Homework 1: Intro to SV Simulation

Patrick Laverty

Due February 3

1 Requirements

In this homework, we will ensure that each input/output can handle up to 64 bits. We will also utilize SystemVerilog as our choice of programming language to then simulate in ModelSim. I will also be using the same .do file format that was given on canvas, as it works well for each of these parts. (If it isn't broke don't fix it)

2 Part 1: Arithmetic

In this part we implement a module to do basic arithmetic. I put together the simple module:

simplearithmetic.sv

```
module simplearithmetic #(parameter N = 64) (
    input logic [N-1:0] A,B,C,
    output logic [N-1:0] Z);
    assign Z = (A * B) + C;
endmodule
```

I then put together my testbench to try with different random numbers, and then I checked each output manually to verify if it was correct.

Part 1: simplearithmetic_tb.sv

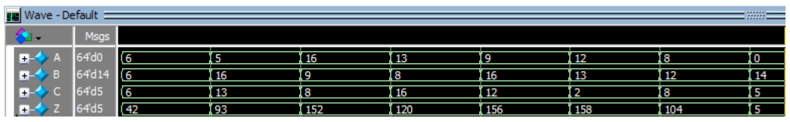
```
`timescale 1ps/1ps
module simplearithmetic_tb;
    logic [63:0] A, B, C;
    logic [63:0] Z;

    simplearithmetic #(.N(64)) mut(.A(A),
                                   .B(B),
                                   .C(C),
                                   .Z(Z));


    integer i;
    integer seed = 0; // Change this number to get different random numbers.
    initial begin
        #0 $display ("Using␣seed:␣%0d", seed);
        for (i = 0; i <= 10; i = i + 1) // We will test this module 10 times.
        begin
            // Generate random numbers based on the seed given
            assign A = $urandom(seed + i)%20;
            assign B = $urandom(seed + i * 2)%20;
            assign C = $urandom(seed + i * 3)%20;

            // Display iteration, random values, and the result
            #5 $display ("Iteration␣%0d", i);
            #0 $display ("Value␣of␣A␣=␣%0d,␣B␣=␣%0d,␣C␣=␣%0d", A, B, C);
            #0 $display ("Result␣of␣Z␣=␣A␣*␣B␣+␣C␣=␣%0d", Z);
            #0 $display ("");
        end
    end
endmodule
```

Part 1: Resulting Transcript

```
# Using seed: 0
# Iteration 0
# Value of A = 6, B = 6, C = 6
# Result of Z = A * B + C = 42
#
# Iteration 1
# Value of A = 5, B = 16, C = 13
# Result of Z = A * B + C = 93
#
# Iteration 2
# Value of A = 16, B = 9, C = 8
# Result of Z = A * B + C = 152
#
# Iteration 3
# Value of A = 13, B = 8, C = 16
# Result of Z = A * B + C = 120
#
# Iteration 4
# Value of A = 9, B = 16, C = 12
# Result of Z = A * B + C = 156
#
# Iteration 5
# Value of A = 12, B = 13, C = 2
# Result of Z = A * B + C = 158
#
# Iteration 6
# Value of A = 8, B = 12, C = 8
# Result of Z = A * B + C = 104
#
# Iteration 7
# Value of A = 0, B = 14, C = 5
# Result of Z = A * B + C = 5
#
# Iteration 8
# Value of A = 16, B = 18, C = 17
# Result of Z = A * B + C = 305
#
# Iteration 9
# Value of A = 16, B = 8, C = 8
# Result of Z = A * B + C = 136
#
# Iteration 10
# Value of A = 13, B = 0, C = 10
# Result of Z = A * B + C = 10
```

After doing so, I checked my simulation waveform to verify if it was correct as well:

Part 1: Resulting Simulation



```
Wave - Default
        Msgs
A  64'd0   6   5   16   13   9    12   8    0
B  64'd14  6   16  9    8    16   13   12   14
C  64'd5   6   13  8    16   12   2    8    5
Z  64'd5   42  93  152  120  156  158  104  5
```

Figure 1: Caption

We can clearly see that the arithmetic is correct and assigning Z accordingly.

3 Part 2: 8:1 Multiplexer

In this part we implement a module to do a 8 input 1 output multiplexer based on a 3-bit selection choice. I first put together the module to do the case switch statement based on the state of the selection bytes.

Part 2: mux8.sv

```
module testmux8 #(parameter N = 64)
    (input logic [2:0] s,
    input logic [N-1:0] d0, d1, d2, d3, d4, d5, d6, d7,
    output logic [N-1:0] y);

    always @ (d0, d1, d2, d3, d4, d5, d6, d7, s) begin
        case (s)
            3'b000  : y = d0;
            3'b001  : y = d1;
            3'b010  : y = d2;
            3'b011  : y = d3;
            3'b100  : y = d4;
            3'b101  : y = d5;
            3'b110  : y = d6;
            3'b111  : y = d7;
            default : y = d0;
        endcase
    end
endmodule
```

I then put together my testbench to generate random values for each of the input values. I also put together a randomization to the selection bits to make sure it was choosing the correct values. It will generate a new selection on every clock rising edge.

Part 2: mux8_tb.sv

```
`timescale 1ps/1ps
module mux8_tb;
    logic[2:0] s = 3'b000;
    logic [63:0] d0, d1, d2, d3, d4, d5, d6, d7;
    logic [63:0] y;
    logic clk = 1;
    integer count = 1;
    always clk = #5 ~clk;
    testmux8 #(.N(64)) mut(.s(s),
                          .d0(d0),
                          .d1(d1),
                          .d2(d2),
                          .d3(d3),
                          .d4(d4),
                          .d5(d5),
                          .d6(d6),
                          .d7(d7),
                          .y(y));

    integer seed = 540; // change seed to generate new random numbers
    always @(posedge clk) begin
        assign s = $urandom(seed + (count * 9))%7;
        assign count = count + 1;

        #0 $display("Value␣of␣y␣=␣%d", y);
    end
    initial begin
        // Generate random numbers based on the seed given
        assign d0 = $urandom(seed)%20;
        assign d1 = $urandom(seed * 2)%20;
        assign d2 = $urandom(seed * 3)%20;
        assign d3 = $urandom(seed * 4)%20;
        assign d4 = $urandom(seed * 5)%20;
        assign d5 = $urandom(seed * 6)%20;
        assign d6 = $urandom(seed * 7)%20;
        assign d7 = $urandom(seed * 8)%20;
        // Display iteration, random values, and the result
        #0 $display ("Value␣of␣d0␣=␣%0d", d0);
        #0 $display ("Value␣of␣d1␣=␣%0d", d1);
        #0 $display ("Value␣of␣d2␣=␣%0d", d2);
        #0 $display ("Value␣of␣d3␣=␣%0d", d3);
        #0 $display ("Value␣of␣d4␣=␣%0d", d4);
        #0 $display ("Value␣of␣d5␣=␣%0d", d5);
        #0 $display ("Value␣of␣d6␣=␣%0d", d6);
        #0 $display ("Value␣of␣d7␣=␣%0d", d7);
    end
endmodule
```

I then checked my simulation to ensure that it was operating properly.

Part 2: Resulting Simulation

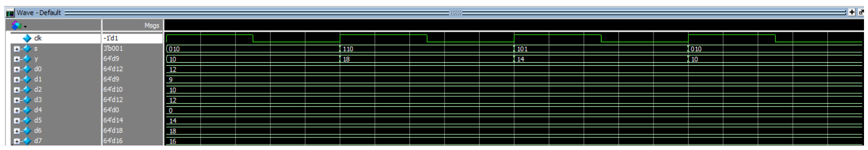


Figure 2: Caption

We can see clearly that this is correct due to each 3-bit selection corresponds to the desired output. The randomization of selection gives us a clear picture that our multiplexer is choosing the correct values.

4 Conclusion

In this homework we reviewed SystemVerilog and its capabilities to create simple arithmetic and multiplexer modules that can be further expanded out into adders and other complex uses that we will utilize in the future. We also learned the basics of simulating in ModelSim as a means to testing our designs.