CCSYA

Data Representation

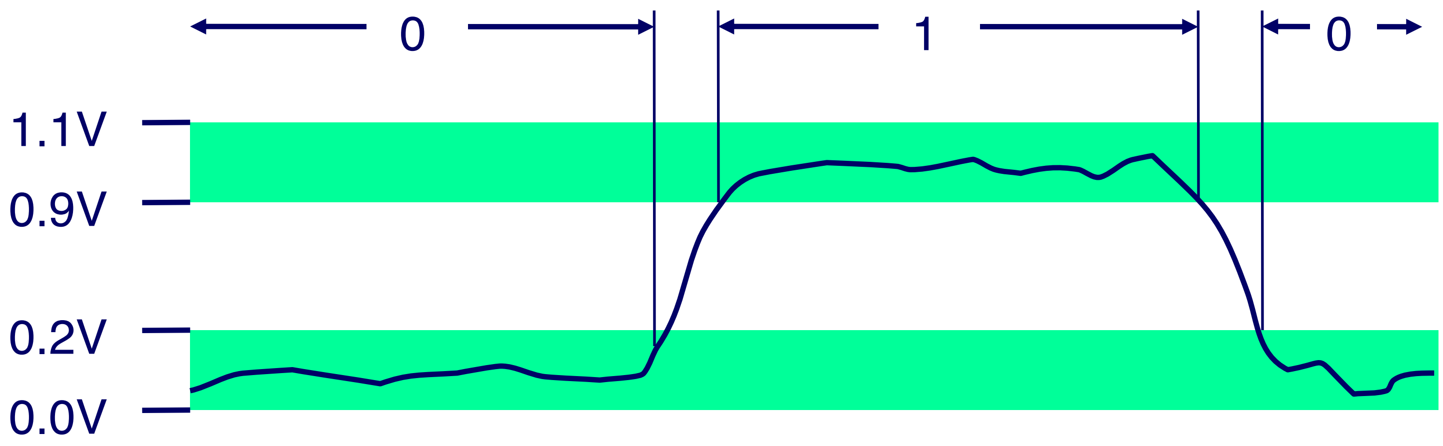
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

Everything is Bits

- **Binary digits (*bits*) form the basis of the digital revolution**
  - Each bit is 0 or 1

- **Why bits?**
  - Digital transistors operate in high and low voltage ranges
  - Voltage range dictates binary value on wire
  - Reliably transmitted on noisy and inaccurate wires



Everything is Bits

- **In isolation, a single bit is not very useful but, by encoding/interpreting sets of bits in various ways**
  - Computers determine what to do (instructions)
  - … and represent and manipulate numbers, sets, strings, etc…
- **Most computers use blocks of eight bits, or bytes, as the smallest addressable unit of memory**
  - Common sizes: 1, 2, 4, 8, or 16 bytes

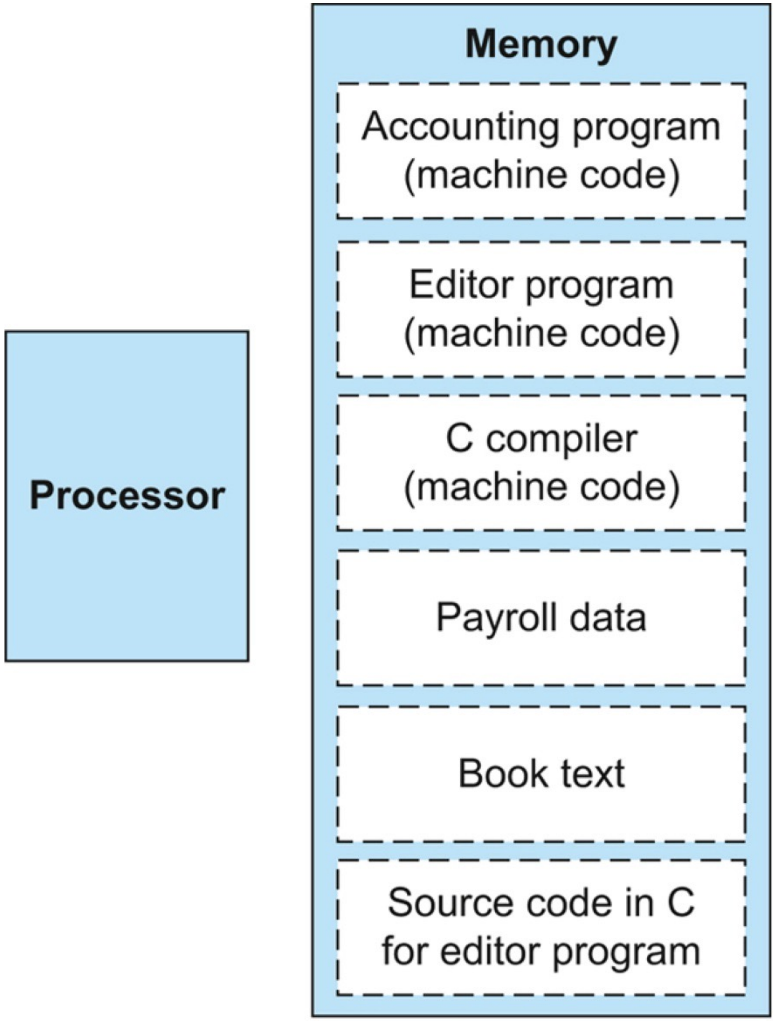
Encoding Byte Values

- **A byte value can be interpreted in many ways!**
  - Depends upon how it’s used
- **For example, consider a byte with value $01010101_{(2)}$**
  - As text: ‘U’
  - As integer: $85_{(10)}$
  - As a IA32 instruction: pushl %ebp
  - As part of an address or real number
  - As a medium gray pixel in a gray-scale image
  - Could be interpreted in MANY other ways…

Stored Program Concept

- **Modern computers are built on two key principles:**
  - Instructions are represented as numbers
  - Programs are stored in memory to be read or written, just like data
- **These principles lead to the stored program concept**
  - No distinction between data and program in memory
  - Programs are shipped as files of binary numbers
  - Computers can inherit ready-made software provided they are compatible with an existing instruction set

Stored Program Concept



Using Bits to Represent Numbers

- **Just like decimal except there are only two digits**
  - 0 and 1
- **Everything is based on powers of 2 (1, 2, 4, 8, 16, ...)**
  - Instead of powers of 10 (1, 10, 100, 1000, ...)
- **Binary numbers are sums of powers of 2**
  - $11011_{(2)} = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
    $= 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$
    $= 27_{(10)}$
  - $11101101101101_{(2)} = 15213_{(10)}$
  - $1.1101101101101_{(2)} \text{ X } 2^{13} = 1.5213 \text{ X } 10^4{}_{(10)}$

Using Bits to Represent Numbers

- **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’

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

15213: $\underbrace{0011}_{3}\ \underbrace{1011}_{B}\ \underbrace{0110}_{6}\ \underbrace{1101}_{D}$

Unsigned Binary Integers

- **Given an n-bit number:**

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- **Range: 0 to $+2^n - 1$**
  - Using 32 bits
    - 0 to +4,294,967,295

- **Example**
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

2s-Complement Signed Integers

- **Given an n-bit number**

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- **Range: -2^{n-1} to $+2^{n-1} - 1$**
  - Using 32 bits
    - –2,147,483,648 to +2,147,483,647
- **Example**
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

2s-Complement Signed Integers

- **Most significant bit is a sign bit**
  - 1 for negative numbers
  - 0 for non-negative numbers
- **$-(-2^{n-1})$ can't be represented**
- **Non-negative numbers have the same unsigned and 2s-complement representation**
- **Some specific numbers**
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

2s-Complement Signed Integers

- **Complement and add 1**
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- **Example: negate +2**
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

Data representations

- Computers use a **limited number of bits** to encode a number
- Hence, some **operations can *overflow/underflow*** when the results are too large to be represented

C data type	Intel IA32	x86-64
char	1	1
short	2	2
int	4	4
long	4	8
long long	8	8
float	4	4
double	8	8
long double	10/12	10/16
pointer	4	8

Other Types of Data

- **Digital representation means that everything is represented by numbers only**
  - Text, code, sound, pictures, …

- **For sound, pictures, other “real-world” values**
  - Make accurate measurements
  - Convert them to numeric values

- **The usual sequence:**
  - Data is converted into numbers by some mechanism
  - Numbers can be stored, retrieved, processed, transmitted
  - Numbers might be reconstituted into a version of the original

Using Bits to Represent Characters

- **Each character encoded in ASCII format**
  - American Standard Code for Information Interchange
  - Standard 7-bit encoding of character set
- **Each value between 0 and 127 represents a specific character**
- **Most computers extend the ASCII character set to use the full range of 256 characters available in a byte**
  - The upper 128 characters handle special things like accented characters

Using Bits to Represent Characters

<table>
<tr><th></th><th>000</th><th>001</th><th>010</th><th>011</th><th>100</th><th>101</th><th>110</th><th>111</th></tr>
<tr><td>0000</td><td>NULL</td><td>DLE</td><td></td><td>0</td><td>@</td><td>P</td><td>`</td><td>p</td></tr>
<tr><td>0001</td><td>SOH</td><td>DC1</td><td>!</td><td>1</td><td>A</td><td>Q</td><td>a</td><td>q</td></tr>
<tr><td>0010</td><td>STX</td><td>DC2</td><td>"</td><td>2</td><td>B</td><td>R</td><td>b</td><td>r</td></tr>
<tr><td>0011</td><td>ETX</td><td>DC3</td><td>#</td><td>3</td><td>C</td><td>S</td><td>c</td><td>s</td></tr>
<tr><td>0100</td><td>EDT</td><td>DC4</td><td>$</td><td>4</td><td>D</td><td>T</td><td>d</td><td>t</td></tr>
<tr><td>0101</td><td>ENQ</td><td>NAK</td><td>%</td><td>5</td><td>E</td><td>U</td><td>e</td><td>u</td></tr>
<tr><td>0110</td><td>ACK</td><td>SYN</td><td>&</td><td>6</td><td>F</td><td>V</td><td>f</td><td>v</td></tr>
<tr><td>0111</td><td>BEL</td><td>ETB</td><td>'</td><td>7</td><td>G</td><td>W</td><td>g</td><td>w</td></tr>
<tr><td>1000</td><td>BS</td><td>CAN</td><td>(</td><td>8</td><td>H</td><td>X</td><td>h</td><td>x</td></tr>
<tr><td>1001</td><td>HT</td><td>EM</td><td>)</td><td>9</td><td>I</td><td>Y</td><td>i</td><td>y</td></tr>
<tr><td>1010</td><td>LF</td><td>SUB</td><td>*</td><td>:</td><td>J</td><td>Z</td><td>j</td><td>z</td></tr>
<tr><td>1011</td><td>VT</td><td>ESC</td><td>+</td><td>;</td><td>K</td><td>[</td><td>k</td><td>{</td></tr>
<tr><td>1100</td><td>FF</td><td>FS</td><td>,</td><td><</td><td>L</td><td>\</td><td>l</td><td>|</td></tr>
<tr><td>1101</td><td>CR</td><td>GS</td><td>-</td><td>=</td><td>M</td><td>]</td><td>m</td><td>}</td></tr>
<tr><td>1110</td><td>SO</td><td>RS</td><td>.</td><td>></td><td>N</td><td>^</td><td>n</td><td>~</td></tr>
<tr><td>1111</td><td>SI</td><td>US</td><td>/</td><td>?</td><td>O</td><td>_</td><td>o</td><td>DEL</td></tr>
</table>

Using Bits to Represent Code

- **A program, from the perspective of the machine, is simply a sequence of bytes**
  - It has no information about the original source program (except some auxiliary tables maintained to aid in debugging)

- **Consider the following C function:**

```
int sum(int x, int y){
  return x + y;
}
```

Using Bits to Represent Code

- **When compiled on a set of sample machines, we generate machine code having the following byte representations**

Linux 32:	55 89 e5 8b 45 0c 03 45 08 c9 c3
Windows:	55 89 e5 8b 45 0c 03 45 08 5d c3
Sun:	81 c3 e0 08 90 02 00 09
Linux 64:	55 48 89 e5 89 7d fc 89 75 f8 03 45 fc c9 c3

- **Different machine types use different and incompatible instructions and encodings**
  - Even identical processors running different OSes have differences in their coding conventions and hence are not binary compatible

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

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
  - Encode “True” as 1 and “False” as 0

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

Extending Boolean Algebra

- **Operate on bit vectors: operations applied bitwise**

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

- **Bitwise operations have many properties in common with integer arithmetic**
  - & → Intersection
  - | → Union
  - ^ → Symmetric difference
  - ~ → Complement

Bit-level Operations in C

- **Operations &, |, ~, ^ available in C**
  - Apply to any "integral" data type (`signed` and `unsigned`)
    - `long, int, short, char, …`
  - View arguments as bit vectors
  - Arguments applied bit-wise

Examples (char data type)

```
~0x41(16)                         →  0xBE(16)
~01000001(2)                      →  10111110(2)
~0x00(16)                         →  0xFF(16)
~00000000(2)                      →  11111111(2)

0x69(16)     &  0x55(16)          →  0x41(16)
01101001(2)  &  01010101(2)       →  01000001(2)
0x69(16)     |  0x55(16)          →  0x7D(16)
01101001(2)  |  01010101(2)       →  01111101(2)
```

Contrast: Logic Operations in C

- **Contrast to logical operators &&, ||, !**
  - View 0 as “False”
  - Anything nonzero as “True”
  - Always return 0 or 1
  - **Early termination**

Examples (char data type)

```
!0x41           →   0x00  (false)
!0x00           →   0x01  (true)
!!0x41          →   0x01  (true)

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

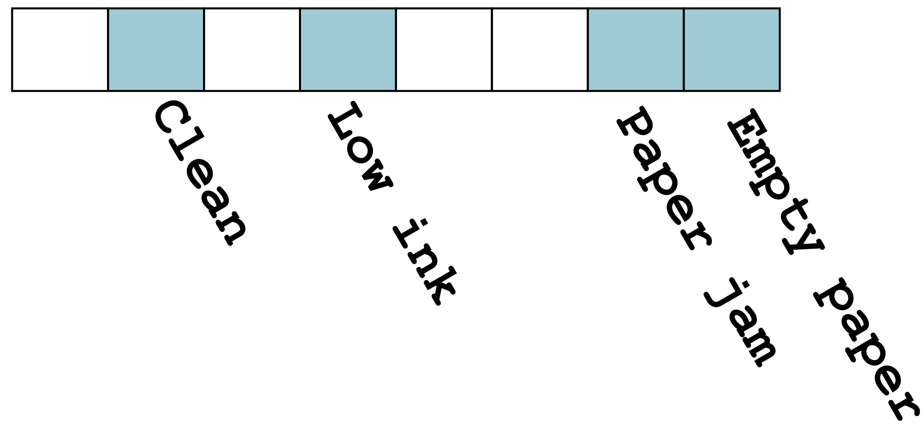
Bit-level Operations

- **One common use of bit-level operations is to implement *masking* operations**
  - A *mask* is a bit pattern that indicates a selected set of bits within a word

- **Very useful in several practical applications**
  - IP addressing and subnetting
  - Hash tables
  - Controlling devices
  - Image processing
  - …

Example: Printer Status Register

```c
#define EMPTY   01
#define JAM     02
#define LOW_INK 16
#define CLEAN   64
```




```c
char status;

if (status == (EMPTY | JAM)) ...;
if (status == EMPTY || status == JAM) ...;
while (!(status & LOW_INK)) ...;

status |= CLEAN;  /* turns on CLEAN bit */
status &= ~JAM;/* turns off JAM bit */
```

Shift operations

- **C also provides a set of shift operations for shifting bit patterns to the left and to the right**
- **Arithmetic operators have precedence over shifts**
  - Getting the precedence wrong in C expressions is a common source of program errors, and often these are difficult to spot by inspection
- **For a n-word size value, the shift amount should be a value between 0 and n − 1**
  - **Undefined behavior** when shift amount < 0 or ≥ word size

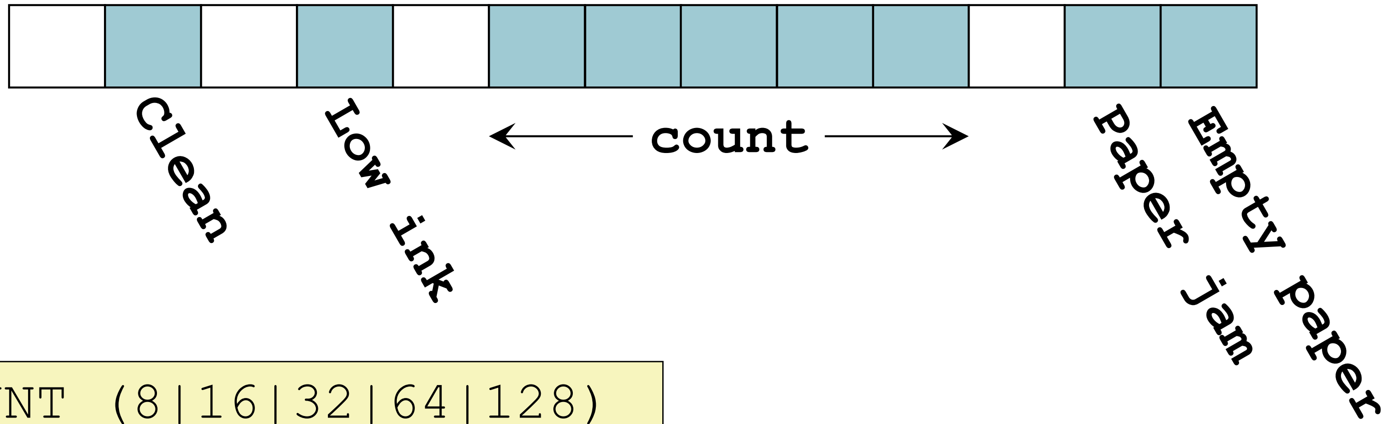
Shift operations

- **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**

Argument **`x`**	`01100010`
`<< 3`	`00010000`
Log. `>> 2`	`00011000`
Arith. `>> 2`	`00011000`

Argument **`x`**	`10100010`
`<< 3`	`00010000`
Log. `>> 2`	`00101000`
Arith. `>> 2`	`11101000`

Example: Printer status register

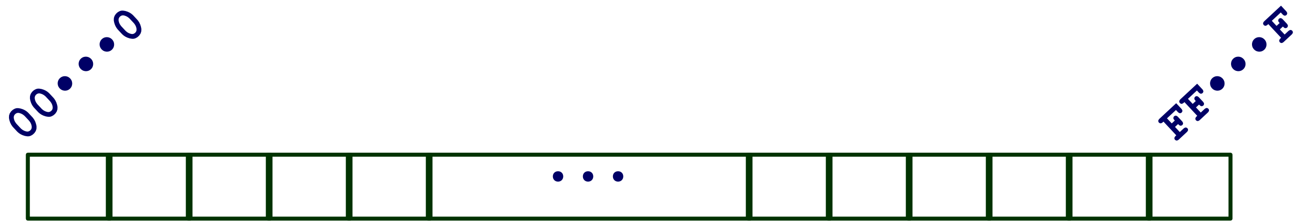



```
#define COUNT (8|16|32|64|128)
```

```
/* extract to c */
unsigned int c = (status & COUNT) >> 3;

/* insert v */
status = ((v << 3) | ~COUNT) | (status & ~COUNT);
```

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

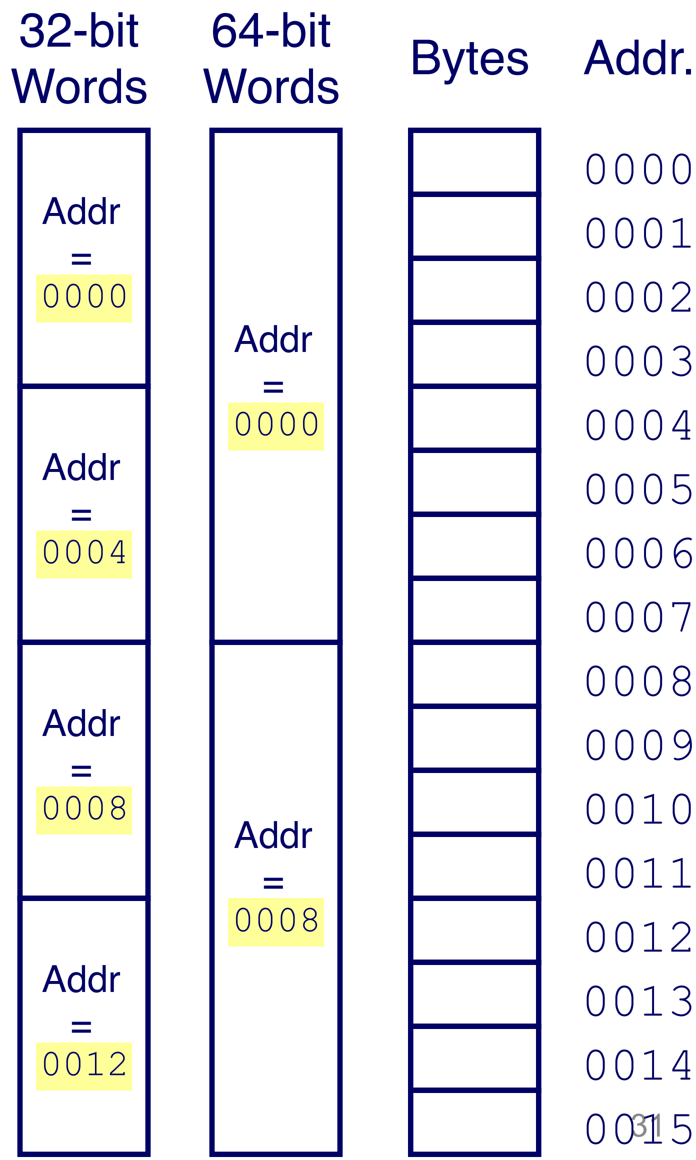
- **The OS provides private address spaces to each process**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

Machine Words

- **Any given computer has a “word size”**
  - Nominal size of integer-valued data and of addresses
- **Until recently, most machines used 32-bit word size**
  - Limits addresses to 4GB (2^{32} bytes)
- **Increasingly, machines have 64-bit word size**
  - Potentially, could have 18 PB (petabytes) of addressable memory
  - That’s 18.4 X 10^{15}
- **Machines still support multiple data formats**
  - Fractions or multiples of word size
  - Always integral number of bytes

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)**



Pointers in C

- **A** **pointer** **is a reference to another variable (memory location) in a program**

```
int b = -15213;
int *p1 = &b;
char *p2 = (char*)&b;
```

- **The value of a pointer in C is the address of the first byte of some block of storage to which it points to**
  - Whether it points to an integer, a structure, or some other program object

Pointers in C

- **Pointer type impacts pointer arithmetic and the number of bytes that are written/read to/from memory**
  - The computed value is scaled according to the size of the data type referenced by the pointer

```
int b = -15213;
int *p1 = &b;              // p1+1 increments 4 bytes
char *p2 = (char*)&b;      // p2+1 increments 1 byte
```

- **Pointer size is determined by the the full word size of the machine**
  - 4 bytes in 32-bit architectures, 8 bytes in 64-bit ones

Byte Ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**

- **Big Endian convention**
  - Least significant byte has highest address
  - Adopted by Sun, PPC Mac, Internet

- **Little Endian convention**
  - Least significant byte has lowest address
  - Adopted by x86, ARM processors running Android, iOS, and Windows

Byte Ordering Example

- Assume `int x` has value of 0x01234567…
- … and the address given by `&x` is 0x100

Big Endian

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

Little Endian

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

Examining Data Representations

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

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

printf directives:
%p: Print pointer
%x: Print hexadecimal

Example: `show_bytes` Execution

```
int a = 15213; /* 0x3B6D */
show_bytes((unsigned char*) &a, sizeof(int));
```

Output (Linux/x86-64):

```
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

Concluding Remarks

- **Computers encode information as bits, generally organized as sequences of bytes**
- **Different models of computers use different conventions for encoding numbers and for ordering the bytes within multi-byte data**
- **High-level languages are designed to accommodate a wide range of word sizes and numeric encodings**
  - Understanding these encodings at the bit level is important for writing programs that operate correctly over the full range of numeric values