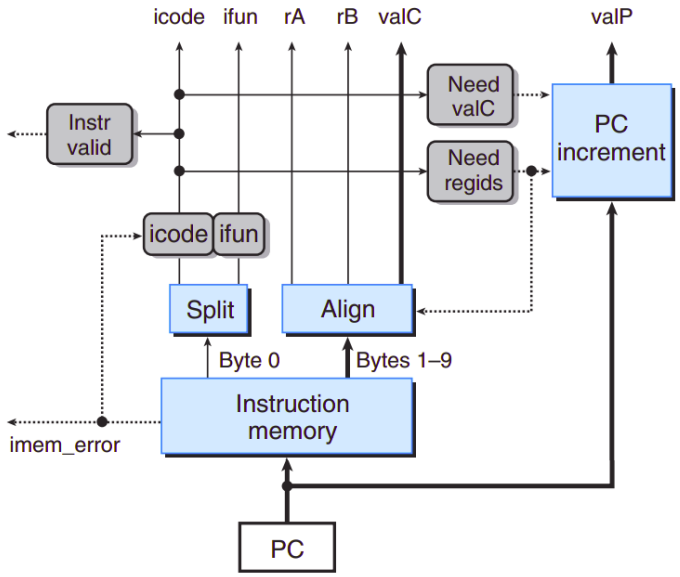
IPA Project report

U. S. S. Sasanka
2019102036

1. Processor architecture design of Y-86 ISA:

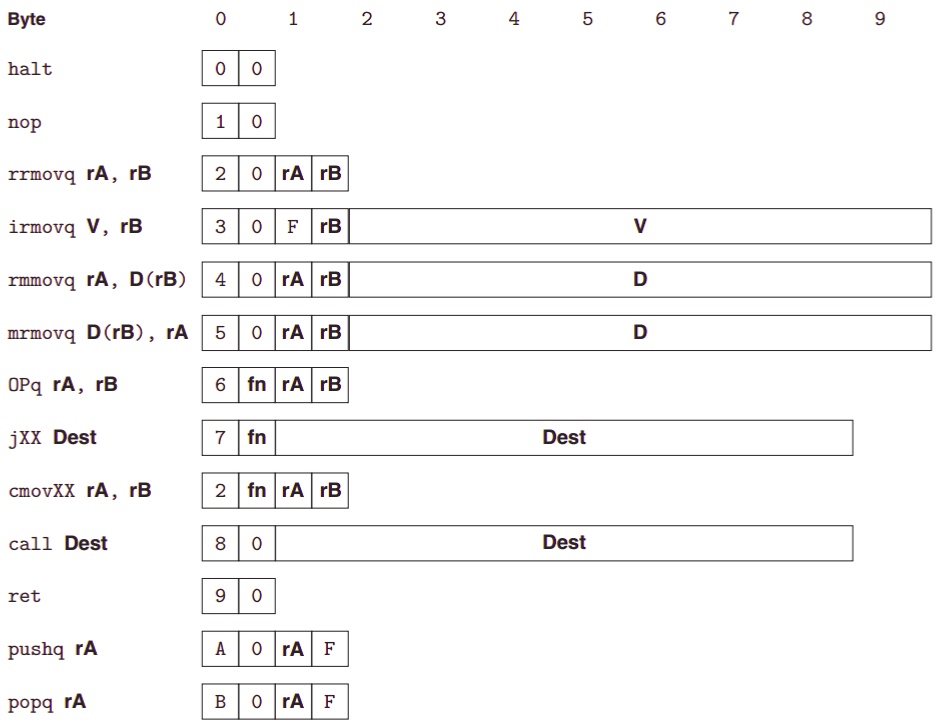
We have 5 main modules in this which will be explained and later, the working of the combined module design is explained.

Fetch:



Inputs: Program counter, Instruction memory
Outputs: icode, ifun, rA, rB, valC, valP

The program counter is given as the input to the fetch block. We then fetch the 10-byte instruction from the instruction memory.
The first byte is obtained using the pc address.
The first byte contains two parts icode and ifun. icode helps us to understand the type of instruction and ifun helps us to understand the function.



The second byte helps us to know the registers rA and rB from which we do the necessary operations.
The remaining bytes give us the constant term(valC).
The instr_valid is a flag which is set to 1 from 0 when the instruction length is greater than 10 bytes or any other invalid length.
The needvalC is set to 1 from 0 when the instruction has a constant term.
The needregids is set to 1 from 0 when the instruction has a register byte.
We also calculate the pc value which is updated in the later stages.

Code and explanations

```
input [0:639]instmem;
input [0:63]pc;

output reg [3: 0] icode, ifun;
output reg [3: 0] rA, rB;
output reg [63: 0] valC;
output reg [0: 63]valP;
output reg fetchflag;
```

The inputs and outputs described above are defined. An additional fetchflag which helps us while connecting the modules.

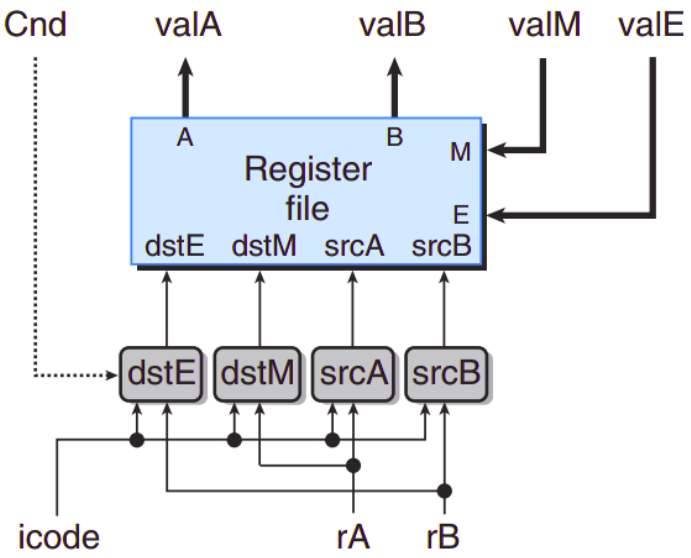
```
always @ (pc)
begin
fetchflag=0;
        for(i=0; i<4; i=i+1)
        begin
                icode[3-i]=instmem[pc+i];
                ifun[3-i]=instmem[pc+4+i];
        end
fhbyte=icode;
shbyte=ifun;

        if (fhbyte==0)
        begin
                $display("Halt instruction is fetched");
                fetchflag=0;
                $finish;
        end

        if ((fhbyte>=2 && fhbyte<=6) || fhbyte==4'ha||fhbyte==4'hb)
        begin
                for (i=0;i<4;i=i+1)
                begin
                        rA[3-i]=instmem[pc+8+i];
                        rB[3-i]=instmem[pc+12+i];
                end
        end
```

In this part of the code, we have extracted the values of icode and ifun from the instruction. Also the halt operation which stops the execution of a cycle is coded above.

```
//calculating the value of the constant i.e. valC
        if (fhbyte<6 && fhbyte>2)
        begin
                for (i=0;i<64;i=i+1)
                valC[63-i]=instmem[pc+16+i];
        end

        if (fhbyte==7||fhbyte==8)
        begin
                for (i=0;i<64;i=i+1)
                valC[63-i]=instmem[pc+8+i];
        end
```

Here, we have calculated the value of the constant.
The rest of the code contains calculation of program counter and other small operations

Decode and writeback:



Inputs: icode, rA, rB(For decode)
valM, valE(For writeback)
Outputs: valA, valB

In this stage, we have scrA and scrB which are the read ports(used in decode stage) and dstE and dstM which are the write ports(used in writeback stage).

We have a register file where there are 15 registers. Few important ones are rA, rB, %rsp(the 4th register) and 4'hf(kind of a null register). Based on the operation which can be obtained from icode, we obtain the values from the registers and decode the values present in it which are valA and valB. These values are sent to the next stage.

Also, we have the writeback happening after sometime(after completion of execute, memory stages) into the register through valM and valE. These values are written back into the required registers

Code and explanations:

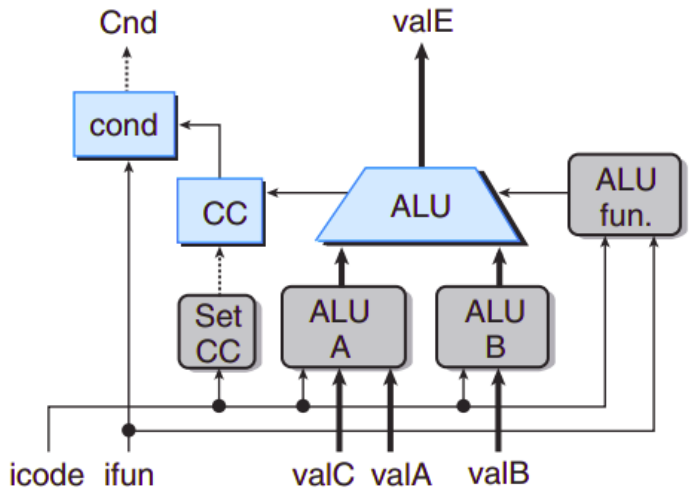
```
//Decode stage
always @ (fetchflag)
begin
        decodeflag=0;
        //Reading value of rA
        if(icode==9||icode==4'hb)
        srcA=4;
        else if(icode==2||icode==4||icode==6||icode==4'ha)
        srcA=rA;
        else
        srcA=4'hf;

        if(srcA!=4'hf)
        valA=sfbitreg[srcA];

        //reading value of rB
        if(icode==8||icode==9||icode==4'ha||icode==4'hb)
        srcB=4;
        else if(icode==4||icode==5||icode==6)
        srcB=rB;
        else
        srcB=4'hf;

        if(srcB!=4'hf)
        valB=sfbitreg[srcB];
decodeflag=1;

$display("Decode is done\n");
$display(" srcA=%h srcB=%h valA=%0h valB=%0h", srcA, srcB, valA, valB);
end
```

In the decode stage, we have read the value from the register which is 4 or rA depending on the operation that needs to be performed.

```
//writeback stage
always @ (posedge clock)

begin
        //writing values into the register
        if (icode==8||icode==9||icode==4'ha||icode==4'hb)
        destE=4'h4;
        else if(icode==2||icode==3||icode==6)
        destE=rB;
        else
        destE=4'hf;

        if(destE!=4'hf)
        sfbitreg[destE]=valE;

        if(icode==5||icode==4'hb)
        destM=rA;
        else
        destM=4'hf;

        if(destM!=4'hf)
        sfbitreg[destM]=valM;

$display("Writeback is done");
//Displaying the registers
for (i=0;i<15;i=i+1)
$display("Reg %0d = %d",i,sfbitreg[i]);
$display("destE=%h destM=%h valE=%h valM=%h", destE, destM, valE, valM);

end
```

The write back is done into the registers and the values are printed.

Execute:



Inputs: icode, ifun, valA, valB, valC
Ouputs: Cnd, valE

In this stage, we simply have ALU as the main unit which can performs addition, subtraction, multiplication and XOR of the inputs. In general based on the value of icode and ifun, we have the inputs to ALU A and ALU B. ALU A can be valC, valA, 8 or -8 depending on our input. ALU B is in general 0 or valB.
For most operations, we perform ALU A+ ALU B. However, when icode=6, it represents operation between A and B which is the four operations mentioned above to get valE. Also whenever these operations are performed, we set the condition codes which is zero flag, signed flag and overflow flag.
These three quantities when manipulated help us to activate cnd to 1 which are helpful for conditional moves and conditional jumps.

Code and explanations:

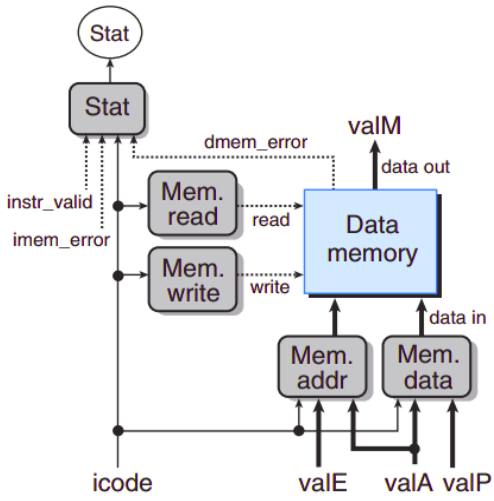
```
if(icode==6)
begin
        if(ifun==0)
        begin
        valE=aluA+aluB;
        end

        if(ifun==1)
        begin
        valE=aluA-aluB;
        end

        if(ifun==2)
        begin
        valE=aluA&aluB;
        end

        if(ifun==3)
        begin
        valE=aluA^aluB;
        end

end
```

We perform the operations by using the direct symbols and not the ALU of the first assignment.

```
cnd=0;
if(icode==2||icode==7)
begin
        if(ifun==0)// always
        cnd=1;
        if(ifun==1&& (sign^overflow|zero))//less than equal to
        cnd=1;
        if(ifun==2&& (sign^overflow))// less than
        cnd=1;
        if(ifun==3&& zero)//equal to
        cnd=1;
        if(ifun==4&& ~zero)// not equal to
        cnd=1;
        if(ifun==5&& ~(sign^overflow))// greater than equal to
        cnd=1;
        if(ifun==6&& ((~(sign^overflow))& (~zero))  )// greater than
        cnd=1;
end
```

In this part of the code we manipulate the three parameters to implement conditions like greater than, less than and so on.

Memory:



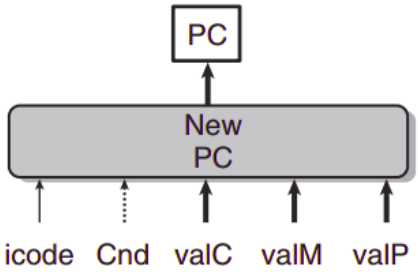
Inputs: icode, valE, valA, valP
Outputs: valM, stat
The memory stage is a very simple stage. We just read the value from the data memory or dump the value obtained in the execute stage into the data memory.
The icode is useful in determining whether we need to read from the data memory or write into the data memory. The stat is an error flag.

Code and explanations:

```
//Reading from memory
if(icode==5||icode==9||icode==4'hb)
memread=1;

if(memread==1)
begin
        for(i=0; i<64; i=i+1)
        valM[i]=datamem[address][i];
end
//Dumping into memory
if(icode==4||icode==8||icode==4'ha)
memwrite=1;

if(memwrite==1)
begin
         for(i=0; i<64; i=i+1)
         datamem[address][i]=datain[i];
end
```

The we have read and dumped values into the memory.

PC update:



Inputs: valC, valM, valP, icode
Ouput: updated pc

After the memory stage, if it is necessary, we write back into the register file as mentioned before. Now the last stage is simply updating the pc value so that the fetch and decode cycle continues.

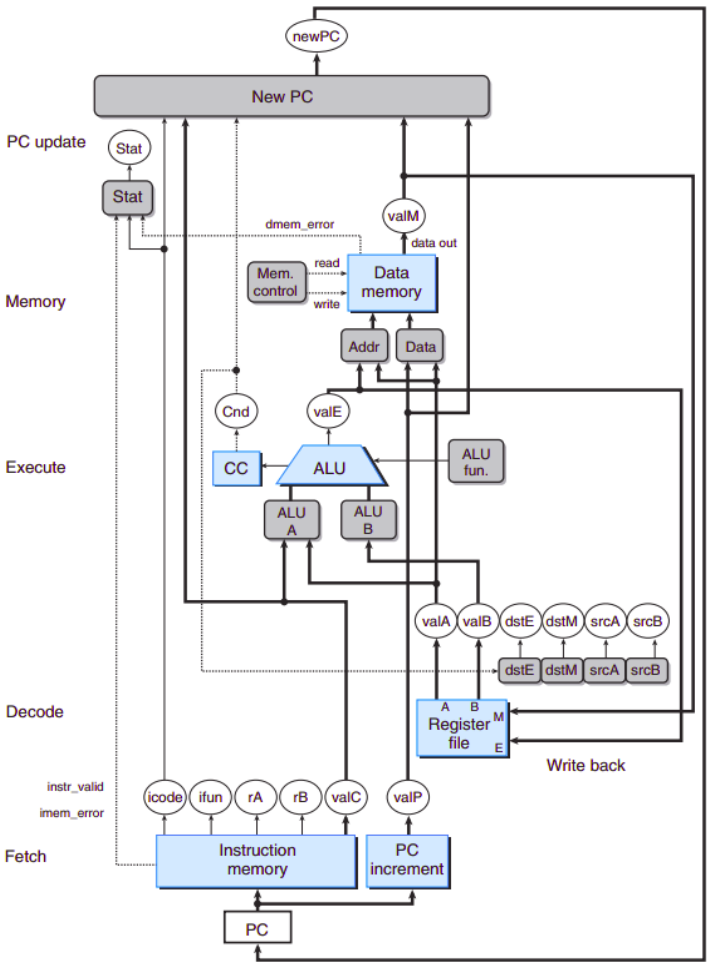
Code and explanations:

```
always @ (posedge clock)
begin
        if(icode==7&&cnd==1||icode==8)
        pc=valC;
        else if (icode==9)
        pc=valM;
        else
        pc=valP;
```

Based on our icode value, we update the pc which in most cases is the calculated value in fetch stage valP. We have also defined the updated pc value for the other cases as well.

Processor design:

We combine these 5 modules to design our required processor for Y-86 ISA. The diagram of this design is shown below.



We take the pc to be initially zero and then take an input file which contains the instruction memory. Until the code finds 00(ie halt), it keeps executing the instructions.

Code:

```
reg clock;
reg [0:639] instmem;
wire[0:63] valP, pc;
wire cnd, fetchflag, decodeflag, executeflag, err;
wire[3:0] icode, ifun, rA, rB;
wire [63:0] valA, valB, valC, valE, valM;
reg [7:0] mem[0: 199];

always #5 clock=~clock;

fetch seqf(instmem, pc, icode, ifun, rA, rB, valC, valP, fetchflag);
decode seqd(fetchflag, clock, icode, rA, rB, valA, valB, valM, valE, decodeflag);
execute seqe(decodeflag,valA, valB, valC, valE, icode, ifun, cnd, executeflag);
memory seqm(executeflag, icode, valE, valA, valM, err);
pcupdate seqwb(clock, cnd, valC, valM, icode, valP, pc);
```

Here, we integrated all these stages together and are able to obtain the outputs which will be shown later.

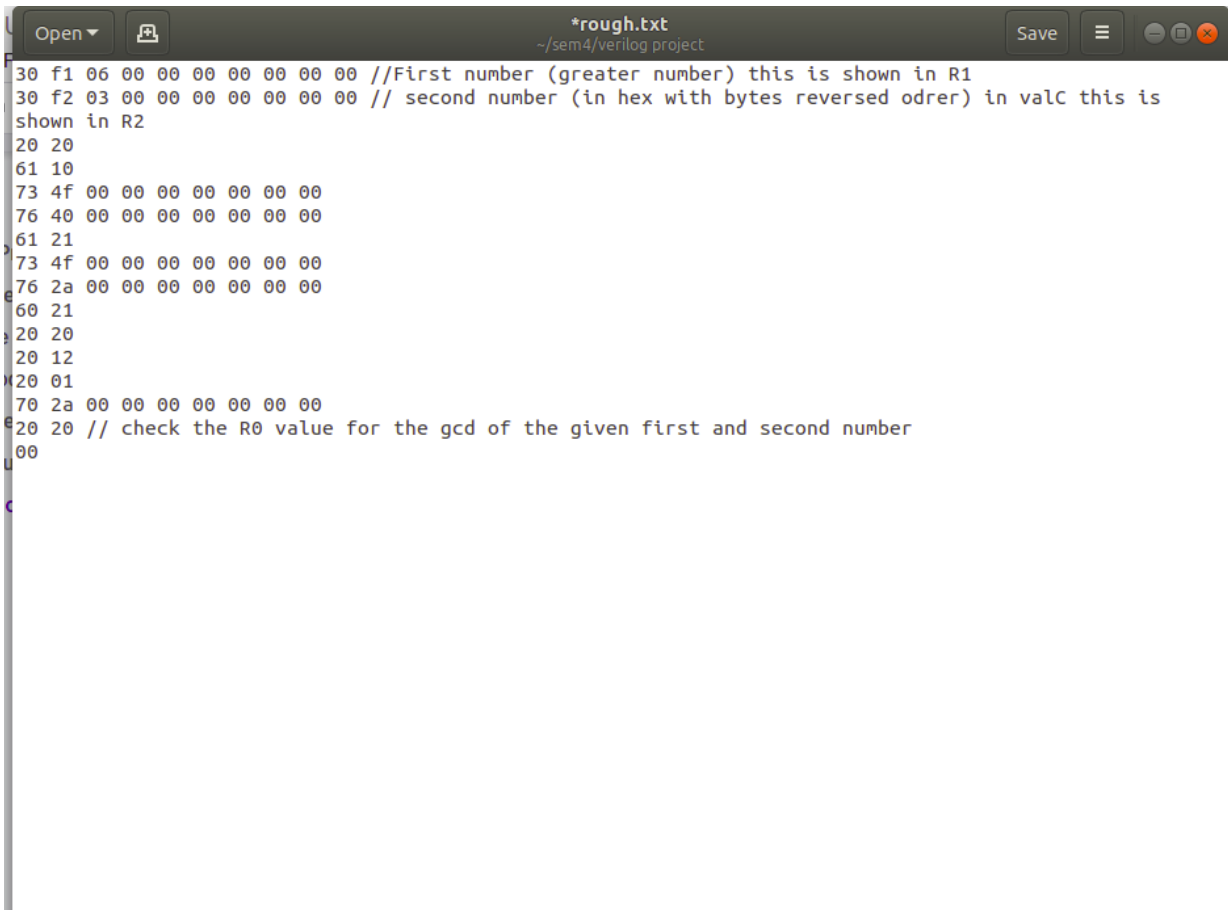
2. Instructions supported by my design

The design is a sequential design which supports all the operations that an ideal sequential processor should have.
We can perform all functions like moving, operations, jump, call, return, push, pop.
However, there are certain situations where the code might not give the correct output due to overflow, or due to avoiding certain error flags especially in the memory module and other reasons which I was not able to figure out.

3. HCF of two numbers

The code is able to find the hcf of two numbers.

The test bench that has been written takes input as 2 numbers.

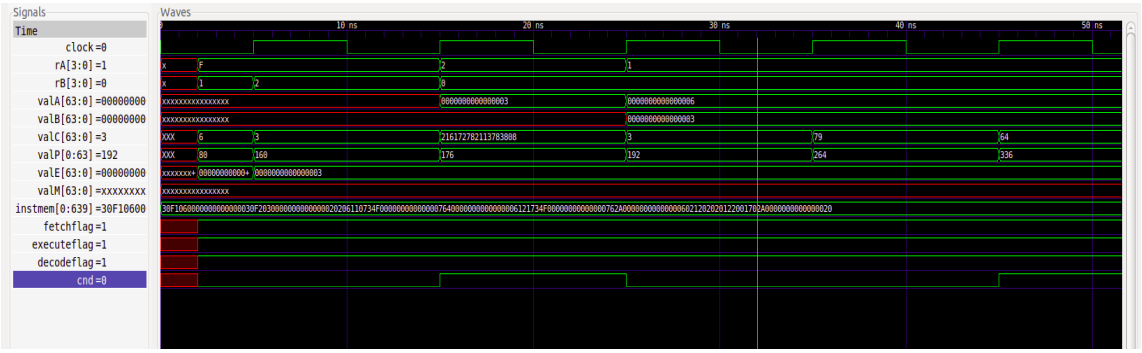


The following is the code and we observe that we are able to obtain the GCD for the test case.

```
Reg 13 =                    x
Reg 14 =                    x
destE=f destM=f valE=0000000000000003 valM=xxxxxxxxxxxxxxxx
Fetch is done
icode=7 ifun=6 rA=1 rB=0 valC=0000000000000040 valP=336
Decode is done

 scrA=f srcB=f valA=6 valB=3
Execute stage is done
Condition codes zero flag=0 sign flag=0 overflow flag=0 valE=3
Memory is done

valM=x
Writeback is done
Reg 0 =                     3
Reg 1 =                     6
Reg 2 =                     3
Reg 3 =                     x
Reg 4 =                    10
Reg 5 =                     x
Reg 6 =                     x
Reg 7 =                     x
Reg 8 =                     x
Reg 9 =                     x
Reg 10 =                    x
Reg 11 =                    x
Reg 12 =                    x
Reg 13 =                    x
Reg 14 =                    x
destE=f destM=f valE=0000000000000003 valM=xxxxxxxxxxxxxxxx
PC update is done

PC=64
Halt instruction is fetched
Decode is done

Execute stage is done
surya@surya-HP-Laptop-15-da0xxx:~/sem4/verilog project$
```

For the test case, we have taken R1=6, R2=3. The GCD is in R0=3

GTK Wave:

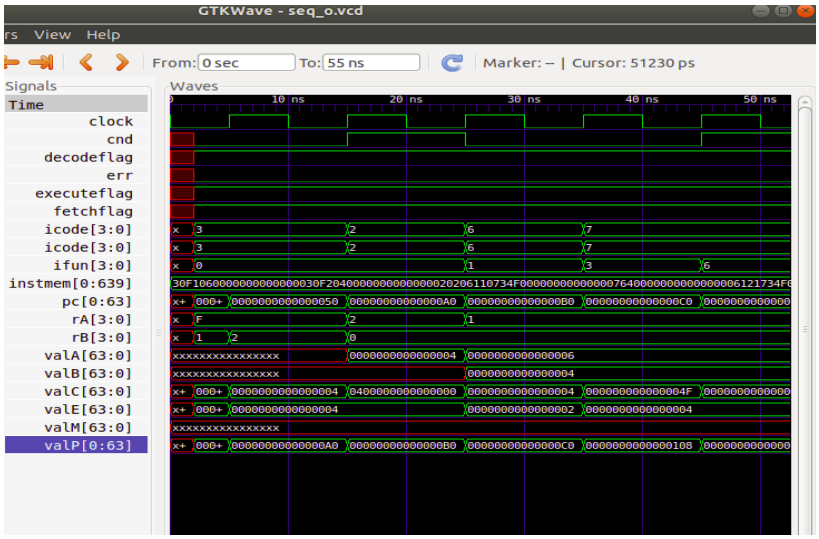


Test case 2

R1=6, R2=4

```
Decode is done

 scrA=f srcB=f valA=6 valB=4
Execute stage is done
Condition codes zero flag=0 sign flag=0 overflow flag=0 valE=4
Memory is done

valM=x
Writeback is done
Reg 0 =                    2
Reg 1 =                    6
Reg 2 =                    4
Reg 3 =                    x
Reg 4 =                   10
Reg 5 =                    x
Reg 6 =                    x
Reg 7 =                    x
Reg 8 =                    x
Reg 9 =                    x
Reg 10 =                    x
Reg 11 =                    x
Reg 12 =                    x
Reg 13 =                    x
Reg 14 =                    x
destE=f destM=f valE=0000000000000004 valM=xxxxxxxxxxxxxxxx
PC update is done

PC=64
Halt instruction is fetched
Decode is done

Execute stage is done
surya@surya-HP-Laptop-15-da0xxx:~/sem4/verilog project$
```

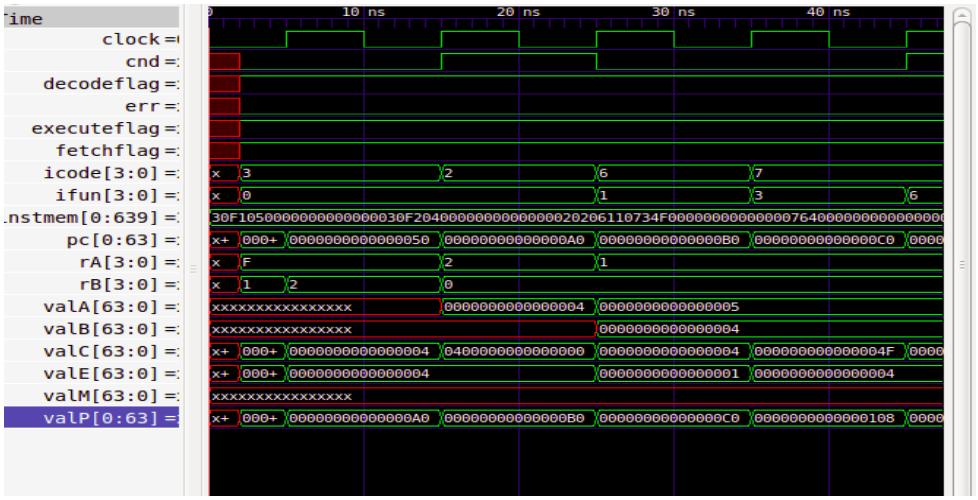
R0=2 is the GCD.



Test case 3: R1= 5 and R2=4

```
 scrA=f srcB=f valA=5 valB=4
Execute stage is done
Condition codes zero flag=0 sign flag=0 overflow flag=0 valE=4
Memory is done

valM=x
Writeback is done
Reg 0 =                    1
Reg 1 =                    5
Reg 2 =                    4
Reg 3 =                    x
Reg 4 =                   10
Reg 5 =                    x
Reg 6 =                    x
Reg 7 =                    x
Reg 8 =                    x
Reg 9 =                    x
Reg 10 =                    x
Reg 11 =                    x
Reg 12 =                    x
Reg 13 =                    x
Reg 14 =                    x
destE=f destM=f valE=0000000000000004 valM=xxxxxxxxxxxxxxxx
PC update is done

PC=64
Halt instruction is fetched
Decode is done

Execute stage is done
surya@surya-HP-Laptop-15-da0xxx:~/sem4/verilog project$
```

R0=1 because they are coprimes



This code works fine for certain numbers but gives wrong answers as well in some situations.

4. Other test benches

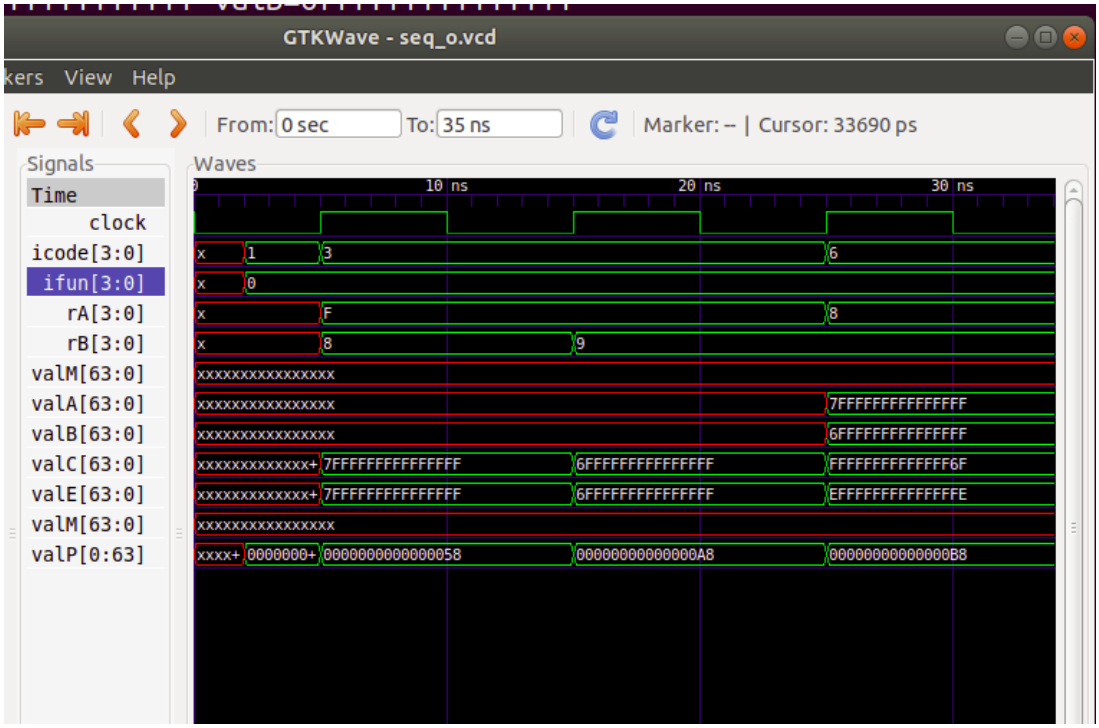
```
10
30 f8 ff ff ff ff ff ff ff 7f
30 f9 ff ff ff ff ff ff ff 6f
60 89
00
10
80 58 00 00 00 00 00 00 00
00
90
30 f8 ff ff ff ff ff ff ff 7f
30 f9 ff ff ff ff ff ff ff 7f
61 89
22 91
00
76 10 01 00 00 00 00 00 00
61 98
00
20 81
40 19 05 00 00 00 00 00 00 00
50 29 05 00 00 00 00 00 00 00
00
90
a0 5f
b0 6f
~
```

This test bench helps us to demonstrate the different possibilities that exist. The code gets executed every time it finds a 00. So comment out certain parts in this code to obtain the values and perform the operations.

The code when run as it is adds two numbers

```
Reg 3 =                  x
Reg 4 =                 10
Reg 5 =                  x
Reg 6 =                  x
Reg 7 =                  x
Reg 8 =  9223372036854775807
Reg 9 =  8070450532247928831
Reg 10 =                  x
Reg 11 =                  x
Reg 12 =                  x
Reg 13 =                  x
Reg 14 =                  x
destE=9 destM=f valE=6fffffffffffff valM=xxxxxxxxxxxxxxx
Fetch is done
icode=6 ifun=0 rA=8 rB=9 valC=ffffffffffffff6f valP=184
Decode is done

 scrA=8 srcB=9 valA=7fffffffffffff valB=6fffffffffffff
Execute stage is done
Condition codes zero flag=0 sign flag=1 overflow flag=1 valE=17293822569102704638
Memory is done

valM=x
Writeback is done
Reg 0 =                  x
Reg 1 =                  x
Reg 2 =                  x
Reg 3 =                  x
Reg 4 =                 10
Reg 5 =                  x
Reg 6 =                  x
Reg 7 =                  x
Reg 8 =  9223372036854775807
Reg 9 = 17293822569102704638
Reg 10 =                  x
Reg 11 =                  x
```

GTK wave:



Test case 2:

```
/*10
30 f8 ff ff ff ff ff ff ff 7f
30 f9 ff ff ff ff ff ff ff 6f
60 89
00*/
10
80 58 00 00 00 00 00 00 00
90
30 f8 ff ff ff ff ff ff ff 7f
30 f9 ff ff ff ff ff ff ff 7f
61 89
22 91
00
76 10 01 00 00 00 00 00 00
61 98
00
20 81
40 19 05 00 00 00 00 00 00 00
50 29 05 00 00 00 00 00 00 00
00
90
a0 5f
`b0 6f
```

So as mentioned above, we can execute our code and produce outputs.

5. Instructions to run the code

We have the seq.v module which is our controller. It reads from a file called rought.txt. Take the test cases in rough.txt. Compile seq.v and using the command iverilog seq.v and run it using the command ./a.out. The rough.txt already contains a few test cases. Comment and uncomment them or add commands in the same fashion i.e just provide an instruction in the normal format discussed in class and textbook.