CE-321L/CS-330L: Computer Architecture

Project Report

Building a 5-stage pipelined processor capable of executing any one array sorting algorithm

Huzaifa Riaz - 07741
Humayun Hasan - 06527
Mohammed Haider Abbas - 06418

27-April-2023

Introduction:

The primary objective of this project is to build a 5-stage pipelined processor capable of executing any one array sorting algorithm. This project aims to convert an existing lab 11 single-cycle processor architecture to a pipelined one with minor modifications to the circuit. The project involves implementing the bgt or blt instruction to swap two values. The project is divided into three parts, and the recommended course of action involves choosing a suitable sorting algorithm, converting its pseudocode to assembly using the RISC-V instruction set, and testing its working on the assembly code simulator. In part two, the existing processor is modified to make it a pipelined one (5 stages), and each instruction is tested separately to ensure that the pipelined processor version can execute one instruction correctly in isolation. In part three, circuitry is introduced to detect hazards (data, control, and structural if needed) and handle them in hardware using forwarding, stalling, and flushing the pipeline. By completing all three parts of this project, the array sorting code should be able to function correctly on the pipelined processor architecture.

Task 1 - Single Cycle Processor With Bubble Sort

Sort Used – Bubble sort:

C code for bubble sort

```
void bubbleSort(int arr[], int n) {
   int i, j;
   for (i = 0; i < n-1; i++) {
      for (j = 0; j < n-i-1; j++) {
         if (arr[j] > arr[j+1]) {
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
```

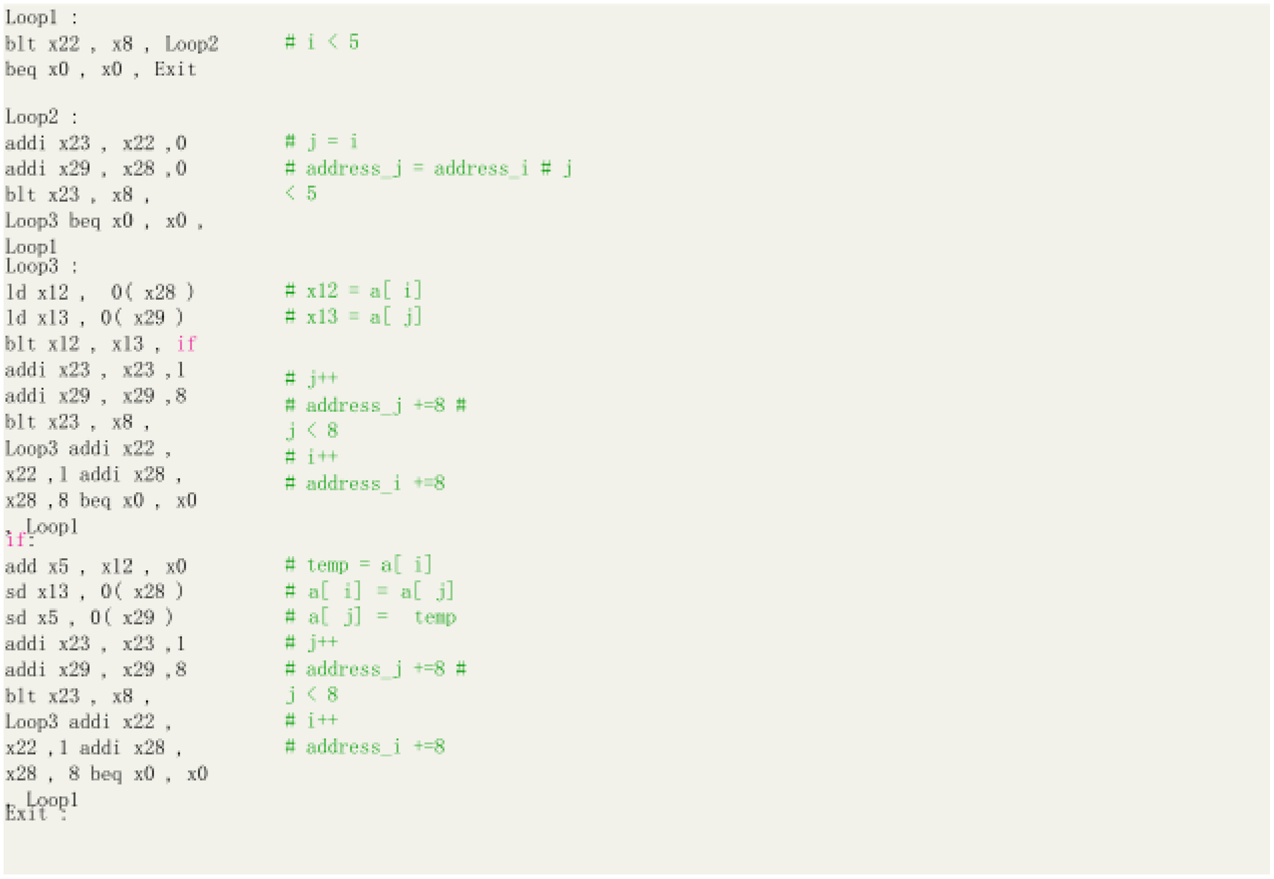
We used bubble sort as our choice of sort mainly because of two reasons.

1. Simplicity: Bubble sort is one of the simplest sorting algorithms to understand and implement. It only requires a few lines of code, making it easy to implement and debug.

2. Small data sets: Bubble sort has a time complexity of O(n^2), which means that for small data sets, bubble sort can be faster than more complex algorithms because it has a small constant factor.

Bubble Sort Assembly Code

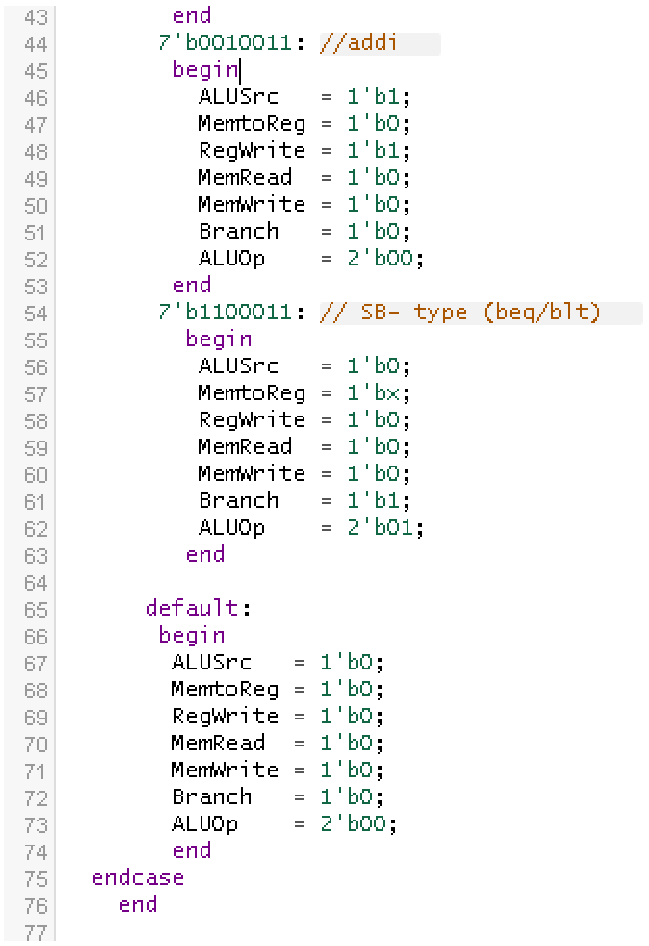
x22 = *i*, x23 = *j*, x28 = address_*i*, x29 = address_*j*, x5 = temp, x8 = 5



```
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 # j
blt x23 , x8 ,              < 5
Loop3 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 ,              j < 8
Loop3 addi x22 ,            # i++
x22 ,1 addi x28 ,           # address_i +=8
x28 ,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 ,              j < 8
Loop3 addi x22 ,            # i++
x22 ,1 addi x28 ,           # address_i +=8
x28 , 8 beq x0 , x0
, Loop1
Exit :
```

Changes to Lab 11 – Single Cycle Processor For blt Instruction

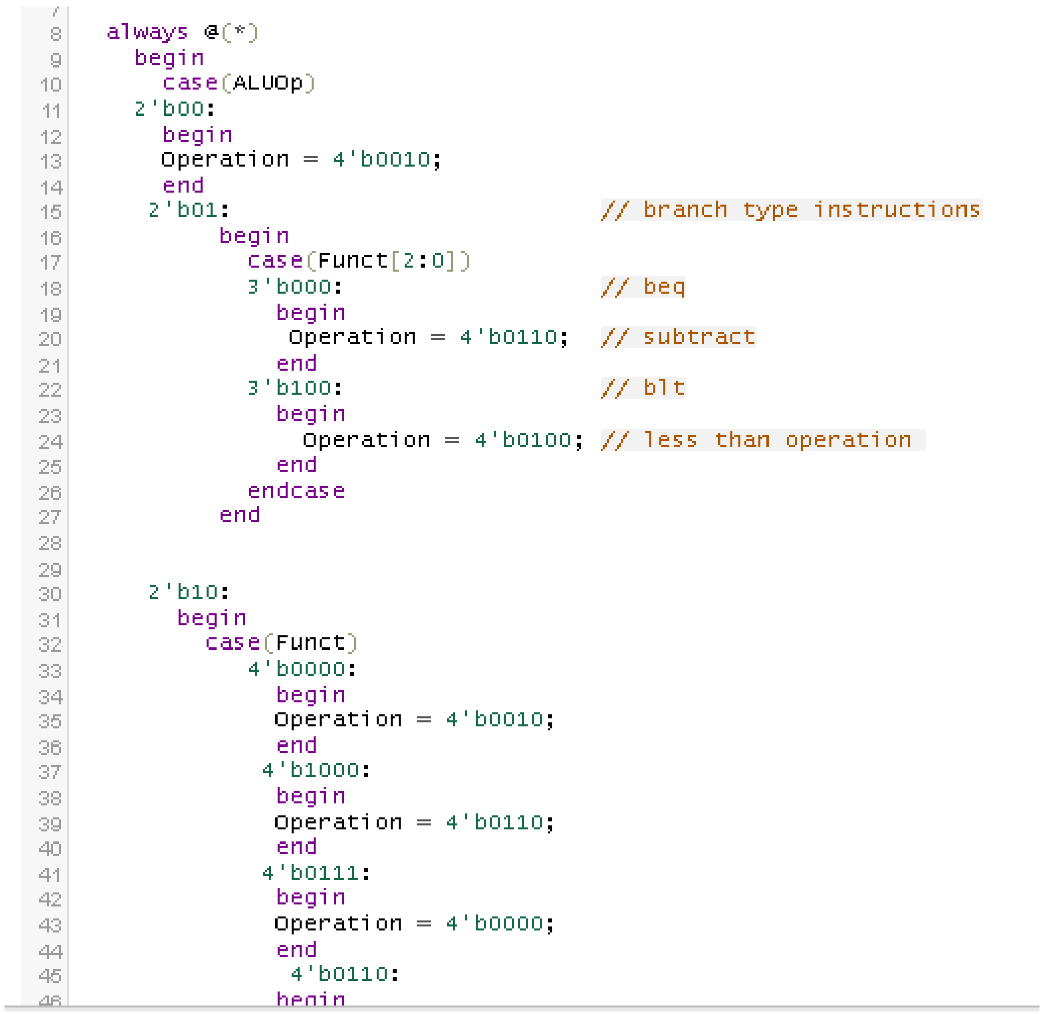
Control Unit:



```
      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;
      end
endcase
  end
```

- SB-type (beq/blt)

ALU Control:



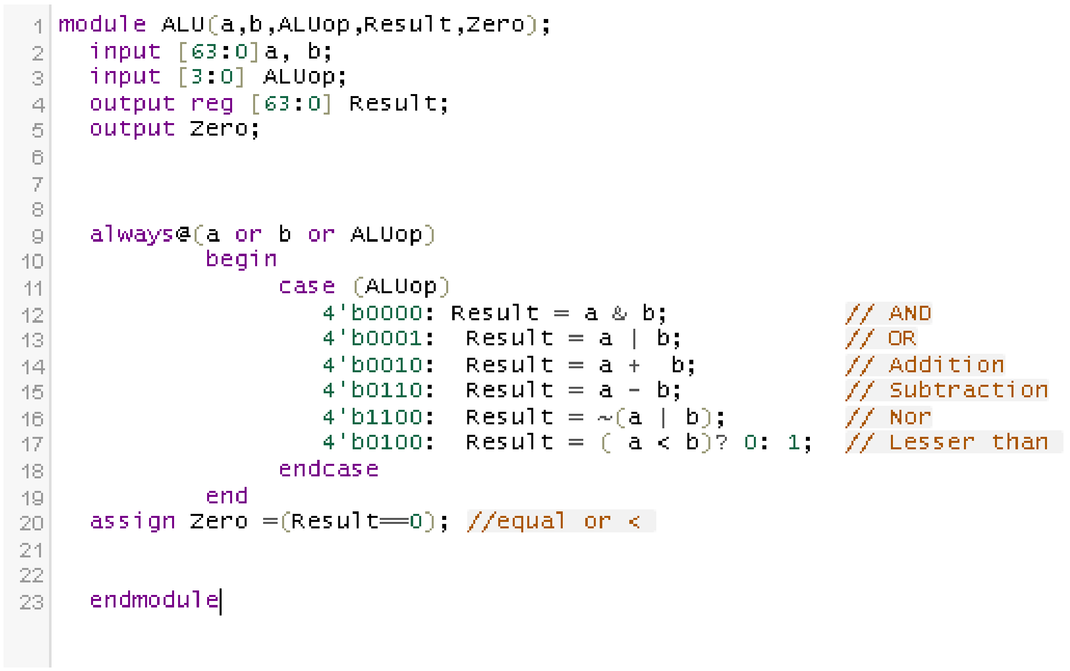
```verilog
always @(*)
  begin
    case(ALUOp)
  2'b00:
    begin
    Operation = 4'b0010;
    end
   2'b01:                          // branch type instructions
        begin
          case(Funct[2:0])
          3'b000:                  // beq
            begin
             Operation = 4'b0110;  // subtract
            end
          3'b100:                  // blt
            begin
              Operation = 4'b0100; // less than operation
            end
          endcase
        end


   2'b10:
     begin
       case(Funct)
          4'b0000:
            begin
            Operation = 4'b0010;
            end
           4'b1000:
            begin
            Operation = 4'b0110;
            end
           4'b0111:
            begin
            Operation = 4'b0000;
            end
             4'b0110:
            begin
```

- 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:

1. 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.
2. 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.

<u>ALU:</u>



```verilog
module ALU(a,b,ALUop,Result,Zero);
  input [63:0]a, b;
  input [3:0] ALUop;
  output reg [63:0] Result;
  output Zero;



  always@(a or b or ALUop)
          begin
               case (ALUop)
                  4'b0000: Result = a & b;            // AND
                  4'b0001:  Result = a | b;           // OR
                  4'b0010:  Result = a +  b;          // Addition
                  4'b0110:  Result = a - b;           // Subtraction
                  4'b1100:  Result = ~(a | b);        // Nor
                  4'b0100:  Result = ( a < b)? 0: 1;  // Lesser than
               endcase
          end
  assign Zero =(Result==0); //equal or <


  endmodule
```

- Modified to add lesser than
- 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.

Changes To Lab 11 - For Running Bubble Sort

Instruction Memory:

Major changes were made in Instruction memory where each instruction of assembly code was converted into 32 bit binary address, broken down and passed to InstMemory as 8 bits.

```
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;
```

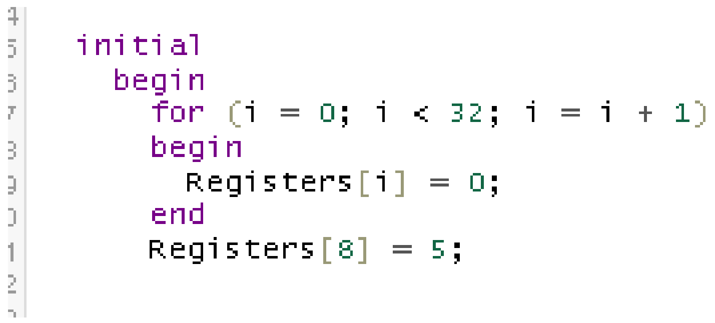
```
//ld x13, 0(x29) - x13 = a[j]
InstMemory[28]=8'b10000011;
InstMemory[29]=8'b10110110;
InstMemory[30]=8'b00001110;
InstMemory[31]=8'b00000000;

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

//addi x23,x23,1 - j++
InstMemory[36]=8'b10010011;
InstMemory[37]=8'b10001011;
InstMemory[38]=8'b00011011;
InstMemory[39]=8'b00000000;

//addi x29,x29,8 - locj+= 8
InstMemory[40]=8'b10010011;
InstMemory[41]=8'b10001110;
InstMemory[42]=8'b10001110;
InstMemory[43]=8'b00000000;

//blt x23,x8,Loop3
InstMemory[44]=8'b11100011;
InstMemory[45]=8'b11000110;
InstMemory[46]=8'b10001011;
InstMemory[47]=8'b11111110;
```

```
//addi x22,x22,1 #i++
InstMemory[48]=8'b00010011;
InstMemory[49]=8'b00001011;
InstMemory[50]=8'b00011011;
InstMemory[51]=8'b00000000;

//addi x28, x28,8 - loci+=8
InstMemory[52]=8'b00010011;
InstMemory[53]=8'b00001110;
InstMemory[54]=8'b10001110;
InstMemory[55]=8'b0;

//beq x0,x0,Loop1
InstMemory[56]=8'b11100011;
InstMemory[57]=8'b00000100;
InstMemory[58]=8'b00000000;
InstMemory[59]=8'b11111100;

        //add x5,x12,x0 # temp = a[i]
InstMemory[60]=8'b10110011;
InstMemory[61]=8'b00000010;
InstMemory[62]=8'b00000110;
InstMemory[63]=8'b0;

        //sd x13, 0(x28) #a[i] = a[j]
InstMemory[64]=8'b00100011;
InstMemory[65]=8'b00110000;
InstMemory[66]=8'b11011110;
InstMemory[67]=8'b0;

//sd x5, 0(x29) -a[j] = temp
InstMemory[68]=8'b00100011;
InstMemory[69]=8'b10110000;
InstMemory[70]=8'b01011110;
```

```
InstMemory[71]=8'b0;


        //addi x23,x23,1 - j++
InstMemory[72]=8'b10010011;
InstMemory[73]=8'b10001011;
InstMemory[74]=8'b00011011;
InstMemory[75]=8'b0;

        //addi x29,x29,8 - locj+= 8
InstMemory[76]=8'b10010011;
InstMemory[77]=8'b10001110;
InstMemory[78]=8'b10001110;
InstMemory[79]=8'b00000000;

        //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
```

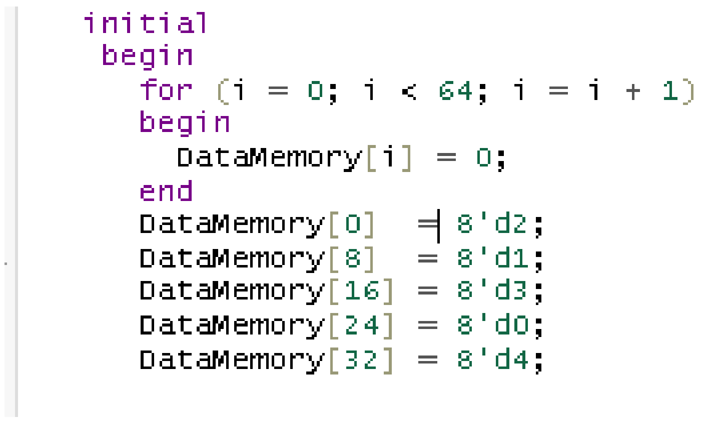
Register Files:



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

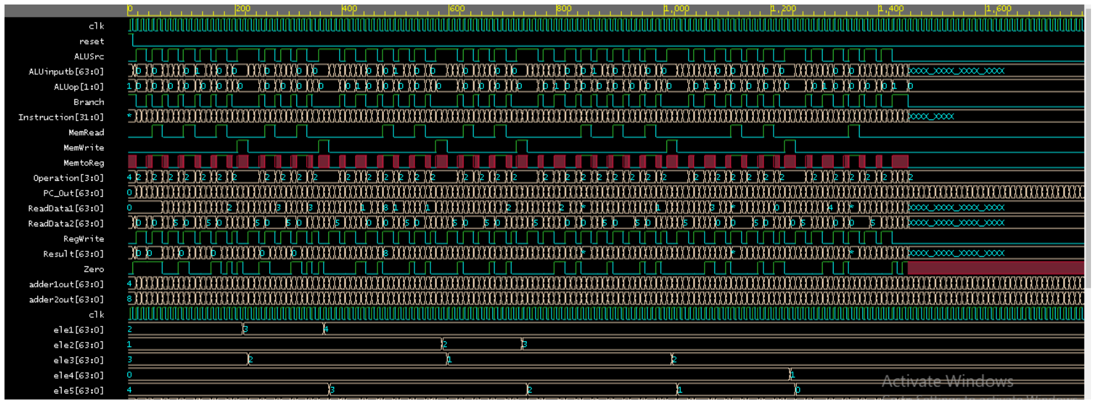
- Assigned the 8th register (x8) the value 5 ie number the elements we'll be sorting in bubble sort and all the other registers are assigned the value 0.

Data Memory:

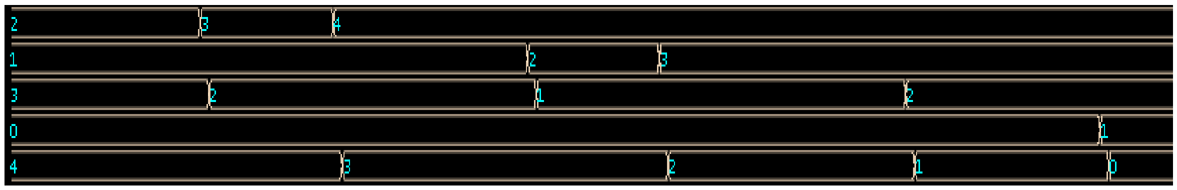


```
initial
 begin
   for (i = 0; i < 64; i = i + 1)
   begin
     DataMemory[i] = 0;
   end
   DataMemory[0]  = 8'd2;
   DataMemory[8]  = 8'd1;
   DataMemory[16] = 8'd3;
   DataMemory[24] = 8'd0;
   DataMemory[32] = 8'd4;
```

- Passed 2,1,3,0,4 in the data memory for sorting to be applied on them
- If the sorting works properly it should be sorted as 4,3,2,1,0



Full wave



Sorting on single cycle processor in working condition

EDA link for viewing detailed wave = https://www.edaplayground.com/x/wdjz

Task 2 - Introducing Pipelining

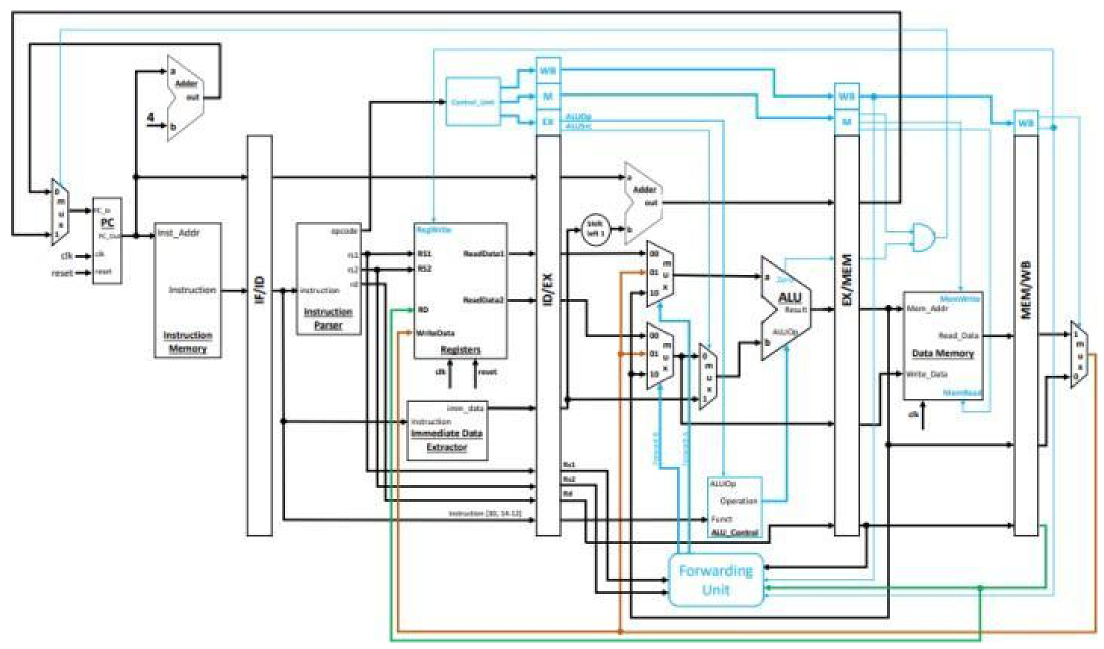
The first part of the discussion presented a single cycle processor capable of running Bubble sort. However, this type of implementation executes one instruction at a time, leading to inefficiency and wasted processing power. To address this, pipelining is introduced to enable the execution of multiple instructions simultaneously, depending on the number of stages in the pipeline. In a five-stage pipeline, different components of the processor work on different parts of multiple instructions at the same time. This allows the processor to work on up to five instructions simultaneously, increasing efficiency and reducing wasted processing power.

The processor implemented is divided in the following 5 stages:

- **IF :** Instruction Fetch
- **ID :** Instruction decode
- **EX :** Execution
- **MEM :** Data memory
- **WB :** Write back

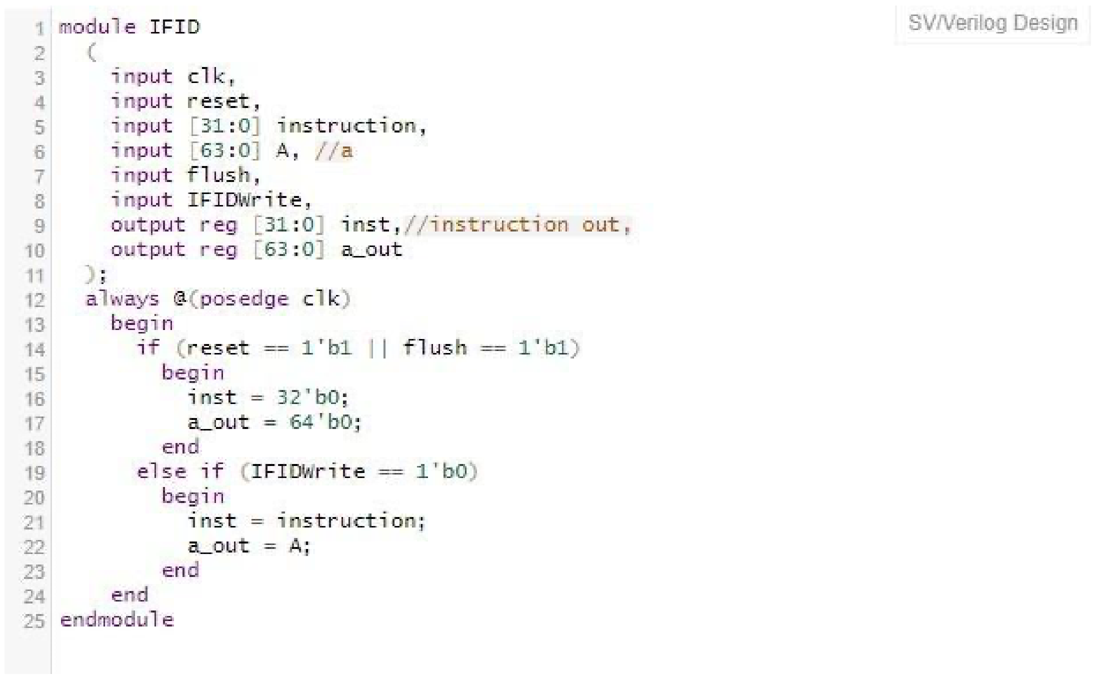
Prior to diving into the specifics of each stage, its function, and discussing the Verilog implementation, let's take a look at the overall concept that we are aiming to implement.

This can be visualized through the following figure:



Stage 1 - Instruction Fetch (IF):

It is the first stage of our processor, which is located on the left side of the figure. It is responsible for fetching the instruction from memory. It accomplishes this by calculating the instruction's address using the program counter (PC) and accessing the instruction memory module. The fetched instruction is then sent to the next stage through the IF/ID register. Additionally, this stage handles jump addresses in the event that the previous instruction was a branch instruction. It does so by receiving the offset from the EX/MEM register.



```
module IFID
  (
    input clk,
    input reset,
    input [31:0] instruction,
    input [63:0] A, //a
    input flush,
    input IFIDWrite,
    output reg [31:0] inst,//instruction out,
    output reg [63:0] a_out
  );
  always @(posedge clk)
    begin
      if (reset == 1'b1 || flush == 1'b1)
        begin
          inst = 32'b0;
          a_out = 64'b0;
        end
      else if (IFIDWrite == 1'b0)
        begin
          inst = instruction;
          a_out = A;
        end
    end
endmodule
```

The outputs from this stage are the intermediate connections between the Instruction Fetch and the Instruction decode stage.

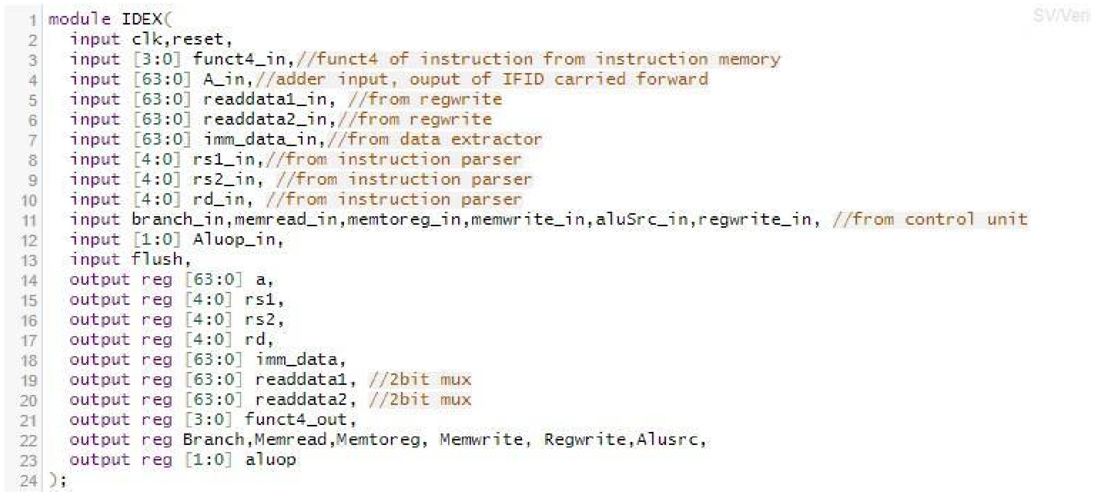
<u>Stage 2 - Instruction Decode (ID):</u>

In the pipeline, this stage follows the initial stage of instruction fetching and is responsible for decoding instructions and performing register reads or writes. It starts by receiving the instruction fetched by the IF stage and proceeds to extract the relevant information.

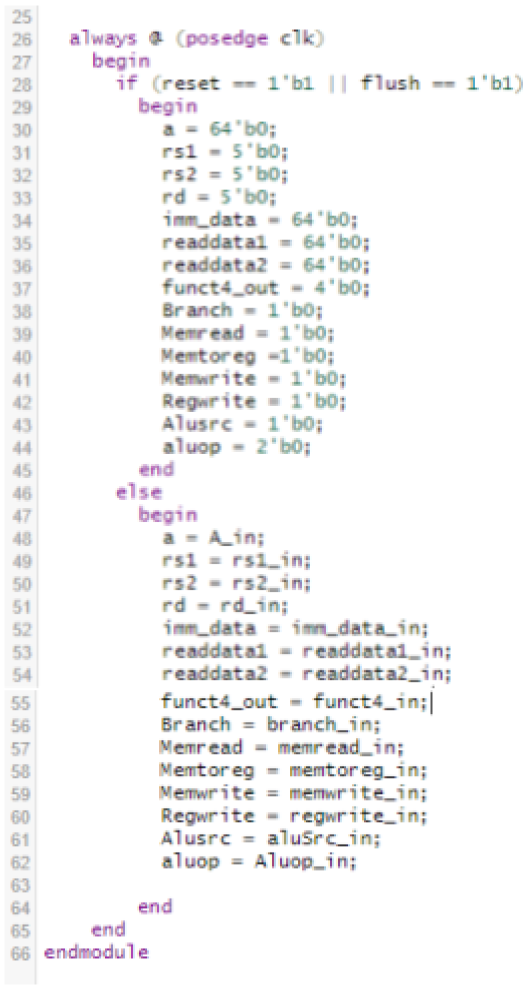
After the instruction is fetched in the initial stage, it is passed on to the second stage of our pipeline where it undergoes decoding using the Instruction Parser and Data Extractor module. This enables us to determine the opcode, rd, rs1, and rs2 of the 32-bit instruction.
Following this, the RegisterFile module reads the contents of the relevant registers or writes back to them as necessary, with signals from the MEM/WEB register used

to determine whether it is a read or write operation. The Control Unit plays a crucial role in this stage, utilizing the opcode to forward signals to control the ALU and other operations in the processor.

Throughout this process, the flow of the program is maintained as signals are forwarded from right to left. Finally, all intermediate wire contents from the IF/ID stage are transmitted to the ID/EX stage.



```
module IDEX(
  input clk,reset,
  input [3:0] funct4_in,//funct4 of instruction from instruction memory
  input [63:0] A_in,//adder input, ouput of IFID carried forward
  input [63:0] readdata1_in, //from regwrite
  input [63:0] readdata2_in,//from regwrite
  input [63:0] imm_data_in,//from data extractor
  input [4:0] rs1_in,//from instruction parser
  input [4:0] rs2_in, //from instruction parser
  input [4:0] rd_in, //from instruction parser
  input branch_in,memread_in,memtoreg_in,memwrite_in,aluSrc_in,regwrite_in, //from control unit
  input [1:0] Aluop_in,
  input flush,
  output reg [63:0] a,
  output reg [4:0] rs1,
  output reg [4:0] rs2,
  output reg [4:0] rd,
  output reg [63:0] imm_data,
  output reg [63:0] readdata1, //2bit mux
  output reg [63:0] readdata2, //2bit mux
  output reg [3:0] funct4_out,
  output reg Branch,Memread,Memtoreg, Memwrite, Regwrite,Alusrc,
  output reg [1:0] aluop
);
```



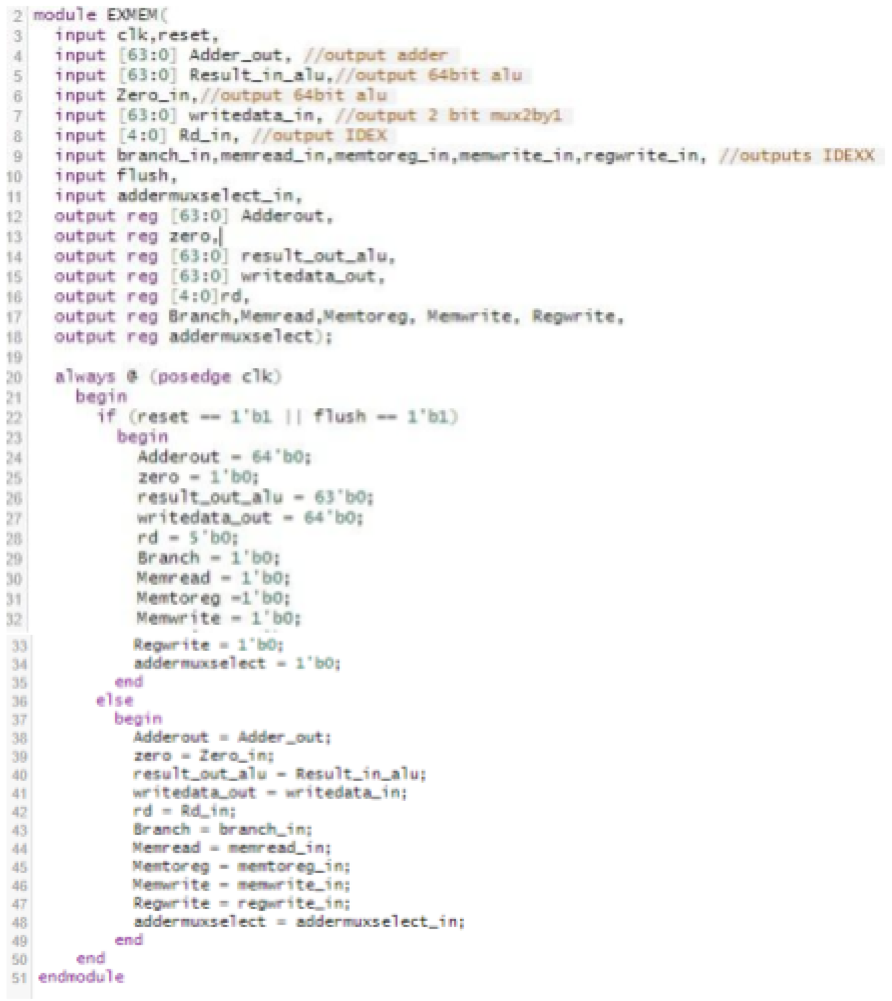
```

  always @ (posedge clk)
    begin
      if (reset == 1'b1 || flush == 1'b1)
        begin
          a = 64'b0;
          rs1 = 5'b0;
          rs2 = 5'b0;
          rd = 5'b0;
          imm_data = 64'b0;
          readdata1 = 64'b0;
          readdata2 = 64'b0;
          funct4_out = 4'b0;
          Branch = 1'b0;
          Memread = 1'b0;
          Memtoreg =1'b0;
          Memwrite = 1'b0;
          Regwrite = 1'b0;
          Alusrc = 1'b0;
          aluop = 2'b0;
        end
      else
        begin
          a = A_in;
          rs1 = rs1_in;
          rs2 = rs2_in;
          rd = rd_in;
          imm_data = imm_data_in;
          readdata1 = readdata1_in;
          readdata2 = readdata2_in;
          funct4_out = funct4_in;
          Branch = branch_in;
          Memread = memread_in;
          Memtoreg = memtoreg_in;
          Memwrite = memwrite_in;
          Regwrite = regwrite_in;
          Alusrc = aluSrc_in;
          aluop = Aluop_in;

        end
    end
endmodule
```

<u>Stage 3 - Execution:</u>

Here all the calculations take place. This stage is responsible for two main tasks:

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

After receiving the instruction from the IF stage, the Instruction Decode register provides the **'aluop'**, which serves as the control signal for the ALU. The data to be written to the registers is determined by two MUX, but we will cover this in our discussion of the Forwarding Unit and data hazard handling. For now, let's focus on the Execution stage and its role. Once all necessary calculations are completed, the Execution stage sends both the control signals obtained from the ID/EX stage and the calculated values to the Memory stage through the EX/MEM stage.

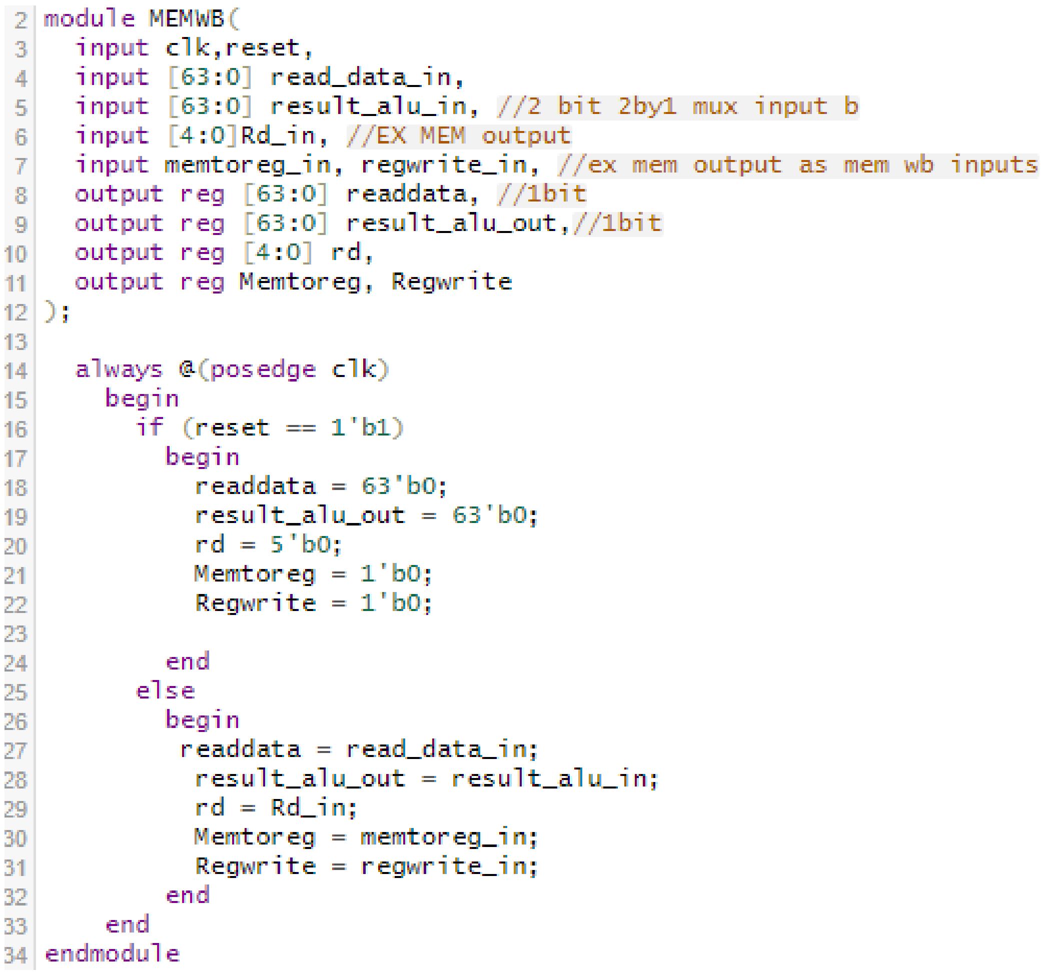


```verilog
module EXMEM(
  input clk,reset,
  input [63:0] Adder_out, //output adder
  input [63:0] Result_in_alu,//output 64bit alu
  input Zero_in,//output 64bit alu
  input [63:0] writedata_in, //output 2 bit mux2by1
  input [4:0] Rd_in, //output IDEX
  input branch_in,memread_in,mentoreg_in,memwrite_in,regwrite_in, //outputs IDEXX
  input flush,
  input addermuxselect_in,
  output reg [63:0] Adderout,
  output reg zero,
  output reg [63:0] result_out_alu,
  output reg [63:0] writedata_out,
  output reg [4:0]rd,
  output reg Branch,Memread,Mentoreg, Memwrite, Regwrite,
  output reg addermuxselect);

  always @ (posedge clk)
    begin
      if (reset == 1'b1 || flush == 1'b1)
        begin
          Adderout = 64'b0;
          zero = 1'b0;
          result_out_alu = 63'b0;
          writedata_out = 64'b0;
          rd = 5'b0;
          Branch = 1'b0;
          Memread = 1'b0;
          Mentoreg =1'b0;
          Memwrite = 1'b0;
          Regwrite = 1'b0;
          addermuxselect = 1'b0;
        end
      else
        begin
          Adderout = Adder_out;
          zero = Zero_in;
          result_out_alu = Result_in_alu;
          writedata_out = writedata_in;
          rd = Rd_in;
          Branch = branch_in;
          Memread = memread_in;
          Mentoreg = mentoreg_in;
          Memwrite = memwrite_in;
          Regwrite = regwrite_in;
          addermuxselect = addermuxselect_in;
        end
    end
endmodule
```

Stage 4 - Memory Access

Only Data Memory module is present in this stage, but this stage also sends signals back and forth. It checks whether the MemRead or MemWrite signals are high, and then performs the corresponding operation while setting the necessary control lines to write data to or read data from memory.

Additionally, it sends the register contents to the Execution stage for calculations to handle data hazards. The main objective of this register is to write data to memory if the MemWrite signal is high and read data from memory into the designated register if the MemRead signal is high. Finally, it transmits the contents of the registers and other control signals to the last stage of the pipeline through the MEM/WB register.



```
module MEMWB(
  input clk,reset,
  input [63:0] read_data_in,
  input [63:0] result_alu_in, //2 bit 2by1 mux input b
  input [4:0]Rd_in, //EX MEM output
  input memtoreg_in, regwrite_in, //ex mem output as mem wb inputs
  output reg [63:0] readdata, //1bit
  output reg [63:0] result_alu_out,//1bit
  output reg [4:0] rd,
  output reg Memtoreg, Regwrite
);

  always @(posedge clk)
    begin
      if (reset == 1'b1)
        begin
          readdata = 63'b0;
          result_alu_out = 63'b0;
          rd = 5'b0;
          Memtoreg = 1'b0;
          Regwrite = 1'b0;

        end
      else
        begin
         readdata = read_data_in;
          result_alu_out = result_alu_in;
          rd = Rd_in;
          Memtoreg = memtoreg_in;
          Regwrite = regwrite_in;
        end
    end
endmodule
```

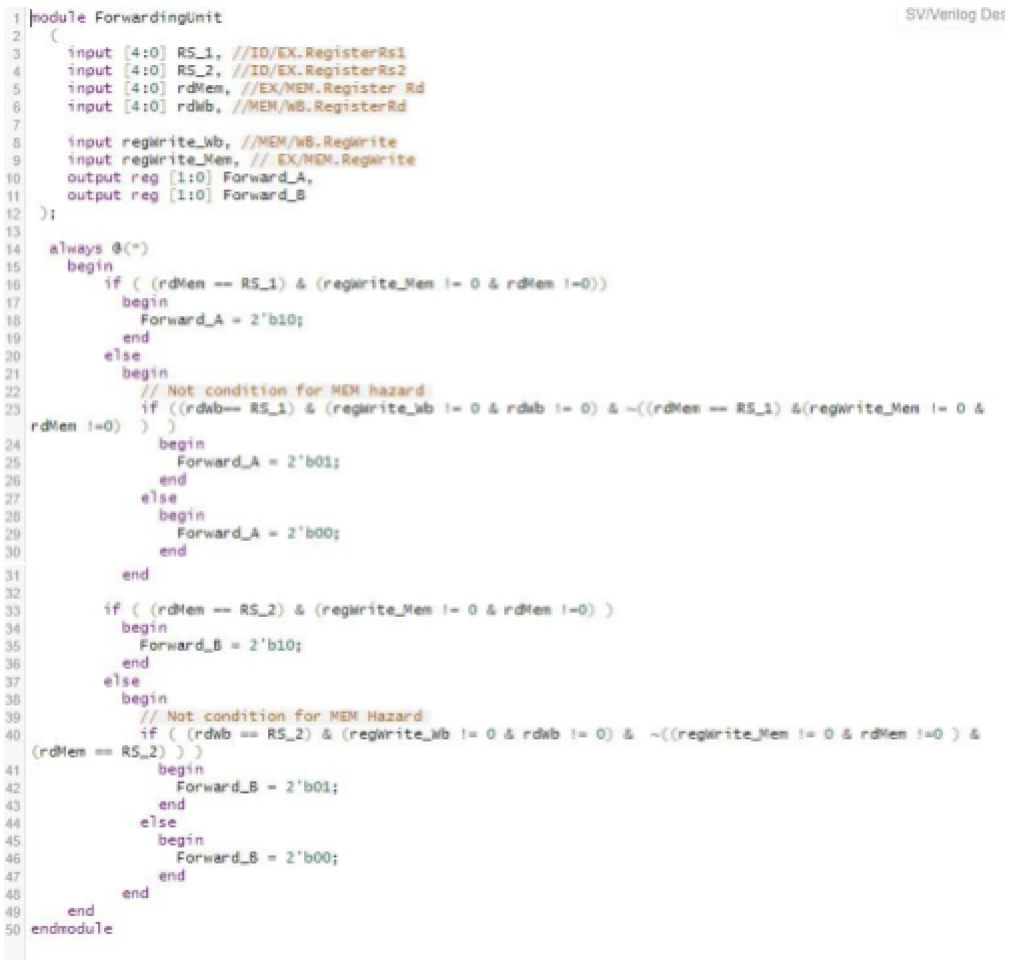
Stage 5 - Write Back

This is the final stage of the processor, the right most part of the figure.

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:

To address data hazards, we utilize techniques like stalling and forwarding. Of the two, forwarding is more effective and is the method we employ in our processor. Forwarding allows the value calculated in the execution stage to be directly sent to the ID stage when it's needed, without having to wait for it to be loaded into a register and then read from there.



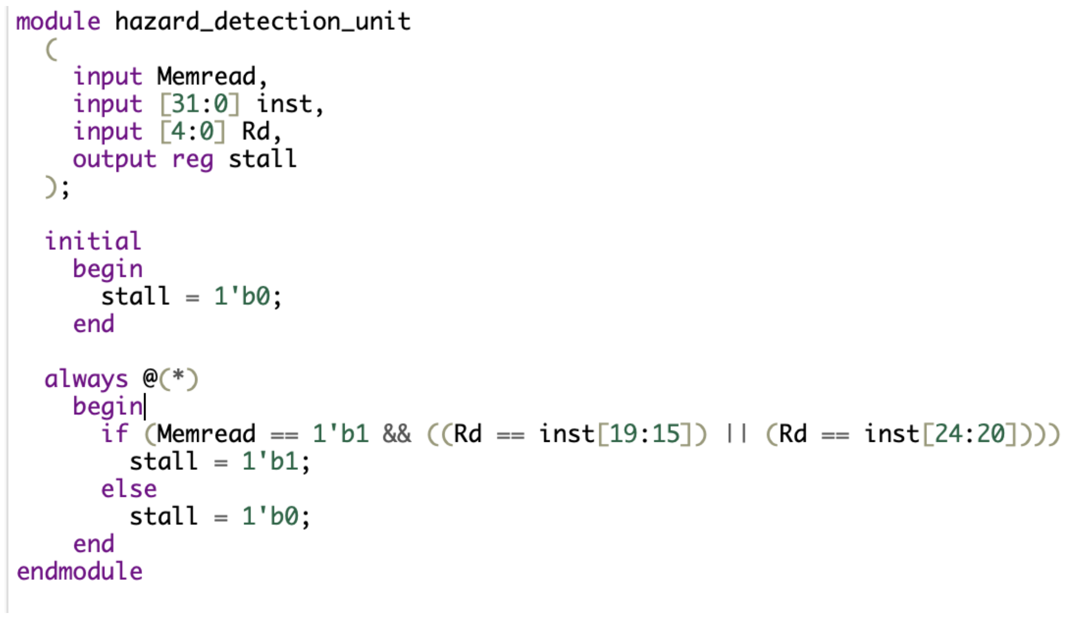
SV/Verilog Des

```
module ForwardingUnit
  (
    input [4:0] RS_1, //ID/EX.RegisterRs1
    input [4:0] RS_2, //ID/EX.RegisterRs2
    input [4:0] rdMem, //EX/MEM.Register Rd
    input [4:0] rdWb, //MEM/WB.RegisterRd

    input regWrite_Wb, //MEM/WB.RegWrite
    input regWrite_Mem, // EX/MEM.RegWrite
    output reg [1:0] Forward_A,
    output reg [1:0] Forward_B
 );

  always @(*)
    begin
        if ( (rdMem == RS_1) & (regWrite_Mem != 0 & rdMem !=0))
          begin
            Forward_A = 2'b10;
          end
        else
          begin
            // Not condition for MEM hazard
            if ((rdWb== RS_1) & (regWrite_Wb != 0 & rdWb != 0) & ~((rdMem == RS_1) &(regWrite_Mem != 0 &
rdMem !=0)  )  )
              begin
                Forward_A = 2'b01;
              end
            else
              begin
                Forward_A = 2'b00;
              end
          end

        if ( (rdMem == RS_2) & (regWrite_Mem != 0 & rdMem !=0) )
          begin
            Forward_B = 2'b10;
          end
        else
          begin
            // Not condition for MEM Hazard
            if ( (rdWb == RS_2) & (regWrite_Wb != 0 & rdWb != 0) &  ~((regWrite_Mem != 0 & rdMem !=0 ) &
(rdMem == RS_2) ) )
              begin
                Forward_B = 2'b01;
              end
            else
              begin
                Forward_B = 2'b00;
              end
          end
    end
endmodule
```

Task 3 - Flushing and Stalling

We have implemented Hazard detection, forwarding and flushing in our pipelined processor. Forwarding is used to pass data between instructions to avoid data hazards, stalling is used to resolve hazards that cannot be avoided through forwarding, and flushing is used to discard instructions in the pipeline when a hazard is detected to avoid stalls.

Hazard Detection:

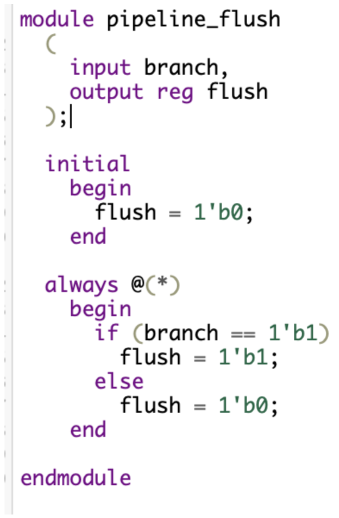


```
module hazard_detection_unit
  (
    input Memread,
    input [31:0] inst,
    input [4:0] Rd,
    output reg stall
  );

  initial
    begin
      stall = 1'b0;
    end

  always @(*)
    begin
      if (Memread == 1'b1 && ((Rd == inst[19:15]) || (Rd == inst[24:20])))
        stall = 1'b1;
      else
        stall = 1'b0;
    end
endmodule
```

This hazard detection unit detects RAW hazards by checking whether a memory read operation is being performed and whether the destination register for the current instruction matches the source register for a previous instruction. If a hazard is detected, the unit sets the stall signal to 1, indicating that the pipeline should be stalled until the hazard is resolved.

Flush:

This pipeline flush module detects branch instructions by checking whether the branch signal is 1. If a branch instruction is detected, the module sets the flush signal to 1, indicating that the pipeline should be flushed to prevent the execution of instructions that are no longer valid.



```
module pipeline_flush
  (
    input branch,
    output reg flush
  );

  initial
    begin
      flush = 1'b0;
    end

  always @(*)
    begin
      if (branch == 1'b1)
        flush = 1'b1;
      else
        flush = 1'b0;
    end

endmodule
```

Forwarding:

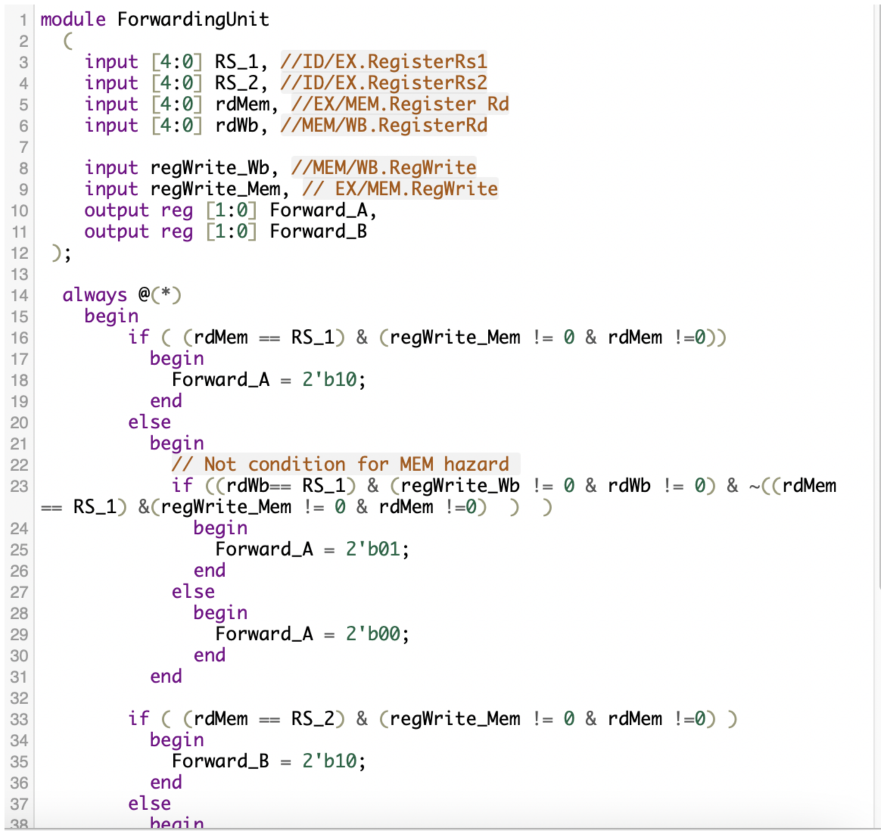
Explained above as well, but to elaborate further more for its role with respect to task 3.

The code checks for two types of data hazards:

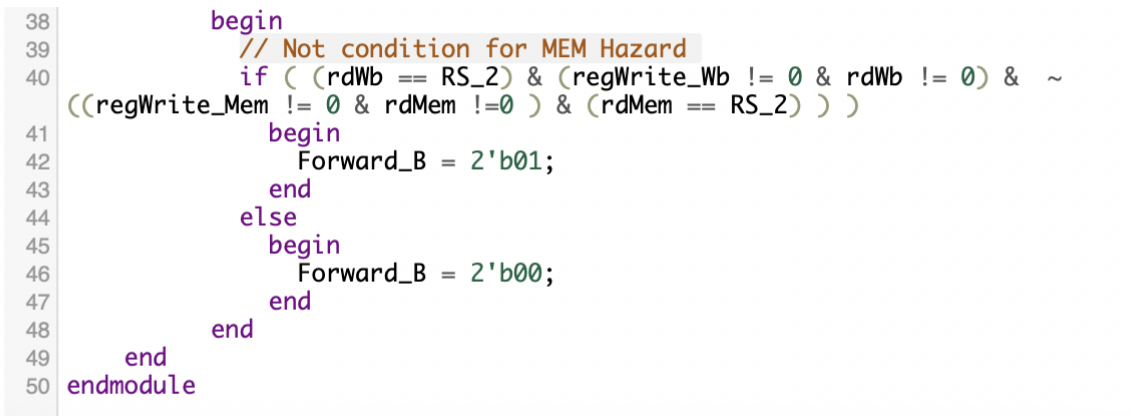
1. MEM Hazard: When the register destination of the memory stage (rdMem) matches the first or second source register of the current stage (RS_1 or RS_2) and there is a register write (regWrite_Mem=1), then data forwarding is required. The code sets Forward_A or Forward_B to 2'b10 to forward the data from the memory stage.
2. WB Hazard: When the register destination of the write-back stage (rdWb) matches the first or second source register of the current stage (RS_1 or RS_2) and there is a register write (regWrite_Wb=1), then data forwarding is required. The code sets Forward_A or Forward_B to 2'b01 to forward the

data from the write-back stage.

This Forwarding Unit module detects data hazards in the pipeline and forwards data from previous pipeline stages to the current stage to avoid stalling the pipeline. It does so by checking the destination register of each stage (rdMem and rdWb) and comparing it to the source registers of the current stage (RS_1 and RS_2). Depending on the hazard detected, the module sets Forward_A and Forward_B signals to forward the data from the appropriate stage.



```
module ForwardingUnit
  (
    input [4:0] RS_1, //ID/EX.RegisterRs1
    input [4:0] RS_2, //ID/EX.RegisterRs2
    input [4:0] rdMem, //EX/MEM.Register Rd
    input [4:0] rdWb, //MEM/WB.RegisterRd

    input regWrite_Wb, //MEM/WB.RegWrite
    input regWrite_Mem, // EX/MEM.RegWrite
    output reg [1:0] Forward_A,
    output reg [1:0] Forward_B
 );

  always @(*)
    begin
        if ( (rdMem == RS_1) & (regWrite_Mem != 0 & rdMem !=0))
          begin
            Forward_A = 2'b10;
          end
        else
          begin
            // Not condition for MEM hazard
            if ((rdWb== RS_1) & (regWrite_Wb != 0 & rdWb != 0) & ~((rdMem == RS_1) &(regWrite_Mem != 0 & rdMem !=0)  )  )
              begin
                Forward_A = 2'b01;
              end
            else
              begin
                Forward_A = 2'b00;
              end
          end

        if ( (rdMem == RS_2) & (regWrite_Mem != 0 & rdMem !=0) )
          begin
            Forward_B = 2'b10;
          end
        else
          begin
```



```
     begin
      // Not condition for MEM Hazard
      if ( (rdWb == RS_2) & (regWrite_Wb != 0 & rdWb != 0) & ~((regWrite_Mem != 0 & rdMem !=0 ) & (rdMem == RS_2) ) )
       begin
        Forward_B = 2'b01;
       end
      else
       begin
        Forward_B = 2'b00;
       end
     end
  end
endmodule
```

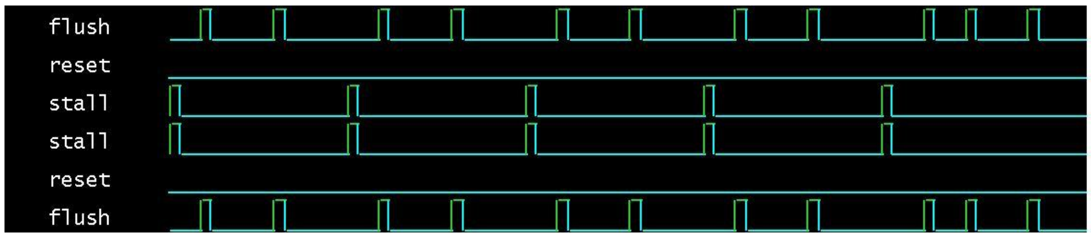
<u>Testing Flush signals and stall signals:</u>

In order to ensure the proper functioning of the hazard detection unit and stall mechanism, a series of tests were carried out on the pipelined processor. These tests were designed to simulate different types of hazards that can occur in the pipeline and verify that the hazard detection unit is able to detect these hazards and stall the pipeline when necessary.

The first set of tests focused on data hazards, where a dependent instruction tries to read a register that has not been written to yet by the previous instruction. To simulate this, a series of instructions were loaded into the pipeline, where the second instruction tried to read from a register that was written to by the first instruction. The hazard detection unit was expected to detect this hazard and stall the pipeline until the first instruction had finished writing to the register.

The second set of tests focused on control hazards, where a branch instruction is executed and the pipeline needs to be flushed. To simulate this, a series of instructions were loaded into the pipeline, and a branch instruction was inserted in the middle. The hazard detection unit was expected to detect the branch hazard and signal the pipeline flush module to flush the pipeline.

The third set of tests focused on structural hazards, where multiple instructions try to access the same resource at the same time. To simulate this, a series of instructions were loaded into the pipeline, where two instructions tried to access the memory at the same time. The hazard detection unit was expected to detect this hazard and stall the pipeline until the memory access was completed.



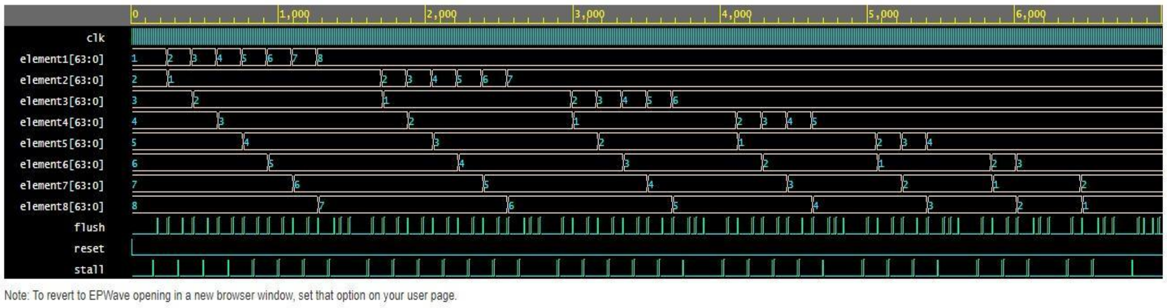
The above figure shows the testing for stalling and flushing.

Task 4 - Performance comparison

Task 4 not implemented due to time constraints

Results:

- Passed 1,2,3,4,5,6,7,8 in the data memory for sorting to be applied on them
- If the sorting works properly it should be sorted as 8,7,6,5,4,3,2,1



Sorting in working condition on pipelined processor

Flush and stall also working as can been seen from signals generated

EDA link for detailed wave: https://www.edaplayground.com/x/AU4D

Challenges:

The project posed several challenges that needed to be addressed to achieve the primary objective. The following are some of the challenges encountered during the project:

1. Pipelining the Processor: Converting the existing single-cycle processor to a pipelined one was one of the major challenges in this project. Pipelining required careful consideration of the hardware design, and various stages had to be implemented to allow for the proper flow of instructions.

3. Hazards and their Detection: The detection of hazards, such as data, control, and structural hazards, was another significant challenge in this project. These hazards can cause delays or incorrect execution of instructions in the pipeline, leading to incorrect results.

4. Testing and Debugging: Testing and debugging were essential to ensure the proper functioning of the processor. Different types of instructions had to be tested with various test cases to identify any errors and ensure that the processor could execute them correctly.

5. Time Management: Completing the project within the given timeline was also a challenge. The project involved several complex tasks that required careful planning and execution. Managing time efficiently was critical to ensure that the project was completed within the deadline.

6. Implementing Sorting Algorithm: Implementing a sorting algorithm was another significant challenge as the project required the conversion of pseudocode into assembly language using the RISC-V instruction set. Additionally, the sorting algorithm needed to be thoroughly tested to ensure that it was executing correctly on the pipelined processor.

Task Divison:

1. Task 1 - Huzaifa Riaz
2. Task 2 - Humayun Hasan
3. Task 3 - Mohammed Haider Abbas

Conclusion:

Although the project was challenging not only theoretically but the time frame which was given to complete the project was a challenge in itself. Overall, this project has helped us enhance our skills in digital design, processor architecture, pipelining, and testing, making us better equipped to take on more complex projects in the future.

Appendices:

Link to EDA Playground for Task 1: https://www.edaplayground.com/x/wdjz
Link to EDA Playground for Task 2 and 3 : https://www.edaplayground.com/x/AU4D