

Computer Architecture

Final Project: Report

Habib Shahzad, hs05888

Hammad Ahmed, ha05890

TASK I (Link: [RISC V Processor](#))

A single-cycle processor which can run the bubble sort code

Parsing the instruction

```
1 module instructionParser(  
2     input [31:0] instruction,  
3  
4     output [6:0] opcode,  
5     output [4:0] rd,  
6     output [2:0] funct3,  
7     output [4:0] rs1,  
8     output [4:0] rs2,  
9     output [6:0] funct7  
10 );  
11  
12     assign funct7 = instruction[31:25];  
13     assign rs2 = instruction[24:20];  
14     assign rs1 = instruction[19:15];  
15     assign funct3 = instruction[14:12];  
16     assign rd = instruction[11:7];  
17     assign opcode = instruction[6:0];  
18  
19 endmodule
```

Adder

```
59 module adder (  
60     input [63:0] a,  
61     input [63:0] b,  
62     output reg [63:0] out  
63 );  
64  
65     always @(*)  
66     begin  
67         out = a + b;  
68     end  
69  
70 endmodule
```

Multiplexer

```
1 module mux(A, B, S, Y);  
2  
3     output [63:0] Y;  
4     input S;  
5     input [63:0] A, B;  
6  
7     assign Y = (S ? B:A);  
8  
9 endmodule
```

Program Counter

```
38 module Program_Counter (
39     input clk,
40     input reset,
41     input [63:0] PC_in,
42     output reg [63:0] PC_out
43 );
44
45 always @(posedge clk or posedge reset)
46     begin
47         if (reset)
48             begin
49                 PC_out = 0;
50             end
51         else
52             begin
53                 PC_out = PC_in;
54             end
55     end
56
57 endmodule
```

ALU Control

```
1 module ALU_Control(
2     input [1:0] ALUOp,
3     input [3:0] Funct,
4
5     output reg [3:0] Operation
6 );
7
8     always @(*)
9         begin
10             case ( ALUOp )
11                 2'b00: Operation = 4'b0010 ;
12                 2'b01: Operation = 4'b0110 ;
13                 2'b10:
14                     begin
15                         case (Funct)
16                             4'b0000: Operation = 4'b0010 ;
17                             4'b1000: Operation = 4'b0110 ;
18                             4'b0111: Operation = 4'b0000 ;
19                             4'b0110: Operation = 4'b0001 ;
20                         endcase
21                     end
22                 default: Operation = 4'b0 ;
23             endcase
24         end
25
26 endmodule
```

Immediate Data Extractor

```
1 module immediateGenerator(  
2     input [31:0] instruction,  
3     output reg[63:0] imm_data  
4 );  
5  
6     wire [6:0] opcode;  
7     assign opcode = instruction[6:0];  
8  
9     always @ (*)  
10    begin  
11        case ( opcode[6:5] )  
12  
13            2'b00: imm_data[11:0] = {instruction[31:20]} ;  
14  
15            2'b01: imm_data[11:0] = {instruction[31:25], instruction[11:7]  
16        } ;  
17  
18            2'b11: imm_data[11:0] = {instruction[31], instruction[7],  
19                instruction[30:25], instruction[11:8] } ;  
20  
21            default: imm_data[11:0] = 12'b0;  
22        endcase  
23  
24        imm_data = { {52 { imm_data[11] } } , imm_data[11:0] } ;  
25    end  
endmodule
```

SV/Verilog Design

Control Unit

```
1 module Control_Unit(  
2     input [6:0] opcode,  
3  
4     output reg branch,  
5     output reg MemRead,  
6     output reg MemToReg,  
7     output reg [1:0] ALUOp,  
8     output reg MemWrite,  
9     output reg ALUSrc,  
10    output reg RegWrite  
11 );
```

```
13 always @(opcode)  
14     begin  
15         case ( opcode )  
16             7'b0110011: // R  
17                 begin  
18                     ALUSrc = 1'b0 ;  
19                     MemToReg = 1'b0 ;  
20                     RegWrite = 1'b1 ;  
21                     MemRead = 1'b0 ;  
22                     MemWrite = 1'b0 ;  
23                     branch = 1'b0 ;  
24                     ALUOp = 2'b10;  
25                 end  
26             7'b0000011: // I (ld)  
27                 begin  
28                     ALUSrc = 1'b1 ;  
29                     MemToReg = 1'b1 ;  
30                     RegWrite = 1'b1 ;  
31                     MemRead = 1'b1 ;  
32                     MemWrite = 1'b0 ;  
33                     branch = 1'b0 ;  
34                     ALUOp = 2'b00;  
35                 end  
36             7'b0100011: // I (sd)  
37                 begin  
38                     ALUSrc = 1'b1 ;  
39                     MemToReg = 1'bx ;  
40                     RegWrite = 1'b0 ;  
41                     MemRead = 1'b0 ;  
42                     MemWrite = 1'b1 ;  
43                     branch = 1'b0 ;  
44                     ALUOp = 2'b00;  
45                 end  
46             7'b1100011: // S (beq)  
47                 begin  
48                     ALUSrc = 1'b0 ;  
49                     MemToReg = 1'bx ;  
50                     RegWrite = 1'b0 ;  
51                     MemRead = 1'b0 ;  
52                     MemWrite = 1'b0 ;  
53                     branch = 1'b1 ;  
54                     ALUOp = 2'b01;  
55                 end  
56             7'b0010011: // I (addi)  
57                 begin  
58                     ALUSrc = 1'b1 ;  
59                     MemToReg = 1'b0 ;  
60                     RegWrite = 1'b1 ;  
61                     MemRead = 1'b0 ;  
62                     MemWrite = 1'b0 ;  
63                     branch = 1'b0 ;  
64                     ALUOp = 2'b00;  
65                 end  
66             default:  
67                 begin  
68                     ALUSrc = 1'b0 ;  
69                     MemToReg = 1'b0 ;  
70                     RegWrite = 1'b0 ;  
71                     MemRead = 1'b0 ;  
72                     MemWrite = 1'b0 ;  
73                     branch = 1'b0 ;  
74                     ALUOp = 2'b0 ;  
75                 end  
76             endcase  
77         end  
78     endmodule
```

```
46             7'b1100011: // S (beq)  
47                 begin  
48                     ALUSrc = 1'b0 ;  
49                     MemToReg = 1'bx ;  
50                     RegWrite = 1'b0 ;  
51                     MemRead = 1'b0 ;  
52                     MemWrite = 1'b0 ;  
53                     branch = 1'b1 ;  
54                     ALUOp = 2'b01;  
55                 end  
56             7'b0010011: // I (addi)  
57                 begin  
58                     ALUSrc = 1'b1 ;  
59                     MemToReg = 1'b0 ;  
60                     RegWrite = 1'b1 ;  
61                     MemRead = 1'b0 ;  
62                     MemWrite = 1'b0 ;  
63                     branch = 1'b0 ;  
64                     ALUOp = 2'b00;  
65                 end  
66             default:  
67                 begin  
68                     ALUSrc = 1'b0 ;  
69                     MemToReg = 1'b0 ;  
70                     RegWrite = 1'b0 ;  
71                     MemRead = 1'b0 ;  
72                     MemWrite = 1'b0 ;  
73                     branch = 1'b0 ;  
74                     ALUOp = 2'b0 ;  
75                 end  
76             endcase  
77         end  
78     endmodule
```

Bubble Sort Code (Where A is an array)

```
for i in range(len(A)):
    for j in range(i, len(A)):
        if A[i] < A[j]:
            temp = A[i]
            A[i] = A[j]
            A[j] = temp
```

CONVERTING THE CODE INTO RISC-V →

OuterLoop:

beq x27, x11, Exit

InnerLoop:

add x21 x10 x5

add x22 x10 x6

ld x13 0(x21) #a[j]

ld x14 0(x22) #a[i]

blt x14 x13 swap

x:

addi x28, x28, 1

addi x5 x5 4

blt x28, x11, InnerLoop

ResetInnerLoop:

addi x27, x27, 1

add x28, x27, x0

addi x6 x6 4

add x5 x0 x6

beq x0, x0, OuterLoop

swap:

add x30 x0 x14

sd x13 0(x22)

sd x30 0(x21)

beq x0 x0 x

Exit:

Hex Codes of our instructions

04bd8463

00550ab3

00650b33

000ab683

000b3703

02d74263

001e0e13

00828293

febe42e3

001d8d93

000d8e33

00830313

006002b3

fc0006e3

00e00f33

00db3023

01eab023

fc000ae3

Helping Us Display the Verilog code:

```
bins = []
for hexCode in hexCodes:
    bins.append( convertToBinary(hexCode) )

bins = bins[::-1]
def printVerilogCode():
    k = (len(bins)*4) - 1
    print(f'\treg [7:0] memory [{k}:0];')
    print('\tinitial begin')
    for i in range(len(bins)):
        x = bins[i]
        a = x[0:8]
        b = x[8:16]
        c = x[16:24]
        d = x[24:32]
        print ( f"\t memory[{k-0}] = 8'b{a};")
        print ( f"\t memory[{k-1}] = 8'b{b};")
        print ( f"\t memory[{k-2}] = 8'b{c};")
        print ( f"\t memory[{k-3}] = 8'b{d};")
        print()
        k -= 4
    print('\tend')
printVerilogCode()
```

Instruction Memory

```

1 module Instruction_Memory (
2     input [63:0] Inst_Address,
3     output reg [31:0] Instruction
4 );

```

```

6 reg [7:0] memory [71:0];
7     initial begin
8         memory[71] = 8'b11111100;
9         memory[70] = 8'b00000000;
10        memory[69] = 8'b00001010;
11        memory[68] = 8'b11100011;
12
13        memory[67] = 8'b00000001;
14        memory[66] = 8'b11101010;
15        memory[65] = 8'b10110000;
16        memory[64] = 8'b00100011;
17
18        memory[63] = 8'b00000000;
19        memory[62] = 8'b11011011;
20        memory[61] = 8'b00110000;
21        memory[60] = 8'b00100011;
22
23        memory[59] = 8'b00000000;
24        memory[58] = 8'b11100000;
25        memory[57] = 8'b00001111;
26        memory[56] = 8'b00110011;
27
28        memory[55] = 8'b11111100;
29        memory[54] = 8'b00000000;
30        memory[53] = 8'b00000110;
31        memory[52] = 8'b11100011;
32
33        memory[51] = 8'b00000000;
34        memory[50] = 8'b01100000;
35        memory[49] = 8'b00000010;
36        memory[48] = 8'b10110011;
37
38        memory[47] = 8'b00000000;
39        memory[46] = 8'b10000011;
40        memory[45] = 8'b00000011;
41        memory[44] = 8'b00010011;

```

```

43        memory[43] = 8'b00000000;
44        memory[42] = 8'b00001101;
45        memory[41] = 8'b10001110;
46        memory[40] = 8'b00110011;
47
48        memory[39] = 8'b00000000;
49        memory[38] = 8'b00011101;
50        memory[37] = 8'b10001101;
51        memory[36] = 8'b10010011;
52
53        memory[35] = 8'b11111110;
54        memory[34] = 8'b10111110;
55        memory[33] = 8'b01000010;
56        memory[32] = 8'b11100011;
57
58        memory[31] = 8'b00000000;
59        memory[30] = 8'b10000010;
60        memory[29] = 8'b10000010;
61        memory[28] = 8'b10010011;
62
63        memory[27] = 8'b00000000;
64        memory[26] = 8'b00011110;
65        memory[25] = 8'b00001110;
66        memory[24] = 8'b00010011;
67
68        memory[23] = 8'b00000010;
69        memory[22] = 8'b11010111;
70        memory[21] = 8'b01000010;
71        memory[20] = 8'b01100011;
72
73        memory[19] = 8'b00000000;
74        memory[18] = 8'b00001011;
75        memory[17] = 8'b00110111;
76        memory[16] = 8'b00000011;
77

```

```

78        memory[15] = 8'b00000000;
79        memory[14] = 8'b00001010;
80        memory[13] = 8'b10110110;
81        memory[12] = 8'b10000011;
82
83        memory[11] = 8'b00000000;
84        memory[10] = 8'b01100101;
85        memory[9] = 8'b00001011;
86        memory[8] = 8'b00110011;
87
88        memory[7] = 8'b00000000;
89        memory[6] = 8'b01010101;
90        memory[5] = 8'b00001010;
91        memory[4] = 8'b10110011;
92
93        memory[3] = 8'b00000100;
94        memory[2] = 8'b10111101;
95        memory[1] = 8'b10000100;
96        memory[0] = 8'b01100011;
97
98     end

```

```

105 always @(*)
106     begin
107         Instruction = {
108             memory[Inst_Address+3],
109             memory[Inst_Address+2],
110             memory[Inst_Address+1],
111             memory[Inst_Address+0]
112         };
113     end
114 endmodule

```


Data Memory

```
1 module Data_Memory (  
2   input clk,  
3   input MemWrite,  
4   input [63:0] Mem_Addr,  
5   input MemRead,  
6   input [63:0] Write_Data,  
7   output reg [63:0] Read_Data  
8 );
```

Storing elements in the memory (lines 19-26) and storing elements from memory to an array and displaying the array in the log window (line 47).

```
10 reg [63:0] array [7:0];  
11 reg [7:0] memory [63:0];  
12 int i;  
13  
14 initial begin  
15   for (i = 0; i < 64; i = i + 1)  
16     begin  
17       memory[i] = 0;  
18     end  
19   memory[0] = 1;  
20   memory[8] = 2;  
21   memory[16] = 3;  
22   memory[24] = 4;  
23   memory[32] = 5;  
24   memory[40] = 6;  
25   memory[48] = 7;  
26   memory[56] = 8;  
27 end  
28 int k;  
29 always @(*) begin  
30   k = 0;  
31   for (i = 0; i < 8; i = i + 1)  
32     begin  
33       array[i] = {  
34         memory[k+7],  
35         memory[k+6],  
36         memory[k+5],  
37         memory[k+4],  
38         memory[k+3],  
39         memory[k+2],  
40         memory[k+1],  
41         memory[k+0]  
42       };  
43       k = k + 8;  
44     end  
45 end
```

```
47 always @(*) $display ("%p", array);  
48  
49 always @(posedge clk)  
50   begin  
51     if (MemWrite == 1)  
52       begin  
53         memory[Mem_Addr+7] = Write_Data[63:56];  
54         memory[Mem_Addr+6] = Write_Data[55:48];  
55         memory[Mem_Addr+5] = Write_Data[47:40];  
56         memory[Mem_Addr+4] = Write_Data[39:32];  
57         memory[Mem_Addr+3] = Write_Data[31:24];  
58         memory[Mem_Addr+2] = Write_Data[23:16];  
59         memory[Mem_Addr+1] = Write_Data[15:8];  
60         memory[Mem_Addr+0] = Write_Data[7:0];  
61       end  
62     end  
63   always @(*)  
64     begin  
65       if (MemRead == 1)  
66         begin  
67           Read_Data = {  
68             memory[Mem_Addr+7],  
69             memory[Mem_Addr+6],  
70             memory[Mem_Addr+5],  
71             memory[Mem_Addr+4],  
72             memory[Mem_Addr+3],  
73             memory[Mem_Addr+2],  
74             memory[Mem_Addr+1],  
75             memory[Mem_Addr+0]  
76           };  
77         end  
78     end  
79 endmodule
```

Register File (storing the length in 11th register)

```
1 module registerFile(  
2     input [4:0] rs1,  
3     input [4:0] rs2,  
4     input [4:0] rd,  
5     input [63:0] WriteData,  
6     output reg [63:0] ReadData1,  
7     output reg [63:0] ReadData2,  
8     input clk,  
9     input reset,  
10    input RegWrite  
11 );
```

```
12    reg [63:0] registers [31:0];  
13    int i;  
14    initial  
15        begin  
16            for (i = 0; i < 32; i+=1)  
17                begin  
18                    registers[i] = 0;  
19                end  
20            registers[11] = 8;  
21        end  
22  
23    always @ (*)  
24        begin  
25            if (reset == 1)  
26                begin  
27                    ReadData1 = 64'b0;  
28                    ReadData2 = 64'b0;  
29                end  
30            else  
31                begin  
32                    ReadData1 = registers[rs1];  
33                    ReadData2 = registers[rs2];  
34                end  
35        end  
36    always @ (posedge clk)  
37        begin  
38            if (RegWrite)  
39                begin  
40                    registers[rd] = WriteData;  
41                end  
42        end  
43 endmodule
```

ALU 64 bit

```
1 module ALU_64_bit (  
2     input [63:0] a,  
3     input [63:0] b,  
4     input [3:0] ALUOp,  
5     output reg [63:0] result,  
6     input operation,  
7     output reg branching  
8 );
```

```
9     always @ (a or b or ALUOp)  
10     begin  
11         case (ALUOp)  
12             4'b0000: result = a & b;    // AND  
13             4'b0001: result = a | b;    // OR  
14             4'b0010: result = a + b;    // add  
15             4'b0110: result = a - b;    // sub  
16             4'b1100: result = ~(a | b); // NOR  
17         endcase  
18     end
```

```
19     int bool;  
20     always@(*)  
21     begin  
22         case (operation)  
23             1'b0:    // beq  
24             begin  
25                 branching = (result == 0) ? 1:0;  
26             end  
27             1'b1:    // blt  
28             begin  
29                 bool = (a < b);  
30                 branching = bool ? 1:0;  
31             end  
32             default: branching = 0;  
33         endcase  
34     end  
35  
36 endmodule
```

This was the most difficult (yet the easiest thing) we found in task 1. We essentially make the control signal named as branching either low or high depending on the instruction. We determine the instruction type by the seeing the input signal named as operation (which is essentially the most significant bit of funct3). If this signal is low, then we check if a and b are equal (beq) and if we if it low, we check if a is less than b (blt).

Declaring the wires

```
19  wire [63:0] PC_Out;
20
21  wire [63:0] add_out1;
22  wire [63:0] add_out2;
23
24  wire [31:0] instruction;
25  wire [63:0] imm_data;
26
27  wire [63:0] Read_Data;
28  wire [63:0] Read_Data1;
29  wire [63:0] Read_Data2;
30
31  wire [3: 0] Operation;
32  wire [63:0] Result;
33
34  wire [6:0] opcode;
35  wire [4:0] rd;
36  wire [2:0] funct3;
37  wire [4:0] rs1;
38  wire [4:0] rs2;
39  wire [6:0] funct7;
40
41  wire branch;
42  wire MemRead;
43  wire MemToReg;
44  wire [1:0] ALUOp;
45  wire MemWrite;
46  wire ALUSrc;
47  wire RegWrite;
48
49  wire [63:0] mux_out1;
50  wire [63:0] mux_out2;
51  wire [63:0] mux_out3;
52
53  wire branching;
```

Making the connections

```
55 Program_Counter pc(  
56     .PC_in(mux_out1),  
57     .reset(reset),  
58     .clk(clk),  
59     .PC_out(PC_Out)  
60 );  
61  
62 Instruction_Memory im(  
63     .Inst_Address(PC_Out),  
64     .Instruction(instruction)  
65 );  
66  
67 adder a1(  
68     .a(PC_Out),  
69     .b(64'd4),  
70     .out(add_out1)  
71 );  
72  
73 adder a2(  
74     .a(PC_Out),  
75     .b(imm_data << 1),  
76     .out(add_out2)  
77 );  
78  
79 instructionParser ip(  
80     .instruction(instruction),  
81     .opcode(opcode),  
82     .rd(rd),  
83     .funct3(funct3),  
84     .rs1(rs1),  
85     .rs2(rs2),  
86     .funct7(funct7)  
87 );  
88
```

```
89 registerFile rf(  
90     .clk(clk),  
91     .reset(reset),  
92     .rs1(rs1),  
93     .rs2(rs2),  
94     .rd(rd),  
95     .WriteData(mux_out2),  
96     .RegWrite(RegWrite),  
97     .ReadData1(Read_Data1),  
98     .ReadData2(Read_Data2)  
99 );  
100  
101 Control_Unit cu(  
102     .opcode(opcode),  
103     .branch(branch),  
104     .MemRead(MemRead),  
105     .MemToReg(MemToReg),  
106     .ALUOp(ALUOp),  
107     .MemWrite(MemWrite),  
108     .ALUSrc(ALUSrc),  
109     .RegWrite(RegWrite)  
110 );  
111  
112 ALU_64_bit alu(  
113     .a(Read_Data1),  
114     .b(mux_out3),  
115     .ALUOp(Operation),  
116     .result(Result),  
117     .branching(branching),  
118     .operation(funct3[2])  
119 );  
120
```

Making some more connections

```
121     ALU_Control aluc(  
122         .ALUOp(ALUOp),  
123         .Funct({instruction[30],instruction[14:12]}),  
124         .Operation(Operation)  
125     );  
126  
127     mux m1(  
128         .A(add_out1),  
129         .B(add_out2),  
130         .S(branch & branching),  
131         .Y(mux_out1)  
132     );  
133  
134     mux m2(  
135         .A(Result),  
136         .B(Read_Data),  
137         .S(MemToReg),  
138         .Y(mux_out2)  
139     );  
140  
141     mux m3(  
142         .A(Read_Data2),  
143         .B(imm_data),  
144         .S(ALUSrc),  
145         .Y(mux_out3)  
146     );  
147  
148     immediateGenerator ig(  
149         .instruction(instruction),  
150         .imm_data(imm_data)  
151     );
```

```
152  
153     Data_Memory dm(  
154         .clk(clk),  
155         .Mem_Addr(Result),  
156         .Write_Data(Read_Data2),  
157         .MemWrite(MemWrite),  
158         .MemRead(MemRead),  
159         .Read_Data(Read_Data)  
160     );  
161
```

Simulation (LOG)

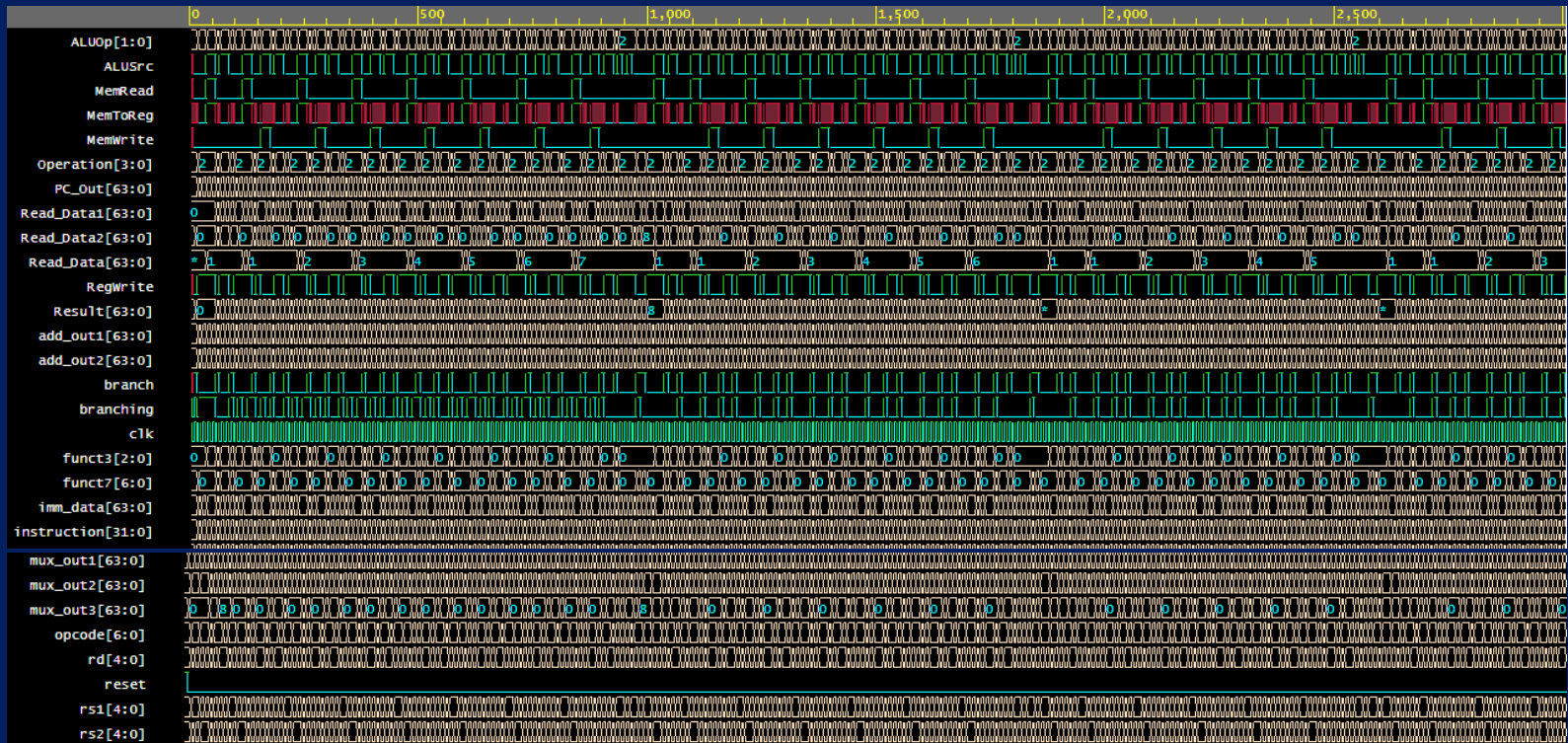
```
# '{8, 7, 6, 5, 4, 3, 2, 1}'  
# '{8, 7, 6, 5, 4, 3, 2, 2}'  
# '{8, 7, 6, 5, 4, 3, 1, 2}'  
# '{8, 7, 6, 5, 4, 3, 1, 3}'  
# '{8, 7, 6, 5, 4, 2, 1, 3}'  
# '{8, 7, 6, 5, 4, 2, 1, 4}'  
# '{8, 7, 6, 5, 3, 2, 1, 4}'  
# '{8, 7, 6, 5, 3, 2, 1, 5}'  
# '{8, 7, 6, 4, 3, 2, 1, 5}'  
# '{8, 7, 6, 4, 3, 2, 1, 6}'  
# '{8, 7, 5, 4, 3, 2, 1, 6}'  
# '{8, 7, 5, 4, 3, 2, 1, 7}'  
# '{8, 6, 5, 4, 3, 2, 1, 7}'  
# '{8, 6, 5, 4, 3, 2, 1, 8}'  
# '{7, 6, 5, 4, 3, 2, 1, 8}'  
# '{7, 6, 5, 4, 3, 2, 2, 8}'  
# '{7, 6, 5, 4, 3, 1, 2, 8}'  
# '{7, 6, 5, 4, 3, 1, 3, 8}'  
# '{7, 6, 5, 4, 2, 1, 3, 8}'  
# '{7, 6, 5, 4, 2, 1, 4, 8}'  
# '{7, 6, 5, 3, 2, 1, 4, 8}'  
# '{7, 6, 5, 3, 2, 1, 5, 8}'  
# '{7, 6, 4, 3, 2, 1, 5, 8}'  
# '{7, 6, 4, 3, 2, 1, 6, 8}'  
# '{7, 5, 4, 3, 2, 1, 6, 8}'  
# '{7, 5, 4, 3, 2, 1, 7, 8}'  
# '{6, 5, 4, 3, 2, 1, 7, 8}'
```

```
# '{6, 5, 4, 3, 2, 2, 7, 8}'  
# '{6, 5, 4, 3, 1, 2, 7, 8}'  
# '{6, 5, 4, 3, 1, 3, 7, 8}'  
# '{6, 5, 4, 2, 1, 3, 7, 8}'  
# '{6, 5, 4, 2, 1, 4, 7, 8}'  
# '{6, 5, 3, 2, 1, 4, 7, 8}'  
# '{6, 5, 3, 2, 1, 5, 7, 8}'  
# '{6, 4, 3, 2, 1, 5, 7, 8}'  
# '{6, 4, 3, 2, 1, 6, 7, 8}'  
# '{5, 4, 3, 2, 1, 6, 7, 8}'  
# '{5, 4, 3, 2, 2, 6, 7, 8}'  
# '{5, 4, 3, 1, 2, 6, 7, 8}'  
# '{5, 4, 3, 1, 3, 6, 7, 8}'  
# '{5, 4, 2, 1, 3, 6, 7, 8}'  
# '{5, 4, 2, 1, 4, 6, 7, 8}'  
# '{5, 3, 2, 1, 4, 6, 7, 8}'  
# '{5, 3, 2, 1, 5, 6, 7, 8}'  
# '{4, 3, 2, 1, 5, 6, 7, 8}'  
# '{4, 3, 2, 2, 5, 6, 7, 8}'  
# '{4, 3, 1, 2, 5, 6, 7, 8}'  
# '{4, 3, 1, 3, 5, 6, 7, 8}'  
# '{4, 2, 1, 3, 5, 6, 7, 8}'  
# '{4, 2, 1, 4, 5, 6, 7, 8}'  
# '{3, 2, 1, 4, 5, 6, 7, 8}'  
# '{3, 2, 2, 4, 5, 6, 7, 8}'  
# '{3, 1, 2, 4, 5, 6, 7, 8}'
```

Sorted array →

```
# '{3, 1, 3, 4, 5, 6, 7, 8}'  
# '{2, 1, 3, 4, 5, 6, 7, 8}'  
# '{2, 2, 3, 4, 5, 6, 7, 8}'  
# '{1, 2, 3, 4, 5, 6, 7, 8}'  
# exit
```

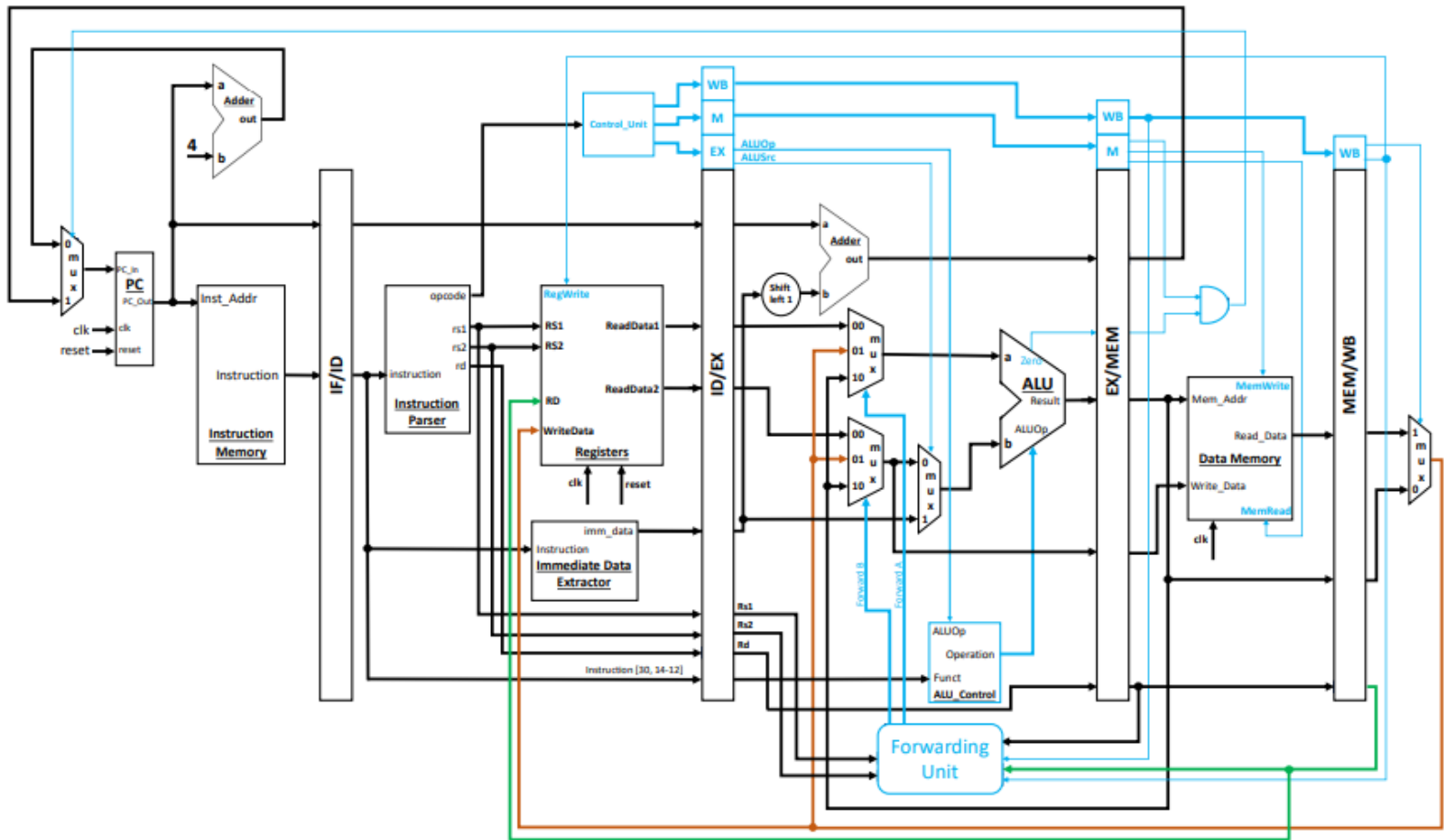
EMP wave



TASK I (Link: [RISC V Processor](#))

TASK (Link: [Pipelined RISC Processor](#))

A pipelined processor based on the following diagram



The following modules (IFID, IDEX, EXMEM, MEMVWB) essentially just sets the output equal to the input at the positive edge of the clock.

Instruction Fetch/Instruction Decode

```
1 module IFID (  
2     input clk, reset,  
3     input [31:0] instructionIn,  
4     input [63:0] addressIn,  
5  
6     output reg [31:0] instructionOut,  
7     output reg [63:0] addressOut  
8 );
```

```
10     always @(posedge clk)  
11         begin  
12             if (reset)  
13                 begin  
14                     instructionOut = 0;  
15                     addressOut     = 0;  
16                 end
```

```
17             else  
18                 begin  
19                     instructionOut = instructionIn;  
20                     addressOut     = addressIn;  
21                 end  
22             end  
23  
24 endmodule
```

Instruction Decode/Execute

```
1 module IDEX (
2   input clk,reset,
3   input MemRead,
4   input MemtoReg,
5   input MemWrite,
6   input ALUSrc,
7   input RegWrite,
8   input branch,
9   input [63:0] ReadData1,
10  input [63:0] ReadData2,
11  input [63:0] imm_data,
12  input [63:0] address,
13  input [3:0] funct,
14  input [4:0] rd,
15  input [4:0] rs1,
16  input [4:0] rs2,
17  input [1:0] AluOp,
18
19  output reg MemReadOut,
20  output reg MemtoRegOut,
21  output reg MemWriteOut,
22  output reg ALUSrcOut,
23  output reg RegWriteOut,
24  output reg branchOut,
25  output reg [63:0] ReadData1Out,
26  output reg [63:0] ReadData2Out,
27  output reg [63:0] imm_dataOut,
28  output reg [63:0] addressOut,
29  output reg [3:0] functOut,
30  output reg [4:0] rdOut,
31  output reg [4:0] rs1Out,
32  output reg [4:0] rs2Out,
33  output reg [1:0] AluOpOut
34 );
```

```
36 always @(posedge clk )
37   begin
38     if (reset)
39       begin
40         MemReadOut    = 0;
41         MemtoRegOut   = 0;
42         MemWriteOut   = 0;
43         ALUSrcOut     = 0;
44         RegWriteOut   = 0;
45         branchOut     = 0;
46         ReadData1Out  = 0;
47         ReadData2Out  = 0;
48         imm_dataOut   = 0;
49         addressOut    = 0;
50         functOut      = 0;
51         rdOut         = 0;
52         rs1Out        = 0;
53         rs2Out        = 0;
54         AluOpOut      = 0;
55       end
```

```
56     else
57       begin
58         MemReadOut    = MemRead;
59         MemtoRegOut   = MemtoReg;
60         MemWriteOut   = MemWrite;
61         ALUSrcOut     = ALUSrc;
62         RegWriteOut   = RegWrite;
63         branchOut     = branch;
64         ReadData1Out  = ReadData1;
65         ReadData2Out  = ReadData2;
66         imm_dataOut   = imm_data;
67         addressOut    = address;
68         functOut      = funct;
69         rdOut         = rd;
70         rs1Out        = rs1;
71         rs2Out        = rs2;
72         AluOpOut      = AluOp;
73       end
74     end
75 endmodule
```

Execute/Memory Access

```
2 module EXMEM(  
3   input clk,reset,  
4   input MemRead,  
5   input memToReg,  
6   input MemWrite,  
7   input regWrite,  
8   input branch,  
9   input [63:0] WriteData,  
10  input [63:0] add2,  
11  input [4:0] rd,  
12  input [63:0] AluResult,  
13  input zero,  
14  
15  output reg MemWriteOut,  
16  output reg MemReadOut,  
17  output reg memToRegOut,  
18  output reg regWriteOut,  
19  output reg branchOut,  
20  output reg [63:0] WriteDataOut,  
21  output reg [63:0] add2Out,  
22  output reg [4:0] rdOut,  
23  output reg [63:0] AluResultOut,  
24  output reg zeroOut  
25 );
```

```
27 always @(posedge clk )  
28   begin  
29     if (reset)  
30       begin  
31         MemWriteOut = 0;  
32         MemReadOut = 0;  
33         memToRegOut = 0;  
34         regWriteOut = 0;  
35         branchOut = 0;  
36         WriteDataOut = 0;  
37         add2Out = 0;  
38         rdOut = 0;  
39         AluResultOut = 0;  
40         zeroOut = 0;  
41       end
```

```
42     else  
43       begin  
44         MemReadOut = MemRead;  
45         memToRegOut = memToReg;  
46         MemWriteOut = MemWrite;  
47         regWriteOut = regWrite;  
48         branchOut = branch;  
49         WriteDataOut = WriteData;  
50         add2Out = add2;  
51         rdOut = rd;  
52         AluResultOut = AluResult;  
53         zeroOut = zero;  
54       end  
55     end  
56 endmodule
```

Memory Access/Write back

```
1 module MEMWB (  
2     input clk,reset,  
3     input MemToReg,  
4     input RegWrite,  
5     input [63:0] ReadData,  
6     input [63:0] AluResult,  
7     input [4:0] rd,  
8  
9     output reg MemToRegOut,  
10    output reg RegWriteOut,  
11    output reg [63:0] ReadDataOut ,  
12    output reg [63:0] AluResultOut,  
13    output reg [4:0]rdOut  
14 );
```

```
16     always @(posedge clk)  
17         begin  
18             if (reset)  
19                 begin  
20                     MemToRegOut  = 0;  
21                     RegWriteOut  = 0;  
22                     ReadDataOut  = 0;  
23                     AluResultOut = 0;  
24                     rdOut       = 0;  
25                 end
```

```
26         else  
27             begin  
28                 MemToRegOut  = MemToReg;  
29                 RegWriteOut  = RegWrite;  
30                 ReadDataOut  = ReadData;  
31                 AluResultOut = AluResult;  
32                 rdOut       = rd;  
33             end  
34         end  
35     endmodule
```

Forwarding Unit (pseudocode referenced from the book)

```
1 module forwardingUnit (
2     input [4:0] rs1,
3     input [4:0] rs2,
4
5     input [4:0] exRd, // EXMEM RD
6     input [4:0] wbRd, //MEMWB RD
7
8     input exRegWrite, //EXMEM REGWRITE
9     input wbRegWrite, //MEMWB REGWRITE
10
11     output reg [1:0] forwardA,
12     output reg [1:0] forwardB
13 ).
14
15 always @(*)
16 begin
17     if ( (exRd == rs1) & (exRegWrite != 0 & exRd !=0))
18         forwardA = 2'b10;
19     else
20         begin
21             if (
22                 (wbRd == rs1) &
23                 (wbRegWrite != 0 & wbRd != 0) &
24                 ~((exRd == rs1) & (exRegWrite != 0 & exRd !=0) )
25             )
26                 forwardA = 2'b01;
27             else
28                 forwardA = 2'b00;
29         end
30     if (
31         (exRd == rs2) &
32         (exRegWrite != 0 & exRd !=0)
33     )
34         forwardB = 2'b10;
35     else
36         begin
37             if (
38                 (wbRd == rs2) &
39                 (wbRegWrite != 0 & wbRd != 0) &
40                 ~((exRegWrite != 0 & exRd !=0 ) &
41                     (exRd == rs2) )
42             )
43                 forwardB = 2'b01;
44             else
45                 forwardB = 2'b00;
46         end
47     end
48 endmodule
```

Declaring the wires (Now we will have some additional wires for each stage)

```
27 wire [63:0] PC_Out;
28 wire [63:0] IFIDPC_Out;
29 wire [63:0] IDEXPC_Out;
30
31 wire [63:0] add_out1;
32 wire [63:0] add_out2;
33 wire [63:0] EXMEMadd_out2;
34
35 wire [31:0] instruction;
36 wire [31:0] IFIDinstruction;
37
38 wire [63:0] imm_data;
39 wire [63:0] IDEXimm_data;
40
41 wire [63:0] ReadData;
42
43 wire [63:0] Read_Data1;
44 wire [63:0] IDEXRead_Data1;
45
46 wire [63:0] Read_Data2;
47 wire [63:0] IDEXRead_Data2;
48
49 wire [3: 0] operation;
50
51 wire [63:0] Result;
52 wire [63:0] EXMEMResult;
53 wire [63:0] MEMWBResult;
54
55 wire [4:0] rd;
56 wire [4:0] IDEXrd;
57 wire [4:0] EXMEMrd;
58 wire [4:0] MEMWBrd;
```

```
60 wire [6:0] opcode;
61 wire [2:0] funct3;
62 wire [6:0] funct7;
63
64 wire [4:0] rs1;
65 wire [4:0] IDEXrs1;
66
67 wire [4:0] rs2;
68 wire [4:0] IDEXrs2;
69
70 wire [3:0] IDEXfunct;
71
72 wire branch;
73 wire IDEXbranch;|
74
75 wire MemRead;
76 wire IDEXMemRead;
77
78 wire MemToReg;
79 wire IDEXMemToReg;
80 wire MEMWBMemToReg;
81
82 wire [1:0] ALUOp;
83 wire [1:0] IDEXAluOp;
84
85 wire MemWrite;
86 wire IDEXMemWrite;
87
88 wire ALUSrc;
89 wire IDEXALUSrc;
90
91 wire RegWrite;
92 wire IDEXRegWrite;
93 wire EXMEMRegWrite;
94 wire MEMWBRegWrite;
```

```
96 wire [63:0] mux_out1;
97 wire [63:0] MEMWBmux_out1;
98
99 wire [63:0] mux_out2;
100 wire [63:0] EXMEMmux_out2;
101 wire [63:0] MEMWBmux_out2;
102
103 wire [63:0] mux_out3;
104
105 wire [63:0] threemux_out1;
106
107 wire [63:0] threemux_out2;
108 wire [63:0] EXMEMthreemux_out2;
109
110 wire zero;
111 wire EXMEMzero;
112
113 wire [1:0] forwardA;
114 wire [1:0] forwardB;
```

Making the connections

```
117 Program_Counter pc (  
118     .PC_in(mux_out1),  
119     .clk(clk),  
120     .reset(reset),  
121     .PC_out(PC_Out)  
122 );  
123  
124 Instruction_Memory im (  
125     .Inst_Address(PC_Out),  
126     .Instruction(instruction)  
127 );  
128  
129 adder a1 (  
130     .a(PC_Out),  
131     .b(64'd4),  
132     .out(add_out1)  
133 );  
134  
135 adder a2 (  
136     .a(IDEXPC_Out),  
137     .b(IDEXimm_data << 1),  
138     .out(add_out2)  
139 );
```

```
152 instructionParser ip (  
153     .instruction(IFIDinstruction),  
154     .opcode(opcode),  
155     .rd(rd),  
156     .funct3(funct3),  
157     .rs1(rs1),  
158     .rs2(rs2),  
159     .funct7(funct7)  
160 );  
161  
162  
163 registerFile rf (  
164     .clk(clk),  
165     .reset(reset),  
166     .rs1(rs1),  
167     .rs2(rs2),  
168     .rd(MEMWBrd),  
169     .WriteData(mux_out2),  
170     .RegWrite(MEMWBRegWrite),  
171     .ReadData1(Read_Data1),  
172     .ReadData2(Read_Data2)  
173 );
```

```
215 Control_Unit cu (  
216     .opcode(opcode),  
217     .branch(branch),  
218     .MemRead(MemRead),  
219     .MemToReg(MemToReg),  
220     .MemWrite(MemWrite),  
221     .ALUSrc(ALUSrc),  
222     .RegWrite(RegWrite),  
223     .ALUOp(ALUOp)  
224 );  
225  
226  
227 ALU_64_bit alu64 (  
228     .a(threemux_out1),  
229     .b(mux_out3),  
230     .ALUOp(operation),  
231     .result(Result),  
232     .branching(zero),  
233     .operation(funct3[2])  
234 );  
235  
236  
237 ALU_Control ac (  
238     .ALUOp(IDEXAluOp),  
239     .Funct(IDEXfunct),  
240     .Operation(operation)  
241 );
```


Making the connections

```
243 mux mu1 (  
244     .A(add_out1),  
245     .B(EXMEMadd_out2),  
246     .S(EXMEMzero & EXMEMbranch),  
247     .Y(mux_out1)  
248 );  
249  
250 mux m2 (  
251     .A(MEMWBmux_out2),  
252     .B(MEMWBmux_out1),  
253     .S(MEMWBMemToReg),  
254     .Y(mux_out2)  
255 );  
256  
257 mux m3(  
258     .A(threemux_out2),  
259     .B>IDEXimm_data,  
260     .S>IDEXAluSrc,  
261     .Y(mux_out3)  
262 );  
263  
264 threemux m31 (  
265     .a>IDEXRead_Data1,  
266     .b(mux_out2),  
267     .c(EXMEMResult),  
268     .S(forwardA),  
269     .out(threemux_out1)  
270 );  
271  
272 threemux m32(  
273     .a>IDEXRead_Data2,  
274     .b(mux_out2),  
275     .c(EXMEMResult),  
276     .S(forwardB),  
277     .out(threemux_out2)  
278 );
```

```
282 immediateGenerator ig (  
283     .instruction(IFIDinstruction),  
284     .imm_data(imm_data)  
285 );  
286  
287  
288 Data_Memory dm (  
289     .Write_Data(EXMEMmux_out2),  
290     .Mem_Addr(EXMEMResult),  
291     .MemWrite(EXMEMMemWrite),  
292     .clk(clk),  
293     .MemRead(EXMEMMemRead),  
294     .Read_Data(ReadData)  
295 );  
296
```

```
341 forwardingUnit fu (  
342     .rs1>IDEXrs1,  
343     .rs2>IDEXrs2,  
344     .exRd(EXMEMrd),  
345     .wbRd(MEMWBrd),  
346     .wbRegWrite(MEMWBRegWrite),  
347     .exRegWrite(EXMEMRegWrite),  
348     .forwardA(forwardA), .forwardB(forwardB)  
349 );
```

Making the connections

```
141 IFID idif (
142     .clk(clk),
143     .reset(reset),
144
145     .instructionIn(instruction),
146     .addressIn(PC_Out),
147
148     .instructionOut(IFIDinstruction),
149     .addressOut(IFIDPC_Out)
150 );
```

```
177 IDEX exid (
178     .clk(clk),
179     .reset(reset),
180
181     .branch(branch),
182     .ALUSrc(ALUSrc),
183     .RegWrite(RegWrite),
184     .MemRead(MemRead),
185     .MemtoReg(MemtoReg),
186     .MemWrite(MemWrite),
187     .funct({IFIDinstruction[30],IFIDinstruction[14:12]}),
188     .address(IFIDPC_Out),
189     .ReadData1(Read_Data1),
190     .ReadData2(Read_Data2),
191     .imm_data(imm_data),
192     .rs1(rs1),
193     .rs2(rs2),
194     .rd(rd),
195     .ALUOp(ALUOp),
196
197     .branchOut(IDEXbranch),
198     .MemReadOut(IDEXMemRead),
199     .MemtoRegOut(IDEXMemtoReg),
200     .MemWriteOut(IDEXMemWrite),
201     .RegWriteOut(IDEXRegWrite),
202     .ALUSrcOut(IDEXALUSrc),
203     .addressOut(IDEXPC_Out),
204     .rs1Out(IDEXrs1),
205     .rs2Out(IDEXrs2),
206     .rdOut(IDEXrd),
207     .imm_dataOut(IDEXimm_data),
208     .ReadData1Out(IDEXRead_Data1),
209     .ReadData2Out(IDEXRead_Data2),
210     .functOut(IDEXfunct),
211     .ALUOpOut(IDEXALUOp)
212 );
```

```
297 EXMEM memex(
298     .clk(clk),
299     .reset(reset),
300     .add2(add_out2),
301     .AluResult(Result),
302     .zero(zero),
303     .WriteData(threemux_out2),
304     .rd(IDEXrd),
305     .branch(IDEXbranch),
306     .MemRead(IDEXMemRead),
307     .memToReg(IDEXMemtoReg),
308     .MemWrite(IDEXMemWrite),
309     .regWrite(IDEXRegWrite),
310
311     .add2Out( EXMEMadd_out2),
312     .zeroOut(EXMEMzero),
313     .AluResultOut(EXMEMResult),
314     .WriteDataOut(EXMEMmux_out2),
315     .rdOut(EXMEMrd),
316     .branchOut(EXMEMbranch),
317     .MemReadOut(EXMEMMemRead),
318     .memToRegOut(EXMEMMEMMemtoReg),
319     .MemWriteOut(EXMEMMemWrite),
320     .regWriteOut(EXMEMRegWrite)
321 );
```

```
324 MEMWB mwb (
325     .clk(clk),
326     .reset(reset),
327
328     .ReadData(ReadData),
329     .AluResult(EXMEMResult),
330     .rd(EXMEMrd),
331     .MemtoReg(EXMEMMEMMemtoReg),
332     .RegWrite(EXMEMRegWrite),
333
334     .ReadDataOut(MEMWBmux_out1),
335     .AluResultOut(MEMWBmux_out2),
336     .rdOut(MEMWBrd),
337     .MemtoRegOut(MEMWBMemtoReg),
338     .RegWriteOut(MEMWBRegWrite)
339 );
```

Hex Codes of our instructions we are going to check

```
add x25 x26 x27
add x26 x20 x23
add x23 x21 x27
add x27 x22 x26
```



```
01bd0cb3
017a0d33
01ba8bb3
01ab0db3
```

```
add x25 x26 x27
add x26 x25 x23
add x23 x27 x26
add x27 x23 x26
```



```
01bd0cb3
017c8d33
01ad8bb3
01ab8db3
```

Venus simulation

```
addi x23 x0 3
addi x25 x0 9
addi x26 x0 4
addi x27 x0 6

add x25 x26 x27
add x26 x20 x23
add x23 x21 x27
add x27 x22 x26
```



s7 (x23)	6
s8 (x24)	0
s9 (x25)	10
s10 (x26)	3
s11 (x27)	3

```
addi x23 x0 3
addi x25 x0 9
addi x26 x0 4
addi x27 x0 6

add x25 x26 x27
add x26 x25 x23
add x23 x27 x26
add x27 x23 x26
```



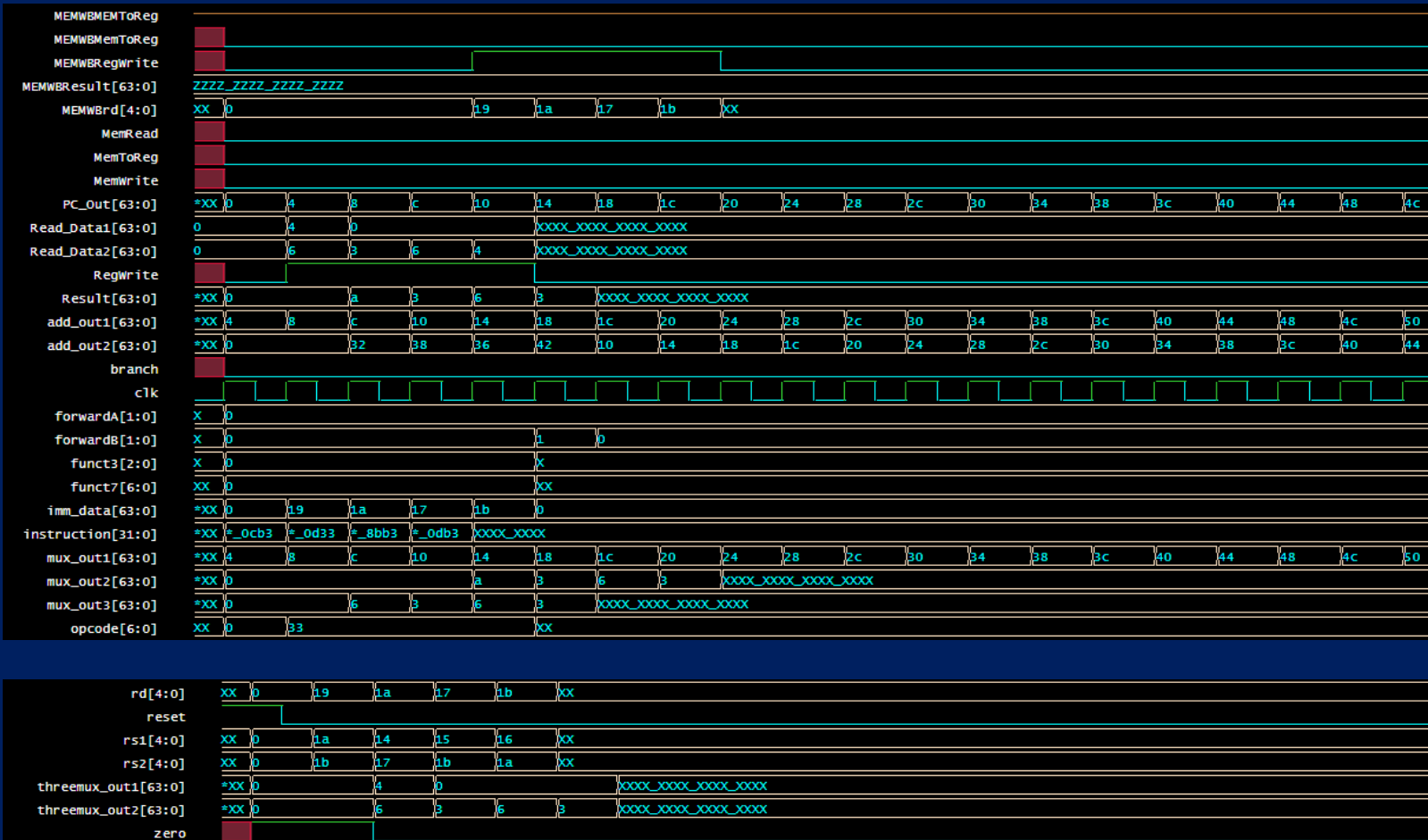
s7 (x23)	19
s8 (x24)	0
s9 (x25)	10
s10 (x26)	13
s11 (x27)	32

Verilog simulation: We essentially display our registers array to view the changes being made. As we can observe the following registers, they successfully match with our venus simulation registers. (x23 – x27) for both the set of instructions we checked. We made the necessary changes in data memory and instruction memory to incorporate our instructions in the processor

```
# '{0, 0, 0, 0, 6, 4, 9, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
# '{0, 0, 0, 0, 6, 4, 10, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
# '{0, 0, 0, 0, 6, 3, 10, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
# '{0, 0, 0, 0, 6, 3, 10, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
# '{0, 0, 0, 0, 3, 3, 10, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

# '{0, 0, 0, 0, 6, 4, 9, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
# '{0, 0, 0, 0, 6, 4, 10, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
# '{0, 0, 0, 0, 6, 13, 10, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
# '{0, 0, 0, 0, 6, 13, 10, 0, 19, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
# '{0, 0, 0, 0, 32, 13, 10, 0, 19, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```





Link: [Pipelined RISC Processor](#)

TASK 3 (Link: [Pipelined RISC Processor Task 3](#))

Stalling unit

```
413 module stallingUnit (  
414     input IDEXMemRead,  
415     input [4:0] IDEXrd,  
416     input [4:0] IFIDrs1,  
417     input [4:0] IFIDrs2,  
418     output reg stall  
419 );  
420  
421 always @(*)  
422 if (IDEXMemRead &  
423     ((IDEXrd == IFIDrs1) |  
424     (IDEXrd == IFIDrs2)))  
425     stall = 1;  
426 else  
427     stall = 0;  
428 endmodule
```

Flushing logic

```
398 module flushing (  
399     input EXMEMbranch,  
400     output reg flush  
401 );  
402  
403     always @(*)  
404     begin  
405         if (EXMEMbranch)  
406             flush = 1;  
407         else  
408             flush = 0;  
409     end  
410 endmodule
```

First in the modules (IFID, IDEX, EXMEM, MEMWB) we were making the input equal to the output in the positive edge of the clock and when reset was not zero. Now we will also say do not do this operation if the flush signal is high.

For stalling, we will say, when stalling is high, we do not make the PC_Out equal to PC_in in the program counter module. We also make the control signals equal to zero when stalling needs to be done as can be seen in the picture below.

Flushing is high when the branch after being anded is 1.

Stalling is high when IDEXrd = IFIDrs1 and IDEXrd = IFIDrs2 and IDEXmemRead is high

```

377     assign IDEXMemToReg = stall ? 0 : MemToReg ;
378     assign IDEXRegWrite = stall ? 0 : RegWrite ;
379     assign IDEXbranch   = stall ? 0 : branch   ;
380     assign IDEXMemWrite = stall ? 0 : MemWrite ;
381     assign IDEXMemRead  = stall ? 0 : MemRead  ;
382     assign IDEXALUSrc   = stall ? 0 : ALUSrc   ;
383     assign IDEXAluOp     = stall ? 0 : ALUOp    ;

```

```

1 module Program_Counter (
2     input clk,
3     input reset,
4     input stall,
5     input [63:0] PC_in,
6     output reg [63:0] PC_out
7 );
8
9     always @(posedge clk)
10         begin
11             if (reset | stall)
12                 begin
13                     PC_out = 0;
14                 end
15             else
16                 begin
17                     if (!stall)
18                         begin
19                             PC_out = PC_in;
20                         end
21                 end
22             end
23         end
24     end
25
26 endmodule

```

We will alter our pipeline modules according to this

```

17     always @(posedge clk)
18         begin
19             if (reset | flush)
20                 begin
21                     MemToRegOut = 0;
22                     RegWriteOut = 0;
23                     ReadDataOut = 0;
24                     AluResultOut = 0;
25                     rdOut = 0;
26                 end
27             else
28                 begin
29                     MemToRegOut = MemToReg;
30                     RegWriteOut = RegWrite;
31                     ReadDataOut = ReadData;
32                     AluResultOut = AluResult;
33                     rdOut = rd;
34                 end
35             end
36         endmodule

```

According to our logic everything in task3 is working but the flush signal is being high wrongly; We may alter our code in the link attached for task3.

[THIS PAGE IS INTENTIONALLY LEFT BLANK]