# CSED211 : Introduction to Computer SW Architecture

## Lecture 1: Introduction

Jong Kim

Pohang Univ. of Sci. & Tech.
Dept. of Comp. Sci. & Eng.

# Course Goal

- Goal of this course
  - Provide the basic concept of computer organization and low-level programming using microprocessor's internal structure and assembly programming.
  - Study the Intel and ARM assembly languages and practice assembly programming skill
  - Provide the concept of managing system components in low-level through system programming
- This course consists of Lecture and Lab
  - Lecture covers the basic concept of computer organization, low-level programming using assembly language, and system programming.
  - During the Lab, you will study the details of assembly programing skill and perform projects (with the help of TAs).

# Text and References

- Main text book
  - Computer Systems: A programmer's Perspective by R. Bryant and D. O'Hallaron, Prentice Hall, 3rd ed.

# Evaluation Policy

- Midterm 20%
  Final exam 30%
  Homework/Quiz  15% (quiz without notice)
  Labs & Term Projects 30%
  Class-attendance 5%

- Term project
  - Proposal, Progress report, Demo & Presentation, and Final report

- Attendance (+ alpha)
  - At least, your minimal requirement must be satisfied strictly.
    - If not, “F” with no exception and no excuse

# Class Homepage

- Power point slides and announcement are released through

  POVIS LMS & e-class system

- Also, all discussions will be made through LMS system.

# FYI

- Contact information
  - Office: PIRL #438
  - Office Hour : Monday 4:00pm ~ 5:00pm
  - Thursday: 11:00~12:00am
  - Tel: 2257
  - Email: jkim@postech.ac.kr
  - TAs: 조범진, 박남규, 박채용

# Course Overview

# Overview

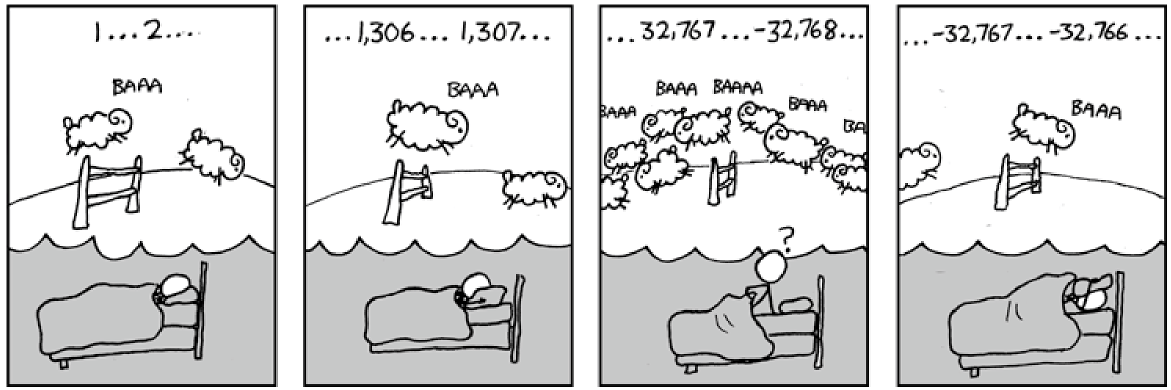
- Course theme
- Five real-world issues
- Course & Lab Info

# Course Theme: Learn how real-world systems work!

- Most CS and CE courses emphasize abstraction
  - Abstract data types e.g., integer, character, ..
  - Asymptotic analysis e.g., O(*n*), O(*n Log(n)*), ..
- These abstractions have limits
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- Useful outcomes
  - Become more effective programmers
    - Able to find and eliminate bugs efficiently
    - Able to understand and tune for program performance
  - Prepare for later "systems" classes
    - Computer Architecture, Operating Systems, Compiler, Networks, Embedded Systems, …

# Great Reality #1: Ints are not Integers, Floats are not Reals

- Example 1: Is $x^2 \geq 0$?
  - Float's: Yes!




  - Int's:
    - 40000 * 40000 → 1600000000
    - 50000 * 50000 → ??
- Example 2: Is (x + y) + z = x + (y + z)?
  - Unsigned & Signed Int's: Yes!
  - Float's:
    - (1e20 + -1e20) + 3.14 --> 3.14
    - 1e20 + (-1e20 + 3.14) --> ??

Source: xkcd.com/571

# Computer Arithmetic

- Does not generate random values
  - Arithmetic operations have their own important mathematical properties
- Cannot assume all “usual” mathematical properties
  - Due to finiteness of representations
  - Integer operations satisfy “ring” properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy “ordering” properties
    - Monotonicity, values of signs
- Observation
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers

# Great Reality #2: You Must Know Assembly

- Chances are, you'll never write programs in assembly
  - Compilers are much better & more patient than you are
- But: Understanding assembly is key to machine-level execution model
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system & system software
    - CPU understands only assembly-based execution model
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!

# Great Reality #3: Memory Matters

## Random Access Memory (RAM) is an Unphysical Abstraction

- Memory is not unbounded
  - It must be allocated and managed
  - Many applications are memory dominated
- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
- Memory performance is not uniform
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

# Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

| | | |
|---|---|---|
| `fun(0)` | → | 3.14 |
| `fun(1)` | → | 3.14 |
| `fun(2)` | → | 3.1399998664856 |
| `fun(3)` | → | 2.00000061035156 |
| `fun(4)` | → | 3.14 |
| `fun(6)` | → | Segmentation fault |

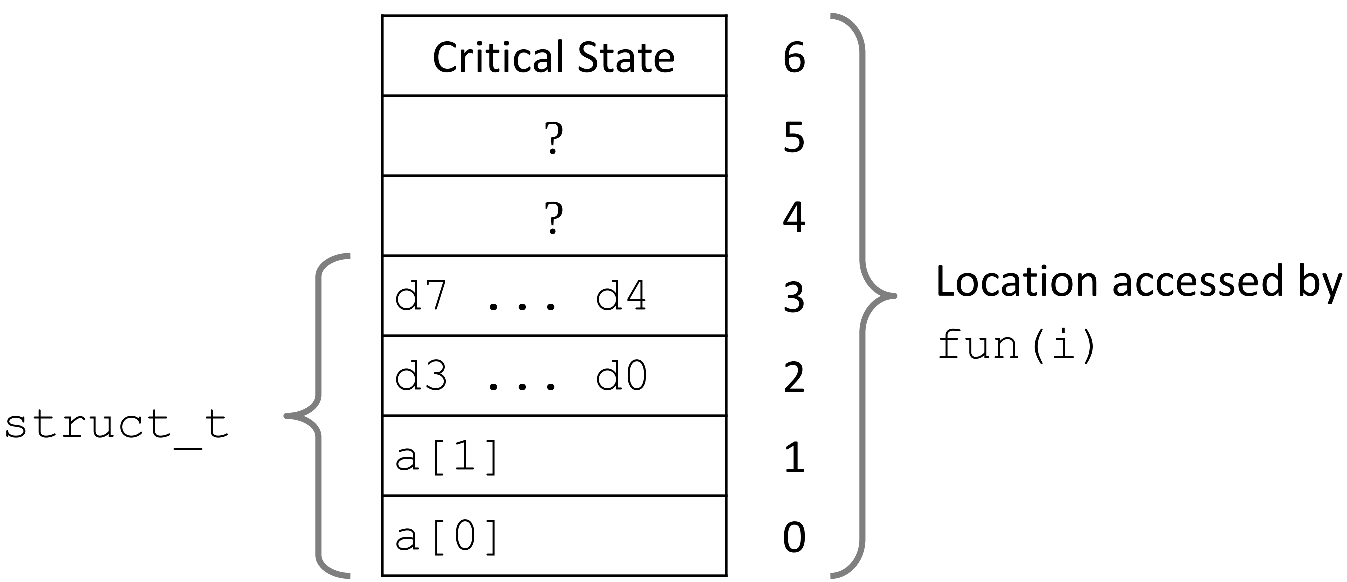
– Result is system specific

# Memory Referencing Bug Example

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  ➙  3.14
fun(1)  ➙  3.14
fun(2)  ➙  3.1399998664856
fun(3)  ➙  2.00000061035156
fun(4)  ➙  3.14
fun(6)  ➙  Segmentation fault
```

**Explanation:**



# Memory Referencing Errors

- C and C++ do not provide any memory protection
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- Can lead to nasty bugs
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- How can I deal with this?
  - Program in other languages (e.g, Java, Ruby or ML)
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)

# Great Reality #4: There's more to performance than asymptotic complexity

- Constant factors matter too!
- And even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs are compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

# Memory System Performance Example

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

59,393,288 clock cycles

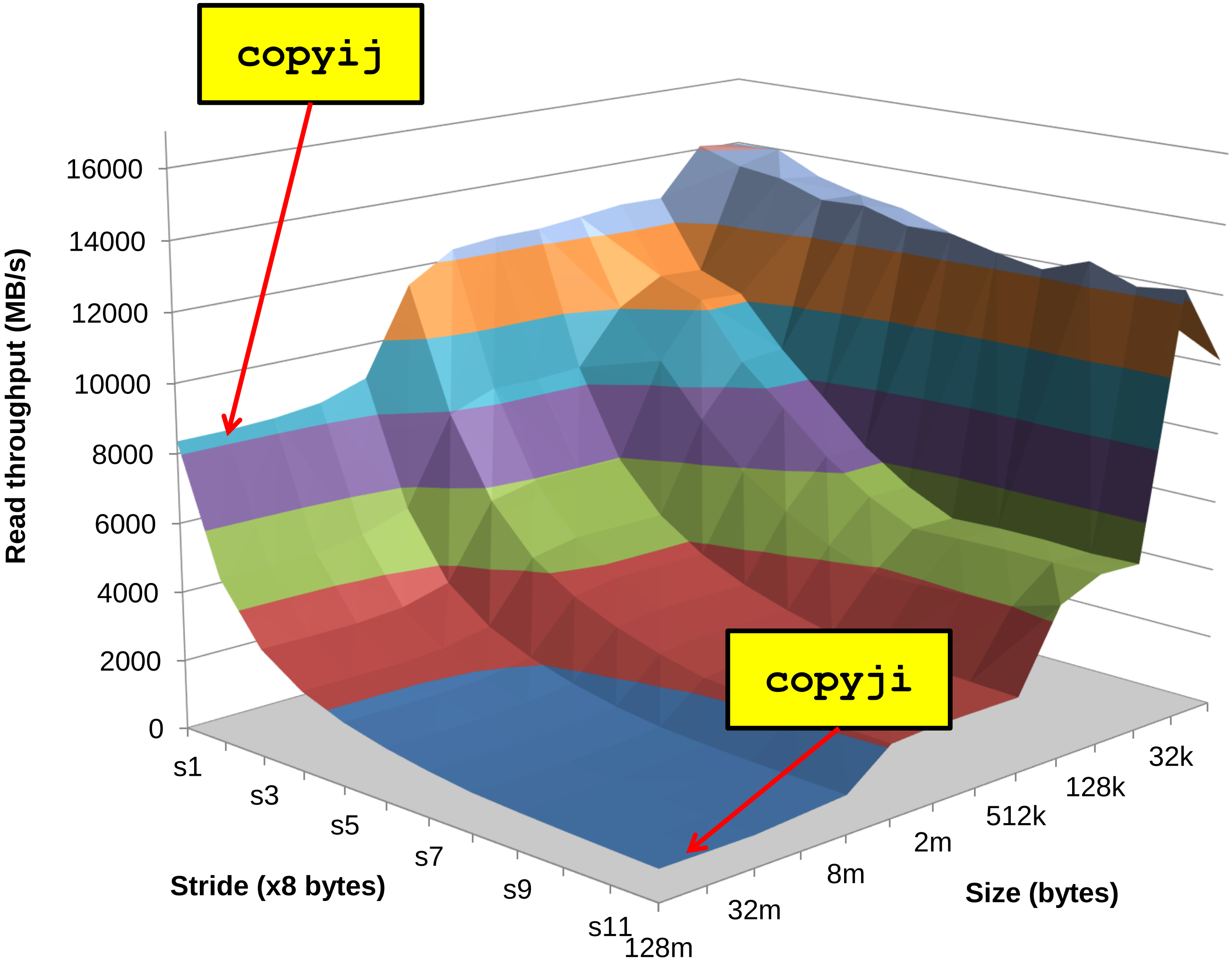
```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

1,277,877,876 clock cycles

21 times slower
(Pentium 4)

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array

# Why The Performance Differs



# Great Reality #5: Computers do more than execute programs

- They need to get data in and out
  - I/O system critical to program reliability and performance

- They communicate with each other over networks
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues

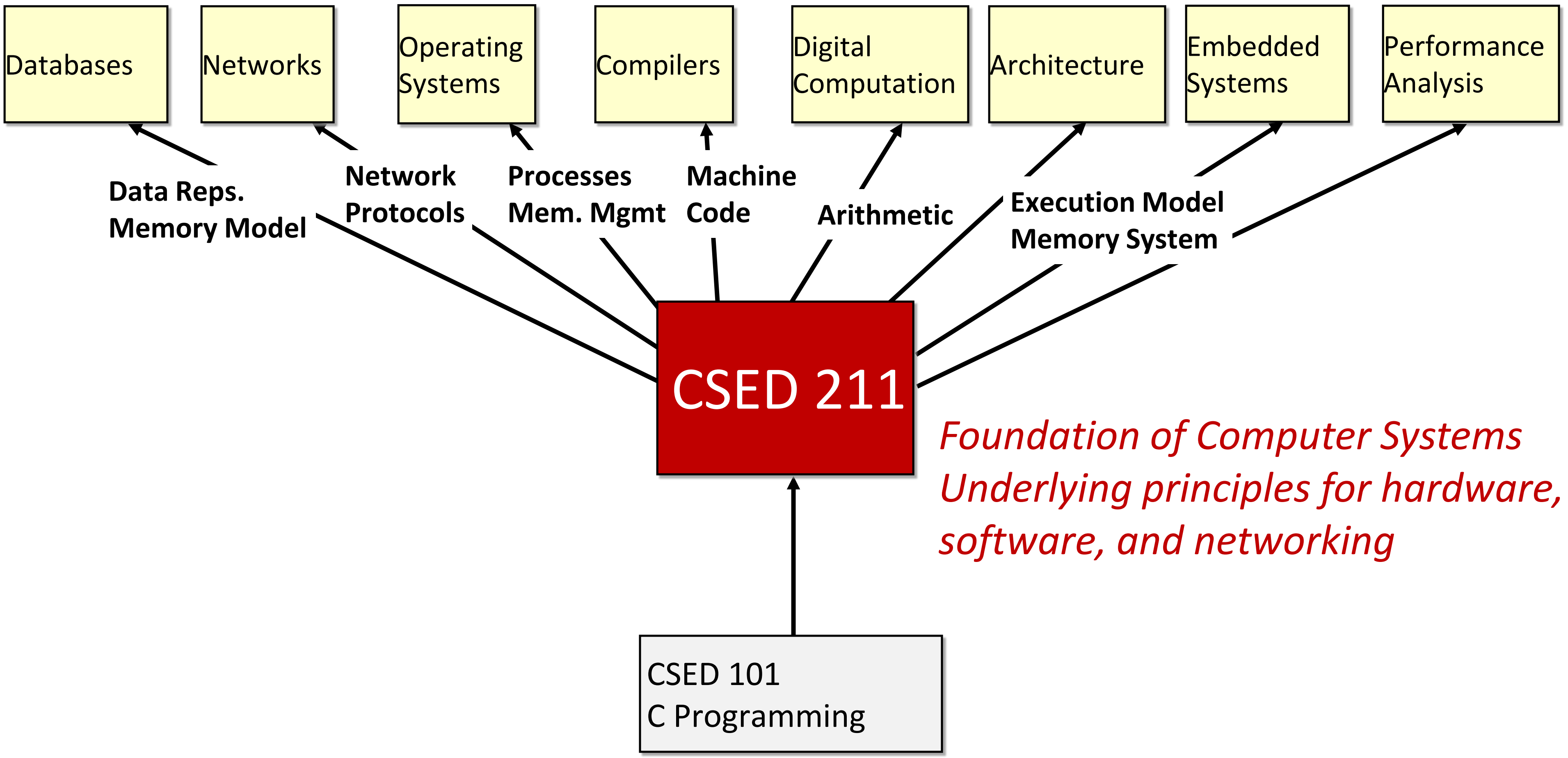
# Course Perspective

- Most Systems Courses are Builder-Centric
  - Computer Architecture
    - Design pipelined processor in Verilog
  - Operating Systems
    - Implement sample portions of operating system
  - Compilers
    - Write compiler for simple language
  - Networking
    - Implement and simulate network protocols

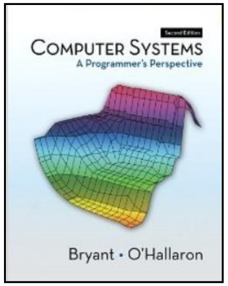
# Course Perspective (Cont.)

- Our Course is Programmer-Centric
  - Purpose is to show that by knowing more about the underlying system, one can be more effective as a programmer
  - Enable you to
    - Write programs that are more reliable and efficient
    - Incorporate features that require hooks into OS
      - E.g., concurrency, signal handlers
  - Cover material in this course that you won't see elsewhere
  - Not just a course for dedicated hackers
    - **We bring out the hidden hacker in everyone!**

# Role within CS Curriculum



# Be a good starter!




**Start Reading Today!**

| | |
|---|---|
| **Ch. 1** | **(today)** |
| **Ch. 2.1** | **(next lecture)** |
| **Ch. 2.2~2.3** | **(following lecture)** |