Boolean Algebra and binary system

Giuseppe Lipari
http://feanor.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

January 19, 2010

Outline

Outline

An algebra for logic

- Domain: {true, false}
- Basic operations: {and, or, not}
- Truth tables:

a and b	**false**	**true**
false	false	false
true	false	true

a or b	**false**	**true**
false	false	true
true	true	true

a	not a
false	true
true	false

Examples of logic predicates

- Axioms
  - Today is raining
  - John carries an umbrella
  - John wears sunglasses
- Predicates
  - Today is raining and John carries an umbrella is true and true $\equiv$ true

Examples of logic predicates

- Axioms
  - Today is raining
  - John carries an umbrella
  - John wears sunglasses
- Predicates
  - Today is raining and John carries an umbrella is true and true $\equiv$ true
  - not today is raining and John wears sunglasses $\equiv$ not true and true $\equiv$ false

Examples of logic predicates

- Axioms
  - Today is raining
  - John carries an umbrella
  - John wears sunglasses
- Predicates
  - Today is raining and John carries an umbrella is true and true $\equiv$ true
  - not today is raining and John wears sunglasses $\equiv$ not true and true $\equiv$ false
  - not today is raining or John wears sunglasses $\equiv$ not true or true $\equiv$ true

Other operators

- a nand b $\equiv$ not (a and b)
- a nor b $\equiv$ not (a or b)
- a $\rightarrow$ b $\equiv$ not (a and not b)
- a xor b $\equiv$ (a or b) and not (a and b)
- It can be shown that every operator can be derived by either nand or nor

Properties

- It is an algebra, thus it has the following properties:
  - the identity for and is true
  - the identity for or is false
  - the null element for and is false
  - commutativity. ex: a and b $\equiv$ b and a
  - associativity. ex: a or (b or c) $\equiv$ (a or b) or c

Boolean algebra in digital electronic systems

- It is possible to build electronic logic gates that
  - Interpret high voltage as true and low voltage as false
  - Implement logic operations like nand and nor

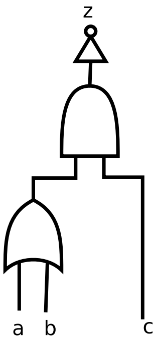


Figure: A logic circuit that implements z $\equiv$ not ((a or b) and c)

Outline

Positional notation

- Humans use a positional notation in base 10
- We have 10 symbols: from 0 to 9

$$176435_{b(10)} = \\ 1 \cdot 10^5 + 7 \cdot 10^4 + 6 \cdot 10^3 + 4 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$$

- In base 8 (octal), we have 8 symbols: from 0 to 7
- The same "number", expressed in base 8 would be:

$$176435_{b(8)} = \\ 1 \cdot 8^5 + 7 \cdot 8^4 + 6 \cdot 8^3 + 4 \cdot 8^2 + 3 \cdot 8^1 + 5 \cdot 8^0 = \\ 64797_{b(10)}$$

Boolean algebra in computers

- In digital electronic systems, high and low voltages are interpreted as two different symbols, 1 and 0 respectively
- It is possible to build arithmetic using binary encoding of numbers and symbols
- Definitions:
  - one binary digit (0 or 1) is a *bit*
  - a group of 8 binary digits is a *byte*
  - a *word* in current processor is 4 bytes (32 bits)

Binary encoding integer numbers

- Translation from decimal to binary and viceversa
  - Let's start from positive integer numbers
  - the mimimum number is `0000 0000` (0 in decimal)
  - the maximum number is `1111 1111` (255 in decimal)
  - how to translate a binary number:

$$0100\ 1011 =$$
$$0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 75$$

$$0011\ 0110 =$$
$$0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 + 0 \cdot 2^6 + 0 \cdot 2^7 = 54$$

Summing integer numbers

- By using boolean logic, we can implement binary adders
- Truth table of an adder: s = x + y, plus the carry

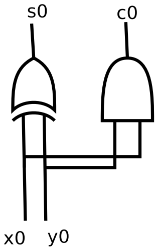
x\ y	0	1
0	0	1
1	1	0

x\ y	0	1
0	0	0
1	0	1

- By interpreting 0 as false and 1 as true, the sum can be expressed as:
  - s = x xor y
  - c = x and y

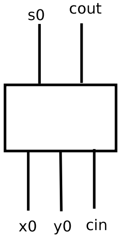
Basic adder and full adder

The following diagram represent a 2-bit adder



Full adder

Let's consider a component that can be used to build more complex adders:



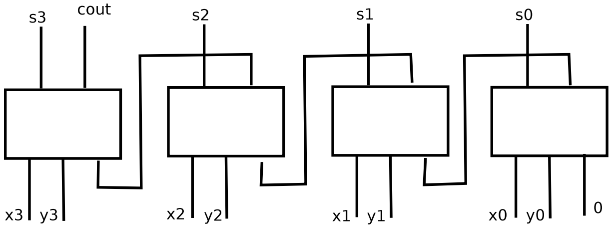
- Truth table:

x_0	0	1	0	0	1	1	0	1
y_0	0	0	1	0	1	0	1	1
c_{in}	0	0	0	1	0	1	1	1
$c_{out}\, s_0$	00	01	01	01	10	10	10	11

- $s_0 = x_0$ xor y_0 xor c_{in}
- $c_{out} = (x_0$ and $y_0)$ or $(x_0$ and $c_{in})$ or $(y_0$ and $c_{in})$

Full adders

To implement a 4-bit adder, we compose 4 full-adders:



How to represent negative numbers

There are many ways to represent negative integers

1. use the first bit as a sign: 0 is positive, 1 is negative
   - 0111 1111 corresponds to 127
   - 1111 1111 corresponds to -127
   - Problems:
     - zero is represented twice, 1000 0000 and 0000 0000
     - Not possible to directly use this representation in sums
2. Two's complement
   - represent positive numbers up to 127 normally
   - represent negative numbers as the positive, negated (bit by bit) plus 1

   Example: represent -58 on 8 bits:
   - 58 is: 0011 1010
   - negation is: 1100 0101
   - plus 1: 1100 0110

   Hence, the representation of -58 is 1100 0110

Advantages of two's complement

- Range with 9 bits: $(-128 \; ; +127)$
- By summing positive and negative numbers using two's complement representation, the result is correct if it is in range

- Example 1:
  - -58 is 1100 0110
  - 64 is: 0100 0000

Sum

1100 0110	+
0100 0000	=
0000 0110	(6)

- Example 2:
  - -58 is 1100 0110
  - 32 is: 0010 0000

Sum

1100 0110	+
0010 0000	=
1110 0110	(-26)

Outline

How to represent everything

- The challenge is to represent everything with just two symbols
  - Numbers, text, drawings, images, pictures, sounds, complex music, etc.
- Let's start with numbers:
  - We already know that with 8 bits we can represent integers from -128 to 127
  - with 16 bits (2 bytes) we can represent from $-2^{15}(-32768_{b(10)})$ to $2^{15}-1(32767_{b(10)})$
  - with 32 bits (4 bytes) we represent from $-2^{31}(-2,147,483,648_{b(10)})$ to $2^{31}-1(2,147,483,647_{b(10)})$
  - with 64 bits (8 bytes) we represent up to $2^{63}-1$ which is $9,223,372,036,854,775,807_{b(10)} \approx 10^{20}$ (approximately the number of grains of sand on earth)

Representing decimal numbers

Two possible systems:

- Fixed point representation: a fixed number of bits are for the integer part, the remaining for the rational part
  - Used in some embedded system (DSP) because calculations are usually faster
  - fixed precision, limited range
- Floating point representation: a fixed number of bits to represent the mantissa, and the remaining to represent the exponent
  - Used in modern PCs
  - very wide range, variable precision

IEEE 754 standard for floating point

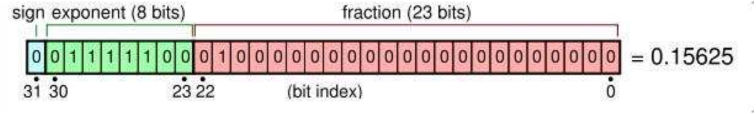


Figure: Floating point, single precision

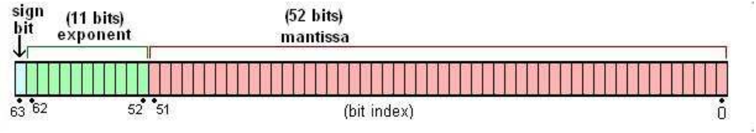


Figure: Floating point, double precision

Symbols and meaning

- Obviously, symbols are meaningless *per-se*
- The meaning the we attach to them depends on the context
- A string of 64 bits can be:
  - a large integer, two smaller 32-bit integers, four 16-bits integers, eight 8-bits integers
  - or, two 32-bits double precision floating point numbers
  - or, four 16 bits single precision gloating point numbers
  - or . . . ?

Representing characters

- It is possible to represent characters and strings of characters using an appropriate encoding
- The ASCII encoding assigns each character a number between 0 and 255
- Some example of character encoding:

bin	dec	glyph
011 0000	48	'0'
011 0001	49	'1'
011 0010	50	'2'
011 0011	51	'3'
011 0100	52	'4'
011 0101	53	'5'
011 0110	54	'6'
011 0111	55	'7'
011 1000	56	'8'
011 1001	57	'9'

bin	dec	glyph
110 0001	97	a
110 0010	98	b
110 0011	99	c
110 0100	100	d
110 0101	101	e
110 0110	102	f
110 0111	103	g
110 1000	104	h
110 1001	105	i
110 1010	106	j

Representing text

- A simple text:

  This course is valid 3 credits.

- And its representation

T	h	i	s		c	o	u
01010100	01101000	01101001	01110011	00010000	01100011	01101111	01110101
r	s	e		i	s		v
01110010	01110011	01100101	00010000	01101001	01110011	00010000	01110110
a	l	i	d		3		c
01100001	01101100	01101001	01100100	00010000	00110011	00010000	01100011
r	e	d	i	t	s	.	
01110010	01100101	01100100	01101001	01110100	01110011	00101110	

Figure: A string of text, and its binary representation.

Other characters

- There are many characters in the world
  - Chinese, japanese, hindu, arabic, ...
- The new standard Unicode covers all possible characters, needs 16 bits per character
- An Unicode representation of the same text will take double the space of the correponding ASCII encoding (16 bits per character, instead of 8 bit per character)
- It is possible to compact a text by using an appropriate compression algorithm (tries to avoid repetition of symbols by using a different (and more compact) encoding

Representing signals

- With bits and bytes we can represent numbers
- To represent functions (i.e. signal), we can store sequence of numerical values.
  - For example, to represent music, we can store one function $f_i(t)$ for each instrument
  - Since the function is continuous, we first *sample* it in small intervals of time.

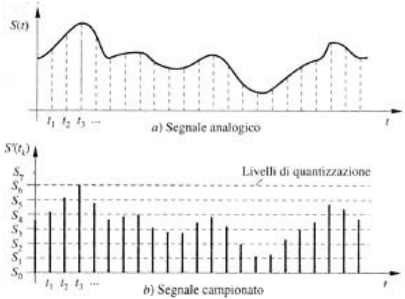


Figure: Sampling

Representing music

- Like in the case of the text, raw music representation has a lot of redundant information, and take a lot of space
  - For example, 1 msec sampling, each value with 32 bits, means approximately 4 Kb per second per channel

File Type	44.1 Khz	22.05 Khz	11.025 Khz
16 bit stereo	10.1	5.05	2.52
16 bit mono	5.05	2.52	1.26

Table: Memory requirement for 1 minute of Digital Audio (all numbers in Mbytes)

- Of course, there is a tradeoff between sampling rate and *quality*.
  - See `http://www.cs.cf.ac.uk/Dave/Multimedia/node150.html` for a comparison
- It is possible to *compress* such representation by using appropriate encoding alg orithms (e.g. mp3, ogg, etc.), although some quality gets lost.

Pictures

- Something similar is done with pictures
  - A picture is first divided into *pixels*
  - Each pixel is represented as a number or a set of numbers
  - Most common representation is RGB (Red-Green-Blue)
  - By using 8 bits for each of the three colors, each pixel is represented by 24 bits
    - A 1024x800 image is large $3x1024x800 \approx 2$ Mbytes.
  - Of course, it is possible to compress pictures as well
- Finally, movies are just sequences of pictures. Here compression is utterly necessary!

Outline

Representing information

- At the end, every information is coded as a sequence of just two symbols: 0 and 1
- A processor just acts on such two symbols to perform any kind of computation
- How does a processor know what to do?
- Processors are *programmable* machines
- They take
  1. A *program*, i.e. a sequence of *instructions* (the recipe!)
  2. any sequence of bits as input,
  3. and perform *transformations* (computations) on this sequence according to the program, to produce a sequence of bits in output
- In the next, we will give an overview of how this process works

Outline

Questions

1. How would you represent rational numbers?
   - A rational number is for example $\frac{1}{3}$. It cannot be finitely represented as a decimal number because it has infinite digits $1.3333\ldots$.
2. Is there any way to represent irrational numbers with infinite precision?
   - For example $\sqrt{2}$, *e*, π, etc.
3. Is there a way to represent an integer number with an arbitrary large number of digits?