INTRO TO PROCESSOR ARCHITECTURE

FINAL PROJECT

ANANYA VARMA – 2022102064

VISHNU PRIYA - 2022102023

AIM:

- To develop a processor architecture based on the Y86-64 design.
- It should contain both a sequential model as well as a 5-stage pipelined model.

TYPES OF IMPLEMENTATIONS:

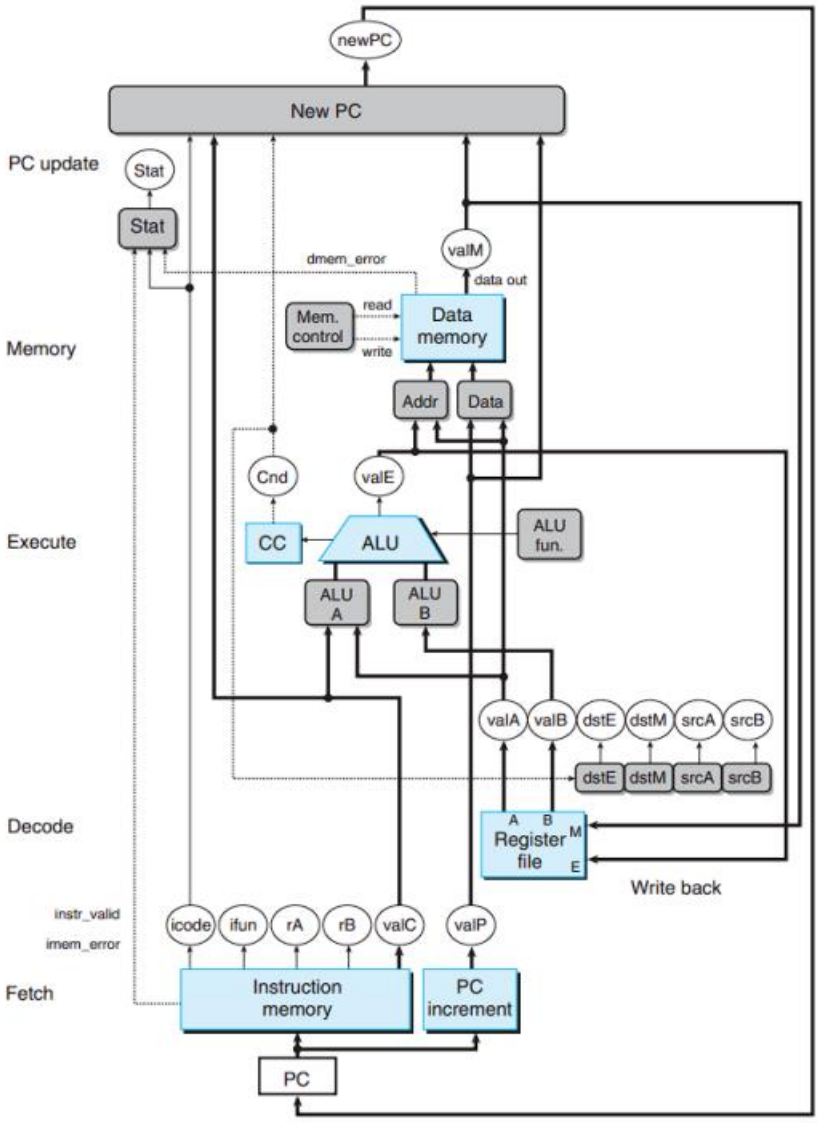
- **SEQUENTIAL PROCESSOR**:
  In this type of implementation, we have the complete cycle for one instruction and then we will have to go the next instruction.
  It is a time taking process but will be easy to implement.
- **PIPELINING:**
  We add the pipeline registers in between each of the registers.
  We then need to take the clock of the longest time taking process and go on taking the clock cycles at that rate so that we can complete many instructions simultaneously of different instructions.

SEQUENTIAL IMPLEMENTATION

- The sequential part of the y86 processor is a basic implementation where instructions are executed one at a time in sequence.
- Each instruction is fetched from memory, decoded, and executed before moving on to the next instruction.
- The sequential implementation of the y86 processor is a simple and efficient approach that can be used to teach the basics of computer architecture and assembly language programming.
- Its advantages include simplicity, predictability, and efficiency for certain types of applications.

In the sequential processor we have many stages which takes place for a particular instruction one after the another.

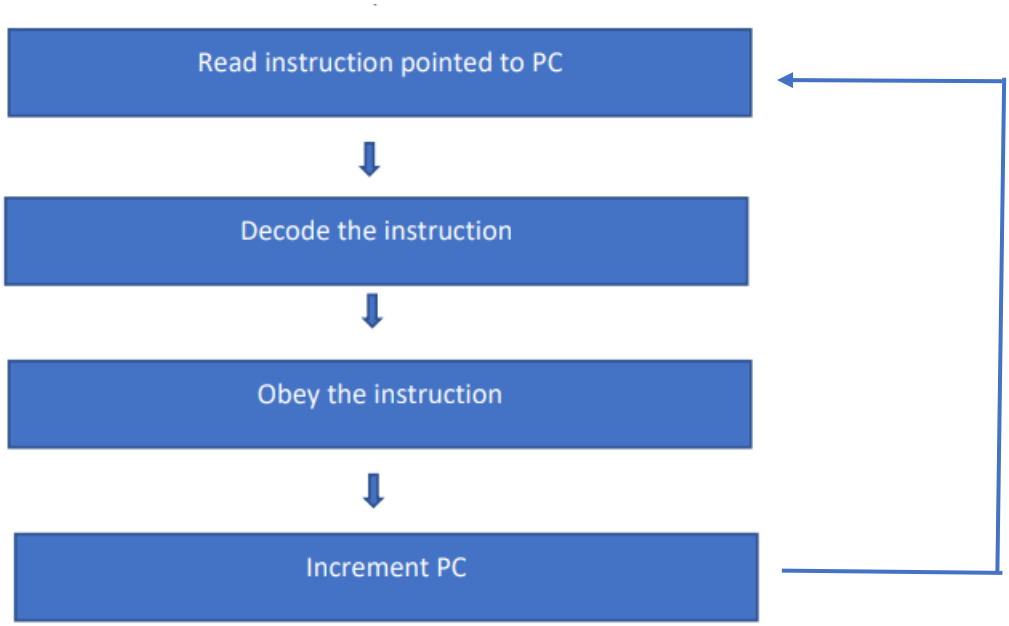
BLOCK DIAGRAM:



STAGE	REGISTERS	DESCRIPTION
1. Fetch	• icode, ifun • rA, rB • valC • valP	Read Instruction byte Read Register byte Read constant word Compute next PC
2. Decode	• ValA, srcA • ValB ,srcB	Read operand A Read operand B
3. Execute	• ValE • Cnd	Perform ALU operation Set/use condition codes
4. Memory	• ValM	Memory read/write
5. Writeback	• DstE • DstM	Writeback ALU result Writeback Mem result

6. PC Update	• PC	Update PC

FLOW OF SEQ:



STAGES:

The Y86 processor is a simple 8-stage pipeline processor with six major blocks:

FETCH:

- reads bytes of an instruction from memory by using PC value.
- It extracts icode and ifun from the instruction memory i.e. first four bits represents the icode and next four represents ifun.
- It also fetches ValC and it helps find ValP.

FLOW OF THE CODE:

- For the implementation of the fetch operation, we need the instruction memory. So firstly, we define the instruction memory that is of 1024 array of 8 bits (1 byte) each.

```
reg [7:0] inst_mem [0:1023];
```

- Fetch the 10 bytes (80 bits) of memory into some variable which is required or the maximum possible length of any instruction and store it as an array taking the values from the inst_mem [PC] to inst_mem [PC+9].

```
inst={
    inst_mem[PC],
    inst_mem[PC+1],
    inst_mem[PC+2],
    inst_mem[PC+3],
    inst_mem[PC+4],
    inst_mem[PC+5],
    inst_mem[PC+6],
    inst_mem[PC+7],
    inst_mem[PC+8],
    inst_mem[PC+9]
};
```

- In the fetch instruction we need to first find the value of the icode and ifun which are represented as follows,

```
icode = inst[0:3];
ifun = inst[4:7];
```

- Now we need to decide whether we need the values of the rA and rB and the valC from the values of the icode.

FETCH FOR HALT OPERATION:

- If the value of the icode = 0, then it is a halt operation
- We do not want any of the variables rA and rB and valC
- Therefore, we update the value of the program counter to point out the next instruction, which is present in the instruction code which is of length 1 byte or 8 bits. (4 for icode and 4 for ifun). Therefore, we have

```
  task halt_operation;
begin
  hlt = 1;
  valP = PC + 64'd1;
end
  endtask
```

FETCH FOR NOP OPERATION:

- If the value of the icode = 1, then it is a NOP operation
- We do not want any of the variables rA and rB and valC and therefore we update the value of the program counter
- Reads bytes of an instruction from memory by using PC value.
- It extracts icode and ifun from the instruction i.e., first four bits represents the icode and next four represents the ifun.

- It also fetches valC and it helps in finding valP to point out the next instruction, which is present in the instruction code which is of length 1 byte or 8 bits. (4 for icode and 4 for ifun). valP = PC + 64'd1;

```
task nop_operation;
begin
  valP = PC + 64'd1;
end
endtask
```

FETCH FOR CMOVXX OPERATION:

- If the value of the icode = 2, then the function is conditional move operation where there is requirement of reading the value of the rA and rB for the purpose of doing the move operation if the condition is satisfied.
- Therefore, the value of the new program counter will be due to icode (4 bits), ifun (4 bits) and code of the register rA and rB which is of total size of the 4 bits each.
- Therefore, total of 2 bytes. valP = PC + 64'd2;

```
task cmovxx_operation;
begin
  rB = inst[12:15];
  rA = inst[8:11];
  valP = PC + 64'd2;
end
endtask
```

FETCH FOR IRMOV OPERATION:

- If the value of the icode = 3, then the function is IRMOV operation where there is requirement of reading the value of the rA and rB for the purpose of doing the move operation and the value of the valC for the value of the element stored there.
- Therefore, the value of the new program counter will be due to icode (4 bits), ifun (4 bits) and code of the register F and rB which is of total size of the 4 bits each and also the valC which is the immediate which is of 64 bits or 8 bytes.
- Therefore, total of 10 bytes. valP = PC + 64'd10;

```
task irmov_operation;
begin
  valC = inst[16:79];
  valP = PC + 64'd10;
  rB = inst[12:15];
end
endtask
```

FETCH FOR RMMOV OPERATION:

- If the value of the icode = 4, then the function is RMMOV operation where there is requirement of reading the value of the rA and rB for the purpose of doing the move operation and also the value of the valC for finding the address of the destination.
- Therefore, the value of the new program counter will be due to icode (4 bits), ifun (4 bits) and code of the register rA and rB which is of total size of the 4 bits each and also the valC which is of 64 bits or 8 bytes.
- Therefore, total of 10 bytes. valP = PC + 64'd10;

```
task rmmov_operation;
 begin
      valC = inst[16:79];
      valP = PC + 64'd10;
      rA = inst[8:11];
      rB = inst[12:15];
    end
endtask
```

FETCH FOR MRMOV OPERATION:

- If the value of the icode = 5, then the function is MRMOV operation where there is requirement of reading the value of the rA and rB for the purpose of doing the move operation and the value of the valC for finding the address of the source.
- Therefore, the value of the new program counter will be due to icode (4 bits), ifun (4 bits) and code of the register rA and rB which is of total size of the 4 bits each and also the valC which is of 64 bits or 8 bytes.
- Therefore, total of 10 bytes. valP = PC + 64'd10;

```
task mrmov_operation;
 begin
      valC = inst[16:79];
      valP = PC + 64'd10;
      rA = inst[8:11];
      rB = inst[12:15];
    end
endtask
```

FETCH FOR OPQ OPERATION:

- If the value of the icode = 6, then it is the ALU operation where there is requirement of reading the value of the rA and rB for finding the values of the operands.
- Therefore, the value of the new program counter will be due to icode (4 bits), ifun (4 bits) and code of the register rA and rB which is of total size of the 4 bits each.
- Therefore, total of 2 bytes. valP = PC + 64'd2;

```
task OPq_operation;
 begin
      valP = PC + 64'd2;
      rA = inst[8:11];
      rB = inst[12:15];
    end
endtask
```

FETCH FOR JUMP OPERATION:

- If the value of the icode = 7, then the function is jump operation where we have some condition and if that condition satisfies, we need to jump to the destination which is defined by the destination or the valC which is of 64 bits.
- Therefore, the value of the new program counter will be due to icode (4 bits), ifun (4 bits) and also the valC which is of 64 bits or 8 bytes.
- Therefore, total of 9 bytes. valP = PC + 64'd9;

```
task jXX_operation;
 begin
      valC = inst[8:71];
      valP = PC + 64'd9;
    end
endtask
```

FETCH FOR CALL OPERATION:

- If the value of the icode = 8, then the function is call operation where we require valC which is of 64 bits which denotes the destination of the call function.
- Therefore, the value of the new program counter will be due to icode (4 bits), ifun (4 bits) and also the valC which is of 64 bits or 8 bytes.
- Therefore, total of 9 bytes. valP = PC + 64'd9;

```
task call_operation;
 begin
      valC = inst[8:71];
      valP = PC + 64'd9;
    end
endtask
```

FETCH FOR RETURN OPERATION:

- If the value of the icode = 9, then the function is return operation where we just pop the value and return the address.
- Therefore, we do not valP = PC + 64'd1;

```
task ret_operation;
 begin
      valP = PC + 64'd1;
    end
endtask
```

CODE:

```
module fetch(clk,icode,ifun,rA,rB,valC,valP,hlt,inst_valid,mem_error,PC);

input clk;
input [63:0] PC;


output reg hlt;
output reg inst_valid;
output reg mem_error;


output reg [63:0] valP;
output reg [63:0] valC;

output reg [3:0] rA;
output reg [3:0] rB;

output reg [3:0] icode;
```

```
output reg [3:0] ifun;

reg [7:0] inst_mem [0:1023];
reg [0:79] inst;

initial begin
    inst_mem[0] = 8'b00110000;  //icode:ifun
    inst_mem[1] = 8'b11110001;  // rA:rB
    inst_mem[2] = 8'b00000000;  //dest
    inst_mem[3] = 8'b00000000;
    inst_mem[4] = 8'b00000000;
    inst_mem[5] = 8'b00000000;
    inst_mem[6] = 8'b00000000;
    inst_mem[7] = 8'b00000000;
    inst_mem[8] = 8'b00000001;
    inst_mem[9] = 8'b00000000;

    inst_mem[10] = 8'b00110000;  //icode:ifun
```

```
  inst_mem[11] = 8'b11110010;  // rA:rB
  inst_mem[12] = 8'b00000000;  //dest
  inst_mem[13] = 8'b00000000;
  inst_mem[14] = 8'b00000000;
  inst_mem[15] = 8'b00000000;
  inst_mem[16] = 8'b00000000;
  inst_mem[17] = 8'b00000000;
  inst_mem[18] = 8'b00000010;
  inst_mem[19] = 8'b00000000;

  inst_mem[20] = 8'b01100001;  //opq:addq
  inst_mem[21] = 8'b00010010;  // rcx:rdx

  // inst_mem[22] = 8'b00010000;  //nop
  // inst_mem[23] = 8'b00010000;  //nop
  // inst_mem[24] = 8'b00000000; // halt
```

```
end

  task halt_operation;
begin
  hlt = 1;
  valP = PC + 64'd1;
end
  endtask

task nop_operation;
begin
  valP = PC + 64'd1;
end
endtask

task cmovxx_operation;
begin
  rB = inst[12:15];
  rA = inst[8:11];
```

```
  valP = PC + 64'd2;
end
endtask

task irmov_operation;
begin
  valC = inst[16:79];
  valP = PC + 64'd10;
  rB = inst[12:15];
end
endtask



task rmmov_operation;
 begin
```

```
      valC = inst[16:79];
      valP = PC + 64'd10;
      rA = inst[8:11];
      rB = inst[12:15];
    end
endtask

task mrmov_operation;
 begin
      valC = inst[16:79];
      valP = PC + 64'd10;
      rA = inst[8:11];
      rB = inst[12:15];
    end
endtask
```

```
task OPq_operation;
 begin
      valP = PC + 64'd2;
      rA = inst[8:11];
      rB = inst[12:15];
    end
endtask

task jXX_operation;
 begin
      valC = inst[8:71];
      valP = PC + 64'd9;
    end
endtask

task call_operation;
```

```
 begin
        valC = inst[8:71];
        valP = PC + 64'd9;
     end
endtask

task ret_operation;
 begin
        valP = PC + 64'd1;
     end
endtask

task push_operation;
 begin
        valP = PC + 64'd2;
        rA = inst[8:13];
        rB=inst[12:15];
```

```
     end
endtask

task pop_operation;
 begin
       rA = inst[8:13];
       rB=inst[12:15];
       valP = PC + 64'd2;
     end
endtask

always @ (posedge clk)
begin

    mem_error = 0;           // finding if the given instruction is within the
```

```
    if(PC>1023)              // instruction memory or not
    begin
      mem_error = 1;
    end

    inst_valid=1;

    inst={
        inst_mem[PC],
        inst_mem[PC+1],
        inst_mem[PC+2],
        inst_mem[PC+3],
        inst_mem[PC+4],
        inst_mem[PC+5],
        inst_mem[PC+6],
        inst_mem[PC+7],
        inst_mem[PC+8],
```

```
        inst_mem[PC+9]
    };

    icode = inst[0:3];
    ifun = inst[4:7];

    case (icode)
        4'b0000: halt_operation;
        4'b0001: nop_operation;
        4'b0010: cmovxx_operation;
        4'b0011: irmov_operation;
        4'b0100: rmmov_operation;
        4'b0101: mrmov_operation;
        4'b0110: OPq_operation;
        4'b0111: jXX_operation;
        4'b1000: call_operation;
```

```
        4'b1001: ret_operation;
        4'b1010: push_operation;
        4'b1011: pop_operation;
        default: inst_valid = 0;
    endcase

end

endmodule
```

DECODE AND WRITE BACK:

- Decode will read the two operands from the register file. And it will give values ValA and ValB. Here ValA represents the value in the first register file and ValB represents the value in the second register file.
- It writes the two results back into the register file.

FLOW OF THE CODE:

- We came up with a logic where we can process both decode and write back in one file.
- We do this because we access the register files in both stages.
- So here we initiated 15 registers such that those can be used for storing data in decode stage and those can be reused again for getting the value back in write back stage.

```
output reg [63:0] rax,
output reg [63:0] rcx,
output reg [63:0] rdx,
output reg [63:0] rbx,
output reg [63:0] rsp,
output reg [63:0] rbp,
output reg [63:0] rsi,
output reg [63:0] rdi,
output reg [63:0] r8,
output reg [63:0] r9,
output reg [63:0] r10,
output reg [63:0] r11,
output reg [63:0] r12,
output reg [63:0] r13,
output reg [63:0] r14
```

DECODE:

- The main function of the decode stage is to assign the value that to be stored in the register as ValA and ValB
- We also make use of icode for tracking the certain operation for decode stage.

DECODE FOR NOP OPERATION:

- If icode is 0 or 1 then we observe no change in decode stage as they are halt and NOP operations.
- NOP is no operation.

DECODE FOR CMOVXX OPERATION:

- If icode == 2, then the operation is cmovxx.
- So, in this case the value of that is in register rA should be stored in ValA.
- ValA <-- R[rA].

```
4'b0010: begin  // cmove operation
    valA = reg_mem[rA];
end
```

DECODE FOR RMMOVQ OPERATION:

- If icode == 4, then the operation is RMMOVQ.
- So, in this case the value the value that is in register rA should be stored in ValA and the value that is in register rB should be stored in ValB.
- So, ValA<-- R[rA] and ValB<-- R[rB]

```
4'b0100: begin  // rmmove operation
    valA = reg_mem[rA];
    valB = reg_mem[rB];
end
```

DECODE FOR MRMOVQ OPERATION:

- If icode == 5, then the operation is MRMOVQ.
- So, in this case the value the value that is in register rA should be stored in ValA and the value that is in register rB should be stored in ValB.
- So, ValA<-- R[rA] and ValB<-- R[rB].

```
4'b0101: begin  // mrmove operation
    valB = reg_mem[rB];
end
```

DECODE FOR OPQ OPERATION:

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

DECODE FOR CALL OPERATION:

- If icode is 8, then the operation is call.
- So, in this case the we need to read the stack pointer and the data in the register %esp is stored in ValB.
- ValB<-- R[%esp]

```
4'b1000: begin  // call operation
    valB = reg_mem[4];
end
```

DECODE FOR RETURN OPERATION:

- If icode is 9, then the operation is ret (return).
- So, in this case we store the value that is present in %esp in ValA and ValB.
- ValA <-- R[%esp] and ValB<--R[%esp].

```
4'b1001: begin  // return operation
    valA = reg_mem[4];
    valB = reg_mem[4];
end
```

DECODE FOR PUSH OPERATION:

- If icode is 10, then the operation is push.
- So, in this case first we decrement the address and then try to store to value in the pointer.
- So, the updated one after the decrement is stored in ValB

- ValB<--R[%esp] and the value of the register rA is stored in ValA.

```
4'b1010: begin  // push operation
    valA = reg_mem[rA];
    valB = reg_mem[4];
end
```

DECODE FOR POP OPERATION:

- If icode is 11, then the operation is pop.
- In this case, we first pop out the value and then increment back the address.
- So, ValA<--R[%esp] and ValB<--R[%esp].

```
4'b1011: begin //pop operation
    valA = reg_mem[4];
    valB = reg_mem[4];
end
```

WRITEBACK:

- The main use of this stage is to write back the value in the specified register at the end of all stages.
- It was told before that we initialised 15 registers and used them for storing the values. As shown below:

```
rax = reg_mem[0];
rcx = reg_mem[1];
rdx = reg_mem[2];
rbx = reg_mem[3];
rsp = reg_mem[4];
rbp = reg_mem[5];
rsi = reg_mem[6];
rdi = reg_mem[7];
r8 = reg_mem[8];
r9 = reg_mem[9];
r10 = reg_mem[10];
r11 = reg_mem[11];
r12 = reg_mem[12];
r13 = reg_mem[13];
r14 = reg_mem[14];
```

WRITEBACK FOR CMOVXX OPERATION:

- If icode is 2 then this is cmovxx operation.
- During write back stage ValE value is stored back in register.
- R[rB]<-- ValE.

- ValE is taken from the execute stage.

```
4'b0010: begin // cmove operation
    if (cnd == 1'b1) reg_mem[rB] = valE;
end
```

WRITEBACK FOR IRMOVQ OPERATION:

- If icode is 3, then it is irmovq operation.
- During this stage, ValE is passed through the register Rb.
- R[rB]<-- ValE

```
4'b0011: begin        //
    reg_mem[rB] = valE;
end
```

WRITEBACK FOR MRMOVQ OPERATION:

- If icode is 5, then it is mrmovq operation (mrmovq D(rB) rA).
- So, the rA register will store the final result.
- Thus, ValE is written back to register rA

```
4'b0101: begin   // mrmovq D(rB) rA
    reg_mem[rA] = valM;     // Here
end
```

WRITEBACK FOR OPQ OPERATION:

- If icode is 6, then it is Opq operation (Opq rA rB).
- So, the rB register will store the final result.
- Thus, ValE is written back to register rB.

```
4'b0110: begin  // OPq rA rB
    reg_mem[rB] = valE;
end
```

WRITEBACK FOR CALL OPERATION:

- If icode is 8, then it is call operation (call Dest).
- So, reg_mem [4] will store the value in ValE as reg_mem [4] is %esp.

```
4'b1000: begin  // call operation
    valB = reg_mem[4];
end
```

WRITEBACK FOR RETURN OPERATION:

- If icode is 9, then it is ret (return) operation.
- So, reg_mem[4] will store the value in ValE as reg_mem[4] is %esp.

```
4'b1001: begin  // return operation
    valA = reg_mem[4];
    valB = reg_mem[4];
end
```

WRITEBACK FOR PUSHQ OPERATION:

- If icode is 10, then it is pushq operation.
- So, here the value ValE is stored in %esp i.e., stack pointer.
- As the value is pushed on to the stack

```
    4'b1010: begin  // push operation
        valA = reg_mem[rA];
        valB = reg_mem[4];
    end
```

WRITEBACK FOR POP OPERATION:

- If icode is 11, then it is pop operation.
- So, here the value ValE is stored in %esp i.e., stack pointer.
- As the value is popped out from stack.

```
4'b1011: begin //pop operation
    valA = reg_mem[4];
    valB = reg_mem[4];
end
```

CODE:

```
module decode(
    input clk,
    input cnd,

    input [3:0] icode,

    input [3:0] rA,
    input [3:0] rB,

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

    input [63:0] valC,
    input [63:0] valE,
    input [63:0] valM,

    output reg [63:0] rax,
    output reg [63:0] rcx,
    output reg [63:0] rdx,
    output reg [63:0] rbx,
    output reg [63:0] rsp,
    output reg [63:0] rbp,
    output reg [63:0] rsi,
    output reg [63:0] rdi,
```

```
    output reg [63:0] r8,
    output reg [63:0] r9,
    output reg [63:0] r10,
    output reg [63:0] r11,
    output reg [63:0] r12,
    output reg [63:0] r13,
    output reg [63:0] r14

);

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

initial begin

    reg_mem[0] = rax;
    reg_mem[1] = rcx;
    reg_mem[2] = rdx;
    reg_mem[3] = rbx;
    reg_mem[4] = rsp;
    reg_mem[5] = rbp;
    reg_mem[6] = rsi;
    reg_mem[7] = rdi;
```

```
    reg_mem[8] = r8;
    reg_mem[9] = r9;
    reg_mem[10] = r10;
    reg_mem[11] = r11;
    reg_mem[12] = r12;
    reg_mem[13] = r13;
    reg_mem[14] = r14;
end

always @(*) begin
    case (icode)

        4'b0010: begin  // cmove operation
            valA = reg_mem[rA];
        end

        4'b0100: begin  // rmmove operation
            valA = reg_mem[rA];
            valB = reg_mem[rB];
        end

        4'b0101: begin  // mrmove operation
```

```
            valB = reg_mem[rB];
        end

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

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

        4'b1001: begin  // return operation
            valA = reg_mem[4];
            valB = reg_mem[4];
        end

        4'b1010: begin  // push operation
            valA = reg_mem[rA];
            valB = reg_mem[4];
```

```
        end

        4'b1011: begin //pop operation
            valA = reg_mem[4];
            valB = reg_mem[4];
        end


    endcase


    rax = reg_mem[0];
    rcx = reg_mem[1];
    rdx = reg_mem[2];
    rbx = reg_mem[3];
    rsp = reg_mem[4];
    rbp = reg_mem[5];
    rsi = reg_mem[6];
    rdi = reg_mem[7];
    r8 = reg_mem[8];
```

```
    r9 = reg_mem[9];
    r10 = reg_mem[10];
    r11 = reg_mem[11];
    r12 = reg_mem[12];
    r13 = reg_mem[13];
    r14 = reg_mem[14];
end

always @(negedge clk) begin
    case (icode)
        4'b0010: begin // cmove operation
            if (cnd == 1'b1) reg_mem[rB] = valE;
        end

        4'b0011: begin        // irmovq $0xx rB
            reg_mem[rB] = valE;       // Here the rB register will store the final result
        end

        4'b0101: begin   // mrmovq D(rB) rA
            reg_mem[rA] = valM;     // Here the rA register will store the final result
        end
```

```
    4'b0110: begin  // OPq rA rB
        reg_mem[rB] = valE;     // Here the rB register will store the final result
    end

    4'b1000: begin  // call Dest
        reg_mem[4] = valE;       // Here reg_mem[4] is the %esp(stack pointer) .Update stack pointer
    end

    4'b1001: begin  // ret
        reg_mem[4] = valE;      // reg_mem[4] is the %esp and we update the stack pointer
    end

    4'b1010: begin  // pushq
        reg_mem[4] = valE;     // In push, we first decrement the address and then push the data.
    end

    4'b1011: begin  // popq
        reg_mem[4] = valE;   // In this, first we pop out the data from the stack and then increment the a
        reg_mem[rA] = valM;   // So, the popped-out data is again restored in register rA
    end
```

```
     default: begin

     end
  endcase

  // Update register values
  rax = reg_mem[0];
  rcx = reg_mem[1];
  rdx = reg_mem[2];
  rbx = reg_mem[3];
  rsp = reg_mem[4];
  rbp = reg_mem[5];
  rsi = reg_mem[6];
  rdi = reg_mem[7];
  r8 = reg_mem[8];
  r9 = reg_mem[9];
  r10 = reg_mem[10];
  r11 = reg_mem[11];
  r12 = reg_mem[12];
  r13 = reg_mem[13];
```

```
    r13 = reg_mem[13];
    r14 = reg_mem[14];
end

endmodule
```

EXECUTE:

- By using ifun, execute will perform the respective operation.
- Condition codes are set.
- For jump instruction, tests condition code and branch condition to determine if branch should be taken or not.

• In this stage which is the third stage of the operation we will have the calculation part. In this part we will set the flags wherein we will include the ALU also for the calculation part.

• In this stage we take the values of the ValA and ValB from the previous parts and then put into the ValE or by using the valC and then we set the values of the ValE.

FLAGS:

The flags are more important to perform compare operations or the conditional operations in the processor and therefore after any of the ALU operations the flags are going to be set and then we perform the operations in the next stages.

1. SIGN FLAG – It denotes whether the value of the signed integer is positive or negative depending upon the value of the most significant digit. Hence if the value of the MSD is 1 then the value as negative. And if the value of the MSD is zero then the MSD is 0.

2. ZERO FLAG – It denotes whether the output answer is zero or not. If the output is 1 then the value of the of the output is 0 else the value of the output is 0.

3. OVERFLOW FLAG – If the value of the inputs are both positive and value of the output is negative or we have another case in which the value of both the inputs are negative and we have the output as positive and therefore in these cases we pull the overflow flag to be one.

FLOW OF THE CODE:

EXECUTE FOR IRMOV OPERATION:

- If the value of the icode = 3, then the function is IRMOV.
- Here we need to set the value of the ValE to be valC and then we do the further operation in the next step.

EXECUTE FOR RMMOV OPERATION:

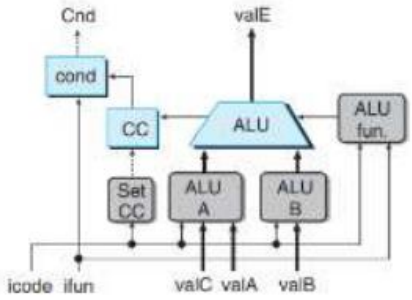
- If the value of the icode =4, then the function is rmmov
- The value of the ValE = ValB + ValC.

EXECUTE FOR MRMOV OPERATION:

- If the value of the icode =5, then the function is mrmov
- We have ValE = ValB + ValC.

EXECUTE FOR OPQ OPERATION:

- If the value of the icode = 6, then we have the function opq then we have to perform the required operation and store them in the value of the valE depending on the value of the ifun.
- If the value of the ifun to be any of 0 to 3 then the function is add, subtract, add and then xor, and then we set the flags of the sign flag, overflow flag, zero flag as shown in the appendix for each operation.
- This part has been done in the ALU operations.



- If Ifun=0 ,then add operation is done.
- If Ifun =1, then sub operation is done.
- If Ifun=2, the and operation is done.
- If Ifun=3, then xor operation is done.

EXECUTE FOR JUMP FUNCTION:

- If the value of the icode =7, then the function is jump function
- Here we find whether the condition is satisfied or not
- We return the conditions to the next steps.

EXECUTE FOR CALL FUNCTION:

- If the value of the icode= 8, then the function is the call function
- We basically need to update the stack pointer to add the new address into the stack.
- Therefore, we decrease the value of the stack pointer by 8.

EXECUTE FOR RETURN OPERATION:

- If the value of the icode = 9, then the function is the return function
- Here we need to pop the value which is present in the bottom of the stack
- We need to use that to go to that pointer.

EXECUTE FOR PUSH OPERATION:

- If the icode is = A, then the function is the push function and here we need to add an element into the stack.
- We add the new element to the bottom of the stack

- Therefore, we decrease the value of the stack pointer by 8 and then in the further steps we put the value of the valC into that place where we have created in the stack.

EXECUTE FOR POP OPERATION:

- If the value of icode is B, then the function is the pop operation.
- We have increased the value of the stack pointer by 8 and then we need to update the new stack pointer value.

CODE:

```
module FullAdderBit(A, B, Cin, Sum, Cout);
    input A, B, Cin;
    output Sum, Cout;

    wire temp, P, Q, R;

    xor G0(temp, A, B);
    xor G1(Sum, temp, Cin);
    and G2(P, A, B);
    and G3(Q, B, Cin);
    and G4(R, A, Cin);
    or G5(Cout, P, Q, R);
endmodule

module Add(A, B, Sum, Cout,carry_overflow);
    input [63:0] A;
    input [63:0] B;
    output [63:0] Sum;
    output Cout,carry_overflow;

    wire [64:0] Carry;

    assign Carry[0] = 1'b0;

    genvar i;
    generate
        for (i = 0; i < 64; i = i + 1) begin
            FullAdderBit adderBit(
                .A(A[i]),
                .B(B[i]),
                .Cin(Carry[i]),
                .Sum(Sum[i]),
                .Cout(Carry[i + 1])
            );
        end
    endgenerate

    assign Cout=Carry[64];
    xor g6(carry_overflow,Cout,Carry[63]);


endmodule

module Sub(A, B, Sum, Cout,carry_overflow);
    input [63:0] A;
    input [63:0] B;

    output [63:0] Sum;
    output Cout,carry_overflow;

    wire [64:0] Carry;
    wire [63:0] B_com;
```

```
  assign Carry[0] = 1'b1;

  genvar j;
  generate
      for (j = 0; j < 64; j = j+ 1) begin
          xor g1(B_com[j],B[j],Carry[0]);
      end
  endgenerate


  genvar i;
  generate
      for (i = 0; i < 64; i = i + 1) begin
          FullAdderBit adderBit(
              .A(A[i]),
              .B(B_com[i]),
              .Cin(Carry[i]),
              .Sum(Sum[i]),
              .Cout(Carry[i + 1])
          );
      end
  endgenerate

  assign Cout=Carry[64];
  xor g7(carry_overflow,Cout,Carry[63]);
```

```
endmodule

module And(
    input [63:0] A,
    input [63:0] B,
    output [63:0] Y
);

    generate
        genvar i;
        for (i = 0; i < 64; i = i + 1) begin
            and a1(Y[i],A[i],B[i]);
        end
    endgenerate

endmodule

module Xor(
    input [63:0] A,
    input [63:0] B,
    output [63:0] Y
);

    generate
        genvar i;
        for (i = 0; i < 64; i = i + 1) begin
            xor x1(Y[i],A[i],B[i]);
        end
```

```
        end
    endgenerate

endmodule

module ALU(
    input [63:0] A, B,
    input [1:0] S,
    output reg [63:0] result,
    output Cout_add, Cout_sub,carry_overflow_add,carry_overflow_sub
);

    wire [63:0] SA, SSub, AndOut, XorOut;


    Add adder(A, B,SA, Cout_add,carry_overflow_add);
    Sub subtractor(A, B,SSub, Cout_sub,carry_overflow_sub);
    And and1(A, B, AndOut);
    Xor xor1(A, B, XorOut);

    always @(*) begin
        case (S)
            2'b00: result = SA;
            2'b01: result = SSub;
            2'b10: result = AndOut;
            2'b11: result = XorOut;
            default: result = 64'bx;
        endcase
```

```
endmodule

module execute(clk,icode,ifun,valA,valB,valC,valE,Z_F,O_F,S_F,cnd);

input [3:0]icode;
input [3:0]ifun;
input clk;

input [63:0]valA;
input [63:0]valB;
input [63:0]valC;

wire [63:0]add_op_out;
wire [63:0]sub_op_out;
wire [63:0]xor_op_out;
wire [63:0]and_op_out;

wire Overflow_add;
wire Overflow_sub;
wire C_Add;
wire C_Sub;

Add A(valA,valB,add_op_out,C_Add,Overflow_add);
Sub B(valA,valB,sub_op_out,C_Sub,Overflow_sub);

Xor C(valA,valB,xor_op_out);
And D(valA,valB,and_op_out);

output reg [63:0]valE;
```

```
output reg [63:0]valE;
output reg Z_F;
output reg O_F;
output reg S_F;
output reg cnd=0;

wire xor_1;
wire xor_2;
wire out;

xor E(xor_1,valA[63],valB[63]);
xor F(xor_2,valA[63],valE[63]);

xor G(out,O_F,S_F);


always @(*)
begin
  if(clk==1)
  begin

  if(icode==4'b0110) //OPq
  begin
    O_F=0;
    S_F=0;
    Z_F=0;
    case(ifun)
  4'b0000: // Add
    begin
```

```
4'b0000: // Add
  begin
    valE = add_op_out;
    O_F = (xor_1 == 0 && xor_2 == 1);
    S_F = (valE[63] == 1);
    Z_F = (valE[63:0] == 0);
  end

4'b0001: // Sub
  begin
    valE = sub_op_out;
    O_F = (xor_1 == 1 && xor_2 == 1);
    S_F = (valE[63] == 1);
    Z_F = (valE[63:0] == 0);
  end

4'b0010: // And
  begin
    valE = and_op_out;
    S_F = (valE[63] == 1);
    Z_F = (valE[63:0] == 0);
  end

4'b0011: // Xor
  begin
    valE = xor_op_out;
    S_F = (valE[63] == 1);
```

```
    Z_F = (valE[63:0] == 0);
  end
endcase

 end


 else if(icode==4'b0010) //cmoveX
 begin
    case(ifun)
 4'b0000: // rrmovq
  begin
    cnd = 1;
  end

 4'b0001: // cmovle
  begin
    if (out == 1 || Z_F == 1) cnd = 1;
  end

 4'b0010: // cmovl
  begin
    if (out == 1) cnd = 1;
  end

 4'b0011: // cmove
```

```
  begin
    if (Z_F == 1) cnd = 1;
  end

4'b0100: // cmovne
  begin
    if (Z_F == 0) cnd = 1;
  end

4'b0101: // cmovge
  begin
    if (out == 0) cnd = 1;
  end

4'b0110: // cmovg
  begin
    if (out == 0 || Z_F == 0) cnd = 1;
  end

default: // Default case (if ifun doesn't match any known values)
  valE = valA;
endcase

 end
```

```
else if(icode==4'b0111) //jxx
begin
    if(ifun==4'b0000) cnd=1; //jump
    else if(ifun==4'b0001) //jle
    begin
        if(out==1 || Z_F==1) cnd=1;
    end
    else if(ifun==4'b0010) //jl
    begin
      if(out==1) cnd=1;
    end
    else if(ifun==4'b0011) //je
    begin
      if(Z_F==1) cnd=1;
    end
    else if(ifun==4'b0100) //jne
    begin
      if(Z_F==0) cnd=1;
    end
    else if(ifun==4'b0101) //jge
    begin
        if(out==0) cnd=1;
    end
    else if(ifun==4'b0110) //jg
    begin
        if(out==0 || Z_F==0) cnd=1;
    end
```

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

else if (icode == 4'd9)
begin
  valE = valB + 64'd8; // ret instruction
end

else if (icode == 4'd10)
begin
  valE = valB +(-64'd8); // pushq instruction
end

else if (icode == 4'd11)
begin
  valE = valB + 64'd8; // popq instruction
end

else if (icode == 4'd3)
begin
  valE = 64'd0 + valC; // irmovq instruction
end

else if (icode == 4'd4)
begin
  valE = valB + valC; // rmmovq instruction
```

```
  else if (icode == 4'd5)
  begin
    valE = valB + valC; // mrmovq instruction
  end

end
end
endmodule
```

MEMORY:

- Read and write data from/to memory happens.
- Value read referred as ValM
- In the memory stage we generally read or write the values from or to the memory, and therefore the value of ValM is generally assigned in this place

FLOW OF THE CODE:

MEMORY FOR CMOV OPERATION:

- If the function is cmov or the conditional move
- We don't have to do anything with the memory as in this case we are just moving using the conditions.

MEMORY FOR IRMOV OPERATION:

- If the function is irmov then we have to put the value of the valE into the memory of rB to complete the operation.

MEMORY FOR RMMOV OPERATION:

- If the function is rmmov then we have write the value of the ValA into the memory of valE that is given by M[valE] = ValA;

```
    4'b0100: Mem_Mem_output[valE] = valA;    // rmmovq
```

MEMORY FOR MRMOV OPERATION:

- If the function is mrmov then we have to put the value of the ValM into the M[valE].

```
    4'b0101: valM = Mem_Mem_output[valE];    // mrmovq
```

MEMORY FOR CALL OPERATION:

- If the function is call then we decrement the stack pointer by 8 and add the program counter into the stack memory.

```
4'b1000: Mem_Mem_output[valE] = valP;    // call
```

MEMORY FOR RETURN OPERATION:

- If we have the icode as 9 then the function is the return function.
- Therefore, we have to return the value at the memory of the stack pointer(valA) to valM.

```
4'b1001: valM = Mem_Mem_output[valA];    // ret
```

MEMORY FOR PUSH FUNCTION:

- If the value of the icode is 10 then the function is the push function so we have we have to increase the stack pointer to valE which is already done in the execute step
- Here we just put the value of valA into the memory of valE that is M[valE] = valA.

```
4'b1010: Mem_Mem_output[valE] = valA;    // pushq
```

MEMORY FOR POP FUNCTION:

- If the value of the icode is 11 or B in the hexadecimal system then we have the pop function.
- Here in this function, we need to take the increased value of the stack pointer in the previous step.
- Here we need to put the value of the valM in the memory of valA that is M[valA] = valM.

```
4'b1011: valM = Mem_Mem_output[valE];    // popq
```

CODE:

```
module memory(clk, icode, valA, valB, valE, valP, valM, Mem_output);

    reg [63:0] Mem_Mem_output [0:1023];
    integer i;

    input clk;
    input [3:0] icode;
    input signed [63:0] valA, valB;
    input [63:0] valE, valP;
    output reg [63:0] valM, Mem_output;

    initial begin
        for (i = 0; i < 1024; i = i + 1) begin
            Mem_Mem_output[i] = 64'd2;
        end
    end

    always @(*) begin

        case (icode)
            4'b0100: Mem_Mem_output[valE] = valA;   // rmmovq
            4'b0101: valM = Mem_Mem_output[valE];   // mrmovq
            4'b1000: Mem_Mem_output[valE] = valP;   // call
            4'b1001: valM = Mem_Mem_output[valA];   // ret
            4'b1010: Mem_Mem_output[valE] = valA;   // pushq
            4'b1011: valM = Mem_Mem_output[valE];   // popq
        endcase

        Mem_output = Mem_Mem_output[valE];
    end
```

PC UPDATE:

▪ In this function we just update the value of the program counter which stores the value of the address of the next instruction.

▪ The next instruction is valP.

FLOW OF THE CODE:

- Generally, we increase the value of the program counter as discussed in the fetch stage if there is no need to jump to any other instructions.
- In some of the cases which are discussed below requires the program counter to jump a address which is different from the valP.
  - In the case of the jump case or when the value of the icode is 7, then we need to jump to the value of the destination which is given by the value of valC.
  - In case of the call function, we need to update the value of the program counter to the position where the new program counter will tend to which is given by the value of valC.
  - In case of the return function, we need the value of the valM which will be the new value of the program counter as it is the value of the stack and which is to be returned.
  - In all other cases we have the new program counter update to the value of the valP which is calculated in the fetch stage of the program.

CODE:

```
module pc_update(
    input clk,
    input cnd,
    input [63:0] PC,
    input [3:0] icode,
    input [63:0] valC,
    input [63:0] valM,
    input [63:0] valP,
    output reg [63:0] PC_updated
);

always @(*)
begin
    case (icode)
        4'b0000, 4'b0001, 4'b0010, 4'b0011, 4'b0100, 4'b0101, 4'b0110:
            PC_updated = valP;

        4'b0111:
            PC_updated = (cnd) ? valC : valP;

        4'b1000:
            PC_updated = valC;

        4'b1001:
            PC_updated = valM;

        4'b1010, 4'b1011:
            PC_updated = valP;
    endcase
end
```

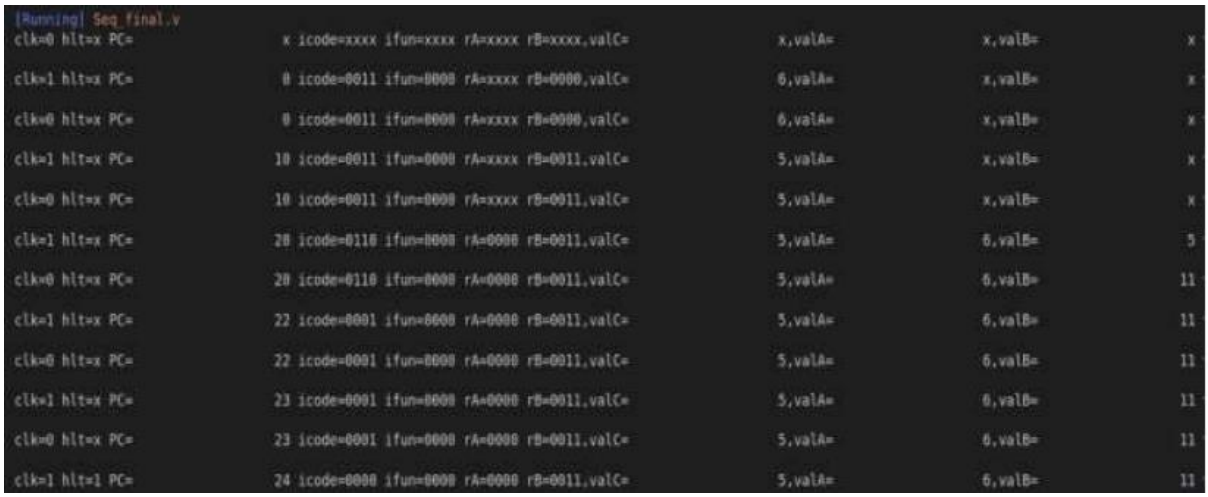
TESTING AND GTK WAVE OUTPUTS:

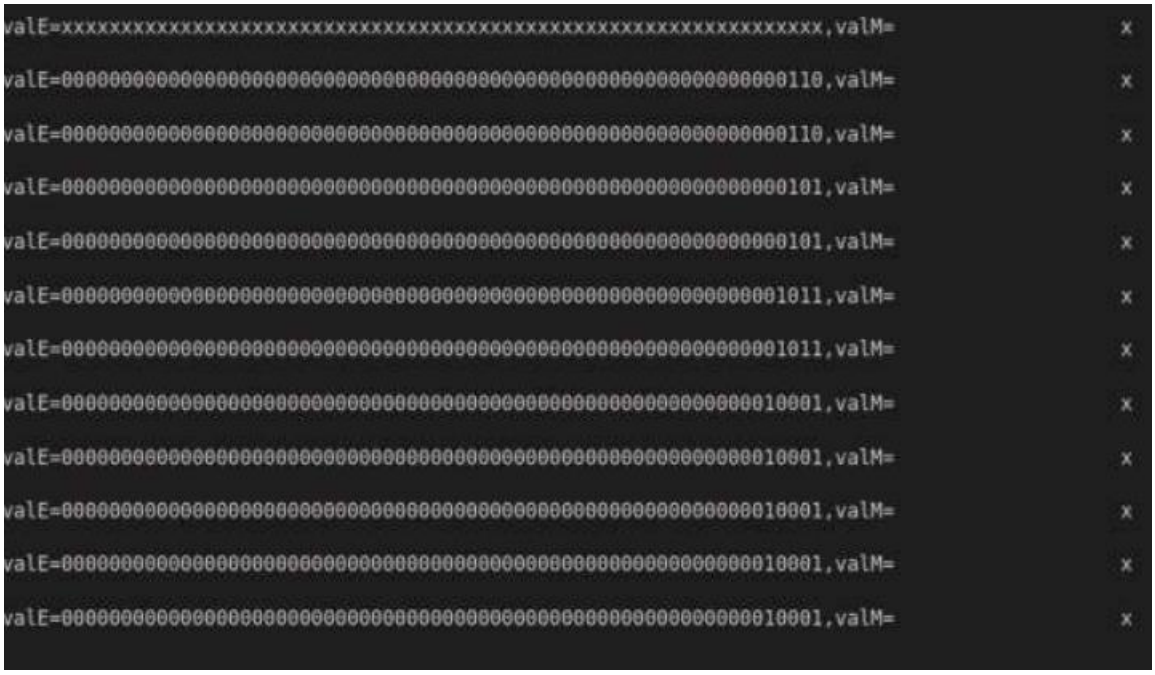
FOR THIS TEST CASE:

```
inst_mem[0] = 8'b00110000;  //icode:ifun
inst_mem[1] = 8'b11110000;  // rA:rB
inst_mem[2] = 8'b00000000;  //dest
inst_mem[3] = 8'b00000000;
inst_mem[4] = 8'b00000000;
inst_mem[5] = 8'b00000000;
inst_mem[6] = 8'b00000000;
inst_mem[7] = 8'b00000000;
inst_mem[8] = 8'b00000000;
inst_mem[9] = 8'b00000110;

inst_mem[10] = 8'b00110000;  //icode:ifun
inst_mem[11] = 8'b11110011;  // rA:rB
inst_mem[12] = 8'b00000000;  //dest
inst_mem[13] = 8'b00000000;
inst_mem[14] = 8'b00000000;
inst_mem[15] = 8'b00000000;
inst_mem[16] = 8'b00000000;
inst_mem[17] = 8'b00000000;
inst_mem[18] = 8'b00000000;
inst_mem[19] = 8'b00000101;

inst_mem[20] = 8'b01100000;  //opq:addq
inst_mem[21] = 8'b00010011;  // rcx:rdx
```

- Here the first line represent the icode and ifun.

- Second line represent the registers and remaining are the nop to fill all other bytes in that instruction.
- Similarly, does the same below.

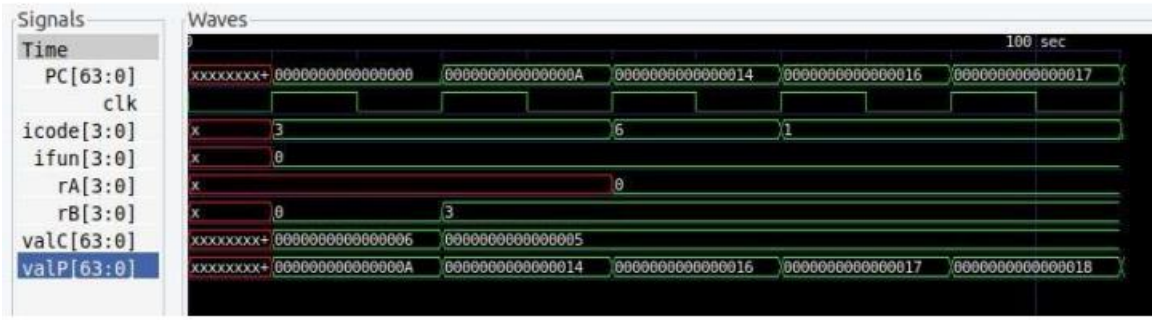




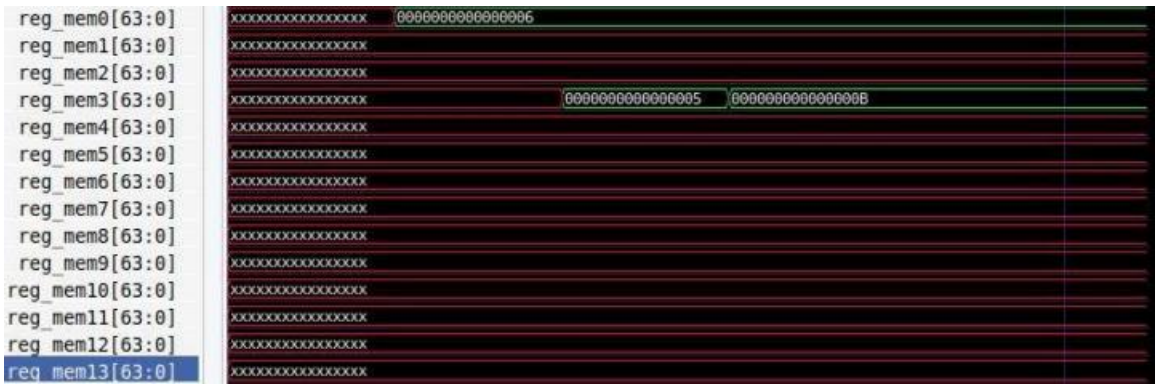
- The above pic contains clk, icode, ifun, rA, rB, valA and valB.
- As we have given value 6 to rA and 5 to rB.
- So, we see that valA has 6 and valB is 5.
- From here we say fetch and decode are working fine.
- Then we also see the ValB output is updated to 11 as we done the sum operation.
- So, execute is also working fine.

WAVEFORM:

FROM FETCH:



FROM DECODE:



- As we considered %rax and %rbx, we see only those are filled with values and all are in high impedance state.

FROM EXECUTE:

Thus, we were done with sequential part of y86-64 processor and we also verified.

5 - STAGED PIPELINE IMPLEMENTATION

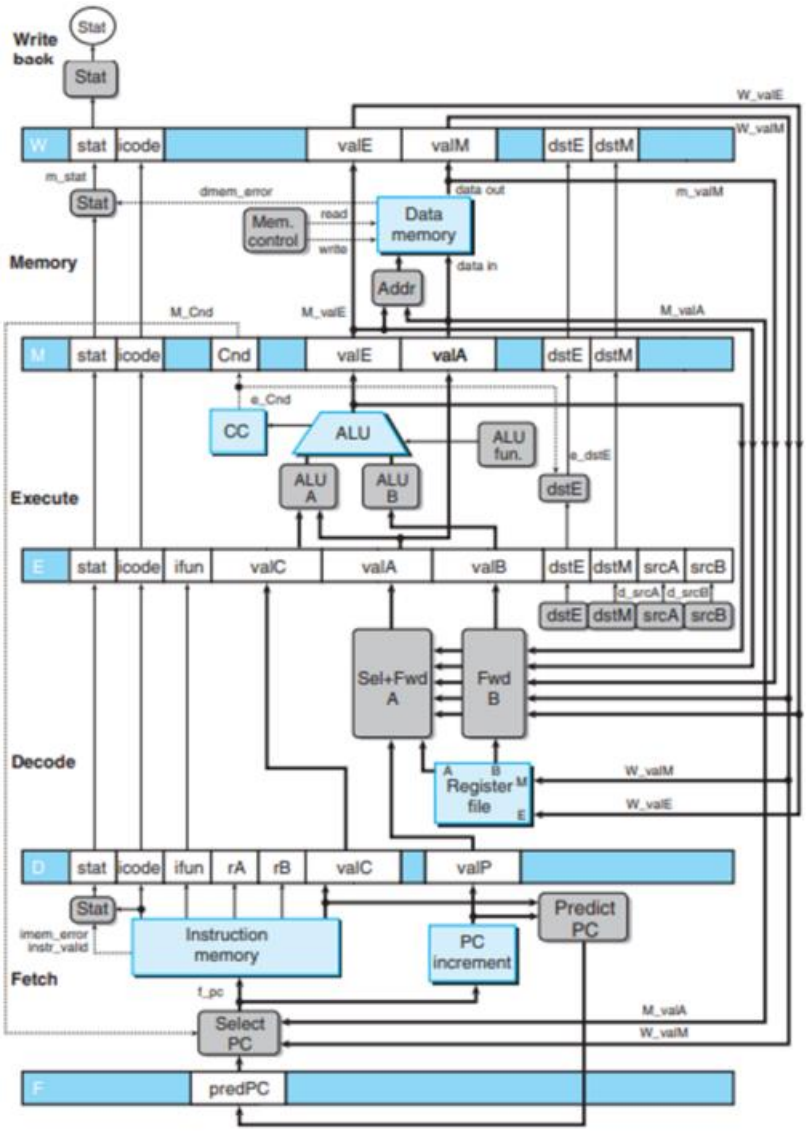
PIPELINING:

- In case of sequential, the next instruction starts processing after the first instruction is done.
- So, sequential will be needing many clock cycles and throughput (number of instructions per second) will be minimum.
- So, the main purpose of using pipelining is to reduce time delay and to increase the throughput
- Each instruction is broken down into smaller tasks, and each task is executed by a different stage of the pipeline.

- This allows multiple instructions to be in different stages of execution at the same time, leading to improved performance and throughput.
- One of the main advantages of pipeline implementation is increased speed. Because each instruction is broken down into smaller tasks, multiple instructions can be executing simultaneously, resulting in a higher rate of instruction throughput.
- This leads to faster execution of programs and increased overall performance. Another advantage is improved efficiency.
- Pipeline implementation allows for better resource utilization by minimizing idle time and reducing the amount of time that resources are unused.
- Additionally, pipeline implementation allows for more efficient use of resources such as memory and registers.

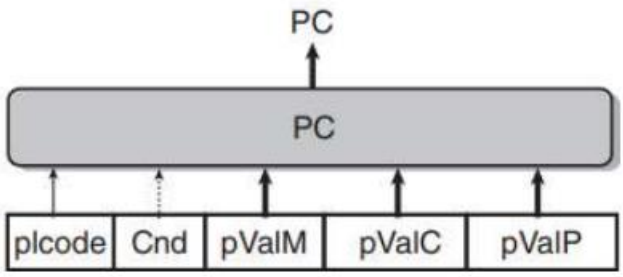
PIPELINED Y-86 IMPLEMENTATION:

BLOCK DIAGRAM:

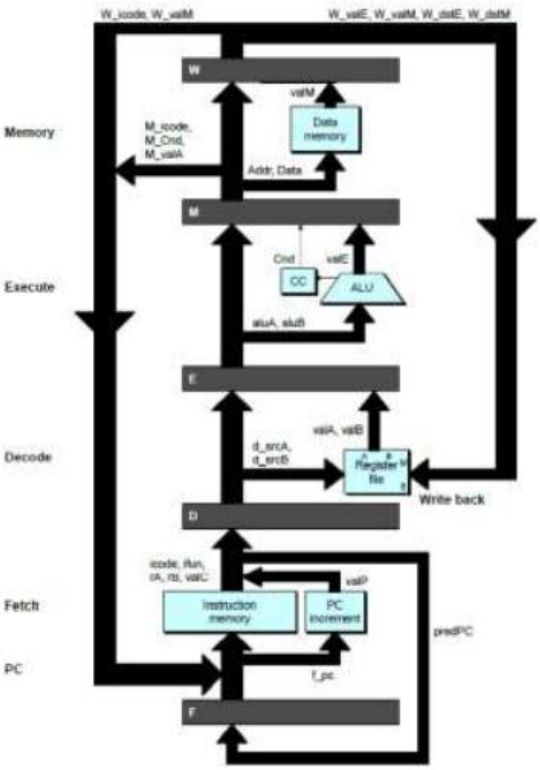


SHIFTING THE TIMING OF THE PC COMPUTATION:

- In sequential, we track the next instruction address at the end of all stages. But in case of pipelined architecture, we need to have the PC at the beginning of the cycle so that it can continuously fetch the instruction. This is known to be **circuit retiming.**
- We create state registers to hold the signals computed during an instruction. Then, as a new cycle begins, the values propagate through the exact same logic to compute the PC for the nowcurrent instruction. We label the registers “pIcode”, “pCnd” and so on, to indicate that on any given cycle, they hold the control signals generated during the previous cycle.



INSERTING THE PIPELINE REGISTERS:



- These pipeline registers are inserted such that no signal from on stage gets into other stage.
- The pipeline registers are labelled as follows:
  - **F** holds a predicted value of the program counter.
  - **D** sits between fetch and decode stages.
  - **E** sits between decode and execute stages.
  - **M** sits between execute and memory stages.
  - **W** sits between the memory and feedback paths.

PIPELINE STAGES:

FETCH	Select current PC, Read instruction, Compute increment PC
DECODE	Read program registers
EXECUTE	Operate ALU
MEMORY	Read or write memory

WRITEBACK	Update register file

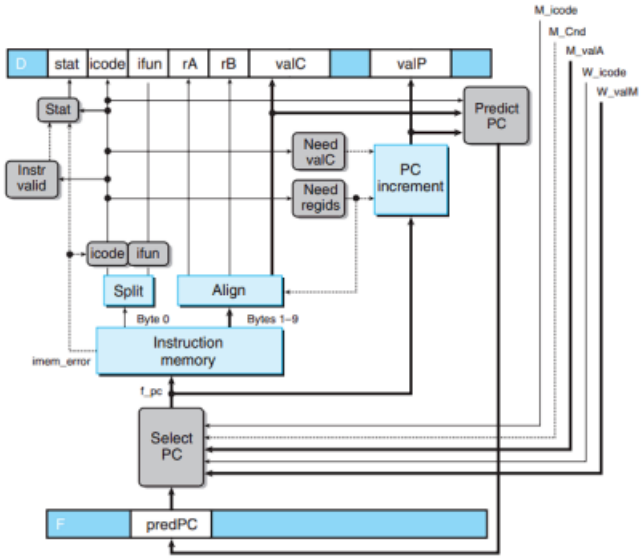
SIGNAL NAMING CONVENTIONS:

- S_Field – Value of field held in stage S pipeline register. (Here S and F both seem to be capital)
- s_Field – Value of field computed in stage S. (Here's is small).

PIPE STAGE IMPLEMENTATIONS:

1. PC SELECTION AND FETCH STAGE:

- The PC selection logic chooses between three program counter sources. As a mispredicted branch enters the memory stage, the value of valP for this instruction is read from pipeline register M (signal M_valA)
- The PC prediction logic chooses valC for the fetched instruction when it is either a call or a jump and valP otherwise.



- The fetch stage is the first stage in the pipeline.
- Its primary function is to fetch the instruction from memory and load it into an instruction register for the subsequent stages to execute.
- The process starts with incrementing the program counter (PC) to point to the next instruction in memory.
- Then, the instruction is fetched from memory using the address in the PC, and once the instruction has been fetched, it is loaded into an instruction register in the processor.
- This stage is critical to the pipeline implementation of the y86 processor, as it sets the stage for the subsequent stages in the pipeline.

- By fetching the instruction and loading it into the instruction register, the processor can begin decoding and executing the instruction in the subsequent stages, allowing for more efficient use of resources and improved performance.
- In the pipelined implementation we additionally have the predict_PC part where we predict the PC as follows:

CODE:

```
module fetch(F_predPC, M_icode, M_Cnd, M_valA, W_icode, W_valM, f_stat, f_icode, f_ifun, f_rA, f_rB, f_valC, f_valP, f_predPC);

    // inputs
    input M_Cnd;
    input [3:0] M_icode, W_icode;

    input [63:0] F_predPC,M_valA, W_valM;

    reg [7:0] inst_mem[0:1023];
    // outputs
    output reg [3:0] f_stat, f_icode, f_ifun, f_rA, f_rB;

    output reg signed [63:0] f_valC,f_valP, f_predPC;

    // Setting up instruction memory
    reg [63:0] PC;
    reg instr_valid;
    reg imem_error;

    initial begin
        // // addition works
    inst_mem[0] = 8'b00110000;  // irmovq
    inst_mem[1] = 8'b11110011;  // rA = 15, rB = 3
    inst_mem[2] = 8'b00000000;
    inst_mem[3] = 8'b00000000;
    inst_mem[4] = 8'b00000000;
    inst_mem[5] = 8'b00000000;
    inst_mem[6] = 8'b00000000;
    inst_mem[7] = 8'b00000000;
    inst_mem[8] = 8'b00000001;
```

```
    inst_mem[8] = 8'b00000001;
    inst_mem[9] = 8'b00000000;  // valC
    inst_mem[10] = 8'b00110000; // irmovq
    inst_mem[11] = 8'b11110010; // rA = 15, rB = 2
    inst_mem[12] = 8'b00000000;
    inst_mem[13] = 8'b00000000;
    inst_mem[14] = 8'b00000000;
    inst_mem[15] = 8'b00000000;
    inst_mem[16] = 8'b00000000;
    inst_mem[17] = 8'b00000000;
    inst_mem[18] = 8'b00000010;
    inst_mem[19] = 8'b00000000;  // valC
    inst_mem[20] = 8'b01100011;  // OPq add
    inst_mem[21] = 8'b00100011;  // rA = 2, rB = 3
    inst_mem[22] = 8'b00000000;  // halt


    end



    // Select PC


    always @(*)
    begin

        if (W_icode == 9)
```

```
            PC <= W_valM;
        else if (M_icode == 7 && M_Cnd == 0)
            PC <= M_valA;
        else
            PC <= F_predPC;
    end

    // Select f_icode, f_ifun, imem_error


    always @(*)
    begin
        if (PC >= 0 && PC < 4096) begin
            imem_error <= 0;
            f_ifun     <= inst_mem[PC][3:0];
            f_icode    <= inst_mem[PC][7:4];
        end
        else begin
            f_ifun     <= 0;
            imem_error <= 1;
            f_icode    <= 1;

        end
    end

    // instr_valid flag
```

```
module fetch(F_predPC, M_icode, M_Cnd, M_valA, W_icode, W_valM, f_stat, f_icode, f_ifun, f_rA, f_rB, f_valC, f_valP, f_predPC);


    always @(*)
    begin
        if (f_icode >= 0 && f_icode <= 11)
            instr_valid <= 1;
        else
            instr_valid <= 0;
    end

    // Setting rA, rB, valC, valP
    always @(*) begin
        if (f_icode == 4'h2) begin // cmov

            f_valC = 0;

            f_valP = PC + 2;

            f_rB   = inst_mem[PC + 1][3:0];

            f_rA   = inst_mem[PC + 1][7:4];

        end
        else if (f_icode == 4'h3) begin // irmovq



            f_valP = PC + 10;
```

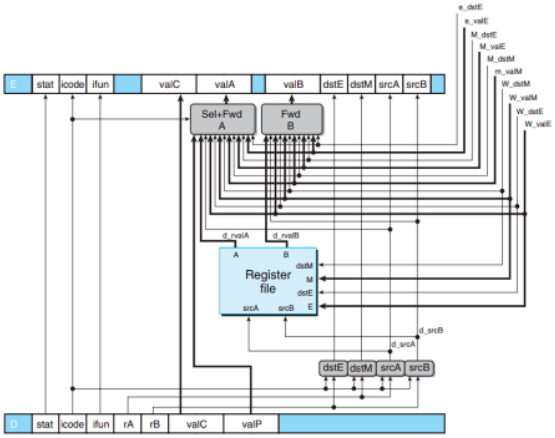
```
            f_valC = {inst_mem[PC+2],inst_mem[PC+3],inst_mem[PC+4],inst_mem[PC+5],inst_mem[PC+6],inst_mem[PC+7],inst_mem[PC+8],inst_mem[PC+

            f_rB   = inst_mem[PC + 1][3:0];

            f_rA   = inst_mem[PC + 1][7:4];

        end
        else if (f_icode == 4'h4) begin // rmmovq


            f_valP = PC + 10;

            f_valC = {inst_mem[PC+2],inst_mem[PC+3],inst_mem[PC+4],inst_mem[PC+5],inst_mem[PC+6],inst_mem[PC+7],inst_mem[PC+8],inst_mem[PC+

            f_rB   = inst_mem[PC + 1][3:0];

            f_rA   = inst_mem[PC + 1][7:4];

        end
        else if (f_icode == 4'h5) begin // mrmovq

            f_valC = {inst_mem[PC+2],inst_mem[PC+3],inst_mem[PC+4],inst_mem[PC+5],inst_mem[PC+6],inst_mem[PC+7],inst_mem[PC+8],inst_mem[PC+

            f_rB   = inst_mem[PC + 1][3:0];
```

```
        f_rA   = inst_mem[PC + 1][7:4];

        f_valP = PC + 10;
    end
    else if (f_icode == 4'h6) begin // OPq
        f_valC = 0;
        f_valP = PC + 2;
        f_rB   = inst_mem[PC + 1][3:0];

        f_rA   = inst_mem[PC + 1][7:4];

    end
    else if (f_icode == 4'h7) begin // jxx
        f_valP = PC + 9;
        f_valC = {inst_mem[PC+1],inst_mem[PC+2],inst_mem[PC+3],inst_mem[PC+4],inst_mem[PC+5],inst_mem[PC+6],inst_mem[PC+7],inst_mem[PC+

    end
    else if (f_icode == 4'h8) begin // call
        f_valP = PC + 9;
        f_valC = {inst_mem[PC+1],inst_mem[PC+2],inst_mem[PC+3],inst_mem[PC+4],inst_mem[PC+5],inst_mem[PC+6],inst_mem[PC+7],inst_mem[PC+

    end
    else if (f_icode == 4'h9) begin // ret
        f_valP = PC + 1;
    end
```

```
always @(*) begin
    else if (f_icode == 4'hA) begin // pushq
        f_valC = 0;
        f_rA   = inst_mem[PC + 1][7:4];

        f_valP = PC + 2;

        f_rB   = inst_mem[PC + 1][3:0];


    end
    else if (f_icode == 4'hB) begin // popq
        f_valP = PC + 2;
        f_rB   = inst_mem[PC + 1][3:0];
        f_valC = 0;
        f_rA   = inst_mem[PC + 1][7:4];

    end
    else begin
        f_rA   = 15;
        f_rB   = 15;
        f_valP = PC + 1;
    end
end

// Predicting the PC
always @(*) begin
    if (f_icode == 4'h7)
```

```
    // Predicting the PC
    always @(*) begin
        if (f_icode == 4'h7)
            f_predPC = f_valC;
        else if (f_icode == 4'h8)
            f_predPC = f_valC;
        else
            f_predPC = f_valP;
    end

    // Generating Stat Codes
    always @(*) begin
        if (imem_error)
            f_stat = 4'b0010;
        else if (!instr_valid)
            f_stat = 4'b0001;
        else if (f_icode == 4'h0)
            f_stat = 4'b0100;
        else
            f_stat = 4'b1000;
    end

endmodule
```

DECODE AND WRITEBACK STAGE:



FLOW OF THE CODE:

- Firstly, we need to create a f_reg to store all the values which is input to the fetch stage. So, we tried implementing it and did it.
- By using that we can do data forwarding.
- The most important part, we focus here is on the pc update.
- As we use select PC in y86-64 pipelined process, we wrote modules for select PC, predict pc and incrementing PC.
- Based on the value of icode we write the value of PC.

CODE:

DECODE AND WRITEBACK STAGES:

- This stage is associated with the forwarding logic. The forwarding logic choose the one in the execute stage, since it represents the most recently generated value for this register.
- The block labelled “Sel+Fwd” serves two roles. It merges valP signal into the valA signal for later stages in order to reduce the amount of state in the pipeline register.
- It also implements the forwarding logic for source operand valA.
- When valP is needed and is in still before the memory stage then the selection is controlled by the icode signal for this stage.
- When signal D_icode matches the instruction code for wither call or jxx this block should select D_valP as its output.

Based as shown below, the decode stage picks the valuer from the forwarding source:

DATA WORD	REGISTER ID
e_valE	E_dstE
m_valM	M_dstM
M_valE	M_dstE
W_valM	W_dstM

W_valE	W_dstE

FLOW OF THE CODE:

- In the decode stage of pipelining we have to do many works which we have implemented using the Verilog modules.
- If the functions are cmovxx, rmmovq, opq and push then we will have the new register position RA = rA, and now if the function is return or pop the value of rA turns out to be the stack pointer or the %rsp which is given by the register index 4.
- Similarly in the case of the other register RB if we have the functions mrmovq,rmmovq,opq then we have the RB = rB, else if we have the functions call, return, push or pop then we will have to update the value of the RB to the value of the stack pointer which is given by %rsp and has the index of 4 else the value of the RB is taken to be the dummy register of index 15 which is not used in y86-64 processors.
- Now after finding we values of the index of the registers; we must be able to retrieve the values stored in the registers.
- These are the values valA and valB, which are required for the subsequent stages of the implementation.
- This above explained point is shown in the select valA, valB module where in the outputs are valA and valB depending on the values of icode.
- Also, in this stage we will be in a position to set the destinations of the register depending on the value of the icode only.
- This is implemented in a module named DestSET_E_M.
- This is used in the writeback stage wherein we need the value of the destination of where we will be writing the register.
- For add, cmovxx, irmovq we set the destination register to be same as that of the rB. This can potentially kill the data present in the same register which we get in the decode stage.

CODE:

```
module decode (
    input clk,

    // Inputs from D register
    input [3:0] D_stat,D_icode, D_ifun,D_rA, D_rB,
    input signed [63:0] D_valC,
    input [63:0] D_valP,

    // Inputs forwarded from execute stage
    input [3:0] e_dstE,
    input signed [63:0] e_valE,

    // Inputs forwarded from M register and memory stage
    input [3:0] M_dstE, M_dstM,
    input signed [63:0] M_valE, m_valM,

    // Inputs forwarded from W register
    input [3:0] W_dstM, W_dstE,
    input signed [63:0] W_valM, W_valE,

    // Outputs
    output reg[3:0] d_stat,d_icode, d_ifun,
    output reg signed[63:0] d_valA, d_valB, d_valC,
    output reg[3:0] d_dstE, d_dstM,d_srcA, d_srcB,

    output reg signed [63:0] reg_file0,reg_file1,reg_file2,reg_file3,reg_file4,reg_file5,reg_file6,reg_file7,
    reg_file8,reg_file9,reg_file10,reg_file11,reg_file12,reg_file13,reg_file14
);

   reg [63:0] reg_array[0:15];
```

```
reg [63:0] reg_array[0:15];

 always @(*)
 begin
     reg_file0 = reg_array[0];
     reg_file1 = reg_array[1];
     reg_file2 = reg_array[2];
     reg_file3 = reg_array[3];
     reg_file4 = reg_array[4];
     reg_file5 = reg_array[5];
     reg_file6 = reg_array[6];
     reg_file7 = reg_array[7];
     reg_file8 = reg_array[8];
     reg_file9 = reg_array[9];
     reg_file10 = reg_array[10];
     reg_file11 = reg_array[11];
     reg_file12 = reg_array[12];
     reg_file13 = reg_array[13];
     reg_file14 = reg_array[14];
 end

 initial
 begin
     reg_array[0] = 64'd0;
     reg_array[1] = 64'd0;
     reg_array[2] = 64'd0;
     reg_array[3] = 64'd0;
     reg_array[4] = 64'd0;
```

```
     reg_array[5] = 64'd0;
     reg_array[6] = 64'd0;
     reg_array[7] = 64'd0;
     reg_array[8] = 64'd0;
     reg_array[9] = 64'd0;
     reg_array[10] = 64'd0;
     reg_array[11] = 64'd0;
     reg_array[12] = 64'd0;
     reg_array[13] = 64'd0;
     reg_array[14] = 64'd0;
     reg_array[15] = 64'd0;

 end
     always @(posedge clk)
 begin
     reg_array[W_dstM] <= W_valM;
     reg_array[W_dstE] <= W_valE;
 end

 // No change wires
 always @(*)
 begin
     d_valC  <= D_valC;
     d_stat  <= D_stat;
     d_icode <= D_icode;
     d_ifun  <= D_ifun;
```

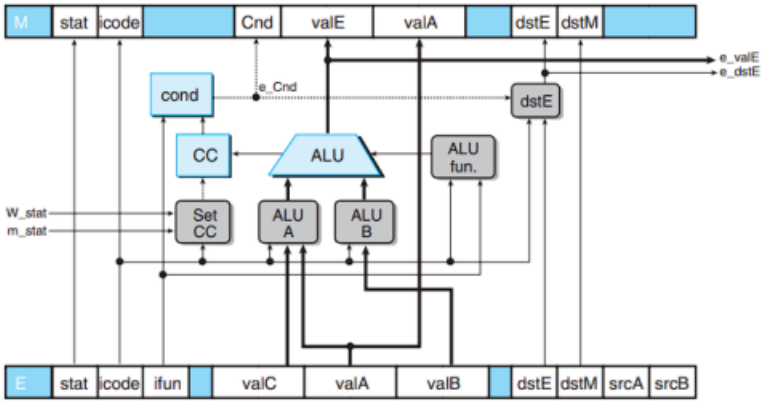
```
reg [63:0] reg_array[0:15];

  // d_dstE
  always @(*) begin
  if (d_icode == 4'h2 || d_icode == 4'h3 || d_icode == 4'h6)
      d_dstE <= D_rB;
  else if (d_icode == 4'h8 || d_icode == 4'h9 || d_icode == 4'hA || d_icode == 4'hB)
      d_dstE <= 4;
  else
      d_dstE <= 15;
end


  // d_dstM
  // d_dstM
always @(*) begin
  if (d_icode == 4'h5 || d_icode == 4'hB)
      d_dstM = D_rA;
  else
      d_dstM = 15;
end

// d_srcA
always @(*) begin
  if (d_icode == 4'h2 || d_icode == 4'h4 || d_icode == 4'h6)
      d_srcA = D_rA;
  else if (d_icode == 4'h9 || d_icode == 4'hA || d_icode == 4'hB)
      d_srcA = 4;
```

```
    else
        d_srcA = 15;
end

// d_srcB
always @(*) begin
    if (d_icode == 4'h4 || d_icode == 4'h5 || d_icode == 4'h6)
        d_srcB = D_rB;
    else if (d_icode == 4'h8 || d_icode == 4'h9 || d_icode == 4'hA || d_icode == 4'hB)
        d_srcB = 4;
    else
        d_srcB = 15;
end


    // Data forwarding of A
    always @(*) begin
    case (1)
        d_icode == 7 || d_icode == 8:
            d_valA <= D_valP;
        d_srcA == e_dstE:
            d_valA <= e_valE;
        d_srcA == M_dstM:
            d_valA <= m_valM;
        d_srcA == M_dstE:
            d_valA <= M_valE;
        d_srcA == W_dstM:
            d_valA <= W_valM;
```

```
            d_valA <= W_valM;
        d_srcA == W_dstE:
            d_valA <= W_valE;
        default:
            d_valA <= reg_array[d_srcA];
    endcase
end

    // Data forwarding of B
always @(*) begin
    case (d_srcB)
        e_dstE: d_valB <= e_valE;
        M_dstM: d_valB <= m_valM;
        M_dstE: d_valB <= M_valE;
        W_dstM: d_valB <= W_valM;
        W_dstE: d_valB <= W_valE;
        default: d_valB <= reg_array[d_srcB];
    endcase
end


endmodule
```

EXECUTE STAGE:

- The hardware unit are identical in place lo SEQ and pipelined version. The only difference is the logic labelled Set CC which determines whether or not update the condition codes.
- The signals e_valE and e_dstE directed towards the decode stage as one of the forwarding sources.



FLOW OF THE CODE:

- Here, we firstly focus on ALU. We write the codes for all operations needed in alu i.e., Add, Sub, Xor, And.
- We done these operations without using the operators. •
- We also wrote a module for modifying valA and valB. If the instruction is cmovxx and OPq then the valA is same.
- But if instruction is irmovq, mrmovq and rmvovq then it takes valC for the value of valA.
- We also wrote a block for setting the e_dstE in execute block.
- So, if E_icode is cmovxx and e_cnd is 0 then e_dstE should be the unused last register i.e., %r15.
- We also focus on setting conditional codes.

CODE:

```
module execute (
    input clk,

    // Inputs
    input [3:0] E_stat,E_icode, E_ifun,
    input signed [63:0] E_valC, E_valA, E_valB,
    input [3:0] E_dstE, E_dstM,W_stat, m_stat,

    // Outputs
    output reg[3:0] e_stat,e_icode,
    output reg signed [63:0] e_valE,e_valA,
    output reg[3:0] e_dstE,e_dstM,
    output reg e_Cnd,ZF, SF, OF
);
    wire signed [63:0] ans;
    reg signed [63:0] aluA, aluB;
    wire overflow, set_cc;
    reg [1:0] control;

    // No change wires
    always @(*)
    begin
        e_stat  <= E_stat;
        e_icode <= E_icode;
        e_valA  <= E_valA;
        e_dstM  <= E_dstM;
    end
```

```
// ask
assign set_cc = ((e_icode == 6) && (m_stat == 4'b1000) && (W_stat == 4'b1000)) ? 1 : 0;


// ask
alu alu1(.control(control), .a(aluA), .b(aluB), .overflow(overflow), .out(ans));

always @(*)
begin
    if (e_icode == 4'b0010)  // cmovXX rA, rB
    begin
        e_Cnd = 0;
        if (E_ifun == 4'b0000)  // cmove(unconditional)
            e_Cnd = 1;
        else if (E_ifun == 4'b0001 && ((SF^OF)|ZF))  // cmovle
            e_Cnd = 1;
        else if (E_ifun == 4'b0010 && (SF^OF))  // cmovl
            e_Cnd = 1;
        else if (E_ifun == 4'b0011 && ZF)  // cmove
            e_Cnd = 1;
        else if (E_ifun == 4'b0100 && !ZF)  // cmovne
            e_Cnd = 1;
        else if (E_ifun == 4'b0101 && !(SF^OF))  // cmovge
            e_Cnd = 1;
        else if (E_ifun == 4'b0110 && !(SF^OF) && !ZF)  // cmovg
            e_Cnd = 1;

        aluA = E_valA;
```

```
        aluA = E_valA;
        aluB = 64'd0;
        control = 2'b00;  // valE = valA + 0
    end
    else if (e_icode == 4'b0011)  // irmovq V, rB
    begin
        aluA = E_valC;
        aluB = 64'd0;
        control = 2'b00;  // valE = 0 + valC
    end
    else if (e_icode == 4'b0100)  // rmmovq rA, D(rB)
    begin
        aluA = E_valC;
        aluB = E_valB;
        control = 2'b00;  // valE = valB + valC
    end
    else if (e_icode == 4'b0101)  // mrmovq D(rB), rA
    begin
        aluA = E_valC;
        aluB = E_valB;
        control = 2'b00;  // valE = valB + valC
    end
    else if (e_icode == 4'b0110)  // Opq rA, rB
    begin
        aluA = E_valB;
        aluB = E_valA;
```

```
        control = E_ifun[1:0];
    end
    else if (e_icode == 4'b0111)  // jXX Dest
    begin
        e_Cnd = 0;
        if (E_ifun == 4'b0000)  // jmp
            e_Cnd = 1;  // unconditional jump
        else if (E_ifun == 4'b0001 && ((SF^OF)|ZF))  // jle
            e_Cnd = 1;
        else if (E_ifun == 4'b0010 && (SF^OF))  // jl
            e_Cnd = 1;
        else if (E_ifun == 4'b0011)  // je
            e_Cnd = ZF ? 1 : 0;
        else if (E_ifun == 4'b0100 && !ZF)  // jne
            e_Cnd = 1;
        else if (E_ifun == 4'b0101 && !(SF^OF))  // jge
            e_Cnd = 1;
        else if (E_ifun == 4'b0110 && !(SF^OF) && !ZF)  // jg
            e_Cnd = 1;
    end
    else if (e_icode == 4'b1000)  // call Dest
    begin
        aluA = -64'd8;
        aluB = E_valB;
        control = 2'b00;  // valE = valB - 8
    end
```

```
    else if (e_icode == 4'b1001)  // ret
    begin
        aluA = 64'd8;
        aluB = E_valB;
        control = 2'b00;  // valE = valB + 8
    end
    else if (e_icode == 4'b1010)  // pushq rA
    begin
        aluA = -64'd8;
        aluB = E_valB;
        control = 2'b00;  // valE = valB - 8
    end
    else if (e_icode == 4'b1011)  // popq rA
    begin
        aluA = 64'd8;
        aluB = E_valB;
        control = 2'b00;  // valE = valB + 8
    end
    else
        control = 2'b00;  // default control value

    e_valE = ans;
end

always @(posedge clk )
begin
    if (set_cc == 1)
```

```
    always @(posedge clk )
    begin
        if (set_cc == 1)
        begin
            ZF = (e_valE == 64'b0); // zero flag
            SF = (e_valE[63] == 1); // signed flag
            OF = (aluA[63] == aluB[63]) && (e_valE[63] != aluA[63]); // overflow flag
        end
    end

    // e_dstE
    always @(*)
    begin
        if (e_icode == 2 && e_Cnd == 0)
        begin
            e_dstE <= 15;
        end
        else
        begin
            e_dstE <= E_dstE;
        end
    end
endmodule
```

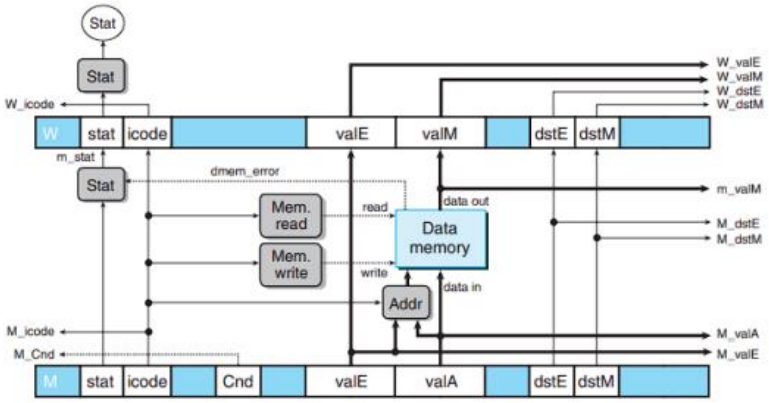
- The execute stage performs the actual operation specified by the instruction, such as arithmetic or logical operations, data movement, or branching.
- The operands for the operation are typically fetched from the register file or memory in the previous stage, and are available for use by the execute stage.
- During the execute stage, the ALU (Arithmetic Logic Unit) performs the specified operation on the operands.
- The result of the operation may be written to a temporary register, which is used to store intermediate results during the execution of multi-stage instructions.
- The result may also be used to update the condition codes, which are used to determine whether subsequent instructions should be executed.
- Once the operation is complete, the result is passed on to the next stage in the pipeline.

- The outputs obtained from this block include M_stat, M_icode, Cnd, M_valE, M_valA, M_dstE, M_dstM, e_valE and e_dstE. Here E_stat, E_ifun, E_icode, E_valA, E_valB, E_valC, E_dstE and E_dstM which are the inputs when passed through this block compute M_stat, M_icode, Cnd, M_valE, M_valA, M_dstE, M_dstM and e_valE as outputs in the similar way to that of sequential.
- The value e_dstE is computed based on e_cnd which will make it either E_dstE or an empty register.

MEMORY STAGE:

• If we compare this memory stage in pipeline with the sequential stage then, Data block is replaced with Sel+Fwd A.

• Many of the signals from pipeline registers M and W are passed down to earlier stages to provide write-back results, instruction addresses, and forwarded results.



FLOW OF THE CODE:

- Firstly, in the memory block we need to find the location to which we are pointing to and fetch the data or write the data into the memory.
- If the function is rmmovq, mrmovq, pushq, call then the positing of writing is into valE.
- If we compare this memory stage in pipeline with the sequential stage then, Data block is replaced with Sel+Fwd A.
- Many of the signals from pipeline registers M and W are passed down to earlier stages to provide write-back results, instruction addresses, and forwarded results.
- If the instruction tells us that the function is return or pop, then we have the location in the memory to be valA.
- After finding the location in the memory we need to set the read and write enables for better assess of the memory without destroying the computer.
- If the function is rmmovq, call and push then we only need to read the memory and therefore it is sufficient.
- If the function is mrmovq, return or pop then we need to write into the memory and therefore need to make the write enable as 1 and then get permission to write into the memory.
- After setting the enable pins we need to check for the data memory error and therefore setting or updating the status to the new_status.

- This must happen only if we use the data_memory and thus we have this condition only for the functions rmmovq, mrmovq, call, ret, push and pop function.

CODE:

```
module memory(
    input clk,
    input [3:0] M_stat,M_icode,
    input signed [63:0] M_valE, M_valA,
    input [3:0] M_dstE, M_dstM,
    output reg[3:0] m_stat,m_icode,
    output reg signed[63:0] m_valE, m_valM,
    output reg[3:0] m_dstE, m_dstM
);

   reg [63:0] Data_Mem[0:4095];
    integer i;

    initial begin
        for (i = 0; i < 4096; i = i+1)
        begin
            Data_Mem[i] <= 0;
        end

        // $dumpvars(0, Data_Mem[0], Data_Mem[1], Data_Mem[2], Data_Mem[3], Data_Mem[4], Data_Mem[5], Data_Mem[6], Data_Mem[7], Data_Mem[8]
    end

    // No change wires
    always @(*)
    begin
        m_icode <= M_icode;
        m_valE  <= M_valE;
        m_dstE  <= M_dstE;
        m_dstM  <= M_dstM;
```

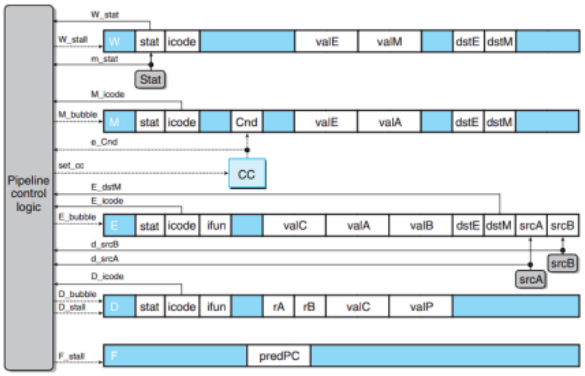
```
reg [63:0] Data_Mem[0:4095];
 end

 // Mem_write block
 reg Mem_write;

 always @(*)
 begin
     if (m_icode == 4'h4 || m_icode == 4'h8 || m_icode == 4'hA)
         Mem_write <= 1;
     else
         Mem_write <= 0;
 end

 // Mem_read block
 reg Mem_read;

 always @(*)
 begin
     if (m_icode == 4'h5 || m_icode == 4'h9 || m_icode == 4'hB)
         Mem_read <= 1;
     else
         Mem_read <= 0;
 end

 // Selecting Address
 reg [63:0] m_addr;
```

```
always @(*)
begin
    if (m_icode == 4'h4 || m_icode == 4'h5 || m_icode == 4'h8 || m_icode == 4'hA)
        m_addr <= m_valE;
    else if (m_icode == 4'h9 || m_icode == 4'hB)
        m_addr <= M_valA;
    else
        m_addr <= 4095;
end

// Checking memory error
reg dmem_error;     // Memory error flag

always @(*)
begin
    if (m_addr < 4096 && m_addr >= 0)
        dmem_error <= 0;
    else
        dmem_error <= 1;
end

wire [63:0] m_data_in;

assign m_data_in = M_valA;

// Writing back to memory
always @(posedge clk)
```

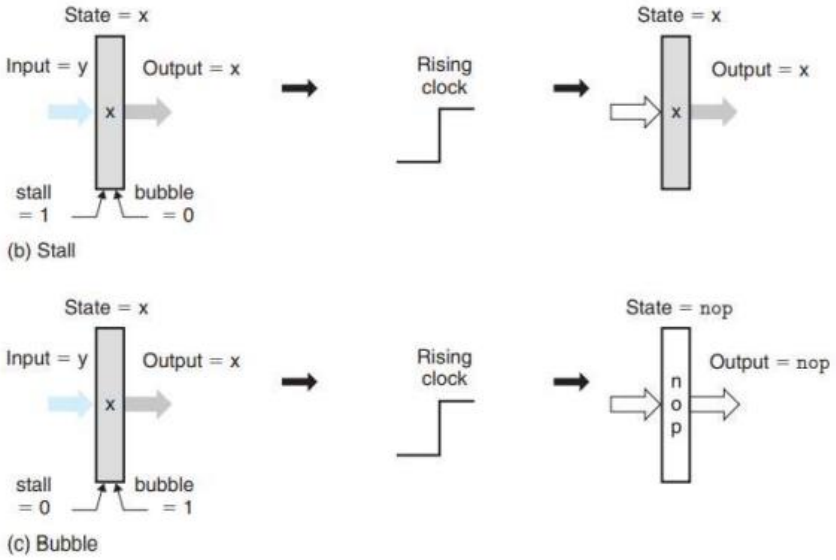
```
    always @(posedge clk)
    begin
        if (dmem_error == 0 && Mem_write == 1)
            Data_Mem[m_addr] <= m_data_in;
    end

    // Reading from memory
    always @(*)
    begin
        if (dmem_error == 0 && Mem_read == 1)
            m_valM <= Data_Mem[m_addr];
        else
            m_valM <= 0;
    end

    // m_stat
    always @(*)
    begin
        if (dmem_error == 1)
            m_stat <= 4'b0010;
        else
            m_stat <= M_stat;
    end
endmodule
```

- During the memory stage, the processor may access the data cache or main memory to read or write data.
- For memory read operations, the processor fetches data from the memory location specified by the instruction and stores it in a temporary register.
- For memory write operations, the processor stores the data from a register into the specified memory location.
- If the instruction being executed does not involve any memory operations, the memory stage is simply used to pass the result of the execute stage to the next stage of the pipeline.
- Memory stage allows the processor to access memory simultaneously with other pipeline stages.
- By overlapping the memory access with other stages, the processor can minimize the number of cycles needed to complete an instruction, improving overall performance
- This takes the inputs as the outputs from the memory pipelined register which include M_stat, M_icode, M_Cnd, M_valE, M_valA, M_dstE and M_dstM.
- The outputs obtained from this block include W_stat, W_icode, W_valE, W_valM, W_dstE, W_dstM and m_valM.

PIPELINING USING THE ABOVE MODULES:



- We separately created a module where we can do the pipelining based on the condition and by using the above modules.
- We created these modules based on the idea of stalling and bubbling.



- In case of stalling, the operation will give the same previously stored value until the value reaches the write back stage.
- In case of bubble, the nop operation will take place.
- So, we wrote if reset is 1 then bubble has to happen and it has to come back to the reset value and no operation is done.
- And if it is 0 then the operation becomes stall and it takes the input value as output value

CODE:

```
module PIPE_con(
    input [3:0] D_icode, E_icode, M_icode,
    input [3:0] d_srcA, d_srcB,
    input [3:0] E_dstM,
    input e_Cnd,
    input [3:0] m_stat, W_stat,

    output reg W_stall, D_stall, F_stall,
    output reg M_bubble, E_bubble, D_bubble
);

    wire Ret = (D_icode == 9 || E_icode == 9 || M_icode == 9);
    wire LU_Haz = ((E_icode == 5 || E_icode == 11) && (E_dstM == d_srcA || E_dstM == d_srcB));
    wire Miss_Pred = (E_icode == 7 && e_Cnd == 0);

    // F_stall
    always @(*)
        F_stall = (Ret || LU_Haz || (Ret && Miss_Pred)) ? 1 : 0;

    // D_stall
    always @(*)
        D_stall = (LU_Haz || (Ret && LU_Haz)) ? 1 : 0;

    // D_bubble
    always @(*)
        D_bubble = (D_stall == 0) ? (Ret || Miss_Pred) : 0;

    // E_bubble
    always @(*)
```

```
    // E_bubble
    always @(*)
        E_bubble = (LU_Haz && Ret) || (Ret && Miss_Pred) || LU_Haz || Miss_Pred;

    // M_bubble
    always @(*)
        M_bubble = (m_stat != 4'b1000) || (W_stat != 4'b1000);

    // W_stall
    always @(*)
        W_stall = (W_stat != 4'b1000);

endmodule
```

- The pipeline control logic is responsible for managing the pipeline stages and ensuring that instructions are executed in the correct order.
- The pipeline control logic coordinates the flow of instructions through the pipeline, ensuring that each stage completes its operation before passing the instruction on to the next stage.
- The pipeline control logic solves the following issues:
- Load/use hazards: The pipeline must stall for one cycle between an instruction that reads a value from memory and an instruction that uses this value.
- Processing return: The pipeline must stall until the ret instruction reaches the write-back stage.
- Mis-predicted branches: By the time the branch logic detects that a jump should not have been taken, several instructions at the branch target will have started down the pipeline.
- These instructions must be cancelled, and fetching should begin at the instruction following the jump instruction
- Exceptions: When an instruction causes an exception, we want to disable the updating of the programmer-visible state by later instructions and halt execution once the excepting instruction reaches the write-back stage.

FINAL IMPLEMENTATION OF THE 5 STAGED PIPELINE:

```
module PIPE();

    reg clk;

    // F pipeline register
    reg [63:0] F_predPC = 0;

    // D pipeline register
    reg [3:0] D_stat = 4'b1000;
    reg [3:0] D_icode = 1;
    reg [3:0] D_ifun ,D_rA ,D_rB = 0;
    reg signed [63:0] D_valC = 0;
    reg [63:0] D_valP = 0;


    // M pipeline register
    reg [3:0] M_stat = 4'b1000;
    reg [3:0] M_icode = 1;
    reg M_Cnd = 0;
    reg signed [63:0] M_valE = 0;
    reg signed [63:0] M_valA = 0;
    reg [3:0] M_dstE = 0;
    reg [3:0] M_dstM = 0;

    // E pipeline register
    reg [3:0] E_stat = 4'b1000;
    reg [3:0] E_icode = 1;
    reg [3:0] E_ifun = 0;

    reg signed [63:0] E_valA, E_valB ,E_valC = 0;
    reg [3:0] E_dstE ,E_dstM,E_srcA ,E_srcB = 0;
    wire ZF,OF,SF;

        // Registers
    wire signed [63:0] reg_file0,reg_file1,reg_file2,reg_file3,reg_file4,reg_file5,reg_file6,reg_file7,
    reg_file8,reg_file9,reg_file10,reg_file11,reg_file12,reg_file13,reg_file14;

        // Memory
    wire [3:0] m_stat,m_icode;
    wire signed [63:0] m_valE, m_valM;
    wire [3:0] m_dstE, m_dstM;

    wire [3:0] Stat;

    // Execute
    wire [3:0] e_stat,e_icode;
    wire e_Cnd;
    wire signed [63:0] e_valE, e_valA;
    wire [3:0] e_dstE, e_dstM;

    // W pipeline register
    reg [3:0] W_stat = 4'b1000;
    reg [3:0] W_icode = 1;
    reg signed [63:0] W_valE,W_valM = 0;
    reg [3:0] W_dstE ,W_dstM = 0;

    // Fetch
    wire [3:0] f_stat,f_icode, f_ifun,f_rA, f_rB;
    wire signed [63:0] f_valC;
    wire [63:0] f_valP, f_predPC;

    // Decode
    wire [3:0] d_stat,d_icode, d_ifun;
    wire signed [63:0] d_valC, d_valA, d_valB;
    wire [3:0] d_dstE, d_dstM,d_srcA, d_srcB;
```

```
// Passing inputs to FETCH stage
fetch f(
    // Inputs from F register
    .F_predPC(F_predPC),

    // Inputs forwarded from M register
    .M_icode(M_icode),.M_Cnd(M_Cnd),.M_valA(M_valA),

    // Inputs forwarded from W register
    .W_icode(W_icode),.W_valM(W_valM),

    // Outputs
    .f_stat(f_stat),.f_icode(f_icode),.f_ifun(f_ifun),.f_rA(f_rA),.f_rB(f_rB),.f_valC(f_valC),.f_valP(f_val
    .f_predPC(f_predPC)
        );


execute e(
    .clk(clk),

    // Inputs
    .E_stat(E_stat),.E_icode(E_icode),.E_ifun(E_ifun),.E_valC(E_valC),.E_valA(E_valA),
    .E_valB(E_valB),.E_dstE(E_dstE),.E_dstM(E_dstM),.W_stat(W_stat),.m_stat(m_stat),

    // Outputs
    .e_stat(e_stat),.e_icode(e_icode),.e_Cnd(e_Cnd),.e_valE(e_valE),.e_valA(e_valA),
    .e_dstE(e_dstE),.e_dstM(e_dstM),
    .ZF(ZF),
    .OF(OF),
    .SF(SF)
);


decode d(
    .clk(clk),

    // Inputs from D register
    .D_stat(D_stat),.D_icode(D_icode),.D_ifun(D_ifun),.D_rA(D_rA),.D_rB(D_rB),.D_valC(D_valC),.D_valP(D_val

    // Inputs forwarded from execute stage
    .e_dstE(e_dstE),.e_valE(e_valE),

    // inputs forwarded from M register and memory stage
    .M_dstE(M_dstE),.M_valE(M_valE),.M_dstM(M_dstM),.m_valM(m_valM),

    // Inputs forwarded from W register
    .W_dstM(W_dstM),.W_valM(W_valM),.W_dstE(W_dstE),.W_valE(W_valE),

    // Outputs
    .d_stat(d_stat),.d_icode(d_icode),.d_ifun(d_ifun),.d_valC(d_valC),.d_valA(d_valA),.d_valB(d_valB),
    .d_dstE(d_dstE),.d_dstM(d_dstM),.d_srcA(d_srcA),.d_srcB(d_srcB),


    .reg_file0(reg_file0),.reg_file1(reg_file1),.reg_file2(reg_file2),.reg_file3(reg_file3),
    .reg_file4(reg_file4),.reg_file5(reg_file5),.reg_file6(reg_file6),.reg_file7(reg_file7),
    .reg_file8(reg_file8),.reg_file9(reg_file9),.reg_file10(reg_file10),.reg_file11(reg_file11),
    .reg_file12(reg_file12),.reg_file13(reg_file13),.reg_file14(reg_file14)
);

memory m(
    .clk(clk),


    // Outputs
```

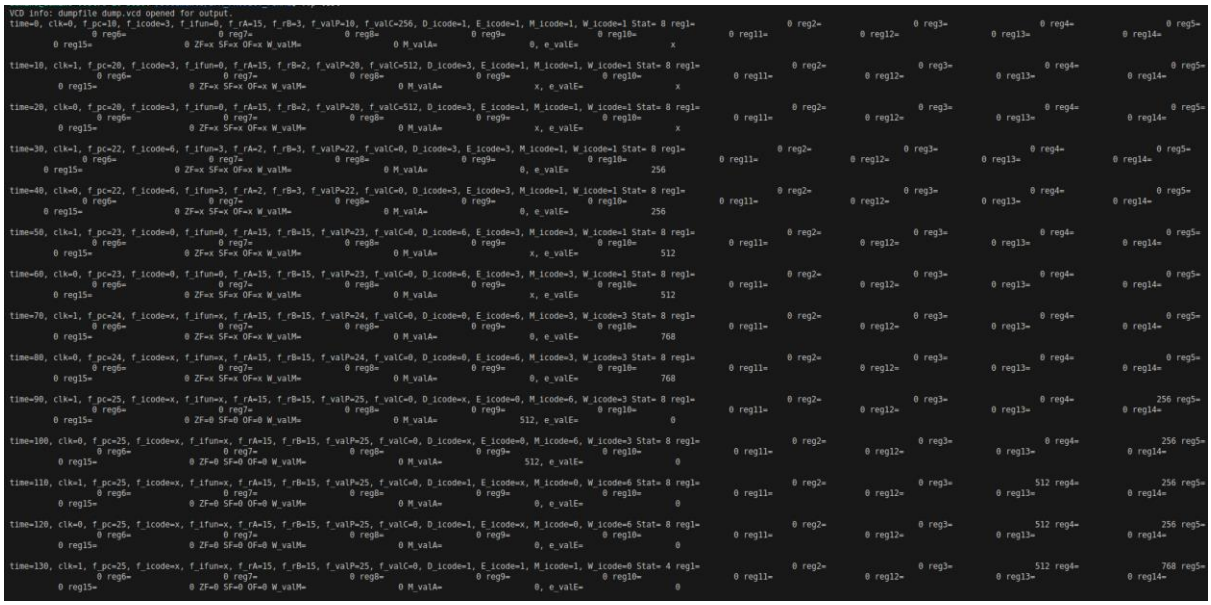
```
memory m(
    .clk(clk),


    // Outputs
    .m_stat(m_stat),
    .m_icode(m_icode),
    .m_valE(m_valE),
    .m_valM(m_valM),
    .m_dstE(m_dstE),
    .m_dstM(m_dstM),

    // Inputs
    .M_stat(M_stat),
    .M_icode(M_icode),
    .M_valE(M_valE),
    .M_valA(M_valA),
    .M_dstE(M_dstE),
    .M_dstM(M_dstM)

);

// Stat Code
assign Stat = W_stat;

// Control Logic
wire W_stall, D_stall, F_stall;
wire M_bubble, E_bubble, D_bubble;

    // F register
always @(posedge clk)
begin
    if (F_stall != 1)
        begin
            F_predPC <= f_predPC;
        end
end

PIPE_con pipe_con(
    // Inputs
    .D_icode(D_icode),
    .d_srcA(d_srcA),
    .d_srcB(d_srcB),
    .E_icode(E_icode),
    .E_dstM(E_dstM),
    .e_Cnd(e_Cnd),
    .M_icode(M_icode),
    .m_stat(m_stat),
    .W_stat(W_stat),

    // Outputs
    .W_stall(W_stall),
    .M_bubble(M_bubble),
    .E_bubble(E_bubble),
    .D_bubble(D_bubble),
    .D_stall(D_stall),
    .F_stall(F_stall)
);




// D register
always @(posedge clk)
begin
```

```
// D register
always @(posedge clk)
begin
    if (D_stall == 0)
        begin
            if (D_bubble == 0)
            begin
                D_stat  <= f_stat;
                D_icode <= f_icode;
                D_ifun  <= f_ifun;

                D_valC  <= f_valC;
                D_valP  <= f_valP;

                D_rA    <= f_rA;
                D_rB    <= f_rB;


            end
            else
            begin
                D_stat  <= 4'b1000;

                D_valC  <= 0;
                D_valP  <= 0;

                D_icode <= 1;
                D_ifun  <= 0;

                D_rA    <= 0;
                D_rB    <= 0;

            end
        end
end

// E register
always @(posedge clk)
begin
    if (E_bubble == 1)
        begin
            E_stat  <= 4'b1000;
            E_icode <= 1;
            E_ifun  <= 0;


            E_dstE  <= 0;
            E_dstM  <= 0;


            E_valA  <= 0;
            E_valB  <= 0;
            E_valC  <= 0;


            E_srcA  <= 0;
            E_srcB  <= 0;
        end
    else
        begin
            E_stat  <= d_stat;
            E_icode <= d_icode;
            E_ifun  <= d_ifun;
```

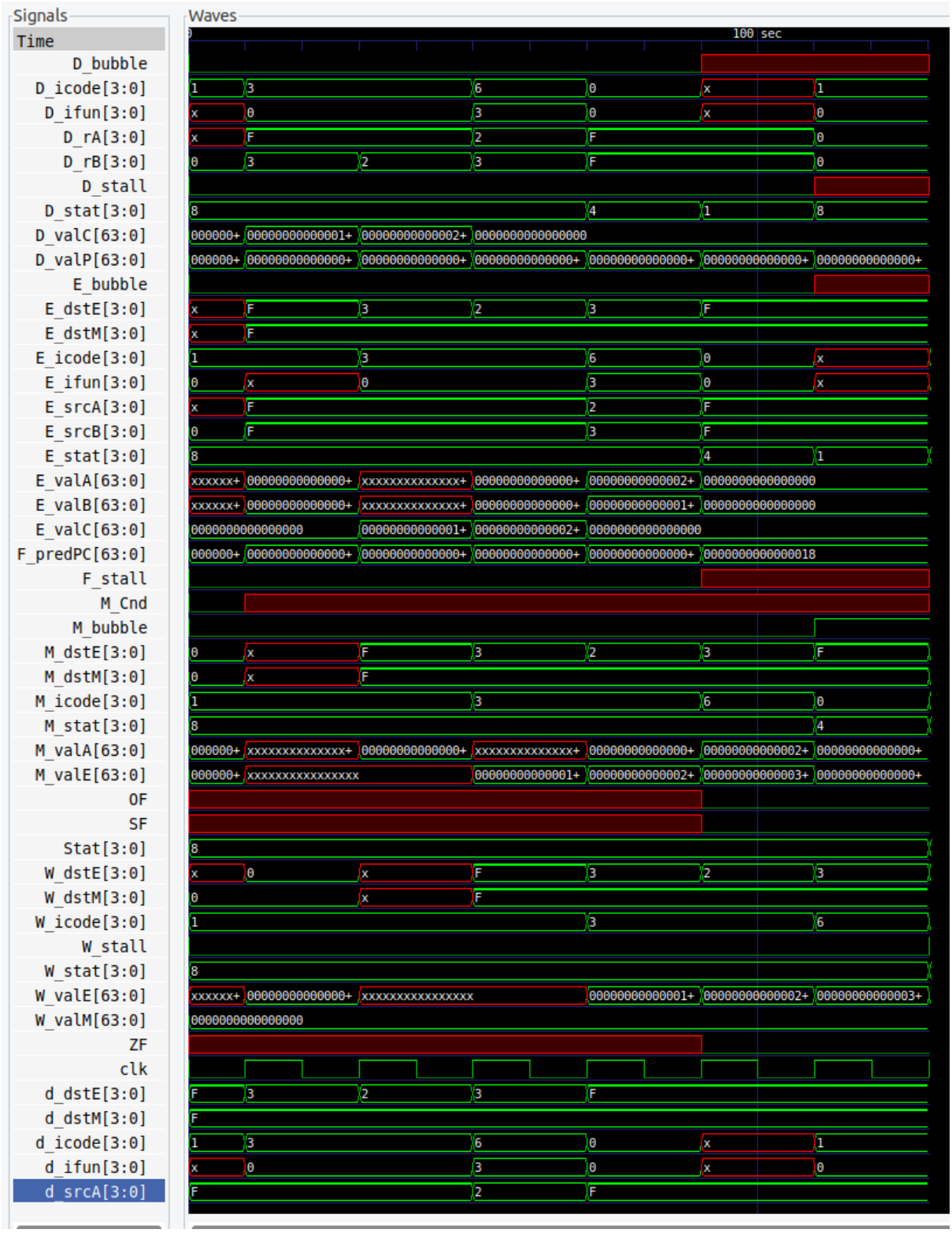
```
            E_srcA  <= d_srcA;
            E_srcB  <= d_srcB;


            E_valA  <= d_valA;
            E_valB  <= d_valB;
            E_valC  <= d_valC;

            E_dstE  <= d_dstE;
            E_dstM  <= d_dstM;

        end
end

// W register
always @(posedge clk)
begin
    if (W_stall != 1)
        begin
            W_stat  <= m_stat;
            W_icode <= m_icode;

            W_dstE  <= m_dstE;
            W_dstM  <= m_dstM;

            W_valE  <= m_valE;
            W_valM  <= m_valM;


        end
end
// M register
always @(posedge clk)
begin
    if (M_bubble == 1)
        begin
            M_stat  <= 4'b1000;
            M_icode <= 1;
            M_Cnd   <= 0;

            M_dstE  <= 0;
            M_dstM  <= 0;

            M_valE  <= 0;
            M_valA  <= 0;


        end
    else
        begin
            M_stat  <= e_stat;
            M_icode <= e_icode;
            M_Cnd   <= e_Cnd;

            M_dstE  <= e_dstE;
            M_dstM  <= e_dstM;

            M_valE  <= e_valE;
            M_valA  <= e_valA;


        end
end
```

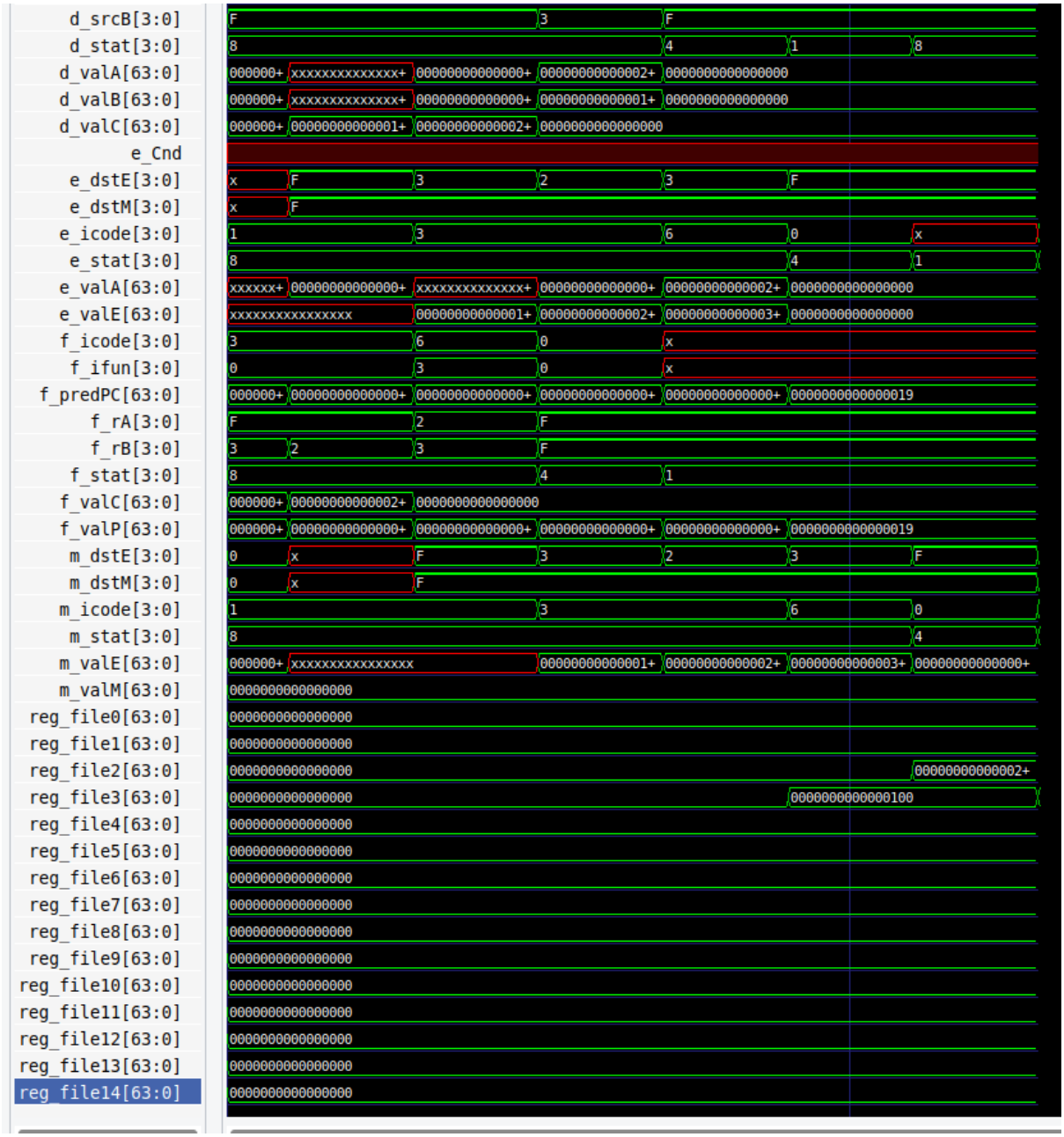
```
    initial begin

        $dumpfile("dump.vcd");
        $dumpvars;

    end

    initial begin
        clk <= 0;
        forever begin
            #10 clk <= ~clk;
        end
    end

    initial begin
        $monitor("time=%0d, clk=%0d, f_pc=%0d, f_icode=%0d, f_ifun=%0d, f_rA=%0d, f_rB=%0d, f_valP=%0d, f_valC=%0d, D_icode=%0d,
         E_icode=%0d, M_icode=%0d, W_icode=%0d Stat=%d reg1=%d reg2=%d reg3=%d reg4=%d reg5=%d reg6=%d reg7=%d reg8=%d reg9=%d reg10=%d
         reg11=%d reg12=%d reg13=%d reg14=%d reg15=%d ZF=%d SF=%d OF=%d W_valM=%d M_valA=%d, e_valE=%d\n",
            $time, clk, f_predPC, f_icode, f_ifun, f_rA, f_rB, f_valP, f_valC, D_icode, E_icode, M_icode, W_icode, Stat, reg_file0,
            reg_file1, reg_file2, reg_file3, reg_file4, reg_file5, reg_file6, reg_file7, reg_file8, reg_file9, reg_file10, reg_file11,
            reg_file12, reg_file13, reg_file14, ZF, SF, OF, W_valM, M_valA, e_valE);
    end

    always @(*)
    begin
        if(Stat == 4'b0100)
            begin
                $finish;
            end
    end


endmodule
```

OUTPUTS:



GTKWAVES:


Signals
Waves
100 sec
Time
D_bubble
D_icode[3:0]
D_ifun[3:0]
D_rA[3:0]
D_rB[3:0]
D_stall
D_stat[3:0]
D_valC[63:0]
D_valP[63:0]
E_bubble
E_dstE[3:0]
E_dstM[3:0]
E_icode[3:0]
E_ifun[3:0]
E_srcA[3:0]
E_srcB[3:0]
E_stat[3:0]
E_valA[63:0]
E_valB[63:0]
E_valC[63:0]
F_predPC[63:0]
F_stall
M_Cnd
M_bubble
M_dstE[3:0]
M_dstM[3:0]
M_icode[3:0]
M_stat[3:0]
M_valA[63:0]
M_valE[63:0]
OF
SF
Stat[3:0]
W_dstE[3:0]
W_dstM[3:0]
W_icode[3:0]
W_stall
W_stat[3:0]
W_valE[63:0]
W_valM[63:0]
ZF
clk
d_dstE[3:0]
d_dstM[3:0]
d_icode[3:0]
d_ifun[3:0]
d_srcA[3:0]



CHALLENGES FACED DURING PIPELINE IMPLEMENTATION:

Implementing a pipeline architecture for the y86 processor presents several challenges that need to be addressed to ensure the proper functioning of the processor. Here are some of the key challenges:

1. Hazards:
    - Hazards occur when one instruction depends on the result of a previous instruction, which may not yet be available due to the pipeline's stages.
    - Handling these dependencies is essential to prevent incorrect results, and it requires additional logic to ensure that instructions are executed in the correct order.
    - This is resolved by introducing bubble in appropriate stage.
2. Branches and Jumps:

- Branches and jumps can also create challenges in a pipelined architecture, as they can disrupt the normal flow of instructions in the pipeline.
- To handle these cases, the pipeline must detect these instructions and introduce bubble in appropriate stage so that the mis predicted branch is taken care of.

3. Data dependencies:
    - When multiple instructions need to access the same data, such as a register or memory location, they may contend for that resource, leading to delays or incorrect results.
    - We resolved this issue using data forwarding.
4. Return Instruction:
    - In case of a ret instruction, we can never predict the target location as we did in jump.
    - Therefore, we have to stall/bubble the instruction succeeding ret instructions and wait till the ret instruction reaches the memory stage, to find out the exact target location.

5. Instruction set design:

    - The y86 instruction set is relatively simple, but some instructions require multiple pipeline stages to complete, which can introduce additional complexity into the pipeline design.
    - The pipeline must also support all of the y86 instructions, including complex instructions like the divide instruction.