Intro to Processor Architecture

Course Project

Abhinav S - 2022102037
Chirag Goyal - 2022102041

March 7, 2024

Overview

The main objective of the project is to code a processor in verilog capable of executing all Y86-64 ISA instriuctions. First, a sequential architecture has been considered. The final goal is to design a 5-stage pipelined implementation.

Sequential Processor

We describe a processor called SEQ (for "sequential" processor). On eachclock cycle, SEQ performs all the steps required to process a complete instruction.This would require a very long cycle time, however, and so the clock rate would be unacceptably low. Our purpose in developing SEQ is to provide a first step toward our ultimate goal of implementing an efficient pipelined processor.

Organizing Processing into Stages

1. Fetch
2. Decode
3. Execute
4. Memory
5. Write-back
6. PC update

Basic Overview of Each Stage

1. Fetch: The fetch stage reads the bytes of an instruction from memory, using the program counter (PC) as the memory address. From the instruction it extracts the two 4-bit portions of the instruction specifier byte, referred to as icode (the instruction code) and ifun (the instruction function). It possibly fetches a register specifier byte, giving one or both of the register operand specifiers rA and rB. It also possibly fetches an 8-byte constant word $valC$. It computes $valP$ to be the address of the instruction following the current one in sequential order. That is, $valP$ equals the value of the PC plus the length of the fetched instruction.

2. Decode: The decode stage reads up to two operands from the register file, giving values $valA$ and/or $valB$. Typically, it reads the registers designated by instruction fields rA and rB, but for some instructions, it reads register %rsp.

3. Execute: In the execute stage, the arithmetic/logic unit (ALU) performs the operation specified by the instruction (according to the value of ifun), computes the effective address of a memory reference, or increments or decrements the stack pointer. We refer to the resulting value as $valE$. The condition codes are possibly set. For a conditional move instruction, the stage will evaluate the condition codes and move condition (given

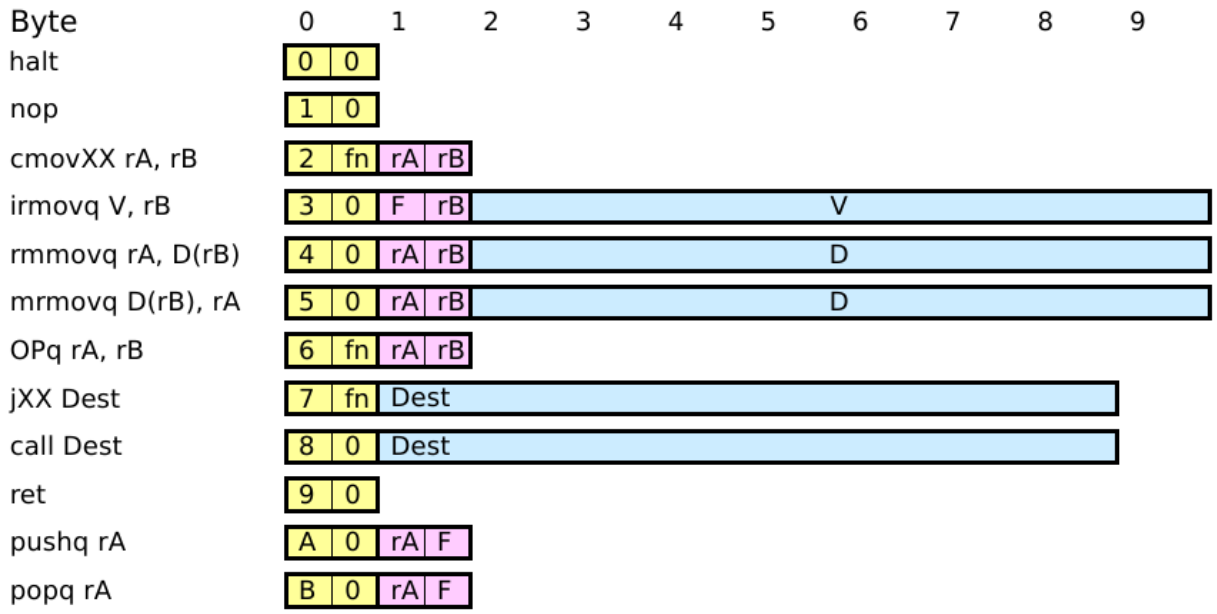
by ifun) and enable the updating of the destination register only if the condition holds by updating rB to 15 if conditions are not met. Similarly, for a jump instruction, it determines whether or not the branch should be taken.

4. Memory: The memory stage may write data to memory, or it may read data from memory. We refer to the value read as $valM$.

5. Write-back: The write-back stage writes up to two results to the register file.

6. PC Update: The PC is set to the address of the next instruction.

Opcodes and function codes for all the instruction we need to implement



1)Fetch: In the fetch stage we are required to read instruction by instruction from the instruction memory and find the values of icode , ifun , rA , rB and valC according to the instruction.

1. The fetch unit of the processor fetches the instruction from the memory location indicated by the program counter (PC).

2. After fetching the instruction, the program counter is typically incremented to point to the next instruction in memory. This prepares the processor to fetch the net instruction during the next cycle.

3. While not strictly part of the fetch stage, often the fetched instruction undergoes initial decoding to determine its type and any additional information needed for execution. This information may include opcode, operand addresses, and other control signals.

4. It also computes $valP$ which is the address where next instruction will be present. In case of jump or call, the next instruction might be in a different address.

Fetch code for different instructions:

Instruction	**Code**
OPq rA,rB	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC+1]$ $valP \leftarrow PC+2$
rrmovq rA,rB	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC+1]$ $valP \leftarrow PC+2$
irmovq V,rB	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
rmmovq rA,D(rB)	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
mrmovq D(rB),rA	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
pushq rA	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC+1]$ $valP \leftarrow PC+2$

popq rA	$icode : ifun \leftarrow M_1[PC]$ $rA : rB \leftarrow M_1[PC+1]$ $valP \leftarrow PC+2$
jXX Dest	$icode : ifun \leftarrow M_1[PC]$ $valC \leftarrow M_8[PC+1]$ $valP \leftarrow PC+9$
call Dest	$icode : ifun \leftarrow M_1[PC]$ $valC \leftarrow M_8[PC+1]$ $valP \leftarrow PC+9$
ret	$icode : ifun \leftarrow M_1[PC]$ $valP \leftarrow PC+1$

Fetch Code

```
module fetch(input clk,input [63:0] pc,output reg
    [63:0] valP,output reg [7:0] opcode,output reg
    [7:0] rArB,output reg [63:0] valC,output reg [1:0]
    status);
     reg addrerror;
     reg instrerror;
     reg halterror;
     wire addrerror1,addrerror2,addrerror3,addrerror4;
     reg [7:0] instructionmemory [1023:0];
      initial begin
        $readmemb("Demo.txt", instructionmemory);
      end
      reg [63:0] val_2;
      reg [63:0] val_1;

      checkpc inst1_pc(pc,addrerror1);
      checkpc inst2_pc(pc+1,addrerror2);
      checkpc inst3_pc(pc+9,addrerror3);
      checkpc inst4_pc(pc+8,addrerror4);

     integer i;

```

```
20
21      always @(posedge clk) begin
22       if(addrerror1==0) begin
23          opcode=instructionmemory[pc];
24       end
25       else begin
26          opcode=8'b00000000;
27       end
28
29       if(addrerror2==0) begin
30          rArB=instructionmemory[pc+1];
31       end
32       else begin
33          rArB=0;
34       end
35
36       if(addrerror3==0) begin
37          val_2[63:56]=instructionmemory[pc+9];
               val_2[55:48]=instructionmemory[pc+8];
               val_2[47:40]=instructionmemory[pc+7];
               val_2[39:32]=instructionmemory[pc+6];
38          val_2[31:24]=instructionmemory[pc+5];
               val_2[23:16]=instructionmemory[pc+4];
               val_2[15:8]=instructionmemory[pc+3];
               val_2[7:0]=instructionmemory[pc+2];
39       end
40       else begin
41          val_2=0;
42       end
43
44        if(addrerror4==0) begin
45          val_1[63:56]=instructionmemory[pc+8];
               val_1[55:48]=instructionmemory[pc+7];
               val_1[47:40]=instructionmemory[pc+6];
               val_1[39:32]=instructionmemory[pc+5];
46          val_1[31:24]=instructionmemory[pc+4];
               val_1[23:16]=instructionmemory[pc+3];
               val_1[15:8]=instructionmemory[pc+2];
               val_1[7:0]=instructionmemory[pc+1];
47       end
48       else begin
49          val_1=0;
50       end
51
52      if(opcode==8'b00010000||opcode==8'b00000000
53          ||opcode==8'b10010000) begin
54          instrerror=0;
55          addrerror=addrerror1;
56          valP=pc+1;
57      end
```

```
58      else if(opcode[7:4]==4'b0010||opcode[7:4]==4'b0110
59          ||opcode[7:4]==4'b1010||opcode[7:4]==4'b1011)
                begin
60          if((opcode[7:4]==4'b0010&&opcode[3:0]>6)
61          ||(opcode[7:4]==4'b0110&&opcode[3:0]>3)
62          ||(opcode[7:4]==4'b1011&&rArB[3:0]!=4'b1111)
63          ||(opcode[7:4]==4'b1010&&rArB[3:0]!=4'b1111))
                begin
64              instrerror=1;
65          end
66          else begin
67              instrerror=0;
68          end
69          addrerror=addrerror1|addrerror2;
70          valP=pc+2;
71      end
72      else
            if(opcode[7:4]==4'b0111||opcode[7:4]==4'b1000)
            begin
73          if(opcode[7:4]==4'b0111&&opcode[3:0]>6) begin
74              instrerror=1;
75          end
76          else begin
77              instrerror=0;
78          end
79          valC=val_1;
80          addrerror=addrerror1|addrerror4;
81          valP=pc+9;
82      end
83      else if(opcode[7:4]==4'b0011||opcode[7:4]==4'b0100
84          ||opcode[7:4]==4'b0101) begin
85          valC=val_2;
86          addrerror=addrerror1|addrerror2|addrerror3;
87          if(opcode[7:4]==4'b0011&&rArB[7:4]!=4'b1111)
                begin
88          instrerror=1;
89          end
90          else begin
91              instrerror=0;
92          end
93          valP=pc+10;
94      end
95      else begin
96          addrerror=addrerror1;
97          instrerror=1;
98          valP=pc+1;
99      end
100
101     if(opcode==8'b00000000) begin
102      halterror=1;
```

```
103     end
104     else begin
105      halterror=0;
106     end
107
108      if(addrerror==1)begin
109        status=2;//Invalid address
110      end
111      else if(halterror==1)begin
112          status=1;//1-halt
113      end
114      else if(instrerror==1)begin
115          status=3;//3-Invalid instruction
116      end
117      else begin
118          status=0;//0-correct instruction
119      end
120      end
121 endmodule
```

Listing 1: Fetch Block

Code Explanation:

1. The module declaration specifies the inputs and outputs of the module:
   - **input** clk: Clock signal input.
   - **input** [63:0] pc: Program Counter input of 64 bits.
   - **output reg** [63:0] valP: Value of the Program Counter output of 64 bits.
   - **output reg** [7:0] opcode: Opcode output of 8 bits.
   - **output reg** [7:0] rArB: Register A and B output of 8 bits.
   - **output reg** [63:0] valC: Value C output of 64 bits.
   - **output reg** [1:0] status: Status output of 2 bits.
2. Several internal variables are used in the module:-
   - **reg** addrerror: Register indicating whether there is an address error.
   - **reg** instrerror: Register indicating whether there is an instruction error.
   - **reg** halterror: Register indicating whether there is a halt error.
   - **wire** addrerror1, addrerror2, addrerror3, addrerror4: Wires used for address error checking.
   - **wire** [63:0] val_1, **wire** [63:0] val_2:These are variables that store certain bytes of instruction memory which may be part of the instruction being fetched.
   - **reg** [7:0] instructionmemory[1023:0]: Array representing instruction memory.

3. In the fetch stage, instructions are read from a text file "Demo.txt" which contains instructions in binary form. .
4. The code uses the checkpc function to determine if there are any address errors (addrerror1, addrerror2, addrerror3, addrerror4) for specific program counter (pc) values.
5. The fetch is inside a Synchronous always block and is executed at every positive edge of the clock cycle.
6. Inside the **always** block, the code updates the values of opcode, rArB, val_2, val_1, instrerror , addrerror, and valP based on the conditions specified.
7. The conditions in the `if` statements check the opcode value and update the variables `valC,valP` accordingly. There are also conditions that handle different instruction types and error checking cases.
8. The code also sets the `halterror` value based on the opcode.
9. Based on the values of `addrerror`, `halterror`, and `instrerror`, the code sets the value of `status` to indicate the current status of the processor.

2)Decode:

Decode code for different instructions:

Instruction	**Code**
OPq rA,rB	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
rrmovq rA,rB	$valA \leftarrow R[rA]$
rmmovq rA,D(rB)	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$
mrmovq D(rB),rA	$valB \leftarrow R[rB]$
pushq rA	$valA \leftarrow R[rA]$ $valB \leftarrow R[\%esp]$
popq rA	$valA \leftarrow R[\%esp]$ $valB \leftarrow R[\%esp]$

call Dest	$valB \leftarrow R[\%esp]$
ret	$valA \leftarrow R[\%esp]$ $valB \leftarrow R[\%esp]$

Decode Code

```
module decode (
  input [7:0] opcode,
  input [7:0] rArB,
  input [63:0] valC,
  output reg error,
  output reg [3:0] registernumber1,
  output reg [3:0] registernumber2
);  //based on opcoded,sets register number to be
    read which goes to register which reads and
    gives the output
  reg error1,error2;
  always @(*) begin
    if(rArB[7:4]==4'b1111) begin
          error1=1;
    end //checks if rA,rB are valid registers or
       not
    else begin
      error1=0;
    end
    if(rArB[3:0]==4'b1111) begin
      error2=1;
    end
    else begin
      error2=0;
    end
    if ((opcode[7:4] == 4'b0100) || (opcode[7:4]
       == 4'b0101) || (opcode[7:4] == 4'b0110))
       begin
      registernumber1 = rArB[7:4];
      registernumber2 = rArB[3:0];
      error = error1|error2;
    end
    else if (opcode[7:4] == 4'b0010) begin
      registernumber1 = rArB[7:4];
      registernumber2 = 4'b0000;
      error = error1|error2;
    end
    else if (opcode[7:4] == 4'b0011) begin
```

```
34        error=error2;
35        registernumber1 = rArB[3:0];
36        registernumber2 = 4'b0000;
37      end
38      else if ((opcode[7:4] == 4'b1001) ||
            (opcode[7:4] ==
            4'b1000)||(opcode[7:4]==4'b1011)) begin
39        registernumber1 = 4'b0100;
40        registernumber2 = 4'b0100;
41        error = 0;
42      end
43      else if(opcode[7:4]==4'b1010) begin
44        registernumber1=rArB[7:4];
45        registernumber2=4'b0100;
46        error=error1;
47      end
48      else begin
49        error = 0;
50        registernumber1 = 4'b0000;
51        registernumber2 = 4'b0000;
52      end
53    end
54
55  endmodule
```

Listing 2: Decode Block

Code Explanation:

- The decode module is responsible for decoding the instruction opcode and the rArB value to determine the register numbers to be read. Let's break down the code and explain its functionality:
  - **input** [7:0] opcode: Input opcode of 8 bits.
  - **input** [7:0] rArB: Input rArB value of 8 bits.
  - **input** [63:0] valC: Input value C of 64 bits.
  - **output reg** error: Output register indicating whether there is an error.
  - **output reg** [3:0] registernumber1: Output register indicating the first register number to be read.
  - **output reg** [3:0] registernumber2: Output register indicating the second register number to be read.
- The code declares registers 'error1' and 'error2' to keep track of errors in rA and rB.
- The decode block is an **always** @(*) block and change its output whenever any input to it changes.
- Inside the **always** @(*) block, the code checks if 'rArB[7:4]' and 'rArB[3:0]' are valid registers by comparing with '4'b1111'. If a register number is equal to '4'b1111', it means it is an invalid register. Based on the opcode and the values of 'rArB', the code determines

the register numbers to be read and assigns them to ‘registernumber1‘ and ‘registernumber2‘ respectively.

- The code also checks various conditions to set the ‘error‘ flag. If ‘registernumber1‘ is read and error1 is 1,then error is set.Similarly for the other registernumber.
- The code uses if-else statements to handle different opcode cases and assign appropriate values to the output registers.
- If none of the opcode conditions are met, indicating an unknown or unsupported opcode, the code assigns default values (‘0‘) to ‘error‘, ‘registernumber1‘, and ‘registernumber2‘.

3)Execute:

Execute code for different instructions:

Instruction	**Code**
OPq rA,rB	$valE \leftarrow valB\ OP\ valA$ $setCC$
rrmovq rA,rB	$valE \leftarrow 0 + valA$
irmovq V,rB	$valE \leftarrow 0 + valC$
rmmovq rA,D(rB)	$valE \leftarrow valB + valC$
mrmovq D(rB),rA	$valE \leftarrow valB + valC$
pushq rA	$valE \leftarrow valB + (-8)$
popq rA	$valE \leftarrow valB + 8$
jXX Dest	$Cnd \leftarrow Cond(CC, ifun)$
call Dest	$valE \leftarrow valB + (-8)$
ret	$valE \leftarrow valB + 8$

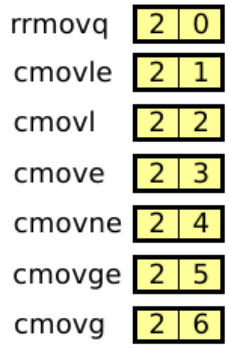


Figure 1: Function Codes for cmovq

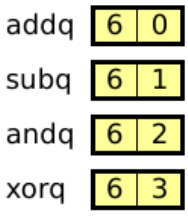


Figure 2: Function Codes for OPq

Figure 3: Function Codes for jXX

Execute Code

```
module execute(input [7:0] opcode,input [7:0]
    rArB,input [63:0] valA,input [63:0] valB,input
    [63:0] valC,input [2:0] cc,output [63:0]
    valE,output reg [7:0] rArB_execute,output reg
    Cnd,output reg [2:0] cc_out);
reg [1:0] control;
wire overflow;
reg Cnd_temp;
reg [63:0] input1;
reg [63:0] input2;
reg op;
ALU instance_execute
(input1,input2,control,valE,overflow); //ALU
    driven by input which varies with respecto to
    the opcode
always @(*) begin
if(opcode[7:4]==4'b0110) begin//6    // changing
    input1,input2 based on icode
     control=opcode[1:0];
     input1=valB;
     input2=valA;
     op=1;
end
else if(opcode[7:4]==4'b0100
||opcode[7:4]==4'b0101)begin//4,5
     control=2'b00;
     input1=valB;
     input2=valC;
     op=0;
end
else if(opcode[7:4]==4'b1011
||opcode[7:4]==4'b1001)begin//9,11
     control=2'b00;
     input1=valB;
     input2=64'd8;
     op=0;
end
else if(opcode[7:4]==4'b0010) begin//2
      control=2'b00;
      input1=valA;
      input2=64'd0;
      op=0;
end
else if(opcode[7:4]==4'b1000
||opcode[7:4]==4'b1010) begin//8,10
     control=2'b01;
     input1=valB;
```

```
41      input2=64'd8;
42      op=0;
43
44  end
45  else if(opcode[7:4]==4'b0011) begin//3
46      control=2'b00;
47       input1=valC;
48      input2=64'd0;
49      op=0;
50  end
51  else begin
52    op=0;
53  end
54
55  if(op==1) begin//updating cc if opq has occured
56    cc_out[2]=overflow;//setting overflow bit after
          execute
57    if(valE==0) begin//setting zero cc bit after
          execute
58      cc_out[0]=1;
59    end
60    else begin
61      cc_out[0]=0;
62    end
63
64    if(valE[63]==1'b1) begin//setting output sign
          bit after execute
65      cc_out[1]=1;
66    end
67    else begin
68      cc_out[1]=0;
69    end
70    end
71
72  else begin//setting output cc to input if no op
      in execute
73    cc_out=cc;
74  end
75
76  if(opcode[3:0]==4'b0000) begin
77    Cnd_temp=1;
78  end
79  else if(opcode[3:0]==4'b0001) begin//le
80      Cnd_temp=(cc_out[1]^cc_out[2])|cc_out[0];
81  //setting Cnd_temp for jump and cmov cases
      otherwise not used at all
82    end
83
84    else if(opcode[3:0]==4'b0010) begin//l
85      Cnd_temp=cc_out[1]^cc_out[2];
```

```
86    end
87    else if(opcode[3:0]==4'b0011)begin//e
88      Cnd_temp=cc_out[0];
89    end
90    else if(opcode[3:0]==4'b0100)begin//ne
91        Cnd_temp=~cc_out[0];
92    end
93    else if(opcode[3:0]==4'b0101)begin//ge
94        Cnd_temp=~(cc_out[1]^cc_out[2]);
95    end
96    else if(opcode[3:0]==4'b0110)begin//g
97    Cnd_temp= ~(cc_out[1]^cc_out[2])&~(cc_out[0]);
98  end
99  else begin
100   Cnd_temp=0;
101 end
102 if(opcode[7:4]==4'b0010) begin//changing rB to 15
      if condition for cmov is not satisfied
103    if(Cnd_temp==1) begin
104      rArB_execute=rArB;
105    end
106    else begin
107      rArB_execute[7:4]=rArB[7:4];
108       rArB_execute[3:0]=4'b1111;
109    end
110 end
111 else begin
112   rArB_execute=rArB;
113 end
114
115 if(opcode [7:4]==4'b0111) begin//setting Cnd for
      jump case.If Cnd is 1,jump to Dest,other wise
      no jump
116   Cnd=Cnd_temp;
117 end
118 else begin
119   Cnd=0;
120 end
121 end
122 endmodule
```

Listing 3: Execute Block

Code Explanation:

- The "execute" module takes several inputs, including opcode, rArB, valA, valB, valC, and cc. It also provides outputs such as valE, rArB_execute, Cnd, and cc_out. These inputs and outputs are used to perform various operations within the module.
- The module declares internal registers and wires, including **reg** [1:0] control,

wire overflow, and **reg** Cnd_temp, which are used in different stages of execute.

- An Arithmetic Logic Unit (ALU) named "instance_execute" is instantiated within the module. It takes input1, input2, control, valE, and an overflow wire as inputs to perform arithmetic and logical operations.
- Depending on the opcode, the code sets the values of control , input1, input2, and op. These values control the behavior of the ALU and determine the inputs for arithmetic or logical operations.
- If the op is set to 1 indicating *OpQ*,flags such as overflow, zero, and sign are updated accordingly in cc_out.
- The code updates the condition code (cc_out) based on the result of the ALU operation. Overflow, zero, and sign bits are set based on the values of overflow and valE.
- The lower 4 bits of the opcode are evaluated to determine the value of Cnd_temp, representing the condition evaluation result based on the condition code (cc_out). Various conditions such as less than, greater than, and equal to are checked, and Cnd_temp is updated accordingly.
- For certain opcodes (e.g., cmovq), the code checks the condition (Cnd_temp) and updates the destination register (rArB_execute) accordingly. If the condition is not met, the destination register is set to 15 (invalid register).
- For jump instructions, the code sets the Cnd flag based on the evaluated condition (Cnd_temp). If the condition is met, the code will perform the jump; otherwise, no jump will occur.

4)Memory:

Memory code for different instructions:

Instruction	**Code**
rmmovq rA,D(rB)	$M_8[valE] \leftarrow valA$
mrmovq D(rB),rA	$valM \leftarrow M_8[valE$
pushq rA	$M_8[valE] \leftarrow valA$
popq rA	$valM \leftarrow M_8[valA$

call Dest	$M_8[valE] \leftarrow valP$
ret	$valM \leftarrow M_8[valA$

Memory Code

```
module memorywrite (input clk,input [7:0]
    opcode,input [7:0] rArB,input [63:0]
    valA,input [63:0] valE,input [63:0]
    valP,output reg reset,output reg [63:0]
    addr,output reg [63:0] val_write,output reg
    wrEn,output reg reEn);

  always @(*) begin
    if ((opcode[7:4] == 4'b0100) || (opcode[7:4]
       == 4'b1010)) begin
      reset = 0; addr = valE; val_write = valA;
         wrEn = 1; reEn = 0;
    end
    else if (opcode[7:4] == 4'b0101) begin
      reset = 0; addr = valE; val_write = valA;
         wrEn = 0; reEn = 1;
    end
    else if (opcode[7:4] == 4'b1011) begin
      reset = 0; addr = valA; val_write = valA;
         wrEn = 0; reEn = 1;
    end
    else if (opcode[7:4] == 4'b1000) begin
      reset = 0; addr = valE; val_write = valP;
         wrEn = 1; reEn = 0;
    end
    else if (opcode[7:4] == 4'b1001) begin
      reset = 0; addr = valA; val_write = valA;
         wrEn = 0; reEn = 1;
    end
    else begin
      reset = 0; addr = 0; val_write = 0; wrEn =
         0; reEn = 0;
    end
  end
endmodule
```

Listing 4: Memory Block

Code Explanation:

- – The code checks the value of opcode[7:4] to determine the specific memory write operation to be performed.
  – Depending on the conditions of the opcode, the outputs of the module are updated as follows:
    * If opcode[7:4] is equal to 4'b0100 or 4'b1010, it indicates a specific memory write operation. In this case, the reset signal is set to 0, the addr is set to the value of valE, val_write is set to the value of valA, wrEn (write enable) is set to 1, and reEn (read enable) is set to 0.
    * Similar conditions are checked for other opcode values, and the outputs are updated accordingly.
    * If none of the conditions are met, the else block is executed, setting reset, addr, val_write, wrEn, and reEn to 0, representing a default state when no specific memory write or read operation is being performed.
- In summary, the "memorywrite" module interprets the opcode and other input values to select and perform specific memory write operations. It sets different control signals and data values based on the opcode value, allowing for versatile memory write capabilities based on the given conditions.

5)Write-back:

Write-back code for different instructions:

Instruction	**Code**
OPq rA,rB	$R[rB] \leftarrow valE$
rrmovq rA,rB	$R[rB] \leftarrow valE$
irmovq V,rB	$R[rB] \leftarrow valE$
mrmovq D(rB),rA	$R[rA] \leftarrow valM$
pushq rA	$R[\%rsp] \leftarrow valE$
popq rA	$R[\%rsp] \leftarrow valE$ $R[rA] \leftarrow valM$

call Dest	$R[\%rsp] \leftarrow valE$
ret	$R[\%rsp] \leftarrow valE$

Write-back Code

```
module registerwrite (input [7:0] opcode,input
    [7:0] rArB,input [63:0] valE,input [63:0]
    valM,output reg reset,
output reg [3:0] registernumber1,output reg [3:0]
    registernumber2,output reg [63:0]
    val_write1,output reg [63:0] val_write2,output
    reg wrEn);

  wire error1, error2;
  wire [63:0] temp1;
  wire [63:0] temp2;

  always @(*) begin
    if ((opcode[7:4] == 4'b0110) || (opcode[7:4]
        == 4'b0011) || (opcode[7:4] == 4'b0010))
        begin
      reset = 0;
      registernumber1 = rArB[3:0];
      registernumber2 = 4'b1111;
      val_write1 = valE;
      val_write2 = 64'hFFFFFFFFFFFFFFFF;
      wrEn=1;
    end
    else if (opcode[7:4] == 4'b0101) begin
      reset = 0;
      registernumber1 = rArB[7:4];
      registernumber2 = 4'b1111;
      val_write1 = valM;
      val_write2 = 64'hFFFFFFFFFFFFFFFF;
      wrEn=1;
    end
    else if (opcode[7:4] == 4'b1011) begin
      reset = 0;
      registernumber1 = 4'b0100;
      registernumber2 = rArB[7:4];
      val_write1 = valE;
      val_write2 = valM;
      wrEn=1;
    end
```

```
33      else if ((opcode[7:4] == 4'b1000) ||
            (opcode[7:4] == 4'b1001) || (opcode[7:4]
            == 4'b1010)) begin
34        reset = 0;
35        registernumber1 = 4'b0100;
36        registernumber2 = 4'b1111;
37        val_write1 = valE;
38        val_write2 = valE;
39        wrEn=1;
40      end
41      else begin
42        reset = 0;
43        registernumber1 = 4'b1111;
44        registernumber2 = 4'b1111;
45        val_write1 = valE;
46        val_write2 = valE;
47        wrEn=0;
48      end
49    end
50  endmodule
```

Listing 5: Write-Back Block

Code Explanation:

- The module declares internal wires, including error1, error2, temp1, and temp2, to facilitate error checking and temporary storage of values.
-
  - The code checks the value of opcode[7:4] to determine the specific register write operation to be performed.
  - Depending on the conditions of the opcode, the outputs of the module are updated as follows:
    - If opcode[7:4] is equal to 4'b0110, 4'b0011, or 4'b0010, it indicates specific register write operations. In this case, the reset signal is set to 0, registernumber1 is set to the lower 4 bits of rArB, registernumber2 is set to 4'b1111, val_write1 is set to valE, val_write2 is set to 64'hFFFFFFFFFFFFFFFF, and wrEn is set to 1.
    - Similar conditions are checked for other opcode values, and the outputs are updated accordingly.
    - If none of the conditions are met, the else block is executed, setting reset, registernumber1, registernumber2, val_write1, val_write2, and wrEn to their default values.

6)PC-update:

PC-update code for different instructions:

Instruction	Code
Opq,rrmovq,irmovq,rmmovq,mrmovq,pushq,popq	$PC \leftarrow valP$
jXX Dest	$PC \leftarrow Cnd?valC : valP$
call Dest	$PC \leftarrow valC$
ret	$PC \leftarrow valM$

PC-update Code

```
module PCupdate(input [7:0] opcode,input [63:0]
    valP,input [63:0] valM,input [63:0] valC,input
    Cnd,output reg [63:0] finalval_PC);
  always @(*) begin
  if(opcode [7:4]==4'b0111) begin
    if(Cnd==1) begin//if cnd is 1 jump to dest
      finalval_PC=valC;
    end
    else begin
      finalval_PC=valP;
    end
  end
  else if(opcode[7:4]==4'b1000) begin
     finalval_PC=valC;
  end
  else if(opcode[7:4]==4'b1001) begin
    finalval_PC=valM;
  end
  else begin
    finalval_PC=valP;
  end
  end
endmodule
```

Listing 6: PC-update Block

- The code checks the value of opcode[7:4] to determine the specific operation to be performed.

- If opcode[7:4] is equal to 4'b0111, it indicates a jump operation. Depending on the control condition (Cnd), the value of finalval_PC is updated:
  - If Cnd is 1, the program counter is updated to the value of valC (destination).
  - If Cnd is not 1, the program counter remains unchanged (finalval_PC = valP).
- If opcode[7:4] is equal to 4'b1000, the program counter is updated to the value of valC.
- If opcode[7:4] is equal to 4'b1001, the program counter is updated to the value of valM.
- If none of the above conditions are met, the else block is executed, setting finalval_PC to the value of valP (default behavior).

Limitations of Sequential Processor

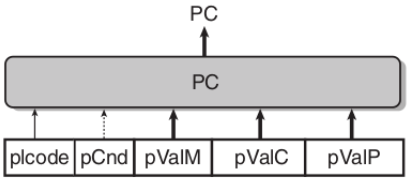
- In sequential processor, instructions are executed one after the another. So, we need time for all stages of a instruction to be executed. So, clock time period is high and clock frequency is low.
- When one particular stage of a instruction is being executed, other stages are idle. This leads to waste of hardware and power.
- So,sequential processor is inefficient and this leads to need for pipelined processor in which multiple instructions are executed at a time across different stages.

Pipelined Implementation

The first step to pipelining is the rearrangement of computation stages.

Rearranging Stages

- The PC update stage in the SEQ implementation is the last stage in the cycle of an instruction.
- For the pipelined implementation, we should bring the PC update stage to the beginning of the cycle. This change allows us to continuously fetch the next instruction without having to wait for the PC update stage of the previous instruction to end, had it been at the end of the cycle.This rearrangement is known as circuit retiming, and it changes the general presentation of the circuit without affecting its local behavior.
- Moreover, it enables us to balance the delays between stages in the pipelined system.Now, with the PC update stage at the beginning of the cycle, it can continuously provide updated PC values to the fetch stage using the required values from different stages from instructions that have passed that stage.



Inserting Pipeline Registers

- The next step to pipelining is inserting the pipeline registers. We know that in a pipelined implementation, we rearrange some of the hardware and signals in the SEQ implementation and insert pipeline registers between each stage. These registers stop the signals from one stage from flowing into the next stage and affecting the processing happening there.
- **F** the register inserted before the fetch stage holds a predicted value of the program counter, it is denoted as F.
- **D** sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.
- **E** sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.

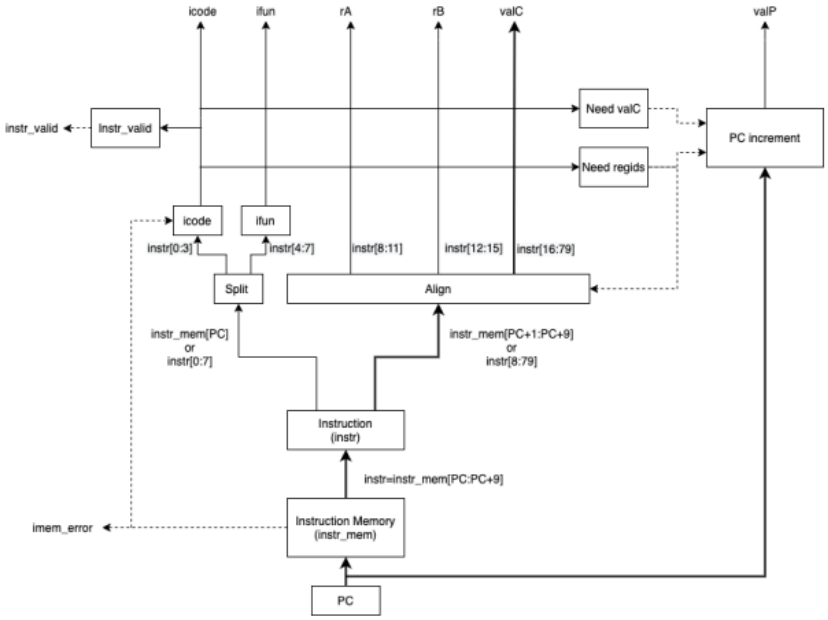
- M sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.
- W sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a ret instruction.

Rearranging and relabelling signals

- In the pipelined implementation we will have all the signals of an instruction pass through every stage one by one and these will have to be names with respect to the stage it is currently in as it is not possible to have one signal icode and have ti account for all the 5 instructions running at the same time.
- So we maintain the signal at each stage and label them with respect to the stage as $f_i code, d_i code, w_i code, etc.$

Architecture diagram

1)Fetch Stage-



Fetch-Stage Code

```
1  module fetch(input clk,input [3:0] M_icode,input
       M_Cnd,input [63:0] M_valA,input [3:0]
       W_icode,input [63:0] W_valM,input [63:0]
       F_predPC,input F_stall,
2  input D_stall,input [63:0] D_valP,input [7:0]
       D_opcode,input [7:0] D_rArB,input [63:0]
       D_valC,input [1:0] D_stat,input D_bubble,
3  output reg [63:0] f_valP,output reg
       [7:0]f_opcode,output reg [7:0] f_rArB,output
       reg [63:0] f_valC,output reg [1:0]
       f_stat,output reg [63:0] f_predPC);
4      reg addrerror;
5      reg instrerror;
6      reg halterror;
7      reg
           addrerror1,addrerror2,addrerror3,addrerror4;
8      reg [7:0] instructionmemory [4095:0];
9       initial begin
10        $readmemb("Demo.txt", instructionmemory);
11        f_pc=0;
12        stall=0;
13      end
14      reg [63:0] val_2;
15      reg [63:0] f_pc;
16      reg [63:0] val_1;
17      integer i;
18      reg stall;
19
20
21
22     always @(*) begin
23       if(stall) begin
24         f_pc=F_predPC;
25       end
26
27       if(M_icode==4'b0111&&M_Cnd==0) begin
28            f_pc=M_valA;
29       end
30       else if(W_icode==4'b1001) begin
31           f_pc=W_valM;
32       end
33       else begin
34         f_pc=F_predPC;
35       end
36
37
38       if(f_pc>4095) begin
```

```
39          addrerror1=1;
40        end
41        else begin
42          addrerror1=0;
43        end
44
45        if(f_pc+1>4095) begin
46          addrerror2=1;
47        end
48        else begin
49          addrerror2=0;
50        end
51        if(f_pc+9>4095) begin
52          addrerror3=1;
53        end
54        else begin
55          addrerror3=0;
56        end
57
58        if(f_pc+8>4095) begin
59          addrerror4=1;
60        end
61        else begin
62          addrerror4=0;
63        end
64
65      if(addrerror1==0) begin
66        f_opcode=instructionmemory[f_pc];
67      end
68      else begin
69        f_opcode=8'b00000000;
70      end
71
72      if(addrerror2==0) begin
73         f_rArB=instructionmemory[f_pc+1];
74      end
75      else begin
76         f_rArB=0;
77      end
78
79      if(addrerror3==0) begin
80         val_2[63:56]=instructionmemory[f_pc+9];
               val_2[55:48]=instructionmemory[f_pc+8];
               val_2[47:40]=instructionmemory[f_pc+7];
               val_2[39:32]=instructionmemory[f_pc+6];
81         val_2[31:24]=instructionmemory[f_pc+5];
               val_2[23:16]=instructionmemory[f_pc+4];
               val_2[15:8]=instructionmemory[f_pc+3];
               val_2[7:0]=instructionmemory[f_pc+2];
82      end
```

```
83      else begin
84         val_2=0;
85      end
86
87       if(addrerror4==0) begin
88         val_1[63:56]=instructionmemory[f_pc+8];
              val_1[55:48]=instructionmemory[f_pc+7];
              val_1[47:40]=instructionmemory[f_pc+6];
              val_1[39:32]=instructionmemory[f_pc+5];
89         val_1[31:24]=instructionmemory[f_pc+4];
              val_1[23:16]=instructionmemory[f_pc+3];
              val_1[15:8]=instructionmemory[f_pc+2];
              val_1[7:0]=instructionmemory[f_pc+1];
90      end
91      else begin
92         val_1=0;
93      end
94
95     if(f_opcode==8'b00010000||f_opcode==8'b00000000
96||f_opcode==8'b10010000) begin
97         instrerror=0;
98         addrerror=addrerror1;
99         f_valP=f_pc+1;
100    end
101    else if(f_opcode[7:4]==4'b0010
102||f_opcode[7:4]==4'b0110
103||f_opcode[7:4]==4'b1010||f_opcode[7:4]==4'b1011)
       begin
104         if((f_opcode[7:4]==4'b0010&&f_opcode[3:0]>6)
105         ||(f_opcode[7:4]==4'b0110&&f_opcode[3:0]>3)
106||(f_opcode[7:4]==4'b1011&&f_rArB[3:0]!=4'b1111)
107||(f_opcode[7:4]==4'b1010&&f_rArB[3:0]!=4'b1111))
       begin
108             instrerror=1;
109         end
110         else begin
111             instrerror=0;
112         end
113         addrerror=addrerror1|addrerror2;
114         f_valP=f_pc+2;
115    end
116    else if(f_opcode[7:4]==4'b0111
117||f_opcode[7:4]==4'b1000) begin
118         if(f_opcode[7:4]==4'b0111&&f_opcode[3:0]>6)
              begin
119             instrerror=1;
120         end
121         else begin
122             instrerror=0;
123         end
```

```
124          f_valC=val_1;
125          addrerror=addrerror1|addrerror4;
126          f_valP=f_pc+9;
127      end
128      else
            if(f_opcode[7:4]==4'b0011||f_opcode[7:4]==4'b0100
129 ||f_opcode[7:4]==4'b0101) begin
130          f_valC=val_2;
131          addrerror=addrerror1|addrerror2|addrerror3;
132          if(f_opcode[7:4]==4'b0011&&f_rArB[7:4]!=4'b1111)
                begin
133          instrerror=1;
134          end
135          else begin
136              instrerror=0;
137          end
138          f_valP=f_pc+10;
139      end
140      else begin
141          addrerror=addrerror1;
142          instrerror=1;
143          f_valP=f_pc+1;
144      end
145
146     if(F_stall==0) begin
147      stall=0;
148     if(f_opcode
            [7:4]==4'b0111||f_opcode[7:4]==4'b1000)
            begin
149       f_predPC=f_valC;
150     end
151     else begin
152       f_predPC=f_valP;
153     end
154     end
155     else begin
156      f_predPC=f_pc;
157      stall=1;
158     end
159
160
161     if(f_opcode==8'b00000000) begin
162      halterror=1;
163     end
164     else begin
165      halterror=0;
166     end
167
168      if(addrerror==1)begin
169        f_stat=2;
```

```
170     end
171     else if(halterror==1)begin
172         f_stat=1;
173     end
174     else if(instrerror==1)begin
175         f_stat=3;
176     end
177     else begin
178         f_stat=0;
179     end
180 
181     if(F_stall) begin
182       f_stat=0;
183     end
184 
185     if(D_bubble) begin
186      f_opcode=8'b00010000;
187      f_stat=0;
188     end
189 
190     if(D_stall)begin
191      f_opcode=D_opcode;
192       f_valP=D_valP;
193       f_valC=D_valC;
194       f_rArB=D_rArB;
195       f_stat=D_stat;
196     end
197     end
198 
199     //  initial begin
200     //   $monitor("%d",f_pc);
201     // end
202 endmodule
```

Listing 7: Fetch Block

Explanation of Fetch Module

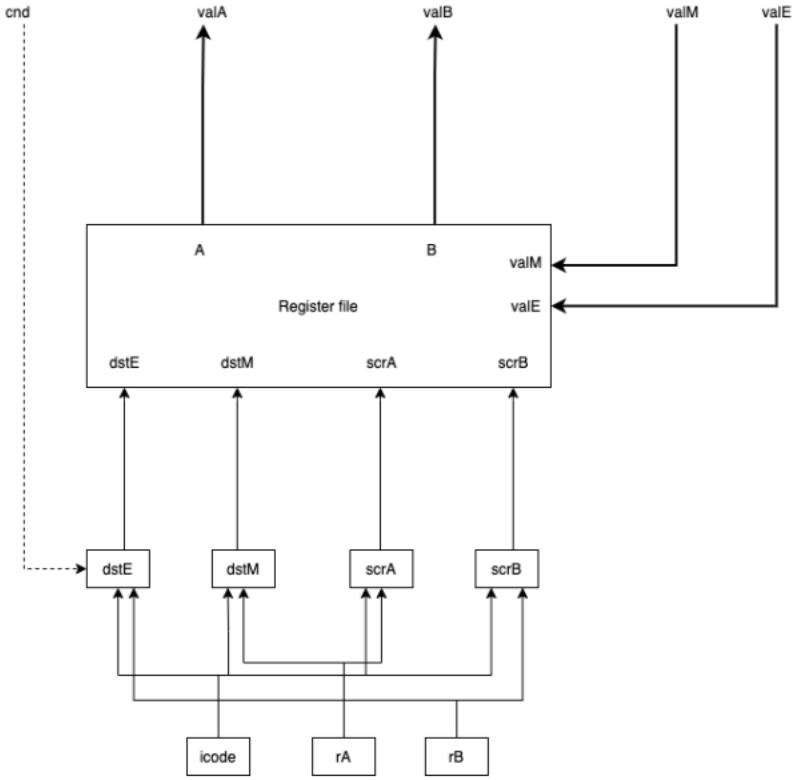
The provided Verilog code represents a module called `fetch`, which is responsible for fetching instructions from memory in a pipelined processor architecture. Below is a detailed explanation of the code:

- – Inputs:
  - `clk`: Clock signal for synchronous operation.
  - `M_icode`: Instruction code from the memory stage.
  - `M_Cnd`: Condition flag from the memory stage.
  - `M_valA`: Value A from the memory stage.
  - `W_icode`: Instruction code from the writeback stage.

    - `W_valM`: Value M from the writeback stage.
    - `F_predPC`: Predicted Program Counter (PC) from the fetch stage.
    - `F_stall`: Stall signal from the fetch stage.
    - `D_stall`: Stall signal from the decode stage.
    - `D_valP`: Value P from the decode stage.
    - `D_opcode`: Opcode from the decode stage.
    - `D_rArB`: `rArB` from the decode stage.
    - `D_valC`: Value C from the decode stage.
    - `D_stat`: Status from the decode stage.
    - `D_bubble`: Bubble signal from the decode stage.
  - Outputs:
    - `f_valP`: Value P for the next stage.
    - `f_opcode`: Opcode for the next stage.
    - `f_rArB`: `rArB` for the next stage.
    - `f_valC`: Value C for the next stage.
    - `f_stat`: Status for the next stage.
    - `f_predPC`: Predicted Program Counter (PC) for the next cycle.
- – Reads instruction memory from a file (`Demo.txt`) into the `instructionmemory` array.
  – Initializes variables `f_pc` and `stall`.
- – Updates the value of `f_pc` based on control signals and conditions.
  – Checks for address errors (`addrerror1`, `addrerror2`, `addrerror3`, `addrerror4`) based on the calculated PC value.
  – Reads instruction bytes from memory based on the PC value and updates related variables (`f_opcode`, `f_rArB`, `val_1`, `val_2`).
  – Determines the next PC value (`f_valP`) and updates control signals (`f_stat`) based on the fetched opcode and other conditions.
  – Handles stall conditions based on control signals from the previous stages.
  – Sets `halterror` flag if the fetched opcode indicates a halt instruction.
  – Updates the `f_predPC` value based on stall conditions and the fetched opcode.
- – Sets address error flags (`addrerror1`, `addrerror2`, `addrerror3`, `addrerror4`) if the calculated PC value exceeds the memory size.
  – Handles cases where instruction bytes cannot be read due to address errors.
- – Detects instruction errors (`instrerror`) based on the fetched opcode and its associated conditions.
  – Updates the `f_stat` signal accordingly.
- – Manages stall conditions based on control signals (`F_stall`) from the fetch stage and (`D_stall`) from the decode stage.

- Adjusts the next PC value (`f_predPC`) accordingly.
- – Sets the `halterror` flag if a halt instruction is encountered.
- – Sets the output signals for the next stage based on the calculated values.

2)Decode Stage-



Decode-Stage Code

```
module decode (
  input [7:0] D_opcode,
  input [7:0] D_rArB,
  input [63:0] D_valC,
  input [63:0] D_valP,
  input [1:0] D_stat,input E_bubble,
  input [3:0] e_dstE,input [63:0] e_valE,input
      [3:0] M_dstE,input [63:0] M_valE,input [3:0]
      M_dstM,input [63:0] m_valM,input [3:0]
      W_dstM,input [63:0] W_valM,
  input [3:0] W_dstE,input [63:0] W_valE,
  output reg [1:0] d_stat,
  output reg [7:0] d_opcode,
  output reg [63:0] d_valC,
  output reg [63:0] d_valA,
  output reg [63:0] d_valB,
  output reg [3:0] d_dstE,
  output reg [3:0] d_dstM,
  output reg [3:0] d_srcA,
  output reg [3:0] d_srcB
);
  reg error1,error2;
  reg error;
  reg check1,check2;
  reg [63:0] registers [15:0];
  integer i;
 initial begin
  for(i=0;i<16;i=i+1)begin
    registers[i]=0;
  end
  registers[4]=512;
 end

  always @(*) begin



    if(D_rArB[7:4]==4'b1111) begin
          error1=1;
    end
    else begin
      error1=0;
    end
    if(D_rArB[3:0]==4'b1111) begin
      error2=1;
    end
    else begin
```

```
45        error2=0;
46      end
47
48
49      if(D_opcode[7:4]==4'b0010||D_opcode[7:4]==4'b0100
50  ||D_opcode[7:4]==4'b0110||D_opcode[7:4]==4'b1010)
        begin
51        d_srcA=D_rArB[7:4];
52       check1=1;
53      end
54      else if(D_opcode[7:4]==4'b1001
55  || D_opcode[7:4]==4'b1011) begin
56        d_srcA=4'b0100;
57        check1=0;
58      end
59      else begin
60        d_srcA=4'b1111;
61        check1=0;
62      end
63
64      if(D_opcode[7:4]==4'b0100
65  ||D_opcode[7:4]==4'b0101
66  ||D_opcode[7:4]==4'b0110) begin
67        d_srcB=D_rArB[3:0];
68        check2=1;
69      end
70      else if(D_opcode[7:4]==4'b1000
71  ||D_opcode[7:4]==4'b1001
72  ||D_opcode[7:4]==4'b1010
73  ||D_opcode[7:4]==4'b1011) begin
74        d_srcB=4'b0100;
75        check2=0;
76      end
77      else begin
78        d_srcB=4'b1111;
79        check2=0;
80      end
81
82      error=(check1&error1)|(check2&error2);
83
84     if(D_opcode[7:4]==4'b0011
85  ||D_opcode[7:4]==4'b0110||D_opcode[7:4]==4'b0010)begin
86          d_dstE=D_rArB[3:0];
87      end
88      else if(D_opcode[7:4]==4'b1000
89  ||D_opcode[7:4]==4'b1001
90  ||D_opcode[7:4]==4'b1010||D_opcode[7:4]==4'b1011)
        begin
91          d_dstE=4'b0100;
92      end
```

```
93      else begin
94        d_dstE=4'b1111;
95      end
96
97      if(D_opcode[7:4]==4'b0101
98  ||D_opcode[7:4]==4'b1011) begin
99        d_dstM=D_rArB[7:4];
100     end
101     else begin
102       d_dstM=4'b1111;
103     end
104
105
106     if(error==0) begin
107       d_stat=D_stat;
108     end
109     else begin
110       d_stat=3;
111     end
112
113
114     if(D_opcode[7:4]==4'b0111||D_opcode[7:4]==4'b1000)
            begin
115       d_valA=D_valP;
116     end
117     else if(d_srcA==e_dstE) begin
118       d_valA=e_valE;
119     end
120     else if(d_srcA==M_dstM)begin
121       d_valA=m_valM;
122     end
123     else if(d_srcA==M_dstE) begin
124       d_valA=M_valE;
125     end
126     else if(d_srcA==W_dstM) begin
127       d_valA=W_valM;
128     end
129     else if(d_srcA==W_dstE) begin
130       d_valA=W_valE;
131     end
132     else begin
133       d_valA=registers[d_srcA];
134     end
135
136     if(d_srcB==e_dstE) begin
137       d_valB=e_valE;
138     end
139     else if(d_srcB==M_dstM)begin
140       d_valB=m_valM;
141     end
```

```
142     else if(d_srcB==M_dstE) begin
143       d_valB=M_valE;
144     end
145     else if(d_srcB==W_dstM) begin
146       d_valB=W_valM;
147     end
148     else if(d_srcB==W_dstE) begin
149       d_valB=W_valE;
150     end
151     else begin
152       d_valB=registers[d_srcB];
153     end
154
155     if(E_bubble) begin
156          d_opcode=8'b00010000;
157          d_dstE=4'b1111;
158          d_dstM=4'b1111;
159          d_stat=0;
160     end
161     else begin
162      d_opcode=D_opcode;
163      d_valC=D_valC;
164     end
165
166     registers[W_dstM]=W_valM;
167     registers[W_dstE]=W_valE;
168
169
170
171   end
172
173 endmodule
```

Listing 8: Decode Block

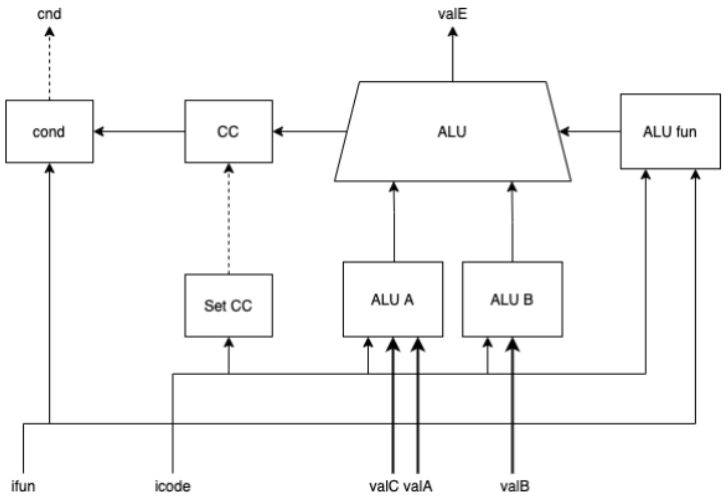
Explanation of Decode Module

The provided Verilog code represents a module called `decode`, responsible for decoding instructions in a pipelined processor architecture. Below is a detailed explanation of the code:

- – Inputs:
  - Various control signals (`D_opcode`, `D_rArB`, `D_valC`, `D_valP`, `D_stat`) from the previous pipeline stages.
  - Bubble signal (`E_bubble`) from the execute stage.
  - Results from the execute (`e_dstE`, `e_valE`), memory (`M_dstE`, `M_valE`, `M_dstM`, `m_valM`), and writeback (`W_dstM`, `W_valM`, `W_dstE`, `W_valE`) stages.

  - Outputs:
    * Decoded control signals (`d_stat`, `d_opcode`, `d_valC`, `d_valA`, `d_valB`, `d_dstE`, `d_dstM`, `d_srcA`, `d_srcB`) for the next pipeline stage.
- – Initializes an array of registers (`registers`) with zero values, except for register 4 which is set to 512 for stackpointer.
- – Determines source registers (`d_srcA` and `d_srcB`) based on the opcode (`D_opcode`) and `D_rArB`.
  - Handles errors and sets the status (`d_stat`) accordingly.
  - Computes values (`d_valA` and `d_valB`) for source operands using values from the execute, memory, and writeback stages, or from register file based on the source register indices.
  - Determines destination registers (`d_dstE` and `d_dstM`) the destination of `valE` and `valM` based on the opcode and `D_rArB` for that instruction.
  - Sets output signals based on the bubble signal (`E_bubble`) from the execute stage.
- – Updates the register file (`registers`) with values from the write-back stage.
- It also write data `W_valM` and `W_valE` into the registers using `W_dstM` and `W_dstE` as destinations respectively.These are outputs of the last pipelined register `W`.In instructions where nothing needs to be written, these destinations are set to $4'b1111$.

3)Execute Stage-



Execute-Stage Code

```
1  module execute(input [1:0] E_status,input [7:0]
       E_opcode,input [63:0] E_valA,input [63:0]
       E_valB,input [63:0] E_valC,input [3:0]
       E_dstE,input [3:0] E_dstM,
2  input [3:0] E_srcA,input [3:0] E_srcB,
3  output reg [1:0] e_status,output reg [3:0]
       e_icode,output [63:0] e_valE,output reg [63:0]
       e_valA,output reg [3:0] e_dstM,output reg
       e_Cnd,output reg [3:0] e_dstE);
4  reg [1:0] alufun;
5  wire overflow;
6  reg e_Cnd_temp;
7  reg [63:0] aluA;
8  reg [63:0] aluB;
9  reg op;
10 reg [2:0] cc;
11
12 initial begin cc=0;
13 end
14 ALU
       instance_execute(aluA,aluB,alufun,e_valE,overflow);
15 always @(*) begin
16
17   if(E_opcode[7:4]==4'b0110
18 ||E_opcode[7:4]==4'b0100
19 ||E_opcode[7:4]==4'b0101
20 ||E_opcode[7:4]==4'b1011||E_opcode[7:4]==4'b1001
21 ||E_opcode[7:4]==4'b1000||E_opcode[7:4]==4'b1010)
22   begin
23     aluA=E_valB;
24   end
25   else begin
26     aluA=0;
27   end
28
29
30   if(E_opcode[7:4]==4'b0110
31 ||E_opcode[7:4]==4'b0010) begin
32     aluB=E_valA;
33   end
34   else if(E_opcode[7:4]==4'b0011
35 ||E_opcode[7:4]==4'b0100
36 ||E_opcode[7:4]==4'b0101) begin
37     aluB=E_valC;
38   end
39   else if(E_opcode[7:4]==4'b1011
40 ||E_opcode[7:4]==4'b1001||E_opcode[7:4]==4'b1000
```

```
||E_opcode[7:4]==4'b1010)begin
    aluB=8;
 end
else begin
  aluB=0;
end

if(E_opcode[7:4]==4'b0110) begin
  alufun=E_opcode[1:0];
  op=1;
end
else if(E_opcode[7:4]==4'b1000
||E_opcode[7:4]==4'b1010) begin
  alufun=1;
  op=0;
end
else begin
  alufun=0;
  op=0;
end

  e_status=E_status;
  e_icode=E_opcode[7:4];
  e_valA=E_valA;
  e_dstM=E_dstM;

if(E_opcode[3:0]==4'b0000) begin
  e_Cnd_temp=1;
end
else if(E_opcode[3:0]==4'b0001) begin
    e_Cnd_temp=(cc[1]^cc[2])|cc[0];
  end

  else if(E_opcode[3:0]==4'b0010) begin
    e_Cnd_temp=cc[1]^cc[2];
  end
  else if(E_opcode[3:0]==4'b0011)begin
    e_Cnd_temp=cc[0];
  end
  else if(E_opcode[3:0]==4'b0100)begin
      e_Cnd_temp=~cc[0];
  end
  else if(E_opcode[3:0]==4'b0101)begin
      e_Cnd_temp=~(cc[1]^cc[2]);
  end
  else if(E_opcode[3:0]==4'b0110)begin
  e_Cnd_temp= ~(cc[1]^cc[2])&~(cc[0]);
end
else begin
  e_Cnd_temp=0;
```

```
91  end
92
93  if(E_opcode[7:4]==4'b0010) begin
94      if(e_Cnd_temp==1) begin
95        e_dstE=E_dstE;
96      end
97      else begin
98         e_dstE=4'b1111;
99      end
100 end
101 else begin
102   e_dstE=E_dstE;
103 end
104
105 if(E_opcode [7:4]==4'b0111) begin
106   e_Cnd=e_Cnd_temp;
107 end
108 else begin
109   e_Cnd=0;
110 end
111
112 if(op==1) begin
113   cc[2]=overflow;
114   if(e_valE==0) begin
115     cc[0]=1;
116   end
117   else begin
118     cc[0]=0;
119   end
120
121   if(e_valE[63]==1'b1) begin
122     cc[1]=1;
123   end
124   else begin
125     cc[1]=0;
126   end
127 end
128 end
129
130
131 endmodule
```

Listing 9: Execute Block

Explanation of Execute Module

- – Inputs:
  - ∗ Status signals (`E_status`) and opcode (`E_opcode`) from the previous pipeline stage.
  - ∗ Values of operands (`E_valA`, `E_valB`, `E_valC`) and destination registers (`E_dstE`, `E_dstM`) from the previous pipeline stage.
  - ∗ Source register indices (`E_srcA`, `E_srcB`) from the decode stage.
- – Outputs:
  - ∗ Status signals (`e_status`) and opcode (`e_icode`) for the next pipeline stage.
  - ∗ Computed values (`e_valE`) and destination register (`e_dstE`) for the writeback stage.
  - ∗ Condition flag (`e_Cnd`) for conditional branches.
- – `alufun`: Signal to control ALU operation.
- – `overflow`: Signal indicating ALU overflow.
- – `e_Cnd_temp`: Temporary condition flag value.
- – `aluA` and `aluB`: Inputs to the ALU.
- – `op`: Operation control signal.
- – `cc`: Condition code register storing flags.
- – Sets `aluA` and `aluB` based on the opcode.
- – Determines `alufun` based on the opcode for ALU operation selection.
- – Computes ALU result (`e_valE`) and sets overflow flag based on the ALU operation.
- – Computes condition flags based on the opcode and ALU result.
- – Updates the condition code register (`cc`) based on the result.
- – Determines the condition flag (`e_Cnd`) for conditional branches based on the opcode.
- – Sets destination register (`e_dstE`) based on the condition flag.
- – Updates the condition code register (`cc`) and destination registers (`e_dstE`) based on the ALU operation and opcode.

4) Memory Stage

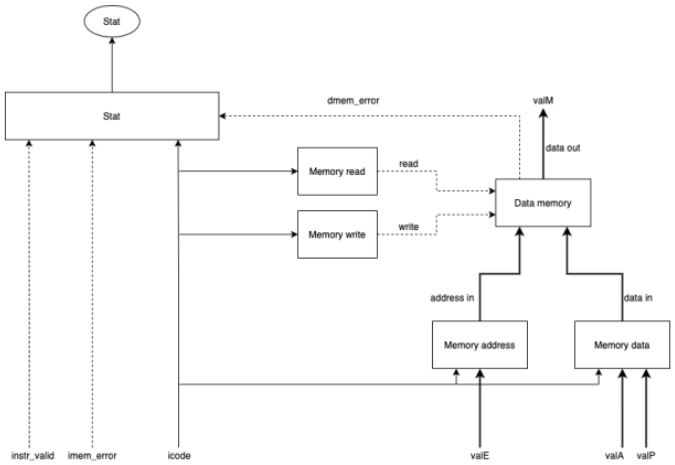


Figure 4: Memory Stage

Memory-Stage Code

```
module memorywrite (input [1:0]M_status,input
    [3:0] M_icode,input M_Cnd,input [3:0]
    M_dstE,input [3:0] M_dstM,input [63:0]
    M_valA,input [63:0] M_valE,
output reg [1:0] m_status,output reg [3:0]
    m_icode,output reg [63:0] m_valE,output reg
    [63:0] m_valM,output reg [3:0] m_dstE,output
    reg [3:0] m_dstM);
    reg [63:0] mem_addr;
    reg [7:0] datamemory [4095:0];
    reg mem_read,mem_write;
    integer i;
    reg dmem_error;
    initial begin
     for(i=0;i<4095;i=i+1) begin
       datamemory[i]=0;
     end
     datamemory[200]=20;
     datamemory[208]=21;
     datamemory[216]=22;
     datamemory[224]=23;
    end
   always @(*) begin
       if ((M_icode == 4'b0100)
```

```
|| (M_icode == 4'b1010)|(M_icode ==4'b0101)
|| (M_icode ==4'b1000)) begin
        mem_addr = M_valE;
    end
    else if((M_icode == 4'b1011)
||(M_icode == 4'b1001)) begin
      mem_addr=M_valA;
    end
    else begin
      mem_addr=0;
    end

   m_valM[63:56]=datamemory[mem_addr+7];
      m_valM[55:48]=datamemory[mem_addr+6];
      m_valM[47:40]=datamemory[mem_addr+5];
      m_valM[39:32]=datamemory[mem_addr+4];
        m_valM[31:24]=datamemory[mem_addr+3];
        m_valM[23:16]=datamemory[mem_addr+2];
        m_valM[15:8]=datamemory[mem_addr+1];
        m_valM[7:0]=datamemory[mem_addr];

    if ((M_icode == 4'b0100)
|| (M_icode == 4'b1010)|(M_icode ==4'b1000)) begin
        mem_write =1;
    end
    else begin
      mem_write=0;
    end

    if ((M_icode == 4'b0101)
|| (M_icode == 4'b1011)|(|M_icode ==4'b1001))
   begin
        mem_read =1;
    end
    else begin
      mem_read=0;
    end



    if(mem_read|mem_write) begin
      if(mem_addr>4088) begin
        dmem_error=1;
      end
      else begin
        dmem_error=0;
      end
    end
    else begin
      dmem_error=0;
```

```
68      end
69
70      if(dmem_error==0) begin
71        m_status=M_status;
72
73      if(mem_write) begin
74        datamemory[mem_addr+7]=M_valA[63:56];
75 datamemory[mem_addr+6]=M_valA[55:48]
76 ;datamemory[mem_addr+5]=M_valA[47:40];
77        datamemory[mem_addr+4]=M_valA[39:32]
78 ;datamemory[mem_addr+3]=M_valA[31:24]
79 ;datamemory[mem_addr+2]=M_valA[23:16];
80        datamemory[mem_addr+1]=M_valA[15:8]
81 ;datamemory[mem_addr]=M_valA[7:0];
82       end
83       m_icode=M_icode;
84       m_valE=M_valE;
85       m_dstE=M_dstE;
86       m_dstM=M_dstM;
87      end
88      else begin
89        m_status=2;
90      end
91    end
92
93
94
95 endmodule
```

Listing 10: Memory Block

Explanation of Memory Write Module

The provided Verilog code represents a module called `memorywrite`, responsible for writing data to memory in a pipelined processor architecture and also for reading data.

- – Inputs:
  - ∗ Status signals (`M_status`) and instruction code (`M_icode`) from the previous pipeline stage.
  - ∗ Condition (`M_Cnd`), destination registers (`M_dstE`, `M_dstM`), and values (`M_valA`, `M_valE`) from the execute stage.
- – Outputs:
  - ∗ Status signals (`m_status`) and instruction code (`m_icode`) for the next pipeline stage.
  - ∗ Values (`m_valE`, `m_valM`) and destination registers (`m_dstE`, `m_dstM`) for the next pipeline stage.

- – Determines the memory address (`mem_addr`) based on the instruction code (`M_icode`) and associated values (`M_valA`, `M_valE`).
- – Reads or writes data to the memory (`datamemory`) based on the instruction code (`M_icode`) and associated values (`M_valA`, `M_valE`).
  – Updates the memory values if a write operation is performed.
- – Checks for memory access errors, if we the address accessed is larger than datamemory.
  – Sets an error flag (`dmem_error`) if an error occurs.
- – Sets the status signals (`m_status`) based on the memory access and error conditions. as well as other values requried for next stage.

5) PipelinecontrllogicStage

Pipeline Contrl Logic Code

```
module Pipelinecontrllogic(input [3:0]
    E_icode,input [3:0] D_icode,input [3:0]
    M_icode,input [3:0] E_dstM,input [3:0]
    d_srcA,input [3:0] d_srcB,input e_Cnd,
output reg F_stall,output reg D_stall,output reg
    D_bubble,output reg E_bubble);
    always @(*) begin
    if((((E_icode==4'b0101)
||(E_icode==4'b1011))&&((E_dstM==d_srcA)
||(E_dstM==d_srcB)))||((D_icode==4'b1001)
||(E_icode==4'b1001)||(M_icode==4'b1001))) begin
        F_stall=1;
    end
    else begin
     F_stall=0;
    end

    if(((E_icode==4'b0101)
||(E_icode==4'b1011))&&((E_dstM==d_srcA)
||(E_dstM==d_srcB)))begin
        D_stall=1;
    end
    else begin
        D_stall=0;
    end

    if(((E_icode==4'b0111)&&(!e_Cnd))
||(((D_icode==4'b1001)||(E_icode==4'b1001)
||(M_icode==4'b1001))&&
    !(((E_icode==4'b0101)
```

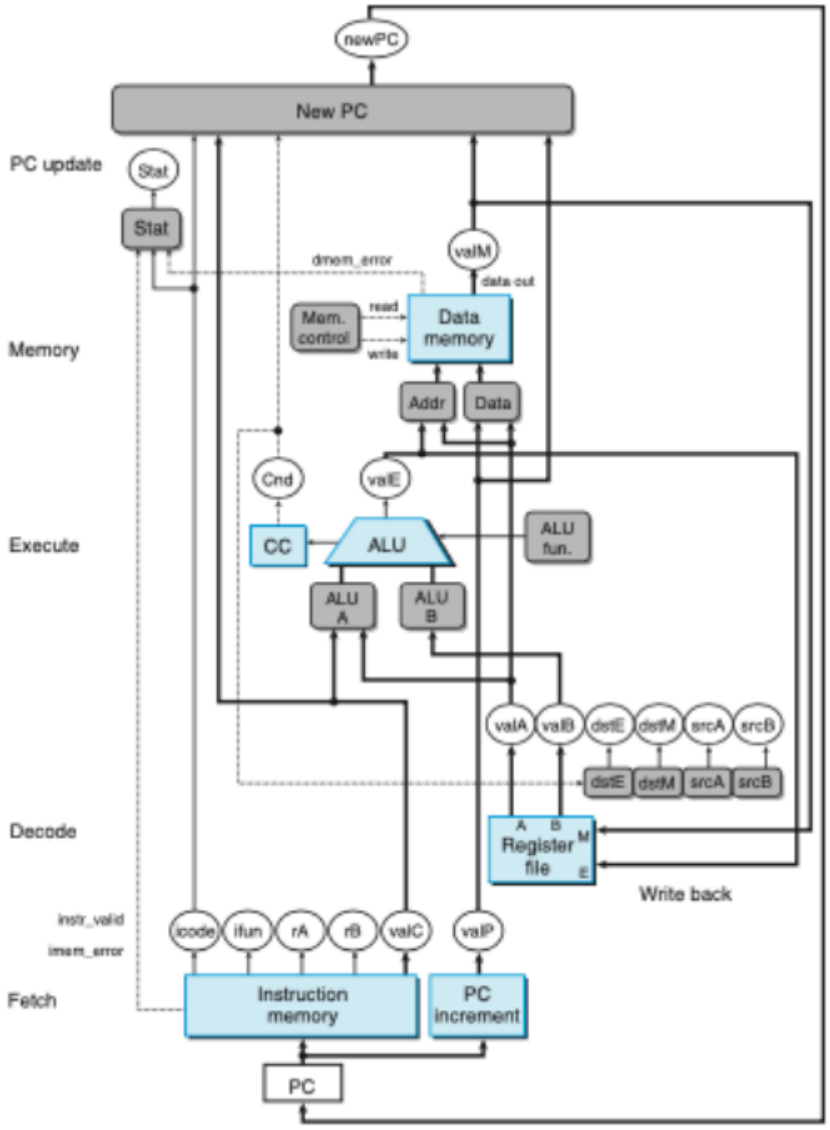
```
27 ||(E_icode==4'b1011))&&((E_dstM==d_srcA)
28 ||(E_dstM==d_srcB))))) begin
29       D_bubble=1;
30    end
31    else begin
32       D_bubble=0;
33    end
34
35 if(((E_icode==4'b0111)&&(!e_Cnd))
36 ||((E_icode==4'b0101)
37 ||(E_icode==4'b1011))&&((E_dstM==d_srcA)
38 ||(E_dstM==d_srcB))) begin
39       E_bubble=1;
40
41    end
42    else begin
43       E_bubble=0;
44    end
45    end
46 endmodule
```

Listing 11: Pipeline Control Logic

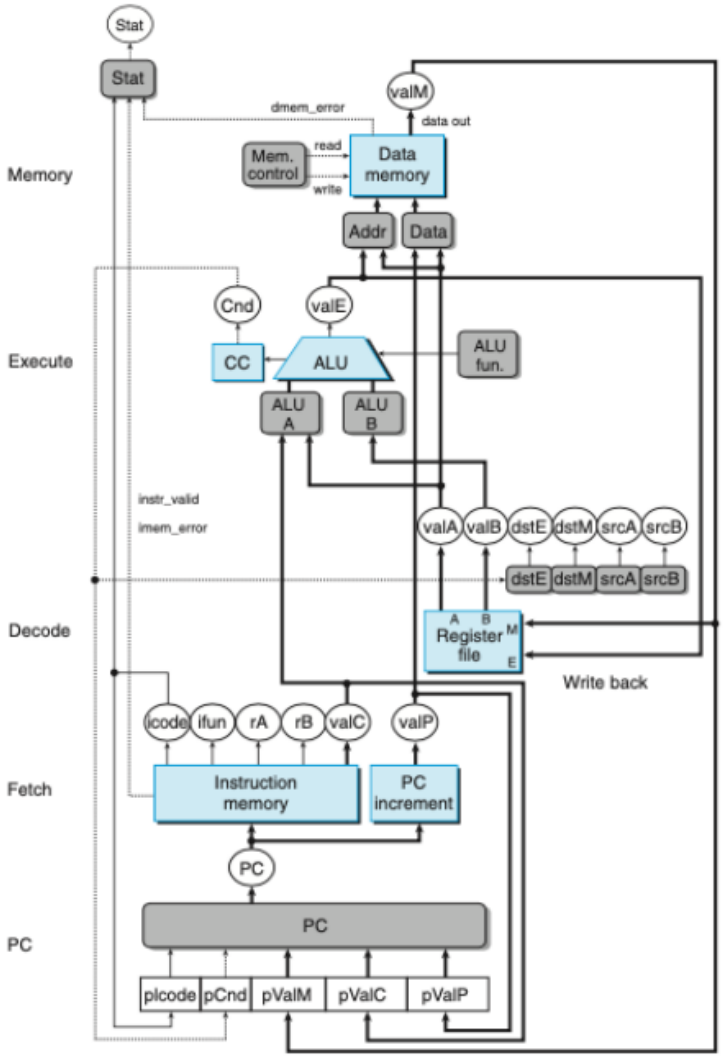
- Inputs:
  - `E_icode`: 4-bit input representing the instruction code in the Execute stage
  - `D_icode`: 4-bit input representing the instruction code in the Decode stage
  - `M_icode`: 4-bit input representing the instruction code in the Memory stage
  - `E_dstM`: 4-bit input representing the destination in the Execute stage
  - `d_srcA`: 4-bit input representing the source A in the Decode stage
  - `d_srcB`: 4-bit input representing the source B in the Decode stage
  - `e_Cnd`: Input representing the condition in the Execute stage
- Outputs:
  - `F_stall`: Output indicating whether to stall fetching an instruction
  - `D_stall`: Output indicating whether to stall decoding an instruction
  - `D_bubble`: Output indicating whether to insert a bubble in the Decode stage
  - `E_bubble`: Output indicating whether to insert a bubble in the Execute stage

  .
- It determines `F_stall`,`D_stall`,`D_bubble` and `E_bubble` which determines whether or not to stall or bubble different stages to correct pipeline hazards.

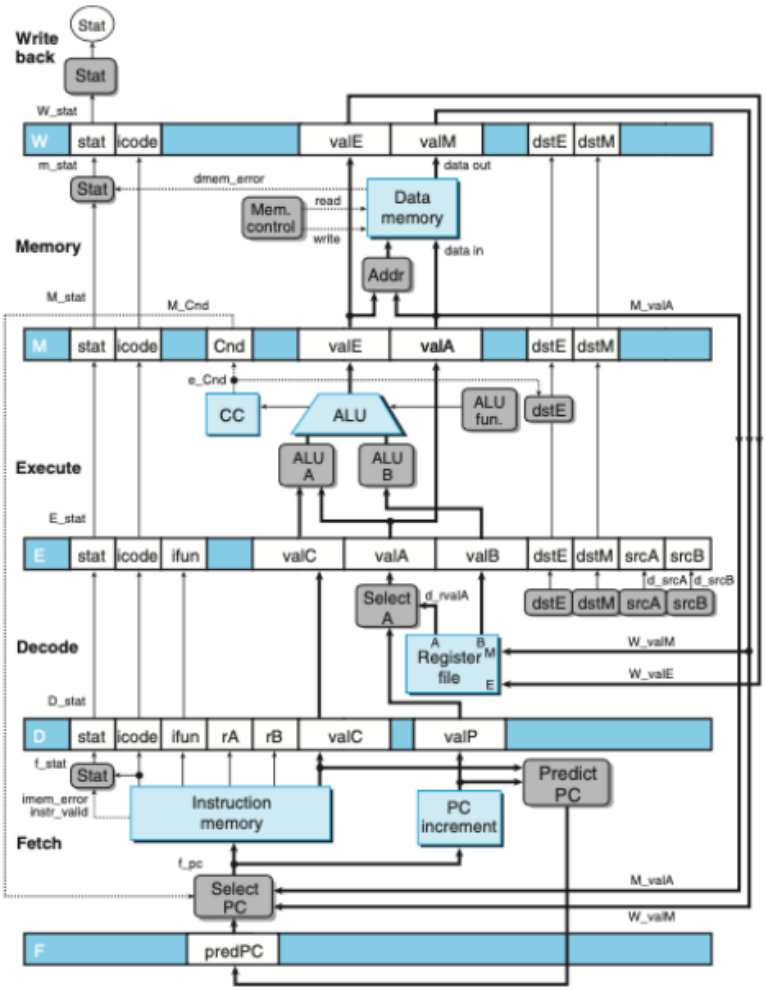
Sequential procesor SEQ hardware structure



SEQ+ hardware structure



Pipelined Processor hardware structure



Processor Features

1) Latency/Clock-frequency $= \frac{1}{\text{clock-period}} = \frac{1}{10ns} = 0.1$ GHz
2) Data-Memory Size $=4096bytes= 4KB$
3) Instruction-Memory Size $=4096bytes= 4KB$

NOTE:-We have different memory for instructions and data,i.e, we have a Harvard Architecture memory.

Testing of Pipelined Processor

Different testcases have been written which covers most of the pipeline hazards.The gtk plots of sequential and pipelined version are shown and the plot of pipelined version is described briefly.

- **1) Test Case 1:**

Listing 12: Demo1

```
irmovq $10,%rax
irmovq $11,%rcx
addq %rax,%rcx
```

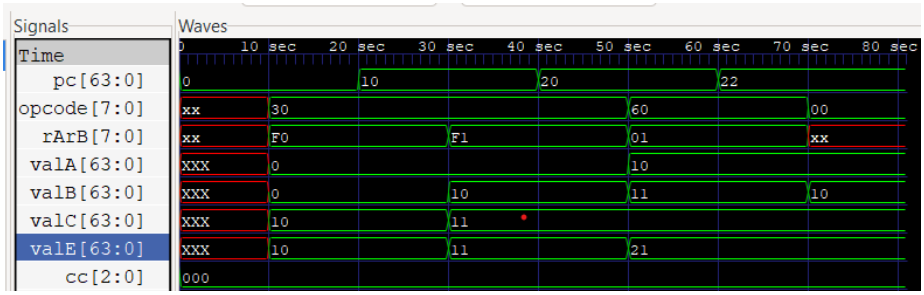


Figure 5: Sequential Output

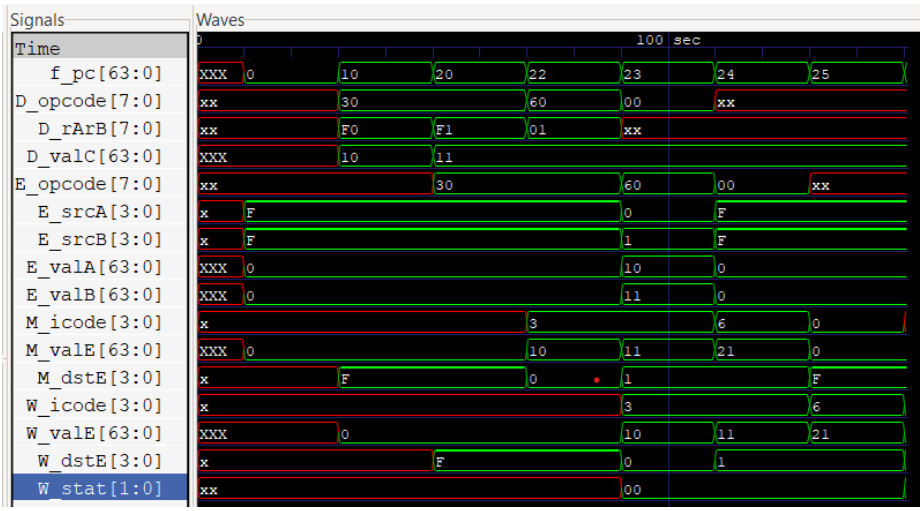


Figure 6: Pipelined Output

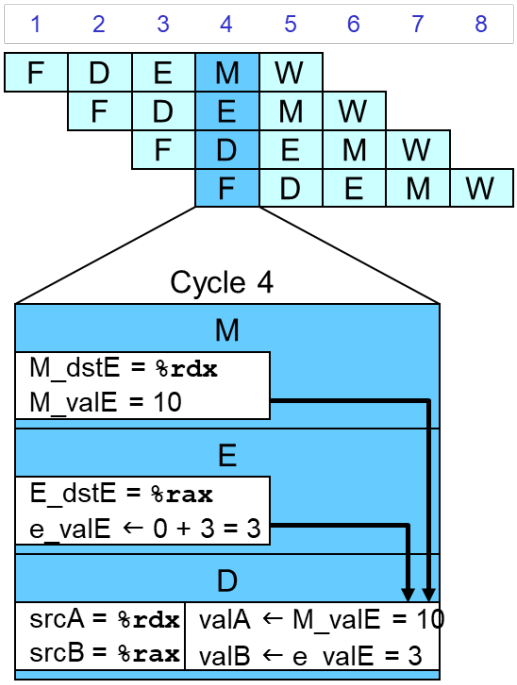
Here, when *addq* is being executed, in its decode stage, we need values from *rax* and *rcx*. These are being written into using *irmovq*. During decode stage of *addq*, the irmovq writing into *rax* is in execute stage and irmovq writing into *rcx* is in memory stage. These are forwarded

into the decode stage. We can see that the forwarded values are correct and we get an output of $10 + 11 = 21$ in M_valE. Finally, when the halt reaches write stage, the processor stops.

- Register rdx
  Generated by ALU during previous cycle
  Forward from memory as valA

- Register rax
  Value just generated by ALU
  Forward from execute as valB

```
# Demo1.txt
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



- **2.Test Case 2:-**

Listing 13: Demo2

```
1 irmovq $1 %rax
2 irmovq $2 %rax
3 irmovq $3 %rax
4 rmmovq %rax %rdx
5 halt
```

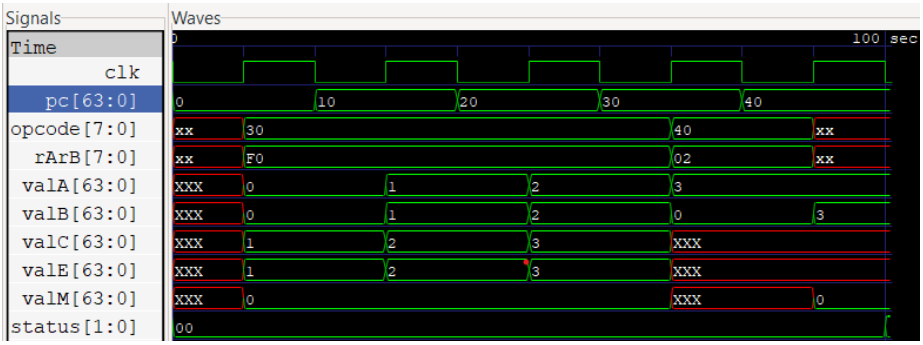


Figure 7: Sequential Output

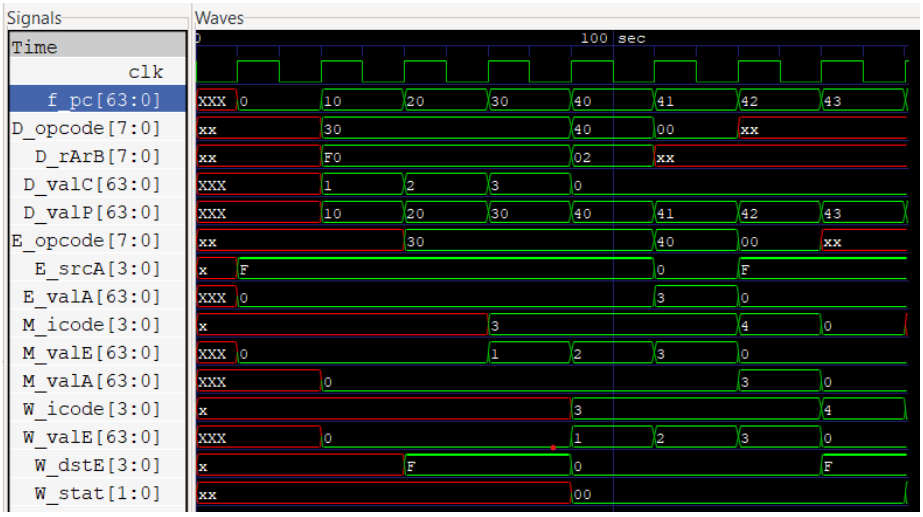
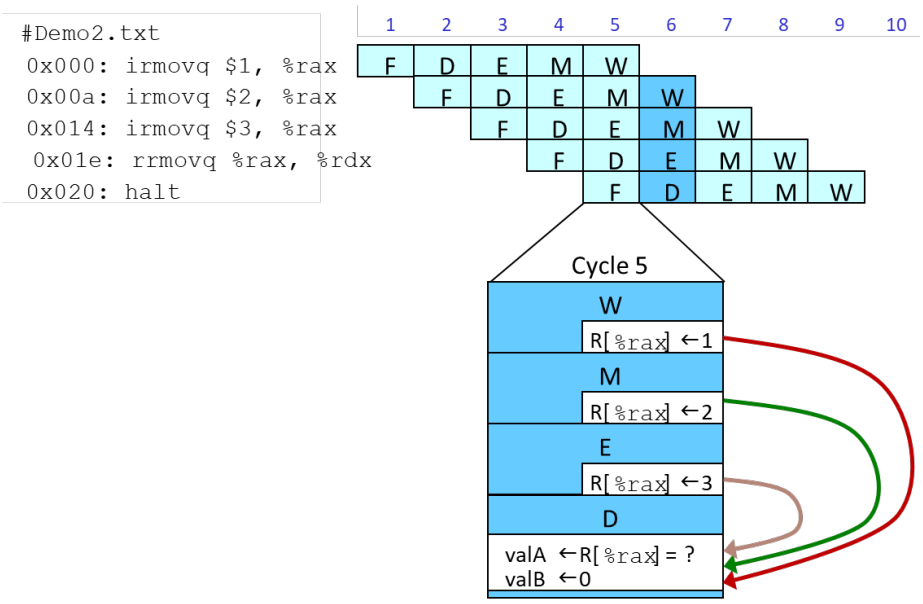


Figure 8: Pipelined Output

Multiple registers are writing into *rax*. In that case, we have to give priority to the earlier stage.So, the priority order is execute, then memory and then write back.

- If there are multiple forwarding choices
  - Which one should have priority
  - Use matching value from earliest pipeline stage



- **3.Test case 3:-**

Listing 14: Demo3

```
irmovq $128,%rdx
irmovq $3,%rcx
rmmovq %rcx,0(%rdx)
irmovq $10,%rbx
mrmovq 0(%rdx) %rax
addq %rbx,%rax
halt
```

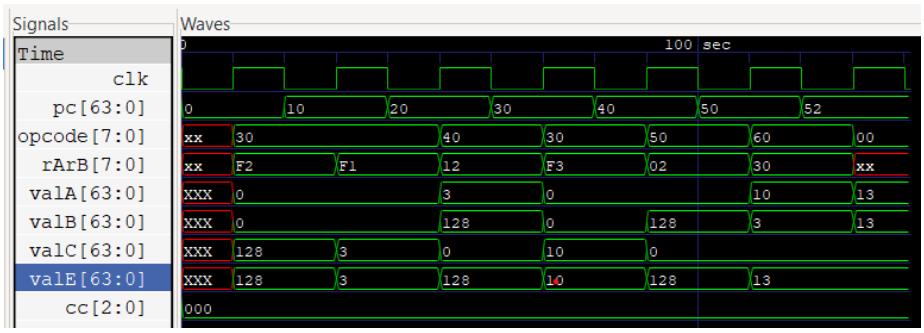


Figure 9: Sequential Output

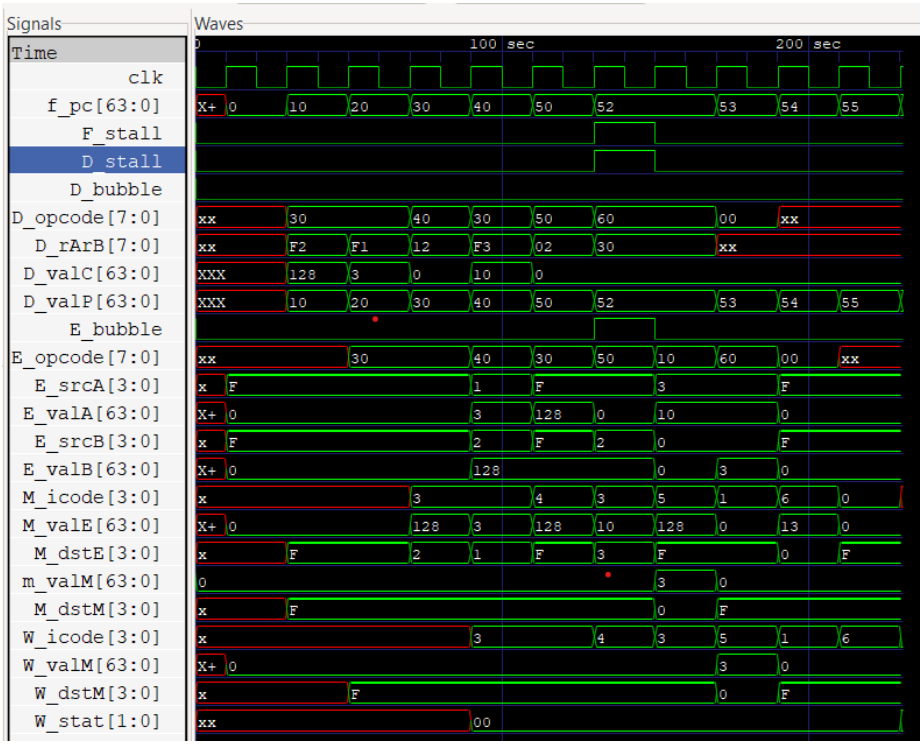
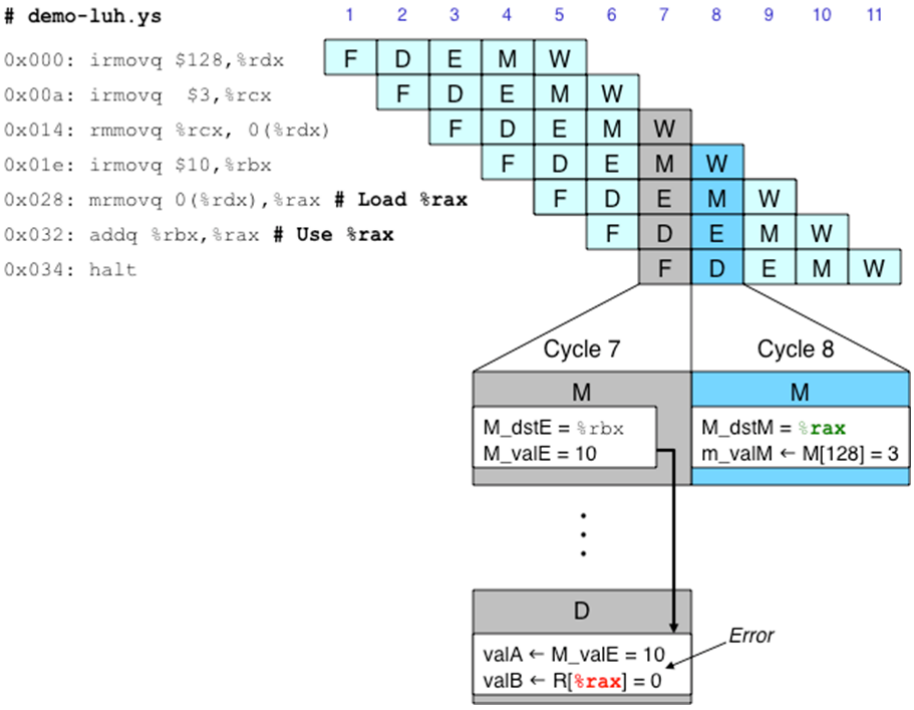


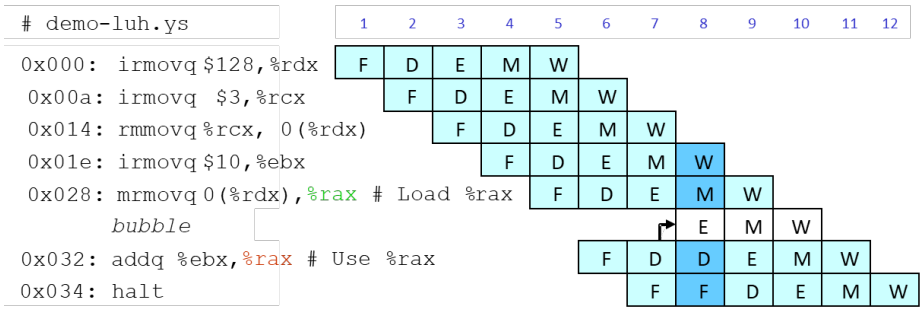
Figure 10: Pipelined Output

Here, *rbx* value is needed for addq in decode. But is written back after reading from memory after two cycle and is read from memory after one cycle. So, this will cause an error if we read it in decode then itself. This is called Load/Use Hazard.



To correct this, we can stall decode and fetch and inject bubble into execute.After that, value can be forward of *rax* directly from memory stage.

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal



- **4.Test case 4:-**

Listing 15: Demo4

```
xorq %rax,%rax
jne target
irmovq $1,%rax
halt

.target
irmovq $2,%rdx
irmovq $3,%rbx
halt
```

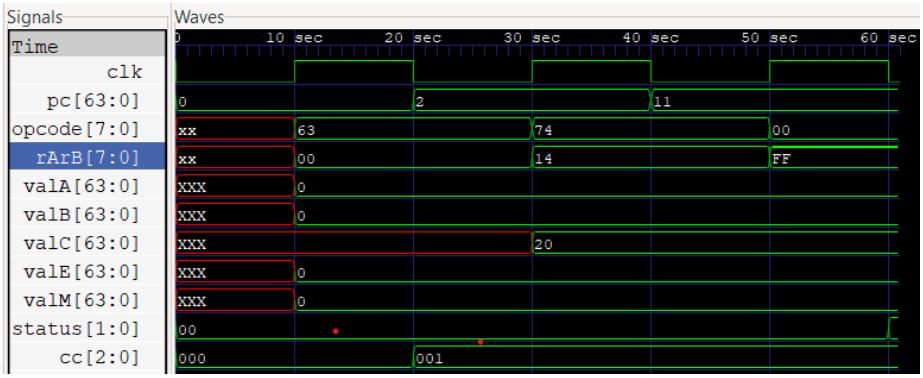


Figure 11: Sequential Output

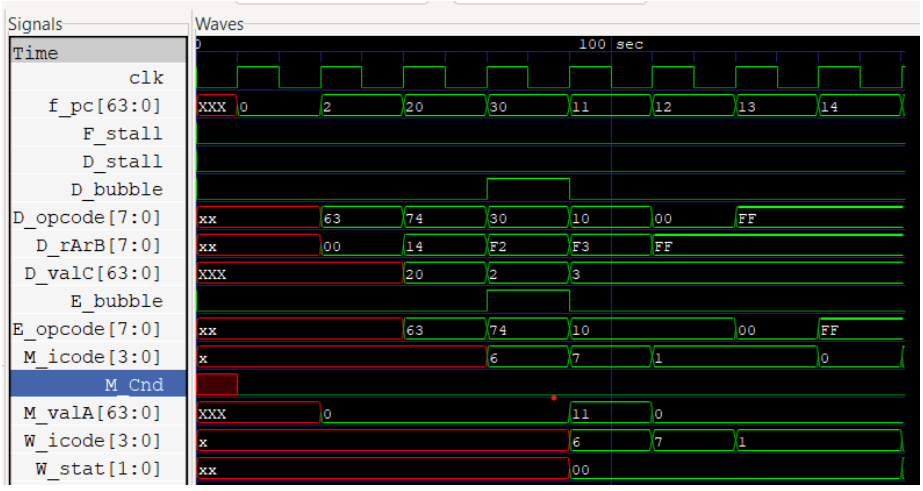


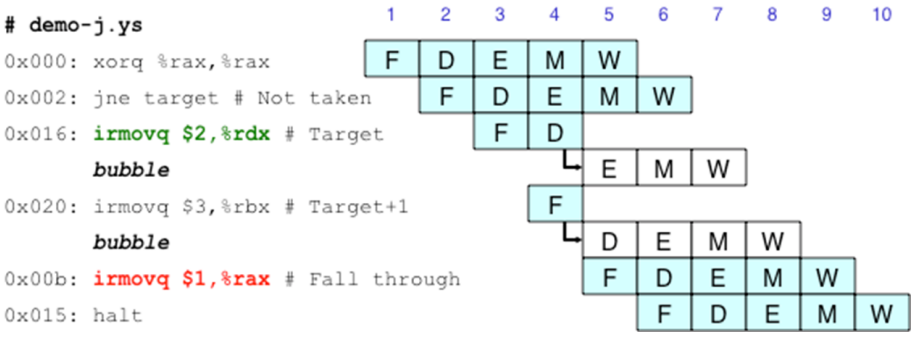
Figure 12: Pipelined Output

In sequential, instructions are executed in sequence. We will know if we should take a jump or not. But in pipelining, we won't know

if jump should be taken or not until we reach execute stage. So, we always take a jump. Later, if the jump is found to be mispredicted in the execute stage, in the next cycle, we update PC by $MvalA$ of jump instruction, which is updated to $valP$ of jump in decode stage. Also, we inject bubbles into decode and execute stage to flush out instruction fetched during the mispredicted jump.

Condition	Trigger
Mispredicted Branch	`E_icode = IJXX & !e_Cnd`

Condition	F	D	E	M	W
Mispredicted Branch	normal	bubble	bubble	normal	normal



- 5.Test case 5:-

Listing 16: Demo5

```
call dest
halt

.dest
return
```

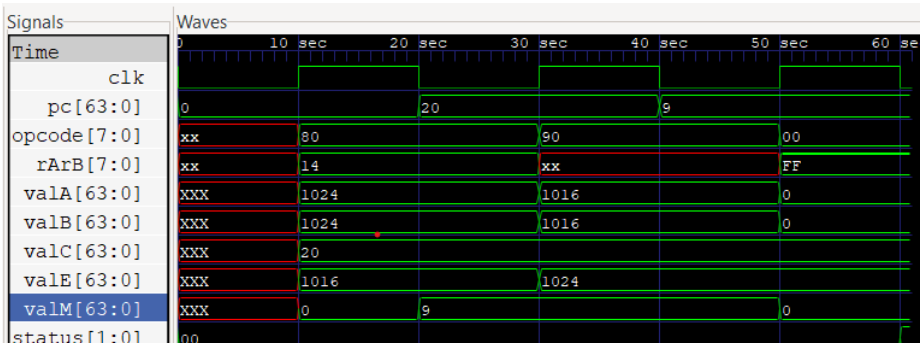


Figure 13: Sequential Output

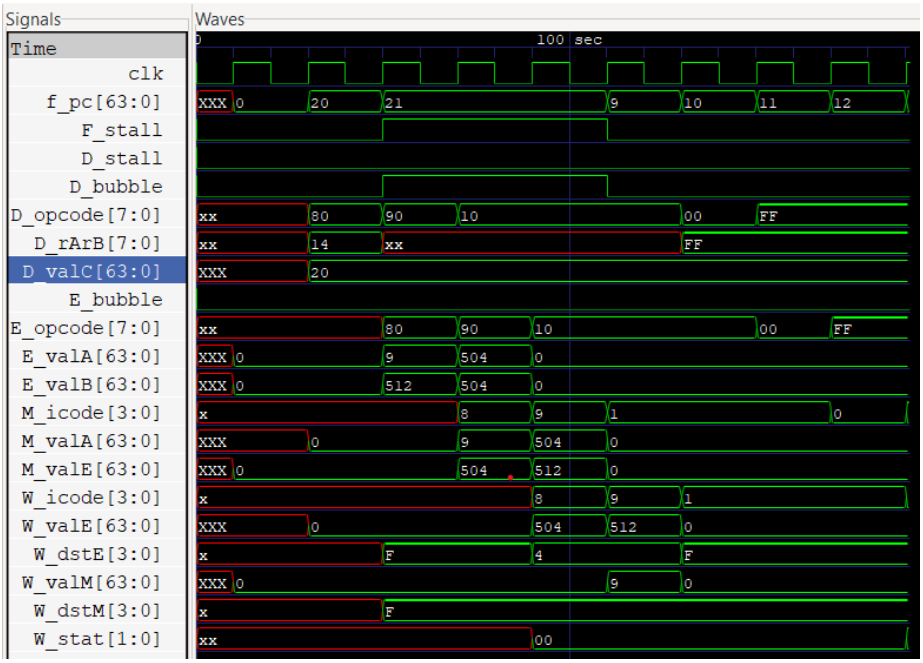
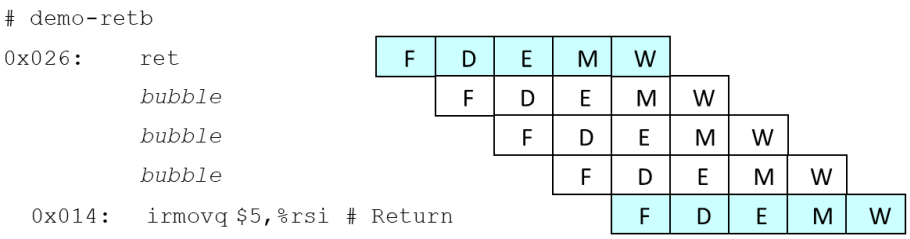


Figure 14: Pipelined Output

During call, we move to a different *PC* address. At that time, the *valP* of call is pushed into a location pointed by stack. During return, we have to read this value from memory. So, we will get the value during write back stage of return only. So, to prevent any permenant stages to processor which might happen if *cc* is set in execute stage of iinstruction following return. So, bubble is injected in decode and fetch is stalled.

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

Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal



- 6.Test case 6:-

Listing 17: Demo6

```
call dest1

.dest1
xorq %rax %rax
jump dest2
nop
nop
nop
return

.dest2
return
```

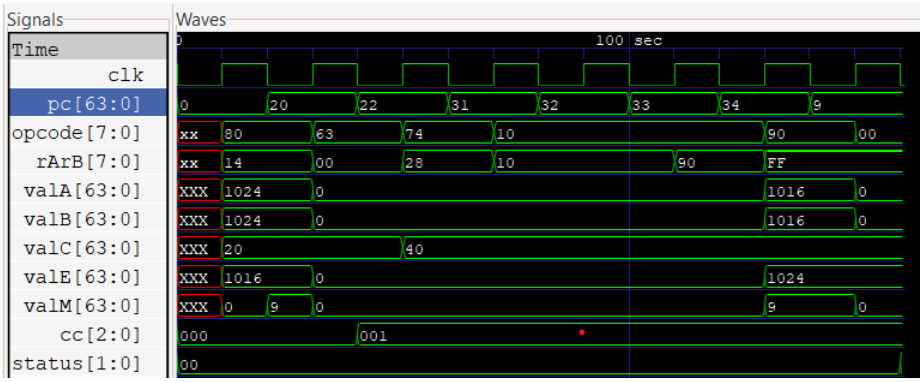


Figure 15: Sequential Output

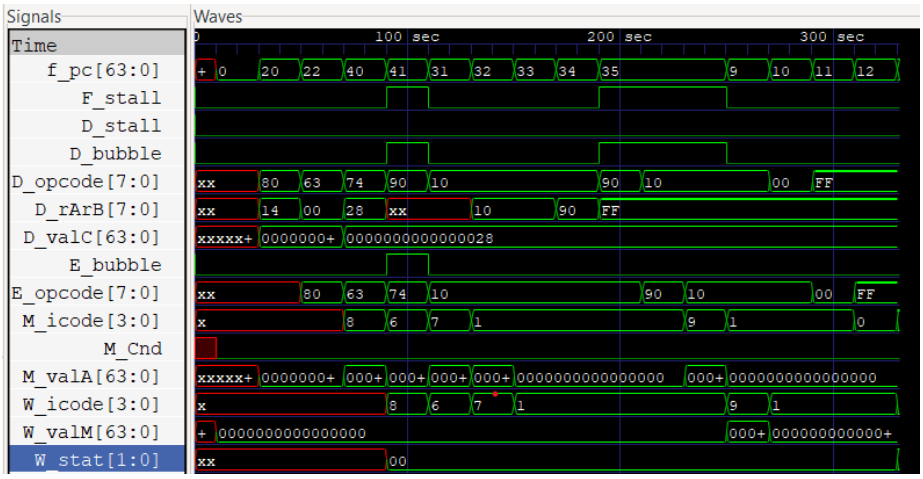
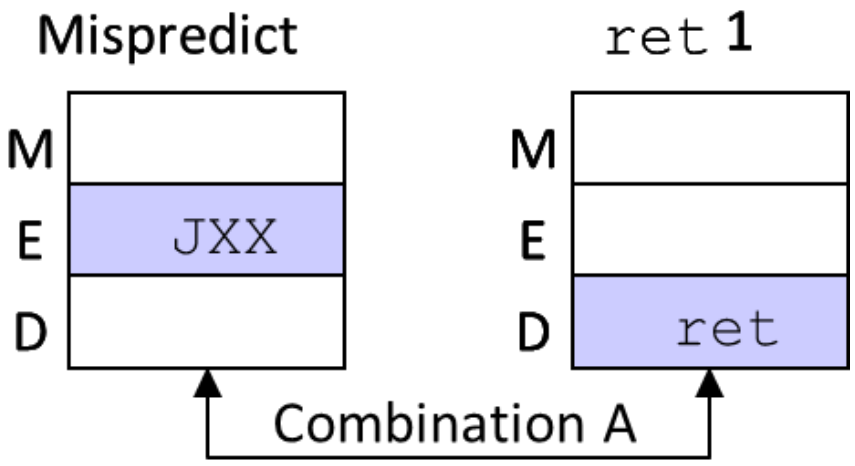


Figure 16: Pipelined Output

This is a test case, which has mispredicted jump, followed by return at the jump destination. When this mispredicted jump is in execute and return at destination location of jump is in decode, in next cycle return calls for stall in fetch and bubble in decode, whereas due to mispredicted jump, it calls for normal opeartion in fetch and bubble in decode and execute. So, combining both, fetch is stalled and decode and execute is bubbled.



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal
Combination	*stall*	*bubble*	*bubble*	*normal*	*normal*

- 7.Test case 7:-

Listing 18: Demo7

```
irmovq $8,%rdx
irmovq $70 %rax
rmmovq  %rdx 0(%rax)
rmmovq %rax 0(%rdx)
nop
nop
nop
mrmovq %rsp 0(%rax)
ret

.71
halt
```

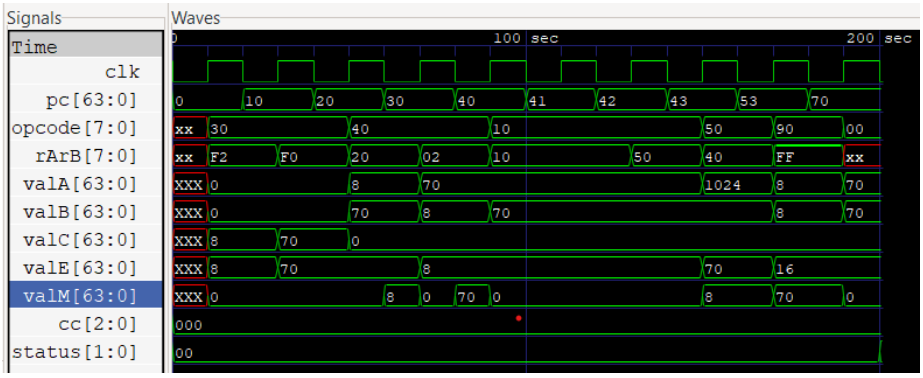


Figure 17: Sequential Output

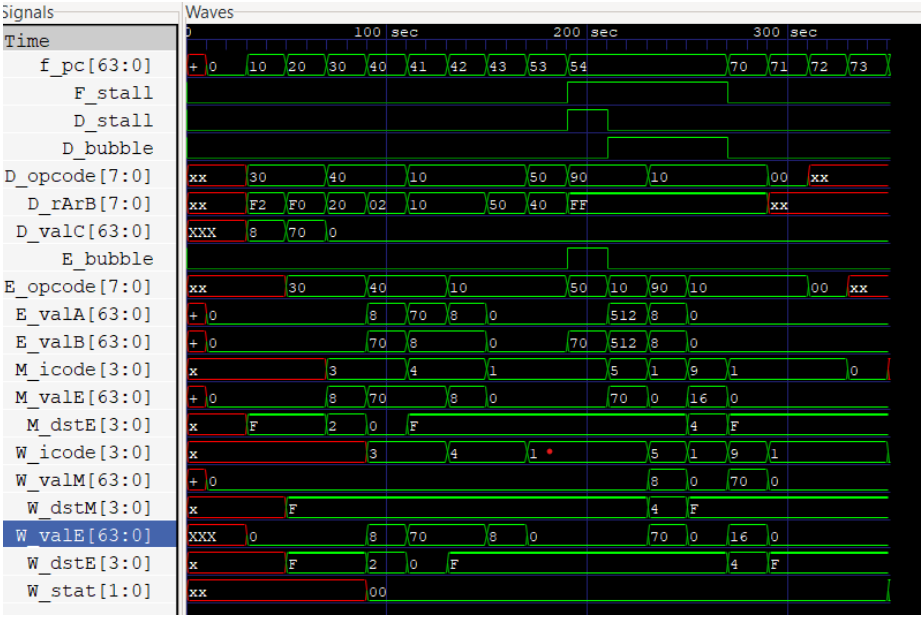
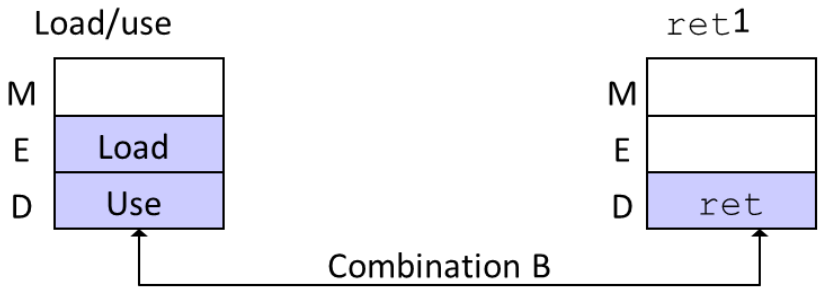


Figure 18: Pipelined Output

In this testcase, we are loading value into stack-pointer *rsp* from memory using *mrmovq*.This is followed by return. When *ret* is in decode cycle and *mrmovq* in execute, in next cycle, *ret* will try to bubble decode and stall fetch, whereas *mrmovq* due to data dependencies or load/use hazard will try to stall fetch and decode and bubble execute. So, there will be attempt to stall and bubble decode which will lead to pipeline error,



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	*stall*	*bubble + stall*	*bubble*	*normal*	*normal*

This is corrected by giving priority to load/use and return will be held in decode stage for next cycle. Then,*mrmovq* will reach memory stage and then we can do decoding of *ret* instruction.Then, return will continue as usual.

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	*stall*	*stall*	*bubble*	*normal*	*normal*

Implementation of Forwarding

- Forwarding, also known as data forwarding or bypassing, is a technique used in pipelined processor designs to improve performance by reducing or eliminating data hazards. Data hazards occur when an instruction depends on the result of a previous instruction that has not yet produced its result, leading to stalls in the pipeline.
- Forwarding allows the processor to detect these hazards and forward the necessary data directly from one stage of the pipeline to another, bypassing intermediate stages, to avoid stalls and maintain correct operation.

```
int d_valA =
[
   Use incremented PC
   D_icode in { ICALL, IJXX } : D_valP;

   Forward valE from execute
   d_srcA == e_dstE : e_valE;

   Forward valM from memory
   d_srcA == M_dstM : m_valM;

   Forward valE from memory
   d_srcA == M_dstE : M_valE;

   Forward valM from write back
   d_srcA == W_dstM : W_valM;

   Forward valE from write back
   d_srcA == W_dstE : W_valE;

   Use value read from register file
   1 : d_rvalA;
];
```

```
int d_valB =
[

   Forward valE from execute
   d_srcB == e_dstE : e_valE;

   Forward valM from memory
   d_srcB == M_dstM : m_valM;

   Forward valE from memory
   d_srcB == M_dstE : M_valE;

   Forward valM from write back
   d_srcB == W_dstM : W_valM;

   Forward valE from write back
   d_srcB == W_dstE : W_valE;

   Use value read from register file
   1 : d_rvalB;
];
```

Special Control Cases

So, based on different pipeline hazards and control combinations discussed in the testcases, we can define the control cases of bubble and stall in the different stages.

Detection

Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & !e_Cnd

Action (on next cycle)

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

Control Combination

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	*stall*	*stall*	*bubble*	*normal*	*normal*

Pipeline control

```
bool F_stall =
Conditions for a load/use hazard
E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
||
Stalling at fetch while ret passes through pipeline
IRET in { D_icode, E_icode, M_icode };

bool D_stall =
Conditions for a load/use hazard
E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };

bool D_bubble =
Mispredicted branch
(E_icode == IJXX && !e_Cnd)
||
Stalling at fetch while ret passes through pipeline
IRET in { D_icode, E_icode, M_icode }
but not condition for a load/use hazard
&& !(E_icode in { IMRMOVQ, IPOPQ }
&& E_dstM in { d_srcA, d_srcB });


bool E_bubble =
Mispredicted branch
(E_icode == IJXX && !e_Cnd)
Load/use hazard
E_icode in { IMRMOVQ, IPOPQ } and E_dstM in { d_srcA, d_srcB };
```

Challenges faced

We faced some problems while transitioning from sequential to pipelined version.We had to make several changed ins sequential apart from moving *PC* stage, relabelling and adding pipelined registers. Also, managing lot of inputs and outputs caused problems as the name mostly had a case difference only.

Note:- Only testcases showing pipeline hazards have been shown in testing. Other testcases have been tried and tested. One particular testcase is one which finds sum of elements in an array which we will be showing during evals.