INTRODUCTION TO PROCESSOR ARCHITECTURE

COURSE PROJECT

TEAM – 8

TEAM MEMBERS:

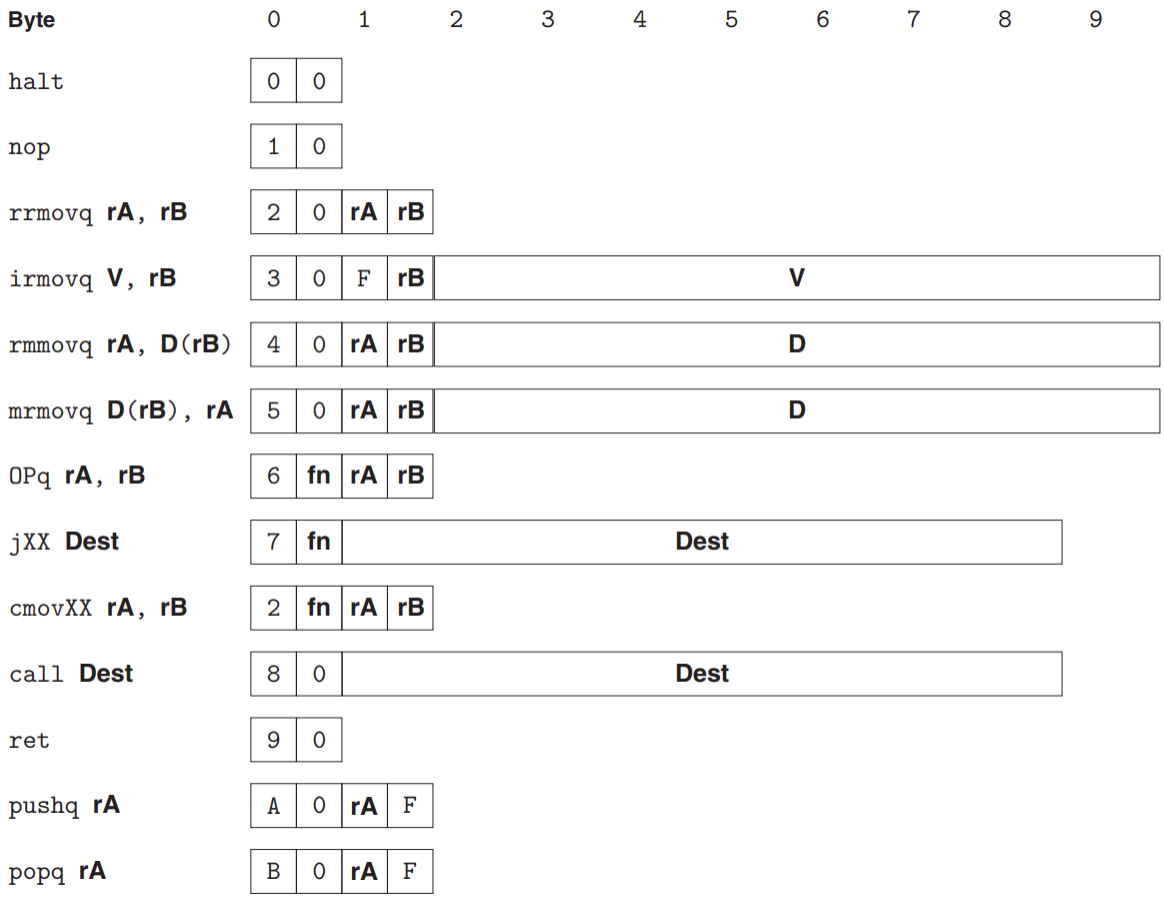
- Abhishek Sharma (2022102004)
- Himanshu Yadav (2022102010)

Introduction

Y86 – 64 instruction set is the simpler version of x86 – 64 ISA which is mainly a subset of the set of instructions available on the x86 – 64 ISA.
It includes only 8-byte integer operations, has fewer addressing modes, and includes a smaller set of operations.
Different Y86-64 instructions are:

- movq instruction of x86-64 is split into four different instructions: irmovq, rrmovq, mrmovq and rmmovq, indicating different sources i.e., immediate(i), register(r) or memory designated by first characters and destination register(r) or memory(m) designated by second characters in the instruction names.
- Operations on integers include addq, subq, andq and xorq. They operate only on register data. The condition codes (ZF, SF, OF) are set by these instructions.
- Conditional jumps denoted by jXX make a choice of taking branches or not based on condition codes.
- Conditional moves denoted by cmovXX make a choice of moving data from one register to another based on condition codes.
- Call pushes the return address on the stack and jumps to destination address. The ret instruction returns from such a call.
- pushq and popq instructions implement push and pop.
- Halt instruction stops instruction execution.

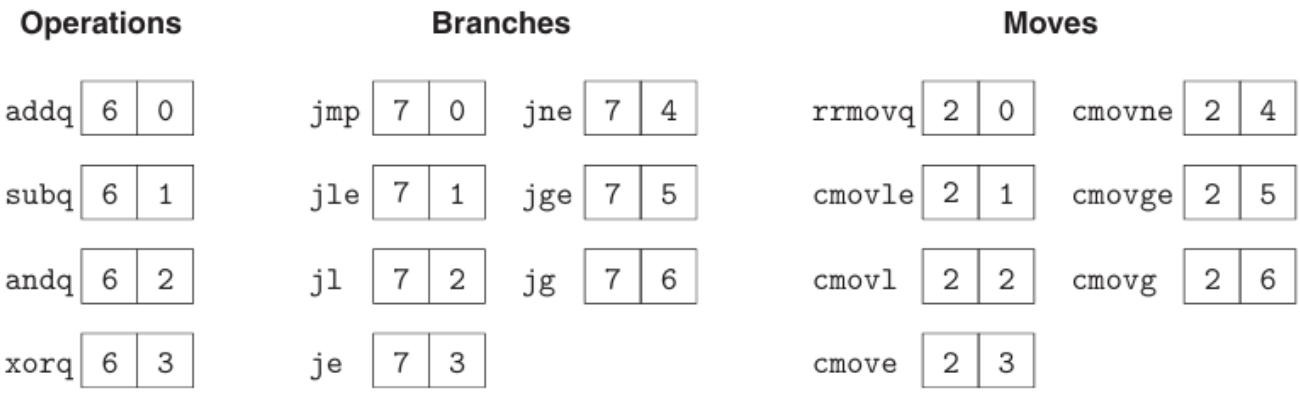
These instructions are implemented at the machine level using machine encoding of these instructions which is done as follows:



Each of the program registers are assigned an id ranging from 0x0 to 0xE. ID value 0xF is used in the instruction encodings and within our hardware designs when we need to indicate that no register should be accessed.

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

As with x86-64, all integers have a little-endian encoding. When the instruction is written in disassembled form, these bytes appear in reverse order.



Above table shows function ids for any given instruction (jXX, OPq, cmovXX). For jXX and cmovXX the function id corresponds to the condition for which branch is to be taken or move is to be completed respectiveery. For OPq, function id describes which operation (addq, subq, andq, or xorq) is to be done on the given set of integers.

Sequential Implementation

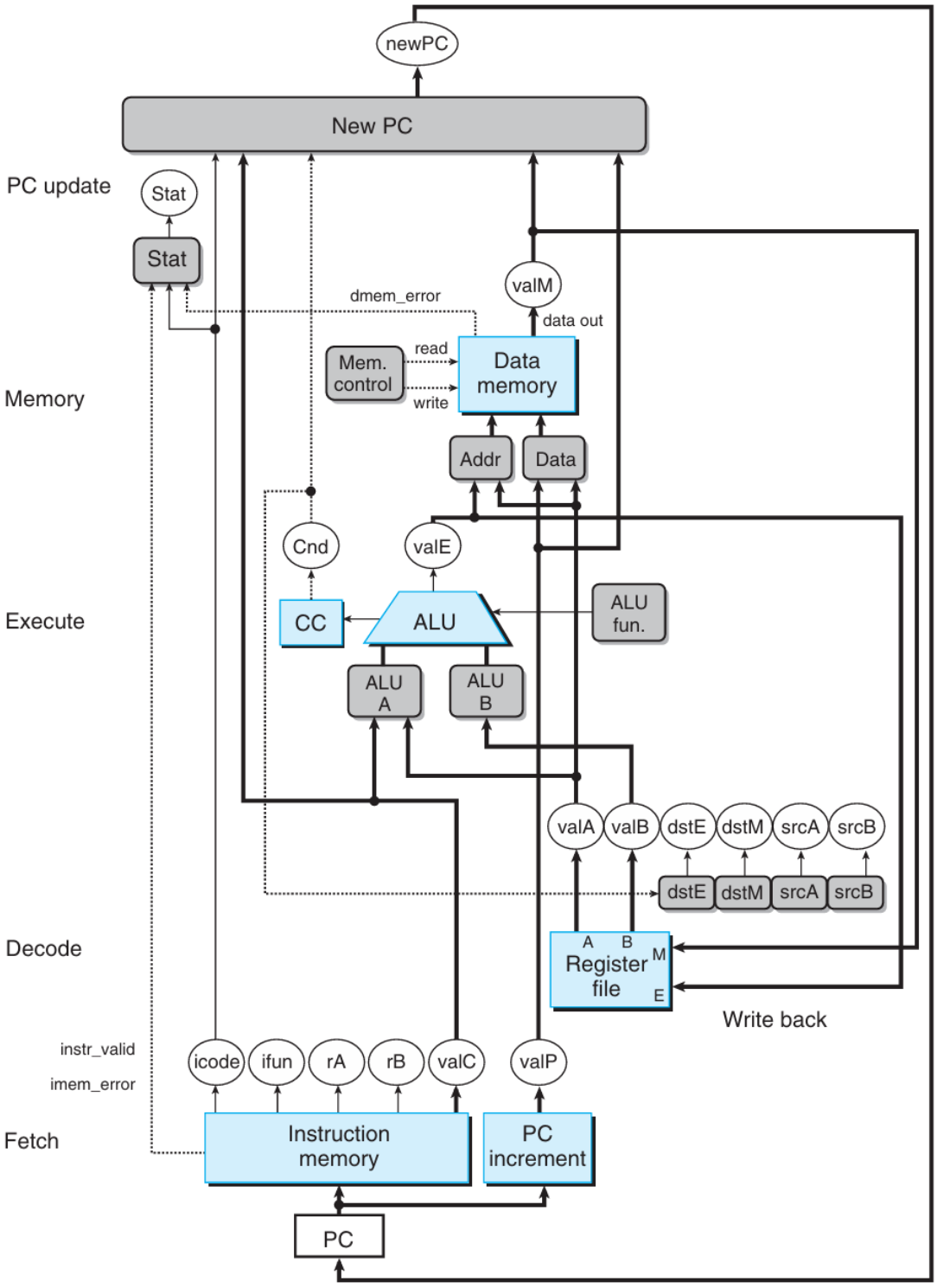


Figure – Hardware Structure of Sequential Implementation

Fetch

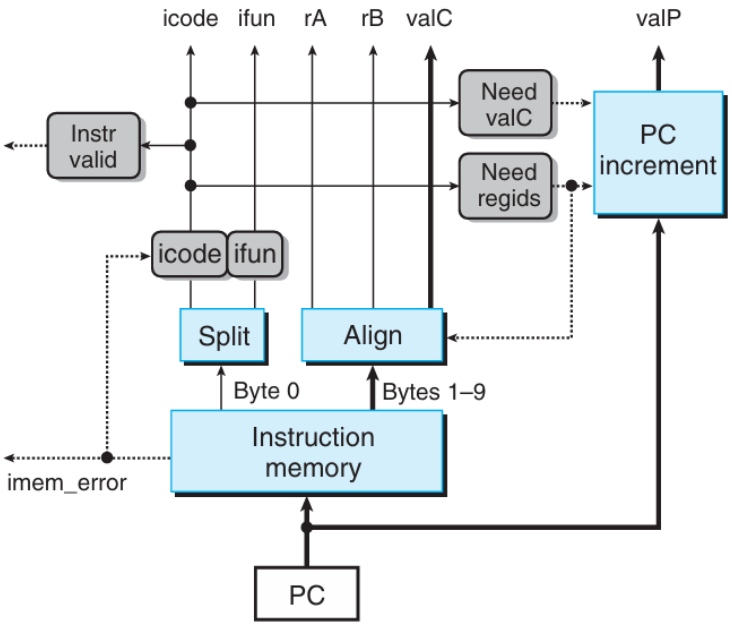


Figure – Fetch Stage

As shown in Figure above the fetch stage includes the instruction memory hardware unit. This unit reads 10 bytes from memory at a time, using the PC as the address of the first byte (byte 0). This byte is interpreted as the instruction byte and is split into two 4-bit quantities, the first 4-bits are assigned to icode and the next 4-bits are assigned to ifun. Now based on the value of the icode we compute three 1-bit signals named instr_valid, need_regids, need_valC.

The logic for these signals is as follows.

need_regids will be 1 when icode is equal to any of the value in [2(rrmovq), 3(irmovq) , 4(rmmovq), 5(mrmovq), 6(opq), 10(pushq), 11(popq)] and 0 otherwise.

need_valC will be 1 when icode is equal to any of the value in [3(irmovq), 4(rmmovq), 5(mrmovq), 7(jxx), 8(call)] and 0 otherwise.

instr_valid will be 1 when when icode is in [1, 2, ...,12] and 0 otherwise.

If need_regids is 1 then the second byte (byte 1) read will be split into two 4-bit quatities and the first 4-bits are assigned to rA and the next 4-bits are assigned to rB.

If need_valC is 1 then the next 8 bytes (from byte 2-10 if need_regids is 1 else from byte 1-9) read are alligned (first byte corresponds to the least significant byte and so on) to valC.

Then valP is calculated as valP = PC + 1 + need_regids + 8*need_valC.

There is one more 1-bit output that comes out from fetch stage that is imem_error which is one when the instruction address is out of bounds, and this generated in the instruction memory hardware unit.

For the instr_valid we are also checking for the validity of the ifun for each ifun.

We are fetching the instructions at the positive edge of the clock.

Source Code
Instruction Memory

```
module instructionMemory(valRead0,valRead1,valRead2,valRead3,valRead4,valRead5,valRead6,valRead7,valRead8,valRead9,imem_error,PC);
    output reg [7:0] valRead0;
    output reg [7:0] valRead1;
    output reg [7:0] valRead2;
    output reg [7:0] valRead3;
    output reg [7:0] valRead4;
    output reg [7:0] valRead5;
    output reg [7:0] valRead6;
    output reg [7:0] valRead7;
    output reg [7:0] valRead8;
    output reg [7:0] valRead9;
    output reg imem_error;
    input [63:0] PC;
    reg [7:0] instructionMem [0:1023];
    initial
    begin
        $readmemb("./../TestCases/t2.txt",instructionMem);
    end
    always @(*)
    begin
        imem_error = 0;
        if (PC > 1023)
        begin
            imem_error = 1;
        end
        else
        begin
            valRead0 = instructionMem[PC];
            valRead1 = instructionMem[PC+1];
            valRead2 = instructionMem[PC+2];
            valRead3 = instructionMem[PC+3];
            valRead4 = instructionMem[PC+4];
            valRead5 = instructionMem[PC+5];
            valRead6 = instructionMem[PC+6];
            valRead7 = instructionMem[PC+7];
            valRead8 = instructionMem[PC+8];
            valRead9 = instructionMem[PC+9];
        end
    end

endmodule
```

Fetch

```
`include "instructionMemory.v"

module fetch(icode,ifun,rA,rB,valC,valP,instr_valid,imem_error,clk,PC);
    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 instr_valid;
    output reg imem_error;
    input clk;
    input wire [63:0] PC;
    reg need_regids;
    reg need_valC;
    wire [7:0] im_out [0:9];
    wire imem_errorw;
    initial
    begin
        valP = 0;
    end

    instructionMemory X1(im_out[0],im_out[1],im_out[2],im_out[3],im_out[4],im_out[5],im_out[6],im_out[7],im_out[8],im_out[9],imem_errorw,PC);
    always @(posedge clk)
    begin
        icode = im_out[0][7:4];
        ifun = im_out[0][3:0];
```

```
        imem_error = imem_errorw;
        if(icode == 2 || icode == 3 || icode == 4 || icode == 5 ||icode == 6 || icode == 10 || icode == 11)
        begin
            need_regids = 1;
            rA = im_out[1][7:4];
            rB = im_out[1][3:0];
        end
        else
        begin
            need_regids = 0;
            rA = 15;
            rB = 15;
        end

        if (icode == 3 || icode == 4 || icode == 5 || icode == 7 || icode == 8)
        begin
            need_valC = 1;
            valC[7:0] = im_out[need_regids+1];
            valC[15:8] = im_out[need_regids+2];
            valC[23:16] = im_out[need_regids+3];
            valC[31:24] = im_out[need_regids+4];
            valC[39:32] = im_out[need_regids+5];
            valC[47:40] = im_out[need_regids+6];
            valC[55:48] = im_out[need_regids+7];
            valC[63:56] = im_out[need_regids+8];
        end
        else
        begin
            need_valC = 0;
        end


        valP = PC + 1 + need_regids + 8*need_valC;
        if(icode >= 0 && icode <= 11)
        begin
            instr_valid = 1;
        end
        if ((icode == 0 || icode == 1 || icode == 3 || icode == 4 || icode == 5||icode == 8 || icode == 9 || icode == 10 || icode == 11) && ifun!=0)
        begin
            instr_valid = 0;
        end

        if (((icode == 2 || icode == 7) && ifun>6) || (icode==6 && ifun>3))
        begin
            instr_valid = 0;
        end
    end
endmodule
```

Decode and Write Back

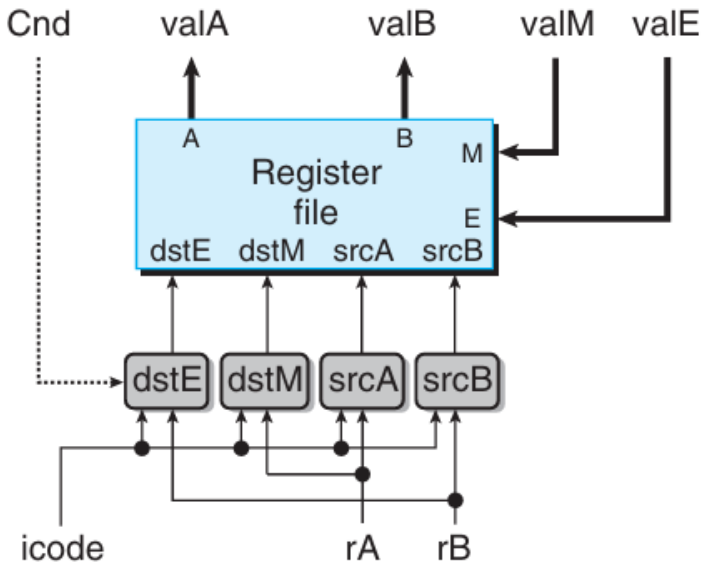


Figure – Decode and Write Back Stage

As shown in Figure above the decode and write back stages are combined since both of them need access to the register file. The register file is a hardware part which has four ports. It supports up to two simultaneous reads (on ports A and B) and two simultaneous writes (on ports E and M). Each port has both an address connection and a data connection, where the address connection is a register ID, and the data connection is a set of 64 wires serving as either an output word (for a read port) or an input word (for a write port) of the register file. The two read ports have address inputs srcA and srcB, while the two write ports have address inputs dstE and dstM. The inputs to the decode and write back module are Cnd, icode, rA, rB.

Decode stage's function is to decode which register is to be used as source and destination for data transfer for operations which needs access to different registers and values stored in them. Here we get inputs from fetch stage the register ids to be used in any particular instruction and then as per the different icode values, we choose them for their functioning.

Since the Writeback stage also needs access to the register file for write function, we have carried out this stage also in the same module. However, since writeback can occur only after execute and memory stages have been carried out thus we have operated this stage on negative edge of the clock and thus the exception that may occur due to operating every stage at same edge of clock cycle is avoided. The destination to which we are to write is also decided based on icode values we get as input from fetch stage.

Whenever there is no need for a register in destination or source registers, we assign it the value 0xF which is designated for no register needed. Thus srcA, srcB, dstE, dstM are assigned oxF in else case.

In case of conditional move (cmovXX), Cnd flag is generated by execute stage which determines whether the move operation is to be carried out or not. Thus, an if condition is added in case of icode = 2 and thus operation does not occur when Cnd flag is 0.

Source Code

Decode and Write Back

```
module decode_WriteBack(valA, valB, rA, rB, valE, valM, icode, Cnd, clk);
    output reg [63:0] valA;
    output reg [63:0] valB;
    input [63:0] valE;
    input [63:0] valM;
    input [3:0] icode;
    input [3:0] rA;
    input [3:0] rB;
    reg [3:0] dstE;
    reg [3:0] dstM;
    input Cnd;
    reg [3:0] srcA;
    reg [3:0] srcB;
    input clk;

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

```
    genvar i;
    generate
        for (i=0; i<16;i = i+1)begin
            initial begin
                reg_file[i] = i;
            end
        end
    endgenerate

    always @(*)
    begin
        if(icode == 2 || icode ==4 || icode == 6 || icode == 10)
        begin
            srcA = rA;
        end
        else if(icode == 11 || icode == 9)
        begin
            srcA = 4;
        end
        else
        begin
            srcA = 15;
        end
        if(icode == 4 || icode == 5 || icode == 6)
        begin
            srcB = rB;
        end
        else if(icode == 10 || icode == 11 || icode == 8 || icode == 9)
        begin
            srcB = 4;
        end
        else
        begin
            srcB = 15;
        end
        valA = reg_file[srcA];
        valB = reg_file[srcB];
    end

    always@ (negedge clk)
    begin
        if(icode == 6 || icode == 3)
        begin
            dstE = rB;
        end
        else if(icode == 2)
        begin
            if(Cnd)
```

```
            begin
                dstE = rB;
            end
            else
            begin
                dstE = 15;
            end
        end
        else if(icode == 10 || icode == 11 || icode == 8 || icode == 9)
        begin
            dstE = 4;
        end
        else
        begin
            dstE = 15;
        end
        if(icode == 5 || icode == 11)
        begin
            dstM = rA;
        end
        else
        begin
            dstM = 15;
        end
        reg_file[dstE] = valE;
        reg_file[dstM] = valM;
    end

    initial
    begin
        $monitor("rax = %d ,rcx = %d, rdx = %d, rbx = %d, rsp = %d, rbp = %d, rsi = %d, rdi =
%d, r8 = %d, r9 = %d, r10 = %d, r11 = %d, r12 = %d, r13 = %d, r14 = %d, r15 =
%d",reg_file[0],reg_file[1],reg_file[2],reg_file[3],reg_file[4],reg_file[5],reg_file[6],reg_f
ile[7],reg_file[8],reg_file[9],reg_file[10],reg_file[11],reg_file[12],reg_file[13],reg_file[1
4], reg_file[15]);
    end
endmodule
```

Execute

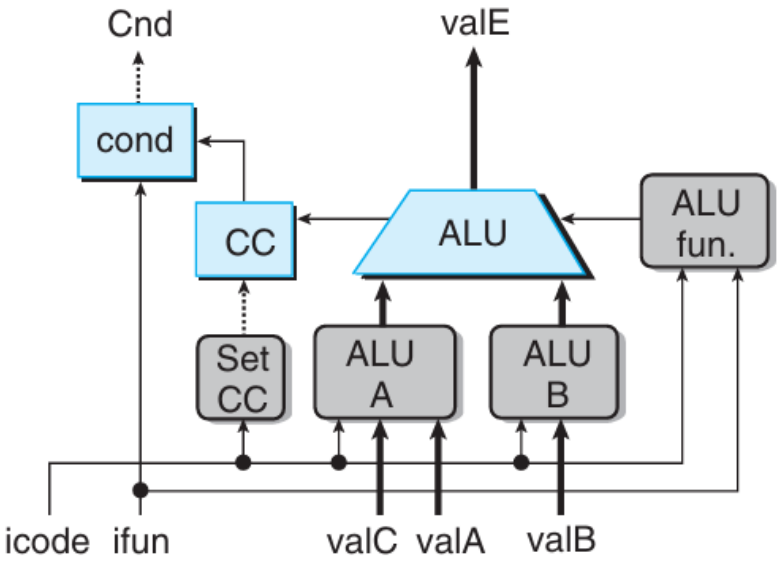


Figure – Execute Stage

As shown in Figure above the inputs to the execute module are icode, ifun, valC, valA, valB. Based on the icode and ifun we calculate the value of the aluA, aluB, and aluFun as follows.

aluA is equal to valA when icode in {2, 6}, valC when icode in {3, 4, 5}, -8 when icode in {8, 10} and 8 when icode in {9, 11}.

aluB is equal to valB when icode in {4, 5, 6, 8, 9, 10, 11} and 0 when icode in {2, 3}.

aluFun is equal to ifun when icode is equals to 6 and 0(add) for all other cases.

When the icode is 6 based on the output of the ALU we are setting the condition codes as follows.

When the output of the ALU is 0 then ZF is set to 1, when output of the ALU is less than zero then SF is set to 1, and when there is an overflow in ALU operation then OF is set to 1. All these condition codes are set at negative edge of the clock this is because the execute block & decode block is working everytime when the input changes and at the negative edge after writeback the valA and valB was changing however the icode is still the same and hence the condition codes were updating unnecessarily.

For the icode in {2, 7} the output Cnd is calculated according to following table.

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

Source Code
Execute

```
`include "ALU_Wrapper.v"

module execute(Cnd,valE,valA,valB,valC,icode,ifun,clk);
    output reg [63:0] valE;
    output reg Cnd;
    input [63:0] valA;
    input [63:0] valB;
    input [63:0] valC;
    input [3:0] icode;
    input [3:0] ifun;
    input clk;
    reg ZF;
    reg SF;
    reg OF;
    wire [63:0] aluOut;
    wire aluOF;
    reg [63:0] aluA;
    reg [63:0] aluB;
    reg [1:0] aluFun;

    ALU_Wrapper X1(aluOF,aluOut,aluFun,aluB, aluA);

    initial begin
        OF = 0;
        ZF = 0;
        SF = 0;
    end

    always @(*)
    begin
        if (icode == 2 || icode == 6)
        begin
            aluA = valA;
        end
        else if (icode == 3 || icode == 4 || icode == 5)
        begin
            aluA  = valC;
        end
        else if (icode == 8 || icode == 10)
        begin
            aluA = -8;
        end
        else if (icode == 9 || icode == 11)
        begin
            aluA = 8;
        end
        else
        begin
            aluA = 0;
        end

        if (icode == 4 || icode == 5 || icode == 6 || icode == 8 || icode == 9 || icode == 10 || icode == 11)
        begin
            aluB = valB;
        end
        else if (icode == 2 || icode == 3)
        begin
            aluB = 0;
```

```
        end
        else
        begin
            aluB = 0;
        end

        if (icode == 6)
        begin
            aluFun = ifun[1:0];
        end
        else
        begin
            aluFun = 0;
        end

        valE = aluOut;

        if (icode == 2 || icode == 7)
        begin
            if (ifun == 0)
            begin
                Cnd = 1;
            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 if (ifun == 6)
            begin
                Cnd = ~(SF^OF) & ~ZF;
            end
        end
        else
        begin
            Cnd = 0;
        end
    end
    always @(negedge clk)
    begin
        if (icode == 6)
        begin
            OF = aluOF;
            if (aluOut[63] == 1)
            begin
                SF = 1;
```

```
            end
            else
            begin
                SF = 0;
            end

            if (aluOut == 0)
            begin
                ZF = 1;
            end
            else
            begin
                ZF = 0;
            end
        end
    end
endmodule
```

Memory

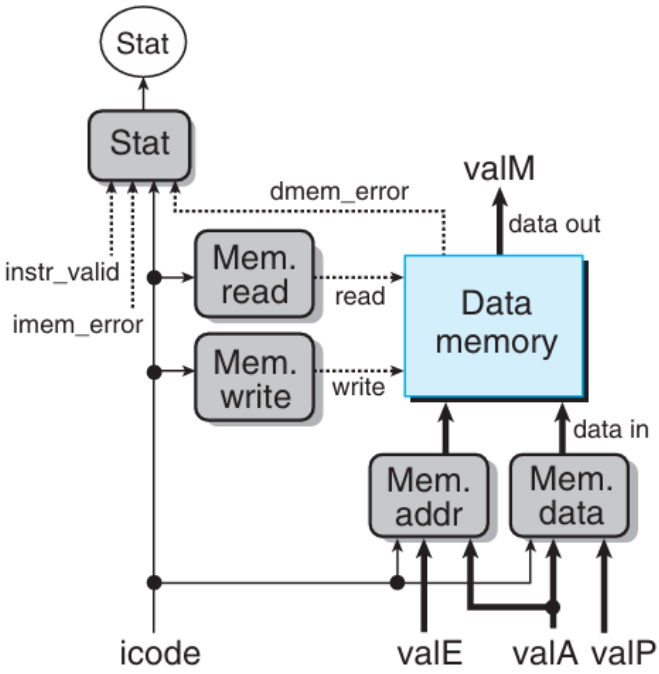


Figure – Memory Stage

Data memory is the main storage unit of our processor design where data is written in case of rmmovq, call, or pushq and data is read from memory in case of mrmovq, ret, or popq. Call writes the return address from this function call and pushq writes data at the new top address of stack. ret takes the return address written in call and thus is used to take the PC to that instruction value and popq reads the data written at top of the stack. Memory stage also generates dmem_error which is activated in case of input data address accesses address which is not inside the limit of predefined memory size. There are 4 signals that need to be calculated to do this which are mem_addr, mem_data, mem_read and mem_write. The logic for all these signals is as follows. mem_addr is equal to valE when icode in {4, 5, 8, 10} and valA when icode in {9, 11}. mem_data is equal to valA when icode in {4, 10} and valP when icode is 8.

mem_read will be 1 when icode in {5, 9, 11}.
mem_write will be 1 when icode in {4, 8, 10}.
Memory stage also computes the value of stat which determines the instruction status. The logic for the stat signal is as follows.
stat will be 2(ADR) when imem_error or dmem_error is 1, 3(INS) when instr_valid is 0, 4(HLT) when icode is 0 and 1(AOK) otherwise.

Source Code

Memory

```
module data_memory(valM, stat, valA, valP, valE, icode, instr_valid, imem_error, clk);
    output reg [2:0] stat;
    output reg [63:0] valM;
    input clk;
    input [63:0] valA;
    input [63:0] valP;
    input [63:0] valE;
    input [3:0] icode;
    input instr_valid;
    input imem_error;
    reg dmemError;
    reg [63:0] memReg[0:8191];
    genvar i;
    generate
        for (i=0; i<64;i = i+1)begin
            initial begin
                memReg[i] = 0;
            end
        end
    endgenerate
    reg [63:0] mem_addr;
    reg [63:0] mem_data;
    reg mem_read;
    reg mem_write;
    always @(*)
    begin
        if(icode == 10 || icode == 8 || icode == 4 || icode == 5)
        begin
            mem_addr = valE;
        end
        else if(icode == 11 || icode == 9)
        begin
            mem_addr = valA;
        end

        if (icode == 4 || icode == 10)
        begin
            mem_data = valA;
        end
```

```
        else if (icode == 8)
        begin
            mem_data = valP;
        end

        if (icode == 5 || icode == 9 || icode == 11)
        begin
            mem_read = 1;
        end
        else
        begin
            mem_read = 0;
        end

        if (icode == 4 || icode == 8 || icode == 10)
        begin
            mem_write = 1;
        end
        else
        begin
            mem_write = 0;
        end
        if (mem_read == 1)
        begin
            valM = memReg[mem_addr];
        end

        dmemError =0;
        if(mem_addr > 8191)
        begin
            dmemError = 1;
        end

        if(instr_valid == 1)
        begin
            stat = 1;
        end
        else if(instr_valid == 0)
        begin
            stat = 3;
        end

        if(imem_error == 1 || dmemError == 1)
        begin
            stat = 2;
        end
        if(icode == 0)
        begin
```

```
            stat = 4;
        end
    end
    always@(negedge clk)
    begin
        if (mem_write == 1)
        begin
            memReg[mem_addr] = mem_data;
        end
    end

endmodule
```

PC Update

The final stage in sequential design is updating the program counter. The logic for new_pc is as follows.
new_pc will be valC when icode is 8(call) or icode is 7(jXX) and Cnd is 1, valM when icode is 9(ret) and valP otherwise.

Source Code

PC Update

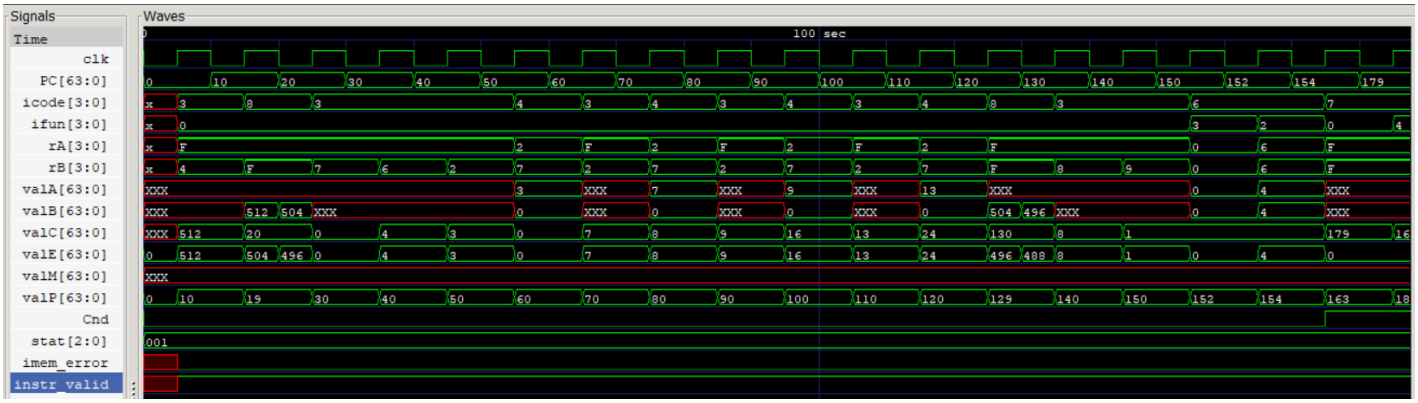
```
module PC_update(PC, icode, valP, valC, valM, Cnd, clk);
    output reg [63:0] PC;
    input [3:0] icode;
    input clk;
    input [63:0] valC;
    input [63:0] valP;
    input [63:0] valM;
    input Cnd;
    always @(*)
    begin
        if(icode == 7)
        begin
            if(Cnd == 1)
            begin
                PC=valC;
            end
            else
            begin
                PC = valP;
            end
        end
        else if(icode == 8)
        begin
            PC = valC;
        end
        else if(icode == 9)
```

```
        begin
            PC=valM;
        end
        else
        begin
            PC = valP;
        end
    end
endmodule
```

Output

Assembly Code

```
 0x0000:                      |     .pos 0
 0x0000: 30f40002000000000000 |     irmovq stack, %rsp # Set up stack pointer
 0x000a: 801400000000000000   |     call main # Execute main program
 0x0013: 00                   |     halt # Terminate program
                              |
 0x0014:                      | main:
 0x0014: 30f70000000000000000 |     irmovq $0,%rdi
 0x001e: 30f60400000000000000 |     irmovq $4,%rsi
 0x0028: 30f20300000000000000 |     irmovq $3,%rdx
 0x0032: 40270000000000000000 |     rmmovq %rdx,(%rdi)
 0x003c: 30f20700000000000000 |     irmovq $7,%rdx
 0x0046: 40270800000000000000 |     rmmovq %rdx,8(%rdi)
 0x0050: 30f20900000000000000 |     irmovq $9,%rdx
 0x005a: 40271000000000000000 |     rmmovq %rdx,16(%rdi)
 0x0064: 30f20d00000000000000 |     irmovq $13,%rdx
 0x006e: 40271800000000000000 |     rmmovq %rdx, 24(%rdi)
 0x0078: 808200000000000000   |     call sum # sum(array, 4)
 0x0081: 90                   |     ret
                              |
                              | # long sum(long *start, long count)
                              | # start in %rdi, count in %rsi
 0x0082:                      | sum:
 0x0082: 30f80800000000000000 |     irmovq $8,%r8 # Constant 8
 0x008c: 30f90100000000000000 |     irmovq $1,%r9 # Constant 1
 0x0096: 6300                 |     xorq %rax,%rax # sum = 0
 0x0098: 6266                 |     andq %rsi,%rsi # Set CC
 0x009a: 70b300000000000000   |     jmp test # Goto test
 0x00a3:                      | loop:
 0x00a3: 50a70000000000000000 |     mrmovq (%rdi),%r10 # Get *start
 0x00ad: 60a0                 |     addq %r10,%rax # Add to sum
 0x00af: 6087                 |     addq %r8,%rdi # start++
 0x00b1: 6196                 |     subq %r9,%rsi # count--. Set CC
 0x00b3:                      | test:
 0x00b3: 74a300000000000000   |     jne loop # Stop when 0
 0x00bc: 90                   |     ret # Return
                              |
                              |      # Stack starts here and grows to lower addresses
 0x00bd:                      |     .pos 0x200
 0x0200:                      | stack:
```



We are fetching the instructions at the positive edge of the clock and updating the PC at the negative edge of the clock. This way we are getting one complete clock cycle for a single instruction.

```
  0x0000: 30f40002000000000000 |     irmovq stack, %rsp # Set up stack pointer.
```

Here we can verify that the values of all the variables are what that should be i.e., icode = 3, valC = 512 etc.

```
  0x000a: 801400000000000000   |     call main # Execute main program.
```

Here valC = 20 and that is also the value of the next PC.

```
  0x0014:                      | main:
  0x0014: 30f70000000000000000 |     irmovq $0,%rdi
  0x001e: 30f60400000000000000 |     irmovq $4,%rsi
  0x0028: 30f20300000000000000 |     irmovq $3,%rdx
  0x0032: 4027000000000000000  |     rmmovq %rdx,(%rdi)
  0x003c: 30f20700000000000000 |     irmovq $7,%rdx
  0x0046: 4027080000000000000  |     rmmovq %rdx,8(%rdi)
  0x0050: 30f20900000000000000 |     irmovq $9,%rdx
  0x005a: 4027100000000000000  |     rmmovq %rdx,16(%rdi)
  0x0064: 30f20d00000000000000 |     irmovq $13,%rdx
  0x006e: 4027180000000000000  |     rmmovq %rdx, 24(%rdi)
```

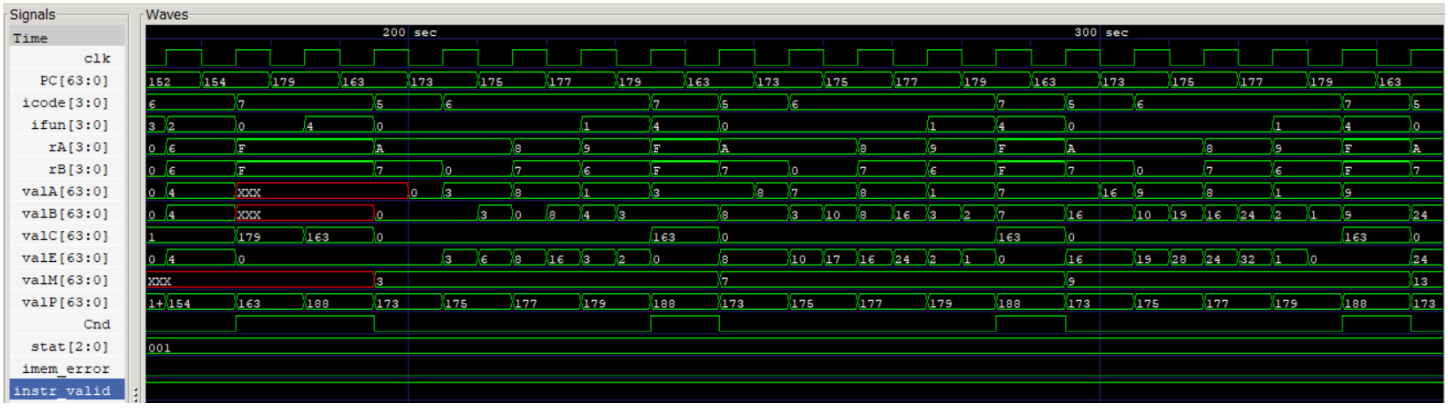
These are the instructions in main. Here the value in %rdi is 0, now for the first rmmovq valE is 0 as there is no any constant given for addressing but for the second, third and fourth the value of valE is 8, 16, 24 respectively and that is the correct value.

```
0x0078: 808200000000000000   |     call sum # sum(array, 4)
```

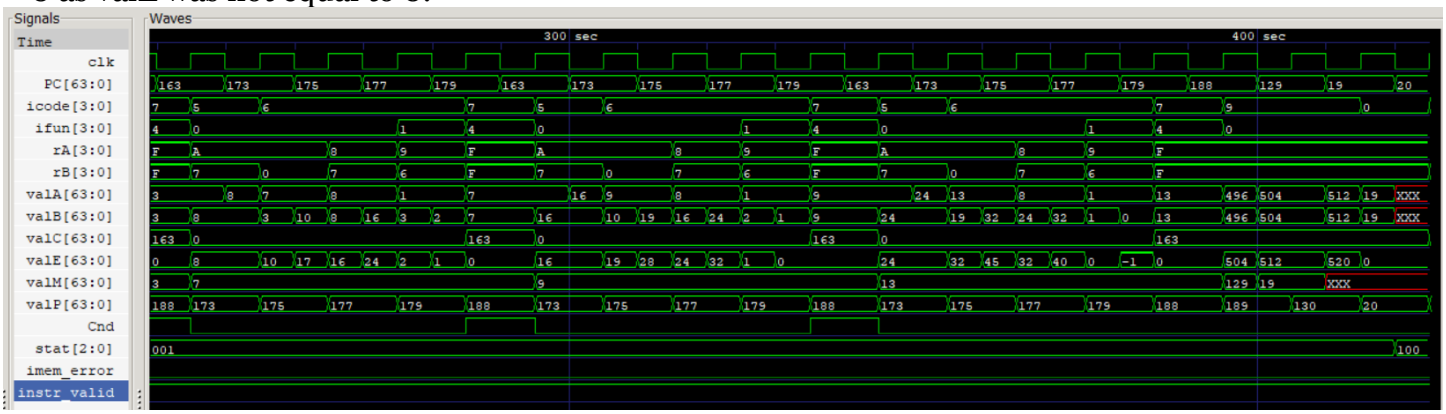
Here valC = 130, and call is working as before.

```
  0x0082:                      | sum:
  0x0082: 30f80800000000000000 |     irmovq $8,%r8 # Constant 8
  0x008c: 30f90100000000000000 |     irmovq $1,%r9 # Constant 1
  0x0096: 6300                 |     xorq %rax,%rax # sum = 0
  0x0098: 6266                 |     andq %rsi,%rsi # Set CC
  0x009a: 70b300000000000000   |     jmp test # Goto test
  0x00a3:                      | loop:
  0x00a3: 50a70000000000000000 |     mrmovq (%rdi),%r10 # Get *start
  0x00ad: 60a0                 |     addq %r10,%rax # Add to sum
  0x00af: 6087                 |     addq %r8,%rdi # start++
  0x00b1: 6196                 |     subq %r9,%rsi # count--. Set CC
  0x00b3:                      | test:
  0x00b3: 74a300000000000000   |     jne loop # Stop when 0
  0x00bc: 90                   |     ret # Return
```

Here for PC = 150, icode = 6, ifun = 3, valE = 0 and that is correct output.
For PC = 152, icode = 6, ifun = 2, valE = 4 that is correct output as (%rsi = 4).
For PC = 154, icode = 7, ifun = 0, valC = 179, Cnd = 1 and next PC = 179 that is correct.



For PC = 179, valC = 163, icode = 7, ifun = 4, Cnd = 1 as for previous opq operation that was andq ZF = 0 as valE was not equal to 0.

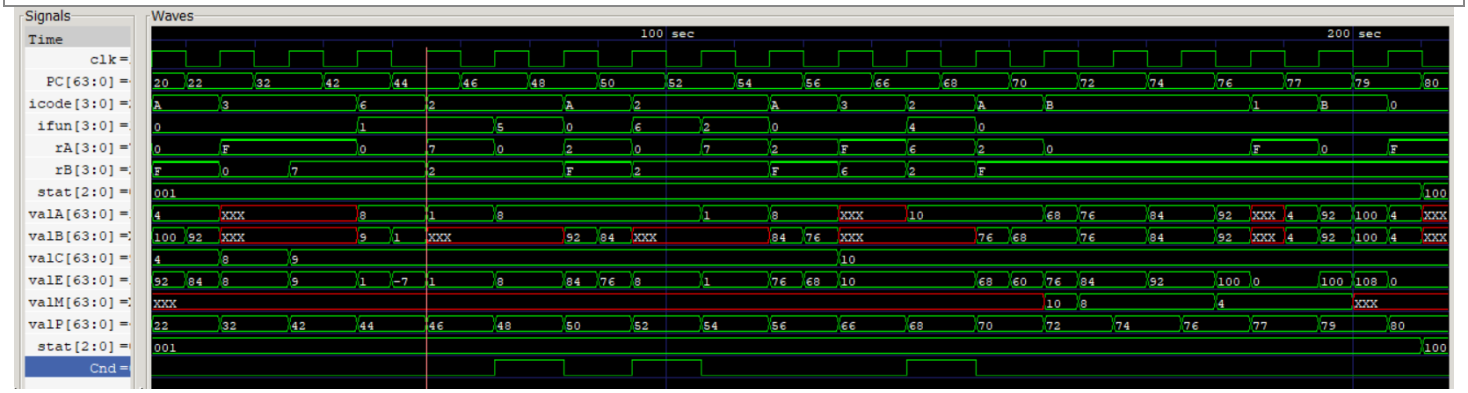


Now the loop runs for 4 iterations till the ZF set by subq operation is 1.
The value of valE corresponding to mrmovq operation of the loop that is at PC = 163 is 0, 8, 16, 24 in respective iteration.
The value of valM corresponding to mrmovq operation of the loop that is at PC = 163 is 3, 7, 9, 13 in respective iteration.
The values of valE corresponding to addq operation of the loop that is at PC = 173 is 3, 10 (3+7), 19(3+7+9), 32(3+7+9+13) in the respective iteration.
After the loop there is a return for the sum function and hence at PC = 188, icode = 9, valM = 129 (this is the address of the next instruction after the call instruction of sum) that is also a ret for main function.
Here valM = 19(this is that address of the next instruction after the call instruction of main) which is a halt instruction which set stat = 4 and the program terminates.

Assembly Code

```
0x0000: 30f46400000000000000   | irmovq $100, %rsp
0x000a: 30f00400000000000000   | irmovq $4, %rax
0x0014: a00f                   | pushq %rax
0x0016: 30f00800000000000000   | irmovq $8, %rax
0x0020: 30f70900000000000000   | irmovq $9, %rdi
0x002a: 6107                   | subq %rax, %rdi
0x002c: 2172                   | cmovle %rdi, %rdx
0x002e: 2502                   | cmovge %rax, %rdx
0x0030: a02f                   | pushq %rdx
0x0032: 2602                   | cmovg %rax, %rdx
0x0034: 2272                   | cmovl %rdi, %rdx
0x0036: a02f                   | pushq %rdx
0x0038: 30f60a00000000000000   | irmovq $10, %rsi
0x0042: 2462                   | cmovne %rsi, %rdx
0x0044: a02f                   | pushq %rdx
0x0046: b00f                   | popq %rax
0x0048: b00f                   | popq %rax
```

```
0x004a: b00f                | popq %rax
0x004c: 10                  | nop
0x004d: b00f                | popq %rax
0x004f: 00                  | halt
```



Here first two instructions are irmovq which are explained earlier in first code.

```
0x0014: a00f                | pushq %rax
```

Here valA = 4 (as the value in %rax was 4), valB = 100 (as the value in %rsp was 100), valE = 92 (this is the new value in %rsp). Now the top of the stack has value 4.

```
0x002c: 2172                | cmovle %rdi, %rdx
0x002e: 2502                | cmovge %rax, %rdx
```

Now at PC = 44, icode = 2, ifun = 1(cmovle), valA = 1 (as %rdi has value 1 after subq), valE = 1 (0+valA), Cnd = 0 (because the previous valE at opq was 1 hence SF and ZF both are zero), Hence the register-to-register move did not happen.
Now at PC = 46, icode = 2, ifun = 5(comge), valA = 8, valE = 8, Cnd = 1 hence the register-to-register move happened in this case and hence value in %rdx is 8.

```
0x0030: a02f                | pushq %rdx
```

Now at PC = 48, icode = 10, valA = 8(value in %rdx), valB = 92(value in %rsp). Now the top of the stack has value 8.
Similarly, we do two more push operations with some cmov operations in this order value 8 and then 10.

```
0x0046: b00f                | popq %rax
```

Now at PC = 8, icode = 11, valA = valB = 68 (value in %rsp), valM = 10 (this was the last value pushed).
Similarly, after 3 more pop operations we get values 8, 8, 4 in respective popq operation as in that order the values are pushed.

Pipelined Implementation

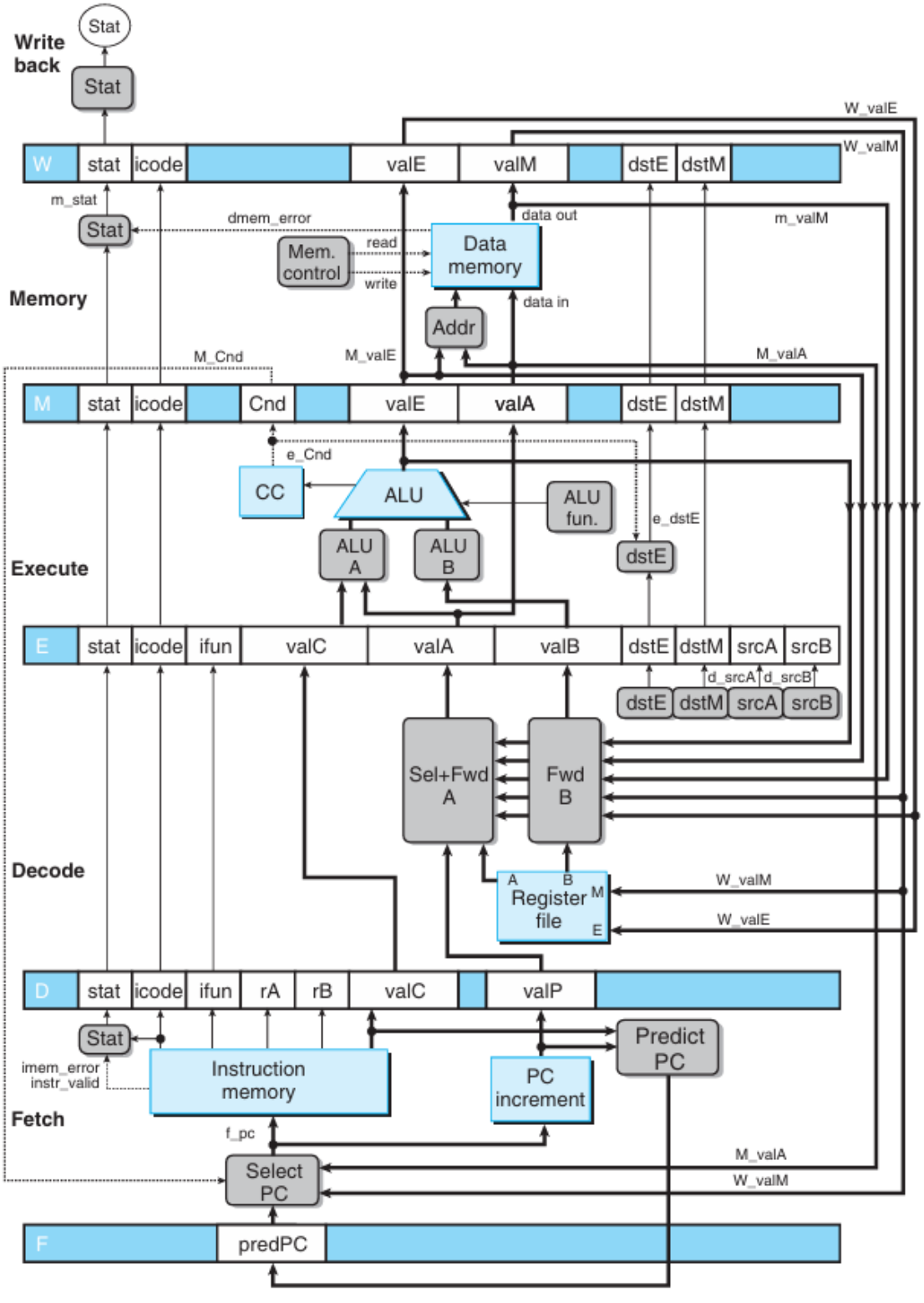
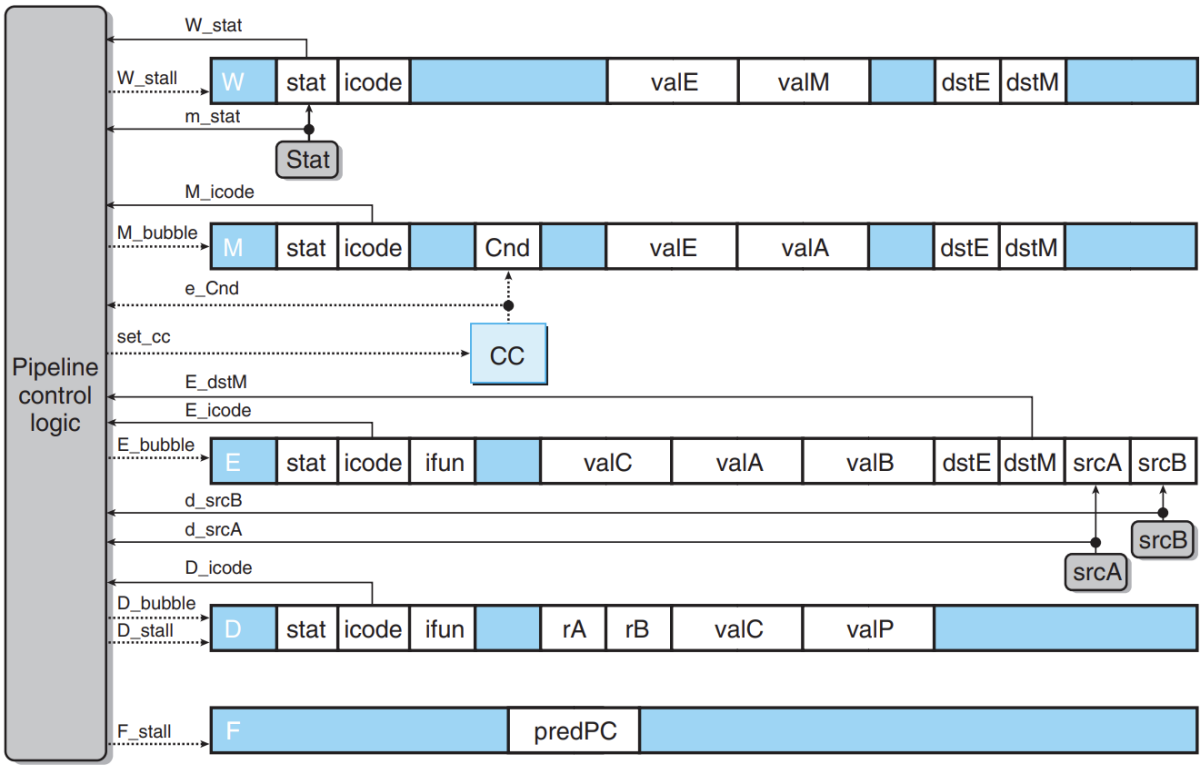


Figure – Hardware Structure of Pipeline Implementation



Pipeline implementation is a better version of sequential version of Y86-64 processor which uses the same set of instruction encodings and hardware is used. Pipeline implementation is done to increase the throughput of our processor which had space to improve since only one instruction was being carried out in each clock cycle and the hardware parts were also idle for most of the clock cycle.
The naming convention used is we have J_variable for variables which have been calculated in stages before J and j_variable for variables which are calculated in the stage J.
Now in pipeline implementation to avoid data and control hazards, bubble and stall have been implemented along with forward which gets the values to be used in decode stage directly from next stages without write back completion.

PC Selection and Fetch Stage

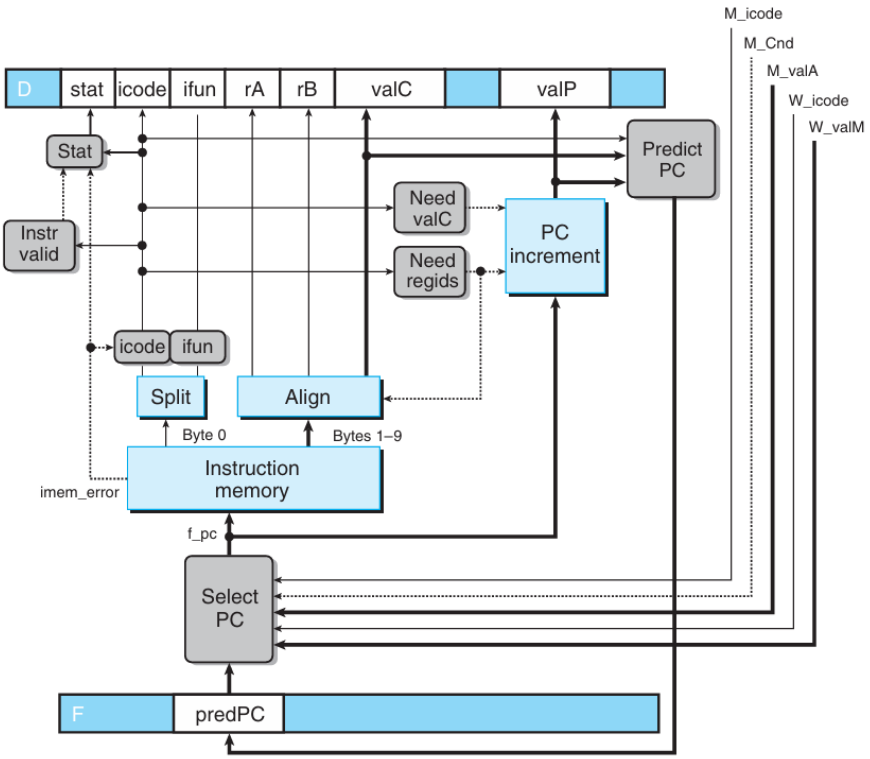


Figure – PC Selection and Fetch Stage

As shown in Figure above f_pc is the address of the starting of current instruction. The logic for the f_pc is as follows.
f_pc will be M_valA when M_icode is 7 and M_Cnd is 0, W_valM when W_icode is 9 and F_predPC for all other instructions.
f_predPC is the predicted address of the next instruction which goes to F_predPC at the positive edge of the clock.
f_predPC will be f_valC when f_icode in {7,8}, f_valP otherwise.
Fetch works as it was working for sequential implementation.

Source Code

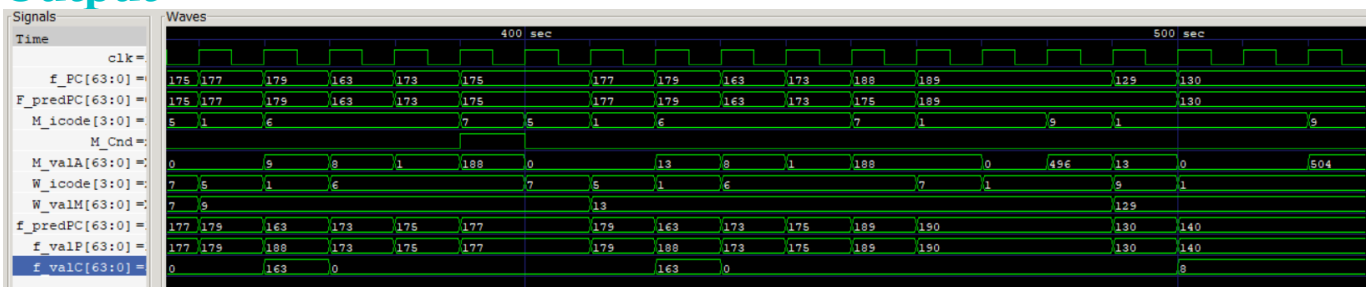
```
`include "instructionMemory.v"
`define IHALT 0
`define INOP 1
`define IRRMOVQ 2
`define IIRMOVQ 3
`define IRMMOVQ 4
`define IMRMOVQ 5
`define IOPQ 6
`define IJXX 7
`define ICALL 8
`define IRET 9
`define IPUSHQ 10
```

```
`define IPOPQ 11
`define FNONE 0
`define RESP 4
`define RNONE 15
`define ALUADD 0
`define SAOK 1
`define SADR 2
`define SINS 3
`define SHLT 4


module
fetch(f_stat,f_icode,f_ifun,f_rA,f_rB,f_valC,f_valP,f_predPC,F_predPC,M_icode,M_Cnd,M_valA,W_
icode,W_valM,clk);
    output reg [2:0] f_stat;
    output reg [3:0] f_icode;
    output reg [3:0] f_ifun;
    output reg [3:0] f_rA;
    output reg [3:0] f_rB;
    output reg [63:0] f_valC;
    output reg [63:0] f_valP;
    output reg [63:0] f_predPC;
    input [63:0] F_predPC;
    input [3:0] M_icode;
    input M_Cnd;
    input [63:0] M_valA;
    input [3:0] W_icode;
    input [63:0] W_valM;
    input clk;
    reg need_regids;
    reg need_valC;
    wire [7:0] im_out [0:9];
    wire imem_errorw;
    reg [63:0] f_PC;
    initial
    begin
        f_valP = 0;
        f_PC = 0;
        f_stat = 1;
    end

    instructionMemory
X1(im_out[0],im_out[1],im_out[2],im_out[3],im_out[4],im_out[5],im_out[6],im_out[7],im_out[8],
im_out[9],imem_errorw,f_PC);

    // Select PC Logic
    always @(*)
    begin
```

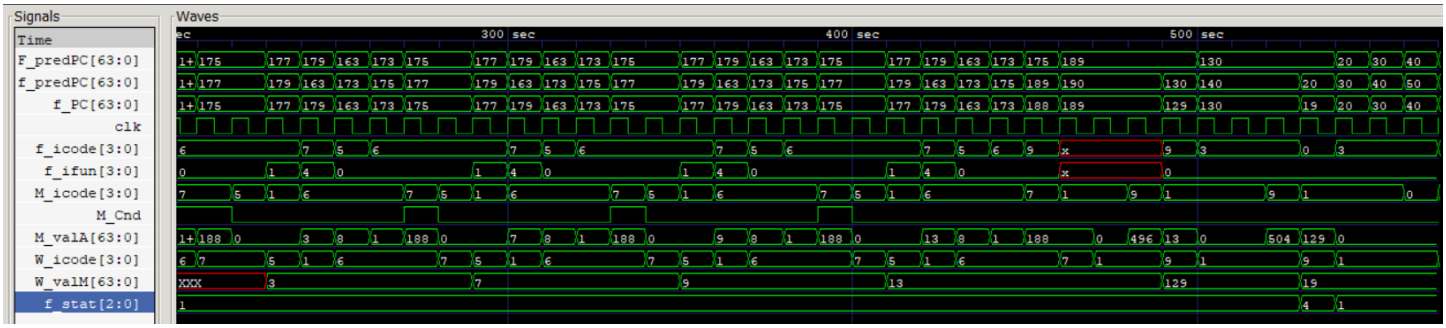
```
        if (M_icode == `IJXX && M_Cnd == 0)
        begin
            f_PC = M_valA;
        end
        else if (W_icode == `IRET)
        begin
            f_PC = W_valM;
        end
        else
        begin
            f_PC = F_predPC;
        end
    end

    always @(*)
    begin
        f_icode = im_out[0][7:4];
        f_ifun = im_out[0][3:0];

        if (f_icode == `IRRMOVQ || f_icode == `IIRMOVQ || f_icode == `IRMMOVQ || f_icode ==
`IMRMOVQ || f_icode == `IOPQ || f_icode == `IPUSHQ || f_icode == `IPOPQ)
        begin
            need_regids = 1;
            f_rA = im_out[1][7:4];
            f_rB = im_out[1][3:0];
        end
        else
        begin
            need_regids = 0;
            f_rA = `RNONE;
            f_rB = `RNONE;
        end

        if (f_icode == `IIRMOVQ || f_icode == `IRMMOVQ || f_icode == `IMRMOVQ || f_icode ==
`IJXX || f_icode == `ICALL)
        begin
            need_valC = 1;
            f_valC[7:0] = im_out[need_regids+1];
            f_valC[15:8] = im_out[need_regids+2];
            f_valC[23:16] = im_out[need_regids+3];
            f_valC[31:24] = im_out[need_regids+4];
            f_valC[39:32] = im_out[need_regids+5];
            f_valC[47:40] = im_out[need_regids+6];
            f_valC[55:48] = im_out[need_regids+7];
            f_valC[63:56] = im_out[need_regids+8];
        end
        else
        begin
```

```
            need_valC = 0;
        end

        // PC Increment
        f_valP = f_PC + 1 + need_regids + 8*need_valC;

        // Predict PC logic
        if (f_icode == `IJXX || f_icode == `ICALL)
        begin
            f_predPC = f_valC;
        end
        else
        begin
            f_predPC = f_valP;
        end

        // Stat logic
        if(f_icode < `IHALT || f_icode > `IPOPQ)
        begin
            f_stat = `SINS;
        end
        else if (imem_errorw == 1)
        begin
            f_stat = `SADR;
        end
        else
        begin
            f_stat = `SAOK;
        end

        if (f_icode == `IHALT)
        begin
            f_stat = `SHLT;
        end
    end
endmodule
```

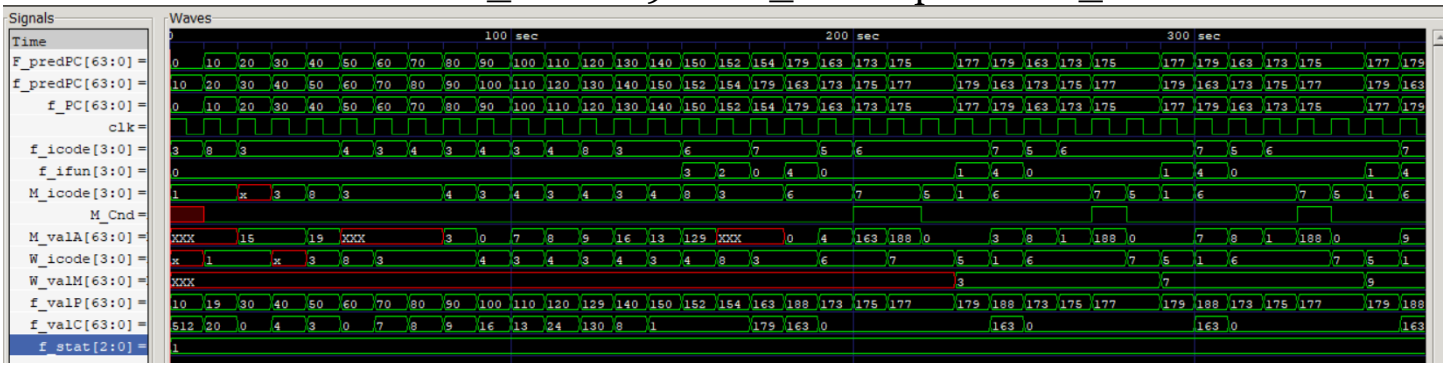
Output



Here we can see that when M_icode is 7 and M_Cnd is 0 then the f_PC is equals to M_valA.



Here we can see that when W_icode is 9 then f_PC is equals to W_valM.



Here we can see that when f_icode is in {7,8} then the f_predPC is equals to f_valC and f_valP otherwise.

In all the above cases at the positive edge of the clock the value of the F_predPC is updated to the value equal to f_predPC.

Decode and Write Back

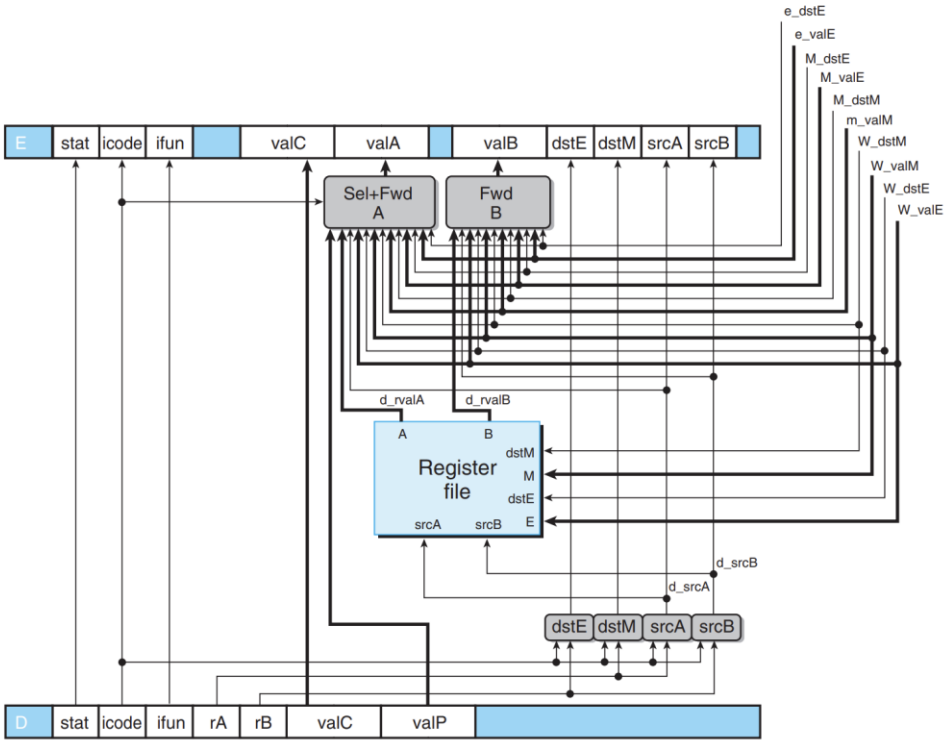


Figure – Decode and Write Back Stage

Decode and write back stage takes input from the Decode register, Write Back register, and execute, memory and write back stage to handle load-use data hazard.
Now, in the upcoming stages we either need valP or valA so based on the icode value we assign E_valA to be d_valA or d_valP.
Now for decoding again we decide the value of srcA, srcB based on the value of D_icode.
Also, here we have used forwarding logic for taking care of load use hazard since for the condition we need a value in register which have not been written back yet we need to use that value of register from the further stages itself before it has been written back in register file.
Now for forwarding logic in case dstE or dstM is same as srcA or srcB we take that value directly from the upcoming stages. The preference is based on which stage comes first so that we don't take the value which is being rewritten in next instruction cycle.
Write-back stage also takes place in this module itself since it also needs access to register file. The dstE and dstM values are calculated based on present value of W_icode and then write back takes place.

Source Code

```
`define IHALT 0
`define INOP 1
`define IRRMOVQ 2
`define IIRMOVQ 3
`define IRMMOVQ 4
`define IMRMOVQ 5
`define IOPQ 6
`define IJXX 7
`define ICALL 8
`define IRET 9
`define IPUSHQ 10
`define IPOPQ 11
`define FNONE 0
`define RESP 4
`define RNONE 15
`define ALUADD 0
`define SAOK 1
`define SADR 2
`define SINS 3
`define SHLT 4

module decode_writeBack(d_stat, d_icode, d_ifun, d_valC, d_valA, d_valB ,d_dstE, d_dstM,
d_srcA, d_srcB, D_stat, D_icode, D_ifun, D_rA, D_rB, D_valC, D_valP, e_dstE, e_valE, M_dstE,
M_valE, M_dstM, m_valM, W_dstM, W_valM, W_dstE, W_valE, clk);
    output reg [2:0] d_stat;
    output reg [3:0] d_icode;
    output reg [3:0] d_ifun;
    output reg [63:0] d_valC;
    output reg [63:0] d_valA;
    output reg [63:0] d_valB;
    output reg [3:0] d_dstE;
    output reg [3:0] d_dstM;
    output reg [3:0] d_srcA;
```

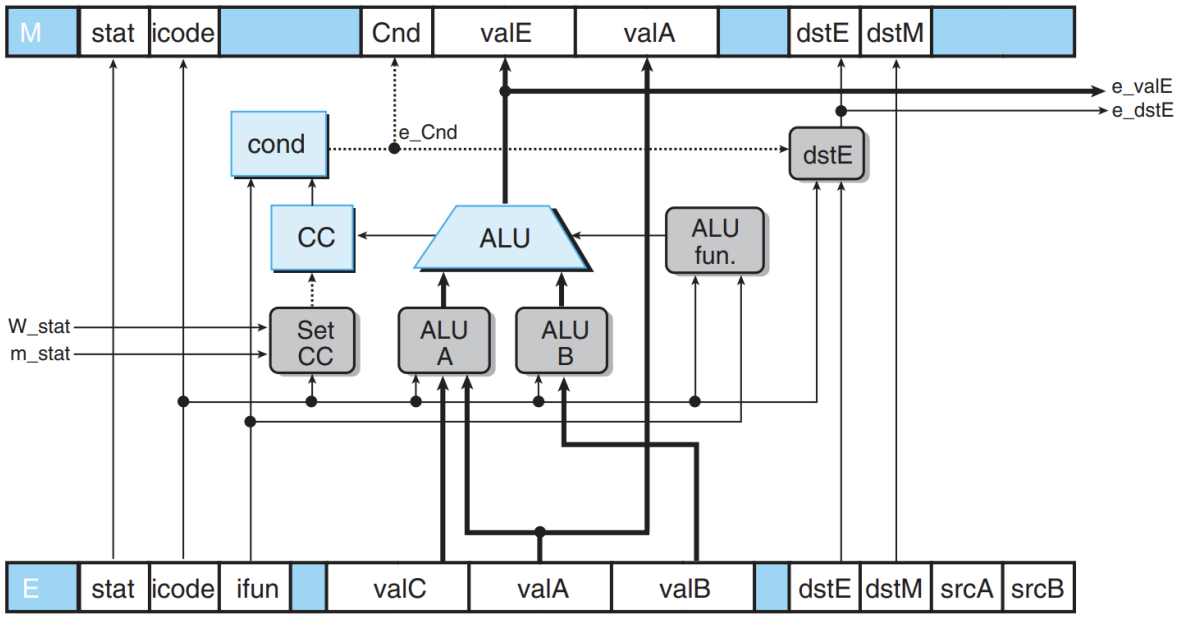
```
    output reg [3:0] d_srcB;
    input [2:0] D_stat;
    input [3:0] D_icode;
    input [3:0] D_ifun;
    input [3:0] D_rA;
    input [3:0] D_rB;
    input [63:0] D_valC;
    input [63:0] D_valP;
    input [3:0] e_dstE;
    input [63:0] e_valE;
    input [3:0] M_dstE;
    input [63:0] M_valE;
    input [3:0] M_dstM;
    input [63:0] m_valM;
    input [3:0] W_dstM;
    input [63:0] W_valM;
    input [3:0] W_dstE;
    input [63:0] W_valE;
    input clk;
    reg [63:0] d_rvalA;
    reg [63:0] d_rvalB;

    reg [63:0] reg_file[15:0];
    genvar i;
    generate
        for (i=0; i<16;i = i+1)begin
            initial begin
                reg_file[i] = i;
            end
        end
    endgenerate

    always@(*)
    begin
        d_stat = D_stat;
        d_icode = D_icode;
        d_ifun = D_ifun;
        d_valC = D_valC;
        if(D_icode == `IRRMOVQ || D_icode ==`IRMMOVQ || D_icode == `IOPQ || D_icode ==
`IPUSHQ)
            begin
                d_srcA = D_rA;
            end
        else if(D_icode == `IPOPQ || D_icode == `IRET)
            begin
                d_srcA = `RESP;
            end
        else
```

```
            begin
                d_srcA = `RNONE;
            end
        if(D_icode == `IRMMOVQ || D_icode == `IMRMOVQ || D_icode == `IOPQ)
            begin
                d_srcB = D_rB;
            end
        else if(D_icode == `IPUSHQ || D_icode == `IPOPQ || D_icode == `ICALL || D_icode ==
`IRET)
            begin
                d_srcB = `RESP;
            end
        else
            begin
                d_srcB = `RNONE;
            end
        d_rvalA = reg_file[d_srcA];
        d_rvalB = reg_file[d_srcB];

        if(D_icode == `ICALL || D_icode == `IJXX)
        begin
            d_valA = D_valP;
        end
        else if(d_srcA == e_dstE)
        begin
            d_valA = e_valE;
        end
        else if(d_srcA == M_dstM)
        begin
            d_valA = m_valM;
        end
        else if(d_srcA == M_dstE)
        begin
            d_valA = M_valE;
        end
        else if(d_srcA == W_dstM)
        begin
            d_valA = W_valM;
        end
        else if(d_srcA == W_dstE)
        begin
            d_valA = W_valE;
        end
        else
        begin
            d_valA = d_rvalA;
        end
```

```
        if(d_srcB == e_dstE)
        begin
            d_valB = e_valE;
        end
        else if(d_srcB == M_dstM)
        begin
            d_valB = m_valM;
        end
        else if(d_srcB == M_dstE)
        begin
            d_valB = M_valE;
        end
        else if(d_srcB == W_dstM)
        begin
            d_valB = W_valM;
        end
        else if(d_srcB == W_dstE)
        begin
            d_valB = W_valE;
        end
        else
        begin
            d_valB = d_rvalB;
        end
    end

    always @(*)
    begin
        if(D_icode == `IOPQ || D_icode == `IIRMOVQ)
        begin
            d_dstE = D_rB;
        end
        else if(D_icode == `IRRMOVQ)
        begin
            d_dstE = D_rB;
        end
        else if(D_icode == `IPUSHQ || D_icode == `IPOPQ || D_icode == `ICALL || D_icode ==
`IRET)
        begin
            d_dstE = `RESP;
        end
        else
        begin
            d_dstE = `RNONE;
        end
        if(D_icode == `IMRMOVQ || D_icode == `IPOPQ)
        begin
            d_dstM = D_rA;
```

```
        end
        else
        begin
            d_dstM = `RNONE;
        end
        reg_file[W_dstE] = W_valE;
        reg_file[W_dstM] = W_valM;
    end

    initial
    begin
        $monitor("rax = %d ,rcx = %d, rdx = %d, rbx = %d, rsp = %d, rbp = %d, rsi = %d, rdi =
%d, r8 = %d, r9 = %d, r10 = %d, r11 = %d, r12 = %d, r13 = %d, r14 = %d, r15 =
%d",reg_file[0],reg_file[1],reg_file[2],reg_file[3],reg_file[4],reg_file[5],reg_file[6],reg_f
ile[7],reg_file[8],reg_file[9],reg_file[10],reg_file[11],reg_file[12],reg_file[13],reg_file[1
4], reg_file[15]);
    end

endmodule
```

Execute



In case of execute, we perform the operation as per the value of E_icode, also the value of e_cnd is calculated to take care of conditional jumps and conditional moves.
ALU fun is decided based on the value of ifun in case of icode = 6(OPq) and otherwise we add or subtract based on the requirement of that specific E_icode.
All the flags are decided based on the outputs of ALU block and thus based on the flag values being 1 or 0 we decide if cnd = 1 or 0.

Also if cnd is 0 in case of cmovXX, e_dstE is assigned F so that the transfer does not take place. In case of cnd is 1, and icode is 2(cmovXX) e_dstE is assigned the value of destination based on decode output.

Source Code

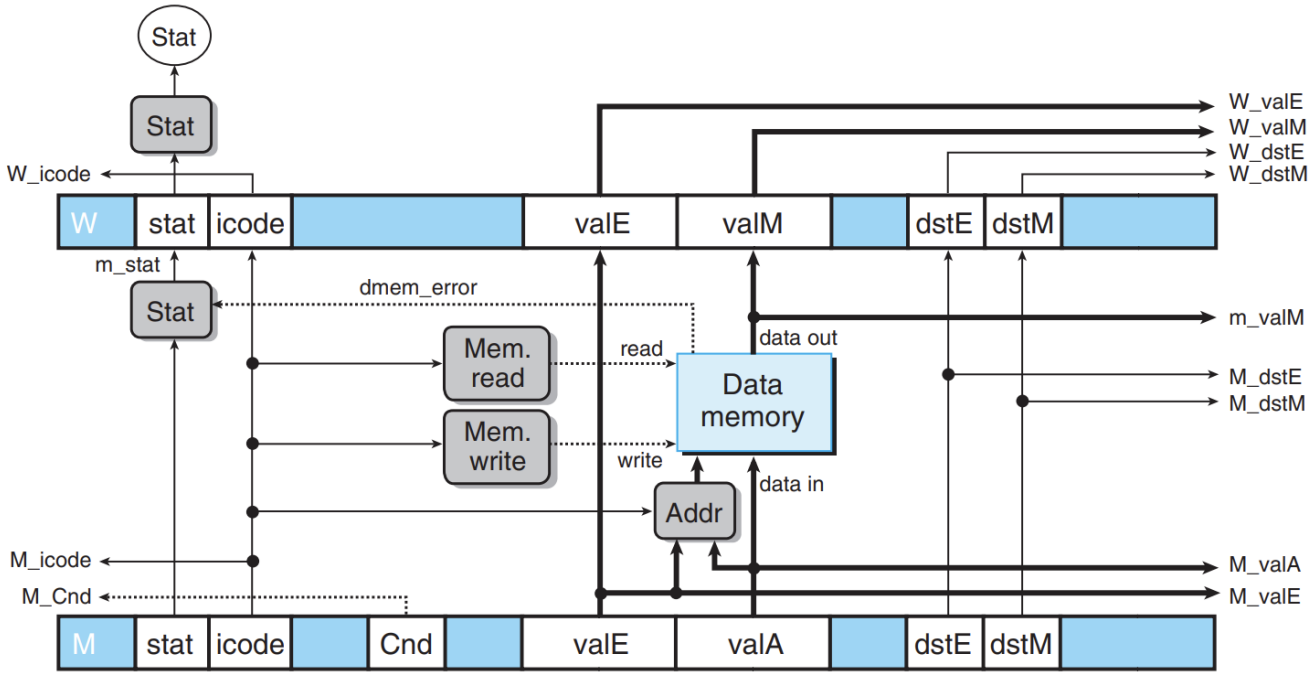
```
`include "./ALU/ALU_Wrapper.v"
`define IHALT 0
`define INOP 1
`define IRRMOVQ 2
`define IIRMOVQ 3
`define IRMMOVQ 4
`define IMRMOVQ 5
`define IOPQ 6
`define IJXX 7
`define ICALL 8
`define IRET 9
`define IPUSHQ 10
`define IPOPQ 11
`define FNONE 0
`define RESP 4
`define RNONE 15
`define ALUADD 0
`define SAOK 1
`define SADR 2
`define SINS 3
`define SHLT 4

module
execute(e_stat,e_icode,e_Cnd,e_valE,e_valA,e_dstE,e_dstM,E_stat,E_icode,E_ifun,E_valC,E_valA,
E_valB,E_dstE,E_dstM,E_srcA,E_srcB,set_CC,clk);
    output reg [2:0] e_stat;
    output reg [3:0] e_icode;
    output reg e_Cnd;
    output reg [63:0] e_valE;
    output reg [63:0] e_valA;
    output reg [3:0] e_dstE;
    output reg [3:0] e_dstM;
    input [2:0] E_stat;
    input [3:0] E_icode;
    input [3:0] E_ifun;
    input [63:0] E_valC;
    input [63:0] E_valA;
    input [63:0] E_valB;
    input [3:0] E_dstE;
    input [3:0] E_dstM;
    input [3:0] E_srcA;
    input [3:0] E_srcB;
```

```
    input set_CC;
    input clk;
    reg ZF;
    reg SF;
    reg OF;
    wire [63:0] aluOut;
    wire aluOF;
    reg [63:0] aluA;
    reg [63:0] aluB;
    reg [1:0] aluFun;
    ALU_Wrapper X1(aluOF,aluOut,aluFun,aluB, aluA);

    initial begin
        OF = 0;
        ZF = 0;
        SF = 0;
    end


    always @(*)
    begin
        e_icode = E_icode;
        e_stat = E_stat;
        e_dstE = E_dstE;
        e_dstM = E_dstM;
        e_valA = E_valA;
        if (E_icode == `IRRMOVQ || E_icode == `IOPQ)
        begin
            aluA = E_valA;
        end
        else if (E_icode == `IIRMOVQ || E_icode == `IRMMOVQ || E_icode == `IMRMOVQ)
        begin
            aluA  = E_valC;
        end
        else if (E_icode == `ICALL || E_icode == `IPUSHQ)
        begin
            aluA = -8;
        end
        else if (E_icode == `IRET || E_icode == `IPOPQ)
        begin
            aluA = 8;
        end
        else
        begin
            aluA = 0;
        end
```

```
        if (E_icode == `IRMMOVQ || E_icode == `IMRMOVQ || E_icode == `IOPQ || E_icode ==
`ICALL || E_icode == `IRET || E_icode == `IPUSHQ || E_icode == `IPOPQ)
        begin
            aluB = E_valB;
        end
        else if (E_icode == `IRRMOVQ || E_icode == `IIRMOVQ)
        begin
            aluB = 0;
        end
        else
        begin
            aluB = 0;
        end

        if (E_icode == `IOPQ)
        begin
            aluFun = E_ifun[1:0];
        end
        else
        begin
            aluFun = `ALUADD;
        end

        e_valE = aluOut;

        if (set_CC==1)
        begin
            OF = aluOF;
            if (e_valE[63] == 1)
            begin
                SF = 1;
            end
            else
            begin
                SF = 0;
            end

            if (e_valE == 0)
            begin
                ZF = 1;
            end
            else
            begin
                ZF = 0;
            end
        end

        if (E_icode == `IRRMOVQ || E_icode == `IJXX)
```

```
        begin
            if (E_ifun == 0)
            begin
                e_Cnd = 1;
            end
            else if (E_ifun == 1)
            begin
                e_Cnd = (SF^OF)|ZF;
            end
            else if (E_ifun == 2)
            begin
                e_Cnd  = SF^OF;
            end
            else if (E_ifun == 3)
            begin
                e_Cnd = ZF;
            end
            else if (E_ifun == 4)
            begin
                e_Cnd = ~ZF;
            end
            else if (E_ifun == 5)
            begin
                e_Cnd = ~(SF^OF);
            end
            else if (E_ifun == 6)
            begin
                e_Cnd = ~(SF^OF) & ~ZF;
            end
        end
        else
        begin
            e_Cnd = 0;
        end

        if (E_icode == `IRRMOVQ && e_Cnd != 1)
        begin
            e_dstE = `RNONE;
        end
    end
endmodule
```

Data Memory



Data memory module consists of the hardware part memory which is the main storage unit of our processor. Here, we calculate the mem_add value and check if it is within bounds of the data memory and thus a new memory address exception may occur so we update the m_stat in case there is data memory address error.

Now we check if the M_icode value corresponds to a read or write operation and thus based on it we calculate if mem_read and mem_write are 1 or 0.

In case of mem_read = 1, we move that value into valM from the memory and in case mem_write is 1, we transfer valE into memory.

Source Code

```
`define IHALT 0
`define INOP 1
`define IRRMOVQ 2
`define IIRMOVQ 3
`define IRMMOVQ 4
`define IMRMOVQ 5
`define IOPQ 6
`define IJXX 7
`define ICALL 8
`define IRET 9
`define IPUSHQ 10
`define IPOPQ 11
`define FNONE 0
`define RESP 4
`define RNONE 15
`define ALUADD 0
`define SAOK 1
`define SADR 2
```

```
`define SINS 3
`define SHLT 4

module data_memory(m_stat, m_icode, m_dstE, m_dstM, m_valE, m_valM, M_stat, M_icode, M_Cnd,
M_valE, M_valA, M_dstE, M_dstM, clk);
    output reg [2:0] m_stat;
    output reg [3:0] m_icode;
    output reg [63:0] m_valE;
    output reg [63:0] m_valM;
    output reg [3:0] m_dstE;
    output reg [3:0] m_dstM;
    input [2:0] M_stat;
    input [3:0] M_icode;
    input M_Cnd;
    input [63:0] M_valE;
    input [63:0] M_valA;
    input [3:0] M_dstE;
    input [3:0] M_dstM;
    input clk;
    reg [63:0] memReg[0:8191];
    genvar i;
    generate
        for (i=0; i<200;i = i+1)begin
            initial begin
                memReg[i] = 0;
            end
        end
    endgenerate
    reg [63:0] mem_addr;
    reg [63:0] mem_data;
    reg mem_read;
    reg mem_write;
    always @(*)
    begin
        m_valE = M_valE;
        m_icode = M_icode;
        m_dstE = M_dstE;
        m_dstM = M_dstM;
        if(M_icode == `IPUSHQ || M_icode == `ICALL || M_icode == `IRMMOVQ || M_icode ==
`IMRMOVQ)
        begin
            mem_addr = M_valE;
        end
        else if(M_icode == `IPOPQ || M_icode == `IRET)
        begin
            mem_addr = M_valA;
        end
```

```
        if (M_icode == `IRMMOVQ || M_icode == `IPUSHQ || M_icode == `ICALL)
        begin
            mem_data = M_valA;
        end

        if (M_icode == `IMRMOVQ || M_icode == `IRET || M_icode == `IPOPQ)
        begin
            mem_read = 1;
        end
        else
        begin
            mem_read = 0;
        end

        if (M_icode == `IRMMOVQ || M_icode == `ICALL || M_icode == `IPUSHQ)
        begin
            mem_write = 1;
        end
        else
        begin
            mem_write = 0;
        end

        if (mem_read == 1)
        begin
            m_valM = memReg[mem_addr];
        end

        if (mem_write == 1)
        begin
            memReg[mem_addr] = mem_data;
        end

        if(mem_addr > 8191)
        begin
            m_stat = `SADR;
        end
        else
        begin
            m_stat = M_stat;
        end
    end
endmodule
```

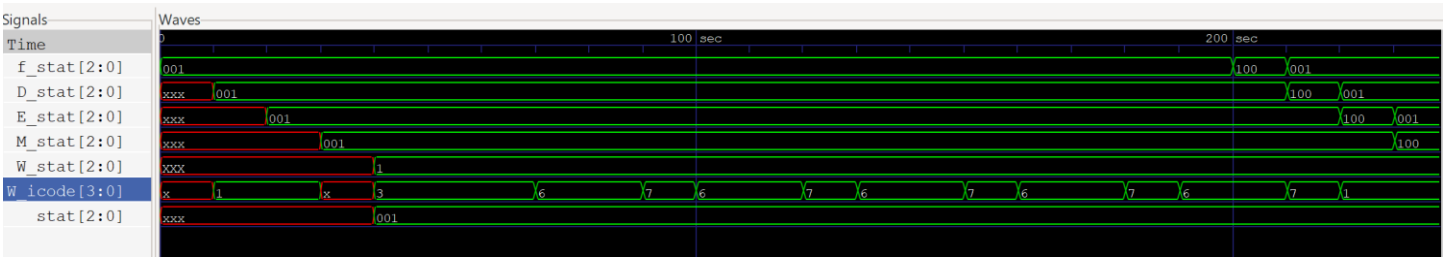
Write Back

In write-back stage, we transfer the value of W_stat into stat which is the state variable we use for operation of this processor since if we use stat from previous stages our operation might get halt or other exceptions before completing previous instructions.

Source Code

```
    always @(*)
    begin
        stat = W_stat;
        f_predPC = predPC;
    end

    always @(*)
    begin
        if (stat  != `SAOK)
        begin
            $finish;
        end
    end
```

PLOT:



As can be seen in above gtkwave plot, W_icode = 6 when f_stat becomes 4, so if we consider f_stat as our stat variable it will cause the OPq instruction to be incomplete. Thus we assign the value of W_icode as our main stat variable.

Pipeline Control

Now for handling different data hazard and control hazard we have to introduce bubbles and stall at various stages to make sure the smooth processing of our instructions.

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

Above are the various conditions that should be applied to each stage for taking care of these data hazards. These conditions are applied based on the icode values at different stages so that each case is taken care of.

Source Code

```
`define IHALT 0
`define INOP 1
`define IRRMOVQ 2
`define IIRMOVQ 3
`define IRMMOVQ 4
`define IMRMOVQ 5
`define IOPQ 6
`define IJXX 7
`define ICALL 8
`define IRET 9
`define IPUSHQ 10
`define IPOPQ 11
`define FNONE 0
`define RESP 4
`define RNONE 15
`define ALUADD 0
`define SAOK 1
`define SADR 2
`define SINS 3
`define SHLT 4

module pipelineControl(F_stall, D_stall, D_bubble, E_bubble, M_bubble, W_stall, set_CC,
D_icode, d_srcA, d_srcB, E_icode, E_dstM, e_Cnd, M_icode, m_stat, W_stat);
    output reg F_stall;
    output reg D_stall;
    output reg D_bubble;
    output reg E_bubble;
    output reg M_bubble;
    output reg W_stall;
    output reg set_CC;
    input [3:0] D_icode;
    input [3:0] d_srcA;
    input [3:0] d_srcB;
    input [3:0] E_icode;
    input [3:0] E_dstM;
    input e_Cnd;
    input [3:0] M_icode;
    input [2:0] m_stat;
    input [2:0] W_stat;

    always @(*)
    begin
        if (((E_icode == `IMRMOVQ || E_icode == `IPOPQ) && (E_dstM == d_srcA || E_dstM ==
d_srcB)) || (D_icode == `IRET || E_icode == `IRET || M_icode == `IRET))
        begin
            F_stall = 1;
```

```
        end
        else
        begin
            F_stall = 0;
        end

        if ((E_icode == `IMRMOVQ || E_icode == `IPOPQ) && (E_dstM == d_srcA || E_dstM ==
d_srcB))
        begin
            D_stall = 1;
        end
        else
        begin
            D_stall = 0;
        end


        if ((E_icode == `IJXX && e_Cnd == 0) || !((E_icode == `IMRMOVQ || E_icode == `IPOPQ)
&& (E_dstM == d_srcA || E_dstM == d_srcB)) && (D_icode == `IRET || E_icode == `IRET ||
M_icode == `IRET))
        begin
            D_bubble = 1;
        end
        else
        begin
            D_bubble = 0;
        end

        if ((E_icode == `IJXX && e_Cnd == 0) || ((E_icode == `IMRMOVQ || E_icode == `IPOPQ)
&& (E_dstM == d_srcA || E_dstM == d_srcB)))
        begin
            E_bubble = 1;
        end
        else
        begin
            E_bubble = 0;
        end

        if (m_stat == `SADR || m_stat == `SINS || m_stat == `SHLT || W_stat == `SADR ||
W_stat == `SINS || W_stat == `SHLT)
        begin
            M_bubble = 1;
        end
        else
        begin
            M_bubble = 0;
        end
```

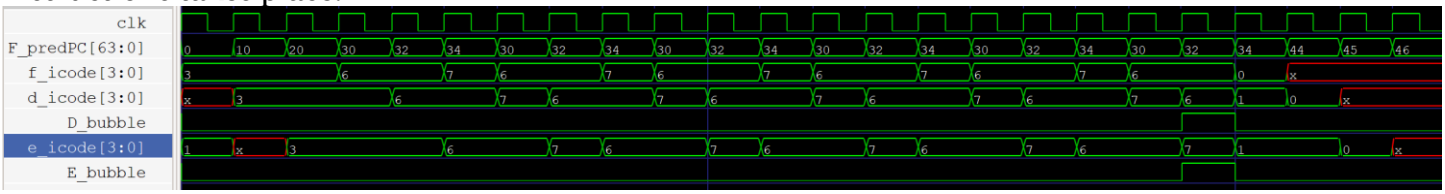
```
        if (W_stat == `SADR || W_stat == `SINS || W_stat == `SHLT)
        begin
            W_stall = 1;
        end
        else
        begin
            W_stall = 0;
        end

        if (E_icode == `IOPQ &&  !(m_stat == `SADR || m_stat == `SINS || m_stat == `SHLT ||
W_stat == `SADR || W_stat == `SINS || W_stat == `SHLT))
        begin
            set_CC = 1;
        end
        else
        begin
            set_CC = 0;
        end

    end

endmodule
```

Output

MISPREDICTED BRANCH:
Assembly Code

```
ps:
    irmovq $0, %rax
    irmovq $5, %rdi
    irmovq $1,%rsi
.L2:
    addq %rdi, %rax
    subq $rsi, %rdi
    jg .L2
halt
```

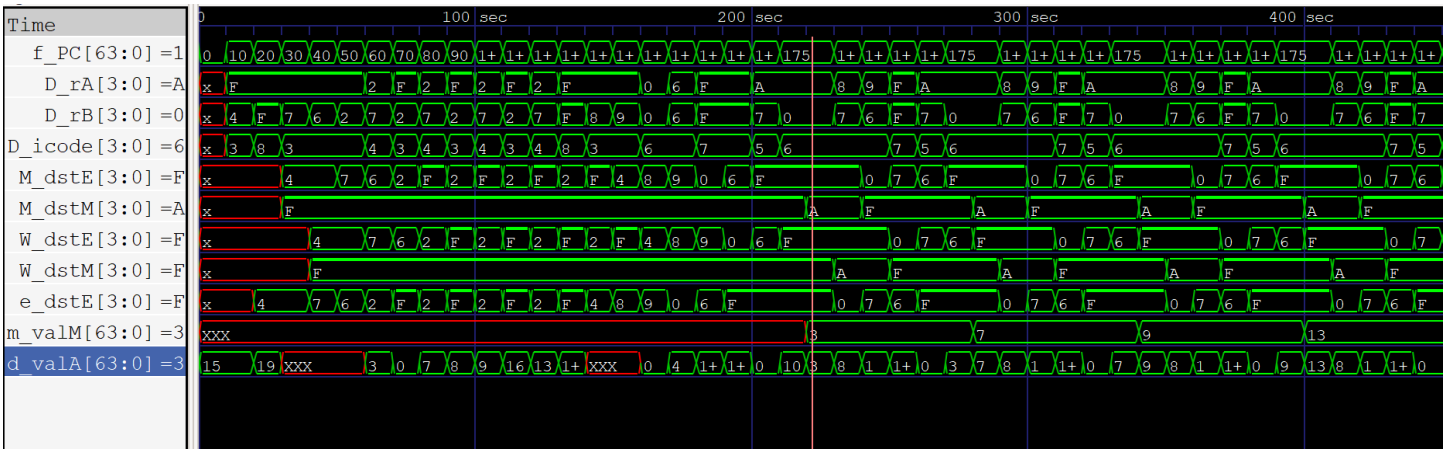
For above code sum of first n integers is calculated, and thus after the jump is called 5th time, we will have a mispredicted branch since the condition is satisfied only for first four times so that we revert back to starting, Now as can be seen in figure below at 5th jump we have introduced bubbles at Decode and Execute so that our cnd is calculated in Execute stage and the processor knows that the jump is mispredicted, and as can be seen below the program reaches to halt after jump and no further wrong instructions takes place.



LOAD USE HAZARD:
Assembly Code

```
0x0000:                         |     .pos 0
  0x0000: 30f40002000000000000 |     irmovq stack, %rsp # Set up stack pointer
  0x000a: 801400000000000000   |     call main # Execute main program
  0x0013: 00                   |     halt # Terminate program
                               |
  0x0014:                      | main:
  0x0014: 30f70000000000000000 |     irmovq $0,%rdi
  0x001e: 30f60400000000000000 |     irmovq $4,%rsi
  0x0028: 30f20300000000000000 |     irmovq $3,%rdx
  0x0032: 40270000000000000000 |     rmmovq %rdx,(%rdi)
  0x003c: 30f20700000000000000 |     irmovq $7,%rdx
  0x0046: 40270800000000000000 |     rmmovq %rdx,8(%rdi)
  0x0050: 30f20900000000000000 |     irmovq $9,%rdx
  0x005a: 40271000000000000000 |     rmmovq %rdx,16(%rdi)
  0x0064: 30f20d00000000000000 |     irmovq $13,%rdx
  0x006e: 40271800000000000000 |     rmmovq %rdx, 24(%rdi)
  0x0078: 808200000000000000   |     call sum # sum(array, 4)
  0x0081: 90                   |     ret
                               |
                               | # long sum(long *start, long count)
                               | # start in %rdi, count in %rsi
  0x0082:                      | sum:
  0x0082: 30f80800000000000000 |     irmovq $8,%r8 # Constant 8
  0x008c: 30f90100000000000000 |     irmovq $1,%r9 # Constant 1
  0x0096: 6300                 |     xorq %rax,%rax # sum = 0
  0x0098: 6266                 |     andq %rsi,%rsi # Set CC
  0x009a: 70b300000000000000   |     jmp test # Goto test
  0x00a3:                      | loop:
  0x00a3: 50a70000000000000000 |     mrmovq (%rdi),%r10 # Get *start
  0x00ad: 60a0                 |     addq %r10,%rax # Add to sum
  0x00af: 6087                 |     addq %r8,%rdi # start++
  0x00b1: 6196                 |     subq %r9,%rsi # count--. Set CC
  0x00b3:                      | test:
  0x00b3: 74a300000000000000   |     jne loop # Stop when 0
  0x00bc: 90                   |     ret # Return
                               |
                               |      # Stack starts here and grows to lower addresses
  0x00bd:                      |     .pos 0x200
  0x0200:                      | stack:
```

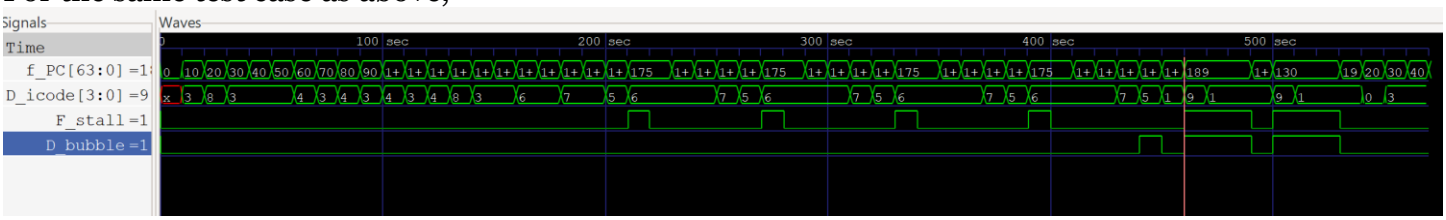
Now for above code there are multiple instances of mrmovq followed by OPq and thus there should be stall inserted in Fetch and Decode stage and bubble is introduced in Execute stage. As can be seen at the highlighted part in above plot of gtkwave.



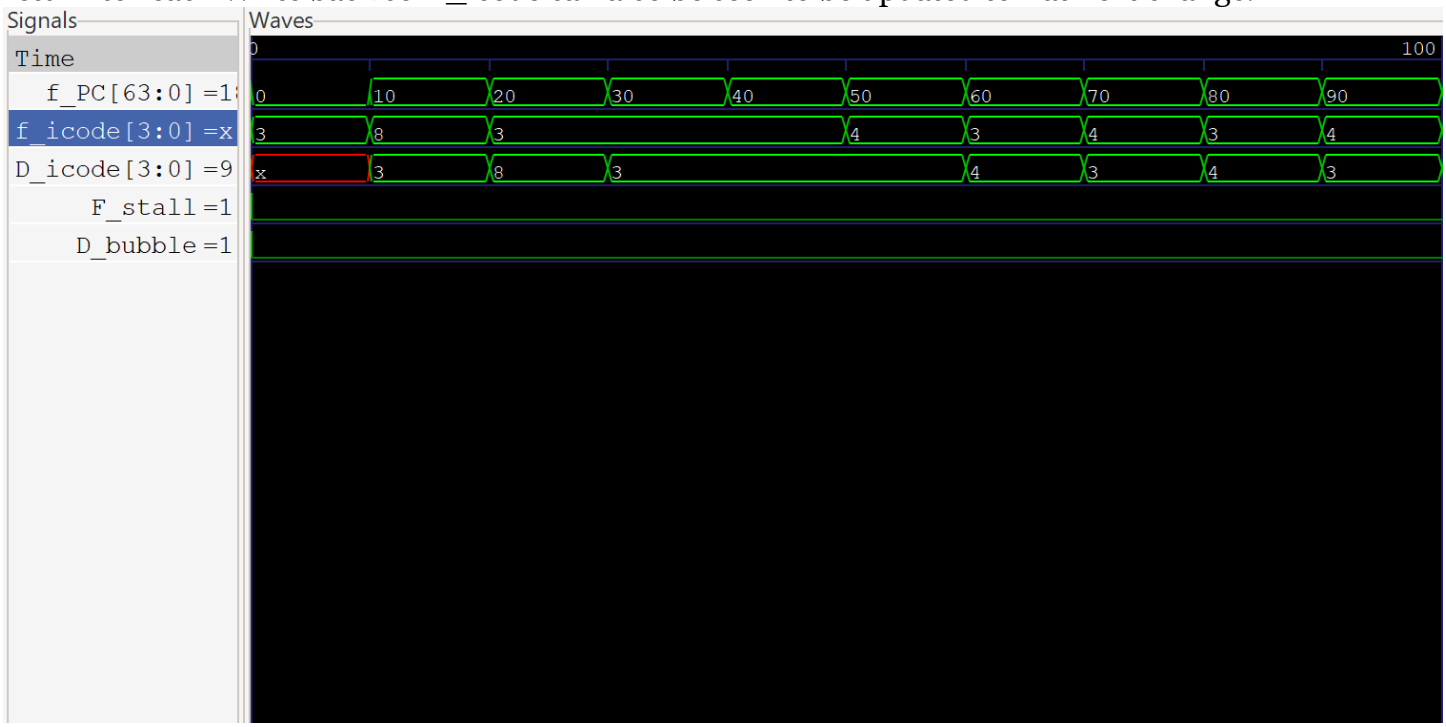
Now at the highlighted part in above figure, after implementing stall in fetch and decode and bubble in execute, m_valM contains the value needed in next operation and thus M_dstM is same as D_rA (A) which causes the value forwarded from memory to be used in the decode stage instead of written back value. Thus d_valA takes the same value as m_valM as required.

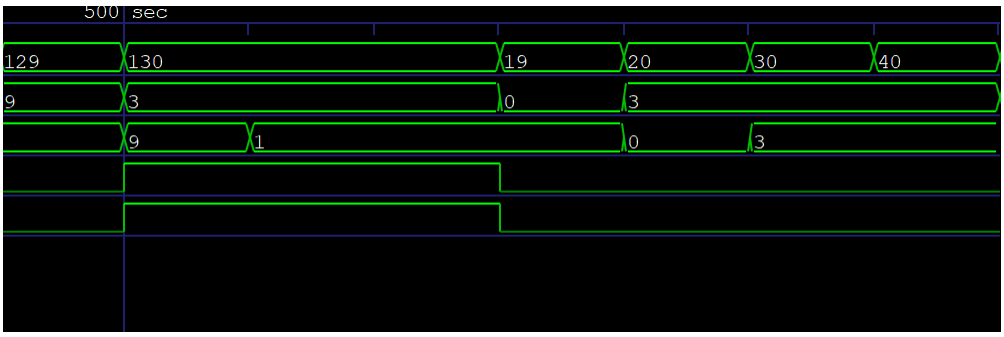
PROCESSING RETURN:

For the same test case as above,



Now as shown in the highlighted part, icode = 9(return) so we need to insert stall in fetch stage and bubble in Decode stage. Bubble introduced can also seen as we have updated icode of next instruction as 1 as practically what bubble does is introduce a nop in between the instructions to give time to return to reach Write back so D_icode can also be seen to be updated to 1 at next change.





Also the correctness can be seen above as we have a call operation at PC value 10 and thus return also causes PC to go back to 19(instruction next to call) after the current PC value.
Now since F_stall is 1 the next icode(3) remains in the fetch stage.