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:

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

2.6 Sequential Circuits

Unlike combinatorial circuits, the output of a sequential circuit also depends on the current state of the circuit. We define two types of sequential circuits:

- Asynchronous Circuits: circuit behaviour is determined by signals at **any instant in time and the order in which input signals change**
- Synchronous Circuits: circuit behaviour is determined from the knowledge of signal values at **discrete instances of time**

2.6.1 Clock Signals

A **clock signal** is used to control the behaviour of a circuit at discrete instances in time. They are periodic signals.
When the signal transitions from $0 \to 1$, we call it a rising edge. If it's $1 \to 0$, we call it a falling edge.

2.7 Latches

Latches are a storage element. They perform when a control signal is at 0 or 1 and not at any changing edge. This element is extremely useful to understand flip flops.
There exists several types of latches(`http://en.wikipedia.org/wiki/Flip-flop_(electronics)` for more detail and images)

2.7.1 SR Latch

S	R	Q	$\bar{Q}$
1	0	0	1
1	1	0	1
0	1	1	0
1	1	1	0
0	0	1	1

- When $S = 0$, $R = 1$, the output is $Q = 1$, and the circuit is in the **set state**
- When $S = 1$, $R = 0$, the output is $Q = 0$, and the circuit is in the **reset state**
- When $S = 1$, $R = 1$, it implies the reset state
- When $S = 0$, $R = 0$, the output holds at its previous value(storage)

2.7.2 SR Latch

S	R	Q	$\bar{Q}$
1	0	0	1
1	1	0	1
0	1	1	0
1	1	1	0
0	0	1	1

- When $S = 1$, $R = 0$, the output is $Q = 1$, and the circuit is in the **set state**
- When $S = 0$, $R = 1$, the output is $Q = 0$, and the circuit is in the **reset state**
- When $S = 0$, $R = 0$, it implies the reset state
- When $S = 1$, $R = 1$, the output holds at its previous value(storage)

2.7.3 SR Latch

C	S	R	next value of Q
0	X	X	hold
1	0	0	hold
1	0	1	0(reset)
1	1	0	1(set)
1	1	1	undefined

- When the control input $C = 0$, both latches are in hold state
- When the control input $C = 1$, the latch acts like an SR Latch(active high)

2.7.4 SR Latch

C	D	next value of Q
0	X	hold
1	0	0
1	1	1

In previous latches, we had the undesirable situation where the output was undefined due to both outputs being the same. To avoid this, we construct a latch where S and R can never be the same where $S = R'$

2.7.5 Issues

Latches don't allow precise control since they're only level sensitive. This creates an interval in time where the output can change rather than an instant. It would be better if the output can only change at an instant for more precise control.

2.8 Flip Flops

Unlike latches, flip flops allow more precise control on the output by letting the output change only during triggering i.e. changing edges.

2.8.1 DFF

Refer to `http://en.wikipedia.org/wiki/Flip-flop_(electronics)#D_flip-flop` for a much better explanation of how DFF works. Here's a rough explanation:
A DFF consists of 2 D Latches connected in series where the output of the first is the D input of the second. The clock input of each are inverses of the other.
While $CLK = 1$, the output of the first latch will follow the D input. When $CLK = 0$, the output will be disconnected from the first latch and connected to the second since $CLK = 1$ for the second(since the clocks are inverses of each other). The output of the second is then changed when the clock is triggered.

Depending on the type of trigger the flip flop depends on, we can position the inverter of the clocks on different inputs i.e. either the first or the second latch.

2.8.2 Sets, Resets and Enables

Flip flops can have additional control signals that will force Q to a known value.

- An asynchronous signal that forces $Q = 1$ is called an asynchronous **set** and remain that way if Q stays 1
- An asynchronous signal that forces $Q = 0$ is called an asynchronous **reset** and remain that way if Q stays 0
- A signal that prevents the clock from causing changes in Q according to D is called a clock enable

2.8.3 Characteristic Tables of Flip Flops

DFF

D	$Q(t+1)$
0	0
1	1

$Q(t+1) = D$

TFF

T	$Q(t+1)$
0	$Q(t)$
1	$Q'(t)$

$Q(t+1) = TQ'(t) + T'Q(t) = T \oplus Q(t)$

JKFF

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$Q'(t)$

$Q(t+1) = JQ'(t) + K'Q(t)$

2.8.4 Timing Analysis

In reality, it takes time for gates to change their outputs. There exist 3 timing parameters that are especially important:

- Setup Time(TSU): time it takes for data inputs to be stable prior to arrival of an active clock edge
- Hold Time(TH): time it takes for the data inputs need to be held stable after the arrival of the active clock edge
- Clock-To-Output Time(TCO): time it takes for the output to become stable after the arrival of the active clock edge