Yousef Jarrar & Nicholas Chiodini
CSE 401 – Dr Gomez
Lab 3

1 Introduction

The objective of this lab is to implement and test the Execution (EX) pipeline stage and integrate it with the IF and ID stages. There are 5 components to implement for this stage which are: ADDER, TOP_MUX, BOTTOM_MUX, ALU, ALU_CONTROL. We also need to implement a EX_MEM module that connects the wires from the IF/ID stage. Then we instantiate a I_EXECUTE module in the pipeline where it will connect with the other stages.

2 Interface

The ADDER module will take in two 32-bit addresses which will be added together to compute the branch target address.

Table 1: adder Inputs

Name	Function
add_in1	32-bit address
add_in2	32-bit address

Table 2: adder2 Output

Name	Function
add_out	The 32-bit address which has the branch target address

The TOP_MUX module takes in two 32-bit addresses and a select wire that will determine the result that will go through the ALU in which a 32-bit address will have that result.

`

Table 3: top_mux Inputs

Name	Function
a	The 32-bit address (source)
b	The 32-bit address (target)
alusrc	The 1-bit select wire

Table 4: top_mux Output

Name	Function
y	The 32-bit address that contains the result that goes to the ALU

The BOTTOM_MUX module works similarly to the TOP_MUX module, but the difference is that it takes in two 5-bit wires instead of a 32-bit wire.

Table 5: bottom_mux Inputs

Name	Function
a	The 5-bit wire that will pass in instr[20:16]
b	The 5-bit wire that will pass in instr[15:11]
sel	The 1-bit select wire that will determine to use either a or b

Table 6: bottom_mux Outputs

Name	Function
y	The 5-bit wire that will contain either a or b that goes through to the next stage (MEM)

The ALU module takes in two 32-bit wires that will be used to compute the result of the ALU depending what the 3 control bits that determines to perform the operations of addition, subtraction, and, or, and slt. If the result is 0, then we assign a 1 to make it true; otherwise, assign it 0 if it's a nonzero number.

Table 7: alu Inputs

Name	Function
a	The 32-bit wire that comes from the ID/EX latch
b	The 32-bit wire that comes from the ID_EX latch
control	The 4 control bits that determines what operation to use

`

Table 8: alu Outputs

Name	Function
result	The 32-bit register that contains the result of the ALU
zero	The wire that contains either 1 or 0 depending on the result

The ALU_CONTROL module takes in a 2-bit control wire that will determine if the opcode is valid or not. If it is valid, then it assigns a value to select depending on the bits of the funct field. If the control wire receives an invalid opcode, then the output of the alu will have 32 x's.

Table 9: alu_control Inputs

Name	Function
funct	The 6-bit parameter that determines which ALU operation to perform depending on the bits that is in the funct field
aluop	The 2-bit control wire that determines if the opcode is valid or not.

Table 10: alu_control Outputs

Name	Function
select	The 3-bit register that contains the control bits to determine which action the ALU will take.

The EX_MEM module takes in a 2-bit control wire that will go into the next stage, a 3-bit control wire that determines reading, writing, and branching from memory, a 1-bit control wire that contains a zero from the ALU, a 32-bit wire that has the result from the ALU, the
32-bit data that goes into the next stage that has not been implemented yet, and a 5-bit wire from the bottom_mux component. Note that the 3-bit control wire gets divided into 3 wires.

Table 11: ex_mem Inputs

Name	Function
ctlwb_out	The 2-bit control wire that controls
ctlm_out	The 3-bit control wire that determines reading, writing, and branching memory
adder_out	The 32-bit wire that carries the result from the adder module
aluzero	The 1-bit wire that carries the result of a 1 or 0 from the ALU_CONTROL module
aluout	The 32-bit wire that carries the result of the ALU
readdata2	The 32-bit wire that carries the data from the target

`

	register from the previous stage
muxout	The 5-bit wire that carries the result from the bottom_mux module

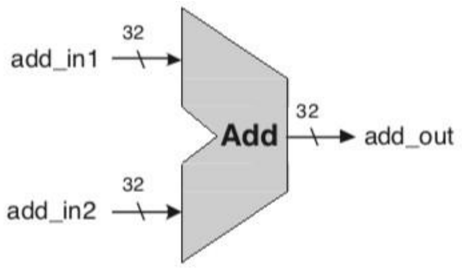
Table 12: ex_mem Outputs

Name	Function
wb_ctlout	The 4-bit register that contains the result from the ctlwb_out wire
branch	The 1-bit register that has the contents of ctlm_out[0]
memread	The 1-bit register that has the contents of ctlm_out[1]
memwrite	The 1-bit register that has the contents of ctlm_out[2]
add_result	The 32-bit register that has the result from adder_out
zero	The 1-bit register that contains whether the aluzero contains a 1 or 0
alu_result	The 32-bit register that contains the result from aluout
rdata2out	The 32-bit register that contains the contents of readdata2
five_bit_muxout	The 5-bit register that contains the result of muxout

3 Design

The design of the ADDER module two 32-bit addresses and outputs the result of those two 32-bit addresses.

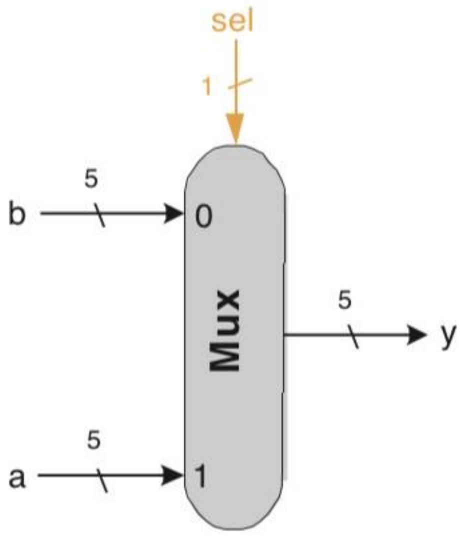
Figure 1: Adder design



The design of the MUX module takes in two 5-bit wires with a 1-bit select wire that determines to use either of the 5-bit wires.

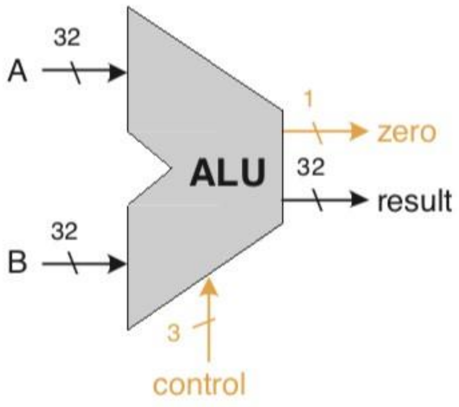
`

Figure 2: MUX design



The design of the ALU module takes in two 32-bit wires with a 3-bit control wire that controls what the result will be.

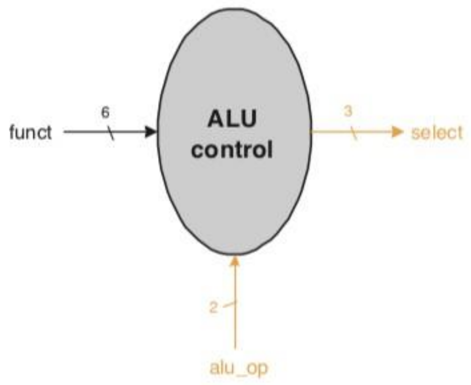
Figure 3: ALU design



The design for the ALU CONTROL module takes in a 6-bit wire and as well a 3-bit wire that determines what actions the ALU will take.

`

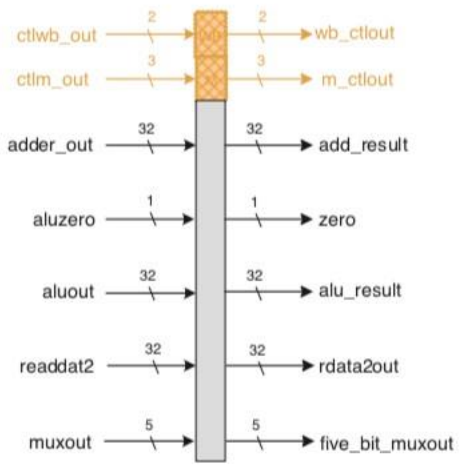
Figure 4: ALU CONTROL module design



The design of the EX_MEM module takes in a 5-bit control wire in which controls reading, writing, and branching memory, a 32-bit wire that carries the result of the adder, a 1-bit wire that has either a 1 or 0, a 32-bit wire that holds the result from the ALU, a
32-bit wire that contains the data from the previous stage, and a 5-bit wire that carries the result from the mux module.

`

Figure 5: EX MEM module design



4 Implementation

Listing 1: Implementation for adder

```
`timescale 1ns / 1ps

module adder (
     input wire [31:0] add_in1,
     input wire [31:0] add_in2,
     output wire [31:0] add_out
    );

    assign add_out = add_in1 + add_in2;


endmodule   // adder
```

Listing 2: Implementation for top_mux

```
`timescale 1ns / 1ps


module top_mux(
          output wire [31:0] y,
        input  wire [31:0] a, b, input
        wire alusrc
     );
```

`

```
     assign y = alusrc ? a : b;

endmodule   // top_mux
```

Listing 3: Implementation for bottom_mux

```
`timescale 1ns / 1ps

module bottom_mux(
     output wire [4:0] y,
     input wire [4:0] a, b,
     input wire sel
   );

      assign y = sel ? a : b;

endmodule // bottom_mux
```

Listing 4: Implementation for alu

```
`timescale 1ns / 1ps

module alu (
     input wire [31:0] a,
     input wire [31:0] b,
     input wire [2:0] control, output
     reg  [31:0]  result,  output  wire
     zero
   );

   parameter ALUadd = 3'b010,
                ALUsub   =  3'b110,
                ALUand   =  3'b000,
                ALUor    =  3'b001,
                ALUslt   =  3'b111;


    // Handles negative inputs wire
    sign_mismatch;
    assign sign_mistach = a[31] ^ b[31];

    initial
     result <= 0;

     always @ *
          case( control )
               ALUadd : result = a + b; ALUsub
               : result = a - b; ALUand :
               result = a & b; ALUor  : result
               = a || b;
               ALUslt : result = a < b ? 1 - sign_mismatch : 0 +
  sign_mismatch;
               default : result = 32'bX;
```

`

```
           endcase

           /* Check to see if result is equal to zero. If it is assign true
(1),
               false (0) otherwise, meaning it is a non-zero
number */
           assign zero = ( result == 0 ) ? 1 : 0;

endmodule // alu
```

Listing 5: Implementation for the alu_control module

```
`timescale 1ns / 1ps

module alu_control (
     input wire [5:0] funct,
     input wire [1:0] aluop,
     output reg [2:0] select
    );

    parameter   Rtype  =  2'b10;
                Radd   =  2'b10,
                Rsub   =  2'b10,
                Rand   =  2'b10,
                Ror    =  2'b10,
                Rslt   =  2'b10;

    parameter   lwsw   =  2'b00,       // Since LW and     SW use the    same
bit pattern, only way to store them as a parameter
                Itype = 2'b01,     // beq, branch
                xis   = 6'bXXXXXX;

    // ALU Control Inputs Designation
    parameter   ALUadd   =  3'b010,   // For R-type instructions
                ALUsub   =  3'b110,
                ALUand   =  3'b000,
                ALUor    =  3'b001,
                ALUslt   =  3'b111;

    // If the input information does not correspond to any valid instruction,
    // alu_op = 2'b11 which sets control = ALUx = 3'b011
    // and ALU output is 32 x's as required by lab manual parameter unknown
    = 2'b11,  // For invalid opcode, ALUop =
2'b11
                ALUx    = 3'b011;

    // Funct field
    parameter FUNCTadd      =  6'b100000,
                FUNCTsub    =  6'b100010,
                FUNCTand    =  6'b100100,
                FUNCTor     =  6'b100101,
```

`

```
                FUNCTslt = 6'b101010;

    initial
     select <= 0;

    always @ * begin
     if ( aluop == Rtype )
      begin
          case ( funct )
               // Assign the correct select value based on the
 function field
               // Use Fig 3.2 to aid you in this.
               // Desired ALU action
               FUNCTadd : select <= ALUadd;
               FUNCTsub : select <= ALUsub;
               FUNCTand : select <= ALUand;
               FUNCTor  : select <= ALUor;
               FUNCTslt : select <= ALUslt;
               default  : select <= ALUx;
          endcase
      end

      // For all other types. Use figure 3.2 to help you out.
      // Feel free to reuse any of the parameters defined above.
      else if ( aluop == lwsw )
      begin
          select <= ALUadd;
      end

      else if ( aluop == Itype )
      begin
          select <= ALUsub;
      end

      else if ( aluop == unknown )
      begin
          select <= ALUx;
      end

      else
      begin
          select <= select;
      end
    end
endmodule // alu_control
```

Listing 6: Implementation for the ex_mem module

```
 `timescale 1ns / 1ps

 module ex_mem(
      input wire [1:0] ctlwb_out,
      input wire [2:0] ctlm_out,
```

`

```
    input wire [31:0] adder_out, input
    wire aluzero,
    input wire [31:0] aluout, readdata2, input
    wire [4:0] muxout,
    output reg [1:0] wb_ctlout,
    output reg branch, memread, memwrite, output
    reg [31:0] add_result,
    output reg zero,
    output reg [31:0] alu_result, rdata2out, output
    reg [4:0] five_bit_muxout
);

initial begin
    wb_ctlout <= 0;
    branch <= 0; memread <= 0; memwrite <= 0;
    add_result <= 0;
    zero <= 0;
    alu_result <= 0; rdata2out <= 0;
    five_bit_muxout <= 0;
end

always @ * begin
    #1 // Update Delay

    // Use FIg 3.7 to assign the inputs to the outputs wb_ctlout
    <= ctlwb_out;
    branch <= ctlm_out[0];
    memread <= ctlm_out[1];
    memwrite <= ctlm_out[2];
    add_result <= adder_out; zero
    <= aluzero; alu_result <=
    aluout; rdata2out <=
    readdata2;
    five_bit_muxout <= muxout;
  end
endmodule // ex_mem
```

5 Test Bench Design

The ALU test bench is designed to show different ALUOp codes, and the result from those ALUOp codes. It will cycle through different control bits which determines what happens to the A and B wires.

`

The ALU CONTROL test bench is designed to show what ALUOp codes are being used and what function fields are being used in the process.

The MUX test bench is designed to show what wire is being selected depending if the wire is 1 or 0.

Listing 6: ALU test bench

```
`timescale 1ns / 1ps

module test ();

     // Register Declarations
     reg [31:0] a, b;
     reg [2:0] control;

     // Wire Ports
     wire [31:0] result;
     wire        zero;

     initial begin
          a <= 'b1010; b <=
          'b0111; control <=
          'b011;

          $display("A = %b\tB = %b", a, b);
          $monitor("ALUOP = %b\tresult = %b", control, result);

          #1
          control <= 'b100;

          #1
          control <= 'b010;

          #1
          control <= 'b111;

          #1
          control <= 'b110;

          #1
          control <= 'b001;

          #1
          control <= 'b000;
          #1
          $finish;
      end
```

`

```
             alu alu2 ( .a( a ), .b( b ), .control( control ), .result( result
), .zero( zero ) );

endmodule
```

Listing 7: ALU CONTROL Test Bench

```
  module test ();

       // Wire ports
       wire [2:0] select;

       // Register Declarations reg
       [1:0] aluop;
       reg [5:0] funct;

       alu_control alu_control2 ( .funct( funct ), .aluop( aluop ),
    .select( select ) );

       initial begin
            aluop = 2'b00;
            funct = 6'b100000;
            $monitor("ALUOp = %b\tfunct = %b\tselect = %b", aluop, funct, select);

            #1
            aluop  =  2'b01;
            funct  =  6'b100000;


            #1
            aluop  =  2'b10;
            funct  =  6'b100000;

            #1
            funct  =  6'b100010;
```

`

```
        #1
        funct  =  6'b100100;

        #1
        funct  =  6'b100101;

        #1
        funct  =  6'b101010;


        #1
        $finish;

    end

endmodule
```

Listing 8: MUX Test Bench

```
module test ();

   // Wire ports
   wire [4:0] y;

   // Register Declarations reg
   [4:0] a, b;
   reg sel;

   mux mux2( .y( y ), .a( a ), .b( b ), .sel( sel ) );

   initial begin
        a = 5'b01010;
        b = 5'b10101;
        sel = 1'b1;

        #10
        a = 5'b00000;

        #10
        sel = 1'b1;

        #10
        b = 5'b11111;

        #5
        a = 5'b00101;

        #5
```

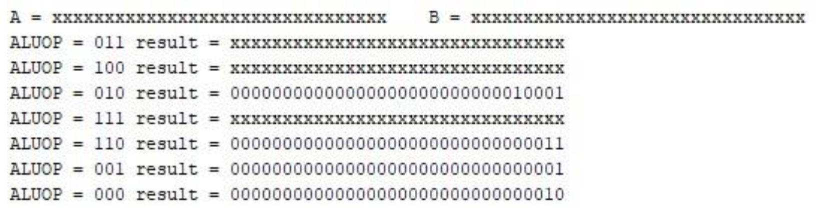
`

```
          sel = 1'b0;
          b = 5'b11101;
          #5
          sel = 1'bx;
      end


     always @ ( a or b or sel )
          #1
          $display("At t = %0d sel = %b A = %b B = %b Y = %b", $time,
sel, a, b, y);

endmodule
```

`

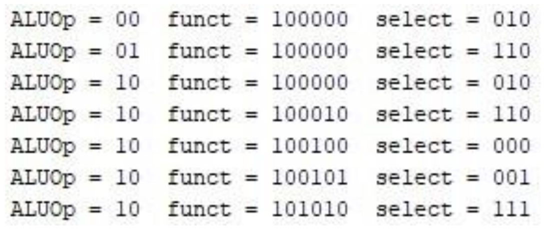
6 Simulation

Figure 6: ALU test results from Listing 6



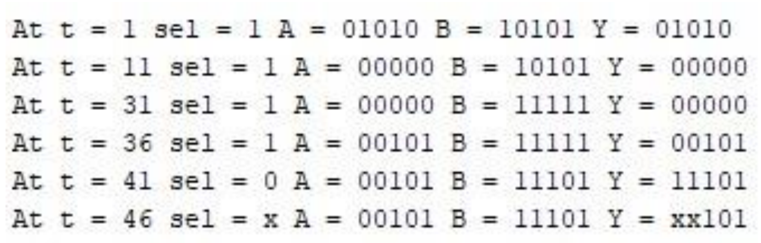
```
A = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx    B = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ALUOP = 011 result = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ALUOP = 100 result = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ALUOP = 010 result = 00000000000000000000000000010001
ALUOP = 111 result = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ALUOP = 110 result = 00000000000000000000000000000011
ALUOP = 001 result = 00000000000000000000000000000001
ALUOP = 000 result = 00000000000000000000000000000010
```

Figure 7: ALU CONTROL test results from Listing 7



```
ALUOp = 00  funct = 100000  select = 010
ALUOp = 01  funct = 100000  select = 110
ALUOp = 10  funct = 100000  select = 010
ALUOp = 10  funct = 100010  select = 110
ALUOp = 10  funct = 100100  select = 000
ALUOp = 10  funct = 100101  select = 001
ALUOp = 10  funct = 101010  select = 111
```

Figure 8: MUX test results from Listing 8



```
At t = 1 sel = 1 A = 01010 B = 10101 Y = 01010
At t = 11 sel = 1 A = 00000 B = 10101 Y = 00000
At t = 31 sel = 1 A = 00000 B = 11111 Y = 00000
At t = 36 sel = 1 A = 00101 B = 11111 Y = 00101
At t = 41 sel = 0 A = 00101 B = 11101 Y = 11101
At t = 46 sel = x A = 00101 B = 11101 Y = xx101
```

`