Habib University

Dhanani School of Science and Engineering

EE/CS 371/330 - Computer Architecture

Final Project Report

Risc-V Processor

Hafsa Irfan (hi05946) **Haania Siddiqui (hs06188)** **Adnan Asif (aa06204)**

June 3, 2021

Task 1 - Single Cycle Processor With Bubble Sort

1. Bubble Sort Assembly Code

x22 = i, x23 = j, x28 = address_i, x29 = address_j, x5 = temp, x8 = 5

```
Loop1:
blt x22, x8,Loop2     # i < 5
beq x0,x0,Exit

Loop2:
addi x23,x22,0        # j = i
addi x29,x28,0        # address_j = address_i
blt x23,x8,Loop3      # j < 5
beq x0,x0, Loop1

Loop3:
ld x12, 0(x28)        # x12 = a[i]
ld x13, 0(x29)        # x13 = a[j]
blt x12,x13,if
addi x23,x23,1        # j++
addi x29,x29,8        # address_j +=8
blt x23,x8,Loop3      # j < 8
addi x22,x22,1        # i++
addi x28, x28,8       # address_i +=8
beq x0,x0,Loop1

if:
add x5,x12,x0         # temp = a[i]
sd x13, 0(x28)        # a[i] = a[j]
sd x5, 0(x29)         # a[j] = temp
addi x23,x23,1        # j++
addi x29,x29,8        # address_j +=8
blt x23,x8,Loop3      # j < 8
addi x22,x22,1        # i++
addi x28, x28, 8      # address_i +=8
beq x0,x0,Loop1

Exit:


```

Listing 1: main

Bubble Sort with only add, addi, sub, ld, sd, blt and beq instructions.

The C++ Code for Bubble Sort

```
   for (int i = 0; i<5; i++){
      for (int j = i; j<5; j++){
          if (a[i] < a[j]){
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
          }
        }
    }

```

Listing 2: main

2. Changes to The Single - Cycle Processor For Blt Instruction

a CONTROL UNIT

```
module control_Unit(OpCode, ALUOp, Branch, MemRead, MemtoReg, MemWrite, ALUSrc,
    RegWrite) ;
         input [6:0] OpCode;
     output  reg Branch;
     output reg MemRead;
     output reg MemtoReg;
     output reg MemWrite;
     output reg ALUSrc;
```

```
    output reg RegWrite;
        output reg [1:0] ALUOp;

always @(OpCode)
  begin
    case(OpCode)
    7'b0110011:   //R-type
      begin
        ALUSrc   = 1'b0;
        MemtoReg = 1'b0;
        RegWrite = 1'b1;
        MemRead  = 1'b0;
        MemWrite = 1'b0;
        Branch   = 1'b0;
        ALUOp    = 2'b10;
      end
   7'b0000011:   //I-type (ld)
      begin
        ALUSrc   = 1'b1;
        MemtoReg = 1'b1;
        RegWrite = 1'b1;
        MemRead  = 1'b1;
        MemWrite = 1'b0;
        Branch   = 1'b0;
        ALUOp    = 2'b00;
      end
    7'b0100011: //I-type (sd)
      begin
        ALUSrc   = 1'b1;
        MemtoReg = 1'bx;
        RegWrite = 1'b0;
        MemRead  = 1'b0;
        MemWrite = 1'b1;
        Branch   = 1'b0;
        ALUOp    = 2'b00;
      end
     7'b0010011: //addi
      begin
        ALUSrc   = 1'b1;
        MemtoReg = 1'b0;
        RegWrite = 1'b1;
        MemRead  = 1'b0;
        MemWrite = 1'b0;
        Branch   = 1'b0;
        ALUOp    = 2'b00;
      end
     7'b1100011: // SB- type (beq/blt)
       begin
        ALUSrc   = 1'b0;
        MemtoReg = 1'bx;
        RegWrite = 1'b0;
        MemRead  = 1'b0;
        MemWrite = 1'b0;
        Branch   = 1'b1;
        ALUOp    = 2'b01;
       end

    default:
     begin
      ALUSrc   = 1'b0;
      MemtoReg = 1'b0;
      RegWrite = 1'b0;
      MemRead  = 1'b0;
      MemWrite = 1'b0;
      Branch   = 1'b0;
      ALUOp    = 2'b00;
```

```
74          end
75    endcase
76      end
77
78    endmodule
```

Listing 3: main

Lines 54 - 63:

- The seven control signals are set based on the input signals to the control unit, which are the OpCode bits 6:0.
- The OpCode for beq/blt instruction is same and so their signals are also same since both require jumping to a specific memory address with no reading/writing.
- ALUOp is assigned '01' for both blt and beq

b ALU CONTROL

```
1  module ALU_Control( ALUOp, Funct, Operation);
2    input [1:0] ALUOp;
3    input [3:0] Funct;
4    output reg [3:0] Operation;
5
6
7
8    always @(*)
9      begin
10       case(ALUOp)
11     2'b00:
12       begin
13       Operation = 4'b0010;
14       end
15      2'b01:                          // branch type instructions
16           begin
17             case(Funct[2:0])
18             3'b000:                  // beq
19               begin
20                Operation = 4'b0110;  // subtract
21               end
22             3'b100:                  // blt
23               begin
24                 Operation = 4'b0100; // less than operation
25               end
26                 endcase
27           end
28
29
30      2'b10:
31        begin
32          case(Funct)
33             4'b0000:
34               begin
35               Operation = 4'b0010;
36               end
37              4'b1000:
38               begin
39               Operation = 4'b0110;
40               end
41              4'b0111:
42               begin
43               Operation = 4'b0000;
44               end
45                4'b0110:
46               begin
47               Operation = 4'b0001;
48               end
49          endcase
```

```
50        end
51        endcase
52      end
53
54  endmodule
```

Listing 4: main

- Modified the ALU Control that generates the 4-bit ALU Control input.
- The Control Unit takes as input the Func Field [1-bit from funct7 field (bit 30) + 3-bit funct3 field (bits 14:12)] and a 2-bit control field which we call ALUOp. The output is a 4-bit signal (determined by Func and ALUOp field) that directly controls the ALU by generating one of six operations to be performed in our case.
- ALUOp indicates whether the operation to be performed should be add (00) for loads and stores, or be determined by the operation encoded in the funct7 and funct3 fields (10, 01). We added an additional case structure when ALUOp was '01' i.e. when there was a branch type instruction.
  We checked for two types under that.
  Lines 14 - 26:
  (a) Beq instruction (tests for equality): When Func field's least three significant bits were '000' (i.e. when func3 field was '000'), we would require a subtract operation to subtract and test if equals 0 and so subtract operation was assigned.
  (b) Blt instruction (tests for lesser than): When Func field's least three significant bits were '100' (i.e. when func3 field was '100'), we would require a check less than operation and so less than operation was assigned.

c ALU

```
1   module ALU(a,b,ALUop,Result,Zero);
2     input [63:0]a, b;
3     input [3:0] ALUop;
4     output reg [63:0] Result;
5     output Zero;
6
7
8
9     always@(a or b or ALUop)
10            begin
11                 case (ALUop)
12                    4'b0000: Result = a & b;            // AND
13                    4'b0001:  Result = a | b;           // OR
14                    4'b0010:  Result = a +  b;          // Addition
15                    4'b0110:  Result = a - b;           // Subtraction
16                    4'b1100:  Result = ~(a | b);        // Nor
17                    4'b0100:  Result = ( a < b)? 0: 1;  // Lesser than
18                 endcase
19            end
20    assign Zero =(Result==0); //equal or <
21
22
23  endmodule
```

Listing 5: main

- For each of the operations assigned in ALU_Control, appropriate operation is performed in ALU. Modified the ALU to add '<' operation for the blt instruction.
- If first value is lesser than the second value, then '0' is assigned to Result. If Result == 0 then just like the beq instruction, '0' would be assigned to Zero . With this, no additional changes in the hardware are required to check for an additional branch type instruction.
- When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is high (when $a < b$ or $a - b = 0$) inline with our hardware structure where a selection line of mux which decides the address of the next instruction is Branch & Zero.

3. Changes To The Single Cycle Processor For Running Bubble Sort

a INSTRUCTION MEMORY

```
module InstructionMemory(Inst_Address,Instruction);
  input [63:0] Inst_Address;
  output reg [31:0]Instruction;
  reg [7:0] InstMemory [95:0];
  integer i;

  initial
    begin
      //blt x22, x8,8
      InstMemory[0] = 8'b01100011;
      InstMemory[1] = 8'b01000100;
      InstMemory[2] = 8'b10001011;
      InstMemory[3] = 8'b0;

      //beq x0,x0,92
          InstMemory[4] = 8'b01100011;
      InstMemory[5] = 8'b00001110;
      InstMemory[6] = 8'b00000000;
      InstMemory[7] = 8'b00000100;

      //addi x23,x22,0
      InstMemory[8] = 8'b10010011;
      InstMemory[9] = 8'b00001011;
      InstMemory[10] = 8'b00001011;
      InstMemory[11] = 8'b0;

      //addi x29,x28,0
      InstMemory[12] = 8'b10010011;
      InstMemory[13] = 8'b00001110;
      InstMemory[14] = 8'b00001110;
      InstMemory[15] = 8'b0;

      //blt x23,x8,Loop3
      InstMemory[16]= 8'b01100011;
      InstMemory[17] = 8'b11000100;
      InstMemory[18] = 8'b10001011;
      InstMemory[19] = 8'b0;

      //beq x0,x0, Loop1
      InstMemory[20] = 8'b11100011;
      InstMemory[21] = 8'b00000110;
      InstMemory[22] = 8'b00000000;
      InstMemory[23] = 8'b11111110;

      //ld x12, 0(x28) - x12 = a[i]
      InstMemory[24] = 8'b00000011;
      InstMemory[25] = 8'b00110110;
      InstMemory[26] = 8'b00001110;
      InstMemory[27] = 8'b00000000;


      //ld x13, 0(x29) - x13 = a[j]
      InstMemory[28]=8'b10000011;
      InstMemory[29]=8'b10110110;
      InstMemory[30]=8'b00001110;
      InstMemory[31]=8'b00000000;

      //blt x12,x13,if
      InstMemory[32]=8'b01100011;
      InstMemory[33]=8'b01001110;
      InstMemory[34]=8'b11010110;
      InstMemory[35]=8'b00000000;

```

```
64        //addi x23,x23,1 - j++
65        InstMemory[36]=8'b10010011;
66        InstMemory[37]=8'b10001011;
67        InstMemory[38]=8'b00011011;
68        InstMemory[39]=8'b00000000;
69
70        //addi x29,x29,8 - locj+= 8
71        InstMemory[40]=8'b10010011;
72        InstMemory[41]=8'b10001110;
73        InstMemory[42]=8'b10001110;
74        InstMemory[43]=8'b00000000;
75
76        //blt x23,x8,Loop3
77        InstMemory[44]=8'b11100011;
78        InstMemory[45]=8'b11000110;
79        InstMemory[46]=8'b10001011;
80        InstMemory[47]=8'b11111110;
81
82        //addi x22,x22,1 #i++
83        InstMemory[48]=8'b00010011;
84        InstMemory[49]=8'b00001011;
85        InstMemory[50]=8'b00011011;
86        InstMemory[51]=8'b00000000;
87
88        //addi x28, x28,8 - loci+=8
89        InstMemory[52]=8'b00010011;
90        InstMemory[53]=8'b00001110;
91        InstMemory[54]=8'b10001110;
92        InstMemory[55]=8'b0;
93
94        //beq x0,x0,Loop1
95        InstMemory[56]=8'b11100011;
96        InstMemory[57]=8'b00000100;
97        InstMemory[58]=8'b00000000;
98        InstMemory[59]=8'b11111100;
99
100           //add x5,x12,x0 - temp = a[i]
101       InstMemory[60]=8'b10110011;
102       InstMemory[61]=8'b00000010;
103       InstMemory[62]=8'b00000110;
104       InstMemory[63]=8'b0;
105
106           //sd x13, 0(x28) - a[i]=a[j]
107       InstMemory[64]=8'b00100011;
108       InstMemory[65]=8'b00110000;
109       InstMemory[66]=8'b11011110;
110       InstMemory[67]=8'b0;
111
112       //sd x5, 0(x29) - a[j]=temp
113       InstMemory[68]=8'b00100011;
114       InstMemory[69]=8'b10110000;
115       InstMemory[70]=8'b01011110;
116       InstMemory[71]=8'b0;
117
118           //addi x23,x23,1 - j++
119       InstMemory[72]=8'b10010011;
120       InstMemory[73]=8'b10001011;
121       InstMemory[74]=8'b00011011;
122       InstMemory[75]=8'b0;
123
124           //addi x29,x29,8 - locj+= 8
125       InstMemory[76]=8'b10010011;
126       InstMemory[77]=8'b10001110;
127       InstMemory[78]=8'b10001110;
128       InstMemory[79]=8'b00000000;
129
```

```
            //blt x23,x8,Loop3:
        InstMemory[80]=8'b11100011;
        InstMemory[81]=8'b11000100;
        InstMemory[82]=8'b10001011;
        InstMemory[83]=8'b11111100;

            //addi x22,x22,1  - i++
        InstMemory[84]=8'b00010011;
        InstMemory[85]=8'b00001011;
        InstMemory[86]=8'b00011011;
        InstMemory[87]=8'b0;

            //addi x28, x28,8 - loci+=8
        InstMemory[88]=8'b00010011;
        InstMemory[89]=8'b00001110;
        InstMemory[90]=8'b10001110;
        InstMemory[91]=8'b0;

            //beq x0,x0,Loop1
        InstMemory[92]=8'b11100011;
        InstMemory[93]=8'b00000010;
        InstMemory[94]=8'b00000000;
        InstMemory[95]=8'b11111010;

    end
  always @(Inst_Address)
    begin
      Instruction = {InstMemory[Inst_Address + 3], InstMemory[Inst_Address + 2],
    InstMemory[Inst_Address + 1], InstMemory[Inst_Address]} ;
    end

endmodule

```

Listing 6: main

Each instruction of the assembly language bubble sort was converted into a 32-bit address.

b REGISTER FILE

```
initial
    begin
      for (i = 0; i < 32; i = i + 1)
      begin
        Registers[i] = 0;
      end
      Registers[8] = 5;
```

Listing 7: main

Assigned the 8th register (x8) the value 5 ie number the elements we'll be sorting in bubble sort and all the other registers are assigned the value 0.

c DATA MEMORY

```
 initial
    begin
      for (i = 0; i < 64; i = i + 1)
      begin
        DataMemory[i] = 0;
      end
      DataMemory[0]  = 8'd2;
      DataMemory[8]  = 8'd1;
      DataMemory[16] = 8'd3;
      DataMemory[24] = 8'd0;
      DataMemory[32] = 8'd4;
```

Listing 8: main

The elements in memory are initially kept in the order: 2,1,3,0,4.
After sorting they must be in order: 4,3,2,1,0.

4. Bubble Sort Results

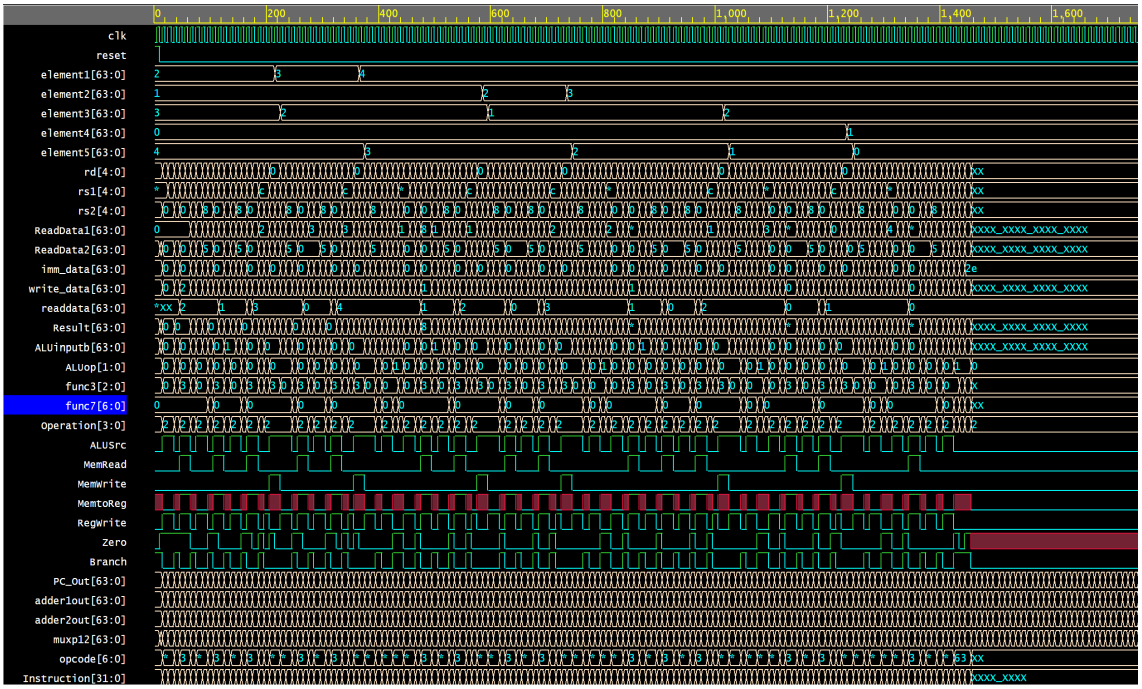


Figure 1:

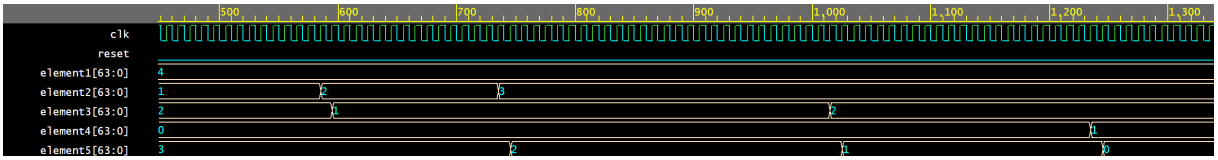


Figure 2: Memory Contents

EDA Link for Detailed EP Wave: **https://www.edaplayground.com/x/a4hy**

Task 2 - Introducing Pipelining

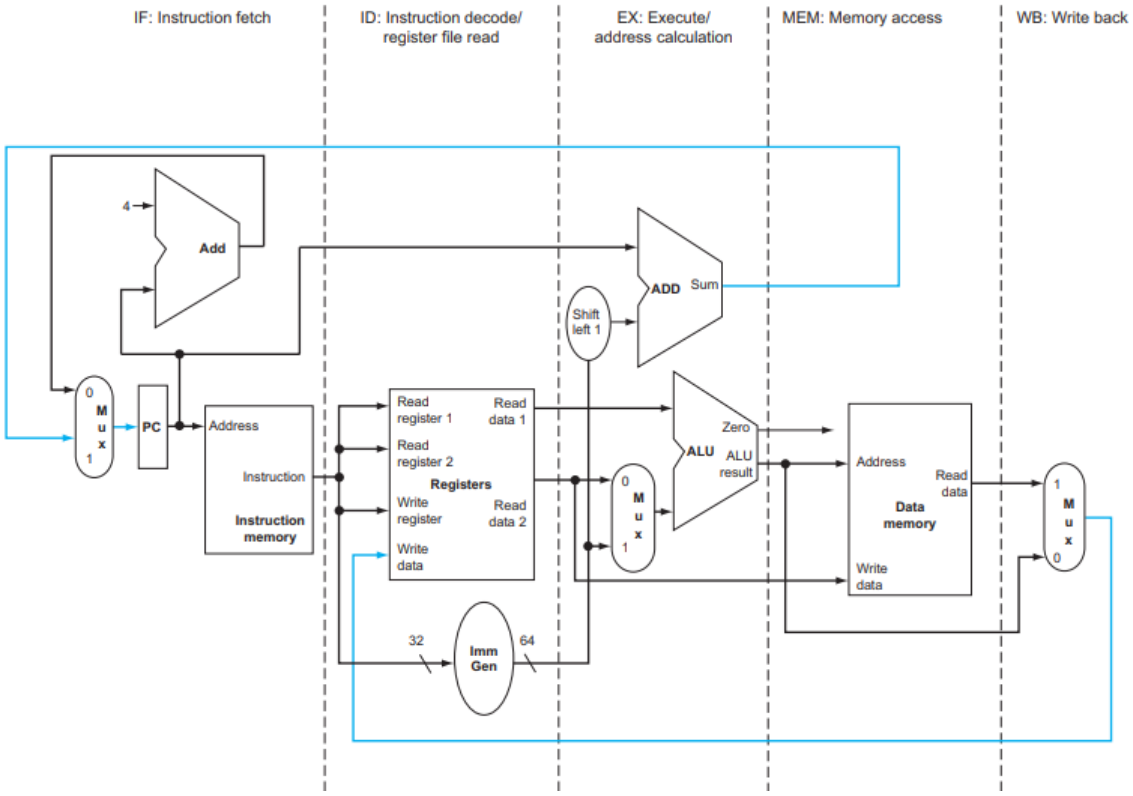
In the first part we saw a single cycle processor which was capable of running Bubble sort, but a problem with that type of implementation is that the processor executes a single instruction at a time, and once that instruction is completed only then execution of the next instruction is started. We can already see how this would be extremely inefficient and would waste a lot of processing power, since most of the components of our processors would be doing nothing. This is why in this part we would try to overcome this by modifying our Single cycle processor by introducing pipelining.

Pipelining would let us execute multiple instructions at the same time, this depends on the number of stages we have in our pipeline. This section would discuss how this works in depth, but for now one can think about this as if one component is working on certain part of the instruction then the other would be working on the other part. For our Risc-V processor, we'll be adding a five stage pipeline, this means that our processor would be able to work on 5 instructions at a time. We'll divide the processor we implemented in following 5 stages:

- IF : Instruction Fetch
- ID : Instruction decode and register file read

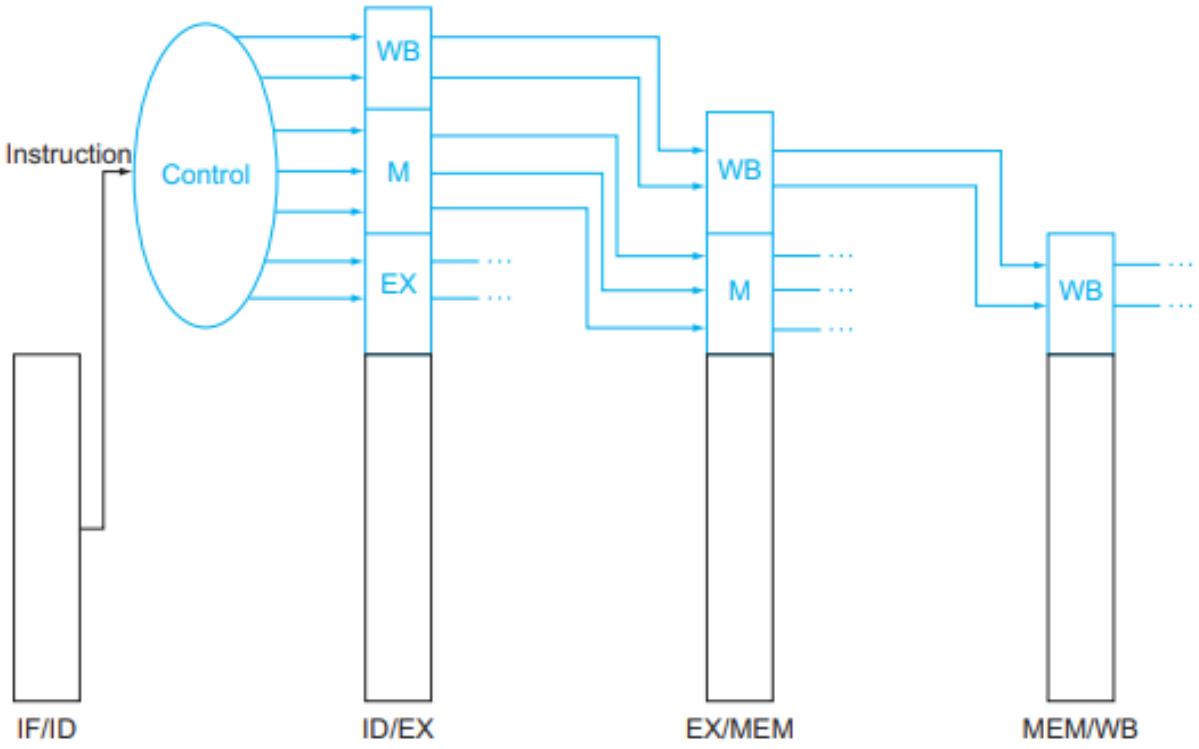
- EX : Execution or address calculation
- MEM : Data memory access
- WB : Write back

The figure below shows how we'll be dividing the single cycle processor. Note that this diagram doesn't represent the final pipelining.

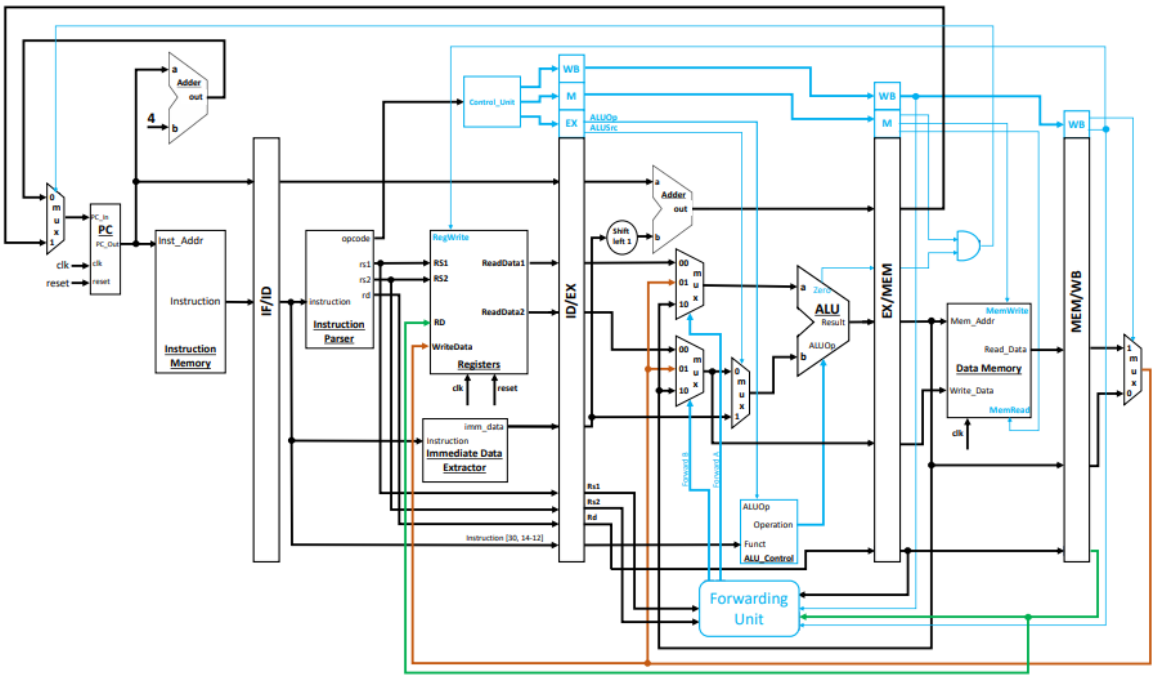


The ideal pipeline would be the one where instructions only move forward, but on the figure above, we can see two wire represented by blue, which show that output from a component is going back, these are two exceptions in out lefft to right intrsuction flow. This is due to the write-back stage, which places the result back into the register file in the middle of the datapath and the selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.

Now of-course the main question that arises is that how are we going to actually implement this, and what exactly is the significance of each stage. So let's get into it. So, in order to add pipelines to our current processor, we'll add 4 intermediate registers, a control line and a forwarding unit. Along with this, we have also extended these registers to store the control lines passed from one stage to another. These registers would be synchronized with the clock and on each positive edge they would either send the stored contents for further processing or they would be flushed. The figure below represents the registers we are going to add in our processor.



Before we get into the explanation of how each stage works and what does it exactly do, and before getting into the verilog implementation, let's see the bigger picture through the following figure, which represents what we are trying to implement.



Stage 1 - Instruction Fetch (IF)

This is the first stage of our processor, and as your can see in the figure above, the left most part. As the name suggests, this stage takes care of fetching the instruction from the memory, and in order to do this, it first calculates the address of the instruction to be fetched through the PC counter, and uses the Instruction memory module to fetch the instruction and then send it to the next stage through the IF/ID register. This also takes care of the jump address if the instruction before it was a branch instruction, in such case it receives the offset from the EX/MEM register.

The modules used in this stage are already explained in the task one, so now let us see the wires we use, that are then connected to the IF/ID register.

```
module IF_ID(clk,reset,PC_Out, Instruction, IF_ID_Inst, IF_ID_PC_Out);

  input clk, reset;
  input [63:0] PC_Out;
  input [31:0] Instruction;
  output reg [31:0] IF_ID_Inst;
  output reg [63:0] IF_ID_PC_Out;

  always @(*)

    if (reset)
        begin
          IF_ID_Inst=0;

          IF_ID_PC_Out=0;

        end
   always@(posedge clk)
     if (clk)
        begin
          IF_ID_Inst=Instruction;
          IF_ID_PC_Out=PC_Out;
        end

endmodule
```

Listing 9: IF/ID register

In our top level module, following connections are made before sending everything to the IF/ID register which on the next clock cycle would send the contents to the further stage. The the outputs from this register are the intermediate connections between the Instruction Fetch and the Instruction decode stage. Also note that in the code below, the muxp12 is the offset value if the previous intruction was a branch instruction,

```

 Program_Counter p1(clk, reset, muxp12,PC_Out, PC_Write);

  Adder AddInc (PC_Out, 64'd4, adder1out);

  InstructionMemory I1(PC_Out, Instruction);

  IF_ID if_id (clk,reset,PC_Out, Instruction, IF_ID_Inst, IF_ID_PC_Out);
```

Stage 2 - Instruction Decode (ID)

This is the second stage of our pipeline and this takes care of decoding the instruction and reading from the registers or writing to the register. So first it gets the instruction fetched by the IF stage and this receives the information through following wires.

```
    wire [63:0] IF_ID_PC_Out;
    wire [31:0] IF_ID_Inst;
```

And then this Instruction is passed on to the instruction parser and the Data Extractor module, so now the 32 bit instructions is decoded and we know it's opcode, rd, rs1, rs2 and then the RegisterFile reads the contents of the registers or write back to it (Note than in order to write back we'll get signals from the MEM/WEB register which means it is a right to left operation but it doesn't disturb the flow of the program). An essential part of this stage is the Control Unit which uses opcode to to forward signals which would be the control line for ALU, and other signals that would act as a register to determine other operations in the processor. Then all the contents of the intermediate wires of IF/ID stage are forward to the ID/EX stage.

```
    wire [6 :0] opcode;
  wire [4:0] IF_ID_rd;
  wire [2:0] IF_ID_funct3;
  wire [4:0] IF_ID_rs1;
```

```
  wire [4:0] IF_ID_rs2;
  wire [6:0] IF_ID_funct7;
  //output of Data Extractor
  wire [63:0] IF_ID_imm_data;
  // output of CU
  wire IF_ID_Branch, IF_ID_MemRead,  IF_ID_MemtoReg,IF_ID_MemWrite,IF_ID_ALUsrc,
    IF_ID_RegWrite;
  wire [1:0] IF_ID_ALUop;
  //output of register file

  wire [63:0] IF_ID_ReadData1;
  wire [63:0] IF_ID_ReadData2;


  InstructionParser ip1 (IF_ID_Inst, opcode,IF_ID_rd, IF_ID_funct3, IF_ID_rs1,
    IF_ID_rs2, IF_ID_funct7);

  imm_data_extractor d1(IF_ID_Inst, IF_ID_imm_data);


  Control_Unit c1(opcode,IF_ID_Branch, IF_ID_MemRead,  IF_ID_MemtoReg,IF_ID_ALUop,
    IF_ID_MemWrite,IF_ID_ALUsrc, IF_ID_RegWrite);


  registerFile r1 (clk, reset, MEM_WB_mux, IF_ID_rs1, IF_ID_rs2, MEM_WB_rd,
    MEM_WB_RegWrite, IF_ID_ReadData1, IF_ID_ReadData2);



  ID_EX i2 (clk,reset
            ,{IF_ID_Inst[30],IF_ID_Inst[14:12]},
            IF_ID_rs1,
            IF_ID_rs2,
            IF_ID_rd,
            IF_ID_PC_Out,
            IF_ID_ReadData1,
            IF_ID_ReadData2,
            IF_ID_imm_data,
            IF_ID_Branch,
            IF_ID_MemRead,
            IF_ID_MemtoReg,
            IF_ID_ALUop,
            IF_ID_MemWrite,
            IF_ID_ALUsrc,
            IF_ID_RegWrite,
            ID_EX_Inst,
            ID_EX_rs1,
            ID_EX_rs2,
            ID_EX_rd,
            ID_EX_PC_Out,
            ID_EX_ReadData1,
            ID_EX_ReadData2,
            ID_EX_imm_data,
            ID_EX_Branch,
            ID_EX_MemRead,
            ID_EX_MemtoReg,
            ID_EX_ALUop,
            ID_EX_MemWrite,
            ID_EX_ALUsrc,
            ID_EX_RegWrite );
```

Now on the next positive edge the contents are forwarded to the Execution stage through the ID/EX wire, through the following register.

```
module ID_EX(
    input clk,reset,
```

```
    input [3:0] IF_ID_instruction,
        input [4:0] IF_ID_rd, IF_ID_rs1, IF_ID_rs2,
        input [63:0] IF_ID_ReadData1, IF_ID_ReadData2,
        input [63:0] IF_ID_imm_data, IF_ID_PC_Out,
        input [1:0] IF_ID_ALUOp,
    input IF_ID_ALUSrc,

        input IF_ID_BranchEq,
        input IF_ID_BranchGt,

        input IF_ID_MemRead,
        input IF_ID_MemWrite,
        input IF_ID_RegWrite,
        input IF_ID_MemtoReg,

    output reg [3:0] ID_EX_instruction,
        output reg [4:0] ID_EX_rd, ID_EX_rs2, ID_EX_rs1,
        output reg [63:0] ID_EX_imm_data, ID_EX_ReadData2,
        output reg [63:0] ID_EX_ReadData1, ID_EX_PC_Out,
        output reg ID_EX_ALUSrc,
        output reg [1:0] ID_EX_ALUOp,

        output reg ID_EX_BranchEq,
        output reg ID_EX_BranchGt,

        output reg ID_EX_MemRead,
        output reg ID_EX_MemWrite,
        output reg ID_EX_RegWrite,
        output reg ID_EX_MemtoReg
);

always@(posedge clk or reset)
begin
    if(reset)
        begin
            ID_EX_instruction = 0;
                        ID_EX_rd = 0;
                        ID_EX_rs2 = 0;
                        ID_EX_rs1 = 0;
                        ID_EX_imm_data = 0;
                        ID_EX_ReadData2 = 0;
                        ID_EX_ReadData1 = 0;
                        ID_EX_PC_Out = 0;
                        ID_EX_ALUOp = 0;
            ID_EX_ALUSrc = 0;
                        ID_EX_MemRead = 0;
                        ID_EX_MemWrite = 0;
                        ID_EX_RegWrite = 0;
                        ID_EX_MemtoReg = 0;
                        ID_EX_BranchEq = 0;
                ID_EX_BranchGt = 0;
        end
    else if(clk)
        begin
            ID_EX_instruction = IF_ID_instruction;
                        ID_EX_rd = IF_ID_rd;
                        ID_EX_rs2 = IF_ID_rs2;
                        ID_EX_rs1 = IF_ID_rs1;
                        ID_EX_imm_data = IF_ID_imm_data;
                        ID_EX_ReadData2 = IF_ID_ReadData2;
                        ID_EX_ReadData1 = IF_ID_ReadData1;
                        ID_EX_PC_Out = IF_ID_PC_Out;
                        ID_EX_ALUOp = IF_ID_ALUOp;
            ID_EX_ALUSrc = IF_ID_ALUSrc;
                        ID_EX_MemRead = IF_ID_MemRead;
                        ID_EX_MemWrite = IF_ID_MemWrite;
```

```
                        ID_EX_RegWrite = IF_ID_RegWrite;
                        ID_EX_MemtoReg = IF_ID_MemtoReg;
                        ID_EX_BranchEq = IF_ID_BranchEq;
                ID_EX_BranchGt = IF_ID_BranchGt;
        end
end
endmodule
```

Stage 3 - Execution

This is the part where all the calculations happen. This stage is responsible for two main tasks:

- If the instruction is a branch instruction, then the offset value to be added in order to find the address of the next location is calculated by adder.
- The ALU resides here, so all the operations are executed here.

Now from the Instruction Decode register we got the ALUop, which is the control line for the ALU, and then the value that is to be send to the registers is controlled by the two MUX, but we'll discuss them when we discuss the Forwarding Unit and how the data hazards are handled. For now we can focus on what exactly is the Execution stage doing. Once all the values are calculated, we send the control lines we got from the ID/EX stage and the values we calculated to the Memory stage, through EX/MEM stage.

```

  Adder branch(ID_EX_PC_Out,ID_EX_imm_data << 1, adder2out);

  ALU_Control alu1( ID_EX_ALUop,ID_EX_Inst,ID_EX_OP);



  EX_MEM reg3 (clk, reset,
              ID_EX_rd, ID_EX_mux_ForwardB, ID_EX_ALU, ID_EX_ALUzero, adder2out,
    ID_EX_Branch, ID_EX_MemRead, ID_EX_MemWrite, ID_EX_RegWrite, ID_EX_MemtoReg,
              EX_MEM_rd, EX_MEM_mux_ForwardB, EX_MEM_mux_Alu, EX_MEM_ALUzero,
    EX_MEM_adder2out, EX_MEM_Branch, EX_MEM_MemRead, EX_MEM_MemWrite, EX_MEM_RegWrite
    , EX_MEM_MemtoReg);
```

The ID/EX muxForwardA and ForwardB are the outputs from the mux whose control line is handled by the Forwarding unit, which we would further explain while explaining the forwarding unit. On the next cycle this is forwarded to the Memory stage through the following register.

```
module EX_MEM(
        input clk, reset,
        input [4:0] ID_EX_rd,
        input [63:0] ID_EX_mux_ForwardB, ID_EX_mux_ALU,
        input ID_EX_ALUzero,
        input [63:0] adder2out,
        input ID_EX_Branch,
        input ID_EX_MemRead,
        input ID_EX_MemWrite,
        input ID_EX_RegWrite,
        input ID_EX_MemtoReg,

        output reg [4:0] EX_MEM_rd,
    output reg [63:0] EX_MEM_mux_ForwardB, EX_MEM_mux_ALU,
        output reg EX_MEM_ALUzero,
        output reg [63:0] EX_MEM_adder2out,
        output reg EX_MEM_Branch,
        output reg EX_MEM_MemRead,
        output reg EX_MEM_MemWrite,
        output reg EX_MEM_RegWrite,
        output reg EX_MEM_MemtoReg

);

```

```
  always @ (posedge clk or reset)
    begin
      if (reset)
        begin
          EX_MEM_rd = 0;
          EX_MEM_mux_ForwardB = 0;
          EX_MEM_ALUzero = 0;
          EX_MEM_mux_ALU = 0;
          EX_MEM_adder2out =0;
          EX_MEM_Branch = 0;
          EX_MEM_MemRead =  0;
          EX_MEM_MemWrite = 0;
          EX_MEM_RegWrite = 0;
          EX_MEM_MemtoReg = 0;
        end
      else if (clk)
        begin
          EX_MEM_rd = ID_EX_rd;
          EX_MEM_mux_ForwardB = ID_EX_mux_ForwardB;
          EX_MEM_mux_ALU = ID_EX_mux_ALU;
          EX_MEM_ALUzero = ID_EX_ALUzero;
          EX_MEM_adder2out = adder2out;
          EX_MEM_Branch = ID_EX_Branch;
          EX_MEM_MemRead = ID_EX_MemRead ;
          EX_MEM_MemWrite = ID_EX_MemWrite ;
          EX_MEM_RegWrite = ID_EX_RegWrite;
          EX_MEM_MemtoReg = ID_EX_MemtoReg;
        end
    end
endmodule
```

Stage 4 - Memory Access

This stage has the only module which is Data Memory, but this register is also used to send back some signals, so it checks if the MemRead or MemWrite are high and accordingly performs the operation and sets the control lines to write data or read from the memory. This also sends back the contents of the registers to the Execution stage for calculations in order to handle data hazards. The main purpose of this register is to Write Data to the memory if the MemWrite signal is high, and read from the memory from the given register if the MemRead signal is high. This sends the contents of the registers and further control signals to the last stage of the pipeline through the MEM/WB register, which we'll also see below,

```
  Data_Memory dm1(clk, EX_MEM_mux_Alu, EX_MEM_mux_ForwardB,  EX_MEM_MemWrite,
    EX_MEM_MemRead, EX_MEM_readData,element1,element2,element3,element4,element5);

  MEM_WB reg4 (clk, reset,
               EX_MEM_rd, EX_MEM_mux_Alu, EX_MEM_readData, EX_MEM_RegWrite,
    EX_MEM_MemtoReg,
              MEM_WB_rd, MEM_WB_mux_ALU, MEM_WB_readData, MEM_WB_RegWrite,
    MEM_WB_MemtoReg);

  MUX xd (MEM_WB_mux_ALU, MEM_WB_read
```

```
module MEM_WB(
    input clk, reset,
        input [4:0] EX_MEM_rd,
        input [63:0] EX_MEM_mux_ALU,
    input [63:0] EX_MEM_readData,
        input EX_MEM_RegWrite,
        input EX_MEM_MemtoReg,

        output reg [4:0] MEM_WB_rd,
    output reg [63:0] MEM_WB_mux_ALU,
        output reg [63:0] MEM_WB_readData,
        output reg MEM_WB_RegWrite,
```

```
        output reg MEM_WB_MemtoReg

);

  always @ (posedge clk or posedge reset)
    begin
      if (reset)
        begin
          MEM_WB_rd = 0;
         // MEM_WB_mux_ALU = 0;
                MEM_WB_readData = 0;
            MEM_WB_RegWrite = 0;
                MEM_WB_MemtoReg = 0;
        end
      else if (clk)
        begin
          MEM_WB_rd <= EX_MEM_rd;
          MEM_WB_mux_ALU <= EX_MEM_mux_ALU;
                MEM_WB_readData <= EX_MEM_readData;
            MEM_WB_RegWrite <= EX_MEM_RegWrite;
                MEM_WB_MemtoReg <= EX_MEM_MemtoReg;
        end

    end

endmodule
```

Stage 5 - Write Back

This is the final stage of the processor, the right most part of the figure, it recieves following contents from the MEM/WB register

```
     wire [4:0] MEM_WB_rd;
     wire [63:0] MEM_WB_mux_ALU;
     wire [63:0] MEM_WB_readData;
     wire MEM_WB_RegWrite;
     wire MEM_WB_MemtoReg;
```

MemtoReg and RegWrite are the two control lines, MemtoReg decides between sending the ALU result or the register value and RegWrite, writes the chosen value to the register. This then is sent back to the ID stage to the registerfile so the contents of registers can be written back.

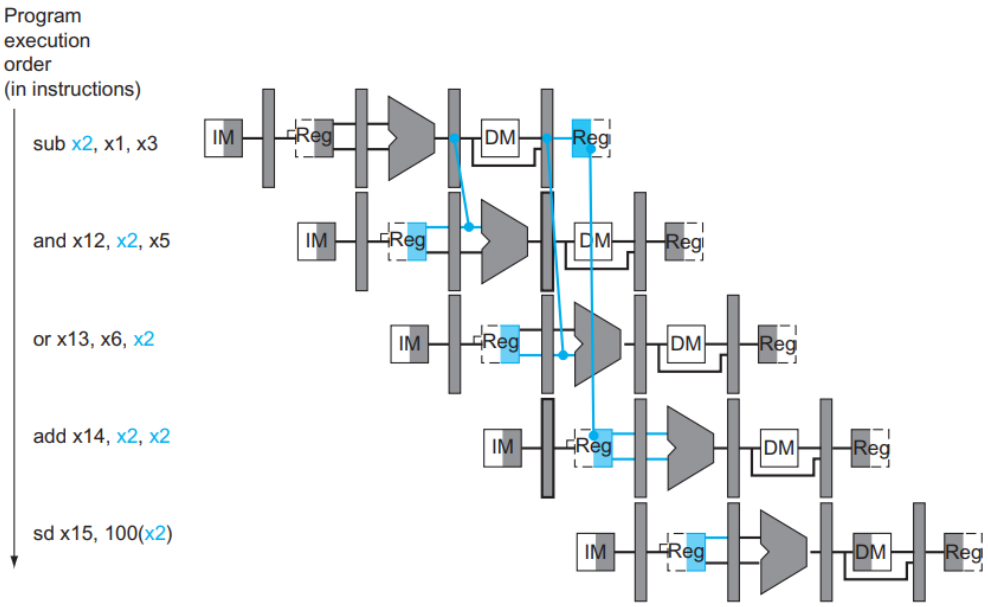
Forwarding Unit

Let's say we have to run the following set of instructions on the pipelined version of the processor we just made.

```
    add x1, x2, x3
    add x4, x1, x2
```

Now the first instruction would be executed just fine by our processor, but let's try to analyze the second instruction, so when the first instruction would be in the execution stage, the second instruction would be in the Instruction decoding stage, and as we have seen, this stage is also responsible for reading the values of the register, so while reading the values stored in the register, the value in x1 for the second instruction should be the sum of values in x2 and x3, and even though their sum has been calculated our first instruction hasn't reached the write back stage, so there won't be any value stored in x1, at least not the correct value that we want. this means that the processor would read an incorrect value from x1 and the result in x4 would be incorrect.

This is what we call a data hazard. In order to overcome this, we have techniques such as stalling and forwarding. From the two, the latter is the most efficient and is exactly what we implement in our processor. So what forwarding does is that now once the value is calculated in the execution stage and is need in the ID stage, it sends the value directly instead of waiting for it to load it in the register and then reading from the register.
A visual representation of how forwarding is executed is shown in the figure below



In the example above, the blue lines represent the forwarding, in which we can see that how from different stages the data is send to the values being used when they are not already stored in the register.

```
module Forwarding(
  input [4:0] EX_MEM_rd,
  input  EX_MEM_RegWrite, //WB of EX_MEM
  input [4:0] MEM_WB_rd,
  input  MEM_WB_RegWrite, //WB of MEM_WB
  input [4:0] ID_EX_rs1,
  input [4:0] ID_EX_rs2,
  output reg [1:0] forwardA,
  output reg [1:0] forwardB
);



  always @(*)
    begin
      //For ForwardA

      if ( (EX_MEM_RegWrite) && (EX_MEM_rd == ID_EX_rs1) && (EX_MEM_rd != 0) )
        forwardA = 2'b10;

      else if ( (MEM_WB_RegWrite) && (MEM_WB_rd == ID_EX_rs1) && (MEM_WB_rd != 0) && !((
    EX_MEM_RegWrite) && (EX_MEM_rd == ID_EX_rs1) && (EX_MEM_rd != 0)) )
      forwardA =  2'b01;

      else
        forwardA = 2'b00;


      //For ForwardB

      if ( (EX_MEM_RegWrite) && (EX_MEM_rd == ID_EX_rs2) && (EX_MEM_rd != 0) )
        forwardB = 2'b10;

      else if ( (MEM_WB_RegWrite) && (MEM_WB_rd == ID_EX_rs2) && (MEM_WB_rd != 0) && !((
    EX_MEM_RegWrite) && (EX_MEM_rd == ID_EX_rs2) && (EX_MEM_rd != 0)) )
      forwardB =  2'b01;

      else
        forwardB = 2'b00;

    end
endmodule
```

For Forwarding, there are three cases to consider. The first one is EX hazard, This forwards the result from the previous instruction to either input of the ALU. If the previous instruction is going to write to the register file, and the write register number is equal to the read register number of ALU inputs A or B, then the multiplexor will pick the value from register EX/MEM.

The next case is Data hazard,As mentioned above,sometimes the result is directly needed from MEM stage because at times, the result is stored several times in a particular register, hence to get the most recent one, we take it directly MEM Stage. In our code, the multiplexers values are generated according to following table:

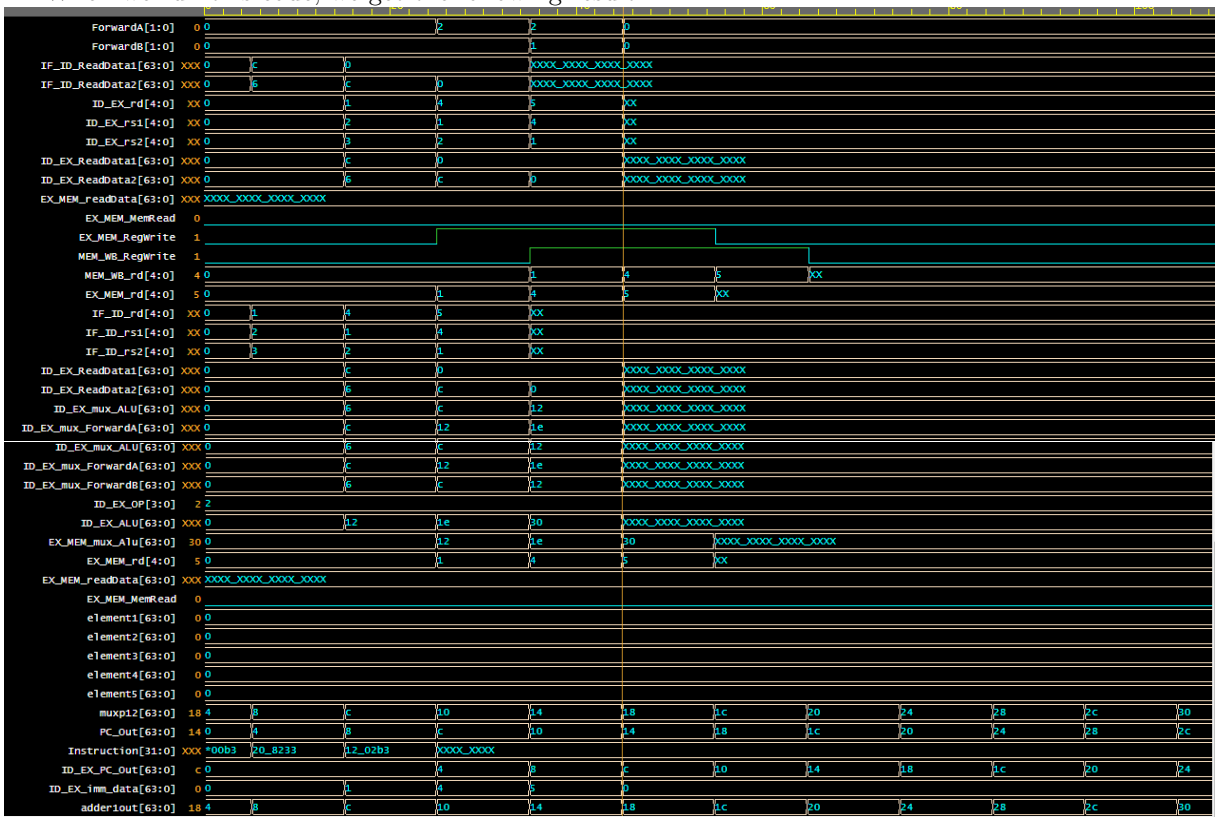
Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Results

Now lets see how this all comes together, so if we run the following code with data hazards, our forwarding unit should be able to handle it and pipelines should work fin, so lets test is

```
      add x1, x2, x3
      add x4, x1, x2
      add x5, x4, x1
```

When we run this code, we get the following result:



Task 3 - Flushing And Stalling

When an instruction tries to read a register following a load instruction that writes the same register, it can't be solved through forwarding. In addition to a forwarding unit, we need a hazard detection unit. It operates during the ID stage so that it can insert the stall between the load and the instruction dependent on it. Following is the code for Hazard Detection Unit.

```
module Hazard
  (input [4:0] IF_ID_rs1,
   input [4:0] IF_ID_rs2,
   input [4:0] ID_EX_rd,
```

```
   input ID_EX_MemRead,
   output reg IF_ID_Write,
   output reg PC_Write,


   output reg stall
);


  always @ (*)
    begin
      if ( (ID_EX_MemRead) && ((ID_EX_rd == IF_ID_rs1) || (ID_EX_rd == IF_ID_rs2)))
        begin
        stall = 1;
        PC_Write=0;
        IF_ID_Write=0;
        end

      else
        begin
        stall = 0;

        PC_Write=1;
        IF_ID_Write=1;
        end

    end

endmodule
```

In both load and R-type instructions,we use the registerRd to refer the register specified instruction bits 11:7. Since the only instruction that reads data memory is a load, The first line tests to see if the instruction is a load. The following two lines check to see if the destination register field of the load in the EX stage is equal to either source register of the instruction in the ID stage. If they are equal, the instruction stalls one clock cycle. Since stall occurs, it prevents increment in values of address and does not write any registers in IF/ID, hence the two become 0 and stall becomes 1. because if the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, we would lose the fetched instruction. The registers in the ID stage will continue to be read using the same instruction fields in the IF/ID pipeline register.

Inserting NOPs: we set all seven control signals in the EX, MEM, and WB stages will create a "do nothing" or nop instruction. By identifying the hazard in the ID stage, we can insert a bubble into the pipeline by changing the EX, MEM, and WB control fields of the ID/EX pipeline register to 0. These benign control values are percolated forward at each clock cycle with the proper effect: no registers or memories are written if the control values are all 0.)

Flushing: Although, stalling is a good option for Conditional Hazards but it might take more time if we stall on conditional branches. Flushing discards contents of pipeline registers when needed by using just one control signal which tells when flush is needed, when that Flush is high, we make outputs of the pipelined registers 0.Flush is needed Whenever it the branch is true. This is detected after EX/MEM stage, so before this is detected, the pipeline registers already have fetched other instructions in sequence, but these are not needed now as branch is to be taken. Branch is true is detected through AND of two control lines, Branch and Aluzero and was given input as flush to IF/ID, ID/EX and EX/Mem pipeline registers.

Following part of code shows flushing of ID/EX register if flush is true.

```
      if (reset==1'b1 ||ID_IX_Flush)
        begin
           ID_EX_Inst=0;
            ID_EX_rs1=0;
            ID_EX_rs2=0;
            ID_EX_rd = 0;
            ID_EX_imm_data=0;
            ID_EX_ReadData2=0;
            ID_EX_ReadData1=0;
            ID_EX_PC_Out=0;
            ID_EX_ALUsrc=0;
            ID_EX_ALUop=0;
            ID_EX_Branch=0;
            ID_EX_MemRead=0;
            ID_EX_MemWrite=0;
            ID_EX_RegWrite=0;
            ID_EX_MemtoReg=0;

        end
```

1 Final Comments

We had difficulties in executing the third task, although we tried our utmost to execute bubble sort and gave our time and efforts to this but we were not able to execute it. It was an interesting project and we are eager to learn about our mistakes.Thank you.