INTRO TO PROCESSOR ARCHITECTURE

COURSE PROJECT

IMPLEMENTING THE Y-86 PROCESSOR DESIGN

DONE BY:

DEEPTANSH SHARMA (2021102011)

PRATHAM MISHRA (2021102036)

GUIDED BY:

DEEPAK GANGADHARAN SIR (PROFESSOR)

AKSHIT GUREJA SIR (TEACHING ASSISTANT)

The Y86 implementation in the course goes in two ways,

The sequential implementation.

The pipelined implementation.

Sequential Y86:

In a sequential Y86, each instruction is executed one after the other in a sequential manner. That is, an instruction is fetched from memory, decoded, executed, and then the result is stored in the memory or a register. Only after the execution of one instruction is completed, the next instruction is fetched and processed. The sequential implementation of Y86 is simple to design and understand, but it has a disadvantage in terms of performance. The sequential implementation is slower because the execution of each instruction must wait until the previous instruction is completely executed**.**

Pipelined Y86:

Pipelined Y86 is designed to overcome the performance limitation of the sequential implementation. In a pipelined Y86, the processor is divided into multiple stages, and each stage is responsible for a particular operation, such as instruction fetch, instruction decode, execution, and write-back.

The instructions are processed concurrently, and each instruction is in a different stage of the pipeline at any given time. When an instruction is fetched, the next instruction can be decoded, and the instruction after that can be executed, and so on.

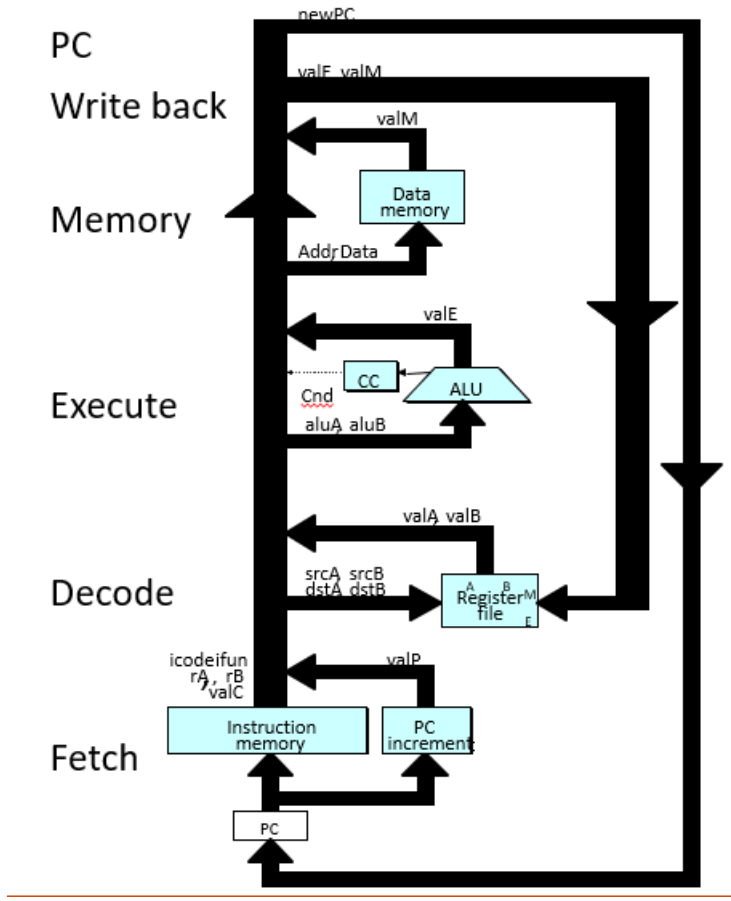
This overlapping of instructions in different stages of the pipeline enables a higher instruction throughput, which translates to higher performance. However, pipelining introduces additional complexity to the design and may cause problems such as pipeline stalls, where a stage is delayed because a preceding stage has not completed its work.

SEQUENTIAL IMPLEMENTATION

Each instruction sequentially goes through the following common stages:

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

The processor loops indefinitely, performing the functions in each stage unless any exception condition occurs.



The different blocks of the sequential implementation perform different functions, as mentioned below.

Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

- Read or write data

Write Back

- Write program registers

PC

- Update program counter

Fetch:

- Reads bytes of an instruction from memory using the PC value as address → Extracts the two 4-bit portions of instruction specifier byte referred to as **icode** and **ifun**
- Possibly fetches the register specifier byte giving one or both of the register operand specifiers rA and rB
- Also possibly fetches an 8-byte constant word valC → Computes valP as the address of the next instruction in the sequence , i.e. valP = PC + length of fetched instruction

```
else if (icode == IRMOVQ)
begin
rA = mem[PC + 1][7:4];
rB = mem[PC + 1][3:0];
valC = {mem[PC + 9], mem[PC + 8], mem[PC + 7], mem[PC + 6], mem[PC + 5], mem[PC + 4], mem[PC + 3], mem[PC + 2]};
valP = PC + 10;
end

else if (icode == RMMOVQ)
begin
rA = mem[PC + 1][7:4];
rB = mem[PC + 1][3:0];
valC = {mem[PC + 9], mem[PC + 8], mem[PC + 7], mem[PC + 6], mem[PC + 5], mem[PC + 4], mem[PC + 3], mem[PC + 2]};
valP = PC + 10;
end

else if (icode == MRMOVQ)
begin
rA = mem[PC + 1][7:4];
rB = mem[PC + 1][3:0];
valC = {mem[PC + 9], mem[PC + 8], mem[PC + 7], mem[PC + 6], mem[PC + 5], mem[PC + 4], mem[PC + 3], mem[PC + 2]};
valP = PC + 10;
end

else if (icode == OPQ)
begin
rA = mem[PC + 1][7:4];
rB = mem[PC + 1][3:0];
valP = PC + 2;
end

else if (icode == JXX)
begin
valC = {mem[PC + 8], mem[PC + 7], mem[PC + 6], mem[PC + 5], mem[PC + 4], mem[PC + 3], mem[PC + 2], mem[PC + 1]};
valP = PC + 9;
end
```

Here as you can see in our code, implemented what fetch does for all the various, taking out the various values such as rA, rB , valC, valP.

Fetch for Arithmetic/Logical operations

Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2

Fetch for rmmovq operation

Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10

Fetch for popq operation

Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2

Fetch for cmovxx operation

Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2

Fetch for conditional Jump operation

Fetch	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9

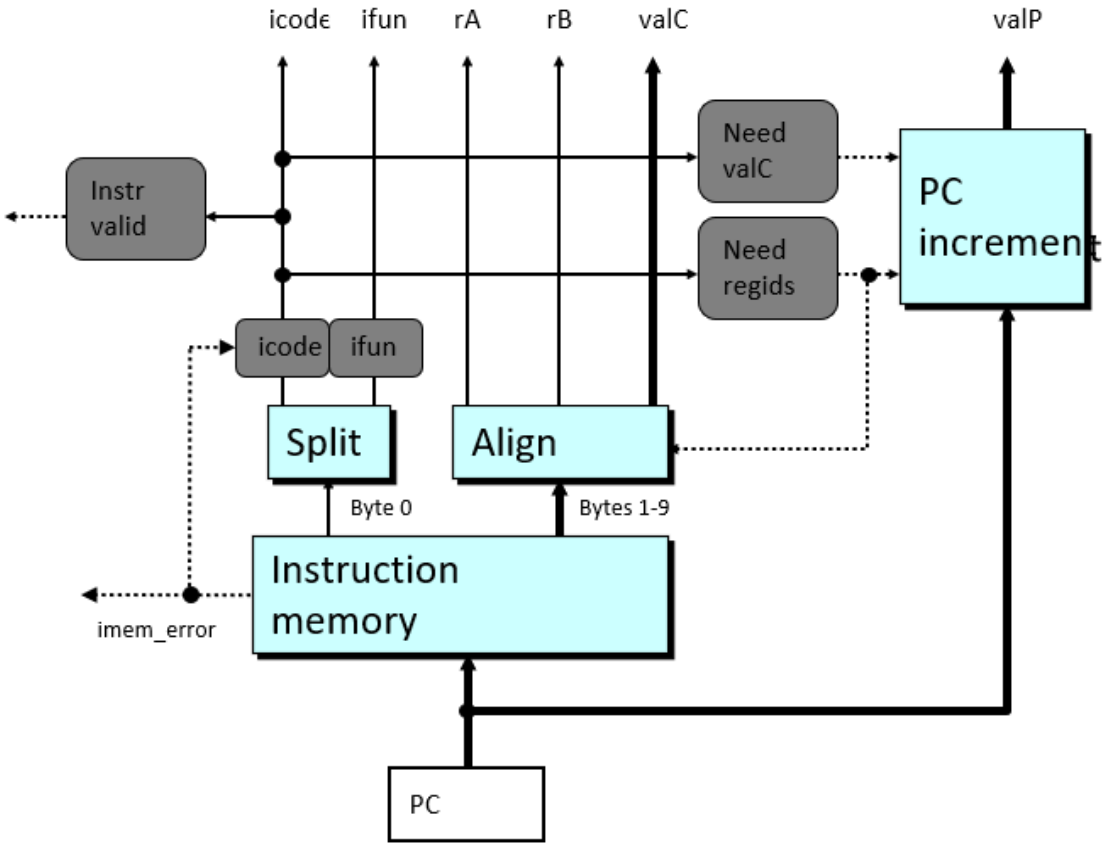
Fetch for call operation

Fetch	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9

Fetch for ret operation

Fetch	icode:ifun ← M_1[PC]

The logic/implementation of the fetch block is given by the below block diagram.



DECODE:

- Reads up to to two operands from the register file giving values valA and/or valB
- For some instructions, it reads register %rsp.
- We have implemented the writeback stage in the decode itself, for easy access to valE and the instruction memory.

```
always @(*) begin

    case(icode)

    CMOVXX:
    begin
    valA = rm[rA];
    end

    RMMOVQ:
    begin
    valA = rm[rA];
    valB = rm[rB];
    end

    MRMOVQ:
    begin
    valB = rm[rB];
    end

    OPQ:
    begin
    valA = rm[rA];
```

```
CALL:
begin
valB = rm[4];
end

RET:
begin
valA = rm[4];
valB = rm[4];
end

PUSHQ:
begin
valA = rm[rA];
valB = rm[4];
end

POPQ:
begin
valA = rm[4];
valB = rm[4];
end

endcase
```

The code for the decode is just assigning the valA and valB wherever necessary. The idea for implementing the following has been taken from the lectures itself.

The values of valA and valB for various instructions:

Arithmetic\Logic instructions:

Decode	valA ← R[rA] valB ← R[rB]

rmmovq instructions:

Decode	valA ← R[rA] valB ← R[rB]

popq instruction:

Decode	valA ← R[%rsp] valB ← R[%rsp]

cmovxx instructions:

Decode	valA ← R[rA] valB ← 0

Jxx instructions:

Decode	

In the jump instructions, we don’t do anything.

Call instruction:

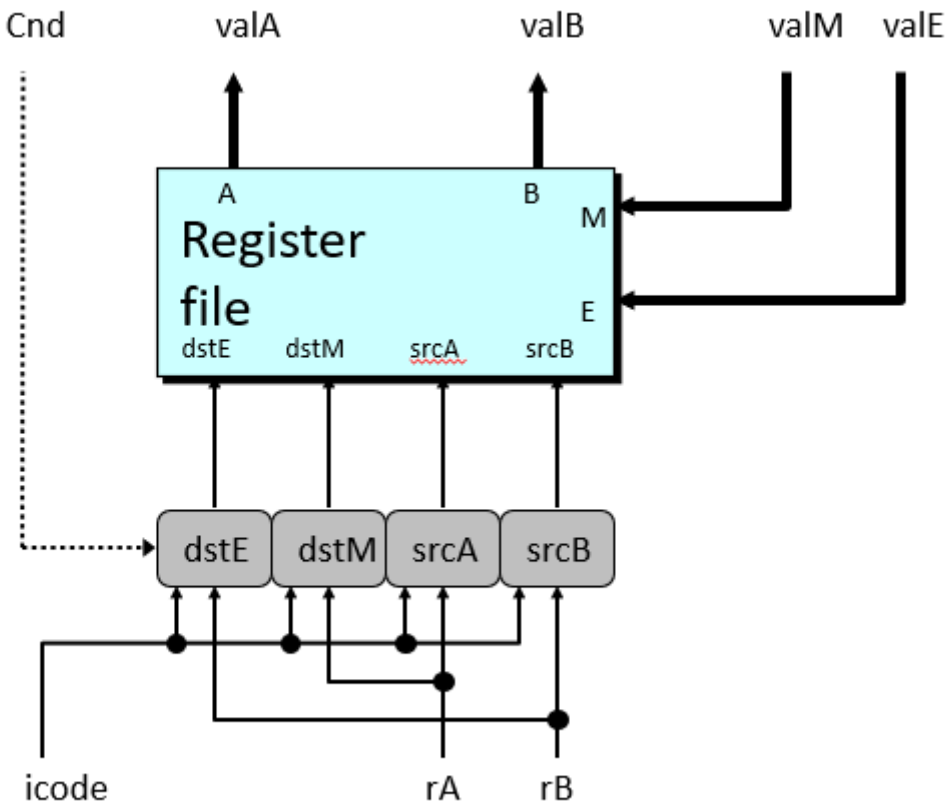
Decode	valB ← R[%rsp]

Ret instruction:

Decode	valA ← R[%rsp] valB ← R[%rsp]

We need to consider the cases where we have to use the %rsp, instead of the other registers, usually done when we use the stack pointers.

The logic used in executing the decode stage is given by the following block diagram:


Cnd
valA
valB
valM
valE
A
B
M
Register
file
E
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
icode
rA
rB

WriteBack:

- Writes up to two results to the register file.
- WriteBack stage is very important, as the values are assigned to the registers.
- The execution of this stage is not much complex though!

```
always @ (posedge clk)
    begin
        if(icode == CMOVXX && ifun == 4'b0)
        begin
            rm[rB] = valE;
        end
        if(icode == CMOVXX && ifun != 4'b0 && cnd == 1)
        begin
            // if(cnd)
            // begin
                rm[rB] = valE;
```

```
else if(icode == IRMOVQ)
begin
rm[rB] = valE;
end

else if(icode == MRMOVQ)
begin
rm[rA] = valF;
end

else if(icode == OPQ)
begin
rm[rB] = valE;
end

else if(icode == CALL)
begin
rm[4] = valE;
end

else if(icode == RET)
begin
rm[4] = valE;
end

else if(icode == PUSHQ)
begin
rm[4] = valE;
end

else if(icode == POPQ)
begin
rm[4] = valE;
rm[rA] = valF;
end
```

The code for the writeback just assigns the final values to the respective registers.

The idea for implementing the following stage has been inspired by the lectures themselves.

The writeback is been executed the following way:

Arithmetic\Logic instructions:

Write back	R[rB] ← valE

rmmovq instruction:

Write back	

We don’t do anything in writeback for rmmovq.

Popq instruction:

Write back	R[%rsp] ← valE R[rA] ← valM

Conditional move instructions:

Write back	R[rB] ← valE

Jump instructions:

Write back	

Call instruction:

Write back	R[%rsp] ← valE

Ret instruction:

Write back	R[%rsp] ← valE

Execute:

- ALU either performs operation given by ifun, computes effective address of a memory reference, or increments or decrements the stack pointer. Resulting value → valE
- Condition codes are possibly set
- For a jump instruction, tests condition code and branch condition (referred to by ifun) to determine if branch should be taken or not.

```
// DEFINING IFUN PARAMETERS

parameter FRRMOVQ = 4'b0000;
parameter FCMOVLE = 4'b0001;
parameter FCMOVL = 4'b0010;
parameter FCMOVE = 4'b0011;
parameter FCMOVNE = 4'b0100;
parameter FCMOVGE = 4'b0101;
parameter FCMOVG = 4'b0110;
parameter FOPADD = 4'b0000;
parameter FOPSUB = 4'b0001;
parameter FOPAND = 4'b0010;
parameter FOPXOR = 4'b0011;
parameter FJMP = 4'b0000;
parameter FJLE = 4'b0001;
parameter FJL = 4'b0010;
parameter FJE = 4'b0011;
parameter FJNE = 4'b0100;
parameter FJGE = 4'b0101;
parameter FJG = 4'b0110;

// DEFINING SELECT LINES FOR ALU OPERATIONS

parameter OPAND = 2'b00;
parameter OPXOR = 2'b01;
parameter OPADD = 2'b10;
parameter OPSUB = 2'b11;
```

We defined the values for all the Ifun and the select lines for the ALU, this is very necessary to simplify the execute stage.

```
initial begin
    z = 0;
    s = 0;
    o = 0;
end

// select, p, q, r, ofw

reg [2:0] select;
reg signed [63:0] p;
reg signed [63:0] q;
wire signed [63:0] r;
wire ofw;

alu inst1(
    .select(select),
    .p(p),
    .q(q),
    .r(r),
    .ofw(ofw)
);

reg signed A1, A2, O1, O2, X1, X2, N1;
wire signed AO, OO, XO, NO;

and gate1(AO, A1, A2);
or gate2(OO, O1, O2);
xor gate3(XO, X1, X2);
not gate4(NO, N1);
```

We initialize the flags to 0, and create select, p, q and r.

Here p and q are inputs, and r is the output.

Now we pass these created parameters into our ALU that we created in our 1st assignment.

The logic of calculating and assigning the values to valE is been inspired from the lectures itself.

The Execute is been executed the following way:

Arithmetic\Logic instructions:

Execute	valE ← valB OP valA Set CC

Rmmovq instruction:

Execute	valE ← valB + valC

Popq instruction:

Execute	valE ← valB + 8

Conditional move instructions:

Execute	valE ← valB + valA If ! Cond(CC,ifun) rB ← 0xF

Jump instructions:

Execute	Cnd ← Cond(CC,ifun)

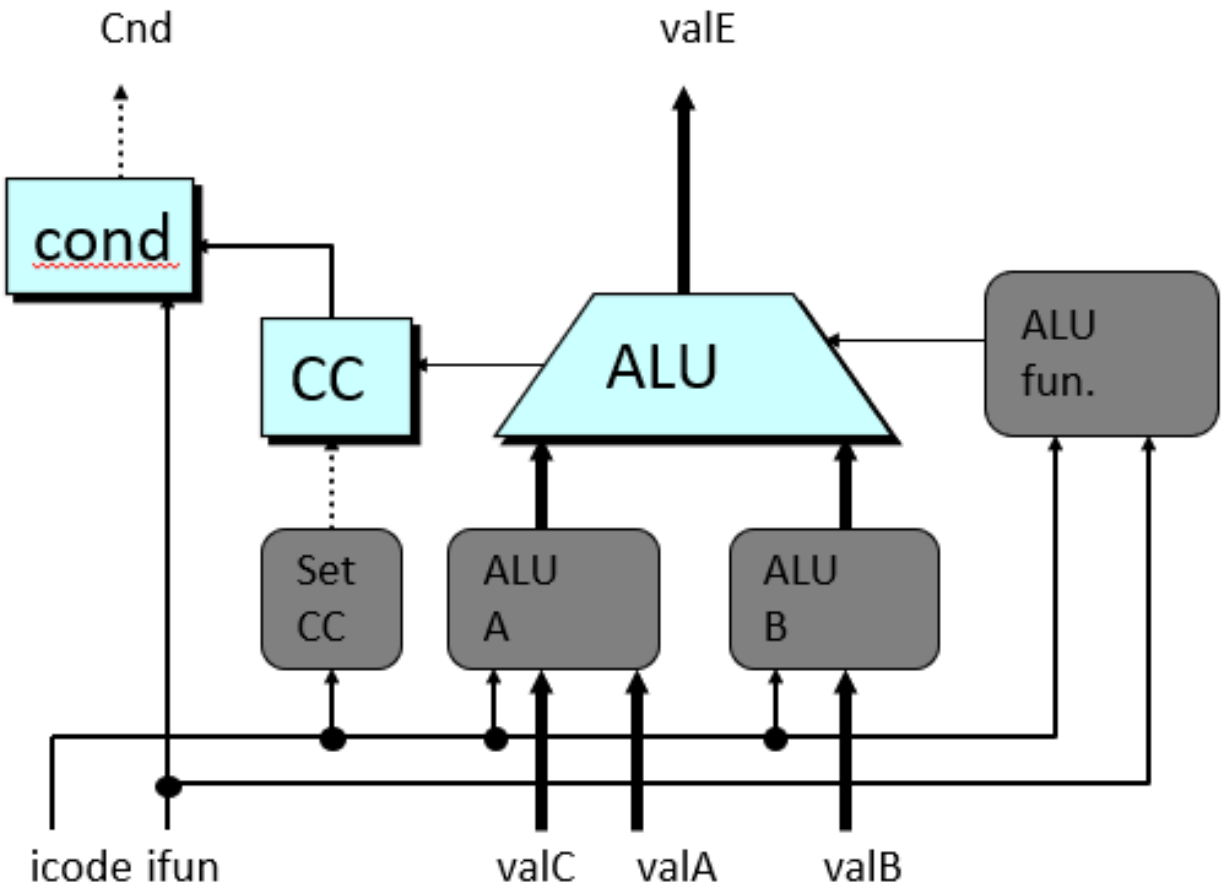
Call instruction:

Execute	valE ← valB + –8

Ret instruction:

Execute	valE ← valB + 8

The Execute is made with the following block diagram:



MEMORY:

Memory

- Reads or writes memory word
- May read or write data from/to memory respectively. Value read referred to as valM.

Control Logic

- stat: What is instruction status?
- Mem. read: should the word be read?
- Mem. write: should the word be written?
- Mem. addr.: Select the address.
- Mem. data.: Select data.

```
case(icode)

RMMOVQ:
begin
    mem[valE] = valA;
end

MRMOVQ:
begin
    valF = mem[valE];
end

CALL:
begin
    mem[valE] = valP;
end

RET:
begin
    valF = mem[valA];
end

PUSHQ:
begin
mem[valE] = valA;
end

POPQ:
begin
valF = mem[valE];
end

endcase

curr_mem = mem[valE];
```

We put different values in memory based on the different instructions.

The idea and implementation of various values of memory is studied from the class lectures.

The Memory is been executed the following way:

Arithmetic\Logic instructions:

Memory	

Rmmovq instruction:

Memory	$M_8[valE] \leftarrow valA$

Popq instruction:

Memory	$valM \leftarrow M_8[valA]$

Conditional move instructions:

Memory	

Jump instructions:

Memory	

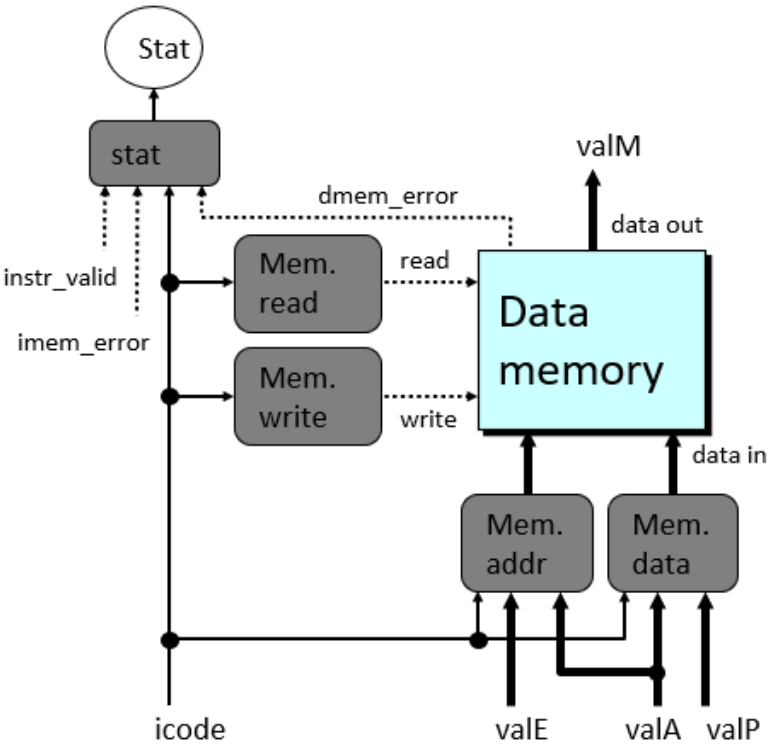
Call instruction:

Memory	$M_8[valE] \leftarrow valP$

Ret instruction:

Memory	$valM \leftarrow M_8[valA]$

The Memory is made with the following block diagram:



PC UPDATE:

PC is set to address of next instruction or valP.

New PC

- Select next value of PC

```
always @(*)
begin

    case(icode)

    JXX:
    begin
        if(cnd == 1)
        begin
            nPC = valC;
        end
        else
        begin
            nPC = valP;
        end
    end

    CALL:
    begin
        nPC = valC;
    end

    RET:
    begin
        nPC = valF;
    end

    default:
    begin
        nPC = valP;
    end

    endcase

end
```

Here as you can see in this code, the new pc is set according to the various instructions.

This was implemented through what was taught to us in the lectures.

The PC Update is been executed the following way:

Arithmetic\Logic instructions:

PC update	PC ← valP

Rmmovq instruction:

PC update	PC ← valP

Popq instruction:

PC update	PC ← valP

Conditional move instructions:

PC update	PC ← valP

Jump instructions:

PC update	PC ← Cnd ? valC : valP

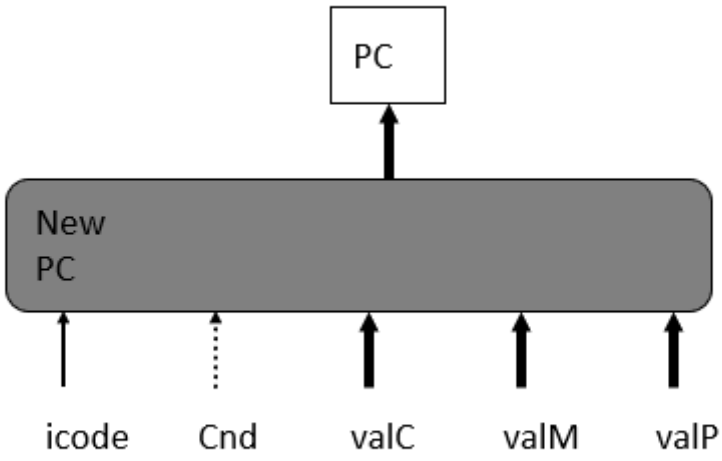
Call instruction:

PC update	PC ← valC

Ret instruction:

PC update	PC ← valM

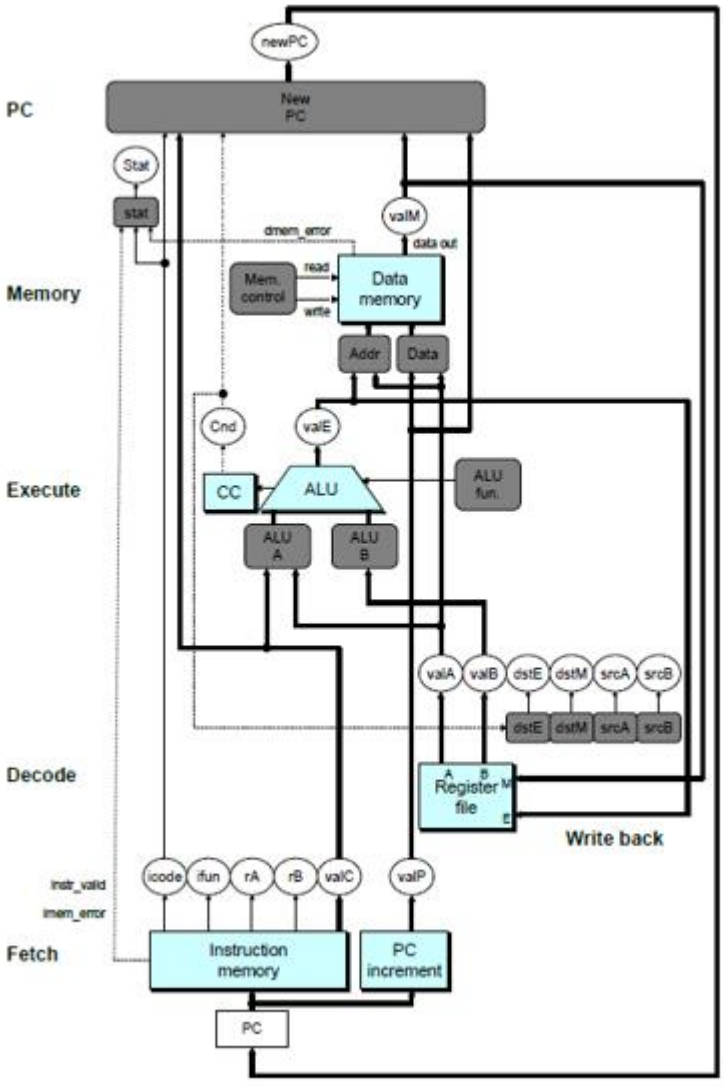
The Memory is made with the following block diagram:



SEQUENTIAL Y86:

In the above parts of the report, we showed you how we managed to make the different stages of the Y86 sequential. Now we will have to combine all these stages so that we can create a working sequential Y86.

The following block diagram explains the combination:



```
`timescale 1ns / 10ps

`include "fetch.v"
`include "decode.v"
`include "execute.v"
`include "memory.v"
`include "pc_update.v"

module processor();

reg clk;
reg [63:0] PC;

wire [3:0]icode, ifun, rA, rB;

wire [64:1] valA, valB, valC, valP, valE, valF;

wire [63:0] rm0, rm1, rm2, rm3, rm4, rm5, rm6, rm7, rm8, rm9, rm10, rm11, rm12, rm13, rm14;

wire cnd, z, s, o;

wire [63:0] curr_mem;
wire [63:0] nPC;
```

Firstly we include all the stages which we created earlier, and then declare all the necessary registers and wires that we require for each of the stages to function.

```
fetch fetch_insta(
    .clk(clk),
    .PC(PC),
    .icode(icode),
    .ifun(ifun),
    .rA(rA),
    .rB(rB),
    .valC(valC),
    .valP(valP)
    );
```

```
decode decode_instb(
    .clk(clk),
    .cnd(cnd),
    .icode(icode),
    .ifun(ifun),
    .rA(rA),
    .rB(rB),
    .valE(valE),
    .valF(valF),
    .valA(valA),
    .valB(valB),
    .rm0(rm0),
    .rm1(rm1),
    .rm2(rm2),
    .rm3(rm3),
    .rm4(rm4),
    .rm5(rm5),
    .rm6(rm6),
    .rm7(rm7),
    .rm8(rm8),
    .rm9(rm9),
    .rm10(rm10),
    .rm11(rm11),
    .rm12(rm12),
    .rm13(rm13),
    .rm14(rm14)
);
```

```
execute execute_instc(
    .clk(clk),
    .icode(icode),
    .ifun(ifun),
    .valA(valA),
    .valB(valB),
    .valC(valC),
    .valE(valE),
    .cnd(cnd),
    .z(z),
    .s(s),
    .o(o)
);
```

```
memory memory_instd(
    .clk(clk),
    .icode(icode),
    .ifun(ifun),
    .valA(valA),
    .valB(valB),
    .valE(valE),
    .valP(valP),
    .valF(valF),
    .curr_mem(curr_mem)
);
```

```
pc_update pc_update_inste(
    .clk(clk),
    .cnd(cnd),
    .icode(icode),
    .ifun(ifun),
    .valC(valC),
    .valP(valP),
    .valF(valF),
    .PC(PC),
    .nPC(nPC)
);
```

Now we simply call all the stages that we included above and pass all the required parameters in them.

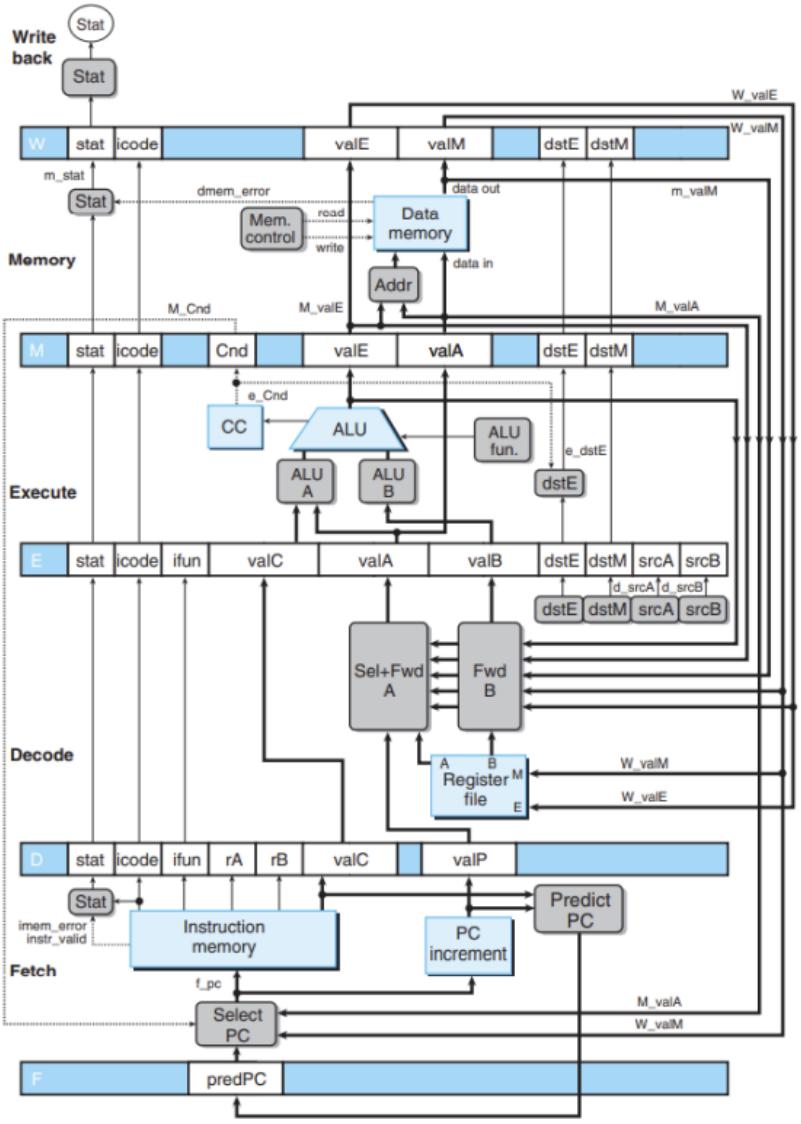
OUTPUT (SEQ)

```
clk = 0 PC =                  12 nPC =                  12 icode =  3
 ifun =  0 rA = 15 rB =  3 valC =                 203
 valP =                  22 valE =                  203 valF =                  x valA =                  0
 valB =                   0 cnd = 0 z = 0 o = 0
 s = 0 curr_mem =                  x rm3 =                  3 rm2 =                170 rm4 =                  4
 rm14 =                  14


clk = 1 PC =                  12 nPC =                  22 icode =  3
 ifun =  0 rA = 15 rB =  3 valC =                 203
 valP =                  22 valE =                  203 valF =                  x valA =                  0
 valB =                   0 cnd = 0 z = 0 o = 0
 s = 0 curr_mem =                123 rm3 =                  3 rm2 =                170 rm4 =                  4
 rm14 =                  14


clk = 0 PC =                  22 nPC =                  22 icode =  2
 ifun =  0 rA =  3 rB =  4 valC =                 203
 valP =                  24 valE =                  203 valF =                  x valA =                203
 valB =                   0 cnd = 1 z = 0 o = 0
 s = 0 curr_mem =                123 rm3 =                203 rm2 =                170 rm4 =                  4
 rm14 =                  14
```

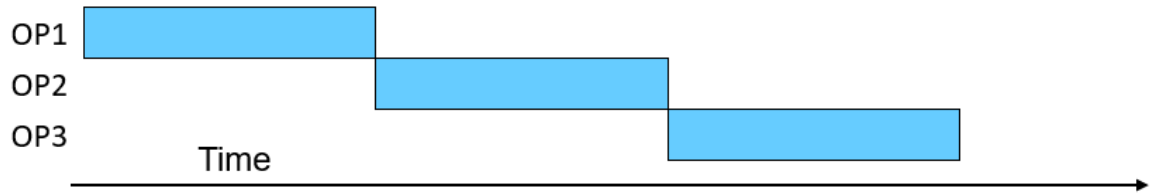
```
clk = 0 PC =                  39 nPC =                 117 icode =  5
 ifun =  0 rA = 14 rB =  3 valC =                   0
 valP =                  49 valE =                 203 valF =                  x valA =                198
 valB =                 203 cnd = 0 z = 0 o = 0
 s = 1 curr_mem =                  x rm3 =                203 rm2 =                170 rm4 =                373
 rm14 =                  14


clk = 1 PC =                  39 nPC =                  49 icode =  5
 ifun =  0 rA = 14 rB =  3 valC =                   0
 valP =                  49 valE =                 203 valF =                123 valA =                198
 valB =                 203 cnd = 0 z = 0 o = 0
 s = 1 curr_mem =                123 rm3 =                203 rm2 =                170 rm4 =                373
 rm14 =                  14


clk = 0 PC =                  49 nPC =                  49 icode =  8
 ifun =  2 rA = 14 rB =  3 valC =                 112
 valP =                  58 valE =                 365 valF =                123 valA =                198
 valB =                 373 cnd = 0 z = 0 o = 0
 s = 1 curr_mem =                123 rm3 =                203 rm2 =                170 rm4 =                373
 rm14 =                 123
```

PIPELINED IMPLEMENTATION

The pipelined is implemented by adding the pipelined registers in between the various stages of the processor. So now the instructions don't have to wait for the earlier instruction to finish. Thus saving us a lot of time that was wasted in the sequential implementation.

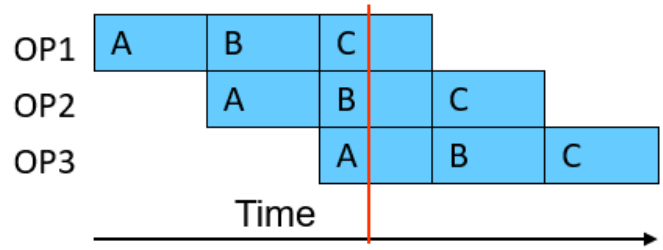


UNPIPELINED:



As we can see that the next instruction only starts only once the previous one is finished.

PIPELINED:



Here we can see the next instruction starts without waiting for the previous one to finish. Thus taking less time to execute and giving us an improved throughput.

We now need registers for each stage to store the required data by each of the stages.

FETCH:

```
module freg(
    input clk,
    input [63:0] prPC,
    output reg [63:0] FpredPC
);

always @(posedge clk)
begin
    FpredPC <= prPC;
end

endmodule
```

This is the fetch register is present before the Fetch stage and as you can see it stores our predicted PC.

DECODE:

```
module dreg(
    input clk,
    input [3:0] ficode,
    input [3:0] fifun,
    input [3:0] frA,
    input [3:0] frB,
    input [63:0] fvalC,
    input [63:0] fvalP,

    output reg [3:0] Dicode,
    output reg [3:0] Difun,
    output reg [3:0] DrA,
    output reg [3:0] DrB,

    output reg [63:0] DvalC,
    output reg [63:0] DvalP
);

always @(posedge clk)
begin
    Dicode <= ficode;
    Difun <= fifun;
    DvalC <= fvalC;
    DvalP <= fvalP;
    DrA <= frA;
    DrB <= frB;
end

endmodule
```

The Decode register is between the fetch and the decode, and the outputs from the last fetch statement are updated in it.

EXECUTE:

```
module ereg(
    input clk,
    input [3:0] dicode,
    input [3:0] difun,
    input [3:0] ddstE,
    input [3:0] ddstM,
    input [3:0] dsrcA,
    input [3:0] dsrcB,

    input [63:0] dvalA,
    input [63:0] dvalB,
    input [63:0] dvalC,

    output reg [3:0] Eicode,
    output reg [3:0] Eifun,
    output reg [3:0] EdstM,
    output reg [3:0] EdstE,
    output reg [3:0] EsrcA,
    output reg [3:0] EsrcB,

    output reg [63:0] EvalA,
    output reg [63:0] EvalB,
    output reg [63:0] EvalC
);

always @(posedge clk)
begin
    Eicode <= dicode;
    Eifun <= difun;
    EdstM <= ddstM;
    EdstE <= ddstE;
    EsrcA <= dsrcA;
    EsrcB <= dsrcB;
    EvalA <= dvalA;
    EvalB <= dvalB;
    EvalC <= dvalC;
end
```

The Execute register is placed between the Decode and the Execute stages.

MEMORY:

```
module mreg(
    input clk,
    input ecnd,

    input [3:0] eicode,
    input [3:0] eifun,
    input [3:0] edstM,
    input [3:0] edstE,

    input [63:0] evalE,
    input [63:0] evalA,

    output reg [3:0] Micode,
    output reg [3:0] MdstE,
    output reg [3:0] MdstM,

    output reg [63:0] MvalA,
    output reg [63:0] MvalE,

    output reg Mcnd
);

always @(posedge clk)
    begin
        Micode <= eicode;
        Mifun <= eifun;
        Mcnd <= ecnd;
        MvalA <= evalA;
        MdstE <= edstE;
        MdstM <= edstM;
        MvalE <= evalE;
    end
endmodule
```

The memory register is in between the Execute and Memory stages.

WRITEBACK:

```
module wbreg(
    input clk,
    input [3:0] micode,
    input [3:0] mdstM,
    input [3:0] mdstE,
    input [63:0] mvalE,
    input [63:0] mvalM,

    output reg [3:0] Wicode,
    output reg [3:0] Wifun,
    output reg [3:0] WdstE,
    output reg [3:0] WdstM,

    output reg [63:0] WvalE,
    output reg [63:0] WvalM,

    output reg mcnd
);

always @(posedge clk)
    begin
        Wicode <= micode;
        Wifun <= mifun;
        WvalM <= mvalM;
        WdstE <= mdstE;
        WdstM <= mdstM;
        WvalE <= mvalE;
    end
endmodule
```

The writeback register is present between the memory and writeback stages.

PIPELINE STAGES:

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

- Operate ALU

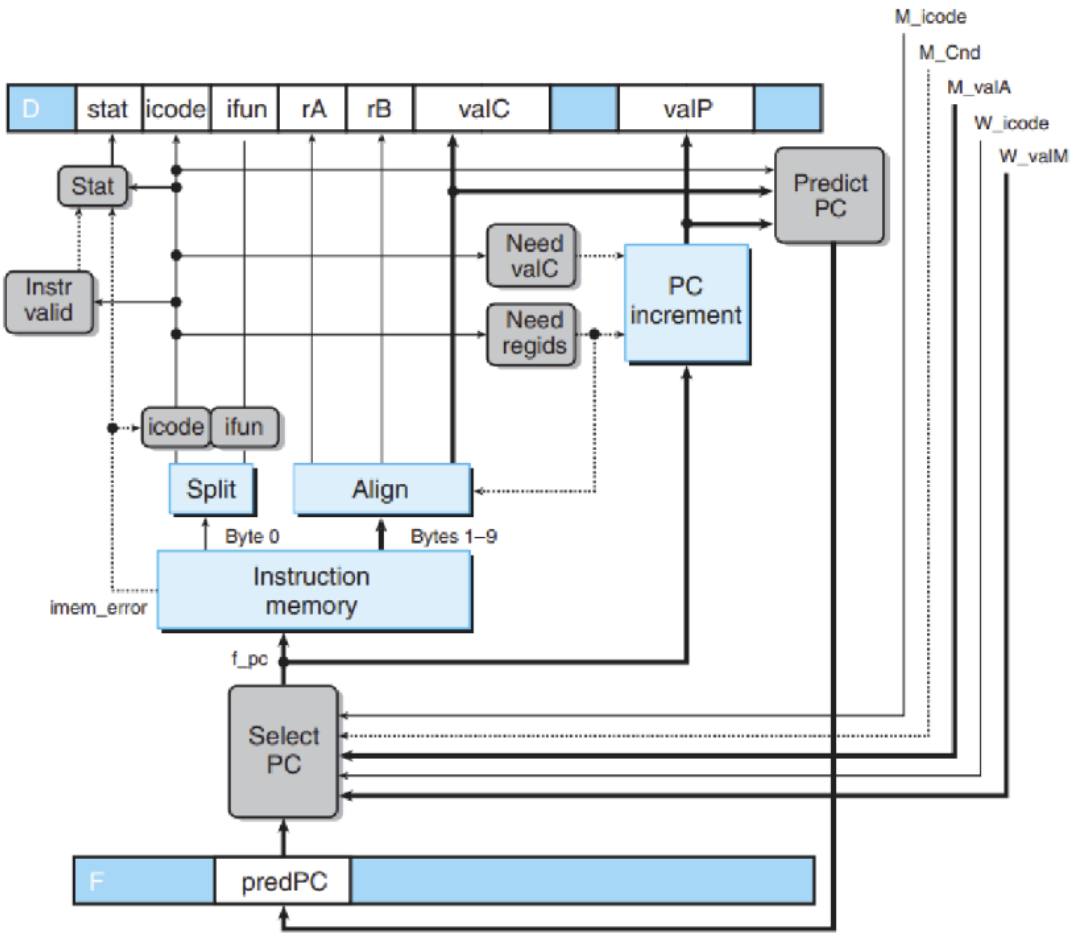
Memory

- Read or write data memory

Write Back

- Update register file

FETCH STAGE:



In the fetch stage we send the predicted PC value to the select PC and using the inputs from the previous instructions, if the predicted PC is correct then the instruction enters the fetch stage.

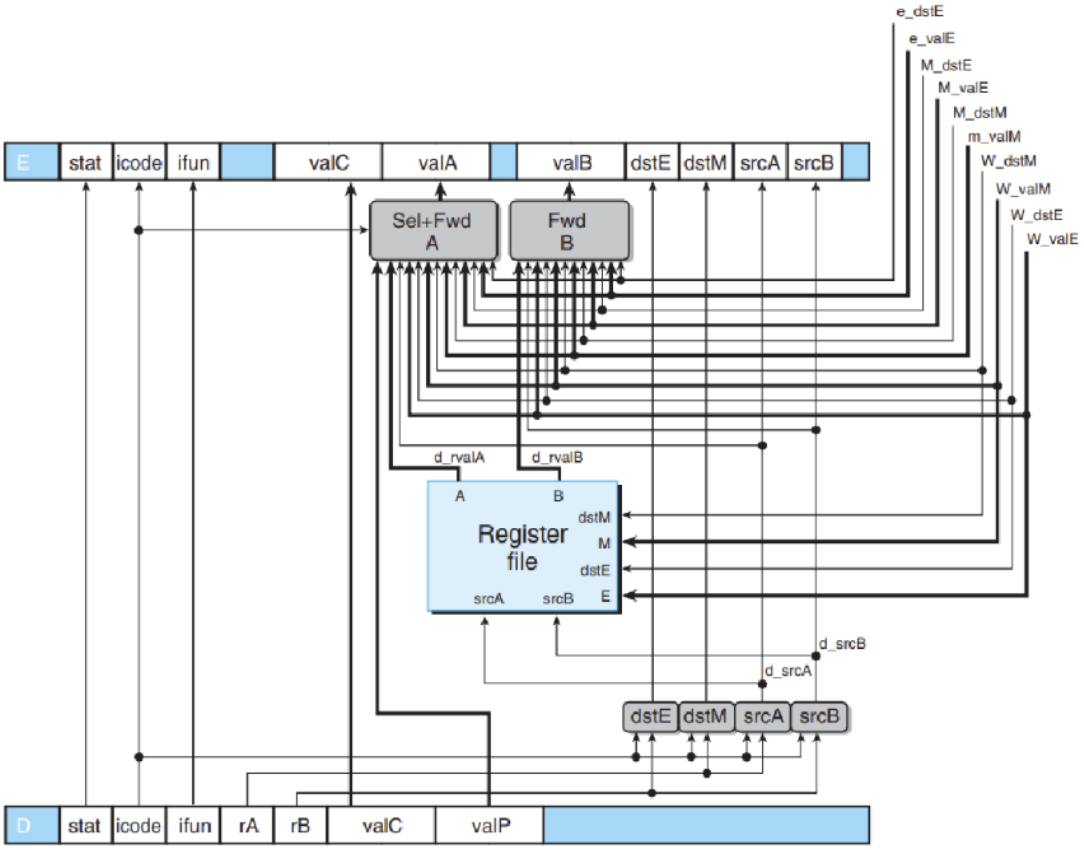
The output of the Fetch stage is then stored in the Decode register.

```
// PC Prediction (Separated in a callback module within fetch)

module predict_pc(input [63:0]fvalP,
                  input [63:0]fvalC,
                  input [63:0]fPC,
                  input [63:0]ficode,
                  output reg [63:0]prPC);

always @(*)
begin
    if(ficode == 4'b0111 || ficode == 4'b1000)
    begin
        prPC = fvalC;
    end
    else
    begin
        prPC = fvalP;
    end
end
endmodule
```

The code shown in the side is how we predict the PC value, this is written as a part of the fetch stage.

The remaining part of the module is the same as that of the Fetch module in the sequential Y86.

DECODE STAGE:



The above block diagram shows the Decode stage, where the data from the Fetch will be taken from the decode register, and the data will be decoded and then sent to the execute register.

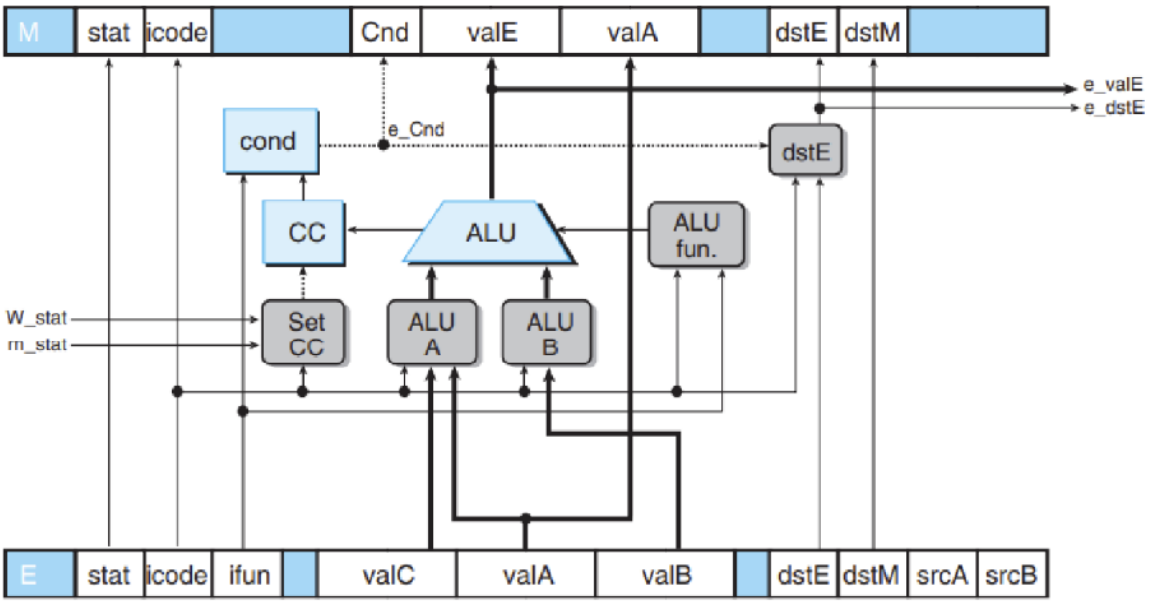
We also implement the forwarding in the Decode stage.

Forwarding:

```
always@(*)
begin
    if(Dicode == JXX || Dicode == CALL)
        dvalA = DvalP;
    else if(dsrcA == edstM)
        dvalA = evalE;
    else if(dsrcA == MdstM)
        dvalA = mvalM;
    else if(dsrcA == MdstE)
        dvalA = MvalE;
    else if(dsrcA == WdstM)
        dvalA = WvalM;
    else if(dsrcA == WdstE)
        dvalA = WvalE;
    else
        dvalA = rvalA;


    if(dsrcB == edstE)
        dvalB = evalE;
    else if(dsrcB == MdstM)
        dvalB = mvalM;
    else if(dsrcB == MdstE)
        dvalB = MvalE;
    else if(dsrcB == WdstM)
        dvalB = WvalM;
    else if(dsrcB == WdstE)
        dvalB = WvalE;
    else
        dvalB = rvalB;
end

always@(*)
begin
    dicode = Dicode;
    difun = Difun;
    dvalC = DvalC;
end
```

The code here shows Data forwarding. In a pipeline architecture, data forwarding is a technique used to improve performance by reducing stalls in the pipeline caused by dependencies between instructions. When an instruction in the pipeline requires data that is not yet available, it must wait until the data becomes available, causing a stall in the pipeline. Data forwarding solves this problem by allowing the data to be forwarded directly from the output of one instruction to the input of another instruction that requires it, without first storing the data in memory. This technique can reduce the number of stalls in the pipeline and improve performance. Data forwarding can be implemented using dedicated hardware, such as bypass circuits, that allows data to be forwarded between pipeline stages.

EXECUTE STAGE:



The execute contains the ALU, the ALU used here is the same as that we created during our assignment.

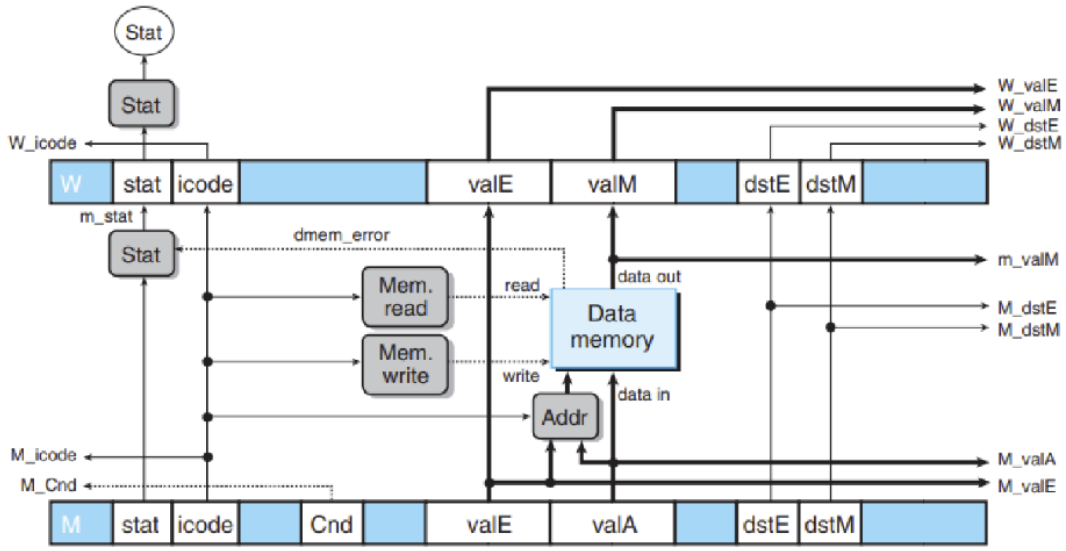
The inputs to the execute stage come from the execute register. After the execution, the results are sent to the Memory register.

We also set condition codes as the destE also depends on it.

```
always@(*)
begin
    if((Eicode == CMOVXX) && (!ecnd))
        edstE = 4'b1111;
    else
        edstE = EdstE;
end

always@(*)
begin
    eicode = Eicode;
    eifun = Eifun;
    evalA = EvalA;
    edstM = EdstM;
end
```

Here we can see how we set the values for edestE.

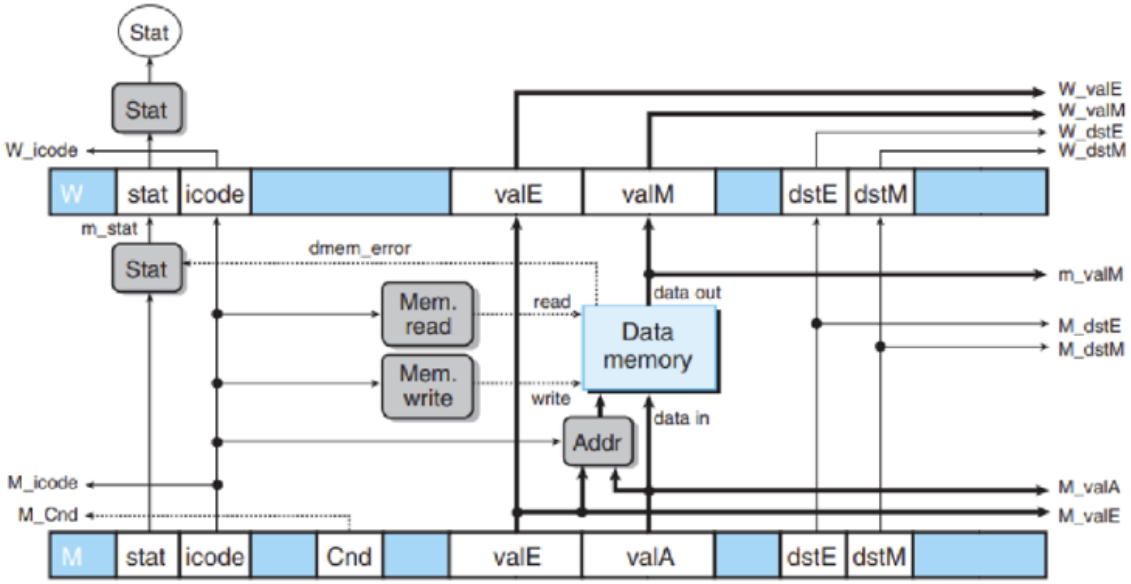
MEMORY STAGE:



Data is read from or written to the memory during the memory stage. From M to W, the required information is sent. Since it is defined to be independent from the instruction memory, the memory is not used during the other stages.

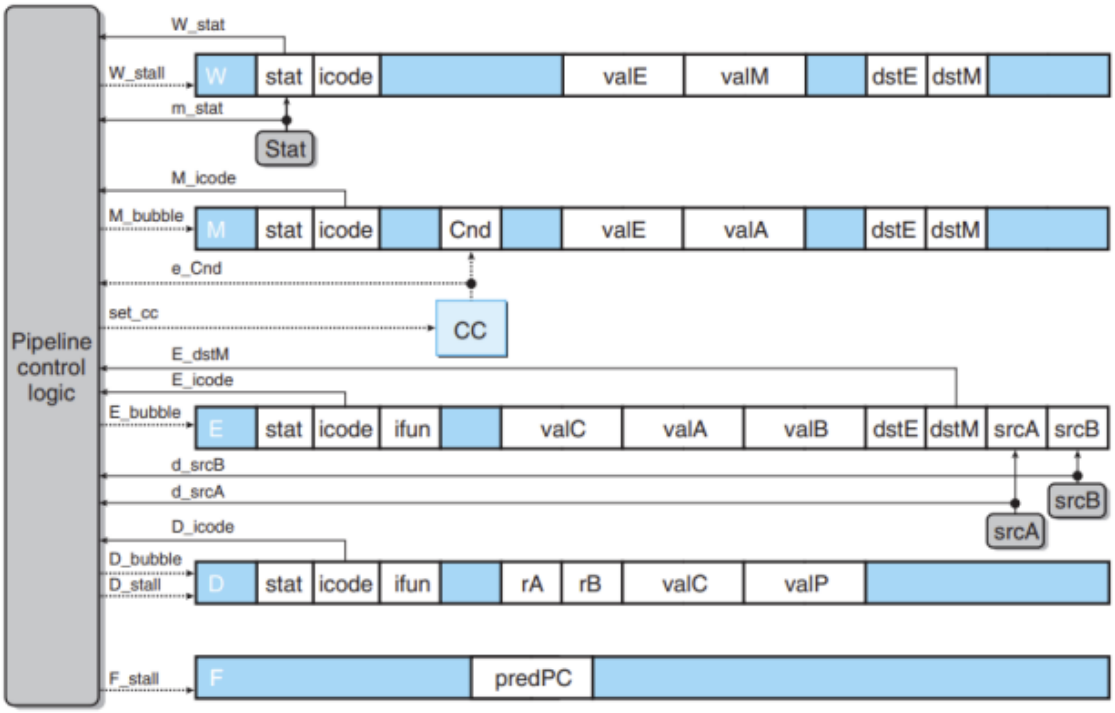
only accessible during this period. The enormous amount of signals transmitted from M, the stage itself, and W to the earlier instructions (for potential data issues of subsequent instructions to be addressed) is a remarkable aspect of this stage.

WRITEBACK STAGE:



We use the write back register(s) outputs to write up to two results into the register file during the write back stage (s). W_valE, W_valM are the corresponding results to be written into the registers, while W_dstE, and W_dstM indicate the registers that could be potentially written into.

CONTROL LOGIC:



There are several control scenarios that data forwarding and branch prediction cannot fully handle. The following cases are listed: -

1. Load/use hazards: The pipeline must stall for one cycle for every two instructions when the first reads a value from memory and the second uses that value.

2. Executing ret: The pipeline needs to be stopped until ret reaches write back while the ret instruction is being executed.

3. Incorrect branches: Whenever a leap that was not intended to occur happens, the pipeline must cancel all previous instructions and acquire the instruction immediately following the jump instruction.

4. Distinctions

```
always @(*)
begin
    Fb = 0;
    Fs = 0;
    Db = 0;
    Ds = 0;
    Eb = 0;
    Es = 0;

    if(Eicode == 4'b0111 & ecnd == 4'b0000)
    begin
        Db = 1;
        Eb = 1;
    end

    else if((Eicode == 4'b0101 | Eicode == 4'b1011) & (EdstM == dsrcA | EdstM == dsrcB))
    begin
        Fs = 1;
        Ds = 1;
        Eb = 1;
    end

    else if(Eicode == 4'b1001 | Micode == 4'b1001 | Dicode == 4'b1001)
    begin
        Fs = 1;
        Db = 1;
    end
end
```

OUTPUT

```
clk=1, fPC=12, ficode=3, fifun=0, frA=15, frB=3, fvalP=22, fvalC=203, Dicode=3, Eicode=3, Micode=3, Wicode=x, halt=0, r0=0, r1=1, r2=2, r3
r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=22 FpredPC=22WdstE=x WvalE=x
clk=0, fPC=12, ficode=3, fifun=0, frA=15, frB=3, fvalP=22, fvalC=203, Dicode=3, Eicode=3, Micode=3, Wicode=x, halt=0, r0=0, r1=1, r2=2, r3
r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=22 FpredPC=22WdstE=x WvalE=x
clk=1, fPC=22, ficode=2, fifun=0, frA=3, frB=4, fvalP=24, fvalC=203, Dicode=3, Eicode=3, Micode=3, Wicode=3, halt=0, r0=0, r1=1, r2=2, r3=
r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=24 FpredPC=22WdstE=2 WvalE=170
clk=0, fPC=22, ficode=2, fifun=0, frA=3, frB=4, fvalP=24, fvalC=203, Dicode=3, Eicode=3, Micode=3, Wicode=3, halt=0, r0=0, r1=1, r2=170,
r3=3, r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=24 FpredPC=22WdstE=2 WvalE=170
clk=1, fPC=22, ficode=2, fifun=0, frA=3, frB=4, fvalP=24, fvalC=203, Dicode=2, Eicode=3, Micode=3, Wicode=3, halt=0, r0=0, r1=1, r2=170,
r3=3, r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=24 FpredPC=24WdstE=2 WvalE=170
clk=0, fPC=22, ficode=2, fifun=0, frA=3, frB=4, fvalP=24, fvalC=203, Dicode=2, Eicode=3, Micode=3, Wicode=3, halt=0, r0=0, r1=1, r2=170,
r3=3, r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=24 FpredPC=24WdstE=2 WvalE=170
clk=1, fPC=24, ficode=6, fifun=0, frA=2, frB=4, fvalP=26, fvalC=203, Dicode=2, Eicode=2, Micode=3, Wicode=3, halt=0, r0=0, r1=1, r2=170,
r3=3, r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=26 FpredPC=24WdstE=3 WvalE=203
clk=0, fPC=24, ficode=6, fifun=0, frA=2, frB=4, fvalP=26, fvalC=203, Dicode=2, Eicode=2, Micode=3, Wicode=3, halt=0, r0=0, r1=1, r2=170,
r3=203, r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=26 FpredPC=24WdstE=3 WvalE=203
clk=1, fPC=24, ficode=6, fifun=0, frA=2, frB=4, fvalP=26, fvalC=203, Dicode=6, Eicode=2, Micode=2, Wicode=3, halt=0, r0=0, r1=1, r2=170,
r3=203, r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=26 FpredPC=26WdstE=3 WvalE=203
clk=0, fPC=24, ficode=6, fifun=0, frA=2, frB=4, fvalP=26, fvalC=203, Dicode=6, Eicode=2, Micode=2, Wicode=3, halt=0, r0=0, r1=1, r2=170,
r3=203, r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=26 FpredPC=26WdstE=3 WvalE=203
clk=1, fPC=26, ficode=6, fifun=1, frA=3, frB=5, fvalP=28, fvalC=203, Dicode=6, Eicode=6, Micode=2, Wicode=2, halt=0, r0=0, r1=1, r2=170,
r3=203, r4=4, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=28 FpredPC=26WdstE=4 WvalE=203
clk=0, fPC=26, ficode=6, fifun=1, frA=3, frB=5, fvalP=28, fvalC=203, Dicode=6, Eicode=6, Micode=2, Wicode=2, halt=0, r0=0, r1=1, r2=170,
r3=203, r4=203, r5=5, r6=6, r7=7, r8=8, r9=9, r10=10, r11=11, r12=12, r13=13, r14=14prPC=28 FpredPC=26WdstE=4 WvalE=203
```