Bits, Bytes and Integers

Computer Systems Organization (Spring 2016)
CSCI-UA 201, Section 2

Instructor: Joanna Klukowska

Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

Everything is a bit

- Each bit is 0 or 1 (well, at least on our human interpretation of a bit)

- Everything on a computer is encoded as sets of bits:
  - programs: all instructions of a program stored on disk and running are represented using binary sequences
  - data: the data that programs are manipulating are represented using binary sequences (numbers, strings, characters, images, audio, ...)

- Why bits? Why binary system and not base 3, or base 4, or base 10?
  - Electronic implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

More than a byte: KB, MB, GB, ...

Confusion due to binary and decimal uses of Kilo-, Mega-, Giga- prefixes.
In this course, we will be using them in binary sense.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB</td>
<td>1024 bytes = 1,024 bytes</td>
</tr>
<tr>
<td>1 MB</td>
<td>1048,576 bytes = 1,048,576 bytes</td>
</tr>
<tr>
<td>1 GB</td>
<td>1,073,741,824 bytes</td>
</tr>
<tr>
<td>1 TB</td>
<td>1,099,511,627,716 bytes</td>
</tr>
<tr>
<td>1 KB</td>
<td>= 10^3 bytes = 1,000 bytes</td>
</tr>
<tr>
<td>1 MB</td>
<td>= 10^6 bytes = 1,000,000 bytes</td>
</tr>
<tr>
<td>1 GB</td>
<td>= 10^9 bytes = 1,000,000,000 bytes</td>
</tr>
<tr>
<td>1 TB</td>
<td>= 10^12 bytes = 1,000,000,000,000 bytes</td>
</tr>
</tbody>
</table>

Bytes

1 byte = 8 bits

Range of representable values:
- binary: 00000000, to 11111111
- decimal: 0_10 to 255_10
- hexadecimal: 00_16 to FF_16

Hexadecimal: shorthand notation for binary (easier to write one symbol than four)
- Base 16 number representation
- Use characters 0 to 9 and A to F
- Write FA/1D37B_16 in C as
  - 0xFA/1D37B
  - 0x1D37B

DNHI:

Convert the following numbers to the other two representations (by hand) and then write a C program that does the same:

Binary: 101101010, 10001110010, 101010110001, 100001
Decimal: 255, 64, 1025
Hexadecimal: 0xab, 0x123, 0xff, 0xf0
Boolean algebra

Developed by George Boole in 19th century for logical operations. Claude E. Shannon (1916–2001) adapted this concept to electrical switches and relays; this eventually lead to our computers “speaking” binary.

### AND

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### OR

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### NOT (complement)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

### XOR

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Bit-vector operations using Boolean operators

To operate on vectors of bits, a Boolean operation is applied to bits at corresponding positions.

**Example:**

- 1100 & 0011 = 0010
- 0110 | 1010 = 1110
- 0110 ^ 1010 = 1100

These operators in C are called bitwise operators.

**Warning:**

do not confuse bit-wise operators (&, |, ^) with logical operators (&&, ||, !)

---

Bit-vector operations using shift operators

**Left Shift:** \( x << y \)

- Shift bit-vector \( x \) left by \( y \) positions
- Throw away extra bits on left
- Fill with 0’s on right

**Right Shift:** \( x >> y \)

- Shift bit-vector \( x \) right by \( y \) positions
- Throw away extra bits on right
- Logical shift: fill with 0’s on left
- Arithmetic shift: replicate most significant bit on left

**Example 1:** \( x \) is 01100010

\( x << 3 \) is 00010000

(both) \( x >> 2 \) is 00010100

**Example 2:** \( x \) is 10100010

\( x << 3 \) is 00010000

(both) \( x >> 2 \) is 11101000

**Warning:** shifting the a value < 0 or >= word size is undefined in C.

---

Swapping values of variables without a tmp

Swapping values of two variables normally requires a temporary storage.

Using the bitwise exclusive or operator we can actually do this using only the storage of the two bit-vectors.

```c
void inplace_swap ( int * x, int * y ) {
    *x = *x ^ *y;
    *y = *x ^ *y;
    *x = *x ^ *y;
}
```

**DNH:** Try it on paper with several different values to convince yourself that this works and how.

---

Encoding of Integers

**Unsigned:**

\[
B2U(X) = \sum_{i=0}^{n-1} x_i \cdot 2^i
\]

**Two’s Complement (Signed):**

\[
B2T(X) = -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i
\]

**Example:** C short is 2 bytes long

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>15213</td>
<td>3b 4d 00111011 01101111</td>
</tr>
<tr>
<td>y</td>
<td>-15213</td>
<td>C4 93 11000100 10010011</td>
</tr>
</tbody>
</table>

**Sign bit:** for 2’s complement notation it is the most significant bit (leftmost)

- 0 indicates non-negative number
- 1 indicates negative number

**ToDo:** try these formulas for \( w=5 \) just for practice. What is the integer encoded by 01011, 10010 using both unsigned and two’s complement encodings?
Numerical Ranges

Unsigned:
- $U_{\text{min}} = 0$
- $U_{\text{max}} = 2^n - 1$

Two's Complement:
- $T_{\text{min}} = -2^{n-1}$
- $T_{\text{max}} = 2^n - 1$

Assume $n = 5$:
- smallest unsigned: $00000_2 = 0_{10}$
- largest unsigned: $11111_2 = 31_{10}$

Assume $n = 5$:
- smallest 2's comp: $10000_2 = -16_{10}$
- largest 2's comp: $01111_2 = 15_{10}$

Umax, Tmin, Tmax for standard word sizes

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_{\text{max}}$</td>
<td>255</td>
<td>65,535</td>
<td>4,294,967,295</td>
<td>18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>$T_{\text{max}}$</td>
<td>127</td>
<td>32,767</td>
<td>2,147,483,647</td>
<td>9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>$T_{\text{min}}$</td>
<td>128</td>
<td>32,767</td>
<td>2,147,483,648</td>
<td>9,223,372,036,854,775,808</td>
</tr>
</tbody>
</table>

Notice:
- the range of 2's complement values is not symmetric
  - $|T_{\text{min}}| = |T_{\text{max}}| + 1$
- for a given value of $n$
  - $U_{\text{max}} = 2 \times T_{\text{max}} + 1$

In C:
- To access the values of largest/smallest values use `#include <climits.h>`
- The constants are named
  - INT_MAX
  - INT_MIN
  - UINT_MAX
  - INT_MIN
  - (these numbers are system specific)

Comparison of Unsigned and Two's Comp.

<table>
<thead>
<tr>
<th>$X$</th>
<th>$B2U(X)$</th>
<th>$B2T(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0110</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0111</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>-1</td>
</tr>
</tbody>
</table>

Equivalence:
- Same encodings for nonnegative values
- $+/-16$ for negative 2's comp and positive unsigned

Uniqueness:
- Every bit pattern represents unique integer value
- Each representable integer has unique bit encoding

Conversion Between Signed and Unsigned Values

- not always a good idea

Signed and Unsigned in C

Constants
- By default, signed integers
- Unsigned with "U" as suffix: 0U, 4294967256U

Casting
- Explicit casting between signed & unsigned
  - $x = \text{unsigned}(y)$
  - $y = \text{int}(x)$
- Implicit casting also occurs via assignments and function calls
  - $x = y$
  - $y = x$

Casting Surprises

If there is a mix of unsigned and signed in single expression, signed values implicitly cast to unsigned, including expressions with logical comparison operations $<, >, <=, >=$.

What will the following code print?

```c
if ( -1 < 0 )
    printf ("-1 < 0 is true" );
else
    printf ("-1 < 0 is false" );
if ( -1 < 0U )
    printf ("-1 < 0U is true" );
else
    printf ("-1 < 0U is false" );
```
DNHL: What does this code do?

1) Array indexes are always non-negative. So it should be a good idea to use
unsigned values to represent them. For example:

```c
unsigned i;
short a[10] = {1,2,3,4,5,6,7,8,9,10};
for (i = 9; i >= 0 ; i--) 
    printf("%d, ", a[i]);
printf("\n");
```

What do you think, the above code fragment will do? Test it in a program.

2) Here is another program that seems like it should work. What does this do?

```c
int i;
short a[10] = {1,2,3,4,5,6,7,8,9,10};
for (i = 0; i < sizeof(char) >= 0 ; i--) 
    printf("%d, ", a[i]);
printf("\n");
```

Expanding, truncating

Sign Extension

**TASK:** Given a w-bit signed integer $X$, convert it to (w+$k$)-bit integer $X'$ with
the same value.

**Solution:** make $k$ copies of the sign bit

$$X = x_{w-1} \cdots x_{w}, x_0$$

$$X' = x_0 \cdots x_{w-1} x_{w-1} \cdots x_0$$

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

Sign Extension

C automatically performs sign extension when converting from "smaller" to "larger"
data type.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$3b$</td>
<td>0011101101101</td>
</tr>
<tr>
<td>$ix$</td>
<td>$00000000$</td>
<td>000000000000111011101101</td>
</tr>
<tr>
<td>$y$</td>
<td>$-15213$</td>
<td>C4 93</td>
</tr>
<tr>
<td>$iy$</td>
<td>FF FF C4 93</td>
<td>11111111 11111111 110000100 10010011</td>
</tr>
</tbody>
</table>

Truncating

- **Example:** from int to short (i.e. from 32-bit to 16-bit)
- High-order bits are truncated
- Value is altered and must be reinterpreted
- This (non-intuitive) behavior can lead to buggy code!

Example:

```c
int i = 1572539;
short si = (short) i;
printf("\n\n", i, si);
```

prints

```
i = 1572539
si = -325
```
Negation

Task: given a bit-vector $x$ compute $-x$

Solution: $-x = -x + 1$

(negating a value can be done by computing its complement and adding 1)

Example: $x = 011010_2 = 26_{10}$

$-x = 100101_2 = -26_{10}$

$-x + 1 = 100110_2 = -25_{10}$

Notice that for any signed integer $x$, we have $-x + x = 111...11 = -1_{2}$

Addition of signed numbers

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: keep $w$ bits

- True sum requires $w+1$ bits, addition ignores the carry bit.
- If $\text{TAdd}(u, v)$ $> 2^{w-1}$, then sum becomes negative (positive overflow)
- If $\text{TAdd}(u, v) < -2^{w-1}$, then sum becomes positive (negative overflow)

\[
\begin{align*}
10010_2 & = -14_{10} \\
+ 11011_2 & = -5_{10} \\
\hline
101111_2 & = 13_{10} \\
01101_2 & = 13_{10}
\end{align*}
\]

Addition for unsigned numbers

Operands: $w$ bits

True Sum: $w+1$ bits

Discard Carry: keep $w$ bits

- Standard addition function ignores carry bits and implements modular arithmetic:
  \[ \text{UAdd}(u, v) = (u + v) \mod 2^w \]

\[
\begin{align*}
10010_2 & = 18_{10} \\
* 11011_2 & = 27_{10} \\
\hline
101111_2 & = 45_{10} \\
01101_2 & = 13_{10} & 45_{10} \mod 2^5
\end{align*}
\]

Multiplication

Task: Computing Exact Product of $w$-bit numbers $x, y$ (either signed or unsigned)

Ranges of results:
- Unsigned multiplication requires up to $2w$ bits to store result:
  \[ 0 \leq x \times y \leq (2^w - 1)^2 = 2^{2w} - 2^w + 1 \]
- Two's complement smallest possible value requires up to $2w-1$ bits:
  \[ x \times y \geq (-2^{w-1}) \times (2^{w-1}-1) = -2^{2w-2} + 2^w \]
- Two's complement largest possible value requires up to $2w$ bits (in one case):
  \[ x \times y \leq (-2^{w-1})^2 = 2^{2w-2} \]

Maintaining exact results would need to keep expanding word size with each product computed.

Multiplication by power of 2 (left shift)

Multiplication by a power of two is equivalent to the left shift operation:

\[ u \times 2^k \text{ is the same as } u << k \]

For example:

\[
\begin{align*}
u << 3 & = u \times 8 \\
(u << 5) & = (u << 3) \times 8 = u \times 24 \\
(u + (u << 3)) << 2 & = u \times 12
\end{align*}
\]

- Most machines shift and add faster than multiply
- Compiler convert some multiplication to shift operations automatically.
Division by powers of 2 (right shift)

Unsigned integer division by a power of two is equivalent to right shift

\[ \text{floor} \left( \frac{u}{2^k} \right) \text{ is the same as } u \gg k \]

With signed integers, when \( u \) is negative the results are rounded incorrectly.

Memory Organization

Word size

Every computer has a "word size". Word size determines the number of bits used to store a memory address (a pointer in C):

- This determines the maximum size of virtual memory (virtual address space)
- Until recently, most machines used 32-bit (4-byte) words
  Limits total memory for a program to 4GB (too small for memory-intensive applications)

\[ 2^{32} B = \frac{2^{32} B}{GB} = 2^{32} GB = 4GB \]

- These days, most new systems use 64-bit (8-byte) words
  Potential address space = 1.8 X 10^{10} bytes
  x86-64 machines support 48-bit addresses: 256 Terabytes

\[ 2^{64} B = \frac{2^{64} B}{TB} = 2^{64} TB \]

Word oriented memory organization

- Address of a word in memory is the address of the first byte in that word.
- Consecutive word addresses differ by 4 (32-bit) or 8 (64-bit).

Byte ordering in a word

There are two different conventions of byte ordering in a word:

- **Big Endian**: Sun, PowerPC Mac, Internet
  Least significant byte has highest address
- **Little Endian**: x86, ARM processors running Android, iOS, and Windows
  Least significant byte has lowest address

Example:
variable \( x \) has 4-byte value of \( 0x01234567 \), address given by \( &x \) is \( 0x100 \)

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x0100 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01 23 45 67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67 45 23 01</td>
</tr>
</tbody>
</table>

Byte ordering example

Deciphering Numbers:

- Number in decimal: \( 321560 \)
- Number in hex: \( 0xe5618 \)
- Pad to 32-bits: \( 0x00000e5618 \)
- Split into bytes: 00 04 E8 18
- Big Endian byte order: 00 04 E8 18
- Little Endian byte order: 18 E8 04 00
  (reverse bytes, not the content of bytes!)

DNH:
For each of the following decimal numbers show how they would be stored as bytes using Big Endian and Little Endian conventions. Assume that the word size is 32 bits.
5789021, 10, 1587, 989795, 341, 2491
Examining Data Representation in C

```c
typedef unsigned char * pointer;

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

- Casting any pointer to `unsigned char *` allows is to treat the memory as a byte array.
- Using `printf` format specifiers:
  - `%p` - print pointer
  - `%x` - print value in hexadecimal

Running the following code

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

produces

```
int a = 15213;
0x7ffdf530b0ac 0x6d
0x7ffdf530b0ad 0x3b
0x7ffdf530b0ae 0x00
0x7ffdf530b0af 0x00
```

on Linux x86-64 PC

and

```
int a = 15213;
0xfffbf4e 0x00
0xfffbf4d 0x00
0xfffbf4c 0x00
```

on Sun Solaris machine (32-bit)