Mid-Eval Report

PANTHERS

Y86-64 Reference

Instruction Format

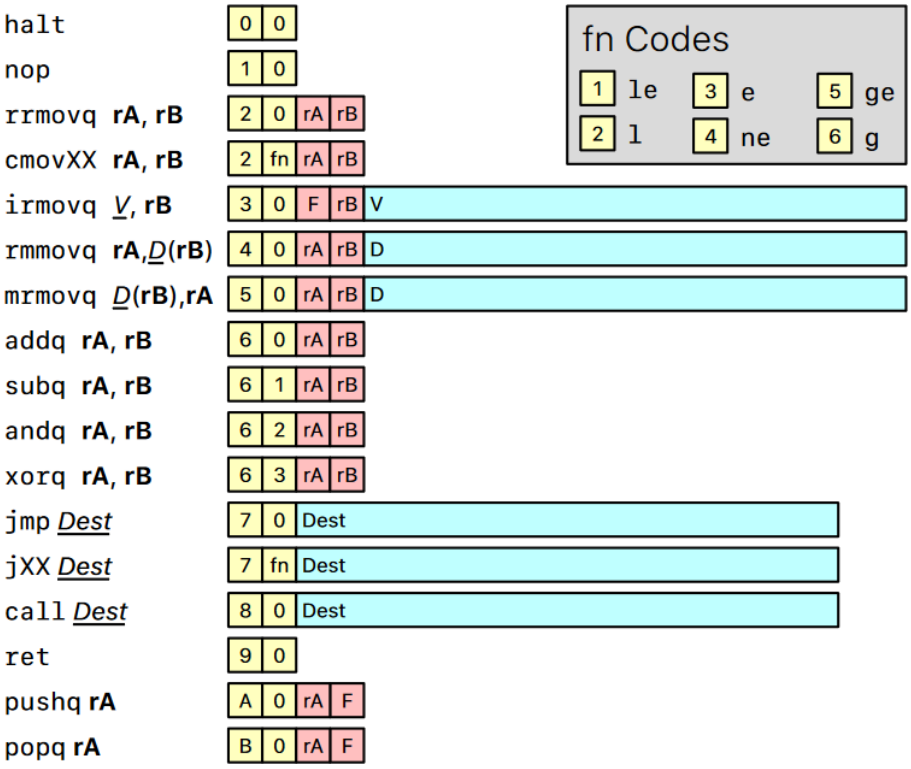


Figure 1: Y86-64 ISA

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:

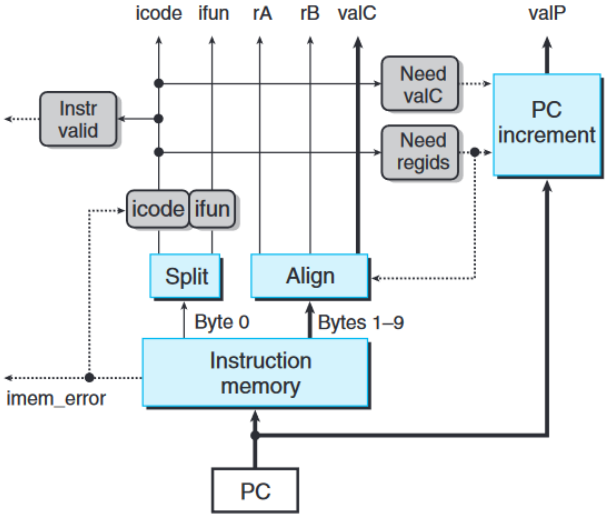


Figure 3: SEQ Fetch Stage

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.

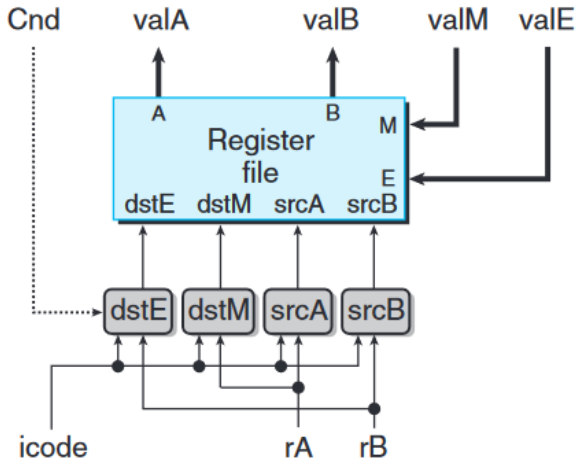
```
else if(icode==4'h2)    //cmovxx
begin
    instr_valid<=1;
    need_regids<=1;
    need_valC<=0;
end
```

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.

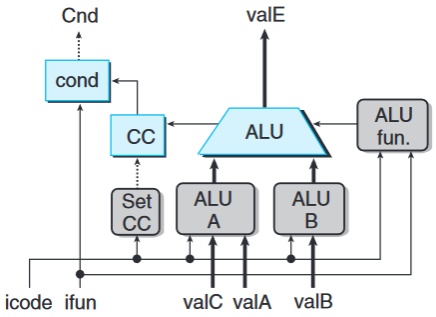
```
else if((icode==4'h2) && (Cnd==1'b1))   //IRRMOVQ
begin
    destM<=4'hF;
    destE<=rB;
end
else if(icode==4'h3)    //IIRMOVQ
begin
    destM<=4'hF;
    destE<=rB;
end
```

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
```

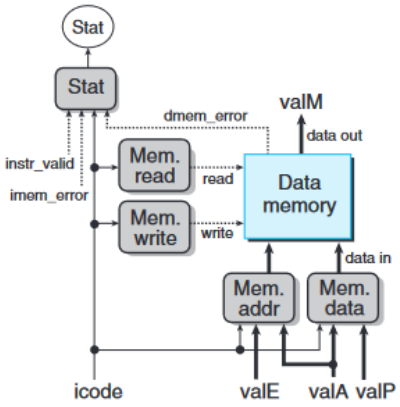
We also read the previous condition code to set up the Cnd in case conditional branch or data transfer should take place(cmov or jmp):

```
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
```

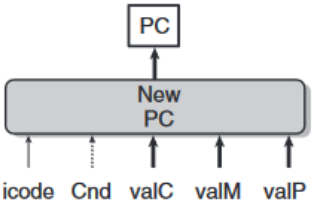
If mem_addr to be read or write to larger than the max data memory, dmem_error will be set to 1 indicating the memory address is out of bounds. Then we can indicate the status of the processor based on these values:

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

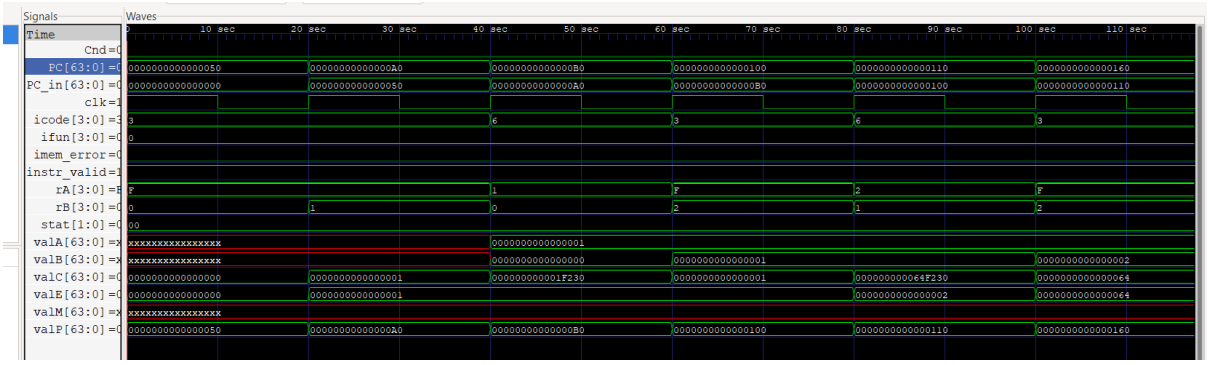


Figure 6: GTKWave output of the program

REG_MEM.txt

```
1   // 0x00000000
2   00000000000013ba
3   0000000000000064
4   0000000000000001
5   0000000000000000
6   0000000000000000
7   0000000000000000
8   0000000000000000
9   0000000000000000
10  0000000000000000
11  0000000000000000
12  0000000000000000
13  0000000000000000
14  0000000000000000
15  0000000000000000
16  0000000000000000
17
```

Figure 7: Register Memory output

13BA in HEX= 5050 in DEC =Σ100

TB for Fetch:

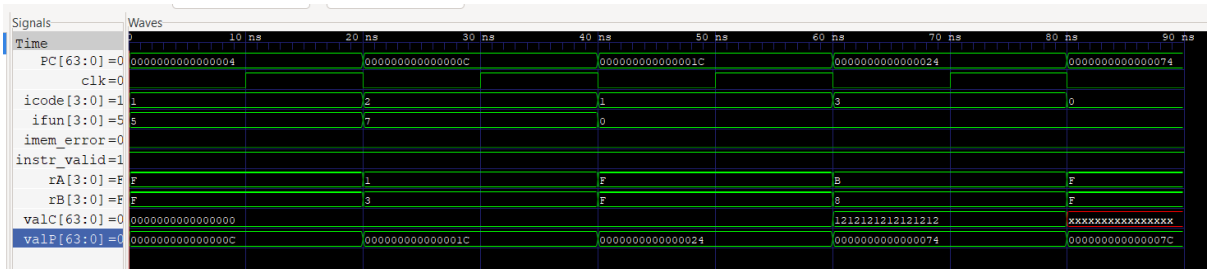
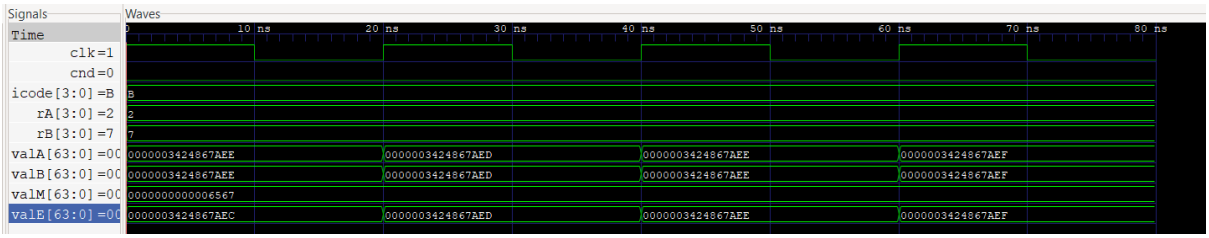


Figure 8: INS:0152713103OB8121212121212121200 PC:4

TB for Decode/WB:



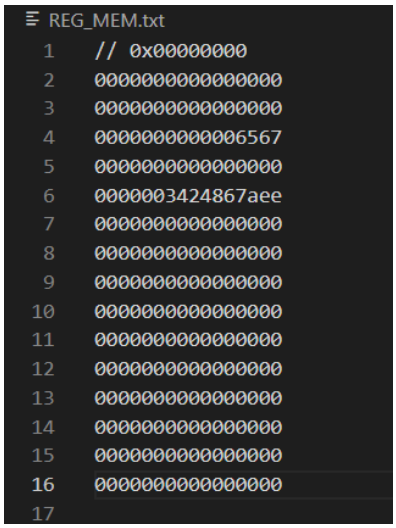


Figure 9: AEF isn't written until next clock cycle; popq:rsp<-valE and rA<-valM

TB for Execute:

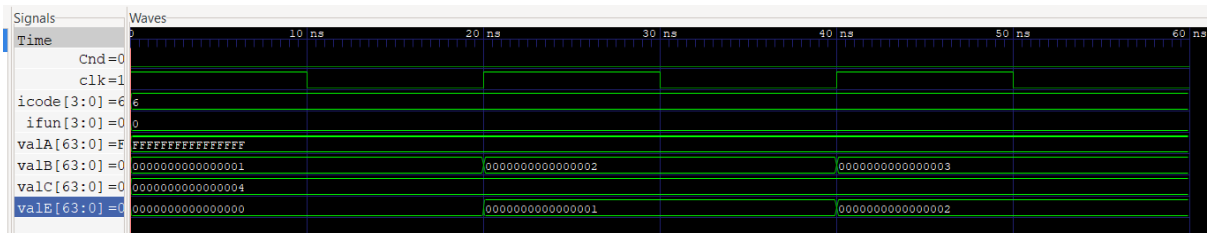


Figure 10: valE=valA+valB; same as old ALU

TB for Memory:

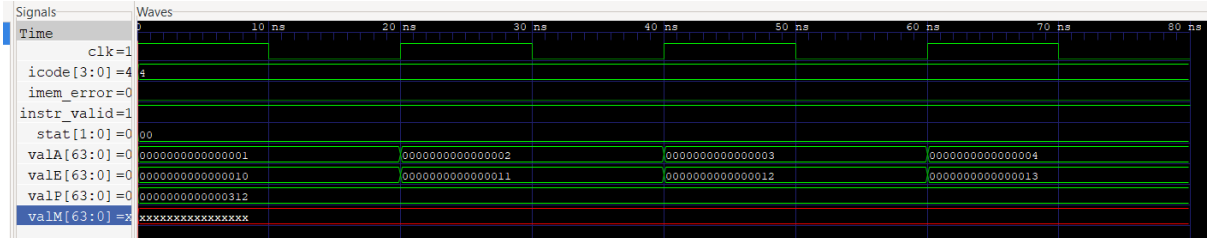


Figure 11: icode 4->rmmovq: M[valE]<-valA

```
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
```