Project

Khubaib Naeem Kasbaati - Murtaza Faisal Shafi
kk04333 - mf04319

1 Bubble Sort Code

1.1 Code

```
# x13 = i = 0
# x14 = j = 0
# x15 = temp
# x10 = a = 0 or also the base address if not null 0x200 User Input
# x11 = len = User Input
# x20 = offsetted address i
# x21 = offsetted address j
# x22 = a[i]
# x23 = a[j]
addi x25, x0, 3
addi x26, x0, 1
addi x27, x0, 5
addi x28, x0, 2
addi x13, x0, 0 # i = 0
addi x14, x0, 0 # j = 0
addi x10, x0, 0x200 # a = User Input

sw x25, 0(x10)
sw x26, 4(x10)
sw x27, 8(x10)
sw x28, 12(x10)
addi x11, x0, 4 # len = User Input
     Loop:
          beq x13, x11, Exit # if i >= len, Exit
          add x14, x0, x13 # j = i
          add x21, x0, x20
        Loop2:
              beq x14, x11, Loop2Exit # if j >= len, Loop2Exit
              add x24, x20, x10 # x20 now has address of a[i]
              add x25, x21, x10 # x20 now has address of a[j]
              lw x22, 0(x24) # x22 = a[i]
              lw x23, 0(x25) # x21 = a[j]
              addi x21, x21, 4
              addi x14, x14, 1 # j += 1
```

```
                bge x23, x22, Loop2 # if a[j] >= a[i], Loop2
                add x15, x0, x22 # temp = a[i]
                sw x23, 0(x24) # a[i] = a[j]
                sw x15, 0(x25) # a[j] = temp
                beq x0, x0, Loop2 # loop repeat
        Loop2Exit:
               addi x20, x20, 4
                addi x13, x13, 1 # i += 1
                beq x0, x0, Loop # loop repeat
Exit:
```

2 Adder

2.1 Code

```
module adder
(
  input [63:0]a, [63:0]b,
  output [63:0]out
);

reg [63:0]Out;

always @(*)
begin
  Out = a + b;
end

assign out = Out;

endmodule
```

3 Alu 64

3.1 Code

```
module alu_64(
    input [63:0]a, [63:0]b, [3:0]ALUOp,
    output [63:0]Result,
    output reg Zero, GreaterThanEqualZero // ! Line Changed
);

reg [63:0]result;
```

```
always @(*) begin
    begin
        case (ALUOp)
            4'b0000: result = a & b;
            4'b0001: result = a | b;
            4'b0010: result = a + b;
            4'b0110: result = a - b;
            4'b1100: result = ~(a | b);
            default: result = 0;
        endcase
        Zero <= result ? 0:1;
        GreaterThanEqualZero <= result[63] ? 0 : 1; // ! Line added
    end
end

assign Result = result;

endmodule // alu_64
```

4 ALU Control

4.1 Code

```
module ALU_control(
    input [1:0] ALUOp, [3:0] funct,
    output reg [3:0] operation
);
    always @(*)
    begin
        if (ALUOp == 2'b00)
            operation = 4'b0010;

        else if (ALUOp == 2'b01)
            operation = 4'b0110;

        else if (ALUOp == 2'b10)

            if (funct == 4'b0000)
                operation = 4'b0010;

            else if (funct == 4'b1000)
                operation = 4'b0110;

            else if (funct == 4'b0111)
                operation = 4'b0000;

            else if (funct == 4'b0110)
```

```
                operation = 4'b0001;
    end
endmodule
```

5 Control Unit

5.1 Code

```
module Control_Unit
(
    input [6:0]Opcode,
    output [1:0]ALUop,
    output MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, Branch
);

reg [1:0]ALUop1;
reg MemRead1;
reg MemtoReg1;
reg MemWrite1;
reg ALUSrc1;
reg RegWrite1;
reg Branch1;

always @(*)
begin
    case (Opcode)
        7'b0110011 :  begin
                      ALUSrc1 = 0;
                      MemtoReg1 = 0;
                      RegWrite1 = 1;
                      MemRead1 = 0;
                      MemWrite1 = 0;
                      Branch1 = 0;
                      ALUop1 = 2'b10;
                      end
        7'b0000011 :  begin
                      ALUSrc1 = 1;
                      MemtoReg1 = 1;
                      RegWrite1 = 1;
                      MemRead1 = 1;
                      MemWrite1 = 0;
                      Branch1 = 0;
                      ALUop1 = 2'b00;
                      end
        7'b0100011 :  begin
                      ALUSrc1 = 1;
                      MemtoReg1 = 1'bx;
```

```
                            RegWrite1 = 0;
                            MemRead1 = 0;
                            MemWrite1 = 1;
                            Branch1 = 0;
                            ALUop1 = 2'b00;
                            end
            7'b1100011 :    begin
                            ALUSrc1 = 0;
                            MemtoReg1 = 1'bx;
                            RegWrite1 = 0;
                            MemRead1 = 0;
                            MemWrite1 = 0;
                            Branch1 = 1;
                            ALUop1 = 2'b01;
                            end
            7'b0010011:
                    begin
                        Branch1 = 0;
                        MemRead1 = 1;
                        MemtoReg1 = 0;
                        MemWrite1 = 0;
                        ALUSrc1 = 1;
                        RegWrite1 = 1;
                        ALUop1 = 2'b00;
                    end
    endcase
end

assign ALUSrc = ALUSrc1;
assign MemtoReg = MemtoReg1;
assign RegWrite = RegWrite1;
assign MemRead = MemRead1;
assign MemWrite = MemWrite1;
assign Branch = Branch1;
assign ALUop = ALUop1;

endmodule // Control_Unit
```

5.2 Explaination

We added a new input, Funct3 as well as a new output, BranchGeq. This ensures that the 'bge' instruction works with the RISC-V Processor.

6 Data Memory

6.1 Code

```
module data_memory (
   input [63:0]Mem_Addr, [63:0]Write_Data,
   input clk, MemWrite, MemRead,
   output reg [63:0]Read_Data
);

reg [7:0] Array[63:0];

initial
begin
{Array[63], Array[62], Array[61], Array[60], Array[59], Array[58],
    Array[57], Array[56]} = 64'h6;
{Array[55], Array[54], Array[53], Array[52], Array[51], Array[50],
    Array[49], Array[48]} = 64'h9;
{Array[47], Array[46], Array[45], Array[44], Array[43], Array[42],
    Array[41], Array[40]} = 64'h10;
{Array[39], Array[38], Array[37], Array[36], Array[35], Array[34],
    Array[33], Array[32]} = 64'h4;
{Array[31], Array[30], Array[29], Array[28], Array[27], Array[26],
    Array[25], Array[24]} = 64'h5;
{Array[23], Array[22], Array[21], Array[20], Array[19], Array[18],
    Array[17], Array[16]} = 64'h9;
{Array[15], Array[14], Array[13], Array[12], Array[11], Array[10],
    Array[9], Array[8]} = 64'h15;
{Array[7], Array[6], Array[5], Array[4], Array[3], Array[2], Array[1],
    Array[0]} = 64'h7;
end

always @(*) begin
   if (MemRead)
      Read_Data = {Array[Mem_Addr + 7], Array[Mem_Addr + 6],
          Array[Mem_Addr + 5], Array[Mem_Addr + 4], Array[Mem_Addr +
          3], Array[Mem_Addr + 2], Array[Mem_Addr + 1],
          Array[Mem_Addr]};
end

always @(posedge clk) begin
   if (MemWrite)
   begin
      Array[Mem_Addr] = Write_Data[7:0];
   Array[Mem_Addr + 1] = Write_Data[15:8];
   Array[Mem_Addr + 2] = Write_Data[23:16];
   Array[Mem_Addr + 3] = Write_Data[31:24];
   Array[Mem_Addr + 4] = Write_Data[39:32];
   Array[Mem_Addr + 5] = Write_Data[47:40];
   Array[Mem_Addr + 6] = Write_Data[55:48];
   Array[Mem_Addr + 7] = Write_Data[63:56];
   end
```

```
end

endmodule // data_memory
```

7 Data Extract

7.1 Code

```
module dataExtract
(
    input [31:0]instruction,
    output reg [63:0]imm_data
);

always @ (*)
begin
case({instruction[6], instruction[5]})
    2'b00 : imm_data = {{52{instruction[31]}}, instruction[31:20]};
    2'b01 : imm_data = {{52{instruction[31]}}, instruction[31:25],
        instruction[11:7]};
    default : imm_data = {{52{instruction[31]}}, instruction[31],
        instruction[7], instruction[30:25], instruction[11:8]};
endcase
end

endmodule
```

8 Instruction Memory

8.1 Code

```
module Instruction_Memory
(
    input [63:0]Inst_Address,
    output [31:0]Instruction
);


// reg [7:0] inst_mem [7:0];
// initial {inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} =
    32'h016bd463;
// initial {inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} =
    32'h00d00733;
```

```
// initial {inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} =
    32'h00508093;
// initial {inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} =
    32'h00608093;

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

initial {inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} =
    32'h00800593; // done

initial {inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} =
    32'h04b68463; // done
initial {inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} =
    32'h00d00733;
initial {inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} =
    32'h01400ab3;
initial {inst_mem[19], inst_mem[18], inst_mem[17], inst_mem[16]} =
    32'h02b70863;
initial {inst_mem[23], inst_mem[22], inst_mem[21], inst_mem[20]} =
    32'h00aa0c33;
initial {inst_mem[27], inst_mem[26], inst_mem[25], inst_mem[24]} =
    32'h00aa8cb3;

initial {inst_mem[31], inst_mem[30], inst_mem[29], inst_mem[28]} =
    32'h000C3B03; // ld
initial {inst_mem[35], inst_mem[34], inst_mem[33], inst_mem[32]} =
    32'h000CBB83; // ld

initial {inst_mem[39], inst_mem[38], inst_mem[37], inst_mem[36]} =
    32'h008a8a93;
initial {inst_mem[43], inst_mem[42], inst_mem[41], inst_mem[40]} =
    32'h00170713;
initial {inst_mem[47], inst_mem[46], inst_mem[45], inst_mem[44]} =
    32'hff6bd2e3;
initial {inst_mem[51], inst_mem[50], inst_mem[49], inst_mem[48]} =
    32'h016007b3;

initial {inst_mem[55], inst_mem[54], inst_mem[53], inst_mem[52]} =
    32'h017C3023; // sd
initial {inst_mem[59], inst_mem[58], inst_mem[57], inst_mem[56]} =
    32'h00FCB023; // sd

initial {inst_mem[63], inst_mem[62], inst_mem[61], inst_mem[60]} =
    32'hfc000ae3;
initial {inst_mem[67], inst_mem[66], inst_mem[65], inst_mem[64]} =
    32'h008a0a13;
initial {inst_mem[71], inst_mem[70], inst_mem[69], inst_mem[68]} =
    32'h00168693;
initial {inst_mem[75], inst_mem[74], inst_mem[73], inst_mem[72]} =
    32'hfa000ee3;
```

```
initial {inst_mem[79], inst_mem[78], inst_mem[77], inst_mem[76]} =
    32'h00000013;

assign Instruction = {inst_mem[Inst_Address + 3], inst_mem[Inst_Address
    + 2], inst_mem[Inst_Address + 1], inst_mem[Inst_Address]};

endmodule // Instruction_Memory
```

8.2 Explaination

Initially we had a very small Register File but now we have increased the size so that all instructions from the bubble sort assembly code can be executed in the RISC-V Processor.

9 Instruction Parser

9.1 Code

```
module InstructionParser
(
   input [31:0]instruction,
   output [6:0]opcode, [4:0]rd, [2:0]funct3, [4:0]rs1, [4:0]rs2,
       [6:0]funct7
);

assign opcode = instruction[6:0];
assign rd = instruction[11:7];
assign funct3 = instruction[14:12];
assign rs1 = instruction[19:15];
assign rs2 = instruction[24:20];
assign funct7 = instruction[31:25];

endmodule
```

10 MUX2x1

10.1 Code

```
module MUX2x1(
   input [63:0]a, [63:0]b,
   input sel,
   output reg [63:0]data_out
);
```

```
always @ (*) begin
  case(sel)
    1'b0 : data_out = a;
    1'b1 : data_out = b;
    default : data_out = 0;
  endcase
end


endmodule
```

11 Program Counter

11.1 Code

```
module program_counter
(
  input [63:0]PC_In,
  input reset, clk,
  output [63:0]PC_Out
);

reg [63:0]pcOut;


always @(posedge reset or posedge clk)
begin
  if (reset)
    begin
      pcOut = 0;
    end
 else
    begin
      pcOut = PC_In;
    end
end

assign PC_Out = pcOut;



endmodule
```

12 Pipeline Register IFID

12.1 Code

```
module PipelineRegisterIFID(
    input clk,maintain,[31:0]Instruction,[63:0] Address,
    output reg [31:0] OutInstruction,[63:0] OutAddress
);

always @(posedge maintain or posedge clk)
begin
  if (!maintain & clk)
  begin
      OutInstruction=Instruction;
      OutAddress=Address;
  end
end

initial
begin
  OutInstruction=0;
  OutAddress=0;
end

endmodule // PipelineRegisterIFID
```

13 Pipeline Register IDEX

13.1 Code

```
module PipelineRegisterIDEX(
    input MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite,
        Branch,BranchGeq,clk,maintain,[63:0]ReadData1,[63:0]ReadData2,[63:0]Imm,
        [63:0] Address, [3:0] funct, [4:0]WriteReg, [4:0] R1,[4:0] R2,
    input [1:0]ALUop,
    output reg OutMemRead,reg OutMemtoReg,reg OutMemWrite,reg OutALUSrc,
        reg OutRegWrite,reg OutBranch, reg OutBranchGeq, reg [63:0]
        OutReadData1, reg [63:0] OutReadData2, reg [63:0] OutImm,reg
        [63:0] OutAddress,reg [3:0] Outfunct,reg [4:0]OutWriteReg, reg
        [1:0]OutALUop,reg [4:0] OutR1,reg [4:0] OutR2
);

always @(posedge maintain or posedge clk)
begin
  if (!maintain & clk)
  begin
```

```
      OutMemRead=MemRead;
      OutMemtoReg=MemtoReg;
      OutMemWrite=MemWrite;
      OutALUSrc=ALUSrc;
      OutRegWrite=RegWrite;
      OutBranch=Branch;
      OutBranchGeq=BranchGeq;
      OutReadData1=ReadData1;
      OutReadData2=ReadData2;
      OutImm=Imm;
      OutAddress=Address;
      Outfunct=funct;
      OutWriteReg=WriteReg;
      OutALUop=ALUop;
      OutR1=R1;
      OutR2=R2;
  end
end
initial
begin
      OutMemRead=0;
      OutMemtoReg=0;
      OutMemWrite=0;
      OutALUSrc=0;
      OutRegWrite=0;
      OutBranch=0;
      OutBranchGeq=0;
      OutReadData1=0;
      OutReadData2=0;
      OutImm=0;
      OutAddress=0;
      Outfunct=0;
      OutWriteReg=0;
      OutALUop=0;
      OutR1=0;
      OutR2=0;
end
endmodule // PipelineRegisterIDEX
```

14 Pipeline Register EXMEM

14.1 Code

```
module PipelineRegisterEXMEM(
    input Zero, GreaterThanEqualZero,MemRead, MemtoReg, MemWrite,
        RegWrite, Branch,BranchGeq,clk,maintain,[63:0]WriteData, [63:0]
        Address, [4:0]WriteReg,[63:0] ALUResult,
    output reg OutZero,reg OutGreaterThanEqualZero,reg OutMemRead,reg
        OutMemtoReg,reg OutMemWrite, reg OutRegWrite,reg OutBranch, reg
        OutBranchGeq, reg [63:0] OutWriteData, reg [63:0] OutImm,reg
        [63:0] OutAddress,reg [4:0]OutWriteReg, reg [63:0] OutALUResult
);

always @(posedge maintain or posedge clk)
begin
  if (!maintain & clk)
  begin
      OutZero=Zero;
      OutGreaterThanEqualZero=GreaterThanEqualZero;
      OutMemRead=MemRead;
      OutMemtoReg=MemtoReg;
      OutMemWrite=MemWrite;
      OutRegWrite=RegWrite;
      OutBranch=Branch;
      OutBranchGeq=BranchGeq;
      OutWriteData=WriteData;
      OutAddress=Address;
      OutWriteReg=WriteReg;
      OutALUResult=ALUResult;

  end
end
initial
begin
      OutZero=0;
      OutGreaterThanEqualZero=0;
      OutMemRead=0;
      OutMemtoReg=0;
      OutMemWrite=0;
      OutRegWrite=0;
      OutBranch=0;
      OutBranchGeq=0;
      OutWriteData=0;
      OutAddress=0;
      OutWriteReg=0;
      OutALUResult=0;

  end
endmodule // PipelineRegisterEXMEM
```

15 Pipeline Register MEMWB

15.1 Code

```
module PipelineRegisterMEMWB(
    input MemtoReg, RegWrite,
        clk,maintain,[63:0]ReadData,[63:0]ReadData2,
        [4:0]WriteReg,[63:0] WriteData,
    output reg OutMemtoReg, reg OutRegWrite, reg [63:0] OutReadData, reg
        [63:0] OutReadData2, reg [4:0]OutWriteReg, reg [63:0]
        OutWriteData
);

always @(posedge maintain or posedge clk)
begin
  if (!maintain & clk)
  begin
      OutMemtoReg=MemtoReg;
      OutRegWrite=RegWrite;
      OutReadData=ReadData;
      OutReadData2=ReadData2;
      // OutAddress=Address;
      OutWriteReg=WriteReg;
      OutWriteData=WriteData;
      // OutALUResult=ALUResult;
  end
end
initial
begin
      OutMemtoReg=0;
      OutRegWrite=0;
      OutReadData=0;
      OutReadData2=0;
      // OutAddress=Address;
      OutWriteReg=0;
      // OutALUResult=ALUResult;
  end
endmodule // PipelineRegisterMEMWB
```

16 Forwading Unit

16.1 Code

```
module ForwardingUnit(
    input EXMEMRegWrite,MEMWBRegWrite,[4:0]EXMEMRegRd,[4:0]MEMWBRegRd,
        [4:0]IDEXR1,[4:0] IDEXR2,
    output reg [1:0] ForwardA,reg [1:0] ForwardB
);

always @(*)
begin
```

```
    if (EXMEMRegWrite & (EXMEMRegRd != 0) & (EXMEMRegRd == IDEXR1))
    begin
        ForwardA=2'b10;
    end
    if (EXMEMRegWrite & (EXMEMRegRd != 0) & (EXMEMRegRd == IDEXR2))
    begin
        ForwardB=2'b10;
    end
    if (MEMWBRegWrite & (MEMWBRegRd != 0) & (MEMWBRegRd == IDEXR1))
    begin
        ForwardA=2'b01;
    end
    if (MEMWBRegWrite & (MEMWBRegRd != 0) & (MEMWBRegRd == IDEXR2))
    begin
        ForwardB=2'b01;
    end

end

initial
begin
    ForwardA=2'b00;
    ForwardB=2'b00;
end

endmodule // ForwardingUnit
```

17 Register File

17.1 Code

```
module registerFile(
    input [63:0]WriteData, [4:0]RS1, [4:0]RS2, [4:0]RD,
    input RegWrite, clk, reset,
    output [63:0]ReadData1, [63:0]ReadData2
);

reg[63:0] Register[31:0];
reg [63:0]ReadDatareg1;
reg [63:0]ReadDatareg2;

initial Register[0] = 64'd0;
initial Register[1] = 64'd0;
initial Register[2] = 64'd0;
initial Register[3] = 64'd0;
initial Register[4] = 64'd0;
initial Register[5] = 64'd0;
```

```verilog
initial Register[6] = 64'd0;
initial Register[7] = 64'd0;
initial Register[8] = 64'd0;
initial Register[9] = 64'd00;
initial Register[10] = 64'd0;
initial Register[11] = 64'd1;
initial Register[12] = 64'd00;
initial Register[13] = 64'd00;
initial Register[14] = 64'd00;
initial Register[15] = 64'd00;
initial Register[16] = 64'd00;
initial Register[17] = 64'd00;
initial Register[18] = 64'd00;
initial Register[19] = 64'd00;
initial Register[20] = 64'd00;
initial Register[21] = 64'd00;
initial Register[22] = 64'd00;
initial Register[23] = 64'd00;
initial Register[24] = 64'd00;
initial Register[25] = 64'd00;
initial Register[26] = 64'd00;
initial Register[27] = 64'd00;
initial Register[28] = 64'd00;
initial Register[29] = 64'd00;
initial Register[30] = 64'd00;
initial Register[31] = 64'd00;

always @(posedge clk) begin
    if (RegWrite == 1 & reset == 0)
        Register[RD] = WriteData;
end

always @(*) begin
    if (reset == 1)
        begin
            ReadDatareg1 = 64'd0;
            ReadDatareg2 = 64'd0;
        end
    else
        begin
            ReadDatareg1 = Register[RS1];
            ReadDatareg2 = Register[RS2];
        end

end

assign ReadData1 = ReadDatareg1;
assign ReadData2 = ReadDatareg2;

endmodule // registerFile
```

18 RISC V Processor

18.1 Code

```
module RISC_V_Processor(
   input clk, reset

);

wire [63:0]PC_Out;

wire [31:0]Instruction;

wire [63:0]out;
wire [63:0]out1;

wire [6:0]opcode;
wire [4:0]rd;
wire [2:0]funct3;
wire [4:0]rs1;
wire [4:0]rs2;
wire [6:0]funct7;

wire [63:0]ReadData1;
wire [63:0]ReadData2;

wire [1:0]ALUop;
wire MemRead;
wire MemtoReg;
wire MemWrite;
wire ALUSrc;
wire RegWrite;
wire Branch;

wire [63:0]data_out;
wire [63:0]data_out1;
wire [63:0]data_out2;

wire [63:0]imm_data;

wire [63:0]Result;

wire [63:0]Read_Data;

wire [3:0]Operation;
wire Zero;
wire GreaterThanEqualZero;
wire andGateOut;

program_counter pc (
    .PC_In(data_out2),
    .reset(reset),
    .clk(clk),
    .PC_Out(PC_Out)
);

Instruction_Memory imm(
    .Inst_Address(PC_Out),
    .Instruction(Instruction)
);
// PC + 4
adder a1(
    .a(PC_Out),
    .b(64'd4),
    .out(out)
);

// PC + Imm data
adder a2(
    .a(PC_Out),
    .b(imm_data << 1),
    .out(out1)
);

assign andGateOut = Branch & (Zero | (GreaterThanEqualZero & funct3[2]));


InstructionParser ip(
    .instruction(Instruction),
    .opcode(opcode),
    .rd(rd),
    .funct3(funct3),
    .rs1(rs1),
    .rs2(rs2),
    .funct7(funct7)
);
// * dataout1 is data sent by WB
registerFile rf(
    .clk(clk),
    .reset(reset),
    .RS1(rs1),
    .RS2(rs2),
    .RD(rd),
    .RegWrite(RegWrite),
    .WriteData(data_out1),
    .ReadData1(ReadData1),
    .ReadData2(ReadData2)
);

Control_Unit cu(
    .Opcode(opcode),
    .Branch(Branch),
    .MemRead(MemRead),
    .MemtoReg(MemtoReg),
    .ALUop(ALUop),
    .MemWrite(MemWrite),
    .ALUSrc(ALUSrc),
    .RegWrite(RegWrite)
);

alu_64 a(
    .a(ReadData1),
    .b(data_out),
    .ALUOp(Operation),
    .Result(Result),
    .Zero(Zero),
    .GreaterThanEqualZero(GreaterThanEqualZero) // ! Added Greater than
        Zero
);

//  MUX between Register file and ALU
MUX2x1 m1(
    .a(ReadData2),
    .b(imm_data),
    .sel(ALUSrc),
    .data_out(data_out)
);

// Immediate Data Extractor
dataExtract dE(
    .instruction(Instruction),
    .imm_data(imm_data)
);



data_memory dm(
    .Mem_Addr(Result),
    .Write_Data(ReadData2),
    .clk(clk),
    .MemWrite(MemWrite),
    .MemRead(MemRead),
    .Read_Data(Read_Data)
);
```

```
// * MUX after Data memory
MUX2x1 m2(
    .a(Result),
    .b(Read_Data),
    .sel(MemtoReg),
    .data_out(data_out1)
);


ALU_control aa(
    .ALUOp(ALUop),
    .funct({Instruction[30],Instruction[14:12]}),
    .operation(Operation)
);

// Choose whether to +4 or add imm
MUX2x1 m3(
    .a(out),
    .b(out1),
    .sel(Branch & (Zero | (GreaterThanEqualZero & funct3[2]))),
    .data_out(data_out2)
);

endmodule // RISC_V_Processor
```

18.2 Explaination

In the 2 by 1 MUX which is called 'm3' in the module, we are now passing the BranchGeq wire which gets its output from the Control Unit. Then, we make modify the RISC-V Processor to make it into a pipeplined processor with 5 stages. This is done by adding new wires and calling new register modules which are made, 'PipelineRegisterIFID', 'PipelineRegisterIDEX', 'PipelineRegisterEXMEM' and 'PipelineRegisterMEMWB'. We also call a forwading module which helps us to stop the problem of Data Hazards in our pipeplined processor.

19 Test Bench

19.1 Code

```
module tb(

);

reg clk, reset;
```

```
RISC_V_Processor r(
    .clk(clk),
    .reset(reset)

);

initial
begin
clk = 1'b1;
reset = 1'b1;
#8
reset = 1'b0;
end

always
begin
#5 clk = ~clk;
end

endmodule // tb
```

20 run.do File

20.1 Code

```
vlog registerFile.v alu_64.v ALU_Control.v Control_Unit.v data_memory.v
    dataExtract.v program_counter.v adder.v Instruction_Memory.v tb.v
    instructionParser.v MUX2x1.v RISC_V_Processor.v

vsim -novopt work.tb

view wave

add wave -r /*

run 5000ns
```

21 Github Repository

https://github.com/murtaza854/CA-Project

22 Project Explaination

We were initially using the assembly language instructions 'slli' in the bubble sort code so we converted the slli instruction into multiple add and addi instructions which resulted in the same output as 'slli'. The array is sorting regardless of clock starting at 0 or 1. Our first instruction in instruction memory is, '32'h00800593' which basically sets the size of the array(Data Memory) which right now is set to 8 as we have put 8 values in the array. All instructions are working perfectly in a single cycle processor and is executing the bubble sorting algorithim perfectly. 'Pipelining is an implementation technique in which multiple instructions are overlapped during execution'. This statement states what does a pipeplined processor achieve and that is what we are achieving here, now we have tested each instruction in the bubble sort algorithim in the pipeplined processor all of them execute perfectly. These instructions include, 'add', 'addi', 'beq', 'bge', 'sd' and 'ld'. Also, implementation of a pipeplined processor can result in hazards such as data hazards and that is being dealt with by the forwading module. But the forwading module is not just enough for the data hazards. We must stall the pipepline for the combination of the load instruction which is followed by another instruction that reads a result. So this leads us to introducing the hazard detection unit. It operates during the Instruction Decode stage so that it can insert a stall between the load instruction and the instruction that is dependant on it.