IPA - Project - Final Report

Om Mehta

2022102046

Devansh Kantesaria

2022112003

Y-86-64 Processor:

Our project aim was to build a y86 - x64 processor from scratch. The Y86 processor is a kind of a reduced subset of the x86 instruction set. We have here implemented both the sequential and the pipelined version of the y86 processor. More details about the y86 instruction set is given in the presentation. We will only discuss the implementation in verilog here.

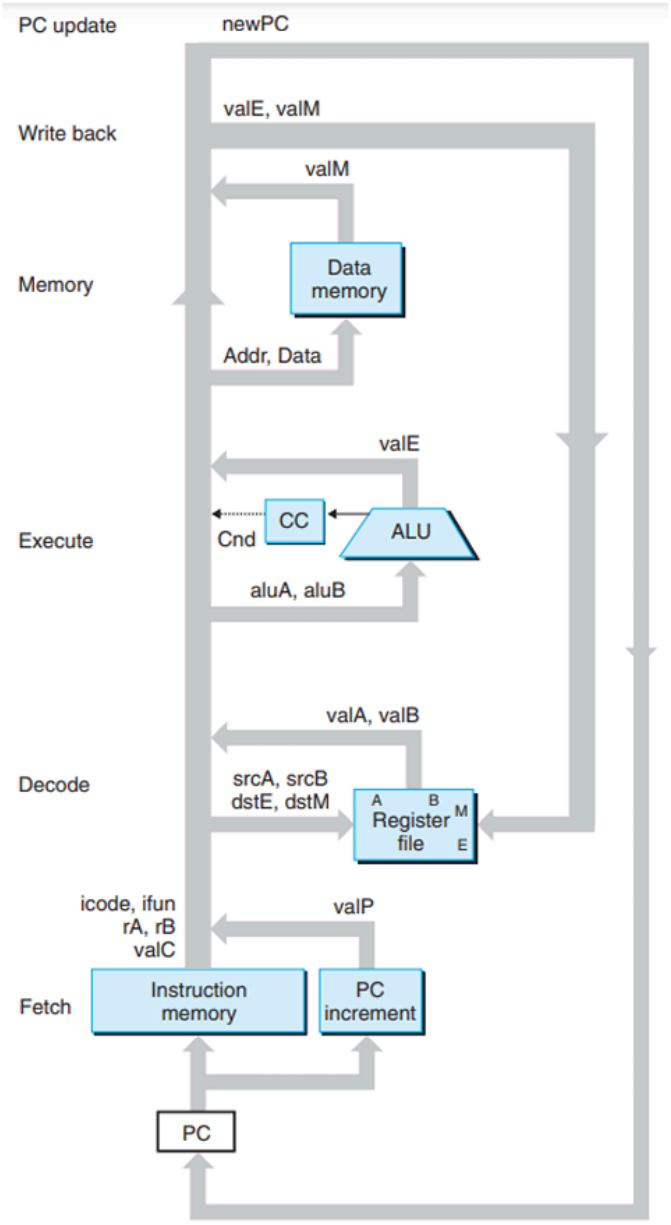
Sequential Processor

The sequential y86 processor diagram is given below. As the name suggests it is a sequential processor hence every instruction is processed on only after the previous instruction is done being processed.

As given below we have 6 stages for our sequential processor,

1. Fetch
2. Decode
3. Execute
4. Memory Read/Write
5. Write-back
6. PC-update

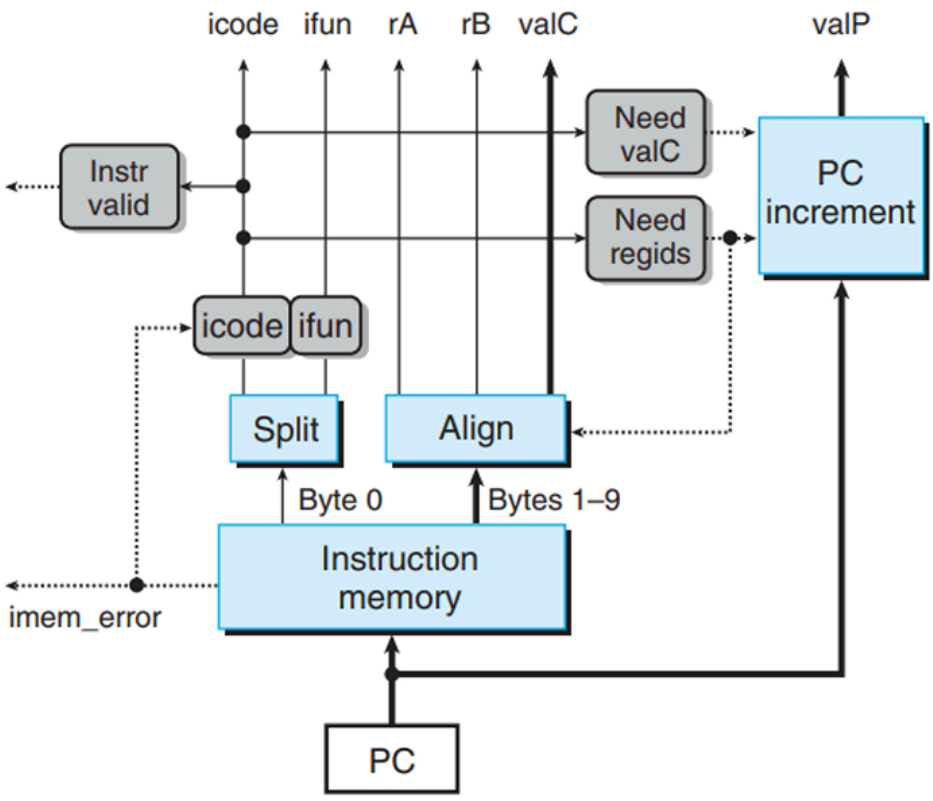
We have explained all the above stage implementation below in our verilog code.



1. Fetch Stage

In our fetch stage we first load our program onto our instruction memory. We have separated instruction memory from the real memory as in real processors we also have similar systems like caches, etc which do this. Also to implement this on verilog we have kept the memory size to 256 byte modules and the instruction memory to 128 byte modules.

Now we also define our program counter variable as 0 initially which will be the PC for our processor. So now we can run our program with this setup.



Our verilog code first checks if the PC value is inside our range or not. If it is outside of our capacity we set the error variable memory_error = 1. So we will then halt our processor if we find that this is true so we will just give back the $finish instruction.

```
invalid_instr=0;
  if(PC > 255)
  begin
      memory_error = 1;
  end
```

Now if we find that we do not have a memory error we can now finally go along with our fetch.

In the fetch stage we will first read the first line and store the two 4 bit parts as icode and ifun. These will define what kind of operation we want to do on the below data.

```
ifun = instruction_memory[PC][3:0];
icode = instruction_memory[PC][7:4];
```

If there is nop or halt signal in the icode and ifun then the processor will just conclude the given instruction. If not then the processor will continue with the next lines where it will read first the two operand registers , rA,rB. Then if required, it will read a 64 bit data piece and store it as valC. Whether to do any of the above is all given in if else statements, an example is given below.

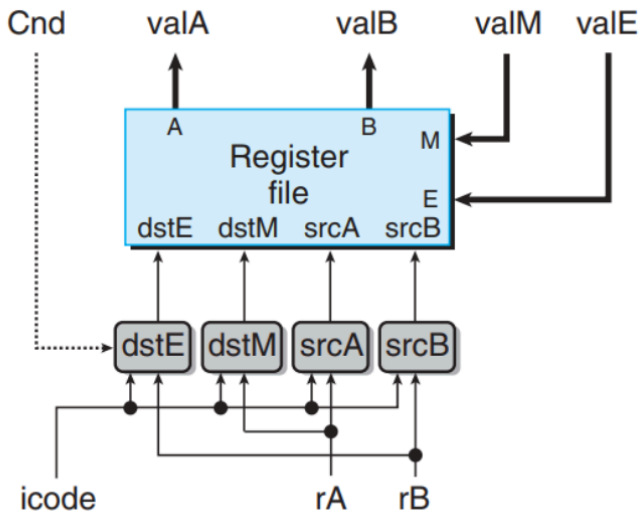
```
else if(icode == 4'b1000) //call
begin

    valP = PC + 64'd9;
    valC[7:0] = instruction_memory[PC+1];
    valC[15:8] = instruction_memory[PC+2];
    valC[23:16] = instruction_memory[PC+3];
    valC[31:24] = instruction_memory[PC+4];
    valC[39:32] = instruction_memory[PC+5];
    valC[47:40] = instruction_memory[PC+6];
    valC[55:48] = instruction_memory[PC+7];
    valC[63:56] = instruction_memory[PC+8];


end
```

Now finally it will increment the PC according to the above read instructions. In case of instructions like return it will set it to the next PC until the other stage sets it to the specified value.

So this concludes our fetching of the instructions.

2. Decode + Write-Back Stage

We have merged the decode stage with the write back stage as both of them require access to the same register file.



First for the decode stage we just decode the values given in rA and rB and store them in valA and valB. This is a simple case of using if else statements for each kind of instruction. We find here that if we do not need valB for example in irmovq we have just kept it x.

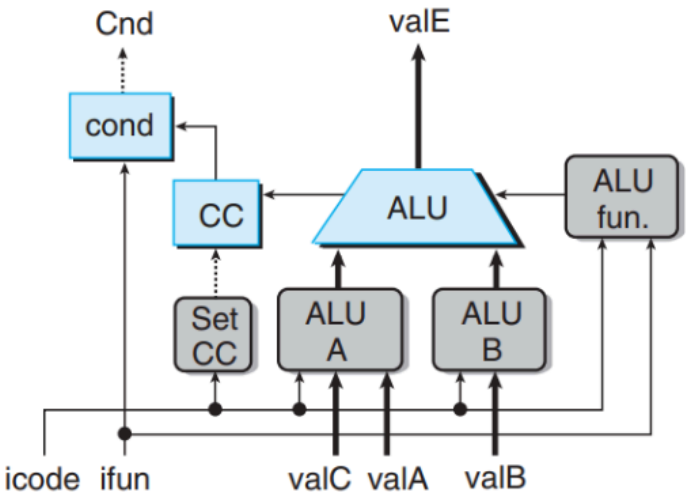
```
if(icode==4'b0010)
begin
    valA = reg_memory[rA];
end

// rmmovq rA, D(rB)
else if(icode==4'b0100)
begin
    valA = reg_memory[rA];
    valB = reg_memory[rB];
end

// call Dest
else if(icode==4'b1000)
begin
    valB = reg_memory[4'b0100];
end

// ret
else if(icode==4'b1001)
begin
    valA = reg_memory[4'b0100];
    valB = reg_memory[4'b0100];
end

// pushq rA
else if(icode==4'b1010)
begin
    valA = reg_memory[rA];
    valB = reg_memory[4'b0100];
end

// mrmovq D(rB), rA
else if(icode==4'b0101)
begin
    valB = reg_memory[rB];
end
```

The write-back stage is also very simple where we just assign the valE values computed from the execute stage to a register which depends on the icode and ifun of the instruction. This is again a case of if else statements as shown below.

```
// cmovxx rA, rB
if(icode==4'b0010)
begin
reg_memory[rB] = valE;
end

// Opq rA, rB
else if(icode==4'b0110)
begin
    reg_memory[rB] = valE;
end

// call
else if(icode==4'b1000)
begin
    reg_memory[4'b0100] = valE;
end

// ret
else if(icode==4'b1001)
begin
    reg_memory[4'b0100] = valE;
end

// irmovq V, rB
else if(icode==4'b0011)
begin
    reg_memory[rB] = valE;
end

// mrmovq D(rB), rA
else if(icode==4'b0101)
begin
    reg_memory[rA] = valM;
end

// pushq rA
else if(icode==4'b1010)
begin
    reg_memory[4'b0100] = valE;
```

3. Execute Stage

This is the main execution of our processor. Here we attach our ALU which does the mathematical operations on our data. We also define our condition codes and how we set and compare for the condition codes here.



We mainly have three condition flags for our Y86 processor - ZF (Zero Flag - output is zero) , SF (Sign Flag - output is negative or not) , OF (Overflow Flag - output is out of bounds or not). In order to update any of these flags, we have used again if else statements here for each of these flags as given below.

```
always @(*)
begin
    if(((e_icode == 6) && set_cc))
    begin

        if(alu_output==0) ZF=1;
        else ZF=0;

        if(alu_output[63] == 1) SF=1;
        else SF=0;

        if((A[63] == B[63]) && (alu_output[63] != A[63])) OF=1;
        else OF=0;

    end

end
```

Now for the execution of the instruction, we will again use if else statements to check whether the ALU has to do something or not. The output of the execution is stored in the variable ValE which will be forwarded to the Memory and the Write - Back Stage. Here we will either use the ALU for determining the ValE or we will use the Flags for our condition codes if we want to take a branch or not. Examples are:

```
    if(ifun==4'b0000)     // jmp
begin
    condition = 1; // unconditional jump
end
if(ifun==4'b0001)     // jle
begin

    if((SF^OF)|ZF)
        condition = 1;


end
if(ifun==4'b0010)     // jl
begin

    if(SF^OF)
        condition = 1;


end
 if(ifun==4'b0011)     // je
begin
    if(ZF)
        condition = 1;
     else
        condition = 0;
end
```

The output of the condition is stored in the condition variable. If the instruction is unconditional we will use the ALU and store its output in ValE.

ALU:

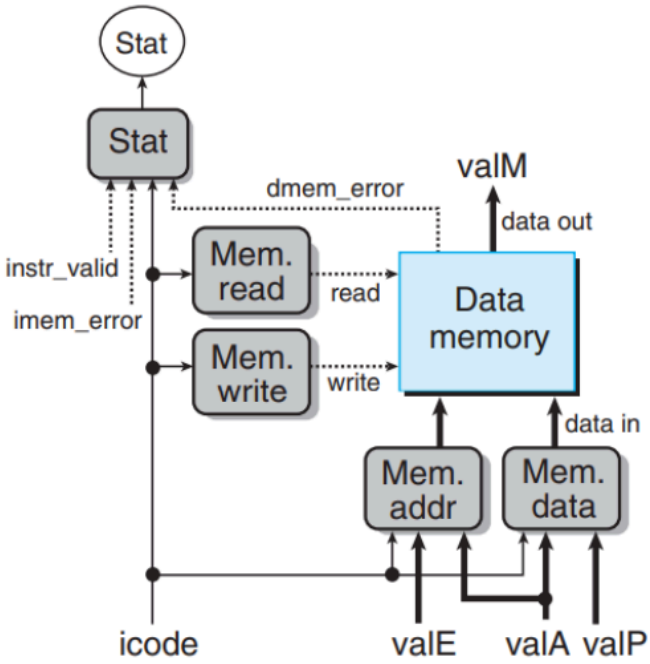
As made many times, it comes with the operations Add, Subtract,Compare,AND and many more. Here we have only made it for Add,Sub,XOR and AND.

4. Memory Stage

In the memory stage we just read or write a value from the memory depending on the icode and the ifun of the instruction. For writing this address value is given by ValE again depending on the icode and ifun (And Hence if-else statements).

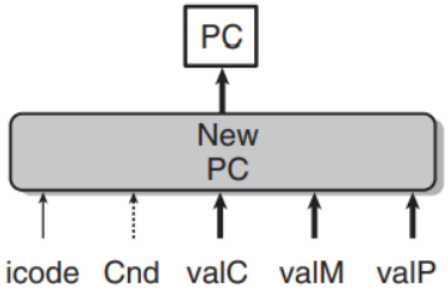
```
if(icode == 4'b0101) //mrmovq
begin
if(valE>255) data_error=1;
else valM = memory[valE];
end

else if(icode == 4'b1011) //popq
begin
if(valA>255) data_error=1;
else valM = memory[valA];
end

else if(icode == 4'b1001) //ret
begin
if(valA>255) data_error=1;
else valM = memory[valA];
end
else if(icode == 4'b0100) //rmmovq
begin
if(valE>255) data_error=1;
else memory[valE] = valA;
end

else if(icode == 4'b1000) //call
begin
if(valE>255) data_error=1;
else memory[valE] = valP;
end

else if(icode ==  4'b1010) //pushq
begin
if(valE>255) data_error=1;
else memory[valE] = valA;
end
```

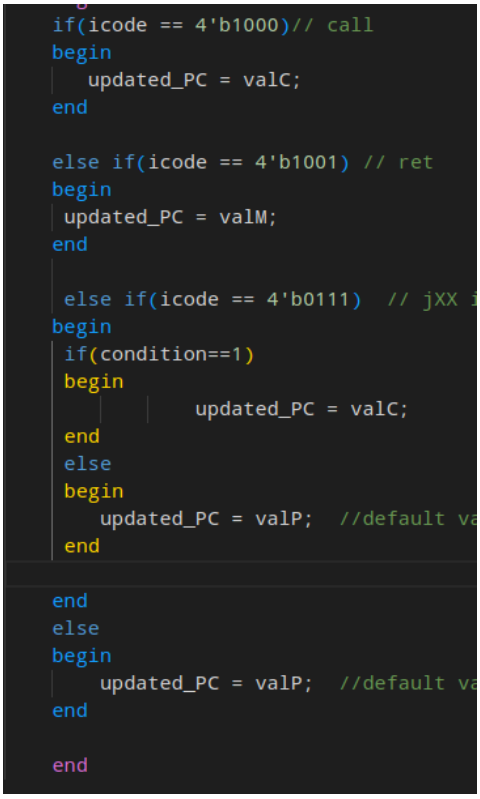
If we want to read from the memory, the read data here is stored in a special variable called ValM. The read data is mostly used to write to a register.



5. PC-update

Finally we modify ValP as per the given icode and ifun. This will only be different from +1 if there is a Call, Ret, or Jxx instruction. The code is shown below.





```
if(icode == 4'b1000)// call
begin
   updated_PC = valC;
end

else if(icode == 4'b1001) // ret
begin
 updated_PC = valM;
end

 else if(icode == 4'b0111)  // jXX
begin
 if(condition==1)
 begin
          updated_PC = valC;
 end
 else
 begin
    updated_PC = valP;  //default va
 end

end
else
begin
    updated_PC = valP;  //default va
end

end
```

Pipelined Processor

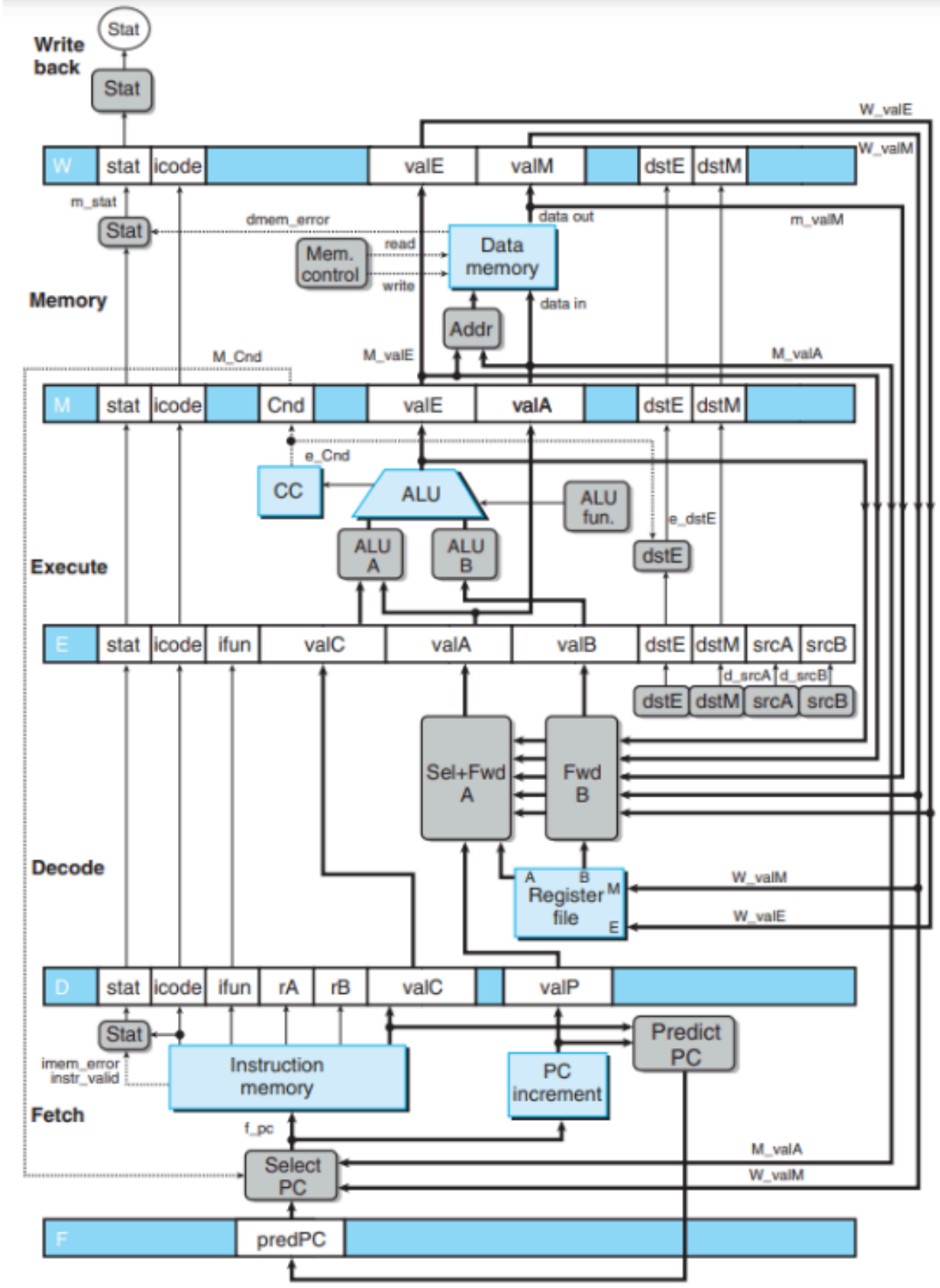
We find that the sequential implementation of the processor is very inefficient as multiple stages of the processor are sitting idle when the instruction passes through a particular stage. This problem can be solved through the idea of pipelining where we concurrently run multiple instructions on the same 5 stages one by one parallelly.

So we basically have the same sequential processor as above but there some changes to it:

1. Addition of Pipelining registers to keep track of the next instruction data.
2. The addition of Data Forwarding to solve inter dependencies on data
3. The addition of Stalling and Bubbling to solve inter dependencies\
4. Prediction of PC as a necessity for the working
5. Controlling Hazards in the pipeline.

We will now explain each of the stage changes here and tell which have changed by which ways. Before going there we have renamed all our variables to individual stages.

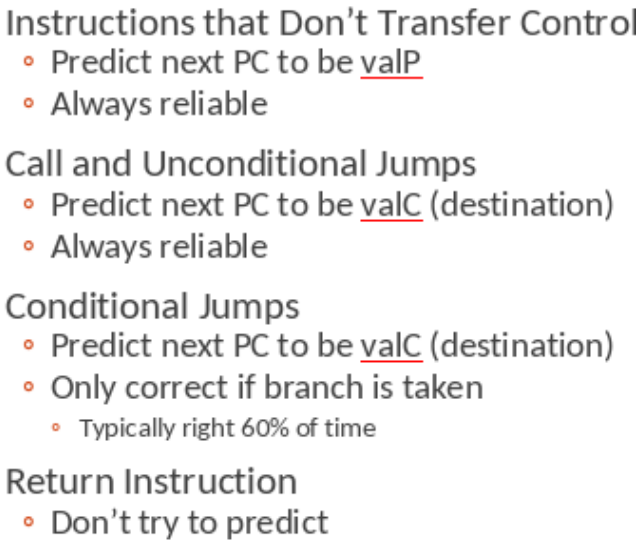
For example icode to f_icode, valA to f_valA etc. Below shows the entire diagram of the pipelined y86 processor:


Write back
Stat
Stat
W_valE
W_valM
W
stat
icode
valE
valM
dstE
dstM
m_stat
Stat
dmem_error
data out
m_valM
Mem. control
read
write
Data memory
Memory
data in
Addr
M_Cnd
M_valE
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
e_dstE
dstE
ALU A
ALU B
Execute
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
dstE
dstM
srcA
srcB
Sel+Fwd A
Fwd B
Decode
A
B
Register file
M
E
W_valM
W_valE
D
stat
icode
ifun
rA
rB
valC
valP
Stat
Predict PC
imem_error
instr_valid
Instruction memory
PC increment
Fetch
f_pc
M_valA
W_valM
Select PC
F
predPC

1. Fetch Stage and Prediction of PC:

The fetch stage main content remains the exact same except for the prediction and misprediction code of the PC. The problem here is that when we insert the instruction, we do not know if it is the one we should be executing next as there are many other instructions like jXX, ret etc. which branch out PC value to something else. Hence we find the need to handle this type of behaviour in our processor.

For the prediction we store the predicted PC value in the F_pc_prelim which conveys the guessed value of the PC when there is no stalling in the processor. This is decided based on the following logic:



Below is our implementation of this above:

```
if(M_icode == 7)begin
    if(M_cnd==0)
    begin
    f_pc = M_valA;
    end
end
else if(W_icode == 9)
begin
    f_pc = W_valM;
end
else
begin
    f_pc = F_pc_prelim;
end
```

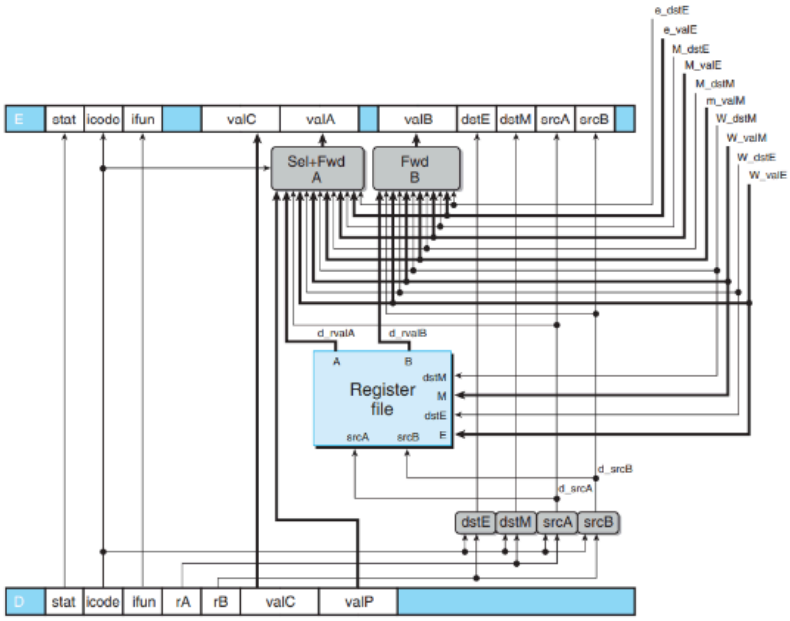
We do it like this as there is a very good chance as given above that this will be correct.

In case of Misprediction we need to take out the existing instructions or stall or bubble the processor until the next PC is determined. This is shown below after detecting misprediction:

```
if(f_icode==7 || f_icode==8)  f_predPC = f_valC;
else f_predPC = f_valP;
```

2. Decode + Write_Back stage

Here we forward most of our register parameters from the fetch stage except we also add the ValA and ValB values to the pipeline registers.



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. 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.

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

If none of the forwarding conditions hold, the block should select d_rvalA, the value read from register port A, as its output. The implementation in our ode is given below:

```
always @(posedge clk) begin
    if(W_stat==0 && W_dstM!=15)  registers[W_dstM] = W_valM;
    if(W_stat==0 && W_dstE!=15)  registers[W_dstE] = W_valE;

end
```

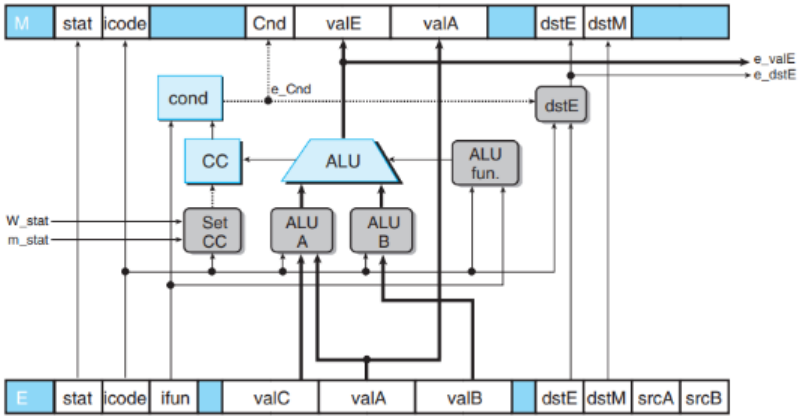
So now instead of storing it in ValA and ValB we store everything in d_srcA,d_srcB,d_dstE and d_dstM. An example of irmovq is shown below:

```
else if(d_icode==3) //irmovq
begin
        d_srcA = 15;
        d_srcB = D_rB;
        d_dstE = D_rB;
        d_dstM = 15;

end
```

Also below is our implementation of Data Forwarding to the Decode stage:

```
if(d_srcA != 15) begin

    if(e_dstE == d_srcA)
    begin
        d_valA = e_valE;
    end
    else if(M_dstM == d_srcA) begin
        d_valA = m_valM;
    end
    else if(M_dstE == d_srcA) begin
        d_valA = M_valE;
    end
    else if(W_dstE == d_srcA) begin
        d_valA = W_valE;
        end
    else if(W_dstM == d_srcA) begin
        d_valA = W_valM;
         end
end
```

3. Execute Stage:



We see that the ALU and the condition codes part still remains the same. The only difference here is that the notation of the variables has changed and data forwarding has been added. Data Forwarding and other aspects have been covered in the next section control module of the Processor. Below is our implementation of it:

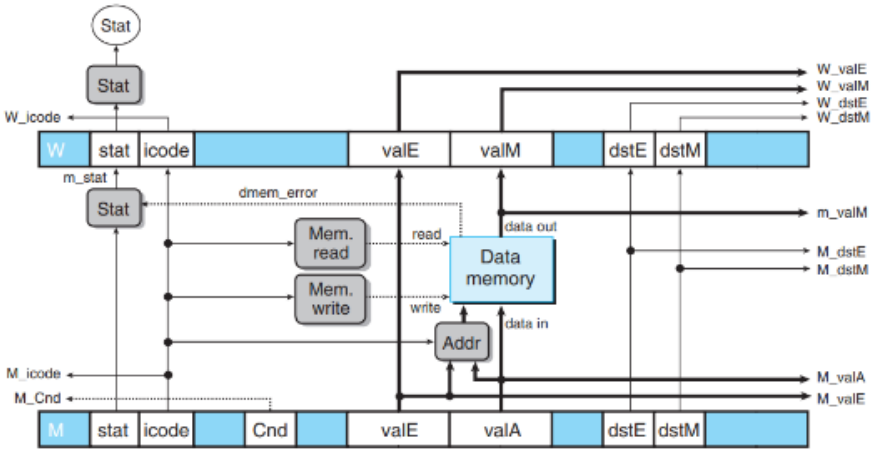
```
if(e_icode==2)//cmovxx
begin
    A = E_valA;
    B = E_valB;

     S1=0;
    S0=0;


    if(E_ifun==0) e_cnd = 1; //uncondtional
    if(E_ifun==1)e_cnd = (SF ^ OF) | ZF;// cmovle
    if(E_ifun==2) e_cnd = SF ^ OF;// cmovl
    if(E_ifun==3) e_cnd = ZF;// cmove
    if(E_ifun==4)  e_cnd = ~ZF;// cmovne
    if(E_ifun==5) e_cnd = ~(SF ^ OF); // cmovge
    if(E_ifun==6) e_cnd = ~((SF ^ OF) | ZF);// cmovg

end

if(e_icode==3)   e_valE = E_valC; //irmovq

if(e_icode==4) //rmmovq
begin
   A = E_valB;
            B = E_valC;

             S1=0;
            S0=0;
```

```
if(e_icode==3)   e_valE = E_valC; //irmovq

if(e_icode==4) //rmmovq
begin
   A = E_valB;
            B = E_valC;

             S1=0;
            S0=0;

end
if(e_icode==5)//mrmovq
begin
   A = E_valB;
            B = E_valC;

             S1=0;
             S0=0;

end

if(e_icode==6) //opq
begin
    A = E_valB;
            B = E_valA;


             S1=E_ifun[1];
             S0=E_ifun[0];

end
```

```
if(e_icode==7) //jxx
begin

  if(E_ifun==0)  e_cnd = 1; // unconditional
  if(E_ifun==1)  e_cnd= (SF ^ OF) | ZF;// jle
  if(E_ifun==2)  e_cnd = SF ^ OF;// jl
  if(E_ifun==3)  e_cnd = ZF;// je
  if(E_ifun==4)  e_cnd = ~ZF;// jne
  if(E_ifun==5)  e_cnd = ~(SF ^ OF); // jge
  if(E_ifun==6)  e_cnd = ~((SF ^ OF) | ZF);// jg

end

if(e_icode==8) //call
begin
     A = E_valB;
            B = -8;

            S1=0;
           S0=0;


end

if(e_icode==9)//ret
begin
    A = E_valB;
    B = 8;

    S1=0;
    S0=0;


end
```

```
    if(e_icode==10)//pushq
    begin

        A = E_valB;
        B = -8;
         S1=0;
        S0=0;



    end

    if(e_icode==11) //popq
    begin

                A = E_valB;
                B = 8;

                 S1=0;
                S0=0;



    end


    if(e_icode!=7 && e_icode!=3)
    begin
      e_valE = alu_output;
    end

end
```

4. Memory Stage:



Comparing the Pipelined memory stage to the memory stage for SEQ, we see that, the block labeled “Mem. data” in SEQ is not present in PIPE. This block served to select between data sources valP (for call instructions) and valA, but this selection is now performed by the block labeled “Sel+Fwd A” in the decode stage. Most other blocks in this stage are identical to their counterparts in SEQ, with an appropriate renaming of the signals. In this figure, we can also see that many of the values in pipeline registers and M and W are supplied to other parts of the circuit as part of the forwarding and pipeline control logic.

We see that other than this and the change in notation the Memory stage is almost the same as in sequential. Also we have changed the implementation of the accessing of the memory block as shown below:

```
m_valM = {memory_block[(M_valE)+7],
          memory_block[(M_valE)+6],
          memory_block[(M_valE)+5],
          memory_block[(M_valE)+4],
          memory_block[(M_valE)+3],
          memory_block[(M_valE)+2],
          memory_block[(M_valE)+1],
          memory_block[M_valE]};
```

This was a mistake in our sequential implementation which we then corrected here.

We also forward the dste and dstm:

```
always @(*) begin
    if(mem_error==0) m_stat = M_stat;

    m_icode = M_icode;
    m_valE = M_valE;
    m_dstE = M_dstE;
    m_dstM = M_dstM;
end
```

5. Control Logic:

This module is the main difference between the sequential implementation and the pipelined implementation. Here we mainly denote all the necessary conditions for stalling and bubbling of the processor. Below is our implementation of it:

```
//F_stall conditions
if((D_icode == 9 || E_icode == 9 || M_icode == 9) || ((E_icode == 5 || E_icode == 11) && (E_dstM == d_srcA || E_dstM == d_srcB)))   F_stall = 1;
else F_stall=0;


//D_stall conditions
if((E_icode == 5 || E_icode == 11) && (E_dstM == d_srcA || E_dstM == d_srcB)) D_stall = 1;
else D_stall=0;

//D_bubble conditions
if((E_icode == 7 && !e_cnd) || (!((E_icode == 5 || E_icode == 11) && (E_dstM == d_srcA || E_dstM == d_srcB)) && (D_icode == 9 || E_icode == 9 ||
else D_bubble=0;

//E_bubble conditions
if((E_icode == 7 && !e_cnd)|| ((E_icode == 5 || E_icode == 11)&& (E_dstM == d_srcA || E_dstM == d_srcB))) E_bubble = 1;
else E_bubble = 0;


//M_bubble conditions
if((m_stat == 1 || m_stat == 2 || m_stat == 3) || (W_stat == 1 || W_stat == 2 || W_stat == 3)) M_bubble = 1;
else M_bubble = 0;

//W_stall conditions
if(W_stat == 1 || W_stat == 2 || W_stat == 3)  W_stall = 1;
else W_stall=0;


//condition conditions
if((E_icode == 6) && !(m_stat!=0) && !(W_stat != 0)) condition = 1;
else condition = 0;
```

We see that the stalling and bubbling is only done when a few select conditions are met. This includes conditions when there is a jump, call ,return or cmovxx instructions. We mostly resolve the remaining data interdependencies here by introducing bubbling and stalling.

6. Final Compilation

Finally we have compiled all of these together with the main pipelining module. This just connects all of the modules together.We have also executed the remaining part of bubble and stalling as shown below:

```
    if(M_bubble==0)
     begin
        M_stat <= e_stat;
        M_cnd <= e_cnd;
        M_valE <= e_valE;
        M_valA <= e_valA;
        M_icode <= e_icode;
        M_dstE <= e_dstE;
        M_dstM <= e_dstM;
    end

    else begin
        M_stat <= 0;
        M_cnd <= 0;
        M_valE <= 0;
        M_valA <= 0;
        M_icode <= 1;
        M_dstE <= 15;
        M_dstM <= 15;
    end
end


always @(posedge clk ) begin

    if(W_stall==0)
    begin
        W_stat <= m_stat;
        W_icode <= m_icode;
        W_valE <= m_valE;
        W_valM <= m_valM;
        W_dstE <= m_dstE;
        W_dstM <= m_dstM;
    end
end
```

```
if(F_stall==0)
begin
 F_predPC = f_predPC;
end
end


  always @(posedge clk )
   begin

    if(D_stall == 0)
    begin



    if(D_bubble == 1)
    begin
        D_stat <= 0;
        D_icode <= 1;
        D_ifun <= 0;
        D_rA <= 15;
        D_rB <= 15;
        D_valC <= 0;
        D_valP <= 0;

    end

    else if(f_stat != 0) begin
        D_stat <= f_stat;
        D_icode <= 0;
        D_ifun <= 0;
        D_rA <= 15;
        D_rB <= 15;
        D_valC <= 0;
        D_valP <= 0;
```

```
        else begin
            D_stat <= f_stat;
            D_icode <= f_icode;
            D_ifun <= f_ifun;
            D_rA <= f_rA;
            D_rB <= f_rB;
            D_valC <= f_valC;
            D_valP <= f_valP;

        end
       end
    end


lways @ (posedge clk) begin

    if(E_bubble==1)
    begin

        E_stat <= 0;
        E_valC <= 0;
        E_valB <= 0;
        E_valA <= 0;
        E_icode <= 1;
        E_dstE <= 15;
        E_dstM <= 15;
        E_srcA <= 15;
        E_srcB <= 15;

    end
```

Conclusion:

Hence, in this way were able to make our y86 processor work in verilog and run it in both the Sequential version and the Pipelined version. We also came across many challenges such as the verbose code length and the syntax issues for many other types of code. Hence, this project helped us understand better how a y86 processor works and also gave us more experience with verilog.