Lab 2: The ID Pipeline Stage

George Suarez and Patrick Duggin

October 31, 2018

1 Introduction

The objective of this lab is to implement and test the Instruction Decode (ID) pipeline stage and integrate it with the IF stage. There are five modules to implement which are: I_DECODE, CONTROL, REG, S_EXTEND, and ID_EX.

2 Interface

The CONTROL module receives the 6-bits of instruction code from the opcode field of IF_ID_instr and divides it into EX, M, and WB control lines.

Table 1: control Inputs

Name	Function
opcode	The 6-bits from the opcode field of IF_ID_instr

Table 2: control Outputs

Name	Function
EX	The 4-control bit
M	The 3-control bit
WB	The 2-control bit

The REGISTER module fills 32-bit registers/addresses with zero where a reg_write control line specifies a 1 for writing data and a 0 for not writing data. This module outputs two 32-bit registers where one of them holds the contents for the source registers and the contents for the target registers.

Table 3: register Inputs

Name	Function
rs	The 5-bit source register
rt	The 5-bit target register
rd	The 5-bit destination register
write_data	The 32-bit data
reg_write	The 1-bit that toggles between 1 or 0

Table 4: register Outputs

Name	Function
A	The 32-bit register that holds the result of REG[rs]
B	The 32-bit register that holds the result of REG[rd]

The combinational module S_EXTEND extends a 16-bit number into a 32-bit number and controls if the number is a positive or negative.

Table 5: s_extend Inputs

Name	Function
Next_end	The 16-bit immediate field of IF_ID_instr

Table 6: s_extend Outputs

Name	Function
extend	The 32-bit sign-extended value

The ID_EX module is the pipeline register that inputs and outputs the program counter and instruction.

Table 7: id_ex Inputs

Name	Function
ctlwb_out	The 2-control bit for write back (WB)
ctlm_out	The 3-control bit for memory (M)
ctlex_out	The 4-control bit for execute (EX)
npc	The 32-bit new program counter
readdata1	The 32-bit data from REG[rs]
readdata2	The 32-bit data from REG[rt]
signext_out	The 32-bit value from s_extend
instr_2016	The 5-bit instruction code from instr[20:16]

instr_1511	The 5-bit instruction code from instr[15:11]

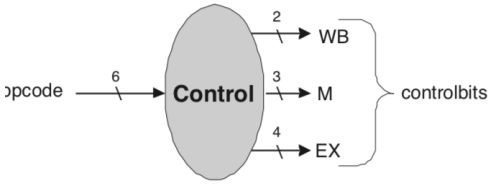
Table 8: id_ex Outputs

Name	Function
wb_ctlout	The 2-bit register that holds the result from ctlwb_out
wb_ctlm	The 3-bit register that holds the result from ctlm_out
wb_ctlex	The 4-bit register that holds the result from ctlex_out
regdst	The 1-bit register that holds the result of ctlex_out[3]
alusrc	The 1-bit register that holds the result of ctlex_out[0]
npcout	The 32-bit register that holds the result of npcout
rdata1out	The 32-bit register that holds the result of readdata1
rdata2out	The 32-bit register that holds the result of readdata2
s_extendout	The 32-bit register that holds the result of signext_out
instrout_2016	The 5-bit register that holds the result of instr_2016
instrout_1511	The 5-bit register that holds the result of instr_1511

3 Design

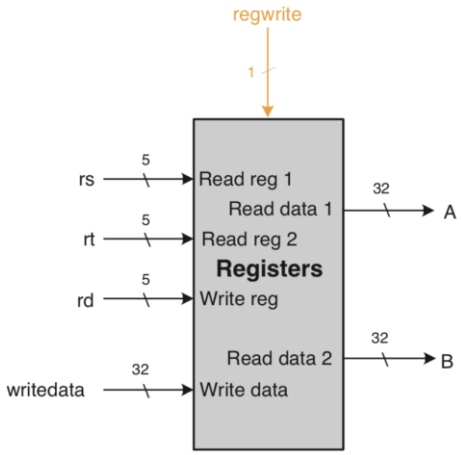
The design of the CONTROL module takes in a 6-bit opcode that outputs the control bits for WB, M, and EX.

Figure 1: Control design



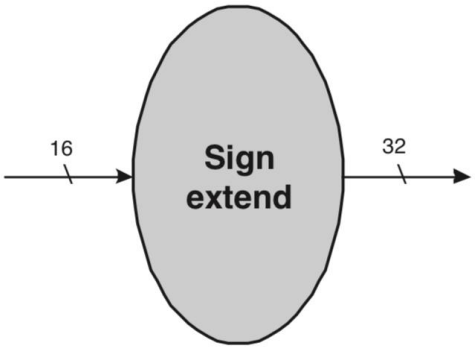
The design of the REG module takes in a 5-bit source register, a 5-bit target register, a 5-bit destination register, and a 32-bit data register that holds the data. Then it outputs two 32-bit registers that holds the contents of the source and target registers respectively.

Figure 2: REG design



The design of the S_EXTEND module takes in a 16-bit number and outputs the 32-bit version of that 16-bit number. Also maintains the characteristics of the number whether it is a positive or negative.

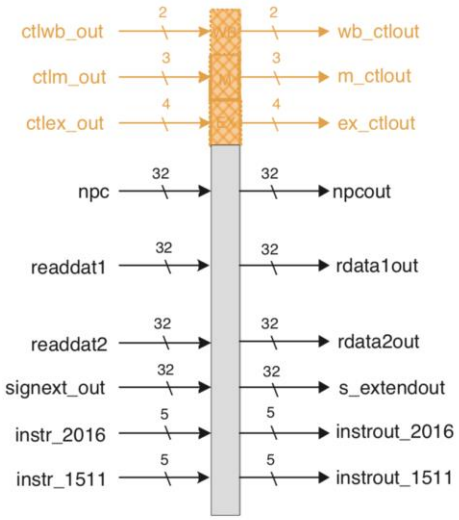
Figure 3: S_EXTEND design



The design for the ID_EX memory module has 9 control bits that is divided into 3 control wires that handles the write back, memory, and execution. It takes in a 32-bit NPC, two 32-bit read data registers, 32-bit sign extender, and two 5-bit instruction code

registers. Then it outputs the program counter and instructions.

Figure 4: ID_EX module design



4 Implementation

Listing 1: Implementation for control

```
`timescale 1ns / 1ps

module control (
    input wire [5:0] opcode,
    output reg [3:0] EX,
    output reg [2:0] M,
    output reg [1:0] WB
 );

 parameter RTYPE = 6'b000000;
 parameter LW = 6'b100011;
 parameter SW = 6'b101011;
 parameter BEQ = 6'b000100;
 parameter NOP = 6'b100000;

 initial begin
 /* Assign decimal representation of 0 to our output
    REG's here. Note the difference
```

```
 */
    EX <= 0;
    M <= 0;
    WB <= 0;
 end

 /* Assign the don't cares (X) to high impedance (Z)...
    For design correctness and more proper MIPS emulation
 */

 always @ * begin
    case (opcode)
        RTYPE: begin
            EX <= 4'b1100; /* Note use of non-blocking operator
(<=) vs blocking operator (=) */
            M <= 3'b000;
            WB <= 2'b10;
        end
        /* Assign the remaining values according to the chart
in Lab Manuel.
           Either parametrize it, or hardcode at as is done for
RTYPE.
        */
        LW: begin
            EX <= 4'b0001;
            M <= 3'b010;
            WB <= 2'b11;
        end

        SW: begin
            EX <= 4'b0001;
            M <= 3'bz01;
            WB <= 2'b0z;
        end

        BEQ: begin
            EX <= 4'bz010;
            M <= 3'b100;
            WB <= 2'b0z;
        end

        NOP: begin
            EX <= 4'b0000;
            M <= 3'b000;
            WB <= 2'b00;
        end

        default: $display ("Opcode not recognized.");
     endcase
    end
 endmodule // control
```

Listing 2: Implementation for register

```
`timescale 1ns / 1ps

module register (
    input [4:0] rs,
    input [4:0] rt,
    input [4:0] rd,
    input [31:0] write_data,
    input reg_write,
    output reg [31:0] A,    // rs output
    output reg [31:0] B     // rd output
  );

  // Register declaration
  reg [31:0] REG [0:31]; // Gives us 32 registers, each 32 bits
long

  integer i;

  initial begin
    A <= 0;
    B <= 0;

    // Initialize our registers
    for (i = 0; i < 32; i = i + 1)
        REG[i] <= 0;

        // Display contents of the first 9 registers
        $display("From Register Memory:");
        for (i = 0; i < 9; i = i + 1)
            $display("\tREG[%0d] = %0d", i, REG[i]);

            // Display last register
            $display("\t...");
            $display("\tREG[%0d] = %0d", 31, REG[31]);
  end

  always @ * begin
    A <= REG[rs];
    B <= REG[rt];

    // Write data using index rd
    if (rd != 0 && reg_write)
        REG[rd] <= write_data;
    end

endmodule // register
```

Listing 3: Implementation for s_extend

```
`timescale 1ns / 1ps

module s_extend(
    input wire [15:0] next_end,
    output reg [31:0] extend
  );

  always @ * begin
    // Replicate signed bit 16 times then cancatinate
    extend = { {16{next_end[15]}}, next_end };
  end

endmodule // s_extend
```

Listing 4: Implementation for id_ex

```
`timescale 1ns / 1ps

module id_ex(
      input wire [1:0]  ctlwb_out,
      input wire [2:0]  ctlm_out,
      input wire [3:0]  ctlex_out,
      input wire [31:0] npc, readdata1, readdata2, signext_out,
      input wire [4:0]  instr_2016, instr_1511,
      output reg [1:0]  wb_ctlout,
      output reg [2:0]  m_ctlout,
      output reg  regdst, alusrc,
      output reg  [1:0] aluop,
      output reg  [31:0] npcout, rdata1out, rdata2out,
s_extendout,
      output reg [4:0] instrout_2016, instrout_1511
 );

initial begin
      // Assign 0's to everything
      wb_ctlout <= 0;
      m_ctlout <= 0;
      regdst <= 0;
      aluop <= 0;
      alusrc <= 0;
      npcout <= 0;
      rdata1out <= 0;
      rdata2out <= 0;
      s_extendout <= 0;
      instrout_2016 <= 0;
      instrout_1511<= 0;
end

always @ * begin
     // Wire the inputs to the outputs corresponding outputs
             #1
```

```
            wb_ctlout <= ctlwb_out;
            m_ctlout <= ctlm_out;
            regdst <= ctlex_out[3];
            aluop <= ctlex_out[2:1];
            alusrc <= ctlex_out[0];
            npcout <= npc;
            rdata1out <= readdata1;
            rdata2out <= readdata2;
            s_extendout <= signext_out;
            instrout_2016 <= instr_2016;
            instrout_1511    <= instr_1511;
      end
endmodule // id_ex
```

Listing 5: Implementation for the top-level I_DECODE module

```
`timescale 1ns / 1ps

module I_DECODE(
    input wire [31:0] IF_ID_instrout,
    input wire [31:0] IF_ID_npcout,
    input wire [4:0]  MEM_WB_rd,
    input wire MEM_WB_regwrite,
    input wire [31:0] WB_mux5_writedata,
    output wire [1:0] wb_ctlout,
    output wire [2:0] m_ctlout,
    output wire regdst, alusrc,
    output wire [1:0] aluop,
    output wire [31:0] npcout, rdata1out, rdata2out,
s_extendout,
    output wire [4:0] instrout_2016, instrout_1511
   );

    // Signals
    wire [3:0] ctlex_out;
    wire [2:0] ctlm_out;
    wire [1:0] ctlwb_out;
    wire [31:0] readdata1, readdata2, signext_out;

    // instantiations
    control control2(
        .opcode( IF_ID_instrout[31:26] ), // input
        .EX( ctlex_out ),         // outputs
        .M( ctlm_out ),
        .WB( ctlwb_out )
      );

    register register2(
        .rs( IF_ID_instrout[25:21] ), // inputs
        .rt( IF_ID_instrout[20:16] ),
        .rd( MEM_WB_rd ),
        .writedata( WB_mux5_writedata ),
```

```
        .regwrite( MEM_WB_regwrite ),
        .A(readdata1),           //outputs
        .B(readdata2)
      );

    s_extend s_extend2(
        .nextend( IF_ID_instrout[15:0] ),
        .extend( signext_out )
      );

    id_ex id_ex2(
        .ctlwb_out( ctlwb_out ),  // 9 inputs
        .ctlm_out( ctlm_out ),
        .ctlex_out( ctlex_out ),
        .npc( IF_ID_npcout ),
        .readdat1( readdata1 ),
        .readdat2( readdata2 ),
        .signext_out( signext_out ),
        .instr_2016( IF_ID_instrout[20:16] ),
        .instr_1511( IF_ID_instrout[15:11] ),
        .wb_ctlout( wb_ctlout ),  // outputs
        .m_ctlout( m_ctlout ),
        .regdst( regdst ),
        .alusrc( alusrc ),
        .aluop( aluop ),
        .npcout( npcout ),
        .rdata1out( rdata1out ),
        .rdata2out( rdata2out ),
        .s_extendout( s_extendout ),
        .instrout_2016( instrout_2016 ),
        .instrout_1511( instrout_1511 )
      );

endmodule // I_DECODE
```

5 Test Bench Design

The CONTROL test bench is designed to display the 6-bit opcode from the instructions code register and output what control bits are being used from the given instructions. Some opcodes should not be recognized if the control bits do not match any of the instructions code (see Listing 6 for the Verilog code).

The REG test bench is designed to assign source, target, and destination registers with random data with toggling the reg_write wire to either 1 for writing data or 0 for not writing data. Then the timing diagram will show a 5*ns* simulation that shows the contents of the registers (see Listing 7 for the Verilog code).

The S_EXTEND test bench is designed to display two 16-bit numbers with one having 1 has the most significant bit and the other having 1 as the least significant bit. Then it displays the extended 32-bit version of those two numbers (see Listing 8 for the Verilog code).

Listing 6: Control Test Bench

```
module test ();
    // Port wires
    wire [08:0] controls;

    // Register Declarations
    reg [31:0] instr;

    initial begin
        instr[31:26] <= 6'b0;
        $display("Time\t\tOPCODE\t\t\t\t\Controlbits\n");
        $monitor("%0d\t\t%b\t%b ", $time, instr, controls);
        #20; $finish;
    end

    always begin
        #1 instr[31:26] = 35;
        #1 instr[31:26] = 43;
        #1 instr[31:26] = 4;
        #1 instr[31:26] = 100;
        #1 instr[31:26] = 0;
     end

     control control1 ( instr[31:26], controls );
endmodule
```

Listing 7: Register Test Bench

```
`timescale 1ns / 1ps

module test;

    // Outputs
    wire [31:0] A, B;

    // Inputs
    reg [4:0] rs, rt, rd;
    reg [31:0] write_data;
    reg reg_write;

    register uut( .rs( rs ),
                  .rt( rt ),
                  .rd( rd ),
                  .write_data( write_data ),
                  .reg_write( reg_write ),
                  .A( A ),
                  .B( B ) );

    initial
    begin
        rs = 0;
        rt = 1;
        reg_write = 0;

        #1
        rs = 2;
        rt = 3;
        rd = 3;
        write_data = 100;

        #1
        rs = 4;
        rt = 5;
        reg_write = 1;

        #1
        reg_write = 0;
        rt = 3;

        #1
        rs = 6;
        reg_write = 1;
        rd = 6;
        write_data = 100;

        #1 $finish;
    end
endmodule
```

Listing 8: Sign Extend Test Bench

```
module test();
    // Port Wires
    wire [31:0] se_out;

    // Register Declarations
    reg [15:0] se_in;

    initial begin
        se_in = 0;
        #1 se_in = 16'b10;
        #1 se_in = 16'b01;
    end

    initial begin
        $monitor("Time = %0d\tse_in = %b\tse_out = %b", $time, se_in,
se_out);
    end

    s_extend s_extend1( .next_end( se_in ), .extend( se_out ) );

endmodule
```

6 Simulation

Figure 5: Control test results from Listing 6

```
Time        OPCODE              Controlbits

0       000000xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz1100
1       100011xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz0001
2       101011xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz0001
3       000100xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzzz010
Opcode not recognized.
4       100100xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzzz010
5       000000xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz1100
6       100011xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz0001
7       101011xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz0001
8       000100xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzzz010
Opcode not recognized.
9       100100xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzzz010
10      000000xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz1100
11      100011xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz0001
12      101011xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz0001
13      000100xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzzz010
Opcode not recognized.
14      100100xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzzz010
15      000000xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz1100
16      100011xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz0001
17      101011xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzz0001
18      000100xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzzz010
Opcode not recognized.
19      100100xxxxxxxxxxxxxxxxxxxxxxxxxx    zzzzzz010
```

Figure 6: Register Timing diagram from 0ns – 5ns

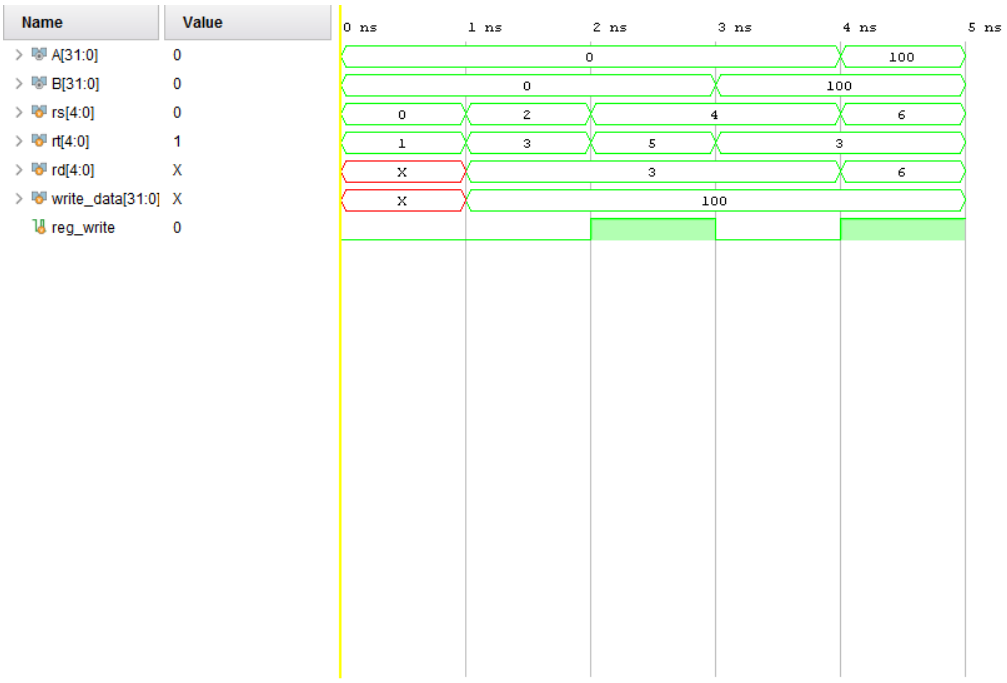


Figure 7: Sign extend test results from Listing 8

```
Time = 0     se_in = 0000000000000000     se_out = 00000000000000000000000000000000
Time = 1     se_in = 0000000000000010     se_out = 00000000000000000000000000000010
Time = 2     se_in = 0000000000000001     se_out = 00000000000000000000000000000001
```

7 Conclusions

The instruction decode stage for this lab was successfully implemented. The main thing we learned in doing this lab is how to run individual test benches in Vivado, and how to connect the instruction fetch stage with the instruction decode stage together. What we would have done differently in this lab is to implement a module, and then test that module before starting on the next module.