ECE 124 - Digital Circuits and Systems

Andy Zhang

Winter 2014

Contents

Chapter 1

Binary Logic

1.1 Introduction

Despite being able to count in base 10, computers aren't cool enough to do that. They only count in base 2. Consequently, to communicate with computers, we need to understand how base 2 works.

1.2 Converting from One Base to Another

The best way to explain this is through pseudocode:

$size \leftarrow \lceil \log_2 number \rceil$
$binary[size]$
$A \leftarrow -1$
$index \leftarrow size - 1$
while $A \neq 0$ **do**
  $binary[index] \leftarrow number\%2$
  $index \leftarrow index + 1$
  $A \leftarrow number/2$
end while

Example: $53 = (110101)_2$

1.3 Truth Tables

We define a logic function as a truth table. We exhaust all the different combinations of the input along with their output.

x	y	f
0	0	f_0
0	1	f_1
1	0	f_2
1	1	f_3

1.4 Basic Logical Operations

Here's a table of all the logic functions

Logic Operator	Symbol	Example
AND	$\cdot$, nothing	f_0
OR	$+$	f_1
NOT	!,$'$,$\neg$, $\bar{a}$	f_2

Please refer to `http://en.wikipedia.org/wiki/Logic_gate` for an image of each gate and their respective logic tables.

1.5 Properties of Boolean Algebra

- $x + 0 = x$, $x \cdot 1 = x$
- $x + x' = 1$, $x \cdot x' = 0$
- $x + x ==$, $x \cdot x = x$
- $x + 1 = 1$, $x \cdot 0 = 0$
- $(x')' = x$
- $x + y = y + x$, $x \cdot y = y \cdot x$
- $(x + y) + z = x + (y + z)$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
- $x \cdot (y + z) = x \cdot y + x \cdot z$, $x + y \cdot z = (x + y) \cdot (x + z)$
- $(x + y)' = x' \cdot y'$, $(x \cdot y)' = x' + y'$
- $x + x \cdot y = x$, $x \cdot (x + y) = x$

1.6 Circuit Costs

NOT's are ignored, each gate costs 1 and each input costs 1

1.7 Minterms and Maxterms

Minterm:
For each row of the truth table, create an AND of the literals according to the following rule: If a variable has value 1 in the row, include its **+ve iteral**. If a variable x has a value 0 in the row, include its -ve literal. **Maxterm**:
For each row of the truth table, create an OR of the literals according to the following rule: If a variable x has the value 0 in the row, include its +ve literal. If a variable x has value 1 in the row, include its -ve literal.

We can then define a function as $f = m_a m_b m_c = m_d + m_e + m_f$

1.8 Karnaugh Maps

A Karnaugh Map contains all of the same information as a truth table but in a different form. The coordinates of each square determine the its value.
While labeling the rows and columns, it's important that only 1 value changes between adjacent rows and columns.

We can encompass groups of minterms via rectangles that are sizes of powers of 2(i.e. 1,2,4,8...). We can then represent that rectangle based on its coordinates that don't change. As a result, enclosing larger rectangles leaves less literals which is better.
Note that we can enclose minterms via the rectangles, from one side to another or adjacent.

Above 4 variables, we require more than 1 Karnaugh Map since we can only fit 4 variables in 1.

The same can be done with maxterms where you may enclose all of the 0s and create a Product of Sums.

Don't cares(X) are useful as they can be any value which can make minimizing a function simpler.

1.9 Karnaugh Map Elements

- A product term is called an **Implicant** if the logic function outputs a 1 for all minterms in the product term.
- An Implicant is called a **Prime Implicant** if the removal of any literal from the implicant results in a new product term that is not an implicant.
- A prime implicant is called an **Essential Prime Implicant** if it includes a minterm that is not found in any other implicant.

1.10 NAND NOR

OR Gate via NAND: $f = (x' \cdot y') = x + y$
NAND Gate via OR: $f = x' + y' + z' = x\bar{y}z$

When changing a circuit, a NOT on all inputs leads to a toggle between AND/OR and a NOT at the output. Ex: OR with all NOT input = NAND and vice versa.

1.11 XOR Gate

XOR gates are often hidden and can save a lot in terms of circuit costs.
$f_{XOR} = x_1 \oplus x_2 = \bar{x_1}x_2 + x_1\bar{x_2}$
$f_{XNOR} = x_1 \bar{\oplus} x_2 = \bar{x_1} \cdot \bar{x_2} + x_1 \cdot x_2$

Chapter 2

Circuits

2.1 Combinational Circuit

A combinatorial circuit is one that consists of logic gates with outputs that are determined entirely by the present value of the inputs.
Two operations we may perform:

- Analysis: given what is in the box, what function does it perform? We analyze by determining a function for the output starting from the input.
- Given functions to perform, what do we need in the box? We derive a truth table based on specification followed by simplifying it and implementing a circuit.

2.2 Comparator Algorithms

Equality: It's pretty straight forward. $A_i = B_i$ for any i. This can be done by letting $e_i = a_i b_i + a_i' b_i'$. If $e_i e_{i-1} ... e_2 e_1 = 1$, then $A = B$.
What if they weren't equal? We let $i = n$, the most significant bit(MSB). If $a_i > b_i$, then $A > B$ and we stop and vice versa. Otherwise, let $i = i - 1$. Stop once i=0 and all bits are considered.

$(A > B) = a_n b_n' + (e_n)(a_{n-1} b_{n-1}') + (e_n e_{n-1})(a_{n-2} b_{n-2}') + ... + (e_n e_{n-1} ... e_2 e_1 e_0)(a_0 b_0')$
$(A < B) = a_n' b_n + (e_n)(a_{n-1}' b_{n-1}) + (e_n e_{n-1})(a_{n-2}' b_{n-2}) + ... + (e_n e_{n-1} ... e_2 e_1 e_0)(a_0' b_0) = (A > B)'$

2.3 Decoders

When we have n bits, we can represent 2^n distinct patterns. We have **n-to-m decoders** with $m = 2^n$.
Often, some decoders have an enable signal. When enable = 1, then the decoder functions as desired, otherwise, all outputs = 0. For an image or more details, refer to `http://en.wikipedia.org/wiki/Decoder`
Smaller decoders become useful to implement larger ones.

2.4 Encoders

Encoders perform the inverse of a decoder. They have 2^n or fewer input and n output. If there are more than 1 high input, then there's an undefined output which leads to priority encoders which gives priority to higher numbered inputs.

2.5 Multiplexers

Multiplexers have data inputs, select inputs and single data outputs. The output data is based on the select input and the input data. Refer to `http://en.wikipedia.org/wiki/Multiplexer` for images and more detailed explanation.
Multiplexers are useful in many ways and can also be used to implement a n input function using a $n-1$ input multiplexer. Based on each pair of output, we can determine the input for the multiplexer.

2.5.1 Shannon Decomposition

Breaking a function down for a MUX implementation is called **Shannon Decomposition**
This works because given a boolean function $f = f(x_0, x_1...x_n)$, we can split the function into to as the following:

$$f = f(x_0, x_1...x_n) = x_0' f(0, x_1, x_2...x_n) + x_0 f(1, x_1, x_2...x_n)$$