

Course Overview

Computer Architecture

Instructor:

Dr. R. Shathanaa

Overview

- Course theme
- Five realities
- How the course fits into the curriculum

Poll

■ What is your career choice?

Course Theme: Abstraction Is Good But Don't Forget Reality

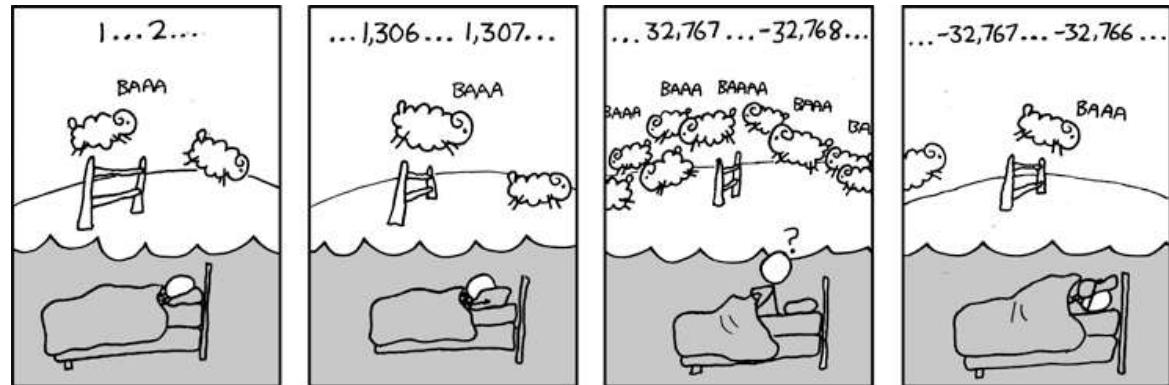
- Most CS courses emphasize abstraction
 - Abstract data types
 - Asymptotic analysis
- These abstractions have limits
 - Especially in the presence of bugs
 - Need to understand details of underlying implementations
- Useful outcomes
 - Become more effective programmers
 - Able to find and eliminate bugs efficiently
 - Able to understand and tune for program performance
 - Prepare for later “systems” classes in CS & ECE
 - Compilers, Operating Systems, Networks, Advanced Computer Architecture, Embedded Systems, Storage Systems, etc.

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:

- $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
- $1e20 + (-1e20 + 3.14) \rightarrow ??$

Computer Arithmetic

■ Does not generate random values

- Arithmetic operations have 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

Poll

■ How confident are you in programming?

Great Reality #2:

You've Got to 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 software
 - 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 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

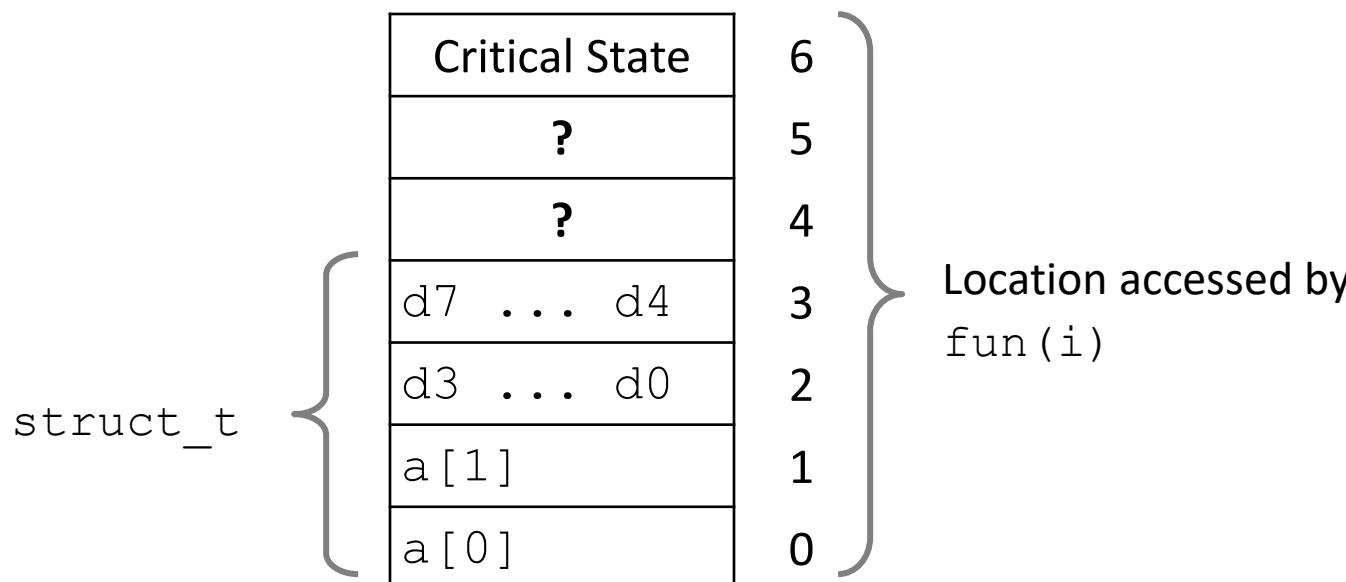
- Result is system specific

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 Java, Ruby, Python, 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 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];  
}
```

```
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];  
}
```

4.3ms 2.0 GHz Intel Core i7 Haswell 81.8ms

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

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

Now, why take this course?

- End of Moore's law
- Multiprocessor architectures
- Emergence of new platforms

- To be better, a hardware designer, compiler designer, OS designer AND a software developer need to know about the basic hardware.

Textbooks

■ Randal E. Bryant and David R. O'Hallaron,

- *Computer Systems: A Programmer's Perspective*, **Third Edition** (CS:APP3e), Pearson, 2016
- <http://csapp.cs.cmu.edu>
- This book really matters for the course!
 - How to solve labs
 - Practice problems typical of exam problems

■ Brian Kernighan and Dennis Ritchie,

- *The C Programming Language*, Second Edition, Prentice Hall, 1988
- Still the best book about C, from the originators

*Welcome
and Enjoy!*

A Tour of Systems

C Hello Program

code/intro/hello.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("hello, world\n");
6     return 0;
7 }
```

code/intro/hello.c

Information Is Bits + Context

The representation of `hello.c` illustrates a fundamental idea: All information in a system—including disk files, programs stored in memory, user data stored in memory, and data transferred across a network—is represented as a bunch of bits.

The only thing that distinguishes different data objects is the context in which we view them.

For example, in different contexts, the same sequence of bytes might represent an integer, floating-point number, character string, or machine instruction.

#	i	n	c	l	u	d	e	SP	<	s	t	d	i	o	.
35	105	110	99	108	117	100	101	32	60	115	116	100	105	111	46
h	>	\n	\n	i	n	t	SP	m	a	i	n	()	\n	{
104	62	10	10	105	110	116	32	109	97	105	110	40	41	10	123
\n	SP	SP	SP	p	r	i	n	t	f	("	h	e	l	
10	32	32	32	32	112	114	105	110	116	102	40	34	104	101	108
l	o	,	SP	w	o	r	l	d	\	n	")	;	\n	SP
108	111	44	32	119	111	114	108	100	92	110	34	41	59	10	32
SP	SP	SP	r	e	t	u	r	n	SP	0	;	\n	}	\n	
32	32	32	114	101	116	117	114	110	32	48	59	10	125	10	

Figure 1.2 The ASCII text representation of `hello.c`.

Programs Are Translated by Other Programs into Different Forms

linux> *gcc -o hello hello.c*

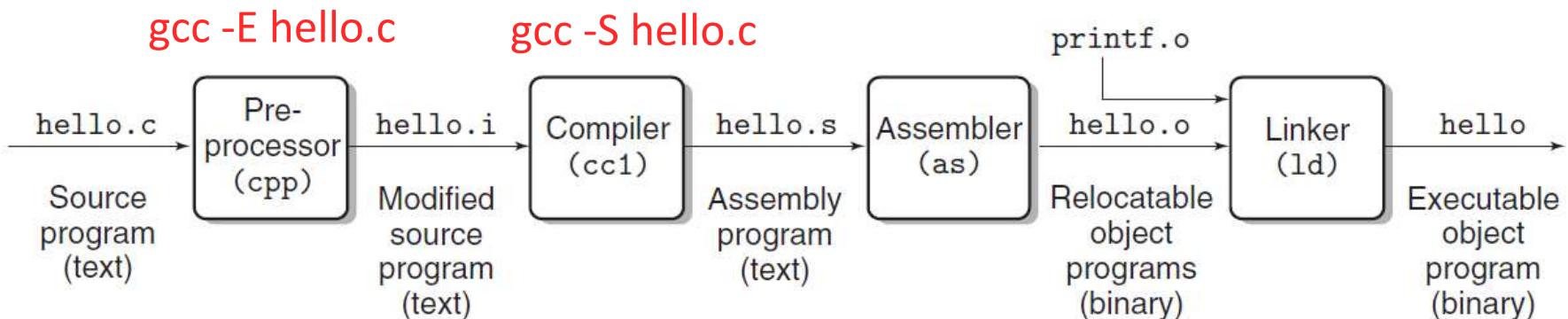


Figure 1.3 The compilation system.

gcc –o hello.o hello.c

- The programs that perform the four phases (*preprocessor*, *compiler*, *assembler*, and *linker*) are known collectively as the *compilation system*.

- *Preprocessing phase.* The preprocessor (cpp) modifies the original C program according to directives that begin with the ‘#’ character. For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text. The result is another C program, typically with the `.i` suffix.
- *Compilation phase.* The compiler (cc1) translates the text file `hello.i` into the text file `hello.s`, which contains an *assembly-language program*. This program includes the following definition of function `main`:

```
1 main:  
2     subq    $8, %rsp  
3     movl    $.LC0, %edi  
4     call    puts  
5     movl    $0, %eax  
6     addq    $8, %rsp  
7     ret
```

Each of lines 2–7 in this definition describes one low-level machine-language instruction in a textual form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.

- *Assembly phase.* Next, the assembler (`as`) translates `hello.s` into machine-language instructions, packages them in a form known as a *relocatable object program*, and stores the result in the object file `hello.o`. This file is a binary file containing 17 bytes to encode the instructions for function `main`. If we were to view `hello.o` with a text editor, it would appear to be gibberish.
- *Linking phase.* Notice that our `hello` program calls the `printf` function, which is part of the *standard C library* provided by every C compiler. The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program. The linker (`ld`) handles this merging. The result is the `hello` file, which is an executable object file (or simply *executable*) that is ready to be loaded into memory and executed by the system.

It Pays to Understand How Compilation Systems Work

- *Optimizing program performance*
- *Understanding link-time errors*
- *Avoiding security holes*

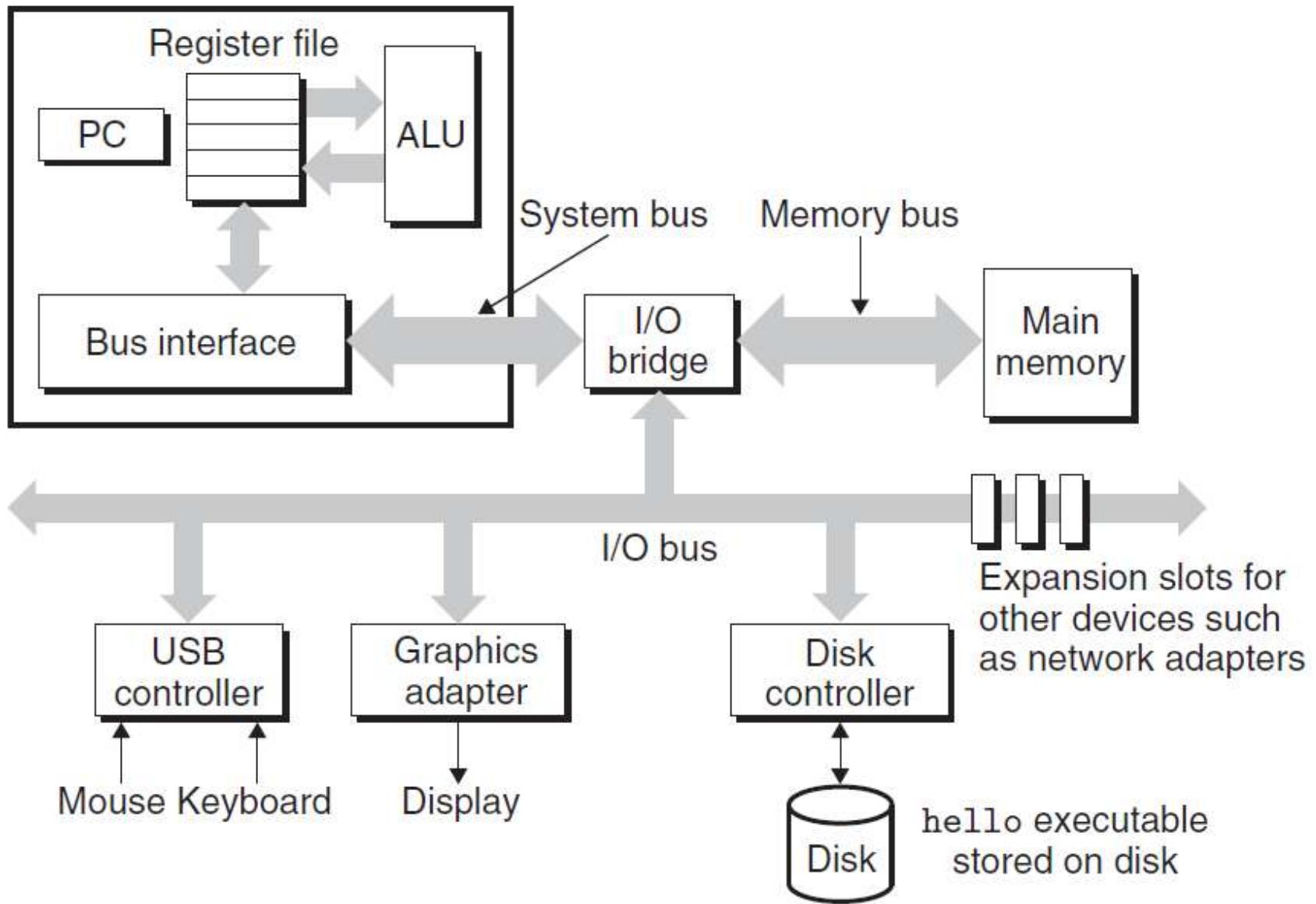
Processors Read and Interpret Instructions Stored in Memory

At this point, our `hello.c` source program has been translated by the compilation system into an executable object file called `hello` that is stored on disk. To run the executable file on a Unix system, we type its name to an application program known as a *shell*:

```
linux> ./hello
hello, world
linux>
```

The shell is a command-line interpreter that prints a prompt, waits for you to type a command line, and then performs the command. If the first word of the command line does not correspond to a built-in shell command, then the shell assumes that it is the name of an executable file that it should load and run. So in this case, the shell loads and runs the `hello` program and then waits for it to terminate. The `hello` program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.

CPU



Hardware Organization of a System

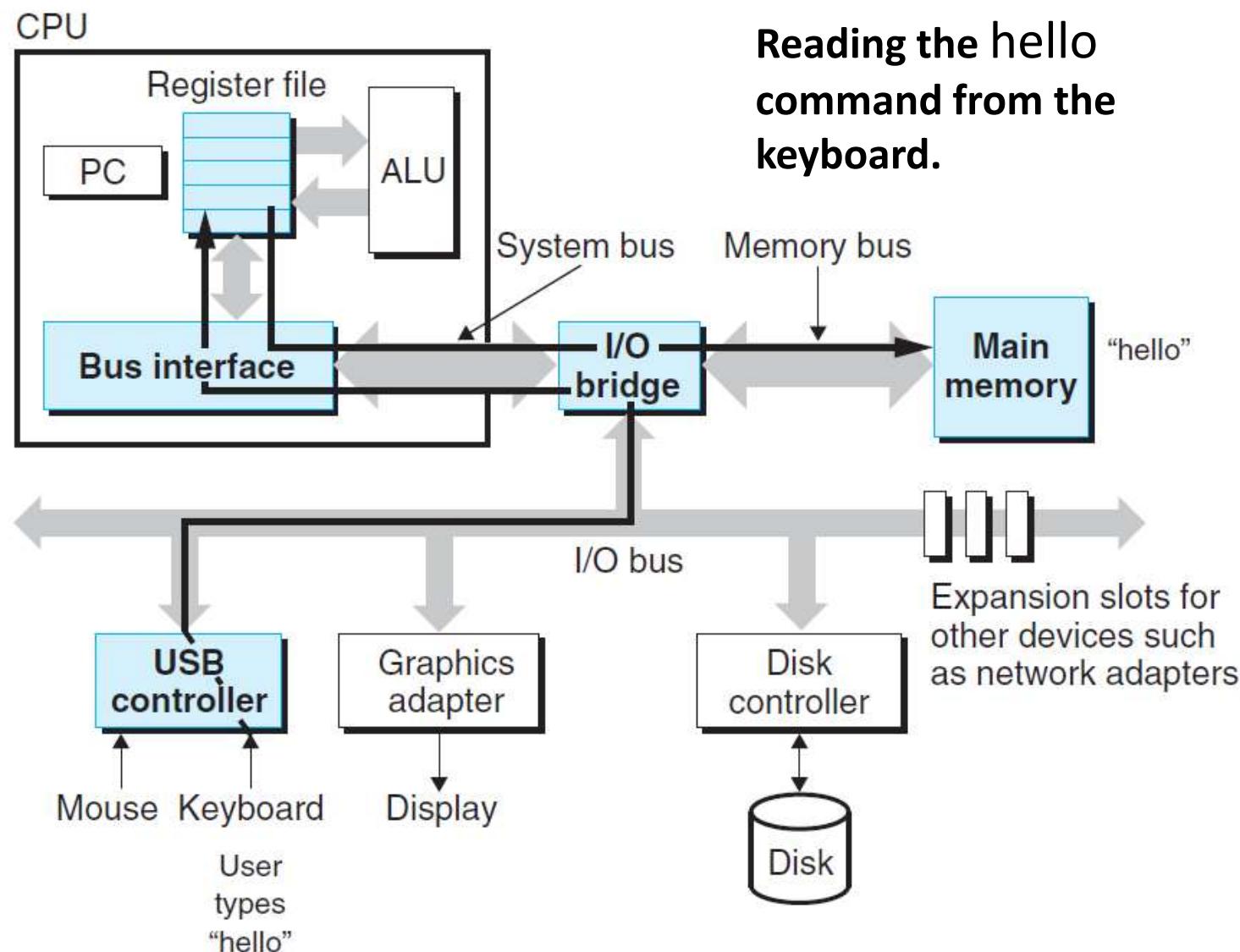
Inside your computer



Buses

Running throughout the system is a collection of electrical conduits called *buses* that carry bytes of information back and forth between the components. Buses are typically designed to transfer fixed-size chunks of bytes known as *words*. The number of bytes in a word (the *word size*) is a fundamental system parameter that varies across systems. Most machines today have word sizes of either 4 bytes (32 bits) or 8 bytes (64 bits). In this book, we do not assume any fixed definition of word size. Instead, we will specify what we mean by a “word” in any context that requires this to be defined.

Running the hello Program



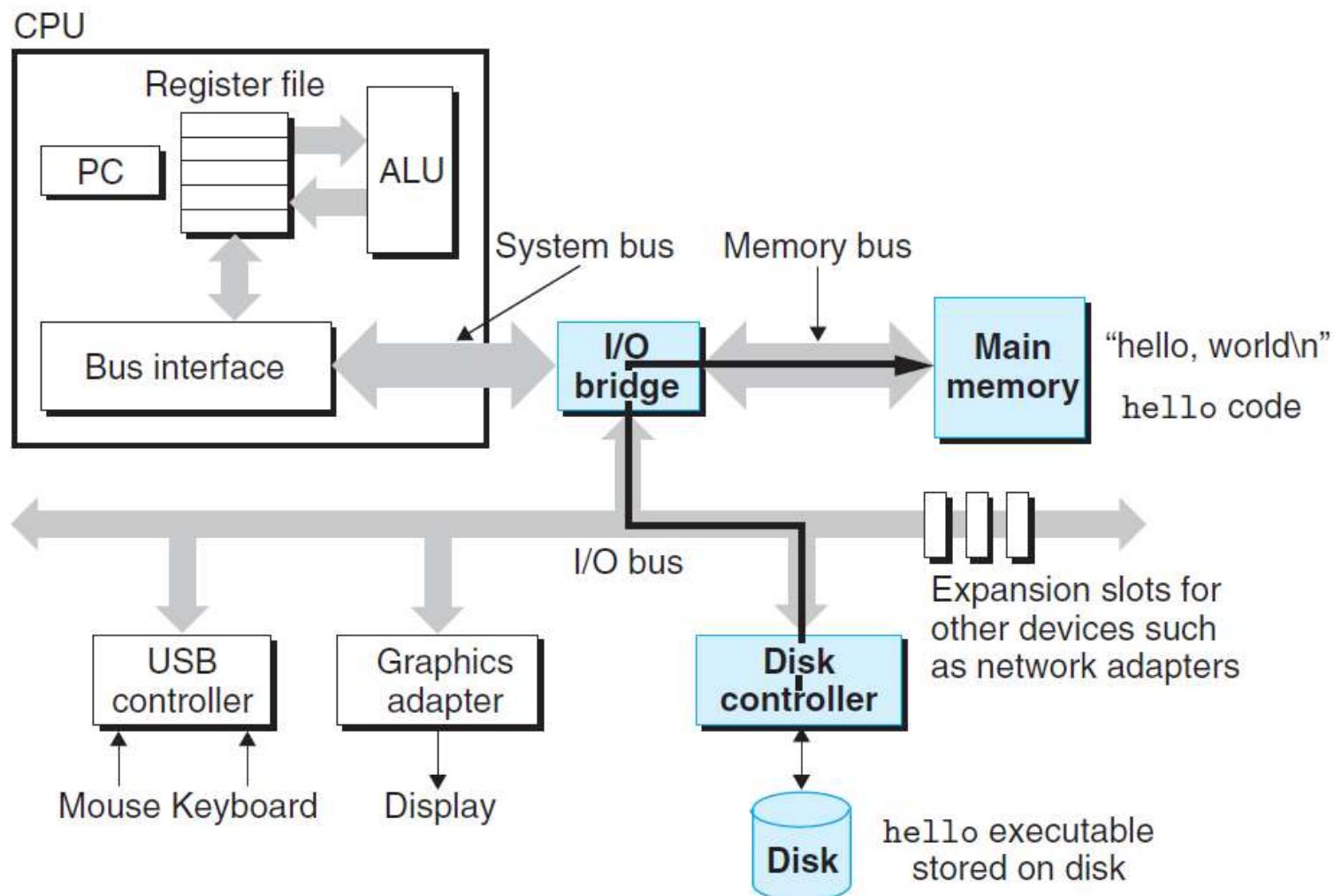


Figure 1.6 Loading the executable from disk into main memory.

CPU

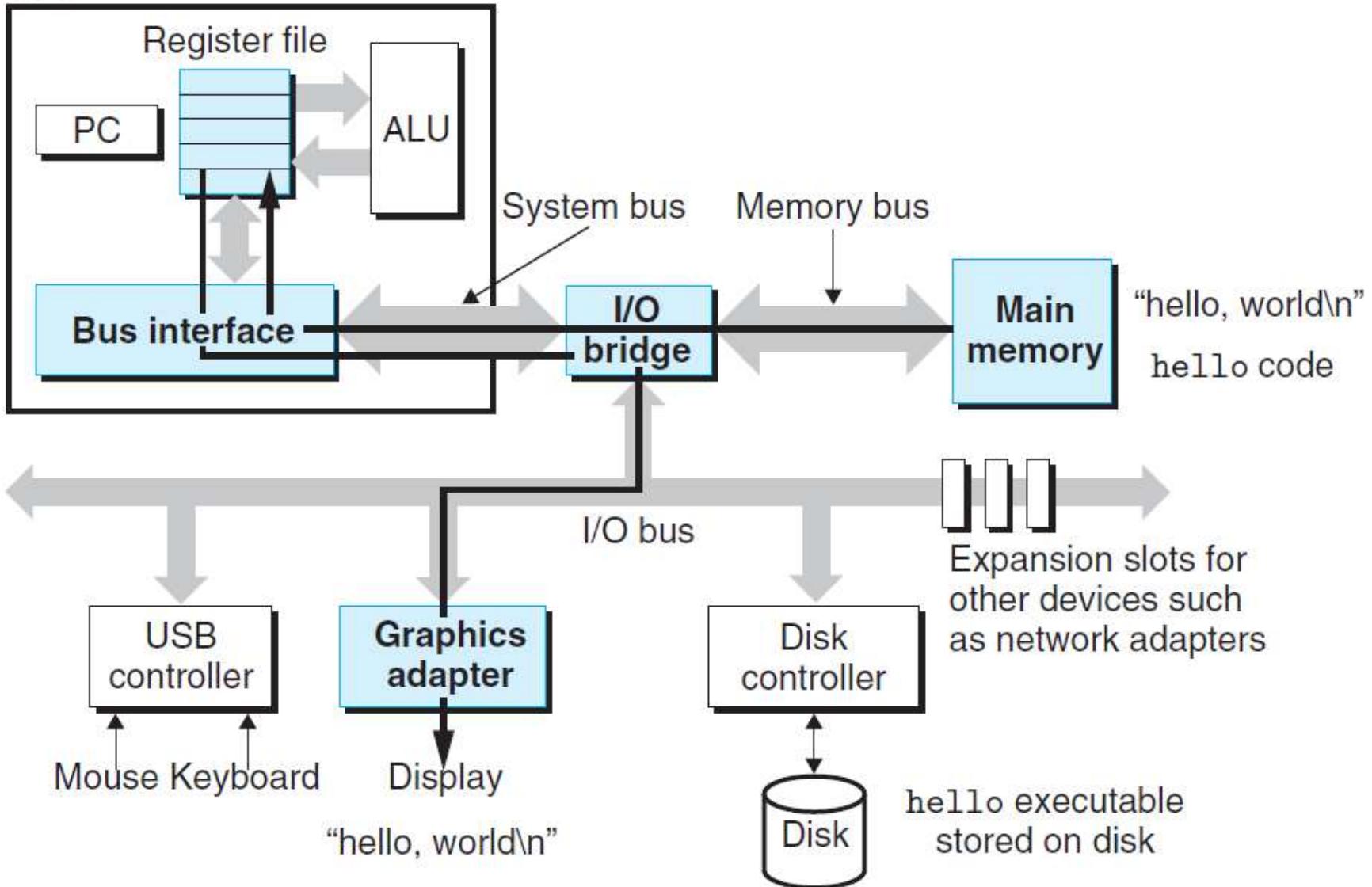


Figure 1.7 Writing the output string from memory to the display.

Processor Memory Gap

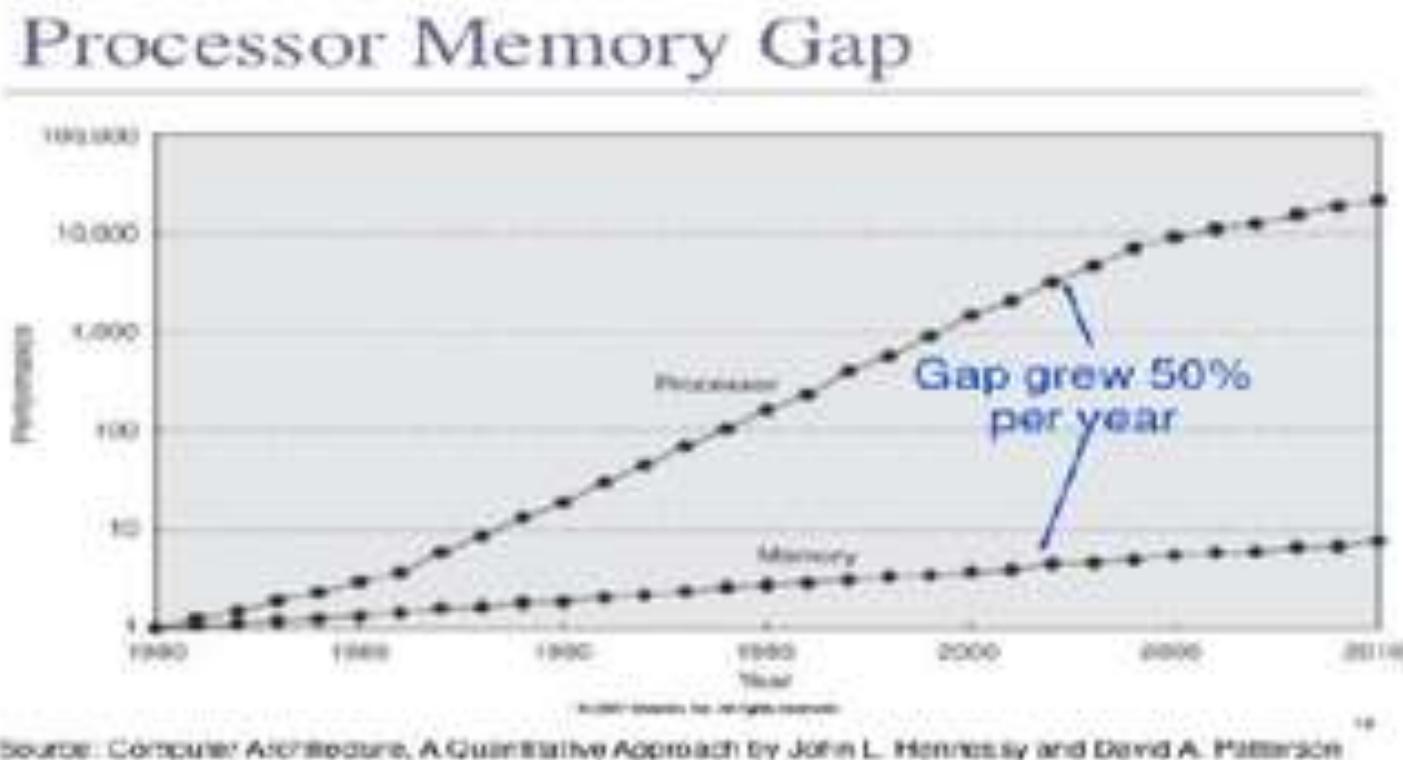
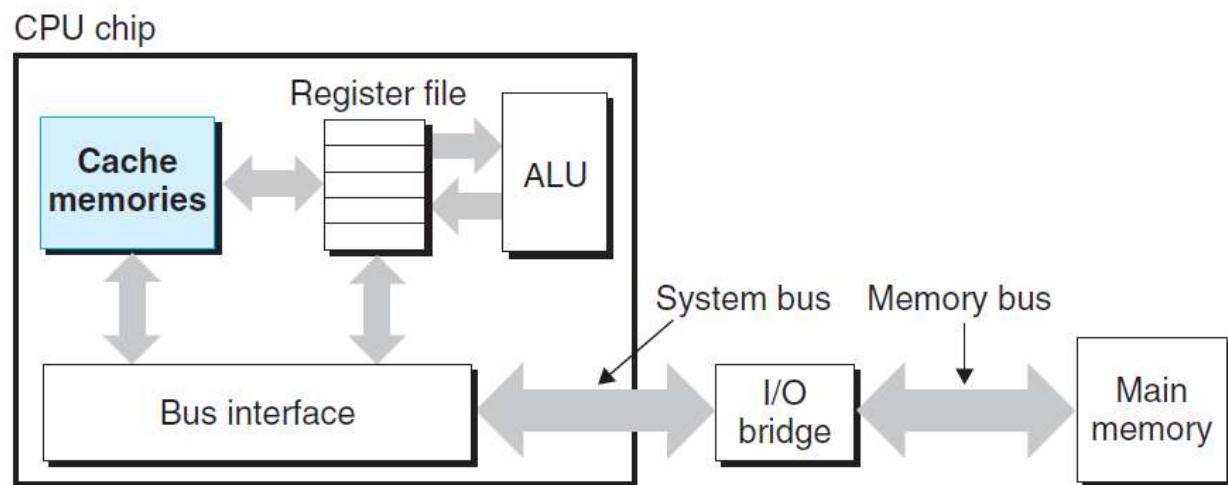


Fig. 1

Cache Matters

Figure 1.8
Cache memories.



Memory Hierarchy

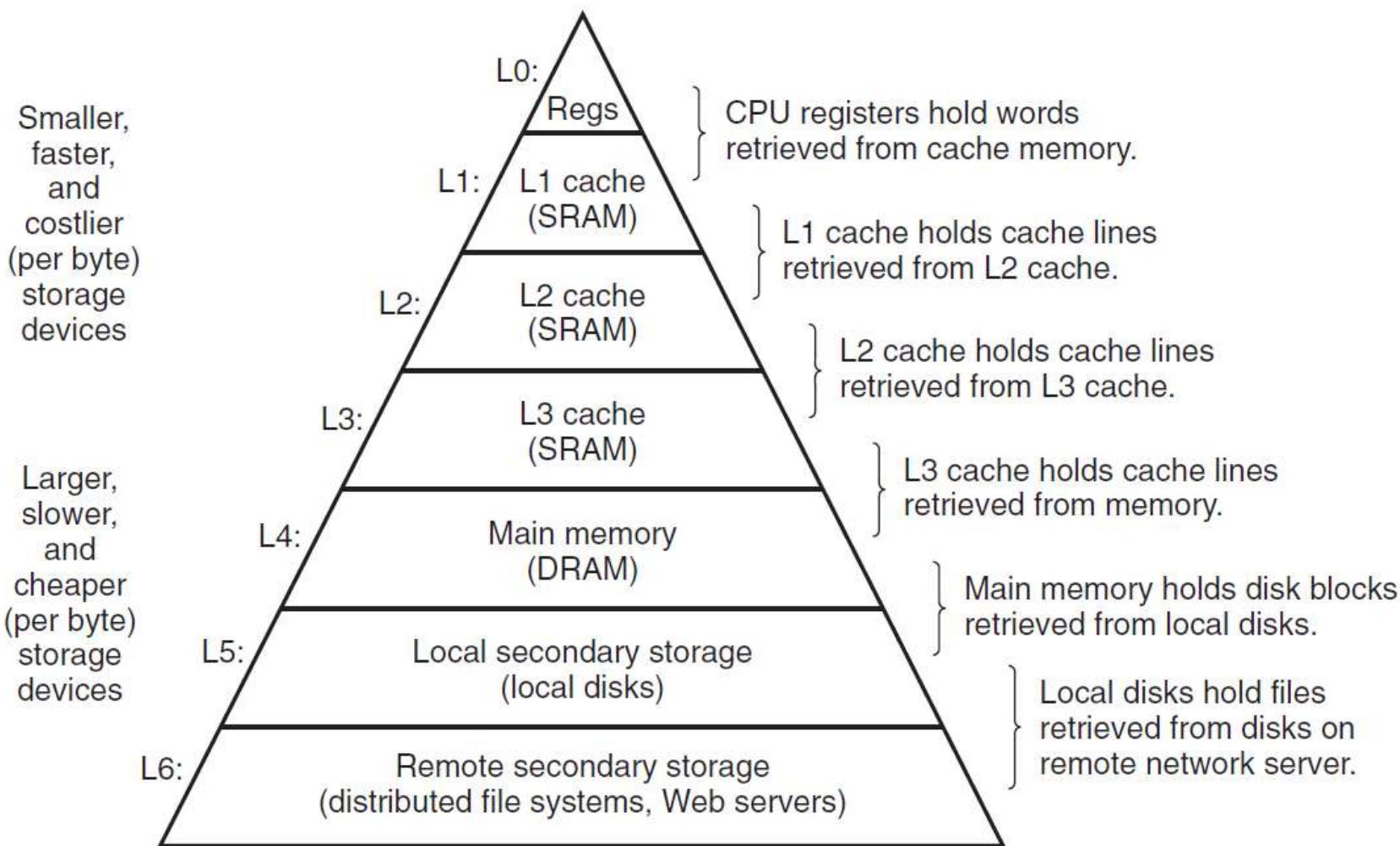


Figure 1.9 An example of a memory hierarchy.

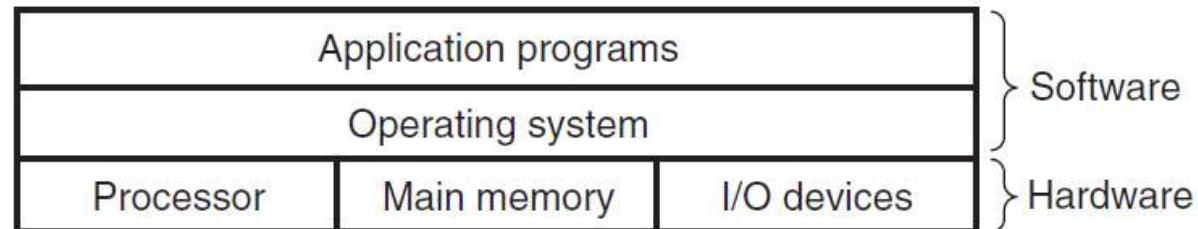
The Operating System Manages the Hardware

Back to our `Hello` example. When the shell loaded and ran the `Hello` program, and when the `Hello` program printed its message, neither program accessed the keyboard, display, disk, or main memory directly. Rather, they relied on the services provided by the *operating system*.

We can think of the operating system as a layer of software interposed between the application program and the hardware, as shown in Figure 1.10.

All attempts by an application program to manipulate the hardware must go through the operating system.

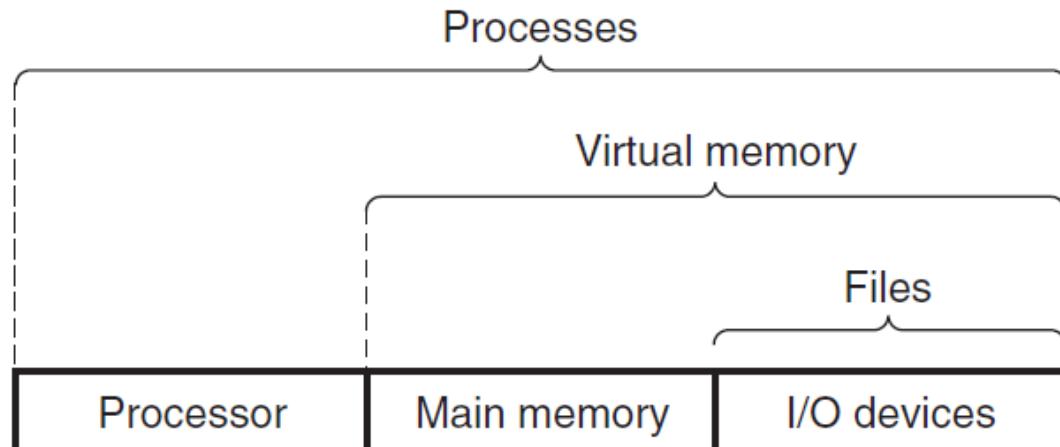
Figure 1.10
Layered view of a computer system.



The operating system has two primary purposes: (1) to protect the hardware from misuse by runaway applications and (2) to provide applications with simple and uniform mechanisms for manipulating complicated and often wildly different low-level hardware devices.

The operating system achieves both goals via the fundamental abstractions shown in Figure 1.11: *processes*, *virtual memory*, and *files*. As this figure suggests, files are abstractions for I/O devices, virtual memory is an abstraction for both the main memory and disk I/O devices, and processes are abstractions for the processor, main memory, and I/O devices.

Figure 1.11
Abstractions provided by
an operating system.



Processes

When a program such as `hello` runs on a modern system, the operating system provides the illusion that the program is the only one running on the system.

A *process* is the operating system's abstraction for a running program. Multiple processes can run concurrently on the same system, and each process appears to have exclusive use of the hardware.

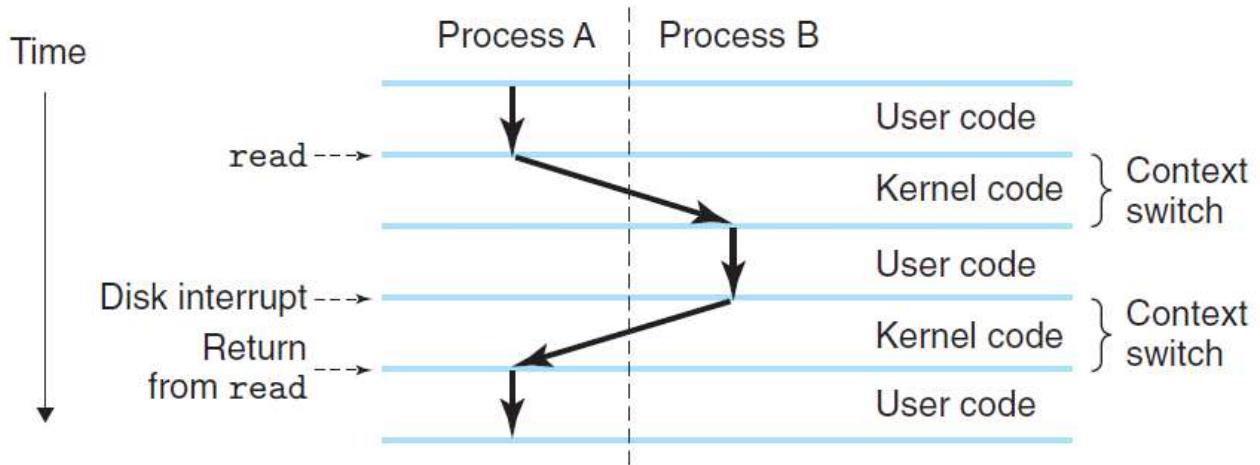
The operating system performs this interleaving with a mechanism known as *context switching*. To simplify the rest of this discussion, we consider only a *uniprocessor system* containing a single CPU

The operating system keeps track of all the state information that the process needs in order to run. This state, which is known as the *context*, includes information such as the current values of the PC, the register file, and the contents of main memory.

At any point in time, a uniprocessor system can only execute the code for a single process.

When the operating system decides to transfer control from the current process to some new process, it performs a *context switch* by saving the context of the current process, restoring the context of the new process, and then passing control to the new process. The new process picks up exactly where it left off. Figure 1.12 shows the basic idea for our example hello scenario.

Figure 1.12
Process context switching.



Threads

Although we normally think of a process as having a single control flow, in modern systems a process can actually consist of multiple execution units, called *threads*, each running in the context of the process and sharing the same code and global data.

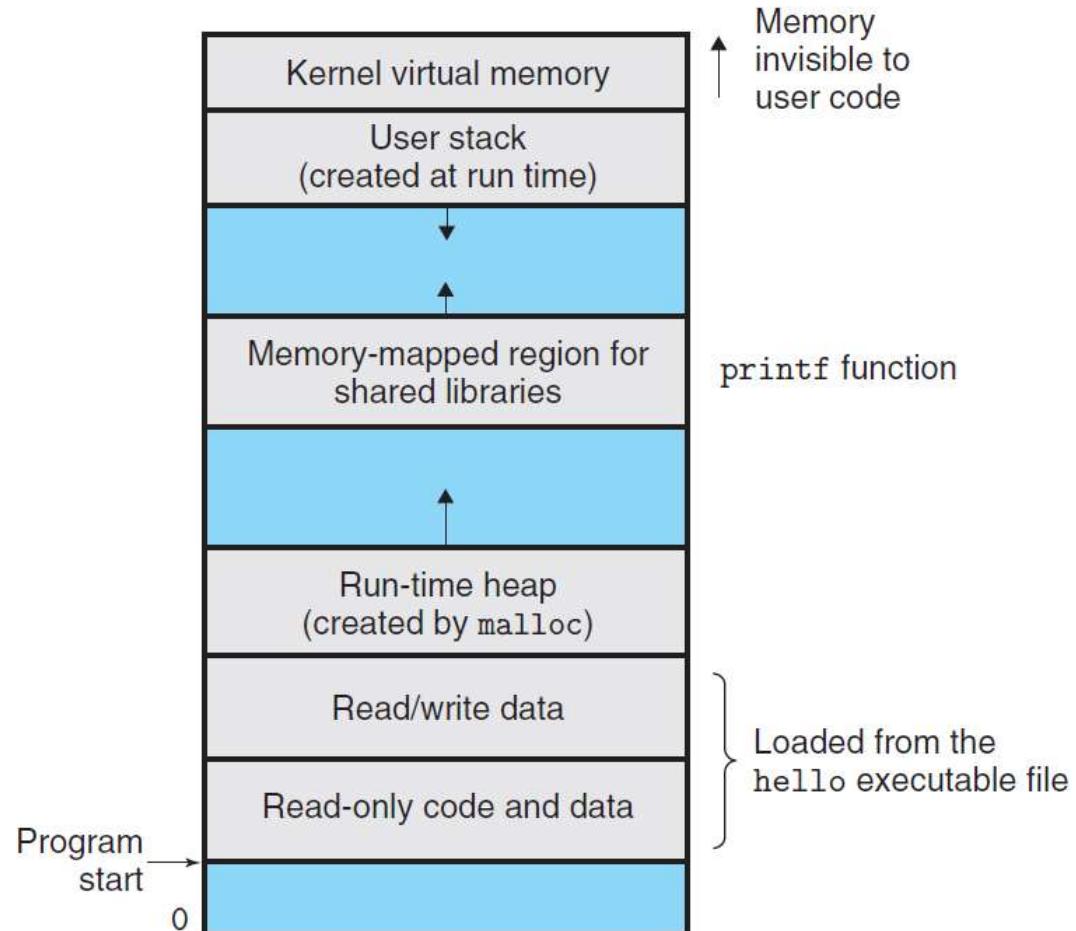
Multi-threading is also one way to make programs run faster when multiple processors are available

Virtual Memory

- *Virtual memory* is an abstraction that provides each process with the illusion that it has exclusive use of the main memory.
- Each process has the same uniform view of memory, which is known as its *virtual address space*

Figure 1.13

Process virtual address space. (The regions are not drawn to scale.)

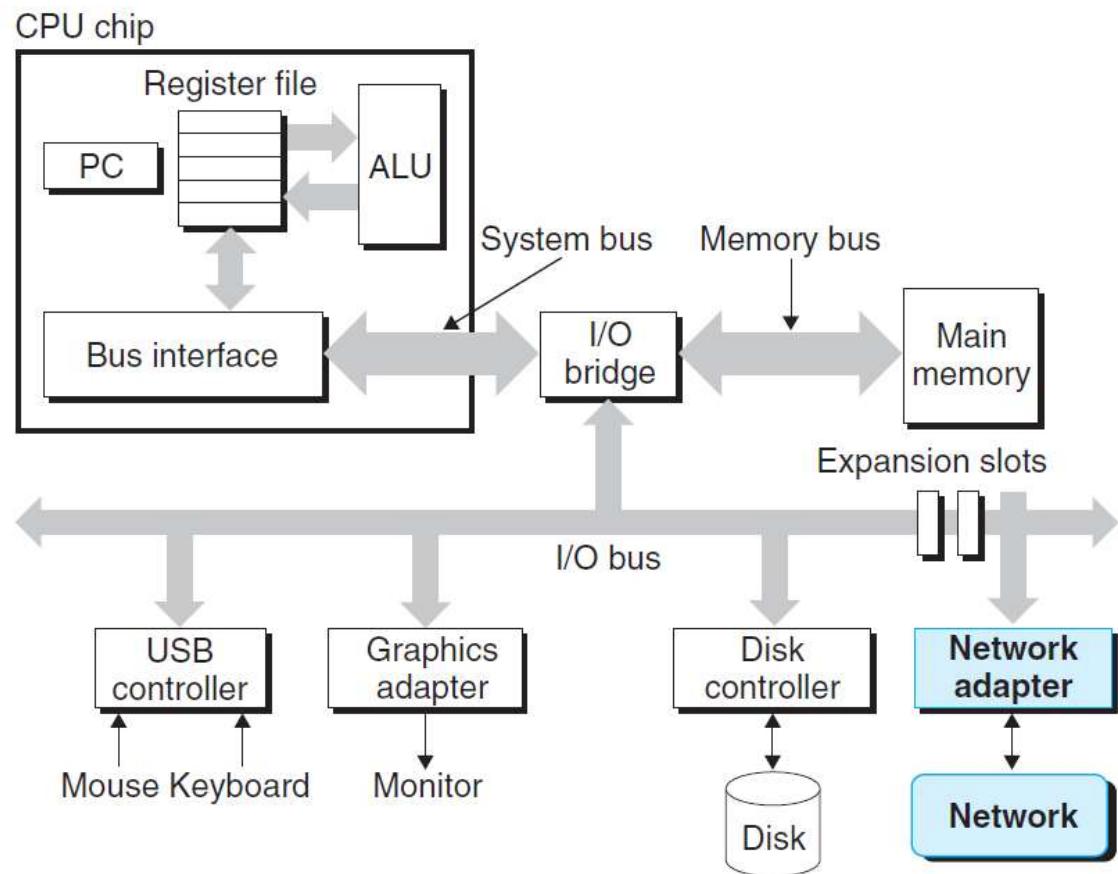


Systems Communicate with Other Systems Using Networks

Up to this point in our tour of systems, we have treated a system as an isolated collection of hardware and software. In practice, modern systems are often linked to other systems by networks.

Figure 1.14

A network is another I/O device.



How is a CPU made?

<https://youtu.be/qm67wbB5Gml>



COMPUTER ARCHITECTURE



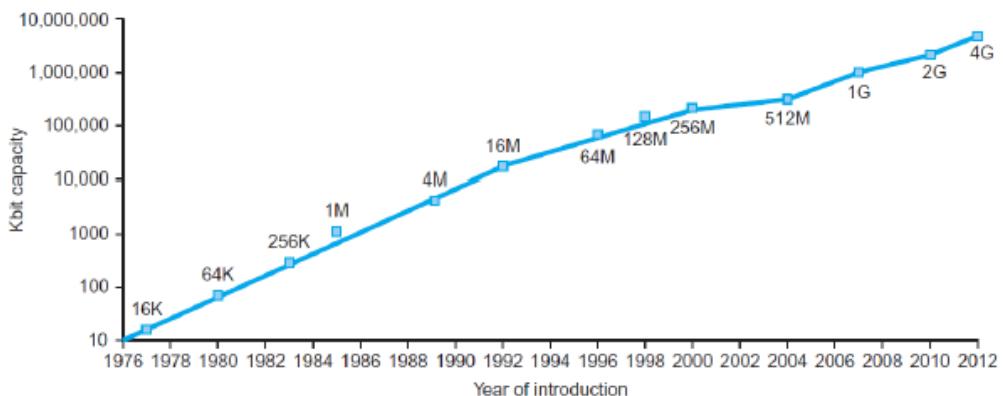
MEASURING PERFORMANCE

COMPUTER TECHNOLOGY

- Rapidly changing field:
 - vacuum tube -> transistor -> IC -> VLSI
 - doubling every 1.5 years:
 - memory capacity
 - processor speed (due to advances in technology and hardware organization)
 - cute example: if Boeing had kept up with IBM we could *fly from Bangkok to Newyork City in 10 minutes*
 - Things we'll be learning:
 - how computers work, what's a good design, what's not
 - issues affecting modern processors (e.g., caches, pipelines)

- Electronics technology continues to evolve

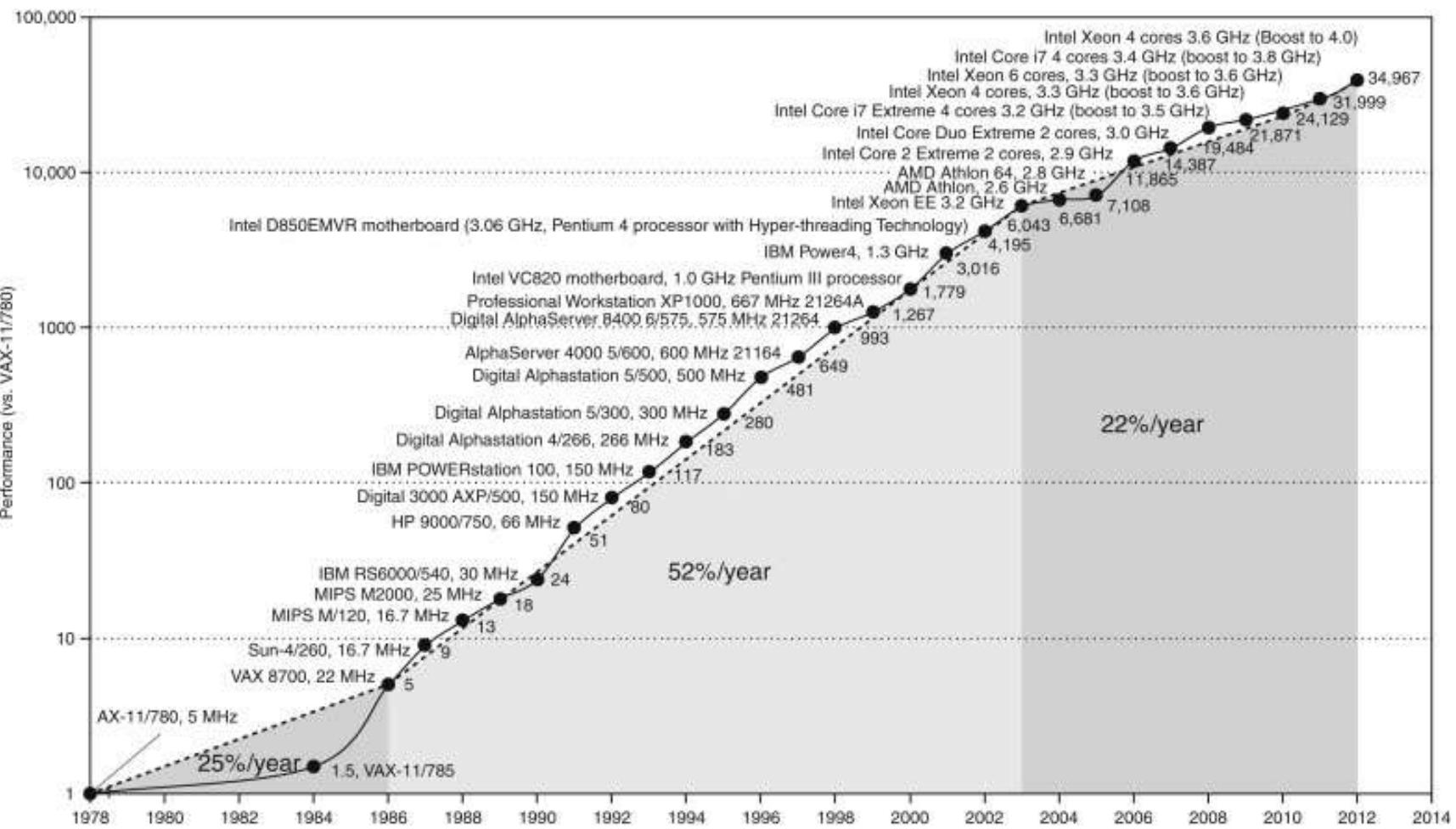
- Increased capacity and performance
- Reduced cost



DRAM capacity

Year	Technology	Relative performance/cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit (IC)	900
1995	Very large scale IC (VLSI)	2,400,000
2013	Ultra large scale IC	250,000,000,000

POWER WALL



MICROPROCESSOR PERFORMANCE

- Running out of ideas to improve single thread performance
- Power wall makes it harder to add complex features
- Power wall makes it harder to increase frequency
- Additional performance provided by: more cores, occasional spikes in frequency, accelerators

PERFORMANCE

- Performance is the key to understanding underlying motivation for the hardware and its organization
- Measure, report, and summarize performance to enable users to
 - make intelligent choices
 - see through the marketing hype!
- Why is some hardware better than others for different programs?
- What factors of system performance are hardware related?
(e.g., do we need a new machine, or a new operating system?)
- How does the machine's instruction set affect performance?

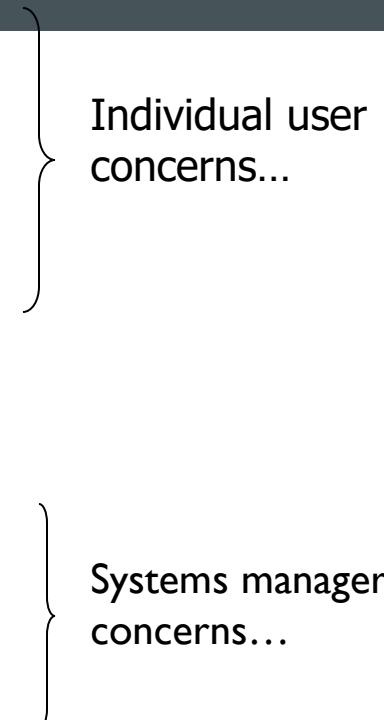
WHAT DO WE MEASURE? DEFINE PERFORMANCE....



<u>Airplane</u>	<u>Passengers</u>	<u>Range (mi)</u>	<u>Speed (mph)</u>
Boeing 737-100	101	630	598
Boeing 747	470	4150	610
BAC/Sud Concorde	132	4000	1350
Douglas DC-8-50	146	8720	544

- How much faster is the Concorde compared to the 747?
- How much bigger is the Boeing 747 than the Douglas DC-8?
- *So which of these airplanes has the best performance??*

COMPUTER PERFORMANCE: TIME, TIME, TIME!!!

- **Response Time (elapsed time, latency):**
 - how long does it take for my job to run?
 - how long does it take to execute (start to
 - finish) my job?
 - how long must I wait for the database query?
 - **Throughput:**
 - how many jobs can the machine run at once?
 - what is the average execution rate?
 - how much work is getting done?
 - *If we upgrade to a machine with a new processor what do we increase?*
 - *If we add a new machine to the lab what do we increase?*
- 

Performance

Wall Clock Time



Elapsed Time



- counts everything from start to finish
- a useful number, but often not good for comparison purposes
- **elapsed time = CPU time + wait time**

CPU Time



- I/O or time spent running other programs not counted
- $\text{CPU time} = \text{user CPU time} + \text{system CPU time}$

User CPU Time



- CPU execution time or, simply, execution time
- **Time spent executing the lines of code that are in our program**

DEFINITION OF PERFORMANCE

- For some program running on machine X:

$$\text{Performance}_X = 1 / \text{Execution time}_X$$

- *X is n times faster than Y means:*

$$\text{Performance}_X / \text{Performance}_Y = n$$

CLOCK CYCLES

- Instead of reporting execution time in seconds, we often use *cycles*. In modern computers hardware events progress cycle by cycle: in other words, each event, e.g., multiplication, addition, etc., is a sequence of cycles
- *Clock ticks* indicate start and end of cycles
- *cycle time* = time between ticks = seconds per cycle
- *clock rate (frequency)* = cycles per second
- (1 Hz. = 1 cycle/sec, 1 MHz. = 10^6 cycles/sec)
- *Example:* A 200 MHz. clock has a cycle time:

$$\frac{1}{200 \times 10^6} = 5 \times 10^{-9} \text{ seconds}$$



PERFORMANCE EQUATION I

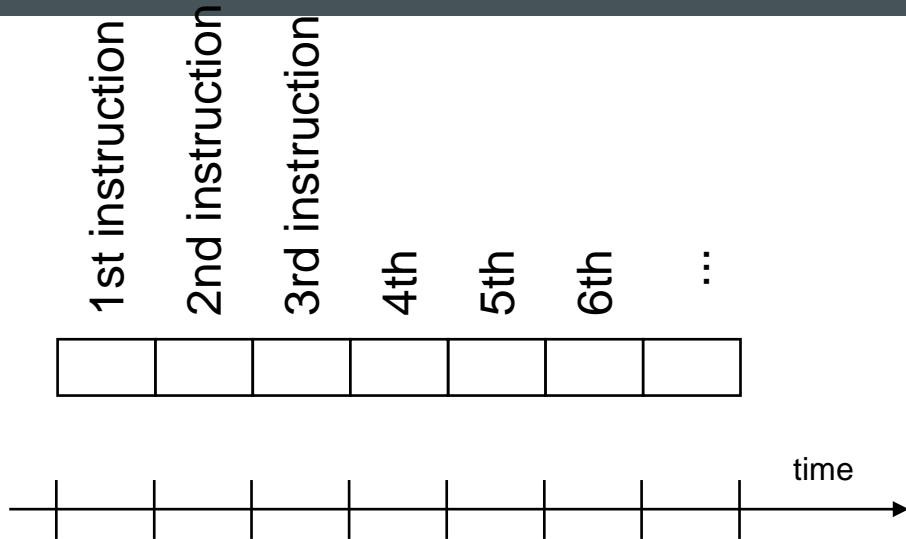
$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

equivalently

$$\begin{array}{lcl} \text{CPU execution time} & = & \text{CPU clock cycles} \\ \text{for a program} & & \times \\ & & \text{Clock cycle time} \end{array}$$

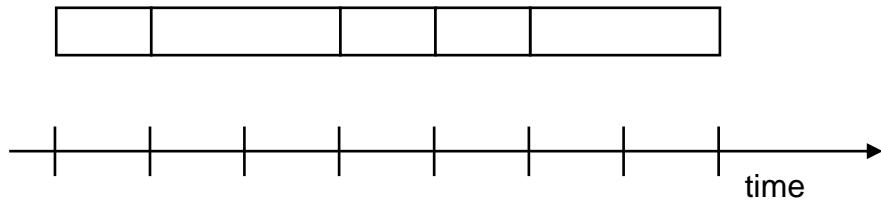
- So, to improve performance one can either:
 - reduce the number of cycles for a program, or
 - reduce the clock cycle time, or, equivalently,
 - increase the clock rate

HOW MANY CYCLES ARE REQUIRED FOR A PROGRAM?



- Could assume that # of cycles = # of instructions
 - *This assumption is incorrect!* Because:
 - Different instructions take different amounts of time (cycles)
 - Why...?

HOW MANY CYCLES ARE REQUIRED FOR A PROGRAM?



- Multiplication takes more time than addition
- Floating point operations take longer than integer ones
- Accessing memory takes more time than accessing register

EXAMPLE

- Our favorite program runs in 10 seconds on computer A, which has a 400MHz. clock. Calculate the number of clock cycles taken by the program.

EXAMPLE

- Our favorite program runs in 10 seconds on computer A, which has a 400MHz. clock. Calculate the number of clock cycles.
- We are trying to help a computer designer build a new machine B, that will run this program in 6 seconds. The designer can use new (or perhaps more expensive) technology to substantially increase the clock rate, but has informed us that this increase will affect the rest of the CPU design, causing machine B to require 1.2 times as many clock cycles as machine A for the same program.
- *What clock rate should we tell the designer to target?*

SOLUTION

Assume the program takes X cycles and the desired rate is Y

$$(X/400/10^6) / (1.2X/Y)$$

$$= 10/6$$

therefore Y = 800 MHz

TERMINOLOGY

- A given program will require:
 - some number of instructions (machine instructions)
 - some number of cycles
 - some number of seconds
- We have a vocabulary that relates these quantities:
 - *cycle time* (seconds per cycle)
 - *clock rate* (cycles per second)
 - **(average) CPI (cycles per instruction)**
 - a floating point intensive application might have a higher average CPI

PERFORMANCE EQUATION II

CPU execution time = Instruction count \times average CPI \times Clock
for a program for a program

- *Derive the above equation from Performance Equation I*
- **CPU execution time = IC * CPI * Clock cycles**
 - IC : Instruction count
 - CPI : *Clock cycles per instruction*

CPI EXAMPLE I

- Suppose we have two computers
 - Computer A has a clock cycle time of 250 ps and a CPI of 2.0 for some program
 - Computer B has a clock cycle time of 500 ps and a CPI of 1.2 for the same program.
- Which machine is faster for this program, and by how much?

SOLUTION

We know that each computer executes the same number of instructions for the program; let's call this number I . First, find the number of processor clock cycles for each computer:

$$\text{CPU clock cycles}_A = I \times 2.0$$

$$\text{CPU clock cycles}_B = I \times 1.2$$

Now we can compute the CPU time for each computer:

$$\begin{aligned}\text{CPU time}_A &= \text{CPU clock cycles}_A \times \text{Clock cycle time} \\ &= I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}\end{aligned}$$

Likewise, for B:

$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Clearly, computer A is faster. The amount faster is given by the ratio of the execution times:

$$\frac{\text{CPU performance}_A}{\text{CPU performance}_B} = \frac{\text{Execution time}_B}{\text{Execution time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

We can conclude that computer A is 1.2 times as fast as computer B for this program.

AMDAHL'S LAW

- the performance improvement to be gained from using some faster mode of execution is limited by the **fraction of the time** the faster

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

Alternatively,

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

AMDAHL'S LAW

- Performance Improvement depends on two factors:
 - The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement
 - The improvement gained by the enhanced execution mode, that is, how much faster the task would run if the enhanced mode were used for the entire program

$$= \frac{\text{Execution time after improvement}}{\frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}}$$

AMDAHL'S LAW

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

EXAMPLE

Let a program have 40 percent of its code enhanced to run 2.3 times faster. What is the overall system speedup S?

SOLUTION

Let a program have 40 percent of its code enhanced to run 2.3 times faster. What is the overall system speedup S?

$$\text{Fraction}_{\text{Enhanced}} = 0.4$$

$$\text{Speedup}_{\text{Enhanced}} = 2.3$$

$$\text{Speedup}_{\text{Overall}} = \frac{1}{(1 - 0.4) + \frac{0.4}{2.3}} = 1.292$$

COMMON CASE:AMDAHL'S LAW

- *Example:*
 - Suppose a program runs in 100 seconds on a machine, with multiplication responsible for 80 seconds of this time.
 - *How much do we have to improve the speed of multiplication if we want the program to run 5 times faster?*

AMDAHL'S LAW

$$\text{Execution time after improvement} = \frac{80 \text{ seconds}}{n} + (100 - 80 \text{ seconds})$$

Since we want the performance to be five times faster, the new execution time should be 20 seconds, giving

$$20 \text{ seconds} = \frac{80 \text{ seconds}}{n} + 20 \text{ seconds}$$

$$0 = \frac{80 \text{ seconds}}{n}$$

■ Design Principle: *Make the common case fast*

PRACTICE

- Suppose we enhance a machine making all floating-point instructions run five times faster. The execution time of some benchmark before the floating-point enhancement is 10 seconds.
 - *What will the speedup be if half of the 10 seconds is spent executing floating-point instructions?*
- We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware.
 - *How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?*

SUMMARY

- Performance specific to a particular program
 - total execution time is a consistent summary of performance
- For a given architecture performance increases come from:
 - increases in clock rate
 - improvements in processor organization that lower CPI
 - compiler enhancements that lower CPI and/or instruction count
- expecting improvement in one aspect of a machine's performance to affect the total performance

NEXT CLASS ...

- Number Representation
- **Before next class: Revise**
 - Binary Representation
 - Binary to Decimal and Decimal to Binary conversion
 - Binary Addition

Bits, Bytes, and Integers

Today: Bits, Bytes, and Integers

- **Representing information as bits**
- **Bit-level manipulations**
- **Integers**
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
- **Summary**

Number representations

- Understand the ranges of values that can be represented and the properties of the different arithmetic operations.
- This understanding is critical to writing programs that work correctly over the full range of numeric values and that are portable across different combinations of machine, operating system, and compiler
- Whereas in an earlier era program bugs would only inconvenience people when they happened to be triggered, there are now legions of hackers who try to exploit any bug they can find to obtain unauthorized access to other people's systems.
- This puts a higher level of obligation on programmers to understand how their programs work and how they can be made to behave in undesirable ways.

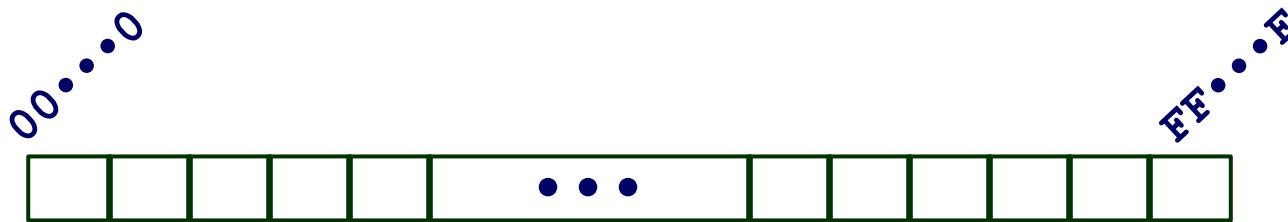
Encoding Byte Values

■ Byte = 8 bits

- Binary 0000000_2 to 1111111_2
- Decimal: 0_{10} to 255_{10}
- Hexadecimal 00_{16} to FF_{16}
 - Base 16 number representation
 - Use characters '0' to '9' and 'A' to 'F'
 - Write $FA1D37B_{16}$ in C as
 - `0xFA1D37B`
 - `0xfa1d37b`

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

Byte-Oriented Memory Organization



■ Programs Refer to Virtual Addresses

- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
- System provides address space private to particular “process”
 - Program being executed
 - Program can clobber its own data, but not that of others

■ Compiler + Run-Time System Control Allocation

- Where different program objects should be stored
- All allocation within single virtual address space

Machine Words

■ Machine Has “Word Size”

- Nominal size of integer-valued data
 - Including addresses
- Most current machines use 32 bits (4 bytes) words
 - Limits addresses to 4GB
 - Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
- Machines support multiple data formats
 - Fractions or multiples of word size
 - Always integral number of bytes

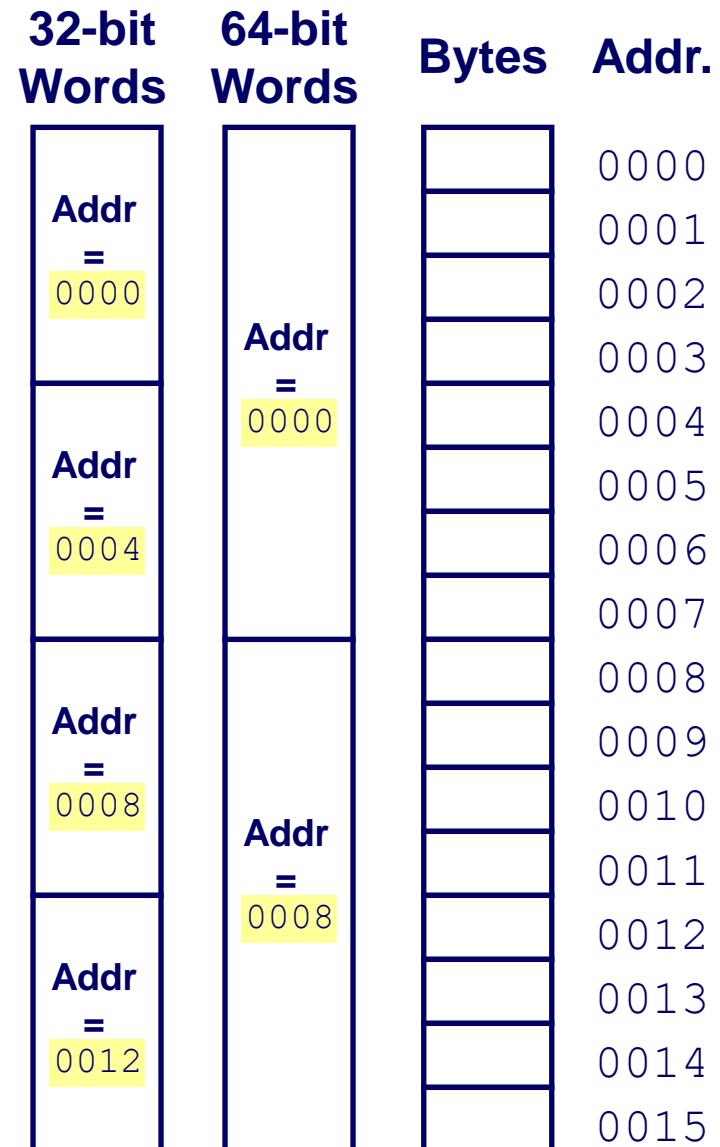
Address accessible and Word size

- 32 bit word size
- Address range: $0-2^{32}-1$
- 4 GB RAM
- Check your PCs specifications

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)



In recent years, there has been a widespread shift from machines with 32-bit word sizes to those with word sizes of 64 bits.

Most 64-bit machines can also run programs compiled for use on 32-bit machines, a form of backward compatibility. So, for example, when a program `prog.c` is compiled with the directive

`linux> gcc -m32 prog.c`

then this program will run correctly on either a 32-bit or a 64-bit machine. On the other hand, a program compiled with the directive

`linux> gcc -m64 prog.c`

will only run on a 64-bit machine. We will therefore refer to programs as being either “32-bit programs” or “64-bit programs,” since the distinction lies in how a program is compiled, rather than the type of machine on which it runs.

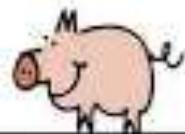
Data Representations

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

Byte Ordering

- How should bytes within a multi-byte word be ordered in memory?
- Example: suppose a variable `x` of type `int` has address `0x100`, that is, the value of the address expression `&x` is `0x100`. Then the 4 bytes of `x` would be stored in memory locations `0x100`, `0x101`, `0x102`, and `0x103`.
- Conventions
 - Big Endian: Sun, PPC Mac, Internet
 - most significant byte comes first
 - Little Endian: x86
 - least significant byte comes first

SIMPLY EXPLAINED



BIG-ENDIAN



LITTLE-ENDIAN

oxCAFEBABE
will be stored as
CA | FE | BA | BE

oxCAFEBABE
will be stored as
BE | BA | FE | CA

the terms “little endian” and “big endian” come from the book *Gulliver’s Travels* by Jonathan Swift

Byte Ordering Example

■ BigEndian

- Least significant byte has highest address

■ LittleEndian

- Least significant byte has lowest address

■ Example

- Variable x has 4-byte representation 0x01234567
- Address given by &x is 0x100

BigEndian

0x100 0x101 0x102 0x103



LittleEndian

0x100 0x101 0x102 0x103



At times, byte ordering becomes an issue. The first is when binary data are communicated over a network between different machines.

A common problem is for data produced by a little-endian machine to be sent to a big-endian machine, or vice versa, leading to the bytes within the words being in reverse order for the receiving program.

To avoid such problems, code written for networking applications must follow established conventions for byte ordering to make sure the sending machine converts its internal representation to the network standard, while the receiving machine converts the network standard to its internal representation

Reading Byte-Reversed Listings

Objdump -d test.o

■ 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:
- Pad to 32 bits:
- Split into bytes:
- Reverse:

0x12ab

0x000012ab

00 00 12 ab

ab 12 00 00

Examining Data Representations

■ Code to Print Byte Representation of Data

- Casting pointer to unsigned char * creates byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.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):

```
int a = 15213;  
0x11ffffcb8 0x6d  
0x11ffffcb9 0x3b  
0x11ffffcba 0x00  
0x11ffffcbb 0x00
```

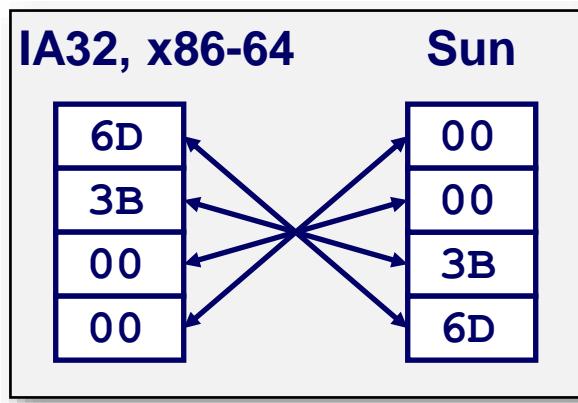
```
1 void test_show_bytes(int val) {
2     int ival = val;
3     float fval = (float) ival;
4     int *pval = &ival;
5     show_int(ival);
6     show_float(fval);
7     show_pointer(pval);
8 }
```

What do you notice?

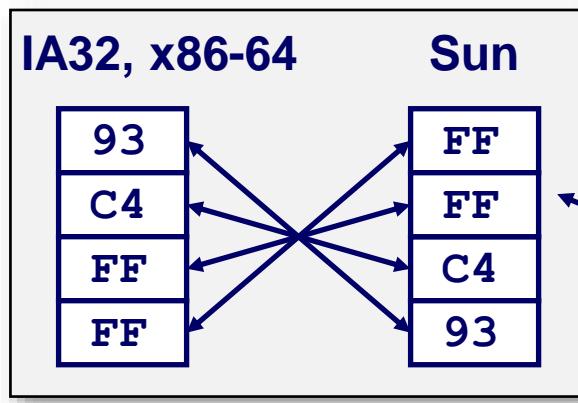
Machine	Value	Type	Bytes (hex)
Linux 32	12,345	int	39 30 00 00
Windows	12,345	int	39 30 00 00
Sun	12,345	int	00 00 30 39
Linux 64	12,345	int	39 30 00 00
Linux 32	12,345.0	float	00 e4 40 46
Windows	12,345.0	float	00 e4 40 46
Sun	12,345.0	float	46 40 e4 00
Linux 64	12,345.0	float	00 e4 40 46
Linux 32	&ival	int *	e4 f9 ff bf
Windows	&ival	int *	b4 cc 22 00
Sun	&ival	int *	ef ff fa 0c
Linux 64	&ival	int *	b8 11 e5 ff ff 7f 00 00

Representing Integers

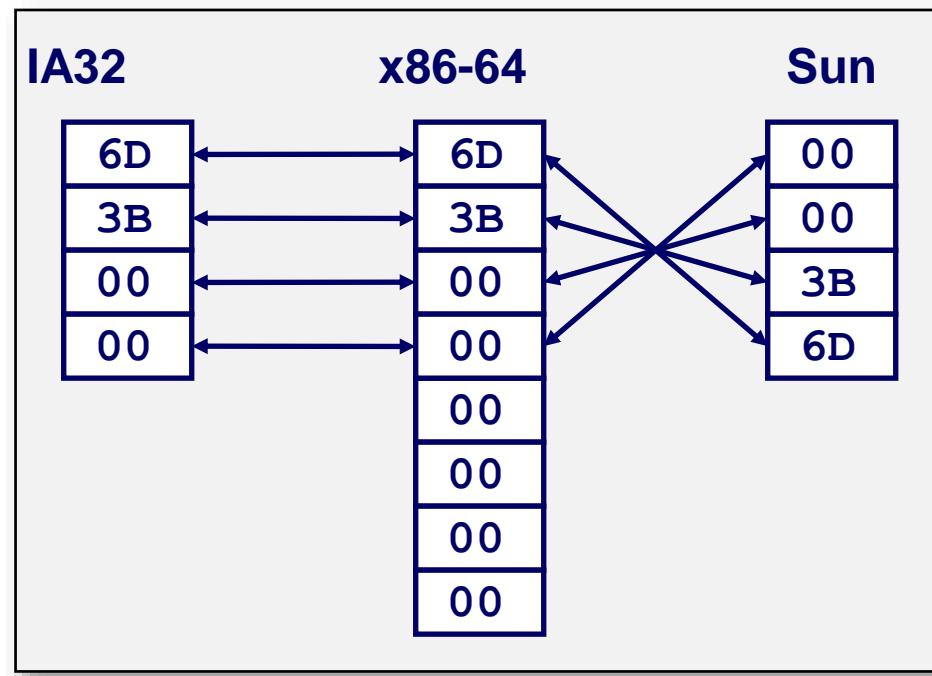
`int A = 15213;`



`int B = -15213;`



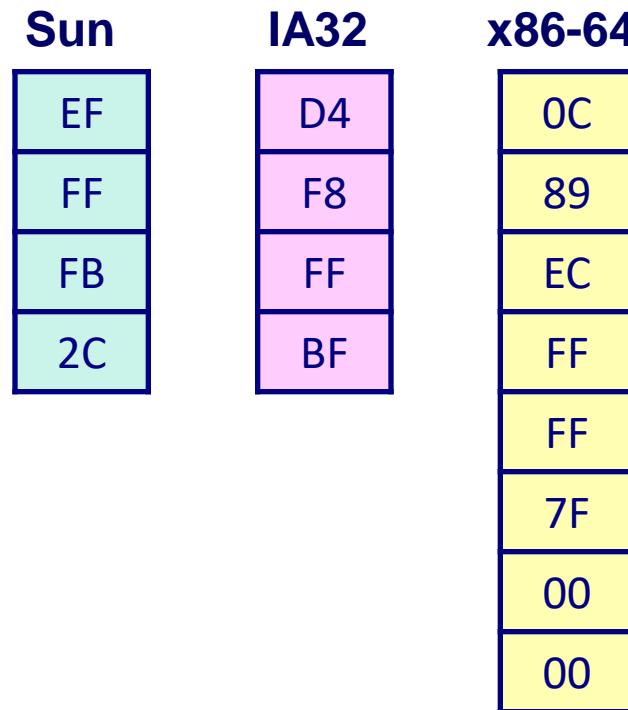
`long int C = 15213;`



Two's complement representation
(Covered later)

Representing Pointers

```
int B = -15213;  
int *P = &B;
```



Different compilers & machines assign different locations to objects

Representing Strings

```
char S[6] = "18243";
```

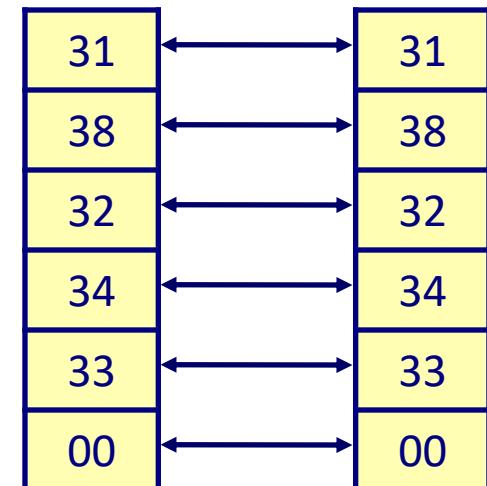
■ Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
 - 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

Linux/Alpha **Sun**



If you have a simple 8-bit character representation (e.g. extended ASCII), then no, endianness does not affect the layout, because each character is one byte.

If you have a multi-byte representation, such as UTF-16, then yes, endianness is still important

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
- Summary

Boolean Algebra

- Developed by George Boole in 19th Century, applied to logic reasoning

- Algebraic representation of logic

And ▪ Encode “True” as 1 and “False” as 0

- $A \& B = 1$ 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

- $\sim A = 1$ when $A=0$

\sim	
0	1
1	0

Exclusive-Or (Xor)

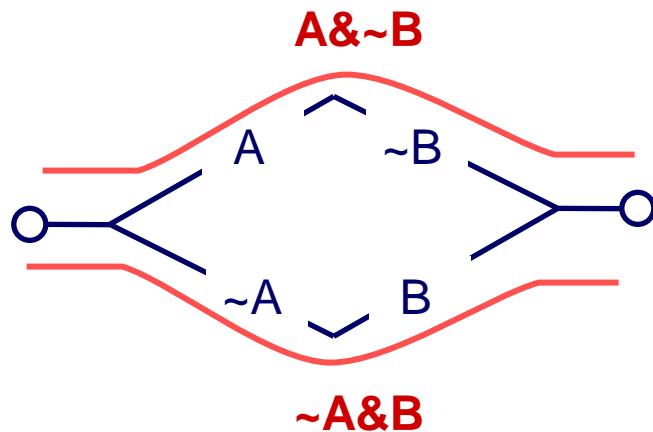
- $A ^ B = 1$ when either $A=1$ or $B=1$, but not both

$^$	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

$$A \& \sim B \mid \sim A \& B$$

$$= A^{\wedge}B$$

General Boolean Algebras

■ Operate on Bit Vectors

- Operations applied bitwise

$$\begin{array}{rcl} \begin{array}{r} 01101001 \\ \& 01010101 \end{array} & \begin{array}{r} 01101001 \\ | \quad 01010101 \end{array} & \begin{array}{r} 01101001 \\ ^\wedge \quad 01010101 \end{array} \\ \hline \begin{array}{r} 01000001 \\ 01111101 \end{array} & \begin{array}{r} 01111101 \\ 00111100 \end{array} & \begin{array}{r} 10101010 \\ 00111100 \end{array} \end{array}$$

■ All of the Properties of Boolean Algebra Apply

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)

- $\sim 0x41 = 0xBE$
 - $\sim 01000001_2 = 10111110_2$
- $\sim 0x00 = 0xFF$
 - $\sim 00000000_2 = 11111111_2$
- $0x69 \& 0x55 = 0x41$
 - $01101001_2 \& 01010101_2 = 01000001_2$
- $0x69 | 0x55 = 0x7D$
 - $01101001_2 | 01010101_2 = 01111101_2$

Contrast: Logic Operations in C

■ Contrast to Logical Operators

- `&&`, `||`, `!`
 - View 0 as “False”
 - Anything nonzero as “True”
 - Always return 0 or 1

■ Examples (char data type)

- `!0x41 = 0x00`
- `!0x00 = 0x01`
- `!!0x41 = 0x01`
- `0x69 && 0x55 = 0x01`
- `0x69 || 0x55 = 0x01`
- `p && *p` (avoids null pointer access) (**short-circuit**)

Shift Operations

■ Left Shift: $x \ll y$

- Shift bit-vector x left y positions
 - Throw away extra bits on left
 - Fill with 0's on right

■ Right Shift: $x \gg y$

- Shift bit-vector x right y positions
 - Throw away extra bits on right
- Logical shift
 - Fill with 0's on left
- Arithmetic shift
 - Replicate most significant bit on right

■ Undefined Behavior

- Shift amount < 0 or \geq word size

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

Bits, Bytes, and Integers

Today: Bits, Bytes, and Integers

- **Representing information as bits**
- **Bit-level manipulations**
- **Integers**
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
- **Summary**

Number representations

- Understand the ranges of values that can be represented and the properties of the different arithmetic operations.
- This understanding is critical to writing programs that work correctly over the full range of numeric values and that are portable across different combinations of machine, operating system, and compiler
- Whereas in an earlier era program bugs would only inconvenience people when they happened to be triggered, there are now legions of hackers who try to exploit any bug they can find to obtain unauthorized access to other people's systems.
- This puts a higher level of obligation on programmers to understand how their programs work and how they can be made to behave in undesirable ways.

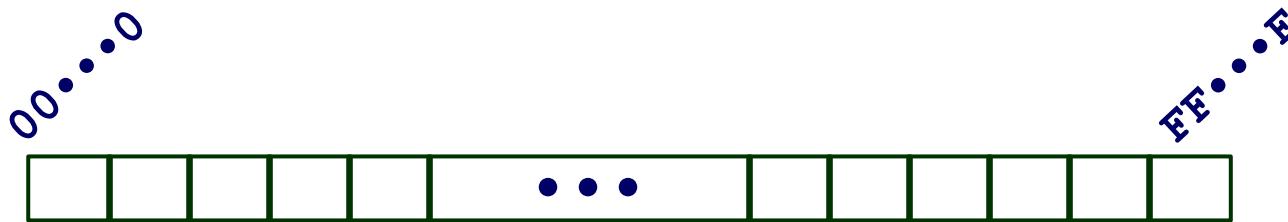
Encoding Byte Values

■ Byte = 8 bits

- Binary 0000000_2 to 1111111_2
- Decimal: 0_{10} to 255_{10}
- Hexadecimal 00_{16} to FF_{16}
 - Base 16 number representation
 - Use characters '0' to '9' and 'A' to 'F'
 - Write $FA1D37B_{16}$ in C as
 - `0xFA1D37B`
 - `0xfa1d37b`

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

Byte-Oriented Memory Organization



■ Programs Refer to Virtual Addresses

- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
- System provides address space private to particular “process”
 - Program being executed
 - Program can clobber its own data, but not that of others

■ Compiler + Run-Time System Control Allocation

- Where different program objects should be stored
- All allocation within single virtual address space

Machine Words

■ Machine Has “Word Size”

- Nominal size of integer-valued data
 - Including addresses
- Most current machines use 32 bits (4 bytes) words
 - Limits addresses to 4GB
 - Becoming too small for memory-intensive applications
- High-end systems use 64 bits (8 bytes) words
- Machines support multiple data formats
 - Fractions or multiples of word size
 - Always integral number of bytes

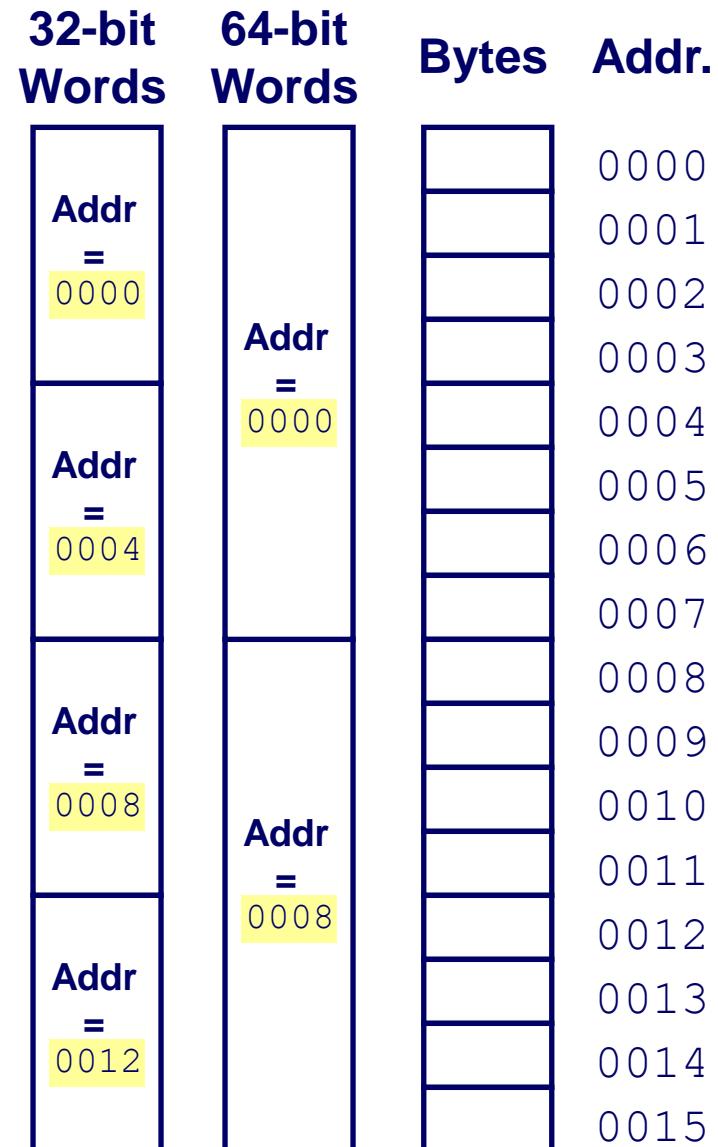
Address accessible and Word size

- 32 bit word size
- Address range: $0-2^{32}-1$
- 4 GB RAM
- Check your PCs specifications

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)



In recent years, there has been a widespread shift from machines with 32-bit word sizes to those with word sizes of 64 bits.

Most 64-bit machines can also run programs compiled for use on 32-bit machines, a form of backward compatibility. So, for example, when a program `prog.c` is compiled with the directive

`linux> gcc -m32 prog.c`

then this program will run correctly on either a 32-bit or a 64-bit machine. On the other hand, a program compiled with the directive

`linux> gcc -m64 prog.c`

will only run on a 64-bit machine. We will therefore refer to programs as being either “32-bit programs” or “64-bit programs,” since the distinction lies in how a program is compiled, rather than the type of machine on which it runs.

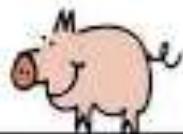
Data Representations

C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

Byte Ordering

- How should bytes within a multi-byte word be ordered in memory?
- Example: suppose a variable `x` of type `int` has address `0x100`, that is, the value of the address expression `&x` is `0x100`. Then the 4 bytes of `x` would be stored in memory locations `0x100`, `0x101`, `0x102`, and `0x103`.
- Conventions
 - Big Endian: Sun, PPC Mac, Internet
 - most significant byte comes first
 - Little Endian: x86
 - least significant byte comes first

SIMPLY EXPLAINED



BIG-ENDIAN



LITTLE-ENDIAN

oxCAFEBABE
will be stored as
CA | FE | BA | BE

oxCAFEBABE
will be stored as
BE | BA | FE | CA

the terms “little endian” and “big endian” come from the book *Gulliver’s Travels* by Jonathan Swift

Byte Ordering Example

■ BigEndian

- Least significant byte has highest address

■ LittleEndian

- Least significant byte has lowest address

■ Example

- Variable x has 4-byte representation 0x01234567
- Address given by &x is 0x100

BigEndian

0x100 0x101 0x102 0x103



LittleEndian

0x100 0x101 0x102 0x103



At times, byte ordering becomes an issue. The first is when binary data are communicated over a network between different machines.

A common problem is for data produced by a little-endian machine to be sent to a big-endian machine, or vice versa, leading to the bytes within the words being in reverse order for the receiving program.

To avoid such problems, code written for networking applications must follow established conventions for byte ordering to make sure the sending machine converts its internal representation to the network standard, while the receiving machine converts the network standard to its internal representation

Reading Byte-Reversed Listings

Objdump -d test.o

■ 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:
- Pad to 32 bits:
- Split into bytes:
- Reverse:

0x12ab

0x000012ab

00 00 12 ab

ab 12 00 00

Examining Data Representations

■ Code to Print Byte Representation of Data

- Casting pointer to unsigned char * creates byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.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):

```
int a = 15213;  
0x11ffffcb8 0x6d  
0x11ffffcb9 0x3b  
0x11ffffcba 0x00  
0x11ffffcbb 0x00
```

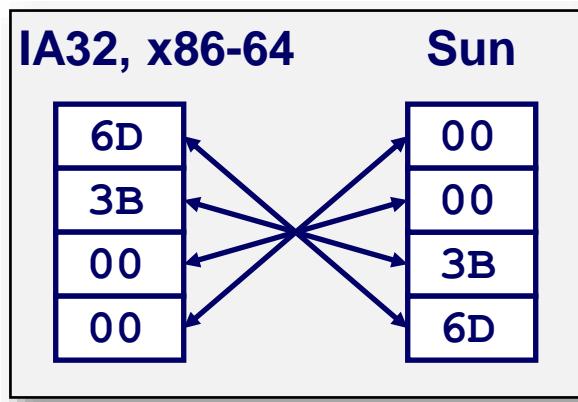
```
1 void test_show_bytes(int val) {
2     int ival = val;
3     float fval = (float) ival;
4     int *pval = &ival;
5     show_int(ival);
6     show_float(fval);
7     show_pointer(pval);
8 }
```

What do you notice?

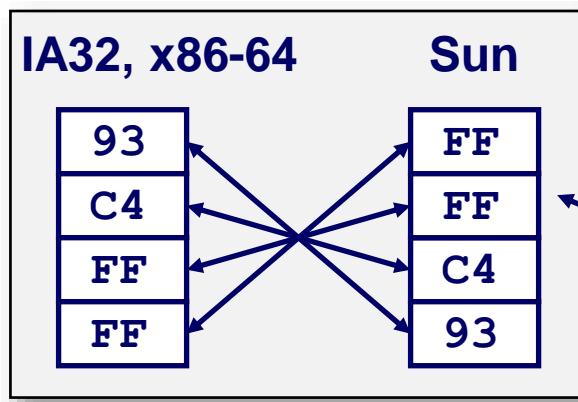
Machine	Value	Type	Bytes (hex)
Linux 32	12,345	int	39 30 00 00
Windows	12,345	int	39 30 00 00
Sun	12,345	int	00 00 30 39
Linux 64	12,345	int	39 30 00 00
Linux 32	12,345.0	float	00 e4 40 46
Windows	12,345.0	float	00 e4 40 46
Sun	12,345.0	float	46 40 e4 00
Linux 64	12,345.0	float	00 e4 40 46
Linux 32	&ival	int *	e4 f9 ff bf
Windows	&ival	int *	b4 cc 22 00
Sun	&ival	int *	ef ff fa 0c
Linux 64	&ival	int *	b8 11 e5 ff ff 7f 00 00

Representing Integers

`int A = 15213;`



`int B = -15213;`

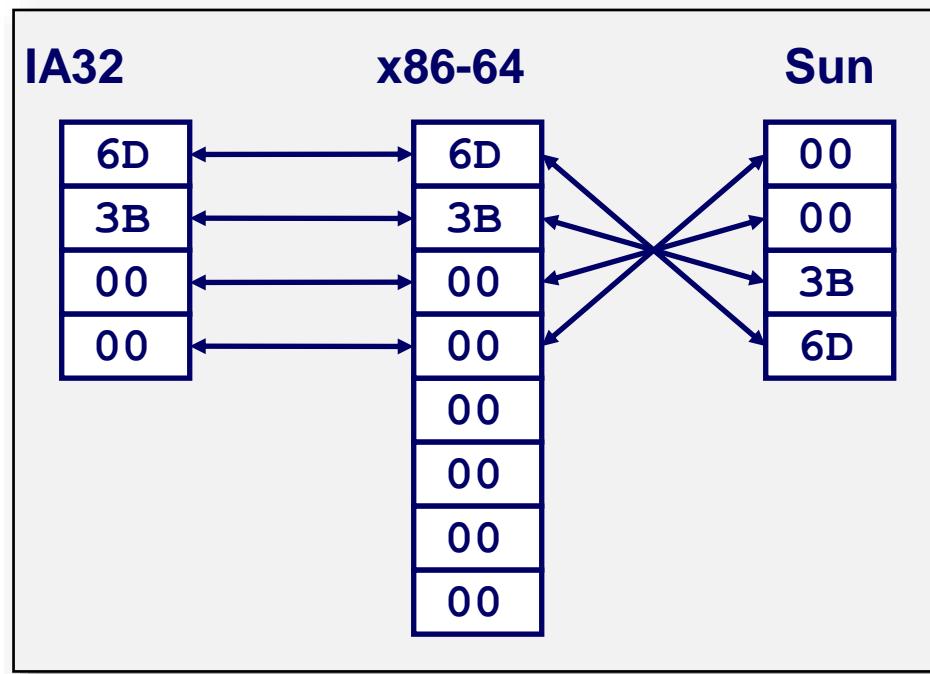


Decimal: 15213

Binary: 0011 1011 0110 1101

Hex: 3 B 6 D

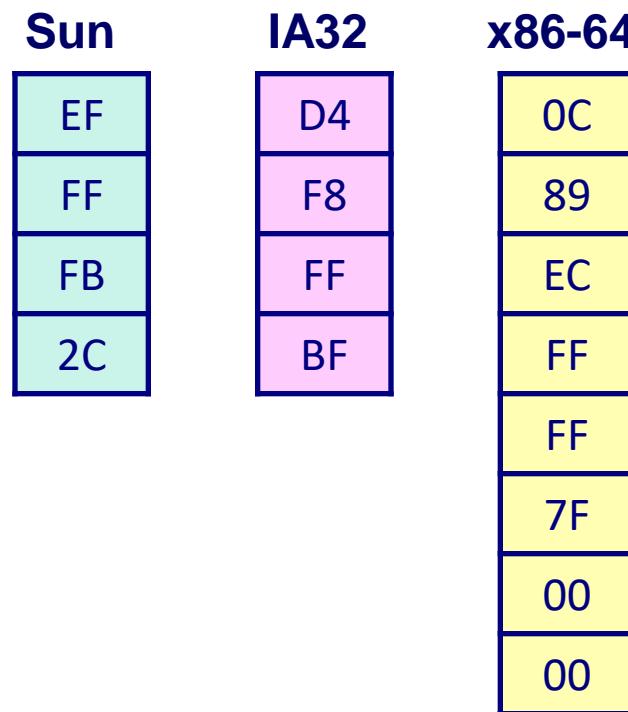
`long int C = 15213;`



Two's complement representation
(Covered later)

Representing Pointers

```
int B = -15213;  
int *P = &B;
```



Different compilers & machines assign different locations to objects

Representing Strings

```
char S[6] = "18243";
```

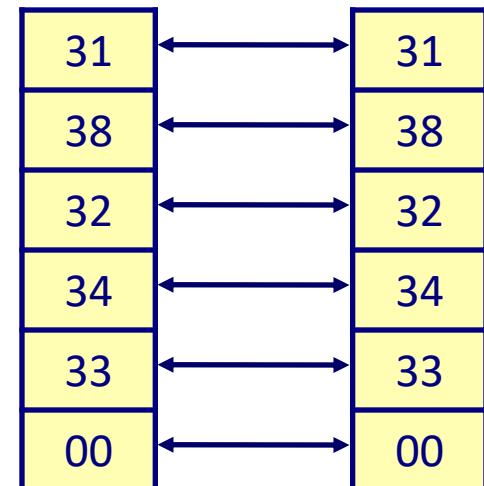
■ Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
 - 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

Linux/Alpha **Sun**



If you have a simple 8-bit character representation (e.g. extended ASCII), then no, endianness does not affect the layout, because each character is one byte.

If you have a multi-byte representation, such as UTF-16, then yes, endianness is still important

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
- Summary

Boolean Algebra

- Developed by George Boole in 19th Century, applied to logic reasoning

- Algebraic representation of logic

And ▪ Encode “True” as 1 and “False” as 0

&	0	1
0	0	0
1	0	1

Or

- $A \& B = 1$ when both $A=1$ and $B=1$

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

	0	1
0	0	1
1	1	1

Not

- $\sim A = 1$ when $A=0$

\sim	
0	1
1	0

Exclusive-Or (Xor)

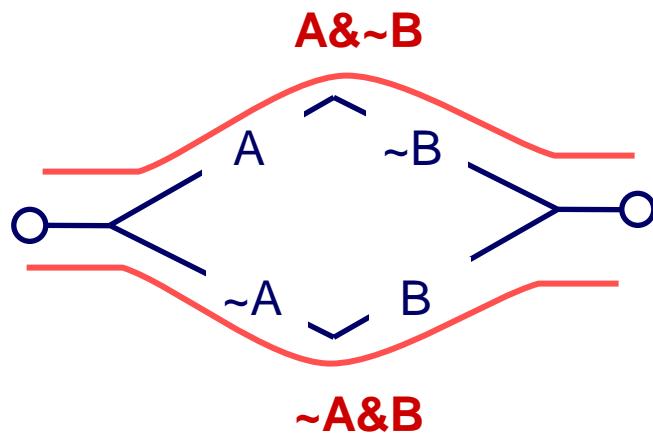
- $A ^ B = 1$ when either $A=1$ or $B=1$, but not both

\wedge	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

$$A \& \sim B \mid \sim A \& B$$

$$= A \wedge B$$

General Boolean Algebras

■ Operate on Bit Vectors

- Operations applied bitwise

$$\begin{array}{rcl} \begin{array}{c} 01101001 \\ \& 01010101 \end{array} & \begin{array}{c} 01101001 \\ | \quad 01010101 \end{array} & \begin{array}{c} 01101001 \\ ^\wedge \quad 01010101 \end{array} \\ \hline \begin{array}{c} 01000001 \\ 01111101 \end{array} & \begin{array}{c} 01111101 \\ 00111100 \end{array} & \begin{array}{c} 10101010 \\ 00111100 \end{array} \end{array}$$

■ All of the Properties of Boolean Algebra Apply

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)

- $\sim 0x41 = 0xBE$
 - $\sim 01000001_2 = 10111110_2$
- $\sim 0x00 = 0xFF$
 - $\sim 00000000_2 = 11111111_2$
- $0x69 \& 0x55 = 0x41$
 - $01101001_2 \& 01010101_2 = 01000001_2$
- $0x69 | 0x55 = 0x7D$
 - $01101001_2 | 01010101_2 = 01111101_2$

Contrast: Logic Operations in C

■ Contrast to Logical Operators

- `&&`, `||`, `!`
 - View 0 as “False”
 - Anything nonzero as “True”
 - Always return 0 or 1

■ Examples (char data type)

- `!0x41 = 0x00`
- `!0x00 = 0x01`
- `!!0x41 = 0x01`
- `0x69 && 0x55 = 0x01`
- `0x69 || 0x55 = 0x01`
- `p && *p` (avoids null pointer access) (**short-circuit**)

Shift Operations

■ Left Shift: $x \ll y$

- Shift bit-vector x left y positions
 - Throw away extra bits on left
 - Fill with 0's on right

■ Right Shift: $x \gg y$

- Shift bit-vector x right y positions
 - Throw away extra bits on right
- Logical shift
 - Fill with 0's on left
- Arithmetic shift
 - Replicate most significant bit on right

■ Undefined Behavior

- Shift amount < 0 or \geq word size

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
 - **Representation: unsigned and signed**
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
- Summary

Encoding Integers

Unsigned

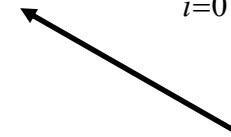
$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

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

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

Sign
Bit



■ 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

Encoding Example (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	

C data type	Minimum	Maximum
char	-128	127
unsigned char	0	255
short [int]	-32,768	32,767
unsigned short [int]	0	65,535
int	-2,147,483,648	2,147,483,647
unsigned [int]	0	4,294,967,295
long [int]	-2,147,483,648	2,147,483,647
unsigned long [int]	0	4,294,967,295
long long [int]	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long [int]	0	18,446,744,073,709,551,615

Figure 2.8 Typical ranges for C integral data types on a 32-bit machine. Text in square brackets is optional.

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

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
 - Representation: unsigned and signed
 - **Conversion, casting**
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
- Summary

Signed numbers

- Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative.
- Add a separate sign, which conveniently can be represented in a single bit; the name for this representation is *sign and magnitude*.
 - where to put the sign bit. To the right? To the left ?
 - adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be.
 - separate sign bit means that sign and magnitude has both a positive and a negative zero

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

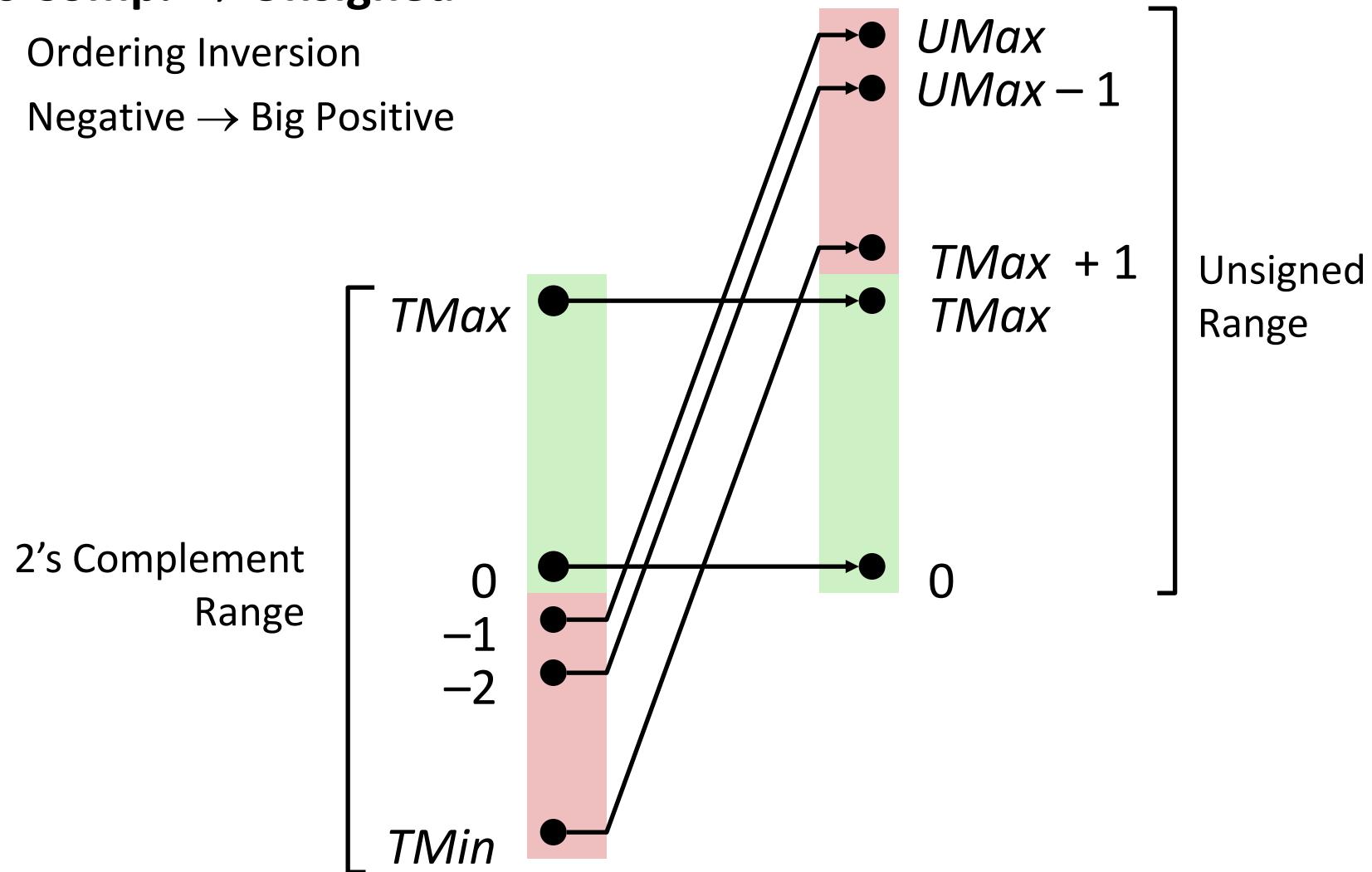
- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
 - 1111 1111 1111 1111 1111 1111 1111 1100₂
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$
- Using 32 bits
 - -2,147,483,648 to +2,147,483,647

leading 0s mean positive, and leading 1s mean negative

Conversion Visualized

■ 2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



2s-Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

EXAMPLE**Binary to Decimal Conversion**

What is the decimal value of this 32-bit two's complement number?

1111 1111 1111 1111 1111 1111 1111 1100_{two}

Substituting the number's bit values into the formula above:

$$\begin{aligned} & (1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \dots + (1 \times 2^1) + (0 \times 2^0) + (0 \times 2^0) \\ &= -2^{31} + 2^{30} + 2^{29} + \dots + 2^2 + 0 + 0 \\ &= -2,147,483,648_{\text{ten}} + 2,147,483,644_{\text{ten}} \\ &= -4_{\text{ten}} \end{aligned}$$

ANSWER

Signed Negation - shortcut

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111\dots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
 - $+2 = 0000\ 0000\dots0010_2$
 - $-2 = 1111\ 1111\dots1101_2 + 1$
 $= 1111\ 1111\dots1110_2$

Signed vs. Unsigned in C

■ Constants

- By default are considered to be **signed** integers
- Unsigned if have “U” as suffix

0U, 4294967259U

■ Casting

- Explicit casting between signed & unsigned same as U2T and T2U

```
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;
```

```
1 short    int     v = -12345;
2 unsigned short uv = (unsigned short) v;
3 printf("v = %d, uv = %u\n", v, uv);
```

When run on a two's-complement machine, it generates the following output:

v = -12345, uv = 53191

```
1 int x = -1;
2 unsigned u = 2147483648; /* 2 to the 31st */
3
4 printf("x = %u = %d\n", x, x);
5 printf("u = %u = %d\n", u, u);
```

When run on a 32-bit machine, it prints the following:

x = 4294967295 = -1
u = 2147483648 = -2147483648

In both cases, `printf` prints the word first as if it represented an unsigned number, and second as if it represented a signed number. We can see the conversion routines in action: $T2U_{32}(-1) = UMax_{32} = 2^{32} - 1$ and $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$.

Casting Surprises

When an operation is performed where one operand is signed and the other is unsigned, C implicitly casts the **signed argument to unsigned** and performs the operations assuming the numbers are nonnegative.

Expression	Type	Evaluation
<code>0 == 0U</code>	unsigned	1
<code>-1 < 0</code>	signed	1
<code>-1 < 0U</code>	unsigned	0 *
<code>2147483647 > -2147483647-1</code>	signed	1
<code>2147483647U > -2147483647-1</code>	unsigned	0 *
<code>2147483647 > (int) 2147483648U</code>	signed	1 *
<code>-1 > -2</code>	signed	1
<code>(unsigned) -1 > -2</code>	unsigned	1

Figure 2.18 Effects of C promotion rules. Nonintuitive cases marked by '*'.

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

getpeername

- There are legions of smart people trying to find vulnerabilities in programs

Typical Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Malicious Usage

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

Summary

Casting Signed \leftrightarrow Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w

- Expression containing signed and unsigned int
 - int is cast to unsigned!!

Floating Point

Instructor

Dr. R. Shathanaa

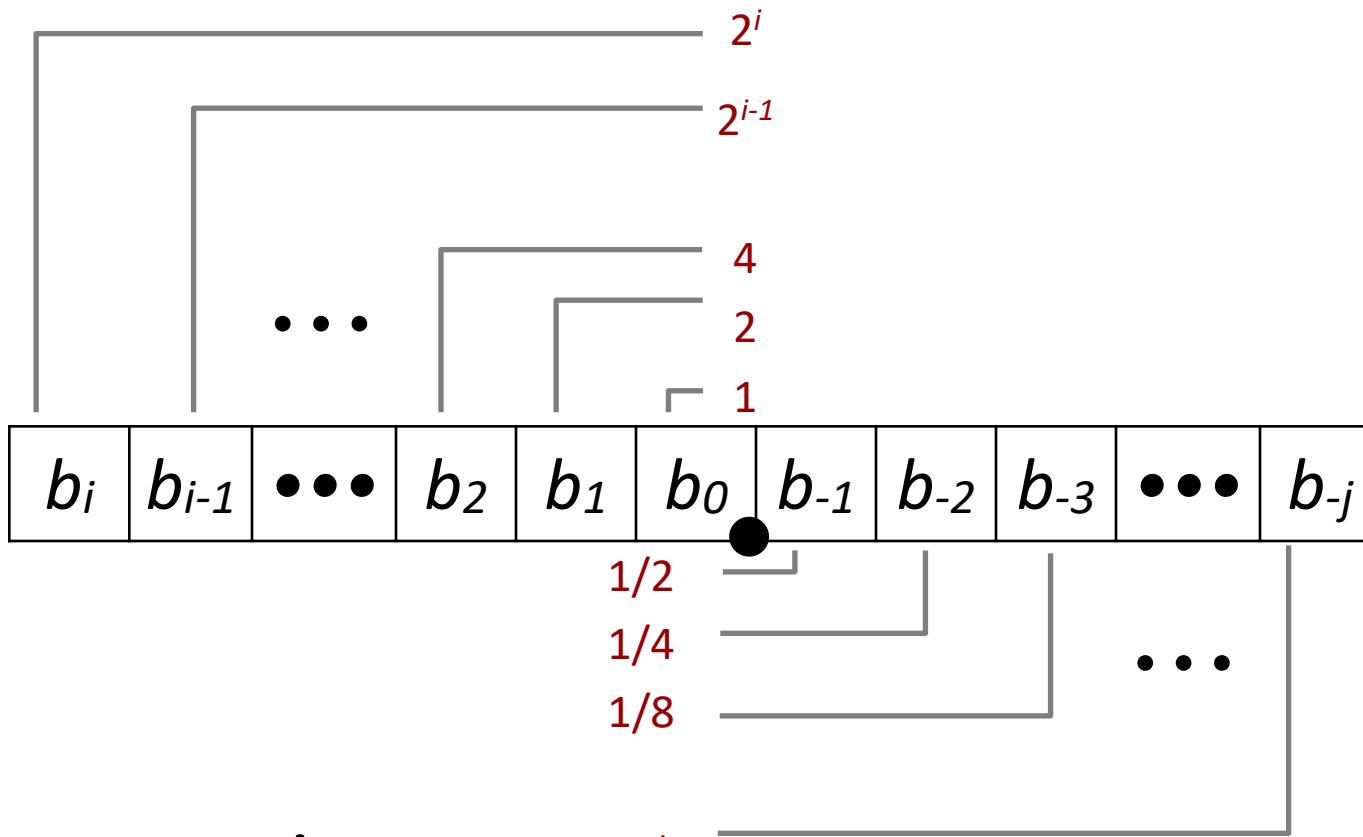
Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Fractional binary numbers

- What is 1011.101_2 ?

Fractional Binary Numbers



■ Representation

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number:

$$\sum_{k=-j}^i b_k \times 2^k$$

Fractional Binary Numbers: Examples

■ Value Representation

5 3/4 101.11₂

2 7/8 10.111₂

63/64 1.0111₂

■ Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form 0.111111...₂ are just below 1.0
 - $1/2 + 1/4 + 1/8 + \dots + 1/2^i + \dots \rightarrow 1.0$
 - Use notation $1.0 - \varepsilon$

Representable Numbers

■ Limitation

- Can represent $x \times 2^y$
- Other rational numbers have repeating bit representations

■ Value Representation

■ 1/3	0.0101010101[01]... ₂
■ 1/5	0.001100110011[0011]... ₂
■ 1/10	0.0001100110011[0011]... ₂

Motivation

- On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dhahran, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people. A report of the General Accounting office, GAO/IMTEC-92-26, entitled Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia reported on the cause of the failure. It turns out that the cause was an inaccurate calculation of the time since boot due to computer arithmetic errors.



- Specifically, the time in tenths of second as measured by the system's internal clock was multiplied by 1/10 to produce the time in seconds. This calculation was performed using a 24 bit fixed point register. In particular, the value 1/10, which has a non-terminating binary expansion, was chopped at 24 bits after the radix point. The Patriot battery had been up around 100 hours, and an easy calculation shows that the resulting time error due to the magnified chopping error was about 0.34 seconds. (The number 1/10 equals $1/24+1/25+1/28+1/29+1/212+1/213+\dots$. In other words, the binary expansion of 1/10 is 0.0001100110011001100110011001100.... Now the 24 bit register in the Patriot stored instead 0.000110011001100110011001100 introducing an error of 0.00000000000000000000000011001100... binary, or about 0.000000095 decimal. Multiplying by the number of tenths of a second in 100 hours gives $0.000000095 \times 100 \times 60 \times 60 \times 10 = 0.34$.)
- A Scud travels at about 1,676 meters per second, and so travels more than half a kilometer in this time. This was far enough that the incoming Scud was outside the "range gate" that the Patriot tracked. Ironically, the fact that the bad time calculation had been improved in some parts of the code, but not all, contributed to the problem, since it meant that the inaccuracies did not cancel, as discussed here.

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

IEEE Floating Point

■ IEEE Standard 754

- Established in 1985 as uniform standard for floating point arithmetic
 - Before that, many idiosyncratic formats
- Supported by all major CPUs

■ Driven by numerical concerns

- Nice standards for rounding, overflow, underflow
- Hard to make fast in hardware
 - Numerical analysts predominated over hardware designers in defining standard

IEEE Floating-Point Format

single: 8 bits

double: 11 bits

single: 23 bits

double: 52 bits

S	Exponent	Fraction/Mantissa
---	----------	-------------------

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$



Significand

- **S: sign bit ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)**
- **Normalize significand: $1.0 \leq |\text{significand}| < 2.0$**
 - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
 - Significand is Fraction with the “1.” restored
- **Exponent: excess representation: actual exponent + Bias**
 - Ensures exponent is unsigned
 - Single: Bias = 127; Double: Bias 1023

Floating-Point Example

■ Represent -0.75

- $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
- $S = 1$
- Fraction = $1000...00_2$
- Exponent = $-1 + \text{Bias}$
 - Single: $-1 + 127 = 126 = 01111110_2$
 - Double: $-1 + 1023 = 1022 = 011111111110_2$

■ Single: $101111101000...00$

■ Double: $1011111111101000...00$

Reason for bias

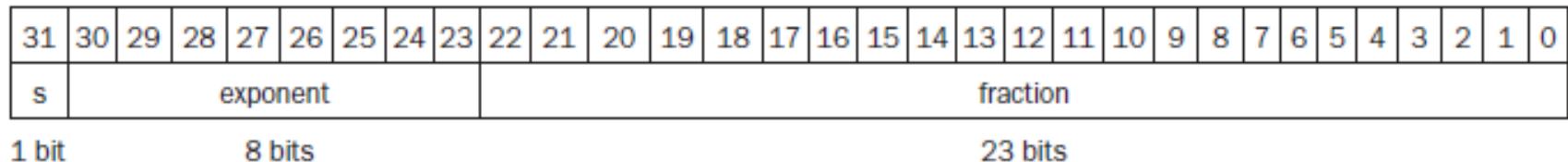
Negative exponents pose a challenge to simplified sorting. If we use two's complement or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field, a negative exponent will look like a big number. For example, $1.0_{\text{two}} \times 2^{-1}$ would be represented as

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(Remember that the leading 1 is implicit in the significand.) The value $1.0_{\text{two}} \times 2^{+1}$ would look like the smaller binary number

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Single Precision



- Overflow - the exponent is too large to be represented in the exponent field
- Underflow - nonzero fraction has become so small that it cannot be represented

Single-Precision Range

- Exponents **00000000** and **11111111** reserved
- Smallest value
 - Exponent: 00000001
 \Rightarrow actual exponent = $1 - 127 = -126$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
 - exponent: 11111110
 \Rightarrow actual exponent = $254 - 127 = +127$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double Precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																															
s	exponent																									fraction																																				
1 bit	11 bits											20 bits																																																		
fraction (continued)																																																														
32 bits																																																														

Double-Precision Range

■ Exponents 0000...00 and 1111...11 reserved

■ Smallest value

- Exponent: 00000000001
 \Rightarrow actual exponent = $1 - 1023 = -1022$
- Fraction: 000...00 \Rightarrow significand = 1.0
- $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$

■ Largest value

- Exponent: 11111111110
 \Rightarrow actual exponent = $2046 - 1023 = +1023$
- Fraction: 111...11 \Rightarrow significand ≈ 2.0
- $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Example

- What number is represented by the single-precision float

11000000101000...00

- S = **1**
 - Fraction = 01000...00₂
 - Exponent = **10000001**₂ = 129
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$
- $$\begin{aligned} &= (-1) \times 1.25 \times 2^2 \\ &= -5.0 \end{aligned}$$

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1–254	Anything	1–2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Single Precision Examples

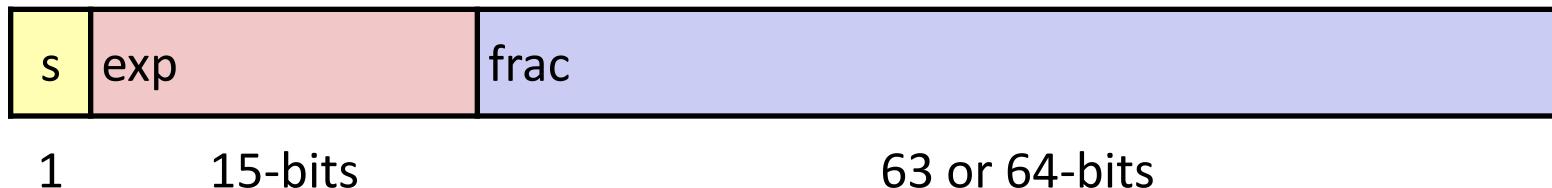
■ Denormalized Numbers

Type	Sign	Actual Exponent	Exp (biased)	Exponent field	Fraction field	Value
Zero	0	-126		0 0000 0000	000 0000 0000 0000 0000 0000 0000 0000	0.0
Negative zero	1	-126		0 0000 0000	000 0000 0000 0000 0000 0000 0000 0000	-0.0
One	0	0	127	0111 1111	000 0000 0000 0000 0000 0000 0000 0000	1.0
Minus One	1	0	127	0111 1111	000 0000 0000 0000 0000 0000 0000 0000	-1.0
Smallest denormalized number	*	-126		0 0000 0000	000 0000 0000 0000 0000 0000 0000 0001	$\pm 2^{-23} \times 2^{-126} = \pm 2^{-149} \approx \pm 1.4 \times 10^{-45}$
"Middle" denormalized number	*	-126		0 0000 0000	100 0000 0000 0000 0000 0000 0000 0000	$\pm 2^{-1} \times 2^{-126} = \pm 2^{-127} \approx \pm 5.88 \times 10^{-39}$
Largest denormalized number	*	-126		0 0000 0000	111 1111 1111 1111 1111 1111	$\pm(1-2^{-23}) \times 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Smallest normalized number	*	-126		1 0000 0001	000 0000 0000 0000 0000 0000 0000 0000	$\pm 2^{-126} \approx \pm 1.18 \times 10^{-38}$
Largest normalized number	*	127	254	1111 1110	111 1111 1111 1111 1111 1111	$\pm(2-2^{-23}) \times 2^{127} \approx \pm 3.4 \times 10^{38}$
Positive infinity	0	128	255	1111 1111	000 0000 0000 0000 0000 0000 0000 0000	$+\infty$
Negative infinity	1	128	255	1111 1111	000 0000 0000 0000 0000 0000 0000 0000	$-\infty$
Not a number	*	128	255	1111 1111	non zero	NaN

* Sign bit can be either 0 or 1.

Precisions

- Extended precision: 80 bits (Intel only)



Special Properties of Encoding

■ FP Zero Same as Integer Zero

- All bits = 0

■ Can (Almost) Use Unsigned Integer Comparison

- Must first compare sign bits
- Must consider $-0 = 0$
- NaNs problematic
 - Will be greater than any other values
 - What should comparison yield?
- Otherwise OK
 - Denorm vs. normalized
 - Normalized vs. infinity

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- **Rounding, addition, multiplication**
- Floating point in C
- Summary

Floating Point Operations: Basic Idea

- $x +_f y = \text{Round}(x + y)$
- $x \times_f y = \text{Round}(x \times y)$
- **Basic idea**
 - First **compute exact result**
 - Make it fit into desired precision
 - Possibly overflow if exponent too large
 - Possibly **round to fit into `frac`**

Rounding

■ Rounding Modes (illustrate with \$ rounding)

	\$1.40	\$1.60	\$1.50	\$2.50	-\$1.50
■ Towards zero	\$1	\$1	\$1	\$2	-\$1
■ Round down ($-\infty$)	\$1	\$1	\$1	\$2	-\$2
■ Round up ($+\infty$)	\$2	\$2	\$2	\$3	-\$1
■ Nearest Even (default)	\$1	\$2	\$2	\$2	-\$2

■ What are the advantages of the modes?

Closer Look at Round-To-Even

■ Default Rounding Mode

- All others are statistically biased
 - Sum of set of positive numbers will consistently be over- or under-estimated

■ Applying to Other Decimal Places / Bit Positions

- When exactly halfway between two possible values
 - Round so that least significant digit is even
- E.g., round to nearest hundredth

1.2349999 1.23 (Less than half way)

1.2350001 1.24 (Greater than half way)

1.2350000 1.24 (Half way—round up)

1.2450000 1.24 (Half way—round down)

Rounding Binary Numbers

■ Binary Fractional Numbers

- “Even” when least significant bit is 0
- “Half way” when bits to right of rounding position = 100...₂

■ Examples

- Round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.000 11 ₂	10.00 ₂	(<1/2—down)	2
2 3/16	10.00 110 ₂	10.01 ₂	(>1/2—up)	2 1/4
2 7/8	10.11 100 ₂	11.00 ₂	(1/2—up)	3
2 5/8	10.10 100 ₂	10.10 ₂	(1/2—down)	2 1/2

Floating-Point Addition

Consider a 4-digit decimal example

$$9.999 \times 10^1 + 1.610 \times 10^{-1}$$

1. Align decimal points

Shift number with smaller exponent

$$9.999 \times 10^1 + 0.016 \times 10^1$$

2. Add significands

$$9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$$

3. Normalize result & check for over/underflow

$$1.0015 \times 10^2$$

4. Round and renormalize if necessary. Assume only four digits are allowed for significant and two digits for exponent

$$1.002 \times 10^2$$

Floating-Point Addition

Now consider a 4-digit binary example

$$1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2} (0.5 + -0.4375)$$

1. Align binary points

Shift number with smaller exponent

$$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$$

2. Add significands

$$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$$

3. Normalize result & check for over/underflow

$$1.000_2 \times 2^{-4}, (\text{no over/underflow } 127 \geq -4 \geq -126)$$

4. Round and renormalize if necessary

$$1.000_2 \times 2^{-4} (\text{no change}) = 0.0625$$

Floating-Point Multiplication

- Consider a 4-digit decimal example

- $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$

- 1. Add exponents

- For biased exponents, subtract bias from sum
 - New exponent = $10 + -5 = 5$

- 2. Multiply significands

$$1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$$

- 3. Normalize result & check for over/underflow

$$1.0212 \times 10^6$$

- 4. Round and renormalize if necessary

$$1.021 \times 10^6$$

- 5. Determine sign of result from signs of operands

$$+1.021 \times 10^6$$

Biased $10 + 127 = 137$, and $-5 + 127 = 122$,

New exponent

$137 + 122 = 259$ Wrong !!!

$(10 + 127) + (-5 + 127) = (5 + 2 * 127) = 259$

we must subtract the bias from the sum:

New exponent $137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$

Floating-Point Multiplication

■ Now consider a 4-digit binary example

- $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)

■ 1. Add exponents

- Unbiased: $-1 + -2 = -3$
- Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$

■ 2. Multiply significands

- $1.000_2 \times 1.110_2 = 1.110000_2 \Rightarrow 1.110_2 \times 2^{-3}$

■ 3. Normalize result & check for over/underflow

- $1.110_2 \times 2^{-3}$ (no change) & $127 \geq -3 \geq -126$ no over/underflow

■ 4. Round and renormalize if necessary

- $1.110_2 \times 2^{-3}$ (no change)

■ 5. Determine sign: +ve \times -ve \Rightarrow -ve

- $-1.110_2 \times 2^{-3} = -0.21875$

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- **Floating point in C**
- Summary

Floating Point in C

■ C Guarantees Two Levels

- **float** single precision
- **double** double precision

■ Conversions/Casting

- Casting between **int**, **float**, and **double** changes bit representation
- **double/float → int**
 - Truncates fractional part
 - Like rounding toward zero
 - Not defined when out of range or NaN: Generally sets to TMin
- **int → double**
 - Exact conversion, as long as **int** has \leq 53 bit word size
- **int → float**
 - Will round according to rounding mode

Floating Point Puzzles

■ For each of the following C expressions, either:

- Argue that it is true for all argument values
- Explain why not true

- $x == (\text{int})(\text{float}) x$
- $x == (\text{int})(\text{double}) x$
- $f == (\text{float})(\text{double}) f$
- $d == (\text{float}) d$
- $f == -(-f);$
- $2/3 == 2/3.0$
- $d < 0.0 \Rightarrow ((d^2) < 0.0)$
- $d > f \Rightarrow -f > -d$
- $d * d >= 0.0$
- $(d+f)-d == f$

```
int x = ...;
float f = ...;
double d = ...;
```

Assume neither
d nor **f** is NaN

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers of form $M \times 2^E$
- One can reason about operations independent of implementation
 - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
 - Violates associativity/distributivity
 - Makes life difficult for compilers & serious numerical applications programmers

Bits, Bytes, and Integers

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
 - Representation: unsigned and signed
 - Conversion, casting
 - **Expanding, truncating**
 - Addition, negation, multiplication, shifting
- Summary

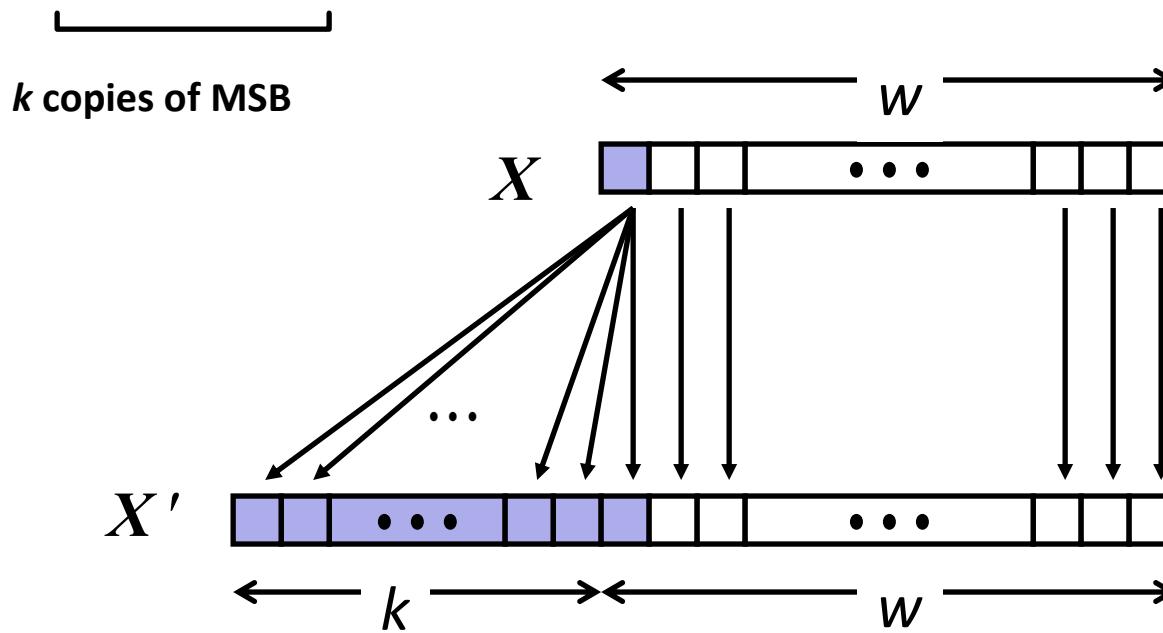
Sign Extension

■ Task:

- Given w -bit signed integer x
- Convert it to $w+k$ -bit integer with same value

■ Rule:

- Make k copies of sign bit:
- $X' = x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0$



Sign Extension Example

```
1 short sx = -12345;          /* -12345 */
2 unsigned short usx = sx;    /* 53191 */
3 int x = sx;                /* -12345 */
4 unsigned ux = usx;          /* 53191 */
5
6 printf("sx = %d:\t", sx);
7 show_bytes((byte_pointer) &sx, sizeof(short));
8 printf("usx = %u:\t", usx);
9 show_bytes((byte_pointer) &usx, sizeof(unsigned short));
10 printf("x = %d:\t", x);
11 show_bytes((byte_pointer) &x, sizeof(int));
12 printf("ux = %u:\t", ux);
13 show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

When run on a 32-bit big-endian machine using a two's-complement representation, this code prints the output

```
sx = -12345: cf c7
usx = 53191: cf c7
x = -12345: ff ff cf c7
ux = 53191: 00 00 cf c7
```

Sign Extension

One point worth making is that the relative order of conversion from one data size to another and between unsigned and signed can affect the behavior of a program.

```
1 short sx = -12345;          /* -12345 */
2 unsigned uy = sx;           /* Mystery! */
3
4 printf("uy = %u:\t", uy);
5 show_bytes((byte_pointer) &uy, sizeof(unsigned));

uy = 4294954951: ff ff cf c7
```

This shows that, when converting from short to unsigned, the program first changes the size and then the type. That is, (unsigned) sx is equivalent to (unsigned) (int) sx, evaluating to 4,294,954,951, not (unsigned) (unsigned short) sx, which evaluates to 53,191.

Truncating

```
int x = 53191;
short sx = (short) x; /* -12345 */
int y = sx; /* -12345 */
53191 = 00000000 00000000 11001111 11000111
-12345 = 11001111 11000111
```

- When truncating a w -bit number $x = [x_{w-1}, x_{w-2}, \dots, x_0]$ to a k -bit number, we drop the high-order $w - k$ bits
- Truncating a number can alter its value—a form of overflow.
- For an unsigned number x , the result of truncating it to k bits is equivalent to computing $x \bmod 2^k$

$$\begin{aligned}
 B2U_w([x_{w-1}, x_{w-2}, \dots, x_0]) \bmod 2^k &= \left[\sum_{i=0}^{w-1} x_i 2^i \right] \bmod 2^k \\
 &= \left[\sum_{i=0}^{k-1} x_i 2^i \right] \bmod 2^k \\
 &= \sum_{i=0}^{k-1} x_i 2^i \\
 &= B2U_k([x_{k-1}, x_{k-2}, \dots, x_0])
 \end{aligned}$$

In this derivation, we make use of the property:

$$2^i \bmod 2^k = 0 \text{ for any } i \geq k$$

The same is applicable for signed numbers

Summary:

Expanding, Truncating: Basic Rules

■ Expanding (e.g., short int to int)

- Unsigned: zeros added
- Signed: sign extension
- Both yield expected result

■ Truncating (e.g., unsigned to unsigned short)

- Unsigned/signed: bits are truncated
- Result reinterpreted
- Unsigned: mod operation
- Signed: similar to mod
- For small numbers yields expected behaviour

Practice

Suppose we truncate a 4-bit value (represented by hex digits 0 through F) to a 3-bit value (represented as hex digits 0 through 7). Fill in the table below showing the effect of this truncation for some cases, in terms of the unsigned and two's-complement interpretations of those bit patterns.

Hex		Unsigned		Two's complement	
Original	Truncated	Original	Truncated	Original	Truncated
0	0	0	_____	0	_____
2	2	2	_____	2	_____
9	1	9	_____	-7	_____
B	3	11	_____	-5	_____
F	7	15	_____	-1	_____

Today: Bits, Bytes, and Integers

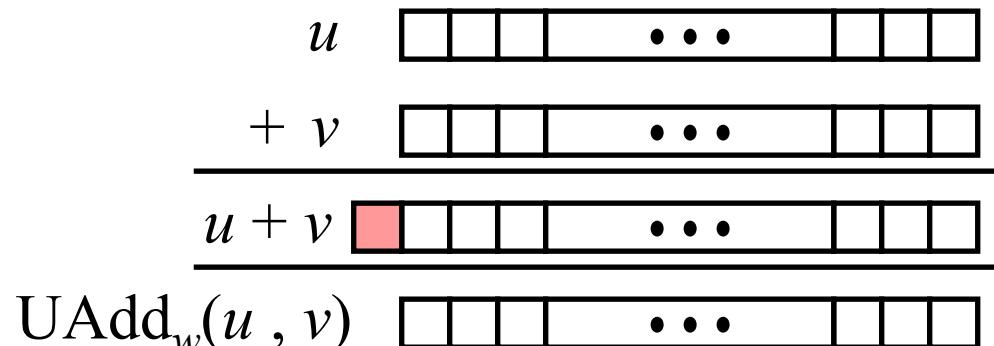
- Representing information as bits
- Bit-level manipulations
- **Integers**
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - **Addition, negation, multiplication, shifting**
- Summary

Unsigned Addition

Operands: w bits

True Sum: $w+1$ bits

Discard Carry: w bits



■ Standard Addition Function

- Ignores carry output

■ Implements Modular Arithmetic

$$s = \text{UAdd}_w(u, v) = u + v \bmod 2^w$$

$$\text{UAdd}_w(u, v) = \begin{cases} u + v & u + v < 2^w \\ u + v - 2^w & u + v \geq 2^w \end{cases}$$

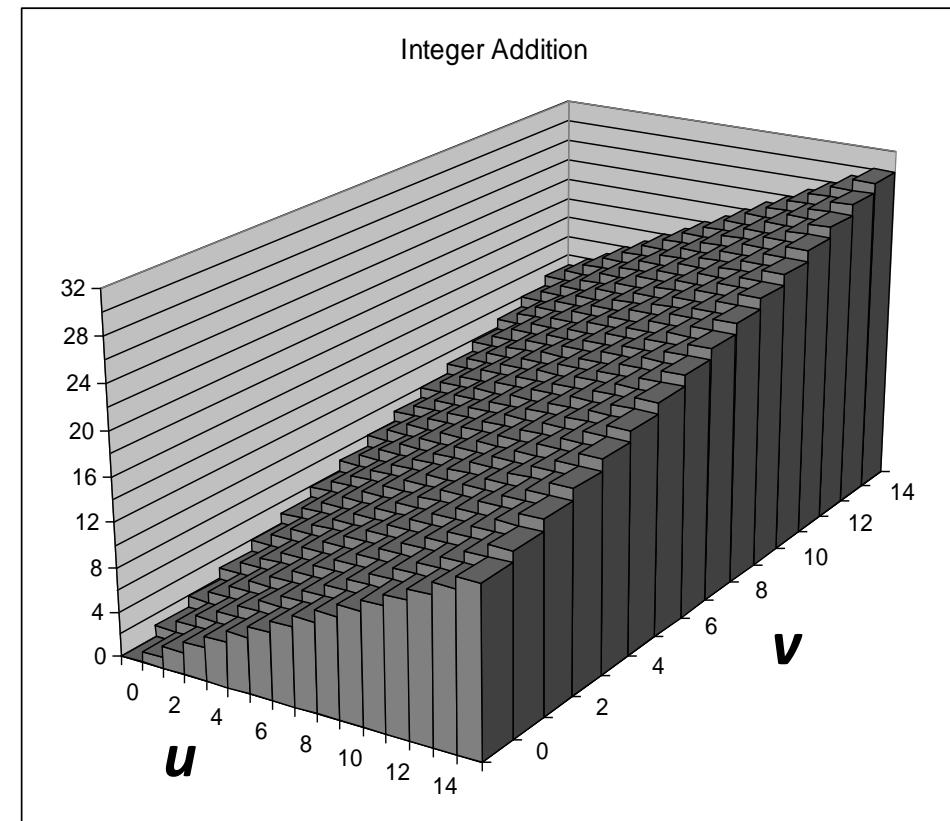
For example, consider a 4-bit number representation with $x = 9$ and $y = 12$, having bit representations [1001] and [1100], respectively. Their sum is 21, having a 5-bit representation [10101]. But if we discard the high-order bit, we get [0101], that is, decimal value 5. This matches the value $21 \bmod 16 = 5$

Visualizing (Mathematical) Integer Addition

■ Integer Addition

- 4-bit integers u, v
- Compute true sum
 $\text{Add}_4(u, v)$
- Values increase linearly
with u and v
- Forms planar surface

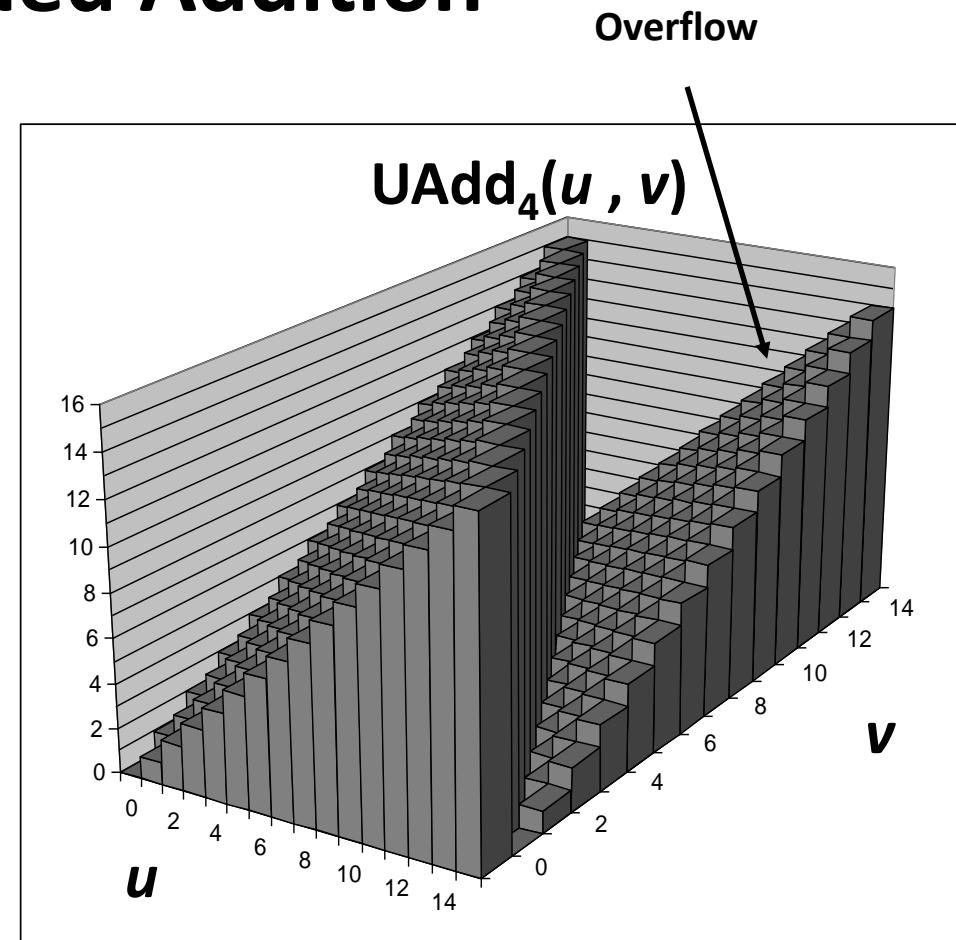
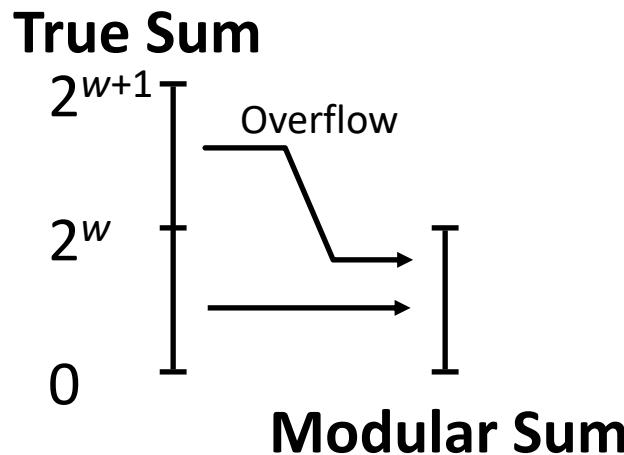
$\text{Add}_4(u, v)$



Visualizing Unsigned Addition

Wraps Around

- If true sum $\geq 2^w$
- At most once



When executing C programs, overflows are not signaled as errors. At times, however, we might wish to determine whether overflow has occurred. For example, suppose we compute $s = x + y$, and we wish to determine whether s equals $x + y$. We claim that overflow has occurred if and only if $s < x$ (or equivalently, $s < y$)

Mathematical Properties

■ Modular Addition Forms an *Abelian Group*

- **Closed** under addition

$$0 \leq \text{UAdd}_w(u, v) \leq 2^w - 1$$

- **Commutative**

$$\text{UAdd}_w(u, v) = \text{UAdd}_w(v, u)$$

- **Associative**

$$\text{UAdd}_w(t, \text{UAdd}_w(u, v)) = \text{UAdd}_w(\text{UAdd}_w(t, u), v)$$

- **0** is additive identity

$$\text{UAdd}_w(u, 0) = u$$

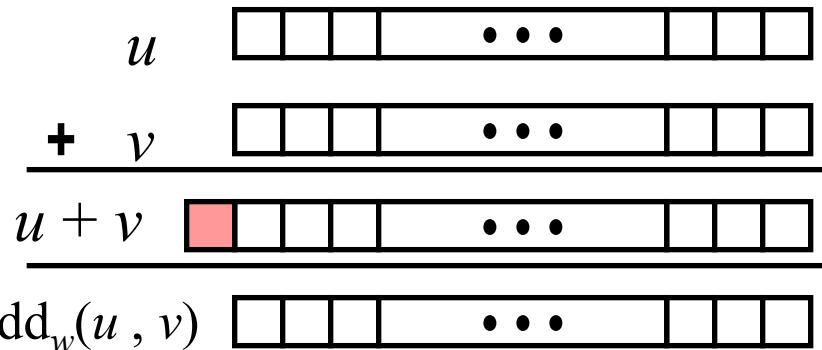
- Every element has additive **inverse**

- Let $\text{UComp}_w(u) = 2^w - u$

$$\text{UAdd}_w(u, \text{UComp}_w(u)) = 0$$

Two's Complement Addition

Operands: w bits



True Sum: $w+1$ bits

Discard Carry: w bits

TAdd_w(u, v) \cdots

■ TAdd and UAdd have Identical Bit-Level Behavior

- Signed vs. unsigned addition in C:

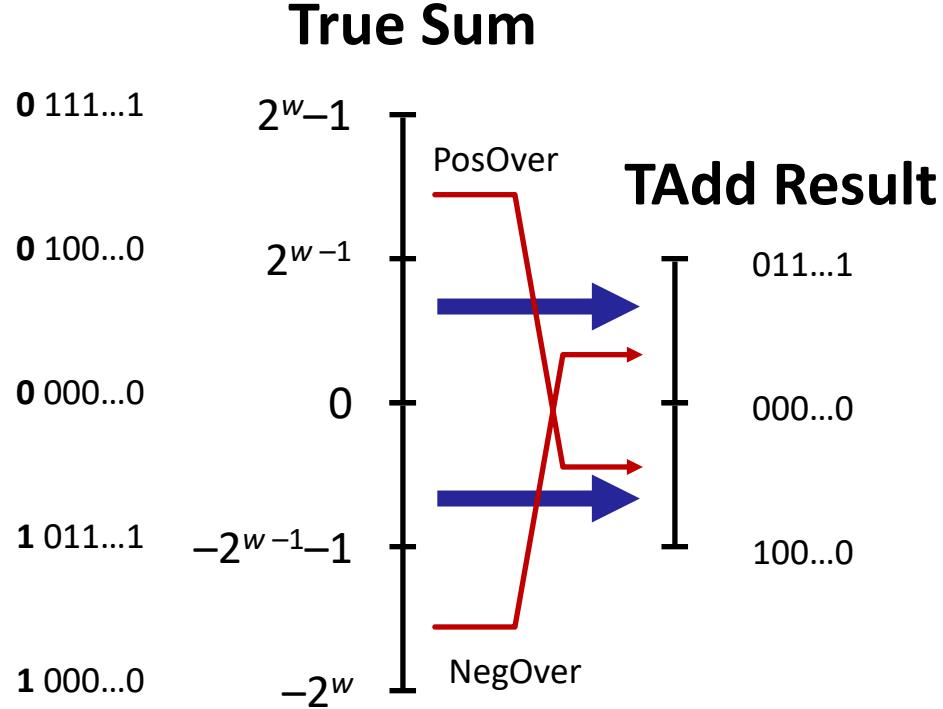
```
int s, t, u, v;  
s = (int) ((unsigned) u + (unsigned) v);  
t = u + v
```

- Will give $s == t$

TAdd Overflow

■ Functionality

- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer



Visualizing 2's Complement Addition

■ Values

- 4-bit two's comp.
- Range from -8 to +7

■ Wraps Around

- If sum $\geq 2^{w-1}$
 - Becomes negative
 - At most once
- If sum $< -2^{w-1}$
 - Becomes positive
 - At most once

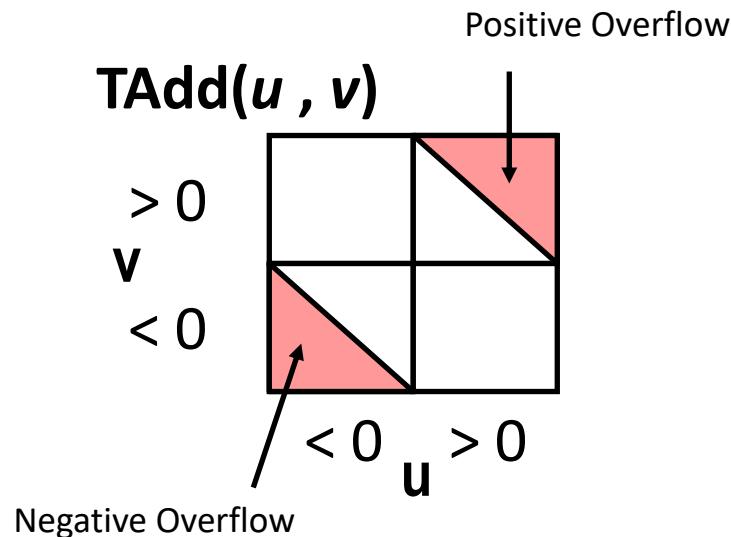
x	y	$x + y$	$x +_4^t y$	Case
-8 [1000]	-5 [1011]	-13 [10011]	3 [0011]	1
-8 [1000]	-8 [1000]	-16 [10000]	0 [0000]	1
-8 [1000]	5 [0101]	-3 [11101]	-3 [1101]	2
2 [0010]	5 [0101]	7 [00111]	7 [0111]	3
5 [0101]	5 [0101]	10 [01010]	-6 [1010]	4

Figure 2.24 Two's-complement addition examples. The bit-level representation of the 4-bit two's-complement sum can be obtained by performing binary addition of the operands and truncating the result to 4 bits.

Characterizing TAdd

■ Functionality

- True sum requires $w+1$ bits
- Drop off MSB
- Treat remaining bits as 2's comp. integer



$$TAdd_w(u, v) = \begin{cases} u + v + 2^w & u + v < TMin_w \text{ (NegOver)} \\ u + v & TMin_w \leq u + v \leq TMax_w \\ u + v - 2^w & TMax_w < u + v \text{ (PosOver)} \end{cases}$$

Mathematical Properties of TAdd

■ Isomorphic Group to unsigned with UAdd

- $TAdd_w(u, v) = U2T(UAdd_w(T2U(u), T2U(v)))$
 - Since both have identical bit patterns

■ Two's Complement Under TAdd Forms a Group

- Closed, Commutative, Associative, 0 is additive identity
- Every element has additive inverse

$$TComp_w(u) = \begin{cases} -u & u \neq TMin_w \\ TMin_w & u = TMin_w \end{cases}$$

Practice

x	y	$x + y$	$x +_5^t y$	Case
[10100]	[10001]	_____	_____	_____
[11000]	[11000]	_____	_____	_____
[10111]	[01000]	_____	_____	_____
[00010]	[00101]	_____	_____	_____
[01100]	[00100]	_____	_____	_____

Multiplication

■ Computing Exact Product of w -bit numbers x, y

- Either signed or unsigned

■ Ranges

- Unsigned: $0 \leq x * y \leq (2^w - 1)^2 = 0$ and $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
- Two's complement min: $x * y \geq (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$
- Two's complement max: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- Up to $2w$ bits

■ Maintaining Exact Results

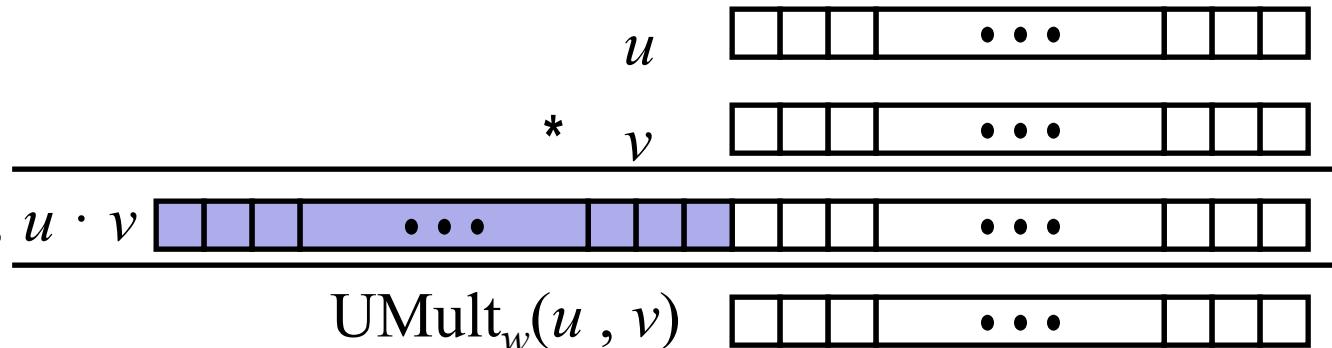
- Would need to keep expanding word size with each product computed
- Done in software by “arbitrary precision” arithmetic packages

Unsigned Multiplication in C

Operands: w bits

True Product: 2^w bits

Discard w bits: w bits



■ Standard Multiplication Function

- Ignores high order w bits

■ Implements Modular Arithmetic

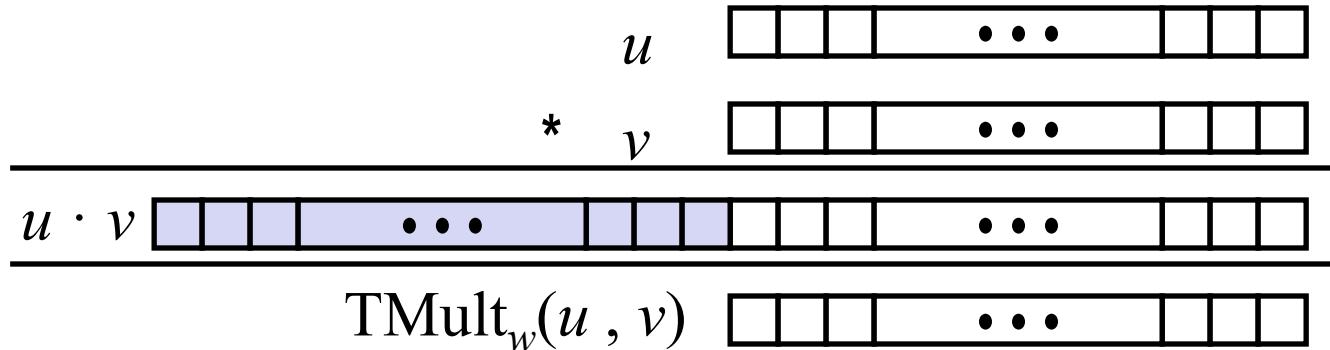
$$\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$$

Signed Multiplication in C

Operands: w bits

True Product: 2^w bits

Discard w bits: w bits



■ Standard Multiplication Function

- Ignores high order w bits
- Some of which are different for signed vs. unsigned multiplication
- Lower bits are the same

Truncating a two's-complement number to w bits is equivalent to first computing its value modulo 2^w and then converting from unsigned to two's complement.

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
Two's comp.	-3	[101]	3	[011]	-9	[110111]	-1	[111]
Unsigned	4	[100]	7	[111]	28	[011100]	4	[100]
Two's comp.	-4	[100]	-1	[111]	4	[000100]	-4	[100]
Unsigned	3	[011]	3	[011]	9	[001001]	1	[001]
Two's comp.	3	[011]	3	[011]	9	[001001]	1	[001]

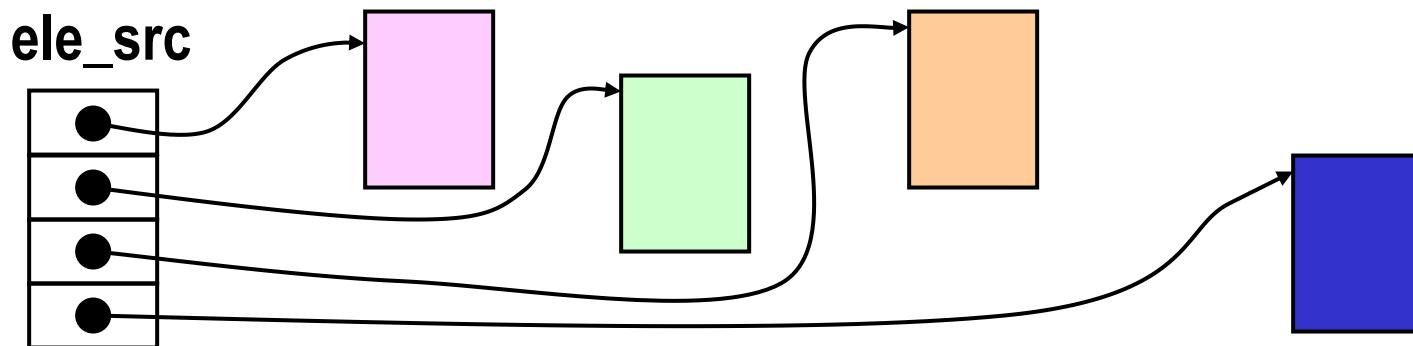
Figure 2.26 Three-bit unsigned and two's-complement multiplication examples. Although the bit-level representations of the full products may differ, those of the truncated products are identical.

Code Security Example #2

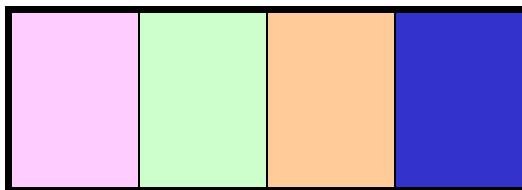
■ SUN XDR library

- Widely used library for transferring data between machines

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```



`malloc(ele_cnt * ele_size)`



XDR Code

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

XDR Vulnerability

`malloc(ele_cnt * ele_size)`

■ What if:

- `ele_cnt` = $2^{20} + 1$
- `ele_size` = 4096 = 2^{12}
- Allocation = ??
- Allocation needed: 4,294,971,392
- Actual allocation: 4096

■ How can I make this function secure?

Power-of-2 Multiply with Shift

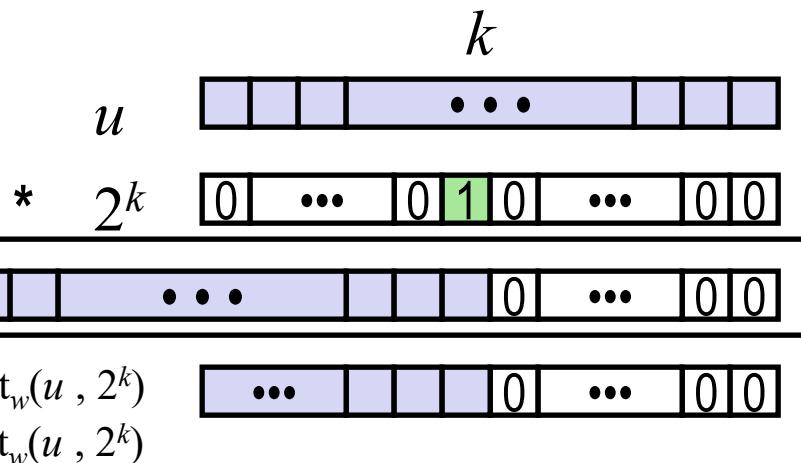
■ Operation

- $u \ll k$ gives $u * 2^k$
- Both signed and unsigned

Operands: w bits

True Product: $w+k$ bits

Discard k bits: w bits



■ Examples

- $u \ll 3 == u * 8$
- $u \ll 5 - u \ll 3 == u * 24$ ($24 = 2^5 - 2^3$)
- Most machines shift and add faster than multiply
 - Compiler generates this code automatically

Compiled Multiplication Code

C Function

```
int mul12(int x)
{
    return x*12;
}
```

Compiled Arithmetic Operations

```
leal (%eax,%eax,2), %eax
sall $2, %eax
```

Explanation

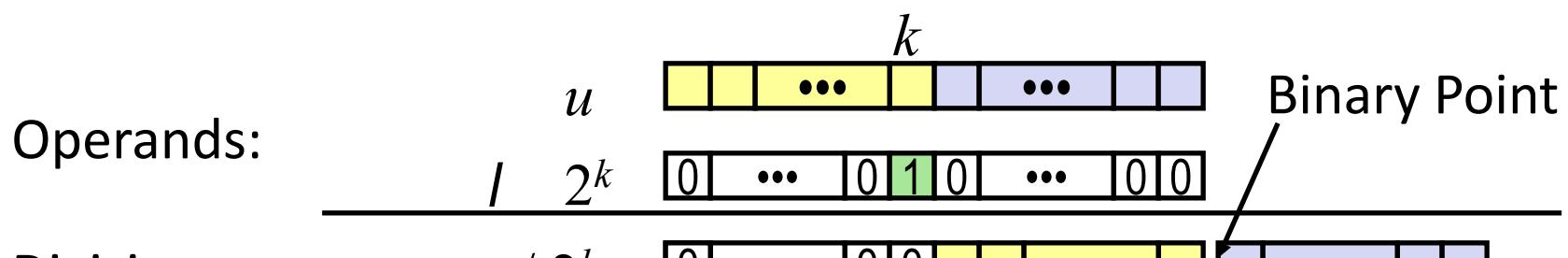
```
t <- x+x*2
return t << 2;
```

- C compiler automatically generates shift/add code when multiplying by constant

Unsigned Power-of-2 Divide with Shift

■ Quotient of Unsigned by Power of 2

- $u \gg k$ gives $\lfloor u / 2^k \rfloor$
- Uses logical shift



Result: $\lfloor u / 2^k \rfloor$

k	$\gg k$ (Binary)	Decimal	$12340/2^k$
0	0011000000110100	12340	12340.0
1	0001100000011010	6170	6170.0
4	0000001100000011	771	771.25
8	0000000000110000	48	48.203125

Figure 2.27 Dividing unsigned numbers by powers of 2. The examples illustrate how performing a logical right shift by k has the same effect as dividing by 2^k and then rounding toward zero.

Compiled Unsigned Division Code

C Function

```
unsigned udiv8(unsigned x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

```
shrl $3, %eax
```

Explanation

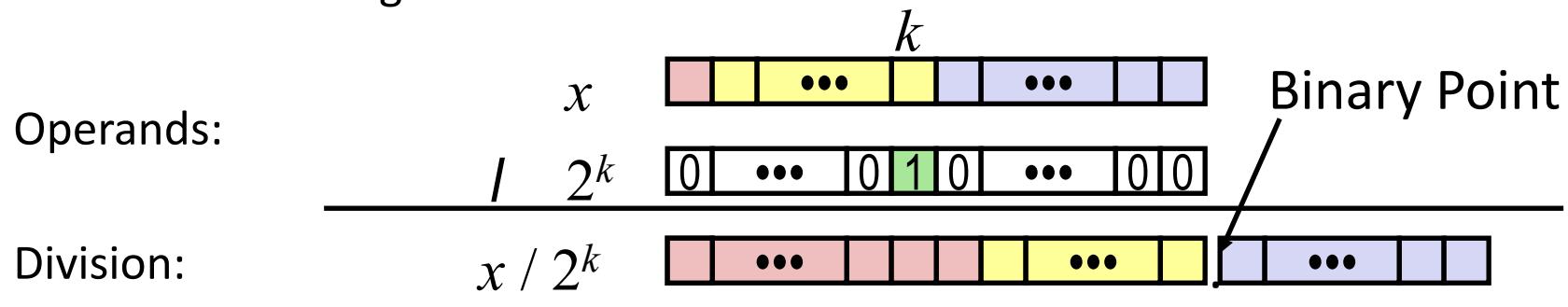
```
# Logical shift
return x >> 3;
```

- Uses logical shift for unsigned
- For Java Users
 - Logical shift written as >>>

Signed Power-of-2 Divide with Shift

■ Quotient of Signed by Power of 2

- $x \gg k$ gives $\lfloor x / 2^k \rfloor$
 - Uses arithmetic shift
 - Rounds wrong direction when $u < 0$



k	>> k (Binary)	Decimal	$-12340/2^k$
0	110011111001100	-12340	-12340.0
1	111001111100110	-6170	-6170.0
4	1111110011111100	-772	-771.25
8	111111111001111	-49	-48.203125

Figure 2.28 Applying arithmetic right shift. The examples illustrate that arithmetic right shift is similar to division by a power of 2, except that it rounds down rather than toward zero.

Correct division

■ Quotient of Negative Number by Power of 2

- Want $\lceil x / 2^k \rceil$ (Round Toward 0)
- Compute as $\lfloor (x+2^k-1) / 2^k \rfloor$
 - In C: `(x + (1<<k)-1) >> k`
 - Biases dividend toward 0

This technique exploits the property that $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$ for integers x and y such that $y > 0$. As examples, when $x = -30$ and $y = 4$, we have $x + y - 1 = -27$, and $\lceil -30/4 \rceil = -7 = \lfloor -27/4 \rfloor$.

k	Bias	$-12,340 + \text{Bias (Binary)}$	$>> k$ (Binary)	Decimal	$-12340/2^k$
0	0	1100111111001100	1100111111001100	-12340	-12340.0
1	1	1100111111001101	1110011111100110	-6170	-6170.0
4	15	1100111111011011	1111110011111101	-771	-771.25
8	255	1101000011001011	1111111111010000	-48	-48.203125

Figure 2.29 Dividing two's-complement numbers by powers of 2. By adding a bias before the right shift, the result is rounded toward zero.

Compiled Signed Division Code

C Function

```
int idiv8(int x)
{
    return x/8;
}
```

Compiled Arithmetic Operations

```
testl %eax, %eax
js    L4
L3:
    sarl $3, %eax
    ret
L4:
    addl $7, %eax
    jmp  L3
```

Explanation

```
if x < 0
    x += 7;
# Arithmetic shift
return x >> 3;
```

- Uses arithmetic shift for int
- For Java Users
 - Arith. shift written as >>

Arithmetic: Basic Rules

■ Addition:

- Unsigned/signed: Normal addition followed by truncate, same operation on bit level
- Unsigned: addition mod 2^w
 - Mathematical addition + possible subtraction of 2^w
- Signed: modified addition mod 2^w (result in proper range)
 - Mathematical addition + possible addition or subtraction of 2^w

■ Multiplication:

- Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
- Unsigned: multiplication mod 2^w
- Signed: modified multiplication mod 2^w (result in proper range)

Arithmetic: Basic Rules

■ Left shift

- Unsigned/signed: multiplication by 2^k
- Always logical shift

■ Right shift

- Unsigned: logical shift, div (division + round to zero) by 2^k
 - Signed: arithmetic shift
 - Positive numbers: div (division + round to zero) by 2^k
 - Negative numbers: div (division + round away from zero) by 2^k
- Use biasing to fix

Today: Integers

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

Integer C Puzzles

Initialization

```
int x = foo();  
  
int y = bar();  
  
unsigned ux = x;  
  
unsigned uy = y;
```

- $x < 0 \Rightarrow ((x^2) < 0)$
- $ux \geq 0$
- $x \& 7 == 7 \Rightarrow (x \ll 30) < 0$
- $ux > -1$
- $x > y \Rightarrow -x < -y$
- $x * x \geq 0$
- $x > 0 \&& y > 0 \Rightarrow x + y > 0$
- $x \geq 0 \Rightarrow -x \leq 0$
- $x \leq 0 \Rightarrow -x \geq 0$
- $(x|-x) \gg 31 == -1$
- $ux \gg 3 == ux/8$
- $x \gg 3 == x/8$
- $x \& (x-1) != 0$

Machine-Level Programming I: Basics

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Intel x86 Processors

- x86 is a family of instruction set architectures initially developed by Intel based on the Intel 8086 microprocessor and its 8088 variant.
- The 8086 was introduced in 1978 as a fully 16-bit extension of Intel's 8-bit 8080 microprocessor, with memory segmentation as a solution for addressing more memory than can be covered by a plain 16-bit address.
- The term "x86" came into being because the names of several successors to Intel's 8086 processor end in "86", including the 80186, 80286, 80386 and 80486 processors.
- Dominate laptop/desktop/server market

Intel x86 Processors

■ Evolutionary design

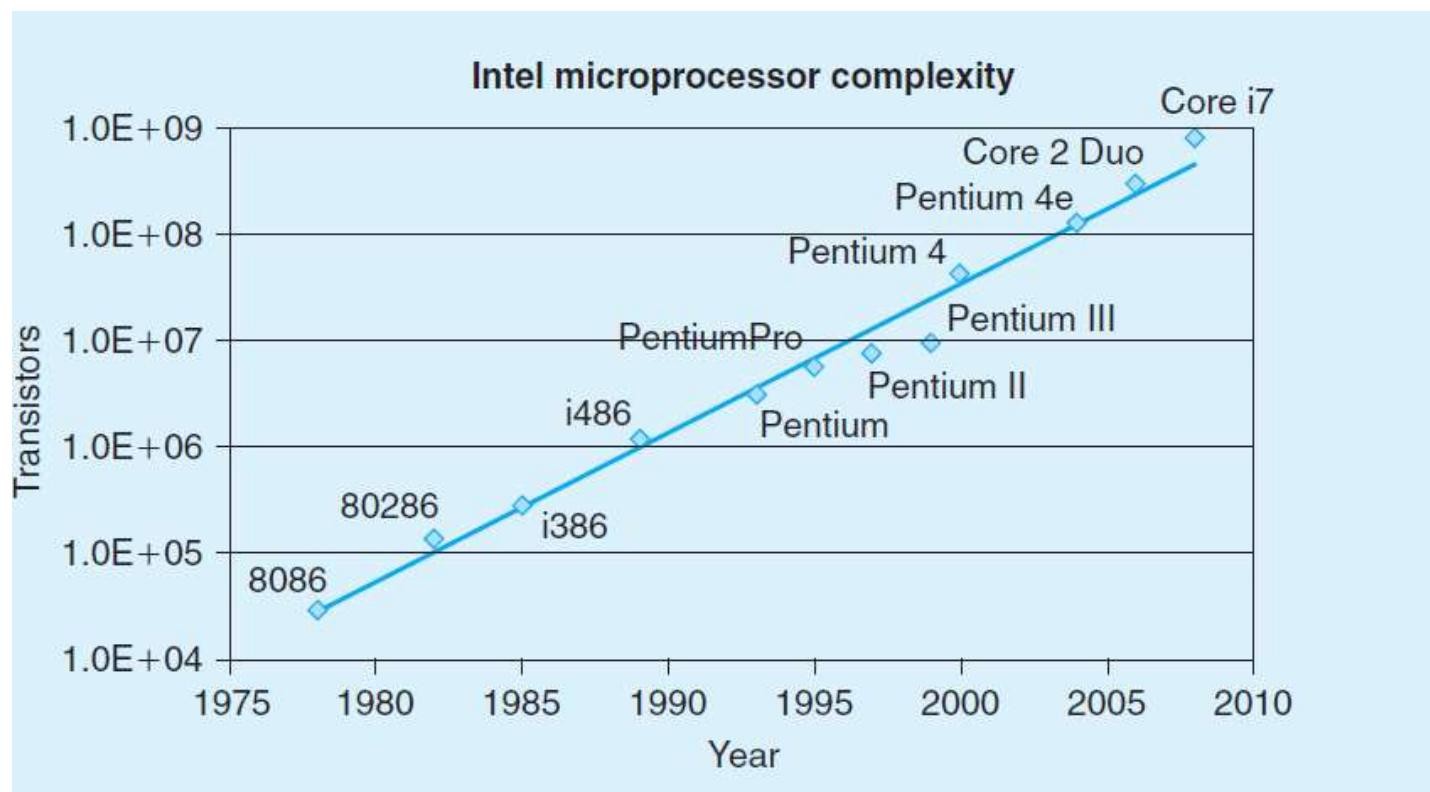
- Backwards compatible up until 8086, introduced in 1978
- Added more features as time goes on

■ Complex instruction set computer (CISC)

- Many different instructions with many different formats
 - But, only small subset encountered with Linux programs
- Hard to match performance of Reduced Instruction Set Computers (RISC)
- But, Intel has done just that!
 - In terms of speed. Less so for low power.

Intel x86 Evolution: Milestones

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
■ 8086	1978	29K	5-10
			<ul style="list-style-type: none">▪ First 16-bit Intel processor. Basis for IBM PC & DOS▪ 1MB address space
■ 386	1985	275K	16-33
			<ul style="list-style-type: none">▪ First 32 bit Intel processor , referred to as IA32▪ Added “flat addressing”, capable of running Unix
■ Pentium 4E	2004	125M	2800-3800
			<ul style="list-style-type: none">▪ First 64-bit Intel x86 processor, referred to as x86-64
■ Core 2	2006	291M	1060-3500
			<ul style="list-style-type: none">▪ First multi-core Intel processor
■ Core i7	2008	731M	1700-3900
			<ul style="list-style-type: none">▪ Four cores (our shark machines)



- **Moore's Law:** In 1965, Gordon Moore, a founder of Intel Corporation, extrapolated from the chip technology of the day, in which they could fabricate circuits with around 64 transistors on a single chip, to predict that the number of transistors per chip would double every year for the next 10 years. Over more than 45 years, the semiconductor industry has been able to double transistor counts on average every 18 months.

x86 Clones: Advanced Micro Devices (AMD)

■ Historically

- AMD has followed just behind Intel
- A little bit slower, a lot cheaper

■ Then

- Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

■ Recent Years

- Intel got its act together
 - Leads the world in semiconductor technology
- AMD has fallen behind
 - Relies on external semiconductor manufacturer

**Update!!!
The scenario has
changed yet again**

Our Primary Focus

- The processor (CPU)...
 - datapath
 - control
- ...implemented using millions of transistors
- ...impossible to understand by looking at individual transistors
- we need...

Abstraction

- Delving into the depths reveals more information, but...
 - An abstraction omits “unneeded” detail, helps us cope with complexity
 - *From the figure on the right, how does abstraction help the programme and how does she avoid too much detail?*

High-level language program (in C)

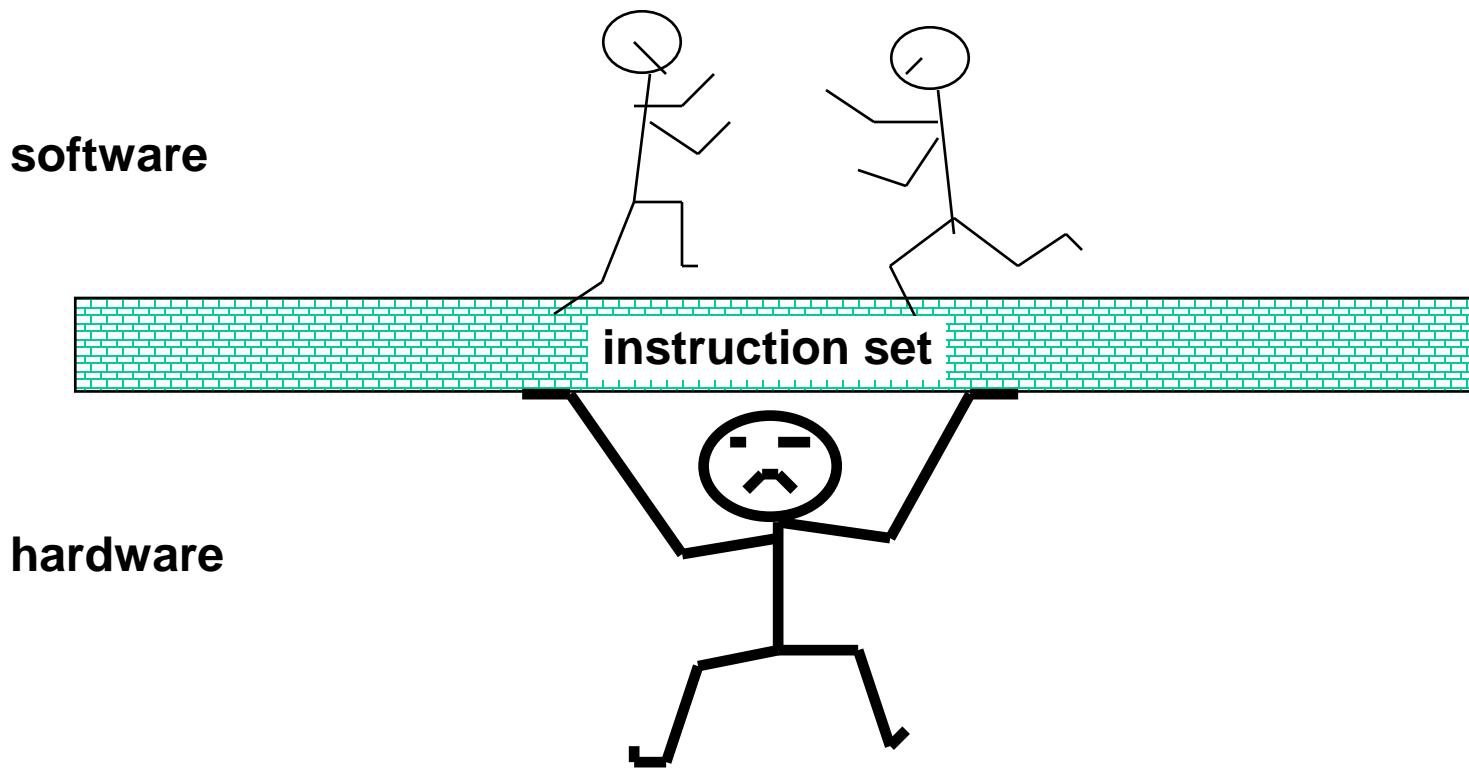
```
swap(int v[], int k)
{int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Assembly language program (for MIPS)

```
swap:  
    muli $2, $5,4  
    add $2, $4,$2  
    lw   $15, 0($2)  
    lw   $16, 4($2)  
    sw   $16, 0($2)  
    sw   $15, 4($2)  
    jr   $31
```

Binary machine language program (for MIPS)

The Instruction Set: a Critical Interface

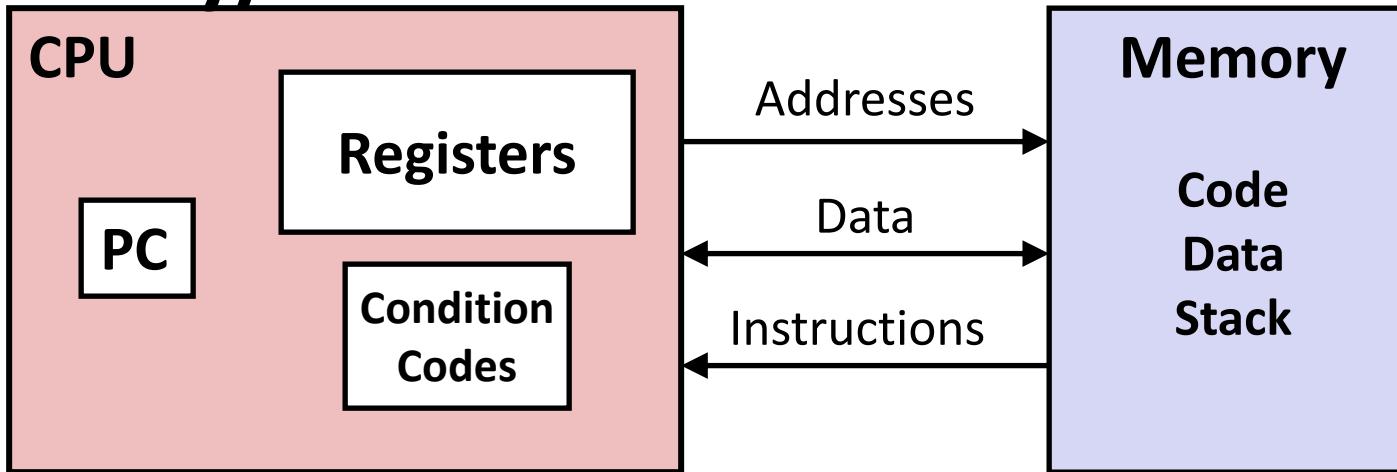


Instruction Set Architecture

- A very important abstraction:
 - *interface* between hardware and low-level software
 - *standardizes* instructions, machine language bit patterns, etc.
 - advantage: *allows different implementations of the same architecture*
- Modern instruction set architectures:
 - x86, IA32, Itanium, x86-64/Pentium/K6, PowerPC, DEC Alpha, MIPS, SPARC, HP, ARM

Microarchitecture: Implementation of the architecture.
Examples: cache sizes and core frequency.

Assembly/Machine Code View

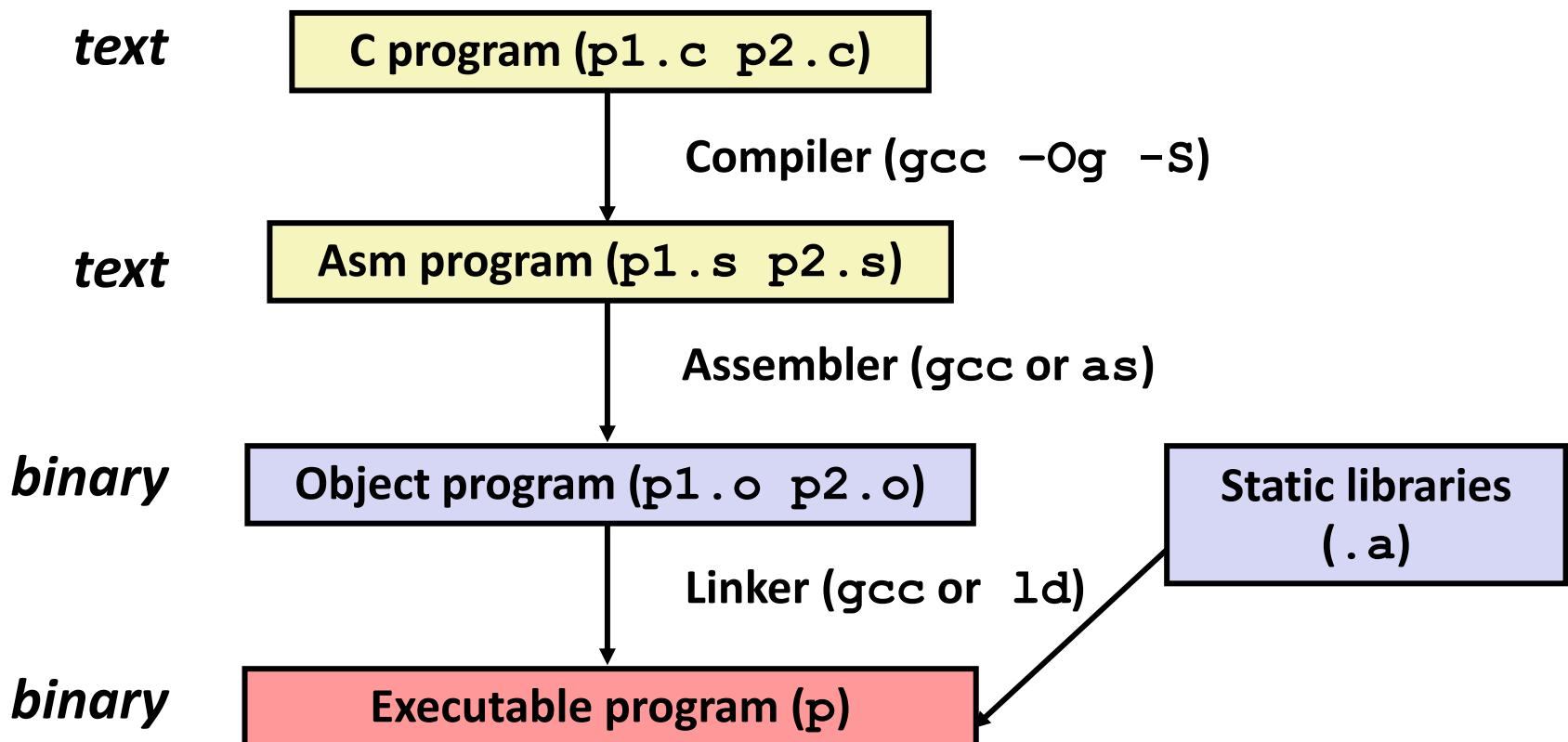


Programmer-Visible State

- **PC: Program counter**
 - Address of next instruction
 - Called %rip - “RIP” (x86-64)
- **Register file**
 - Heavily used program data 16×64 bits
- **Condition codes**
 - Store status information about most recent arithmetic or logical operation
 - Used for conditional branching
- **Vector registers**
- **Memory**
 - Byte addressable array
 - Code and user data
 - Stack to support procedures
 - No distinguishing between different datatypes, int, pointers, arrays etc.

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
 - Use basic optimizations (`-Og`) [New to recent versions of GCC]
 - Put resulting binary in file `p`



Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);  
  
void sumstore(long x, long y,  
              long *dest)  
{  
    long t = plus(x, y);  
    *dest = t;  
}
```

Generated x86-64 Assembly

```
sumstore:  
    pushq   %rbx  
    movq   %rdx, %rbx  
    call    plus  
    movq   %rax, (%rbx)  
    popq   %rbx  
    ret
```

Obtain with command

```
gcc -Og -S sum.c
```

Produces file sum.s

Warning: Will get very different results on different machines (Andrew Linux, Mac OS-X, ...) due to different versions of gcc and different compiler settings.

Object Code

Code for sumstore

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

- Total of 14 bytes
- Each instruction 1, 3, or 5 bytes
- Starts at address 0x0400595

■ Assembler

- Translates .s into .o
- Binary encoding of each instruction
- Nearly-complete image of executable code
- Missing linkages between code in different files

■ Linker

- Resolves references between files
- Combines with static run-time libraries
 - E.g., code for **malloc**, **printf**
- Some libraries are *dynamically linked*
 - Linking occurs when program begins execution

Machine Instruction Example

```
*dest = t;
```

■ C Code

- Store value **t** where designated by **dest**

```
movq %rax, (%rbx)
```

■ Assembly

- Move 8-byte value to memory
 - Quad words in x86-64 parlance
- Operands:
 - t:** Register **%rax**
 - dest:** Register **%rbx**
 - *dest:** Memory **M[%rbx]**

```
0x40059e: 48 89 03
```

■ Object Code

- 3-byte instruction
- Stored at address **0x40059e**

Disassembling Object Code

Disassembled

```
0000000000400595 <sumstore>:  
 400595: 53          push    %rbx  
 400596: 48 89 d3    mov      %rdx,%rbx  
 400599: e8 f2 ff ff ff  callq   400590 <plus>  
 40059e: 48 89 03    mov      %rax,(%rbx)  
 4005a1: 5b          pop     %rbx  
 4005a2: c3          retq
```

■ Disassembler

`objdump -d sum`

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either `a.out` (complete executable) or `.o` file

Alternate Disassembly

Object

0x0400595:

0x53
0x48
0x89
0xd3
0xe8
0xf2
0xff

0xff
0xff

0x48
0x89
0x03
0x5b
0xc3

Disassembled

```
Dump of assembler code for function sumstore:  
0x0000000000400595 <+0>: push    %rbx  
0x0000000000400596 <+1>: mov     %rdx,%rbx  
0x0000000000400599 <+4>: callq   0x400590 <plus>  
0x000000000040059e <+9>: mov     %rax,(%rbx)  
0x00000000004005a1 <+12>:pop    %rbx  
0x00000000004005a2 <+13>:retq
```

■ Within gdb Debugger

gdb sum

disassemble sumstore

- Disassemble procedure

x/14xb sumstore

- Examine the 14 bytes starting at sumstore

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:      file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

Reverse engineering forbidden by
Microsoft End User License Agreement

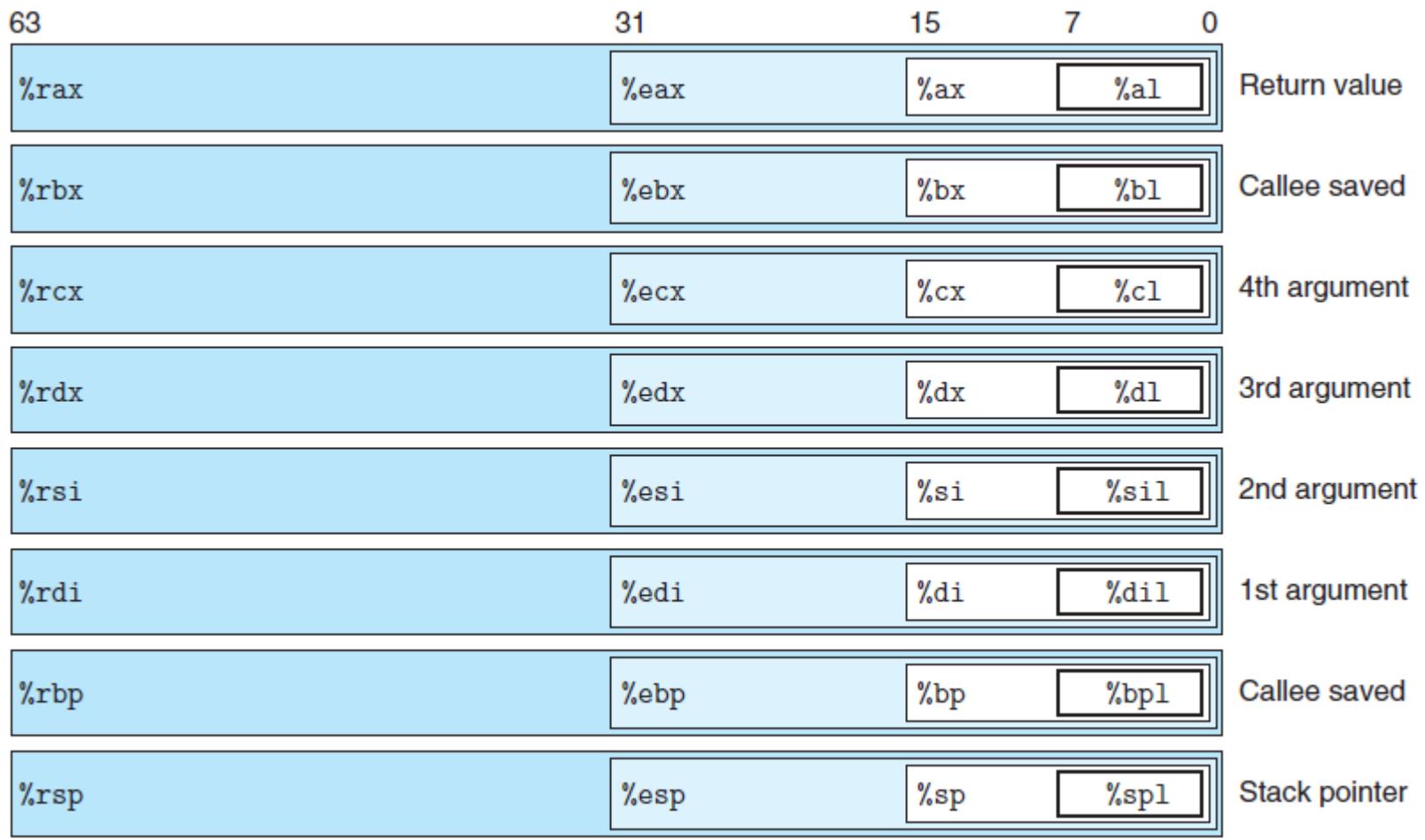
- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

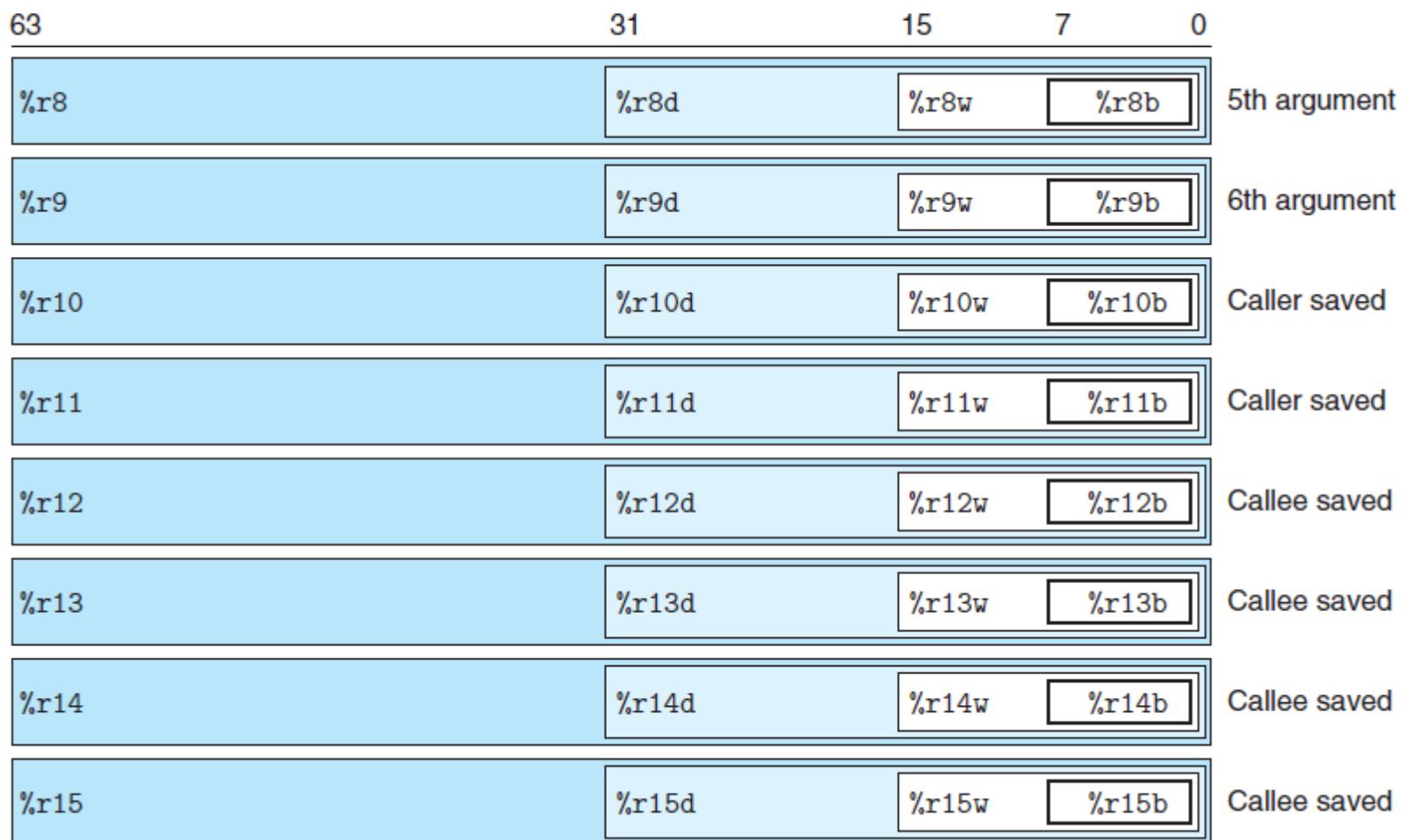
Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Quad word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

Figure 3.1 Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.





Operand Specifiers – Addressing Modes

- Most instructions have one or more *operands* specifying the source values to use
- Operand types: **Immediate, Register, Memory**

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Example

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Fill in the following table showing the values for the indicated operands:

Operand	Value
%rax	_____
0x104	_____
\$0x108	_____
(%rax)	_____
4(%rax)	_____
9(%rax,%rdx)	_____
260(%rcx,%rdx)	_____
0xFC(%rcx,4)	_____
(%rax,%rdx,4)	_____

Example

Operand	Value	Comment
%rax	0x100	Register
0x104	0xAB	Absolute address
\$0x108	0x108	Immediate
(%rax)	0xFF	Address 0x100
4(%rax)	0xAB	Address 0x104
9(%rax,%rdx)	0x11	Address 0x10C
260(%rcx,%rdx)	0x13	Address 0x108
0xFC(,%rcx,4)	0xFF	Address 0x100
(%rax,%rdx,4)	0x11	Address 0x10C

Moving Data

■ Moving Data

`movq Source, Dest:`

■ Operand Types

- **Immediate:** Constant integer data

- Example: `$0x400`, `$-533`
- Like C constant, but prefixed with '\$'
- Encoded with 1, 2, or 4 bytes

- **Register:** One of 16 integer registers

- Example: `%rax`, `%r13`
- But `%rsp` reserved for special use
- Others have special uses for particular instructions

- **Memory:** 8 consecutive bytes of memory at address given by register

- Simplest example: (`%rax`)
- Various other “address modes”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

movq Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	<i>Imm</i>	<i>Reg</i>	movq \$0x4,%rax	temp = 0x4;
		<i>Mem</i>	movq \$-147,(%rax)	*p = -147;
	<i>Reg</i>	<i>Reg</i>	movq %rax,%rdx	temp2 = temp1;
	<i>Reg</i>	<i>Mem</i>	movq %rax,(%rdx)	*p = temp;
	<i>Mem</i>	<i>Reg</i>	movq (%rax),%rdx	temp = *p;

Cannot do memory-memory transfer with a single instruction

- For most cases, the mov instructions will only update the specific register bytes or memory locations indicated by the destination operand.
- **The only exception is that when `movl` has a register as the destination, it will also set the high-order 4 bytes of the register to 0.**
- This exception arises from the convention, adopted in x86-64, that any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0.

movabsq	<i>I, R</i>	<i>R</i> \leftarrow <i>I</i>	Move absolute quad word
----------------	-------------	--------------------------------	-------------------------

This instruction is for dealing with 64-bit immediate data.

The regular **movq** instruction can only have immediate source operands that can be represented as 32-bit two's-complement numbers. This value is then sign extended to produce the 64-bit value for the destination.

The **movabsq** instruction can have an arbitrary 64-bit immediate value as its source operand and can only have a register as a destination.

Aside Understanding how data movement changes a destination register

As described, there are two different conventions regarding whether and how data movement instructions modify the upper bytes of a destination register. This distinction is illustrated by the following code sequence:

```
1  movabsq $0x0011223344556677, %rax    %rax = 0011223344556677  
2  movb    $-1, %al                      %rax = 00112233445566FF  
3  movw    $-1, %ax                      %rax = 001122334455FFFF  
4  movl    $-1, %eax                     %rax = 00000000FFFFFFFF  
5  movq    $-1, %rax                      %rax = FFFFFFFFFFFFFF
```

In the following discussion, we use hexadecimal notation. In the example, the instruction on line 1 initializes register %rax to the pattern 0011223344556677. The remaining instructions have immediate value -1 as their source values. Recall that the hexadecimal representation of -1 is of the form FF \cdots F, where the number of F's is twice the number of bytes in the representation. The `movb` instruction (line 2) therefore sets the low-order byte of %rax to FF, while the `movw` instruction (line 3) sets the low-order 2 bytes to FFFF, with the remaining bytes unchanged. The `movl` instruction (line 4) sets the low-order 4 bytes to FFFFFFFF, but it also sets the high-order 4 bytes to 00000000. Finally, the `movq` instruction (line 5) sets the complete register to FFFFFFFFFFFFFF.

Instruction	Effect	Description
MOVZ S, R	$R \leftarrow \text{ZeroExtend}(S)$	Move with zero extension
movzbw		Move zero-extended byte to word
movzbl		Move zero-extended byte to double word
movzwl		Move zero-extended word to double word
movzbq		Move zero-extended byte to quad word
movzwq		Move zero-extended word to quad word

Figure 3.5 Zero-extending data movement instructions. These instructions have a register or memory location as the source and a register as the destination.

Instruction	Effect	Description
MOVS S, R	$R \leftarrow \text{SignExtend}(S)$	Move with sign extension
movsbw		Move sign-extended byte to word
movsbl		Move sign-extended byte to double word
movswl		Move sign-extended word to double word
movsbq		Move sign-extended byte to quad word
movswq		Move sign-extended word to quad word
movslq		Move sign-extended double word to quad word
cltq	$\%rax \leftarrow \text{SignExtend}(\%eax)$	Sign-extend %eax to %rax

Figure 3.6 Sign-extending data movement instructions. The MOVS instructions have a register or memory location as the source and a register as the destination. The CLTQ instruction is specific to registers %eax and %rax.

Practice Problem

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands. (For example, mov can be rewritten as movb, movw, movl, or movq.)

```
mov____ %eax, (%rsp)
mov____ (%rax), %dx
mov____ $0xFF, %bl
mov____ (%rsp,%rdx,4), %dl
mov____ (%rdx), %rax
mov____ %dx, (%rax)
```

Practice Problem

Each of the following lines of code generates an error message when we invoke the assembler. Explain what is wrong with each line.

```
movb $0xF, (%ebx)
movl %rax, (%rsp)
movw (%rax),4(%rsp)
movb %al,%sl
movq %rax,$0x123
movl %eax,%rdx
movb %si, 8(%rbp)
```

Example

(b) Assembly code

```
long exchange(long *xp, long y)
xp in %rdi, y in %rsi
1 exchange:
2     movq    (%rdi), %rax      Get x at xp. Set as return value.
3     movq    %rsi, (%rdi)      Store y at xp.
4     ret                      Return.
```

Figure 3.7 C and assembly code for exchange routine. Registers %rdi and %rsi hold parameters xp and y, respectively.

Example

(a) C code

```
long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

(b) Assembly code

```
long exchange(long *xp, long y)
xp in %rdi, y in %rsi
1 exchange:
2     movq    (%rdi), %rax      Get x at xp. Set as return value.
3     movq    %rsi, (%rdi)      Store y at xp.
4     ret                      Return.
```

Figure 3.7 C and assembly code for exchange routine. Registers %rdi and %rsi hold parameters xp and y, respectively.

Pushing and Popping Stack Data

Instruction	Effect	Description
pushq S	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	Push quad word
popq D	$D \leftarrow M[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	Pop quad word

Figure 3.8 Push and pop

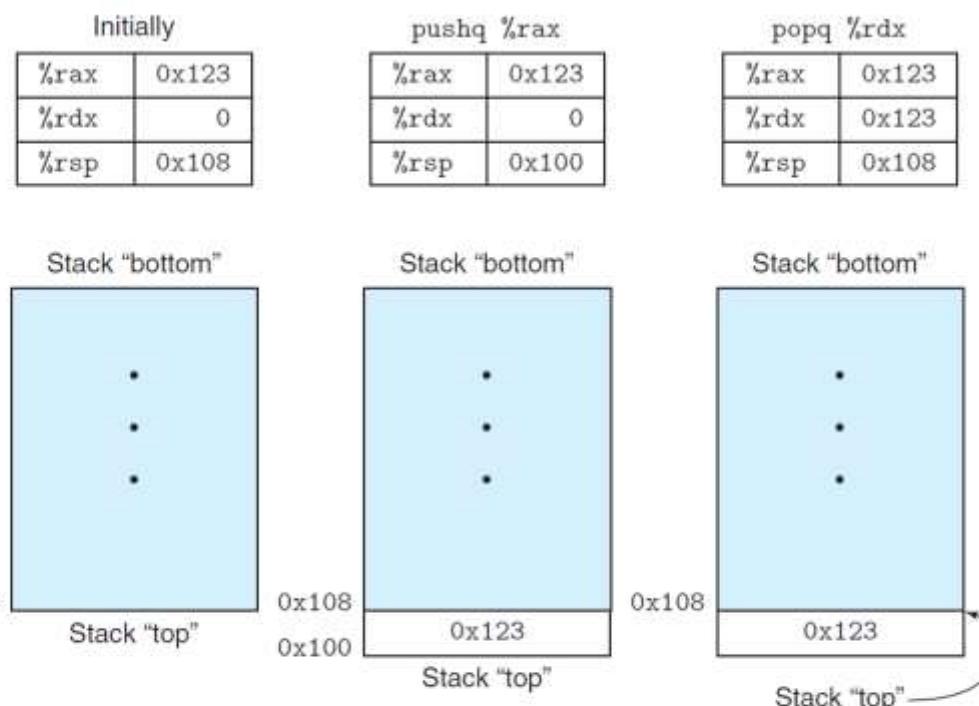


Figure 3.9 Illustration of stack operation. By convention, we draw stacks upside down, so that the “top” of the stack is shown at the bottom. With x86-64, stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register %rsp) and storing to memory, while popping involves reading from memory and incrementing the stack pointer.

Example of Simple Addressing Modes

swap:

```
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

Example of Simple Addressing Modes

```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

swap:

movq	(%rdi), %rax
movq	(%rsi), %rdx
movq	%rdx, (%rdi)
movq	%rax, (%rsi)
ret	

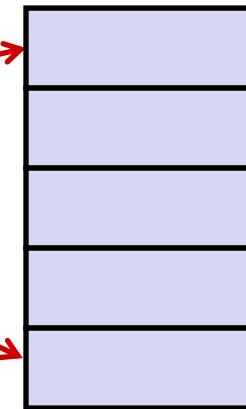
Understanding Swap()

```
void swap
    (long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Memory

Registers

%rdi	
%rsi	
%rax	
%rdx	



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

swap:

```
        movq    (%rdi), %rax    # t0 = *xp
        movq    (%rsi), %rdx    # t1 = *yp
        movq    %rdx, (%rdi)    # *xp = t1
        movq    %rax, (%rsi)    # *yp = t0
        ret
```

Understanding Swap()

Registers

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

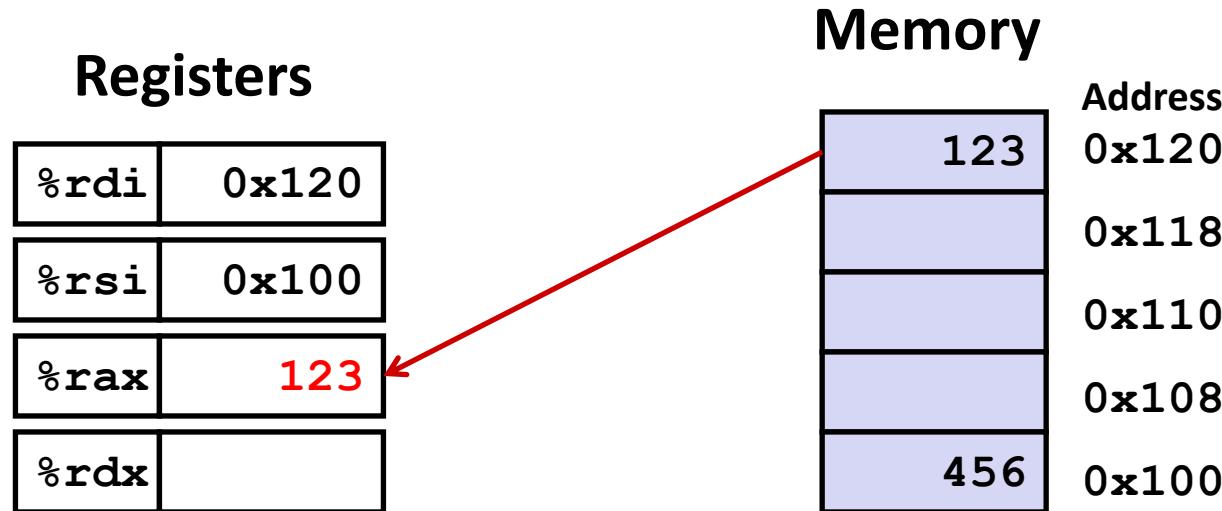
Memory

123	Address 0x120
	0x118
	0x110
	0x108
456	0x100

swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

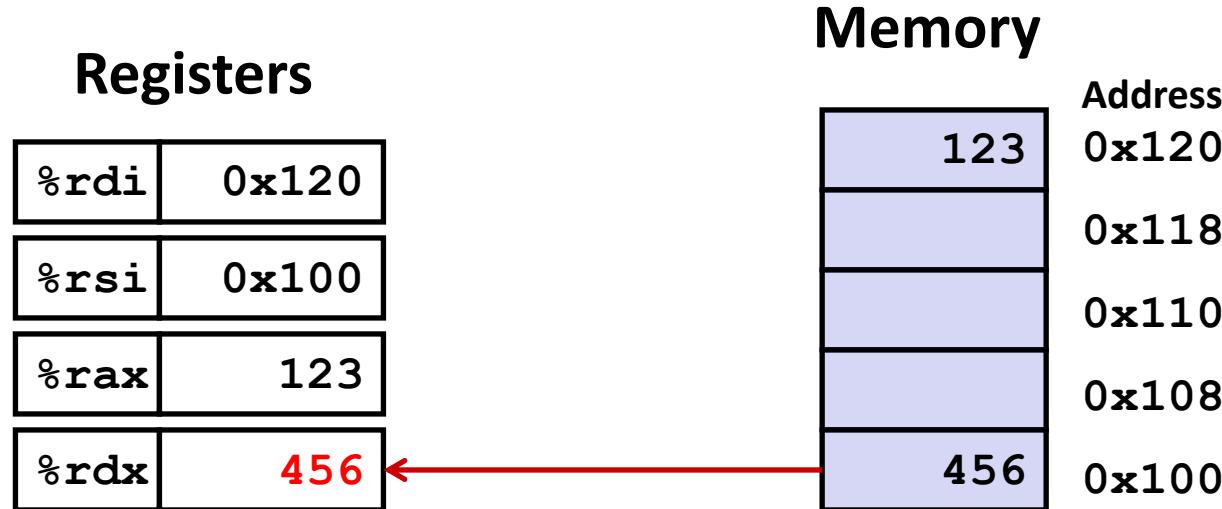
Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Understanding Swap()

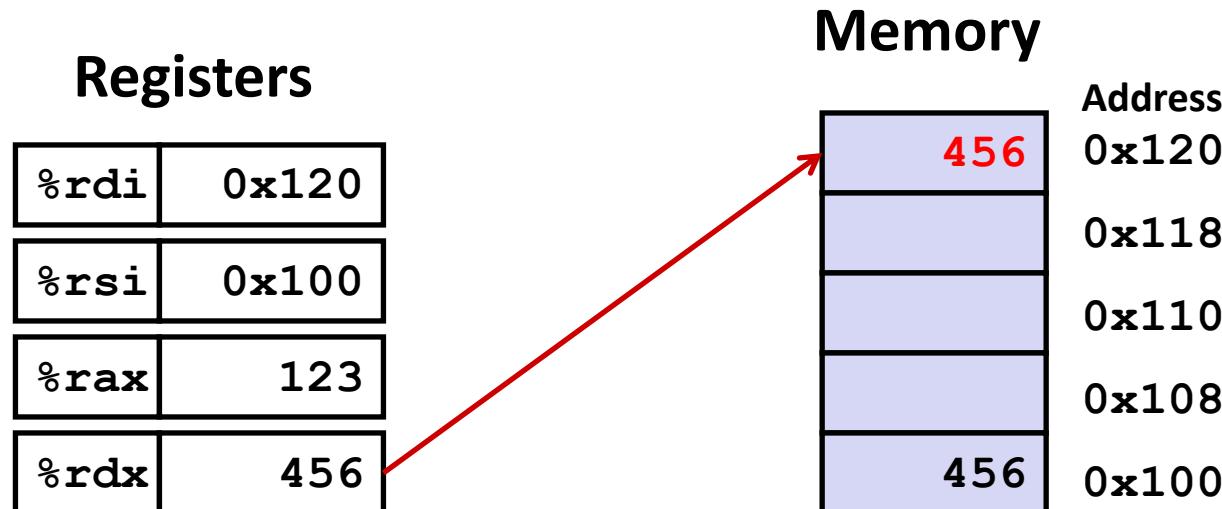


swap:

```

    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
  
```

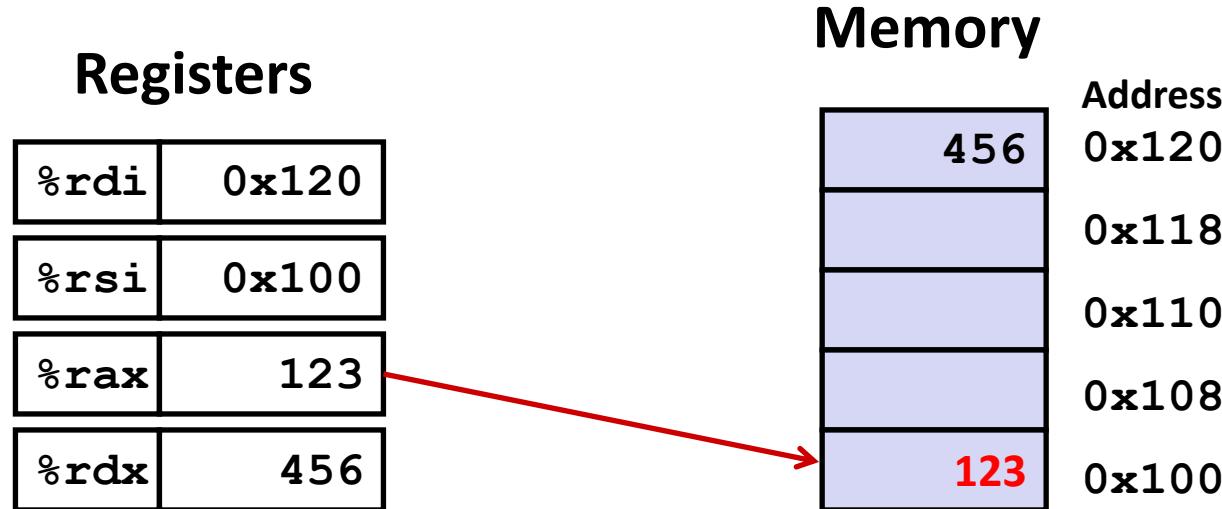
Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Understanding Swap()



swap:

```
    movq    (%rdi), %rax    # t0 = *xp
    movq    (%rsi), %rdx    # t1 = *yp
    movq    %rdx, (%rdi)    # *xp = t1
    movq    %rax, (%rsi)    # *yp = t0
    ret
```

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations

Address Computation Instruction

■ **leaq Src, Dst**

- *Src* is address mode expression
- Set *Dst* (*must be a register*) to address (**Effective address**) denoted by expression

leaq S,D //D \leftarrow &S; Load effective address

■ **Uses**

- Computing addresses without a memory reference
 - E.g., translation of $p = \&x[i];$
- Computing arithmetic expressions of the form $x + k*y$
 - $k = 1, 2, 4, \text{ or } 8$

```
long m12(long x)
{
    return x*12;
}
```

Compilers often find clever uses of leaq that have nothing to do with effective address computations. The destination operand must be a register.

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax           # return t<<2
```

Some Arithmetic Operations

■ One Operand Instructions

incq	<i>Dest</i>	$Dest = Dest + 1$
decq	<i>Dest</i>	$Dest = Dest - 1$
negq	<i>Dest</i>	$Dest = -Dest$
notq	<i>Dest</i>	$Dest = \sim Dest$

■ See book for more instructions

Some Arithmetic Operations

■ Two Operand Instructions:

<i>Format</i>	<i>Computation</i>		
addq	<i>Src,Dest</i>	Dest = Dest + Src	
subq	<i>Src,Dest</i>	Dest = Dest – Src	
imulq	<i>Src,Dest</i>	Dest = Dest * Src	
salq	<i>Src,Dest</i>	Dest = Dest << Src	<i>Also called shlq</i>
sarq	<i>Src,Dest</i>	Dest = Dest >> Src	<i>Arithmetic</i>
shrq	<i>Src,Dest</i>	Dest = Dest >> Src	<i>Logical</i>
xorq	<i>Src,Dest</i>	Dest = Dest ^ Src	
andq	<i>Src,Dest</i>	Dest = Dest & Src	
orq	<i>Src,Dest</i>	Dest = Dest Src	

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)

Shift

Both arithmetic and logical right shifts are possible. The different shift instructions can specify the shift amount either as an immediate value or with the single-byte register %cl. (These instructions are unusual in only allowing this specific register as the operand.)

With x86-64, a shift instruction operating on data values that are w bits long determines the shift amount from the low-order m bits of register %cl, where $2m = w$. The higher-order bits are ignored. So, for example, when register %cl has hexadecimal value 0xFF, then instruction salb would shift by 7, while salw would shift by 15, sall would shift by 31, and salq would shift by 63.

Arithmetic Expression Example

(a) C code

```
long arith(long x, long y, long z)
{
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0x0F0F0F0F;
    long t4 = t2 - t3;
    return t4;
}
```

(b) Assembly code

```
long arith(long x, long y, long z)
x in %rdi, y in %rsi, z in %rdx
1    arith:
2        xorq    %rsi, %rdi          t1 = x ^ y
3        leaq    (%rdx,%rdx,2), %rax   3*z
4        salq    $4, %rax           t2 = 16 * (3*z) = 48*z
5        andl    $252645135, %edi      t3 = t1 & 0x0F0F0F0F
6        subq    %rdi, %rax          Return t2 - t3
7        ret
```

Figure 3.11 C and assembly code for arithmetic function.

Special Arithmetic Operations

Multiplying two 64-bit signed or unsigned integers can yield a product that requires 128 bits to represent. The x86-64 instruction set provides limited support for operations involving 128-bit (16-byte) numbers.

The `imulq` instruction has two different forms. One form, serves as a “two operand” multiply instruction, generating a 64-bit product from two 64-bit operands. The other version is given below:

Instruction	Effect	Description
<code>imulq S</code>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Signed full multiply
<code>mulq S</code>	$R[\%rdx]:R[\%rax] \leftarrow S \times R[\%rax]$	Unsigned full multiply
<code>cqto</code>	$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Convert to oct word
<code>idivq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Signed divide
<code>divq S</code>	$R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \bmod S;$ $R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S$	Unsigned divide

Figure 3.12 Special arithmetic operations. These operations provide full 128-bit multiplication and division, for both signed and unsigned numbers. The pair of registers `%rdx` and `%rax` are viewed as forming a single 128-bit oct word.

Why separate instructions for signed multiplication and division?

Addition and subtraction are the same, as is the low-half of a multiply. A full multiply, however, is not. Simple example:

In 32-bit twos-complement, -1 has the same representation as the unsigned quantity $2^{32} - 1$. However:

$$\begin{aligned}-1 * -1 &= +1 \\(2^{32} - 1) * (2^{32} - 1) &= (2^{64} - 2^{33} + 1)\end{aligned}$$

Machine Programming I: Summary

- **History of Intel processors and architectures**
 - Evolutionary design leads to many quirks and artifacts
- **C, assembly, machine code**
 - New forms of visible state: program counter, registers, ...
 - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- **Assembly Basics: Registers, operands, move**
 - The x86-64 move instructions cover wide range of data movement forms
- **Arithmetic**
 - C compiler will figure out different instruction combinations to carry out computation

Machine-Level Programming II: Control

Instructors:

Dr. R. Shathanaa

Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

Processor State (x86-64, Partial)

■ Information about currently executing program

- Temporary data (`%rax`, ...)
- Location of runtime stack (`%rsp`)
- Location of current code control point (`%rip`, ...)
- Status of recent tests (`CF`, `ZF`, `SF`, `OF`)

Current stack top

Registers

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

`%rip`

Instruction pointer

CF

ZF

SF

OF

Condition codes

Condition Codes (Implicit Setting)

■ Single bit registers

- CF Carry Flag (for unsigned) SF Sign Flag (for signed)
- ZF Zero Flag OF Overflow Flag (for signed)

■ Implicitly set (think of it as side effect) by arithmetic operations

Example: `addq Src,Dest` $\leftrightarrow t = a+b$

CF set if carry out from most significant bit (unsigned overflow)

ZF set if $t == 0$

SF set if $t < 0$ (as signed)

OF set if two's-complement (signed) overflow

$(a>0 \ \&\& \ b>0 \ \&\& \ t<0) \ || \ (a<0 \ \&\& \ b<0 \ \&\& \ t>=0)$

■ Not set by `leaq` instruction

Condition Codes (Explicit Setting: Compare)

■ Explicit Setting by Compare Instruction

- **cmpq Src2, Src1**
- **cmpq b, a** like computing $a - b$ without setting destination
- **CF set** if carry out from most significant bit (used for unsigned comparisons)
- **ZF set** if $a == b$
- **SF set** if $(a - b) < 0$ (as signed)
- **OF set** if two's-complement (signed) overflow
$$(a>0 \ \&\& \ b<0 \ \&\& \ (a-b)<0) \ \|\ (a<0 \ \&\& \ b>0 \ \&\& \ (a-b)>0)$$

Condition Codes (Explicit Setting: Test)

■ Explicit Setting by Test instruction

- **testq Src2, Src1**

 - **testq b, a** like computing **a&b** without setting destination

- Sets condition codes based on value of *Src1* & *Src2*

- Useful to have one of the operands be a mask

- **ZF set** when **a&b == 0**

- **SF set** when **a&b < 0**

Instruction	Based on	Description
CMP S_1, S_2	$S_2 - S_1$	Compare
cmpb		Compare byte
cmpw		Compare word
cmpl		Compare double word
cmpq		Compare quad word
TEST S_1, S_2	$S_1 \& S_2$	Test
testb		Test byte
testw		Test word
testl		Test double word
testq		Test quad word

Reading Condition Codes

■ SetX Instructions

- Set low-order byte of destination to 0 or 1 based on combinations of condition codes
- Does not alter remaining 7 bytes

Instruction	Synonym	Effect	Set condition
sete <i>D</i>	setz	$D \leftarrow ZF$	Equal / zero
setne <i>D</i>	setnz	$D \leftarrow \sim ZF$	Not equal / not zero
sets <i>D</i>		$D \leftarrow SF$	Negative
setns <i>D</i>		$D \leftarrow \sim SF$	Nonnegative
setg <i>D</i>	setnle	$D \leftarrow \sim (SF \wedge OF) \& \sim ZF$	Greater (signed >)
setge <i>D</i>	setnl	$D \leftarrow \sim (SF \wedge OF)$	Greater or equal (signed \geq)
setl <i>D</i>	setnge	$D \leftarrow SF \wedge OF$	Less (signed <)
setle <i>D</i>	setng	$D \leftarrow (SF \wedge OF) \mid ZF$	Less or equal (signed \leq)
seta <i>D</i>	setnbe	$D \leftarrow \sim CF \& \sim ZF$	Above (unsigned >)
setae <i>D</i>	setnb	$D \leftarrow \sim CF$	Above or equal (unsigned \geq)
setb <i>D</i>	setnae	$D \leftarrow CF$	Below (unsigned <)
setbe <i>D</i>	setna	$D \leftarrow CF \mid ZF$	Below or equal (unsigned \leq)

x86-64 Integer Registers

%rax	%al	%r8	%r8b
%rbx	%bl	%r9	%r9b
%rcx	%cl	%r10	%r10b
%rdx	%dl	%r11	%r11b
%rsi	%sil	%r12	%r12b
%rdi	%dil	%r13	%r13b
%rsp	%spl	%r14	%r14b
%rbp	%bpl	%r15	%r15b

- Can reference low-order byte

Reading Condition Codes (Cont.)

■ SetX Instructions:

- Set single byte based on combination of condition codes

■ One of addressable byte registers

- Does not alter remaining bytes
- Typically use `movzbl` to finish job
 - 32-bit instructions also set upper 32 bits to 0

```
int gt (long x, long y)
{
    return x > y;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
cmpq    %rsi, %rdi    # Compare x:y
setg    %al             # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

Jumping

■ jX Instructions : Jump to different part of code depending on condition codes

Instruction	Synonym	Jump condition	Description
jmp <i>Label</i>		1	Direct jump
jmp <i>*Operand</i>		1	Indirect jump
je <i>Label</i>	jz	ZF	Equal / zero jmp *%rax or jmp *(%rax)
jne <i>Label</i>	jnz	~ZF	Not equal / not zero
js <i>Label</i>		SF	Negative
jns <i>Label</i>		~SF	Nonnegative
jg <i>Label</i>	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge <i>Label</i>	jnl	~(SF ^ OF)	Greater or equal (signed >=)
jl <i>Label</i>	jnge	SF ^ OF	Less (signed <)
jle <i>Label</i>	jng	(SF ^ OF) ZF	Less or equal (signed <=)
ja <i>Label</i>	jnbe	~CF & ~ZF	Above (unsigned >)
jae <i>Label</i>	jnb	~CF	Above or equal (unsigned >=)
jb <i>Label</i>	jnae	CF	Below (unsigned <)
jbe <i>Label</i>	jna	CF ZF	Below or equal (unsigned <=)

Conditional Branch Example (Old Style)

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

absdiff:

cmpq jle movq subq ret	%rsi, %rdi # x:y .L4 %rdi, %rax %rsi, %rax
.L4:	# x <= y
movq subq ret	%rsi, %rax %rdi, %rax

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Expressing with Goto Code

- C allows goto statement
- Jump to position designated by label

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j
    (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

General Conditional Expression Translation (Using Branches)

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
ntest = !Test;  
if (ntest) goto Else;  
val = Then_Expr;  
goto Done;  
Else:  
    val = Else_Expr;  
Done:  
    . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves

■ Conditional Move Instructions

- Instruction supports:
if (Test) Dest \leftarrow Src
- Supported in post-1995 x86 processors
- GCC tries to use them
 - But, only when known to be safe

■ Why?

- Branches are very disruptive to instruction flow through pipelines
- Conditional moves do not require control transfer

C Code

```
val = Test  
? Then_Expr  
: Else_Expr;
```

Goto Version

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```

Conditional Move Example

```
long absdiff
    (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

absdiff:

```

        movq    %rdi, %rax  # x
        subq    %rsi, %rax  # result = x-y
        movq    %rsi, %rdx
        subq    %rdi, %rdx  # eval = y-x
        cmpq    %rsi, %rdi  # x:y
        cmovle %rdx, %rax  # if <=, result = eval
        ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

“Do-While” Loop Example

C Code

```
long pcount_do
(unsigned long x) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

- Count number of 1's in argument **x** (“popcount”)
- Use conditional branch to either continue looping or to exit loop

“Do-While” Loop Compilation

Goto Version

```
long pcount_goto
(unsigned long x) {
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
        movl    $0, %eax      # result = 0
.L2:                           # loop:
        movq    %rdi, %rdx
        andl    $1, %edx      # t = x & 0x1
        addq    %rdx, %rax    # result += t
        shrq    $1, %rdi      # x >>= 1
        jne     .L2          # if (x) goto loop
        rep; ret
```

General “Do-While” Translation

C Code

```
do  
  Body  
  while ( Test );
```

Goto Version

```
loop:  
  Body  
  if ( Test )  
    goto loop
```

■ **Body:** {
 Statement₁;
 Statement₂;
 ...
 Statement_n;
}

General “While” Translation #1

- “Jump-to-middle” translation
- Used with -Og

While version

```
while ( Test)  
    Body
```



Goto Version

```
goto test;  
loop:  
    Body  
test:  
    if ( Test)  
        goto loop;  
done:
```

While Loop Example #1

C Code

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Jump to Middle

```
long pcount_goto_jtm
(unsigned long x) {
    long result = 0;
    goto test;
loop:
    result += x & 0x1;
    x >>= 1;
test:
    if(x) goto loop;
    return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

General “While” Translation #2

While version

```
while ( Test)  
    Body
```

- “Do-while” conversion
- Used with -O1

Do-While Version

```
if ( ! Test)  
    goto done;  
do  
    Body  
    while( Test );  
done:
```

Goto Version

```
if ( ! Test)  
    goto done;  
loop:  
    Body  
    if ( Test )  
        goto loop;  
done:
```

While Loop Example #2

C Code

```
long pcount_while
(unsigned long x) {
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

Do-While Version

```
long pcount_goto_dw
(unsigned long x) {
    long result = 0;
    if (!x) goto done;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
done:
    return result;
}
```

- Compare to do-while version of function
- Initial conditional guards entrance to loop

“For” Loop Form

General Form

```
for (Init; Test; Update)  
    Body
```

```
#define WSIZE 8*sizeof(int)  
long pcount_for  
(unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    for (i = 0; i < WSIZE; i++)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
    }  
    return result;  
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

“For” Loop → While Loop

For Version

```
for (Init; Test; Update)  
    Body
```



While Version

```
Init;  
  
while (Test) {  
    Body  
    Update;  
}
```

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

```
long pcount_for_while  
(unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    i = 0;  
    while (i < WSIZE)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
        i++;  
    }  
    return result;  
}
```

“For” Loop Do-While Conversion

Goto Version C Code

```
long pcount_for
(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

- Initial test can be optimized away

```
long pcount_for_goto_dw
(unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (! (i < WSIZE)) Init
        goto done; ! Test
loop:
{
    unsigned bit =
        (x >> i) & 0x1; Body
    result += bit;
}
Update
if (i < WSIZE)
    goto loop; Test
done:
    return result;
}
```

Today

- Control: Condition codes
- Conditional branches
- Loops
- Switch Statements

```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
        case 1:
            w = y*z;
            break;
        case 2:
            w = y/z;
            /* Fall Through */
        case 3:
            w += z;
            break;
        case 5:
        case 6:
            w -= z;
            break;
        default:
            w = 2;
    }
    return w;
}
```

Switch Statement Example

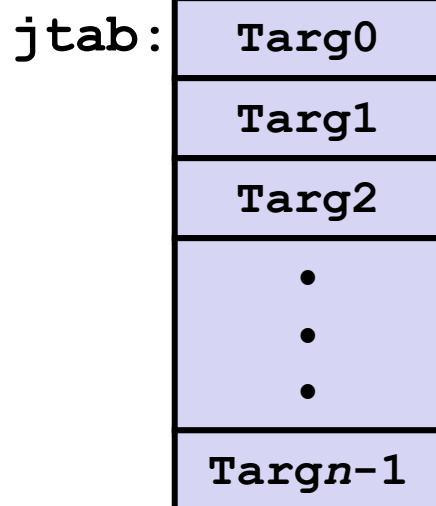
- **Multiple case labels**
 - Here: 5 & 6
- **Fall through cases**
 - Here: 2
- **Missing cases**
 - Here: 4

Jump Table Structure

Switch Form

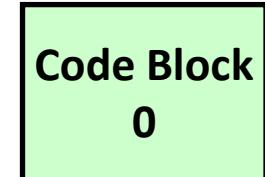
```
switch(x) {
    case val_0:
        Block 0
    case val_1:
        Block 1
    • • •
    case val_{n-1}:
        Block n-1
}
```

Jump Table

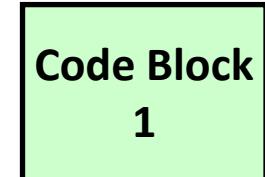


Jump Targets

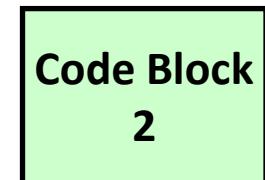
Targ0:



Targ1:

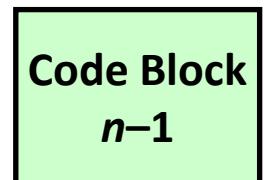


Targ2:



•
•
•

Targ{n-1}:



Translation (Extended C)

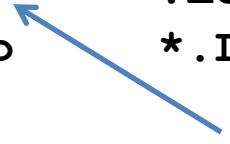
```
goto *JTab[x];
```

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi    # x:6
    ja     .L8
    jmp    * .L4(,%rdi,8)
```



**What range of values
takes default?**

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Note that **w** not
initialized here

Switch Statement Example

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
        . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi      # x:6
    ja     .L8           # Use default
    Indirect jump → jmp    * .L4(,%rdi,8) # goto *JTab[x]
```

Jump table

```
.section  .rodata
.align 8
.L4:
    .quad   .L8  # x = 0
    .quad   .L3  # x = 1
    .quad   .L5  # x = 2
    .quad   .L9  # x = 3
    .quad   .L8  # x = 4
    .quad   .L7  # x = 5
    .quad   .L7  # x = 6
```

Assembly Setup Explanation

■ Table Structure

- Each target requires 8 bytes
- Base address at `.L4`

■ Jumping

- **Direct:** `jmp .L8`
- Jump target is denoted by label `.L8`

- **Indirect:** `jmp * .L4(,%rdi,8)`
- Start of jump table: `.L4`
- Must scale by factor of 8 (addresses are 8 bytes)
- Fetch target from effective Address `.L4 + x*8`
 - Only for $0 \leq x \leq 6$

Jump table

```
.section    .rodata
.align 8
.L4:
.quad      .L8    # x = 0
.quad      .L3    # x = 1
.quad      .L5    # x = 2
.quad      .L9    # x = 3
.quad      .L8    # x = 4
.quad      .L7    # x = 5
.quad      .L7    # x = 6
```

Jump Table

Jump table

```
.section    .rodata
.align 8
.L4:
.quad      .L8  # x = 0
.quad      .L3  # x = 1
.quad      .L5  # x = 2
.quad      .L9  # x = 3
.quad      .L8  # x = 4
.quad      .L7  # x = 5
.quad      .L7  # x = 6
```

```
switch(x) {
    case 1:          // .L3
        w = y*z;
        break;
    case 2:          // .L5
        w = y/z;
        /* Fall Through */
    case 3:          // .L9
        w += z;
        break;
    case 5:
    case 6:          // .L7
        w -= z;
        break;
    default:         // .L8
        w = 2;
}
```

Code Blocks ($x == 1$)

```
switch(x) {  
    case 1:          // .L3  
        w = y*z;  
        break;  
    . . .  
}
```

```
.L3:  
    movq    %rsi, %rax  # y  
    imulq   %rdx, %rax  # y*z  
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Handling Fall-Through

```
long w = 1;  
.  
.  
switch(x) {  
.  
.case 2:  
    w = y/z;  
    /* Fall Through */  
case 3:  
    w += z;  
    break;  
.  
.  
}
```

```
case 2:  
    w = y/z;  
    goto merge;
```

```
case 3:  
    w = 1;  
  
merge:  
    w += z;
```

Code Blocks ($x == 2$, $x == 3$)

```

long w = 1;
. . .
switch(x) {
. . .
case 2:
    w = y/z;
    /* Fall Through */
case 3:
    w += z;
    break;
. . .
}

```

```

.L5:                                # Case 2
    movq    %rsi, %rax
    cqto
    idivq   %rcx          # y/z
    jmp     .L6            # goto merge
.L9:                                # Case 3
    movl    $1, %eax        # w = 1
.L6:                                # merge:
    addq    %rcx, %rax    # w += z
    ret

```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Code Blocks ($x == 5$, $x == 6$, default)

```
switch(x) {
    . . .
    case 5: // .L7
    case 6: // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

```
.L7:                      # Case 5,6
    movl $1, %eax      # w = 1
    subq %rdx, %rax   # w -= z
    ret
.L8:                      # Default:
    movl $2, %eax      # 2
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Summarizing

■ C Control

- if-then-else
- do-while
- while, for
- switch

■ Assembler Control

- Conditional jump
- Conditional move
- Indirect jump (via jump tables)
- Compiler generates code sequence to implement more complex control

■ Standard Techniques

- Loops converted to do-while or jump-to-middle form
- Large switch statements use jump tables
- Sparse switch statements may use decision trees (if-elseif-elseif-else)

Summary

■ Today

- Control: Condition codes
- Conditional branches & conditional moves
- Loops
- Switch statements

■ Next Time

- Stack
- Call / return
- Procedure call discipline

Machine-Level Programming III: Procedures

Instructor:

R. Shathanaa

Mechanisms in Procedures

■ Passing control

- To beginning of procedure code
- Back to return point

■ Passing data

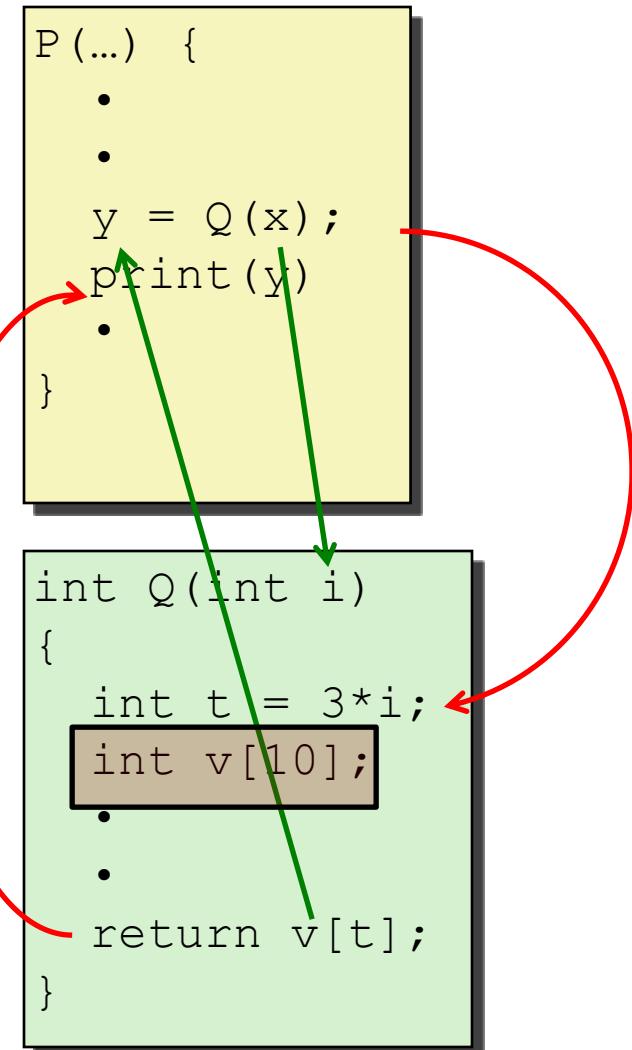
- Procedure arguments
- Return value

■ Memory management

- Allocate during procedure execution
- Deallocate upon return

■ Mechanisms all implemented with machine instructions

■ x86-64 implementation of a procedure uses only those mechanisms required



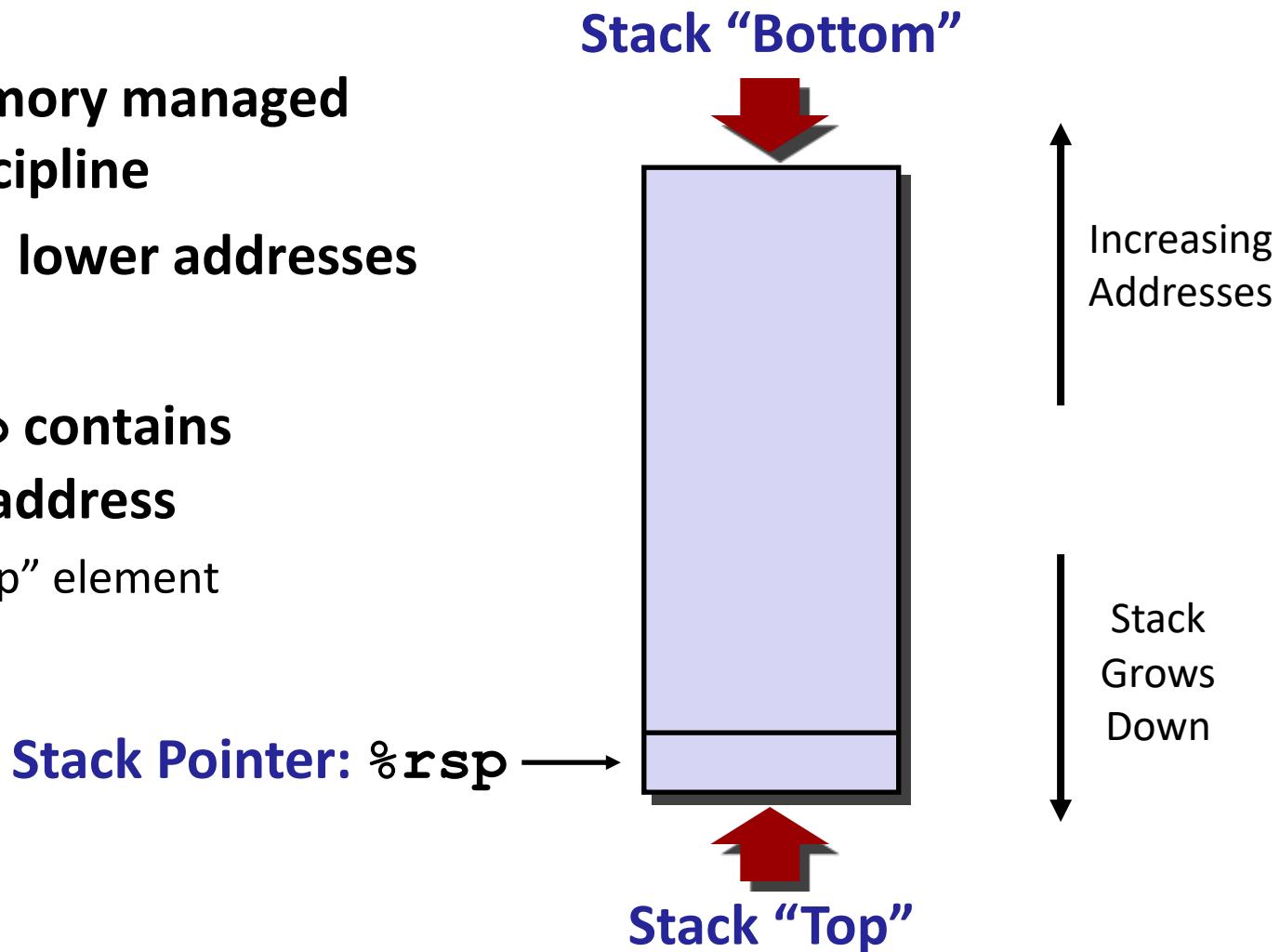
Today

■ Procedures

- Stack Structure
- Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
- Illustration of Recursion

x86-64 Stack

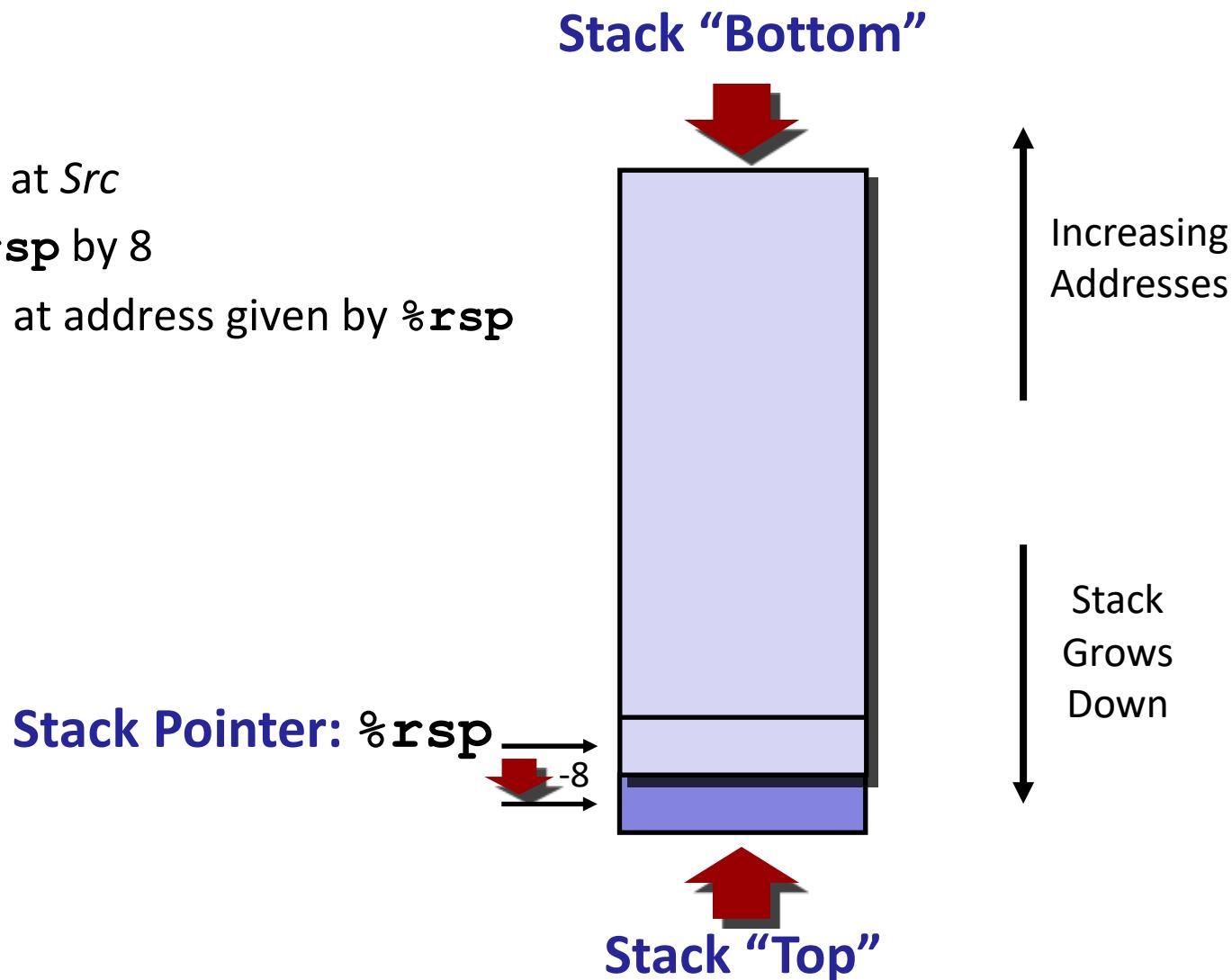
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address
 - address of “top” element



x86-64 Stack: Push

■ **pushq Src**

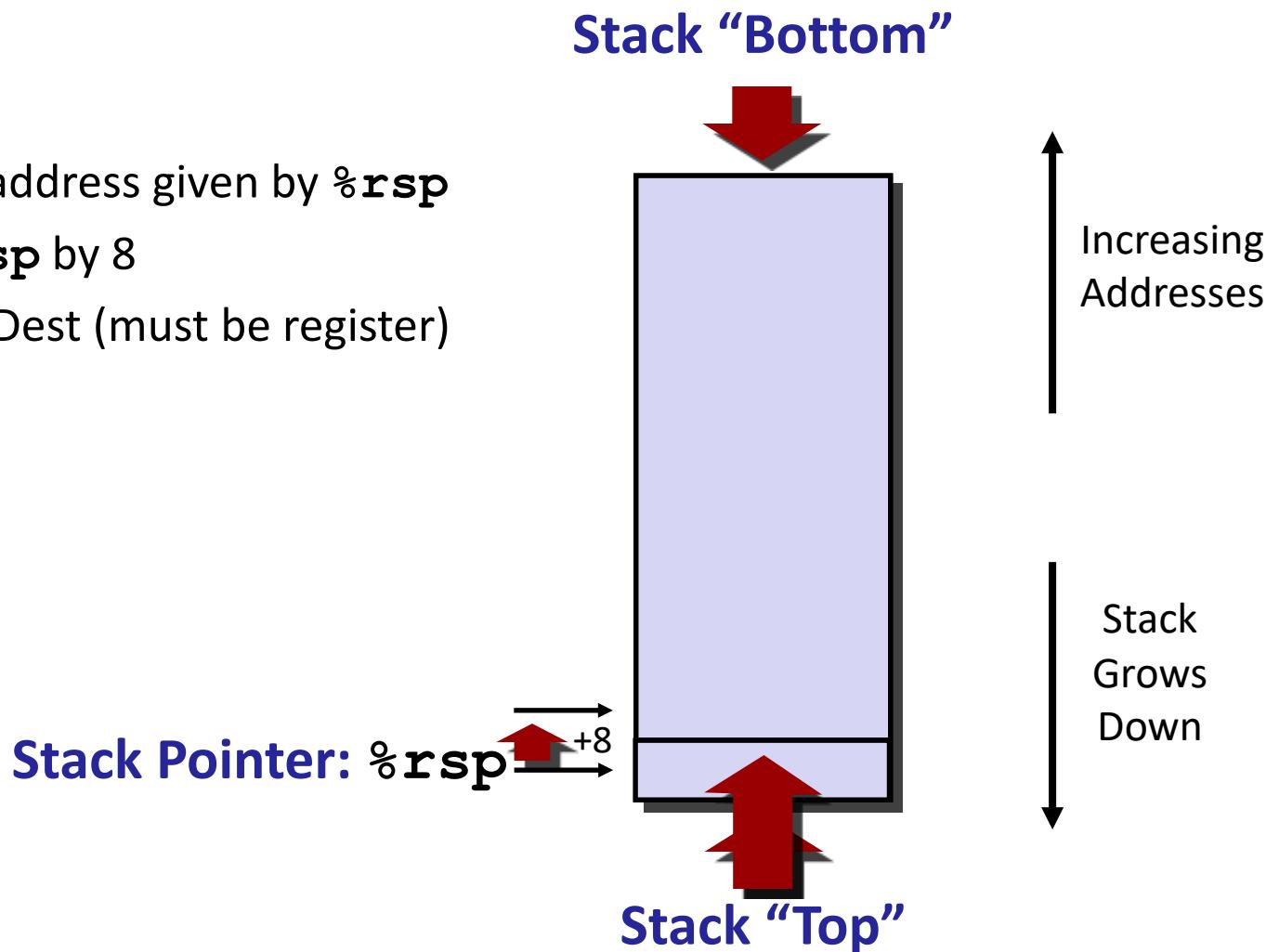
- Fetch operand at *Src*
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`



x86-64 Stack: Pop

■ **popq Dest**

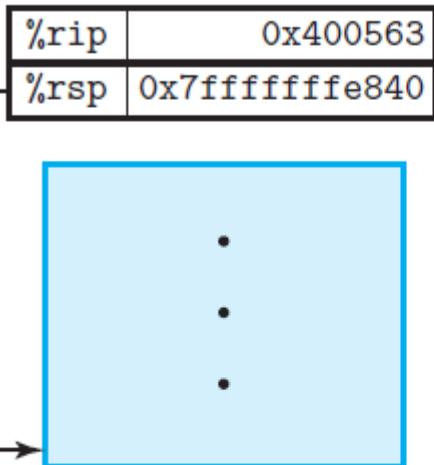
- Read value at address given by `%rsp`
- Increment `%rsp` by 8
- Store value at Dest (must be register)



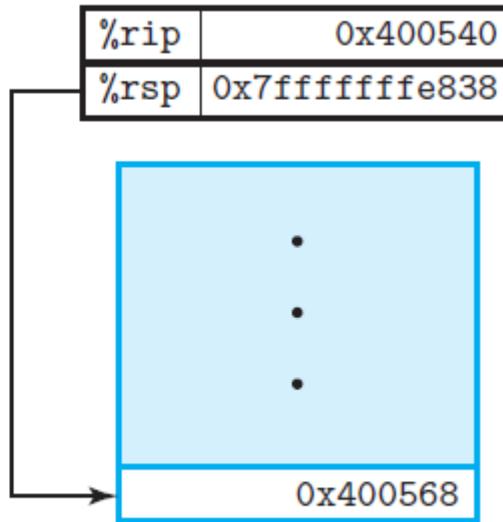
Today

■ Procedures

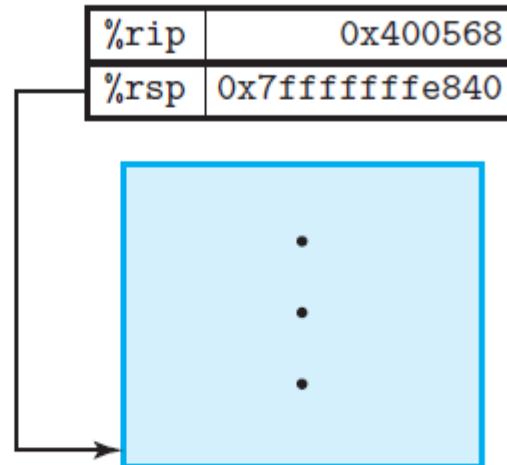
- Stack Structure
- Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
- Illustration of Recursion



(a) Executing call



(b) After call



(c) After ret

Beginning of function multstore

```

1 0000000000400540 <multstore>:
2   400540: 53           push    %rbx
3   400541: 48 89 d3     mov     %rdx,%rbx
...

```

Return from function multstore

```

4   40054d: c3           retq
...

```

Call to multstore from main

```

5   400563: e8 d8 ff ff ff  callq  400540 <multstore>
6   400568: 48 8b 54 24 08  mov    0x8(%rsp),%rdx

```

Procedure Control Flow

■ Use stack to support procedure call and return

■ Procedure call: `call label`

- Push return address on stack
- Jump to *label*

Instruction	Description
<code>call Label</code>	Procedure call
<code>call *Operand</code>	Procedure call
<code>ret</code>	Return from call

■ Return address:

- Address of the next instruction right after call
- Example from disassembly

■ Procedure return: `ret`

- Pop address from stack
- Jump to address

Disassembly of leaf(long y)

y in %rdi

1 0000000000400540 <leaf>:

2 400540: 48 8d 47 02 lea 0x2(%rdi),%rax L1: z+2

3 400544: c3 retq L2: Return

4 0000000000400545 <top>:

Disassembly of top(long x)

x in %rdi

5 400545: 48 83 ef 05 sub \$0x5,%rdi T1: x-5

6 400549: e8 f2 ff ff ff callq 400540 <leaf> T2: Call leaf(x-5)

7 40054e: 48 01 c0 add %rax,%rax T3: Double result

8 400551: c3 retq T4: Return

. . .

Call to top from function main

9 40055b: e8 e5 ff ff ff callq 400545 <top> M1: Call top(100)

10 400560: 48 89 c2 mov %rax,%rdx M2: Resume

(b) Execution trace of example code

Instruction			State values (at beginning)					Description
Label	PC	Instruction	%rdi	%rax	%rsp	*%rsp		
M1	0x40055b	callq	100	—	0x7fffffff820	—	—	Call top(100)
T1	0x400545	sub	100	—	0x7fffffff818	0x400560	0x400560	Entry of top
T2	0x400549	callq	95	—	0x7fffffff818	0x400560	0x400560	Call leaf(95)
L1	0x400540	lea	95	—	0x7fffffff810	0x40054e	0x40054e	Entry of leaf
L2	0x400544	retq	—	97	0x7fffffff810	0x40054e	0x40054e	Return 97 from leaf
T3	0x40054e	add	—	97	0x7fffffff818	0x400560	0x400560	Resume top
T4	0x400551	retq	—	194	0x7fffffff818	0x400560	0x400560	Return 194 from top
M2	0x400560	mov	—	194	0x7fffffff820	—	—	Resume main

Today

■ Procedures

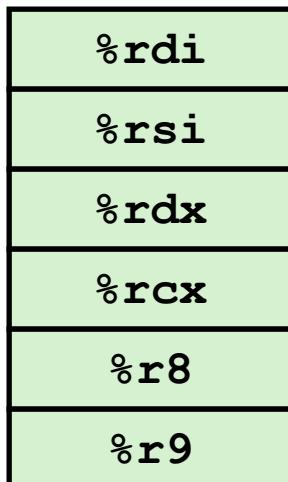
- Stack Structure
- Calling Conventions
 - Passing control
 - **Passing data**
 - Managing local data
- Illustrations of Recursion & Pointers

Procedure Data Flow

Registers

Stack

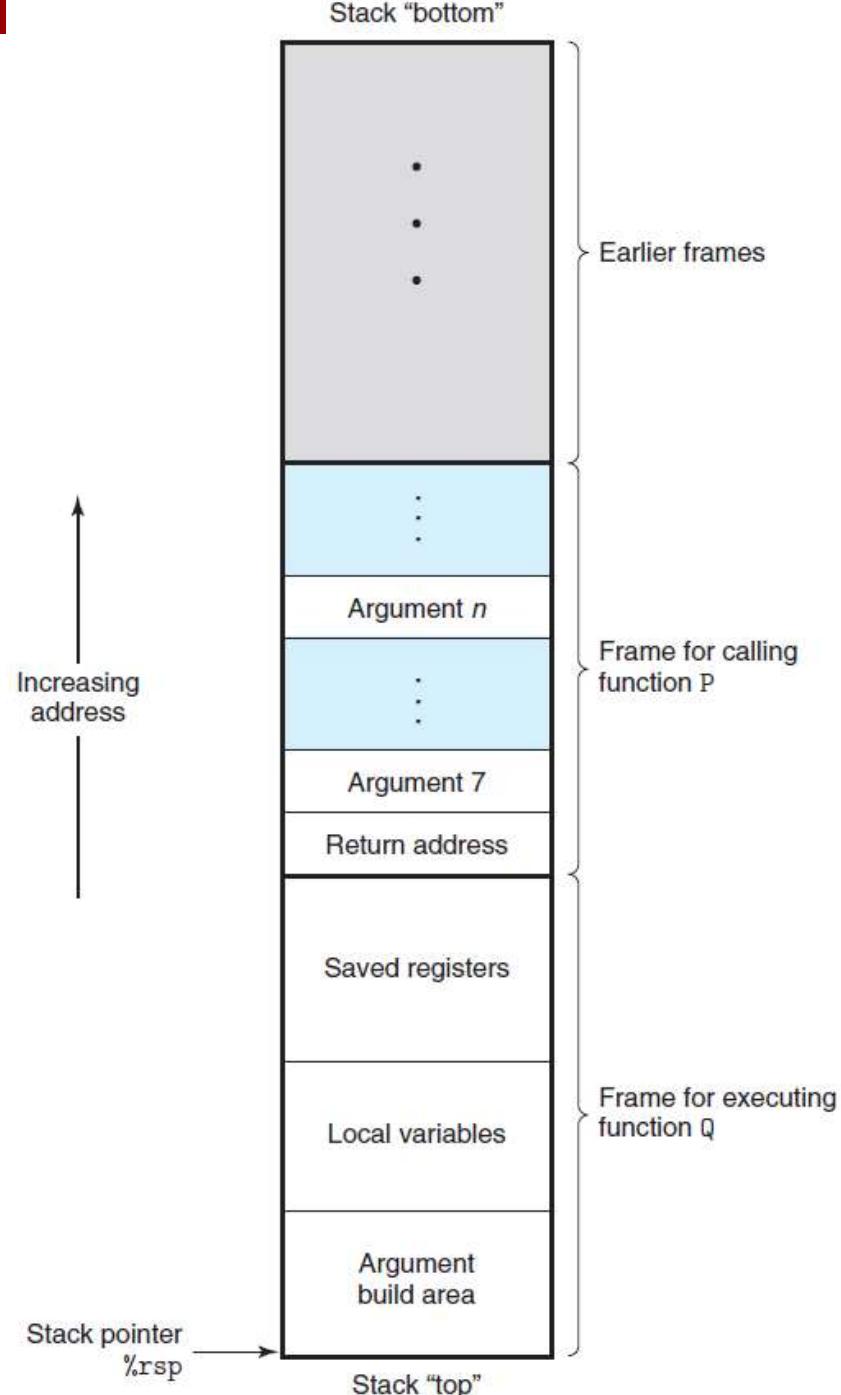
■ First 6 arguments



■ Return value

%rax

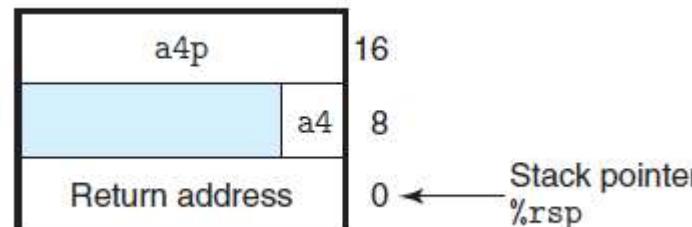
- Only allocate stack space when needed



Operand size (bits)	Argument number					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dl	%sil	%dl	%cl	%r8b	%r9b

(a) C code

```
void proc(long a1, long *a1p,  
         int a2, int *a2p,  
         short a3, short *a3p,  
         char a4, char *a4p)  
{  
    *a1p += a1;  
    *a2p += a2;  
    *a3p += a3;  
    *a4p += a4;  
}
```



(b) Generated assembly code

```
void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)  
Arguments passed as follows:  
    a1    in %rdi          (64 bits)  
    a1p   in %rsi          (64 bits)  
    a2    in %edx          (32 bits)  
    a2p   in %rcx          (64 bits)  
    a3    in %r8w          (16 bits)  
    a3p   in %r9           (64 bits)  
    a4    at %rsp+8        ( 8 bits)  
    a4p   at %rsp+16       (64 bits)  
  
1 proc:  
2     movq  16(%rsp), %rax      Fetch a4p  (64 bits)  
3     addq  %rdi, (%rsi)       *a1p += a1  (64 bits)  
4     addl  %edx, (%rcx)       *a2p += a2  (32 bits)  
5     addw  %r8w, (%r9)        *a3p += a3  (16 bits)  
6     movl  8(%rsp), %edx      Fetch a4    ( 8 bits)  
7     addb  %dl, (%rax)        *a4p += a4  ( 8 bits)  
8     ret                      Return
```

Today

■ Procedures

- Stack Structure
- Calling Conventions
 - Passing control
 - Passing data
 - Managing local data
- Illustration of Recursion

Stack-Based Languages

■ Languages that support recursion

- e.g., C, Pascal, Java
- Code must be “*Reentrant*”
 - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
 - Arguments
 - Local variables
 - Return pointer

■ Stack discipline

- State for given procedure needed for limited time
 - From when called to when return
- Callee returns before caller does

■ Stack allocated in *Frames*

- state for single procedure instantiation

Call Chain Example

```
yoo (...)
```

```
{
```

```
•  
•  
who () ;  
•  
•
```

```
}
```

```
who (...)
```

```
{
```

```
• • •  
amI () ;  
• • •  
amI () ;  
• • •
```

```
}
```

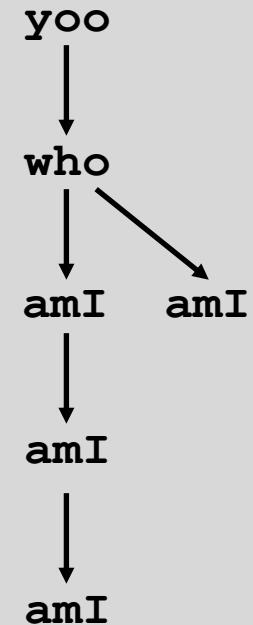
```
amI (...)
```

```
{
```

```
•  
•  
amI () ;  
•  
•
```

```
}
```

Example Call Chain

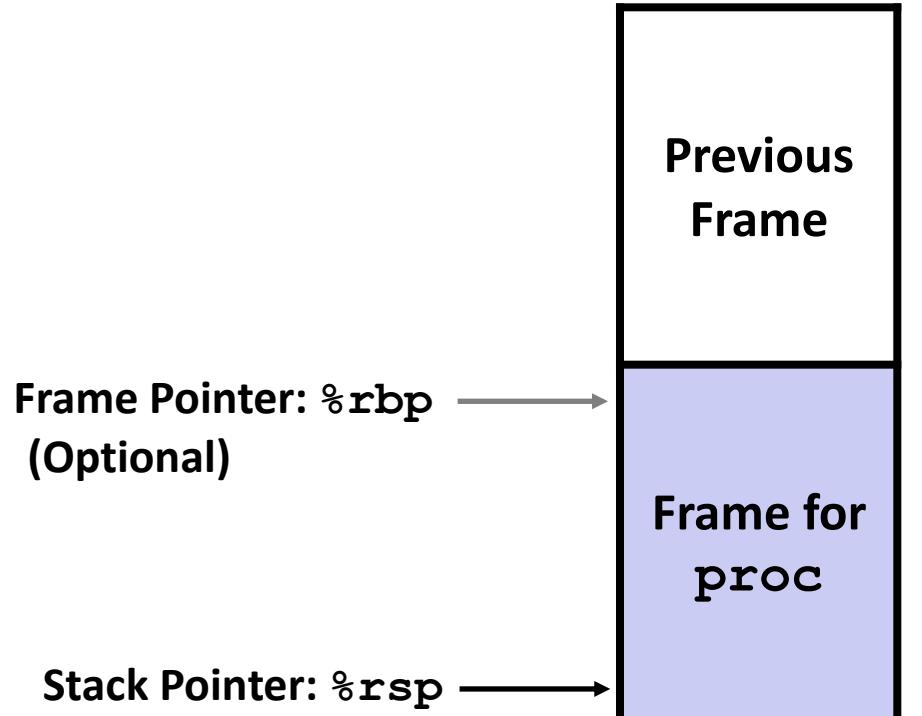


Procedure **amI ()** is recursive

Stack Frames

■ Contents

- Return information
- Local storage (if needed)
- Temporary space (if needed)



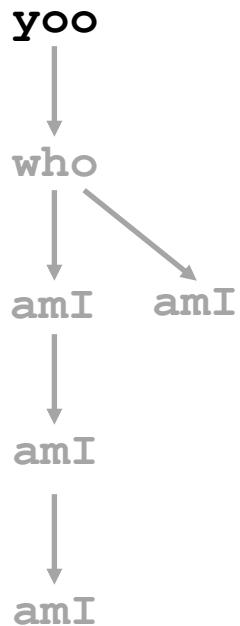
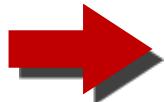
■ Management

- Space allocated when enter procedure
 - “Set-up” code
 - Includes push by **call** instruction
- Deallocated when return
 - “Finish” code
 - Includes pop by **ret** instruction

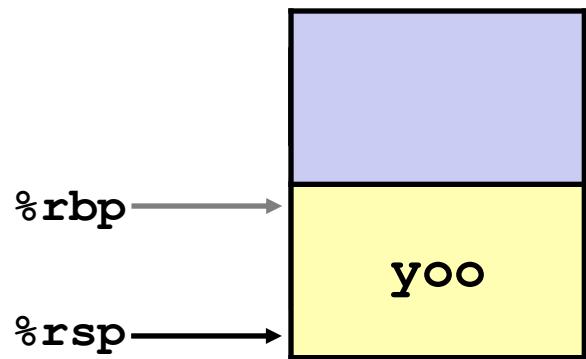
Stack “Top”

Example

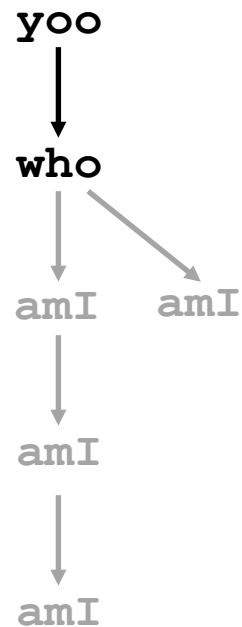
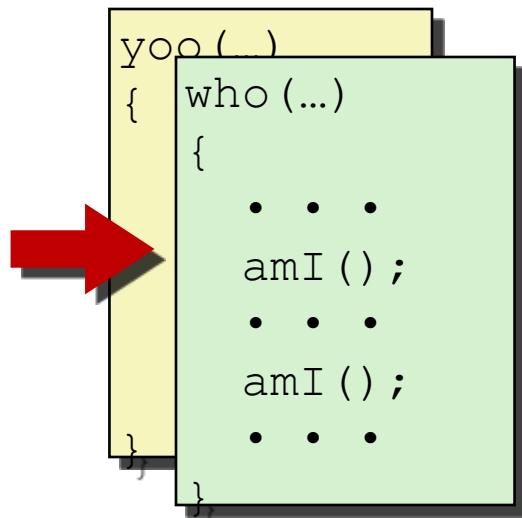
```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```



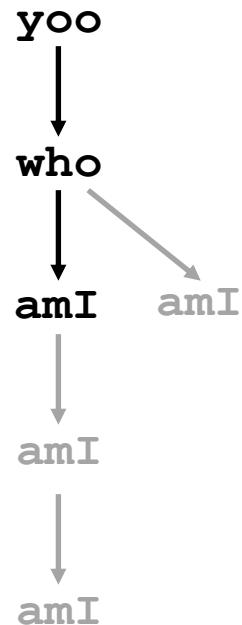
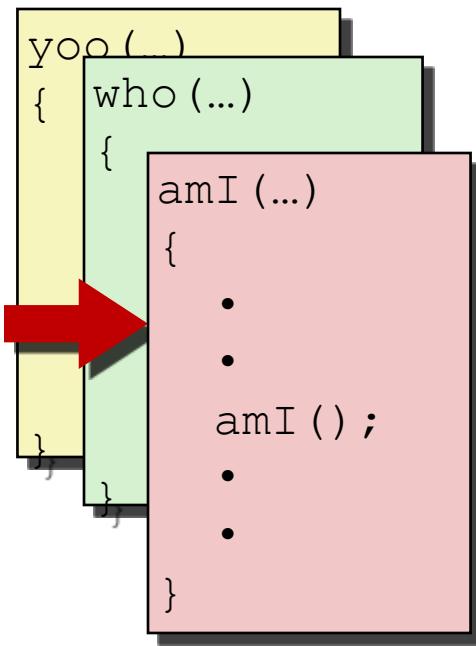
Stack



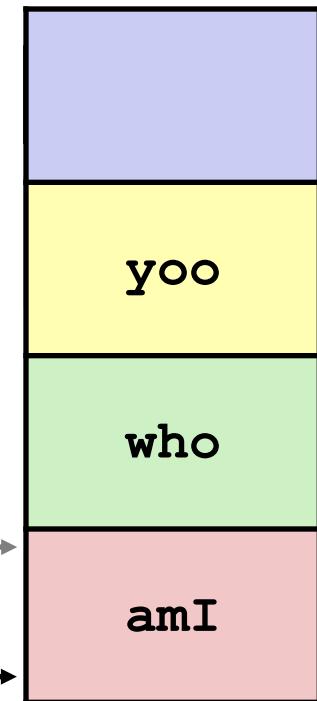
Example



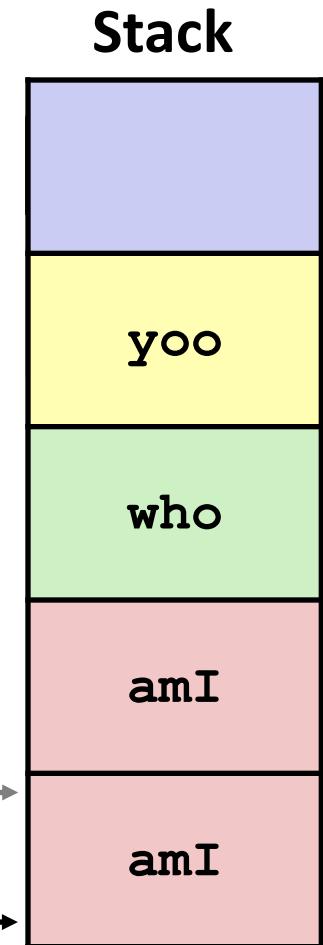
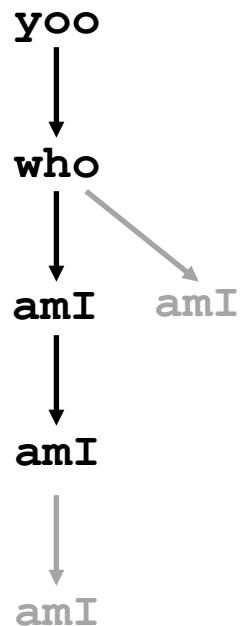
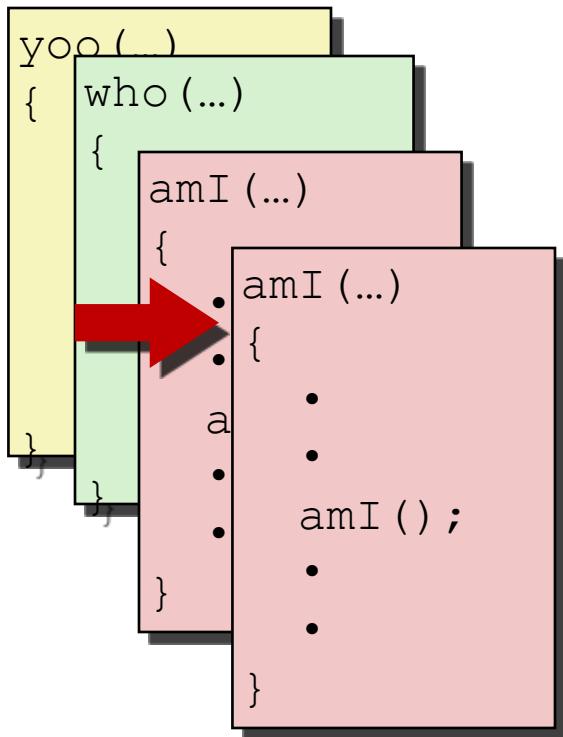
Example



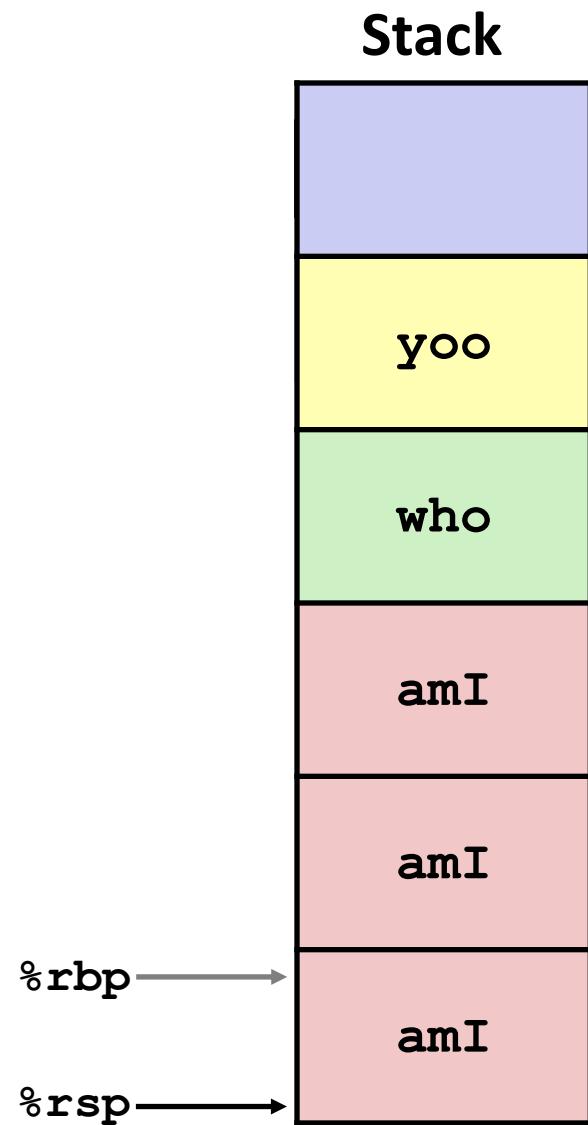
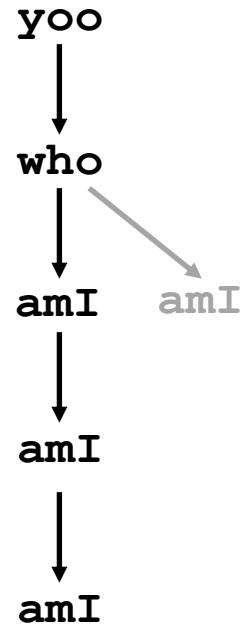
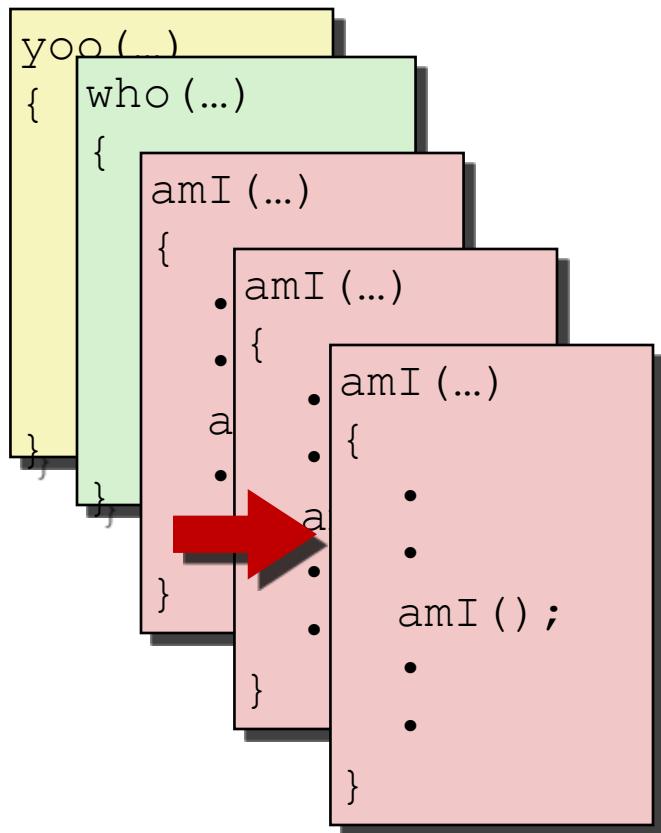
Stack



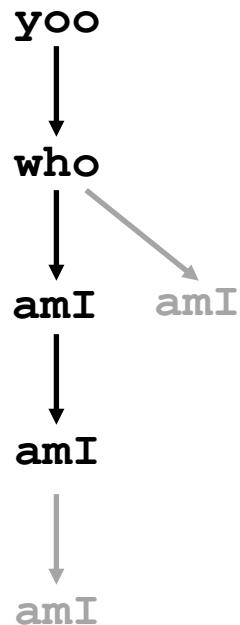
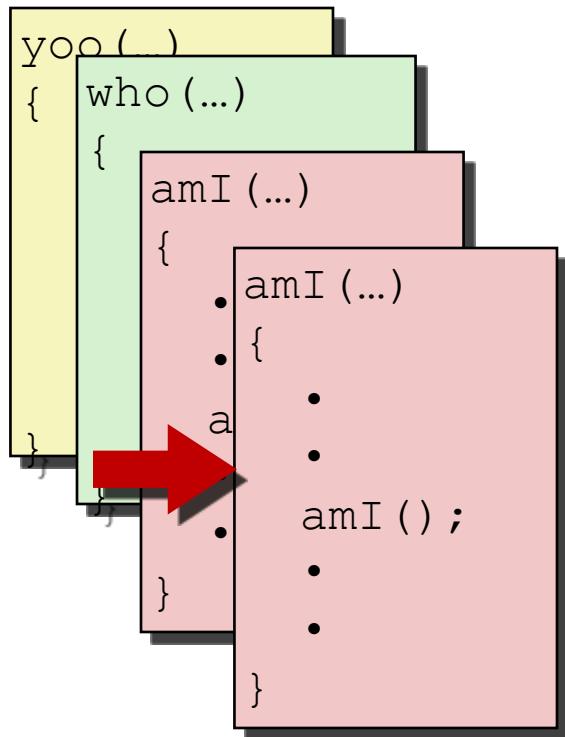
Example



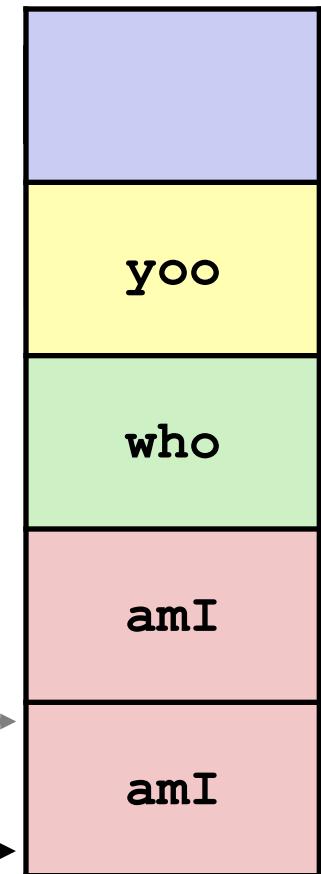
Example



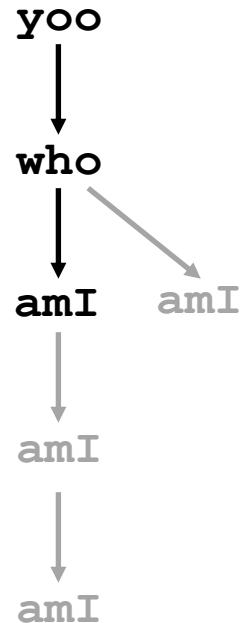
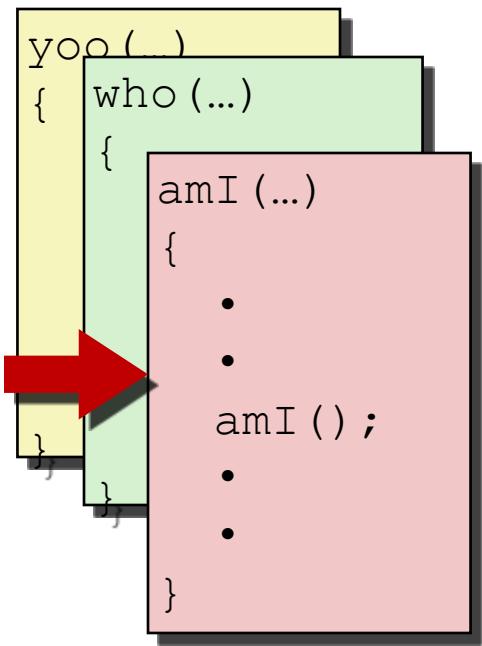
Example



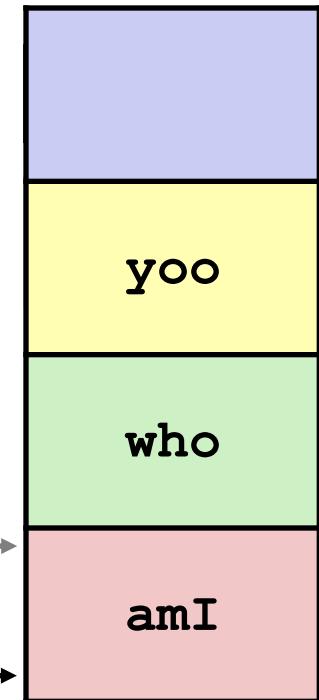
Stack



Example

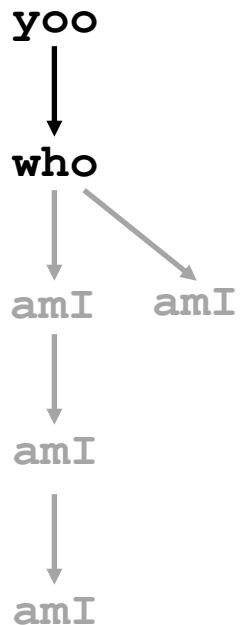


Stack

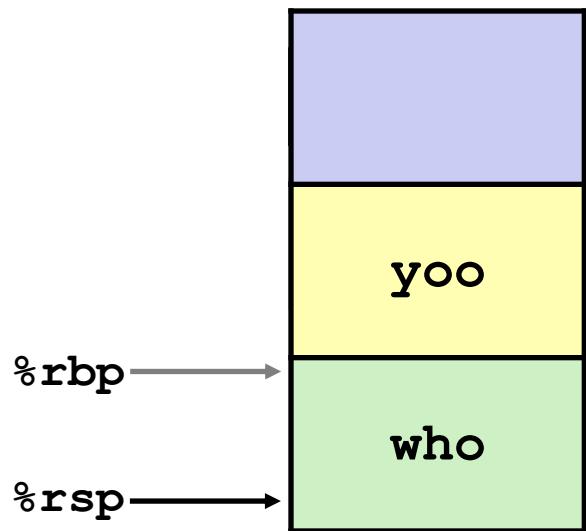


Example

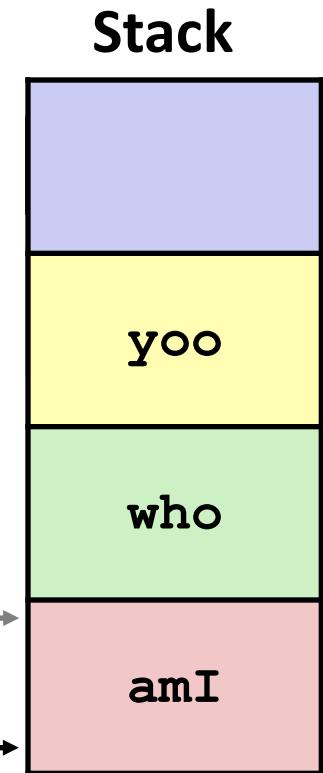
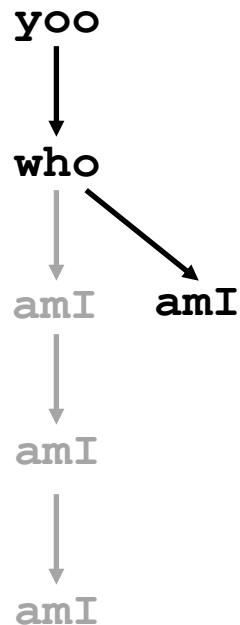
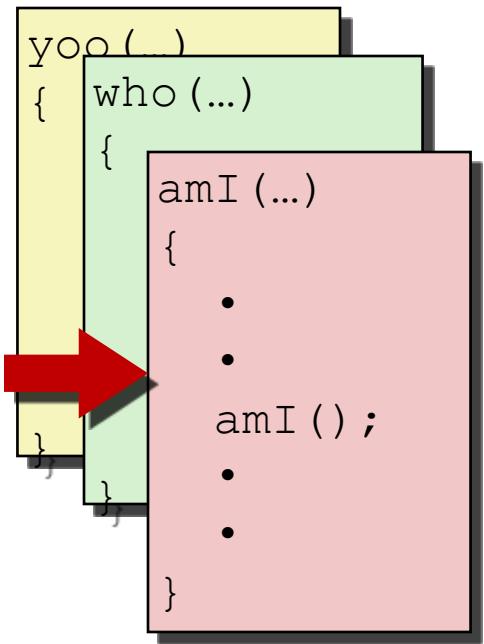
```
yoo(...)  
{    who(...)  
{  
    • • •  
    amI();  
    • • •  
    amI();  
    • • •  
}
```



Stack

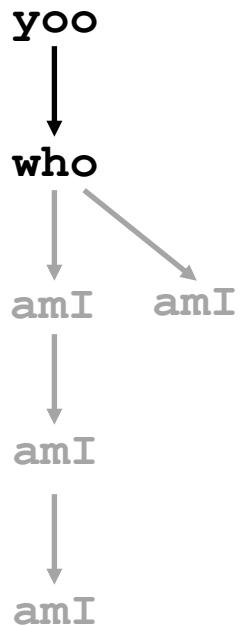


Example

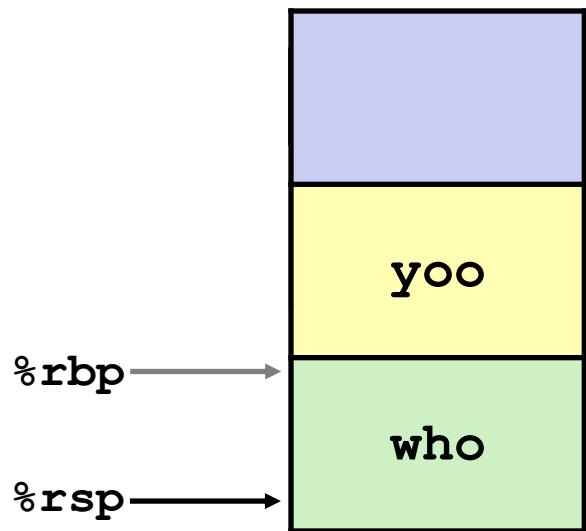


Example

```
yoo(...)  
{  
    who(...)  
    {  
        . . .  
        amI();  
        . . .  
        amI();  
        . . .  
    }  
}
```

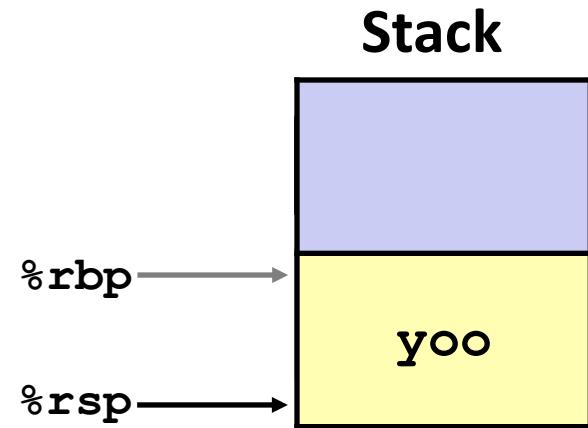
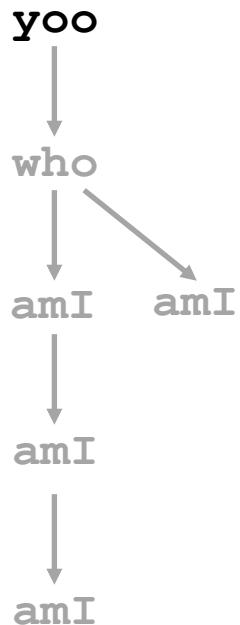


Stack



Example

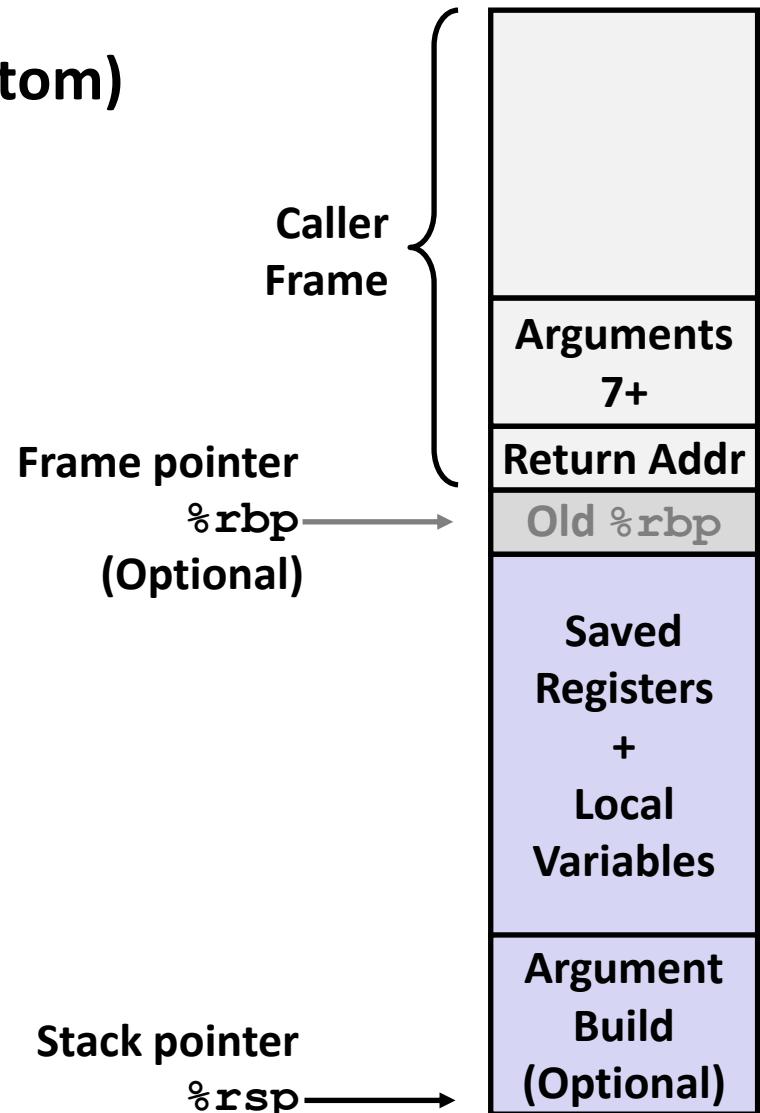
```
yoo (...)  
{  
    •  
    •  
    who () ;  
    •  
    •  
}
```



x86-64/Linux Stack Frame

■ Current Stack Frame (“Top” to Bottom)

- “Argument build:”
Parameters for function about to call
- Local variables
If can’t keep in registers
- Saved register context
- Old frame pointer (optional)



■ Caller Stack Frame

- Return address
 - Pushed by **call** instruction
- Arguments for this call

(a) Code for swap_add and calling function

```
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

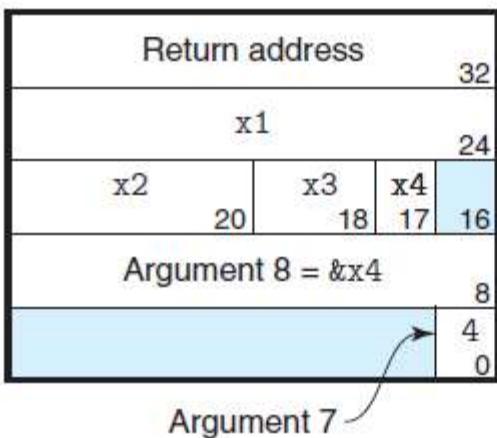
long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

(b) Generated assembly code for calling function

```
long caller()
1   caller:
2       subq    $16, %rsp          Allocate 16 bytes for stack frame
3       movq    $534, (%rsp)      Store 534 in arg1
4       movq    $1057, 8(%rsp)    Store 1057 in arg2
5       leaq    8(%rsp), %rsi     Compute &arg2 as second argument
6       movq    %rsp, %rdi        Compute &arg1 as first argument
7       call    swap_add         Call swap_add(&arg1, &arg2)
8       movq    (%rsp), %rdx      Get arg1
9       subq    8(%rsp), %rdx      Compute diff = arg1 - arg2
10      imulq   %rdx, %rax      Compute sum * diff
11      addq    $16, %rsp        Deallocate stack frame
12      ret
```

(a) C code for calling function

```
long call_proc()
{
    long x1 = 1; int x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

*long call_proc()*

```
call_proc:
    Set up arguments to proc
    subq    $32, %rsp
    movq    $1, 24(%rsp)
    movl    $2, 20(%rsp)
    movw    $3, 18(%rsp)
    movb    $4, 17(%rsp)
    leaq    17(%rsp), %rax
    movq    %rax, 8(%rsp)
    movl    $4, (%rsp)
    leaq    18(%rsp), %r9
    movl    $3, %r8d
    leaq    20(%rsp), %rcx
    movl    $2, %edx
    leaq    24(%rsp), %rsi
    movl    $1, %edi
    Call proc
    call    proc
    Retrieve changes to memory
    movslq  20(%rsp), %rdx
    addq    24(%rsp), %rdx
    movswl  18(%rsp), %eax
    movsbl  17(%rsp), %ecx
    subl    %ecx, %eax
    cltq
    imulq  %rdx, %rax
    addq    $32, %rsp
    ret
```

Allocate 32-byte stack frame

Store 1 in &x1

Store 2 in &x2

Store 3 in &x3

Store 4 in &x4

Create &x4

Store &x4 as argument 8

Store 4 as argument 7

Pass &x3 as argument 6

Pass 3 as argument 5

Pass &x2 as argument 4

Pass 2 as argument 3

Pass &x1 as argument 2

Pass 1 as argument 1

Get x2 and convert to long

Compute x1+x2

Get x3 and convert to int

Get x4 and convert to int

Compute x3-x4

Convert to long

Compute (x1+x2) * (x3-x4)

Deallocate stack frame

Return