CS 237 Meeting 8 — 9/24/12

Announcements

1. None?

Quick Review

1. Last time, we started out by exploring the connection between electrical switches and the logical operations and, or, and not.
2. We saw (vaguely) that a transistor is basically a switch that is controlled electronically rather than mechanically.
3. We say how circuits for nor, not and or in which ground voltage represented 0 and some positive voltage represented 0 could be constructed from transistors.
   - We proudly abstracted away many details discussed in the text. In particular, we pretended 0 and 1 were represented by exact voltage levels while in reality each is represented by some range of voltages.
4. We saw a "practical" example of how such logic gates could be used to construct a circuit that might drive on segment of a 7-segment display.
5. We noted that many functions performed within computers can be seen as collections of functions that map a group of input signals into the bits of a multi-bit result.
6. We began discussing truth tables ...

Truth Tables and Finiteness

1. In our discussions, we will use three "notations" for describing boolean functions. You have already seen two:
   - Circuits built from gates like the 7-segment display example,
   - Truth tables like the ones we showed to summarize what the circuits using switches and transistors actually did.
2. Generally, when we want to describe a boolean function, we will give variable names to its inputs.
   - For example, it would be natural to call the 4 digits input to our 7-segment display example as d_3, d_2, d_1, and d_0.
3. To describe a function as a truth-table, we create one column for each input/variable and one column for the output.
4. We want to put every possible combination of values of the input variables in the variable columns. If there are n variables, there will be 2^n combinations so our table will need 2^n rows.
5. We fill the variable columns by counting from 0 to 2^n in binary.
6. After we fill the variable columns, will fill in the output.
7. To illustrate this construction, consider the relational operator $>$ as a boolean function. That is, in any real computer $>$ would take two binary numbers of some fixed size n and return a single 0 or 1 indicating whether the first number was less than the second.
8. To keep the table size down, we will work with very short binary numbers — two bits each.
9. How many possible different output columns are there?

$$2^{2^n}$$

10. This is important! For any given number of inputs there are only a finite number of different boolean functions.
11. For a single input, there are 4 functions: negation, identity, constant 0, and constant 1.
12. For two inputs there are 16 functions: AND, OR, NOR, NAND, ???
    - Let's try to name them all.
      - The two most familiar are and and or.
      - Last time we met nand and nor.

- There is one other popular function: exclusive or. It outputs one when exactly one of the inputs is 0.
- Note that or, and, nand, nor, and exclusive or can all be naturally extended to many inputs.
- There are the constant functions that always return 0 and 1.
- There are the functions that ignore one input and simply return the other input or its negation.
- To find the others, it helps to form a truth table and to start filling in the four output cells by just counting from 0 to 15 in binary.

Boolean formulas

1. As an alternative to gate diagrams or truth tables, we can describe a boolean function using a formula over the input variables using boolean operators.

   The most commonly used operators in such formulas are:

   AND written as multiplication by simply putting two variables or terms next to one another,

   OR written as addition, and

   NOT written by drawing a bar over a variable or term.

2. For example, while earlier we said the top segment should be off if the 1st **and** 3rd bits of the numbers encoding are 0, **and** either 2nd **or** 4th bit is 0, **and** either 2nd **or** 4th bit is 1, we might now write:

$$(\bar{d_1}\bar{d_3})(\bar{d_2}+\bar{d_4})(d_2+d_4)$$

Disjunctive Normal (i.e. Sum of Products) Form

1. There is a simple procedure for converting any truth table into a formula using only AND, OR, and NOT that describes the same boolean function as the truth table.

2. The trick is to write a term that describes each combination of inputs that corresponds to a 1 row in the truth table using only variables, their negations and the and operator.
   - Such a term is called a *minterm* (presumably because it is true for as few combinations of the input variables as possible (just 1!)).

3. Then, write a formula that takes the or of all of the minterms.

4. This way of describing a function is called *disjunctive normal form* (DNF) or sum-of-products form.

5. To illustrate this, we can construct a DNF formula for the comparison function described earlier.

$$\bar{x_1}x_0\bar{y_1}\bar{y_0} + x_1\bar{x_0}\bar{y_1}\bar{y_0} + x_1\bar{x_0}\bar{y_1}y_0 + x_1x_0\bar{y_1}\bar{y_0} + x_1x_0y_1\bar{y_0} + x_1x_0\bar{y_1}y_0$$

6. Alternately, for each combination of input values that yields 0 we can write a term using variables, negations, and or that is true for all other combinations of input values. Such a term is called a *maxterm*. The and of all of a truth tables functions provides another way to describe the associated boolean function. It is called a *product-of-sums* or *conjunctive normal form* formula.

7. The formulas produced might these constructions may not be the smallest or best, but they provide a proof that any boolean function can be described using just and, or and not. Since we know how to build and interconnect gates the compute and, or and not, this then proves that we can built a digital circuit for any boolean function!

Manipulating Boolean Formulas Algebraically

1. There are a number of axioms/identities akin to the commutative law or distributive law for arithmetic that can be used to manipulate formulas over boolean variables and operators. Tables 2.1, 2.2, and 2.3 in the text list these axioms, so I will not repeat them all here. They can be used to simplify the description of a boolean function and thereby ultimately simplify the circuit built to implement it.

2. We will focus on the axiom the book calls "combining":

$$(BC) + (B\bar{C}) = B$$

3. Using this rule, we can simplify the DNF formula for the comparison operation.

$$\bar{x_1}x_0\bar{y_1}\bar{y_0} + x_1\bar{x_0}\bar{y_1}\bar{y_0} + x_1\bar{x_0}\bar{y_1}y_0 + x_1x_0\bar{y_1}\bar{y_0} + x_1x_0y_1\bar{y_0} + x_1x_0\bar{y_1}y_0$$

   - The second and third terms are identical except the last literal in one is y_0 while in the next it is $\bar{y_0}$. So, we can combine these two terms:

$$\bar{x_1}x_0\bar{y_1}\bar{y_0} + x_1\bar{x_0}\bar{y_1} + x_1x_0\bar{y_1}\bar{y_0} + x_1x_0y_1\bar{y_0} + x_1x_0\bar{y_1}y_0$$

   - Next, the middle and last term only differ between y_0 and $\bar{y_0}$ so ...

$$\bar{x_1}x_0\bar{y_1}\bar{y_0} + x_1\bar{x_0}\bar{y_1} + x_1x_0\bar{y_1} + x_1x_0y_1\bar{y_0}$$

   - The middle two terms can then be combined by eliminating x_0 and $\bar{x_0}$

$$\bar{x_1}x_0\bar{y_1}\bar{y_0} + x_1\bar{y_1} + x_1x_0y_1\bar{y_0}$$

   Building a circuit based on this form rather than the original would clearly save some transistors.

Another Example

To make sure all of this makes sense, I would like to work on sketching out implementations of a few boolean functions that might be of use in a computer.

1. First, let's consider a 3-input "majority logic" circuit.

   - The backstory/motivation for this circuit comes from the idea or doing redundant computation for reliability.
   - Imagine that you build three separate computers to determine the answer to some important yes/no question and in the event they disagree you want to go with the majority vote.
   - That is, we want a circuit with three inputs whose output is the same as the most common input.

2. First, we can construct a truth table for this function.
   - It will have 3 columns for inputs and one for its output.
   - It will have $2^3 = 8$ rows.

3. Given this truth table we can construct a sum-of-products formula for the majority function. We could simplify it using axioms of boolean algebra.

4. Alternately, a little thought should make it obvious that the simpler expression $x_0x_1 + x_0x_2 + x_1x_2$ describes the function.

5. The good news, is that we can find good formulas and build good circuits for many tasks with such a common sense approach.

Circuits with Multiple Outputs

1. Finally, I would like to consider one circuit whose usefulness depends on the fact that it has more than a single output.

2. The circuit I have in mind is a 1-bit adder.

   - When we add two bits, we might get a 2-bit answer (if there is a carry). This is where the two outputs come from. One will be the "sum" of the two bits (i.e., what we would write under them if we were doing tabular addition) and the other will be the carry.

3. The reason the carry becomes essential, is that it will also be used as an input to a 1-bit adder.

   - When we do tabular addition, each column's carry comes from the carry before it.
   - Therefore, if we design a 1-column adder circuit to take three input bits (two for its column and one carry from the preceding column) we can easily interconnect multiple copies of this device to build a useful adder!

4. We can still build truth tables for the individual outputs of this device and use them to synthesize a circuit.

   - Or, as soon as we build the truth tables we might recognize the functions involved as exclusive or and carry and save ourselves some time.