# 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.

### 1. Fetch

- 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

- 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,reg_
mem2,reg_mem3,reg_mem4,reg_mem5,reg_mem6,reg_mem7,reg_mem8,reg_mem9,reg_me
m10,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

- 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.

- 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

- 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.

- 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.

- 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.

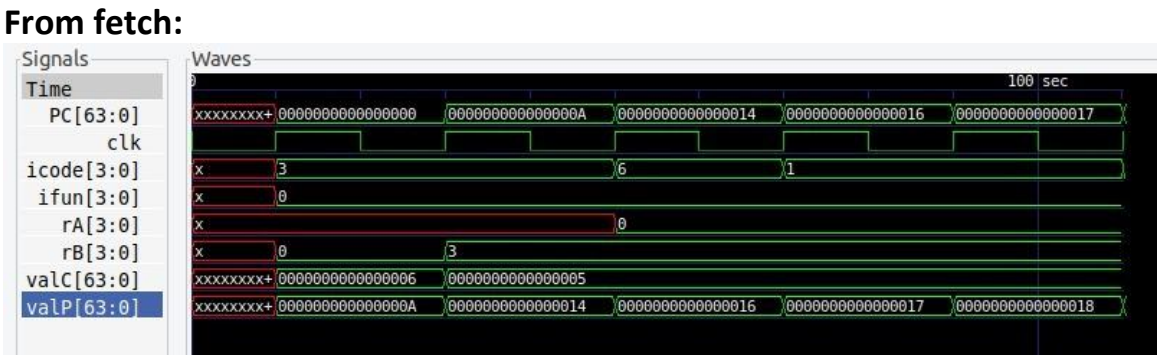
# Result:

```
[Running] Seq_final.v
clk=0 hlt=x PC=                 x icode=xxxx ifun=xxxx rA=xxxx rB=xxxx,valC=                x,valA=                x,valB=                x
clk=1 hlt=x PC=                 0 icode=0011 ifun=0000 rA=xxxx rB=0000,valC=                6,valA=                x,valB=                x
clk=0 hlt=x PC=                 0 icode=0011 ifun=0000 rA=xxxx rB=0000,valC=                6,valA=                x,valB=                x
clk=1 hlt=x PC=                10 icode=0011 ifun=0000 rA=xxxx rB=0011,valC=                5,valA=                x,valB=                x
clk=0 hlt=x PC=                10 icode=0011 ifun=0000 rA=xxxx rB=0011,valC=                5,valA=                x,valB=                x
clk=1 hlt=x PC=                20 icode=0110 ifun=0000 rA=0000 rB=0011,valC=                5,valA=                6,valB=                5
clk=0 hlt=x PC=                20 icode=0110 ifun=0000 rA=0000 rB=0011,valC=                5,valA=                6,valB=               11
clk=1 hlt=x PC=                22 icode=0001 ifun=0000 rA=0000 rB=0011,valC=                5,valA=                6,valB=               11
clk=0 hlt=x PC=                22 icode=0001 ifun=0000 rA=0000 rB=0011,valC=                5,valA=                6,valB=               11
clk=1 hlt=x PC=                23 icode=0001 ifun=0000 rA=0000 rB=0011,valC=                5,valA=                6,valB=               11
clk=0 hlt=x PC=                23 icode=0001 ifun=0000 rA=0000 rB=0011,valC=                5,valA=                6,valB=               11
clk=1 hlt=1 PC=                24 icode=0000 ifun=0000 rA=0000 rB=0011,valC=                5,valA=                6,valB=               11
valE=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,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
```

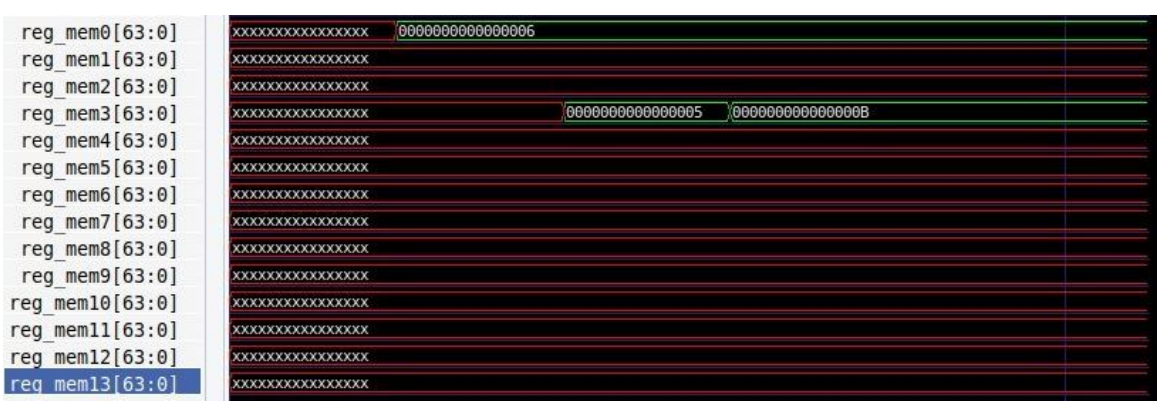
- The above pic contains clk, icode,ifun,rA,rB,valA and valB.
- As we have given value 6 to rA and 5 to rB.So, we see that valA has 6 and valB is 5.
- From here we say fetch and decode are working fine.
- Then we also see the valB output is updated to 11 as we done the sum operation. So, execute is also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