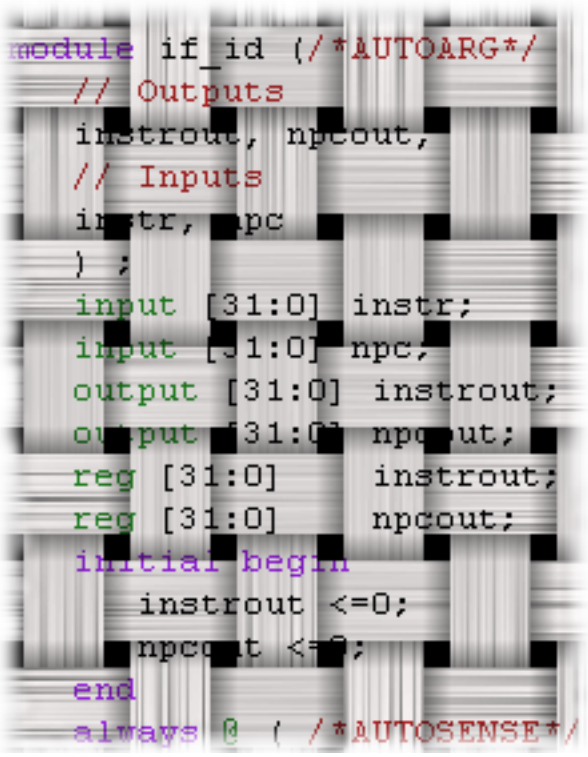
Computer Architecture
Implementing a Datapath in Verilog
A Lab Manual



George M. Georgiou and Scott McWilliams

Computer Science Department
California State University, San Bernardino

October 2003
Revision: 1.3, May 3, 2010

Contents

List of Code Listings

List of Figures

Part I

Lab Manual

LAB 1

The MIPS datapath in Verilog: The IF stage

Objective: To implement and test the Instruction Fetch (IF) pipeline stage of the MIPS five stage pipeline.

The series of labs in this manual has ultimate objective to implement and simulate in Verilog the MIPS pipeline datapath Figure 6.30 in Paterson and Hennessy's textbook [4]. The model will be structural (as opposed to behavioral), but with one exception: basic units, such as multiplexors and ALU's, may implemented as behavioral models. This approach reinforces the object-oriented style of programming, while at the same time relieving from the burden of structurally defining the basic units, which can be quite tedious, time consuming, and beyond the scope of this lab series.

The slightly revised MIPS datapath to be implemented is in figure 1.1 on page Lab 1–2.

For this week, you will implement the IF stage and test the fetching of instructions from memory. The IF stage isolated from the rest of the datapath can be seen in figure 1.2 on page Lab 1–2.

- The names of the pipeline registers are IF_ID, ID_EX, EX_MEM, MEM_WB. For now, you will need only IF_ID and EX_MEM.
- The instruction memory has 128 32-bit words. Later it will be expanded. All instructions and the PC are 32-bit wide. (Simply the 7 least significant bits (2^7 = 128) are used for the time being.)
- Implement the instruction memory, 2x1 MUX, and Incrementer-by-4 as separate modules. For the time being consider that the 1-bit signal PCSrc comes from a 1-bit register, PC_choose.
- Initialize IF_ID_IR (The instruction field of IF/ID) to 32 zeros.
- Initialize IF_ID_NPC to 32 zeros. Initialize PC_choose and EX_MEM_NPC to zeros. They will not change during this simulation. Initialize the first 10 words of memory (with addresses 0, 4, 8, etc.) with the following HEX values:

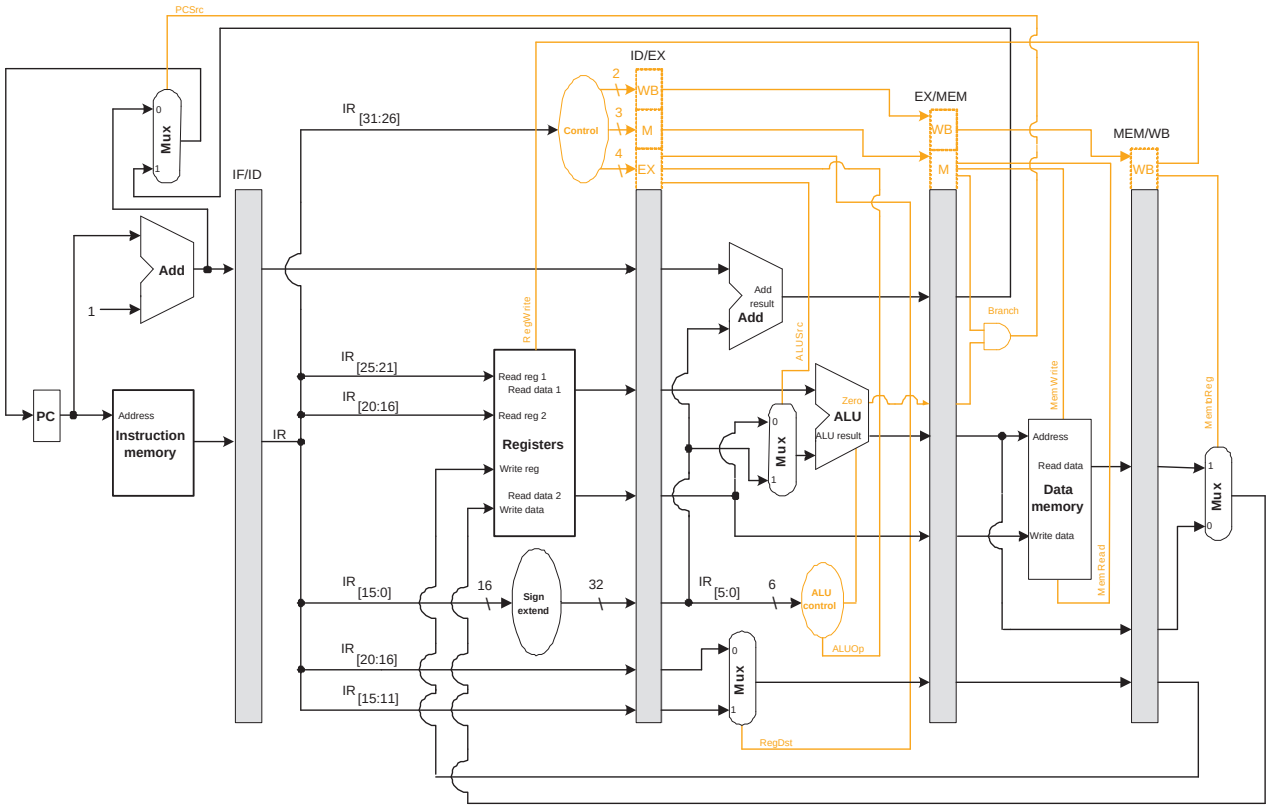


Figure 1.1: The revised MIPS datapath

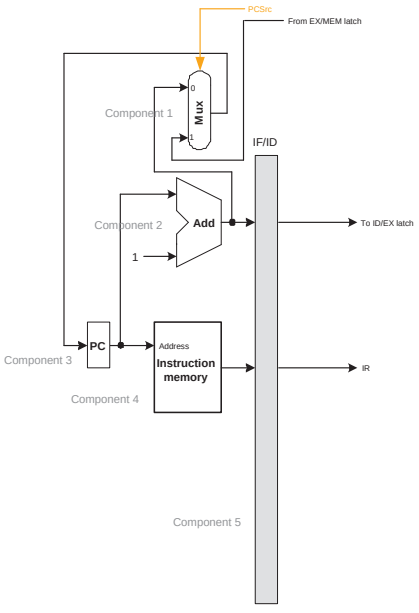


Figure 1.2: The IF stage

```
module mux (a,b,sel,y);
  input [31:0] a, b;
  input  sel;
  output [31:0] y;
   assign       y = sel ? a : b;
endmodule
```

Listing 1.1: Verilog code for the multiplexer.

```
A00000AA
10000011
20000022
30000033
40000044
50000055
60000066
70000077
80000088
90000099
```

- Turn in the source code and the printout of the clock cycle number, the contents of the PC (in decimal), IF_ID_IR (in hex), and IF_ID_NPC (in decimal) for 10 cycles of simulation. Be ready to demonstrate.

Note: The code in listing 1.1 implements the multiplexer in the IF stage as a combinational circuit.

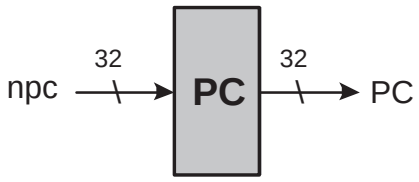


Figure 1.3: The program counter (PC)

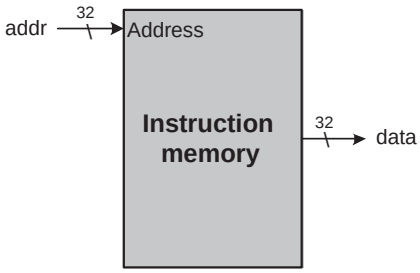


Figure 1.4: The instruction memory

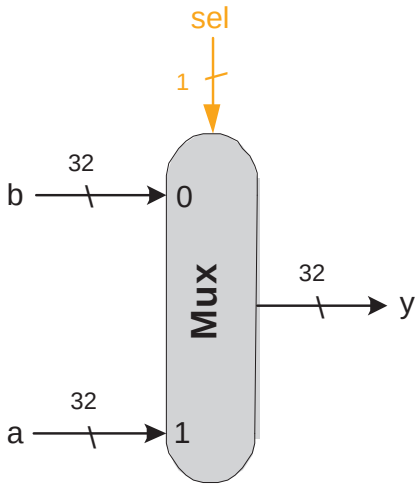


Figure 1.5: The multiplexer

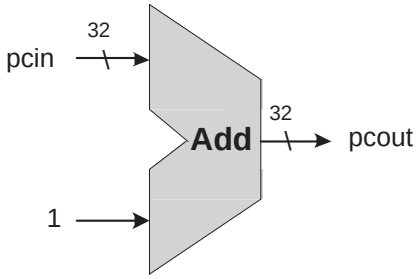


Figure 1.6: The incrementer by 1

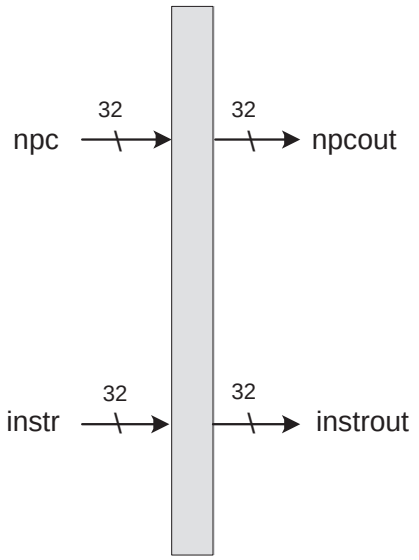


Figure 1.7: The IF/ID pipeline register (latch)

1.1 Testbenches

Testbenches help us verify that the design is correct. In this subsection we show two testbenches: One in listing 1.2 on page Lab 1–6 for the multiplexer of figure 1.5 on page Lab 1–4 and one in listing 1.3 on page Lab 1–7 for the incrementer of figure 1.6 on page Lab 1–4 . The results of the running the testbench are in figure 1.8 and in figure 1.9, respectively. In the latter, and in other testbench runs in the labs that follow, the standard messages of the runs will be largely omitted.

```
Beginning Compile
Beginning Phase I
Compiling source file: muxtest.v
Compiling included source file 'mux.v'
Continuing compilation of source file 'muxtest.v'
Finished Phase I
Entering Phase II...
Finished Phase II
Entering Phase III...
Finished Phase III
Highest level modules:   test_mux
Compile Complete
.
Running...
At t = 11 sel = 1 A = 00000000 B = 55555555 Y = 00000000
At t = 31 sel = 1 A = 00000000 B = ffffffff Y = 00000000
At t = 36 sel = 1 A = a5a5a5a5 B = ffffffff Y = a5a5a5a5
At t = 41 sel = 0 A = a5a5a5a5 B = dddddddd Y = dddddddd
At t = 46 sel = x A = a5a5a5a5 B = dddddddd Y = XXXXXXXX
0 Errors, 0 Warnings
Compile time = 0.00000, Load time = 0.00000, Execution time = 0.00000

Normal exit
```

Figure 1.8: The output when running the testbench for the multiplexer (listing 1.2 on page Lab 1–6)

```
Running...
Time = 11 A=3 IncrOut=4
Time = 21 A=15 IncrOut=16
Time = 31 A=64 IncrOut=65
```

Figure 1.9: The output when running the testbench for the incrementer (listing 1.3 on page Lab 1–7)

```
// Filename         : test_mux.v
// Description      : Testing the 32bit_mux module
//                    of the IF stage of the pipeline.

`include "mux.v"
module test_mux;

// Wire Ports
   wire [31:0] Y;

// Register Declarations
   reg  [31:0] A, B;
   reg         sel;

   MUX mux1 (Y, A, B, sel);  //instantiate the mux

   initial begin

     A = 32'hAAAAAAAA;
     B = 32'h55555555;
     sel = 1'b1;
     #10;
     A = 32'h00000000;
     #10
     sel = 1'b1;
     #10;
     B = 32'hFFFFFFFF;
     #5;
     A = 32'hA5A5A5A5;
     #5;
     sel = 1'b0;
     B = 32'hDDDDDDDD;
     #5;
     sel = 1'bx;
   end

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

endmodule  // test
```

Listing 1.2: The testbench for the multiplexor in figure 1.5 on page Lab 1–4

```
// Filename         : test-incr.v
// Description      : Test for incr.v, an incrementer by 1
//                    (32-bit input)

`include "incr.v"

module test ();

// Port Wires
   wire [31:0] IncrOut;

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

   INCR incr1 (IncrOut, A);  //instantiate the incrementer

   initial begin
     #10
     A = 3;
     #10;
     A = 15;
     #10
     A = 64;
     #5;
   end

   always @(A)
     #1 $display("Time = %0d\tA=%0d\tIncrOut=%0d",$time, A,
         IncrOut);

endmodule  // test
```

Listing 1.3: The testbench for the incrementer in figure 1.6 on page Lab 1–4

LAB 2

The ID pipeline stage.

Objective: To implement and test the Instruction Decode (ID) pipeline stage and integrate it with the IF stage.

This is part of a series of labs to implement the MIPS Datapath (figure 1.1 on page Lab 1–2) as a behavioral model in Verilog and simulate it.

For this week, you will implement the ID stage figure 2 on page Lab 2–2, integrate it together with the IF stage of last week, and test both together.

The parent module PIPELINE instantiates I_FETCH (from the previous lab) and I_DECODE.

- The module I_DECODE instantiates the modules CONTROL, REG, S_EXTEND, and ID_EX
  - The CONTROL module has input the opcode field of the IF_ID_instr and output is the 9-bit control bits which are shown in figure 2.2 on page Lab 2–2.
  - The register file REG, which has 32 general purpose registers, and has input the rs and rt fields of IF_ID_instr, MEM_WB_Writereg, MEM_WB_Writedata, and RegWrite (for the time being it can be from anywhere). Outputs are the contents of register rs and register rt.
  - The combinational module S_EXTEND receives as input the 16-bit immediate field of IF_ID_instr and output is the 32 bit sign-extended value.
  - The ID_EX module includes the pipeline register ID/EX and inputs the outputs of the CONTROL, REG, S_EXTEND modules, as well as the IF_ID_NPC, IF_ID_Instr[20-16] and IF_ID_Instr[15-11]. Outputs are the control bits (9 bits) NPC, Reg[rs], Reg[rt], signExtended (32 bit), Instr[20-16], and Instr[15-11].
- Testing: Initialize memory to the following hex values, beginning with location 0, and label and print out the outputs of the ID_EX register. The control bits should be binary and all other values should be decimal. Simulate for sufficient cycles so that all instructions go through the ID_EX register.

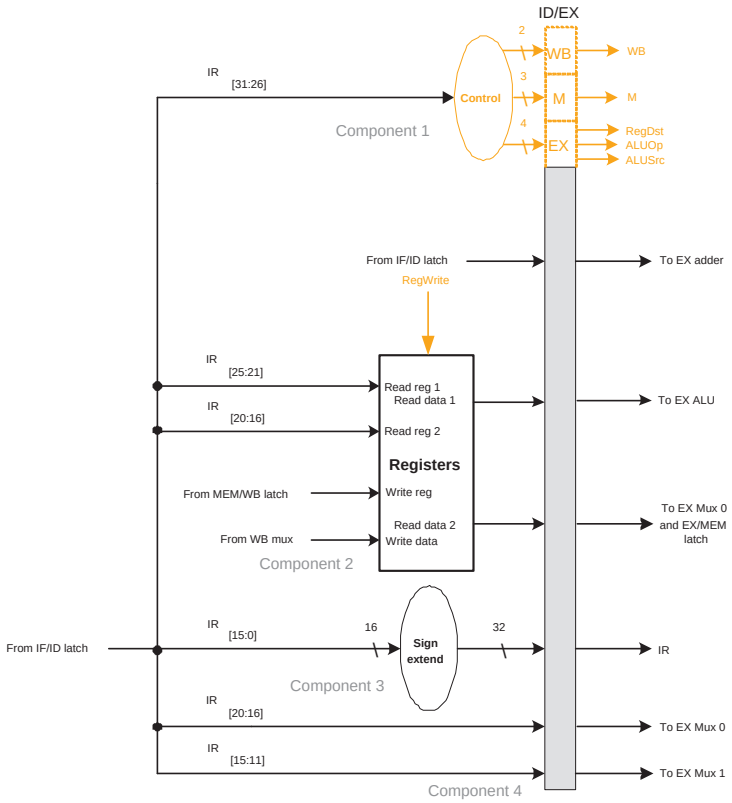


Figure 2.1: The ID stage

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Figure 2.2: ALUOp Control Bit and Function Code Sets (after [4])

```
002300AA
10654321
00100022
8C123456
8F123456
AD654321
13012345
AC654321
12012345
```

- Turn in the source code and the printout of the clock cycle number and outputs off the ID_EX register, as in testing above. Be ready to demonstrate.

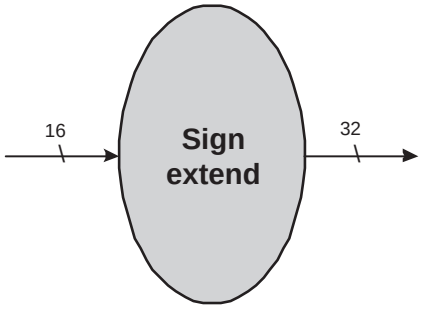


Figure 2.3: The sign-extend unit

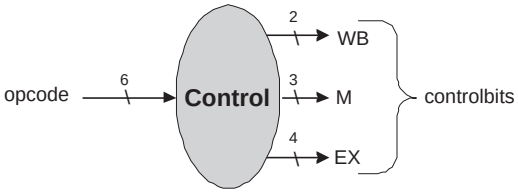


Figure 2.4: The control unit

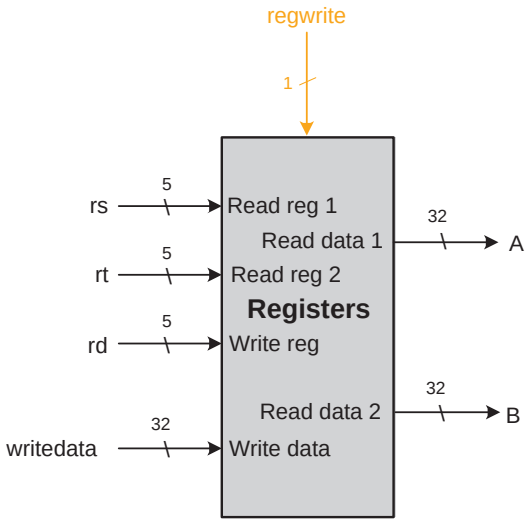


Figure 2.5: The register file

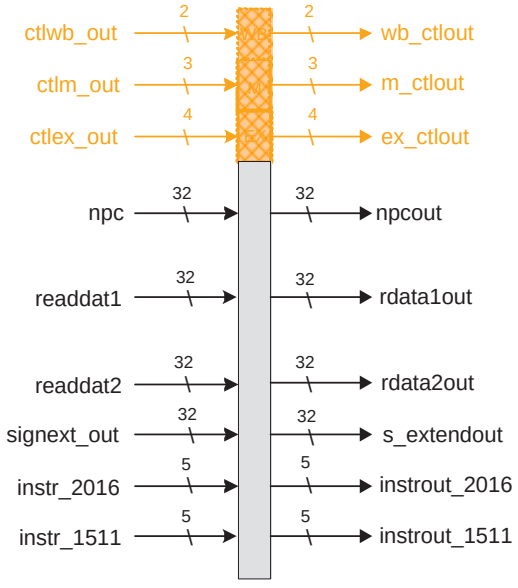


Figure 2.6: The ID/EX pipeline register (latch)

LAB 3

The EX pipeline stage

Objective: To implement and test the Execution (EX) pipeline stage and integrate it with the IF and ID stages.

This is part of a series of labs to implement the MIPS Datapath (figure 1.1 on page Lab 1–2) as a behavioral model in Verilog and simulate it.

For this week, you will implement the EX stage figure 3 on page Lab 3–2, integrate it together with the IF and ID stages of the previous week, and test three of them together.

The parent module PIPELINE instantiates I_FETCH (from the previous lab) and I_DECODE.

CHANGE: Please note that the "Shift left 2" unit that exists in Figure 6.30 in Paterson and Hennessy's book [4], is eliminated in figure 1.1 on page Lab 1–2. Now there is direct line from the input of "Shift left 2" to its output.

The parent module PIPELINE instantiates I_FETCH, and I_DECODE, and I_EXECUTE.

- The module I_EXECUTE instantiates the following modules
  - ADDER for the branch target address computation.
  - The ALU_CONTROL. Inputs are ALUop bits and the function bits. The specification is found in figure 1 on page Lab 1–3.
  - The instructions to be implemented are specified in the figure 3.2 on page Lab 3–2, and they are add, subtract, and, or, set less than. If the input information does not correspond to any valid instruction, ALUop = 11 and ALU output is 32 x's.
  - BOTTOM_MUX. Notice that the inputs and output are 5 bits.
  - ALU_MAX. Notice that you may instantiate the previously made MUX in the IF stage.
  - ID_MEM, the pipeline register.
- In the I_FETCH module we have resetmargins=true

```
reg          EX_MEM_PCSrc;
reg  [31:0]  EX_MEM_NPC;
```

- Move the above to the EX_MEM module, since EX_MEM exists now, and make them as inputs to the I_FETCH module.

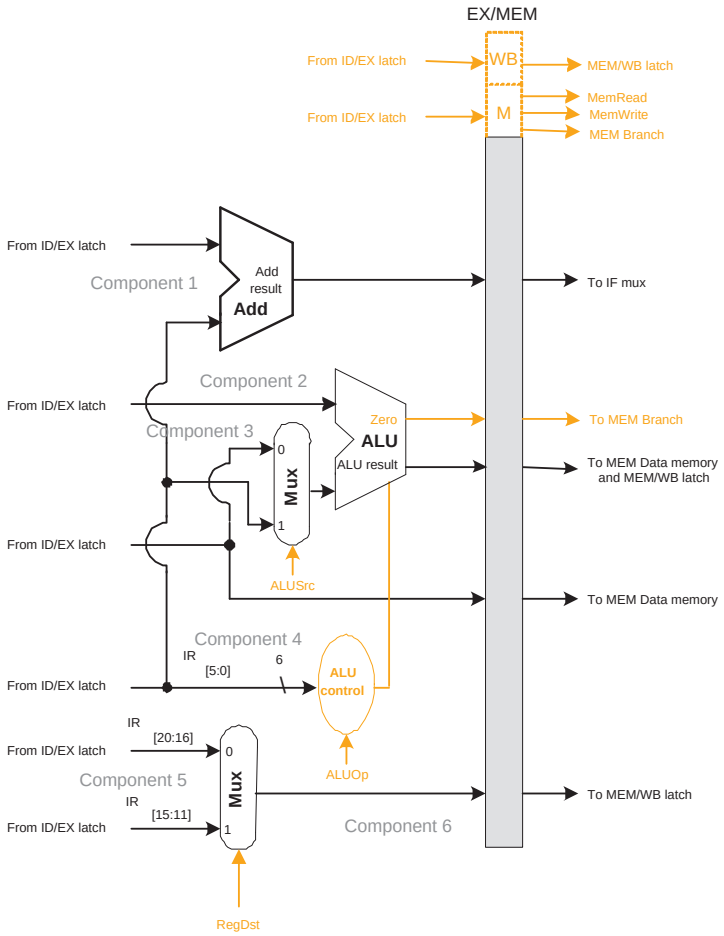


Figure 3.1: The EX stage

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less than	101010	set on less than	111

Figure 3.2: ALUOp Control Bit and Function Code Sets (after [4])

- Testing: Initialize memory to the following hex values, beginning with location 0, and label and print out the outputs of the ID_EX and EX_MEM registers. The control bits should be binary and all other values should be decimal. Simulate for sufficient cycles so that all instructions go through the EX_MEM register.

```
002300AA
10654321
00100022
8C123456
8F123456
AD654321
13012345
AC654321
12012345
```

- Be ready to use initialize memory with a given different set of instructions. Turn in the source code and the printout of the clock cycle number and outputs of the ID_EX register and the EX_MEM register. Be ready to demonstrate.

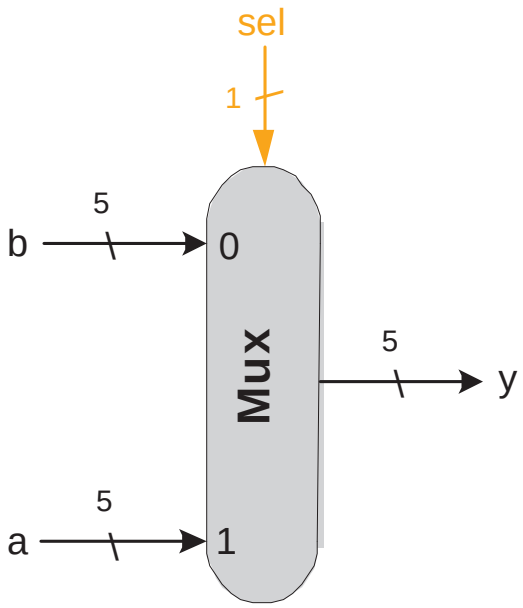


Figure 3.3: The multiplexer

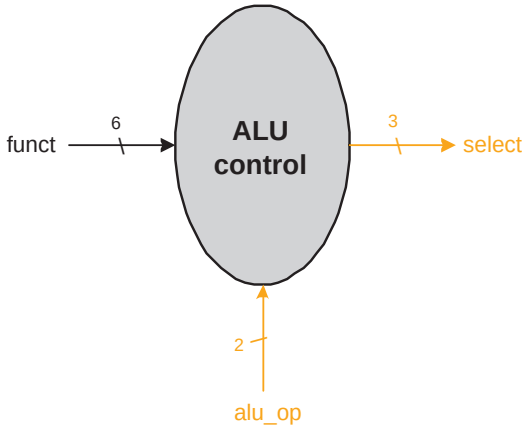


Figure 3.4: The ALU control unit

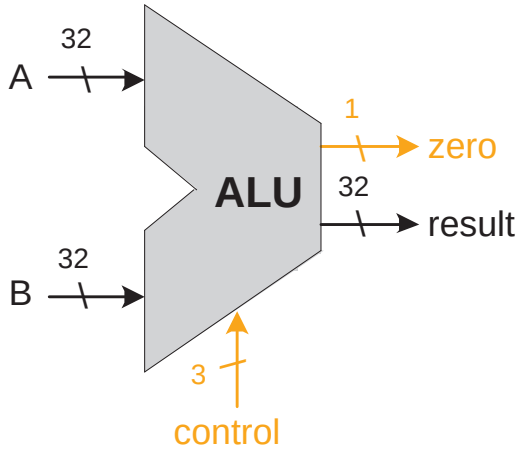


Figure 3.5: The ALU

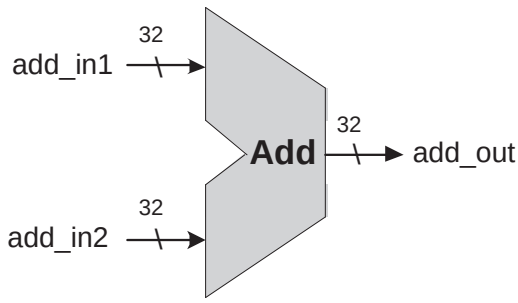


Figure 3.6: The adder

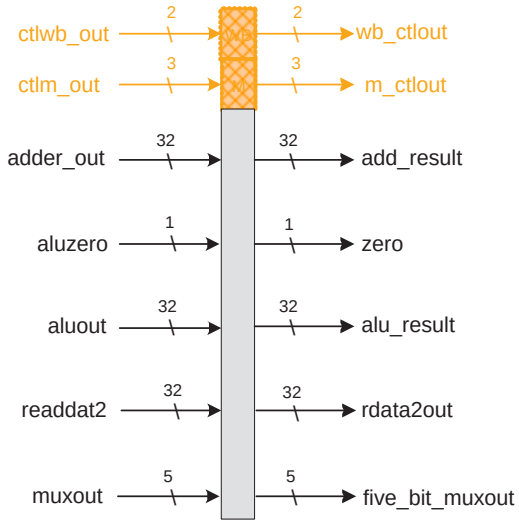


Figure 3.7: The EX/MEM pipeline register (latch)

3.1 Testbenches

Testbenches and their corresponding outputs are given for the 5-bit multiplexer (figure 3.3 on page Lab 3–3 and listing 3.1 on page Lab 3–6), the ALU control unit (figure 3.4 on page Lab 3–4 and listing 3.2 on page Lab 3–7), and the ALU (figure 3.5 on page Lab 3–4 and listing 3.3 on page Lab 3–9). The results of running the testbench for the multiplexer are in figure 3.8, for the ALU control unit in figure 3.9 on page Lab 3–8, and for the ALU in figure 3.10 on page Lab 3–8.

```
Running...
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
0 Errors, 0 Warnings
Normal exit
```

Figure 3.8: The output when running the testbench for the 5-bit multiplexer (listing 3.1 on page Lab 3–6)

```
// Filename          : test_5bitmux.v
// Description       : Testing the 5bit_mux module
//                     of the EX stage of the pipeline.

`include "5bit_mux.v"

module test();

// Wire Ports
   wire [4:0] Y;

// Register Declarations
   reg  [4:0] A, B;
   reg        sel;

   MUX5 mux1 (Y, A, B, sel);  //instantiate the mux

   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  // test
```

Listing 3.1: The testbench for the 5-bit multiplexor in figure 3.3 on page Lab 3–3

```
// Filename         : test-alucontrol.v
// Description      : Testing the ALU control module
//                    of the EX stage of the pipeline.
`include "alu-control.v"

module test ();
// Wire Ports
wire [2:0] select;

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

ALU_CONTROL alucontrol1 (select, alu_op, funct, );

initial begin
   alu_op = 2'b00;
   funct = 6'b100000;
   $monitor("ALUOp = %b\tfunct = %b\tselect = %b", alu_op, funct,
       select);
   #1
   alu_op = 2'b01;
   funct = 6'b100000;
   #1
   alu_op = 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 // test
```

Listing 3.2: The testbench for the ALU control in figure 3.4 on page Lab 3–4

```
Running...
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
Exiting VeriLogger at simulation time 7000
0 Errors, 0 Warnings

Normal exit
```

Figure 3.9: The output when running the testbench for the ALU control (listing 3.2 on page Lab 3–7)

```
Running...
A = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
B = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ALUOp = 011 result = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ALUOp = 100 result = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ALUOp = 010 result = 00000000000000000000000000010001
ALUOp = 111 result = 00000000000000000000000000000000
ALUOp = 011 result = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ALUOp = 110 result = 00000000000000000000000000000011
ALUOp = 001 result = 00000000000000000000000000001111
ALUOp = 000 result = 00000000000000000000000000000010
Exiting VeriLogger at simulation time 8000
0 Errors, 0 Warnings

Normal exit
```

Figure 3.10: The output when running the testbench for the ALU (listing 3.3 on page Lab 3–9)

```
// Filename         : test-alu.v
// Description      : Test module for the ALU

`include "alu.v"

module test ();

// Register Declarations
   reg  [31:0] A,B;
   reg  [02:0] 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 <= 'b011;
      #1
      control <= 'b110;
      #1
      control <= 'b001;
      #1
      control <= 'b000;
      #1
      $finish;

   end

   ALU ALU1(result, zero, A, B, control);

endmodule // test
```

Listing 3.3: The testbench for the ALU in figure 3.5 on page Lab 3–4

LAB 4

The MEM pipeline stage.

Objective: To implement and test the Memory (MEM) pipeline stage and integrate it with the IF, ID, and EX stages.

This is part of a series of labs to implement the MIPS Datapath (figure 1.1 on page Lab 1–2) as a behavioral model in Verilog and simulate it.

For this week, you will implement the MEM stage figure 4 on page Lab 4–2, and integrate it together with the IF, ID, and EX stages of previous weeks, and test all of them together.

The parent module PIPELINE instantiates I_FETCH (from the previous lab) and I_DECODE.

The parent module PIPELINE instantiates I_FETCH, I_DECODE, I_EXECUTE, MEM, and WB modules.

- The module MEMORY instantiates the following modules:
  - D_MEM: the data memory module. Data memory has 256 32-bit words.
  - MEM_WB: The pipeline register MEM/WB. The I_FETCH module should receive inputs 'write data", "write register" and RegWrite from the MEM modules.
- Testing: Initialize memory to the following hex values, beginning with location 0, and label and print out the outputs of the ID_EX, EX_MEM, and MEM registers. The control bits should be binary and all other values should be decimal. Simulate for sufficient cycles so that all instructions go through the MEM_WB register.

```
002300AA
10654321
00100022
8C123456
8F123456
AD654321
13012345
AC654321
12012345
```

- Be ready to use initialize memory with a given different set of instructions.
- Turn in the source code and the printout of the clock cycle number and outputs the ID_EX register, the EX_EM register, and the MEM_WB register. Be ready to demonstrate.

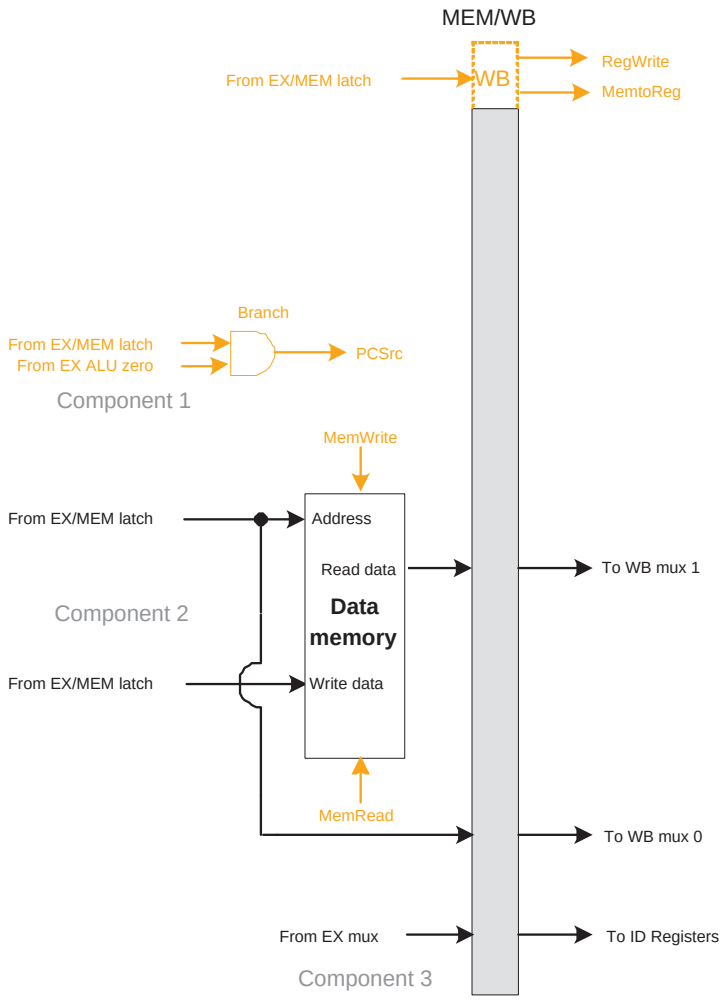


Figure 4.1: The MEM stage

Figure 4.2: The and gate

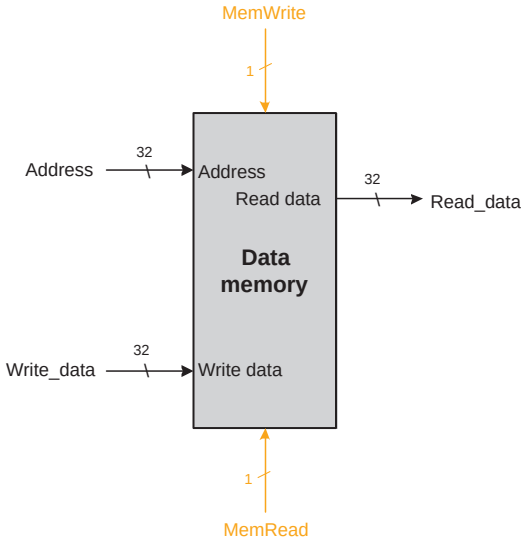


Figure 4.3: The data memory unit

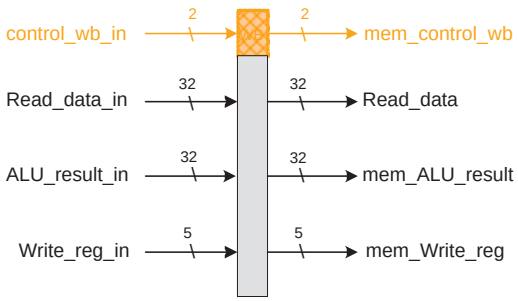


Figure 4.4: The MEM/WB pipeline register (latch)

LAB 5

The WB pipeline stage

Objective: To implement and test the Write-back (WB) pipeline stage and integrate it with the IF, ID, EX, and MEM stages.

This is part of a series of labs to implement the MIPS Datapath (figure 1.1 on page Lab 1–2) as a behavioral model in Verilog and simulate it.

For this week, you will implement the WB stage figure 5, and integrate it together with the IF, ID, EX, and MEM stages of previous weeks, and test all of them together.

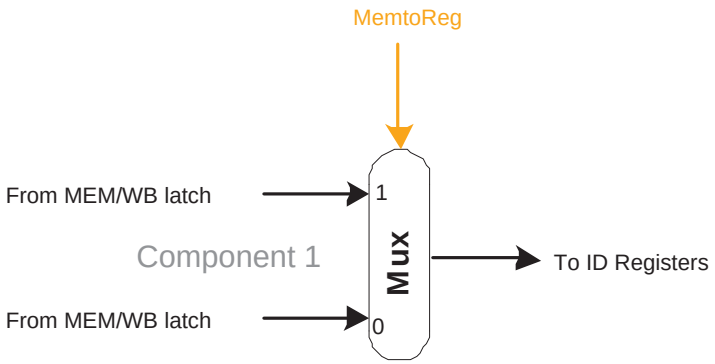


Figure 5.1: The WB stage

The parent module PIPELINE instantiates I_FETCH (from the previous lab) and I_DECODE.

The parent module PIPELINE instantiates I_FETCH, I_DECODE, I_EXECUTE, MEM, and WB modules.

- The module WB instantiates the following module:
  - MUX, the multiplexer at the output of MEM/WB. Instantiate the multiplexer at the IF stage.
- The WB module receives inputs from the MEM_WB module MemeReg, ReadData, ALUResult, and gives output WriteData.

- Testing: Initialize memory to the following hex values, beginning with location 0, and label and print out the outputs of the ID_EX, EX_MEM, and MEM registers. The control bits should be binary and all other values should be decimal. Simulate for sufficient cycles so that all instructions go through the MEM_WB register.

```
002300AA
10654321
00100022
8C123456
8F123456
AD654321
13012345
AC654321
12012345
```

  Be ready to use initialize memory with a given different set of instructions.

- Turn in the source code and the printout of the clock cycle number and outputs the ID_EX register, the EX_EM register, and the MEM_WB register. Be ready to demonstrate.

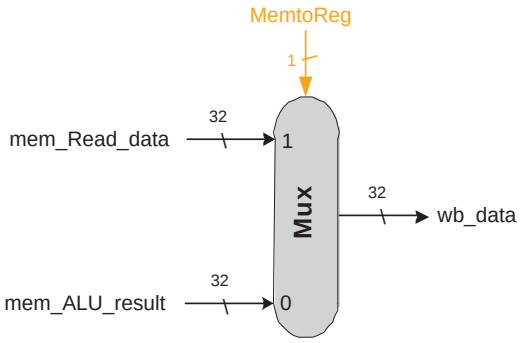


Figure 5.2: The multiplexer

LAB 6

Testing the MIPS datapath

Objective: To implement and test the MIPS datapath which was built in the previous labs.

This is part of a series of labs to implement the MIPS Datapath (figure 1.1 on page Lab 1–2) as a behavioral model in Verilog and simulate it.

For this week you will test the datapath with the given binary program in listing 6.1 on page Lab 6–2. Save it to file named "risc.txt". Also the initial contents of memory are given in listing 6.2 on page Lab 6–2. Save it to a file named "data.txt".

How to read data in the instruction memory (in the instruction memory module):

```
initial begin
   $readmemb("risc.txt",MEM);
   for (i=0;i< 24; i = i+1)
     $display(MEM[i]);
end
```

Similarly for the data memory module:

```
initial begin
   $readmemb("data.txt",MEM);
   for (i=0;i< 6; i = i+1)
     $display(MEM[i]);
end
```

- You must add the NOP instruction in the control module. Its opcode is 100000 and the output of the control unit should be all zeros.
- Make sure that r0 is initialized to zero.
- For testing, display registers r1, r2, and r3 from the ID/EX module. Register r1 should have values 1, 3, 6, 12, ... Simulate for 24 cycles.

```
// Program that adds the numbers (1 + 2) + 3 + 6 + 0 = 12
// And places 12 in register 1
100011_00000_00001_0000_0000_0000_0001  // LW r1, 1(r0)
100011_00000_00001_0000_0000_0000_0010  //LW r2, 2(r0)
100011_00000_00001_0000_0000_0000_0011  //LW r3, 3(r0)
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
000000_00001_00010_00001_00000_100000   //ADD r1, r1, r2
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
000000_00001_00011_00001_00000_100000   //ADD r1, r1, r3
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
000000_00001_00001_00001_00000_100000   //ADD r1, r1, r1
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
000000_00001_00000_00001_00000_100000   //ADD r1, r1, r0
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
1000_0000_0000_0000_0000_0000_0000_0000 //NOP
```

Listing 6.1: Binary code for testing the MIPS datapath.

```
// Contents of data memory
0000_0000_0000_0000_0000_0000_0000_0000 // Data 0
0000_0000_0000_0000_0000_0000_0000_0001 // Data 1
0000_0000_0000_0000_0000_0000_0000_0010 // Data 2
0000_0000_0000_0000_0000_0000_0000_0011 // Data 3
0000_0000_0000_0000_0000_0000_0000_0100 // Data 4
0000_0000_0000_0000_0000_0000_0000_0101 // Data 5
```

Listing 6.2: Initial data for memory.

Bibliography

[1] Michael D. Ciletti. *Modeling, synthesis, and rapid prototyping with the Verilog HDL.* Prentice-Hall, 1999.

[2] Dr. Daniel C. Hyde. *CSCI 320 Computer Architecture Handbook on Verilog HDL.* Bucknell University, 1997.

[3] IEEE - Institute of Electrical and Electronics Engineers. *IEEE 1364-1995 Verilog Language Reference Manual*, 1995.

[4] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/-Software Interface (Second Edition).* Morgan Kaufmann, San Mateo, 1998.

[5] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic, 1991.