Lecture 11: Circuits for logical and arithmetic operations

This lecture describes circuits to compute some of the operations that are required in the arithmetic-logical unit (ALU) of a computer.

Compare-for-equality

Suppose we have two 4-bit quantities and we want a circuit that outputs 1 if they are equal, 0 if they are different. Pictorially:

```
 x3  x2  x1  x0  y3  y2  y1  y0
 |   |   |   |   |   |   |   |
-------------------------------
|                             |
-------------------------------
               |
               z
```

Each input wire, x_i or y_i, is either a 0 or a 1, representing one bit of the input x or y, respectively. The output wire z should have a 1 if the corresponding bits of x and y are all equal, that is, $x_0 = y_0$ and $x_1 = y_1$ and $x_2 = y_2$, and $x_3 = y_3$.

As an intermediate step, we'll design a circuit that has two inputs, x and y, and gives an output of 1 if the two input bits are equal, and 0 if they are not equal. This function we call 1-CE, for 1-bit compare-for-equality. Its truth table is:

x	y	1-CE
0	0	1
0	1	0
1	0	0
1	1	1

If we directly apply the sum-of-products algorithm to this truth table, we get the boolean expression;

$$x'y' + xy,$$

which translates into the combinational circuit:

```
x-*---|NOT>---|
  |           |AND>---|
y-|-*-|NOT>---|       |-----|
  | |                       |OR>-------- z = 1-CE(x,y)
  |-|---------|       |-----|
    |         |AND>---|
    |---------|
```

Thus, with 5 gates we can implement the 1-bit compare-for-equality function, 1-CE. Now we abstract this circuit as a box that we may use in other circuits. This is analogous to defining and using a procedure in a program.

If we compare each pair of bits x_i and y_i using a 1-CE circuit, then if all of them return 1, the 4-bit compare-for-equality circuit should return 1. If any of them return 0, the 4-bit compare-for-equality circuit should return 0. Thus, if we AND together the results of the four 1-bit comparisons, we'll have the correct overall answer:

```
     --------
x0---|      |
     | 1-CE |----|
y0---|      |    |AND>---|
     --------  |-|       |AND>----|
               |       |-|        |AND>---- z = 4-bit CE
               |       |        |-|
     --------  |       |        |
x1---|      |  |       |        |
     | 1-CE |--|       |        |
y1---|      |          |        |
     --------          |        |
                       |        |
                       |        |
     --------          |        |
x2---|      |          |        |
     | 1-CE |----------|        |
y2---|      |                   |
     --------                   |
                                |
                                |
     --------                   |
x3---|      |                   |
     | 1-CE |-------------------|
y3---|      |
     --------
```

Note that for clarity, I've rearranged the inputs so that the pairs to be compared are adjacent.

The AND gates could be combined in a different order, for example:

```
a --|
    |AND>---|
b --|       |---|
                |AND>---
c --|       |---|
    |AND>---|
d --|
```

This corresponds to the expression $((a \cdot b) \cdot (c \cdot d))$ rather than $(((a \cdot b) \cdot c) \cdot d)$. The advantage of this form is that it can compute its result faster. In a physical realization of a gate (as transistors or relays or optical elements), there is some small time delay between the time that the input signals settle down and the time the output signal settles down. An attempt to use the output signal before it has settled could lead to an error. The time is a gate delay, and depends on the implementation of the gate and its type. When the output of one gate is an input to another gate, then the output of the first gate must settle down, and only after that can the output of the second gate begin to settle down. Thus, at least two gate delays are required for a reliable output in this case.

The AND structure in the diagram for the 4-bit CE requires 3 gate delays, whereas the AND structure above only requires 2 gate delays. In general, by arranging the AND's in a binary tree, we can compute the

AND of n inputs with $\lceil \log n \rceil$ gate delays, as opposed to $n-1$, an exponential improvement. Because the output of an n-bit AND depends on all the inputs, and we assume we have only 2-input AND gates, there is also a lower bound of $\lceil \log n \rceil$ on the depth of a circuit to compute an n-bit AND. Thus, the tree-structured AND-circuit is optimal under these assumptions.

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:

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

Continuing in this way, we finally get:

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

Converting back into decimal to check, we have $11 + 7 = 18$.

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:

```
       -------
x-*----|     |
  |    | XOR |-- z
y-|--*-|     |
  |  | ------
  |  |
  |  -----|
  |       |AND>- c
  |-------|
```

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

```
   x   y
   |   |
  ----------
  |        |
  |   HA   |
  ----------
    |    |
    c    z
```

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 b The value of the sum bit is determined by the following table:

x	y	b	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:

$$x'y'b + x'yb' + xy'b' + xyb$$

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

```
        -------
 x------|     |
        | XOR |--|  -------
 y------|     |  |--|     |
        -------     | XOR |-- z
 b------------------|     |
                    -------
```

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	b	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:

$$x'yb + xy'b + xyb' + xyb.$$

A simpler expression for the same function is:

$$yb + xb + xy.$$

Implementing this as a circuit, we get:

```
 x----*-|
      | |AND>--|
 y-*--|-|      |
   |  |        |---|
   |--|-|          |OR>---|
      | |AND>------|      |OR>--- c
 b-*--|-|               --|
   |  |                 |
   |  --|               |
   |    |AND>-----------|
   -----|
```

(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 b, and outputs z and c, we get a full-adder, which we'll symbolize thus:

```
    x   y   b
    |   |   |
   ----------
   |        |
   |   FA   |
   ----------
     |    |
     c    z
```

The full adder is what we need to compute the rest of the columns of the addition. So, 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:

```
    x3  y3  ----    x2  y2  ----    x1  y1  ----   x0  y0
    |   |   |  |    |   |   |  |    |   |   |  |   |   |
   ----------  |   ----------  |   ----------  |  ----------
   |        |  |   |        |  |   |        |  |  |        |
   |   FA   |  |   |   FA   |  |   |   FA   |  |  |   HA   |
   ----------  |   ----------  |   ----------  |  ----------
     |    |    ------|    |    ------|    |    -----|     |
     c3   z3         c2   z2         c1   z1        c0   z0
```

Referring back to the addition problem that we started with:

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

`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:

```
    1   0   ----    0   1   ----    1   1   ----   1   1
    |   |   |  |    |   |   |  |    |   |   |  |   |   |
   ----------  |   ----------  |   ----------  |  ----------
   |        |  |   |        |  |   |        |  |  |        |
   |   FA   |  |   |   FA   |  |   |   FA   |  |  |   HA   |
   ----------  |   ----------  |   ----------  |  ----------
     |    |    ------|    |    ------|    |    -----|     |
     c3   |          c2   |          c1   |         c0    |
          z3              z2              z1             z0
```

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:

```
         1               1               1
 1   0   ----    0   1   ----    1   1   ----   1   1
 |   |   |  |    |   |   |  |    |   |   |  |   |   |
----------  |   ----------  |   ----------  |  ----------
|        |  |   |        |  |   |        |  |  |        |
|   FA   |  |   |   FA   |  |   |   FA   |  |  |   HA   |
----------  |   ----------  |   ----------  |  ----------
  |    |    ------|    |    ------|    |    -----|    |
  1    |               |               |              |
       0               0               1              0
```

Including the carry out of the leftmost bit, the result bits are 10010, as desired.

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.