Computer Architecure Lab Project Report

Nimra Sohail	ns06867
Areeb Adnan Khan	ak06865
Hamad Abdul Razzaq	hr06899

R.A: Miss Aimen Najeeb

Task 1 - Implementing Sorting Algorithm on RISC-V Single Cycle Processor

Selection of Algorithm, Assembly Language Code & Verification

The selected Sorting Algorithm is the Insertion Sort Algorithm. Its Assembly Language Code can be found in Module 1.1 in the Appendix. The Module was tested on a RISC-V Simulator and the initial and final arrays can be seen in Figure 1 and Figure 2 respectively.

0x000000ec	75	0	0	0
0x000000e8	31	0	0	0
0x000000e4	66	0	0	0
0x000000e0	71	0	0	0
0x000000dc	46	0	0	0
0x000000d8	98	0	0	0
0x000000d4	63	0	0	0
0x000000d0	94	0	0	0
0x000000cc	62	0	0	0
0x000000c8	18	0	0	0

Figure 1: Initialized (Unsorted) Array

0x000000ec	98	0	0	0
0x000000e8	94	0	0	0
0x000000e4	75	0	0	0
0x000000e0	71	0	0	0
0x000000dc	66	0	0	0
0x000000d8	63	0	0	0
0x000000d4	62	0	0	0
0x000000d0	46	0	0	0
0x000000cc	31	0	0	0
0x000000c8	18	0	0	0

Figure 2: Sorted Array

Modifications in Single Cycle Processor & Challenges Addressed

For executing the assembly language code of insertion sort on our lab-made single cycle processor, Following were the required modifications and the way we addressed it.

- **Inclusion of `blt, bge, bne, beq` Instructions** - As any Sorting Algorithm requires less than, greater than, equal to and not equal to comparisions, therefore there was a need to include these four instructions. To do so, a seperate module named Branch Module was created. This module, in addition with the Zero Flag also takes a Positve Flag as an input which indicates whether the Operand 1 is greater than Operand 2 or not. The Positive Flag is outputed from ALU by simply looking at the most significant bit of the result. So first, we check the branch signal to see if the instruction is a branch instruction. Then, to differentiate between these four branch instructions, we compare their funct3 values along with the zero and the positive flags and the instruction whose specified conditions are matched is set to high and at the same time, the other instructions are set to low. This module also includes a `to_branch` output which is high if branch condition is met. So, the AND Gate from our previous design is replaced by this branch module.

- **Addition of Load Word and Store Word Instructions** - The Load Word and Store Word functionality was added in the Data Memory. This required because our insertion sort algorithm is implemented on a word array. So, inorder to differentiate between the Load and Store instructions for Word and Double, the funct3 values of the instructions were checked and based on that the load and store instructions for word and double wereintegrated.

Testing and Verification

For testing purposes, There were 10 outputs added to the Data Memory module. These 10 outpus are for the 10 different indices of the array to be sorted. This is not the art of the Single Cycle Processor, rather they are just hard wired to make the verification easier. Upon Converting the assembly language code into machine code and initlizing it in the Instruction Memory, the code was executed and the sorted array can be verified from the waveform shown in Figure 3.

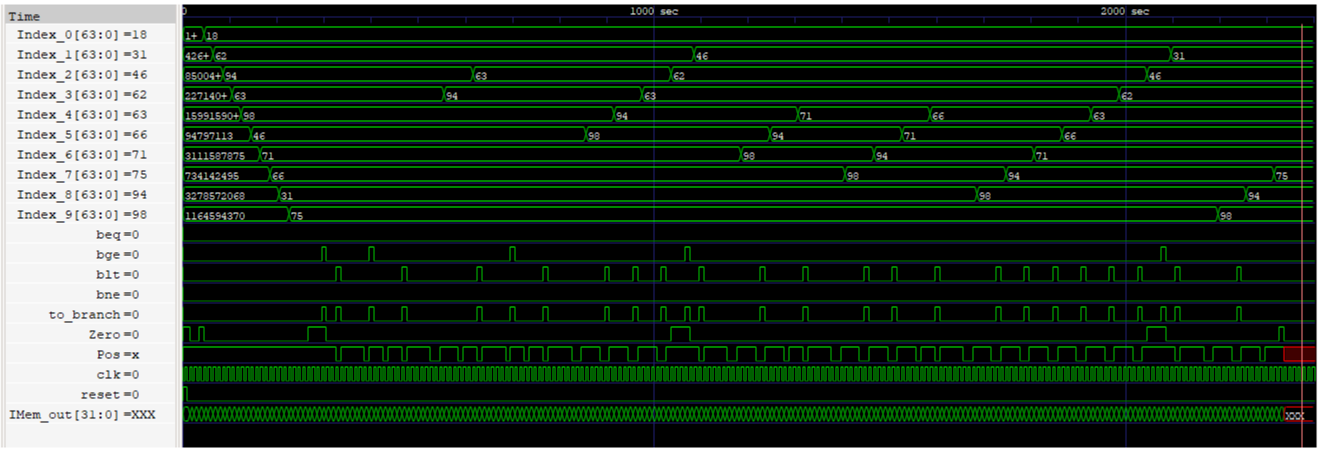

Figure 3: Waveform - Single Cycle Processor

Task 2 - RISC-V Pipelined Architecture

Inclusion of 5 Stages and Integration

To Convert the Single Cycle Processor into Pipeline, 5 Stages were made which were synchornized with the clock. Along with Program Counter, these include the IF/ID, ID/EX, EX/MEM and MEM/WB stage registers. These 5 stage registers were implemented and then they were integrated in the Top Level Module to make our processor pipelined.

Testing Instructions in Isolation

The 3 instructions listed in Module 1.2 of Appendix were tested separately to verify the working of pipeline. The Following three waveforms shows the different stages of the pipeline during the execution of each of these instructions.

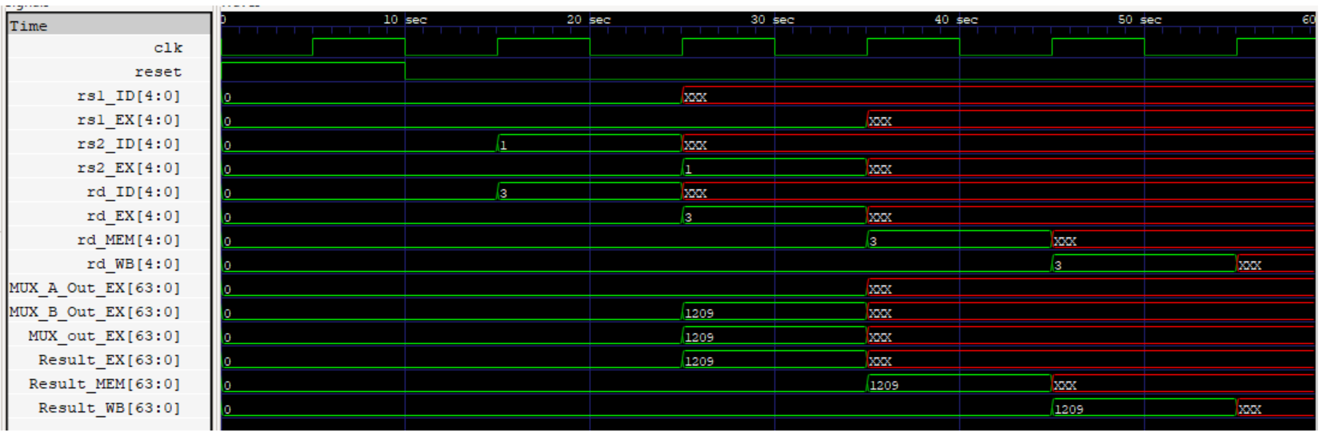

Figure 4: Waveform for the instruction `add x3, x0, x1`

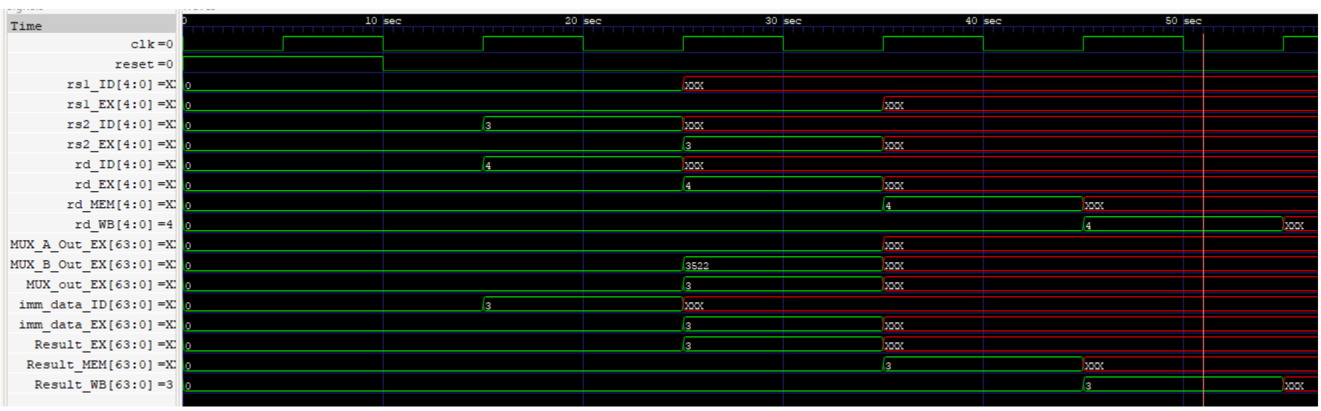

Figure 5: Waveform for the instruction `addi x4, x0, 3`

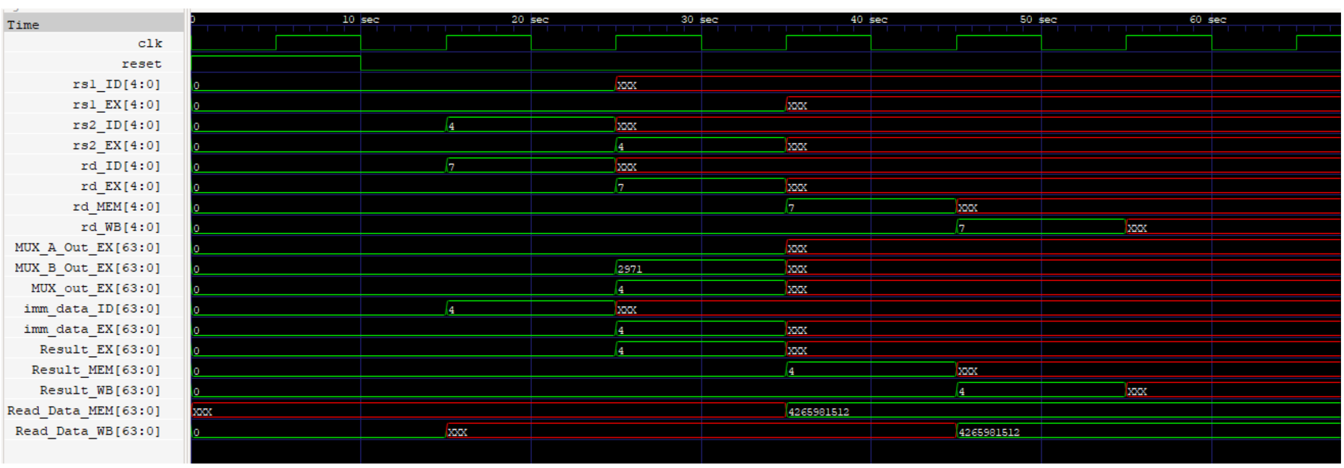

Figure 6: Waveform for the instruction `lw x7, 4(x0)`

Task 3 - Detection of Hazards

Handling Data Hazards

For handling Data Hazards, the Forwarding Unit and the Hazard Detection Unit were integrated to the pipelined architecture. The Forwarding Unit is used along with 2 additional MUX for bypassing the value of ALU Result from MEM stage to the EX stage, or the value of either ALUResult or the Data Memory Output from the WB Stage to the EX Stage. The Hazard Detection Unit handles the case where the data is being loaded into a register and that register is used in the instruction following the load instruction. To handle this, we introoduce a stall. This is done by not changing the values in the Program Counter and IF/ID stages and setting the control values to zero in ID/EX Stage to make the instruction analogous to stall. This is done by adding a Write signal to the first two stages which is set low when the values should remain unchanged and the Control of the third stage is set to zero by inluding an 8 by 2 MUX with one set of inputs hardwired to low.

Handling Control Hazards

The Control Hazard here is handled in the deafult way, that is, by stalling thrice whenever we have to branch. To do this the `to_branch` signal coming from the Branch Module is sent as a Flush signal to the IF/ID, ID/EX and EX/MEM stages. The Flush signal simply makes the Control part of the instructions zero which makes the next 3 instructions equivalent to stall. After that, the branch instruction is fetched and Hence, the branch is made.

Running Insertion Sort on Pipelined Processor

To check the final functionality of the Pipelined Processor with hazards handled, The Insertion Sort Code was executed on it and the following Figure verifies the sorting code working to perfection on the pipelined architecture.

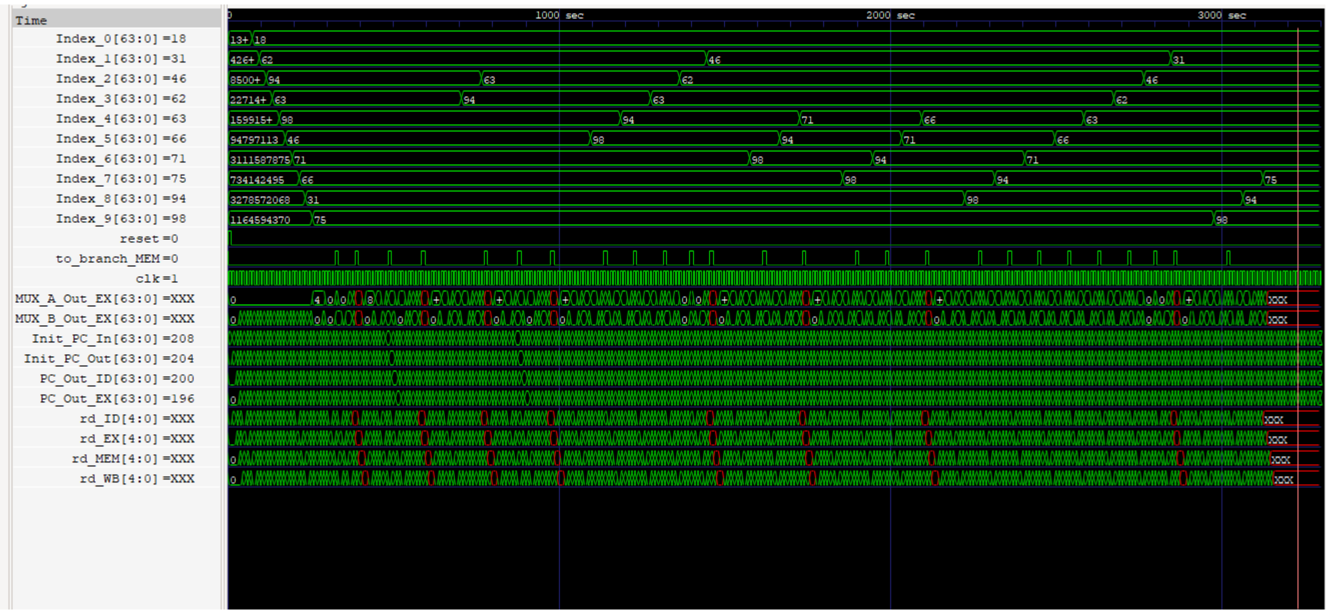

Figure 7: Sorted Array Waveform - Pipelined Architecture

Appendix

Section 1 - Assembly Language Codes

Module 1.1 - Insertion Sort

```
addi x10, x0, 200                # Base Address of the array
addi x11, x0, 40                 # Length of Array {x11 = len(A)}
#----Initializing the array----#
addi x9,  x0, 18
SW   x9,  0(x10)
addi x9,  x0, 62
SW   x9,  4(x10)
addi x9,  x0, 94
SW   x9,  8(x10)
addi x9,  x0, 63
SW   x9,  12(x10)
addi x9,  x0, 98
SW   x9,  16(x10)
addi x9,  x0, 46
SW   x9,  20(x10)
addi x9,  x0, 71
SW   x9,  24(x10)
addi x9,  x0, 66
SW   x9,  28(x10)
addi x9,  x0, 31
SW   x9,  32(x10)
addi x9,  x0, 75
SW   x9,  36(x10)
# Array Initialization Done
#----Sorting via insertion sort----#
addi x1, x0, 4                   # j = 1 {x1 = j}
blt  x1, x10, Outer_Loop         # if j < len(A) -> Start the loop
Outer_Loop:
add  x3, x1, x10                 # j + base address
LW   x4, 0(x3)                   # key = A[j] {x4 = A[j]}
addi x5, x1, -4                  # i = j - 1 {x5 = i}
add  x6, x5, x10                 # i + base address
LW   x7, 0(x6)                   # {x7 = A[i]}
ble  x5, x0, Inner_Loop_Exit     # if i <= 0 -> break
ble  x7, x4, Inner_Loop_Exit     # if A[i] <= key -> break
Inner_Loop:
SW   x7, 4(x6)                   # A[i+1] = A[i]
addi x5, x5, -4                  # i = i - 1
add  x6, x5, x10                 # i + base address
LW   x7, 0(x6)                   # {x7 = A[i]}
ble  x5, x0, Inner_Loop_Exit     # if i <= 0 -> break
bgt  x7, x4, Inner_Loop          # if A[i] > key -> continue
Inner_Loop_Exit:
```

```
SW   x4, 4(x6)                # A[i+1] = key
addi x1, x1, 4                # j = j + 1
blt  x1, x11, Outer_Loop      # if j< Len(A) -> continue
#----End of Code----#
```

Module 1.2 - Isolation Tests

```
add x3, x0, x1
addi x4, x0, 3
lw x7, 4(x0)
```

Section 2 - RISC-V Single Cycle Processor Modules

Module 2.1 - Adder

```
module Adder(
  input [63:0] a,
  input [63:0] b,
  output [63:0] out
);
  assign out = a+b;
endmodule
```

Module 2.2 - ALU (64-bit)

```
module ALU_64_bit(
  input [63:0] a,
  input [63:0] b,
  input [3:0]ALUOp,
  output reg Zero,
  output reg [63:0] Result,
  output reg Pos
);

  always @(*) begin
    if (ALUOp == 4'b0000) begin
     Result = a & b;
  end
    else if (ALUOp == 4'b0001) begin
     Result = a | b;
  end
    else if (ALUOp == 4'b0010) begin
     Result = a + b;
  end
    else if (ALUOp == 4'b0110) begin
     Result = a - b;
  end
    else if (ALUOp == 4'b1100) begin
      Result = ~(a|b);
  end
    if (Result == 0)
```

```
      Zero = 1;
    else
      Zero = 0;
    Pos <= ~Result[63];
  end
endmodule
```

Module 2.3 - ALU Control

```
module ALU_Control(
  input [1:0] ALUOp ,
  input [3:0] Funct ,
  output reg [3:0] Operation
);
  always @(*) begin
    if (ALUOp == 2'b00) begin
      Operation <= 4'b0010;
    end
    else if (ALUOp == 2'b01) begin
      Operation <= 4'b0110;
    end
    else if (ALUOp == 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
          Operation <= 4'b0001;
        end
        default:begin
          Operation <= 4'b0000;
        end
      endcase
    end
  end
endmodule
```

Module 2.4 - Branch Module

```
module branch_module(
  input zero,
  input pos,
  input branch,
  input [2:0] funct3,
  output reg bne,
```

```
  output reg beq,
  output reg bge,
  output reg blt,
  output reg to_branch
);
  always @(*)
    begin
      if (branch) begin
        if (zero && funct3 == 3'b000) begin
            beq <= 1'b1;
            bne <= 1'b0;
            bge <= 1'b0;
            blt <= 1'b0;
        end
        else if (~zero && funct3 == 3'b001) begin
            bne <= 1'b1;
            beq <= 1'b0;
            bge <= 1'b0;
            blt <= 1'b0;
        end
        else if ((pos || zero) && funct3 ==3'b101) begin
            bne <= 1'b0;
            beq <= 1'b0;
            bge <= 1'b1;
            blt <= 1'b0;
        end
        else if ((~pos && ~zero) && funct3 ==3'b100) begin
            bne <= 1'b0;
            beq <= 1'b0;
            blt <= 1'b1;
            bge <= 1'b0;
        end
        else begin
            bne <= 1'b0;
            beq <= 1'b0;
            blt <= 1'b0;
            bge <= 1'b0;
        end
      end
      else begin
            bne <= 1'b0;
            beq <= 1'b0;
            blt <= 1'b0;
            bge <= 1'b0;
      end
      to_branch <= branch && (bne || beq || blt || bge);
    end
endmodule
```

Module 2.5 - Control Unit

```
module Control_Unit(
  input [6:0] opcode,
  output reg [1:0] ALUOp,
  output reg Branch,
  output reg MemRead,
  output reg MemtoReg,
  output reg MemWrite,
  output reg ALUSrc,
  output reg RegWrite
);
  always@(*) begin
    case (opcode)
    7'b0110011: begin
      ALUOp = 2'b10;
      Branch = 1'b0;
      MemWrite = 1'b0;
      MemRead = 1'b0;
      RegWrite = 1'b1;
      MemtoReg = 1'b0;
      ALUSrc = 1'b0;
    end
    7'b0000011: begin
      ALUOp = 2'b00;
      Branch = 1'b0;
      MemWrite = 1'b0;
      MemRead = 1'b1;
      RegWrite = 1'b1;
      MemtoReg = 1'b1;
      ALUSrc = 1'b1;
    end
    7'b0100011: begin
      ALUOp = 2'b00;
      Branch = 1'b0;
      MemWrite = 1'b1;
      MemRead = 1'b0;
      RegWrite = 1'b0;
      MemtoReg = 1'bx;
      ALUSrc = 1'b1;
    end
    7'b1100011: begin
      ALUOp = 2'b01;
      Branch = 1'b1;
      MemWrite = 1'b0;
      MemRead = 1'b0;
      RegWrite = 1'b0;
      MemtoReg = 1'bx;
      ALUSrc = 1'b0;
    end
```

```
    7'b0010011: begin
      ALUOp = 2'b00;
      Branch = 1'b0;
      MemWrite = 1'b0;
      MemRead = 1'b0;
      RegWrite = 1'b1;
      MemtoReg = 1'b0;
      ALUSrc = 1'b1;
    end
    default: begin
      ALUOp = 2'b00;
      Branch = 1'b0;
      MemWrite = 1'b0;
      MemRead = 1'b0;
      RegWrite = 1'b0;
      MemtoReg = 1'b0;
      ALUSrc = 1'b0;
    end
    endcase

  end


endmodule
```

Module 2.6 - Data Memory

```
module Data_Memory(
  input clk,
  input [63:0] Mem_Addr,
  input [63:0] Write_Data,
  input MemWrite,
  input MemRead,
  output reg [63:0] Read_Data,
  output reg [63:0] Index_0,
  output reg [63:0] Index_1,
  output reg [63:0] Index_2,
  output reg [63:0] Index_3,
  output reg [63:0] Index_4,
  output reg [63:0] Index_5,
  output reg [63:0] Index_6,
  output reg [63:0] Index_7,
  output reg [63:0] Index_8,
  output reg [63:0] Index_9,
  input [2:0] funct3
);
  reg [7:0] DMem [63:0];
  initial begin
        DMem[0] = 8'b11101011;
        DMem[1] = 8'b10010;
```

```
DMem[2] = 8'b1111000;
DMem[3] = 8'b1001111;
DMem[4] = 8'b1001000;
DMem[5] = 8'b10110110;
DMem[6] = 8'b1000101;
DMem[7] = 8'b11111110;
DMem[8] = 8'b110000;
DMem[9] = 8'b10000;
DMem[10] = 8'b10001;
DMem[11] = 8'b101;
DMem[12] = 8'b11000010;
DMem[13] = 8'b11011111;
DMem[14] = 8'b1100010;
DMem[15] = 8'b10000111;
DMem[16] = 8'b110101;
DMem[17] = 8'b111011;
DMem[18] = 8'b1010001;
DMem[19] = 8'b1011111;
DMem[20] = 8'b111001;
DMem[21] = 8'b1111101;
DMem[22] = 8'b10100110;
DMem[23] = 8'b101;
DMem[24] = 8'b100011;
DMem[25] = 8'b10000;
DMem[26] = 8'b1110111;
DMem[27] = 8'b10111001;
DMem[28] = 8'b11111;
DMem[29] = 8'b100000;
DMem[30] = 8'b11000010;
DMem[31] = 8'b101011;
DMem[32] = 8'b100100;
DMem[33] = 8'b1010;
DMem[34] = 8'b1101011;
DMem[35] = 8'b11000011;
DMem[36] = 8'b11000010;
DMem[37] = 8'b1001100;
DMem[38] = 8'b1101010;
DMem[39] = 8'b1000101;
DMem[40] = 8'b1001;
DMem[41] = 8'b0;
DMem[42] = 8'b0;
DMem[43] = 8'b0;
DMem[44] = 8'b0;
DMem[45] = 8'b0;
DMem[46] = 8'b0;
DMem[47] = 8'b0;
DMem[48] = 8'b11111000;
DMem[49] = 8'b10101010;
DMem[50] = 8'b11101001;
DMem[51] = 8'b10010011;
```

```
      DMem[52] = 8'b10010110;
      DMem[53] = 8'b11010110;
      DMem[54] = 8'b11111001;
      DMem[55] = 8'b10000000;
      DMem[56] = 8'b1101011;
      DMem[57] = 8'b1101;
      DMem[58] = 8'b1101000;
      DMem[59] = 8'b11111;
      DMem[60] = 8'b11111100;
      DMem[61] = 8'b1000011;
      DMem[62] = 8'b101101;
      DMem[63] = 8'b11100;
end
always @(posedge clk) begin
  if (MemWrite) begin
    if (funct3 == 3'b010) begin
    DMem[Mem_Addr] = Write_Data[7:0];
    DMem[Mem_Addr+1] = Write_Data[15:8];
    DMem[Mem_Addr+2] = Write_Data[23:16];
    DMem[Mem_Addr+3] = Write_Data[31:24];
    end
    else if (funct3 == 3'b011) begin
    DMem[Mem_Addr] = Write_Data[7:0];
    DMem[Mem_Addr+1] = Write_Data[15:8];
    DMem[Mem_Addr+2] = Write_Data[23:16];
    DMem[Mem_Addr+3] = Write_Data[31:24];
    DMem[Mem_Addr+4] = Write_Data[39:32];
    DMem[Mem_Addr+5] = Write_Data[47:40];
    DMem[Mem_Addr+6] = Write_Data[55:48];
    DMem[Mem_Addr+7] = Write_Data[63:56];
    end

  end
end
always @(*) begin
  if (MemRead) begin
    if (funct3 == 3'b010) begin
      Read_Data = {32'd0, DMem[Mem_Addr + 3], DMem[Mem_Addr + 2], DMem[
         Mem_Addr + 1], DMem[Mem_Addr]};
    end
    else if (funct3 == 3'b011) begin
      Read_Data = {DMem[Mem_Addr + 7], DMem[Mem_Addr + 6], DMem[
         Mem_Addr + 5], DMem[Mem_Addr+4], DMem[Mem_Addr + 3], DMem[
         Mem_Addr + 2], DMem[Mem_Addr + 1], DMem[Mem_Addr]};
    end
  end
  Index_0 <= {32'b0,DMem[3],DMem[2],DMem[1],DMem[0]};
  Index_1 <= {32'b0,DMem[7],DMem[6],DMem[5],DMem[4]};
  Index_2 <= {32'b0,DMem[11],DMem[10],DMem[9],DMem[8]};
  Index_3 <= {32'b0,DMem[15],DMem[14],DMem[13],DMem[12]};
```

```
    Index_4 <= {32'b0,DMem[19],DMem[18],DMem[17],DMem[16]};
    Index_5 <= {32'b0,DMem[23],DMem[22],DMem[21],DMem[20]};
    Index_6 <= {32'b0,DMem[27],DMem[26],DMem[25],DMem[24]};
    Index_7 <= {32'b0,DMem[31],DMem[30],DMem[29],DMem[28]};
    Index_8 <= {32'b0,DMem[35],DMem[34],DMem[33],DMem[32]};
    Index_9 <= {32'b0,DMem[39],DMem[38],DMem[37],DMem[36]};
  end
endmodule
/////////
```

Module 2.7 - Immediate Data Generator

```
module imm_data_gen(
  input [31:0] instruction,
  output reg [63:0] imm_data
);
  always @(instruction) begin
  if (instruction[6:5]==2'b00) begin
  imm_data[11:0] = instruction[31:20];
  end
  else if (instruction[6:5]==2'b01) begin
    imm_data[11:5] = instruction[31:25];

    imm_data[4:0] = instruction[11:7];
  end
  else begin
    imm_data[11] = instruction[31];
    imm_data[9:4] = instruction[30:25];
    imm_data[3:0] = instruction[10:8];
    imm_data[10] = instruction[7];
  end
   imm_data[63:12] = {52{imm_data[11]}};
  end
    endmodule
```

Module 2.8 - Instruction Decoder

```
module instruction(
  input [31:0] ins,
  output [6:0] op,
  output [4:0] rd,
  output [2:0] f3,
  output [4:0] rs1,
  output [4:0] rs2,
  output [6:0] f7

);
  assign op = ins[6:0];
  assign rd = ins[11:7];
  assign f3 = ins[14:12];
  assign rs1 = ins[19:15];
```

```
  assign rs2 = ins[24:20];
  assign f7 = ins[31:25];


  endmodule
```

Module 2.9 - Instruction Memory

```
module Instruction_Memory(
  input [63:0] Inst_Address,
  output [31:0] Instruction
);
  reg [7:0] IMem [159:0] ;
  initial begin
    // blt x1 x11 -60
    IMem[159] = 8'b11111100;
    IMem[158] = 8'b10110000;
    IMem[157] = 8'b11000010;
    IMem[156] = 8'b11100011;
    // addi x1 x1 4
    IMem[155] = 8'b00000000;
    IMem[154] = 8'b01000000;
    IMem[153] = 8'b10000000;
    IMem[152] = 8'b10010011;
    // sw x4 4(x6)
    IMem[151] = 8'b00000000;
    IMem[150] = 8'b01000011;
    IMem[149] = 8'b00100010;
    IMem[148] = 8'b00100011;
    // blt x4 x7 -20
    IMem[147] = 8'b11111110;
    IMem[146] = 8'b01110010;
    IMem[145] = 8'b01000110;
    IMem[144] = 8'b11100011;
    // bge x0 x5 8
    IMem[143] = 8'b00000000;
    IMem[142] = 8'b01010000;
    IMem[141] = 8'b01010100;
    IMem[140] = 8'b01100011;
    // lw x7 0(x6)
    IMem[139] = 8'b00000000;
    IMem[138] = 8'b00000011;
    IMem[137] = 8'b00100011;
    IMem[136] = 8'b10000011;
    // add x6 x5 x10
    IMem[135] = 8'b00000000;
    IMem[134] = 8'b10100010;
    IMem[133] = 8'b10000011;
    IMem[132] = 8'b00110011;
    // addi x5 x5 -4
```

```
IMem[131] = 8'b11111111;
IMem[130] = 8'b11000010;
IMem[129] = 8'b10000010;
IMem[128] = 8'b10010011;
// sw x7 4(x6)
IMem[127] = 8'b00000000;
IMem[126] = 8'b01110011;
IMem[125] = 8'b00100010;
IMem[124] = 8'b00100011;
// bge x4 x7 28
IMem[123] = 8'b00000000;
IMem[122] = 8'b01110010;
IMem[121] = 8'b01011110;
IMem[120] = 8'b01100011;
// bge x0 x5 32
IMem[119] = 8'b00000010;
IMem[118] = 8'b01010000;
IMem[117] = 8'b01010000;
IMem[116] = 8'b01100011;
// lw x7 0(x6)
IMem[115] = 8'b00000000;
IMem[114] = 8'b00000011;
IMem[113] = 8'b00100011;
IMem[112] = 8'b10000011;
// add x6 x5 x10
IMem[111] = 8'b00000000;
IMem[110] = 8'b10100010;
IMem[109] = 8'b10000011;
IMem[108] = 8'b00110011;
// addi x5 x1 -4
IMem[107] = 8'b11111111;
IMem[106] = 8'b11000000;
IMem[105] = 8'b10000010;
IMem[104] = 8'b10010011;
// lw x4 0(x3)
IMem[103] = 8'b00000000;
IMem[102] = 8'b00000001;
IMem[101] = 8'b10100010;
IMem[100] = 8'b00000011;
// add x3 x1 x10
IMem[99] = 8'b00000000;
IMem[98] = 8'b10100000;
IMem[97] = 8'b10000001;
IMem[96] = 8'b10110011;
// blt x1 x10 4
IMem[95] = 8'b00000000;
IMem[94] = 8'b10100000;
IMem[93] = 8'b11000010;
IMem[92] = 8'b01100011;
// addi x1 x0 4
```

```
IMem[91] = 8'b00000000;
IMem[90] = 8'b01000000;
IMem[89] = 8'b00000000;
IMem[88] = 8'b10010011;
// sw x9 36(x10)
IMem[87] = 8'b00000010;
IMem[86] = 8'b10010101;
IMem[85] = 8'b00100010;
IMem[84] = 8'b00100011;
// addi x9 x0 75
IMem[83] = 8'b00000100;
IMem[82] = 8'b10110000;
IMem[81] = 8'b00000100;
IMem[80] = 8'b10010011;
// sw x9 32(x10)
IMem[79] = 8'b00000010;
IMem[78] = 8'b10010101;
IMem[77] = 8'b00100000;
IMem[76] = 8'b00100011;
// addi x9 x0 31
IMem[75] = 8'b00000001;
IMem[74] = 8'b11110000;
IMem[73] = 8'b00000100;
IMem[72] = 8'b10010011;
// sw x9 28(x10)
IMem[71] = 8'b00000000;
IMem[70] = 8'b10010101;
IMem[69] = 8'b00101110;
IMem[68] = 8'b00100011;
// addi x9 x0 66
IMem[67] = 8'b00000100;
IMem[66] = 8'b00100000;
IMem[65] = 8'b00000100;
IMem[64] = 8'b10010011;
// sw x9 24(x10)
IMem[63] = 8'b00000000;
IMem[62] = 8'b10010101;
IMem[61] = 8'b00101100;
IMem[60] = 8'b00100011;
// addi x9 x0 71
IMem[59] = 8'b00000100;
IMem[58] = 8'b01110000;
IMem[57] = 8'b00000100;
IMem[56] = 8'b10010011;
// sw x9 20(x10)
IMem[55] = 8'b00000000;
IMem[54] = 8'b10010101;
IMem[53] = 8'b00101010;
IMem[52] = 8'b00100011;
// addi x9 x0 46
```

```
IMem[51] = 8'b00000010;
IMem[50] = 8'b11100000;
IMem[49] = 8'b00000100;
IMem[48] = 8'b10010011;
// sw x9 16(x10)
IMem[47] = 8'b00000000;
IMem[46] = 8'b10010101;
IMem[45] = 8'b00101000;
IMem[44] = 8'b00100011;
// addi x9 x0 98
IMem[43] = 8'b00000110;
IMem[42] = 8'b00100000;
IMem[41] = 8'b00000100;
IMem[40] = 8'b10010011;
// sw x9 12(x10)
IMem[39] = 8'b00000000;
IMem[38] = 8'b10010101;
IMem[37] = 8'b00100110;
IMem[36] = 8'b00100011;
// addi x9 x0 63
IMem[35] = 8'b00000011;
IMem[34] = 8'b11110000;
IMem[33] = 8'b00000100;
IMem[32] = 8'b10010011;
// sw x9 4(x10)
IMem[31] = 8'b00000000;
IMem[30] = 8'b10010101;
IMem[29] = 8'b00100100;
IMem[28] = 8'b00100011;
// addi x9 x0 94
IMem[27] = 8'b00000101;
IMem[26] = 8'b11100000;
IMem[25] = 8'b00000100;
IMem[24] = 8'b10010011;
// sw x9 4(x10)
IMem[23] = 8'b00000000;
IMem[22] = 8'b10010101;
IMem[21] = 8'b00100010;
IMem[20] = 8'b00100011;
// addi x9 x0 62
IMem[19] = 8'b00000011;
IMem[18] = 8'b11100000;
IMem[17] = 8'b00000100;
IMem[16] = 8'b10010011;
// sw x9 0(x10)
IMem[15] = 8'b00000000;
IMem[14] = 8'b10010101;
IMem[13] = 8'b00100000;
IMem[12] = 8'b00100011;
// addi x9 x0 18
```

```
    IMem[11]  = 8'b00000001;
    IMem[10]  = 8'b00100000;
    IMem[9]   = 8'b00000100;
    IMem[8]   = 8'b10010011;
    // addi x11 x0 40
    IMem[7]   = 8'b00000010;
    IMem[6]   = 8'b10000000;
    IMem[5]   = 8'b00000101;
    IMem[4]   = 8'b10010011;
    // addi x10 x0 0
    IMem[3]   = 8'b00000000;
    IMem[2]   = 8'b00000000;
    IMem[1]   = 8'b00000101;
    IMem[0]   = 8'b00010011;
  end
  assign Instruction[31:0] = {IMem[Inst_Address+2'b11], IMem[Inst_Address
     +2'b10], IMem[Inst_Address+1'b1], IMem[Inst_Address]};

endmodule
```

Module 2.10 - MUX (64-bit, 2 by 1)

```
module MUX(
  input [63:0] X,
  input [63:0] Y,
  input S,
  output [63:0] O
);
  assign O = S?Y:X;
endmodule
```

Module 2.11 - Program Counter

```
module Program_Counter(
  input clk,
  input reset,
  input [63:0] PC_In,
  output reg [63:0] PC_Out
);
  always @(posedge clk or posedge reset)
    begin
      if (reset) begin
        PC_Out <= 64'd0;
      end
      else begin
        PC_Out <= PC_In;
      end
    end
endmodule
```

Module 2.12 - Register File

```
module registerFile(
  input clk,
  input reset,
  input [4:0] rs1,
  input [4:0] rs2,
  input [4:0] rd,
  input [63:0] write_data,
  input reg_write,
  output reg [63:0] readdata1,
  output reg [63:0] readdata2
);

  reg [63:0] Registers [31:0];
  initial begin
        Registers[0] = 64'd0;
        Registers[1] = 64'd1209;
        Registers[2] = 64'd751;
        Registers[3] = 64'd3522;
        Registers[4] = 64'd2971;
        Registers[5] = 64'd72;
        Registers[6] = 64'd1135;
        Registers[7] = 64'd1141;
        Registers[8] = 64'd2919;
        Registers[9] = 64'd2467;
        Registers[10] = 64'd0;
        Registers[11] = 64'd3033;
        Registers[12] = 64'd3278;
        Registers[13] = 64'd3214;
        Registers[14] = 64'd3656;
        Registers[15] = 64'd1765;
        Registers[16] = 64'd736;
        Registers[17] = 64'd2985;
        Registers[18] = 64'd2717;
        Registers[19] = 64'd863;
        Registers[20] = 64'd1916;
        Registers[21] = 64'd13;
        Registers[22] = 64'd701;
        Registers[23] = 64'd3479;
        Registers[24] = 64'd2489;
        Registers[25] = 64'd1937;
        Registers[26] = 64'd523;
        Registers[27] = 64'd210;
        Registers[28] = 64'd1043;
        Registers[29] = 64'd425;
        Registers[30] = 64'd2434;
        Registers[31] = 64'd988;
  end
  always @(posedge clk) begin
    if (reg_write & rd != 5'd0) begin
        Registers[rd] <= write_data;
```

```
    end
  end
  always @(*) begin
    if (reset) begin
      readdata1 <= 64'b0;
      readdata2 <= 64'b0;
    end
    else begin
      readdata1 <= Registers[rs1];
      readdata2 <= Registers[rs2];
    end
  end
endmodule
```

Module 2.13 - RISC-V Processor (Top Level Module)

```
`include "Adder.v"
`include "ALU_64_bit.v"
`include "ALU_Control.v"
`include "branch_module.v"
`include "Control_Unit.v"
`include "Data_Memory.v"
`include "imm_data_gen.v"
`include "Instruction_Memory.v"
`include "instruction.v"
`include "MUX.v"
`include "registerFile.v"
`include "shift_left.v"
`include "Program_Counter.v"
module RISC_V_Processor(
  input clk,
  input reset
);
  wire [63:0] PC_Out;
  wire [63:0] PC_In;
  wire [63:0] PC_offset_4;
  wire [31:0] IMem_out;
  wire [6:0] opcode;
  wire [4:0] rs1;
  wire [4:0] rs2;
  wire [4:0] rd;
  wire [2:0] funct3;
  wire [6:0] funct7;
  wire [63:0] imm_data;
  wire [1:0] ALUOp;
  wire Branch;
  wire MemRead;
  wire MemtoReg;
  wire MemWrite;
  wire ALUSrc;
```

```
  wire RegWrite;
  wire [63:0] write_data;
  wire [63:0] readdata1;
  wire [63:0] readdata2;
  wire [3:0] Operation;
  wire Zero;
  wire [63:0] Result;
  wire [63:0] mux_1_out;
  wire [63:0] DMem_Read;
  wire [63:0] shifted_imm_data;
  wire [63:0] PC_offset_branch;
  wire mux3_select;
  wire bge, blt, bne, beq;
  wire Pos;
  wire [63:0] Index_0;
  wire [63:0] Index_1;
  wire [63:0] Index_2;
  wire [63:0] Index_3;
  wire [63:0] Index_4;
  wire [63:0] Index_5;
  wire [63:0] Index_6;
  wire [63:0] Index_7;
  wire [63:0] Index_8;
  wire [63:0] Index_9;
  wire to_branch;
  Program_Counter p1(clk, reset, PC_In, PC_Out);
  Adder a1(PC_Out, 64'd4, PC_offset_4);
  Instruction_Memory i1(PC_Out, IMem_out);
  instruction i3(IMem_out, opcode, rd, funct3, rs1, rs2, funct7);
  imm_data_gen i2(IMem_out, imm_data);
  Control_Unit c1(opcode, ALUOp, Branch, MemRead, MemtoReg, MemWrite,
     ALUSrc, RegWrite);
  registerFile r1(clk, reset, rs1, rs2, rd, write_data, RegWrite,
     readdata1, readdata2);
  ALU_Control a2(ALUOp,{IMem_out[30],IMem_out[14:12]},Operation);
  MUX m1(readdata2,  imm_data, ALUSrc, mux_1_out);
  ALU_64_bit a3(readdata1, mux_1_out, Operation, Zero, Result, Pos);
  Data_Memory d1(clk, Result, readdata2, MemWrite, MemRead, DMem_Read,
     Index_0, Index_1, Index_2, Index_3, Index_4, Index_5, Index_6,
     Index_7, Index_8, Index_9, funct3);
  MUX m2(Result,DMem_Read,  MemtoReg, write_data);
  shift_left s1(imm_data, shifted_imm_data);
  Adder a4(PC_Out, shifted_imm_data, PC_offset_branch);
  branch_module b1(Zero, Pos, Branch, funct3, bne, beq, bge, blt,
     to_branch);
  MUX m3(PC_offset_4, PC_offset_branch, to_branch, PC_In);


endmodule
```

Module 2.14 - Shift Left

```
module shift_left(
  input [63:0] a ,
  output [63:0] b
);
  assign b ={a[62:0],1'b0};
endmodule
```

Module 2.15 - Test Bench

```
`include "RISC_V_Processor.v"
module tb();
  reg dclk;
  reg dreset;

  RISC_V_Processor r1(dclk, dreset);
  initial begin
    dclk = 1'b0;
  end
  always begin
    #5
    dclk = ~dclk;
  end
  initial begin
    dreset = 1'b1;
    #10
    dreset = 1'b0;
    #4000
    dreset = 1'b1;
    $finish;
  end

  initial begin
    $dumpfile("tests.vcd");
    $dumpvars(3,tb);
  end
endmodule
```

Section 3 - RISC-V Pipeline Processor Modules

Module 3.1 - Adder

```
module Adder(
  input [63:0] a,
  input [63:0] b,
  output [63:0] out
);
  assign out = a+b;
endmodule
```

Module 3.2 - ALU (64-bit)

```
module ALU_64_bit(
  input [63:0] a,
  input [63:0] b,
  input [3:0]ALUOp,
  output reg Zero,
  output reg [63:0] Result,
  output reg Pos
);

  always @(*) begin
    if (ALUOp == 4'b0000) begin
     Result = a & b;
  end
    else if (ALUOp == 4'b0001) begin
     Result = a | b;
  end
    else if (ALUOp == 4'b0010) begin
     Result = a + b;
  end
    else if (ALUOp == 4'b0110) begin
     Result = a - b;
  end
    else if (ALUOp == 4'b1100) begin
      Result = ~(a|b);
  end
    if (Result == 0)
      Zero = 1;
    else
      Zero = 0;
    Pos <= ~Result[63];
  end
endmodule
```

Module 3.3 - ALU Control

```
module ALU_Control(
  input [1:0] ALUOp ,
  input [3:0] Funct ,
  output reg [3:0] Operation
);
  always @(*) begin
    if (ALUOp == 2'b00) begin
      Operation <= 4'b0010;
    end
    else if (ALUOp == 2'b01) begin
      Operation <= 4'b0110;
    end
    else if (ALUOp == 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
          Operation <= 4'b0001;
        end
        default:begin
          Operation <= 4'b0000;
        end
      endcase
    end
  end
endmodule
```

Module 3.4 - Branch Module

```
module branch_module(
  input zero,
  input pos,
  input branch,
  input [2:0] funct3,
  output reg bne,
  output reg beq,
  output reg bge,
  output reg blt,
  output reg to_branch
);
  always @(*)
    begin
      if (branch) begin
        if (zero && funct3 == 3'b000) begin
            beq <= 1'b1;
            bne <= 1'b0;
            bge <= 1'b0;
            blt <= 1'b0;
        end
        else if (~zero && funct3 == 3'b001) begin
            bne <= 1'b1;
            beq <= 1'b0;
            bge <= 1'b0;
            blt <= 1'b0;
        end
        else if ((pos || zero) && funct3 ==3'b101) begin
```

```
            bne <= 1'b0;
            beq <= 1'b0;
            bge <= 1'b1;
            blt <= 1'b0;
        end
        else if ((~pos && ~zero) && funct3 ==3'b100) begin
            bne <= 1'b0;
            beq <= 1'b0;
            blt <= 1'b1;
            bge <= 1'b0;
        end
        else begin
            bne <= 1'b0;
            beq <= 1'b0;
            blt <= 1'b0;
            bge <= 1'b0;
        end
      end
      else begin
            bne <= 1'b0;
            beq <= 1'b0;
            blt <= 1'b0;
            bge <= 1'b0;
      end
      to_branch <= branch && (bne || beq || blt || bge);
    end
endmodule
```

Module 3.5 - Control Unit

```
module Control_Unit(
  input [6:0] opcode,
  output reg [1:0] ALUOp,
  output reg Branch,
  output reg MemRead,
  output reg MemtoReg,
  output reg MemWrite,
  output reg ALUSrc,
  output reg RegWrite
);
  always@(*) begin
    case (opcode)
    7'b0110011: begin
      ALUOp = 2'b10;
      Branch = 1'b0;
      MemWrite = 1'b0;
      MemRead = 1'b0;
      RegWrite = 1'b1;
      MemtoReg = 1'b0;
      ALUSrc = 1'b0;
```

```
    end
    7'b0000011: begin
      ALUOp = 2'b00;
      Branch = 1'b0;
      MemWrite = 1'b0;
      MemRead = 1'b1;
      RegWrite = 1'b1;
      MemtoReg = 1'b1;
      ALUSrc = 1'b1;
    end
    7'b0100011: begin
      ALUOp = 2'b00;
      Branch = 1'b0;
      MemWrite = 1'b1;
      MemRead = 1'b0;
      RegWrite = 1'b0;
      MemtoReg = 1'bx;
      ALUSrc = 1'b1;
    end
    7'b1100011: begin
      ALUOp = 2'b01;
      Branch = 1'b1;
      MemWrite = 1'b0;
      MemRead = 1'b0;
      RegWrite = 1'b0;
      MemtoReg = 1'bx;
      ALUSrc = 1'b0;
    end
    7'b0010011: begin
      ALUOp = 2'b00;
      Branch = 1'b0;
      MemWrite = 1'b0;
      MemRead = 1'b0;
      RegWrite = 1'b1;
      MemtoReg = 1'b0;
      ALUSrc = 1'b1;
    end
    default: begin
      ALUOp = 2'b00;
      Branch = 1'b0;
      MemWrite = 1'b0;
      MemRead = 1'b0;
      RegWrite = 1'b0;
      MemtoReg = 1'b0;
      ALUSrc = 1'b0;
    end
    endcase

  end
```

```
endmodule
```

Module 3.6 - Data Memory

```
module Data_Memory(
  input clk,
  input [63:0] Mem_Addr,
  input [63:0] Write_Data,
  input MemWrite,
  input MemRead,
  output reg [63:0] Read_Data,
  output reg [63:0] Index_0,
  output reg [63:0] Index_1,
  output reg [63:0] Index_2,
  output reg [63:0] Index_3,
  output reg [63:0] Index_4,
  output reg [63:0] Index_5,
  output reg [63:0] Index_6,
  output reg [63:0] Index_7,
  output reg [63:0] Index_8,
  output reg [63:0] Index_9,
  input [2:0] funct3
);
  reg [7:0] DMem [63:0];
  initial begin
        DMem[0] = 8'b11101011;
        DMem[1] = 8'b10010;
        DMem[2] = 8'b1111000;
        DMem[3] = 8'b01001111;
        DMem[4] = 8'b1001000;
        DMem[5] = 8'b10110110;
        DMem[6] = 8'b1000101;
        DMem[7] = 8'b11111110;
        DMem[8] = 8'b110000;
        DMem[9] = 8'b10000;
        DMem[10] = 8'b10001;
        DMem[11] = 8'b101;
        DMem[12] = 8'b11000010;
        DMem[13] = 8'b11011111;
        DMem[14] = 8'b1100010;
        DMem[15] = 8'b10000111;
        DMem[16] = 8'b110101;
        DMem[17] = 8'b111011;
        DMem[18] = 8'b1010001;
        DMem[19] = 8'b1011111;
        DMem[20] = 8'b111001;
        DMem[21] = 8'b1111101;
        DMem[22] = 8'b10100110;
        DMem[23] = 8'b101;
```

```
      DMem[24] = 8'b100011;
      DMem[25] = 8'b10000;
      DMem[26] = 8'b1110111;
      DMem[27] = 8'b10111001;
      DMem[28] = 8'b11111;
      DMem[29] = 8'b100000;
      DMem[30] = 8'b11000010;
      DMem[31] = 8'b101011;
      DMem[32] = 8'b100100;
      DMem[33] = 8'b1010;
      DMem[34] = 8'b1101011;
      DMem[35] = 8'b11000011;
      DMem[36] = 8'b11000010;
      DMem[37] = 8'b1001100;
      DMem[38] = 8'b1101010;
      DMem[39] = 8'b1000101;
      DMem[40] = 8'b1001;
      DMem[41] = 8'b0;
      DMem[42] = 8'b0;
      DMem[43] = 8'b0;
      DMem[44] = 8'b0;
      DMem[45] = 8'b0;
      DMem[46] = 8'b0;
      DMem[47] = 8'b0;
      DMem[48] = 8'b11111000;
      DMem[49] = 8'b10101010;
      DMem[50] = 8'b11101001;
      DMem[51] = 8'b10010011;
      DMem[52] = 8'b10010110;
      DMem[53] = 8'b11010110;
      DMem[54] = 8'b11111001;
        DMem[55] = 8'b10000000;
      DMem[56] = 8'b1101011;
      DMem[57] = 8'b1101;
      DMem[58] = 8'b1101000;
      DMem[59] = 8'b11111;
      DMem[60] = 8'b11111100;
      DMem[61] = 8'b1000011;
      DMem[62] = 8'b101101;
      DMem[63] = 8'b11100;
end
always @(posedge clk) begin
  if (MemWrite) begin
    if (funct3 == 3'b010) begin
    DMem[Mem_Addr] = Write_Data[7:0];
    DMem[Mem_Addr+1] = Write_Data[15:8];
    DMem[Mem_Addr+2] = Write_Data[23:16];
    DMem[Mem_Addr+3] = Write_Data[31:24];
    end
    else if (funct3 == 3'b011) begin
```

```
      DMem[Mem_Addr] = Write_Data[7:0];
      DMem[Mem_Addr+1] = Write_Data[15:8];
      DMem[Mem_Addr+2] = Write_Data[23:16];
      DMem[Mem_Addr+3] = Write_Data[31:24];
      DMem[Mem_Addr+4] = Write_Data[39:32];
      DMem[Mem_Addr+5] = Write_Data[47:40];
      DMem[Mem_Addr+6] = Write_Data[55:48];
      DMem[Mem_Addr+7] = Write_Data[63:56];
      end

    end
  end
  always @(*) begin
    if (MemRead) begin
      if (funct3 == 3'b010) begin
        Read_Data = {32'd0, DMem[Mem_Addr + 3], DMem[Mem_Addr + 2], DMem[
           Mem_Addr + 1], DMem[Mem_Addr]};
      end
      else if (funct3 == 3'b011) begin
        Read_Data = {DMem[Mem_Addr + 7], DMem[Mem_Addr + 6], DMem[
           Mem_Addr + 5], DMem[Mem_Addr+4], DMem[Mem_Addr + 3], DMem[
           Mem_Addr + 2], DMem[Mem_Addr + 1], DMem[Mem_Addr]};
      end
    end
    Index_0 <= {32'b0,DMem[3],DMem[2],DMem[1],DMem[0]};
    Index_1 <= {32'b0,DMem[7],DMem[6],DMem[5],DMem[4]};
    Index_2 <= {32'b0,DMem[11],DMem[10],DMem[9],DMem[8]};
    Index_3 <= {32'b0,DMem[15],DMem[14],DMem[13],DMem[12]};
    Index_4 <= {32'b0,DMem[19],DMem[18],DMem[17],DMem[16]};
    Index_5 <= {32'b0,DMem[23],DMem[22],DMem[21],DMem[20]};
    Index_6 <= {32'b0,DMem[27],DMem[26],DMem[25],DMem[24]};
    Index_7 <= {32'b0,DMem[31],DMem[30],DMem[29],DMem[28]};
    Index_8 <= {32'b0,DMem[35],DMem[34],DMem[33],DMem[32]};
    Index_9 <= {32'b0,DMem[39],DMem[38],DMem[37],DMem[36]};
  end
endmodule
/////////
```

Module 3.7 - EX/MEM Stage Register

```
// Code your design here
module EX_MEM(
  input clk,
  input reset,
  input [4:0] rd_inp,        //ID-EX
  input Branch_inp,          //Control_unit
  input MemWrite_inp,
  input MemRead_inp,
  input MemtoReg_inp,
  input RegWrite_inp,
```

```
    input [63:0] Adder_B_1,        //Program_Counter
    input [63:0] Result_inp,  //ALU
    input ZERO_inp,
    input [63:0] data_inp,         //Adder
    input [2:0] funct3_Ex,
    input pos_EX,
    input flush,
    output reg [63:0] data_out,
    output reg [63:0] Adder_B_2,
    output reg [4:0] rd_out,
    output reg Branch_out,
    output reg MemWrite_out,
    output reg MemRead_out,
    output reg MemtoReg_out,
    output reg RegWrite_out,
    output reg [63:0] Result_out,
    output reg ZERO_out,
    output reg [2:0] funct3_MEM,
    output reg pos_MEM

);


    always @ (posedge clk or posedge reset)
       begin
         if (reset == 1'b1)
           begin
           Adder_B_2<= 0;
           Result_out <=0;
           ZERO_out <= 0;
             MemtoReg_out <= 0;
               RegWrite_out <= 0;
             Branch_out <= 0;
             MemWrite_out <= 0;
             MemRead_out <= 0;
             rd_out <= 0;
             data_out <= 0;
           funct3_MEM <= 0;
           pos_MEM <= 0;
           end
         else
           begin
             pos_MEM <= pos_EX;
           funct3_MEM <= funct3_Ex;
           Adder_B_2<= Adder_B_1;
           Result_out <= Result_inp;
           ZERO_out <= ZERO_inp ;
             MemtoReg_out <= MemtoReg_inp;
               RegWrite_out <= RegWrite_inp;
             Branch_out <= Branch_inp;
```

```
          MemWrite_out <= MemWrite_inp;
          MemRead_out <= MemRead_inp;
          rd_out <= rd_inp;
          data_out <= data_inp;
        end
        if (flush == 1'b1) begin
          MemtoReg_out <= 0;
            RegWrite_out <= 0;
          Branch_out <= 0;
          MemWrite_out <= 0;
          MemRead_out <= 0;
        end
    end
endmodule
```

Module 3.8 - Forwarding Unit

```
module forwarding_unit(
    input [4:0] rd_WB,
    input [4:0] rd_MEM,
    input [4:0] rs1,
    input [4:0] rs2,
    input RegWrite_WB,
    input RegWrite_MEM,
    output reg [1:0] Forward_A,
    output reg [1:0] Forward_B
);
    always @(*) begin
      if (rs1 == rd_MEM && RegWrite_MEM && rd_MEM != 5'd0 ) begin
        Forward_A <= 2'b10;
      end
      else if (rs1 == rd_WB && RegWrite_WB && rd_WB != 5'd0) begin
        Forward_A <= 2'b01;
      end
      else begin
        Forward_A <= 2'b00;
      end
      if (rs2 == rd_MEM && RegWrite_MEM && rd_MEM != 5'd0) begin
        Forward_B <= 2'b10;
      end
      else if (rs2 == rd_WB && RegWrite_WB && rd_WB != 5'd0) begin
        Forward_B <= 2'b01;
      end
      else begin
        Forward_B <= 2'b00;
      end
    end

endmodule
```

Module 3.9 - Hazard Detection Unit

```
module Hazard_Detection(
    input MemRead_Ex,
    input [4:0] rd_EX,
    input [4:0] rs1_ID,
    input [4:0] rs2_ID,
    output reg IF_ID_Write,
    output reg PC_Write,
    output reg Ctrl
);
always @(*) begin
  if (MemRead_Ex && (rd_EX == rs1_ID || rd_EX == rs2_ID)) begin
    IF_ID_Write <= 1'b0;
    PC_Write <= 1'b0;
    Ctrl <= 1'b1;
  end
  else begin
    IF_ID_Write <= 1'b1;
    PC_Write <= 1'b1;
    Ctrl <= 1'b0;
  end
end


endmodule
```

Module 3.10 - ID/EX Stage Register

```
module ID_EX(
  input clk,
  input reset,
  input [3:0] Funct_inp,        //ALU_Control
  input [1:0] ALUOp_inp,
  input MemtoReg_inp,       //Control_Unit
  input RegWrite_inp,
  input Branch_inp,
  input MemWrite_inp,
  input MemRead_inp,
  input ALUSrc_inp,
  input [63:0] ReadData1_inp,       //registerFile
  input [63:0]ReadData2_inp,
  input [4:0] rd_inp,             //Instruction_Parser
  input [4:0] rs1_in,
  input [4:0] rs2_in,
  input [63:0] imm_data_inp,        //ImmediateDataExtractor
  input [63:0] PC_In,       //Program_Counter
  input [2:0] f3_ID,
  input flush,
  output reg [63:0] PC_Out,
```

```
    output reg [3:0] Funct_out,
    output reg [1:0] ALUOp_out,
    output reg MemtoReg_out,
    output reg RegWrite_out,
    output reg Branch_out,
    output reg MemWrite_out,
    output reg MemRead_out,
    output reg ALUSrc_out,
    output reg [63:0] ReadData1_out,
    output reg [63:0]ReadData2_out,
    output reg [4:0] rs1_out,
    output reg [4:0] rs2_out,
    output reg [4:0] rd_out,
    output reg [63:0] imm_data_out,
    output reg [2:0] f3_EX

);

     always @ (posedge clk or posedge reset)
       begin
         if (reset == 1'b1 )
           begin
             PC_Out<= 0;
             Funct_out <= 0;
             ALUOp_out <= 0;
             MemtoReg_out <= 0;
             RegWrite_out <= 0;
             Branch_out <= 0;
             MemWrite_out <= 0;
             MemRead_out <= 0;
             ALUSrc_out <= 0;
             ReadData1_out <= 0;
             ReadData2_out <= 0;
             rs1_out <= 0;
             rs2_out <= 0;
             rd_out <= 0;
             imm_data_out <= 0;
           f3_EX <= 0;
           end
         else
           begin
             f3_EX <= f3_ID;
             PC_Out<= PC_In;
             Funct_out <= Funct_inp ;
             ALUOp_out <= ALUOp_inp;
             MemtoReg_out <= MemtoReg_inp;
             RegWrite_out <= RegWrite_inp;
             Branch_out <= Branch_inp;
             MemWrite_out <= MemWrite_inp;
             MemRead_out <= MemRead_inp;
```

```
            ALUSrc_out <= ALUSrc_inp;
            ReadData1_out <=  ReadData1_inp;
            ReadData2_out <=  ReadData2_inp;
            rs1_out <= rs1_in;
            rs2_out <= rs2_in;
            rd_out <= rd_inp;
            imm_data_out <= imm_data_inp;
          end
        if (flush == 1'b1) begin
        ALUOp_out <= 0;
        MemtoReg_out <= 0;
        RegWrite_out <= 0;
        Branch_out <= 0;
        MemWrite_out <= 0;
        MemRead_out <= 0;
            ALUSrc_out <= 0;
        end
     end
endmodule
```

Module 3.11 - IF/ID Stage Register

```
module IF_ID(
  input clk,
  input reset,
  input [63:0] PC_In,
  input  [31:0] Inst_input,
  input IF_ID_Write,
  input flush,
  output reg [31:0] Inst_output,
  output reg [63:0] PC_Out

);

  always @ (posedge clk or posedge reset)
    begin
    if (reset == 1'b1)
      begin
         PC_Out <= 64'd0;
         Inst_output <= 32'd0;
      end
    else if (IF_ID_Write != 1'b0)
       begin
         PC_Out <= PC_In;
         Inst_output <= Inst_input;
       end
    if (flush) begin
      Inst_output <= 32'd0;
    end
    end
```

```
endmodule
```

Module 3.12 - Immediate Data Generator

```
module imm_data_gen(
  input [31:0] instruction,
  output reg [63:0] imm_data
);
  always @(instruction) begin
  if (instruction[6:5]==2'b00) begin
  imm_data[11:0] = instruction[31:20];
  end
  else if (instruction[6:5]==2'b01) begin
    imm_data[11:5] = instruction[31:25];

    imm_data[4:0] = instruction[11:7];
  end
  else begin
    imm_data[11] = instruction[31];
    imm_data[9:4] = instruction[30:25];
    imm_data[3:0] = instruction[10:8];
    imm_data[10] = instruction[7];
  end
   imm_data[63:12] = {52{imm_data[11]}};
  end
    endmodule
```

Module 3.13 - Instruction Decoder

```
module instruction(
  input [31:0] ins,
  output [6:0] op,
  output [4:0] rd,
  output [2:0] f3,
  output [4:0] rs1,
  output [4:0] rs2,
  output [6:0] f7

);
  assign op = ins[6:0];
  assign rd = ins[11:7];
  assign f3 = ins[14:12];
  assign rs1 = ins[19:15];
  assign rs2 = ins[24:20];
  assign f7 = ins[31:25];


  endmodule
```

Module 3.14 - Instruction Memory

```
module Instruction_Memory(
  input [63:0] Inst_Address,
  output [31:0] Instruction
);
  reg [7:0] IMem [159:0] ;
  initial begin
    // blt x1 x11 -60
    IMem[159] = 8'b11111100;
    IMem[158] = 8'b10110000;
    IMem[157] = 8'b11000010;
    IMem[156] = 8'b11100011;
    // addi x1 x1 4
    IMem[155] = 8'b00000000;
    IMem[154] = 8'b01000000;
    IMem[153] = 8'b10000000;
    IMem[152] = 8'b10010011;
    // sw x4 4(x6)
    IMem[151] = 8'b00000000;
    IMem[150] = 8'b01000011;
    IMem[149] = 8'b00100010;
    IMem[148] = 8'b00100011;
    // blt x4 x7 -20
    IMem[147] = 8'b11111110;
    IMem[146] = 8'b01110010;
    IMem[145] = 8'b01000110;
    IMem[144] = 8'b11100011;
    // bge x0 x5 8
    IMem[143] = 8'b00000000;
    IMem[142] = 8'b01010000;
    IMem[141] = 8'b01010100;
    IMem[140] = 8'b01100011;
    // lw x7 0(x6)
    IMem[139] = 8'b00000000;
    IMem[138] = 8'b00000011;
    IMem[137] = 8'b00100011;
    IMem[136] = 8'b10000011;
    // add x6 x5 x10
    IMem[135] = 8'b00000000;
    IMem[134] = 8'b10100010;
    IMem[133] = 8'b10000011;
    IMem[132] = 8'b00110011;
    // addi x5 x5 -4
    IMem[131] = 8'b11111111;
    IMem[130] = 8'b11000010;
    IMem[129] = 8'b10000010;
    IMem[128] = 8'b10010011;
    // sw x7 4(x6)
    IMem[127] = 8'b00000000;
    IMem[126] = 8'b01110011;
    IMem[125] = 8'b00100010;
```

```
IMem[124] = 8'b00100011;
// bge x4 x7 28
IMem[123] = 8'b00000000;
IMem[122] = 8'b01110010;
IMem[121] = 8'b01011110;
IMem[120] = 8'b01100011;
// bge x0 x5 32
IMem[119] = 8'b00000010;
IMem[118] = 8'b01010000;
IMem[117] = 8'b01010000;
IMem[116] = 8'b01100011;
// lw x7 0(x6)
IMem[115] = 8'b00000000;
IMem[114] = 8'b00000011;
IMem[113] = 8'b00100011;
IMem[112] = 8'b10000011;
// add x6 x5 x10
IMem[111] = 8'b00000000;
IMem[110] = 8'b10100010;
IMem[109] = 8'b10000011;
IMem[108] = 8'b00110011;
// addi x5 x1 -4
IMem[107] = 8'b11111111;
IMem[106] = 8'b11000000;
IMem[105] = 8'b10000010;
IMem[104] = 8'b10010011;
// lw x4 0(x3)
IMem[103] = 8'b00000000;
IMem[102] = 8'b00000001;
IMem[101] = 8'b10100010;
IMem[100] = 8'b00000011;
// add x3 x1 x10
IMem[99] = 8'b00000000;
IMem[98] = 8'b10100000;
IMem[97] = 8'b10000001;
IMem[96] = 8'b10110011;
// blt x1 x10 4
IMem[95] = 8'b00000000;
IMem[94] = 8'b10100000;
IMem[93] = 8'b11000010;
IMem[92] = 8'b01100011;
// addi x1 x0 4
IMem[91] = 8'b00000000;
IMem[90] = 8'b01000000;
IMem[89] = 8'b00000000;
IMem[88] = 8'b10010011;
// sw x9 36(x10)
IMem[87] = 8'b00000010;
IMem[86] = 8'b10010101;
IMem[85] = 8'b00100010;
```

```
IMem[84] = 8'b00100011;
// addi x9 x0 75
IMem[83] = 8'b00000100;
IMem[82] = 8'b10110000;
IMem[81] = 8'b00000100;
IMem[80] = 8'b10010011;
// sw x9 32(x10)
IMem[79] = 8'b00000010;
IMem[78] = 8'b10010101;
IMem[77] = 8'b00100000;
IMem[76] = 8'b00100011;
// addi x9 x0 31
IMem[75] = 8'b00000001;
IMem[74] = 8'b11110000;
IMem[73] = 8'b00000100;
IMem[72] = 8'b10010011;
// sw x9 28(x10)
IMem[71] = 8'b00000000;
IMem[70] = 8'b10010101;
IMem[69] = 8'b00101110;
IMem[68] = 8'b00100011;
// addi x9 x0 66
IMem[67] = 8'b00000100;
IMem[66] = 8'b00100000;
IMem[65] = 8'b00000100;
IMem[64] = 8'b10010011;
// sw x9 24(x10)
IMem[63] = 8'b00000000;
IMem[62] = 8'b10010101;
IMem[61] = 8'b00101100;
IMem[60] = 8'b00100011;
// addi x9 x0 71
IMem[59] = 8'b00000100;
IMem[58] = 8'b01110000;
IMem[57] = 8'b00000100;
IMem[56] = 8'b10010011;
// sw x9 20(x10)
IMem[55] = 8'b00000000;
IMem[54] = 8'b10010101;
IMem[53] = 8'b00101010;
IMem[52] = 8'b00100011;
// addi x9 x0 46
IMem[51] = 8'b00000010;
IMem[50] = 8'b11100000;
IMem[49] = 8'b00000100;
IMem[48] = 8'b10010011;
// sw x9 16(x10)
IMem[47] = 8'b00000000;
IMem[46] = 8'b10010101;
IMem[45] = 8'b00101000;
```

```
IMem[44] = 8'b00100011;
// addi x9 x0 98
IMem[43] = 8'b00000110;
IMem[42] = 8'b00100000;
IMem[41] = 8'b00000100;
IMem[40] = 8'b10010011;
// sw x9 12(x10)
IMem[39] = 8'b00000000;
IMem[38] = 8'b10010101;
IMem[37] = 8'b00100110;
IMem[36] = 8'b00100011;
// addi x9 x0 63
IMem[35] = 8'b00000011;
IMem[34] = 8'b11110000;
IMem[33] = 8'b00000100;
IMem[32] = 8'b10010011;
// sw x9 4(x10)
IMem[31] = 8'b00000000;
IMem[30] = 8'b10010101;
IMem[29] = 8'b00100100;
IMem[28] = 8'b00100011;
// addi x9 x0 94
IMem[27] = 8'b00000101;
IMem[26] = 8'b11100000;
IMem[25] = 8'b00000100;
IMem[24] = 8'b10010011;
// sw x9 4(x10)
IMem[23] = 8'b00000000;
IMem[22] = 8'b10010101;
IMem[21] = 8'b00100010;
IMem[20] = 8'b00100011;
// addi x9 x0 62
IMem[19] = 8'b00000011;
IMem[18] = 8'b11100000;
IMem[17] = 8'b00000100;
IMem[16] = 8'b10010011;
// sw x9 0(x10)
IMem[15] = 8'b00000000;
IMem[14] = 8'b10010101;
IMem[13] = 8'b00100000;
IMem[12] = 8'b00100011;
// addi x9 x0 18
IMem[11] = 8'b00000001;
IMem[10] = 8'b00100000;
IMem[9]  = 8'b00000100;
IMem[8]  = 8'b10010011;
// addi x11 x0 40
IMem[7]  = 8'b00000010;
IMem[6]  = 8'b10000000;
IMem[5]  = 8'b00000101;
```

```
    IMem[4]  = 8'b10010011;
    // addi x10 x0 0
    IMem[3]  = 8'b00000000;
    IMem[2]  = 8'b00000000;
    IMem[1]  = 8'b00000101;
    IMem[0]  = 8'b00010011;
    //----- Sorting Code ^^ -----//
  end
  assign Instruction[31:0] = {IMem[Inst_Address+2'b11], IMem[Inst_Address
     +2'b10], IMem[Inst_Address+1'b1], IMem[Inst_Address]};

endmodule
```

Module 3.15 - MEM/WB Stage Register

```
// Code your design here
module MEM_WB(
  input clk,
  input reset,
  input [63:0] Result_inp,  //ALU
  input [63:0] Read_Data_inp,       //Data Memory
  input [4:0] rd_inp,       //EX-MEM
  input MemtoReg_inp,       //Control_unit
  input RegWrite_inp,
  output reg MemtoReg_out,
  output reg RegWrite_out,
  output reg [63:0] Result_out,
  output reg [63:0]Read_Data_out,
  output reg [4:0] rd_out

);


  always @ (posedge clk or posedge reset)
    begin
    if (reset == 1'b1)
      begin
        Result_out <= 0;
        Read_Data_out <= 0;
        rd_out <= 5'b0;
        MemtoReg_out <= 0;
        RegWrite_out <= 0;
      end
     else
       begin
        Result_out <= Result_inp;
        Read_Data_out <= Read_Data_inp;
        rd_out <= rd_inp;
        MemtoReg_out <= MemtoReg_inp;
        RegWrite_out <= RegWrite_inp;
```

```
        end
    end
endmodule
```

Module 3.16 - MUX (64-bit, 2 by 1)

```
module MUX(
  input [63:0] X,
  input [63:0] Y,
  input S,
  output [63:0] O
);
  assign O = S?Y:X;
endmodule
```

Module 3.17 - MUX (64-bit, 3 by 1)

```
module MUX_3(
    input [63:0] a,
    input [63:0] b,
    input [63:0] c,
    input [1:0] s,
    output [63:0] out
);
assign out = s[1]?(s[0]?64'bx:c):(s[0]?b:a);

endmodule
```

Module 3.18 - MUX (ALU Control, 16 by 8)

```
module MUX_Control(
    input Ctrl,
    input [1:0] ALUOp,
    input Branch,
    input MemRead,
    input MemtoReg,
    input MemWrite,
    input ALUSrc,
    input RegWrite,
    output [1:0] ALUOp_Out,
    output Branch_Out,
    output MemRead_Out,
    output MemtoReg_Out,
    output MemWrite_Out,
    output ALUSrc_Out,
    output RegWrite_Out
);
assign ALUOp_Out = Ctrl?2'b0:ALUOp;
assign Branch_Out = Ctrl?1'b0:Branch;
assign MemRead_Out = Ctrl?1'b0:MemRead;
assign MemtoReg_Out = Ctrl?1'b0:MemtoReg;
```

```
assign MemWrite_Out = Ctrl?1'b0:MemWrite;
assign ALUSrc_Out = Ctrl?1'b0:ALUSrc;
assign RegWrite_Out = Ctrl?1'b0:RegWrite;

endmodule
```

Module 3.19 - Program Counter

```
module Program_Counter(
  input clk,
  input reset,
  input [63:0] PC_In,
  input PC_Write,
  output reg [63:0] PC_Out
);
  always @(posedge clk or posedge reset)
    begin
      if (reset) begin
        PC_Out <= 64'd0;
      end
      else if (PC_Write != 1'b0) begin
        PC_Out <= PC_In;
      end
      else begin
        PC_Out <= PC_Out; // The most useless line I have ever written
      end
    end
endmodule
```

Module 3.20 - Register File

```
module registerFile(
  input clk,
  input reset,
  input [4:0] rs1,
  input [4:0] rs2,
  input [4:0] rd,
  input [63:0] write_data,
  input reg_write,
  output reg [63:0] readdata1,
  output reg [63:0] readdata2
);
  // reg a = 1'b0;
  reg [63:0] Registers [31:0];
  initial begin
        Registers[0] = 64'd0;
        Registers[1] = 64'd1209;
        Registers[2] = 64'd751;
        Registers[3] = 64'd3522;
        Registers[4] = 64'd2971;
        Registers[5] = 64'd72;
```

```
        Registers[6] = 64'd1135;
        Registers[7] = 64'd1141;
        Registers[8] = 64'd2919;
        Registers[9] = 64'd2467;
        Registers[10] = 64'd0;
        Registers[11] = 64'd3033;
        Registers[12] = 64'd3278;
        Registers[13] = 64'd3214;
        Registers[14] = 64'd3656;
        Registers[15] = 64'd1765;
        Registers[16] = 64'd736;
        Registers[17] = 64'd2985;
        Registers[18] = 64'd2717;
        Registers[19] = 64'd863;
        Registers[20] = 64'd1916;
        Registers[21] = 64'd13;
        Registers[22] = 64'd701;
        Registers[23] = 64'd3479;
        Registers[24] = 64'd2489;
        Registers[25] = 64'd1937;
        Registers[26] = 64'd523;
        Registers[27] = 64'd210;
        Registers[28] = 64'd1043;
        Registers[29] = 64'd425;
        Registers[30] = 64'd2434;
        Registers[31] = 64'd988;
  end
  always @(posedge clk or posedge reg_write or rs1 or rs2 or reset) begin
    if (reg_write & rd != 5'd0) begin
        Registers[rd] = write_data;
    end
    if (reset) begin
      readdata1 = 64'b0;
      readdata2 = 64'b0;
    end
    else begin
      readdata1 = Registers[rs1];
      readdata2 = Registers[rs2];
    end
  end
endmodule
```

Module 3.21 - RISC-V Processor (Top Level Module)

```
`include "Program_Counter.v"
`include "Adder.v"
`include "MUX.v"
`include "Instruction_Memory.v"
`include "IF_ID.v"
`include "ID_EX.v"
```

```
`include "EX_MEM.v"
`include "MEM_WB.v"
`include "instruction.v"
`include "imm_data_gen.v"
`include "registerFile.v"
`include "Control_Unit.v"
`include "ALU_64_bit.v"
`include "ALU_Control.v"
`include "shift_left.v"
`include "branch_module.v"
`include "Data_Memory.v"
`include "MUX_3.v"
`include "forwarding_unit.v"
`include "Hazard_Detection.v"
`include "MUX_Control.v"
module RISC_V_Pipeline(
    input clk,
    input reset
);
wire [63:0] Init_PC_In;
wire [63:0] Init_PC_Out;
wire [63:0] MUX1_Input1;
wire [63:0] MUX1_Input2;
wire [31:0] Instruction_IF;
wire [31:0] Instruction_ID;
wire [63:0] PC_Out_ID;
// wire to_branch;
wire IF_ID_Write;
wire PC_Write;
wire Ctrl;
wire [1:0] ALUOp_Out;
wire Branch_Out;
wire MemRead_Out;
wire MemtoReg_Out;
wire MemWrite_Out;
wire ALUSrc_Out;
wire RegWrite_Out;
wire [6:0] opcode_ID;
wire [4:0] rd_ID;
wire [2:0] f3_ID;
wire [4:0] rs1_ID;
wire [4:0] rs2_ID;
wire [6:0] f7_ID;
wire [63:0] imm_data_ID;
wire [63:0] MUX5_Out;
wire [4:0] rd_WB;
wire [63:0] Read_Data_1_ID;
wire [63:0] Read_Data_2_ID;
wire RegWrite_WB;
wire [1:0] ALUOp_ID;
```

```
wire Branch_ID;
wire MemRead_ID;
wire MemtoReg_ID;
wire MemWrite_ID;
wire ALUSrc_ID;
wire RegWrite_ID;
wire Branch_EX;
wire MemRead_EX;
wire MemtoReg_EX;
wire MemWrite_EX;
wire ALUSrc_EX;
wire RegWrite_EX;
wire [63:0] Read_Data_1_EX;
wire [63:0] Read_Data_2_EX;
wire [63:0] PC_Out_EX;
wire [1:0] ALUOp_EX;
wire [63:0] imm_data_EX;
wire [3:0] Funct_EX;
wire [2:0] f3_EX;
wire [4:0] rs1_EX;
wire [4:0] rs2_EX;
wire [4:0] rd_EX;
wire [3:0] Operation_EX;
wire [63:0] shift_Left_out;
wire [63:0] Branch_Adder_Out_EX;
wire [63:0] MUX_out_EX;
wire [63:0] Result_EX;
wire Zero_EX;
wire pos_EX;
wire RegWrite_MEM;
wire MemtoReg_MEM;
wire MemWrite_MEM;
wire MemRead_MEM;
wire Branch_MEM;
wire Zero_MEM;
wire [63:0] Result_MEM;
wire [63:0] Branch_Adder_Out_MEM;
wire [63:0] Read_Data_2_MEM;
wire [4:0] rd_MEM;
wire pos_MEM;
wire to_branch_MEM;
wire blt_MEM;
wire bge_MEM;
wire bne_MEM;
wire beq_MEM;
wire [2:0] funct3_MEM;
wire [63:0] Read_Data_MEM;
wire MemtoReg_WB;
wire [63:0] Read_Data_WB;
wire [63:0]Result_WB;
```

```
wire [63:0] MUX_A_Out_EX;
wire [63:0] MUX_B_Out_EX;
wire [1:0] F_A;
wire [1:0] F_B;
wire [63:0] Index_0;
wire [63:0] Index_1;
wire [63:0] Index_2;
wire [63:0] Index_3;
wire [63:0] Index_4;
wire [63:0] Index_5;
wire [63:0] Index_6;
wire [63:0] Index_7;
wire [63:0] Index_8;
wire [63:0] Index_9;
Program_Counter p1(clk, reset, Init_PC_In, PC_Write, Init_PC_Out);
Adder a1(Init_PC_Out, 64'd4, MUX1_Input1);
MUX m1(MUX1_Input1,Branch_Adder_Out_MEM, to_branch_MEM,Init_PC_In);
Instruction_Memory i1(Init_PC_Out, Instruction_IF);
IF_ID i2(clk, reset, Init_PC_Out, Instruction_IF, IF_ID_Write,
   to_branch_MEM, Instruction_ID, PC_Out_ID);
Hazard_Detection h1(MemRead_EX, rd_EX, rs1_ID, rs2_ID, IF_ID_Write,
   PC_Write, Ctrl);
instruction i3(Instruction_ID, opcode_ID, rd_ID, f3_ID, rs1_ID, rs2_ID,
   f7_ID);
imm_data_gen i4(Instruction_ID, imm_data_ID);
registerFile r1(clk, reset, rs1_ID, rs2_ID, rd_WB, MUX5_Out, RegWrite_WB,
    Read_Data_1_ID, Read_Data_2_ID);
Control_Unit c1(opcode_ID, ALUOp_ID, Branch_ID, MemRead_ID, MemtoReg_ID,
   MemWrite_ID, ALUSrc_ID, RegWrite_ID);
MUX_Control m6(Ctrl, ALUOp_ID, Branch_ID, MemRead_ID, MemtoReg_ID,
   MemWrite_ID, ALUSrc_ID, RegWrite_ID, ALUOp_Out, Branch_Out,
   MemRead_Out, MemtoReg_Out, MemWrite_Out, ALUSrc_Out, RegWrite_Out);
ID_EX i5(clk, reset, {Instruction_ID[30], Instruction_ID[14:12]},
   ALUOp_Out, MemtoReg_Out, RegWrite_Out, Branch_Out, MemWrite_Out,
   MemRead_Out, ALUSrc_Out, Read_Data_1_ID, Read_Data_2_ID, rd_ID, rs1_ID
   , rs2_ID, imm_data_ID, PC_Out_ID, f3_ID, to_branch_MEM, PC_Out_EX,
   Funct_EX, ALUOp_EX, MemtoReg_EX, RegWrite_EX, Branch_EX, MemWrite_EX,
   MemRead_EX, ALUSrc_EX, Read_Data_1_EX, Read_Data_2_EX, rs1_EX, rs2_EX,
    rd_EX, imm_data_EX, f3_EX);
ALU_Control a2(ALUOp_EX, Funct_EX, Operation_EX);
shift_left s1(imm_data_EX, shift_Left_out);
Adder a3(PC_Out_EX, shift_Left_out, Branch_Adder_Out_EX);
forwarding_unit f1(rd_WB, rd_MEM, rs1_EX, rs2_EX, RegWrite_WB,
   RegWrite_MEM, F_A, F_B);
MUX_3 m3(Read_Data_1_EX, MUX5_Out, Result_MEM, F_A, MUX_A_Out_EX);
MUX_3 m4(Read_Data_2_EX, MUX5_Out, Result_MEM, F_B, MUX_B_Out_EX);
MUX m2(MUX_B_Out_EX, imm_data_EX, ALUSrc_EX, MUX_out_EX);
ALU_64_bit a4(MUX_A_Out_EX, MUX_out_EX, Operation_EX, Zero_EX, Result_EX,
    pos_EX);
EX_MEM e1(clk, reset, rd_EX, Branch_EX, MemWrite_EX, MemRead_EX,
```

```
    MemtoReg_EX, RegWrite_EX, Branch_Adder_Out_EX, Result_EX, Zero_EX,
    MUX_B_Out_EX, f3_EX, pos_EX, to_branch_MEM,  Read_Data_2_MEM,
    Branch_Adder_Out_MEM, rd_MEM, Branch_MEM, MemWrite_MEM, MemRead_MEM,
    MemtoReg_MEM, RegWrite_MEM, Result_MEM, Zero_MEM, funct3_MEM, pos_MEM)
    ;
// Stage 4
branch_module b1(Zero_MEM, pos_MEM, Branch_MEM, funct3_MEM, bne_MEM,
    beq_MEM, bge_MEM, blt_MEM, to_branch_MEM);
Data_Memory d1(clk, Result_MEM, Read_Data_2_MEM, MemWrite_MEM,
    MemRead_MEM, Read_Data_MEM, Index_0, Index_1, Index_2, Index_3,
    Index_4, Index_5, Index_6, Index_7, Index_8, Index_9, funct3_MEM);
MEM_WB m0(clk, reset, Result_MEM, Read_Data_MEM, rd_MEM, MemtoReg_MEM,
    RegWrite_MEM, MemtoReg_WB, RegWrite_WB, Result_WB, Read_Data_WB, rd_WB
    );
MUX m5(Result_WB, Read_Data_WB, MemtoReg_WB, MUX5_Out);

endmodule
```

Module 3.22 - Shift Left

```
module shift_left(
  input [63:0] a ,
  output [63:0] b
);
  assign b ={a[62:0],1'b0};
endmodule
```

Module 3.23 - Test Bench

```
`include "RISC_V_Pipeline.v"
module tb();
  reg dclk;
  reg dreset;

  RISC_V_Pipeline r1(dclk, dreset);
  initial begin
    dclk = 1'b0;
  end
  always begin
    #5
    dclk = ~dclk;
  end
  initial begin
    dreset = 1'b1;
    #10
    dreset = 1'b0;
    #4500
    dreset = 1'b1;
    $finish;
  end
```

```
  initial begin
    $dumpfile("tests.vcd");
    $dumpvars(3,tb);
  end
endmodule
```