Physics 364, Fall 2014, reading due 2014-11-23.

Email your answers to `ashmansk@hep.upenn.edu` by 11pm on Sunday

Course materials and schedule are at `positron.hep.upenn.edu/p364`

Assignment: (a) First read this week's notes, starting on the next page. (b) Then email me your answers to the questions below.

1. If I write the Verilog statement `wire [5:0] a = 37;` what are the resulting values of `a[5]`, `a[4]`, `a[3]`, `a[2]`, `a[1]`, and `a[0]` ?

2. If I write the Verilog statement `wire [5:0] a = ((2+2==5) ? 37 : 13);` what is the resulting value of `a` ?

3. What does this Verilog module do?

```
module mystery (output o, input a, input b);
  assign o = !(a && b);
endmodule
```

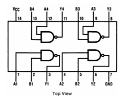
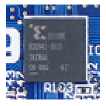
4. Is there anything from this reading assignment that you found confusing and would like me to try to clarify? If you didn't find anything confusing, what topic did you find most interesting?

5. How much time did it take you to complete this assignment?

In Lab 24, we will start working with FPGAs (Field Programable Gate Arrays), which in our case are manufactured by a company called Xilinx. FPGAs will let us wire up large numbers of logic gates, flip flops, etc., without having to run physical wires on a breadboard. (I think that after wiring up the four-bit adder in Lab 23, you can appreciate how much effort this saves!) The "wires" are all internal to the FPGA chip itself. There are two ways to tell the Xilinx software what you want your FPGA to do: the first way is to draw a schematic diagram by pointing and clicking on your computer screen; the second way is to write programs, in a language called **Verilog**, that represent the desired logic. In these notes and in the upcoming labs, you will see some short Verilog programs that demonstrate a few of the things you can do with logic gates, flip-flops, adders, counters, etc. In many cases, I will show you the schematic diagram that corresponds to the Verilog program, so that you can see that a Verilog program is really just an alternative way of representing a schematic diagram for digital logic.

By the way, remember that we will not have class on the Wednesday before Thanksgiving. But there will be another set of my notes for you to read over the Thanksgiving weekend, to serve as background material for Labs 25 and 26. Also, my plan is to give you a week to work on the take-home final exam, e.g. to put it online Dec 8 or 9 and to have it due on Dec 15 or so.

One key goal of the next four labs will be to try to fill in as much as possible of the conceptual gap between basic logic gates (things you know how to build out of Field Effect Transistors) and a simplified computer (vaguely along the lines of an Arduino). It quickly becomes impractical to wire up anything larger than about a dozen logic gates out of chips that look like this:

. So we're instead using Field Programmable Gate Arrays[1] (FPGAs) so that all of the little logic gates and wires can be connected for us automatically inside a little chip that looks like this: . To use an FPGA, you describe the logic that you want to implement, in a form that is isomorphic to a schematic diagram, and the FPGA compiler then figures out how to fit that logic into your chosen FPGA. While it is possible to provide this logic description by literally drawing a schematic diagram from within the FPGA software, hand-drawn wires can quickly become just as messy as wires on a breadboard, so it turns out to be much more convenient to describe your circuit in a text-based computer language called Verilog.[2]

Verilog has a C-like syntax, so if you are already familiar with other programming languages that use C-like syntax (the Arduino language, C, Java, Python, etc.), then much of the syntax will look natural to you. Strictly speaking, only a subset of

[1]http://en.wikipedia.org/wiki/Field-programmable_gate_array#Architecture
[2]http://en.wikipedia.org/wiki/Verilog

possible Verilog programs are isomorphic to schematic diagrams. We are using the **synthesizable** subset of the Verilog language, which is used to describe the logic gates, wires, etc. that you want the FPGA to implement in its little transistors. (The same synthesizable subset of Verilog is also used to design integrated circuits, e.g. if you want to design your own Pentium processor.) Another major use of the Verilog language is to make *simulation* programs that exercise the features of a hypothetical digital circuit before you manufacture it. (For example, you want to mimic the external behavior of all of the input/output pins of your hypothetical successor of the Pentium processor.) Verilog simulation programs can do things that go far beyond simply representing what is connected to what: they can read and write data files, print out warning messages, and make elaborate computations. In this course, I will only describe the features of the language that the FPGA compiler knows how to convert into logic gates, flip-flops, and their interconnecting wires. If in the future you use FPGAs in a large project, you will probably also need to learn how to use the simulation features of Verilog, so that you can debug your circuit before you load it into your real FPGA.

Before getting into the specifics of Verilog, let's spell out the FPGA hardware and software that we're using in the lab. Our FPGA manufacturer is **Xilinx, Inc.**.[3] Xilinx makes many different FPGA models. The family of Xilinx FPGA we are using is called **Spartan-3E**, and the specific model is **XC3S100E**. (Some families are newer than other families, have more bells and whistles to aid in complicated designs, operate faster, consume less power, etc. Specific models within a family vary mainly in size: the number of logic gates they can internally implement. The XC3S100E is the smallest member of the Spartan-3E family, containing the equivalent of about 100,000 logic gates. The largest Spartan-3E FPGAs contain nearly 2 million logic gates, and newer FPGA families from Xilinx contain even larger FPGAs.) The physical package in which our FPGA is housed is called **CP132**, which indicates 132 ball-shaped pins arranged in a rectangular lattice beneath the chip, to connect the chip to the printed circuit board. This FPGA comes in two different speed grades, called "-4" and "-5." Our chip is of **speed grade** $\mathbf{-4}$, which is the slower of the two grades. So again, the FPGA is **Spartan-3E / XC3S100E / CP132 /** $\mathbf{-4}$. The Spartan-3E family dates from about 2005 and is no longer used in new circuit designs, but the key ideas are the same as in newer FPGAs.

The FPGA is mounted on a printed circuit board (PCB) manufactured by **Digilent, Inc.**,[4] called the BASYS2 board.[5] The BASYS2 board makes the FPGA more usable by providing a USB connection to a PC to supply power and to permit new programs to be loaded, providing LEDs, switches, and push buttons, etc.

The software that compiles your Verilog design into the long sequence of ones and

[3] http://www.xilinx.com

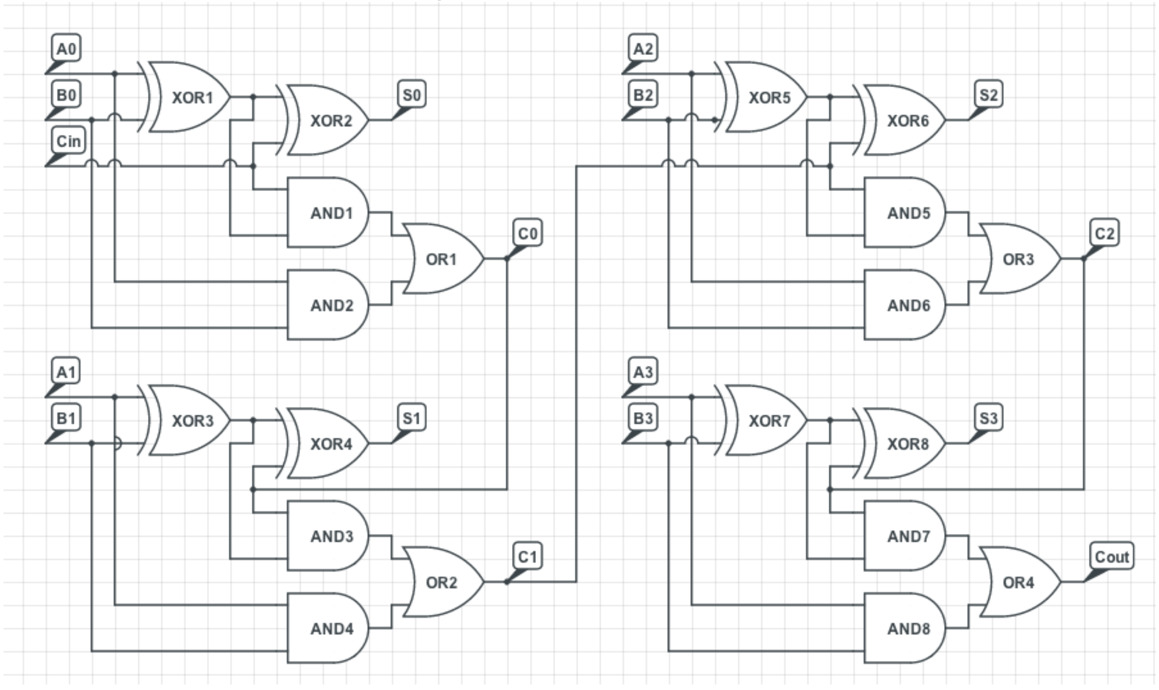
[4] http://digilentinc.com

[5] http://www.digilentinc.com/Products/Detail.cfm?Prod=BASYS2

zeros that tells the FPGA what to do is called **Xilinx ISE Project Navigator**, version 14.7, by Xilinx, Inc. The output of the Xilinx ISE software is a file named *projectName*`.bit` . The `*.bit` file is then programmed into your FPGA using a USB cable. This programming of the `*.bit` file into the FPGA can be done directly from the Xilinx ISE software or via Digilent-supplied program called **Digilent ADEPT**. The ADEPT software is easy to use and has a simpler user interface, so my preference is to use ADEPT to load the `*.bit` file into the board. In any case, keep in mind that we will need to open both the *Xilinx ISE Project Navigator* application and the *Digilent ADEPT* application to work with the BASYS2 board. Xilinx calls the `*.bit` file the *programming file.* So the operation to re-make the `*.bit` file in ISE is called *generate programming file.*

More specific instructions for using the ISE and ADEPT software can be found inside `lab24.pdf`, the instructions for Lab 24.

The basic unit of a Verilog program is a **module**. A module represents a type of component of your circuit. For example, you may want to describe a 4-bit adder in terms of logic gates. This description necessarily contains a list of input and output pins as well as the set of gates and wires that make up the adder's functionality. Here is the CircuitLab schematic diagram for the 4-bit adder that we have seen before:



The adder's inputs are A_3, A_2, A_1, A_0 (a 4-bit number), B_3, B_2, B_1, B_0 (another 4-bit number), and C_{in} (a carry bit). The adder's outputs are S_3, S_2, S_1, S_0 (a 4-bit sum: $S = A + B$) and C_{out} (a carry bit). We can represent this schematic diagram directly in Verilog like this:

```
module add4bit (output s3, output s2, output s1, output s0,
                output cout,
                input a3, input a2, input a1, input a0,
                input b3, input b2, input b1, input b0, input cin);
  // define wires for outputs of intermediate xor/and/or gates
  wire oxor1, oand1, oand2, c0;
  // instantiate gates for bit 0 of adder
  xor xor1 (oxor1, a0,    b0);
  xor xor2 (s0,    oxor1, cin);
  and and1 (oand1, cin,   oxor1);
  and and2 (oand2, a0,    b0);
  or  or1  (c0,    oand1, oand2);

  // define wires & instantiate gates for bit 1 of adder
  wire oxor3, oand3, oand4, c1;
  xor xor3 (oxor3, a1,    b1);
  xor xor4 (s1,    oxor3, c0);
  and and3 (oand3, c0,    oxor3);
  and and4 (oand4, a1,    b1);
  or  or2  (c1,    oand3, oand4);

  // define wires & instantiate gates for bit 2 of adder
  wire oxor5, oand5, oand6, c2;
  xor xor5 (oxor5, a2,    b2);
  xor xor6 (s2,    oxor5, c1);
  and and5 (oand5, c1,    oxor5);
  and and6 (oand6, a2,    b2);
  or  or3  (c2,    oand5, oand6);

  // define wires & instantiate gates for bit 3 of adder
  wire oxor7, oand7, oand8;
  xor xor7 (oxor7, a3,    b3);
  xor xor8 (s3,    oxor7, c2);
  and and7 (oand7, c2,    oxor7);
  and and8 (oand8, a3,    b3);
  or  or4  (cout,  oand7, oand8);
endmodule
```

On both the schematic diagram and in the Verilog program, every gate is given a name: the eight XOR gates are called `xor1` … `xor8`, etc. Notice also that we had to give every wire a name in Verilog, whereas on the schematic diagram most wires that represent intermediate results are left unnamed. For instance, I used `oxor1` as the name for the wire coming out of gate `xor1`. A wire in Verilog plays the same role as a wire on the schematic diagram: for instance, the wire `c0` connects the output of

OR gate `or1` to the second input of XOR gate `xor4` and to the first input of AND gate `AND3`.

The statement `wire oxor1, oand1, oand2, c0;` means, "Create wires whose names are `oxor1`, `oand1`, `oand2`, and `c0`. I will tell you later what is connected to them." The statement `xor xor3 (oxor3, a1, b1);` means, "Create an `xor` whose name is `xor3`; connect its output to the wire named `oxor3`; and connect its inputs to the wires named `a1` and `b1`." How does Verilog know what an `xor` is? It turns out that Verilog has built-in logic gates `and`, `or`, `xor`, `nand`, `nor`, `xnor`, each of which has one output (the first argument) and 2 or more inputs (the second, third, etc., arguments).

If we didn't want to use Verilog's built-in `xor`, `and`, and `or` gates, we could have defined our own modules e.g. called `xorgate`, `andgate`, and `orgate`, like this:

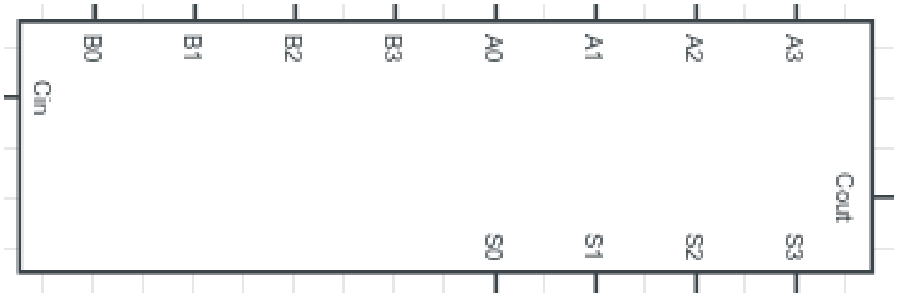
```
module andgate (output o, input a, input b);
  assign o = a & b;
endmodule;

module orgate (output o, input a, input b);
  assign o = a | b;
endmodule;

module xorgate (output o, input a, input b);
  assign o = a ^ b;
endmodule;
```

Then the lines for bit 0 of the adder would have looked like this instead:

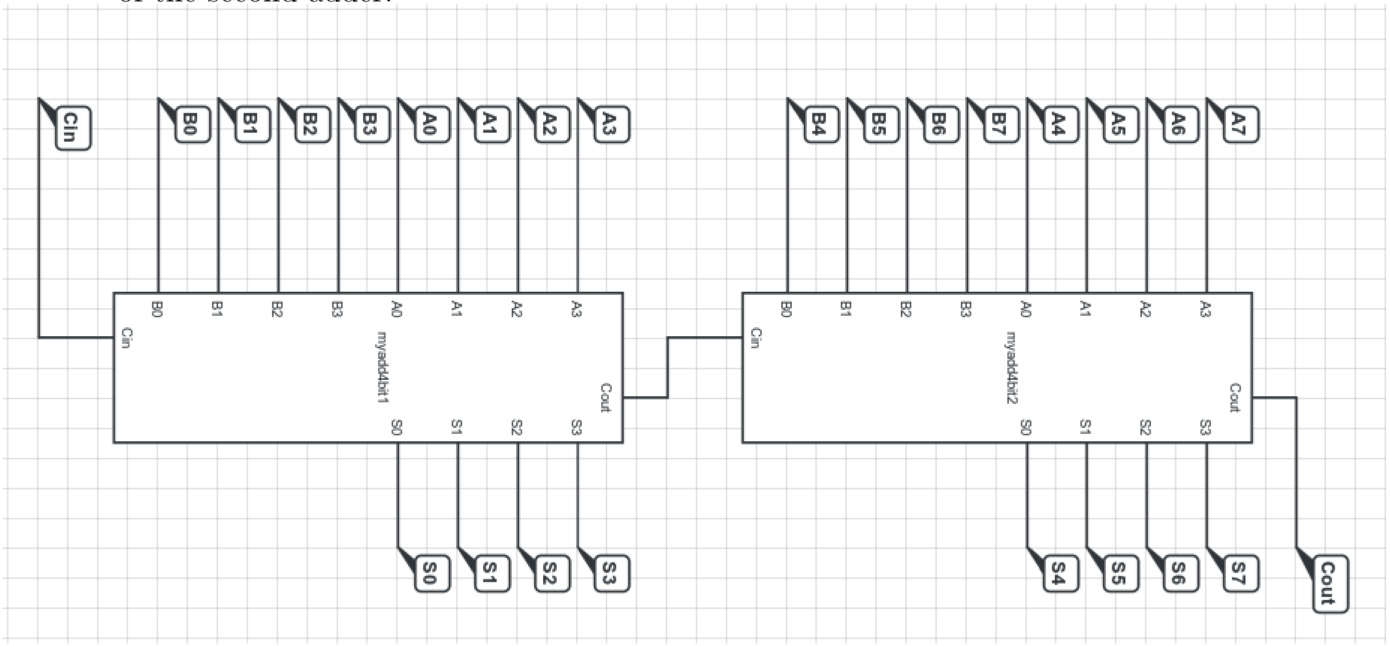
```
  // instantiate gates for bit 0 of adder
  xorgate xor1 (oxor1, a0,    b0);
  xorgate xor2 (s0,    oxor1, cin);
  andgate and1 (oand1, cin,   oxor1);
  andgate and2 (oand2, a0,    b0);
  orgate  or1  (c0,    oand1, oand2);
```

The lines from `module andgate` to `endmodule` provide Verilog with a definition of what I mean by an `andgate`. The line `andgate and1 (oand1, cin, oxor1);` then asks Verilog to create an `andgate` with a given name ("`and1`") and with given connections to output and input wires. (Its output should be connected to the wire called "`oand1`," and its inputs should be connected to the wires called "`cin`" and "`oxor1`.")

By analogy, the lines from `module add4bit` to `endmodule` up above defined what I mean by an `add4bit`. Now if I want to make an 8-bit adder, I can simply combine two 4-bit adders. Here is how I would do that on a schematic. First I would put my 4-bit adder into a little box like this (which is equivalent to the Verilog line `module add4bit (output s3, ..., input cin);`):



Then I would connect two instances of `add4bit` together to make a new module called `add8bit`, which implements an 8-bit adder. Shown below is the way I would draw that on a schematic diagram. There are inputs $A_7 \ldots A_0$, $B_7 \ldots B_0$, and C_{in}, and outputs $S_7 \ldots S_0$ and C_{out}. One of the instances of `add4bit` is named `myadd4bit1` and the other is named `myadd4bit2`. The C_{out} of the first adder is wired to the C_{in} of the second adder.



Shown below is the Verilog code that is isomorphic to the above schematic diagram. We create two instances of `add4bit`, which are named `myadd4bit1` and `myadd4bit2`. We connect the low 4 bits of A, B, and S to the first adder and the high 4 bits to the second adder. The wire named `carry` connects the `cout` output of the first adder to the `cin` input of the second adder.

```
module add8bit (output s7, output s6, output s5, output s4,
                output s3, output s2, output s1, output s0,
                output cout,
                input a7, input a6, input a5, input a4,
                input a3, input a2, input a1, input a0,
                input cin);
  wire carry;  // connects cout of 1st adder to cin of 2nd adder
  add4bit myadd4bit1 (s3, s2, s1, s0, carry,
                      a3, a2, a1, a0, b3, b2, b1, b0, cin);
  add4bit myadd4bit2 (s7, s6, s5, s4, cout,
                      a7, a6, a5, a4, b7, b6, b5, b4, carry);
endmodule
```

First we told Verilog how to make an `andgate`, an `orgate`, and an `xorgate`. Then we told Verilog how to *instantiate* and connect several instances of those three modules to make an `add4bit`. Then we told Verilog how to instantiate and connect two instances of `add4bit` modules to define a new module called `add8bit`. I hope you see that writing Verilog code in this way is analogous to drawing a schematic diagram.

If you had typed in by hand the above definition of the `add8bit` module, you might have asked yourself whether Verilog has a more concise way to write all of those separate bits of S, A, and B. And as a matter of fact, Verilog does have a concise notation for groups of many wires that travel together and are indexed by an integer. In Verilog, such a group is called a **vector**. (In electronics, a group of several wires that travel together is often called a **bus**. But in Verilog the name is *vector*. By the way, the thing that Verilog calls a *wire* is sometimes referred to as a *net* in Computer-Aided Design systems; but we'll follow Verilog and call it a *wire*.) We can replace the eight individual wires `s7`...`s0` with a single vector `s[7:0]`, like this:

```
module add8bit (output [7:0] s, output cout,
                input  [7:0] a, input  [7:0] b, input cin);
  wire carry;
  add4bit myadd4bit1 (s[3], s[2], s[1], s[0], carry,
                      a[3], a[2], a[1], a[0],
                      b[3], b[2], b[1], b[0], cin);
  add4bit myadd4bit2 (s[7], s[6], s[5], s[4], cout,
                      a[7], a[6], a[5], a[4],
                      b[7], b[6], b[5], b[4], carry);
endmodule
```

Now, if we rewrite the `add4bit` module to use vectors for its inputs and outputs, we can make `add8bit` look even nicer. Here is the rewritten `add4bit` module:

```
module add4bit (output [3:0] s, output cout,
                input  [3:0] a, input  [3:0] b, input cin);
    // define wires for outputs of intermediate xor/and/or gates
    wire oxor1, oand1, oand2, c0;
    // instantiate gates for bit 0 of adder
    xor xor1 (oxor1, a[0],  b[0]);
    xor xor2 (s[0],  oxor1, cin);
    and and1 (oand1, cin,   oxor1);
    and and2 (oand2, a[0],  b[0]);
    or  or1  (c0,    oand1, oand2);
    // define wires & instantiate gates for bit 1 of adder
    wire oxor3, oand3, oand4, c1;
    xor xor3 (oxor3, a[1],  b[1]);
    xor xor4 (s[1],  oxor3, c0);
    and and3 (oand3, c0,    oxor3);
    and and4 (oand4, a[1],  b[1]);
    or  or2  (c1,    oand3, oand4);
    // define wires & instantiate gates for bit 2 of adder
    wire oxor5, oand5, oand6, c2;
    xor xor5 (oxor5, a[2],  b[2]);
    xor xor6 (s[2],  oxor5, c1);
    and and5 (oand5, c1,    oxor5);
    and and6 (oand6, a[2],  b[2]);
    or  or3  (c2,    oand5, oand6);
    // define wires & instantiate gates for bit 3 of adder
    wire oxor7, oand7, oand8;
    xor xor7 (oxor7, a[3],  b[3]);
    xor xor8 (s[3],  oxor7, c2);
    and and7 (oand7, c2,    oxor7);
    and and8 (oand8, a[3],  b[3]);
    or  or4  (cout,  oand7, oand8);
 endmodule
```

Using this rewritten add4bit, here is the simplified add8bit:

```
module add8bit (output [7:0] s, output cout,
                input  [7:0] a, input  [7:0] b, input cin);
  wire carry;
  add4bit myadd4bit1 (s[3:0], carry, a[3:0], b[3:0], cin);
  add4bit myadd4bit2 (s[7:4], cout,  a[7:4], b[7:4], carry);
endmodule
```

The vector s is eight bits wide. Notice that I can select only the lower four bits of s by writing `s[3:0]`, and I can select only the upper four bits of `s` by writing `s[7:4]`. I can pick out a single bit (e.g. bit 3) of the vector `s` by writing `s[3]`. So the expression `s[3]` behaves like a wire, and the expression `s[3:0]` behaves like a 4-bit vector. The expression `s` (or equivalently the expression `s[7:0]`) behaves like an 8-bit vector.

Meanwhile, using the C-like operators for the AND (`&`), OR (`|`), and XOR (`^`) operations, we can simplify the `add4bit` module even further (though this so far isn't much of a simplification):

```
module add4bit (output [3:0] s, output cout,
                input  [3:0] a, input  [3:0] b, input cin);
    // define wires for outputs of intermediate xor/and/or gates
    wire oxor1, oand1, oand2, c0;
    // instantiate gates for bit 0 of adder
    assign oxor1 = a[0]  ^ b[0];
    assign s[0]  = oxor1 ^ cin;
    assign oand1 = cin   & oxor1;
    assign oand2 = a[0]  & b[0];
    assign c0    = oand1 | oand2;
    // define wires & instantiate gates for bit 1 of adder
    wire oxor3, oand3, oand4, c1;
    assign oxor3 = a[1]  ^ b[1];
    assign s[1]  = oxor3 ^ c0;
    assign oand3 = c0    & oxor3;
    assign oand4 = a[1]  & b[1];
    assign c1    = oand3 | oand4;
    // define wires & instantiate gates for bit 2 of adder
    wire oxor5, oand5, oand6, c2;
    assign oxor5 = a[2]  ^ b[2];
    assign s[2]  = oxor5 ^ c1;
    assign oand5 = c1    & oxor5;
    assign oand6 = a[2]  & b[2];
    assign c2    = oand5 | oand6;
    // define wires & instantiate gates for bit 3 of adder
    wire oxor7, oand7, oand8;
    assign oxor7 = a[3]  ^ b[3];
    assign s[3]  = oxor7 ^ c2;
    assign oand7 = c2    & oxor7;
    assign oand8 = a[3]  & b[3];
    assign cout  = oand7 | oand8;
 endmodule
```

We introduced two new Verilog features in this last step. First, we used the C-like operators `&`, `|`, and `^` for the AND, OR, and XOR operations. Second, we used the

Verilog `assign` statement to connect a wire (on the left-hand side of the `=` sign) to an expression (on the right-hand side). Now here's one more Verilog feature: we can combine the `wire` declaration and the `assign` statement into a single statement:

```
module add4bit (output [3:0] s, output cout,
                input  [3:0] a, input  [3:0] b, input cin);
    // instantiate gates for bit 0 of adder
    wire   oxor1 = a[0]  ^ b[0];
    assign s[0]  = oxor1 ^ cin;
    wire   oand1 = cin   & oxor1;
    wire   oand2 = a[0]  & b[0];
    wire   c0    = oand1 | oand2;
    // define wires & instantiate gates for bit 1 of adder
    wire   oxor3 = a[1]  ^ b[1];
    assign s[1]  = oxor3 ^ c0;
    wire   oand3 = c0    & oxor3;
    wire   oand4 = a[1]  & b[1];
    wire   c1    = oand3 | oand4;
    // define wires & instantiate gates for bit 2 of adder
    wire   oxor5 = a[2]  ^ b[2];
    assign s[2]  = oxor5 ^ c1;
    wire   oand5 = c1    & oxor5;
    wire   oand6 = a[2]  & b[2];
    wire   c2    = oand5 | oand6;
    // define wires & instantiate gates for bit 3 of adder
    wire   oxor7 = a[3]  ^ b[3];
    assign s[3]  = oxor7 ^ c2;
    wire   oand7 = c2    & oxor7;
    wire   oand8 = a[3]  & b[3];
    assign cout  = oand7 | oand8;
 endmodule
```

If we use the C-like logical operators, we can avoid creating so many intermediate wire names. So we can rewrite the `add4bit` module in this much shorter form:

```
module add4bit (output [3:0] s, output cout,
                input  [3:0] a, input  [3:0] b, input cin);
    assign s[0] =  a[0] ^ b[0] ^ cin;
    wire   c0   = (a[0] & b[0]) | (a[0] & cin) | (b[0] & cin);
    assign s[1] =  a[1] ^ b[1] ^ c0;
    wire   c1   = (a[1] & b[1]) | (a[1] & c0)  | (b[1] & c0);
    assign s[2] =  a[2] ^ b[2] ^ c1;
    wire   c2   = (a[2] & b[2]) | (a[2] & c1)  | (b[2] & c1);
    assign s[3] =  a[3] ^ b[3] ^ c2;
    assign cout = (a[3] & b[3]) | (a[3] & c2)  | (b[3] & c2);
 endmodule
```

It seems a bit annoying to have to write such similar equations four separate times. Maybe there is a way to combine them, taking advantage of the fact that `s`, `a`, and `b` are Verilog vectors? In fact there is, using two tricks. The first trick is the Verilog **concatenation operator**, which is expressed using `{}` curly braces. If you stick n wires side-by-side inside a set of curly braces, the resulting expression behaves like an n-bit vector. You can also concatenate a mixture of wires and vectors, with the resulting expression having a width (number of bits) that is the sum of the widths of the individual wires and vectors. So if I have wires named `c2`, `c1`, `c0`, `cin`, then the expression `{c2,c1,c0,cin}` behaves like a 4-bit vector. In other words, the expression `{s[3],s[2],s[1],s[0]}` behaves like the expression `s[3:0]`. The second trick is the fact that you can apply the C-like operators `&`, `|`, and `^` to vectors, with the result that the AND, OR, or XOR operation is performed bit-by-bit on each vector. So a statement like `assign o[3:0] = a[3:0] & b[3:0];` has exactly the same effect as the four separate statements

```
  assign o[3] = a[3] & b[3];   assign o[2] = a[2] & b[2];
  assign o[1] = a[1] & b[1];   assign o[0] = a[0] & b[0];
```

So we can make the `add4bit` module even more concise by using the power of Verilog bit-vector operations. (Make sure you followed that last paragraph, by carefully studying this next example.)

```
module add4bit (output [3:0] s, output cout,
                input  [3:0] a, input  [3:0] b, input cin);
    wire c0, c1, c2;
    wire [3:0] k = {c2, c1, c0, cin};
    assign s[3:0] = a[3:0] ^ b[3:0] ^ k[3:0];
    assign {cout,k[3:1]} =
      (a[3:0] & b[3:0]) | (a[3:0] & k[3:0]) | (b[3:0] & k[3:0]);
 endmodule
```

The last statement above is equivalent to the four statements (in any order):

```
  assign k[1] = (a[0] & b[0]) | (a[0] & k[0]) | b[0] & k[0]);
  assign k[2] = (a[1] & b[1]) | (a[1] & k[1]) | b[1] & k[1]);
  assign k[3] = (a[2] & b[2]) | (a[2] & k[2]) | b[2] & k[2]);
  assign cout = (a[3] & b[3]) | (a[3] & k[3]) | b[3] & k[3]);
```

Notice that `k[3]` is the same as `c2`, etc.

We could be even more concise by noticing that since the vector `a` is only 4 bits wide, we can simply write `a` instead of going to the trouble of writing `a[3:0]`. This is certainly more concise, but it is up to you to decide whether it is more or less clear. Sometimes it is helpful to make it obvious that you are manipulating a bit vector, rather than a wire, by writing the longer expression `a[3:0]` when you could just write `a`. In any case, here is the more concise version:

```
module add4bit (output [3:0] s, output cout,
                input  [3:0] a, input  [3:0] b, input cin);
    wire c0, c1, c2;
    wire [3:0] k = {c2,c1,c0,cin};
    assign s = a ^ b ^ k;
    assign {cout,k[3:1]} = (a & b) | (a & k) | (b & k);
endmodule
```

Now let me show you something that almost looks like cheating. It turns out that Verilog knows how to do addition: it understands how to convert the addition of two integers (represented as bit vectors) into the corresponding collection of logic gates. So in fact there was no need (except for pedagogical reasons) for us to spell out the module `add4bit` with individual logical operations. We can in fact re-write the `add4bit` module in this unbelievably simple form:

```
module add4bit (output [3:0] s, output cout,
                input  [3:0] a, input  [3:0] b, input cin);
    assign {cout,s} = a + b + cin;
endmodule
```

Each of the expressions `a`, `b`, and `s` is four bits wide. Each of the expressions `cin` and `cout` is only a single bit wide. The expression `{cout,s}` is five bits wide. The largest possible value for `a + b + cin` is (in decimal) $15 + 15 + 1 = 31$, which just fits into five bits. Keep in mind that when the Xilinx Verilog compiler sees this last version of `add4bit`, it will be smart enough to expand it into the equivalent combination of logic gates, as we had previously done by hand.

Here's another bit-vector trick that you might find helpful: you can assign an integer constant to a bit vector. And that constant can be expressed in decimal (base 10), in binary (base 2), or in hexadecimal (base 16). So I can write (equivalently)

```
  wire [7:0] anote = 220;
  wire [7:0] anote = 'b11011100;
  wire [7:0] anote = 'hdc;
```

and the result in all three cases is that bits 7, 6, 4, 3, and 2 of `anote` will be HIGH and bits 5, 1, and 0 of `anote` will be LOW. That's $220_{10} = 11011100_2 = \mathrm{DC}_{16}$. (I called it `anote` for the frequency in Hz of the musical A note below middle C.)

In many programming languages (in Verilog, as well as in C, Python, Java, etc.), there are two forms of the AND, OR, NOT, etc., operations: one version ("bitwise") that operates bit-by-bit, treating each bit separately as TRUE (1) or FALSE (0), and another version ("logical") that treats an entire expresssion as TRUE (nonzero) or FALSE (zero). For example, the **bit-wise** NOT operator is `~` , while the **logical** NOT operator is `!` . For the logical operators, any non-zero expression is considered TRUE, while a zero is considered FALSE. The result of a logical operator is a single bit: 1 for TRUE and 0 for FALSE. This is probably most easily illustrated with a table. Suppose that `A` and `B` are both 4-bit vectors. For example,

```
  wire [3:0] A; wire [3:0] B;
```

Let's look at the result of applying various bitwise and logical operators to `A` and `B`, and putting the result into another 4-bit vector `C`. For instance, the `A&B` column is the result of `wire [3:0] C = A&B;` . The operators are `~` (bit-wise NOT), `!` (logical NOT), `&` (bit-wise AND), `&&` (logical AND), `|` (bit-wise OR), `||` (logical OR), and `^` (bit-wise XOR). (There is no logical XOR operator.) All of these operators work the same way in Verilog as in Arduino, C, Python, Java, Processing, etc.

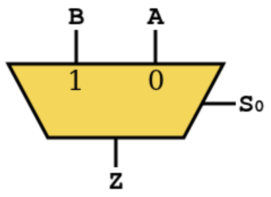
A	B	~B	!B	A&B	A&&B	A|B	A||B	A^B
0	0	15	1	0	0	0	0	0
0	1	14	0	0	0	1	1	1
0	2	13	0	0	0	2	1	2
1	1	14	0	1	1	1	1	0
1	2	13	0	0	1	3	1	3
2	2	13	0	2	1	2	1	0

Here is the same table in binary. (The previous table was in decimal.)

A	B	~B	!B	A&B	A&&B	A|B	A||B	A^B
0000	0000	1111	0001	0000	0000	0000	0000	0000
0000	0001	1110	0000	0000	0000	0001	0001	0001
0000	0010	1101	0000	0000	0000	0010	0001	0010
0001	0001	1110	0000	0001	0001	0001	0001	0000
0001	0010	1101	0000	0000	0001	0011	0001	0011
0010	0010	1101	0000	0010	0001	0010	0001	0000

Also, the expression `A==B` evaluates to 1 if `A` and `B` are **equal**; otherwise, it evaluates to 0. The expresssion `A!=B` evaluates to 1 if `A` and `B` are **not equal**; otherwise, it evaluates to 0. A mysterious operator called the **ternary operator** (because it has three operands) is remarkably useful in Verilog (and in C and Java, but not in Python). The expression `Q ? B : A` evaluates to `B` if `Q` is nonzero and evaluates to `A` if `Q` is zero. This allows you to select between two different values, depending on whether or not some condition `Q` is satisfied. We will see examples of this in several of the upcoming labs. The logic-gate equivalent of the ternary operator is called a **multiplexer**: it uses a `select` signal (called S_0 in the diagram below) to choose which of two inputs (called A and B in the diagram) is assigned to the output (called Z in the diagram). The diagram below performs the same function as the following Verilog module:

```
module multiplexer (output Z, input S, input A, input B);
  assign Z = (S ? B : A);
endmodule
```



There was a brief description of multiplexers in Chapter 8 of Eggleston's book. There is also a Wikipedia description at
http://en.wikipedia.org/wiki/Multiplexer#Digital_multiplexers
which I think is worth reading, because we will use digital multiplexers many times in the FPGA labs, as a way of selecting among several alternative inputs.

I think that's enough Verilog for you to digest in one sitting. I hope that it helps you to make more sense of Lab 24. I plan to give you some places in the FPGA labs in which you need to fill in a few lines of your own Verilog code within a mostly-finished skeleton that I provide.