstM = 4'd15;
end

//popq
4'b1011:
begin
    d_srcA = 4'd4;
    d_srcB = 4'd4;
    d_dstE = 4'd4;
    d_dstM = D_rA;
end
endcase
```

Sel + Fwd A block

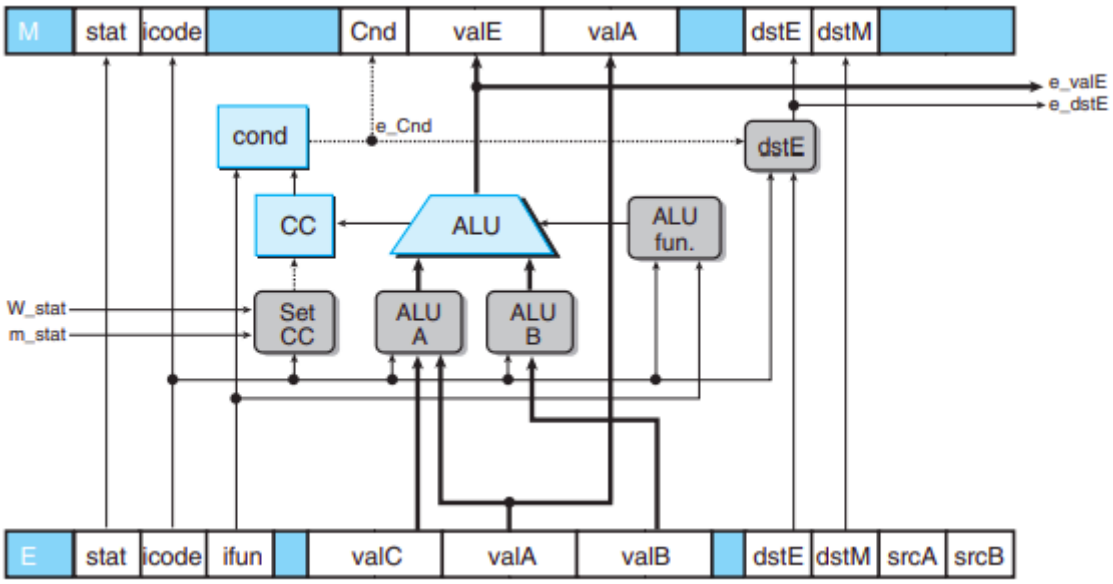
```
// Sel+Fwd A Logic --> Mentioned in book
// The order matters here
always @(*)
begin
    if(D_icode == 4'd7 || D_icode == 4'd8)
    begin d_valA = D_valP; end

    else if(d_srcA == e_dstE)
    begin d_valA = e_valE; end

    else if(d_srcA == M_dstM)
    begin d_valA = m_valM; end

    else if(d_srcA == M_dstE)
    begin d_valA = M_valE; end

    else if(d_srcA == W_dstM)
    begin d_valA = W_valM; end

    else if(d_srcA == W_dstE)
    begin d_valA = W_valE; end

    else
    begin d_valA = memReg[d_srcA]; end

end
```

Fwd B Block

```
// Fwd B Logic --> Logic similar to Fwd A
// The order matters here
always @(*)
begin

    if(d_srcB == e_dstE)
    begin d_valB = e_valE; end

    else if(d_srcB == M_dstM)
    begin d_valB = m_valM; end

    else if(d_srcB == M_dstE)
    begin d_valB = M_valE; end

    else if(d_srcB == W_dstM)
    begin d_valB = W_valM; end

    else if(d_srcB == W_dstE)
    begin d_valB = W_valE; end

    else
    begin d_valB = memReg[d_srcB]; end

end
```

Execute

The execute stage contains the ALU. The instruction that was decoded is executed as required. The appropriate information is sent from E to M. The condition codes are set, which lets us know whether to update them or not and condition is forwarded to M accordingly. The implementation, for the larger part, is very similar to that in SEQ.



E block (reg)

```
// Params with no change, wires
always @(*)
begin
    e_stat = E_stat;
    e_icode = E_icode;
    e_valA = E_valA;
    e_dstM = E_dstM;
end
```

ALU

```
reg signed [63:0] A_input;
reg signed [63:0] B_input;
reg [1:0] select_lines;

wire signed [63:0] ALU_Output;
wire overflow;

alu alu1(.A(A_input), .B(B_input), .S0(select_lines[0]),
.S1(select_lines[1]), .Output(ALU_Output), .Overflow(overflow));
```

Flags

```
always @(posedge clk)
begin
    if((e_icode == 4'b0110) && (m_stat == 1) && (W_stat == 1)) //remember stat = 1 indicates normal operation
    ZF <= (ALU_Output == 64'b0);
    SF <= (ALU_Output < 64'b0);
    OF <= (A_input < 64'b0 == B_input < 64'b0) && (ALU_Output < 64'b0 != A_input < 64'b0);
end
```

Assign ALU values

```
always @(*)
begin
    // Perform operations based on icode
    case (e_icode)

        4'd0, 4'd1: begin end // do nothing

        4'd2: // cmovXX rA, rB

            //Since this is Y-86, we only have [zf, sf, of] (in this
            //That leaves us with the six conditional move instructio

            // 1. cmovle (SF ^ OF) | ZF
            // 2. cmovl SF ^ OF
            // 3. cmove ZF
            // 4. cmovne ~ZF
            // 5. cmovge ~ (SF ^ OF)
            // 6. cmovg ~ ((SF ^ OF) | ZF)

            begin
                select_lines[0] = 1'd0;
                select_lines[1] = 1'd0; // Select Addition

                if(E_ifun == 4'd0) //Unconditional move
                begin
                    A_input = E_valA;
                    B_input = 64'd0;
                    e_valE = ALU_Output;
                    e_cnd = 1'b1;
                end

                else if (E_ifun == 4'd1 && ((SF ^ OF) | ZF)) //cmovle
                begin
                    A_input = E_valA;
                    B_input = 64'd0;
                    e_valE = ALU_Output;
                    e_cnd = 1'b1;
                end
```

```
    else if (E_ifun == 4'd2 && (SF ^ OF)) //cmovl
    begin
        A_input = E_valA;
        B_input = 64'd0;
        e_valE = ALU_Output;
        e_cnd = 1'b1;
    end
    else if (E_ifun == 4'd3 && ZF) //cmove
    begin
        A_input = E_valA;
        B_input = 64'd0;
        e_valE = ALU_Output;
        e_cnd = 1'b1;
    end
    else if (E_ifun == 4'd4 && !(ZF)) //cmovne
    begin
        A_input = E_valA;
        B_input = 64'd0;
        e_valE = ALU_Output;
        e_cnd = 1'b1;
    end
    else if (E_ifun == 4'd5 && !(SF ^ OF)) //cmovge
    begin
        A_input = E_valA;
        B_input = 64'd0;
        e_valE = ALU_Output;
        e_cnd = 1'b1;
    end
    else if (E_ifun == 4'd6 && !((SF ^ OF) || ZF)) //cmovg
    begin
        A_input = E_valA;
        B_input = 64'd0;
        e_valE = ALU_Output;
        e_cnd = 1'b1;
    end
end
```

```
4'd3: // irmovq
    begin
        select_lines[0] = 1'd0;
        select_lines[1] = 1'd0; // Select Addition
        A_input = E_valC;
        B_input = 64'd0;
        e_valE = ALU_Output; //valE = valC + 0
    end

4'd4, 4'd5: // rmmovq, mrmovq
    begin
        select_lines[0] = 1'd0;
        select_lines[1] = 1'd0; // Select Addition
        A_input = E_valC;
        B_input = E_valB;
        e_valE = ALU_Output; // valE = valB + valC
    end

4'd6: // OPq
    begin

        select_lines[0] = E_ifun[0];
        select_lines[1] = E_ifun[1];

        A_input = E_valB;
        B_input = E_valA;
        e_valE = ALU_Output; // VAlE = valB (op) valA

        ZF = (ALU_Output == 64'b0);
        SF = (ALU_Output < 64'b0);
        OF = (A_input < 64'b0 == B_input < 64'b0) && (ALU_Output < 64'b0 != A_input < 64'b0);

    end
```

```
4'd7: // jXX
    begin
        e_cnd = 1'b0;

        if (E_ifun == 4'd0) // jmp (unconditional)
            e_cnd = 1'b1;
        else if (E_ifun == 4'd1 && (ZF || (SF ^ OF))) // jle
            e_cnd = 1'b1;
        else if (E_ifun == 4'd2 && (SF ^ OF)) // jl
            e_cnd = 1'b1;
        else if (E_ifun == 4'd3 && ZF) // je
            e_cnd = 1'b1;
        else if (E_ifun == 4'd4 && !(ZF)) // jne
            e_cnd = 1'b1;
        else if (E_ifun == 4'd5 && !(SF ^ OF)) // jge
            e_cnd = 1'b1;
        else if (E_ifun == 4'd6 && !((SF ^ OF) || ZF)) // jg
            e_cnd = 1'b1;
    end
```

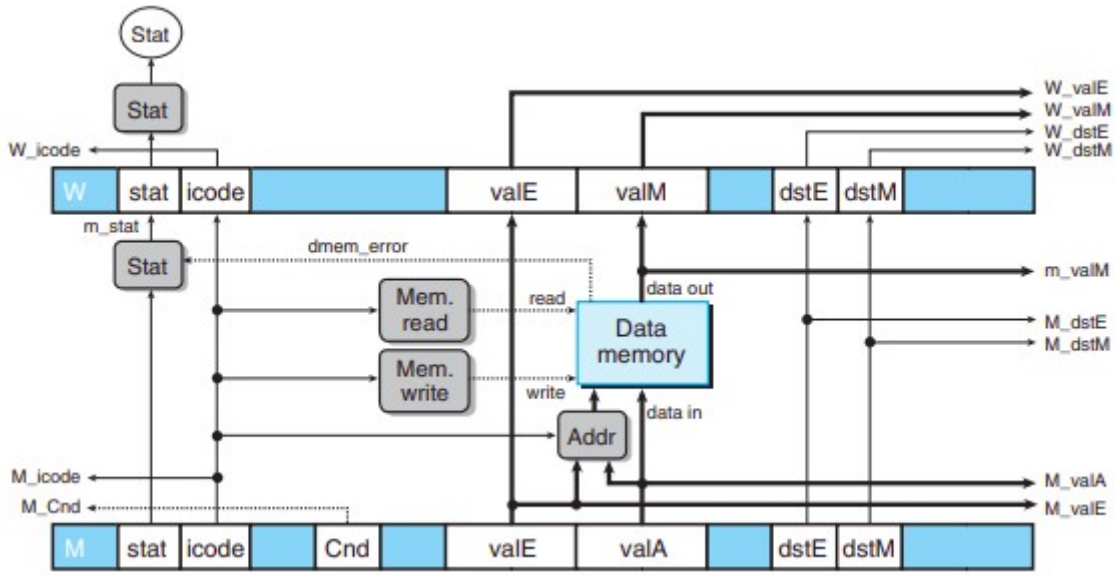
```
4'd11, 4'd9: // popq or ret
    begin
        select_lines[0] = 1'b0;
        select_lines[1] = 1'b0;
        A_input = E_valB;
        B_input = 64'd8;
        e_valE = ALU_Output; //valE = valB + 8
    end

4'd10, 4'd8: // pushq or call
    begin
        select_lines[0] = 1'b1;
        select_lines[1] = 1'b0; //Choosing subtraction
        A_input = E_valB;
        B_input = 64'd8;
        e_valE = ALU_Output; //valE = valB - 8
    end
```

Destination Value

```
// Setting up destination register for execute
always @(*)
begin
    if((e_icode == 4'b0010) && (e_cnd == 1'b0))
    begin e_dstE = 4'd15; end // rB < -- 0xF, refer slides

    else
    begin e_dstE = E_dstE; end
end
```

Memory

The memory stage is where the data is read from or written to the memory. The appropriate information is sent from M to W. The memory is declared separate from the instruction memory and does not see use in the other stages as it is accessed only in this stage. A striking feature of this stage is the large number of signals that are sent to the earlier instructions (for potential data problems of later instructions to take care of) from M, the stage itself and W.



Initiating the data memory

```
// Initiating data memory
reg [63:0] mem[0:1023]; //considering 256 words to work with
integer i;

initial
    begin
        for (i = 1023; i >= 0; i = i - 1)
            begin
                mem[i] <= 0;
            end
    end
```

Error and Status Block

```
always @(*)
begin
    if((m_icode == 4'd10) || (m_icode == 4'd8) || (m_icode == 4'd4) || (m_icode == 4'd5))
    begin address_memory = m_valE; end

    else if((m_icode == 4'd9) || (m_icode == 4'd11))
    begin address_memory = M_valA; end

    else
    begin address_memory = 1023; end
end

// Checking memory error
reg dmem_error;
always @(*)
    begin
        dmem_error = (address_memory > 1023 || address_memory < 0);
    end
```

Write Block

```
// Writing back to data memory anytime
always @(posedge clk)
    begin
        if(!dmem_error)
        begin
                case (m_icode)
                    4'd10: //pushq
                    begin
                        mem[m_valE] <= M_valA;
                    end

                    4'd8: //call
                    begin
                        mem[m_valE] <= M_valA;
                    end

                    4'd4: //rmmovq
                    begin
                        mem[m_valE] <= M_valA;
                    end
                endcase
        end
    end
```

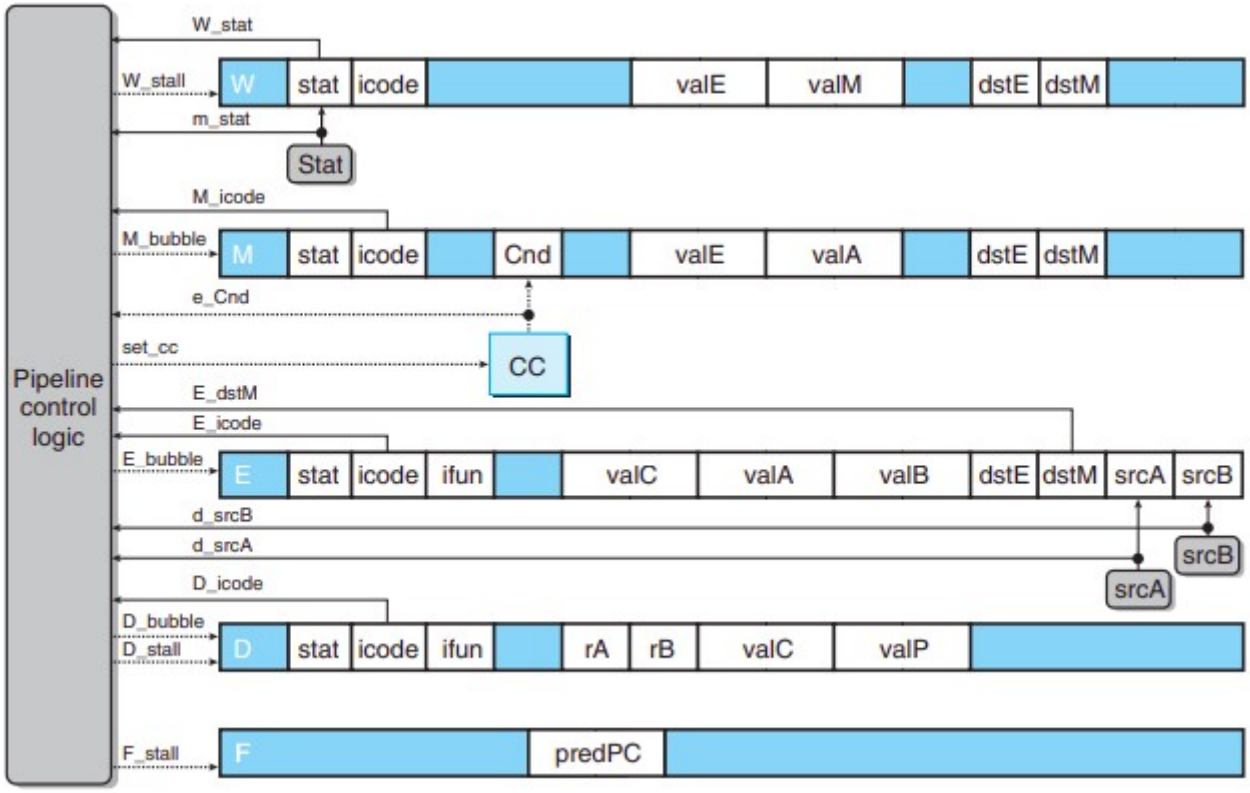
Read Block

```
//Reading from the memory at any instant
always @(*)
    begin
        if(!dmem_error)
            begin
                case (m_icode)

                    4'd9, 4'd11: //popq and ret
                    begin
                        m_valM = mem[M_valA];
                    end

                    4'd5: //mrmovq
                    begin
                        m_valM = mem[M_valE];
                    end

                    default:
                    begin
                        m_valM = 0;
                    end
                endcase
            end
    end
```

M block (reg)

```
// Params with no change, wires
always @(*)
    begin
        m_icode = M_icode;
        m_valE = M_valE;
        m_dstE = M_dstE;
        m_dstM = M_dstM;
    end
```

Control



There are certain control cases which cannot completely be handled by data forwarding and branch prediction. These cases are listed below: -

1. Load/use hazards: For two consecutive instructions where the first reads a value from memory and the second uses that value requires the pipeline to stall for a single cycle.

2. Processing ret: While the ret instruction is being run, the pipeline must be stalled until ret reaches write back.

3. Mispredicted branches: If a jump that was not supposed to happen occurs, the pipeline must cancel all the instructions that have already entered the pipeline and fetch the instruction just after the jump instruction.

4. Exceptions

Predict Hazards

```
// Using assign statement because we are using wires --> refer textbook for logic
assign processing_ret = ((M_icode == 4'd9) || (E_icode == 4'd9) || (D_icode == 4'd9)) ? 1 : 0;
assign lu_hazard = (((E_icode == 4'd5) || (E_icode == 4'd11)) && ((E_dstM == d_srcA) || (E_dstM == d_srcB))) ? 1 : 0;
assign mispredicted_branch = ((E_icode == 4'd7) && !e_cnd) ? 1 : 0;
assign execption = ((m_stat == 3'd2) || (m_stat == 3'd3) || (m_stat == 3'd4) || (W_stat == 3'd2) || (W_stat == 3'd3) || (W_stat == 3'd4)) ? 1 : 0;
```

Generating Stalls and Bubbles

```
// Assigning stall and bubble values --> Everything can be find in text and its solutions

// 1.  Pipeline register F must be stalled for either a load/use hazard or a ret instruction:
always @(*)
begin F_stall = (processing_ret == 1 || lu_hazard == 1); end

// 2. Pipeline register D must be set to bubble for a mispredicted branch or a ret instruction
always @(*)
begin D_stall = (lu_hazard == 1); end

always @(*)
begin D_bubble = ((mispredicted_branch == 1 || processing_ret == 1) && !(D_stall)); end

// 3. pipeline register E must be set to bubble for a load/use hazard or for a mispredicted branch
always @(*)
begin E_bubble = (lu_hazard == 1 || mispredicted_branch ==1); end

// 4. Injecting a bubble into the memory stage on the next cycle involves checking for
// an exception in either the memory or the write-back stage during the current cycle

always @(*)
begin M_bubble = (execption == 1); end

// 5.For stalling the write-back stage, we check only the status of the instruction
// in this stage.

always @(*)
begin W_stall = ((W_stat == 3'd2) || (W_stat == 3'd3) || (W_stat == 3'd4)); end
```

Testing and Output:

1) cmovxx

```
subq %rbx, %rdx
cmove %rbp, %rsi
halt
```

Initial Register value

```
clk= 0
reg0=                  4 (rax)
reg1=                  2 (rcx)
reg2=                 16 (rdx)
reg3=                 16 (rbx)
reg4=                255 (rsp)
reg5=                 33 (rbp)
reg6=                 19 (rsi)
reg7=                  7 (rdi)
reg8=                  8 (r8)
reg9=                  9 (r9)
reg10=                 10 (r10)
reg11=                 11 (r11)
reg12=                 12 (r12)
reg13=                 13 (r13)
reg14=                 14 (r14)
```

Final Register value

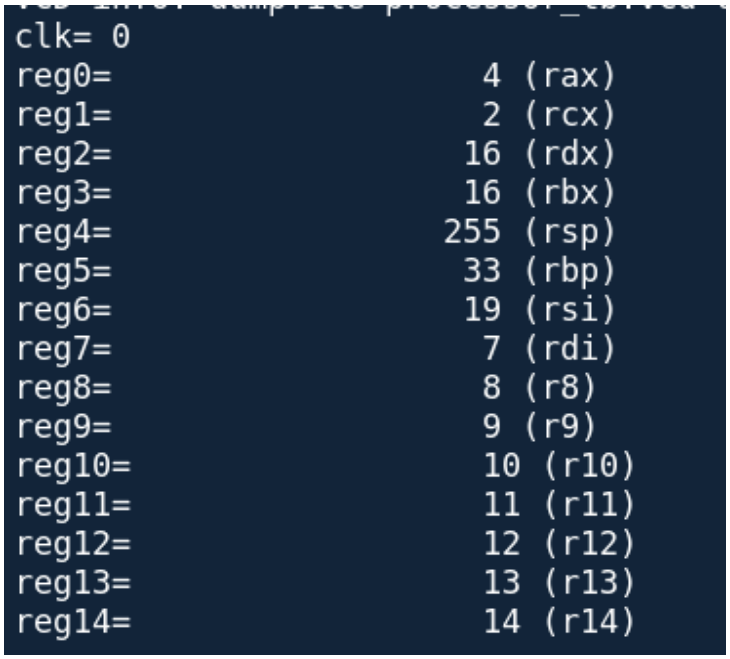
```
clk= 1
reg0=                  x (rax)
reg1=                  2 (rcx)
reg2=                  0 (rdx)
reg3=                 16 (rbx)
reg4=                255 (rsp)
reg5=                 33 (rbp)
reg6=                 33 (rsi)
reg7=                  7 (rdi)
reg8=                  8 (r8)
reg9=                  9 (r9)
reg10=                 10 (r10)
reg11=                 11 (r11)
reg12=                 12 (r12)
reg13=                 13 (r13)
reg14=                 14 (r14)
```

Since reg2 (%rax)= reg3 (%rbx)= 16. Therefore, conditional for move is satisfied and value of reg5 = 33 (%rbp) is moved to reg6 (%rsi).

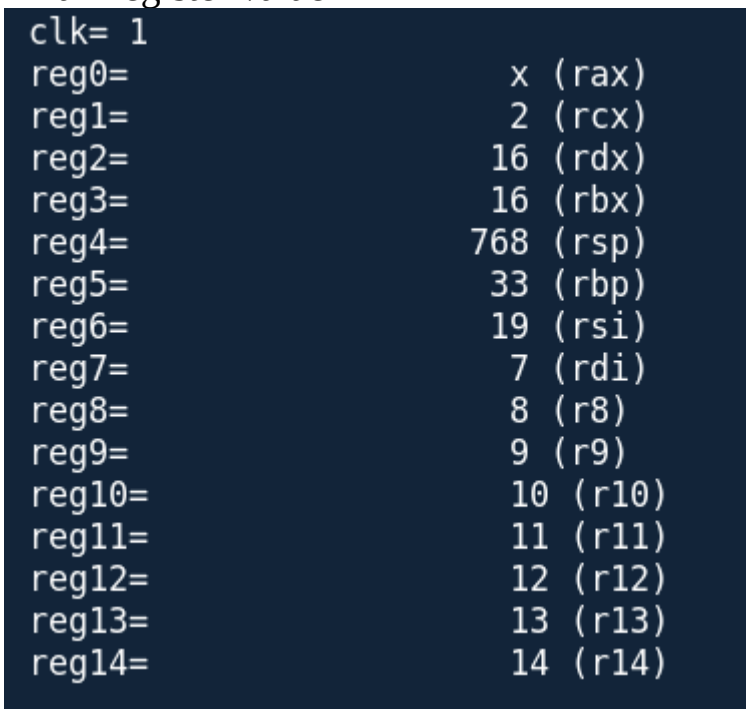
2) irmovq

```
irmovq $0x300, %rsp
```

Initial Register value



Final Register value



immediate value 0x300 is moved to reg4 (%rsp) = 768.

3) rmmovq and mrmovq

```
rmmovq %r8, (%r9)
mrmovq (%r9), %r10
```

Initial Register value

```
clk= 0
reg0=                   4 (rax)
reg1=                   2 (rcx)
reg2=                  16 (rdx)
reg3=                  16 (rbx)
reg4=                 255 (rsp)
reg5=                  33 (rbp)
reg6=                  19 (rsi)
reg7=                   7 (rdi)
reg8=                   8 (r8)
reg9=                   9 (r9)
reg10=                 10 (r10)
reg11=                 11 (r11)
reg12=                 12 (r12)
reg13=                 13 (r13)
reg14=                 14 (r14)
```

Final Register value

```
clk= 1
reg0=                   x (rax)
reg1=                   2 (rcx)
reg2=                  16 (rdx)
reg3=                  16 (rbx)
reg4=                 255 (rsp)
reg5=                  33 (rbp)
reg6=                  19 (rsi)
reg7=                   7 (rdi)
reg8=                   8 (r8)
reg9=                   9 (r9)
reg10=                  8 (r10)
reg11=                 11 (r11)
reg12=                 12 (r12)
reg13=                 13 (r13)
reg14=                 14 (r14)
```

In **rmmovq %r8, (%r9**) => register 8 value (i.e., 8) is moved to the address of the memory stored in register 9
In **mrmovq (%r9), %r10** => value at the memory address stored in register 9 is moved to register 10. Therefore final value of register 10 = 8.

4) OPq

```
subq %rbx, %rdx
halt
```

Initial Register value

```
clk= 0
reg0=                4 (rax)
reg1=                2 (rcx)
reg2=               16 (rdx)
reg3=               16 (rbx)
reg4=              255 (rsp)
reg5=               33 (rbp)
reg6=               19 (rsi)
reg7=                7 (rdi)
reg8=                8 (r8)
reg9=                9 (r9)
reg10=               10 (r10)
reg11=               11 (r11)
reg12=               12 (r12)
reg13=               13 (r13)
reg14=               14 (r14)
```

Final Register value

```
clk= 1
reg0=                x (rax)
reg1=                2 (rcx)
reg2=                0 (rdx)
reg3=               16 (rbx)
reg4=              255 (rsp)
reg5=               33 (rbp)
reg6=               19 (rsi)
reg7=                7 (rdi)
reg8=                8 (r8)
reg9=                9 (r9)
reg10=               10 (r10)
reg11=               11 (r11)
reg12=               12 (r12)
reg13=               13 (r13)
reg14=               14 (r14)
```

subq %rbx, %rdx performs value at reg2 – value at reg3 and stores it in reg2 Here, value at reg2 = 16 and value at reg3 = 16. Hence after performing subtraction 0 is stored in reg2.

5) jXX

```
subq %rdx, %rdx
je .L1
irmovq $0x300, %rbx
halt
.L1:
    irmovq $0x100, %rbx
    halt
```

Initial Register value

```
clk= 0
reg0=                   4 (rax)
reg1=                   2 (rcx)
reg2=                  16 (rdx)
reg3=                  16 (rbx)
reg4=                 255 (rsp)
reg5=                  33 (rbp)
reg6=                  19 (rsi)
reg7=                   7 (rdi)
reg8=                   8 (r8)
reg9=                   9 (r9)
reg10=                 10 (r10)
reg11=                 11 (r11)
reg12=                 12 (r12)
reg13=                 13 (r13)
reg14=                 14 (r14)
```

Final Register value

```
clk= 1
reg0=                   x (rax)
reg1=                   2 (rcx)
reg2=                   0 (rdx)
reg3=                 100 (rbx)
reg4=                 255 (rsp)
reg5=                  33 (rbp)
reg6=                  19 (rsi)
reg7=                   7 (rdi)
reg8=                   8 (r8)
reg9=                   9 (r9)
reg10=                 10 (r10)
reg11=                 11 (r11)
reg12=                 12 (r12)
reg13=                 13 (r13)
reg14=                 14 (r14)
```

Since the last operation is equal hence it will jump to .L1 and the value $0x100 is moved to register 3 (value at reg3 = 100)

6) call and return

```
irmovq $0x300, %rsp
call main
halt

main: irmovq $0x100, %rbx
    irmovq $0x200, %rdx
    addq %rdx, %rbx
    ret
```

Initial Register value

```
clk= 0
reg0=                   4 (rax)
reg1=                   2 (rcx)
reg2=                  16 (rdx)
reg3=                  16 (rbx)
reg4=                 255 (rsp)
reg5=                  33 (rbp)
reg6=                  19 (rsi)
reg7=                   7 (rdi)
reg8=                   8 (r8)
reg9=                   9 (r9)
reg10=                 10 (r10)
reg11=                 11 (r11)
reg12=                 12 (r12)
reg13=                 13 (r13)
reg14=                 14 (r14)
```

Final Register value

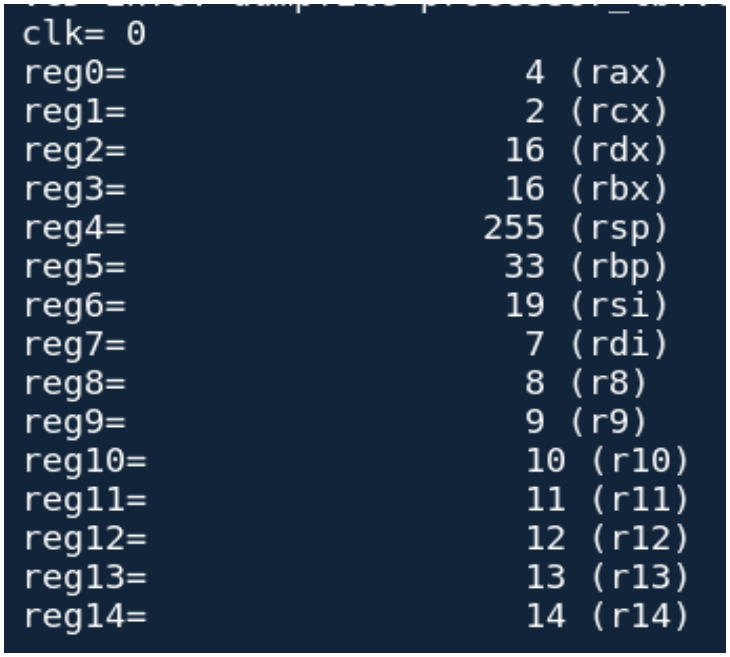
```
clk= 1
reg0=                   x (rax)
reg1=                   2 (rcx)
reg2=                 512 (rdx)
reg3=                 768 (rbx)
reg4=                 768 (rsp)
reg5=                  33 (rbp)
reg6=                  19 (rsi)
reg7=                   7 (rdi)
reg8=                   8 (r8)
reg9=                   9 (r9)
reg10=                 10 (r10)
reg11=                 11 (r11)
reg12=                 12 (r12)
reg13=                 13 (r13)
reg14=                 14 (r14)
```

The above call and return function is working properly as the value of reg3 is being updated to 768.

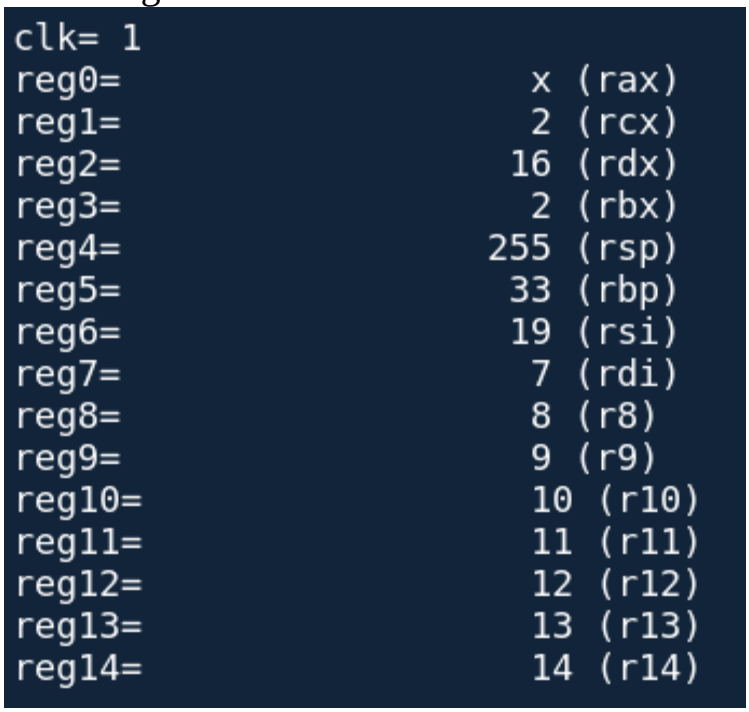
7) push and pop

```
pushq %rcx
popq %rbx
halt
```

Initial Register value



Final Register value



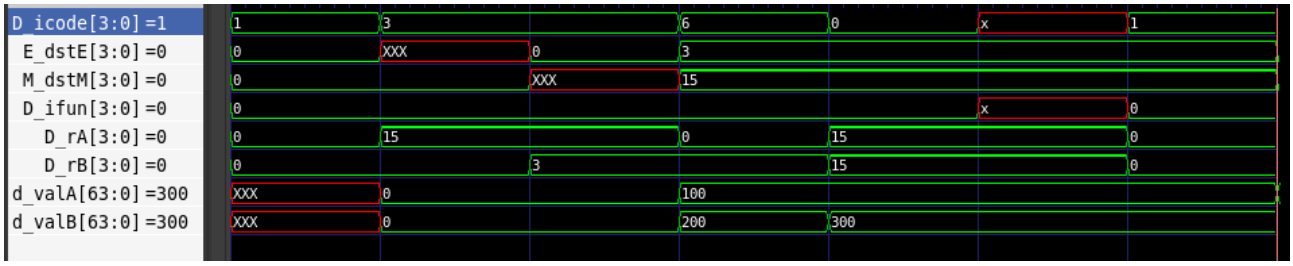
First the value of register 1 is pushed to the stack pointer and then its popped and stored in register 3 and hence value of reg3 = 2.

Hazards Handling:

We have tested the code for different hazards

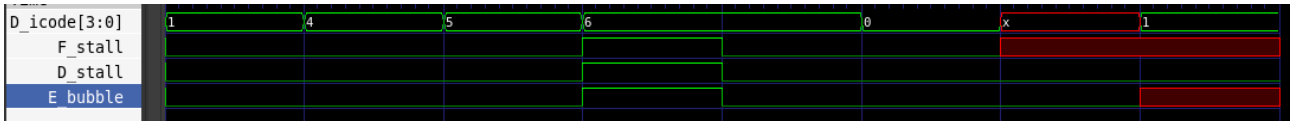
1) Data Forwarding

```
irmovq $100, %rax
irmovq $200, %rbx
addq %rax, %rbx
```



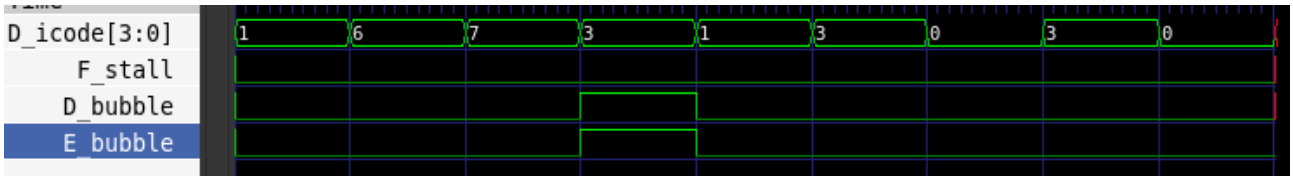
2) Load/Use Hazard

```
rmmovq %rax, (%rdx)
mrmovq (%rdx), %rbp
addq %rcx, %rbp
halt
```



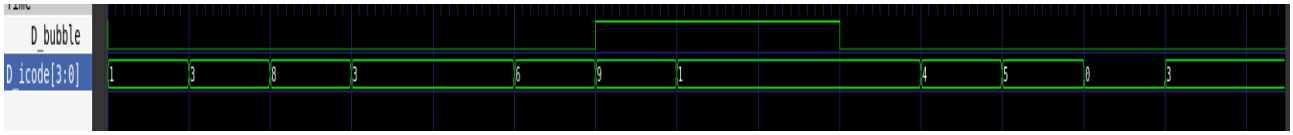
3) Mispredicted Branch

```
subq %rdx, %rax
je .L1
irmovq $300, %rbx
halt
.L1:
    irmovq $100, %rbx
    halt
```



4) Processing ret

```
irmovq $0x300, %rsp
call main
rmmovq %rbx, (%r9)
mrmovq (%r9), %rax
halt

main: irmovq $0x100, %rbx
    irmovq $0x200, %rdx
    addq %rdx, %rbx
    ret
```



Hence all the exceptions are being handled carefully according to the below table.

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

Challenges Faced:

1. Figuring out where to implement always@(*) and where to implement always@(posedge clk).
2. Differentiating between the various PC signals in the fetch stage proved to be confusing.
3. Because the pipelined implementation more than doubles the number of wires, keeping track of all of them in the various modules is difficult.
4. Keeping decode and write back in the pipelined implementation in separate files for clarity's sake proved to be difficult.
5. Figuring out the control logic and implementing it correctly was also rather challenging especially in case of the pipelined processor

Acknowledgement

Working on this project has been a great learning experience. Over the last few weeks, we developed a deeper understanding of processor architecture design, instruction set architecture, memory and many of the other concepts required for this project. I would like to thank Prof. Deepak Gangadharan and the TAs for guiding us throughout the duration of this project.