INTRO TO PROCESSOR ARCHITECTURE

Y-86 64 ARCHITECTURE

PROJECT REPORT

Ananya Vaibhavi (2021102003)

Arya Marda (2021102021)

Project Description:

This project involves the implementation of Y-86 64 processor architecture design using hardware coding language VERILOG.
Steps involved in building the project:

Step-1: Sequential implementation.

Step-2: Pipeline implementation.
The aim of the project is to build a 5-stage pipeline implementation of Y86-64 processor with features like forwarding, halt, stall instructions.

This report includes the sequential and pipeline implementation of the processor. The main blocks of the processor are Fetch logic, Decode logic, Execute logic, Memory Logic, Writeback stage. The sequential implementation also includes a pc-update stage which is taken out in pipeline implementation. The report also describes the test-bench implementation used as processor to run all the stages.

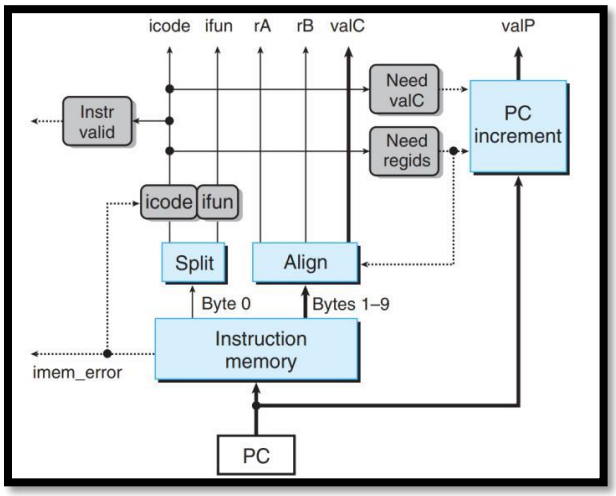
Sequential implementation:

The sequential implementation of the Y86-64 processor works with fetch, decode, execute, memory, writeback and PC update according to the following table:

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

The Blocks are as follows:

1. Fetch:



Processes in the fetch block:

1. PC register contains the current instruction to be executed from the instruction memory and passed to the instruction memory.

2. The instruction memory gives an imem_error if the PC value obtained is greater than the maximum instruction in instruction memory. In our project we have set our maximum number of instructions to 1024, i.e: PC can range from 0 to 1023. If PC is within instruction range memory passes 10 bytes of instruction further.

3. icode, ifun, valC, Instr_valid, valP are taken as output of this stage. These values are derived from the instruction passed from Instruction memory according to following rules:

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

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

1. **icode, ifun** defines the instructions to be performed.
2. **rA, rB** specifies the registers on which these instructions are being performed.
3. **valC** is initialized with V/d/Dest according to the icode's.
4. **Instr_error** occurs when icode or ifun are not in valid range. Rage of ifun is defined according to the current icode instruction.
5. If halt (icode = 0) is encountered the **HLT** flag is set to 1 and the processor stops running due to a $finish statement.

Fetch module:

```
module fetch(
    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 mem_error,
    output reg instr_error,
    input clk,
    input [63:0] PC,
    input [0:79] instr
);
```

Error update:

1. HLT:

```
always @(icode) begin
    if(icode==0 )
        $finish;
end
```

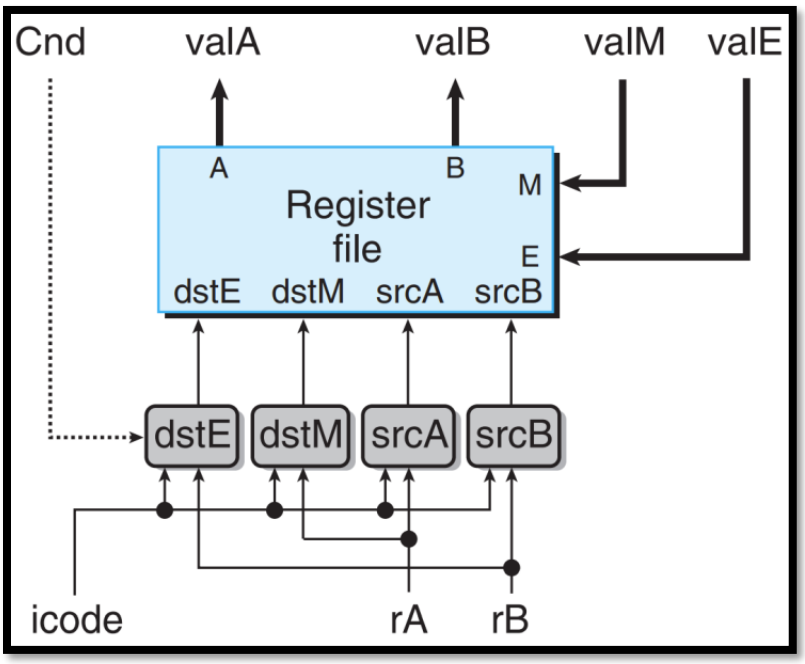
2. istr_error, mem_error:

```
always @(mem_error) begin
    if(mem_error == 1 || instr_err == 1)
        $monitor("Wrong instr add\n");
end
```

```
if (PC > 1023) begin
    mem_error = 1'b1;
end
else begin
    mem_error = 1'b0;
end
```

2. Decode:

The purpose of the decode block is to read from the registers and assign the values of **valA** and **valB.** For this we need a register file, conaining 15 registers which updates throughout the program's runtime.



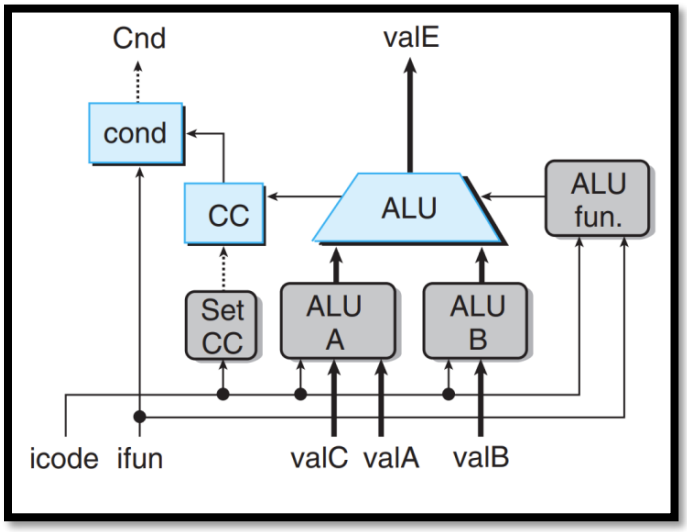
We have initialized these registers in the testbench as:

```
reg_file_in[0] = 64'd0;
reg_file_in[1] = 64'd1;
reg_file_in[2] = 64'd2;
reg_file_in[3] = 64'd3;
reg_file_in[4] = 64'd60;  //stack pointer
reg_file_in[5] = 64'd5;
reg_file_in[6] = 64'd6;
reg_file_in[7] = 64'd7;
reg_file_in[8] = 64'd8;
reg_file_in[9] = 64'd9;
reg_file_in[10] = 64'd10;
reg_file_in[11] = 64'd11;
reg_file_in[12] = 64'd12;
reg_file_in[13] = 64'd13;
reg_file_in[14] = 64'd14;
```

Decode module declaration:

```
module decode(clk,rA,rB,icode,reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14,valA,valB);
input clk;
input [3:0] rA,rB,icode;
input [63:0] reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14;
wire [63:0] reg_file_in [0:14];
output reg [63:0] valA,valB;

assign reg_file_in[0] = reg0;
assign reg_file_in[1] = reg1;
assign reg_file_in[2] = reg2;
assign reg_file_in[3] = reg3;
assign reg_file_in[4] = reg4;
assign reg_file_in[5] = reg5;
assign reg_file_in[6] = reg6;
assign reg_file_in[7] = reg7;
assign reg_file_in[8] = reg8;
assign reg_file_in[9] = reg9;
assign reg_file_in[10] = reg10;
assign reg_file_in[11] = reg11;
assign reg_file_in[12] = reg12;
assign reg_file_in[13] = reg13;
assign reg_file_in[14] = reg14;
```

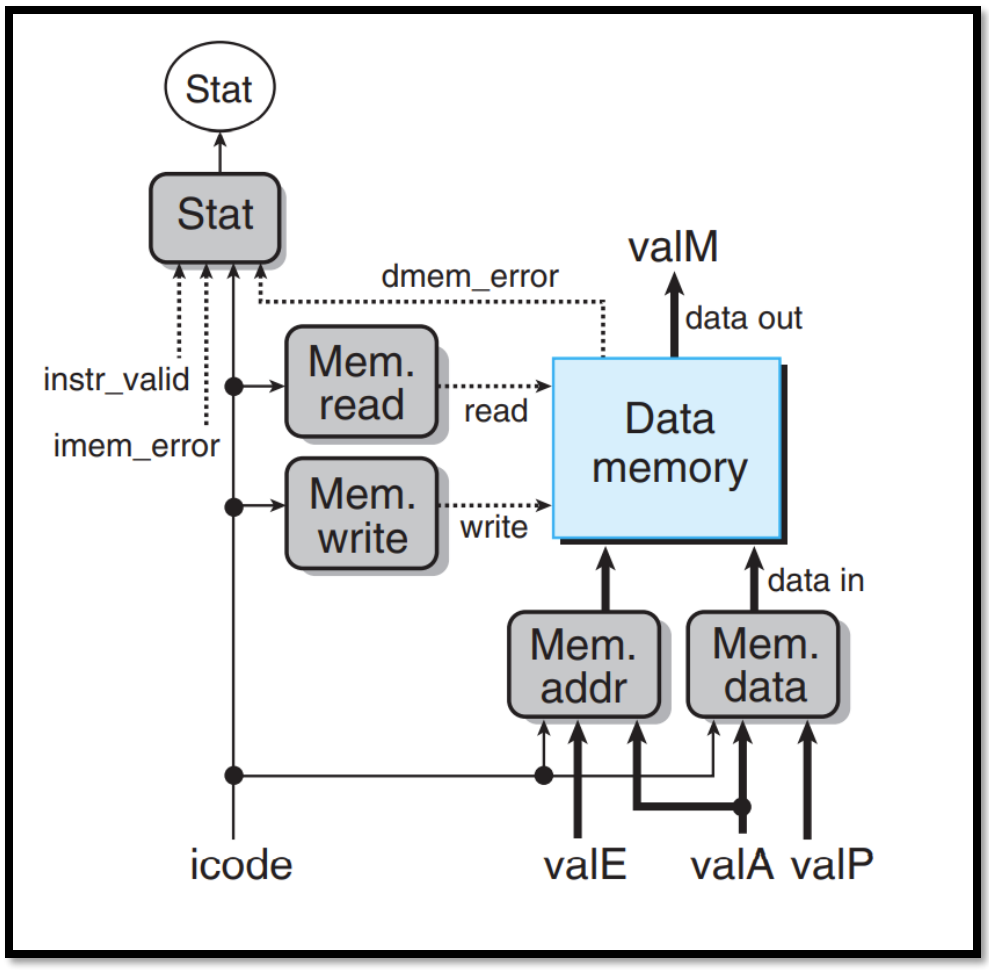
3. Execute



The execute stage has valA, valB, ifun, icode, valC as input's which are passed through 3 ALU blocks as shown in figure to compute **valE**, this usually refers to the effective address. Also, an important function of Execute stage is to set **Cnd.**
The execution of instructions is controlled by icode and ifun values.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

Execute module declarations:

4. Memory:



The memory stage in the sequential architecture of the Y86-64 processor is responsible for reading from and writing to memory. It retrieves the memory address from the previous stage and sends a request to the memory system to read or write data from or to that address.

If the request is a read, the memory stage retrieves the data from memory and passes it on to the next stage. If the request is a write, the memory stage stores the data to memory at the specified address.

Additionally, the memory stage also checks for memory-related errors, such as invalid memory accesses or page faults, and sets appropriate flags in the status register. The status register is then passed on to the next stage, where it can be used to make decisions or trigger interruptions.
We have made a temporary memory in our module, it can show the temporary changes that has been made in memory, also we have hard-coded the data_mem[valA] value for the pop and data_mem[valE] memory to register move.

This is done as:

```
data_mem[valA] = 8'd23; //for pop ooperation.
data_mem[valE] = 8'd200;
```

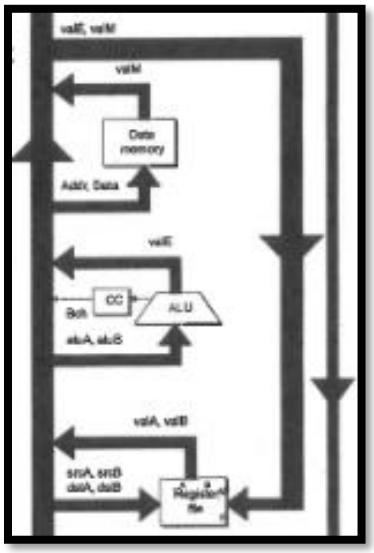
And Display statement:

```
always@(data_mem[valE])
  $display("mem_allocated->%b",data_mem[valE]);
```

Memory module declaration :

```
module memory(
  clk,icode,valA,valE,valP,valM,datamem
);
```

5. Write back

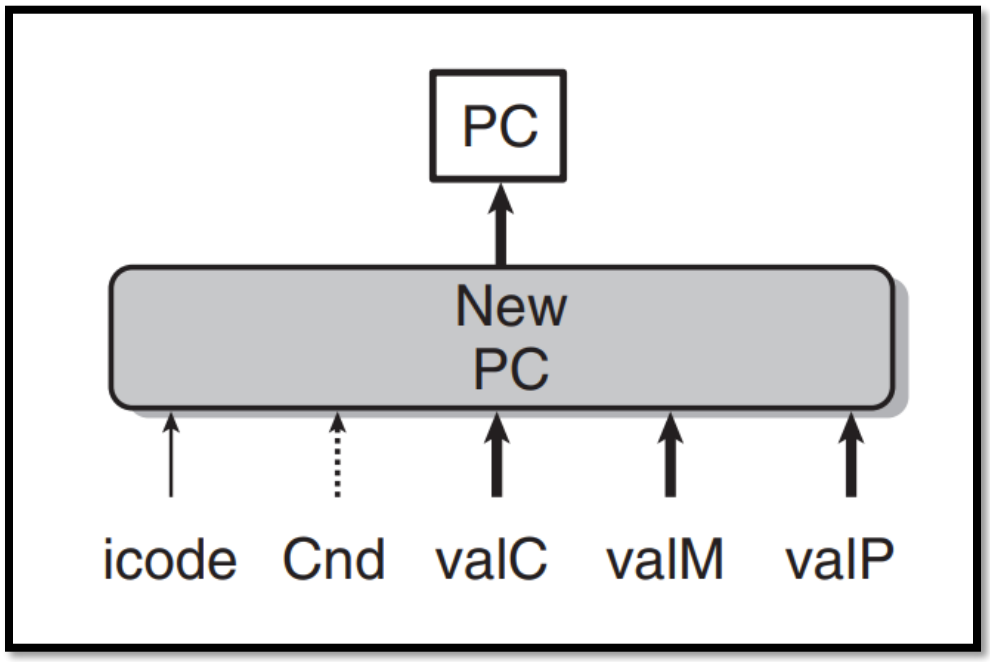


This stage updates register file corresponding tothe icode and ifun of the current instruction, the values of valE, valM are passed to the register file, the valE and valM are the final values of the updated register. Between ValE, ValM the register is updated accoring to the icode.

Writeback module declaration:

```
module write_back(clk,icode,ifun,rA,rB,valA,valB,valE,valM,
reg_in0,reg_in1,reg_in2,reg_in3,reg_in4,reg_in5,reg_in6,reg_in7,reg_in8,reg_in9,reg_in10,reg_in11,reg_in12,reg_in13,reg_in14,
reg_out0,reg_out1,reg_out2,reg_out3,reg_out4,reg_out5,reg_out6,reg_out7,reg_out8,reg_out9,reg_out10,reg_out11,reg_out12,
reg_out13,reg_out14
);
```

6. PC update



The purpose of PC update is to point to the address of the next instruction.

The PC Update block has icode, Cnd, valC, valM and valP as inputs and the value of updated PC as output.

The PC Update part works on switch statements which selects the instructions according to the values of icode and Cnd and updates the value of PC accordingly.

```
module pcupdate(clk,icode,cnd,valC,valM,valP,pcnxt);
```

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

Working of sequential implementation:

Execution of following instruction is shown below:

cmov0 1 3
irmov F 2 17
rmmov 5 2 1
mrmov 7 0 1
opq 0 2 3
cmov 3 4
cmovge 5 3
Halt

Testbench instructions:

```
  //cmovxx
   instr_mem[0]=8'b00010000;
   instr_mem[1]=8'b00100000; //2 fn
   instr_mem[2]=8'b00010011; //rA rB

 //irmovq
   instr_mem[3]=8'b00110000; //3 0
   instr_mem[4]=8'b00000010; //F rB
   instr_mem[5]=8'b00000000; //V
   instr_mem[6]=8'b00000000; //V
   instr_mem[7]=8'b00000000; //V
   instr_mem[8]=8'b00000000; //V
   instr_mem[9]=8'b00000000; //V
   instr_mem[10]=8'b00000000; //V
   instr_mem[11]=8'b00000000; //V
   instr_mem[12]=8'b00010001; //V=17

 //rmmovq
   instr_mem[13]=8'b01000000; //4 0
   instr_mem[14]=8'b01010010; //rA rB
   instr_mem[15]=8'b00000000; //D
   instr_mem[16]=8'b00000000; //D
   instr_mem[17]=8'b00000000; //D
   instr_mem[18]=8'b00000000; //D
   instr_mem[19]=8'b00000000; //D
   instr_mem[20]=8'b00000000; //D
   instr_mem[21]=8'b00000000; //D
   instr_mem[22]=8'b00000001; //D

 //mrmovq
   instr_mem[23]=8'b01010000; //5 0
   instr_mem[24]=8'b01110000; //rA rB
   instr_mem[25]=8'b00000000; //D
   instr_mem[26]=8'b00000000; //D
   instr_mem[27]=8'b00000000; //D
   instr_mem[28]=8'b00000000; //D
   instr_mem[29]=8'b00000000; //D
   instr_mem[30]=8'b00000000; //D
   instr_mem[31]=8'b00000000; //D
   instr_mem[32]=8'b00000001; //D

 //  OPq
   instr_mem[33]=8'b01100000; //6 fn
   instr_mem[34]=8'b00100011; //rA rB5

 // cmovxx
   instr_mem[35]=8'b00100000; //2 fn
   instr_mem[36]=8'b00110100; //rA rB

   instr_mem[37]=8'b00100101; // 2 ge
   instr_mem[38]=8'b01010011; // rA rB

 //halt
instr_mem[39]=8'b00000000; // 0 0
```

Output:

```
mem_allocated->01100100
                   0
clk:0
rA: x rB: x icode: 1 ifun: 0 valC:                  x valP:                  1 mem_error:0
 valA:                   x valB:                  x
cf_in:0 valE:                    0 cnd:0 cf_out:x
valM:                    x
r0:                     0
r1:                     1
 r2:                     2
 r3:                     3
 r4:                    60
 r5:                     5
 r6:                     6
 r7:                     7
 r8:                     8
 r9:                     9
 r10:                    10
 r11:                    11
 r12:                    12
 r13:                    13
 r14:                    14
PC:                     0

mem_allocated->xxxxxxxx
mem_allocated->11001000
mem_allocated->01100100
                   5
clk:1
rA: 1 rB: 3 icode: 2 ifun: 0 valC:                  x valP:                  3 mem_error:0
 valA:                   1 valB:                  0
cf_in:0 valE:                    1 cnd:1 cf_out:x
valM:                    x
r0:                     0
r1:                     1
 r2:                     2
 r3:                     3
 r4:                    60
 r5:                     5
 r6:                     6
 r7:                     7
 r8:                     8
 r9:                     9
 r10:                    10
 r11:                    11
 r12:                    12
 r13:                    13
 r14:                    14
PC:                     1

                  10
clk:0
rA: 1 rB: 3 icode: 2 ifun: 0 valC:                  x valP:                  3 mem_error:0
 valA:                   1 valB:                  0
cf_in:0 valE:                    1 cnd:1 cf_out:x
valM:                    x
r0:                     0
r1:                     1
 r2:                     2
 r3:                     1
 r4:                    60
 r5:                     5
 r6:                     6
 r7:                     7
 r8:                     8
 r9:                     9
 r10:                    10
 r11:                    11
 r12:                    12
 r13:                    13
 r14:                    14
PC:                     1

mem_allocated->xxxxxxxx
mem_allocated->01100100
```

```
                 15
clk:1
rA: 0 rB: 2 icode: 3 ifun: 0 valC:                  17 valP:                  13 mem_error:0
 valA:                   1 valB:                  0
cf_in:0 valE:                  17 cnd:0 cf_out:x
valM:                    x
r0:                     0
r1:                     1
 r2:                     2
 r3:                     1
 r4:                    60
 r5:                     5
 r6:                     6
 r7:                     7
 r8:                     8
 r9:                     9
 r10:                    10
 r11:                    11
 r12:                    12
 r13:                    13
 r14:                    14
PC:                     3

                 20
clk:0
rA: 0 rB: 2 icode: 3 ifun: 0 valC:                  17 valP:                  13 mem_error:0
 valA:                   1 valB:                  0
cf_in:0 valE:                  17 cnd:0 cf_out:x
valM:                    x
r0:                     0
r1:                     1
 r2:                    17
 r3:                     1
 r4:                    60
 r5:                     5
 r6:                     6
 r7:                     7
 r8:                     8
 r9:                     9
 r10:                    10
 r11:                    11
 r12:                    12
 r13:                    13
 r14:                    14
PC:                     3

mem_allocated->00000001
mem_allocated->00000101
mem_allocated->xxxxxxxx
mem_allocated->xxxxxxxx
mem_allocated->00000101
                 25
clk:1
rA: 5 rB: 2 icode: 4 ifun: 0 valC:                   1 valP:                  23 mem_error:0
 valA:                   5 valB:                 17
cf_in:0 valE:                  18 cnd:0 cf_out:x
valM:                    x
r0:                     0
r1:                     1
 r2:                    17
 r3:                     1
 r4:                    60
 r5:                     5
 r6:                     6
 r7:                     7
 r8:                     8
 r9:                     9
 r10:                    10
 r11:                    11
 r12:                    12
 r13:                    13
 r14:                    14
PC:                    13
```

```
                30
clk:0
rA: 5 rB: 2 icode: 4 ifun: 0 valC:                 1 valP:                23 mem_error:0
 valA:                  5 valB:                17
cf_in:0 valE:                18 cnd:0 cf_out:x
valM:                  x
r0:                   0
r1:                   1
 r2:                  17
 r3:                   1
 r4:                  60
 r5:                   5
 r6:                   6
 r7:                   7
 r8:                   8
 r9:                   9
 r10:                  10
 r11:                  11
 r12:                  12
 r13:                  13
 r14:                  14
PC:                  13

mem_allocated->01100100
mem_allocated->xxxxxxxx
mem_allocated->01100100
mem_allocated->11001000
mem_allocated->01100100
                35
clk:1
rA: 7 rB: 0 icode: 5 ifun: 0 valC:                 1 valP:                33 mem_error:0
 valA:                  5 valB:                0
cf_in:0 valE:                  1 cnd:0 cf_out:x
valM:                100
r0:                   0
r1:                   1
 r2:                  17
 r3:                   1
 r4:                  60
 r5:                   5
 r6:                   6
 r7:                   7
 r8:                   8
 r9:                   9
 r10:                  10
 r11:                  11
 r12:                  12
 r13:                  13
 r14:                  14
PC:                  23

                40
clk:0
rA: 7 rB: 0 icode: 5 ifun: 0 valC:                 1 valP:                33 mem_error:0
 valA:                  5 valB:                0
cf_in:0 valE:                  1 cnd:0 cf_out:x
valM:                100
r0:                   0
r1:                   1
 r2:                  17
 r3:                   1
 r4:                  60
 r5:                   5
 r6:                   6
 r7:                 100
 r8:                   8
 r9:                   9
 r10:                  10
 r11:                  11
 r12:                  12
 r13:                  13
 r14:                  14
PC:                  23

mem_allocated->11001000
mem_allocated->01100100
```

```
mem_allocated->01100100
                45
clk:1
rA: 2 rB: 3 icode: 6 ifun: 0 valC:                  1 valP:                 35 mem_error:0
 valA:                 17 valB:                1
cf_in:0 valE:                 18 cnd:0 cf_out:0
valM:                100
r0:                  0
r1:                  1
 r2:                 17
 r3:                  1
 r4:                 60
 r5:                  5
 r6:                  6
 r7:                100
 r8:                  8
 r9:                  9
 r10:                 10
 r11:                 11
 r12:                 12
 r13:                 13
 r14:                 14
PC:                  33

mem_allocated->xxxxxxxx
mem_allocated->xxxxxxxx
mem_allocated->01100100
                50
clk:0
rA: 2 rB: 3 icode: 6 ifun: 0 valC:                  1 valP:                 35 mem_error:0
 valA:                 17 valB:                18
cf_in:0 valE:                 35 cnd:0 cf_out:0
valM:                100
r0:                  0
r1:                  1
 r2:                 17
 r3:                 18
 r4:                 60
 r5:                  5
 r6:                  6
 r7:                100
 r8:                  8
 r9:                  9
 r10:                 10
 r11:                 11
 r12:                 12
 r13:                 13
 r14:                 14
PC:                  33

mem_allocated->xxxxxxxx
mem_allocated->01100100
mem_allocated->11001000
mem_allocated->01100100
                55
clk:1
rA: 3 rB: 4 icode: 2 ifun: 0 valC:                  1 valP:                 37 mem_error:0
 valA:                 18 valB:                0
cf_in:0 valE:                 18 cnd:1 cf_out:0
valM:                100
r0:                  0
r1:                  1
 r2:                 17
 r3:                 18
 r4:                 60
 r5:                  5
 r6:                  6
 r7:                100
 r8:                  8
 r9:                  9
 r10:                 10
 r11:                 11
 r12:                 12
 r13:                 13
 r14:                 14
PC:                  35
```

```
                  60
clk:0
rA: 3 rB: 4 icode: 2 ifun: 0 valC:                  1 valP:                 37 mem_error:0
 valA:                 18 valB:                 0
cf_in:0 valE:                 18 cnd:1 cf_out:0
valM:                 100
r0:                   0
r1:                   1
 r2:                  17
 r3:                  18
 r4:                  18
 r5:                   5
 r6:                   6
 r7:                 100
 r8:                   8
 r9:                   9
 r10:                  10
 r11:                  11
 r12:                  12
 r13:                  13
 r14:                  14
PC:                  35

mem_allocated->11001000
mem_allocated->01100100
                  65
clk:1
rA: 5 rB: 3 icode: 2 ifun: 5 valC:                  1 valP:                 39 mem_error:0
 valA:                  5 valB:                 0
cf_in:0 valE:                  5 cnd:1 cf_out:0
valM:                 100
r0:                   0
r1:                   1
 r2:                  17
 r3:                  18
 r4:                  18
 r5:                   5
 r6:                   6
 r7:                 100
 r8:                   8
 r9:                   9
 r10:                  10
 r11:                  11
 r12:                  12
 r13:                  13
 r14:                  14
PC:                  37

                  70
clk:0
rA: 5 rB: 3 icode: 2 ifun: 5 valC:                  1 valP:                 39 mem_error:0
 valA:                  5 valB:                 0
cf_in:0 valE:                  5 cnd:1 cf_out:0
valM:                 100
r0:                   0
r1:                   1
 r2:                  17
 r3:                   5
 r4:                  18
 r5:                   5
 r6:                   6
 r7:                 100
 r8:                   8
 r9:                   9
 r10:                  10
 r11:                  11
 r12:                  12
 r13:                  13
 r14:                  14
PC:                  37

mem_allocated->01100100
```

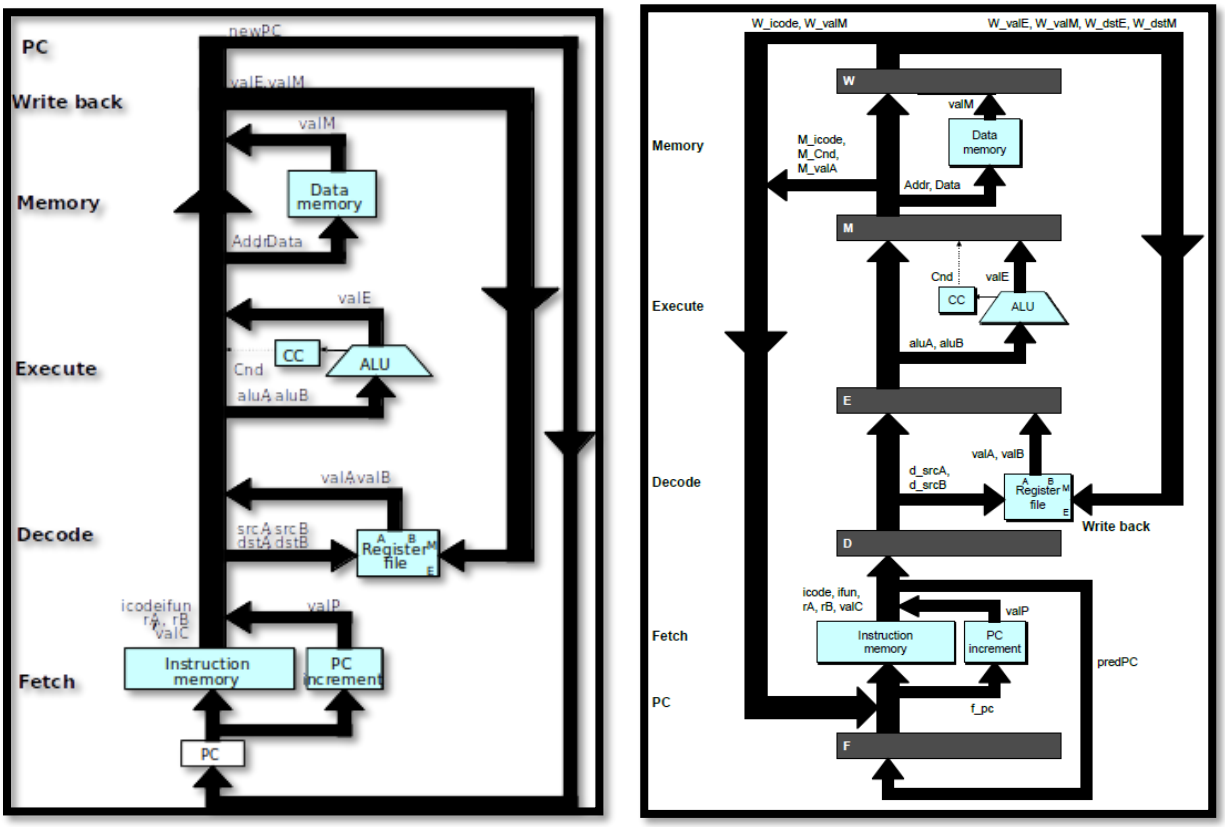
```
mem_allocated->01100100
                 75
clk:1
rA: 5 rB: 3 icode: 0 ifun: 0 valC:                  1 valP:                  40 mem_error:0
 valA:                  5 valB:                  0
cf_in:0 valE:                  0 cnd:0 cf_out:0
valM:                 100
r0:                   0
r1:                   1
 r2:                  17
 r3:                   5
 r4:                  18
 r5:                   5
 r6:                   6
 r7:                 100
 r8:                   8
 r9:                   9
 r10:                 10
 r11:                 11
 r12:                 12
 r13:                 13
 r14:                 14
PC:                  39
```

Pipelined Implementation:

Implementation of the pipeline involved the addition of the registers between each e that can pass designated outputs from one stage register as inputs to the next stage register. Another change is in the PC update. We do not have a separate PC update block, instead, we integrated PC update with the Fetch so that before every instruction executes, the Fetch stage will determine the current PC value and also the next predicted PC. In addition to these, there is a control logic too involved for Bubble, Stall and Forwarding implementation to eliminate pipeline hazards.

ADDING PIPELINE REGISTERS

SEQUENTIAL *PIPELINE*



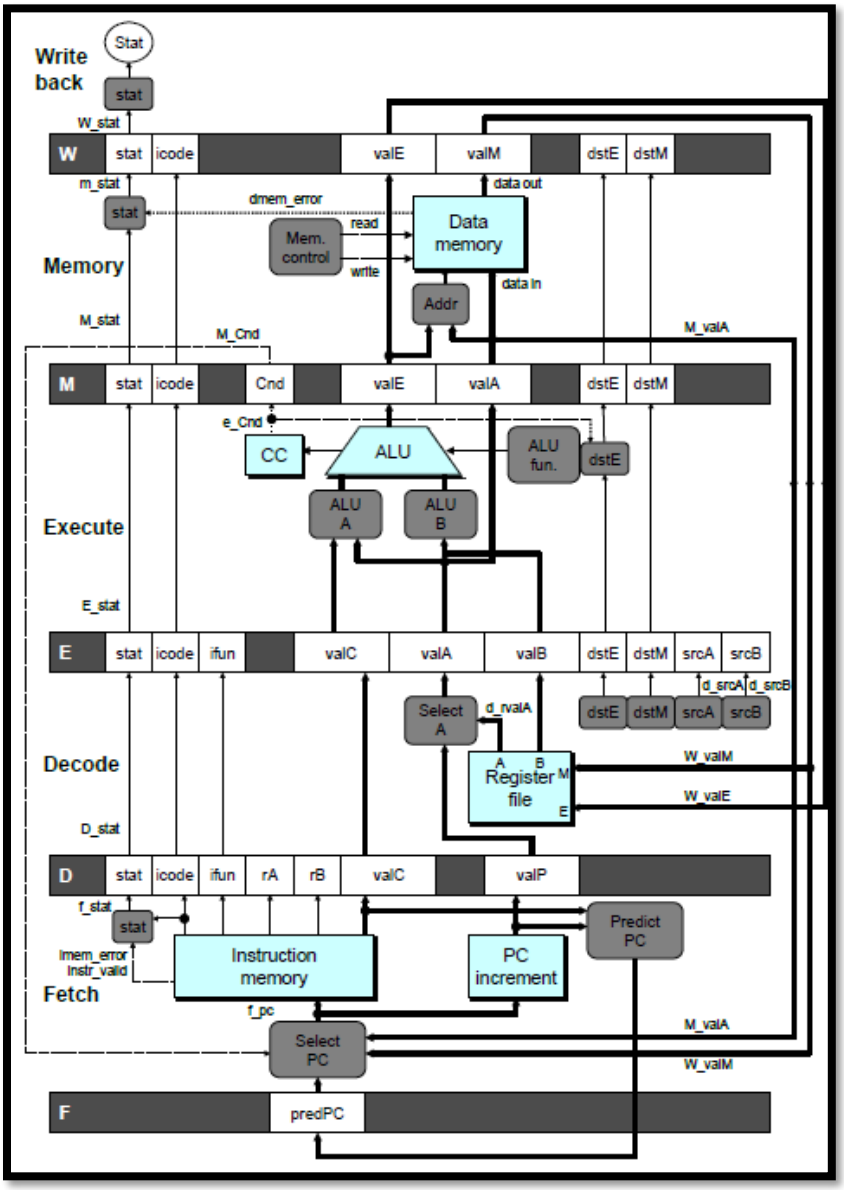
MODULES

- *"fetch.v" - Operates the Fetch block, selects PC and updates predicted PC. It also implements the D_bubble, D_stallandF_stall which implements a Bubble in*

Decode and a Stall condition in Decode and Fetch respectively and updates the Decode Register.

- *"decode.v" - Operates the Decode Block, Implements the E_bubble which adds a Bubble in Execute and updates the Execute Register. Also implements Forwarding Logic.*
- *"execute.v"- Operates the Execute Block and updates the Memory Register*
- *"memory.v"- Operates the Memory Block and updates the Write Back Register.*
- *"write_back.v"- Operates the Write Back Register.*

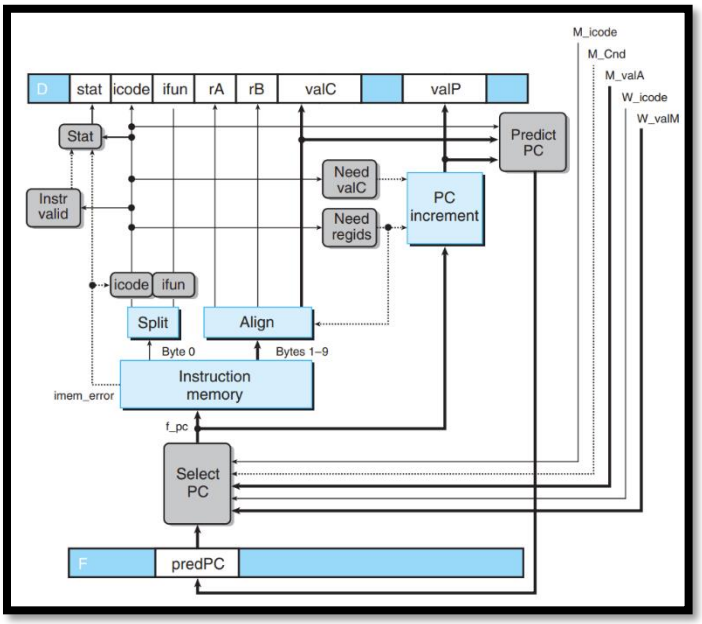
PIPELINE HARDWARE



DESCRIPTION OF INDIVIDUAL BLOCKS

1. FETCH

FETCH HARDWARE



- *INPUTS and OUTPUTS:*

```
module fetch(
    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,
    input clk,
    input [63:0] PC,                //PC value we predicted in the previous stage
    output reg [63:0] temp_pred_pc,
    input [0:79] instr,
    output reg [2:0] D_stat,
    output reg [63:0]pred_pc,       //PC value we predict for the next stage
    input F_stall,
    input D_bubble,
    input D_stall,
    input [63:0]jump_instr_pred,
    input jump_instr_cnd
);
```

The **jump_instr_pred** and **jump_instr_cnd** are used to calculate the **pred_pc.** The jump_instr_cnd will tell if we must choose PC value as predicted PC or any other destination in cases of jump and return as follows: (This code is written in the **"test_bench.v".)**

```
always @(posedge clk)
begin

  if(E_icode==4'h7 && !M_cnd)
        begin  // Jump not taken
            jump_instr_cnd = 1;
            jump_instr_pred = E_valA;
        end
        else if(M_icode==4'h9) begin     // Return statement
            jump_instr_pred = m_valM;
            jump_instr_cnd = 1;
        end
        else
        begin
            jump_instr_cnd = 0;
        end
```

Based on this, we choode the pred_pc as follows: (This code is in the fetch block itself.)

if(jump_instr_cnd)

pred_pc = jump_instr_pred;

else

pred_pc = temp_pred_pc;

- PC UPDATE:

Our code takes the PC as input which is the predicted PC from the previous stage, and it selects the PC value based on the present Instructions and it outputs the pred_pc which the PC value we predicted for the next instruction. Now, this predicted PC we output will be assigned to PC which will serve as the input for the next instruction in the "test_bench.v" file using an always block as follows:

```
always@*
PC = pred_pc;
```

PC PREDICTION STRATEGY:

Instructions that Don't Transfer Control
- Predict next PC to be valP
- Always reliable

Call and Unconditional Jumps
- Predict next PC to be valC (destination)
- Always reliable

Conditional Jumps
- Predict next PC to be valC (destination)
- Only correct if branch is taken
  - Typically right 60% of time

Return Instruction
- Don't try to predict

The PC prediction logic chooses valC for the fetched instruction when it is either a call or a jump, and valP otherwise:

```
word f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];
```

- UPDATING DECODE REGISTER:

It fetches the instruction, and it outputs the rA, rB, valC, icode, ifun, valP and D_stat in the same way as in Sequential which will be updated in the Decode Register.

- D_stat

The D_stat will check the status of the program and it a register of size 3 which has the following bits:

→ *AOK:* This is assigned to 1 if everything is okay in the program else 0
→ *HLT:* This is assigned to 1 if there is halt instruction else 0
→ *INS/MEM*: This is assigned to 1 if there is an invalid instruction or memory address else 0

This is implemented in the fetch block as follows:

```
always @(posedge clk)begin

    if (PC > 1023) begin
        D_stat[2] = 1'b1; // mem/instruction error (INS/ADR)
        D_stat[0] = 1'b0; // not AOK
    end

    else begin
        D_stat[0] = 1'b1; // AOK
        D_stat[1] = 1'b0; //no HLT
        D_stat[2] = 1'b0; //no mem, inst error. (INS/ADR)
    end
```

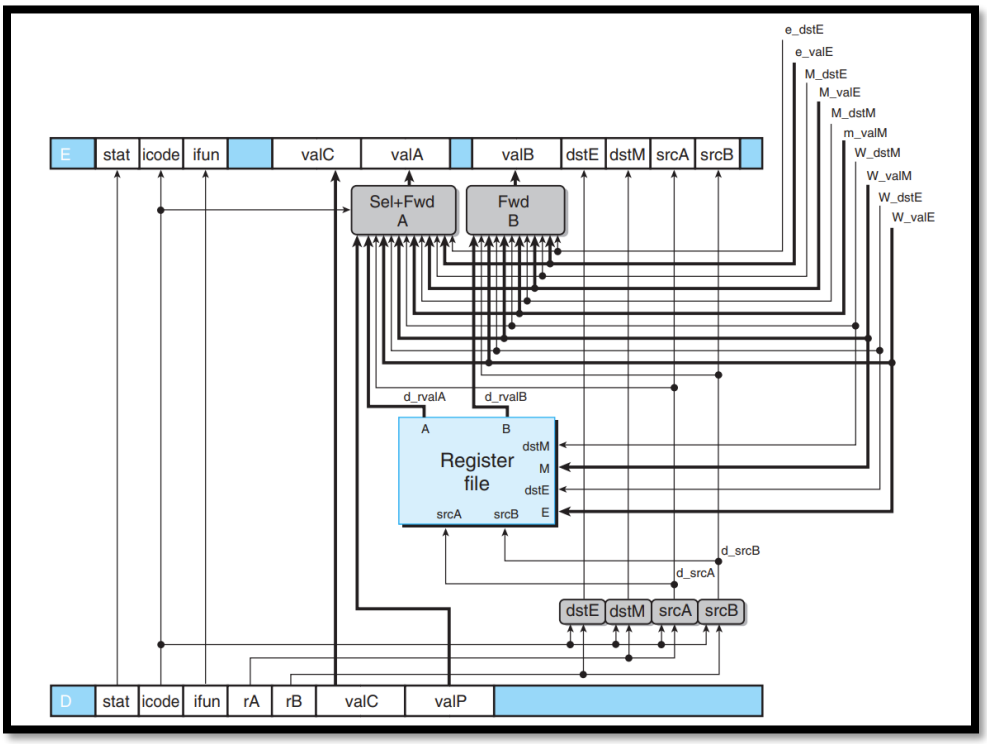
- PIPELINE CONTROL LOGIC:

The implementation of D_bubble, D_stall, F_stall and no control logic is done as follows:

```
if(F_stall)
 begin
    pred_pc =   PC;
    $display("F_stall");

 end

 if(D_stall)
 begin
    $display("D_stall");
    //do nothing
 end



 if(D_bubble)
 begin
    if(jump_instr_cnd)
    pred_pc = jump_instr_pred;
    $display("%d, D_bubble, pred_pc:%d",clk,pred_pc);
    icode <= 4'b0001;
    ifun <= 4'b0000;
    rA <= 4'b0000;
    rB <= 4'b0000;
    valC <= 64'b0;
    valP <= 64'b0;
    D_stat <= 3'b001; //INS/ADR HLT AOK
 end

 if(!F_stall && !D_bubble && !D_stall)
 begin
    rA<=f_rA;
    rB<=f_rB;
    icode = f_icode;
    ifun = f_ifun;
    valC = f_valC;
    valP = f_valP;
    if(jump_instr_cnd)
    pred_pc = jump_instr_pred;
    else
    pred_pc = temp_pred_pc;
 end
```

2. DECODE

DECODE HARDWARE



- INPUTS and OUTPUTS:

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. THe registers and its initial values are defined in the test bench and we input those here to acces register values and output valA and valB values.

```
module decode(clk,D_stat,D_rA,D_rB,D_icode,D_ifun, D_valC, D_valP,reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14,
E_valA,E_valB, E_stat, E_icode, E_ifun, E_valC,
E_dstE, E_dstM, E_srcA, E_srcB,d_srcA, d_srcB, e_dstE, e_valE, m_valM,M_dstM,M_dstE,M_valE, W_dstM, W_dstE, W_valM, W_valE, E_bubble);
input clk, E_bubble;
input [3:0] D_rA,D_rB,D_icode, D_ifun,M_dstM,M_dstE,W_dstM, W_dstE,e_dstE;
input [63:0]  e_valE, W_valM, W_valE,m_valM, M_valE,D_valC, D_valP, reg0,reg1,reg2,reg3,reg4,reg5,reg6,reg7,reg8,reg9,reg10,reg11,reg12,reg13,reg14;
input [2:0]D_stat;
output reg [2:0] E_stat;
output reg [3:0] E_icode, E_ifun, E_dstE, E_dstM, E_srcA, E_srcB, d_srcA,d_srcB;
output reg [63:0] E_valA,E_valB, E_valC;
```

- UPDATING EXECUTE REGISTER:

It outputs the values that are used to update Execute Register such as E_icode, E_dstM etc as shown in the figure above.

```
d_icode = D_icode;
d_ifun  = D_ifun;
d_valC = D_valC;
d_stat = D_stat;
d_srcA = 4'hF;
d_srcB = 4'hF;
d_dstE = 4'hF;
d_dstM = 4'hF;

if(D_icode == 4'b0010) //cmovxx 2
begin
    d_tempvalA=reg_file_in[D_rA];
    d_tempvalB = 0;
    d_srcA = D_rA;
    d_dstE = D_rB;
end

else if(D_icode == 4'b0011) //irmov 3
begin
    d_dstE = D_rB;
end

else if(D_icode == 4'b0100) //rmmov 4
begin
    d_tempvalA = reg_file_in[D_rA];
    d_tempvalB = reg_file_in[D_rB];
    d_srcA = D_rA;
    d_srcB = D_rB;
end

else if (D_icode == 4'b0101) //mrmov 5
begin
    d_tempvalB = reg_file_in[D_rB];
    d_srcB = D_rB;
    d_dstM = D_rA;
end

else if(D_icode == 4'b0110) //opq 6
begin
    d_tempvalA = reg_file_in[D_rA];
    d_tempvalB = reg_file_in[D_rB];
    d_srcA = D_rA;
    d_srcB = D_rB;
    d_dstE = D_rB;
end
```

```
else if(D_icode == 4'b0111) //jxx 7
begin
end

    else if(D_icode == 4'b1000) //call 8
begin
    d_tempvalB = reg_file_in[4];
    d_srcB = 4;
    d_dstE = 4;
end

else if(D_icode == 4'b1001) // ret 9
begin
    d_tempvalA = reg_file_in[4];
    d_tempvalB = reg_file_in[4];
    d_srcA = 4;
    d_srcB = 4;
    d_dstE = 4;
end

else if (D_icode == 4'b1010) //push A
begin
    d_tempvalA = reg_file_in[D_rA];
    d_tempvalB = reg_file_in[4]; //%rsp
    d_srcA = D_rA;
    d_srcB = 4;
    d_dstE = 4;
end

else if(D_icode == 4'b1011) //pop B
begin
    d_tempvalA = reg_file_in[4]; //%rsp
    d_tempvalB = reg_file_in[4]; //%rsp
    d_srcA = 4;
    d_srcB = 4;
    d_dstE = 4;
    d_dstM = D_rA;
end
```

The value of dst registers and valA, valB are set as above:

tempvalA and tempvalB are the temporary values which we are setting which will be assigned to the final E_valA and E_valB if the jump and return conditions don't cause any branch mispredictions.

- FORWARDING LOGIC:

The following logic is implemented for forwarding:

```
## 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;
];
```

Forwarding is coded as follows:

```
    //values setting for d_valA and d_valB
    //FORWARDING
   if(D_icode == 4'd8 ||D_icode == 4'd7) //use incremented PC
       d_valA = D_valP;

   if(D_icode == 4'd8 ||D_icode == 4'd7)
       d_valA = D_valP;
   else if((d_srcA == e_dstE && e_dstE!=4'hF))
       d_valA = e_valE;
   else if(d_srcA == M_dstM && M_dstM!=4'hF)
       d_valA = m_valM;
   else if(d_srcA == M_dstE && M_dstE!=4'hF)
       d_valA=M_valE;
   else if(d_srcA == W_dstM && W_dstM!=4'hF)
       d_valA = W_valM;
   else if(d_srcA == W_dstE && W_dstE!=4'hF)
       d_valA = W_valE;
   else
       d_valA = d_tempvalA;

   // same for updating valB
   if((d_srcB == e_dstE) && e_dstE!=4'hF)
       d_valB = e_valE;
   else if(d_srcB == M_dstM && M_dstM!=4'hF)
       d_valB = m_valM;
   else if(d_srcB == M_dstE && M_dstE!=4'hF)
       d_valB=M_valE;
   else if(d_srcB == W_dstM && W_dstM!=4'hF)
       d_valB = W_valM;
   else if(d_srcB == W_dstE && W_dstE!=4'hF)
       d_valB = W_valE;
   else
       d_valB = d_tempvalB;
```

E_bubble IMPLEMENTATION:

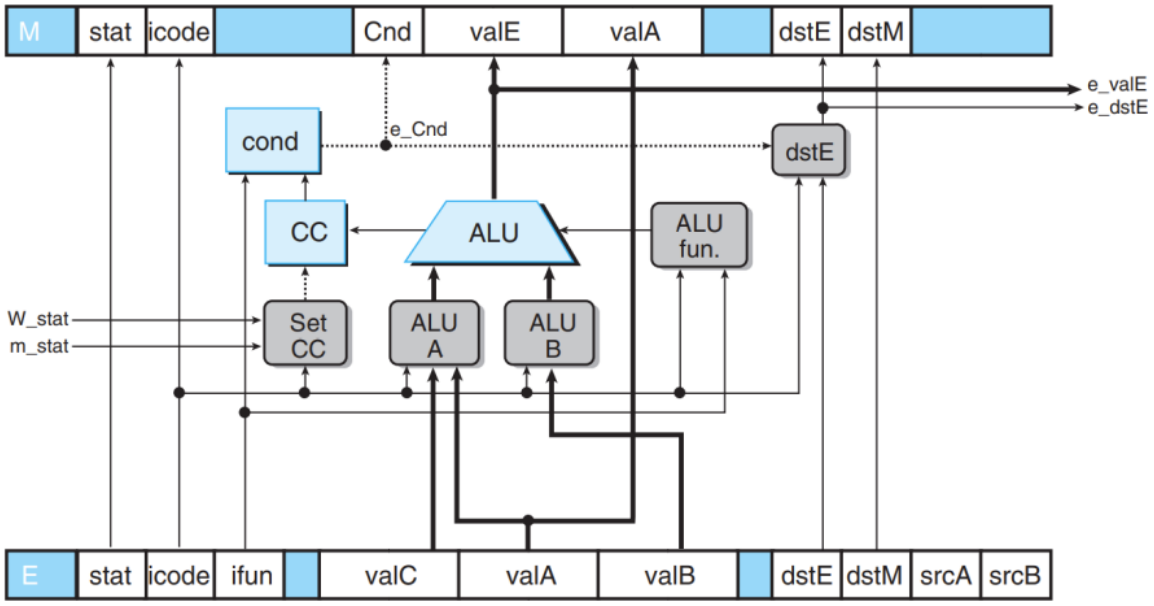
The Implementation of the pipe control logic E_bubble is done as follows:

```
if(E_bubble)
begin
    E_stat <= 4'b1000;
    E_icode <= 4'b0001;
    E_ifun <= 4'b0000;
    E_dstE <= 4'hF;
    E_dstM <= 4'hF;
    E_srcA <= 4'hF;
    E_srcB <= 4'hF;
    E_valA <= 4'b0000;
    E_valB <= 4'b0000;
    E_valC <= 4'b0000;
end
```

```
else
begin
    E_stat <= d_stat;
    E_icode<= d_icode;
    E_ifun <= d_ifun;
    E_dstE <= d_dstE;
    E_dstM <= d_dstM;
    E_srcA <= d_srcA;
    E_srcB <= d_srcB;
    E_valA <= d_valA;
    E_valB <= d_valB;
    E_valC <= d_valC;
end
```

3. EXECUTE

EXECUTE HARDWARE



- INPUTS and OUTPUTS:

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.

The cf_in is nothing the condition flags that are set from earlier stages. Cf_out is the new values that are set in the execute stage.

```
module execute(clk,valE,cnd,cf_out,icode,ifun,valC,valA,valB,cf_in,
M_valA , M_icode, M_dstE, M_dstM , E_stat ,M_stat , E_dstE, E_dstM,e_dstE, e_valE);

  output reg [63:0] valE, M_valA, e_valE;
  output reg[3:0] M_icode , M_dstE , M_dstM, e_dstE;
  output reg cnd;
  output reg [2:0] cf_out, M_stat;

  input [2:0] E_stat;
  input clk;
  input [3:0] icode,ifun, E_dstE, E_dstM;
  input [63:0] valC,valA,valB;
  input [2:0] cf_in;
```

- SETTING FLAGS:

The cf_in is taken, then computation of flags is done when E_icode is 6 (opq) and based on that, cf_out is set as follows:

```
// setting flags;
wire zf ,sf,of;
assign zf = cf_in[0];
assign sf = cf_in[1];
assign of = cf_in[2];
// cf_out will only be set for opq ->icode.
//We have to set cnd-> for jumpxx, and cmovxx(conditional mov.)
```

```
4'b0110: //opq.
begin
    e_valE = valE_op;
    //setting cf_out:zf,sf,of
    // cf_out[0] <= 0;
    if(valE == 0) cf_out[0] <= 1;
    else cf_out[0] <= 0;;
    cf_out[1] <= e_valE[63];//sf//lastbit 1.
    cf_out[2] <= overflow_use;
end
```

After seeting this, for the cf_out to become cf_in for the next opq instruction, we do thefollowing in the test bench:

```
always @*
begin
cf_in=cf_out;
end
```

- SETTING CONDITION cnd:

This is done when E_icode either represents jumpxx or cmovxx

If this is 1 if jump/move must be done and condition is met else if 0, jump/move not taken. The value of e_dstE is computed based on the cnd which will make it either

E_dstE or an empty register

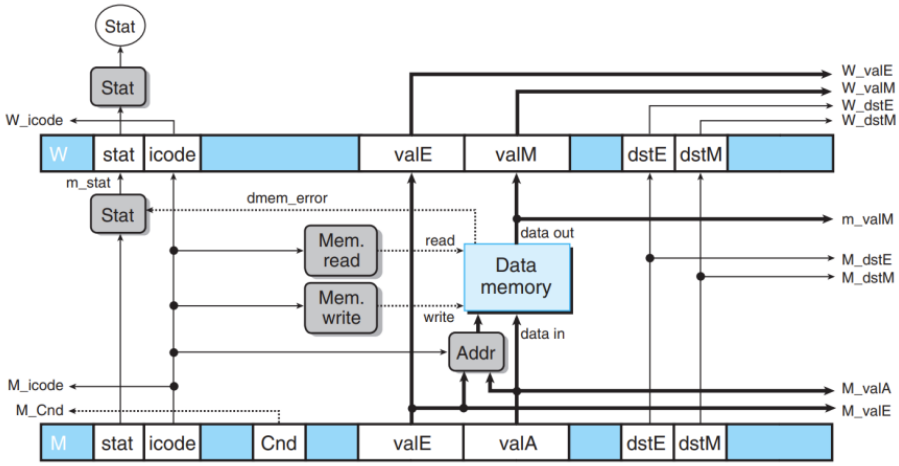
```
 //setting cnd
cnd=0;
    if(icode == 2 || icode == 7)
    //uc,le,l,e,e,ne,ge,g

    begin
        case(ifun)
        4'b0000:
        begin
            cnd = 1;
        end
        4'b0001://le
        begin
            cnd = (sf^of)|zf;
        end
        4'b0010:
        begin
            cnd = of^sf;
        end
        4'b0011:
        begin
            cnd = zf;
        end
        4'b0100:
        begin
            cnd = ~zf;
            $display("cnd->>%d",cnd);
        end
        4'b0101:
        begin
            cnd = ~(zf^of);
        end
        4'b0110:
        begin
            cnd = ~(zf^of)&(~zf);
        end
        endcase

        if(icode==2)
        begin
            if(cnd == 1)
                e_dstE = E_dstE;
            else
                e_dstE = 4'hF;
        end
        else
            e_dstE = E_dstE;
    end

    else
        e_dstE = E_dstE;

  end
```

So, after this, the place where it must move in case of cmovxx I.e M_dstE is updated as:

M_dstE = cnd ? e_dstE : 4'hf;

1. MEMORY

MEMORY HARDWARE



- INPUTS and OUTPUTS:

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_valA,M_valE,M_dstE,M_dstM,
  m_valM,W_stat,W_icode, W_valE, W_valM,W_dstE, W_dstM, m_stat
);

  input clk, M_cnd;
  input [2:0] M_stat;
  input [3:0] M_icode;
  input [63:0] M_valA, M_valE;
  input [3:0] M_dstE, M_dstM;
  input [63:0] valP;//next instr in row

  output reg [2:0] W_stat;
  output reg [3:0] W_icode,W_dstE,W_dstM;
  output reg [63:0] W_valE, W_valM, m_valM;
  output reg [63:0] datamem;
  output reg [2:0] m_stat;
```

This block functions the same way as that of the sequential memory block. It also sets the m_stat and W_stat as per dmem_error. The register gets updated once the clk hits and the output for W_stat,

W_icode, W_valE, W_valM, W_dstE and W_dstM is available as an output

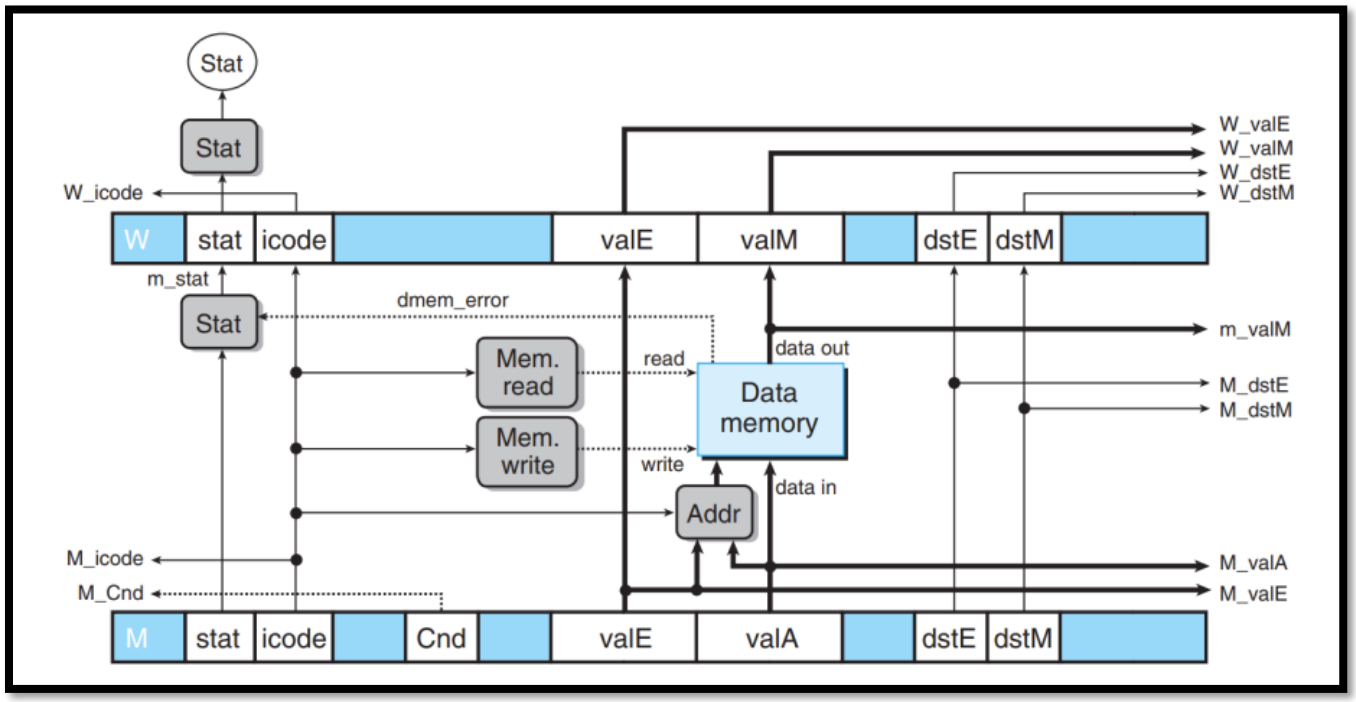
of the memory register.

NOTE: *In memory.v, when we have to read memory, we have to access the memory, but since memory can be quite large and is complex to implement due to storage issues, a small change we made here to make it simple is to already assign fixed values to the memory values of the addresses M_valE and M_valE, so that whenever we want to access memory, we use one of these addresses to access it and we get output as these predefined values.*

```
data_mem[M_valA] = 8'd200; //for pop ooperation.
// data_mem[M_valA] = 8'd26; //for return
data_mem[M_valE] = 8'd100;
```

1. WRITE BACK

WRITEBACK HARDWARE



- INPUTS and OUTPUTS:

We get W_stat, W_icode, W_valE, W_valM, W_dstE, W_dstM as the inputs and we update/write back to registers as output as per input instructions and values. WE get registers as both inputs and outputs.

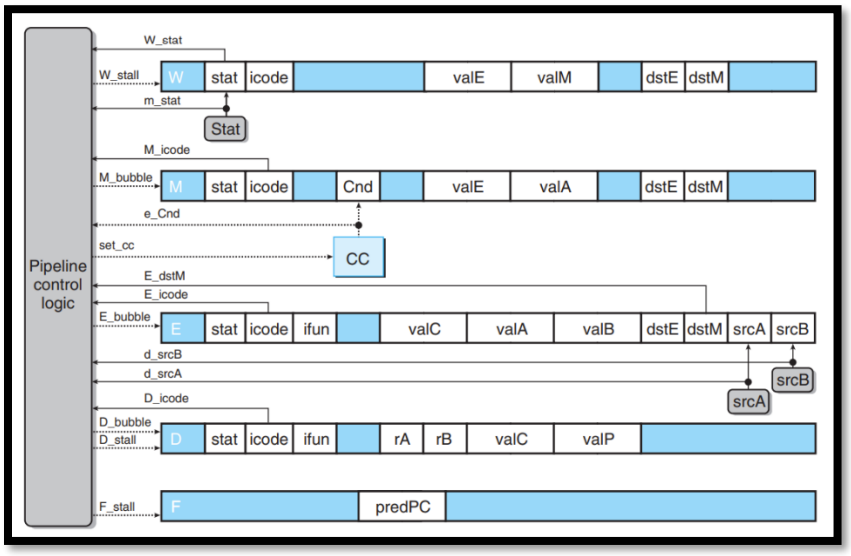
```
module write_back(clk, W_stat, W_icode,W_valE,W_valM,W_dstE,W_dstM,
reg_in0,reg_in1,reg_in2,reg_in3,reg_in4,reg_in5,reg_in6,reg_in7,reg_in8,reg_in9,reg_in10,reg_in11,reg_in12,reg_in13,reg_in14,
reg_out0,reg_out1,reg_out2,reg_out3,reg_out4,reg_out5,reg_out6,reg_out7,reg_out8,reg_out9,reg_out10,reg_out11,reg_out12,reg_out13,reg_out14
);

input [2:0] W_stat;
input [3:0] W_icode,W_dstE , W_dstM ;
input [63:0] W_valE,W_valM;
input [63:0] reg_in0,reg_in1,reg_in2,reg_in3,reg_in4,reg_in5,reg_in6,reg_in7,reg_in8,reg_in9,reg_in10,reg_in11,reg_in12,reg_in13,reg_in14;
output[63:0] reg_out0,reg_out1,reg_out2,reg_out3,reg_out4,reg_out5,reg_out6,reg_out7,reg_out8,reg_out9,reg_out10,reg_out11,reg_out12,reg_out13,reg_out14;
reg [63:0] reg_file [0:14];
input clk;
```

The logic to update registers is the same as in sequential design.

PIPELINE CONTROL LOGIC

(Bubble,Stall and Forwarding)

The control logic is implemented in the respective blocks which is explained earlier as per the figure given below:



The conditions for implementing these for few data hazards are as follows:

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

Detection of these data hazards

Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & !e_Cnd

Now based on these, we can frame conditions for when to trigger D_stall, F_stall, D_bubble and E_bubble as follows:

```
bool F_stall =
     # Conditions for a load/use hazard
     E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB } ||
     # Stalling at fetch while ret passes through pipeline
     IRET in { D_icode, E_icode, M_icode };

bool D_stall =
     # Conditions for a load/use hazard
     E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };

bool D_bubble =
     # Mispredicted branch
     (E_icode == IJXX && !e_Cnd) ||
     # Stalling at fetch while ret passes through pipeline
      IRET in { D_icode, E_icode, M_icode };

bool E_bubble =
     # Mispredicted branch
     (E_icode == IJXX && !e_Cnd) ||
     # Load/use hazard
     E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };
```

In our code, we have registers D_stall, F_stall etc which will be updated based on the given conditions and when updated, based on these values, the respective control logic is implemented in the respective functional blocks of the processor.

CODE TO TRIGGER CONTROL LOGIC

```
always @(posedge clk)
begin

  if(E_icode==4'h7 && !M_cnd)
      begin  // Jump not taken
          jump_instr_cnd = 1;
          jump_instr_pred = E_valA;
      end
      else if(M_icode==4'h9) begin     // Return statement
          jump_instr_pred = m_valM;
          jump_instr_cnd = 1;
      end
      else
      begin
          jump_instr_cnd = 0;
      end

  if(E_icode == 4'h7 & !M_cnd)
  begin
    D_bubble = 1;
    E_bubble = 1;
    F_stall = 0;
    D_stall = 0;
  end
  else if((E_icode == 4'h5 | E_icode == 4'hB) & (E_dstM==d_srcA | E_dstM==d_srcB))
  begin
    F_stall = 1;
    D_stall = 1;
    E_bubble = 1;
     D_bubble = 0;
  end
  else if(E_icode == 4'h9 | M_icode == 4'h9 | D_icode == 4'h9)
  begin
      F_stall = 1;
      D_bubble = 1;
      D_stall = 0;
      E_bubble = 0;
  end
  else if(E_icode == 4'h0 | m_stat!=4'b1000 | W_stat!=4'b1000)
    begin
        cf_in = 0;
        F_stall = 0;
        D_stall = 0;
        E_bubble = 0;
        D_bubble = 0;
    end
  else
  begin
    F_stall = 0;
        D_stall = 0;
        E_bubble = 0;
        D_bubble = 0;
  end
end
```

ERROR HANDLING

(Done in test bench using stat)

At every stage, we set the sat register as D_stat, E_stat, M_stat, W_stat and if any of these point to any error,then we have toimplement the given and stop the program as follows:

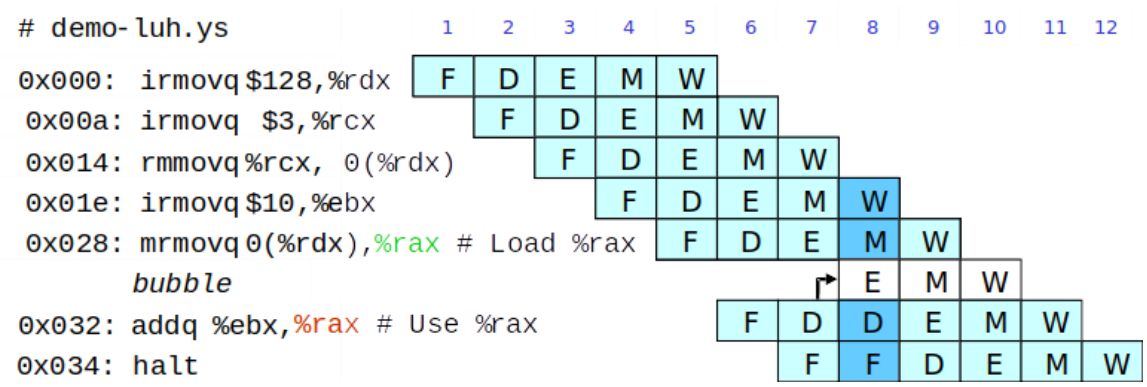
```
always @(*) begin
  if(D_stat[1] == 1 || E_stat[1] ==1  || M_stat[1] == 1 || W_stat[1]==1)
  begin
    $display("BYE-BYE\n");
    $finish;
  end
  if(D_stat[2] == 1 || E_stat[2] ==1  || M_stat[2] == 1 || W_stat[2]==1)
  begin
    $display("BYE-BYE2\n");
    $finish;
  end
end
```

(Refer to the D_stat section under fetch block of Pipeline Implementation section to understand how this stat register is structured and what each of the 3-bits mean and its initial values)

TEST BENCH RESULTS

- *LOAD/STORE INSTRUCTIONS:*

ASSEMBLY CODE:



MACHINE CODE:

```
// <--start of load/store hazard---
instr_mem[1]=8'b00110000; //3 0
instr_mem[2]=8'b11110010; //F rB(2)
instr_mem[3]=8'b00000000; //V
instr_mem[4]=8'b00000000; //V
instr_mem[5]=8'b00000000; //V
instr_mem[6]=8'b00000000; //V
instr_mem[7]=8'b00000000; //V
instr_mem[8]=8'b00000000; //V
instr_mem[9]=8'b00000000; //V
instr_mem[10]=8'b10000000; //V=128

instr_mem[11]=8'b00110000; //3 0
instr_mem[12]=8'b11110001; //F rB(1)
instr_mem[13]=8'b00000000; //V
instr_mem[14]=8'b00000000; //V
instr_mem[15]=8'b00000000; //V
instr_mem[16]=8'b00000000; //V
instr_mem[17]=8'b00000000; //V
instr_mem[18]=8'b00000000; //V
instr_mem[19]=8'b00000000; //V
instr_mem[20]=8'b00000011; //V=3

instr_mem[21]=8'b01000000; //4 0 //rmmov
instr_mem[22]=8'b00010010; //rA rB(1,2)
instr_mem[23]=8'b00000000; //D
instr_mem[24]=8'b00000000; //D
instr_mem[25]=8'b00000000; //D
instr_mem[26]=8'b00000000; //D
instr_mem[27]=8'b00000000; //D
instr_mem[28]=8'b00000000; //D
instr_mem[29]=8'b00000000; //D
instr_mem[30]=8'b00000000; //D
instr_mem[31]=8'b00110000; //3 0//irmov
instr_mem[32]=8'b11110011; //F rB(3)
instr_mem[33]=8'b00000000; //V
instr_mem[34]=8'b00000000; //V
instr_mem[35]=8'b00000000; //V
instr_mem[36]=8'b00000000; //V
instr_mem[37]=8'b00000000; //V
instr_mem[38]=8'b00000000; //V
instr_mem[39]=8'b00000000; //V
instr_mem[40]=8'b00001010; //V=10

instr_mem[41]=8'b01010000; //5 0//mrmov
instr_mem[42]=8'b00000011; //rA rB(0,3)
instr_mem[43]=8'b00000000; //D
instr_mem[44]=8'b00000000; //D
instr_mem[45]=8'b00000000; //D
instr_mem[46]=8'b00000000; //D
instr_mem[47]=8'b00000000; //D
instr_mem[48]=8'b00000000; //D
instr_mem[49]=8'b00000000; //D
instr_mem[50]=8'b00000000; //D

instr_mem[51]=8'b01100000; //6 fn
instr_mem[52]=8'b00110000; //rA rB(3,0)

instr_mem[53] = 8'b00010000; //nop
instr_mem[54] = 8'b00010000; //nop
instr_mem[55] = 8'b00010000; //nop
instr_mem[56] = 8'b00010000; //nop
instr_mem[57] = 8'b00010000; //nop
instr_mem[58] = 8'b00010000; //nop
instr_mem[59] = 8'b00000000; //halt
// ----- end of load/store hazard>
```

OUTPUT:

The output can be seen by observing the icodes at each clock to know the pipeline flow and the initial and final register values to know if the program is executed properly.

```
                  0
 clK:0 , D_icode: x , E_icode: x , M_icode= x , W_icode= x, PC=                  1
                  5
 clK:1 , D_icode: x , E_icode: x , M_icode= x , W_icode= x, PC=                  1
men_allocated->xxxxxxxx
                 10
 clK:0 , D_icode: 3 , E_icode: x , M_icode= x , W_icode= x, PC=                 11
                 15
 clK:1 , D_icode: 3 , E_icode: x , M_icode= x , W_icode= x, PC=                 11
men_allocated->01100100
                 20
 clK:0 , D_icode: 3 , E_icode: 3 , M_icode= x , W_icode= x, PC=                 21
                 25
 clK:1 , D_icode: 3 , E_icode: 3 , M_icode= x , W_icode= x, PC=                 21
men_allocated->01100100
                 30
 clK:0 , D_icode: 4 , E_icode: 3 , M_icode= 3 , W_icode= x, PC=                 31
                 35
 clK:1 , D_icode: 4 , E_icode: 3 , M_icode= 3 , W_icode= x, PC=                 31
men_allocated->01100100
                 40
 clK:0 , D_icode: 3 , E_icode: 4 , M_icode= 3 , W_icode= 3, PC=                 41
                 45
 clK:1 , D_icode: 3 , E_icode: 4 , M_icode= 3 , W_icode= 3, PC=                 41
men_allocated->01100100
                 50
 clK:0 , D_icode: 5 , E_icode: 3 , M_icode= 4 , W_icode= 3, PC=                 51
                 55
 clK:1 , D_icode: 5 , E_icode: 3 , M_icode= 4 , W_icode= 3, PC=                 51
men_allocated->00000011
                 60
 clK:0 , D_icode: 6 , E_icode: 5 , M_icode= 3 , W_icode= 4, PC=                 53
                 65
 clK:1 , D_icode: 6 , E_icode: 5 , M_icode= 3 , W_icode= 4, PC=                 53
F_stall
D_stall
men_allocated->01100100
                 70
 clK:0 , D_icode: 6 , E_icode: 1 , M_icode= 5 , W_icode= 3, PC=                 53
                 75
 clK:1 , D_icode: 6 , E_icode: 1 , M_icode= 5 , W_icode= 3, PC=                 53
men_allocated->01100100
                 80
 clK:0 , D_icode: 1 , E_icode: 6 , M_icode= 1 , W_icode= 5, PC=                 54
                 85
 clK:1 , D_icode: 1 , E_icode: 6 , M_icode= 1 , W_icode= 5, PC=                 54
men_allocated->01100100
                 90
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 6 , W_icode= 1, PC=                 55
                 95
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 6 , W_icode= 1, PC=                 55
men_allocated->01100100
                100
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 6, PC=                 56
                105
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 6, PC=                 56
men_allocated->01100100
                110
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                 57
                115
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                 57
men_allocated->01100100
                120
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                 58
                125
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                 58
men_allocated->01100100
                130
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                 59
BYE-BYE


                135
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                 59
```

INITIAL REGISTER VALUES

```
r0:                0
r1:                1
 r2:               2
 r3:               3
 r4:              60
 r5:               5
 r6:               6
 r7:               7
 r8:               8
 r9:               9
 r10:             10
 r11:             11
 r12:             12
 r13:             13
 r14:             14
```

FINAL UPDATED REGISTER VALUES

```
r0:              110
r1:                3
 r2:             128
 r3:              10
 r4:              60
 r5:               5
 r6:               6
 r7:               7
 r8:               8
 r9:               9
 r10:             10
 r11:             11
 r12:             12
 r13:             13
 r14:             14
```

We thus get the desired output. To check if rmmov wprks, we have printed the value being enteredinto memory whenever there is a memory write and as seen in the below figure at the bottom left, there is value 3 which is the value in reg 1, to memory.

```
                55
clk:1
D_rA: 0 D_rB: 3
D_icode: 5 E_icode: 3 M_icode:  4 W_icode: 3
D_valC:                  0 e_valE:                10 M_valE:              128 W_valE:                3
 E_valA:                 1 E_valB:                 2
cf_in:0 M_valE:               128 M_cnd:0 cf_out:x
W_valM:                 x
e_dstE: 3 E_dstE: 3 M_dstE:15 W_dstE: 1
r0:                   0
r1:                   3
 r2:                128
 r3:                  3
 r4:                 60
 r5:                  5
 r6:                  6
 r7:                  7
 r8:                  8
 r9:                  9
 r10:                10
 r11:                11
 r12:                12
 r13:                13
 r14:                14
PC:                 51

mem_allocated->00000011
```

The operations we checked here are: irmov, rmmov, mrmov, opq, halt and we also checked the forwarding and bubble inserting implementation, and we can confirm that they all work well.

- *BRANCH MISPREDICTION WITH JUMP INSTRUCTION:*

ASSEMBLY CODE:

```
0x000:     xorq %rax,%rax
0x002:     jne  t              # Not taken
0x00b:     irmovq $1, %rax     # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx     # Target (Should not execute)
0x023:     irmovq $4, %rcx     # Should not execute
0x02d:     irmovq $5, %rdx     # Should not execute
```

MACHINE CODE:

```
instr_mem[1]=8'b01100011; //6 xor
instr_mem[2]=8'b00110011; //rA rB (0,0)

instr_mem[3] = 8'b01110100;//7,4 (jne, ne)
instr_mem[4]=8'b00000000; //V
instr_mem[5]=8'b00000000; //V
instr_mem[6]=8'b00000000; //V
instr_mem[7]=8'b00000000; //V
instr_mem[8]=8'b00000000; //V
instr_mem[9]=8'b00000000; //V
instr_mem[10]=8'b00000000; //V
instr_mem[11]=8'b00011010; //V v->26

instr_mem[12]=8'b00110000; //3 0//irmov
instr_mem[13]=8'b11110000; //F rB(0)
instr_mem[14]=8'b00000000; //V
instr_mem[15]=8'b00000000; //V
instr_mem[16]=8'b00000000; //V
instr_mem[17]=8'b00000000; //V
instr_mem[18]=8'b00000000; //V
instr_mem[19]=8'b00000000; //V
instr_mem[20]=8'b00000000; //V
instr_mem[21]=8'b00000001; //V=1

instr_mem[22] = 8'b00010000; //nop
instr_mem[23] = 8'b00010000; //nop
instr_mem[24] = 8'b00010000; //nop
instr_mem[25] = 8'b00000000; //halt
```

```
instr_mem[26]=8'b00110000; //3 0//irmov
instr_mem[27]=8'b11110010; //F rB(2)
instr_mem[28]=8'b00000000; //V
instr_mem[29]=8'b00000000; //V
instr_mem[30]=8'b00000000; //V
instr_mem[31]=8'b00000000; //V
instr_mem[32]=8'b00000000; //V
instr_mem[33]=8'b00000000; //V
instr_mem[34]=8'b00000000; //V
instr_mem[35]=8'b00000011; //V=3

instr_mem[36]=8'b00110000; //3 0//irmov
instr_mem[37]=8'b11110001; //F rB(1)
instr_mem[38]=8'b00000000; //V
instr_mem[39]=8'b00000000; //V
instr_mem[40]=8'b00000000; //V
instr_mem[41]=8'b00000000; //V
instr_mem[42]=8'b00000000; //V
instr_mem[43]=8'b00000000; //V
instr_mem[44]=8'b00000000; //V
instr_mem[45]=8'b00000100; //V=4

instr_mem[46]=8'b00110000; //3 0//irmov
instr_mem[47]=8'b11110010; //F rB(2)
instr_mem[48]=8'b00000000; //V
instr_mem[49]=8'b00000000; //V
instr_mem[50]=8'b00000000; //V
instr_mem[51]=8'b00000000; //V
instr_mem[52]=8'b00000000; //V
instr_mem[53]=8'b00000000; //V
instr_mem[54]=8'b00000000; //V
instr_mem[55]=8'b00000101; //V=5
```

OUTPUT:

The output can be seen by observing the icodes at each clock to know the pipeline flow and the initial and final register values to know if the program is executed properly.

```
clK:0 , D_icode: x , E_icode: x , M_icode= x , W_icode= x, PC=                1
                5
clK:1 , D_icode: x , E_icode: x , M_icode= x , W_icode= x, PC=                1
mem_allocated->xxxxxxxx
                10
clK:0 , D_icode: 6 , E_icode: x , M_icode= x , W_icode= x, PC=                3
                15
clK:1 , D_icode: 6 , E_icode: x , M_icode= x , W_icode= x, PC=                3
mem_allocated->01100100
                20
clK:0 , D_icode: 7 , E_icode: 6 , M_icode= x , W_icode= x, PC=               26
                25
clK:1 , D_icode: 7 , E_icode: 6 , M_icode= x , W_icode= x, PC=               26
Jump to->               12
mem_allocated->01100100
                30
clK:0 , D_icode: 3 , E_icode: 7 , M_icode= 6 , W_icode= x, PC=               36
cnd->>0
                35
clK:1 , D_icode: 3 , E_icode: 7 , M_icode= 6 , W_icode= x, PC=               36
mem_allocated->01100100
0, D_bubble, pred_pc:                12
                40
clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 7 , W_icode= 6, PC=               12
                45
clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 7 , W_icode= 6, PC=               12
mem_allocated->01100100
                50
clK:0 , D_icode: 3 , E_icode: 1 , M_icode= 1 , W_icode= 7, PC=               22
                55
clK:1 , D_icode: 3 , E_icode: 1 , M_icode= 1 , W_icode= 7, PC=               22
mem_allocated->01100100
                60
clK:0 , D_icode: 1 , E_icode: 3 , M_icode= 1 , W_icode= 1, PC=               23
                65
clK:1 , D_icode: 1 , E_icode: 3 , M_icode= 1 , W_icode= 1, PC=               23
mem_allocated->01100100
                70
clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 3 , W_icode= 1, PC=               24
                75
clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 3 , W_icode= 1, PC=               24
mem_allocated->01100100
                80
clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 3, PC=               25
BYE-BYE

                85
clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 3, PC=               25
```

We thus get the desired outputs. The jump instruction also works well

INITIAL REGISTER VALUES

```
r0:                  0
r1:                  1
 r2:                 2
 r3:                 3
 r4:                60
 r5:                 5
 r6:                 6
 r7:                 7
 r8:                 8
 r9:                 9
 r10:               10
 r11:               11
 r12:               12
 r13:               13
 r14:               14
```

FINAL UPDATED REGISTER VALUES

```
r0:                  1
r1:                  1
 r2:                 2
 r3:                 0
 r4:                60
 r5:                 5
 r6:                 6
 r7:                 7
 r8:                 8
 r9:                 9
 r10:               10
 r11:               11
 r12:               12
 r13:               13
 r14:               14
```

- *CALL and RETURN*

ASSEMBLY CODE:

```
  0x000:    irmovq Stack,%rsp  # Intialize stack pointer
0x00a:    nop                # Avoid hazard on %rsp
0x00b:    nop
0x00c:    nop
0x00d:    call p             # Procedure call
0x016:    irmovq $5,%rsi     # Return point
0x020:    halt
0x020: .pos 0x20
0x020: p: nop                 # procedure
0x021:    nop
0x022:    nop
0x023:    ret
0x024:    irmovq $1,%rax      # Should not be executed
0x02e:    irmovq $2,%rcx      # Should not be executed
0x038:    irmovq $3,%rdx      # Should not be executed
0x042:    irmovq $4,%rbx      # Should not be executed
0x100: .pos 0x100
0x100: Stack:                 # Initial stack pointer
```

MACHINE CODE:

```
//call
instr_mem[13]=8'b00110000; //4 0//irmov
instr_mem[14]=8'b11110100; //F rB(4-stack pointer)
instr_mem[15]=8'b00000000; //V
instr_mem[16]=8'b00000000; //V
instr_mem[17]=8'b00000000; //V
instr_mem[18]=8'b00000000; //V
instr_mem[19]=8'b00000000; //V
instr_mem[20]=8'b00000000; //V
instr_mem[21]=8'b00000000; //V
instr_mem[22]=8'b01000000; //V=64

instr_mem[23] = 8'b00010000;  //nop
instr_mem[24] = 8'b00010000;  //nop
instr_mem[25] = 8'b00010000;  //nop

instr_mem[26]=8'b10000000;//8, call
instr_mem[27]=8'b00000000; //V
instr_mem[28]=8'b00000000; //V
instr_mem[29]=8'b00000000; //V
instr_mem[30]=8'b00000000; //V
instr_mem[31]=8'b00000000; //V
instr_mem[32]=8'b00000000; //V
instr_mem[33]=8'b00000000; //V
instr_mem[34]=8'b00110010; //v->50

instr_mem[35]=8'b00110000; //3 0//irmov
instr_mem[36]=8'b11110110; //F rB(6)
instr_mem[37]=8'b00000000; //V
instr_mem[38]=8'b00000000; //V
instr_mem[39]=8'b00000000; //V
instr_mem[40]=8'b00000000; //V
instr_mem[41]=8'b00000000; //V
instr_mem[42]=8'b00000000; //V
instr_mem[43]=8'b00000000; //V
instr_mem[44]=8'b00000101; //V=5

instr_mem[45] = 8'b00010000;  //nop
instr_mem[46] = 8'b00010000;  //nop
instr_mem[47] = 8'b00010000;  //nop
instr_mem[48] = 8'b00010000;  //nop
instr_mem[49] = 8'b00000000;  //halt
```

```
// p:
    instr_mem[50] = 8'b00010000;  //nop
    instr_mem[51] = 8'b00010000;  //nop
    instr_mem[52] = 8'b00010000;  //nop

    instr_mem[53]=8'b10010000;  //return

    instr_mem[54]=8'b00110000; //3 0//irmov
    instr_mem[55]=8'b11110000; //F rB(0)
    instr_mem[56]=8'b00000000; //V
    instr_mem[57]=8'b00000000; //V
    instr_mem[58]=8'b00000000; //V
    instr_mem[59]=8'b00000000; //V
    instr_mem[60]=8'b00000000; //V
    instr_mem[61]=8'b00000000; //V
    instr_mem[62]=8'b00000000; //V
    instr_mem[63]=8'b00000001; //V=1

    instr_mem[64]=8'b00110000; //3 0//irmov
    instr_mem[65]=8'b11110001; //F rB(1)
    instr_mem[66]=8'b00000000; //V
    instr_mem[67]=8'b00000000; //V
    instr_mem[68]=8'b00000000; //V
    instr_mem[69]=8'b00000000; //V
    instr_mem[70]=8'b00000000; //V
    instr_mem[71]=8'b00000000; //V
    instr_mem[72]=8'b00000000; //V
    instr_mem[73]=8'b00000010; //V=2


    instr_mem[74]=8'b00110000; //3 0//irmov
    instr_mem[75]=8'b11110010; //F rB(2)
    instr_mem[76]=8'b00000000; //V
    instr_mem[77]=8'b00000000; //V
    instr_mem[78]=8'b00000000; //V
    instr_mem[79]=8'b00000000; //V
    instr_mem[80]=8'b00000000; //V
    instr_mem[81]=8'b00000000; //V
    instr_mem[82]=8'b00000000; //V
    instr_mem[83]=8'b00000011; //V=3


    instr_mem[84]=8'b00110000; //3 0//irmov
    instr_mem[85]=8'b11110011; //F rB(3)
    instr_mem[86]=8'b00000000; //V
    instr_mem[87]=8'b00000000; //V
    instr_mem[88]=8'b00000000; //V
    instr_mem[89]=8'b00000000; //V
    instr_mem[90]=8'b00000000; //V
    instr_mem[91]=8'b00000000; //V
    instr_mem[92]=8'b00000000; //V
    instr_mem[93]=8'b00000100; //V=4
```

OUTPUT:

```
 clK:0 , D_icode: x , E_icode: x , M_icode= x , W_icode= x, PC=                13
                  5
 clK:1 , D_icode: x , E_icode: x , M_icode= x , W_icode= x, PC=                13
men_allocated->xxxxxxxx
                 10
 clK:0 , D_icode: 3 , E_icode: x , M_icode= x , W_icode= x, PC=                23
                 15
 clK:1 , D_icode: 3 , E_icode: x , M_icode= x , W_icode= x, PC=                23
men_allocated->01100100
                 20
 clK:0 , D_icode: 1 , E_icode: 3 , M_icode= x , W_icode= x, PC=                24
                 25
 clK:1 , D_icode: 1 , E_icode: 3 , M_icode= x , W_icode= x, PC=                24
men_allocated->01100100
                 30
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 3 , W_icode= x, PC=                25
                 35
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 3 , W_icode= x, PC=                25
men_allocated->01100100
                 40
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 3, PC=                26
                 45
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 3, PC=                26
men_allocated->01100100
                 50
 clK:0 , D_icode: 8 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                50
                 55
 clK:1 , D_icode: 8 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                50
Jump to->                  35
men_allocated->01100100
                 60
 clK:0 , D_icode: 1 , E_icode: 8 , M_icode= 1 , W_icode= 1, PC=                51
                 65
 clK:1 , D_icode: 1 , E_icode: 8 , M_icode= 1 , W_icode= 1, PC=                51
men_allocated->01100100
                 70
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 8 , W_icode= 1, PC=                52
                 75
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 8 , W_icode= 1, PC=                52
men_allocated->00100011
                 80
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 8, PC=                53
                 85
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 8, PC=                53
men_allocated->01100100
                 90
 clK:0 , D_icode: 9 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                54
                 95
 clK:1 , D_icode: 9 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                54
men_allocated->01100100
F stall
```

```
, D_bubble, pred_pc:                  54
                  100
 clK:0 , D_icode: 1 , E_icode: 9 , M_icode= 1 , W_icode= 1, PC=                54
                  105
 clK:1 , D_icode: 1 , E_icode: 9 , M_icode= 1 , W_icode= 1, PC=                54
_stall
, D_bubble, pred_pc:                  54
en_allocated->01100100
                  110
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 9 , W_icode= 1, PC=                54
                  115
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 9 , W_icode= 1, PC=                54
en_allocated->01100100
_stall
, D_bubble, pred_pc:                  35
                  120
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 9, PC=                35
                  125
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 9, PC=                35
en_allocated->01100100
                  130
 clK:0 , D_icode: 3 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                45
                  135
 clK:1 , D_icode: 3 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                45
en_allocated->01100100
                  140
 clK:0 , D_icode: 1 , E_icode: 3 , M_icode= 1 , W_icode= 1, PC=                46
                  145
 clK:1 , D_icode: 1 , E_icode: 3 , M_icode= 1 , W_icode= 1, PC=                46
en_allocated->01100100
                  150
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 3 , W_icode= 1, PC=                47
                  155
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 3 , W_icode= 1, PC=                47
en_allocated->01100100
                  160
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 3, PC=                48
                  165
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 3, PC=                48
en_allocated->01100100
                  170
 clK:0 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                49
BYE-BYE

                  175
 clK:1 , D_icode: 1 , E_icode: 1 , M_icode= 1 , W_icode= 1, PC=                49
```

INITIAL REGISTER VALUES

```
r0:                   0
r1:                   1
 r2:                   2
 r3:                   3
 r4:                  60
 r5:                   5
 r6:                   6
 r7:                   7
 r8:                   8
 r9:                   9
 r10:                  10
 r11:                  11
 r12:                  12
 r13:                  13
 r14:                  14
```

FINAL UPDATED REGISTER VALUES

```
r0:                   0
r1:                   1
 r2:                   2
 r3:                   3
 r4:                  64
 r5:                   5
 r6:                   5
 r7:                   7
 r8:                   8
 r9:                   9
 r10:                  10
 r11:                  11
 r12:                  12
 r13:                  13
 r14:                  14
```