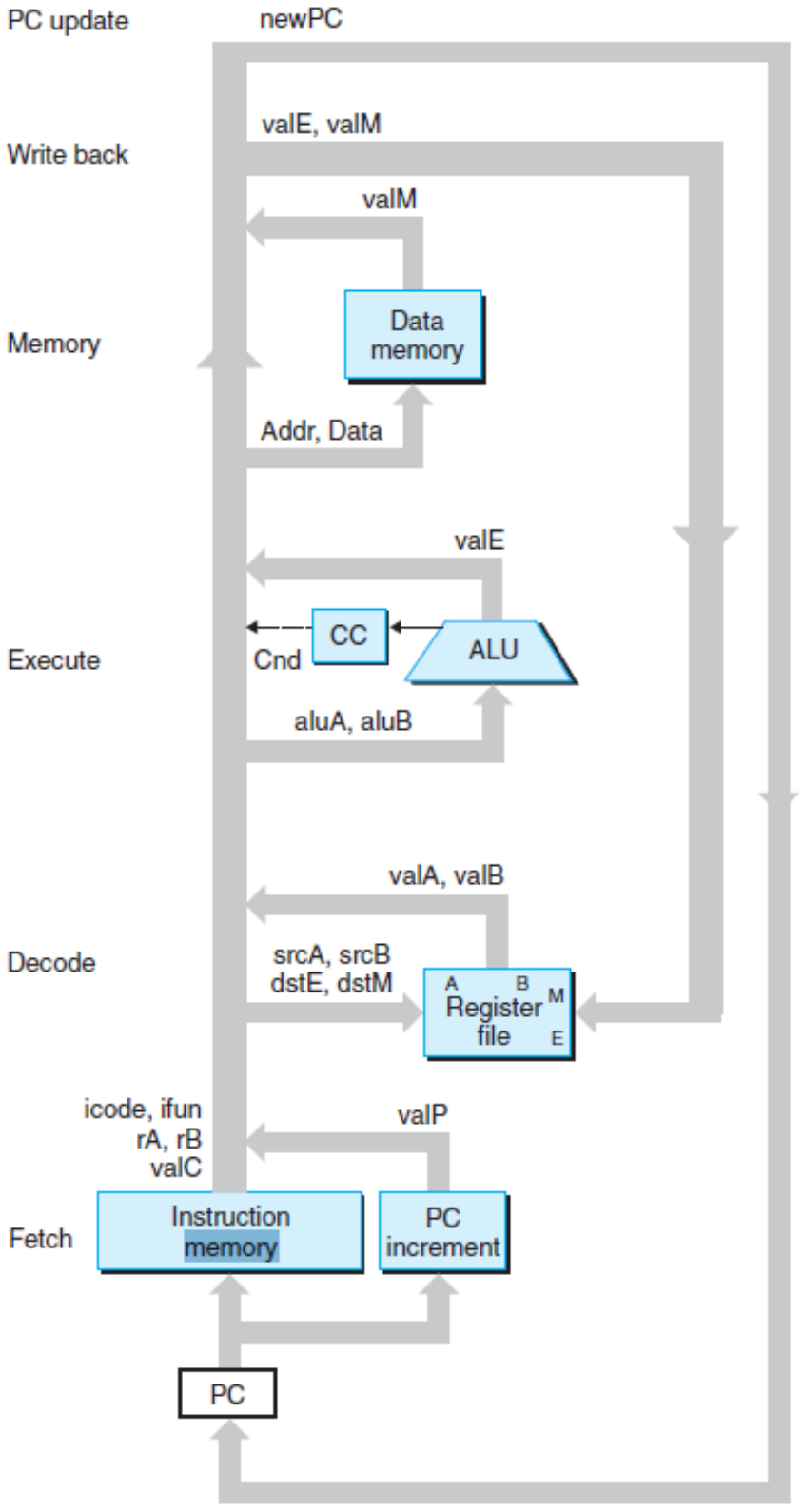
IPA Project

Mid-evals Project Report

By:Aakash Gorla(2020102034)
Anish Mathur(2020102044)

Overview:

The main aim of this part of the project is to implement a sequential design based on the Y86-64 ISA protocol in Verilog. We split the sequential design into 6 parts: Fetch,Decode, Execute, Memory, Pc update and write back.


PC update
newPC
Write back
valE, valM
valM
Memory
Data memory
Addr, Data
valE
Execute
CC
ALU
Cnd
aluA, aluB
valA, valB
Decode
srcA, srcB
dstE, dstM
A
B
Register file
M
E
icode, ifun
rA, rB
valC
valP
Fetch
Instruction memory
PC increment
PC

Fetch:

- In this stage we read each instruction from the memory and determine the values of icode, ifun, rA, rB, valC.
- Here is a description of the implementation of fetch:
  - The PC value gets updated on the positive edge of the clock and checks the instruction memory.
  - The process will only continue if PC is not out of bounds. It will then take 10 bytes and form a register
  - Now icode and ifun can be determined from the bytes of the reg instr.

Source Code:

```
module fetch(
  clk,PC,
  icode,ifun,rA,rB,valC,valP,instr_val,imem_er,halt
);

  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_val;
  output reg imem_er;
  output reg halt;

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

  reg [0:79] instr;

  initial begin
  //Instruction memory
```

```
    instr_mem[0]=8'b00100000;
     instr_mem[1]=8'b00010011;
     instr_mem[2]= 8'b00110000;
    instr_mem[3]= 8'b00000010;
     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;
     instr_mem[10]=8'b00000000;
     instr_mem[11]=8'b00010001;
     instr_mem[12]=8'b01000000;
     instr_mem[13]=8'b01010010;
     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'b00000000;
     instr_mem[20]=8'b00000000;
     instr_mem[21]=8'b00000001;
     instr_mem[22]=8'b01010000;
     instr_mem[23]=8'b01110010;
     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'b00000000;
     instr_mem[30]=8'b00000000;
     instr_mem[31]=8'b00000001;
     instr_mem[32]=8'b00000000;
  instr_mem[31]=8'b00010000;
    instr_mem[32]=8'b01100000;
    instr_mem[33]=8'b00100011;
    instr_mem[34]=8'b01110000;
    instr_mem[35]=8'b00000000;
```

```
    instr_mem[36]=8'b00000000;
    instr_mem[37]=8'b00000000;
    instr_mem[38]=8'b00000000;
    instr_mem[39]=8'b00000000;
    instr_mem[40]=8'b00000000;
    instr_mem[41]=8'b00000000;
    instr_mem[42]=8'b00100000;
  instr_mem[43]=8'b00010000;
  instr_mem[44]=8'b00010000;
  instr_mem[45]=8'b00010000;
  instr_mem[46]=8'b00010000;
  instr_mem[47]=8'b00010000;
  instr_mem[48]=8'b00010000;
  instr_mem[49]=8'b00010000;
    instr_mem[50]=8'b00000000;
    instr_mem[34]=8'b00010000;
   instr_mem[35]=8'b00010000;
     instr_mem[36]=8'b00100000;
     instr_mem[37]=8'b00000100;
     instr_mem[39]=8'b00010000;
     instr_mem[40]=8'b00010000;
     instr_mem[41]=8'b00010000;
     instr_mem[42]=8'b00010000;
     instr_mem[43]=8'b00010000;
     instr_mem[38]=8'b00010000;
     instr_mem[39]=8'b00010000;
     instr_mem[40]=8'b00010000;
     instr_mem[41]=8'b00010000;
     instr_mem[37]=8'b00010000;
     instr_mem[43]=8'b10000000;
     instr_mem[44]=8'b00000000;
     instr_mem[45]=8'b00000000;
     instr_mem[46]=8'b00000000;
    instr_mem[47]=8'b00000000;
     instr_mem[48]=8'b00000000;
     instr_mem[49]=8'b00000000;
     instr_mem[50]=8'b00000000;
     instr_mem[51]=8'b00000000;
```

```
  instr_mem[52]=8'b10010000;
   instr_mem[53]=8'b10100000;
   instr_mem[54]=8'b00000000;
   instr_mem[55]=8'b10110000;
   instr_mem[56]=8'b00000000;
   instr_mem[57]=8'b00010000;
   instr_mem[58]=8'b00000000;
   instr_mem[0]=8'b00110000;
   instr_mem[1]=8'b00000000;
   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;
   instr_mem[10]=8'b00110000;
   instr_mem[11]=8'b00000010;
   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;
   instr_mem[20]=8'b00110000;
   instr_mem[21]=8'b00000011;
   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;
   instr_mem[30]=8'b01110000;
```

```
    instr_mem[31]=8'b00000000;
    instr_mem[32]=8'b00000000;
    instr_mem[33]=8'b00000000;
    instr_mem[34]=8'b00000000;
    instr_mem[35]=8'b00000000;
    instr_mem[36]=8'b00000000;
    instr_mem[37]=8'b00000000;
    instr_mem[38]=8'b00100111;
    instr_mem[39]=8'b01100000;
    instr_mem[40]=8'b00000011;
    instr_mem[41]=8'b01110011;
    instr_mem[42]=8'b00000000;
    instr_mem[43]=8'b00000000;
    instr_mem[44]=8'b00000000;
    instr_mem[45]=8'b00000000;
    instr_mem[46]=8'b00000000;
    instr_mem[47]=8'b00000000;
    instr_mem[48]=8'b00000000;
    instr_mem[49]=8'b01111010;
    instr_mem[50]=8'b01100000;
    instr_mem[51]=8'b00000010;
    instr_mem[52]=8'b01110011;
    instr_mem[53]=8'b00000000;
    instr_mem[54]=8'b00000000;
    instr_mem[55]=8'b00000000;
    instr_mem[56]=8'b00000000;
    instr_mem[57]=8'b00000000;
    instr_mem[58]=8'b00000000;
    instr_mem[59]=8'b00000000;
    instr_mem[60]=8'b01111101;
    instr_mem[61]=8'b01110000;
    instr_mem[62]=8'b00000000;
    instr_mem[63]=8'b00000000;
    instr_mem[64]=8'b00000000;
    instr_mem[65]=8'b00000000;
    instr_mem[66]=8'b00000000;
    instr_mem[67]=8'b00000000;
    instr_mem[68]=8'b00000000;
```

```
    instr_mem[69]=8'b01000110;
    instr_mem[70]=8'b00100000;
    instr_mem[71]=8'b00100110;
    instr_mem[72]=8'b00100000;
    instr_mem[73]=8'b00110111;
    instr_mem[74]=8'b01100001;
    instr_mem[75]=8'b00110110;
    instr_mem[76]=8'b01110001;
    instr_mem[77]=8'b00000000;
    instr_mem[78]=8'b00000000;
    instr_mem[79]=8'b00000000;
    instr_mem[80]=8'b00000000;
    instr_mem[81]=8'b00000000;
    instr_mem[82]=8'b00000000;
    instr_mem[83]=8'b00000000;
    instr_mem[84]=8'b01100000;
    instr_mem[85]=8'b01100001;
    instr_mem[86]=8'b00100111;
    instr_mem[87]=8'b01110001;
    instr_mem[88]=8'b00000000;
    instr_mem[89]=8'b00000000;
    instr_mem[90]=8'b00000000;
    instr_mem[91]=8'b00000000;
    instr_mem[92]=8'b00000000;
    instr_mem[93]=8'b00000000;
    instr_mem[94]=8'b00000000;
    instr_mem[95]=8'b01101101;
    instr_mem[96]=8'b00100000;
    instr_mem[97]=8'b00110010;
    instr_mem[98]=8'b00100000;
    instr_mem[99]=8'b01100011;
    instr_mem[100]=8'b01110000;
    instr_mem[101]=8'b00000000;
    instr_mem[102]=8'b00000000;
    instr_mem[103]=8'b00000000;
    instr_mem[104]=8'b00000000;
    instr_mem[105]=8'b00000000;
    instr_mem[106]=8'b00000000;
```

```
 instr_mem[107]=8'b00000000;
 instr_mem[108]=8'b00100111;
 instr_mem[109]=8'b00100000;
 instr_mem[110]=8'b00110010;
 instr_mem[111]=8'b00100000;
 instr_mem[112]=8'b01110011;
 instr_mem[113]=8'b01110000;
 instr_mem[114]=8'b00000000;
 instr_mem[115]=8'b00000000;
 instr_mem[116]=8'b00000000;
 instr_mem[117]=8'b00000000;
 instr_mem[118]=8'b00000000;
 instr_mem[119]=8'b00000000;
 instr_mem[120]=8'b00000000;
 instr_mem[121]=8'b00100111;
 instr_mem[122]=8'b00100000;
 instr_mem[123]=8'b00100001;
 instr_mem[124]=8'b00000000;


 instr_mem[125]=8'b00100000;
 instr_mem[126]=8'b00110001;

 instr_mem[127]=8'b00000000;

end

always@(posedge clk)
begin

  imem_er=0;
  if(PC>1023)
  begin
    imem_er=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_val=1'b1;
  case(icode)
    4'd0: //halt
    begin
      halt=1;
      valP=PC+64'd1;
    end
    4'd1: //nop
    begin
      valP=PC+64'd1;
    end
    4'd2://cmovxx
    begin
      rA=instr[8:11];
      rB=instr[12:15];
      valP=PC+64'd2;
    end
    4'd3: //irmovq
    begin
      rA=instr[8:11];
      rB=instr[12:15];
      valC=instr[16:79];
      valP=PC+64'd10;
    end
```

```
    4'd4: //rmmovq
    begin
      rA=instr[8:11];
      rB=instr[12:15];
      valC=instr[16:79];
      valP=PC+64'd10;
    end
    4'd5: //mrmovq
    begin
      rA=instr[8:11];
      rB=instr[12:15];
      valC=instr[16:79];
      valP=PC+64'd10;
    end
    4'd6: //OPq
    begin
      rA=instr[8:11];
      rB=instr[12:15];
      valP=PC+64'd2;
    end
    4'd7: //jxx
    begin
      valC=instr[8:71];
      valP=PC+64'd9;
    end
    4'd8: //call
    begin
      valC=instr[8:71];
      valP=PC+64'd9;
    end
    4'd9: //ret
    begin
      valP=PC+64'd1;
    end
    4'd10: //pushq
    begin
      rA=instr[8:11];
      rB=instr[12:15];
```

```
      valP=PC+64'd2;
    end
    4'd11: //popq
    begin
      rA=instr[8:11];
      rB=instr[12:15];
      valP=PC+64'd2;
    end
    default:
    begin
      instr_val=1'b0;
    end
  endcase
  end

endmodule
```

Output:

```
clk=0 PC=                0 icode=xxxx ifun=xxxx rA=xxxx rB=xxxx,valC=                x,valP=                x
clk=1 PC=               32 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               33
clk=0 PC=               32 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               33
clk=1 PC=               33 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               34
clk=0 PC=               33 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               34
clk=1 PC=               34 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               35
clk=0 PC=               34 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               35
clk=1 PC=               35 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               36
clk=0 PC=               35 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               36
clk=1 PC=               36 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               37
clk=0 PC=               36 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               37
clk=1 PC=               37 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               38
clk=0 PC=               37 icode=0000 ifun=0000 rA=xxxx rB=xxxx,valC=                x,valP=               38
clk=1 PC=               38 icode=0010 ifun=0111 rA=0110 rB=0000,valC=                x,valP=               40
clk=0 PC=               38 icode=0010 ifun=0111 rA=0110 rB=0000,valC=                x,valP=               40
clk=1 PC=               40 icode=0000 ifun=0011 rA=0110 rB=0000,valC=                x,valP=               41
clk=0 PC=               40 icode=0000 ifun=0011 rA=0110 rB=0000,valC=                x,valP=               41
clk=1 PC=               41 icode=0111 ifun=0011 rA=0110 rB=0000,valC=              122,valP=               50
clk=0 PC=               41 icode=0111 ifun=0011 rA=0110 rB=0000,valC=              122,valP=               50
clk=1 PC=               50 icode=0110 ifun=0000 rA=0000 rB=0010,valC=              122,valP=               52
clk=0 PC=               50 icode=0110 ifun=0000 rA=0000 rB=0010,valC=              122,valP=               52
clk=1 PC=               52 icode=0111 ifun=0011 rA=0000 rB=0010,valC=              125,valP=               61
clk=0 PC=               52 icode=0111 ifun=0011 rA=0000 rB=0010,valC=              125,valP=               61
clk=1 PC=               61 icode=0111 ifun=0000 rA=0000 rB=0010,valC=               70,valP=               70
```

Decode:

In this stage we calculate the values of valA and valB based on the values in rA and rB. For some instructions it also looks at the stack pointer (%rsp).

Description:

We create the register array, and then compute $valA$ and $valB$, by using the rA and rB values as indices of the register array.

Then for different values of $icode$ we assign the values of valA and valB.

Source Code:

```
module decode (
    input clk,
    input [3:0] icode,
    input [3:0] rA,
    input [3:0] rB,
    input [63:0] mem_regA,
    input [63:0] mem_regB,
    input [63:0] stack_ptr,
    output reg [63:0] valA,
    output reg [63:0] valB
);

always @(posedge clk)
begin
    case (icode)
        4'b0110: begin //OPq
            valA = mem_regA;
            valB = mem_regB;
        end

        4'b0100: begin //rmmovq
            valA = mem_regA;
            valB = mem_regB;
        end

        4'b1011: begin //popq
            valA = stack_ptr;
```

```
            valB = stack_ptr;
        end

        4'b0010: begin //cmovxx
            valA = mem_regA;
            valB = 64'b0;
        end

        4'b0111: begin //jxx
        end

        4'b1000: begin // call
            valB = stack_ptr;
        end

        4'b1010: begin //pushq
            valA = mem_regA;
            valB = stack_ptr;
        end

        4'b0011: begin //irmovq
        end

        4'b1001: begin //ret
            valA = stack_ptr;
            valB = stack_ptr;
        end

        4'b0101: begin //mrmovq
            valB = mem_regB;
        end

    endcase
end

endmodule
```

Output:

```
clk=1 icode=0011 ifun=0000 rA=0000 rB=0000 valA=                x valB=                x
clk=0 icode=0011 ifun=0000 rA=0000 rB=0000 valA=                x valB=                x
clk=1 icode=0011 ifun=0000 rA=0000 rB=0010 valA=                x valB=                x
clk=0 icode=0011 ifun=0000 rA=0000 rB=0010 valA=                x valB=                x
clk=1 icode=0011 ifun=0000 rA=0000 rB=0011 valA=                x valB=                x
clk=0 icode=0011 ifun=0000 rA=0000 rB=0011 valA=                x valB=                x
clk=1 icode=0111 ifun=0000 rA=0000 rB=0011 valA=                x valB=                x
clk=0 icode=0111 ifun=0000 rA=0000 rB=0011 valA=                x valB=                x
clk=1 icode=0110 ifun=0000 rA=0000 rB=0011 valA=                x valB=                x
clk=0 icode=0110 ifun=0000 rA=0000 rB=0011 valA=                x valB=                x
clk=1 icode=0111 ifun=0011 rA=0000 rB=0011 valA=                x valB=                x
clk=0 icode=0111 ifun=0011 rA=0000 rB=0011 valA=                x valB=                x
clk=1 icode=0110 ifun=0000 rA=0000 rB=0010 valA=                x valB=                x
clk=0 icode=0110 ifun=0000 rA=0000 rB=0010 valA=                x valB=                x
clk=1 icode=0111 ifun=0011 rA=0000 rB=0010 valA=                x valB=                x
clk=0 icode=0111 ifun=0011 rA=0000 rB=0010 valA=                x valB=                x
clk=1 icode=0111 ifun=0000 rA=0000 rB=0010 valA=                x valB=                x
clk=0 icode=0111 ifun=0000 rA=0000 rB=0010 valA=                x valB=                x
clk=1 icode=0010 ifun=0000 rA=0010 rB=0110 valA=                2 valB=                0
clk=0 icode=0010 ifun=0000 rA=0010 rB=0110 valA=                2 valB=                0
clk=1 icode=0010 ifun=0000 rA=0011 rB=0111 valA=                2 valB=                0
clk=0 icode=0010 ifun=0000 rA=0011 rB=0111 valA=                2 valB=                0
clk=1 icode=0110 ifun=0001 rA=0011 rB=0110 valA=                3 valB=                6
clk=0 icode=0110 ifun=0001 rA=0011 rB=0110 valA=                3 valB=                6
```

Execute:

The arithmetic/logic unit (ALU) performs the operation provided by the instruction (based on the value of ifun), computes the effective address of a memory reference, or increments or decrements the stack pointer during the execute stage. The resulting value is referred to as valE. It's possible that the condition codes have been established. The stage will analyse the condition codes and move condition (provided by ifun) for a conditional move instruction and only update the destination register if the condition holds. It also determines whether or not the branch should be taken in the case of a jump command.

Source Code:

```
`include "./ALU/alu.v"

module execute (
    input clk,
    input [3:0] icode,
    input [3:0] ifun,
    output reg [3:0] rB,
    input [63:0] valA,
    input [63:0] valB,
    input [63:0] valC,

    output reg [63:0] valE,
    output reg condition_bit,
    output reg zf,
    output reg sf,
    output reg of
);

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

reg signed [1:0] control;
reg signed [63:0] a;
reg signed [63:0] b;
wire signed [63:0] ans;
```

```
reg signed [63:0] ans_final;
wire overflow;

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

wrapper ALU (
    .control(control),
    .a(a),
    .b(b),
    .Out(ans),
    .overflow_bit(overflow)
);

always @(*)
begin
    if (clk==1)
    begin
        //OPq
        if (icode == 4'b0110)
        begin
            if (ifun == 4'b0000) begin //Add
                control = 2'b00;
                a = valA;
                b = valB;
            end

            else if (ifun == 4'b0001) begin //Subtract
                control = 2'b01;
                a = valA;
                b = valB;
            end

            else if (ifun == 4'b0010) begin //And
                control = 2'b10;
```

```
            a = valA;
            b = valB;
        end

        else if (ifun == 4'b0011) begin //Xor
            control = 2'b11;
            a = valA;
            b = valB;
        end

        assign ans_final = ans;
        assign valE = ans_final;

        if (ans == 64'b0)
        begin
            zf = 1;
        end
        else
        begin
            zf = 0;
        end

        if (ans < 64'b0)
        begin
            sf = 1;
        end
        else
        begin
            sf = 0;
        end

        if ((a<64'b0==b<64'b0)&&(ans<64'b0!=a<64'b0))
        begin
            of = 1;
        end
        else
        begin
            of = 0;
```

```
            end
        end

        //CMOVXX
        else if (icode == 4'b0010)
        begin
            condition_bit = 0;
            if (ifun == 4'b0000) //rrmovq
            begin
                condition_bit = 1;
            end

            else if (ifun == 4'b0001) //cmovle
            begin
                if (sf^of|zf)
                begin
                    condition_bit = 1;
                end
            end

            else if (ifun == 4'b0010) //cmovl
            begin
                if (sf^of)
                begin
                    condition_bit = 1;
                end
            end

            else if (ifun == 4'b0011) //cmove
            begin
                if (zf)
                begin
                    condition_bit = 1;
                end
            end

            else if (ifun == 4'b0100) //cmovne
            begin
```

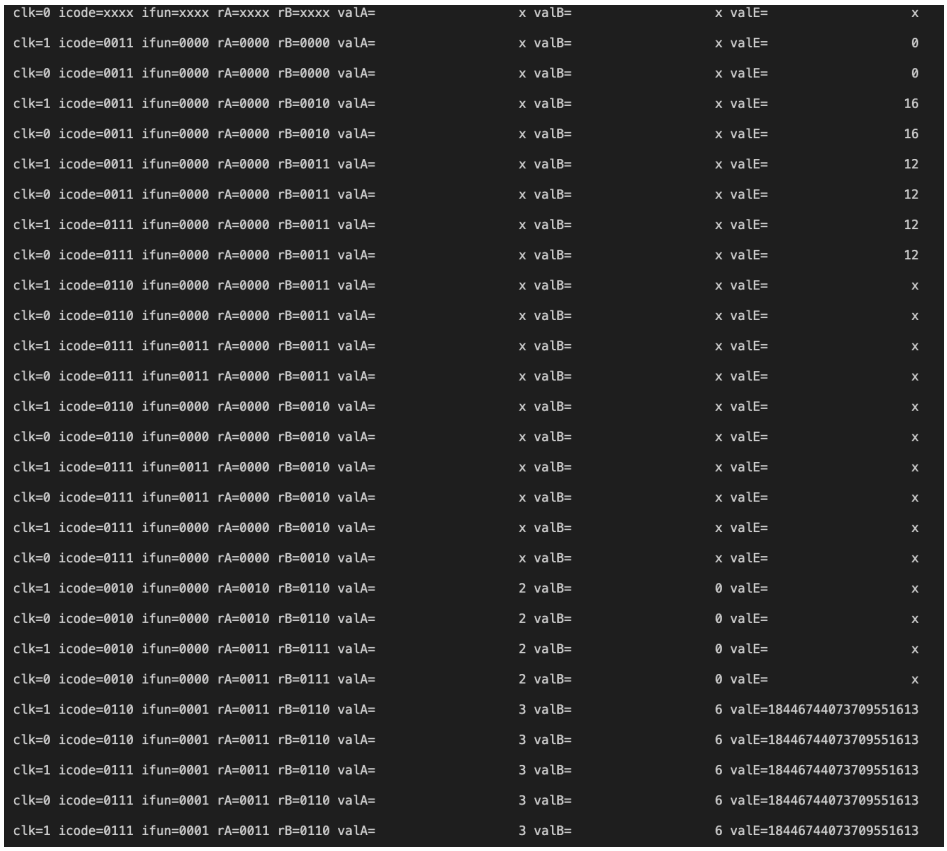
```
                if (~zf)
                begin
                    condition_bit = 1;
                end
            end

            else if (ifun == 4'b0101) //cmovge
            begin
                if (~(sf^of))
                begin
                    condition_bit = 1;
                end
            end

            else if (ifun == 4'b0110) //cmovg
            begin
                if (~(sf^of)&(~zf))
                begin
                    condition_bit = 1;
                end
            end
            valE = 64'b0 + valA;

            if (condition_bit == 1'b0)
            begin
                rB = 4'b1111;
            end
        end

        else if (icode == 4'b0011) //irmovq
        begin
            valE = 64'b0 + valC;
        end

        else if ((icode == 4'b0100) || (icode == 4'b0101)) //rmmovq or
mrmovq
        begin
            valE = valB + valC;
```

```
        end

        else if (icode == 4'b1000) //call
        begin
            valE = valB - 64'd8;
        end

        else if (icode == 4'b1001) //ret
        begin
            valE = valB + 64'd8;
        end

        else if (icode == 4'b1010) //pushq
        begin
            valE = valB - 64'd8;
        end

        else if (icode == 4'b1011) //popq
        begin
            valE = valB + 64'd8;
        end

        //jxx
        else if (icode == 4'b0111)
        begin
            if (ifun == 4'b0000) //jmp
            begin
                condition_bit = 1;
            end

            else if (ifun == 4'b0001) //jle
            begin
                if ((sf^of)|zf)
                begin
                    condition_bit = 1;
                end
            end
```

```
            else if (ifun == 4'b0010) //jl
            begin
                if (sf^of)
                begin
                    condition_bit = 1;
                end
            end

            else if (ifun == 4'b0011) //je
            begin
                if (zf)
                begin
                    condition_bit = 1;
                end
            end

            else if (ifun == 4'b0100) //jne
            begin
                if (~zf)
                begin
                    condition_bit = 1;
                end
            end

            else if (ifun == 4'b0101) //jge
            begin
                if (~(sf^of))
                begin
                    condition_bit = 1;
                end
            end

            else if (ifun == 4'b0110) //jg
            begin
                if (~(sf^of)&zf)
                begin
                    condition_bit = 1;
                end
```

```
            end
        end

    end
end
endmodule
```

Output:



Memory:

The memory stage can either write data to or read data from memory. The value read is referred to as valM. When a memory command is executed, the data memory reads or writes a word of memory. Both the instruction and data memories have access to the identical memory locations, but they are used for distinct purposes.

Source Code:

```
module memory (
    input [3:0] icode,
    input [63:0] valA,
    input [63:0] valP,
    input [63:0] valE,

    output reg [63:0] valM,
    output reg [63:0] mem_array[0:512]
);

reg [63:0] dummy_mem_array[0:512];
reg [63:0] dummy valM;
always @(*)
begin
    if (icode == 4'b0100) //rmmovq
    begin
        dummy_mem_array[valE] = valA;
        assign mem_array[valE] = dummy_mem_array[valE];
    end

    else if (icode == 4'b0101) //mrmovq
    begin
        dummy_valM = dummy_mem_array[valE];
        assign mem_array[valE] = dummy_mem_array[valE];
        assign valM = dummy_valM;
    end

    else if (icode == 4'b1010) //pushq
    begin
```

```
        dummy_mem_array[valE] = valA;
        assign mem_array[valE] = dummy_mem_array[valE];
    end

    else if (icode == 4'b1011) //popq
    begin
        dummy_valM = dummy_mem_array[valA];
        assign valM = dummy_valM;
        assign mem_array[valA] = dummy_mem_array[valA];
    end

    else if (icode = 4'b1000) //call
    begin
        dummy_mem_array[valE] = valP;
        assign mem_array[valE] = dummy_mem_array[valE];
    end

    else if (icode == 4'b1001) //ret
    begin
        dummy_valM = dummy_mem_array[valA];
        assign valM = dummy_valM;
        assign mem_array[valA] = dummy_mem_array[valA];
    end
    assign
end
endmodule
```

Write Back:

The write-back stage writes up to two results to the register file. In this stage, valE and valM are written into the register file (rA, rB, or rsp) based on the value of icode.

Description:

Source Code:

```
module write_back (
    input clk,
    input [3:0] icode,
    input [3:0] rA,
    input [3:0] rB,
    input [63:0] valM,
    input [63:0] valE,

    output reg [63:0] reg_arr[0:14]
);

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

always @(*)
begin
    if ((icode == 4'b0010) || (icode == 4'b0011) || (icode == 4'b0110))
//cmovxx | irmovq | OPq
    begin
        dummy_reg_arr[rB] = valE;
        assign reg_arr[rB] = dummy_reg_arr[rB];
    end

    else if ((icode == 4'b0101)) //mrmovq
    begin
        dummy_reg_arr[rA] = valM;
        assign reg_arr[rA] = dummy_reg_arr[rA];
    end

    else if ((icode == 4'b1000) || (icode == 4'b1001)) //call
    begin
```

```
        dummy_reg_arr[4] = valE; //the 5th register (index 4) is the
stack pointer %rsp
        assign reg_arr[4] = dummy_reg_arr[4];
    end

    else if (icode == 4'b1010) //pushq
    begin
        dummy_reg_arr[4] = valE;
        assign reg_arr[4] = dummy_reg_arr[4];
    end

    else if (icode == 4'b1011) //popq
    begin
        dummy_reg_arr[4] = valE;
        assign reg_arr[4] = dummy_reg_arr[4];
        dummy_reg_arr[rA] = valM;
        assign reg_arr[rA] = dummy_reg_arr[rA];
    end
end
endmodule
```

Pc Update:

The PC is now set to the next instruction's address.The new value of the programme counter is either valP, which is the address of the next instruction, valC, which is the address indicated by a call or jump instruction, or valM, which is the address read from memory.

Source Code:

```
module PC_update (
    input clk,
    input condition_bit,
    input [3:0] icode,
    input [63:0] valC,
    input [63:0] valP,
    input [63:0] valM,
    input [63:0] PC,

    output reg [63:0] final_PC
);

reg [63:0] dummy_PC;
always @(*)
begin
    if ((icode == 4'b0110) || (icode == 4'b0011) || (icode == 4'b0100)
|| (icode == 4'b0101) || (icode == 4'b1010) || (icode == 4'b1011))
//OPq | irmovq | rmmovq | mrmovq | pushq | popq
    begin
        dummy_PC = valP;
    end

    else if (icode == 4'b1000) //call
    begin
        dummy_PC = valC;
    end

    else if (icode == 4'b1001) //ret
    begin
        dummy_PC = valM;
    end

    else if (icode == 4'b0111) //Jxx
```

```
    begin
        if (condition_bit == 1'b1)
        begin
            dummy_PC = valC;
        end
        else
        begin
            dummy_PC = valP;
        end
    end

    else if (icode == 4'b0010) //cmovxx
    begin
        dummy_PC = valP;
    end
    assign final_PC = dummy_PC;
end

endmodule
```