Single Cycle Processor - Lab7

by

Yamin Yee

Anthony Paguio

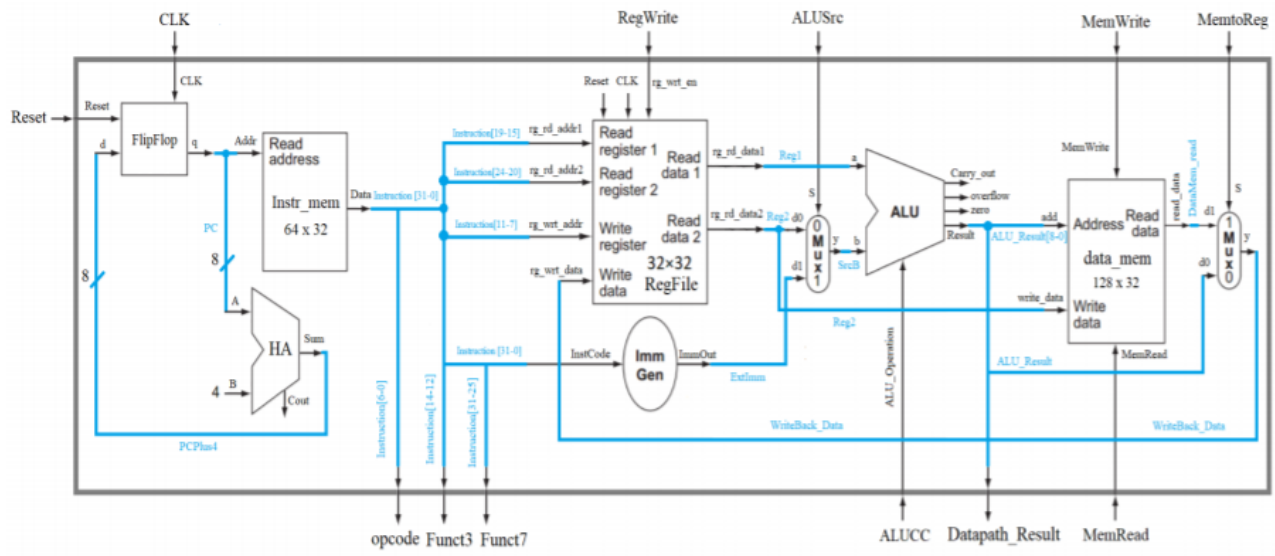
CECS - 440- Computer Architecture

May 8, 2020

Single Cycle Processor

The single cycle processor's instruction execution starts by using the counter to apply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by the fields of that instruction. Once the operands have been fetched, they can operate on to the compute a memory address and equality check. The result for the ALU must be written to a register as well. If the operation is load or store, the result of ALU is used as an address to either store a value from the registers or load a value from memory into the registers. The result from the ALU or memory is written back into the register file.

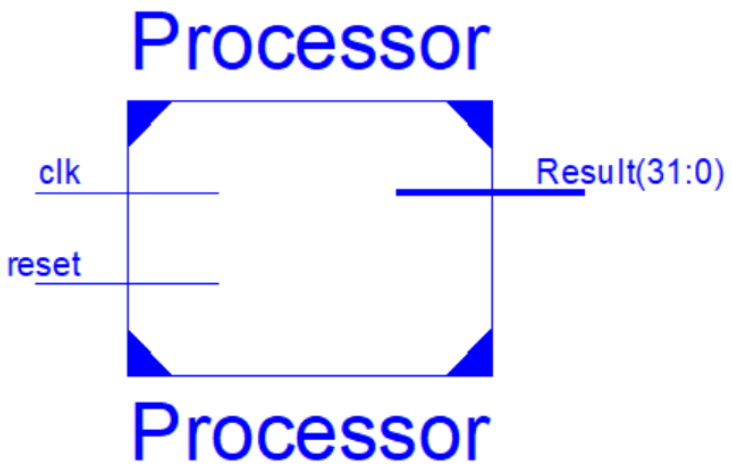
Under the Single Cycle Processor, there are controls signals which are also the inputs, under this project I use the name as RegWrite, ALUSrc, MemWrite, MemtoReg, MemRead which are controlled by the control but since we are not working on control module yet in this project. The given top level is the top level we are given to implement.

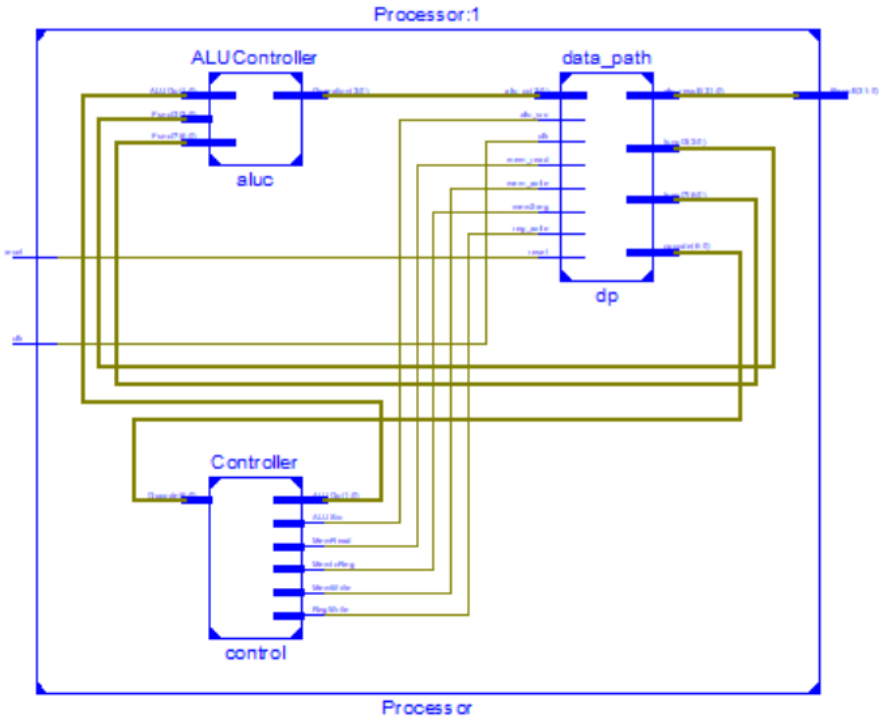


Top Level For Single Cycle Processor

Schematic of Processor

Top of Processor



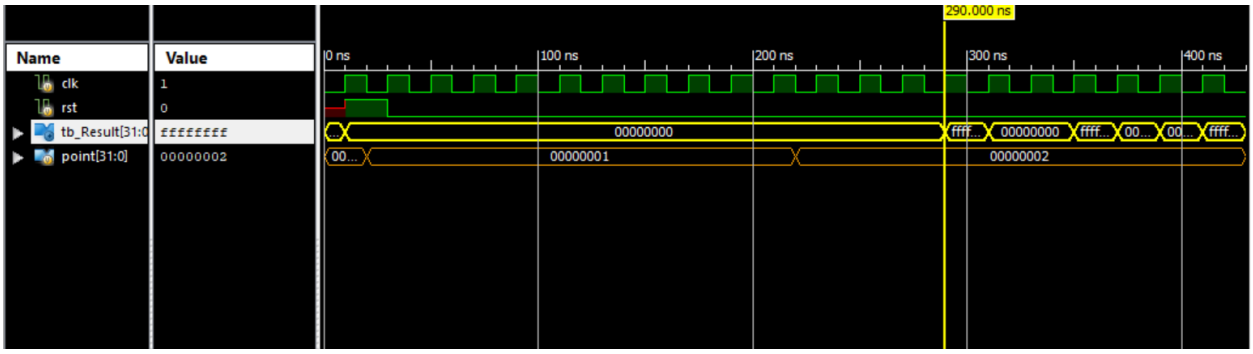


This is the complete design of our single cycle processor. Under our processor, there are ALU control and Controller as you see under our schematic. Processor verilog is simple since we just had to look up the wires of each design connection and just connect each module together.

Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:54:33 05/07/2020
// Design Name:
// Module Name:    Processor
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module processor(

   input clk, reset,
   output [31:0]Result

    );

//wires
wire [6:0] Funct7;
wire [2:0] Funct3;
wire [6:0] Opcode;
wire ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite;
wire [1:0] ALUOp;
wire [3:0] Operation;

//Connected Modules

//Data Path
data_path dp( .clk(clk) , .reset(reset) , .reg_write(RegWrite) , .mem2reg(MemtoReg) ,
              .alu_src(ALUSrc), .mem_write(MemWrite) , .mem_read(MemRead) ,
              .alu_cc(Operation) , .opcode(Opcode) , .funct7(Funct7) , .funct3(Funct3) ,
              .alu_result(Result) );



//ALU Controller
ALUController  aluc( .ALUOp(ALUOp), .Funct7(Funct7), .Funct3(Funct3), .Operation(Operation) );


//Controller

 Controller control( .Opcode(Opcode), .ALUSrc(ALUSrc), .MemtoReg(MemtoReg), .RegWrite(RegWrite),
                     .MemRead(MemRead), .MemWrite(MemWrite), .ALUOp(ALUOp) );



endmodule
```

Simulation



We got into the problem that our tb_Result wasn't shwoning the way we wanted, we tried to fix it but everything is working and all the modules are working individually due to the waveform and testbench we tested. We checked all the wires and every port connection and all the wires are used. So I don't see anything wrong but at the same time, we really cannot find the way to fix it.

TestBench

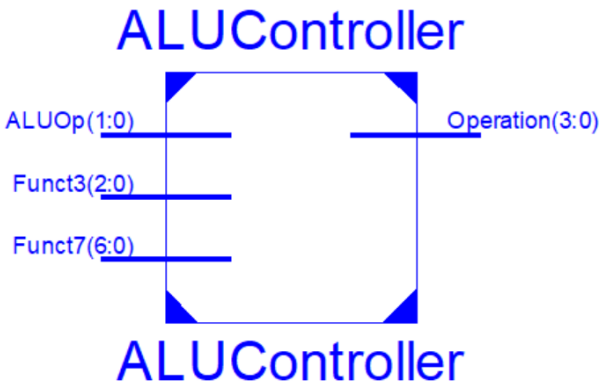
```
module tb_processor;

   /** Clock & reset **/
reg clk , rst ;
wire [31:0] tb_Result ;

Processor  processor_inst(
   . clk ( clk ),
   . reset ( rst ) ,
   . Result ( tb_Result )
);


always begin
#10;
clk = ~ clk ;
end
initial begin
clk = 0;
@( posedge clk );
rst = 1;
@( posedge clk );
rst = 0;
end
```

```
integer point =0;

always @ (*)
begin
#20;
if ( tb_Result == 32'h00000000 ) // and
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h00000001 ) // addi
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h00000002 ) // addi
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h00000004 ) // addi
begin
point = point + 1;
point = point + 1;
end ;
#20;
if ( tb_Result == 32'h00000005 ) // addi
begin
point = point + 1;
end ;
#20;
if ( tb_Result == 32 'h00000007 ) // addi
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h00000008 ) // addi
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h0000000b )// addi
begin
point = point + 1;
#20;
if ( tb_Result == 32'h00000003 ) // add
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'hfffffffe ) // sub
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h00000000 ) // and
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h00000005 ) // or
begin
point = point + 1;
end
# 20;
if ( tb_Result == 32'h00000001 )// SLT
begin
point = point + 1;
```

```
#20;
if ( tb_Result == 32'hffffffff4 ) // NOR
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h000004D2 ) // andi
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'hffffff8d7 ) // ori
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h00000001 ) // SLT
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'hffffffb2c ) // nori
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'hffffffb2c ) // nori
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h00000030 ) // sw
begin
point = point + 1;
end
#20;
if ( tb_Result == 32'h00000030 ) // lw
begin
   point = point + 1;
   end
   $display ( "%s %d" ," The number of correct test cases is :" , point );
end


   initial begin
   #430;
   $finish;
   end

endmodule
```

ALU Controller

Schematic

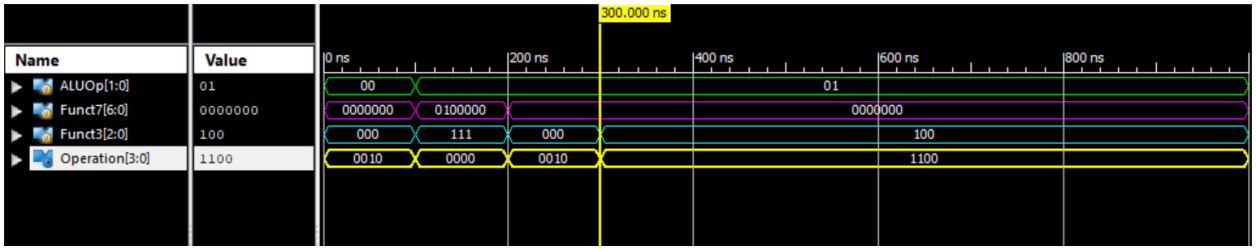


Our inputs of the ALU controller are the ALUOp, Funct3, Funct7, and output is Operation. ALUOp comes from the Controller and Funct3 and Funct7 come from the Datapath. We use the table which was provided in order to get the result.

Verilog Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:56:13 05/07/2020
// Design Name:
// Module Name:    ALUController
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module ALUController( ALUOp, Funct7, Funct3, Operation );
input [1:0] ALUOp;
input [6:0] Funct7;
input [2:0] Funct3;
output reg [3:0] Operation;

wire   [9:0] Funct7_Funct3;
assign Funct7_Funct3 = { Funct7, Funct3 };

always@(ALUOp , Funct7_Funct3) begin
if(ALUOp == 0 )
   Operation <= 4'b0010; // LW and SWE use add
else if(ALUOp == 1)
   Operation <= 4'b0110; // Subtract
   case(Funct7_Funct3)
       10'b0000000_111: Operation <= 4'b0000; //AND
       10'b0000000_110: Operation <= 4'b0001; //OR
       10'b0000000_100: Operation <= 4'b1100; //NOR
       10'b0000000_010: Operation <= 4'b0111; //SLT
       10'b0000000_000: Operation <= 4'b0010; //ADD
       10'b0100000_000: Operation <= 4'b0110; //SUB
       10'b0100000_111: Operation <= 4'b0000; //ANDI
       10'b0100000_110: Operation <= 4'b0001; //ORI
       10'b0100000_100: Operation <= 4'b1100; //NORI

       10'b0100000_010: Operation <= 4'b0111; //SLTI
       10'b0100000_000: Operation <= 4'b0010; //ADDI
       10'b0100000_010: Operation <= 4'b0010; //LW
       10'b1000000_010: Operation <= 4'b0010; //SW
   endcase
end

endmodule
```

Simulation



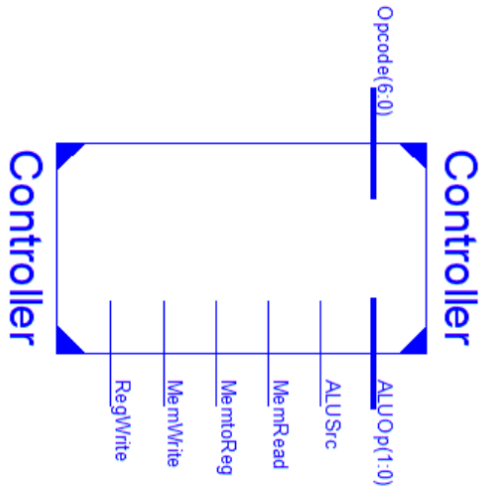
TestBench

```
module alucont_tb;

    // Inputs
    reg [1:0] ALUOp;
    reg [6:0] Funct7;
    reg [2:0] Funct3;

    // Outputs
    wire [3:0] Operation;

    // Instantiate the Unit Under Test (UUT)
    ALUController uut (
        .ALUOp(ALUOp),
        .Funct7(Funct7),
        .Funct3(Funct3),
        .Operation(Operation)
    );

    initial begin
        // Initialize Inputs
        ALUOp = 0;
        Funct7 = 0;
        Funct3 = 0;

        // Wait 100 ns for global reset to finish
```

```
        // Wait 100 ns for global reset to finish
        //ANDI
        #100;
        ALUOp = 1;
        Funct7 = 7'b0100000;
        Funct3 = 3'b111;

        //ADD
        #100;
        ALUOp = 1;
        Funct7 = 7'b0000000;
        Funct3 = 3'b000;

        #100;
        //10'b0000000_100: Operation <= 4'b1100; //NOR
        ALUOp = 1;
        Funct7 = 7'b0000000;
        Funct3 = 3'b100;

        #100;

        // Add stimulus here

    end
```

Controller

Schematic

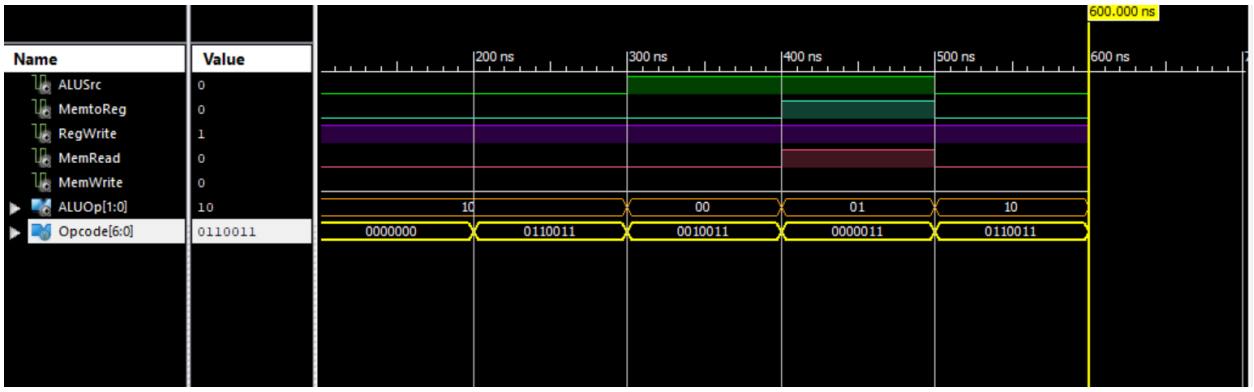


Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:56:48 05/07/2020
// Design Name:
// Module Name:    Controller
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module Controller(
    Opcode,
    ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite,
    ALUOp
     );

input [6:0] Opcode;

output reg ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite;
output reg [1:0] ALUOp;

always@(*)begin
   /*Default*/
   ALUOp[1:0] <= 2'b10;
   ALUSrc     <= 1'b0;
   MemtoReg   <= 1'b0;
   MemRead    <= 1'b0;
   MemtoReg   <= 1'b0;
   MemWrite   <= 1'b0;
   RegWrite   <= 1'b1;

   case(Opcode)
      7'b0110011: begin                // AND, OR, ADD, SUB, SLT, NOR
                  MemtoReg   <= 1'b0;
                  MemWrite   <= 1'b0;
                  MemRead    <= 1'b0;
                  ALUSrc     <= 1'b0;
                  RegWrite   <= 1'b1;
                  ALUOp[1:0] <= 2'b10;
                  end
```

```
      7'b0010011: begin                    // ANDI, ORI, ADDI,SLTI, NORI
                  MemtoReg   <= 1'b0;
                  MemWrite   <= 1'b0;
                  MemRead    <= 1'b0;
                  ALUSrc     <= 1'b1;
                  RegWrite   <= 1'b1;
                  ALUOp[1:0] <= 2'b00;
                  end

      7'b0000011: begin                    // LW
                  MemtoReg   <= 1'b1;
                  MemWrite   <= 1'b0;
                  MemRead    <= 1'b1;
                  ALUSrc     <= 1'b1;
                  RegWrite   <= 1'b1;
                  ALUOp[1:0] <= 2'b01;
                  end

      7'b0110011: begin                    //SW
                  MemtoReg   <= 1'b0;
                  MemWrite   <= 1'b1;
                  MemRead    <= 1'b0;
                  ALUSrc     <= 1'b1;
                  RegWrite   <= 1'b0;
                  ALUOp[1:0] <= 2'b01;
                 end
   endcase
end

endmodule
```

Schematic

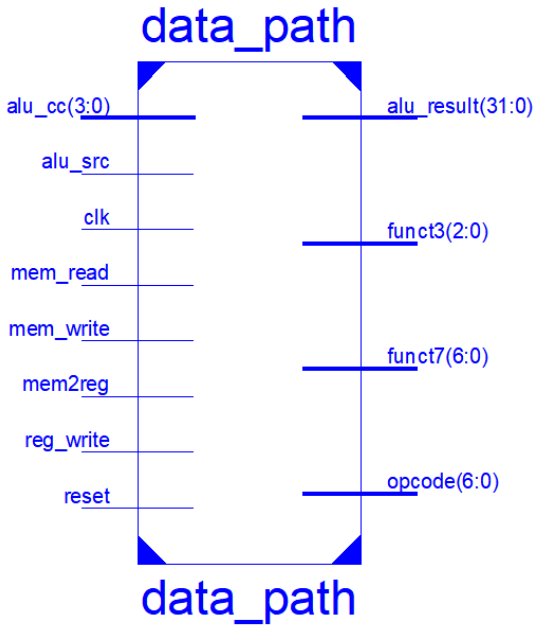


TestBench

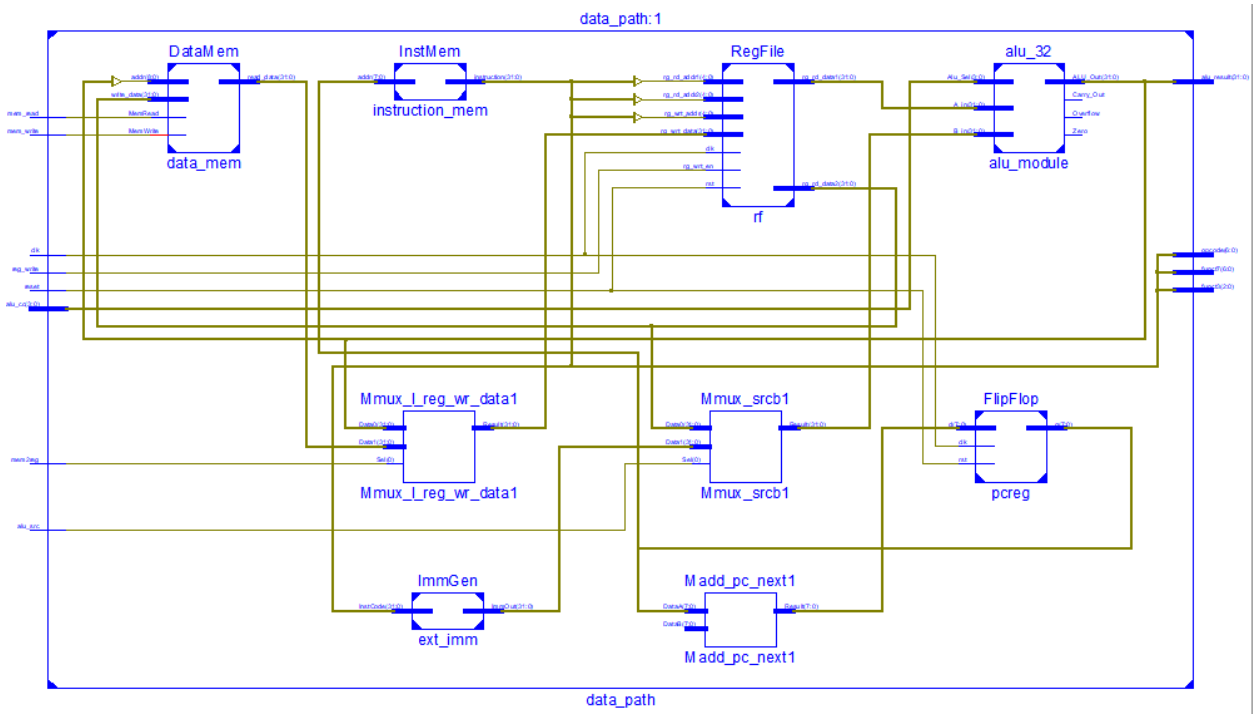
```
module control_tb;

    // Inputs
    reg [6:0] Opcode;

    // Outputs
    wire ALUSrc;
    wire MemtoReg;
    wire RegWrite;
    wire MemRead;
    wire MemWrite;
    wire [1:0] ALUOp;

    // Instantiate the Unit Under Test (UUT)
    Controller uut (
        .Opcode(Opcode),
        .ALUSrc(ALUSrc),
        .MemtoReg(MemtoReg),
        .RegWrite(RegWrite),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .ALUOp(ALUOp)
    );


   initial begin
      // Initialize Inputs
      #100;
      Opcode = 7'b0000000;

      // Wait 100 ns for global reset to finish
      #100;
      Opcode = 7'b0110011;

      #100;
      Opcode = 7'b0010011;

      #100;
      Opcode = 7'b0000011;

      #100;
      Opcode = 7'b0110011;

      #100;

   end

endmodule
```

DataPath

Schematic



Schematic for Top Level



Blocks under the Top Level

Our top level diagram has 9 modules such as flip flop(pc), instruction memory, alu, mux(2 muxes), immediate generator, register file, adder, and data memory. We created the data memory but since we don't need to create the data memory yet we attached it.

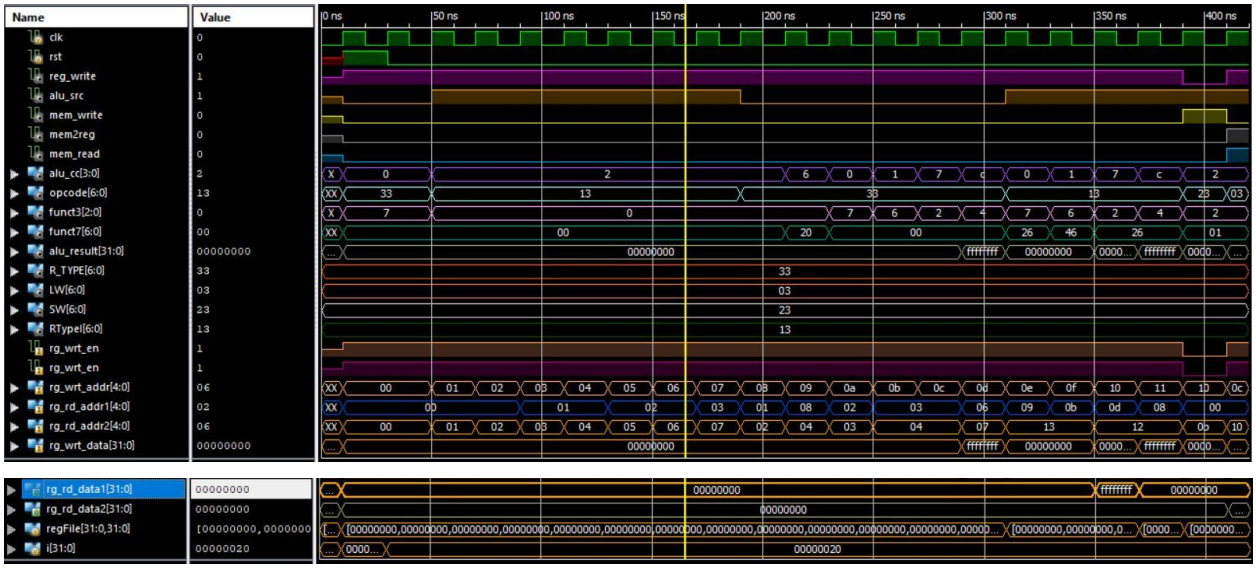
Verilog Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:55:15 05/07/2020
// Design Name:
// Module Name:    data_path
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module data_path #(
parameter PC_W = 8, // Program Counter
parameter INS_W = 32 , // Instruction Width
parameter RF_ADDRESS = 5 , // Register File Address
parameter DATA_W = 32 , // Data WriteData
parameter DM_ADDRESS = 9 , // Data Memory Address
parameter ALU_CC_W = 4 // ALU Control Code Width
)(
input clk ,
input reset ,
input reg_write ,
input mem2reg ,
input alu_src ,
input mem_write ,
input mem_read ,
input [ ALU_CC_W -1:0] alu_cc ,
output [6:0] opcode ,
output [6:0] funct7 ,
output [2:0] funct3 ,
output [DATA_W-1:0] alu_result );

wire [ PC_W-1:0] pc ;
wire [ PC_W-1:0] pc_next ;
wire [ INS_W -1:0] instruction ;
```

```
wire [ 31:0] l_alu_result ;
wire [ DATA_W-1:0] reg1 , reg2 ;
wire [ DATA_W-1:0] l_read_data ;
wire [ DATA_W-1:0] l_reg_wr_data ;
wire [ DATA_W-1:0] srcb ;
wire [ DATA_W-1:0] extimm ;

// next pc
assign pc_next = pc + 8'h4;

//FlipFlop(clk, rst, d, q );
FlipFlop pcreg( .clk(clk) , .rst(reset) , .d(pc_next) , .q(pc) );

//adder
//adder add(.A(), .Sum(pc_next));

// instruction memory
//InstMem( [7:0] addr, [31:0] instruction);
InstMem instruction_mem( .addr(pc) , .instruction(instruction) );


assign opcode = instruction [6:0];
assign funct7 = instruction [31:25];
assign funct3 = instruction [14:12];


// register file
//clk, rst, rg_wrt_en, rg_wrt_addr, rg_rd_addr1, rg_rd_addr2, rg_wrt_data, rg_rd_data1, rg_rd_data2);
RegFile rf (
. clk ( clk ),
. rst ( reset ),
. rg_wrt_en ( reg_write ),
. rg_wrt_addr ( instruction [11:7] ),
. rg_rd_addr1 ( instruction [19:15] ) ,
. rg_rd_addr2 ( instruction [24:20] ) ,
. rg_wrt_data ( l_reg_wr_data ),
. rg_rd_data1 ( reg1 ),
. rg_rd_data2 ( reg2 ));


assign l_reg_wr_data = mem2reg ? l_read_data : l_alu_result ;

// sign extend

//ImmGen(InstCode, ImmOut);
ImmGen ext_imm ( .InstCode(instruction) , .ImmOut(extimm) );

// alu
assign srcb = alu_src ? extimm : reg2 ;

// alu_32(A_in,B_in, Alu_Sel,ALU_Out,Carry_Out,Zero,Overflow);
alu_32 alu_module ( . A_in ( reg1 ), . B_in ( srcb ), . Alu_Sel ( alu_cc ) ,
                . ALU_Out ( l_alu_result ), . Carry_Out () , . Zero () ,
                . Overflow ());

assign alu_result = l_alu_result ;

//DataMem( MemRead, MemWrite, addr, write_data, read_data);
DataMem data_mem ( . MemRead ( mem_read ),. MemWrite ( mem_write ),  .addr (  l_alu_result[ DM_ADDRESS-1:0]) ,
                .write_data( reg2 ) ,  . read_data ( l_read_data ) );

endmodule
```

Under the top level module, I created the wires to connect with nine other modules as you see in the verilog code.

Simulation


Name
Value
clk
rst
reg_write
alu_src
mem_write
mem2reg
mem_read
alu_cc[3:0]
opcode[6:0]
funct3[2:0]
funct7[6:0]
alu_result[31:0]
R_TYPE[6:0]
LW[6:0]
SW[6:0]
RTypeI[6:0]
rg_wrt_en
rg_wrt_en
rg_wrt_addr[4:0]
rg_rd_addr1[4:0]
rg_rd_addr2[4:0]
rg_wrt_data[31:0]
rg_rd_data1[31:0]
rg_rd_data2[31:0]
regFile[31:0,31:0]
i[31:0]
0 ns
50 ns
100 ns
150 ns
200 ns
250 ns
300 ns
350 ns
400 ns

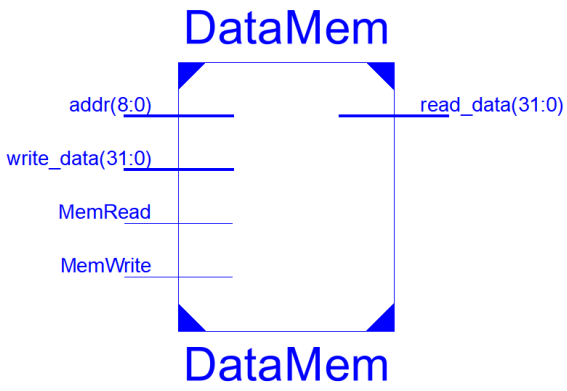
TestBench

```
module data_tb_top();

 /** Clock & reset **/
   reg clk , rst ;
   always begin
   #10;
   clk = ~clk;
   end

   initial begin
   clk = 0;
   @( posedge clk );
   rst = 1;
   @( posedge clk );
   rst = 0;
   end

   /** DUT Instantiation **/
   wire reg_write ;
   wire mem2reg ;
   wire alu_src ;
   wire mem_write ;
   wire mem_read ;
   wire [ 3 : 0 ] alu_cc ;
   wire [ 6 : 0 ] opcode ;
   wire [ 6 : 0 ] funct7 ;
   wire [ 2 : 0 ] funct3 ;
   wire [ 31 : 0 ] alu_result;


   data_path dp_inst (
   .clk ( clk ) ,
   . reset ( rst ) ,
   . reg_write ( reg_write ) ,
   . mem2reg ( mem2reg ) ,
   . alu_src ( alu_src ) ,
   . mem_write ( mem_write ) ,
   . mem_read ( mem_read ) ,
   . alu_cc ( alu_cc ) ,
   . opcode ( opcode ) ,
   . funct7 ( funct7 ) ,
   . funct3 ( funct3 ) ,
   . alu_result ( alu_result )
```

```
);

/** Stimulus **/
wire [ 6 : 0 ] R_TYPE , LW , SW , RTypeI ;

assign R_TYPE = 7'b0110011 ;
assign LW = 7'b0000011 ;
assign SW = 7'b0100011 ;
assign RTypeI = 7'b0010011 ;


assign alu_src = ( opcode == LW || opcode == SW || opcode == RTypeI );
assign mem2reg = ( opcode == LW );
assign reg_write = ( opcode == R_TYPE || opcode == LW || opcode == RTypeI );
assign mem_read = ( opcode == LW );
assign mem_write = ( opcode == SW );

assign alu_cc = (( opcode == R_TYPE || opcode == RTypeI )
               && ( funct7 == 7'b0000000 ) && ( funct3 == 3'b000 )) ? 4'b0010 :
                  (( opcode == R_TYPE || opcode == RTypeI )
               && ( funct7 == 7'b0100000 )) ? 4'b0110 :
                  (( opcode == R_TYPE || opcode == RTypeI )
               && ( funct7 == 7'b0000000 ) && ( funct3 == 3'b100 )) ? 4'b1100 :
                  (( opcode == R_TYPE || opcode == RTypeI )
               && ( funct7 == 7'b0000000 ) && ( funct3 == 3'b110 )) ? 4'b0001 :
                  (( opcode == R_TYPE || opcode == RTypeI )
               && ( funct7 == 7'b0000000 ) && ( funct3 == 3'b111 )) ? 4'b0000 :
                  (( opcode == R_TYPE || opcode == RTypeI )
               && ( funct7 == 7'b0000000 ) && ( funct3 == 3'b010 )) ? 4'b0111 :
                  (( opcode == R_TYPE || opcode == RTypeI )
               && ( funct3 == 3'b100 )) ? 4'b1100 :
                  (( opcode == R_TYPE || opcode == RTypeI )
               && ( funct3 == 3'b110 )) ? 4'b0001 :
                  (( opcode == R_TYPE || opcode == RTypeI )
               && ( funct3 == 3'b010 )) ? 4'b0111 :
                  (( opcode == LW || opcode == SW)
               && ( funct3 == 3'b010 ))? 4'b0010 : 0;

   initial begin
   #420;
   $finish ;
   end
endmodule
```

Data Memory

Schematic

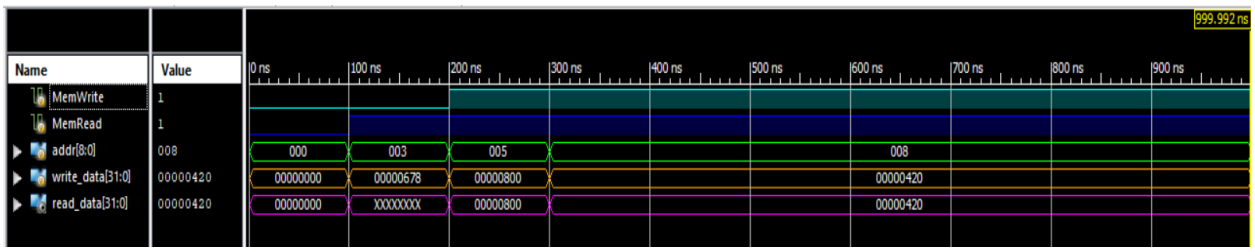


Data memory is byte addressable.To address 128x4 = 512 bytes, we need 9 bits address which will come from 9 LSBs of the outputs of ALU. To read the data we need an address and read enable signal(MemRead).

Verilog Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:00:28 05/07/2020
// Design Name:
// Module Name:    DataMem
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module DataMem( MemRead, MemWrite, addr, write_data, read_data);
   input [8:0] addr;
   input wire [31:0] write_data;
   input  MemWrite, MemRead;

   output [31:0] read_data;

   reg [31:0] Mem[127:0]; //2^8 of 32 bit memory

   assign read_data = MemRead ? Mem[addr[8:0]] : 0;


   //always@(addr,write_data)
   always@(*)
   begin
      if(MemWrite) begin
         Mem[addr[8:0]] <= write_data;
      end
   end

endmodule
```

Simulation



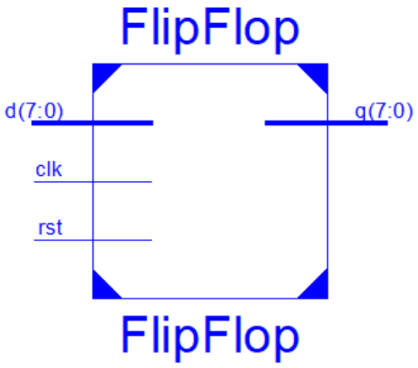
Under this simulation, you will see there is XXXXX for ReadData because I set up MemWrite is zero and MemRead is 1 , therefore , it will not get data and just show XXXXX. Please see my testbench below.

Test Bench

```
module dm_tb;

   // Inputs
   reg [8:0] addr;
   reg MemWrite;
   reg MemRead;
   reg [31:0] write_data;

   // Outputs
   wire [31:0] read_data;

   // Instantiate the Unit Under Test (UUT)
   DataMem uut (
      .addr(addr),
      .MemWrite(MemWrite),
      .MemRead(MemRead),
      .write_data(write_data),
      .read_data(read_data)
   );

   initial begin
      // Initialize Inputs
      addr = 0;
      MemWrite = 0;
      MemRead = 0;
      write_data = 0;

      // Wait 100 ns for global reset to finish
      #100;
      addr = 3;
      MemWrite = 0;
      MemRead = 1;
      write data = 32'h678;

      #100;
      addr = 5;
      MemWrite = 1;
      MemRead = 1;
      write_data = 32'h800;
      // Add stimulus here

      #100;
      addr = 8;
      MemWrite = 1;
      MemRead = 1;
      write_data = 32'h420;

      end

endmodule
```

Flip Flop

Schematic



Flip flop also known as PC is the flop to store signals. It stores the value of its data input signal in the internal memory. It has three inputs and one output. Under my schematic, as you see , my inputs are 8 bits d, clk and rst. My output is 8 bit q. Under the verilog code, you will see the clear procedure which is at the posedge clk and posedge rst, if the rst is active, output will get zero else the output will get the value of d.

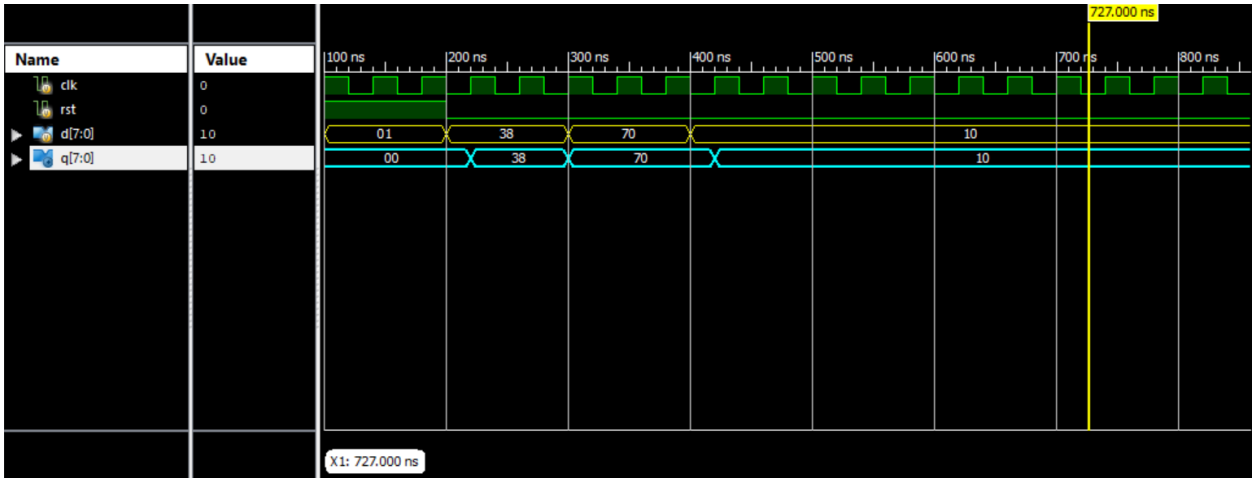
This Flip Flop is also the register to hold the address of the current instruction which will send this address to the instruction memory to read the current instruction.

Verilog Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Class:          CECS 440
// Engineer:       Yamin Yee
//
// Create Date:    18:20:13 04/05/2020
// Design Name:
// Module Name:    FlipFlop
// Project Name:   Signal-cycle Processor
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module FlipFlop(clk, rst, d, q );
//Input
input clk, rst;
input [7:0] d; //8 bits
```

```
always@(posedge clk,posedge rst)
if(rst)
   q <= 8'b0;
else
   q <= d;


endmodule
```

Simulation



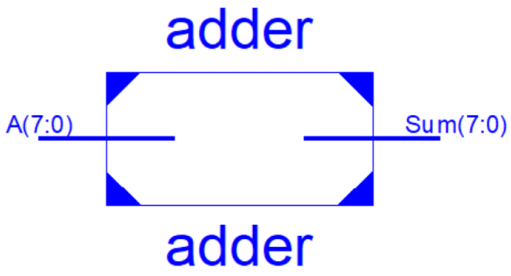
Under the wave from, you can see the right result of the wave which the q will get when the reset is inactive.

TestBench

```
module FlipFlop_test;

   // Inputs
   reg clk;
   reg rst;
   reg [7:0] d;

   // Outputs
   wire [7:0] q;

   // Instantiate the Unit Under Test (UUT)
   FlipFlop uut (
      .clk(clk),
      .rst(rst),
      .d(d),
      .q(q)
   );


  initial begin
     // Initialize Inputs
     clk = 0;
     rst = 0;
     d = 0;
  end

  always#20 clk = ~clk; //20ns clk duration

     // Wait 100 ns for global reset to finish
     initial
     begin
     #100;
     rst = 1; //reset is active
     d = 7'h1;

     #100;
     rst = 0; //reset is inactive
     d = 7'h38;

     #100;
     rst = 0; //reset is inactive
     d = 7'hf0;

       #100;
       rst = 0; //reset is inactive
       d = 7'hf0;

       #100;
       rst = 0; //reset is inactive
       d = 7'h10;

       // Add stimulus here
    end

endmodule
```

Adder

Schematic



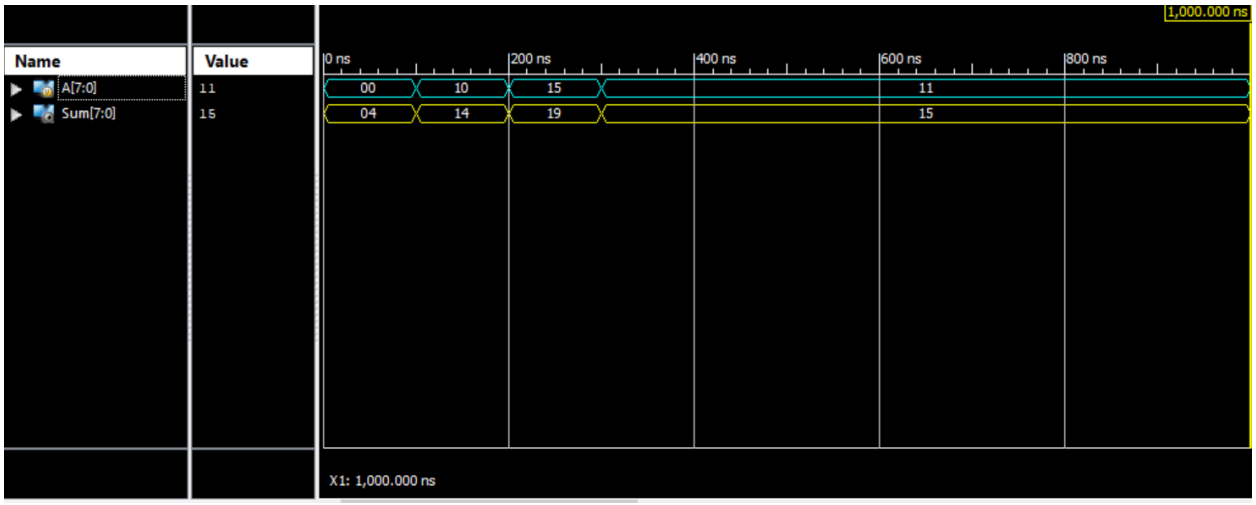
We also need an adder to increment the PC with 4 to get the address of the next instruction. This adder is implemented as shown below. I didn use Cout and B since we won't be using them for the design. But in this module, the input is 8-bit A which is the data getting from PC and also output is Sum which is 8-bit and it will increment the PC by adding 4.

Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    19:48:48 04/05/2020
// Design Name:
// Module Name:    adder
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module adder( A, Sum );
input [7:0] A;
//input [7:0] B ;
output reg [7:0] Sum;
```

```
always@(*)
   begin
      Sum = A + 8'h4;
      //Cout = (A & B) ;
   end

endmodule
```

Simulation Wave



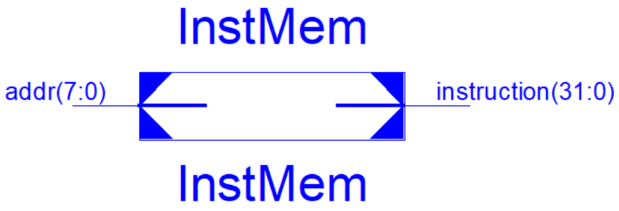
As you see the simulation, Sum increases 4 per each value of A.

Test Bench for Adder

```
module adder_tb;

   // Inputs
   reg [7:0] A;

   // Outputs
   wire [7:0] Sum;

   // Instantiate the Unit Under Test (UUT)
   adder uut (
      .A(A),
      .Sum(Sum)
   );

   initial begin
      // Initialize Inputs
      A = 8'h0;

      // Wait 100 ns for global reset to finish
      #100;

      A =8'h10;



      #100

      A = 8'h15;

      #100;

      A = 8'h11;


      #100;

      // Add stimulus here
   end

endmodule
```

Instruction Memory

Schematic



Instruction memory has one input which is an 8-bit addr and one output which is the instruction we read the memory which is instruction 32-bit. the 8-bit address comes from the PC(flipflip), we address 2^8 bytes and then there are 32 bits which is 4 bytes and so we can store 2^8/4 = 64 instructions in the instruction memory.

In the instruction memory we need internal storage to store the instructions. We register as [31:0] memory [63:0] which is a new type of 2D array and define single memory from the type InstMem. I also need to assign the instruction to get the address from memory.

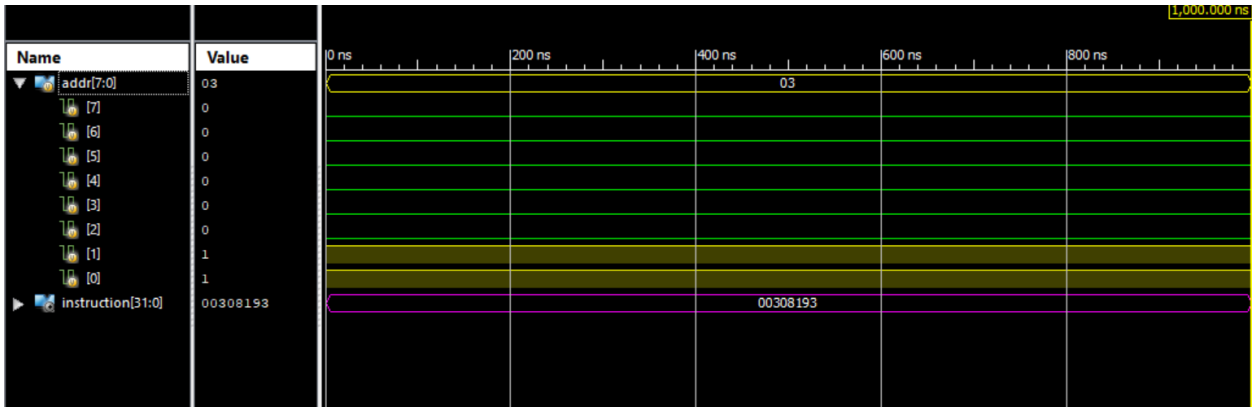
We also need the instruction code and so we also wrote the instruction code as the professor gave.

Verilog Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:26:52 04/05/2020
// Design Name:      Yamin Yee
// Module Name:    Instr_mem
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments: Instruction memory, we need internal storage to store the instructions
//                      We need to define a 2D array to store 64 instruction each with 4 bytes(32bits)
//                      So it is the 64x32 2D array.
//
//////////////////////////////////////////////////////////////////////////////////
module InstMem(
            //Inputs
```

```
            //Inputs
            input [7:0] addr,
            //Outputs
            output [31:0] instruction
            );

//reg [31:0] instruction;
// Instruction Memory for 2D Array for Internal Storage
reg [31:0] memory [63:0];

assign instruction = memory[addr];

   initial
   begin
         // Insturction Codes
         memory[0] <= 32'h00007033; // and r0, r0, r0 32'h00000000
         memory[1] <= 32'h00100093; // addi r1, r0, 1 32'h00000001
         memory[2] <= 32'h00200113; // addi r2, r0, 2 32'h00000002
         memory[3] <= 32'h00308193; // addi r3, r1, 3 32'h00000004
         memory[4] <= 32'h00408213; // addi r4, r1, 4 32'h00000005
         memory[5] <= 32'h00510293; // addi r5, r2, 5 32'h00000007
         memory[6] <= 32'h00610313; // addi r6, r2, 6 32'h00000008

         memory[7] <= 32'h00718393; // addi r7, r3, 7 32'h0000000B
         memory[8] <= 32'h00208433; // add r8, r1, r2 32'h00000003
         memory[9] <= 32'h404404b3; // sub r9, r8, r4 32'hfffffffe
         memory[10]<= 32'h00317533; // and r10, r2, r3 32'h00000000
         memory[11]<= 32'h0041e5b3; // or r11, r3, r4 32'h00000005
         memory[12]<= 32'h0041a633; // if r3 is less than r4 then r12 = 1 32'h00000001
         memory[13]<= 32'h007346b3; // nor r13, r6, r7 32'hfffffff4
         memory[14]<= 32'h4d34f713; // andi r14, r9, "4D3" 32'h000004D2
         memory[15]<= 32'h8d35e793; // ori r15, r11, "8d3" 32'hfffff8d7
         memory[16]<= 32'h4d26a813; // if r13 is less than 32'h000004D2 then r16 = 1 32'h00000001
         memory[17]<= 32'h4d244893; // nori r17, r8, "4D2" 32'hfffffb2C
   end


endmodule
```

Simulation Wave



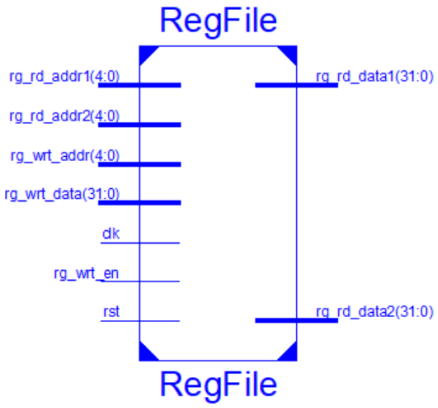
Here is the waveform and as you see it shows the instruction 03 which we assigned under the test bech and then instruction should show the 32-bit value which we already assigned it under the verilog code.

Test Bench

```
module InstMem_test;

    // Inputs
    reg [7:0] addr;
    wire [31:0] instruction;
    // Instantiate the Unit Under Test (UUT)
    InstMem uut (
        .addr(addr),
        .instruction(instruction)
    );

    initial begin
        // Initialize Inputs
        addr = 3;

        // Wait 100 ns for global reset to finish
        #100;



        // Add stimulus here
    end

endmodule
```

Register File

Schematic



Register file consists of a set of registers that can be read and written by supplying a register number to be accessed. For reading from a register file, we only need the register number. From writing to register we need 3 inputs: a register number, data write, clk that controls the writing into the register. The register file has two read ports and one write port.

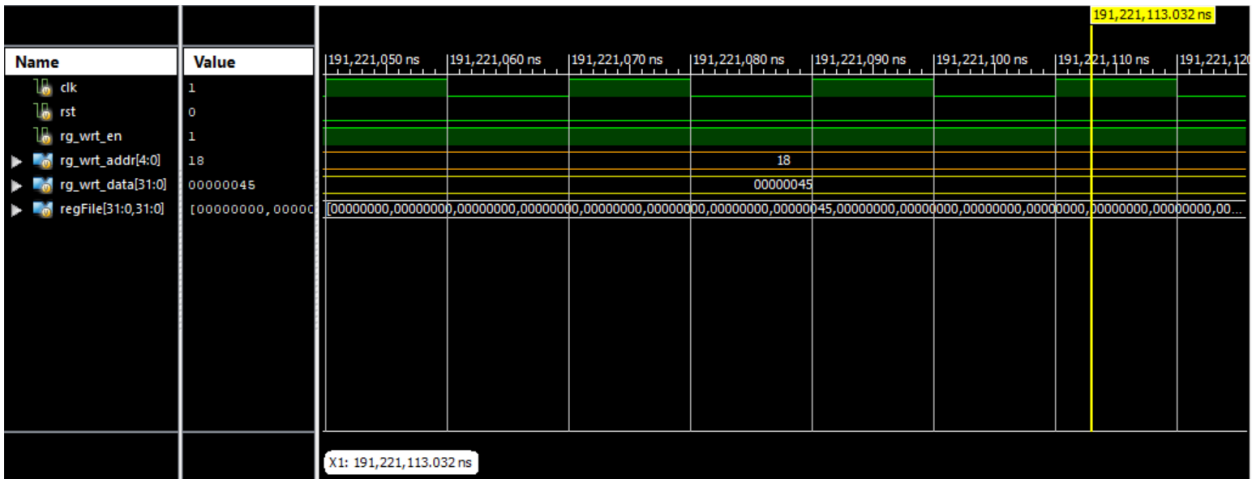
In my module, there are two register addresses : rg_rd_addr1 and rg_rd_addr2. These two input signals have 5 bits with 5 bits we can dress as 32 registers. As you can see from my code, the clk or rst, we send two registers through two output lines rg_rd_data1 and rg_rd_data2. To write the register file, i use if else, if rst is zero, the rg_wrt_en is ‘1’ on then in the rising edge (posedge) of clk, I will write the data from the input line rg_wrt_data to the register number rg_wrt_addr.

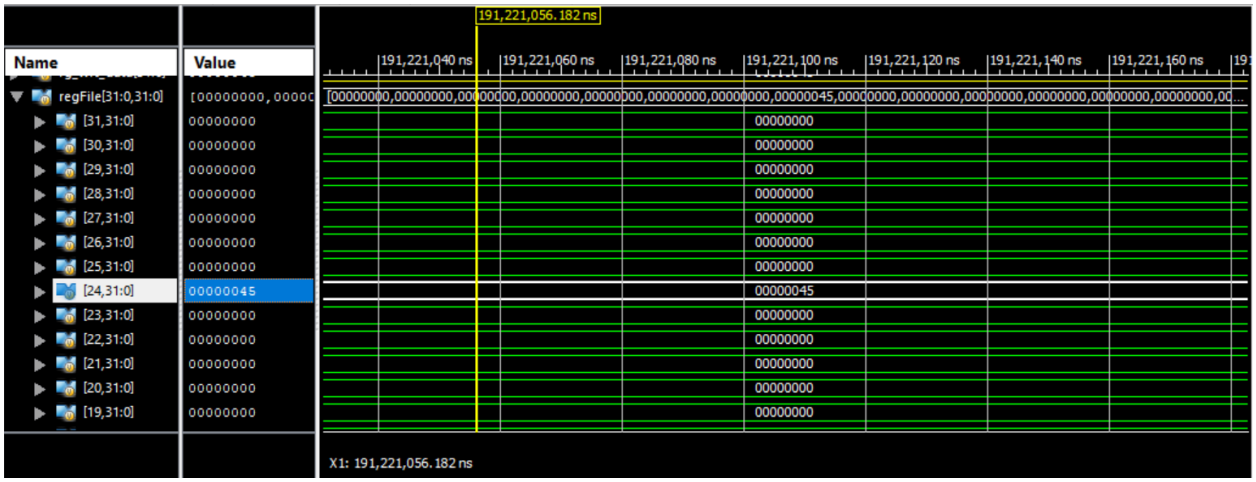
At the reset signal, we should reset the register file from the beginning which is 000000.

Verilog Code

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:52:10 04/05/2020
// Design Name:      Yamin Yee
// Module Name:    RegFile
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module RegFile( clk, rst, rg_wrt_en, rg_wrt_addr, rg_rd_addr1, rg_rd_addr2, rg_wrt_data, rg_rd_data1, rg_rd_data2);
input clk,rst;
input rg_wrt_en;//write
input [4:0] rg_wrt_addr; // Write Register Address
input [4:0] rg_rd_addr1, rg_rd_addr2; // Two Register
input [31:0] rg_wrt_data; // Write Data

output wire [31:0] rg_rd_data1, rg_rd_data2;

// signal declaration
reg [31:0] regFile [31:0];

integer i;

assign rg_rd_data1 = regFile[rg_rd_addr1];
assign rg_rd_data2 = regFile[rg_rd_addr2];

 always @(posedge rst, posedge clk)
    begin
        if (rst) begin
            for(i=0; i<32; i=i+1)
                regFile[i] <= 32'b0;
        end else if(rg_wrt_en) begin
            regFile[rg_wrt_addr] <= rg_wrt_data;
        end
    end


endmodule
```

Simulation Waveform





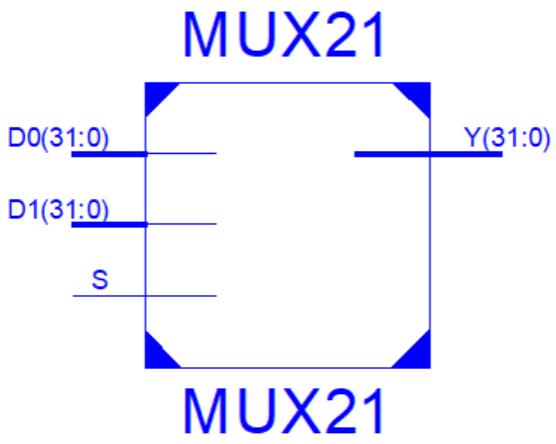
Under this waveform, as you see the rg_wrt_addr is chosen reg number 18 to write into it and then set the rg_wrt_data to 32'h45. Then it shows the value as we see and also when we expand, it shows where it is.

Testbench

```
module RegFile_Test;

   // Inputs
   reg clk;
   reg rst;
   reg rg_wrt_en;
   reg [4:0] rg_wrt_addr;
   reg [4:0] rg_rd_addr1;
   reg [4:0] rg_rd_addr2;
   reg [31:0] rg_wrt_data;


   // Outputs
   wire [31:0] rg_rd_data1;
   wire [31:0] rg_rd_data2;

   // Instantiate the Unit Under Test (UUT)
   RegFile uut (
      .clk(clk),
      .rst(rst),
      .rg_wrt_en(rg_wrt_en),
      .rg_wrt_addr(rg_wrt_addr),
      .rg_rd_addr1(rg_rd_addr1),
      .rg_rd_addr2(rg_rd_addr2),
      .rg_wrt_data(rg_wrt_data),
      .rg_rd_data1(rg_rd_data1),
      .rg_rd_data2(rg_rd_data2)
   );



      always begin
      #10;

      clk = ~clk;
      end

      initial begin
        clk = 0;
        rst = 1;
        #100;
        rst = 0;
        #10;
        rg_wrt_en = 1;
        rg_wrt_addr = 8'h18;
        rg_wrt_data = 8'h0000_0045;
   end


endmodule
```

Result Mux (Mux21)

Schematic



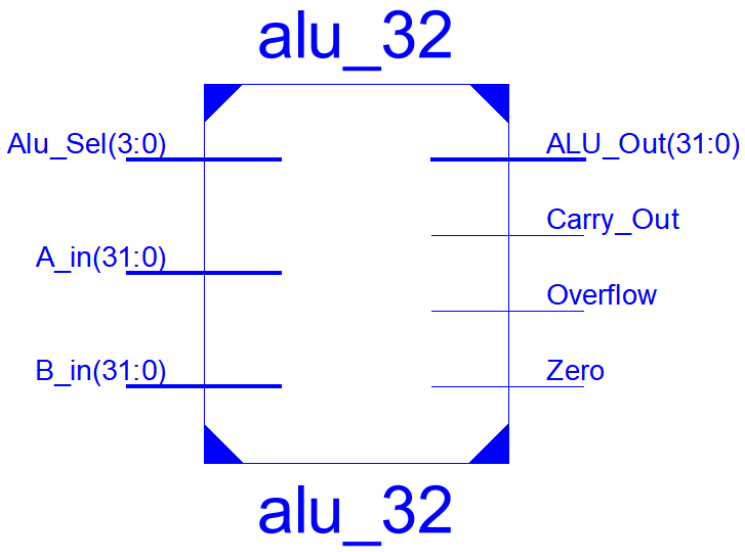
The mux on the output of the DataMemory will decide whether the writing data to the register file should come from the ALU or come from the Data Memory which we haven't done since it is not included in this lab.

Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:      Yamin Yee
//
// Create Date:    19:23:33 04/05/2020
// Design Name:
// Module Name:    MUX21
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module MUX21(D0, D1, S, Y);

input S; //select line
input [31:0] D0, D1;

output [31:0] Y;

assign Y = (!S & D0) | (S & D1);


endmodule
```

ALU

Schematic

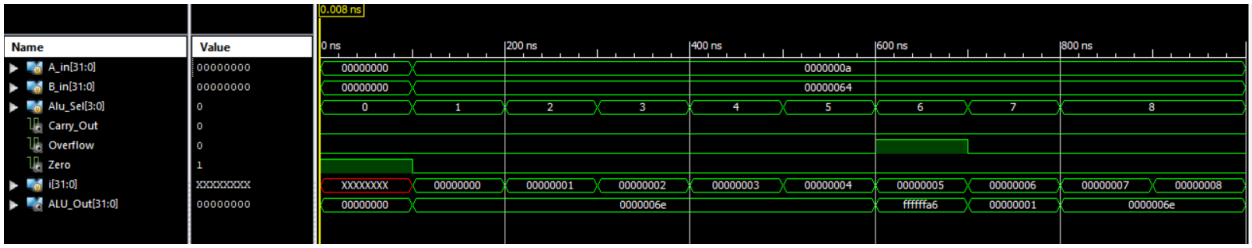


This is our ALU from the previous lab.

Verilog Module

```
module alu_32(

input [31:0] A_in,B_in,
input [3:0] Alu_Sel,
output [31:0] ALU_Out,
output reg Carry_Out,
output Zero,
output reg Overflow = 1'b0

);

reg [31:0] ALU_Result;
reg [32:0] temp;
reg [32:0] twos_com; //to hold 2'sc of second source of ALU

assign ALU_Out = ALU_Result;
assign Zero = (ALU_Result == 0);

always@(*)
begin
   Overflow  = 1'b0;
   Carry_Out = 1'b0;
   case(Alu_Sel)
   4'b0000: ALU_Result = A_in & B_in ;//AND

   4'b0001: ALU_Result = A_in | B_in ;//OR

   4'b0010: // Signed Additon with Overflow and Carry_Out Checking
      begin
      ALU_Result = $signed(A_in) + $signed(B_in) ;
      temp = {1'b0, A_in} + {1'b0, B_in};
      Carry_Out = temp[32];
      if((A_in[31] & B_in[31] & ~ALU_Out[31]) | (~A_in[31] & ~B_in[31] & ALU_Out[31]))
         Overflow = 1'b1;
      else
         Overflow = 1'b0;
      end


      4'b0110: // Signed Subtraction with Overflow  Checking
         begin
         ALU_Result = $signed(A_in) - $signed(B_in) ;
         twos_com = ~(B_in) + 1'b1;
         if((A_in[31] & B_in[31] & ~ALU_Out[31]) | (~A_in[31] & ~B_in[31] & ALU_Out[31]))
               Overflow = 1'b1;
         else
               Overflow = 1'b0;
      end

      4'b0111: // Signed less than camparison
         ALU_Result = ($signed(A_in) < $signed(B_in))?32'd1:32'd0;

      4'b1100: // NOR
         ALU_Result = ~(A_in | B_in);

      4'b0111: // Equal Camparison
         ALU_Result = (A_in == B_in)?32'd1:32'd0;

      default: ALU_Result = A_in + B_in;
   endcase
   end


endmodule
```

Simulation

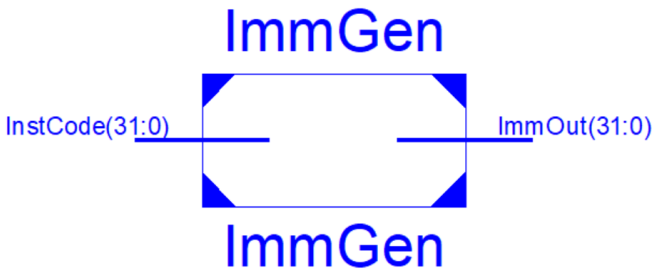


TestBench

```
module tb_alu;
   // Inputs
   reg [31:0] A_in;
   reg [31:0] B_in;
   reg [3:0] Alu_Sel;
   // Outputs
   wire Carry_Out;
   wire Overflow;
   wire Zero;
   wire [31:0] ALU_Out;

   integer i;

   // Instantiate the Unit Under Test (UUT)
   alu_32 uut (
      .A_in(A_in),
      .B_in(B_in),
      .Alu_Sel(Alu_Sel),
      .Carry_Out(Carry_Out),
      .Overflow(Overflow),
      .Zero(Zero),
      .ALU_Out(ALU_Out)
   );

   initial begin
      // Initialize Inputs
      A_in = 0;
      B_in = 0;
      Alu_Sel = 0;

      // Wait 100 ns for global reset to finish
      #100;
      A_in = 32'h0A;
      B_in = 32'h64;
      // Add stimulus here

      for(i=0; i<=7; i=i+1)
      begin
         Alu_Sel = Alu_Sel + 4'h01;
         #100;
      end
   end
endmodule
```

Immediate Generator

Schematic



Verilog Code

```
module ImmGen(InstCode, ImmOut);
input [31:0] InstCode;
output reg [31:0] ImmOut;

always @(InstCode)
begin
   case (InstCode[6:0])
   7'b0000_0011 : ImmOut = {InstCode[31] ? {20{1'b1}} : 20'b0, InstCode[31:20]};
   7'b0010_0011 : ImmOut = {InstCode[31] ? {20{1'b1}} : 20'b0, InstCode[31:20]};
   7'b0100_0011 : ImmOut = {InstCode[31] ? {20{1'b1}} : 20'b0, InstCode[31:25], InstCode[11:7]};
   7'b0001_0111 : ImmOut = {InstCode[31:12], 12'b0};
   default        ImmOut = {32'b0};
   endcase
end

endmodule
```