Lecture 15: Multiplexor and binary addition circuits

This lecture describes multiplexor and binary additon circuits. Hardware design shares with programming the technique of abstracting common tasks into reusable modules, separating the implementation of the module from its use. In some other ways, it is quite different, since in hardware everything is "trying to happen at once" and the designer's effort is to control and orchestrate the activity, while in (sequential) software, there is a single focus of activity, which the programmer is attempting to direct in useful and economical ways. Though not exact, one possible comparison is football to golf.

Multiplexors

The physical devices implementing wires and gates are reused for different parts of one computation, something like highways carrying different flows of traffic during the day. One of the circuits that helps control the flow of data through the devices is a multiplexor. In the simplest case, we have one wire that should carry data from one source at some times, and from another source at different times, under the control of a control input that specifies at any time which source should be used.

If we call the two data inputs x and y, the control input s, and the output z, then we would like this circuit to have the behavior that if $s = 0$, then $z = x$, and if $s = 1$ then $z = y$. Constructing a truth table for this function we have the following.

```
 s  x  y  |  z
--------------
 0  0  0  |  0
 0  0  1  |  0
 0  1  0  |  1
 0  1  1  |  1
 1  0  0  |  0
 1  0  1  |  1
 1  1  0  |  0
 1  1  1  |  1
```

If we apply our sum-of-products algorithm to this truth table, we get the Boolean expression

$$z = s'xy' + s'xy + sx'y + sxy.$$

This can be simplified using the laws of Boolean algebra as follows.

$$z = s'x(y' + y) + sy(x' + x) = s'x + sy.$$

This is intuitively correct: the output should be 1 precisely when $s = 0$ and $x = 1$ or when $s = 1$ and $y = 1$. We can implement this as a circuit using 4 gates as follows.

```
s----*-|NOT>--|
     |        |AND>---|
x----|--------|       ----|
     |                    |OR>----- z
     |--------|       ----|
              |AND>---|
y-------------|
```

Suppose instead we wanted to have the output carry the data from one of four data sources, depending on control inputs. We need 2 control inputs, to specify one of four possibilities. This gives a circuit with four data inputs, say x_0, x_1, x_2, x_3, and two control inputs, say s_0 and s_1. In this case, we want the output z to have the property that $z = x_0$ if $s_0 = 0$ and $s_1 = 0$, and $z = x_1$ if $s_0 = 0$ and $s_1 = 1$, and $z = x_2$ if $s_0 = 1$ and $s_1 = 0$, and $z = x_3$ if $s_0 = 1$ and $s_1 = 1$. A truth table for a function with 6 inputs has $2^6 = 64$ lines, so we choose not to use the method of truth table and sum-of-products algorithm to find an expression for this function. Instead, we generalize the expression above, and obtain

$$x_0 s_0' s_1' + x_1 s_0' s_1 + x_2 s_0 s_1' + x_3 s_0 s_1.$$

For example, note that if we substitute $s_0 = 1$ and $s_1 = 0$ into this expression and simplify, we get exactly x_2. If we use this expression directly to design a circuit, we get a multiplexor for 4 data and 2 control inputs with 2 NOT gates, 8 AND gates, and 3 OR gates. Generalizing this construction, we could design a multiplexor with 2^n data inputs and n control inputs, for any positive integer n.

A binary addition circuit

In this part, we consider the problem of building a circuit to add two 4-bit binary numbers. Here is an example addition:

```
   0 0 1 1
+  0 1 1 1
----------
```

Starting from the rightmost bits, we add 1 and 1 to get 2, which is 10 in binary, so we put down the digit 0 and show the carry of a 1 into the next column:

```
       1
   0 0 1 1
+  0 1 1 1
----------
         0
```

Now we add up the three 1's, getting 3, which is 11 in binary, so we put down the digit 1 and show the carry of a 1 into the next column:

```
     1 1
   0 0 1 1
+  0 1 1 1
----------
       1 0
```

Continuing in this way, we finally get:

```
   1 1 1
   0 0 1 1
+  0 1 1 1
----------
   1 0 1 0
```

Converting back into decimal to check, we have $3 + 7 = 10$.

To construct a circuit, we focus on one column at a time. The rightmost column has two inputs, say x and y, and the rightmost result bit, say z is determined by the truth table:

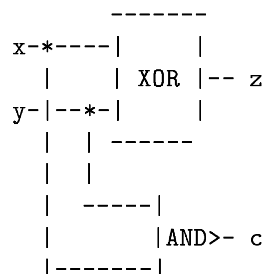
x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

We observe that this is just the exclusive-or function. We'll assume we have gates available to compute exclusive-or, abbreviated XOR. If XOR gates are not available, we can construct one by using the sum-of-products method to construct a boolean expression:

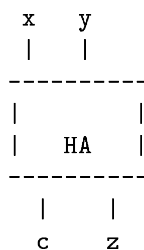
$$xy' + x'y,$$

which can be converted into a circuit with 2 NOT gates, 2 AND gates, and an OR gate.

The other thing we need to compute for the rightmost column is whether the carry into the next column is 1 or 0. The carry is 1 exactly in the case that x and y are both 1. Letting c denote the carry output, we have $c = xy$, implementable with just one AND gate. The final circuit for the rightmost column is:



Abstracting this to a box with inputs x and y and outputs z and c, we get the half-adder:



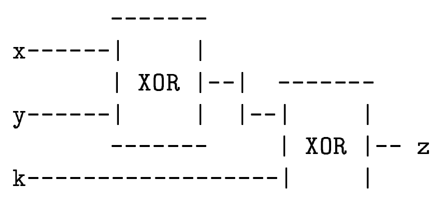
What about the next column? In the next column, we have the possibility of a carry-in to the addition. The function we consider has 3 inputs: x, y, and k The value of the sum bit is determined by the following table:

x	y	k	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

This function can be realized as a sum-of-products:

$$x'y'k + x'yk' + xy'k' + xyk$$

and the corresponding circuit. Or, we can note that two XOR gates will give the correct output:



```
        -------
x------|       |
       | XOR |--|   -------
y------|       |  |--|       |
        -------      | XOR |-- z
k------------------|       |
                     -------
```

This works because combining three inputs with XOR returns the parity of the sum of the three inputs: 1 if the sum is odd (1 or 3) and 0 if the sum is even (0 or 2).

The other output that we have to generate for this column is whether there is a carry into the next column. The truth-table for this function is:

x	y	k	c
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

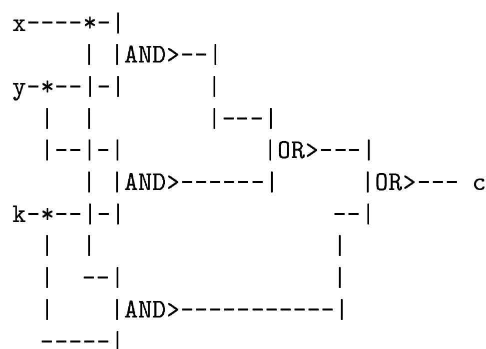
Note that there is a carry out when two or more of the inputs are 1. The sum-of-products algorithm gives the expression:

$$x'yk + xy'k + xyk' + xyk.$$

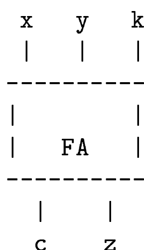
A simpler expression for the same function is:

$$yk + xk + xy.$$

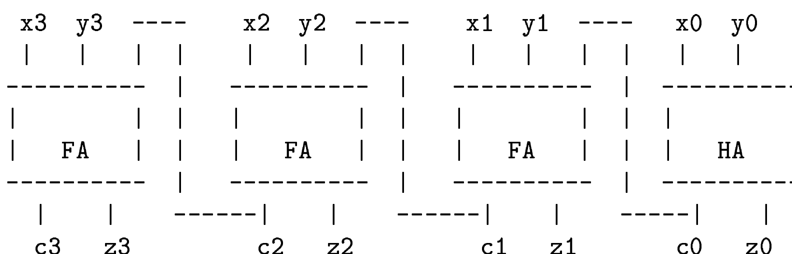
Implementing this as a circuit, we get:



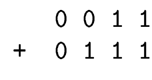
(Yes, we'll all be happier when I don't draw ASCII circuits!) Putting these last two circuits together into a box with inputs x, y, and k, and outputs z and c, we get a full-adder, which we'll symbolize thus:



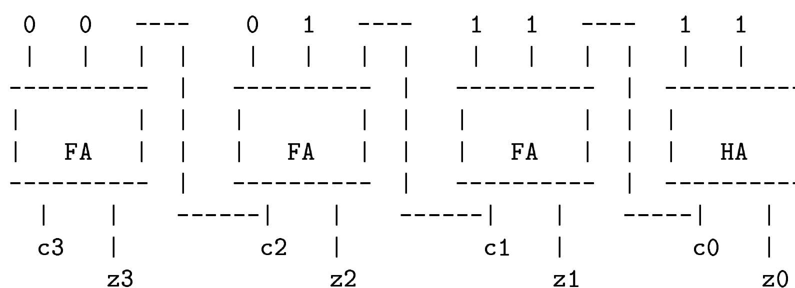
The full adder is what we need to compute the rest of the columns of the addition. Our four-bit addition can be computed by the following circuit connecting the carry-out from each addition to the carry-in of the next addition as shown below.



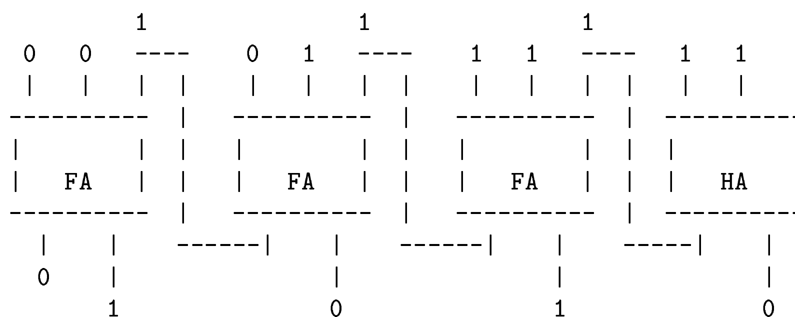
Referring back to the addition problem that we started with:



x0 and y0 are the two bits in the rightmost column, x1 and y1 are the two bits in the column to its left, and so on. Thus, we have inputs as shown below.



The outputs z0 = 0 and c0 = 1 are computed, and the value of c0 is an input to the next circuit to the left, and so on, giving:



The resulting bits z3, z2, z1, z0 are 1010, as desired. Note that the last carry out bit (0 in this example) could be used to tell whether the sum is correctly expressed by the four result bits.

This circuit could be generalized to handle any number of bits. This is a ripple-carry adder, named for the way the carry "ripples" from the low-order to the high order bit. Note that for n bits, there will be about $3n$ gate delays before all the output bits can be assumed to have properly settled down. There are alternative designs for circuits to add two n-bit numbers that involve a gate delay proportional to $\log n$ rather than n, which are used in actual machines.