CS 270: Computer Organization

Bits, Bytes, and Integers

Instructor:
Professor Stephen P. Carl

Everything is Bits

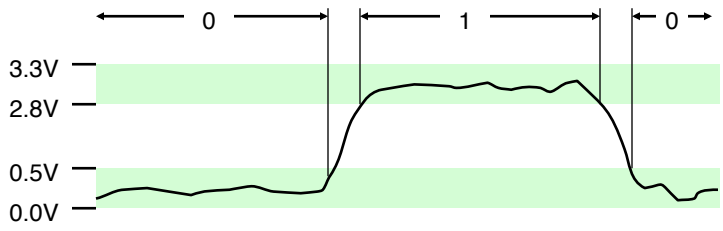
A *bit* (binary digit) is either 0 or 1

By Interpreting "strings" of bits we can

- Represent 15213_{10} as 11101101101101_2
- Represent 1.20_{10} as $1.0011001100110011[0011]\ldots_2$
- Represent individual machine instructions

Electronic Implementation of bits:

- Easy to store with bistable electronics
- Reliably transmitted on noisy and inaccurate wires



Base 10 (decimal) numbers

Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Example:

3271 =

$$(3x10^3) + (2x10^2) + (7x10^1) + (1x10^0)$$

Number Base B ⇒ B symbols per digit:

Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Base 2 (Binary): 0, 1

Base 2 (binary) numbers

Number representation:

$d_{31}d_{30} \ldots d_1d_0$ is a 32 digit number

value = $d_{31} \times 2^{31} + d_{30} \times 2^{30} + \ldots + d_1 \times 2^1 + d_0 \times 2^0$

Binary: 0,1 (A binary digit is called a bit)

In C, written as 0b… (e.g., 0b11011)

0b11010 $= 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$

$= 16 + 8 + 2$

$= 26$

Can we find a base that converts to binary easily?

Base 16 (hexadecimal) numbers

Hex digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Normal digits + 6 more from the alphabet

In C, written as 0x… (e.g., **0xFAB5** or **0xfab5**)

Conversion: Binary⇔Hex

1 hex digit represents 16 decimal values

4 binary digits represent 16 decimal values

⇒1 hex digit replaces 4 binary digits

One hex digit is a nibble; two make a byte

Example:

1010 1100 0011 (binary) = 0x_____ ?

Decimal vs. Hex vs. Binary

Examples:

$1010\ 1100\ 0011_2$ = 0xAC3

10111_2 = $0001\ 0111_2$ = 0x17

0x3F9 = $0011\ 1111\ 1001_2$

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

MEMORIZE!

Encoding Byte Values

Byte = 8 bits

Binary: **0000 0000**$_2$ to **1111 1111**$_2$

Decimal: **0**$_{10}$ to **255**$_{10}$

Note: In C, first digit must not be 0

Hexadecimal **00**$_{16}$ to **FF**$_{16}$

Base 16 number representation

Use characters '0' to '9' and 'A' to 'F'

Use subscripts when it isn't clear what representation is being used.

For example:

100 could be 100$_2$ (4), 100$_{10}$ (100), or 100$_{16}$ (256)

Machine Words

All Machines define a "Word Size", which is the nominal size of integer-valued data / machine addresses

- Until recently, most machines used 32 bit words (how many bytes?)
  - This limits physical address space to just over 4GB (2^{32} bytes)
  - Fast becoming too small for memory-intensive applications
- Newer systems use 64 bit words
  - **Intel IA64 and EMT64T (Itanium, Xeon, and others), AMD64 (Opteron and Athlon)**
  - Potential address space ≈ 1.8 X 10^{19} bytes (16,384 petabytes)
  - Our x86-64 machines support 48-bit addresses: 256 terabytes
- Machines support **multiple** data formats
  - Fractions or multiples of word size
  - Always **integral number** of bytes

Example Data Representations

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

Byte Ordering

How should bytes within multi-byte words be ordered in memory?

Two Conventions

- Big Endian: Sun, PPC Mac, Internet
  - Least significant byte has highest address
- Little Endian: Intel x86, MIPS
  - Least significant byte has lowest address

Former Sen. Dale Bumpers (D-Arkansas) allegedly proposed a third

Byte Ordering Example

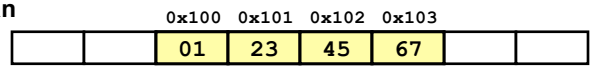
Big Endian - Least significant byte has highest address

Little Endian - Least significant byte has lowest address

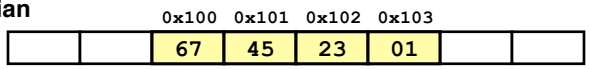
Example

- Variable `x` has 4-byte representation `0x01234567`
- Address given by `&x` is `0x100`

Big Endian



Little Endian



Examining Data Representations

Code to Print Byte Representation of Data

```
// Prints len 8-bit data items
// starting at start as a byte array
typedef unsigned char *pointer;

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

printf directives:

`%p`: Print pointer

`%x`: Print Hexadecimal

show_bytes Execution Example

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

Result (Linux):

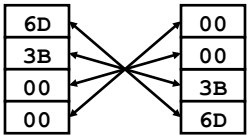
```
int a = 15213;
0x11ffffcb8   0x6d
0x11ffffcb9   0x3b
0x11ffffcba   0x00
0x11ffffcbb   0x00
```


Representing Integers

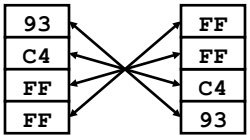
```
int A = 15213;
int B = -15213;
long int C = 15213;
```

Decimal: 15213
Binary: 0011 1011 0110 1101
Hex: 3 B 6 D

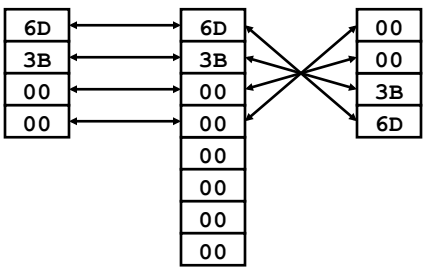
A : IA32/x86-64 Sun



B : IA32/x86-64 Sun



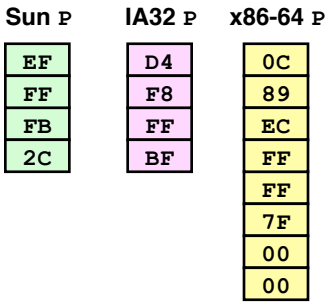
C : IA32 x86-64 Sun



Two's complement representation
(Covered later)

Representing Pointers

```
int B = -15213;
int *P = &B;
```



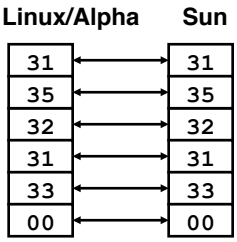
Different compilers & machines assign different locations to objects

Representing Strings

```
char S[6] = "15213";
```

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*
- Strings must be *null-terminated*
  - Final character = (char) 0



Compatibility

- Byte ordering not an issue

Boolean Algebra

Developed by George Boole in 19th Century, Boolean Algebra is an algebraic representation of logic, which encodes "True" as 1 and "False" as 0

Basic operations: AND, OR, NOT, XOR

Given boolean variables P, Q, the operations mean the following:

- P AND Q -> 1 if both P and Q are 1, and 0 otherwise
- P OR Q -> 1 if either P or Q are 1, and 0 otherwise
- NOT P -> 1 if P is 0, and 0 otherwise
- P XOR Q -> 1 if P and Q are different, and 0 otherwise

Truth Tables in Boolean Algebra

And is represented by '&' in C:

p & q = 1 when both p=1 and q=1

&	0	1
0	0	0
1	0	1

Or is represented by 'I' in C:

p I q = 1 when either p=1 or q=1

I	0	1
0	0	1
1	1	1

Boolean Algebra

Not is represented by the '~' in C:

~p = 1 when p=0

~	
0	1
1	0

Exclusive-Or (Xor) is the '^' in C:

p^q = 1 when either p=1 or q=1, but not both

^	0	1
0	0	1
1	1	0

Application of Boolean Algebra

Boolean algebra was first applied to Digital Systems by Claude Shannon in his MIT Master's Thesis, which showed that boolean operations could be modeled in electronic circuits (1937).

The stage was set for building computers out of digital electronics.

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

Representing & Manipulating Sets

Representation

- A ***w***-bit vector can represent subsets of **{0, …, *w*–1}**
- a_j **= 1** if $j \in A$

```
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 }**

Bit-Level Operations in C

Operations &, |, ~, ^ all available in C

- Can apply to any "integral" data type
  - `long, int, short, char, unsigned`
- View each argument as a bit vector
- Operations applied bit-wise

Examples (using char data type)

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


Contrast: Logic Operations in C

Recall the Logical Operators &&, ||, and !

- View 0 as "False"
- View all nonzero values as "True"
- Always return 0 or 1
- Early termination (short circuit)

Examples (using char data type)

```
!0x41  -->  0x00
!0x00  -->  0x01
!!0x41 -->  0x01

0x69 && 0x55  -->  0x01
0x69 || 0x55  -->  0x01
p && *p       (avoids null pointer access)
```

Shift Operations

Left Shift `x << y` means:

- Shifts bit-vector `x` left `y` positions
- Throw away extra bits on left
- Fill with 0's on right

Right Shift `x >> y` means:

- Shift bit-vector `x` right `y` positions
- Throw away extra bits on right
- *Logical shift* fill with 0's on left
- *Arithmetic shift* replicates the most significant bit on right

Shift is Undefined if:

- Shift amount < 0 or ≥ word size

Examples of using Shift

Variable x is:	01100010
x << 3	00010*000*
x >> 2 (logical)	*00*011000
x >> 2 (arith.)	*00*011000

Argument x	10100010
x << 3	00010*000*
x >> 2 (logical)	*00*101000
x >> 2 (arith.)	*11*101000