Introduction to Verilog

Structure of a Verilog Program

- A Verilog program is structured as a set of modules, which may represent anything from a collection of logic gates to a complete system.
- A module specifies its input and output ports, which describe the incoming and outgoing connections of a module.
- A module may also declare additional variables.
- The body of a module consists of
  - initial constructs, which can initialize reg variables
  - continuous assignments, which define only combinational logic
  - always constructs, which can define either sequential or combinational logic
  - instances of other modules, which are used to implement the module being defined

Key things to remember

- A module in Verilog is NOT a function in software.
  - A function is a piece of code that can be called.
  - A module defines a functionality. A module can be USED, NOT CALLED. Once you use a module, a physical piece of hardware will be allocated in the chip. If you use a module twice, there will be two pieces of such hardware.
- Verilog is a Hardware Description Language.
  - You **describe** what you need.

Data Types

- A **wire** specifies a combinational signal.
  - Think of it as an actual wire.
- A **reg** (register) holds a value.
  - A reg need not necessarily correspond to an actual register in an implementation, although it often will.

Constants

- Constants are represented by prefixing the value with a decimal number specifying its size in bits.
- For example:
  - 4’b0100 specifies a 4-bit binary constant with the value 4, as does 4’d4.

Values

- The possible values for a register or wire in Verilog are
  - 0 or 1, representing logical false or true
  - x, representing unknown, the initial value given to all registers and to any wire not connected to something
  - z, representing the high-impedance state for tristate gates

Operators

- Verilog provides the full set of unary and binary operators from C, including
  - the arithmetic operators (+, –, *, /),
  - the logical operators (&, |, ^, ~),
  - the comparison operators (==, !=, >, <, <=, >=),
  - the shift operators (<<, >>)
  - Conditional operator (?, which is used in the form condition ? expr1 :expr2 and returns expr1 if the condition is true and expr2 if it is false).

The half-adder. Example of continuous assignments

```
module half_adder (A,B,Sum,Carry);
      input A,B;
      output Sum, Carry;
      assign Sum = A ^ B;
      assign Carry = A & B;
endmodule
```

- **assign**: continuous assignments. Any change in the input is reflected immediately in the output.
- Wires may be assigned values only with continuous assignments.

Behavioral description – The always block

```
module two_one_Selector (A,B,Sel,O);
    input A,B,Sel;
    output reg O;

    always @(A, B, Sel)
        if (Sel == 0)
            O <= A;
        else
            O <= B;
endmodule
```

always

- always @(A, B, Sel) – means that the block is reevaluated every time any one of the signals in the list changes value
- NOT A FUNCTION CALL
- If no sensitive list, always evaluated
- Always keep in mind that it is used to describe the behavior of a piece of hardware you wish to design. Basically, it is used to tell Verilog what kind of gates should be used.

Always block continued

- Only reg variables can be assigned values in the always block – output reg O;
- When we want to describe combinational logic using an always block, care must be taken to ensure that the reg does not synthesize into a register.

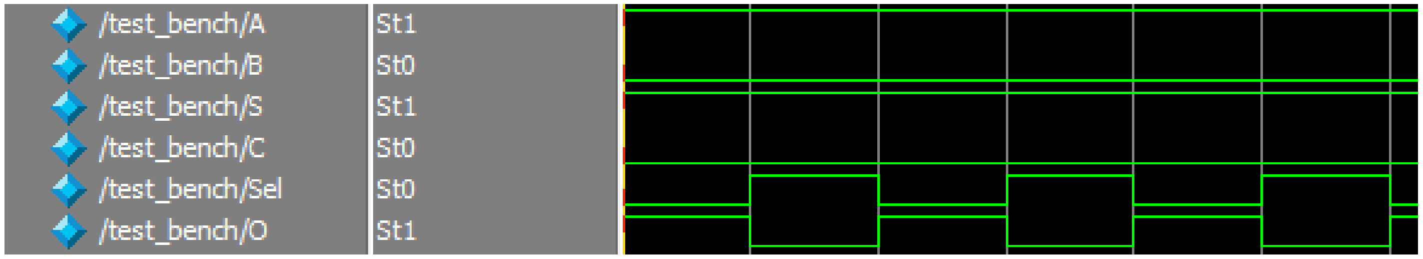
Always continued

- reg variables can be assigned values in the always block in two ways:
  - ``=’’ the blocking assignment. Just like C. The assignment will be carried out one-by-one. One thing does not happen until the one before it happens. This is the behavior of the circuit!
  - ``<=’’ the nonblocking assignment. All assignment happen at the same time.

A Sample Verilog code

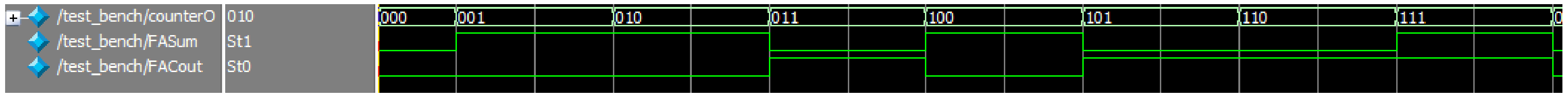
```
module half_adder (A,B,Sum,Carry);
            input A,B;
            output Sum, Carry;

            assign Sum = A ^ B;
            assign Carry = A & B;

endmodule

module two_one_Selector (A,B,Sel,O);
            input A,B,Sel;
            output reg O;
            //output O;

            always @(A, B, Sel)
                        if (Sel == 0)
                                    O <= A;
                        else
                                    O <= B;


endmodule
```

```
module half_adder_test_bench ();

            wire A,B,S,C,Sel,O;
            reg osc;

            initial begin
                        osc = 0;
            end

            always begin
                        #10 osc = ~osc;
            End

            assign A=1;
            assign B=0;
            assign Sel=osc;

            half_adder A1(A, B, S, C);
            two_one_Selector S1(A,B,Sel,O);

endmodule
```



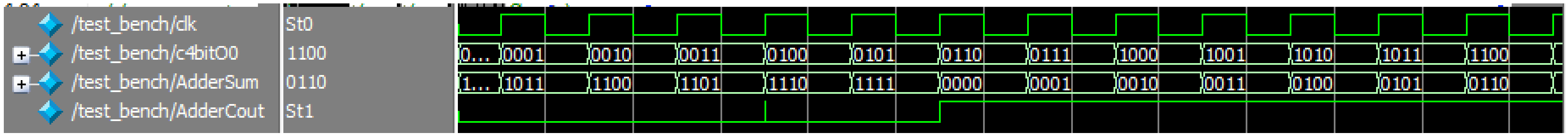
One-bit Full Adder

```
module full_adder (A,B,Cin,Sum, Cout);
                input A,B,Cin;
                output Sum, Cout;

                assign Sum = (A & B & Cin) | (~A & ~B & Cin) | (~A & B & ~Cin) | (A & ~B & ~Cin);
                assign Cout = (A & Cin) | (A & B) | (B & Cin);

endmodule
```



Four-bit Adder

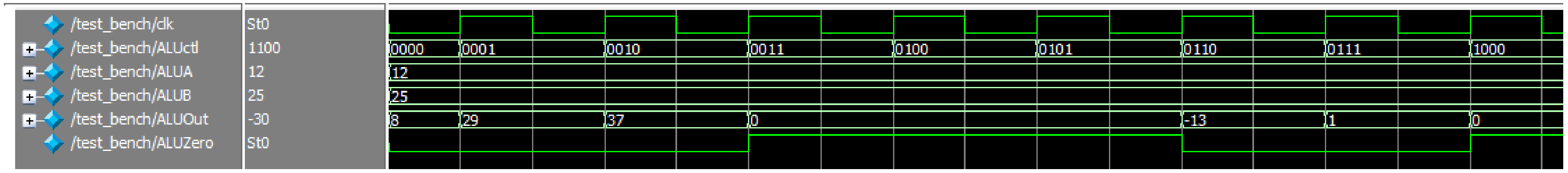
```
module four_bit_adder (A,B,Cin,Sum, Cout);
        input [3:0] A;
        input [3:0] B;
        input Cin;
        output [3:0] Sum;
        output Cout;

        wire C0, C1, C2;

        full_adder FA1(A[0], B[0], Cin, Sum[0], C0);
        full_adder FA2(A[1], B[1], C0, Sum[1], C1);
        full_adder FA3(A[2], B[2], C1, Sum[2], C2);
        full_adder FA4(A[3], B[3], C2, Sum[3], Cout);

endmodule
```



```
/test_bench/clk        St0
/test_bench/c4bitO0    1100   0... 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100
/test_bench/AdderSum   0110   1... 1011 1100 1101 1110 1111 0000 0001 0010 0011 0100 0101 0110
/test_bench/AdderCout  St1
```

MIPS ALU

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
        input [3:0] ALUctl;
        input [31:0] A,B;
        output reg [31:0] ALUOut;
        output Zero;

        assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere

        always @(ALUctl, A, B) //reevaluate if these change
        case (ALUctl)
                0: ALUOut <= A & B;
                1: ALUOut <= A | B;
                2: ALUOut <= A + B;
                6: ALUOut <= A - B;
                7: ALUOut <= A < B ? 1:0;
                12: ALUOut <= ~(A | B); // result is nor
                default: ALUOut <= 0; //default to 0, should not happen;
        endcase

endmodule
```



```
/test_bench/clk       St0
/test_bench/ALUctl    1100   0000 0001 0010 0011 0100 0101 0110 0111 1000
/test_bench/ALUA      12     12
/test_bench/ALUB      25     25
/test_bench/ALUOut    -30    8 29 37 0 -13 1 0
/test_bench/ALUZero   St0
```

Instructions about the Verilog Simulator

- You may download and install Altera ModelSim by first going to page: https://www.altera.com/download/sw/dnl-sw-index.jsp

  then, under the select software tab, download a version of “ModelSim-Altera Starter Edition.” I’m using “10.1d for Quartus II v13.0 for Windows. The file is 800MB and took me around 15 minutes to download.

To run a simulation, for v13.0, you may

- Open your Verilog file.
- In the menu bar, click “Compile.” In the drop-down menu, click “Compile.” A dialogue window should pop-up. Click the “Compile” button. If there is no problem with your code, the compile should pass, and then you should click the “Done” button to close the dialogue window.
- In the menu bar, click “Simulate.” In the drop-down menu, click “Start Simulation.” A dialogue window should pop-up. There should be a list showing up in the window. Click on the “+” sign on “work” which is the first item in the list. Click on “test_bench.” Then click the “OK” button. The dialogue should disappear.
- After a little while, maybe 1 second, left click on “test_bench” in the “work space” window to select it, then right click. In the menu, select “Add”, then “To Wave,” then “All items in region.”
- There will be a new window waveform popping up. First, find the box showing “100 ps” and change it to “1000 ps.” Then click the run simulation sign right next to the box. The waveform should be ready!

Midterm

Midterm

- Grades:

A:	3
B:	2
C:	2
D:	3
F:	13

- Stats:

Mean:	54
Median:	51
Standard Deviation:	21

- Letter Ranges

A:	85<= S
B:	73<=S<85
C:	64<=S<73
D:	57<=S<64
F:	S<57

Midterm

- I scaled the grades by 24 points. You can think of it as:
  - Drop two short answer questions and half of a multiple choice question
  - Drop three multiple choice questions
- To determine your grade, look at the number on your exam and add 24