IPA Project : Y-86 Instruction Set Architecture

Team Name - SIGMA
Team members - Pallav Subrahmanyam Koppisetti (2020102070) , KNV Karthikeya (2020102003)

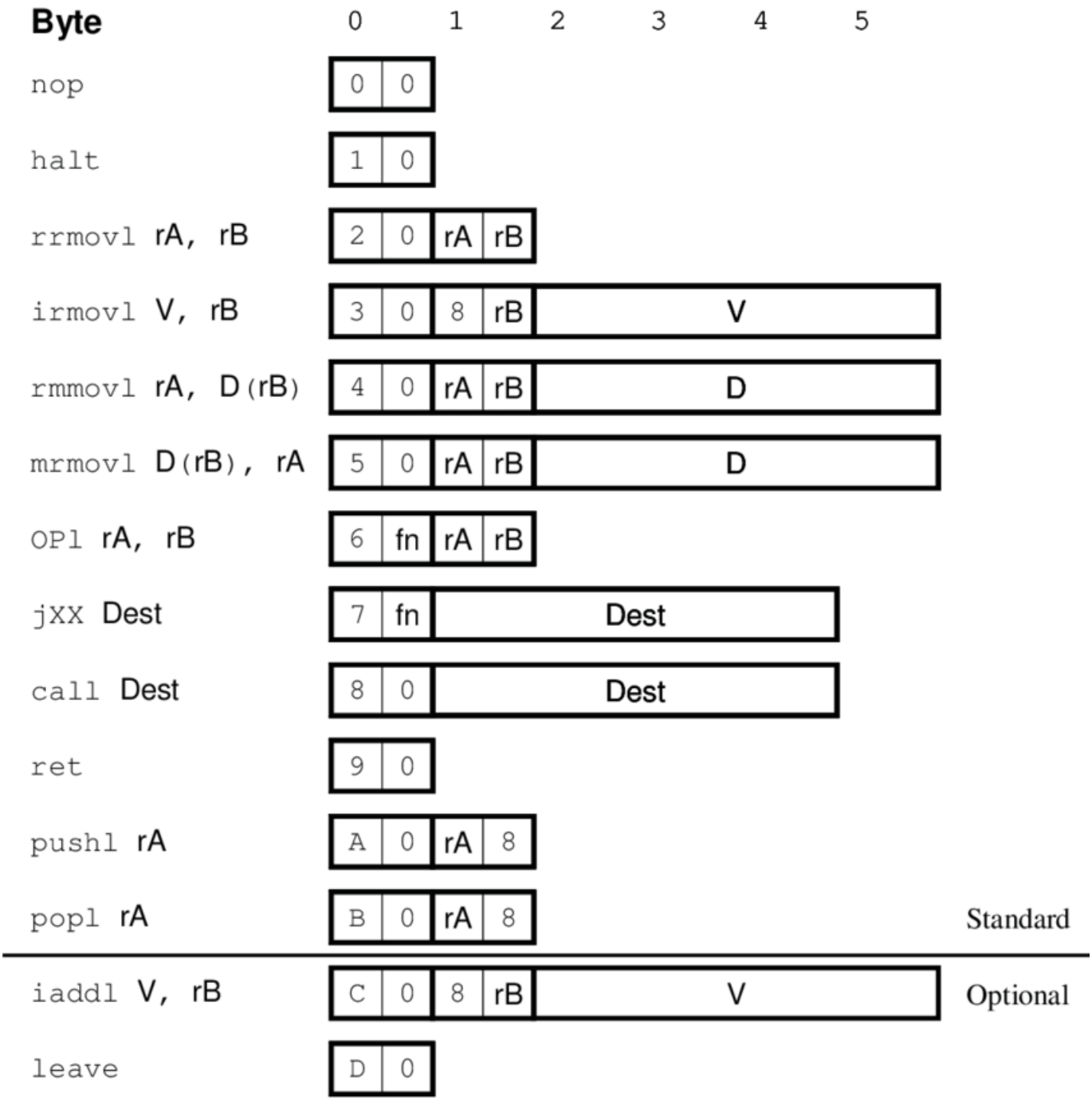
Final report :-

We have implemented a sequential y86 architecture processor using Verilog, which can be divided into 5 modules.

- Fetch
- Decode
- Execute
- Memory
- PC update/writeback

The commands supported by our processor are :-

- ihalt
- inop
- irrmovq
- iirmovq
- irmmovq
- imrmovq
- iopq
- ijxx
- icall
- iret
- ipushq
- ipop



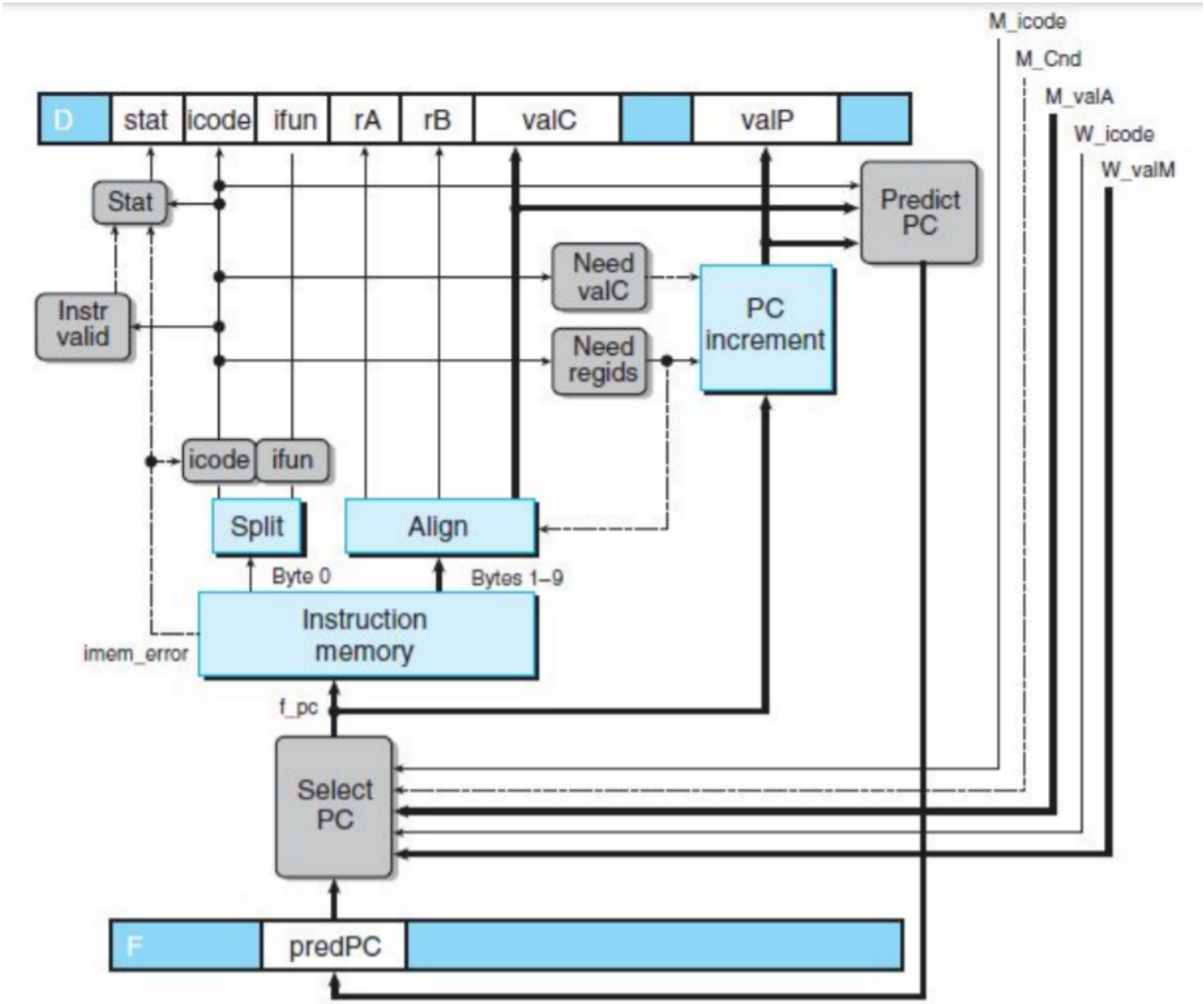
The above image depicts the Y86-64 Instruction set

Sequential Processor

Fetch :-

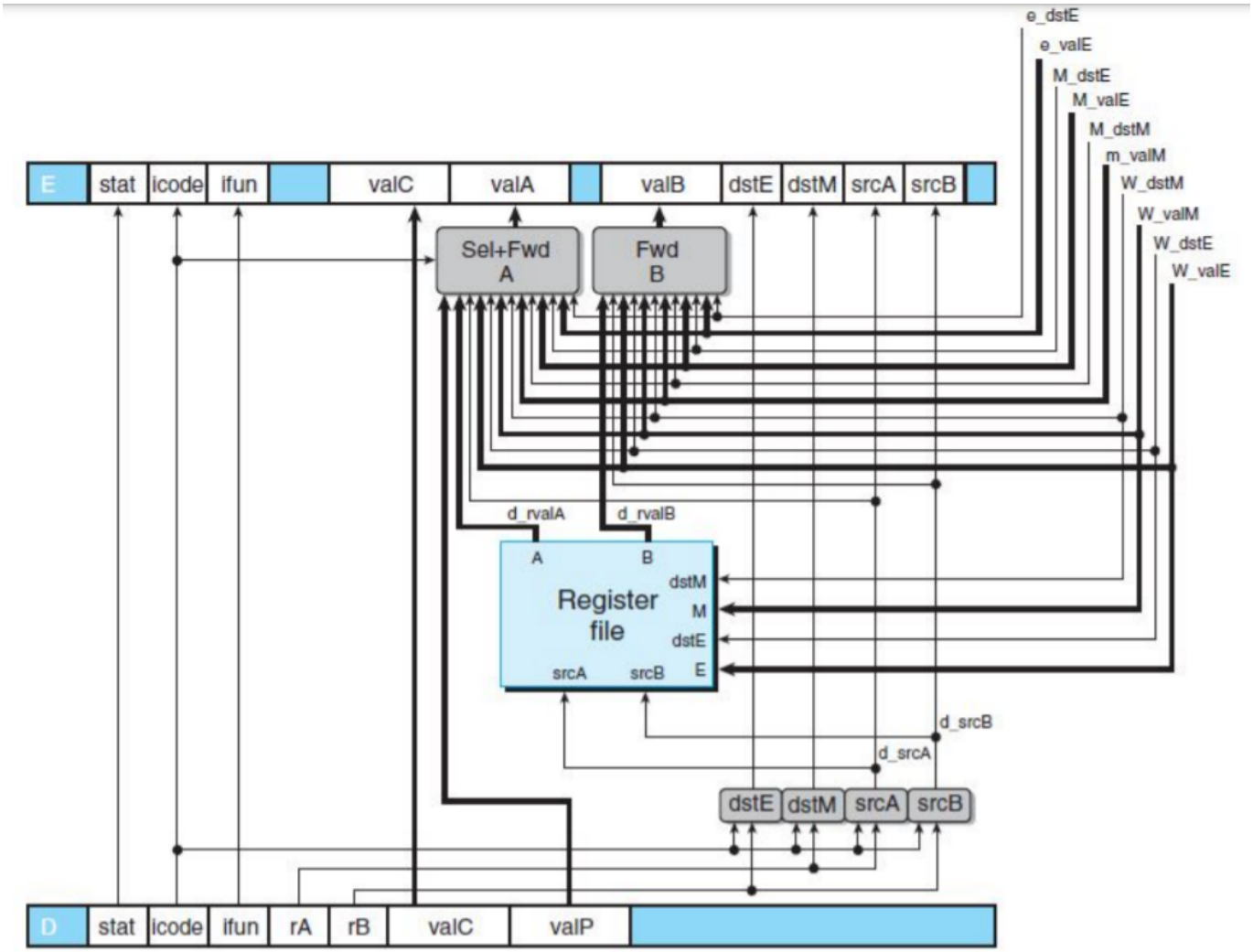
In the fetch stage, foremost step is to fetch the instruction from the memory address that is currently stored in the PC (program counter) and stored into the instruction register. Now, when this step ends, the PC points to the next instruction to be read in the next cycle and the cycle continues like this to fetch all the instructions. The first four bits are assigned to icode and the next 4 bits are assigned to ifun by the split module, if the pc points to a valid address in the

instruction memory. The align module describes how the processor extracts the remaining fields from an instruction, depending on whether or not the instruction has a register specifier byte. The last module used in Fetch module is known as pc_inc_ that is used to generate signal named valP, which is based upon the other values from other units.

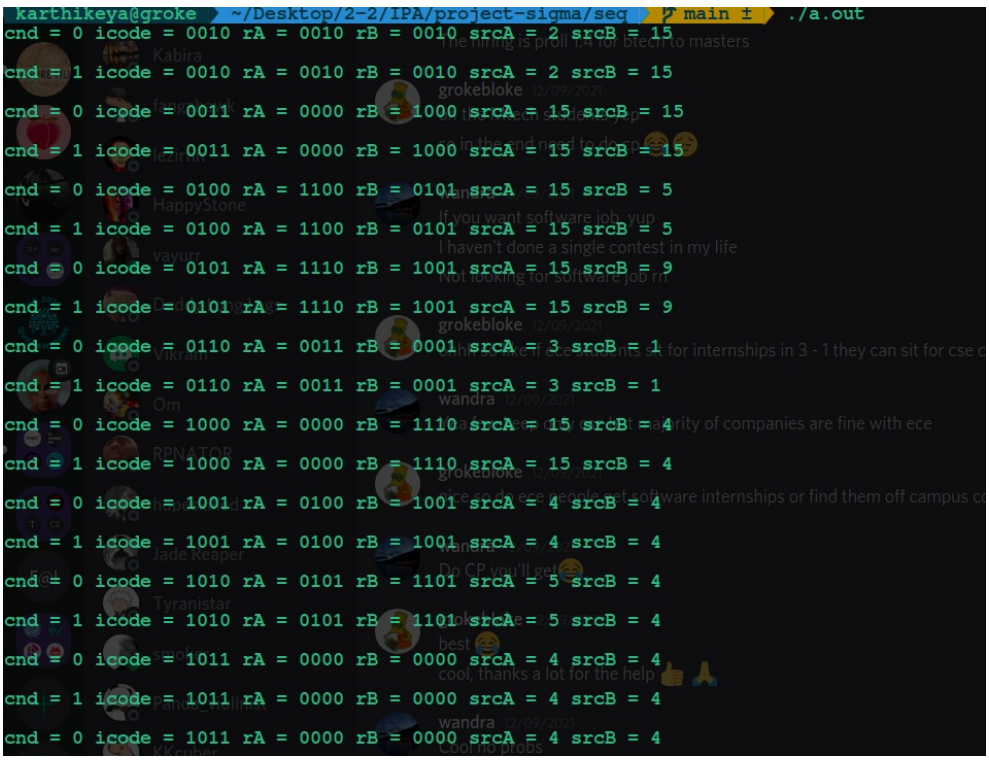


Decode :-

In the decode, we take icode,rA and rB as inputs and then depending on the value of icode we rA and assign it to valA, also read rB, and assign it to valB. The srcA and srcB blocks will determine whether we need to write valA, valB depending on icode. The block named register file in the diagram is used to get the values from the registers used in the instructions which is to be decoded. d. The ports A and B have address inputs as srcA and srcB. Ports E and M have address inputs dstE and dstM. The destination where valE is stored is held by the register ID dstE, similarly for valM its held by dstM. In the write back, valE and valM needs to write into the register file depending on the icode.



Attached below is the simulated output for the given testbench of the decode module -



Execute :-

The execute module is used to carry out the functional operations which are implemented by ALU. This stage computes the effective memory address, increments or decrements of the stack pointer. The module is used to send the decoded data as control signals to the functional units and then the required instructions are carried out before being transferred to the ALU, and finally the result being stored in the register. For conditional move instruction , the condition codes will be evaluated here, having inputs valC ,valA,valB,icode and ifun. Based on the inputs the module outputs valE and sets the cnd bit.

Our ALU generates the three signals everytime it operates on which the condition codes are based, i.e zero,sign,overflow.The execute stage connects to the memory stage, that determines where to store the computed values in memory.

64 Bit ALU

Functions performed by the ALU:

1. ADD - 64 bits
2. SUBTRACT - 64 bits
3. AND - 64 bits
4. XOR - 64 bits

Objective: To Build a 64 Bit ALU which can carry put the following functions.

- ADD
- SUB
- AND
- XOR

Two 64 bit inputs and a 2 - bit control signal is given to the ALU. The 2 - bit control signal controls the final output of the ALU and decides which functions output to take.

```
Control 0 - ADD x and y


Control 1 - Subtract x and y


Control 2 - AND x and y


Control 3 - XOR x and y
```

Modules

1. ADD

```
Control Signal for ADD = 00
```

Deployed a full adder with initial carry bit = 0.

An array of 64 full adders is used. 64 length long arrays of $sum[64]$ and $carry[64]$ are used. The carry_out of the full adder is the carry_in of the next full adder. S

Since we are making a full adder the initial carry is 0. $\therefore carry[0] = 0$.

For calculating the sum and carry of each full adder we use the following equations:

Taking A and B to be 2 - 64 bit inputs.

$$sum[i] = (A[i] \oplus B[i]) \oplus carry[i]$$
$$carry[i+1] = (A[i] \oplus B[i]) \cdot carry[i] + A[i] \cdot B[i]$$
$$\therefore carry_{out} = (A[i] \oplus B[i]) \cdot carry_{in} + A[i] \cdot B[i]$$

Truth Table:

| $A[i]$ | $B[i]$ | $carry[i]$ | $carry[i+1]$ | $sum[i]$ |

| -------- | --------- | ------------ | ---------------- | ----------- |

| 0 | 0 | 0 | 0 | 0 |

0	9	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

To detect the overflow we xor the last 2 carry's.

Overflow = $carry[63] \oplus carry[64]$

Code for ADD operation:

```
module add64(a,b,sum,of);

input signed [63:0] a,b;

output signed [63:0] sum;

output of;


// carry out

wire [64:0] carry;

assign carry[0] = 1'b0;


genvar i;

generate

for (i = 0; i < 64; i = i + 1)

begin

 add11 x(a[i], b[i], carry[i], sum[i],carry[i + 1]);

end

endgenerate


xor(of, carry[64], carry[63]);


endmodule
```

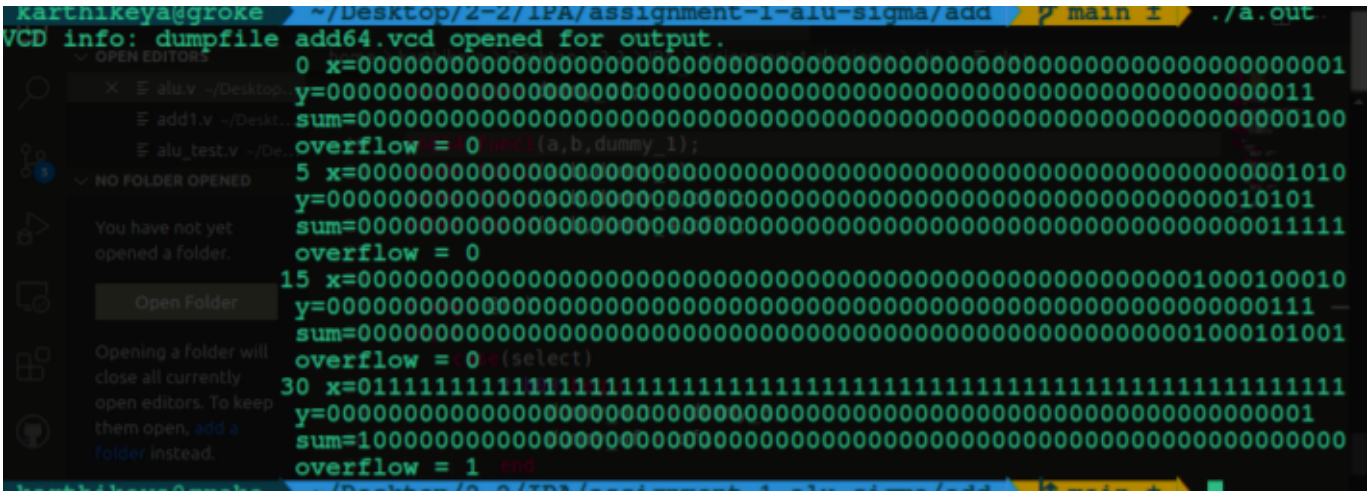
Code for full adder (module add11)

```
module add11(a,b,carry_in,sum,carry_out);

input a,b,carry_in;

output sum,carry_out;



wire t1,t2,t3;

xor (sum,a,b,carry_in);

xor (t1,a,b);

and (t2,t1,carry_in);

and (t3,a,b);

or (carry_out,t2,t3);



endmodule
```

Testbench

```
module testbench;

reg signed [63:0] a;

reg signed [63:0] b;

wire signed [63:0] sum;

wire signed of;

add64 y(a,b,sum,of);



initial begin

 $dumpfile("add64.vcd");

 $dumpvars(0,testbench);

 $monitor($time, " x=%b \n\t\t y=%b \n\t\t sum=%b\n\t\t overflow = %b", a, b, sum,of);

 a = 64'b01;

 b = 64'b11;

 #5 a = 64'd10; b = 64'd21;

 #10 a = 64'd546 ; b = 64'd7;

 #15 a = 64'd9223372036854775807; b = 64'd1;


end


endmodule
```

Output:



2. SUBTRACT

Very similar to the ADD operation, but instead of using the input B directly we use the negation of B, i.e $\neg B$. We are doing the Subtract operation from the 2's complement method.

To do SUB operation we need to use the control signal 1.

```
Control signal = 01
```

2's complement:

We negate all the bits which is nothing but 1's complement and add 1 to it.

i.e $\neg B + 1$.

We do 1's complement by negating the 64 bits of the input B .

We add 1 by assigning the initial carry as 1. $carry[0] = 1$. This is one important difference between the ADD and SUB operation.

After doing 2's complement we pass the initial input A and 2's complement of B into the full adder.

Here also we detect overflow by doing the xor operation on the last two carry's.

Overflow = $carry[63] \oplus carry[64]$.

Code for SUB:

```
module sub64(a,b,sum,of);

input signed [63:0] a,b;

output signed [63:0] sum;

output of;


// 2's complement addition


wire signed [63:0] not_b;


not64 func(b,not_b);


wire [64:0] carry;

assign carry[0] = 1'b1;


genvar i;

generate

for (i = 0; i < 64; i = i + 1)

begin

 add11 x(a[i], not_b[i], carry[i], sum[i],carry[i + 1]);

end

endgenerate


xor(of, carry[64], carry[63]);


endmodule
```

Code for the "NOT" module:

```
module not64(b,not_b);

input signed [63:0] b;

output signed [63:0] not_b;



genvar i;

generate for (i = 0; i < 64 ; i = i + 1)

begin

 not(not_b[i],b[i]);

end

endgenerate



endmodule
```

Testbench:

```
module testbench;

reg signed [63:0] a;

reg signed [63:0] b;

wire signed [63:0] sum;

wire signed of;

sub64 y(a,b,sum,of);



initial begin

 $dumpfile("sub64.vcd");

 $dumpvars(0,testbench);

 $monitor($time, " x=%b \n\t\t y=%b \n\t\t sum=%b\n\t\t overflow = %b", a, b, sum,of);

 a = 64'b01;

 b = 64'b11;

 #5 a = 64'd10; b = 64'd21;

 #10 a = 64'd546 ; b = 64'd7;

 #15 a = 64'd9223372036854775807; b = ~64'd1;

end


endmodule
```

Output:

3. AND

```
Control signal = 10
```

We do a simple AND operation on the two 64 - input bits. We do it bit by bit through a for loop.

Truth Table:

| $A[i]$ | $B[i]$ | Out[i] |

| -------- | ---------- | ------ |

| 0 | 0 | 0 |

| 0 | 1 | 0 |

| 1 | 0 | 1 |

| 1 | 1 | 1 |

$$Out[i] = A[i] \cdot B[i]$$

The overflow is 0 for the AND operation.

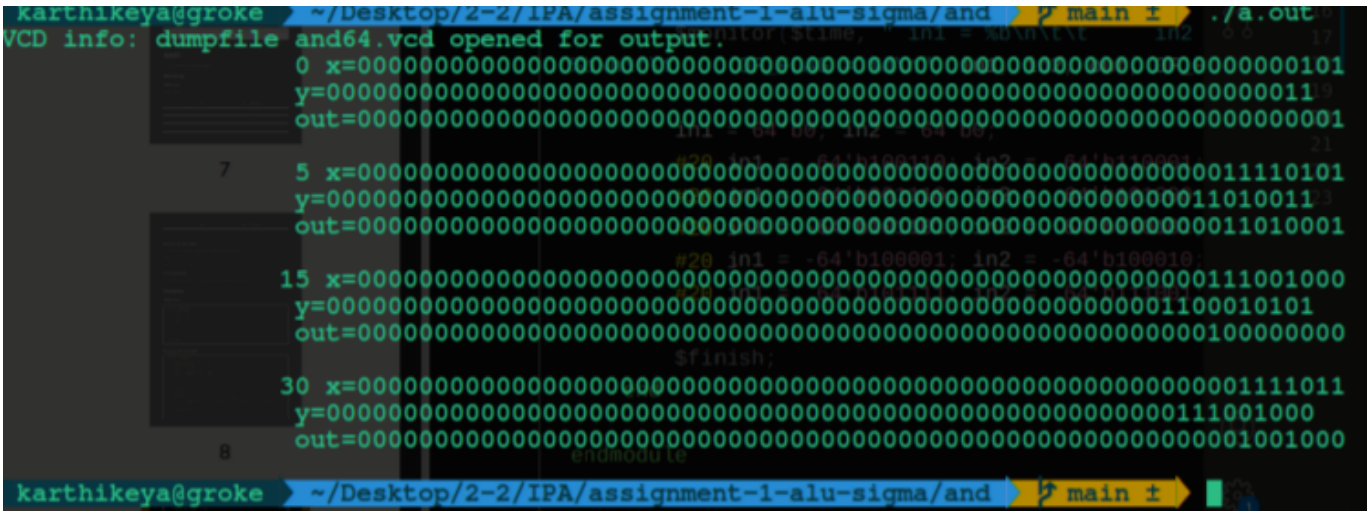
Code:

```
module and64(a,b,out);

input signed [63:0] a;

input signed [63:0] b;

output signed [63:0] out;

genvar i;

generate for (i = 0 ; i <= 63 ; i = i + 1) begin

 and (out[i],a[i],b[i]);

end

endgenerate

endmodule
```

Testbench:

```
module testbench;

reg signed [63:0] a;

reg signed [63:0] b;

wire signed [63:0] out;

and64 func(a,b,out);



initial begin

 $dumpfile("and64.vcd");

 $dumpvars(0,testbench);

 $monitor($time, " x=%b \n\t\t y=%b \n\t\t out=%b\n\t\t" , a, b, out);

 a = 64'b0101;

 b = 64'b11;

 #5 a = 64'b11110101; b = 64'd211;

 #10 a = 64'd456; b = 64'd789;

 #15 a = 64'd123; b = 64'd456;


end


endmodule
```

Output:



4. XOR

```
Control Signal = 11
```

We do a simple XOR operation on the two 64 - input bits. We do it bit by bit through a for loop.

Overflow is 0 for the XOR operation.

Truth Table:

| $A[i]$ | $B[i]$ | $Out[i]$ |

| -------- | -------- | -------------- |

| 0 | 0 | 0 |

| 0 | 1 | 1 |

| 1 | 0 | 1 |

| 1 | 1 | 0 |

$$Out[i] = A[i] \oplus B[i]$$

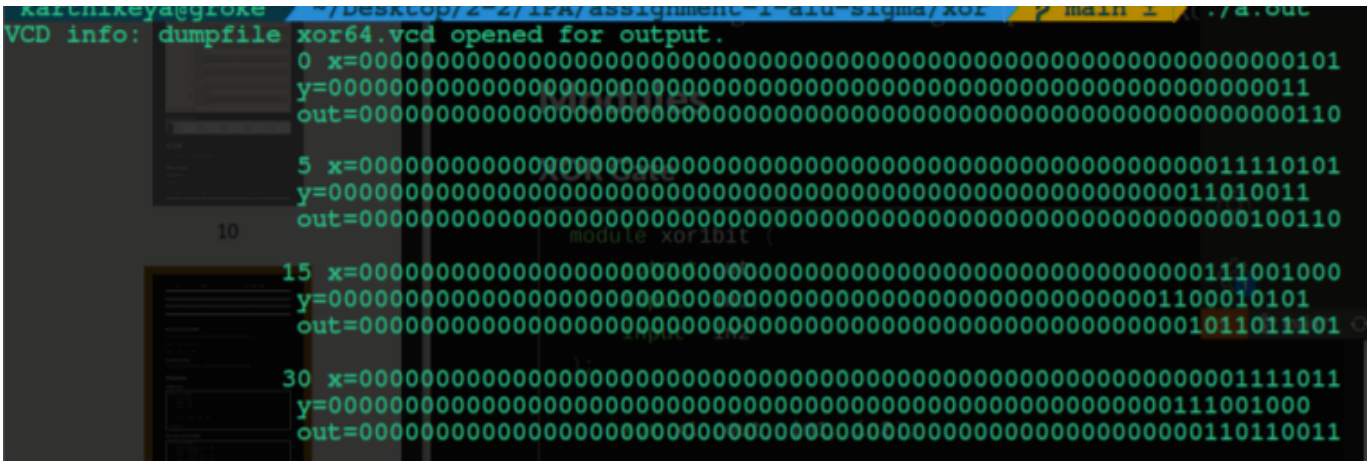
Code for XOR:

```
module xor64(a,b,out);

input signed [63:0] a;

input signed [63:0] b;

output signed [63:0] out;

genvar i;

generate for (i = 0 ; i <= 63 ; i = i + 1) begin

 xor (out[i],a[i],b[i]);

end

endgenerate

endmodule
```

Testbench:

```
module testbench;

reg signed [63:0] a;

reg signed [63:0] b;

wire signed [63:0] out;

xor64 func(a,b,out);



initial begin

 $dumpfile("xor64.vcd");

 $dumpvars(0,testbench);

 $monitor($time, " x=%b \n\t\t y=%b \n\t\t out=%b\n\t\t" , a, b, out);

 a = 64'b0101;

 b = 64'b11;

 #5 a = 64'b11110101; b = 64'd211;

 #10 a = 64'd456; b = 64'd789;

 #15 a = 64'd123; b = 64'd456;


end


endmodule
```

Output:



ALU

The ALU performs arithmetic,bitwise and logic operations on integer binary numbers. The ALU module contains the 4 operational modules i.e,

- ADD
- SUB
- AND
- XOR

Depending upon the control signal bit, the output is chosen. Similarly the overflow is also detected. We use the switch cases in order to select the operation. We initially compute all the operation (i.e modules of all operations), and the output is stored in a form of a dummy output. After getting the control signal input we assign the final output and overflow.

Code:

ALU wrapper

```
`include "../xor/xor64.v"

`include "../add/add1.v"

`include "../sub/not.v"

`include "../add/add64.v"

`include "../sub/sub64.v"

`include "../and/and64.v"


module alu(a,b,select,out,of);

input signed [63:0] a,b;

input signed [1:0] select;

output signed [63:0] out;

output of;


wire signed [63:0] dummy_1,dummy_2,dummy_3,dummy_4;

wire signed of1,of2;

reg signed [63:0] dummy_out;

reg signed dummy_of;


and64 func1(a,b,dummy_1);

xor64 func2(a,b,dummy_2);

add64 func3(a,b,dummy_3,of1);

sub64 func4(a,b,dummy_4,of2);



always @(*)

begin
```

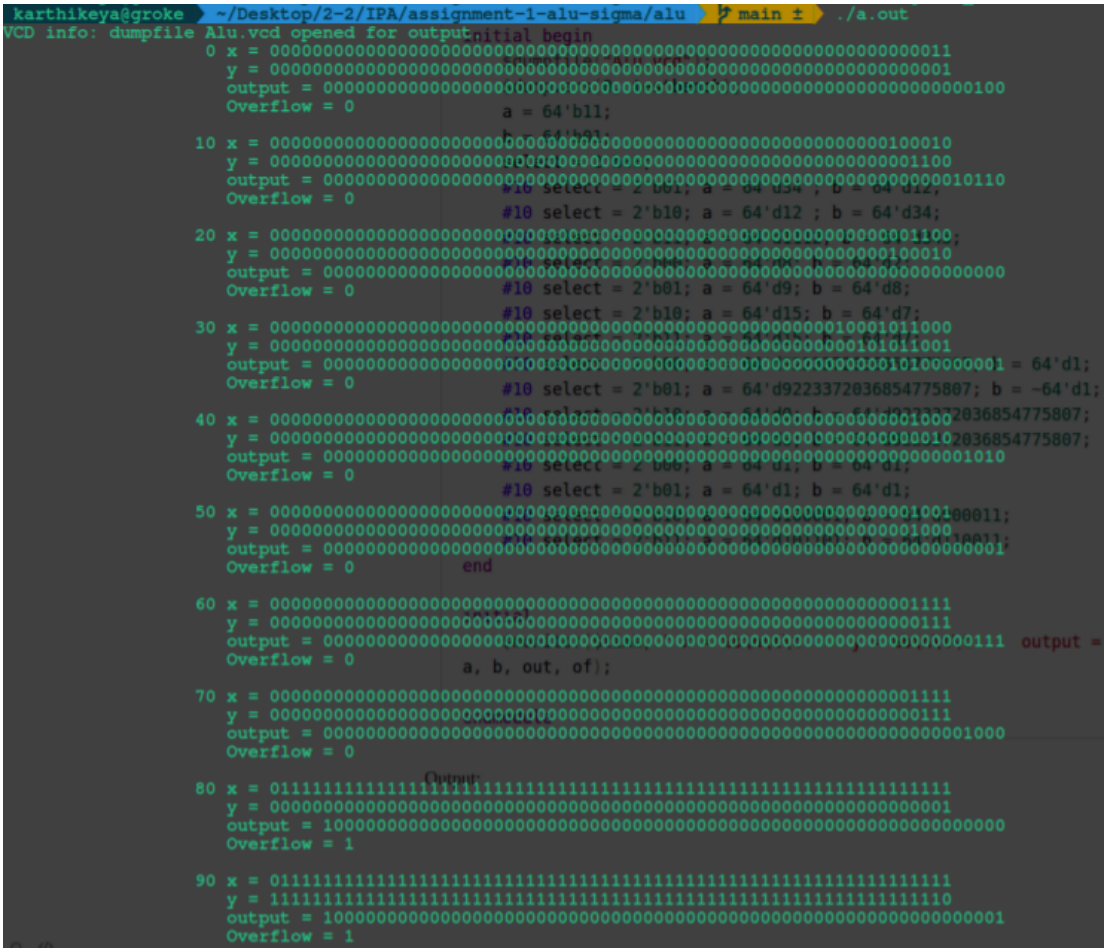
```
 case(select)
 2'b00:begin
 dummy_out = dummy_3;
 dummy_of = of1;
 end
 2'b01:begin
 dummy_out = dummy_4;
 dummy_of = of2;
 end
 2'b10:begin
 dummy_out = dummy_1;
 dummy_of = 1'b0;
 end
 2'b11:begin
 dummy_out = dummy_2;
 dummy_of = 1'b0;
 end
 endcase
end


assign out = dummy_out;
assign of = dummy_of;


endmodule
```

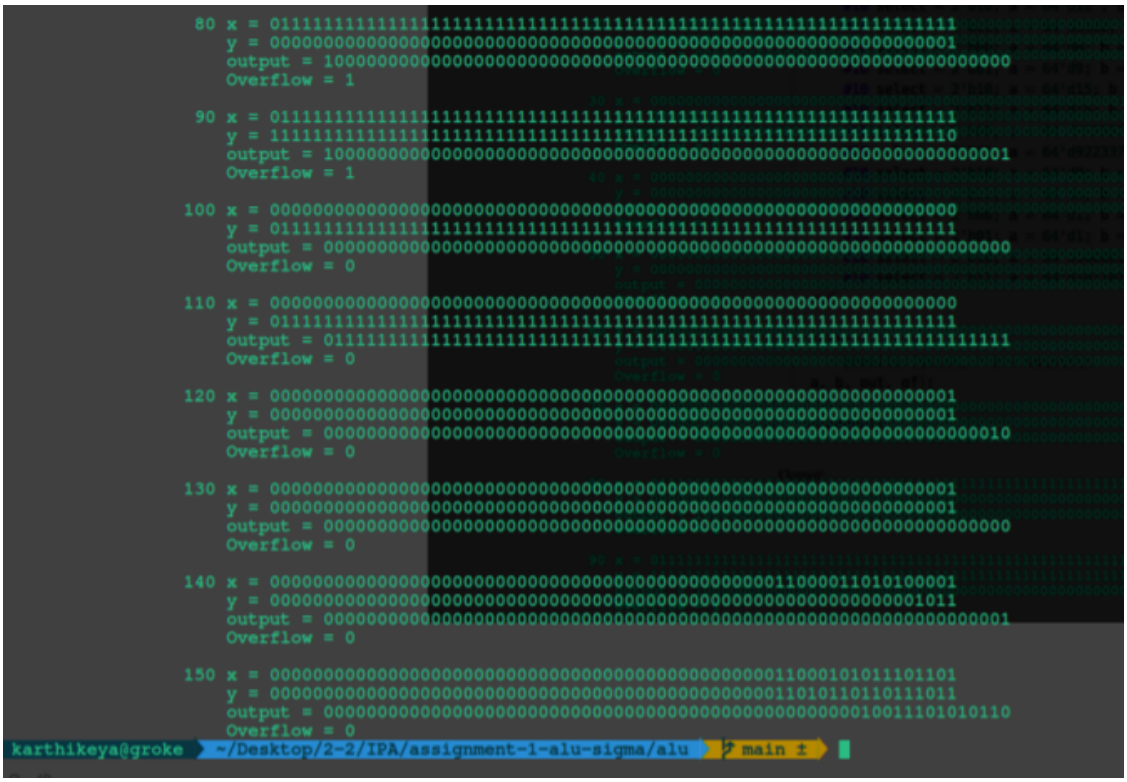
Testbench:

```
module testbench;

reg [1:0] select;

reg signed [63:0] a;

reg signed [63:0] b;


wire signed [63:0] out;

wire of;


alu func(a,b,select,out,of);


initial begin

 $dumpfile("Alu.vcd");

 $dumpvars(0, testbench);

 a = 64'b11;

 b = 64'b01;

 select = 2'b00;

 #10 select = 2'b01; a = 64'd34 ; b = 64'd12;

 #10 select = 2'b10; a = 64'd12 ; b = 64'd34;

 #10 select = 2'b11; a = 64'd1112; b = 64'd345;

 #10 select = 2'b00; a = 64'd8; b = 64'd2;

 #10 select = 2'b01; a = 64'd9; b = 64'd8;

 #10 select = 2'b10; a = 64'd15; b = 64'd7;

 #10 select = 2'b11; a = 64'd15; b = 64'd7;

 #10 select = 2'b00; a = 64'd9223372036854775807; b = 64'd1;

 #10 select = 2'b01; a = 64'd9223372036854775807; b = ~64'd1;
```

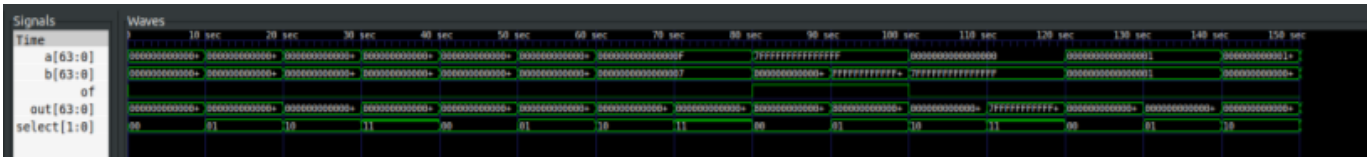
```
 #10 select = 2'b10; a = 64'd0; b = 64'd9223372036854775807;

 #10 select = 2'b11; a = 64'd0; b = 64'd9223372036854775807;

 #10 select = 2'b00; a = 64'd1; b = 64'd1;

 #10 select = 2'b01; a = 64'd1; b = 64'd1;

 #10 select = 2'b10; a = 64'd100001; b = 64'd000011;

 #10 select = 2'b11; a = 64'd101101; b = 64'd110011;

end



initial

 $monitor($time, " x = %b\n\t\t y = %b\n\t\t output = %b\n\t\t Overflow = %d\n", a, b,
out, of);



endmodule
```

Output:

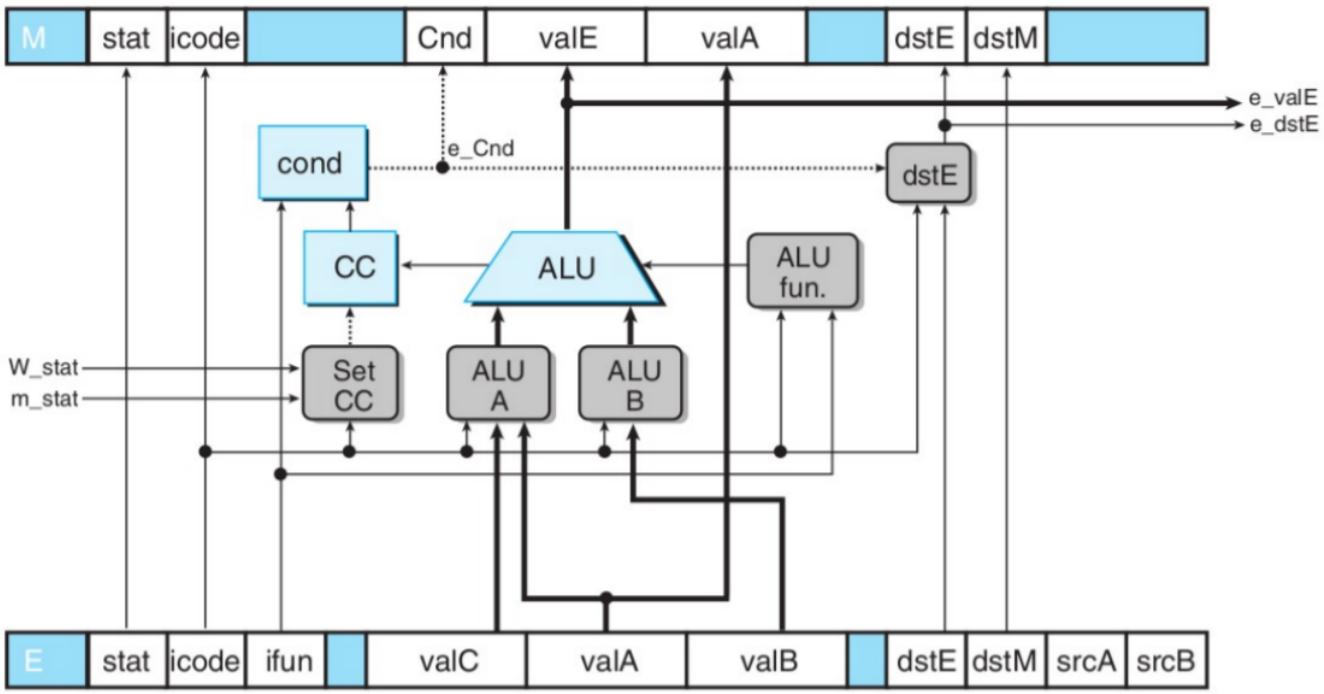

karthikeya@groke ~/Desktop/2-2/IPA/assignment-1-alu-sigma/alu main ± ./a.out
VCD info: dumpfile Alu.vcd opened for output.
0 x = 00000000000000000000000000000000000000000000000000000000000000000011
y = 00000000000000000000000000000000000000000000000000000000000000000001
output = 00000000000000000000000000000000000000000000000000000000000000000100
Overflow = 0
a = 64'b11;
10 x = 00000000000000000000000000000000000000000000000000000000000000100010
y = 00000000000000000000000000000000000000000000000000000000000000001100
output = 00000000000000000000000000000000000000000000000000000000000010110
Overflow = 0
#10 select = 2'b10; a = 64'd12 ; b = 64'd34;
20 x = 00000000000000000000000000000000000000000000000000000000000001100
y = 00000000000000000000000000000000000000000000000000000000000100010
output = 00000000000000000000000000000000000000000000000000000000000000000
Overflow = 0
#10 select = 2'b01; a = 64'd9; b = 64'd8;
#10 select = 2'b10; a = 64'd15; b = 64'd7;
30 x = 00000000000000000000000000000000000000000000000000000010001011000
y = 00000000000000000000000000000000000000000000000000000000101011001
output = 00000000000000000000000000000000000000000000000010100000001
Overflow = 0
#10 select = 2'b01; a = 64'd9223372036854775807; b = -64'd1;
40 x = 00000000000000000000000000000000000000000000000000000000001000
y = 00000000000000000000000000000000000000000000000000000000000010
output = 00000000000000000000000000000000000000000000000000000001010
Overflow = 0
#10 select = 2'b01; a = 64'd1; b = 64'd1;
50 x = 00000000000000000000000000000000000000000000000000000010100011
y = 00000000000000000000000000000000000000000000000000000000001000
output = 00000000000000000000000000000000000000000000000000000000000001
Overflow = 0
end
60 x = 00000000000000000000000000000000000000000000000000000000001111
y = 00000000000000000000000000000000000000000000000000000000000011
output = 00000000000000000000000000000000000000000000000000000000000111
Overflow = 0
a, b, out, of);
70 x = 00000000000000000000000000000000000000000000000000000000001111
y = 00000000000000000000000000000000000000000000000000000000000111
output = 00000000000000000000000000000000000000000000000000000000001000
Overflow = 0
Output:
80 x = 01111111111111111111111111111111111111111111111111111111111111111111
y = 00000000000000000000000000000000000000000000000000000000000000000001
output = 10000000000000000000000000000000000000000000000000000000000000000000
Overflow = 1
90 x = 01111111111111111111111111111111111111111111111111111111111111111111
y = 11111111111111111111111111111111111111111111111111111111111111111110
output = 10000000000000000000000000000000000000000000000000000000000000000001
Overflow = 1


80 x = 01111111111111111111111111111111111111111111111111111111111111111111
y = 00000000000000000000000000000000000000000000000000000000000000000001
output = 10000000000000000000000000000000000000000000000000000000000000000000
Overflow = 1
90 x = 01111111111111111111111111111111111111111111111111111111111111111111
y = 11111111111111111111111111111111111111111111111111111111111111111110
output = 10000000000000000000000000000000000000000000000000000000000000000001
Overflow = 1
100 x = 00000000000000000000000000000000000000000000000000000000000000000000
y = 01111111111111111111111111111111111111111111111111111111111111111111
output = 00000000000000000000000000000000000000000000000000000000000000000000
Overflow = 0
110 x = 00000000000000000000000000000000000000000000000000000000000000000000
y = 01111111111111111111111111111111111111111111111111111111111111111111
output = 01111111111111111111111111111111111111111111111111111111111111111111
Overflow = 0
120 x = 00000000000000000000000000000000000000000000000000000000000000000001
y = 00000000000000000000000000000000000000000000000000000000000000000001
output = 00000000000000000000000000000000000000000000000000000000000000000010
Overflow = 0
130 x = 00000000000000000000000000000000000000000000000000000000000000000001
y = 00000000000000000000000000000000000000000000000000000000000000000001
output = 00000000000000000000000000000000000000000000000000000000000000000000
Overflow = 0
140 x = 00000000000000000000000000000000000000000000000011000011010100001
y = 00000000000000000000000000000000000000000000000000000000000000001011
output = 00000000000000000000000000000000000000000000000000000000000000000001
Overflow = 0
150 x = 00000000000000000000000000000000000000000000000011000101011101101
y = 00000000000000000000000000000000000000000000000011010110110111011
output = 00000000000000000000000000000000000000000000000000000000100111010010110
Overflow = 0
karthikeya@groke ~/Desktop/2-2/IPA/assignment-1-alu-sigma/alu main ±

GTKWAVE:

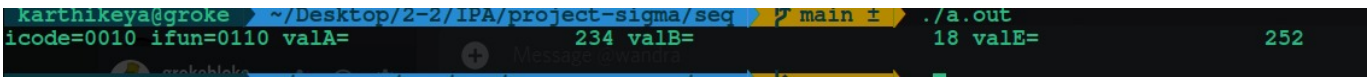


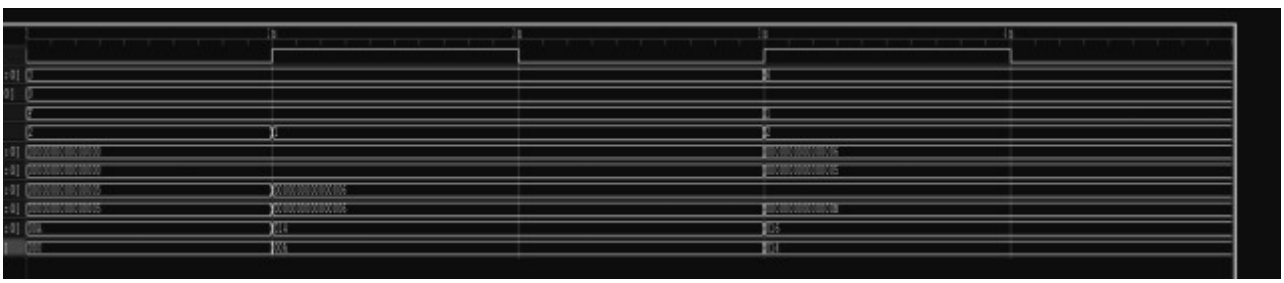
Successfully implemented a 64 bit ALU which can ADD,SUB,AND,XOR.



Output -

Attached below is the simulated output for the given testbench of the decode module -



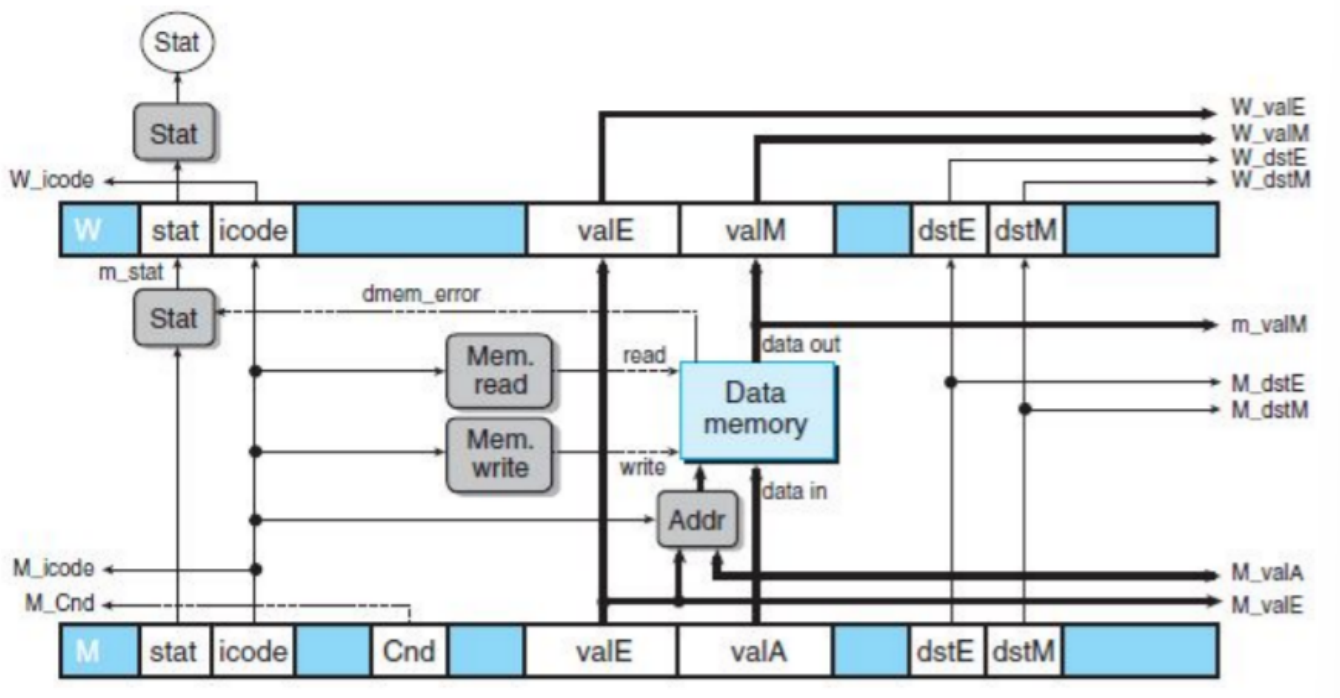


Memory :-

Memory stage can either read data from the memory or write data into the memory. The inputs for the memory stage are icode, valA, valE, valP and icode. The output is valM and also displays

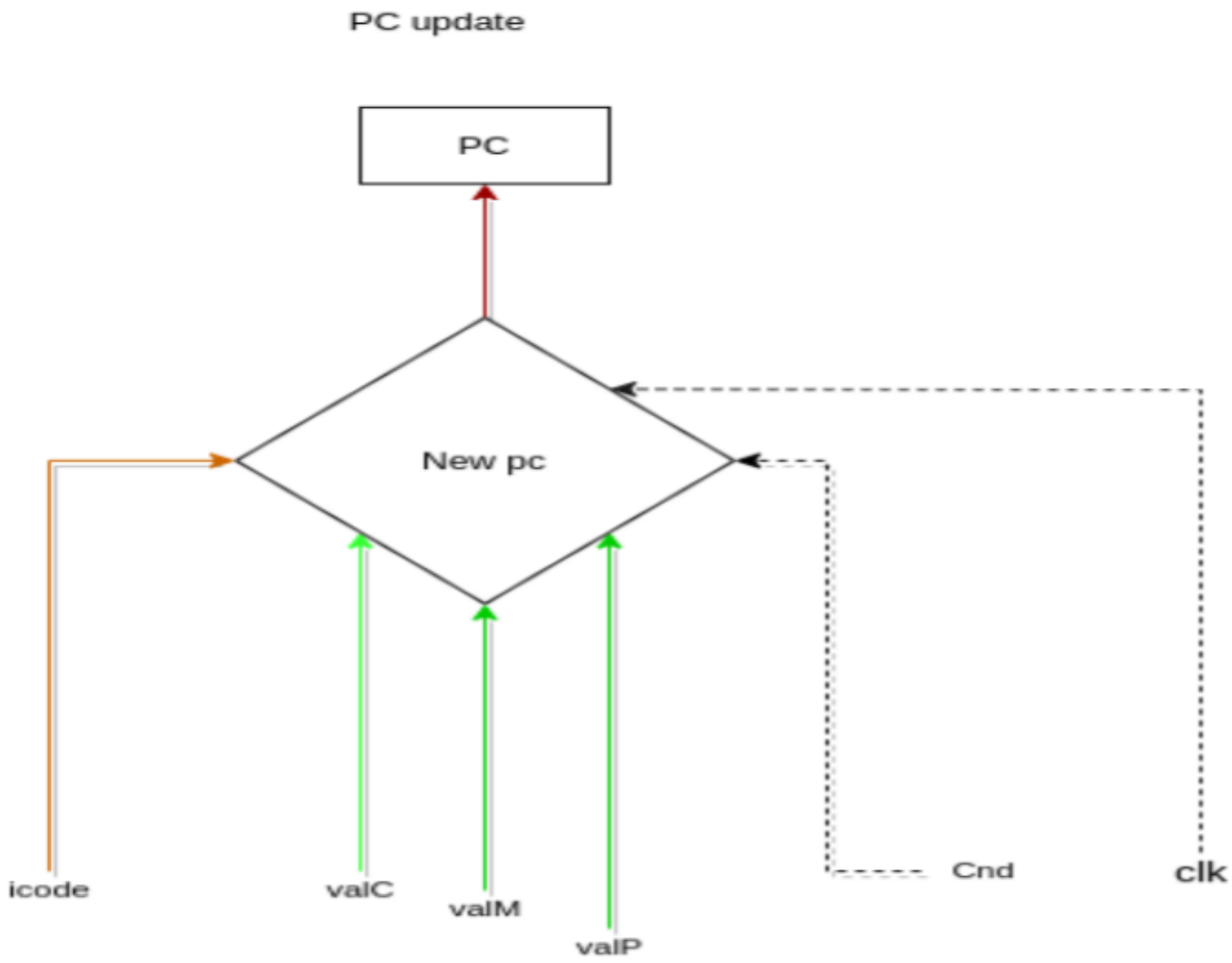
the current instruction status of the processor. When the read operation is performed, valM is generated. valE and valM are addresses for memory read and write.

mem_addr output the address of the memory that needs to be read or written. **The mem_data** outputs the data if anything needs to be written into the memory. mem_r and mem_w sets the read or write flags to 1 based on the instruction (icode).

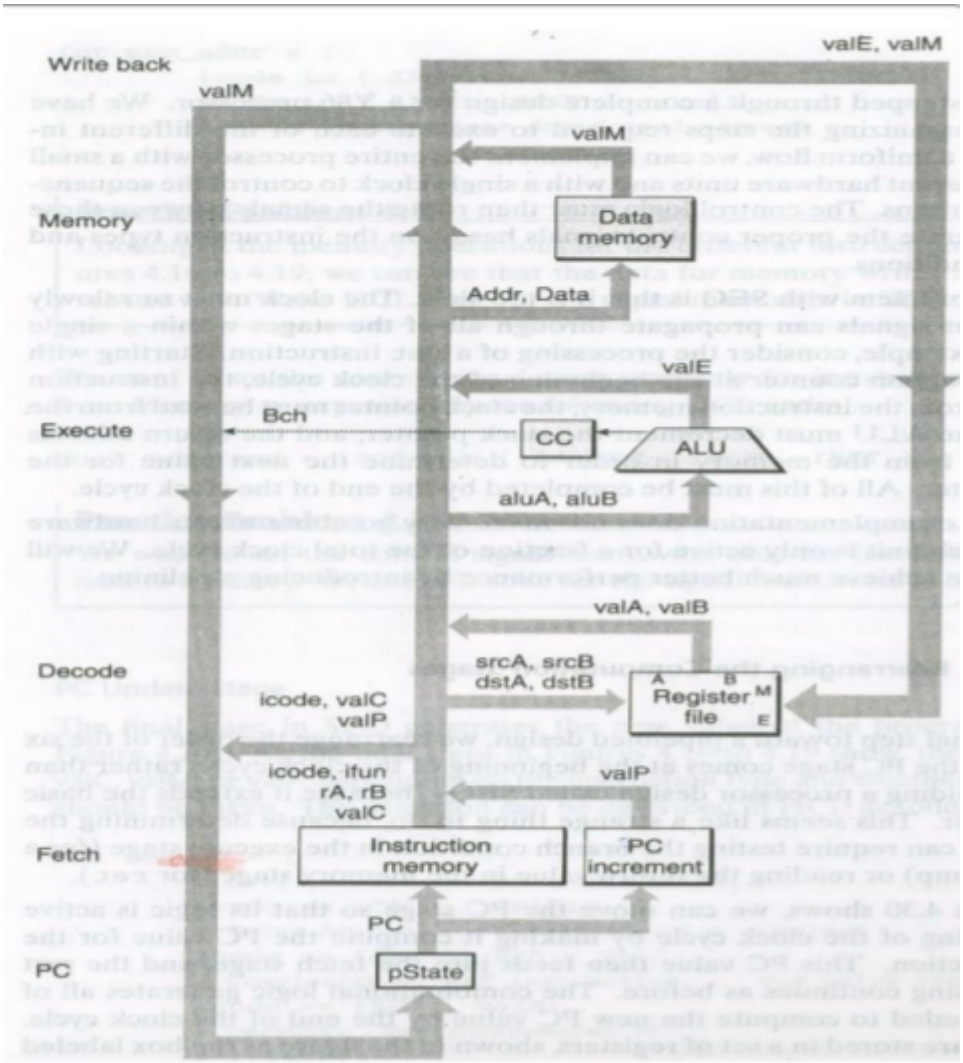


PC update :-

PC update is a small module updating the PC value to the next instruction it needs to be pointed. The PC value can change to the next instruction or jump to a destination pc value if the condition is satisfied or based on call, ret instruction. All this is done based on the icode. The output of the pcupdate block is the next pc value. The pc update is only updated when all the previous modules are completed. We have used a clock and for every rising edge of the clk we update the pc value.



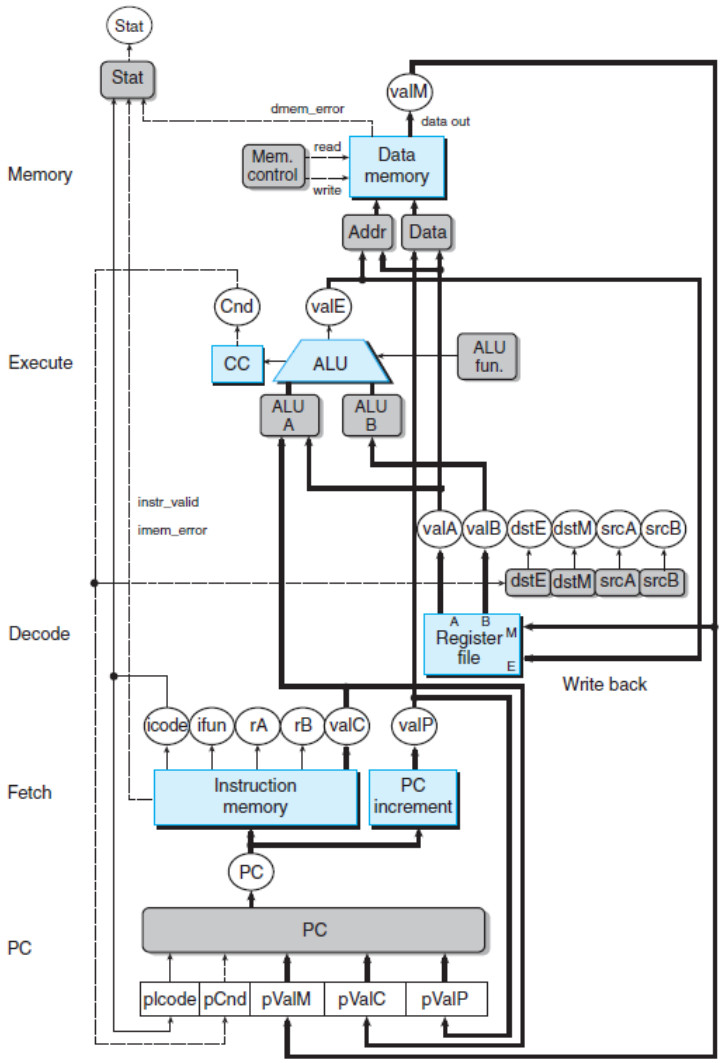
Overall Implementation Flowchart for the Sequential Processor :



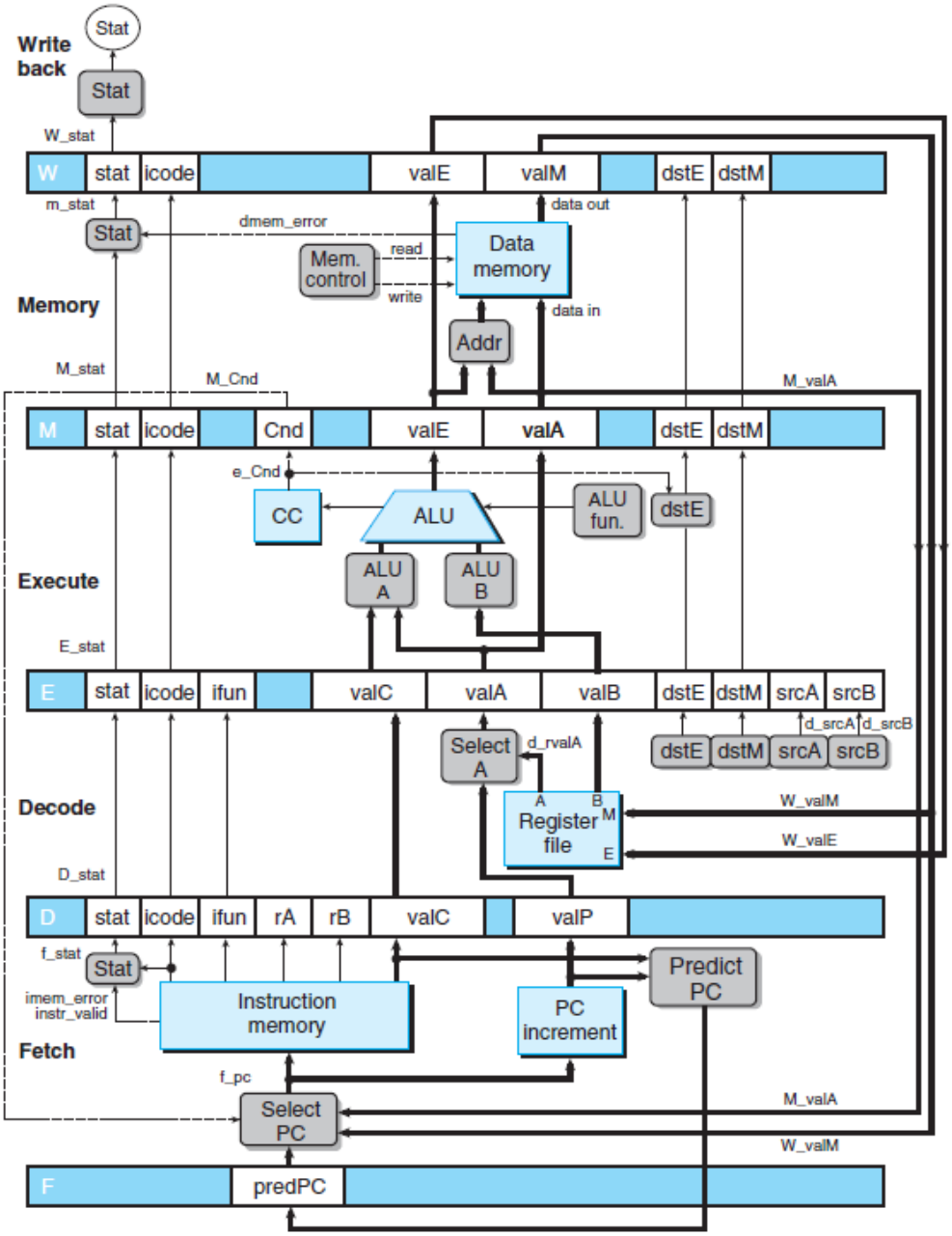
Pipeline processor :-

The performance of a processor is optimized by implimenting pipelining(pipeline registers) into our present sequential processor design. We can observe that any module after finishing the execution of the instruction , the module will remain at rest until the next instruction is fetched or its previous blocks completes their execution. This is a clear waste of resources on running the module blocks only for small amount of time in a clock cycle. In a pipelined architecture , every block will start their execution at every clock cycle. Pipelining is the process of accumulating instruction from the processor through a pipeline. It enables storing and executing instructions in an efficient process. Pipelining increases the overall instruction throughput.

Before pipelining implementation :-



After pipelining implementation :-

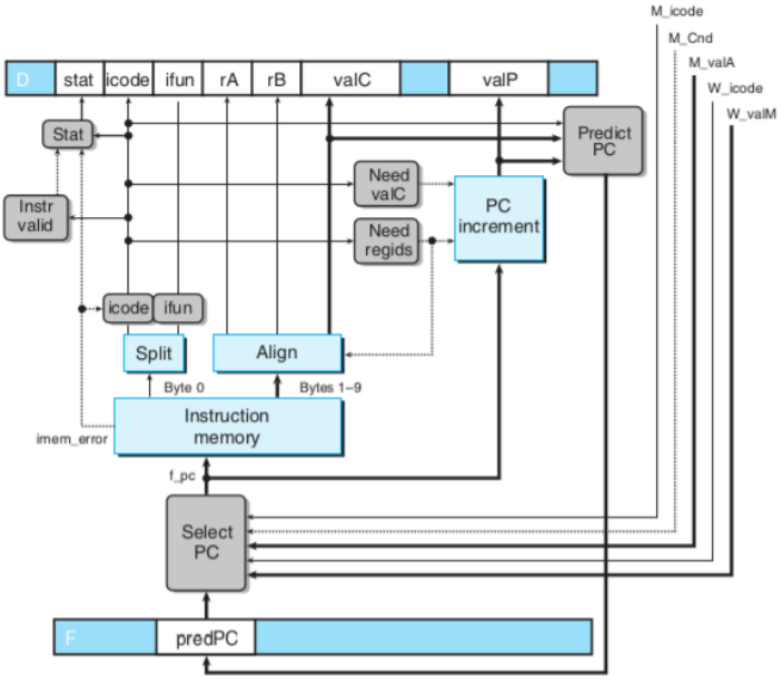


Funtionality of pipeline in individual modules :-

Fetch -

The key difference now observed will be that now the PC update ,which includes selecting the current PC and then computing the incremented PC, will happen in the fetch stage so that it doesnt need to wait for all the remaining stages to complete execution.

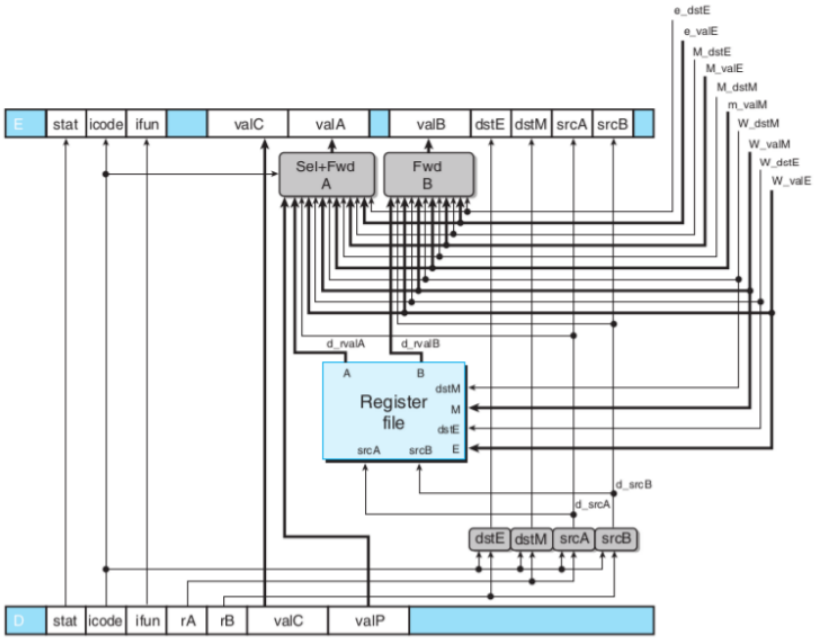
Attached below is the flow chart of the fetch stage :-



Decode -

The pipeline registers present as a part of this module, hold the precalculated value of PC and the other holds the most recently decoded instruction and sends the read values(from the register file) to the execute stage.

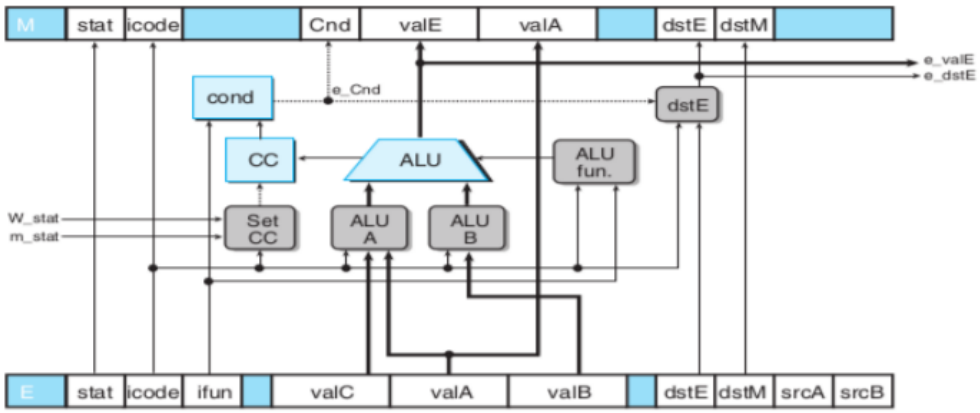
Attached below is the flow chart of the decode stage :-



Execute -

The pipeline registers present as a part of this module, take inputs from the decode module(M register) at rising edge of clk and writing the output to the W register, whose values are going to be used in the decode module. The register holds information about branch conditions and branch targets for conditional jumps.

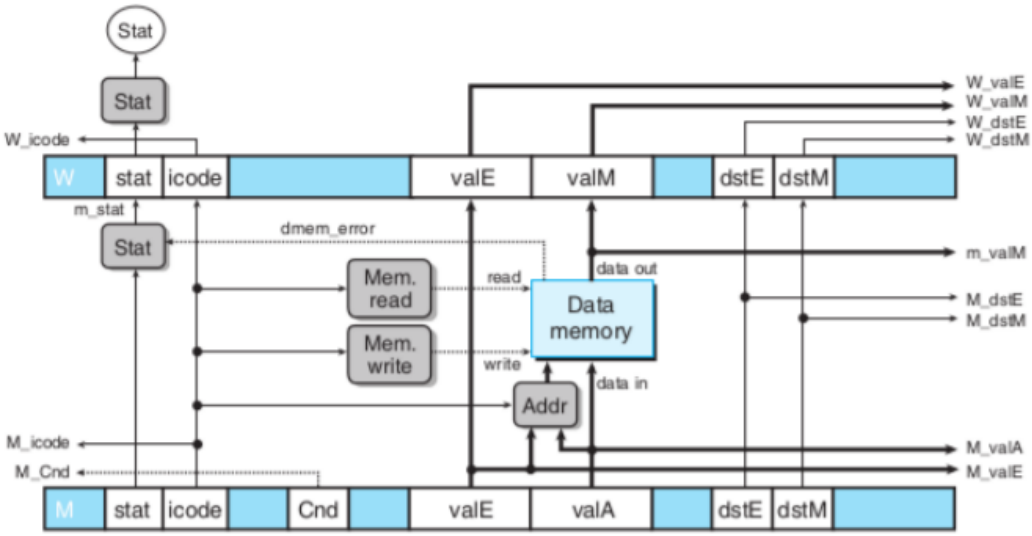
Attached below is the flow chart of the execute stage :-



Memory -

The pipeline registers present as a part of this module, supply the computed results to the register file for writing and the return address.

Attached below is the flow chart of the memory stage :-



Limitations (Pipelining Hazards) :-

We weren't able to handle instructions which had dependencies when instructions follow too closely. There are data dependencies where one instruction reads the register while the next one

reads it. We also will face control dependencies, such mispredicted branch and return and also an instruction might set PC in a way that the pipeline cannot predict it correctly.

Challenges faced :-

- Initially , we faced issues with implementing the clock cycle of our sequential circuit.
- Working and implementing pipelined registers.
- Implementing the jump instruction properly and efficiently.