CS 33

Data Representation

Copyright © 2012 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.

Today: Bits, Bytes, and Integers

- **Representing information as bits**
- Bit-level manipulations
- Integers
  - representation: unsigned and signed
  - conversion, casting
  - expanding, truncating
  - addition, negation, multiplication, shifting
  - summary
- Representations in memory, pointers, strings

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

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 © 2012 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
      - $1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 + 2^5 + 1 \cdot 2^6 + 1 \cdot 2^7 + 1 \cdot 2^8 + 1 \cdot 2^9 + 1 \cdot 2^{10}$
  - 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 © 2012 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 invented “hexadecimal” in the 1960s, didn’t think their corporate image could withstand “sexadecimal”.

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 © 2012 Thomas W. Doeppner. All rights reserved.

This routine prints the base *base* representation of *num*. (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 © 2012 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. Setting *obase* to 16 sets the base for output to 16. Similarly, one can change the base of input numbers by setting *ibase*.

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

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Data Representations

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

The chart gives the number of bytes used to represent the indicated C data types. A long double actually required just ten bytes of storage; however, due to alignment constraints (a topic we discuss later in the course), it is given 12 bytes (aligned to a 4-byte boundary) on the 32-bit IA32 and 16 bytes (aligned to an 8-byte boundary) on the 64-bit x86-64.

Supplied by CMU.

Today: Bits, Bytes, and Integers

- Representing information as bits
- **Bit-level manipulations**
- Integers
  - representation: unsigned and signed
  - conversion, casting
  - expanding, truncating
  - addition, negation, multiplication, shifting
  - summary
- Representations in memory, pointers, strings

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

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

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

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

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

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 }

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Bit-Level Operations in C

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

  ~0x41 → 0xBE

  $\sim 01000001_2 \rightarrow 10111110_2$

  ~0x00 → 0xFF

  $\sim 00000000_2 \rightarrow 11111111_2$

  0x69 & 0x55 → 0x41

  $01101001_2 \,\&\, 01010101_2 \rightarrow 01000001_2$

  0x69 | 0x55 → 0x7D

  $01101001_2 \mid 01010101_2 \rightarrow 01111101_2$

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

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)

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

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

- !
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41 → 0x01

  0x69 && 0x55 → 0x01
  0x69 || 0x55 → 0x01
  p && *p (avoids null pointer access)

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

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

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

The distinction between logical and arithmetic shifts should be clear by the end of this lecture.

Supplied by CMU.

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - **representation: unsigned and signed**
  - conversion, casting
  - expanding, truncating
  - addition, negation, multiplication, shifting
  - summary
- Representations in memory, pointers, strings
- Summary

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

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!**

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Signed Integers

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

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

$$\left.\begin{aligned} &= \sum_{i=0}^{w-2} b_i \cdot 2^i \quad \text{if } b_{w-1}=0 \\ &= \sum_{i=0}^{w-2} (b_i - 1)\cdot 2^i \quad \text{if } b_{w-1}=1 \end{aligned}\right\} \text{two zeroes!}$$

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Note that the most-significant bit serves as the sign bit.

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

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

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Note there's only one zero!

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

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

$$= -(\sim b_{w-1})2^{w-1} + \left(\sum_{i=0}^{w-2} (\sim b_i)2^i\right) + 1$$

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Signed Integers

- **Negating two's complement (continued)**

$$value + (\sim value + 1)$$

$$= -(b_{w-1} + \sim b_{w-1})2^{w-1} + \left(\sum_{i=0}^{w-2}(b_i + \sim b_i)2^i\right) + 1$$

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

$$= 2^w$$

$=$ 1 0 0 0 … 0 0 0 (w)

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

So, 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.

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1
- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- **Other Values**
  - Minus 1
    111…1

Values for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Values for Different Word Sizes

	W			
	8	16	32	64
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **Observations**
  - $|TMin| = TMax + 1$
    - » Asymmetric range
  - $UMax = 2 * TMax + 1$
- **C Programming**
  - **#include** <limits.h>
  - declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - values platform-specific

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Unsigned & Signed Numeric Values

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **Equivalence**
  - **same encodings for nonnegative values**
- **Uniqueness**
  - **every bit pattern represents unique integer value**
  - **each representable integer has unique bit encoding**
- **⇒ Can invert mappings**
  - **U2B(*x*) = B2U^{-1}(*x*)**
    - » **bit pattern for unsigned integer**
  - **T2B(*x*) = B2T^{-1}(*x*)**
    - » **bit pattern for two's comp integer**

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

B2U means "binary to unsigned". B2T means "binary to two's complement". Supplied by CMU.

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - representation: unsigned and signed
  - **conversion, casting**
  - expanding, truncating
  - addition, negation, multiplication, shifting
  - Summary
- representations in memory, pointers, strings

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Mapping Between Signed & Unsigned

- **Mappings between unsigned and two's complement numbers: keep bit representations and reinterpret**

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Mapping Signed ↔ Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5	→ T2U →	5
0110	6	← U2T ←	6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Mapping Signed ↔ Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3	=	3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5	+/- 16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Signed vs. Unsigned in C

- Constants
  - by default are considered to be signed integers
  - unsigned if have "U" as suffix

```
0U, 4294967259U
```

- Casting
  - explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - implicit casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```


Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Casting Surprises

- Expression evaluation
  - if there is a mix of unsigned and signed in single expression, *signed values implicitly cast to unsigned*
  - including comparison operations <, >, ==, <=, >=
  - examples for *W* = 32: TMIN = -2,147,483,648 , TMAX = 2,147,483,647

$Constant_1$	$Constant_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Summary
Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w

- Expression containing signed and unsigned int
  - `int` is cast to `unsigned`!!

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - representation: unsigned and signed
  - conversion, casting
  - **expanding, truncating**
  - addition, negation, multiplication, shifting
  - summary
- Representations in memory, pointers, strings

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Sign Extension

- **Task:**
  - given *w*-bit signed integer *x*
  - convert it to *w*+*k*-bit integer with same value
- **Rule:**
  - make *k* copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Sign Extension Example

```
short int x =  15213;
int      ix = (int) x;
short int y = -15213;
int      iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- **Converting from smaller to larger integer data type**
  - **C automatically performs sign extension**

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Summary: Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
  - unsigned: zeros added
  - signed: sign extension
  - both yield expected result
- Truncating (e.g., unsigned to unsigned short)
  - unsigned/signed: bits are truncated
  - result reinterpreted
  - unsigned: mod operation
  - signed: similar to mod
  - for small numbers yields expected behavior

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - representation: unsigned and signed
  - conversion, casting
  - Expanding, truncating
  - **addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits

- **Standard addition function**
  - **ignores carry output**
- **Implements modular arithmetic**

  $s = \text{UAdd}_w(u, v) = u + v \bmod 2^w$

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Visualizing (Mathematical) Integer Addition

- **Integer addition**
  - **4-bit integers *u*, *v***
  - **compute true sum $Add_4(u, v)$**
  - **values increase linearly with *u* and *v***
  - **forms planar surface**

$Add_4(u, v)$

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Visualizing Unsigned Addition

- **Wraps around**
  - **if true sum ≥ 2^w**
  - **at most once**

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits

- **TAdd and UAdd have identical bit-level behavior**
  - **signed vs. unsigned addition in C:**

```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v;
```

  - **will give** `s == t`

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

TAdd Overflow

- **Functionality**
  - **true sum requires w+1 bits**
  - **drop off MSB**
  - **treat remaining bits as 2's complement integer**

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Visualizing 2's Complement Addition

- **Values**
  - 4-bit two's comp.
  - range from -8 to +7
- **Wraps around**
  - if sum $\geq 2^{w-1}$
    - » becomes negative
    - » at most once
  - if sum $< -2^{w-1}$
    - » becomes positive
    - » at most once

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Multiplication

- **Goal: computing product of *w*-bit numbers *x*, *y***
  - **either signed or unsigned**
- **But, exact results can be bigger than *w* bits**
  - **unsigned: up to 2*w* bits**
    - » Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - **two's complement min (negative): up to 2*w*-1 bits**
    - » Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - **two's complement max (positive): up to 2*w* bits, but only for $(TMin_w)^2$**
    - » result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
- **So, maintaining exact results…**
  - **would need to keep expanding word size with each product computed**
  - **is done in software, if needed**
    - » **e.g., by "arbitrary precision" arithmetic packages**

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

The "bc" command on Linux provides arbitrary-precision arithmetic. Supplied by CMU.

Unsigned Multiplication in C

- **Standard multiplication function**
  - **ignores high order *w* bits**
- **Implements modular arithmetic**

  $\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Signed Multiplication in C

- **Standard multiplication function**
  - **ignores high order *w* bits**
  - **some of which are different for signed vs. unsigned multiplication**
  - **lower bits are the same**

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Power-of-2 Multiply with Shift

- **Operation**
  - `u << k` gives `u * 2`^{k}
  - both signed and unsigned

- **Examples**

```
u << 3 ==      u * 8
u << 5 - u << 3 == u * 24
```

  - most machines shift and add faster than multiply
    - » compiler generates this code automatically

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Unsigned Power-of-2 Divide with Shift

- **Quotient of unsigned by power of 2**
  - `u >> k` **gives** $\lfloor u / 2^k \rfloor$
  - **uses logical shift**

	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Signed Power-of-2 Divide with Shift

- **Quotient of signed by power of 2**
  - `x >> k` gives $\lfloor x / 2^k \rfloor$
  - uses arithmetic shift
  - rounds wrong direction when `x < 0`

	Division	Computed	Hex	Binary
`y`	-15213	-15213	C4 93	11000100 10010011
`y >> 1`	-7606.5	-7607	E2 49	11100010 01001001
`y >> 4`	-950.8125	-951	FC 49	11111100 01001001
`y >> 8`	-59.4257813	-60	FF C4	11111111 11000100

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Correct Power-of-2 Divide

- **Quotient of negative number by power of 2**
  - **want ⌈ `x / 2`^{k} ⌉ (round toward 0)**
  - **compute as ⌊ `(x+2`^{k}`-1) / 2`^{k} ⌋**
    - » **in C: `(x + (1<<k)-1) >> k`**
    - » **biases dividend toward 0**

Case 1: no rounding

Biasing has no effect

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Correct Power-of-2 Divide (Cont.)

Case 2: rounding

Biasing adds 1 to final result

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - representation: unsigned and signed
  - conversion, casting
  - expanding, truncating
  - addition, negation, multiplication, shifting
  - **summary**
- Representations in memory, pointers, strings

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Arithmetic: Basic Rules

- **Addition:**
  - unsigned/signed: normal addition followed by truncate, same operation on bit level
  - unsigned: addition mod 2^w
    - » mathematical addition + possible subtraction of 2^w
  - signed: modified addition mod 2^w (result in proper range)
    - » mathematical addition + possible addition or subtraction of 2^w
- **Multiplication:**
  - unsigned/signed: normal multiplication followed by truncate, same operation on bit level
  - unsigned: multiplication mod 2^w
  - signed: modified multiplication mod 2^w (result in proper range)

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Why Should I Use Unsigned?

- *Don't* use just because number nonnegative
  - easy to make mistakes

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

  - can be very subtle

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

- *Do* use when performing modular arithmetic
  - multiprecision arithmetic
- *Do* use when using bits to represent sets
  - logical right shift, no sign extension

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - representation: unsigned and signed
  - conversion, casting
  - expanding, truncating
  - addition, negation, multiplication, shifting
  - summary
- **Representations in memory, pointers, strings**

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Byte-Oriented Memory Organization

- Programs refer to data by address
  - conceptually, envision it as a very large array of bytes
    - » in reality, it's not, but can think of it that way
  - an address is like an index into that array
    - » and, a pointer variable stores an address
- Note: system provides private address spaces to each "process"
  - think of a process as a program being executed
  - so, a program can clobber its own data, but not that of others

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Machine Words

- Any given computer has a "word size"
  - nominal size of integer-valued data
    - » and of addresses
  - until recently, most machines used 32 bits (4 bytes) as word size
    - » limits addresses to 4GB (2^{32} bytes)
    - » become too small for memory-intensive applications
      - leading to emergence of computers with 64-bit word size
  - machines still support multiple data formats
    - » fractions or multiples of word size
    - » always integral number of bytes

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Word-Oriented Memory Organization

- Addresses specify byte locations
  - address of first byte in word
  - addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Byte Ordering

- Four-byte integer
  - 0x01234567
- Stored at location 0x100
  - which byte is at 0x100?
  - which byte is at 0x103?

?

67	45	23	01
0x100	0x101	0x102	0x103

Little-endian

01	23	45	67
0x100	0x101	0x102	0x103

Big-endian

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Here we have a four-byte integer one. In the big-endian representation, the address of the integer is the address of the byte containing its most-significant bits (the big end), while in the little-endian representation, the address of the integer is the address of the byte containing its least-significant bits (the little end). Suppose we pass a pointer to this integer to some procedure. However, in a type-mismatch, the procedure assumes that what is passed it is a two-byte integer. On a big-endian system, it would think it was passed a zero, but on a little-endian system, it would think it was passed a one.

This is not an argument in favor of either approach, but simply an observation that behaviors could be different.

Supplied by CMU.

Examining Data Representations

- **Code to print byte representation of data**
  - **casting pointer to unsigned char * allows treatment as a byte array**

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len){
  int i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n",start+i, start[i]);
  printf("\n");
}
```

printf format codes:

%p: Print pointer

%x: Print Hexadecimal

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Note: the "%.2x" in the format string means to print at least two characters of the value. Thus a value of 0 would be printed 00.

Supplied by CMU.

`show_bytes` Execution Example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux on IA32):

```
int a = 15213;
0xbf97669c  0x6d
0xbf97669d  0x3b
0xbf97669e  0x00
0xbf97669f  0x00
```


Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Representing Strings

```
char S[6] = "18243";
```

- **Strings in C**
  - represented by array of characters
  - each character encoded in ASCII format
    - » standard 7-bit encoding of character set
    - » character "0" has code 0x30
      - digit *i* has code 0x30+*i*
  - string should be null-terminated
    - » final character = 0
- **Compatibility**
  - byte ordering not an issue

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- **There are legions of smart people trying to find vulnerabilities in programs**

Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```


Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.

Malicious Usage

```
typedef unsigned int size_t;
...
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
```


Copyright © 2012 Thomas W. Doeppner. All rights reserved.

Supplied by CMU.