Intro To Processor Architecture

Project Report

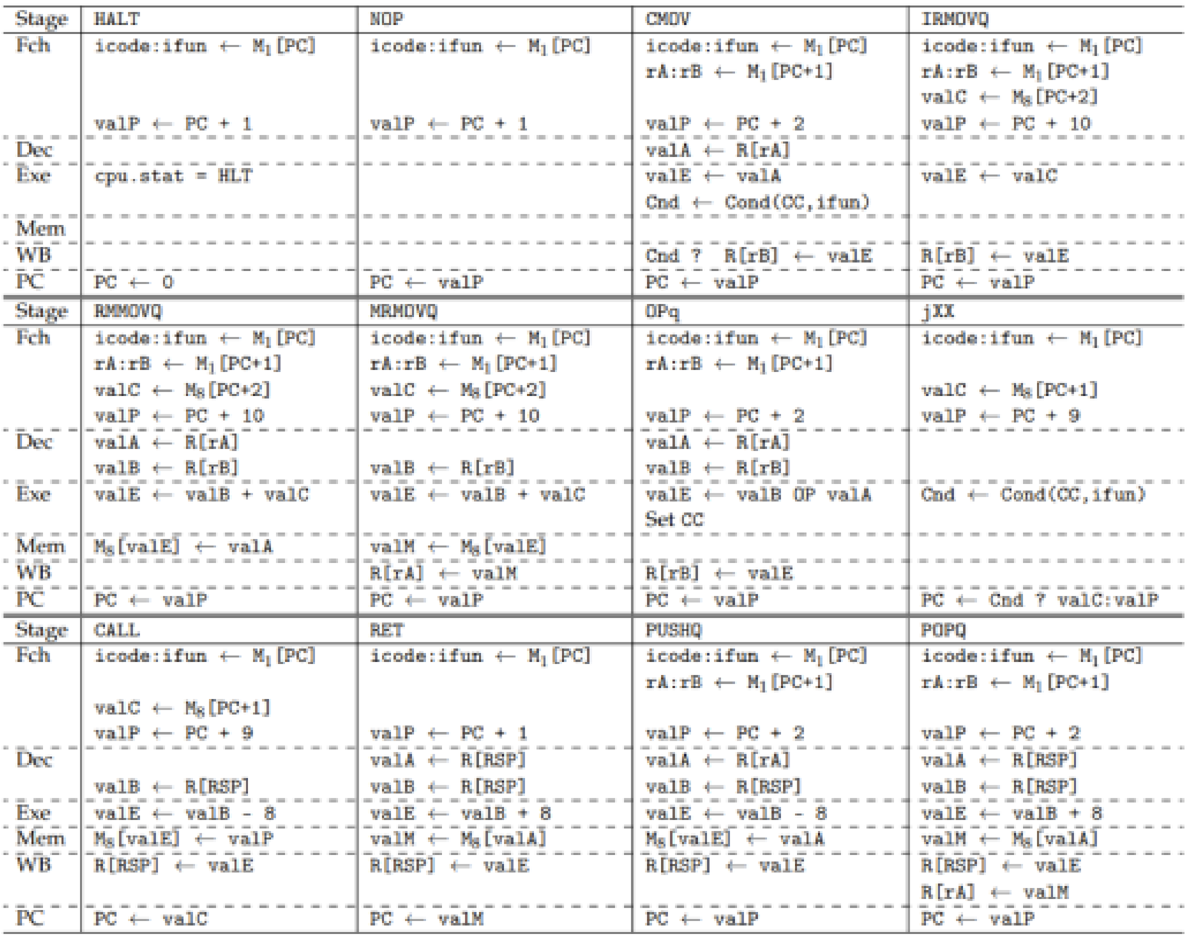
Team 11: Snapdragon

Revanth Sai 2020102068

Macharla Harish 2020102062

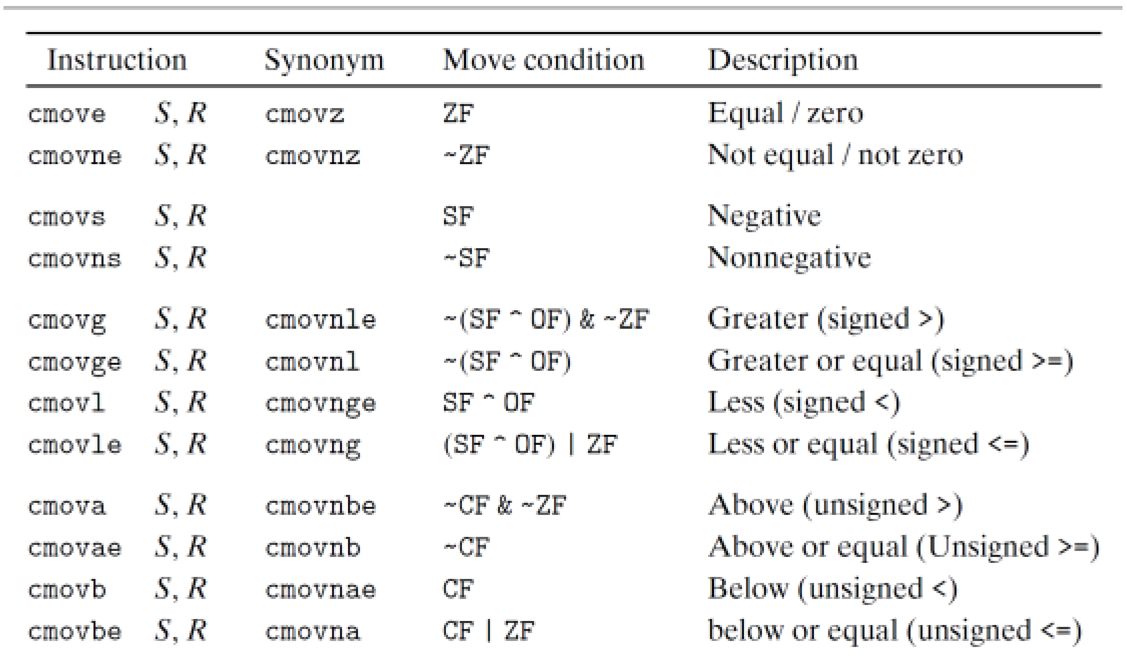
Report Details:

This report includes the sequential implementation of the Y86-64 processor which contains the fetch, decode - write back, execute, memory and the PC update blocks, their testbenches and the combined testbench.



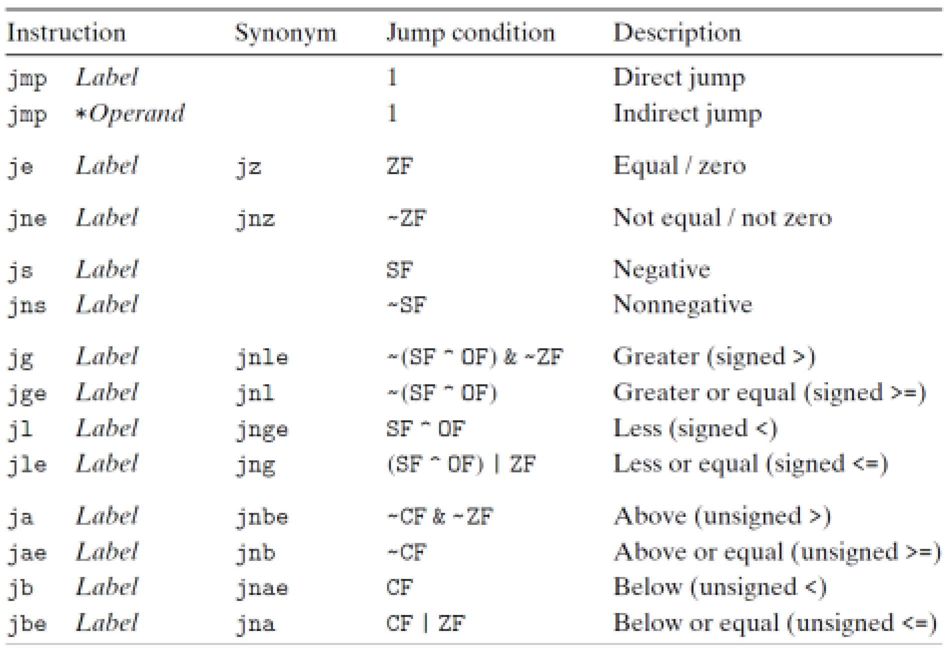
Stage	HALT	NOP	CMOV	IRMOVQ
Fch	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]
			rA:rB ← M_1[PC+1]	rA:rB ← M_1[PC+1]
				valC ← M_8[PC+2]
	valP ← PC + 1	valP ← PC + 1	valP ← PC + 2	valP ← PC + 10
Dec			valA ← R[rA]	
Exe	cpu.stat = HLT		valE ← valA	valE ← valC
			Cnd ← Cond(CC,ifun)	
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]	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]
	rA:rB ← M_1[PC+1]	rA:rB ← M_1[PC+1]	rA:rB ← M_1[PC+1]	
	valC ← M_8[PC+2]	valC ← M_8[PC+2]		valC ← M_8[PC+1]
	valP ← PC + 10	valP ← PC + 10	valP ← PC + 2	valP ← PC + 9
Dec	valA ← R[rA]		valA ← R[rA]	
	valB ← R[rB]	valB ← R[rB]	valB ← R[rB]	
Exe	valE ← valB + valC	valE ← valB + valC	valE ← valB OP valA	Cnd ← Cond(CC,ifun)
			Set CC	
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]	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]	icode:ifun ← M_1[PC]
			rA:rB ← M_1[PC+1]	rA:rB ← M_1[PC+1]
	valC ← M_8[PC+1]			
	valP ← PC + 9	valP ← PC + 1	valP ← PC + 2	valP ← PC + 2
Dec		valA ← R[RSP]	valA ← R[rA]	valA ← R[RSP]
	valB ← R[RSP]	valB ← R[RSP]	valB ← 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

Conditional Codes for Conditional Move statements:



Instruction		Synonym	Move condition	Description
`cmove`	*S*, *R*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *R*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *R*		`SF`	Negative
`cmovns`	*S*, *R*		`~SF`	Nonnegative
`cmovg`	*S*, *R*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *R*	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	*S*, *R*	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	*S*, *R*	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	*S*, *R*	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	*S*, *R*	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	*S*, *R*	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	*S*, *R*	`cmovna`	`CF | ZF`	below or equal (unsigned <=)

Conditional Codes for Jump statements:



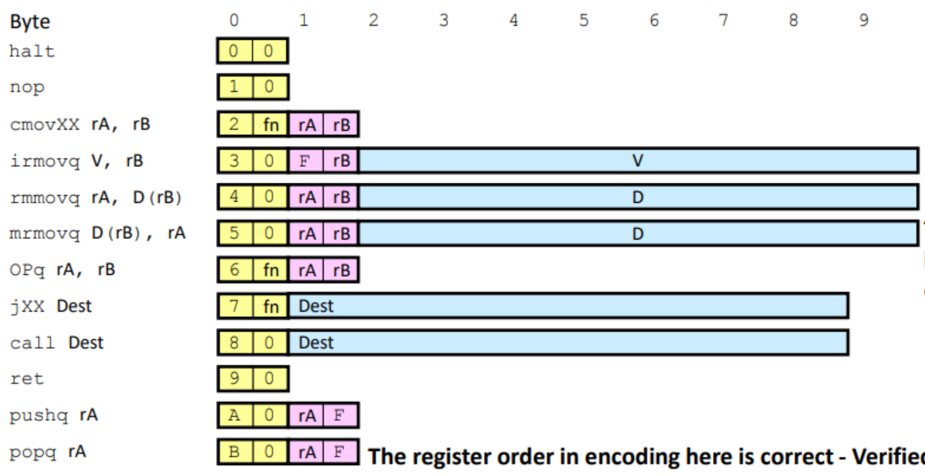
Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	`ZF`	Equal / zero
`jne`	*Label*	`jnz`	`~ZF`	Not equal / not zero
`js`	*Label*		`SF`	Negative
`jns`	*Label*		`~SF`	Nonnegative
`jg`	*Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater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

Arithmetic and Logical Operation:

Two Operand Instructions:

Format		*Computation*
addq	*Src,Dest*	Dest = Dest + Src
subq	*Src,Dest*	Dest = Dest – Src
imulq	*Src,Dest*	Dest = Dest * Src
salq	*Src,Dest*	Dest = Dest << Src
sarq	*Src,Dest*	Dest = Dest >> Src
shrq	*Src,Dest*	Dest = Dest >> Src
xorq	*Src,Dest*	Dest = Dest ^ Src
andq	*Src,Dest*	Dest = Dest & Src
orq	*Src,Dest*	Dest = Dest | Src

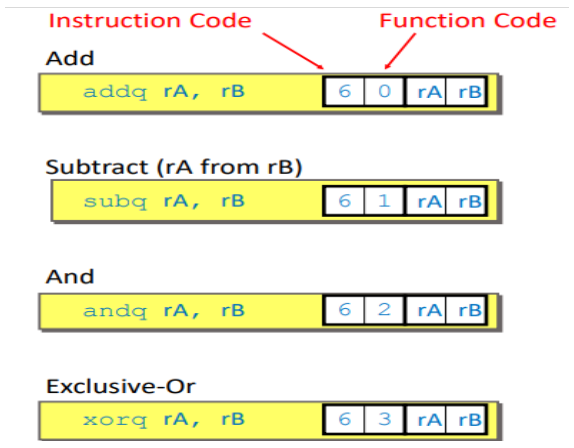
Y86-64 Instruction Set



The register order in encoding here is correct - Verified

The register order in encoding here is correct - Verified

Arithmetic and Logical Operations:



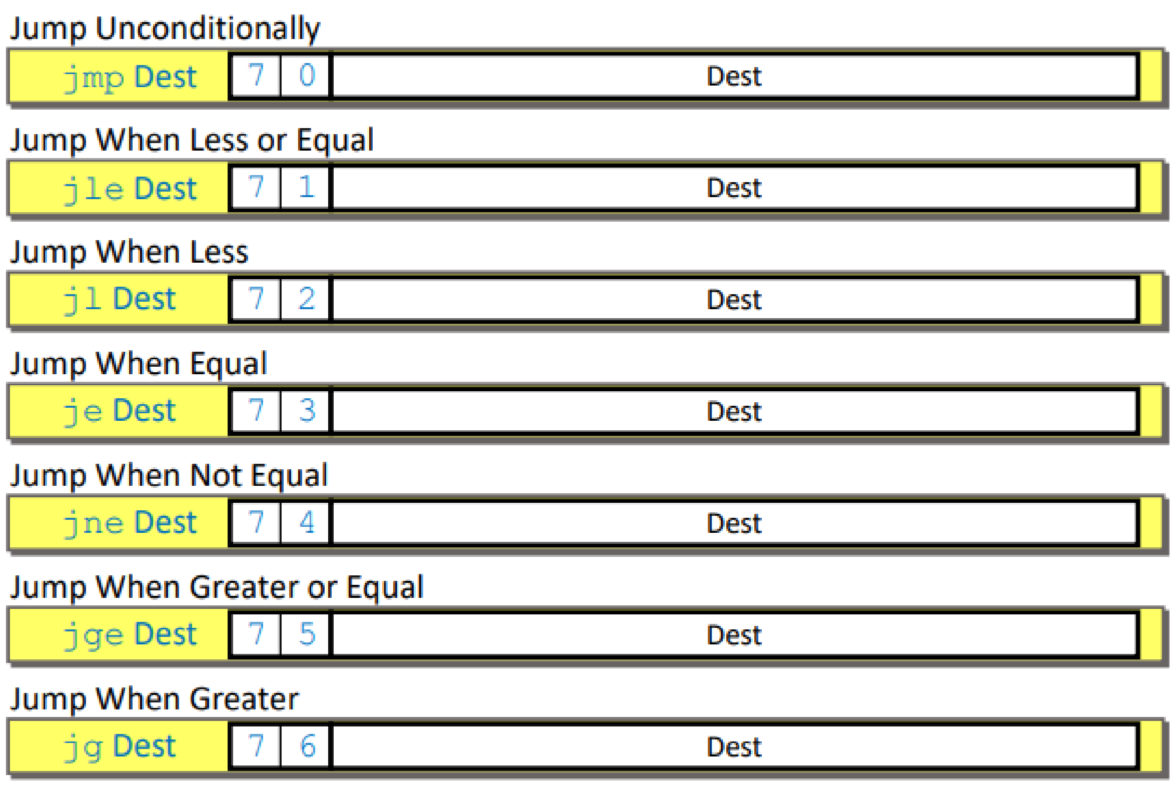
Move Operations:

`rrmovq` rA, rB	2	0				Register ➔ Register
`irmovq` V, rB	3	0	F	rB	V	Immediate ➔ Register
`rmmovq` rA, D (rB)	4	0	rA	rB	D	Register ➔ Memory
`mrmovq` D (rB), rA	5	0	rA	rB	D	Memory ➔ Register

Conditional Move Instructions:

Move Unconditionally	`rrmovq` rA, rB	2	0	rA	rB
Move When Less or Equal	`cmovle` rA, rB	2	1	rA	rB
Move When Less	`cmovl` rA, rB	2	2	rA	rB
Move When Equal	`cmove` rA, rB	2	3	rA	rB
Move When Not Equal	`cmovne` rA, rB	2	4	rA	rB
Move When Greater or Equal	`cmovge` rA, rB	2	5	rA	rB
Move When Greater	`cmovg` rA, rB	2	6	rA	rB

JUMP instructions



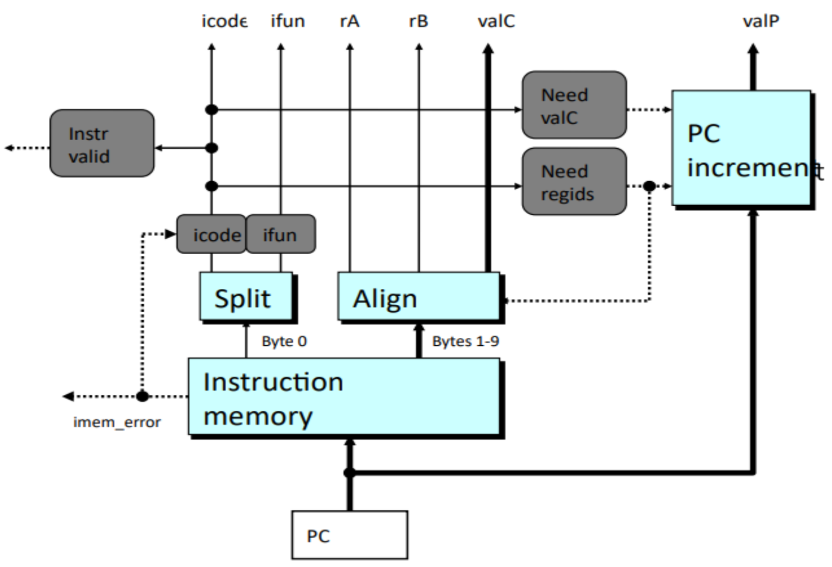
Sequential Y86 Instruction Stages

Each instruction sequentially goes through following common stages:

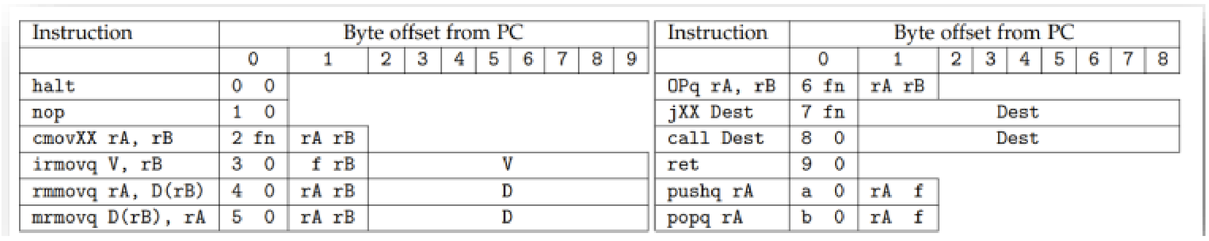
- Fetch
- Decode
- Execute
- Memory
- Write-back
- PC update

The blocks are as follows:

Fetch:



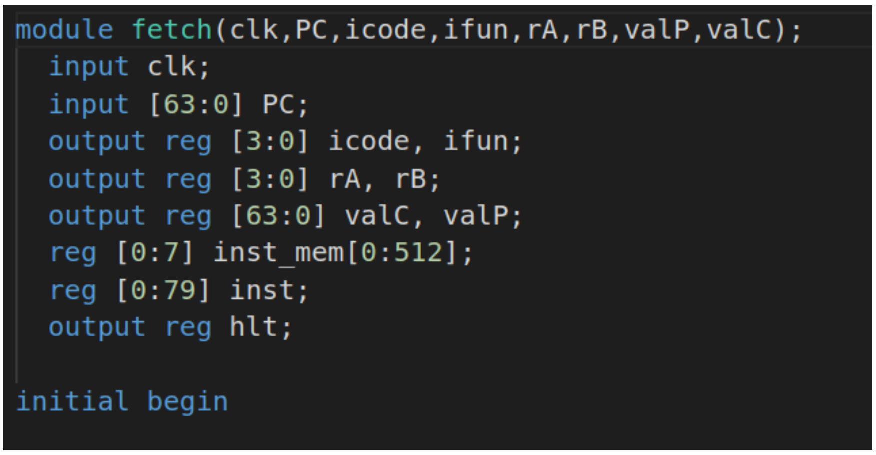
The fetch block fetches the instruction from the instruction memory which is pointed by the PC as an input and it encodes the instruction into **icode, ifun**, registers **rA** and **rB**, **valC, valP**. PC incrementation is also done in the fetch block based on the instruction size. The first 8 bits in the instruction pointed by the PC is decoded as the icode(first 4 bits) and ifun (next four bits). Next 8 bits represents the registers rA and rB. Based the operation there may be valC or not.



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							

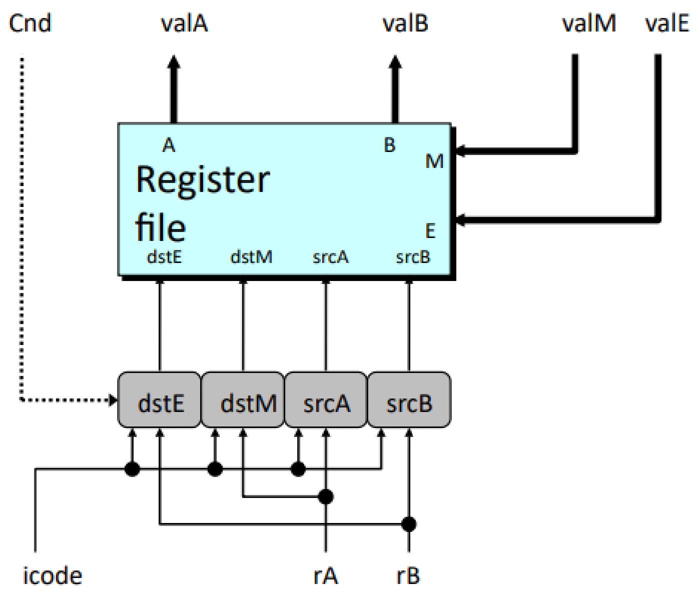
If a fetch statement encounters halt instruction, then entire execution is halted. If it encounters the nop instruction then it sits idle for one clock cycle and continues with the next instruction pointed by the PC.

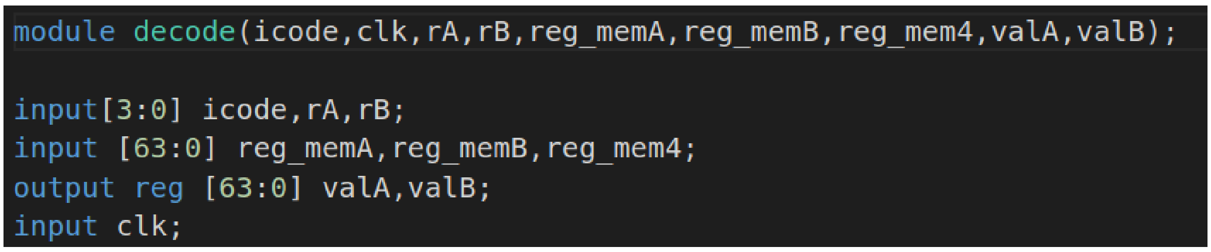


```
module fetch(clk,PC,icode,ifun,rA,rB,valP,valC);
  input clk;
  input [63:0] PC;
  output reg [3:0] icode, ifun;
  output reg [3:0] rA, rB;
  output reg [63:0] valC, valP;
  reg [0:7] inst_mem[0:512];
  reg [0:79] inst;
  output reg hlt;

initial begin
```

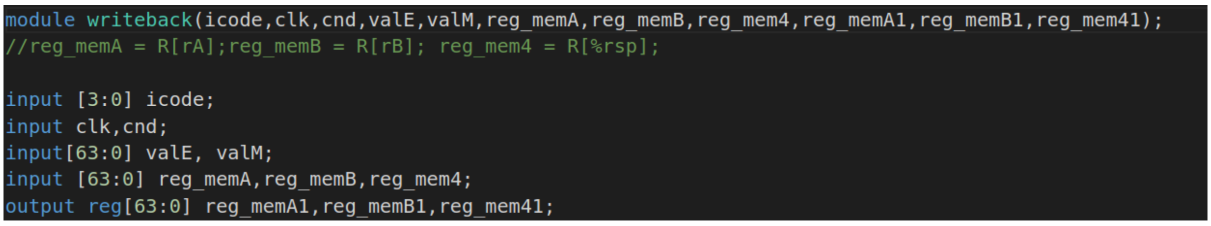
Decode and writeback:

Decode block is used to read the values **valA** and **valB** from the registers rA and rB. Writeback block is used to writeback the values into registers based on the cnd. valA and valB are the outputs of the decode block and they are written back in the respective registers based on the cnd in the writeback stage.





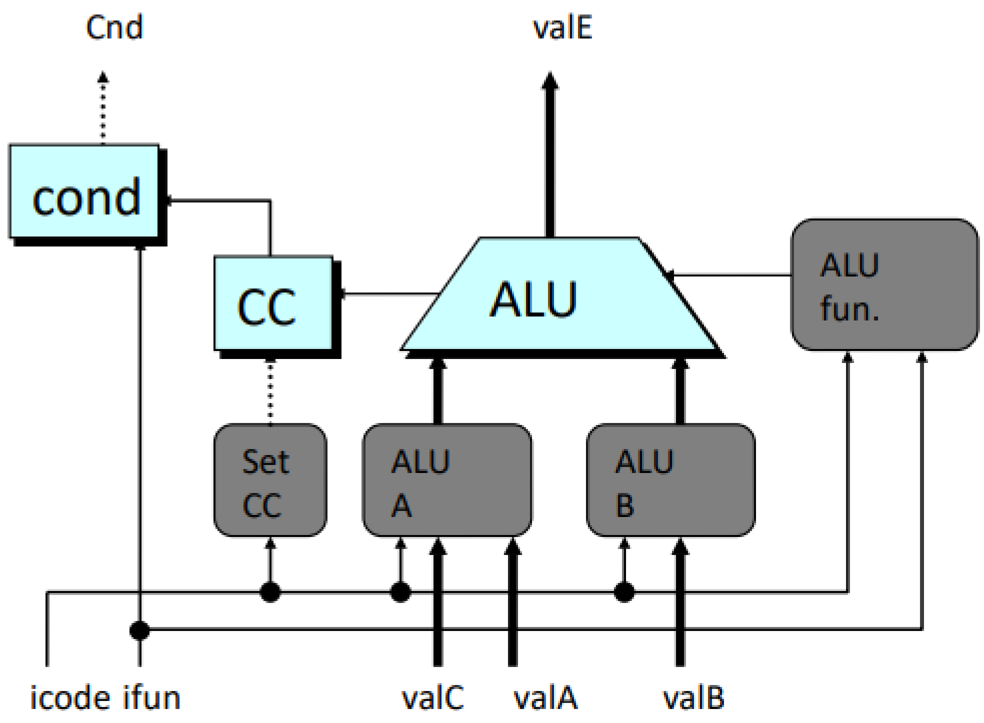
```
module decode(icode,clk,rA,rB,reg_memA,reg_memB,reg_mem4,valA,valB);

input[3:0] icode,rA,rB;
input [63:0] reg_memA,reg_memB,reg_mem4;
output reg [63:0] valA,valB;
input clk;
```

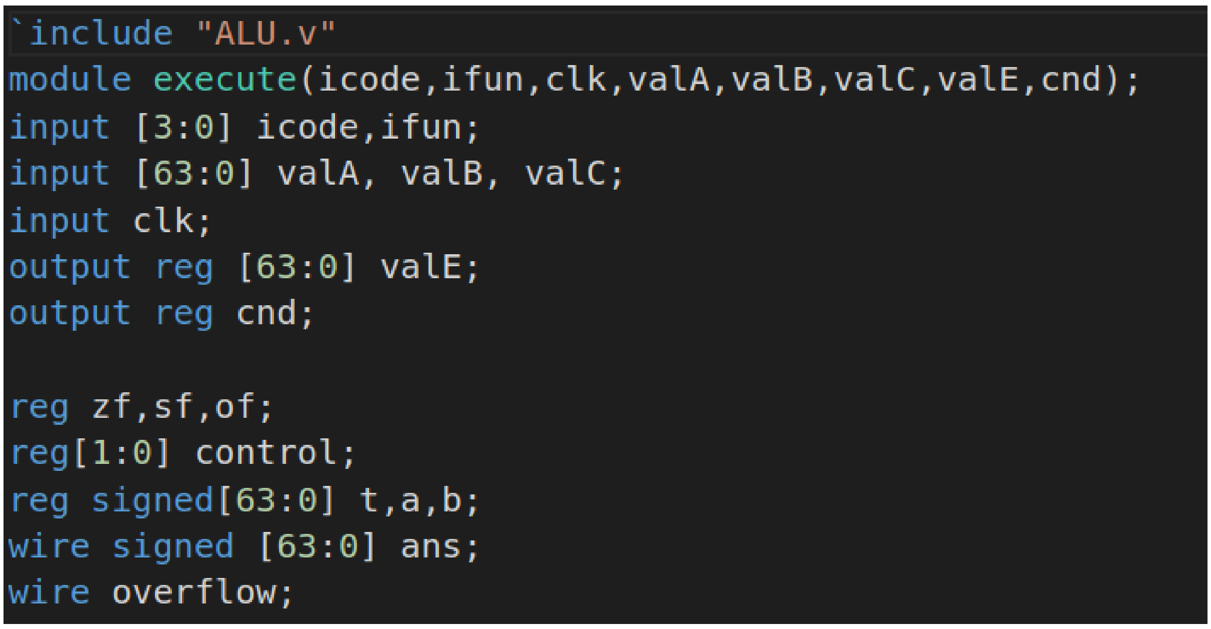


```
module writeback(icode,clk,cnd,valE,valM,reg_memA,reg_memB,reg_mem4,reg_memA1,reg_memB1,reg_mem41);
//reg_memA = R[rA];reg_memB = R[rB]; reg_mem4 = R[%rsp];

input [3:0] icode;
input clk,cnd;
input[63:0] valE, valM;
input [63:0] reg_memA,reg_memB,reg_mem4;
output reg[63:0] reg_memA1,reg_memB1,reg_mem41;
```

Execute:

Execute block is to implements the given instruction. Effective address is also computed here. Basic ALU operations and conditions codes are done here. The execute block takes **icode, ifun, valC, valA** and **valB** as inputs and gives the outputs as **Cnd** and **valE**. The condition codes are also set for the instructions **jXX** and **CMOV** and **valE** is computed using ALU.

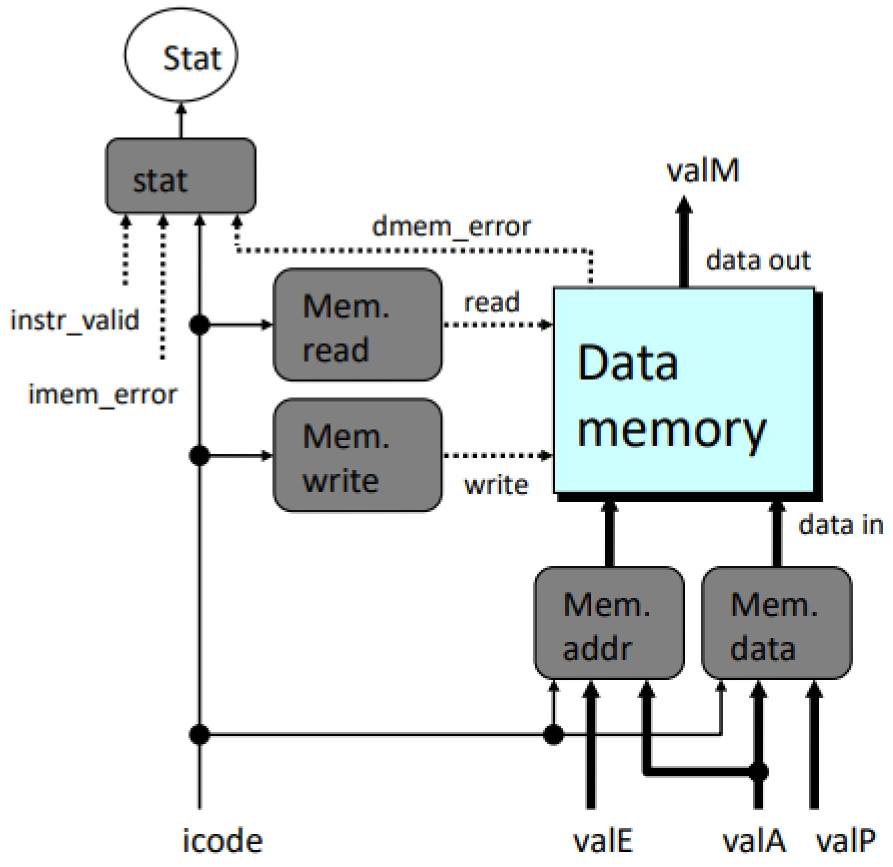


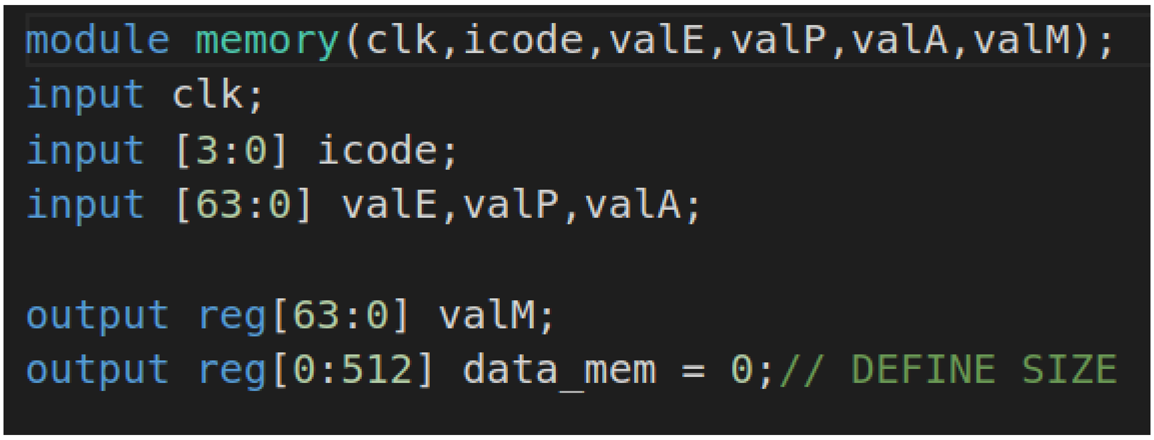


```
`include "ALU.v"
module execute(icode,ifun,clk,valA,valB,valC,valE,cnd);
input [3:0] icode,ifun;
input [63:0] valA, valB, valC;
input clk;
output reg [63:0] valE;
output reg cnd;

reg zf,sf,of;
reg[1:0] control;
reg signed[63:0] t,a,b;
wire signed [63:0] ans;
wire overflow;
```

Memory:

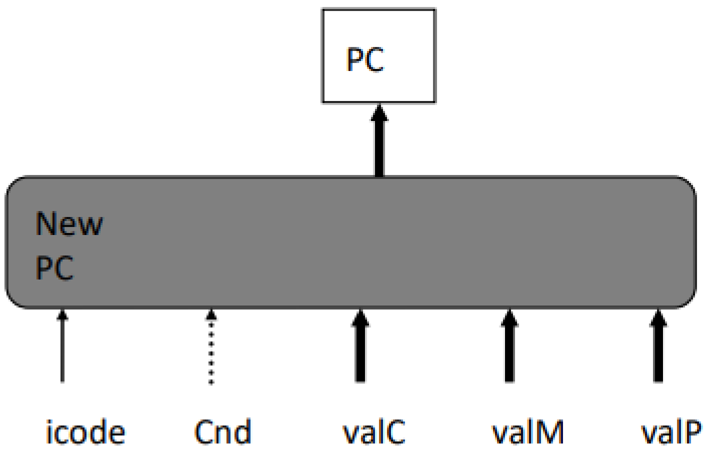
The purpose of memory block is to read and write the values from the memory. The memory block has input values as **icode, valE, valA and valP** and the output values as **valM** • The value of valM is read from the registers and the vice versa according to the value of icode where the instruction is selected using switch statements.

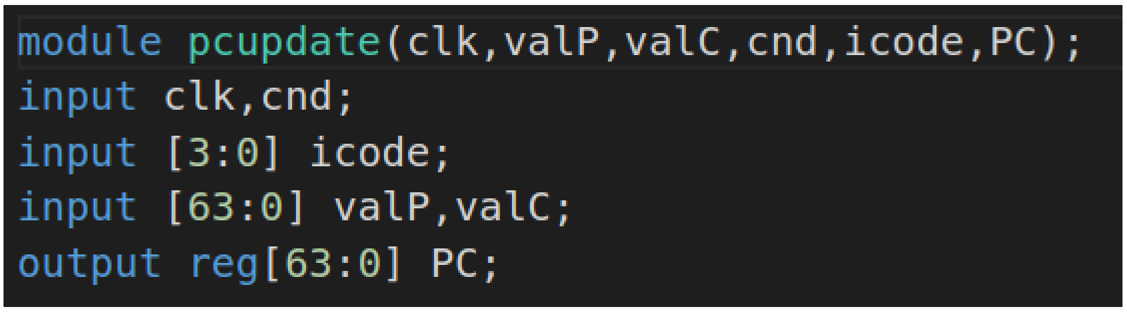



```
module memory(clk,icode,valE,valP,valA,valM);
input clk;
input [3:0] icode;
input [63:0] valE,valP,valA;

output reg[63:0] valM;
output reg[0:512] data_mem = 0;// DEFINE SIZE
```

PC-UPDATE:

The PC update is to point the PC 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 updated PC value can be **valM, valC, valP**.





```
module pcupdate(clk,valP,valC,cnd,icode,PC);
input clk,cnd;
input [3:0] icode;
input [63:0] valP,valC;
output reg[63:0] PC;
```

TestBench Results:

Initially,

reg_mem[0]=64'd0;

reg_mem[1]=64'd1;

reg_mem[2]=64'd2;

reg_mem[3]=64'd3;

reg_mem[4]=64'd4;

reg_mem[5]=64'd5;

reg_mem[6]=64'd6;

reg_mem[7]=64'd7;

reg_mem[8]=64'd8;

reg_mem[9]=64'd9;

reg_mem[10]=64'd10;

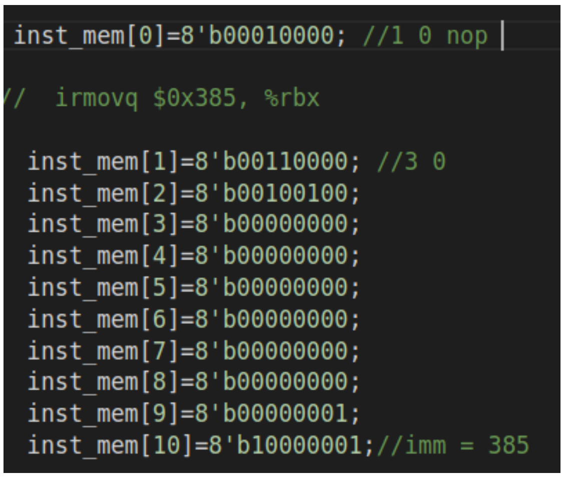
reg_mem[11]=64'd11;

reg_mem[12]=64'd12;

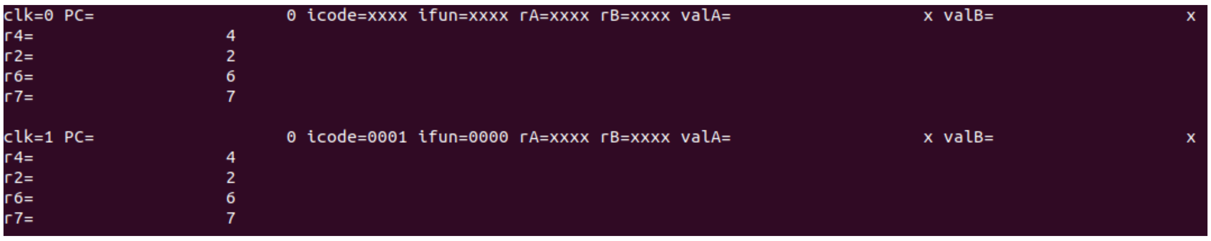
reg_mem[13]=64'd13;

reg_mem[14]=64'd14;

reg_mem[15]=64'd15;

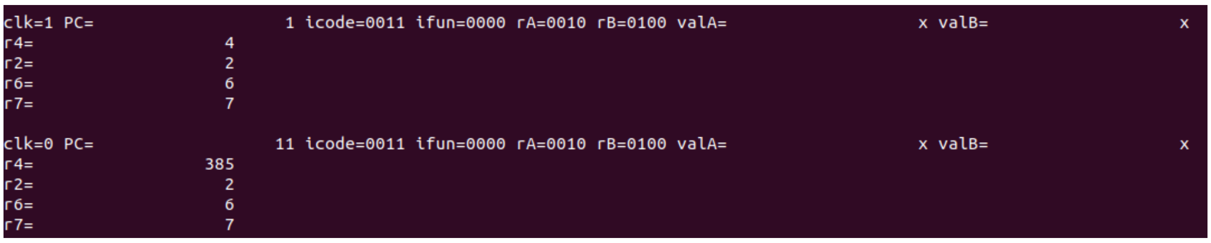


```
inst_mem[0]=8'b00010000; //1 0 nop

//  irmovq $0x385, %rbx

 inst_mem[1]=8'b00110000; //3 0
 inst_mem[2]=8'b00100100;
 inst_mem[3]=8'b00000000;
 inst_mem[4]=8'b00000000;
 inst_mem[5]=8'b00000000;
 inst_mem[6]=8'b00000000;
 inst_mem[7]=8'b00000000;
 inst_mem[8]=8'b00000000;
 inst_mem[9]=8'b00000001;
 inst_mem[10]=8'b10000001;//imm = 385
```



```
clk=0 PC=                    0 icode=xxxx ifun=xxxx rA=xxxx rB=xxxx valA=                    x valB=                    x
r4=                    4
r2=                    2
r6=                    6
r7=                    7

clk=1 PC=                    0 icode=0001 ifun=0000 rA=xxxx rB=xxxx valA=                    x valB=                    x
r4=                    4
r2=                    2
r6=                    6
r7=                    7
```

We can see that the first instruction is nop. So, the PC will increment to 1.

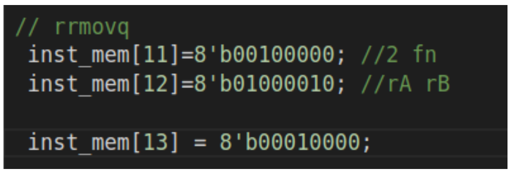


```
clk=1 PC=                    1 icode=0011 ifun=0000 rA=0010 rB=0100 valA=                    x valB=                    x
r4=                    4
r2=                    2
r6=                    6
r7=                    7

clk=0 PC=                   11 icode=0011 ifun=0000 rA=0010 rB=0100 valA=                    x valB=                    x
r4=                  385
r2=                    2
r6=                    6
r7=                    7
```

During the next positive edge of the clock cycle, we can see the next instruction is immediate to register move. PC will increment by 10.

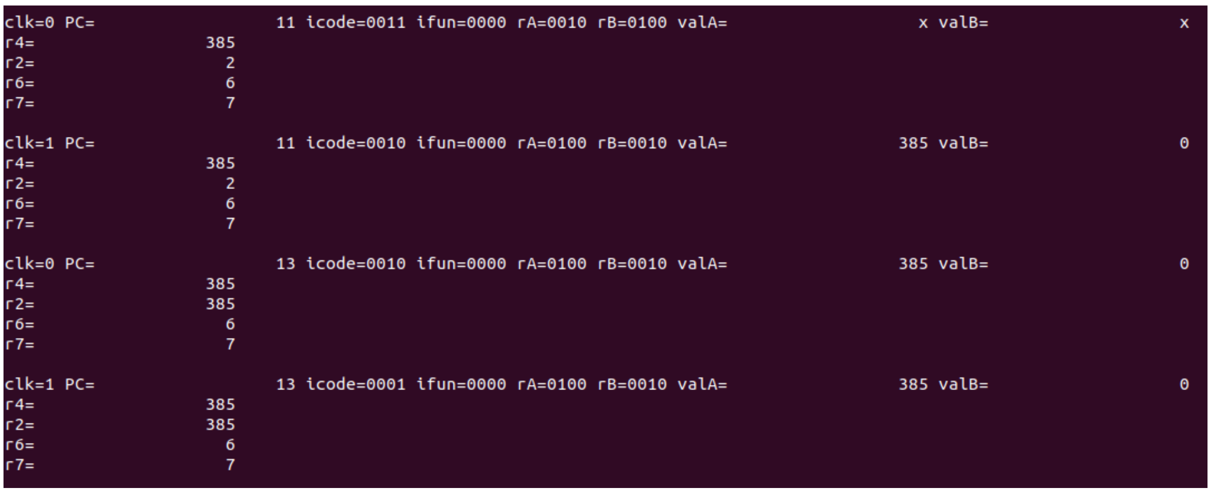
icode = 0011, ifun = 0000, rA = 0010 rB = 0100 and next 8 bytes is valC = 385(here).

R[rB] = valC. => R[0010] = valC => r4 = 385. As we can see in the above figure.



```
// rrmovq
 inst_mem[11]=8'b00100000; //2 fn
 inst_mem[12]=8'b01000010; //rA rB

 inst_mem[13] = 8'b00010000;
```

In the next positive cycle, a new instruction is fetched. Let’s say reg to reg move and followed by the nop instruction.

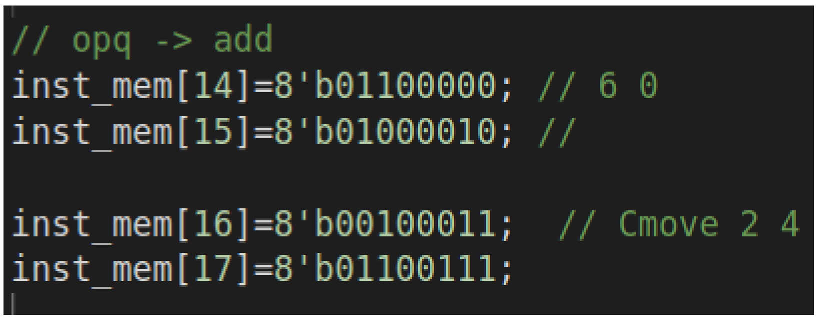


```
clk=0 PC=                  11 icode=0011 ifun=0000 rA=0010 rB=0100 valA=                   x valB=                   x
r4=                   385
r2=                     2
r6=                     6
r7=                     7

clk=1 PC=                  11 icode=0010 ifun=0000 rA=0100 rB=0010 valA=                 385 valB=                   0
r4=                   385
r2=                     2
r6=                     6
r7=                     7

clk=0 PC=                  13 icode=0010 ifun=0000 rA=0100 rB=0010 valA=                 385 valB=                   0
r4=                   385
r2=                   385
r6=                     6
r7=                     7

clk=1 PC=                  13 icode=0001 ifun=0000 rA=0100 rB=0010 valA=                 385 valB=                   0
r4=                   385
r2=                   385
r6=                     6
r7=                     7
```

When PC = 11; reg to reg move => icode = 0011, rA = 0010, rB = 0100

R[rA] = R [0011] = 385; => R[rB] = R[0010] = R[rA] = 385;

The values are updated in the next cycle as we can see from the above result and PC is incremented by one.



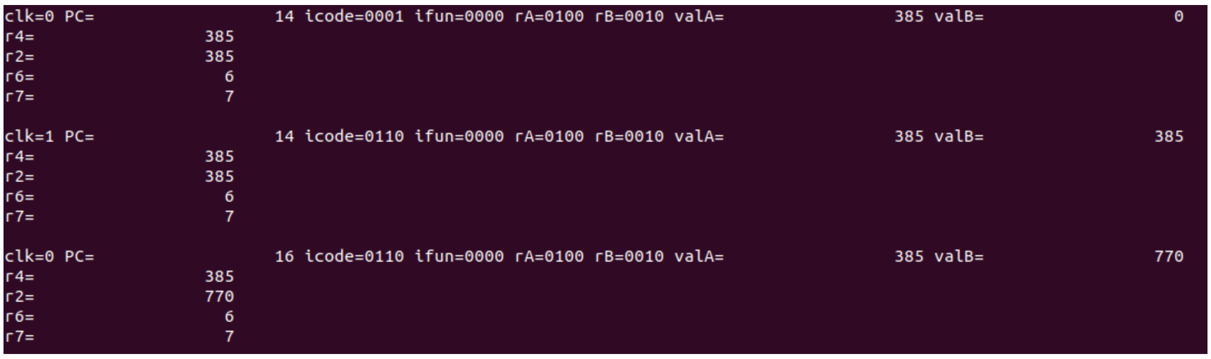
```
// opq -> add
inst_mem[14]=8'b01100000; // 6 0
inst_mem[15]=8'b01000010; //

inst_mem[16]=8'b00100011;  // Cmove 2 4
inst_mem[17]=8'b01100111;
```

In next clock cycle it fetches another instruction as shown, here it is add operation and PC is incremented by twice.

Icode = 0110 ifun = 0000. rA = 0110 rB = 0010.

R[rA] = 385; R[rB] = 385.

R[rB] = R[rA] + R[rB] = 770.

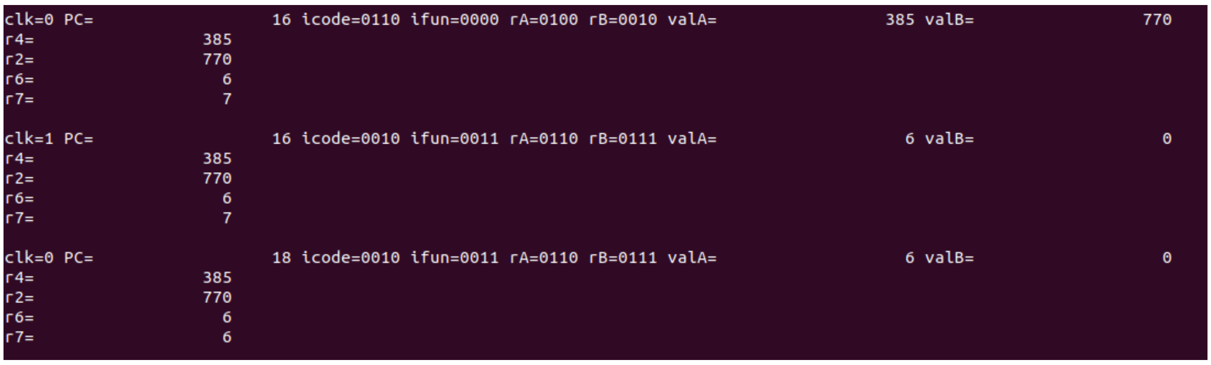

```
clk=0 PC=                  14 icode=0001 ifun=0000 rA=0100 rB=0010 valA=                 385 valB=                   0
r4=                  385
r2=                  385
r6=                    6
r7=                    7

clk=1 PC=                  14 icode=0110 ifun=0000 rA=0100 rB=0010 valA=                 385 valB=                 385
r4=                  385
r2=                  385
r6=                    6
r7=                    7

clk=0 PC=                  16 icode=0110 ifun=0000 rA=0100 rB=0010 valA=                 385 valB=                 770
r4=                  385
r2=                  770
r6=                    6
r7=                    7
```

Next instruction is fetched let’s say it is conditional move icode = 0010 ifun = 0011

rA = 0110 rB = 0111;

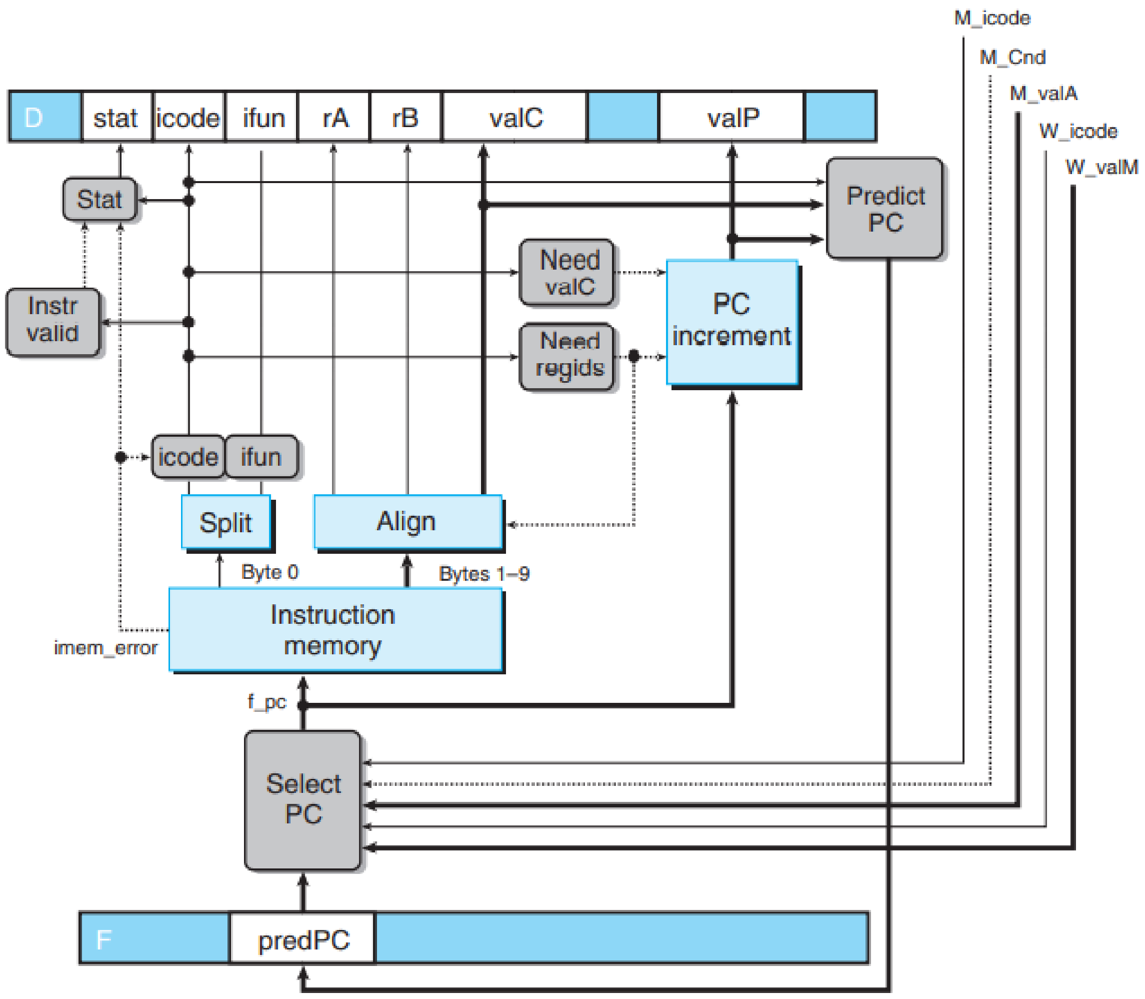
valA = 6; valB = 0.

R[rB] = R[rA] = 6.


```
clk=0 PC=                  16 icode=0110 ifun=0000 rA=0100 rB=0010 valA=                 385 valB=                 770
r4=                  385
r2=                  770
r6=                    6
r7=                    7

clk=1 PC=                  16 icode=0010 ifun=0011 rA=0110 rB=0111 valA=                   6 valB=                   0
r4=                  385
r2=                  770
r6=                    6
r7=                    7

clk=0 PC=                  18 icode=0010 ifun=0011 rA=0110 rB=0111 valA=                   6 valB=                   0
r4=                  385
r2=                  770
r6=                    6
r7=                    6
```

Pipelining Of Y86-64 Processor

Fetch and PC increment



Fetch includes instruction memory hardware unit from where it reads the 10 bytes from the memory at a time, using the PC as the address of the first byte. The first byte of these 10 bytes indicates the instruction type and this first byte is split into 2blocks of 4 bits. The first four bits gives the **"icode"** and next four bits "**ifun**". Figure above provides a detailed view of the PIPE fetch stage logic. As discussed earlier, this stage must also select a current value for the program counter and predict the next PC value. The hardware units for reading the instruction from memory and for extracting the different instruction fields are the same as those we considered for SEQ. The PC selection logic chooses between three program counter sources. As a mis predicted branch enters the memory stage, the value of valP for this instruction (indicating the address of the following instruction) is read from pipeline register M (signal M_valA). When a ret instruction enters the write-back stage, the return address is read from pipeline register W (signal W_valM). All other cases use the predicted value of the

PC, stored in pipeline register F (signal F_predPC).

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;
];
```

Decode and Writeback

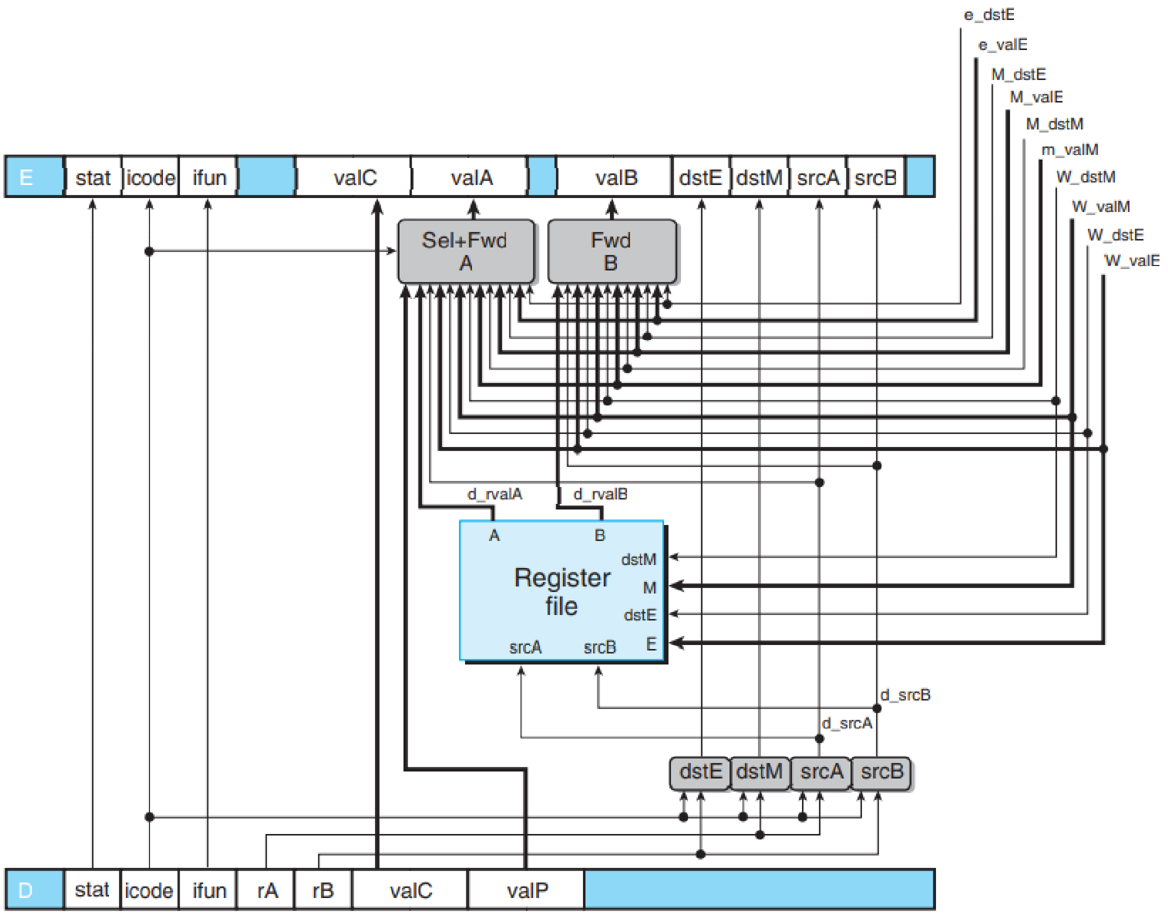
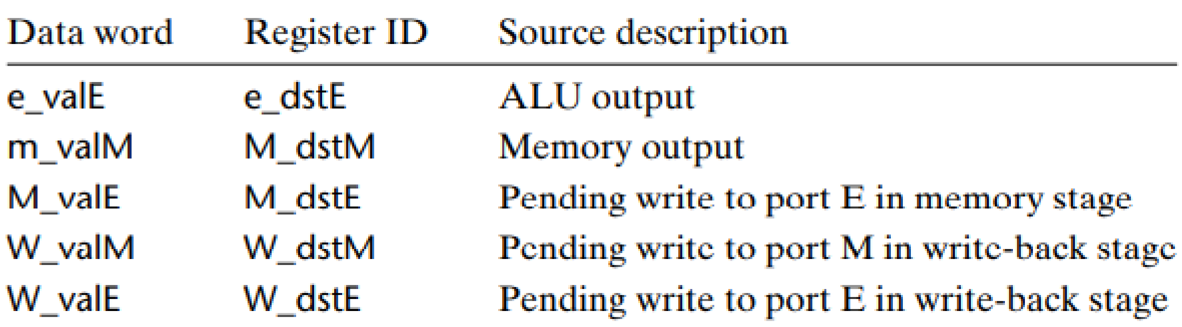


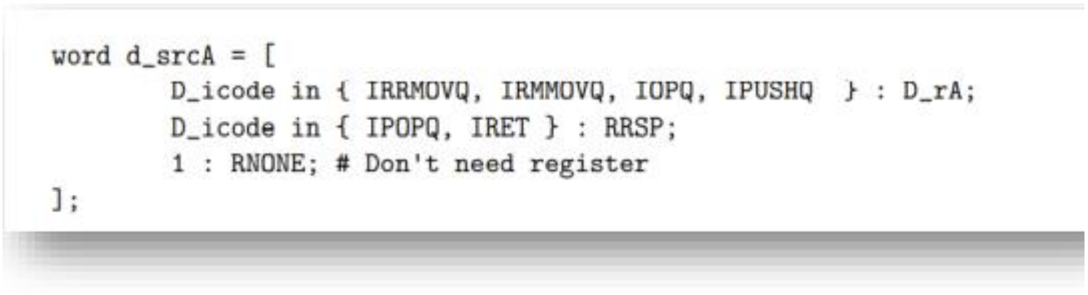
Figure below gives a detailed view of the decode and write-back logic for PIPE. The blocks labeled dstE, dstM, srcA, and srcB are very similar to their counterparts in the implementation of SEQ. Observe that the register IDs supplied to the write ports come from the write-back stage (signals W_dstE and W_dstM), rather than from the decode stage. This is because we want the writes to occur to the destination registers specified by the instruction in the write-back stage.

Most of the complexity of this stage is associated with the forwarding logic. As mentioned earlier, the block labeled “Sel+Fwd A” serves two roles. It merges the valP signal into the valA signal for later stages in order to reduce the amount of state in the pipeline register. It also implements the forwarding logic for source operand valA. The merging of signals valA and valP exploits the fact that only the call and jump instructions need the value of valP in later stages, and these instructions do not need the value read from the A port of the register file. This selection is controlled by the icode signal for this stage. When signal D_icode matches the instruction code for either call or jXX, this block should select D_valP as its output.

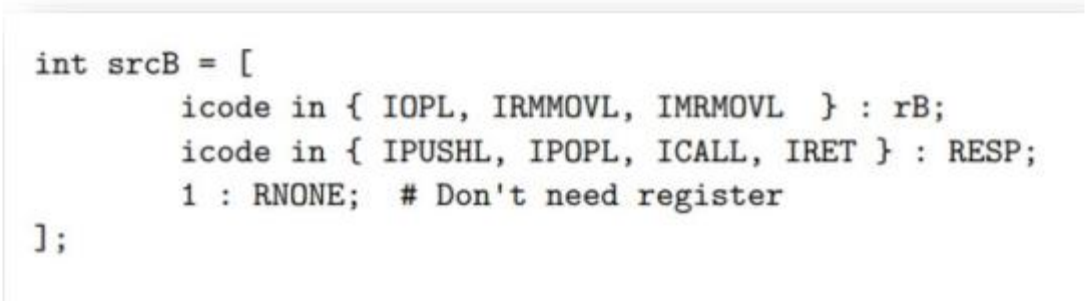


Data word	Register ID	Source description
e_valE	e_dstE	ALU output
m_valM	M_dstM	Memory output
M_valE	M_dstE	Pending write to port E in memory stage
W_valM	W_dstM	Pending write to port M in write-back stage
W_valE	W_dstE	Pending write to port E in write-back stage

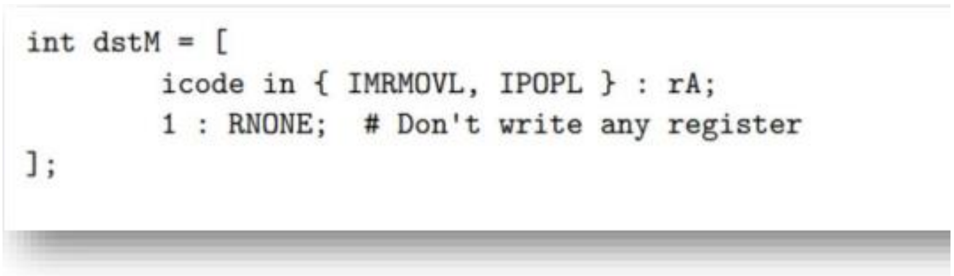
d_srcA, d_srcB, d_dstE and **d_dstM** are computed based on the following:



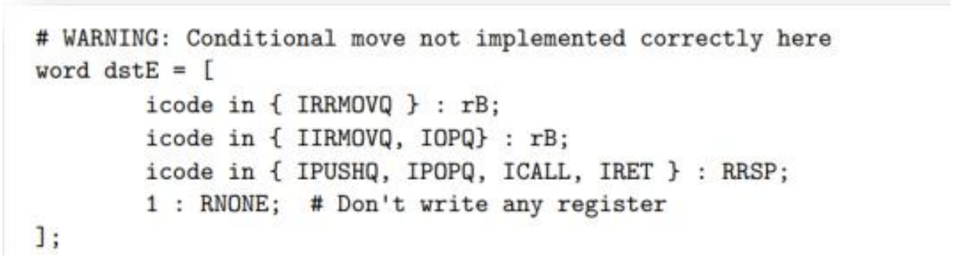
```
word d_srcA = [
        D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : D_rA;
        D_icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];
```



```
int srcB = [
        icode in { IOPL, IRMMOVL, IMRMOVL  } : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];
```

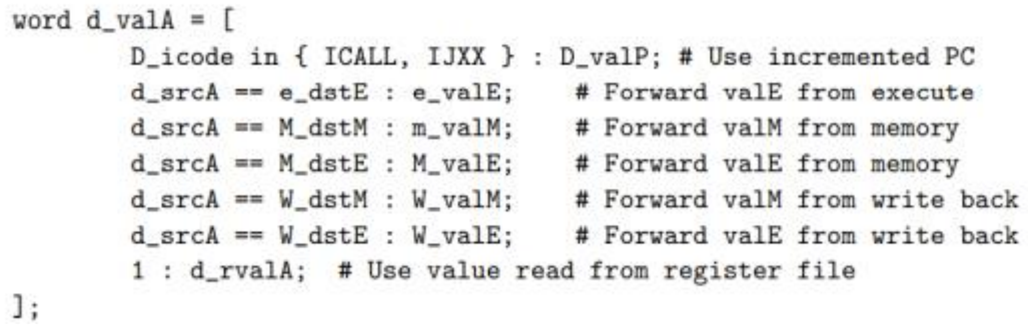


```
int dstM = [
        icode in { IMRMOVL, IPOPL } : rA;
        1 : RNONE;  # Don't write any register
];
```



```
# WARNING: Conditional move not implemented correctly here
word dstE = [
        icode in { IRRMOVQ } : rB;
        icode in { IIRMOVQ, IOPQ} : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];
```

Conditions used for Data Forwarding



```
word d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == e_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];
```

Execute

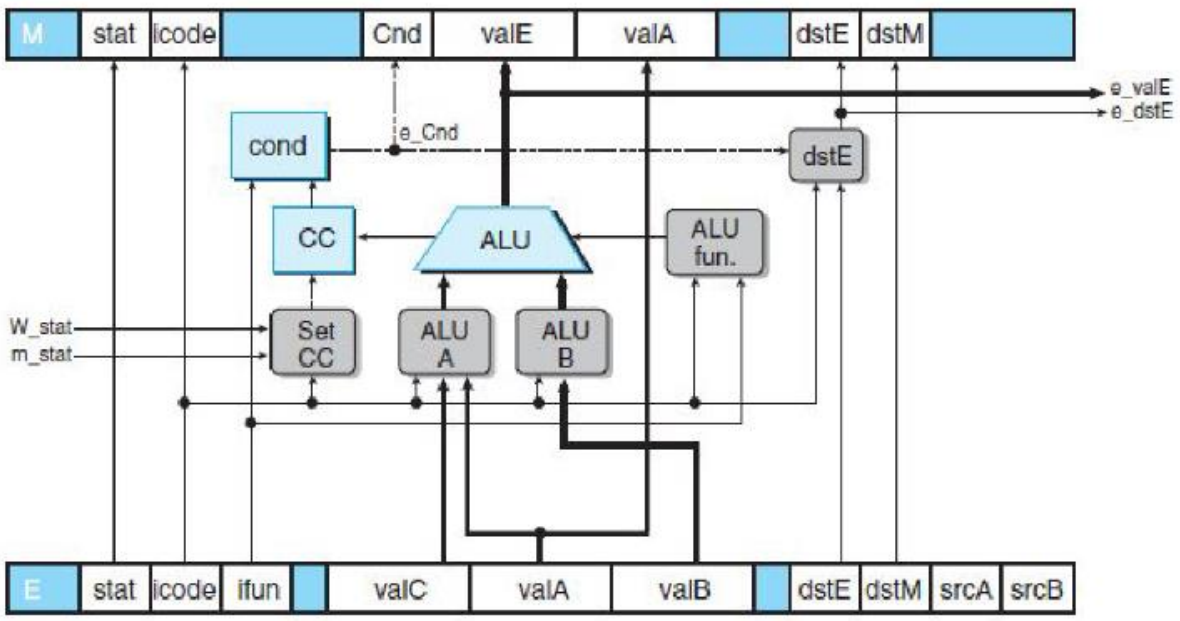
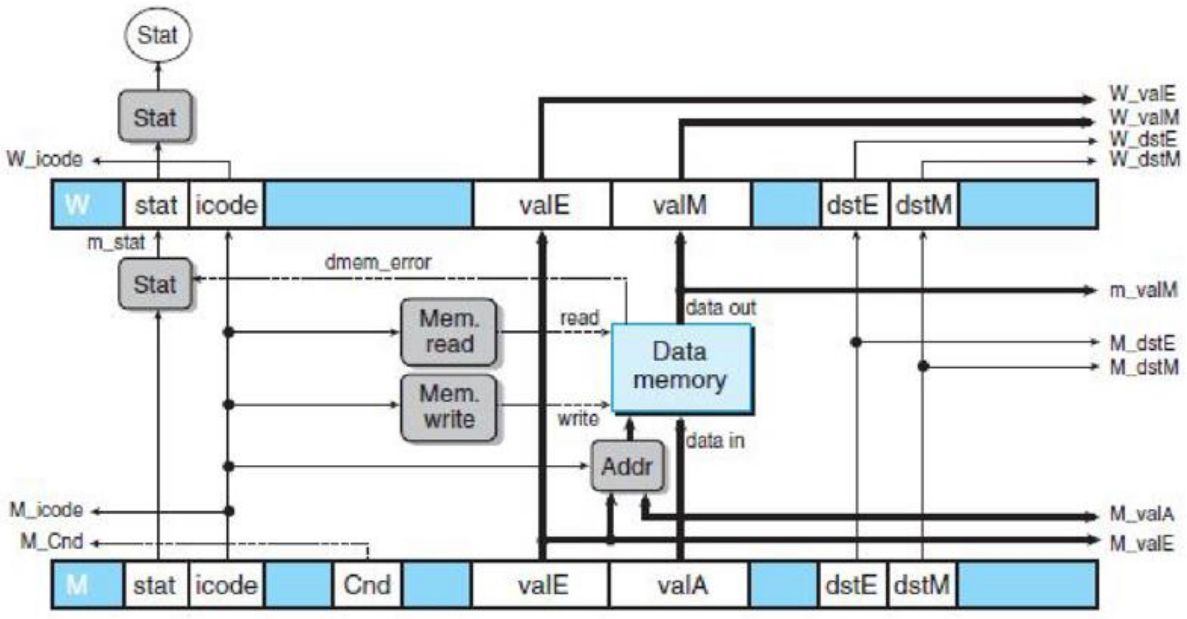


Figure above shows the execute stage logic for PIPE. The hardware units and the logic blocks are identical to those in SEQ, with an appropriate renaming of signals. We can see the signals e_valE and e_dstE directed toward the decode stage as one of the forwarding sources. One difference is that the logic labeled “Set CC,” which determines whether or not to update the condition codes, has signals m_stat and W_stat as inputs. These signals are used to detect cases where an instruction causing an exception is passing through later pipeline stages, and therefore any updating of the condition codes should be suppressed.

Memory

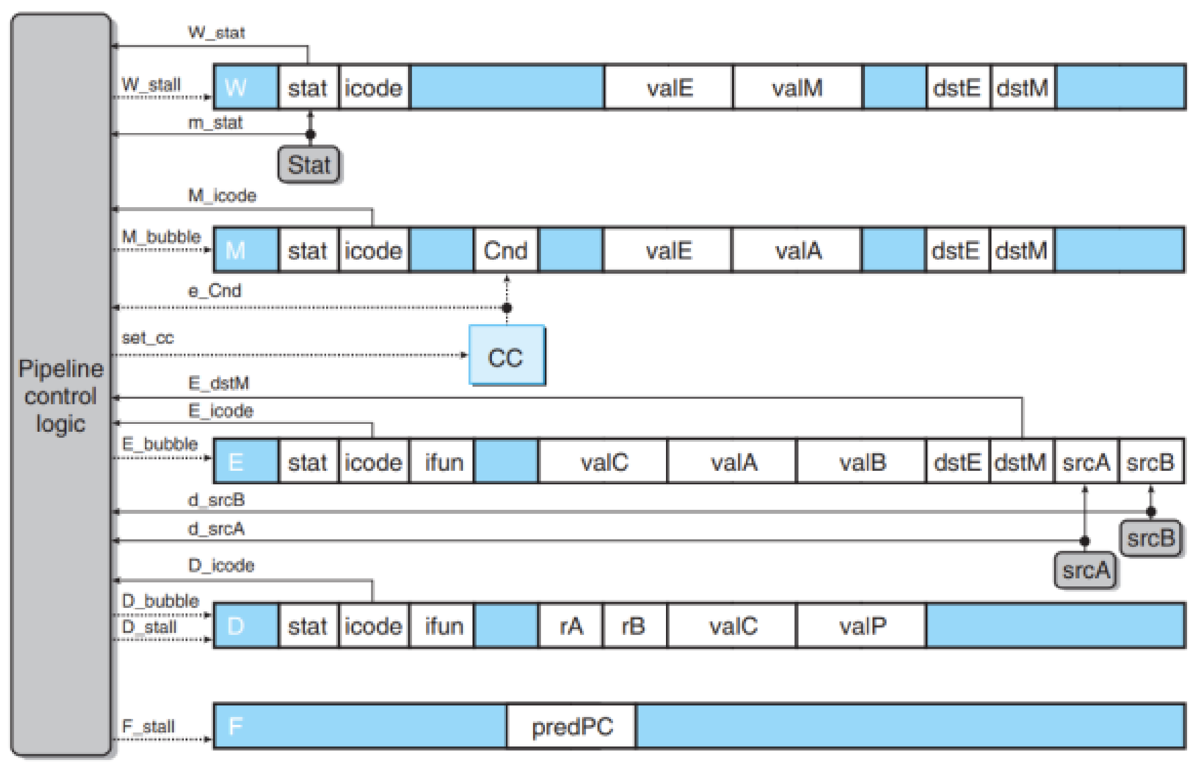


The memory has the task of either reading or writing program data. Two control blocks generate the values for the memory address and the memory input data for write operations. Two other blocks generate the control signals indicating whether to perform a read or write operation. When a read operation is performed the data memory generates the valM. We set the control signals mem_Read only for instructions that read data from memory. Many of the values in pipeline registers and M and W are supplied to other parts of the circuits as the part of forwarding and pipeline control logic.

Note: As we can see, the functions used are modified for also eliminating the data and control hazards which requires a pipeline control logic which is implemented in a separate module which works according to the following logic

	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

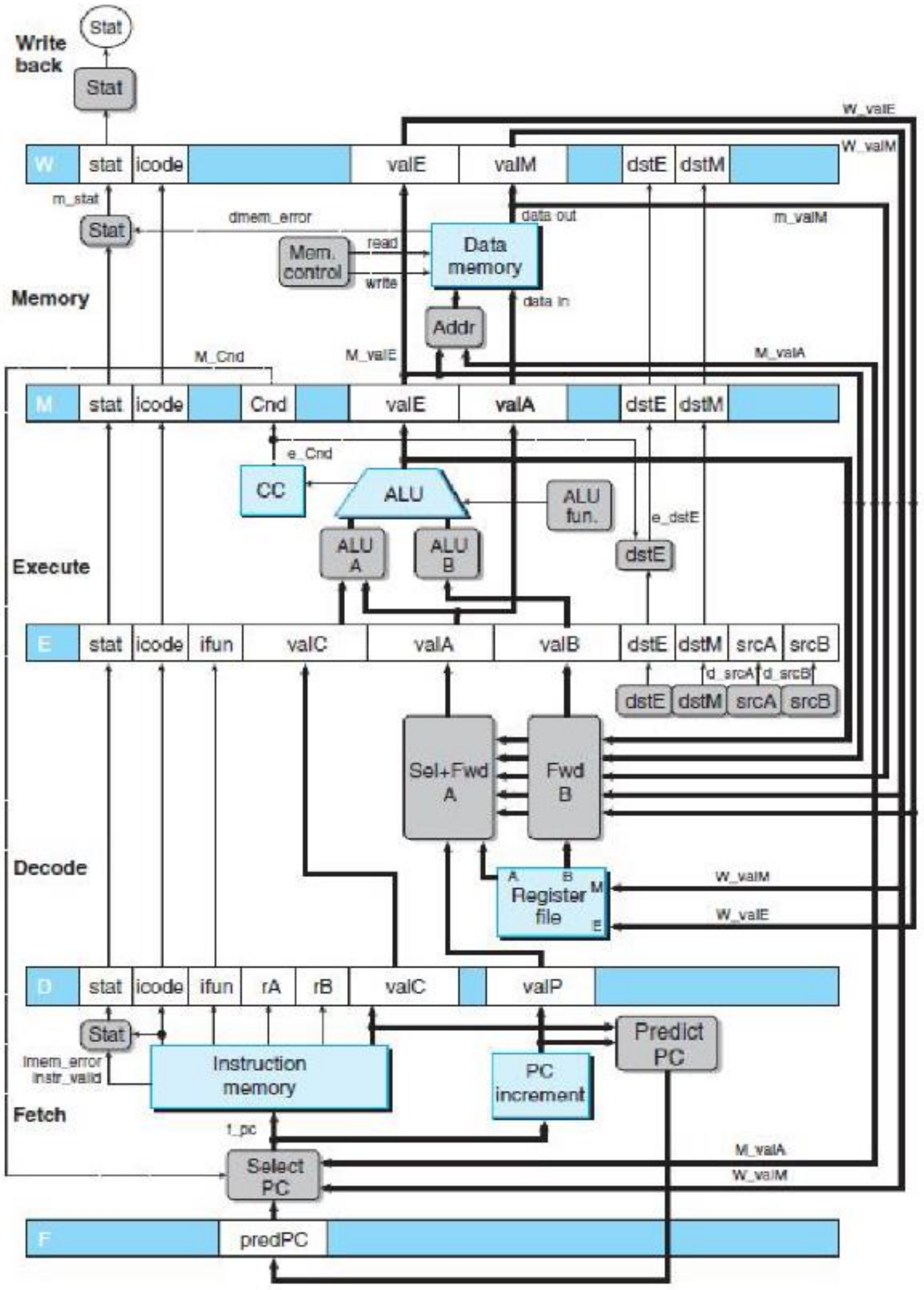
This when implemented along with the registers will be as the following



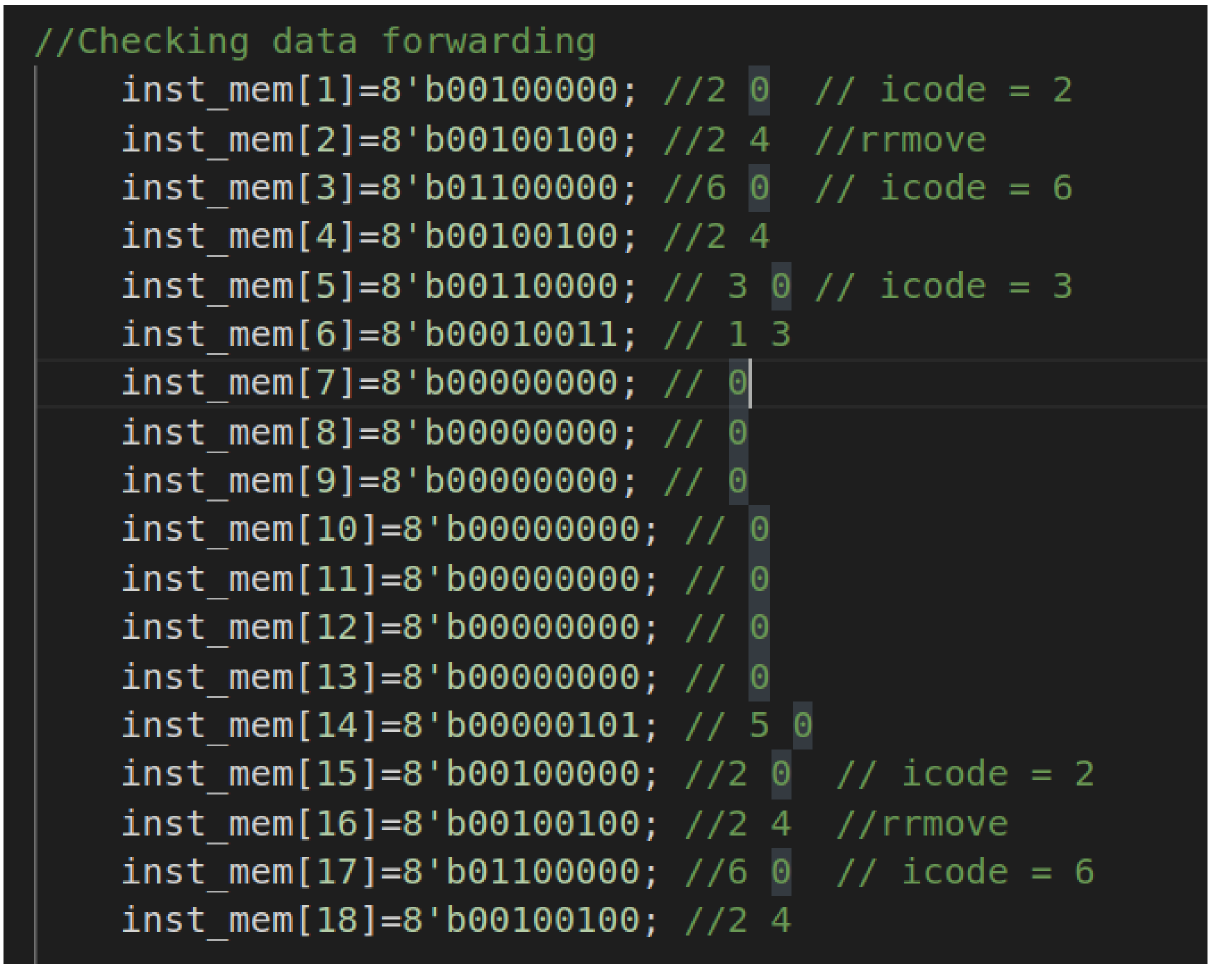
```
bool F_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB } ||
        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };
```

```
bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&
          IRET in { D_icode, E_icode, M_icode };
```

Overall Implementation of Y86 processor- 5 stage pipeline

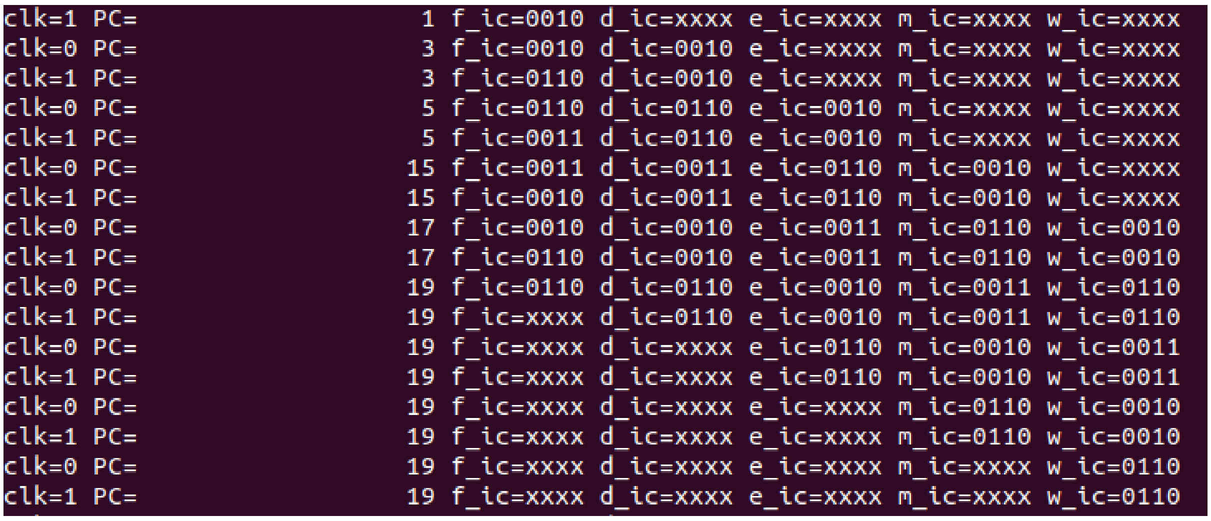


Terminal Results:



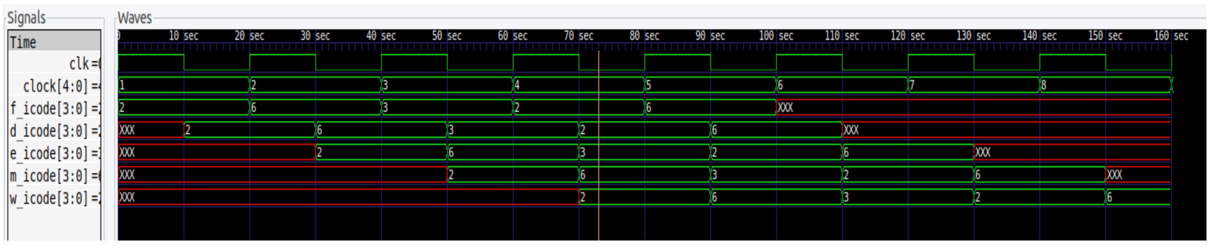
```
//Checking data forwarding
    inst_mem[1]=8'b00100000; //2 0  // icode = 2
    inst_mem[2]=8'b00100100; //2 4  //rrmove
    inst_mem[3]=8'b01100000; //6 0  // icode = 6
    inst_mem[4]=8'b00100100; //2 4
    inst_mem[5]=8'b00110000; // 3 0 // icode = 3
    inst_mem[6]=8'b00010011; // 1 3
    inst_mem[7]=8'b00000000; // 0
    inst_mem[8]=8'b00000000; // 0
    inst_mem[9]=8'b00000000; // 0
    inst_mem[10]=8'b00000000; // 0
    inst_mem[11]=8'b00000000; // 0
    inst_mem[12]=8'b00000000; // 0
    inst_mem[13]=8'b00000000; // 0
    inst_mem[14]=8'b00000101; // 5 0
    inst_mem[15]=8'b00100000; //2 0  // icode = 2
    inst_mem[16]=8'b00100100; //2 4  //rrmove
    inst_mem[17]=8'b01100000; //6 0  // icode = 6
    inst_mem[18]=8'b00100100; //2 4
```

The above figure shows the instruction memory which displays how data forwarding works

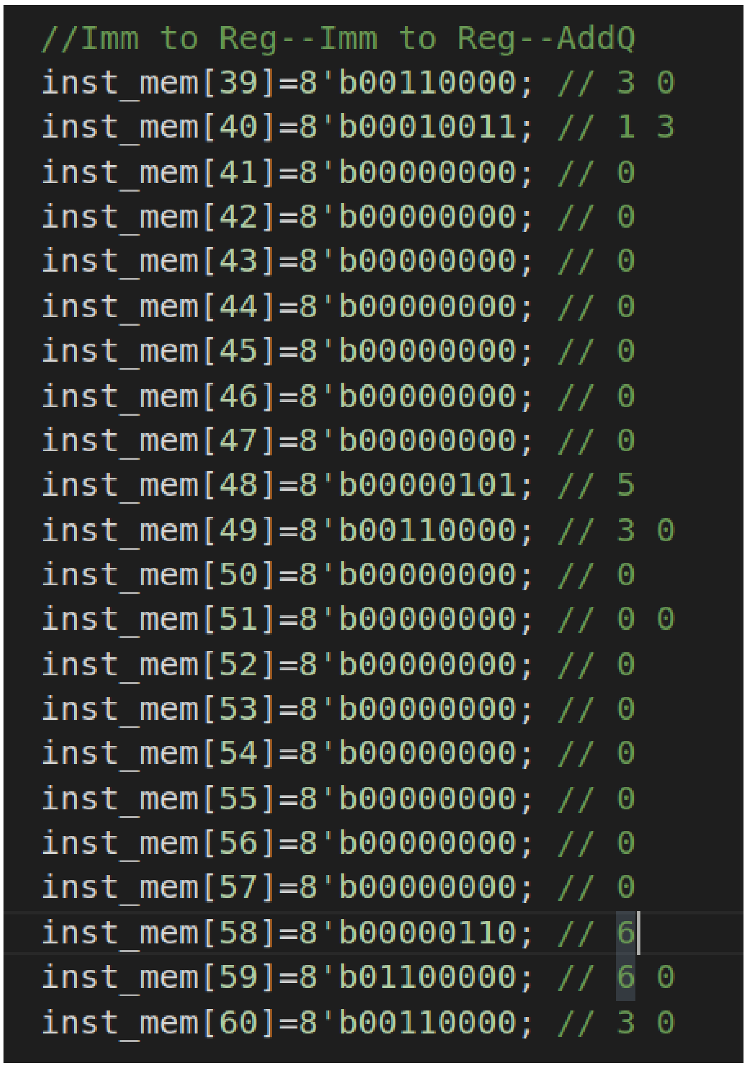


```
clk=1 PC=                    1 f_ic=0010 d_ic=xxxx e_ic=xxxx m_ic=xxxx w_ic=xxxx
clk=0 PC=                    3 f_ic=0010 d_ic=0010 e_ic=xxxx m_ic=xxxx w_ic=xxxx
clk=1 PC=                    3 f_ic=0110 d_ic=0010 e_ic=xxxx m_ic=xxxx w_ic=xxxx
clk=0 PC=                    5 f_ic=0110 d_ic=0110 e_ic=0010 m_ic=xxxx w_ic=xxxx
clk=1 PC=                    5 f_ic=0011 d_ic=0110 e_ic=0010 m_ic=xxxx w_ic=xxxx
clk=0 PC=                   15 f_ic=0011 d_ic=0011 e_ic=0110 m_ic=0010 w_ic=xxxx
clk=1 PC=                   15 f_ic=0010 d_ic=0011 e_ic=0110 m_ic=0010 w_ic=xxxx
clk=0 PC=                   17 f_ic=0010 d_ic=0010 e_ic=0011 m_ic=0110 w_ic=0010
clk=1 PC=                   17 f_ic=0110 d_ic=0010 e_ic=0011 m_ic=0110 w_ic=0010
clk=0 PC=                   19 f_ic=0110 d_ic=0110 e_ic=0010 m_ic=0011 w_ic=0110
clk=1 PC=                   19 f_ic=xxxx d_ic=0110 e_ic=0010 m_ic=0011 w_ic=0110
clk=0 PC=                   19 f_ic=xxxx d_ic=xxxx e_ic=0110 m_ic=0010 w_ic=0011
clk=1 PC=                   19 f_ic=xxxx d_ic=xxxx e_ic=0110 m_ic=0010 w_ic=0011
clk=0 PC=                   19 f_ic=xxxx d_ic=xxxx e_ic=xxxx m_ic=0110 w_ic=0010
clk=1 PC=                   19 f_ic=xxxx d_ic=xxxx e_ic=xxxx m_ic=0110 w_ic=0010
clk=0 PC=                   19 f_ic=xxxx d_ic=xxxx e_ic=xxxx m_ic=xxxx w_ic=0110
clk=1 PC=                   19 f_ic=xxxx d_ic=xxxx e_ic=xxxx m_ic=xxxx w_ic=0110
```

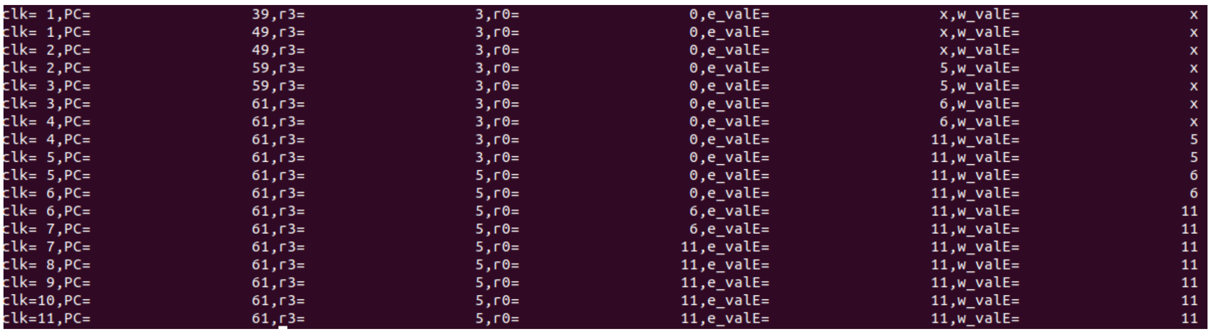
Here we can see how pipelining works, when PC = 17 we can see that different stages in the pipeline are carrying out different instructions. Initially fetch gets first instruction then it goes to decode in next clock cycle then new instruction enters fetch and this process carries on until writeback stage. We can even see how pipeling works using GTK waves as shown below



Here we took a random example to show the working of pipelining along with data forwarding implementation

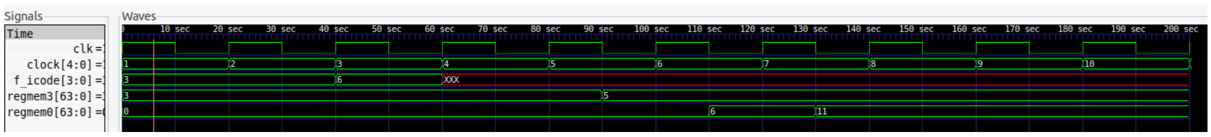


```
//Imm to Reg--Imm to Reg--AddQ
inst_mem[39]=8'b00110000; // 3 0
inst_mem[40]=8'b00010011; // 1 3
inst_mem[41]=8'b00000000; // 0
inst_mem[42]=8'b00000000; // 0
inst_mem[43]=8'b00000000; // 0
inst_mem[44]=8'b00000000; // 0
inst_mem[45]=8'b00000000; // 0
inst_mem[46]=8'b00000000; // 0
inst_mem[47]=8'b00000000; // 0
inst_mem[48]=8'b00000101; // 5
inst_mem[49]=8'b00110000; // 3 0
inst_mem[50]=8'b00000000; // 0
inst_mem[51]=8'b00000000; // 0 0
inst_mem[52]=8'b00000000; // 0
inst_mem[53]=8'b00000000; // 0
inst_mem[54]=8'b00000000; // 0
inst_mem[55]=8'b00000000; // 0
inst_mem[56]=8'b00000000; // 0
inst_mem[57]=8'b00000000; // 0
inst_mem[58]=8'b00000110; // 6
inst_mem[59]=8'b01100000; // 6 0
inst_mem[60]=8'b00110000; // 3 0
```



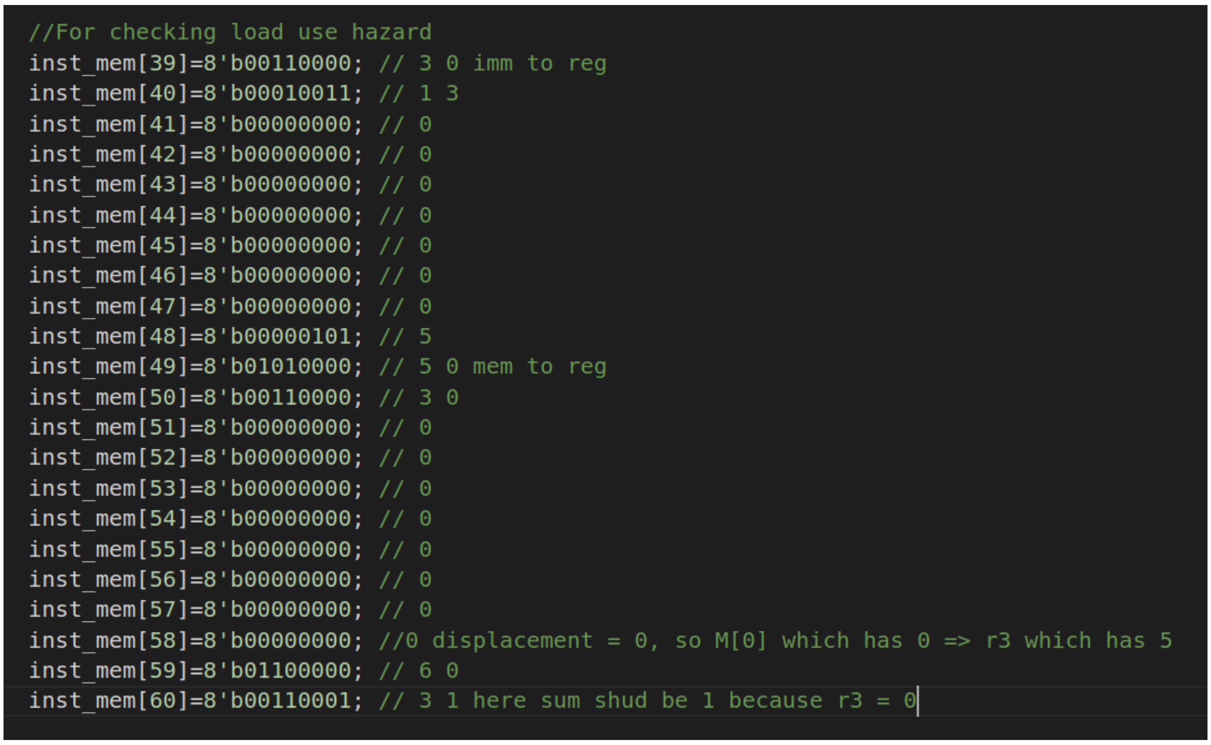
```
clk= 1,PC=                    39,r3=                 3,r0=                  0,e_valE=                 x,w_valE=                  x
clk= 1,PC=                    49,r3=                 3,r0=                  0,e_valE=                 x,w_valE=                  x
clk= 2,PC=                    49,r3=                 3,r0=                  0,e_valE=                 x,w_valE=                  x
clk= 2,PC=                    59,r3=                 3,r0=                  0,e_valE=                 5,w_valE=                  x
clk= 3,PC=                    59,r3=                 3,r0=                  0,e_valE=                 5,w_valE=                  x
clk= 3,PC=                    61,r3=                 3,r0=                  0,e_valE=                 6,w_valE=                  x
clk= 4,PC=                    61,r3=                 3,r0=                  0,e_valE=                 6,w_valE=                  x
clk= 4,PC=                    61,r3=                 3,r0=                  0,e_valE=                11,w_valE=                  5
clk= 5,PC=                    61,r3=                 3,r0=                  0,e_valE=                11,w_valE=                  5
clk= 5,PC=                    61,r3=                 5,r0=                  0,e_valE=                11,w_valE=                  6
clk= 6,PC=                    61,r3=                 5,r0=                  0,e_valE=                11,w_valE=                  6
clk= 6,PC=                    61,r3=                 5,r0=                  6,e_valE=                11,w_valE=                 11
clk= 7,PC=                    61,r3=                 5,r0=                  6,e_valE=                11,w_valE=                 11
clk= 7,PC=                    61,r3=                 5,r0=                 11,e_valE=                11,w_valE=                 11
clk= 8,PC=                    61,r3=                 5,r0=                 11,e_valE=                11,w_valE=                 11
clk= 9,PC=                    61,r3=                 5,r0=                 11,e_valE=                11,w_valE=                 11
clk=10,PC=                    61,r3=                 5,r0=                 11,e_valE=                11,w_valE=                 11
clk=11,PC=                    61,r3=                 5,r0=                 11,e_valE=                11,w_valE=                 11
```

Here we can see even though w_valE gets written in future clock cycles, e_valE gets data because of data forwarding from forward stages to previous stages



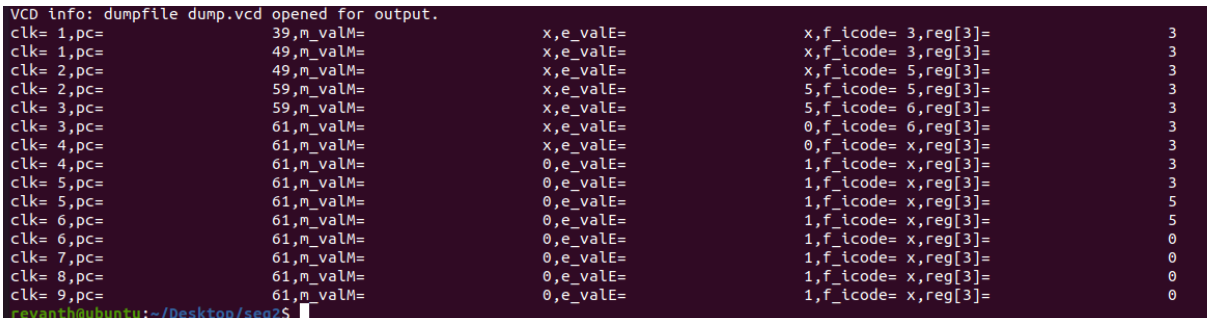
This is gtkwaves for above taken example

Now we check for load use hazard working, and we take the following example as shown in the instruction memory

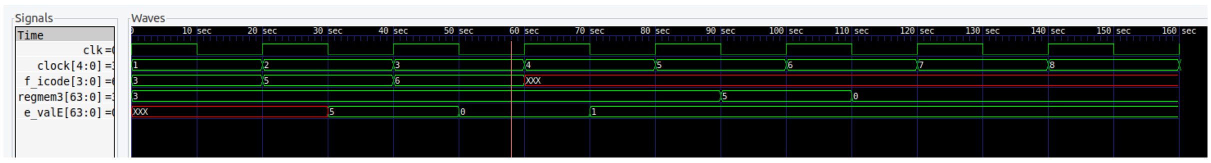


```
//For checking load use hazard
inst_mem[39]=8'b00110000; // 3 0 imm to reg
inst_mem[40]=8'b00010011; // 1 3
inst_mem[41]=8'b00000000; // 0
inst_mem[42]=8'b00000000; // 0
inst_mem[43]=8'b00000000; // 0
inst_mem[44]=8'b00000000; // 0
inst_mem[45]=8'b00000000; // 0
inst_mem[46]=8'b00000000; // 0
inst_mem[47]=8'b00000000; // 0
inst_mem[48]=8'b00000101; // 5
inst_mem[49]=8'b01010000; // 5 0 mem to reg
inst_mem[50]=8'b00110000; // 3 0
inst_mem[51]=8'b00000000; // 0
inst_mem[52]=8'b00000000; // 0
inst_mem[53]=8'b00000000; // 0
inst_mem[54]=8'b00000000; // 0
inst_mem[55]=8'b00000000; // 0
inst_mem[56]=8'b00000000; // 0
inst_mem[57]=8'b00000000; // 0
inst_mem[58]=8'b00000000; //0 displacement = 0, so M[0] which has 0 => r3 which has 5
inst_mem[59]=8'b01100000; // 6 0
inst_mem[60]=8'b00110001; // 3 1 here sum shud be 1 because r3 = 0
```

Here due to adding a bubble in the execute stage and stalling in the fetch and decode stage we are able to get enough time to read from memory and write it to register and then the next instruction OPq gets correct values from registers

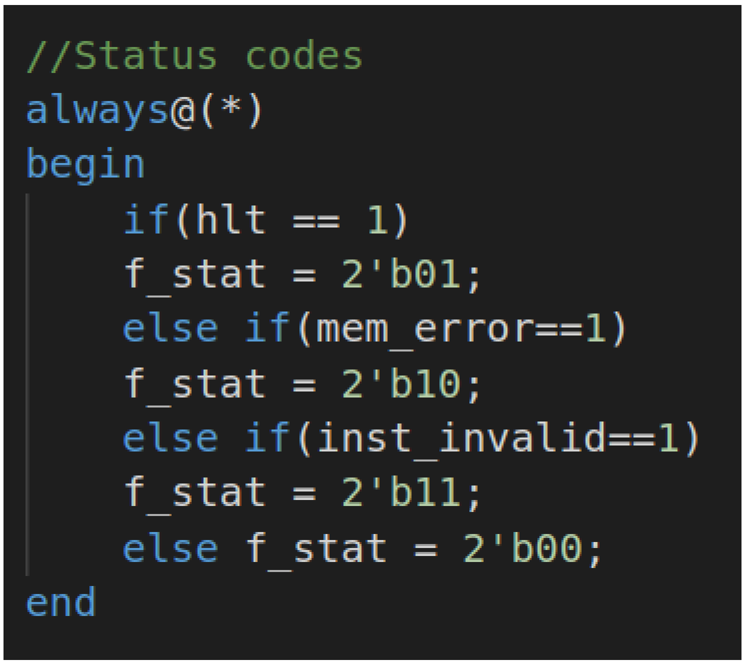


```
VCD info: dumpfile dump.vcd opened for output.
clk= 1,pc=                  39,m_valM=                  x,e_valE=                  x,f_icode= 3,reg[3]=                  3
clk= 1,pc=                  49,m_valM=                  x,e_valE=                  x,f_icode= 3,reg[3]=                  3
clk= 2,pc=                  49,m_valM=                  x,e_valE=                  x,f_icode= 5,reg[3]=                  3
clk= 2,pc=                  59,m_valM=                  x,e_valE=                  5,f_icode= 5,reg[3]=                  3
clk= 3,pc=                  59,m_valM=                  x,e_valE=                  5,f_icode= 6,reg[3]=                  3
clk= 3,pc=                  61,m_valM=                  x,e_valE=                  0,f_icode= 6,reg[3]=                  3
clk= 4,pc=                  61,m_valM=                  x,e_valE=                  0,f_icode= x,reg[3]=                  3
clk= 4,pc=                  61,m_valM=                  0,e_valE=                  1,f_icode= x,reg[3]=                  3
clk= 5,pc=                  61,m_valM=                  0,e_valE=                  1,f_icode= x,reg[3]=                  3
clk= 5,pc=                  61,m_valM=                  0,e_valE=                  1,f_icode= x,reg[3]=                  5
clk= 6,pc=                  61,m_valM=                  0,e_valE=                  1,f_icode= x,reg[3]=                  5
clk= 6,pc=                  61,m_valM=                  0,e_valE=                  1,f_icode= x,reg[3]=                  0
clk= 7,pc=                  61,m_valM=                  0,e_valE=                  1,f_icode= x,reg[3]=                  0
clk= 8,pc=                  61,m_valM=                  0,e_valE=                  1,f_icode= x,reg[3]=                  0
clk= 9,pc=                  61,m_valM=                  0,e_valE=                  1,f_icode= x,reg[3]=                  0
revanth@ubuntu:~/Desktop/seq2$
```

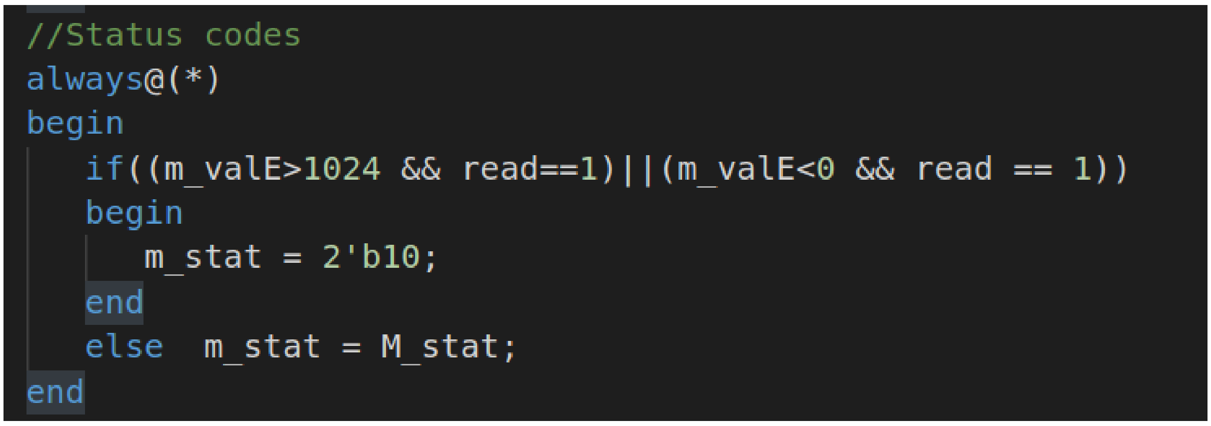


The above shows GTKwave output for the same load use hazard

The below figures shows how we set the status codes, and first figure updates status codes in fetch stage while the second figure updates status codes in memory stage if needed



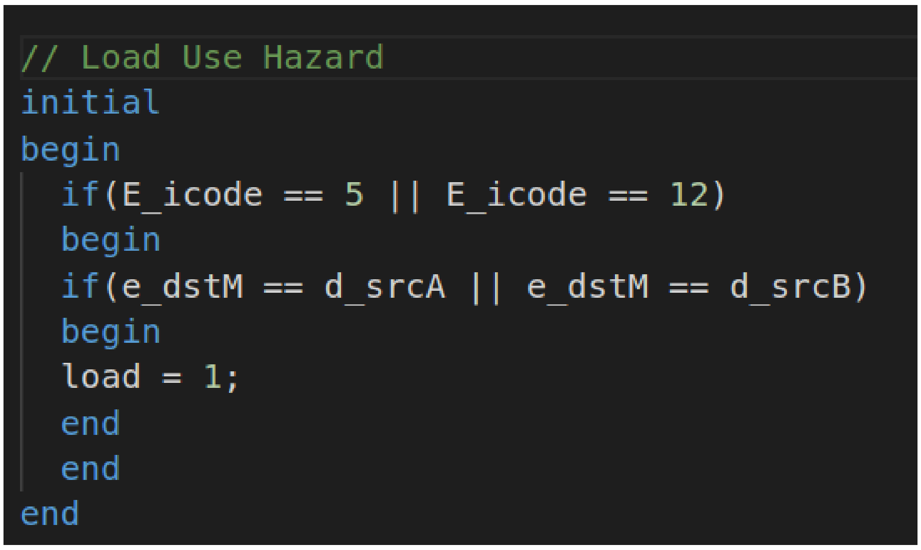
```
//Status codes
always@(*)
begin
    if(hlt == 1)
    f_stat = 2'b01;
    else if(mem_error==1)
    f_stat = 2'b10;
    else if(inst_invalid==1)
    f_stat = 2'b11;
    else f_stat = 2'b00;
end
```



```
//Status codes
always@(*)
begin
   if((m_valE>1024 && read==1)||(m_valE<0 && read == 1))
   begin
      m_stat = 2'b10;
   end
   else  m_stat = M_stat;
end
```

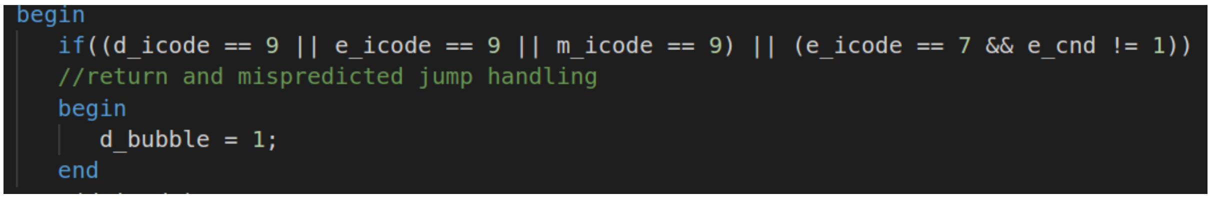
Hazards:

This figure shows how we handled load use hazard



```
// Load Use Hazard
initial
begin
  if(E_icode == 5 || E_icode == 12)
  begin
  if(e_dstM == d_srcA || e_dstM == d_srcB)
  begin
  load = 1;
  end
  end
end
```

This figure shows how we handled mispredicted jump and return



```
begin
  if((d_icode == 9 || e_icode == 9 || m_icode == 9) || (e_icode == 7 && e_cnd != 1))
  //return and mispredicted jump handling
  begin
    d_bubble = 1;
  end
```

Challenges Encountered:

- Implementing logic for Load Hazard controls.
- Executing and Testing Call and Return controls.
- Dealing with the mispredicted PC and mispredicted jumps.