Y86_PIPELINED_REPORT

Member 1 : Name : Mallika Garg

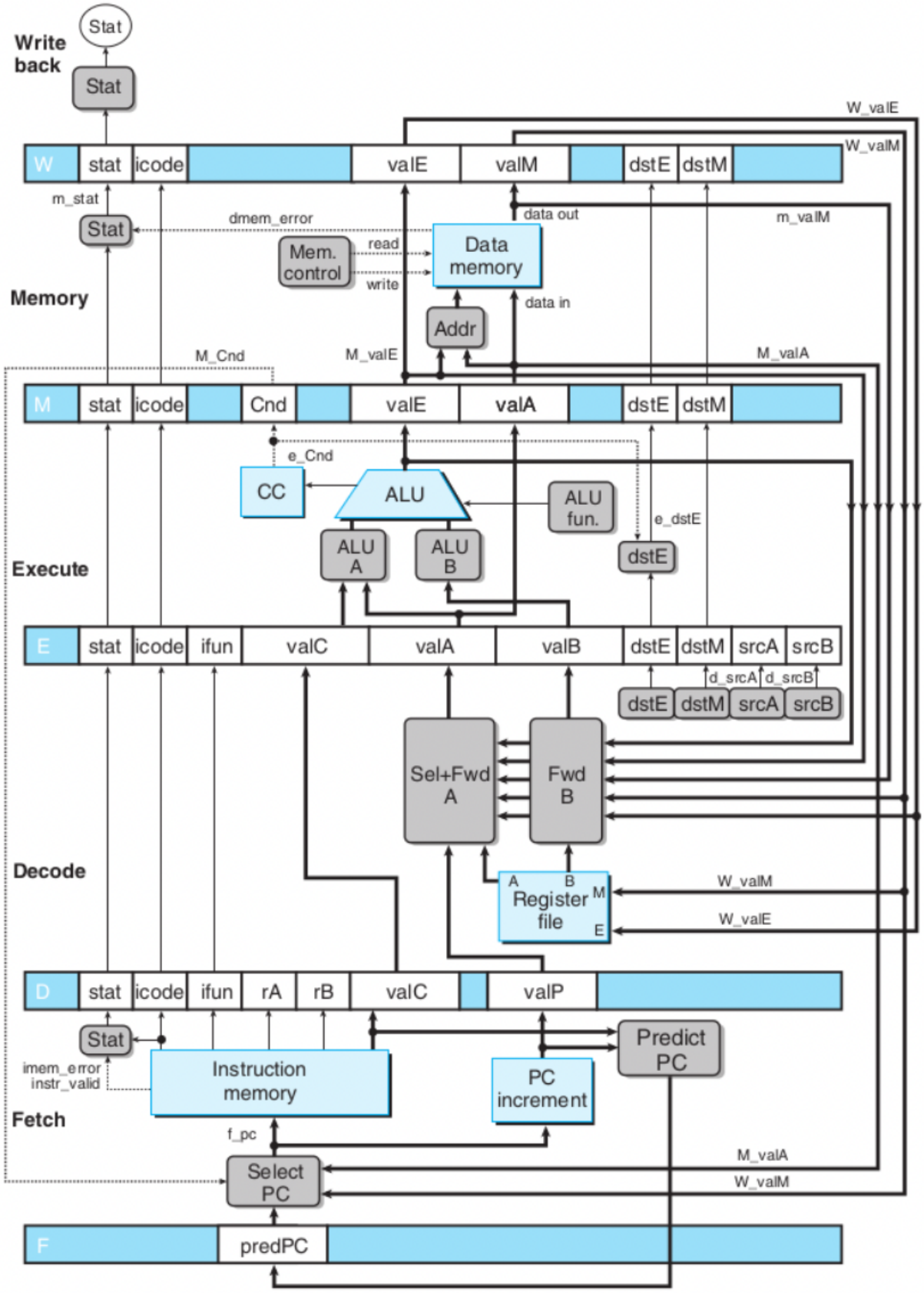
Roll No : 2021102012

Member 1 : Name : Nipun Goyal

Roll No : 2021102029

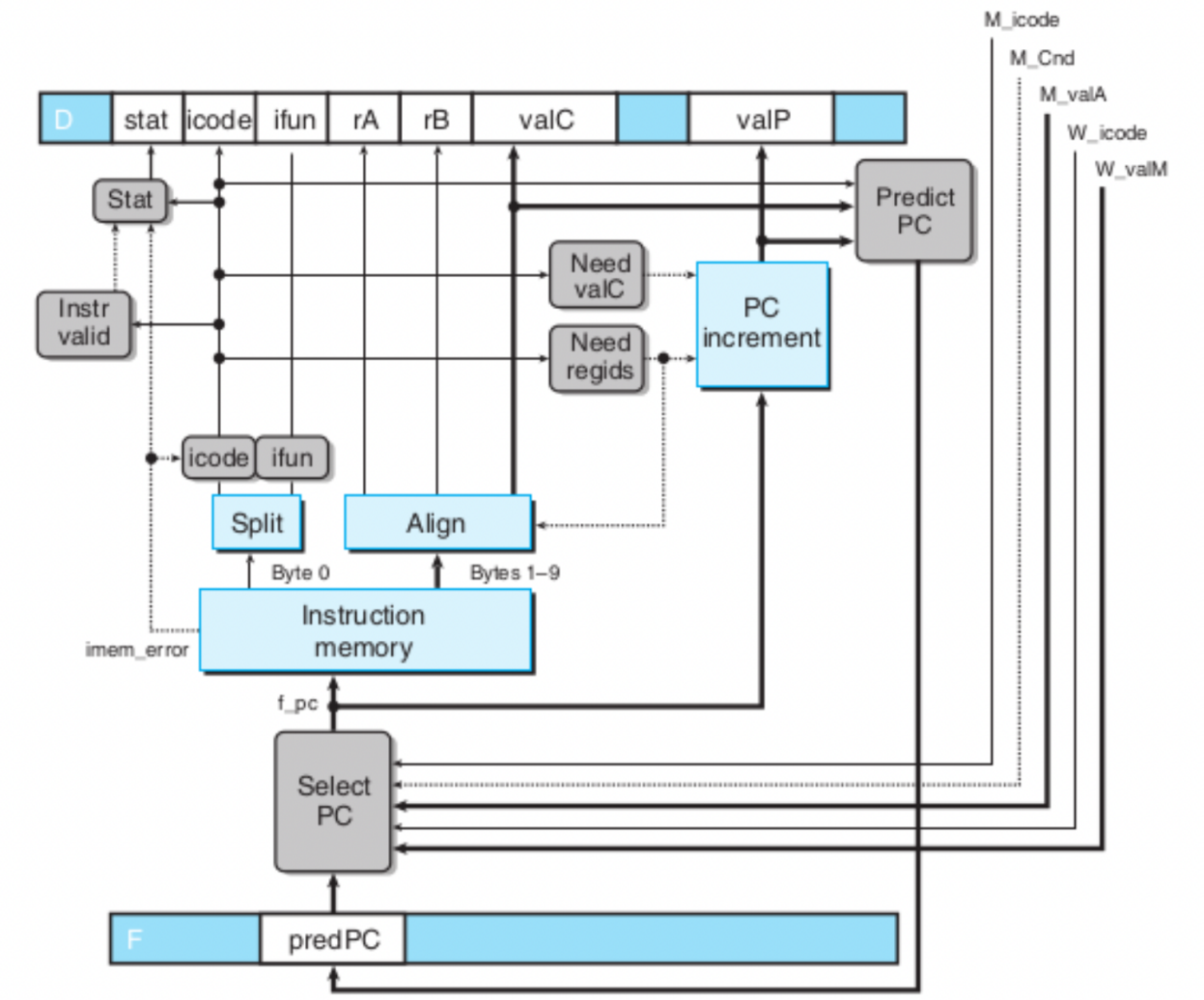
Motivation

The SEQ implementation of our processor was too slow and so we modified the design of our processor by adding registers between each stage and by using techniques such as forwarding , stalling and addition of bubbles to rectify the predicted PC.
The diagram below shows how each stage in connected with other using pipelining registers.


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
imem_error
instr_valid
Instruction memory
PC increment
Predict PC
Fetch
f_pc
M_valA
W_valM
Select PC
F
predPC

Working Of Stages

FETCH



This stage, depending on f_pc, will read 10 bytes and decide how many bytes it actually has to read. On the basis of this decision, the Predict PC block predicts the location of the next instruction in the instruction memory based on the valP or valC values. After that SelectPC decides the PC for the current stage based on the values of valM or M_Cnd or pred_PC.
The instruction memory is implemented by using an 8 X 147 memory array

and the values in the memory are initialized at the beginning of the first cycle of the processor.

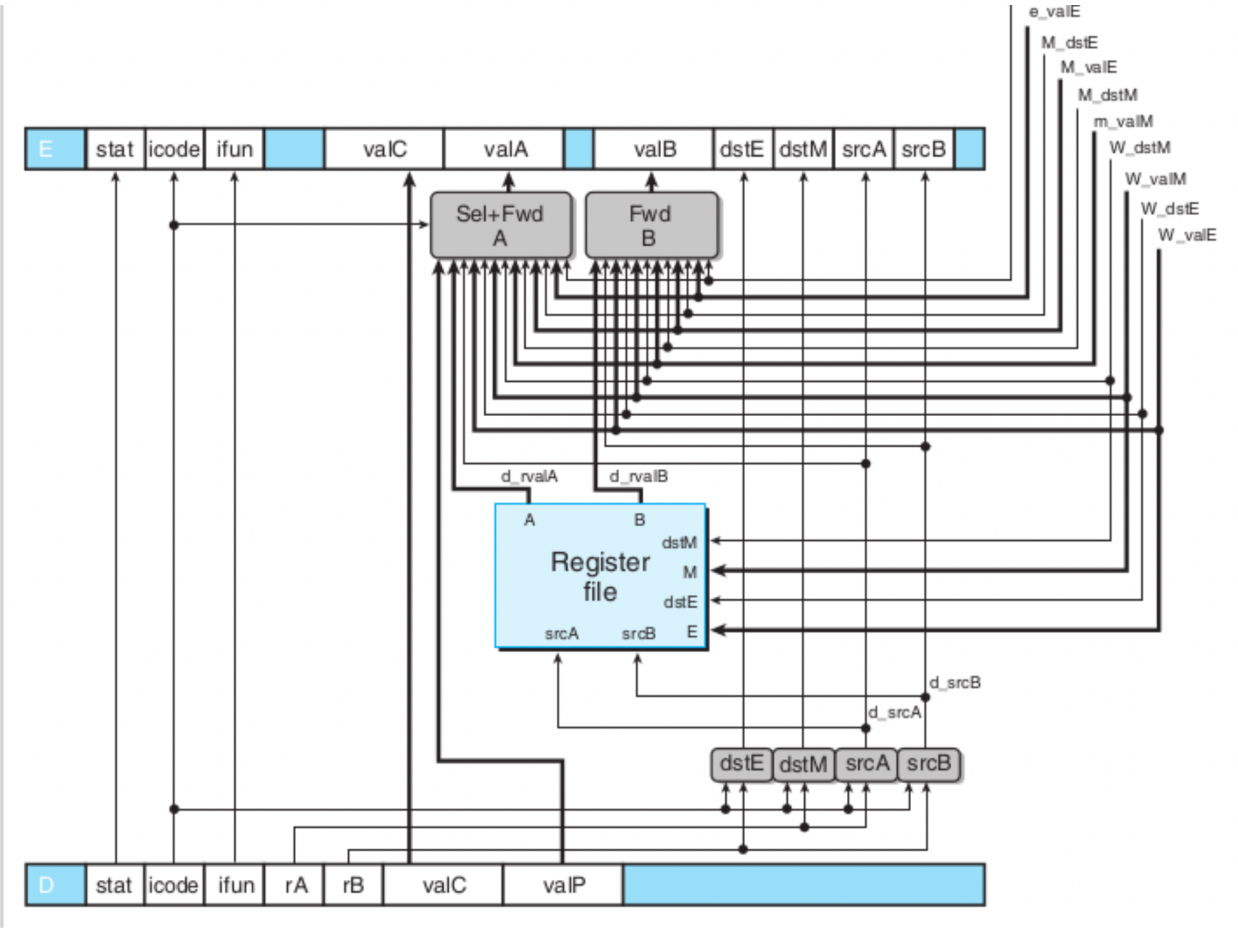
Another function that it performs is to split the instruction between icode,ifun,rA,rB,valC if required based on which other stages perform their respective tasks.

```
instr={
        instruction[f_pc],
        instruction[f_pc+1],
        instruction[f_pc+2],
        instruction[f_pc+3],
        instruction[f_pc+4],
        instruction[f_pc+5],
        instruction[f_pc+6],
        instruction[f_pc+7],
        instruction[f_pc+8],
        instruction[f_pc+9]
      };
```

```
f_icode=instr[0:3];
f_ifun=instr[4:7];
```

```
if(f_icode==4'd3)
    begin

        f_rA<=instr[8:11];
        f_rB<=instr[12:15];
        f_valC<=instr[16:79];
    end
```

DECODE



This stage, on the basis of the D pipelining registers, decides what value to pass to the E pipelined registers, ie,

- d_ValA
- d_ValB
- dstE
- dstM
- srcA
- srcB
  Also, since there can be load/use hazards we need to add two more blocks namely
- Sel + Fwd A
- Fwd B

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. Similar functionality is of Fwd B.

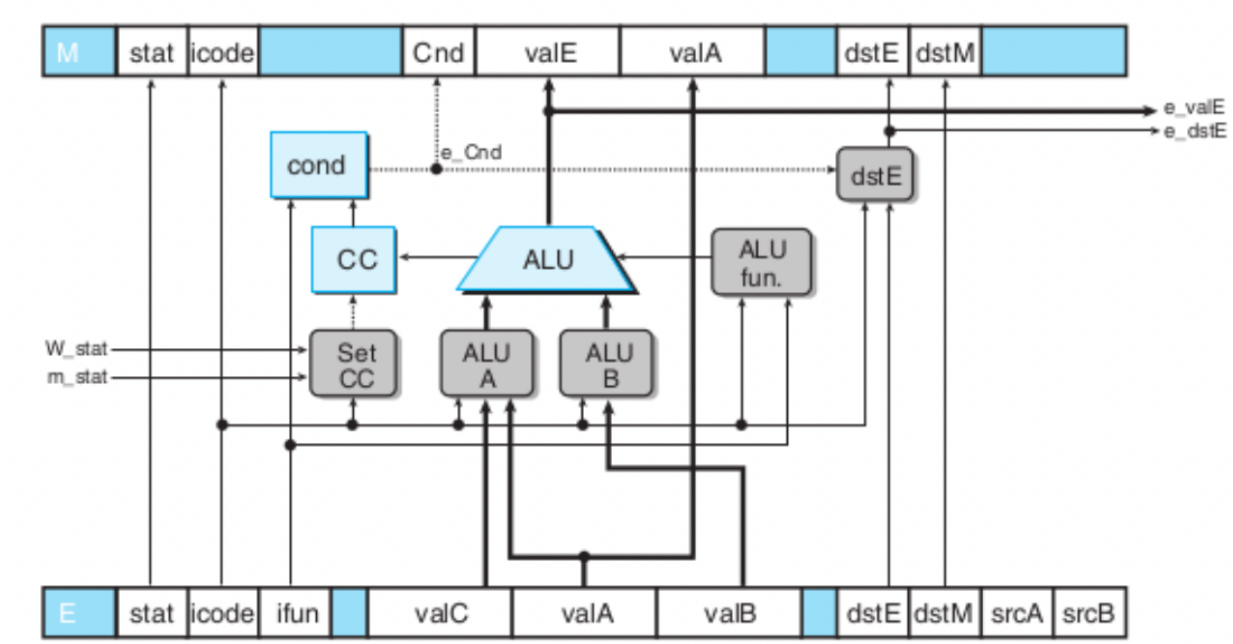
The merging of signals valA and valP exploits the fact that only the call and jump instructions need the value of valP.
The priority order for Sel + Fwd A is :

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

The reason for the above priorities is that we need the most recent values to get forwarded to the decode stage.

```
always @ (posedge clk)
begin
    #50
    if(d_srcA!=4'd15)
    begin
        d_rvalA <= Registors[d_srcA] ;
    end

    if(d_srcB!=4'd15)
    begin
        d_rvalB <= Registors[d_srcB] ;
    end

end
always @(posedge clk)
begin
    #1

    if(W_icode==4'd9)
    begin
        Registors[W_dstE] <= W_valE;
    end
    else
    begin

    if(W_dstE!=4'd15)
    begin
        Registors[W_dstE] <= W_valE;

    end

    if(W_dstM!=4'd15)
    begin
        Registors[W_dstM] <= W_valM;
    end

    end
```

EXECUTE

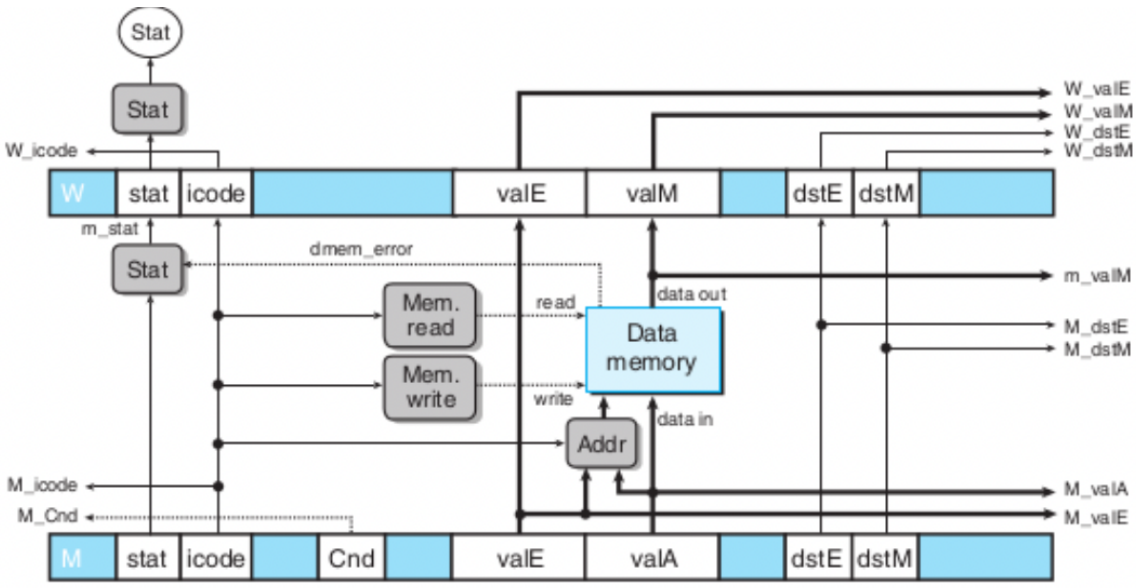


The hardware units and the logic blocks are identical to those in SEQ architechture. We can see the signals e_valE and e_dstE directed toward the decode stage as one of the forwarding sources. One difference is that the logic labeled “Set CC,” which determines whether or not to update the condition codes, has signals m_stat and W_stat as inputs.

e_dstE also gets updated based on the values of icode , if icode is 2 or 0 it becomes F otherwise passes E_dstE.

```
else if((E_icode == 4'b0010)&&(E_ifun == 4'b0100)) //cmovne
    begin
        if(~zf)
        begin
            e_Cnd<=1'b1;
        end
    end

else if((E_icode == 4'b0010)&&(E_ifun == 4'b0101)) //cmovge
    begin
        if(~(sf^of))
        begin
            e_Cnd<=1'b1;
        end
    end

else if((E_icode == 4'b0010)&&(E_ifun == 4'b0110)) //cmovg
    begin
        if((~(sf^of)&(~zf)))
        begin
            e_Cnd<=1'b1;
        end
    end
```

MEMORY

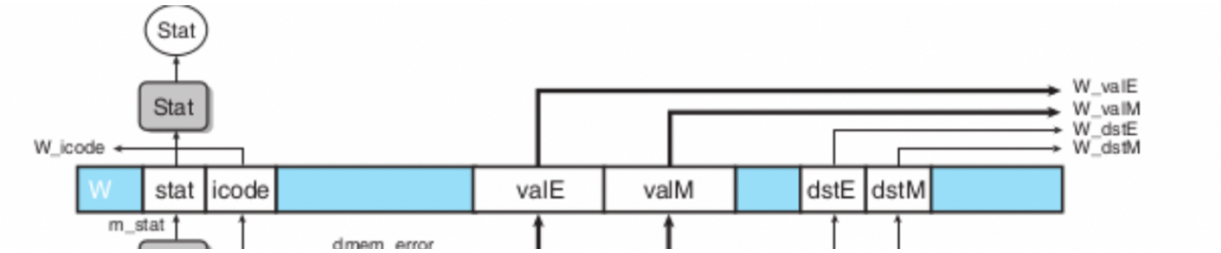


This stage has the main component which is Data memory where data can be stored. The read and write wires determine which operation has to be followed and mem_addr is the address of the memory location where data has to be written ie “M_valA” or read, ie, to m_valM. It also passes the necessary data to the next pipeline registers.

It doesn't pass Cnd to the next stage because it is useless in the next stage. valM, dstE, dstM valA, and valE also gets feedback from this stage to decode for forwarding conditions.

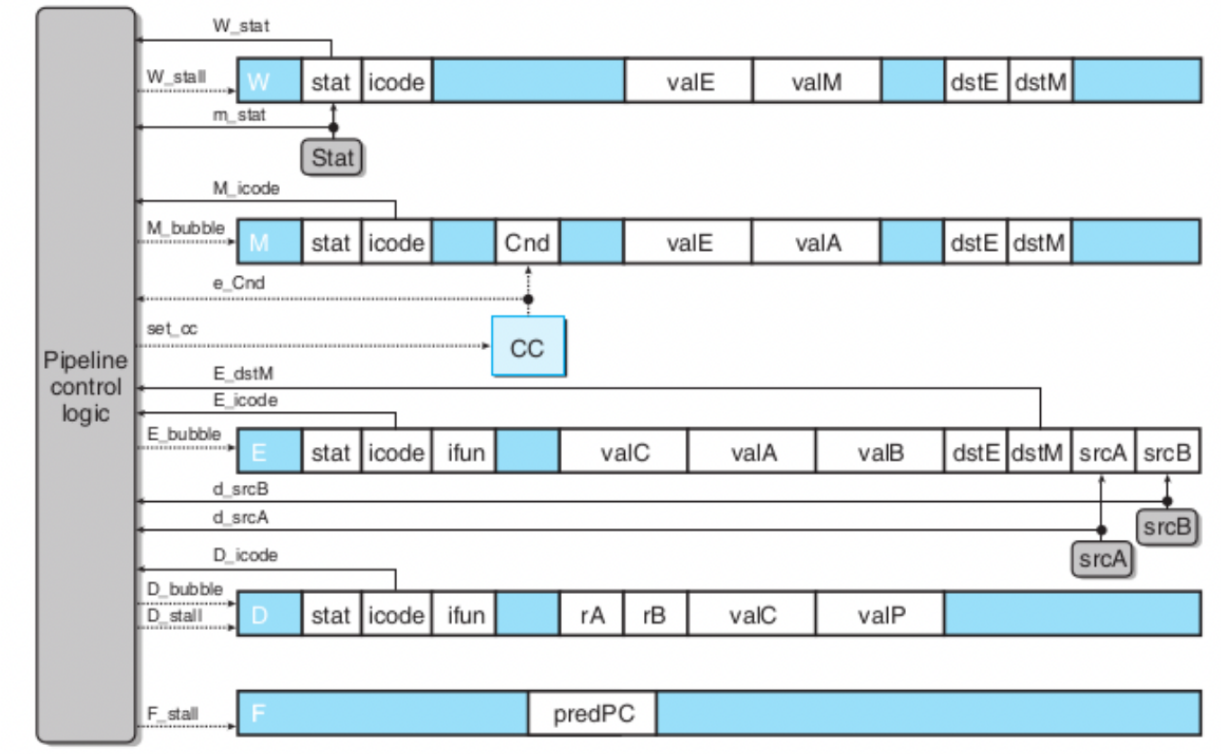
```
always @ (posedge clk)
begin
    #10
    if (read==1'b1)
    begin
        if (write==1'b0)
        begin
            m_valM <= memory[Addr_out];
        end
    end

end

always @(posedge clk)
begin
    #10
    if (write==1'b1)
    begin
        if (read==1'b0)
        begin
            memory[Addr_out] <= M_valA;
        end
    end
end
```

WRITE-BACK



This stage of my implementation contains only register because based on the value of dstE and dstM and conditions we store values in register file which is present in decode stage where we can store valE and valM values.

CONTROL LOGIC



My processor handles these four case with the help of this pipeline logic :

- Load/use hazards. The pipeline must stall for one cycle between an instruction that reads a value from memory and an instruction that uses this value.
- Processingret.The pipeline must stall until their instruction reaches the

write-back stage.

- Mispredictedbranches.By the time the branch logic detects that a jump

  should not have been taken, several instructions at the branch target will have started down the pipeline. These instructions must be canceled, and fetching should begin at the instruction following the jump instruction.

- Exceptions. When an instruction causes an exception, we want to disable the

  updating of the programmer-visible state by later instructions and halt execution once the excepting instruction reaches the write-back stage.

  Below is a diagram of the above conditions.

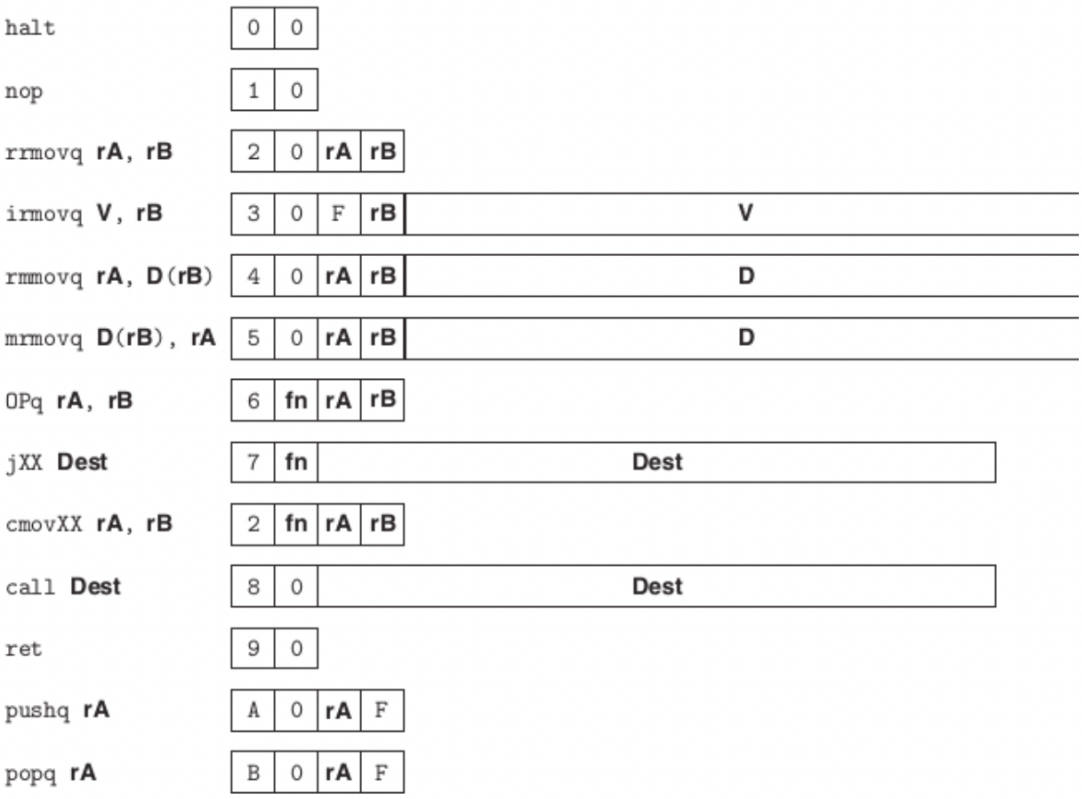
	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

```
if(W_stat == 4'd2 ||  W_stat == 4'd1 || W_stat == 4'd3) // errors
begin
    M_bubble <= 1'b1;
end
else if(m_stat == 4'd2 ||  m_stat == 4'd1 || m_stat == 4'd3) // errors
begin
    M_bubble <= 1'b1;
end
else if(D_icode == 4'h9 || E_icode == 4'h9 || M_icode == 4'h9) // return statements
begin
    F_stall <= 1'b1;
    D_bubble <= 1'b1;
end
else if(E_icode == 4'h5 || E_icode == 4'hB)
begin
    if(E_dstM == d_srcA || E_dstM == d_srcB)
    begin
    F_stall <= 1'b1;
    D_stall <= 1'b1;
    E_bubble <= 1'b1;
    end
end
else if( E_icode == 4'h7 && !e_Cnd )
begin
    D_bubble <= 1'b1;
    E_bubble <= 1'b1;
end
```

Set of Instructions supported by this processor

- rrmovq
- irmovq
- rmmovq
- mrmovq
- Opq
- jXX
- cmovXX
- pushq
- popq
- halt
- nop
- call
- ret

All the instructions in the Y86 instruction set are supported in the processor. Each of the stages has been tested on an individual testbench before integrating all the modules together into one main module namely ‘test_bench.v’.
The figure below shows the OPCODES of each of these instructions.



Registors Used

rax : 0

rdx : 1

rcx : 2

rbx : 3

rsp : 4

rbp : 5

rsi : 6

rdi : 7

r8 : 8

r9 : 9

r10 : 10

r11 : 11

r12 : 12

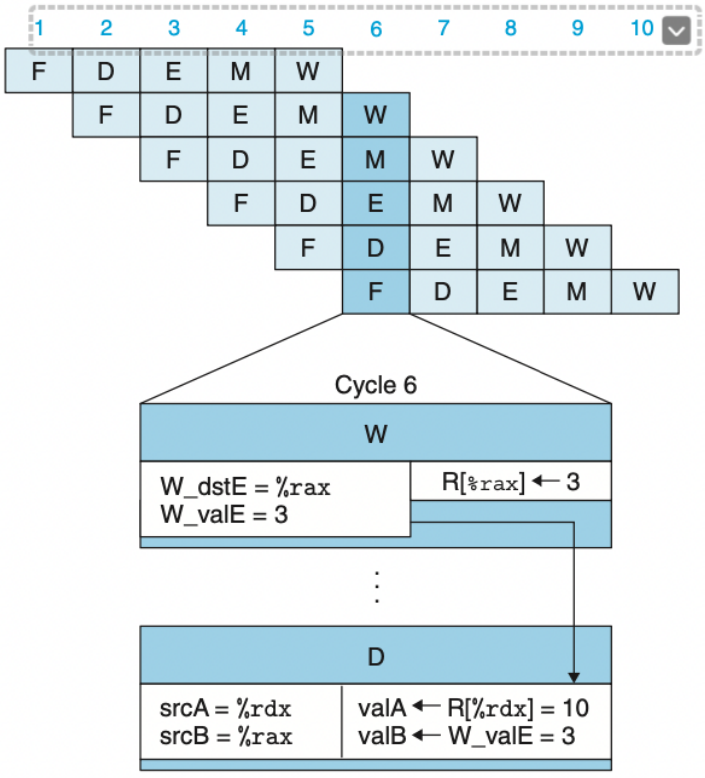
r13 : 13

r14 : 14

Forwarding

Our design for PIPE− reads source operands from the register file in the decode stage, but there can also be a pending write to one of these source registers in the write-back stage. Rather than stalling until the write has been completed, it can simply pass the value that is about to be written to pipeline register E as the source operand.

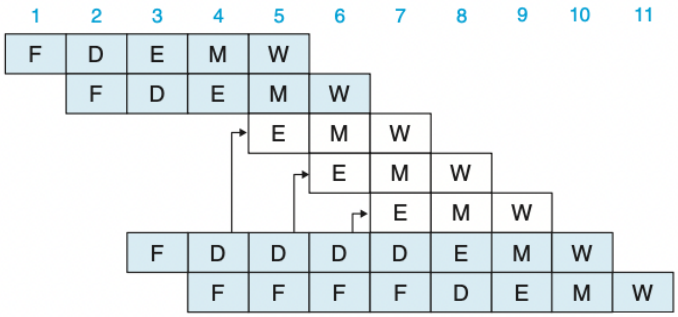
```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



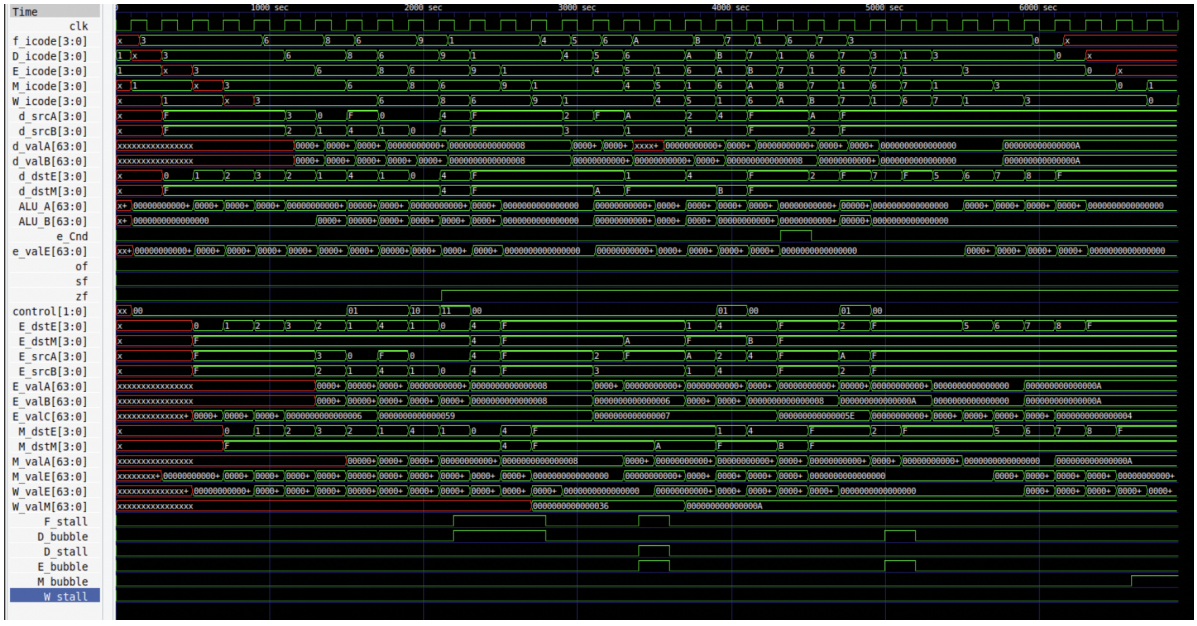
STALLING

One very general technique for avoiding hazards involves *stalling*, where the processor holds back one or more instructions in the pipeline until the hazard condition no longer holds. Our processor can avoid data hazards by holding back an instruction in the decode stage until the instructions generating its source operands have passed through the write-back stage

```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
       bubble
       bubble
       bubble
0x014: addq %rdx,%rax
0x016: halt
```



GTKWAVE



In the above gtkwave plot we can see that the instructions are passing from one stage to next one very smoothly in a synchronised manner.

—> We haven’t assigned any initial values to baby register hence for few initial values some of them are showing red boxes which mean don't care conditions (x).

HOW to RUN

1) Download the zip folder from the github.

2) Extract all the files

3) Open terminal in this folder.

4) Type “iverilog -o test_tb.vpp fetch.v decode.v memory.v execute.v write_back.v pipe_control_logic.v test_bench.v”

5) Type “vvp test_tb.vvp”

6) This would print all the outputs to terminal .

7) Now to open gtkwave type “gtkwave” in terminal to open gtkwave window.

8) After this open test_bench.vcd in the gtkwave terminal .

9) After that plot your desired variables.