Final-Eval Report

PANTHERS

Y86-64 Reference

Instruction Format

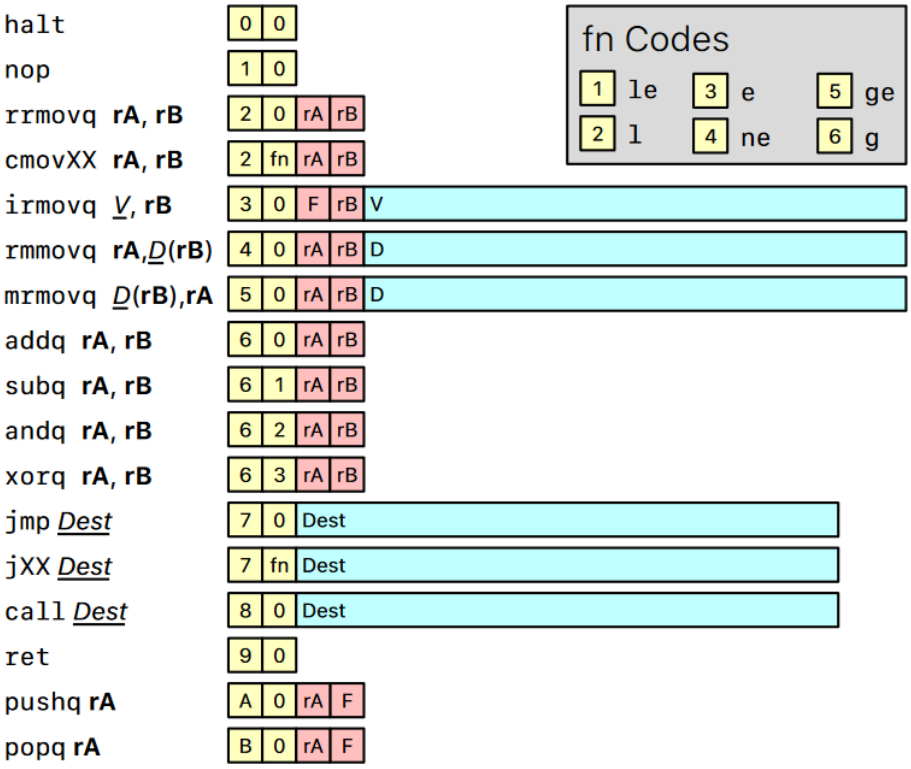


Figure 1: Y86-64 ISA

stage	register(s)	description
Fetch	icode,ifun	Read instruction byte
	rA,rB	Read register byte
	valC	Read constant word
	valP	Compute next PC
Decode	valA,srcA	Read operand A
	valB,srcB	Read operand B
Execute	valE	Perform ALU operation
	cnd	Set/Use Condition Code
Memory	valM	Memory Read/Write
Writeback	dstE	Write back ALU result
	dstM	Write back Mem result
PC Update	PC	Update PC

Figure 2: Y86 Stages

Fetch Stage:

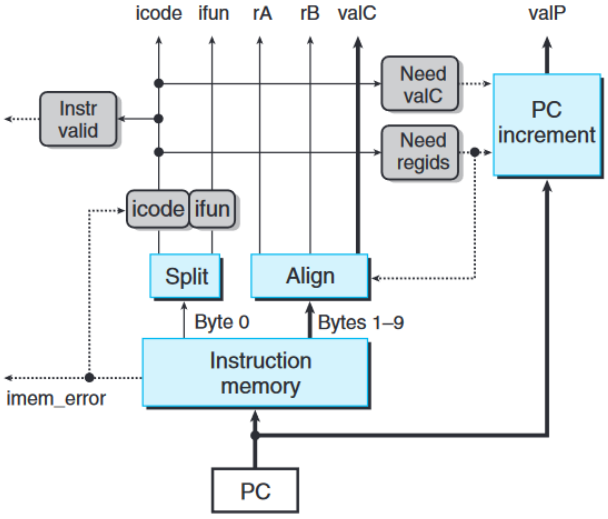


Figure 3: SEQ Fetch Stage

We will read 10 bytes from the instruction memory, using the PC as the address of the 1'st byte.

```
ins<=Instruction_Mem[PC+:80];
icode<=ins[0:3];
ifun<=ins[4:7];
```

Then based on icode, we decide if we need regids or valC. We also decide if we are supplied a valid instruction.

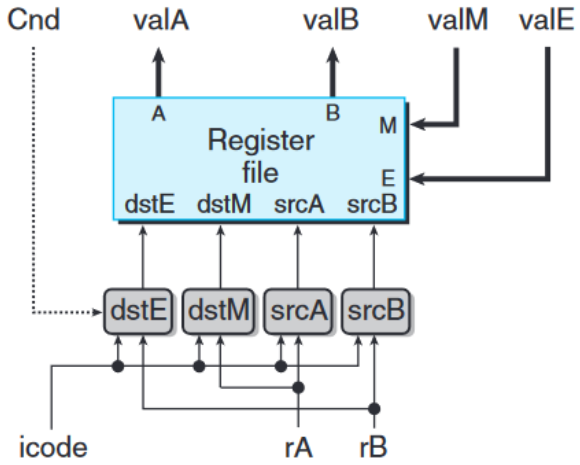
```
else if(icode==4'h2)    //cmovxx
begin
    instr_valid<=1;
    need_regids<=1;
    need_valC<=0;
end
```

Also, if the PC points outside the instruction memory, we consider that an instruction mem error has occurred. Then, based on need_regids and need_valC, we take input as needed by the instruction, and set the next PC, valP.

```
if(need_valC==1 && need_regids==1)
begin
    //valC<=ins[16:79];//6483==8364
    valC[7:0]<=  ins[16:23];
    valC[15:8]<= ins[24:31];
    valC[23:16]<=ins[32:39];
    valC[31:24]<=ins[40:47];
    valC[39:32]<=ins[48:55];
    valC[47:40]<=ins[56:63];
    valC[55:48]<=ins[64:71];
    valC[63:56]<=ins[72:79];
end
```

Figure 4: Y86 uses little Endian(LSB first) to store valC

Decode and Write-Back Stage:



Decode and write-back stages have been merged because they both access the register file. Firstly, let us look at the decode stage:

Based on the icode, we decide if we are going to read rA or the stack pointer (or none, which is denoted by F).

```
else if(icode==4'h2)    //IRRMOVQ
begin
    srcA<=rA;
    srcB<=4'hF;
end
```

Then if srcA is not F, we set the output valA as srcA register and same goes for B.

For the write-back stage:

Based on the icode and input values valE(from execute stage) and valM(memory stage), we set the destA to the required register.

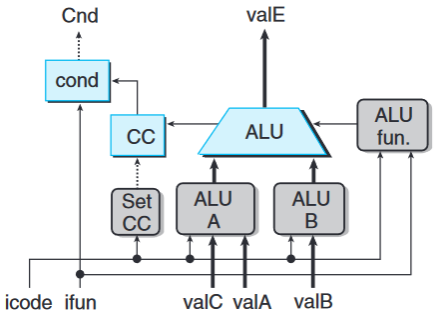
```
else if((icode==4'h2) && (Cnd==1'b1))   //IRRMOVQ
begin
    destM<=4'hF;
    destE<=rB;
end
else if(icode==4'h3)    //IIRMOVQ
begin
    destM<=4'hF;
    destE<=rB;
end
```

Figure 5: CMOV is executed only when the condition is true

Then at the positive edge of the clock we, write back valE or valM based on destE and destM so that the update takes place at the next clock cycle.

Note: For the case of popq %rsp, both E and M write ports are active with different valE, valM but writing to the same register 4. In this case valM takes higher priority.

Execute Stage:



On the basis of icode, we decide aluA to be one of valC, valA,-8 or +8. AluB can either be 0 or valB. And based on ifun, we will get the output of the ALU as valE

```
else if(icode==4'h3)    //IIRMOVQ
begin
    alufun<=2'b00;
    aluA<=valC;
    aluB<=64'b0;
end
```

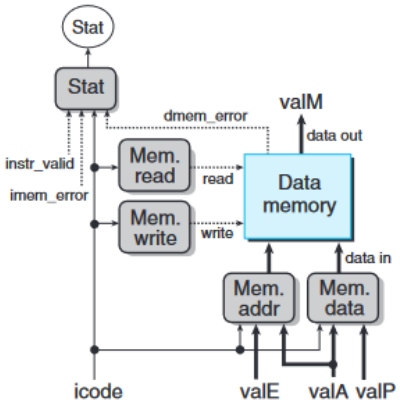
We also read the previous condition code to set up the Cnd in case conditional branch or data transfer should take place(cmov or jmp):

```
if(icode==4'h2 || icode==4'h7)
begin
    if(ifun==4'h0)
        Cnd=1'b1;
    else if(ifun==4'h1)//le
        Cnd=((SF^OF)|ZF);
    else if(ifun==4'h2)//l
        Cnd=SF^OF;
    else if(ifun==4'h3)//e
        Cnd<=ZF;
    else if(ifun==4'h4)//ne
        Cnd=~ZF;
    else if(ifun==4'h5)//ge
        Cnd=(~SF^OF);
    else if(ifun==4'h6)//g
        Cnd=((~SF^OF)&~ZF);
end
```

Then if icode is 6(OPq), we update the condition codes in the memory.

```
if(valE==0)
    ZF<=1'b1;
else
    ZF<=1'b0;
SF<=valE[63];
OF<=overflow;
```

Memory Stage:



We decide if we read or write to memory based on the icode, we also decide the mem addr to read from and mem_data to write into memory also case of write.

```
else if(icode==4'h9)    //IRET
begin
    mem_read<=1'b1;
    mem_addr<=valA;
end
else if(icode==4'hA)    //IPUSHQ
begin
    mem_write<=1'b1;
    mem_addr<=valE;
    mem_data<=valA;
end
```

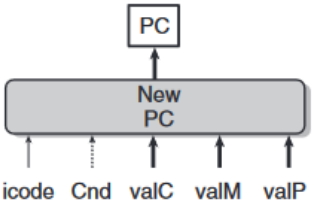
If mem_addr to be read or write to larger than the max data memory, dmem_error will be set to 1 indicating the memory address is out of bounds. Then we can indicate the status of the processor based on these values:

Status Conditions

AOK	1	Normal
HLT	2	Halt Encountered
ADR	3	Bad Address
INS	4	Invalid Instruction

Then at the next clock cycle the new value is written to the memory.

PC Update Stage:



In case we encounter a jump statement with Cnd=1, we set the new PC to valC.

For call statement too, we set the PC to valC. For return, we set it to valM.

For all other cases the PC is set to valP by default.

Sequential Processor:

All 5 blocks mentioned above have been merged and one program has been written to test the processor.

The program stores the summation till 100 using loops in %rax.

To run:

iverilog -o test seq.v

vvp test

gtkwave seq.vcd

irmovq $0,%rax #sum=0	30f00000000000000000
irmovq $1,%rcx #num=1	30f10100000000000000
Loop: addq %rcx,%rax #sum+=num	6010
irmovq $1,%rdx #tmp=1	30f20100000000000000
addq %rdx,%rcx #num++	6021
irmovq $100,%rdx #lim=100	30f26400000000000000
subq %rcx,%rdx #if lim-num>=0	6112
jge Loop #jump to loop	75A000000000000000
halt	00

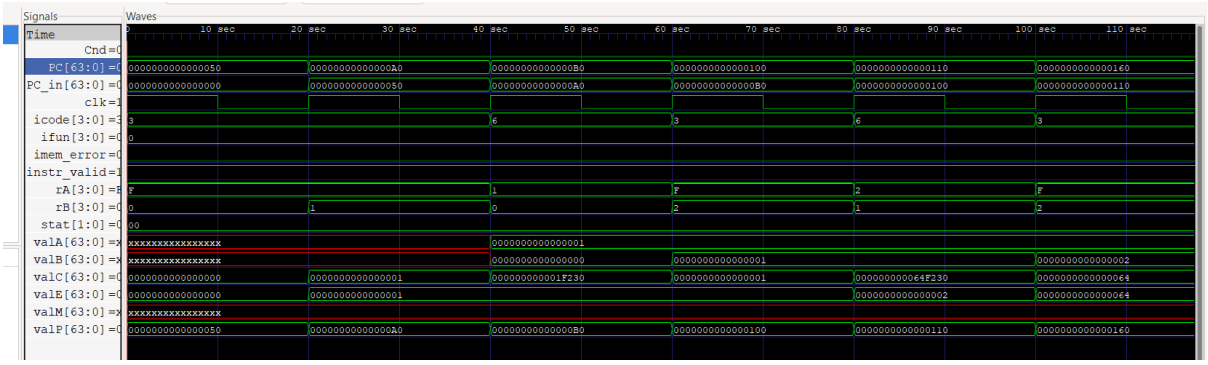


Figure 6: GTKWave output of the program

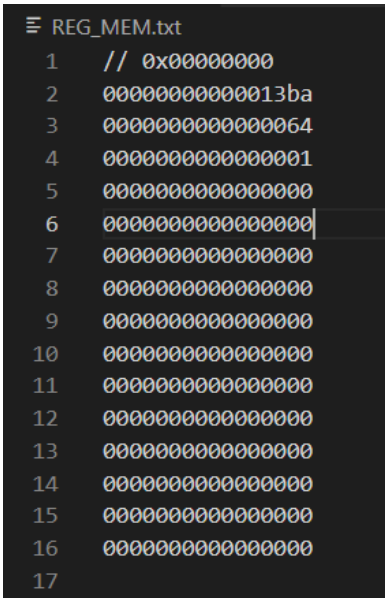


Figure 7: Register Memory output

13BA in HEX= 5050 in DEC =Σ100

TB for Fetch:

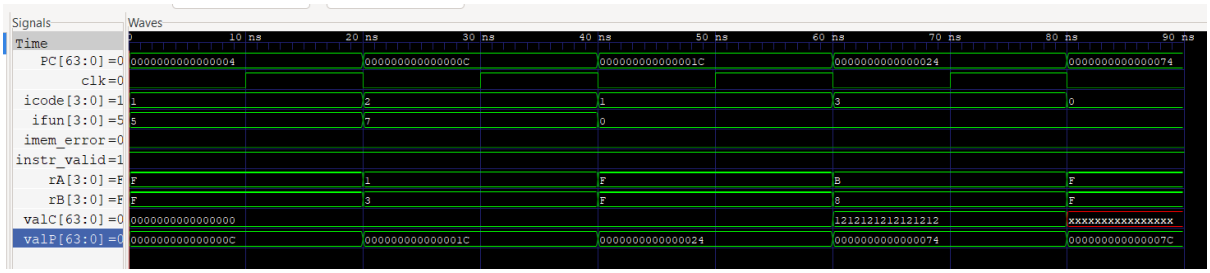
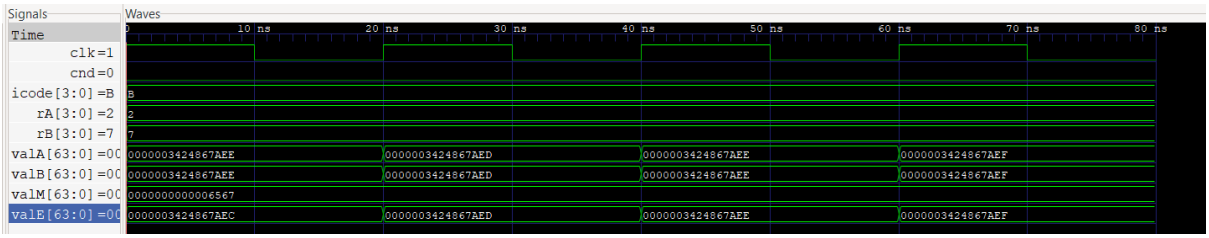


Figure 8: INS:01527131030B8121212121212121200 PC:4

TB for Decode/WB:



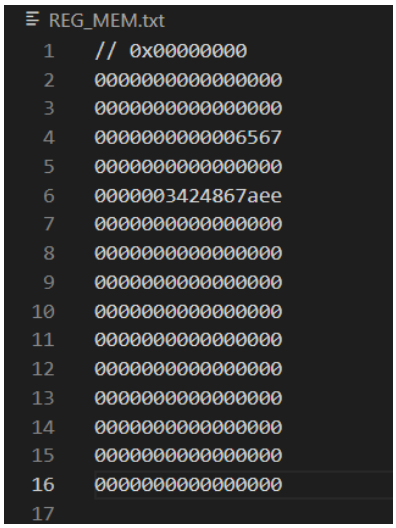


Figure 9: AEF isn't written until next clock cycle; popq:rsp<-valE and rA<-valM

TB for Execute:

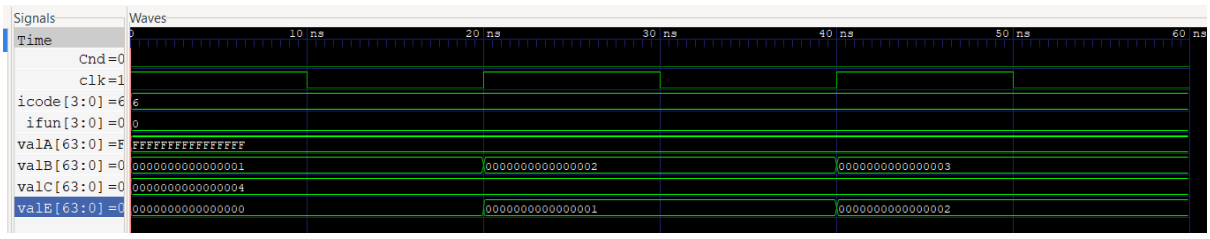


Figure 10: valE=valA+valB; same as old ALU

TB for Memory:

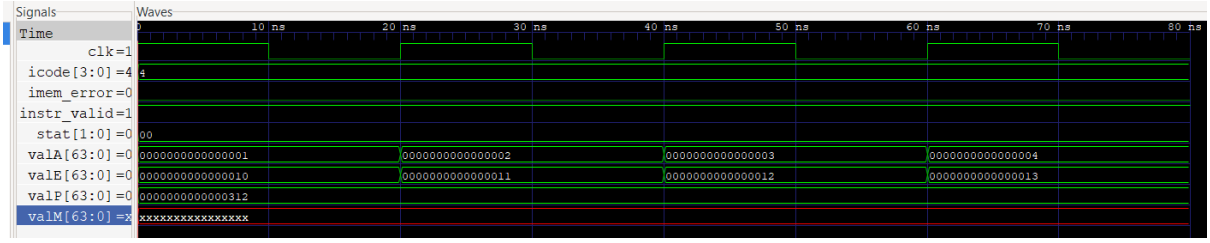


Figure 11: icode 4->rmmovq: M[valE]<-valA

```
14    0000000000000000
15    0000000000000000
16    0000000000000000
17    0000000000000000
18    // 0x00000010
19    0000000000000001
20    0000000000000002
21    0000000000000003
22    0000000000000000
23    0000000000000000
24    0000000000000000
```

Figure 12: Data Memory contents

TB for PC update is skipped:

```
reg [63:0]new_pc;
always @(*)
begin
        if(icode==4'h7 && Cnd==1'b1)    //jxx
        new_pc<=valC;
        else if(icode==4'h8)    //call
        new_pc<=valC;
        else if(icode==4'h9)    //ret
        new_pc<=valM;
        else
        new_pc<=valP;
    PC<=new_pc;
end
```

Pipelining

We have implemented a fully pipelined processor capable of handling load/use hazards, mispredicted branches and return statements with data forwarding.

The pipelining consists of five register blocks: fetch, decode, execute, memory and the write-back block, each block is placed in front of the respective sequential blocks. The pipelining is important because it increases the number of instructions that are executed for a second and reduces the length of each clock cycle significantly compared to the sequential processor. The new values will be loaded into the pipeline registers when there is a positive clock edge.

In the pipelined processor, we will move our PC update stage to the beginning of each cycle, this would allow us to load a new instruction while the previous one is passed down the pipeline of the other stages.

We have also implemented a much more robust register storage file, compared to the previous implementation of storing the values to a file. This update would allow to view the register values through each clock cycle.

```
// Clocked register with enable signal and synchronous reset
module cenrreg(out, in, enable, reset, resetval, clock);
parameter width = 8;
output [width-1:0] out;
reg [width-1:0] out;
input [width-1:0] in;
input enable;
input reset;
input [width-1:0] resetval;
input clock;
always @(posedge clock)
begin
    if (reset)
        out <= resetval;
    else if (enable)
        out <= in;
end
endmodule
```

Figure 13: Clocked Register

CENRREG: Conditionally enabled, resettable register.

The CENRREG is used as a building block for our pipeline registers and register file.

```
module preg(out, in, stall, bubble, bubbleval, clock);
parameter width = 8;
output [width-1:0] out;
input [width-1:0] in;
input stall, bubble;
input [width-1:0] bubbleval;
input clock;

cenrreg #(width) r(out, in, ~stall, bubble, bubbleval, clock);
endmodule
```

Figure 14: Pipeline register

We will now go through each of the block, highlighting the updates made to the pipelined register from the sequential processor.

The names with capital letters, ‘D’, ’E’,’M’ and ‘W’ refer to the pipeline registers. And the names with lowercase letters denote the values generated by the stage.

FETCH

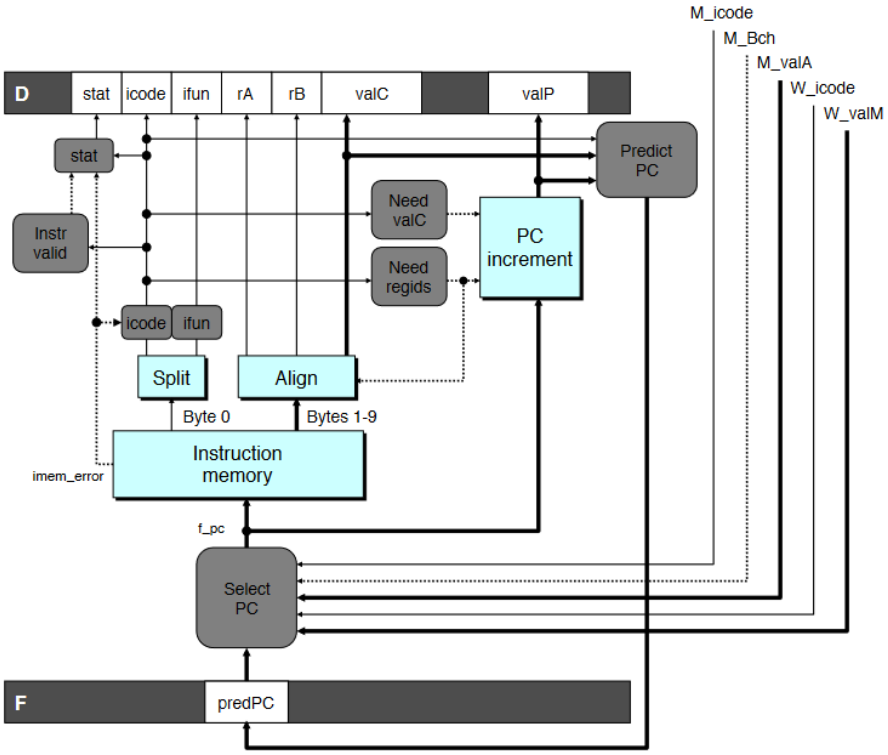


Figure 15: PC Selection and fetch logic

```
if((M_icode == 4'h7) & M_Cnd==0)//jmp
    f_pc<= M_valA;
else if(W_icode==4'h9)//ret
    f_pc<=W_valM;
else if(F_predPC)
    f_pc<= F_predPC;
```

For a mispredicted branch, the next instruction will be read from the pipeline register M(M_valA). When a ret instruction enters the write-back stage, the return address is read from pipeline register W.

The default case will be the predicted PC value, as we have seen in the SEQ, will just be valC or valP.

```
always @(*)
begin
        if(icode==4'h7) //jxx
        new_pc<=valC;
        else if(icode==4'h8)//call
        new_pc<=valC;
        else
        new_pc<=valP;
    predPC<=new_pc;
```

Pipeline control logic for FETCH

Data hazard	Condition for next clock cycle
Load and use hazard	stall
Processing return	stall
Mispredicted branch	normal

```
if((E_icode == 4'h5 | E_icode == 4'hB) & (E_dstM == d_srcA | E_dstM== d_srcB))//5-MRMOVQ, B-POPQ
    F_stall<=1;
else if((D_icode==4'h9)|(E_icode==4'h9)|(M_icode==4'h9))//RET
    F_stall<=1;
else
    F_stall<=0;
F_bubble<=0;
```

DECODE

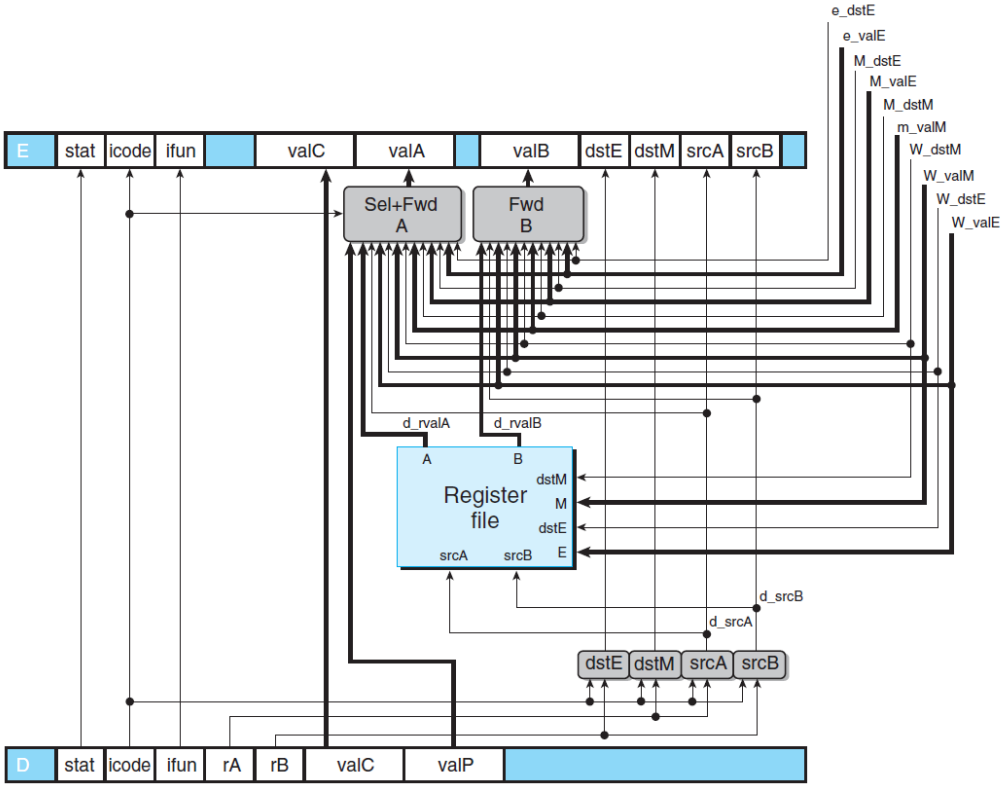


Figure 16: Decode and Write-back logic

No instruction requires both valP and the value read from register port A, and so these two can be merged to form the signal valA for later stages. This stage includes the logic used in data forwarding for valA and valB with values from stages down the pipeline, like the execute memory and write-back stages.

For d_valA:

- For call and jump, we use D_valP.
- If d_srcA and e_dstE match, we use e_valE.
- If d_srcA and M_dstM match, we use m_valM.
- If d_srcA and M_dstE match, we use M_valE.
- If d_srcA and W_dstM match, we use W_valM.
- If d_srcA and W_dstE matcfh, we use W_valE
- Default is d_rvalA read from register port A.

For d_valB, if d_srcB matches with the following, we use the corresponding value:

- E_dstE=>e_valE
- M_dstM=>m_valM
- M_dstE=>M_valE
- W_dstM=>W_valM
- W_dstE=>W_valE
- Default is d_rvalB, from register port B

```
reg [63:0]temp11,temp12;
assign d_valA=temp11;
assign d_valB=temp12;

always @(*) //Select and forward for A and B
begin
    if((D_icode == 4'h7 | D_icode == 4'h8))//jmp or call
        temp11= D_valP;
    else if(d_srcA == e_dstE)
        temp11=e_valE;
    else if(d_srcA == M_dstM)
        temp11=m_valM;
    else if(d_srcA == M_dstE)
        temp11=M_valE;
    else if(d_srcA == W_dstM)
        temp11=W_valM;
    else if(d_srcA == W_dstE)
        temp11=W_valE;
    else
        temp11=d_rvalA;


    if(d_srcB == e_dstE)
        temp12=e_valE;
    else if(d_srcB == M_dstM)
        temp12=m_valM;
    else if(d_srcB== M_dstE)
        temp12=M_valE;
    else if(d_srcB == W_dstM)
        temp12=W_valM;
    else if(d_srcB ==W_dstE)
        temp12=W_valE;
    else
        temp12=d_rvalB;
end
```

Figure 17: Data forwarding logic

Pipeline control logic for DECODE

Data hazard	Condition for next clock cycle
Load and use hazard	stall
Processing return	bubble
Mispredicted branch	bubble

```
if((E_icode == 4'h5 | E_icode == 4'hB) & (E_dstM == d_srcA | E_dstM == d_srcB))//5-MRMOVQ, B-POPQ, 7-JXX, 9-RET
    D_stall=1;
else
    D_stall=0;
if((E_icode == 4'h7) & (e_Cnd==0))//Mispredicted Branch
    D_bubble=1;
else if(~((E_icode == 4'h5 | E_icode ==4'hB) & (E_dstM == d_srcA | E_dstM == d_srcB)) & ( D_icode ==4'h9  | E_icode==4'h9  | M_icode==4'h9 ))
    D_bubble=1;
else
   D_bubble=0;
```

EXECUTE:

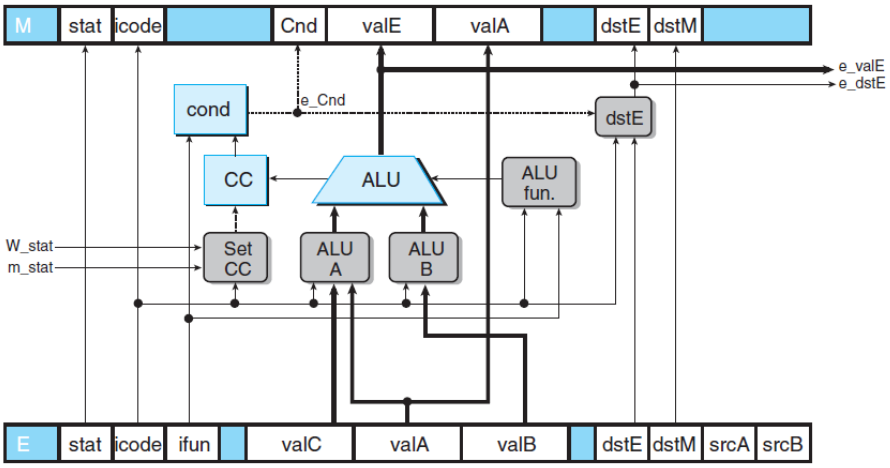


Figure 18: Execute stage logic

The execute stage in PIPE follows very closely to the one in SEQ.

Let us see the condition code register:

```
module cc(cc, new_cc, set_cc, clock);
output[2:0] cc;
input [2:0] new_cc;
input set_cc;
input clock;

cenrreg #(3) c(cc, new_cc, set_cc, 1'b0, 3'b100, clock);
endmodule
```

```
if(set_cc==1'b1)
begin
    if(valE==0)
        new_cc[0]<=1'b1;
    else
        new_cc[0]<=1'b0;
    new_cc[1]<=valE[63];
    new_cc[2]<=overflow;
end
```

Figure 19: set_cc is true when OPQ is encountered

```
if((E_icode==4'h2)& e_Cnd==0)
    e_dstE=4'hF;
else
    e_dstE=E_dstE;
```

Figure 20: Condition move check

This execute register block is placed in front of the sequential execute block and after the sequential decode block. It loads the outputs of the sequential decode block at the time of positive edge of the clock it holds those values until the next positive edge of the clock comes.

There are the stall and bubble conditions for the execute block these are to avoid the data hazards, these conditions will be applied based on the type of data hazard it is:

Pipeline control logic for EXECUTE

Data hazard	**Condition for next clock cycle**
Load and use hazard	bubble
Processing return	normal
Mispredicted branch	bubble

There is no use of stall in the execute block.

```
E_stall=0;
if((E_icode == 4'h7)&(e_Cnd==0))//Mispredicted branch
    E_bubble=1;
else if((E_icode == 4'h5 | E_icode == 4'hB) & (E_dstM == d_srcA | E_dstM == d_srcB))//5-MRMOVQ, B-POPQ
    E_bubble=1;
else
    E_bubble=0;
```

MEMORY

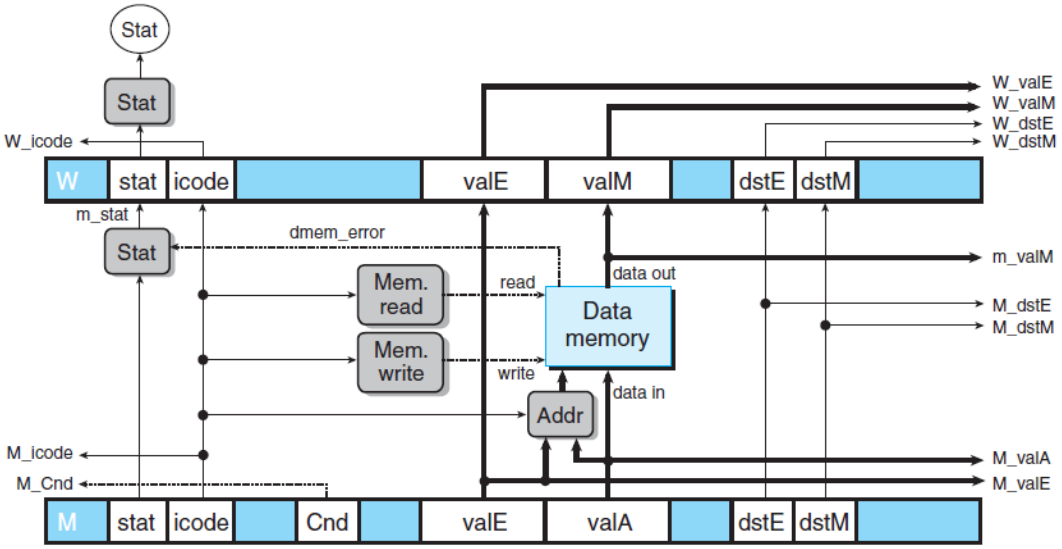


Figure 21: Memory Stage

Memory stage is also like the SEQ, but various values are forwarded back to the previous stages as part of forwarding and pipeline control logic.

Pipelined Processor

We will test our pipelined processor with the same program used in the sequential processor.

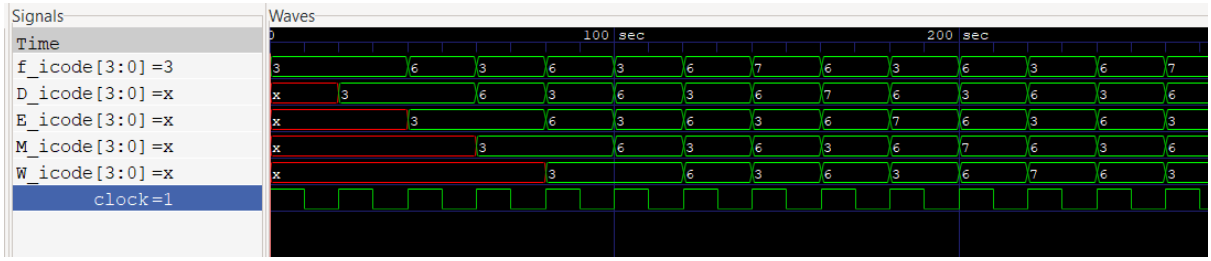
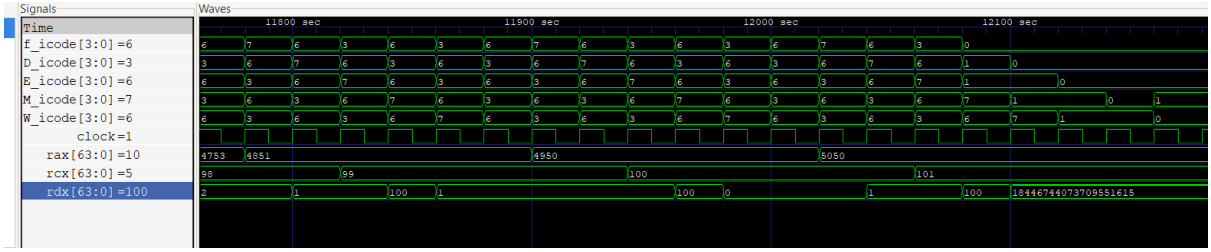


Figure 22: Instructions passing through pipeline registers



Note how the value in the rax register is 5050, sum till hundred and the successful implementation of the jump if greater than statement at the end(rdx) value.