**Y86-64 ISA Implementation**

# Intro To Processor Architecture

**Team 12**

**Eswara Rohan (2020102002)**

**Keerthi Pothalaraju (2020102010)**

## Objective -

**Objective of this project is to develop a processor architecture design based on the Y86 ISA using Verilog. The processor should be implemented in both sequential and pipelined manner. The design approach is modular. The processor should be able to execute all the instructions in Y86-64 ISA.**

# Y86-64 ISA Implementation

## Supported Instructions -

| Instruction | What does it do? |
|---|---|
| IHALT | Code for halt instruction |
| INOP | Code for nop instruction |
| IRRMOVQ | Code for rrmovq (register to register) instruction |
| IIRMOVQ | Code for irmovq (immediate to register) instruction |
| IRMMOVQ | Code for rmmovq (register to memory) instruction |
| IMRMOVQ | Code for mrmovq (memory to register) instruction |
| IOPL | Code for integer operation instructions |
| IJXX | Code for jump instructions |
| ICALL | Code for call instruction |
| IRET | Code for ret instruction |
| IPUSHQ | Code for pushq instruction |
| IPOPQ | Code for popq instruction |

# Y86-64 ISA Implementation

## halt

- halt stops instruction execution.
- It requires only a single byte.

## nop

- nop stands for no operation
- It also requires a single byte as in halt.

**X86-64 data movement instruction movq is split into four cases - rrmovq , irmovq , rmmovq , mrmovq. Memory referencing uses a register plus displacement address computation.**

## rrmovq

- This is register to register instruction.
- rrmovq instruction is a special case of a conditional move,where the move condition always holds.
- rrmovq has same format as cmovXX, but the destination register updated only if the condition codes satisfy the required constraints.
- **cmovXX** instruction represents seven different branch Instructions with different branch conditions. Branching is based on the setting of the condition codes by the arithmetic instructions.
- The seven instructions include - **rrmovq, cmovle, cmovl, cmove, cmovne, cmovge and cmovg.**

**irmovq, rmmovq, mrmovq**

- irmovq is immediate to register instruction.
- rmmovq is register to memory instruction.
- mrmovq is memory to register instruction.

# Y86-64 ISA Implementation

## OPq

- OPq instruction performs four different arithmetic and logical operations

➔ ADD,
➔ SUBTRACT
➔ XOR
➔ AND

## jXX

- jXX instructions represents seven different branch Instructions with different branch conditions. Branches are taken according to the type of branch and the settings of the conditional codes. The branch conditions are the same as with x86-64.

## call

- call instruction pushes the return address onto the stack and then jumps to the designation.

## ret

- ret instruction pops the return address from the stack and jumps to that location.

## push

- pushq instruction pushes 8 byte words onto the stack.
- Pushing involves first decrementing the stack pointer by eight and then writing a word to the address given by the stack pointer.

## pop

- popq instruction pops 8 byte words off the stack and involves reading the top word on the stack and then incrementing the stack pointer by eight.

# Y86-64 ISA Implementation

## Sequential Y86-64 Implementation

**Description –**

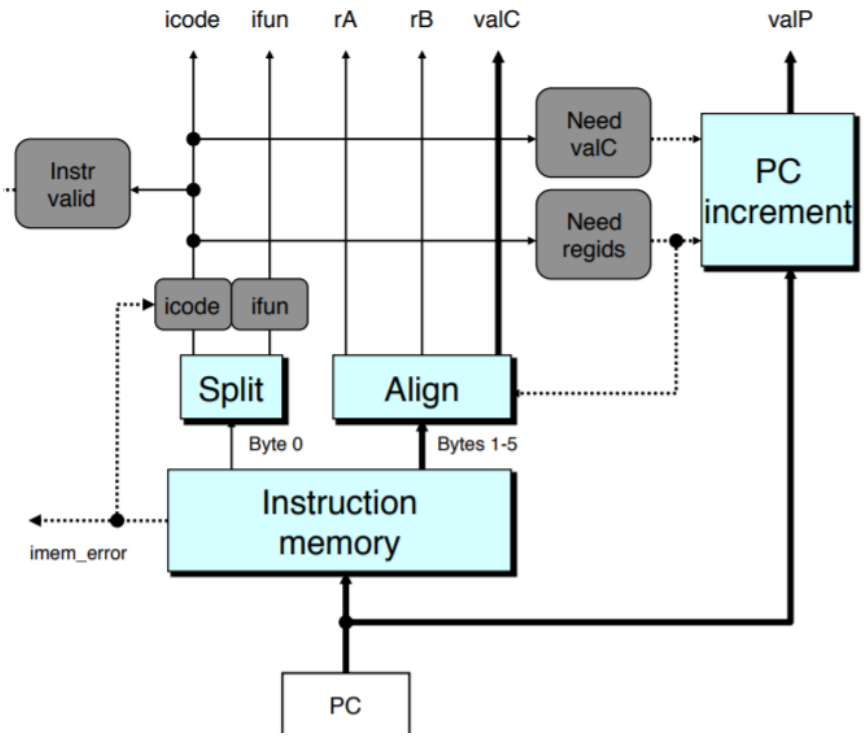
We build the Y86-64 processor in stages. On each clock cycle, SEQ performs all the steps required to process a complete instruction.

The steps and operations performed on each step are described below –

1. **Fetch** - Read instruction from instruction memory.
2. **Decode** - Read program registers
3. **Execute** - Compute value or address
4. **Memory -** Read or write back data.
5. **Write Back -** Write program registers.
6. **PC Update -** Update the program counter

# Y86-64 ISA Implementation

# Fetch




-> This stage reads the bytes of an instruction from memory using Program Counter (PC)  as the memory address.

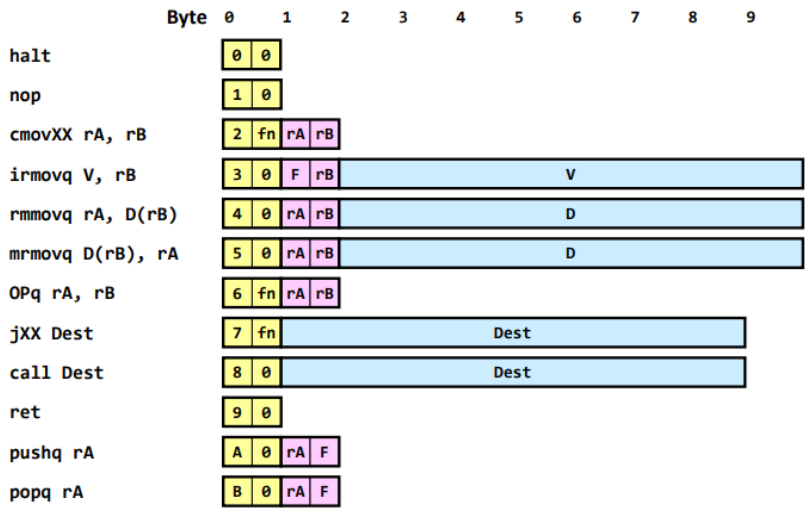
**Computed Values in this stage are -**

**icode** – Instruction Code
**ifun** – Function Code
**rA** – Inst.  Register A
**rB** – Inst. Register B
**valC** – Instruction Constant
**valP** – Incremented Program Counter

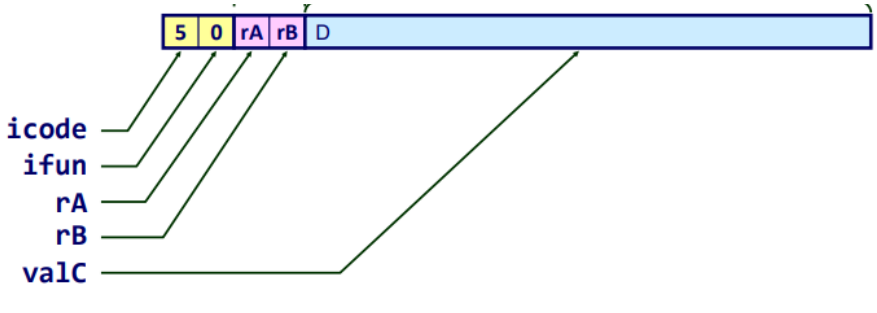
# Y86-64 ISA Implementation

## Implementation Of Fetch Stage -

Below is Y86-64 Instruction Set -




Below is the method to determine icode, ifun, rA, rB and valC of a given instruction.




Based on the above data, we can determine icode, ifun, rA, rB, valC values.

# Y86-64 ISA Implementation

- In case of halt, icode = 0, ifun = 0, valP = PC+64'd1

- In case of nop, icode = 1, ifun = 0, valP = PC+64'd2

- In case of cmovXX, icode = 2, ifun for - rrmovq = 0, cmovle = 1, cmovl = 2, cmove = 3 , cmovne = 4, cmovge = 5, cmovg = 6
  valP = PC+64'd2

- In case of irmovq, icode = 3, ifun = 0, valP = PC+64'd10

- In case of rmmovq, icode = 4, ifun = 0, valP = PC+64'd10.

- In case of mrmovq, icode = 5, ifun = 0, valP = PC+64'd10.

- In case of OPq, icode = 6, ifun for - addq = 0, subq = 1, andq = 2, xorq = 3. valP = PC+64'd2.

- In case of jXX, icode = 7, ifun for - jmp = 0, jle = 1, jl = 2, je = 3, jne = 4,jge = 5, jg = 6 , valP = PC+64'd9.

- In case of call, icode = 8, ifun = 0, valP = PC+64'd9.
- In case of ret, icode = 9, ifun = 0, valP = PC+64'd1.

# Y86-64 ISA Implementation

- In case of pushq, icode = 10, ifun = 0, valP = PC+64'd2

- In case of popq, icode = 11, ifun = 0, valP = PC+64'd2.

# Y86-64 ISA Implementation

```
module Fetch(clk, PC , icode , ifun , rA , rB , valC , valP , halt_prog , is_instruction_valid,pcvalid) ;

parameter SIZE = 63 ;

// 1. In the fetch stage, we need to read the instruction from Instruction_memory using the PC value
// 2. The first instruction byte is divided into two 4-bits referred to as icode and ifun
//    icode tells us the instruction
//    ifun tells the function of instruction ,else it is 0


// The inputs
input [63:0] PC ;
input clk ;

// The outputs
output reg [3:0] ifun ;
output reg [3:0] icode ;
output reg [3:0] rA ;
output reg [3:0] rB ;
output reg signed[63:0] valC ;
output reg signed[63:0] valP ;
output reg is_instruction_valid ;
output reg halt_prog ;
output reg pcvalid ;

// Registers
reg [7:0] Instruction_memory[0:128]; // Consider the Instruction_memory contains the instruction/data/storage etc at values given by PC
                                     // and you can have 1024 values of PC


reg [0:7] byte1 ;
reg [0:7] byte2 ;

reg signed [0:63] if_valC_req ;

initial begin


    Instruction_memory[0] = 8'b01100000; //6 add
    Instruction_memory[1] = 8'b00000011; //%rax %rbx and store in rbx

    Instruction_memory[2] = 8'b00100000; // rrmovq
    Instruction_memory[3] = 8'b00000011; // src = %rax dest = %rdx
```

# Y86-64 ISA Implementation

```
46
47      Instruction_memory[4] = 8'b01000000; //4-rmmovq
48      Instruction_memory[5] = 8'b00000011; //rax and (rbx)
49      Instruction_memory[6] = 8'b00000000;  //VALC ---->from 6 to 13
50      Instruction_memory[7] = 8'b00000000;
51      Instruction_memory[8] = 8'b00000000;
52      Instruction_memory[9] = 8'b00000000;
53      Instruction_memory[10] = 8'b00000000;
54      Instruction_memory[11] = 8'b00000000;
55      Instruction_memory[12] = 8'b00000000;
56      Instruction_memory[13] = 8'b00001111;
57
58
59      Instruction_memory[14] = 8'b00010000; //no operation
60      Instruction_memory[15] = 8'b00010000; //no operation
61      Instruction_memory[16] = 8'b00000000; //halt
62
63
64  end
65
66  always@(posedge clk)
67    begin
68
69      byte1 = {Instruction_memory[PC]} ;
70      byte2 = {Instruction_memory[PC+1]} ;
71
72      icode = byte1[0:3];
73      ifun  = byte1[4:7];
74
75      if_valC_req = 64'd0 ;
76      valC = 0;
77
78      if(icode==4'b0011 || icode==4'b0100 || icode==4'b0101 || icode==4'b0111 || icode ==4'b1000)
79      begin
80      if_valC_req = {Instruction_memory[PC+2] , Instruction_memory[PC+3] , Instruction_memory[PC+4] , Instruction_memory[PC+5] , Instruction_memory[PC+6] , Instruction_memory[PC+7] , Instruction_memory[PC+8] ,
81      end
82
83
84      is_instruction_valid = 1'b1 ;
85
86      halt_prog = 0 ;
87
88      // icode gives the instruction type
89
```

# Y86-64 ISA Implementation

AND.v > ...

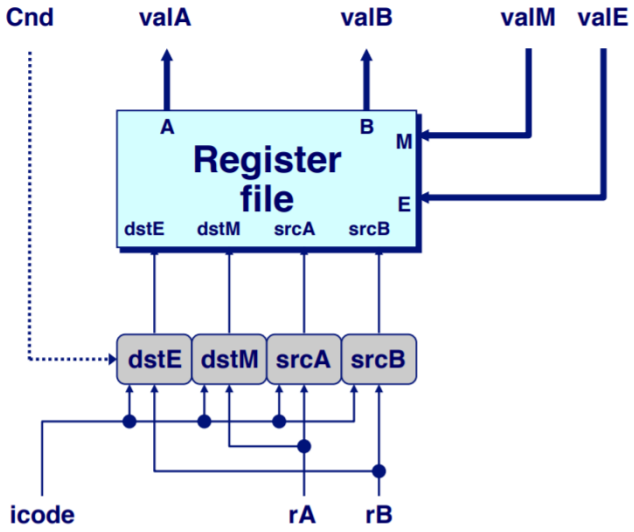
```
90      if(icode == 4'b0000) // Halt instruction should be called
91      begin
92        halt_prog = 1;
93        valP = PC + 64'd1;  // since only 1byte
94      end
95
96      else if(icode == 4'b0001) //nop
97      begin
98        valP = PC + 64'd1;
99      end
100
101     else if(icode == 4'b0010) //cmovxx
102     begin
103       rA = byte2[0:3];
104       rB = byte2[4:7];
105       valP = PC + 64'd2;
106     end
107
108     else if(icode == 4'b0011) //irmovq
109     begin
110       rA = byte2[0:3];
111       rB = byte2[4:7];
112       valC = if_valC_req;
113       valP = PC + 64'd10;
114     end
115
116     else if(icode == 4'b0100) //rmmovq
117     begin
118       rA = byte2[0:3];
119       rB = byte2[4:7];
120       valC = if_valC_req;
121       valP = PC + 64'd10;
122     end
123
124     else if(icode == 4'b0101) //mrmovq
125     begin
126       rA = byte2[0:3];
127       rB = byte2[4:7];
128       valC = if_valC_req;
129       valP = PC + 64'd10;
130     end
131
132     else if(icode == 4'b0110) //OPq
133     begin
134       rA = byte2[0:3];
```

# Y86-64 ISA Implementation

```
    else if(icode == 4'b0110) //OPq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
    end

    else if(icode==4'b0111) //jxx
    begin
      valC = if_valC_req;
      valP = PC + 64'd9;
    end

    else if(icode == 4'b1000) //call
    begin
      valC = if_valC_req;
      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 = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
    end

    else if(icode==4'b1011) //popq
    begin
      rA = byte2[0:3];
      rB = byte2[4:7];
      valP = PC + 64'd2;
    end

    else
    begin
      is_instruction_valid = 1'b0;
    end

    pcvalid = 0;
    if(PC > 1023)
    begin
      pcvalid = 1 ;
    end

  end

endmodule
```

**Y86-64 ISA Implementation**

# Decode and Write-Back




Decode reads the registers designated by rA and rB and output values valA and valB but for some instructions it reads register %rsp.
Write-Back write program registers.

During the decode stage, we read both operands. These are supplied to the ALU in the execute stage, along with the function specifier ifun, so that valE becomes the instruction result.

**Computed Values in this stage are -**

**valA -** Register Value A
**valB -** Register Value B

# Y86-64 ISA Implementation

## Implementation Of Decode and Write Back Stage -

| | | | |
|---|---|---|---|
| OPq | Decode | valA ← R[rA] | Read operand A |
| rmmovq | Decode | valA ← R[rA] | Read operand A |
| mrmovq | Decode | | |
| irmovq | Decode | | |
| pushq | Decode | valA ← R[rA] | Read operand A |
| popq | Decode | valA ← R[%rsp] | Read stack pointer |
| cmovXX | Decode | valA ← R[rA] | Read operand A |
| jXX | Decode | | |
| call | Decode | | |
| ret | Decode | valA ← R[%rsp] | Read stack pointer |

| | | | |
|---|---|---|---|
| OPq | Write Back | R[rB] ← valE | Write back result |
| rmmovq | Write Back | | |
| mrmovq | Write Back | | |
| irmovq | Write Back | R[rB] ← valE | Write back result |
| pushq | Write Back | R[%rsp] ← valE | Update stack pointer |
| popq | Write Back | R[%rsp] ← valE | Update stack pointer |
| cmovXX | Write Back | R[rB] ← valE | Write back result |
| jXX | Write Back | | |
| call | Write Back | R[%rsp] ← valE | Update stack pointer |
| ret | Write Back | R[%rsp] ← valE | Update stack pointer |

# Y86-64 ISA Implementation

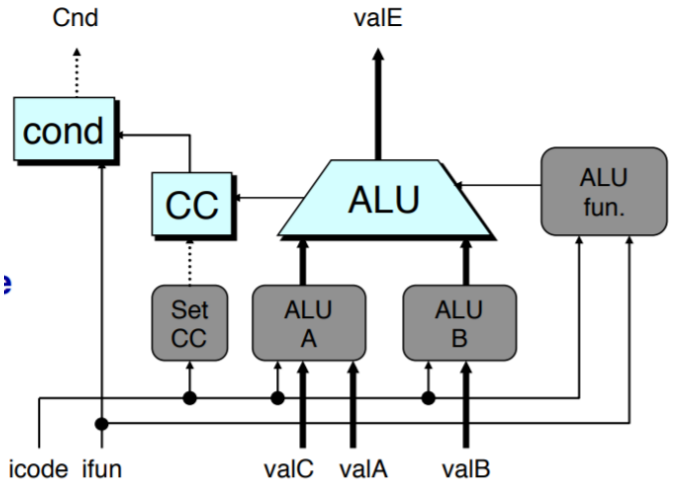
```
module decode(valA , valB , clk , icode , rA , rB ) ; // Don't get confused we don not need to send halt coz we are decoding

parameter SIZE = 63 ;


input clk ;
input [3:0] rA ;
input [3:0] rB ;
//input reg [3:0] rsp ;
input [3:0] icode ;
reg [63:0] register_memory [0:14] ;

assign rsp = 64'd4 ;

// If we were to consider that we have 15 register_memorys from %rax to %r14, the stack pointer is %rsp and it is in the 4th place

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

initial
  begin
    register_memory[0] = 64'd20;
    register_memory[1] = 64'd21;
    register_memory[2] = 64'd54;
    register_memory[3] = 64'd1;
    register_memory[4] = 64'd31;
    register_memory[5] = 64'd63;
    register_memory[6] = 64'd12;
    register_memory[7] = 64'd95;
    register_memory[8] = 64'd72;
    register_memory[9] = 64'd52;
    register_memory[10] = 64'd3;
    register_memory[11] = 64'd8;
    register_memory[12] = 64'd9;
    register_memory[13] = 64'd4;
    register_memory[14] = 64'd6;
    begin


always@(posedge clk)
  begin

    if(icode == 4'b0000) // Halt instruction should be called
    begin
      valA = 0 ;
      valB = 0 ;
    end

    else if(icode == 4'b0001) //nop
    begin
      valA = 0 ;
      valB = 0 ;
    end

    else if(icode == 4'b0010) //cmovxx
    begin
```

# Y86-64 ISA Implementation

```
57          valA = register_memory[rA] ;
58          valB = register_memory[rB] ;
59        end
60      /*  else if(icode == 4'b0011) //irmovq
61        begin
62          valB = register_memory[rB] ;
63        end */
64
65        else if(icode == 4'b0100) //rmmovq
66        begin
67          valA = register_memory[rA] ;
68          valB = register_memory[rB] ;
69        end
70
71        else if(icode == 4'b0101) //mrmovq
72        begin
73          valA = 0 ;
74          valB = register_memory[rB] ;
75        end
76
77        else if(icode == 4'b0110) //OPq
78        begin
79          valA = register_memory[rA] ;
80          valB = register_memory[rB] ;
81        end
82
83       /* else if(icode==4'b0111) //jxx
84        begin
85
86        end */
87
88        else if(icode == 4'b1000) //call
89        begin
90          valA = 0;
91          valB = register_memory[rsp] ;
92        end
93
94        else if(icode == 4'b1001) //ret
95        begin
96          valA = register_memory[rsp] ;
97          valB = register_memory[rsp] ;
98        end
99
100       else if(icode == 4'b1010) //pushq
101       begin
102         valA = register_memory[rA] ;
103         valB = register_memory[rsp] ;
104       end
105
106       else if(icode==4'b1011) //popq
107       begin
108         valA = register_memory[rsp] ;
109         valB = register_memory[rsp] ;
110       end
111     end
112   endmodule
```

**Y86-64 ISA Implementation**

# Execute




This stage performs either of the following two actions -

a) ALU performs the operation specified by ifun and computes effective address of memory.
b) Increments (or) Decrements the stack pointer.

**Computed Values in this stage are -**

**valE** - ALU Result
**Cnd** - Constant to determine whether to take a branch or not.

# Y86-64 ISA Implementation

## Implementation Of Execute Stage -

| | | | |
|---|---|---|---|
| OPq | Execute | valE ← valB OP valA | Perform ALU operation |
| rmmovq | Execute | valE ← valB + valC | Compute effective address |
| mrmovq | Execute | valE ← valB + valC | Compute effective address |
| irmovq | Execute | valE ← valB + valC | Pass valC through ALU |
| pushq | Execute | valE ← valB + (-8) | Decrement stack pointer |
| popq | Execute | valE ← valB + 8 | Increment stack pointer |
| cmovXX | Execute | valE ← valB + valA | Pass valA through ALU |
| jXX | Execute | | |
| call | Execute | valE ← valB + (-8) | Decrement stack pointer |
| ret | Execute | valE ← valB + 8 | Increment stack pointer |

## Condition Codes -

1) **Carry Flag (CF) -** This is used to detect overflow for unsigned operations . This is determined by the most recent operation generated by carry out of the most significant bit.
2) **Zero Flag (ZF) -** This flag comes into effect when the most recent operation yields zero.
3) **Sign Flag (SF) -** This flag comes into effect when the most recent operation yields a negative value.
4) **Overflow Flag (OF) -** This flag comes into effect if the most recent operation caused a two's complement overflow - either positive or negative.

➔ In case of logical operations, the carry and overflow flags are set to zero.
➔ In case of shift operations, the carry flag is set the last shifted out, while the overflow flag is set to zero.
➔ INC and DEC instructions set the overflow flag and zero flag but leave the carry flag unchanged.

# Y86-64 ISA Implementation

## Jump instructions and condition codes -

| Instruction | | Synonym | Jump condition | Description |
|---|---|---|---|---|
| `jmp` | *Label* | | 1 | Direct jump |
| `jmp` | **Operand* | | 1 | Indirect jump |
| `je` | *Label* | `jz` | `ZF` | Equal / zero |
| `jne` | *Label* | `jnz` | `~ZF` | Not equal / not zero |
| `js` | *Label* | | `SF` | Negative |
| `jns` | *Label* | | `~SF` | Nonnegative |
| `jg` | *Label* | `jnle` | `~(SF ^ OF) & ~ZF` | Greater (signed >) |
| `jge` | *Label* | `jnl` | `~(SF ^ OF)` | Greater or equal (signed >=) |
| `jl` | *Label* | `jnge` | `SF ^ OF` | Less (signed <) |
| `jle` | *Label* | `jng` | `(SF ^ OF) \| ZF` | Less or equal (signed <=) |
| `ja` | *Label* | `jnbe` | `~CF & ~ZF` | Above (unsigned >) |
| `jae` | *Label* | `jnb` | `~CF` | Above or equal (unsigned >=) |
| `jb` | *Label* | `jnae` | `CF` | Below (unsigned <) |
| `jbe` | *Label* | `jna` | `CF \| ZF` | Below or equal (unsigned <=) |

## cmovXX instructions and condition codes -

| 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 <=) |

# Y86-64 ISA Implementation

```
module AND(O,I1,I2);

  parameter SIZE = 63;

  input signed [63:0] I1;
  input signed [63:0] I2;

  output signed [63:0] O;

  genvar i;

  for(i=0;i<=SIZE;i=i+1)
  begin:bitnum
    and g1 (O[i],I1[i],I2[i]);
  end

endmodule
```

```
module XOR(O,I1,I2);

  parameter SIZE = 63;

  input signed [63:0] I1;
  input signed [63:0] I2;

  output signed [63:0] O;

  genvar i;

  for(i=0;i<=SIZE;i=i+1)
  begin:bitnum
    xor g1 (O[i],I1[i],I2[i]);
  end

endmodule
```

```
module subtraction(O,overflow,I1,I2);
parameter SIZE = 63;
input signed [63:0] I1;
input signed [63:0] I2;
output signed [63:0] O;
output signed overflow;
wire [63:0] Y;
wire [64:0] complementary;
wire [64:0] dummy;
wire [64:0] CARRY;
wire [63:0] SUM;
genvar i;
genvar m;
genvar n;

assign CARRY[0] = 1'b0;
assign dummy[0] = 1'b1;

// complementary calculation
for(m=0;m<=SIZE;m=m+1)
begin:bitnum1
not g6 (Y[m],I2[m]) ;
end
for(n=0;n<=SIZE;n=n+1)
begin:bitnum2
xor g7 (complementary[n],dummy[n],Y[n]) ;
and g8 (dummy[n+1],dummy[n],Y[n]) ;
end

for(i=0;i<=SIZE;i=i+1) begin:bitnum
    wire [63:0] w1;
    wire [63:0] w2;
    wire [63:0] w3;

    xor g1 (w1[i],I1[i],complementary[i]) ;
    xor g2 (O[i],w1[i],CARRY[i]) ;
    and g3 (w2[i],CARRY[i],w1[i]) ;
    and g4 (w3[i],I1[i],complementary[i]) ;
    or g5 (CARRY[i+1],w3[i],w2[i]) ;

  end
  xor g9 (overflow,CARRY[64],CARRY[63]) ;
endmodule
```

```
module addition(O,overflow,I1,I2);

  parameter SIZE = 63;
  input signed [63:0] I1;
  input signed [63:0] I2;
  output signed [63:0] O;
  output signed overflow;
  wire [64:0] CARRY;
  wire [63:0] SUM;
  assign CARRY[0] = 1'b0;
  genvar i;
   for(i=0;i<=SIZE;i=i+1) begin:bitnum
      wire [63:0] w1;
      wire [63:0] w2;
      wire [63:0] w3;

      xor g1 (w1[i],I1[i],I2[i]) ;
      xor g2 (O[i],w1[i],CARRY[i]) ;
      and g3 (w2[i],CARRY[i],w1[i]) ;
      and g4 (w3[i],I1[i],I2[i]) ;
      or g5 (CARRY[i+1],w3[i],w2[i]) ;
  end
  xor g6 (overflow,CARRY[64],CARRY[63]) ;
endmodule
```

# Y86-64 ISA Implementation

```
1   `include "ALU.v"
2   module Execute(icode,ifun,valA,valB,valC,valE,clk,cnd,ZF,SF,OF);
3     parameter SIZE = 63 ;
4     input [3:0] icode; //Instruction Code
5     input [3:0] ifun;  //Instruction Function
6     input signed [63:0] valA; //Register Value A
7     input signed [63:0] valB; //Register Value B
8     input signed [63:0] valC; //Instruction Constant
9     input clk;         //Clock
10    output reg [63:0] valE; //EXECUTION Result
11    output reg cnd;        //1 bit signal which determine whether to take branch or not
12    output reg ZF;         //Zero Flag
13    output reg SF;         //Sign Flag
14    output reg OF;     //Overflow Flag
15    reg in1,in2,in3,in4,in5,in6,in7;
16    wire OUTP1,OUTP2,OUTP3,OUTP4;
17    reg [1:0] CONTROL;
18    reg signed [63:0] Input1;
19    reg signed [63:0] Input2;
20    reg signed [63:0] OP;
21    wire signed [63:0] Output;
22    wire OVERFLOW;
23    not g1 (OUTP1,in1);
24    or g2 (OUTP2,in2,in3);
25    and g3 (OUTP3,in4,in5);
26    xor g4 (OUTP4,in6,in7);
27    initial begin
28      ZF = 0;
29      SF = 0;
30      OF = 0;
31      CONTROL = 2'b00;
32      Input1 = 64'b0;
33      Input2 = 64'b0;
34    end
35    ALU ALU_1(.OVERFLOW(OVERFLOW),.Output(Output),.CONTROL(CONTROL),.Input1(Input1),.Input2(Input2));
36    always @(*) begin
37      if (clk == 1) begin
38        cnd = 0;
39        if (icode == 4'b0010)
40         begin //cmovXX-rrmovq,cmovle,cmovl,cmove,cmovne,cmovge,cmovg
41          if (ifun == 4'b0000)
42          begin //rrmovq
43            valE = valA + 64'd0;
44            cnd = 1;
45          end
46          else if (ifun == 4'b0001) begin //cmovle - Less OR Equal (signed <=) - (SF ^ OF) | ZF
47            valE = valA + 64'd0;
48            in6 = SF;
49            in7 = OF;
50            if (OUTP4) begin
51              cnd = 1;
52            end
53            else if (ZF) begin
54              cnd = 1;
55            end
56          end
```

SystemVerilog: 10 indexed objects

# Y86-64 ISA Implementation

```
57          else if (ifun == 4'b0010) begin //cmovl - Move when Less - (SF ^ OF)
58            valE = valA + 64'd0;
59            in6 = SF;
60            in7 = OF;
61            if (OUTP4) begin
62              cnd = 1;
63            end
64          end
65          else if (ifun == 4'b0011) begin //cmove - Move When Equal - ZF
66            valE = valA + 64'd0;
67            if (ZF) begin
68              cnd = 1;
69            end
70          end
71          else if (ifun == 4'b0100) begin //cmovne - Move When Not Equal - ~ZF
72            valE = valA + 64'd0;
73            in1 = ZF;
74            if (OUTP1) begin
75              cnd = 1;
76            end
77          end
78          else if (ifun == 4'b0101) begin //cmovge - Move When Greater Than Or Equal -- ~(SF ^ OF)
79            valE = valA + 64'd0;
80            in6 = SF;
81            in7 = OF;
82            in1 = OUTP4;
83            if (OUTP1) begin
84              cnd = 1;
85            end
86          end
87          else if (ifun == 4'b0110) begin //cmovg - Move When Greater --  ~(SF ^ OF) & ~ZF
88            valE = valA + 64'd0;
89            in6 = SF;
90            in7 = OF;
91            in1 = OUTP4;
92            if (OUTP1) begin
93              in1 = ZF;
94              if (OUTP1) begin
95                cnd = 1;
96              end
97            end
98          end
99        end
100       else if (icode == 4'b0011) begin //irmovq
101         valE = valC + 64'd0;
102       end
103       else if (icode == 4'b0100) begin //rmmovq
104         valE = valB + valC;
105       end
106       else if (icode == 4'b0101) begin //mrmovq
107         valE = valB + valC;
108       end
109       else if (icode == 4'b0110) begin //OPq - Addition, Subtraction, AND, XOR
110         ZF = (Output == 1'b0);
111         SF = (Output < 1'b0);
112         OF = (Input1 < 1'b0 == Input2 < 1'b0) && (Output < 1'b0 != Input1 < 1'b0);
```

0 SystemVerilog: 10 indexed objects

# Y86-64 ISA Implementation

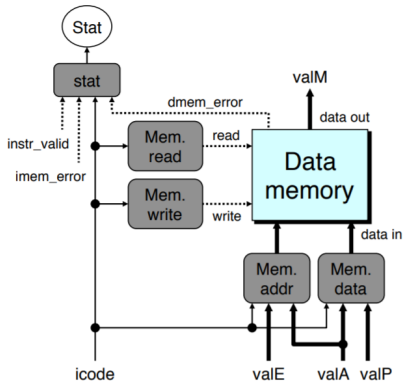
```
113             if (ifun == 4'b0000) begin //ADD
114               CONTROL = 2'b00;
115               Input1 = valA;
116               Input2 = valB;
117             end
118             if (ifun == 4'b0001) begin //SUBTRACT
119               CONTROL = 2'b01;
120               Input1 = valA;
121               Input2 = valB;
122             end
123             if (ifun == 4'b0010) begin //AND
124               CONTROL = 2'b10;
125               Input1 = valA;
126               Input2 = valB;
127             end
128             if (ifun == 4'b0011) begin //XOR
129               CONTROL = 2'b11;
130               Input1 = valA;
131               Input2 = valB;
132             end
133             OP = Output;
134             valE = OP;
135           end
136           else if (icode == 4'b0111) begin //jxx- jmp, jle, jl, je, jne, jge, and jg
137             if (ifun == 4'b0000) begin //jmp - Jump  Unconditionally
138               cnd = 1;
139             end
140             else if (ifun == 4'b0001) begin //jle - Jump when Less than or equal -- (SF ^ OF) | ZF
141               in6 = SF;
142               in7 = OF;
143               if (OUTP4) begin
144                 cnd = 1;
145               end
146               else if (ZF) begin
147                 cnd = 1;
148               end
149             end
150             else if (ifun == 4'b0010) begin //jl - Jump when Less -- (SF ^ OF)
151               in6 = SF;
152               in7 = OF;
153               if (OUTP4) begin
154                 cnd = 1;
155               end
156             end
157             else if (ifun == 4'b0011) begin //je - Jump when Equal or Zero -- ZF
158               if (ZF) begin
159                 cnd = 1;
160               end
161             end
162             else if (ifun == 4'b0100) begin //jne - Jump when not equal -- ~ZF
163               in1 = ZF;
164               if (OUTP1) begin
165                 cnd = 1;
166               end
167             end
168             else if (ifun == 4'b0101) begin //jge - Jump when greater than or equal -- ~(SF ^ OF)
```

# Y86-64 ISA Implementation

```
168         else if (ifun == 4'b0101) begin //jge - Jump when greater than or equal -- ~(SF ^ OF)
169           in6 = SF;
170           in7 = OF;
171           in1 = OUTP4;
172           if (OUTP1) begin
173             cnd = 1;
174           end
175         end
176         else if (ifun == 4'b0110) begin //jg - Jump when Greater -- ~(SF ^ OF) & ~ZF
177           in6 = SF;
178           in7 = OF;
179           in1 = OUTP4;
180           if (OUTP1) begin
181             in1 = ZF;
182             if (OUTP1) begin
183               cnd = 1;
184             end
185           end
186         end
187       end
188       else if (icode == 4'b1000) begin //Call
189         valE = valB - 64'd8;
190       end
191       else if (icode == 4'b1001) begin //Ret
192         valE = valB + 64'd8;
193       end
194       else if (icode == 4'b1010) begin //pushq
195         valE = valB - 64'd8;
196       end
197       else if (icode == 4'b1011) begin //popq
198         valE = valB + 64'd8;
199       end
200     end
201   end
202 endmodule
```

# Y86-64 ISA Implementation

# Memory




Memory either reads data from memory or writes data to memory.

**Computed Values in this stage are -**

**valM -** Value read from memory

### Implementation Of Memory Stage -

| OPq | Memory | | |
|---|---|---|---|
| rmmovq | Memory | $M_8$[valE] ← valA | Write value to memory |
| mrmovq | Memory | valM ← $M_8$[valE] | Read value from memory |
| irmovq | Memory | | |
| pushq | Memory | $M_8$[valE] ← valA | Write to stack |
| popq | Memory | valM ← $M_8$[valA] | Read from stack |
| cmovXX | Memory | | |
| jXX | Memory | | |
| call | Memory | $M_8$[valE] ← valP | Update stack pointer |
| ret | Memory | valM ← $M_8$[valA] | Update stack pointer |

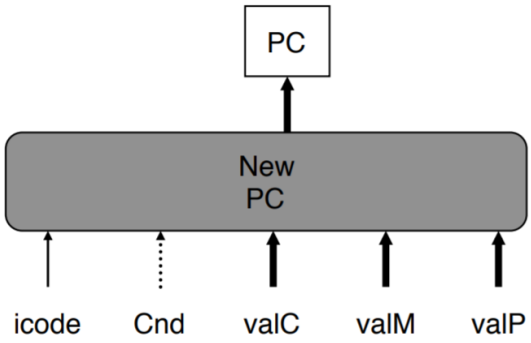
In case of rmovq, call and pushq we write to memory. Whereas, in case of mrmovq, ret and popq we read from memory.

# Y86-64 ISA Implementation

```
module data_memory( clk , icode , valE , valA , valM , valP) ;

// valP is only used in 'call'

parameter memory_size = 4095  ;

  input clk;
  input [3:0] icode;
  input [63:0] valA;
  input [63:0] valE;
  input [63:0] valP;
  output reg [63:0] valM;
  reg [63:0] data_memory[0:memory_size];

  always@(*)
  begin
    valM = 64'd0 ;
    // rmmovq
    if(icode == 4'b0100)
    begin
      data_memory[valE] = valA;
      $display("\nThe memory contained all 3 earlier but now memory[valE] has changed to %d in the following\n",data_memory[valE]);
    end
    //mrmovq
    if(icode == 4'b0101)
    begin
      valM = data_memory[valE] ;
      $display("\nvalM was originally 0, now after shifting from memory to register valM is %d after the following instruction ",valM) ;
    end
    // call
    if(icode == 4'b1000)
    begin
      data_memory[valE] = valP;
      $display("\nThe memory contained all 3 earlier but now memory[valE] has changed to %d after the following instruction\n",data_memory[valE]);
    end
    // ret
    if(icode == 4'b1001)
    begin
      valM = data_memory[valA];
      $display("\nvalM was originally 0, now after shifting from memory to register valM is %d after the following instruction ",valM) ;
    end
    // pushq
    if(icode == 4'b1010)
    begin
      data_memory[valE] = valA;
      $display("\nThe memory contained all 3 earlier but now memory[valE] has changed to %d in the following\n",data_memory[valE]);
    end
    // popq
    if(icode == 4'b1011)
    begin
      valM = data_memory[valE];
      $display("\nvalM was originally 0, now after shifting from memory to register valM is %d after the following instruction",valM) ;
    end
  end
endmodule
```

# Y86-64 ISA Implementation

# PC Update




New value of the PC is taken in one of valC, valM, valP.

**Computed Values in this stage are -**

**PC__Update -** Updated Program Counter

### Implementation Of PC Update Stage -

| | | | |
|---|---|---|---|
| OPq | PC Update | PC ← valP | Update PC |
| rmmovq | PC Update | PC ← valP | Update PC |
| mrmovq | PC Update | PC ← valP | Update PC |
| irmovq | PC Update | PC ← valP | Update PC |
| pushq | PC Update | PC ← valP | Update PC |
| popq | PC Update | PC ← valP | Update PC |
| cmovXX | PC Update | PC ← valP | Update PC |
| jXX | PC Update | PC ← Cnd? valC : valP | Update PC |
| call | PC Update | PC ← valC | Update PC |
| ret | PC Update | PC ← valM | Update PC |

# Y86-64 ISA Implementation

```
1
2   module PC_UPDATE(valP,valC,valM,Cnd,icode,PC,PC__Update);
3     input [63:0] valP; //Incremented PC
4     input [63:0] valC; //Insruction Constant
5     input [63:0] valM; //Value from Memory
6     input Cnd;         //Branch Flag
7     input [3:0] icode; //Instruction Code
8     input [63:0] PC;
9     output reg [63:0] PC__Update;
10
11
12    always@(*) begin
13      if (icode == 4'b0111) begin
14        //jxx - jmp, jle, jl, je, jne, jge, and jg
15        //Program Counter is set to Dest if branch is taken (Takes valC)
16        //Otherwise PC is incremented by 9 (Takes valP)
17
18        if  (Cnd == 1'b1) begin
19          PC__Update = valC;
20        end
21        else begin
22          PC__Update = valP;
23        end
24      end
25
26      if (icode == 4'b1000) begin
27        //call
28        //Program Counter is set to Dest (Takes valC)
29
30        PC__Update = valC;
31      end
32
33      if (icode == 4'b1001) begin
34        //ret
35        //Program Counter is set to return address (Takes valP)
36
37        PC__Update = valM;
38      end
39
40      else begin
41        PC__Update = valP;
42      end
43    end
44  endmodule
```

# Y86-64 ISA Implementation

```
47      .PC(PC),
48      .icode(icode),
49      .ifun(ifun),
50      .rA(rA),
51      .rB(rB),
52      .valC(valC),
53      .valP(valP),
54      .halt_prog(halt_prog),
55      .is_instruction_valid(is_instruction_valid),
56      .pcvalid(pcvalid)
57  );
58
59  Execute execute(
60      .icode(icode),
61      .ifun(ifun),
62      .valA(valA),
63      .valB(valB),
64      .valC(valC),
65      .valE(valE),
66      .clk(clk),
67      .cnd(cnd),
68      .ZF(ZF),
69      .SF(SF),
70      .OF(OF)
71  );
72
73
74  decode_and_writeback decode_and_wb(
75      .valA(valA),
76      .valB(valB),
77      .valE(valE),
78      .valM(valM),
79      .clk(clk),
80      .rA(rA),
81      .rB(rB),
82      .icode(icode),
83      .cnd(cnd),
84      .register_memory0(register_memory0),
85      .register_memory1(register_memory1),
86      .register_memory2(register_memory2),
87      .register_memory3(register_memory3),
88      .register_memory4(register_memory4),
89      .register_memory5(register_memory5),
90      .register_memory6(register_memory6),
91      .register_memory7(register_memory7),
92      .register_memory8(register_memory8),
```

# Y86-64 ISA Implementation

```
93          .register_memory9(register_memory9),
94          .register_memory10(register_memory10),
95          .register_memory11(register_memory11),
96          .register_memory12(register_memory12),
97          .register_memory13(register_memory13),
98          .register_memory14(register_memory14)
99      );
100
101     data_memory Memory(
102         .clk(clk),
103         .icode(icode),
104         .valE(valE),
105         .valA(valA),
106         .valM(valM),
107         .valP(valP)
108     );
109
110     PC_UPDATE PCUPDATE(
111         .valP(valP),
112         .valC(valC),
113         .valM(valM),
114         .Cnd(Cnd),
115         .icode(icode),
116         .PC(PC),
117         .PC__Update(PC__Update)
118     );
119
120     initial begin
121             clk=0;
122
123
124         #3 clk=~clk;PC=64'd0;
125         #3 clk=~clk;
126         #3 clk=~clk;PC=valP;
127         #3 clk=~clk;
128         #3 clk=~clk;PC=valP;
129         #3 clk=~clk;
130         #3 clk=~clk;PC=valP;
131         #3 clk=~clk;
132     end
133
134     initial begin
135         //$dumpfile("SEQ.vcd");
136        // $dumpvars(0,SEQ);
137         Arr[0] = 1; //AOK
138         Arr[1] = 0; //INS
```

# Y86-64 ISA Implementation

addition.v > processor

```
1   `include "Fetch.v"
2   `include "execute.v"
3   `include "decode_and_writeback.v"
4   `include "memory.v"
5   `include "PC_UPDATE.v"
6
7   module processor;
8       reg clk;
9       reg [63:0] PC;
10      reg Arr[0:2];
11      wire [3:0] icode;
12      wire [3:0] ifun;
13      wire [3:0] rA;
14      wire [3:0] rB;
15      wire [63:0] valC;
16      wire [63:0] valP;
17      wire halt_prog;
18      wire is_instruction_valid;
19      wire pcvalid;
20      wire cnd;
21      wire [63:0] valA;
22      wire [63:0] valB;
23      wire [63:0] valE;
24      wire [63:0] valM;
25      wire [63:0] register_memory0;
26      wire [63:0] register_memory1;
27      wire [63:0] register_memory2;
28      wire [63:0] register_memory3;
29      wire [63:0] register_memory4;
30      wire [63:0] register_memory5;
31      wire [63:0] register_memory6;
32      wire [63:0] register_memory7;
33      wire [63:0] register_memory8;
34      wire [63:0] register_memory9;
35      wire [63:0] register_memory10;
36      wire [63:0] register_memory11;
37      wire [63:0] register_memory12;
38      wire [63:0] register_memory13;
39      wire [63:0] register_memory14;
40      wire ZF;
41      wire SF;
42      wire OF;
43      wire [63:0] PC__Update;
44
45  Fetch fetch(
46      .clk(clk),
```

SystemVerilog: 10 indexed objects

# Y86-64 ISA Implementation

```
    Arr[2] = 0; //HLT
    clk = 0;
    PC = 64'd48;
end

always @(*) begin
    PC = PC__Update;
end

always @(*) begin
    if (halt_prog) begin
        Arr[0] = 1'b0; //AOK
        Arr[1] = 1'b0; //INS
        Arr[2] = halt_prog; //HLT
    end

    else if (is_instruction_valid) begin
        Arr[0] = 1'b0; //AOK
        Arr[1] = is_instruction_valid; //INS
        Arr[2] = 1'b0; //HLT
    end

    else begin
        Arr[0] = 1'b1; //AOK
        Arr[1] = 1'b0; //INS
        Arr[2] = 1'b0; //HLT
    end

end

always @(*) begin

    if (Arr[2] == 1'b1) begin
        $finish;
    end
end

initial
$monitor("clk = %d icode = %b ifun = %b  rA = %b rB = %b valA = %d valB = %d  valC=%d valE=%d valM=%d Halt_Prog=%d InstructionValid=%d pcvalid=%d cnd=%d\nRegMem0=%d\nRegMem1=%d\nRegMem2=%d\nRegMem3=%d\nRegMe

  //  $monitor("clk=%d icode=%b ifun=%b rA=%b rB=%b valA=%d valB=%d valE=%d\n reg1=%d\n reg2=%d \n reg3=%d \n reg4=%d \n reg5=%d \n reg6=%d \n reg7=%d \n reg8=%d \n reg9=%d \n reg10=%d\n reg11=%d\n reg12=%d\

endmodule
```

# Y86-64 ISA Implementation

**Important points:**

- In the Y86 processor implementation there are 5 stages.
    1. Fetch
    2. Decode
    3. Execute
    4. Memory
    5. Write back
    6. PC update

```
Instruction_memory[0] = 8'b01100000; //6 add
Instruction_memory[1] = 8'b00000011; //%rax %rbx and store in rbx

Instruction_memory[2] = 8'b00100000; // rrmovq
Instruction_memory[3] = 8'b00000011; // src = %rax dest = %rdx



Instruction_memory[4] = 8'b01000000; //4-rmmovq
Instruction_memory[5] = 8'b00000011; //rax and (rbx)
Instruction_memory[6] = 8'b00000000;
Instruction_memory[7] = 8'b00000000;
Instruction_memory[8] = 8'b00000000;
Instruction_memory[9] = 8'b00000000;
Instruction_memory[10] = 8'b00000000;
Instruction_memory[11] = 8'b00000000;
Instruction_memory[12] = 8'b00000000;
Instruction_memory[13] = 8'b00001111;


Instruction_memory[14] = 8'b00010000; //no operation
Instruction_memory[15] = 8'b00010000; //no operation
Instruction_memory[16] = 8'b00000000; //halt
```

# Y86-64 ISA Implementation

```
clk = 1 icode = 0110 ifun = 0000  rA = 0000 rB = 0011 valA =                20 valB =                 1  valC=                0 valE=               21 valM=                0 Halt_Prog=0 In
structionValid=1 pcvalid=0 cnd=0
RegMem0=                  20
RegMem1=                  21
RegMem2=                  54
RegMem3=                   1
RegMem4=                  31
RegMem5=                  63
RegMem6=                  12
RegMem7=                  95
RegMem8=                  72
RegMem9=                  52
RegMem10=                  3
RegMem11=                  8
RegMem12=                  9
RegMem13=                  4
RegMem14=                  6
ZF=0
SF=0
OF=0
clk = 0 icode = 0110 ifun = 0000  rA = 0000 rB = 0011 valA =                20 valB =                21  valC=                0 valE=               21 valM=                0 Halt_Prog=0 In
structionValid=1 pcvalid=0 cnd=0
RegMem0=                  20
RegMem1=                  21
RegMem2=                  54
RegMem3=                  21
RegMem4=                  31
RegMem5=                  63
RegMem6=                  12
RegMem7=                  95
RegMem8=                  72
RegMem9=                  52
RegMem10=                  3
RegMem11=                  8
RegMem12=                  9
RegMem13=                  4
RegMem14=                  6
ZF=0
SF=0
OF=0
```

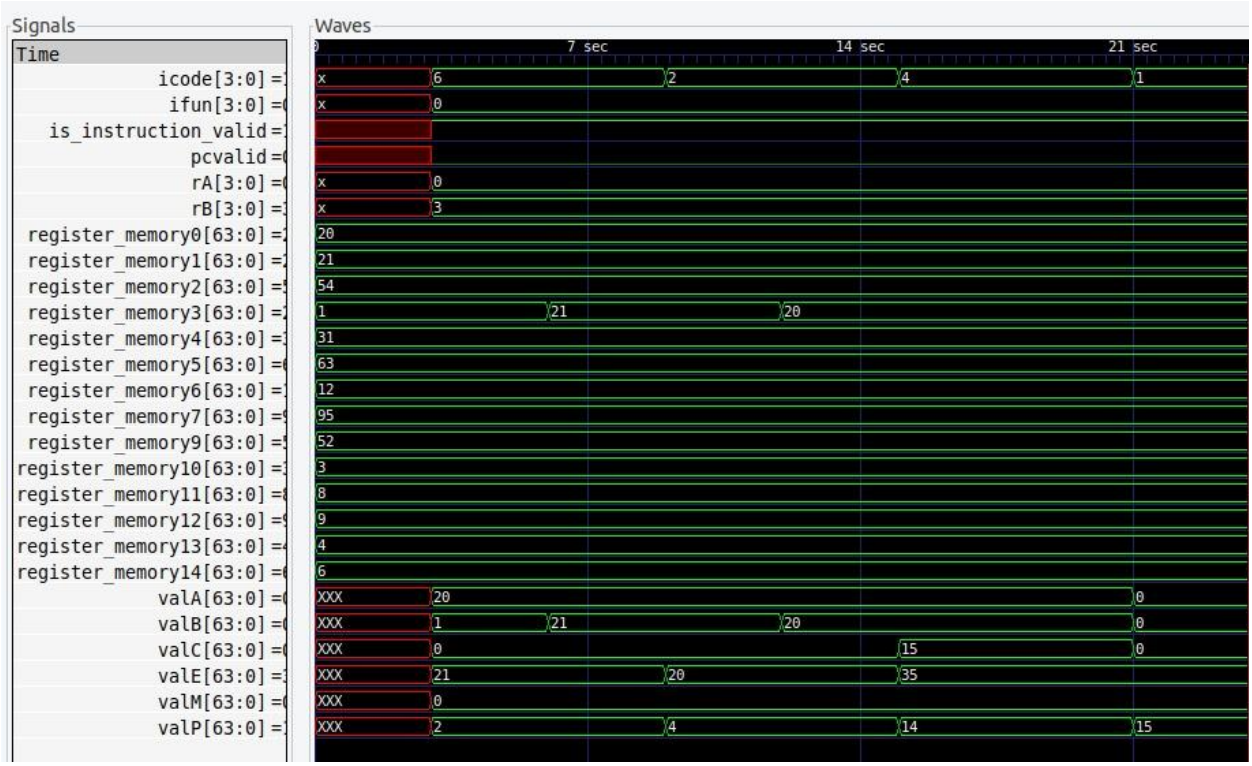
```
clk = 1 icode = 0010 ifun = 0000  rA = 0000 rB = 0011 valA =                20 valB =                21  valC=                0 valE=               20 valM=                0 Halt_Prog=0 In
structionValid=1 pcvalid=0 cnd=1
RegMem0=                  20
RegMem1=                  21
RegMem2=                  54
RegMem3=                  21
RegMem4=                  31
RegMem5=                  63
RegMem6=                  12
RegMem7=                  95
RegMem8=                  72
RegMem9=                  52
RegMem10=                  3
RegMem11=                  8
RegMem12=                  9
RegMem13=                  4
RegMem14=                  6
ZF=0
SF=0
OF=0
clk = 0 icode = 0010 ifun = 0000  rA = 0000 rB = 0011 valA =                20 valB =                20  valC=                0 valE=               20 valM=                0 Halt_Prog=0 In
structionValid=1 pcvalid=0 cnd=1
RegMem0=                  20
RegMem1=                  21
RegMem2=                  54
RegMem3=                  20
RegMem4=                  31
RegMem5=                  63
RegMem6=                  12
RegMem7=                  95
RegMem8=                  72
RegMem9=                  52
RegMem10=                  3
RegMem11=                  8
RegMem12=                  9
RegMem13=                  4
RegMem14=                  6
ZF=0
SF=0
OF=0
```

# Y86-64 ISA Implementation

```
The memory contained all 3 earlier but now memory[valE] has changed to                 20 in the following

clk = 1 icode = 0100 ifun = 0000  rA = 0000 rB = 0011 valA =                 20 valB =                 20  valC=                15 valE=                35 valM=                 0 Halt_Prog=0 In
structionValid=1 pcvalid=0 cnd=0
RegMem0=                  20
RegMem1=                  21
RegMem2=                  54
RegMem3=                  20
RegMem4=                  31
RegMem5=                  63
RegMem6=                  12
RegMem7=                  95
RegMem8=                  72
RegMem9=                  52
RegMem10=                  3
RegMem11=                  8
RegMem12=                  9
RegMem13=                  4
RegMem14=                  6
ZF=0
SF=0
OF=0
clk = 0 icode = 0100 ifun = 0000  rA = 0000 rB = 0011 valA =                 20 valB =                 20  valC=                15 valE=                35 valM=                 0 Halt_Prog=0 In
structionValid=1 pcvalid=0 cnd=0
RegMem0=                  20
RegMem1=                  21
RegMem2=                  54
RegMem3=                  20
RegMem4=                  31
RegMem5=                  63
RegMem6=                  12
RegMem7=                  95
RegMem8=                  72
RegMem9=                  52
RegMem10=                  3
RegMem11=                  8
RegMem12=                  9
RegMem13=                  4
RegMem14=                  6
ZF=0
SF=0
OF=0
```

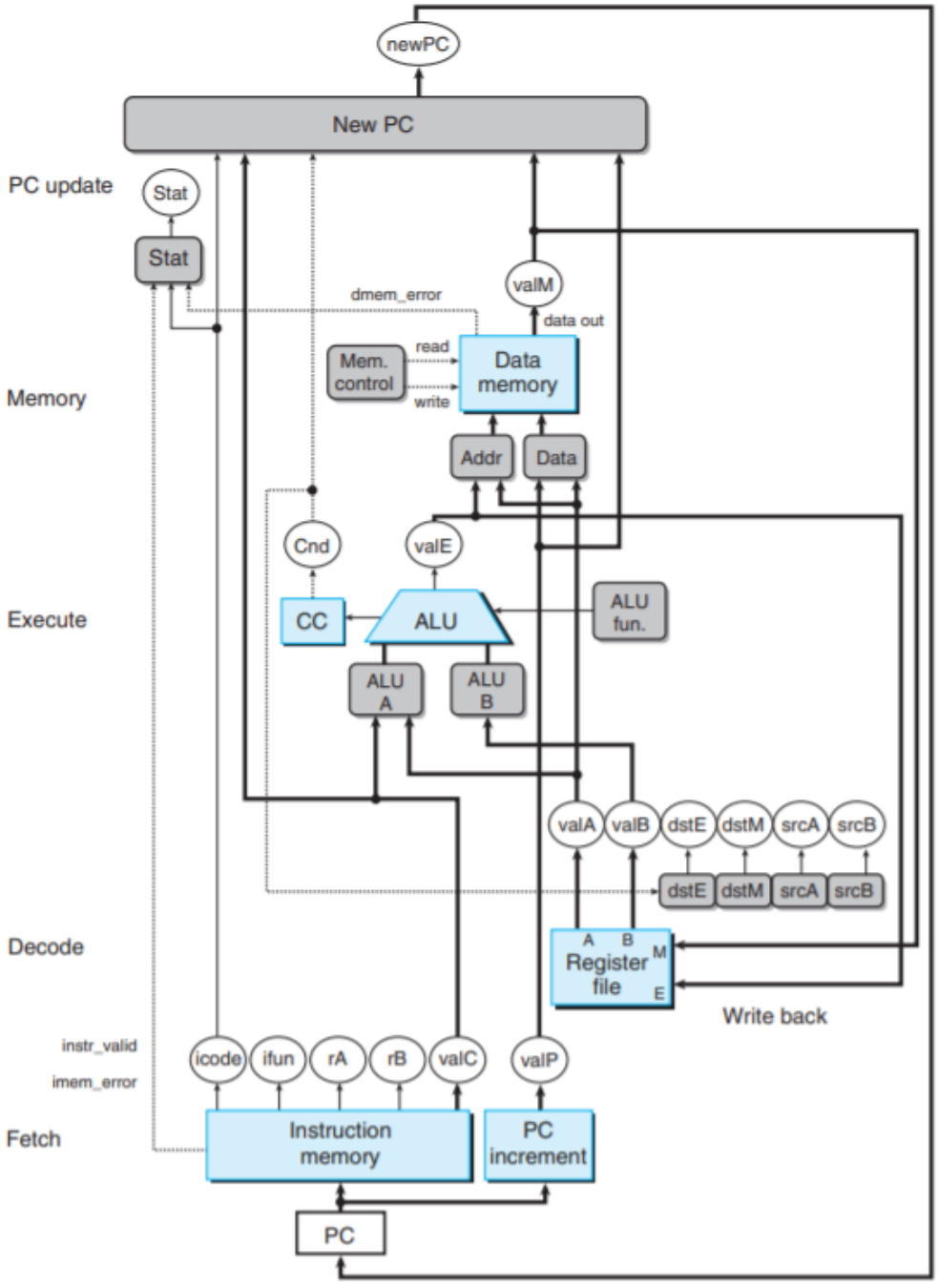
```
clk = 1 icode = 0001 ifun = 0000  rA = 0000 rB = 0011 valA =                  0 valB =                  0  valC=                 0 valE=                35 valM=                 0 Halt_Prog=0 In
structionValid=1 pcvalid=0 cnd=0
RegMem0=                  20
RegMem1=                  21
RegMem2=                  54
RegMem3=                  20
RegMem4=                  31
RegMem5=                  63
RegMem6=                  12
RegMem7=                  95
RegMem8=                  72
RegMem9=                  52
RegMem10=                  3
RegMem11=                  8
RegMem12=                  9
RegMem13=                  4
RegMem14=                  6
ZF=0
SF=0
OF=0
clk = 0 icode = 0001 ifun = 0000  rA = 0000 rB = 0011 valA =                  0 valB =                  0  valC=                 0 valE=                35 valM=                 0 Halt_Prog=0 In
structionValid=1 pcvalid=0 cnd=0
RegMem0=                  20
RegMem1=                  21
RegMem2=                  54
RegMem3=                  20
RegMem4=                  31
RegMem5=                  63
RegMem6=                  12
RegMem7=                  95
RegMem8=                  72
RegMem9=                  52
RegMem10=                  3
RegMem11=                  8
RegMem12=                  9
RegMem13=                  4
RegMem14=                  6
ZF=0
SF=0
OF=0
```

# Y86-64 ISA Implementation



# Y86-64 ISA Implementation

## Hardware Structure Of SEQ Implementation -



**Y86-64 ISA Implementation**

# Y86-64 Pipeline Implementation-

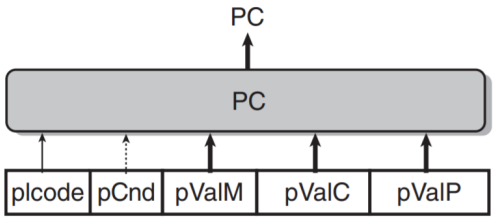
The term Pipelining refers to a technique of decomposing a sequential process into sub-operations, with each sub-operation being executed in a dedicated segment that operates concurrently with all other segments.

A key feature of pipelining is that it increases the throughput of the system. This is the simplest technique for improving performance through hardware parallelism with smaller cycles time.

## Steps to design a pipeline –

### 1) Rearranging the computation stage-

- As a transitional step toward a pipelined design, we must slightly rearrange the order of the five stages in SEQ so that the PC update stage comes at the beginning of the clock cycle, rather than at the end.
- This step is also called circuit retiming as we can continuously fetch the next instruction without having to wait for the PC Update stage of the previous instruction.
- Retiming changes the state representation for a system without changing its logical behavior. It is often used to balance the delays between the different stages of a pipelined system.



# Y86-64 ISA Implementation

## 2) Inserting Pipeline Registers-

- Second step of creating a pipelined Y86-64 processor is inserting pipeline registers.
- We insert pipeline registers between each stage and rearrange signals.

**Pipeline registers are labeled as follows -**

**F ->** Holds the 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.

## 3) Rearranging and Relabeling Signals-

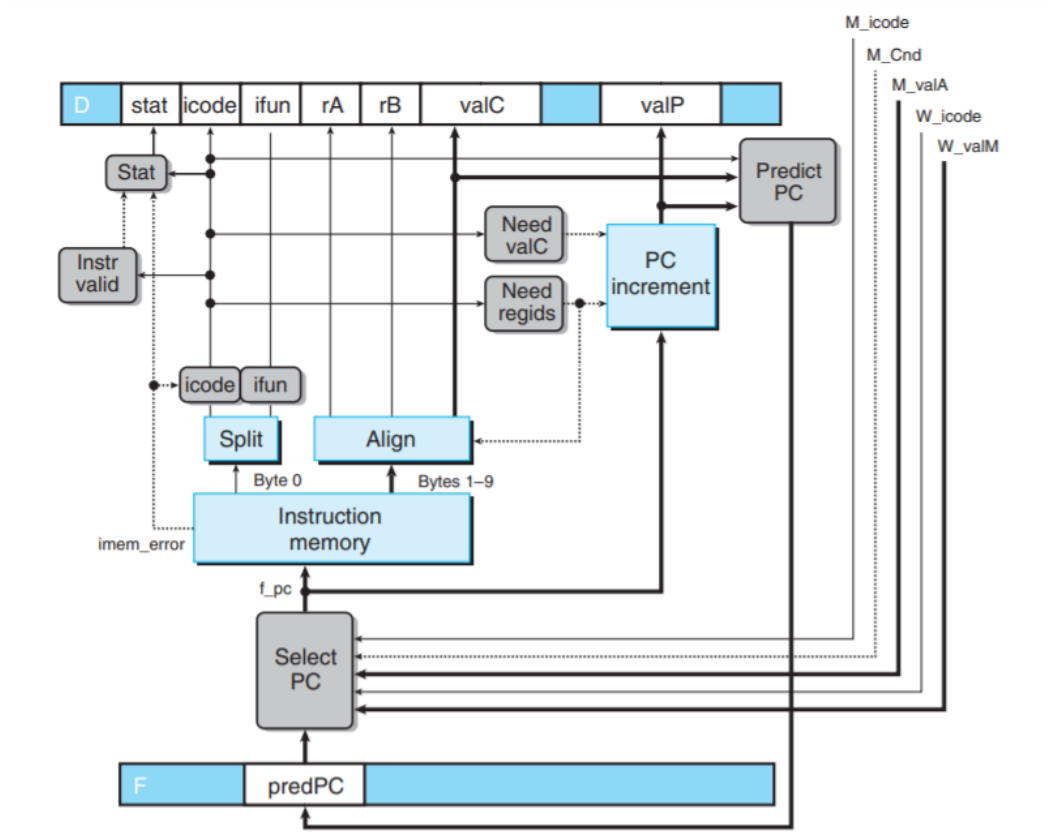
- In this type of implementation, signals pass through every stage one by one.
- We adopt a naming scheme where a signal stored in a pipeline register can be uniquely identified by prefixing its name with that of the pipe register written in uppercase.
- Signals that have just been computed within a stage are labeled by prefixing the signal name with the first character of the stage name, written in lowercase.

# Y86-64 ISA Implementation

## Architecture Diagram -

# PC Selection and Fetch Stage

- Select current PC
- Read instruction
- Compute incremented PC



# Y86-64 ISA Implementation

## PC selection Logic -

- The Program Counter, or PC, is a register that holds the address that is presented to the instruction memory. At the start of a cycle, the address is presented to instruction memory. Then during the cycle, the instruction is being read out of instruction memory, and at the same time a calculation is done to determine the next PC.
- As a mispredicted branch enters the memory stage, the value of valP for this instruction (indicating the address of the following instruction) is read from pipeline register M (signal M_valA).
- When a ret instruction enters the write-back stage, the return address is read from pipeline register W (signal W_valM).
- All other cases use the predicted value of the PC, stored in pipeline register F (signal F_predPC) -

```
word f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        M_icode == IJXX && !M_Cnd : M_valA;
        # Completion of RET instruction
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];
```

- The PC prediction logic chooses valC for the fetched instruction when it is either a call or a jump, and valP otherwise:

```
 word f_predPC = [
         f_icode in { IJXX, ICALL } : f_valC;
         1 : f_valP;
 ];
```

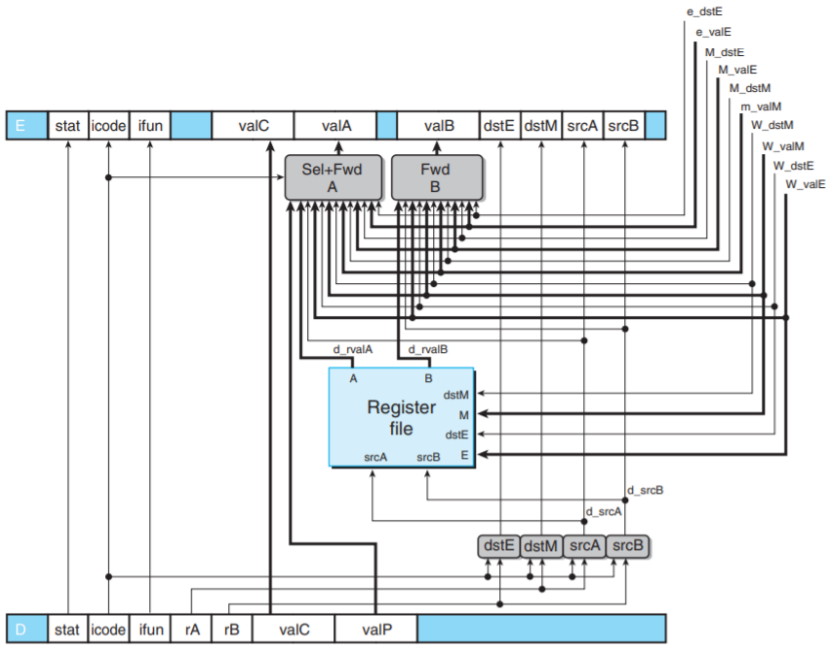
- Unlike in SEQ, we must split the computation of the instruction status into two parts. In the fetch stage, we can test for a memory error due to an out-of-range instruction address, and we can detect an illegal instruction or a halt instruction. Detecting an invalid data address must be deferred to the memory stage.

**Y86-64 ISA Implementation**

# Decode and Write-Back Stages

◦ Read program registers

◦ Update register file




- Register has four ports in which two are read ports and two are write ports.
- It supports two simultaneous reads and two simultaneous writes.
- The two read ports have address inputs srcA and srcB and the two write ports have address inputs dstE and dstM.

**srcA** - Indicate which register should be read to generate valA.
**srcB** - Indicate which register should be read to generate valB
**dstE** - Indicate the destination register for write port E where valE is stored.
**dstM** - Indicate the destination register for write port M where valM is stored.

- These four blocks dstE, dstM, srcA, srcB, generate the four different register IDs for the register file, based on the instruction code icode,

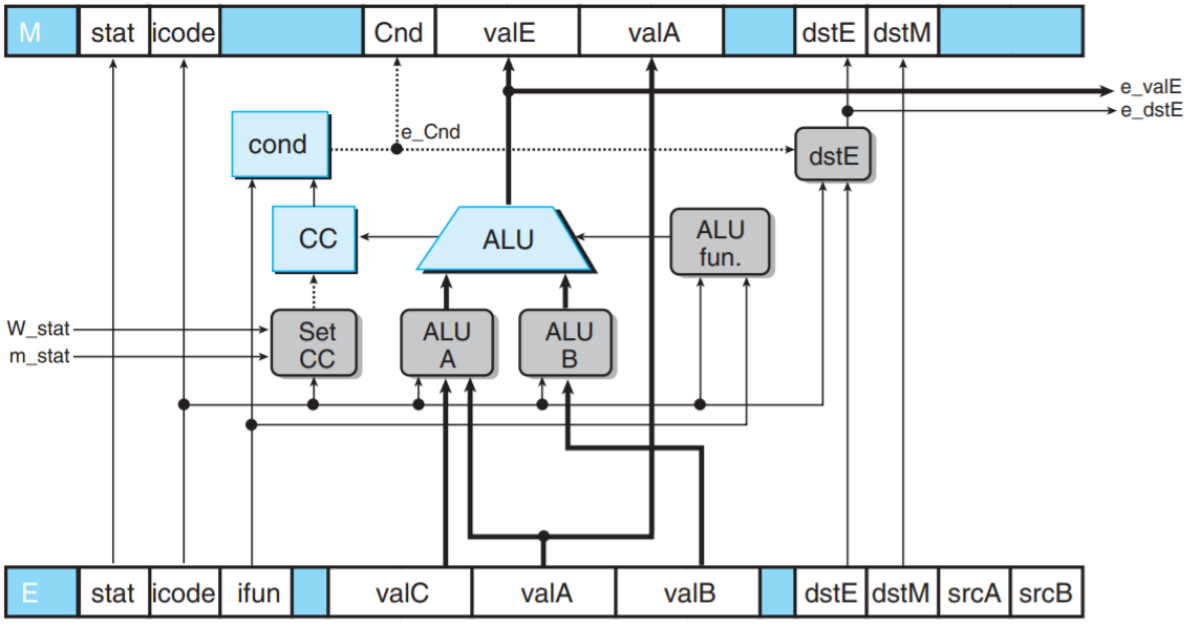
the register specifiers rA and rB, and possibly the condition signal Cnd computed in the execute stage.

- Data forwarding takes place in this stage.
- Block “Sel+Fwd A” merges valP into valA for later stages in order to reduce the amount of state in the pipeline register as only call and jump instructions need valP in further stages instead of valA.
- This block also implements the forwarding logic for source operand valA.
- Block “Fwd B” implements the forwarding logic for source operand valB.
- We also introduce a status register “stat” to indicate whether the program is executing normally or an exception occurred.
  This is needed as the code should indicate either AOK or one of the three exception conditions. Exceptional conditions include when an
  Invalid instruction is fetched, or a halt instruction is executed.
- Consider bubble in the Write-Back stage as AOK.

# Y86-64 ISA Implementation

# Execute

◦ Operate ALU

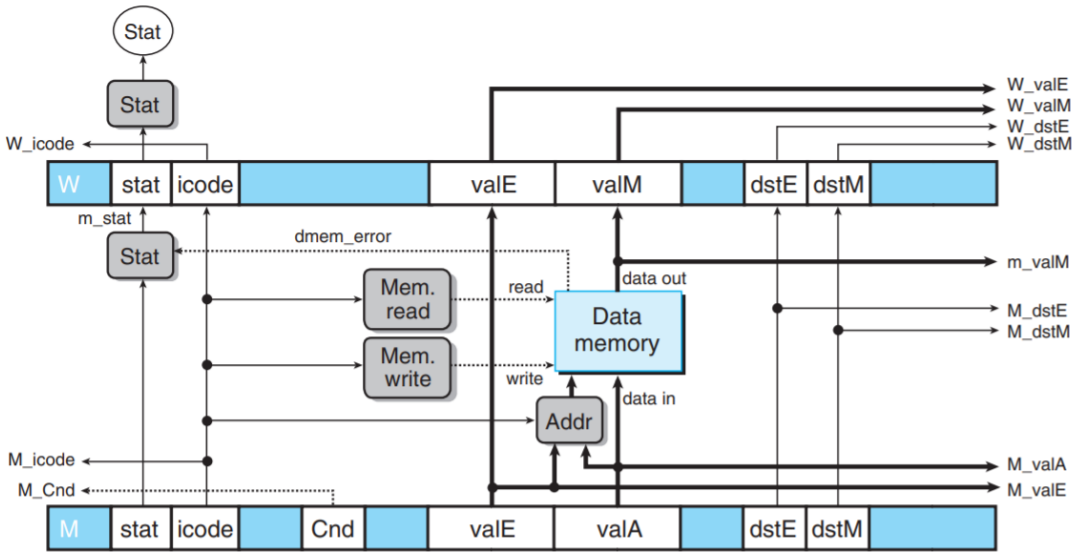



- Pipeline implementation of execute stage is similar to the sequential implementation.
- In pipeline implementation, the logic “Set CC” has signals m_stat and W_stat as inputs.
- The signals e_valE and e_dstE are directed towards the decode stage as forwarding sources.

# Y86-64 ISA Implementation

# Memory

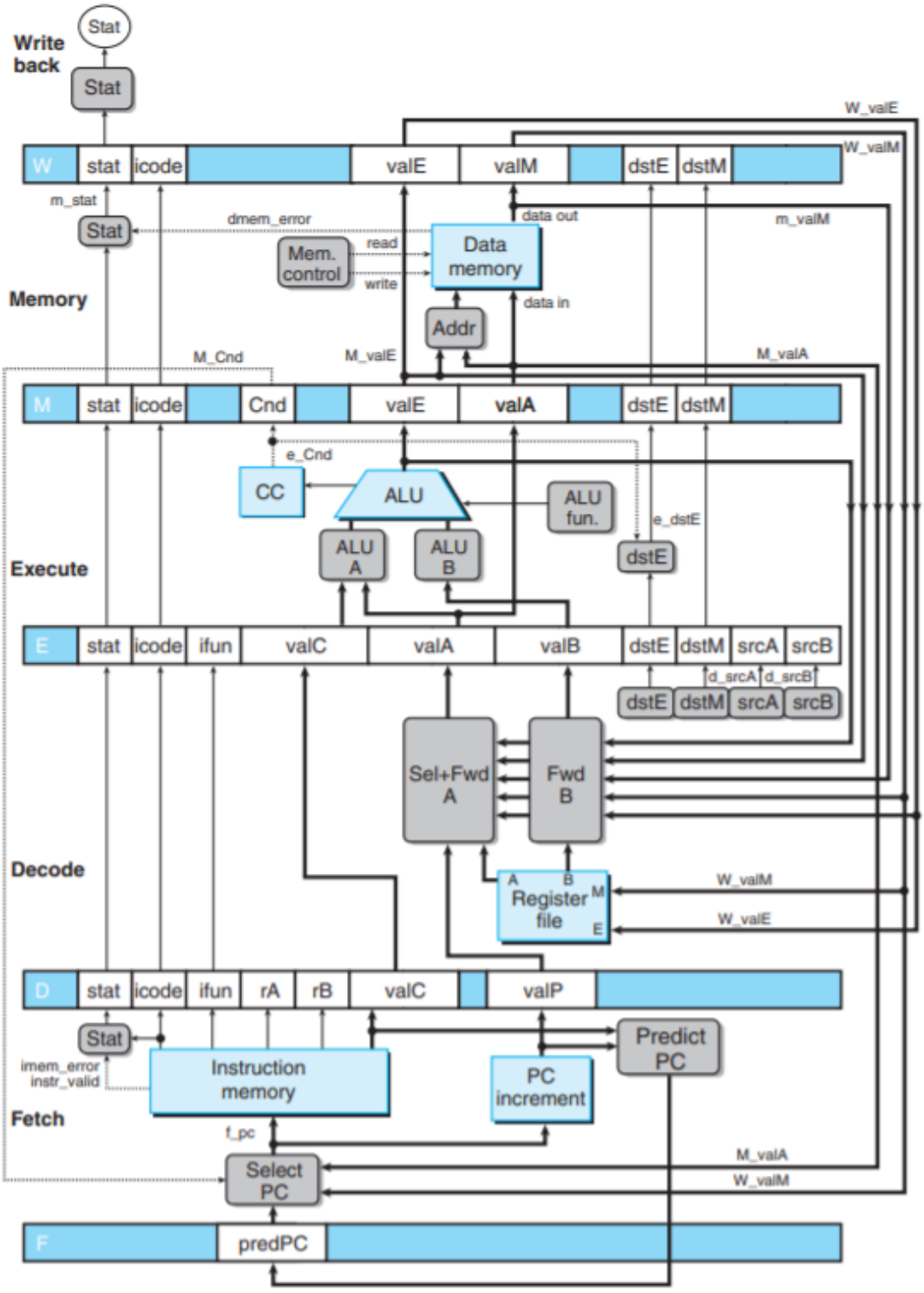
◦ Read or write data memory




- Memory block either reads or writes the program data.
- Memory stage in pipeline lacks “Mem.data” block present in SEQ as the task is performed by “Sel+Fwd A” block in decode stage.

# Y86-64 ISA Implementation

## Overall implementation of Y86-64 processor (5 Stage Pipeline)



# Y86-64 ISA Implementation

## Data Forwarding-

- ➢ In Naïve Pipeline, Register isn't written until completion of write-back stage and Source operands read from register file in decode stage.
- ➢ In data forwarding, we take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units that need it that cycle.
- ➢ In case of multiple forwarding choices, use matching value from the earliest pipeline stage.

### Implementation-

◦ Add additional feedback paths from E, M, and W pipeline registers into decode stage

◦ Create logic blocks to select from multiple sources for valA and valB in decode stage

◦ Forwarding Sources -

| Data word | Register ID | Source description |
|---|---|---|
| e_valE | e_dstE | ALU output |
| m_valM | M_dstM | Memory output |
| M_valE | M_dstE | Pending write to port E in memory stage |
| W_valM | W_dstM | Pending write to port M in write-back stage |
| W_valE | W_dstE | Pending write to port E in write-back stage |

# Y86-64 ISA Implementation

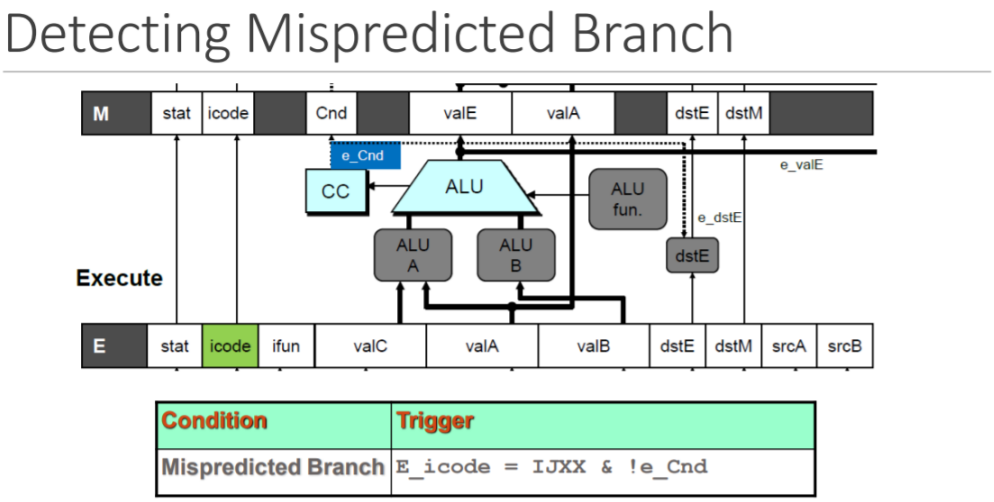
## What should be the A value?

```
int d_valA = [
  # Use incremented PC
     D_icode in { ICALL, IJXX } : D_valP;
  # Forward valE from execute
     d_srcA == e_dstE : e_valE;
  # Forward valM from memory
     d_srcA == M_dstM : m_valM;
  # Forward valE from memory
     d_srcA == M_dstE : M_valE;
  # Forward valM from write back
      d_srcA == W_dstM : W_valM;
  # Forward valE from write back
     d_srcA == W_dstE : W_valE;
  # Use value read from register file
     1 : d_rvalA;
];
```

# Y86-64 ISA Implementation

## Branch Misprediction Case -

Branch misprediction occurs mainly in the case of jump (jXX).
A misprediction can incur a serious penalty causing a serious degradation of program performance.



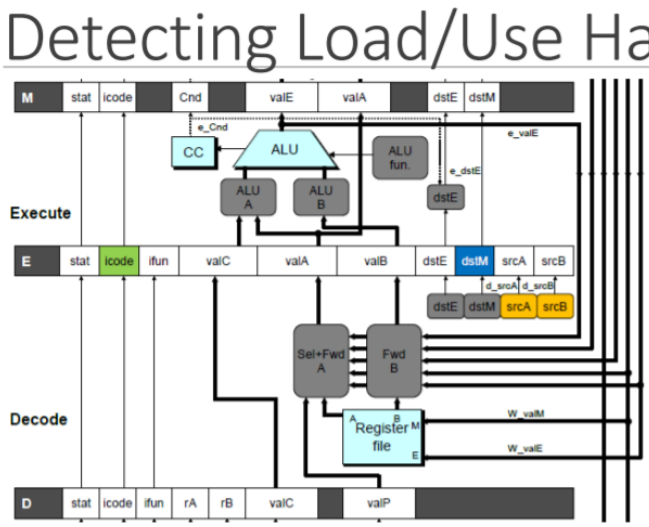

### Handling Misprediction -

- Fetch 2 instructions at the target where branch is taken
- In execute stage, detect whether branch is taken or not, cancel When mispredicted.
- For no side effects, on the following cycle, replace instructions in execute and decode bubbles.

# Y86-64 ISA Implementation

## Load/Use Hazard Case -

➔ A load-use hazard requires delaying the execution of the using instruction until the result from the loading instruction can be made available to the using instruction.




| Condition | Trigger |
|---|---|
| Load/Use Hazard | `E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }` |

### Control for Load/Use Hazard -

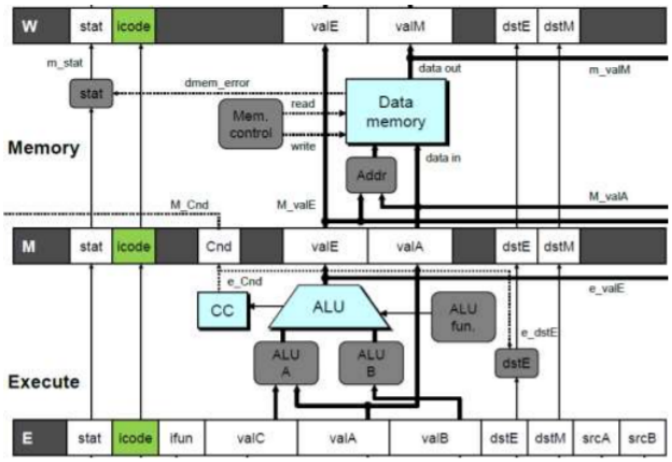
◦ Stall instructions in fetch and decode stages
◦ Inject bubble into execute stage

# Y86-64 ISA Implementation

## Return Condition -

➔ For the return condition to be implemented, the return point should be known.
➔ Before the instruction's return point is executed, next instructions are fetched in between which should not be executed.
➔ Return point is known in the memory stage of the return instruction. Hence we need to handle this special control case.

## Detecting Return




| Condition | Trigger |
|---|---|
| Processing `ret` | `IRET in { D_icode, E_icode, M_icode }` |

## Handling ret case -

-> As ret passes through the pipeline, stall at the fetch stage.
-> Inject bubble into the decode stage.
-> Release stall when reach write-back stage.

# Y86-64 ISA Implementation

**Fot the given Instructions set the OUTPUTS:**

```
  Instruction_memory[0] = 8'b01000000;
  Instruction_memory[1] = 8'b00000111;
  Instruction_memory[2] = 8'b00000001;
  Instruction_memory[3] = 8'b00000011;
  Instruction_memory[4] = 8'b00000100;
  Instruction_memory[5] = 8'b00000011;
  Instruction_memory[6] = 8'b00000000;
  Instruction_memory[7] = 8'b00000000;
  Instruction_memory[8] = 8'b00000000;
  Instruction_memory[9] = 8'b00000000;

  Instruction_memory[10] = 8'b00010000;
  Instruction_memory[11] = 8'b00010000;

  Instruction_memory[12] = 8'b00100000;
  Instruction_memory[13] = 8'b01100000;
  Instruction_memory[14] = 8'b00010000;
```

# Y86-64 ISA Implementation

```
  Instruction_memory[15] = 8'b00010000;
  Instruction_memory[16] = 8'b00010000;

  Instruction_memory[17] = 8'b00110000;
  Instruction_memory[18] = 8'b00000111;
  Instruction_memory[19] = 8'b00000001;
  Instruction_memory[20] = 8'b00000011;
  Instruction_memory[21] = 8'b00000100;
  Instruction_memory[22] = 8'b00000011;
  Instruction_memory[23] = 8'b00000000;
  Instruction_memory[24] = 8'b00000000;
  Instruction_memory[25] = 8'b00000000;
  Instruction_memory[26] = 8'b00000000;

  Instruction_memory[27] = 8'b01100000;
  Instruction_memory[28] = 8'b00100010;

  Instruction_memory[29] = 8'b00010000;

  Instruction_memory[30] = 8'b00010000;
  Instruction_memory[31] = 8'b00000000;
```

# Y86-64 ISA Implementation

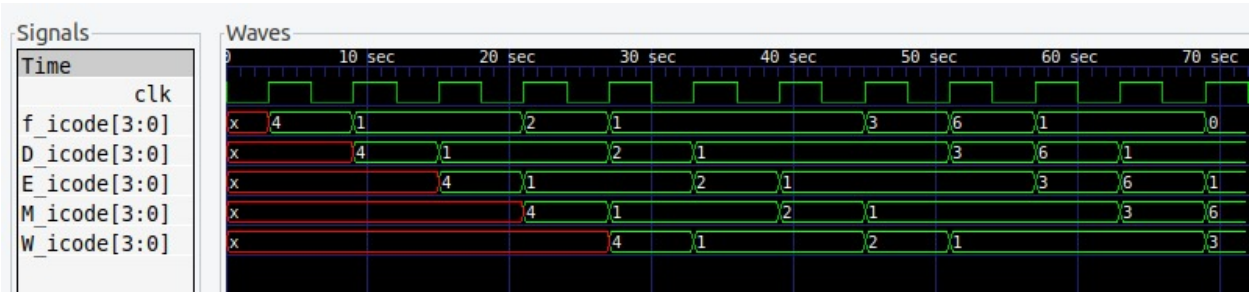
```
clk = 0 , f_icode=xxxx , f_ifun=xxxx , rA= x , rB= x
D_stat=x , D_icode=xxxx , D_ifun=xxxx , D_valC=                  x
E_stat=x , E_icode=xxxx , E_ifun=xxxx , E_valA=                  x , E_valB=                  x , E_valC=                  x
M_stat=x , M_icode=xxxx , M_valA=                  x
W_stat=x , W_icode=xxxx , W_valM=                  x

clk = 1 , f_icode=0100 , f_ifun=0000 , rA= 0 , rB= 7
D_stat=0 , D_icode=xxxx , D_ifun=xxxx , D_valC=                  0
E_stat=x , E_icode=xxxx , E_ifun=xxxx , E_valA=                  x , E_valB=                  x , E_valC=                  x
M_stat=x , M_icode=xxxx , M_valA=                  x
W_stat=x , W_icode=xxxx , W_valM=                  x

clk = 0 , f_icode=0100 , f_ifun=0000 , rA= 0 , rB= 7
D_stat=0 , D_icode=xxxx , D_ifun=xxxx , D_valC=                  0
E_stat=x , E_icode=xxxx , E_ifun=xxxx , E_valA=                  x , E_valB=                  x , E_valC=                  x
M_stat=x , M_icode=xxxx , M_valA=                  x
W_stat=x , W_icode=xxxx , W_valM=                  x

clk = 1 , f_icode=0001 , f_ifun=0000 , rA= 0 , rB= 7
D_stat=1 , D_icode=0100 , D_ifun=0000 , D_valC=   72906429899472896
E_stat=0 , E_icode=xxxx , E_ifun=xxxx , E_valA=                  x , E_valB=                  x , E_valC=                  0
M_stat=x , M_icode=xxxx , M_valA=                  x
W_stat=x , W_icode=xxxx , W_valM=                  x

clk = 0 , f_icode=0001 , f_ifun=0000 , rA= 0 , rB= 7
D_stat=1 , D_icode=0100 , D_ifun=0000 , D_valC=   72906429899472896
E_stat=0 , E_icode=xxxx , E_ifun=xxxx , E_valA=                  x , E_valB=                  x , E_valC=                  0
M_stat=x , M_icode=xxxx , M_valA=                  x
W_stat=x , W_icode=xxxx , W_valM=                  x

clk = 1 , f_icode=0001 , f_ifun=0000 , rA= 0 , rB= 7
D_stat=1 , D_icode=0001 , D_ifun=0000 , D_valC=                  0
E_stat=1 , E_icode=0100 , E_ifun=0000 , E_valA=                  0 , E_valB=                  0 , E_valC=   72906429899472896
M_stat=0 , M_icode=xxxx , M_valA=                  x
W_stat=x , W_icode=xxxx , W_valM=                  x

clk = 0 , f_icode=0001 , f_ifun=0000 , rA= 0 , rB= 7
D_stat=1 , D_icode=0001 , D_ifun=0000 , D_valC=                  0
E_stat=1 , E_icode=0100 , E_ifun=0000 , E_valA=                  0 , E_valB=                  0 , E_valC=   72906429899472896
M_stat=0 , M_icode=xxxx , M_valA=                  x
W_stat=x , W_icode=xxxx , W_valM=                  x
```

# Y86-64 ISA Implementation

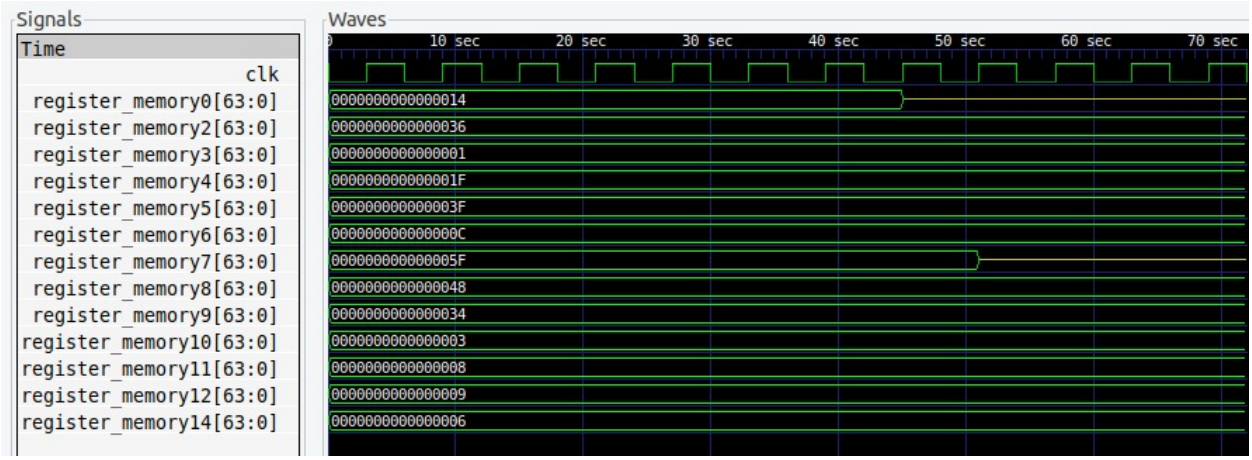
```
The memory contained all 3 earlier but now memory[M_valE] has changed to                  0 in the following

 clk = 1 , f_icode=0010 , f_ifun=0000 , rA= 6 , rB= 0
 D_stat=1 , D_icode=0001 , D_ifun=0000 , D_valC=                  0
 E_stat=1 , E_icode=0001 , E_ifun=0000 , E_valA=                  0 , E_valB=                  0 , E_valC=                  0
 M_stat=1 , M_icode=0100 , M_valA=                  0
 W_stat=0 , W_icode=xxxx , W_valM=                  x

 clk = 0 , f_icode=0010 , f_ifun=0000 , rA= 6 , rB= 0
 D_stat=1 , D_icode=0001 , D_ifun=0000 , D_valC=                  0
 E_stat=1 , E_icode=0001 , E_ifun=0000 , E_valA=                  0 , E_valB=                  0 , E_valC=                  0
 M_stat=1 , M_icode=0100 , M_valA=                  0
 W_stat=0 , W_icode=xxxx , W_valM=                  x

 clk = 1 , f_icode=0001 , f_ifun=0000 , rA= 6 , rB= 0
 D_stat=1 , D_icode=0010 , D_ifun=0000 , D_valC=                  0
 E_stat=1 , E_icode=0001 , E_ifun=0000 , E_valA=                 20 , E_valB=                 95 , E_valC=                  0
 M_stat=1 , M_icode=0001 , M_valA=                  0
 W_stat=1 , W_icode=0100 , W_valM=                  0

 clk = 0 , f_icode=0001 , f_ifun=0000 , rA= 6 , rB= 0
 D_stat=1 , D_icode=0010 , D_ifun=0000 , D_valC=                  0
 E_stat=1 , E_icode=0001 , E_ifun=0000 , E_valA=                 20 , E_valB=                 95 , E_valC=                  0
 M_stat=1 , M_icode=0001 , M_valA=                  0
 W_stat=1 , W_icode=0100 , W_valM=                  0

 clk = 1 , f_icode=0001 , f_ifun=0000 , rA= 6 , rB= 0
 D_stat=1 , D_icode=0001 , D_ifun=0000 , D_valC=                  0
 E_stat=1 , E_icode=0010 , E_ifun=0000 , E_valA=                  0 , E_valB=                  0 , E_valC=                  0
 M_stat=1 , M_icode=0001 , M_valA=                 20
 W_stat=1 , W_icode=0001 , W_valM=                  0

 clk = 0 , f_icode=0001 , f_ifun=0000 , rA= 6 , rB= 0
 D_stat=1 , D_icode=0001 , D_ifun=0000 , D_valC=                  0
 E_stat=1 , E_icode=0010 , E_ifun=0000 , E_valA=                  0 , E_valB=                  0 , E_valC=                  0
 M_stat=1 , M_icode=0001 , M_valA=                 20
 W_stat=1 , W_icode=0001 , W_valM=                  0

 clk = 1 , f_icode=0001 , f_ifun=0000 , rA= 6 , rB= 0
 D_stat=1 , D_icode=0001 , D_ifun=0000 , D_valC=                  0
 E_stat=1 , E_icode=0001 , E_ifun=0000 , E_valA=                  0 , E_valB=                  0 , E_valC=                  0
 M_stat=1 , M_icode=0010 , M_valA=                  0
 W_stat=1 , W_icode=0001 , W_valM=                  0

 clk = 0 , f_icode=0001 , f_ifun=0000 , rA= 6 , rB= 0
 D_stat=1 , D_icode=0001 , D_ifun=0000 , D_valC=                  0
 E_stat=1 , E_icode=0001 , E_ifun=0000 , E_valA=                  0 , E_valB=                  0 , E_valC=                  0
 M_stat=1 , M_icode=0010 , M_valA=                  0
 W_stat=1 , W_icode=0001 , W_valM=                  0
```

# Y86-64 ISA Implementation

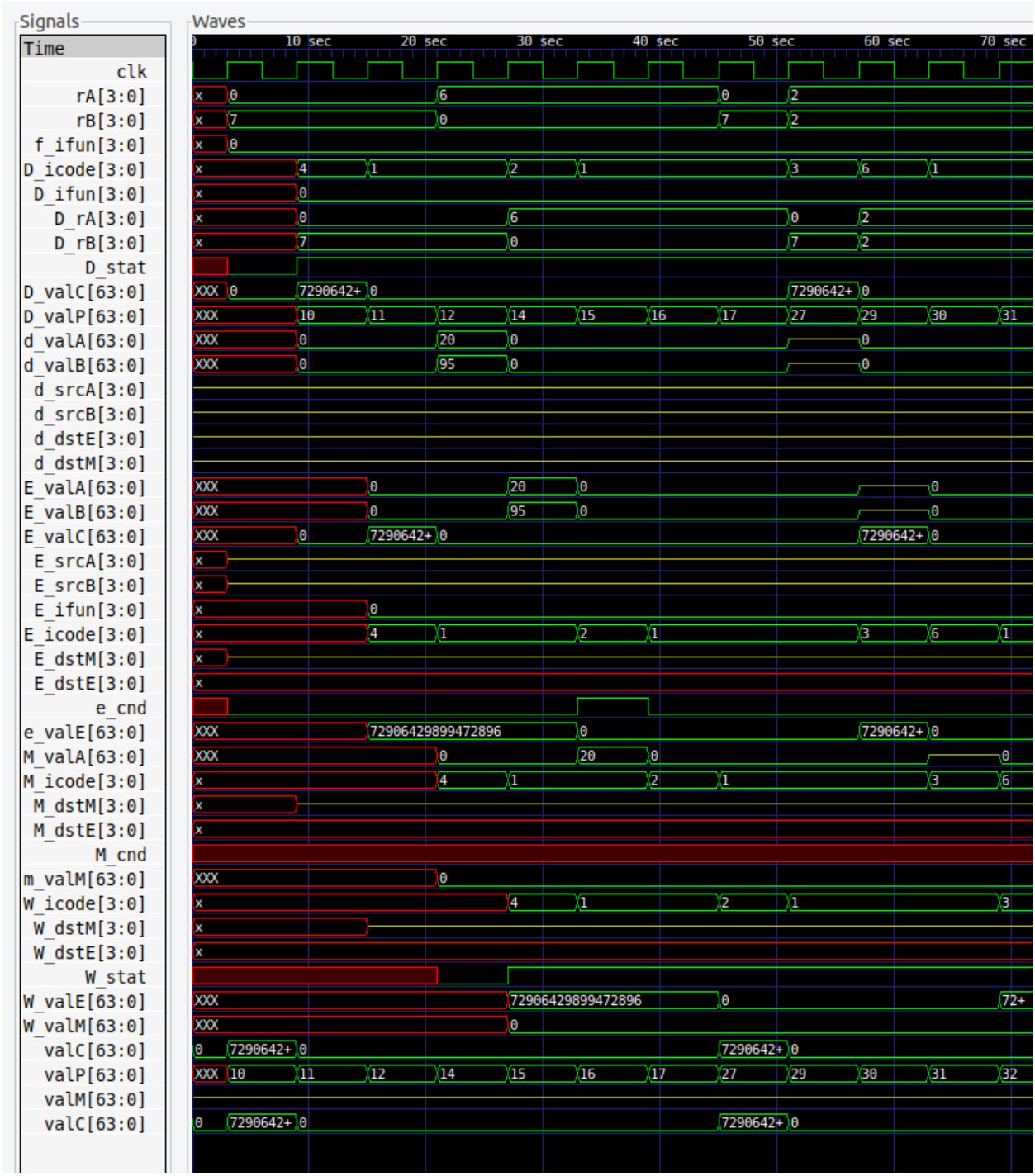
```
clk = 1 , f_icode=0011 , f_ifun=0000 , rA= 0 , rB= 7
D_stat=1 , D_icode=0001 , D_ifun=0000 , D_valC=                    0
E_stat=1 , E_icode=0001 , E_ifun=0000 , E_valA=                    0 , E_valB=                    0 , E_valC=                    0
M_stat=1 , M_icode=0001 , M_valA=                    0
W_stat=1 , W_icode=0010 , W_valM=                    0

clk = 0 , f_icode=0011 , f_ifun=0000 , rA= 0 , rB= 7
D_stat=1 , D_icode=0001 , D_ifun=0000 , D_valC=                    0
E_stat=1 , E_icode=0001 , E_ifun=0000 , E_valA=                    0 , E_valB=                    0 , E_valC=                    0
M_stat=1 , M_icode=0001 , M_valA=                    0
W_stat=1 , W_icode=0010 , W_valM=                    0

clk = 1 , f_icode=0110 , f_ifun=0000 , rA= 2 , rB= 2
D_stat=1 , D_icode=0011 , D_ifun=0000 , D_valC=   72906429899472896
E_stat=1 , E_icode=0001 , E_ifun=0000 , E_valA=                    0 , E_valB=                    0 , E_valC=                    0
M_stat=1 , M_icode=0001 , M_valA=                    0
W_stat=1 , W_icode=0001 , W_valM=                    0

clk = 0 , f_icode=0110 , f_ifun=0000 , rA= 2 , rB= 2
D_stat=1 , D_icode=0011 , D_ifun=0000 , D_valC=   72906429899472896
E_stat=1 , E_icode=0001 , E_ifun=0000 , E_valA=                    0 , E_valB=                    0 , E_valC=                    0
M_stat=1 , M_icode=0001 , M_valA=                    0
W_stat=1 , W_icode=0001 , W_valM=                    0

clk = 1 , f_icode=0001 , f_ifun=0000 , rA= 2 , rB= 2
D_stat=1 , D_icode=0110 , D_ifun=0000 , D_valC=                    0
E_stat=1 , E_icode=0011 , E_ifun=0000 , E_valA=                    z , E_valB=                    z , E_valC=   72906429899472896
M_stat=1 , M_icode=0001 , M_valA=                    0
W_stat=1 , W_icode=0001 , W_valM=                    0

clk = 0 , f_icode=0001 , f_ifun=0000 , rA= 2 , rB= 2
D_stat=1 , D_icode=0110 , D_ifun=0000 , D_valC=                    0
E_stat=1 , E_icode=0011 , E_ifun=0000 , E_valA=                    z , E_valB=                    z , E_valC=   72906429899472896
M_stat=1 , M_icode=0001 , M_valA=                    0
W_stat=1 , W_icode=0001 , W_valM=                    0
```



# Y86-64 ISA Implementation



# Y86-64 ISA Implementation



# Y86-64 ISA Implementation

## Challenges Faced -

-> We faced difficulty in implementing data forwarding.
-> We also faced difficulty in implementing stalls and bubbles.
-> Initially we took time to understand how the 5 stages of the pipeline are implemented in a single clock cycle.
-> It also took time while transitioning from sequential to pipeline implementation.

## Acknowledgment -

Working on this project is interesting and provided us with a great learning experience. Over the last month, we have learnt a lot about Processor Architecture. However, it would not have been possible without the kind support and help of our TA Amar Gwari. He never hesitated to reply to our messages and correct our mistakes. He also provided us with additional sources . We are highly indebted to him.

We would also like to express our gratitude towards Prof. Deepak Gangadharan for his support in completing the project.

- Keerthi Pothalaraju
  Rohan Eswara