Introduction to Processor Architecture – EC2.204

Course Project - **Y86-64 Processor**

Team – **37**

Rupak Antani | 2022102045

Soham Vaishnav | 2022112002

Table of Contents:

Overview

Section 1: Sequential Processor Design

- **Introduction**
- **Stages of the Processor**
  1. Fetch
  2. Decode
  3. Execute
  4. Memory
  5. Write-back and PC-update
- **Testing and Results**

Section 2: Pipelined Processor Design

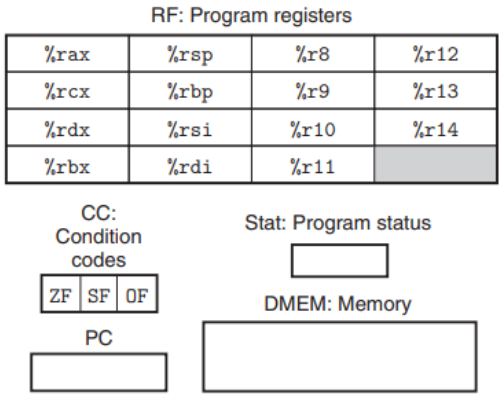
- **Introduction**
- **Pipeline Registers**
- **Hazards and Solutions**
- **Stages of the Processor**
  1. Fetch
  2. Decode
  3. Execute
  4. Memory
  5. Write-back
- **Pipeline Control Module**
- **Testing and Results**

Section 3: Conclusion

- **Challenges Faced**
- **Conclusion**

Overview

Y86 – 64 Instruction Set Architecture (ISA) is an ISA which is slightly modified form of the X86 – 64 ISA. The processor made using the Y86 ISA is a 64-bit architecture which consists of 16 64-bit registers, a condition code register consisting of 3 flags – ZF, OF and SF, a program status register of 4 bits, the program counter (PC) and the memory.



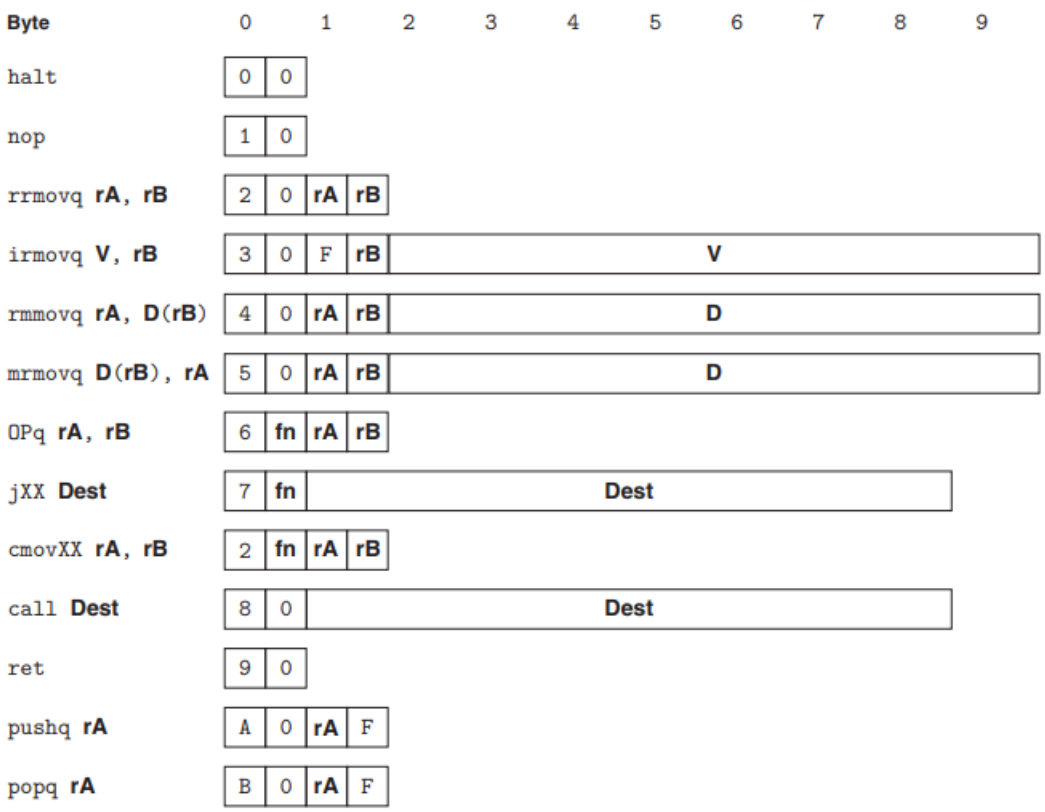
The register file is numbered in the below manner:

Number	Register name	Number	Register name
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	No register

Note: The program register file consists of 16 registers but we using only 15 registers. The 16th register has a don't care (64'bx) value stored in it and is denoted by F. It is useful for carrying out some of the instructions. Also, the register %rsp stores the address of the top of the stack which is present in the memory.

This processor can perform mainly 12 operations and the instructions for each of these operations is of the form as shown below:

P.T.O

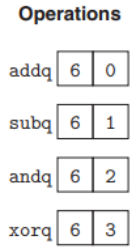


The first byte of each of the instruction specifies the icode and ifun (fn) that are 4-bits each corresponding to the instruction we want to perform. The next byte consists of the register numbers rA and rB for some of the instructions. Some of the instructions also consist of an 8-byte data.

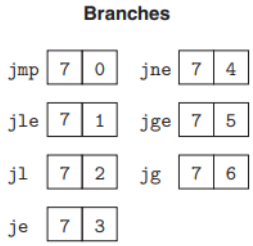
The Y86 ISA can perform the following instructions:

- **Halt –** It has icode and ifun both as 0. When the processor encounters this instruction, it immediately stops the working of the processor.
- **NOP –** It stands for no operation. It has icode 1 and ifun 0. When the processor encounters this instruction, it does not perform any operation.
- **rrmovq –** It is used for register-to-register move. It has icode 2 and ifun 0. It is responsible for copying data from the register rA to the register rB.
- **irmovq –** It is used to for immediate-to-register move. It has icode 3 and ifun 0. It moves the 8-byte immediate data value V into the register rB.
- **rmmovq –** It is used for register-to-memory move of data. It has icode 4 and ifun 0. D is an 8-byte value which gives the displacement value. It moves the data in register rA to the memory address pointed by (value of data in register rB + displacement D).
- **mrmovq –** It is used for memory-to-register move of data. It has icode 5 and ifun 0. D is an 8-byte value which gives the displacement value. It moves the data in the memory address pointed by (value of data in register rB + displacement D) to register rA.

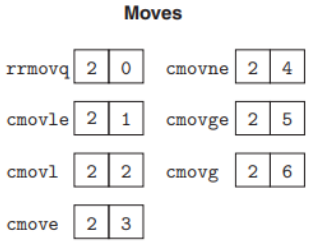
- **Opq –** It is used to perform arithmetic and logical operations on the data in registers in rA and rB. Its icode is 6 and ifun can be 0,1,2 or 3 based on which function we want to perform.



- **jXX** – It is used to jump to a particular instruction address given by the 8-bytes value Dest. It has icode 7 and ifun can be between 0-6. If ifun is 0, then it is an unconditional jump whereas for other ifun, jump takes place only if the corresponding condition is satisfied.



- **cmovXX** – It is used to conditionally move data from register rA to rB. It has icode 2 and ifun can be between 1-6. Based on whether the condition corresponding to the ifun is satisfied or not, the conditional move takes place.



- **call –** It is used to call a function in the program. Its icode is 8 and ifun is 0. Dest is an 8-byte instruction address value of the function we want to call.
- **ret –** It is used to return from a function which was previously called. Its icode is 9 and ifun is 0. It causes the PC to be changed to the instruction address of the instruction just after the call statement.
- **pushq –** It is used to push an 8-byte value stored in register rA in the stack present in the memory. Its icode is A or 10 and ifun is 0.
- **popq –** It is used to pop the data that is at the top of the stack present in memory and store that value in the register rA. Its icode is B or 11 and ifun is 0.

Also, the Status register consists of 4 bits:

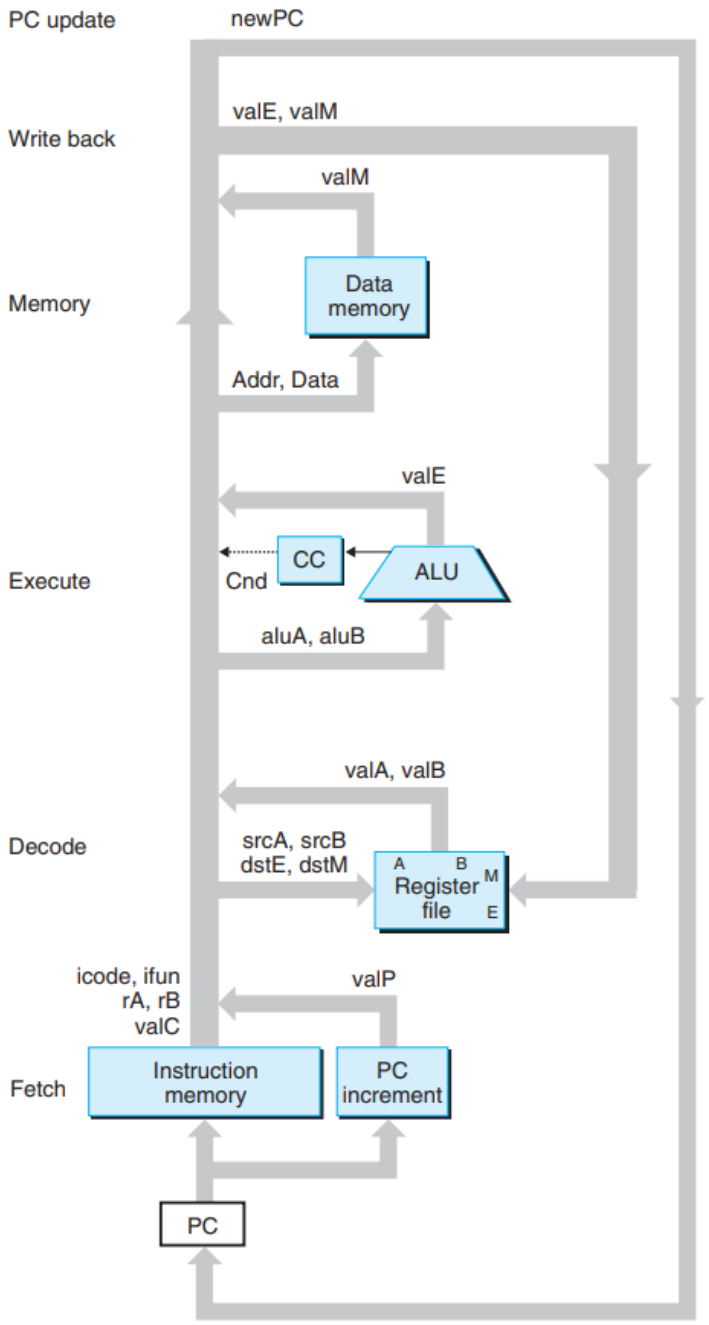
1. AOK – The value of this bit is 1 is there is normal operation.
2. HLT – The value of this bit is 1 is the halt instruction is encountered.
3. ADR – The value of this bit is 1 if any invalid address is encountered.
4. INS – The value of this bit is 1 if any invalid instruction is encountered.

The Y86-64 ISA processor consists of 5 stages – Fetch, Decode, Execute, Memory, Write-back which are described in detail in this report. Also, there is a PC update stage but it can be included in either the fetch stage or the wrapper. This processor can be implemented in 2 ways – Sequential and Pipelined.

Section 1 – Sequential Processor Design

Introduction

In this type of implementation of the Y86-64, all the 5 stages perform their respective roles based on the instruction one after the other. Therefore, at a time only a single stage is working on the instruction. Instructions are fetched one-by-one by the processor and all the 5 stages are executed during a single clock cycle. The control flow of the sequential processor is shown below:



Stages of the Processor:

1. **Fetch:**

   The first and foremost block of the SEQ processor design is the Fetch stage. As the name suggests, the function of this stage is to extract the instructions from the instruction memory based on the current location that PC register points to.

   It is evident therefore that this module will have PC register as an input alongside the clock which emphasises on the fact that in the SEQ processor a new instruction is always fetched at the rising clock edge. Thereafter, during that current clock cycle, before the falling edge occurs, all the subsequent stages are supposed to complete their work. This is done in order to ensure that by the falling edge of the clock cycle, the data, if any, is ready to be written back into the register file or the memory.

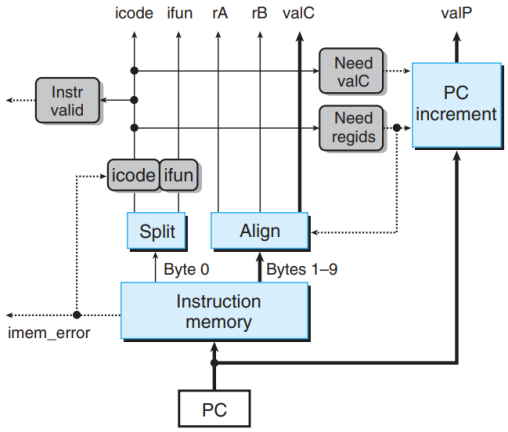


Figure showing architecture of the SEQ Fetch stage

The main parts of the SEQ fetch stage include an Instruction Memory, error flags, instruction blocks, PC register.

a. **PC counter** – This is a register which contains the location of the instruction that has to be fetched from the Instruction memory at the rising edge of the current cycle. PC counter is updated at the falling edge of the previous clock cycle so that it is available for use in the current cycle.

b. **Instruction Memory** – This is a **byte-addressable** array of 8 bit registers which contain the instruction that the program needs to execute. At a time, the max number of bytes that can be read from the Instruction memory is 10 and least that needs to be read is 1.

c. **Instruction Blocks** – An instruction is composed of three primary components which are **icode:ifun** (1 byte), **rA:rB** (1 byte) and **valC** (8 bytes). **icode:ifun** is the necessary and sufficient 1 byte that the fetch stage has to read from the memory in order to know which instruction is there in the first place. Based on this 1 byte we get to know the length of the instruction that needs to be read and thereby set the new value that the PC register will most likely take in the next cycle. Next is the **rA:rB** values that help in decode

stage to know the source and destination registers. Finally, based on **icode:ifun** values we know whether we need **valC** which is the 8 byte word denoting the destination to jump to in case of **jmp** or **call**, displacement in case of **rmmovq** or **mrmovq**, and immediate value in case of **irmovq**.

d. **Error flags** – The error flags include **imem_error** and **instr_valid** (in our case, we call it **func_error**). **imem_error**, as the name suggests, raises an error when the address used to extract an instruction exceeds the size of the instruction memory. **func_error**, on the other hand, raises an error when the instruction fetched, for some peculiar reason, does not belong to any of the 12 instructions compatible with this architecture.

Code:

```
module fetch(
    input [63:0] PC,
    input clk,
    output reg [3:0] icode,
    output reg [3:0] ifun,
    output reg [3:0] rA,
    output reg [3:0] rB,
    output reg [63:0] valC,
    output reg [63:0] valP,
    output reg imem_error,
    output reg func_error
);
    reg [7:0] ROM [1023:0];
    reg [7:0] icode_ifun;
    reg [7:0] rA_rB;

    initial begin
    //Instruction memory
        //write the instructions that need to be performed
        $readmemb("4.txt", ROM);
    end
```

Figure showing the inputs and outputs to the fetch stage along with the declaration and initialisation of Instr_Memory

P.T.O

```
always @ (posedge clk) begin
    imem_error = 0;
    func_error = 0;
    // halt = 0;
    // nop = 0;
    if (PC >= 1024 || PC < 0 && PC !== 64'bx) begin
        imem_error = 1;
        icode = 4'bx;
        ifun = 4'bx;
        rA = 4'bx;
        rB = 4'bx;
        // halt = 1;
    end
    if (imem_error == 0) begin
        icode = ROM[PC][7:4];
        ifun = ROM[PC][3:0];
        case (icode)
            4'b0000 : begin //halt
                // halt = 1;
                valP = PC + 1;
            end
            4'b0001 : begin //nop
                // nop = 1;
                valP = PC + 1;
            end
            4'b0010 : begin //cmovxx and rrmovq
                rA = ROM[PC+1][7:4];
                rB = ROM[PC+1][3:0];
                valP = PC+2;
            end
```

```
            (4'b0011), (4'b0100), (4'b0101) : begin //irmovq or rmmovq or mrmovq
                rA = ROM[PC+1][7:4];
                rB = ROM[PC+1][3:0];
                valC[7:0] = ROM[PC+2];
                valC[15:8] = ROM[PC+3];
                valC[23:16] = ROM[PC+4];
                valC[31:24] = ROM[PC+5];
                valC[39:32] = ROM[PC+6];
                valC[47:40] = ROM[PC+7];
                valC[55:48] = ROM[PC+8];
                valC[63:56] = ROM[PC+9];
                valP = PC+10;
            end
            4'b0110 : begin //OPq
                rA = ROM[PC+1][7:4];
                rB = ROM[PC+1][3:0];
                valP = PC+2;
            end
            (4'b0111), (4'b1000) : begin //jXX or call
                valC[7:0] = ROM[PC+1];
                valC[15:8] = ROM[PC+2];
                valC[23:16] = ROM[PC+3];
                valC[31:24] = ROM[PC+4];
                valC[39:32] = ROM[PC+5];
                valC[47:40] = ROM[PC+6];
                valC[55:48] = ROM[PC+7];
                valC[63:56] = ROM[PC+8];
                valP = PC+9;
            end
```

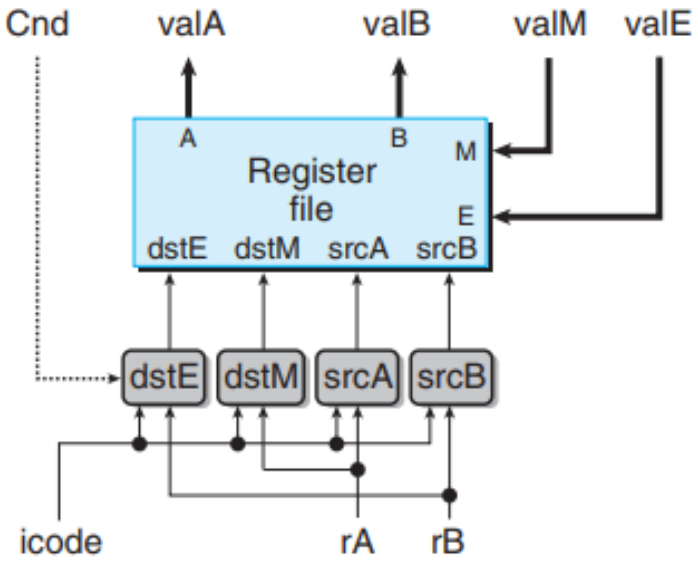
P.T.O

```
            4'b1001 : begin //ret
                valP = PC+1;
            end
            (4'b1010), (4'b1011) : begin //pushq or popq
                rA = ROM[PC+1][7:4];
                rB = ROM[PC+1][3:0];
                valP = PC+2;
            end
            default: begin
                if (icode =/= 4'bx && ifun =/= 4'bx)
                func_error = 1;
                // nop = 1;
            end
        endcase
    end
end
```

Figures showing the main fetch code that runs at posedge clk

2. **Decode:**

The decode stage takes the register numbers, that is, rA and rB as input and is responsible for giving the values stored in these registers, that is, valA and valB as outputs. The decode stage consists of the register file and has a structure as shown below:



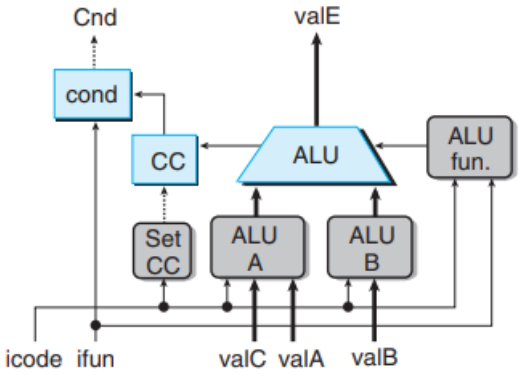
The two read ports have address inputs srcA and srcB, while the two write ports have address inputs dstE and dstM. The below image shows the Verilog code for this stage:

```
module decode_reg_block(
    input clk, cnd,
    input [3:0] icode, ifun,
    input [3:0] rA, rB,
    input write_enable,
    input [63:0] valE, valM,
    input [63:0] valP, valC,
    output reg [63:0] valA, valB,
    output reg[63:0] PC_updated,
    output reg reg_error
);

reg [63:0] storage [15:0];

initial
begin
    storage[0] = 64'b1010; //%rax
    storage[1] = 64'b1110; //%rcx
    storage[2] = 64'b1011; //%rdx
    storage[3] = 64'b1111; //%rbx
    storage[4] = 64'd2047; //%rsp -> points
    storage[5] = 64'b1010; //%rbp
    storage[6] = 64'b0001; //%rsi
    storage[7] = 64'b0010; //%rdi
    storage[8] = 64'b1000; //%r8
    storage[9] = 64'b1110; //%r9
    storage[10] = 64'b1010; //%r10
    storage[11] = 64'b1001; //%r11
    storage[12] = 64'b1110; //%r12
    storage[13] = 64'b0111; //%r13
    storage[14] = 64'b0101; //%r14
    storage[15] = 64'bx; //no register -> F
end

always@(*)
begin
    if (write_enable != 1)
    begin
        if((icode == 0) || (icode == 1) || (icode == 7))
        begin
            valA = storage[15];
            valB = storage[15];
            reg_error = 0;
        end
        else if((icode == 3))
        begin
            if((rB >= 0)&&(rB < 15))
            begin
                valA = storage[15];
                valB = storage[rB];
                reg_error = 0;
            end
            else reg_error = 1;
        end
        else if((icode == 10))
        begin
            if((rA >= 0)&&(rA < 4'b1111))
            begin
                valA = storage[rA];
                valB = storage[4];
                reg_error = 0;
            end
            else reg_error = 1;
        end
        else if ((icode == 8) || (icode == 9) || (icode == 11))
        begin
            valA = storage[4];
            valB = storage[4];
        end
        else
        begin
            if ((rA >= 0)&&(rA < 15)&&(rB >=0)&&(rB < 15))
            begin
                valA = storage[rA];
                valB = storage[rB];
                reg_error = 0;
            end
            else reg_error = 1;
        end
    end
end
```

Here, first we are defining the input and output ports. Next, we declare the register file named ‘storage’ and assign some initial values to the data in these registers which can later be modified by the instructions. Next, within the always block we define what the decode stage has to do for each value of icode using the if-else conditions. The register 4 is %rsp register which stores the address to the top of the stack. So, for the instructions which involve the %rsp register, we assign the value in register 4 to valA, valB or both as per the requirement. Lastly, there is also a reg_error bit. This bit is 1 if the invalid register address is given, else 0.

3. **Execute:**

   The execute stage is responsible for carrying out arithmetic and logic operations on the data valA and valB and giving the output valE and also for computing the condition bit ‘cnd’. It consists of the ALU and the condition codes register as shown below:

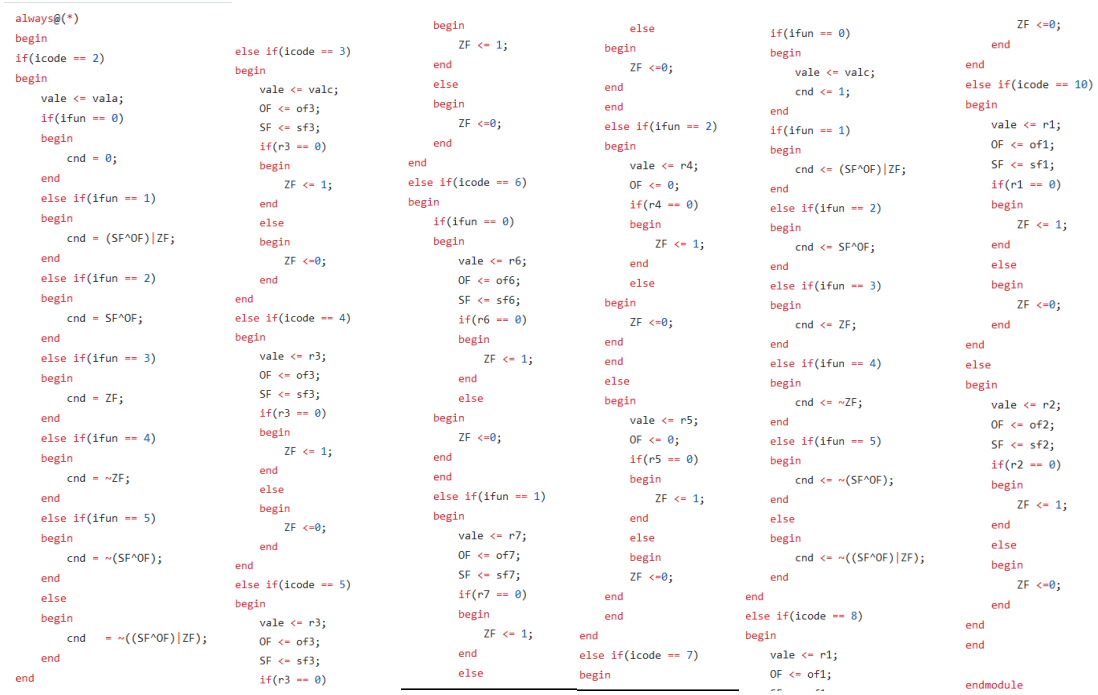


The condition codes register consists of 3 flags:

1. ZF – It is 1 if the computed value is 0, else 0.
2. SF – It is 1 if the computed value is negative, else 0.
3. OF – It is 1 if the computed value has overflowed beyond 64 bits, else 0.

The Verilog code for the execute stage is shown below:

```
`include "add_sub_1_bit.v"
`include "add_sub_64_bit.v"
`include "and_1_bit.v"
`include "and_64_bit.v"
`include "xor_1_bit.v"
`include "xor_64_bit.v"

module execute_block(vala,valb,valc,icode,ifun,vale,cnd,SF,ZF,OF);

input [63:0] vala,valb,valc;
input [3:0] icode,ifun;
output reg [63:0] vale;
output reg cnd,SF,ZF,OF;

/*reg SF,ZF,OF;*/
reg z = 0;
reg nz = 1;

reg [63:0] p = 4'b1000;
wire [62:0] r1,r2,r3,r6,r7,r8,r9;
wire [63:0] r4,r5;
wire of1,of2,of3,of6,of7,of8,of9,sf1,sf2,sf3,sf6,sf7,sf8,sf9;

add_sub_64_bit m1(.a(valb),.b(p),.m(nz),.sum(r1),.over(of1),.sign(sf1)); /* valb - 8 */
add_sub_64_bit m2(.a(valb),.b(p),.m(z),.sum(r2),.over(of2),.sign(sf2)); /* valb + 8 */
add_sub_64_bit m3(.a(valc),.b(valb),.m(z),.sum(r3),.over(of3),.sign(sf3)); /* valb + valc */
and_64_bit a3(.a(vala),.b(valb),.c(r4)); /* vala and valb */
xor_64_bit a4(.a(vala),.b(valb),.c(r5)); /* vala xor valb */
add_sub_64_bit m6(.a(valb),.b(vala),.m(z),.sum(r6),.over(of6),.sign(sf6)); /* valb + vala */
add_sub_64_bit m7(.a(valb),.b(vala),.m(nz),.sum(r7),.over(of7),.sign(sf7)); /* valb - vala */
add_sub_64_bit m8(.a(vala),.b(p),.m(z),.sum(r8),.over(of8),.sign(sf8)); /* vala + 8 */
add_sub_64_bit m9(.a(vala),.b(p),.m(nz),.sum(r9),.over(of9),.sign(sf9)); /* vala - 8 */
```



```
always@(*)
begin
if(icode == 2)
begin
    vale <= vala;
    if(ifun == 0)
    begin
        cnd = 0;
    end
    else if(ifun == 1)
    begin
        cnd = (SF^OF)|ZF;
    end
    else if(ifun == 2)
    begin
        cnd = SF^OF;
    end
    else if(ifun == 3)
    begin
        cnd = ZF;
    end
    else if(ifun == 4)
    begin
        cnd = ~ZF;
    end
    else if(ifun == 5)
    begin
        cnd = ~(SF^OF);
    end
    else
    begin
        cnd   = ~((SF^OF)|ZF);
    end
end
else if(icode == 3)
begin
    vale <= valc;
    OF <= of3;
    SF <= sf3;
    if(r3 == 0)
    begin
        ZF <= 1;
    end
    else
    begin
        ZF <=0;
    end
end
else if(icode == 4)
begin
    vale <= r3;
    OF <= of3;
    SF <= sf3;
    if(r3 == 0)
    begin
        ZF <= 1;
    end
    else
    begin
        ZF <=0;
    end
end
else if(icode == 5)
begin
    vale <= r3;
    OF <= of3;
    SF <= sf3;
    if(r3 == 0)
    begin
        ZF <= 1;
    end
    else
    begin
        ZF <=0;
    end
end
else if(icode == 6)
begin
    if(ifun == 0)
    begin
        vale <= r6;
        OF <= of6;
        SF <= sf6;
        if(r6 == 0)
        begin
            ZF <= 1;
        end
        else
    begin
        ZF <=0;
    end
    end
    else if(ifun == 1)
    begin
        vale <= r7;
        OF <= of7;
        SF <= sf7;
        if(r7 == 0)
        begin
            ZF <= 1;
        end
        else
    begin
        ZF <=0;
    end
    end
    else if(ifun == 2)
    begin
        vale <= r4;
        OF <= 0;
        if(r4 == 0)
        begin
            ZF <= 1;
        end
        else
    begin
        ZF <=0;
    end
    end
    else
    begin
        vale <= r5;
        OF <= 0;
        if(r5 == 0)
        begin
            ZF <= 1;
        end
        else
        begin
        ZF <=0;
    end
    end
end
else if(icode == 7)
begin
    if(ifun == 0)
    begin
        vale <= valc;
        cnd <= 1;
    end
    if(ifun == 1)
    begin
        cnd <= (SF^OF)|ZF;
    end
    else if(ifun == 2)
    begin
        cnd <= SF^OF;
    end
    else if(ifun == 3)
    begin
        cnd <= ZF;
    end
    else if(ifun == 4)
    begin
        cnd <= ~ZF;
    end
    else if(ifun == 5)
    begin
        cnd <= ~(SF^OF);
    end
    else
    begin
        cnd <= ~((SF^OF)|ZF);
    end
end
else if(icode == 8)
begin
    vale <= r1;
    OF <= of1;
        ZF <=0;
    end
end
else if(icode == 10)
begin
    vale <= r1;
    OF <= of1;
    SF <= sf1;
    if(r1 == 0)
    begin
        ZF <= 1;
    end
    else
    begin
        ZF <=0;
    end
end
else
begin
    vale <= r2;
    OF <= of2;
    SF <= sf2;
    if(r2 == 0)
    begin
        ZF <= 1;
    end
    else
    begin
        ZF <=0;
    end
end
end

endmodule
```

In this code, we first include the files that will be required for addition, subtraction, AND and XOR operations and define the inputs and outputs. Next, we declare all the modules that we will be making use for any of the instructions. Note that, we have included these modules outside the always block because modules cannot be included inside the always block as it is not possible in the hardware. Next, inside the always block we assign the values to valE, SF, OF and ZF based on the outputs that we get after doing the required computation based on the icode. For icode 2 and 7, that is, for cmov and jmp instructions, the condition bit ‘cnd’ is computed based on the condition corresponding to the ifun value.

4. Memory:

The main function of this stage is to assist the processor in reading and writing into the memory file (in our case we call it the RAM). It is this memory file that includes a certain set of the array dedicated to stack which is accessed by functions like **pushq**, **popq**, **call** and **ret**.

In our implementation, we have used a memory which is byte addressable of size 2KB, of which 64 bytes are dedicated to stack operations. At once, the processor reads 8 bytes from and writes 8 bytes into the memory. Thus, the inputs to the memory are **icode:ifun**, **valA**, **valB**, **valE**, and **valP**, and the output is the value **valM** which will be read from the memory in some cases.

P.T.O

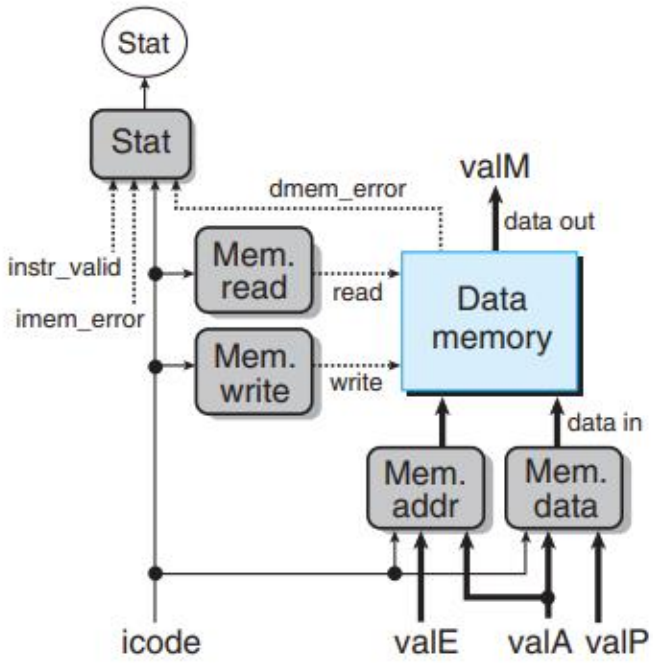


Figure showing the architecture of the memory stage

In the above figure, we can see that the memory stage also outputs the **stat** register which is a 4 bit register containing the status codes of the program. The status codes are used to determine the current working condition of the processor, that is, whether all is ok, or a halt has been encountered, or if there is some addressing error, or maybe the instruction fetched is not valid at all!

But, in our implementation, we have included the updating of status codes in the final combined module for the processor (and not in the memory module specifically).

The main components of the memory stage are Data memory, error flags, and valM.

a. **Data Memory** – A byte addressable block of registers that helps in storing data which can be then be read as and when required. The data memory also comes with the ability to be written into. It is this data memory that dedicates a few registers to the stack operations (in our case – 64 bytes). The operation of writing into the memory occurs only at the negative edge of the clock based on the assumption that the data to be written will be ready by that instance.
b. **Error Flags** – The only error flag that will be raised in this module will be regarding the addressing issue while reading from or writing into the memory. In our implementation, we call this **dmem_error**.
c. **valM** – This is the value that we get when we read from the memory. This value is then passed onto the next stage (write-back) or to the PC-update depending on the **icode:ifun** values.

Code:

```
module memory (
    input clk,
    input [3:0] icode, ifun,
    input [63:0] valA, valB, valC,
    input [63:0] valP, valE,
    // input write_enable,
    output reg [63:0] valM,
    output reg dmem_error
);
    reg [8:0] RAM [2047:0]; //2048 bytes memory
    // reg [63:0] RAM[1023:0];
    // last 64 locations have been reserved for stack operations
    // → from 1984 to 2047 locations are reserved for stack operations
    reg [63:0] address;
    reg [63:0] data_in;
```

Figure showing the inputs and outputs to the memory module and the declaration and initiation of the data memory (RAM)

```
always @ (*) begin //all the "reading from mem." operations
    case (icode)
    4'b0101 : begin //mrmovq
    //here valE is the address and output is valM
        address = valE;
        if (address < 2047 && address >= 0) begin
            dmem_error = 0;
            // if (write_enable == 0) begin
                // valM = RAM[address];
                valM[7:0] = RAM[address];
                valM[15:8] = RAM[address+1];
                valM[23:16] = RAM[address+2];
                valM[31:24] = RAM[address+3];
                valM[39:32] = RAM[address+4];
                valM[47:40] = RAM[address+5];
                valM[55:48] = RAM[address+6];
                valM[63:56] = RAM[address+7];
            // end
        end
        else dmem_error = 1;
    end
```

```
    4'b1001 : begin //ret
    //here valA is the address and output is valM
        address = valA;
        if (address < 2047 && address >= 0) begin
            dmem_error = 0;
            // if (write_enable == 0) begin
                // valM = RAM[address];
                valM[7:0] = RAM[address];
                valM[15:8] = RAM[address+1];
                valM[23:16] = RAM[address+2];
                valM[31:24] = RAM[address+3];
                valM[39:32] = RAM[address+4];
                valM[47:40] = RAM[address+5];
                valM[55:48] = RAM[address+6];
                valM[63:56] = RAM[address+7];
            // end
        end
        else dmem_error = 1;
    end
```

```
4'b1011 : begin //popq
//here valE is the address and output is valM
    address = valA;
    if (address < 2047 && address >= 0) begin
        dmem_error = 0;
        // if (write_enable == 0) begin
            // valM = RAM[address];
            valM[7:0] = RAM[address];
            valM[15:8] = RAM[address+1];
            valM[23:16] = RAM[address+2];
            valM[31:24] = RAM[address+3];
            valM[39:32] = RAM[address+4];
            valM[47:40] = RAM[address+5];
            valM[55:48] = RAM[address+6];
            valM[63:56] = RAM[address+7];
        // end
    end
    else dmem_error = 1;
end
default : begin
    if ((icode == 4'b0000) || (icode == 4'b0001) || (icode == 4'b0010) || (icode == 4'b0011)
    || (icode == 4'b0110) || (icode == 4'b0111) || (icode == 4'b1000) || (icode == 4'b1010)
    || (icode == 4'b0100)) begin
        dmem_error = 0;
    end
end
endcase
```

Figures showing the code which "reads from" the memory

```
always @(negedge clk) begin //all the "writing into mem." operations
    case (icode)
    4'b0100 : begin //rmmovq
    //here valE is the address and valA is the input data
        address = valE;
        data_in = valA;
        if (address < 2047 && address >= 0) begin
            dmem_error = 0;
            // if (write_enable == 1) begin
                // RAM[address] = data_in;
                RAM[address] = data_in[7:0];
                RAM[address+1] = data_in[15:8];
                RAM[address+2] = data_in[23:16];
                RAM[address+3] = data_in[31:24];
                RAM[address+4] = data_in[39:32];
                RAM[address+5] = data_in[47:40];
                RAM[address+6] = data_in[55:48];
                RAM[address+7] = data_in[63:56];
            // end
        end
        else dmem_error = 1;
    end
```

```
    4'b1000 : begin //call
    //here valE is the address and valP is the input data
        address = valE;
        data_in = valP;
        if (address < 2047 && address >= 0) begin
            dmem_error = 0;
            // if (write_enable == 1) begin
                // RAM[address] = data_in;
                RAM[address] = data_in[7:0];
                RAM[address+1] = data_in[15:8];
                RAM[address+2] = data_in[23:16];
                RAM[address+3] = data_in[31:24];
                RAM[address+4] = data_in[39:32];
                RAM[address+5] = data_in[47:40];
                RAM[address+6] = data_in[55:48];
                RAM[address+7] = data_in[63:56];
            // end
        end
        else dmem_error = 1;
    end
```

```
    4'b1010 : begin //pushq
    //here valE is the address and valA is the input data
        address = valE;
        data_in = valA;
        if (address < 2047 && address >=0) begin
            dmem_error = 0;
            // if (write_enable == 1) begin
                // RAM[address] = data_in;
                RAM[address] = data_in[7:0];
                RAM[address+1] = data_in[15:8];
                RAM[address+2] = data_in[23:16];
                RAM[address+3] = data_in[31:24];
                RAM[address+4] = data_in[39:32];
                RAM[address+5] = data_in[47:40];
                RAM[address+6] = data_in[55:48];
                RAM[address+7] = data_in[63:56];
            // end
        end
        else dmem_error = 1;
    end
    default : begin
        if ((icode == 4'b0000) || (icode == 4'b0001) || (icode == 4'b0010) || (icode == 4'b0011)
        || (icode == 4'b0110) || (icode == 4'b0111) || (icode == 4'b1001) || (icode == 4'b1011)
        || (icode == 4'b0101)) begin
            dmem_error = 0;
        end
    end
    endcase
end
```

Figures showing the code which "writes into" the memory

5. **Write-back and PC-update:**

   This stage is the final stage of the SEQ processor wherein all the data available at the end of the memory stage is written back into the register file, if required by the instruction that is being processed. Since the write-back stage has to exclusively work with the register file, it forms an important part of the decode stage itself.

   The decode stage is facilitated with a "write-enable" function for this particular requirement of the write-back stage. It is important to note that the write-back stage will have no error flags of itself due to the fact that being the last stage serving the purpose of simply writing into the register file, if there were any errors occurring, they would have already been raised in the previous stages. Although the write-back stage will display the status codes to show the working condition of the processor at the end processing every instruction.

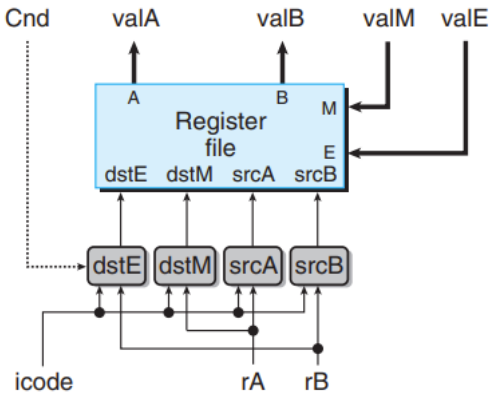


Figure showing the architecture for write-back stage.
Note that it is the same as the decode stage.

In the above figure, **valM** and **valE** are the inputs to the register files at **dstM** and **dstE** respectively which form the basis of the write-back stage.

Now, in our implementation of the SEQ processor, we have included the PC-update stage here itself (along with write-back). Since the inputs to the decode stage (as shown in the code) include **valP**, **valC**, and **valM**, and those three being the only values that PC counter can take, this stage can also act as a PC updating stage. Later, in the combined processor module, we can assign this **PC_updated** value to the **PC** (input to fetch stage) at the negative edge of the clock.

Note that the write-back stage, like writing into memory stage, will perform its function at the negative edge of the clock taking into consideration the same assumption that drove writing into memory stage

Code:

```
always @ (negedge clk) begin //write_back stage operations
    if(write_enable == 1) begin
    case (icode)
    (4'b0010) : begin //cmovxx, rrmovq
        if (ifun == 0) begin
            if (rB < 15 && rB >= 0) begin
                storage[rB] = valE;
                reg_error = 0;
            end
            else reg_error = 1;
        end
        else begin
            if (cnd == 1) begin
                if (rB < 15 && rB >= 0) begin
                    storage[rB] = valE;
                    reg_error = 0;
                end
                else reg_error = 1;
            end
        end
        PC_updated = valP;
    end
    (4'b0011), (4'b0110) : begin //irmovq, OPq
        if (rB < 15 && rB >= 0) begin
            storage[rB] = valE;
            reg_error = 0;
        end
        else reg_error = 1;
        PC_updated = valP;
    end
```

```
    4'b0101 : begin //mrmovq
        if (rB < 15 && rB >= 0) begin
            storage[rB] = valM;
            reg_error = 0;
        end
        else reg_error = 1;
        PC_updated = valP;
    end
    (4'b1000), (4'b1010) : begin //call or pushq
        storage[4] = valE;
        reg_error = 0;
        if (icode == 4'b1000) begin
            PC_updated = valC;
        end
        else begin
            PC_updated = valP;
        end
    end
    4'b1001 : begin //ret
        storage[4'b0100] = valE;
        PC_updated = valM;
    end
    4'b1011 : begin //popq
        if (rA < 15 && rA >= 0) begin
            storage[4'b0100] = valE;
            storage[rA] = valM;
            reg_error = 0;
        end
        else reg_error = 1;
        PC_updated = valP;
    end
```

```
    4'b0111 : begin //jXX
        if (ifun == 0) begin
            PC_updated = valC;
        end
        else begin
            if (cnd == 1) begin
                PC_updated = valC;
            end
            else PC_updated = valP;
        end
        reg_error = 0;
    end
    default : begin
        reg_error = 0;
        PC_updated = valP;
    end
    endcase
    end
end

endmodule
```

Figures showing the code in the decode stage for carrying out Write-back and PC-update stages

Testing and Results:

1. **Test-cases for the individual stages**

   The test-cases for the individual stages include testing the individual instructions.

   Results:

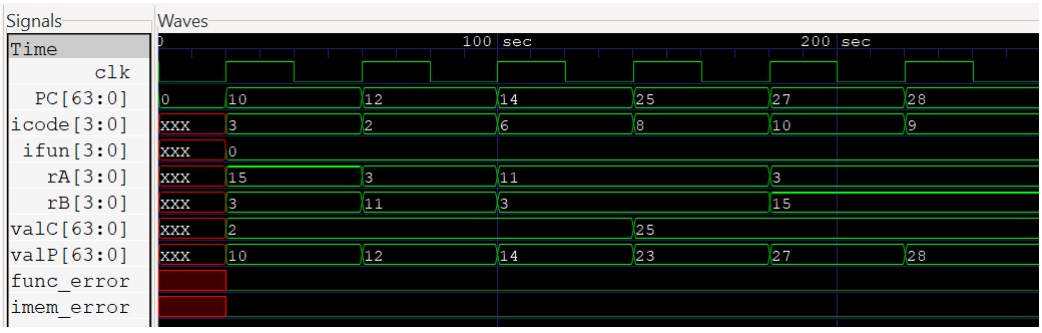
   a. **Fetch:**

```
ROM[0] = 8'b00110000; //irmovq
ROM[1] = 8'b11110011;
ROM[2] = 8'b00000010;
ROM[3] = 8'b00000000;
ROM[4] = 8'b00000000;
ROM[5] = 8'b00000000;
ROM[6] = 8'b00000000;
ROM[7] = 8'b00000000;
ROM[8] = 8'b00000000;
ROM[9] = 8'b00000000;
ROM[10] = 8'b00100000; //rrmovq
ROM[11] = 8'b00111011;
ROM[12] = 8'b01100000; //addq
ROM[13] = 8'b10110011;
ROM[14] = 8'b10000000; //call
ROM[15] = 8'b00011001;
ROM[16] = 8'b00000000;
ROM[17] = 8'b00000000;
ROM[18] = 8'b00000000;
ROM[19] = 8'b00000000;
ROM[20] = 8'b00000000;
ROM[21] = 8'b00000000;
ROM[22] = 8'b00000000;
ROM[23] = 8'b10110000; //popq
ROM[24] = 8'b00111111;
ROM[25] = 8'b10100000; //pushq → call dst
ROM[26] = 8'b00111111;
ROM[27] = 8'b10010000; //ret
```

Above is the testcase for SEQ fetch stage

```
PC = 0, icode = x, ifun =  x, rA =  x, rB = x, valc = x, valp = x
PC = 10, icode = 3, ifun =  0, rA = 15, rB = 3, valc = 2, valp = 10
PC = 12, icode = 2, ifun =  0, rA =  3, rB = 11, valc = 2, valp = 12
PC = 14, icode = 6, ifun =  0, rA = 11, rB = 3, valc = 2, valp = 14
PC = 25, icode = 8, ifun =  0, rA = 11, rB = 3, valc = 25, valp = 23
PC = 27, icode = 10, ifun =  0, rA =  3, rB = 15, valc = 25, valp = 27
PC = 28, icode = 9, ifun =  0, rA =  3, rB = 15, valc = 25, valp = 28
PC = 28, icode = x, ifun =  x, rA =  3, rB = 15, valc = 25, valp = 28
```

Above is the result of the shown testcase | Note that the "ret" instruction at the end is not returning to the location right after "call" due to absence of "valM" which is available only after memory stage is crossed.



Above is the GTK plot for the same result

   b. **Decode:**

   The below image shows the values stored in the register file as well as the inputs and outputs that we are giving:

```
storage[0] = 64'b1010; //%rax
storage[1] = 64'b1110; //%rcx
storage[2] = 64'b1011; //%rdx
storage[3] = 64'b1111; //%rbx
storage[4] = 64'd2047; //%rsp -> points
storage[5] = 64'b1010; //%rbp
storage[6] = 64'b0001; //%rsi
storage[7] = 64'b0010; //%rdi
storage[8] = 64'b1000; //%r8
storage[9] = 64'b1110; //%r9
storage[10] = 64'b1010; //%r10
storage[11] = 64'b1001; //%r11
storage[12] = 64'b1110; //%r12
storage[13] = 64'b0111; //%r13
storage[14] = 64'b0101; //%r14
storage[15] = 64'bx; //no register -> F
```

```
icode = x, ifun = x, rA = x, rB = x, A = x, B = x , reg_error = 1
icode = 0, ifun = 0, rA = 4, rB = 7, A = x, B = x , reg_error = 1
icode = 0, ifun = 0, rA = 4, rB = 7, A = x, B = x , reg_error = 0
icode = 1, ifun = 0, rA = 11, rB = 8, A = x, B = x , reg_error = 0
icode = 2, ifun = 0, rA = 12, rB = 13, A = x, B = x , reg_error = 0
icode = 2, ifun = 0, rA = 12, rB = 13, A = 14, B = 7 , reg_error = 0
icode = 2, ifun = 2, rA = 0, rB = 5, A = 14, B = 7 , reg_error = 0
icode = 2, ifun = 2, rA = 0, rB = 5, A = 10, B = 10 , reg_error = 0
icode = 3, ifun = 0, rA = 12, rB = 9, A = 10, B = 10 , reg_error = 0
icode = 3, ifun = 0, rA = 12, rB = 9, A = x, B = 14 , reg_error = 0
icode = 4, ifun = 0, rA = 1, rB = 14, A = x, B = 14 , reg_error = 0
icode = 4, ifun = 0, rA = 1, rB = 14, A = 14, B = 5 , reg_error = 0
icode = 10, ifun = 0, rA = 7, rB = 6, A = 14, B = 5 , reg_error = 0
icode = 10, ifun = 0, rA = 7, rB = 6, A = 2, B = 2047 , reg_error = 0
```

We can see that when icode = 0 and 1, A = x and B = x. This is because for halt we do not need A and B values. Next, for icode = 2,4, when we give the value of rA and rB, we are getting the corresponding correct values of A and B in the output. For icode = 3, we require only B value and A should be x. So, when we give both rA and rB values, we get A = x and B as the value in register rB. Lastly, for icode = 10, B should be the value stored in the %rsp(4) register irregardless of what the rB value is and A value should be value stored in rA which we can see clearly.

c. **Execute:**

The below image shows the inputs and corresponding outputs for each icode and ifun value:

```
icode = 4, ifun = 1, A = 9, B = 1, C = 11, E = x cnd = x
icode = 4, ifun = 1, A = 9, B = 1, C = 11, E = 12 cnd = x
icode = 5, ifun = 5, A = 8, B = 5, C = 10, E = 12 cnd = x
icode = 5, ifun = 5, A = 8, B = 5, C = 10, E = 15 cnd = x
icode = 6, ifun = 0, A = 6, B = 2, C = 1, E = 15 cnd = x
icode = 6, ifun = 0, A = 6, B = 2, C = 1, E = 8 cnd = x
icode = 6, ifun = 0, A = 9223372036854775807, B = 9223372036854775807, C = 1, E = 8 cnd = x
icode = 6, ifun = 0, A = 9223372036854775807, B = 9223372036854775807, C = 1, E = 9223372036854775806 cnd = x
icode = 7, ifun = 5, A = 5, B = 15, C = 0, E = 9223372036854775806 cnd = x
icode = 7, ifun = 5, A = 5, B = 15, C = 0, E = 9223372036854775806 cnd = 0
icode = 6, ifun = 2, A = 6, B = 2, C = 1, E = 9223372036854775806 cnd = 0
icode = 6, ifun = 2, A = 6, B = 2, C = 1, E = 2 cnd = 0
icode = 8, ifun = 3, A = 6, B = 10, C = 15, E = 2 cnd = 0
icode = 9, ifun = 14, A = 12, B = 2, C = 12, E = 2 cnd = 0
icode = 9, ifun = 14, A = 12, B = 2, C = 12, E = 10 cnd = 0
icode = 10, ifun = 1, A = 12, B = 15, C = 2, E = 10 cnd = 0
icode = 10, ifun = 1, A = 12, B = 15, C = 2, E = 7 cnd = 0
icode = 11, ifun = 5, A = 3, B = 2, C = 10, E = 7 cnd = 0
icode = 11, ifun = 5, A = 3, B = 2, C = 10, E = 10 cnd = 0
```

As rmmovq and mrmovq instructions have icode 4 and 5 respectively, the role of the execute stage is to add valB and valC which is correctly getting added here. For icode 6 and ifun 0, additon takes place (6 +2 =

8), for ifun 1, subtraction takes place , for ifun 2 AND takes place (0110 & 0010 = 0010) and for ifun 3 XOR takes place which are taking place correctly here. Next, for icode 7, cnd is computed. Here, as ifun is 5 which corresponds for greater than or equal, as the condition is not satisfied, cnd is 0. For icode 9 and 11, the execute stage gives output E as valB + 8 which is correctly done here. Also, for icode 10, the execute stage gives output E as valB – 8, which takes takes place correctly here.

d. **Memory:**

```
clk = 0; //0

#20 clk = !clk; //1
icode = 3; ifun = 0;
valA = 64'bx; valB = 15;
valC = 1; valP = 10;
valE = 1;

#20 clk = !clk; //0

#20 clk = !clk; //1
icode = 2; ifun = 0;
valA = 1; valB = 14;
valC = 1; valP = 12;
valE = 1;

#20 clk = !clk; //0
// PC = valP;

#20 clk = !clk; //1
icode = 6; ifun = 0;
valA = 1; valB = 1;
valC = 1; valP = 14;
valE = 2;
```

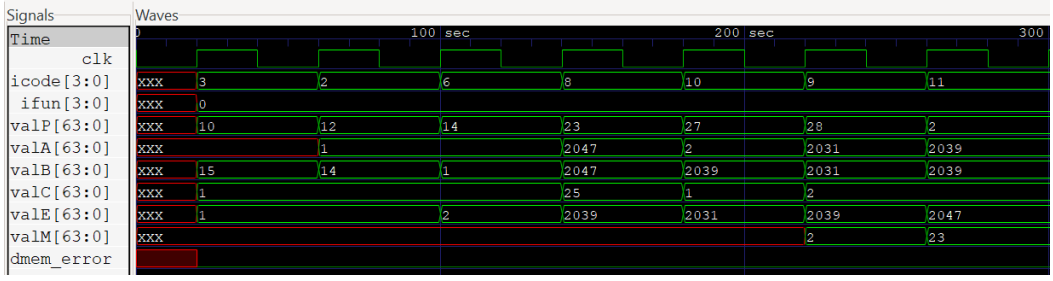
```
#20 clk = !clk; //0
// PC = valP;

#20 clk = !clk; //1
icode = 8; ifun = 0;
valA = 2047; valB = 2047;
valC = 25; valP = 23;
valE = 2039;

#20 clk = !clk; //0
// PC = valP;

#20 clk = !clk; //1
icode = 10; ifun = 0;
valA = 2; valB = 2039;
valC = 1; valP = 27;
valE = 2031;

#20 clk = !clk; //0
// PC = valP;

#20 clk = !clk; //1
icode = 9; ifun = 0;
valA = 2031; valB = 2031;
valC = 2; valP = 28;
valE = 2039;
```

```
#20 clk = !clk; //0
// PC = valP;

#20 clk = !clk; //1
icode = 11; ifun = 0;
valA = 2039; valB = 2039;
valC = 2; valP = valM;
valE = 2047;

#20 clk = !clk; //0
// PC = valP;

#20 clk = !clk; //1

#20 clk = !clk; //0
// PC = valP;

#20 clk = !clk; //1

#20 clk = !clk; //0
// PC = valP;

#20 clk = !clk; //1

#20 clk = !clk; //0
// PC = valP;
```

Testcases in accordance to the previous instructions set

```
icode = x, ifun = x, valA = x, valB = x, valC = x, valE = x, valP = x, valM = x, dmem_error = x
icode = 3, ifun = 0, valA = x, valB = 15, valC = 1, valE = 1, valP = 10, valM = x, dmem_error = 0
icode = 2, ifun = 0, valA = 1, valB = 14, valC = 1, valE = 1, valP = 12, valM = x, dmem_error = 0
icode = 6, ifun = 0, valA = 1, valB = 1, valC = 1, valE = 2, valP = 14, valM = x, dmem_error = 0
icode = 8, ifun = 0, valA = 2047, valB = 2047, valC = 25, valE = 2039, valP = 23, valM = x, dmem_error = 0
icode = 10, ifun = 0, valA = 2, valB = 2039, valC = 1, valE = 2031, valP = 27, valM = x, dmem_error = 0
icode = 9, ifun = 0, valA = 2031, valB = 2031, valC = 2, valE = 2039, valP = 28, valM = 2, dmem_error = 0
icode = 11, ifun = 0, valA = 2039, valB = 2039, valC = 2, valE = 2047, valP = 2, valM = 23, dmem_error = 0
```

Above is the result obtained on running the given testcase



Above is the GTK plot for the shown result

e. **Write-back and PC-update**

```
#20 clk = 0;
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 0; ifun = 0;
rA = 4; rB = 7;
valE = 10; valM = 12;
valP = 69;
write_enable = 0;
PC = PC_updated;

#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 1; ifun = 0;
rA = 3; rB = 5;
valE = 114; valM = 102;
valP = 68;
write_enable = 0;
PC = PC_updated;

#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 2; ifun = 0;
rA = 9; rB = 12;
valE = 109; valM = 120;
valP = 67;
write_enable = 0;
PC = PC_updated;
```

```
#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 2; ifun = 3;
rA = 12; rB = 14;
valE = 137; valM = 192;
valP = 66;
write_enable = 0;
PC = PC_updated;

#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 3; ifun = 0;
rA = 3; rB = 14;
valE = 100; valM = 912;
valP = 65;
write_enable = 0;
PC = PC_updated;

#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 4; ifun = 0;
rA = 13; rB = 10;
valE = 31; valM = 702;
valP = 64;
write_enable = 0;
PC = PC_updated;
```

```
#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 5; ifun = 0;
rA = 1; rB = 6;
valE = 19; valM = 52;
valP = 63;
write_enable = 0;
PC = PC_updated;

#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 6; ifun = 0;
rA = 2; rB = 11;
valE = 15; valM = 732;
valP = 62;
write_enable = 0;
PC = PC_updated;

#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 7; ifun = 0;
rA = 4; rB = 7;
valE = 1; valM = 72;
valP = 61;
write_enable = 0;
PC = PC_updated;
```

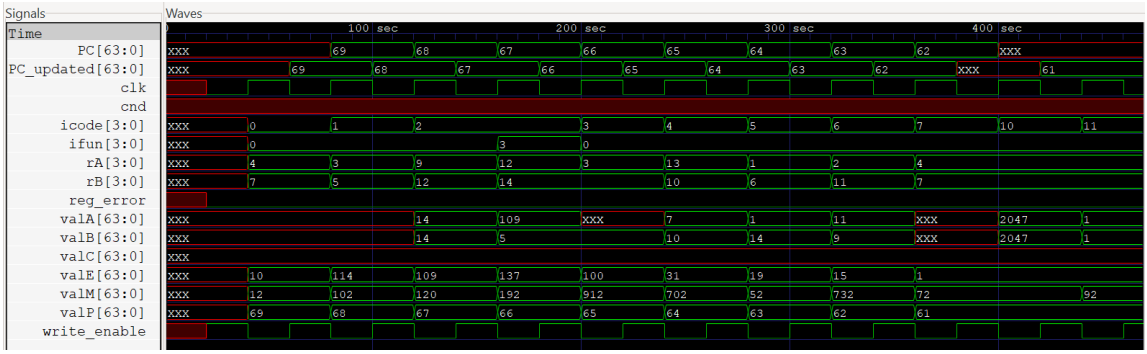
```
#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 10; ifun = 0;
rA = 4; rB = 7;
valE = 1; valM = 72;
write_enable = 0;
PC = PC_updated;

#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 11; ifun = 0;
rA = 4; rB = 7;
valE = 1; valM = 92;
write_enable = 0;

#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 clk = !clk; //clk → 1 : ready to write back
icode = 20; ifun = 0;
rA = 4; rB = 7;
valE = 1; valM = 72;
write_enable = 0;

#20 clk = !clk; //clk → 0 : registers updated
write_enable = 1;

#20 $finish;
```

Different set of testcase for WB stage (randomly made)

```
icode =  x, ifun =  x, rA =  x, rB =  x, we = x, E = x, M = x, P = x, C = x, A = x, B = x, P_up = x, PC = x
icode =  x, ifun =  x, rA =  x, rB =  x, we = 1, E = x, M = x, P = x, C = x, A = x, B = x, P_up = x, PC = x
icode =  0, ifun =  0, rA =  4, rB =  7, we = 0, E = 10, M = 12, P = 69, C = x, A = x, B = x, P_up = x, PC = x
icode =  0, ifun =  0, rA =  4, rB =  7, we = 1, E = 10, M = 12, P = 69, C = x, A = x, B = x, P_up = 69, PC = x
icode =  1, ifun =  0, rA =  3, rB =  5, we = 0, E = 114, M = 102, P = 68, C = x, A = x, B = x, P_up = 69, PC = 69
icode =  1, ifun =  0, rA =  3, rB =  5, we = 1, E = 114, M = 102, P = 68, C = x, A = x, B = x, P_up = 68, PC = 69
icode =  2, ifun =  0, rA =  9, rB = 12, we = 0, E = 109, M = 120, P = 67, C = x, A = 14, B = 14, P_up = 68, PC = 68
 PC = 63
icode =  6, ifun =  0, rA =  2, rB = 11, we = 1, E = 15, M = 732, P = 62, C = x, A = 11, B = 9, P_up = 62, PC = 63
icode =  7, ifun =  0, rA =  4, rB =  7, we = 0, E = 1, M = 72, P = 61, C = x, A = x, B = x, P_up = 62, PC = 62
icode =  7, ifun =  0, rA =  4, rB =  7, we = 1, E = 1, M = 72, P = 61, C = x, A = x, B = x, P_up = x, PC = 62
icode = 10, ifun =  0, rA =  4, rB =  7, we = 0, E = 1, M = 72, P = 61, C = x, A = 2047, B = 2047, P_up = x, PC = x
icode = 10, ifun =  0, rA =  4, rB =  7, we = 1, E = 1, M = 72, P = 61, C = x, A = 2047, B = 2047, P_up = 61, PC = x
icode = 11, ifun =  0, rA =  4, rB =  7, we = 0, E = 1, M = 92, P = 61, C = x, A = 1, B = 1, P_up = 61, PC = x
icode = 11, ifun =  0, rA =  4, rB =  7, we = 1, E = 1, M = 92, P = 61, C = x, A = 1, B = 1, P_up = 61, PC = x
icode =  4, ifun =  0, rA =  4, rB =  7, we = 0, E = 1, M = 72, P = 61, C = x, A = 92, B = 2, P_up = 61, PC = x
icode =  4, ifun =  0, rA =  4, rB =  7, we = 1, E = 1, M = 72, P = 61, C = x, A = 92, B = 2, P_up = 61, PC = x
```

Above is the result obtained on running the given testcase

P.T.O



Above is the GTK plot obtained on plotting the shown result

2. Test-cases for the wrapper code

Wrapper Code:

```
`include "fetch.v"
`include "decode_reg_block.v"
`include "execute_block.v"
`include "memory.v"

module memory_execute_decode_fetch_tb;
    reg [63:0] PC;
    reg clk;

    wire [3:0] icode;
    wire [3:0] ifun;
    wire [3:0] rA;
    wire [3:0] rB;
    wire [63:0] valC;
    wire [63:0] valP;
    wire imem_error;
    wire func_error;
    wire halt, nop;
    wire [63:0] valA,valB;
    wire [63:0] PC_updated;

    reg reg_write_enable;
    reg mem_write_enable;
    wire dmem_error;
    wire reg_error;
    wire [63:0] valE, valM;
    wire cnd, SF, ZF, OF;

    reg [3:0] Stat; // AOK:ADR:INS:HLT
```

```
fetch f1(.PC(PC),.icode(icode),.ifun(ifun),.rA(rA),.rB(rB),.valC(valC),.valP(valP),.imem_error(imem_error),.func_error(func_

decode_reg_block d1(clk, cnd, icode, ifun, rA, rB, reg_write_enable, valE, valM, valP, valC, valA, valB, PC_updated, reg_err

execute_block e1(.vala(valA),.valb(valB),.valc(valC),.icode(icode),.ifun(ifun),.vale(valE),.cnd(cnd),.SF(SF),.ZF(ZF),.OF(OF)

memory m1(clk, icode, ifun, valA, valB, valC, valP, valE, valM, dmem_error);

decode_reg_block wb1(clk, cnd, icode, ifun, rA, rB, reg_write_enable, valE, valM, valP, valC, valA, valB, PC_updated, reg_er
```

```
always @(*) begin
    if (icode = 4'b0000) begin
        Stat = 4'b0001;
    end
    else if (imem_error = 1 || dmem_error = 1) begin
        $display("Error! Invalid addressing.\n");
        Stat = 4'b0100;
    end
    else if (func_error = 1) begin
        $display("Error! Invalid instruction.\n");
        Stat = 4'b0010;
    end
end
```

```
always @ (*) begin
    // $display("Stat = %d", Stat);
    if (Stat == 4'b0001) begin
        $display("Halt instruction encountered!\n");
        $finish;
    end
    if (Stat == 4'b0010) begin
        $display("Halted due to invalid instruction!\n");
        $finish;
    end
    if (Stat == 4'b0100) begin
        $display("Halted due to invalid addressing!\n");
        $finish;
    end
end

always begin
    #0.50 clk = !clk;
end

always @ (posedge clk) begin
    reg_write_enable = 0;
end

always @ (PC_updated) begin
    PC = PC_updated;
end

always @ (negedge clk) begin
    reg_write_enable = 1;
end
```

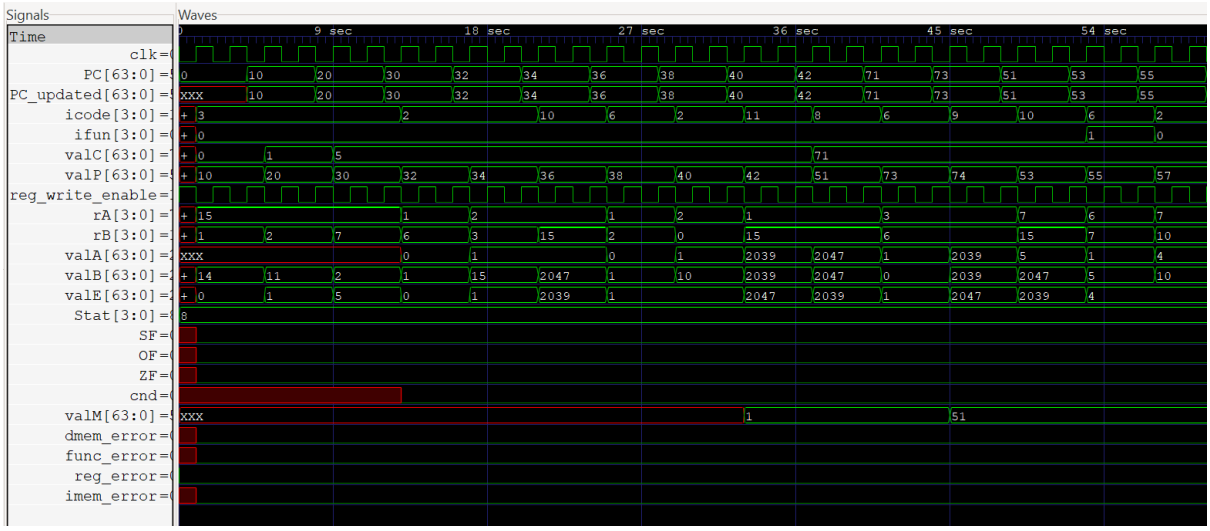
Above set of figures show the wrapper code for SEQ processor design

In the third image (from the top out of the above four) is the one which shows the calculation and setting of the Status Codes and the last image shows the effect the status codes will have on the running of the programme. In our SEQ design, if the Status Codes show any other sign than AOK, then we halt the processor at whatever stage it is.

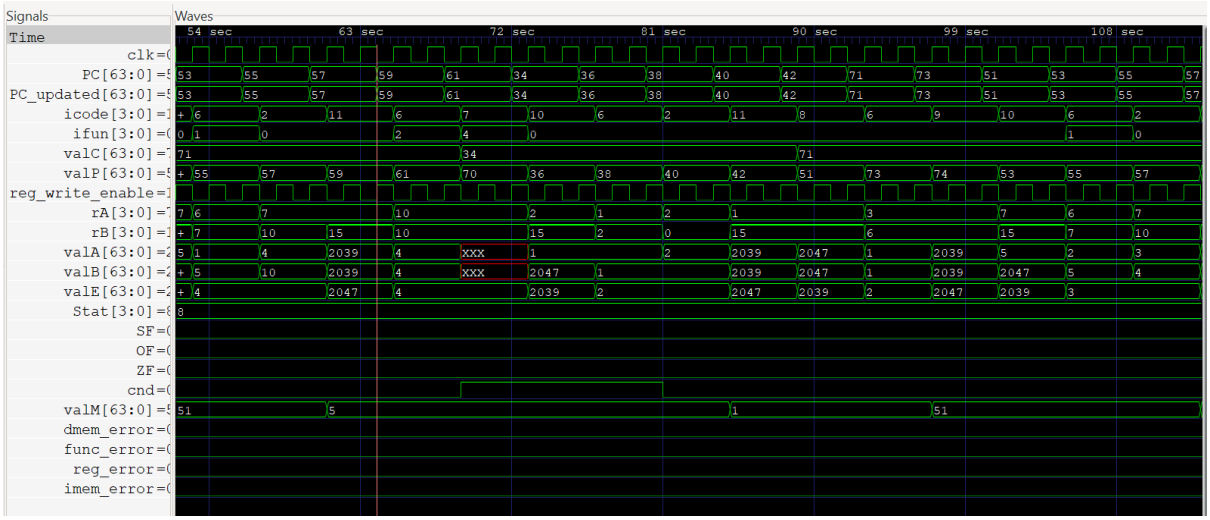
The following test-case for the Fibonacci Series has been used to test the wrapper code:

```
00110000 // irmovq
11110001 // F, %rcx
00000000 // $0
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00110000 // irmovq
11110010 // F, %rdx
00000001 // &1
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00110000 // irmovq
11110111 // F, %rdi
00000101 // $5
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00100000 // rrmovq
00010110 // %rcx, %rsi
00100000 // rrmovq
00100011 // %rdx, %rbx
10100000 // pushq → fib_loop
00101111 // %rdx
01100000 // addq
00010010 // %rcx, %rdx
00100000 // rrmovq
00100000 // %rdx, %rax
10110000 // popq
00011111 // %rbx
10000000 // call inc
01000111 // $71
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
10100000 // pushq
01111111 // %rdi
01100001 // subq
01100111 // %rsi, %rdi
00100000 // rrmovq
01111010 // %rdi, %r10
10110000 // popq
01111111 // %rdi
01100010 // andq
10101010 // %r10, %r10
01110100 // jne fib_loop
00100010 // $ 34
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000 // halt
01100000 // addq → inc
00110110 // %rbx, %rsi
10010000 // ret
```

Above is the testcase for the Fibonacci Series that we wrote for the Wrapper Code First two numbers of the series are taken to be 0 and 1 and the code calculates up to 5 numbers after 0 and 1 – 1, 2, 3, 5, 8

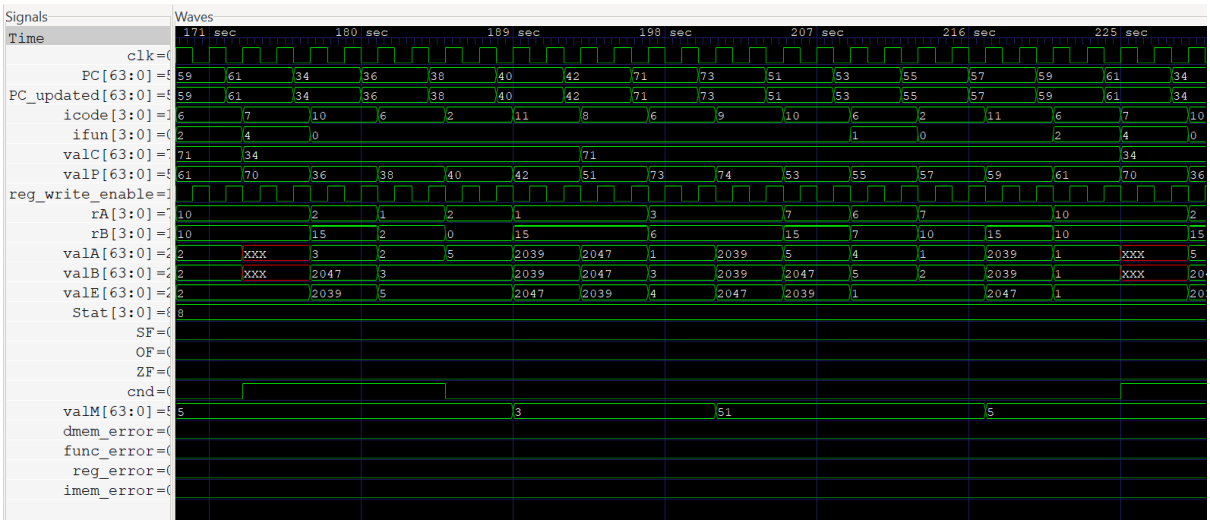


Here, valA is where Fibonacci series shows up | valA = 1

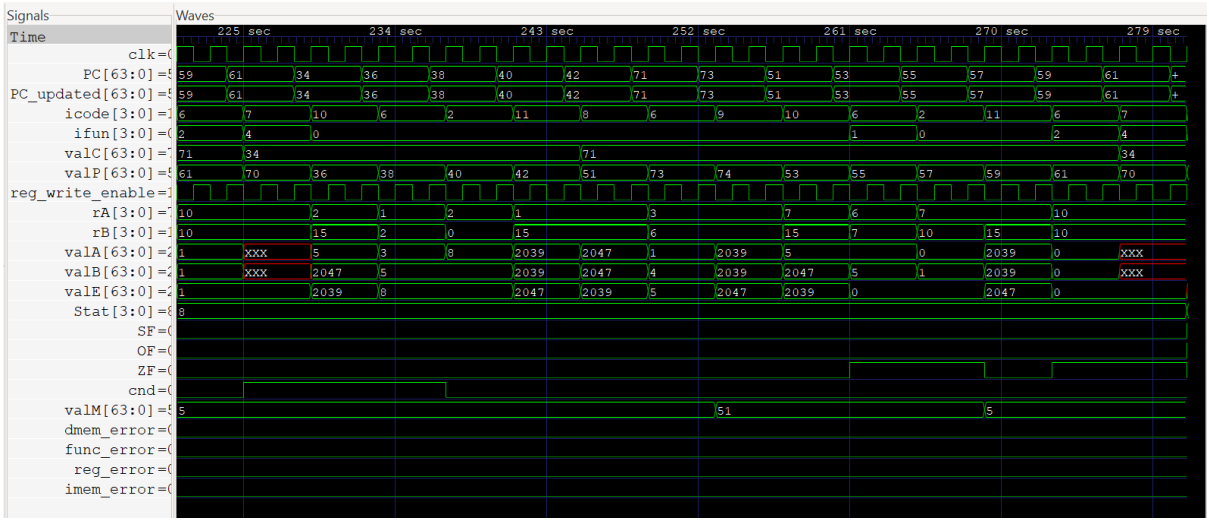


Here, valA is where Fibonacci series shows up | valA = 2

Here, valA is where Fibonacci series shows up | valA = 3



Here, valA is where Fibonacci series shows up | valA = 5

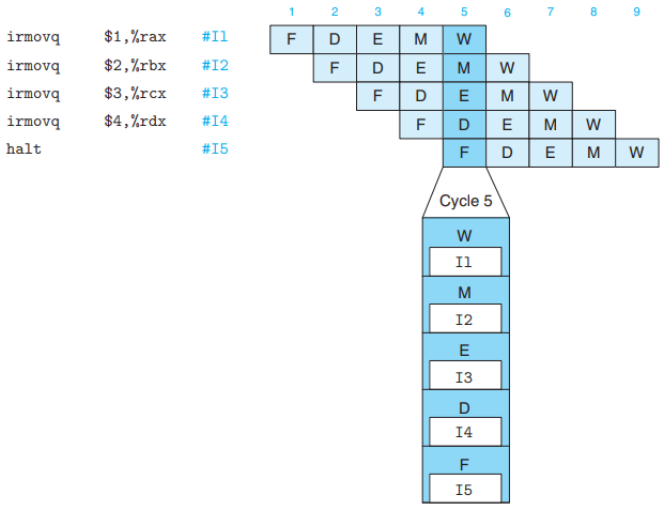


Here, valA is where Fibonacci series shows up | valA = 8

Section 2 – Pipelined Processor Design

Introduction

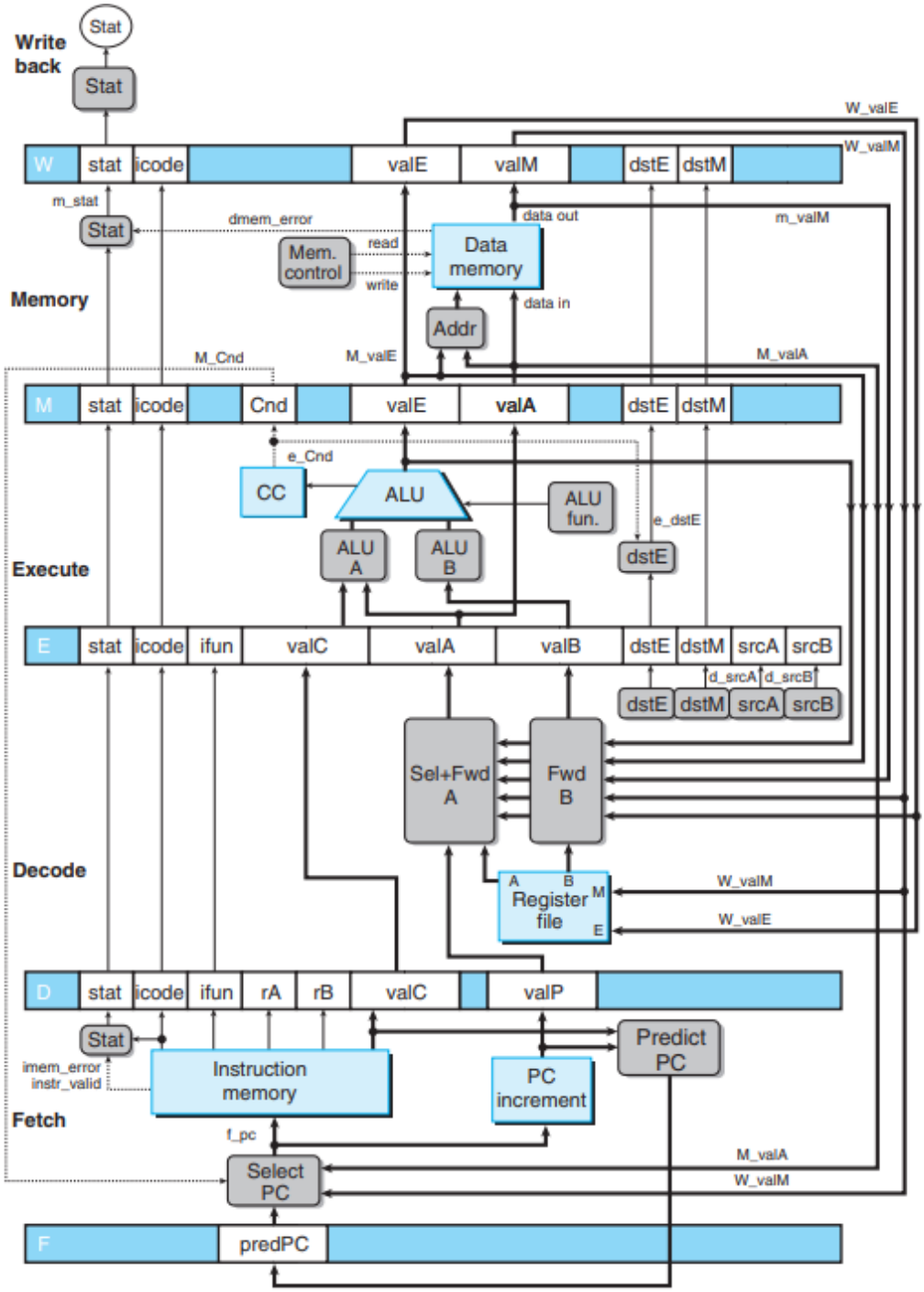
In the pipelined implementation of the Y86 processor, all the stages of the processor work on different instructions at the same time. Each stage of the processor carries out the required task within a single clock cycle. For example, in the below image we can see that in clock cycle 5, all 5 stages are carring out different instructions:



Hence, a new instruction enters the processor at each positive edge of the clock and an instruction is written back at each negative edge of the clock. Therefore, the throughput of this kind of a processor is 1 instruction per clock cycle which is much faster than the sequential implementation of the prcocessor which required 5 clock cycles to carry out each instruction. In order to make a pipelined Y86 processor, the following changes are made to the sequenctial processor:

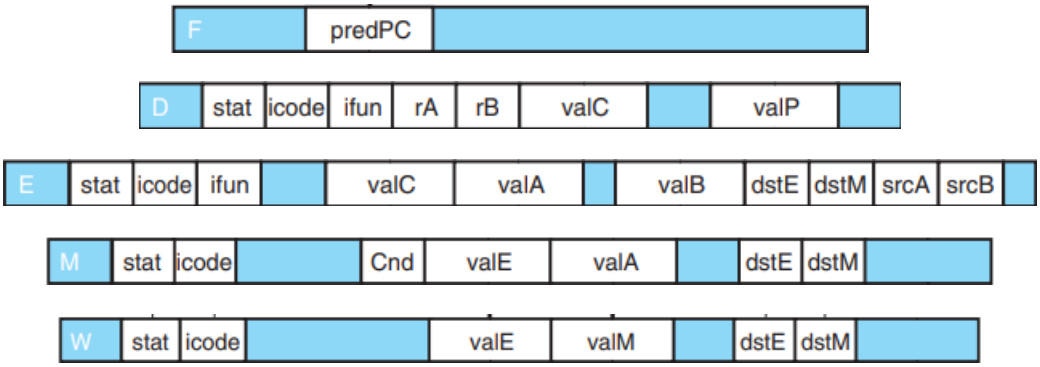
1. Pipeline registers are introduced before each stage which help in storing the information regarding the instruction being executed.
2. PC update stage comes at the beginning of the clock cycle, rather than at the end.
3. We perform PC prediction to predict the next value of the PC in order to deal with the jump and return instructions.
4. The naming of different signals is changed. The uppercase prefixes ‘D’, ‘E’, ‘M’, and ‘W’ refer to pipeline registers, and so X_name refers to the input given to stage X and x_name refers to the output signal generated in the stage X.
5. We introduce a pipeline control logic block which helps us to introduce bubble or stall a particular stall which helps us to resolve certain problems with pipelining which will be discussed later.

The pipelined Y86 has the structure as shown below:



Pipeline Registers

Pipeline registers are the registers that are inserted before each of the stages of the processor. There are 5 pipeline registers – F, D, E, M and W. They are named based on the stage they give input to. The pipeline registers consist of all the inputs that need to be given to the corresponding stage for carrying out the desired operation for the current instruction. The name of all the values stored in it are of the form X_reg, where X is the pipeline register name(F,D,E,M,W) and reg is the name of the register. The pipeline registers for each of the stages is shown below:



For example, the Verilog code for the decode pipeline register (D) is shown below:

```
input clk, D_stall, D_bubble;
input [3:0] f_stat, f_icode, f_ifun, f_rA, f_rB;
input [63:0] f_valC, f_valP;
output reg [3:0] D_stat, D_icode, D_ifun, D_rA, D_rB;
output reg [63:0] D_valC, D_valP;

reg [3:0] stat, icode, ifun, rA, rB;
reg [63:0] valC, valP;

always@(*)
begin
    stat = f_stat;
    icode = f_icode;
    ifun = f_ifun;
    rA = f_rA;
    rB = f_rB;
    valC = f_valC;
    valP = f_valP;
end
```

```
always@(posedge clk)
begin
    if (!D_stall && !D_bubble) begin
        D_stat <= stat;
        D_icode <= icode;
        D_ifun <= ifun;
        D_rA <= rA;
        D_rB <= rB;
        D_valC <= valC;
        D_valP <= valP;
    end
    else if (D_bubble && !D_stall) begin
        D_stat <= 4'bx;
        D_icode <= 4'b0001;
        D_ifun <= 4'b0000;
        D_rA <= 4'bx;
        D_rB <= 4'bx;
        D_valC <= 64'bx;
        D_valP <= 64'bx;
    end
end
```

In this code we first define the inputs, outputs of the register module as well as internal registers. Next, we store the input values in the internal registers and then at the positive edge of the clock, we give these stored values to the outputs is there is no bubble and no stall. If there is bubble then we give the output as don't care or 0 in certain cases.

Hazards and Solutions:

1. **Data Dependencies**

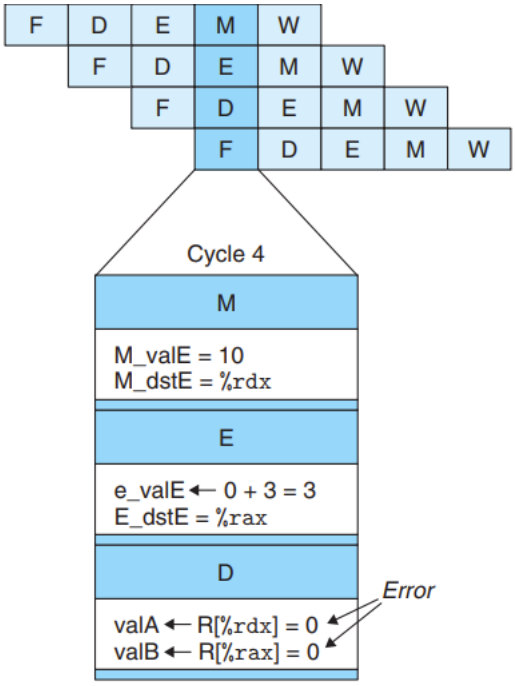
   In the pipelined processor, data dependencies become a major obstacle while achieving a smooth working.

   The general cases of data hazard involve the occurrence of operational instructions like **OPq** after data movement instructions like **irmovq** and **rrmovq** or **cmovxx**. Other special case of data dependency occurs when an operational or a **use** instruction like **OPq** follows a movement or a **load** instruction like **mrmovq** or **popq**. We will look at both separately.

   a. **General Case:**

      Here, the following images depict the hazard that we face:

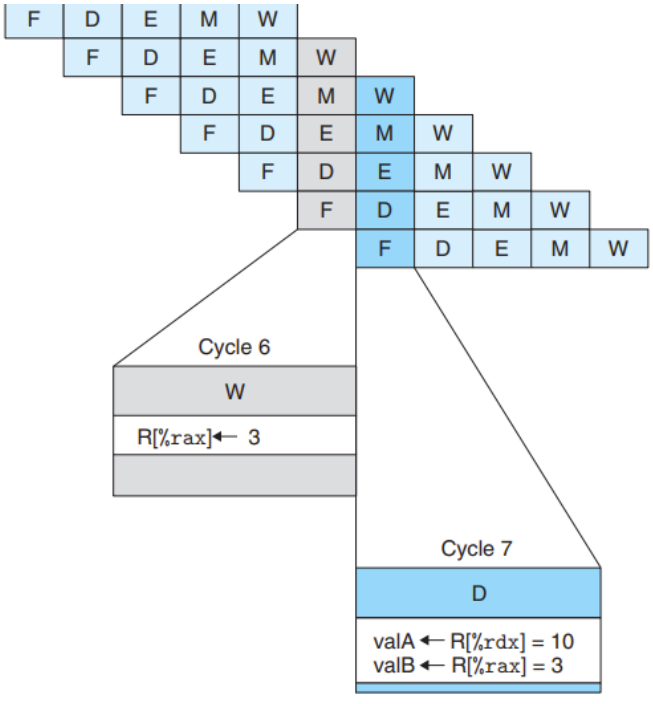
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



As indicated by the extrapolated block, the values in **%rax** and **%rdx** are being used by **addq** even before the earlier instructions can update the register file, that is, even before they reach the WB stage.

The most generic way of solving the issue is delaying the instruction **addq** till the above both instructions cross WB stage, which we achieve by adding 3 **nop** instructions before the operational instruction. This way the operational instruction will reach the decode stage only after the register files are updated with the right data. Done as follows:

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```

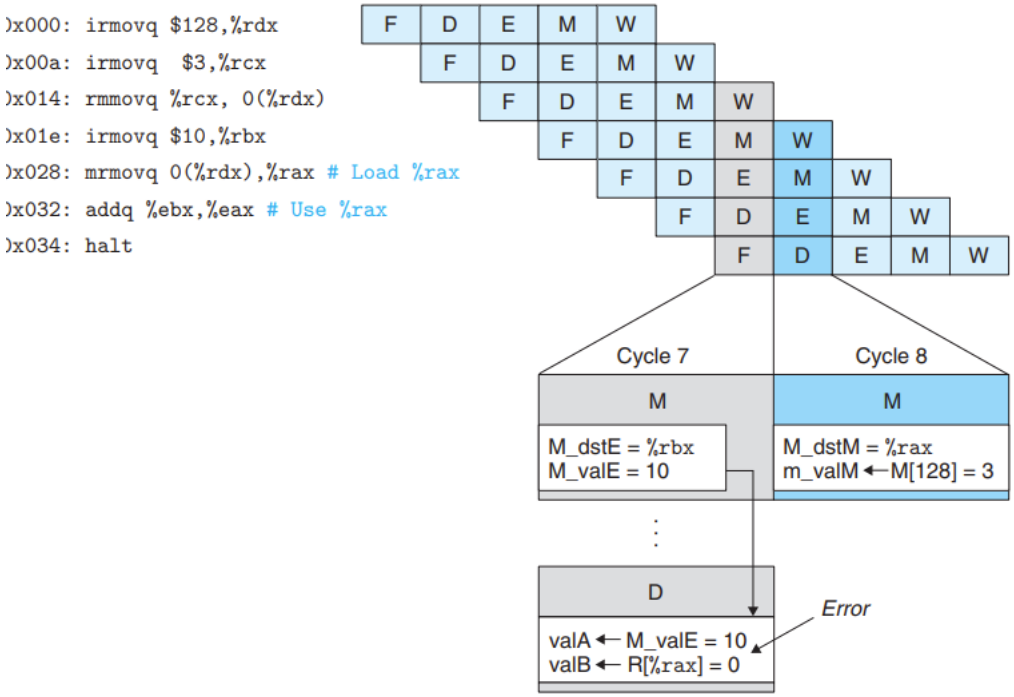


Clearly, the issue of data forwarding is solved. This is called **Stalling.** Note, however, that if the number of **nops** used any less number of times than 3,

then the issue of data dependency persists due to either of the earlier instructions having still not crossed the WB stage.

b. **Load/Use:**

Here, as mentioned earlier, the hazard occurs because the act of loading finishes when the data is brought from the memory block and written into the register file. This is different from the earlier hazard, in that the data can only be taken after the earlier data movement instruction crosses the memory **and** WB stages. Issue shown in the figure below:



Solutions:

The widely used and an optimum solution is taken assistance of and thereby implementing the technique of **Data forwarding**. Data forwarding simply implies setting up feedback paths from various stages ahead of the decode stage in order to eliminate the need to stall or hold the instructions at decode stage in case of data dependencies. We detect the data dependency and need to use data forwarding by checking whether any of the source destinations of the current instruction in the decode stage matches with any of the "possible" write-back destinations that may be in any of the upper stages. We "forward" or draw feedback from the paths such as – **m_valM**, **e_valE**, **M_valE**, **W_valM** and **W_valE** (currently not in correct order).

Now, for the load/use hazard, we still need to use **stall** functionality along with using data forwarding due to the fact that by the time the loading instruction reaches memory stage, the operational instruction would have crossed the decode stage and would be into the execute stage with wrong register values. So, here, we stall the decode stage till the previous loading

instruction reaches the memory stage and then use data forwarding to retrieve the updated value of the register, thereby lifting stall. (Code for this is shown in the **decode** stage)

2. **Control Hazards**
   a. **Mispredicted jump**
      This is a major issue for pipelined architecture which mainly occurs due to the tendency (and requirement) of the pipelined architecture to "predict" the new PC value as soon as the current instruction is fetched. This makes it very much possible for the prediction mechanism to assign the wrong value to PC counter in case of conditional jump instructions. To elaborate, if the predicted PC points at the instruction at the jump destination and if the condition for taking the jump fails which can only be known after the **jXX** instruction reaches the fetch block, then we need to flush out two wrong instructions which have reached the **decode** and **fetch** stages and belong to the "not-to-be-taken" destination.
   b. **ret**
      This hazard comes up again due to wrong prediction of the next PC value but only due to unavailability of the correct PC value. Note that in the working of the **ret** instruction, the new PC value is supposed to be the value which is read from the memory stack – **valM**. Evidently, this value will only be available once the **ret** instructions reaches the memory stage, but the problem being that the instructions right after **ret** which are not supposed to be fetched (because **ret** takes us to the location after **call**) already reach the execute stage and the condition codes of the processor get affected instantly! Thus, this posed to be a grave hazard.

   Solutions:

   The most widespread solution used is that of making use of pipeline functionalities such as **stall** and **bubble**.

   In case of mispredicted jump, we flush out the instructions at fetch and decode stages by inserting a **bubble** in D and E pipelines at the next rising edge of the clock, thereby prevent those two instructions from proceeding to their next stages. No **stalling** required here.

   In case of **ret**, at the next rising clock edge, we **stall** the instruction at the fetch stage and insert a **bubble** in the decode stage. We keep the **stall** and **bubble** signals high until **ret** reaches the right back stage after which with the help of data forwarding we can retrieve the correct new PC value!

Stages of the Processor:

1. **Fetch**

   The major difference between the SEQ fetch and PIPE fetch is that the PIPE version has an inbuilt mechanism for predicting the next PC value in order to uphold parallelism in fetching new instructions while older instructions proceed through the upper stages. The fetch block, therefore, has two separate and crucial components called **selectPC** and **predictPC** alongside the normal circuitry of SEQ design.

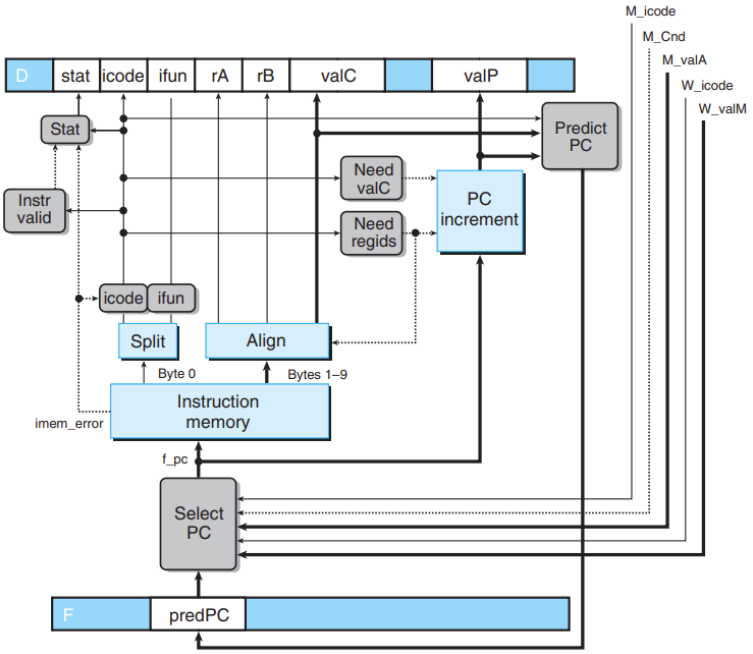


Figure showing the architecture of PIPE fetch stage

Note the presence of feedback paths that are used for choosing the current and thus the next PC value during fetching. Also, one more important thing that takes place in PIPE fetch is the setting of **stat** (the status codes) right from the beginning.

Code:

```
always @ (*) begin //selectPC
    if (M_icode == 4'b0111) begin //if jXX
        if (M_Cnd == 0) begin //misprediction correction
            f_pc = M_valA;
        end
    end
    else if (W_icode == 4'b1001) begin //if ret
        f_pc = W_valM;
    end
    else f_pc = predPC;
end
```

Code for SelectPC block

```
always @ (*) begin //predictPC
    if (f_icode == 4'b0111 || f_icode == 4'b1000) begin
        f_predPC = f_valC;
    end
    else f_predPC = f_valP;
end

always @ (*) begin //status codes
    f_Stat = 4'b1000; //AOK : ADR : INS : HLT
    if (f_icode == 4'b0000) begin
        f_Stat = 4'b0001;
    end
    else if (imem_error == 1) begin
        f_Stat = 4'b0100;
    end
    else if (func_error == 1) begin
        f_Stat = 4'b0010;
    end
end
```

Code for predictPC block and setting status codes

The rest of the code is same as that in SEQ

2. **Decode**

The PIPE decode stage is quite different than its SEQ counterpart, in that it includes the mechanism of data forwarding for handling data dependencies. Also, there are pathways facilitating pipeline-to-pipeline direct pathways. One important point of difference in PIPE decode is in the deciding of **valA**, which can take the value as read from the register file or in some cases carry ahead the value **valP**. This has been successfully incorporated in the code.

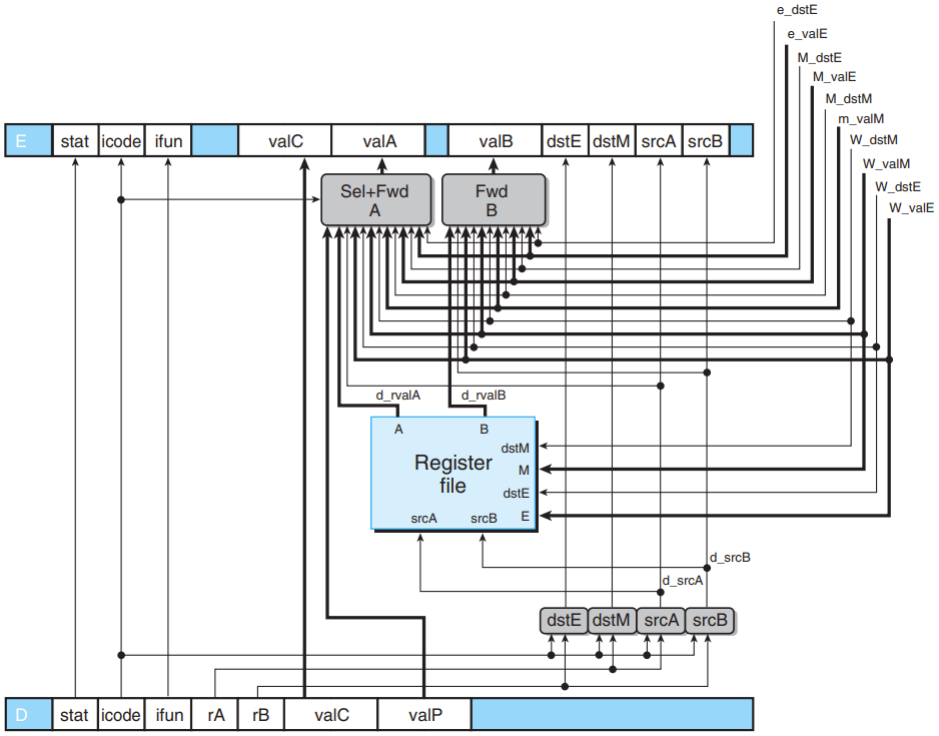


Figure showing architecture for the PIPE decode stage

Code:

```
always @ (*) begin //decode stage operations for valA (including data forwarding)
    if (D_icode == 4'b1000 || D_icode == 4'b0111) begin //call or jXX → use valP
        d_valA = D_valP;
    end
    else if (srcA == e_dstE) begin //if data dependency in execute stage
        d_valA = e_valE;
    end
    else if (srcA == M_dstM) begin //if data dependency in memory stage → valM
        d_valA = m_valM;
    end
    else if (srcA == M_dstE) begin //if data dependency in memory stage → valE
        d_valA = M_valE;
    end
    else if (srcA == W_dstM) begin //if data dependency in write-back stage → valM
        d_valA = W_dstM;
    end
    else if (srcA == W_dstE) begin //if data dependency in write-back stage → valE
        d_valA = W_valE;
    end
    else d_valA = d_rvalA; //default case of decoding as in seq
end
```

Data forwarding for valA

```
always @ (*) begin //decode stage operations for valB (including data forwarding)
    if (srcB == e_dstE) begin //if data dependency in execute stage
        d_valB = e_valE;
    end
    else if (srcB == M_dstM) begin //if data dependency in memory stage → valM
        d_valB = m_valM;
    end
    else if (srcB == M_dstE) begin //if data dependency in memory stage → valE
        d_valB = M_valE;
    end
    else if (srcB == W_dstM) begin //if data dependency in write-back stage → valM
        d_valB = W_dstM;
    end
    else if (srcB == W_dstE) begin //if data dependency in write-back stage → valE
        d_valB = W_valE;
    end
    else d_valB = d_rvalB; //default case of decoding as in seq
end
```

Data forwarding for valB

```
always @ (*) begin //logic for dstE and dstM
    if (D_icode == 4'b1000 || D_icode == 4'b1010) begin //call or pushq
        dstE = 4'b0100;
        dstM = 4'bx;
    end
    else if (D_icode == 4'b1001 || D_icode == 4'b1011) begin //ret or popq
        dstE = 4'b0100;
        dstM = D_rA;
    end
    else if (D_icode == 4'b0101) begin //mrmovq
        dstM = D_rB;
        dstE = 4'bx;
    end
    else begin
        dstE = D_rB;
        dstM = 4'bx;
    end
    d_dstE = dstE; //pipeline - pipeline direct connections
    d_dstM = dstM; //pipeline - pipeline direct connections
end
```

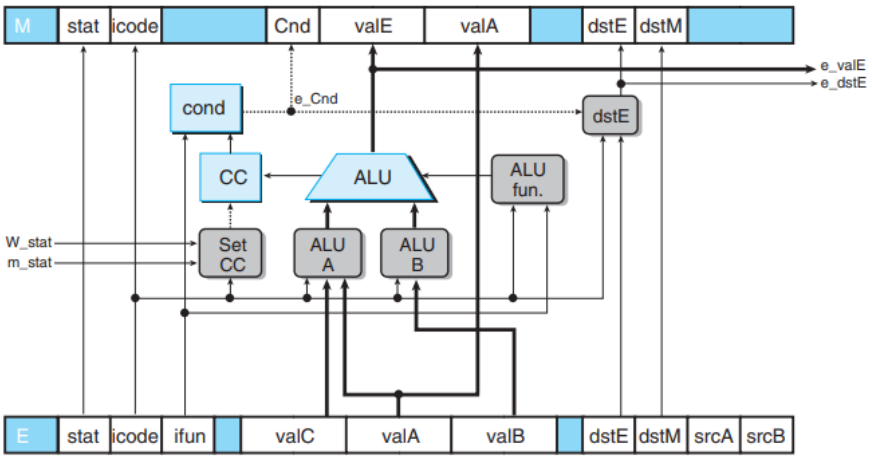
Selection of values for dstE and dtsM

```
always @ (*) begin //logic for srcA and srcB
    if (D_icode == 4'b0011) begin //irmovq
        srcA = 15;
        srcB = D_rB;
    end
    else if (D_icode == 4'b1010) begin //pushq
        srcA = D_rA;
        srcB = 4'b0100;
    end
    else if (D_icode == 4'b1000 || D_icode == 4'b1001 || D_icode == 4'b1011) begin //call, ret,
        srcA = 4;
        srcB = 4;
    end
    else begin
        srcA = D_rA;
        srcB = D_rB;
    end
    d_srcA = srcA; //pipeline - pipeline direct connections
    d_srcB = srcB; //pipeline - pipeline direct connections

    if (D_icode == 4'b0101) begin //mrmovq
        d_rvalA = storage[srcB];
        d_rvalB = storage[srcA];
    end
    else begin
        d_rvalA = storage[srcA];
        d_rvalB = storage[srcB];
    end
end
```

Selection of values for srcA and srcB

3. **Execute**

The execute stage is responsible for carrying out arithmetic and logic operations on the data valA and valB and giving the output valE and also for computing the condition bit 'cnd'. The below image shows the structure of the execute stage:



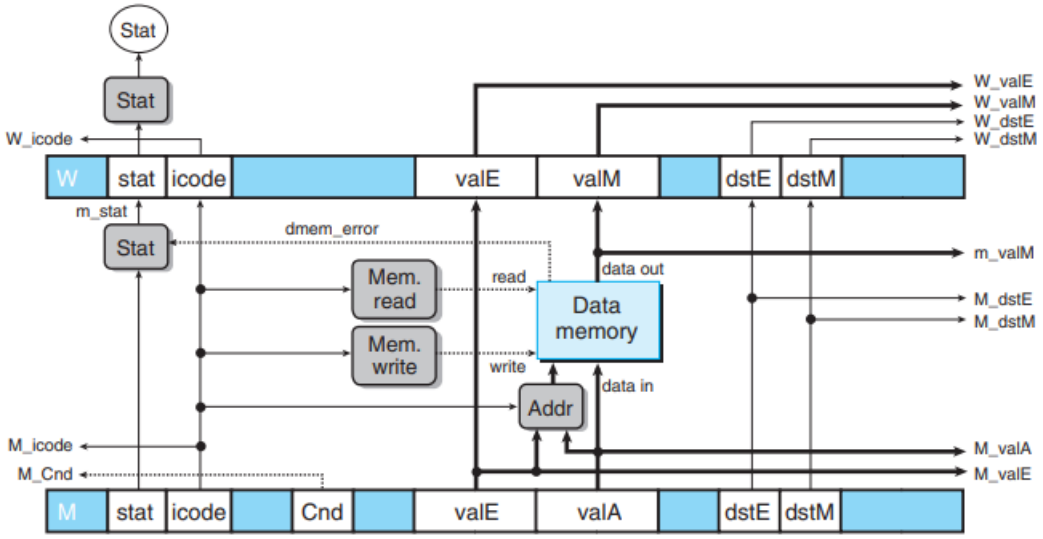
The only difference between the execute stage of sequential and pipelined implementation is that the name of the input and output signals are different and that the values e_valE and e_dstE are sent to the decode stage and the pipeline control logic stage which can be useful for data forwarding, avoiding hazards and for inserting bubble and stall.

The Verilog code of the pipelined execute stage is almost same as that of the sequential execute stage. The only differerences are that the names of the input and output signals are changed and secondly we directly pass on the certain values directly from input to output which are not getting changed within the block as shown below:

```
always@(*)
begin
    e_stat = E_stat;
    e_icode = E_icode;
    e_dstE = E_dstE;
    e_dstM = E_dstM;
    e_valA = E_valA;
end
```

4. **Memory**

The role of the memory stage is to read and write data in the memory. This includes the push and pop operations that are performed on the stack present in the memory. The structure of the memory stage is as shown below:



The difference between the sequential and pipelined memory stage is that the name of input and output signals are different. Also, some values such as m_valM, M_valA, M_valE and M_icode are given to the decode stage and the pipeline control stage which would be useful for data forwarding, avoiding hazards and for inserting bubble and stall.

The Verilog code for the pipelined memory stage is almost same as that of the sequential memory stage. One of things that we have added is as shown below:

```
always@(*)
begin
    m_icode = M_icode;
    m_valE = M_valE;
    m_dstE = M_dstE;
    m_dstM = M_dstM;
end
```

Here, we directly pass on the certain values directly from input to output which are not getting changed within the block. The other thing that we have added is that we assign the 2nd bit of the status register, that is, ADR

as dmem_error which is 1 if there is an invalid error, else 0 as shown below:

```
always@(*)
begin
    m_stat = M_stat;
    m_stat[2] = dmem_error;
end
```

5. **Write-back**

   Write-back stage in PIPE architecture is one of the easiest stages for the sheer simplicity in implementing it. Write-back has no conditions to follow, in that it has to straightforwardly write into the register file.

   The architecture design for this stage is same as the decode block because after all both function with the same register file.

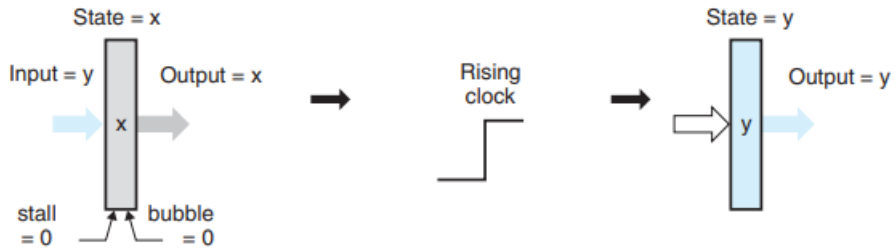
   Code:

```
always @ (*) begin //write-back stage operations
    // if (write_enable == 1) begin
        storage[W_dstE] = W_valE;
        storage[W_dstM] = W_valM;
    end
// end
```

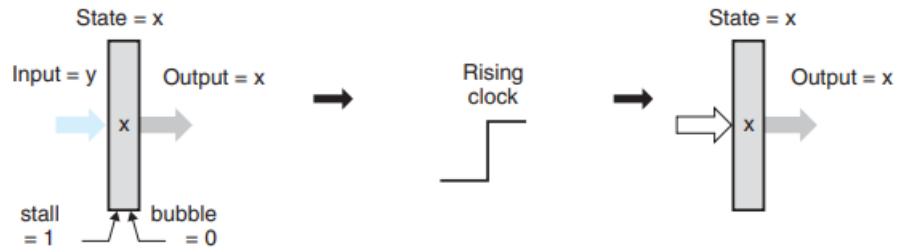
Code for PIPE write-back stage

Pipeline Control Logic

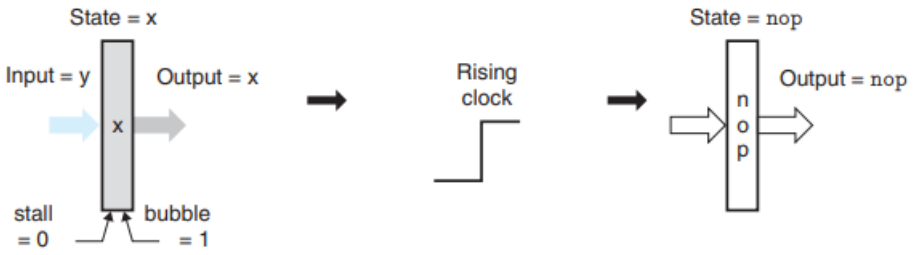
The pipeline control logic is a block in the piplelined Y86 architecture processor which helps us to resolving the issues caused by load/use hazards, return intruction and mispredicted jump by inserting bubbles in certain stages and/or stalling certain stages based on our requirement. A normal instruction is of the below form where the input is y and the output will also be is y:



Now, when we insert a bubble then for the input y, the output will be don't care(x):



Also, when we stall a stage, for input y, the output will be a nop instruction :



The below image shows the Verilog code of the pipeline control logic:

```
module pipe_control_logic(W_Stat, m_stat, M_icode, E_dstM, E_icode, d_srcB, d_srcA, D_icode, e_cnd, W_stall, M_bubble, cnd, E_bubble, D_bubble, D_stall, F_stall);

input [3:0] W_Stat, m_stat, M_icode, E_dstM, E_icode, d_srcB, d_srcA, D_icode ;
input e_cnd;
output reg W_stall, M_bubble, cnd, E_bubble, D_bubble, D_stall, F_stall;

always@(*)
begin
    M_bubble = 0;
    W_stall = 0;
    if ((((E_icode == 5) || (E_icode == 11)) && ((E_dstM == d_srcA) || (E_dstM == d_srcB))) || ((D_icode == 9)||(E_icode == 9)||(M_icode == 9))
    || (D_icode == 4'b0000))
    begin
        F_stall = 1;
    end
    else F_stall = 0;

    if(((E_icode == 7) && (e_cnd == 0)) || ((E_icode != 5)&&(E_icode != 11) || ((E_dstM != d_srcA)&&(E_dstM != d_srcB))) && ((D_icode == 9)||(E_icode == 9)||(M_icode == 9)))
    begin
        D_bubble = 1;
    end
    else D_bubble = 0;

    if(((E_icode == 7) && (e_cnd == 0)) || (((E_icode == 5) || (E_icode == 11)) && ((E_dstM == d_srcA) || (E_dstM == d_srcB)))
    || (D_icode == 4'b0000))
    begin
        E_bubble = 1;
    end
    else begin
        E_bubble = 0;
    end

    if ((((E_icode == 5) || (E_icode == 11)) && ((E_dstM == d_srcA) || (E_dstM == d_srcB)))
    || (D_icode == 4'b0000)) begin
        D_stall = 1;
    end
    else D_stall = 0;
```

The code starts with writing down the inputs that will be required to figure out when and where to insert the bubble and stall. The outputs of this module includes E_bubble, D_bubble, F_stall, D_stall, W_stall and M_bubble. Next, inside the always we have different cases:

1. If we detect load/use hazard or control hazard due to return instruction then we stall in fetch stage.
2. If we detect mispredicted branch or a control hazard due to return instruction then we introduce a bubble in the decode stage.

3. If we detect mispredicted branch and load/use hazard simultaneously then we introduce bubble in execute stage.

If we detect load/use hazard then we will need to stall decode stage.

Testing and Results:

1. **Test-cases for the individual blocks**
   a. **Fetch**

```
ROM[0] = 8'b00110000; //irmovq
ROM[1] = 8'b11110011;
ROM[2] = 8'b00000010;
ROM[3] = 8'b00000000;
ROM[4] = 8'b00000000;
ROM[5] = 8'b00000000;
ROM[6] = 8'b00000000;
ROM[7] = 8'b00000000;
ROM[8] = 8'b00000000;
ROM[9] = 8'b00000000;
ROM[10] = 8'b00100000; //rrmovq
ROM[11] = 8'b00111011;
ROM[12] = 8'b01100000; //addq
ROM[13] = 8'b10110011;
ROM[14] = 8'b10000000; //call
ROM[15] = 8'b00011001;
ROM[16] = 8'b00000000;
ROM[17] = 8'b00000000;
ROM[18] = 8'b00000000;
ROM[19] = 8'b00000000;
ROM[20] = 8'b00000000;
ROM[21] = 8'b00000000;
ROM[22] = 8'b00000000;
ROM[23] = 8'b10110000; //popq
ROM[24] = 8'b00111111;
ROM[25] = 8'b10100000; //pushq → call dst
ROM[26] = 8'b00111111;
ROM[27] = 8'b10010000; //ret
```

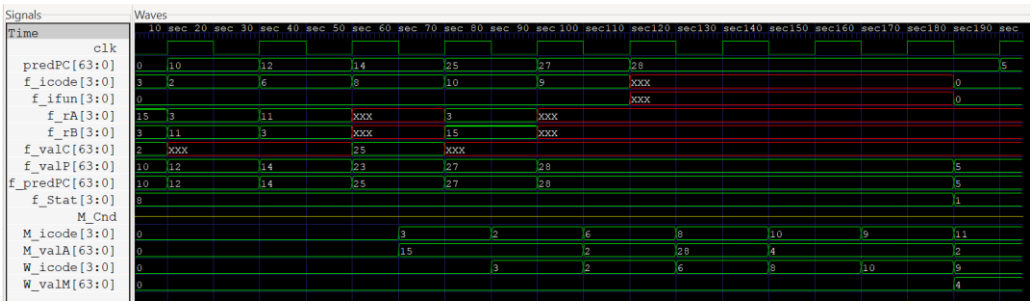
Testcase for PIPE fetch – rmmov, mrmov and jmp is tested in the final wrapper

```
clk = 0, fic = 3, fif = 0, frA = 15, frb = 3, C = 2, P = 10, f_predPC = 10, f_Stat = 8
clk = 1, predicted Pc = 10
clk = 0, fic = 2, fif = 0, frA = 3, frb = 11, C = x, P = 12, f_predPC = 12, f_Stat = 8
clk = 1, predicted Pc = 12
clk = 0, fic = 6, fif = 0, frA = 11, frb = 3, C = x, P = 14, f_predPC = 14, f_Stat = 8
clk = 1, predicted Pc = 14
clk = 0, fic = 8, fif = 0, frA = x, frb = x, C = 25, P = 23, f_predPC = 25, f_Stat = 8
clk = 1, predicted Pc = 25
clk = 0, fic = 10, fif = 0, frA = 3, frb = 15, C = x, P = 27, f_predPC = 27, f_Stat = 8
clk = 1, predicted Pc = 27
```

Result for the given testcase (half)

```
clk = 0, fic = 9, fif = 0, frA = x, frb = x, C = x, P = 28, f_predPC = 28, f_Stat = 8
clk = 1, predicted Pc = 28
clk = 0, fic = x, fif = x, frA = x, frb = x, C = x, P = 28, f_predPC = 28, f_Stat = 8
clk = 1, predicted Pc = 28
clk = 0, fic = x, fif = x, frA = x, frb = x, C = x, P = 28, f_predPC = 28, f_Stat = 8
clk = 1, predicted Pc = 28
clk = 0, fic = x, fif = x, frA = x, frb = x, C = x, P = 28, f_predPC = 28, f_Stat = 8
clk = 1, predicted Pc = 28
clk = 0, fic = 0, fif = 0, frA = x, frb = x, C = x, P = 5, f_predPC = 5, f_Stat = 1
clk = 1, predicted Pc = 5
```

Result for the given testcase (remaining half)



GTK plot for the above result

Note that the last few cycles show “x” values due to presence of the ret function at the end which has to wait till it goes to WB.

b. **Decode**

```
clk = 0;

#5 clk = !clk;
D_icode = 3; D_ifun = 0; D_Stat = 8;
D_rB = 3; D_valC = 2; D_rA = 15; D_valP = 10;

#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %0d, A
d_icode, d_ifun, d_valC, d_valA, d_valB, d_dst

#5 clk = !clk;
D_icode = 2; D_ifun = 0; D_Stat = 8;
D_rB = 11; D_valC = 2; D_rA = 3; D_valP = 12;
e_dstE = 3; e_valE = 2;

#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %0d, A
d_icode, d_ifun, d_valC, d_valA, d_valB, d_dst

#5 clk = !clk;
D_icode = 6; D_ifun = 0; D_Stat = 8;
D_rB = 3; D_valC = 2; D_rA = 11; D_valP = 14;
e_dstE = 3; e_valE = 2;
M_dstE = 11; M_valE = 2;
```

```
#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %0d,
d_icode, d_ifun, d_valC, d_valA, d_valB, d_

#5 clk = !clk;
D_icode = 8; D_ifun = 0; D_Stat = 8;
D_valC = 25; D_valP = 23;

#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %0d,
d_icode, d_ifun, d_valC, d_valA, d_valB, d_

#5 clk = !clk;
D_icode = 10; D_ifun = 0; D_Stat = 8;
D_valC = 25; D_valP = 27;
M_valE = 4; M_dstE = 11;

#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %0d,
d_icode, d_ifun, d_valC, d_valA, d_valB, d_

#5 clk = !clk;
D_icode = 9; D_ifun = 0; D_Stat = 8;
D_valC = 25; D_valP = 28;
```

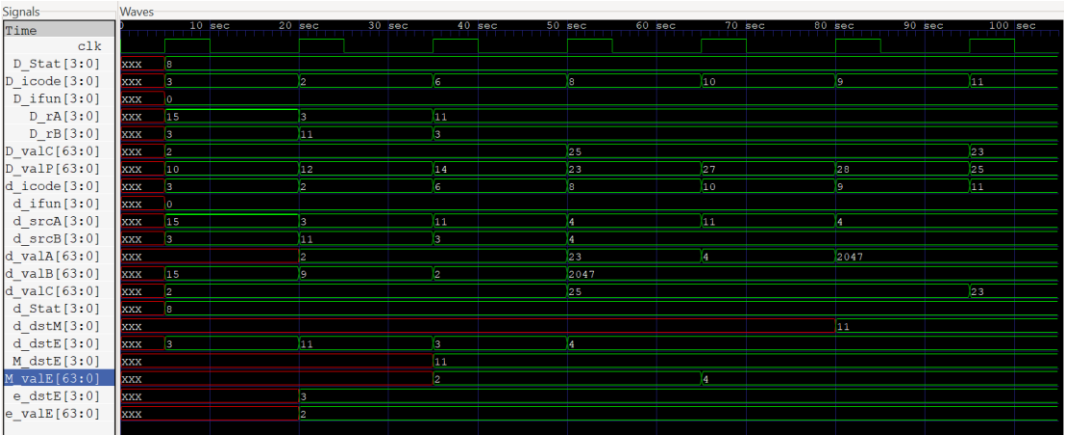
```
#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %
d_icode, d_ifun, d_valC, d_valA, d_valB,

#5 clk = !clk;
D_icode = 11; D_ifun = 0; D_Stat = 8;
D_valC = 23; D_valP = 25;

#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %
d_icode, d_ifun, d_valC, d_valA, d_valB,
```

Testcases based on previous testcase used in fetch

P.T.O

```
dic = 3, dif = 0, C = 2, A = x, B = 15, dstE = 3, dtsM = x, srcA = 15, srcB = 3, d_stat = 8

dic = 2, dif = 0, C = 2, A = 2, B = 9, dstE = 11, dtsM = x, srcA = 3, srcB = 11, d_stat = 8

dic = 6, dif = 0, C = 2, A = 2, B = 2, dstE = 3, dtsM = x, srcA = 11, srcB = 3, d_stat = 8

dic = 8, dif = 0, C = 25, A = 23, B = 2047, dstE = 4, dtsM = x, srcA = 4, srcB = 4, d_stat = 8

dic = 10, dif = 0, C = 25, A = 4, B = 2047, dstE = 4, dtsM = x, srcA = 11, srcB = 4, d_stat = 8

dic = 9, dif = 0, C = 25, A = 2047, B = 2047, dstE = 4, dtsM = 11, srcA = 4, srcB = 4, d_stat = 8

dic = 11, dif = 0, C = 23, A = 2047, B = 2047, dstE = 4, dtsM = 11, srcA = 4, srcB = 4, d_stat = 8
```

Results for the above testcases



GTK plot for shown results

c. **Execute**

The below image shows the output of the pipelined execute stage for various E_icode and E_ifun values:

```
E_icode = 4, E_ifun = 1, E_valA = 9, E_valB = 1, E_valC = 11, e_valE = x, e_cnd = x
E_icode = 4, E_ifun = 1, E_valA = 9, E_valB = 1, E_valC = 11, e_valE = 12, e_cnd = x
E_icode = 5, E_ifun = 5, E_valA = 8, E_valB = 5, E_valC = 10, e_valE = 12, e_cnd = x
E_icode = 5, E_ifun = 5, E_valA = 8, E_valB = 5, E_valC = 10, e_valE = 15, e_cnd = x
E_icode = 6, E_ifun = 0, E_valA = 6, E_valB = 2, E_valC = 1, e_valE = 15, e_cnd = x
E_icode = 6, E_ifun = 0, E_valA = 6, E_valB = 2, E_valC = 1, e_valE = 8, e_cnd = x
E_icode = 6, E_ifun = 0, E_valA = 9223372036854775807, E_valB = 9223372036854775807, E_valC = 1, e_valE = 8, e_cnd = x
E_icode = 6, E_ifun = 0, E_valA = 9223372036854775807, E_valB = 9223372036854775807, E_valC = 1, e_valE = 9223372036854775806, e_cnd = x
E_icode = 7, E_ifun = 5, E_valA = 5, E_valB = 15, E_valC = 0, e_valE = 9223372036854775806, e_cnd = x
E_icode = 6, E_ifun = 2, E_valA = 6, E_valB = 2, E_valC = 1, e_valE = 9223372036854775806, e_cnd = x
E_icode = 6, E_ifun = 2, E_valA = 6, E_valB = 2, E_valC = 1, e_valE = 2, e_cnd = x
E_icode = 8, E_ifun = 3, E_valA = 6, E_valB = 10, E_valC = 15, e_valE = 2, e_cnd = x
E_icode = 9, E_ifun = 14, E_valA = 12, E_valB = 2, E_valC = 12, e_valE = 2, e_cnd = x
E_icode = 9, E_ifun = 14, E_valA = 12, E_valB = 2, E_valC = 12, e_valE = 10, e_cnd = x
E_icode = 10, E_ifun = 1, E_valA = 12, E_valB = 15, E_valC = 2, e_valE = 10, e_cnd = x
E_icode = 10, E_ifun = 1, E_valA = 12, E_valB = 15, E_valC = 2, e_valE = 7, e_cnd = x
E_icode = 11, E_ifun = 5, E_valA = 3, E_valB = 2, E_valC = 10, e_valE = 7, e_cnd = x
E_icode = 11, E_ifun = 5, E_valA = 3, E_valB = 2, E_valC = 10, e_valE = 10, e_cnd = x
```

As rmmovq and mrmovq instructions have E_icode 4 and 5 respectively, the role of the execute stage is to add E_valB and E_valC which is correctly getting added here. For E_icode 6 and E_ifun 0, additon takes place (6 +2 = 8), for E_ifun 1, subtraction takes place , for ifun 2 AND takes place (0110 & 0010 = 0010) and for E_ifun 3 XOR takes place which are taking place correctly here. Next, for E_icode 7, cnd is computed. Here, as E_ifun is 5 which corresponds for greater than or equal, as the condition is not satisfied, cnd is 0. For E_icode 9 and 11, the execute stage gives output E as valB + 8 which is correctly done here. Also, for E_icode 10, the execute stage gives output E as valB – 8, which takes takes place correctly here.

d. **Memory**

The below image shows the output of the pipelined execute stage for various M_icode and M_ifun values:

```
M_icode = x, M_valA = x, M_valE = x, m_valM = x
M_icode = 4, M_valA = 66, M_valE = 62, m_valM = x
M_icode = 5, M_valA = 1, M_valE = 62, m_valM = 66
M_icode = 6, M_valA = 1, M_valE = 2, m_valM = 66
M_icode = 10, M_valA = 33, M_valE = 2039, m_valM = 66
M_icode = 11, M_valA = 2039, M_valE = 2047, m_valM = 33
```

When icode is 4, rmmovq is performed. Hence, the value in M_valA, that is, 66 is stored in the memory. We can verify that it is stored correctly by carrying out mrmovq. When icode is 5, mrmovq is performed. Hence, the value stored in M_valE, that is, 66 becomes the value of m_valM. Also, when icode is 10, we perform push. Here, we are pushing the value 33. Now, when we perform pop we can see that we get the value of m_valM as 33 which is correct.

e. **Write-back**

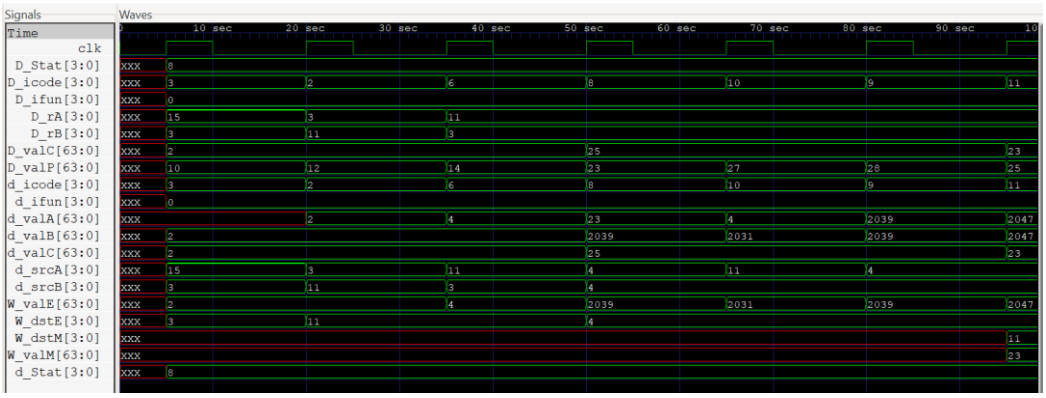
```
clk = 0;

#5 clk = !clk;
D_icode = 3; D_ifun = 0; D_Stat = 8;
D_rB = 3; D_valC = 2; D_rA = 15; D_valP = 10;
W_dstE = 3; W_valE = 2;

#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %0d, A = %
d_icode, d_ifun, d_valC, d_valA, d_valB, d_dstE,

#5 clk = !clk;
D_icode = 2; D_ifun = 0; D_Stat = 8;
D_rB = 11; D_valC = 2; D_rA = 3; D_valP = 12;
// e_dstE = 3; e_valE = 2;
W_dstE = 11; W_valE = 2;

#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %0d, A = %
d_icode, d_ifun, d_valC, d_valA, d_valB, d_dstE,

#5 clk = !clk;
D_icode = 6; D_ifun = 0; D_Stat = 8;
D_rB = 3; D_valC = 2; D_rA = 11; D_valP = 14;
// e_dstE = 3; e_valE = 2;
// M_dstE = 11; M_valE = 2;
W_dstE = 11; W_valE = 4;
```

```
#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %0d, A = %0d, B = %0d, dstE = %0d, dtsM = %0d, s
d_icode, d_ifun, d_valC, d_valA, d_valB, d_dstE, d_dstM, d_srcA, d_srcB, d_Stat);

#5 clk = !clk;
D_icode = 11; D_ifun = 0; D_Stat = 8;
D_valC = 23; D_valP = 25;
W_dstM = 11; W_valM = 23;
W_dstE = 4; W_valE = 2047;

#5 clk = !clk;
#5 $display("dic = %0d, dif = %0d, C = %0d, A = %0d, B = %0d, dstE = %0d, dtsM = %0d, s
d_icode, d_ifun, d_valC, d_valA, d_valB, d_dstE, d_dstM, d_srcA, d_srcB, d_Stat);
```

Testcase for WB stage based on the previous

```
dic = 3, dif = 0, C = 2, A = x, B = 2, dstE = 3, dtsM = x, srcA = 15, srcB = 3, d_stat = 8

dic = 2, dif = 0, C = 2, A = 2, B = 2, dstE = 11, dtsM = x, srcA = 3, srcB = 11, d_stat = 8

dic = 6, dif = 0, C = 2, A = 4, B = 2, dstE = 3, dtsM = x, srcA = 11, srcB = 3, d_stat = 8

dic = 8, dif = 0, C = 25, A = 23, B = 2039, dstE = 4, dtsM = x, srcA = 4, srcB = 4, d_stat = 8

dic = 10, dif = 0, C = 25, A = 4, B = 2031, dstE = 4, dtsM = x, srcA = 11, srcB = 4, d_stat = 8

dic = 9, dif = 0, C = 25, A = 2039, B = 2039, dstE = 4, dtsM = 11, srcA = 4, srcB = 4, d_stat = 8

dic = 11, dif = 0, C = 23, A = 2047, B = 2047, dstE = 4, dtsM = 11, srcA = 4, srcB = 4, d_stat = 8
```

Results for the above testcase



GTK plot for the above result

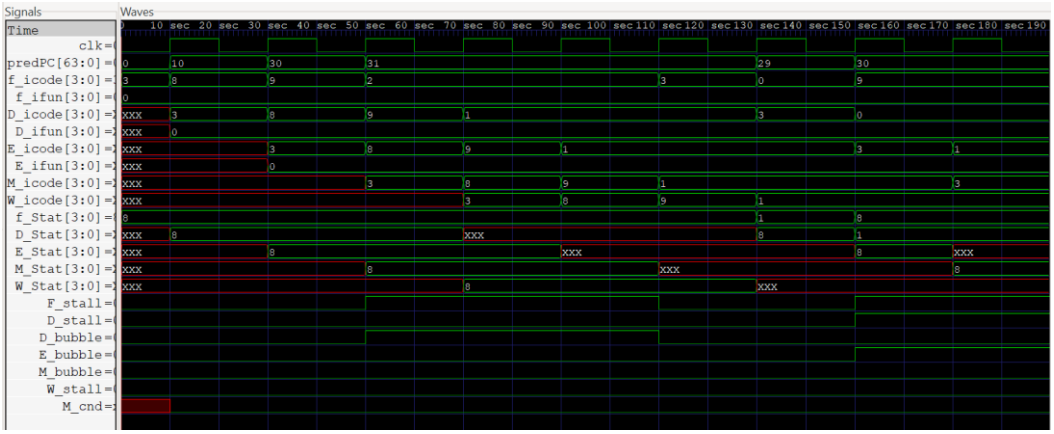
2. Test-cases for the wrapper code

1. **ret** hazard testing

```
0x000:     irmovq stack,%rsp  #   Initialize stack pointer
0x00a:     call proc          #   Procedure call
0x013:     irmovq $10,%rdx    #   Return point
0x01d:     halt
0x020: .pos 0x20
0x020: proc:                  # proc:
0x020:     ret                #   Return immediately
0x021:     rrmovq %rdx,%rbx   #   Not executed
0x030: .pos 0x30
0x030: stack:                 # stack: Stack pointer
```

```
// ROM[0] = 8'b00110000;  // ROM[19] = 8'b00110000;
// ROM[1] = 8'b11110100;  // ROM[20] = 8'b11110010;
// ROM[2] = 8'b11111111;  // ROM[21] = 8'b00001010;
// ROM[3] = 8'b00000111;  // ROM[22] = 8'b00000000;
// ROM[4] = 8'b00000000;  // ROM[23] = 8'b00000000;
// ROM[5] = 8'b00000000;  // ROM[24] = 8'b00000000;
// ROM[6] = 8'b00000000;  // ROM[25] = 8'b00000000;
// ROM[7] = 8'b00000000;  // ROM[26] = 8'b00000000;
// ROM[8] = 8'b00000000;  // ROM[27] = 8'b00000000;
// ROM[9] = 8'b00000000;  // ROM[28] = 8'b00000000;
// ROM[10] = 8'b10000000; // ROM[29] = 8'b00000000;
// ROM[11] = 8'b00011110; // ROM[30] = 8'b10010000;
// ROM[12] = 8'b00000000; // ROM[31] = 8'b00100000;
// ROM[13] = 8'b00000000; // ROM[32] = 8'b00100011;
// ROM[14] = 8'b00000000;
// ROM[15] = 8'b00000000;
// ROM[16] = 8'b00000000;
// ROM[17] = 8'b00000000;
// ROM[18] = 8'b00000000;
```

*Testcase and its binary version for **ret** hazard*



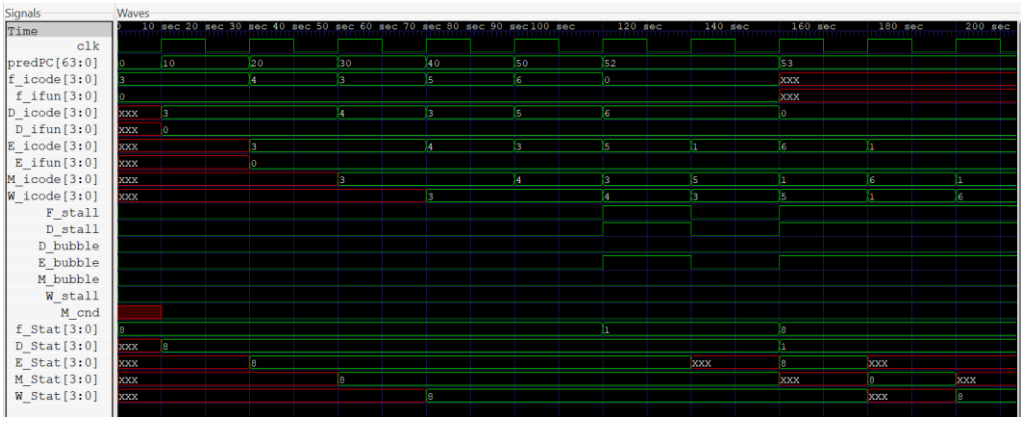
*GTK plot for the above testcase – clearly the F_stall and D_bubble get activated at correct instants and the **ret** hazard is resolved*

2. **load/use** hazard testing

```
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %ebx,%eax # Use %rax
0x034: halt
```

```
// ROM[0] = 8'b00110000;   // ROM[27] = 8'b00000000;
// ROM[1] = 8'b11110010;   // ROM[28] = 8'b00000000;
// ROM[2] = 8'b10000000;   // ROM[29] = 8'b00000000;
// ROM[3] = 8'b00000000;   // ROM[30] = 8'b00110000;
// ROM[4] = 8'b00000000;   // ROM[31] = 8'b11110011;
// ROM[5] = 8'b00000000;   // ROM[32] = 8'b00001010;
// ROM[6] = 8'b00000000;   // ROM[33] = 8'b00000000;
// ROM[7] = 8'b00000000;   // ROM[34] = 8'b00000000;
// ROM[8] = 8'b00000000;   // ROM[35] = 8'b00000000;
// ROM[9] = 8'b00000000;   // ROM[36] = 8'b00000000;
// ROM[10] = 8'b00110000;  // ROM[37] = 8'b00000000;
// ROM[11] = 8'b11110001;  // ROM[38] = 8'b00000000;
// ROM[12] = 8'b00000011;  // ROM[39] = 8'b00000000;
// ROM[13] = 8'b00000000;  // ROM[40] = 8'b01010000;
// ROM[14] = 8'b00000000;  // ROM[41] = 8'b00100000;
// ROM[15] = 8'b00000000;  // ROM[42] = 8'b00000000;
// ROM[16] = 8'b00000000;  // ROM[43] = 8'b00000000;
// ROM[17] = 8'b00000000;  // ROM[44] = 8'b00000000;
// ROM[18] = 8'b00000000;  // ROM[45] = 8'b00000000;
// ROM[19] = 8'b00000000;  // ROM[46] = 8'b00000000;
// ROM[20] = 8'b01000000;  // ROM[47] = 8'b00000000;
// ROM[21] = 8'b00010010;  // ROM[48] = 8'b00000000;
// ROM[22] = 8'b00000000;  // ROM[49] = 8'b00000000;
// ROM[23] = 8'b00000000;  // ROM[50] = 8'b01100000;
// ROM[24] = 8'b00000000;  // ROM[51] = 8'b00110000;
// ROM[25] = 8'b00000000;  // ROM[52] = 8'b00000000;
// ROM[26] = 8'b00000000;
```

*Testcase and its binary version for **load/use** hazard*



GTK plot for the above testcase | clearly the stall and bubble operations are happening at correct instants

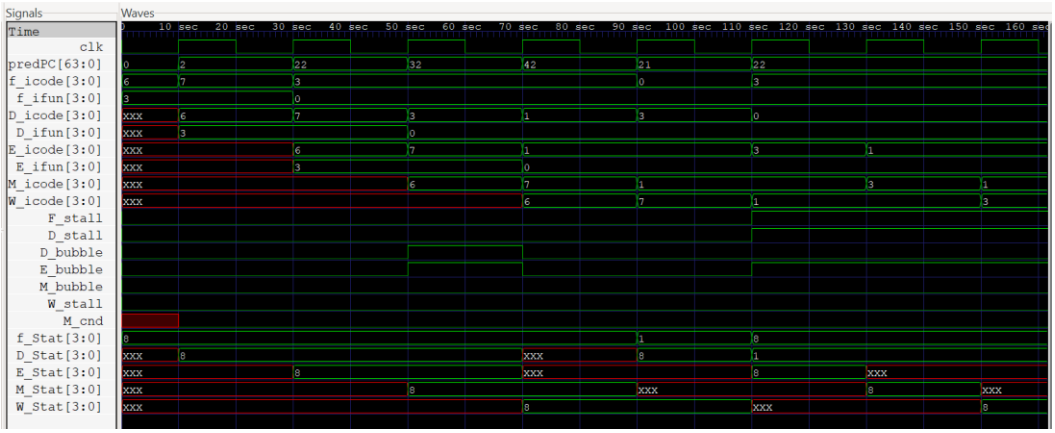
3. **mispredicted jmp** hazard testing

```
0x000:     xorq %rax,%rax
0x002:     jne  target         # Not taken
0x00b:     irmovq $1, %rax     # Fall through
0x015:     halt
0x016: target:
0x016:     irmovq $2, %rdx     # Target
0x020:     irmovq $3, %rbx     # Target+1
0x02a:     halt
```

```
// ROM[0] = 8'b01100011;
// ROM[1] = 8'b00000000;
// ROM[2] = 8'b01110011;
// ROM[3] = 8'b00010110;
// ROM[4] = 8'b00000000;
// ROM[5] = 8'b00000000;
// ROM[6] = 8'b00000000;
// ROM[7] = 8'b00000000;
// ROM[8] = 8'b00000000;
// ROM[9] = 8'b00000000;
// ROM[10] = 8'b00000000;
// ROM[11] = 8'b00110000;
// ROM[12] = 8'b11110000;
// ROM[13] = 8'b00000001;
// ROM[14] = 8'b00000000;
// ROM[15] = 8'b00000000;
// ROM[16] = 8'b00000000;
// ROM[17] = 8'b00000000;
// ROM[18] = 8'b00000000;
// ROM[19] = 8'b00000000;
// ROM[20] = 8'b00000000;
// ROM[21] = 8'b00000000;
// ROM[22] = 8'b00110000;
```

```
// ROM[23] = 8'b11110010;
// ROM[24] = 8'b00000010;
// ROM[25] = 8'b00000000;
// ROM[26] = 8'b00000000;
// ROM[27] = 8'b00000000;
// ROM[28] = 8'b00000000;
// ROM[29] = 8'b00000000;
// ROM[30] = 8'b00000000;
// ROM[31] = 8'b00000000;
// ROM[32] = 8'b00110000;
// ROM[33] = 8'b11110011;
// ROM[34] = 8'b00000011;
// ROM[35] = 8'b00000000;
// ROM[36] = 8'b00000000;
// ROM[37] = 8'b00000000;
// ROM[38] = 8'b00000000;
// ROM[39] = 8'b00000000;
// ROM[40] = 8'b00000000;
// ROM[41] = 8'b00000000;
// ROM[42] = 8'b00000000;
```

*Testcase and binary versions for **mispricticted jmp***

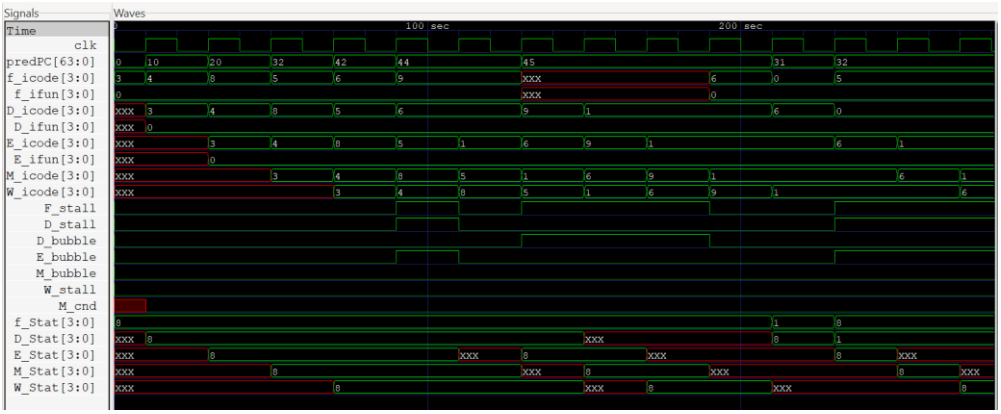


GTK plot for the above testcase | clearly the bubble operations to flush out wrong instructions are triggered at the correct instances

4. **Load/Use and ret combo**

```
irmovq $1, %rax
rmmovq %rax, 0(%rdx)
call combo
addq %rcx, %rax
halt
combo:
    mrmovq 0(%rdx), %rcx
    addq %rcx, %rax
    ret
```

Self-designed testcase for mentioned combo of hazard

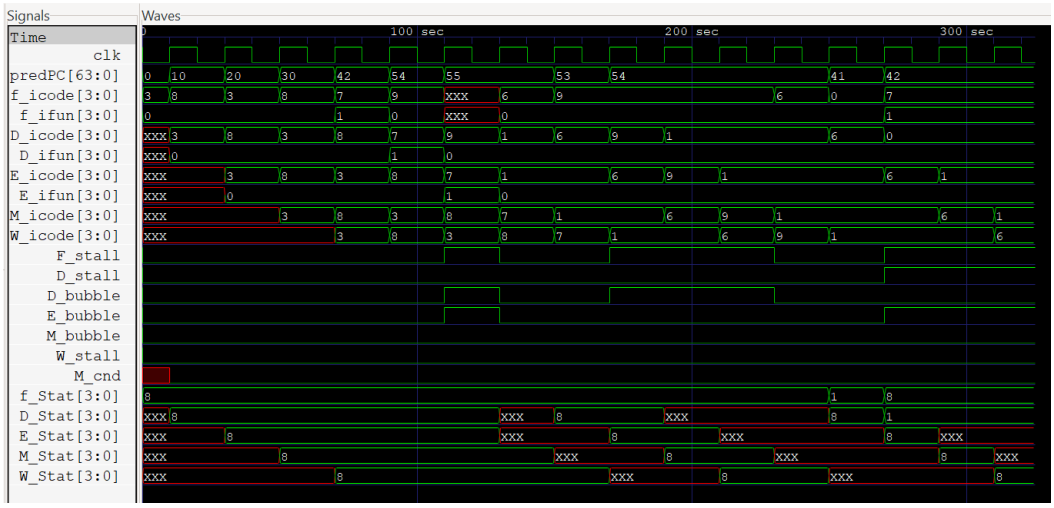


GTK plot for the above testcase | evidently stall and bubble are triggered appropriately

5. **Mispredicted jmp and ret combo**

```
irmovq $2, %rcx
call main
halt
main:
    irmovq $1, %rax
    call combo
    addq %rcx, %rax
    halt
    combo:
        jle miss
        addq %rax, %rax
        ret
    miss:
        ret
```

Self-designed testcase for given combo



GTK plot for the given testcase | bubble and stall applied correctly

6. **Fibonacci Series (self-made complex testcase)**

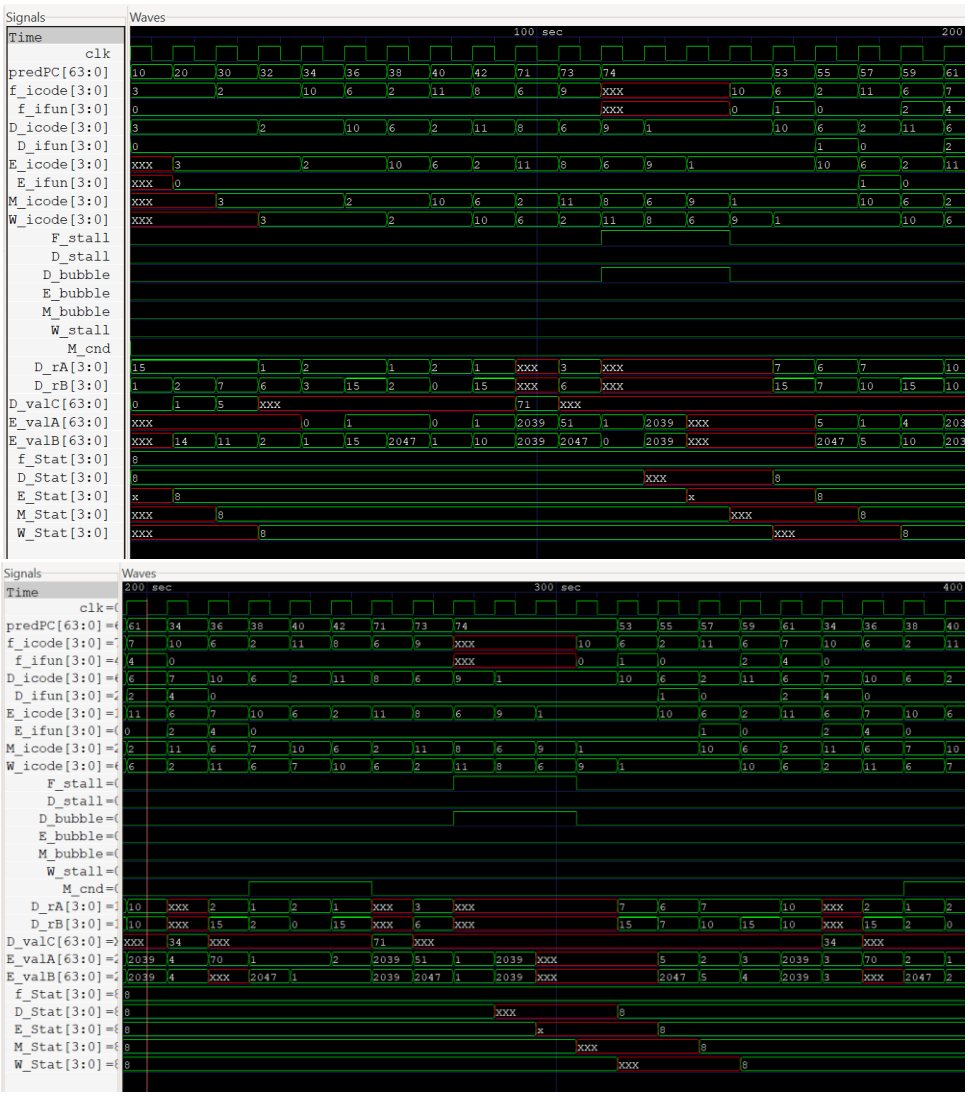
P.T.O

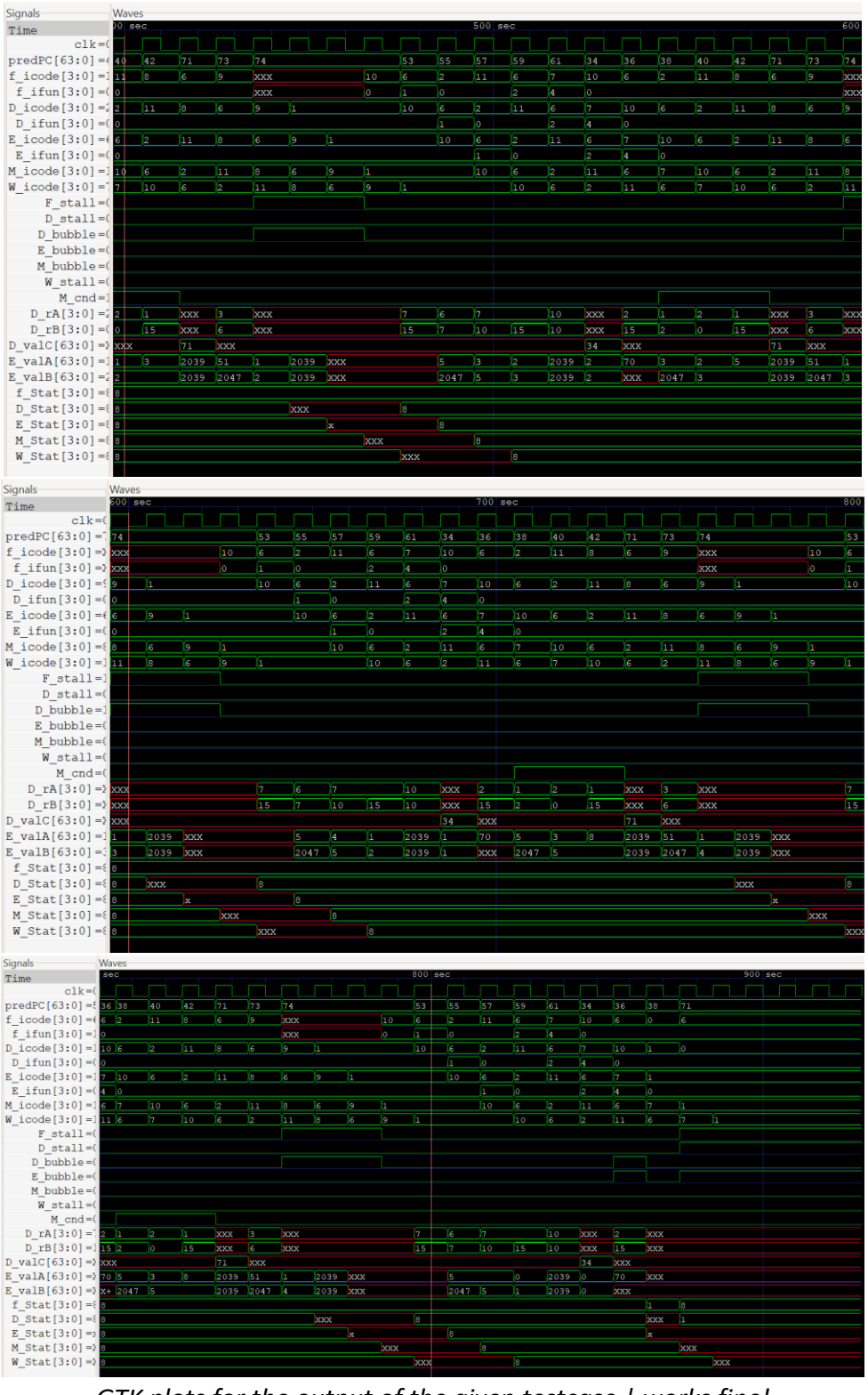
```
00110000 // irmovq
11110001 // F, %rcx
00000000 // $0
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00110000 // irmovq
11110010 // F, %rdx
00000001 // &1
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00110000 // irmovq
11110111 // F, %rdi
00000101 // $5
00000000
00000000
00000000
00000000
00000000
00000000
00000000
```

```
00100000 // rrmovq
00010110 // %rcx, %rsi
00100000 // rrmovq
00100011 // %rdx, %rbx
10100000 // pushq → fib_loop
00101111 // %rdx
01100000 // addq
00010010 // %rcx, %rdx
00100000 // rrmovq
00100000 // %rdx, %rax
10110000 // popq
00011111 // %rbx
10000000 // call inc
01000111 // $71
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
10100000 // pushq
01111111 // %rdi
01100001 // subq
01100111 // %rsi, %rdi
00100000 // rrmovq
01111010 // %rdi, %r10
10110000 // popq
01111111 // %rdi
01100010 // andq
10101010 // %r10, %r10
```

```
01110100 // jne fib_loop
00100010 // $ 34
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000 // halt
01100000 // addq → inc
00110110 // %rbx, %rsi
10010000 // ret
```

Testcase for Fibonacci series calculation





GTK plots for the output of the given testcase | works fine!

Section 3 – Conclusion

Challenges Faced

1. The register file is present in the decode stage but it needs to be accessed by the write back stage as well. For this we need to pass the register file to the write back stage everytime, but this might be a problem. So, to resolve this we have written the decode and execute stage in the same module so that both can access the register file.
2. For certain modules, writing the testbench of individual modules and checking whether it is working or not is tougher than testing the combination of stages.
3. Combing all the stages of the processor and ensuring that all the stages work together properly is a tough task.
4. Ensuring that there is a definite value in all the registers at all time is a challenge. If a register has don't care, then it might take up a random value which could cause an issue later.
5. In the pipelined implementation of Y86, tackling the different types of hazards and coming up with the pipeline control logic was difficult

Combination of multiple types of hazards causes a lot of issues. Hence, resolving it is a very tough task.

Conclusion

Through this project we realised what all goes behind making even a simple version of what is otherwise a cutting-edge technology in today's world! The concepts learnt along the way have definitely helped us get a sneak-peak into the world of computer organisation and architecture; this is no less than a design marvel.

We express our immense gratitude to Professor Deepak Gangadharan and the TAs who were involved in the smooth conduct of the course and for extending constant help as and when it was required by the students.