Sergio A. Alvarez
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609
USA

web: http://www.cs.wpi.edu/∼alvarez
e-mail: alvarez@cs.wpi.edu
phone: (508) 831–5118
fax: (508) 831–5776

CS2011, A Term 1999
Notes on the Digital Logic Level
by Sergio A. Alvarez

In the hierarchy of levels presented at the start of the course, the digital logic level is at the bottom. Digital logic describes the chip-level operation of a digital computer. There is actually a level underneath this level - the device level - but I won't say very much about it (see note below, though). ECE courses provide further information about the device level.

The basic language at the digital logic level is Boolean algebra. This refers to operations on true/false-valued variables. The values 0 and 1 are often used instead of true and false. The following notes summarize some salient points of the class discussion on Boolean algebra and minimization of Boolean expressions (Karnaugh maps). These notes are not self-contained; they are expected to be helpful mostly to students attending the corresponding CS 2011 lectures.

Physical representation of binary notation

At the device level, 0 and 1 are represented by values of some physical quantity, typically voltage; 0 means about 1.2 Volts or less, while 1 means 2.5 Volts or more.

Positive vs. negative logic

Usually (positive logic), high voltage means true and low voltage means false. However, the opposite convention may also be used (negative logic). The same circuit will compute different Boolean functions in the two conventions. An example was given in class. See Fig 3-9.

Boolean algebra: the language of digital logic

Suppose that X and Y are variable symbols, each of which can assume the values true and false (or 1 and 0). Then X and Y can be transformed and combined to form new true/false (1/0) variables by way of the following operations:

Basic Boolean operations

1. Negation: from X, create "not X", denoted $\bar{X}$ (also X')
2. Conjunction: from X, Y, create "X and Y", denoted XY
3. Disjunction: from X, Y, create "X or Y", denoted X+Y

Gate symbols / truth tables

See Tanenbaum Fig 3-2.

Counting Boolean functions

Each of the above transformation/combination operations corresponds to choosing a function $f : \{true, false\}^n \rightarrow \{true, false\}$ for either n=1 (in the case of negation) or n=2 (conjunction, disjunction). There are other possible such functions; each function gives rise to a "Boolean operation".

Exercise: How many different Boolean operations are there that combine two different variables X, Y? Answer: There's one for each choice of a true/false 4-vector defining the truth table for the operation. There are $2^4 = 16$ such vectors. More generally, consider n true/false variables X1,...,Xn. There are 2^{2^n} different Boolean n-ary operations on these variables. (Can you see why?)

Disjunctive normal form

A Boolean function is determined by specifying which combinations of its input variables produce a true (1) output. This leads to a "sum of products" expression for the function.

Examples:

- A XOR B = AB' + A'B
- MAJORITY1(A,B,C) = ABC + ABC' + AB'C + A'BC (see Fig 3-3)

Equivalent Boolean expressions

Different Boolean expressions may be equivalent in that they have exactly the same true/false values for all choices of the input variable values. See Tanenbaum Fig 3-5.

Example:

```
MAJORITY2(A,B,C) = AB + AC + BC  (c.f. equivalent expression above)
```

There are several ways of checking equivalence:

1. **Brute force.** Compute the truth tables of two candidates for equivalence, and check that they are identical (try all input combinations explicitly).

   Example:

   ```
   A  B  C  MAJORITY1   MAJORITY2

   0  0  0
   0  0  1
   0  1  0
   0  1  1      1           1
   1  0  0
   1  0  1      1           1
   1  1  0      1           1
   1  1  1      1           1
   ```

2. **Algebraic manipulation.** Use Boolean algebra identities to transform a given Boolean expression into an equivalent one.

   Example:

   ```
   MAJORITY1(A,B,C) = ABC + ABC' + AB'C + A'BC
                    = ABC + ABC + ABC + ABC' + AB'C + A'BC
                    = ABC + ABC' + ABC + AB'C + ABC + A'BC
                    = AB(C+C') + A(B+B')C + (A+A')BC
                    = AB + AC + BC
   ```

Some Boolean identities

See Fig 3-6. The identities yield equivalent expressions for the basic gates. See Fig 3-7. For example, XOR can be computed in several ways (Fig 3-8).

Complete Boolean functions

There are binary Boolean functions in terms of which any Boolean function can be expressed using only composition. These special "complete" functions are NAND and NOR.

Answer to the "why" question (for NAND):

- X NAND X = NOT x, so NAND yields NOT
- NOT (X NAND Y) = X AND Y, so NAND yields AND
- NOT (NOT X AND NOT Y) = X OR Y, so NAND yields OR
- NOT, AND, and OR combined yield all disjunctive normal forms

See Fig 3-4.

Simplifying Boolean expressions using Karnaugh maps

Karnaugh maps provide a graphical aid to find simplified equivalent forms of a given Boolean expression. The basic (planar) Karnaugh map technique works for functions of up to 4 Boolean variables.

The Karnaugh map technique is based on the fact that the list of all pairs of values for two Boolean variables may be cyclically ordered in such a way that neighboring pairs in the list differ in only one of the two variables: 00 01 11 10 (and the list wraps around to 00 again). Notice that this differs from the common convention of listing the pairs in "counting order" (00 01 10 11).

Example: the majority function again

		C: 0	1
A	B		
0	0	0	0
0	1	0	1
1	1	1	1
1	0	0	1

Seek 2x2 squares full of 1's. There aren't any, so seek segments of two 1's. There are three, and they account for all the 1's. Read off the simplified expression. This yields AB + AC + BC.

See the handout provided in class for further information about Karnaugh maps.