Y86-64 ISA

Tanish Taneja

2021112011

Processor Frequency - 1GHz (Clock Frequency = 5ns)

Instruction Memory - 1Kb (8192 Bits = 1024 Bytes)

Data Memory - 16Kb (131072 Bits = 16384 Bytes)

Instructions

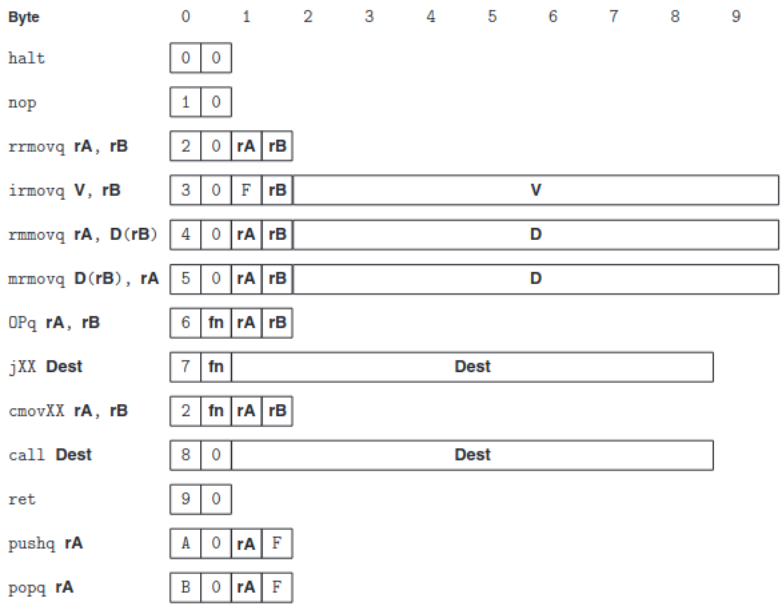
- Y86-64 Instructions can have a length of 1-10 bytes.
- Byte 0 is divided into 2 4-bit blocks :

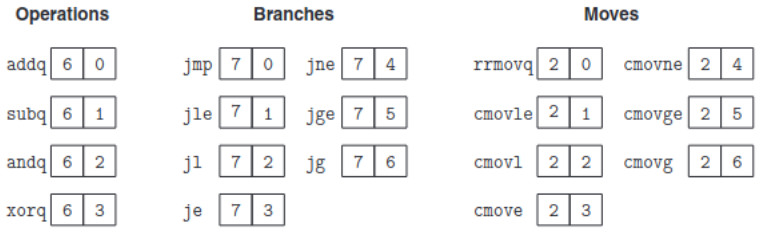
  1) icode : gives the instruction

  2) ifun : gives the specific function (for instructions like cmovXX, OPq, jXX)
- Additionally, if the instruction requires register IDs, Byte 1 is again divided into 2 4-bit blocks :

  1) rA : gives first register

  2) rB : gives second register
- If the instruction also requires a constant value, it is contained in the next 8 bytes.





- For our processor, the following instructions are hard coded into the instruction memory as the input

```
instr_memory[0]  = 8'b00010000; // nop
instr_memory[1]  = 8'b00110000; // irmovq
instr_memory[2]  = 8'b11110010; // F, rB = 2;
instr_memory[3]  = 8'b00000110; // 6
instr_memory[4]  = 8'b00000000;
instr_memory[5]  = 8'b00000000;
instr_memory[6]  = 8'b00000000;
instr_memory[7]  = 8'b00000000;
instr_memory[8] = 8'b00000000;
instr_memory[9] = 8'b00000000;
instr_memory[10] = 8'b00000000;
instr_memory[11]  = 8'b00110000; // irmovq
instr_memory[12]  = 8'b11110011; // F, rB = 3;
instr_memory[13]  = 8'b00000101; // 5
instr_memory[14]  = 8'b00000000;
instr_memory[15]  = 8'b00000000;
instr_memory[16]  = 8'b00000000;
instr_memory[17]  = 8'b00000000;
instr_memory[18] = 8'b00000000;
instr_memory[19] = 8'b00000000;
instr_memory[20] = 8'b00000000;
instr_memory[21]  = 8'b00110000; // irmovq
instr_memory[22]  = 8'b11110111; // F, rB = 7;
instr_memory[23]  = 8'b00001011; // 11
instr_memory[24]  = 8'b00000000;
instr_memory[25]  = 8'b00000000;
instr_memory[26]  = 8'b00000000;
instr_memory[27]  = 8'b00000000;
instr_memory[28] = 8'b00000000;
instr_memory[29] = 8'b00000000;
instr_memory[30] = 8'b00000000;
instr_memory[31]  = 8'b00110000; // irmovq
instr_memory[32]  = 8'b11110001; // F, rB = 1;
instr_memory[33]  = 8'b00001010; // 10
instr_memory[34]  = 8'b00000000;
instr_memory[35]  = 8'b00000000;
instr_memory[36]  = 8'b00000000;
instr_memory[37]  = 8'b00000000;
instr_memory[38] = 8'b00000000;
instr_memory[39] = 8'b00000000;
instr_memory[40] = 8'b00000000;
instr_memory[41] = 8'b00100000; // rrmovq
instr_memory[42] = 8'b00010101; // rA = 1, rB = 5
instr_memory[43]  = 8'b01000000; // rmmovq
instr_memory[44]  = 8'b01110010; // rA = 7, rB = 2;
instr_memory[45]  = 8'b00000110; // 6
instr_memory[46]  = 8'b00000000;
instr_memory[47]  = 8'b00000000;
instr_memory[48]  = 8'b00000000;
instr_memory[49]  = 8'b00000000;
instr_memory[50] = 8'b00000000;
instr_memory[51] = 8'b00000000;
instr_memory[52] = 8'b00000000;
instr_memory[53]  = 8'b01010000; // mrmovq
instr_memory[54]  = 8'b10010101; // rA = 9, rB = 5;
instr_memory[55]  = 8'b00000010; // 2
instr_memory[56]  = 8'b00000000;
instr_memory[57]  = 8'b00000000;
instr_memory[58]  = 8'b00000000;
instr_memory[59]  = 8'b00000000;
instr_memory[60] = 8'b00000000;
instr_memory[61] = 8'b00000000;
instr_memory[62] = 8'b00000000;
instr_memory[63] = 8'b01100000; // OPq ADD
```

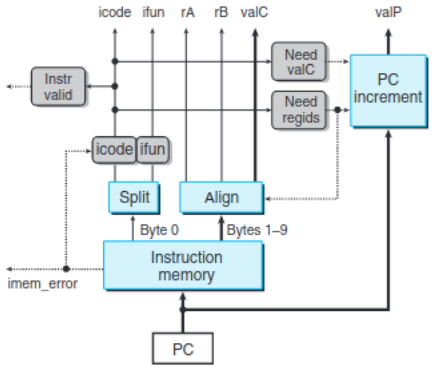
```
instr_memory[64] = 8'b00100011; // rA = 2, rB = 3;
instr_memory[65] = 8'b01110000; // jXX
instr_memory[66] = 8'b01001010; // 74
instr_memory[67] = 8'b00000000;
instr_memory[68] = 8'b00000000;
instr_memory[69] = 8'b00000000;
instr_memory[70] = 8'b00000000;
instr_memory[71] = 8'b00000000;
instr_memory[72] = 8'b00000000;
instr_memory[73] = 8'b00000000;
instr_memory[74] = 8'b01100000; // OPq ADD
instr_memory[75] = 8'b00010010; // rA = 1, rB = 2;
instr_memory[76] = 8'b10100000; // pushq
instr_memory[77] = 8'b10101111; // rA = 10, F;
instr_memory[78] = 8'b10110000; // popq
instr_memory[79] = 8'b01101111; // rA = 6, F;
instr_memory[80] = 8'b10000000; // call
instr_memory[81] = 8'b01011011; // 91
instr_memory[82] = 8'b00000000;
instr_memory[83] = 8'b00000000;
instr_memory[84] = 8'b00000000;
instr_memory[85] = 8'b00000000;
instr_memory[86] = 8'b00000000;
instr_memory[87] = 8'b00000000;
instr_memory[88] = 8'b00000000;
instr_memory[89] = 8'b00000000;
instr_memory[90] = 8'b10010000; // ret
instr_memory[91] = 8'b00010000; // nop
instr_memory[92] = 8'b00000000; // halt
```

SEQ Processor

- To reach our eventual goal of developing an efficient pipelined processor, we start off by having a sequential implementation (called SEQ) of the same by organising it into stages.

Fetch	icode, ifun rA, rB valC valP
Decode	srcA, valA srcB, valB
Execute	valE Cnd
Memory	valM
Write Back	dstE, dstM
PC Update	PC

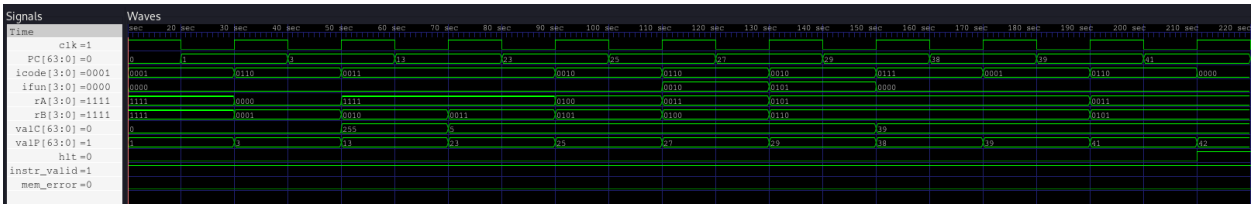
1) Fetch



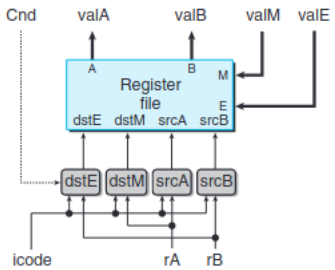
- The instruction is read from the instruction memory. `icode` and `ifun` are obtained from Byte 0. If the instruction involves registers, `rA` and `rB` are obtained from Byte 1. Bytes 2-9 hold any other parameters (like constant or destination) required by the instruction and is read into `valC`.
- `valP` is computed as the next value of the PC (program counter).
- `mem_error` is used to keep track of whether the memory address is valid and `instr_valid` is used to check the validity of the instruction.

```
module SEQfetch(clk, PC, icode, ifun, rA, rB, valC, valP, hlt, mem_error, instr_valid);
    input clk;
    input [63:0] PC;
    output reg hlt, mem_error, instr_valid;
    output reg [3:0] icode, ifun, rA, rB;
    output reg [63:0] valC, valP;

    initial begin
        rA = 4'hF;
        rB = 4'hF;
        valC = 0;
        valP = 0;
        hlt = 0;
        mem_error = 0;
        instr_valid = 1;
    end

    reg [7:0] opcode, regids;

    always @(posedge clk) begin
        if (PC > 1023) begin
            mem_error = 1;
        end
        opcode = processor.instr_memory[PC];
        icode = opcode[7:4];
        ifun = opcode[3:0];
        if (icode == 4'b0000) begin // halt
            hlt = 1;
            valP = PC+1;
        end
        else if (icode == 4'b0001) begin // nop
            valP = PC+1;
        end
        else if (icode == 4'b0010) begin // cmovXX
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valP = PC+2;
        end
        else if (icode == 4'b0011) begin // irmovq
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valC = {processor.instr_memory[PC+9], processor.instr_memory[PC+8], processor.instr_memory[PC+7], processor.instr_memory[PC+6],
            valP = PC+10;
```

```
        end
        else if (icode == 4'b0100) begin // rmmovq
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valC = {processor.instr_memory[PC+9], processor.instr_memory[PC+8], processor.instr_memory[PC+7], processor.instr_memory[PC+6],
            valP = PC+10;
        end
        else if (icode == 4'b0101) begin // mrmovq
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valC = {processor.instr_memory[PC+9], processor.instr_memory[PC+8], processor.instr_memory[PC+7], processor.instr_memory[PC+6],
            valP = PC+10;
        end
        else if (icode == 4'b0110) begin // OPq
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valP = PC+2;
        end
        else if (icode == 4'b0111) begin // jXX
            valC = {processor.instr_memory[PC+8], processor.instr_memory[PC+7], processor.instr_memory[PC+6], processor.instr_memory[PC+5],
            valP = PC+9;
        end
        else if (icode == 4'b1000) begin // call
            valC = {processor.instr_memory[PC+8], processor.instr_memory[PC+7], processor.instr_memory[PC+6], processor.instr_memory[PC+5],
            valP = PC+9;
        end
        else if (icode == 4'b1001) begin // ret
            valP = PC+1;
        end
        else if (icode == 4'b1010) begin // pushq
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valP = PC+2;
        end
        else if (icode == 4'b1011) begin // popq
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valP = PC+2;
        end
        else begin
            instr_valid = 0;
        end
    end
endmodule
```

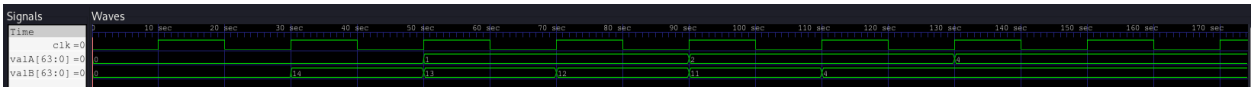


2) Decode

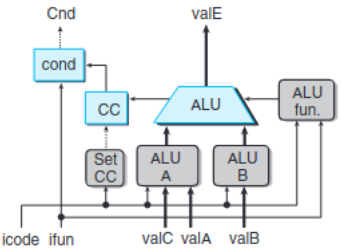


- `valA` and `valB` are read from the registers rA and rB.

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

    initial begin
        valA = 0;
        valB = 0;
    end

    always @(posedge clk) begin
        if (icode == 4'b0000) begin // halt
        end
        else if (icode == 4'b0001) begin // nop
        end
        else if (icode == 4'b0010) begin // cmovXX
            valA = processor.registers[rA];
        end
        else if (icode == 4'b0011) begin // irmovq
            valB = 0;
        end
        else if (icode == 4'b0100) begin // rmmovq
            valA = processor.registers[rA];
            valB = processor.registers[rB];
        end
        else if (icode == 4'b0101) begin // mrmovq
            valB = processor.registers[rB];
        end
        else if (icode == 4'b0110) begin // OPq
            valA = processor.registers[rA];
            valB = processor.registers[rB];
        end
        else if (icode == 4'b0111) begin // jXX
        end
        else if (icode == 4'b1000) begin // call
            valB = processor.registers[4];
        end
        else if (icode == 4'b1001) begin // ret
            valA = processor.registers[4];
            valB = processor.registers[4];
        end
        else if (icode == 4'b1010) begin // pushq
            valA = processor.registers[rA];
            valB = processor.registers[4];
        end
        else if (icode == 4'b1011) begin // popq
            valA = processor.registers[4];
            valB = processor.registers[4];
        end
    end
endmodule
```



3) Execute

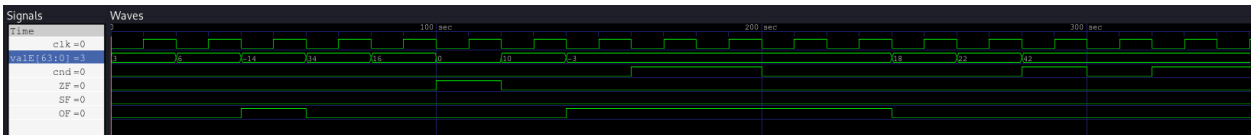


- The values read into valA and valB are passed into the ALU as inputs and the required operation based on the values of icode and ifun is performed. The output is stored in `valE`.
- The condition code `Cnd` and the flags `ZF`, `SF` and `OF` are also set in this stage based on the outputs obtained.

```
module SEQexecute(clk, icode, ifun, valA, valB, valC, valE, cnd, ZF, SF, OF);
    input clk;
    input [3:0] icode, ifun;
    input [63:0] valA, valB, valC;
    output reg cnd, ZF, OF, SF;
    output reg [63:0] valE;

    initial begin
      cnd = 0;
      valE = 0;
      ZF = 0;
      OF = 0;
      SF = 0;
    end

    reg [1:0] control;
    reg signed [63:0] aluInputA, aluInputB;
    wire tempZF, tempSF, tempOF;
    wire overflow;
    wire signed [63:0] out;
    initial begin
      control = 0;
      aluInputA = 0;
      aluInputB = 0;
    end

    alu execute_ALU(aluInputA, aluInputB, control, out, overflow, tempZF, tempOF, tempSF);

    always @(*) begin
      // calculating values using ALU
      if (icode == 4'b0000) begin // halt
      end
      else if (icode == 4'b0001) begin // nop
      end
      else if (icode == 4'b0010) begin // cmovXX
          aluInputA = valA;
          aluInputB = 0;
          control = 2'b00;
          valE = out;
          if (ifun == 4'b0000) begin
            cnd = 1;
          end
          else if (ifun == 4'b0001) begin
            cnd = (SF ^ OF) | ZF;
          end
          else if (ifun == 4'b0010) begin
            cnd = SF^OF;
          end
          else if (ifun == 4'b0011) begin
            cnd = ZF;
          end
          else if (ifun == 4'b0100) begin
            cnd = ~ZF;
          end
          else if (ifun == 4'b0101) begin
            cnd = ~(SF^OF);
          end
          else if (ifun == 4'b0110) begin
            cnd = ~(SF^OF)&(~ZF);
          end
      end
      else if (icode == 4'b0011) begin // irmovq
          aluInputA = valC;
          aluInputB = 0;
          control = 2'b00;
          valE = out;
      end
      else if (icode == 4'b0100) begin // rmmovq
          aluInputA = valC;
          aluInputB = valB;
          control = 2'b00;
          valE = out;
      end
```

```
      else if (icode == 4'b0101) begin // mrmovq
          aluInputA = valC;
          aluInputB = valB;
          control = 2'b00;
          valE = out;
      end
      else if (icode == 4'b0110) begin // OPq
          aluInputA = valA;
          aluInputB = valB;
          if (ifun == 4'b0000) begin // ADD
            control = 2'b00;
          end
          else if (ifun == 4'b0001) begin // SUB
            control = 2'b01;
          end
          else if (ifun == 4'b0010) begin // AND
            control = 2'b10;
          end
          else if (ifun == 4'b0011) begin // XOR
            control = 2'b11;
          end
          valE = out;
      end
      else if (icode == 4'b0111) begin // jXX
          if (ifun == 4'b0000) begin
            cnd = 1;
          end
          else if (ifun == 4'b0001) begin
            cnd = (SF ^ OF) | ZF;
          end
          else if (ifun == 4'b0010) begin
            cnd = SF^OF;
          end
          else if (ifun == 4'b0011) begin
            cnd = ZF;
          end
          else if (ifun == 4'b0100) begin
            cnd = ~ZF;
          end
          else if (ifun == 4'b0101) begin
            cnd = ~(SF^OF);
          end
          else if (ifun == 4'b0110) begin
            cnd = ~(SF^OF)&(~ZF);
          end
      end
      else if (icode == 4'b1000) begin // call
          aluInputA = -4'b1000;
          aluInputB = valB;
          control = 2'b00;
          valE = out;
      end
      else if (icode == 4'b1001) begin // ret
          aluInputA = 4'b1000;
          aluInputB = valB;
          control = 2'b00;
          valE = out;
      end
      else if (icode == 4'b1010) begin // pushq
          aluInputA = -4'b1000;
          aluInputB = valB;
          control = 2'b00;
          valE = out;
      end
      else if (icode == 4'b1011) begin // popq
          aluInputA = 4'b1000;
          aluInputB = valB;
          control = 2'b00;
          valE = out;
      end
      // setting flags
     assign ZF = tempZF;
      assign OF = tempOF;
      assign SF = tempSF;
    end

endmodule
```

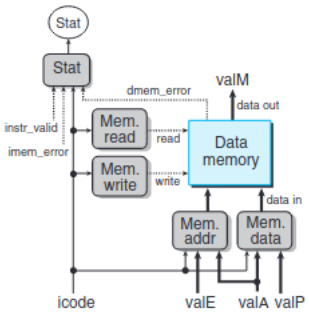
Instruction		Synonym	Jump condition	Description
`jmp`	*Label*		1	Direct jump
`jmp`	**Operand*		1	Indirect jump
`je`	*Label*	`jz`	ZF	Equal / zero
`jne`	*Label*	`jnz`	~ZF	Not equal / not zero
`js`	*Label*		SF	Negative
`jns`	*Label*		~SF	Nonnegative
`jg`	*Label*	`jnle`	~(SF ^ OF) & ~ZF	Greater (signed >)
`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 <=)



- The ALU implemented to be used in the execute stage also sets the value of ZF, SF and OF.

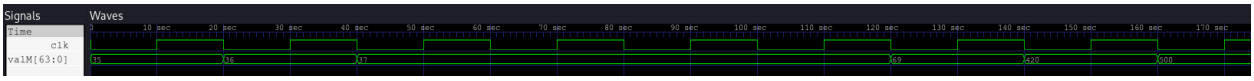
```
module alu(input signed [63:0]a, input signed [63:0]b, input [1:0] control, output signed [63:0] ans, output overflow, output tempZF, outpu
    wire signed [63:0] tempadd; wire overflowadd;
    wire signed [63:0] tempsub; wire overflowsub;
    wire signed [63:0] tempand;
    wire signed [63:0] tempxor;
    reg signed [63:0] tempans;
    reg tempoverflow;
    wire tempZF, tempSF, tempOF;
    add64 G1 (a,b,tempadd, overflowadd);
    sub64 G2 (a,b,tempsub, overflowsub);
    and64 G3 (a,b,tempand);
    xor64 G4 (a,b,tempxor);
    always @(*)
    begin
        case(control)
            2'b00: // ADD
            begin
                tempans = tempadd;
                tempoverflow = overflowadd;
            end
            2'b01: // SUB
            begin
                tempans = tempsub;
                tempoverflow = overflowsub;
            end
            2'b10: // AND
            begin
                tempans = tempand;
                tempoverflow= 1'b0;
            end
            2'b11: // XOR
            begin
                tempans = tempxor;
                tempoverflow= 1'b0;
            end
        endcase
    end
    assign ans = tempans;
    assign overflow = tempoverflow;
    assign tempZF = (ans == 1'b0);
    assign tempOF = (overflow != 0);
    assign tempSF = (ans[63] == 1);

endmodule
```

4) Memory



- In this stages, values are either read from or written to the data memory based on the instruction. Values are read into `valM`. They are written into memory with location valE based on other parameters such as icode, ifun, valA, valP etc.

```
module SEQmemory(clk, icode, valA, valE, valP, valM);
    input clk;
    input [3:0] icode;
    input [63:0] valA, valE, valP;
    output reg [63:0] valM;

    always @(*) begin
         if (icode == 4'b0101) begin // mrmovq
            valM = processor.data_memory[valE];
        end
        else if (icode == 4'b1001) begin // ret
            valM = processor.data_memory[valA];
        end
        else if (icode == 4'b1011) begin // popq
            valM = processor.data_memory[valA];
        end
    end
    always @(posedge clk) begin
       if (icode == 4'b0100) begin // rmmovq
            processor.data_memory[valE] = valA;
    end
        else if (icode == 4'b1000) begin // call
            processor.data_memory[valE] = valP;
        end
        else if (icode == 4'b1010) begin // pushq
            processor.data_memory[valE] = valA;
        end
    end
endmodule
```

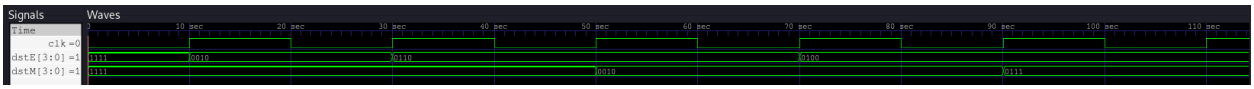


5) Writeback

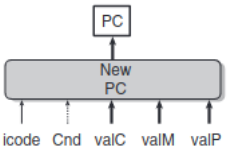
- This stage is used to write the values into registers. `dstE` and `dstM` are obtained in this stage and values are written into their corresponding register.
- The decode and writeback stage can be combined into a single stage as well.

```
module SEQwriteback(clk, icode, rA, rB, valA, valB, valE, valM, cnd, dstE, dstM);
    input clk, cnd;
    input [3:0] icode, rA, rB;
    input [63:0] valA, valB, valE, valM;
    output reg [3:0] dstE, dstM;

    initial begin
    dstE = 4'hF;
    dstM = 4'hF;
    end

    always @(posedge clk) begin
        if (icode == 4'b0000) begin // halt
```

```
        end
        else if (icode == 4'b0001) begin // nop
        end
        else if ((icode == 4'b0010) && (cnd == 1)) begin // cmovXX
            dstE = rB;
            processor.registers[rB] = valE;
        end
        else if (icode == 4'b0011) begin // irmovq
            dstE = rB;
            processor.registers[rB] = valE;
        end
        else if (icode == 4'b0100) begin // rmmovq
        end
        else if (icode == 4'b0101) begin // mrmovq
            dstM = rA;
            processor.registers[rA] = valM;
        end
        else if (icode == 4'b0110) begin // OPq
            dstE = rB;
            processor.registers[rB] = valE;
        end
        else if (icode == 4'b0111) begin // jXX
        end
        else if (icode == 4'b1000) begin // call
            dstE = 4;
            processor.registers[4] = valE;
        end
        else if (icode == 4'b1001) begin // ret
            dstE = 4;
            processor.registers[4] = valE;
        end
        else if (icode == 4'b1010) begin // pushq
            dstE = 4;
            processor.registers[4] = valE;
        end
        else if (icode == 4'b1011) begin // popq
            dstE = 4;
            dstM = rA;
            processor.registers[4] = valE;
            processor.registers[rA] = valM;
        end
    end
endmodule
```



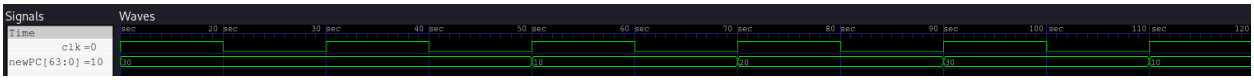
6) PC Update



- This is the last stage of our sequential implementation. It computes the new value of PC based on parameters like icode, Cnd, valC, valM and valP and then sets the value `newPC`.

```
module SEQPCupdate(clk, PC, icode, cnd, valC, valM, valP, newPC);
    input clk, cnd;
    input [3:0] icode;
    input [63:0] PC, valC, valM, valP;
    output reg [63:0] newPC;
    always @(posedge clk) begin
        if (icode == 4'b0111) begin // jXX
            newPC = (cnd == 1) ? valC : valP;
        end
        else if (icode == 4'b1000) begin // call
            newPC = valC;
        end
        else if (icode == 4'b1001) begin // ret
```

```
            newPC = valM;
        end
        else begin // all other instructions
            newPC = valP;
        end
    end
endmodule
```

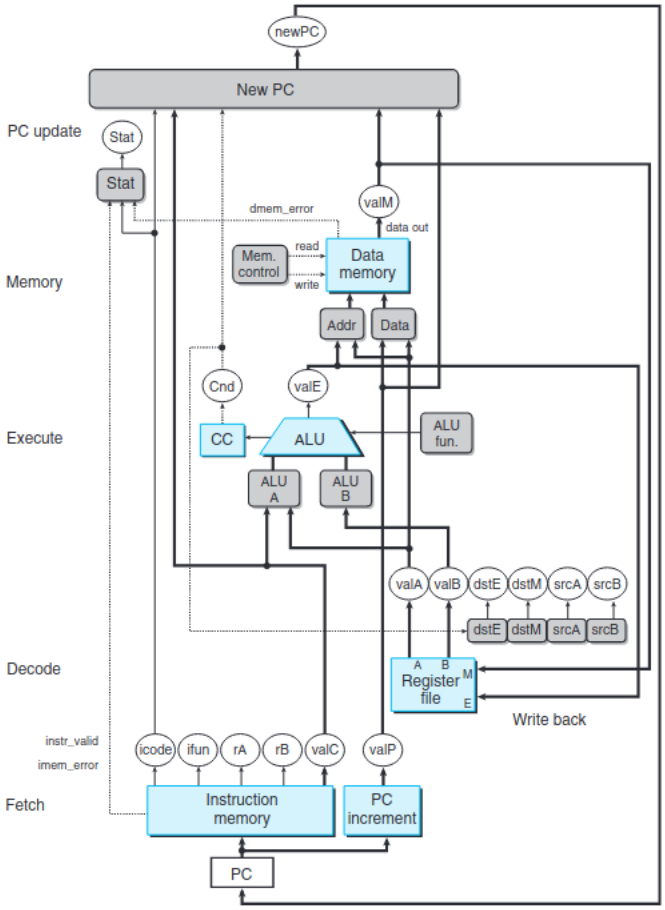


Processor

- The final processor is then made using all the stages. For our case, the instructions being executed are part of the `instr_memory`.

```
module processor;
    reg clk;
    reg [3:0] status_codes; // AOK, HLT, ADR, INS
    reg [63:0] PC;
    wire cnd, hlt, mem_error, instr_valid, ZF, SF, OF;
    wire [3:0] icode, ifun, rA, rB, dstE, dstM;
    wire [63:0] valA, valB, valC, valE, valM, valP, newPC;
    reg [7:0] instr_memory[0:1023];
    reg [63:0] registers [0:14];
    reg [63:0] data_memory[2047:0];
    initial begin
        //setting values of instr_memory, registers and data_memory
    end

    always #5 clk = ~clk;
    always @(*) begin
        PC = newPC;
    end

    SEQfetch proc_fetch(clk, PC, icode, ifun, rA, rB, valC, valP, hlt, mem_error, instr_valid);
    SEQdecode proc_decode(clk, icode, rA, rB, valA, valB);
    SEQexecute proc_execute(clk, icode, ifun, valA, valB, valC, valE, cnd, ZF, SF, OF);
    SEQmemory proc_memory(clk, icode, valA, valE, valP, valM);
    SEQwriteback proc_writeback(clk, icode, rA, rB, valA, valB, valE, valM, cnd, dstE, dstM);
    SEQPCupdate proc_PCupdate(clk, PC, icode, cnd, valC, valM, valP, newPC);

    initial begin
        $dumpfile("processor.vcd");
        $dumpvars(0, processor);
        clk = 0;
        PC = 0;
        status_codes[0] = 1;
        status_codes[1] = 0;
        status_codes[2] = 0;
        status_codes[3] = 0;
    end

    always @(posedge clk) begin
        if (instr_valid) begin
                status_codes[0] = 1;
                status_codes[1] = 0;
                status_codes[2] = 0;
                status_codes[3] = 0;
        end
        else if (mem_error) begin
            status_codes[0] = 0;
            status_codes[1] = 0;
            status_codes[2] = 1;
            status_codes[3] = 0;
        end
        else if (hlt) begin
            status_codes[0] = 0;
            status_codes[1] = 1;
            status_codes[2] = 0;
            status_codes[3] = 0;
        end
        else if (!instr_valid) begin
            status_codes[0] = 0;
            status_codes[1] = 0;
            status_codes[2] = 0;
```

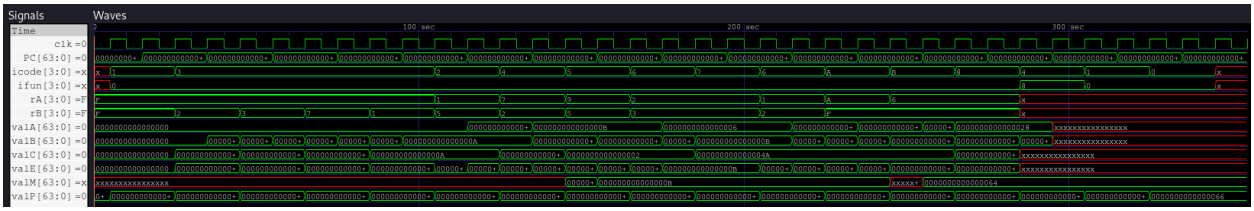
```
            status_codes[3] = 1;
        end

        if (status_codes[1] == 1 || status_codes[3] == 1) begin
            $finish;
        end
    end

    always @(*) begin
        $monitor("clk = %0d \tPC = %0d\nicode = %0b (%0d)\tifun = %0b (%0d)\nrA = %0d \trB = %0d \tvalA = %0d \tvalB = %0d \nvalC = %0d \tv
    end

endmodule
```



- The output obtained for our given instructions can be seen in the terminal by executing the processor output file as `./processor` .

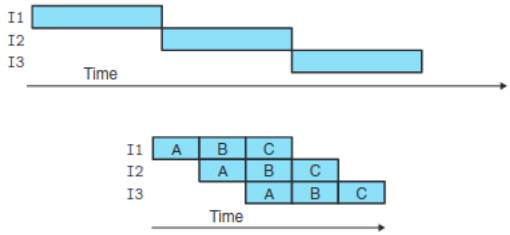
```
clk = 1          PC = 63
icode = 101 (5) ifun = 0 (0)
rA = 9  rB = 5  valA = 11        valB = 10
valC = 2         valE = 12       valM = 11        valP = 63
AOK = 1          HLT = 0         ADR = 0          INS = 0

clk = 0          PC = 63
icode = 101 (5) ifun = 0 (0)
rA = 9  rB = 5  valA = 11        valB = 10
valC = 2         valE = 12       valM = 11        valP = 63
AOK = 1          HLT = 0         ADR = 0          INS = 0

clk = 1          PC = 63
icode = 110 (6) ifun = 0 (0)
rA = 2  rB = 3  valA = 11        valB = 10
valC = 2         valE = 21       valM = 11        valP = 65
AOK = 1          HLT = 0         ADR = 0          INS = 0

clk = 0          PC = 63
icode = 110 (6) ifun = 0 (0)
rA = 2  rB = 3  valA = 11        valB = 10
valC = 2         valE = 21       valM = 11        valP = 65
AOK = 1          HLT = 0         ADR = 0          INS = 0

clk = 1          PC = 65
icode = 110 (6) ifun = 0 (0)
rA = 2  rB = 3  valA = 6         valB = 5
valC = 2         valE = 11       valM = 11        valP = 65
AOK = 1          HLT = 0         ADR = 0          INS = 0

clk = 0          PC = 65
icode = 110 (6) ifun = 0 (0)
rA = 2  rB = 3  valA = 6         valB = 5
valC = 2         valE = 11       valM = 11        valP = 65
AOK = 1          HLT = 0         ADR = 0          INS = 0

clk = 1          PC = 65
icode = 111 (7) ifun = 0 (0)
rA = 2  rB = 3  valA = 6         valB = 11
valC = 74        valE = 11       valM = 11        valP = 74
AOK = 1          HLT = 0         ADR = 0          INS = 0
```

Sample Output



PIPE Processor

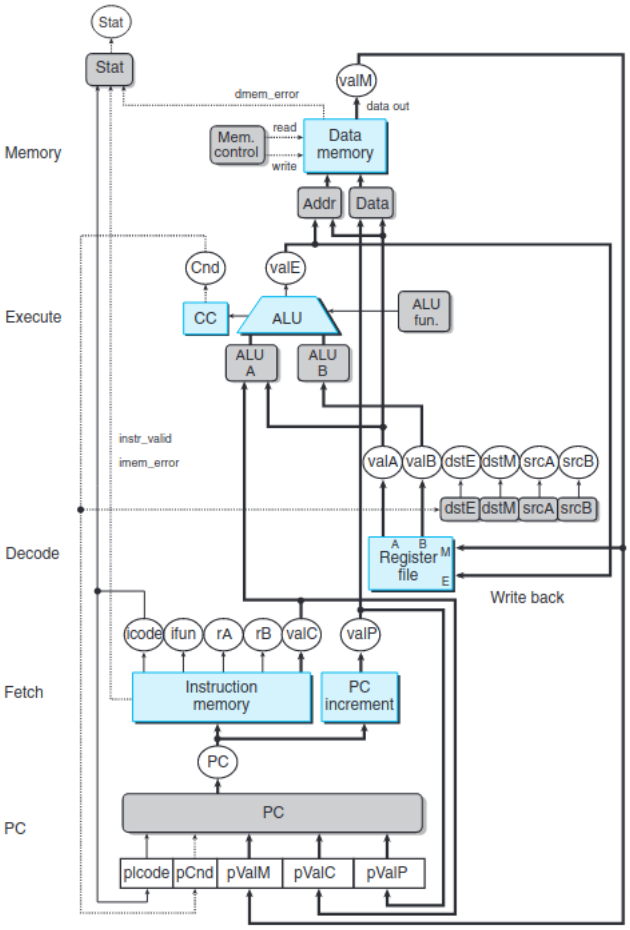
Sequential vs Pipeline



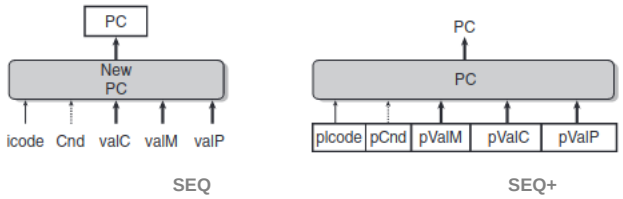
- Here, we see a sequential and 3 stage pipelined implementation of the same 3 instructions. While SEQ would take 9 clock cycles, PIPE can do the same in 5 cycles.
- The biggest advantage of pipeline implementation is an increased throughput, even if slightly increases latency.

Pipeline Basics

- To implement our pipelined processor, we make a small adaptation to the sequential processor (SEQ) and shift the computation of the PC into the fetch stage. This is done so that we are able to continuously fetch the next instruction without having to wait for the PC Update stage (which used to be at the end) to go to completion.
- We refer to this modified design as SEQ+. This process is called circuit retiming and and it only changes the state representation without bothering the logical behavior.



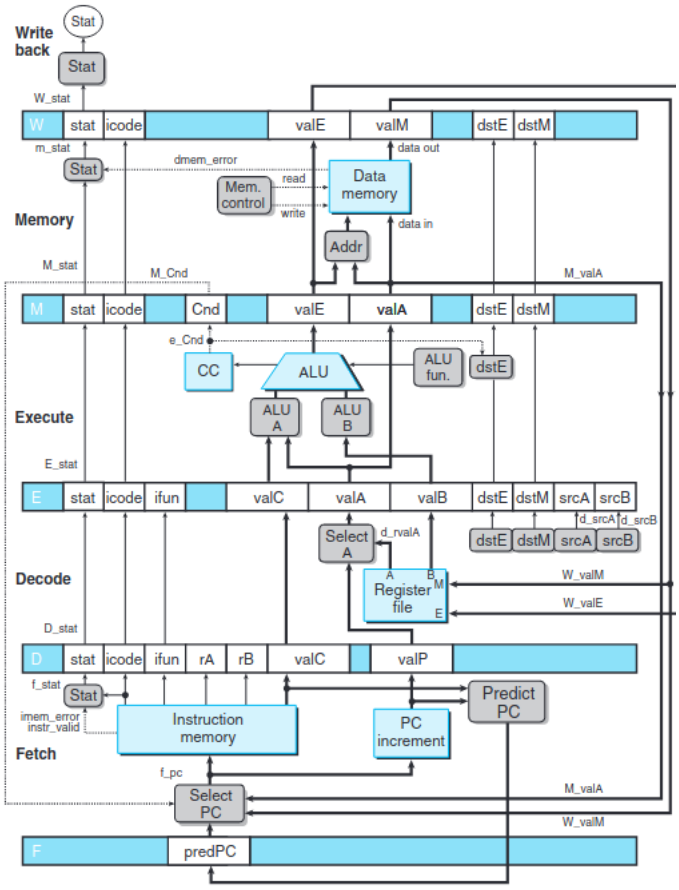
- SEQ vs SEQ+ PC Computation thus differs and can be seen as



- We also require some more changes to get our desired pipelined implementation such as
  - support for data forwarding
  - PC prediction
  - control logic (to eliminate hazards)

Inserting Pipeline Registers

- Now, we rearrange some of the hardware and signals in the SEQ implementation by inserting our pipelining registers in between the various stages and thus, obtain the PIPE- implementation. These registers stop the signals from one stage from flowing into the next stage and affecting the processing supposed to happen there



1) F

- before the fetch stage
- holds value of predicted PC

```
module regF(clk, predPC, f_PC);
    input clk;
    input [63:0] predPC;
    output reg [63:0] f_PC;

    always @(posedge clk) begin
        f_PC <= predPC;
    end
endmodule
```

2) D

- in between fetch and decode stage
- holds information about the most recently fetched instruction, which gets sent to be decoded

```
module regD(clk, f_stat, f_icode, f_ifun, f_rA, f_rB, f_valC, f_valP, d_stat, d_icode, d_ifun, d_rA, d_rB, d_valC, d_valP);
    input clk;
    input [2:0] f_stat;
    input [3:0] f_icode, f_ifun, f_rA, f_rB;
    input [63:0] f_valC, f_valP;
    output reg [2:0] d_stat;
    output reg [3:0] d_icode, d_ifun, d_rA, d_rB;
    output reg [63:0] d_valC, d_valP;

    always @(posedge clk) begin
        d_stat <= f_stat;
        d_icode <= f_icode;
        d_ifun <= f_ifun;
        d_rA <= f_rA;
        d_rB <= f_rB;
        d_valC <= f_valC;
        d_valP <= f_valP;
    end
endmodule
```

3) E

- in between decode and execute stage
- holds information about the most recently decoded instruction, which gets sent to be executed

```
module regE(clk, d_stat, d_icode, d_ifun, d_rA, d_rB, d_valA, d_valB, d_valC, d_valP,e_stat, e_icode, e_ifun, e_rA, e_rB, e_valA, e_valB, e
    input clk;
    input [2:0] d_stat;
    input [3:0] d_icode, d_ifun, d_rA, d_rB;
    input [63:0] d_valA, d_valB, d_valC, d_valP;
    output reg [2:0] e_stat;
    output reg [3:0] e_icode, e_ifun, e_rA, e_rB;
    output reg [63:0] e_valA, e_valB, e_valC, e_valP;

    always @(posedge clk) begin
        e_stat <= d_stat;
        e_icode <= d_icode;
        e_ifun <= d_ifun;
        e_rA <= d_rA;
        e_rB <= d_rB;
        e_valA <= d_valA;
        e_valB <= d_valB;
        e_valC <= d_valC;
        e_valP <= d_valP;
    end
endmodule
```

4) M

- in between the execute and memory stage
- holds the results of the most recently executed instruction

- also holds branch conditions and targets, which are relevant to conditional jumps

```
module regM(clk, e_stat, e_icode, e_rA, e_rB, e_valA, e_valB, e_valC, e_valE, e_valP,e_Cnd, m_stat, m_icode, m_rA, m_rB, m_valA, m_valB, m_
    input clk;
    input e_Cnd;
    input [2:0] e_stat;
    input [3:0] e_icode, e_rA, e_rB;
    input [63:0] e_valA, e_valB, e_valC, e_valE, e_valP;
    output reg m_Cnd;
    output reg [2:0] m_stat;
    output reg [3:0] m_icode, m_rA, m_rB;
    output reg [63:0] m_valA, m_valB, m_valC, m_valE, m_valP;

    always @(posedge clk) begin
        m_Cnd <= e_Cnd;
        m_stat <= e_stat;
        m_icode <= e_icode;
        m_rA <= e_rA;
        m_rB <= e_rB;
        m_valA <= e_valA;
        m_valB <= e_valB;
        m_valC <= e_valC;
        m_valE <= e_valE;
        m_valP <= e_valP;
    end
endmodule
```

5) W

- in between memory and writeback stage
- connected to the feedback paths which supply the computed results to be written to the registers
- also holds the return address for the PC selection logic when completing a ret instruction

```
module regW(clk, m_stat, m_icode, m_rA, m_rB, m_valA, m_valB, m_valC, m_valE, m_valM, m_valP, m_Cnd, w_stat, w_icode, w_rA, w_rB, w_valA, w
    input clk;
    input m_Cnd;
    input [2:0] m_stat;
    input [3:0] m_icode, m_rA, m_rB;
    input [63:0] m_valA, m_valB, m_valC, m_valE, m_valM, m_valP;
    output reg w_Cnd;
    output reg [2:0] w_stat;
    output reg [3:0] w_icode, w_rA, w_rB;
    output reg [63:0] w_valA, w_valB, w_valC, w_valE, w_valM, w_valP;

    always @(posedge clk) begin
        w_Cnd <= m_Cnd;
        w_stat <= m_stat;
        w_icode <= m_icode;
        w_rA <= m_rA;
        w_rB <= m_rB;
        w_valA <= m_valA;
        w_valB <= m_valB;
        w_valC <= m_valC;
        w_valE <= m_valE;
        w_valM <= m_valM;
        w_valP <= m_valP;
    end
endmodule
```

Rearranging and Relabeling Signals

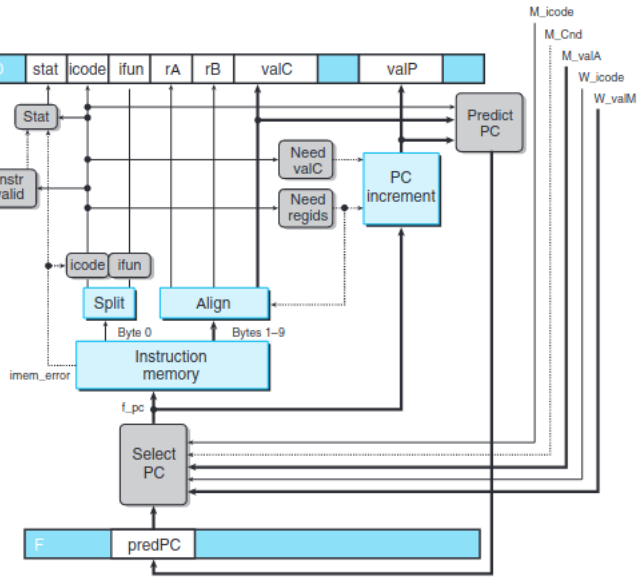
- Another important change required for obtaining the pipelined implementation is the rearranging and relabeling of signals. We maintain the signal separately at each stage and thus they are now each labelled with a prefix corresponding to the stage they are relevant to.
- The prefix is in uppercase if the signal is stored in a pipeline register (same letter prefix as that of register).
- The prefix is in lowercase if the signal is only computed within a stage (same letter prefix as that of stage).

Next PC Prediction (Branch Prediction)

- To also increase the throughput (ideally 1 new instruction per cycle), we also need to determine the location of the next instruction right after fetching the current instruction. This process is called PC Prediction.
- Apart from conditional jumps, ret and call, the value of PC will always be valP. For call and unconditional jump, the value will be valC.
- However, this prediction is still not fully correct for jumps and doesn't work for ret too meaning we have to deal with incorrect predictions which can require a lot of hardware to resolve. Instead, in our implementation, we simply stall until the ret instruction has passed through the writeback stage.
- Another way to deal with misprediction of branches is by cancelling all the instructions (so far) at the branch target and continue fetching from the next instruction (after the jump).

Stages

1) Fetch

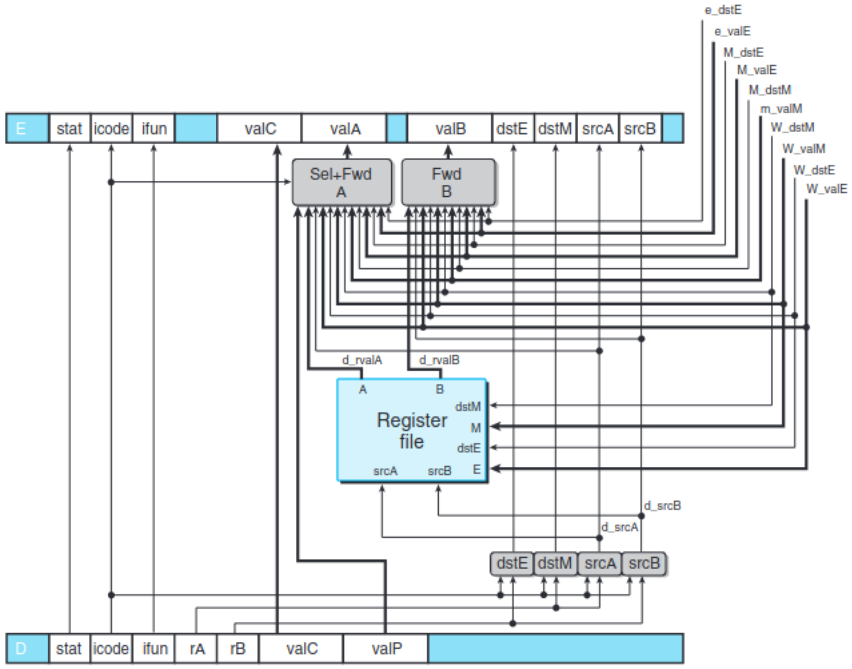



```
module PIPEfetch(clk, M_Cnd, F_stall, D_stall, D_bubble, M_icode, W_icode, M_valA, W_valM, F_predPC, D_stat, D_icode, D_ifun, D_rA, D_rB, D
    input clk, M_Cnd, F_stall, D_stall, D_bubble;
    input [3:0] M_icode, W_icode;
    input [63:0] M_valA, W_valM, F_predPC;
    output reg [3:0] D_stat, D_icode, D_ifun, D_rA, D_rB;
    output reg [63:0] D_valC, D_valP, f_predPC;

    reg hlt, mem_error, instr_valid;
    reg [3:0]stat, icode, ifun, rA, rB;
    reg [63:0] PC, valC, valP;

    always @(*) begin
        if (M_icode == 4'b0111 && M_Cnd == 0) begin
            PC = M_valA;
        end
        else if (W_icode == 4'b1001) begin
            PC = W_valM;
        end
        else begin
            PC = F_predPC;
        end
    end

    reg [7:0] opcode, regids;

    always @(posedge clk) begin
```

```
        if (PC > 1023) begin
            mem_error = 1;
        end
        opcode = instr_memory[PC];
        icode = opcode[7:4];
        ifun = opcode[3:0];
        if (icode == 4'b0000) begin // halt
            hlt = 1;
            valP = PC+1;
            f_predPC = valP;
        end
        else if (icode == 4'b0001) begin // nop
            valP = PC+1;
            f_predPC = valP;
        end
        else if (icode == 4'b0010) begin // cmovXX
            regids = instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valP = PC+2;
            f_predPC = valP;
        end
        else if (icode == 4'b0011) begin // irmovq
            regids = instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valC = {instr_memory[PC+9], instr_memory[PC+8], instr_memory[PC+7], instr_memory[PC+6], instr_memory[PC+5], instr_memory[PC+4],
            valP = PC+10;
            f_predPC = valP;
        end
        else if (icode == 4'b0100) begin // rmmovq
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valC = {instr_memory[PC+9], instr_memory[PC+8], instr_memory[PC+7], instr_memory[PC+6], instr_memory[PC+5], instr_memory[PC+4],
            valP = PC+10;
            f_predPC = valP;
        end
        else if (icode == 4'b0101) begin // mrmovq
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valC = {instr_memory[PC+9], instr_memory[PC+8], instr_memory[PC+7], instr_memory[PC+6], instr_memory[PC+5], instr_memory[PC+4],
            valP = PC+10;
            f_predPC = valP;
        end
        else if (icode == 4'b0110) begin // OPq
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valP = PC+2;
            F_predPC = valP;
        end
        else if (icode == 4'b0111) begin // jXX
            valC = {instr_memory[PC+8], instr_memory[PC+7], instr_memory[PC+6], instr_memory[PC+5], instr_memory[PC+4], instr_memory[PC+3],
            valP = PC+9;
            f_predPC = valP;
        end
        else if (icode == 4'b1000) begin // call
            valC = {instr_memory[PC+8], instr_memory[PC+7], instr_memory[PC+6], instr_memory[PC+5], instr_memory[PC+4], instr_memory[PC+3],
            valP = PC+9;
            f_predPC = valP;
        end
        else if (icode == 4'b1001) begin // ret
            valP = PC+1;
        end
        else if (icode == 4'b1010) begin // pushq
            regids = instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valP = PC+2;
            f_predPC = valP;
        end
        else if (icode == 4'b1011) begin // popq
            regids = processor.instr_memory[PC+1];
            rA = regids[7:4];
            rB = regids[3:0];
            valP = PC+2;
            f_predPC = valP;
        end
        else begin
```

```
            instr_valid = 0;
        end

        if (instr_valid == 0) begin
            stat = 4'b0001;
        end
        else if (mem_error == 1) begin
            stat = 4'b0010;
        end
        else if (icode == 4'b0000) begin
            stat = 4'b0100;
        end
        else begin
          stat = 4'b1000;
        end
    end

    regD proc_regD(clk, F_stall, D_stall, D_bubble, PC, F_predPC, D_icode, D_ifun, D_rA, D_rB, D_valC, D_valP, D_stat, icode, fun, rA, rB, 

endmodule
```

2 Decode + Writeback

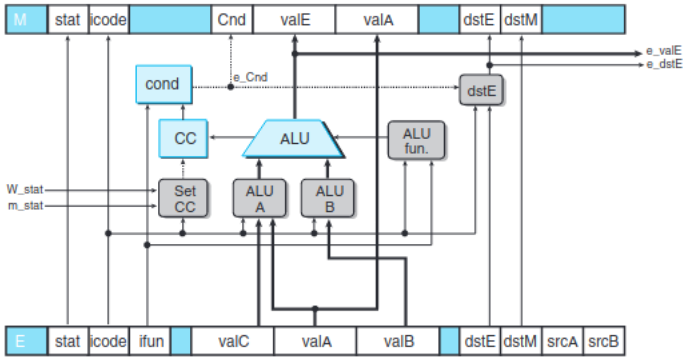



```
module PIPEdecode_writeback(clk, d_icode, d_rA, d_rB, d_valA, d_valB, w_icode, w_rA, w_rB, w_valE, w_valM, w_Cnd);
    input clk;
    input E_bubble;
    input [3:0] D_icode, D_ifun, D_rA, D_rB, D_stat, W_icode, e_dstE, M_dstE, M_dstM, W_dstE, W_dstM;
    input [63:0] D_valC, D_valP, e_valE, M_valE, m_valM, W_valE, W_valM;
    output reg [3:0] E_stat, E_icode, E_ifun, E_dstE, E_dstM, E_srcA, E_srcB, d_srcA, d_srcB;
    output reg [63:0] E_valA, E_valB, E_valC;

    reg[3:0] d_dstE, dst_M;
    reg [63:0] d_rvalA, d_rvalB, d_valA, d_valB;
    reg [63:0] registers[0:14];
    integer i;
    initial begin
        for(i = 0; i <= 14; i=i+1) begin
            registers[i] = 10*i;
        end
    end

    always @(*) begin
        d_srcA = 4'b1111;
```

```
        d_srcB = 4'b1111;
        d_dstE = 4'b1111;
        d_dstM = 4'b1111;
        if (D_icode == 4'b0000) begin // halt
        end
        else if (D_icode == 4'b0001) begin // nop
        end
        else if (D_icode == 4'b0010) begin // cmovXX
            d_srcA = D_rA;
            d_dstE = D_rB;
            d_rvalA = registers[d_rA];
            d_rvalB = 0;
        end
        else if (d_icode == 4'b0011) begin // irmovq
            d_dstE = D_rB;
            d_rvalB = 0;
        end
        else if (d_icode == 4'b0100) begin // rmmovq
            d_srcA = D_rA;
            d_srcB = D_rB;
            d_rvalA = registers[D_rA];
            d_rvalB = registers[D_rB];
        end
        else if (d_icode == 4'b0101) begin // mrmovq
            d_srcB = D_rB;
            d_dstM = D_rA;
            d_rvalB = registers[D_rB];
        end
        else if (d_icode == 4'b0110) begin // OPq
            d_srcA = D_rA;
            d_srcB = D_rB;
            d_dstE = D_rB;
            d_rvalA = registers[D_rA];
            d_rvalB = registers[D_rB];
        end
        else if (d_icode == 4'b0111) begin // jXX
        end
        else if (d_icode == 4'b1000) begin // call
            d_srcB = 4;
            d_dstE = 4;
            d_rvalB = registers[4];
        end
        else if (d_icode == 4'b1001) begin // ret
            d_srcA = 4;
            d_srcB = 4;
            d_dstE = 4;
            d_rvalA = processor.registers[4];
            d_rvalB = processor.registers[4];
        end
        else if (d_icode == 4'b1010) begin // pushq
            d_srcA = D_rA;
            d_srcB = 4;
            d_dstE = 4;
            d_rvalA = registers[D_rA];
            d_rvalB = registers[4];
        end
        else if (d_icode == 4'b1011) begin // popq
            d_srcA = 4;
            d_srcB = 4;
            d_dstE = 4;
            d_dstM = D_rA;
            d_rvalA = processor.registers[4];
            d_rvalB = processor.registers[4];
        end

            // forwarding A
        if(D_icode == 4'b0111 || D_icode == 4'b1000) begin
        d_valA = D_valP;
        end
        else if(d_srcA == e_dstE) begin
        d_valA = e_valE;
        end
        else if(d_srcA == M_dstM) begin
        d_valA = m_valM;
        end
        else if(d_srcA == W_dstM) begin
        d_valA = W_valM;
        end
        else if(d_srcA == M_dstE) begin
        d_valA = M_valE;
        end
```

```
        else if(d_srcA == W_dstE) begin
        d_valA = W_valE;
        end
        else begin
            d_valA = d_rvalA;
        end

        // forwarding B
        if(d_srcB == e_dstE) begin
        d_valB = e_valE;
        end
        else if(d_srcb == M_dstM) begin
        d_valB = m_valM;
        end
        else if(d_srcB == W_dstM) begin
        d_valB = W_valM;
        end
        else if(d_srcB == M_dstE) begin
        d_valB = M_valE;
        end
        else if(d_srcB == W_dstE) begin
        d_valB = W_valE;
        end
        else begin
            d_valB = d_rvalB;
        end
    end

    regE proc_regE(clk, E_bubble, d_stat, d_icode, d_ifun, d_rA, d_rB, d_valA, d_valB, d_valC, d_valP,e_stat, e_icode, e_ifun, e_rA, e_rB, 

    always @(posedge clk) begin
        if (W_icode == 4'b0000) begin // halt
        end
        else if (W_icode == 4'b0001) begin // nop
        end
        else if ((W_icode == 4'b0010) && (w_Cnd == 1)) begin // cmovXX
            registers[W_dstE] = W_valE;
        end
        else if (W_icode == 4'b0011) begin // irmovq
            registers[W_dstE] = W_valE;
        end
        else if (W_icode == 4'b0100) begin // rmmovq
        end
        else if (W_icode == 4'b0101) begin // mrmovq
            registers[W_dstM] = W_valM;
        end
        else if (W_icode == 4'b0110) begin // OPq
            registers[W_dstE] = W_valE;
        end
        else if (W_icode == 4'b0111) begin // jXX
        end
        else if (W_icode == 4'b1000) begin // call
            registers[W_dstE] = W_valE;
        end
        else if (W_icode == 4'b1001) begin // ret
            registers[W_dstE] = W_valE;
        end
        else if (W_icode == 4'b1010) begin // pushq
            registers[W_dstE] = W_valE;
        end
        else if (W_icode == 4'b1011) begin // popq
            registers[W_dstE] = W_valE;
            registers[W_dstM] = W_valM;
        end
    end
endmodule
```

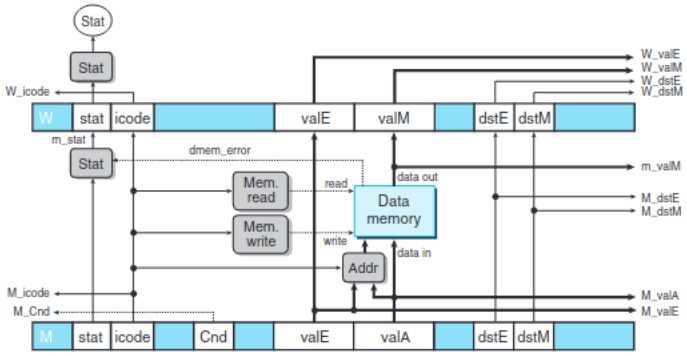
3) Execute




```
module PIPEexecute(clk, E_stat, E_icode, E_ifun, E_dstE, E_dstM, m_stat, W_stat, icode, ifun, valA, valB, valC, valE, cnd, ZF, SF, OF);
    input clk, M_bubble;
    input [3:0] E_stat, E_icode, E_ifun, E_dstE, E_dstM, m_stat, W_stat;
    input [63:0] E_valA, E_valB, E_valC;
    output reg M_Cnd, e_Cnd = 1, ZF, OF, SF;
    output reg [3:0] M_stat, M_icode, M_dstE, M_dstM, e_dstE;
    output reg [63:0] M_valA, M_valE, e_valE;

    initial begin
      e_valE = 0;
      ZF = 0;
      OF = 0;
      SF = 0;
    end

    reg [1:0] control;
    reg signed [63:0] aluInputA, aluInputB;
    wire tempZF, tempSF, tempOF;
    wire overflow;
    wire signed [63:0] out;
    initial begin
      control = 0;
      aluInputA = 0;
      aluInputB = 0;
    end

    alu execute_ALU(aluInputA, aluInputB, control, out, overflow, tempZF, tempOF, tempSF);

    always @(*) begin
        if (E_icode == 4'b0010 || E_icode == 4'b0111) begin
            if (iE_fun == 4'b0000) begin
            E_Cnd = 1;
          end
          else if (E_ifun == 4'b0001) begin
            e_Cnd = (SF ^ OF) | ZF;
          end
          else if (E_ifun == 4'b0010) begin
            e_Cnd = SF^OF;
          end
          else if (E_ifun == 4'b0011) begin
            e_Cnd = ZF;
          end
          else if (E_ifun == 4'b0100) begin
            e_Cnd = ~ZF;
          end
          else if (E_ifun == 4'b0101) begin
            e_Cnd = ~(SF^OF);
          end
          else if (E_ifun == 4'b0110) begin
            e_Cnd = ~(SF^OF)&(~ZF);
          end
          e_dstE = e_Cnd ? E_dstE : 4'b1111;
        end
        else begin
            e_dstE = E_dstE;
        end
    end

    always @(*) begin
      // calculating values using ALU
```

```
      if (E_icode == 4'b0000) begin // halt
      end
      else if (E_icode == 4'b0001) begin // nop
      end
      else if (E_icode == 4'b0010) begin // cmovXX
          aluInputA = E_valA;
          aluInputB = 0;
          control = 2'b00;
          e_valE = out;
      end
      else if (icode == 4'b0011) begin // irmovq
          aluInputA = E_valC;
          aluInputB = 0;
          control = 2'b00;
          e_valE = out;
      end
      else if (icode == 4'b0100) begin // rmmovq
          aluInputA = E_valC;
          aluInputB = E_valB;
          control = 2'b00;
          e_valE = out;
      end
      else if (icode == 4'b0101) begin // mrmovq
          aluInputA = E_valC;
          aluInputB = E_valB;
          control = 2'b00;
          e_valE = out;
      end
      else if (icode == 4'b0110) begin // OPq
          aluInputA = E_valA;
          aluInputB = E_valB;
          if (E_ifun == 4'b0000) begin // ADD
            control = 2'b00;
          end
          else if (E_ifun == 4'b0001) begin // SUB
            control = 2'b01;
          end
          else if (E_ifun == 4'b0010) begin // AND
            control = 2'b10;
          end
          else if (E_ifun == 4'b0011) begin // XOR
            control = 2'b11;
          end
          e_valE = out;
      end
      else if (icode == 4'b0111) begin // jXX
      end
      else if (icode == 4'b1000) begin // call
          aluInputA = -4'b1000;
          aluInputB = E_valB;
          control = 2'b00;
          e_valE = out;
      end
      else if (icode == 4'b1001) begin // ret
          aluInputA = 4'b1000;
          aluInputB = E_valB;
          control = 2'b00;
          e_valE = out;
      end
      else if (icode == 4'b1010) begin // pushq
          aluInputA = -4'b1000;
          aluInputB = E_valB;
          control = 2'b00;
          e_valE = out;
      end
      else if (icode == 4'b1011) begin // popq
          aluInputA = 4'b1000;
          aluInputB = E_valB;
          control = 2'b00;
          e_valE = out;
      end
      // setting flags
     assign ZF = tempZF;
      assign OF = tempOF;
      assign SF = tempSF;
    end

    regM proc_regM(clk, M_bubble, e_stat, e_icode, e_rA, e_rB, e_valA, e_valB, e_valC, e_valE, e_valP,e_Cnd, m_stat, m_icode, m_rA, m_rB, m

endmodule
```

4) Memory



Pipeline Control Logic

Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal