Notes for ECE 27000 - Introduction to Digital System Design

Zeke Ulrich

January 22, 2025

Contents

Course Description

An introduction to digital system design, with an emphasis on practical design techniques and circuit implementation.

Number Systems

In daily life, we primarily interact with the familiar base-10 numbers. However, when interaction with digital systems, we must also concern ourselves with base-2, base-8, base-16, and other bases which are friendly to binary states. Unless completely unambiguous, the base of a number is written as a right subscript such as 144_{10} for base-10 or 1001_2 for base-2.

For binary numbers, each digit represents a power of two. To convert a binary number to decimal, you sum the products of each binary digit with its corresponding power of two. For example, the binary number 1001 is calculated as

$$\begin{aligned} 2^3 \times 1 + 2^2 \times 0 + 2^1 \times 0 + 2^0 \times 1 &= 8 + 0 + 0 + 1 && (1) \\ &= 9_{10} && (2) \end{aligned}$$

To convert from hexadecimal to decimal, each digit represents a power of sixteen. For instance, the hexadecimal number f1 is calculated as

$$\begin{aligned} 15 \times 161 + 1 \times 160 &= 240 + 1 && (3) \\ &= 241_{10} && (4) \end{aligned}$$

where f represents the decimal value 15. When converting to another base, reverse the process by dividing the decimal number by the target base, recording the remainder, and repeating with the quotient until it reaches zero. The remainders give you the digits of the number in the new base, read in reverse order.

To convert a decimal number into binary, for example, you repeatedly divide the number by 2 and record the remainders. For the decimal number 9, dividing by 2 gives a quotient of 4 and a remainder of 1. Dividing 4 by 2 gives a quotient of 2 and a remainder of 0. Dividing 2 by 2 gives a quotient of 1 and a remainder of 0, and finally, dividing 1 by 2 gives a quotient of 0 and a remainder of 1. Reading the remainders from bottom to top, the binary representation of 9 is 1001.

In SystemVerilog, numbers are written in format `[size]'[base][number]`, for example:

- 4'b1001 // binary, 9 in decimal, bit width 4 bits
- 8'hf1 //hex, equals 421, bit width 8 bits
- 3'o3 // octal, 3, bit width 3 bits
- 32'b1001_1101_0101_1111 // binary, 40255, bit width 32 bits

Boolean Algebra

Computers operate in binary. To represent the state of a computer we require a suitable mathematical framework, provided by boolean algebra. In boolean algebra, variables can only take on two values: 0 and 1.

Table 1: Boolean Algebra

Rule	**Expression**
Commutativity	$X + Y = Y + X$ $X \cdot Y = Y \cdot X$
Associativity	$(X + Y) + Z = X + (Y + Z)$ $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$
Distributivity	$X \cdot Y + X \cdot Z = X \cdot (Y + Z)$ $(X + Y) \cdot (X + Z) = X + Y \cdot Z$
Covering	$X + X \cdot Y = X$ $X \cdot (X + Y) = X$
Combining	$X \cdot Y + X \cdot Y = X$ $(X + Y) \cdot (X + Y) = X$
Consensus	$X \cdot Y + X \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$ $(X + Y) \cdot (X + Z) \cdot (Y + Z) = (X + Y) \cdot (X + Z)$
Generalized Idempotency	$X + X + \cdots + X = X$ $X \cdot X \cdot \cdots \cdot X = X$
DeMorgan's Theorems	$(X_1 \cdot X_2 \cdot \cdots \cdot X_n)' = X_1' + X_2' + \cdots + X_n'$ $(X_1 + X_2 + \cdots + X_n)' = X_1' \cdot X_2' \cdot \cdots \cdot X_n'$
Generalized DeMorgan's	$F(X_1, X_2, \ldots, X_n, +, \cdot) = F(X_1, X_2, \ldots, X_n, \cdot, +)'$
Shannon's Expansion	$F(X_1, X_2, \ldots, X_n) = X_1 \cdot F(1, X_2, \ldots, X_n) + X_1' \cdot F(0, X_2, \ldots, X_n)$ $F(X_1, X_2, \ldots, X_n) = [X_1 + F(0, X_2, \ldots, X_n)] \cdot [X_1' + F(1, X_2, \ldots, X_n)]$

An interesting and useful property in boolean algebra is "duality", where replacing all ANDs with ORs and all 1s with 0s gives a valid and equivalent theorem. For instance,

Table 2: Boolean Duality

X AND 0 = 0	X OR 1 = 1
X OR 0 = X	X AND 1 = X

Any logic can be implemented using just the following:

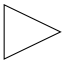
- AND, OR, and NOT gates
- NAND gates
- NOR gates

Logic Gates

We represent boolean operations in circuit diagrams with gates. Below are their symbols and corresponding truth tables.

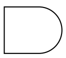
Buffer

A	Output
0	0
1	1



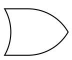
AND

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1



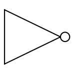
OR

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1



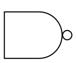
NOT

A	Output
0	1
1	0



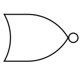
NAND

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0



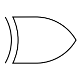
NOR

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0



XOR

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0



XNOR

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1