

CSC2/452 Computer Organization

Bits and Integers

Sreepathi Pai

URCS

September 3, 2019

Outline

Administrivia

Introduction

Bits, Functions and Boolean Algebra

Machine Data Types

Interpreting Bits as Integers

Outline

Administrivia

Introduction

Bits, Functions and Boolean Algebra

Machine Data Types

Interpreting Bits as Integers

Announcements

- ▶ Course website has office hours for TAs
 - ▶ Link to course website announced on Blackboard
- ▶ Use Piazza (link from Blackboard)
- ▶ I plan to incorporate electronic devices into my lectures
 - ▶ Cellphones and/or Laptops will be required
 - ▶ Please be *respectful* of others around you
 - ▶ If you do not want to use cellphones/laptops, please talk to me after class for options

Outline

Administrivia

Introduction

Bits, Functions and Boolean Algebra

Machine Data Types

Interpreting Bits as Integers

Real World Data

- ▶ Numbers
- ▶ Text
- ▶ Pictures
- ▶ Audio
- ▶ Scents
- ▶ ...

Most can be encoded as numbers

Building Blocks of the Digital Universe

And most numbers can be encoded as binary digits (or *bits*), consisting of the values 0 and 1.

Outline

Administrivia

Introduction

Bits, Functions and Boolean Algebra

Machine Data Types

Interpreting Bits as Integers

Functions of 1 input (unary functions)

- ▶ ZERO (output is always zero)

Input	Output
0	0
1	0

- ▶ ONE (output is always one)

Input	Output
0	1
1	1

Functions of 1 input (contd.)

- ▶ IDENTITY (output is always equal to input)

Input	Output
0	0
1	1

- ▶ INVERSE (or NOT) (output is inverse of input)

Input	Output
0	1
1	0

- ▶ There are only four unary bitwise functions (or operations).
- ▶ Bitwise functions are also called boolean functions

Functions of 2 inputs (binary functions)

► *ZERO*

a	b	Output
0	0	0
0	1	0
1	0	0
1	1	0

► *ONE*

a	b	Output
0	0	1
0	1	1
1	0	1
1	1	1

► Note that *ONE* is essentially $NOT(ZERO(a, b))$

Truth Tables and Boolean Functions

- ▶ The tables in the previous slides are called “Truth tables”
 - ▶ the textbook uses a slightly more compact form
- ▶ If there are I inputs, a truth table has $R = 2^I$ rows
- ▶ If the output is a single bit, then there are $F = 2^R$ different outputs
 - ▶ This is also the total number of boolean functions of I inputs
 - ▶ e.g., $I = 1 \rightarrow R = 2^1 \rightarrow F = 2^2 = 4$
 - ▶ e.g., $I = 2 \rightarrow R = 2^2 \rightarrow F = 2^{2^2} = 16$
- ▶ Half of these functions can be obtained by inverting the other half

AND

- ▶ *AND* outputs 1 only when both inputs are 1

<i>a</i>	<i>b</i>	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR

- ▶ *OR* outputs 1 if either input is 1
 - ▶ hence, “inclusive or”
 - ▶ *not* how it is used in English!

a	b	Output
0	0	0
0	1	1
1	0	1
1	1	1

XOR

- ▶ *XOR*, 1 only when exactly one of its input is 1
 - ▶ hence, “exclusive or”
 - ▶ pronounced “ecks-or” (i.e. x-or) or “zor”
 - ▶ I prefer the latter...

a	b	Output
0	0	0
0	1	1
1	0	1
1	1	0

NAND and NOR

- ▶ $NAND = NOT(AND(a, b))$

a	b	Output
0	0	1
0	1	1
1	0	1
1	1	0

- ▶ $NOR = NOT(OR(a, b))$

a	b	Output
0	0	1
0	1	0
1	0	0
1	1	0

- ▶ $NAND$ and NOR are universal gates
 - ▶ Can be used to implement any boolean function

Examples of NAND

- ▶ What should ? be in the following examples to make LHS = RHS?
 - ▶ $NOT(a) = NAND(a, ?)$
 - ▶ $AND(a, b) = NAND(NAND(a, b), ?)$
 - ▶ $OR(a, b) = ?$

Generalizing to inputs longer than one bit

- ▶ Inputs longer than one bit are called:
 - ▶ bit vectors
 - ▶ bit strings
 - ▶ or more specific names for particular names (e.g. 8 bits = byte)

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
	0	1	0	1	1	0	0	1
<i>AND</i>	0	1	1	1	0	1	1	0
	0	1	0	1	0	0	0	0

- ▶ Each bit in the first 8-bit input is ANDed to its corresponding bit in the second input
- ▶ The AND operates on each pair of bits separately

Logic and Boolean Algebra

- ▶ Logical variables take only values TRUE and FALSE
- ▶ Logical operations are operations on these values
 - ▶ e.g., “Not True = False”
- ▶ Systematized by George Boole in 1847
 - ▶ Later expounded in *The Laws of Thought*, 1854
- ▶ Claude Shannon connected boolean algebra to digital circuit design
 - ▶ Originally, to design circuits that used electromechanical relays as switches
 - ▶ Now digital circuits use transistors, but principles are the same
 - ▶ Also coined the word “bit” later...

Outline

Administrivia

Introduction

Bits, Functions and Boolean Algebra

Machine Data Types

Interpreting Bits as Integers

Bits, Bytes, Words, ...

- ▶ Almost no machine allows manipulation of single bits directly
- ▶ Bits are handled as aggregations

Size (bits)	Common Name
8	byte
16	word, halfword
32	word, doubleword
64	word, doubleword, quadword
128	?

- ▶ A *machine* word (sometimes the word “machine” is omitted) is the size (in bits) of data that a machine can manipulate at once.
 - ▶ Hence 16-bit machines, 32-bit machines, 64-bit machines, etc.

Reading a byte

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	1	1	0	1	1	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- ▶ In place-value notation, $b_0 = 1$ and $b_7 = 2^7 = 128$
 - ▶ Hence, this is $1 \times 2^6 + 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 = 109$
- ▶ The grouping of 4 bits together is called a nybble (i.e. half a byte)
 - ▶ Primarily improves readability
 - ▶ But can also be used to easily convert to base-16 (i.e. hexadecimal)
- ▶ b_0 (i.e. rightmost bit) is called the least significant bit (LSB)
 - ▶ contributes the smallest value (2^0)
- ▶ b_7 (i.e. leftmost bit) is called the most significant bit (MSB)
 - ▶ contributes the most value (2^7)

Hexadecimal

- ▶ Numbers in base 16
 - ▶ 0 to 9 and *A* to *F*
 - ▶ Usually indicated by a *0x* prefix, or a 16 subscript
 - ▶ e.g., $0xA = A_{16} = 10_{10} = 1010_2$
- ▶ $109_{10} = 0110\ 1101_2 = 0x6D$
- ▶ Textbook contains a table mapping the 16 nybbles to hexadecimal symbols
- ▶ Hexadecimal is widely used in low-level code
 - ▶ you'll get plenty of opportunities to practice

Multibyte Data Types and Memory Layout

- ▶ The 16-bit value 51996_{10} has hexadecimal representation $0x\text{CAFE}$
 - ▶ Its binary representation is $1100\ 1010\ 1111\ 1110_2$
 - ▶ The value $0xCA$ is its most significant *byte*
 - ▶ The value $0xFE$ is its least significant *byte*
- ▶ RAM is byte addressable
 - ▶ Can read individual bytes of a multibyte value
 - ▶ How should we order each byte of a multibyte value?

Little and Big-endian

- ▶ Storing a 32-bit value $0xDEADCAFE$ in memory
- ▶ Big endian: Most significant byte at lower addresses
- ▶ Little endian: Least significant byte at lower addresses

address	x	$x + 1$	$x + 2$	$x + 3$
big-endian	0xDE	0xAD	0xCA	0xFE
little-endian	0xFE	0xCA	0xAD	0xDE

- ▶ Different machines use different conventions
 - ▶ Intel/AMD usually little endian
 - ▶ SPARC/PowerPC usually big endian
 - ▶ ARM can switch between the two
- ▶ Big endian is sometimes called network byte order
 - ▶ Similar problem: which byte of a word gets on the wire first?

The Interpreter of Bits

- ▶ Does the byte 0x55 in memory indicate:
 - ▶ The integer value 85?
 - ▶ The Intel assembly language instruction `push %rbp` (as seen in the previous lecture)?
- ▶ There is nothing in 0x55 that can distinguish between these two interpretations
 - ▶ Very powerful idea
 - ▶ Code can be data and data can be code

Outline

Administrivia

Introduction

Bits, Functions and Boolean Algebra

Machine Data Types

Interpreting Bits as Integers

Integers

- ▶ The most common interpretation of bytes, words, etc. is that as “integers”
 - ▶ Whole numbers (no fractional part)
 - ▶ Can be positive or negative
- ▶ Examples: -3, -2, -1, 0, 1, 2, 3

Problem to be solved

- ▶ Need to store the magnitude of the integer
 - ▶ i.e. absolute value (e.g. $|-2| = 2$)
- ▶ Need to store sign of the integer

How many bits are required?

- ▶ The number of bits required to store N distinct values is $\lceil \log_2(N) \rceil$
 - ▶ i.e. logarithm of N to the base 2
 - ▶ i.e. find x such that $2^x = N$, and round it up
- ▶ Example #1: There are two possible values for sign, so $N = 2$
 - ▶ $\log_2(2) = 1$, so one bit is required to store sign
- ▶ Example #2: If N is 200, then $x = \log_2(200) = 7.644$, so 8 bits are required

Stuffing numbers into a byte: Sign-Magnitude

- ▶ A byte has 8 bits
- ▶ One bit is used for the sign, 7 bits left
- ▶ Can store magnitudes from 0 to $2^7 = 127$
- ▶ Let MSB be sign bit
- ▶ Let other bits store magnitude
- ▶ Can store numbers from -127 to +127

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$+89_{10}$	0	1	0	1	1	0	0	1
-89_{10}	1	1	0	1	1	0	0	1
0_{10}	0	0	0	0	0	0	0	0
-0_{10}	1	0	0	0	0	0	0	0

Stuffing numbers into a byte: One's Complement

- ▶ Can store magnitudes from 0 to $2^7 = 127$
- ▶ Let MSB be sign bit
- ▶ Let other bits store magnitude
 - ▶ except if sign bit is set, magnitude must be *complemented* (i.e. inverted) to get actual value
 - ▶ one's complement of bit value x is $1 - x$, i.e. the same as $NOT(x)$
- ▶ Represents numbers from -127 to $+127$

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$+89_{10}$	0	1	0	1	1	0	0	1
-89_{10}	1	0	1	0	0	1	1	0
0_{10}	0	0	0	0	0	0	0	0
-0_{10}	1	1	1	1	1	1	1	1

Stuffing numbers into a byte: Two's Complement

- ▶ Can store magnitudes from 0 to $2^7 = 127$
- ▶ Let MSB be sign bit
- ▶ Let other bits store magnitude
 - ▶ To negate a number, complement all its bits and add 1
- ▶ Can store numbers from -128 to 127

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$+89_{10}$	0	1	0	1	1	0	0	1
-89_{10}	1	0	1	0	0	1	1	1
0_{10}	0	0	0	0	0	0	0	0
-0_{10}	0	0	0	0	0	0	0	0
-128_{10}	1	0	0	0	0	0	0	0

Integer Representations

- ▶ There is more than one way to represent the same integer
 - ▶ Sign-magnitude
 - ▶ One's complement
 - ▶ Two's complement
- ▶ Some of them are non-intuitive
 - ▶ negative and positive zeroes
 - ▶ asymmetric ranges $[-128, 127]$
- ▶ All of them have different hardware implications
 - ▶ Addition and subtraction circuits differ
- ▶ Generally, most computers you will encounter use two's-complement arithmetic

Integers in C

- ▶ Basic C types:

```
char a;  
short b; /* alternative form: short int */  
int c;  
long d; /* alternative form: long int */  
long long e;
```

- ▶ C implementations are required to provide a minimum size for each type
 - ▶ char must be at least 8 bits
 - ▶ int must be at least 16 bits
 - ▶ long must be at least 32 bits
 - ▶ long long must be at least 64 bits
- ▶ The prefix `unsigned` (e.g. `unsigned char`) allows all bits to be used to store the magnitude (i.e. there is no sign bit).
 - ▶ char must be able to store $[-127, 127]$
 - ▶ `unsigned char` must be able to store $[0, 255]$
 - ▶ Note C does *not* require machines implement two's complement

Fixed-width Integers in C99

```
#include <stdint.h>
int8_t a; /* signed 8-bit integer */
uint8_t ua; /* unsigned 8-bit integer */

int32_t b; /* signed 32-bit integer */
uint32_t ub; /* unsigned 32-bit integer */

...
```

- ▶ C99 is the C standard “version” 1999.
 - ▶ Finally allowed fixed-width types
 - ▶ Still does not mandate any particular representation
- ▶ The variables `INT8_MIN` and `INT8_MAX` contain the range for `int8_t`
 - ▶ similarly, `UINT8_MIN` and `UINT8_MAX` contain the range `uint8_t`

Summary

- ▶ Bits
- ▶ Functions operating on bits
- ▶ Multibit values and machine data types
- ▶ Storing Integers
- ▶ C Data Types

References and Next Class

- ▶ Today: Chapter 2 of the textbook
- ▶ Next class: Chapter 2 of the textbook
 - ▶ Data conversions
 - ▶ Bitwise Operations
 - ▶ Integer arithmetic