Introduction to Processor Architecture

Team : **BICA**

Teammate 1 : Tadimarri Desika Sreeharsha

Roll Number : 2020102040

Teammate 2 : Sankalp S. Bhat

Roll Number : 2020112018

Project Report

Objective

The objective of this project was to implement a 64-bit processor based on the `y86` Instruction Set Architecture using the Verilog Hardware Descriptive Language. The processor had to be implemented in 2 styles, in a sequential design, and a 5 stage pipelined design, that also handles pipeline hazards.

ALU

Since the execute stage would require an ALU, we have implemented an ALU in Verilog to do the following :-

- AND
- XOR
- ADD and SUB

To simplify the modular building process for the modules, the ADD and SUB operation can be performed in the same block, depending on the carry input to the block.

The three blocks are then combined together using a wrapper ALU block, that is entirely written using structural modelling. To implement this, we have used a 4:1 MUX that computes the result for all 4 operations, and displays the result of the operation of choice by using the values of the Select lines (Control Inputs) as the parameters.

The inputs to the ALU are a bus that contains 2 select lines S_0 and S_1 that are 1 bit each, and signed 64 bit buses a and b which are operated upon.

If S_0 and S_1 are the values of the select lines (control inputs), they correspond to the following operations :

S_1S_0	$\text{Operation performed}$
00	Addition (ADD)
01	Subtraction (SUB)
10	Bitwise AND (AND)
11	Bitwise XOR (XOR)

Individual Operational Modules

ADD

Control Input for the operation : $S_1S_0 = 00$

The operations addition (ADD) and subtraction (SUB) are implemented in the same module.

The addition operation takes in two signed 64 bit numbers `a` and `b` and a carry in bit `c_in` as it's inputs. To perform the addition operation, `c_in` is set to 0. Then, we perform bitwise addition using a full adder module for each bit, storing the final overflow bit (`overflow`) and the output carry (`c_out`) separately.

The module outputs a signed 64-bit number `sum` and an overflow bit `overflow`. We can represent the operation as follows

$$\text{sum} = a \ + \ b \ + \ c_{\text{in}}$$

where, $+$ refers to algebraic addition.

In the case of addition, we have set c_{in} to 0. Hence,

$$\begin{aligned} \text{sum} &= a \ + \ b \ + \ 0 \\ &= \boxed{a \ + \ b} \end{aligned}$$

The 64-bit adder module implements addition by working on adding singular bits using a full adder module, and storing the carry bit and using it as input carry during the next iteration of the addition. This has been implemented by using 64 full adder modules in such a way that the output carry for the bits of

lower significance is the input carry (c_{in}) for the subsequent bits of higher significance. Here, the initial carry in for the least significant bits of `a` and `b` is 0 (for addition).

Codes

1-bit full adder

```
module bit1addsub (sum, c_out, a, b, c_in) ;

// structural modelling of full adder

output sum, c_out   ;
input   a , b, c_in ;

wire w1 ,w2 ,w3 ;  // intermediate wires

// 1st halfadder
xor G1 (w1 , a , b  )     ;    //  w1 = a xor b
and G2 (w2 , a , b )      ;    //  w2 = a.b

// 2nd halfadder
xor G3 (sum , c_in , w1 ) ;   // sum = a xor b xor c_in
and G4 (w3  , c_in , w1  ) ;  //  w3 = ( a xor b).c_in

// OR gate for carry out

or G5 ( c_out , w2 ,  w3 ) ;  // c_out = a.b + (a xor b ).c_in

endmodule
```

64-bit adder-subtractor

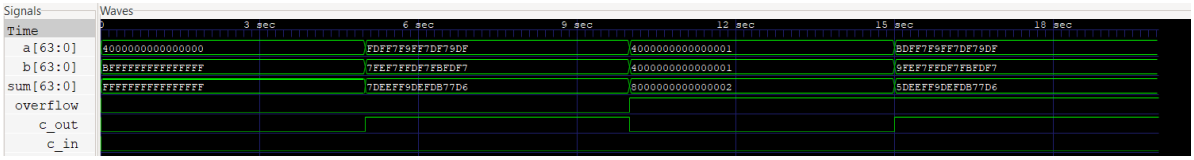
```
module bit64addsub( sum , c_out , a , b , c_in )  ;

output signed [63:0] sum   ;
output        c_out        ;
input  signed [63:0] a     ;
input  signed [63:0] b     ;
input         c_in         ;

wire [62:0] carry ;  // that takes carry from lower bit to the
input of the higher bit
wire [63:0] bx    ;  // data from b is tranferred to bx, and is
complemented if c_in is 1
```

```
// (for loop + generate) is used to create a set of XOR gates for
bitwise data transfer into bx from b
genvar j ;

generate

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

    begin

        xor addorsub( bx[j] , b[j] ,  c_in ) ;

    end

endgenerate

// ---------------------------------------------------------------
----------------- //

bit1addsub GaddLSB (.sum(sum[0]) , .c_out(carry[0]) , .a(a[0]) ,
.b(bx[0]), .c_in(c_in)) ;

// (for loop + generate) is used to create a list of fulladders
one for each bit
genvar i ;

generate

    for ( i = 1 ; i < 63 ; i = i+1 )

    begin

        bit1addsub Gaddinterior(.sum(sum[i]) , .c_out(carry[i]) ,
.a(a[i]) , .b(bx[i]) , .c_in(carry[i-1]) ) ;

    end

endgenerate

bit1addsub GaddMSB ( .sum(sum[63]) , .c_out(c_out) , .a(a[63]) ,
.b(bx[63]) , .c_in(carry[62]) ) ;

endmodule
```

Results from test cases

```
a      = 0100000000000000000000000000000000000000000000000000000000000000 =  4611686018427387904
b      = 1011111111111111111111111111111111111111111111111111111111111111 = -4611686018427387905
sum    = 1111111111111111111111111111111111111111111111111111111111111111 =                   -1
c_out = 0  c_in = 0
overflow = 0


a      = 1111110111111111011111111100111111111011111011111011110011011111 =  -144256338017420833
b      = 0111111111101111011111111111111011111011111111011111110111110111 =  9218727691014634999
sum    = 0111110111101110111111111100111011110111111011011011011111010110 =  9074471352997214166
c_out = 1  c_in = 0
overflow = 0


a      = 0100000000000000000000000000000000000000000000000000000000000001 =  4611686018427387905
b      = 0100000000000000000000000000000000000000000000000000000000000001 =  4611686018427387905
sum    = 1000000000000000000000000000000000000000000000000000000000000010 = -9223372036854775806
c_out = 0  c_in = 0
overflow = 1


a      = 1011110111111111011111111001111111110111110111110111100111011111 = -4755942356444808737
b      = 1001111111011110111111111111110111110111111110111111110111110111 = -6922173373481222665
sum    = 0101110111011101111111111001110111101111110110110111011111010110 =  6768628343783520214
c_out = 1  c_in = 0
overflow = 1
```

Waveforms



SUB

Control Input for the operation : $S_1S_0 = 01$

The operations addition (ADD) and subtraction (SUB) are implemented in the same module.

The subtraction operation takes in two signed 64-bit numbers `a` and `b`, and a carry in bit `c_in` as it's inputs. To perform the subtraction operation, `c_in` is set to 1.

The module outputs a signed 64 bit number `sum` and an overflow bit `overflow`. We can represent the operation as follows

$$\mathrm{sum} = a \ + \ (-b)$$

where, $+$ refers to algebraic addition.

To obtain $-b$ from b, we need to take it's 2's complement and append 1 to it. To take the 2's complement, we flip all the bits in b to give b_{x} and add 1. Hence,

$$-b = b_{\mathrm{x}} \ + \ 1$$

where, b_{x} is the bit-flipped version of b.

Therefore, finding the difference between a and b is equivalent to,

$$\mathrm{sum} = a \ + \ (-b)$$
$$\mathrm{sum} = \boxed{a \ + \ b_{\mathrm{x}} \ + \ 1}$$

Here, we can set the carry bit c_{in} to 1, and perform regular addition on a and b_{x}, analogous to the addition operation where `c_in` is 0.

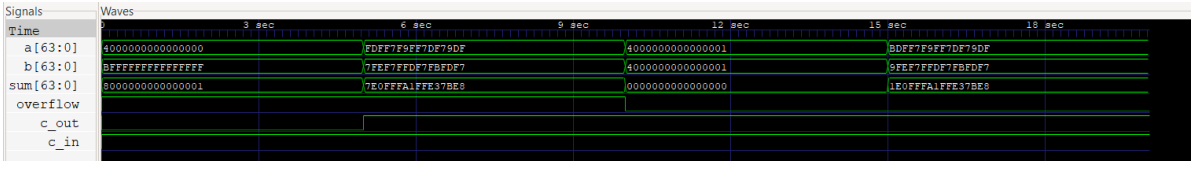
Codes

Same as ADD.

Results from test cases

```
a      = 0100000000000000000000000000000000000000000000000000000000000000 =  4611686018427387904
b      = 1011111111111111111111111111111111111111111111111111111111111111 = -4611686018427387905
sum    = 1000000000000000000000000000000000000000000000000000000000000001 = -9223372036854775807
c_out = 0  c_in = 1
overflow = 1


a      = 1111110111111111011111111001111111110111110111111011100111011111 =  -144256338017420833
b      = 0111111111101111011111111111110111110111111111011111111011110111 =  9218727691014634999
sum    = 0111111000001111111111111010000111111111111000110111101111101000 =  9083760044677495784
c_out = 1  c_in = 1
overflow = 1


a      = 0100000000000000000000000000000000000000000000000000000000000001 =  4611686018427387905
b      = 0100000000000000000000000000000000000000000000000000000000000001 =  4611686018427387905
sum    = 0000000000000000000000000000000000000000000000000000000000000000 =                     0
c_out = 1  c_in = 1
overflow = 0


a      = 1011110111111111011111111001111111110111110111111011100111011111 = -4755942356444808737
b      = 1001111111101111011111111111110111110111111111011111111011110111 = -6922173373481222665
sum    = 0001111000001111111111111010000111111111111000110111101111101000 =  2166231017036413928
c_out = 1  c_in = 1
overflow = 0
```

Waveforms



AND

Control Input for the operation : $S_1S_0 = 10$

The AND operation involves two 64-bit signed integers `a` and `b`, and performs a bitwise AND operation on the corresponding bits of `a` and `b`. The final output is a 64-bit integer `bitand`, and there is not carry generated as it's a bitwise operation that returns only a single bit.

Hence, there is no overflow generated too.

Starting from the LSB (Least Significant Bit) of `a` and `b`, we perform the AND operation on the bits using an `AND` gate via structural modelling. We loop this process over until we reach the MSB (Most Significant Bit), and then the 64-bit product has been computed.

Mathematically,

$$\mathrm{bitand} = a \ \& \ b$$

Codes

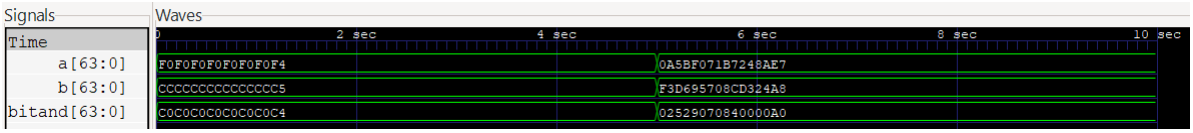
Bitwise AND module for 64-bit integers

```
module bit64and ( bitand , a , b ) ;

output signed [63:0] bitand ;
input  signed [63:0] a      ;
input  signed [63:0] b      ;

genvar i; // variable in the loop

generate

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

    begin

        // one iteration for each bit
        and Gand( bitand[i], a[i], b[i] );

    end

endgenerate

endmodule
```

Results from each case

```
a      = 1111000011110000111100001111000011110000111100001111000011110100 = -1085102592571150092
b      = 1100110011001100110011001100110011001100110011001100110011000101 = -3689348814741910331
bitand = 1100000011000000110000001100000011000000110000001100000011000100 = -4557430888798830396

a      = 0000101001011011111100000111000110110111001001001000101011100111 =   746454534454545127
b      = 1111001111010110100101010111000010001100110100110010010010101000 =  -876348766845459288
bitand = 0000001001010010100100000111000010000100000000000000000010100000 =   167354949091459232
```

Waveforms



XOR

Control Input for the operation : $S_1S_0 = 11$

The XOR operation involved two 64-bit signed integers `a` and `b` and performs a bitwise XOR operation on the corresponding bits of `a` and `b`. The final output is a 64-bit integer `bitxor`, and there is not carry generated as it's a bitwise operation that returns only a single bit.

Hence, there is no overflow generated too.

Starting from the LSB (Least Significant Bit) of `a` and `b`, we perform the XOR operation on the bits using an `XOR` gate via structural modelling. We loop this process over until we reach the MSB (Most Significant Bit), and then the 64-bit product has been computed.

Mathematically,

$$\text{bitxor} = a \oplus b$$

Codes

Bitwise XOR module for 64-bit integers

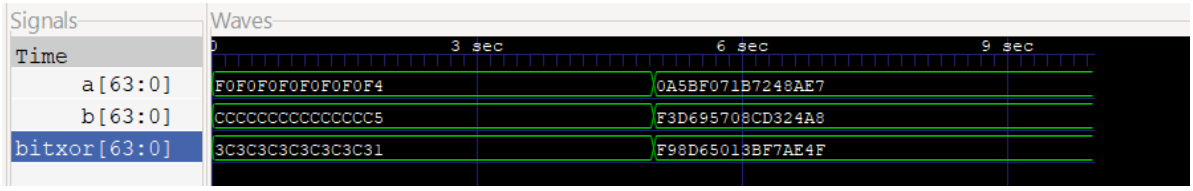
```
module bit64xor ( bitxor , a , b ) ;

output signed [63:0] bitxor ;
input  signed [63:0] a      ;
input  signed [63:0] b      ;

genvar i; // variable in the loop

generate
```

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

    begin

        // one iteration for each bit
        xor Gxor( bitxor[i], a[i], b[i] );

    end

endgenerate

endmodule
```

Results from each case

```
a      = 1111000011110000111100001111000011110000111100001111000011110100 = -1085102592571150092
b      = 1100110011001100110011001100110011001100110011001100110011000101 = -3689348814741910331
bitxor = 0011110000111100001111000011110000111100001111000011110000110001 =  4340410370284600369

a      = 0000101001011011111100000111000110110111001001001000101011100111 =   746454534454545127
b      = 1111001111010110100101010111000010001100110100110010010010101000 =  -876348766845459288
bitxor = 1111001100011010110010100000001001110111111011110101110011001111 =  -464604130573832625
```

Waveforms



Wrapper ALU Module

This module wraps up the ADD-SUB , AND , XOR modules implemented previously.

The inputs to this module are :-

1. `select_op` - 2-bit bus
2. `a` and `b` - 64-bit signed integers.

The outputs to this module are :-

1. `result` - 64-bit signed integer
2. `overflow` - a single bit

`select_op` is used to select the operation (ADD ,SUB , AND , XOR) that should be performed on `a` and `b`. There is an `overflow` bit to tell if the result is out of the bounds of what the ALU can compute.

It takes the value 1 when the `result` is not the actual outcome of the operation selected. It takes the value 0 when the `result` is the actual outcome of the operation selected.

The ALU keeps the results of all the 3 modules ready irrespective of the `select_op` values , but transfers the output of the corresponding module based on `select_op`. When $S_0 = 0$, the ADD operation is chosen and when $S_0 = 1$, the SUB operation is chosen.

It should be noted that there is no possible overflow with AND and XOR operations , but when ADD or SUB is selected , the corresponding overflow generated in that module needs to be given out as the output `overflow` in ALU module . So whenever $S_1 = 0$ (i.e , ADD or SUB is selected), `overflow`= overflow of ADD/SUB module. When $S_1 = 1$ (i.e, AND or XOR is selected) , `overflow` $= 0$. So we use a simple circuit made of a NOT and a XOR gate to do that.

Logically, overflow of an ALU can be written as

$$\mathrm{overflow} = S_1' \ \& \ \mathrm{ADDSUBOF}$$

where, $\mathrm{ADDSUBOF}$ is the overflow generated during the ADD or SUB operation.

Codes

Wrapper ALU unit for 4 operations

```
`include "../AND/bit64and.v"
`include "../XOR/bit64xor.v"
`include "../ADDSUB/bit64addsub.v"
`include "../ADDSUB/bit1addsub.v"
`include "../mux4x1.v"

module bit64ALU ( result , overflow , select_op ,  a , b ) ;

// ALU features --
// two 64-bit inputs and 2 select lines
//  a 64-bit output and a carry out line

input signed [63:0] a ;
input signed [63:0] b ;
// input sel0 , selvan ; // if two separate wires are taken
input [1:0] select_op ;     // if a single bus is used instead
```

```
output signed [63:0] result ;
output overflow ;

wire addsub_overflow; // this tells whether our results are out
of the ALU's capacity
wire c_out ;  // c_out is different from addsub_overflow , not
used if a single ALU is enough
wire select1bar;

// seperate buses to store each operation output
// AND and SUB are not needed to be done simultaneously
wire signed [63:0] op0_1 ; // The same 64 bit ripple adder does
both ADD and SUB
wire signed [63:0] op2 ;
wire signed [63:0] op3 ;

// prepare 3 outputs AND XOR ADD/SUB
// use a MUX to select one of these and transfer it to the output

// operation 0/1 ADD/SUB based on select line
// if selected operation is SUB ( i.e, op1 ), then c_in = 1 ;
bit64addsub operation0_1 ( .sum(op0_1) ,
.overflow(addsub_overflow) , .c_out(c_out) , .a(a) , .b(b) ,
.c_in(select_op[0]) ) ;

// operation2 bitwise AND operation for the 2 inputs
bit64and operation2 ( .bitand(op2), .a(a) , .b(b)) ; // no
overflow

// operation3 bitwise XOR operation for the 2 inputs
bit64xor operation3 ( .bitxor(op3), .a(a) , .b(b)) ; // no
overflow

not Gnot( select1bar, select_op[1]) ; // invert the select line
and Gand( overflow, select1bar , addsub_overflow) ; // overflow
is set if both select lines are 1

// MUX is used to select which of these should pass to 'output
result'
genvar i ;

generate

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

```
    begin

    mux4x1 outselector ( .out(result[i]) , .select(select_op),
.data0(op0_1[i]) , .data1(op0_1[i]), .data2(op2[i]) ,
.data3(op3[i]) ) ;

    end

endgenerate

endmodule
```

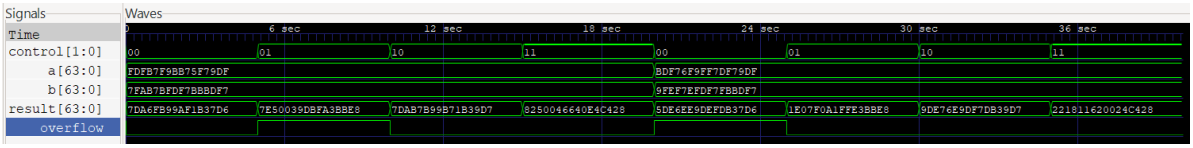
4x1 MUX to switch between 4 lines of data

```
module mux4x1 ( out , select , data0 , data1 , data2 , data3 )  ;

input [1:0] select;
input  data0 , data1 , data2 , data3 ;
output out ;

wire [1:0] selectn ;
wire [3:0] op ;

not Gs0n (selectn[0] , select[0]) ;
not Gs1n (selectn[1], select[1] ) ;

// at any instant only one of the 4 inputs connects to the output

and Gop0 ( op[0] , data0 ,selectn[0] , selectn[1] )   ;  // S0 =
0 , S1 = 0 --- S0'S1'
and Gop1 ( op[1] , data1 ,select [0] , selectn[1] )   ;  // S0 =
1 , S1 = 0 --- S0 S1'
and Gop2 ( op[2] , data2 ,selectn[0] , select [1] )   ;  // S0 =
0 , S1 = 1 --- S0'S1
and Gop3 ( op[3] , data3 ,select [0] , select [1] )   ;  // S0 =
1 , S1 = 1 --- S0 S1

// these intermediate outputs are high only when their operation
is selected

// the selected one is transferred to the final output
or Gfinalout ( out , op[0] , op[1] , op[2] , op[3]) ;

endmodule
```

Results from each case

```
control= 0
a      = 1111110111111011011111111001101110110111010111110111100111011111 =  -145382256186263073
b      = 0111111110101011011101111111101111101111011101110111011110111110111 =  9199582994547588599
result = 0111110110100110111110111001100110101111000110110011011111010110 =  9054200738361325526
overflow = 0

control= 1
a      = 1111110111111011011111111001101110110111010111110111100111011111 =  -145382256186263073
b      = 0111111110101011011101111111101111101111011101110111011110111110111 =  9199582994547588599
result = 0111111001010000000000111001110110111111101000111011101111101000 =  9101778822975699944
overflow = 1

control= 2
a      = 1111110111111011011111111001101110110111010111110111100111011111 =  -145382256186263073
b      = 0111111110101011011101111111101111101111011101110111011110111110111 =  9199582994547588599
result = 0111110110101011011101110011001101101110001101100111001110101111 =  9055467375890741719
overflow = 0

control= 3
a      = 1111110111111011011111111001101110110111010111110111100111011111 =  -145382256186263073
b      = 0111111110101011011101111111101111101111011101110111011110111110111 =  9199582994547588599
result = 1000001001010000000001000110011001000000111001001100010000101000 = -9056734013420157912
overflow = 0

control= 0
a      = 1011110111110111011011111001111111110111110111110111100111011111 = -4758211748444538401
b      = 1001111111101111011111101111110111110111111111011101111011111110111 = -6922174472992866825
result = 0101110111100110111011101001110111101111110110110011011111010110 =  6766357852272146390
overflow = 1

control= 1
a      = 1011110111110111011011111001111111110111110111110111100111011111 = -4758211748444538401
b      = 1001111111101111011111101111110111110111111111011101111011111110111 = -6922174472992866825
result = 0001111000000111111100001010000111111111111000111011101111101000 =  2163962724548328424
overflow = 0

control= 2
a      = 1011110111110111011011111001111111110111110111110111100111011111 = -4758211748444538401
b      = 1001111111101111011111101111110111110111111111011101111011111110111 = -6922174472992866825
result = 1001110111100111011011101001110111111011111011011001110011101111 = -7068559465387443753
overflow = 0

control= 3
a      = 1011110111110111011011111001111111110111110111110111100111011111 = -4758211748444538401
b      = 1001111111101111011111101111110111110111111111011101111011111110111 = -6922174472992866825
result = 0010001000011000000100010110001000000000001001001100010000101000 =  2456732709337482280
overflow = 0
```

Waveforms



Sequential Design

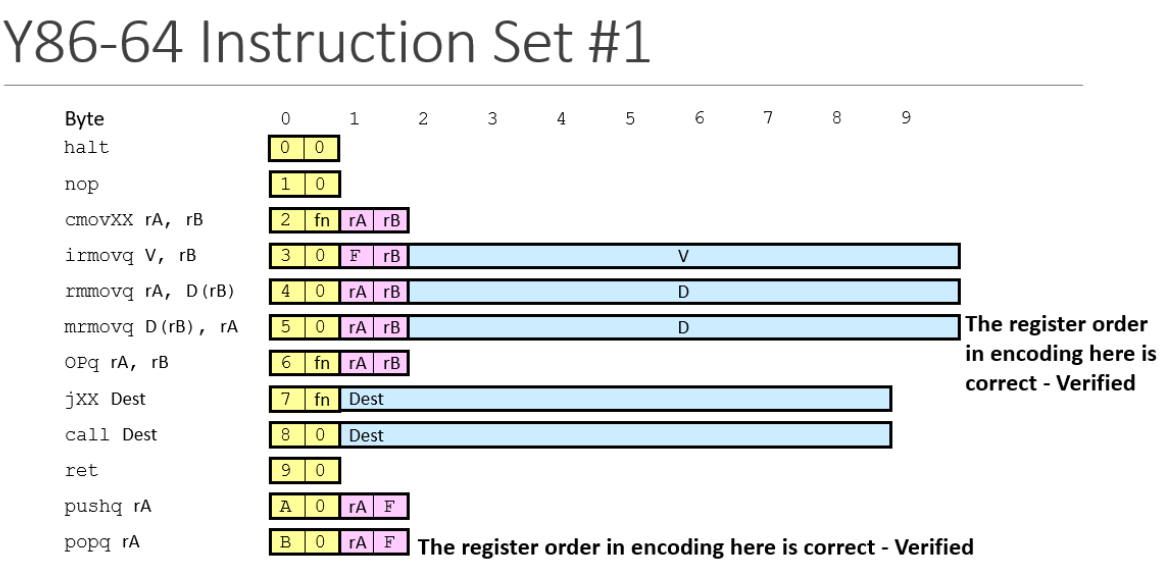
We have implemented a sequentially designed `y86` processor in Verilog, that is built upon the 5 stage modular design methodology.

The processor supports the following class of commands:

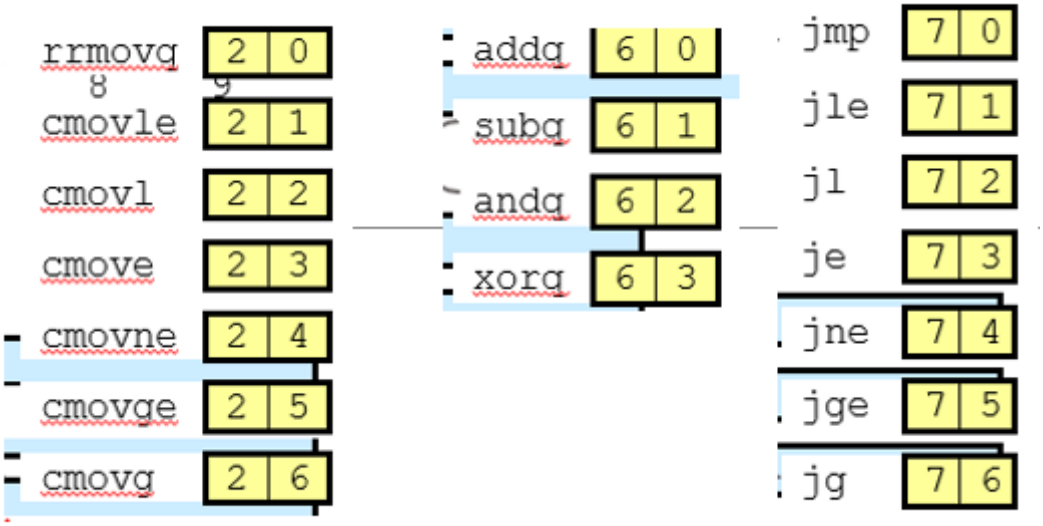
1. `ihalt`

2. `inop`
3. `icmovxx`
4. `iirmovq`
5. `irmmovq`
6. `imrmovq`
7. `iOPq`
8. `ijXX`
9. `icall`
10. `iret`
11. `ipushq`
12. `ipopq`

The above commands along with their instruction data are described below:



The commands `cmovxx` , `OPq` and `jxx` are elaborated as follows, given along with their `icode` and `ifun` value.



The processor consists of the following stages, described as below:

1. Fetch

The `split` register checks the first byte from the instruction pointer by the Program Counter, and stores it, to check the class of instruction that is read. Splits into `icode` and `ifun`.

The `align` register describes how the processor extracts the remaining fields from an instruction, depending on whether or not the instruction has a register specifier byte.

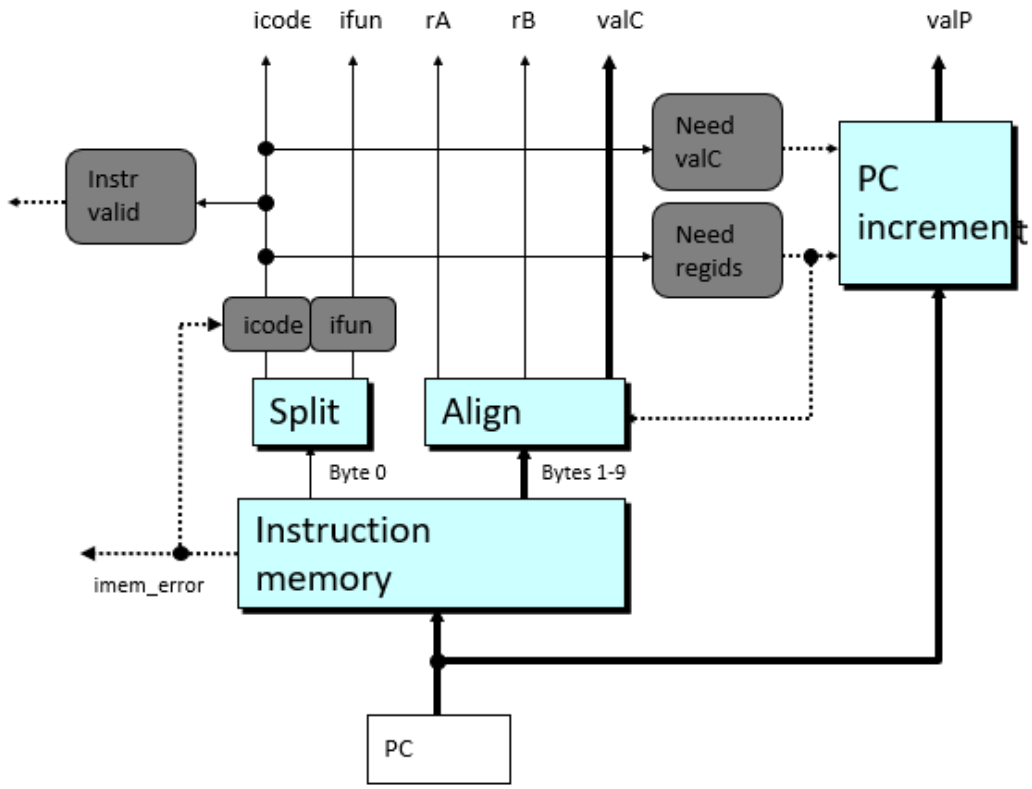
`valP` denotes by how much the program counter must be incremented, and is computed based on the class of instruction.

`instr_valid` determines if the read instruction is valid or not.

`needs_regids` and `needs_valC` determines the amount by which the program counter has to be incremented.

This stage connects to take input from the instruction memory and connects to give its output to the decode stage.

We have implemented the fetch stage in 2 parts, one component of which is instruction memory, and the other of which take input from instruction memory to fetch the task.



2. Decode

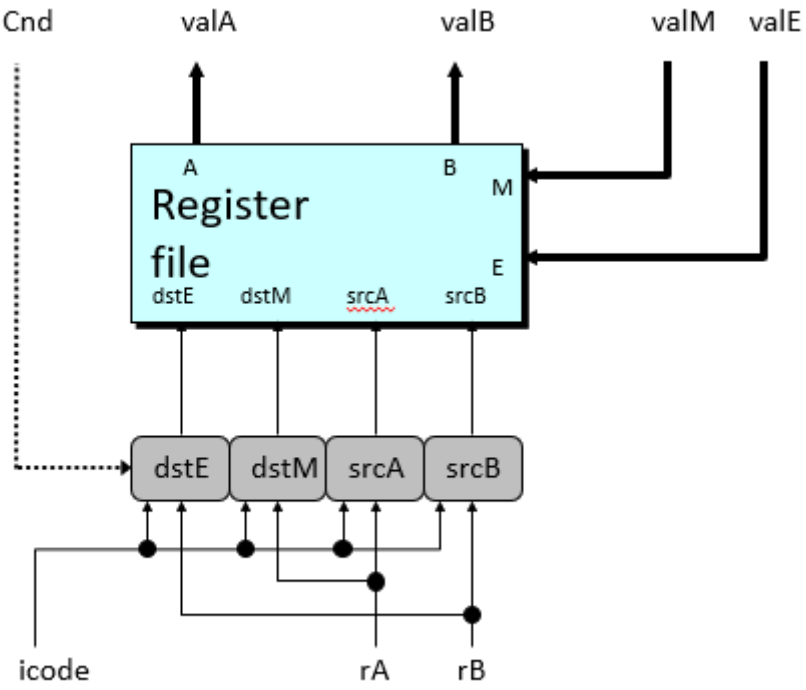
The decode stage has been implemented in 2 parts, one of which is `decodetask` that sends inputs to `reg file` , based on inputs from the previous stage (Fetch).

`srcA` and `srcB` determine the source register for `valA` and `valB`, that are inputs to the next stages.

`dstE` and `dstM` determine the destination register for instructions that use writeback.

`regfile` is the second component of decode stage, that has information regarding the 16 registers.

The decode stage works in co-ordination with the writeback stage, which is built into `regfile.v`. The `regfile.v` file additionally writes the current register status after each instruction has been executed into a text file called `reg_as_it_is.txt`.



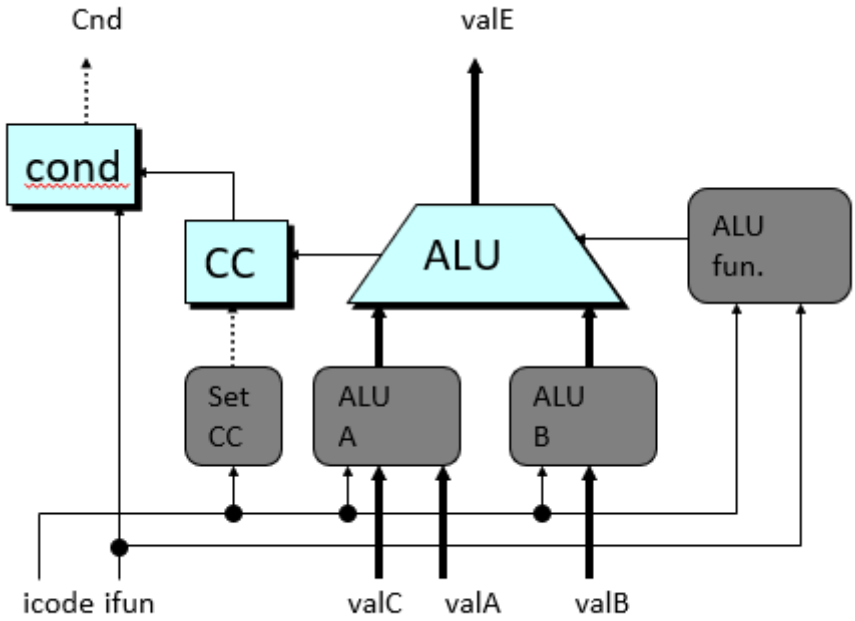
3. Execute

The execute stage uses an ALU that we have built in the previous assignment, working to take input from decode stage, and performs the execution of the instructions read from the previous stage.

The operation to be done in the ALU depends on the parameter `select_op`, that does `ADD`, `SUB`, `AND` or `XOR` depending on the `ifun` value.

`setCC` allows an instruction to set conditional codes.

The execute stage connects to the memory stage, that determines where to store the computed values in memory. It also connects to register files if we need to store the values in register, depending on the class of instructions.



4. Memory

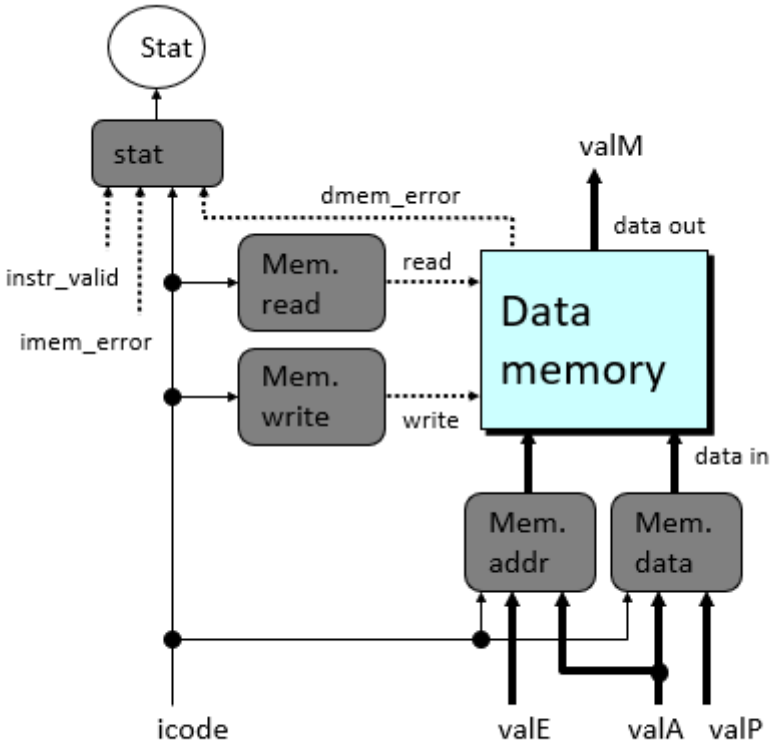
Memory stage has been implemented in 2 modules, `memtask` and `datamem`

`memtask` gives the memory address and some data of memory to store in that address.

`datamem` is the memory block that takes care of the storage based on the orders given by `memtask`, based on the parameters `read_enable` and `write_enable`, that decide whether data has to be read from or written to the memory block.

The memory block connects to writeback that has been previously implemented, and also connects to the final stage.

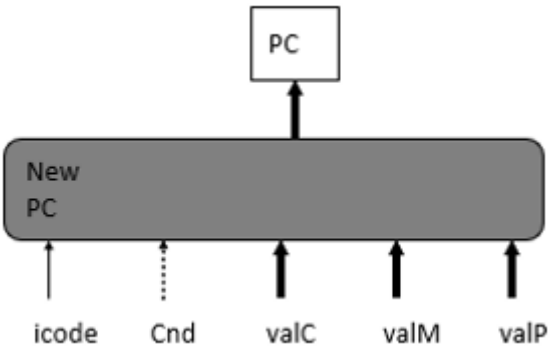
The current status of memory is also written into a text file called `mem_as_it_is.txt` everytime an instruction has been executed that needs memory access.



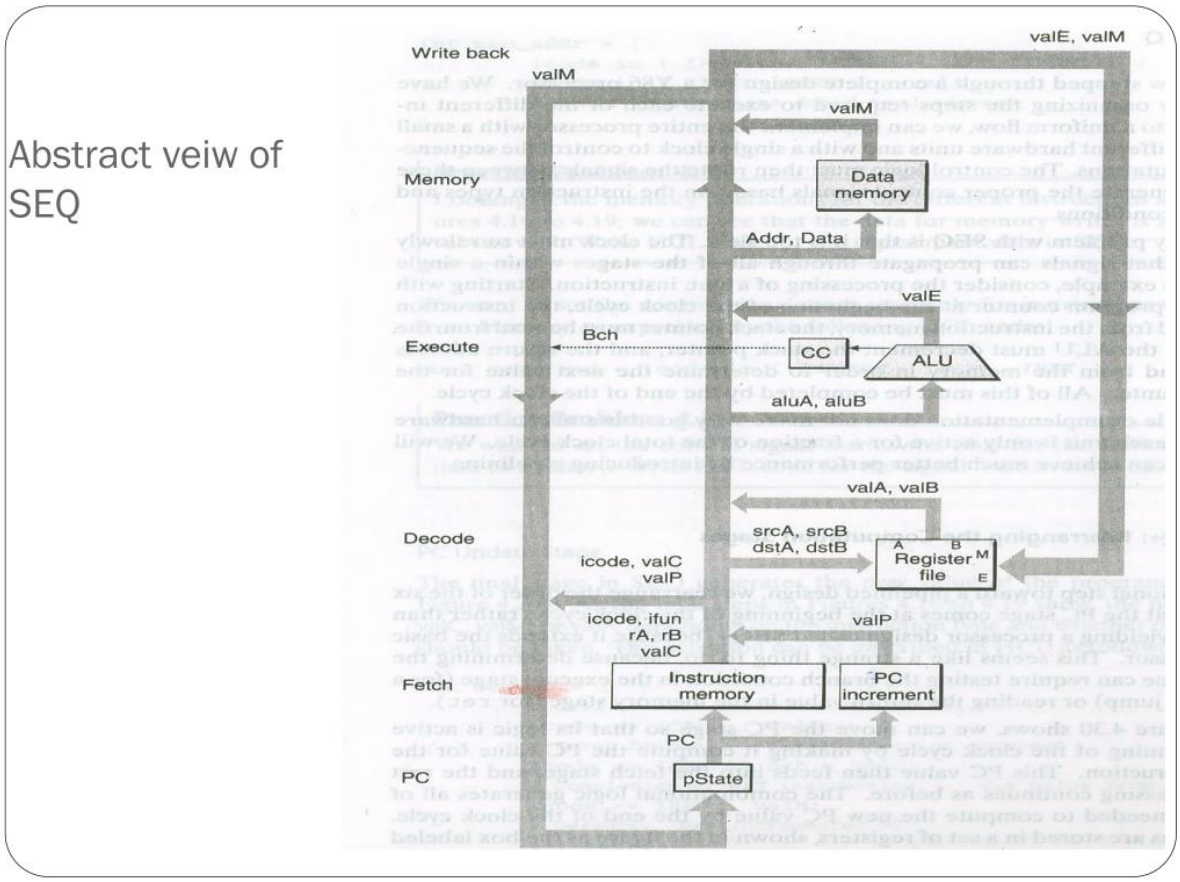
5. Writeback and PC Update

The writeback stage simply alters the values in the register if needed based on `icode`.

PC Update increments or decrements the program counter to point it to the next instruction, based on the instruction read previously and the intermediate values generated such as `valP`, `valC`, `valM` and `valE`. This is to get the processor ready for the next instruction that has to be read.



6. Overall Implementation



7. Testing

The processor can be run by `iverilog y86SEQ.v test86.v -grelative-include` , followed by `vvp ./a.out` in the `seq` directory (currently supported only on Linux due to iverilog relative file path issues on Windows).

To test the processor, we have written some memory encoded instructions in a file called `opcodes.txt` that is read by the instruction memory of the processor, and aided by a testbench that controls the clock frequency of the processor.

The testbench monitors the staus of the processor every clock cycle and displays the critical values that change every clock cycle, so as to keep track of the running of the processor.

The output of the processor is as follows:

```
in instruction_memory.v,  split = 30  align = 0000000000000020f4
for received  pc = 0000000000000000
in fetchsuper.v   instr_valid = 1 , need_regids = 1 , need_valC =
1 , and valP = 000000000000000a , icode = 3 , ifun = 0, rA = f ,
rB = 4 , valC = 0000000000000020 for recevied  split = 30 and
align = 0000000000000020f4
in decodetask.v   srcA = f , srcB = f dstE = 4 , dstM = f for
received rA = f , rB = 4 , icode = 3 ,cnd =1
```

```
in regfile.v      valA = 0000000000000000 , valB =
0000000000000000
in executesuper.v ALUa = 0000000000000020 , ALUb =
0000000000000000 , select_op = 0 ,  valE = 0000000000000020
hello status = x
in memtask.v     mem_data = 000000000000000a  , valA =
0000000000000000
clk = 0

WARNING: ././2Decode/regfile.v:44: $writememh: Standard
inconsistency, following 1364-2005.
value 0000000000000020 is being inserted into register 4
hello status = 0
in memtask.v     mem_data = 000000000000000a  , valA =
0000000000000000
clk = 0

WARNING: ././2Decode/regfile.v:44: $writememh: Standard
inconsistency, following 1364-2005.
value 0000000000000020 is being inserted into register 4
in instruction_memory.v,  split = 30  align = 00000000000000f3
for received  pc = 000000000000000a
in fetchsuper.v   instr_valid = 1 , need_regids = 1 , need_valC =
1 , and valP = 0000000000000014 , icode = 3 , ifun = 0, rA = f ,
rB = 3 , valC = 0000000000000007 for recevied  split = 30 and
align = 00000000000000f3
in decodetask.v   srcA = f , srcB = f dstE = 3 , dstM = f for
received rA = f , rB = 3 , icode = 3 ,cnd =1
in executesuper.v ALUa = 0000000000000007 , ALUb =
0000000000000000 , select_op = 0 ,  valE = 0000000000000007
hello status = 0
in memtask.v     mem_data = 0000000000000014  , valA =
0000000000000000
clk = 0

WARNING: ././2Decode/regfile.v:44: $writememh: Standard
inconsistency, following 1364-2005.
value 0000000000000007 is being inserted into register 3
in instruction_memory.v,  split = 30  align = 00000000000005f8
for received  pc = 0000000000000014
in fetchsuper.v   instr_valid = 1 , need_regids = 1 , need_valC =
1 , and valP = 000000000000001e , icode = 3 , ifun = 0, rA = f ,
rB = 8 , valC = 0000000000000005 for recevied  split = 30 and
align = 00000000000005f8
```

```
in decodetask.v   srcA = f , srcB = f dstE = 8 , dstM = f for
received rA = f , rB = 8 , icode = 3 ,cnd =1
in executesuper.v ALUa = 0000000000000005 , ALUb =
0000000000000000 , select_op = 0 ,  valE = 0000000000000005
hello status = 0
in memtask.v     mem_data = 000000000000001e  , valA =
0000000000000000
clk = 0

WARNING: ./2Decode/regfile.v:44: $writememh: Standard
inconsistency, following 1364-2005.
value 0000000000000005 is being inserted into register 8
in instruction_memory.v,  split = 60  align = 000000000007834038
for received  pc = 000000000000001e
in fetchsuper.v   instr_valid = 1 , need_regids = 1 , need_valC =
0 , and valP = 0000000000000020 , icode = 6 , ifun = 0, rA = 3 ,
rB = 8 , valC = 0000000000000000 for recevied  split = 60 and
align = 000000000007834038
in decodetask.v   srcA = 3 , srcB = 8 dstE = 8 , dstM = f for
received rA = 3 , rB = 8 , icode = 6 ,cnd =1
in regfile.v      valA = 0000000000000007 , valB =
0000000000000005
conditions are being set .... OF = 0 , SF = 0 , ZF = 0
hello status = 0
in memtask.v     mem_data = 0000000000000020  , valA =
0000000000000007
clk = 0

WARNING: ./2Decode/regfile.v:44: $writememh: Standard
inconsistency, following 1364-2005.
value 000000000000000c is being inserted into register 8
in instruction_memory.v,  split = 40  align = 0000000000000000783
for received  pc = 0000000000000020
in fetchsuper.v   instr_valid = 1 , need_regids = 1 , need_valC =
1 , and valP = 000000000000002a , icode = 4 , ifun = 0, rA = 8 ,
rB = 3 , valC = 0000000000000007 for recevied  split = 40 and
align = 0000000000000000783
in decodetask.v   srcA = 8 , srcB = 3 dstE = f , dstM = f for
received rA = 8 , rB = 3 , icode = 4 ,cnd =1
in regfile.v      valA = 000000000000000c , valB =
0000000000000007
in executesuper.v ALUa = 0000000000000007 , ALUb =
0000000000000007 , select_op = 0 ,  valE = 000000000000000e
hello status = 0
```

```
WARNING: ././4Memory/datamem.v:37: $writememh: Standard
inconsistency, following 1364-2005.
writing..... wrote 000000000000000c into memory location
000000000000000e
in memtask.v     mem_data = 000000000000000c  , valA =
000000000000000c
clk = 0

in instruction_memory.v,  split = 40  align = 0000000000000000038
for received  pc = 000000000000002a
in fetchsuper.v   instr_valid = 1 , need_regids = 1 , need_valC =
1 , and valP = 0000000000000034 , icode = 4 , ifun = 0, rA = 3 ,
rB = 8 , valC = 0000000000000000 for recevied  split = 40 and
align = 0000000000000000038
in decodetask.v   srcA = 3 , srcB = 8 dstE = f , dstM = f for
received rA = 3 , rB = 8 , icode = 4 ,cnd =1
in regfile.v      valA = 0000000000000007 , valB =
000000000000000c
in executesuper.v ALUa = 0000000000000000 , ALUb =
000000000000000c , select_op = 0 ,  valE = 000000000000000c
hello status = 0
WARNING: ././4Memory/datamem.v:37: $writememh: Standard
inconsistency, following 1364-2005.
writing..... wrote 0000000000000007 into memory location
000000000000000c
in memtask.v     mem_data = 0000000000000007  , valA =
0000000000000007
clk = 0

in instruction_memory.v,  split = 50  align = 00000000000000000a8
for received  pc = 0000000000000034
in fetchsuper.v   instr_valid = 1 , need_regids = 1 , need_valC =
1 , and valP = 000000000000003e , icode = 5 , ifun = 0, rA = a ,
rB = 8 , valC = 0000000000000000 for recevied  split = 50 and
align = 00000000000000000a8
in decodetask.v   srcA = f , srcB = 8 dstE = f , dstM = a for
received rA = a , rB = 8 , icode = 5 ,cnd =1
in regfile.v      valA = 0000000000000000 , valB =
000000000000000c
in executesuper.v ALUa = 0000000000000000 , ALUb =
000000000000000c , select_op = 0 ,  valE = 000000000000000c
hello status = 0
reading..... got 0000000000000007 from memory location
000000000000000c
```

```
in memtask.v     mem_data = 000000000000003e  , valA =
0000000000000000
clk = 0

WARNING: ././2Decode/regfile.v:56: $writememh: Standard
inconsistency, following 1364-2005.
value 0000000000000007 is being inserted into register a
in instruction_memory.v,  split = 10  align = 000000000000006180
for received  pc = 000000000000003e
in fetchsuper.v   instr_valid = 1 , need_regids = 0 , need_valC =
0 , and valP = 000000000000003f , icode = 1 , ifun = 0, rA = f ,
rB = f , valC = 0000000000000000 for recevied  split = 10 and
align = 000000000000006180
in decodetask.v   srcA = f , srcB = f dstE = f , dstM = f for
received rA = f , rB = f , icode = 1 ,cnd =1
in regfile.v      valA = 0000000000000000 , valB =
0000000000000000
in executesuper.v ALUa = 0000000000000008 , ALUb =
0000000000000000 , select_op = 1 ,  valE = fffffffffffffff8
hello status = 0
in memtask.v     mem_data = 000000000000003f  , valA =
0000000000000000
clk = 0

in instruction_memory.v,  split = 80  align = 0000000000000000061
for received  pc = 000000000000003f
in fetchsuper.v   instr_valid = 1 , need_regids = 0 , need_valC =
1 , and valP = 0000000000000048 , icode = 8 , ifun = 0, rA = f ,
rB = f , valC = 0000000000000061 for recevied  split = 80 and
align = 0000000000000000061
in decodetask.v   srcA = f , srcB = 4 dstE = 4 , dstM = f for
received rA = f , rB = f , icode = 8 ,cnd =1
in regfile.v      valA = 0000000000000000 , valB =
0000000000000020
in executesuper.v ALUa = 0000000000000008 , ALUb =
0000000000000020 , select_op = 1 ,  valE = 0000000000000018
hello status = 0
WARNING: ././4Memory/datamem.v:37: $writememh: Standard
inconsistency, following 1364-2005.
writing..... wrote 0000000000000048 into memory location
0000000000000018
in memtask.v     mem_data = 0000000000000048  , valA =
0000000000000000
clk = 0
```

```
WARNING: ././2Decode/regfile.v:44: $writememh: Standard
inconsistency, following 1364-2005.
value 0000000000000018 is being inserted into register 4
in instruction_memory.v,  split = 30  align = 00000000000007fc
for received  pc = 0000000000000061
in fetchsuper.v   instr_valid = 1 , need_regids = 1 , need_valC =
1 , and valP = 000000000000006b , icode = 3 , ifun = 0, rA = f ,
rB = c , valC = 0000000000000007 for recevied  split = 30 and
align = 00000000000007fc
in decodetask.v   srcA = f , srcB = f dstE = c , dstM = f for
received rA = f , rB = c , icode = 3 ,cnd =1
in regfile.v      valA = 0000000000000000 , valB =
0000000000000000
in executesuper.v ALUa = 0000000000000007 , ALUb =
0000000000000000 , select_op = 0 ,  valE = 0000000000000007
hello status = 0
in memtask.v     mem_data = 000000000000006b  , valA =
0000000000000000
clk = 0

WARNING: ././2Decode/regfile.v:44: $writememh: Standard
inconsistency, following 1364-2005.
value 0000000000000007 is being inserted into register c
in instruction_memory.v,  split = 30  align = 00000000000005fd
for received  pc = 000000000000006b
in fetchsuper.v   instr_valid = 1 , need_regids = 1 , need_valC =
1 , and valP = 0000000000000075 , icode = 3 , ifun = 0, rA = f ,
rB = d , valC = 0000000000000005 for recevied  split = 30 and
align = 00000000000005fd
in decodetask.v   srcA = f , srcB = f dstE = d , dstM = f for
received rA = f , rB = d , icode = 3 ,cnd =1
in executesuper.v ALUa = 0000000000000005 , ALUb =
0000000000000000 , select_op = 0 ,  valE = 0000000000000005
hello status = 0
in memtask.v     mem_data = 0000000000000075  , valA =
0000000000000000
clk = 0

WARNING: ././2Decode/regfile.v:44: $writememh: Standard
inconsistency, following 1364-2005.
value 0000000000000005 is being inserted into register d
in instruction_memory.v,  split = 60  align = 1010101010101090cd
for received  pc = 0000000000000075
```

```
in fetchsuper.v   instr_valid = 1 , need_regids = 1 , need_valC =
0 , and valP = 0000000000000077 , icode = 6 , ifun = 0, rA = c ,
rB = d , valC = 0000000000000000 for recevied  split = 60 and
align = 10101010101090cd
in decodetask.v   srcA = c , srcB = d dstE = d , dstM = f for
received rA = c , rB = d , icode = 6 ,cnd =1
in regfile.v      valA = 0000000000000007 , valB =
0000000000000005
conditions are being set .... OF = 0 , SF = 0 , ZF = 0
hello status = 0
in memtask.v     mem_data = 0000000000000077  , valA =
0000000000000007
clk = 0

WARNING: ./2Decode/regfile.v:44: $writememh: Standard
inconsistency, following 1364-2005.
value 000000000000000c is being inserted into register d
in instruction_memory.v,  split = 90  align = 101010101010101010
for received  pc = 0000000000000077
in fetchsuper.v   instr_valid = 1 , need_regids = 0 , need_valC =
0 , and valP = 0000000000000078 , icode = 9 , ifun = 0, rA = f ,
rB = f , valC = 0000000000000000 for recevied  split = 90 and
align = 101010101010101010
in decodetask.v   srcA = 4 , srcB = 4 dstE = 4 , dstM = f for
received rA = f , rB = f , icode = 9 ,cnd =1
in regfile.v      valA = 0000000000000018 , valB =
0000000000000018
in executesuper.v ALUa = 0000000000000008 , ALUb =
0000000000000018 , select_op = 0 ,  valE = 0000000000000020
hello status = 0
reading..... got 0000000000000048 from memory location
0000000000000018
in memtask.v     mem_data = 0000000000000078  , valA =
0000000000000018
clk = 0

WARNING: ./2Decode/regfile.v:44: $writememh: Standard
inconsistency, following 1364-2005.
value 0000000000000020 is being inserted into register 4
in instruction_memory.v,  split = 00  align = 101010101010101010
for received  pc = 0000000000000048
in fetchsuper.v   instr_valid = 1 , need_regids = 0 , need_valC =
0 , and valP = 0000000000000049 , icode = 0 , ifun = 0, rA = f ,
rB = f , valC = 0000000000000000 for recevied  split = 00 and
align = 101010101010101010
```

```
in decodetask.v   srcA = f , srcB = f dstE = f , dstM = f for
received rA = f , rB = f , icode = 0 ,cnd =1
in regfile.v      valA = 0000000000000000 , valB =
0000000000000000
in executesuper.v ALUa = 0000000000000008 , ALUb =
0000000000000000 , select_op = 1 ,  valE = fffffffffffffff8
hello status = 1
in memtask.v     mem_data = 0000000000000049  , valA =
0000000000000000
```

As observed, the processor runs through the instructions and correctly updates the program counter for the next instruction. We can also observe that values are successfully being written across memory and registers, as shown below

Register status :

```
// 0x00000000 - from %rax to %r15
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
0000000000000007
0000000000000020
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
000000000000000c
xxxxxxxxxxxxxxxx
0000000000000007
xxxxxxxxxxxxxxxx
0000000000000007
000000000000000c
xxxxxxxxxxxxxxxx
```

Memory status :

```
// 0x00000000
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
```

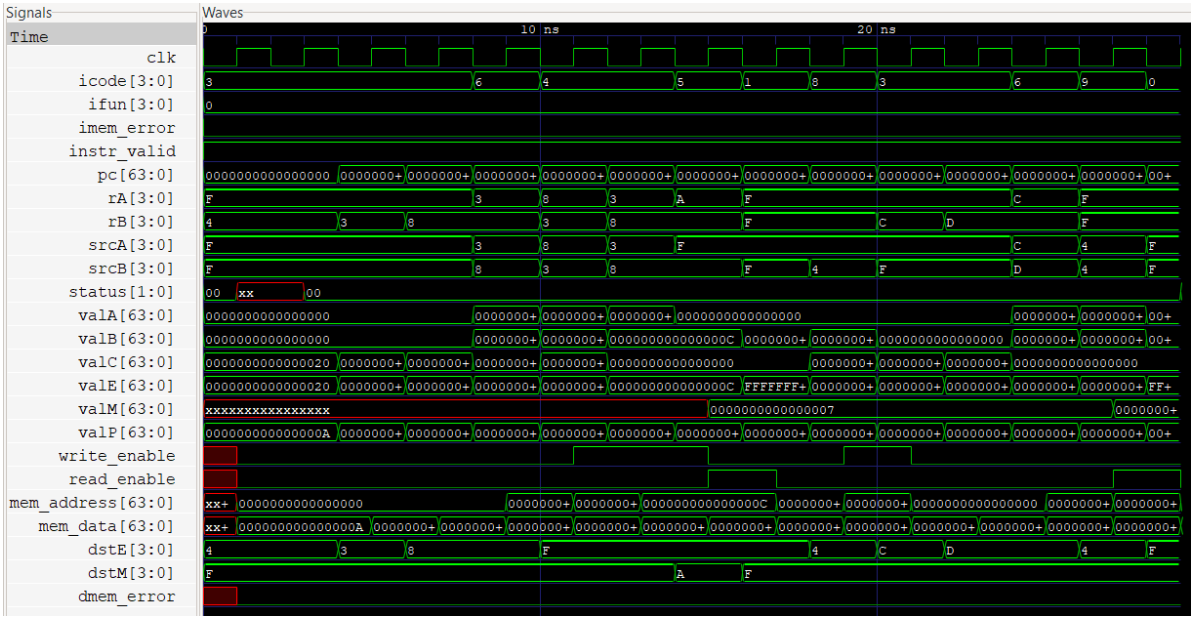
```
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
0000000000000007
xxxxxxxxxxxxxxxx
000000000000000c
xxxxxxxxxxxxxxxx
// 0x00000010
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
0000000000000048
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
```

The rest of the file has been omitted as it is untouched and is too lengthy.

As seen from the terminal output logs, `call` (`icode = 8`) has been used to move to a place, execute 3 commands, and then `ret` (`icode = 9`) has been used to come back to the destination.

Hence, we have verified that the `call` and `ret` functions are working.

Running GTKwave on the processor logs, we see:



Also, the time scale has been increased by 10^9 times for better clarity, which can be set to nanoseconds if wanted.

The processor takes 2 nanoseconds to complete 1 clock cycle, hence the frequency of the processor is

$$f = \frac{1}{t} = \frac{1}{2 * 10^{-9}\text{s}} = 500\text{MHz}$$

Pipelining

We have implemented a 5-stage piplined `y86` processor in Verilog, that is built upon the 5 stage modular design methodology.

The processor supports the following class of commands:

1. `ihalt`
2. `inop`
3. `icmovxx`
4. `iirmovq`
5. `irmmovq`
6. `imrmovq`
7. `iOPq`
8. `ijXX`
9. `icall`
10. `iret`
11. `ipushq`
12. `ipopq`

To begin with implementing a pipelined processor, some changes have to be made to the existing SEQ design.

1. Rearranging PC Update

The PC Update stage goes from being last in the SEQ design to the first in the pipelined design as the next instruction is fetched by the processor without waiting for the first instruction to finish executing, and hence sends updated PC value to the fetch stage from all the different stages in the processor cycle, leaving it up to the other stages to decode and process the information to avoid hazards.

2. Pipeline Registers

In a pipelined processor design, we implement registers before each stage of the processor cycle, and act as a temporary holding buffer for incoming instructions in the cycle, also preventing overflow of instructions into the next stage.

There are 5 registers that are inserted:

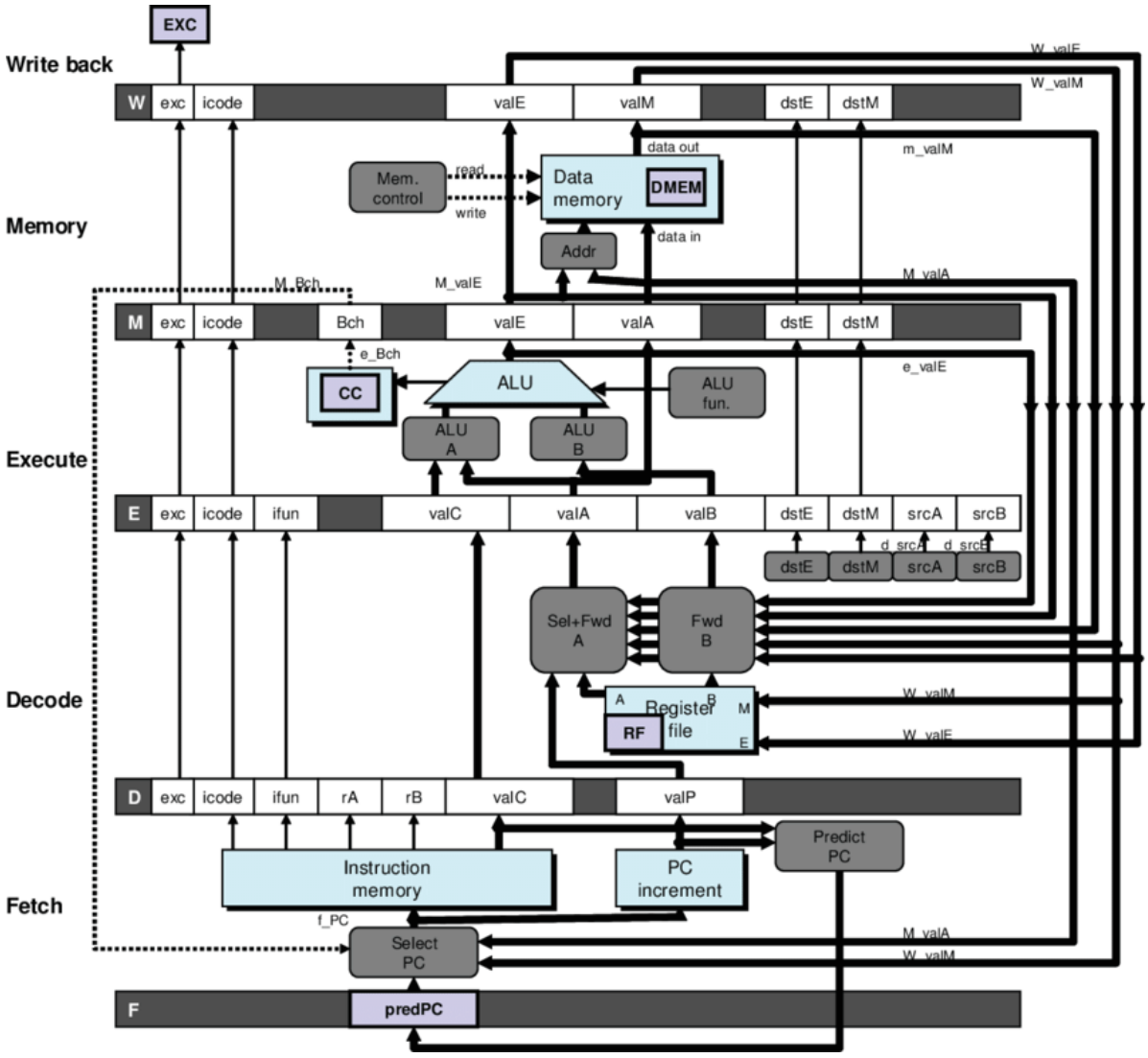
- `F` register : Implemented before the fetch stage holding the value of the predicted program counter.
- `D` register : Implemented between the fetch and decode stage, connected to the fetch stage behind, and holds recently fetched instructions and stores them intermediately before passing them onto the decode stage.
- `E` register : Implemented between the decode and execute stage, connected to the decode stage behind, and holds information about recently decoded instructions from the register file and stores them intermediately before passing them onto the execute stage.
- `M` register : Implemented between the execute and memory stage, connected to the memory stage behind, and holds information about recently executed instructions, and also holds information about branch conditions and branch targets.
- `W` register : Connects to the memory stage and the feedback path to the regfile.

3. Rearranging signals

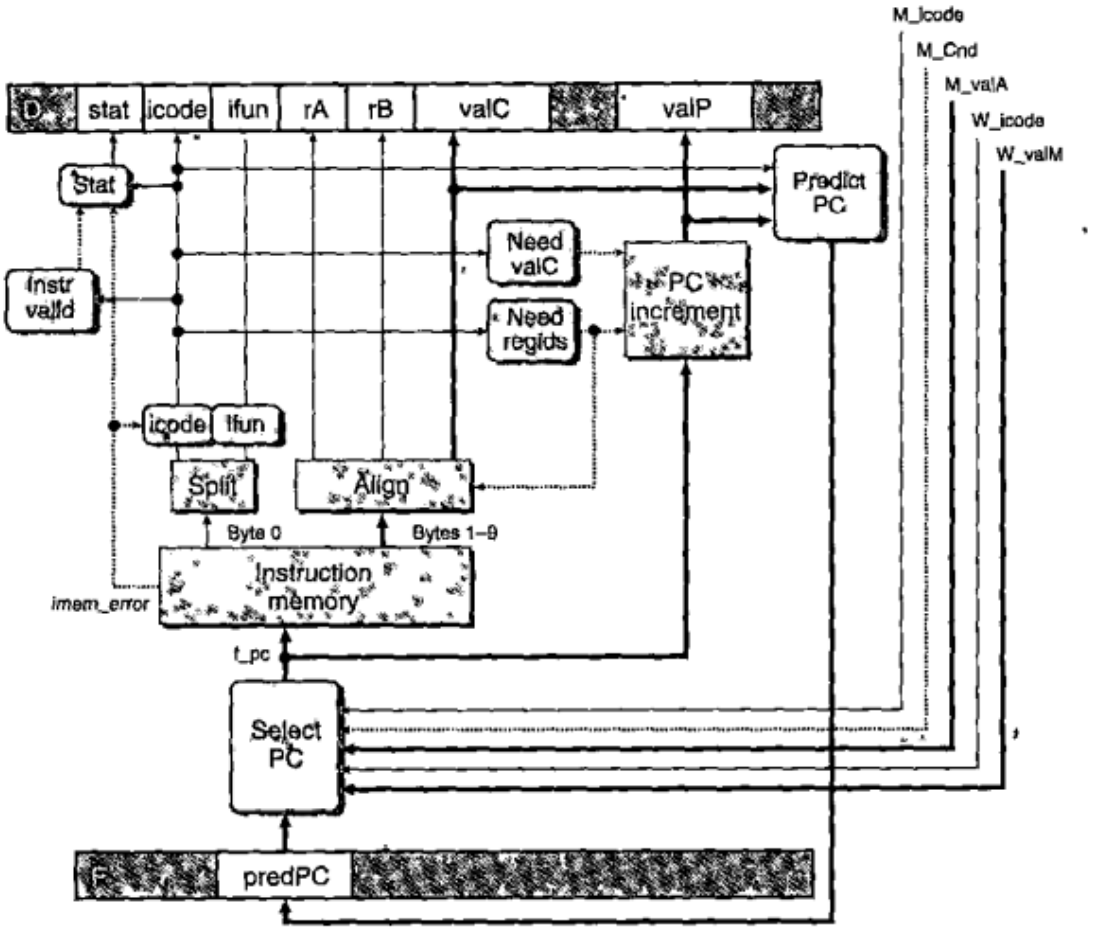
The signals are approprialtely renamed and altered and computed upon as it creates confusion because there can be multiple parameters during the same clock cycle across different stages in the processor execution, all referencing the same variable.

Here is a look at the different stages with hardware implementation:

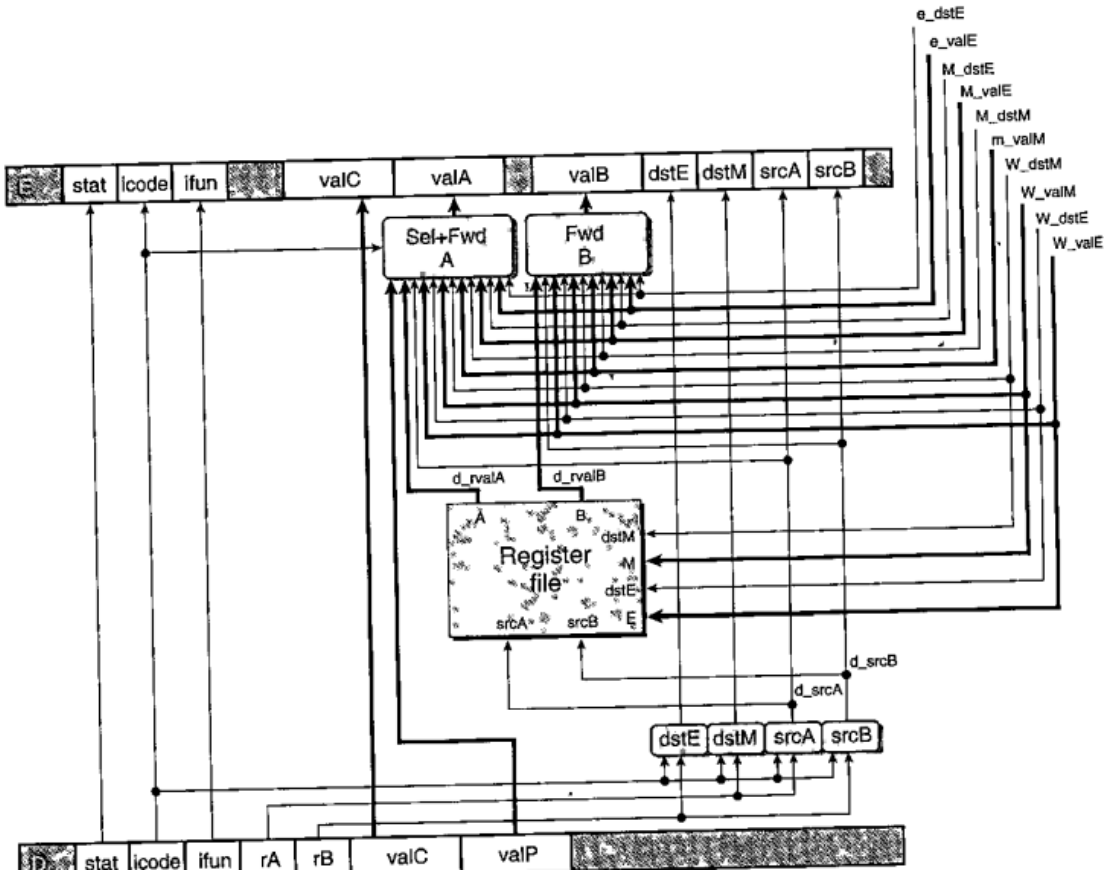
- Pipelined 5 stage processor



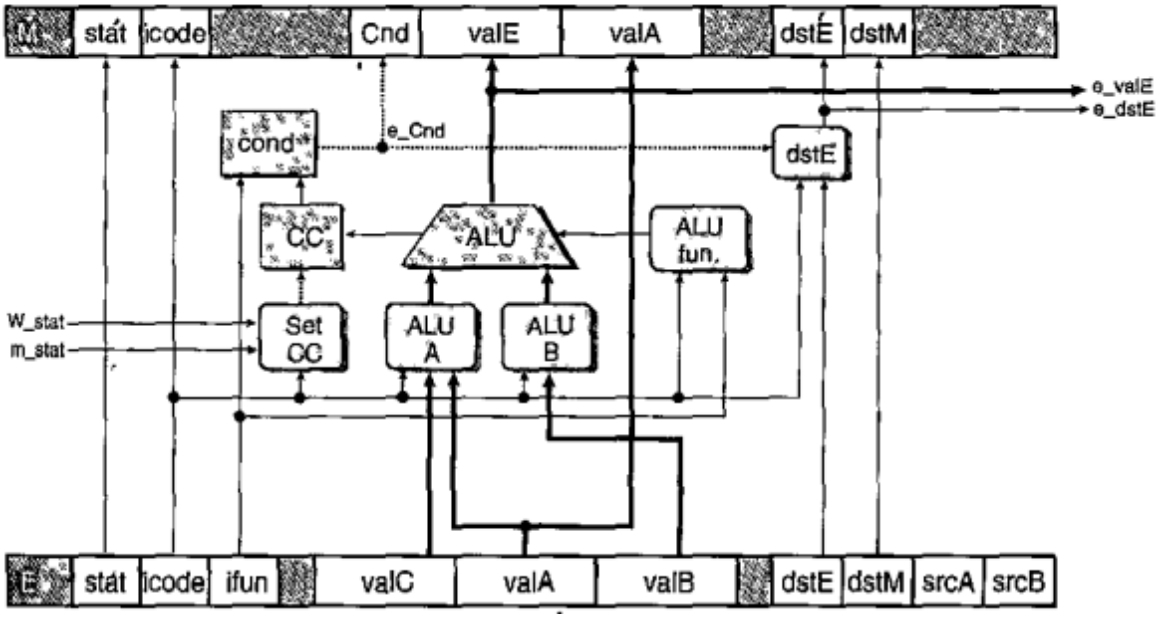
- Fetch stage



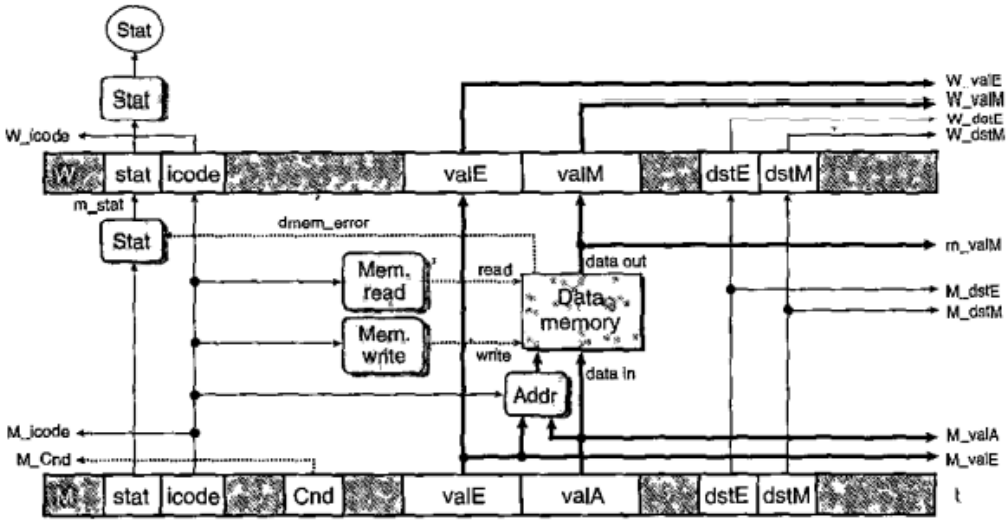
- Decode stage



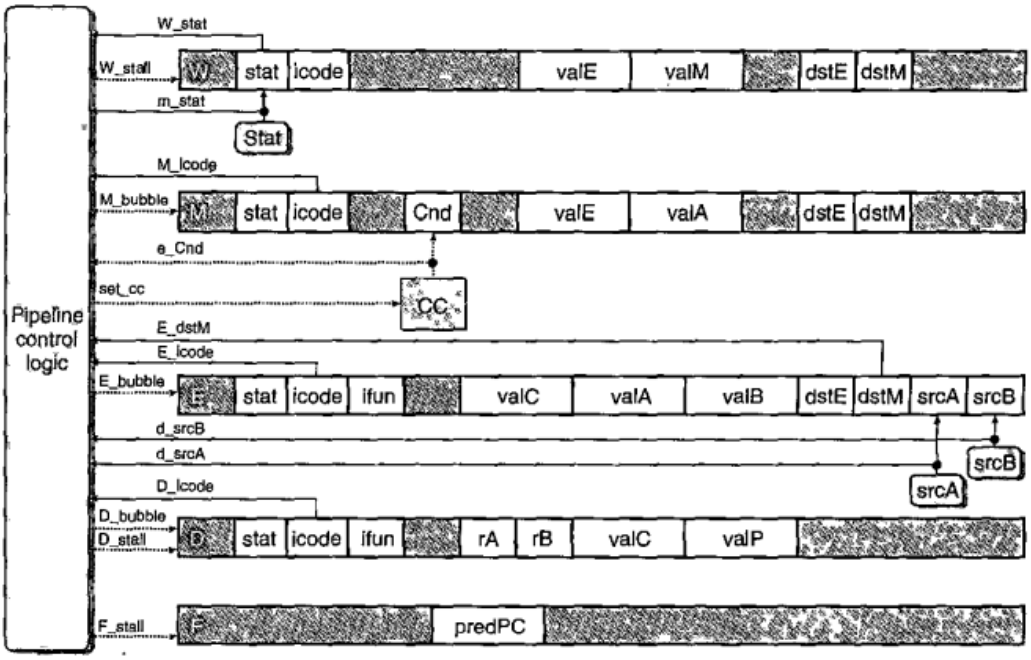
- Execute stage



- Memory stage



- Pipeline control logic



- Pipeline control actions

	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

The above hazard clearance conditions have been taken care of and implemented in files for each stage in the `stagereg` directory.

The files can be run by `iverilog test86.v yo86pipe.v -grelative-include` (currently supported only on Linux).

Design Challenges encountered

It was pretty difficult to implement the call and ret instructions, and to synchronize the regfile and writeback stage operations with the register and memory status files. It was also pretty difficult to implement the pipelined registers due to the high amount of parameters to take in mind.