Project Report

Adithya Sunil Edakkadan **2019102005**

Table of contents

Overview

Module description

- Sequential
  - Fetch
  - Decode and write back
  - Execute
  - Memory
  - PC update
- Pipelining
  - Rearranging stages
  - Inserting pipeline registers
  - Rearranging and relabelling signals

Architecture diagram

Instructions supported

Testing and outputs

Processor features

- Processor frequency
- Memory
- Limitations

Instructions to run

Detailed descriptions and testing

- Fetch
- Decode
- Execute
  - ALU

Challenges faced

Acknowledgement

Overview

The main objective of this project is to develop a processor architecture design based on the Y86-64 ISA using Verilog. The processor should be able to execute all instructions in the Y86-64 ISA. The design approach must be modular. The goal is to produce a 5 stage pipelined Y86-64 implementation.

Module description

Sequential

Fetch

- In the fetch stage we are required to read instruction by instruction from the instruction memory and find the values of `icode`, `ifun`, `rA`, `rB` and `valC` according to the instruction.
- I have declared the instruction memory as a register array `instr_mem` in my fetch stage for the sake of convenience of simulation as recommended in the project discussions.

  ```
  reg [7:0] instr_mem[0:1023];
  ```

- The implementation and working of the fetch stage is described as follows
  - On positive edge of clock signal the fetch stage gets the updated `PC` value and checks for the instruction in the instruction memory.
  - If the `PC` is out of the bounds of the instruction memory it gives rise to an `imem_error`. In case of an `imem_error` we perform a nop operation which means all stages of the processor say idle for that clock cycle.
  - If the PC is valid then it takes 10 consecutive bytes from the instruction memory starting from the PC byte and combines it into a single 10 byte (80 bit) register `instr`. This has been done as the maximum size of a single instruction is 10 bytes.

```
instr={
      instr_mem[PC],
      instr_mem[PC+1],
      instr_mem[PC+2],
      instr_mem[PC+3],
      instr_mem[PC+4],
      instr_mem[PC+5],
      instr_mem[PC+6],
      instr_mem[PC+7],
      instr_mem[PC+8],
      instr_mem[PC+9]
    };
```

- From the instruction register the `icode` and `ifun` values can be obtained by taking the elements `instr[0:3]` and `instr[4:7]`. This is represented by the split block in the architecture diagram.
- We use `icode` to find the values of the conditions `need_regisids` and `need_valC` which represent whether the particular instruction required register IDs and `valC`.
- Depending on the value of `icode` the rest of the output values are obtained from `instr` as follows:
  - `rA` is `instr[8:11]`
  - `rA` is `instr[12:15]`
  - `valC` is `instr[8:71]` in case of jxx and call and `instr[16:79]`

  This is represented by the align block in the architecture diagram.
- valP is also decided based on icode as follows
  - `valP=PC+64'd1` for halt, nop and ret
  - `valP=PC+64'd2` for cmovxx, OPq, pushq and popq
  - `valP=PC+64'd10` for all other instructions

  The general formula for valP is

  valP=PC+1+need_regids+8(need_valC)
- If the `icode` is invalid then the `instr_valid` is set to 0.

Source code

```
`timescale 1ns / 1ps

module fetch(
  clk,PC,
  icode,ifun,rA,rB,valC,valP,instr_valid,imem_error,hlt
);

  input clk;
  input [63:0] PC;
  output reg [3:0] icode;
  output reg [3:0] ifun;
  output reg [3:0] rA;
  output reg [3:0] rB;
  output reg [63:0] valC;
  output reg [63:0] valP;
  output reg instr_valid;
  output reg imem_error;
  output reg hlt;

  reg [7:0] instr_mem[0:1023];

  reg [0:79] instr;

  initial begin
  //Instruction memory
    // //OPq
    //     instr_mem[32]=8'b01100000; //5 fn
    //     instr_mem[33]=8'b00100011; //rA rB

    //     instr_mem[34]=8'b00010000; // 1 0
    //     instr_mem[35]=8'b00010000; // 1 0
    //     instr_mem[36]=8'b00010000; // 1 0

    //   //cmovxx
    //     instr_mem[37]=8'b00100000; //2 fn
    //     instr_mem[38]=8'b00000100; //rA rB

    //     instr_mem[39]=8'b00010000; // 1 0
    //     instr_mem[40]=8'b00010000; // 1 0
    //     instr_mem[41]=8'b00010000; // 1 0
    //     instr_mem[42]=8'b00010000; // 1 0
    //     instr_mem[43]=8'b00010000; // 1 0
    //     instr_mem[44]=8'b00010000; // 1 0

    //   //halt
    //     instr_mem[45]=8'b00000000; // 0 0

  end

  always@(posedge clk)
  begin

    imem_error=0;
    if(PC>1023)
```

```
      begin
        imem_error=1;
      end

      instr={
        instr_mem[PC],
        instr_mem[PC+1],
        instr_mem[PC+2],
        instr_mem[PC+3],
        instr_mem[PC+4],
        instr_mem[PC+5],
        instr_mem[PC+6],
        instr_mem[PC+7],
        instr_mem[PC+8],
        instr_mem[PC+9]
      };

      icode= instr[0:3];
      ifun= instr[4:7];

      instr_valid=1'b1;

      if(icode==4'b0000) //halt
      begin
        hlt=1;
        valP=PC+64'd1;
      end
      else if(icode==4'b0001) //nop
      begin
        valP=PC+64'd1;
      end
      else if(icode==4'b0010) //cmovxx
      begin
        rA=instr[8:11];
        rB=instr[12:15];
        valP=PC+64'd2;
      end
      else if(icode==4'b0011) //irmovq
      begin
        rA=instr[8:11];
        rB=instr[12:15];
        valC=instr[16:79];
        valP=PC+64'd10;
      end
      else if(icode==4'b0100) //rmmovq
      begin
        rA=instr[8:11];
        rB=instr[12:15];
        valC=instr[16:79];
        valP=PC+64'd10;
      end
      else if(icode==4'b0101) //mrmovq
      begin
        rA=instr[8:11];
        rB=instr[12:15];
        valC=instr[16:79];
        valP=PC+64'd10;
      end
```

```
    else if(icode==4'b0110) //OPq
    begin
      rA=instr[8:11];
      rB=instr[12:15];
      valP=PC+64'd2;
    end
    else if(icode==4'b0111) //jxx
    begin
      valC=instr[8:71];
      valP=PC+64'd9;
    end
    else if(icode==4'b1000) //call
    begin
      valC=instr[8:71];
      valP=PC+64'd9;
    end
    else if(icode==4'b1001) //ret
    begin
      valP=PC+64'd1;
    end
    else if(icode==4'b1010) //pushq
    begin
      rA=instr[8:11];
      rB=instr[12:15];
      valP=PC+64'd2;
    end
    else if(icode==4'b1011) //popq
    begin
      rA=instr[8:11];
      rB=instr[12:15];
      valP=PC+64'd2;
    end
    else
    begin
      instr_valid=1'b0;
    end
  end

endmodule
```

Decode and write back

- In the decode stage we are required to output valA and valB based on icode, rA and rB.
- I have created a register array which represents the register memory of the processor as follows within the decode and write back module for the sake of convenience of simulation as these are the only 2 stages that require access to the register memory

```
reg [63:0] reg_mem[0:14];
```

- The implementation and working of the decode stage is described as follows:
    - The values of `valA` and `valB` are present in the registers given by address `rA` and `rB`. We use an `rA` and `rB` and index and reference the `reg_mem` register array for `valA` and `valB`.
    - Depending on the `icode` of the instructions we decide the calues of valA and valB as follows
        - `valA=reg_mem[rA]` for cmovxx
        - `valA=reg_mem[rA]` and `valB=reg_mem[rB]` for rmmovq and OPq
        - `valB=reg_mem[rB]` for mrmovq
        - `valB=reg_mem[4]` for call
        - `valA=reg_mem[4]` and `valB=reg_mem[4]` for ret and popq
        - `valA=reg_mem[rA]` and `valB=reg_mem[4]` for pushq

    💡 Here `reg_mem[4]` represents the rsp register

    These operations are represented by the scrA, srcB and register file blocks in the architecture diagram.

- The implementation and working of the write back stage is described as follows:
    - The values `valE` or `valM` have to be written into `rA`, `rB` or `rsp` according to `icode` of instruction.
    - Depending on icode the write back is done as follows
        - `reg_mem[rB]=valE` if `cnd` is 1 for cmovxx
        - `reg_mem[rB]=valE` for irmovq and OPq
        - `reg_mem[rA]=valM` for mrmovq
        - `reg_mem[4]=valE` for call, ret and pushq
        - `reg_mem[4]=valE` and `reg_mem[rA]=valM` for popq

💡 Here `reg_mem[4]` represents the rsp register

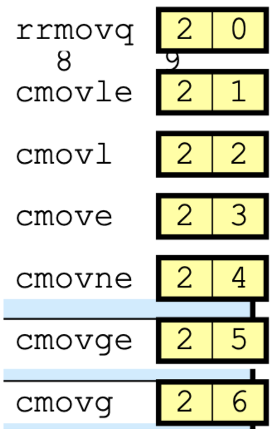
These operations are represented by the dstE, dstM and register file blocks in the architecture diagram.

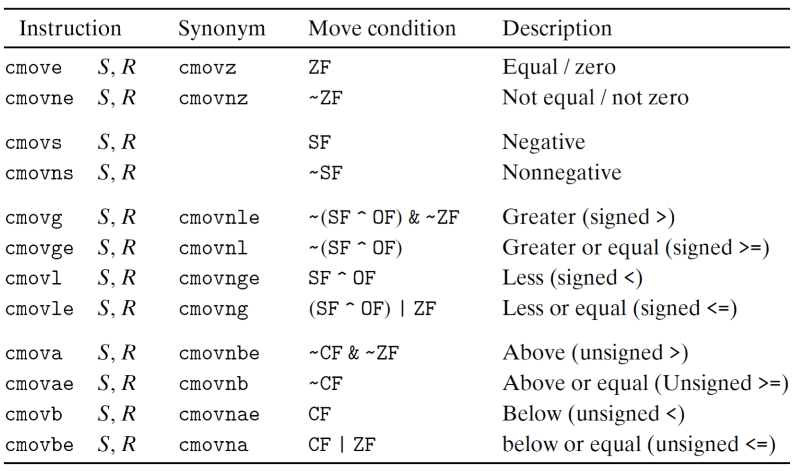
Source code

```
`timescale 1ns / 1ps

module decode_wb(
  clk,icode,rA,rB,cnd,
  valA,valB,
  valE,valM,
  reg_mem0,reg_mem1,reg_mem2,reg_mem3,reg_mem4,reg_mem5,
  reg_mem6,reg_mem7,reg_mem8,reg_mem9,reg_mem10,reg_mem11,
  reg_mem12,reg_mem13,reg_mem14
);

  input clk;
  input cnd;
  input [3:0] icode;
  input [3:0] rA;
  input [3:0] rB;
  output reg [63:0] valA;
  output reg [63:0] valB;
  input [63:0] valE;
  input [63:0] valM;

  output reg [63:0] reg_mem0;
  output reg [63:0] reg_mem1;
  output reg [63:0] reg_mem2;
  output reg [63:0] reg_mem3;
  output reg [63:0] reg_mem4;
  output reg [63:0] reg_mem5;
  output reg [63:0] reg_mem6;
  output reg [63:0] reg_mem7;
  output reg [63:0] reg_mem8;
  output reg [63:0] reg_mem9;
  output reg [63:0] reg_mem10;
  output reg [63:0] reg_mem11;
  output reg [63:0] reg_mem12;
  output reg [63:0] reg_mem13;
  output reg [63:0] reg_mem14;

  reg [63:0] reg_mem[0:14];

  initial begin
    reg_mem[0]=64'd0;
    reg_mem[1]=64'd1;
    reg_mem[2]=64'd2;
    reg_mem[3]=64'd3;
    reg_mem[4]=64'd4;
    reg_mem[5]=64'd5;
    reg_mem[6]=64'd6;
    reg_mem[7]=64'd7;
    reg_mem[8]=64'd8;
    reg_mem[9]=64'd9;
    reg_mem[10]=64'd10;
    reg_mem[11]=64'd11;
    reg_mem[12]=64'd12;
    reg_mem[13]=64'd13;
    reg_mem[14]=64'd14;
  end

  //decode
  always@(*)
  begin
    if(icode==4'b0010) //cmovxx
    begin
      valA=reg_mem[rA];
    end
    else if(icode==4'b0100) //rmmovq
    begin
      valA=reg_mem[rA];
      valB=reg_mem[rB];
    end
    else if(icode==4'b0101) //mrmovq
    begin
      valB=reg_mem[rB];
    end
    else if(icode==4'b0110) //OPq
    begin
      valA=reg_mem[rA];
      valB=reg_mem[rB];
    end
    else if(icode==4'b1000) //call
    begin
      valB=reg_mem[4]; //rsp
    end
    else if(icode==4'b1001) //ret
    begin
      valA=reg_mem[4]; //rsp
      valB=reg_mem[4]; //rsp
    end
    else if(icode==4'b1010) //pushq
    begin
      valA=reg_mem[rA];
      valB=reg_mem[4]; //rsp
    end
    else if(icode==4'b1011) //popq
    begin
      valA=reg_mem[4]; //rsp
      valB=reg_mem[4]; //rsp
    end
    reg_mem0=reg_mem[0];
    reg_mem1=reg_mem[1];
    reg_mem2=reg_mem[2];
    reg_mem3=reg_mem[3];
    reg_mem4=reg_mem[4];
    reg_mem5=reg_mem[5];
    reg_mem6=reg_mem[6];
    reg_mem7=reg_mem[7];
    reg_mem8=reg_mem[8];
    reg_mem9=reg_mem[9];
    reg_mem10=reg_mem[10];
    reg_mem11=reg_mem[11];
    reg_mem12=reg_mem[12];
    reg_mem13=reg_mem[13];
    reg_mem14=reg_mem[14];
  end

  //write_back
  always@(negedge clk)
  begin
    if(icode==4'b0010) //cmovxx
    begin
      if(cnd==1'b1)
      begin
        reg_mem[rB]=valE;
      end
    end
    else if(icode==4'b0011) //irmovq
    begin
      reg_mem[rB]=valE;
    end
    else if(icode==4'b0101) //mrmovq
    begin
      reg_mem[rA]=valM;
    end
    else if(icode==4'b0110) //OPq
    begin
      reg_mem[rB]=valE;
    end
    else if(icode==4'b1000) //call
    begin
      reg_mem[4]=valE;
    end
    else if(icode==4'b1001) //ret
    begin
      reg_mem[4]=valE;
    end
    else if(icode==4'b1010) //pushq
    begin
      reg_mem[4]=valE;
    end
    else if(icode==4'b1011) //popq
    begin
      reg_mem[4]=valE;
      reg_mem[rA]=valM;
    end
    reg_mem0=reg_mem[0];
    reg_mem1=reg_mem[1];
    reg_mem2=reg_mem[2];
    reg_mem3=reg_mem[3];
    reg_mem4=reg_mem[4];
    reg_mem5=reg_mem[5];
    reg_mem6=reg_mem[6];
    reg_mem7=reg_mem[7];
    reg_mem8=reg_mem[8];
    reg_mem9=reg_mem[9];
    reg_mem10=reg_mem[10];
    reg_mem11=reg_mem[11];
    reg_mem12=reg_mem[12];
    reg_mem13=reg_mem[13];
    reg_mem14=reg_mem[14];
  end

endmodule
```

Execute

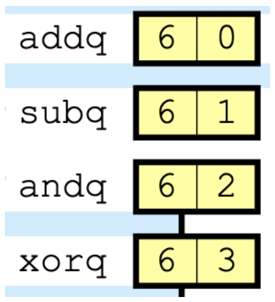
- The execute stage includes the ALU.
- Based on `icode` and `ifun` if the instruction requires an `ifun` we can decide the value to be assigned to `valE` accordingly.
- The implementation and working of the execute stage is as follows:
  - For cmov we need to check the following move conditions according to `ifun` and set `cnd` to 1 or leave it as zero accordingly



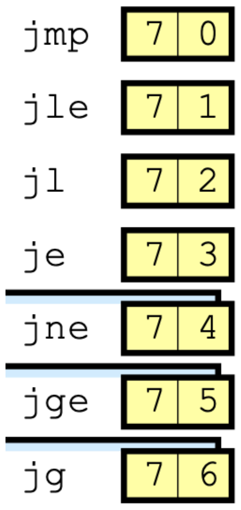


Instruction		Synonym	Move condition	Description
cmove	*S*, *R*	cmovz	ZF	Equal / zero
cmovne	*S*, *R*	cmovnz	~ZF	Not equal / not zero
cmovs	*S*, *R*		SF	Negative
cmovns	*S*, *R*		~SF	Nonnegative
cmovg	*S*, *R*	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge	*S*, *R*	cmovnl	~(SF ^ OF)	Greater or equal (signed >=)
cmovl	*S*, *R*	cmovnge	SF ^ OF	Less (signed <)
cmovle	*S*, *R*	cmovng	(SF ^ OF) | ZF	Less or equal (signed <=)
cmova	*S*, *R*	cmovnbe	~CF & ~ZF	Above (unsigned >)
cmovae	*S*, *R*	cmovnb	~CF	Above or equal (Unsigned >=)
cmovb	*S*, *R*	cmovnae	CF	Below (unsigned <)
cmovbe	*S*, *R*	cmovna	CF | ZF	below or equal (unsigned <=)

  - `valE=valC` for irmovq
  - `valE=valB+valC` for mrmovq and rmmovq
  - For OPx we need to again check the `ifun` values and use ALU to perform the required operation and `valE=ans` where ans is the ALU output



- For jXX also we need to check the ifun value and check if the jump conditions are satisfied. The jump conditions are similar to the move conditions.



- `valE=-64'd8+valB` for call and pushq
- `valE=64'd8+valB` for ret and popq

Source code

```
`timescale 1ns / 1ps

`include "./ALU/alu.v"

module execute(
  clk,icode,ifun,valA,valB,valC,
  valE,cnd,zf,sf,of
);

  input clk;
  input [3:0] icode;
  input [3:0] ifun;
  input [63:0] valA;
  input [63:0] valB;
```

```
    input [63:0] valC;

    output reg [63:0] valE;
    output reg cnd;

    output reg zf;
    output reg sf;
    output reg of;

    always@(*)
    begin
      if(icode==4'b0110 && clk==1)
      begin
        zf=(ans==1'b0);
        sf=(ans<1'b0);
        of=(a<1'b0==b<1'b0)&&(ans<1'b0!=a<1'b0);
      end
    end

    initial begin
      zf=0;
      sf=0;
      of=0;
    end

    reg signed [63:0]anss;
    reg [1:0]control;
    reg signed [63:0]a;
    reg signed [63:0]b;

    wire signed [63:0]ans;
    wire overflow;

    alu alu1(
      .control(control),
      .a(a),
      .b(b),
      .ans(ans),
      .overflow(overflow)
    );


    reg xin1;
    reg xin2;
    reg oin1;
    reg oin2;
    reg ain1;
    reg ain2;
    reg nin1;
    wire xout;
    wire oout;
    wire aout;
    wire nout;

    xor g1(xout,xin1,xin2);
    or g2(oout,oin1,oin2);
    and g3(aout,ain1,ain2);
    not g4(nout,nin1);
```

```
initial
begin
  control=2'b00;
  a = 64'b0;
  b = 64'b0;
end

always@(*)
begin
  if(clk==1)
  begin
    cnd=0;
    if(icode==4'b0010) //cmovxx
    begin
      if(ifun==4'b0000)//rrmovq
      begin
        cnd=1;
      end
      else if(ifun==4'b0001)//cmovle
      begin
      // (sf^of)||zf
        xin1=sf;
        xin2=of;
        if(xout)
        begin
          cnd=1;
        end
        else if(zf)
        begin
          cnd=1;
        end
      end
      else if(ifun==4'b0010)//cmovl
      begin
      // sf^of
        xin1=sf;
        xin2=of;
        if(xout)
        begin
          cnd=1;
        end
      end
      else if(ifun==4'b0011)//cmove
      begin
      // zf
        if(zf)
        begin
          cnd=1;
        end
      end
      else if(ifun==4'b0100)//cmovne
      begin
      // !zf
        nin1=zf;
        if(nout)
        begin
          cnd=1;
        end
      end
      else if(ifun==4'b0101)//cmovge
      begin
      // !(sf^of)
        xin1=sf;
        xin2=of;
        nin1=xout;
        if(nout)
        begin
          cnd=1;
        end
      end
      else if(ifun==4'b0110)//cmovg
      begin
      //!(sf^of)) && (!zf)
        xin1=sf;
        xin2=of;
        nin1=xout;
        if(nout)
        begin
          nin1=zf;
          if(nout)
          begin
            cnd=1;
          end
        end
      end
      valE=64'd0+valA;
    end
    else if(icode==4'b0011) //irmovq
    begin
      valE=64'd0+valC;
    end
    else if(icode==4'b0100) //rmmovq
    begin
      valE=valB+valC;
    end
    else if(icode==4'b0101) //mrmovq
    begin
      valE=valB+valC;
    end
    else if(icode==4'b0110) //OPq
    begin
      if(ifun==4'b0000) //add
      begin
        //valE=valA+valB;
        control=2'b00;
        a = valA;
        b = valB;
      end
      else if(ifun==4'b0001) //sub
      begin
        //valE=valA-valB;
        control=2'b01;
        a = valB;
        b = valA;
      end
      else if(ifun==4'b0010) //and
      begin
        //valE=valA.valB;
        control=2'b10;
        a = valA;
        b = valB;
      end
      else if(ifun==4'b0011) //xor
      begin
        //valE=valA^valB;
        control=2'b11;
        a = valA;
        b = valB;
      end
      assign anss=ans;
      valE=anss;
    end
    if(icode==4'b0111) //jxx
    begin
      if(ifun==4'b0000)//jmp
      begin
        cnd=1;
      end
      else if(ifun==4'b0001)//jle
      begin
      // (sf^of)||zf
        xin1=sf;
        xin2=of;
        if(xout)
        begin
          cnd=1;
        end
        else if(zf)
        begin
          cnd=1;
        end
      end
      else if(ifun==4'b0010)//jl
      begin
      // sf^of
        xin1=sf;
        xin2=of;
        if(xout)
        begin
          cnd=1;
        end
      end
      else if(ifun==4'b0011)//je
      begin
      // zf
        if(zf)
        begin
          cnd=1;
        end
      end
      else if(ifun==4'b0100)//jne
      begin
      // !zf
```

```
          nin1=zf;
          if(nout)
          begin
            cnd=1;
          end
        end
        else if(ifun==4'b0101)//jge
        begin
        // !(sf^of)
          xin1=sf;
          xin2=of;
          nin1=xout;
          if(nout)
          begin
            cnd=1;
          end
        end
        else if(ifun==4'b0110)//jg
        begin
        //!(sf^of)) && (!zf)
          xin1=sf;
          xin2=of;
          nin1=xout;
          if(nout)
          begin
            nin1=zf;
            if(nout)
            begin
              cnd=1;
            end
          end
        end
      end
      if(icode==4'b1000) //call
      begin
        valE=-64'd8+valB;
      end
      if(icode==4'b1001) //ret
      begin
        valE=64'd8+valB;
      end
      if(icode==4'b1010) //pushq
      begin
        valE=-64'd8+valB;
      end
      if(icode==4'b1011) //popq
      begin
        valE=64'd8+valB;
      end
    end
  end

endmodule
```

Memory

- The memory stage is responsible for reading and writing to memory.
- I have declared the data memory as a register array in the memory stage as access to the data memory is only required here.

  ```
  reg [63:0] data_mem[0:255];
  ```

- Based on the `icode` we can decide whether we are required to read or write from or to memory respectively.
- The data address is calculated using icode, valE and valA and the data input is calculated using icode, valA and valP.
- The implementation and working of the memory stage is described as follows:
  - For rmmovq, call, pushq we need to write to memory.
    - `data_mem[valE]=valA` for rmmovq and pushq
    - `data_mem[valE]=valP` for call
  - For mrmovq, ret and popq we need to read from memory.
    - `valM=data_mem[valE]` for mrmovq and popq
    - `valM=data_mem[valA]` for ret

  These operations are represented by the memory read, memory write, memory address and memory data blocks in the architecure

Source code

```
`timescale 1ns / 1ps

module memory(
  clk,icode,valA,valB,valE,valP,valM,datamem
);

  input clk;

  input [3:0] icode;
  input [63:0] valA;
  input [63:0] valB;
  input [63:0] valE;
  input [63:0] valP;

  output reg [63:0] valM;
```

```
  output reg [63:0] datamem;

  reg [63:0] data_mem[0:1023];

  always@(*)
  begin
    if(icode==4'b0100) //rmmovq
    begin
      data_mem[valE]=valA;
    end
    if(icode==4'b0101) //mrmovq
    begin
      valM=data_mem[valE];
    end
    if(icode==4'b1000) //call
    begin
      data_mem[valE]=valP;
    end
    if(icode==4'b1001) //ret
    begin
      valM=data_mem[valA];
    end
    if(icode==4'b1010) //pushq
    begin
      data_mem[valE]=valA;
    end
    if(icode==4'b1011) //popq
    begin
      valM=data_mem[valE];
    end
    datamem=data_mem[valE];
  end

endmodule
```

PC update

- The PC update module is responsible for finding the next value of PC after an instruction has finished executing.
- The PC is updated by the instruction length in bytes for all except jXX, call and ret.
- The implementation and working of the PC update stage is as follows:
    - `updated_pc=valC` for jXX if cnd is 1 and `updated_pc=valP` otherwise
    - `updated_pc=valC` for call
    - `updated_pc=valM` for ret
    - `updated_pc=valP` for all other instructions

Source code

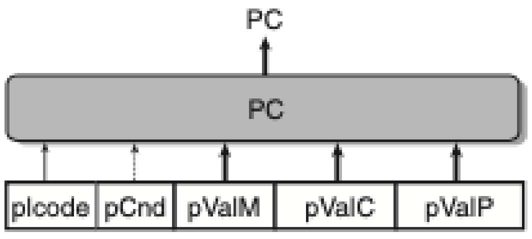
```
`timescale 1ns / 1ps

module pc_update(
  clk,PC,cnd,icode,valC,valM,valP,
  updated_pc
);
  input clk;
  input cnd;
  input [3:0] icode;
  input [63:0] valC;
  input [63:0] valP;
  input [63:0] valM;
  input [63:0] PC;
  output reg [63:0] updated_pc;

  always@(*)
  begin
    if(icode==4'b0111) //jxx
    begin
      if(cnd==1'b1)
      begin
        updated_pc=valC;
      end
      else
      begin
        updated_pc=valP;
      end
    end
    else if(icode==4'b1000) //call
    begin
      updated_pc=valC;
    end
    else if(icode==4'b1001) //ret
    begin
      updated_pc=valM;
    end
    else
    begin
      updated_pc=valP;
    end
  end

endmodule
```

Pipelining

- The first step to pipelining is the rearrangement of computation stages.

Rearranging stages

- The PC update stage in the SEQ implementation is the last stage in the cycle of an instruction.
- For the pipelined implementation we should bring the PC update stage to the beginning of the cycle as we want to be able to continuously fetch the next instruction without having to wait for the PC update stage of the previous instruction to end had it been at the end of the cycle. This is known as circuit retiming. This changes the general sentation of the circuit without affecting its local behavior. This also allows us to balance the delays between stages in the pipelined system.
- Now the PC update stage at the beginning of the cycle can keep providing updated PC values to the fetch stage using the required values from different stages from instructions that have passed that stage.



Inserting pipeline registers

- The next step to pipelining is inserting the pipeline registers.
- We know that in a pipelined implementation we rearrange some of the hardware and signals in the SEQ implmentation and insert pipeline register between each stage.
- These registers stop the signals from one stage from flowing into the next stage and affecting the processing happening there.
- `F` the register inserted before the fetch stage holds a predicted value of the program counter.
- `D` sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.
- `E` sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.

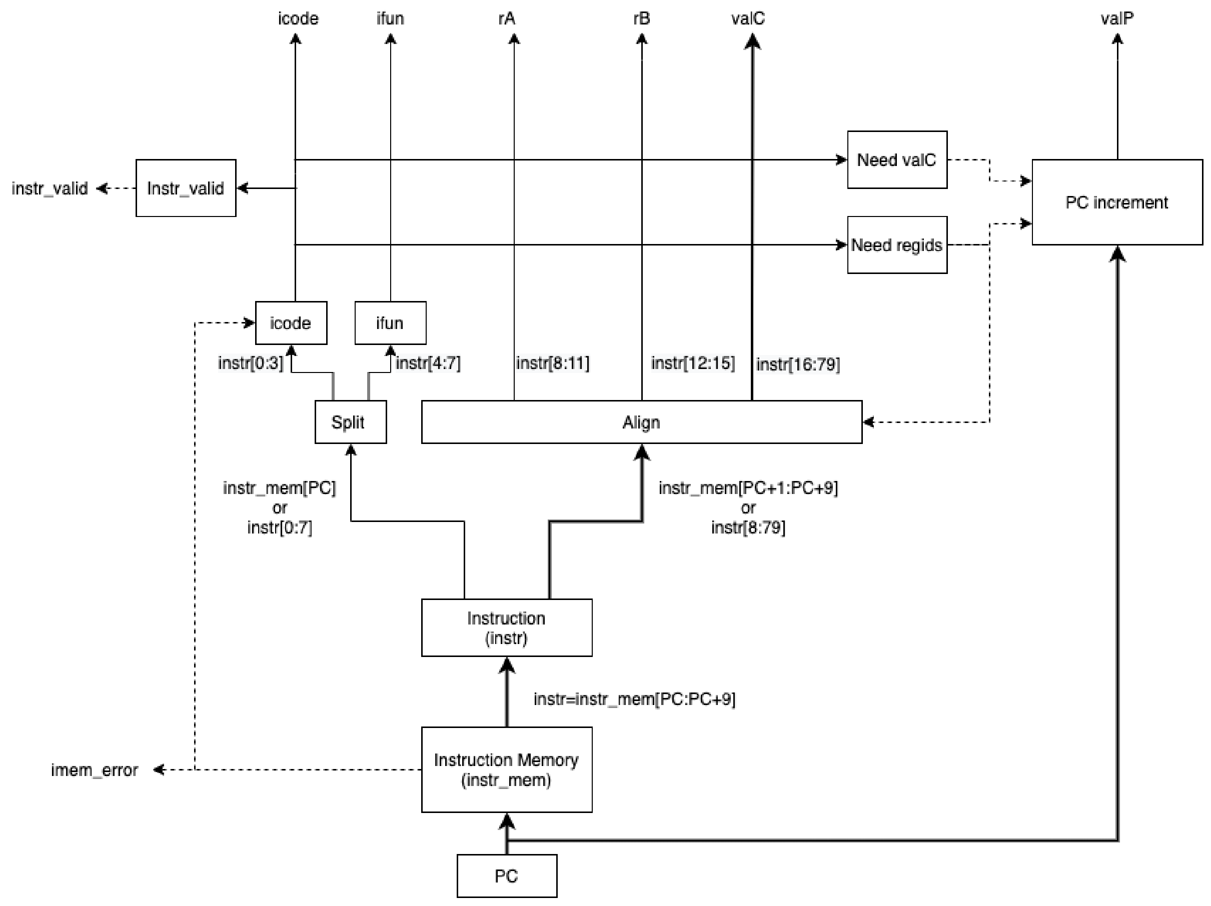
- `M` sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.
- `W` sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a ret instruction.

Rearranging and relabelling signals

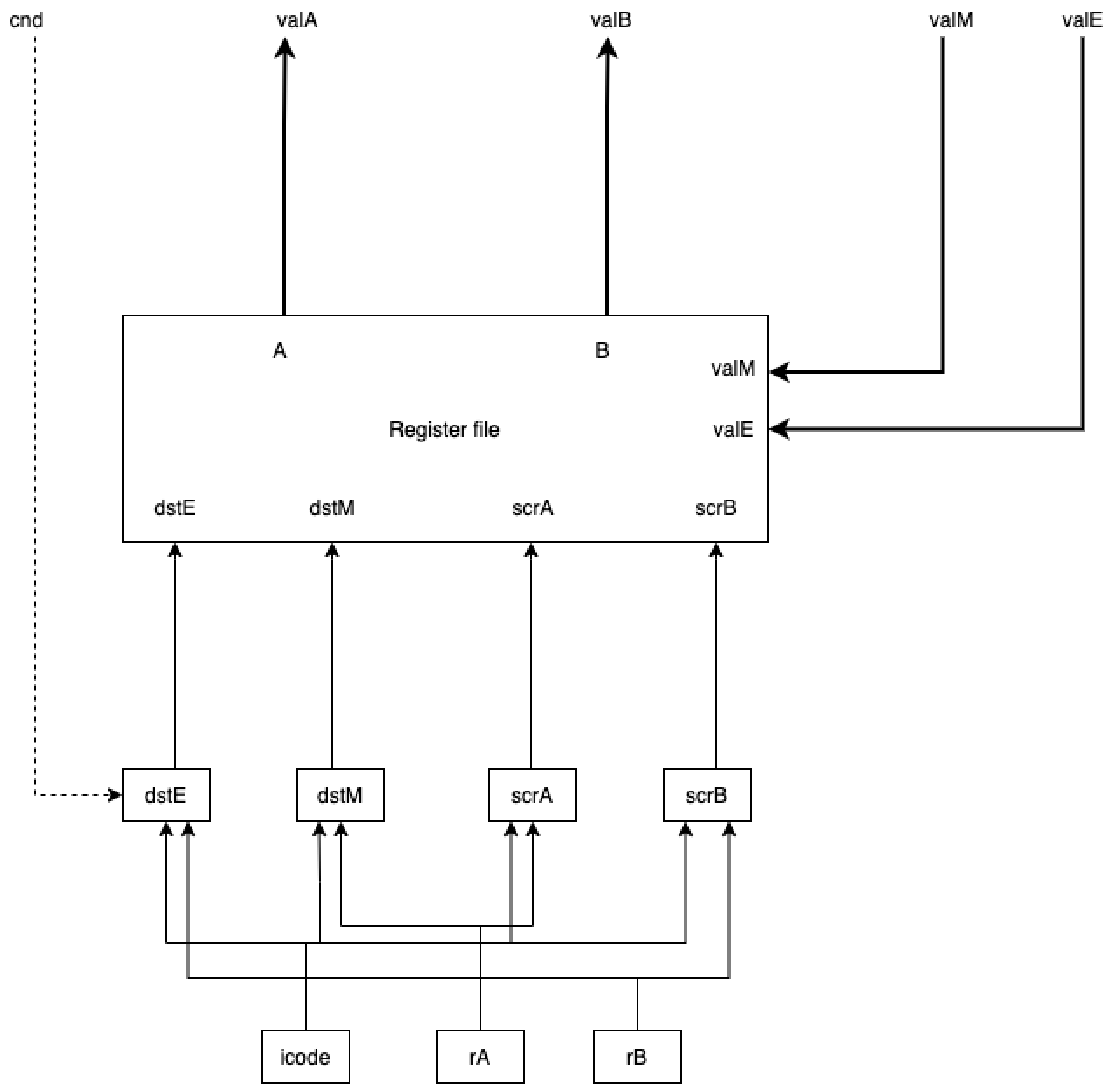
- In the pipelined implementation we will have all the signals of an instruction pass through every stage one by one and these will have to be names with respect to the stage it is currently in as it is not possible to have one signal icode and have ti account for all the 5 instructions running at the same time.
- So we maintain the signal at each stage and label them with respect to the stage as f_icode,d_icode,w_icode,etc.

Architecture diagram

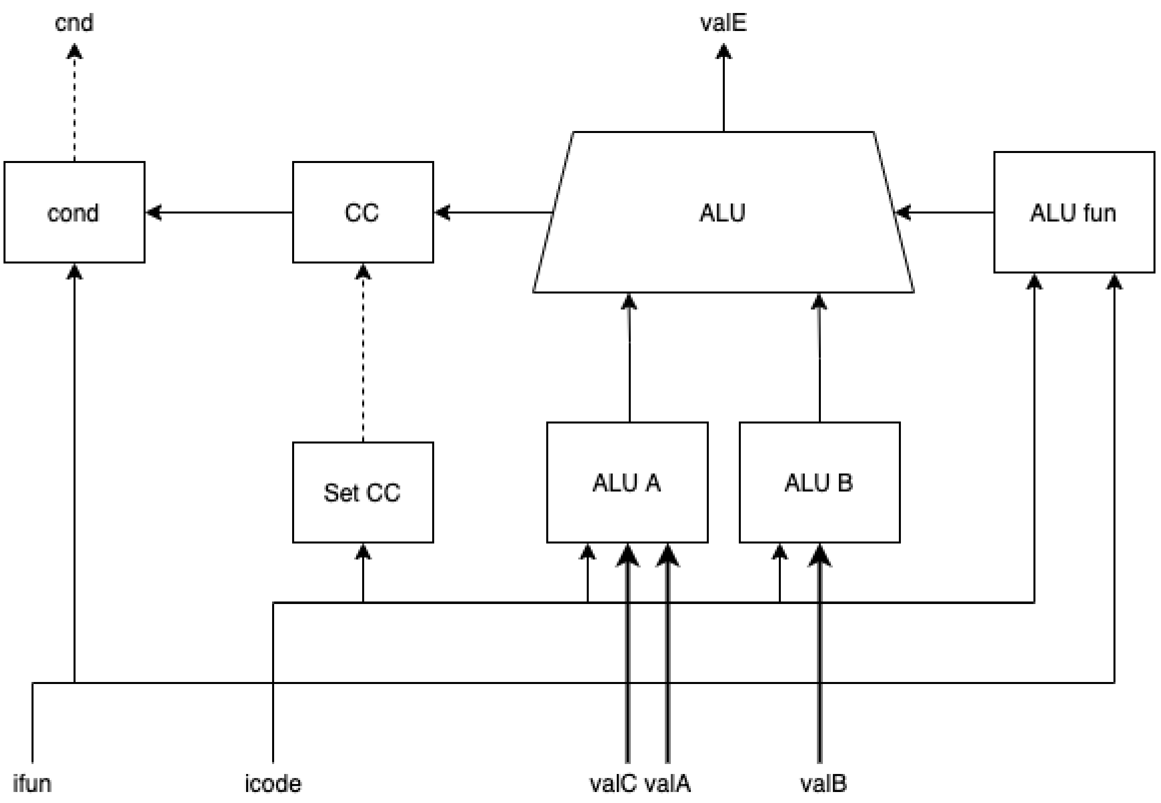
Fetch stage



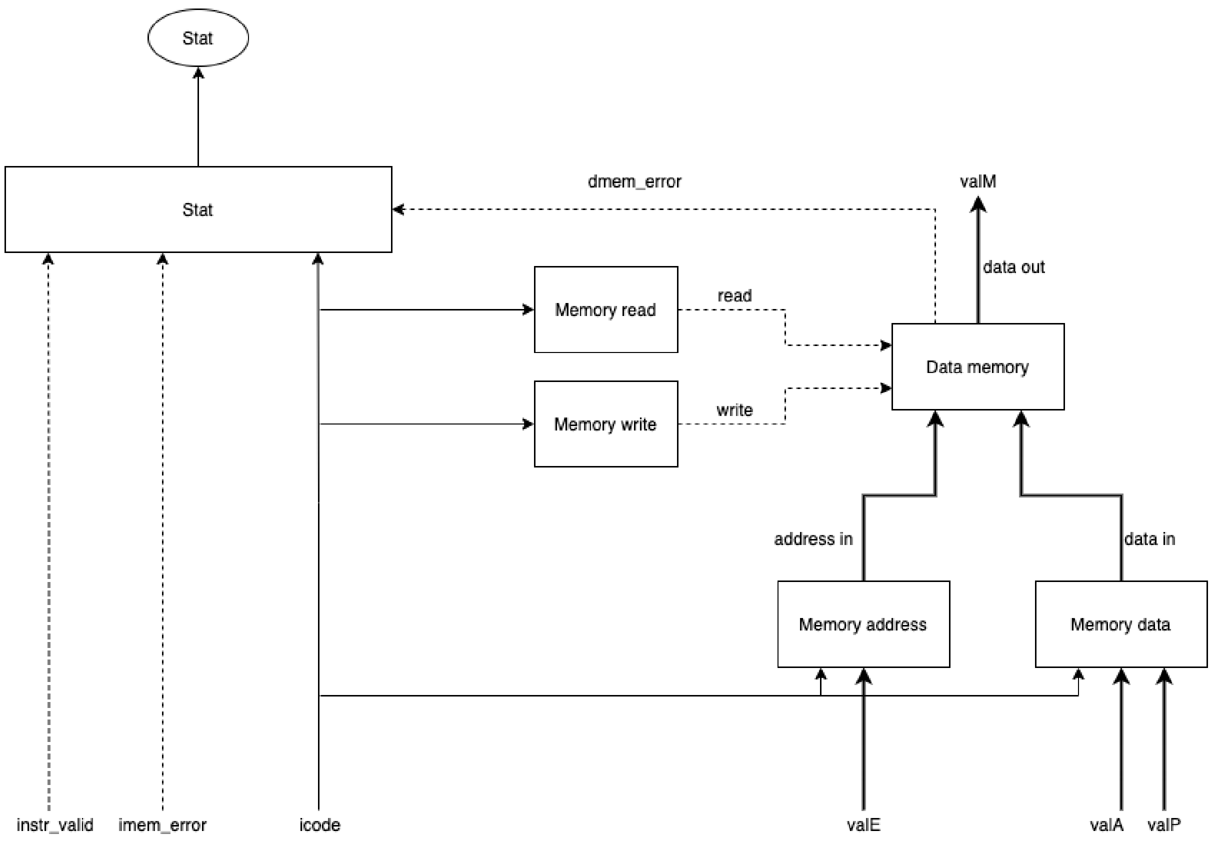
Decode and write back stages



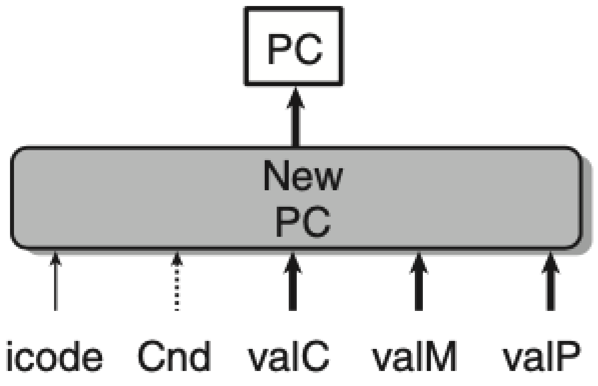
Execute stage



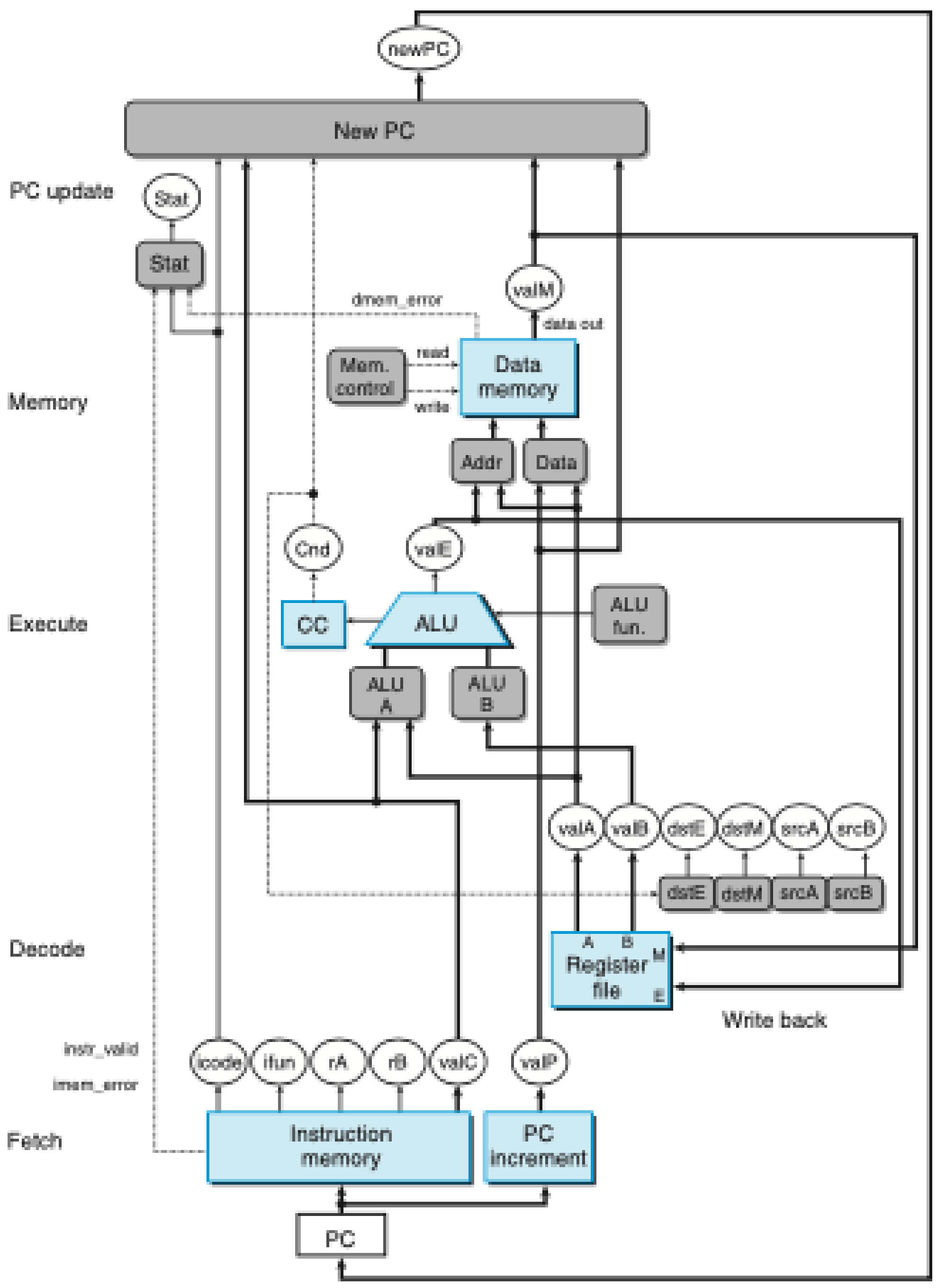
Memory stage



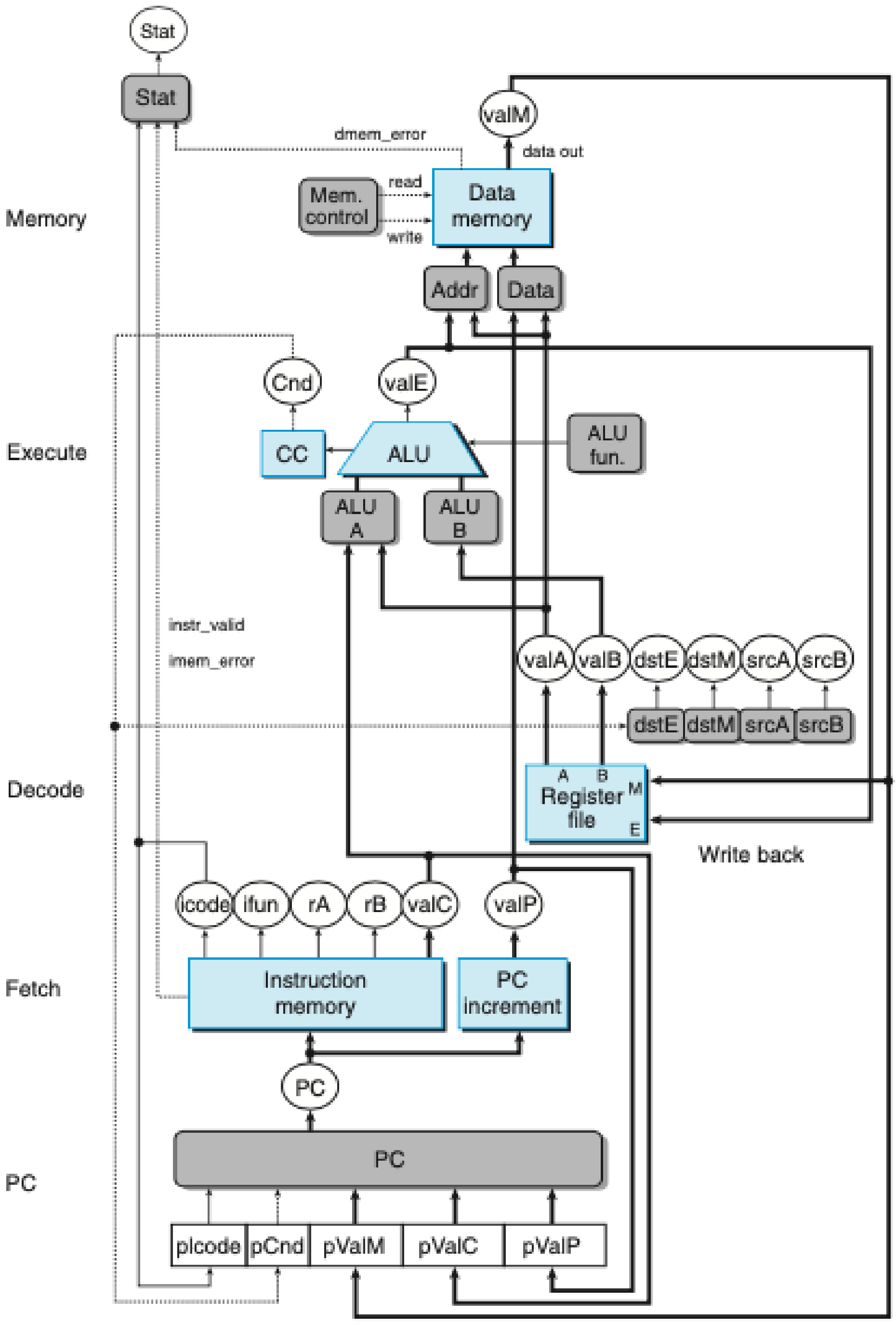
PC update stage



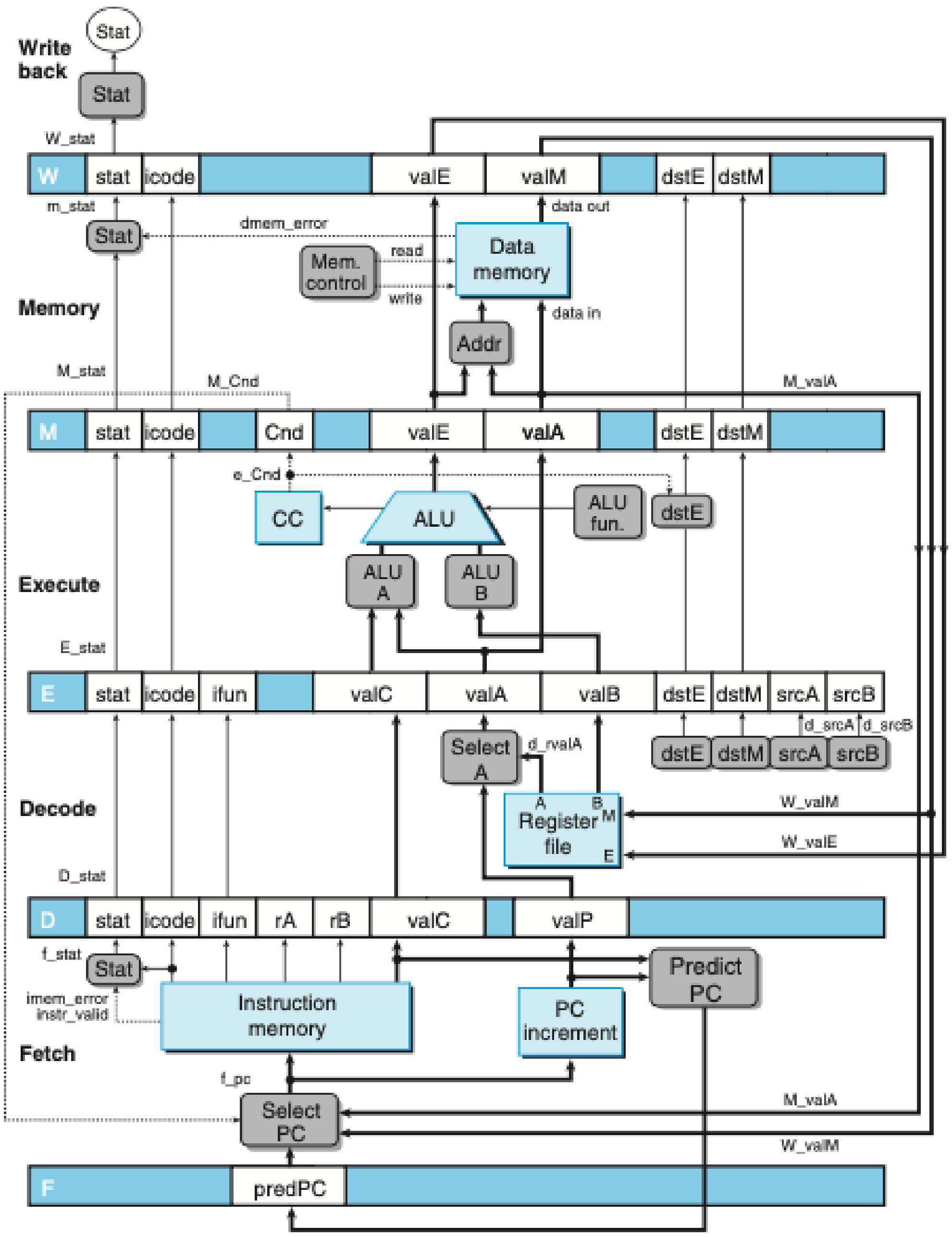
Sequential processor (SEQ) hardware structure (P.T.O)



Sequential procesor (SEQ+) hardware structure



Pipelined processor hardware structure



Instructions supported

The supported instructions are

- halt
- nop
- cmovXX
- irmovq

- rmmovq
- OPq
- jXX
- call
- ret
- pushq
- popq

The processor supports all instructions in the Y86-64 instruction set.

Testing and outputs

The initial functioning of the sequential processor and pipelined processor has been tested and verified using the following set of instructions

```
addq %rdx, %rbx
nop
nop
rrmovq %rax, %rsp
nop
nop
nop
halt
```

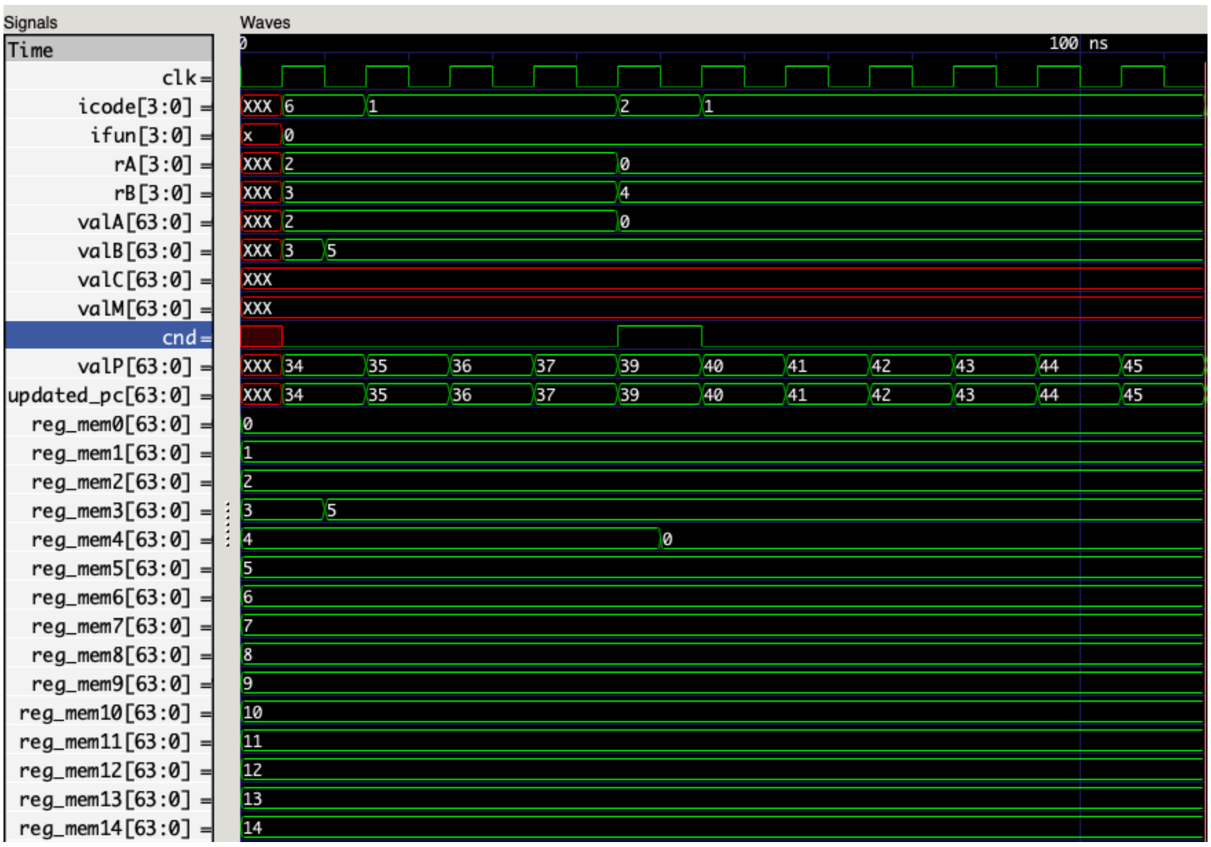
These can be encoded in binary as follows

```
//addq
    instr_mem[32]=8'b01100000; //5 fn
    instr_mem[33]=8'b00100011; //rA rB
//nop
    instr_mem[34]=8'b00010000; // 1 0
    instr_mem[35]=8'b00010000; // 1 0
//rrmovq
    instr_mem[36]=8'b00100000; //2 fn
    instr_mem[37]=8'b00000100; //rA rB
//nop
    instr_mem[38]=8'b00010000; // 1 0
    instr_mem[39]=8'b00010000; // 1 0
    instr_mem[40]=8'b00010000; // 1 0
//halt
    instr_mem[41]=8'b00000000; // 0 0
```

The outputs from the sequential processor are as follows

```
clk=0 icode=xxxx ifun=xxxx rA=xxxx rB=xxxx valA=                   x valB=
x valC=                   x valE=                   x valM=                   x insval
=x memerr=x cnd=x halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   4 zf=0 sf=0 of=0
clk=1 icode=0110 ifun=0000 rA=0010 rB=0011 valA=                   2 valB=
3 valC=                   x valE=                   5 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   4 zf=0 sf=0 of=0
clk=0 icode=0110 ifun=0000 rA=0010 rB=0011 valA=                   2 valB=
5 valC=                   x valE=                   5 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   4 zf=0 sf=0 of=0
clk=1 icode=0001 ifun=0000 rA=0010 rB=0011 valA=                   2 valB=
5 valC=                   x valE=                   5 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   4 zf=0 sf=0 of=0
clk=0 icode=0001 ifun=0000 rA=0010 rB=0011 valA=                   2 valB=
5 valC=                   x valE=                   5 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   4 zf=0 sf=0 of=0
clk=1 icode=0001 ifun=0000 rA=0010 rB=0011 valA=                   2 valB=
5 valC=                   x valE=                   5 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   4 zf=0 sf=0 of=0
clk=0 icode=0001 ifun=0000 rA=0010 rB=0011 valA=                   2 valB=
5 valC=                   x valE=                   5 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   4 zf=0 sf=0 of=0
clk=1 icode=0001 ifun=0000 rA=0010 rB=0011 valA=                   2 valB=
5 valC=                   x valE=                   5 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   4 zf=0 sf=0 of=0
clk=0 icode=0001 ifun=0000 rA=0010 rB=0011 valA=                   2 valB=
5 valC=                   x valE=                   5 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   4 zf=0 sf=0 of=0
clk=1 icode=0010 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=1 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   4 zf=0 sf=0 of=0
clk=0 icode=0010 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=1 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=1 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=0 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=1 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
```

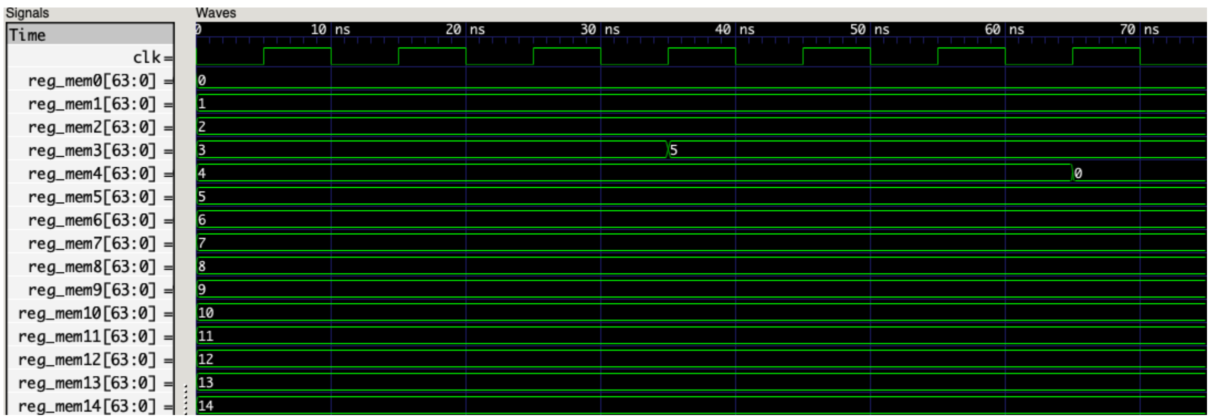
```
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=0 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=1 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=0 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=1 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=0 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=1 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=0 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=1 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=0 icode=0001 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
clk=1 icode=0000 ifun=0000 rA=0000 rB=0100 valA=                   0 valB=
5 valC=                   x valE=                   0 valM=                   x insval
=1 memerr=0 cnd=0 halt=1 0=                   0 1=                   1 2=
2 3=                   5 4=                   0 zf=0 sf=0 of=0
```



The input instruction values and output register values are clearly visible above.

The outputs from the pipelined processor are as follows

```
clk=0 halt=0 0=                   0 1=                   1 2=                   2 3=
3 4=                    4
clk=1 halt=0 0=                   0 1=                   1 2=                   2 3=
3 4=                    4
clk=0 halt=0 0=                   0 1=                   1 2=                   2 3=
3 4=                    4
clk=1 halt=0 0=                   0 1=                   1 2=                   2 3=
3 4=                    4
clk=0 halt=0 0=                   0 1=                   1 2=                   2 3=
3 4=                    4
clk=1 halt=0 0=                   0 1=                   1 2=                   2 3=
3 4=                    4
clk=0 halt=0 0=                   0 1=                   1 2=                   2 3=
3 4=                    4
clk=1 halt=0 0=                   0 1=                   1 2=                   2 3=
5 4=                    4
clk=0 halt=0 0=                   0 1=                   1 2=                   2 3=
5 4=                    4
clk=1 halt=0 0=                   0 1=                   1 2=                   2 3=
5 4=                    4
clk=0 halt=0 0=                   0 1=                   1 2=                   2 3=
5 4=                    4
clk=1 halt=0 0=                   0 1=                   1 2=                   2 3=
5 4=                   4
clk=0 halt=0 0=                   0 1=                   1 2=                   2 3=
5 4=                   4
clk=1 halt=1 0=                   0 1=                   1 2=                   2 3=
5 4=                   0
```

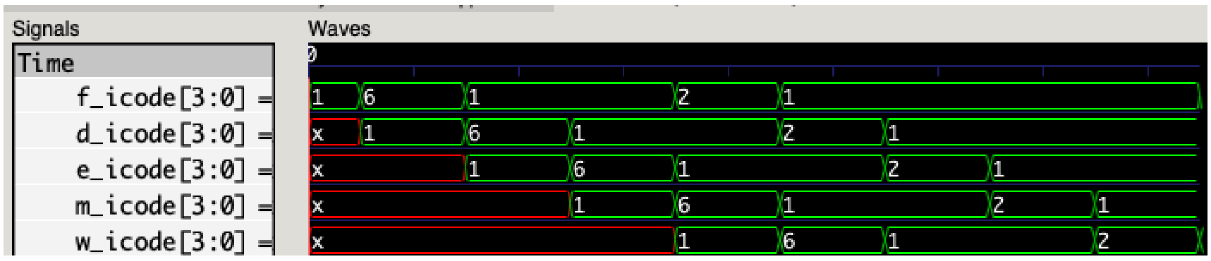


We can see that the outputs match in both cases and also match the values if calculated manually.

The pipelined processor is also tested to verify that the stages are actually functioning in a pipelined manner by printing the icode at each stage for every clock cycle and the following terminal output is obtained

```
clk=0 f= x d= x e= x m= x wb= x
clk=1 f= 6 d= x e= x m= x wb= x
clk=0 f= 6 d= x e= x m= x wb= x
clk=1 f= 1 d= 6 e= x m= x wb= x
clk=0 f= 1 d= 6 e= x m= x wb= x
clk=1 f= 1 d= 1 e= 6 m= x wb= x
clk=0 f= 1 d= 1 e= 6 m= x wb= x
clk=1 f= 2 d= 1 e= 1 m= 6 wb= x
clk=0 f= 2 d= 1 e= 1 m= 6 wb= x
clk=1 f= 1 d= 2 e= 1 m= 1 wb= 6
clk=0 f= 1 d= 2 e= 1 m= 1 wb= 6
clk=1 f= 1 d= 1 e= 2 m= 1 wb= 1
clk=0 f= 1 d= 1 e= 2 m= 1 wb= 1
clk=1 f= 1 d= 1 e= 1 m= 2 wb= 1
clk=0 f= 1 d= 1 e= 1 m= 2 wb= 1
clk=1 f= 1 d= 1 e= 1 m= 1 wb= 2
clk=0 f= 1 d= 1 e= 1 m= 1 wb= 2
clk=1 f= 0 d= 1 e= 1 m= 1 wb= 1
```

We can clearly see that an instruction propagates forward only at positive edge of clock cycle and there is no data leakage between stages or delays in propagation.



Note: The GTKWaveform was generated after running the same code with an extra nop operation in the beginning for the sake of making sure that the propagation of instructions through all the stages is visible.

The processors have also been tested on other instructions such as rmmovq,mrmovq and irmovq.

```
irmovq $0x11, %rax
rmmovq %rbp, %rdx
mrmovq %rdi, %rcx
halt
```

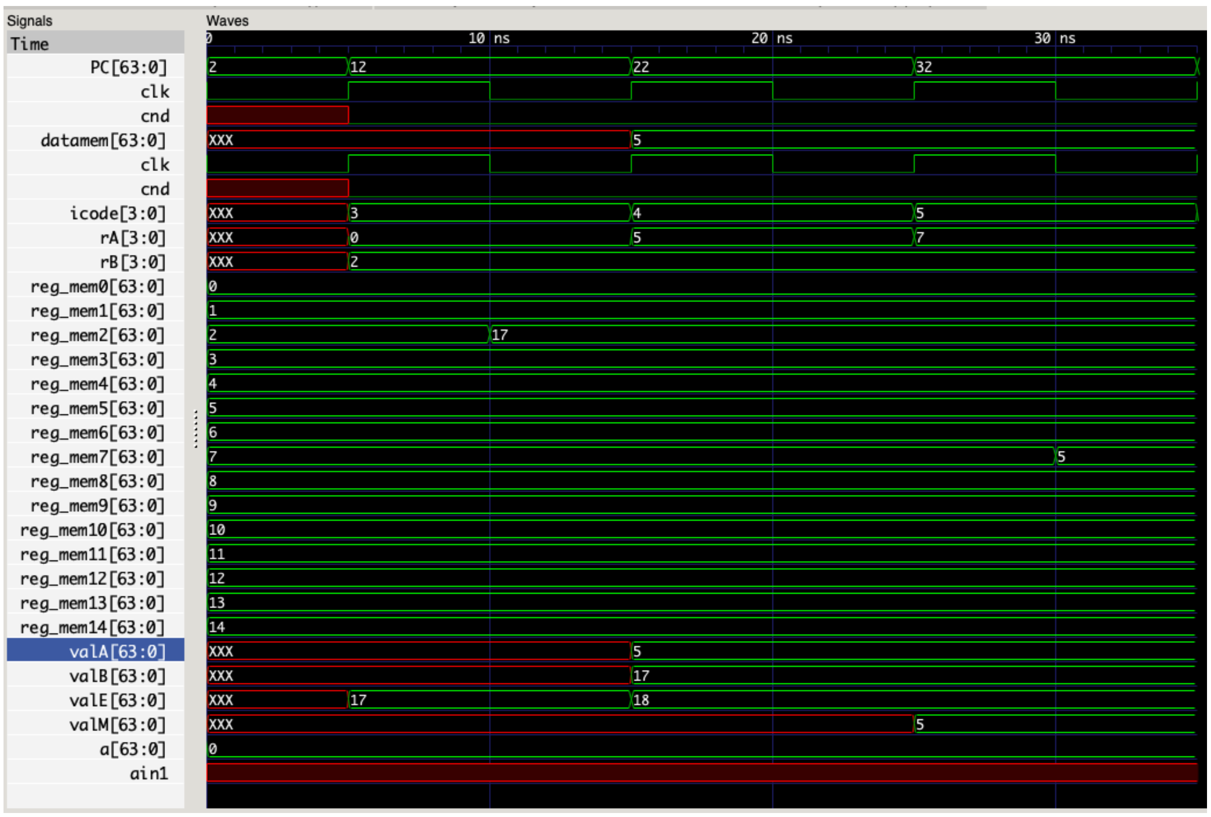
The corresponding binary encoded instruction set will be

```
//irmovq
    instr_mem[2]=8'b00110000; //3 0
    instr_mem[3]=8'b00000010; //F rB
    instr_mem[4]=8'b00000000; //V
    instr_mem[5]=8'b00000000; //V
    instr_mem[6]=8'b00000000; //V
    instr_mem[7]=8'b00000000; //V
    instr_mem[8]=8'b00000000; //V
    instr_mem[9]=8'b00000000; //V
    instr_mem[10]=8'b00000000; //V
    instr_mem[11]=8'b00010001; //V=17

  //rmmovq
    instr_mem[12]=8'b01000000; //4 0
    instr_mem[13]=8'b01010010; //rA rB
    instr_mem[14]=8'b00000000; //D
    instr_mem[15]=8'b00000000; //D
    instr_mem[16]=8'b00000000; //D
    instr_mem[17]=8'b00000000; //D
    instr_mem[18]=8'b00000000; //D
    instr_mem[19]=8'b00000000; //D
    instr_mem[20]=8'b00000000; //D
    instr_mem[21]=8'b00000001; //D

  //mrmovq
    instr_mem[22]=8'b01010000; //5 0
    instr_mem[23]=8'b01110010; //rA rB
    instr_mem[24]=8'b00000000; //D
    instr_mem[25]=8'b00000000; //D
    instr_mem[26]=8'b00000000; //D
    instr_mem[27]=8'b00000000; //D
    instr_mem[28]=8'b00000000; //D
    instr_mem[29]=8'b00000000; //D
    instr_mem[30]=8'b00000000; //D
    instr_mem[31]=8'b00000001; //D

  //halt
    instr_mem[32]=8'b00000000; // 0 0
```

The outputs of the processors are as follows

```
clk=0 icode=xxxx ifun=xxxx rA=xxxx rB=xxxx valA=                   x valB=
x valC=                   x valE=                   x valM=                   x insval
=x memerr=x cnd=x halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=1 icode=0011 ifun=0000 rA=0000 rB=0010 valA=                   x valB=
x valC=                  17 valE=                  17 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=0 icode=0011 ifun=0000 rA=0000 rB=0010 valA=                   x valB=
x valC=                  17 valE=                  17 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
17 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=1 icode=0100 ifun=0000 rA=0101 rB=0010 valA=                   5 valB=
17 valC=                   1 valE=                  18 valM=                   x insva
l=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
17 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   5
```

```
clk=0 icode=0100 ifun=0000 rA=0101 rB=0010 valA=                   5 valB=
17 valC=                   1 valE=                   18 valM=                    x insva
l=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
17 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                   11 12=                   12 13=                   13 14=
14 datamem=                   5

clk=1 icode=0101 ifun=0000 rA=0111 rB=0010 valA=                   5 valB=
17 valC=                   1 valE=                   18 valM=                    5 insva
l=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
17 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                   11 12=                   12 13=                   13 14=
14 datamem=                   5

clk=0 icode=0101 ifun=0000 rA=0111 rB=0010 valA=                   5 valB=
17 valC=                   1 valE=                   18 valM=                    5 insva
l=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
17 3=                   3 4=                   4 5=                   5 6=
6 7=                   5 8=                   8 9=                   9 10=
10 11=                   11 12=                   12 13=                   13 14=
14 datamem=                   5

clk=1 icode=0000 ifun=0000 rA=0111 rB=0010 valA=                   5 valB=
17 valC=                   1 valE=                   18 valM=                    5 insva
l=1 memerr=0 cnd=0 halt=1 0=                   0 1=                   1 2=
17 3=                   3 4=                   4 5=                   5 6=
6 7=                   5 8=                   8 9=                   9 10=
10 11=                   11 12=                   12 13=                   13 14=
14 datamem=                   5
```



The input instruction values and output register values are clearly visible above.

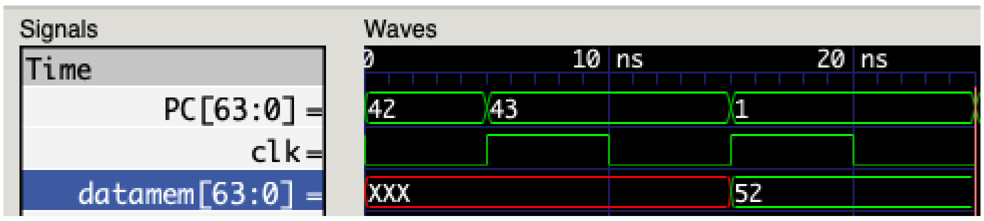
We can clearly observe that the memory and register reads and writes are being performed as expected

For testing call I used the following set of instructions

```
//call
    instr_mem[43]=8'b10000000; //8 0
    instr_mem[44]=8'b00000000; //Dest
    instr_mem[45]=8'b00000000; //Dest
    instr_mem[46]=8'b00000000; //Dest
    instr_mem[47]=8'b00000000; //Dest
    instr_mem[48]=8'b00000000; //Dest
    instr_mem[49]=8'b00000000; //Dest
    instr_mem[50]=8'b00000000; //Dest
    instr_mem[51]=8'b00000001; //Dest
```

The outputs for the sequential processor are as follows

```
clk=0 icode=xxxx ifun=xxxx rA=xxxx rB=xxxx valA=                   x valB=
x valC=                   x valE=                   x valM=                   x insval
=x memerr=x cnd=x halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   9 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=1 icode=1000 ifun=0000 rA=xxxx rB=xxxx valA=                   x valB=
9 valC=                   1 valE=                   1 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   9 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  52

clk=0 icode=1000 ifun=0000 rA=xxxx rB=xxxx valA=                   x valB=
1 valC=                   1 valE=                   1 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   1 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  52

clk=1 icode=0000 ifun=0000 rA=xxxx rB=xxxx valA=                   x valB=
1 valC=                   1 valE=                   1 valM=                   x insval
=1 memerr=0 cnd=0 halt=1 0=                   0 1=                   1 2=
2 3=                   3 4=                   1 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  52
```



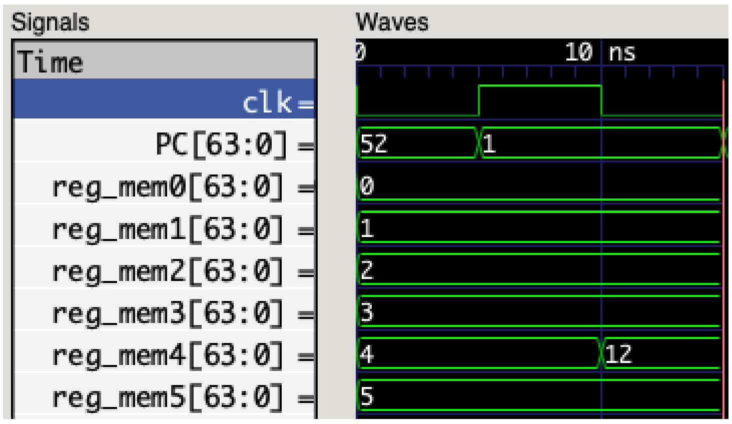
For testing ret I used the following instructions

```
//ret
    instr_mem[52]=8'b10010000; // 9 0


  //halt
    instr_mem[1]=8'b00000000; // 0 0
```

I have present the memory location to contain the address 1 and hence it halts on execution of ret.

The outputs for the sequential processor are as follows

```
clk=0 icode=xxxx ifun=xxxx rA=xxxx rB=xxxx valA=                   x valB=
x valC=                   x valE=                   x valM=                   x insval
=x memerr=x cnd=x halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=1 icode=1001 ifun=0000 rA=xxxx rB=xxxx valA=                   4 valB=
4 valC=                   x valE=                  12 valM=                   1 insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=0 icode=1001 ifun=0000 rA=xxxx rB=xxxx valA=                  12 valB=
12 valC=                   x valE=                  12 valM=                   1 insva
l=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                  12 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=1 icode=0000 ifun=0000 rA=xxxx rB=xxxx valA=                  12 valB=
12 valC=                   x valE=                  12 valM=                   1 insva
l=1 memerr=0 cnd=0 halt=1 0=                   0 1=                   1 2=
2 3=                   3 4=                  12 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x
```



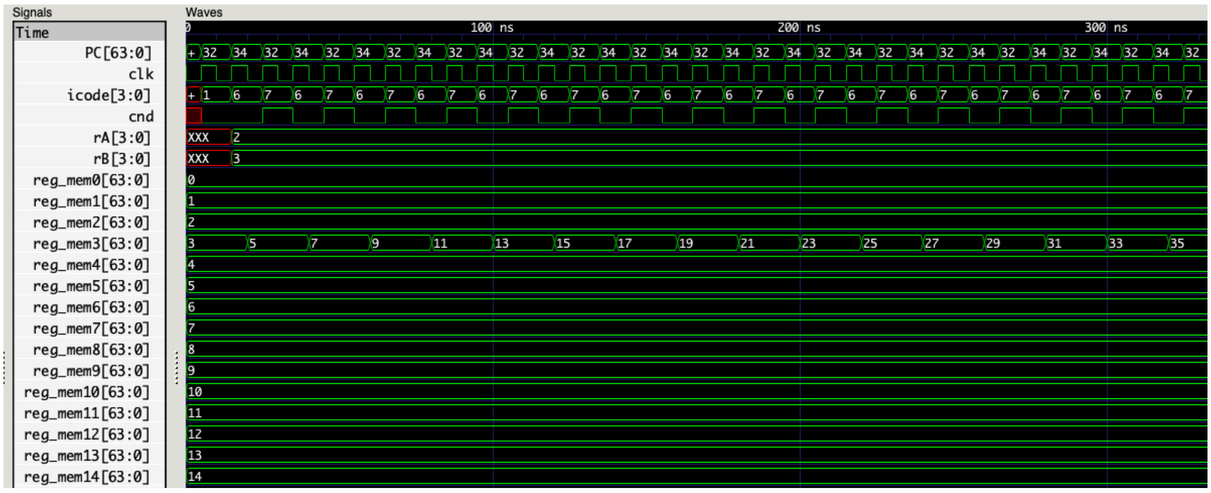
For testing the processors for jump and conditional move statements the following set of instructions is used

```
//OPq
    instr_mem[32]=8'b01100000; //5 fn
    instr_mem[33]=8'b00100011; //rA rB

  //jxx
    instr_mem[34]=8'b01110000; //7 fn
    instr_mem[35]=8'b00000000; //Dest
    instr_mem[36]=8'b00000000; //Dest
    instr_mem[37]=8'b00000000; //Dest
    instr_mem[38]=8'b00000000; //Dest
    instr_mem[39]=8'b00000000; //Dest
    instr_mem[40]=8'b00000000; //Dest
    instr_mem[41]=8'b00000000; //Dest
    instr_mem[42]=8'b00100000; //Dest
```

The outputs for the sequential processor is as follows

```
clk=0 icode=xxxx ifun=xxxx rA=xxxx rB=xxxx valA=                   x valB=
x valC=                   x valE=                   x valM=                   x insval
=x memerr=x cnd=x halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=1 icode=0001 ifun=0000 rA=xxxx rB=xxxx valA=                   x valB=
x valC=                   x valE=                   x valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=0 icode=0001 ifun=0000 rA=xxxx rB=xxxx valA=                   x valB=
x valC=                   x valE=                   x valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=1 icode=0110 ifun=0000 rA=0010 rB=0011 valA=                   2 valB=
3 valC=                   x valE=                   5 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   3 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  x

clk=0 icode=0110 ifun=0000 rA=0010 rB=0011 valA=                  2 valB=
5 valC=                  x valE=                  5 valM=                  x insval
=1 memerr=0 cnd=0 halt=0 0=                  0 1=                  1 2=
2 3=                  5 4=                  4 5=                  5 6=
6 7=                  7 8=                  8 9=                  9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  x

clk=1 icode=0111 ifun=0000 rA=0010 rB=0011 valA=                  2 valB=
5 valC=                  32 valE=                  5 valM=                  x insval
=1 memerr=0 cnd=1 halt=0 0=                  0 1=                  1 2=
2 3=                  5 4=                  4 5=                  5 6=
6 7=                  7 8=                  8 9=                  9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  x

clk=0 icode=0111 ifun=0000 rA=0010 rB=0011 valA=                  2 valB=
5 valC=                  32 valE=                  5 valM=                  x insval
=1 memerr=0 cnd=1 halt=0 0=                  0 1=                  1 2=
2 3=                  5 4=                  4 5=                  5 6=
6 7=                  7 8=                  8 9=                  9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  x

clk=1 icode=0110 ifun=0000 rA=0010 rB=0011 valA=                  2 valB=
5 valC=                  32 valE=                  7 valM=                  x insval
=1 memerr=0 cnd=0 halt=0 0=                  0 1=                  1 2=
2 3=                  5 4=                  4 5=                  5 6=
6 7=                  7 8=                  8 9=                  9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  x

clk=0 icode=0110 ifun=0000 rA=0010 rB=0011 valA=                  2 valB=
7 valC=                  32 valE=                  7 valM=                  x insval
=1 memerr=0 cnd=0 halt=0 0=                  0 1=                  1 2=
2 3=                  7 4=                  4 5=                  5 6=
6 7=                  7 8=                  8 9=                  9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  x

clk=1 icode=0111 ifun=0000 rA=0010 rB=0011 valA=                  2 valB=
7 valC=                  32 valE=                  7 valM=                  x insval
=1 memerr=0 cnd=1 halt=0 0=                  0 1=                  1 2=
2 3=                  7 4=                  4 5=                  5 6=
6 7=                  7 8=                  8 9=                  9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  x

clk=0 icode=0111 ifun=0000 rA=0010 rB=0011 valA=                  2 valB=
7 valC=                  32 valE=                  7 valM=                  x insval
=1 memerr=0 cnd=1 halt=0 0=                  0 1=                  1 2=
2 3=                  7 4=                  4 5=                  5 6=
6 7=                  7 8=                  8 9=                  9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                  x
```

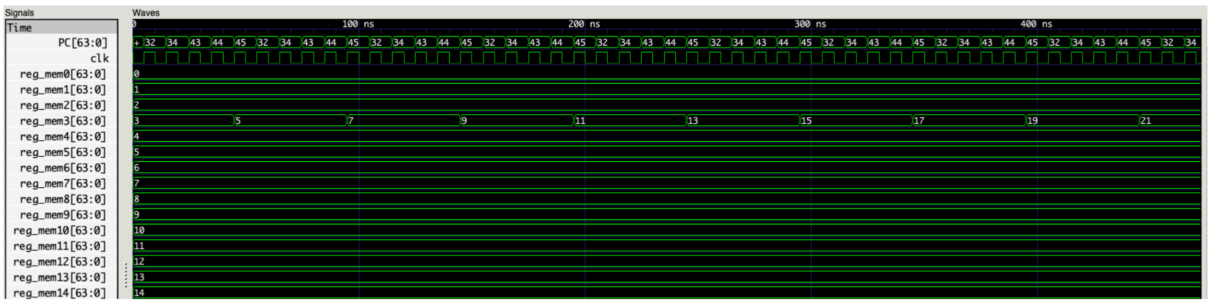
```
clk=1 icode=0110 ifun=0000 rA=0010 rB=0011 valA=                  2 valB=
7 valC=                  32 valE=                   9 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   7 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x

clk=0 icode=0110 ifun=0000 rA=0010 rB=0011 valA=                  2 valB=
9 valC=                  32 valE=                   9 valM=                   x insval
=1 memerr=0 cnd=0 halt=0 0=                   0 1=                   1 2=
2 3=                   9 4=                   4 5=                   5 6=
6 7=                   7 8=                   8 9=                   9 10=
10 11=                  11 12=                  12 13=                  13 14=
14 datamem=                   x
```



The outputs for the pipelined processor is as follows

```
clk=0 f= 1 d= x e= x m= x wb= x
clk=1 f= 6 d= 1 e= x m= x wb= x
clk=0 f= 6 d= 1 e= x m= x wb= x
clk=1 f= 7 d= 6 e= 1 m= x wb= x
clk=0 f= 7 d= 6 e= 1 m= x wb= x
clk=1 f= 1 d= 7 e= 6 m= 1 wb= x
clk=0 f= 1 d= 7 e= 6 m= 1 wb= x
clk=1 f= 1 d= 1 e= 7 m= 6 wb= 1
clk=0 f= 1 d= 1 e= 7 m= 6 wb= 1
clk=1 f= 1 d= 1 e= 1 m= 7 wb= 6
clk=0 f= 1 d= 1 e= 1 m= 7 wb= 6
clk=1 f= 6 d= 1 e= 1 m= 1 wb= 7
clk=0 f= 6 d= 1 e= 1 m= 1 wb= 7
clk=1 f= 7 d= 6 e= 1 m= 1 wb= 1
clk=0 f= 7 d= 6 e= 1 m= 1 wb= 1
clk=1 f= 1 d= 7 e= 6 m= 1 wb= 1
clk=0 f= 1 d= 7 e= 6 m= 1 wb= 1
```

```
clk=1 f= 1 d= 1 e= 7 m= 6 wb= 1
clk=0 f= 1 d= 1 e= 7 m= 6 wb= 1
clk=1 f= 1 d= 1 e= 1 m= 7 wb= 6
clk=0 f= 1 d= 1 e= 1 m= 7 wb= 6
clk=1 f= 6 d= 1 e= 1 m= 1 wb= 7
clk=0 f= 6 d= 1 e= 1 m= 1 wb= 7
clk=1 f= 7 d= 6 e= 1 m= 1 wb= 1
clk=0 f= 7 d= 6 e= 1 m= 1 wb= 1
operation terminated as it will keep looping
```



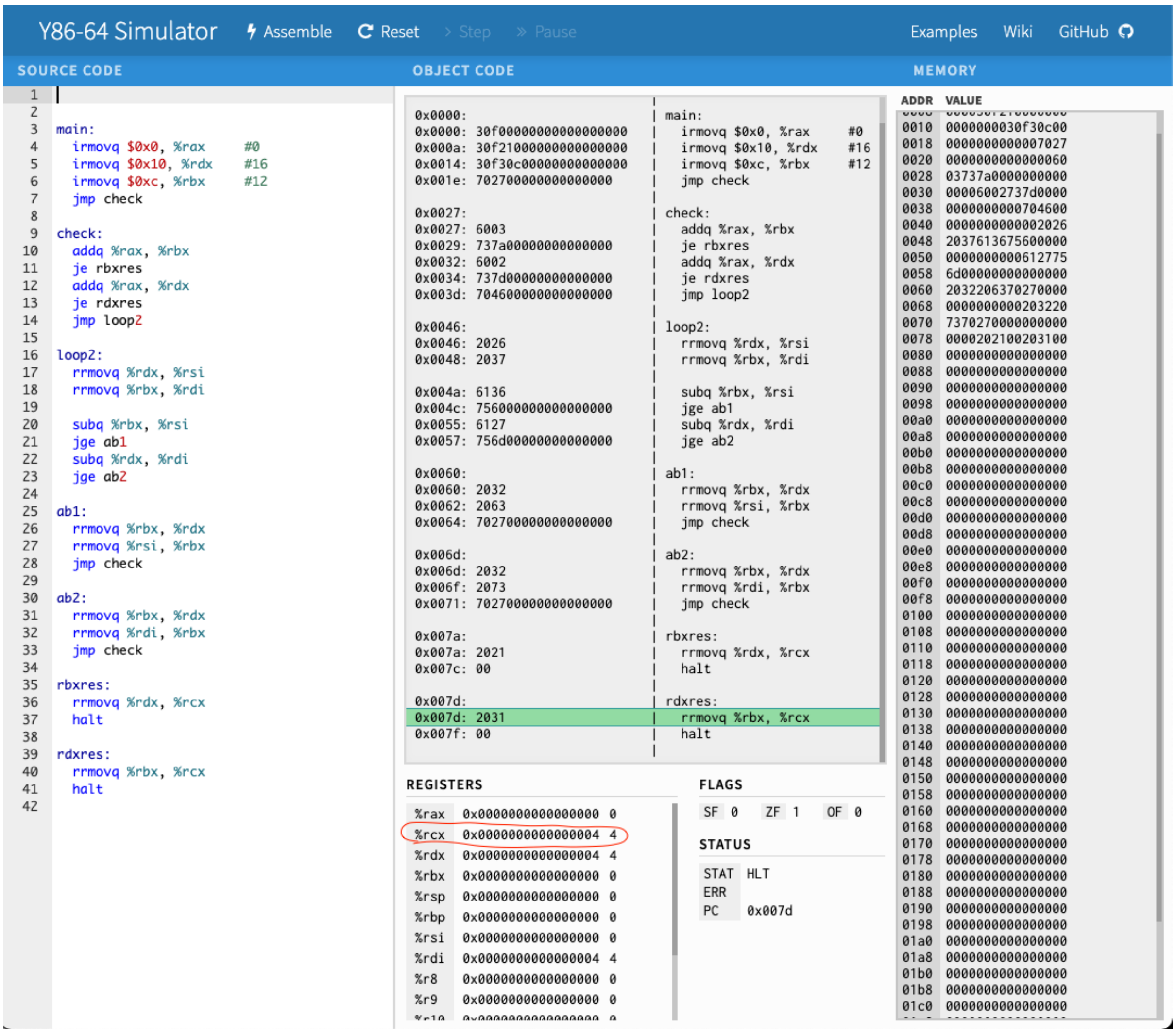
We can clearly see that jump statements are fully functional and looping is possible.

The processor has been tested using an assembly code to find the greatest common divisor or highest common factor of 2 numbers.

The assembly code for this is as follows

```
main:
  irmovq $0x0, %rax
  irmovq $0x12, %rdx
  irmovq $0xc, %rbx
  jmp check

check:
  addq %rax, %rbx
  je rbxres
  addq %rax, %rdx
  je rdxres
  jmp loop2

loop2:
  rrmovq %rdx, %rsi
  rrmovq %rbx, %rdi

  subq %rbx, %rsi
  jge ab1
  subq %rdx, %rdi
  jge ab2

ab1:
  rrmovq %rbx, %rdx
  rrmovq %rsi, %rbx
  jmp check

ab2:
  rrmovq %rbx, %rdx
  rrmovq %rdi, %rbx
  jmp check

rbxres:
  rrmovq %rdx, %rcx
  halt

rdxres:
  rrmovq %rbx, %rcx
  halt
```

This code has been verified on a simulator and the results obtained are as follows



```

main:
  irmovq $0x0, %rax     #0
  irmovq $0x10, %rdx    #16
  irmovq $0xc, %rbx     #12
  jmp check

check:
  addq %rax, %rbx
  je rbxres
  addq %rax, %rdx
  je rdxres
  jmp loop2

loop2:
  rrmovq %rdx, %rsi
  rrmovq %rbx, %rdi

  subq %rbx, %rsi
  jge ab1
  subq %rdx, %rdi
  jge ab2

ab1:
  rrmovq %rbx, %rdx
  rrmovq %rsi, %rbx
  jmp check

ab2:
  rrmovq %rbx, %rdx
  rrmovq %rdi, %rbx
  jmp check

rbxres:
  rrmovq %rdx, %rcx
  halt

rdxres:
  rrmovq %rbx, %rcx
  halt

```

```
0x0000:                        | main:
0x0000: 30f00000000000000000   |   irmovq $0x0, %rax     #0
0x000a: 30f21000000000000000   |   irmovq $0x10, %rdx    #16
0x0014: 30f30c00000000000000   |   irmovq $0xc, %rbx     #12
0x001e: 702700000000000000     |   jmp check
                               |
0x0027:                        | check:
0x0027: 6003                   |   addq %rax, %rbx
0x0029: 737a00000000000000     |   je rbxres
0x0032: 6002                   |   addq %rax, %rdx
0x0034: 737d00000000000000     |   je rdxres
0x003d: 704600000000000000     |   jmp loop2
                               |
0x0046:                        | loop2:
0x0046: 2026                   |   rrmovq %rdx, %rsi
0x0048: 2037                   |   rrmovq %rbx, %rdi
                               |
0x004a: 6136                   |   subq %rbx, %rsi
0x004c: 756000000000000000     |   jge ab1
0x0055: 6127                   |   subq %rdx, %rdi
0x0057: 756d00000000000000     |   jge ab2
                               |
0x0060:                        | ab1:
0x0060: 2032                   |   rrmovq %rbx, %rdx
0x0062: 2063                   |   rrmovq %rsi, %rbx
0x0064: 702700000000000000     |   jmp check
                               |
0x006d:                        | ab2:
0x006d: 2032                   |   rrmovq %rbx, %rdx
0x006f: 2073                   |   rrmovq %rdi, %rbx
0x0071: 702700000000000000     |   jmp check
                               |
0x007a:                        | rbxres:
0x007a: 2021                   |   rrmovq %rdx, %rcx
0x007c: 00                     |   halt
                               |
0x007d:                        | rdxres:
0x007d: 2031                   |   rrmovq %rbx, %rcx
0x007f: 00                     |   halt
                               |
```

REGISTERS

```
%rax  0x0000000000000000  0
%rcx  0x0000000000000004  4
%rdx  0x0000000000000004  4
%rbx  0x0000000000000000  0
%rsp  0x0000000000000000  0
%rbp  0x0000000000000000  0
%rsi  0x0000000000000000  0
%rdi  0x0000000000000004  4
%r8   0x0000000000000000  0
%r9   0x0000000000000000  0
```

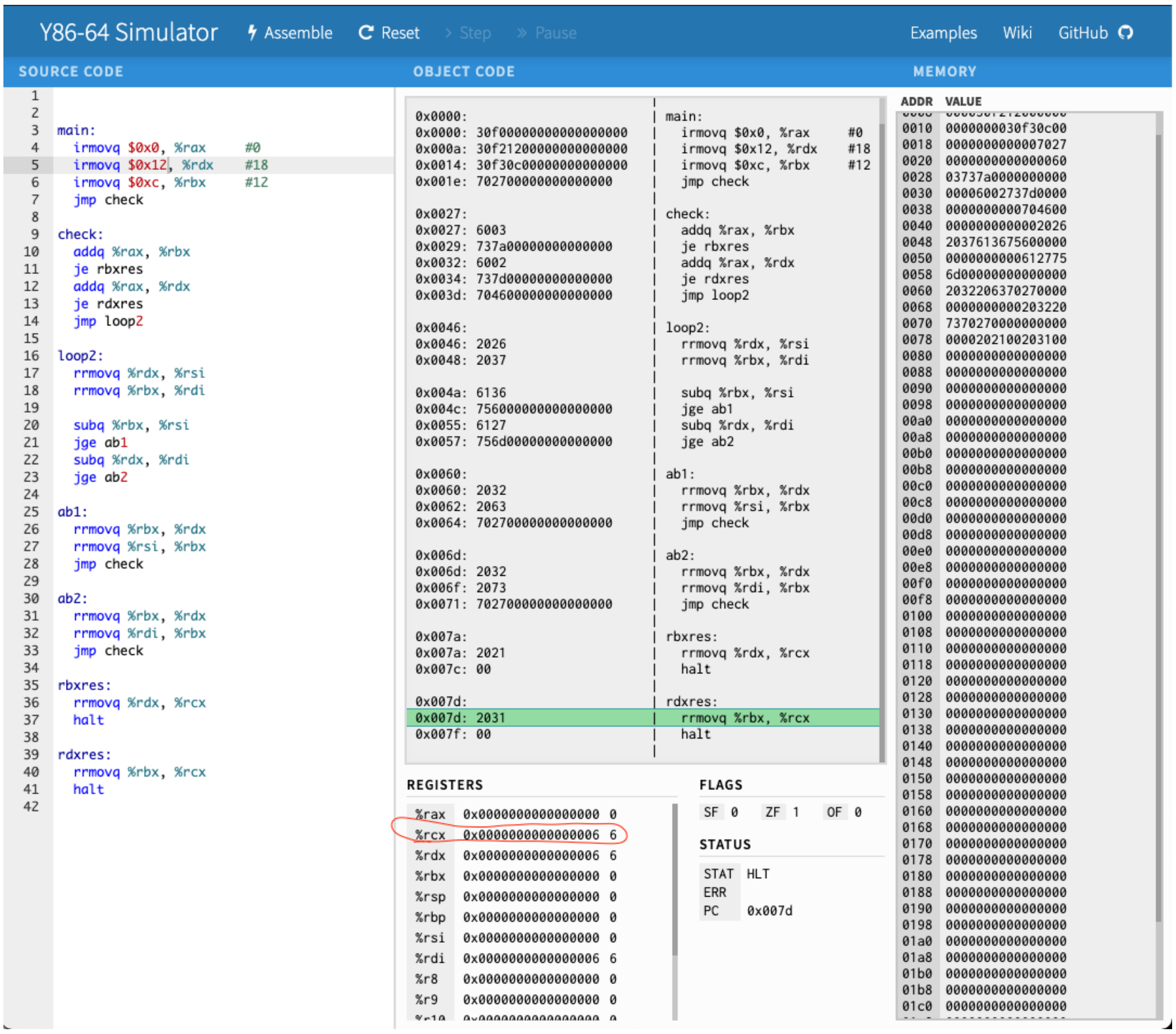
FLAGS

```
SF 0   ZF 1   OF 0
```

STATUS

```
STAT  HLT
ERR
PC    0x007d
```

```
ADDR  VALUE
0010  0000000030f30c00
0018  0000000000007027
0020  0000000000000060
0028  03737a0000000000
0030  00006002737d0000
0038  0000000000704600
0040  0000000000002026
0048  2037613675600000
0050  0000000000612775
0058  6d00000000000000
0060  2032206370270000
0068  0000000000203220
0070  7370270000000000
0078  0000202100203100
0080  0000000000000000
0088  0000000000000000
0090  0000000000000000
0098  0000000000000000
00a0  0000000000000000
00a8  0000000000000000
00b0  0000000000000000
00b8  0000000000000000
00c0  0000000000000000
00c8  0000000000000000
00d0  0000000000000000
00d8  0000000000000000
00e0  0000000000000000
00e8  0000000000000000
00f0  0000000000000000
00f8  0000000000000000
0100  0000000000000000
0108  0000000000000000
0110  0000000000000000
0118  0000000000000000
0120  0000000000000000
0128  0000000000000000
0130  0000000000000000
0138  0000000000000000
0140  0000000000000000
0148  0000000000000000
0150  0000000000000000
0158  0000000000000000
0160  0000000000000000
0168  0000000000000000
0170  0000000000000000
0178  0000000000000000
0180  0000000000000000
0188  0000000000000000
0190  0000000000000000
0198  0000000000000000
01a0  0000000000000000
01a8  0000000000000000
01b0  0000000000000000
01b8  0000000000000000
01c0  0000000000000000
```



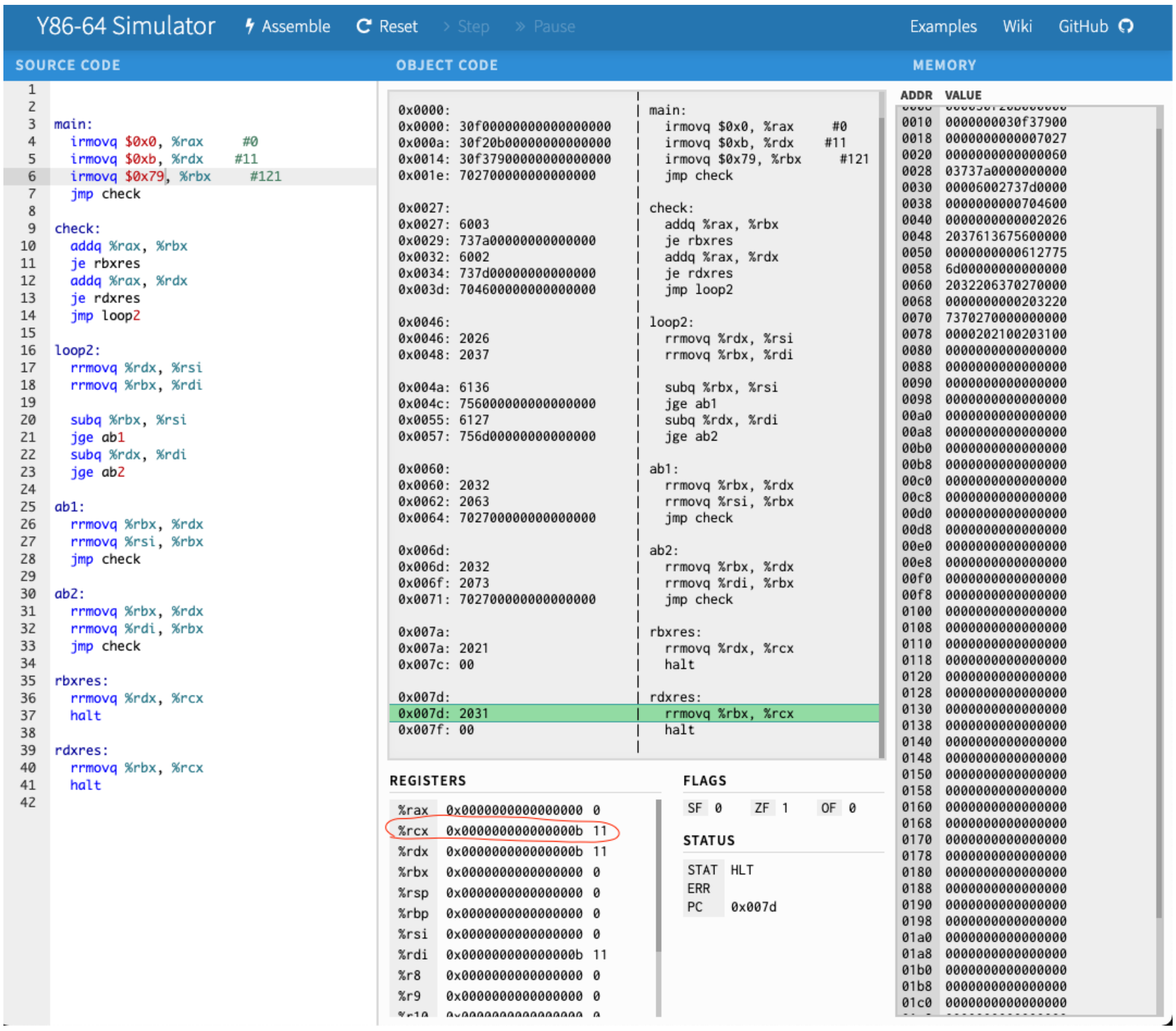
```
main:
  irmovq $0x0, %rax      #0
  irmovq $0x12, %rdx     #18
  irmovq $0xc, %rbx      #12
  jmp check

check:
  addq %rax, %rbx
  je rbxres
  addq %rax, %rdx
  je rdxres
  jmp loop2

loop2:
  rrmovq %rdx, %rsi
  rrmovq %rbx, %rdi

  subq %rbx, %rsi
  jge ab1
  subq %rdx, %rdi
  jge ab2

ab1:
  rrmovq %rbx, %rdx
  rrmovq %rsi, %rbx
  jmp check

ab2:
  rrmovq %rbx, %rdx
  rrmovq %rdi, %rbx
  jmp check

rbxres:
  rrmovq %rdx, %rcx
  halt

rdxres:
  rrmovq %rbx, %rcx
  halt
```

```
                                  |
0x0000:                           | main:
0x0000: 30f00000000000000000      |   irmovq $0x0, %rax      #0
0x000a: 30f21200000000000000      |   irmovq $0x12, %rdx     #18
0x0014: 30f30c00000000000000      |   irmovq $0xc, %rbx      #12
0x001e: 702700000000000000        |   jmp check
                                  |
0x0027:                           | check:
0x0027: 6003                      |   addq %rax, %rbx
0x0029: 737a00000000000000        |   je rbxres
0x0032: 6002                      |   addq %rax, %rdx
0x0034: 737d00000000000000        |   je rdxres
0x003d: 704600000000000000        |   jmp loop2
                                  |
0x0046:                           | loop2:
0x0046: 2026                      |   rrmovq %rdx, %rsi
0x0048: 2037                      |   rrmovq %rbx, %rdi
                                  |
0x004a: 6136                      |   subq %rbx, %rsi
0x004c: 756000000000000000        |   jge ab1
0x0055: 6127                      |   subq %rdx, %rdi
0x0057: 756d00000000000000        |   jge ab2
                                  |
0x0060:                           | ab1:
0x0060: 2032                      |   rrmovq %rbx, %rdx
0x0062: 2063                      |   rrmovq %rsi, %rbx
0x0064: 702700000000000000        |   jmp check
                                  |
0x006d:                           | ab2:
0x006d: 2032                      |   rrmovq %rbx, %rdx
0x006f: 2073                      |   rrmovq %rdi, %rbx
0x0071: 702700000000000000        |   jmp check
                                  |
0x007a:                           | rbxres:
0x007a: 2021                      |   rrmovq %rdx, %rcx
0x007c: 00                        |   halt
                                  |
0x007d:                           | rdxres:
0x007d: 2031                      |   rrmovq %rbx, %rcx
0x007f: 00                        |   halt
                                  |
```

```
REGISTERS
%rax  0x0000000000000000  0
%rcx  0x0000000000000006  6
%rdx  0x0000000000000006  6
%rbx  0x0000000000000000  0
%rsp  0x0000000000000000  0
%rbp  0x0000000000000000  0
%rsi  0x0000000000000000  0
%rdi  0x0000000000000006  6
%r8   0x0000000000000000  0
%r9   0x0000000000000000  0
```

```
FLAGS
SF  0    ZF  1    OF  0

STATUS
STAT  HLT
ERR
PC    0x007d
```

```
ADDR  VALUE
0010  0000000030f30c00
0018  0000000000007027
0020  0000000000000060
0028  03737a0000000000
0030  00006002737d0000
0038  0000000000704600
0040  0000000000002026
0048  2037613675600000
0050  0000000000612775
0058  6d00000000000000
0060  2032206370270000
0068  0000000000203220
0070  7370270000000000
0078  0000202100203100
0080  0000000000000000
0088  0000000000000000
0090  0000000000000000
0098  0000000000000000
00a0  0000000000000000
00a8  0000000000000000
00b0  0000000000000000
00b8  0000000000000000
00c0  0000000000000000
00c8  0000000000000000
00d0  0000000000000000
00d8  0000000000000000
00e0  0000000000000000
00e8  0000000000000000
00f0  0000000000000000
00f8  0000000000000000
0100  0000000000000000
0108  0000000000000000
0110  0000000000000000
0118  0000000000000000
0120  0000000000000000
0128  0000000000000000
0130  0000000000000000
0138  0000000000000000
0140  0000000000000000
0148  0000000000000000
0150  0000000000000000
0158  0000000000000000
0160  0000000000000000
0168  0000000000000000
0170  0000000000000000
0178  0000000000000000
0180  0000000000000000
0188  0000000000000000
0190  0000000000000000
0198  0000000000000000
01a0  0000000000000000
01a8  0000000000000000
01b0  0000000000000000
01b8  0000000000000000
01c0  0000000000000000
```



```
main:
  irmovq $0x0, %rax     #0
  irmovq $0xb, %rdx    #11
  irmovq $0x79, %rbx     #121
  jmp check

check:
  addq %rax, %rbx
  je rbxres
  addq %rax, %rdx
  je rdxres
  jmp loop2

loop2:
  rrmovq %rdx, %rsi
  rrmovq %rbx, %rdi

  subq %rbx, %rsi
  jge ab1
  subq %rdx, %rdi
  jge ab2

ab1:
  rrmovq %rbx, %rdx
  rrmovq %rsi, %rbx
  jmp check

ab2:
  rrmovq %rbx, %rdx
  rrmovq %rdi, %rbx
  jmp check

rbxres:
  rrmovq %rdx, %rcx
  halt

rdxres:
  rrmovq %rbx, %rcx
  halt
```

```
0x0000:                      | main:
0x0000: 30f00000000000000000 |   irmovq $0x0, %rax     #0
0x000a: 30f20b00000000000000 |   irmovq $0xb, %rdx    #11
0x0014: 30f37900000000000000 |   irmovq $0x79, %rbx     #121
0x001e: 702700000000000000   |   jmp check
                             |
0x0027:                      | check:
0x0027: 6003                 |   addq %rax, %rbx
0x0029: 737a00000000000000   |   je rbxres
0x0032: 6002                 |   addq %rax, %rdx
0x0034: 737d00000000000000   |   je rdxres
0x003d: 704600000000000000   |   jmp loop2
                             |
0x0046:                      | loop2:
0x0046: 2026                 |   rrmovq %rdx, %rsi
0x0048: 2037                 |   rrmovq %rbx, %rdi
                             |
0x004a: 6136                 |   subq %rbx, %rsi
0x004c: 756000000000000000   |   jge ab1
0x0055: 6127                 |   subq %rdx, %rdi
0x0057: 756d00000000000000   |   jge ab2
                             |
0x0060:                      | ab1:
0x0060: 2032                 |   rrmovq %rbx, %rdx
0x0062: 2063                 |   rrmovq %rsi, %rbx
0x0064: 702700000000000000   |   jmp check
                             |
0x006d:                      | ab2:
0x006d: 2032                 |   rrmovq %rbx, %rdx
0x006f: 2073                 |   rrmovq %rdi, %rbx
0x0071: 702700000000000000   |   jmp check
                             |
0x007a:                      | rbxres:
0x007a: 2021                 |   rrmovq %rdx, %rcx
0x007c: 00                   |   halt
                             |
0x007d:                      | rdxres:
0x007d: 2031                 |   rrmovq %rbx, %rcx
0x007f: 00                   |   halt
                             |
```

REGISTERS

```
%rax  0x0000000000000000  0
%rcx  0x000000000000000b  11
%rdx  0x000000000000000b  11
%rbx  0x0000000000000000  0
%rsp  0x0000000000000000  0
%rbp  0x0000000000000000  0
%rsi  0x0000000000000000  0
%rdi  0x000000000000000b  11
%r8   0x0000000000000000  0
%r9   0x0000000000000000  0
```

FLAGS

```
SF 0    ZF 1    OF 0
```

STATUS

```
STAT  HLT
ERR
PC    0x007d
```

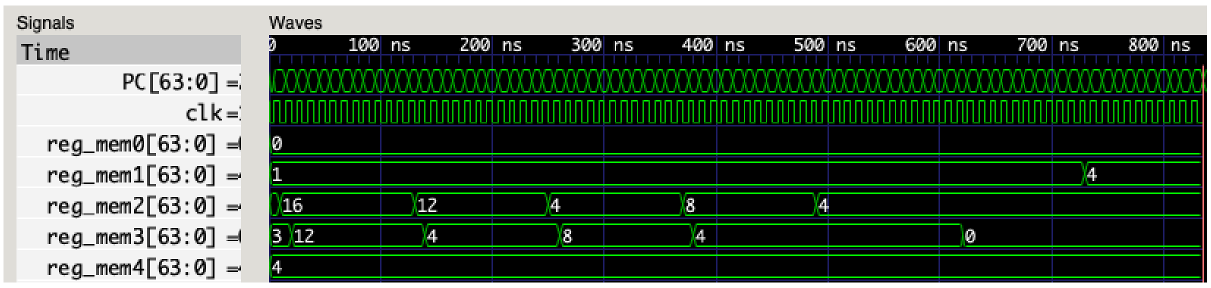
```
ADDR  VALUE
0010  0000000030f37900
0018  0000000000007027
0020  0000000000000060
0028  03737a0000000000
0030  00006002737d0000
0038  0000000000704600
0040  0000000000002026
0048  2037613675600000
0050  0000000000612775
0058  6d00000000000000
0060  2032206370270000
0068  0000000000203220
0070  7370270000000000
0078  0000202100203100
0080  0000000000000000
0088  0000000000000000
0090  0000000000000000
0098  0000000000000000
00a0  0000000000000000
00a8  0000000000000000
00b0  0000000000000000
00b8  0000000000000000
00c0  0000000000000000
00c8  0000000000000000
00d0  0000000000000000
00d8  0000000000000000
00e0  0000000000000000
00e8  0000000000000000
00f0  0000000000000000
00f8  0000000000000000
0100  0000000000000000
0108  0000000000000000
0110  0000000000000000
0118  0000000000000000
0120  0000000000000000
0128  0000000000000000
0130  0000000000000000
0138  0000000000000000
0140  0000000000000000
0148  0000000000000000
0150  0000000000000000
0158  0000000000000000
0160  0000000000000000
0168  0000000000000000
0170  0000000000000000
0178  0000000000000000
0180  0000000000000000
0188  0000000000000000
0190  0000000000000000
0198  0000000000000000
01a0  0000000000000000
01a8  0000000000000000
01b0  0000000000000000
01b8  0000000000000000
01c0  0000000000000000
```

The corresponding binary encoded instructions as inputted to processor in verilog is

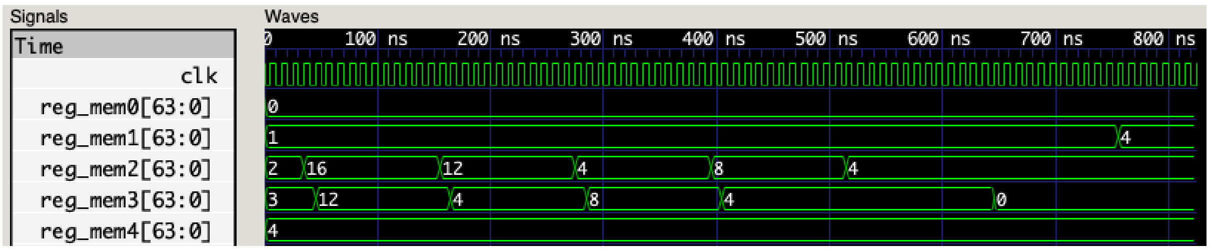
```
//main:
  //irmovq $0x0, %rax
  instr_mem[0]=8'b00110000; //3 0
  instr_mem[1]=8'b00000000; //F rB=0
  instr_mem[2]=8'b00000000;
  instr_mem[3]=8'b00000000;
  instr_mem[4]=8'b00000000;
  instr_mem[5]=8'b00000000;
  instr_mem[6]=8'b00000000;
  instr_mem[7]=8'b00000000;
  instr_mem[8]=8'b00000000;
  instr_mem[9]=8'b00000000; //V=0
  //irmovq $0x10, %rdx
  instr_mem[10]=8'b00110000; //3 0
  instr_mem[11]=8'b00000010; //F rB=2
  instr_mem[12]=8'b00000000;
```

```
    instr_mem[13]=8'b00000000;
    instr_mem[14]=8'b00000000;
    instr_mem[15]=8'b00000000;
    instr_mem[16]=8'b00000000;
    instr_mem[17]=8'b00000000;
    instr_mem[18]=8'b00000000;
    instr_mem[19]=8'b00010000; //V=16
    //irmovq $0xc, %rbx
    instr_mem[20]=8'b00110000; //3 0
    instr_mem[21]=8'b00000011; //F rB=3
    instr_mem[22]=8'b00000000;
    instr_mem[23]=8'b00000000;
    instr_mem[24]=8'b00000000;
    instr_mem[25]=8'b00000000;
    instr_mem[26]=8'b00000000;
    instr_mem[27]=8'b00000000;
    instr_mem[28]=8'b00000000;
    instr_mem[29]=8'b00001100; //V=12
    //jmp check
    instr_mem[30]=8'b01110000; //7 fn
    instr_mem[31]=8'b00000000; //Dest
    instr_mem[32]=8'b00000000; //Dest
    instr_mem[33]=8'b00000000; //Dest
    instr_mem[34]=8'b00000000; //Dest
    instr_mem[35]=8'b00000000; //Dest
    instr_mem[36]=8'b00000000; //Dest
    instr_mem[37]=8'b00000000; //Dest
    instr_mem[38]=8'b00100111; //Dest=39

  // check:
    // addq %rax, %rbx
    instr_mem[39]=8'b01100000; //5 fn
    instr_mem[40]=8'b00000011; //rA=0 rB=3
    // je rbxres
    instr_mem[41]=8'b01110011; //7 fn=3
    instr_mem[42]=8'b00000000; //Dest
    instr_mem[43]=8'b00000000; //Dest
    instr_mem[44]=8'b00000000; //Dest
    instr_mem[45]=8'b00000000; //Dest
    instr_mem[46]=8'b00000000; //Dest
    instr_mem[47]=8'b00000000; //Dest
    instr_mem[48]=8'b00000000; //Dest
    instr_mem[49]=8'b01111010; //Dest=122
    // addq %rax, %rdx
    instr_mem[50]=8'b01100000; //5 fn
    instr_mem[51]=8'b00000010; //rA=0 rB=2
    // je rdxres
    instr_mem[52]=8'b01110011; //7 fn=3
    instr_mem[53]=8'b00000000; //Dest
    instr_mem[54]=8'b00000000; //Dest
    instr_mem[55]=8'b00000000; //Dest
    instr_mem[56]=8'b00000000; //Dest
    instr_mem[57]=8'b00000000; //Dest
    instr_mem[58]=8'b00000000; //Dest
    instr_mem[59]=8'b00000000; //Dest
    instr_mem[60]=8'b01111101; //Dest=125
    // jmp loop2
    instr_mem[61]=8'b01110000; //7 fn=0
    instr_mem[62]=8'b00000000; //Dest
    instr_mem[63]=8'b00000000; //Dest
    instr_mem[64]=8'b00000000; //Dest
    instr_mem[65]=8'b00000000; //Dest
    instr_mem[66]=8'b00000000; //Dest
    instr_mem[67]=8'b00000000; //Dest
    instr_mem[68]=8'b00000000; //Dest
    instr_mem[69]=8'b01000110; //Dest

  // loop2:
    // rrmovq %rdx, %rsi
    instr_mem[70]=8'b00100000; //2 fn=0
    instr_mem[71]=8'b00100110; //rA=2 rB=6
    // rrmovq %rbx, %rdi
    instr_mem[72]=8'b00100000; //2 fn=0
    instr_mem[73]=8'b00110111; //rA=3 rB=7
    // subq %rbx, %rsi
    instr_mem[74]=8'b01100001; //5 fn=1
    instr_mem[75]=8'b00110110; //rA=3 rB=6
    // jge ab1
    instr_mem[76]=8'b01110001; //7 fn=5
    instr_mem[77]=8'b00000000; //Dest
    instr_mem[78]=8'b00000000; //Dest
    instr_mem[79]=8'b00000000; //Dest
    instr_mem[80]=8'b00000000; //Dest
    instr_mem[81]=8'b00000000; //Dest
    instr_mem[82]=8'b00000000; //Dest
    instr_mem[83]=8'b00000000; //Dest
    instr_mem[84]=8'b01100000; //Dest=96
    // subq %rdx, %rdi
    instr_mem[85]=8'b01100001; //5 fn
    instr_mem[86]=8'b00100111; //rA=2 rB=7
    // jge ab2
    instr_mem[87]=8'b01110001; //7 fn=5
    instr_mem[88]=8'b00000000; //Dest
    instr_mem[89]=8'b00000000; //Dest
    instr_mem[90]=8'b00000000; //Dest
    instr_mem[91]=8'b00000000; //Dest
    instr_mem[92]=8'b00000000; //Dest
    instr_mem[93]=8'b00000000; //Dest
    instr_mem[94]=8'b00000000; //Dest
    instr_mem[95]=8'b01101101; //Dest=109

  // ab1:
    // rrmovq %rbx, %rdx
    instr_mem[96]=8'b00100000; //2 fn=0
    instr_mem[97]=8'b00110010; //rA=3 rB=2
    // rrmovq %rsi, %rbx
    instr_mem[98]=8'b00100000; //2 fn=0
    instr_mem[99]=8'b01100011; //rA=6 rB=3
    // jmp check
    instr_mem[100]=8'b01110000; //7 fn=0
    instr_mem[101]=8'b00000000; //Dest
    instr_mem[102]=8'b00000000; //Dest
    instr_mem[103]=8'b00000000; //Dest
    instr_mem[104]=8'b00000000; //Dest
    instr_mem[105]=8'b00000000; //Dest
    instr_mem[106]=8'b00000000; //Dest
    instr_mem[107]=8'b00000000; //Dest
    instr_mem[108]=8'b00100111; //Dest=39

  // ab2:
    // rrmovq %rbx, %rdx
    instr_mem[109]=8'b00100000; //2 fn=0
    instr_mem[110]=8'b00110010; //rA=3 rB=2
    // rrmovq %rdi, %rbx
    instr_mem[111]=8'b00100000; //2 fn=0
    instr_mem[112]=8'b01110011; //rA=7 rB=3
    // jmp check
    instr_mem[113]=8'b01110000; //7 fn=0
    instr_mem[114]=8'b00000000; //Dest
    instr_mem[115]=8'b00000000; //Dest
    instr_mem[116]=8'b00000000; //Dest
    instr_mem[117]=8'b00000000; //Dest
    instr_mem[118]=8'b00000000; //Dest
    instr_mem[119]=8'b00000000; //Dest
    instr_mem[120]=8'b00000000; //Dest
    instr_mem[121]=8'b00100111; //Dest=39

  // rbxres:
    // rrmovq %rdx, %rcx
    instr_mem[122]=8'b00100000; //2 fn=0
    instr_mem[123]=8'b00100001; //rA=2 rB=1
    // halt
    instr_mem[124]=8'b00000000;

  // rdxres:
    // rrmovq %rbx, %rcx
    instr_mem[125]=8'b00100000; //2 fn=0
    instr_mem[126]=8'b00110001; //rA=3 rB=1
    // halt
    instr_mem[127]=8'b00000000;
```

The outputs of the sequential processor is as follows



The outputs of the pipelined processor is as follows



We can see that the register values match the for both cases and final result for the gcd is also right. The changes in the values of the registers can also be verified by doing a manual dry run through the assembly code.

Processor features

Processor frequency

Processor frequency, $f = \frac{1}{T_{clk}}$

where T_{clk} is time period of clock signal which is taken to be 5ns

$f = \frac{1}{1 \times 10^{-9}} = 10^9$ Hz = 1 GHz

Memory

- Harvard style memory design with separate data and instruction memory.
- Data memory= 1024 bytes = 1kB
- Instruction memory= 1024 bytes = 1kB

Limitations

-

Instructions to run

1. Navigate to the folder for seq or pipe respectively

2. Add the instructions to be executes in `fetch.v` under inital values of `instr_mem`.
3. Execute the following commands in the respective directory

```
iverilog -o proc proc.v
vvp proc
```

Detailed descriptions and testing

Fetch

The testbench used for testing the fetch stage is as follows

```
`timescale 1ns / 1ps
module fetchtb;
  reg clk;
  reg [63:0] PC;

  wire [3:0] icode;
  wire [3:0] ifun;
  wire [3:0] rA;
  wire [3:0] rB;
  wire [63:0] valC;
  wire [63:0] valP;

  fetch fetch(
    .clk(clk),
    .PC(PC),
    .icode(icode),
    .ifun(ifun),
    .rA(rA),
    .rB(rB),
    .valC(valC),
    .valP(valP)
  );

  initial begin
    clk=0;
    PC=64'd0;

    #10 clk=~clk;PC=64'd32;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
    #10 clk=~clk;PC=valP;
    #10 clk=~clk;
  end

  initial
    $monitor("clk=%d PC=%d icode=%b ifun=%b rA=%b rB=%b,valC=%d,valP=%d\n",clk,PC,icod
e,ifun,rA,rB,valC,valP);
endmodule
```

The outputs for the testbench are as follows

```
clk=0 PC=                    0 icode=xxxx ifun=xxxx rA=xxxx rB=xxxx,valC=
x,valP=                    x

clk=1 PC=                   32 icode=0110 ifun=0000 rA=0010 rB=0011,valC=
x,valP=                   34

clk=0 PC=                   32 icode=0110 ifun=0000 rA=0010 rB=0011,valC=
x,valP=                   34

clk=1 PC=                   34 icode=0001 ifun=0000 rA=0010 rB=0011,valC=
x,valP=                   35

clk=0 PC=                   34 icode=0001 ifun=0000 rA=0010 rB=0011,valC=
x,valP=                   35

clk=1 PC=                   35 icode=0001 ifun=0000 rA=0010 rB=0011,valC=
x,valP=                   36

clk=0 PC=                   35 icode=0001 ifun=0000 rA=0010 rB=0011,valC=
x,valP=                   36

clk=1 PC=                   36 icode=0001 ifun=0000 rA=0010 rB=0011,valC=
x,valP=                   37

clk=0 PC=                   36 icode=0001 ifun=0000 rA=0010 rB=0011,valC=
x,valP=                   37
```

```
clk=1 PC=                  37 icode=0010 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  39

clk=0 PC=                  37 icode=0010 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  39

clk=1 PC=                  39 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  40

clk=0 PC=                  39 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  40

clk=1 PC=                  40 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  41

clk=0 PC=                  40 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  41

clk=1 PC=                  41 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  42

clk=0 PC=                  41 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  42

clk=1 PC=                  42 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  43

clk=0 PC=                  42 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  43

clk=1 PC=                  43 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  44

clk=0 PC=                  43 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  44

clk=1 PC=                  44 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  45

clk=0 PC=                  44 icode=0001 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  45

clk=1 PC=                  45 icode=0000 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  46

clk=0 PC=                  45 icode=0000 ifun=0000 rA=0000 rB=0100,valC=
x,valP=                  46
```

On manually verifying the values for each clock cycle it is clearly visible that the fetch stage is functioning as expected and the fetch instructions and their respective predicted PC increments are correct.

Decode

The testbench used for testing the decode stage is as follows

```
`timescale 1ns / 1ps

module fetchdecodetb;
  reg clk;
  reg [63:0] PC;
  reg [63:0] reg_mem[0:14];

  wire [3:0] icode;
  wire [3:0] ifun;
  wire [3:0] rA;
  wire [3:0] rB;
  wire [63:0] valC;
  wire [63:0] valP;
  wire [63:0] valA;
  wire [63:0] valB;

  fetch fetch(
    .clk(clk),
    .PC(PC),
    .icode(icode),
    .ifun(ifun),
    .rA(rA),
    .rB(rB),
    .valC(valC),
    .valP(valP)
  );

  decode decode(
    .clk(clk),
    .icode(icode),
    .rA(rA),
    .rB(rB),
    .reg_memrA(reg_mem[rA]),
    .reg_memrB(reg_mem[rB]),
    .valA(valA),
    .valB(valB)
  );

  initial begin
    reg_mem[0]=64'd0;
    reg_mem[1]=64'd1;
    reg_mem[2]=64'd2;
    reg_mem[3]=64'd3;
    reg_mem[4]=64'd4;
    reg_mem[5]=64'd5;
    reg_mem[6]=64'd6;
    reg_mem[7]=64'd7;
    reg_mem[8]=64'd8;
    reg_mem[9]=64'd9;
    reg_mem[10]=64'd10;
    reg_mem[11]=64'd11;
    reg_mem[12]=64'd12;
```

```
      reg_mem[13]=64'd13;
      reg_mem[14]=64'd14;

      clk=0;
      PC=64'd0;

      #10 clk=~clk;PC=64'd0;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;
    end

    initial
      $monitor("clk=%d icode=%b ifun=%b rA=%b rB=%b valA=%d valB=%d\n",clk,icode,ifun,r
  A,rB,valA,valB);
  endmodule
```

Execute

ALU

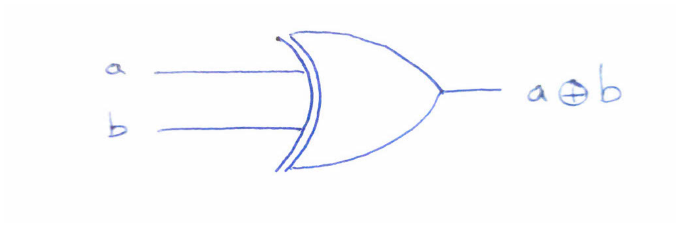
XOR

Logic, Circuit and Module design

The truth table for a XOR logic gate is as follows

a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

The circuit for a XOR logic gate is as follows



Module design

Verilog has built-in gate gate primitives for all the basic gates including XOR, AND, OR, NOT, NAND, NOR and XNOR.

We are required to design a 32 bit XOR module. We have 2 possible approaches to do this. The first would be directly making a module that has 32 XOR gate primitives to operate on each bit. This module is not very modular. The second approach would be making a module that performs XOR on single bit inputs and using the single bit module repeatedly in another module for each of the 32 bits. This is much more modular as we can adjust the number of bits to our convenience without having to modify the initial module that actually performs the operation. We simply have to add or remove instatntiations of the single bit module.

I have chosen the modular approach as it makes it easier to reuse the code and hence is certainly better in the long run. In this approach there is still a choice between instatiating the single bit module 32 times one after the other or using a generate block to multiply the instance. I have made use of the generate block as it serves the purpose well while making the code shorter and cleaner.

Short note on generate block:

The generate block in verilog allows us to multiply module instances or perform conditional instatiation. It is a very convenient way of repeating the same operation or module instatiation multiple times.

Modules

The XOR gate primitive in verilog follows the format

```
xor g1(ans,a,b);
```

where g1 is the name of the gate primitive instance, ans is the output and a and b are the inputs.

Note: More inputs can be added to the xor gate by simple adding more input variables.

The module I have designed for the single bit xor operation is as follows

```
`timescale 1ns / 1ps

module xor1x1(
  input a,
  input b,
  output ans
  );

  xor g1(ans,a,b);

endmodule
```

The module I have designed for the 32 bit operation that uses the single bit operation is as follows

```
`timescale 1ns / 1ps

module xor32x1(
  input signed [31:0]a,
  input signed [31:0]b,
  output signed [31:0]ans
  );

  genvar i;
  generate for(i=0; i<32; i=i+1)
  begin
    xor1x1 g1(a[i],b[i],ans[i]);
  end
  endgenerate
endmodule
```

Testbench

To test the implemented module I have used the following test bench

```
`timescale 1ns / 1ps

module Xor_test;
  reg signed [31:0]a;
  reg signed [31:0]b;

  wire signed [31:0]ans;

  xor32x1 uut(
    .a(a),
    .b(b),
    .ans(ans)
  );

  initial begin
    $dumpfile("Xor_test.vcd");
    $dumpvars(0,Xor_test);
    a = 32'b0;
    b = 32'b0;

    #100;

    #20 a=32'b1011;b=32'b0100;
    #20 a=32'b1011;b=32'b1100;
    #20 a=-32'b1011;b=32'b1100;
    #20 a=32'b1001;b=32'b1001;
    #20 a=-32'd2;b=32'd13;
    #20 a=-32'd2;b=-32'd13;
    #20 a=32'b1001;b=32'b1001;
  end

  initial
    $monitor("a=%b b=%b ans=%b\n",a,b,ans);
endmodule
```

Results

On executing

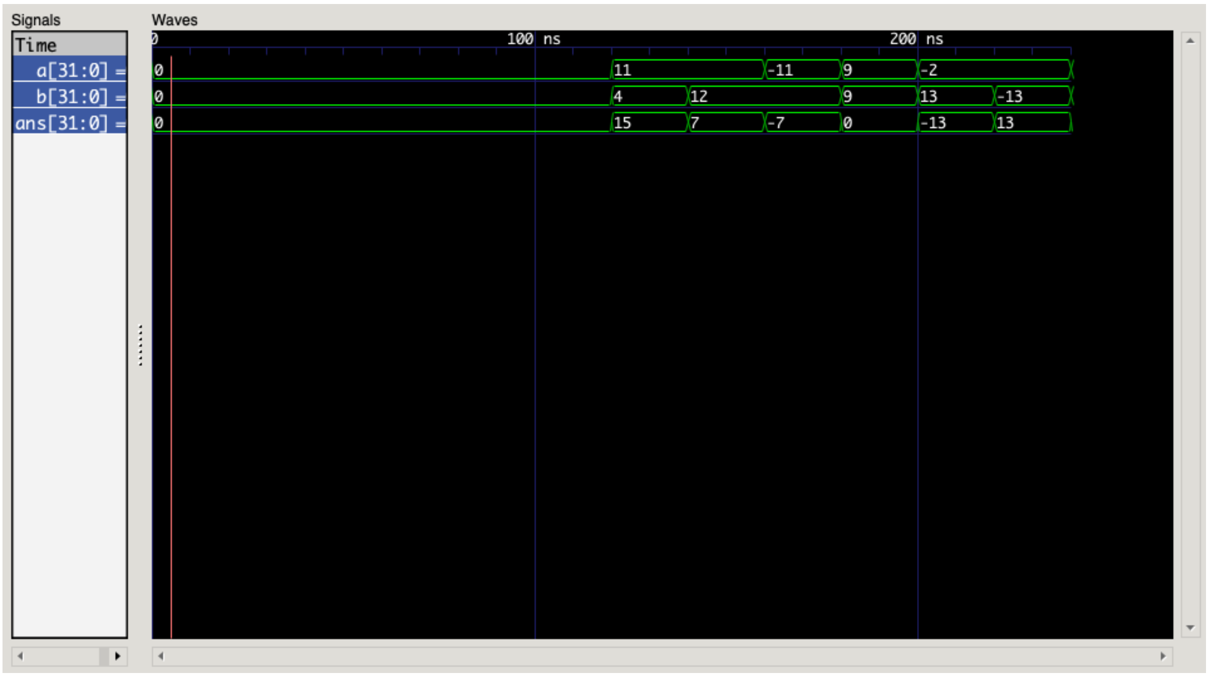
```
iverilog -o xor xor_test.v xor32x1.v xor1x1.v
vvp xor
```

we get the following terminal output

```
a=00000000000000000000000000000000 b=00000000000000000000000000000000 ans=000000000000
00000000000000000000

a=00000000000000000000000000001011 b=00000000000000000000000000000100 ans=000000000000
00000000000000001111

a=00000000000000000000000000001011 b=00000000000000000000000000001100 ans=000000000000
00000000000000000111

a=11111111111111111111111111110101 b=00000000000000000000000000001100 ans=111111111111
11111111111111111001

a=00000000000000000000000000001001 b=00000000000000000000000000001001 ans=000000000000
00000000000000000000

a=11111111111111111111111111111110 b=00000000000000000000000000001101 ans=111111111111
11111111111111110011

a=11111111111111111111111111111110 b=11111111111111111111111111110011 ans=000000000000
00000000000000001101

a=00000000000000000000000000001001 b=00000000000000000000000000001001 ans=000000000000
00000000000000000000
```

We can see that all results are correct for their respective input values of a and b.

The gtkwave output waveform is as follows



The test inputs and their respective outputs can be summarized as follows (in decimal)

a	b	$a \oplus b$
0	0	0
11	4	15
11	12	7
-11	12	-7
9	9	0
-2	13	-13
-2	-13	13

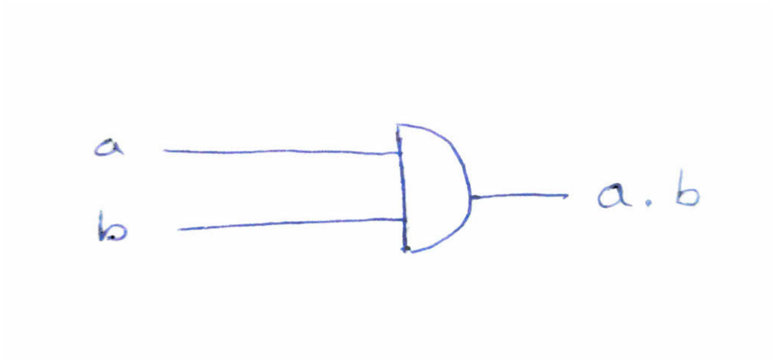
AND

Logic, Circuit and Module design

The truth table for an AND logic gate is as follows

a	b	$a.b$
0	0	0
0	1	0
1	0	0
1	1	1

The circuit for a AND logic gate is as follows



Module design

The design procedure is very similar to the above mentioned 32 bit XOR module design procedure. We make use of the AND gate primitive in verilog to perform the single bit operation. I have made use of the same modular approach where I created a module for the single bit operation and then used a generate block in the 32 bit module to instatiate the single bit module 32 times for the 32 bits of the input.

Modules

The AND gate primitive in verilog follows the format

```
and g1(ans,a,b);
```

where g1 is the name of the gate primitive instance, ans is the output and a and b are the inputs.

Note: More inputs can be added to the and gate by simple adding more input variables.

The module I have designed for the single bit and operation is as follows

```
`timescale 1ns / 1ps

module and1x1(
  input a,
  input b,
  output ans
  );

  and g1(ans,a,b);

endmodule
```

The module I have designed for the 32 bit operation that uses the single bit operation is as follows

```
`timescale 1ns / 1ps

module and32x1(
  input signed [31:0]a,
  input signed [31:0]b,
  output signed [31:0]ans
  );

  genvar i;

  generate for(i=0; i<32; i=i+1)
  begin
    and1x1 g1(a[i],b[i],ans[i]);
  end
  endgenerate
endmodule
```

Testbench

To test the implemented modules I have used the following test bench

```
`timescale 1ns / 1ps

module And_test;
  reg signed [31:0]a;
  reg signed [31:0]b;

  wire signed [31:0]ans;
```

```
    and32x1 uut(
      .a(a),
      .b(b),
      .ans(ans)
    );

    initial begin
      $dumpfile("And_test.vcd");
      $dumpvars(0,And_test);
      a = 32'b0;
      b = 32'b0;

      #100;

      #20 a=32'b1011;b=32'b0100;
      #20 a=32'b1011;b=32'b1100;
      #20 a=-32'b1011;b=32'b1100;
      #20 a=32'b1001;b=32'b1001;
      #20 a=-32'd2;b=32'd13;
      #20 a=-32'd2;b=-32'd13;
      #20 a=32'b1001;b=32'b1001;
    end

    initial
      $monitor("a=%b b=%b ans=%b\n",a,b,ans);
  endmodule
```

Results

On executing

```
iverilog -o and and_test.v and32x1.v and1x1.v
vvp and
```

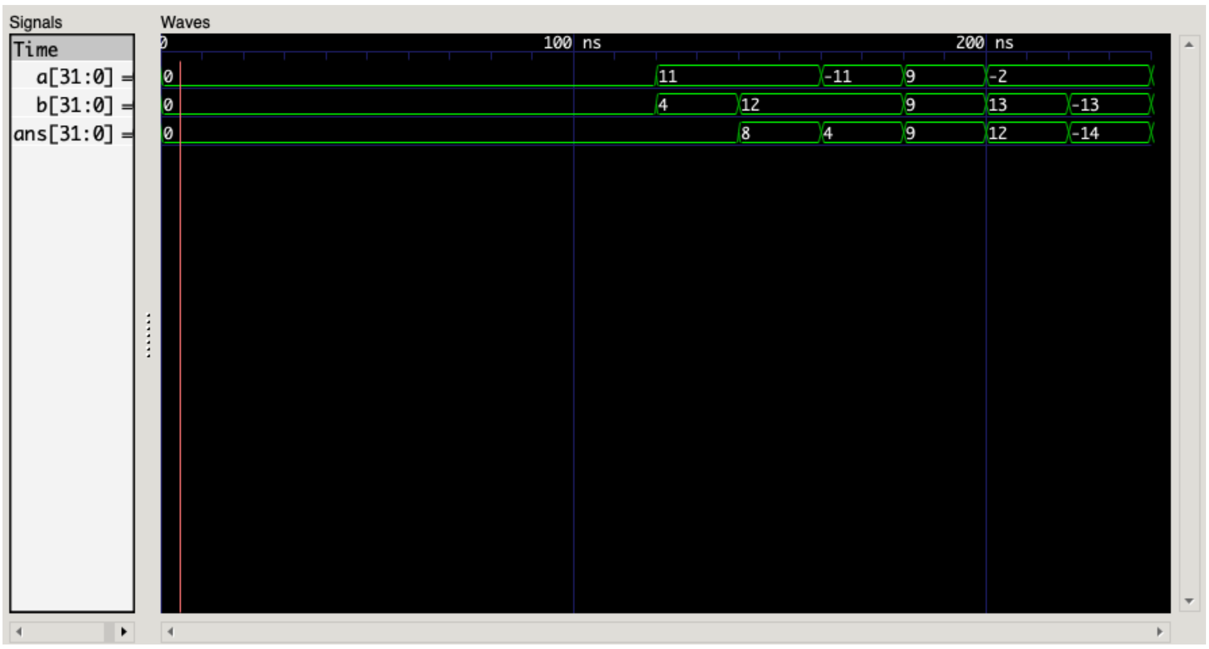
we get the following terminal output

```
a=00000000000000000000000000000000 b=00000000000000000000000000000000 ans=000000000000
00000000000000000000

a=00000000000000000000000000001011 b=00000000000000000000000000000100 ans=000000000000
00000000000000000000

a=00000000000000000000000000001011 b=00000000000000000000000000001100 ans=000000000000
00000000000000001000

a=11111111111111111111111111110101 b=00000000000000000000000000001100 ans=000000000000
00000000000000000100

a=00000000000000000000000000001001 b=00000000000000000000000000001001 ans=000000000000
00000000000000001001
```

```
a=11111111111111111111111111111110 b=00000000000000000000000000001101 ans=000000000000
00000000000000001100

a=11111111111111111111111111111110 b=11111111111111111111111111110011 ans=111111111111
11111111111111110010

a=00000000000000000000000000001001 b=00000000000000000000000000001001 ans=000000000000
00000000000000001001
```

We can see that all results are correct for their respective input values of a and b.

The gtkwave output waveform is as follows



The test inputs and their respective outputs can be summarized as follows (in decimal)

a	b	$a \cdot b$
0	0	0
11	4	0
11	12	8
-11	12	4
9	9	9
-2	13	12
-2	-13	-14

ADD

Logic, Circuit and Module design

We know that the truth table for an ADD function on a single bit should be as follows

a	b	sum	$carry$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

From this we can conclude that we need to make full adders in order to perform the required operation.

From previous knowledge we know that the truth table for a full adder is of the form

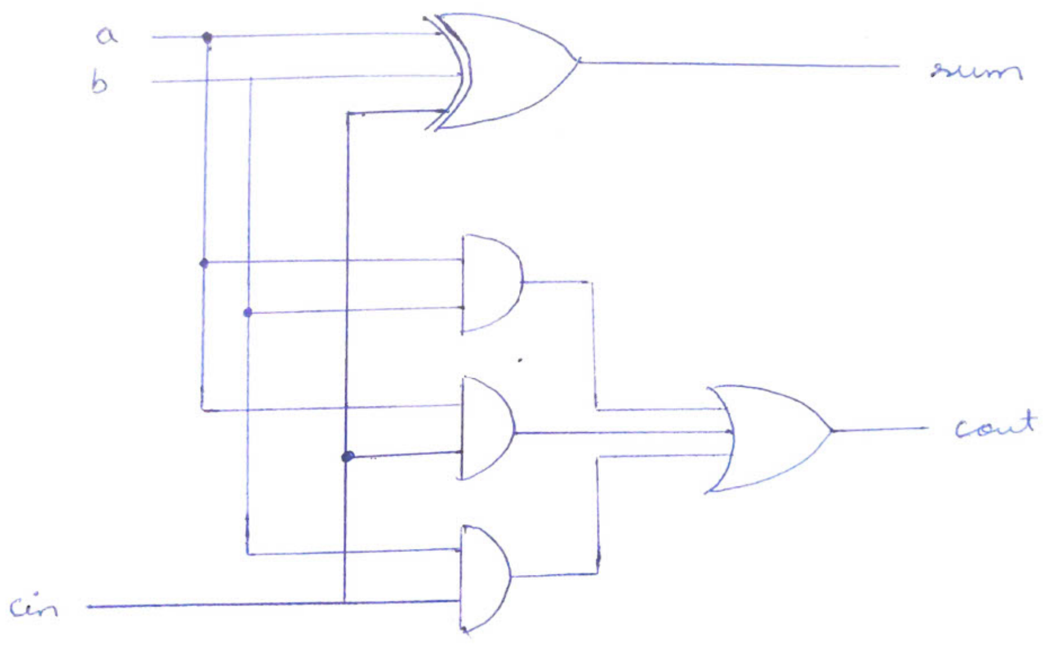
a	b	carryin	sum	$carryout$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

From the truth table we can derive the expressions for sum and carryout to be

$$sum[i] = a[i] \oplus b[i] \oplus carry[i]$$

$$carry[i+1] = a[i] \cdot b[i] + b[i] \cdot c[i] + a[i] \cdot c[i]$$

The full adder circuit is as follows



Module design

The first step in making the ADD module is the single bit full adder. We can construct the full adder as per the circuit shown above. Then we can use the single bit full adder to make the 32 bit ADD module which will use the single bit full adder on each bit of the inputs.

Note: Since we are using singed inputs we need to watch for overflow of the data values into the sign bit.

Modules

The XOR,AND and OR gate primitive in verilog follows the format

```
<gate> g1(ans,a,b);
```

where <gate> can be XOR, AND or OR, g1 is the name of the gate primitive instance, ans is the output and a and b are the inputs.

Note: More inputs can be added to the and gate by simple adding more input variables.

The module I have designed for the single bit full adder is as follows

```
`timescale 1ns / 1ps

module add1x1(
  input a,
  input b,
  input cin,
  output sum,
  output co);

  xor g1(sum,a,b,cin);
  and g2(k,a,b);
  and g3(l,a,cin);
  and g4(m,b,cin);
  or g5(co,k,l,m);

endmodule
```

The module I have designed for the 32 bit add operation that uses the single bit full adder is as follows

```
`timescale 1ns / 1ps

module add32x1(
  input signed [31:0]a,
  input signed [31:0]b,
  output signed [31:0]sum,
  output overflow);

  wire [32:0]c;
  assign c[0]=1'b0;

  genvar i;

  generate for(i=0; i<32; i=i+1)
  begin
    add1x1 g1(a[i],b[i],c[i],sum[i],c[i+1]);
  end
  endgenerate

  xor g2(overflow,c[31],c[32]);

endmodule
```

The overflow bit will be xor of the carry in and carry out of the sign bit addition because overflow ocuurs when:

1. Sum of 2 numbers with sign bits off yields a result with sign bit on
2. Sum of 2 numbers with sign bits on yields a result with sign bit off

Testbenches

To test the implemented single bit full adder I have used the following test bench

```
`timescale 1ns / 1ps

module add_test;
  reg a;
  reg b;
  reg cin;
  wire signed sum;
  wire carryout;

  add1x1 uut(
    .a(a),
    .b(b),
    .cin(cin),
    .sum(sum),
    .co(co)
  );

  initial begin
    $dumpfile("add_test.vcd");
    $dumpvars(0,add_test);
    a = 1'b0;
    b = 1'b0;

    #100;

    #20 a=1'd1;b=1'd0;cin=1'd0;
    #20 a=1'd1;b=1'd1;cin=1'd0;
    #20 a=1'd1;b=1'd0;cin=1'd1;
    #20 a=1'd0;b=1'd0;cin=1'd0;
  end

  initial begin
    $monitor("a=%b b=%b sum=%b carryout=%d\n",a,b,sum,co);
  end
endmodule
```

This testbench was used to test and veify the functioning of the full adder.

To test the implemented 32 bit add module I have used the following test bench

```
`timescale 1ns / 1ps

module add_test;
  reg signed [31:0]a;
  reg signed [31:0]b;

  wire signed [31:0]sum;
  wire overflow;

  add32x1 uut(
    .a(a),
    .b(b),
    .sum(sum),
    .overflow(overflow)
  );

  initial begin
    $dumpfile("add_test.vcd");
    $dumpvars(0,add_test);
    a = 32'b0;
    b = 32'b0;

    #100;

    #20 a=32'd2147483647;b=32'd1;
    #20 a=-32'd2147483648;b=-32'd1;
```

```
      #20 a=32'd23;b=-32'd0;
      #20 a=32'b1001;b=32'b1001;
      #20 a=32'b1001;b=-32'b1001;
      #20 a=-32'd2;b=32'd13;
      #20 a=-32'd2;b=-32'd13;
      #20 a=32'd2;b=-32'd13;
      #20 a=32'd0;b=32'd0;
    end

    initial
      $monitor("a=%d b=%d sum=%d overflow=%d\n",a,b,sum,overflow);
  endmodule
```

Results

On executing

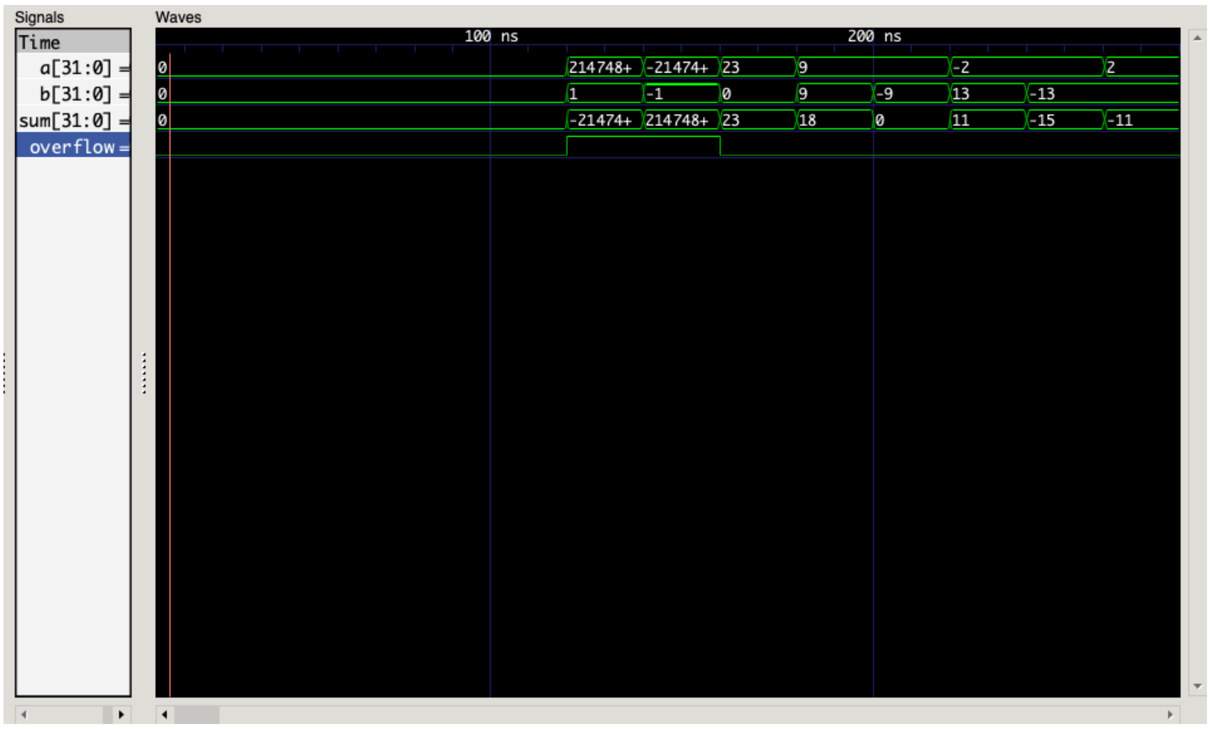
```
iverilog -o add add_test.v add1x1.v add32x1.v
vvp add
```

we get the following terminal output

```
a=          0 b=          0 sum=          0 overflow=0

a= 2147483647 b=          1 sum=-2147483648 overflow=1

a=-2147483648 b=         -1 sum= 2147483647 overflow=1

a=         23 b=          0 sum=         23 overflow=0

a=          9 b=          9 sum=         18 overflow=0

a=          9 b=         -9 sum=          0 overflow=0

a=         -2 b=         13 sum=         11 overflow=0

a=         -2 b=        -13 sum=        -15 overflow=0

a=          2 b=        -13 sum=        -11 overflow=0

a=          0 b=          0 sum=          0 overflow=0
```

We can see that all result values are correct for the respective input values of a and b. We can also note that the overflow bit is working correctly as it is detecting both negative and positive overflows and not giving any false alarms.

The gtkwave output waveform is as follows



The inputs and their respective outputs can be summarized as follows (in decimal)

a	b	$a + b$	overflow
0	0	0	0
2147483647	1	-2147483648	1
-2147483648	-1	2147483647	1
23	0	23	0
9	9	18	0
9	-9	0	0
-2	13	11	0
-2	-13	-15	0
2	-13	-11	0

SUB

Logic, Circuit and Module design

The truth table for the SUB operation will be as follows

a	b	$a-b$	$borrow$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

We can see that rather than implementing a subtractor circuit it will be easier to implement this in the form of an adder since we already have the adder modules ready.

$$y = a - b$$

$$\implies y = a + (-b)$$

To convert this subtraction into addition we need to take the 2s complement of b and then add it to a.

We know that the 2s complement is calculated by inverting all the bits and adding 1. In order to do this we need a 32 bit inverter circuit.

Module design

We have to take the 2s complement of b in order to be able to simply add them. The 32 bit inverter can be constructed in a way similar to the rest of our 32 bit gate modules. We make use of the NOT gate primitive in verilog to perform the single bit inversion. Following the moduar approach I used a generate block in the 32 bit module to instatiate the single bit module 32 times for the 32 bits of the input. This gives us a 32 bit inverter. To find 2s complement we simple add 1 to the result from the inverter. Then we can add a and the 2s complement of b which will give us the result of the subtraction of b from a.

Modules

The XOR,AND and OR gate primitive in verilog follows the format

```
<gate> g1(ans,a,b);
```

where <gate> can be XOR, AND or OR, g1 is the name of the gate primitive instance, ans is the output and a and b are the inputs.

Note: More inputs can be added to the and gate by simple adding more input variables.

The NOT gate primitive in verilog follows the format

```
not g1(ans,b);
```

where g1 is the name of the gate primitive instance, ans is the output and b is the input.

The module I have designed for the single bit inverter is as follows

```
`timescale 1ns / 1ps

module not1x1(
  input a,
  output ans
  );

  not g1(ans,a);

endmodule
```

The module I have designed for the 32 bit inverter that uses the single bit inverter module is as follows

```
`timescale 1ns / 1ps

module not32x1(
  input signed [31:0]a,
  output signed [31:0]ans
  );
  genvar i;

  generate for(i=0; i<32; i=i+1)
  begin
    not1x1 g1(a[i],ans[i]);
  end
  endgenerate

endmodule
```

And using this 32 bit inverter module and the 32 bit adder module I have designed the subtractor module as follows

```
`timescale 1ns / 1ps

module sub32x1(
  input signed [31:0]a,
  input signed [31:0]b,
  output signed [31:0]ans,
  output overflow);

  wire [31:0]nb;
  not32x1 g1(b,nb);

  wire [31:0]l;
  assign l=32'b1;

  wire [31:0]bcomp;
  add32x1 g2(nb,l,bcomp,c);

  add32x1 g3(a,bcomp,ans,overflow);

endmodule
```

Testbench

To test the inverter modules I have used the following test bench

```
`timescale 1ns / 1ps

module not_test;
  reg signed [31:0]a;
  reg signed [31:0]b;

  wire signed [31:0]ans;

  not32x1 uut(
    .a(a),
    .ans(ans)
  );
  initial begin
    $dumpfile("not_test.vcd");
    $dumpvars(0,not_test);
    a = 32'b0;

    #100;

    #20 a=32'b1011;
    #20 a=32'b1011;
    #20 a=-32'b1011;
    #20 a=32'b1001;
    #20 a=-32'd2;
    #20 a=-32'd2;
    #20 a=32'b1001;
  end

  initial begin
    $monitor("a=%b b=%b ans=%b\n",a,b,ans);
  end
endmodule
```

This testbench was used to verify the functioning of the 32 bit inverter module.

To test the subtractor module I have used the following test bench

```
`timescale 1ns / 1ps

module sub_test;
  reg signed [31:0]a;
  reg signed [31:0]b;
  wire signed [31:0]ans;
  wire overflow;

  sub32x1 uut(
    .a(a),
    .b(b),
    .ans(ans),
    .overflow(overflow)
  );

  initial begin
    $dumpfile("sub_test.vcd");
    $dumpvars(0,sub_test);
    a = 32'b0;
    b = 32'b0;

    #100;

    #20 a=32'd2147483647;b=-32'd1;
    #20 a=-32'd2147483648;b=32'd1;
    #20 a=32'd23;b=-32'd0;
    #20 a=32'b1001;b=32'b1001;
    #20 a=32'b1001;b=-32'b1001;
    #20 a=-32'd2;b=32'd13;
    #20 a=-32'd2;b=-32'd13;
    #20 a=32'd2;b=-32'd13;
    #20 a=32'd0;b=32'd0;
  end

  initial
    $monitor("a=%d b=%d ans=%d overflow=%d\n",a,b,ans,overflow);
endmodule
```

Results

On executing

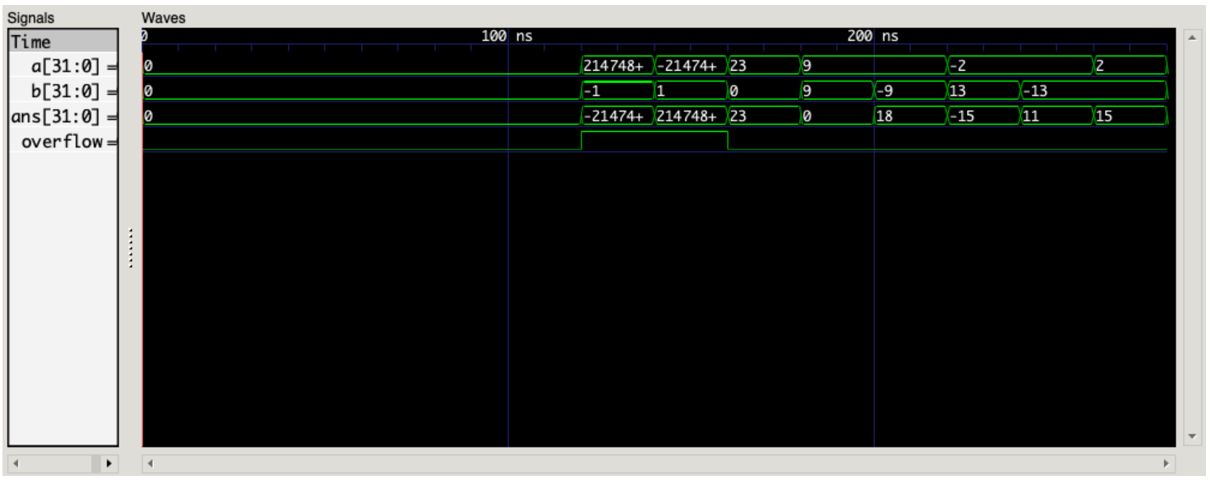
```
iverilog -o sub sub_test.v sub32x1.v not/not1x1.v not/not32x1.v ../Add/add32x1.v ../Ad
d/add1x1.vvvp
vvp sub
```

we get the following terminal output

```
a=          0 b=          0 ans=          0 overflow=0

a= 2147483647 b=         -1 ans=-2147483648 overflow=1

a=-2147483648 b=          1 ans= 2147483647 overflow=1

a=         23 b=          0 ans=         23 overflow=0

a=          9 b=          9 ans=          0 overflow=0

a=          9 b=         -9 ans=         18 overflow=0

a=         -2 b=         13 ans=        -15 overflow=0

a=         -2 b=        -13 ans=         11 overflow=0

a=          2 b=        -13 ans=         15 overflow=0

a=          0 b=          0 ans=          0 overflow=0
```

We can see that all result values are correct for the respective input values of a and b. We can also note that the overflow bit is working correctly as it is detecting both negative and positive overflows and not giving any false alarms.

The gtkwave output waveform is as follows



The inputs and their respective outputs can be summarized as follows (in decimal)

a	b	$a - b$	overflow
0	0	0	0
2147483647	1	-2147483648	1
-2147483648	-1	2147483647	1
23	0	23	0
9	9	0	0
9	-9	18	0
-2	13	-15	0
-2	-13	11	0
2	-13	15	0

ALU

Logic, Circuit and Module design

The final ALU wrapper should be able to call each of the modules implemented above based on the control input. The ALU unit should take as input the control signal and two 32-bit inputs and return the 32-bit output corresponding to the control input chosen.

The control inputs and their respective functionalities are as follows

Control Input		Functionality
2'b00	0	ADD
2'b01	1	SUB
2'b10	2	AND
2'b11	3	XOR

Module design

The ALU wrapper simply consists of switching statements or case statements to select the operation output to be returned as the ALU output based on the control input.

Module

The module I have designed for the ALU wrapper is as follows

```
`timescale 1ns / 1ps

`include "../Add/add32x1.v"
`include "../Add/add1x1.v"
`include "../Sub/sub32x1.v"
`include "../Sub/not/not1x1.v"
`include "../Sub/not/not32x1.v"
`include "../Xor/xor32x1.v"
`include "../Xor/xor1x1.v"
`include "../And/and32x1.v"
`include "../And/and1x1.v"

module alu(
  input [1:0]control,
  input signed [31:0]a,
  input signed [31:0]b,
  output signed [31:0]ans,
  output overflow
  );

  wire signed [31:0]ans1;
  wire signed [31:0]ans2;
  wire signed [31:0]ans3;
  wire signed [31:0]ans4;
  reg signed [31:0]ansfinal;
  reg overflowfinal;

  add32x1 g1(a,b,ans1,overflow1);
  sub32x1 g2(a,b,ans2,overflow2);
```

```
    and32x1 g3(a,b,ans3);
    xor32x1 g4(a,b,ans4);
    always @(*)
    begin
      case(control)
        2'b00:begin
            ansfinal=ans1;
            overflowfinal=overflow1;
          end
        2'b01:begin
            ansfinal=ans2;
            overflowfinal=overflow2;
          end
        2'b10:begin
            ansfinal=ans3;
            overflowfinal=1'b0;
          end
        2'b11:begin
            ansfinal=ans4;
            overflowfinal=1'b0;
          end
      endcase
    end

    assign ans= ansfinal;
    assign overflow= overflowfinal;
  endmodule
```

Testbench

To test the ALU I have used the following test bench

```
`timescale 1ns / 1ps

module Alu_test;
  reg [1:0]control;
  reg signed [31:0]a;
  reg signed [31:0]b;

  wire signed [31:0]ans;
  wire overflow;

  alu uut(
    .control(control),
    .a(a),
    .b(b),
    .ans(ans),
    .overflow(overflow)
  );

  initial begin
    $dumpfile("Alu_test.vcd");
    $dumpvars(0,Alu_test);
    control=2'b00;
    a = 32'b0;
    b = 32'b0;

    #100;

    #20 control=2'b00;a=32'b1011;b=32'b0100;
    #20 control=2'b01;a=32'b1011;b=32'b0100;
    #20 control=2'b10;a=32'b1011;b=32'b0100;
    #20 control=2'b11;a=32'b1011;b=32'b0100;
    #20 control=2'b00;a=-32'b1011;b=32'b0100;
    #20 control=2'b01;a=-32'b1011;b=32'b0100;
    #20 control=2'b10;a=-32'b1011;b=32'b0100;
    #20 control=2'b11;a=-32'b1011;b=32'b0100;
    #20 control=2'b00;a=32'b1011;b=-32'b0100;
    #20 control=2'b01;a=32'b1011;b=-32'b0100;
    #20 control=2'b10;a=32'b1011;b=-32'b0100;
    #20 control=2'b11;a=32'b1011;b=-32'b0100;
    #20 control=2'b00;a=-32'b1011;b=-32'b0100;
    #20 control=2'b01;a=-32'b1011;b=-32'b0100;
    #20 control=2'b10;a=-32'b1011;b=-32'b0100;
    #20 control=2'b11;a=-32'b1011;b=-32'b0100;
    #20 control=2'b00;a=32'd2147483647;b=32'd1;
    #20 control=2'b01;a=32'd2147483647;b=-32'd1;
    #20 control=2'b00;a=32'b0;b=32'b0;
  end

  initial
    $monitor("control=%b a=%b b=%b ans=%b overflow=%b\n",control,a,b,ans,overflow);
endmodule
```

Results

On executing

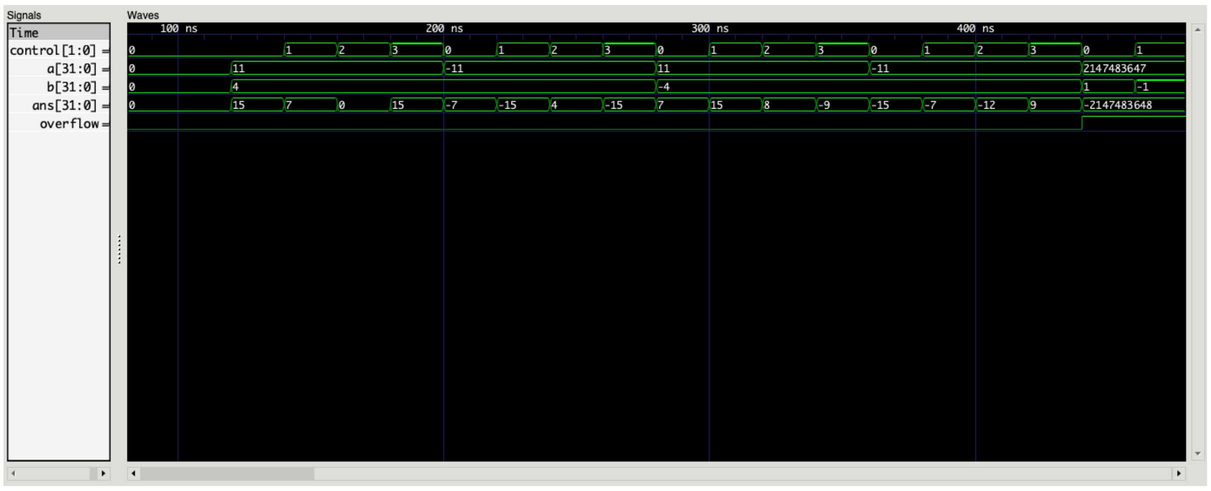
```
iverilog -o alu alu_test.v alu.v
vvp alu
```

we get the following terminal output

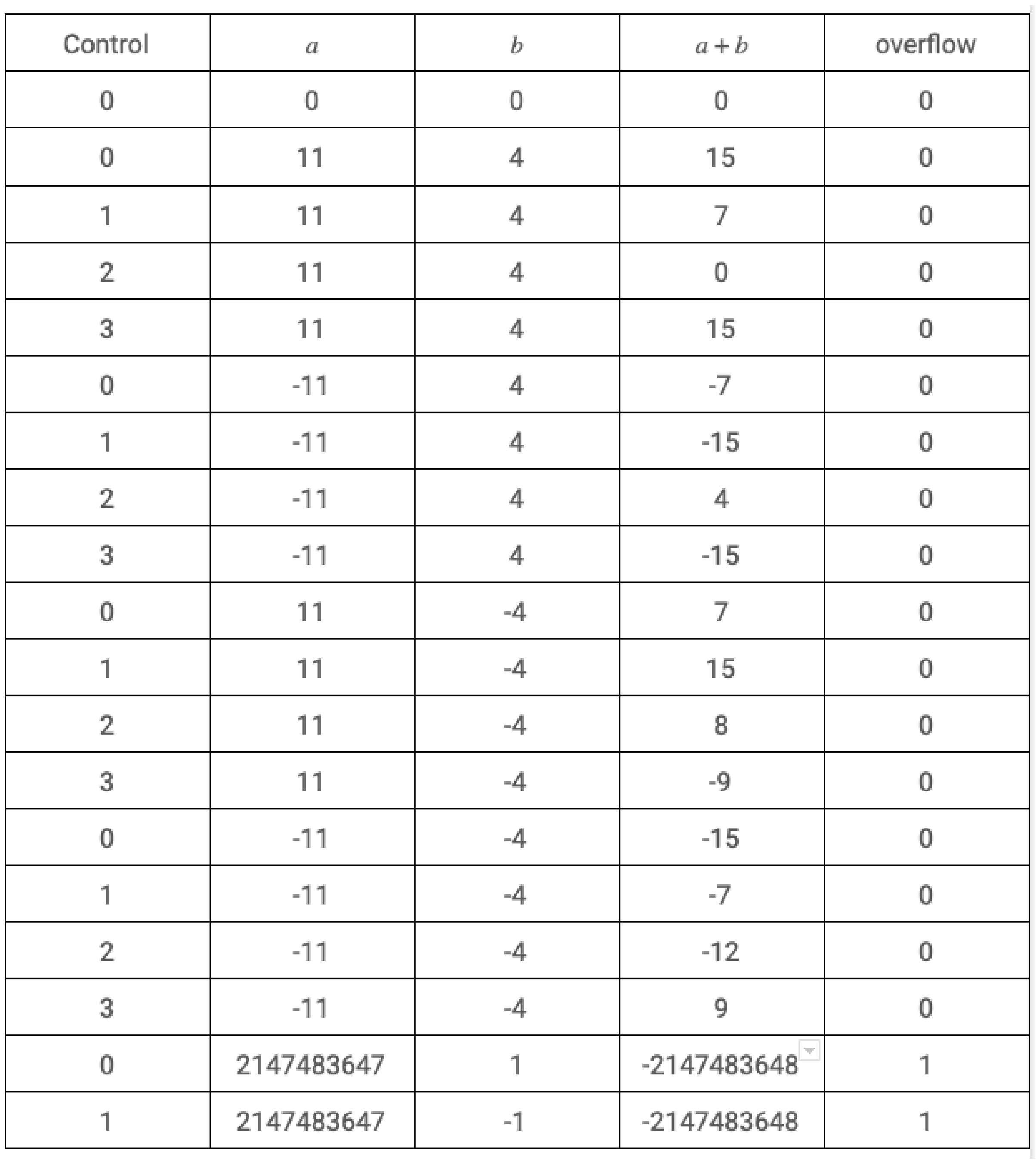
```
control=00 a=00000000000000000000000000000000 b=00000000000000000000000000000000 ans=0
0000000000000000000000000000000 overflow=0

control=00 a=00000000000000000000000000001011 b=00000000000000000000000000000100 ans=0
0000000000000000000000000001111 overflow=0

control=01 a=00000000000000000000000000001011 b=00000000000000000000000000000100 ans=0
0000000000000000000000000000111 overflow=0
```

```
control=10 a=00000000000000000000000000001011 b=00000000000000000000000000000100 ans=0
0000000000000000000000000000000 overflow=0

control=11 a=00000000000000000000000000001011 b=00000000000000000000000000000100 ans=0
0000000000000000000000000001111 overflow=0

control=00 a=11111111111111111111111111110101 b=00000000000000000000000000000100 ans=1
1111111111111111111111111111001 overflow=0

control=01 a=11111111111111111111111111110101 b=00000000000000000000000000000100 ans=1
1111111111111111111111111110001 overflow=0

control=10 a=11111111111111111111111111110101 b=00000000000000000000000000000100 ans=0
0000000000000000000000000000100 overflow=0

control=11 a=11111111111111111111111111110101 b=00000000000000000000000000000100 ans=1
1111111111111111111111111110001 overflow=0

control=00 a=00000000000000000000000000001011 b=11111111111111111111111111111100 ans=0
0000000000000000000000000000111 overflow=0

control=01 a=00000000000000000000000000001011 b=11111111111111111111111111111100 ans=0
0000000000000000000000000001111 overflow=0

control=10 a=00000000000000000000000000001011 b=11111111111111111111111111111100 ans=0
0000000000000000000000000001000 overflow=0

control=11 a=00000000000000000000000000001011 b=11111111111111111111111111111100 ans=1
1111111111111111111111111110111 overflow=0

control=00 a=11111111111111111111111111110101 b=11111111111111111111111111111100 ans=1
1111111111111111111111111110001 overflow=0

control=01 a=11111111111111111111111111110101 b=11111111111111111111111111111100 ans=1
1111111111111111111111111111001 overflow=0

control=10 a=11111111111111111111111111110101 b=11111111111111111111111111111100 ans=1
1111111111111111111111111110100 overflow=0

control=11 a=11111111111111111111111111110101 b=11111111111111111111111111111100 ans=0
0000000000000000000000000001001 overflow=0

control=00 a=01111111111111111111111111111111 b=00000000000000000000000000000001 ans=1
0000000000000000000000000000000 overflow=1

control=01 a=01111111111111111111111111111111 b=11111111111111111111111111111111 ans=1
0000000000000000000000000000000 overflow=1

control=00 a=00000000000000000000000000000000 b=00000000000000000000000000000000 ans=0
0000000000000000000000000000000 overflow=0
```

The gtkwave output waveform is as follows



The inputs and their respective outputs can be summarized as follows (in decimal)



Control	a	b	$a+b$	overflow
0	0	0	0	0
0	11	4	15	0
1	11	4	7	0
2	11	4	0	0
3	11	4	15	0
0	-11	4	-7	0
1	-11	4	-15	0
2	-11	4	4	0
3	-11	4	-15	0
0	11	-4	7	0
1	11	-4	15	0
2	11	-4	8	0
3	11	-4	-9	0
0	-11	-4	-15	0
1	-11	-4	-7	0
2	-11	-4	-12	0
3	-11	-4	9	0
0	2147483647	1	-2147483648	1
1	2147483647	-1	-2147483648	1

We can see that all result values are correct for the respective input values of a and b. We can also note that the overflow bit is working correctly and not giving any false alarms.

Hence, I have successfully created an ALU with the required functionalities and verified the working of the ALU as well as the individual modules for each functionality.

The testbench used for testing the execute stage is as follows

```
`timescale 1ns / 1ps

module fetchdecodetb;
  reg clk;
  reg [63:0] PC;
  reg [63:0] reg_mem[0:14];

  wire [3:0] icode;
  wire [3:0] ifun;
  wire [3:0] rA;
  wire [3:0] rB;
  wire [63:0] valC;
  wire [63:0] valP;
  wire [63:0] valA;
  wire [63:0] valB;
  wire [63:0] valE;
  wire cnd;


  fetch fetch(
    .clk(clk),
    .PC(PC),
    .icode(icode),
    .ifun(ifun),
    .rA(rA),
    .rB(rB),
    .valC(valC),
    .valP(valP)
  );

  decode decode(
    .clk(clk),
    .icode(icode),
    .rA(rA),
    .rB(rB),
    .reg_memrA(reg_mem[rA]),
    .reg_memrB(reg_mem[rB]),
    .reg_memr4(reg_mem[4]),
    .valA(valA),
    .valB(valB)
  );

  execute execute(
    .clk(clk),
    .icode(icode),
    .ifun(ifun),
    .valA(valA),
    .valB(valB),
    .valC(valC),
    .valE(valE),
    .cnd(cnd)
  );

  initial begin
```

```
      reg_mem[0]=64'd0;
      reg_mem[1]=64'd1;
      reg_mem[2]=64'd2;
      reg_mem[3]=64'd3;
      reg_mem[4]=64'd4;
      reg_mem[5]=64'd5;
      reg_mem[6]=64'd6;
      reg_mem[7]=64'd7;
      reg_mem[8]=64'd8;
      reg_mem[9]=64'd9;
      reg_mem[10]=64'd10;
      reg_mem[11]=64'd11;
      reg_mem[12]=64'd12;
      reg_mem[13]=64'd13;
      reg_mem[14]=64'd14;

      clk=0;
      PC=64'd0;

      #10 clk=~clk;PC=64'd0;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;PC=valP;
      #10 clk=~clk;
      #10 clk=~clk;
    end

    initial
      $monitor("clk=%d icode=%b ifun=%b rA=%b rB=%b valA=%d valB=%d valE=%d\n",clk,icod
e,ifun,rA,rB,valA,valB,valE);
endmodule
```

Note: Extensive testing was done for all 5 stages before combining them together into the sequential and pipelined models. Only the key testbenches and outputs are mentioned in this report.

Challenges faced

- Some of the challenges I faced are as follows
    - It took some time to go from thinking about the functional level design of the Y86-64 processor to the hardware circuit level.
    - Figuring out the control logic and implementing it correctly was also rather challenging especially in case of the pipelined processor.

Acknowledgement

Working on this project has been a great learning experience. Over the last mont, I have developed a deeper understanding of processor architecture design, instruction set architecture, memory and many of the other concepts required for this project.

I would like to thank Prof. Deepak Gangadharan and the TAs for guiding me throughout the duration of this project.

Adithya Sunil