IPA Project

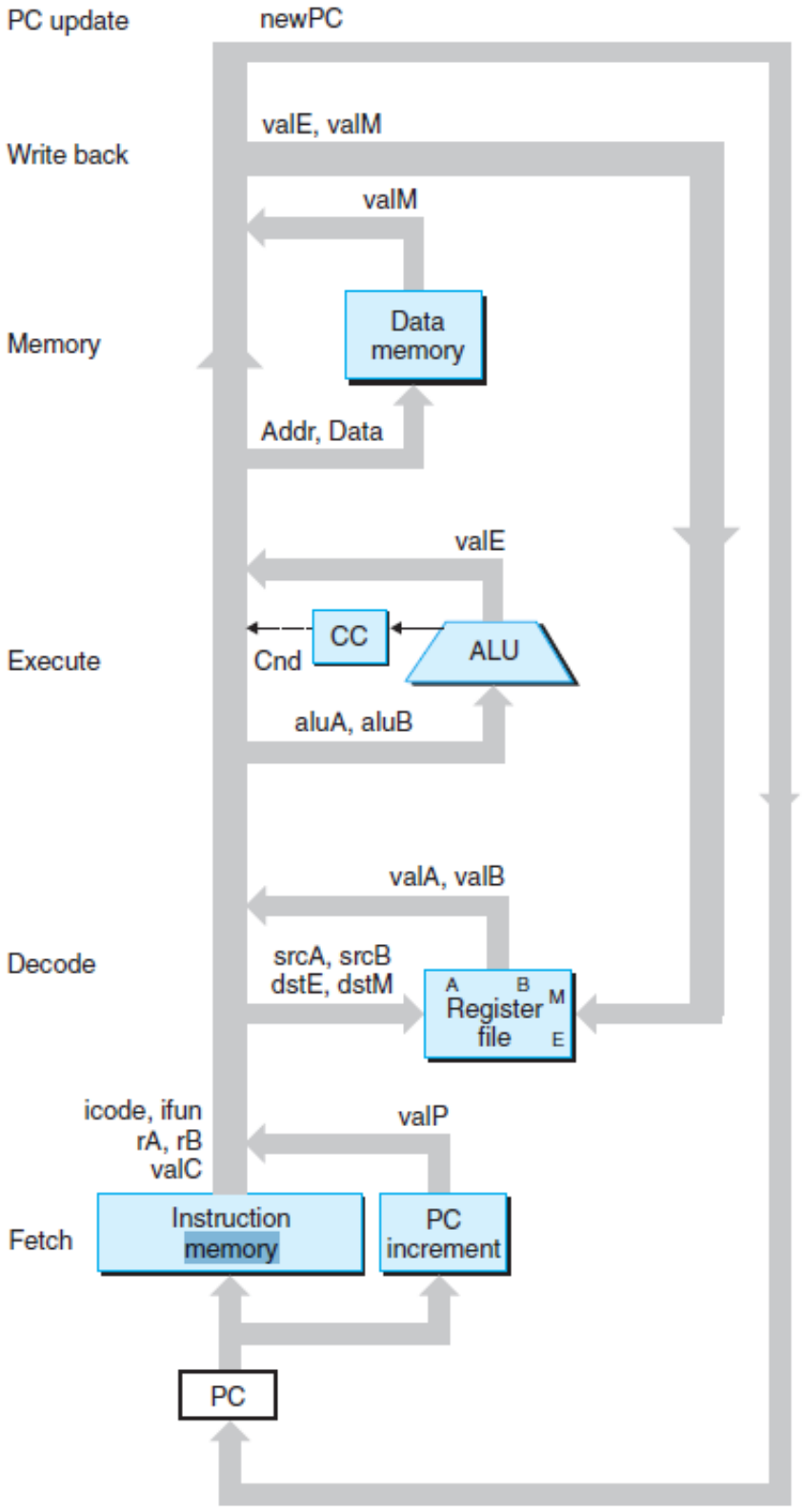
Mid-evals Project Report

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. The pipelined design of the processor is made so that multiple commands can be taken consecutively with a much smaller delay compared to the sequential design.

Sequential:


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

```
1   0x000: 30f20900000000000000 |     irmovq $9,  %rdx
2   0x00a: 30f31500000000000000 |     irmovq $21, %rbx
3   0x014: 6123                 |     subq %rdx, %rbx          # subtract
4   0x016: 30f48000000000000000 |     irmovq $128,%rsp         # Problem 4.13
5   0x020: 4043640000000000000  |     rmmovq %rsp, 100(%rbx)   # store
6   0x02a: a02f                 |     pushq %rdx               # push
7   0x02c: b00f                 |     popq  %rax               # Problem 4.14
8   0x02e: 734000000000000000   |     je done                  # Not taken
9   0x037: 804100000000000000   |     call proc                # Problem 4.18
10  0x040:                      | done:
11  0x040: 00                   |     halt
12  0x041:                      | proc:
13  0x041: 90                   |     ret                      # Return
14                              |
```

Figure 1: Example of Y86-64 machine code

Stage	OPq rA, rB	rrmovq rA, rB	irmovq V, rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_8[PC + 2] valP ← PC + 10
Decode	valA ← R[rA] valB ← R[rB]	valA ← R[rA]	
Execute	valE ← valB OP valA Set CC	valE ← 0 + valA	valE ← 0 + valC
Memory			
Write back	R[rB] ← valE	R[rB] ← valE	R[rB] ← valE
PC update	PC ← valP	PC ← valP	PC ← valP

Figure 2:

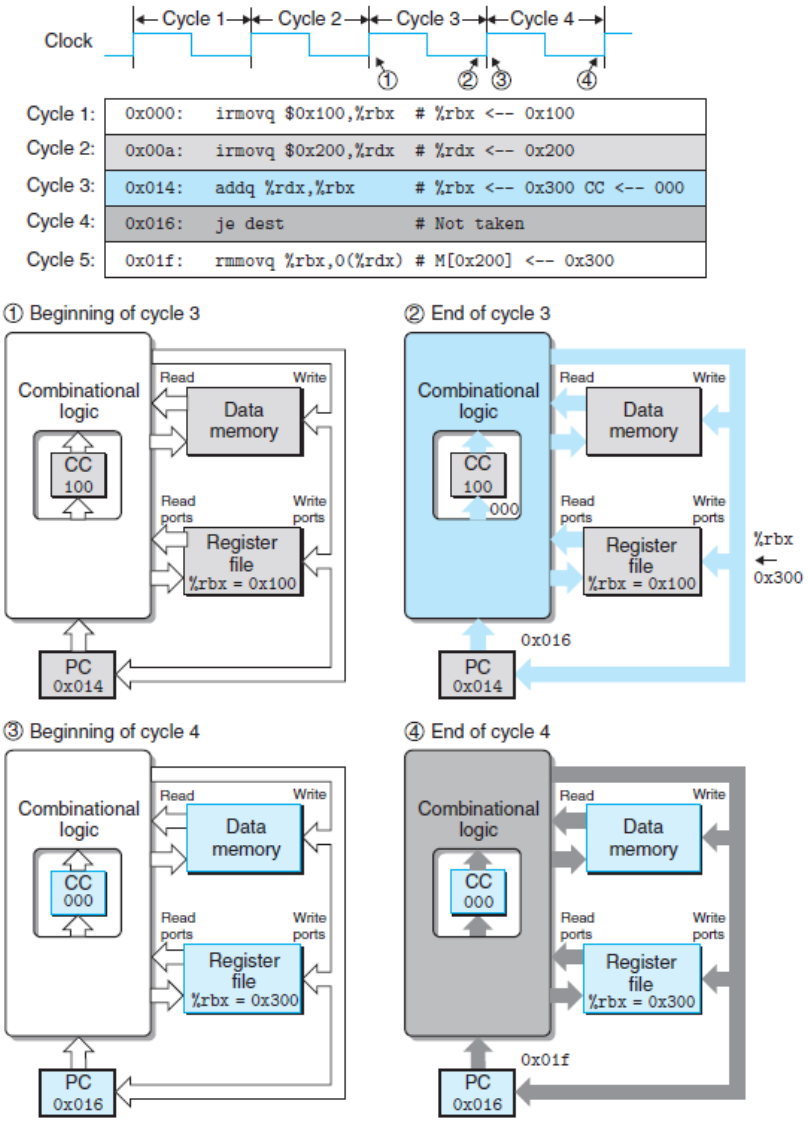

Clock
Cycle 1
Cycle 2
Cycle 3
Cycle 4
①
②
③
④
Cycle 1: 0x000: irmovq $0x100,%rbx # %rbx <-- 0x100
Cycle 2: 0x00a: irmovq $0x200,%rdx # %rdx <-- 0x200
Cycle 3: 0x014: addq %rdx,%rbx # %rbx <-- 0x300 CC <-- 000
Cycle 4: 0x016: je dest # Not taken
Cycle 5: 0x01f: rmmovq %rbx,0(%rdx) # M[0x200] <-- 0x300
① Beginning of cycle 3
② End of cycle 3
③ Beginning of cycle 4
④ End of cycle 4
Combinational logic
CC
100
000
Read
Write
Data memory
Read ports
Write ports
Register file
%rbx = 0x100
%rbx = 0x300
%rbx
0x300
PC
0x014
0x016
0x01f

Figure 3:

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'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'b00100111;
    instr_mem[33]=8'b00001100;
    instr_mem[34]=8'b00100111;
    instr_mem[35]=8'b00000011;
    instr_mem[36]=8'b01110011;
    instr_mem[37]=8'b00100111;
```

```
    instr_mem[38]=8'b00100111;
    instr_mem[39]=8'b01100000;
    instr_mem[40]=8'b00000011;
    instr_mem[41]=8'b01110011;
    instr_mem[42]=8'b01110011;
    instr_mem[43]=8'b00000011;
    instr_mem[44]=8'b00110000;
    instr_mem[45]=8'b01110011;
    instr_mem[46]=8'b01110000;
    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[7:0]=instr[16:23];
      valC[15:8]=instr[24:31];
      valC[23:16]=instr[32:39];
      valC[31:24]=instr[40:47];
      valC[39:32]=instr[48:55];
      valC[47:40]=instr[56:63];
      valC[55:48]=instr[64:71];
      valC[63:56]=instr[72:79];
      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[7:0]=instr[16:23];
      valC[15:8]=instr[24:31];
      valC[23:16]=instr[32:39];
      valC[31:24]=instr[40:47];
      valC[39:32]=instr[48:55];
      valC[47:40]=instr[56:63];
      valC[55:48]=instr[64:71];
      valC[63:56]=instr[72:79];
      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[7:0]=instr[8:15];
      valC[15:8]=instr[16:23];
      valC[23:16]=instr[24:31];
      valC[31:24]=instr[32:39];
      valC[39:32]=instr[40:47];
      valC[47:40]=instr[48:55];
      valC[55:48]=instr[56:63];
      valC[63:56]=instr[64:71];
      valP=PC+64'd9;
    end
    4'd8: //call
    begin
      valC[7:0]=instr[8:15];
```

```
      valC[15:8]=instr[16:23];
      valC[23:16]=instr[24:31];
      valC[31:24]=instr[32:39];
      valC[39:32]=instr[40:47];
      valC[47:40]=instr[48:55];
      valC[55:48]=instr[56:63];
      valC[63:56]=instr[64:71];
      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=1 PC=                32 icode=0010 ifun=0111 rA=0000 rB=1100,valC=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,valP=
 34

clk=0 PC=                32 icode=0010 ifun=0111 rA=0000 rB=1100,valC=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,valP=
 34

clk=1 PC=                34 icode=0010 ifun=0111 rA=0000 rB=0011,valC=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,valP=
 36

clk=0 PC=                34 icode=0010 ifun=0111 rA=0000 rB=0011,valC=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,valP=
 36

clk=1 PC=                36 icode=0111 ifun=0011 rA=0000 rB=0011,valC=00100111001001110110000000000110111001101110011000000110011000,valP=
 45

clk=0 PC=                36 icode=0111 ifun=0011 rA=0000 rB=0011,valC=00100111001001110110000000000110111001101110011000000110011000,valP=
 45
```

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,

    output reg [63:0] valA,
    output reg [63:0] valB,

    output reg [63:0] valE,
    output reg [63:0] valM,

    output reg [63:0] reg_arr0,
    output reg [63:0] reg_arr1,
    output reg [63:0] reg_arr2,
    output reg [63:0] reg_arr3,
    output reg [63:0] reg_arr4,
    output reg [63:0] reg_arr5,
    output reg [63:0] reg_arr6,
    output reg [63:0] reg_arr7,
    output reg [63:0] reg_arr8,
    output reg [63:0] reg_arr9,
    output reg [63:0] reg_arr10,
    output reg [63:0] reg_arr11,
    output reg [63:0] reg_arr12,
    output reg [63:0] reg_arr13,
    output reg [63:0] reg_arr14
);
```

```
reg [63:0] dummy_reg_arr[0:14];
integer i;

initial begin
for (i = 0; i<15; i=i+1) begin
    dummy_reg_arr[i] = 64'd0;
end
end

always @(*)
begin
    case (icode)
        4'b0000: begin //halt
        end

        4'b0001: begin //nop
        end

        4'b0110: begin //OPq
            valA = dummy_reg_arr[rA];
            valB = dummy_reg_arr[rB];
        end

        4'b0100: begin //rmmovq
            valA = dummy_reg_arr[rA];
            valB = dummy_reg_arr[rB];
        end

        4'b1011: begin //popq
            valA = dummy_reg_arr[4];
            valB = dummy_reg_arr[4];
        end

        4'b0010: begin //cmovxx
            valA = dummy_reg_arr[rA];
            valB = 64'b0;
        end
```

```
        4'b0111: begin //jxx
        end

        4'b1000: begin // call
            valB = dummy_reg_arr[4];
        end

        4'b1010: begin //pushq
            valA = dummy_reg_arr[rA];
            valB = dummy_reg_arr[4];
        end

        4'b0011: begin //irmovq
        end

        4'b1001: begin //ret
            valA = dummy_reg_arr[4];
            valB = dummy_reg_arr[4];
        end

        4'b0101: begin //mrmovq
            valB = dummy_reg_arr[rB];
        end
    endcase
    reg_arr0 = dummy_reg_arr[0];
    reg_arr1 = dummy_reg_arr[1];
    reg_arr2 = dummy_reg_arr[2];
    reg_arr3 = dummy_reg_arr[3];
    reg_arr4 = dummy_reg_arr[4];
    reg_arr5 = dummy_reg_arr[5];
    reg_arr6 = dummy_reg_arr[6];
    reg_arr7 = dummy_reg_arr[7];
    reg_arr8 = dummy_reg_arr[8];
    reg_arr9 = dummy_reg_arr[9];
    reg_arr10 = dummy_reg_arr[10];
    reg_arr11 = dummy_reg_arr[11];
    reg_arr12 = dummy_reg_arr[12];
```

```
    reg_arr13 = dummy_reg_arr[13];
    reg_arr14 = dummy_reg_arr[14];
end

// Write back
always @(negedge clk)
begin
    if ((icode == 4'b0010) || (icode == 4'b0011) || (icode == 4'b0110))
//cmovxx | irmovq | OPq
    begin
        dummy_reg_arr[rB] = valE;
    end

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

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

    else if (icode == 4'b1011) //popq
    begin
        dummy_reg_arr[4] = valE;
        dummy_reg_arr[rA] = valM;
    end
    reg_arr0 = dummy_reg_arr[0];
    reg_arr1 = dummy_reg_arr[1];
    reg_arr2 = dummy_reg_arr[2];
    reg_arr3 = dummy_reg_arr[3];
```

```
    reg_arr4 = dummy_reg_arr[4];
    reg_arr5 = dummy_reg_arr[5];
    reg_arr6 = dummy_reg_arr[6];
    reg_arr7 = dummy_reg_arr[7];
    reg_arr8 = dummy_reg_arr[8];
    reg_arr9 = dummy_reg_arr[9];
    reg_arr10 = dummy_reg_arr[10];
    reg_arr11 = dummy_reg_arr[11];
    reg_arr12 = dummy_reg_arr[12];
    reg_arr13 = dummy_reg_arr[13];
    reg_arr14 = dummy_reg_arr[14];
end

endmodule
```

Output:

```
clk=1 PC=0 icode=0011 ifun=0000 rA=0000 rB=0000 valA=x valB=x valE=x valM=x
clk=0 PC=0 icode=0011 ifun=0000 rA=0000 rB=0000 valA=x valB=x valE=x valM=x
clk=1 PC=10 icode=0011 ifun=0000 rA=0000 rB=0010 valA=x valB=x valE=x valM=x
clk=0 PC=10 icode=0011 ifun=0000 rA=0000 rB=0010 valA=x valB=x valE=x valM=x
clk=1 PC=20 icode=0011 ifun=0000 rA=0000 rB=0011 valA=x valB=x valE=x valM=x
clk=0 PC=20 icode=0011 ifun=0000 rA=0000 rB=0011 valA=x valB=x valE=x valM=x
clk=1 PC=30 icode=0111 ifun=0000 rA=0000 rB=0011 valA=x valB=x valE=x valM=x
clk=0 PC=30 icode=0111 ifun=0000 rA=0000 rB=0011 valA=x valB=x valE=x valM=x
```

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
```

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
);

reg [63:0] dummy_mem_array[0:1023];
reg [63:0] dummy_valM;
always @(*)
begin
    if (icode == 4'b0100) //rmmovq
    begin
        dummy_mem_array[valE] = valA;
    end

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

    else if (icode == 4'b1010) //pushq
    begin
        dummy_mem_array[valE] = valA;
    end

    else if (icode == 4'b1011) //popq
```

```
    begin
        dummy_valM = dummy_mem_array[valA];
        valM = dummy_valM;
    end

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

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

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_arr0,
    output reg [63:0] reg_arr1,
    output reg [63:0] reg_arr2,
    output reg [63:0] reg_arr3,
    output reg [63:0] reg_arr4,
    output reg [63:0] reg_arr5,
    output reg [63:0] reg_arr6,
    output reg [63:0] reg_arr7,
    output reg [63:0] reg_arr8,
    output reg [63:0] reg_arr9,
    output reg [63:0] reg_arr10,
    output reg [63:0] reg_arr11,
    output reg [63:0] reg_arr12,
    output reg [63:0] reg_arr13,
    output reg [63:0] reg_arr14
);

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;
```

```
    end

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

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

    else if (icode == 4'b1011) //popq
    begin
        dummy_reg_arr[4] = valE;
        dummy_reg_arr[rA] = valM;
    end
    reg_arr0 = dummy_reg_arr[0];
    reg_arr1 = dummy_reg_arr[1];
    reg_arr2 = dummy_reg_arr[2];
    reg_arr3 = dummy_reg_arr[3];
    reg_arr4 = dummy_reg_arr[4];
    reg_arr5 = dummy_reg_arr[5];
    reg_arr6 = dummy_reg_arr[6];
    reg_arr7 = dummy_reg_arr[7];
    reg_arr8 = dummy_reg_arr[8];
    reg_arr9 = dummy_reg_arr[9];
    reg_arr10 = dummy_reg_arr[10];
    reg_arr11 = dummy_reg_arr[11];
    reg_arr12 = dummy_reg_arr[12];
    reg_arr13 = dummy_reg_arr[13];
    reg_arr14 = dummy_reg_arr[14];
```

```
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'b0001) begin
        dummy_PC = valP;
    end

    else if (icode == 4'b0000) begin
    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
```

Processor:

This module is the wrapper and executes all of the above modules.

Source code:

```
`include "fetch.v"
`include "execute.v"
`include "decode.v"
`include "write_back.v"
`include "memory.v"
`include "PC_update.v"

module processor;
  reg clk = 0;

  reg [63:0] PC;

  reg stat[0:2];

  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 instr_val;
  wire imem_er;
  wire [63:0] valA;
  wire [63:0] valB;
  wire [63:0] valE;
  wire [63:0] valM;
  wire cnd;
  wire hltins;
  wire [63:0] updated_pc;

  // wire [63:0] reg_mem0;
  // wire [63:0] reg_mem1;
  // wire [63:0] reg_mem2;
  // wire [63:0] reg_mem3;
```

```
  // wire [63:0] reg_mem4;
  // wire [63:0] reg_mem5;
  // wire [63:0] reg_mem6;
  // wire [63:0] reg_mem7;
  // wire [63:0] reg_mem8;
  // wire [63:0] reg_mem9;
  // wire [63:0] reg_mem10;
  // wire [63:0] reg_mem11;
  // wire [63:0] reg_mem12;
  // wire [63:0] reg_mem13;
  // wire [63:0] reg_mem14;
  wire [63:0] reg_mem[0:14];
  wire [63:0] datamem;

  wire [63:0] reg_arr0;
  wire [63:0] reg_arr1;
  wire [63:0] reg_arr2;
  wire [63:0] reg_arr3;
  wire [63:0] reg_arr4;
  wire [63:0] reg_arr5;
  wire [63:0] reg_arr6;
  wire [63:0] reg_arr7;
  wire [63:0] reg_arr8;
  wire [63:0] reg_arr9;
  wire [63:0] reg_arr10;
  wire [63:0] reg_arr11;
  wire [63:0] reg_arr12;
  wire [63:0] reg_arr13;
  wire [63:0] reg_arr14;

  wire sf;
  wire zf;
  wire of;

  fetch fetch(
    .clk(clk),
    .PC(PC),
    .icode(icode),
```

```
    .ifun(ifun),
    .rA(rA),
    .rB(rB),
    .valC(valC),
    .valP(valP),
    .instr_val(instr_valid),
    .imem_er(imem_error),
    .halt(hltins)
  );

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

  decode decode(
        .clk(clk),
        .icode(icode),
        .rA(rA),
        .rB(rB),
        .valA(valA),
        .valB(valB),
        .valE(valE),
        .valM(valM),
        .reg_arr0(reg_arr0),
        .reg_arr1(reg_arr1),
        .reg_arr2(reg_arr2),
        .reg_arr3(reg_arr3),
        .reg_arr4(reg_arr4),
```

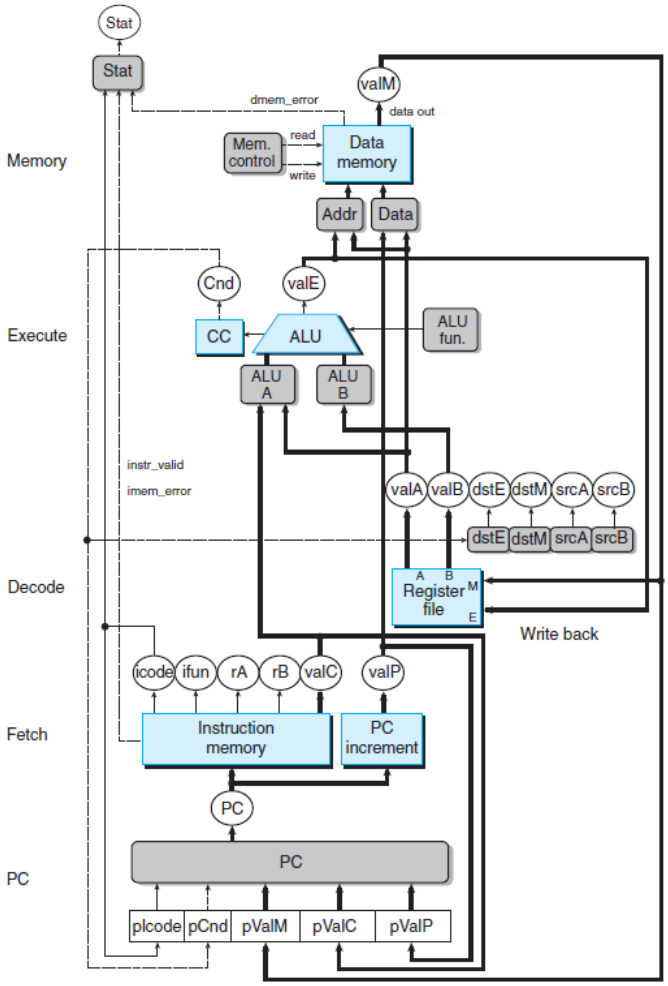
```
        .reg_arr5(reg_arr5),
        .reg_arr6(reg_arr6),
        .reg_arr7(reg_arr7),
        .reg_arr8(reg_arr8),
        .reg_arr9(reg_arr9),
        .reg_arr10(reg_arr10),
        .reg_arr11(reg_arr11),
        .reg_arr12(reg_arr12),
        .reg_arr13(reg_arr13),
        .reg_arr14(reg_arr14)
    );
  // write_back write_back(
  //       .clk(clk),
  //       .icode(icode),
  //       .rA(rA),
  //       .rB(rB),
  //       .valE(valE),
  //       .valM(valM)
  //   );

  memory memory(
    .icode(icode),
    .valA(valA),
    .valE(valE),
    .valP(valP),
    .valM(valM),
    .mem_array(datamem)
  );

  PC_update PC_update(
    // .clk(clk),
    // .PC(PC),
    .icode(icode),
    .condition_bit(cnd),
    .valC(valC),
    .valM(valM),
    .valP(valP),
    .final_PC(updated_pc)
```

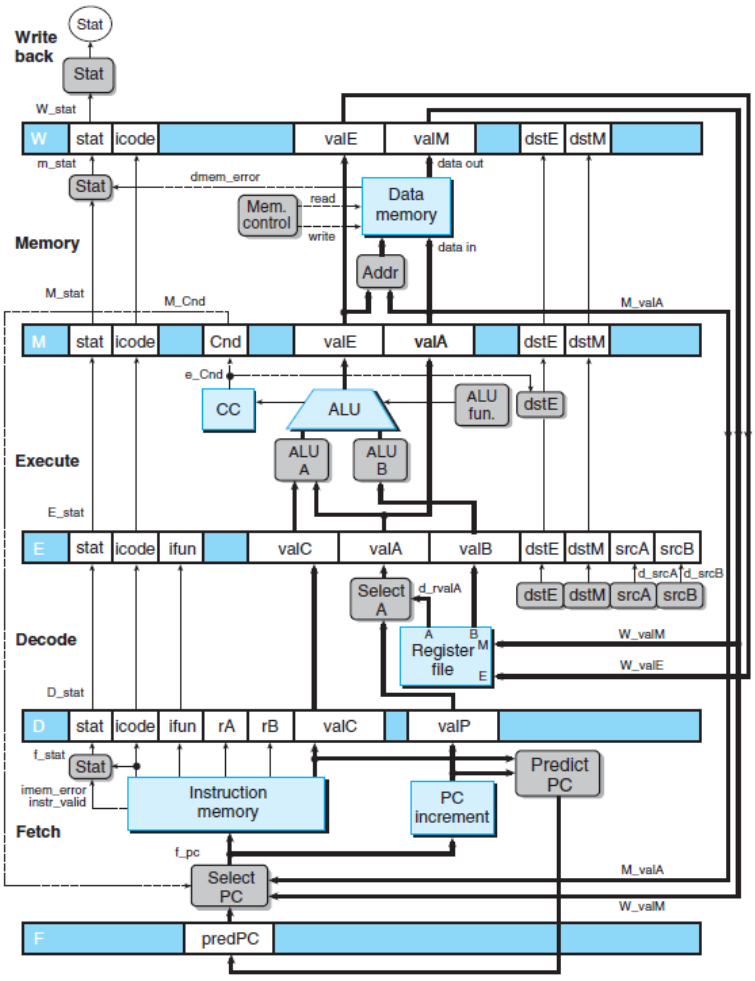
```
);

always #5 clk=~clk;


initial begin
  $dumpfile("processor.vcd");
  $dumpvars(0,processor);
  stat[0]=1;
  stat[1]=0;
  stat[2]=0;
  clk=0;
  PC=64'd32;
end

always@(*)
begin
  PC=updated_pc;
end

always@(*)
begin
  if(hltins)
  begin
    stat[2]=hltins;
    stat[1]=1'b0;
    stat[0]=1'b0;
  end
  else if(instr_val)
  begin
    stat[1]=instr_val;
    stat[2]=1'b0;
    stat[0]=1'b0;
  end
  else
  begin
    stat[0]=1'b1;
    stat[1]=1'b0;
```

```
      stat[2]=1'b0;
    end
  end

  always@(*)
  begin
    if(stat[2]==1'b1)
    begin
      $finish;
    end
  end

  initial
  begin
    $monitor($time , " clk=%0d PC=%0d icode=%b ifun=%b rA=%d rB=%d valA=%0d valB=%0d v
datamem=%0d\n",clk,PC,icode,ifun,rA,rB,valA,valB,valC,valE,valM,instr_val,imem_er,cnd,
  end

endmodule
```

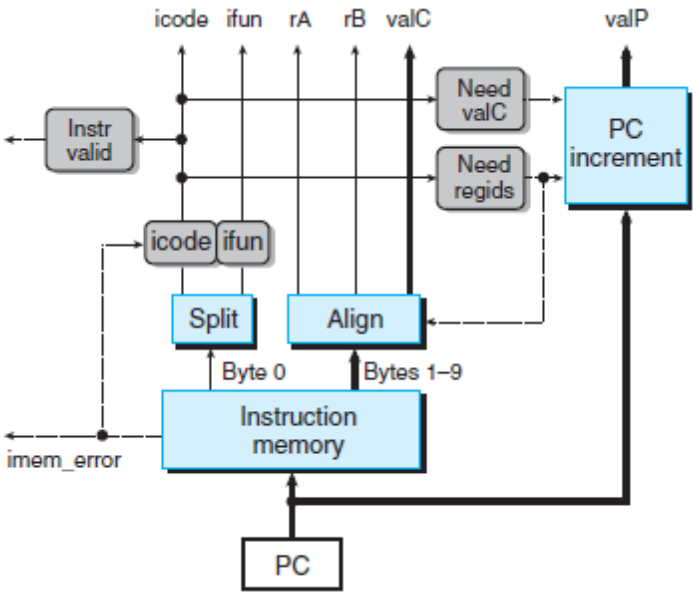
Pipeline:

Architecture Diagrams:

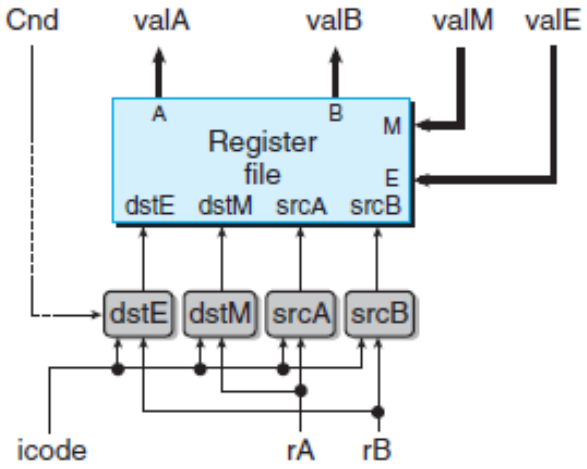



Write back
Stat
Stat
W_stat
W
stat
icode
valE
valM
dstE
dstM
m_stat
data out
Stat
dmem_error
Mem. control
read
write
Data memory
Memory
data in
Addr
M_stat
M_Cnd
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
dstE
Execute
ALU A
ALU B
E_stat
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
Select A
d_rvalA
dstE
dstM
srcA
srcB
Decode
A
B
M
E
Register file
W_valM
W_valE
D_stat
D
stat
icode
ifun
rA
rB
valC
valP
f_stat
Stat
Predict PC
imem_error
instr_valid
Instruction memory
PC increment
Fetch
f_pc
Select PC
M_valA
W_valM
F
predPC

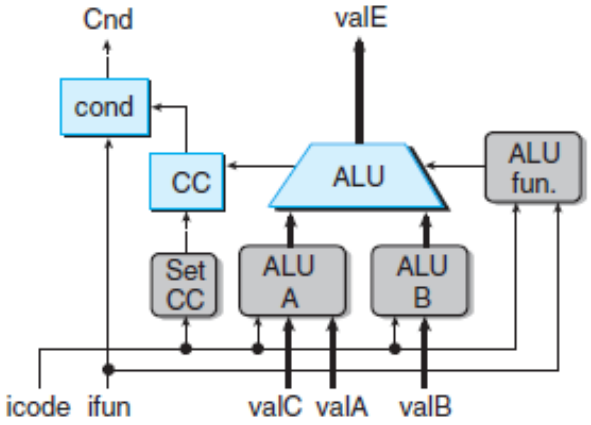
Fetch:



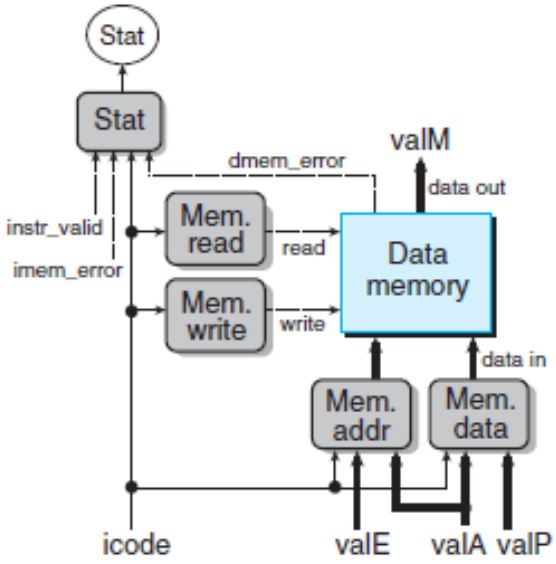
Decode:



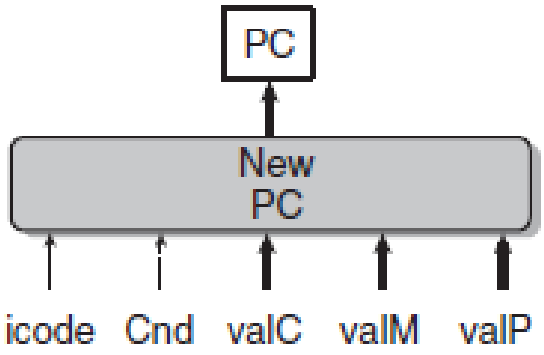
Execute:



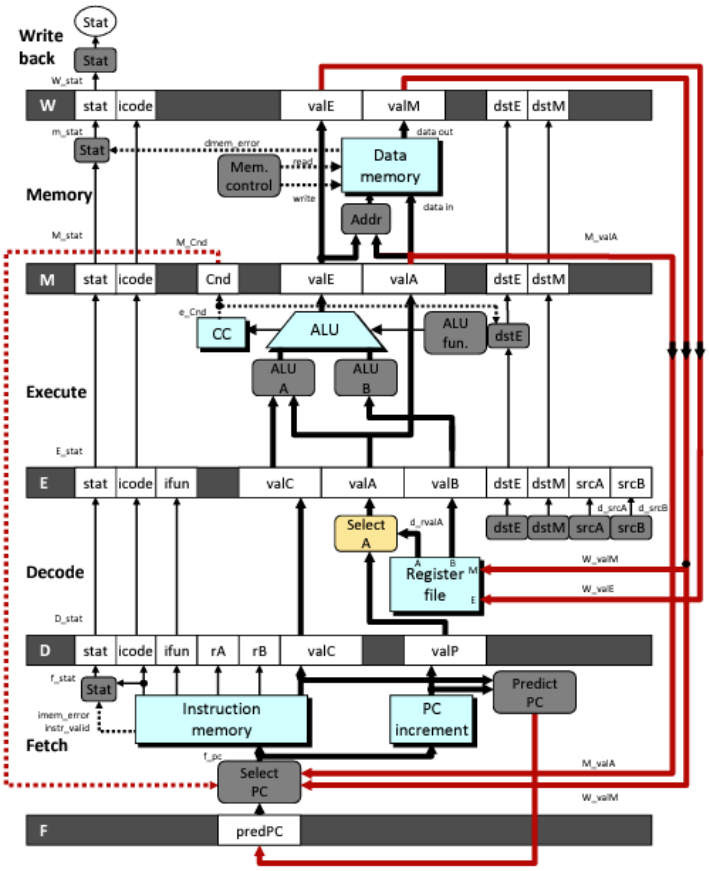
Memory:

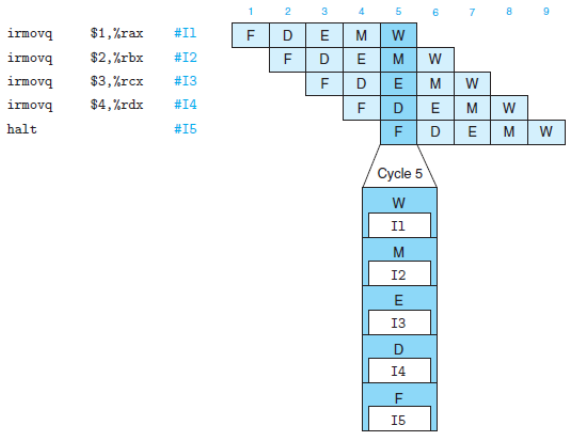


PC Update:



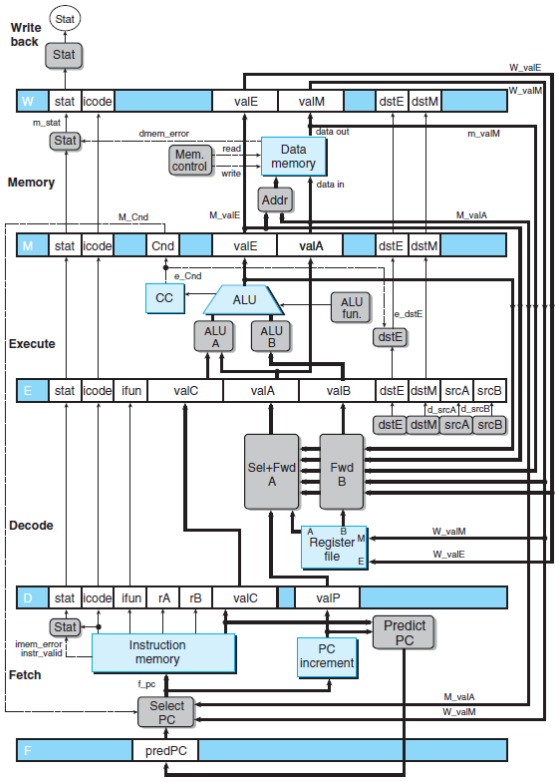
Feedback Loops:





A pipeline diagram for this instruction sequence is shown on the right side of the figure. The flow of each instruction through the pipeline stages is depicted in this graphic, with time rising from left to right. The numbers along the top indicate the clock cycles during which the various processes take place. In cycle 1, for example, instruction I1 is fetched and then sent through the pipeline stages, with the result written to the register file at the end of cycle 5. In cycle 2, instruction I2 is retrieved, and the result is written back at the conclusion of cycle 6, and so on. We present an enlarged view of the pipeline for cycle 5 at the bottom. Right now there is an instruction in each of the pipeline phases.

Load/Use Data Hazards



Because memory reads occur late in the pipeline, one type of data hazard cannot be controlled only by forwarding. In the example above, one instruction (the mrmovq at address 0x028) reads a value from memory for register percent rax, whereas the next instruction (the addq at address 0x032) requires this value as a source operand. In the lower portion of the picture, expanded views of cycles 7 and 8 are presented, with the assumption that all programme registers start with a value of 0. The value of the register is required by the addq instruction in cycle 7, but it is not created until cycle 8 by the mrmovq instruction. The forwarding logic would have to have the value go backward in time in order to "forward" from the mrmovq to the addq! Since this is clearly unachievable, we'll have to come up with another way to deal with this type of data risk.

Pipeline Summary:

- Concept:
  - Break instruction execution into 5 stages
  - Run instructions through in pipelined mode
- Limitations:
  - Can’t handle dependencies between instructions when instructions follow too closely
  - Data dependency
    - One instruction writes register, later one reads it
  - Control dependency

* Instruction sets PC in way that pipeline did not predict correctly
* Mispredicted branch and return