Bits, Bytes, and Integers

Adapted from slides of the textbook: [http://csapp.cs.cmu.edu/](http://csapp.cs.cmu.edu/)

CENG331 - Computer Organization

Carnegie Mellon
Hello World!

What happens under the hood?
```c
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}

#include <stdio.h>

hello.c
```
Compilation of hello.c

Preprocessor (cpp)

Source program

Modifled

Object program (text)

Compiler (cc1)

Assembly program

Assembler (as)

Relocatable object programs (binary)

Linker (ld)

Executable object program (binary)

printf.o

Source program (text)

Modified source program (text)

Assembly program

Assembler (as)

Relocatable object programs (binary)

Linker (ld)

Executable object program (binary)

hello.o

Compiler (cc1)

Object program (text)

Modifled

Source program

Preprocessor (cpp)

hello.c

hello.c

hello.i

Compiler (cc1)

hello.s

Assembler (as)

hello.o

Linker (ld)

hello

Source program

Modified source program (text)

Assembly program

Assembler (as)

Relocatable object programs (binary)

Linker (ld)

Executable object program (binary)

printf.o

Compilation of hello.c
C

```c
#include <stdio.h>

int main() {
    printf("Hello World");
}
```

Preprocessor

Pre-processor (cpp)

Hello.c

Hello.i

```c
#include <stdio.h>

typedef unsigned char __u_char;

typedef unsigned short __u_short;

typedef unsigned __u_int;

typedef unsigned long __u_long;
```

Hello.c

Hello.i
Compiler

# hello.c

# <built-in>

# <command line>

```
hello.c
/usr/include/stdio.h
```

```
typedef unsigned char __u_char;
typedef unsigned short __u_short;
typedef unsigned int __u_int;
typedef unsigned long __u_long;
```

```
int main() {
    printf("Hello World");
}
```

```
gcc -Wall -S hello.s
```

```
Hello World
```

```
main: RET
```

```
Function: main
```

```
Text
```

```
include "hello.c"
```

```
cc1
```

```
file "hello.c"
```

```
section .rodata
.LC0:
.string "Hello World"
```

```
text .globl
main
```

```
.type main, @function
```

```
main:
```

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
andl $-16, %esp
movl $0, %eax
addl $15, %eax
addl $15, %eax
shrl $4, %eax
sall $4, %eax
subl %eax, %esp
subl $12, %esp
pushl $.LC0
call printf
addl $16, %esp
leave
ret
```

```
.size main, .-main
```

```
section .note.GNU-stack,"",@progbits
.ident "GCC: (GNU) 3.4.1"
```

```
ac -Wall -S hello.s > hello.s
```

```
{ printf("Hello World\n");
  } main()
```

```
... ...
```

```
typedef unsigned long int __u_long;
typedef unsigned short int __u_short;
typedef unsigned char __u_char;
```

```
...
```

```
# /usr/include/stdio.h
```

```
# "hello.c"
```

```
"command line"
```

```
"\n"
```

```
"\n"
```

```
"\n"
```

```
"\n"
```

```
"\n"
```
A hello.o

@function

World"...
od – a hello

hello

gcc hello.o –o hello

Hello, world.

printf.c

Program

Object

Executable

Linker

Program

Object

Hello
Finally...

$ gcc hello.o -o hello
$ ./hello
Hello World$

$
How do you say "Hello World"?
Typical Organization of System

- **CPU**
- **Main memory**
- **I/O bridge**
- **Bus Interface**
- **ALU**
- **Register file**
- **Display**
- **Keyboard**
- **Mouse**
- **USB controller**
- **Graphics controller**
- **Disk controller**
- **Expansion slots for other devices such as network adapters**

**System bus**

**Memory bus**

**I/O bus**

**Main memory**

**Disk**

**Hello executable stored on disk**

**Typical Organization of System**

- **Main memory**
- **I/O bridge**
- **Bus interface**
- **ALU**
- **Register file**
- **CPU**
- **Disk**
- **Graphics controller**
- **USB controller**
- **Disk controller**
- **Expansion slots for other devices such as network adapters**

**System bus**

**Memory bus**

**I/O bus**

**Main memory**

**Disk**

**Hello executable stored on disk**
User types "hello"

Reading hello command from keyboard

Main memory

Disk controller

Disk

Expansion slots for other devices such as network adapters

I/O bus

CPU

Bus interface

Register file

ALU

System bus

Memory bus

I/O bridge

Reading hello command from keyboard

Display

Mouse/keyboard controller

Graphics adapter

Graphics

USB controller

Mouse

Keyboard

Display

Disk controller

Disk

Expansion slots for other devices such as network adapters
Today: Bits, Bytes, and Integers

Representations in memory, pointers, strings

Summary

- Addition, negation, multiplication, shifting
- Expanding, truncating
- Conversion, casting
- Representation: unsigned and signed

Integers

Bit-level manipulations

Representing Information as bits
For example, can count in binary.

- Base 2 Number Representation
  - Represent $1.5213 \times 10^4$ as $1.1101101101101101_{2}$
  - Represent $1.20_{10}$ as $1.00110011001100110011_{2}$...
  - Represent $1.5213_{10}$ as $1.1101101101101_{2}$
**Encoding Byte Values**

- **Byte** = 8 bits

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary 8-bit</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>00000001</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>00000010</td>
<td>02</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>255</td>
<td>11111111</td>
<td>FF</td>
</tr>
</tbody>
</table>

- **Binary 00000000** to **11111111**
- **Decimal**: 0 to 255
- **Hexadecimal**: 00 to FF
- **Byte**: 8 bits

Example:
- **Write FA1D37B in C as:**
  - `0xFA1D37B`
  - `0xfa1d37b`

Convert in 4-bit groups:
- 1111 1010 0000 0011
- **1111 1010 0000 0011** 16

**Hexadecimal 0016 to FF16**

- **Decimal**: 00 to 255
- **Binary**: 00000000 to 111111112
### Example Data Representations

<table>
<thead>
<tr>
<th>C Data Type</th>
<th>x86-64-bit</th>
<th>Typical 32-bit</th>
<th>Typical 64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>-10/16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>pointer</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

**Intel**

- 16/32-bit
- 64-bit

**x86-64**

- 64-bit
Today: Bits, Bytes, and Integers

Representations in memory, pointers, strings

Summary
- Addition, negation, multiplication, shifting
- Expanding, truncating
- Conversion, casting
- Representation: unsigned and signed

Integers

Bit-level manipulations

Representing information as bits
Boolean Algebra

Developed by George Boole in 19th Century

Algebraic Representation of Logic

- Boolean Algebra
- Algebraic representation of logic
- Encode "True" as 1 and "False" as 0

And

Or

Exclusive-Or (Xor)

Not

AND

OR


A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not

A = 1 when A=0

A&B = 1 when both A=1 and B=1

A|B = 1 when either A=1 or B=1

~A = 1 when A=0

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

And

Or

Exclusive-Or (Xor)

Not
General Boolean Algebras

- Operate on Bit Vectors
- Operations applied bitwise

All of the properties of Boolean Algebras Apply

| 01101001 | 01010101 | \( \lor \) | 01111101 |
| 01101001 | 01010101 | \( \land \) | 00111100 |
| \sim 01010101 | 10101010 |

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
Example: Representing & Manipulating Sets

Representation

- Bit vector represents subsets of \( \{0, \ldots, w-1\} \)
- \( a_j = 1 \) if \( j \in A \)

Operations

- Intersection
- Union
- Symmetric difference
- Complement

Representation

- \( \{0, 3, 5, 6\} \)
- \( \{0, 2, 4, 6\} \)
- \( \{0, 2, 3, 4, 5, 6\} \)
- \( \{0, 6\} \)

Operations

- \( 01000001 \) \( \{0, 6\} \)
- \( 01111101 \) \( \{0, 2, 3, 4, 5, 6\} \)
- \( 00111100 \) \( \{2, 3, 4, 5\} \)
- \( 10101010 \) \( \{1, 3, 5, 7\} \)

Width of bit vector represents subsets of \( \{0, \ldots, w-1\} \)
Bit-level Operations in C

Available in C

Operations & |, ~ A available in C

Examples (Char data type)
- Arguments applied bit-wise
- View arguments as bit vectors
- Long, int, short, char, unsigned

Apply to any "integral" data type

&:
- 0x69 & 0x55 = 0x41
  \[01101001 \text{ } \& \text{ } 01010101 = 01000001 \text{ (2)}\]

|:
- 0x69 | 0x55 = 0x7D
  \[01101001 \text{ } | \text{ } 01010101 = 01111101 \text{ (2)}\]

~:
- ~0x00 = 0xFF
  \[\sim 00000000 \text{ (2)} = 11111111 \text{ (2)}\]
- ~0x41 = 0xBE
  \[\sim 01101001 \text{ (2)} = 10111110 \text{ (2)}\]
**Contrast: Logic Operations in C**

- **View 0 as “False”**
- **Anything nonzero as “True”**
- **Always return 0 or 1**
- **Early termination**
- **Avoids null pointer access due to lazy evaluation**

---

Examples (char data type):

- `!0x41 = 0x00`
- `!0x00 = 0x01`
- `!!0x41 = 0x01`
- `0x69 && 0x55 = 0x01`
- `0x69 || 0x55 = 0x01`
- `p && *p` (avoids null pointer access due to lazy evaluation)
Contrast: Logical Operators in C

- Examples (char data type)
  - !0x41 = 0x00
  - !0x00 = 0x01
  - !!0x41 = 0x01
  - 0x69 && 0x55 = 0x01
  - 0x69 || 0x55 = 0x01
  - p && *(avoids null pointer access)

- Early termination
- Always return 0 or 1
- Anything nonzero as "True"
- View 0 as "False"
- &&, ||, !

Watch out for && vs. & (and || vs. |)… one of the more common oopsies in C programming.
### Shift Operations

**Left Shift:**
- \( x \ll y \)
- Shift bit vector \( x \) \( y \) positions
- Throw away extra bits on left
- Fill with 0's on right

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

### Undefined Behavior
- Shift amount < 0 or ≥ word size
- Replicate most significant bit on right

### C, however, has only one right shift operator, `>>`. Many C compilers choose which right shift to perform depending on what type of integer is being shifted; often signed integers are shifted using the arithmetic shift, and unsigned integers are shifted using the logical shift.

---

<table>
<thead>
<tr>
<th>Argument</th>
<th>Logical Shift</th>
<th>Arithmetic Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010000</td>
<td>00011000</td>
<td>00011000</td>
</tr>
<tr>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Argument</th>
<th>Left Shift</th>
<th>Right Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010000</td>
<td>00010000</td>
<td>00011000</td>
</tr>
<tr>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>00001000</td>
<td>00001000</td>
<td>00001000</td>
</tr>
</tbody>
</table>
Challenge: Can you code swap function without using a temporary variable?

Hint: Use bitwise XOR (^)

```c
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```
Today: Bits, Bytes, and Integers

Representations in memory, pointers, strings

Summary

Representations: unsigned and signed

Integers

Bit-level manipulations

Conversion, casting, expanding, truncating, negation, multiplication, addition

Summary
Encoding Integers

**Signed 16-Bit Two’s Complement**

\[ B2T(x) = \sum_{i=0}^{15} x_i \cdot 2^i \]

\[ B2T(x) = \sum_{i=0}^{15} x_i \cdot 2^i = -x \cdot 2^{-1} \cdot 2^{15} + \sum_{i=0}^{15} x_i \cdot 2^i \]

**Unsigned 16-Bit Two’s Complement**

\[ B2U(x) = \sum_{i=0}^{15} x_i \cdot 2^i = x \cdot 2^{-1} \cdot 2^{15} + \sum_{i=0}^{15} x_i \cdot 2^i \]

- Short Int Y = -15213
- Short Int X = 15213

Unsigned Integers

```
00000000 0101 0111 1111 0000
```

Signed Integers

```
00000000 0100 1011 1111 1111
```

Carnegie Mellon
Encoding Integers

\[ \text{short} ]
\[ \text{int} ]
\[ x = 15213; \]
\[ y = -15213; \]

- 1 for negative
- 0 for non-negative

For 2's complement, most significant bit indicates sign

$B_2^T(X) = -x$ for signed

$B_2U(X) = x$ for unsigned

\[ \sum_{i=0}^{w-1} 2^i x_i \]

Decimal  | Hex  | Binary  
--------|------|---------
15213   | C4 93 | 10000100 10010011 |
15213   | 15213 00111011 01101101 |

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
Two-complement Encoding Example (cont.)

<table>
<thead>
<tr>
<th>Weight</th>
<th>Sum</th>
<th>-152213</th>
<th>152213</th>
</tr>
</thead>
<tbody>
<tr>
<td>16384</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>128</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

For small $y$, $x = y$: 

-152213: 11000100 10010011
152213: 00111011 01101101

-152213: 00111011 01101101
152213: 11000100 10010011
Numeric Ranges

Unsigned Values

\[ U_{\text{Min}} = 0 \]

\[ U_{\text{Max}} = 2^w - 1 \]

\[ w = 16 \]

\[ U_{\text{Max}} = 2^w - 1 \]

\[ 0 = 0.000...0 \]

\[ U_{\text{Min}} = 0 \]

\[ w = 16 \]

\[ 111...1 \]

Unsigned Values

Integer Binary

Think Binary

Binary to Integer

Carnegie Mellon
Numeric Ranges

Two's Complement Values

\[ \text{TMin} = -2^{w-1} \]

\[ \text{TMax} = 2^w - 1 \]

Other Values

Minus 1

\[ \text{Minus 1} = \underbrace{111...1}_{w-1} - 1 \]

Max

\[ \text{Max} = \underbrace{100...0}_{w-1} \]

Min

\[ \text{Min} = \underbrace{10...0}_{w-1} - 1 \]

Thin

\[ x = \frac{1 - \left( 2^{m-1} + \frac{2^{m-1}}{2^{m-2}} \right)^{\frac{1}{m-1}}}{2 - 1} \]

\[ x \approx \frac{1 - \frac{2}{2^{m-1}}}{2 - 1} = \frac{1 - \frac{1}{2^{m-1}}}{1} = 1 - \frac{1}{2^{m-1}} \]

\[ \text{Max} = \underbrace{111...1}_{w-1} \]

\[ \text{Min} = \underbrace{00...0}_{w-1} \]