IPA Mid-Evals

Date: 23/2/2022

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Sequential y86 Implementation

Team: 211

Aditya Sehgal 2020112013

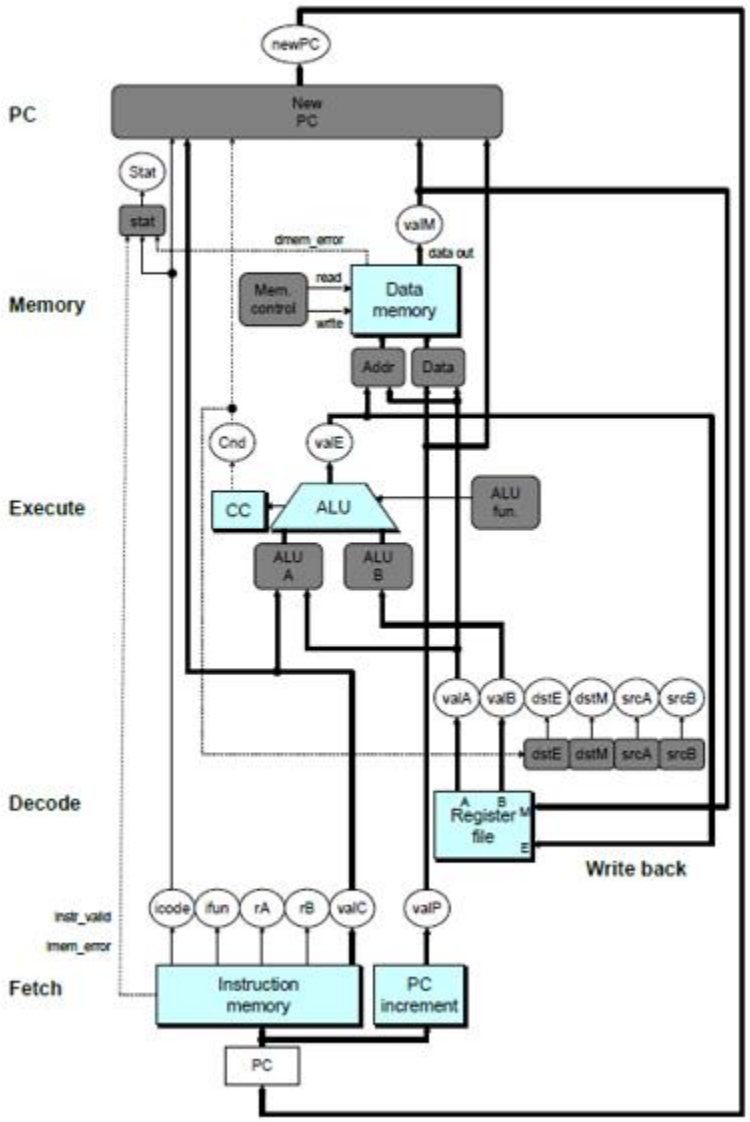
Rishabh Agrawal 2020102038

Index

Overview

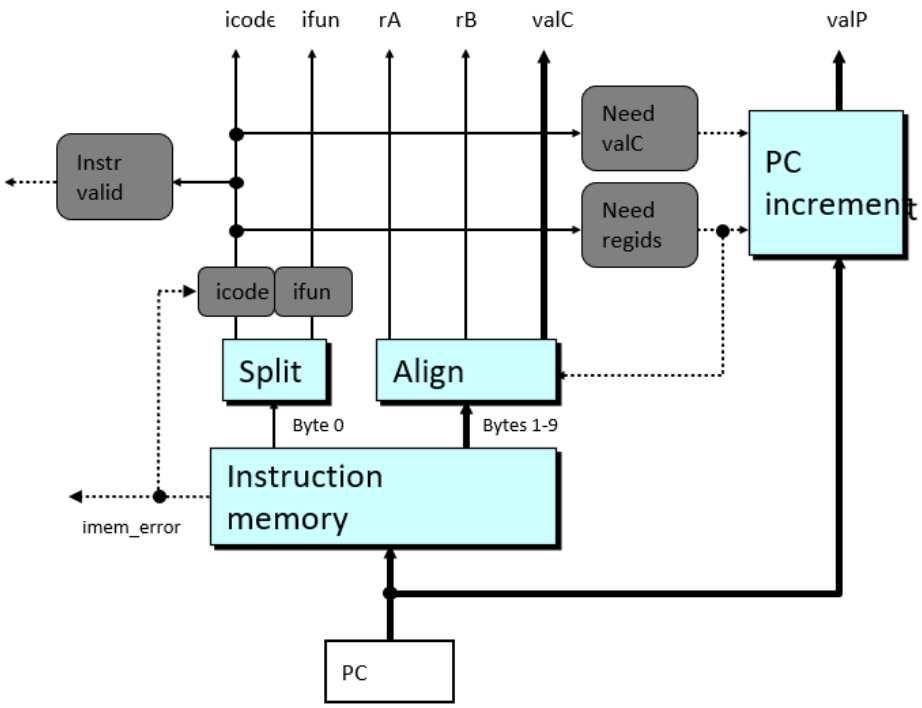
The required specifications in the processor design are as follows:

A bare minimum processor architecture must implement a sequential design.



Modules

Fetch



We have implemented it by taking input/output as:

```
input clk;
input [64:1] pc;
output reg [4:1] icode;
output reg [4:1] ifun;
output reg [4:1] rA;
output reg [4:1] rB;
output reg [64:1] valC;
output reg [64:1] valP;
reg [8:1]instr_mem[100:1];
```

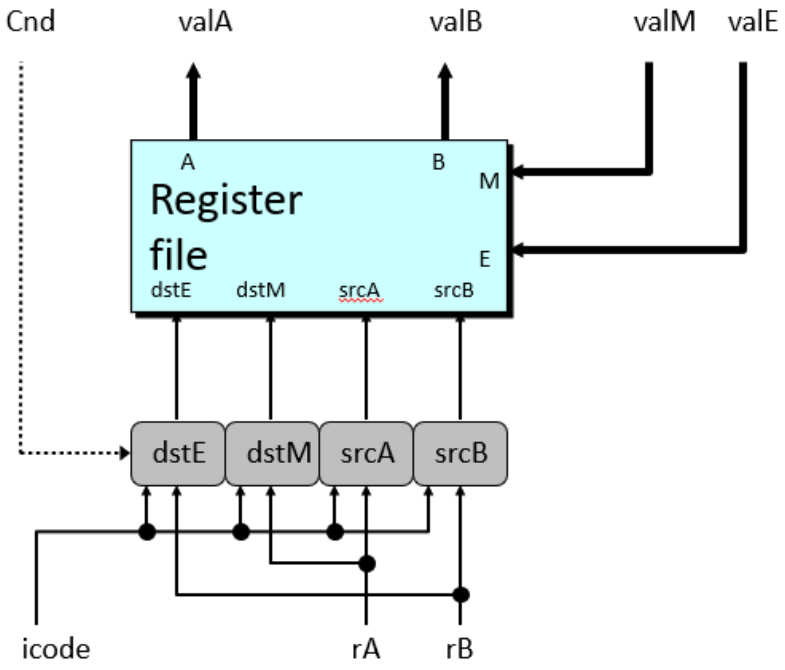
Fetch conditions are implemented using muxes with if statements comparing values of icode for respective functions as:

```
if (icode == 4'b0000) //halt
    begin
```

```
        valP = pc + 64'd1;
    end
```

Since it’s a combinational circuit, always block (values updating) is independent of the rising or falling of clock.

Decode



We have implemented it by taking input/output as:

```
input clk,
    input cnd,
    input [4:1]icode,
    input [4:1] rA,
    input [4:1] rB,
    input [64:1] valE,
    input [64:1] valM,

    output reg [64:1] valA,
    output reg [64:1] valB,
    output reg [64:1] register_mem1,
    output reg [64:1] register_mem2,
    output reg [64:1] register_mem3,
    output reg [64:1] register_mem4,
    output reg [64:1] register_mem5,
    output reg [64:1] register_mem6,
    output reg [64:1] register_mem7,
    output reg [64:1] register_mem8,
    output reg [64:1] register_mem9,
    output reg [64:1] register_mem10,
```

```
    output reg [64:1] register_mem11,
    output reg [64:1] register_mem12,
    output reg [64:1] register_mem13,
    output reg [64:1] register_mem14,
    output reg [64:1] register_mem15
```

Here, all the registers are sent independently as we can't send an 2-D array through a function call in Verilog.

Decode Block:

Decode conditions are implemented using muxes with if statements comparing values of icode for respective functions as:

```
if(icode == 4'b0010) //cmovxx
    begin
        valA = register_mem[rA];
    end
```

Since it's a combinational circuit (only reading of values is occurring), always block (values updating) is independent of the rising or falling of clock.

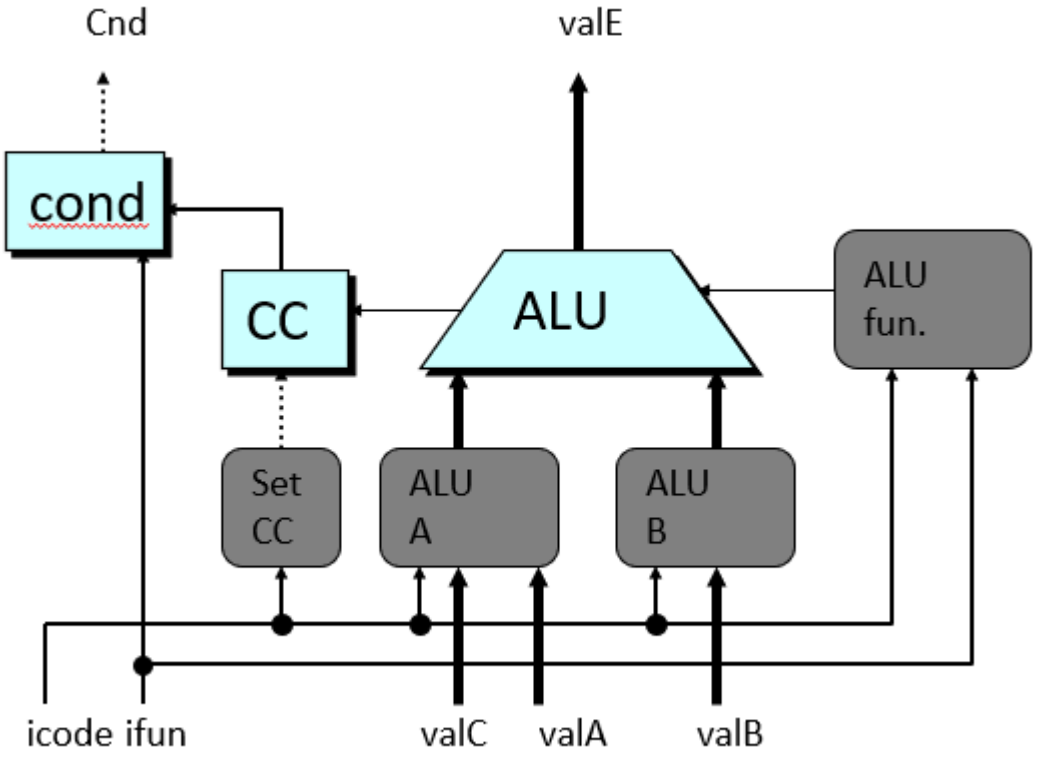
Write Back Block:

Write Back is also implemented using mux but since it's a sequential block, it gets updated on very positive edge of the clock:

```
always @(posedge clk)
begin
    if(icode == 4'b0010) //cmovxx
    begin
        if(cnd == 1'b1)
        begin
            register_mem[rB] = valE;
        end
        else
        begin
            register_mem[15] = valE;
        end
    end
```

Since both depends on register memory, they are written in a single file.

Execute



We have implemented it by taking input/output as:

```
input clk;
input  [4:1]icode;
input  [4:1]ifun;
input  [64:1]valA;
input [64:1]valB;
input [64:1]valC;
output reg [64:1] valE;
output reg  zf, sf, of;
output reg cnd;

reg [2:1]sel;
reg [64:1]a;
reg [64:1]b;
wire [64:1]result;
wire overflow;
reg cin;

alu dut(.sel(sel),.a(a),.b(b),.cin(cin),.result(result),.overflow(overflow));
```

Here, ALU is been called for all Operations carried out through any function.

Setting Condition Code:

Whenever the icode is equal to Operation function, condition codes zf, sf and of are generated. Since they are sequential blocks, they get updated on positive edge of clock:

```
always @(posedge clk)
begin
    if(icode == 4'b0110 || clk==1)  //OP
    begin
        if(result == 64'b0)
            begin
                zf = 1;
            end
        else
            begin
                zf = 0;
            end
        if(result[64] == 1 )
            begin
                sf = 1;
            end
        else
            begin
                sf = 0;
            end
        if((a<64'b0 == b<64'b0) && (result<64'b0 != a<64'b0))
            begin
                of = 1;
            end
        else
            begin
                of = 0;
            end
    end
end
```

Setting values and Cnd for other Functions:

Values and Cnd (for cmovxx and jxx) are set for all the functions as:

```
if(icode == 4'b0110) //OP
    begin
        b = valA;
        a = valB;
      if(ifun == 4'b0000) //ADD
        begin
            sel = 2'b00;
        end
      if(ifun == 4'b0001) //SUB
        begin
```

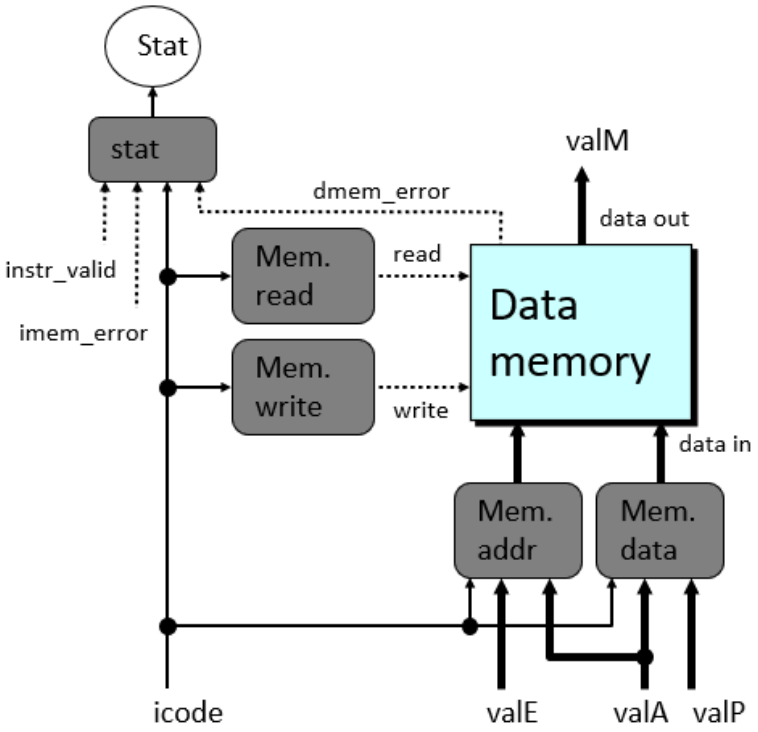
```
            sel = 2'b01;
        end
      if(ifun == 4'b0100) //AND
        begin
            sel = 2'b10;
        end
      if(ifun == 4'b0011) //XOR
      begin
            sel = 2'b11;
      end
      valE = result; //set_CC

   end

   // var assign problem
   else if(icode == 4'b0010) //cmovXX
   begin
       if(ifun == 4'b0000) //rrmovq
       begin
          cnd=1'b1;
       end

       if(ifun == 4'b0001) //cmovle
       begin
           if((sf^of)|zf)
           begin
               cnd=1'b1;
           end
       end

       if(ifun == 4'b0010) //cmovl
       begin
           if(sf^of)
           begin
               cnd=1'b1;
           end
       end

       if(ifun == 4'b0011) //cmove
       begin
           if(zf)
           begin
               cnd=1'b1;
           end
       end

       if(ifun == 4'b0100) //cmone
       begin
           if(~zf)
           begin
```

```
                cnd=1'b1;
            end
        end

        if(ifun == 4'b0101) //cmovge
        begin
            if(~(sf^of))
            begin
                cnd=1'b1;
            end
        end

        if(ifun == 4'b0110) //cmovg
        begin
            if((~(sf^of)&(~zf)))  //note here
            begin
                cnd=1'b1;
            end
        end

        a = valA;
        b = 0;
        sel = 2'b00;   //add
        valE = result;

    end

    else if (icode == 4'b0111) //jxx
    begin
        if(ifun == 4'b0000) //jmp
        begin
            cnd=1'b1;
        end

        if(ifun == 4'b0001) //jle
        begin
            if((sf^of)|zf)
            begin
                cnd=1'b1;
            end
        end

        if(ifun == 4'b0010) //jl
        begin
            if(sf^of)
            begin
                cnd=1'b1;
            end
        end
        if(ifun == 4'b0011) //je
```

```
      begin
        if(zf)
        begin
            cnd=1'b1;
        end
    end
    if(ifun == 4'b0100) //jne
    begin
        if(~zf)
        begin
            cnd=1'b1;
        end
    end
    if(ifun == 4'b0101) //jge
    begin
        if(~(sf^of))
        begin
            cnd=1'b1;
        end
    end
    if(ifun == 4'b0110) //jg
    begin
        if((~(sf^of)&(~zf)))
        begin
            cnd=1'b1;
        end
    end

end

else if(icode == 4'b0100) //rmmovq
begin
    a = valB;
    b = valC;
    sel = 2'b00;
    valE = result;
end

else if(icode == 4'b0101) //mrmovq
begin
    a = valB;
    b = valC;
    sel = 2'b00;
    valE = result;
end

else if(icode == 4'b0011) //irmovq
begin
    a = valC;
    b = 64'b0;
```

```
        sel = 2'b00;
        valE = result;
    end

    else if(icode == 4'b1011 || icode == 4'b1001) //popq || ret
    begin
        a = valB;
        b = 64'b1000;
        sel = 2'b00;
        valE = result;
    end

    else if(icode == 4'b1010 || icode == 4'b1000) //pushq || call
    begin
        a = valB;
        b = 64'b1000;
        sel = 2'b01;
        valE = result;
    end
```

Memory



We have implemented it by taking input/output as:

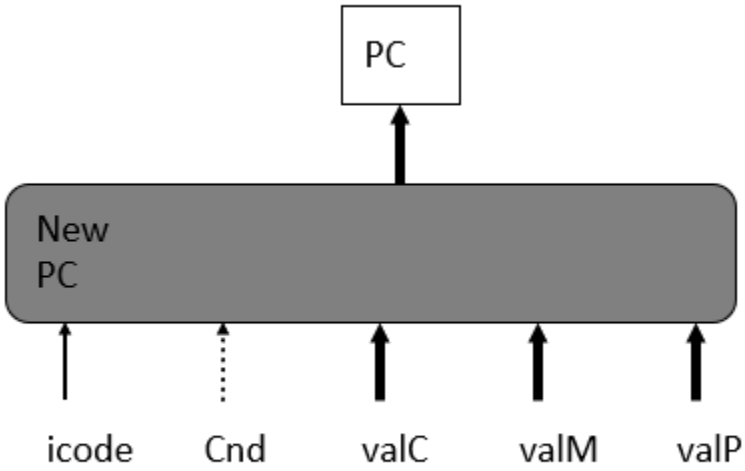
```
input clk;
input [4:1]icode;
input [64:1] valA;
input [64:1] valB;
```

```
input [64:1] valC;
input [64:1] valE;
input [64:1] valP;

output reg [64:1] valM;
output reg [64:1]dram_val;


reg [64:1] ram[1:1024];
```

Values are either taken or written back to memory or register using either valE or valM depending on the location. For eg:

```
if(icode == 4'b0100) //rmmovq
    begin
        ram[valE] = valA;
        dram_val = ram[valE];
    end
    else if(icode == 4'b0101) //mrmovq
    begin
        valM = ram[valE];
        dram_val = valM;
    end
```

PC Update



We have implemented it by taking input/output as:

```
    input clk,
    input [4:1] icode,
```

```
    input cnd,
    input [64:1] valC,
    input [64:1] valM,
    input [64:1] valP,
    output reg [64:1] new_pc
```

Only jxx, ret and call have specifc assignment for pc update. Other functions follow a similar pc incrementation of valP:

```
always @(*)
begin


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

GTKWave Results:

Code 1:

```
//irmove
    instr_mem[32]=8'b00110000;
    instr_mem[33]=8'b11110100; //%7 to 4
    instr_mem[34]=8'b00000111;
    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'b00000000;
    instr_mem[43]=8'b00000000;

// rmmove
    instr_mem[44]=8'b01000000;
    instr_mem[45]=8'b01000011;
    instr_mem[46]=8'b00000011;
    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'b00000000;
    instr_mem[53]=8'b00000000;

    instr_mem[54]=8'b00000000;
    instr_mem[55]=8'b00000000;

//mrmovq
    instr_mem[56]=8'b01010000;
    instr_mem[57]=8'b01110011;
```

```
    instr_mem[58]=8'b00000011;
    instr_mem[59]=8'b00000000;
    instr_mem[60]=8'b00000000;
    instr_mem[61]=8'b00000000;
    instr_mem[62]=8'b00000000;
    instr_mem[63]=8'b00000000;
    instr_mem[64]=8'b00000000;
    instr_mem[65]=8'b00000000;

//subq
    instr_mem[66]=8'b01100000;
    instr_mem[67]=8'b01010011;
    instr_mem[68]=8'b00000000;
```

GTKWave:

Code 2:

```
//irmove
    instr_mem[32]=8'b00110000;
    instr_mem[33]=8'b11110100;
    instr_mem[34]=8'b00000111;
    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'b00000000;
    instr_mem[43]=8'b00000000;

//irmove
    instr_mem[44]=8'b00110000;
    instr_mem[45]=8'b11110110;
    instr_mem[46]=8'b00000111;
    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'b00000000;
    instr_mem[53]=8'b00000000;

    instr_mem[54]=8'b00000000;
    instr_mem[55]=8'b00000000;

//subq
    instr_mem[56]=8'b01100001;
    instr_mem[57]=8'b01000110;
```

```
    instr_mem[58]=8'b00000000;
//cmovl
    instr_mem[59]=8'b00100011;
    instr_mem[60]=8'b01000111;
    instr_mem[61]=8'b00000000;
```

GTKWave: