

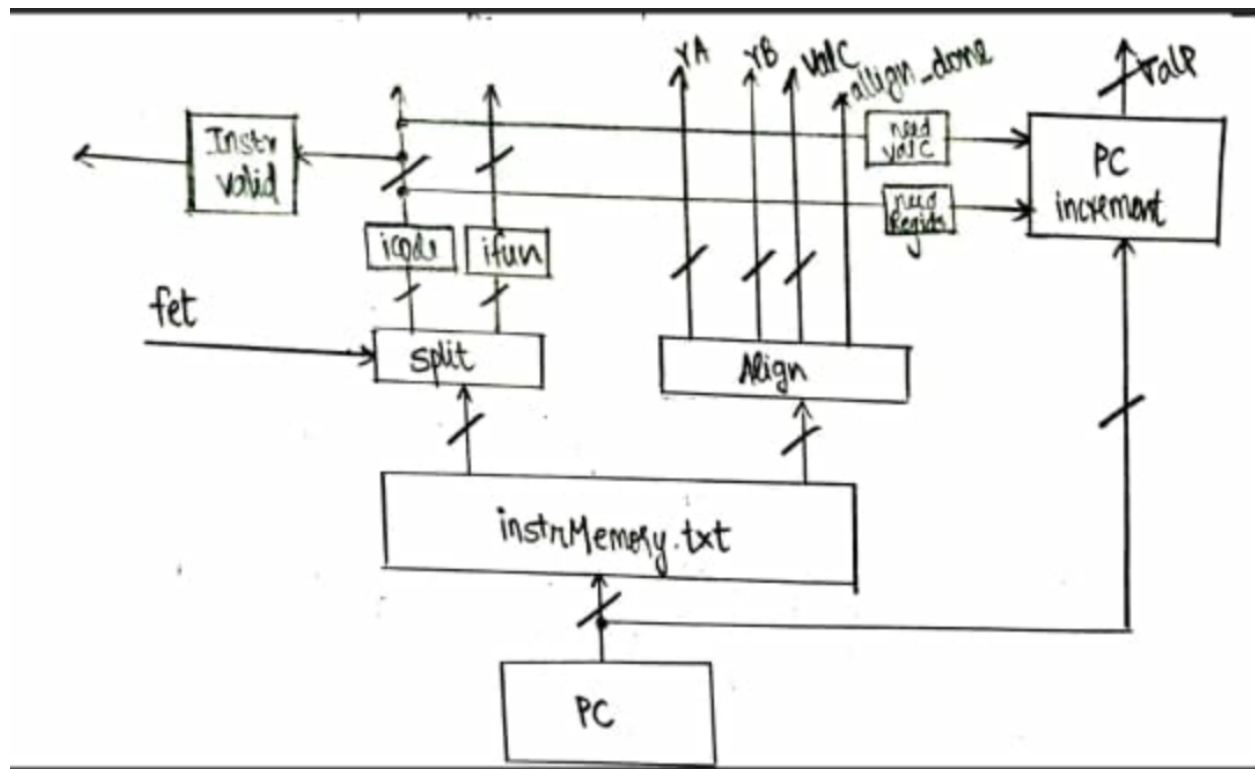


Intro to Processor Architecture - Project Report

Sasanka GRS
2019112017

Module descriptions and Architecture Diagrams:

Fetch Stage:

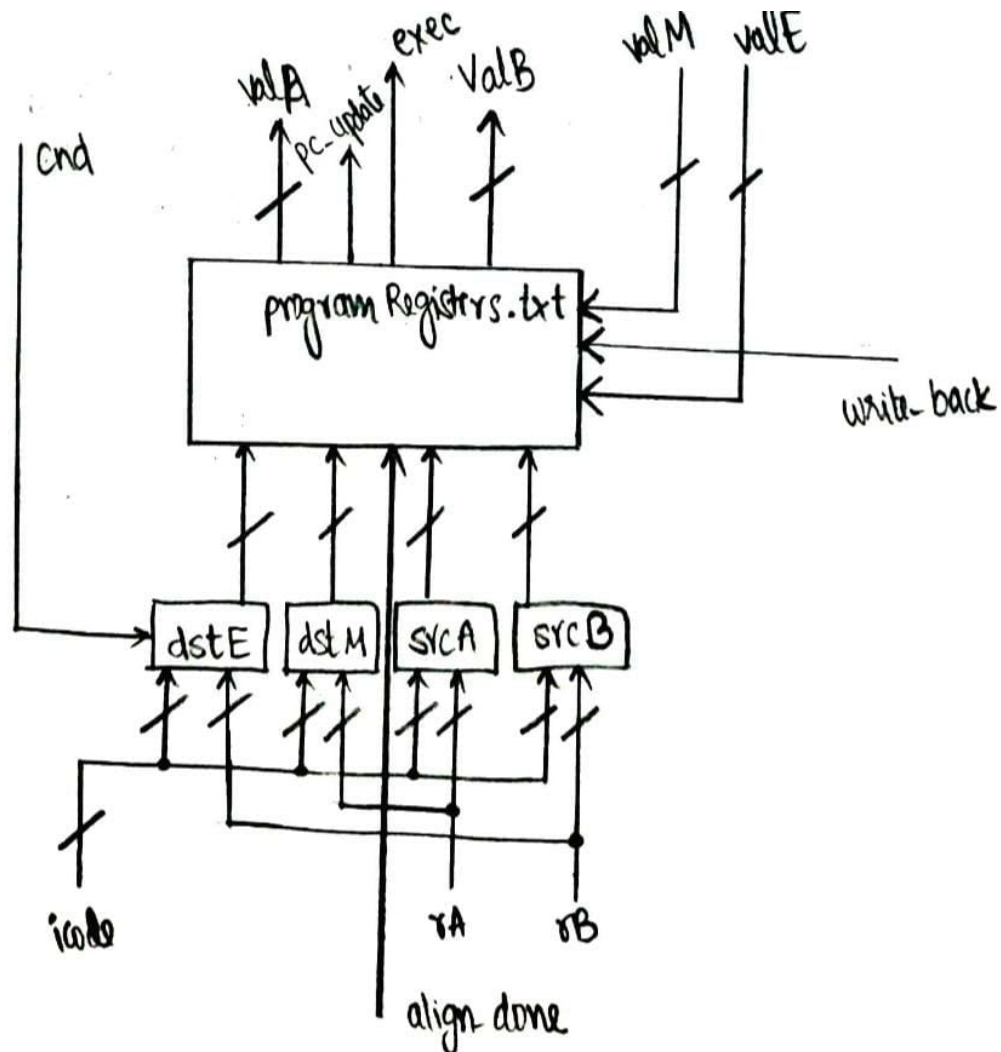


In the fetch stage, the PC is inputted, and instruction memory is a .txt file in the same folder, named as instrMemory.txt . The PC value is used to point to the right line in the instruction memory and retrieve 1 byte of data. This byte decides whether or not to access the next bytes of the retrieved data.

First, once the first byte of memory is received, the module split, breaks it into iCode and ifun. Based on iCode value, we set needRegids and needValC values, based on which the next bytes of information (registers or valC) is decided. Once the values rA, rB, valC have been conditionally read, we calculate valP based on previous PC, needRegids and needValC values.

The input signal fet to the split module preserves the sequential operation of the processor. It is set once the previous stage finishes its execution and in the present stage, it is placed in an always block with posedge so as to start the present execution only when the previous execution completes. Similarly, this stage generates an output signal align_done, which acts as a clock for starting the next stage execution.

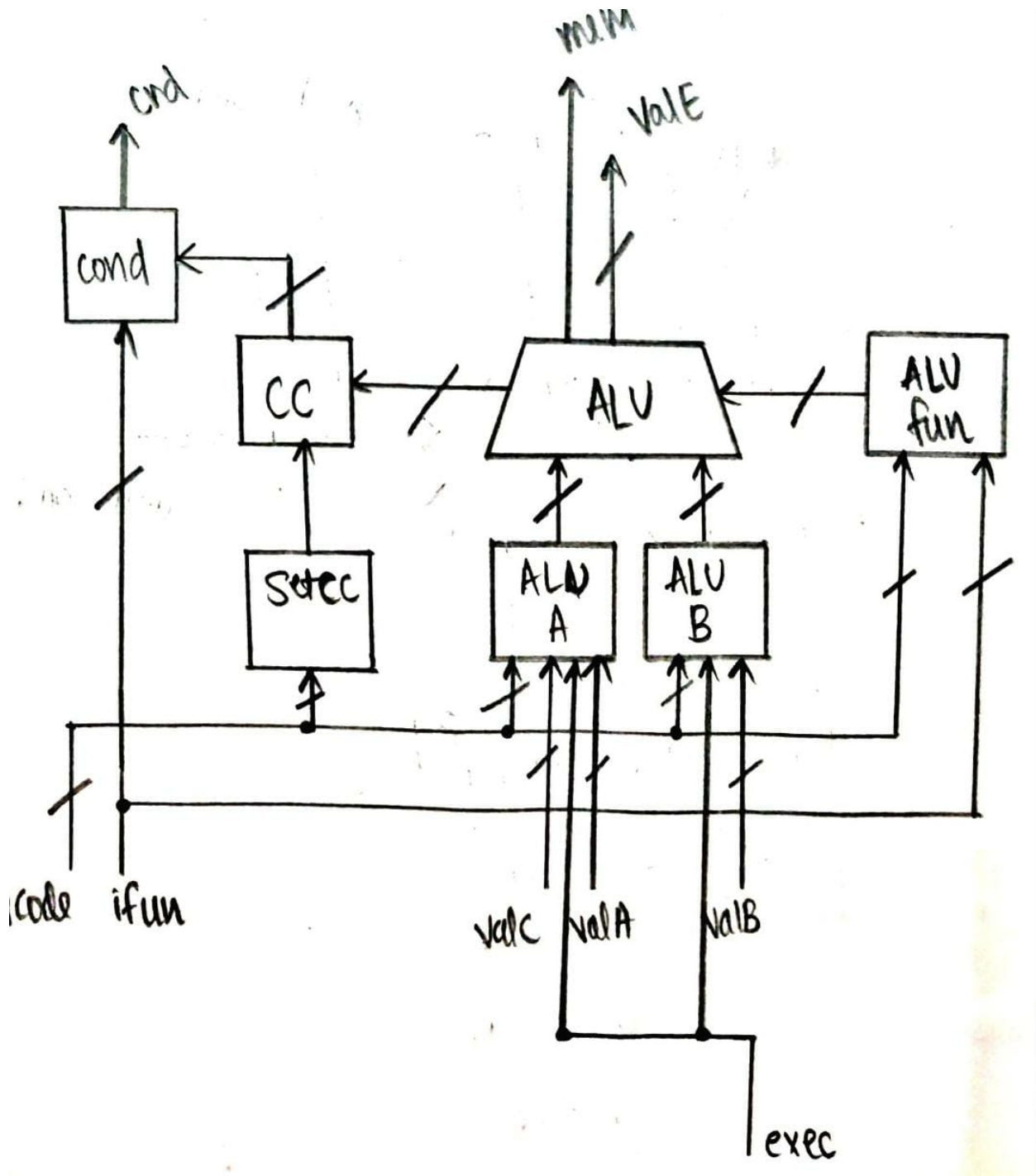
Decode and Write back stage:



In the decode and write back stage, the inputs are icode, rA, rB and align_done from the fetch stage, and valM and valE and write_back, which come from the execute stage and memory stage. The program registers are stored in a .txt file in the same folder, named as programRegisters.txt. All the 16 program registers are read and the values are stored in valA, valB. The read is conditional, that is, it depends on the condition generated in execute stage.

The output signals pc_update and exec act as clocks for the upcoming modules. In the write back operation, the program register file (programRegisters.txt) is overwritten with the current values to be replaced with.

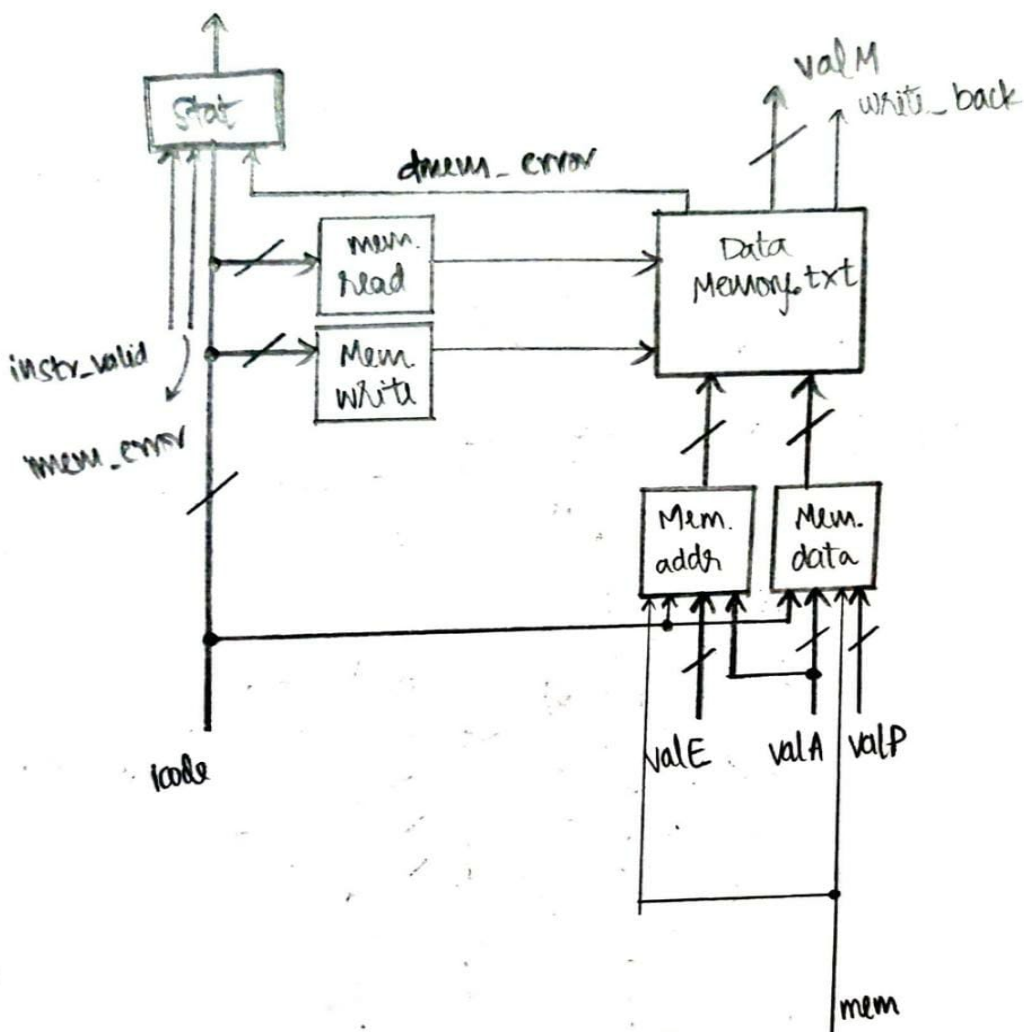
Execute and ALU:



In the execute stage, the inputs are `valC`, `valA`, `valB`, `exec`, `icode`, `ifun`, from the previous stages. Depending on the type of operation the ALU is performing, ALU function is chosen and the value `valE`, and `CC`, and `mem` are generated as outputs, and passed to the next stage.

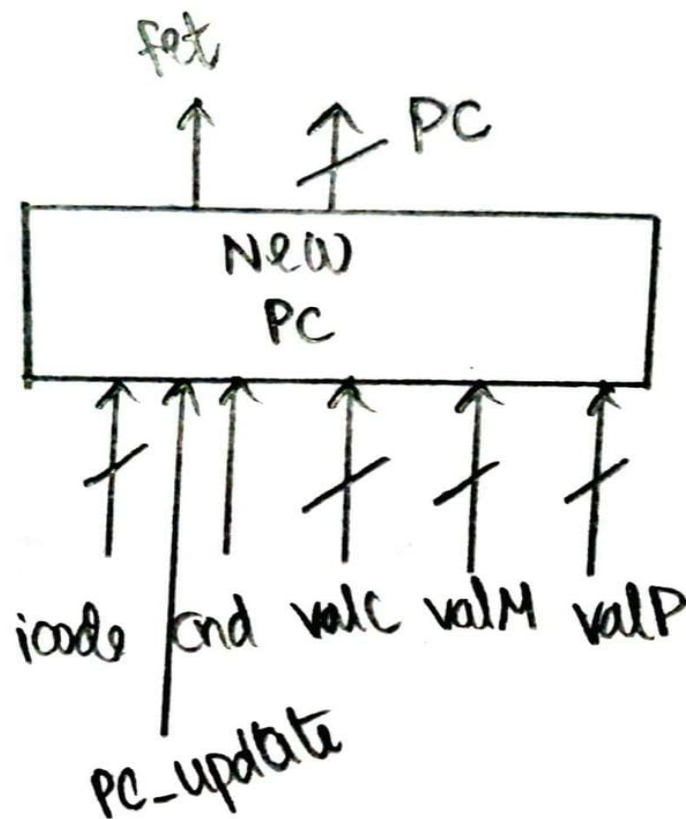
In this stage, we also set `cnd` based on CC values. The signal `mem` acts as a clock for the next stage. For this stage, the ALU function is chosen based on the instruction. For example, for icode 6, ALU function is `ifun`, in the case when icode is 2 or 3, the ALU function is set to 0, and one of the ALU inputs is set to 1 so as to obtain the same input value at the output (`valE`). In `popq` and `pushq` operations, `valB` is 64, and the ALU function is either 0 or 1 depending on whether 64 bits must be subtracted or added to the present stack pointer value.

Memory Stage:



In this stage, read or write to the data memory takes place. The data memory is stored in a .txt file in the same folder, named as dataMemory.txt. In this case mem, valE, valA, and valP and icode are the inputs from the previous stages. Based on the instruction, the read or write enables are set, and the corresponding operation is performed. The output is stored in valM, and the output signal write_back acts as a clock for the write back module in decode and write back stage.

PC update:



Based on the input clock (pc_update), and the valP, valC, valM, cnd and icode values obtained from previous stages helps decide the value of new PC and the PC value and fet are outputted from this module. The fet signal acts as a clock for the new instruction to be fetched.

In this stage, valP value which is precalculated in the fetch stage is assigned if there is no call or ret or jump operation being performed. In case of any of these 3 operations, valC, valM and cnd values are used in conditionally calculating the new PC.

List of Instructions Supported by Processor:

This SEQ processor works for all instructions in y86 instruction set, namely, halt, nop, cmovxx, irmovq, rmmovq, mrmovq, OPq, jxx, call, ret, pushq, popq.

C++ code + assembly code + GTKWave outputs for gcd of 2 numbers:

C++ code:

```
#include <iostream>
using namespace std;
// Recursive function to return gcd of a and b
int gcd(int a, int b)
{
    // Everything divides 0
    if (a == 0)
        return b;
    if (b == 0)
        return a;

    // base case
    if (a == b)
        return a;

    // a is greater
    if (a > b)
        return gcd(a-b, b);
    return gcd(a, b-a);
}
```

```

}

// Driver program to test above function
int main()
{
    int a = 4, b = 6;
    cout<<"GCD of "<<a<<" and "<<b<<" is "<<gcd(a, b);
    return 0;
}

```

Assembly code:

```


irmovq $4,%rax #Change value here
irmovq $6,%rbx #Change value here
subq %rax,%rbx
Basic check:
rrmovq %rbx,%rcx
subq %rax,%rcx
jg swap
jl next
halt
next:
subq $rax,%rbx
j Basic check
swap:
rrmovq %rax,%rcx
rrmovq %rbx,%rax
rrmovq %rcx,%rbx
j next

```



Instruction set examples and GTKWave:

When the values are 4 and 6, the instruction can be given as:

```
00110000
11110001
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000110
00110000
11110010
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000100
00100000
00100000
01100001
00010000
01110011
00000000
```



00000000
00000000
00000000
00000000
00000000
00000010
10001000
01110110
00000000
00000000
00000000
00000000
00000000
00000000
00000010
10001000
01100001
00100001
01110011
00000000
00000000
00000000
00000000
00000000
00000000
00000010
10001000
01110110
00000000




00000000
00000000
00000000
00000000
00000000
00000001
01100000
01100000
00100001
00100000
00100000
00100000
00100000
00010010
00100000
00000001
01110000
00101010
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00100000
00100000
00000000


The GTKWave plot is:



00000000



00000000
00000000
00000000
00000100
00100000
00100000
01100001
00010000
01110011
00000000
00000000
00000000
00000000
00000000
00000000
00000010
10001000
01110110
00000000
00000000
00000000
00000000
00000000
00000000
00000010
10001000
01100001
00100001
01110011



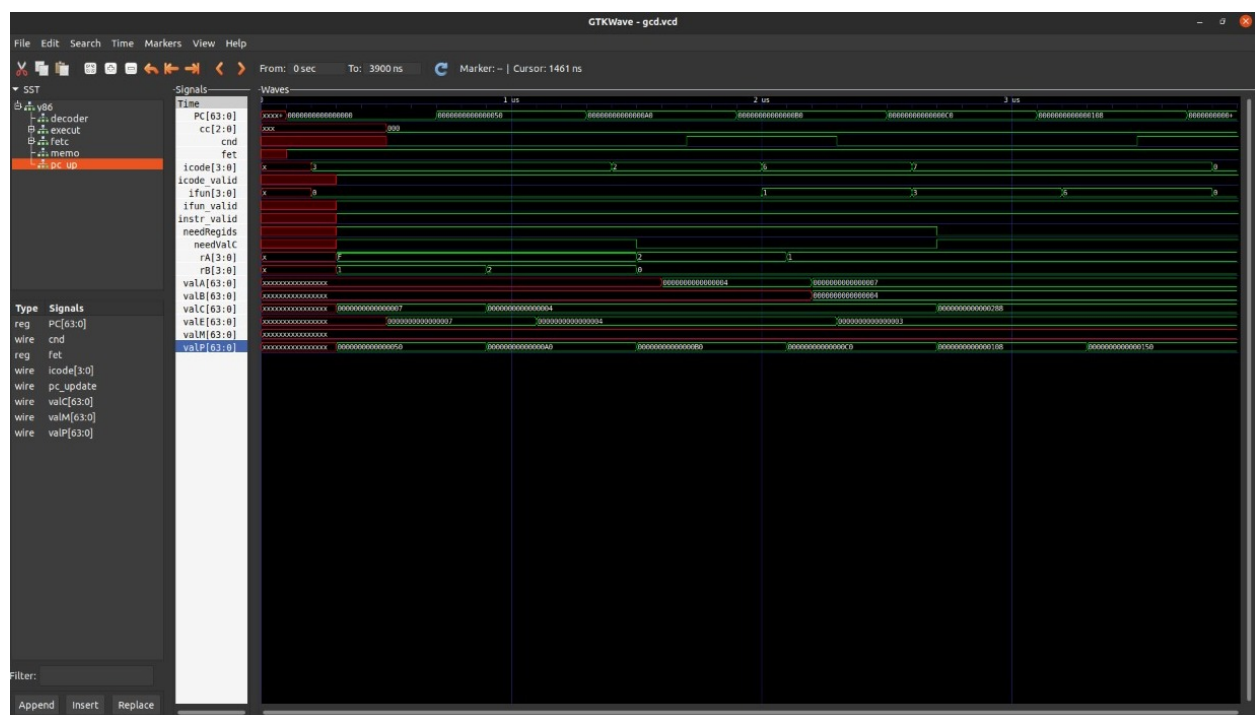
00000000
00000000
00000000
00000000
00000000
00000000
00000010
10001000
01110110
00000000
00000000
00000000
00000000
00000000
00000000
00000001
01100000
01100000
00100001
00100000
00100000
00100000
00010010
00100000
00000001
01110000
00101010
00000000
00000000

```

00000000
00000000
00000000
00000000
00000000
00100000
00100000
00000000

```

The GTKWave output is given as:



When the values are 10 and 15, the instruction can be given as:

```

00110000
11110001
00000000
00000000
00000000
00000000
00000000
00000000

```



00000000

00000000

00001010

00110000

11110010

00000000

00000000

00000000

00000000

00000000

00000000

00000000

00001111

00100000

00100000

01100001

00010000

01110011

00000000

00000000

00000000

00000000

00000000

00000000

00000010

10001000

01110110

00000000

00000000



00000000

00000000

00000000

00000000

00000010

10001000

01100001

00100001

01110011

00000000

00000000

00000000

00000000

00000000

00000000

00000010

10001000

01110110

00000000

00000000

00000000

00000000

00000000

00000000

00000001

01100000

01100000

00100001

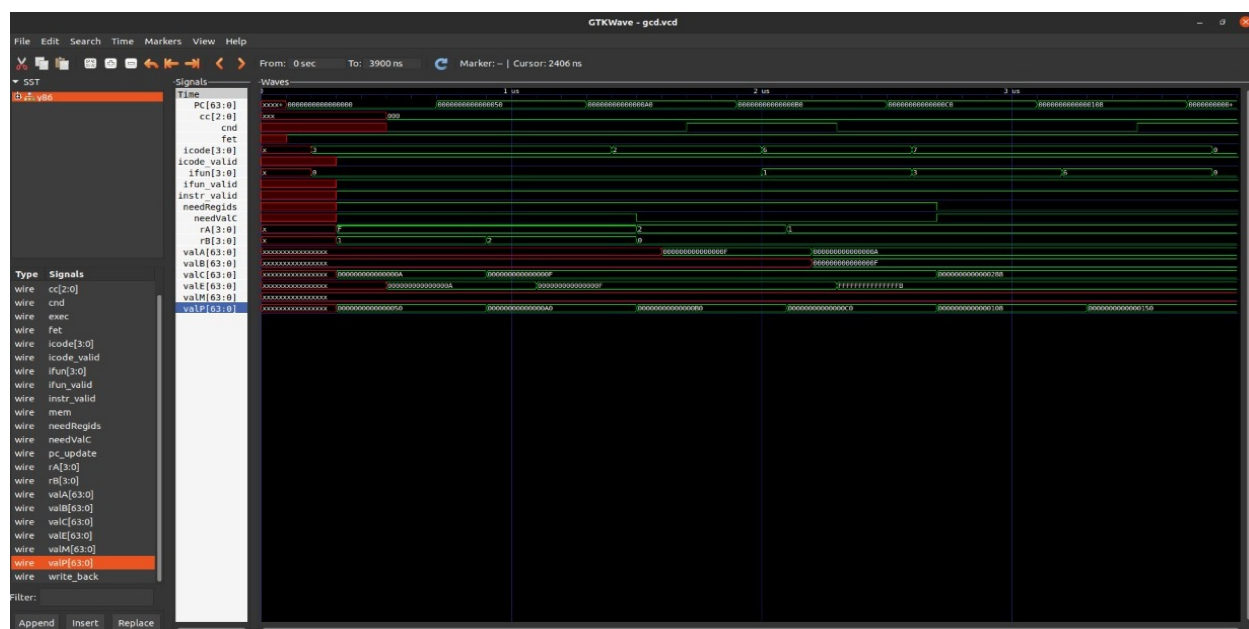
00100000

```

00100000
00100000
00010010
00100000
00000001
01110000
00101010
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00100000
00100000
00000000

```

The GTKWave output is:



01100000

000000001

```
// 0x000000000000
```

10

[illegible]

100

110

[illegible]

[illegible]

.....

[illegible]

.....

~~~~~

.....

\_\_\_\_\_

.....

\_\_\_\_\_

~~~~~

.....

.....

The gcd (10 (2)) is present in register r0 with the code given in the previous section of the report.

The data memory can be read or written is present in the file named dataMemory.txt present in the y86 folder in binary format. This file shows the memory changes for any memory related operations.

For compiling the processor as a whole, it is necessary to compile all modules separately, with the command :

```
iverilog Adder_2.v Adder_4.v ADD.v ALU.v AND.v Compliment_2.v Decoder_Writeback.v  
Execute.v Fetch.v Instr_Split.v Memory.v PC_update.v SUB.v XOR.v Y86_SEQ.v
```

And execute can be done by just using the command ./a.out

The current values of the variables in the processor are printed on the serial monitor and can be seen for each step.

The expected outputs can be seen either on the terminal screen, or the final outputs are available in the program registers and data memory based on the operation performed, and can also be viewed in gtkwave, by appending

initial begin

```
    $dumpfile("file.vcd");
```

```
    $dumpvars(0,y86);
```

```
end
```

In the y86 module, running the code, and running gtkwave file.vcd to observe the outputs in gtkwave.