# Bits, Bytes, and Integers

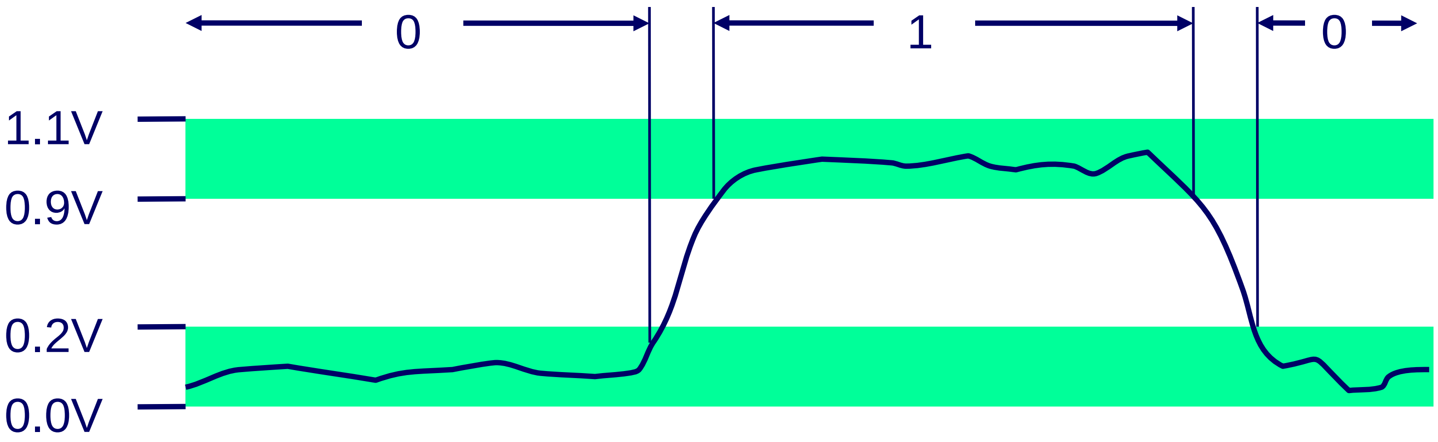
## CS 485G-006: Systems Programming

Lectures 2 and 3: 15–18 Jan 2016

# Overview: Bits, Bytes, and Integers

- **Representing information as bits**
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed

# Everything is bits



# For example, can count in binary

- **Base 2 Number Representation**
  - Represent $15213_{10}$ as $11101101101101_2$
    - 1 x 8192 + 1 x 4096 + 1 x 2048 + 0 x 1024 + ….
  - Represent $1.20_{10}$ as $1.0011001100110011[0011]…_2$
  - Represent $1.5213 \text{ X } 10^4$ as $1.1101101101101_2 \text{ X } 2^{13}$

# Bits, bytes, and octets

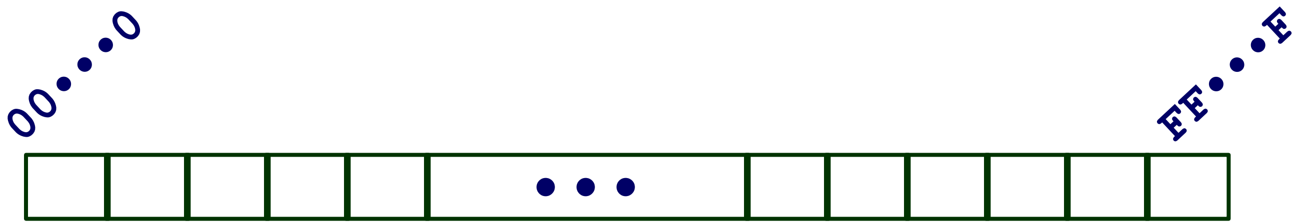
- **Bases**
  - *Decimal* – base 10: our number system
  - *Binary* – base 2: all 0s and 1s
  - *Hexadecimal* – base 16
    - 0-9, A-F
    - $FA1D37B_{16}$ = `0xfa1d37b` in C
    - 4 bits per “digit”
- **Byte = 8 bits (usually)**
  - Machine term: smallest addressable unit of memory
  - Binary $00000000_2$ to $11111111_2$
  - Decimal: $0_{10}$ to $255_{10}$
  - Hexadecimal $00_{16}$ to $FF_{16}$
- **Octet = 8 bits (always)**
  - This is the term used in networking
- **How many bytes to store:** $1001100011101_2 = 131D_{16} = 4893_{10}$

| 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 |

# How big is "mega"

- **Scientists, most engineers say $10^6$**
  - 1,000,000 – decimal megabyte
  - Megabyte – MB (what is Mb?)
- **Programmers, computer engineers, say $2^{20}$**
  - 1,048,576 – binary megabyte
  - About 5% larger
  - "Mebibyte" (MiB) – IEC "standard" term, but *not very common*
- **Networks are based on clock rates in Hz: decimal mega**
  - One megabit per second = $10^6$ bits per second
- **Computer memory is based on powers of 2: binary mega**
  - A megabyte of memory is $2^{20}$ bytes.
- **Hard drives and SSDs?**
  - For marketing reasons, use the decimal system (sounds bigger)
  - Even though sector sizes, Flash blocks are usually powers of two.
- **Kilo, Giga, Tera are similar**

# Byte-Oriented Memory Organization



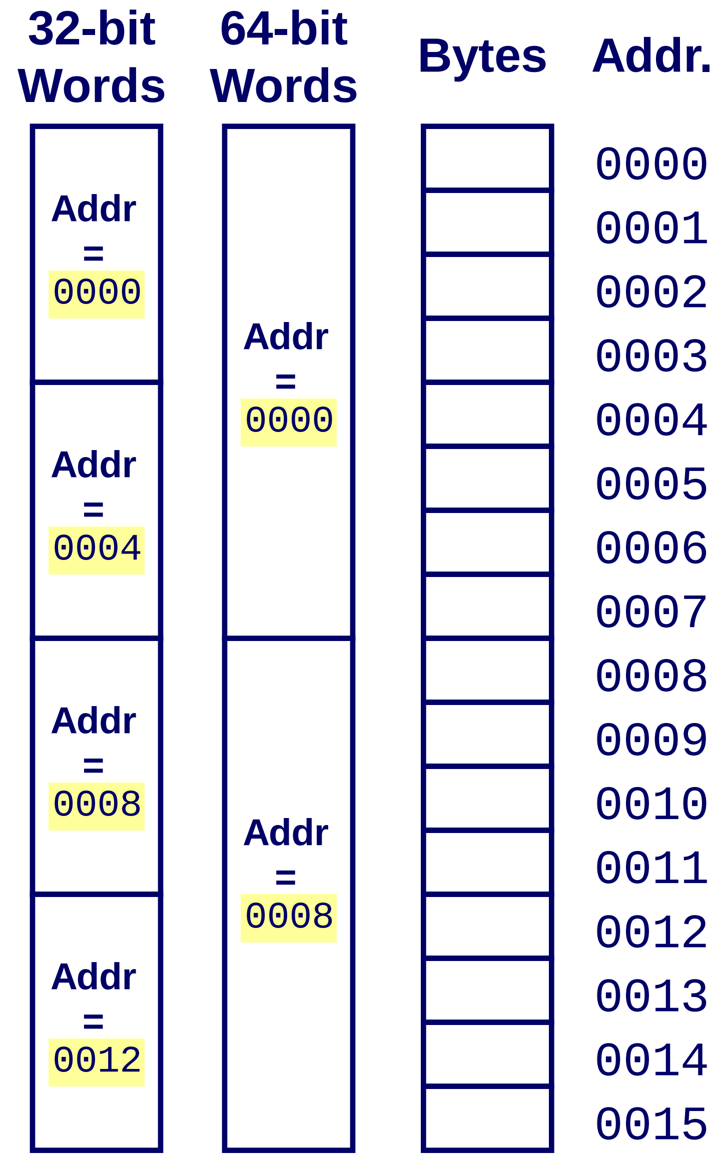

- **Programs refer to data by (virtual) address**
  - Conceptually, envision memory as a very large array of bytes
    - In reality, it's not, but virtual memory makes it look that way
  - An address is like an index into that array
    - and, a pointer variable stores an address
- **System provides private address spaces to each "process"**
  - So, a program can clobber its own data, but not that of others
- **Compiler + OS control the allocation of memory**
  - OS determines where different programs should be stored
  - Compiler determines how data is laid out within a program

# Machine Words

- **Any given computer has a "Word Size"**
  - Nominal size of integer-valued data
    - And/or of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB ($2^{32}$ bytes)
    - Too small for memory-intensive applications
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That's 18.4 X $10^{18}$
    - x86-64 currently supports 48 bits of address: 256 TB
  - 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)
- **Beware terminology**
  - Intel terminology is old-fashioned
  - Doubleword = 4 bytes (32-bit)
  - Quadword = 8 bytes (64-bit)
  - Why? Backwards compatibility.
    - “Word” = 2 bytes (16-bit)
    - Chips still support 16-bit code
    - Even 16+32 bit code, or 32+64 bit, at the same time!



# Example Data Representations

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

# Byte Ordering

- **So, how are the bytes within a multi-byte word ordered in memory?**
- **Conventions**
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: x86, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address

# Byte Ordering Example

- **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 | | |

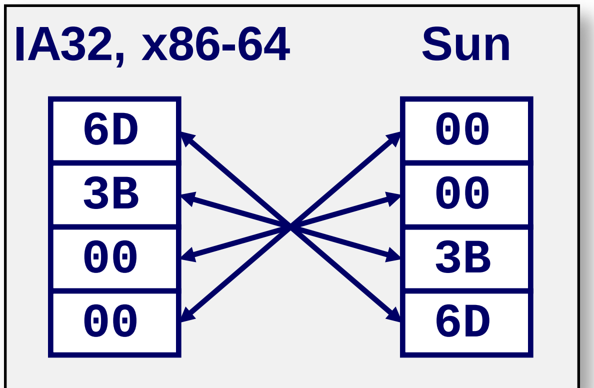
**Little Endian**

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

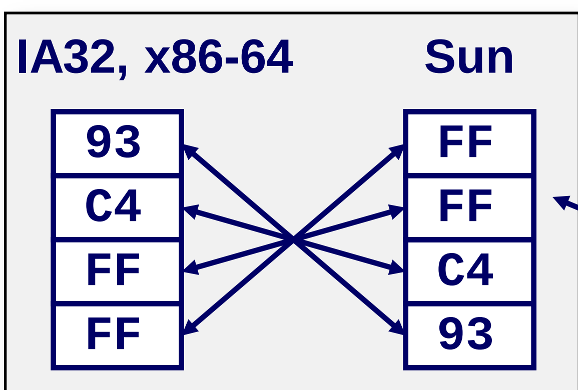
# Representing Integers

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

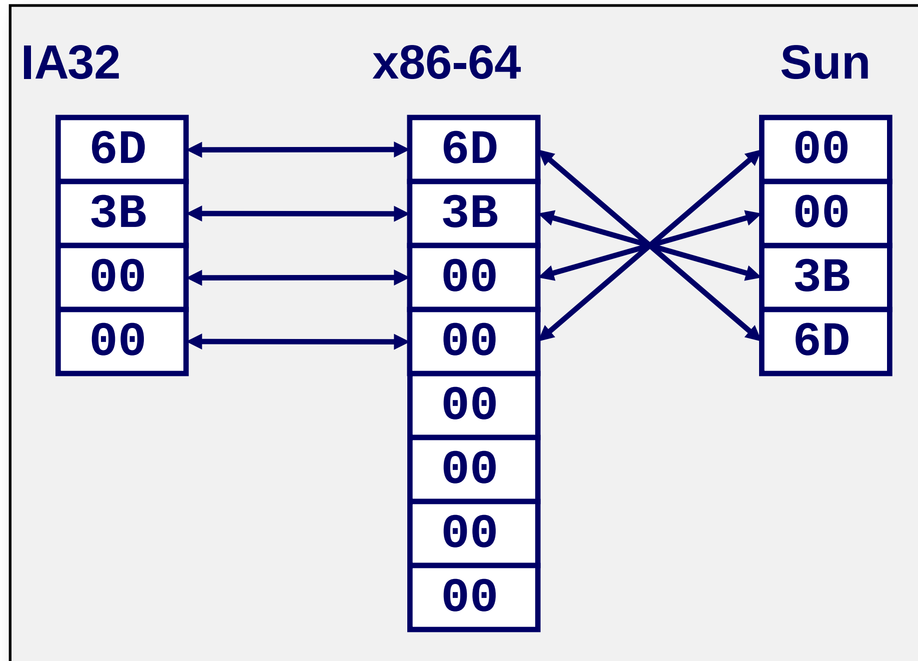
`int A = 15213;`




`int B = -15213;`




`long int C = 15213;`




**Two's complement representation**

# 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, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf(”%p\t%.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 x86-64):**

```
int a = 15213;
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

# Reading Byte-Reversed Listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- **Example Fragment**

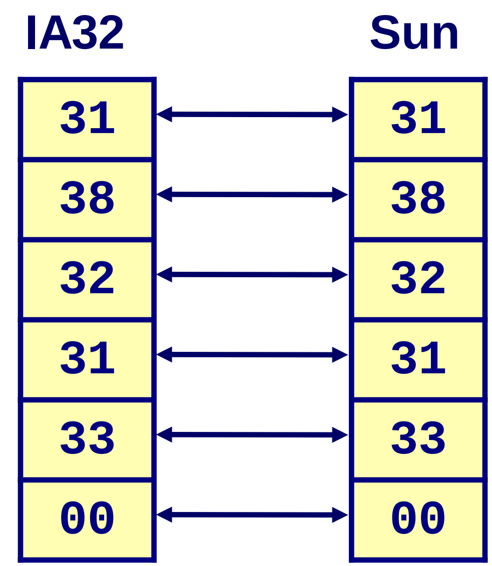
| Address | Instruction Code | Assembly Rendition | |
|---|---|---|---|
| `8048365:` | `5b` | `pop` | `%ebx` |
| `8048366:` | `81 c3 ab 12 00 00` | `add` | `$0x12ab,%ebx` |
| `804836c:` | `83 bb 28 00 00 00 00` | `cmpl` | `$0x0,0x28(%ebx)` |

- **Deciphering Numbers**
  - Value: 0x12ab
  - Pad to 32 bits: 0x000012ab
  - Split into bytes: 00 00 12 ab
  - Reverse: ab 12 00 00

# Representing Strings

```
char S[6] = "18213";
```

- **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



# Strings vs. Buffers

- **Strings and buffers can be easy to confuse**
- **They look alike in C:**
  - String: `char example_string[200];`
  - Buffer: `char example_buffer[200];`
- **The difference? How you use them.**
- **String**
  - Sequence of characters, usually “printable”.
  - Uses a NUL character ‘\0’ (all zero bits) to mark the end.
  - Meaning NUL bytes are not allowed within the string.
- **Buffer**
  - Not explicitly defined by C, but often used in networking, OS, ...
  - An array of bytes.
  - Stores any byte, including 0 – so NUL cannot be a terminator.
  - Requires an additional variable to store the “current size” (how many of the bytes contain meaningful data).

# Overview: Bits, Bytes, and Integers

- Representing information as bits
- **Bit-level manipulations**
- Integers
  - Representation: unsigned and signed

# 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 |

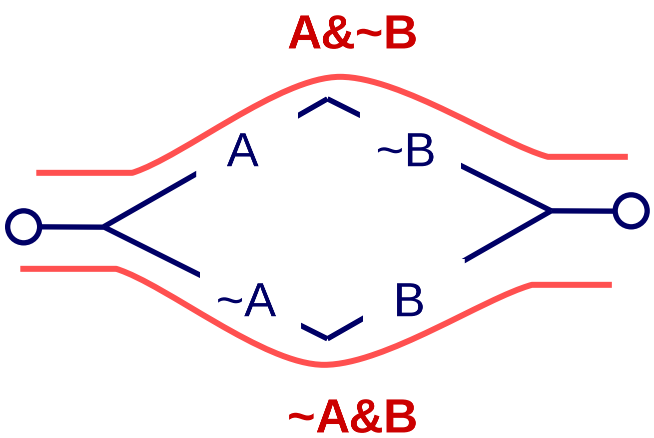
## Exclusive-Or (Xor)

- **A^B = 1 when either A=1 or B=1, but not both**

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

# Application of Boolean Algebra

- **Applied to Digital Systems by Claude Shannon**
  - 1937 MIT Master's Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0




**Connection when**

**A&~B | ~A&B**

**= A^B**

# 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**

# Example: Representing & Manipulating Sets

- **Representation**
  - Width w bit vector represents 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 &, |, ~, ^ 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 data type)**
  - ~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
    - $01101001_2$ | $01010101_2 \rightarrow 01111101_2$

# Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - &&, ||, !
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination ("short-circuiting")
- **Examples (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`**
  - 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 (unsigned)
    - Fill with 0's on left
  - Arithmetic shift (signed)
    - Copy most significant bit on left
- **Undefined Behavior**
  - If the shift amount is < 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 |

# Overview: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - **Representation: unsigned and signed**

# Encoding Integers

**Unsigned**

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

**Two's Complement**

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

**Sign Bit**

```
short int x =  15213;
short int y = -15213;
```

- **C `short` 2 bytes long**

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

- **Sign Bit**
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

# Two's-complement Encoding (Cont.)

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

| Weight | 15213 | | -15213 | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 1 | 2 |
| 4 | 1 | 4 | 0 | 0 |
| 8 | 1 | 8 | 0 | 0 |
| 16 | 0 | 0 | 1 | 16 |
| 32 | 1 | 32 | 0 | 0 |
| 64 | 1 | 64 | 0 | 0 |
| 128 | 0 | 0 | 1 | 128 |
| 256 | 1 | 256 | 0 | 0 |
| 512 | 1 | 512 | 0 | 0 |
| 1024 | 0 | 0 | 1 | 1024 |
| 2048 | 1 | 2048 | 0 | 0 |
| 4096 | 1 | 4096 | 0 | 0 |
| 8192 | 1 | 8192 | 0 | 0 |
| 16384 | 0 | 0 | 1 | 16384 |
| -32768 | 0 | 0 | 1 | -32768 |
| **Sum** | | **15213** | | **-15213** |

# 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 |

# 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

# 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
- **$\Rightarrow$ Can Invert Mappings**
  - $\mathrm{U2B}(x) = \mathrm{B2U}^{-1}(x)$
    - Bit pattern for unsigned integer
  - $\mathrm{T2B}(x) = \mathrm{B2T}^{-1}(x)$
    - Bit pattern for two's comp integer

# Signed vs. Unsigned in C

- **Literals**
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix: **`0U, 4294967259U`**
- **Types:**
  - Signed: int, short, long, long long, signed char
  - Unsigned: unsigned, unsigned short, unsigned long, …, size_t
  - Plain char can be either signed or unsigned on different platforms!
- **Casting**
  - Explicit casting between signed & unsigned keep the bit patterns and reinterpret (U2T(x) = B2T(U2B(x)), T2U(x) = B2U(T2B(x))

```
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;
```

# 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** |

# Why Should I Use Unsigned?

- ***Don't*** **use without understanding implications**
  - 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)
  . . .
```

# Why Should I Use Unsigned? (cont.)

- ***Do*** **Use When Performing Modular Arithmetic**
  - Multiprecision arithmetic
- ***Do*** **Use When Using Bits to Represent Sets**
  - Logical right shift, no sign extension