IPA Project Report

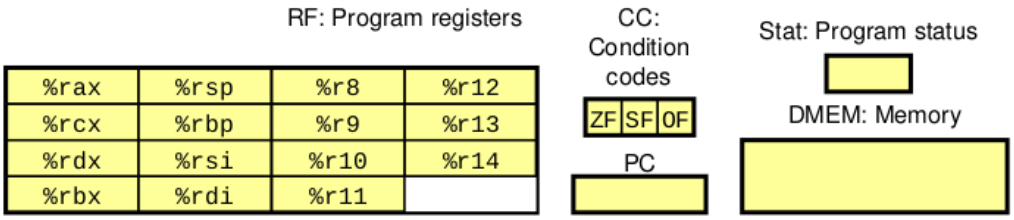
Sreeja Guduri (2021102007), Paridhi Prasad (2021102008)

In this project, a processor architecture following the Y86-64 ISA was implemented using Verilog. The implementation was done modularly, with the first being a sequential implementation, followed by a pipelined implementation. The sequential design executes instructions in 6 stages: Fetch, Decode, Execute, Memory, Write Back, and PC Update. In the pipelined implementation, the PC Update stage is merged with the Fetch stage, and there are subsequently 5 stages.

Y86-64 Instruction Set Architecture

The Y86 ISA is a subset of the X86 ISA. It is simpler, has fewer addressing modes, and has a simpler set of instructions.

Processor State



The Processor has:

- 15 registers
- Condition codes: ZF (Zero Flag), SF (Sign Flag), OF (Overflow Flag)
- PC (Program Counter)
- Memory
- Program Status

Instruction Set

The instruction set defines all the instructions that are part of the instruction set architecture, and the function calls for each instruction.

Y86 Instruction Set Reference

Instruction	Byte offset from PC									
	0	1	2	3	4	5	6	7	8	9
halt	0 0									
nop	1 0									
cmovXX rA, rB	2 fn	rA rB								
irmovq V, rB	3 0	f rB				V				
rmmovq rA, D(rB)	4 0	rA rB				D				
mrmovq D(rB), rA	5 0	rA rB				D				

Instruction	Byte offset from PC								
	0	1	2	3	4	5	6	7	8
OPq rA, rB	6 fn	rA rB							
jXX Dest	7 fn	Dest							
call Dest	8 0	Dest							
ret	9 0								
pushq rA	a 0	rA f							
popq rA	b 0	rA f							

The instruction calls read 1-10 bytes from memory. The icode and ifun values are determined by the first byte of the instructions. The subsequent bytes are as seen in the figure.

In addition, three of the instructions, cmovxx, opq and jxx have sub-instructions defined:

cmovXX:			
rrmovq	20	cmovne	24
cmovle	21	cmovge	25
cmovl	22	cmovg	26
cmove	23		

OPq:	
addq	60
subq	61
andq	62
xorq	63

jXX:			
jmp	70	jne	74
jle	71	jge	75
jl	72	jg	76
je	73		

Instructions

- halt (icode - 0)
  - The halt instruction.
  - stops instruction execution
- nop (icode - 1)
  - This is the no operation instruction.
  - no instruction is executed in that clock cycle
- cmovxx (icode - 2)
  - This is the conditional move instruction.
  - The figure here describes the move conditions for each sub-instruction, determined on the basis of the condition codes, ZF, SF and OF. The sub-instruction itself is chosen based on the ifun value.

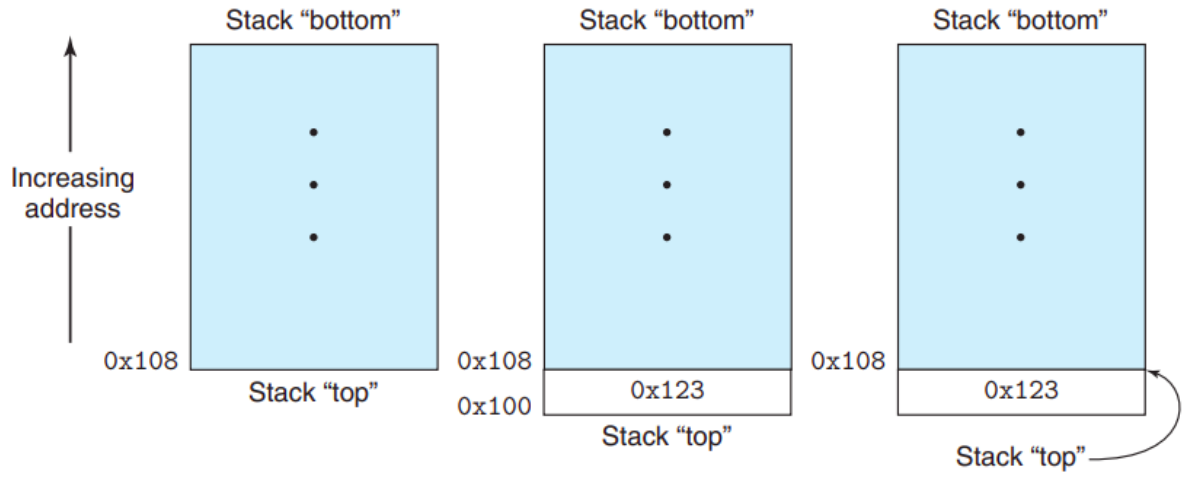
Instruction		Synonym	Move condition	Description
cmove	S, R	cmovz	ZF	Equal / zero
cmovne	S, R	cmovnz	~ZF	Not equal / not zero
cmovs	S, R		SF	Negative
cmovns	S, R		~SF	Nonnegative
cmovg	S, R	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge	S, R	cmovnl	~(SF ^ OF)	Greater or equal (signed >=)
cmovl	S, R	cmovnge	SF ^ OF	Less (signed <)
cmovle	S, R	cmovng	(SF ^ OF) | ZF	Less or equal (signed <=)

  - If the condition is TRUE, then the move instruction is executed.
  - In the case of rrmovq (register to register move), there is no condition, and the move happens in all scenarios.
- irmovq (icode - 3)
  - This is the immediate to register move instruction.
  - The source is immediate (i), and the destination is the specified register.
- rmmmovq (icode- 4)
  - The register to memory move instruction.
  - The source is a register and the destination is memory. The memory address and the register are specified in the instruction call.
- mrmovq (icode - 5)
  - The memory to register move instruction.

    - The source is memory and the destination is a register. The memory address and the register are specified in the instruction call.
- opq (icode - 6)
    - This is the operation instruction. It calls on the ALU to perform mathematical and logical operations.
    - As described in the figure above, the ifun value determines which operation needs to be executed by the ALU.
- jxx (icode - 7)
    - The jump instruction. The ifun value determines the branch condition.
    - The branched jump instruction is executed only if it's respective conditon is satisfied (i.e., TRUE), which itself is decided based on the condition codes.

Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge`	*Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl`	*Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle`	*Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)

- call (icode - 8)
    - This instruction is used to call another instruction.
    - The return address is pushed onto the stack, and a jump is performed to the destination address.
- ret (icode - 9)
    - This is the return instruction.
    - It performs a return from a call instruction.
- pushq (icode - A)
    - The push instruction. It pushes the contents of the specified register on to the stack.
    - The stack is defined as follows:



- popq (icode - B)
  - The pop instruction, analogous to the push instruction.

Module I - Sequential Design

Overview

The sequential processor, or SEQ, implements instructions defined by the Y86 Instruction Set Architecture in 6 stages. In each clock cycle, all stages of one instruction are executed.

As a single instruction takes up a clock cycle to be executed, the SEQ processor requires a very long cycle time, i.e., a very low clock rate. The SEQ implementation has many such shortcomings, but these can be rectified by pipelining. The SEQ implementation can, hence, essentially be viewed as a stepping stone to the pipelined processor, which is far more efficient.

Stages of Processing

The SEQ processor divides the processing of instructions into six stages.

1. Fetch

During this stage, an instruction is fetched from the instruction memory, instr_mem, using the value in the PC as the memory address. The values of icode, ifun, valC, valP, rA and rB are set based on the fetched instruction.

Here is the SEQ implementation of Fetch.

```
`timescale 1ns/10ps

module fetch(clk,PC,icode,ifun,rA,rB,valC,valP,in_mem,in_inst,hlt);

//[63:0] after regA makes it an array but putting it before means its a ner
input clk;
input [63:0] PC; //memory address is 64 bits
output reg [3:0]icode; //first 4 bits of the first byte
output reg [3:0]ifun; //last 4 bits of the first byte
output reg [3:0]rA; //first 4 bits of the second byte
output reg [3:0]rB; //last 4 bits of the second byte
output reg signed [63:0]valC; //stores the constant word
output reg [63:0]valP; //stores the address of next instruction (PC should be this value)
```

```
output reg in_mem; //invalid memory
output reg in_inst; //invalid instruction
output reg hlt; //inavlid halt

//setting up instruction memory
reg [7:0] instr_mem[0:1023]; //this means we have a 100 rows of 1 byte instructions

reg [0:15]instr; //80 bits of an instruction (maximum size of instruction is 10 bytes)

initial
begin
//for testing fetch_tb.v

//opq - add
instr_mem[0] = 8'b01100000;
instr_mem[1] = 8'b00010011;

//irmovq
instr_mem[2] = 8'b00110000;
instr_mem[3] = 8'b11110011;
instr_mem[4] = 8'b00000000; //constant - 8 bytes
instr_mem[5] = 8'b00000000;
instr_mem[6] = 8'b00000000;
instr_mem[7] = 8'b00000000;
instr_mem[8] = 8'b00000000;
instr_mem[9] = 8'b00000010;
instr_mem[10] = 8'b00000000;
instr_mem[11] = 8'b00000100;

//pushq
instr_mem[12] = 8'b10100000;
instr_mem[13] = 8'b00111111;

//nop
instr_mem[14] = 8'b00010000;

//opq
instr_mem[15]=8'b01100000;
instr_mem[16]=8'b00100010;

//jump-zf
instr_mem[17]=8'b01110011;
instr_mem[18]=8'b00000000; //constant - 8 bytes
instr_mem[19]=8'b00000000;
instr_mem[20]=8'b00000000;
instr_mem[21]=8'b00000000;
instr_mem[22]=8'b00000000;
instr_mem[23]=8'b00000000;
instr_mem[24]=8'b00000010;
instr_mem[25]=8'b00000000;

//cmov-zf
instr_mem[26]=8'b00100011;
instr_mem[27]=8'b00100010;

//cmov-
instr_mem[28]=8'b00100010;
instr_mem[29]=8'b00100010;

/*
//call
instr_mem[30]=8'b10000000;
instr_mem[31]=8'b00000000; //constant
instr_mem[32]=8'b00000000;
instr_mem[33]=8'b00000000;
instr_mem[34]=8'b00000000;
instr_mem[35]=8'b00000000;
instr_mem[36]=8'b00000000;
instr_mem[37]=8'b00000010;
instr_mem[38]=8'b00000010;
*/

//mrmov
```

```
instr_mem[30]=8'b01010000;
instr_mem[31]=8'b00010000;
instr_mem[32]=8'b00000000; //constant
instr_mem[33]=8'b00000000;
instr_mem[34]=8'b00000000;
instr_mem[35]=8'b00000000;
instr_mem[36]=8'b00000000;
instr_mem[37]=8'b00000000;
instr_mem[38]=8'b00000000;
instr_mem[39]=8'b00000010;

//halt
instr_mem[40]=8'b00000000;

//rmmov
instr_mem[41]=8'b01000000;
instr_mem[42]=8'b00010011;
instr_mem[43]=8'b00000000; //constant
instr_mem[44]=8'b00000000;
instr_mem[45]=8'b00000000;
instr_mem[46]=8'b00000000;
instr_mem[47]=8'b00000000;
instr_mem[48]=8'b00000000;
instr_mem[49]=8'b00000000;
instr_mem[50]=8'b00000100;

//popq
instr_mem[51]=8'b10110000;
instr_mem[52]=8'b00011111;

//call

//ret


end

always@(posedge clk)
begin
    in_mem = 1'b0; //invalid memory = 0
    in_inst = 1'b0;
    hlt = 1'b0;

    if(PC > 1023)
    begin
        //this is an invalid instruction
        in_mem = 1'b1;
    end

    //every instruction will be (PC + 10) max length
    instr = {instr_mem[PC],instr_mem[PC+1]};

    icode = instr[0:3];
    ifun = instr[4:7];

    case(icode)
    4'b0000:
    begin
        hlt = 1'b1; //halt
        valP = PC + 64'd1; //next instruction is at PC + 1
    end

    4'b0001:
    begin
    valP = PC + 64'd1; //nop
    hlt = 1'b0;
    end

    4'b0010:
    begin
    //cmovxx or rrmovq
    rA = instr[8:11]; //first 4 bits of register byte
    rB = instr[12:15];
```

```
valP = PC + 64'd2;
hlt = 1'b0;
end

4'b0011:
//irmov
begin
rA = instr[8:11]; //will be F because it is constant
rB = instr[12:15];
valC = {instr_mem[PC+2],instr_mem[PC+3],instr_mem[PC+4],instr_mem[PC+5],instr_mem[PC+6],instr_mem[PC+7],instr_mem[PC+8],in
valP = PC + 64'd10;
hlt = 1'b0;
end

4'b0011:
//irmov
begin
rA = instr[8:11];
rB = instr[12:15];
valC = {instr_mem[PC+2],instr_mem[PC+3],instr_mem[PC+4],instr_mem[PC+5],instr_mem[PC+6],instr_mem[PC+7],instr_mem[PC+8],in
valP = PC + 64'd10;
hlt = 1'b0;
end

4'b0100:
//rmmov
begin
rA = instr[8:11];
rB = instr[12:15];
valC = {instr_mem[PC+2],instr_mem[PC+3],instr_mem[PC+4],instr_mem[PC+5],instr_mem[PC+6],instr_mem[PC+7],instr_mem[PC+8],in
valP = PC + 64'd10;
hlt = 1'b0;
end

4'b0101:
//mrmov
begin
rA = instr[8:11];
rB = instr[12:15];
valC = {instr_mem[PC+2],instr_mem[PC+3],instr_mem[PC+4],instr_mem[PC+5],instr_mem[PC+6],instr_mem[PC+7],instr_mem[PC+8],in
valP = PC + 64'd10;
hlt = 1'b0;
end

4'b0110:
//op
begin
rA = instr[8:11];
rB = instr[12:15];
valP = PC + 64'd2;
hlt = 1'b0;
end

4'b0111:
//jXX
begin
valC = {instr_mem[PC+1],instr_mem[PC+2],instr_mem[PC+3],instr_mem[PC+4],instr_mem[PC+5],instr_mem[PC+6],instr_mem[PC+7],in
valP = PC + 64'd9;
hlt = 1'b0;
end

4'b1000:
//call
begin
valC = {instr_mem[PC+1],instr_mem[PC+2],instr_mem[PC+3],instr_mem[PC+4],instr_mem[PC+5],instr_mem[PC+6],instr_mem[PC+7],in
valP = PC + 64'd9;
hlt = 1'b0;
end

4'b1001:
begin
//ret
valP = PC + 64'd1;
```

```
    hlt = 1'b0;
    end

    4'b1001:
    begin
    //ret
    valP = PC + 64'd1;
    hlt = 1'b0;
    end

    4'b1010:
    begin
        //push
        rA = instr[8:11];
        rB = instr[12:15];
        valP = PC + 64'd2;
        hlt = 1'b0;
    end

    4'b1011:
    begin
        //push
        rA = instr[8:11];
        rB = instr[12:15];
        valP = PC + 64'd2;
        hlt = 1'b0;
    end

    default:
    begin
    in_inst = 1'b1;//the instruction is invalid
    hlt = 1'b0;
    end
    endcase
end
endmodule
```

- The fetch code above can be summarized as follows:
  - The instruction memory values are manually set. These were also used for testing the code.
  - There are three one-bit flags:
    - in_mem: denotes invalid memory
    - in_inst: denotes invalid instruction
    - hlt: for the halt instruction
  - At every positive edge of the clock, a ten-bit instruction is fetched from the instruction memory, using the PC value. Ten bytes are fetched because that's the maximum length of any instruction.
  - The icode value is set as the first four bits of the instruction, and the ifun value is set as the next four bits.
  - After this, based on the icode and ifun values, the rA, rB, and valC values are set for each instruction being fetched from memory in case statements.
  - valP is set as PC + 1 for halt, ret, and nop; PC + 2 for cmovxx, opq and popq; and PC + 10 for all other instructions.

This way, fetch is implemented in SEQ. Here are the results from testing the fetch stage:

```
clk=0  PC =                   0  icode=xxxx  ifun=xxxx  rA=xxxx  rB=xxxx  valC=                  x  valP=
clk=1  PC =                   0  icode=0110  ifun=0000  rA=0001  rB=0011  valC=                  x  valP=
clk=0  PC =                   0  icode=0110  ifun=0000  rA=0001  rB=0011  valC=                  x  valP=
clk=1  PC =                   2  icode=0011  ifun=0000  rA=1111  rB=0011  valC=             131076  valP=                1
clk=0  PC =                   2  icode=0011  ifun=0000  rA=1111  rB=0011  valC=             131076  valP=                1
clk=1  PC =                  12  icode=1010  ifun=0000  rA=0011  rB=1111  valC=             131076  valP=                1
```

```
clk=0  PC =                  12  icode=1010  ifun=0000  rA=0011  rB=1111  valC=             131076  valP=                  1
clk=1  PC =                  14  icode=0001  ifun=0000  rA=0011  rB=1111  valC=             131076  valP=                  1
clk=0  PC =                  14  icode=0001  ifun=0000  rA=0011  rB=1111  valC=             131076  valP=                  1
clk=1  PC =                  15  icode=0110  ifun=0000  rA=0010  rB=0010  valC=             131076  valP=                  1
clk=0  PC =                  15  icode=0110  ifun=0000  rA=0010  rB=0010  valC=             131076  valP=                  1
clk=1  PC =                  17  icode=0111  ifun=0011  rA=0010  rB=0010  valC=                512  valP=                  2
clk=0  PC =                  17  icode=0111  ifun=0011  rA=0010  rB=0010  valC=                512  valP=                  2
clk=1  PC =                  26  icode=0010  ifun=0011  rA=0010  rB=0010  valC=                512  valP=                  2
clk=0  PC =                  26  icode=0010  ifun=0011  rA=0010  rB=0010  valC=                512  valP=                  2
clk=1  PC =                  28  icode=0010  ifun=0010  rA=0010  rB=0010  valC=                512  valP=                  3
clk=0  PC =                  28  icode=0010  ifun=0010  rA=0010  rB=0010  valC=                512  valP=                  3
clk=1  PC =                  30  icode=0101  ifun=0000  rA=0001  rB=0000  valC=                  2  valP=                  4
clk=0  PC =                  30  icode=0101  ifun=0000  rA=0001  rB=0000  valC=                  2  valP=                  4
clk=1  PC =                  40  icode=0100  ifun=0000  rA=0001  rB=0011  valC=                  4  valP=                  5
clk=0  PC =                  40  icode=0100  ifun=0000  rA=0001  rB=0011  valC=                  4  valP=                  5
clk=1  PC =                  50  icode=1000  ifun=0000  rA=0001  rB=0011  valC=                192  valP=                  5
clk=0  PC =                  50  icode=1000  ifun=0000  rA=0001  rB=0011  valC=                192  valP=                  5
clk=1  PC =                  59  icode=1011  ifun=0000  rA=0001  rB=1111  valC=                192  valP=                  6
clk=0  PC =                  59  icode=1011  ifun=0000  rA=0001  rB=1111  valC=                192  valP=                  6
clk=1  PC =                  61  icode=0000  ifun=0000  rA=0001  rB=1111  valC=                192  valP=                  6
clk=0  PC =                  61  icode=0000  ifun=0000  rA=0001  rB=1111  valC=                192  valP=                  6
clk=1  PC =                  62  icode=xxxx  ifun=xxxx  rA=0001  rB=1111  valC=                192  valP=                  6
clk=0  PC =                  62  icode=xxxx  ifun=xxxx  rA=0001  rB=1111  valC=                192  valP=                  6
clk=1  PC =                  62  icode=xxxx  ifun=xxxx  rA=0001  rB=1111  valC=                192  valP=                  6
```

2. Decode

In this stage, the values of the operands are read from the register file. There is only need of a decode stage for instructions that require operands to be executed, i.e., opq, move instructions, call, ret, push and pop.

Here is the SEQ implementation of Decode:

```
`timescale 1ns/10ps

module decode(clk, icode, rA, rB, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, valA, valB);

//IMP -> you can't input a 2D array like reg_file to a module and so we input each register value on its own

input clk;
input [3:0]icode;
//input ifun[3:0];
input [3:0]rA;
input [3:0]rB;

output reg signed [63:0] r1;
output reg signed [63:0] r2;
output reg signed [63:0] r3;
output reg signed [63:0] r4;
output reg signed [63:0] r5;
output reg signed [63:0] r6;
output reg signed [63:0] r7;
output reg signed [63:0] r8;
output reg signed [63:0] r9;
output reg signed [63:0] r10;
output reg signed [63:0] r11;
output reg signed [63:0] r12;
output reg signed [63:0] r13;
output reg signed [63:0] r14;
output reg signed [63:0] r0;

output reg signed [63:0]valA;
output reg signed [63:0]valB;

reg signed [63:0]reg_file [0:14]; //15 reigsters of size 64 bits

//for the purpose of testing testbench
initial
begin
reg_file[0] = 64'd5;
```

```
reg_file[1] = 64'd45;
reg_file[2] = -64'd90;
reg_file[3] = 64'd54;
reg_file[4] = 64'd32;  //stack pointer register
reg_file[5] = 64'd21;
reg_file[6] = 64'd56;
reg_file[7] = 64'd33;
reg_file[8] = -64'd77;
reg_file[9] = 64'd0;
reg_file[10] = 64'd34;
reg_file[11] = 64'd7;
reg_file[12] = 64'd5;
reg_file[13] = -64'd9;
reg_file[14] = 64'd16;
end

always @(*) //we need to do this at * and not just at posedge of clk because we need to make sure that the value of rA is firs
begin
  case(icode)
    //halt - 4'b0000
    //nop - 4'b0001
    //cmovxx
    4'b0010:
    begin
        valA = reg_file[rA];
    end

    //irmovq - 4'b0011
    //rmmovq
    4'b0100:
    begin
        valA = reg_file[rA];
        valB = reg_file[rB];
    end

    //mrmovq
    4'b0101:
    begin
        valB = reg_file[rB];
    end

    //opq
    4'b0110:
    begin
        valA = reg_file[rA];
        valB = reg_file[rB];
    end

    //jxx - 4'b0111
    //call
    4'b1000:
    begin
        valB = reg_file[4]; //register 4 is the stack pointer register (rsp)
    end

    //ret
    4'b1001:
    begin
        valA = reg_file[4];
        valB = reg_file[4];
    end

    //pushq
    4'b1010:
    begin
        valA = reg_file[rA];
        valB = reg_file[4];
    end

    //popq
    4'b1011:
    begin
        valA = reg_file[4];
```

```
        valB = reg_file[4];
    end
    endcase

    r0 = reg_file[0];
    r1 = reg_file[1];
    r2 = reg_file[2];
    r3 = reg_file[3];
    r4 = reg_file[4]; //stack pointer register
    r5 = reg_file[5];
    r6 = reg_file[6];
    r7 = reg_file[7];
    r8 = reg_file[8];
    r9 = reg_file[9];
    r10 = reg_file[10];
    r11 = reg_file[11];
    r12 = reg_file[12];
    r13 = reg_file[13];
    r14 = reg_file[14];
end

endmodule
```

- The code implemented above can be summarized in the following manner:
  - The register memory, reg_file is manually defined (for testing purposes). reg_file[4] is the stack pointer register, %rsp.
  - The operand values are being set as valA and valB. This is because the values of the addreses rA and rB need to be fetched first (in the Fetch stage) which happens in the positive edge of the clock, and the values of valA and valB can only be set after those values are fetched.
  - Therefore, based on the icode and ifun values, the values of valA and valB are set for each instruction:
    - for cmovxx, valA is reg_file[rA]
    - for rmmovq, valA is reg_file[rA], and valB is reg_file[rB]
    - for mrmovq, valB is reg_file[rB]
    - for opq, valA is reg_file[rA], valB is reg_file[rB]
    - for call, valB is reg_file[4]
    - for ret, valA and valB are both reg_file[4]
    - for pushq, valA is reg_file[rA], and valB is reg_file[4]
    - for popq, valA and valB are both reg_file[4]

And so, Decode stage is implemented in SEQ. After testing, these are the obtained results:

```
clk=0 icode=xxxx rA=xxxx rB=xxxx valA=                  x valB=                  x
 r0=                    5 r1=                   45 r2=                  -90 r3=                   54 r4=                   32 r5=
 r8=                  -77 r9=                    0 r10=                   34 r11=                    7 r12=                    5 r1
clk=1 icode=0110 rA=0001 rB=0011 valA=                   45 valB=                   54
 r0=                    5 r1=                   45 r2=                  -90 r3=                   54 r4=                   32 r5=
 r8=                  -77 r9=                    0 r10=                   34 r11=                    7 r12=                    5 r1
clk=0 icode=0110 rA=0001 rB=0011 valA=                   45 valB=                   54
 r0=                    5 r1=                   45 r2=                  -90 r3=                   54 r4=                   32 r5=
 r8=                  -77 r9=                    0 r10=                   34 r11=                    7 r12=                    5 r1
clk=1 icode=0011 rA=1111 rB=0011 valA=                   45 valB=                   54
 r0=                    5 r1=                   45 r2=                  -90 r3=                   54 r4=                   32 r5=
 r8=                  -77 r9=                    0 r10=                   34 r11=                    7 r12=                    5 r1
clk=0 icode=0011 rA=1111 rB=0011 valA=                   45 valB=                   54
 r0=                    5 r1=                   45 r2=                  -90 r3=                   54 r4=                   32 r5=
 r8=                  -77 r9=                    0 r10=                   34 r11=                    7 r12=                    5 r1
clk=1 icode=1010 rA=0011 rB=1111 valA=                   54 valB=                   32
```

```
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=1010 rA=0011 rB=1111 valA=                54 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=0001 rA=0011 rB=1111 valA=                54 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=0001 rA=0011 rB=1111 valA=                54 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=0110 rA=0010 rB=0010 valA=               -90 valB=               -90
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=0110 rA=0010 rB=0010 valA=               -90 valB=               -90
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=0111 rA=0010 rB=0010 valA=               -90 valB=               -90
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=0111 rA=0010 rB=0010 valA=               -90 valB=               -90
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=0010 rA=0010 rB=0010 valA=               -90 valB=               -90
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=0010 rA=0010 rB=0010 valA=               -90 valB=               -90
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=0010 rA=0010 rB=0010 valA=               -90 valB=               -90
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=0010 rA=0010 rB=0010 valA=               -90 valB=               -90
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=0101 rA=0001 rB=0000 valA=               -90 valB=                5
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=0101 rA=0001 rB=0000 valA=               -90 valB=                5
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=0100 rA=0001 rB=0011 valA=                45 valB=                54
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=0100 rA=0001 rB=0011 valA=                45 valB=                54
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=1000 rA=0001 rB=0011 valA=                45 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=1000 rA=0001 rB=0011 valA=                45 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=1011 rA=0001 rB=1111 valA=                32 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=1011 rA=0001 rB=1111 valA=                32 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=0000 rA=0001 rB=1111 valA=                32 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=0000 rA=0001 rB=1111 valA=                32 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=xxxx rA=0001 rB=1111 valA=                32 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=0 icode=xxxx rA=0001 rB=1111 valA=                32 valB=                32
 r0=                  5 r1=                 45 r2=                -90 r3=                 54 r4=                 32 r5=
 r8=                -77 r9=                  0 r10=                 34 r11=                  7 r12=                  5 r1
clk=1 icode=xxxx rA=0001 rB=1111 valA=                32 valB=                32
```

```
 r0=                   5 r1=                  45 r2=                 -90 r3=                  54 r4=                  32 r5=
 r8=                 -77 r9=                   0 r10=                  34 r11=                   7 r12=                   5 r1
```

3. Execute

This stage sees the execution of the functions performed by each instruction with operands, for example, for the opq instruction the ALU is used to execute logical and arithmetic operations. Here is the code used for the Execute stage:

```
`timescale 1ns/10ps

`include "../alu/alu.v"

module execute(clk, icode, ifun, valA, valB, valC, cond, valE, ZF, SF, OF);

input clk;
input [3:0]icode;
input [3:0]ifun;
input signed [63:0]valA;
input signed [63:0]valB;
input signed [63:0]valC;
output reg ZF;      //condition codes
output reg SF;
output reg OF;
output reg cond;    //based on ifun and CCs
output reg signed [63:0]valE;
//output reg [63:0]alu_out;

//alu
reg signed [63:0]P;
reg  signed [63:0]Q;
reg [2:0]select;
wire signed carryout;
wire signed [63:0]Z;
reg signed [63:0]vale;

initial
begin
    //initialising alu variables
    select = 2'b00;
    P = 64'b0;
    Q = 64'b0;
end

//this only happens when the instruction is op ie the condition codes are only set when op is called
always @(*)
begin
    if(icode == 4'b0110)
    ZF = 0;
    OF = 0;
    SF = 0;
    begin
        if (Z == 1'b0) begin
            ZF = 1;
        end
        if (Z[63] == 1'b1) begin
            SF = 1;
        end
        if (((P < 64'b0) == (Q < 64'b0)) && ((Z<64'b0) != (P<64'b0))) begin
            OF = 1;
        end
    end
end

alu alu_alu(P, Q, Z, carryout, select);

always @(*)
begin
    cond = 0;
    case(icode)
```

```
        //halt - 4'b0000
        //nop - 4'b0001

        //cmovxx
        4'b0010:
        begin
            valE = valA;
            case(ifun)
                //rrmovq
                4'b0000:
                begin
                    cond = 1;
                end
                //cmovle
                4'b0001:
                begin
                    if ((SF ^ OF) || (ZF)) begin
                        cond = 1;
                    end
                end
                //cmovl
                4'b0010:
                begin
                    if (SF ^ OF) begin
                       cond = 1;
                    end
                end
                //cmove
                4'b0011:
                begin
                    if (ZF) begin
                        cond = 1;
                    end
                end
                //cmovne
                4'b0100:
                begin
                    if (~ZF) begin
                        cond = 1;
                    end
                end
                //cmovge
                4'b0101:
                begin
                    if (~(SF ^ OF)) begin
                        cond = 1;
                    end
                end
                //cmovg
                4'b0110:
                begin
                    if ((~(SF ^ OF)) && (~ZF)) begin
                        cond = 1;
                    end
                end
            endcase
        end
        //irmovq
        4'b0011:
        begin
            valE = valC;
        end
        //rmmovq
        4'b0100:
        begin
            valE = valB + valC;
        end
        //mrmovq
        4'b0101:
        begin
            valE = valB + valC;
        end
        //opq
```

```
        4'b0110:
        begin
            case(ifun)
                //addq
                4'b0000:
                begin
                    select = 2'b10;
                    P = valA;
                    Q = valB;
                end
                //subq
                4'b0001:
                begin
                    select = 2'b11;
                    P = valA;
                    Q = valB;
                end
                //andq
                4'b0010:
                begin
                    select = 2'b00;
                    P = valA;
                    Q = valB;
                end
                //xorq
                4'b0011:
                begin
                    select = 2'b01;
                    P = valA;
                    Q = valB;
                end
            endcase
        assign vale = Z; //cause Z is a wire
        valE = vale;
        end
        //jxx
        4'b0111:
        begin
            case(ifun)
                //jmp
                4'b0000:
                begin
                    cond = 1;
                end
                //jle
                4'b0001:
                begin
                    if ((SF ^ OF) || (ZF)) begin
                        cond = 1;
                    end
                end
                //jl
                4'b0010:
                begin
                    if (SF ^ OF) begin
                        cond = 1;
                    end
                end
                //je
                4'b0011:
                begin
                    if (ZF) begin
                        cond = 1;
                    end
                end
                //jne
                4'b0100:
                begin
                    if (~ZF) begin
                        cond = 1;
                    end
                end
                //jge
```

```
                4'b0101:
                begin
                    if(~(SF ^ OF)) begin
                        cond = 1;
                    end
                end
                //jg
                4'b0110:
                begin
                    if ((~(SF ^ OF)) && (~ZF)) begin
                        cond = 1;
                    end
                end
            endcase
        end
        //call
        4'b1000:
        begin
            valE = valB + (-64'd8);
        end
        //ret
        4'b1001:
        begin
            valE = valB + 64'd8;
        end
        //pushq
        4'b1010:
        begin
            valE = valB + (-64'd8);
        end
        //popq
        4'b1011:
        begin
            valE = valB + 64'd8;
        end
    endcase
end
endmodule
```

Here is what's happening in the above code:

- The ALU module is called, and the inputs and outputs are defined and initialized.
- The condition codes are set at every instant as follows:
    - ZF (Zero Flag) is set to 1 is the ALU output is 0, 0 otherwise
    - SF (Sign Flag) is set to 1 is the ALU output is a negative value, 0 otherwise
    - OF (Overflow Flag) is set to 1 if the ALU output is an overflow value, 0 otherwise
- For cmovxx, valE is set as valA. Based on the ifun values, the cond value is set as per ZF, SF and OF values.
- For irmovq, valE is set as valC.
- For rmmovq and mrmovq, valE is set as valB + valC.
- For the opq instruction, the ALU is used to execute andq, xorq, addq and subq instructions where valA and valB are the operands. valE is set as the ALU output.
- For jxx, each corresponding ifun value has a cond value set based on ZF, SF and OF.
- For call and pushq, valE is set to valB - 8.
- For ret and popq, valE is set to valB + 8.

Here is the output of the Execute stage:

```
clk=0 icode=xxxx ifun=xxxx valA=                  x valB=                  x valC=                  x cond=x valE=
clk=1 icode=0110 ifun=0000 valA=                 45 valB=                 54 valC=                  x cond=0 valE=
clk=0 icode=0110 ifun=0000 valA=                 45 valB=                 54 valC=                  x cond=0 valE=
clk=1 icode=0011 ifun=0000 valA=                 45 valB=                 54 valC=             131076 cond=0 valE=
clk=0 icode=0011 ifun=0000 valA=                 45 valB=                 54 valC=             131076 cond=0 valE=
clk=1 icode=1010 ifun=0000 valA=                 54 valB=                 32 valC=             131076 cond=0 valE=
clk=0 icode=1010 ifun=0000 valA=                 54 valB=                 32 valC=             131076 cond=0 valE=
clk=1 icode=0001 ifun=0000 valA=                 54 valB=                 32 valC=             131076 cond=0 valE=
clk=0 icode=0001 ifun=0000 valA=                 54 valB=                 32 valC=             131076 cond=0 valE=
clk=1 icode=0110 ifun=0000 valA=                -90 valB=                -90 valC=             131076 cond=0 valE=
clk=0 icode=0110 ifun=0000 valA=                -90 valB=                -90 valC=             131076 cond=0 valE=
clk=1 icode=0111 ifun=0011 valA=                -90 valB=                -90 valC=                512 cond=0 valE=
clk=0 icode=0111 ifun=0011 valA=                -90 valB=                -90 valC=                512 cond=0 valE=
clk=1 icode=0010 ifun=0011 valA=                -90 valB=                -90 valC=                512 cond=0 valE=
clk=0 icode=0010 ifun=0011 valA=                -90 valB=                -90 valC=                512 cond=0 valE=
clk=1 icode=0010 ifun=0010 valA=                -90 valB=                -90 valC=                512 cond=1 valE=
clk=0 icode=0010 ifun=0010 valA=                -90 valB=                -90 valC=                512 cond=1 valE=
clk=1 icode=0101 ifun=0000 valA=                -90 valB=                  5 valC=                  2 cond=0 valE=
clk=0 icode=0101 ifun=0000 valA=                -90 valB=                  5 valC=                  2 cond=0 valE=
clk=1 icode=0100 ifun=0000 valA=                 45 valB=                 54 valC=                  4 cond=0 valE=
clk=0 icode=0100 ifun=0000 valA=                 45 valB=                 54 valC=                  4 cond=0 valE=
clk=1 icode=1000 ifun=0000 valA=                 45 valB=                 32 valC=                192 cond=0 valE=
clk=0 icode=1000 ifun=0000 valA=                 45 valB=                 32 valC=                192 cond=0 valE=
clk=1 icode=1011 ifun=0000 valA=                 32 valB=                 32 valC=                192 cond=0 valE=
clk=0 icode=1011 ifun=0000 valA=                 32 valB=                 32 valC=                192 cond=0 valE=
clk=1 icode=0000 ifun=0000 valA=                 32 valB=                 32 valC=                192 cond=0 valE=
clk=0 icode=0000 ifun=0000 valA=                 32 valB=                 32 valC=                192 cond=0 valE=
clk=1 icode=xxxx ifun=xxxx valA=                 32 valB=                 32 valC=                192 cond=0 valE=
clk=0 icode=xxxx ifun=xxxx valA=                 32 valB=                 32 valC=                192 cond=0 valE=
clk=1 icode=xxxx ifun=xxxx valA=                 32 valB=                 32 valC=                192 cond=0 valE=
```

4. Memory

In this stage data is written to memory, or data is read from memory. The output of this stage is valM, which is the data read from memory. The icode value can be used to determine whether data is read from or written to memory.

Here is the code implemented for the Memory stage:

```
`timescale 1ns/10ps

module memory(clk,icode,valA,valB,valE,valP,valM,data);

//[63:0] after regA makes it an array but putting it before means its a net
input clk;
input [3:0]icode; //first 4 bits of the first byte
input [63:0]valP; //stores the address of next instruction (PC should be this value)
input signed [63:0]valA;
input signed [63:0]valB;
input signed [63:0]valE;

output reg [63:0]valM;
output reg [63:0]data;

//setting up data memory (similar structure to instruction memory)
reg [63:0] data_mem[0:1023]; //this means we have a 1024 rows and each data is 64 bits

initial
begin
//for testing memory_tb.v (giving values based only on what will be computed)
data_mem[7]=64'd24;
data_mem[24] = 64'd17;
data_mem[32] = 64'd88;
data_mem[47] = 64'd55;
end

always @(*)
begin
    case(icode)
```

```
    4'b0100:
    //rmmov
    begin
        data_mem[valE] = valA;
    end

    4'b0101:
    //mrmov
    begin
        valM = data_mem[valE];
    end

    4'b1000:
    //call
    begin
        data_mem[valE] = valP;
    end

    4'b1001:
    //ret
    begin
        valM = data_mem[valA];
    end

    4'b1010:
    //push
    begin
        data_mem[valE] = valA;
    end

    4'b1011:
    //pop
    begin
        valM = data_mem[valA];
    end
    endcase
    data = data_mem[valE];
end
endmodule
```

As we can see, in this stage:

- For instructions rmmovq, call, and push, data is written to memory.
  - for call, data_mem[valE] was set as valP
  - for rmmovq and push, data_mem[valE] was set as valA
- For instructions mrmovq, ret and pop, data is read from memory.
  - For ret, valM was set as data_mem[valE]
  - For mrmovq and pop, valM was set as data_mem[valA]

Here is the output of the memory stage:

```
clk=0 icode=xxxx valA=                 x valB=                 x valE=                 x valP=                 x valM=
clk=1 icode=0110 valA=                45 valB=                54 valE=                99 valP=                 2 valM=
clk=0 icode=0110 valA=                45 valB=                54 valE=                99 valP=                 2 valM=
clk=1 icode=0011 valA=                45 valB=                54 valE=            131076 valP=                12 valM=
clk=0 icode=0011 valA=                45 valB=                54 valE=            131076 valP=                12 valM=
clk=1 icode=1010 valA=                54 valB=                32 valE=                24 valP=                14 valM=
clk=0 icode=1010 valA=                54 valB=                32 valE=                24 valP=                14 valM=
clk=1 icode=0001 valA=                54 valB=                32 valE=                24 valP=                15 valM=
clk=0 icode=0001 valA=                54 valB=                32 valE=                24 valP=                15 valM=
clk=1 icode=0110 valA=               -90 valB=               -90 valE=              -180 valP=                17 valM=
```

```
clk=0 icode=0110 valA=                -90 valB=                -90 valE=               -180 valP=                 17 valM=
clk=1 icode=0111 valA=                -90 valB=                -90 valE=               -180 valP=                 26 valM=
clk=0 icode=0111 valA=                -90 valB=                -90 valE=               -180 valP=                 26 valM=
clk=1 icode=0010 valA=                -90 valB=                -90 valE=                -90 valP=                 28 valM=
clk=0 icode=0010 valA=                -90 valB=                -90 valE=                -90 valP=                 28 valM=
clk=1 icode=0010 valA=                -90 valB=                -90 valE=                -90 valP=                 30 valM=
clk=0 icode=0010 valA=                -90 valB=                -90 valE=                -90 valP=                 30 valM=
clk=1 icode=0101 valA=                -90 valB=                  5 valE=                  7 valP=                 40 valM=
clk=0 icode=0101 valA=                -90 valB=                  5 valE=                  7 valP=                 40 valM=
clk=1 icode=0100 valA=                 45 valB=                 54 valE=                 58 valP=                 50 valM=
clk=0 icode=0100 valA=                 45 valB=                 54 valE=                 58 valP=                 50 valM=
clk=1 icode=1000 valA=                 45 valB=                 32 valE=                 24 valP=                 59 valM=
clk=0 icode=1000 valA=                 45 valB=                 32 valE=                 24 valP=                 59 valM=
clk=1 icode=1011 valA=                 32 valB=                 32 valE=                 40 valP=                 61 valM=
clk=0 icode=1011 valA=                 32 valB=                 32 valE=                 40 valP=                 61 valM=
clk=1 icode=0000 valA=                 32 valB=                 32 valE=                 40 valP=                 62 valM=
clk=0 icode=0000 valA=                 32 valB=                 32 valE=                 40 valP=                 62 valM=
clk=1 icode=xxxx valA=                 32 valB=                 32 valE=                 40 valP=                 62 valM=
clk=0 icode=xxxx valA=                 32 valB=                 32 valE=                 40 valP=                 62 valM=
clk=1 icode=xxxx valA=                 32 valB=                 32 valE=                 40 valP=                 62 valM=
```

5. Write Back

In this stage, one or two register values are written to the register file. Here is the implementation of the Write Back stage:

```
`timescale 1ns/10ps

module writeback(clk,icode,cond,rA,rB,valE,valM,r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14);

//[63:0] after regA makes it an array but putting it before means its a net
input clk;
input [3:0]icode; //first 4 bits of the first byte
input signed [63:0]valM;
input signed [63:0]valE;
input [3:0]rA;
input [3:0]rB;
input cond;

output reg signed [63:0] r1;
output reg signed [63:0] r2;
output reg signed [63:0] r3;
output reg signed [63:0] r4;
output reg signed [63:0] r5;
output reg signed [63:0] r6;
output reg signed [63:0] r7;
output reg signed [63:0] r8;
output reg signed [63:0] r9;
output reg signed [63:0] r10;
output reg signed [63:0] r11;
output reg signed [63:0] r12;
output reg signed [63:0] r13;
output reg signed [63:0] r14;
output reg signed [63:0] r0;

reg signed [63:0]reg_file [0:14]; //15 reigsters of size 64 bits

//for the purpose of testing testbench
initial
begin
reg_file[0] = 64'd5;
reg_file[1] = 64'd45;
reg_file[2] = -64'd90;
reg_file[3] = 64'd54;
reg_file[4] = 64'd32;  //stack pointer register
reg_file[5] = 64'd21;
reg_file[6] = 64'd56;
```

```
reg_file[7] = 64'd33;
reg_file[8] = -64'd77;
reg_file[9] = 64'd0;
reg_file[10] = 64'd34;
reg_file[11] = 64'd7;
reg_file[12] = 64'd5;
reg_file[13] = -64'd9;
reg_file[14] = 64'd16;
end

always @(*)
begin
    case(icode)
    //cmovxx
    4'b0010:
    begin
        if(cond)
        begin
            reg_file[rB] = valE;
        end
    end

    //irmov
    4'b0011:
    begin
        reg_file[rB] = valE;
    end

    //mrmov
    4'b0101:
    begin
        reg_file[rA] = valM;
    end

    //opq
    4'b0110:
    begin
        reg_file[rB] = valE;
    end

    //call
    4'b1000:
    begin
        reg_file[4] = valE;
    end

    //ret
    4'b1001:
    begin
        reg_file[4] = valE;
    end

    //push
    4'b1010:
    begin
        reg_file[4] = valE;
    end

    //pop
    4'b1011:
    begin
        reg_file[4] = valE;
        reg_file[rA] = valM;
    end
    endcase

    r0 = reg_file[0];
    r1 = reg_file[1];
    r2 = reg_file[2];
    r3 = reg_file[3];
    r4 = reg_file[4]; //stack pointer register
    r5 = reg_file[5];
    r6 = reg_file[6];
```

```
    r7 = reg_file[7];
    r8 = reg_file[8];
    r9 = reg_file[9];
    r10 = reg_file[10];
    r11 = reg_file[11];
    r12 = reg_file[12];
    r13 = reg_file[13];
    r14 = reg_file[14];
end


endmodule
```

Based on the icode value, the following occurs in the Write Back stage:

- For cmovxx, if cond (which was set in the decode stage) is 1, then reg_file[rB] is set as valE
- For irmov and opq, reg_file[rB] is set as valE
- For mrmov, reg_file[rA] is set as valM
- For call, ret, and push, reg_file[4] is set as valE
- For pop, reg_file[4] s set as valE and reg_file[rA] is set as valM

As this is the only stage apart from decode that accesses memory, both these stages have been implemented together in the SEQ implementation, and the Write Back output is visible there, along with the decode output.

6. PC Update

In this stage, the PC value is updated, so as to access the address of the next instruction. The output is the updated PC value, newPC. Here is the code used to implement this stage:

```
`timescale 1ns/10ps

module pc_update(clk, newPC, icode, valP, valM, valC, cond);

input clk;
input [3:0]icode;
input [63:0]valC;
input [63:0]valP;
input [63:0]valM;
input cond;
output reg [63:0]newPC;

always @(*)
begin
    //ret
    if (icode == 4'b1001) begin
        newPC = valM;
    end
    //call
    else if (icode == 4'b1000) begin
        newPC = valC;
    end
    //jxx
    else if (icode == 4'b0111) begin
        if (cond == 1) begin
            newPC = valC;
        end
        else begin
            newPC = valP;
        end
    end
    else
    begin
        //every other instruction will be this
        newPC = valP;
```

```
    end
end
endmodule
```

In this stage,

- for ret, newPC is set as valM
- for call, newPC is set as valC
- for jxx, if cond is 1, the newPC is set as valC, or else its set as valP
- for all other instructions, newPC is set as valP

The output of this stage is the final SEQ output.

Final sequential implementation

The code for the final test bench is as shown below:

```
`timescale 1ns/10ps
`include "fetch.v"
`include "decode.v"
`include "execute.v"
`include "memory.v"
`include "writeback.v"
`include "pc_update.v"

module seq_tb;

reg clk;
reg [63:0]PC; //memory address is 64 bits
reg AOK,HALT,ADR,INS;
reg [2:0]status;

wire [3:0]icode; //first 4 bits of the first byte
wire [3:0]ifun; //last 4 bits of the first byte
wire [3:0]rA; //first 4 bits of the second byte
wire [3:0]rB; //last 4 bits of the second byte
wire signed [63:0]valC; //stores the constant word
wire [63:0]valP; //stores the address of next instruction (PC should be this value)
wire in_mem;
wire in_inst;
wire hlt;
wire signed [63:0]valA;
wire signed [63:0]valB;
wire cond;
wire signed [63:0]valE;
wire signed [63:0]valM;
wire signed [63:0]data;
wire ZF;
wire SF;
wire OF;
wire signed[63:0] r1;
wire signed[63:0] r2;
wire signed[63:0] r3;
wire signed[63:0] r4;
wire signed[63:0] r5;
wire signed[63:0] r6;
wire signed[63:0] r7;
wire signed[63:0] r8;
wire signed[63:0] r9;
wire signed[63:0] r10;
wire signed[63:0] r11;
wire signed[63:0] r12;
wire signed[63:0] r13;
wire signed[63:0] r14;
wire signed[63:0] r0;
wire [63:0]newPC;
```

```
fetch UUTA (clk,PC,icode,ifun,rA,rB,valC,valP,in_mem,in_inst,hlt);
decode UUTB (clk, icode, rA, rB, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, valA, valB);
execute UUTC (clk, icode, ifun, valA, valB, valC, cond, valE, ZF, SF, OF);
memory UUTD (clk, icode, valA, valB, valE, valP, valM, data);
writeback UUTE (clk, icode, cond, rA, rB, valE, valM, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14);
pc_update UUTF (clk, newPC, icode, valP, valM, valC, cond);

initial
begin
    $dumpfile("seq_tb.vcd");
    $dumpvars(0,seq_tb);

    AOK = 1;
    HALT = 0;
    ADR = 0;
    INS = 0;
    status = 3'b001;

    clk = 0;
    PC = 64'd0;
end

initial
begin
    #10 clk = ~clk; //positive edge
    #10 clk = ~clk; //negative edge so nothing happens
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk; //0
    #10 clk = ~clk;
    #10 clk = ~clk; //0
end

always@(*)
begin
    PC = newPC;

    //setting the status codes
    if(hlt)
    begin
        AOK = 0;
        HALT= 1;
        INS = 0;
        ADR = 0;
        status = 3'b010;
    end
    else if(in_mem)
    begin
        AOK = 0;
        HALT= 0;
        INS = 0;
        ADR = 1;
        status = 3'b011;
```

```
    end
    else if(in_inst)
    begin
        AOK = 0;
        HALT= 0;
        INS = 1;
        ADR = 0;
        status = 3'b100;
    end
    else
    begin
        AOK = 1;
        HALT = 0;
        INS = 0;
        ADR = 0;
        status = 3'b001;
    end
end

//always @(*)
initial
    $monitor("clk=%d icode=%b ifun=%b rA=%b rB=%b PC=%d cond=%d valA=%d valB=%d \nvalC=%d valE=%d valM=%d data=%d newPC=%d sta
endmodule
```

The output for the given instruction and data memory is given below:

```
clk=0 icode=xxxx ifun=xxxx rA=xxxx rB=xxxx PC=                  0 cond=x valA=                  x valB=                  x
valC=                    x valE=                    x valM=                    x data=                    x valP=
clk=1 icode=0110 ifun=0000 rA=0001 rB=0011 PC=                  x cond=0 valA=                 45 valB=                 54
valC=                    x valE=                   99 valM=                    x data=                    x valP=
clk=0 icode=0110 ifun=0000 rA=0001 rB=0011 PC=                  2 cond=0 valA=                 45 valB=                 99
valC=                    x valE=                   99 valM=                    x data=                    x valP=
clk=1 icode=0011 ifun=0000 rA=1111 rB=0011 PC=                  2 cond=0 valA=                 45 valB=                 99
valC=               131076 valE=               131076 valM=                    x data=                    x valP=
clk=0 icode=0011 ifun=0000 rA=1111 rB=0011 PC=                 12 cond=0 valA=                 45 valB=                 99
valC=               131076 valE=               131076 valM=                    x data=                    x valP=
clk=1 icode=1010 ifun=0000 rA=0011 rB=1111 PC=                 12 cond=0 valA=             131076 valB=                 32
valC=               131076 valE=                   24 valM=                    x data=             131076 valP=
clk=0 icode=1010 ifun=0000 rA=0011 rB=1111 PC=                 14 cond=0 valA=             131076 valB=                 24
valC=               131076 valE=                   24 valM=                    x data=             131076 valP=
clk=1 icode=0001 ifun=0000 rA=0011 rB=1111 PC=                 14 cond=0 valA=             131076 valB=                 24
valC=               131076 valE=                   16 valM=                    x data=                    x valP=
clk=0 icode=0001 ifun=0000 rA=0011 rB=1111 PC=                 15 cond=0 valA=             131076 valB=                 24
valC=               131076 valE=                   16 valM=                    x data=                    x valP=
clk=1 icode=0110 ifun=0000 rA=0010 rB=0010 PC=                 15 cond=0 valA=                -90 valB=                -90
valC=               131076 valE=                 -180 valM=                    x data=                    x valP=
clk=0 icode=0110 ifun=0000 rA=0010 rB=0010 PC=                 17 cond=0 valA=               -180 valB=               -180
valC=               131076 valE=                 -180 valM=                    x data=                    x valP=
clk=1 icode=0111 ifun=0011 rA=0010 rB=0010 PC=                 17 cond=0 valA=               -180 valB=               -180
valC=                  512 valE=                 -180 valM=                    x data=                    x valP=
clk=0 icode=0111 ifun=0011 rA=0010 rB=0010 PC=                 26 cond=0 valA=               -180 valB=               -180
valC=                  512 valE=                 -180 valM=                    x data=                    x valP=
clk=1 icode=0010 ifun=0011 rA=0010 rB=0010 PC=                 26 cond=0 valA=               -180 valB=               -180
valC=                  512 valE=                 -180 valM=                    x data=                    x valP=
clk=0 icode=0010 ifun=0011 rA=0010 rB=0010 PC=                 28 cond=0 valA=               -180 valB=               -180
valC=                  512 valE=                 -180 valM=                    x data=                    x valP=
clk=1 icode=0010 ifun=0010 rA=0010 rB=0010 PC=                 28 cond=1 valA=               -180 valB=               -180
valC=                  512 valE=                 -180 valM=                    x data=                    x valP=
clk=0 icode=0010 ifun=0010 rA=0010 rB=0010 PC=                 30 cond=1 valA=               -180 valB=               -180
valC=                  512 valE=                 -180 valM=                    x data=                    x valP=
clk=1 icode=0101 ifun=0000 rA=0001 rB=0000 PC=                 30 cond=0 valA=               -180 valB=                  5
valC=                    2 valE=                    7 valM=                   24 data=                   24 valP=
clk=0 icode=0101 ifun=0000 rA=0001 rB=0000 PC=                 40 cond=0 valA=               -180 valB=                  5
valC=                    2 valE=                    7 valM=                   24 data=                   24 valP=
clk=1 icode=0100 ifun=0000 rA=0001 rB=0011 PC=                 40 cond=0 valA=                 24 valB=             131076
valC=                    4 valE=               131080 valM=                   24 data=                    x valP=
clk=0 icode=0100 ifun=0000 rA=0001 rB=0011 PC=                 50 cond=0 valA=                 24 valB=             131076
valC=                    4 valE=               131080 valM=                   24 data=                    x valP=
```

```
clk=1 icode=1000 ifun=0000 rA=0001 rB=0011 PC=                50 cond=0 valA=                24 valB=                24
valC=                192 valE=                16 valM=                24 data=                59 valP=
clk=0 icode=1000 ifun=0000 rA=0001 rB=0011 PC=               192 cond=0 valA=                24 valB=                16
valC=                192 valE=                16 valM=                24 data=                59 valP=
clk=1 icode=0001 ifun=0000 rA=0001 rB=0011 PC=               192 cond=0 valA=                24 valB=                16
valC=                192 valE=                 8 valM=                24 data=                 x valP=
clk=0 icode=0001 ifun=0000 rA=0001 rB=0011 PC=               193 cond=0 valA=                24 valB=                16
valC=                192 valE=                 8 valM=                24 data=                 x valP=
clk=1 icode=1001 ifun=0000 rA=0001 rB=0011 PC=               193 cond=0 valA=                16 valB=                16
valC=                192 valE=                24 valM=                59 data=            131076 valP=
clk=0 icode=1001 ifun=0000 rA=0001 rB=0011 PC=                59 cond=0 valA=                24 valB=                24
valC=                192 valE=                24 valM=            131076 data=            131076 valP=
clk=1 icode=1011 ifun=0000 rA=0001 rB=1111 PC=                59 cond=0 valA=                24 valB=                24
valC=                192 valE=                32 valM=            131076 data=                88 valP=
clk=0 icode=1011 ifun=0000 rA=0001 rB=1111 PC=                61 cond=0 valA=                32 valB=                32
valC=                192 valE=                32 valM=                88 data=                88 valP=
```

The instructions have been executed sequentially and the branch and conditional instructions also works as expected.

The gtkwave output for the above inputs is:

Module II - Pipelined Design

The pipelined processor implementation has 5 stages, where the PC Update stage is merged with the Fetch stage. This allows for instructions to be executed in parallel, increasing the processor's throughput. However, due to dependencies between instructions, a hazard may occur where an instruction is dependent on the result of a previous instruction. This is addressed through forwarding, where the result of an instruction is forwarded to another instruction. In addition, stalls may occur to allow for dependencies to be resolved.

Stages of Processing

The pipelined architecture processor (PIPE) has the same six stages as the sequential processor - Fetch, Decode, Execute, Memory, Write-Back and PC-update.

The only difference is the introduction of pipeline registers that store values between different stages and addition of logic for data forwarding and PC prediction. Also, the PC update stage is moved to the beginning of the cycle.

Pipeline Registers

Pipeline registers store the value of signals between two stages, thus preventing them from flowing into the next stage.

- The Fetch register is inserted before the fetch stage and holds the value of the next predicted PC.
- The Decode register is present between the fetch and decode stage and stores information about the instruction that was just fetched.
- The Execute register comes before the execute stage and stores values from the most recently decoded instruction to be used by the execute stage

- The Memory register sits between execute and memory stages and holds information about branch conditions to be used in the memory stage.
- The Write-back register comes before the write-back stage and is very helpful in PC prediction because of the various feedback paths.

Thus, as is evident, the pipeline registers help separate the various stages and signals, enabling in the process of having parallel computations.

Relabelling signals

Since we are dealing with various stages simultaneously, it is important that we distinguish between the signals in the various stages.

- S_signal → value stored in the S pipeline register
- s_signal → value computed in the S stage

Eg - w_icode and W_icode

Data and Control Hazards

In a pipelined processor, there are often data and control hazards that occur because of data dependencies (result computed by one instruction is used in a following instruction) or control dependencies (where one instruction determines the location of the following instructions - like in the case of call, ret etc)

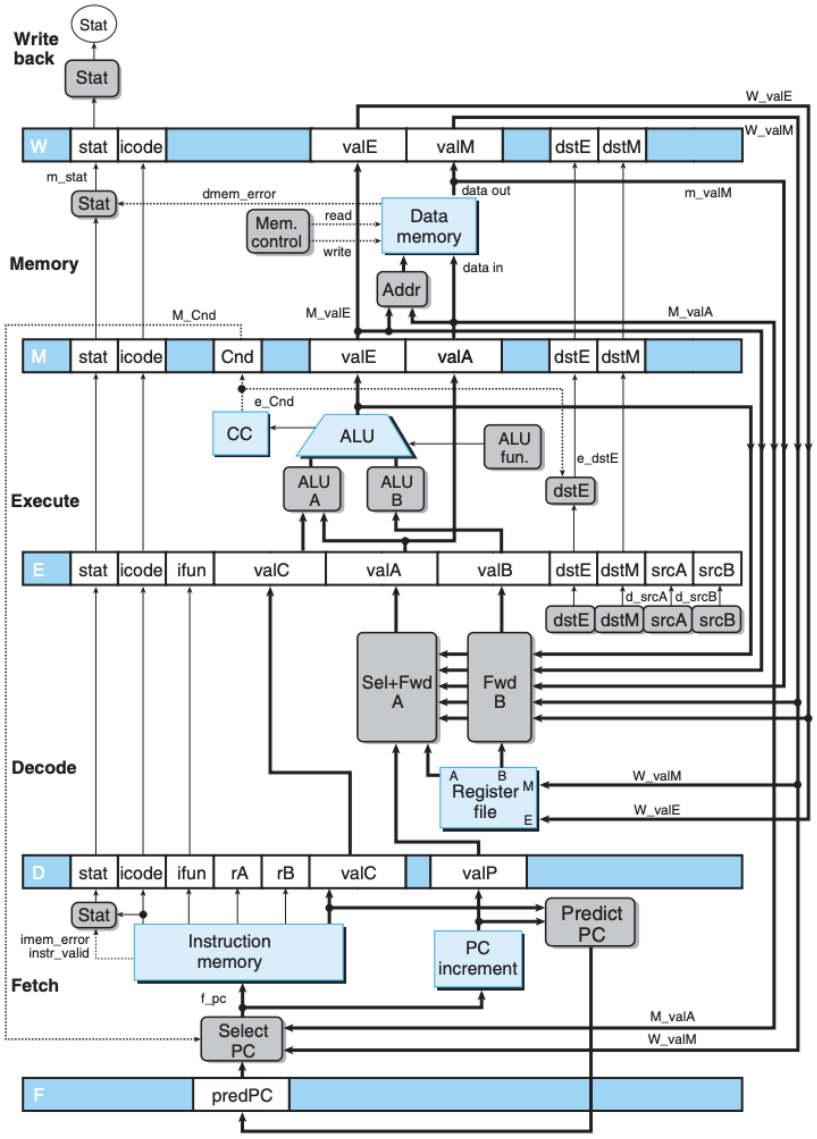
One method of overcoming this problem is stalling, where the processor holds back one or more instructions in a specific stage until the hazard is no longer valid. This is usually done by inserting a bubble into the instruction pipeline. A bubble is essentially a nop instruction which stalls the processor in a specific stage.

Another method of overcoming this problem is data forwarding, which involves passing the data which is available in a preceding stage instead of waiting till the write-back stage for the registers to get updated. This is better than the stalling method because it doesn’t cause delays in the processor pipeline.

Block Diagram

The below block diagram represents the final pipeline architecture with the various signals and feedback paths.

The architecture also includes data forwarding functionality which handles data hazards without stalling the pipeline.



1. Fetch

- In addition to the sequential fetch code, we have the fetch pipeline register which is used to update the PC.
- The rest of the fetch code remains the same as SEQ implementation.

```
// fetch reigster -> register before fetch stage -> gets value of the next PC
module fetch_reg (clk,nextPC,f_nextPC);
    input clk;
    input [63:0] nextPC;

    output reg [63:0] f_nextPC;

    always @(posedge clk)
    begin
        f_nextPC <= nextPC; //this is non blocking assignment meaning that all 64 bits will be assigned at the same time
```

```
    end
endmodule
```

2. Decode

- The decode register implementation is give below

```
//decode reigster -> present between the fetch and decode stage
module decode_reg(clk,f_icode,f_ifun,f_rA,f_rB,f_valC,f_valP,f_hlt,f_in_inst,f_in_mem,d_icode,d_ifun,d_rA,d_rB,d_valC,d_valP,d
    input clk;
    input [3:0] f_icode;
    input [3:0] f_ifun;
    input [3:0] f_rA;
    input [3:0] f_rB;
    input signed [63:0] f_valC;
    input signed [63:0] f_valP;
    input f_hlt;
    input f_in_inst;
    input f_in_mem;

    output reg [3:0] d_icode;
    output reg [3:0] d_ifun;
    output reg [3:0] d_rA;
    output reg [3:0] d_rB;
    output reg signed [63:0] d_valC;
    output reg signed [63:0] d_valP;
    output reg d_hlt;
    output reg d_in_inst;
    output reg d_in_mem;

    always @(posedge clk)
    begin
        d_icode <= f_icode;
        d_ifun <= f_ifun;
        d_rA <= f_rA;
        d_rB <= f_rB;
        d_valC <= f_valC;
        d_valP <= f_valP;
        d_hlt <= f_hlt;
        d_in_inst <= f_in_inst;
        d_in_mem <= f_in_mem;
    end
endmodule
```

3. Execute

```
module execute_reg(clk, e_icode, e_ifun, e_rA, e_rB, e_valA, e_valB, e_valC, e_valP, e_hlt, e_in_mem, e_in_inst, d_icode, d_if

input clk;

input [3:0]d_icode;
input [3:0]d_ifun;
input [3:0]d_rA;
input [3:0]d_rB;
input signed [63:0]d_valA;
input signed [63:0]d_valB;
input signed [63:0]d_valC;
input signed [63:0]d_valP;
input d_hlt;
input d_in_mem;
input d_in_inst;
```

```
output reg [3:0]e_icode;
output reg [3:0]e_ifun;
output reg[3:0]e_rA;
output reg[3:0]e_rB;
output reg signed [63:0]e_valA;
output reg signed [63:0]e_valB;
output reg signed [63:0]e_valC;
output reg signed [63:0]e_valP;
output reg e_hlt;
output reg e_in_mem;
output reg e_in_inst;

always @(posedge clk)
begin
    e_icode <= d_icode;
    e_ifun <= d_ifun;
    e_rA <= d_rA;
    e_rB <= d_rB;
    e_valA <= d_valA;
    e_valB <= d_valB;
    e_valC <= d_valC;
    e_valP <= d_valP;
    e_hlt <= d_hlt;
    e_in_mem <= d_in_mem;
    e_in_inst <= d_in_inst;
end

endmodule
```

4. Memory

```
module memory_reg(clk, m_icode, m_cond, m_rA, m_rB, m_valA, m_valE, m_valP, m_valC, m_hlt, m_in_mem, m_in_inst, e_icode, e_con
input clk;
input [3:0]e_icode;
input e_cond;
input [3:0]e_rA;
input [3:0]e_rB;
input signed [63:0]e_valA;
input signed [63:0]e_valE;
input signed [63:0]e_valP;
input signed [63:0]e_valC;
input e_hlt;
input e_in_mem;
input e_in_inst;

output reg [3:0]m_icode;
output reg m_cond;
output reg [3:0]m_rA;
output reg [3:0]m_rB;
output reg signed [63:0]m_valA;
output reg signed [63:0]m_valE;
output reg signed [63:0]m_valP;
output reg signed [63:0]m_valC;
output reg m_hlt;
output reg m_in_mem;
output reg m_in_inst;

always @(posedge clk)
begin
    m_icode <= e_icode;
    m_cond <= e_cond;
    m_rA <= e_rA;
    m_rB <= e_rB;
    m_valA <= e_valA;
    m_valE <= e_valE;
    m_valP <= e_valP;
    m_valC <= e_valC;
    m_hlt <= e_hlt;
```

```
    m_in_mem <= e_in_mem;
    m_in_inst <= e_in_inst;
end

endmodule
```

5. Write-back

```
//writeback reigster -> present between the memory and writeback stage
module writeback_reg(clk,m_icode,m_rA,m_rB,m_valE,m_valM,m_valC,m_cond,m_hlt,m_in_inst,m_in_mem,w_icode,w_rA,w_rB,w_valE,w_val
    input clk;
    input [3:0]m_icode;
    input [3:0]m_rA;
    input [3:0]m_rB;
    input signed [63:0]m_valE;
    input signed [63:0]m_valM;
    input signed [63:0]m_valC;
    input m_hlt;
    input m_in_inst;
    input m_in_mem;
    input m_cond;

    output reg [3:0] w_icode;
    output reg [3:0]w_rA;
    output reg [3:0]w_rB;
    output reg signed [63:0] w_valE;
    output reg signed [63:0] w_valM;
    output reg signed [63:0] w_valC;
    output reg w_hlt;
    output reg w_in_inst;
    output reg w_in_mem;
    output reg w_cond;

    always @(posedge clk)
    begin
        w_icode <= m_icode;
        w_rA <= m_rA;
        w_rB <= m_rB;
        w_valM <= m_valM;
        w_valE <= m_valE;
        w_valC <= m_valC;
        w_hlt <= m_hlt;
        w_in_inst <= m_in_inst;
        w_in_mem <= m_in_mem;
        w_cond <= m_cond;
    end
endmodule
```

Final Pipeline test bench

The final pipeline test bench implemented is shown below:

```
`timescale 1ns/10ps
`include "fetch_r.v"
`include "dwb_r.v"
`include "execute_r.v"
`include "memory_r.v"
`include "pc_update.v"

module pipe_tb;

reg clk;
```

```
reg [63:0]PC; //memory address is 64 bits
reg AOK,HALT,ADR,INS;
reg [2:0]status;
reg [63:0]nextPC;
wire [63:0]f_nextPC;

wire [3:0]f_icode; //first 4 bits of the first byte
wire [3:0]d_icode;
wire [3:0]e_icode;
wire [3:0]m_icode;
wire [3:0]w_icode;

wire [3:0]f_ifun; //last 4 bits of the first byte
wire [3:0]d_ifun;
wire [3:0]e_ifun;

wire [3:0]f_rA; //first 4 bits of the second byte
wire [3:0]d_rA;
wire [3:0]e_rA;
wire [3:0]m_rA;
wire [3:0]w_rA;

wire [3:0]f_rB; //last 4 bits of the second byte
wire [3:0]d_rB;
wire [3:0]e_rB;
wire [3:0]m_rB;
wire [3:0]w_rB;

wire f_hlt;
wire d_hlt;
wire e_hlt;
wire m_hlt;
wire w_hlt;

wire f_in_mem;
wire d_in_mem;
wire e_in_mem;
wire m_in_mem;
wire w_in_mem;

wire f_in_inst;
wire d_in_inst;
wire e_in_inst;
wire m_in_inst;
wire w_in_inst;


wire signed [63:0]d_valA;
wire signed [63:0]e_valA;
wire signed [63:0]m_valA;

wire signed [63:0]d_valB;
wire signed [63:0]e_valB;

wire signed [63:0]f_valC; //stores the constant word
wire signed [63:0]d_valC;
wire signed [63:0]e_valC;
wire signed [63:0]m_valC;
wire signed [63:0]w_valC;

wire e_cond;
wire m_cond;
wire w_cond;

wire signed [63:0]e_valE;
wire signed [63:0]m_valE;
wire signed [63:0]w_valE;

wire signed [63:0]m_valM;
wire signed [63:0]w_valM;

wire signed [63:0]f_valP; //stores the address of next instruction (PC should be this value)
wire signed [63:0]d_valP;
```

```
wire signed [63:0]e_valP;
wire signed [63:0]m_valP;

wire signed [63:0]data;
wire ZF;
wire SF;
wire OF;

wire signed[63:0] r1;
wire signed[63:0] r2;
wire signed[63:0] r3;
wire signed[63:0] r4;
wire signed[63:0] r5;
wire signed[63:0] r6;
wire signed[63:0] r7;
wire signed[63:0] r8;
wire signed[63:0] r9;
wire signed[63:0] r10;
wire signed[63:0] r11;
wire signed[63:0] r12;
wire signed[63:0] r13;
wire signed[63:0] r14;
wire signed[63:0] r0;
wire [63:0] newPC;

fetch_reg AA (clk, f_valP, f_nextPC); //nextPC will take f_valP because f_valP will store the value of valP after
fetch A (clk, PC, f_icode, f_ifun, f_rA, f_rB, f_valC, f_valP, f_in_mem, f_in_inst, f_hlt);
decode_reg BB (clk, f_icode, f_ifun, f_rA,f_rB,f_valC,f_valP,f_hlt,f_in_inst,f_in_mem,d_icode,d_ifun,d_rA,d_rB,d_valC,d_valP,d
//decode UUTB (clk, d_icode, d_rA, d_rB, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, d_valA, d_valB);
writeback_reg CC(clk, m_icode,m_rA,m_rB,m_valE,m_valM,m_valC,m_cond,m_hlt,m_in_inst,m_in_mem,w_icode,w_rA,w_rB,w_valE,w_valM,w
writeback C (clk, d_icode, d_rA, d_rB, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, d_valA, d_valB, w_valE
execute_reg DD(clk, e_icode, e_ifun, e_rA, e_rB, e_valA, e_valB, e_valC, e_valP, e_hlt, e_in_mem, e_in_inst, d_icode, d_ifun,
execute D (clk, e_icode, e_ifun, e_valA, e_valB, e_valC, e_cond, e_valE, ZF, SF, OF);
memory_reg EE(clk, m_icode, m_cond, m_rA, m_rB, m_valA, m_valE, m_valP, m_valC, m_hlt, m_in_mem, m_in_inst, e_icode, e_cond, e
memory E (clk, m_icode, m_valA, m_valE, m_valP, m_valM, data);
//writeback UUTE (clk, icode, cond, rA, rB, valE, valM, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14);
pc_update F (clk, newPC, w_icode, f_nextPC, w_valM, w_valC, w_cond);

initial
begin
    $dumpfile("pipe_tb.vcd");
    $dumpvars(0,pipe_tb);

    AOK = 1;
    HALT = 0;
    ADR = 0;
    INS = 0;
    status = 3'b001;

    clk = 0;
    PC = 64'd0;
end

initial
begin
    #10 clk = ~clk; //positive edge
    #10 clk = ~clk; //negative edge so nothing happens
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
```

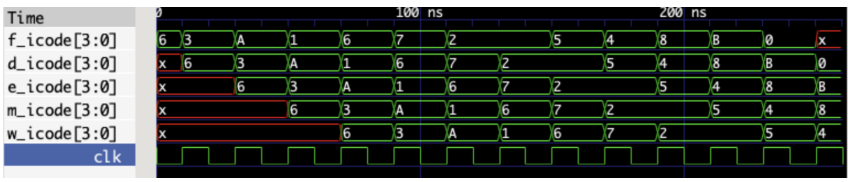
```
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk; //0
    #10 clk = ~clk;
    #10 clk = ~clk; //0
end

always@(*)
begin
    PC = newPC;

    //setting the status codes
    if(w_hlt)
    begin
        AOK = 0;
        HALT= 1;
        INS = 0;
        ADR = 0;
        status = 3'b010;
    end
    else if(w_in_mem)
    begin
        AOK = 0;
        HALT= 0;
        INS = 0;
        ADR = 1;
        status = 3'b011;
    end
    else if(w_in_inst)
    begin
        AOK = 0;
        HALT= 0;
        INS = 1;
        ADR = 0;
        status = 3'b100;
    end
    else
    begin
        AOK = 1;
        HALT = 0;
        INS = 0;
        ADR = 0;
        status = 3'b001;
    end
end

//always @(*)
initial
    $monitor("clk=%d f=%d d=%d e=%d m=%d w=%d \n",clk, f_icode, d_icode, e_icode, m_icode, w_icode);
endmodule
```

The output for the following is:

```
clk=0 f= 6 d= x e= x m= x w= x

clk=1 f= 3 d= 6 e= x m= x w= x

clk=0 f= 3 d= 6 e= x m= x w= x

clk=1 f=10 d= 3 e= 6 m= x w= x

clk=0 f=10 d= 3 e= 6 m= x w= x

clk=1 f= 1 d=10 e= 3 m= 6 w= x
```

```
clk=0 f= 1 d=10 e= 3 m= 6 w= x

clk=1 f= 6 d= 1 e=10 m= 3 w= 6

clk=0 f= 6 d= 1 e=10 m= 3 w= 6

clk=1 f= 7 d= 6 e= 1 m=10 w= 3

clk=0 f= 7 d= 6 e= 1 m=10 w= 3

clk=1 f= 2 d= 7 e= 6 m= 1 w=10

clk=0 f= 2 d= 7 e= 6 m= 1 w=10

clk=1 f= 2 d= 2 e= 7 m= 6 w= 1

clk=0 f= 2 d= 2 e= 7 m= 6 w= 1

clk=1 f= 5 d= 2 e= 2 m= 7 w= 6

clk=0 f= 5 d= 2 e= 2 m= 7 w= 6

clk=1 f= 4 d= 5 e= 2 m= 2 w= 7

clk=0 f= 4 d= 5 e= 2 m= 2 w= 7

clk=1 f= 8 d= 4 e= 5 m= 2 w= 2

clk=0 f= 8 d= 4 e= 5 m= 2 w= 2

clk=1 f=11 d= 8 e= 4 m= 5 w= 2

clk=0 f=11 d= 8 e= 4 m= 5 w= 2

clk=1 f= 0 d=11 e= 8 m= 4 w= 5

clk=0 f= 0 d=11 e= 8 m= 4 w= 5

clk=1 f= x d= 0 e=11 m= 8 w= 4

clk=0 f= x d= 0 e=11 m= 8 w= 4
```

Also, here is the observed GTKwave output for the pipelined processor:



As we notice, the instruction opcodes move through each stage simultaneously and are processed in parallel to each other. Hence, our output is as expected.

Challenges Faced

- The implementation of the Sequential Processor was slightly challenging when it came to conditional instructions.
- However, we faced more challenges when implementing pipelining due to the various signals we have to deal with. In addition, the stall and bubble implementation logic were difficult to figure out.