Intro To Processor Architecture

Sreenya Chitluri - 2020102065
Smruti Biswal - 2020112011

Y-86-64 Processor Design

Project Report

Contents

Introduction

The Y-86-64 processor is a simplified version or the toy version of the X-86-64 processors which is the widely used processor of today. It runs for numbers and data which occupies 64 bits. Generally, these processors are backward compatible. The Y-86 processor lies somewhat in the middle of CISC and RISC architecture wherein it implements almost all the instructions of a real processor but it is much more simplified. Here, in our project we have implemented a -

1. ALU block
2. Sequential implementation
3. Pipelined implementation

These implementations have been done on Verilog which is a HDL. The modular design approach followed tries to take its goals from the Y-86 ISA.

ALU and its working

Addition

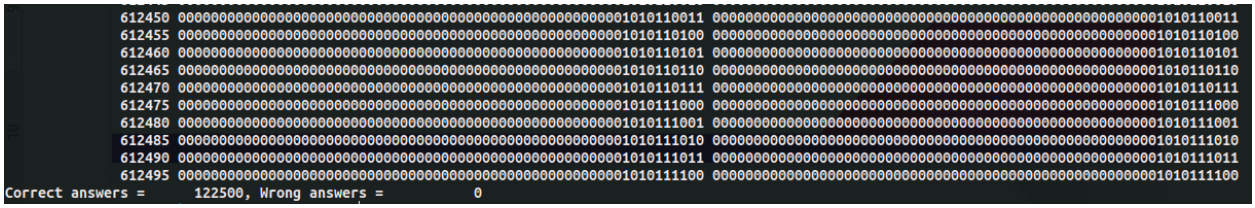
It uses a 4-bit adder which is built with the **carry-lookahead strategy.** This 4-bit adder is then extended to a 64 bit adder by calling the module 16 times. To correctly compute the overflow variable, we have updated the 4-bit adder module for the last 4 bits so that we get values like C3 and Cout which we need for the overflow contribution.

```
xor over(overflow, C3, Cout);
```

This is a part of the module named `module finaladder(S, A, B, Cin, overflow);`

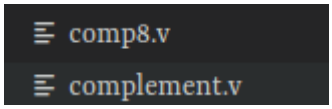
which helps get the correct value of the variable ‘overflow’.

Results obtained -



Subtraction

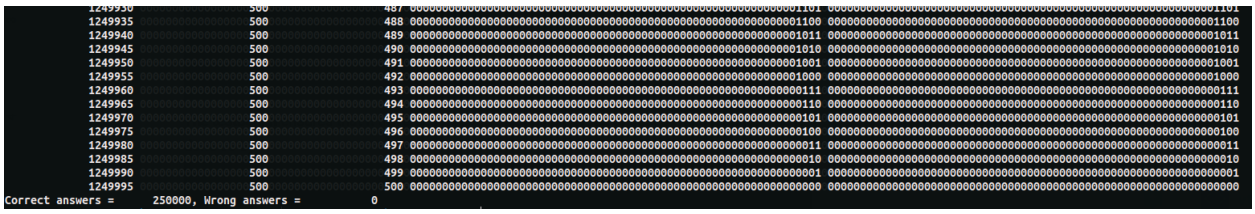
We know that in binary A-B = A+(B)'$_2$ where (B)'$_2$ is the 2's complement of B. So, for implementing subtraction we have created a 64 bit 2s complement which takes a 64 bit integer as the input and gives out the corresponding 2's complement as the output. The files involved in this are -



Hence, the code for the wrapper subtractor is quite neat where the most crucial parts of the module are -

```
complement x1(C,B);

   adder x2(A,C,D,Overflow);
```

Results obtained -



AND

The entire operation is implemented using a single for loop given by -

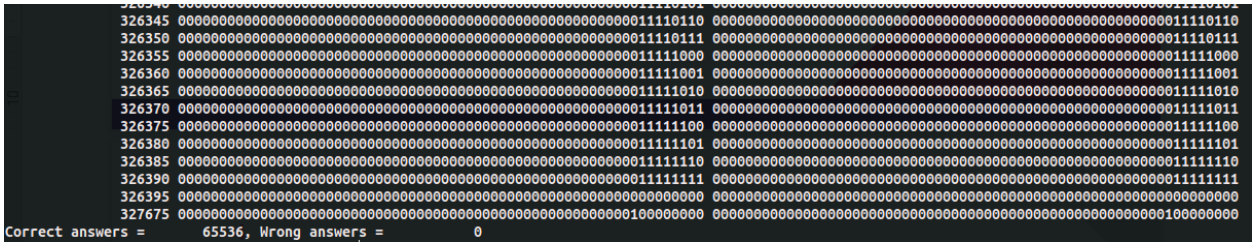
```
for(i=0;i<64;i=i+1) begin
```

```
        and(result[i], a[i], b[i]);
    end
```

The testbench ensures that it traverses through all possible types of cases while testing. This is done so by -

```
for(k = 0; k < 256; k = k + 1)
    begin
        a = a + 1;
        b = 0;
        for(l = 0; l<256; l = l + 1) begin
            b = b + 1;
            test = a & b;
            #5;
            if (test == result) begin
                flagright = flagright + 1;
            end else begin
                flagwrong = flagwrong + 1;
            end
        end
    end
```

The variable `test` is to check if our obtained value `result` is correct or wrong.



The results print the number of correct and wrong answers in the testing.

XOR

The entire operation is implemented using a single for loop given by -

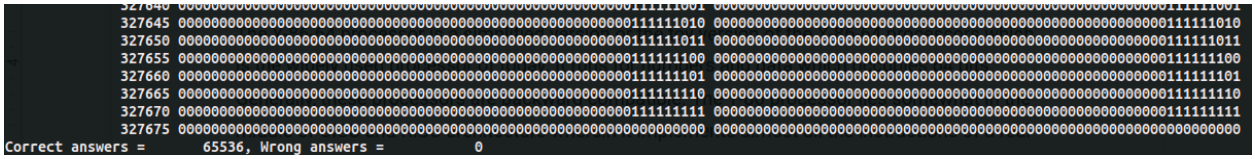
```
for(i=0;i<64;i=i+1) begin

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

   end
```

The testbench ensures that it traverses through all possible types of cases while testing. This is done so by -

```
for(k = 0; k < 256; k = k + 1)

   begin

       a = a + 1;

       b = 0;

       for(l = 0; l<256; l = l + 1) begin

           b = b + 1;

           test = a ^ b;

           #5;

           if (test == result) begin

               flagright = flagright + 1;
```

```
            end else begin

                flagwrong = flagwrong + 1;

            end

        end

    end
```

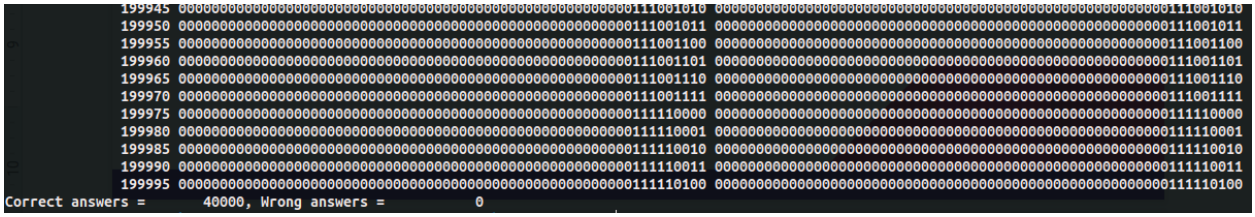
The variable `test` is to check if our obtained value `result` is correct or wrong.



The results print the number of correct and wrong answers in the testing.

Final ALU

Results -



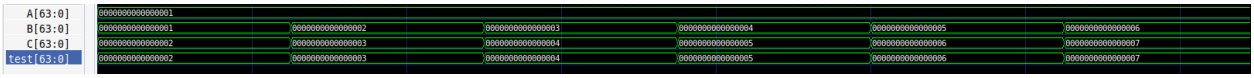
The vital component of the testbench -

```
for (j = 0; j < 4; j = j + 1)

    begin

        Control = j;

        for(k = 0; k < 100; k = k + 1)

        begin
```

```
        A = A + 1;
        B = 0;
        for (l = 0; l < 100; l = l + 1)
        begin
            B = B + 1;
            if (j==0) begin
                test = A + B;
            end
            else if (j==1) begin
                test = A - B;
            end
            else if (j==2) begin
                test = A&B;
            end
            else begin
                test = A^B;
            end
            #5;
            if (test == C)
            begin
                flag_r = flag_r + 1;
            end
```

```
            else begin
                flag_w = flag_w + 1;
            end
        end
    end
end
```

GTK wave plots -



Sequential

Module Descriptions

1. Fetch

The function of the fetch stage is to:

A. read an instruction from memory using the PC value as address
B. extract two 4-bit portions of instruction specifier byte which are called as ‘icode’ and ‘ifun’
C. Get the register specifier byte and give it to either or both register operand rA and rB
D. Get an 8-byte constant word valC
E. Compute address of the next instruction of the sequence using valP

The module fetch is used to achieve all the above. Initially we use instructions to get the actual instruction from the address in PC. Then we use the 8 bits of the first byte of instruction to obtain the icode (1st 4 bits) and ifun (next 4 bits). We used the icode values in a switch statement to get rA, rB, valC and valP.

2. Decode

The decode block has an inbuilt temporary memory which is used to mimic the register memory of a processor. The outputs of the decode block are fed to the execute block where all the operations involving the ALU are done along with a few others. This is a combinational block which runs every time any of the values are changed. The valA and valB values are set depending on the icode value which is done by using a switch case statement. Using the temporary register file that has been generated, we send the outputs to the actual registers.

In this block, since we can't have a 2D array as an input or output in a module of verilog, we declared them as individual registers - 14 of them. Inside the module however, there is a temporary register memory file that is generated which is - `reg [63:0] RegMem[0:14];`

At the end of the module, the individual registers have data extracted/written in the following way -

```
R0 = RegMem[0];
        R1 = RegMem[1];
        R2 = RegMem[2];
        R3 = RegMem[3];
        R4 = RegMem[4];
        R5 = RegMem[5];
        R6 = RegMem[6];
        R7 = RegMem[7];
        R8 = RegMem[8];
        R9 = RegMem[9];
        R10 = RegMem[10];
        R11 = RegMem[11];
```

```
            R12 = RegMem[12];

            R13 = RegMem[13];

            R14 = RegMem[14];
```

3. Execute

The execute block is where all the ALU operations happen and the instructions are executed while the condition code registers are set. The ALU block is called with two inputs as a and b which are set according to the icode value. The control input decides what the ALU will be doing with a and b. This is a combinational circuit which always runs when any of the internal value is modified.

The different variables that are required for setting the condition codes and other purposes have been declared initially in this manner -

```
wire sxoro, sxoroorz;


   xor g1(sxoro, sign, overflow);

   or g2(sxoroorz, sxoro, zero);

   reg [63:0] a, b;

   wire [63:0] answer;

   wire over;

   reg [1:0] control;

   ALU g3(answer, over, a, b, control);
```

These values have been used inside the always block. The ALU has also been called to get updated value everytime required as shown

```
ALU g3(answer, over, a, b, control);
```

4. Writeback

The memory block is where, if any, values are written into the memory as required. In some cases, memory is read to access elements. A temporary memory is created inside the module to mimic how it is done so in a processor. The memory array is, at max, 2^64 bits long as that is the largest value that is accessible by the registers. If the array is bigger, the memory access of them becomes impossible. Here, we assume it to be of length 1024.

5. Memory

The objective of write back is to write upto two results to the register file. Based on icode values i.e. different instructions we have to do, the results are written back to memory or registers. This block might be later merged with the decode block.

Similar to the decode stage, the codes are -

```
reg [63:0] mem[0:1023];
```

And

```
datamemory = mem[valE];
```

6. PC update

This block updates the PC value with the memory address of the next instruction. It takes into account the jump/pop/push instructions where the update is not direct. After this block has been worked, the PC now has the address to the next instruction from where the new instruction is fetched.

Pipelining

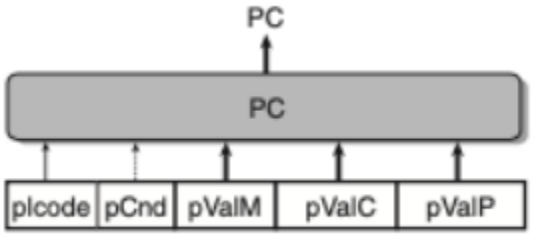
Convention

In each stage of the pipeline, we have register blocks which store all the data corresponding to that stage. The signals are named with the following convention -

1. x_name - This is the naming mannerism used when this is a signal that has been generated in that particular stage and is to be propagated.
2. X_name - This is used when this is a signal just propagated and not generated in that stage.

Rearrangement of individual blocks

The last block of the sequential design that we have implemented was the PC update block. As part of pipeline implementation, the suggested approach would be to eliminate the PC update block as a whole and have a section called predict PC in the fetch block which does the job. The predict PC section takes into account more than just the icode and ifun values to predict the new PC. The modular diagram is given below -



Pipeline register files

There are register files at the end of every stage which store the signals and data from the preceding stage. Since, in a pipeline, we want all the data to be stored at every stage, we use these register files. There are namely - fetch-reg, decode-reg, execute-reg, memory-reg, writeback-reg. Each store the data generated and to be propagated using non blocking

assignments. This is because in a register all the data needs to be fed into it at the posedge of a clock at the same time.

Implementation of data forwarding

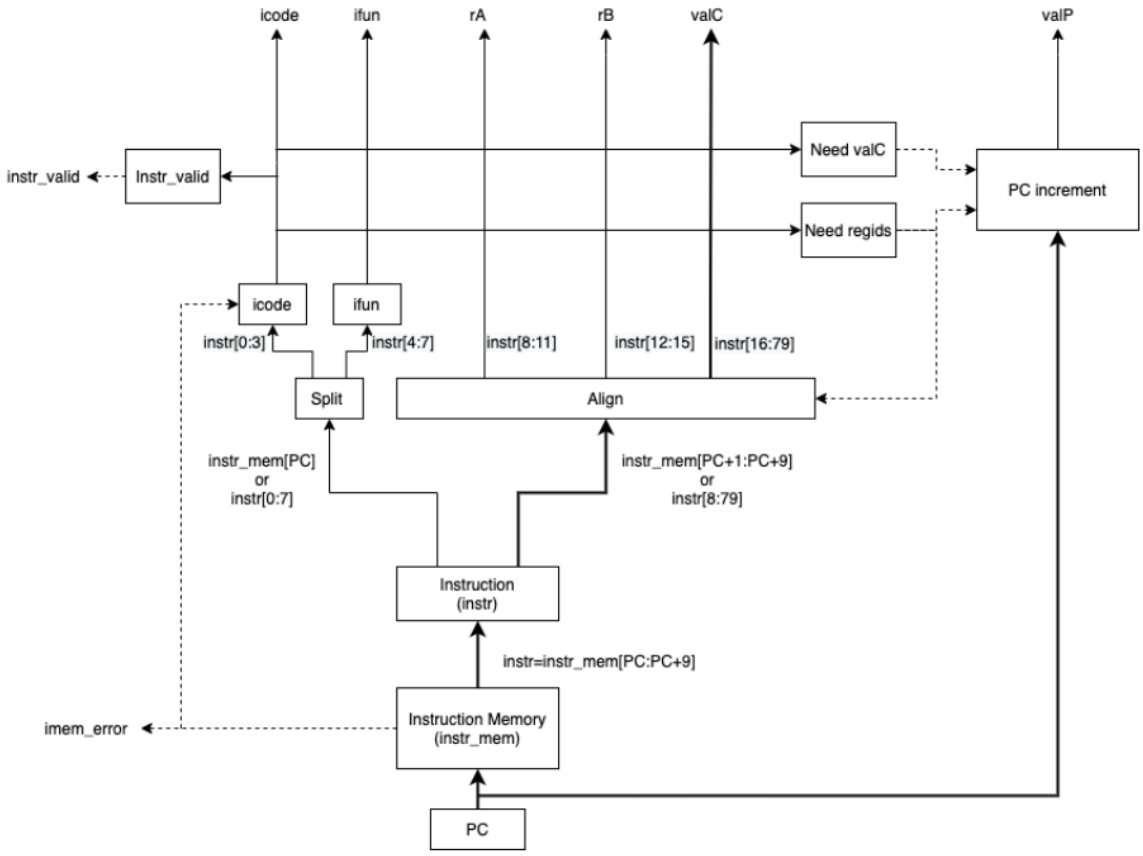
Data forwarding is a technique used to not have any data hazards. Data hazards are when the following instruction uses data from the registers, expecting them to be updated by the previous instructions, yet however that is not the case. In the process, we need valA values from the other instructions which is given by -

```
int d_valA = [
  # Use incremented PC
     D_icode in { ICALL, IJXX } : D_valP;
  # Forward valE from execute
     d_srcA == e_dstE : e_valE;
  # Forward valM from memory
     d_srcA == M_dstM : m_valM;
  # Forward valE from memory
     d_srcA == M_dstE : M_valE;
  # Forward valM from write back
      d_srcA == W_dstM : W_valM;
  # Forward valE from write back
     d_srcA == W_dstE : W_valE;
  # Use value read from register file
     1 : d_rvalA;
];
```

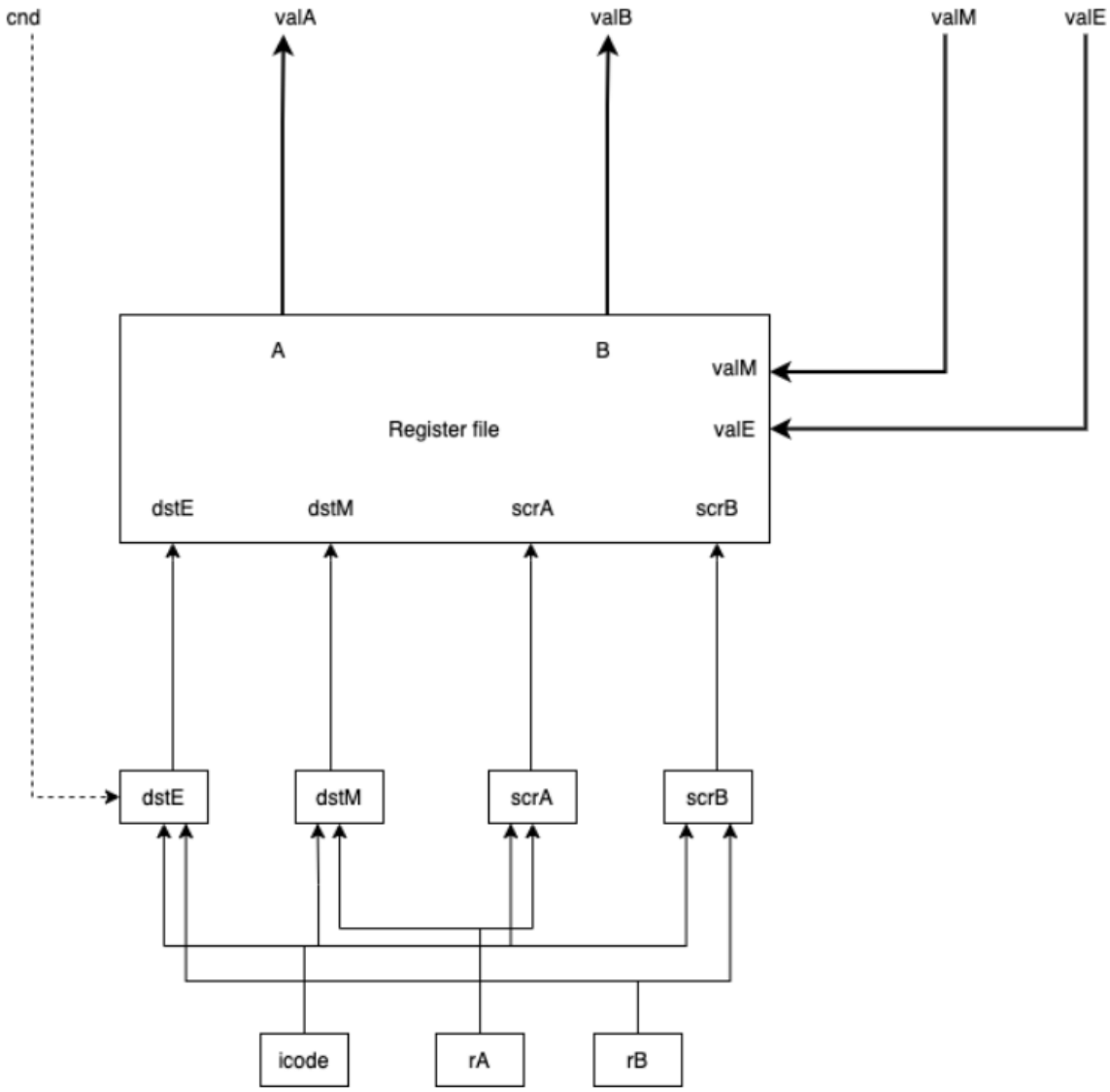
Execution

If we look at the individual blocks in a pipeline design, they are as follows.

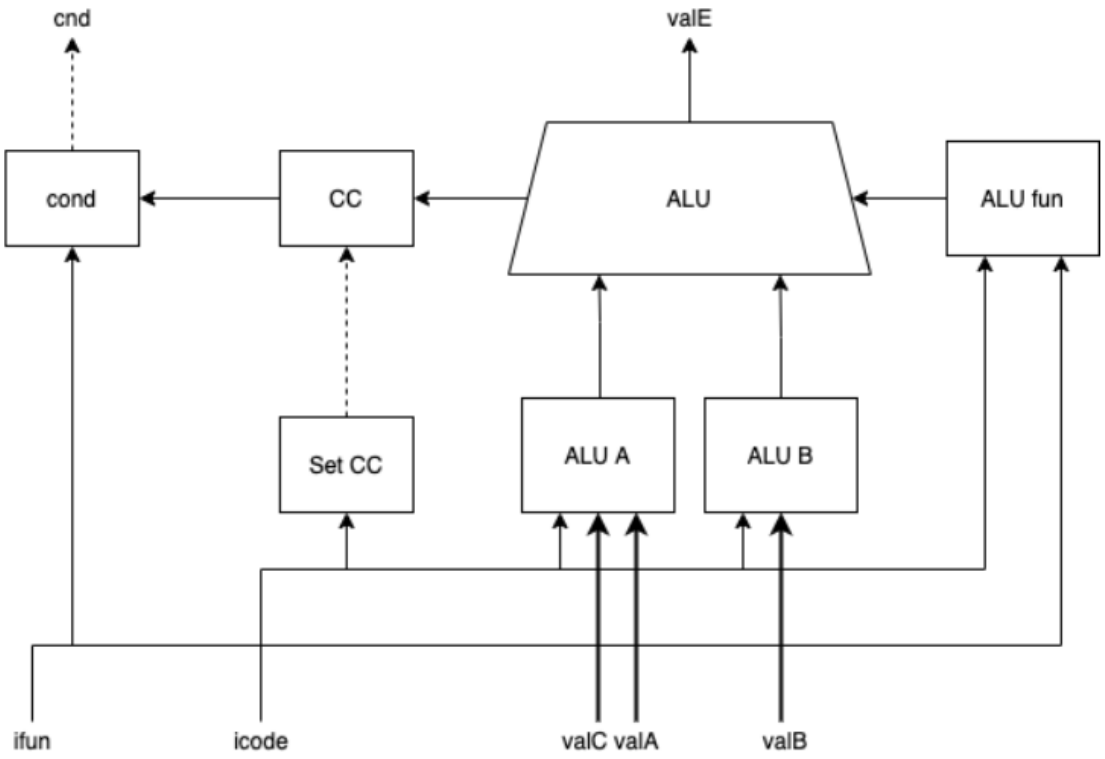
1. Fetch


icode
ifun
rA
rB
valC
valP
Need valC
PC increment
instr_valid
Instr_valid
Need regids
icode
ifun
instr[0:3]
instr[4:7]
instr[8:11]
instr[12:15]
instr[16:79]
Split
Align
instr_mem[PC]
or
instr[0:7]
instr_mem[PC+1:PC+9]
or
instr[8:79]
Instruction
(instr)
instr=instr_mem[PC:PC+9]
imem_error
Instruction Memory
(instr_mem)
PC

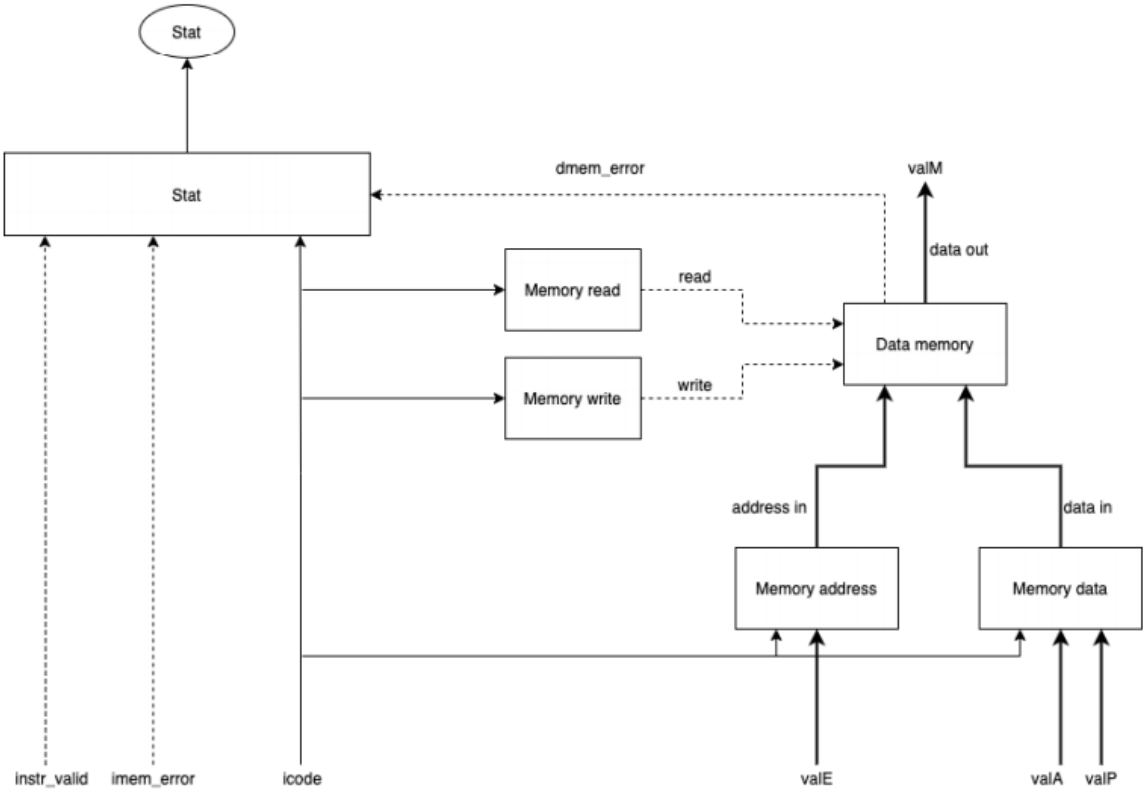
2. Decode / Writeback



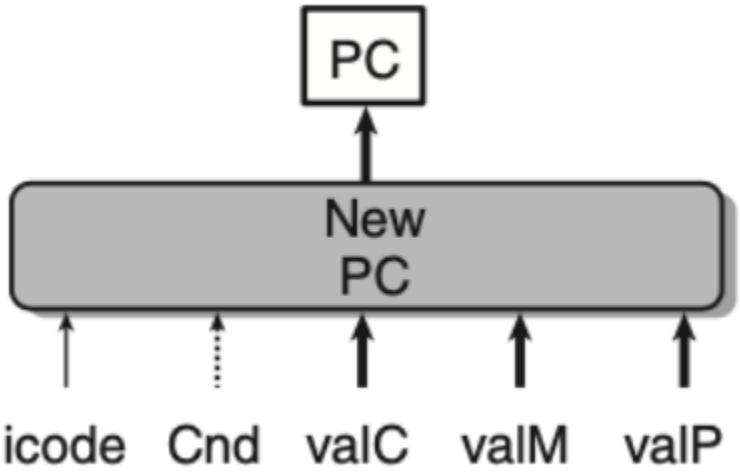
3. Execute


cnd
valE
cond
CC
ALU
ALU fun
Set CC
ALU A
ALU B
ifun
icode
valC valA
valB

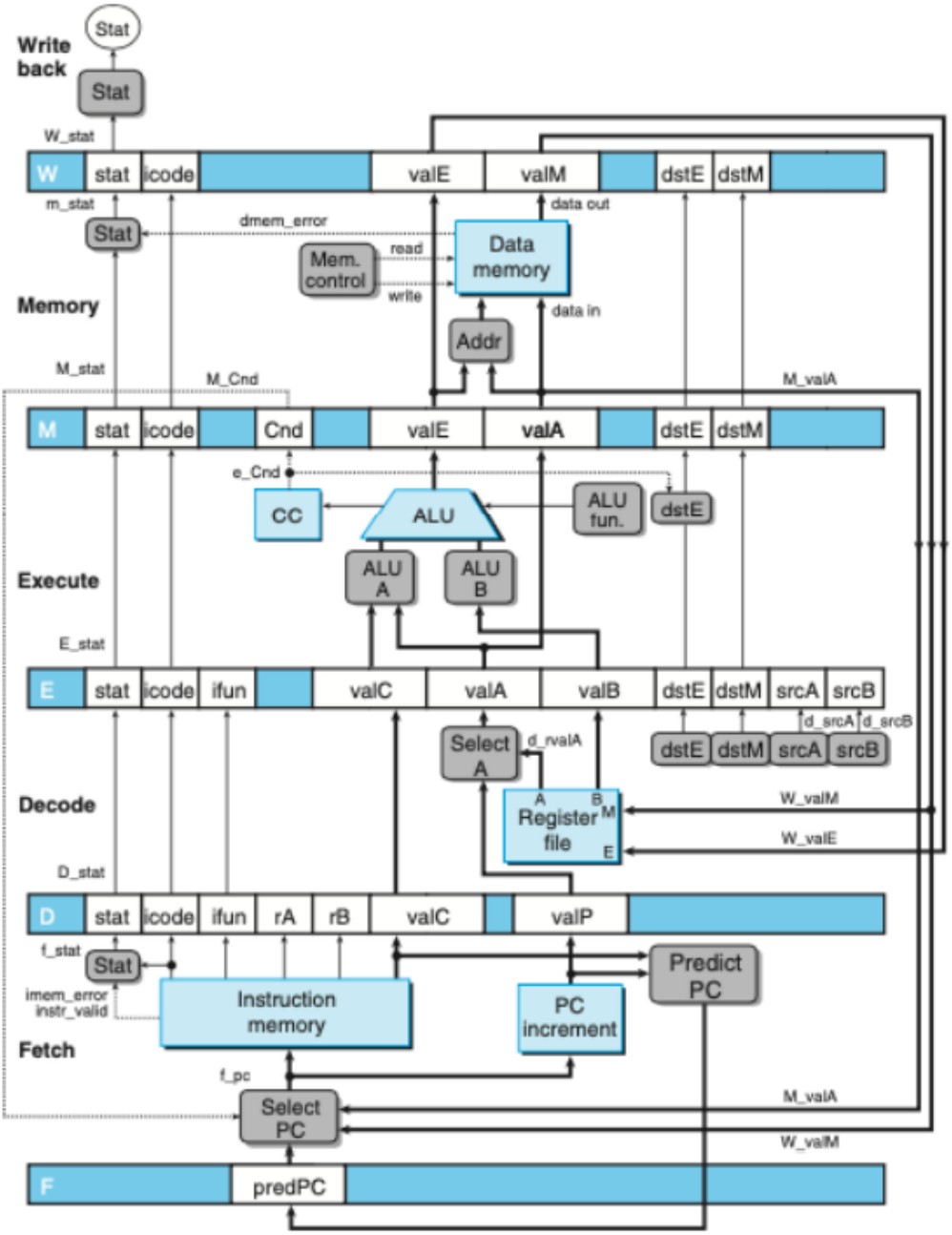
4. Memory



5. PC update



The modular structure for the entire processor design (pipeline) is as following -



Testing

Initially we tested with a simple test case. It is the implementation of the swap function.

The corresponding assembly code is -

```
irmovq $0, %rax
irmovq $17, %rbx
rrmovq %rax, %rsi
rrmovq %rbx, %rax
rrmovq %rsi, %rbx
```

After converting this into machine codes, we get the following which is stored in the instruction memory.

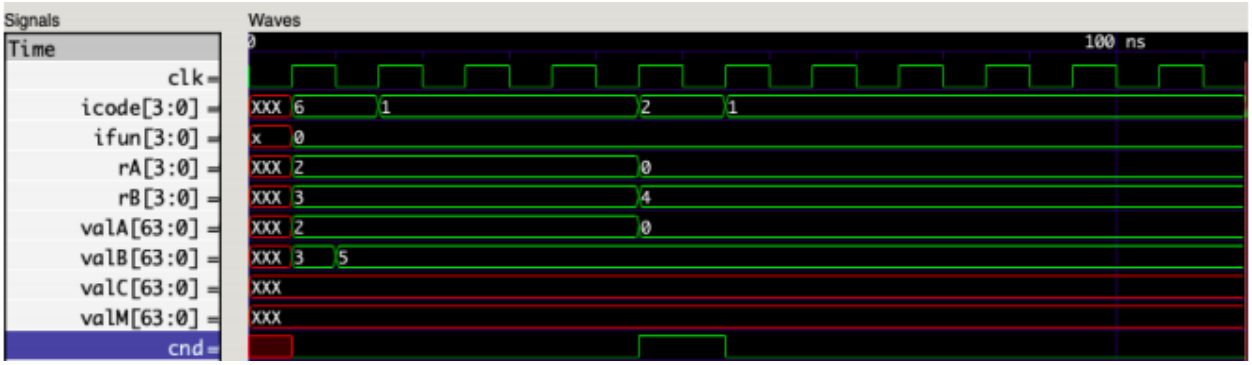
```
imem[0] = 8'b00110000; //30

   imem[1] = 8'b11100000; //E0

   imem[2] = 8'b00000000;

   imem[3] = 8'b00000000;

   imem[4] = 8'b00000000;

   imem[5] = 8'b00000000;

   imem[6] = 8'b00000000;

   imem[7] = 8'b00000000;

   imem[8] = 8'b00000000;

   imem[9] = 8'b00000000; // valC = 0

   imem[10] = 8'b00110000; //30

   imem[11] = 8'b11100011; //E3

   imem[12] = 8'b00000000;

   imem[13] = 8'b00000000;
```

```
imem[14] = 8'b00000000;

imem[15] = 8'b00000000;

imem[16] = 8'b00000000;

imem[17] = 8'b00000000;

imem[18] = 8'b00000000;

imem[19] = 8'b00010001; //v = 17

imem[20] = 8'b00100000; //20

imem[21] = 8'b00000110; //06

imem[22] = 8'b00100000; //20

imem[23] = 8'b00110000; //30

imem[24] = 8'b00100000; //20

imem[25] = 8'b01100000; //60
```

The execution of this showed successful results and the results matched the predicted ones.

After this, we have implemented the function which computes the HCF of two numbers. The corresponding machine codes -

The results we got are as follows -



We used an online simulator to view and confirm the results which are given below -

```
1  # Execution begins at address 0
2      .pos 0
3      irmovq stack, %rsp      # Set up stack pointer
4      call main       # Execute main program
5      halt            # Terminate program
6
7  main:
8  irmovq $0, %rax
9  irmovq $17, %rbx
10 rrmovq %rax, %rsi
11 rrmovq %rbx, %rax
12 rrmovq %rsi, %rbx
13
14
15
16
17 # Stack starts here and grows to lower addresses
18     .pos 0x200
19 stack:
20
21
```

REGISTERS

%rax	0x0000000000000000	0
%rcx	0x0000000000000000	0
%rdx	0x0000000000000000	0
%rbx	0x0000000000000000	0
%rsp	0x0000000000000000	0
%rbp	0x0000000000000000	0
%rsi	0x0000000000000000	0
%rdi	0x0000000000000000	0
%r8	0x0000000000000000	0
%r9	0x0000000000000000	0
%r10	0x0000000000000000	0