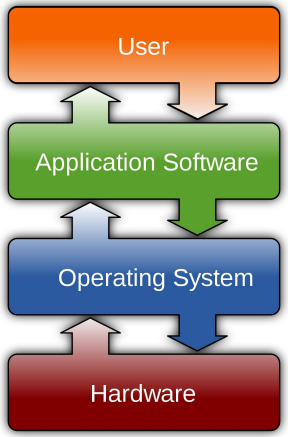
Computer System and Compilation

Computer Systems

- Consist of software and hardware
- Software (SW)
  - Application
  - OS
- Hardware (HW)
  - Processors: CPU, GPU
  - I/O devices: Keyboard, Mouse, Monitor
  - Buses, Memory, etc



System Stack

Example Benefits of Understanding Computer Systems

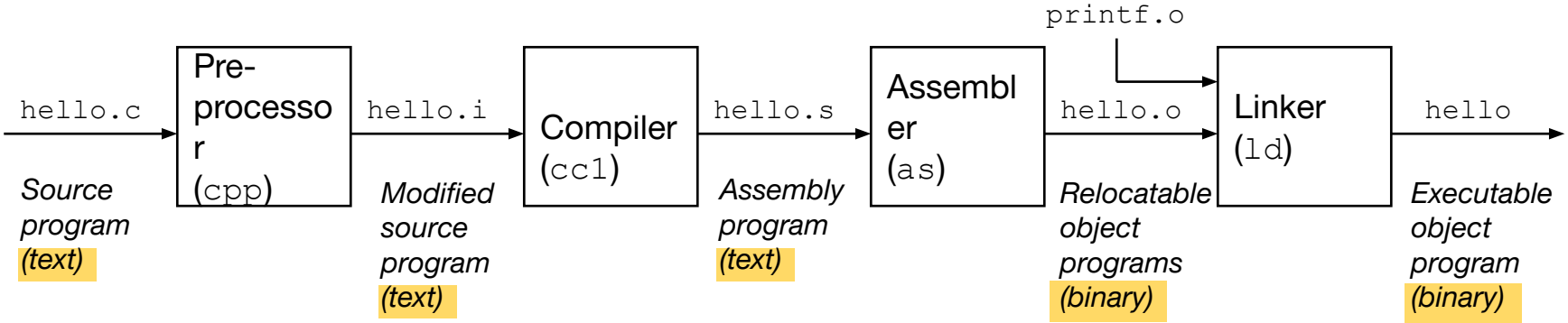
- Avoid strange numerical errors caused by number representation
- Optimize code by using clever tricks exploiting the designs of modern processors and memory systems
- Avoid security holes from buffer overflow vulnerabilities
- Learn the promises and pitfalls of concurrency
- Become a power programmer

Computer Systems

- Everything is represented by a sequence of “0” or “1” (binary) in digital systems. What distinguishes different information is the context in which we view them. The same binary numbers can represent different things such as integer, character string, machine instruction, etc.
- Programs are translated by other programs into different forms
- Processors read and interpret instructions stored in memory
- Storage devices form a hierarchy
- Operating system manages hardware
- Systems communicate with other systems using networks

Compilation

- Translation of program in one language into another language
  - Typically from high-level language to machine language
- Programmer creates a source program (source file), hello.c
  - Source program is a text file where each text character is encoded with ASCII standard (an example on next page)
- Compiler generates a machine program, hello



Example Program Source File (hello.c)

```c
#include <stdio.h>

int main()
{
  printf("Welcome to CSCI 223!");
  return 0;
}
```

```
2369 6e63 6c75 6465 203c 7374 6469 6f2e
683e 0a0a 696e 7420 6d61 696e 2829 0a7b
0a20 2070 7269 6e74 6628 2257 656c 636f
6d65 2074 6f20 4353 4349 2032 3233 2122
293b 0a20 2072 6574 7572 6e20 303b 0a7d
0a
```

- This is the contents of hello.c in binary format (displayed in hex)
- Source file, hello.c, is encoded using ASCII
- Each two digit represents a single character - # (0x23), i (0x69), n (0x6e), c (0x63), l (0x6c), u (0x75), d (0x64), and so on.

Assembly Language

- Low-level programming language which is a human readable, textual representation of machine code
- Each statement corresponds to a single machine instruction
- Each assembly language is specific to a particular processor architecture (Instruction Set Architecture, ISA)
- Unlike high level languages, it contains lots of hardware information

```
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $0, -4(%rbp)
leaq    L_.str(%rip), %rdi
movb    $0, %al
callq   _printf
xorl    %eax, %eax
addq    $16, %rsp
popq    %rbp
retq
```

Programming at Different Levels

- High Level Programming – C/C++, Java, C#, Ruby, Python, etc
- Low Level Programming – ISA specific (x86, ARM, PPC, etc)

(relative comparison)	High Level Programming	Low Level Programming
Programming Difficulty	Low	High
Readability	High	Low
Debugging	Easy	Difficult
Potability	High	Low
Maintainability	High	Low

C Programing Basics

C Programming Language

- A small, simple, close-to-metal programming language
- Originally developed at Bell Labs by Dennis Ritchie in 1972
- One of the oldest and most popular high level programming languages
- Unix was written in almost entirely in C
- Language of choice for systems such as OS and device drivers
- Provide constructs that map efficiently to typical machine instructions
- Support the use of pointers, a type of reference that records the address of an object or function
- C++ is an extension of C with object-oriented programming paradigm
- C is a subset of C++

C Programming Basics

- C program that prints “Hello World” on the screen

```
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

Preprocessor command that includes standard input output header file (stdio.h) from the C library before compiling a C program

Main function from where execution of any C program begins.

Library function that prints the output onto the screen.

Statement that terminates C function with return a value

C Programming Basics

- Keywords and Identifiers
- Variables, Constants and Literals
- Data types
- Input and Output
- Arithmetic operators: +, -, *, /, %, ++, --, +=, -=, &&, ||, etc
- Flow control: if-else, for loop, while loop, break and continue, switch, etc
- Functions
- Arrays and Strings
- Pointers

C Programming Basics

- https://www.programiz.com/c-programming

Source Program Files, Compile, and Run

- C source program typically consists of source and header files
- Source files are stored in a plain text format with .c extension
  - .cpp extension for C++ source files
- Header files are stored in a plain text format with .h extension
  - .hpp extension for C++ header files
- Many C compilers exist
  - https://en.wikipedia.org/wiki/List_of_compilers#C_compilers
  - GCC, an open source GNU C compiler, compiles with the following command in terminal: `gcc hello.c -o hello`
- To run, simply type an executable command and press enter in terminal: `hello`

Compiler Flags

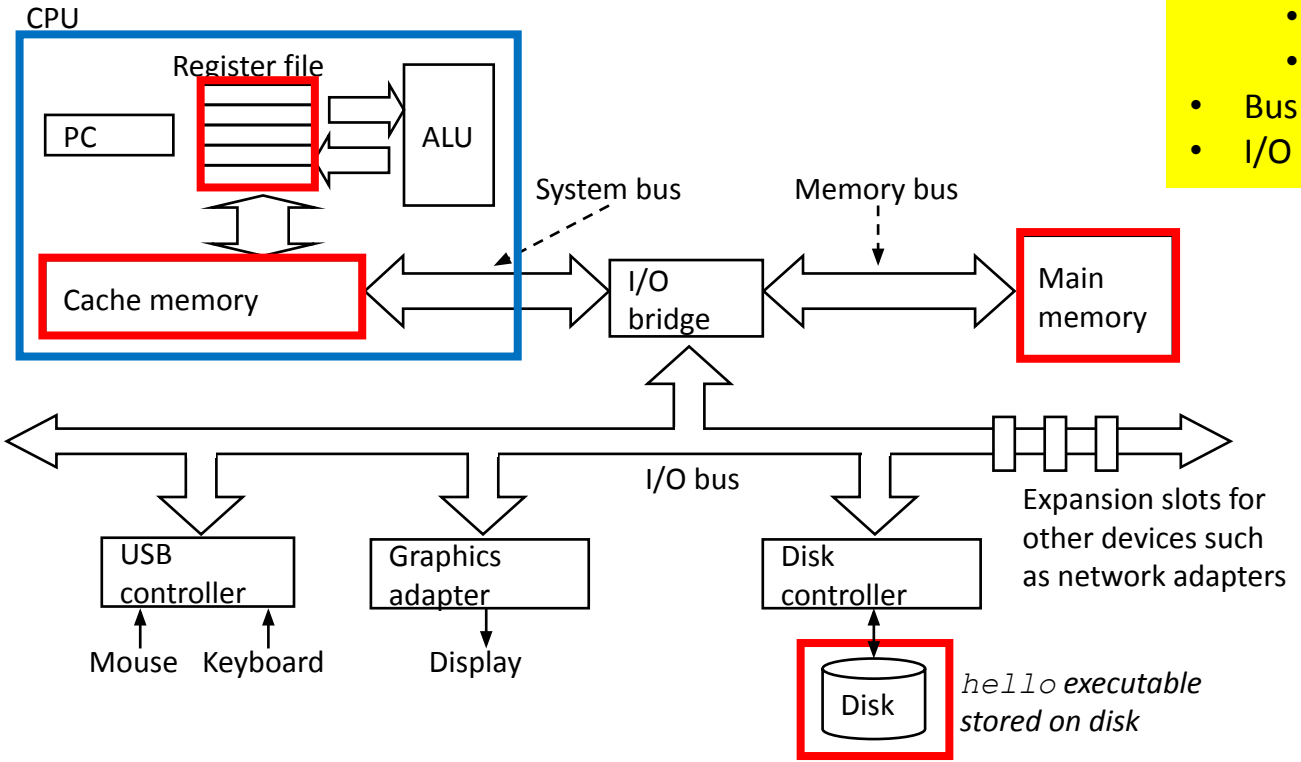
- Compilers (including GCC) offer a number of compilation controls via flags
  - Optimization levels: -O2, -O1, -O0 (e.g., `gcc -O1 hello.c`)
  - Debugging information: -g (e.g., `gcc -g hello.c`)
  - Modified source code: -E (e.g., `gcc -E hello.c`)
    - Compiler stops after the preprocessing stage (1st stage)
  - Assembly program: -S (e.g., `gcc -S hello.c`)
    - Compiler stops after the compiling stage (2nd stage)
    - It outputs "hello.s" assembly program in the current working directory
  - Relocatable object file: -c (e.g., `gcc -c hello.c`)
    -
    - It outputs "hello.o" object file in the current working directory
    - Relocatable object file is no longer a text file
  - To view a manual: `man gcc`
    - On Mac, use "`man clang`" because gcc is soft-linked to clang compiler

System Organization

System Organization

Main components

- CPU
- Memories
  - Disk
  - Main memory
  - Cache
  - Registers
- Bus
- I/O devices



System Components

Processor (CPU)

- Engine that interprets or executes instructions stored in memory
- Main components include ALU, registers (General-purpose, Special-purpose), and cache memories (L1, L2, L3).
- Repeatedly reads the instruction pointed at by the Program Counter (PC, a special register), interprets the bits in the instruction, perform some simple operations dictated by the instruction, and then updates the PC to point to the next instruction.

Register file

- Small yet fast storage consisting of a collection of registers, each with its own unique name (e.g., %rax, %rbx, %rsp, etc)

System Components

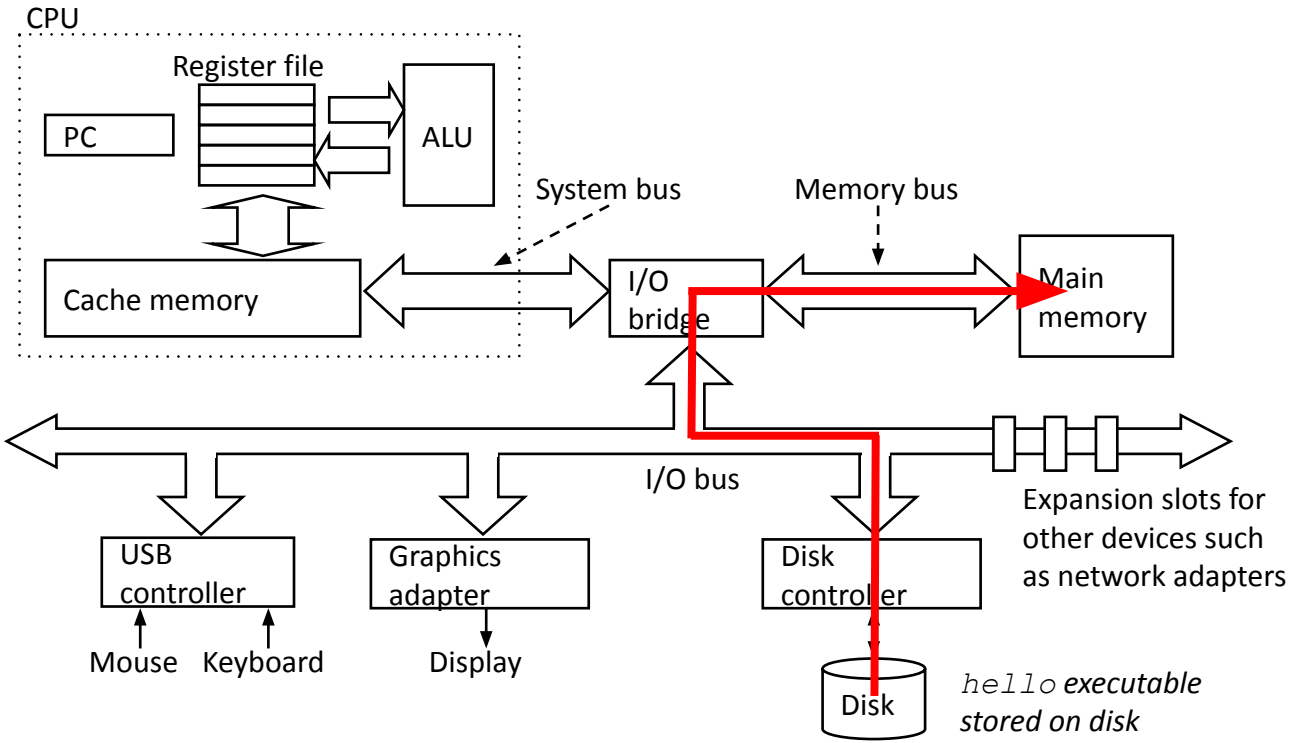
Main memory

- A temporary storage that holds both program and data
- Built with DRAM
- Logically organized as a linear array of bytes, each with its own unique address starting at zero

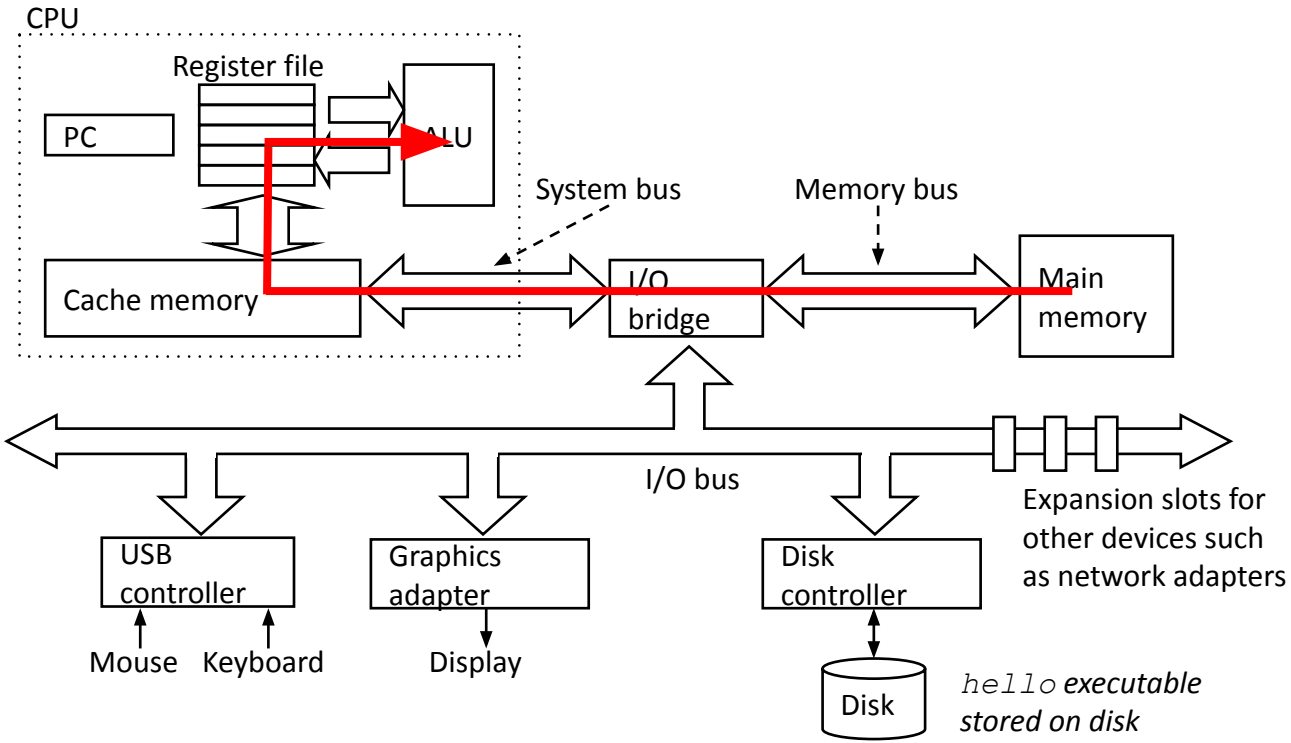
Buses

- A collection of electrical conduits that carry bytes of information back and forth between the components

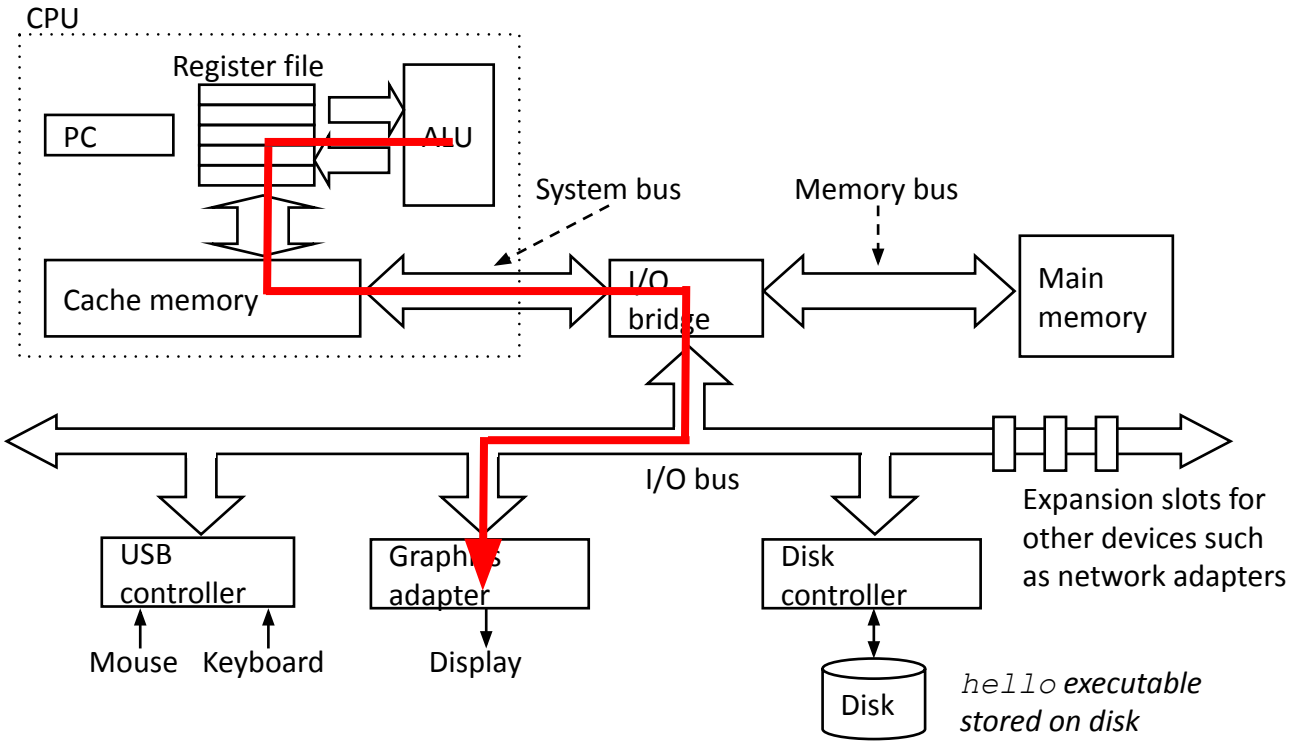
Program Execution



Program Execution



Program Execution



Observations

- System spends lots of time moving programs and data from one place to another
- System has multiple different memory spaces
  - Disk, Main memory, Cache, Registers
  - Memory spaces hold both program and data
  - All memory spaces except disk are temporary storages
- Computer system consists of hardware and system software that cooperate to run application programs
- Information inside the computer is represented as groups of bits that are interpreted in different ways, depending on the context

Amdahl's Law

When we speed up one part of a system, the effect on the overall system performance depends on both how significant this part was and how much it sped up.

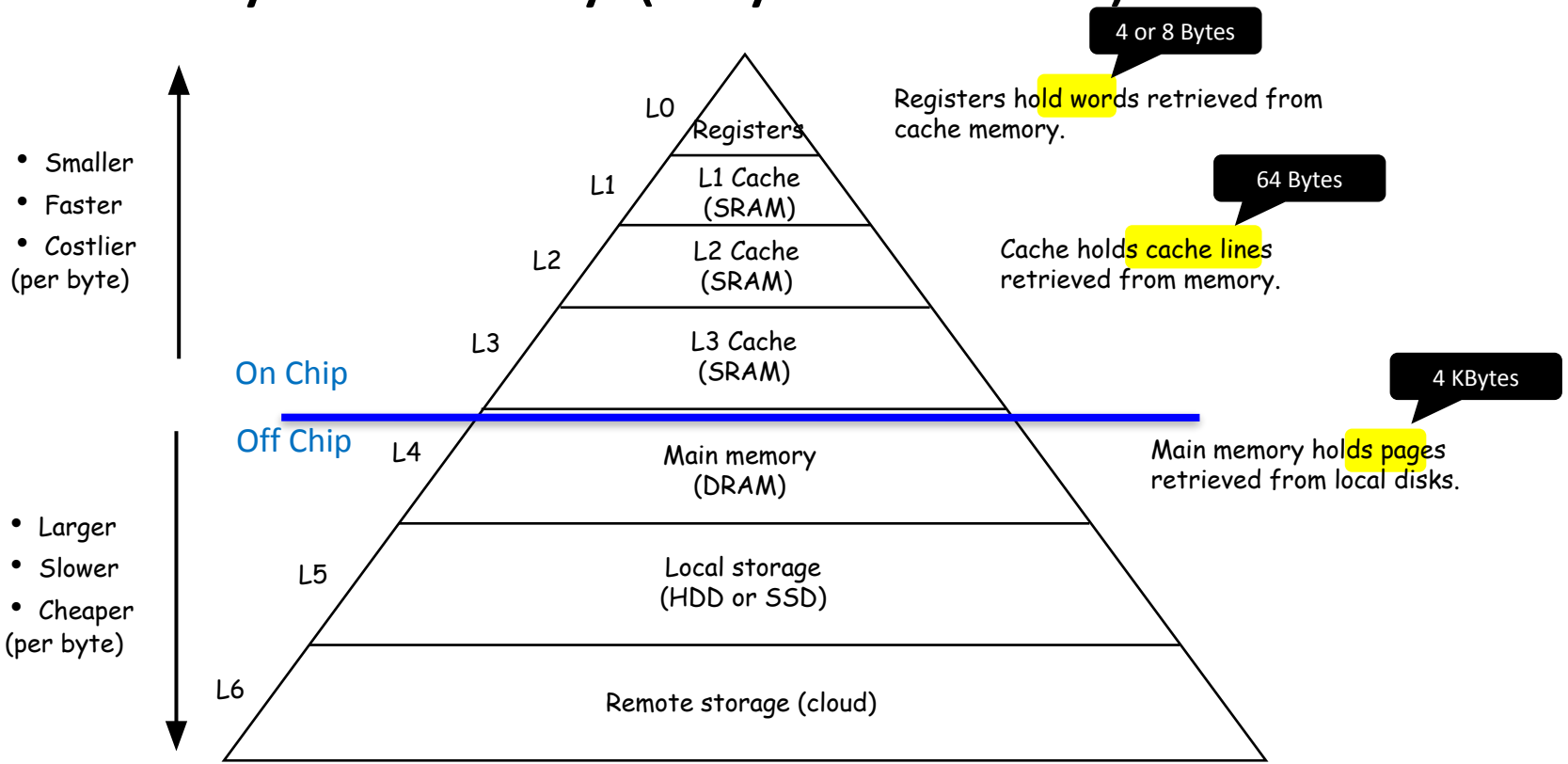
$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

f: Fraction of execution time benefiting from improvement
p: Speedup of the part benefiting from improvement

Example: A part of system that initially consumed 60% of the time (f = 0.6) is sped up by a factor of 3 (p = 3). The speedup is 1/((1-0.6)+0.6/3) which is 1.67x (67%)

Memory Organization

Memory Hierarchy (Physical View)



Memory Hierarchy (Physical View)

Motivation

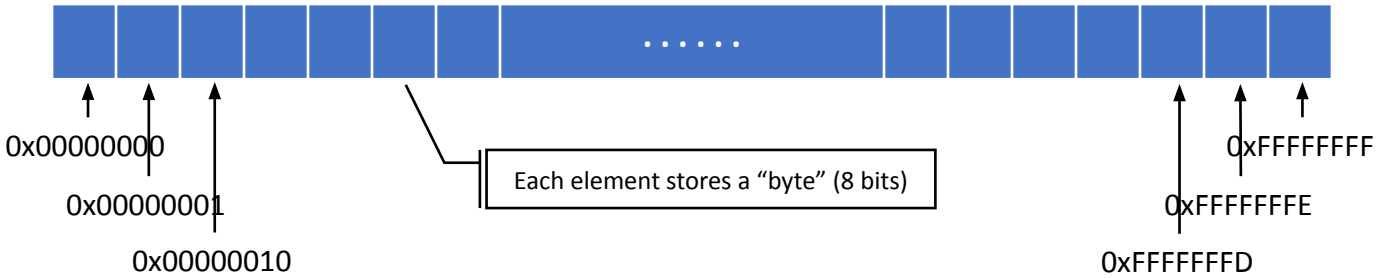
- Larger storage devices are slower than smaller ones
- Faster storages are more expensive to build than their slow counterparts
- Processors run faster than memory and disk (processor-memory performance gap)

Main idea

- Storage at one level serves as a cache for storage at the next lower level (e.g., register file serves as a cache for L1 cache)
- Goal is to feed CPU as fast as possible

Memory (Logical View)

- Program views memory as a very large array of bytes
  - Referred to as Virtual Memory
  - 4294967296 bytes (4GB) in 32-bit machine (2^{32} = 4294967296)
  - Every byte of memory is identified by a unique number called memory address
  - Physically implemented with a hierarchy of different memory spaces (pyramid shape)



Cache Matters

- Cache memory performance depends on memory access patterns

```
// Version 1
int i, j;
for (i = 0; i < 512; i++)
  for (j = 0; j < 512; j++)
    dst[j][i] = src[j][i];
```

```
// Version 2
int i,j;
for (j = 0; j < 512; j++)
  for (i = 0; i < 512; i++)
    dst[j][i] = src[j][i];
```

Q: Which one do you think is faster? By how much?

Row & Column Major Ordering

Methods for storing multidimensional arrays in linear storage such memory

Depending on which elements of an array are contiguous in memory

- Row Major Order
  - ✔ The consecutive elements of a row reside next to each other (row by row)
  - ✔ C/C++ and default in most programming languages

A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]	A[2][0]	A[2][1]	A[2][2]

- Column Major Order
  - ✔ The consecutive elements of a row reside next to each other (column by column)
  - ✔ Fortran

A[0][0]	A[1][0]	A[2][0]	A[0][1]	A[1][1]	A[2][1]	A[0][2]	A[1][2]	A[2][2]

Binary Representation

Binary Representation

- Everything is represented as a sequence of binary numbers in digital systems
- Then, how do we know what a binary number represents?

  0001000101011110010100010101000111101000010101 0…

We need context to understand binary numbers

Standards

- Once context is known, we need to know what standard to use to interpret binary numbers
- ASCII – a widely used standard for characters
  - 01000001 represents “A” and 01100001 represents “a”
- RGB – a widely used standard for colors
  - 1111….1111 represents “white” and 0000….0000 represents “black”
- Two’s complement – a widely used standard for signed integers
  - 0……00000001 represents “1” and 1……11111111 represents “-1”
- IEEE 754 – a widely used standard for floating point numbers
  - 00111110100000……0 represents a float “0.25”

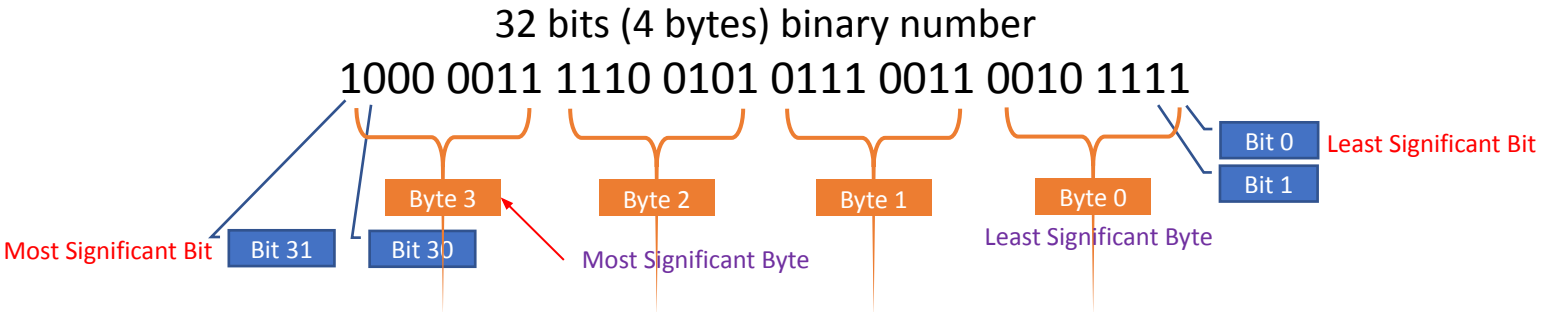
Example of Decoding Binary Numbers

$00000011010000101000010000_2$

Context	Standard to Use	Information Decoded
24-bit signed integer	Two's complement	214280
Color	RGB	
Machine code	32-bit x86	`addl 8(%ebp), %eax`

Bit and Byte

- Bit – each digit in binary representation
  - 0_2 or 1_2
  - Smallest storage unit
- Byte – a group of 8 bits
  - 00000000_2 ~ 11111111_2
  - 256 possible representations
  - Smallest memory access unit



Hexadecimal Notation

- Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’ (10 is A, 11 is B, and 15 is F)
- Prefix “0x” and case insensitive

  e.g., $FA1D37B_{16}$ == 0xFA1D37B == 0xfa1d37b
- Easy Conversions from/to Binary
  - A single hex digit is equivalent to 4 digits of binary (one to one mapping)

    1111 = F

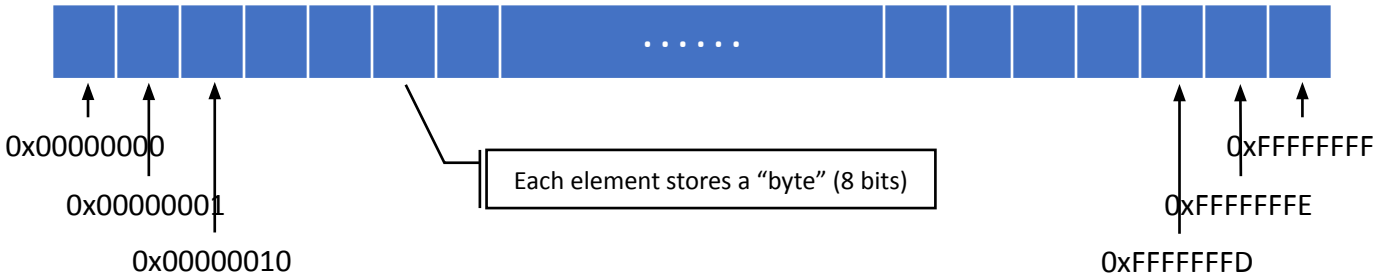
    0011 = 3

    11110011 = F3

    1111001111110011 = F3F3

Memory (Logical View)

- Program views memory as a very large array of bytes
  - Referred to as Virtual Memory
  - 4294967296 bytes (4GB) in 32-bit machine (2^{32} = 4294967296)
  - Every byte of memory is identified by a unique number called memory address
  - Physically implemented with a hierarchy of different memory spaces (pyramid shape)



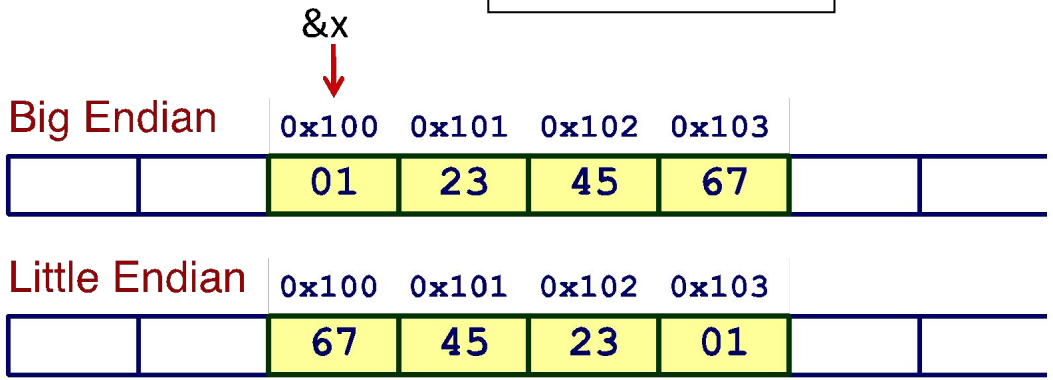
Byte Ordering (Endianness)

- Ordering of bytes within a multi-byte object in memory
  - Bits within a byte remain intact
- Two widely used systems
  - Big Endian System:
    - LSB (least significant byte) at highest address
    - Sun, PPC, Internet
  - Little Endian System:
    - LSB at lowest address
    - x86

Big Endian vs Little Endian

```
int x = 0x1234567;
```

Byte 0 (LSB) = 0x67
Byte 1 = 0x45
Byte 2 = 0x23
Byte 3 (MSB) = 0x01



Data Type, Pointer, Bitwise Operations

Data Types and Sizes

You can check the size of data types using `sizeof()` in C/C++

e.g. `printf("%ld", sizeof(char));`

C Data Type	Intel IA32	x86-64
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	4 bytes	4 bytes
`long`	4 bytes	8 bytes
`long long`	8 bytes	8 bytes
`float`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`pointer`	4 bytes	8 bytes

← Machine dependent

Misuse of Data Type

```
unsigned int tmp = 200 * 300 * 400 * 500;
printf("tmp = 0x%x\n", tmp);
```

0xCB417800

200 * 300 * 400 * 500 = 12,000,000,000
= 0x2CB417800

Word

- Data size that a machine is designed for (each machine has its word size)
- Size of memory address (size of pointer or register)
  - 32-bit machines use 32 bits (4 bytes) words (Limits virtual memory address to 4GB)
    - Became too small for memory-intensive applications
  - 64-bit machines use 64 bits (8 bytes) words (Limits virtual memory address to 1.8×10^{19} Bytes)
    - Most modern machines are designed with 64 bits (8 bytes) words
    - 64-bit x86 machines use 48 bit address (256TB), not all 64 bits – just a practical design choice
    - Backward compatibility (Most 64-bit machines can also run programs compiled for 32-bit machines)
    - To compile for 32-bit processors, use “-m32” flag

```
gcc –m32 hello.c
```

Pointers in C/C++

```
int i = 0x223;
int *p;
p = &i;
int j = *p;
```

- Variables whose contents are interpreted as memory address
  - Supposed to point other objects or functions
- Declaration: T *p; // "T" is a regular data type and "p" is a pointer variable

  int *p; // a pointer variable "p" is supposed to contain the memory address of integer variable

  float *q; // a pointer variable "q" is supposed to contain the memory address of float variable
- Initialization: p = &i // "i" is an integer variable
  - "i" is an integer variable declared earlier
  - & symbol indicates the address of the following object
    - "&i" is the memory address of variable "i"
- Dereferencing: *p; // "p" should be a pointer variable
  - "p" should be a pointer variable
  - Returns the value of the variable that the pointer variable points to
  - *p returns the value of "i" in this example

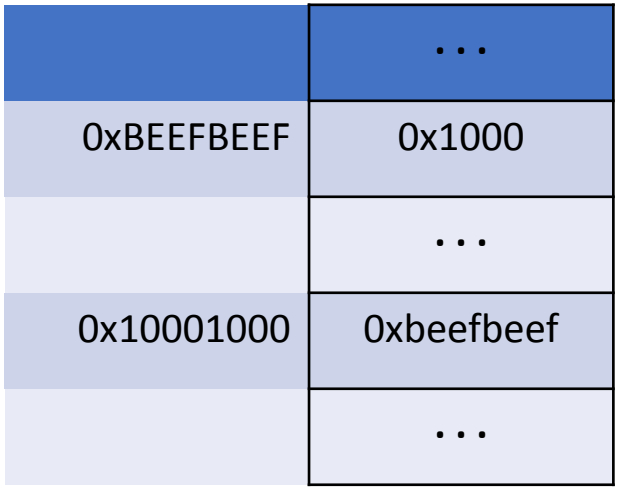
Pointer Example

```
int x = 0x1000;
int *p = &x;
printf("0x%x\n", &x);
printf("0x%x\n", &p);
printf("0x%x\n", p);
printf("0x%x\n", *p);
```

What to be printed on the screen

```
0xbeefbeef
0x10001000
0xbeefbeef
0x1000
```

Assumption
- x will be stored at 0xBEEFBEEF
- p will be stored at 0x10001000



Bit-Level Operations

- & (AND), | (OR), ~ (NOT), ^ (Exclusive OR)
  - Apply to any “integral” data type such as long, int, short, char, etc.
  - View arguments as bit vectors
  - Operations applied bit-wise
- Examples (char data type)
  - ~0x41 & 0xBE
    - $\sim 01000001_2 \;\&\; 10111110_2 = 10111110_2 \;\&\; 10111110_2 = 10111110_2$
  - ~0xF0 | 0xFF
    - $\sim 11110000_2 \mid 11111111_2 = 00001111_2 \mid 11111111_2 = 11111111_2$
- Self XOR (e.g., x ^ x)
  - Resetting (always zero)

Logical Operations

- && (AND), || (OR), ! (NOT)
  - View 0 as “False” and anything nonzero as “True”
  - Always return 0 or 1
  - Early termination (short-circuit evaluation)
- Examples (`char` data type)
  - !0x41 && 0x00 = 0x00 && 0x00 = 0x00 (False)
  - !0x00 || 0x01 = 0x01 || 0x01 = 0x01 (True)
  - !!0x41 && 0x01 = 0x01 && 0x01 = 0x01 (True)

Shift Operations

- Left Shift: x << y
  - Shift bit-vector x left y positions
    - Throw away y MSB
    - Fill the right end with y zeros
  - Same impact as multiply by 2^Y
- Right Shift: x >> y
  - Shift bit-vector x right y positions
    - Throw away y LSB
  - Logical shift
    - Fill the left end with y zeros
  - Arithmetic shift
    - Replicate MSB on the left end
  - Same impact as divide by 2^Y

x = 01100010	x << 3	00010000
	x >> 2 (logical)	00011000
	x >> 2 (arithmetic)	00011000
y = 10100010	y << 3	00010000
	y >> 2 (logical)	00101000
	y >> 2 (arithmetic)	11101000

Practice

Consider a memory mapped I/O system, and you want to turn on an I/O device whose 32-bit control register is mapped to a variable "control" and power control is specifically bit #5 (1: on, 0: off). Write a code that turns on the I/O device (i.e., setting bit #5 to 1) without disturbing other bits.

```
control = control | 0x00000020;
```

Memory mapped I/O system:
https://en.wikipedia.org/wiki/Memory-mapped_I/O_and_port-mapped_I/O

Integer Representation

Integer Representation

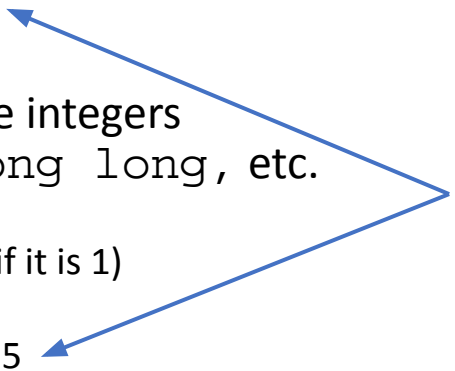
*** Java supports only signed integers

- Unsigned integer
  - Represents zero and positive integers
  - `unsigned char, unsigned short, unsigned int, unsigned long, unsigned long long`, etc.
  - Unsigned encoding standard
  - Example (8-bit)

    $10000011 = 2^7 + 2^1 + 2^0 = 128 + 2 + 1 = 131$
- Signed integer
  - Represents negative, zero, and positive integers
  - `char, short, int, long, long long`, etc.
  - Two's complement standard
    - MSB is a sign bit (big negative value (-2^n) if it is 1)
  - Example (8-bit)

    $10000011 = -2^7 + 2^1 + 2^0 = -128 + 2 + 1 = -125$

    $00000011 = 2^1 + 2^0 = 2 + 1 = 3$



Conversion

- Decimal to Binary
  - 128 (8-bit)
    - Keep dividing by two and read the remainders from the bottom
  - -128 (8-bit)
    - Step 1: Find the bit pattern of the absolute value
    - Step 2: Flip all bits
    - Step 3: Add 1

Min & Max

- Unsigned Integers (w-bit)
  - *Min* = 0

    000…0
  - *Max* = $2^w - 1$

    111…1

- Signed Integers (w-bit)
  - *Min* = -2^{w-1}

    100…0
  - *Max* = $2^{w-1} - 1$

    011…1
  - -1

    111…1

Example: 16-bit integer (*w* = 16)

	Binary	Hex	Decimal
Unsigned Max	111.....1_2	0xFFFF	2^{16} - 1
Signed Max	011.....1_2	0x7FFF	2^{15} - 1
Signed Min	100.....0_2	0x8000	-2^{15}
-1	111.....1_2	0xFFFF	-1
0	000.....0_2	0x0000	0

Range

- The ranges for signed are not symmetric
  - The range of negative integers extends one further

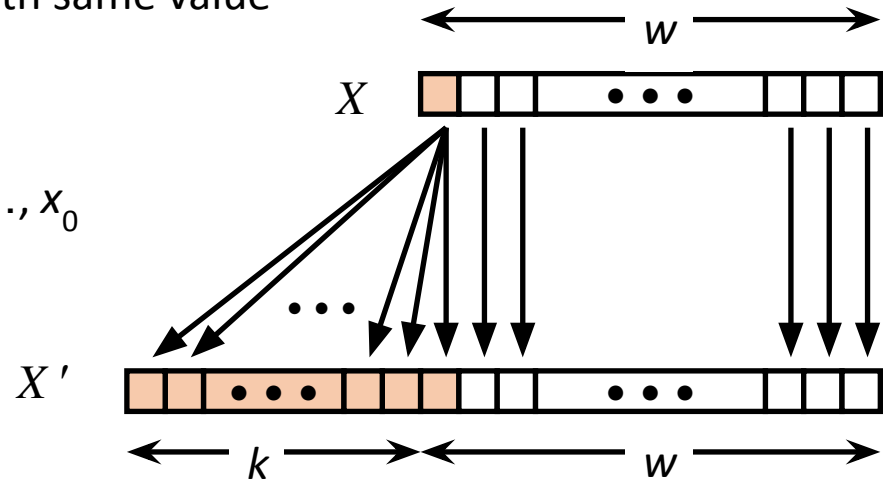
C data type	Minimum	Maximum
`char`	-128	127
`unsigned char`	0	255
`short`	-32,768	32,767
`unsigned short`	0	65,535
`int`	-2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295

Sign Extension

Example

```
short tmp1 = -1;
int tmp2 = (int)tmp1;
```

- Signed integers use sign extension
- Task:
  - Given *w*-bit signed integer *x*
  - Convert it to *w*+*k*-bit integer with same value
- Rule:
  - Make *k* copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Zero Extension

Example

```
unsigned short tmp1 = 0xffff;
unsigned int tmp2 = (unsigned int)tmp1;
```

- Unsigned integers use zero extension
- Task:
  - Given *w*-bit unsigned integer *x*
  - Convert it to *w*+*k*-bit integer with same value
- Rule:
  - Make *k* copies of zero:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of zero}}, x_{w-1}, x_{w-2}, \ldots, x_0$

```
short x =  15213;
int  ix = (int)x;
short 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