Lab 4: The MEM Pipeline Stage

George Suarez and Patrick Duggin

November 07, 2018

# 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.

# 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 |

# Design

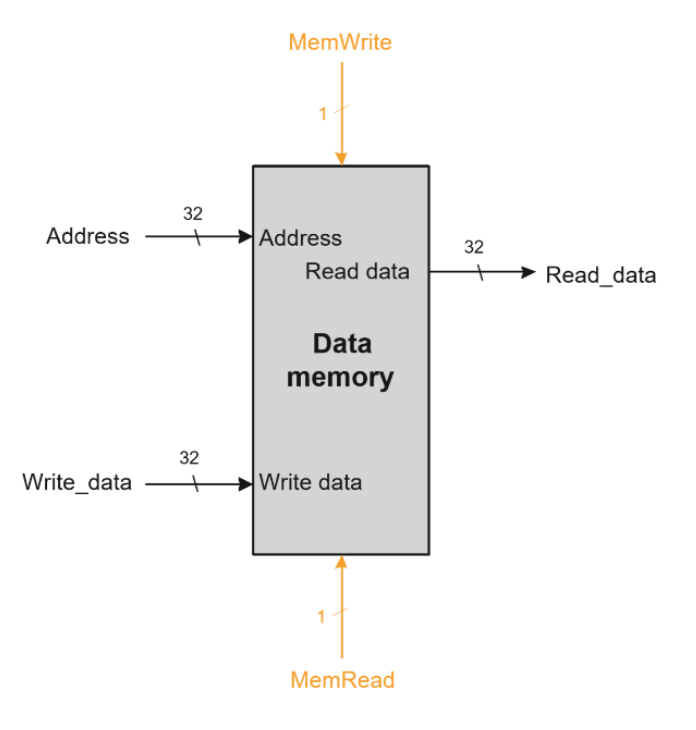
The design of the AND module takes in 2 1-bit wire, and outputs a 1-bit wire.

Figure 1: AND design



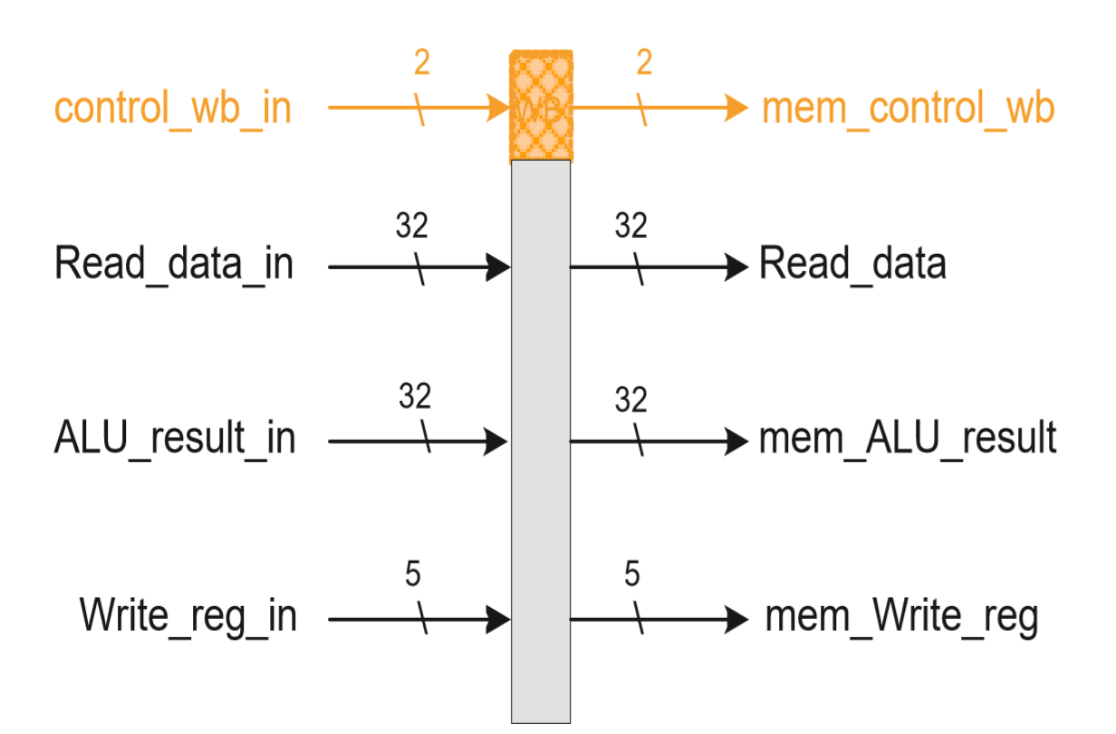
The design of the D\_MEM module takes in two 1-bit control wires and two 32-bit wires and outputs a 32-bit wire.

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



# 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 |
|  |

# Test Bench Design

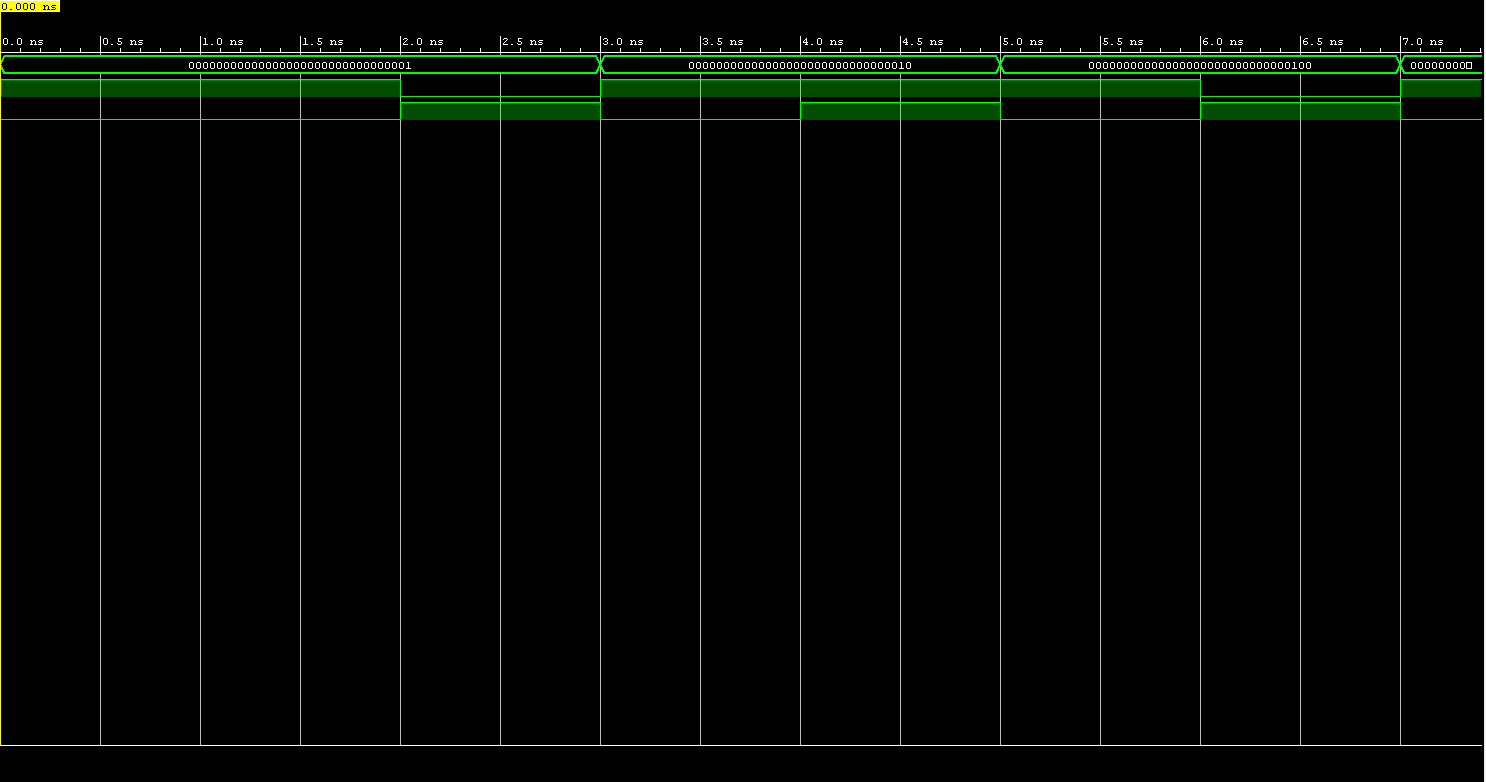
This test bench is designed to control reading and writing to memory. Each cycle will have different configurations with an assigned address in which if that address is being either read or written to memory.

Listing 6: 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 |

# Simulation

Figure 5: Timing Diagram Of The MEM test



# 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.