CS 33

Data Representation, Part 1

 Copyright © 2017 Thomas W. Doeppner. All rights reserved.

Many of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective." 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.

Number Representation

- Hindu-Arabic numerals
  - developed by Hindus starting in 5th century
    - » positional notation
    - » symbol for 0
  - adopted and modified somewhat later by Arabs
    - » known by them as “Rakam Al-Hind” (Hindu numeral system)
  - 1999 rather than MCMXCIX
    - » (try doing long division with Roman numerals!)

Copyright © 2017 Thomas W. Doeppner. All rights reserved.

Which Base?

- 1999
  - base 10
    - » $9 \cdot 10^0+9 \cdot 10^1+9 \cdot 10^2+1 \cdot 10^3$
  - base 2
    - » 11111001111
      - $1 \cdot 2^0+1 \cdot 2^1+1 \cdot 2^2+1 \cdot 2^3+0 \cdot 2^4+0 \cdot 2^5+1 \cdot 2^6+1 \cdot 2^7+1 \cdot 2^8+1 \cdot 2^9+1 \cdot 2^{10}$
  - base 8
    - » 3717
      - $7 \cdot 8^0+1 \cdot 8^1+7 \cdot 8^2+3 \cdot 8^3$
    - » why are we interested?
  - base 16
    - » 7CF
      - $15 \cdot 16^0+12 \cdot 16^1+7 \cdot 16^2$
    - » why are we interested?

 Copyright © 2017 Thomas W. Doeppner. All rights reserved.

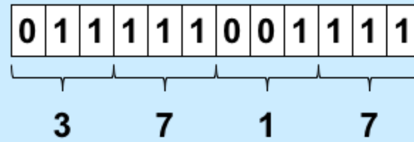
Base 2 is known as “binary” notation.

Base 8 is known as “octal” notation.

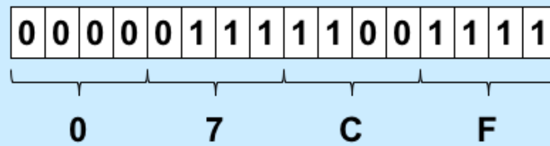
Base 10 is known as “decimal” notation.

Base 16 is known as “hexadecimal” notation. Note that “hexa” is derived from the Greek language and “decimal” is derived from the Latin language. Many people feel you shouldn’t mix languages when you invent words, but IBM, who coined the term “hexadecimal” in the 1960s, didn’t think their corporate image could withstand “sexadecimal”.

Words …



12-bit computer word



16-bit computer word

CS33 Intro to Computer Systems VII–4 Copyright © 2017 Thomas W. Doeppner. All rights reserved.

Note that a byte consists of two hexadecimal digits, which are sometimes known as "nibbles". A 32-bit computer word would then have eight nibbles; a 64-bit computer word would have sixteen nibbles.

Algorithm …

```
void baseX(unsigned int num, unsigned int base) {
   char digits[] = {'0', '1', '2', '3', '4', '5', '6', … };
   char buf[8*sizeof(unsigned int)+1];
   int i;

   for (i = sizeof(buf) - 2; i >= 0; i--) {
      buf[i] = digits[num%base];
      num /= base;
      if (num == 0)
         break;
   }

   buf[sizeof(buf) - 1] = '\0';
   printf("%s\n", &buf[i]);
}
```


Copyright © 2017 Thomas W. Doeppner. All rights reserved.

This routine prints the base *base* representation of *num*. The "%" operator yields the remainder. E.g., "10%3" evaluates to 1: the remainder after dividing 10 by 3. (Note that the "…" is not heretofore unexplained C syntax, but is shorthand for "fill this in to the extent needed.")

Or …

```
$ bc
obase=16
1999
7CF
$
```

 Copyright © 2017 Thomas W. Doeppner. All rights reserved.

"bc" (it stands for basic calculator, or perhaps better calculator) is a standard Unix command that handles arbitrary-precision arithmetic. Among its features is the ability to specify which base to use for input and output of numbers. The default base for both input and output is ten. Setting *obase* to 16 sets the base for output to 16. Similarly, one can change the base for input numbers by setting *ibase*.

Quiz 1

- What's the decimal (base 10) equivalent of 23_{16}?
  a) 19
  b) 33
  c) 35
  d) 37

Copyright © 2017 Thomas W. Doeppner. All rights reserved.

Encoding Byte Values

- Byte = 8 bits
  - binary 00000000_2 to 11111111_2
  - decimal: 0_{10} to 255_{10}
  - hexadecimal 00_{16} to FF_{16}
    - » base 16 number representation
    - » use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - » write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Supplied by CMU.

Boolean Algebra

- Developed by George Boole in 19th Century
  - algebraic representation of logic
    - » encode "true" as 1 and "false" as 0

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

		0	1
0	0	1
1	1	1

Not

- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

Supplied by CMU.

General Boolean Algebras

- Operate on bit vectors
  - operations applied bitwise

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
  01000001      01111101      00111100      10101010
```

- All of the properties of boolean algebra apply

Supplied by CMU.

Example: Representing & Manipulating Sets

- Representation
  - width-w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ iff $j \in A$

```
01101001        { 0, 3, 5, 6 }
76543210

01010101        { 0, 2, 4, 6 }
76543210
```

- Operations

&	intersection	01000001	{ 0, 6 }
|	union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	symmetric difference	00111100	{ 2, 3, 4, 5 }
~	complement	10101010	{ 1, 3, 5, 7 }

Supplied by CMU.

Bit-Level Operations in C

- Operations &, |, ~, ^ available in C
  - apply to any "integral" data type
    - » `long, int, short, char`
  - view arguments as bit vectors
  - arguments applied bit-wise
- Examples (char datatype)

```
~0x41 → 0xBE
  ~01000001₂ → 10111110₂
~0x00 → 0xFF
  ~00000000₂ → 11111111₂
0x69 & 0x55 → 0x41
  01101001₂ & 01010101₂ → 01000001₂
0x69 | 0x55 → 0x7D
  01101001₂ | 01010101₂ → 01111101₂
```


Supplied by CMU.

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - `&&, ||, !`
    - » view 0 as "false"
    - » anything nonzero as "true"
    - » always return 0 or 1
    - » early termination/short-circuited execution
- Examples (char datatype)

```
!0x41 → 0x00
!0x00 → 0x01
!!0x41 → 0x01

0x69 && 0x55 → 0x01
0x69 || 0x55 → 0x01
p && *p
```

(avoids null pointer access)

Supplied by CMU.

Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
    - » vie "false"

Watch out for && vs. & (and || vs. |)…
One of the more common oopsies in C programming

- E

```
!0x41 → 0x00
!0x00 → 0x01
!!0x41 → 0x01

0x69 && 0x55 → 0x01
0x69 || 0x55 → 0x01
p && *p
```

(avoids null pointer access)

Supplied by CMU.

Shift Operations

- **Left Shift: x << y**
  - shift bit-vector x left y positions
    - throw away extra bits on left
    - » fill with 0's on right
- **Right Shift: x >> y**
  - shift bit-vector x right y positions
    - » throw away extra bits on right
  - logical shift
    - » fill with 0's on left
  - arithmetic shift
    - » replicate most significant bit on left
- **Undefined Behavior**
  - shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Supplied by CMU.

Why we need both logical and arithmetic shifts should be clear by the end of the next lecture.

Unsigned Integers

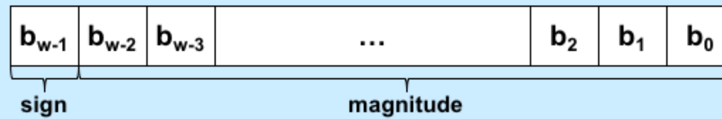
$$\text{value} = \sum_{i=0}^{w-1} b_i \cdot 2^i$$

 Copyright © 2017 Thomas W. Doeppner. All rights reserved.

If a computer word is to be interpreted as an unsigned integer, we can do so as shown in the slide, where w is the number of bits in the word.

Signed Integers

- **Sign-magnitude**



$$\text{value} = (-1)^{b_{w-1}} \cdot \sum_{i=0}^{w-2} b_i \cdot 2^i$$

- **two representations of zero!**
  - **computer must have two sets of instructions**
    - **one for signed arithmetic, one for unsigned**

Copyright © 2017 Thomas W. Doeppner. All rights reserved.

We might also want to interpret the contents of a computer word as a signed integer. There are a few options for how to do this. One straightforward approach is shown in the slide, where we use the high-order (leftmost) bit as the “sign bit”: 0 means positive and 1 means negative. However, this has the somewhat weird result that there are two representations of zero. This further means that the computer would have to have two implementations of arithmetic instructions: one for signed arithmetic, the other for unsigned arithmetic.

Signed Integers

- **Ones' complement**
  - **negate a number by forming its bitwise complement**
    - » **e.g., (-1)·01101011 = 10010100**

value = $-b_{w-1}(2^{w-1}-1)+\sum_{i=0}^{w-2} b_i \cdot 2^i$

$= \sum_{i=0}^{w-2} b_i \cdot 2^i$ if $b_{w-1} = 0$

$= \sum_{i=0}^{w-2} (b_i - 1) \cdot 2^i$ if $b_{w-1} = 1$

two zeroes!

Copyright © 2017 Thomas W. Doeppner. All rights reserved.

Note that the most-significant bit serves as the sign bit. But, as with sign-magnitude, the computer would need two sets of instructions: one for signed arithmetic and one for unsigned.

Signed Integers

- **Two's complement**

  $b_{w-1} = 0 \Rightarrow$ non-negative number

$$\text{value} = \sum_{i=0}^{w-2} b_i \cdot 2^i$$

  $b_{w-1} = 1 \Rightarrow$ negative number

$$\text{value} = (-1) \cdot 2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i$$

one zero!

Copyright © 2017 Thomas W. Doeppner. All rights reserved.

There's only one zero!

Two's complement is used on pretty much all of today's computers to represent signed integers.

Note that the high-order (most-significant) bit represents -2^{w-1}. All the other bits represent positive numbers.

Signed Integers

- **Negating two's complement**

$$value = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

  - **how to compute –*value*?**

    (~value)+1

Copyright © 2017 Thomas W. Doeppner. All rights reserved.

To negate a two's-complement number, simply complement each of its bits, then add one to the result. We show why this works in the next slide.

Signed Integers

- Negating two's complement (continued)

value **+ (~*value* + 1)**

= (*value* + ~*value*) + 1

= $(2^w - 1) + 1$

= 2^w

w

=

 Copyright © 2017 Thomas W. Doeppner. All rights reserved.

If we add to the two's complement representation of a w-bit number the result of adding one to its bitwise complement, we get a w+1-bit number whose low-order w bits are zeroes and whose high-order bit is one. However, since we're constrained to only w bits, the result is a w-bit value of all zeroes, plus an overflow. If we ignore the overflow, the result is zero.

Quiz 2

- We have a computer with 4-bit words that uses two's complement to represent negative numbers. What is the result of subtracting 0010 (2) from 0001 (1)?
  a) 0111
  b) 1001
  c) 1110
  d) 1111

 Copyright © 2017 Thomas W. Doeppner. All rights reserved.