Lab 4: The MEM Pipeline Stage

George Suarez and Patrick Duggin

November 07, 2018

1 Introduction

The objective of this lab is to implement and test the Memory (MEM) pipeline and stage and integrate it with the IF, ID, and EX stages. There are four modules to implement in order to complete this stage which are: AND, D_MEM, MEM_WB, and MEMORY.

2 Interface

The AND module receives two inputs which are 1-bit each, and outputs a 1-bit result.

Table 1: AND Inputs

Name	Function
membranch	The 1-bit wire that determines which branch of memory to go to
zero	The 1-bit wire that carries either a 1 for a zero or 0 for a non-zero

Table 2: AND Output

Name	Function
PCSrc	The 1-bit wire that carries the result of the and operation of the two inputs

The D_MEM module receives 2 32-bit wires, and 2 1-bit wires. Then it outputs a 32-bit wire that will contain the memory address's contents.

Table 3: data_memory Inputs

Name	Function
addr	The 32-bit wire that contains the memory address
write_data	The 32-bit wire that contains the contents of the memory address
mem_write	The 1-bit wire that controls writing to memory
mem_read	The 1-bit wire that controls reading to memory

Table 4: data_memory Outputs

Name	Function
read_data	The 32-bit register that holds the contents of the memory address

The MEM_WB module receives a 2-bit control wire, two 32-bit wires, and a 5-bit wire. The outputs are two 1-bit registers, two 32-bit registers, and a 5-bit register.

Table 5: mem_wb Inputs

Name	Function
control_wb_in	The 2-bit control wire that controls write-backs of the memory
read_data_in	The 32-bit wire that contains the data that is going to be read
alu_result_in	The 32-bit wire that contains the result of the ALU from the EX stage
write_reg	The 5-bit wire that will contain the result of the five-bit-muxout result from the EX stage

Table 6: mem_wb Outputs

Name	Function
regwrite	The 1-bit register that holds the value of control_wb_in[0]
memtoreg	The 1-bit register that holds the value of control_wb_in[1]
read_data	The 32-bit register that holds the data that is passed in by read_data_in
mem_alu_result	The 32-bit register that holds the result of alu_result_in
mem_write_reg	The 5-bit register that holds the result of write_reg

The MEMORY module will connect all the modules that were implemented in this lab. It will receive a 2-bit control wire, four 1-bit wires, two 32-bit wires, and a 5-bit wire. The outputs are three 1-bit wires, two 32-bit wires, and a 5-bit wire.

Table 7: MEMORY Inputs

Name	Function
wb_ctlout	The 2-bit control wire that controls write-back
Branch	The 1-bit wire that determines which branch to take
Memread	The 1-bit wire that handles reading memory
memwrite	The 1-bit wire that handles writing memory
zero	The 1-bit wire carries a 1 for a zero or 0 for a nonzero
alu_result	The 32-bit wire that contains the alu_result from the mem_wb module
rdata2out	The 32-bit wire that contains the data that is going to be read
five_bit_muxout	The 5-bit wire that contains the result of the five_bit_mux module from the EX stage

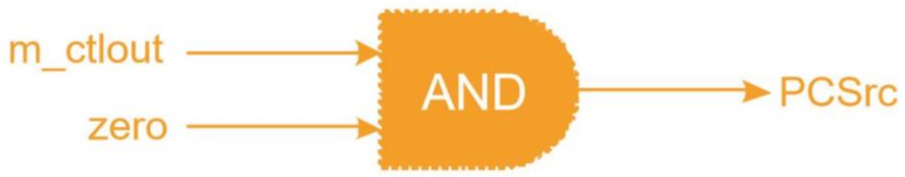
Table 8: MEMORY Outputs

Name	Function
MEM_PCSrc	The 1-bit wire that holds the PC value of the memory address
MEM_WB_regwrite	The 1-bit wire that holds the value of whether to write to a register (Store)
MEM_WB_memtoreg	The 1-bit wire holds the value to either store the memory into the register (Load)
read_data	The 32-bit wire that holds the contents of the memory address
mem_alu_result	The 32-bit wire that holds the result of alu_result
mem_write_reg	The 5-bit wire that holds the result of the five_bit_muxout

3 Design

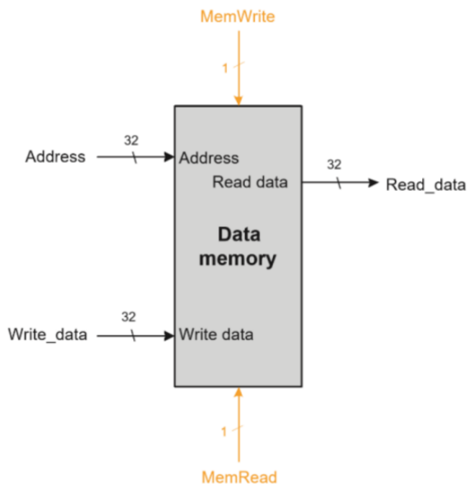
The design of the AND module takes in 2 1-bit wire, and outputs a 1-bit wire.

Figure 1: AND design



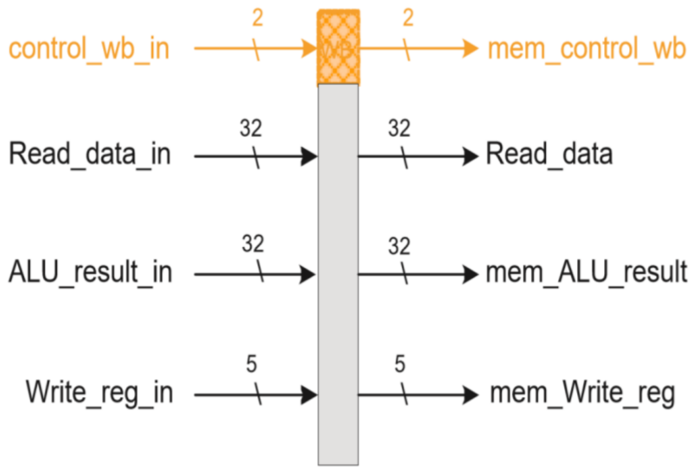
The design of the D_MEM module takes in two 1-bit control wires and two 32-bit wires and outputs a 32-bit wire.

Figure 2: D_MEM design



The design of the MEM_WB module takes in a 2-bit control wire, two 32-bit wires, and a 5-bit wire. Then it outputs the 2-bit register that holds the value of the 2-bit control wire, two 32-bit registers, and a 5-bit register.

Figure 3: MEM_WB design



4 Implementation

Listing 1: Implementation for AND

```
`timescale 1ns / 1ps

module AND(
    input wire membranch, zero,
    output wire PCSrc
   );

   assign PCSrc = membranch & zero;

endmodule
```

Listing 2: Implementation for D_MEM

```
`timescale 1ns / 1ps

module data_memory(
    input wire [31:0] addr, // Memory address
    input wire [31:0] write_data,  // Memory address contents
    input wire        memwrite, memread,
    output reg [31:0] read_data  // Output of memory address
contents
   );

   // Register Declaration
   reg [31:0] DMEM[0:255];  // 256 words of 32-bit memory

   integer i;

   initial begin
        read_data <= 0;

        // Initialize DMEM[0-5] from data.txt
        $readmemb("data.txt", DMEM);

        // Initialize DMEM[6-255] to 6-255
        for(i = 6; i < 256; i = i + 1)
            DMEM[i] = i;

        // Display DMEM[0-5]
        $display("From Data Memory (data.txt): ");
        for(i = 6; i < 10; i = i + 1)
            $display("\tDMEM[%0d] = %0d", i, DMEM[i]);

        // Display DMEM[255]
        $display("\t...");
        $display("\tDMEM[%0d] = %0d", 255, DMEM[255]);
   end

   always @ (addr) begin
        if (memwrite)   // Store
        begin
            DMEM[addr] <= write_data;
        end

        if (memread)    // Load
        begin
            read_data <= DMEM[addr];
        end
   end

endmodule  // data_memory
```

Listing 3: Implementation for MEM_WB

```
`timescale 1ns / 1ps

module mem_wb(
    input wire [1:0] control_wb_in,
    input wire [31:0] read_data_in, alu_result_in,
    input wire [4:0] write_reg_in,
    output reg regwrite, memtoreg,
    output reg [31:0] read_data, mem_alu_result,
    output reg [4:0] mem_write_reg
  );

  initial begin
        regwrite <= 0;
        memtoreg <= 0;
        read_data <= 0;
        mem_alu_result <= 0;
        mem_write_reg <= 0;
  end

  always @ * begin
        #1
        regwrite <= control_wb_in[1];
        memtoreg <= control_wb_in[0];
        read_data <= read_data_in;
        mem_alu_result <= alu_result_in;
        mem_write_reg <= write_reg_in;
  end

endmodule  // mem_wb
```

Listing 4: Implementation for MEMORY

```
`timescale 1ns / 1ps

module MEMORY(
    input wire [1:0] wb_ctlout,
    input wire branch, memread, memwrite, zero,
    input wire [31:0] alu_result, rdata2out,
    input wire [4:0] five_bit_muxout,
    output wire MEM_PCSrc,
    output wire MEM_WB_regwrite, MEM_WB_memtoreg,
    output wire [31:0] read_data, mem_alu_result,
    output wire [4:0] mem_write_reg
  );

  // Signals
  wire [31:0] read_data_in;

  // Instantiations
  AND AND_4(
             .membranch( branch ),
             .zero( zero ),
             .PCSrc( MEM_PCSrc )
           );

  data_memory data_memory4(
    .addr( alu_result ),
    .write_data( rdata2out ),
    .memwrite( memwrite ),
    .memread( memread ),
     .read_data( read_data_in )
    );

    mem_wb mem_wb4(
     .control_wb_in( wb_ctlout ),
     .read_data_in( read_data_in ),
     .alu_result_in( alu_result ),
     .write_reg_in( five_bit_muxout ),
     .regwrite( MEM_WB_regwrite ),
     .memtoreg( MEM_WB_memtoreg ),
     .read_data( read_data ),
     .mem_alu_result( mem_alu_result ),
     .mem_write_reg( mem_write_reg )
    );

 endmodule  // MEMORY
```

5 Test Bench Design

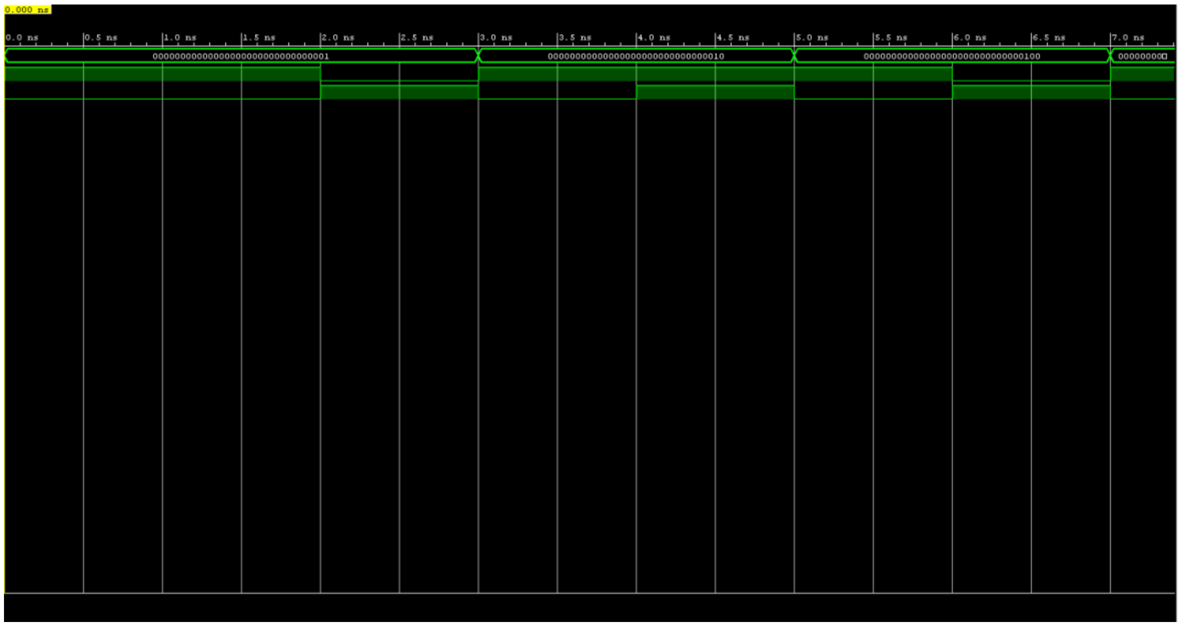
This test bench is designed to control reading and writing to memory. Each cycle will have different configurations with an assigned address in which if that address is being either read or written to memory.

Listing 5: MEM Test Bench

```
`timescale 1ns / 1ps

module mem_test;

    wire [31:0] read_data;

    reg [31:0] address;
    reg [31:0] write_data;
    reg mem_read, mem_write;

    initial begin
        mem_read = 1;
        mem_write = 0;
        address = 32'b00000001;

        #1
        mem_read = 1;
        mem_write = 0;
        address = 32'b00000001;

        #1
        mem_read = 0;
        mem_write = 1;
        address = 32'b00000001;
        write_data = ~address;

        #1
        mem_read = 1;
        mem_write = 0;
        address = 32'b00000010;

        #1
        mem_read = 1;
        mem_write = 1;
        address = 32'b00000010;
        write_data = ~address;

        #1
        mem_read = 1;
        mem_write = 0;
        address = 32'b00000100;

        #1
        mem_read = 0;
        mem_write = 1;
        address = 32'b00000100;
        write_data = ~address;

        #1
        mem_read = 1;
        mem_write = 0;
        address = 32'b00001000;

        #1
        mem_read = 1;
        mem_write = 1;
        address = 32'b00001000;
        write_data = ~address;

        #1 $finish;
    end

    data_memory data_memory4(
    .addr(address),
    .write_data(write_data),
    .memwrite(mem_write),
    .memread(mem_read),
    .read_data(read_data)
   );

endmodule
```

6 Simulation

Figure 4: Timing Diagram Of The MEM test



7 Conclusions

The memory pipeline stage has been successfully implemented. There is nothing that we would have changed in this lab. The main thing that we learned in doing this lab is how the memory pipeline was simple to implement because of the amount of components that needed to be implemented was low compared to the previous stages.