IPA PROJECT REPORT

A Sequential and Pipelined implementation of y86-64 processor in Verilog

Tejas Srivastava
2021102017

Jalluri Ram Gopal
2021102013

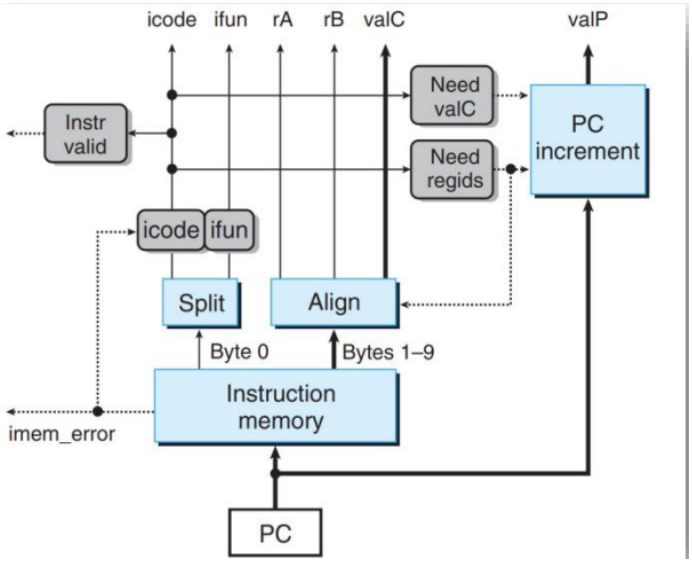
PROJECT DESCRIPTION:

The project involves the implementation of Y86-64 processor architecture design using Verilog. The final goal is to achieve a 5 state pipelined implementation of the processor. This report includes the sequential and the pipelined implementation of the Y86-64 processor which contains the fetch, decode - write back , execute, memory and the PC update blocks, their testbenches and the combined testbench along with data forwarding and support for eliminating pipeline hazards.

SEQUENTIAL IMPLEMENTATION:

The sequential implementation of the Y86-64 processor works with fetch, decode, execute, memory, writeback and PC update. In this implementation, only one instruction will be there in whole architecture in one clock cycle. Each block of sequential implementation look like as follows:

FETCH BLOCK:

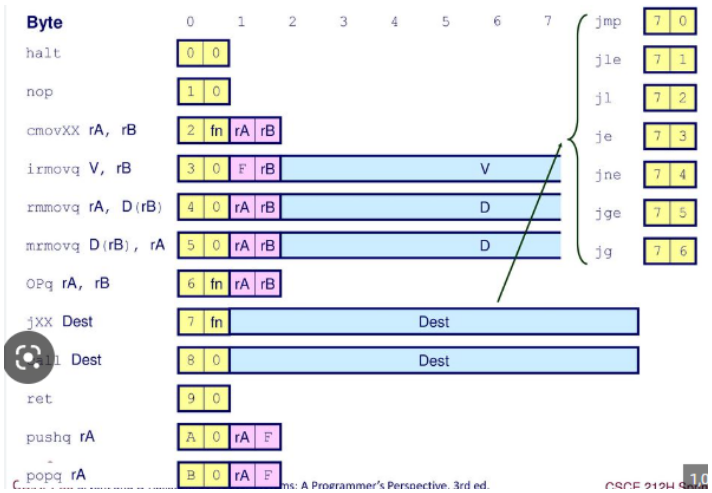


⇒ Here, PC value is extracted and read by instruction memory. Instruction will be divided into two parts and they are split and align.

⇒ Split is of one byte and contains icode and ifun. icode must be always between 0 and 11. Any other number will activate Instr_valid flag as 0. Also if PC value is invalid, imem_error will be 1.

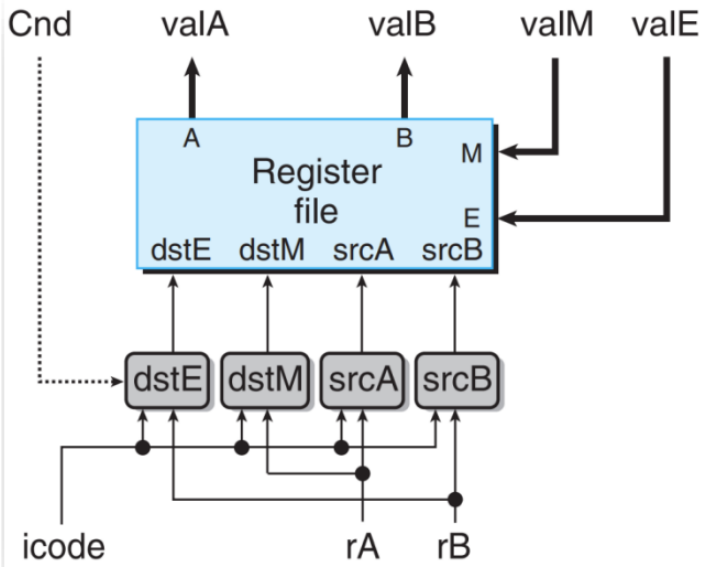
⇒ The purpose of fetch block is to take PC as an input and compute the icode, ifun, rA, rB, valC, valP, Instr valid and the imem_error as the outputs.

⇒ First byte of the instruction represents icode and ifun while rest nine bytes of instruction represent values of registers and rest the immediate value of destination offset depending on icode and ifun according to the following




```
module fetch(clk, PC, instr, icode, ifun, rA, rB, valC, valP, mem_error, instr_invalid);

input clk;
input [63:0] PC;
input [0:79] instr;

output reg[3:0] icode, ifun;
output reg [3:0] rA, rB;
output reg signed[63:0] valC;
output reg [63:0] valP;
```

DECODE BLOCK:



Decode and write-back are implemented in same module because only these two require register block and hardware implementation look like as decode and write-back are together.

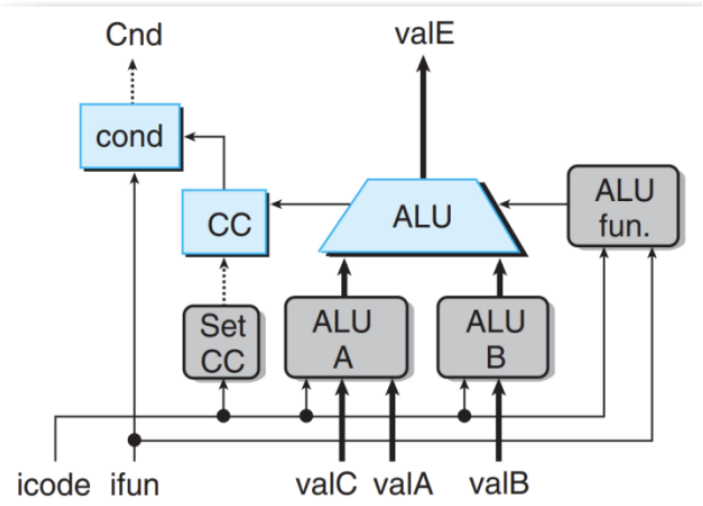
This register file contains 15 registers each of size 64-bit which contains addresses, numbers that are need to be used in further program because values are first extracted from memory to register and then any computations are done.

⇒ Here, icode, rA, rB, valE, valM, cnd are inputs to this module and valA, valB are outputs.

⇒ valE, valM are used to write-back the registers.

```
module decode (
    clk,
    icode,ifun,
    rA,rB,
    valA, valB, valM, valE, cnd,
    R0, R1, R2, R3, R4, R5,  R6, R7, R8, R9, R10, R11, R12, R13, R14
);

input clk;
input [3:0] rA, rB, icode, ifun; // half byte values
input signed[63:0] valE; // val from Execution stage
input [63:0] valM;  // val from Memory stage
input cnd; // condition flag for conditional instructions
output reg signed[63:0] valA, valB;
output reg signed[63:0] R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14;

reg signed[63:0] reg_temp[0:14]; // 15 registers of 64 bit each
```

EXECUTE BLOCK:



Execute consists of main computation block where valA, valB, valC are provided as inputs and required computation is done based on icode, ifun values. Conditional flags will be updated in this module and those are used for cnd computation that is required for conditional move and jump instructions.

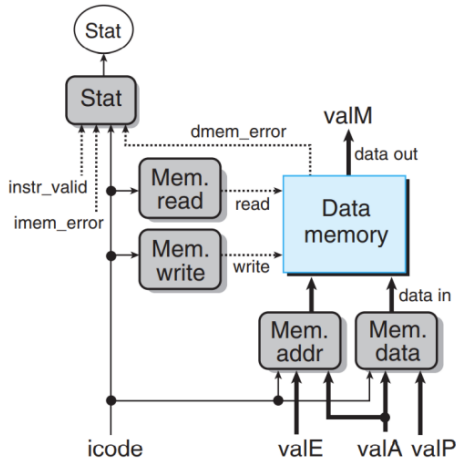
⇒ The execute part works on switch statements which selects the instructions according to the values of icode and ifun.

⇒ The condition codes are also set for the instructions jXX and CMOV and valE is computed using ALU.

⇒ The presence of CC_in and CC_out is due to the inability of Verilog to change the input value so the value of CC_in is declared to that of CC_out for the arithmetic operations i.e. when the condition codes are generated.

```
module execute(clk, icode, ifun, valA, valB, valC, cc_in, valE, cnd, cc_out);

input clk;
input [3:0] icode, ifun;
input signed [63:0] valA, valB, valC;
input [2:0]cc_in;

output reg [63:0] valE;
output reg cnd;
output reg [2:0] cc_out;

wire [63:0] valE_BC, valE_OP, valE_IN, valE_DE;
wire t_OF1, t_OF2, t_OF3;
wire OF;

ALU E1(valB, valC, 2'b00, valE_BC, t_OF1);
ALU E2(valA, valB, ifun[1:0], valE_OP, OF);
ALU E3(valB, 64'd1, 2'b00, valE_IN, t_OF2);
ALU E4(valB, 64'd1, 2'b01, valE_DE, t_OF3);
```

MEMORY BLOCK:



The purpose of memory block is to read and write the values from the memory. Based on icode value, memory stage will be in read or write state.

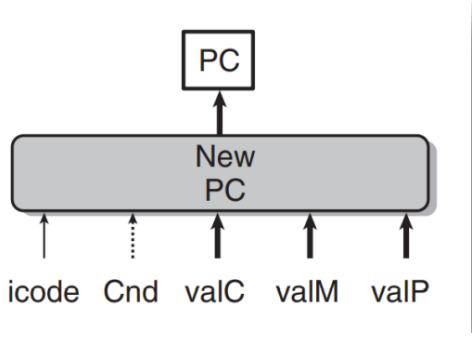
Read state for instructions mrmovq, return, popq and write state for instructions rmmovq, call, pushq.

⇒ The memory block has input values as icode, valE, valA and valP and the output values as valM.

⇒ There is something called mainmem_error that will be activated to 1 if valW or valA is greater than memory block size.

```
module memory(clk, icode, valA, valE, valM, valP, mainmem_error);
    input clk;
    input [3:0] icode;
    input [63:0] valA, valE, valP;
    output reg [63:0] valM;
    output reg mainmem_error;

    reg [63:0] mainmem[255:0];
```

PC UPDATE BLOCK:



In this block, PC decision takes place.

Inputs are icode, cnd, valC, valM, valP and output is PC only which will be used for next fetch.

For all instructions except jump(PC⇒ (cnd==1) ? valC:valP), call(PC⇒ valC), return(PC⇒valM).

```
module pc_update(clk, icode, cnd, PC_new, valC, valM, valP);

input [3:0] icode;
input cnd,clk;
input [63:0] valC, valM, valP;
output reg [63:0] PC_new;
```

TESTBENCH RESULTS:

instr_memory[1] = 8'h10; //nop

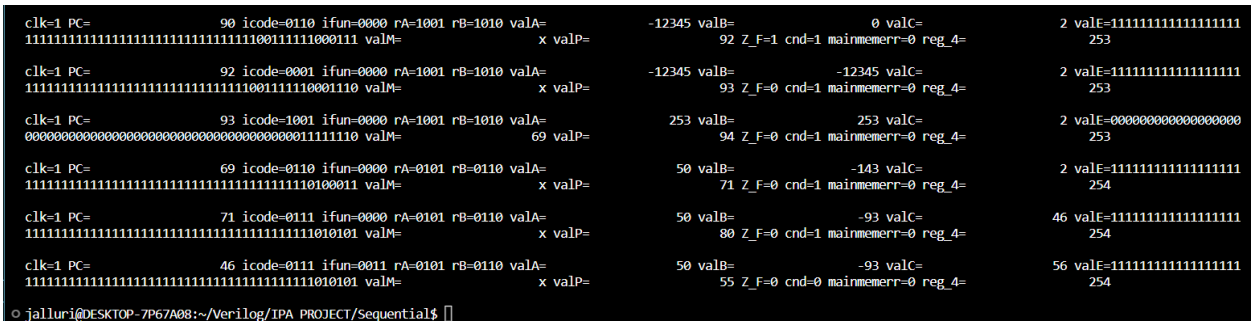
instr_memory[2] = 8'h20; //rrmovq
instr_memory[3] = 8'h12;

instr_memory[4] = 8'h30;//irmovq
instr_memory[5] = 8'hF2;
instr_memory[6] = 8'h00;
instr_memory[7] = 8'h00;
instr_memory[8] = 8'h00;
instr_memory[9] = 8'h00;
instr_memory[10] = 8'h00;
instr_memory[11] = 8'h00;
instr_memory[12] = 8'h00;
instr_memory[13] = 8'b00000010;

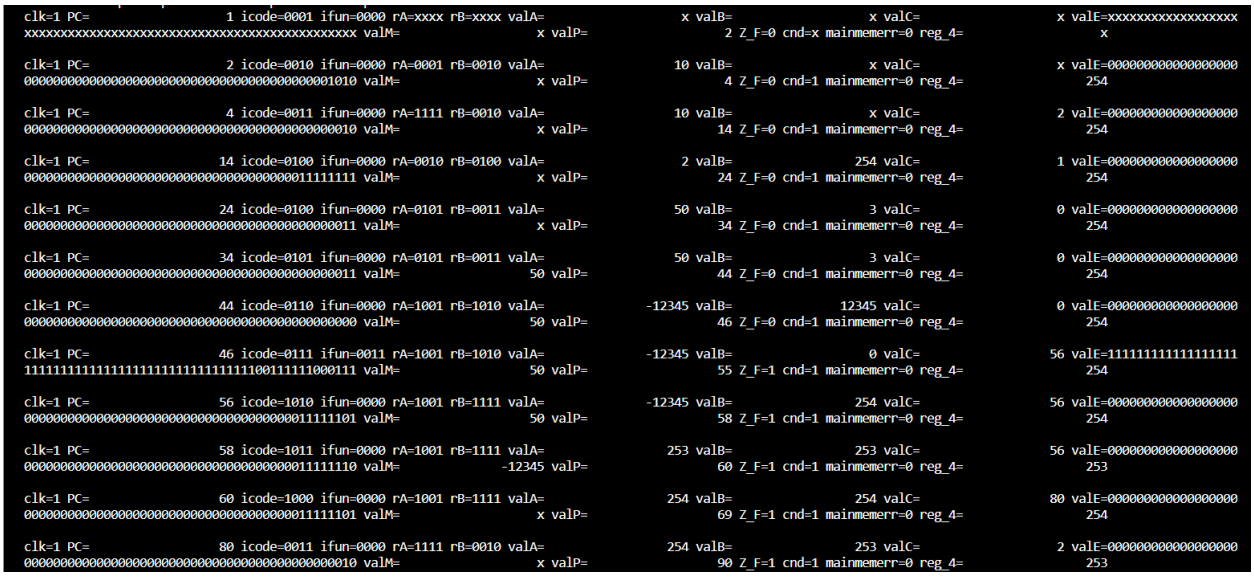
```
instr_memory[14] = 8'h40;//rmmovq
instr_memory[15] = 8'h24;
{instr_memory[16],instr_memory[17],instr_memory[18],instr_memory[19],instr_memory[20],instr_memory[21],instr_memory[22],instr_
 = 64'd1;

instr_memory[24] = 8'h40;//rmmovq
instr_memory[25] = 8'h53;
{instr_memory[26],instr_memory[27],instr_memory[28],instr_memory[29],instr_memory[30],instr_memory[31],instr_memory[32],instr_
 = 64'd0;

instr_memory[34] = 8'h50;//mrmovq
instr_memory[35] = 8'h53;
{instr_memory[36],instr_memory[37],instr_memory[38],instr_memory[39],instr_memory[40],instr_memory[41],instr_memory[42],instr_
 = 64'd0;

instr_memory[44] = 8'h60;
instr_memory[45] = 8'h9A;

instr_memory[46] = 8'h73;
{instr_memory[47],instr_memory[48],instr_memory[49],instr_memory[50],instr_memory[51],instr_memory[52],instr_memory[53],instr_
 = 64'd56;

instr_memory[55] = 8'h00;

instr_memory[56] = 8'hA0;
instr_memory[57] = 8'h9F;

instr_memory[58] = 8'hB0;
instr_memory[59] = 8'h9F;

instr_memory[60] = 8'h80;
{instr_memory[61],instr_memory[62],instr_memory[63],instr_memory[64],instr_memory[65],instr_memory[66],instr_memory[67],instr_
 = 64'd80;

instr_memory[69] = 8'h60;
instr_memory[70] = 8'h56;

// instr_memory[71] = 8'h00;
instr_memory[71] = 8'h70;
{instr_memory[72],instr_memory[73],instr_memory[74],instr_memory[75],instr_memory[76],instr_memory[77],instr_memory[78],instr_
 = 64'd46;

// instr_memory[80] = 8'h63;
// instr_memory[81] = 8'hDE;
// instr_memory[80] = 8'h10;

instr_memory[80] = 8'h30;//irmovq
instr_memory[81] = 8'hF2;
instr_memory[82] = 8'h00;
instr_memory[83] = 8'h00;
instr_memory[84] = 8'h00;
instr_memory[85] = 8'h00;
instr_memory[86] = 8'h00;
instr_memory[87] = 8'h00;
```

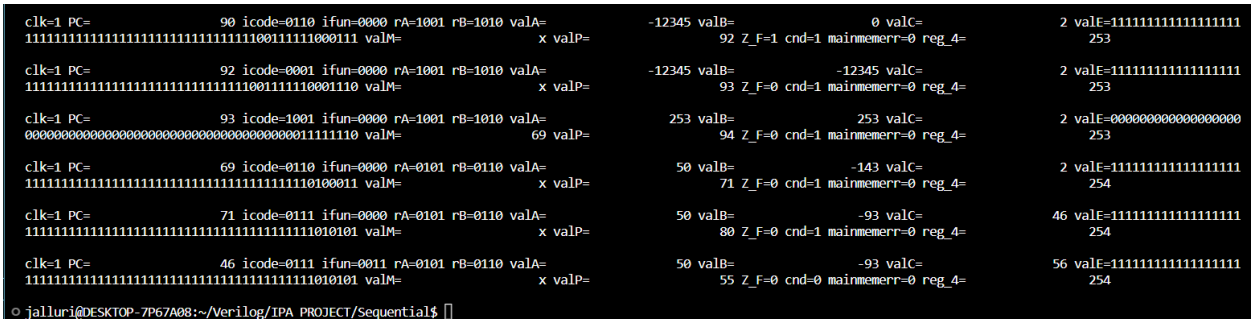
```
instr_memory[88] = 8'h00;
instr_memory[89] = 8'b00000010;

instr_memory[90] = 8'h60;
instr_memory[91] = 8'h9A;

instr_memory[92] = 8'h10;

instr_memory[93] = 8'h90;
```

REGISTER MEMORY VALUES ARE:

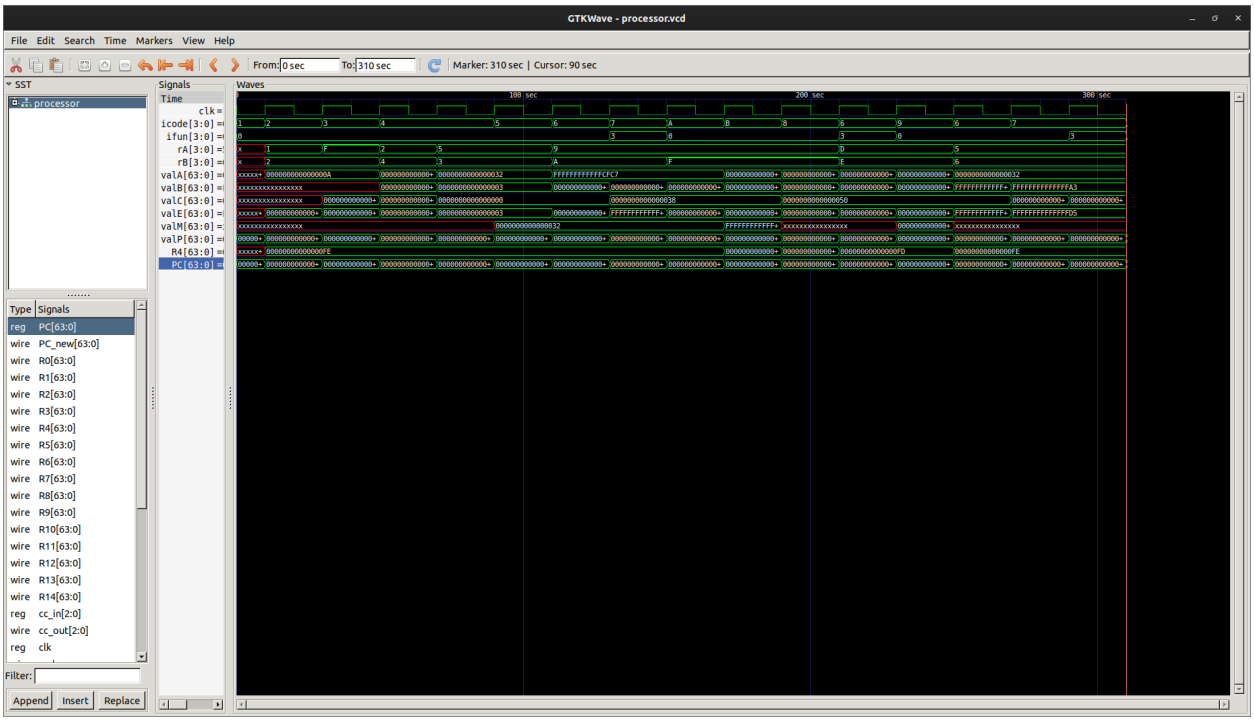


TERMINAL OUTPUTS LOOK LIKE AS





GTKWave for the output

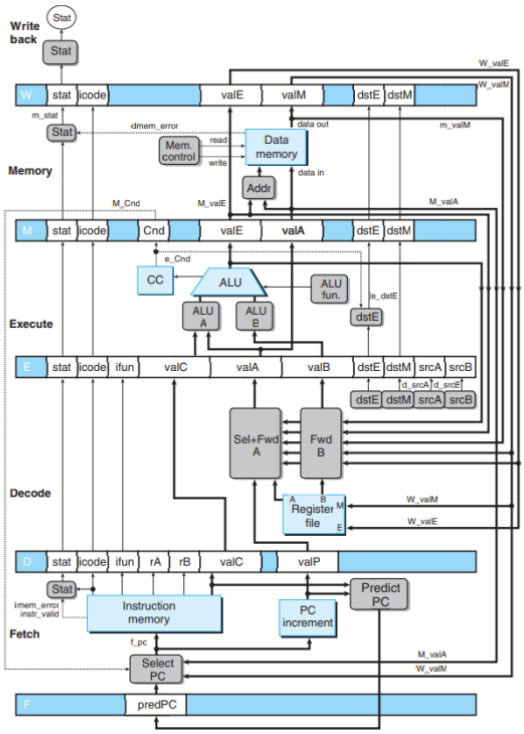


PIPELINED IMPLEMENTATION:

The pipelined implementation of the Y86-64 works the same way as that of the sequential implementation with the modules being same but with inclusion of the pipelined registers, slight change in the fetch and decode blocks, addition of support for data forwarding and PC prediction for improving the performance and the addition of the pipeline control logic for eliminating pipeline hazards. This type implementation increases throughput but increases latency which is a trade-off.

We implemented the processor with a clock time period of 20ns. We can calculate the throughput of the processor as the number of instructions completed per second.

$$throughput \ = \ \frac{1}{T_{cycle}} = \frac{1}{20ns} = 5 \cdot 10^7 ips \ = \ 50 \ Mega \ ips$$



FETCH BLOCK:

In the fetch module, the input ports are the flags for stalling/bubbling (in the form of F_stall, D_stall and D_bubble) and a few pipe registers from Memory and WriteBack stages to account for the misprediction of jXX instruction.

The outputs are the pipeline registers for the decode stage. Note that here the pipeline registers for fetch stage are declared and used locally, since they come directly from the instruction memory.

After splitting the instruction into icode and ifun, the rA, rB, valP, valC are extracted from the instruction and PC value. Moreover, the output predicted PC (f_predicted PC) is also assigned appropriate value.

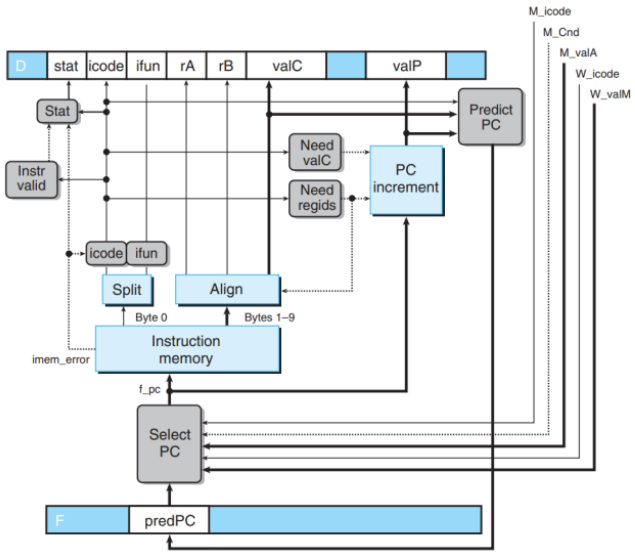
After handling the main function of fetch i.e. extracting values from the instruction to icode, ifun, rA, rB, valC, and valP, we start to check for error flags, i.e.

- **memory error :** when the value of PC exceeds the maximum memory available (or when invalid memory is accessed)
- **instruction error :** activated when invalid instruction is given according to the ISA

Moreover, values from writeback and memory stages are also forwarded for cases of ret statement and jump mis-prediction. If no such case is found, the PC is updated to the next instruction address.

When the positive edge of the clock comes, the values of icode, ifun rA, rB, valP, valC, and stata are given to decode as decode pipe register values. If a bubble/stall is needed to be introduced to the decode stage, we pass these values corresponding to the nop instruction. Moreover, the stat flags are also calculated accordingly and later passed to the decode stage as it is.

Note : In out fetch.v, the instruction memory is also stored, since this is the only module that accesses the instruction memory directly.



Here, according to the above block diagram, the f_pc is initialized to the initial value of PC from the processor.v block. Fetch.v takes f_pc as input and computes values of stat, icode, ifun, rA, rB, valC, valP. These values are sent to DECODE register. Here the fetch part functions the same as that of that of sequential, but the addition of sequential block increases the throughput of the processor. The addition of the Predict PC block adds the functionality of the predicting PC which will be sent to the fetch register in the processor.v block.

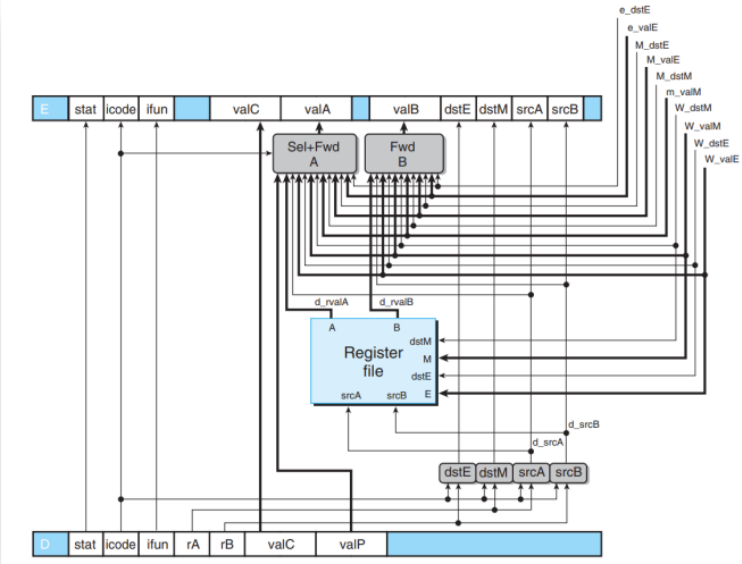
```
input clk;
input F_stall, D_stall, D_bubble, M_cnd;            // stall and bubble and cnd flags
input [3:0] M_icode, W_icode;                       // used for checking misprediction of branch
input [63:0] M_valA, W_valM, F_predictedPC;         // used for checking misprediction of branch and input predicted pc


output reg [63:0] f_predictedPC;                    // output predicted pc
output reg [3:0] D_icode, D_ifun, D_rA, D_rB;       // pipe registers in decode
output reg [63:0] D_valC, D_valP;                   // pipe registers in decode
output reg [0:3] D_stat;                            // status code for decode stage: AllOK, Halt, Adr_error, Instruction_error
```

Here, M_icode, M_valA, W_icode, W_valM are also inputs of this module because these are required for misprediction recovery.

The register gets updated once the clk hits and the output for D_icode, D_ifun, D_rA, D_rB, D_valC, D_stat and D_valP is available as an output of the register.

DECODE BLOCK:

In the decode block, both decode and write-back together are implemented.



Block Diagram for the Decode Stage

The main difference between sequential and pipelined is observed in this block because in this block only, data forwarding, stall and bubbles play main role in avaoiding data hazards.

This takes the inputs from the decode register which are D_icode, D_ifun, D_rA, D_rB, D_valC, D_stat, D_valP and also the inputs required for data forwarding which include e_dstE, e_valE, M_dstE, M_valE, M_dstM, m_valM, W_dstM, W_valM, W_dstE and W_valE.

In decode stage, values from execute, memory and writeback stages involve and depend on srcA, srcB, valA, valB are taken.

We now explain data forwarding, decode, and writeback stages implemented in this modeule in detail.

1. **Register File :** The module contains the 15 registers in the processor. Since the register file is accessed only by the decode and writeback stage, it is declared as an array of registers locally in the module. The values from these local registers are latched on to the register file variables at positive edge in the writeback.

```
// assigning random (no significance) values to the registers initially
initial
begin
    register[0] = 64'd12;        //rax
    register[1] = 64'd10;        //rcx
    register[2] = 64'd101;       //rdx
    register[3] = 64'd3;       //rbx
    register[4] = 64'd254;       //rsp
    register[5] = 64'd50;        //rbp
    register[6] = -64'd143;      //rsi
    register[7] = 64'd10000;     //rdi
    register[8] = 64'd990000;    //r8
    register[9] = -64'd12345;    //r9
    register[10] = 64'd12345;    //r10
    register[11] = 64'd10112;    //r11
    register[12] = 64'd0;        //r12
    register[13] = 64'd1567;     //r13
    register[14] = 64'd8643;     //r14
end
```

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

1. **Data Forwarding :** The decode stage requires to find out the values stored in register rA and rB (conditionally), but these values sometimes need to be forwarded to it by memory, execute and writeback stages depending upon the type of data dependency. The logic for data is given by this pseudo code.

```
## What should be the A value?
int d_valA = [
  # Use incremented PC
  D_icode in { ICALL, IJXX } : D_valP;
  # Forward valE from execute
  d_srcA == e_dstE : e_valE;
  # Forward valM from memory
  d_srcA == M_dstM : m_valM;
  # Forward valE from memory
  d_srcA == M_dstE : M_valE;
  # Forward valM from write back  d_srcA == W_dstM : W_valM;
  # Forward valE from write back
  d_srcA == W_dstE : W_valE;
  # Use value read from register file
  1 : d_rvalA;
];
// SIMILARLY FOR valB
```

2. **Decode :** We decide the values of srcA, srcB for decode based on the icode, and decide the values of dstE, dstM for writeback. The register valA and register valB (rvalA, rvalB) are assigned values from the register file, which is later assigned to the valA and valB if no case of forwarding is found.
3. **WriteBack :** The values {W_valE and W_valM} according to the registers dstE, and dstM are assigned to register array based on the icode. Later, at the positive edge of the clock, all the values in the local register array are assigned to corresponding register file (R0 to R14).
4. **Bubble and Stall :** The bubble flag for execute stage is checked here and if found true, icode, ifun, srcA, etc. according to a nop instruction is passed to the execute stage pipeline registers.

Code snippets for the corresponding sections are shown as below :-

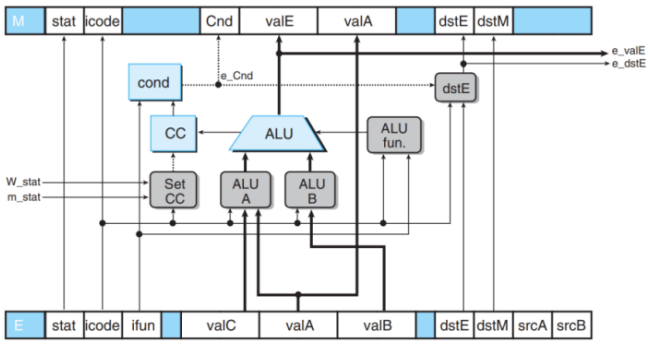
```
input clk;
input [3:0] D_icode, D_ifun, D_rA, D_rB;
input [63:0] D_valC, D_valP;                            // pipe registers for decode stage (input)
output reg[63:0] R0,R1,R2,R3,R4,R5,R6,R7,                   // Register file
                 R8,R9,R10,R11,R12,R13,R14;
input [0:3] D_stat;                                         // status code for decode
output reg[3:0] d_srcA, d_srcB;                             // srcA, and srcB for decode stage (to be calculated here)

output reg [3:0] E_icode, E_ifun;                           // pipe registers for execute stage (output)
output reg [63:0] E_valA, E_valB, E_valC;
output reg [3:0] E_srcA, E_srcB, E_dstE, E_dstM;
output reg [0:3] E_stat;
input E_bubble;                                             // bubble flag for execute stage

input [3:0] e_dstE, M_dstE, M_dstM, W_dstE, W_dstM, W_icode;    // registers used for data forwarding in different cases
input [63:0] e_valE, M_valE, m_valM, W_valE, W_valM;            // values used in data forwarding in different cases
```

Data Forwardings:

```
// implementing data forwarding
always @(*)
begin

    // forwarding data for valA
    if (D_icode==4'h8 | D_icode==4'h7)         // use the incremented PC (for hump and call)
    begin
        d_valA = D_valP;
    end
    else if (d_srcA == e_dstE & e_dstE!=4'hF)   // forward valE from execute
    begin
        d_valA = e_valE;
    end
    else if (d_srcA == M_dstM & M_dstM!=4'hF)   // forward valM from memory
    begin
        d_valA = m_valM;
    end
    else if (d_srcA == M_dstE & M_dstE!=4'hF)   // forward valE from memory
    begin
        d_valA = M_valE;
    end
    else if (d_srcA == W_dstM & W_dstM!=4'hF)   // forward valM from writeback
    begin
        d_valA = W_valM;
    end
    else if (d_srcA == W_dstE & W_dstE!=4'hF)   // forward valE from writeback
    begin
        d_valA = W_valE;
    end
    else                                        // use value read from register
    begin
        d_valA = d_rvalA;
    end


    // forwarding data for valB
    /*if (D_icode==4'h9 | D_icode==4'h7)  // use the incremented PC
    begin
        d_valB = D_valP;
    end*/
    if (d_srcB == e_dstE & e_dstE!=4'hF)          // forward valE from execute
    begin
        d_valB = e_valE;
    end
    else if (d_srcB == M_dstM & M_dstM!=4'hF)         // forward valM from memory
    begin
        d_valB = m_valM;
    end
    else if (d_srcB == M_dstE & M_dstE!=4'hF)         // forward valE from memory
    begin
        d_valB = M_valE;
    end
    else if (d_srcB == W_dstM & W_dstM!=4'hF)         // forward valM from writeback
    begin
        d_valB = W_valM;
    end
    else if (d_srcB == W_dstE & W_dstE!=4'hF)         // forward valE from writeback
    begin
        d_valB = W_valE;
    end
    else                                // use value read from register
    begin
        d_valB = d_rvalB;
    end
end
```

The register gets updated once the clk hits and the output for E_stat, E_ifun, E_icode, E_valA, E_valB and E_valC is available as an output of the execute register.

EXECUTE BLOCK:



This takes the inputs as the outputs from the execute pipelined register which include E_stat, E_ifun, E_icode, E_valA, E_valB, E_valC, E_dstE, E_dstM, W_stat and m_stat.

The outputs obtained from this block include M_stat, M_icode, Cnd, M_valE, M_valA, M_dstE, M_dstM, e_valE and e_dstE.

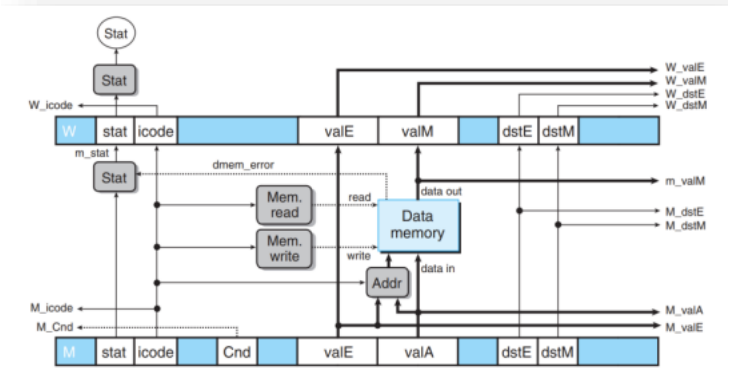
Here E_stat, E_ifun, E_icode, E_valA, E_valB, E_valC, E_dstE and E_dstM which are the inputs when passed through this block compute M_stat, M_icode, Cnd, M_valE, M_valA, M_dstE, M_dstM and e_valE as outputs in the similar way to that of sequential.

The value e_dstE is computed based on e_cnd which will make it either E_dstE or an empty register.

Once the positive edge is hit, new values will be updated in memory register.

```
module execute(clk, E_stat, E_icode, E_ifun, E_valC, E_valA, E_valB, E_dstE, E_dstM, W_stat, m_stat, set_cc,
               M_stat, M_icode, M_cnd, M_valE, M_valA, e_valE, M_dstE, M_dstM, e_dstE, e_cnd);

// upper parameters are inputs and lower ones are outputs

//input parameters

input clk;
input [0:3] E_stat;
input [3:0] E_icode, E_ifun, E_dstE, E_dstM;
input signed [63:0] E_valC, E_valA, E_valB;
input [3:0] W_stat;
input [3:0] m_stat;
input set_cc;

//output parameters

output reg [0:3] M_stat;
output reg [3:0] M_icode;
output reg signed [63:0] M_valE, M_valA, e_valE;
output reg [3:0] M_dstE, M_dstM, e_dstE;
output reg M_cnd;
output reg e_cnd;
```

MEMORY BLOCK:



This takes the inputs as the outputs from the memory pipelined register which include M_stat, M_icode, M_Cnd, M_valE, M_valA, M_dstE and M_dstM.

The outputs obtained from this block include W_stat, W_icode, W_valE, W_valM, W_dstE, W_dstM and m_valM.

```
module memory ( clk, M_stat, M_icode, M_cnd, M_valE, M_valA, M_dstE, M_dstM,
                W_stat, W_icode, W_valE, W_valM, W_dstE, W_dstM, m_valM, m_stat);

input clk;
input [0:3] M_stat;
input [3:0] M_icode;
input M_cnd;
input signed [63:0] M_valE;
input signed [63:0] M_valA;
input [3:0] M_dstE;
input [3:0] M_dstM;

output reg [0:3] W_stat;
output reg [3:0] W_icode;
output reg signed [63:0] W_valE, W_valM;
output reg [3:0] W_dstE, W_dstM;
output reg signed [63:0] m_valM;
output reg [3:0] m_stat;

reg [63:0] mainmem[16383:0];
reg mainmem_err = 0;
```

TESTCASE

```
instr_mem[0] = 8'h10;  //nop instruction

instr_mem[1]  = 8'h10; //nop instruction

instr_mem[2] = 8'h20; //rrmovq instruction
instr_mem[3] = 8'h12;

instr_mem[4] = 8'h30;//irmovq instruction
instr_mem[5] = 8'hF2;
instr_mem[6] = 8'h00;
instr_mem[7] = 8'h00;
instr_mem[8] = 8'h00;
instr_mem[9] = 8'h00;
instr_mem[10] = 8'h00;
instr_mem[11] = 8'h00;
instr_mem[12] = 8'h00;
instr_mem[13] = 8'h02;

instr_mem[14] = 8'h40;//rmmovq instruction
instr_mem[15] = 8'h24;
{instr_mem[16],instr_mem[17],instr_mem[18],instr_mem[19],instr_mem[20],instr_mem[21],instr_mem[22],instr_mem[23]} = 64'd1;

instr_mem[24] = 8'h40;//rmmovq instruction
instr_mem[25] = 8'h53;
{instr_mem[26],instr_mem[27],instr_mem[28],instr_mem[29],instr_mem[30],instr_mem[31],instr_mem[32],instr_mem[33]} = 64'd0;
```

```
instr_mem[34] = 8'h50;//mrmovq instruction
instr_mem[35] = 8'h53;
{instr_mem[36],instr_mem[37],instr_mem[38],instr_mem[39],instr_mem[40],instr_mem[41],instr_mem[42],instr_mem[43]} = 64'd0;

instr_mem[44] = 8'h60;
instr_mem[45] = 8'h9A;

instr_mem[46] = 8'h73;
{instr_mem[47],instr_mem[48],instr_mem[49],instr_mem[50],instr_mem[51],instr_mem[52],instr_mem[53],instr_mem[54]} = 64'd56;

instr_mem[55] = 8'h00;

instr_mem[56] = 8'hA0;
instr_mem[57] = 8'h9F;

instr_mem[58] = 8'hB0;
instr_mem[59] = 8'h9F;

instr_mem[60] = 8'h80;
{instr_mem[61],instr_mem[62],instr_mem[63],instr_mem[64],instr_mem[65],instr_mem[66],instr_mem[67],instr_mem[68]} = 64'd80;

instr_mem[69] = 8'h60;
instr_mem[70] = 8'h56;

instr_mem[71] = 8'h70;
{instr_mem[72],instr_mem[73],instr_mem[74],instr_mem[75],instr_mem[76],instr_mem[77],instr_mem[78],instr_mem[79]} = 64'd46;

instr_mem[80] = 8'h63;
instr_mem[81] = 8'hDE;

instr_mem[82] = 8'h90;
```

OUTPUT

VCD info: dumpfile processor.vcd opened for output.
clk=0 f_predictedPC= 1 F_predictedPC= 0 D_icode= x,E_icode= x, M_icode= x, m_valM= x,
f_stall=x, ifun= x, R1= x, R2= x, R3= x, R4= x, R5= x, R6= x,
R9= x, R10= x, R11= x, R12= x, R13= x, R14= x

clk=1 f_predictedPC= 2 F_predictedPC= 1 D_icode= 1,E_icode= x, M_icode= x, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=0 f_predictedPC= 2 F_predictedPC= 1 D_icode= 1,E_icode= x, M_icode= x, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=1 f_predictedPC= 4 F_predictedPC= 2 D_icode= 1,E_icode= 1, M_icode= x, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=0 f_predictedPC= 4 F_predictedPC= 2 D_icode= 1,E_icode= 1, M_icode= x, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=1 f_predictedPC= 14 F_predictedPC= 4 D_icode= 2,E_icode= 1, M_icode= 1, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=

8643

clk=0 f_predictedPC= 14 F_predictedPC= 4 D_icode= 2,E_icode= 1, M_icode= 1, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=1 f_predictedPC= 24 F_predictedPC= 14 D_icode= 3,E_icode= 2, M_icode= 1, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=0 f_predictedPC= 24 F_predictedPC= 14 D_icode= 3,E_icode= 2, M_icode= 1, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=1 f_predictedPC= 34 F_predictedPC= 24 D_icode= 4,E_icode= 3, M_icode= 2, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=0 f_predictedPC= 34 F_predictedPC= 24 D_icode= 4,E_icode= 3, M_icode= 2, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=1 f_predictedPC= 44 F_predictedPC= 34 D_icode= 4,E_icode= 4, M_icode= 3, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=0 f_predictedPC= 44 F_predictedPC= 34 D_icode= 4,E_icode= 4, M_icode= 3, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=1 f_predictedPC= 46 F_predictedPC= 44 D_icode= 5,E_icode= 4, M_icode= 4, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=0 f_predictedPC= 46 F_predictedPC= 44 D_icode= 5,E_icode= 4, M_icode= 4, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 101, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=1 f_predictedPC= 56 F_predictedPC= 46 D_icode= 6,E_icode= 5, M_icode= 4, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14=
8643

clk=0 f_predictedPC= 56 F_predictedPC= 46 D_icode= 6,E_icode= 5, M_icode= 4, m_valM= x,
f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6=

-143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=1 f_predictedPC= 58 F_predictedPC= 56 D_icode= 7,E_icode= 6, M_icode= 5, m_valM= 50, f_stall=0, ifun= 3, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6= -143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=0 f_predictedPC= 58 F_predictedPC= 56 D_icode= 7,E_icode= 6, M_icode= 5, m_valM= 50, f_stall=0, ifun= 3, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6= -143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=1 f_predictedPC= 60 F_predictedPC= 58 D_icode=10,E_icode= 7, M_icode= 6, m_valM= 50, f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6= -143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=0 f_predictedPC= 60 F_predictedPC= 58 D_icode=10,E_icode= 7, M_icode= 6, m_valM= 50, f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6= -143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=1 f_predictedPC= 80 F_predictedPC= 60 D_icode=11,E_icode=10, M_icode= 7, m_valM= 50, f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6= -143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=0 f_predictedPC= 80 F_predictedPC= 60 D_icode=11,E_icode=10, M_icode= 7, m_valM= 50, f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6= -143, R9= -12345, R10= 12345, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=1 f_predictedPC= 82 F_predictedPC= 80 D_icode= 8,E_icode=11, M_icode=10, m_valM= 50, f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6= -143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=0 f_predictedPC= 82 F_predictedPC= 80 D_icode= 8,E_icode=11, M_icode=10, m_valM= 50, f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6= -143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=1 f_predictedPC= 82 F_predictedPC= 82 D_icode= 6,E_icode= 8, M_icode=11, m_valM= -12345, f_stall=0, ifun= 3, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6= -143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=0 f_predictedPC= 82 F_predictedPC= 82 D_icode= 6,E_icode= 8, M_icode=11, m_valM= -12345, f_stall=0, ifun= 3, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6= -143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=1 f_predictedPC= 82 F_predictedPC= 82 D_icode= 9,E_icode= 6, M_icode= 8, m_valM= -12345, f_stall=1, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 253, R5= 50, R6= -143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=0 f_predictedPC= 82 F_predictedPC= 82 D_icode= 9,E_icode= 6, M_icode= 8, m_valM= -12345,

f_stall=1, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 253, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=1 f_predictedPC= 82 F_predictedPC= 82 D_icode= 1,E_icode= 9, M_icode= 6, m_valM= -12345,
f_stall=1, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 253, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=0 f_predictedPC= 82 F_predictedPC= 82 D_icode= 1,E_icode= 9, M_icode= 6, m_valM= -12345,
f_stall=1, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 253, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=1 f_predictedPC= 82 F_predictedPC= 82 D_icode= 1,E_icode= 1, M_icode= 9, m_valM= 69,
f_stall=1, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 253, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=0 f_predictedPC= 82 F_predictedPC= 82 D_icode= 1,E_icode= 1, M_icode= 9, m_valM= 69,
f_stall=1, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 253, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=1 f_predictedPC= 71 F_predictedPC= 82 D_icode= 1,E_icode= 1, M_icode= 1, m_valM= 69,
f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 253, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=0 f_predictedPC= 71 F_predictedPC= 82 D_icode= 1,E_icode= 1, M_icode= 1, m_valM= 69,
f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 253, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 8643

clk=1 f_predictedPC= 46 F_predictedPC= 71 D_icode= 6,E_icode= 1, M_icode= 1, m_valM= 69,
f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 10204

clk=0 f_predictedPC= 46 F_predictedPC= 71 D_icode= 6,E_icode= 1, M_icode= 1, m_valM= 69,
f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 10204

clk=1 f_predictedPC= 56 F_predictedPC= 46 D_icode= 7,E_icode= 6, M_icode= 1, m_valM= 69,
f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 10204

clk=0 f_predictedPC= 56 F_predictedPC= 46 D_icode= 7,E_icode= 6, M_icode= 1, m_valM= 69,
f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 10204

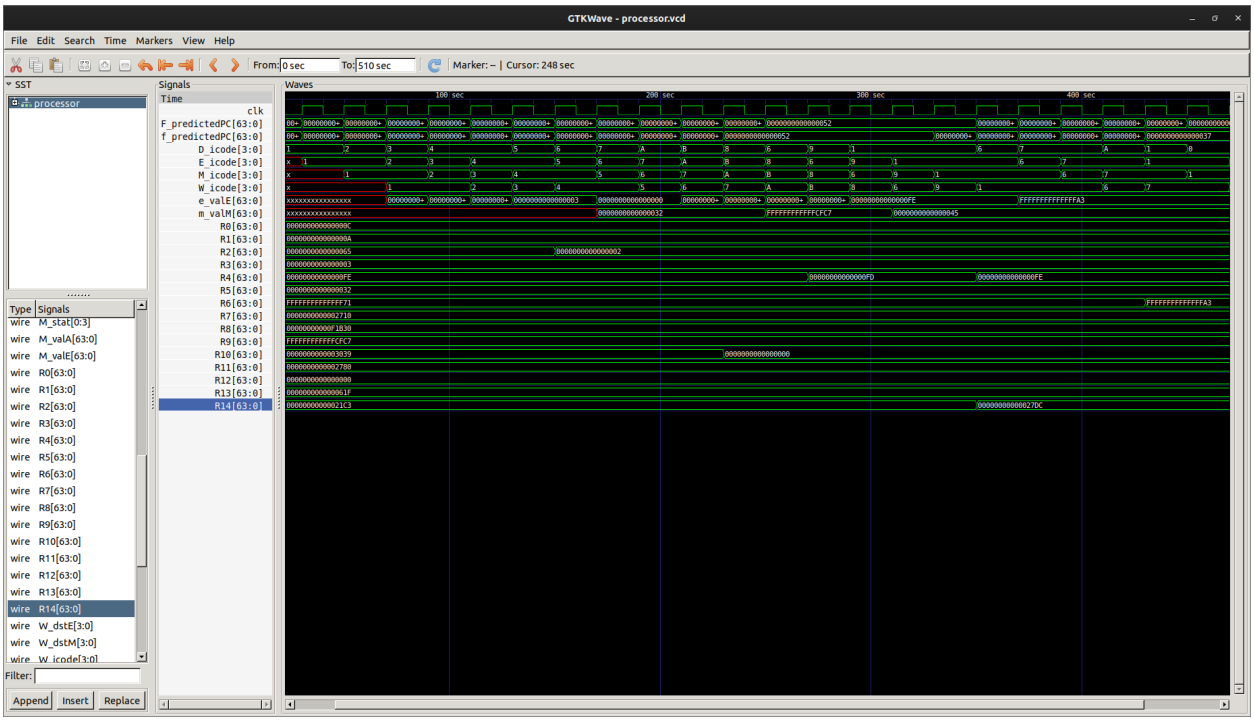
clk=1 f_predictedPC= 58 F_predictedPC= 56 D_icode= 7,E_icode= 7, M_icode= 6, m_valM= 69,
f_stall=0, ifun= 3, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 10204

clk=0 f_predictedPC= 58 F_predictedPC= 56 D_icode= 7,E_icode= 7, M_icode= 6, m_valM= 69,
f_stall=0, ifun= 3, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 10204

clk=1 f_predictedPC= 60 F_predictedPC= 58 D_icode=10,E_icode= 7, M_icode= 7, m_valM= 69,
f_stall=0, ifun= 0, R1= 10, R2= 2, R3= 3, R4= 254, R5= 50, R6=
-143, R9= -12345, R10= 0, R11= 10112, R12= 0, R13= 1567, R14= 10204

```
clk=0 f_predictedPC=          60 F_predictedPC=          58 D_icode=10,E_icode= 7, M_icode= 7, m_valM=          69,
f_stall=0, ifun= 0, R1=          10, R2=          2, R3=          3, R4=          254, R5=          50, R6=
-143, R9=          -12345, R10=          0, R11=          10112, R12=          0, R13=          1567, R14=          10204

clk=1 f_predictedPC=          55 F_predictedPC=          60 D_icode= 1,E_icode= 1, M_icode= 7, m_valM=          69,
f_stall=0, ifun= 0, R1=          10, R2=          2, R3=          3, R4=          254, R5=          50, R6=
-93, R9=          -12345, R10=          0, R11=          10112, R12=          0, R13=          1567, R14=          10204

clk=0 f_predictedPC=          55 F_predictedPC=          60 D_icode= 1,E_icode= 1, M_icode= 7, m_valM=          69,
f_stall=0, ifun= 0, R1=          10, R2=          2, R3=          3, R4=          254, R5=          50, R6=
-93, R9=          -12345, R10=          0, R11=          10112, R12=          0, R13=          1567, R14=          10204

clk=1 f_predictedPC=          55 F_predictedPC=          55 D_icode= 0,E_icode= 1, M_icode= 1, m_valM=          69,
f_stall=0, ifun= 0, R1=          10, R2=          2, R3=          3, R4=          254, R5=          50, R6=
-93, R9=          -12345, R10=          0, R11=          10112, R12=          0, R13=          1567, R14=          10204

clk=0 f_predictedPC=          55 F_predictedPC=          55 D_icode= 0,E_icode= 1, M_icode= 1, m_valM=          69,
f_stall=0, ifun= 0, R1=          10, R2=          2, R3=          3, R4=          254, R5=          50, R6=
-93, R9=          -12345, R10=          0, R11=          10112, R12=          0, R13=          1567, R14=          10204

clk=1 f_predictedPC=          55 F_predictedPC=          55 D_icode= 0,E_icode= 0, M_icode= 1, m_valM=          69,
f_stall=0, ifun= 0, R1=          10, R2=          2, R3=          3, R4=          254, R5=          50, R6=
-93, R9=          -12345, R10=          0, R11=          10112, R12=          0, R13=          1567, R14=          10204

clk=0 f_predictedPC=          55 F_predictedPC=          55 D_icode= 0,E_icode= 0, M_icode= 1, m_valM=          69,
f_stall=0, ifun= 0, R1=          10, R2=          2, R3=          3, R4=          254, R5=          50, R6=
-93, R9=          -12345, R10=          0, R11=          10112, R12=          0, R13=          1567, R14=          10204

clk=1 f_predictedPC=          55 F_predictedPC=          55 D_icode= 0,E_icode= 0, M_icode= 0, m_valM=          69,
f_stall=0, ifun= 0, R1=          10, R2=          2, R3=          3, R4=          254, R5=          50, R6=
-93, R9=          -12345, R10=          0, R11=          10112, R12=          0, R13=          1567, R14=          10204

clk=0 f_predictedPC=          55 F_predictedPC=          55 D_icode= 0,E_icode= 0, M_icode= 0, m_valM=          69,
f_stall=0, ifun= 0, R1=          10, R2=          2, R3=          3, R4=          254, R5=          50, R6=
-93, R9=          -12345, R10=          0, R11=          10112, R12=          0, R13=          1567, R14=          10204

Halt Encounterd, Halting!
clk=1 f_predictedPC=          55 F_predictedPC=          55 D_icode= 0,E_icode= 0, M_icode= 0, m_valM=          69,
f_stall=0, ifun= 0, R1=          10, R2=          2, R3=          3, R4=          254, R5=          50, R6=
-93, R9=          -12345, R10=          0, R11=          10112, R12=          0, R13=          1567, R14=          10204
```

GTKWave



GTKWave corresponding to the output

Challenges Faced in Pipe Lining

In shifting from sequential to pipe lined implementation, we faced several issues.

1. **Data Dependencies :** We might need some register value in an instruction even before write back writes the data into the instruction. To handle this, we use the fact that the data is still somewhere in the pipeline. Therefore, we directly forward the data from that stage to the decode stage of the instruction to use it directly.
2. **Load Use Hazard :** If a data is written to a register in some instruction, and then, later used by some other instruction even before the 1st instruction writes the data into the register, we might end up using the old data in the register and mess up the program. To handle these type of situations a bubble is introduced in the appropriate stage while stalling the previous stages, thereby, waiting for the data to arrive after which we can forward it.
3. **Jump Mis-Prediction :** We are assuming that the conditional jump is taken. This works well for almost 60% of the cases and fails for the rest 40%. In case of failure, which is detected only when the jump instruction reaches the execute stage (where the conditional codes are being set), we must get rid of the 2 stray instructions (not to be executed) which enter the pipeline. We do so by introducing bubbles in the appropriate stages.
4. **Return Instruction :** In case of a ret instruction, we can never predict the target location as we did in jump. Therefore, we **compulsorily** have to stall/bubble the instruction succeeding ret instructions and wait till the ret instruction reaches the memory stage, to find out the exact target location.