Lab 2 - Program Counter

Steve Potter

January 26, 2018

1 Introduction

The goal of this lab is to create an adder module and a multiplexer module. These modules will initially be used in the Fetch stage of our 64-bit ARMv8 processor. The adder will be used to increment the Program Counter (PC). The incremented PC will be used for sequential program execution. The mux will be used to set the PC to either the incremented PC or to a branch address. This selection will be based on the mux control line, which specifies whether the program should branch or continue running sequentially.

2 Interface

The inputs of the adder module are a_in and b_in, which should be populated with the two numbers that you want to add. The output is add_out, which contains the sum that was produced in the adder module.

The inputs of the mux are a_in, b_in, and control. a_in and b_in are the two signals that are available to be selected, and control is the signal that is used to select whether the value of a_in or b_in will be chosen by the mux. The output of the mux is mux_out, which contains the value that the mux selected.

3 Design

The first module to build is the adder module. The adder module is designed to take two 64-bit input values and produce a 64-bit sum as an output.

The second module to build is the mux module. The mux module is designed to take two input values and select one of the two inputs to connect to the output line. This selection is done by evaluating the control line. If the control line is 0, then the mux should output the value of a_in. If the control line is 1, then the mux should output the value of b_in. The size of the a_in, b_in, and mux_out can be assigned when the module is instantiated, allowing this module to be used for signals of any size.

4 Implementation

The adder module is implemented by adding a_in + b_in and assigning the sum to add_out. This is done with a continuous assignment because we are using combinational logic, where the output should be constantly evaluated and update immediately when inputs change. View the adder code in Listing 1 on page 2.

Listing 1: Verilog code for implementing an adder.

```
`include "definitions.vh"

module adder(
    input [`WORD-1:0] a_in,
    input [`WORD-1:0] b_in,
    output [`WORD-1:0] add_out
    );
    assign add_out = a_in+b_in;
endmodule
```

The mux module is implemented using the conditional operator (?). If the control signal is 1, then mux_out is set to b_in. If the control signal is 1, then mux_out is set to a_in. Note that this operation is done in a continuous assign statement. The continuous assignment is used because we need combinational logic, where the output should be constantly evaluated and update immediately when inputs change.

The other interesting aspect of the implementation is the use of the SIZE parameter. The SIZE parameter allows the user of the module to specify the SIZE when instantiating the module, which allows this module to be used as a mux of any size. To utilize this feature to instantiate a 64-bit mux, you would instantiate it as mux#(64) UUT. If you do not specify a size, the default value of 8 will be used. View the mux code in Listing 2 on page 2.

Listing 2: Verilog code for implementing a mux.

```
`include "definitions.vh"

module mux#(
    parameter SIZE=8)(
    input [SIZE-1:0] a_in,
    input [SIZE-1:0] b_in,
    input control,
    output [SIZE-1:0] mux_out
    );
    assign mux_out = control?b_in:a_in;
endmodule
```

5 Test Bench Design

The adder test bench creates regs for a and b as well as a wire for the add_out output of the adder module. All signals are 64-bits. An adder module, UUT, is instantiated, and then a series of a and b inputs are set in the initial section. Regs a and b are set using non-blocking procedural assignments, then a cycle delay is initiated. As defined in definitions.vh, a cycle is 10ns. Then a and b are changed several times, with various amounts of delay. The goal is to test a variety of inputs and outputs at various intervals. The results of the test are verified by evaluating the simulation results and verifying that add_out equals a + b at all times. View the adder_test code in Listing 3 on page 3.

Listing 3: Verilog code for testing the adder.

```
`include "definitions.vh"

module adder_test;

reg [`WORD - 1:0] a;
reg [`WORD - 1:0] b;
wire [`WORD - 1:0] add_out;

adder UUT(
    .a_in(a),
    .b_in(b),
    .add_out(add_out)
    );

initial
begin
    a<='WORD'd0;
    b<='WORD'd5; #`CYCLE;
    a<='WORD'd55; #`CYCLE;
    b<='WORD'd59000; #`CYCLE;
    a<='WORD'd1; #(`CYCLE/2);
    b<='WORD'd24; #(2*`CYCLE);
    b<='WORD'd8; #`CYCLE
    $finish;
end
endmodule
```

The mux test bench creates regs for a, b, and control as well as a wire for the mux_out output of the mux module. All signals are 64-bits. An adder module, UUT, is instantiated with a size of 64 by using the syntax of mux#(64) UUT. Then a series of a and b inputs are set in the initial section, and the control line is set to 0. a and b are varied with various amounts of delay. After several cycles, control is changed to 1, causing the value of b to be sent to mux_out.

And then control is set back to 0 to verify that the mux can switch back and forth effectively. The results of the test are verified by evaluating the simulation results and verifying that:

1. mux_out always equals a when control is 0
2. mux_out always equals b when control is 1

View the mux_test code in Listing 4 on page 4.

Listing 4: Verilog code for testing the mux.

```
`include "definitions.vh"

module mux_test;

reg [`WORD - 1:0] a;
reg [`WORD - 1:0] b;
reg control;
wire [`WORD - 1:0] mux_out;

mux#(64) UUT(
    .a_in(a),
    .b_in(b),
    .control(control),
    .mux_out(mux_out)
    );

initial
begin
    control <=1'b0;
    a<='WORD'd0;
    b<='WORD'd5; #`CYCLE;
    a<='WORD'd55; #`CYCLE;
    control <=1'b1; #(`CYCLE/2);
    b<='WORD'd59000; #`CYCLE;
    a<='WORD'd1; #(`CYCLE/2);
    control <=1'b0;
    b<='WORD'd24; #(2*`CYCLE);
    b<='WORD'd8; #`CYCLE
    $finish;
end
endmodule
```

Figure 1: Timing diagram for the adder test.

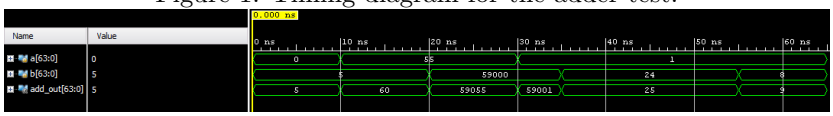
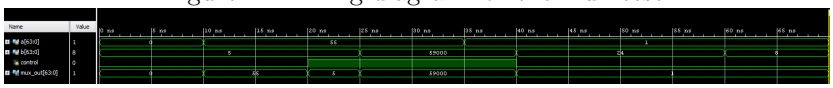


Figure 2: Timing diagram for the mux test.



6 Simulation

The Simulation Results of the adder test confirm that the adder works as expected, with add_out always producing the sum of a and b. The timing diagram can be viewed in Figure 1 on page 5.

The Simulation Results of the mux test confirm that the mux works as expected, with mux_out always matching the conditions listed in the Test Bench Design section. The timing diagram can be viewed in Figure 2 on page 5.

7 Conclusions

An adder module and mux module were successfully created. The adder module can be used in the Fetch stage to increment the PC, and it can be used any other place that 64-bit numbers need to be added. The mux module can now be used in the Fetch stage to select between the incremented PC and the branch target address. Since it was designed with a SIZE parameter, it can be used to instantiate a mux for any data size. While the Fetch stage will use a 64-bit mux, this flexibility will be useful in future stages.