Developing the processor architecture based on the Y86-64 design

Merugu Nanditha - 2020102061

Nitin Shrinivas - 2020102046

Introduction to Y86-64 processors

The Y86-64 processor which is a subset of the X86-64 processor will contain the following elements.

- **Program registers** - 15 registers of 64 bits (8 bytes) each.
- **Conditional Codes** - Contains 3 single bit flags which are Zero flag, Sign flag, Overflow flag.
- **Program counter** - 64-bit (8 byte) register containing the address of the next instruction.
- Program Status - 1 bit element which stores the binary value if the operation has some error or not.
- **Memory** - It is a large array where there can be both memory and the instructions. Each memory element is 64-bit long.

Types of implementations

There are mainly two types of implementations of the Y86-64 processor.

1. **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.
2. **Pipelining** - We will add the registers in between each of the registers and then we need to take the clock of the largest time taking process and go on taking the clock cycles at that rate so

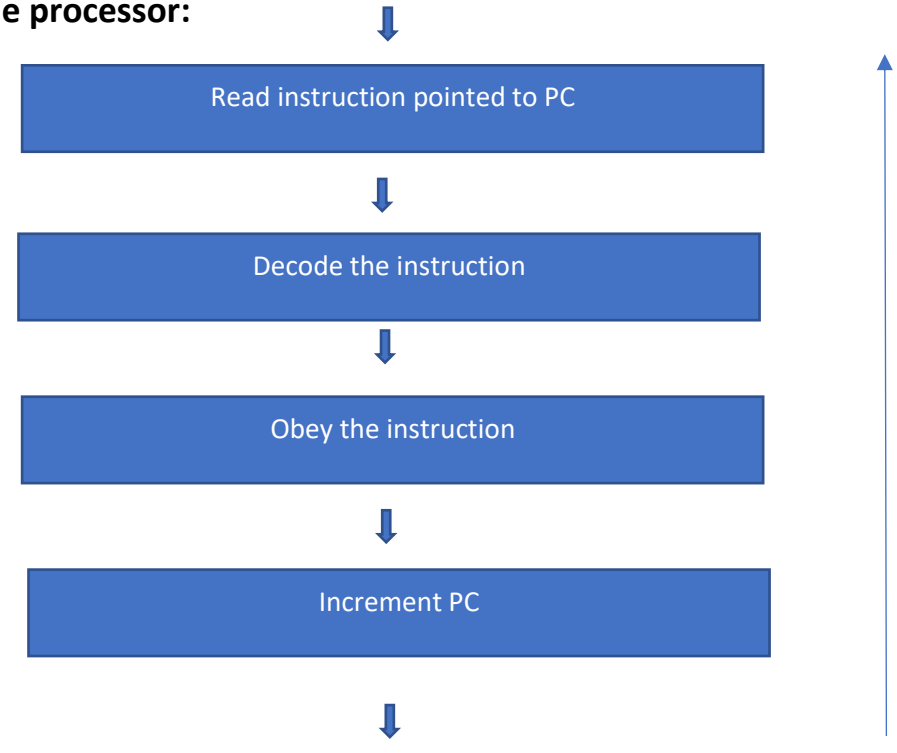
that we can complete many instructions simultaneously of different instructions.

Sequential Processor

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

Stage	Register(s)	description
1. Fetch	Icode,ifun	Read instruction byte
	rA,rB	Read register byte
	valC	Read constant word
	valP	Compute next PC
2. Decode	valA,srcA	Read operand A
	valB,srcB	Read operand B
3. Execute	valE	Perform ALU operation
	cnd	Set/use condition Code
4. Memory	valM	Memory Read/write
5. Writeback	dstE	Write back ALU result
	dstM	Write back Mem result
6. PC Update	PC	Update PC

Flow of the processor:



1. Fetch

- 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.

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]`.

- 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.

  - If the value of the icode = 0, then it is a halt operation and we do not want any of the variables `rA` and `rB` and `valC` and 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

```
valP = PC + 64'd1;
```

  - If the value of the icode = 1, then it is a nop operation and we do not want any of the variables `rA` and `rB` and `valC` and 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).

```
valP = PC + 64'd1;
```

- 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;
```

- 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;
```

- 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;
```

- 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;
```

- 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;
```

- 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;
```

- 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 funciton. 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;
```

- 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;
```

Code:

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

```
input clk;
input [63:0] PC;

output reg [3:0] icode;
output reg [3:0] ifun;
output reg [3:0] rA;
output reg [3:0] rB;
output reg [63:0] valP;
output reg [63:0] valC;
output reg hlt;
output reg inst_valid;
output reg mem_error;

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

// initial begin
//     inst_mem[50]=8'b01100000; // add operation
//     inst_mem[51]=8'b00100011; // rA(2) and rB(3)

//     inst_mem[52]=8'b01100001; // subtrat operation
//     inst_mem[53]=8'b00100001; // rA(2) and rC(1)

//     inst_mem[54]=8'b00010000; // no operation

//     inst_mem[55]=8'b01100000; // add operation
//     inst_mem[56]=8'b00100011; // rA(2) and rB(3)

//     inst_mem[57]=8'b00010000; // no operation

//     inst_mem[58]=8'b00000000; // halt
// end

initial begin
    inst_mem[0] = 8'b00110000;  // A = 5;
    inst_mem[1] = 8'b00000000;  // rax
    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'b00000000;
    inst_mem[9] = 8'b00000110;

    inst_mem[10] = 8'b00110000;  // B =6;
    inst_mem[11] = 8'b00000011;  // rbx
    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'b00000000;
    inst_mem[19] = 8'b00000101;

    inst_mem[20] = 8'b01100000;  //a and b
    inst_mem[21] = 8'b00000011;

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

    // inst_mem[25] = 8'b00100000;  //rr mov
    // inst_mem[26] = 8'b00000001;

    // inst_mem[26] = 8'b00010000;  //nop
    // inst_mem[27] = 8'b00010000;  //nop
    // inst_mem[28] = 8'b00010000;  //nop
    // inst_mem[29] = 8'b00010000;  //nop



    // inst_mem[30] = 8'b01100000;  //a+b
    // inst_mem[31] = 8'b00010011;

end

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];

    if(icode == 4'b0000) begin          // halt operation
      hlt = 1;
      valP = PC + 64'd1;
    end

    else if (icode == 4'b0001) begin    // nop operation
      valP = PC + 64'd1;
    end

    else if (icode == 4'b0010) begin    // cmovxx ----> conditional move
      rA = inst[8:11];
      rB = inst[12:15];
      valP = PC+64'd2;
    end

    else if (icode == 4'b0011) begin    // irmov operation
      rB = inst[12:15];
      valC = inst[16:79];
      valP = PC + 64'd10;
    end

    else if (icode == 4'b0100) begin    // rmmov operation
      rA = inst[8:11];
      rB = inst[12:15];
      valC = inst[16:79];
      valP = PC + 64'd10;
    end

    else if (icode == 4'b0101) begin    // mrmov operation
      rA = inst[8:11];
      rB = inst[12:15];
      valC = inst[16:79];
      valP = PC + 64'd10;
    end

    else if (icode == 4'b0110) begin    // OPq operation
      rA = inst[8:11];
```

```
      rB = inst[12:15];
      valP = PC + 64'd2;
    end

    else if (icode == 4'b0111) begin   // jXX ----> jump operation
      valC = inst[8:71];
      valP = PC + 64'd9;
    end

    else if (icode == 4'b1000) begin   // call operation
      valC = inst[8:71];
      valP = PC + 64'd9;
    end

    else if (icode == 4'b1001) begin   // ret operation
      valP = PC + 64'd1;
    end

    else if (icode == 1010) begin   // push operation
      rA = inst[8:13];
      rB=inst[12:15];
      valP = PC + 64'd2;
    end

    else if (icode == 1011) begin   // pop operation
      rA = inst[8:13];
      rB=inst[12:15];
      valP = PC + 64'd2;
    end

    else inst_valid=0;

end

endmodule
```

2. 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] reg_mem0;
output reg [63:0] reg_mem1;
output reg [63:0] reg_mem2;
output reg [63:0] reg_mem3;
output reg [63:0] reg_mem4;
output reg [63:0] reg_mem5;
output reg [63:0] reg_mem6;
output reg [63:0] reg_mem7;
output reg [63:0] reg_mem8;
output reg [63:0] reg_mem9;
output reg [63:0] reg_mem10;
output reg [63:0] reg_mem11;
output reg [63:0] reg_mem12;
output reg [63:0] reg_mem13;
output reg [63:0] reg_mem14;
```

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.

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

  ```
  valA = reg_mem[rA];
  ```

  - 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].

  ```
  valA = reg_mem[rA];
  ```

- 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].

```
valA = reg_mem[rA];
valB = reg_mem[rB];
```

- 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].

```
valA = reg_mem[rA];
valB = reg_mem[rB];
```

- 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].

```
valB = reg_mem[4];
```

- 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. i.e., valA <-- R[%esp] and valB<--R[%esp].

```
        valA = reg_mem[4];
        valB = reg_mem[4];
```

- 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 i.e., valB<--R[%esp] and the value of the register rA is stored in valA.

```
  valA = reg_mem[rA];
        valB = reg_mem[4];
```

- 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].

```
valA = reg_mem[4];
valB = reg_mem[4];
```

Write back:

- 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:

```
reg_mem0=reg_mem[0];
    reg_mem1=reg_mem[1];
    reg_mem2=reg_mem[2];
    reg_mem3=reg_mem[3];
    reg_mem4=reg_mem[4];
    reg_mem5=reg_mem[5];
    reg_mem6=reg_mem[6];
    reg_mem7=reg_mem[7];
    reg_mem8=reg_mem[8];
    reg_mem9=reg_mem[9];
    reg_mem10=reg_mem[10];
    reg_mem11=reg_mem[11];
    reg_mem12=reg_mem[12];
    reg_mem13=reg_mem[13];
    reg_mem14=reg_mem[14];
```

- If icode is 2. Then this is cmovxx operation. During write back stage valE value is stored back in register i.e., R[rB]<-- valE. ValE is taken from the execute stage.

```
reg_mem[rB]=valE;
```

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

```
 reg_mem[rB] = valE;
```

- 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.

```
reg_mem[rA] = valE;
```

- 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.

```
reg_mem[rB] = valE;
```

- 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.

```
reg_mem[4] = valE;
```

- 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.

- 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.

- 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.

Code:

```
module
decode(clk,icode,rA,rB,valA,valB,valC,valE,valM,cnd,reg_mem0,reg_mem1,r
eg_mem2,reg_mem3,reg_mem4,reg_mem5,reg_mem6,reg_mem7,reg_mem8,reg_mem9,
reg_mem10,reg_mem11,reg_mem12,reg_mem13,reg_mem14);

input clk;
input [3:0] icode;
reg [63:0] reg_mem [14:0];
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] test;
input cnd;

output reg [63:0] reg_mem0;
output reg [63:0] reg_mem1;
```

```
output reg [63:0] reg_mem2;
output reg [63:0] reg_mem3;
output reg [63:0] reg_mem4;
output reg [63:0] reg_mem5;
output reg [63:0] reg_mem6;
output reg [63:0] reg_mem7;
output reg [63:0] reg_mem8;
output reg [63:0] reg_mem9;
output reg [63:0] reg_mem10;
output reg [63:0] reg_mem11;
output reg [63:0] reg_mem12;
output reg [63:0] reg_mem13;
output reg [63:0] reg_mem14;

 initial begin
     reg_mem[0]=reg_mem0;
     reg_mem[1]=reg_mem1;
     reg_mem[2]=reg_mem2;
     reg_mem[3]=reg_mem3;
     reg_mem[4]=reg_mem4;
     reg_mem[5]=reg_mem5;
     reg_mem[6]=reg_mem6;
     reg_mem[7]=reg_mem7;
     reg_mem[8]=reg_mem8;
     reg_mem[9]=reg_mem9;
     reg_mem[10]=reg_mem10;
     reg_mem[11]=reg_mem11;
     reg_mem[12]=reg_mem12;
     reg_mem[13]=reg_mem13;
     reg_mem[14]=reg_mem14;

 end

always @(*)
begin
    // if(icode == 4'b0000) begin        // halt
    //   valA = 0;
    //   valB = 0;
    // end

    // else if(icode == 4'b0001) begin   // no operation
    //   valA = 0;
    //   valB = 0;
    // end

    if(icode == 4'b0010) begin  // cmove operation
      valA = reg_mem[rA];
    end
```

```
else if(icode == 4'b0100) begin  // rmmove operation
  valA = reg_mem[rA];
  valB = reg_mem[rB];
end

else if(icode == 4'b0101) begin  // mrmove operation
  valA = reg_mem[rA];
  valB = reg_mem[rB];
end

else if(icode == 4'b0110) begin  // OPq
  valA = reg_mem[rA];
  valB = reg_mem[rB];
end

else if(icode == 4'b0111) begin  // jump operation
  valA = reg_mem[rA];
  valB = reg_mem[rB];
end

else if(icode == 4'b1000) begin  // call operation
  valB = reg_mem[4];
end

else if(icode == 4'b1001) begin  // return operation
  valA = reg_mem[4];
  valB = reg_mem[4];
end

else if(icode == 4'b1010) begin  // push operation
  valA = reg_mem[rA];
  valB = reg_mem[4];
end

else if(icode == 4'b1011) begin //pop operation
  valA = reg_mem[4];
  valB = reg_mem[4];
end

reg_mem0=reg_mem[0];
reg_mem1=reg_mem[1];
reg_mem2=reg_mem[2];
reg_mem3=reg_mem[3];
reg_mem4=reg_mem[4];
reg_mem5=reg_mem[5];
reg_mem6=reg_mem[6];
reg_mem7=reg_mem[7];
```

```
    reg_mem8=reg_mem[8];
    reg_mem9=reg_mem[9];
    reg_mem10=reg_mem[10];
    reg_mem11=reg_mem[11];
    reg_mem12=reg_mem[12];
    reg_mem13=reg_mem[13];
    reg_mem14=reg_mem[14];
end

always @(negedge clk)
begin
    if(icode==4'b0010) begin //cmovxx
      if(cnd==1'b1) reg_mem[rB]=valE;
    end
    else if(icode == 4'b0011) begin        // irmovq $0xx rB
      reg_mem[rB] = valE;       // Here the rB register will store the
final result
    end

    else if(icode == 4'b0101) begin   // mrmovq D(rB) rA
      reg_mem[rA] = valE;     // Here the rA register will store the
final result
    end

    else if(icode == 4'b0110) begin  // OPq rA rB
      reg_mem[rB] = valE;     // Here the rB register will store the
final result
    end

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

    else if(icode == 4'b1001) begin  // ret
      reg_mem[4] = valE;      // reg_mem[4] is the %esp and we upadate
the stack pointer
    end

    else if(icode == 4'b1010) begin  // pushq
       reg_mem[4] = valE;     // In push we first decrement the address
and then push the data.
    end

    else if(icode == 4'b1011) begin  // popq
        reg_mem[4] = valE;   // In this first we pop out the data from
the stack and then increment the address.
```

```
        reg_mem[rA] = valM;    // So, the popped out data is again
restored in register  rA
    end
    reg_mem0=reg_mem[0];
    reg_mem1=reg_mem[1];
    reg_mem2=reg_mem[2];
    reg_mem3=reg_mem[3];
    reg_mem4=reg_mem[4];
    reg_mem5=reg_mem[5];
    reg_mem6=reg_mem[6];
    reg_mem7=reg_mem[7];
    reg_mem8=reg_mem[8];
    reg_mem9=reg_mem[9];
    reg_mem10=reg_mem[10];
    reg_mem11=reg_mem[11];
    reg_mem12=reg_mem[12];
    reg_mem13=reg_mem[13];
    reg_mem14=reg_mem[14];

end

endmodule
```

3. Execute operation

- 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 where in 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.

Appendix (To perform the flag operations)

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:

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

```
valE = 64'd0 + valC;
```

- If the value of the icode =4, then the function is rmmov then the value of the valE = valB + valC.

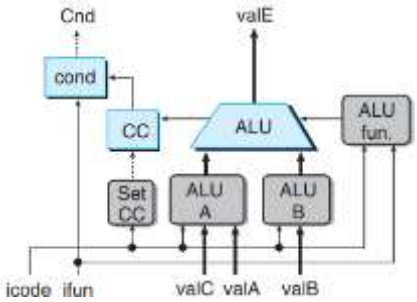
```
valE = valB + valC;
```

- If the value of the icode =5, then the function is mrmov, so we have valE = valB + valC.

```
valE = valB + valC;
```

- 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.

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

- If the value of the icode= 8, then the function is the call function then 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.

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

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

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

Code:

```
`include "alu.v"

module execute(clk,icode,ifun,valA,valB,valC,valE,ZF,OF,SF,cnd);

output reg cnd=0;
```

```
input [63:0]valA;
input [63:0]valB;
input [63:0]valC;
wire [63:0]Add_out;
wire [63:0]Sub_out;
wire [63:0]Xor_out;
wire [63:0]And_out;
wire OF_Add;
wire OF_Sub;
Add A(valA,valB,Add_out,OF_Add);
Sub B(valA,valB,Sub_out,OF_Sub);
Xor C(valA,valB,Xor_out);
And D(valA,valB,And_out);
input [3:0]icode;
input [3:0]ifun;
input clk;
output reg [63:0]valE;
output reg ZF;
output reg OF;
output reg SF;
wire Xor_1;
wire Xor_2;
xor E(Xor_1,valA[63],valB[63]);
xor F(Xor_2,valA[63],valE[63]);
wire out;
xor G(out,OF,SF);


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

  if(icode==4'b0110) //OPq
  begin
    OF=0;
    SF=0;
    ZF=0;
    if(ifun==4'b0000)   //Add
    begin
      valE=Add_out;
      if((Xor_1 == 0) && (Xor_2 == 1))  OF = 1;
      if(valE[63] == 1)  SF = 1;
      if(valE[63:0]==0)  ZF = 1;
    end
    else if(ifun==4'b0001) //Sub
    begin
      valE=Sub_out;
      if((Xor_1 == 1) && (Xor_2) == 1)  OF = 1;
```

```
    if(valE[63] == 1)  SF = 1;
    if(valE[63:0]==0)  ZF = 1;
  end
  else if(ifun==4'b0010) //And
  begin
      valE=And_out;
      if(valE[63] == 1)  SF = 1;
      if(valE[63:0]==0)  ZF = 1;
  end
  else if(ifun==4'b0011) //Xor
  begin
      valE=Xor_out;
      if(valE[63] == 1)  SF = 1;
      if(valE[63:0]==0)  ZF = 1;
  end
end


else if(icode==4'b0010) //cmoveX
begin
    if(ifun==4'b0000) //rrmovq
    begin
        cnd=1;
    end
    else if(ifun==4'b0001) //cmovle
    begin
        if(out==1 || ZF==1) cnd=1;
    end
    else if(ifun==4'b0010) //cmovl
    begin
      if(out==1) cnd=1;
    end
    else if(ifun==4'b0011) //cmove
    begin
      if(ZF==1) cnd=1;
    end
    else if(ifun==4'b0100) //cmovne
    begin
      if(ZF==0) cnd=1;
    end
    else if(ifun==4'b0101) //cmovge
    begin
        if(out==0) cnd=1;
    end
    else if(ifun==4'b0110) // cmovg
    begin
        if(out==0 || ZF==0) cnd=1;
    end
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 || ZF==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(ZF==1) cnd=1;
    end
    else if(ifun==4'b0100) //jne
    begin
      if(ZF==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 || ZF==0) cnd=1;
    end
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
  end

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

end
end
endmodule
```

4. Memory stage

- 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:

- It varies from operation to operation as shown in the follows,

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

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

```
Mem_data[valE]=valA;
```

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

```
valM=Mem_data[valE];
```

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

```
Mem_data[valE]=valP;
```

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

- 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 and here we just put the value of valA into the memory of valE that is M[valE] = valA.

- 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 and then here we need to put the value of the valM in the memory of valA that is M[valA] = valM.

Code:

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

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


reg [63:0] Mem_data[1023:0];


always @(*)
begin
  if(icode==4'b0100) //rmmovq
```

```
  begin             // we need to find the value of the destination
i.e., valB+valC=valE.
    Mem_data[valE]=valA;  // Now we need to put the value stored in
valA to valE
  end
  else if(icode==4'b0101) //mrmovq
  begin             // we need to push the data from the valE to the
register.
    valM=Mem_data[valE]; // Here the destination is valM
  end
  else if(icode==4'b1000) //call
  begin             // We decerement the stack pointer by 8 and add the
program counter into the stack memory
    Mem_data[valE]=valP;
  end
  else if(icode==4'b1001) //ret
  begin              // we have to return the value at the memory of
the stack pointer(valA) to valM
    valM=Mem_data[valA];
  end
  else if(icode==4'b1010) //pushq
  begin              // firstly we have to increase the stack pointer
to valE and then put the value of valA into the memory of valE
    Mem_data[valE]=valA;
  end
  else if(icode==4'b1011) //popq
  begin               //we have to increase the stack pointer and then
the value of valM is the memory of valA
    valM=Mem_data[valA];
  end
  Mem_output=Mem_data[valE];
end
endmodule
```

5. PC (Program counter) update stage

- 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(clk,icode,valP,cnd,valC,valM,PC_updated);

input clk;
input [3:0] icode;
input [63:0] valP;
input [63:0] valC;
input [63:0] valM;
input cnd;
  output reg [63:0]PC_updated;

always @ (posedge clk) begin

  if(icode ==4'b0111) begin    // jump condition
    if(cnd==1'b1) PC_updated=valC;    // if the condition is satisfied
then go to the valC
    else PC_updated=valP;                          // else the jump will
not occue hence PC will be valP
  end

  if(icode == 4'b1000) begin  // call condition
    PC_updated = valC;                             // PC will be ht
eplace where the call function is calling to
```

```
  end

  if(icode == 4'b1001) begin  // return condition
    PC_updated = valM;                          // PC will be the
place where the old stack pointer stored in valM.
  end

  else PC_updated = valP;                       // if nothing of these
instriuctions are present then the vale is valP

end

endmodule
```

Testcase:

We wrote a testcase to check all the stages. So, the operation we considered here is:

- First is the irmovq operation of passing immediate value 6 to a register.
- Second is the irmovq operation of passing immediate value 5 to a register.
- Third is adding those two immediate values.
- Fourth is nop.(no operation)
- Fifth is nop.
- Sixth is hlt. (halt)

Thus, we wrote these cases in fetch stage as follows:

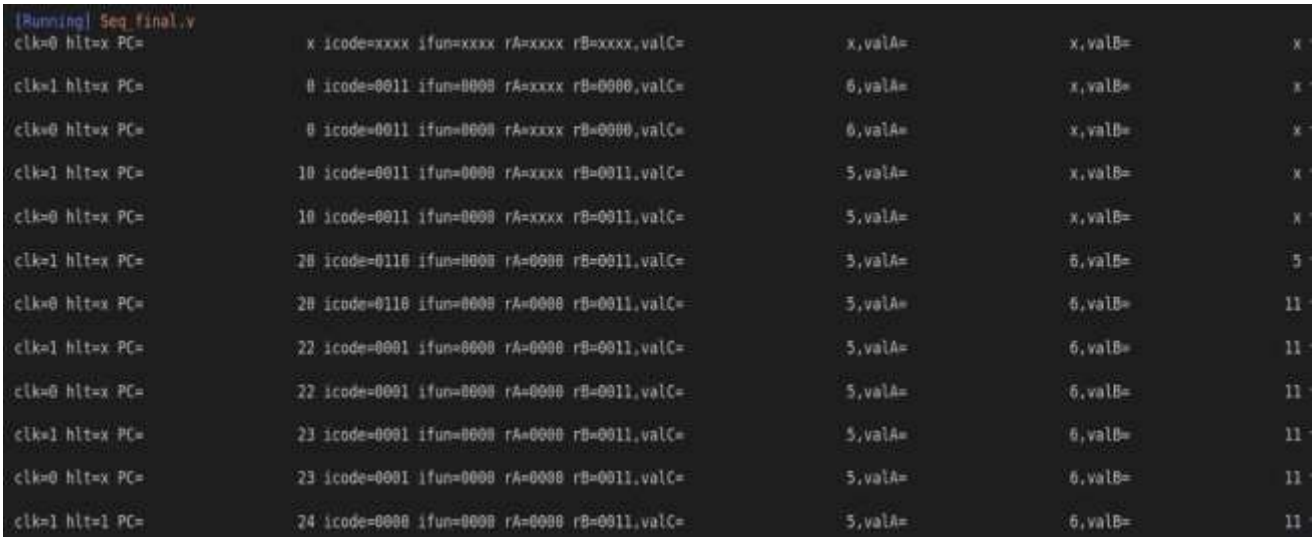
```
inst_mem[0] = 8'b00110000;  // A = 6;
    inst_mem[1] = 8'b00000000;  // rax
    inst_mem[2] = 8'b00000000;
    inst_mem[3] = 8'b00000000;
    inst_mem[4] = 8'b00000000;
    inst_mem[5] = 8'b00000000;
    inst_mem[6] = 8'b00000000;
    inst_mem[7] = 8'b00000000;
    inst_mem[8] = 8'b00000000;
    inst_mem[9] = 8'b00000110;

    inst_mem[10] = 8'b00110000;  // B =5;
    inst_mem[11] = 8'b00000011;  // rbx
    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'b00000000;
    inst_mem[19] = 8'b00000101;

    inst_mem[20] = 8'b01100000;  //a + b
    inst_mem[21] = 8'b00000011;

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

- 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.

Result:

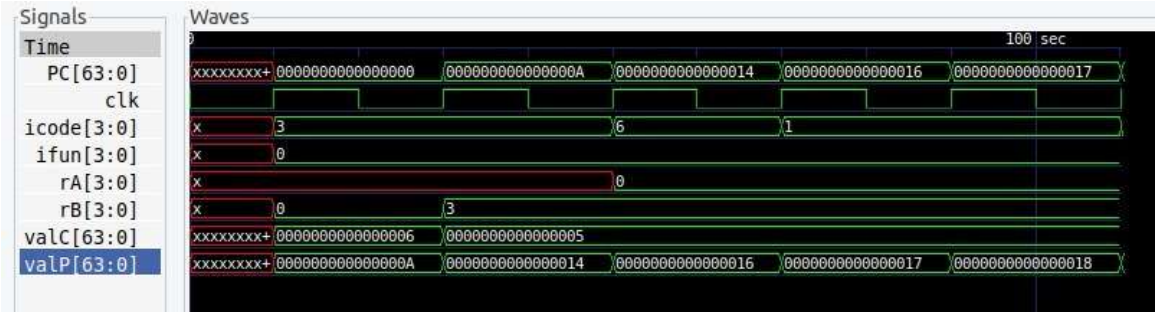


```
valE=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,valM=                x
valE=0000000000000000000000000000000000000000000000000000000000000110,valM=       x
valE=0000000000000000000000000000000000000000000000000000000000000110,valM=       x
valE=0000000000000000000000000000000000000000000000000000000000000101,valM=       x
valE=0000000000000000000000000000000000000000000000000000000000000101,valM=       x
valE=0000000000000000000000000000000000000000000000000000000000001011,valM=       x
valE=0000000000000000000000000000000000000000000000000000000000001011,valM=       x
valE=0000000000000000000000000000000000000000000000000000000000010001,valM=       x
valE=0000000000000000000000000000000000000000000000000000000000010001,valM=       x
valE=0000000000000000000000000000000000000000000000000000000000010001,valM=       x
valE=0000000000000000000000000000000000000000000000000000000000010001,valM=       x
valE=0000000000000000000000000000000000000000000000000000000000010001,valM=       x
```

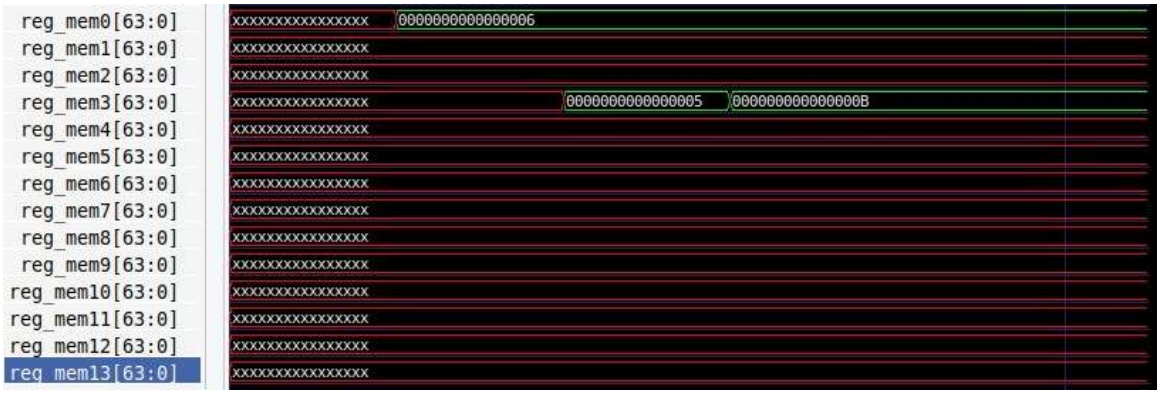
- The above pic contains clk, icode,ifun,rA,rB,valA and valB.
- As we have given value 6 to rA and 5 to rB.So, we see that valA has 6 and valB is 5.
- From here we say fetch and decode are working fine.
- Then we also see the valB output is updated to 11 as we done the sum operation. So, execute is also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

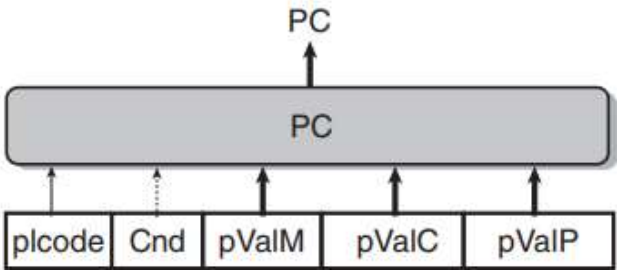
Pipelining

- 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.

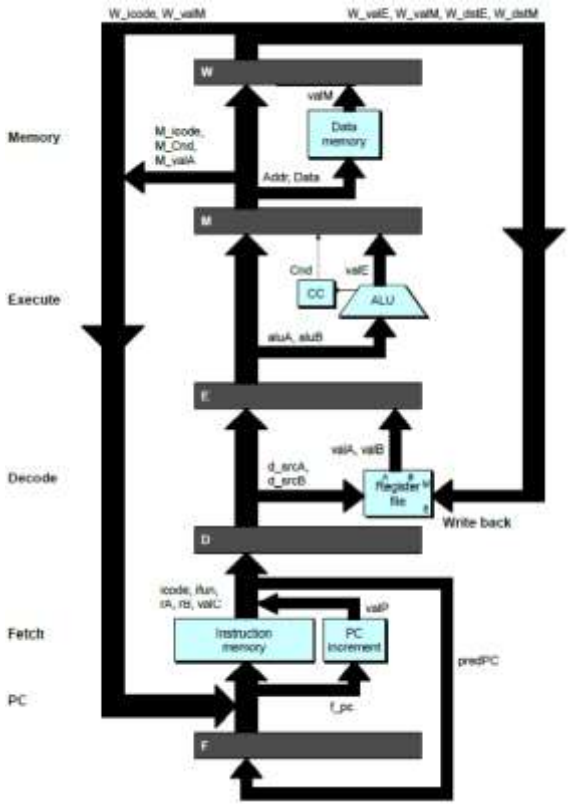
Pipelined y86 implementation:

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 now-current 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 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
Write Back	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 mis-predicted 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.

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.
- **In module Next_PC** (i.e., incrementing PC) ; Based on the value of icode, we write the value of PC. Say if
  - icode = 0 then valP=PC+ 64'd1;
  - icode =1 then valP= PC+64'd1;
  - icode = 2 then valP = PC+ 64'd2;
  - icode=3 then valP= PC+64'd10;

icode = 4 then valP = PC+64'd10;

- icode=5 then valP = PC+64'd10;
- icode = 6 then valP =PC + 64'd2;
- icode =7 then valP = PC + 64'd9;
- icode =8 then valP = PC+64'd9;
- icode = 9 then valP =PC+64'd1;
- icode = 10 then valP = PC+64'd2;
- icode = 11 then valP = PC + 64'd2;

So, we used the same idea in single line as shown below:

```
valP = (need_reg == 1 & need_valC == 1) ? PC + 10 : ((need_reg == 0 & need_valC == 1) ?  PC + 9 : ((need  reg == 0 & need  valC == 1) ? PC +2 : PC+1));
```

- Here, **Need registers** is taken from the module we wrote to check whether the specific icode specification requires the registers or not.

```
assign need_reg = (icode == 4'd0 | icode == 4'd1 | icode == 4'd7 | icode == 4'd8 | icode == 4'd9 ) ? 0 : 1;
```

- And **Need_valC** is taken from the module where we check to find whether the specific icode specifications has valC or not.

```
assign need_valC = (icode == 4'd0 | icode == 4'd1 | icode == 4'd2 | icode == 4'd6 | icode == 4'd9 | icode == 4'd10 | icode == 4'd11) ? 0 : 1;
```

- If registers are needed and valC is present then the valP should be PC+10. If registers are not needed and valC is present then valP should be PC+9. If registers are needed and valC is not present then valP should be PC+2. Thus , this is how valP is calculated in **Next_PC module**.
- We know that if instruction is jXX and call,it will take valP as valC and remaining all will take valP. So, to focus on this we wrote **predict_pc module**.

```
assign predicted_pc = (icode == 4'd7 & icode == 4'd8) ? valC : valP;
```

- For return and jxx other than jump instruction will take valP has W_valM. To assign this, we use **select_PC module.**

Codes:

```
//------------>split
module split(address,werror_icode,werror_ifun);

// In this module we are taking the instructuction address and
splitting the icode and ifun
// Get values of icode and ifun

input [7:0] address;
output [3:0] werror_icode;
output [3:0] werror_ifun;

assign werror_icode = address[3:0];
assign werror_ifun  = address[7:4];

endmodule


//--------> align
module align(rA,rB,valC,need_reg,iremain_addr);

// Get the values of the rA, rB, valC

input [71:0] iremain_addr;
input need_reg;
output [3:0] rA;
output [3:0] rB;
output [63:0] valC;

assign rA = iremain_addr[3:0];
assign rB = iremain_addr[7:4];
// if(need_reg) begin
//     assign valC = iremain_addr[63:0];
// end
// else assign valC = iremain_addr[71:8];

assign valC = need_reg ? iremain_addr[71:8]: iremain_addr [63:0];

endmodule

//-----> updated icode and ifun
module
final_icode_ifun(werror_icode,werror_ifun,imemory_error,icode,ifun);

// This module is to compute the real value of the icode and ifun
// If there is error in the address of the instruction then we have to
nop operation.
// Assign icode to 0001 and ifun to 0000
```

```
input imemory_error;
input [3:0] werror_icode;
input [3:0] werror_ifun;
output [3:0] icode;
output [3:0] ifun;

assign icode = imemory_error ? 4'd1 : werror_icode;
assign ifun = imemory_error ? 4'd0 : werror_ifun;

endmodule


module
need_reg_valC_insterror(icode,need_reg,need_valC,instruction_error);

// we do not want the registers for nop,halt,jump,call,return
// we do not want the valC for halt, nop, cmovxx ,opq, return, pushq,
popq

input [3:0] icode;
output need_reg;
output need_valC;
output instruction_error;

assign instruction_error = (icode == 4'd0 | icode == 4'd1 | icode ==
4'd2 | icode == 4'd3 |icode == 4'd4 | icode == 4'd5 | icode == 4'd6 |
icode == 4'd7 | icode == 4'd8 | icode == 4'd9 | icode == 4'd10 | icode
== 4'd11 ) ? 0 : 1;

assign need_reg = (icode == 4'd0 | icode == 4'd1 | icode == 4'd7 |
icode == 4'd8 | icode == 4'd9 ) ? 0 : 1;
assign need_valC = (icode == 4'd0 | icode == 4'd1 | icode == 4'd2 |
icode == 4'd6 | icode == 4'd9 | icode == 4'd10 | icode == 4'd11) ? 0 :
1;

endmodule

//-----> status
module Status(icode,status,instruction_error,imemory_error);

// 1 is aok
// 2 is halt
// 3 is i_mem error
// 4 is instrution error

input [3:0] icode;
input instruction_error;
input imemory_error;
```

```
output [2:0] status;

assign status = (icode == 4'd0) ? 3'd2 : ((imemory_error == 1) ? 3'd3 :
((instruction_error == 1) ? 3'd4 : 3'd1));

endmodule
```

```
//-----> Incrementing PC based on need registers and valC
module next_PC(valP,PC,need_reg,need_valC);

// valC is 8 bytes
// reg is of 1 byte
// address is of 1 byte

input need_reg;
input need_valC;
input [63:0] PC;
output [63:0] valP;

// if(need_reg == 1 & need_valC == 1) assign valP = PC + 10;
// else if(need_reg == 0 & need_valC == 1) assign valP = PC + 9;
// else if(need_reg == 1 & need_valC == 0) assign valP = PC + 2;
// else assign valP = PC + 1;

assign valP = (need_reg == 1 & need_valC == 1) ? PC + 10 : ((need_reg
== 0 & need_valC == 1) ?  PC + 9 : ((need_reg == 1 & need_valC == 0) ?
PC +2 : PC+1));

endmodule


//-----> predicting PC
module predict_PC(valP,valC,icode,predicted_pc);

// we need to jump directly for jump instrutions to valC
// we need to jump to valC for the call function.

input [3:0] icode;
input [63:0] valP;
input [63:0] valC;
output [63:0] predicted_pc;

assign predicted_pc = (icode == 4'd7 & icode == 4'd8) ? valC : valP;

endmodule
```

```
//--------> Selecting_PC
module
select_PC(predicted_pc,correct_pc,M_cnd,M_icode,W_icode,M_valA,W_valM);

// for jump not satisfied we correct the nxt PC to M_valA
// for return we corerct the pc to W_valM else the predicted PC is the
correct PC

input [63:0] predicted_pc;
input M_cnd;
input [3:0] M_icode;
input [3:0] W_icode;
input [63:0] M_valA;
input [63:0] W_valM;

output [63:0] correct_pc;

assign correct_pc = ((M_cnd == 0) & (M_icode == 4'd7)) ? M_valA :
((W_icode == 4'd9) ? W_valM : predicted_pc);

endmodule
```

2. Decode and Write-Back 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 value 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 is 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 %rspa dn 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 selectvalAvalB 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 an 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:

```
//----------> we need to assign registers to store value based on given
icode.......Here set is happened
module set_register(icode,rA,rB,RA,RB);

// if we have the cmov,rmmovq,opq and push then it is rA
// if return or pop comes then it is RSP or r[4] else it is set to no-
reg

// if we have the mrmovq,rmmovq,opq then it is rB
// if call ,return or pop or push comes then it is RSP or r[4] else it
is set to no-reg

input [3:0] icode;
input [3:0] rA;
input [3:0] rB;
output [3:0] RA;
output [3:0] RB;

assign RA = ((icode == 4'd2) | (icode == 4'd4) | (icode == 4'd6) |
(icode == 4'd10)) ? rA : (((icode == 4'd9) | (icode == 4'd11)) ? 4'h4 :
4'd15);
assign RB = ((icode == 4'd4) | (icode == 4'd5) | (icode == 4'd6))  ? rB
: (((icode == 4'd9)| (icode == 4'd8) | (icode == 4'd10) |(icode ==
4'd11)) ? 4'd4 : 4'd15);

endmodule


//-------> selecting valA and valB
module
selectvalAvalB(valA,valB,icode,valP,RA,RB,e_dstE,e_valE,M_dstM,M_dstE,m
_valM,M_valE,W_dstM,W_valM,W_dstE,W_valE,Ra,Rb);

  output [63:0] valA,valB;
  input [3:0] icode;
    input [3:0] RA,RB;
    input [3:0] e_dstE,M_dstM,M_dstE,W_dstM,W_dstE;
  input [63:0] Ra,Rb,valP,e_valE,m_valM,M_valE,W_valM,W_valE;


  assign valA =
((icode==4'd8|icode==4'd7)?valP:(((RA==e_dstE)&(e_dstE==4'd15))?e_valE:
(((RA==M_dstM)&(M_dstM==4'd15))?m_valM:
(((RA==M_dstE)&(M_dstE==4'd15))?M_valE:(((RA==W_dstE)&(W_dstE==4'd15))?
W_valE:(((RA==W_dstM)&~(W_dstM==4'd15))?W_valM:Ra))))));
```

```
  assign valB =
(((RB==e_dstE)&(e_dstE==4'd15))?e_valE:(((RB==M_dstM)&(M_dstM==4'd15))?
m_valM:(((RB==M_dstE)&~(M_dstE==4'd15))?M_valE:(((RB==W_dstM)&(W_dstM==
4'd15))?W_valM:(((RB==W_dstE)&(W_dstE==4'd15))?W_valE:Rb)))));

endmodule


//---------------> Destination register
module DestSET_E_M(icode,destE,destM,rA,rB);

// set the values of the destination for the valE and valM.
// for add, cmovq,irmovq we set the destination of valE as rB.
// for push pop return and call we set the destination of valE as rsp.
// for mrmovq and pop we set the destination of the ValM to be rA else
the no register for all the other operations. as it is of no use.

input [3:0] icode;
input [3:0] rA,rB;
output [3:0] destE,destM;

assign destE = ((icode == 4'd2) | (icode == 4'd3) | (icode == 4'd6)) ?
rB : (((icode == 4'd8) | (icode == 4'd9) | (icode == 4'd10) | (icode ==
4'd11)) ? 4'd4 : 4'd15);
assign destM = ((icode == 4'd5) | (icode == 4'd11)) ? rA : 4'd15;

endmodule
```

3. 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:

```
`include "alu.v"


//----------->ALU
module ALU(valA,valB,ifun,valE,conditional_codes);
// Here we defined all the operations in OPq and we also set the valE
and conditional codes

input [63:0] valA,valB;
input [3:0] ifun;
output [63:0] valE;
output [2:0] conditional_codes;
wire [63:0] Add_out;
wire [63:0] Sub_out;
wire [63:0] Xor_out;
wire [63:0] And_out;
Add A1(.x(valA),.y(valB),.Sum(Add_out));
Sub A2(.x(valA),.y(valB),.out(Sub_out));
Xor A3(.x(valA),.y(valB),.out(Xor_out));
And A4(.x(valA),.y(valB),.out(And_out));
assign valE= (ifun==4'h0) ? Add_out : ((ifun==4'h1) ? Sub_out : ((ifun
== 4'h2) ? And_out : ((ifun == 4'h3) ? Xor_out : Add_out)));

assign conditional_codes[2] = (valE == 64'h0) ? 1 : 0;
assign conditional_codes[1] = (valE[63] == 1'b1) ? 1 : 0;
assign conditional_codes[0] = ((valA[63] == 0 | valB[63] ==0) &
(valE[63] == 1)) | ((valA[63] == 1 | valB[63] ==1) & (valE[63] == 0)) ?
1 : 0;


endmodule


//--------------------> ValA and ValB
module modified_valA_valB(icode,valC,valA,valB,ValA,ValB);

input [3:0] icode;
input [63:0] valC, valA , valB;
output [63:0] ValA, ValB;
```

```
assign ValA = ((icode==4'd2|icode==4'd6) ? valA :
((icode==4'd3|icode==4'd4|icode==4'd5) ? valC :
  ((icode==4'd7) ? 64'd0 : ((icode==4'd8|icode==4'd10) ? 64'd8 :
  ((icode==4'd9|icode==4'd11) ?
64'b1111111111111111111111111111111111111111111111111111111111111111 :
64'd0)))));

assign ValB = ((icode==4'd2|icode==4'd3|icode==4'd7) ? 64'd0 :
  ((icode==4'd4|icode==4'd5|icode==4'd6|icode==4'd8|icode==4'd9|icode==
4'd10|icode==4'd11) ? valB : 64'd0));

endmodule

//--------------> Destination is found here

module set_dstE(E_dstE,E_icode,e_cnd,e_dstE);

input [3:0] E_icode,E_dstE;
input e_cnd;
output [3:0] e_dstE;

  assign e_dstE = (E_icode==4'h2 & e_cnd ==0 ) ? 4'd15 : E_dstE;

endmodule
```

4. 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 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:

```
//------------> finding the address
module addresses(icode,location,valE,valA);

// This module is to send to set the values of the address of the
location where in if we have the
// rmmovq, mrmovq, pushq, call ---> valE
// return , pop ----> valA
input [3:0] icode;
input [63:0] valE;
input [63:0] valA;
output [63:0] location;

assign location = ((icode == 4'd4 ) | (icode == 4'd5) | (icode == 4'd8)
| (icode == 4'd10)) ? valE : valA;

endmodule


//----------> Set read and write enabled
module set_read_write_enables(icode , write_En, read_En);
// this module is to gice controln over the read-write operation into
the memory.
input [3:0] icode;
output write_En;
output read_En;
```

```
assign write_En = ((icode == 4'd4) | (icode == 4'd8) | (icode ==
4'd10)) ? 1 : 0;
assign read_En  = ((icode == 4'd5) | (icode == 4'd9) | (icode ==
4'd11)) ? 1 : 0;

endmodule


//------->  For finding stat
module set_status_with_memerror(icode,status,new_status,data_memerror);

// if there is memory included in the file and there is data_memory
error then status will be updated.
input [3:0] icode;
input [2:0] status;
input data_memerror;
output [2:0] new_status;

assign new_status = ((icode == 4'd4) | (icode == 4'd5) | (icode ==
4'd8) | (icode == 4'd9) | (icode == 4'd10) | (icode == 4'd11) &
data_memerror) ? 3'd3 : status;
endmodule
```

5. Module for instruction memory

- We have defined the instruction memory as the array of 8 bytes long 1024 lines of code.
- If the value of the PC comes out to be greater than this range then we will have the instruction memory error and then we need to set the status in the remaining stages of the executing.
- We take 8 bits or 1 byte of information at a time and send it into he fetch stage, where it will do the remaining functions that is align, spilt, need_reg, need_valC, and then find the values of rA and rB ,etc.

Code:

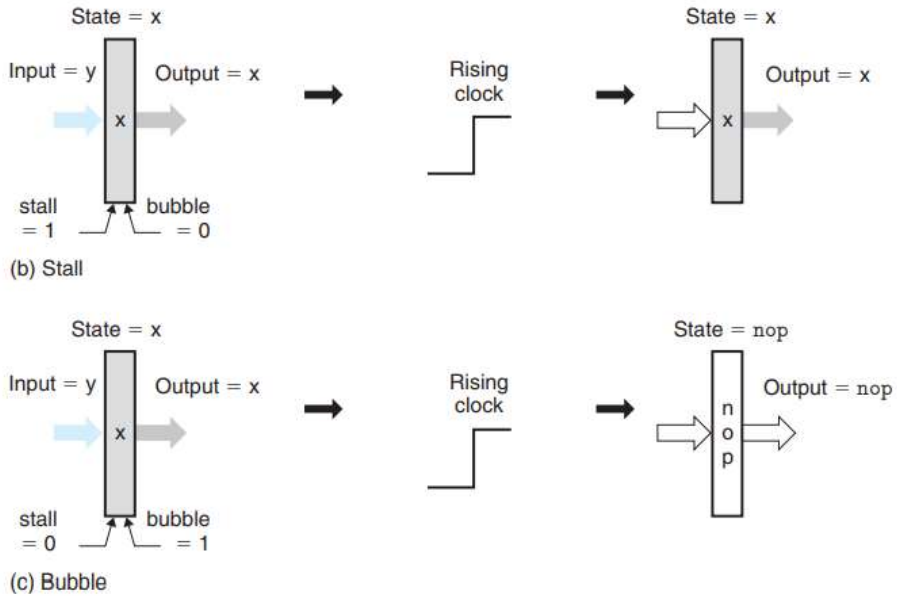
```
module instruction_memory(clk,INST,imemory_error,f_pc);

input clk;
input [63:0] f_pc;
output imemory_error;
```

```
output reg [79:0] INST;

reg [7:0] memory[1023:0];

always @ (negedge clk)
begin
  INST[79:72] <= memory[f_pc+9];
  INST[71:64] <= memory[f_pc+8];
  INST[63:56] <= memory[f_pc+7];
  INST[55:48] <= memory[f_pc+6];
  INST[47:40] <= memory[f_pc+5];
  INST[39:32] <= memory[f_pc+4];
  INST[31:24] <= memory[f_pc+3];
  INST[23:16] <= memory[f_pc+2];
  INST[15:8]  <= memory[f_pc+1];
  INST[7:0]   <= memory[f_pc];
end

assign data_memerror = (f_pc + 10 > 1024) ? 1 : 0;

 initial begin
    memory[0] = 8'b00110000;  // A = 5;
    memory[1] = 8'b00000000;  // rax
    memory[2] = 8'b00000000;
    memory[3] = 8'b00000000;
    memory[4] = 8'b00000000;
    memory[5] = 8'b00000000;
    memory[6] = 8'b00000000;
    memory[7] = 8'b00000000;
    memory[8] = 8'b00000000;
    memory[9] = 8'b00000110;

    memory[10] = 8'b00110000;  // B =6;
    memory[11] = 8'b00000011;  // rbx
    memory[12] = 8'b00000000;
    memory[13] = 8'b00000000;
    memory[14] = 8'b00000000;
    memory[15] = 8'b00000000;
    memory[16] = 8'b00000000;
    memory[17] = 8'b00000000;
    memory[18] = 8'b00000000;
    memory[19] = 8'b00000101;

    memory[20] = 8'b00010000;  //nop
    memory[21] = 8'b00010000;  //nop
    memory[22] = 8'b00010000; // nop


    memory[23] = 8'b01100000;  //a and b
```

```
    memory[24] = 8'b00000011;

    // memory[22] = 8'b00010000;  //nop
    // memory[23] = 8'b00010000;  //nop
    // memory[24] = 8'b00000000; // halt
 end

endmodule
```

6. Pipelining by 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.

```
if(reset)
output_val<=reset_value;
                          else if(En) output_val<=input_val;
```

Code:

```
// In this it is clear that if reset is done then the value should take
the same value as the value it contains.
// But if the bubble occurs then nop operation gets implemented here
module bubble_reg64bit (clk,input_val,output_val,En,reset,reset_value);

input clk,reset,En;
input [63:0] input_val,reset_value;
output reg [63:0] output_val;

always@(*)
begin
  if(reset) output_val<=reset_value;
  else if(En) output_val<=input_val;
end

endmodule


// Using the above in the below, do procces the respective action
module pipeline_reg64
(clk,input_val,output_val,stall,bubble,bubble_val);

input clk,stall,bubble;
input [63:0] input_val,bubble_val;
output [63:0] output_val;

bubble_reg64bit
uut4(clk,input_val,output_val,~stall,bubble,bubble_val);

endmodule


// This is used in writing the data memory
module reg64bit (clk,input_val,output_val,write_En,reset,reset_value);

input clk,reset,write_En;
input [63:0] input_val,reset_value;
output reg [63:0] output_val;

always@(*)
begin
  if(reset) output_val<=reset_value;
  else if(write_En) output_val<=input_val;
end
endmodule
```

Based on this, we run the fetch, decode, execute, memory and write Back in one file. (as shown below in the test bench)

Testcase:

- We wrote a testcase for adding two operands.
- So, the first instruction should be irmovq(0011), second should be also irmovq(0011) , third should be OPq(0110) (ifun should be 0 for Add) and fourth is nop operation (0001).
- We wrote the testbench and tried verifying whether all the instructions are working parallel but not as sequential.
- This testbench contains all the values taken up by each stage register file.

Testbench for this is:

```
// This is the first attempt to the pipelining.
// Here we have some clock problems which an be rectified using reset
and enables.
// The execute stage is giving output as z for valB and valA is
verified to be correct.
//
// This is perfectly workiing for operations without valA and valB.
// Data forwarding is not completly attempted in this.

//.........................................Worked for only few
cases. This gave us to start in new
logic.................................................

`include "fetch.v"
`include "pc_update.v"
`include "fetch_reg.v"
`include "decode_reg.v"
`include "execute_reg.v"
`include "memory_reg.v"
`include "writeBack_reg.v"

module pipe_processor;

reg clk;
reg [63:0]PC;
wire [63:0]PC_updated;
```

```
wire [3:0]  f_status;
wire [63:0] f_valC;
wire [63:0] f_valP;
wire [3:0]  f_icode;
wire [3:0]  f_ifun;
wire [3:0]  f_rA;
wire [3:0]  f_rB;
wire [63:0] f_pc;

wire cnd;

wire [3:0]  d_status;
wire [63:0] d_valC;
wire [63:0] d_valP;
wire [3:0]  d_icode;
wire [3:0]  d_ifun;
wire [3:0]  d_rA;
wire [3:0]  d_rB;
wire [63:0] d_pc;

wire [3:0]  e_status;
wire [63:0] e_valC;
wire [63:0] e_valP;
wire [3:0]  e_icode;
wire [3:0]  e_ifun;
wire [3:0]  e_rA;
wire [3:0]  e_rB;
wire [63:0] e_pc;
wire [63:0] e_valA;
wire [63:0] e_valB;

wire [3:0]  m_status;
wire [63:0] m_valC;
wire [63:0] m_valP;
wire [3:0]  m_icode;
wire [3:0]  m_ifun;
wire [3:0]  m_rA;
wire [3:0]  m_rB;
wire [63:0] m_pc;
wire [63:0] m_valA;
wire [63:0] m_valB;
wire [63:0] m_valE;

wire [3:0]  w_status;
wire [63:0] w_valC;
wire [63:0] w_valP;
wire [3:0]  w_icode;
wire [3:0]  w_ifun;
```

```
wire [3:0]  w_rA;
wire [3:0]  w_rB;
wire [63:0] w_pc;
wire [63:0] w_valA;
wire [63:0] w_valB;
wire [63:0] w_valE;
wire [63:0] w_valM;

wire hlt;
wire inst_valid;
wire mem_error;


fetch_reg B1(
  .clk(clk),.predicted_pc(PC_updated),.pipe_line_pc(f_pc)
);

fetch B2(
  .clk(clk),.PC(PC),.icode(f_icode),.ifun(f_ifun),.rA(f_rA),.rB(f_rB),.
valC(f_valC),.valP(f_valP),.hlt(hlt)
);


decode_reg B3(
  .clk(clk),.f_status(f_status),.f_icode(f_icode),.f_ifun(f_ifun),.f_rA
(f_rA),.f_rB(f_rB),.f_valC(f_valC),.f_valP(f_valP),
  .d_status(d_status),.d_icode(d_icode),.d_ifun(d_ifun),.d_rA(d_rA),.d_
rB(d_rA),.d_valC(d_valC),.d_valP(d_valP)
);

execute_reg B4(
  .clk(clk),.d_status(d_status),.d_icode(d_icode),.d_ifun(d_ifun),.d_rA
(d_rA),.d_rB(d_rB),.d_valC(d_valC),.d_valP(d_valP),
  .e_status(e_status),.e_icode(e_icode),.e_ifun(e_ifun),.e_valA(e_valA)
,.e_valB(e_valB),.e_valC(e_valC),.e_valP(e_valP),.e_rA(e_rA),.e_rB(e_rB
)
);

memory_reg B5(
  .clk(clk),.e_rA(e_rA),.e_rB(e_rB),.m_rA(m_rA),.m_rB(m_rB),.e_icode(e_
icode),.e_ifun(e_ifun),.e_valA(e_valA),.e_valB(e_valB),.e_valP(e_valP),
  .e_valC(e_valC),.e_status(e_status),.m_icode(m_icode),.m_status(m_sta
tus),.m_ifun(m_ifun),.m_valE(m_valE),.m_cnd(m_cnd),
  .m_valC(m_valC),.m_valP(m_valP),.m_valA(m_valA),.m_valB(m_valB)
);

writeBack_reg B6(
  .clk(clk),.m_status(m_status),.m_icode(m_icode),.m_rA(m_rA),.m_rB(m_r
B),.m_ifun(m_ifun),.m_valE(m_valE),.m_valC(m_valC),.m_valP(m_valP),
```

```
  .w_status(w_status),.w_icode(w_icode),.w_ifun(w_ifun),.w_valE(w_valE)
,.w_valM(w_valM),.w_cnd(w_cnd),.w_rA(w_rA),.w_rB(w_rB),
  .w_valP(w_valP),.w_valC(w_valC)
);

pc_update B7(
  .clk(clk),.icode(f_icode),.valC(f_valC),.valP(f_valP),.PC_updated(PC_
updated)
);


initial begin

  clk = 0;

     #5 clk=~clk;PC=64'd0;
    #5 clk=~clk;
    #5 clk=~clk;PC=PC_updated;
    #5 clk=~clk;
    #5 clk=~clk;PC=PC_updated;
    #5 clk=~clk;
    #5 clk=~clk;PC=PC_updated;

end

always@(*) begin
    begin
    $monitor("clk=%d hlt=%d PC=%d icode=%b ifun=%b rA=%b
rB=%b,valC=%d,valA=%d,valB=%d
valE=%b,valM=%d,f_pc=%d\n",clk,hlt,PC,f_icode,f_ifun,f_rA,f_rB,f_valC,w
_valA,w_valB,w_valE,w_valM,w_pc);
    end
  end
endmodule
```

Output:

- From the below picture, it is very clear that during first positive clock, only fetch takes the icode 0011 i.e., irmovq and remaining are not considered.
- But during second positive clock cycle. Decode takes the 0011 and new instruction enters fetch and that is irmovq again (second operand in my test case).

- During third positive cycle, 0011 enters from decode to execute and other 0011 from fetch to decode. And new icode enters fetch i.e., 0110 (OPq).
- So, it proceeds the same as the positive clock cycles comes.
- Thus, the instructions are taking parallel but not sequential.
- Hence, Verified.

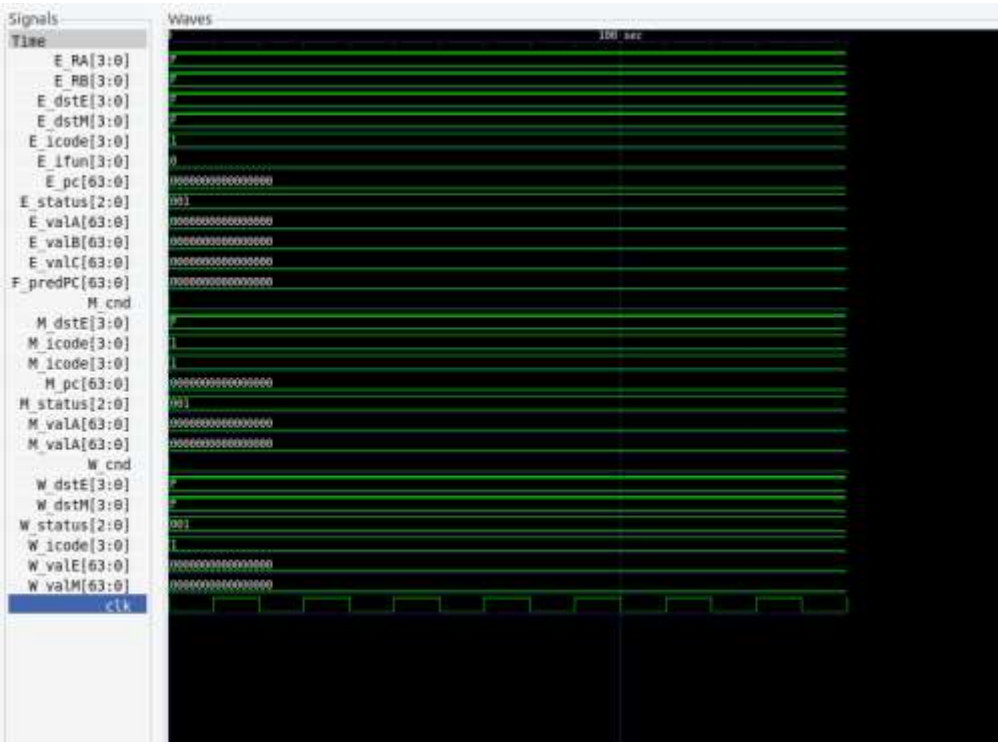
```
[Running] all.v
clk=0 f_icode=xxxx d_icode=xxxx e_icode=xxxx m_icode=xxxx w_icode=xxxx
clk=1 f_icode=0011 d_icode=xxxx e_icode=xxxx m_icode=xxxx w_icode=xxxx
clk=1 f_icode=0011 d_icode=0011 e_icode=xxxx m_icode=xxxx w_icode=xxxx
clk=0 f_icode=0011 d_icode=0011 e_icode=xxxx m_icode=xxxx w_icode=xxxx
clk=1 f_icode=0011 d_icode=0011 e_icode=xxxx m_icode=xxxx w_icode=xxxx
clk=1 f_icode=0011 d_icode=0011 e_icode=0011 m_icode=xxxx w_icode=xxxx
clk=0 f_icode=0011 d_icode=0011 e_icode=0011 m_icode=xxxx w_icode=xxxx
clk=1 f_icode=0110 d_icode=0011 e_icode=0011 m_icode=0011 w_icode=xxxx
clk=1 f_icode=0110 d_icode=0110 e_icode=0011 m_icode=0011 w_icode=xxxx
clk=0 f_icode=0110 d_icode=0110 e_icode=0011 m_icode=0011 w_icode=xxxx
clk=1 f_icode=0001 d_icode=0110 e_icode=0011 m_icode=0011 w_icode=0011
clk=1 f_icode=0001 d_icode=0001 e_icode=0110 m_icode=0011 w_icode=0011
clk=0 f_icode=0001 d_icode=0001 e_icode=0110 m_icode=0011 w_icode=0011
clk=1 f_icode=0001 d_icode=0001 e_icode=0110 m_icode=0110 w_icode=0011
clk=1 f_icode=0001 d_icode=0001 e_icode=0001 m_icode=0110 w_icode=0011
clk=0 f_icode=0001 d_icode=0001 e_icode=0001 m_icode=0110 w_icode=0011
clk=1 f_icode=0000 d_icode=0001 e_icode=0001 m_icode=0001 w_icode=0110
clk=1 f_icode=0000 d_icode=0000 e_icode=0001 m_icode=0001 w_icode=0110
clk=0 f_icode=0000 d_icode=0000 e_icode=0001 m_icode=0001 w_icode=0110
clk=1 f_icode=xxxx d_icode=0000 e_icode=0001 m_icode=0001 w_icode=0001
clk=1 f_icode=xxxx d_icode=xxxx e_icode=0000 m_icode=0001 w_icode=0001
clk=0 f_icode=xxxx d_icode=xxxx e_icode=0000 m_icode=0001 w_icode=0001
clk=1 f_icode=xxxx d_icode=xxxx e_icode=0000 m_icode=0000 w_icode=0001
clk=1 f_icode=xxxx d_icode=xxxx e_icode=xxxx m_icode=0000 w_icode=0001
clk=0 f_icode=xxxx d_icode=xxxx e_icode=xxxx m_icode=0000 w_icode=0001
clk=1 f_icode=xxxx d_icode=xxxx e_icode=xxxx m_icode=xxxx w_icode=0000
[Done] exit with code=0 in 0.255 seconds
```

→ Data forwarding is also working fine.

→ We also tried implementing stall and bubble. We were almost done but its giving us unexpected outputs.

Gtkwave for the inputs I considered:

- I considered a halt operation for testing the complete pipelining file with data forwarding, stall and bubble.
- We have observed all the correct values for all the variables which is expected.


Signals
Waves
Time
E_RA[3:0]
E_RB[3:0]
E_dstE[3:0]
E_dstM[3:0]
E_icode[3:0]
E_ifun[3:0]
E_pc[63:0]
E_status[2:0]
E_valA[63:0]
E_valB[63:0]
E_valC[63:0]
F_predPC[63:0]
M_cnd
M_dstE[3:0]
M_icode[3:0]
M_icode[3:0]
M_pc[63:0]
M_status[2:0]
M_valA[63:0]
M_valA[63:0]
W_cnd
W_dstE[3:0]
W_dstM[3:0]
W_status[2:0]
W_icode[3:0]
W_valE[63:0]
W_valM[63:0]
clk