# 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)

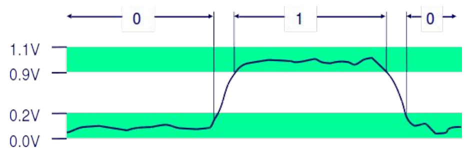
---

## Bits, bytes and bit-vectors



---

## 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 bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires



---

## Bytes

**1 byte = 8 bits**

Range of representable values:

- binary: $00000000_2$ to $11111111_2$
- decimal: $0_{10}$ to $255_{10}$
- hexadecimal: $00_{16}$ to $FF_{16}$

**Hexadecimal**: shorthand notation for binary (easier to write one symbol than four) used by humans

- 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 |

## 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.

| Binary | Decimal |
|---|---|
| 1 KB (1KiB) = 2^10 bytes = 1,024 bytes | 1 KB = 10^3 bytes = 1,000 bytes |
| 1MB (1MiB) = 2^20 bytes = 1,048,576 bytes | 1 MB = 10^6 bytes = 1,000,000 bytes |
| 1GB (1GiB) = 2^30 bytes = 1,073,741,824 bytes | 1 GB = 10^9 bytes = 1,000,000,000 bytes |
| 1TB (1TiB) = 2^40 bytes = 1,099,511,627,776 bytes | 1 TB = 10^12 bytes = 1,000,000,000,000 bytes |



## 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, 10101011010001, 100001

Decimal: 255, 64, 100, 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**

| & | 0 | 1 |
|---|---|---|
| **0** | 0 | 0 |
| **1** | 0 | 1 |

**OR**

| \| | 0 | 1 |
|---|---|---|
| **0** | 0 | 1 |
| **1** | 1 | 1 |

**NOT (complement)**

| ~ | |
|---|---|
| **0** | 1 |
| **1** | 0 |

**XOR**

| ^ | 0 | 1 |
|---|---|---|
| **0** | 0 | 1 |
| **1** | 1 | 0 |



## 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
0110 & 1010 = 0010
0110 | 1010 = 1110
0110 ^ 1010 = 1100
```

DNHI:

Pick a bit vector, say 101110. Xor it with an arbitrary other bit vector b, save the result in r. Xor r with b. What do you get?

These operators in C are called **bitwise operators**

**Warning:**

**do not confuse bitwise 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
**>>** (logical) x >> 2 is 00011000 (arith.) x >> 2 is 00011000

Example 2: x is 10100010
**<<** x << 3 is 00010000
**>>** (logical) x >> 2 is 00101000 (arith.) x >> 2 is 11101000

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



# Integer encoding



## Swapping values of variables without a temp

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

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

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



## Encoding of Integers

**Unsigned:**

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

**Two's Complement (Signed):**

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

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

| | Decimal | Hex | Binary |
|---|---|---|---|
| x | 15213 | 3B 6D | 00111011 01101101 |
| y | -15213 | C4 93 | 11000100 10010011 |

**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:**

$U_{min} = 0$

$U_{max} = 2^w - 1$

Assume w = 5:

- smallest unsigned:

  $00000_2 = 0_{10}$

- largest unsigned:

  $11111_2 = 31_{10}$

**Two's Complement:**

$T_{min} = -2^{w-1}$

$T_{max} = 2^{w-1} - 1$

Assume w = 5:

- smallest 2's comp:

  $10000_2 = -16_{10}$

- largest 2's comp:

  $01111_2 = 15_{10}$



## Umax, Tmin, Tmax for standard 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 |

Notice:

- the range of 2's complement values is not symmetric

  |Tmin| = |Tmax|+1

- for a given value of w

  Umax = 2 * Tmax + 1

In C:

- To access the values of largest/smallest values use `#include<limits.h>`
- The constants are named
  - `UINT_MAX`
  - `INT_MAX`
  - `INT_MIN`

  (these numbers are system specific)



## Comparison of Unsigned and Two's Comp.

| *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
- +/- 16 (in general 2^w) 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, 4294967259U

**Casting**

- Explicit casting between signed & unsigned
    - int tx, ty;
    - unsigned ux, uy;
    - tx = (int) ux;
    - uy = (unsigned) ty;
- Implicit casting also occurs via assignments and function calls
    - tx = ux;
    - uy = ty;



## 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?

```
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" );
```

```
-1 < 0 is true

-1 < 0u is false
```



## DNHI: 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:

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

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



# Expanding, truncating

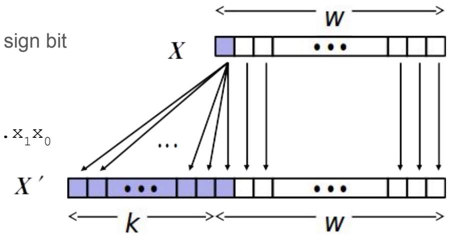
## Sign Extension

**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

$X = x_{w-1}\ x_{w-2}...x_1 x_0$

$X' = \underbrace{\mathbf{x_{w-1}}...\mathbf{x_{w-1}}}_{\leftarrow\ k\ times\ \rightarrow} x_{w-1}\ x_{w-2}...x_1 x_0$




```
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.

```
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 |



## 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:

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

prints

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



# Arithmetic Operations: Negation, Addition, Multiplication, (Multiplication using Shifting)

# 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**: $X = 011001_2 = 25_{10}$

$\sim X = 100110_2 = -26_{10}$

$\sim X+1 = 100111_2 = -25_{10}$

Notice that for any signed integer `X`, we have $\sim X + X = 111...11_2 = -1_{10}$



# 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:

UAdd(u , v) = (u + v) mod $2^w$

$$\begin{array}{rcl} 10010_2 & = & 18_{10} \\ +\ 11011_2 & = & 27_{10} \\ \hline 101101_2 & = & 45_{10} \\ \hline 01101_2 & = & 13_{10} \quad = 45_{10} \ \% \ 2^5 \end{array}$$

DNHI:
Show the results of adding the following unsigned bit vectors. Assume w = 5.
$11111_2 + 11111_2$
$00101_2 + 10010_2$
$10101_2 + 01111_2$



# 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 `TAddw(u,v)` $>= 2^{w-1}$, then sum becomes negative (**positive overflow**)
- If `TAddw(u,v)` $< -2^{w-1}$, then sum becomes positive (**negative overflow**)

$$\begin{array}{rcl} 10010_2 & = & -14_{10} \\ +\ 11011_2 & = & -5_{10} \\ \hline 101101_2 & = & -19_{10} \\ \hline 01101_2 & = & 13_{10} \end{array}$$

DNHI:
Show the results of adding the following signed bit vectors. Assume w = 5.
$11111_2 + 11111_2$
$00101_2 + 10010_2$
$10101_2 + 01111_2$



# 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:

  $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$

- Two's complement smallest possible value requires up to `2w-1` bits:

  $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$

- Two's complement largest possible value requires up to 2w bits (in one case):

  $x * y \le (-2^{w-1})^2 = 2^{2w-2}$

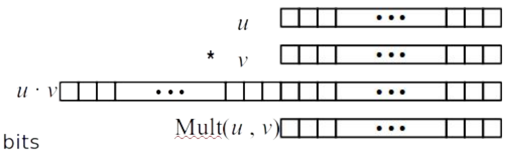
Maintaining exact results would need to keep expanding word size with each product computed.

## Multiplication signed/unsigned

Operands: *w* bits

True Product: 2*$w$ bits

Discard *w* bits: keep *w* bits




Multiplication results for signed and unsigned bit vectors ignore the high order bits.

DNHI:
- Show the results of multiplying the following signed bit vectors. Assume w = 5.
  $11111_2$ * $11111_2$ $00101_2$ * $10010_2$ $10101_2$ * $01111_2$
- If you multiply two very large numbers (large enough that the product cannot be stored in w bits), can you predict if the result is positive or negative?



## Multiplication by power of 2 (left shift)

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

`u * 2^k` is the same as `u << k`

For example:

```
u << 3  ==  u * 8

(u << 5) - (u << 3)  ==  u * 24

(u + (u << 1)) << 2  ==  u * 12
```

- 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

`floor ( u / 2^k )` is the same as `u >> 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)

$$2^{32}B = \frac{2^{32}B}{2^{30}B/GB} = 2^2GB = 4GB$$

- These days, most systems use 64-bit (8-byte) words
  (Potential address space ≈ 1.8 X $10^{19}$ bytes)
  x86-64 machines support 48-bit addresses: 256 Terabytes

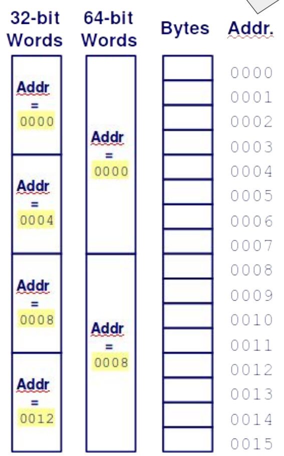
$$2^{64}B = \frac{2^{64}B}{2^{40}B/TB} = 2^{14}TB$$



## Word oriented memory organization

Note: memory addresses are not usually expressed in decimal.

- 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`

**Big Endian**

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

**Little Endian**

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



## Byte ordering example

How are numbers stored in memory?

- Number in decimal: `321560`
- Number in hex: `0x4E818`
- Pad to 32-bits: `0x0004E818`
- 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!)

DNHI:
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

```
typedef unsigned char * pointer;

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

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



## Examining Data Representation in C

Running the following code

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

produces

```
int a = 15213;
0x7ffd1530b0ac    0x6d
0x7ffd1530b0ad    0x3b
0x7ffd1530b0ae    0x00
0x7ffd1530b0af    0x00
```

on Linux x86-64 PC

and

```
int a = 15213;
ffbffb4c    0x00
ffbffb4d    0x00
ffbffb4e    0x3b
ffbffb4f    0x6d
```

on Sun Solaris machine (32-bit)