Computer Architecture

Final Project: Report

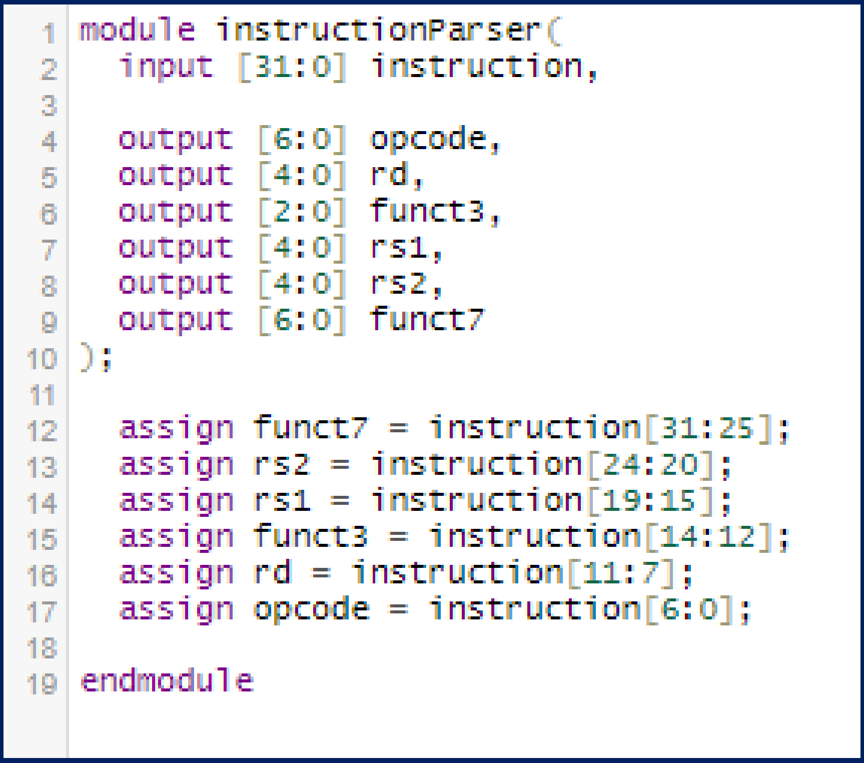
Habib Shahzad, hs05888

Hammad Ahmed, ha05890

TASK 1 (Link: RISC_V Processor)

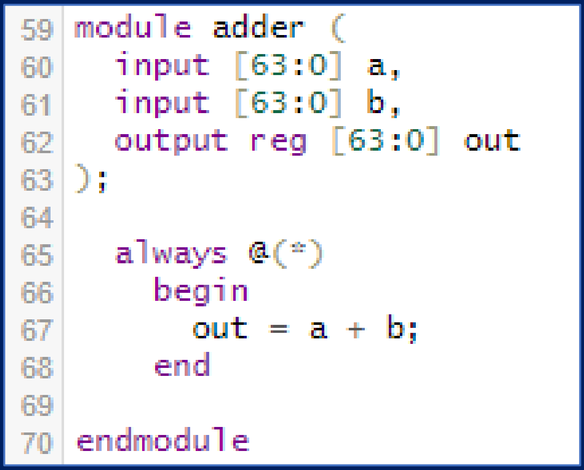
A single-cycle processor which can run the bubble sort code

Parsing the instruction



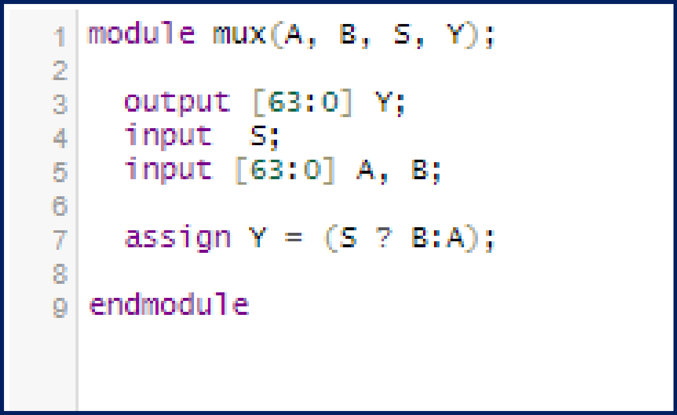
```
module instructionParser(
  input [31:0] instruction,

  output [6:0] opcode,
  output [4:0] rd,
  output [2:0] funct3,
  output [4:0] rs1,
  output [4:0] rs2,
  output [6:0] funct7
);

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

endmodule
```

Adder



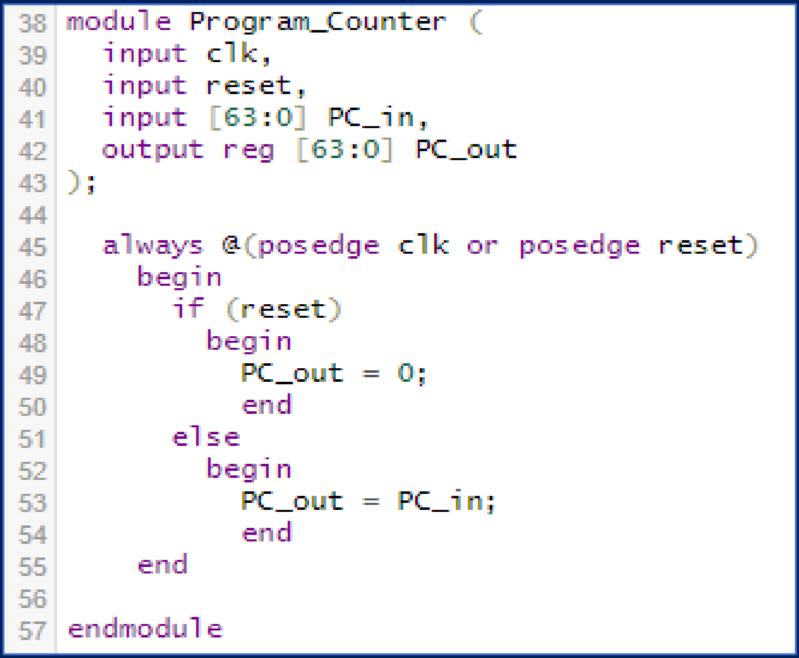
```
module adder (
  input [63:0] a,
  input [63:0] b,
  output reg [63:0] out
);

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

endmodule
```

Multiplexer



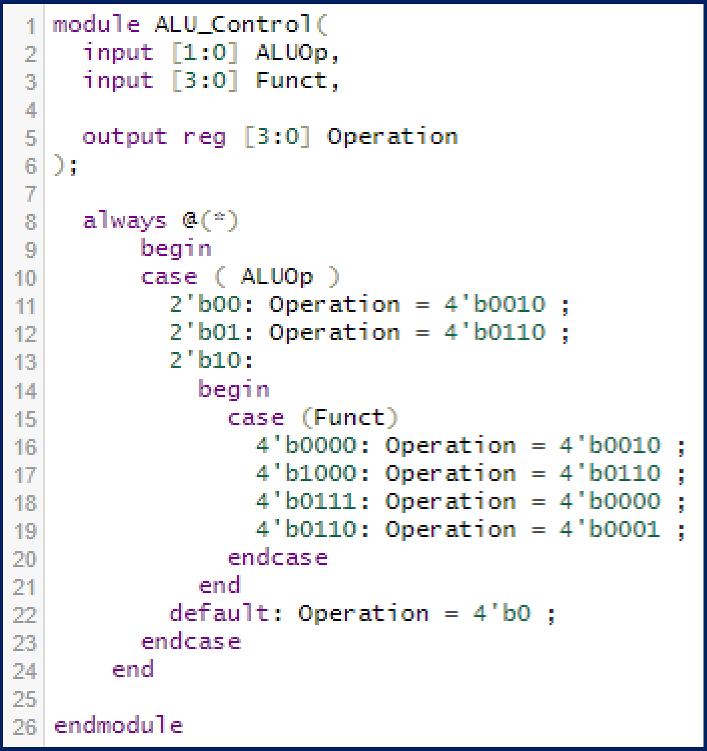
```
module mux(A, B, S, Y);

  output [63:0] Y;
  input  S;
  input [63:0] A, B;

  assign Y = (S ? B:A);

endmodule
```

Program Counter



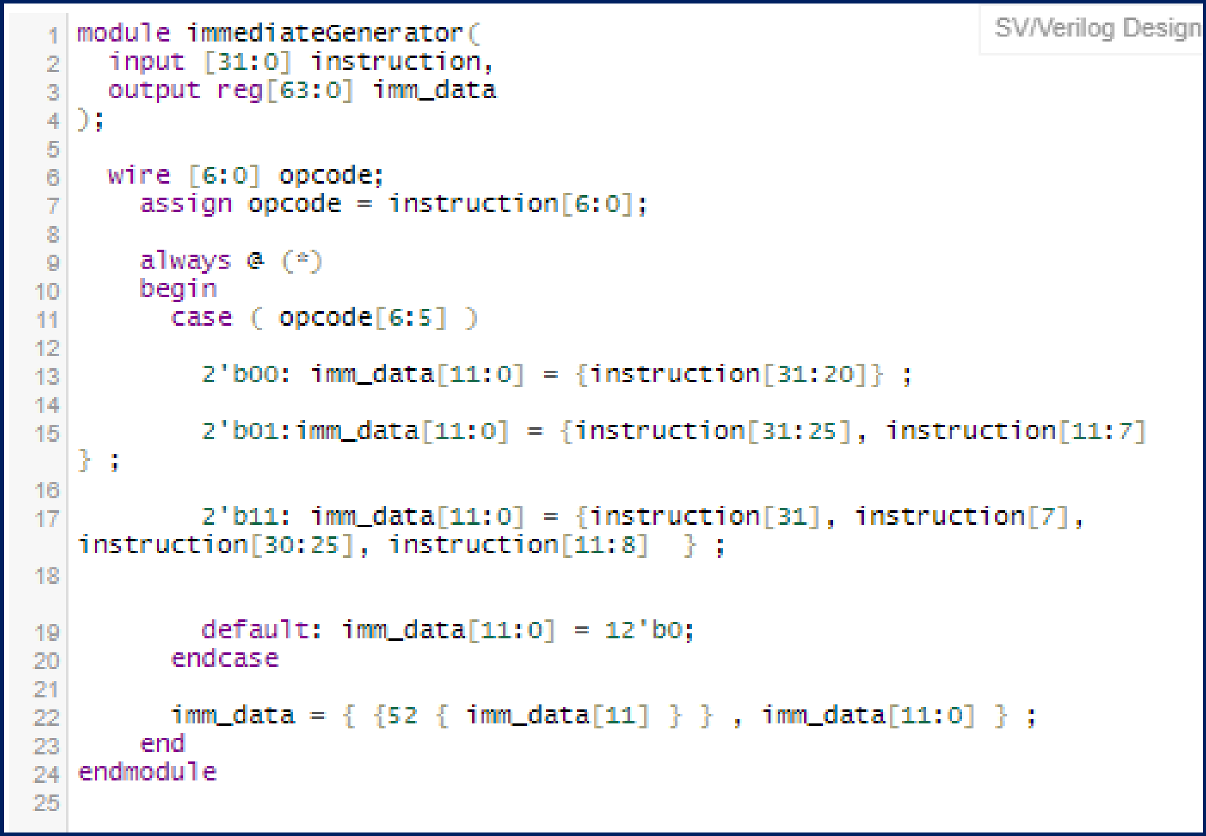
```
module Program_Counter (
  input clk,
  input reset,
  input [63:0] PC_in,
  output reg [63:0] PC_out
);

  always @(posedge clk or posedge reset)
    begin
      if (reset)
        begin
          PC_out = 0;
          end
      else
        begin
          PC_out = PC_in;
          end
    end

endmodule
```

ALU Control



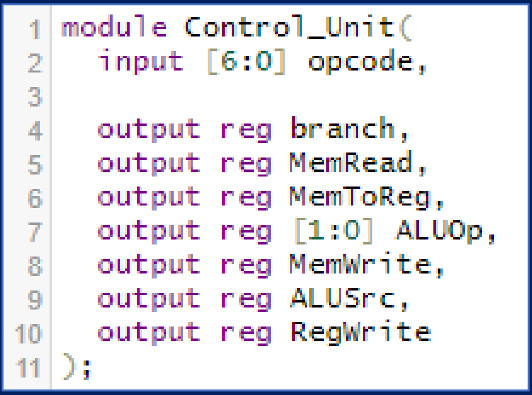
```
module ALU_Control(
  input [1:0] ALUOp,
  input [3:0] Funct,

  output reg [3:0] Operation
);

  always @(*)
      begin
      case ( ALUOp )
        2'b00: Operation = 4'b0010 ;
        2'b01: Operation = 4'b0110 ;
        2'b10:
          begin
            case (Funct)
              4'b0000: Operation = 4'b0010 ;
              4'b1000: Operation = 4'b0110 ;
              4'b0111: Operation = 4'b0000 ;
              4'b0110: Operation = 4'b0001 ;
            endcase
          end
        default: Operation = 4'b0 ;
      endcase
    end

endmodule
```

Immediate Data Extractor

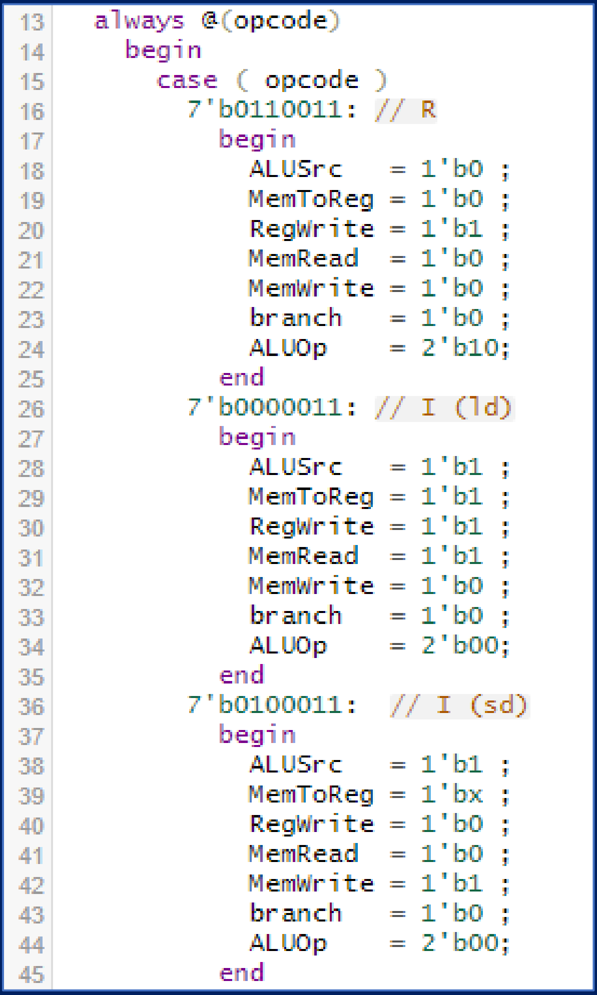


```verilog
module immediateGenerator(
  input [31:0] instruction,
  output reg[63:0] imm_data
);

  wire [6:0] opcode;
    assign opcode = instruction[6:0];

    always @ (*)
    begin
      case ( opcode[6:5] )

        2'b00: imm_data[11:0] = {instruction[31:20]} ;

        2'b01:imm_data[11:0] = {instruction[31:25], instruction[11:7] } ;

        2'b11: imm_data[11:0] = {instruction[31], instruction[7], instruction[30:25], instruction[11:8]  } ;


        default: imm_data[11:0] = 12'b0;
      endcase

      imm_data = { {52 { imm_data[11] } } , imm_data[11:0] } ;
    end
endmodule

```

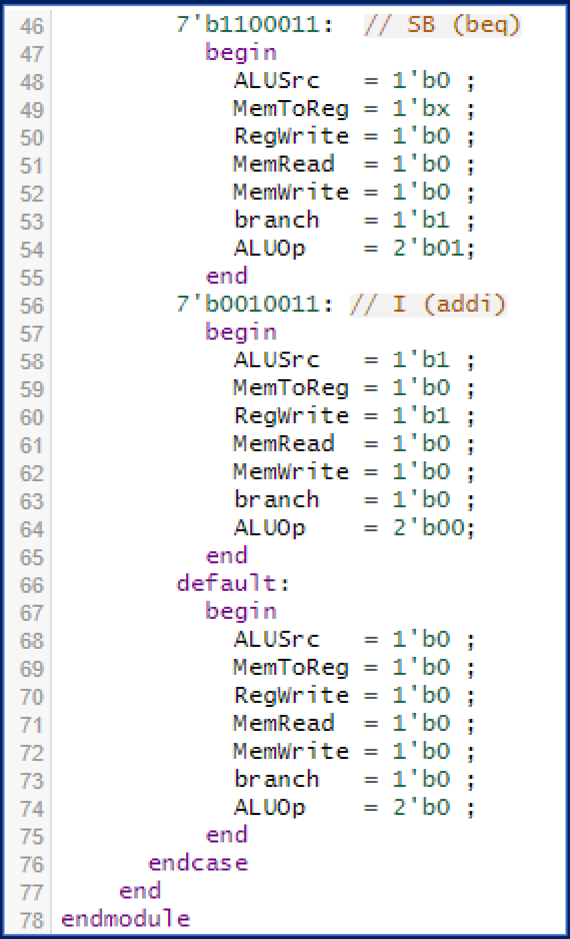
Control Unit



```verilog
module Control_Unit(
  input [6:0] opcode,

  output reg branch,
  output reg MemRead,
  output reg MemToReg,
  output reg [1:0] ALUOp,
  output reg MemWrite,
  output reg ALUSrc,
  output reg RegWrite
);
```

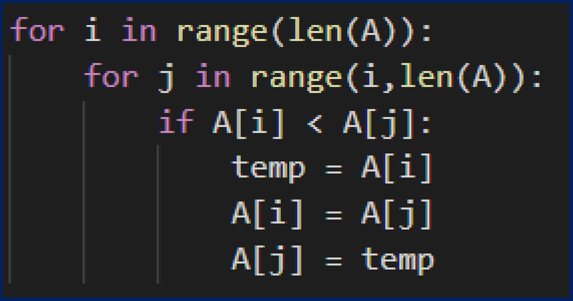


```verilog
  always @(opcode)
    begin
      case ( opcode )
        7'b0110011: // R
          begin
            ALUSrc   = 1'b0 ;
            MemToReg = 1'b0 ;
            RegWrite = 1'b1 ;
            MemRead  = 1'b0 ;
            MemWrite = 1'b0 ;
            branch   = 1'b0 ;
            ALUOp    = 2'b10;
          end
        7'b0000011: // I (ld)
          begin
            ALUSrc   = 1'b1 ;
            MemToReg = 1'b1 ;
            RegWrite = 1'b1 ;
            MemRead  = 1'b1 ;
            MemWrite = 1'b0 ;
            branch   = 1'b0 ;
            ALUOp    = 2'b00;
          end
        7'b0100011:  // I (sd)
          begin
            ALUSrc   = 1'b1 ;
            MemToReg = 1'bx ;
            RegWrite = 1'b0 ;
            MemRead  = 1'b0 ;
            MemWrite = 1'b1 ;
            branch   = 1'b0 ;
            ALUOp    = 2'b00;
          end
```



```verilog
        7'b1100011:  // SB (beq)
          begin
            ALUSrc   = 1'b0 ;
            MemToReg = 1'bx ;
            RegWrite = 1'b0 ;
            MemRead  = 1'b0 ;
            MemWrite = 1'b0 ;
            branch   = 1'b1 ;
            ALUOp    = 2'b01;
          end
        7'b0010011: // I (addi)
          begin
            ALUSrc   = 1'b1 ;
            MemToReg = 1'b0 ;
            RegWrite = 1'b1 ;
            MemRead  = 1'b0 ;
            MemWrite = 1'b0 ;
            branch   = 1'b0 ;
            ALUOp    = 2'b00;
          end
        default:
          begin
            ALUSrc   = 1'b0 ;
            MemToReg = 1'b0 ;
            RegWrite = 1'b0 ;
            MemRead  = 1'b0 ;
            MemWrite = 1'b0 ;
            branch   = 1'b0 ;
            ALUOp    = 2'b0 ;
          end
      endcase
    end
endmodule
```

Bubble Sort Code
(Where A is an array)



```python
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
```

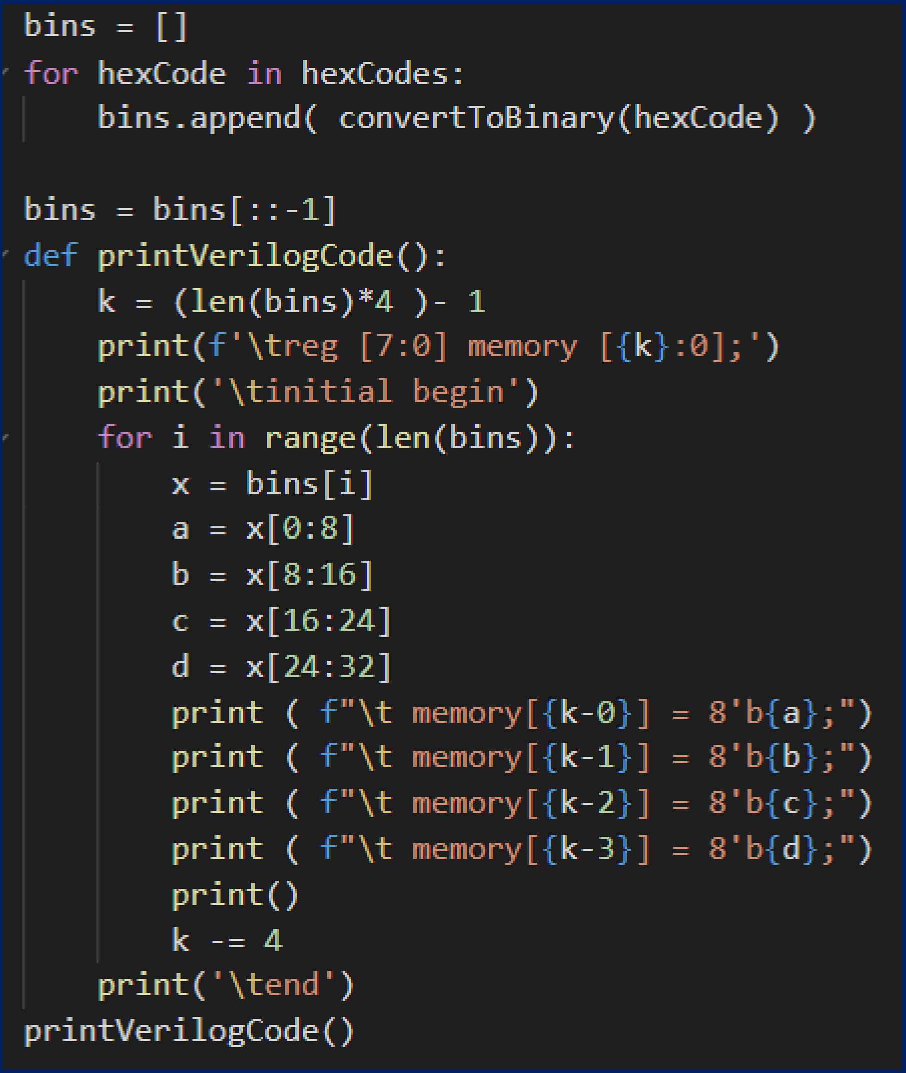
CONVERTING THE CODE INTO RISC-V →

```
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

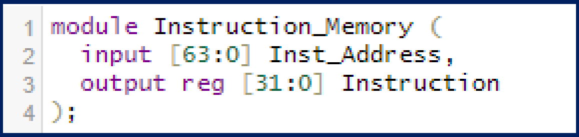
```
04bd8463
00550ab3
00650b33
000ab683
000b3703
02d74263
001e0e13
00828293
febe42e3
001d8d93
000d8e33
00830313
006002b3
fc0006e3
00e00f33
00db3023
01eab023
fc000ae3
```

Helping Us Display the Verilog code:

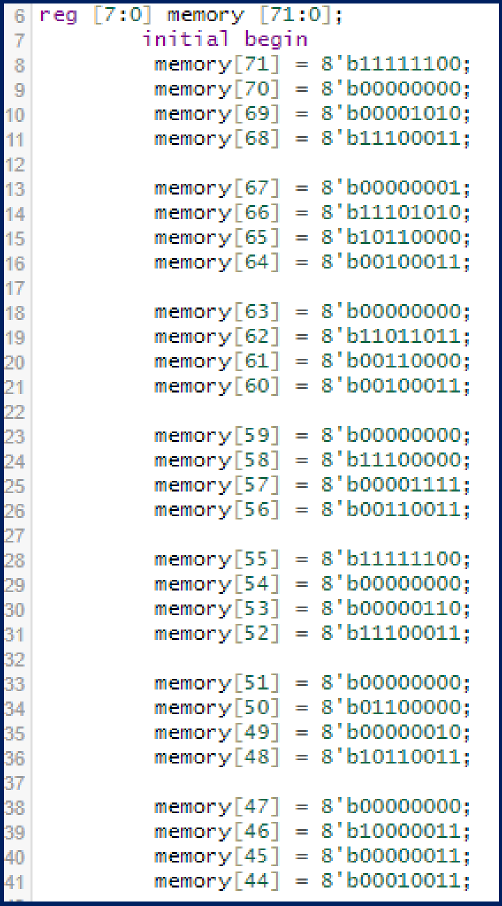


```python
bins = []
for hexCode in hexCodes:
    bins.append( convertToBinary(hexCode) )

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()
```

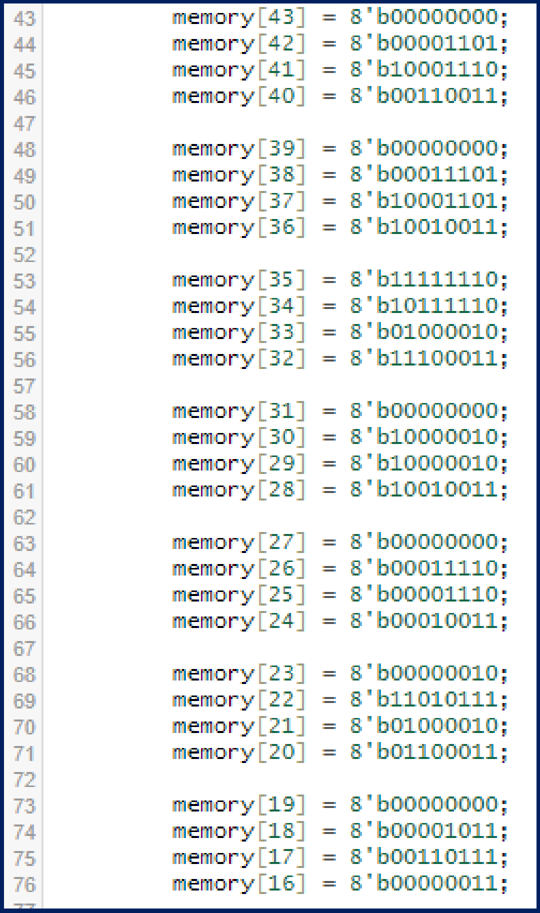
Instruction Memory



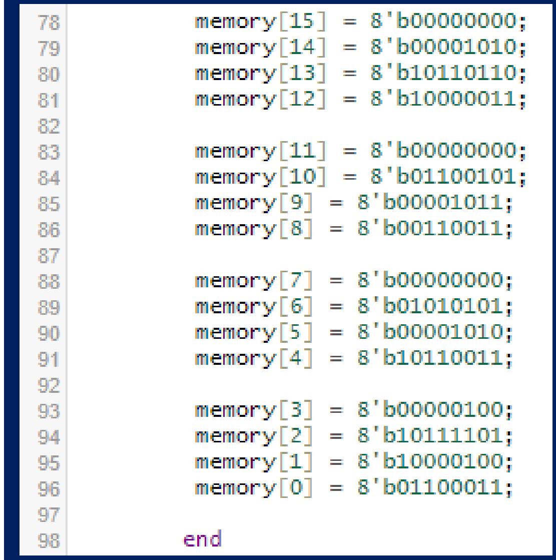
```verilog
module Instruction_Memory (
  input [63:0] Inst_Address,
  output reg [31:0] Instruction
);
```



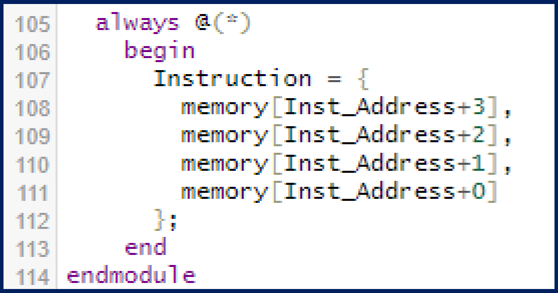
```verilog
reg [7:0] memory [71:0];
        initial begin
         memory[71] = 8'b11111100;
         memory[70] = 8'b00000000;
         memory[69] = 8'b00001010;
         memory[68] = 8'b11100011;

         memory[67] = 8'b00000001;
         memory[66] = 8'b11101010;
         memory[65] = 8'b10110000;
         memory[64] = 8'b00100011;

         memory[63] = 8'b00000000;
         memory[62] = 8'b11011011;
         memory[61] = 8'b00110000;
         memory[60] = 8'b00100011;

         memory[59] = 8'b00000000;
         memory[58] = 8'b11100000;
         memory[57] = 8'b00001111;
         memory[56] = 8'b00110011;

         memory[55] = 8'b11111100;
         memory[54] = 8'b00000000;
         memory[53] = 8'b00000110;
         memory[52] = 8'b11100011;

         memory[51] = 8'b00000000;
         memory[50] = 8'b01100000;
         memory[49] = 8'b00000010;
         memory[48] = 8'b10110011;

         memory[47] = 8'b00000000;
         memory[46] = 8'b10000011;
         memory[45] = 8'b00000011;
         memory[44] = 8'b00010011;
```



```verilog
         memory[43] = 8'b00000000;
         memory[42] = 8'b00001101;
         memory[41] = 8'b10001110;
         memory[40] = 8'b00110011;

         memory[39] = 8'b00000000;
         memory[38] = 8'b00011101;
         memory[37] = 8'b10001101;
         memory[36] = 8'b10010011;

         memory[35] = 8'b11111110;
         memory[34] = 8'b10111110;
         memory[33] = 8'b01000010;
         memory[32] = 8'b11100011;

         memory[31] = 8'b00000000;
         memory[30] = 8'b10000010;
         memory[29] = 8'b10000010;
         memory[28] = 8'b10010011;

         memory[27] = 8'b00000000;
         memory[26] = 8'b00011110;
         memory[25] = 8'b00001110;
         memory[24] = 8'b00010011;

         memory[23] = 8'b00000010;
         memory[22] = 8'b11010111;
         memory[21] = 8'b01000010;
         memory[20] = 8'b01100011;

         memory[19] = 8'b00000000;
         memory[18] = 8'b00001011;
         memory[17] = 8'b00110111;
         memory[16] = 8'b00000011;
```



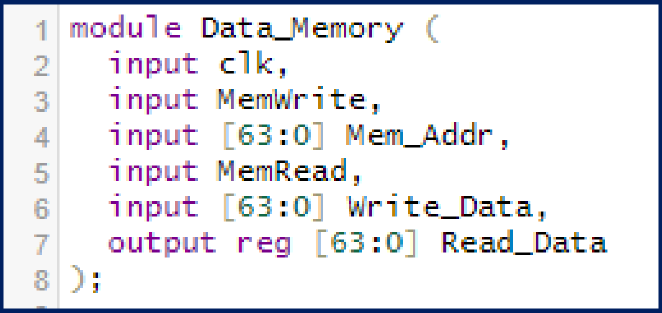
```verilog
         memory[15] = 8'b00000000;
         memory[14] = 8'b00001010;
         memory[13] = 8'b10110110;
         memory[12] = 8'b10000011;

         memory[11] = 8'b00000000;
         memory[10] = 8'b01100101;
         memory[9] = 8'b00001011;
         memory[8] = 8'b00110011;

         memory[7] = 8'b00000000;
         memory[6] = 8'b01010101;
         memory[5] = 8'b00001010;
         memory[4] = 8'b10110011;

         memory[3] = 8'b00000100;
         memory[2] = 8'b10111101;
         memory[1] = 8'b10000100;
         memory[0] = 8'b01100011;

        end
```



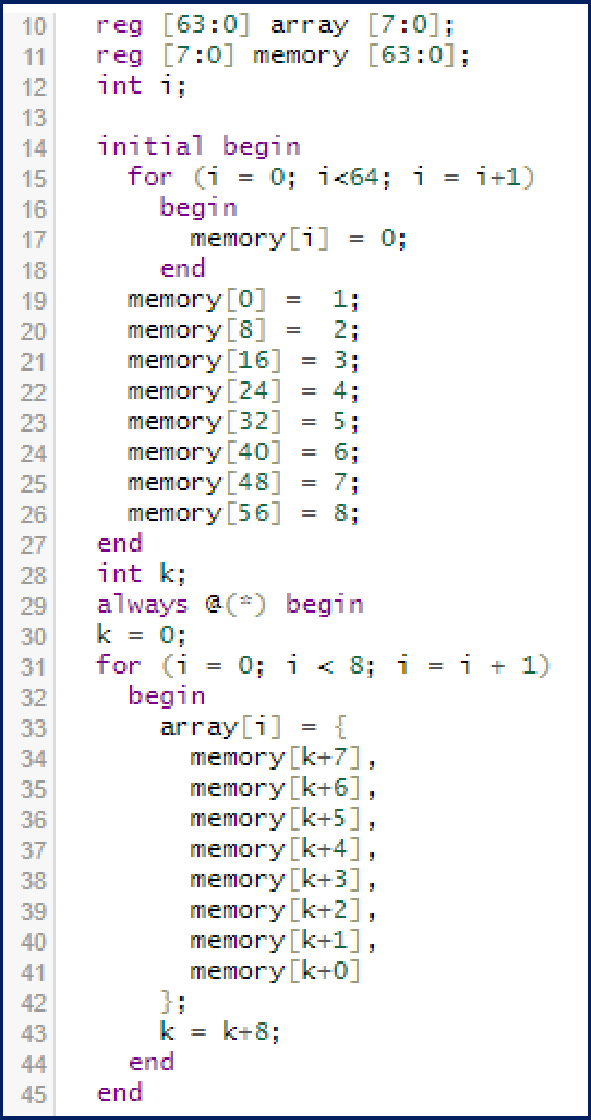
```verilog
  always @(*)
    begin
      Instruction = {
        memory[Inst_Address+3],
        memory[Inst_Address+2],
        memory[Inst_Address+1],
        memory[Inst_Address+0]
      };
    end
endmodule
```

Data Memory

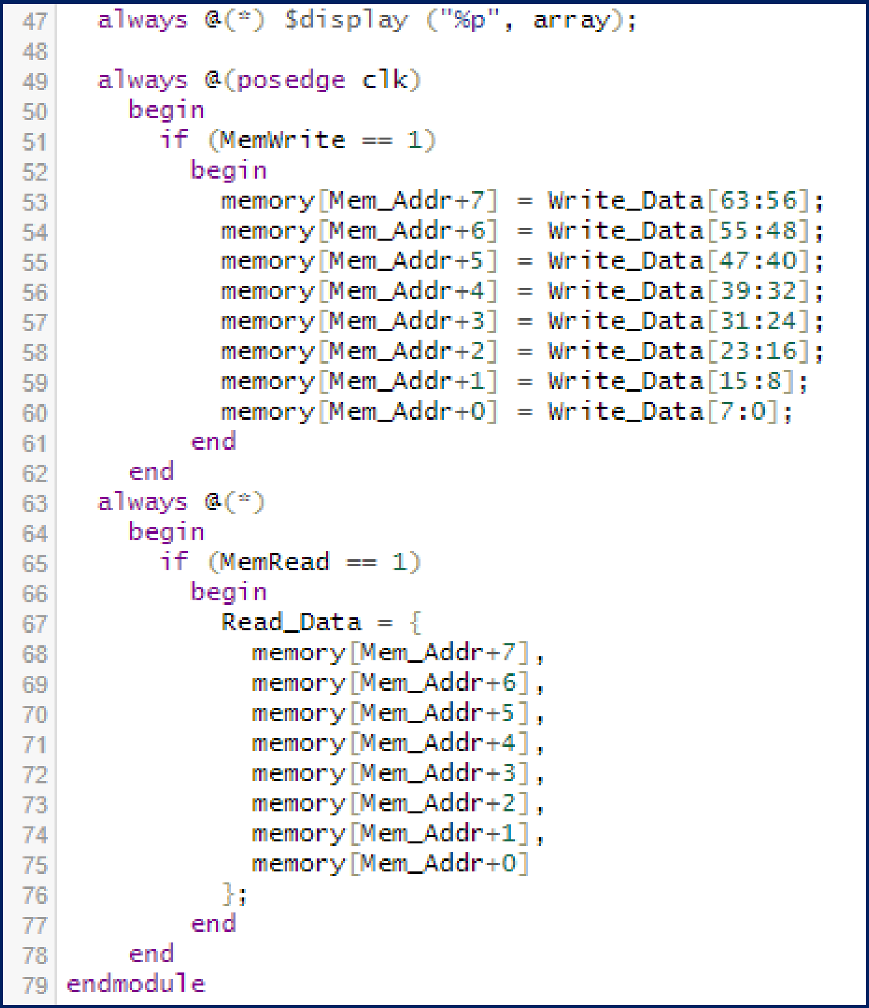
Storing elements in the memory (lines 19-26) and storing elements from memory to an array and displaying the array in the log window (line 47).



```
 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 );
```

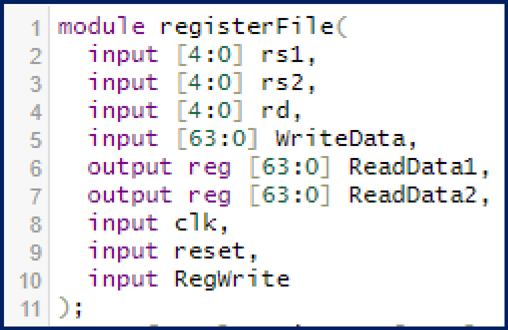


```
10   reg [63:0] array [7:0];
11   reg [7:0] memory [63:0];
12   int i;
13
14   initial begin
15     for (i = 0; i<64; i = i+1)
16       begin
17         memory[i] = 0;
18       end
19     memory[0]  =  1;
20     memory[8]  =  2;
21     memory[16] = 3;
22     memory[24] = 4;
23     memory[32] = 5;
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
```

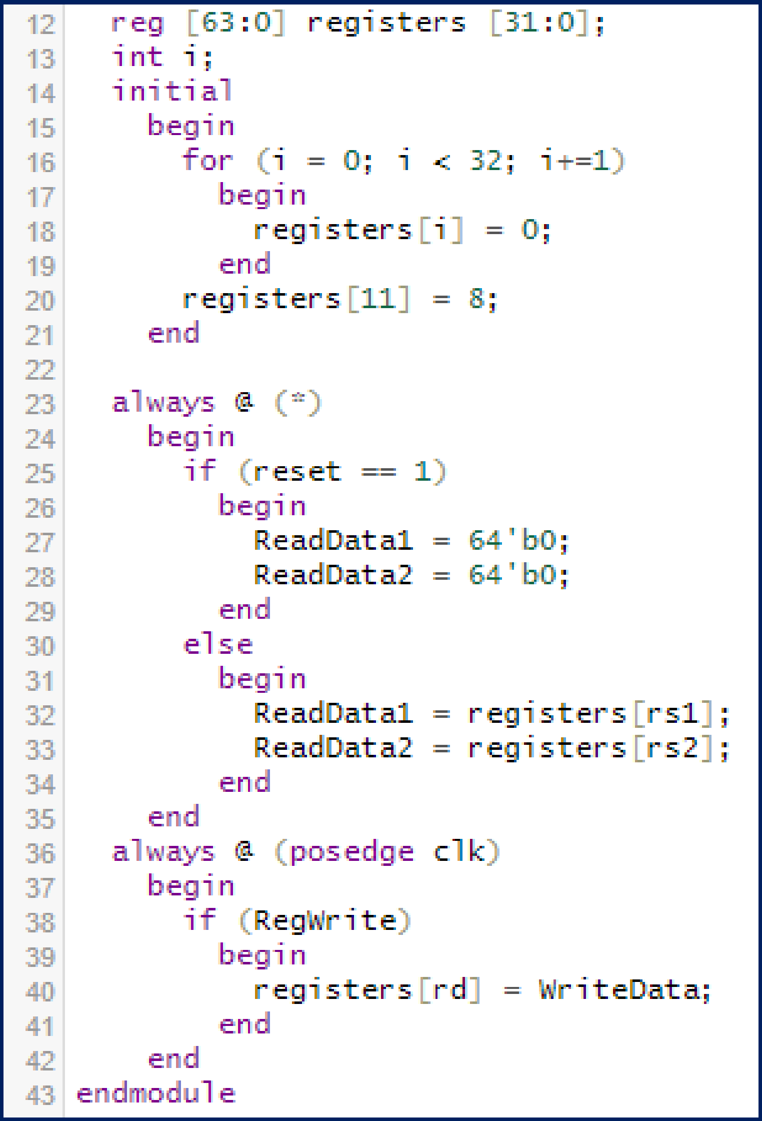


```
47   always @(*) $display ("%p", array);
48
49   always @(posedge clk)
50     begin
51       if (MemWrite == 1)
52         begin
53           memory[Mem_Addr+7]  = Write_Data[63:56];
54           memory[Mem_Addr+6]  = Write_Data[55:48];
55           memory[Mem_Addr+5]  = Write_Data[47:40];
56           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)

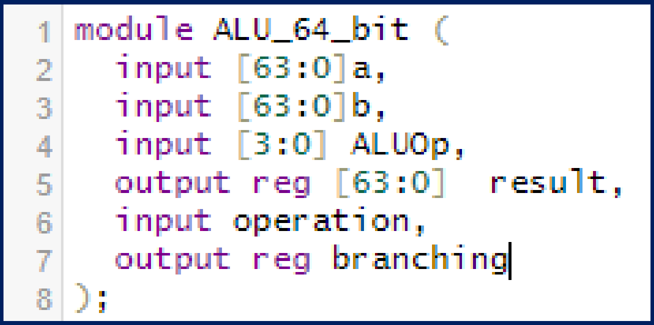


```
module registerFile(
  input [4:0] rs1,
  input [4:0] rs2,
  input [4:0] rd,
  input [63:0] WriteData,
  output reg [63:0] ReadData1,
  output reg [63:0] ReadData2,
  input clk,
  input reset,
  input RegWrite
);
```

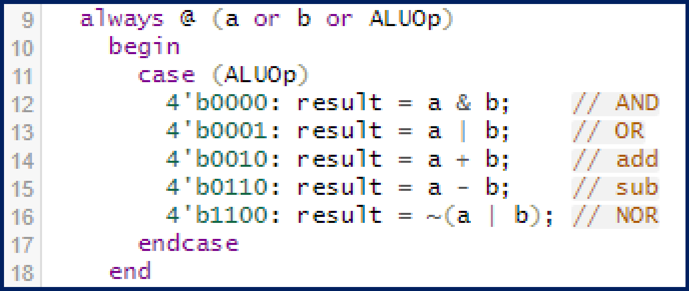


```
  reg [63:0] registers [31:0];
  int i;
  initial
    begin
      for (i = 0; i < 32; i+=1)
        begin
          registers[i] = 0;
        end
      registers[11] = 8;
    end

  always @ (*)
    begin
      if (reset == 1)
        begin
          ReadData1 = 64'b0;
          ReadData2 = 64'b0;
        end
      else
        begin
          ReadData1 = registers[rs1];
          ReadData2 = registers[rs2];
        end
    end
  always @ (posedge clk)
    begin
      if (RegWrite)
        begin
          registers[rd] = WriteData;
        end
    end
endmodule
```

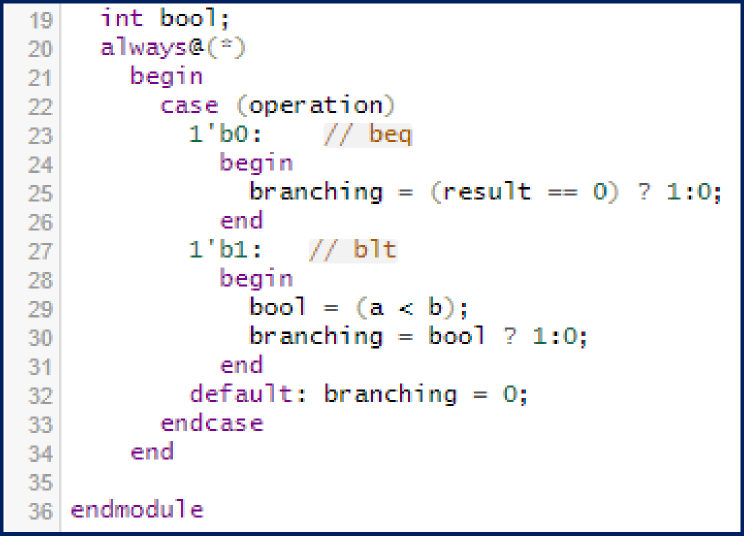
ALU 64 bit



```
module ALU_64_bit (
  input [63:0]a,
  input [63:0]b,
  input [3:0] ALUOp,
  output reg [63:0]  result,
  input operation,
  output reg branching
);
```



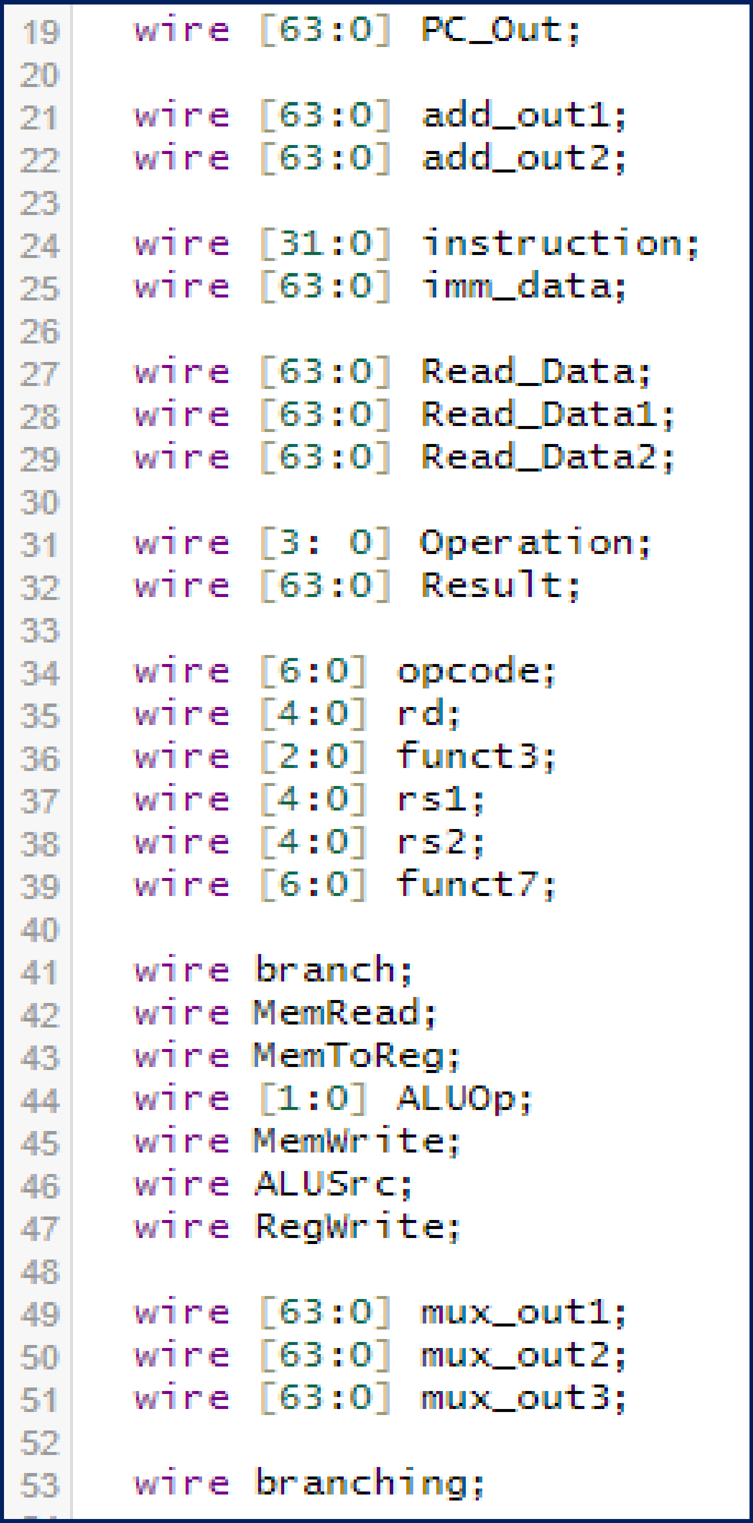
```
  always @ (a or b or ALUOp)
    begin
      case (ALUOp)
        4'b0000: result = a & b;     // AND
        4'b0001: result = a | b;     // OR
        4'b0010: result = a + b;     // add
        4'b0110: result = a - b;     // sub
        4'b1100: result = ~(a | b);  // NOR
      endcase
    end
```



```
  int bool;
  always@(*)
    begin
      case (operation)
        1'b0:     // beq
          begin
            branching = (result == 0) ? 1:0;
          end
        1'b1:    // blt
          begin
            bool = (a < b);
            branching = bool ? 1:0;
          end
        default: branching = 0;
      endcase
    end

endmodule
```

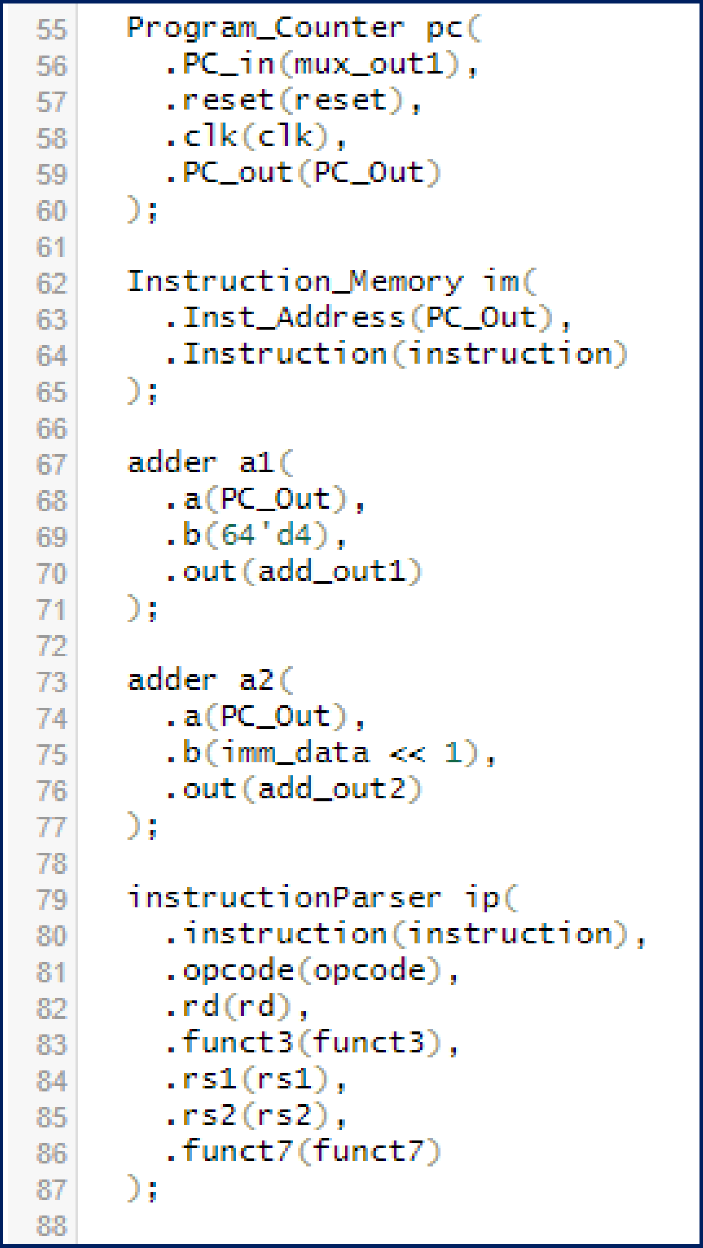
This was the most difficult (yet the easiest thing) we found in task 1. We essentially make the control signal named as branching either low or high depending on the instruction. We determine the instruction type by the seeing the input signal named as operation (which is essentially the most significant 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

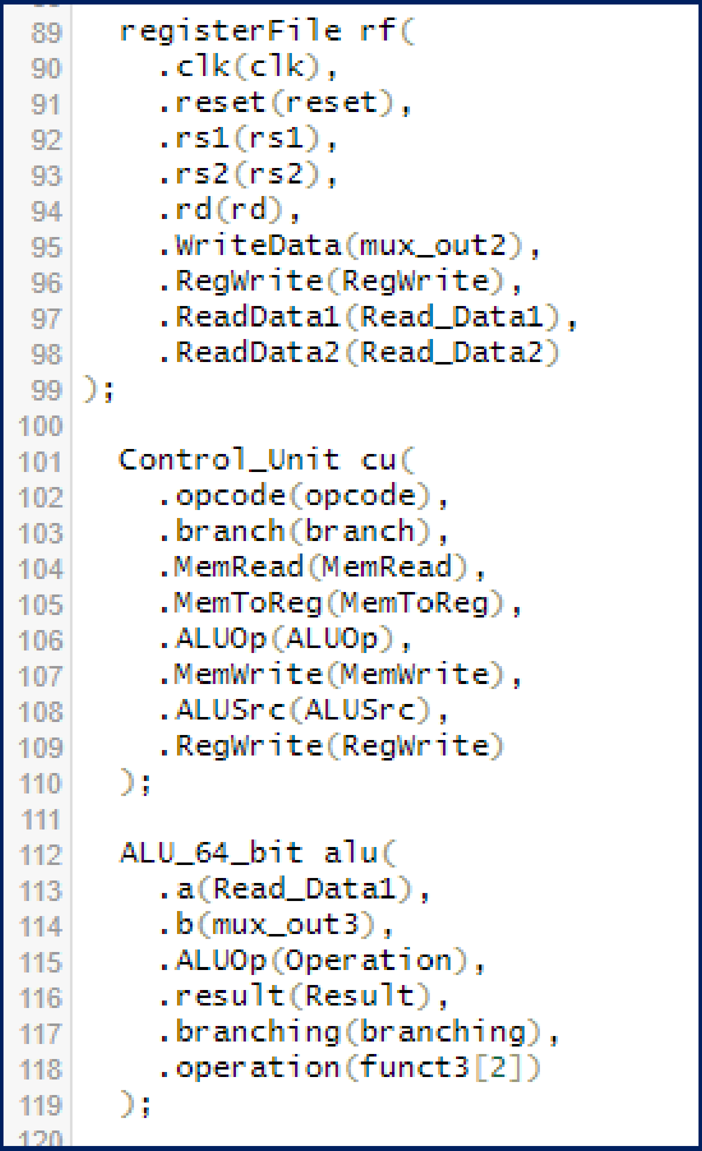


```
    wire [63:0] PC_Out;

    wire [63:0] add_out1;
    wire [63:0] add_out2;

    wire [31:0] instruction;
    wire [63:0] imm_data;

    wire [63:0] Read_Data;
    wire [63:0] Read_Data1;
    wire [63:0] Read_Data2;

    wire [3: 0] Operation;
    wire [63:0] Result;

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

    wire branch;
    wire MemRead;
    wire MemToReg;
    wire [1:0] ALUOp;
    wire MemWrite;
    wire ALUSrc;
    wire RegWrite;

    wire [63:0] mux_out1;
    wire [63:0] mux_out2;
    wire [63:0] mux_out3;

    wire branching;
```

Making the connections

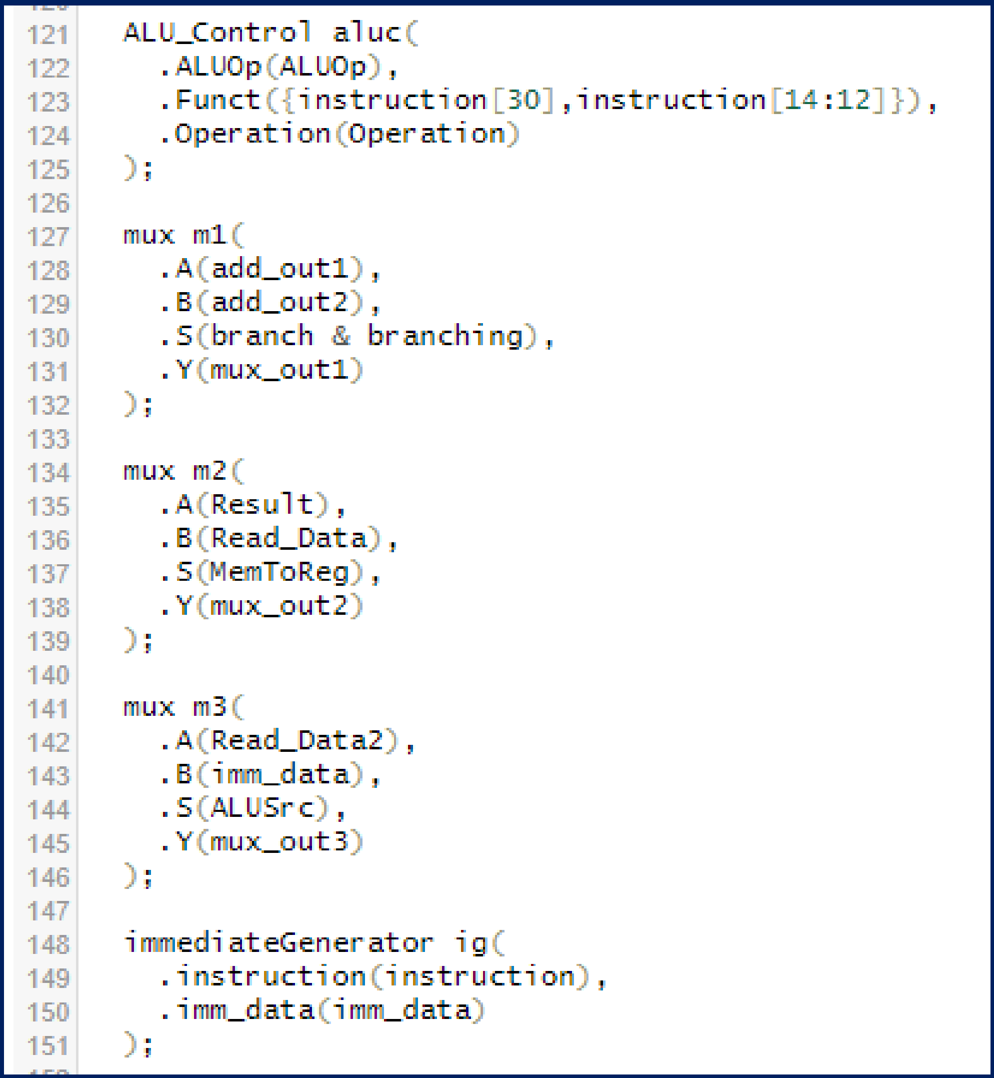


```
Program_Counter pc(
  .PC_in(mux_out1),
  .reset(reset),
  .clk(clk),
  .PC_out(PC_Out)
);

Instruction_Memory im(
  .Inst_Address(PC_Out),
  .Instruction(instruction)
);

adder a1(
  .a(PC_Out),
  .b(64'd4),
  .out(add_out1)
);

adder a2(
  .a(PC_Out),
  .b(imm_data << 1),
  .out(add_out2)
);

instructionParser ip(
  .instruction(instruction),
  .opcode(opcode),
  .rd(rd),
  .funct3(funct3),
  .rs1(rs1),
  .rs2(rs2),
  .funct7(funct7)
);

```

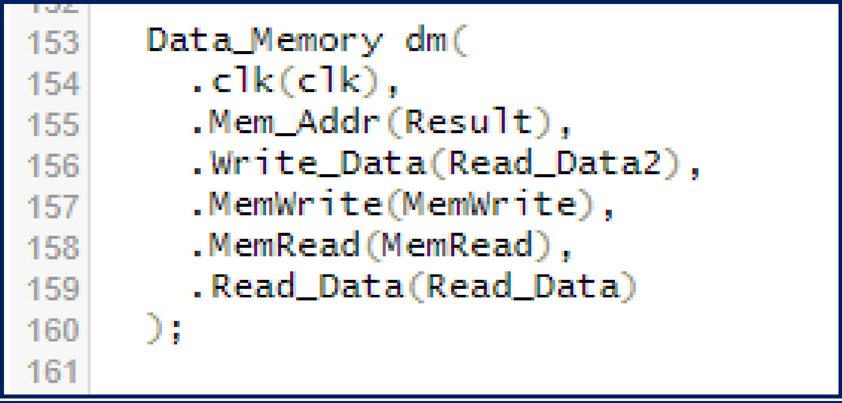


```
registerFile rf(
  .clk(clk),
  .reset(reset),
  .rs1(rs1),
  .rs2(rs2),
  .rd(rd),
  .WriteData(mux_out2),
  .RegWrite(RegWrite),
  .ReadData1(Read_Data1),
  .ReadData2(Read_Data2)
);

Control_Unit cu(
  .opcode(opcode),
  .branch(branch),
  .MemRead(MemRead),
  .MemToReg(MemToReg),
  .ALUOp(ALUOp),
  .MemWrite(MemWrite),
  .ALUSrc(ALUSrc),
  .RegWrite(RegWrite)
);

ALU_64_bit alu(
  .a(Read_Data1),
  .b(mux_out3),
  .ALUOp(Operation),
  .result(Result),
  .branching(branching),
  .operation(funct3[2])
);
```

Making some more connections

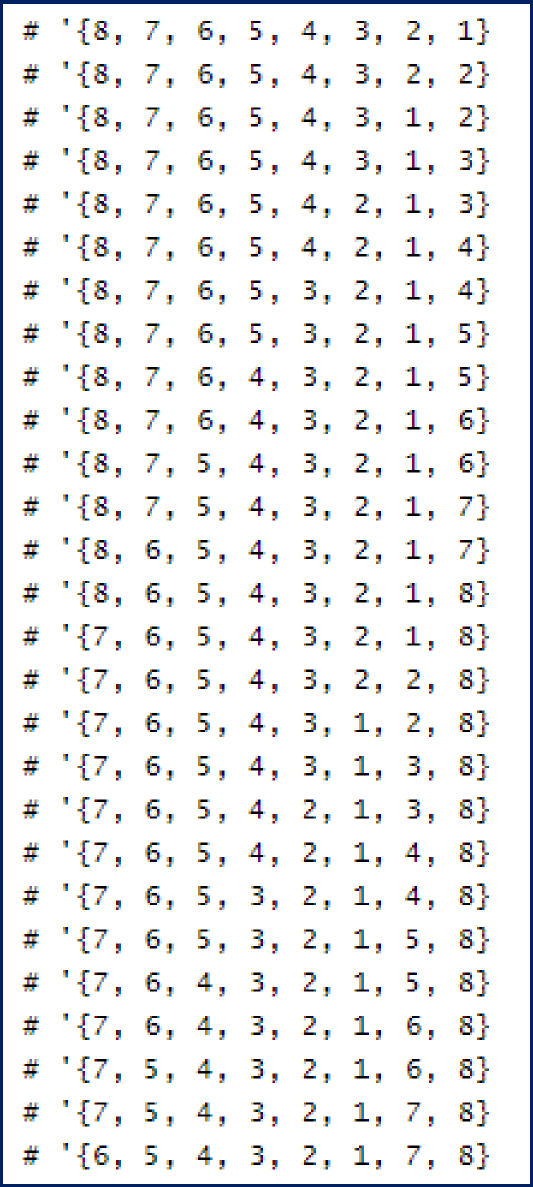


```
ALU_Control aluc(
  .ALUOp(ALUOp),
  .Funct({instruction[30],instruction[14:12]}),
  .Operation(Operation)
);

mux m1(
  .A(add_out1),
  .B(add_out2),
  .S(branch & branching),
  .Y(mux_out1)
);

mux m2(
  .A(Result),
  .B(Read_Data),
  .S(MemToReg),
  .Y(mux_out2)
);

mux m3(
  .A(Read_Data2),
  .B(imm_data),
  .S(ALUSrc),
  .Y(mux_out3)
);

immediateGenerator ig(
  .instruction(instruction),
  .imm_data(imm_data)
);
```

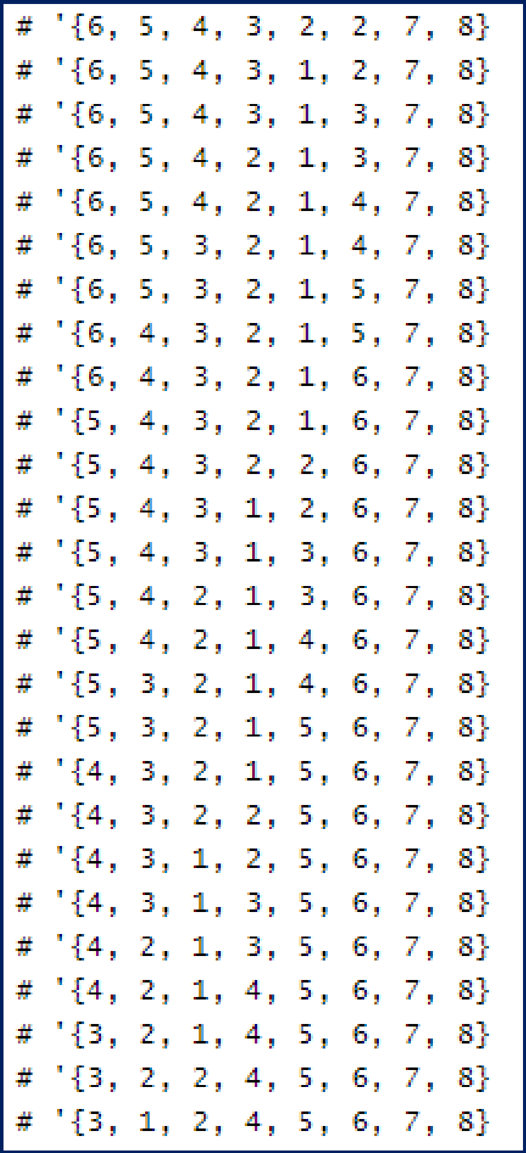


```
Data_Memory dm(
  .clk(clk),
  .Mem_Addr(Result),
  .Write_Data(Read_Data2),
  .MemWrite(MemWrite),
  .MemRead(MemRead),
  .Read_Data(Read_Data)
);
```

Simulation (LOG)

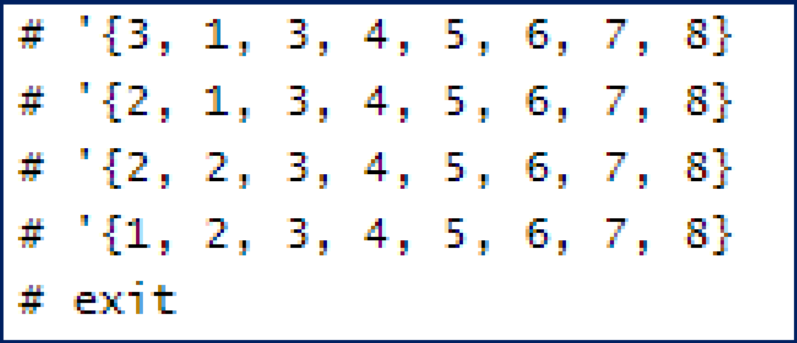


```
# '{8, 7, 6, 5, 4, 3, 2, 1}
# '{8, 7, 6, 5, 4, 3, 2, 2}
# '{8, 7, 6, 5, 4, 3, 1, 2}
# '{8, 7, 6, 5, 4, 3, 1, 3}
# '{8, 7, 6, 5, 4, 2, 1, 3}
# '{8, 7, 6, 5, 4, 2, 1, 4}
# '{8, 7, 6, 5, 3, 2, 1, 4}
# '{8, 7, 6, 5, 3, 2, 1, 5}
# '{8, 7, 6, 4, 3, 2, 1, 5}
# '{8, 7, 6, 4, 3, 2, 1, 6}
# '{8, 7, 5, 4, 3,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



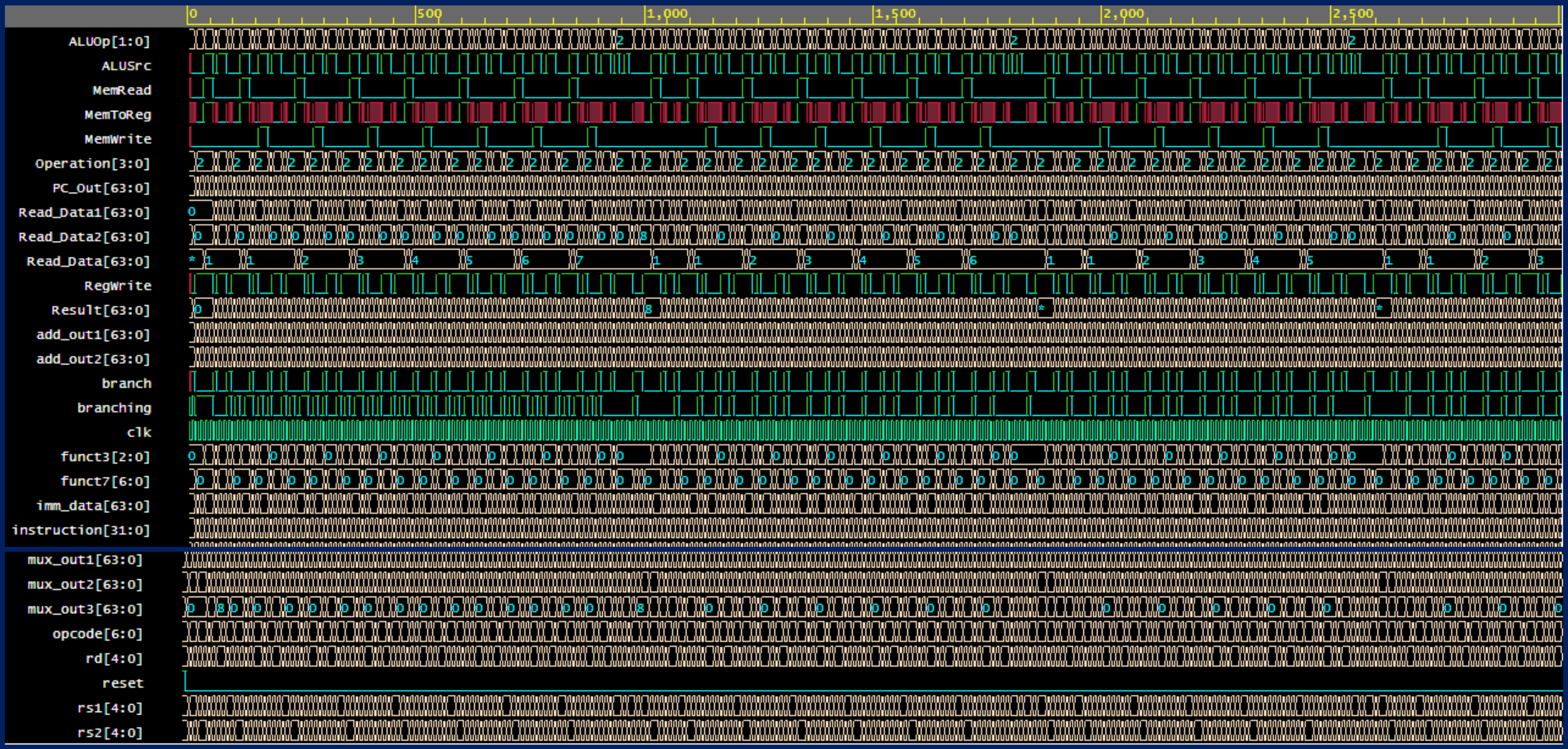
```
# '{6, 5, 4, 3, 2, 2, 7, 8}
# '{6, 5, 4, 3, 1, 2, 7, 8}
# '{6, 5, 4, 3, 1, 3, 7, 8}
# '{6, 5, 4, 2, 1, 3, 7, 8}
# '{6, 5, 4, 2, 1, 4, 7, 8}
# '{6, 5, 3, 2, 1, 4, 7, 8}
# '{6, 5, 3, 2, 1, 5, 7, 8}
# '{6, 4, 3, 2, 1, 5, 7, 8}
# '{6, 4, 3, 2, 1, 6, 7, 8}
# '{5, 4, 3, 2, 1, 6, 7, 8}
# '{5, 4, 3, 2, 2,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

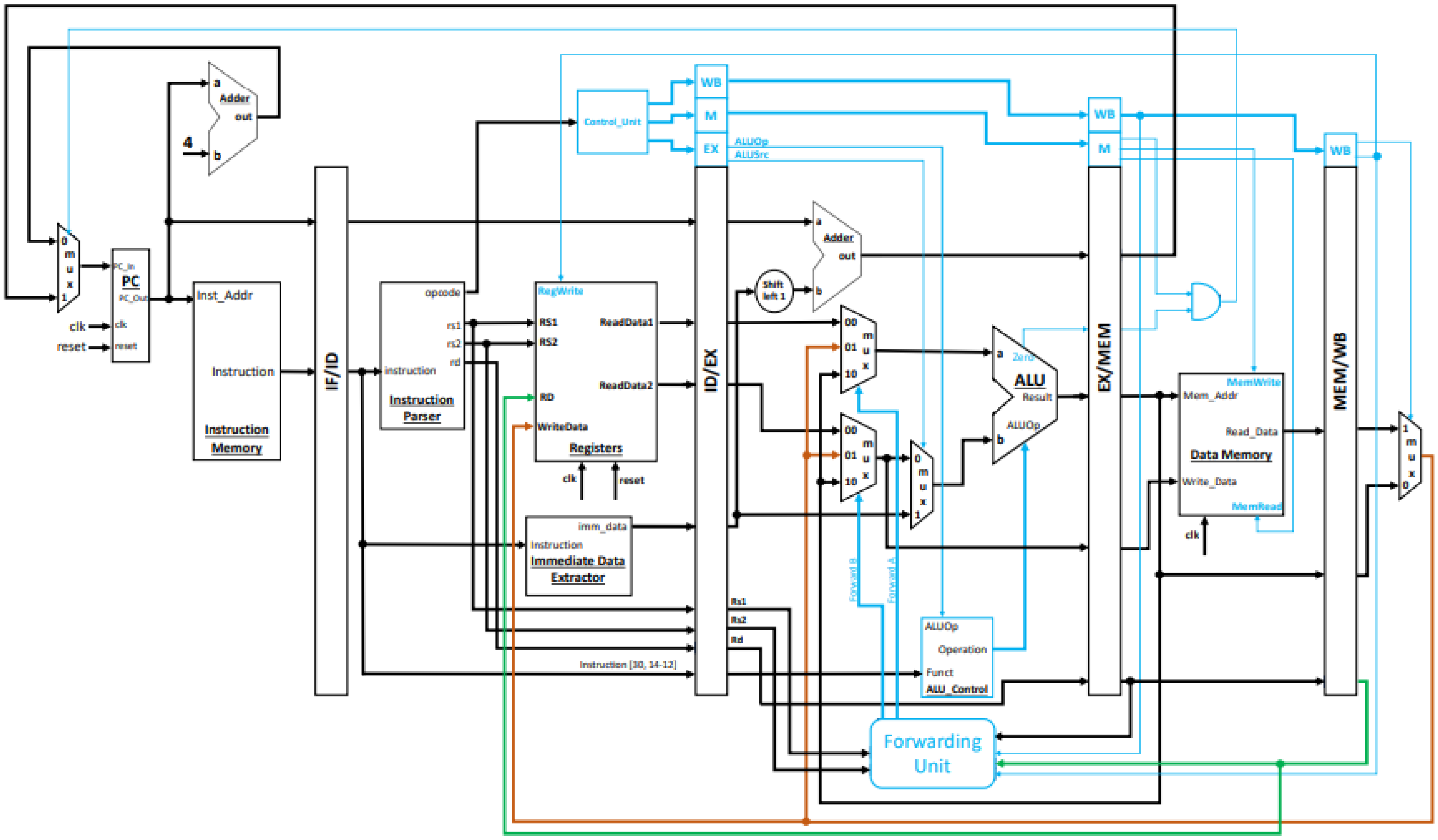
EMP wave



TASK 1 (Link: RISC_V Processor)

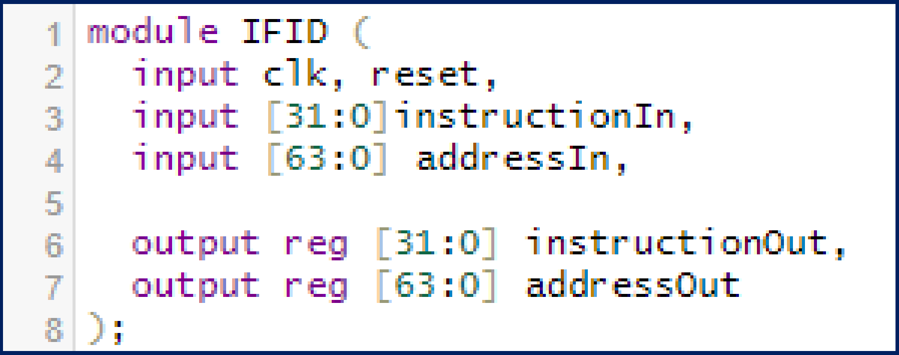
TASK (Link: Pipelined RISC Processor)

A pipelined processor based on the following diagram

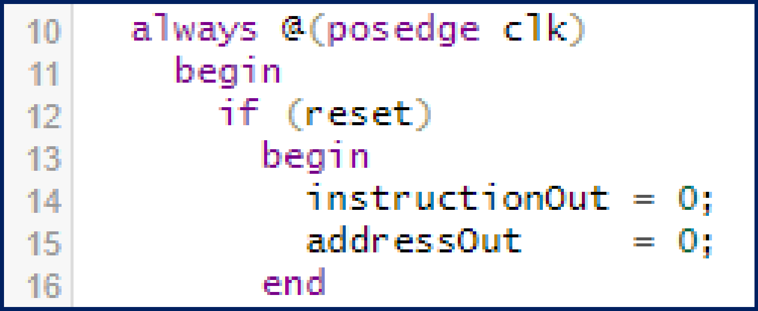


The following modules (IFID, IDEX, EXMEM, MEMWB) essentially just sets the output equal to the input at the positive edge of the clock.

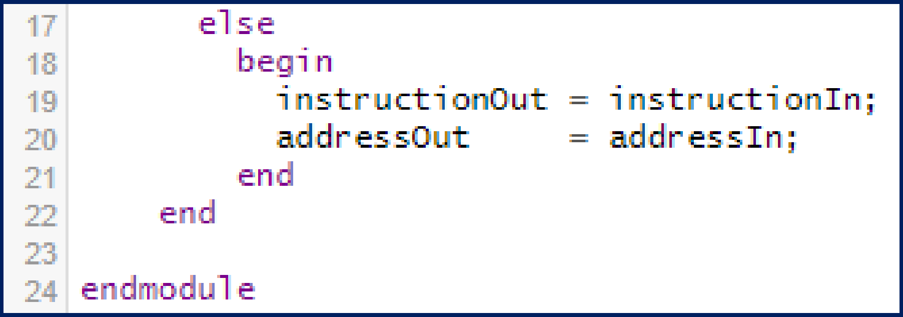
Instruction Fetch/Instruction Decode



```verilog
module IFID (
  input clk, reset,
  input [31:0]instructionIn,
  input [63:0] addressIn,

  output reg [31:0] instructionOut,
  output reg [63:0] addressOut
);
```

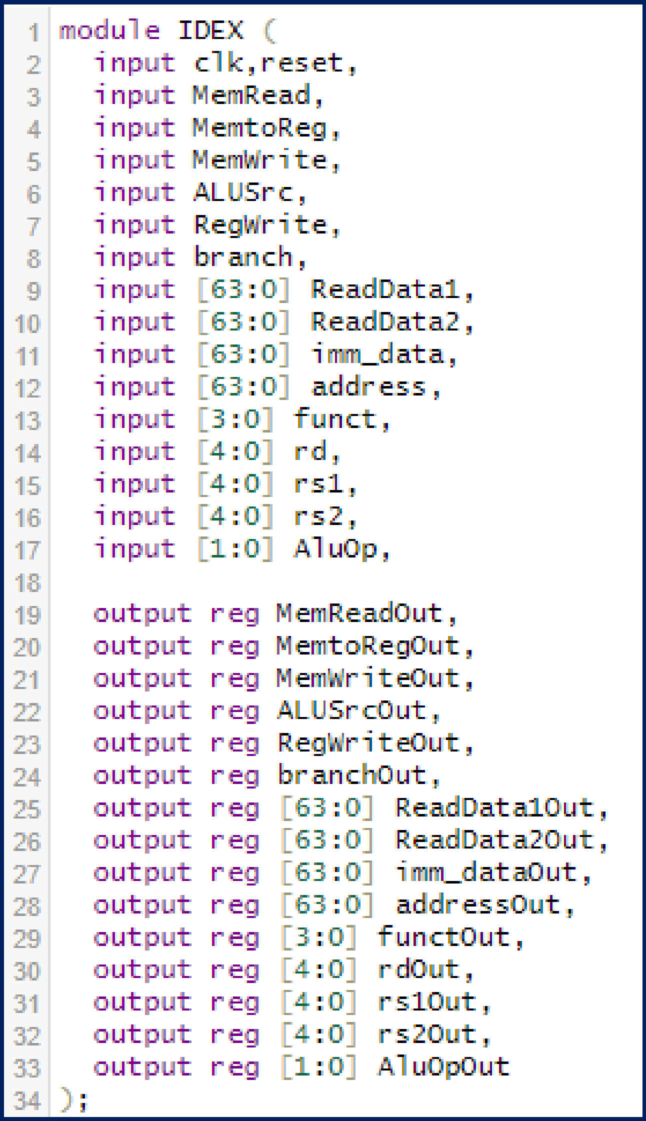


```verilog
  always @(posedge clk)
    begin
      if (reset)
        begin
          instructionOut = 0;
          addressOut     = 0;
        end
```

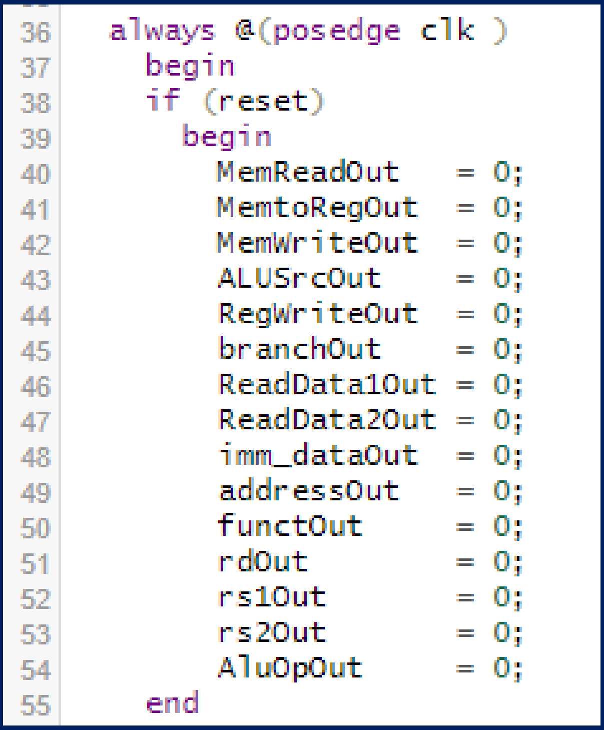


```verilog
      else
        begin
          instructionOut = instructionIn;
          addressOut     = addressIn;
        end
    end

endmodule
```

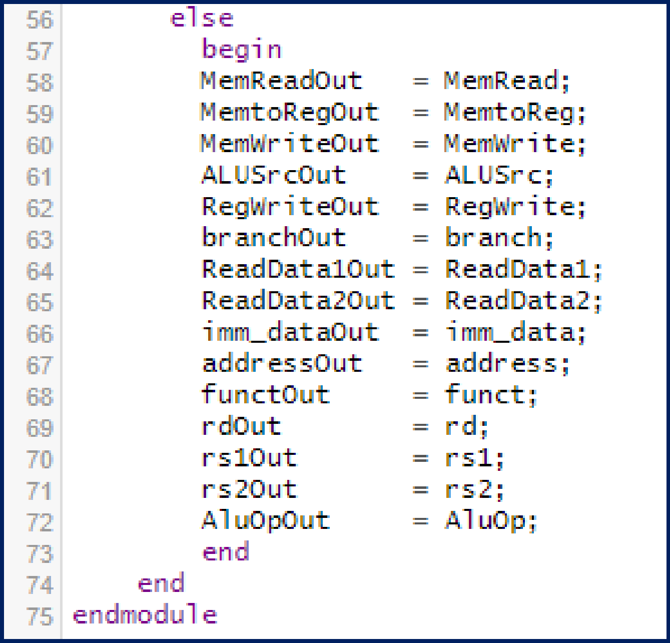
Instruction Decode/Execute



```verilog
module IDEX (
  input clk,reset,
  input MemRead,
  input MemtoReg,
  input MemWrite,
  input ALUSrc,
  input RegWrite,
  input branch,
  input [63:0] ReadData1,
  input [63:0] ReadData2,
  input [63:0] imm_data,
  input [63:0] address,
  input [3:0] funct,
  input [4:0] rd,
  input [4:0] rs1,
  input [4:0] rs2,
  input [1:0] AluOp,

  output reg MemReadOut,
  output reg MemtoRegOut,
  output reg MemWriteOut,
  output reg ALUSrcOut,
  output reg RegWriteOut,
  output reg branchOut,
  output reg [63:0] ReadData1Out,
  output reg [63:0] ReadData2Out,
  output reg [63:0] imm_dataOut,
  output reg [63:0] addressOut,
  output reg [3:0] functOut,
  output reg [4:0] rdOut,
  output reg [4:0] rs1Out,
  output reg [4:0] rs2Out,
  output reg [1:0] AluOpOut
);
```

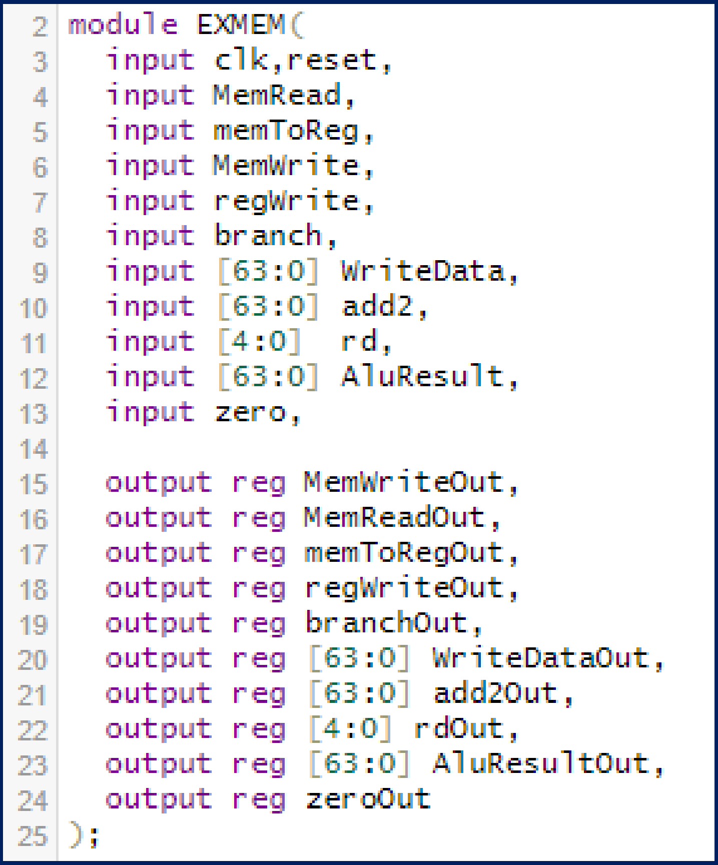


```verilog
  always @(posedge clk )
    begin
    if (reset)
      begin
        MemReadOut    = 0;
        MemtoRegOut   = 0;
        MemWriteOut   = 0;
        ALUSrcOut     = 0;
        RegWriteOut   = 0;
        branchOut     = 0;
        ReadData1Out  = 0;
        ReadData2Out  = 0;
        imm_dataOut   = 0;
        addressOut    = 0;
        functOut      = 0;
        rdOut         = 0;
        rs1Out        = 0;
        rs2Out        = 0;
        AluOpOut      = 0;
    end
```

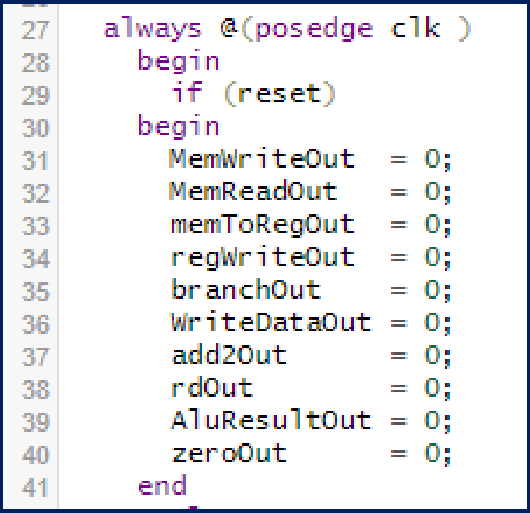


```verilog
      else
        begin
        MemReadOut    = MemRead;
        MemtoRegOut   = MemtoReg;
        MemWriteOut   = MemWrite;
        ALUSrcOut     = ALUSrc;
        RegWriteOut   = RegWrite;
        branchOut     = branch;
        ReadData1Out  = ReadData1;
        ReadData2Out  = ReadData2;
        imm_dataOut   = imm_data;
        addressOut    = address;
        functOut      = funct;
        rdOut         = rd;
        rs1Out        = rs1;
        rs2Out        = rs2;
        AluOpOut      = AluOp;
        end
    end
endmodule
```

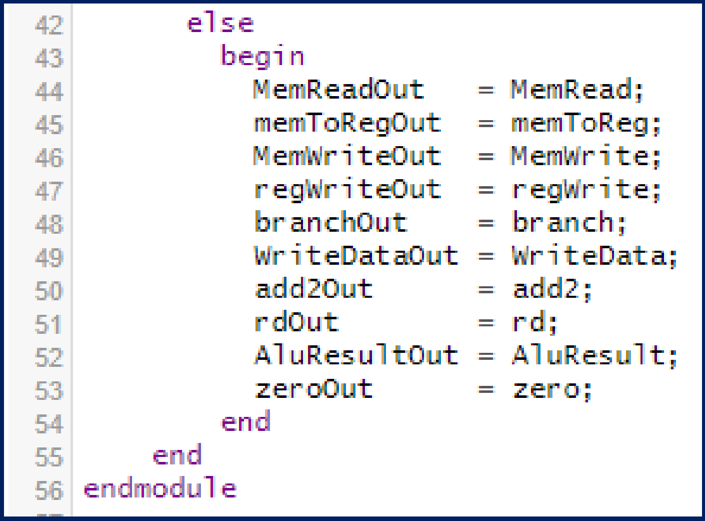
Execute/Memory Access



```verilog
module EXMEM(
  input clk,reset,
  input MemRead,
  input memToReg,
  input MemWrite,
  input regWrite,
  input branch,
  input [63:0] WriteData,
  input [63:0] add2,
  input [4:0]  rd,
  input [63:0] AluResult,
  input zero,

  output reg MemWriteOut,
  output reg MemReadOut,
  output reg memToRegOut,
  output reg regWriteOut,
  output reg branchOut,
  output reg [63:0] WriteDataOut,
  output reg [63:0] add2Out,
  output reg [4:0] rdOut,
  output reg [63:0] AluResultOut,
  output reg zeroOut
);
```

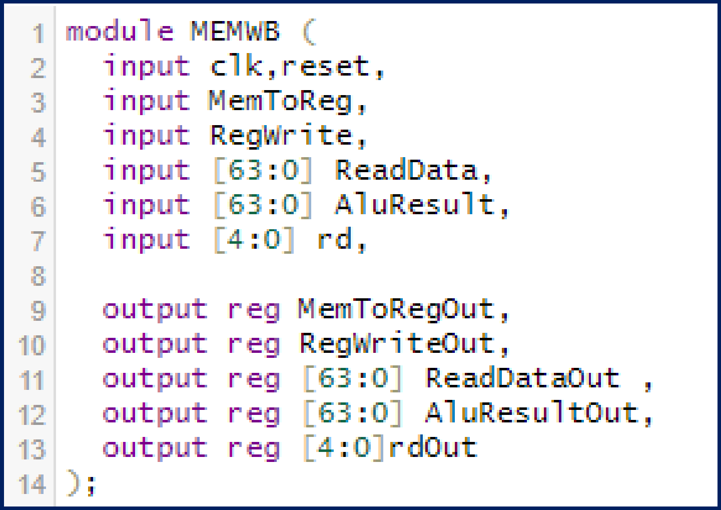


```verilog
  always @(posedge clk )
    begin
      if (reset)
    begin
      MemWriteOut   = 0;
      MemReadOut    = 0;
      memToRegOut   = 0;
      regWriteOut   = 0;
      branchOut     = 0;
      WriteDataOut  = 0;
      add2Out       = 0;
      rdOut         = 0;
      AluResultOut  = 0;
      zeroOut       = 0;
    end
```

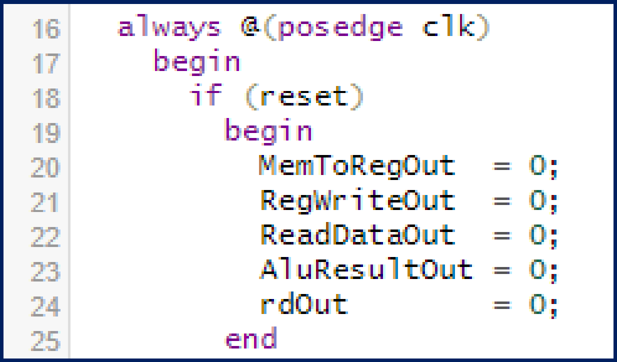


```verilog
      else
        begin
          MemReadOut    = MemRead;
          memToRegOut   = memToReg;
          MemWriteOut   = MemWrite;
          regWriteOut   = regWrite;
          branchOut     = branch;
          WriteDataOut  = WriteData;
          add2Out       = add2;
          rdOut         = rd;
          AluResultOut  = AluResult;
          zeroOut       = zero;
        end
    end
endmodule
```

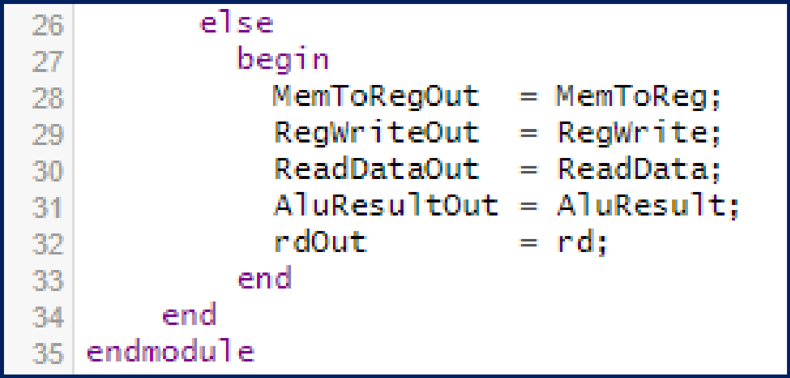
Memory Access/Write back



```
module MEMWB (
  input clk,reset,
  input MemToReg,
  input RegWrite,
  input [63:0] ReadData,
  input [63:0] AluResult,
  input [4:0] rd,

  output reg MemToRegOut,
  output reg RegWriteOut,
  output reg [63:0] ReadDataOut ,
  output reg [63:0] AluResultOut,
  output reg [4:0]rdOut
);
```

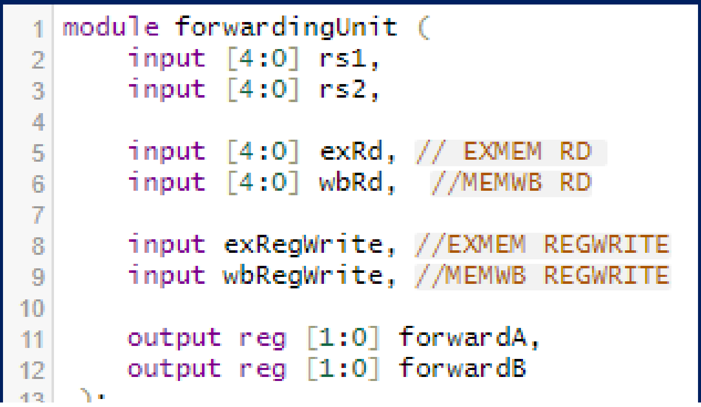


```
  always @(posedge clk)
    begin
      if (reset)
        begin
          MemToRegOut  = 0;
          RegWriteOut  = 0;
          ReadDataOut  = 0;
          AluResultOut = 0;
          rdOut        = 0;
        end
```

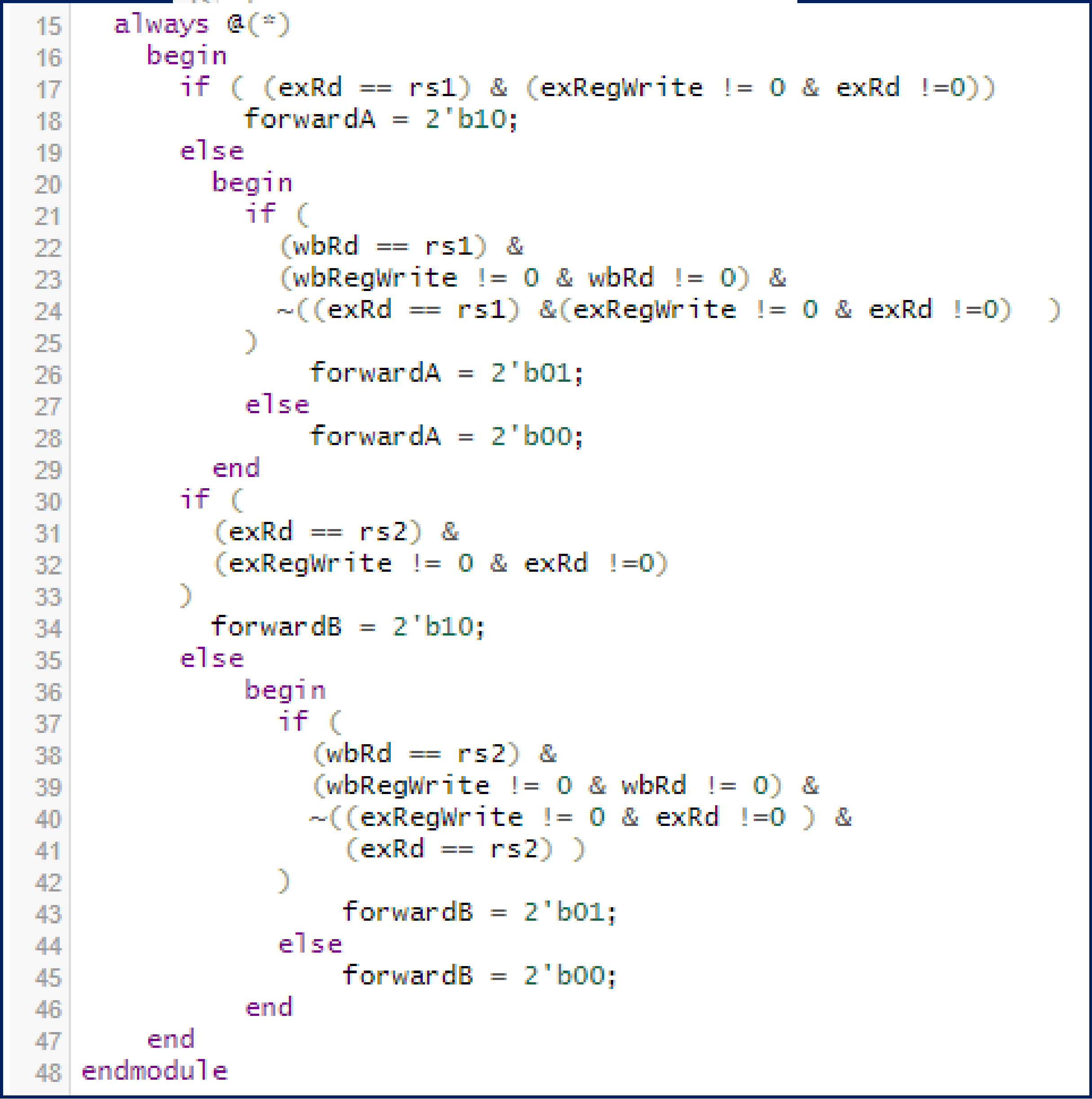


```
      else
        begin
          MemToRegOut  = MemToReg;
          RegWriteOut  = RegWrite;
          ReadDataOut  = ReadData;
          AluResultOut = AluResult;
          rdOut        = rd;
        end
    end
endmodule
```

Forwarding Unit (pseudocode referenced from the book)

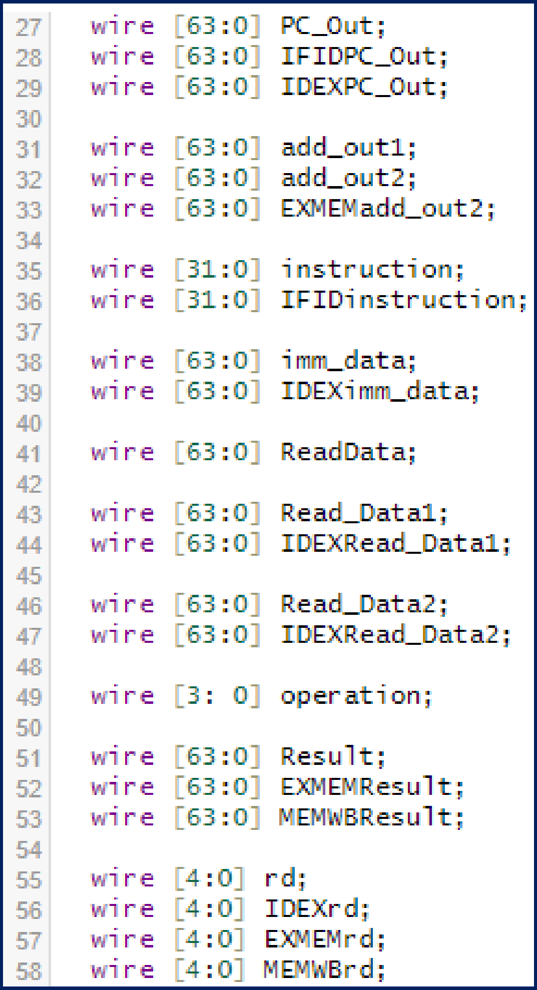


```
module forwardingUnit (
    input [4:0] rs1,
    input [4:0] rs2,

    input [4:0] exRd, // EXMEM RD
    input [4:0] wbRd,  //MEMWB RD

    input exRegWrite, //EXMEM REGWRITE
    input wbRegWrite, //MEMWB REGWRITE

    output reg [1:0] forwardA,
    output reg [1:0] forwardB
);
```

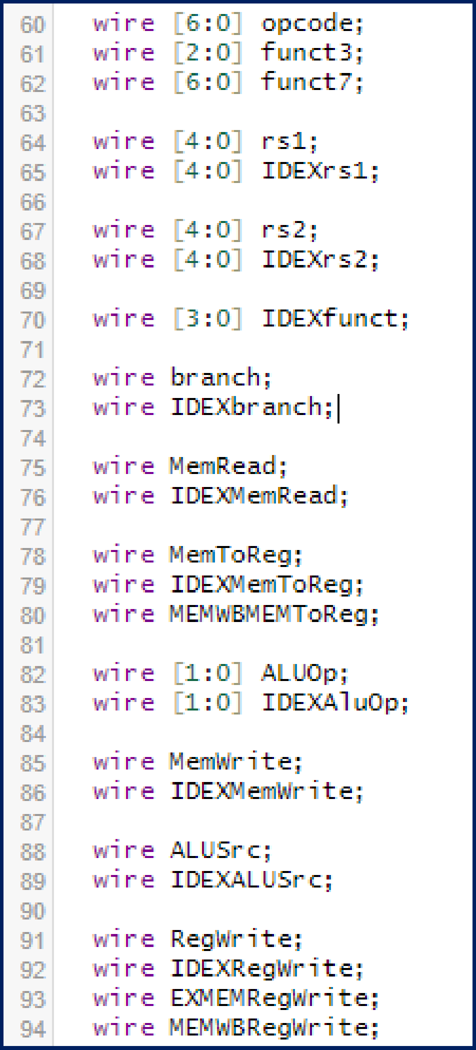


```
 always @(*)
  begin
   if ( (exRd == rs1) & (exRegWrite != 0 & exRd !=0))
     forwardA = 2'b10;
   else
    begin
     if (
      (wbRd == rs1) &
      (wbRegWrite != 0 & wbRd != 0) &
      ~((exRd == rs1) &(exRegWrite != 0 & exRd !=0)   )
     )
       forwardA = 2'b01;
     else
       forwardA = 2'b00;
    end
   if (
    (exRd == rs2) &
    (exRegWrite != 0 & exRd !=0)
   )
    forwardB = 2'b10;
   else
     begin
      if (
       (wbRd == rs2) &
       (wbRegWrite != 0 & wbRd != 0) &
       ~((exRegWrite != 0 & exRd !=0 ) &
        (exRd == rs2) )
      )
        forwardB = 2'b01;
      else
        forwardB = 2'b00;
     end
  end
endmodule
```

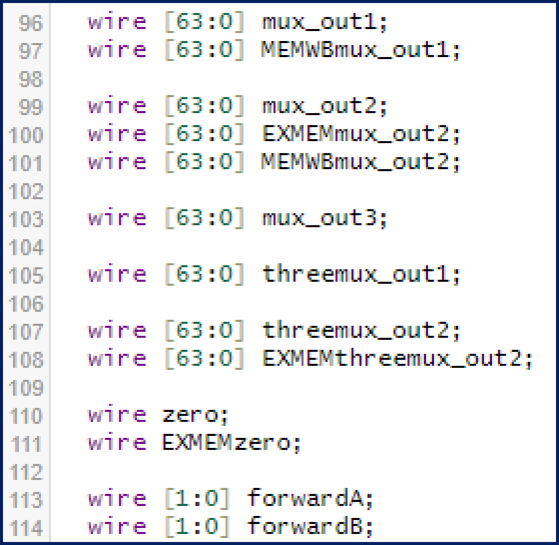
Declaring the wires (Now we will have some additional wires for each stage)



```
wire [63:0] PC_Out;
wire [63:0] IFIDPC_Out;
wire [63:0] IDEXPC_Out;

wire [63:0] add_out1;
wire [63:0] add_out2;
wire [63:0] EXMEMadd_out2;

wire [31:0] instruction;
wire [31:0] IFIDinstruction;

wire [63:0] imm_data;
wire [63:0] IDEXimm_data;

wire [63:0] ReadData;

wire [63:0] Read_Data1;
wire [63:0] IDEXRead_Data1;

wire [63:0] Read_Data2;
wire [63:0] IDEXRead_Data2;

wire [3: 0] operation;

wire [63:0] Result;
wire [63:0] EXMEMResult;
wire [63:0] MEMWBResult;

wire [4:0] rd;
wire [4:0] IDEXrd;
wire [4:0] EXMEMrd;
wire [4:0] MEMWBrd;
```

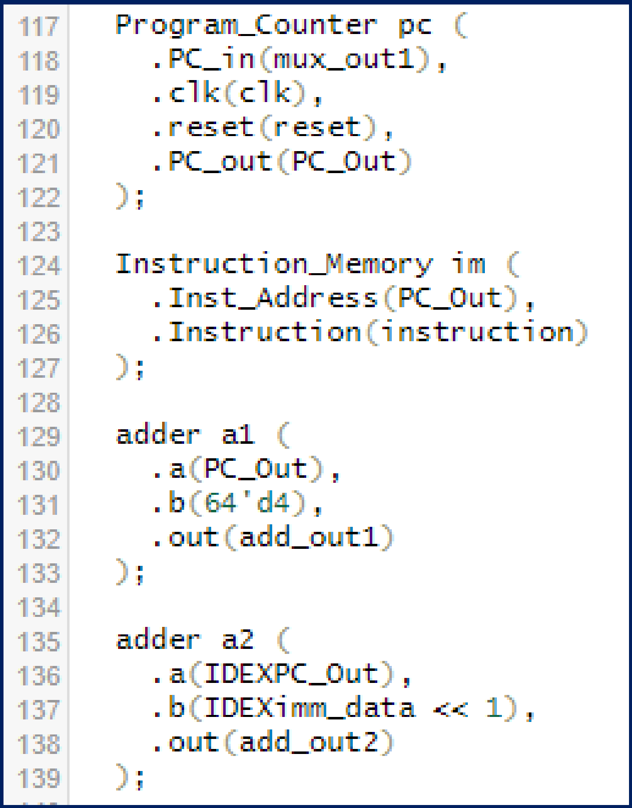


```
wire [6:0] opcode;
wire [2:0] funct3;
wire [6:0] funct7;

wire [4:0] rs1;
wire [4:0] IDEXrs1;

wire [4:0] rs2;
wire [4:0] IDEXrs2;

wire [3:0] IDEXfunct;

wire branch;
wire IDEXbranch;

wire MemRead;
wire IDEXMemRead;

wire MemToReg;
wire IDEXMemToReg;
wire MEMWBMEMToReg;

wire [1:0] ALUOp;
wire [1:0] IDEXAluOp;

wire MemWrite;
wire IDEXMemWrite;

wire ALUSrc;
wire IDEXALUSrc;

wire RegWrite;
wire IDEXRegWrite;
wire EXMEMRegWrite;
wire MEMWBRegWrite;
```

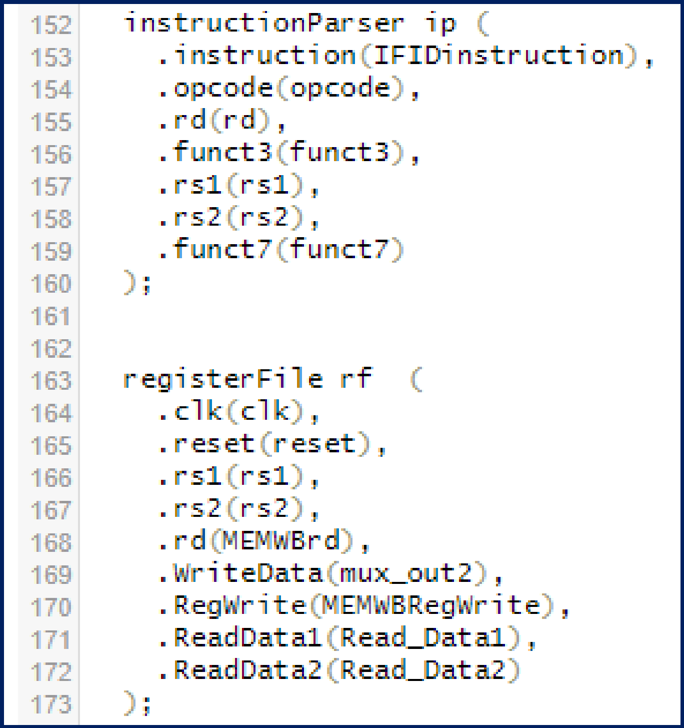


```
wire [63:0] mux_out1;
wire [63:0] MEMWBmux_out1;

wire [63:0] mux_out2;
wire [63:0] EXMEMmux_out2;
wire [63:0] MEMWBmux_out2;

wire [63:0] mux_out3;

wire [63:0] threemux_out1;

wire [63:0] threemux_out2;
wire [63:0] EXMEMthreemux_out2;

wire zero;
wire EXMEMzero;

wire [1:0] forwardA;
wire [1:0] forwardB;
```

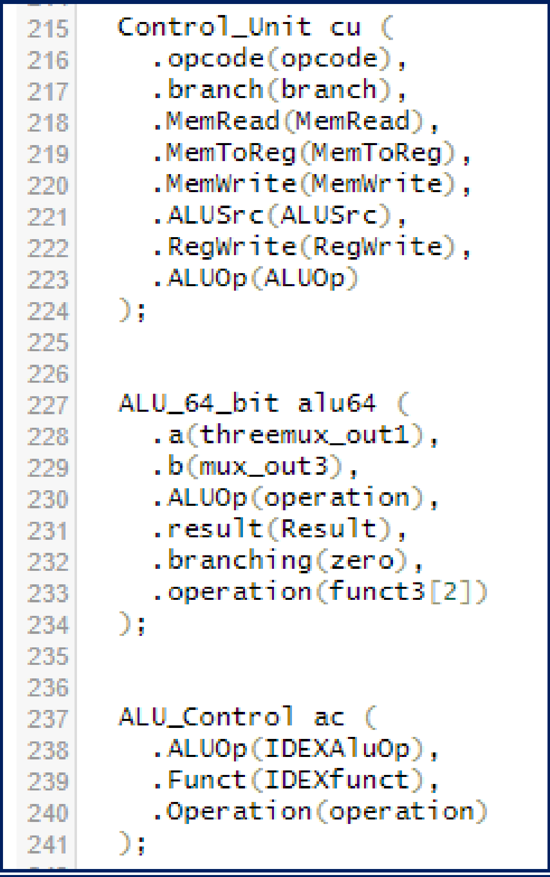
Making the connections



```
Program_Counter pc (
  .PC_in(mux_out1),
  .clk(clk),
  .reset(reset),
  .PC_out(PC_Out)
);

Instruction_Memory im (
  .Inst_Address(PC_Out),
  .Instruction(instruction)
);

adder a1 (
  .a(PC_Out),
  .b(64'd4),
  .out(add_out1)
);

adder a2 (
  .a(IDEXPC_Out),
  .b(IDEXimm_data << 1),
  .out(add_out2)
);
```

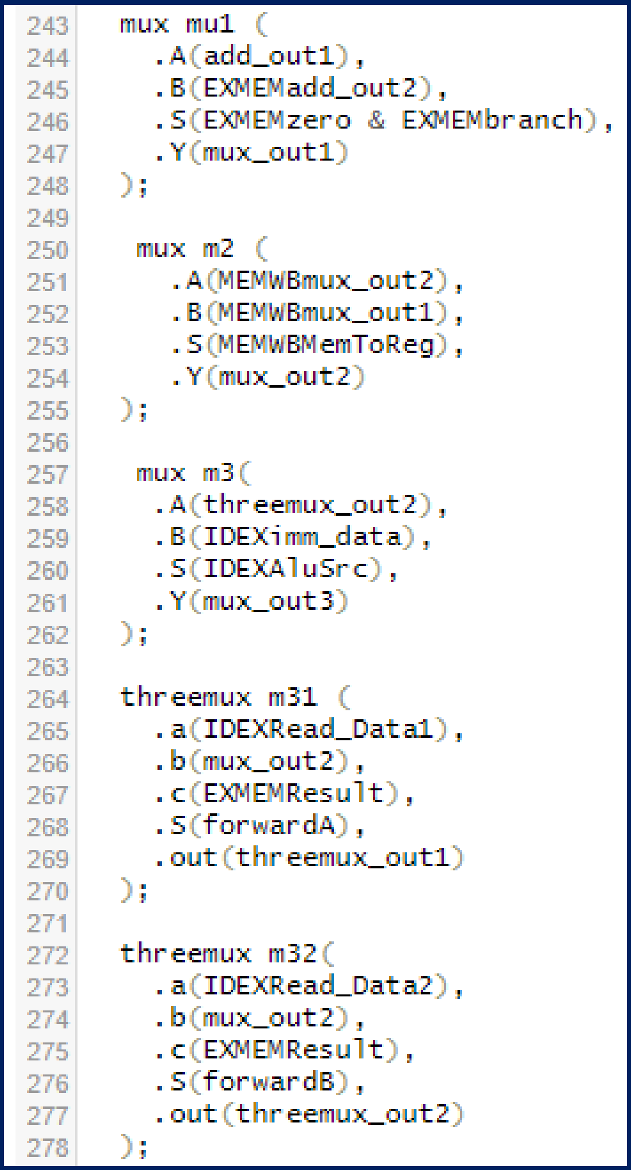


```
instructionParser ip (
  .instruction(IFIDinstruction),
  .opcode(opcode),
  .rd(rd),
  .funct3(funct3),
  .rs1(rs1),
  .rs2(rs2),
  .funct7(funct7)
);


registerFile rf  (
  .clk(clk),
  .reset(reset),
  .rs1(rs1),
  .rs2(rs2),
  .rd(MEMWBrd),
  .WriteData(mux_out2),
  .RegWrite(MEMWBRegWrite),
  .ReadData1(Read_Data1),
  .ReadData2(Read_Data2)
);
```

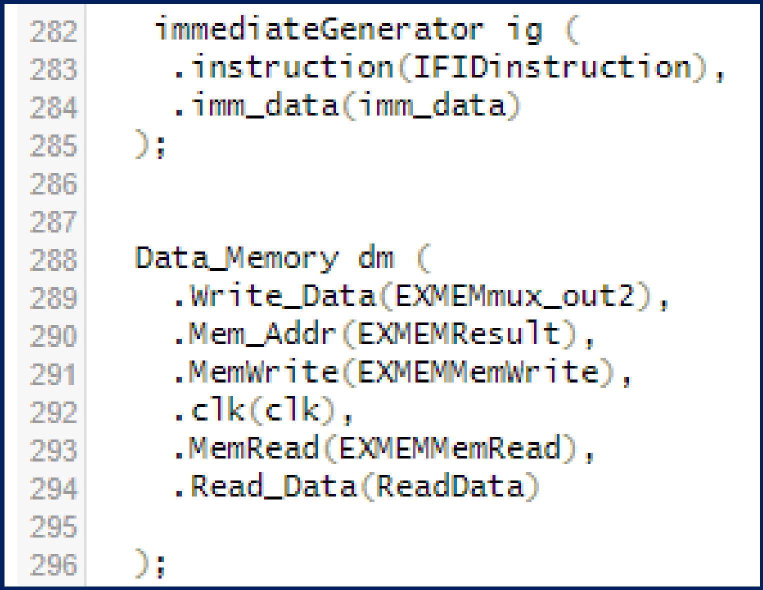


```
Control_Unit cu (
  .opcode(opcode),
  .branch(branch),
  .MemRead(MemRead),
  .MemToReg(MemToReg),
  .MemWrite(MemWrite),
  .ALUSrc(ALUSrc),
  .RegWrite(RegWrite),
  .ALUOp(ALUOp)
);


ALU_64_bit alu64 (
  .a(threemux_out1),
  .b(mux_out3),
  .ALUOp(operation),
  .result(Result),
  .branching(zero),
  .operation(funct3[2])
);


ALU_Control ac (
  .ALUOp(IDEXAluOp),
  .Funct(IDEXfunct),
  .Operation(operation)
);
```

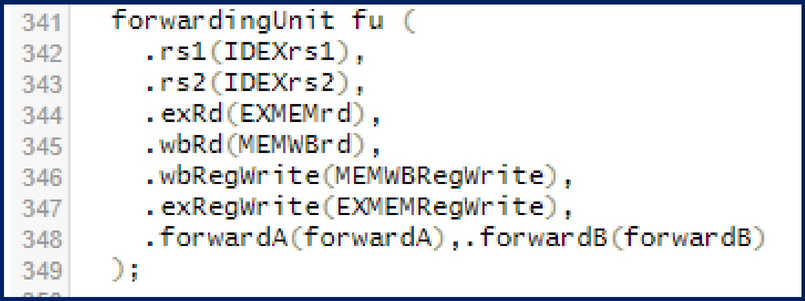
Making the connections



```
mux mu1 (
    .A(add_out1),
    .B(EXMEMadd_out2),
    .S(EXMEMzero & EXMEMbranch),
    .Y(mux_out1)
);

 mux m2 (
    .A(MEMWBmux_out2),
    .B(MEMWBmux_out1),
    .S(MEMWBMemToReg),
    .Y(mux_out2)
);

 mux m3(
    .A(threemux_out2),
    .B(IDEXimm_data),
    .S(IDEXAluSrc),
    .Y(mux_out3)
);

threemux m31 (
    .a(IDEXRead_Data1),
    .b(mux_out2),
    .c(EXMEMResult),
    .S(forwardA),
    .out(threemux_out1)
);

threemux m32(
    .a(IDEXRead_Data2),
    .b(mux_out2),
    .c(EXMEMResult),
    .S(forwardB),
    .out(threemux_out2)
);
```

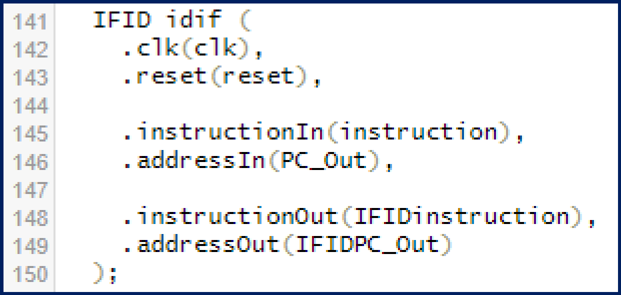


```
 immediateGenerator ig (
  .instruction(IFIDinstruction),
  .imm_data(imm_data)
);


Data_Memory dm (
  .Write_Data(EXMEMmux_out2),
  .Mem_Addr(EXMEMResult),
  .MemWrite(EXMEMMemWrite),
  .clk(clk),
  .MemRead(EXMEMMemRead),
  .Read_Data(ReadData)

);
```

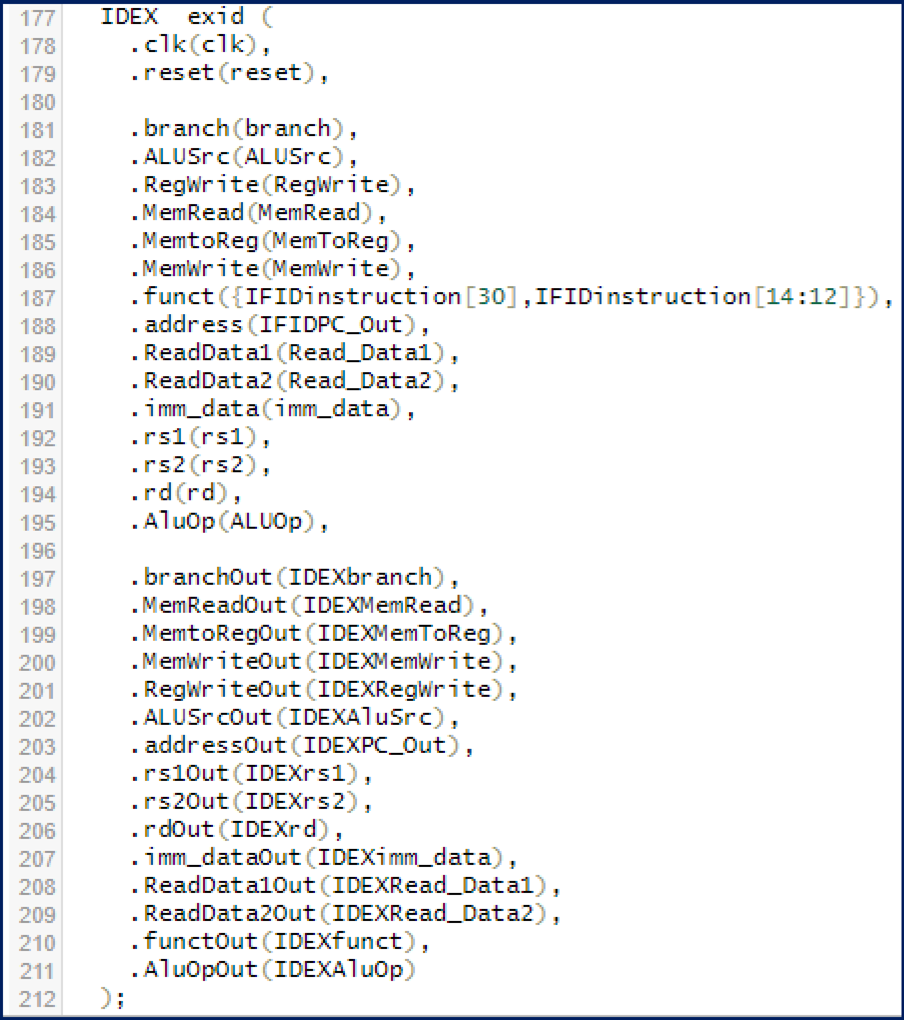


```
forwardingUnit fu (
  .rs1(IDEXrs1),
  .rs2(IDEXrs2),
  .exRd(EXMEMrd),
  .wbRd(MEMWBrd),
  .wbRegWrite(MEMWBRegWrite),
  .exRegWrite(EXMEMRegWrite),
  .forwardA(forwardA),.forwardB(forwardB)
);
```

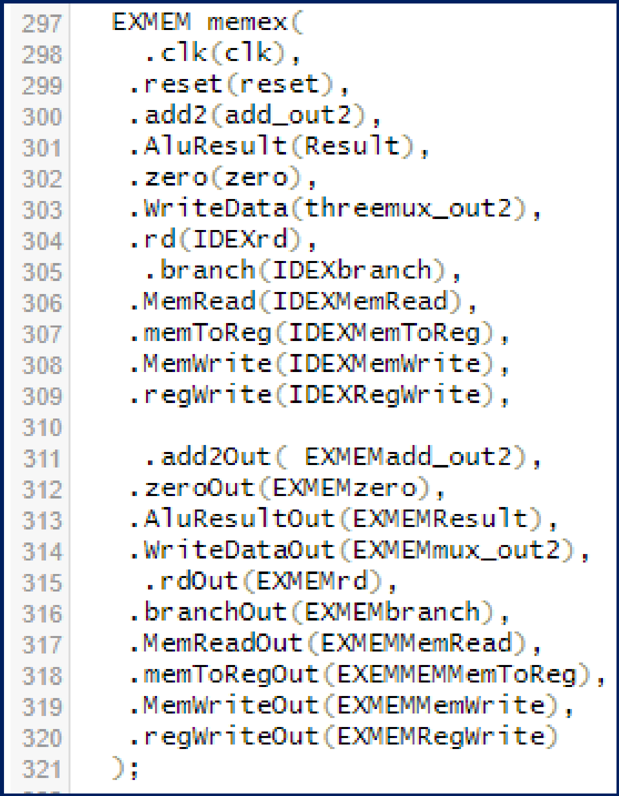
Making the connections



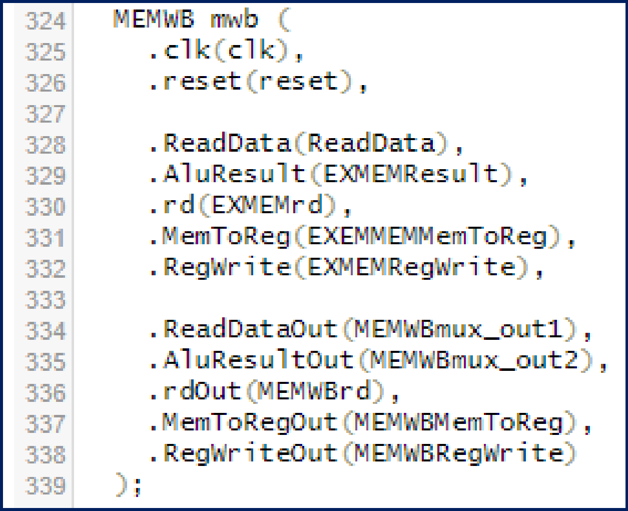
```
IFID idif (
  .clk(clk),
  .reset(reset),

  .instructionIn(instruction),
  .addressIn(PC_Out),

  .instructionOut(IFIDinstruction),
  .addressOut(IFIDPC_Out)
);
```



```
IDEX  exid (
  .clk(clk),
  .reset(reset),

  .branch(branch),
  .ALUSrc(ALUSrc),
  .RegWrite(RegWrite),
  .MemRead(MemRead),
  .MemtoReg(MemToReg),
  .MemWrite(MemWrite),
  .funct({IFIDinstruction[30],IFIDinstruction[14:12]}),
  .address(IFIDPC_Out),
  .ReadData1(Read_Data1),
  .ReadData2(Read_Data2),
  .imm_data(imm_data),
  .rs1(rs1),
  .rs2(rs2),
  .rd(rd),
  .AluOp(ALUOp),

  .branchOut(IDEXbranch),
  .MemReadOut(IDEXMemRead),
  .MemtoRegOut(IDEXMemToReg),
  .MemWriteOut(IDEXMemWrite),
  .RegWriteOut(IDEXRegWrite),
  .ALUSrcOut(IDEXAluSrc),
  .addressOut(IDEXPC_Out),
  .rs1Out(IDEXrs1),
  .rs2Out(IDEXrs2),
  .rdOut(IDEXrd),
  .imm_dataOut(IDEXimm_data),
  .ReadData1Out(IDEXRead_Data1),
  .ReadData2Out(IDEXRead_Data2),
  .functOut(IDEXfunct),
  .AluOpOut(IDEXAluOp)
);
```



```
EXMEM memex(
   .clk(clk),
  .reset(reset),
  .add2(add_out2),
  .AluResult(Result),
  .zero(zero),
  .WriteData(threemux_out2),
  .rd(IDEXrd),
   .branch(IDEXbranch),
  .MemRead(IDEXMemRead),
  .memToReg(IDEXMemToReg),
  .MemWrite(IDEXMemWrite),
  .regWrite(IDEXRegWrite),

   .add2Out( EXMEMadd_out2),
  .zeroOut(EXMEMzero),
  .AluResultOut(EXMEMResult),
  .WriteDataOut(EXMEMmux_out2),
   .rdOut(EXMEMrd),
  .branchOut(EXMEMbranch),
  .MemReadOut(EXMEMMemRead),
  .memToRegOut(EXMMEMMemToReg),
  .MemWriteOut(EXMEMMemWrite),
  .regWriteOut(EXMEMRegWrite)
);
```



```
MEMWB mwb (
   .clk(clk),
   .reset(reset),

   .ReadData(ReadData),
   .AluResult(EXMEMResult),
   .rd(EXMEMrd),
   .MemToReg(EXEMMEMMemToReg),
   .RegWrite(EXMEMRegWrite),

   .ReadDataOut(MEMWBmux_out1),
   .AluResultOut(MEMWBmux_out2),
   .rdOut(MEMWBrd),
   .MemToRegOut(MEMWBMemToReg),
   .RegWriteOut(MEMWBRegWrite)
);
```

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
```

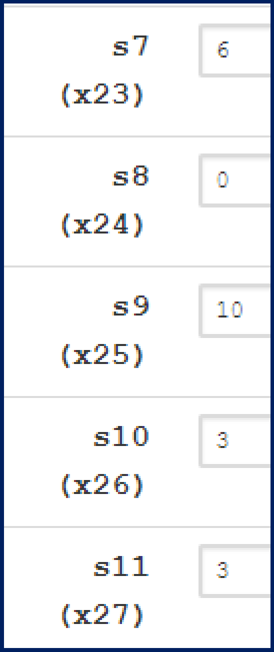
↓

```
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
```

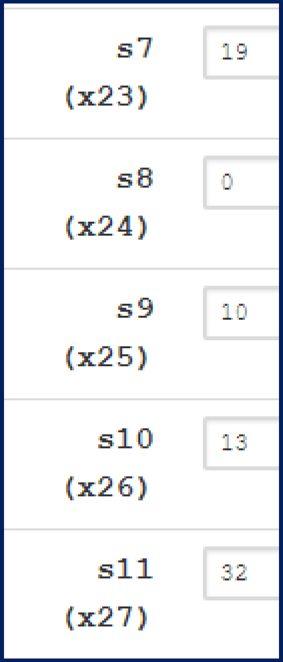
→



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

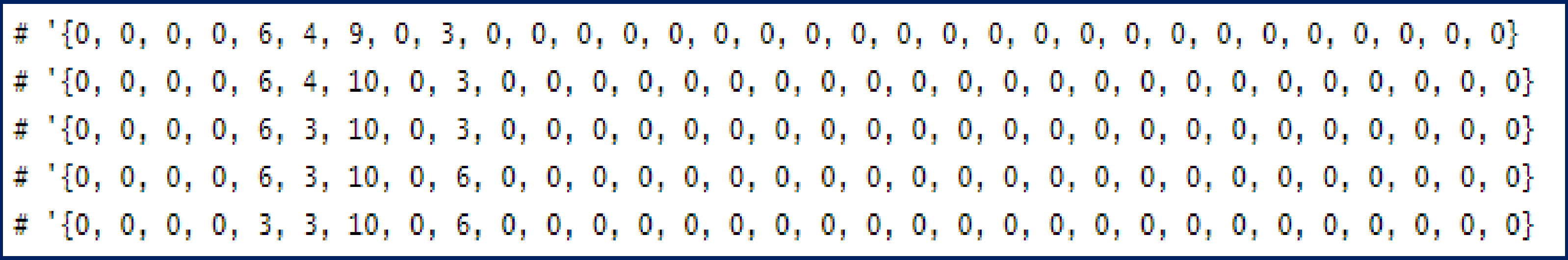
```
addi x23 x0 3
addi x25 x0 9
addi x26 x0 4
addi x27 x0 6

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

→

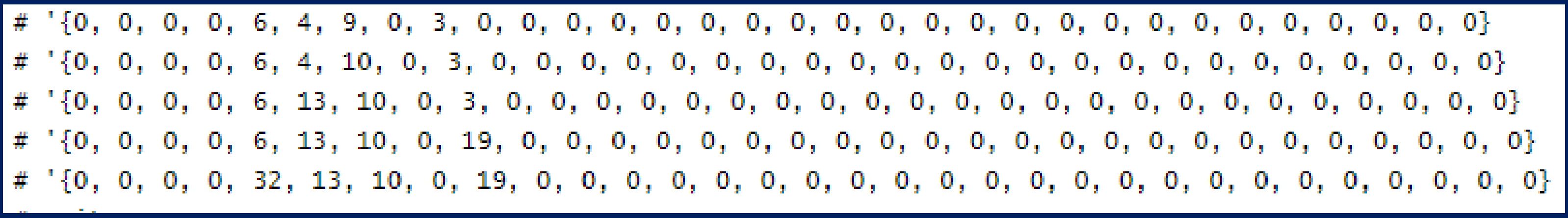


s7 (x23)	19
s8 (x24)	0
s9 (x25)	10
s10 (x26)	13
s11 (x27)	32

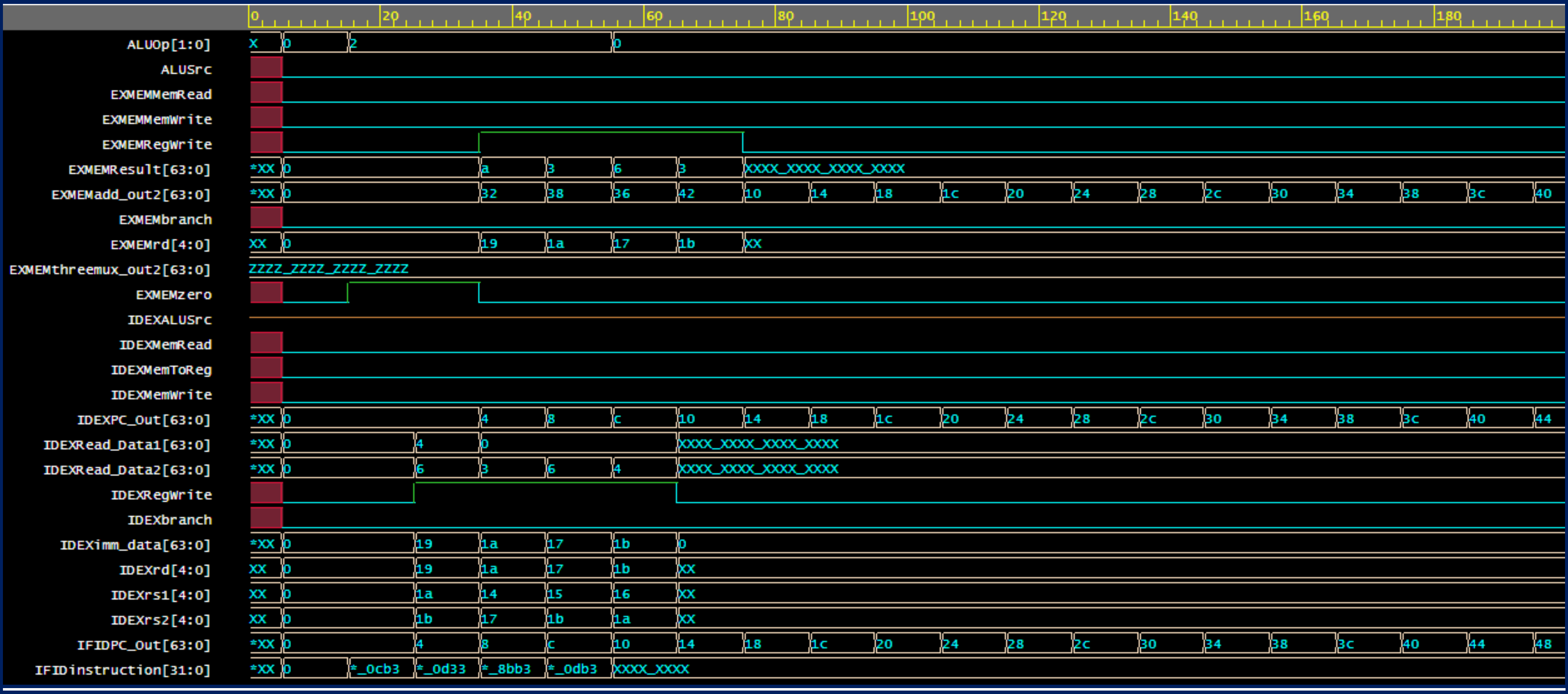
Verilog simulation: We essentially display our registers array to view the changes being made. As we can observe the following registers, they successfully match with our venus simulation registers. (x23 – x27) for both the set of instructions we checked. We made the necessary changes in data memory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

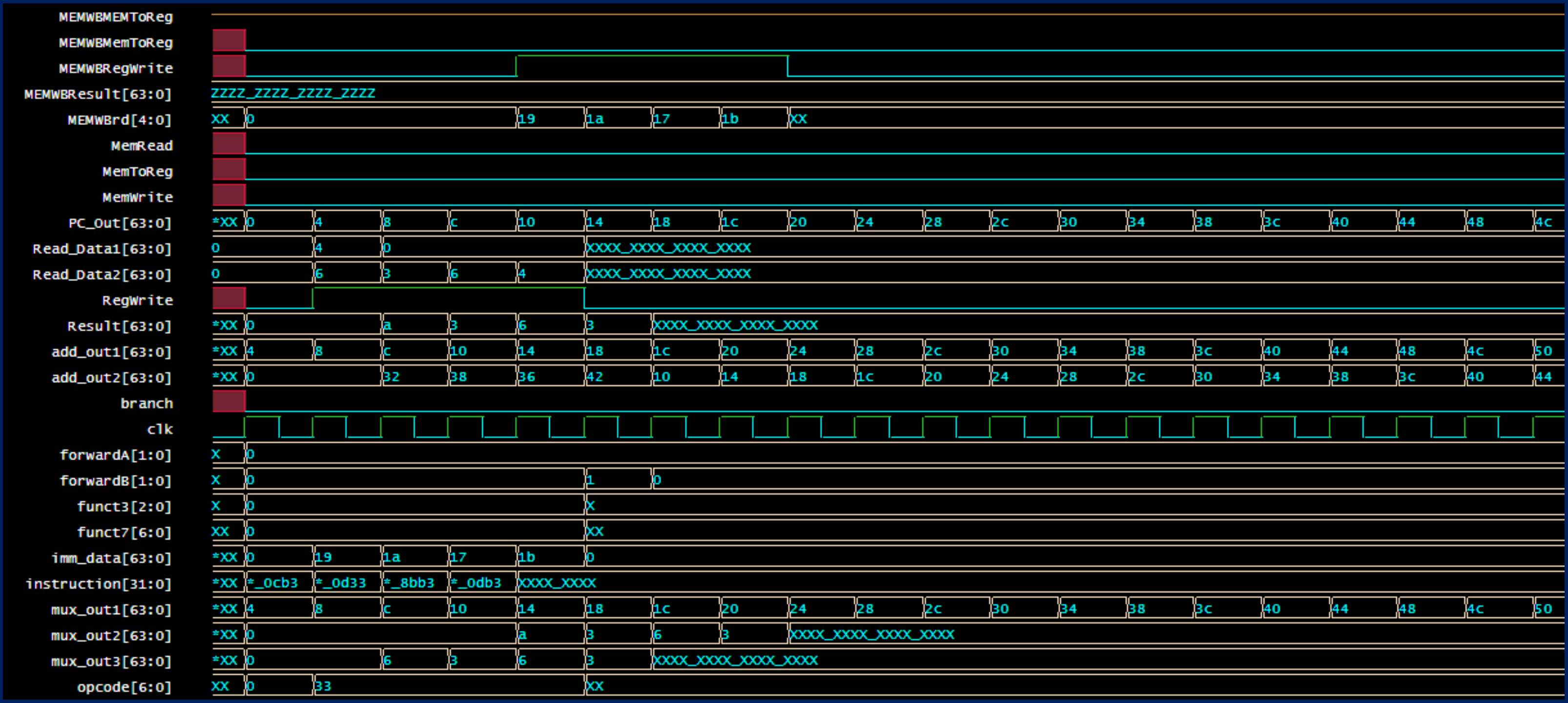


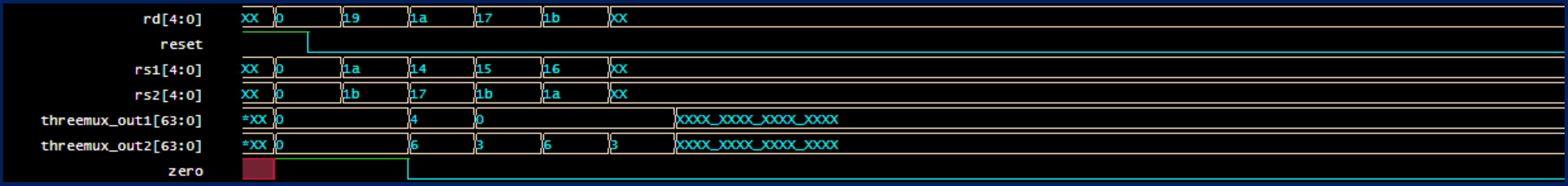
```
# '{0, 0, 0, 0, 6, 4, 9, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
# '{0, 0, 0, 0, 6, 4, 10, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
# '{0, 0, 0, 0, 6, 3, 10, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,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



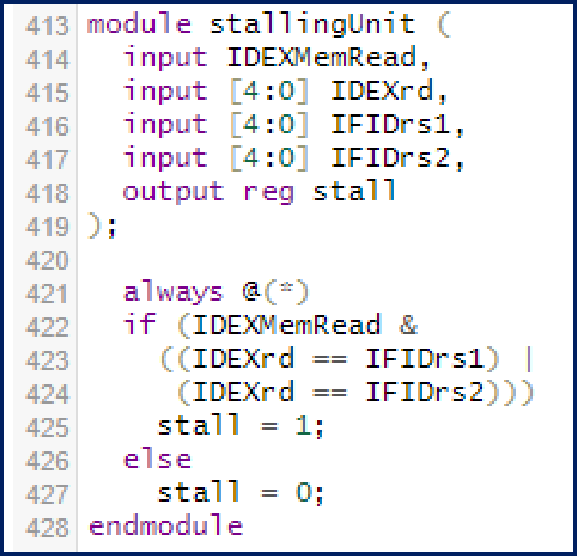

MEMWBMEMToReg
MEMWBMemToReg
MEMWBRegWrite
MEMWBResult[63:0] ZZZZ_ZZZZ_ZZZZ_ZZZZ
MEMWBrd[4:0] XX 0 19 1a 17 1b XX
MemRead
MemToReg
MemWrite
PC_Out[63:0] *XX 0 4 8 c 10 14 18 1c 20 24 28 2c 30 34 38 3c 40 44 48 4c
Read_Data1[63:0] 0 4 0 XXXX_XXXX_XXXX_XXXX
Read_Data2[63:0] 0 6 3 6 4 XXXX_XXXX_XXXX_XXXX
RegWrite
Result[63:0] *XX 0 a 3 6 3 XXXX_XXXX_XXXX_XXXX
add_out1[63:0] *XX 4 8 c 10 14 18 1c 20 24 28 2c 30 34 38 3c 40 44 48 4c 50
add_out2[63:0] *XX 0 32 38 36 42 10 14 18 1c 20 24 28 2c 30 34 38 3c 40 44
branch
clk
forwardA[1:0] X 0
forwardB[1:0] X 0 1 0
funct3[2:0] X 0 X
funct7[6:0] XX 0 XX
imm_data[63:0] *XX 0 19 1a 17 1b 0
instruction[31:0] *XX *_0cb3 *_0d33 *_8bb3 *_0db3 XXXX_XXXX
mux_out1[63:0] *XX 4 8 c 10 14 18 1c 20 24 28 2c 30 34 38 3c 40 44 48 4c 50
mux_out2[63:0] *XX 0 a 3 6 3 XXXX_XXXX_XXXX_XXXX
mux_out3[63:0] *XX 0 6 3 6 3 XXXX_XXXX_XXXX_XXXX
opcode[6:0] XX 0 33 XX


rd[4:0] XX 0 19 1a 17 1b XX
reset
rs1[4:0] XX 0 1a 14 15 16 XX
rs2[4:0] XX 0 1b 17 1b 1a XX
threemux_out1[63:0] *XX 0 4 0 XXXX_XXXX_XXXX_XXXX
threemux_out2[63:0] *XX 0 6 3 6 3 XXXX_XXXX_XXXX_XXXX
zero

Link: Pipelined RISC Processor

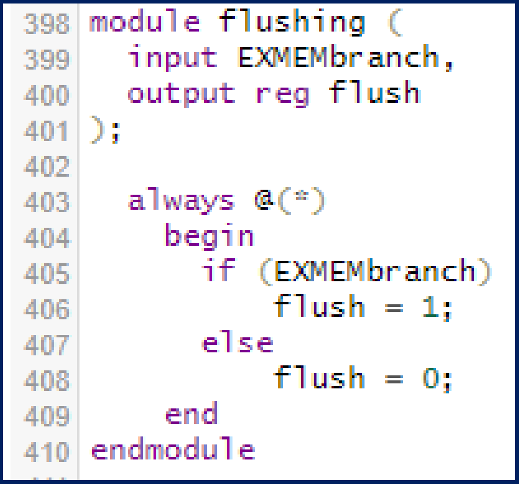
TASK 3 (Link: Pipelined RISC Processor Task 3)

Stalling unit



```
module stallingUnit (
  input IDEXMemRead,
  input [4:0] IDEXrd,
  input [4:0] IFIDrs1,
  input [4:0] IFIDrs2,
  output reg stall
);

  always @(*)
  if (IDEXMemRead &
     ((IDEXrd == IFIDrs1) |
      (IDEXrd == IFIDrs2)))
     stall = 1;
  else
     stall = 0;
endmodule
```

Flushing logic

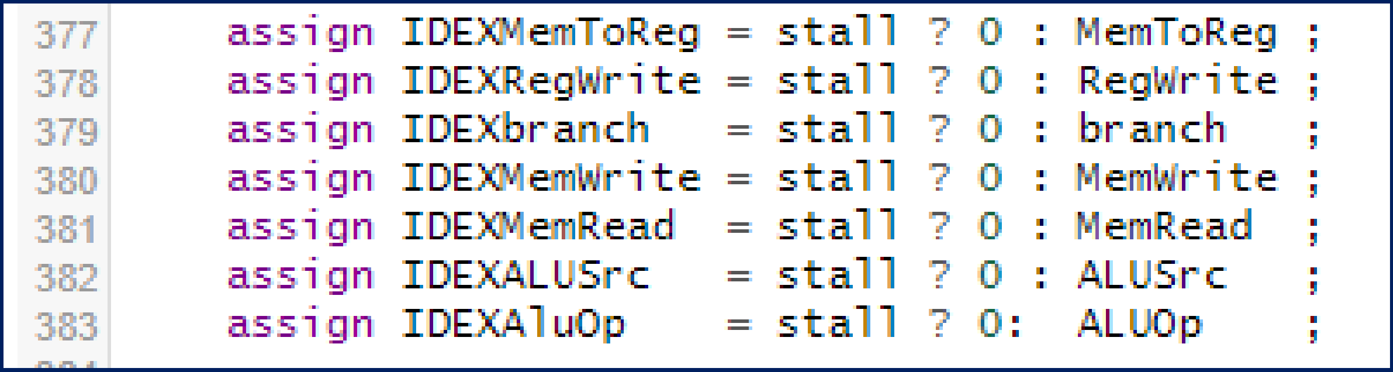


```
module flushing (
  input EXMEMbranch,
  output reg flush
);

  always @(*)
    begin
      if (EXMEMbranch)
          flush = 1;
      else
          flush = 0;
    end
endmodule
```

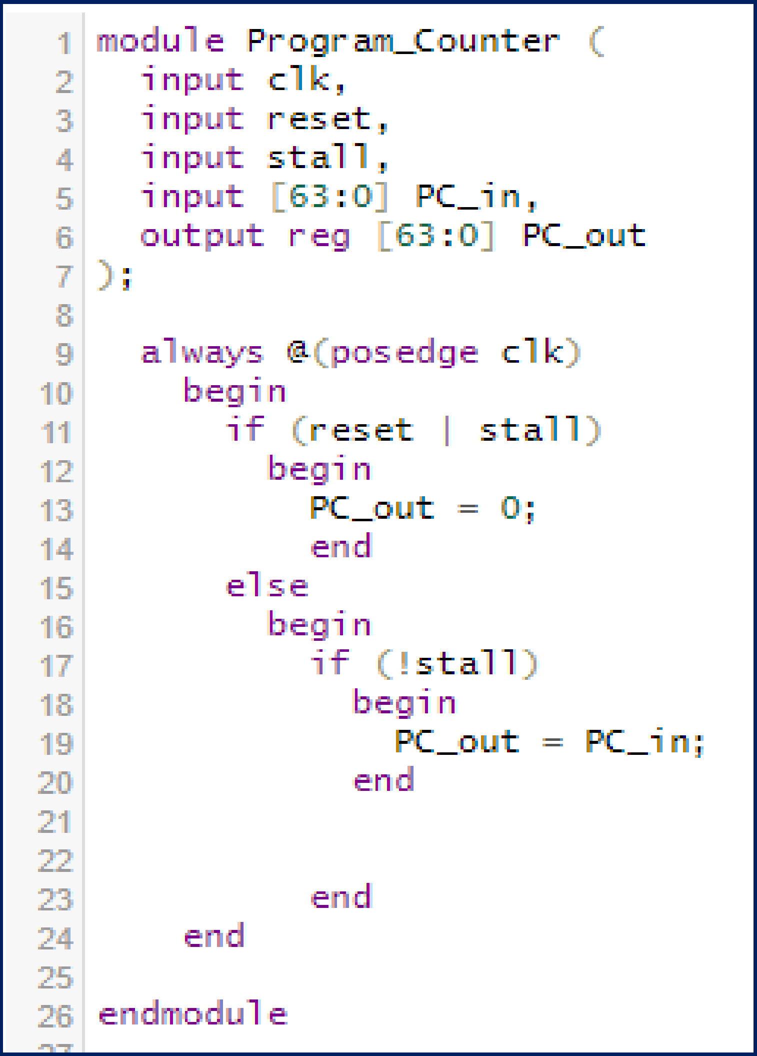
First in the modules (IFID, IDEX, EXMEM, MEMWB) we were making the input equal to the output in the positive edge of the clock and when reset was not zero. Now we will also say do not do this operation if the flush signal is high.

For stalling, we will say, when stalling is high, we do not make the PC_Out equal to PC_in 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

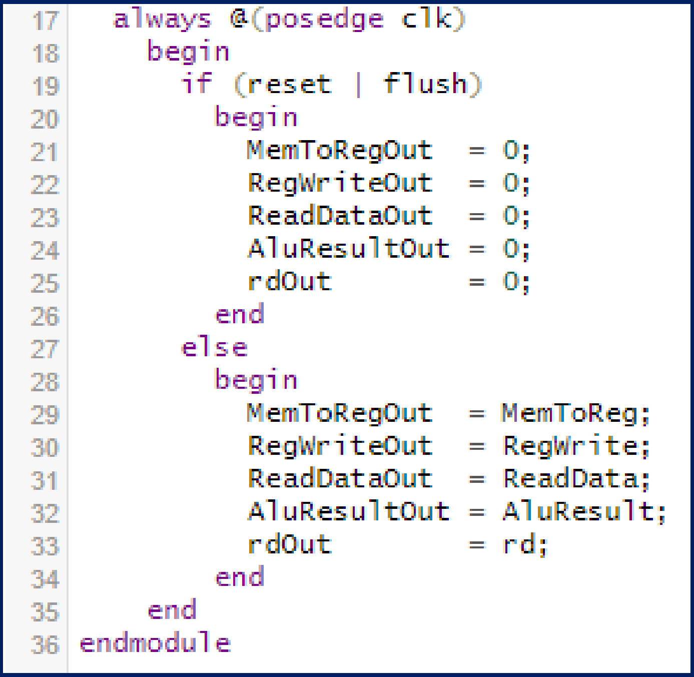


```
    assign IDEXMemToReg = stall ? 0 : MemToReg ;
    assign IDEXRegWrite = stall ? 0 : RegWrite ;
    assign IDEXbranch   = stall ? 0 : branch   ;
    assign IDEXMemWrite = stall ? 0 : MemWrite ;
    assign IDEXMemRead  = stall ? 0 : MemRead  ;
    assign IDEXALUSrc   = stall ? 0 : ALUSrc   ;
    assign IDEXAluOp    = stall ? 0:  ALUOp    ;
```



```
module Program_Counter (
  input clk,
  input reset,
  input stall,
  input [63:0] PC_in,
  output reg [63:0] PC_out
);

  always @(posedge clk)
    begin
      if (reset | stall)
        begin
          PC_out = 0;
          end
      else
        begin
          if (!stall)
            begin
              PC_out = PC_in;
            end


          end
    end

endmodule
```

We will alter our pipeline modules according to this



```
  always @(posedge clk)
    begin
      if (reset | flush)
        begin
          MemToRegOut  = 0;
          RegWriteOut  = 0;
          ReadDataOut  = 0;
          AluResultOut = 0;
          rdOut        = 0;
        end
      else
        begin
          MemToRegOut  = MemToReg;
          RegWriteOut  = RegWrite;
          ReadDataOut  = ReadData;
          AluResultOut = AluResult;
          rdOut        = rd;
        end
    end
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]