EECE 144
Fall 2011

Lab Report #9
Section 4
11/2/2011

Submitted by: Jeremiah Mahler

Signature	Printed Name	Date
	Jeremiah Mahler	**Nov 02, 2011**
	Marvanee Johnson	**Nov 02, 2011**

1 Description/Objectives

The objective of this lab is to introduce Verilog [1] and GTKWave [2] by construction a function to implement a two-bit adder. The function is defined by Table 1.

	A_1	A_0	B_1	B_0	C	S_1	S_0
0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	1
2	0	0	1	0	0	1	0
3	0	0	1	1	0	1	1
4	0	1	0	0	0	0	1
5	0	1	0	1	0	1	0
6	0	1	1	0	0	1	1
7	0	1	1	1	1	0	0
8	1	0	0	0	0	1	0
9	1	0	0	1	0	1	1
10	1	0	1	0	1	0	0
11	1	0	1	1	1	0	1
12	1	1	0	0	0	1	1
13	1	1	0	1	1	0	0
14	1	1	1	0	1	0	1
15	1	1	1	1	1	1	0

Table 1: Truth table of the two bit adder. The two bit number { A_1, A_0 } is added to the two bit number { B_1, B_0 } resulting in the two bit number { S_1, S_0 } with the carry bit C.

2 Procedure

To construct the two bit adder in Verilog requires several steps. First, three functions corresponding to each output must be calculated using the truth table (Table 1). Then these functions must be defined in the Verilog format. Then the Verilog file must be compiled and run and its outputs verified against the desired outputs from the truth table. Additionally, GTKWave can be used to graphically visualize all the state changes over time.

The first function to be found is for the output C. The Karnaugh Map for this function is shown in Figure 1. Grouping the 1s together results in SOP Equation 1.

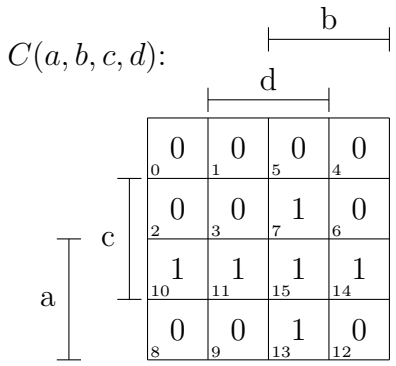

Figure 1: Karnaugh map of function C. The variables a, b, c and d correspond to A_1, A_0, B_1, B_0 in Table 1.

$$C = A_1B_1 + A_0B_1B_0 + A_1A_0B_0 \tag{1}$$

Next the output of the function S_1 is found. Its Karnaugh Map is shown in Figure 2 Grouping the 1s together results in SOP Equation 2.

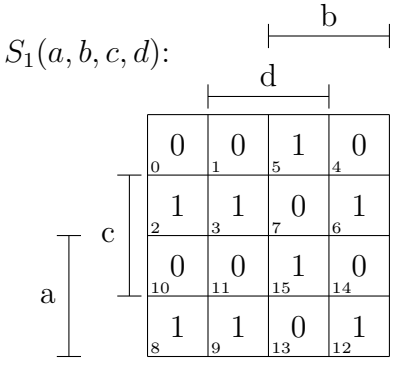

Figure 2: Karnaugh map of function S_1. The variables a, b, c and d correspond to A_1, A_0, B_1, B_0 in Table 1.

$$S_1 = A_1B_1'B_0' + A_1A_0'B_1' + A_1A_0B_1B_0 + A_1'A_0B_1'B_0 + A_1'A_0'B_1 + A_1'B_1B_0' \tag{2}$$

Finally the output of the function S_0 is found. Its Karnaugh Map is shown in Figure 3 Grouping the 1s together results in SOP Equation 3. This equation can be simplified to the XOR of A_0 and B_0 as shown in Equation 4.

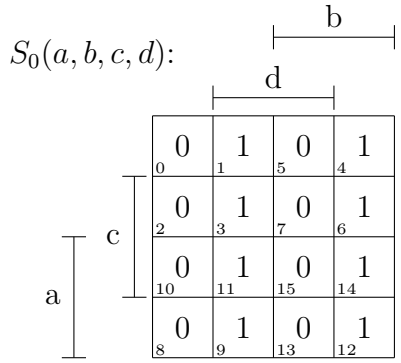

Figure 3: Karnaugh map of function S_1. The variables a, b, c and d correspond to A_1, A_0, B_1, B_0 in Table 1.

$$S_0 = A_0B_0' + A_0'B_0 \tag{3}$$
$$= A_0 \oplus B_0 \tag{4}$$

Now that the three functions corresponding to the three outputs have been found this can be programmed in Verilog. In Verilog there is more than one way to construct a function. In this instance we are using data flow modeling with operators and primitives. This form is very verbose and specifies each step explicitly (see Appendix A for the full source code). In contrast with data flow modeling is behavioral modeling which provides higher level construct such as if, case and looping.

Once the source code is entered it must be compiled. In this case we are using Icarus Verilog [1] under Linux.

```
jeri@bishop verilog$ ls
adder.v
jeri@bishop verilog$ iverilog adder.v
jeri@bishop verilog$ ls
adder.v  a.out
```

The file 'a.out' is the executable produced during compilation. Next the executable is run.

```
jeri@bishop verilog$ ./a.out
```

```
VCD info: dumpfile gtkwave-02.vcd opened for output.
a1 a0 b1 b0 | c s1 s0
--------------------
0  0  0  0  | 0  0  0
0  0  0  1  | 0  0  1
0  0  1  0  | 0  1  0
0  0  1  1  | 0  1  1
0  1  0  0  | 0  0  1
0  1  0  1  | 0  1  0
0  1  1  0  | 0  1  1
0  1  1  1  | 1  0  0
1  0  0  0  | 0  1  0
1  0  0  1  | 0  1  1
1  0  1  0  | 1  0  0
1  0  1  1  | 1  0  1
1  1  0  0  | 0  1  1
1  1  0  1  | 1  0  0
1  1  1  0  | 1  0  1
1  1  1  1  | 1  1  0
jeri@bishop verilog$ ls
adder.v  a.out  gtkwave-02.vcd
```

In this case the `$display` statements were arranged so that a truth table was output. This can be used to verify that the outputs agree with the truth table of the desired values. Also notice that the file `gtkwave-02.vcd` has been created. This file can be used with GTKWave to produce a display of the wave forms over time.

```
jeri@bishop verilog$ gtkwave gtkwave-02.vcd
```

3 Observations

The output values produced by Verilog agreed with those in the truth table of desired values. And GTKWave correctly displayed the bit values over time as shown in Figure 4

Verilog, just like every other programming language, has its own unique quirks and caveats that can trick even seasoned programmers. For example, Verilog uses the concept of 'modules' and they seem similar to functions in C. But their behavior is quite different. Conceptually using a module is like placing a gate on to a circuit, it will operate without needing to be called. Also Verilog is very specific about times. For example, if a `$display$` statement does not have a delay from a previous data assignment it may display the data before the assignment. See the source code in Appendix A for more details.

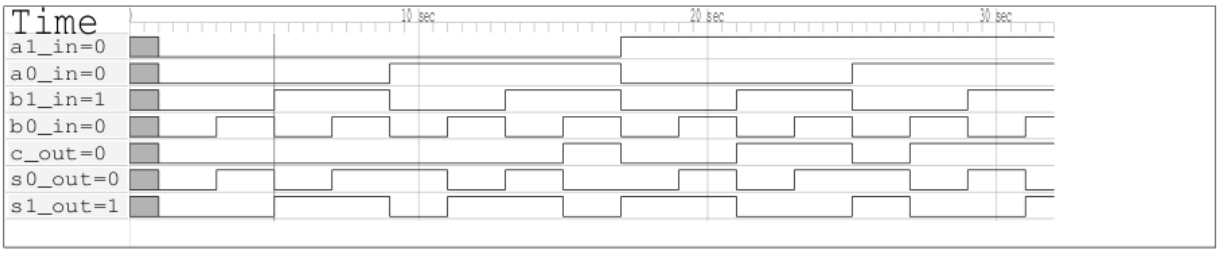

Figure 4: Output of GTKWave. The left column shows the variable names corresponding to their wave form. The names have been changed slightly as compared to the definition in Table 1. For example a0_in is A_0, b1_in is B_1, s1_out is S_1, and so on. In general output variables have _out appended and input variables have _in appended.

4 Conclusion

This lab was a success in introducing Verilog and GTKWave for analysis of digital circuits and creation of a two bit adder.

5 References

[1] S. Williams, "Icarus verilog." `http://iverilog.icarus.com/`, 2011.

[2] T. Bybell, "Gtkwave." `http://gtkwave.sourceforge.net/`, 2011.

A Verilog Source

```
/*
 *
 * Lab 9, EECE-144, Two-Bit Adder in Verilog
 *
 * To compile this file run:
 *
 *   iverilog this_file.v
 *
 * Run the executable:
 *
 *   ./a.out
 * OR
 *   vvp a.out
 *
 * Because $dumpfile and $dumpvars have been
 * added it will generate data for gtkwave
 * in some_file.vcd.
 *
 * This output file can then be shown with Gtkwave.
 *
 *   gtkwave some_file.vcd
 *
 * And from within Gtkwave you can pick and choose
 * variables to see their waveforms over time.
 */

// 'adder' is written in a "data flow modeling" style
// using primitives (and(), or(), ...) and operators(&, |, ~, ...)
// It is mostly written using operators in this case.
module adder(input a1,a0,b1, b0, output c, s1, s0);
    assign c =    (a1 & b1)
                | (a0 & b1 & b0) | (a1 & a0 & b0) ;

    assign s1 =   (a1 & ~b1 & ~b0)
                | (a1 & ~a0 & ~b1)
                | (a1 & a0 & b1 & b0)
                | (~a1 & a0 & ~b1 & b0)
                | (~a1 & ~a0 & b1)
                | (~a1 & b1 & ~b0);

    //assign s0 = (a0 & (~b0))| ((~a0) & b0);
    // simplifies to an XOR
    //assign s0 = a0 ^ b0;
    xor(s0, a0, b0);
endmodule

module test;
    reg a1_in, a0_in, b1_in, b0_in;
```

```
integer i;
    wire c_out, s1_out, s0_out;

     // Dont think of modules as being "called" as in C.
     // Instead think of this like soldering a chip to a board.
    adder adder1(.a1(a1_in), .a0(a0_in), .b1(b1_in), .b0(b0_in),
                    .c(c_out), .s1(s1_out), .s0(s0_out));

    initial begin
        // produce output for GTKWave
        $dumpfile("gtkwave-02.vcd");
        $dumpvars(0,test);

        // header for the formatted output
        $display("a1 a0 b1 b0 | c s1 s0");
        $display("---------------------");

for (i = 0; i < 16; i = i + 1) begin
#1 {a1_in, a0_in, b1_in, b0_in} = i;
#1 $display("%b  %b  %b  %b  | %b  %b  %b",
                a1_in, a0_in, b1_in, b0_in, c_out, s1_out, s0_out);
end
end
endmodule
```