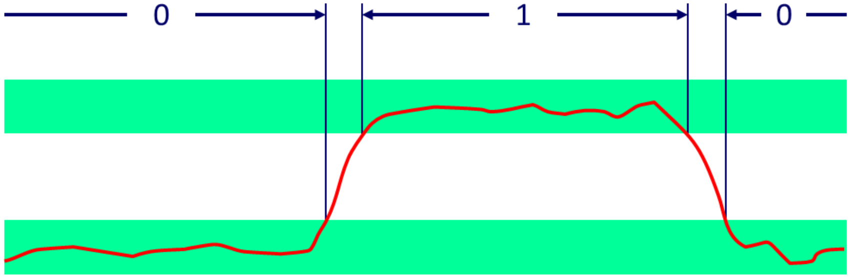
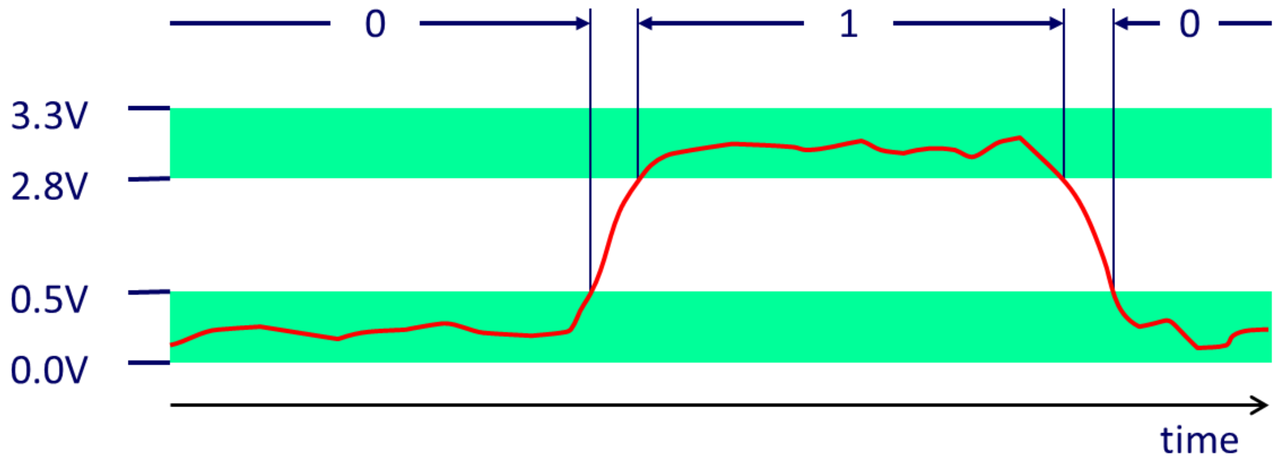
# Data Representations



# Module Outline

- Representing Information
- Data Types and Representations
- From Source Code to Machine Language
- Module Summary

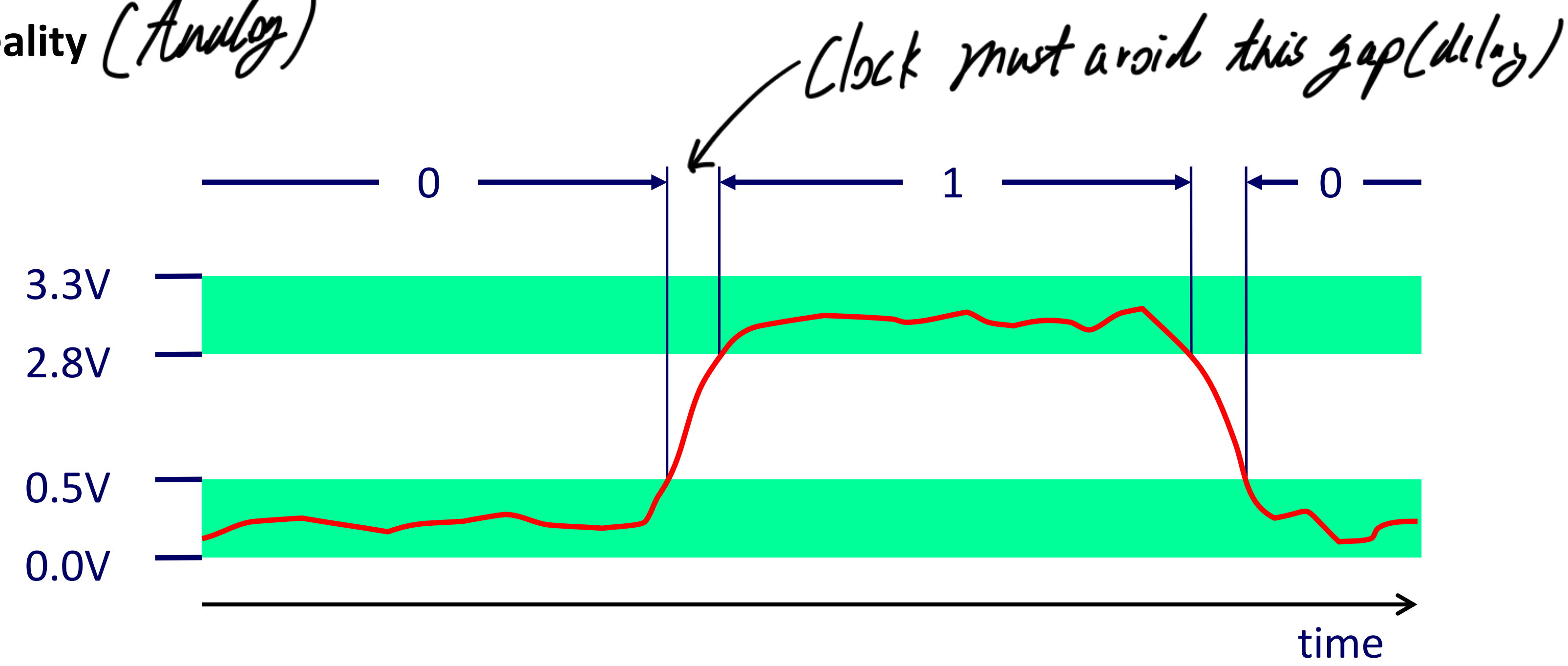



# Representing Information

# Binary Representations

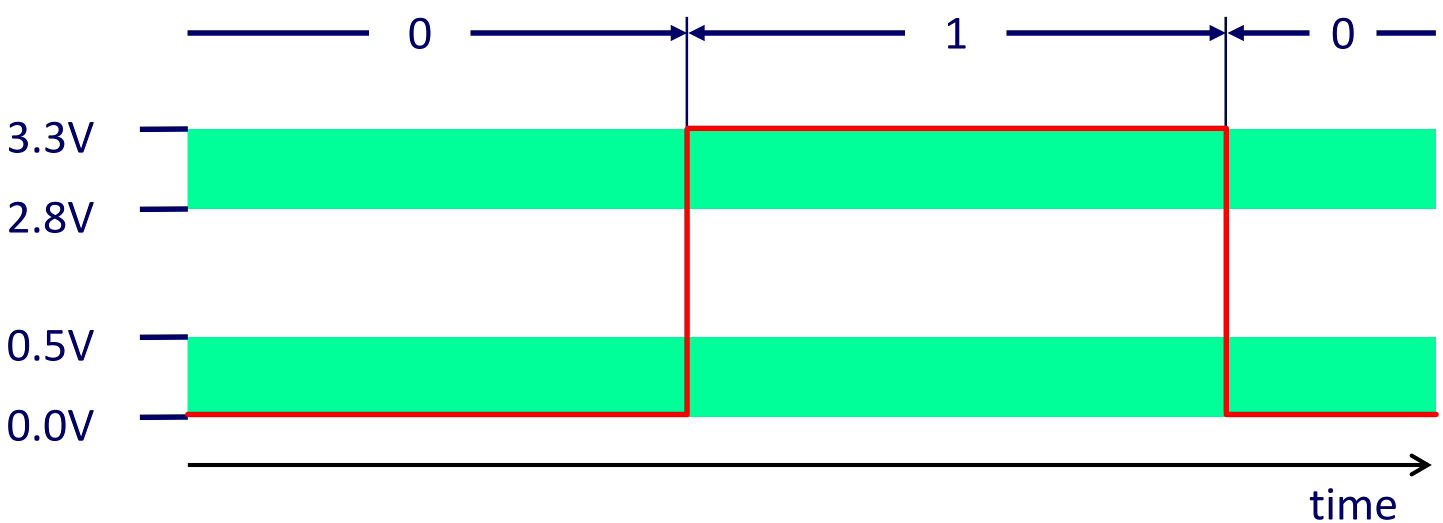
- **Binary representation in computer systems as voltage**

- **In reality** (Analog)



# Binary Representations

- **Binary representation in computer systems as voltage**
- **Abstraction** (digital)



# Representing Information : 0/1

- **Information = Bits + Context**
  - Computers manipulate representations of “things”
  - These “things” are represented at binary digits

- **What can we represent with N bits?**
  - $2^N$ things
  - Numbers, characters, source code, machine instructions, …

**64 bits:** `0100 0011 0110 1111 0110 1101 0111 0000 0100 0001 0111 0010 0110 0011 0110 1000`

$2^4$:

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

# Representing Information

- **Information = Bits + Context**
  - Computers manipulate representations of “things”
  - These “things” are represented at binary digits
- **What can we represent with N bits?**
  - $2^N$ things
  - Numbers, characters, source code, machine instructions, …

| 64 bits: | 0100 0011 | 0110 1111 | 0110 1101 | 0111 0000 | 0100 0001 | 0111 0010 | 0110 0011 | 0110 1000 |
|---|---|---|---|---|---|---|---|---|
| ASCII | ‘C’ | ‘o’ | ‘m’ | ‘p’ | ‘A’ | ‘r’ | ‘c’ | ‘h’ |
| char | 43 | 6f | 6d | 70 | 41 | 72 | 63 | 68 |
| int | 1886220099 | | | | 1751347777 | | | |
| long | 7521981428023521091 | | | | | | | |
| double | 7.09781e+194 | | | | | | | |

→ **The context (data type) gives the bits concrete meaning**

# Data Encodings

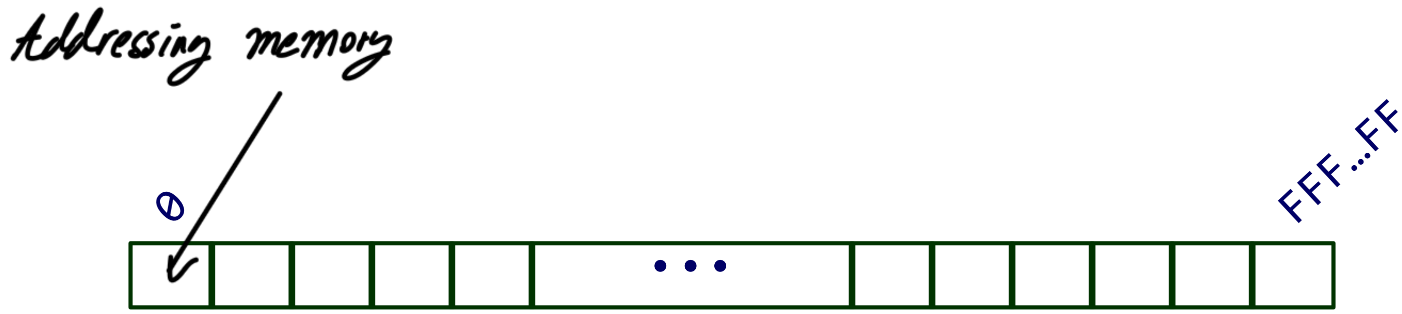
- **1 byte = 8 bits** 0 -1
  - Binary: $00000000_2$ to $11111111_2$
    - C: no direct representation → hex for instance
  - Octal: $0_8$ to $377_8$
    - C: octal number start with a '0'
    - reason
  - Decimal: $0_{10}$ to $255_{10}$
    - C: first digit must not be 0 for values != 0
  - Hexadecimal $00_{16}$ to $FF_{16}$
    - Use characters '0' to '9' and 'A'/'a' (10) to 'F'/'f' (15)
    - C: hexadecimal number start with '0x'

1 hex char = 4bits (half byte) => 0x41 = 65 (dec)

1 byte

| Hex (base = 16) | Decimal (base = 10) | Binary (base = 2) |
|---|---|---|
| 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 |

# Byte-Oriented Memory Organization




- **Memory conceptually very large array of bytes addressed from 0 to n**
  - mem[0] …. mem[*M*-1]

    Index

- **Programs refer to (virtual) addresses** ≠ physical address in NAND Memory, Flash Memory Etc.
  - System provides private address space to each running program
    - program can clobber its own data, but not that of others

      Block: Allocated By Compiler, OS, etc...

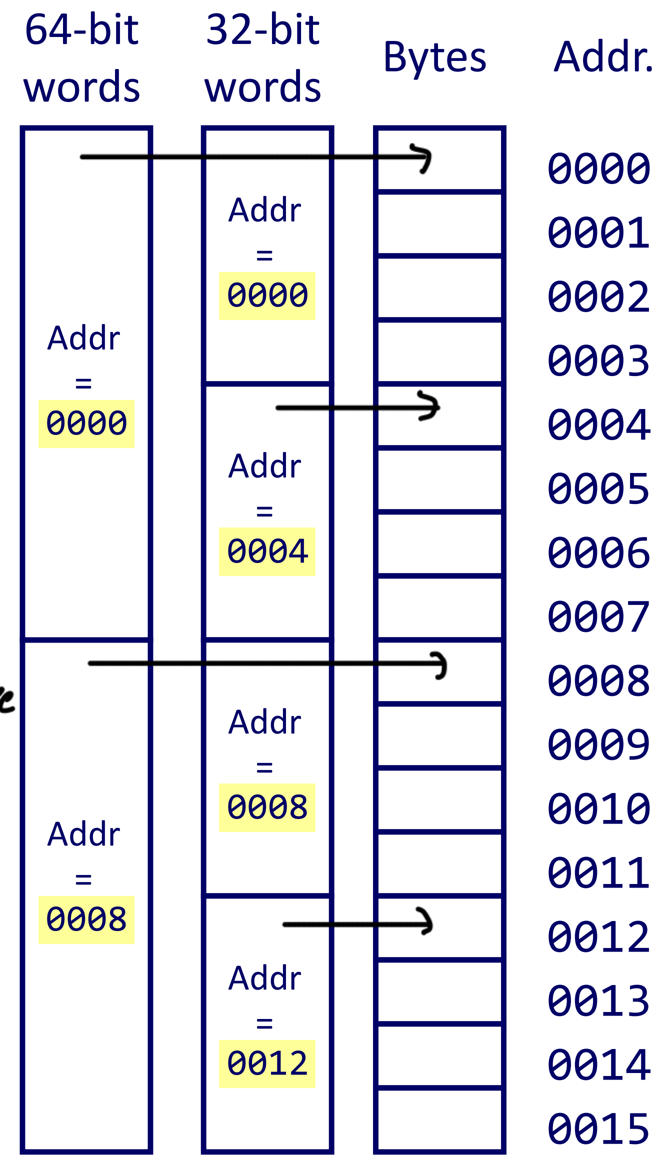
- **Where are the different data stored?**
  - Compiler + run-time system control allocation

# Word-Oriented Memory Organization

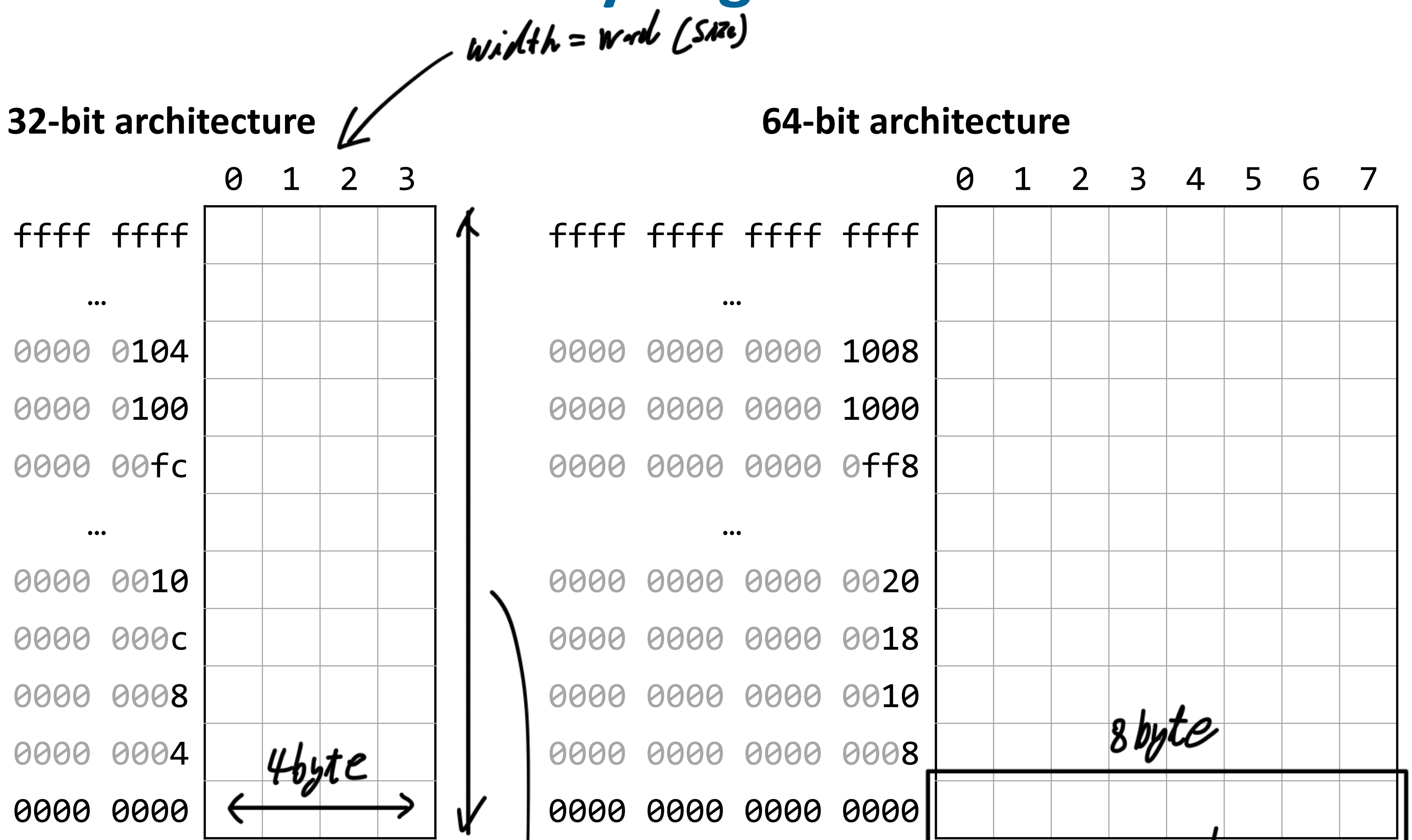
- **Machine has "word" size** Basic size of register
  - nominal size of general-purpose register
  - 64 bit for most machines these days

- **Addresses specify byte locations**
  - Address of first byte in machine word
  - Addresses of successive words differ by 8 (64-bit) or 4 (32-bit)

Normally Address Length == Word Size



# Word-Oriented Memory Organization




- From the processor core's point of view, memory is still byte-addressable one-word.

memory capacity




CSE 컴퓨터공학부
Department of Computer Science & Engineering

| | 0 | 1 | 2 | 3 | | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|---|---|---|---|
| … | | | | | | | | | |
| 0x104 | | | | | | | | | |
| 0x100 | 12 | 34 | 56 | 78 | vs. | 78 | 56 | 34 | 12 |
| 0x0fc | | | | | | | | | |
| … | | | | | | | | | |

# Data Types and Representations

# Data Types

| C Data Type | Size (bytes) Typical 64-bit | x86-64 | 32 bit Intel IA32 |
|---|---|---|---|
| char | 1 | 1 | 1 |
| short | 2 | 2 | troll 2 |
| int | 4 | 4 | 4 |
| **long** | 8 | 8 | 4 |
| long long | 8 | 8 | 8 |
| float emulate floating pointer | 4 | 4 | 4 |
| double or execute in another processor | 8 | 8 | 8 |
| long double | 8 other case, basically same | 10/16 ? | 10/12 |
| *** (pointer)** : memory address | 8 | 8 | 4 |

# Data Types – Integer

- **Encode a natural number in binary representation**
- **Separate signed and unsigned types**
  - `unsigned char, [signed] int, unsigned long long`

| C Data Type | Size (bytes) | Range | |
|---|---|---|---|
| | | Unsigned | Signed |
| char | 1 | 0 – 255 | -128 – 127 |
| short | 2 | 0 – 65,535 | -32,768 – 32,767 |
| int | 4 | 0 – 4,294,967,296 | -2,147,483,648 – 2,147,483,647 |
| long | *(same as int on 32-bit, same as long long on 64-bit architectures)* | | |
| long long | 8 | 0 – 18.4467 x $10^{18}$ | -9.2234 x $10^{18}$ – 9.2234 x $10^{18}$ |


CSE 컴퓨터공학부 Department of Computer Science & Engineering

# Data Types – Floating Point

- **Encode a real number as sign bit, mantissa, and exponent in binary format**
  - Number of bits available for mantissa and exponent differ by type
  - Precision differs in dependence of the encoded number

| C Data Type | Size (bytes) | Range |
|---|---|---|
| float | 4 | 0.0, $\pm 1.40130 \times 10^{-44}$ – $\pm 3.40282 \times 10^{38}$ |
| double | 8 | 0.0, $4.94066 \times 10^{-324}$ – $\pm 1.79769 \times 10^{308}$ |
| long double | 10 | 0.0 – $\pm 1.18 \times 10^{4932}$ |

# Data Types – Pointers

- **Represent a <u>memory address</u>**
  - Size matches architecture's word size

32 bits system

| C Data Type | Size (bytes) | Range | Remarks |
|---|---|---|---|
| pointer | 4 | $0 - 2^{32}$ | 32-bit architecture |
| | 8 | $0 - 2^{64}$ | 64-bit architecture |

in 64 bit system

# Data Types – Strings : array of character

- **C strings**
  - Conceptually an array of characters: char[]
  - Encode ASCII value of each character (https://www.ascii-code.com)
  - Terminated by null-character (null bytes, numeric value 0)

↳ to represent end of string!

newline

`"Hello, CSAP!\n"`, '\t' : tab-space

null-terminate

| H | e | l | l | o | , | | C | S | A | P | ! | \n | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 48 | 65 | 6c | 6c | 6f | 2c | 20 | 43 | 53 | 41 | 50 | 21 | 0a | 00 |

# Data Types – Examples

| C Data Type | Value | Hex Encoding |
|---|---|---|
| char    1 byte | 1 | 01 |
| | 'A' | 41 |
| short    2 byte | 7375 | 1ccf |
| | -1 | ffff |
| unsigned int    4 byte | 0 | 00000000 |
| (1's Complement) | 7375 | 00001ccf |
| long long    8 byte | -1 | ffffffffffffffff |
| float    4 byte | 3.141592653 | db0f4940 |
| int*    8 byte (Memory address) | last word | 00007ffceb5d9734 |
| char[6] | "Hello" \0 | 48656c6c6f00 |

# Byte Ordering

- **How should bytes within a multi-byte word be stored in memory?**
  - int v (4 bytes)

`v = 0x12345678`

| | | | |
|---|---|---|---|
| 12 | 34 | 56 | 78 |

MSB LSB

(last) **M**ost **L**east (first)

**S**ignificant **B**yte

by ChatGPT: In hex representation, closed to 0x: MSB
far from 0x: LSB

  - Two (sensible) options

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| … | | | | |
| 0x104 | | | | |
| 0x100 | 12 | 34 | 56 | 78 |
| 0x0fc | | | | |
| … | | | | |

Big-endian

vs.

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| … | | | | |
| 0x104 | | | | |
| 0x100 | 78 | 56 | 34 | 12 |
| 0x0fc | | | | |
| … | | | | |

Little-endian

# Byte Ordering

- **Big endian** : MSB first
  - Least significant byte has highest address
  - PowerPC, ARM, the Internet
- **Little endian** : LSB first (L-L)
  - Least significant byte has lowest address
  - Intel, RISC-V, ARM

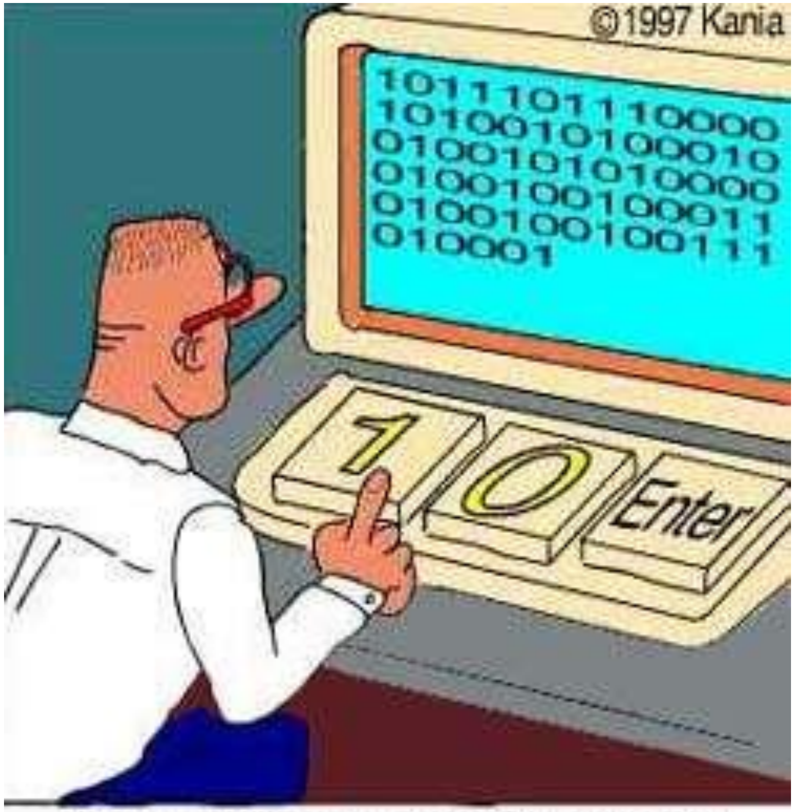
```
*(int*)(0x100) = 0x12345678
```

| | … | 0xff | 0x100 | 0x101 | 0x102 | 0x103 | 0x104 | … |
|---|---|---|---|---|---|---|---|---|
| Big Endian | | | 12 | 34 | 56 | 78 | | |
| Little Endian | | | 78 | 56 | 34 | 12 | | |

# Byte Ordering

- **Byte ordering is only an issue for basic data types with a size > 1 byte**
  - short, int, long, long long
  - float, double, long double
  - pointers

- **<u>Composite data types stored according to separate rules</u>**
  - array
  - struct
  - union
  - more on this later



Real programmers code in binary.

# From Source Code to Machine Language

# From Source Program to Machine Code

- **Processors do not understand high-level programming languages**
  - the interface to control a processor is called the **Application Binary Interface** (ABI)

    $\subset$

  - part of the ABI is the **Instruction Set Architecture (ISA)**
  - the ISA specifies an interface to execute single operations on a processor that change its state in a well-defined manner

    ```
    add 4(%rdi, %rsi, 2), %rax
    ```

- **Most people are not particularly good at writing assembly code**
  - high-level programming languages raise the level of abstraction from the ISA to a (human-friendly) programming language

    ```
    sum += A[1 + i]
    ```

# From Source Program to Machine Code

source code
↓
Preprocessor
↓
preprocessed source code
↓
Compiler
↓
assembly code
↓
Assembler
↓
relocatable machine code
↓
Linker/Loader
↓
target machine code

```c
#include <stdio.h>
#define N 1024

int main() {
  int A[N];

  a[i] = a[i] + 1;
  printf("%d", a[i]);
```

```c
int printf(…);
…

int main() {
  int A[1024];

  a[i] = a[i] + 1;
  printf("%d", a[i]);
```

```
  movl $a, %eax
  addl (%eax, %ebx, 4), %ecx
  …
  call printf
```

What we called as compile

```
6: e8 fc ff ff ff   call 7<main+0x14>
             7: R_386_PC32 printf
```

```
80483ba: e8 09 00 00 00   call 80483c8
```

# From Source Program to Machine Code

- **Preprocessing**

```
#include <stdio.h>
#include <stdlib.h>

#define N 1024

int main(int argc, char *argv[]) {
  int A[N], i;

  for (i=0; i<N; i++) A[i] = N-i;

  i = argc > 1 ? atoi(argv[1]) : 0;

  printf("A[%d] = ", i); fflush(stdout);
  printf("%d\n", A[i]);

  return EXIT_SUCCESS;
}
```

handle preprocessor directive
('#' code)
— #include
— #define
— #ifdef ... etc

```
$ wc p24.c
$ gcc -E p24.c > p24.pp.c
```

preprocess code

```
$ gcc --version
gcc (Gentoo 12.2.1_p20230121-r1 p10) 12.2.1 20230121
```

# From Source Program to Machine Code

- **Preprocessing**

```
#include <stdio.h>
#include <stdlib.h>

#define N 1024

int main(int argc, char *argv[]) {
  int A[N], i;

  for (i=0; i<N; i++) A[i] = N-i;

  i = argc > 1 ? atoi(argv[1]) : 0;

  printf("A[%d] = ", i); fflush(stdout);
  printf("%d\n", A[i]);

  return EXIT_SUCCESS;
}
```

실제 헤더 내용을 덮어씀

모든 N을 대체함

```
$ wc p24.c
$ gcc -E p24.c > p24.pp.c
$ wc p24.pp.c
$ vi p24.pp.c
```

```
# 0 "p24.c"
…
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "/usr/include/stdio.h" 1 3 4
…
# 6 "p24.c"
int main(int argc, char *argv[]) {
  int A[1024], i;

  for (i=0; i<1024; i++)
    A[i] = 1024 -i;

  …

  return 0;
}
```

replaced

formatted

# From Source Program to Machine Code (Compiler)

- **Compiling**

```
# 0 "p24.c"
…
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "/usr/include/stdio.h" 1 3 4
…
# 6 "p24.c"
int main(int argc, char *argv[]) {
  int A[1024], i;

  for (i=0; i<1024; i++)
    A[i] = 1024 -i;

  …

  return 0;
}
```

```
$ gcc -S p24.pp.c
```

to Assembly Code

# From Source Program to Machine Code

- **Compiling**

```
# 0 "p24.c"
…
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "/usr/include/stdio.h" 1 3 4
…
# 6 "p24.c"
int main(int argc, char *argv[]) {
  int A[1024], i;

  for (i=0; i<1024; i++)
    A[i] = 1024 -i;

  …

  return 0;
}
```

```
$ gcc -S p24.pp.c
$ vi p24.pp.s
```

assembly

```
    .file "p24.pp.c"
    .text
    .section      .rodata
…
main:
.LFB6:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
…
.L7:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

# From Source Program to Machine Code

- **Assembling**

```
    .file "p24.pp.c“
    .text
    .section      .rodata
…
main:
.LFB6:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq  %rsp, %rbp
…
.L7:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

```
$ gcc -c p24.pp.s
```

# From Source Program to Machine Code

Assembler

- **Assembling**

```
     .file"p24.pp.c“
     .text
     .section      .rodata
…
main:
.LFB6:
     .cfi_startproc
     pushq%rbp
     .cfi_def_cfa_offset 16
     .cfi_offset 6, -16
     movq %rsp, %rbp
…
.L7:
     leave
     .cfi_def_cfa 7, 8
     ret
     .cfi_endproc
```

Machine Code (Hex)

```
$ gcc -c p24.pp.s
$ hexdump -x p24.pp.o
```

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0001 003e 0001 0000 0000 0000 0000 0000
0000020 0000 0000 0000 0000 0468 0000 0000 0000
0000030 0000 0000 0040 0000 0000 0040 000e 000d
0000040 4855 e589 8148 30ec 0010 8900 dcbd ffef
0000050 48ff b589 efd0 ffff 4864 048b 2825 0000
0000060 4800 4589 31f8 c7c0 ec85 ffef 00ff 0000
0000070 eb00 b823 0400 0000 852b efec ffff c289
```

# From Source Program to Machine Code

- **Linking** : Link program to including file, runtime, etc..
  - p24.pp.o only contains machine code of main. : 즉 지금 Compile된 파일은 main 뿐, printf나 기타 함수는 준비안됨
  - C runtime and startup code missing

⇒ ∴ main에서 활용할 함수들의 Bytecode를 연결해야함 (PLT, GOT)

Dynamic Link

Static Link

```
$ gcc -o p24 p24.pp.o
$ ls -lrt
```

```
-rw-r--r-- 1 bernhard users   270 Feb  4 17:35 p24.c
-rw-r--r-- 1 bernhard users 43252 Feb  4 17:36 p24.pp.c
-rw-r--r-- 1 bernhard users  1372 Feb  4 17:42 p24.pp.s
-rw-r--r-- 1 bernhard users  2024 Feb  4 17:47 p24.pp.o
-rwxr-xr-x 1 bernhard users 15664 Feb  4 17:54 p24
```

-E → -S → -c → -o

PP S O exe



CSE 컴퓨터공학부 Department of Computer Science & Engineering

# From Source Program to Machine Code

- **Inspecting the executable file**

```
$ readelf –a p24
$ objdump –d p24
```

```
p24:     file format elf64-x86-64

Disassembly of section .text:

0000000000001080 <_start>:
 1080:    31 ed                   xor    %ebp,%ebp
 1082:    49 89 d1                mov    %rdx,%r9
 1085:    5e                      pop    %rsi
 ...

0000000000001165 <main>:
 1165:    55                      push   %rbp
 1166:    48 89 e5                mov    %rsp,%rbp
 1169:    48 81 ec 30 10 00 00    sub    $0x1030,%rsp
 1170:    89 bd dc ef ff ff       mov    %edi,-0x1024(%rbp)
 1176:    48 89 b5 d0 ef ff ff    mov    %rsi,-0x1030(%rbp)
 ...
```

# Reading Byte Listings

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

- **C source**

```
int compare(long long v) {
    if (v == 0x123456789L) return 0;
    else return 1;
}
```

**x86-64 machine code**

```
48 b8 89 67
45 23 01 00
00 00 48 39
c7 0f 95 c0
0f b6 c0 c3
```

- We expect to find the constant 0x123456789 in the machine code
  - Do you see it?

# Reading Byte Listings

- **x86-64 machine code (little endian)**

```
Bytecode                            Assembly
…                                   …
48 b8 89 67 45 23 01 00 00 00       movabs  $0x123456789, %rax
48 39 c7                            cmpq    %rax, %rdi
0f 95 0c                            setne   %al
…                                   …
```

# Reading Byte Listings

- **x86-64 machine code (little endian)**

```
Bytecode                                   Assembly
…                                          …
48 b8 89 67 45 23 01 00 00 00              movabs  $0x123456789, %rax
48 39 c7                                   cmpq    %rax, %rdi
0f 95 0c                                   setne   %al
…                                          …
```

Long : 8byte

- value — 0x123456789
- extend to 64-bit — 0x0000000123456789
- split into bytes — 00 00 00 01 23 45 67 89
- little endian — 89 67 45 23 01 00 00 00

LSB

MSB

- [Intel® 64 and IA-32 Architectures Software Developer Manuals]

# Reading Byte Listings

- **RISC-V machine code (little endian)**

```
Address     Bytecode        Assembly                       Comments
  10580     00 01 27 b7     lui  a5, 0x12                  a5 = 0x12000
  10584     02 07 b7 83     ld   a5, 32(a5)                a5 = MEM[a5+32]
  10588     40 f5 05 33     sub  a0, a0, a5
  1058c     00 a0 35 33     snez a0, a0
  10590     00 00 80 67     ret
    ...
  12020     89 67 45 23      constant stored in little endian format and loaded
  12024     01 00 00 00      into register a5
    ...
```

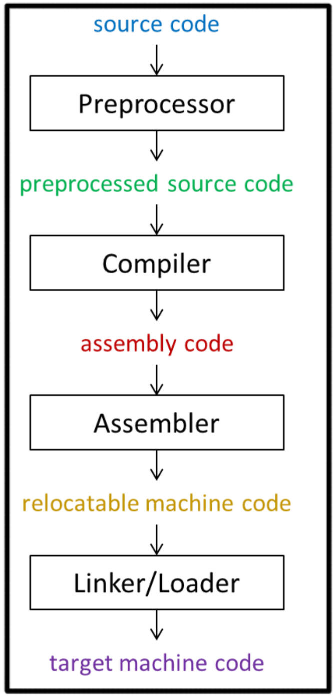
- [The RISC-V Instruction Set Manual, Volume I: User-Level ISA (v2.0)]

# Reading Byte Listings

- **ARM64 machine code (big endian)**

```
Address     Bytecode       Assembly                            Comments
 4006f0     d2 8c f1 21    mov  x1, #0x6789                    bits 15:0
 4006f4     f2 a4 68 a1    movk x1, #0x2345, lsl #16           bits 31:16
 4006f8     f2 c0 00 21    movk x1, #0x1, lsl #32              bits 47:32
 4006fc     eb 01 00 1f    cmp  x0, x1
 400700     9a 9f 07 e0    cset x0, ne
 400704     d6 5f 03 c0    ret
```

  - constant built with first three instructions; not directly visible in bytecode because of the complex immediate encoding of the mov/movk instructions.
  - ARM® A64 Instruction Set Architecture, mov (pp. 765-), see p. 809 for immediate encoding/decoding




# Module Summary

# Module Summary

- **Information = Bits + Context**
  - integral C data types (=context) represented as bits
  - N bits allow for $2^N$ distinct representations
  - different data types occupy a different number of bytes

- **Word-oriented memory organization**
  - little vs. big endian
  - little endian architectures: Intel, RISC-V
  - big endian architectures: ARM, PowerPC, the Internet

- **Machine code**
  - represented as bit sequences
  - compiled from high-level code