

CMPT 2395

Intro to Computer Architecture

Shahriar Khosravi, Ph.D.

Lecture 1



Plan for Today

- Introduction
- The Welcome Survey
- Our Course Policies
- Computer and CPU Components
- UNIX, Linux, and the Command Line Interface
- The C Programming Language (hint: C++ is equivalent to $C = C + 1$)

Shahriar Khosravi (He/Him)

- **Bachelor of Science** in Aerospace Science and Engineering (**University of Michigan**, 2012)
- **Doctor of Philosophy** in Aerospace Science and Engineering (**University of Toronto**, 2016)
 - Thesis topic: Aircraft wing design using numerical optimization based on computational fluid dynamics and finite-element structural analysis
- **Credit Risk Analytics Manager** at BMO Bank of Montreal (Toronto)
 - Data science, software engineering, and numerical modelling using the SAS and SQL scripting languages
- **Structural Dynamics and Vibrations Engineer** at Pratt and Whitney Canada (Toronto)
 - Aircraft engine design from a vibrations perspective and data science for engine component performance prediction
- **Professor of Electronics and Mechanical Engineering Technology** at Seneca College (Toronto)
 - Taught courses on the C, C++, C# programming languages, and the .NET Framework

What You Have Said in the Welcome Survey (Adjectives)

Like
Golf.

Excited!

'Energetically
relaxed'

Like
coffee!

Enjoy math and find
Haskel interesting!

Love playing video
games and hanging out!

I know you
already...

What You Have Said in the Welcome Survey (Concerns)

It is too early!
Also, C++.

The 40% Final is
frightening.

Washroom
breaks during
exams.

Technical jargon.
English please.

Coding and
exams...

I need a good GPA for
transferring. Thank you!

No fun
assignments!

What is CMPT 2395 and Who Should Take This Course?

- Our overarching goal in this course is to introduce you to:
 - Fundamental aspects of computer systems architecture (x86-64)
 - Low-level programming (with C and Assembly)
- To accomplish these goals, we first need to learn:
 - Working with a “UNIX-like” operating system such as Linux that gives us access to lower-level software components
 - Programming with the C language, so we can take advantage of lower-level programming capabilities
- This course is intended for those who would like to develop a deeper understanding of what happens “behind the scenes” when we use higher-level compilers, and how our programs (i.e., software) interact with computer components (i.e., hardware).
- We will not go down to the physics and electromagnetism involved in computer hardware components, but this course will set you up to pursue this further in the future.

What You Need to Succeed in This Course

1. A working computer with the **Windows 10/11 Operating System** that can connect to the Internet
2. The latest version of **Microsoft Visual Studio 2022** (tutorial for installing it is on the Blackboard for your reference)
3. [Linux Ubuntu for Windows 10/11](#) or a Linux emulator such as [Cygwin](#) (in our Lecture and Lab sessions, I have no choice but to use Cygwin because CEIT could not install Ubuntu on the Lab computers)



Our Course Policies



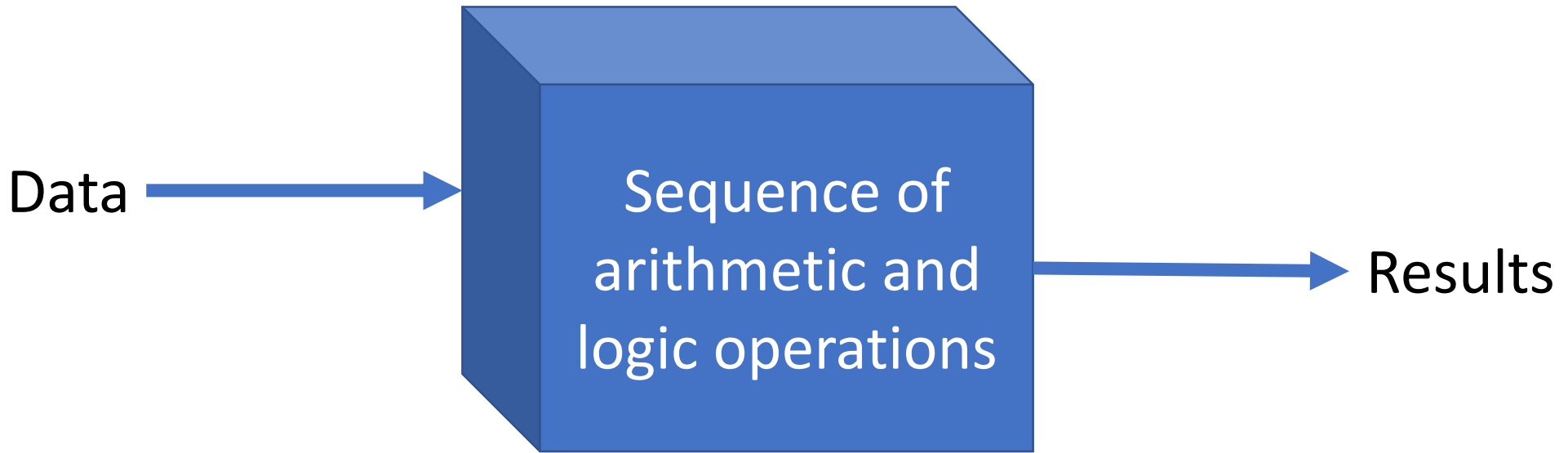
DOUGLAS COLLEGE

Computer and CPU Components

John von Neumann's Computer Architecture

- The vast majority of modern computer designs are based **John von Neumann's** theories at Princeton University.
- This architecture is referred to as the **von Neumann architecture** and is based on three key concepts:
 - Data and instructions are stored in a single read–write memory.
 - This memory's contents are addressable regardless the type of data contained at those addresses.
 - Execution is sequential (unless explicitly modified) from one instruction to the next.
- There are a few basic logic components that can be combined in various ways to store binary data and perform arithmetic and logical operations on that data.
- For a specific computation, we can construct a configuration of logic components designed specifically for that computation.
- We can think of the process of connecting the various components in the desired configuration as a form of programming.
- This “program” is in the form of hardware and is called a hardwired program.

Software and Hardware Approaches

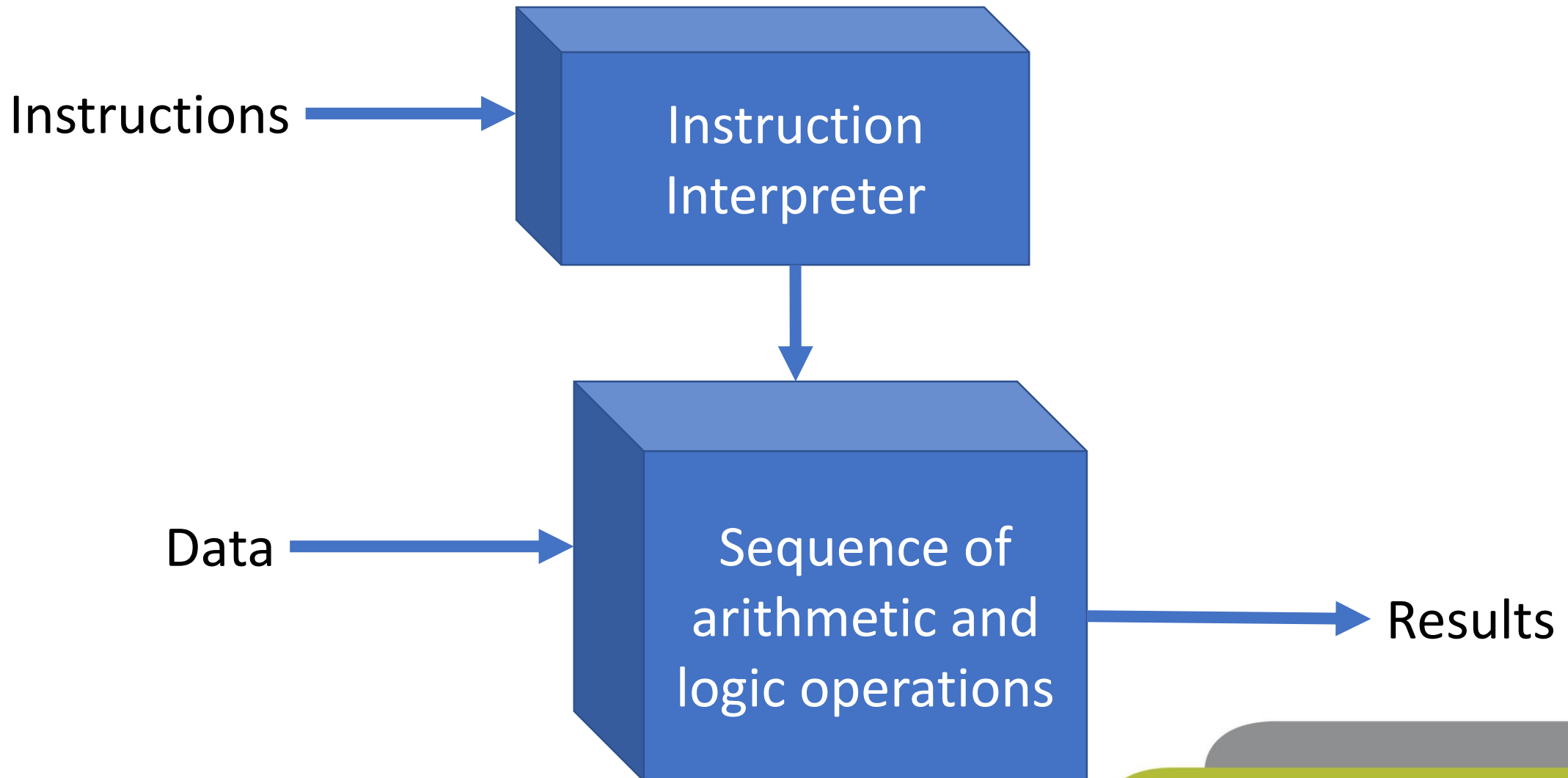


- With this approach, we must make a new hardwired, physical “program” for every single task we want to accomplish.
- This will prove cumbersome because we will end up with different computers for every single computing task.

Software and Hardware Approaches

- Let us consider an alternative approach.
- Suppose we construct a general-purpose configuration of arithmetic and logic functions.
- This hardware configuration will perform different functions on data depending on **control signals** provided to the hardware.
- This system can accept data and control signals as input and produce results as output.
- Therefore, instead of rewiring the hardware for each new program, the programmer simply needs to provide a new set of control signals!
- But how should we provide these control signals to the hardware configuration?
- We need an interpreter component that can accept programmer's code and convert it to a set of control signals that the hardware configuration can understand and perform.
- To standardize this process, we need unique codes for all possible set of control signals that we can send to this new interpreter module.

Software and Hardware Approaches



Software

- With this approach, programming is much easier because instead of rewiring the hardware for each new program, all we need to do is provide a new sequence of codes.
- Each code is, in essence, an instruction, and part of the hardware interprets each instruction and generates control signals.
- To distinguish this new method of programming, a sequence of codes or instructions is called software.
- An instruction interpreter and a module of general-purpose arithmetic and logic functions together constitute the Central Processing Unit (CPU).

Input/output (I/O) Components

- Several other components are still needed to yield a functioning computer.
- Data and instructions must somehow be put into the system.
- For this we need some sort of **input module**.
- This module contains basic components for accepting data and instructions in some form and converting them into an internal form of signals that the system can understand and use.
- We also need a means of reporting results, and this is in the form of an **output module**.
- Taken together, these are referred to as input/output or **I/O components**.

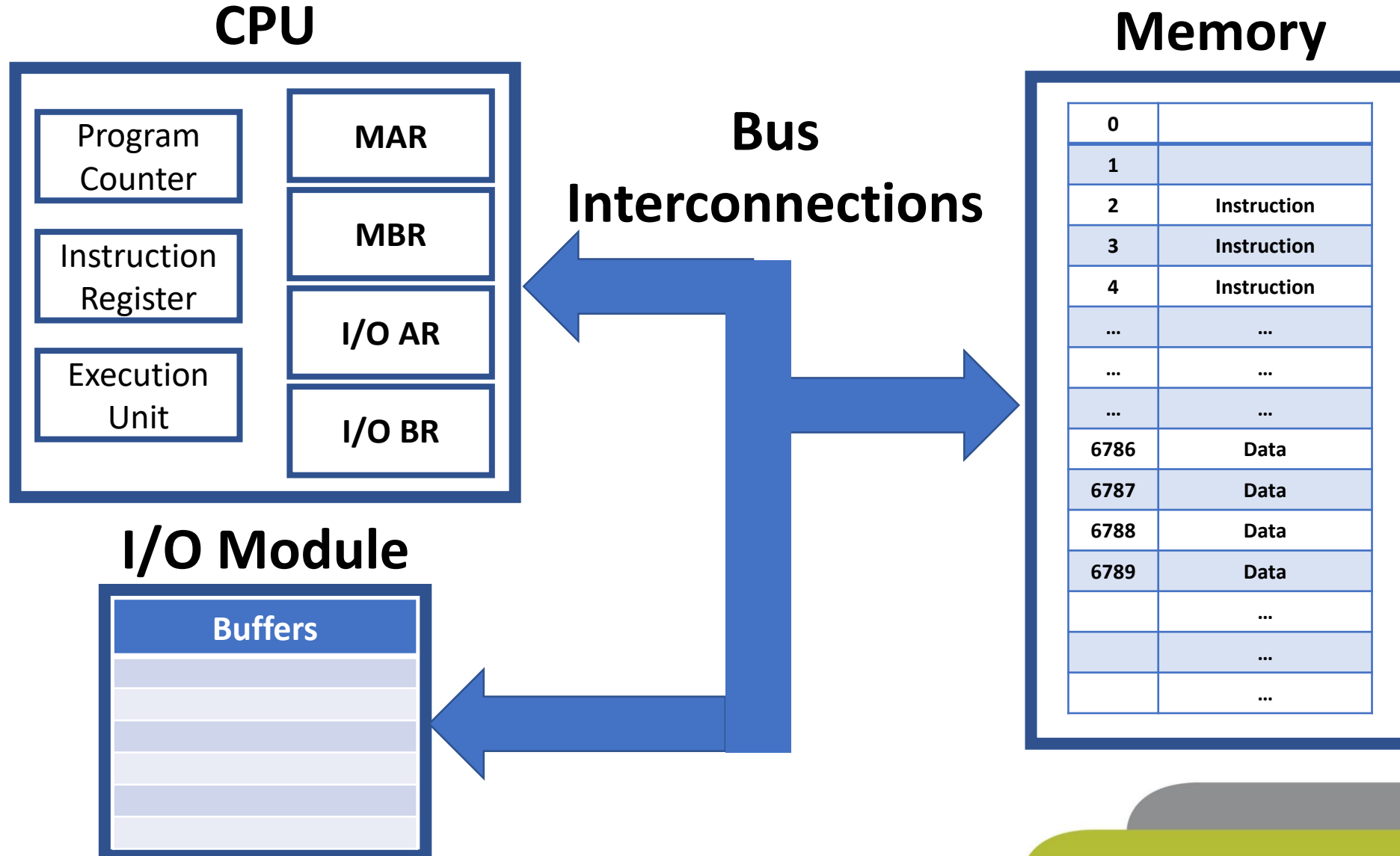
Memory Components

- We still need one more component for a functioning computer.
- An input device will bring instructions and data in sequentially. But a program is not necessarily executed sequentially; it may jump around (e.g., if-statements or a loop).
- Similarly, operations on data may require access to more than just one element at a time in a predetermined sequence.
- Thus, there must be a place to store temporarily both instructions and data.
- That module is called memory, or main memory, to distinguish it from external storage or peripheral devices.
- **Von Neumann pointed out that the same memory could be used to store both instructions and data.**

Memory Components

- The CPU exchanges data with memory. For this purpose, it typically makes use of two internal (to the CPU) registers:
 - a memory address register (MAR), which specifies the address in memory for the next read or write;
 - and a memory buffer register (MBR), which contains the data to be written into memory or receives the data read from memory.
- There is also an I/O address register (I/O AR), which specifies a particular I/O device.
- An I/O buffer register (I/O BR) is used for the exchange of data between an I/O module and the CPU.
- A **memory module** consists of a set of locations, defined by sequentially numbered addresses.
- Each location contains a binary number that can be interpreted as either an instruction or data.
- An I/O module transfers data from external devices to CPU and memory, and vice versa. It contains internal buffers for temporarily holding these data until they can be sent on.

Computer Components Overview



The Linux Operating System

What is Linux?

- Linux is a family of open-source operating systems based on the UNIX operating system.
- It was created by Linus Torvalds in 1991 because he was frustrated with the licensing terms of the Unix Operating System, sold at the time by AT&T.
- Today, there are many open-source and commercial “distributions” of the Linux operating system. Open-source distributions come with thousands of useful software/hardware tools.
- Open-source distributions include CentOS, Fedora, and Ubuntu. Commercial distributions include RedHat and SUSE.
- Navigation in Linux is done using the **Command Line** (i.e., the “**terminal**”).

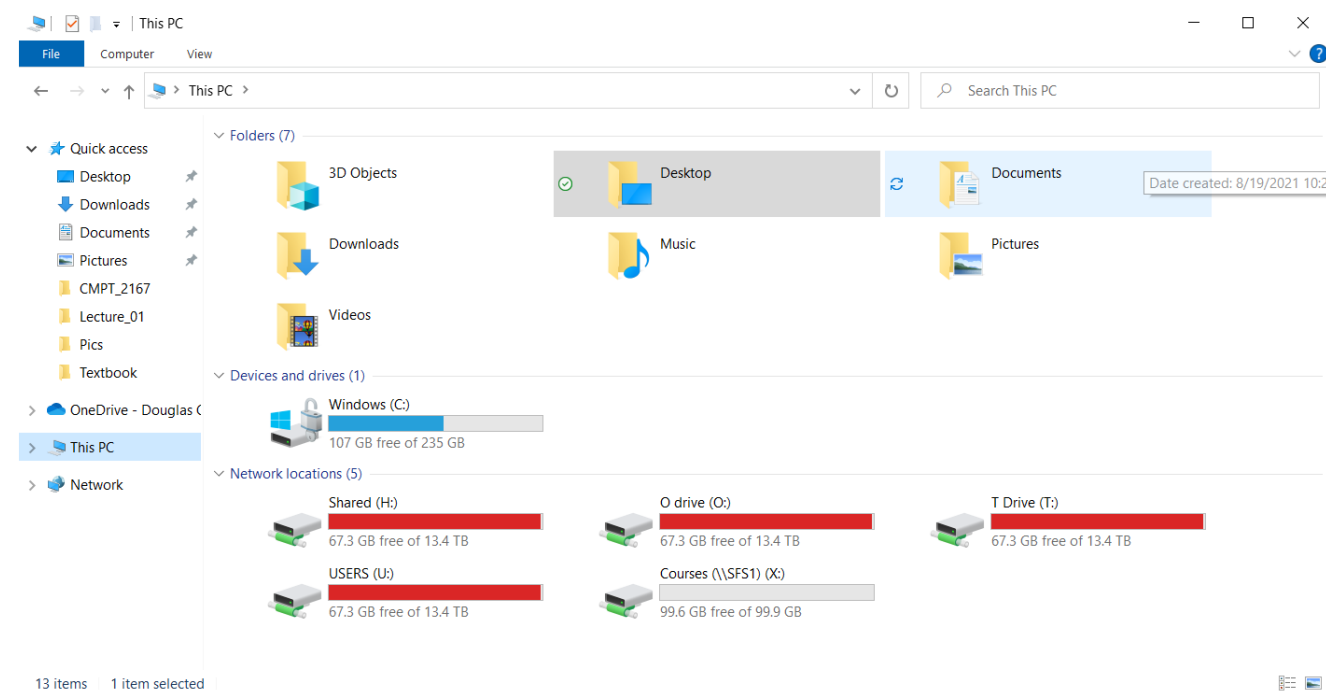


CentOS

What is the Command Line (Terminal)?

- The **Command Line Interface (CLI)** is a text-based **User Interface** for navigating a computer. This is in contrast to a **Graphical User Interface (GUI)**.

```
root@pablo-Lenovo-G550: /home/pablo
File Edit View Search Terminal Help
pablo@pablo-Lenovo-G550:~$ su
Password:
root@pablo-Lenovo-G550:/home/pablo# groupadd sector1
root@pablo-Lenovo-G550:/home/pablo# groupadd sector2
root@pablo-Lenovo-G550:/home/pablo# groupadd sector3
root@pablo-Lenovo-G550:/home/pablo#
```



The Command Line

- Similar to a GUI, a CLI
 - Displays a specific directory on the computer's hard drive
 - Allows us to navigate into and out of folders
 - Allows us to create, edit, and delete new files and folders
 - Allows us to execute programs and applications on the computer
- But why use Linux and the CLI?
- UNIX-based operating systems such as Linux and the Command Line allow us to navigate almost any computer device using the same tools and commands:
 - Servers
 - Laptops and desktops
 - Embedded devices (e.g., Raspberry Pi and Google Coral)
 - Mobile Devices (e.g., Android)

More Incentive for Linux and the CLI

- Linux and the CLI are used frequently by software engineers for:
 - Web development: running servers and web tools on servers
 - Machine learning: processing data on servers, running algorithms
 - Systems: writing operating systems, networking code and embedded software
 - Mobile Development: running tools, managing libraries
 - And much more...
- If you intend to pursue Software Engineering, Hardware Engineering, or Computer Science in general, you must develop a solid understanding of UNIX-like operating systems and useful CLI tools used in the industry on a routine basis.
- In this course, we would have liked to use a Linux distribution, but it is apparently not easy to install it on the Lab computers, so we will use a **Linux emulator** called [Cygwin](#), **which can be installed and executed on Windows**.
- Cygwin is a collection of open-source tools with similar functionality to a Linux distribution.
- You should download and install Cygwin on your own computer for this course.

Linux Commands to Try

- **cd** – to change directories
- **ls** – to list directory contents
- **mkdir** – to make a new directory/folder
- **vi** – to open the **vi** text editor
- **rm** – to remove file(s) or folder(s)
- **man** – to view manual pages or documentation
- You can find a useful “Cheat Sheet” for common UNIX/Linux commands [here](#).
- You should download and install **Cygwin** on your own machine and try to use the commands in the above Cheat Sheet on your own to get familiar with them.
- We will use **Cygwin** and the tools it provides for the purpose of viewing the assembly code for our computer programs

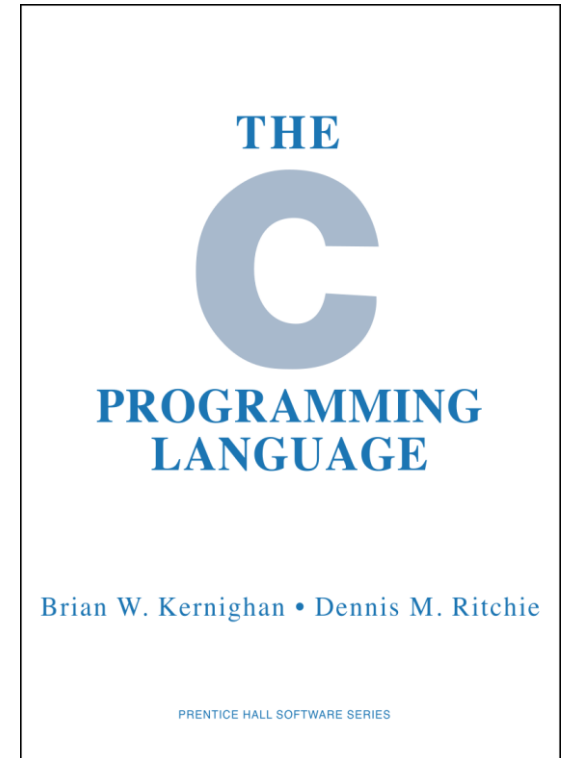


DOUGLAS COLLEGE

The C Programming Language

Introduction

- The C programming language was created around 1970 to simplify creating UNIX tools.
- UNIX itself was originally written in the Assembly language, but later redone using the C programming language!
- C is part of the C/C++/Java family of languages, but C++ and Java were created later to serve different programming paradigms and philosophies.
- The design principles behind the C programming language :
 - Uses simple abstractions of hardware.
 - Has minimalist aesthetic features.
 - Prioritizes efficiency and minimalism over safety and high-level abstractions.



C, C++, and Java Programming Languages

They all share:

- Syntax
- Basic data types
- Arithmetic, relational, and logical operators



C limitations:

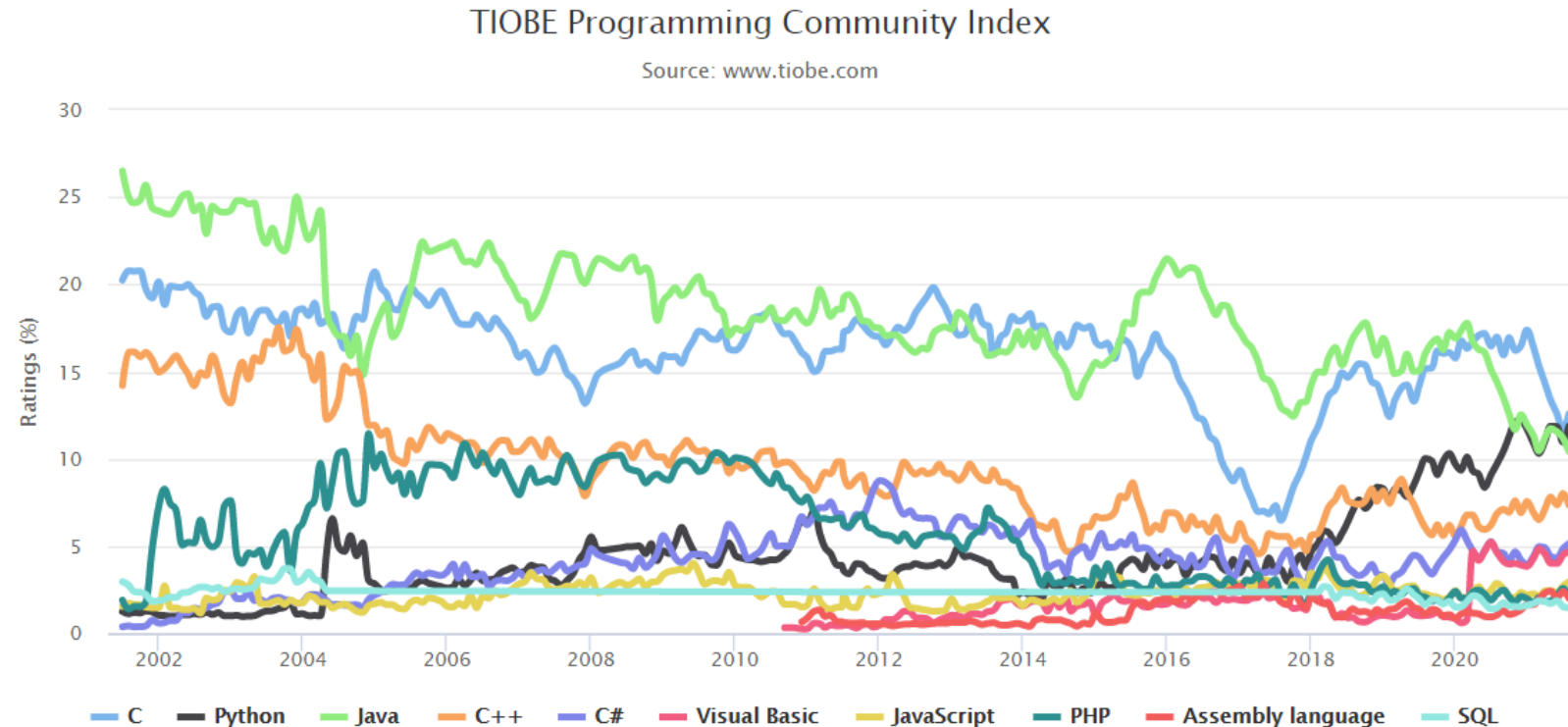
- No advanced features like operator overloading, default arguments, pass by reference, classes and objects, ADTs, etc.
- No extensive libraries (no graphics, networking, etc.) – Small language footprint means not much to learn.
- Weak compiler and almost no runtime checks (which may cause security vulnerabilities).

Programming Languages and Paradigms

- **C** is procedural: we write functions, rather than define new variable types with classes and call methods on objects. C is small, fast, and efficient.
- **C++** is procedural with objects (multi-paradigm): we write functions, and define new variable types with classes, and call methods on objects.
- **Python** is also procedural with objects, but dynamically typed: we still write functions and call methods on objects, but variable types are determined at run-time since there is no compiler.
- **Java** is object-oriented: everything we write needs to conform to the object-oriented design philosophy in software engineering.

Why Use C in Our Course?

- Many tools (and even other languages, like Python) are written using C.
- C is the language of choice for fast and highly efficient programs.
- C is popular for systems programming (e.g., operating systems and networking).
- C is popular for scientific computing (e.g., computer-aided engineering design and analysis).
- C lets you write code at a “lower” level to manipulate and understand the underlying **computer architecture**.



<https://www.tiobe.com/tiobe-index/>

Our First C Program

- Block or inline comments are allowed, just as we have seen in C++.
- Standard C libraries are specified with angle brackets (e.g., `<stdio.h>`)
- For local libraries (i.e., the header files written by us), use quotes instead of angle brackets (e.g., `#include "lib.h"`)
- The `main()` function is the entry point for the program.
- Since the `return` type for this function is specified as `int` in this example, this function must return an integer.
- Here, returning `0` signifies success.
- Function `printf()` prints to the screen.

```
/*  
 * This program prints a message  
 * onto the console screen.  
 */  
#include <stdio.h>      // for printf  
  
int main()  
{  
    printf("Hello, world!\n");  
    return 0;  
}
```


Familiar Syntax from C++

```
#include <stdio.h>
#include <stdbool.h>    // for Boolean type
bool isOdd(long int n)
{
    if (n % 2 == 1)
        return true;

    return false;
}

int main()
{
    int x = 42 + 7 * -5;    // variables, types
    double pi = 3.14159;
    char c = 'Q';    /* two comment styles */

    for (int i = 0; i < 10; i++)    // for loop
    {
        if (i % 2 == 0) // if statement
        {
            x += i;
        }
    }

    while (x <= 0) // while loop
    {
        x++;
    }

    if (isOdd(x))    // function call
        printf("Odd!\n");
    else
        printf("Even!\n");
}
```

Console Output with `printf()`

- Function `printf()` allows us to print the values of variables or expressions onto the terminal.
- We can include “*placeholders*” in our text which will be replaced with the values of passed parameters in the same order as the order of the placeholders.
- These placeholders are officially called **Format Specifiers**.

`%s` (for strings) `%d` (for integers) `%f` (for float) `%lf` (for double)

```
#include <stdio.h>
int main()
{
    char* classPrefix = "CMPT";
    int classNumber = 2395;
    printf("Welcome to %s-%d!", classPrefix, classNumber);
}
```

Console Output with `printf()`

- Function `printf()` is defined in `<stdio.h>` header file.
- Syntax:

```
printf("format_string", variable1, variable2, variable3, ...);
```

- `format_string` is a string that can contain 3 types of elements:
 - Plain characters that will be displayed on the screen (unchanged).
 - Format specifiers used as placeholders, which will be replaced by the values of variables listed after the `format_string`.
 - Format specifier begins with `%`, followed by the data type code.
 - Data type code must match the variable data type.
 - Example: `%c` for a variable of character data type.
 - For each listed variable, there should be one format specifier.
 - Format specifiers are replaced in sequential order.
 - Escape sequences used to control the cursor or insertion point.

Console Output with printf()

```
int varInt = 12;
float varFloat = 3.0;
printf("Character: %c\tInteger: %d\tFloat: %f\n", 'A', varInt, varFloat);
```

format_string *escape sequence* *variable list*

Output:

Character: A Integer: 12 Float: 3.000000

Examples:

```
int size = 7;
float average = 80.5;
printf("Welcome!\n");
printf("a=%d b=%d\n\n", 3, size);
printf("Test Results:\nAverage=%.2f", average);
```

Output:

Welcome!
a=3 b=7

Test Results:

Average = 80.50

Escape Sequences

Escape Sequence	Name	Description
\a	Alert	Sounds a beep
\b	Back space	Backs up 1 character
\f	Form feed	Starts a new screen of page
\n	New line	Moves to the beginning of next line
\r	Carriage return	Moves to the beginning of current line
\t	Horizontal tab	Moves to the next tab position
\v	Vertical tab	Moves down a fixed amount
\\	Back slash	Prints a back slash
\'	Single quotation	Prints a single quotation
\"	Double quotation	Prints a double quotation
\?	Question mark	Prints a question mark

Console Input with `scanf()`

- The `scanf()` function reads data from the input device and stores it in a variable.
- To use `scanf()`, we must `#include <stdio.h>`.
- Syntax:

```
scanf("format_string", &variable1, &variable2, ...);
```

- Missing `&` in `scanf()` is a common error; it leads to abnormal program termination.
- `format_string` specifies the data type of each variable from the list.
- Common format specifiers:

Data Type	Format Specifier
int	%d
float	%f
double	%lf
char	%c
string	%s

Example

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdbool.h>    // for Boolean type

int main()
{
    int number;
    printf("Enter a number: "); //User prompted to enter a number
    scanf("%d", &number); /*scanf reads an int value from the keyboard and stores
    it into variable number*/
    printf("You entered: %d\n", number);

    int age;
    float height;
    printf("Enter your age and height: ");
    scanf("%d%f", &age, &height); // %d is used for variable age, %f for height

    printf("You entered: %d, %.2f", age, height);

    return 0;
}
```

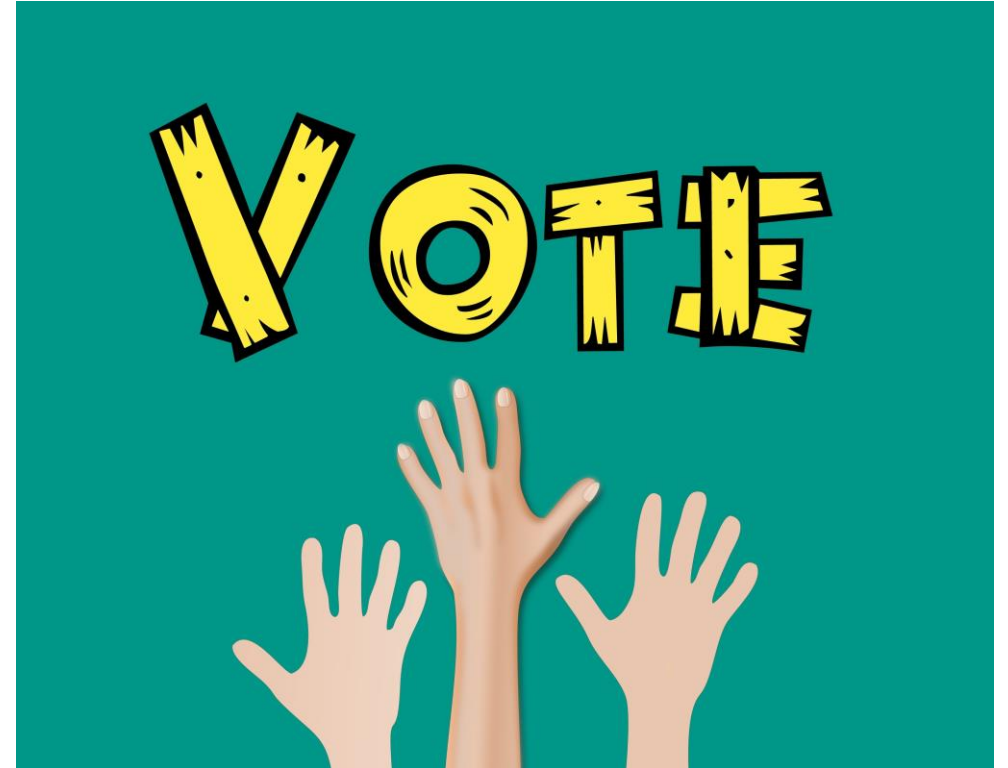
Poll (Extra Credit)

Assuming height is a floating point variable, the following statement will lead to a compilation error.

```
printf("%d\n", height);
```

- a) True
- b) False
- c) Not sure!

Please use the “Poll” window to participate for extra credit!



In-Class Exercise

Follow the instructions provided in the exercise description document provided on the Blackboard webpage under **Lecture Notes 1** item of the **Course Notes** tab.



References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2395

Intro to Computer Architecture

Shahriar Khosravi, Ph.D.

Lecture 2



CMPT 2395

Intro to Computer Architecture

Shahriar Khosravi, Ph.D.

Lecture 3



Plan for Today

- Bitwise Operators
- Bit Masking
- Text Representation: Characters
- Text Representation: C-Strings



DOUGLAS COLLEGE

Bitwise Operators

Bitwise Operators

- We are already familiar with many operators in C:
 - Arithmetic operators: `+`, `-`, `*`, `/`, `%`
 - Comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
 - Logical Operators: `&&`, `||`, `!`
- Today, we're introducing a new category of operators: bitwise operators
 - `&`, `|`, `~`, `^`, `<<`, `>>`
- Bitwise operators allow us to perform low-level operations (in this case, bit-level operations to be precise).

Bitwise AND: &

- The output of bitwise AND (&) is equal to 1 if the corresponding bits of two operands are equal to 1
- If either bit of an operand is equal to 0, the result of the corresponding bit is evaluated to 0

- Example: $12_{10} \& 25_{10} = 8$

$$\begin{array}{rcl}
 12_{10} & = & 00001100_2 \\
 25_{10} & = & 00011001_2 \\
 & & \text{-----} \\
 12\&25 & = & 00001000_2 = 8_{10}
 \end{array}$$

Digit 1	Digit 2	Result
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise AND: &

- A neat example for the application of the bitwise AND operator is checking whether a number is even or odd.
- If the result of $n \& 1$ is equal to **1**, then the number is odd, otherwise even.
- When converting to base-10, if the rightmost bit (or the least significant bit) is equal to 1, then the number must be odd because all other bits are powers of 2 (hence even).

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

bool is_even(int n)
{
    if ((n & 1) == 1)
        return false;
    return true;
}

int main()
{
    is_even(456) ? printf("Even!\n") : printf("Odd!\n");
    is_even(55) ? printf("Even!\n") : printf("Odd!\n");

    return 0;
}
```

Bitwise OR: |

- The output of bitwise OR (|) compares each binary digit across two integers and returns 1 if either of them is equal to 1.
- The OR (|) operator truth table:
- Example: **12 | 25 = 29**

Digit 1	Digit 2	Result
0	0	0
0	1	1
1	0	1
1	1	1

$$\begin{array}{rcl}
 12_{10} & = & 00001100_2 \\
 25_{10} & = & 00011001_2 \\
 & & \text{-----} \\
 12 | 25 & = & 00011101_2 = 29_{10}
 \end{array}$$

Bitwise XOR: ^

- The output of bitwise XOR (^) compares each binary digit across two integers and returns 1 only if one of them is equal to 1.

- The XOR (^) operator truth table:

Digit 1	Digit 2	Result
0	0	0
0	1	1
1	0	1
1	1	0

- Example: **12 ^ 25 = 21**

$$\begin{array}{rcl}
 12_{10} & = & 00001100_2 \\
 25_{10} & = & 00011001_2 \\
 & & \text{-----} \\
 12 \mid 25 & = & 00010101_2 = 21_{10}
 \end{array}$$

Bitwise Complement: $\sim$

- The bitwise complement ($\sim$) is a unary operator (i.e. acts on one operand). It reverses every binary digit in the operand.

- Bitwise ($\sim$) operator truth table:

Digit 1	Digit 2
0	1
1	0

- Example for an unsigned integer: $\sim 37 = 218$

$$\begin{array}{rcl}
 37_{10} & = & 00100101_2 \\
 & & \text{-----} \\
 \sim 37 & = & 11011010_2 = 218_{10}
 \end{array}$$

Bit Vectors and Sets

- We can use bit vectors (i.e., ordered collections of bits) to represent finite sets, and perform operations such as determining the union, intersection, and complement of multiple sets.
- **Example:** we can represent current courses taken using a **char**.

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

Bit Vectors and Sets

- We can use bit vectors (i.e., ordered collections of bits) to represent finite sets, and perform operations such as determining the union, intersection, and complement of multiple sets.
- Example:** we can represent current courses taken using a **char**.

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367



Bit Vectors and Sets

- How can we determine the union of two sets of courses take?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

Bit Vectors and Sets

- How can we determine the union of two sets of courses take?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

- Use Bitwise OR:

```

00100011
| 01100001
-----
01100011

```

Bit Vectors and Sets

- How can we determine the intersection of two sets of courses take?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

Bit Vectors and Sets

- How can we determine the intersection of two sets of courses take?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

- Use bitwise AND:

```

00100011
& 01100001
-----
00100001

```

Bit Vectors and Sets

- How can we update our bit vector to indicate we have taken CMPT-2167?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

Bit Vectors and Sets

- How can we update our bit vector to indicate we have taken CMPT-2167?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

- Use bitwise OR:

```

00100011
| 01000000
-----
01100011

```

Bitmasking

Bit Masking in C

```
#define CMPT2367 0x1    /* 0000 0001 */
#define CMPT2276 0x2    /* 0000 0010 */
#define CMPT2363 0x4    /* 0000 0100 */
#define CMPT2361 0x8    /* 0000 1000 */
#define CMPT2300 0x10   /* 0001 0000 */
#define CMPT2395 0x20   /* 0010 0000 */
#define CMPT2167 0x40   /* 0100 0000 */
#define CMPT1209 0x80   /* 1000 0000 */
```

```
unsigned char myClasses = 0x15;          /* 0001 0101 */
myClasses = myClasses | CMPT2167;        // Add CMPT-2167
```

Bit Masking in C

```
#define CMPT2367 0x1    /* 0000 0001 */
#define CMPT2276 0x2    /* 0000 0010 */
#define CMPT2363 0x4    /* 0000 0100 */
#define CMPT2361 0x8    /* 0000 1000 */
#define CMPT2300 0x10   /* 0001 0000 */
#define CMPT2395 0x20   /* 0010 0000 */
#define CMPT2167 0x40   /* 0100 0000 */
#define CMPT1209 0x80   /* 1000 0000 */

unsigned char myClasses = 0x15; /* 0001 0101 */
myClasses |= CMPT2167;          // Add CMPT-2167
```


Bit Vectors and Sets

- How can we update our bit vector to indicate we have not taken CMPT-2276?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

Bit Vectors and Sets

- How can we update our bit vector to indicate we have not taken CMPT-2276?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

- Use the bitwise complement along with the bitwise and:

```

00100011
& 11111101
-----
00100001

```

```

unsigned char myClasses = 0x23; /* 0010 0011 */
myClasses = myClasses & ~CMPT2276; // Remove CMPT2276

```

Bit Vectors and Sets

- How can we update our bit vector to indicate we have not taken CMPT-2276?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

- Use the bitwise compliment along with the bitwise and:

```

00100011
& 11111101
-----
00100001

```

```

unsigned char myClasses = 0x23; /* 0010 0011 */
myClasses &= ~CMPT2276;      // Remove CMPT2276

```

Bit Vectors and Sets

- How can we check if we have taken CMPT-2276?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

Bit Vectors and Sets

- How can we check if we have taken CMPT-2276?

0	0	1	0	0	0	1	1
CMPT-1209	CMPT-2167	CMPT-2395	CMPT-2300	CMPT-2361	CMPT-2363	CMPT-2276	CMPT-2367

- Use the bitwise and operator:

```

00100011
& 00000010
-----
00000010

```

```

unsigned char myClasses = 0x23; /* 0010 0011 */
if (myClasses & CMPT2276) {...
    // taken CMPT2276

```

Bitwise Shift: << and >>

- Shift operators are used to move/shift the bit patterns either to the right or to the left.

Operand << *n* (Left-shift)

Operand >> *n* (Right-shift)

- The left-shift operator shifts *n* number of bits to the left which pops out the leftmost bits in the expression. Also, *n* bits with a value of 0 will be filled on the right side.
- Similarly, the right shift operation will shift *n* number of bits to the right which pops out the rightmost bits in the expression. Also *n* bits with a value of 0 will be filled on the left side.

- Example: *x* is an unsigned integer
expression $1111_2 = 15_{10}$

$$x \ll 2 = 1111 \ll 2 = 1100 = 12_{10}$$

$$x \gg 2 = 1111 \gg 2 = 0011 = 3_{10}$$

Bitwise Left-shift Operator: <<

- The left-shift operator shifts a bit vector a certain number of positions to the left. New lower order bits are filled in with 0, and bits leaving the bit representation to the left are lost.

```
x << k;           // returns x shifted to the left by k bits  
x <<= k;          // x = (x << k);
```

- 8-bit examples:

```
00110111 << 2 results in 11011100  
01100011 << 4 results in 00110000  
10010101 << 4 results in 01010000
```

Bitwise Right-shift Operator: >>

The right-shift operator shifts a bit vector a certain number of positions to the right. Bits shifted off the bit representation to the right are lost.

```
x >> k;    // returns x shifted to the right by k bits  
x >>= k;   // x = (x >> k);
```

- How should we fill in new higher-order bits?
 - Why don't we just fill with 0 values...

```
short x = 2;           // 0000 0000 0000 0010  
x >>= 1;               // 0000 0000 0000 0001  
printf("%d\n", x);     // 1
```


Bitwise Right-shift Operator: >>

The right-shift operator shifts a bit vector a certain number of positions to the right. Bits shifted off the bit representation to the right are lost.

```
x >> k;    // returns x shifted to the right by k bits  
x >>= k;   // x = (x >> k);
```

- What about signed integers?
 - Why don't we just fill with 0 values...

```
short x = -2;           // 1111 1111 1111 1110  
x >>= 1;                // 0111 1111 1111 1111  
printf("%d\n", x);      // 32767 🤖
```

Bitwise Right-shift Operator: >>

The right-shift operator shifts a bit vector a certain number of positions to the right. Bits shifted off the bit representation to the right are lost.

```
x >> k;    // returns x shifted to the right by k bits
x >>= k;    // x = (x >> k);
```

- What is the solution?
 - Always fill in with the sign bit...

```
short x = -2;           // 1111 1111 1111 1110
x >>= 1;                 // 1111 1111 1111 1111
printf("%d\n", x);      // -1 😊
```

Bitwise Right-shift Operator: >>

The right-shift operator shifts a bit vector a certain number of positions to the right. Bits shifted off the bit representation to the right are lost.

```
x >> k;    // returns x shifted to the right by k bits
x >>= k;    // x = (x >> k);
```

- What is the solution?
 - Always fill in with the sign bit...

```
short x = 2;           // 0000 0000 0000 0010
x >>= 1;                // 0000 0000 0000 0001
printf("%d\n", x);      // 1 😊
```

Bitwise Right-shift Operator: >>

- We have two kinds of right-shifts:
 - Logical Right-Shift: fill new high-order bits with 0 values.
 - Arithmetic Right-Shift: fill new high-order bits with the sign bit.
- Unsigned numbers are right-shifted using the Logical Right-Shift.
- Signed numbers are right-shifted using Arithmetic Right-Shift.
- Technically, the C standard does not precisely define whether a right-shift for signed integers is logical or arithmetic.
- However, almost all compilers use arithmetic right-shift for signed integers.

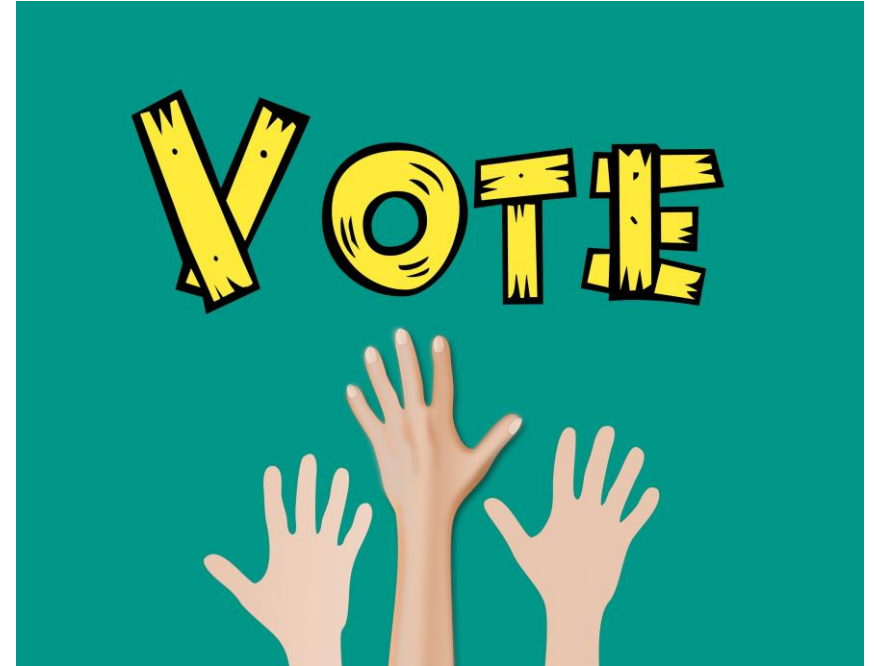
Poll 1 (Extra Credit)

If we want a long int with the index-32 bit equal to 1, we can use the following expression in C.

```
long num = 1 << 32;
```

- a) True
- b) False
- c) It depends...

Please use the “Poll” window to participate for extra credit!





DOUGLAS COLLEGE

Character Representation

Char

- A **char** is a primitive variable type that represents a glyph.

```
char letterB = 'B';  
char plus = '+';  
char zero = '0';  
char space = ' ';  
char newLine = '\n';  
char tab = '\t';  
char singleQuote = '\'';  
char backSlash = '\\';
```

Char and ASCII

- In C, characters are represented by an int value (i.e., its **ASCII** code).
- **ASCII = American Standard Code for Information Interchange**
 - Uppercase letters are sequentially numbered.
 - Lowercase letters are also sequentially numbered.
 - Digits are also sequentially numbered.
 - Lowercase letters are 32 more than their uppercase equivalents.

```
char uppercaseA = 'A'; // ASCII Code 65
```

```
char lowercaseA = 'a'; // ASCII Code 97
```

```
char zeroDigit = '0'; // ASCII Code 48
```


Char and ASCII

- We can take advantage of C representing each char as an integer:

```
bool areEqual = 'A' == 'A';           // true
bool earlierLetter = 'f' < 'c';        // false
char uppercaseB = 'A' + 1;
int diff = 'c' - 'a';                  // 2
int numLettersInAlphabet = 'z' - 'a' + 1;
// or
int numLettersInAlphabet = 'Z' - 'A' + 1;
```

Char and ASCII

- We can take advantage of C representing each char as an integer:

```
// prints out every lowercase character  
for (char ch = 'a'; ch <= 'z'; ch++)  
{  
    printf("%c", ch);  
}
```

Useful `ctype.h` Functions

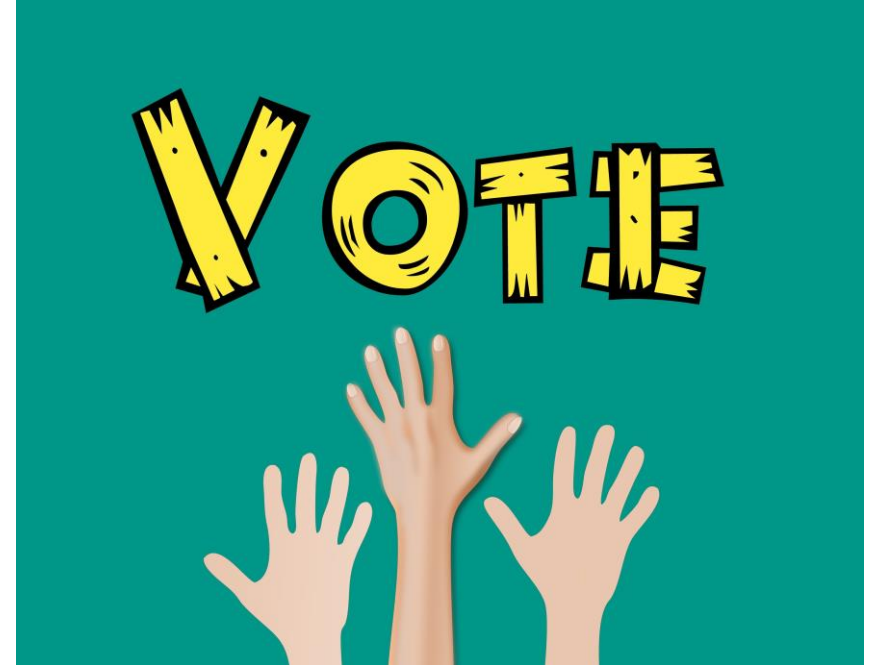
Function	Description
<code>isalpha(<i>ch</i>)</code>	true if <i>ch</i> is 'a' through 'z' or 'A' through 'Z'
<code>islower(<i>ch</i>)</code>	true if <i>ch</i> is 'a' through 'z'
<code>isupper(<i>ch</i>)</code>	true if <i>ch</i> is 'A' through 'Z'
<code>isspace(<i>ch</i>)</code>	true if <i>ch</i> is a space, tab, new line, etc.
<code>isdigit(<i>ch</i>)</code>	true if <i>ch</i> is '0' through '9'
<code>toupper(<i>ch</i>)</code>	returns uppercase equivalent of a letter
<code>tolower(<i>ch</i>)</code>	returns lowercase equivalent of a letter

Poll 2 (Extra Credit)

The `ctype.h` functions in the previous slide can modify an existing char variable.

- a) True
- b) False
- c) Not sure...

Please use the “Poll” window to participate for extra credit!





DOUGLAS COLLEGE

C-Strings

Char and ASCII

- C has no dedicated variable type for strings.
- Strings are represented as arrays of characters with a special character:

"Hello"	<i>index</i>	0	1	2	3	4	5
	<i>char</i>	'H'	'e'	'l'	'l'	'o'	'\0'

- Character `'\0'` is the null-terminating character.
- We always need to allocate one extra space in an array for it.

Char and ASCII

- In C, strings are not objects, so they do not contain any attributes or behavior. As a result, any information regarding their length, for example, must be calculated.
- We can use the provided **strlen()** function to calculate string length.
- The null-terminating character does not count towards the length.
- Function **strlen()** is **O(N)**, meaning the number of operations it requires is on the same order of magnitude as the length of the string.
- This makes the **strlen()** function an expensive function to call, so we should store the length if we plan to use it later rather than recall the function.

```
int length = strlen(myStr); // returns 13 for Hello, World!
```

Passing C-Strings as Parameters

- In C, we pass strings to functions as parameters using **char ***.
- This passes the address of the zeroth element instead of a copy of the entire array of characters.

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

int changeString(char* str)
{
    str[2] = 'C';
    printf("%s\n", str);
}

void main()
{
    char myString[] = "Hello";

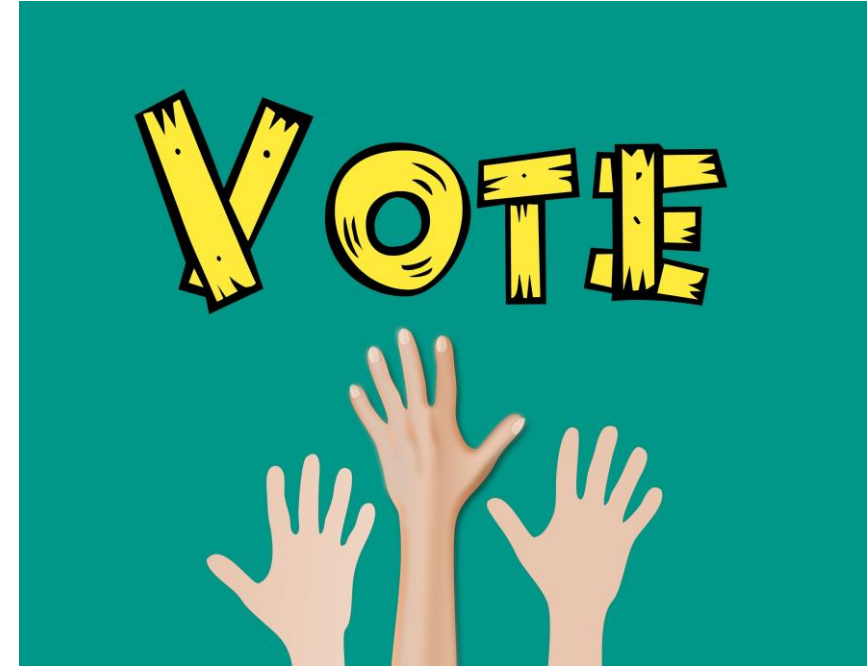
    changeString(&myString);
}
```


Poll 3 (Extra Credit)

In the previous example code, we can remove the address-of operator & in `changeString(&myString);`

- a) True
- b) False
- c) Not sure...

Please use the “Poll” window to participate for extra credit!

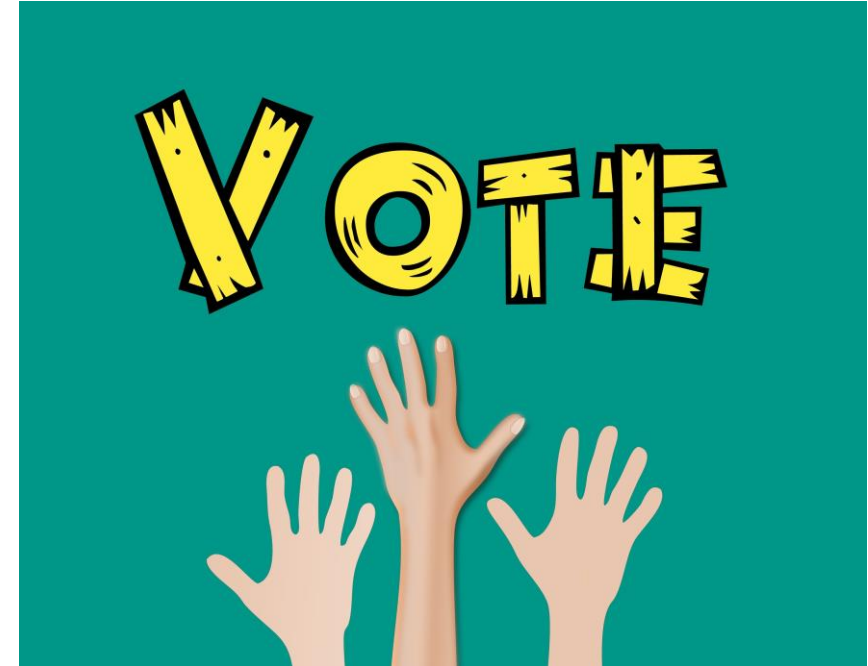


Poll 4 (Extra Credit)

In the previous example code, we can replace `changeString(char* str)` with `changeString(char str[])`.

- a) True
- b) False
- c) Not sure...

Please use the “Poll” window to participate for extra credit!



Passing C-Strings as Parameters

- In C, we pass strings to functions as parameters using **char ***.
- This passes the address of the zeroth element instead of a copy of the entire array of characters.

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

int changeString(char str[])
{
    str[0] = 'C';
    printf("%s\n", str);
}

void main()
{
    char myString[] = "Hello";

    changeString(myString);
}
```

Poll 5 (Extra Credit)

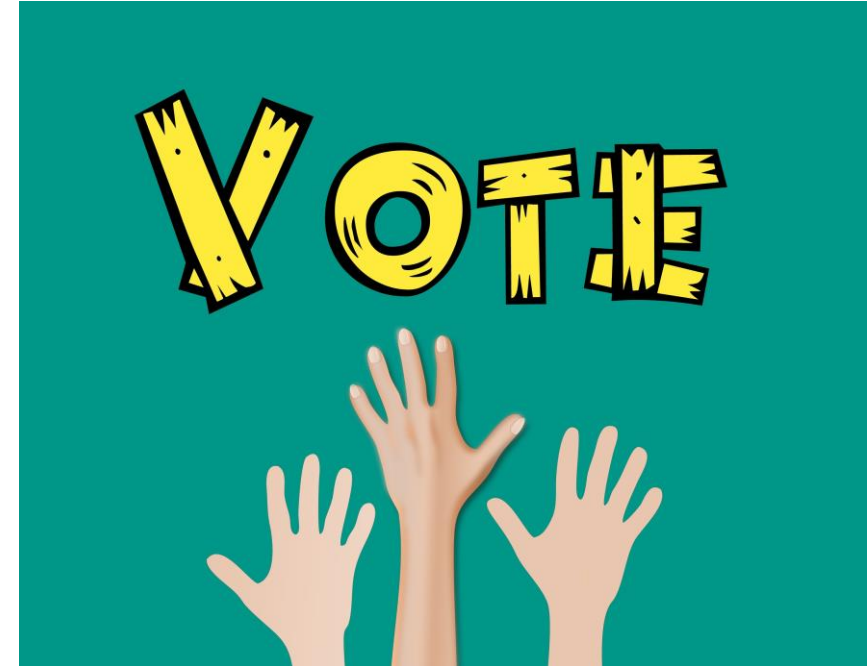
The following if-statement

```
void doSomething(char *str1, char *str2) {  
    if (str1 > str2) {
```

will compare the alphabetical order between str1 and str2.

- a) True
- b) False
- c) Not sure...

Please use the “Poll” window to participate for extra credit!



Useful string.h Functions

Function	Description
<code>strlen(<i>str</i>)</code>	returns the count of chars in a C string (before null-terminating character).
<code>strcmp(<i>str1</i>, <i>str2</i>)</code> , <code>strncmp(<i>str1</i>, <i>str2</i>, <i>n</i>)</code>	compares two strings; returns 0 if identical, negative if <i>str1</i> comes before <i>str2</i> in alphabet, positive if <i>str1</i> comes after <i>str2</i> in alphabet. <i>strncmp</i> stops comparing after at most <i>n</i> characters.
<code>strchr(<i>str</i>, <i>ch</i>)</code> <code>strrchr(<i>str</i>, <i>ch</i>)</code>	character search: returns a pointer to the first occurrence of <i>ch</i> in <i>str</i> , or <i>NULL</i> if <i>ch</i> was not found in <i>str</i> . <i>strrchr</i> finds the last occurrence.
<code>strstr(<i>haystack</i>, <i>needle</i>)</code>	string search: returns a pointer to the start of the first occurrence of <i>needle</i> in <i>haystack</i> , or <i>NULL</i> if <i>needle</i> is not found in <i>haystack</i> .
<code>strcpy(<i>dst</i>, <i>src</i>)</code> , <code>strncpy(<i>dst</i>, <i>src</i>, <i>n</i>)</code>	copies characters in <i>src</i> to <i>dst</i> , including null-terminating character. Assumes enough space in <i>dst</i> . <i>strncpy</i> stops after at most <i>n</i> chars, and <u>does not</u> add null-terminating char.
<code>strcat(<i>dst</i>, <i>src</i>)</code> , <code>strncat(<i>dst</i>, <i>src</i>, <i>n</i>)</code>	concatenate <i>src</i> onto the end of <i>dst</i> . <i>strncat</i> stops concatenating after at most <i>n</i> characters. Always adds a null-terminating character.
<code>strspn(<i>str</i>, <i>accept</i>)</code> , <code>strcspn(<i>str</i>, <i>reject</i>)</code>	<i>strspn</i> returns the length of the initial part of <i>str</i> which contains <u>only</u> characters in <i>accept</i> . <i>strcspn</i> returns the length of the initial part of <i>str</i> which does <u>not</u> contain any characters in <i>reject</i> .

Poll 6 (Extra Credit)

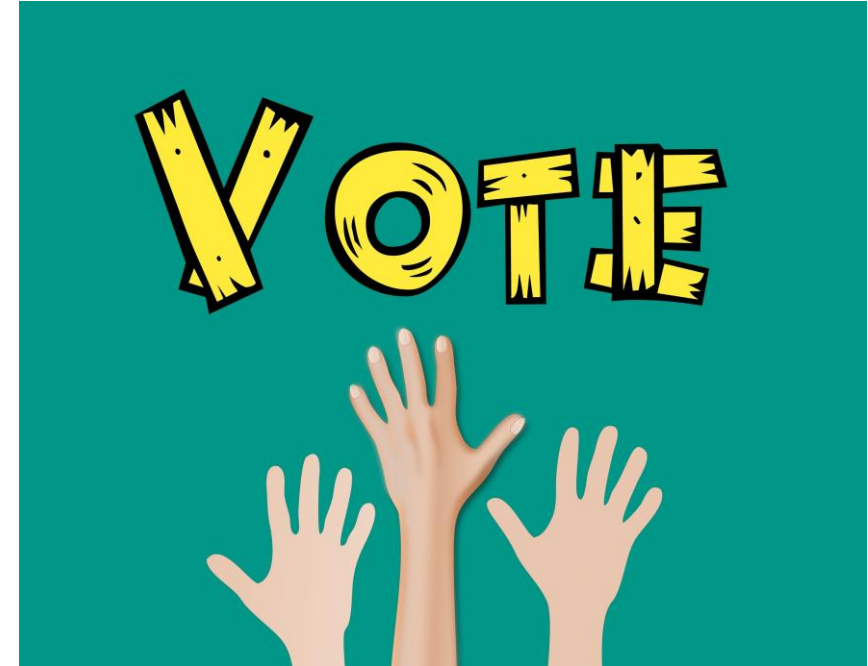
The following code snippet will lead to a compilation error.

```
#include <stdio.h>
#include <stdbool.h>
void main()
{
    char* myString = "Hello";

    myString[0] = 'C';
}
```

- a) True
- b) False
- c) Not sure...

Please use the “Poll” window to participate for extra credit!



References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2395

Intro to Computer Architecture

Lecture 4



Plan for Today

- The Basic Instruction Cycle
- Interrupts and Exceptions
- I/O Function
- Interconnection Structures

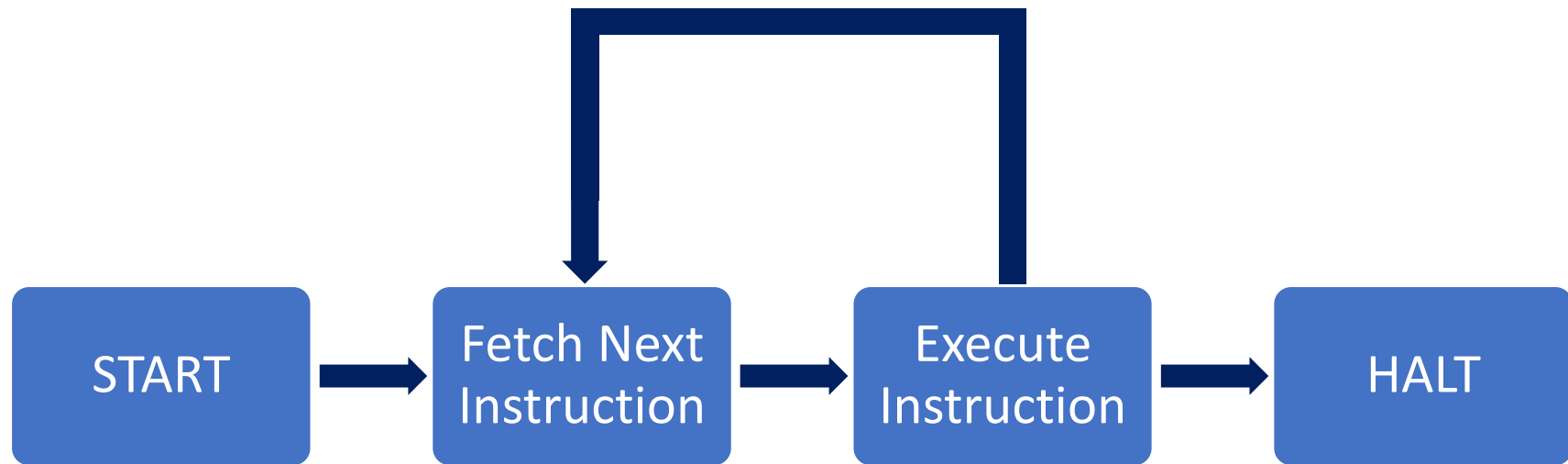


Instruction Cycles

The Basic Instruction Cycle

- The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory.
- The processor does the actual work by executing instructions specified in the program.
- In its simplest form, instruction processing consists of two steps:
 - The processor reads (fetches) instructions from memory one at a time.
 - The processor executes each instruction.
- Program execution consists of repeating the process of instruction fetch and instruction execution.
- The instruction execution may involve several operations and depends on the nature of the instruction.
- The processing required for a single instruction is called an instruction cycle.
- The two steps are referred to as the fetch cycle and the execute cycle.
- Program execution halts **only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.**

The Basic Instruction Cycle



The Fetch Cycle

- At the beginning of each instruction cycle, the processor fetches an instruction from memory.
- In a typical processor, a register called the program counter (PC) holds the address of the instruction to be fetched next.
- Unless told otherwise, **the processor always increments the PC after each instruction fetch**, so that it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address).
- The fetched instruction is loaded (indirectly using MAR and MBR) into a register in the processor known as the instruction register (IR).
- The instruction contains bits that specify the action the processor is to take.
- The processor interprets the instruction and performs the required action.

Action Types

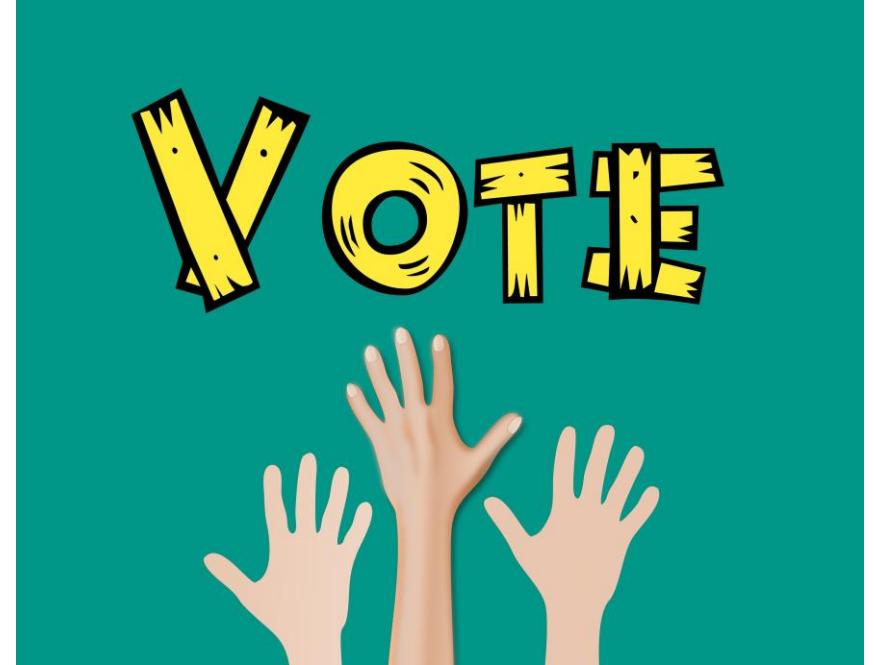
- In general, these actions fall into four categories:
 1. **Processor-memory:** Data may be transferred from processor to memory or from memory to processor.
 2. **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
 3. **Data processing:** The processor may perform some arithmetic or logic operation on data.
 4. **Control:** An instruction may specify that the sequence of execution be altered.
 - For example, the processor may fetch an instruction from location 149, which specifies that the next instruction be from location 182. The processor will remember this fact by setting the program counter to 182. Thus, on the next fetch cycle, the instruction will be fetched from location 182 rather than 150.
- An instruction's execution may involve a combination of these actions.

Poll 1 (Extra Credit)

In C programming, which one of the following will eventually get translated to a “Control” action?

- a) If-else-statements and breaks
- b) Loops
- c) Both a and b
- d) None of the above

Please use the “Poll” window to participate for extra credit!



A Simple Example

- Consider a simple example using a hypothetical machine that includes these characteristics:
 - The processor contains a single data register, called an **accumulator (AC)**.
 - Both instructions and data are 16 bits long. Thus, it is convenient to organize memory using **16-bit words**.
 - The instruction format provides 4 bits for the operation code (**opcode**), so that there can be as many as $2^4 = 16$ different opcodes, and up to $2^{12} = 4096$ words of memory can be directly addressed.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Opcode				Address											

Instruction Format

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sign	Magnitude														

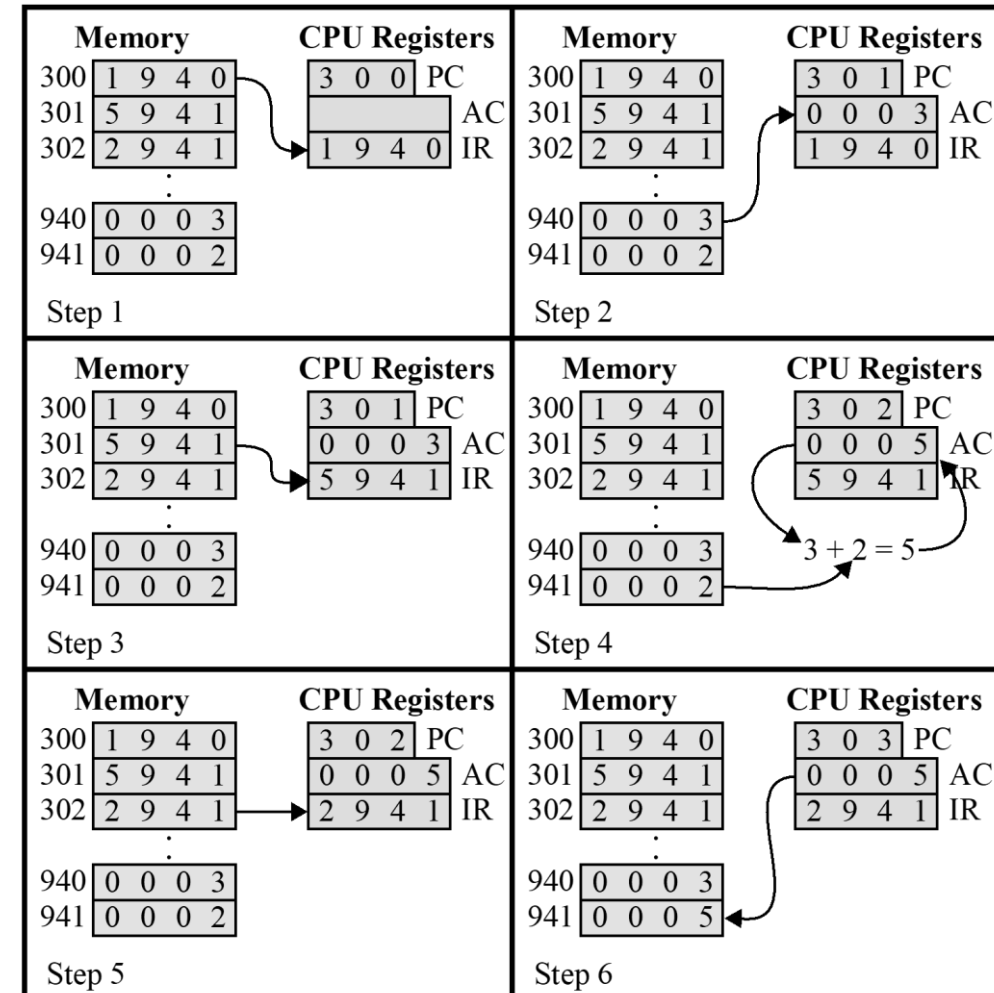
Integer Format

Registers	Opcode	Operation
Program Counter (PC)	0001	Load AC from Memory
Instruction Register (IR)	0010	Store AC to Memory
Accumulator (AC)	0101	Add to AC from Memory

Some Opcodes

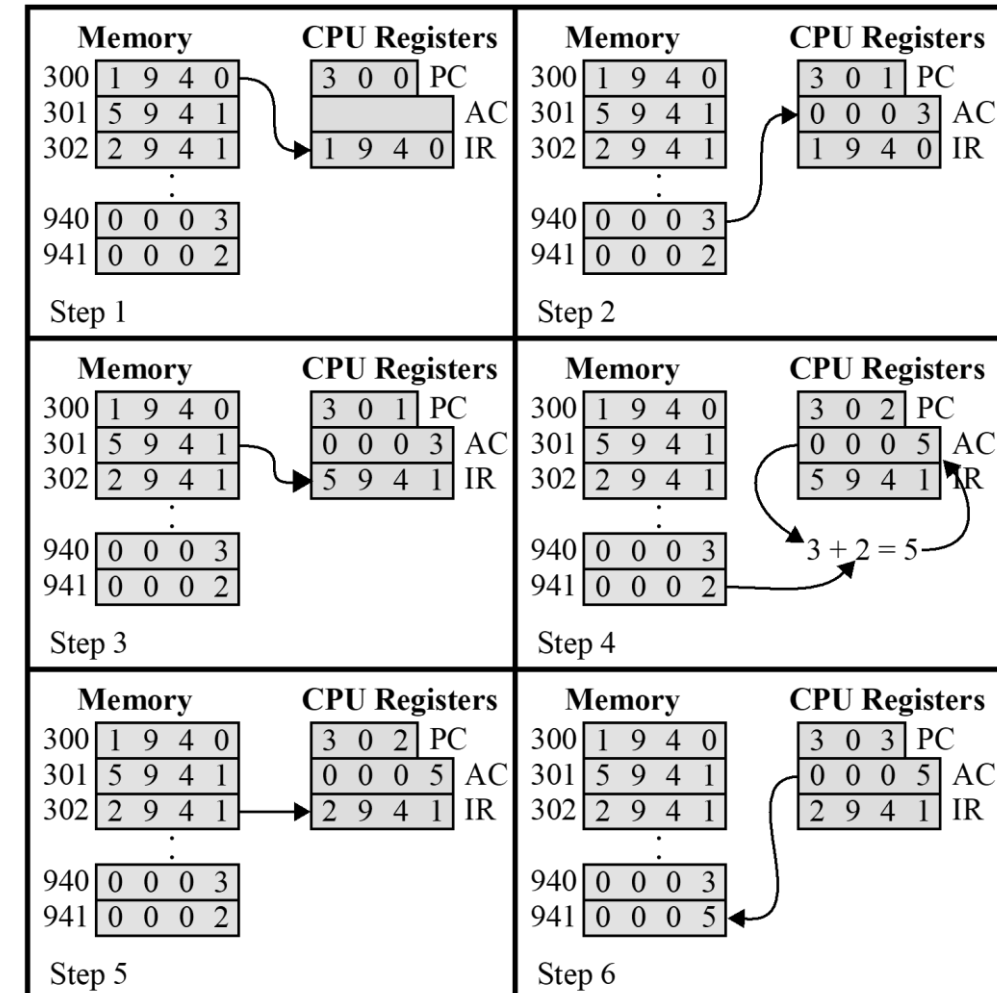
A Simple Example

- This code snippet adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location.
- Three instructions, which can be described as three fetch and three execute cycles, are required:
 - The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IR, and the PC is incremented. Note that this process involves the use of a memory address register and a memory buffer register. For simplicity, these intermediate registers are ignored.



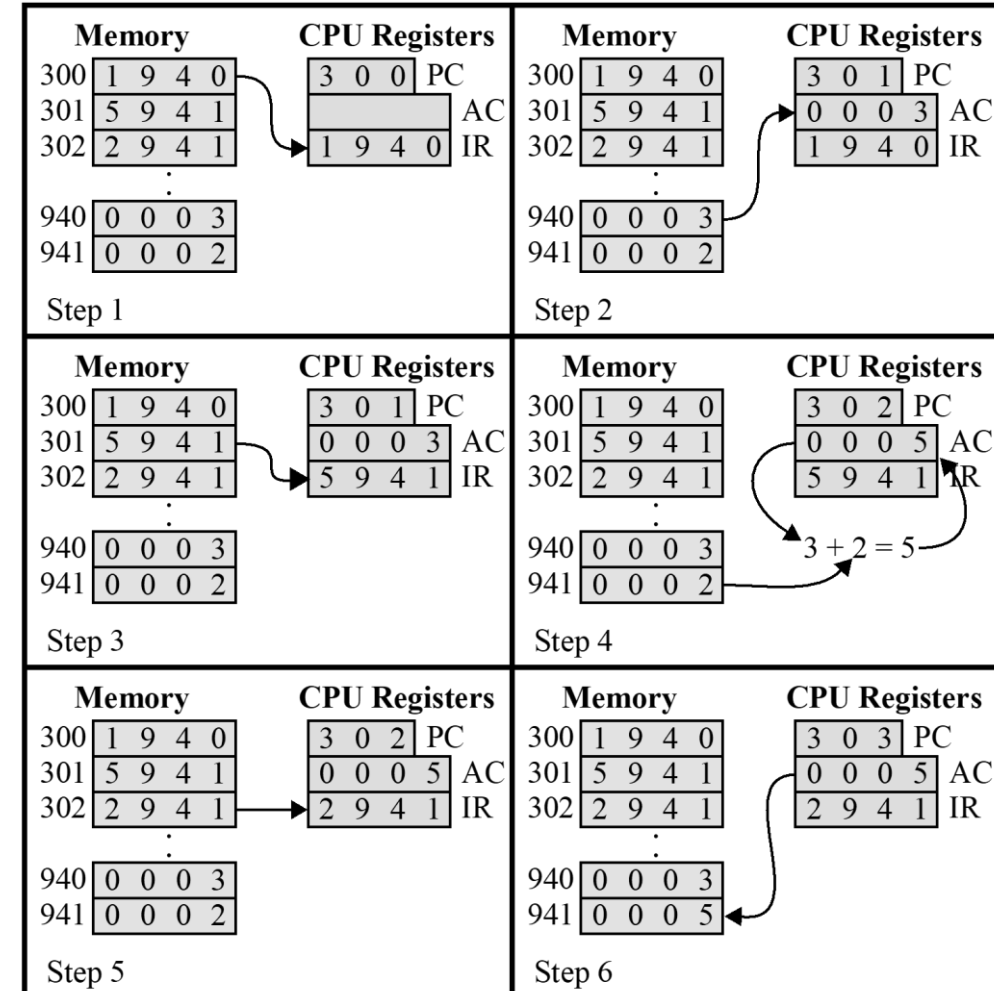
A Simple Example

- The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded.
- The next instruction (5941) is fetched from location 301, and the PC is incremented.
- The old contents of the AC and the contents of location 941 are added, and the result is stored in the AC.
- The next instruction (2941) is fetched from location 302, and the PC is incremented.
- The contents of the AC are stored in location 941.



A Simple Example

- In this example, three instruction cycles, each consisting of a fetch cycle and an execute cycle, are needed to add the contents of location 940 to the contents of 941.
- With a more complex set of instructions, fewer cycles would be needed.
- Some older processors, for example, included instructions that contain more than one memory address.
- Thus, the execution cycle for a particular instruction on such processors could involve more than one reference to memory.
- Also, instead of memory references, an instruction may specify an I/O operation.

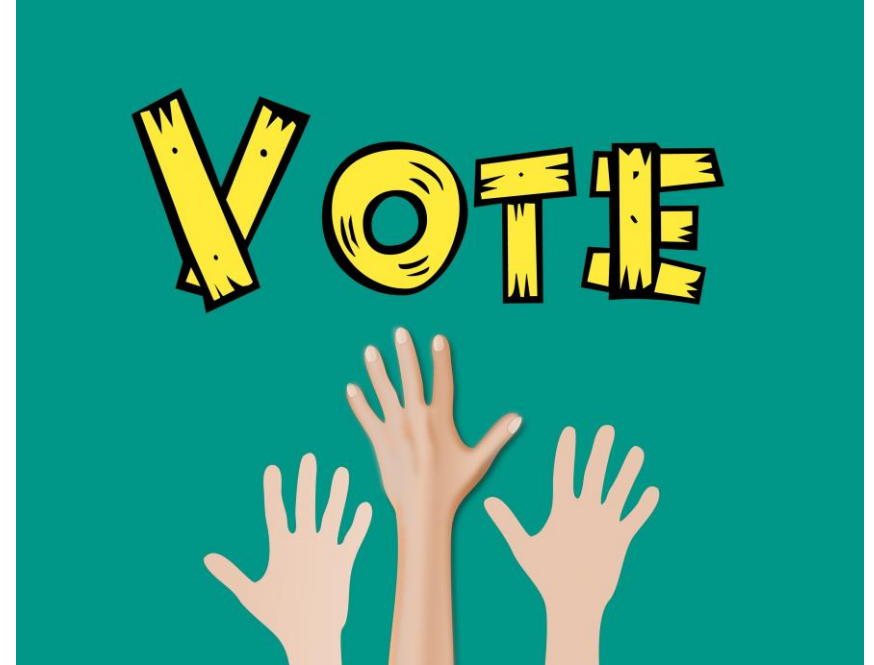


Poll 2 (Extra Credit)

In the previous example, if memory were organized using 8-bit words, what would we need to add to the PC to locate the next instruction to be executed?

- a) 1 (i.e., the same increment as before)
- b) 2
- c) Not sure...

Please use the “Poll” window to participate for extra credit!





DOUGLAS COLLEGE

Interrupts (and Exceptions)

Interrupt Classes

- All computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal processing of the processor.
- The following table lists the most common classes of interrupts.
- We need to introduce the concept now to understand more clearly the nature of the instruction cycle and the implications of interrupts on the interconnection structure.

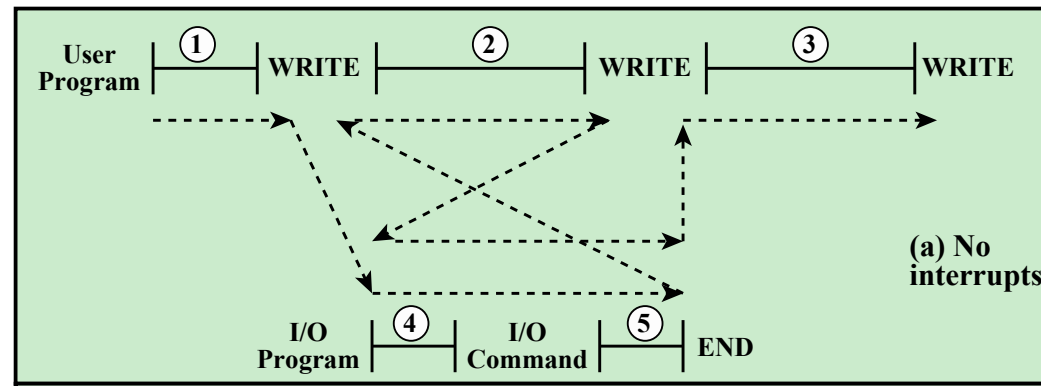
Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation, request service from the processor, or to signal a variety of error conditions.
Hardware Failure	Generated by a failure such as power failure or memory parity error.

Interrupts

- **Interrupts** are provided primarily as a way **to improve processing efficiency**.
- For example, most external devices are much slower than the processor.
- Suppose that the processor is transferring data to a printer using the instruction cycle scheme shown on slide 5.
- After each write operation, the processor must pause and remain idle until the printer catches up.
- The length of this pause may be on the order of many hundreds or even thousands of instruction cycles that do not involve memory.
- Clearly, this is a very wasteful use of the processor.

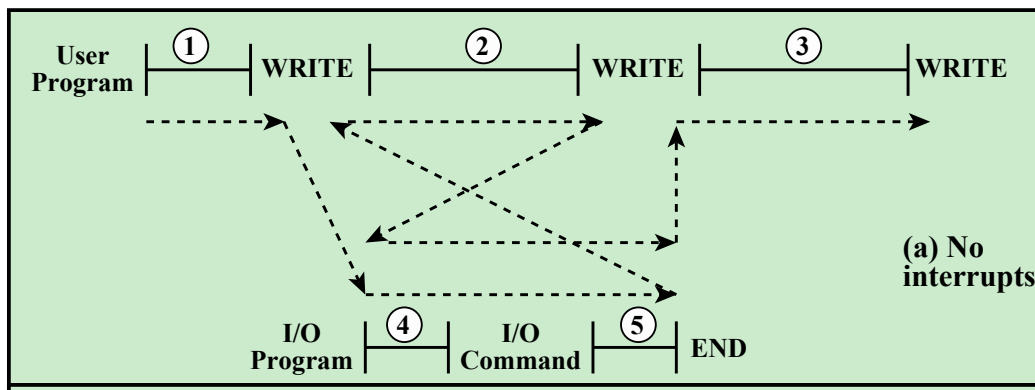
Without Interrupts...

- The user program performs a series of WRITE calls interleaved with processing.
- Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O.
- The WRITE calls are to an I/O program that is a system utility and that will perform the actual I/O operation.
- The I/O program consists of three sections:
 1. A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation.
 2. The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically poll the device). The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.
 3. A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.



Without Interrupts...

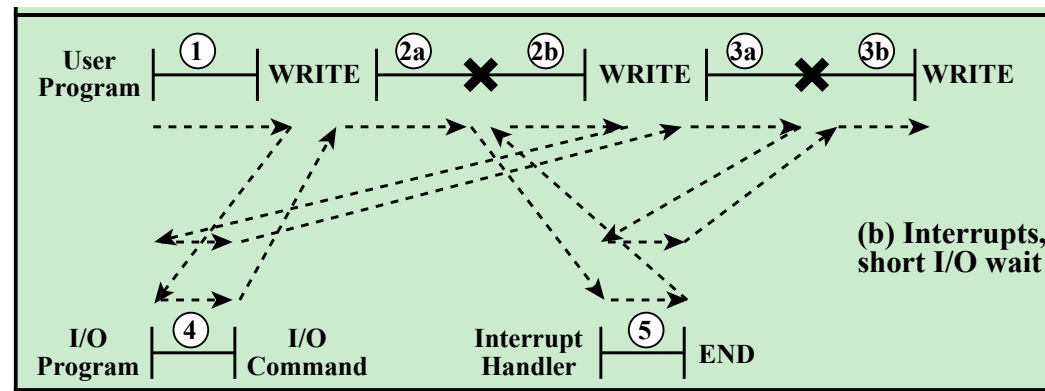
- The user program performs a series of WRITE calls interleaved with processing.
- Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O.
- The WRITE calls are to an I/O program that is a system utility and that will perform the actual I/O operation.
- The I/O program consists of three sections:
 1. A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation.
 2. The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically poll the device). The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.
 3. A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.



Because the I/O operation may take a relatively long time to complete, the I/O program is hung up waiting for the operation to complete; hence, the user program is stopped at the point of the WRITE call for some considerable period of time. 🤪

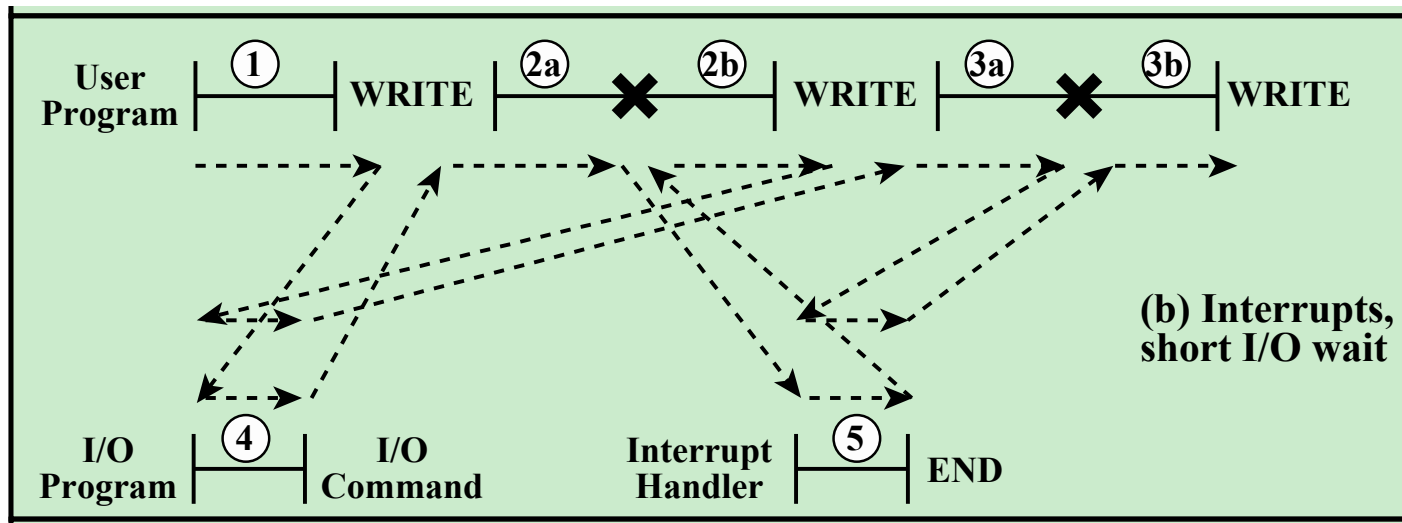
With Interrupts...

- With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress.
- As before, the user program reaches a point at which it makes a system call in the form of a WRITE call.
- The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command.
- After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it.
- This I/O operation is conducted concurrently with the execution of instructions in the user program.



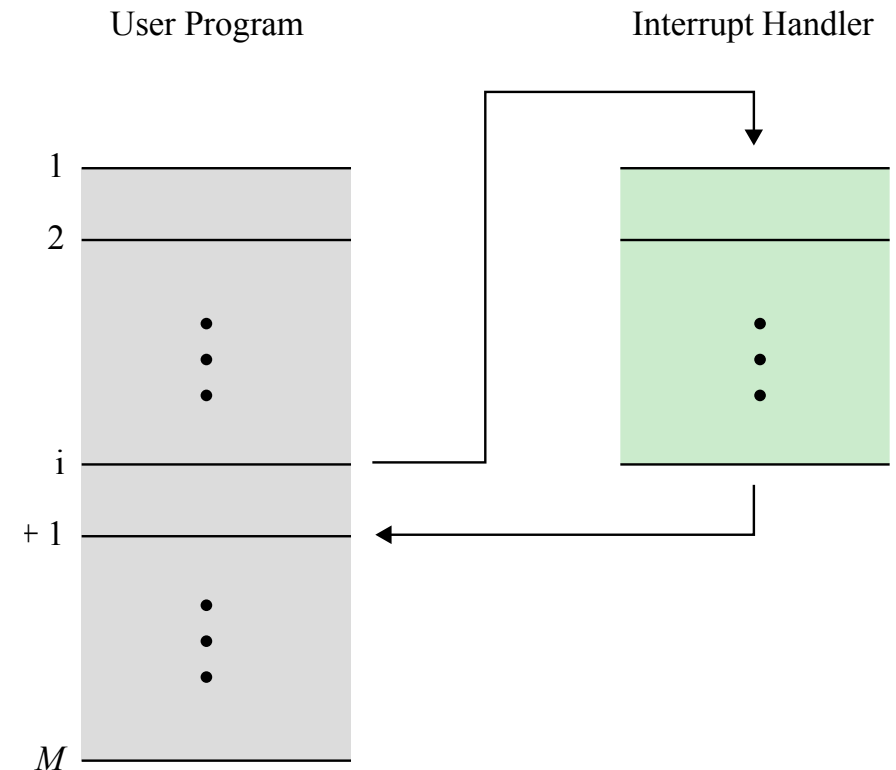
With Interrupts...

- When the external device becomes ready to be serviced—that is, when it is ready to accept more data from the processor—the I/O module for that external device sends an interrupt request signal to the processor.
- The processor responds by suspending operation of the current program, branching off to a program to service that particular I/O device, known as an **interrupt handler**, and resuming the original execution after the device is served.
- The points at which such interrupts occur are indicated by an **X** below.



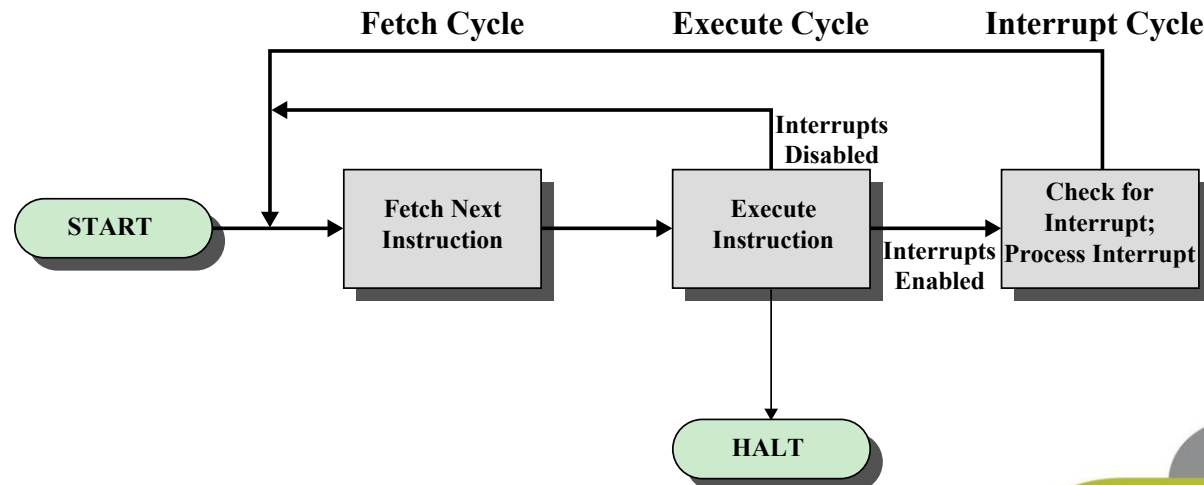
Interrupts

- From the point of view of the user program, an interrupt is just an interruption of the normal sequence of execution.
- When the interrupt processing is completed, execution resumes. Thus, the user program does not have to contain any special code to accommodate interrupts.
- The processor and the operating system are responsible for suspending the user program and then resuming it at the same point.



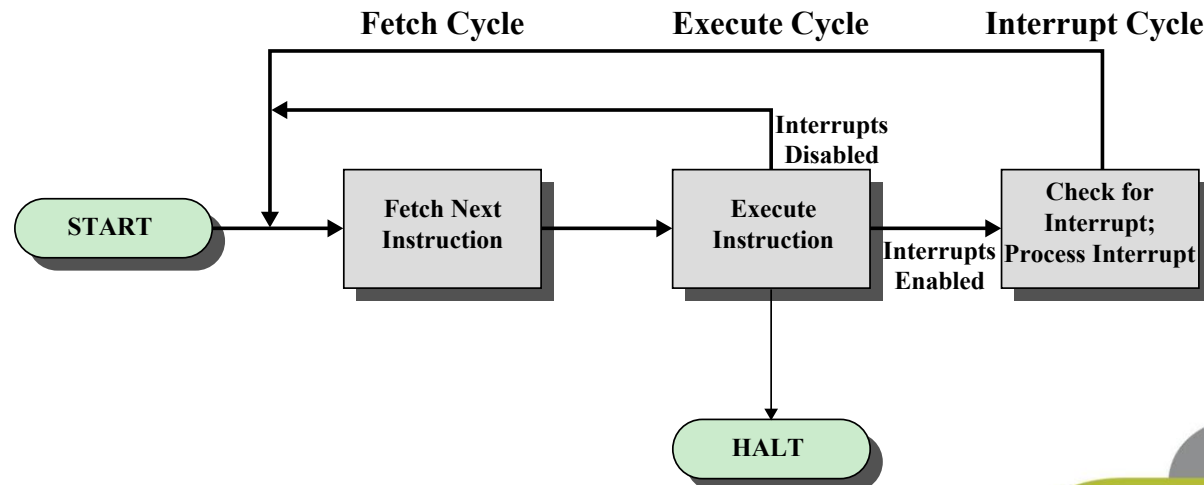
Interrupts

- To accommodate interrupts, an interrupt cycle is added to the instruction cycle, as shown below.
- In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal.
- If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program.
- If an interrupt is pending, the processor does the following:
 1. It suspends execution of the current program being executed and saves its context. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.
 2. It sets the program counter to the starting address of an interrupt handler routine.



Interrupts

- The processor now proceeds to the fetch cycle and fetches the first instruction in the interrupt handler program, which will service the interrupt.
- **The interrupt handler program is generally part of the operating system.** Typically, this program determines the nature of the interrupt and performs whatever actions are needed.
- In the example we have been using, the handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module.
- When the interrupt handler routine is completed, the processor can resume execution of the user program at the point of interruption.

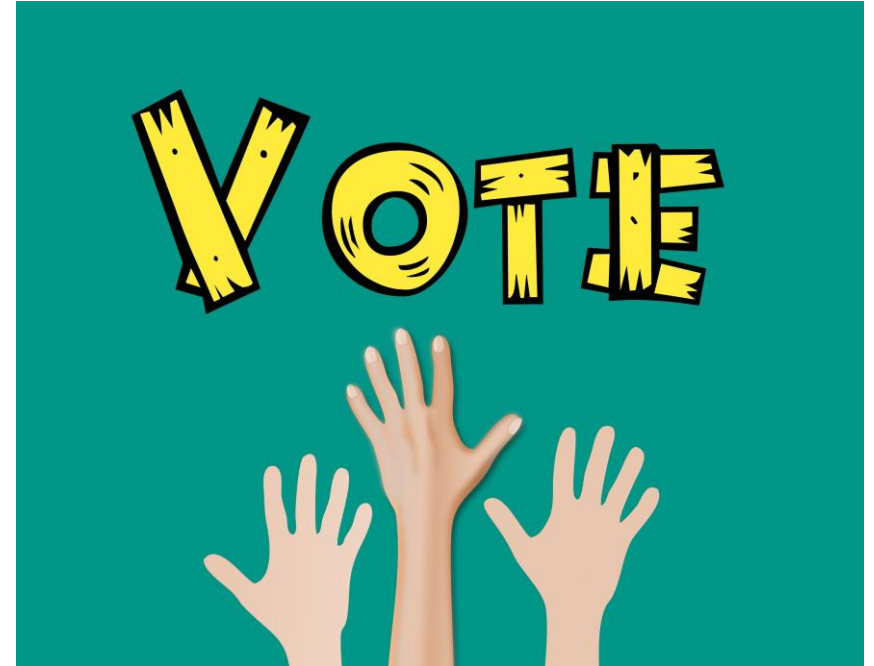


Poll 2 (Extra Credit)

The use of interrupts does not add any additional overhead. This is why having interrupts makes the use of the processor more efficient than not having interrupts.

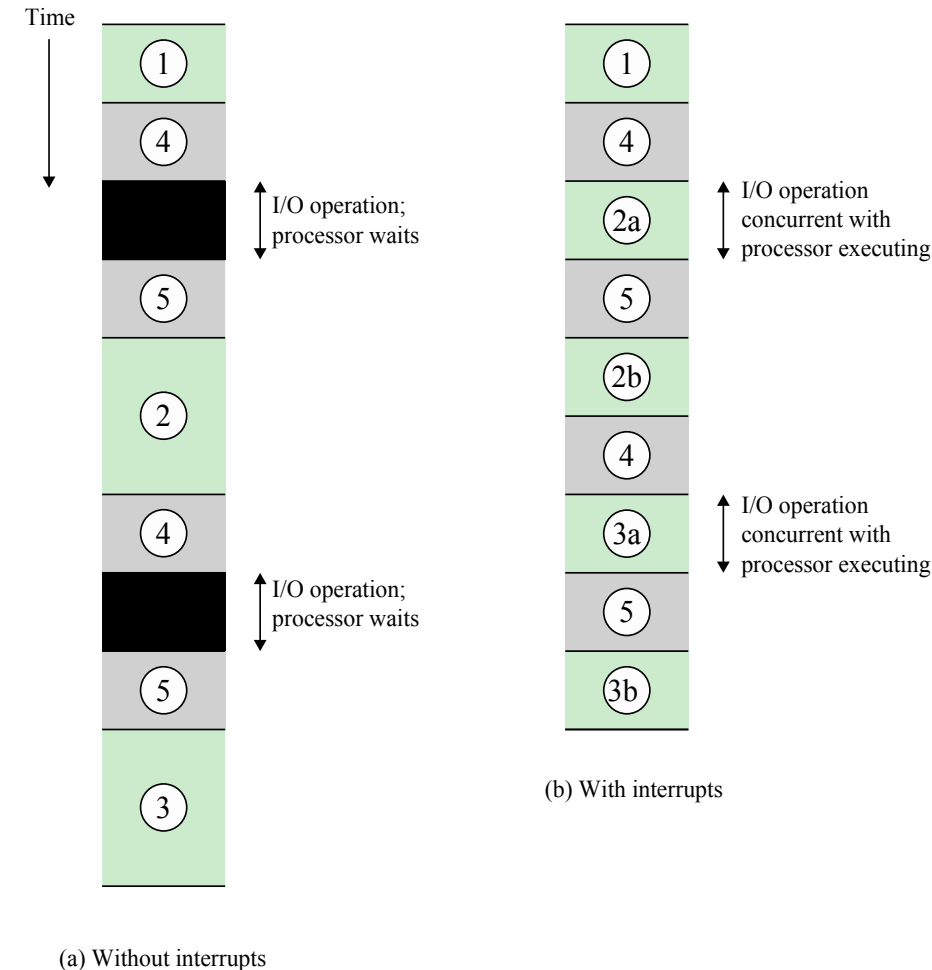
- a) True
- b) False
- c) Not sure...

Please use the “Poll” window to participate for extra credit!

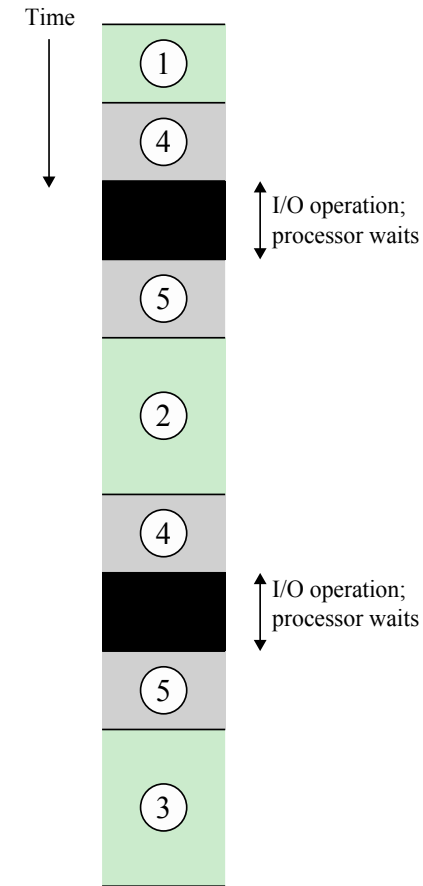
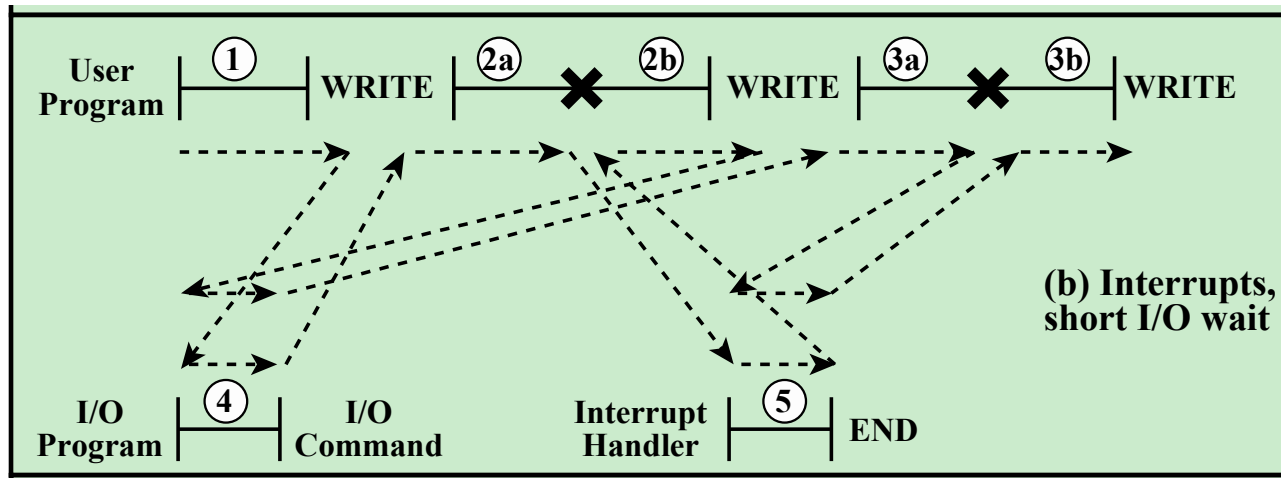
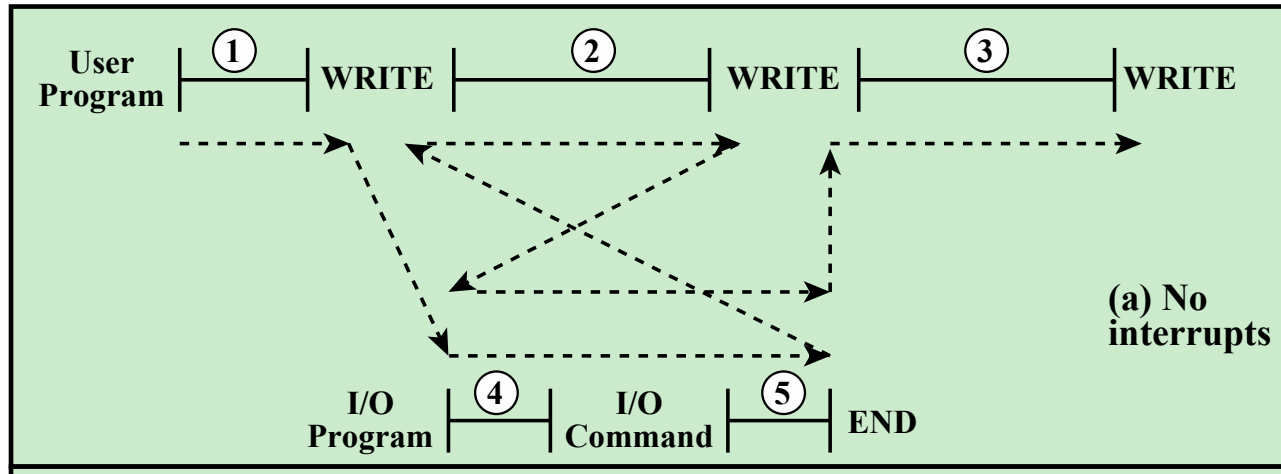


Interrupts

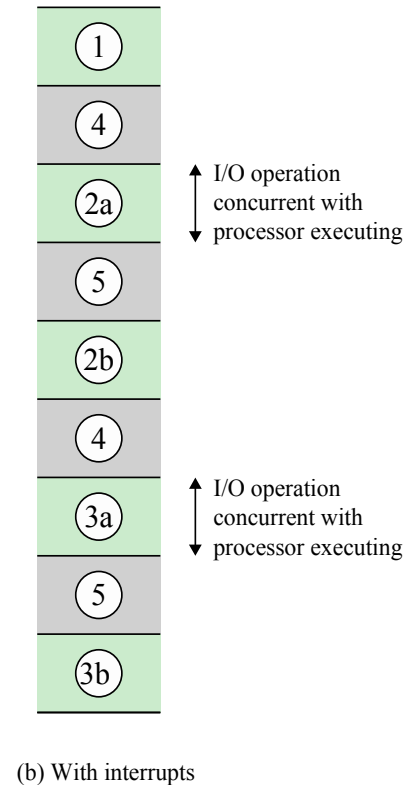
- To appreciate the gain in efficiency, let us review this timing diagram based on the flow of control we saw previously ➔
- Here, user program code segments are shaded green, and I/O program code segments are shaded gray.
- Diagram (a) shows the case in which interrupts are not used. The processor must wait while an I/O operation is performed.
- Both figures assume that the time required for the I/O operation is relatively short (less than the time to complete the execution of instructions between write operations in the user program).
- With interrupts, the segment of code labeled code segment 2 is interrupted.
- A portion of the code (2a) executes (while the I/O operation is performed) and then the interrupt occurs (upon the completion of the I/O operation).
- After the interrupt is serviced, execution resumes with the remainder of code segment 2 (2b).



Interrupts



(a) Without interrupts



(b) With interrupts

Handling Multiple Interrupts

- Two approaches can be taken to dealing with multiple interrupts.
- The first is to disable interrupts while an interrupt is being processed. A disabled interrupt means that the processor can and will ignore that interrupt request signal.
- If an interrupt occurs during this time, it remains pending and will be checked by the processor after the processor has enabled interrupts. Thus, when a user program is executing and an interrupt occurs, interrupts are disabled immediately.
- After the interrupt handler routine completes, interrupts are enabled before resuming the user program, and the processor checks to see if additional interrupts have occurred.
- This approach is nice and simple, as interrupts are handled in strict sequential order.

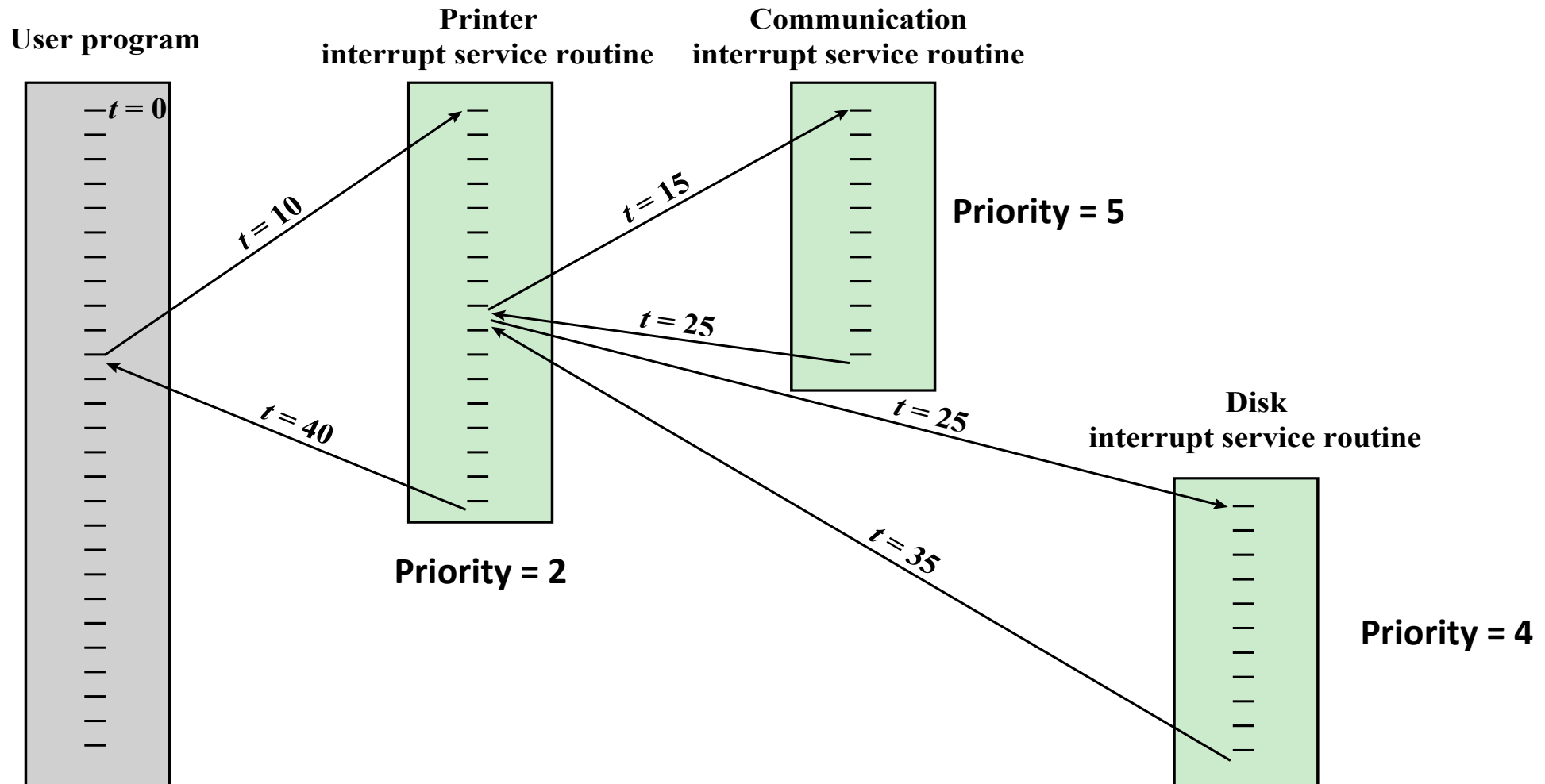
Handling Multiple Interrupts

- A second approach is to define priorities for interrupts and to **allow an interrupt of higher priority to cause a lower-priority interrupt handler to be itself interrupted.**
- As an example of this second approach, consider a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively.
- A user program begins at $t = 0$. At $t = 10$, a printer interrupt occurs; user information is placed on the system stack and execution begins for the printer interrupt using an **interrupt service routine (ISR).**
- While this routine is still executing, at $t = 15$, a communications interrupt occurs. Because the communications line has higher priority than the printer, the communication interrupt must be served now. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues for the communications ISR.

Handling Multiple Interrupts

- While this routine is executing, a disk interrupt occurs ($t = 20$). Because this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion.
- When the communications ISR is complete ($t = 25$), the previous processor state is restored, which is the execution of the printer ISR.
- However, before even a single instruction in that routine can be executed, the processor honors the higher priority disk interrupt and control transfers to the disk ISR.
- Only when that routine is complete ($t = 35$) is the printer ISR resumed. When that routine completes ($t = 40$), control finally returns to the user program.

Handling Multiple Interrupts





DOUGLAS COLLEGE

Input/output (I/O) Function

I/O Function

- An I/O module (e.g., a disk controller) can exchange data directly with the processor.
- Just as the processor can initiate a read or write with memory, designating the address of a specific location, the processor can also read data from or write data to an I/O module.
- In this latter case, the processor identifies a specific device that is controlled by a particular I/O module.
- Thus, an instruction sequence similar to what we have seen for memory could occur, with I/O instructions rather than memory-referencing instructions.

I/O Function

- In some cases, it is desirable to allow I/O exchanges to occur directly with memory.
- In such a case, the processor grants to an I/O module the authority to read from or write to memory, so that the I/O-memory transfer can occur without tying up the processor.
- During such a transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange.
- This operation is known as **direct memory access (DMA)** and is examined later in our course.



Interconnections

Interconnections

- A computer consists of a set of components or modules of three basic types (processor, memory, I/O) that communicate with each other.
- In essence, a computer is a network of basic modules. Thus, there must be paths for connecting the modules.
- The collection of paths connecting the various modules is called **the interconnection structure**.
- The design of this structure will depend on the exchanges that must be made among modules.

Interconnections

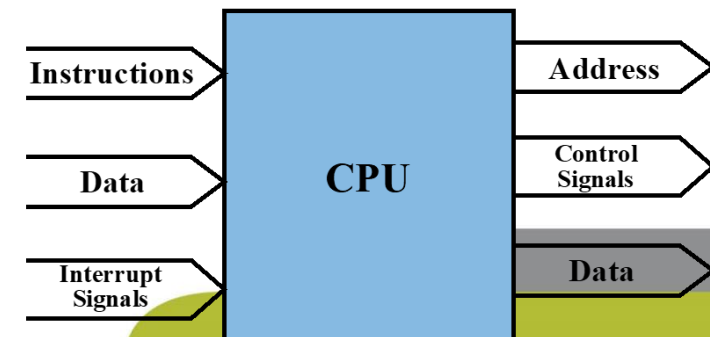
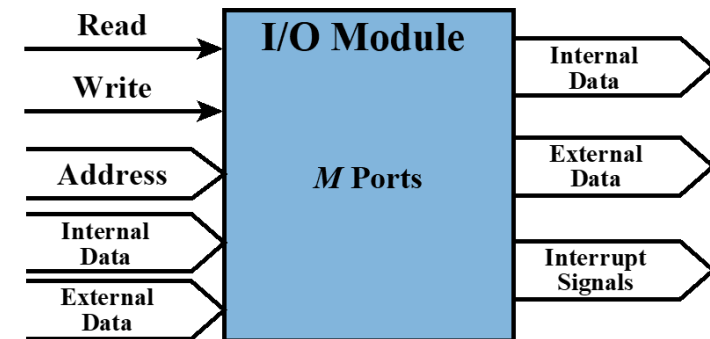
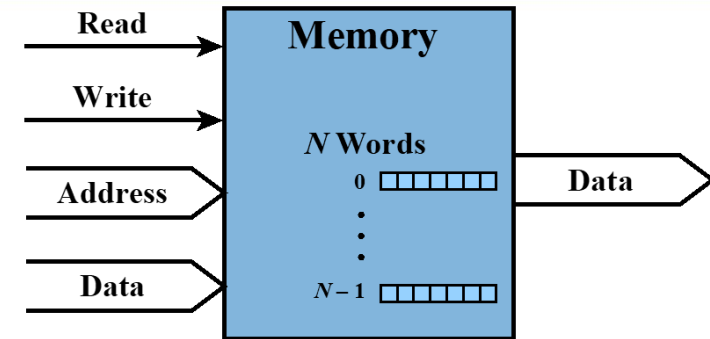
- The types of exchanges that are needed by indicating the major forms of input and output for each module type are:
 1. **Memory**: typically, a memory module will consist of N words of equal length. Each word is assigned a unique numerical address $(0, 1, \dots, N - 1)$. A word of data can be read from or written into the memory. The nature of the operation is indicated by read and write control signals. The location for the operation is specified by an address.

Interconnections

- The types of exchanges that are needed by indicating the major forms of input and output for each module type are:
 2. **I/O module**: From an internal (to the computer system) point of view, I/O is functionally similar to memory. There are two operations, read and write. Also, an I/O module may control more than one external device. We can refer to each of the interfaces to an external device as a port and give each a unique address (e.g., 0, 1, ..., $M - 1$). In addition, there are external data paths for the input and output of data with an external device. Finally, an I/O module may be able to send interrupt signals to the processor.

Interconnection Structure

- The types of exchanges that are needed by indicating the major forms of input and output for each module type are:
- 3. Processor:** The processor reads in instructions and data, writes out data after processing, and uses control signals to control the overall operation of the system. It also receives interrupt signals.



Interconnection Structure

The interconnection structure must support the following types of transfers:

Memory to Processor

Processor reads an instruction or a unit of data from memory.

Processor to Memory

Processor writes a unit of data to memory.

I/O to Processor

Processor reads data from an I/O device via an I/O module.

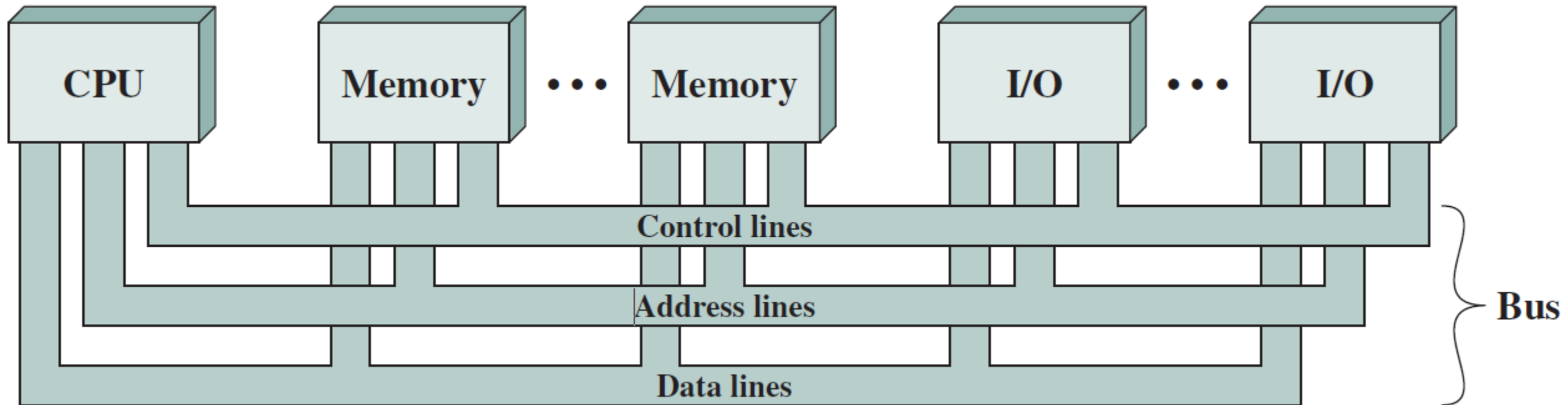
Processor to I/O

Processor sends data to the I/O device.

I/O to or from Memory

An I/O module is allowed to exchange data directly with memory without going through the processor using direct memory access.

The Bus Interconnection



The Bus Interconnection

- Over the years, a number of interconnection structures have been tried. By far the most common are:
 - The bus and various multiple-bus structures; and
 - Point-to-point interconnection structures with packetized data transfer.
- The bus was the dominant means of computer system component interconnection for decades.
- For general-purpose computers, it has gradually given way to various point-to-point interconnection structures, which now dominate computer system design.
- However, bus structures are still commonly used for embedded systems, particularly microcontrollers.

The Bus Interconnection

- A bus is a communication pathway connecting two or more devices.
- A key characteristic of a bus is that it is a **shared transmission medium**.
- Multiple devices connect to the bus, and a signal transmitted by any one device is available for reception by all other devices attached to the bus.
- If two devices transmit during the same time period, their signals will overlap and become garbled.
- Thus, only one device at a time can successfully transmit.
- Typically, a bus consists of multiple communication pathways, or lines.
- Each line is capable of transmitting signals representing binary 1 and binary 0.
- Over time, a sequence of binary digits can be transmitted across a single line.
- Taken together, several lines of a bus can be used to transmit binary digits simultaneously (in parallel).
- For example, an 8-bit unit of data can be transmitted over eight bus lines.

System Bus and Data Bus

- Computer systems contain a number of different buses that provide pathways between components at various levels of the computer system hierarchy.
- A bus that connects major computer components (processor, memory, I/O) is called a **system bus**.
- The most common computer interconnection structures are based on the use of one or more system buses.
- A system bus consists of fifty to hundreds of separate lines.
- The **data lines** provide a path for moving data among system modules. These lines, collectively, are called the **data bus**.
- The data bus may consist of 32, 64, 128, or even more separate lines, the number of lines being referred to as the **width of the data bus**.
- Because **each line can carry only 1 bit at a time**, the number of lines determines how many bits can be transferred at a time.
- For example, if the data bus is 32 bits wide and each instruction is 64 bits long, then the processor must access the memory module twice during each instruction cycle

Address Bus

- The address lines are used to designate the source or destination of the data on the data bus.
- For example, if the processor wishes to read a word (e.g., 8, 16, or 32 bits) of data from memory, it puts the address of the desired word on the address lines.
- The width of the address bus determines the maximum possible memory capacity of the system.
- Furthermore, the address lines are generally also used to address I/O ports. Typically, the higher-order bits are used to select a particular module on the bus, and the lower-order bits select a memory location or I/O port within the module.
- For example, on an 8-bit address bus, address 01111111 and below might reference locations in a memory module (module 0) with 128 words of memory, and address 10000000 and above refer to devices attached to an I/O module (module 1).

Control Bus

- The **control lines** are used to control the access to and the use of the data and address lines.
- Since the data and address lines are shared by all components, there must be a means of controlling their use.
- Control signals transmit both command and timing information among system modules.
- Timing signals indicate the validity of data and address information.
- Command signals specify operations to be performed.

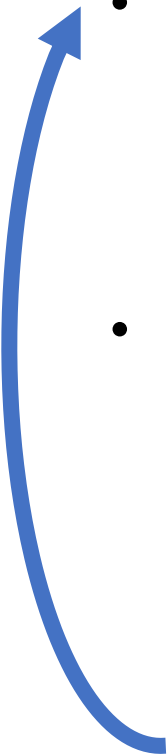
Control Commands

- Command signals specify operations to be performed. Typical control lines include:
 - **Memory write**: causes data on the bus to be written into the addressed location
 - **Memory read**: causes data from the addressed location to be placed on the bus
 - **I/O write**: causes data on the bus to be output to the addressed I/O port
 - **I/O read**: causes data from the addressed I/O port to be placed on the bus
 - **Transfer ACK**: indicates that data have been accepted from or placed on the bus
 - **Bus request**: indicates that a module needs to gain control of the bus
 - **Bus grant**: indicates that a requesting module has been granted control of the bus
 - **Interrupt request**: indicates that an interrupt is pending
 - **Interrupt ACK**: acknowledges that the pending interrupt has been recognized
 - **Clock**: is used to synchronize operations
 - **Reset**: initializes all modules

Operation of the Bus

- The operation of the bus is as follows:
 - If one module wishes to send data to another, it must do two things:
 1. obtain the use of the bus; and
 2. transfer data via the bus.
 - If one module wishes to request data from another module, it must:
 1. obtain the use of the bus; and
 2. transfer a request to the other module over the appropriate control and address lines.
 3. It must then wait for that second module to send the data.

Operation of the Bus

- The operation of the bus is as follows:
 - If one module wishes to send data to another, it must do two things:
 1. obtain the use of the bus; and
 2. transfer data via the bus.
 - If one module wishes to request data from another module, it must:
 1. obtain the use of the bus; and
 2. transfer a request to the other module over the appropriate control and address lines.
 - 3. It must then wait for that second module to send the data.
- 

References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2395

Intro to Computer Architecture

Lecture 5

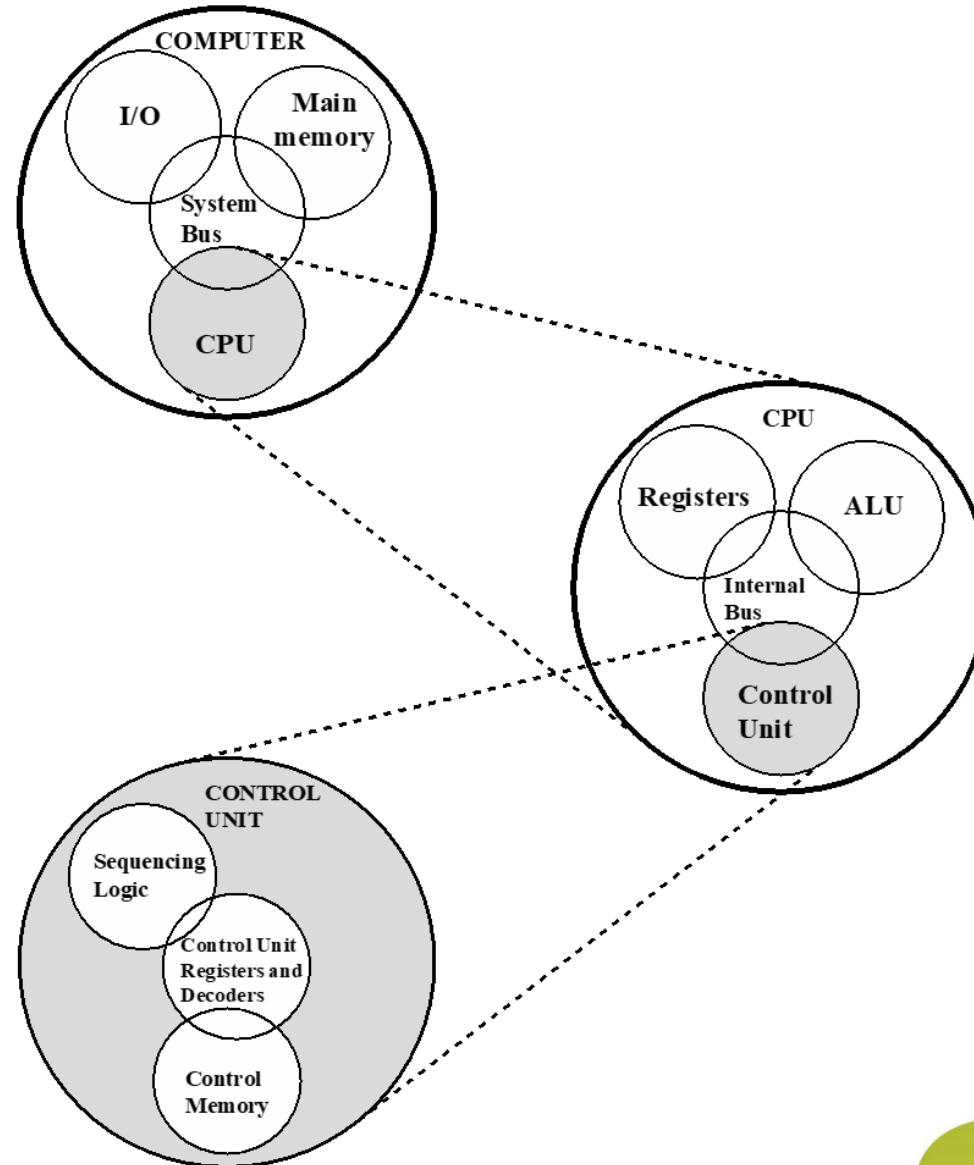


Plan for Today

- Review of the Internal Structure of a Computer
- The Control Unit (CU)
- The Arithmetic Logic Unit (ALU)
- Machine Algorithm for Integer Addition and Subtraction
- Machine Algorithm for Unsigned Integer Multiplication
- Machine Algorithm for Integer Division

The Internal Structure of a Computer

Computer Structure Overview



Computer Structure

- A modern computer has four main structural components:
 1. **Central processing unit (CPU):** Controls the operation of the computer and performs its data processing functions; often simply referred to as processor.
 2. **Main memory:** Stores data.
 3. **I/O:** Moves data between the computer and its external environment.
 4. **System interconnection:** Some mechanism that provides for communication among CPU, main memory, and I/O. A common example of system interconnection is by means of a system bus, consisting of a number of conducting wires to which all the other components attach.

CPU Structure

- A modern computer may have one or more of each of the these components.
- In recent years, there has been increasing use of multiple processors in a single computer.
- Perhaps, the most interesting and in some ways the most complex component is the CPU. Its major structural components are as follows:
 1. **Control unit:** Controls the operation of the CPU and hence the computer.
 2. **Arithmetic and logic unit (ALU):** Performs the computer's data processing functions.
 3. **Registers:** Provides storage internal to the CPU.
 4. **CPU interconnection:** Some mechanism that provides for communication among the control unit, ALU, and registers.

CPU Structure

- There are several approaches to the implementation of the control unit.
- One common approach is a microprogrammed implementation.
- A microprogrammed control unit operates by executing microinstructions that define the functionality of the control unit.
- The instructions we have worked with so far in this course, when examining how a program is executed at the microprocessor level, are examples of microinstruction.

Multicore Computer Structure

- Modern computers generally have multiple processors.
- When these processors all reside on a single chip, the term multicore computer is used, and each processing unit (consisting of a control unit, ALU, registers, and perhaps cache) is called a core.
- To clarify the terminology:
 - **Central processing unit (CPU)** : The unit that fetches and executes instructions. It consists of an **ALU**, a **control unit**, and **registers**. In a system with a single processing unit, it is often simply referred to as a processor .
 - **Core** : An individual processing unit on a processor chip. A core may be equivalent in functionality to a CPU on a single-CPU system.
 - **Processor**: A physical piece of silicon containing one or more cores. The processor is the computer component that interprets and executes instructions. If a processor contains multiple cores, it is referred to as a multicore processor .

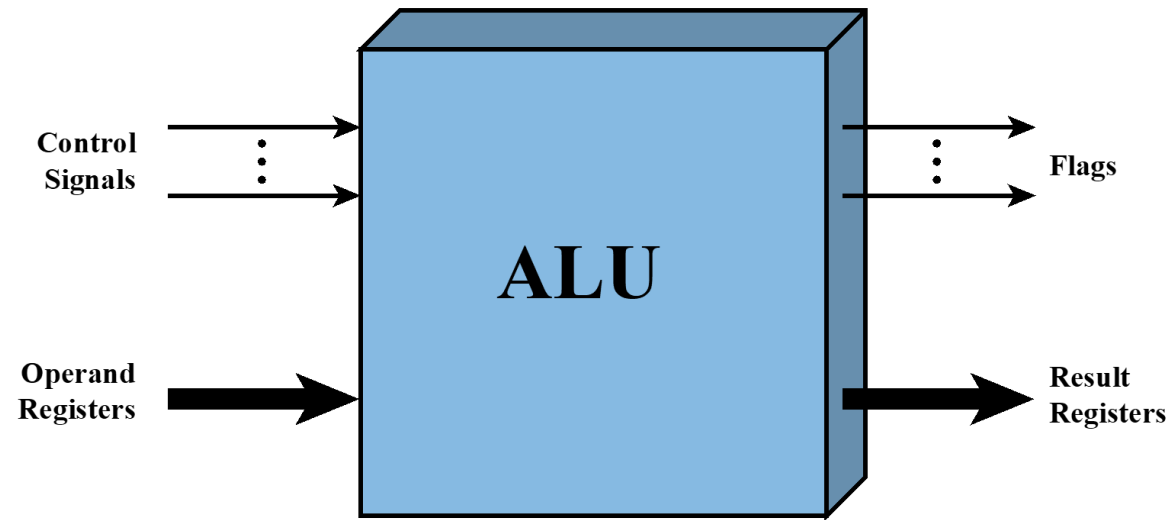
The Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit (ALU)

- The ALU is the part of the computer that actually performs arithmetic and logical operations on data.
- All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out.
- We have reached the essence of a computer when we consider the ALU.
- An ALU and, indeed, all electronic components in the computer are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations.

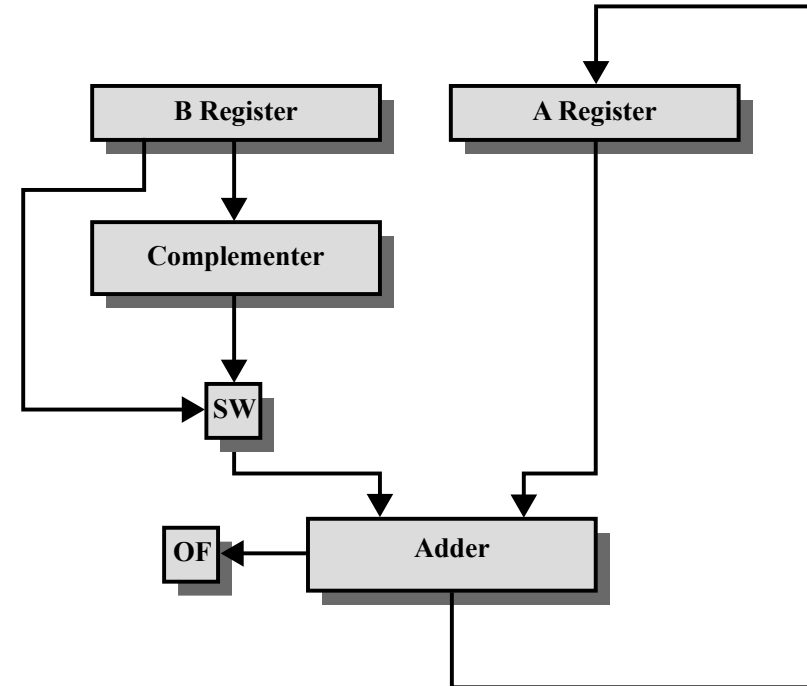
The Arithmetic Logic Unit (ALU)

- Operands for arithmetic and logic operations are presented to the ALU in registers, and the results of an operation are stored in registers.
- These registers are temporary storage locations within the processor that are connected by signal paths to the ALU.
- The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored.
- The flag values are also stored in registers within the processor. The control unit provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.



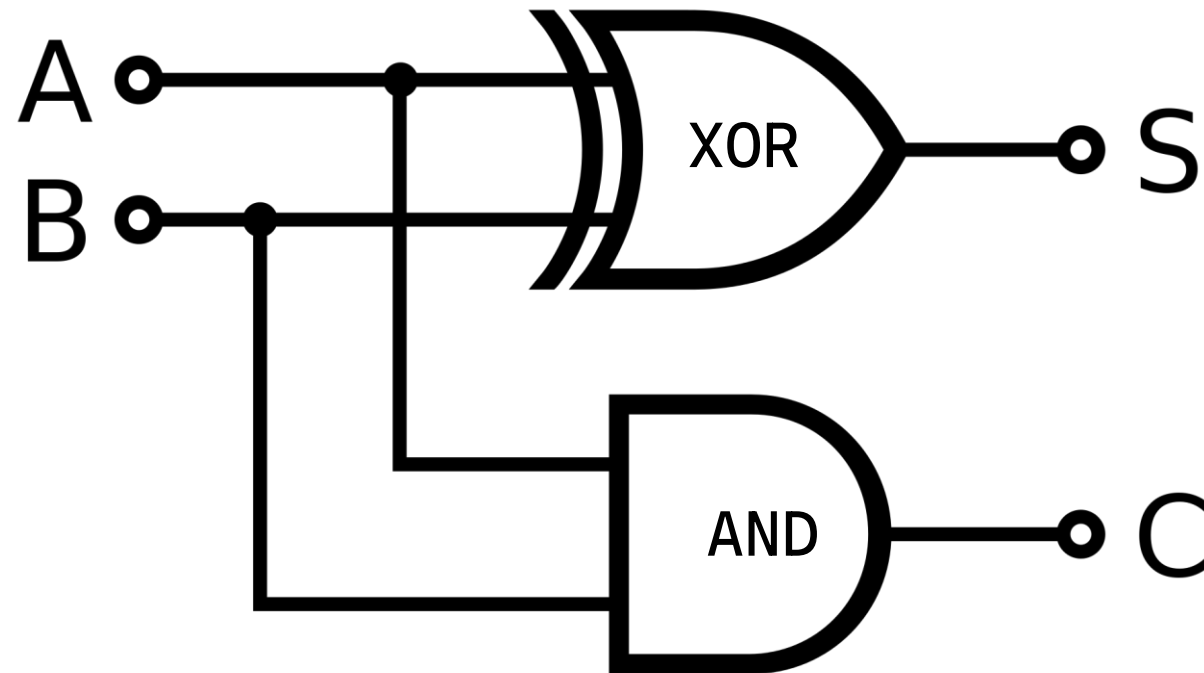
Integer Addition and Subtraction

- The data paths and hardware elements needed to accomplish addition and subtraction are shown here.
- The central element is a **binary adder**, which is given two numbers for addition and produces a sum and an overflow indication.
- The binary adder treats the two numbers as unsigned integers.
- For addition, the two numbers are passed to the adder from two registers, designated here as A and B registers. The result may be stored in one of these registers or in a third register.
- The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; 1 = overflow).
- For subtraction, the B register is passed through a two's complementer so that its two's complement is presented to the adder.
- This diagram only shows the data paths. Control signals are needed to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.

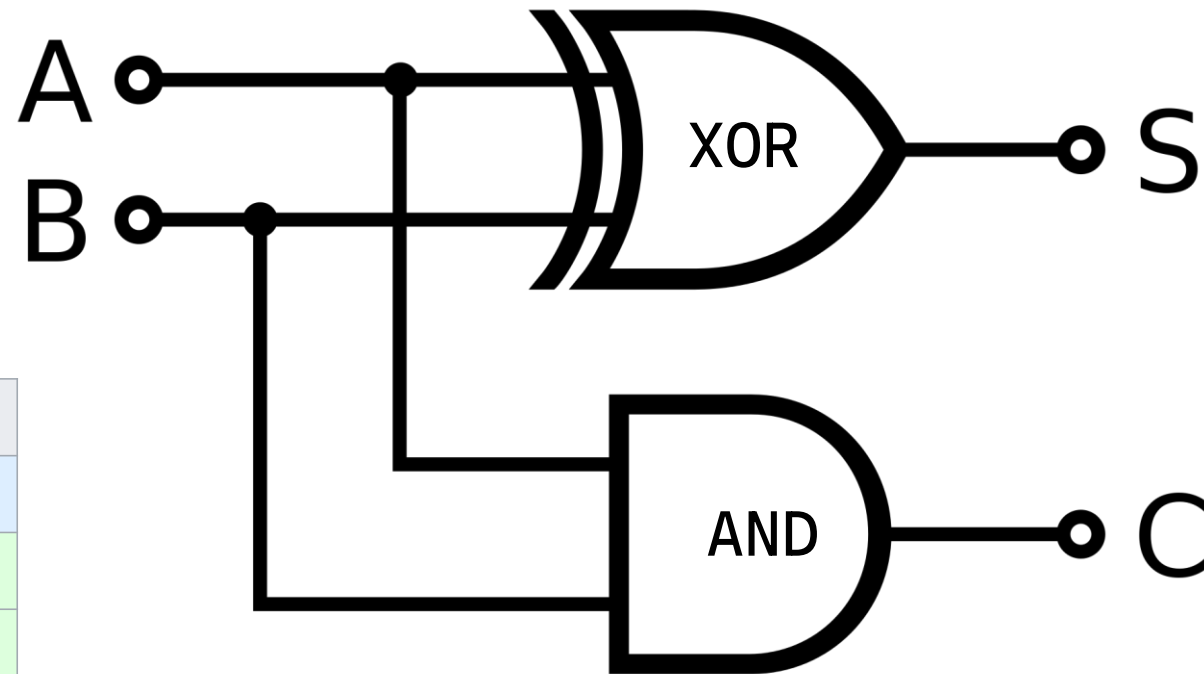


OF = overflow bit
SW = Switch (select addition or subtraction)

Adder Implementation



Adder Implementation



Inputs		Outputs	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Integer Multiplication

- Multiplication involves the generation of **partial products**, one for each digit in the multiplier. These partial products are then summed to produce the final product.
- The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
- The total product is produced by summing the partial products. For this operation, each successive partial product is **shifted one position to the left** relative to the preceding partial product.
- The multiplication of two n-bit binary integers results in a product of up to 2n bits in length (e.g., $11 * 11 = 1001$).

1011	Multiplicand (11)
× 1101	Multiplier (13)
1011	} Partial products
0000	
1011	
1011	} Product (143)
10001111	

Integer Multiplication

- The partial products should be viewed as $2n$ -bit numbers generated from the n -bit multiplicand.
- Thus, as an unsigned integer, the 4-bit multiplicand 1011 is stored in an 8-bit word as 00001011.
- Each partial product (other than that for 2^0) consists of this number shifted to the left, with the unoccupied positions on the right filled with zeros (e.g., a shift to the left of two places yields 00101100).

1011	
×1101	
00001011	$1011 \times 1 \times 2^0$
00000000	$1011 \times 0 \times 2^1$
00101100	$1011 \times 1 \times 2^2$
01011000	$1011 \times 1 \times 2^3$
10001111	

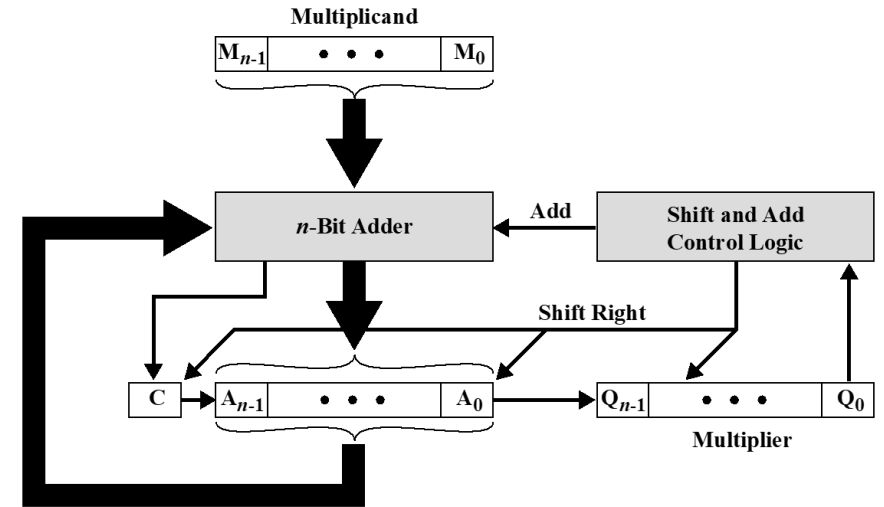
Integer Multiplication

- Compared with theoretical approach, there are several measures we can take to make machine multiplication more efficient.
- First, we can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed.
- Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

1011	Multiplicand (11)
× 1101	Multiplier (13)
1011	} Partial products
0000	
1011	
1011	} Product (143)
10001111	

Integer Multiplication

- This diagram shows a possible implementation for multiplication.
- The multiplier and multiplicand are loaded into two registers (Q and M).
- A third register, the A register, is also needed and is initially set to 0.
- There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition.



(a) Block Diagram

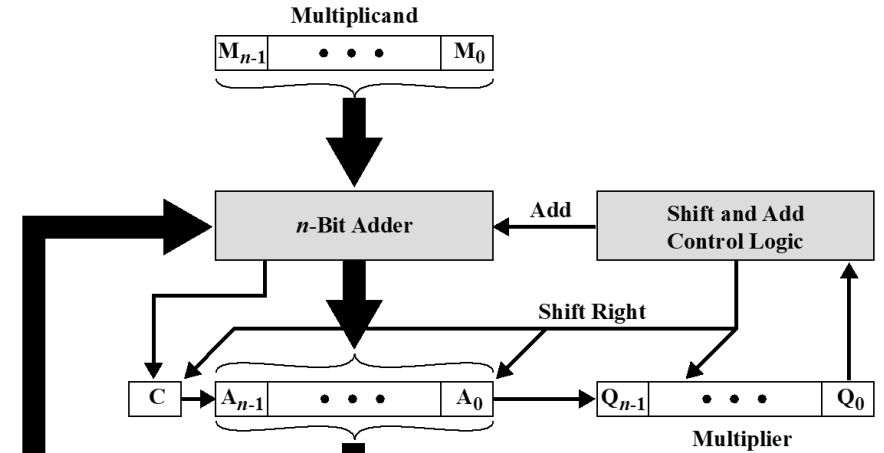
C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	
0	0110	1111	1011	Shift	Third Cycle
1	0001	1111	1011	Add	
0	1000	1111	1011	Shift	Fourth Cycle

Integer Multiplication

1011	Multiplicand (11)
× 1101	Multiplier (13)

1011	} Partial products
0000	
1011	
1011	} Product (143)

10001111	

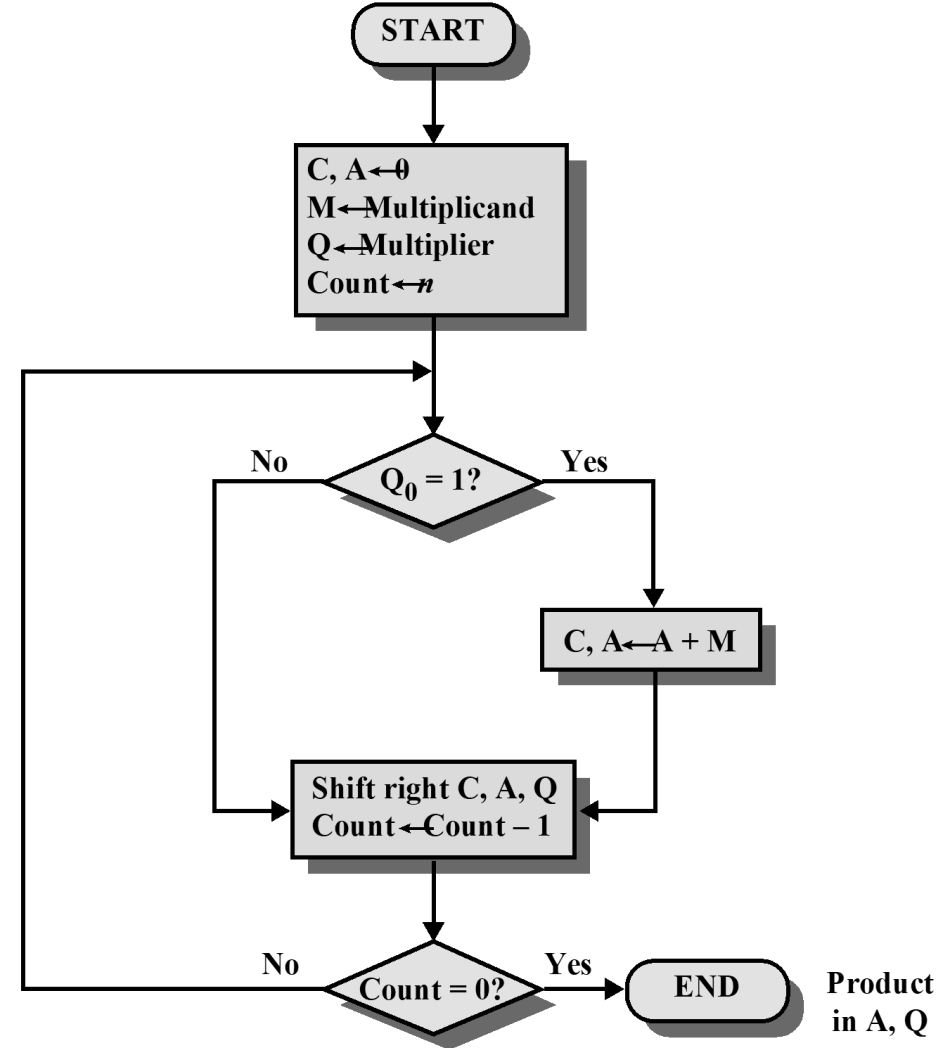


(a) Block Diagram

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	} First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	} Second Cycle
0	1101	1111	1011	Add	
0	0110	1111	1011	Shift	} Third Cycle
1	0001	1111	1011	Add	
0	1000	1111	1011	Shift	} Fourth Cycle

Integer Multiplication

1. Control logic reads the bits of the multiplier one at a time.
2. If Q_0 is 1, then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow.
3. Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1} , A_0 goes into Q_{n-1} and Q_0 is lost.
4. If Q_0 is 0, then no addition is performed, only the right-shift.
5. This process is repeated for each bit of the original multiplier.
6. The resulting $2n$ -bit product is contained in the A and Q registers.

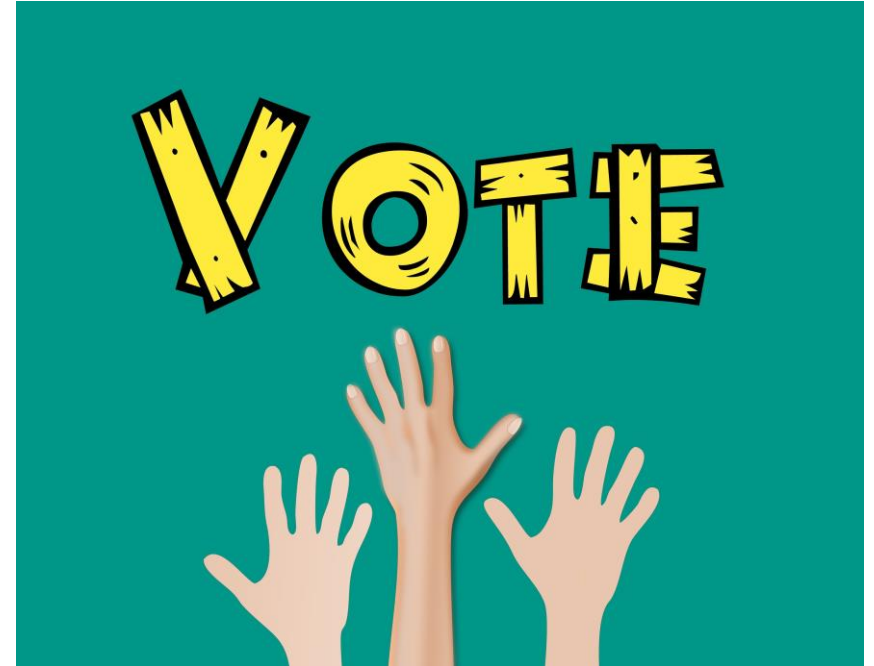


Poll 1 (Extra Credit)

The same algorithm can be used for signed integers.

- a) True
- b) False
- c) Not sure!

Please use the “Poll” window to participate for extra credit!



Integer Multiplication (Signed)

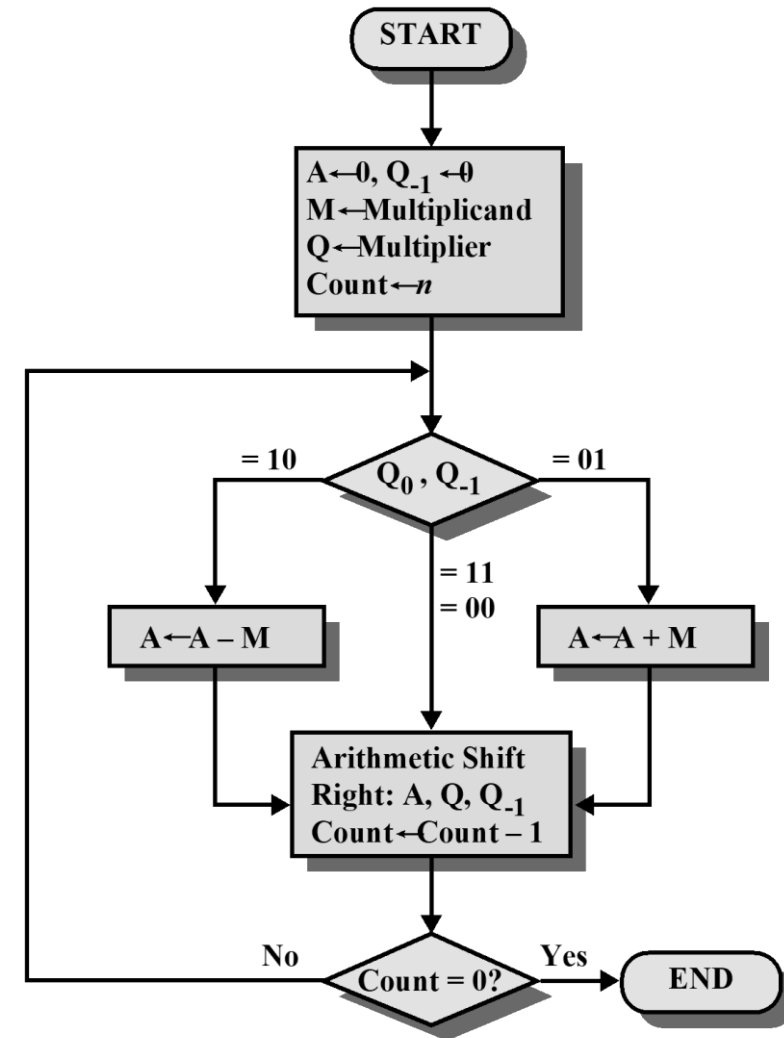
- Unlike addition and subtraction, the straightforward multiplication will not work if the multiplicand is negative.
- Let us examine the multiplication of 1011 by 1101.
- If these are treated as unsigned integers, the multiplication is $11 * 13 = 143$.
- However, if they are interpreted as the twos complement values, then each partial product must be a negative twos complement number of $2n$ (8) bits.
- This is accomplished by padding out each partial product to the left with binary 1s.

Booth's Algorithm for Multiplication

- There are a number of ways to handle multiplication of signed integers.
- One would be to convert both multiplier and multiplicand to positive numbers, perform the multiplication, and then take the two's complement of the result if and only if the sign of the two original numbers differed.
- However, machine implementers have over the years preferred to use algorithms that do not require this final transformation step.
- One approach is proposed by Andrew Donald Booth in 1950s.
- **Booth's algorithm** also speeds up the multiplication process in comparison to the approach we examined earlier.

Booth's Algorithm for Multiplication

- In Booth's algorithm, the multiplier and multiplicand are placed in the Q and M registers, respectively. There is also a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1} .
- The results of the multiplication will appear in the A and Q registers. A and Q_{-1} are initialized to 0.
- The bits of the multiplier are scanned one at a time. As each bit is examined, the bit to its right is also examined.
- If the two bits are the same (1–1 or 0–0), then all of the bits of the A, Q, and Q_{-1} registers are **shifted to the right** by one. If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are 0–1 or 1–0.
- Following the addition or subtraction, the right shift occurs. In either case, the right-shift is the arithmetic shift, because it preserves the sign bit.



Booth's Algorithm for Multiplication – Example

A	Q	Q ₋₁	M	Initial Values	
0000	0011	0	0111		
1001	0011	0	0111	A ← A - M Shift	} First Cycle
1100	1001	1	0111		
1110	0100	1	0111	Shift	} Second Cycle
0101	0100	1	0111	A ← A + M	
0010	1010	0	0111	Shift	} Third Cycle
0001	0101	0	0111	Shift	
					} Fourth Cycle

Integer Division

- Division is more complex than multiplication. The operation involves repetitive shifting and addition or subtraction.
- The bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor. Until this occurs, 0s are placed in the quotient from left to right. When this occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a partial remainder.
- From this point on, the division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. The divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

Integer Division Example

A	Q	
0000	0111	Initial value
0000 <u>1101</u> 1101 0000	1110	Shift Use twos complement of 0011 for subtraction Subtract Restore, set $Q_0 = 0$
0001 <u>1101</u> 1110 0001	1100	Shift Subtract Restore, set $Q_0 = 0$
0011 <u>1101</u> 0000	1000 1001	Shift Subtract, set $Q_0 = 1$
0001 <u>1101</u> 1110 0001	0010 0010	Shift Subtract Restore, set $Q_0 = 0$

References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2395

Intro to Computer Architecture

Lecture 6

Plan for Today

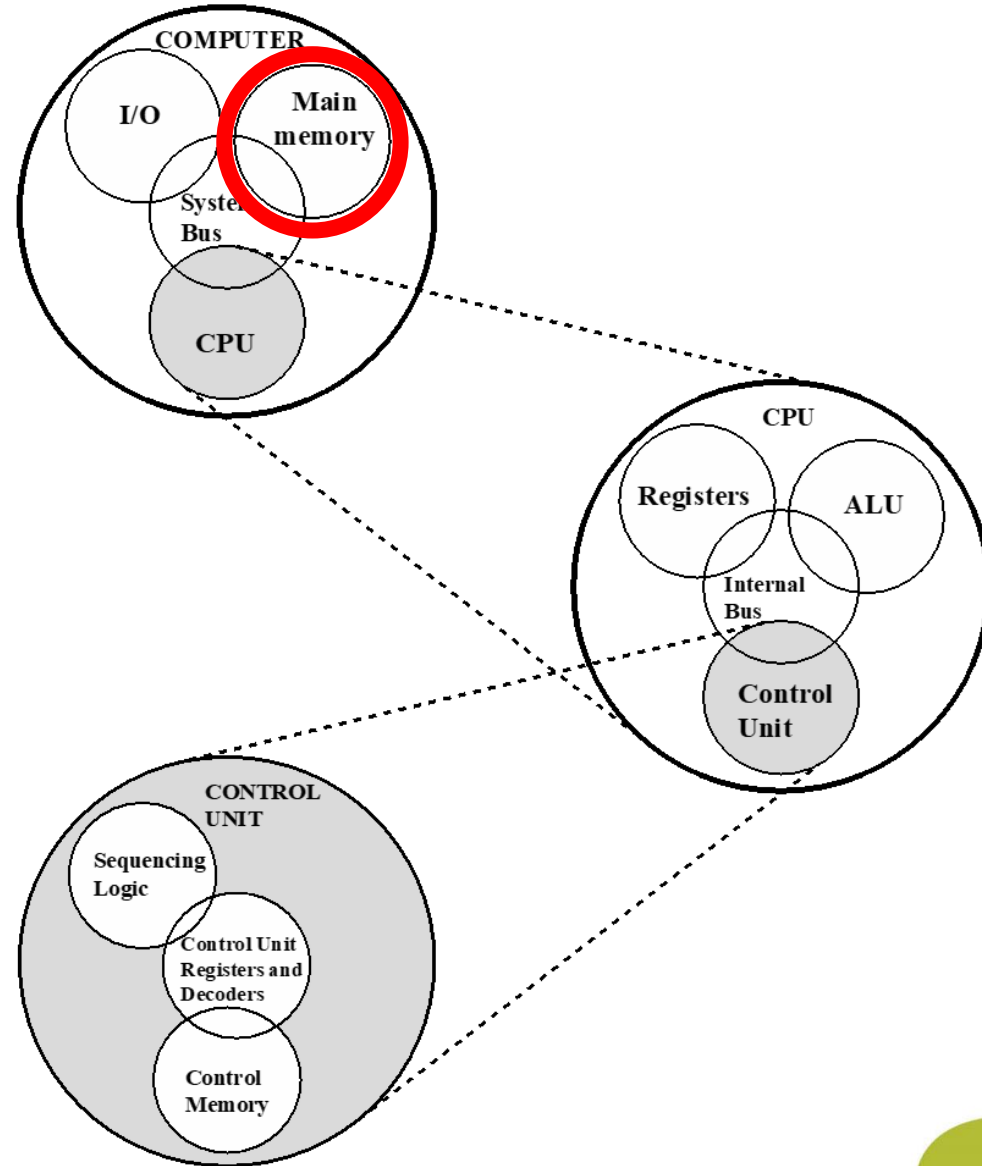
- The Stack
- The Heap and Dynamic Memory in C



DOUGLAS COLLEGE

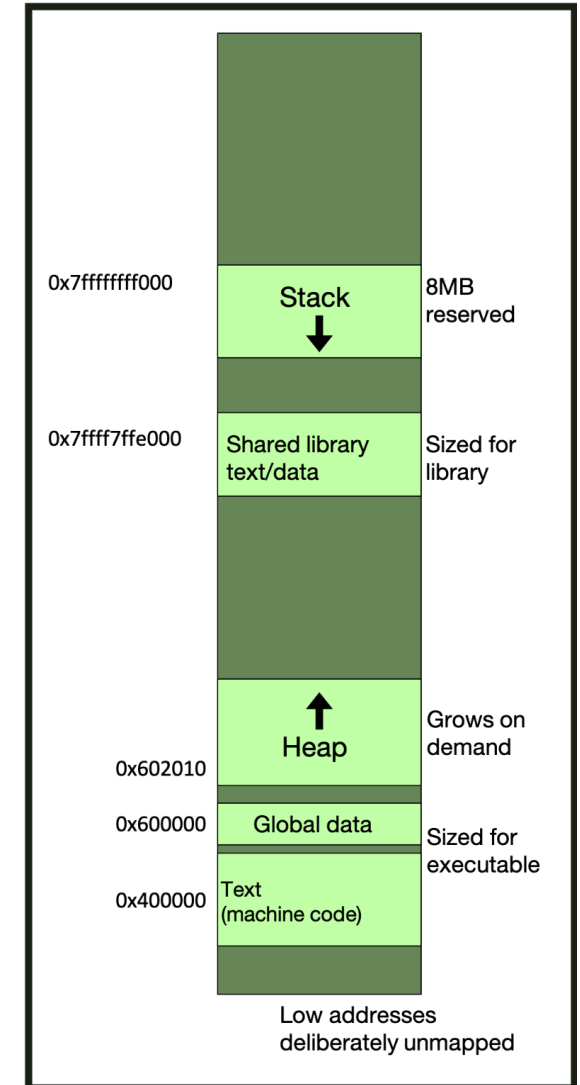
Memory Layout and the Stack

Computer Structure Overview



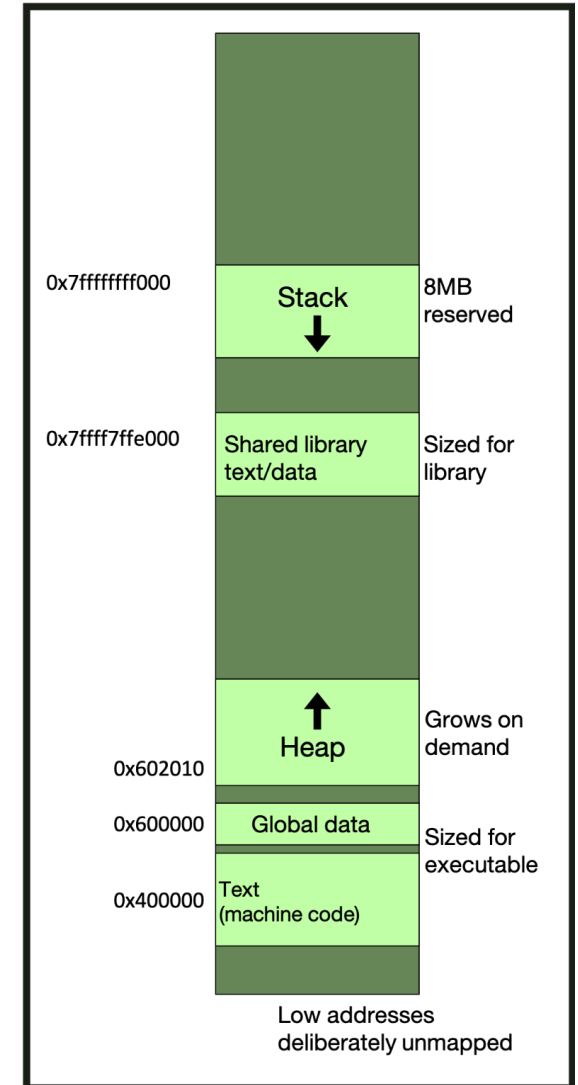
The Stack

- We would like to take a deeper look into different areas of memory used by our programs.
- The stack is the place where all local variables and parameters live for each function.
- A function's stack "frame" goes away when the function returns.
- The stack **grows downwards** when a new function is called and **shrinks upwards** when the function is finished.



The Stack

- The stack operates on the basis of LIFO (Last-In First-Out).
- A new function call **pushes** on a new **stack frame**.
- A completed function call **pops** off the most recent **stack frame**.
- C does not clear out memory when a function's frame is removed. Instead, it simply marks that memory as usable for the next function call.
- A **stack overflow** occurs when we use up all stack memory (e.g., a recursive call with too many function calls).

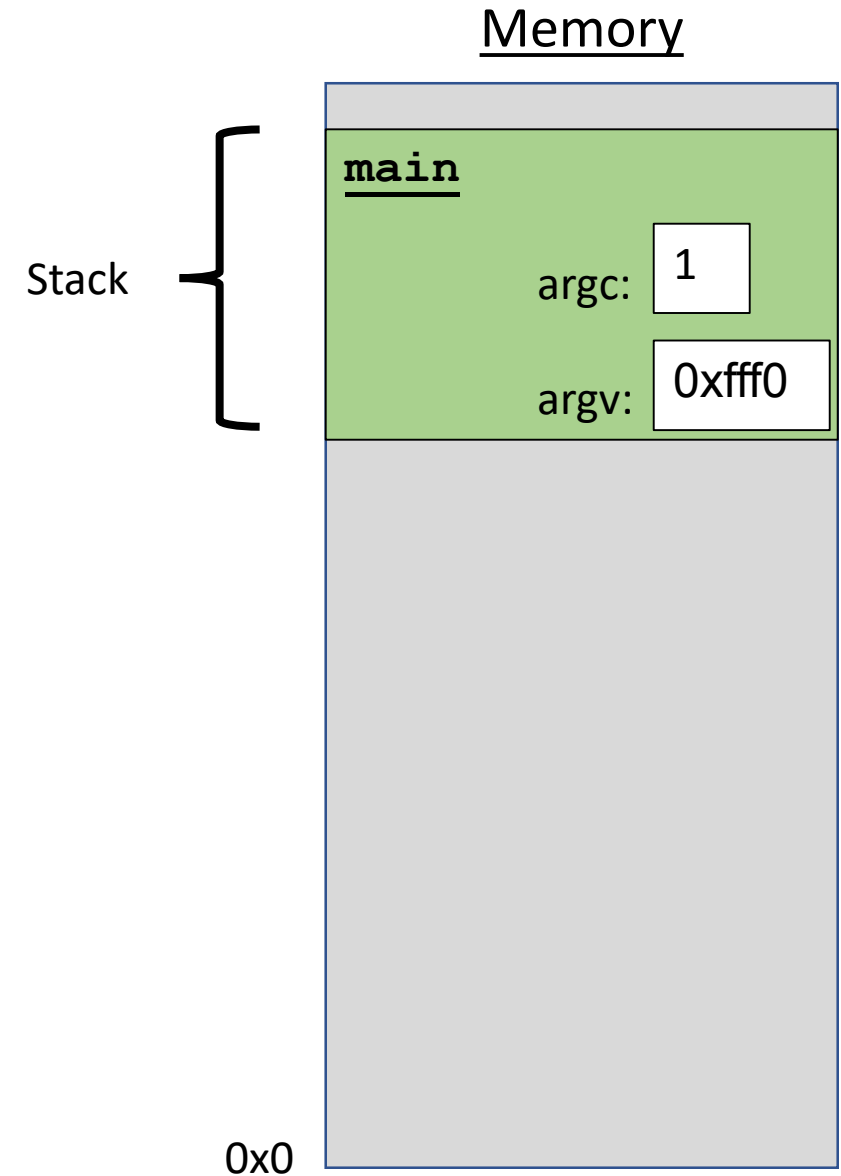


The Stack

```
void fun2() {  
    int d = 0;  
}
```

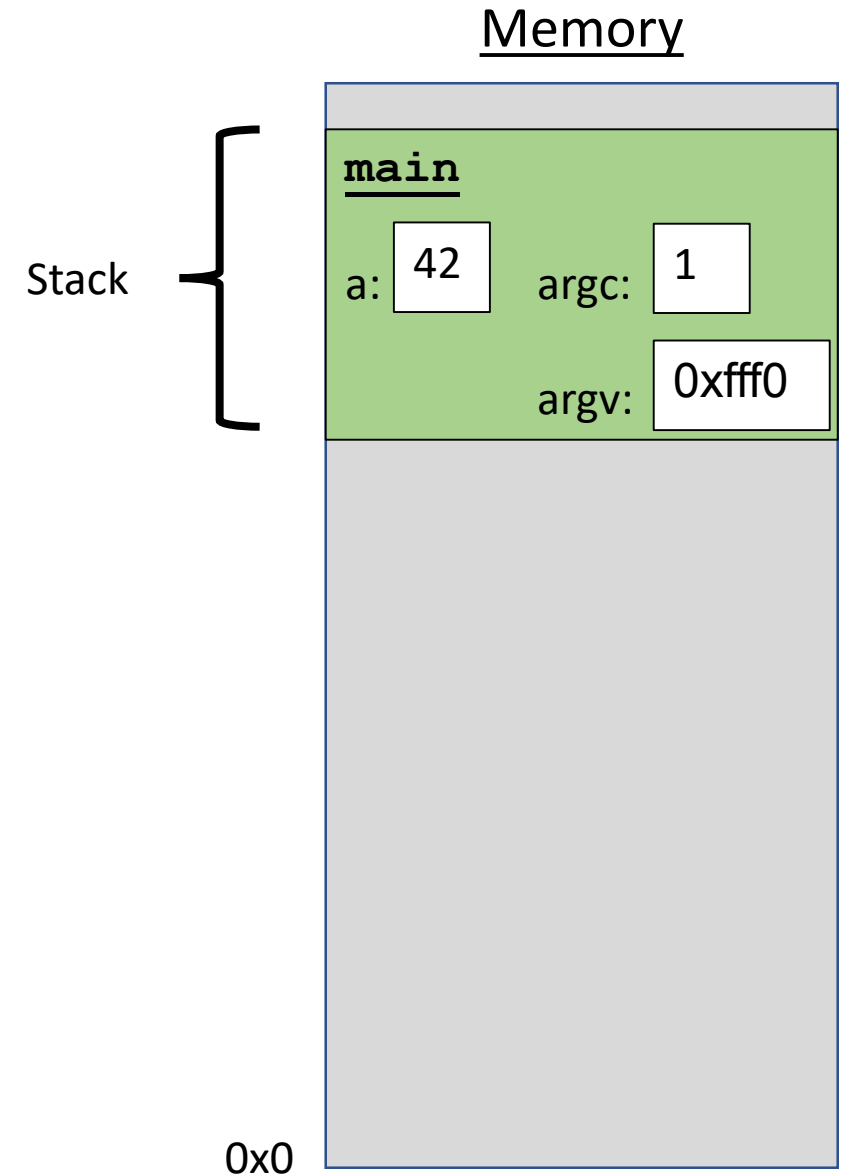
```
void fun1() {  
    int c = 99;  
    fun2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```



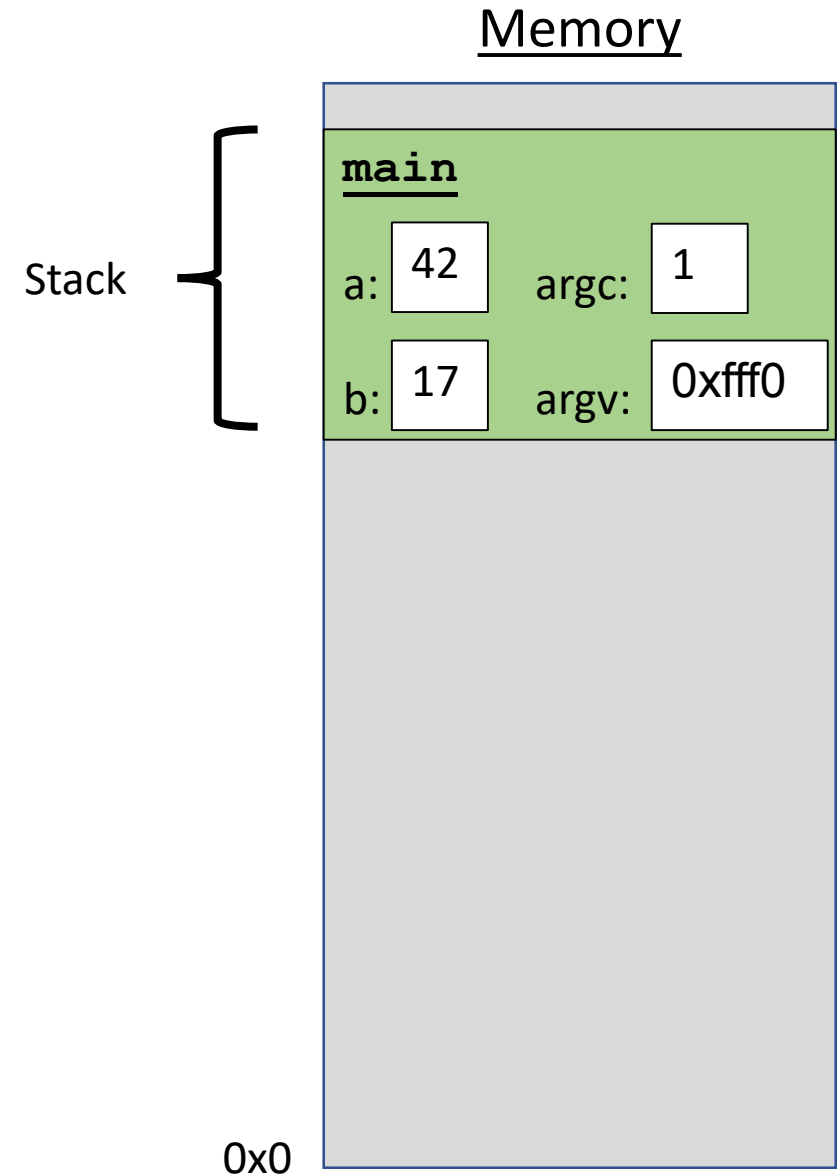
The Stack

```
void fun2() {  
    int d = 0;  
}  
  
void fun1() {  
    int c = 99;  
    fun2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```



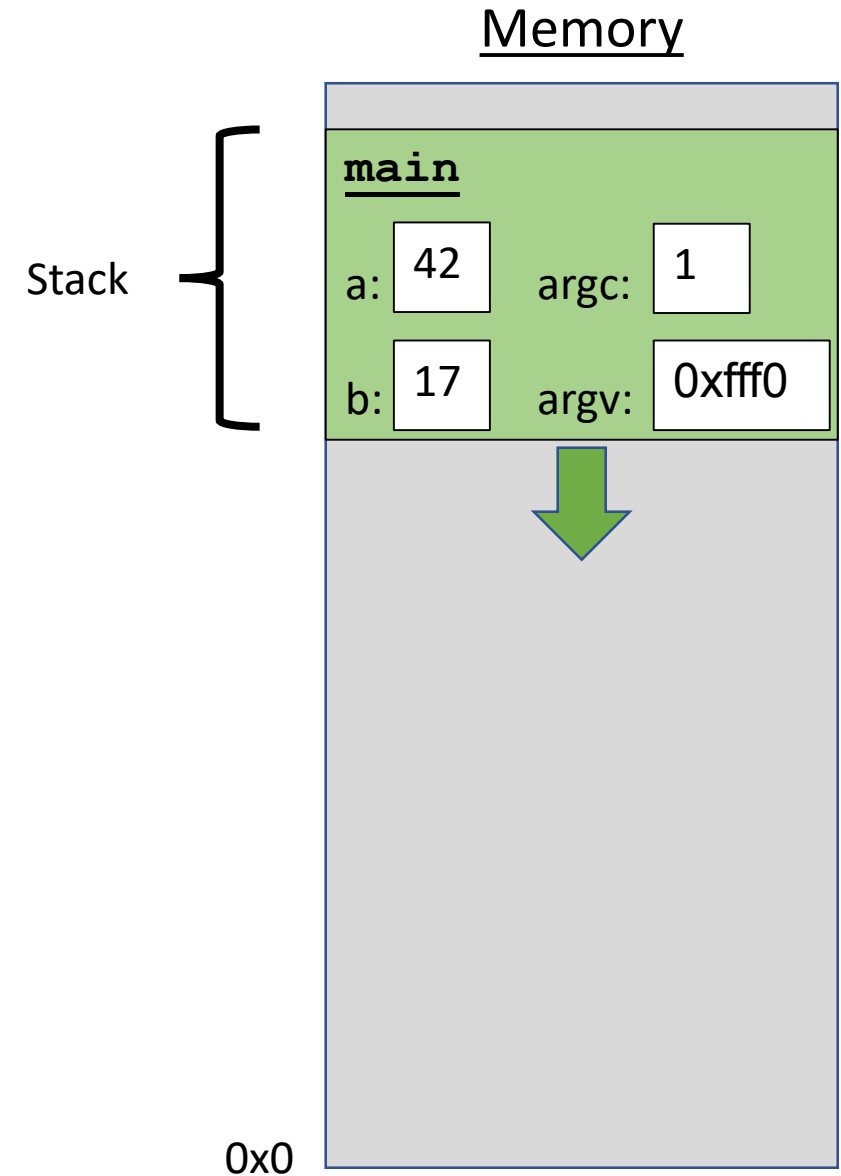
The Stack

```
void fun2() {  
    int d = 0;  
}  
  
void fun1() {  
    int c = 99;  
    fun2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```



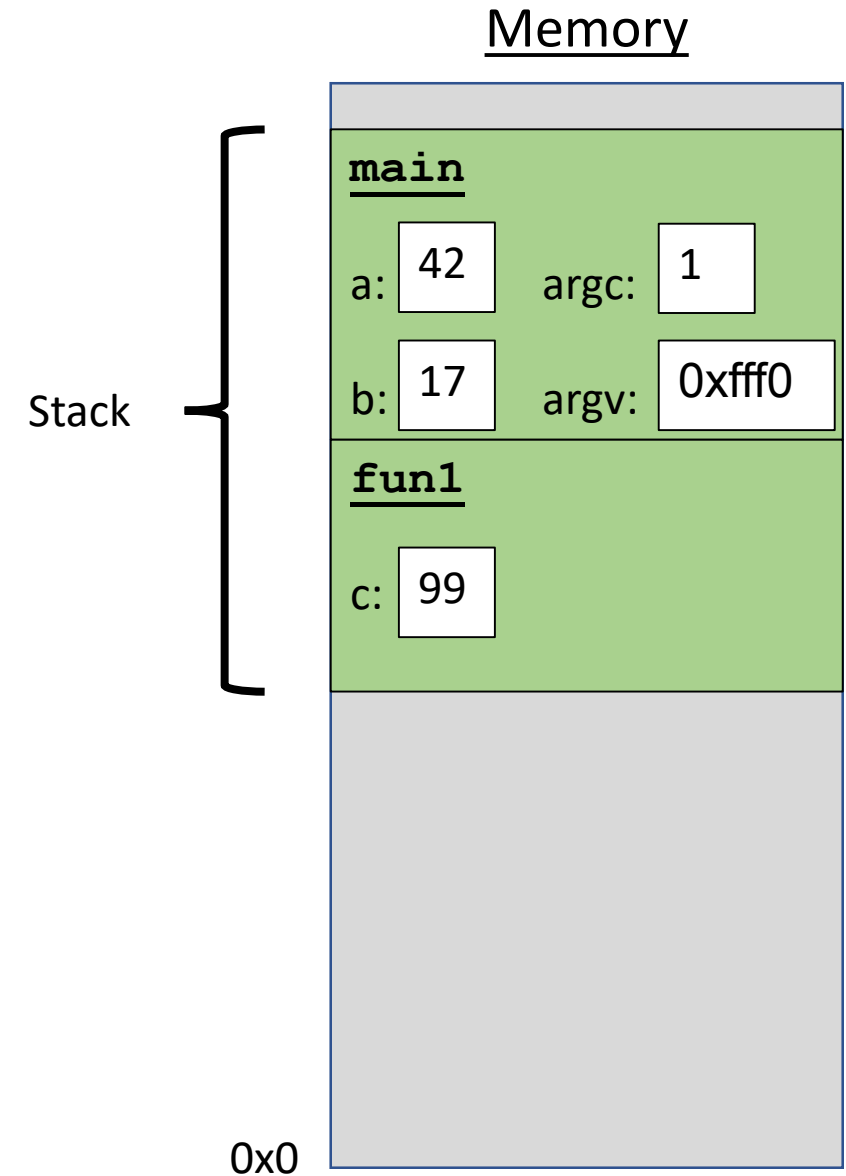
The Stack

```
void fun2() {  
    int d = 0;  
}  
  
void fun1() {  
    int c = 99;  
    fun2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```



The Stack

```
void fun2() {  
    int d = 0;  
}  
  
void fun1() {  
    int c = 99;  
    fun2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```

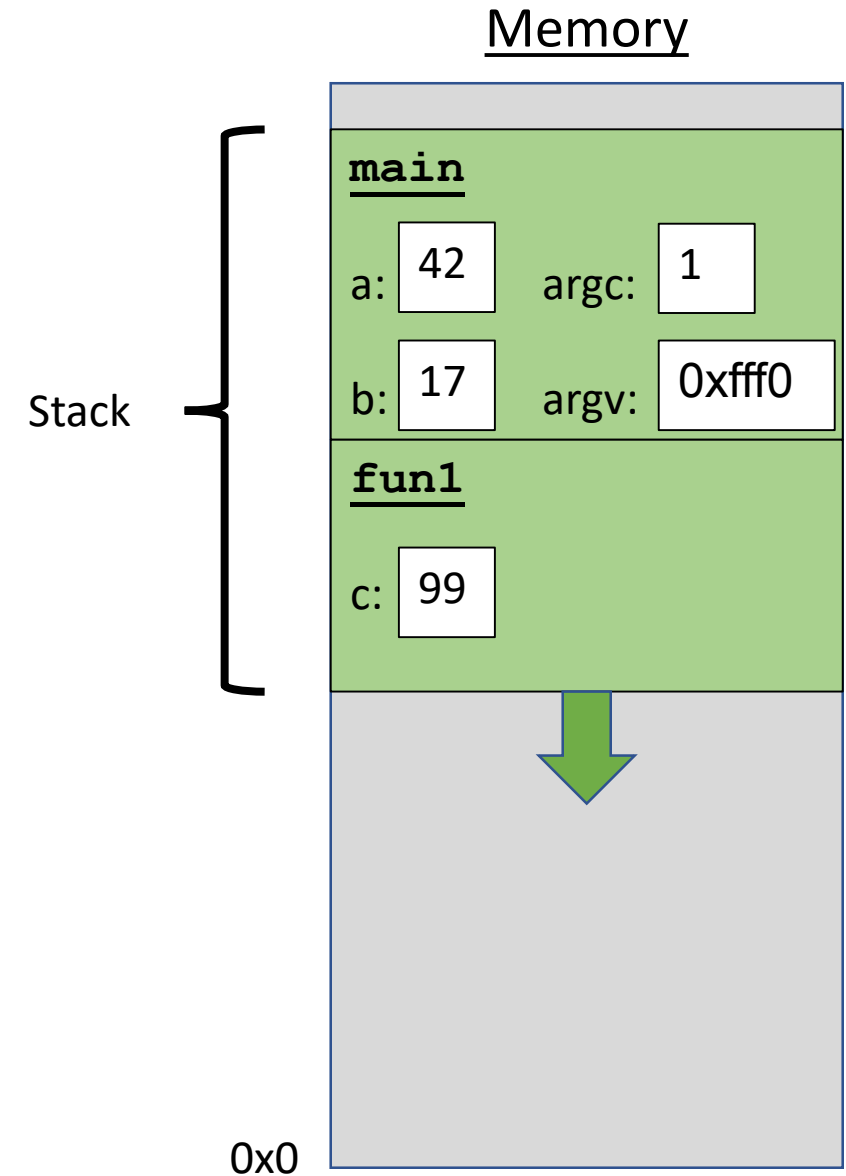


The Stack

```
void fun2() {  
    int d = 0;  
}
```

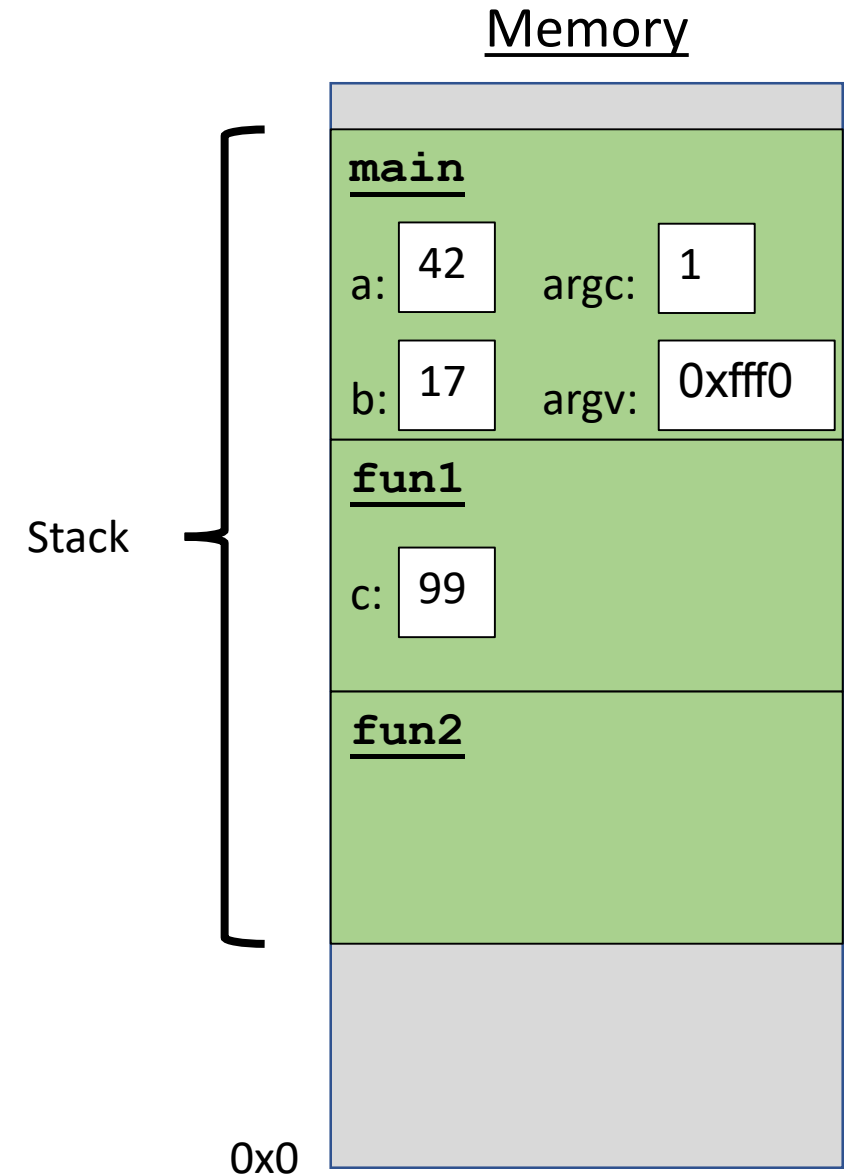
```
void fun1() {  
    int c = 99;  
    fun2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```



The Stack

```
void fun2() {  
    int d = 0;  
}  
  
void fun1() {  
    int c = 99;  
    fun2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```

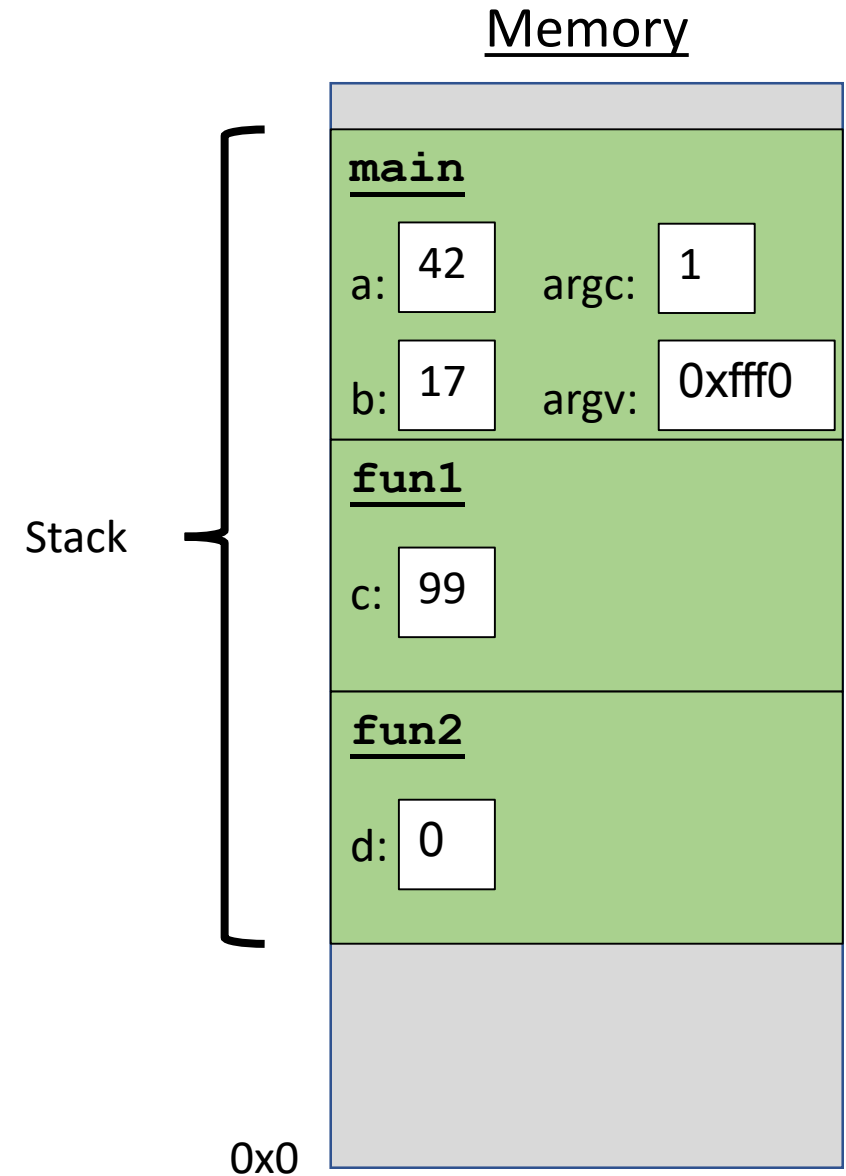


The Stack

```
void fun2() {  
    int d = 0;  
}
```

```
void fun1() {  
    int c = 99;  
    fun2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```

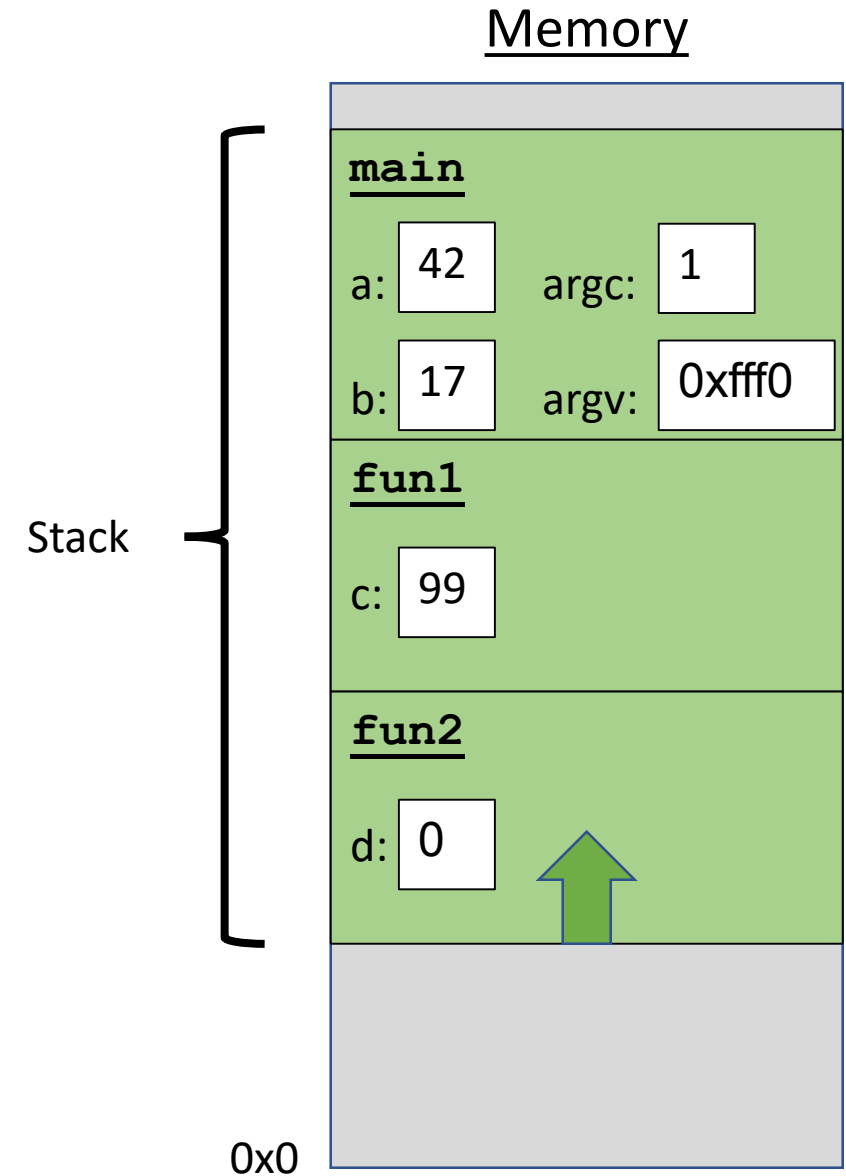


The Stack

```
void fun2() {  
    int d = 0;  
}
```

```
void fun1() {  
    int c = 99;  
    fun2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```

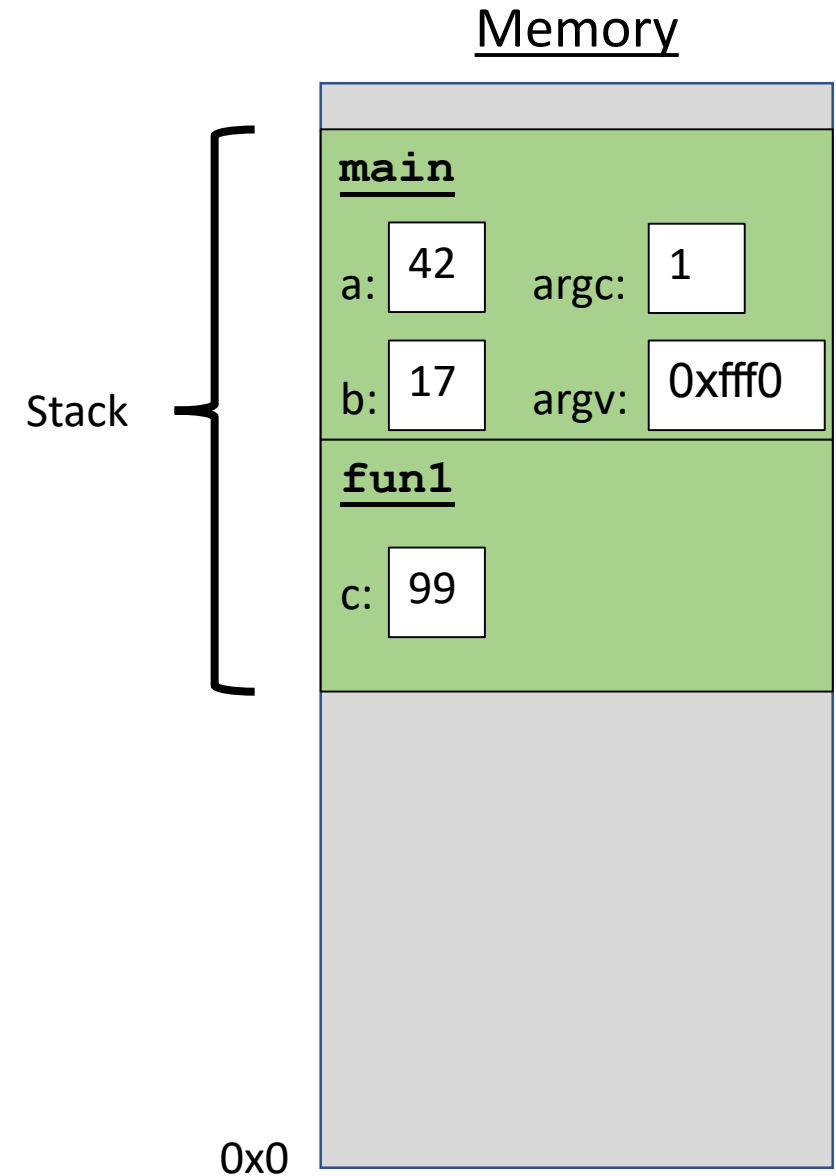


The Stack

```
void fun2() {  
    int d = 0;  
}
```

```
void fun1() {  
    int c = 99;  
    fun2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```

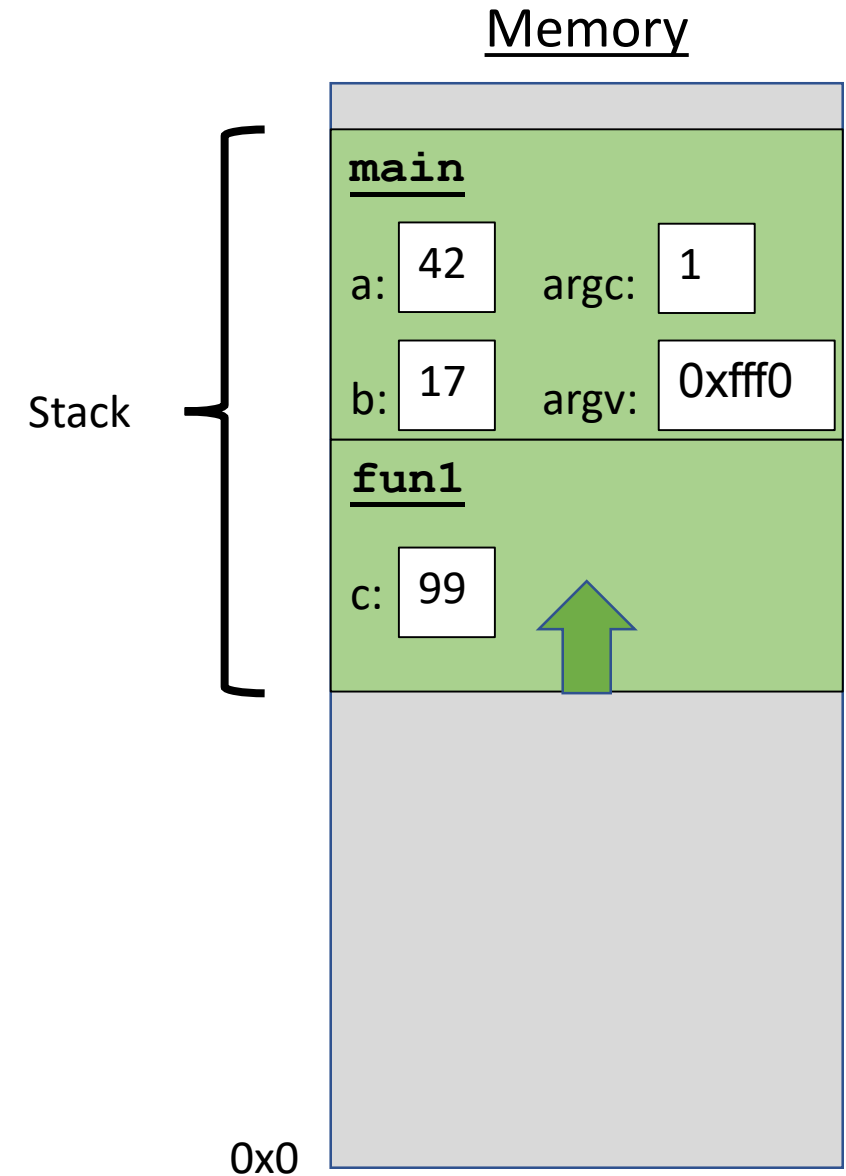


The Stack

```
void fun2() {  
    int d = 0;  
}
```

```
void fun1() {  
    int c = 99;  
    fun2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```

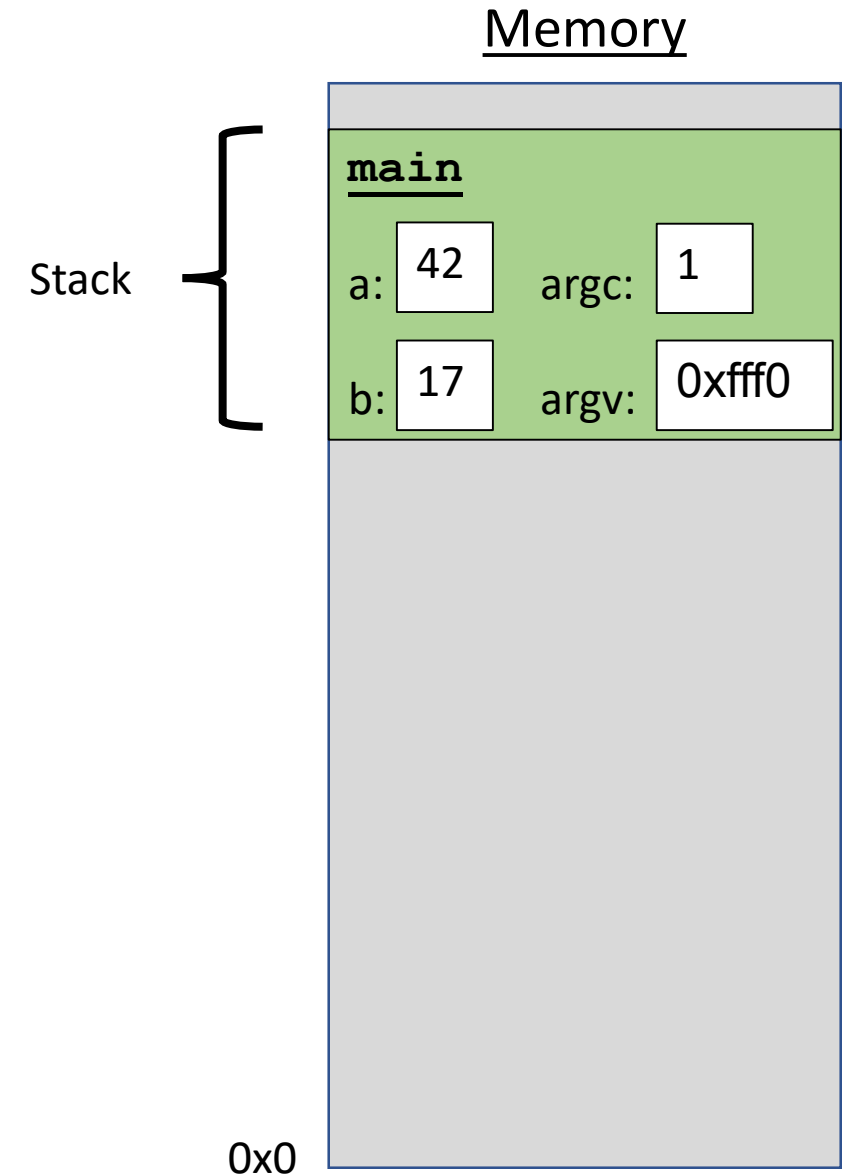


The Stack

```
void fun2() {  
    int d = 0;  
}
```

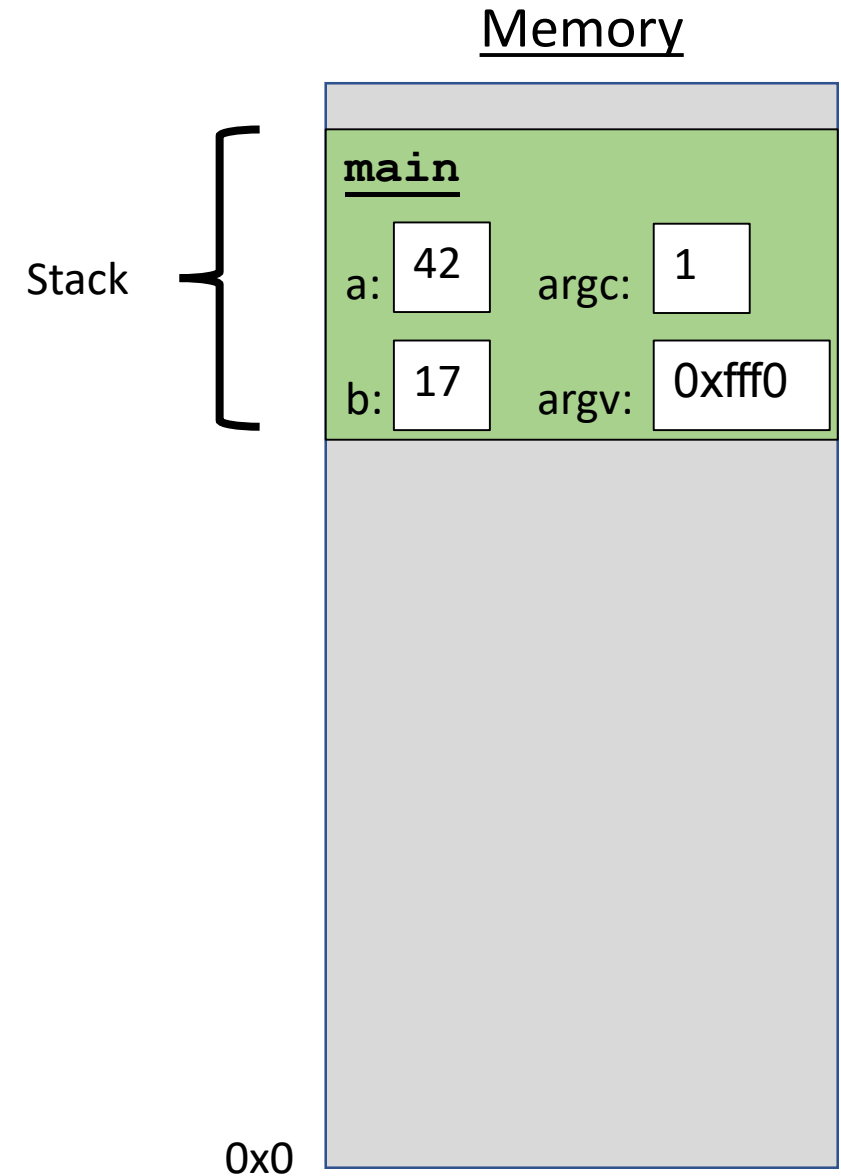
```
void fun1() {  
    int c = 99;  
    fun2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```



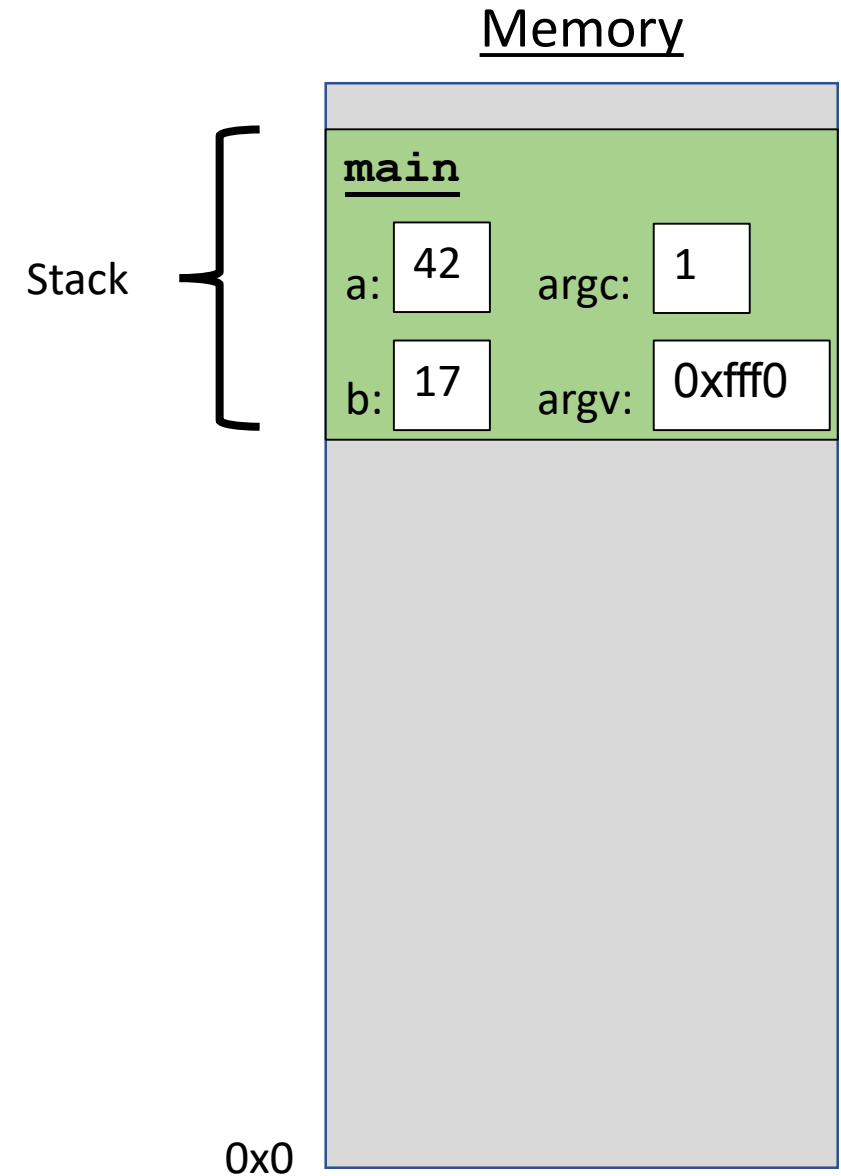
The Stack

```
void fun2() {  
    int d = 0;  
}  
  
void fun1() {  
    int c = 99;  
    fun2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```



The Stack

```
void fun2() {  
    int d = 0;  
}  
  
void fun1() {  
    int c = 99;  
    fun2();  
}  
  
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```

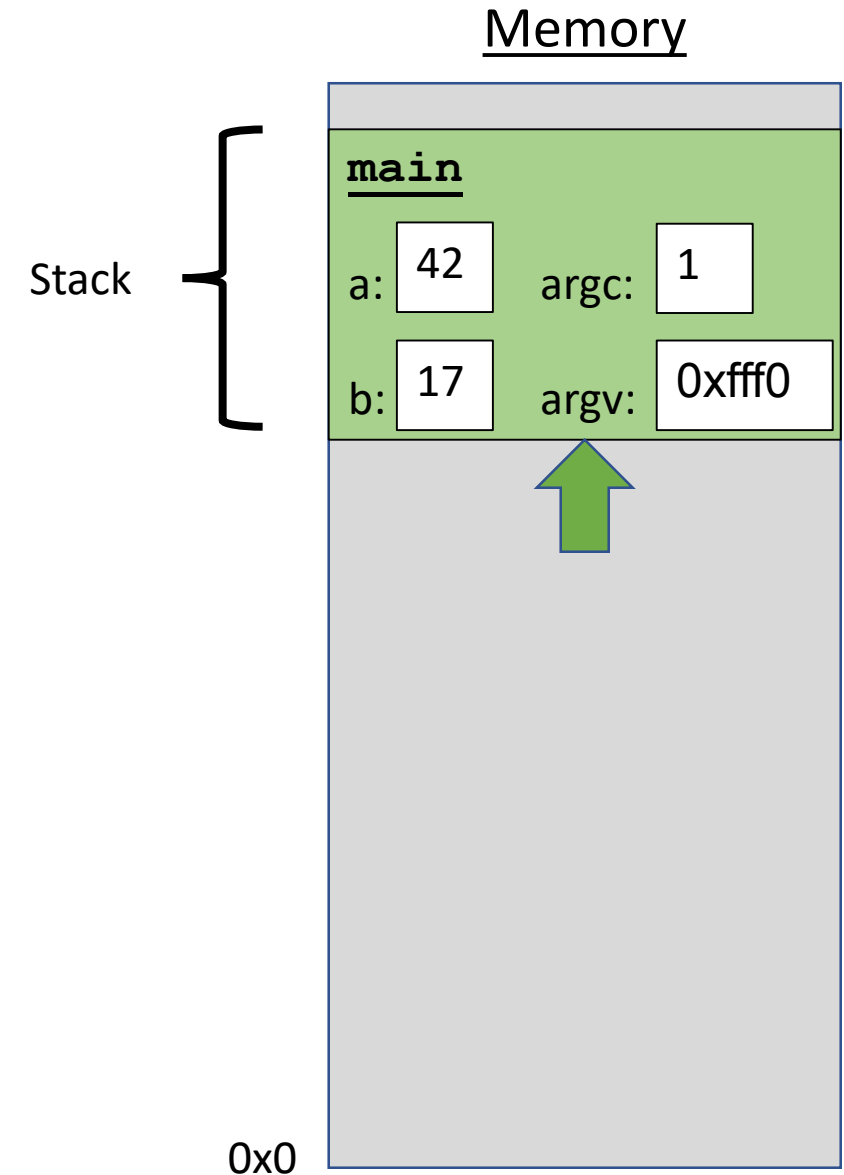


The Stack

```
void fun2() {  
    int d = 0;  
}
```

```
void fun1() {  
    int c = 99;  
    fun2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;  
}
```



The Stack

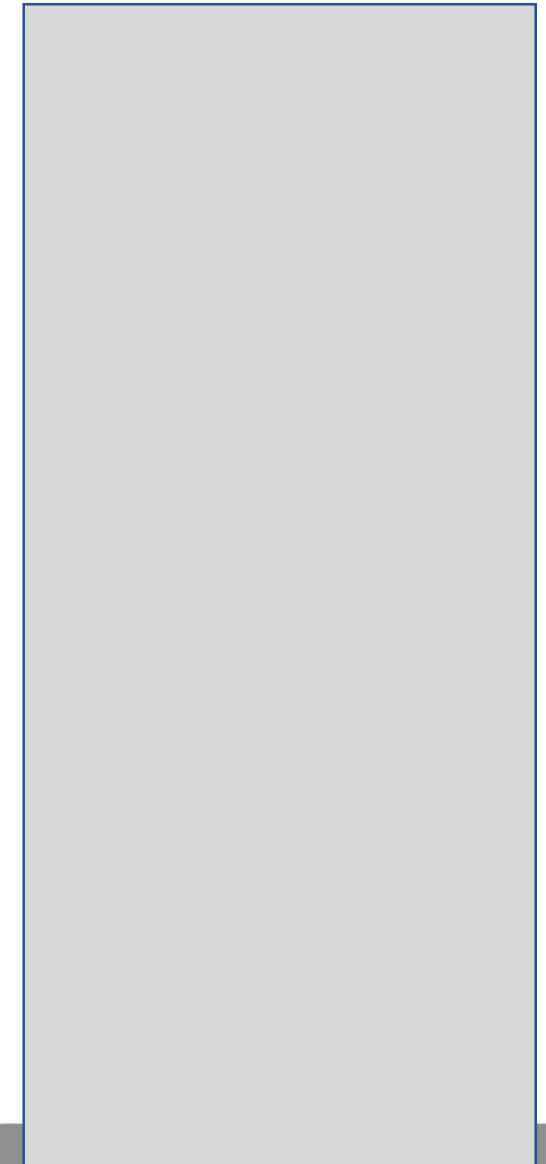
```
void fun2() {  
    int d = 0;  
}
```

```
void fun1() {  
    int c = 99;  
    fun2();  
}
```

```
int main(int argc, char *argv[]) {  
    int a = 42;  
    int b = 17;  
    fun1();  
    printf("Done.");  
    return 0;
```

```
}
```

Memory



0x0

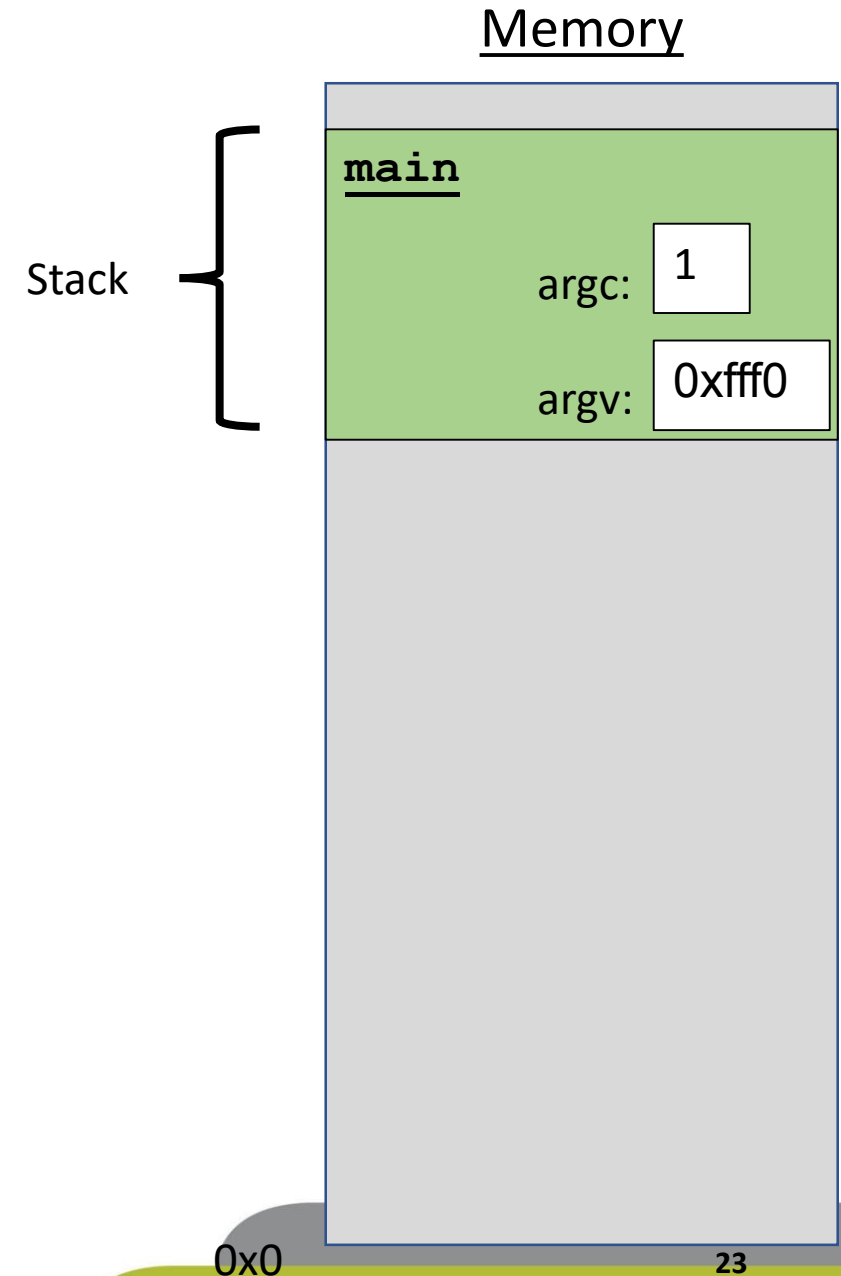
22

The Stack

- Each function call has its own stack frame to store its local variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

```
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```

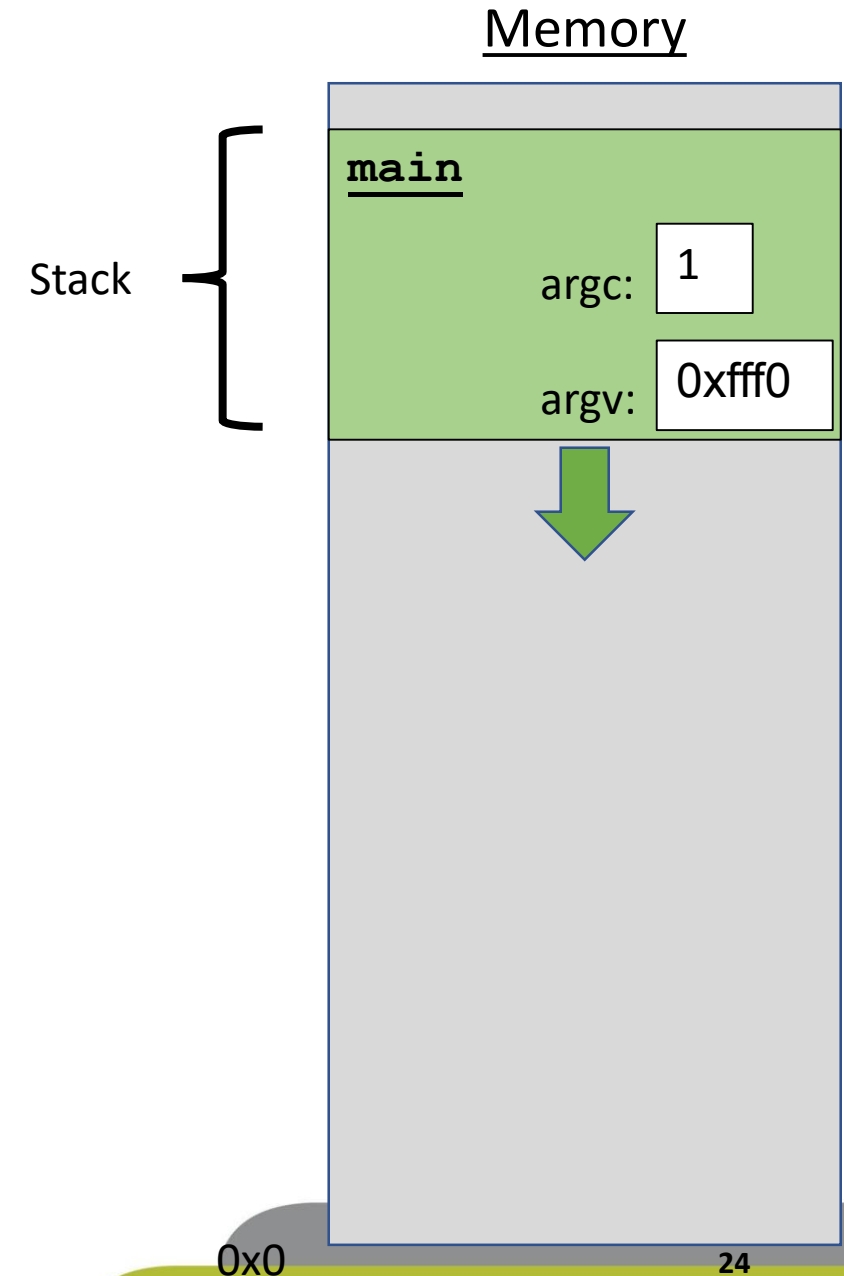


The Stack

- Each function call has its own stack frame to store its local variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

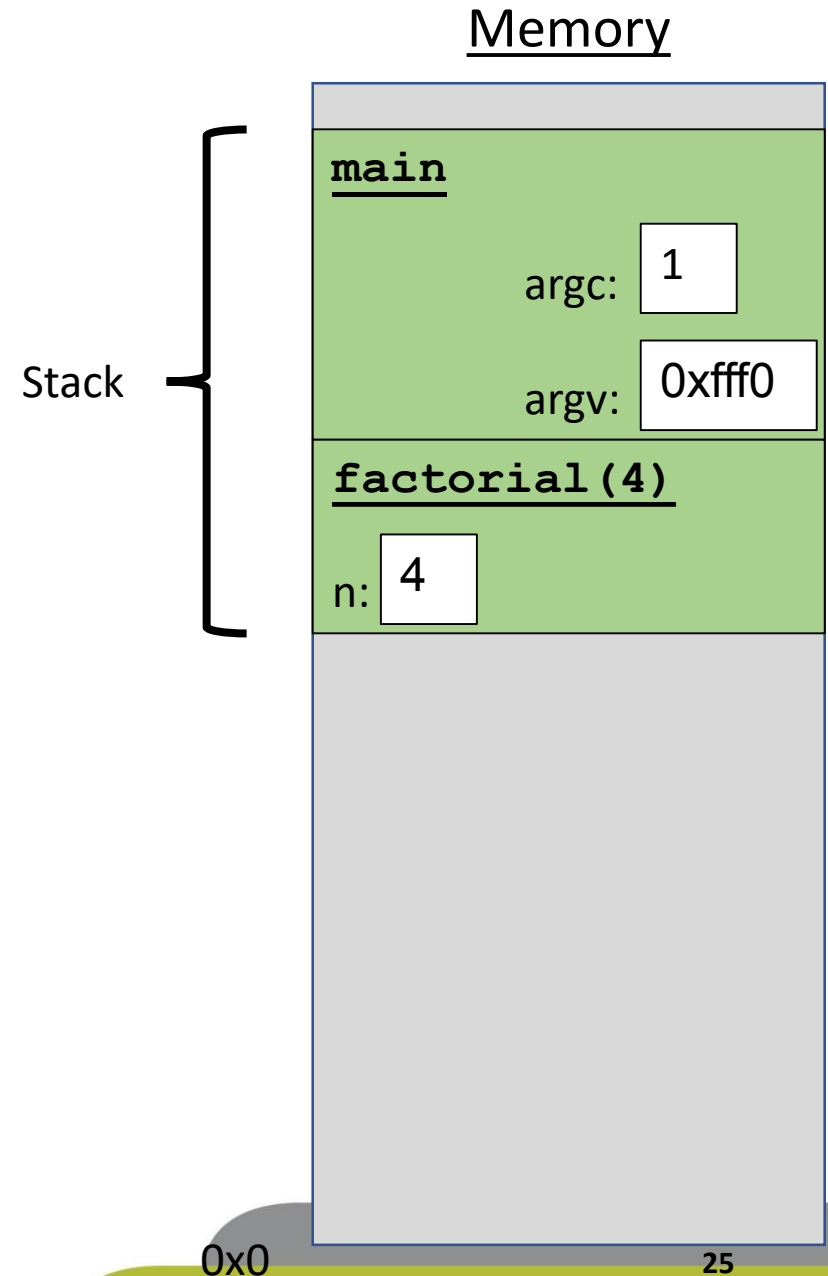
```
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



The Stack

- Each function call has its own stack frame to store its local variables.

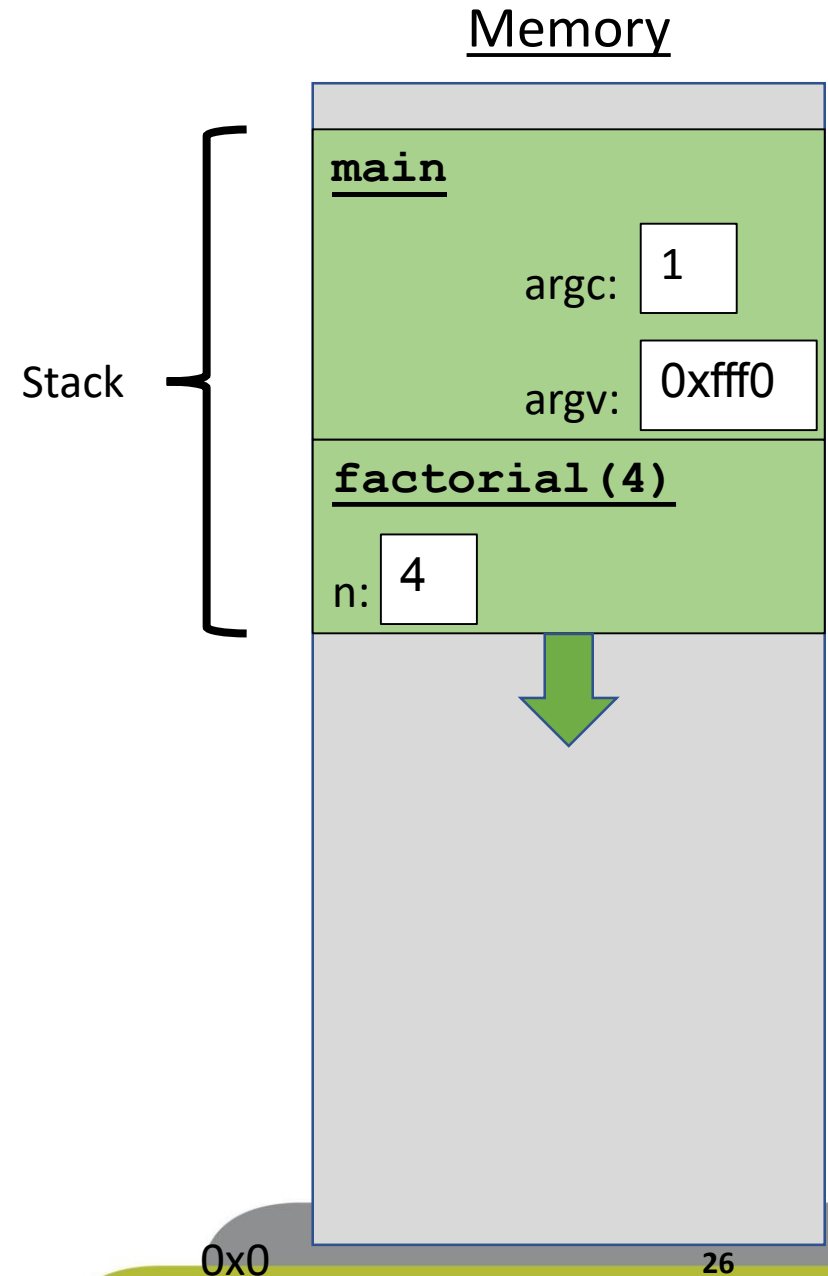
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



The Stack

- Each function call has its own stack frame to store its local variables.

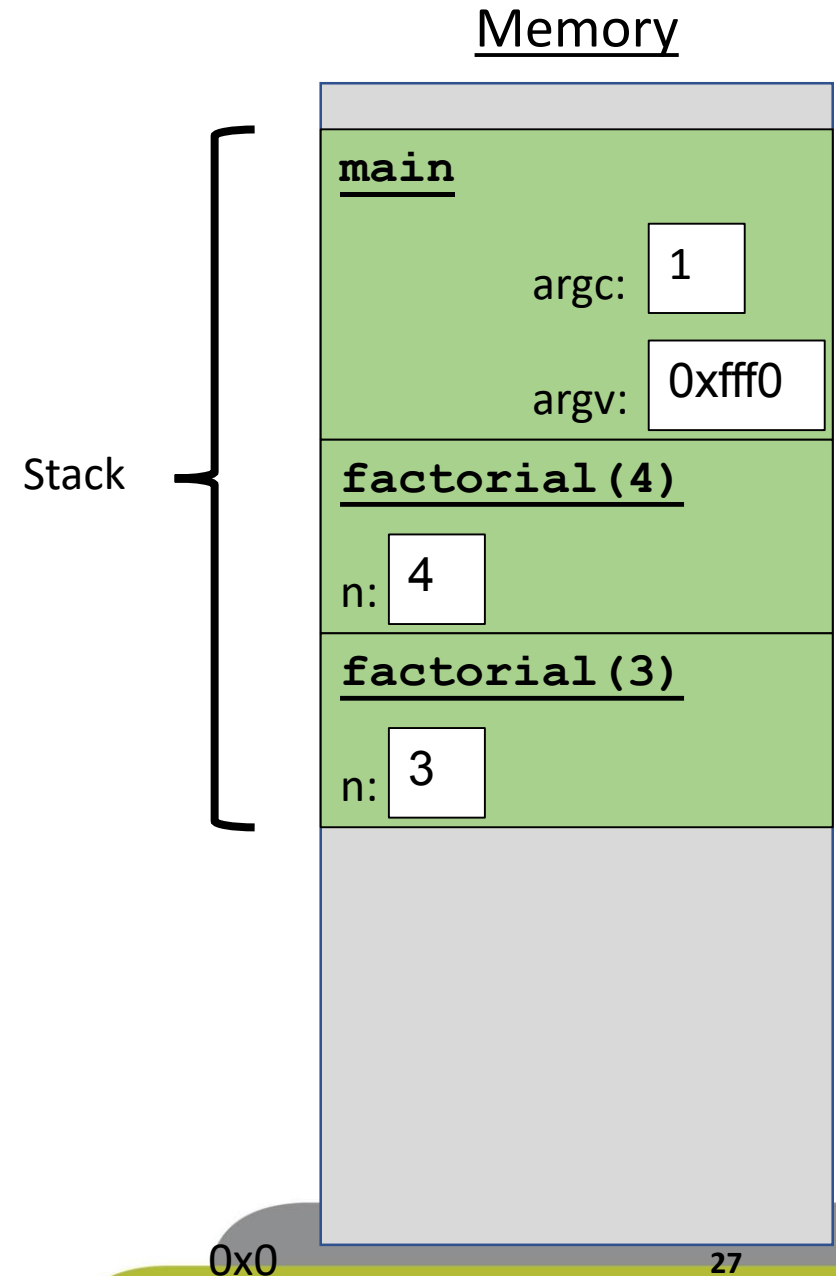
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



The Stack

- Each function call has its own stack frame to store its local variables.

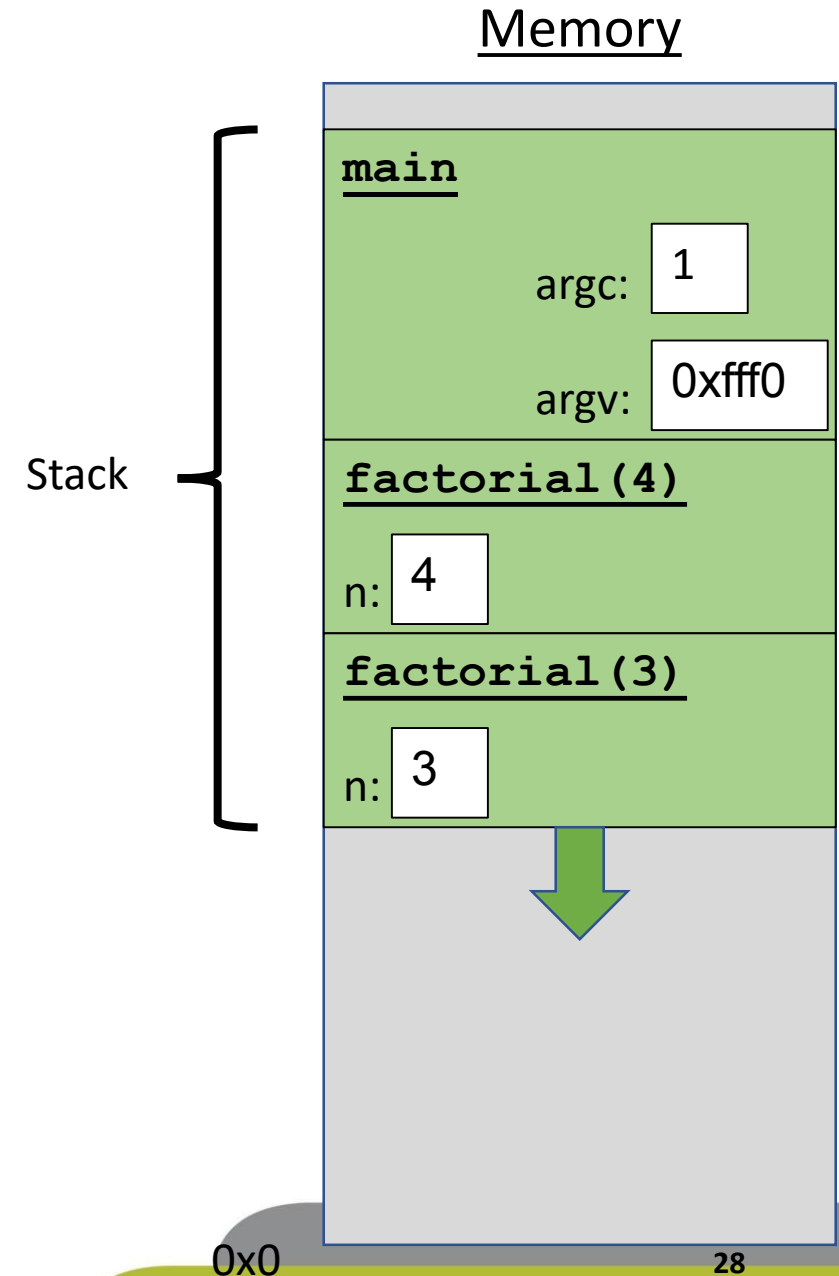
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



The Stack

- Each function call has its own stack frame to store its local variables.

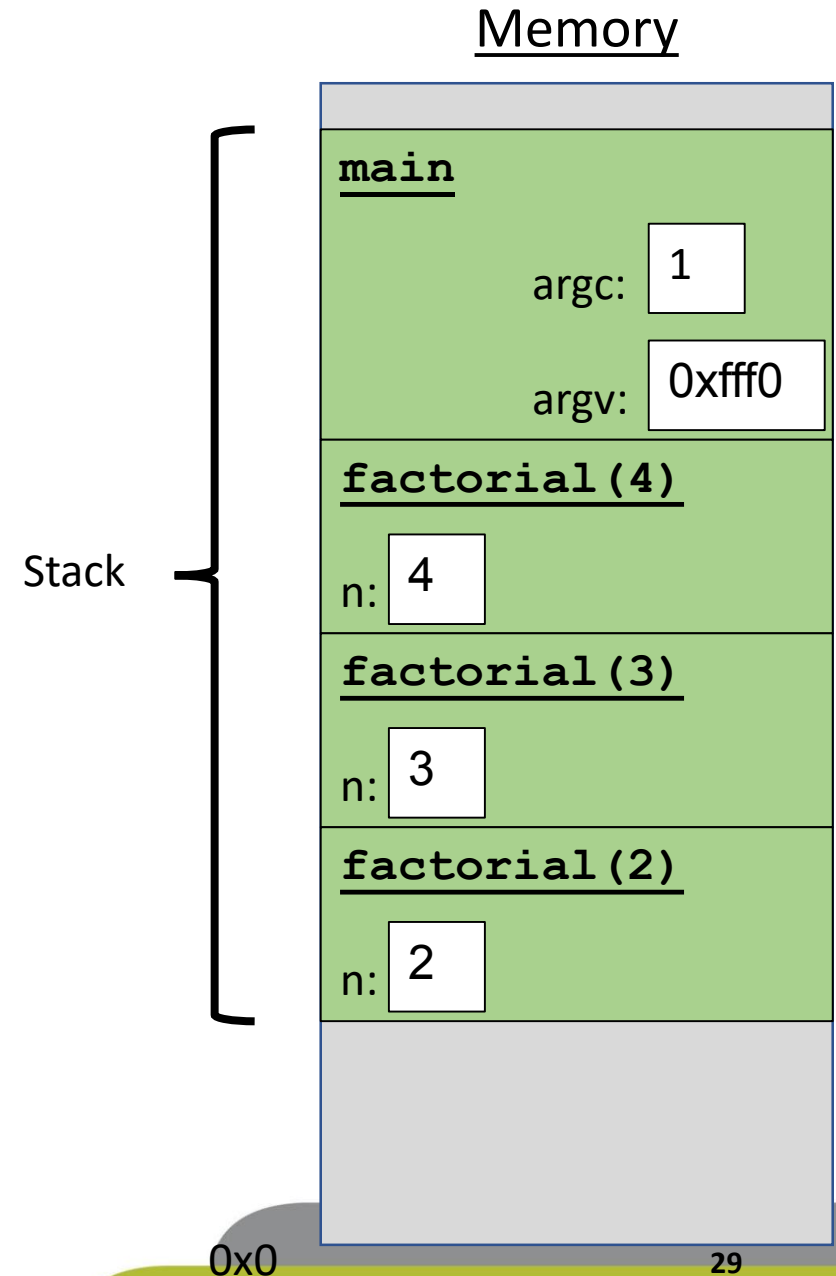
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



The Stack

- Each function call has its own stack frame to store its local variables.

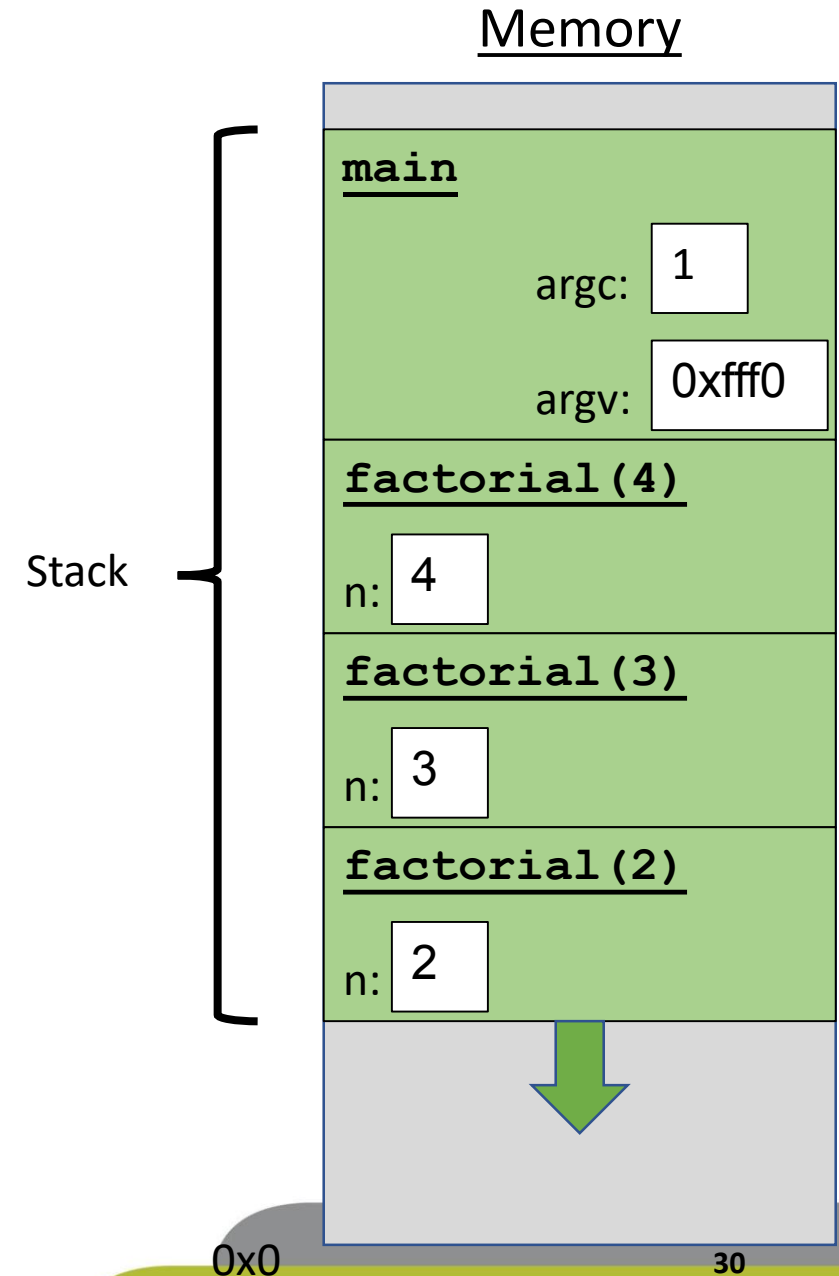
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



The Stack

- Each function call has its own stack frame to store its local variables.

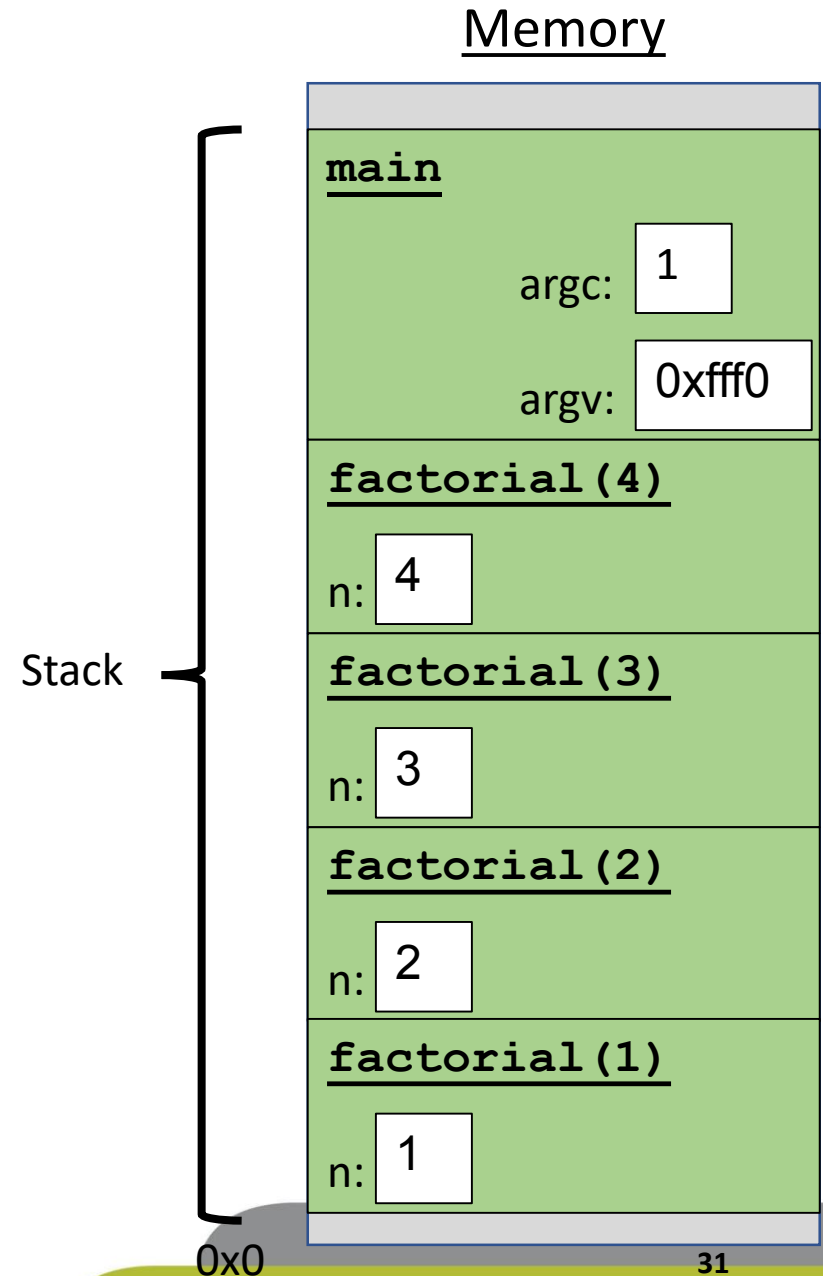
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



The Stack

- Each function call has its own stack frame to store its local variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```

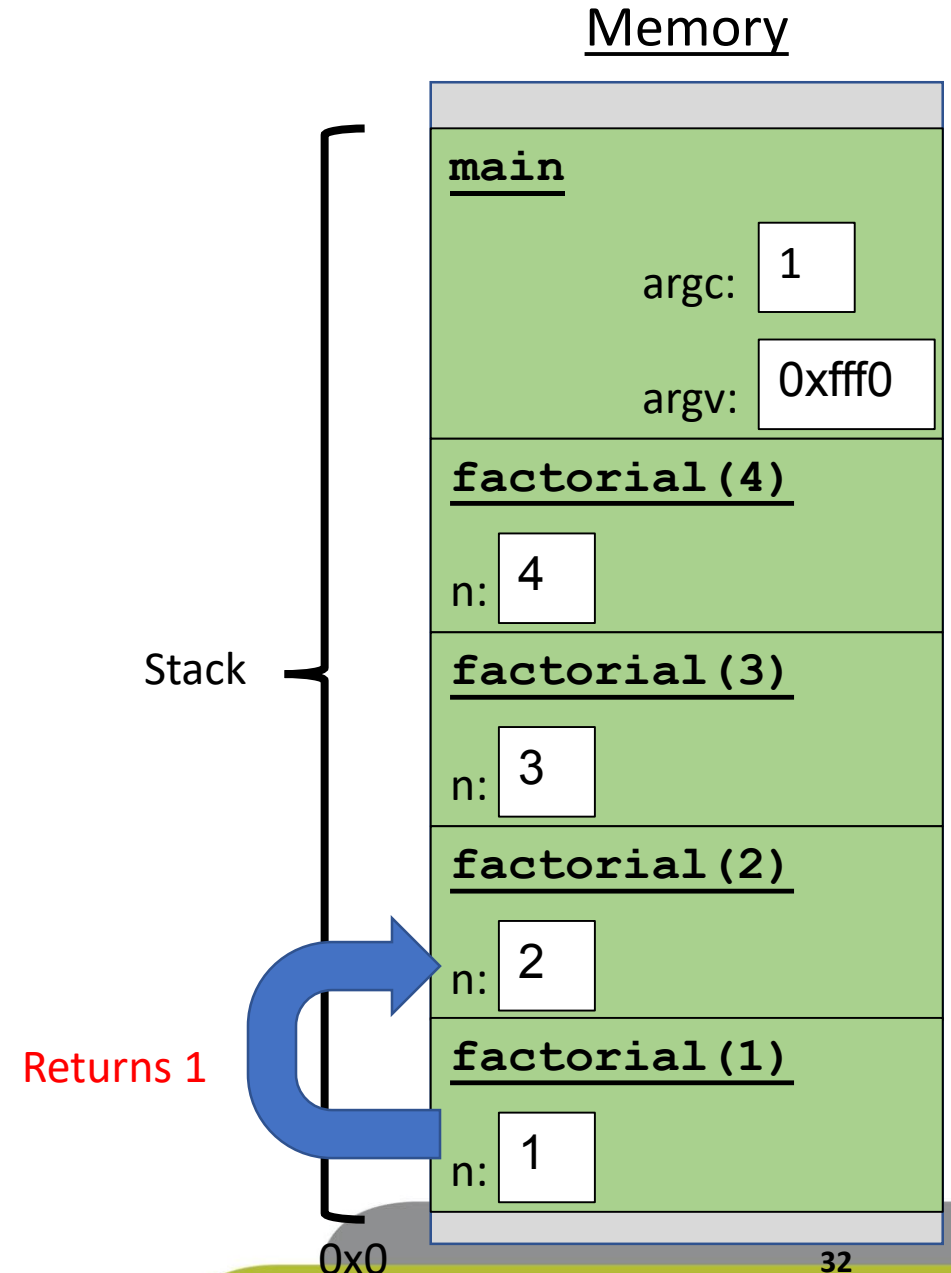


The Stack

- Each function call has its own stack frame to store its local variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

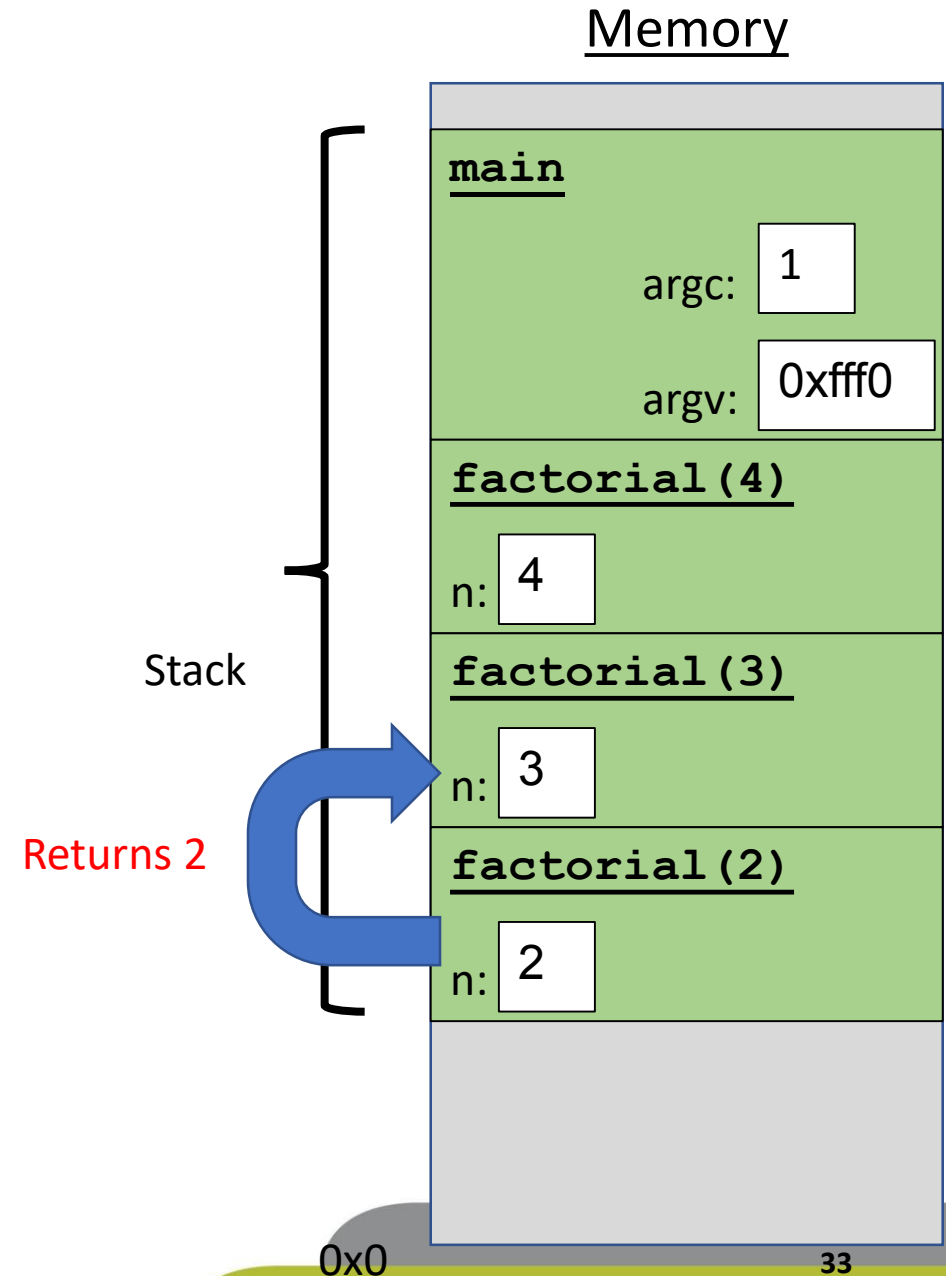
```
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



The Stack

- Each function call has its own stack frame to store its local variables.

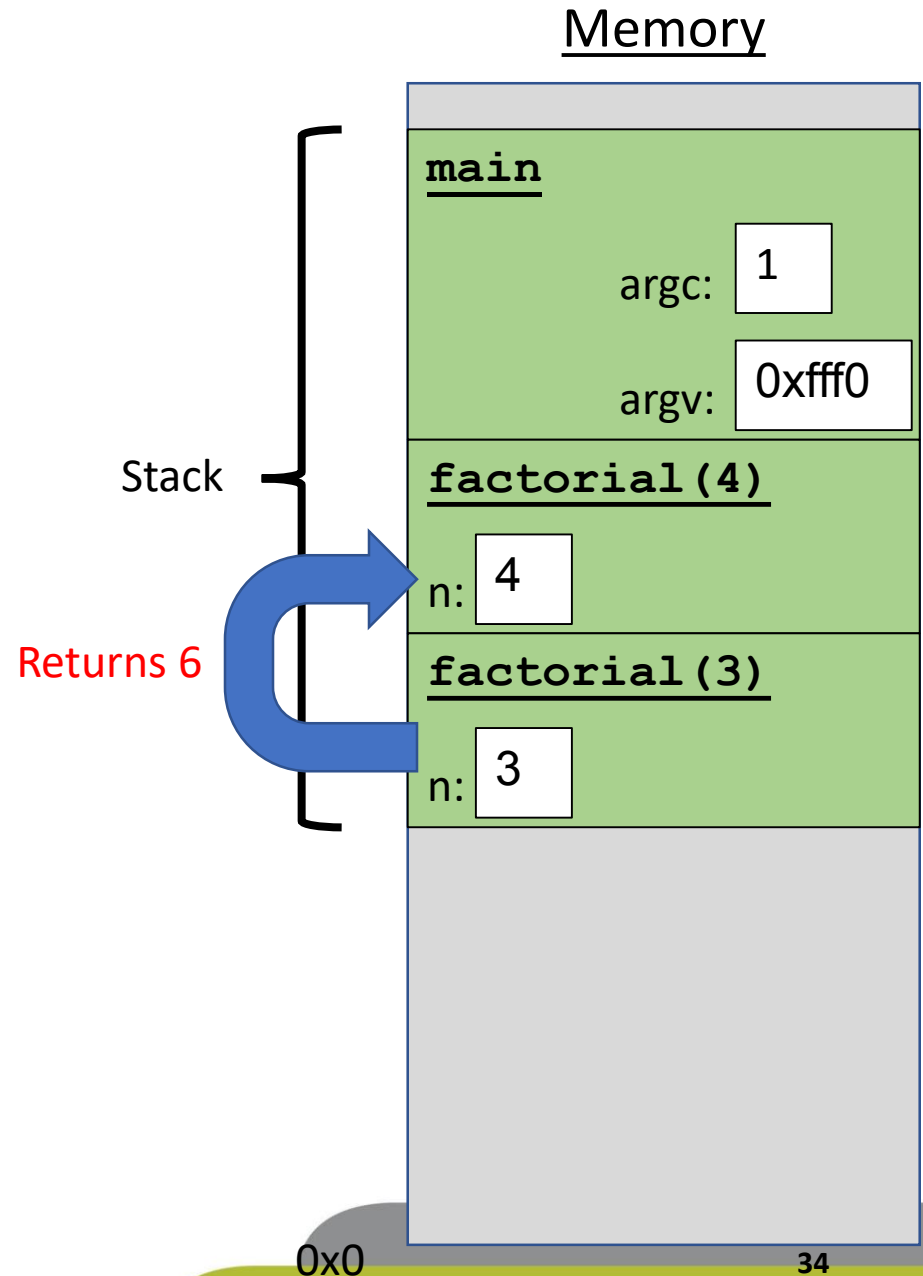
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



The Stack

- Each function call has its own stack frame to store its local variables.

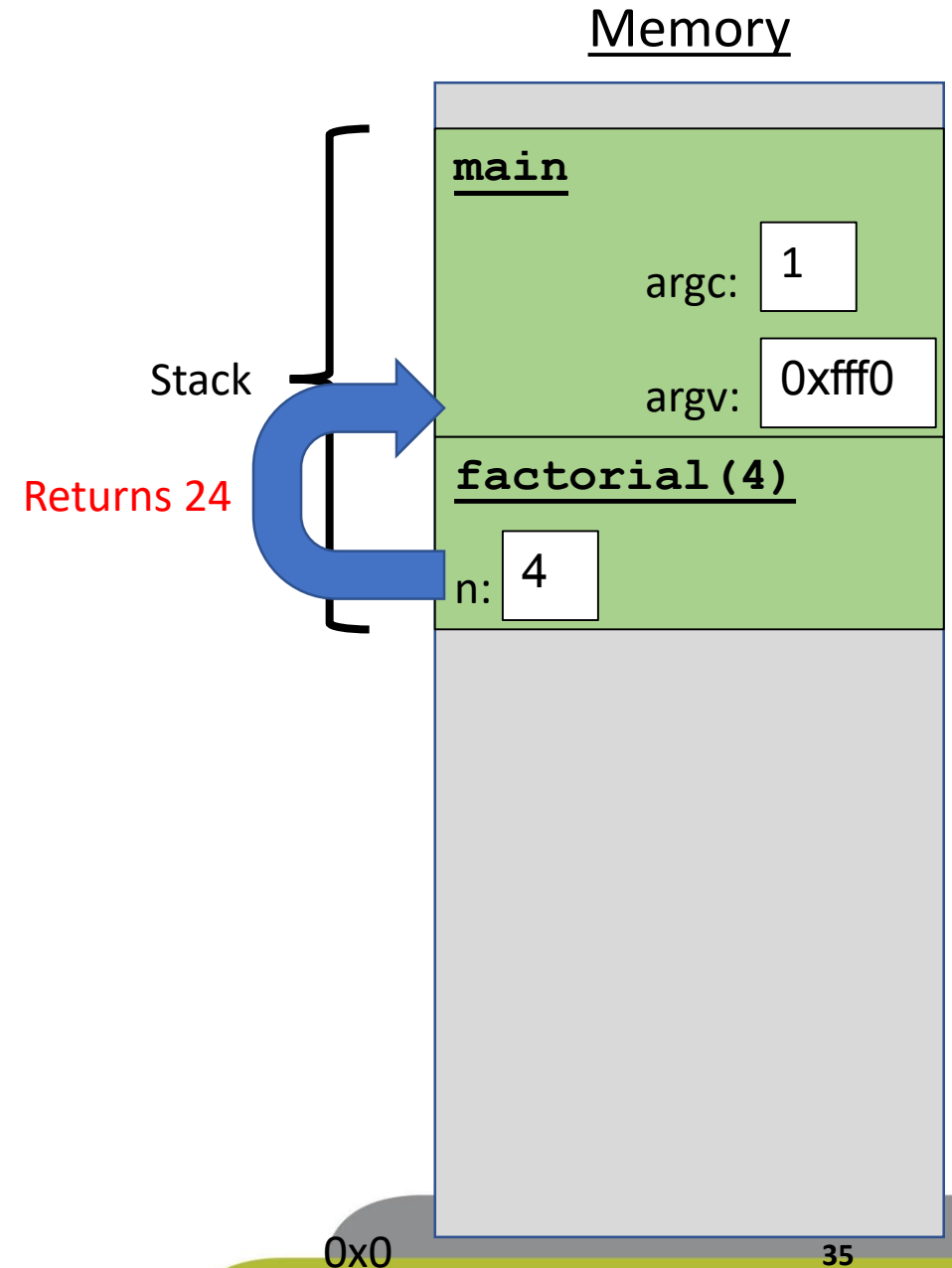
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



The Stack

- Each function call has its own stack frame to store its local variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```

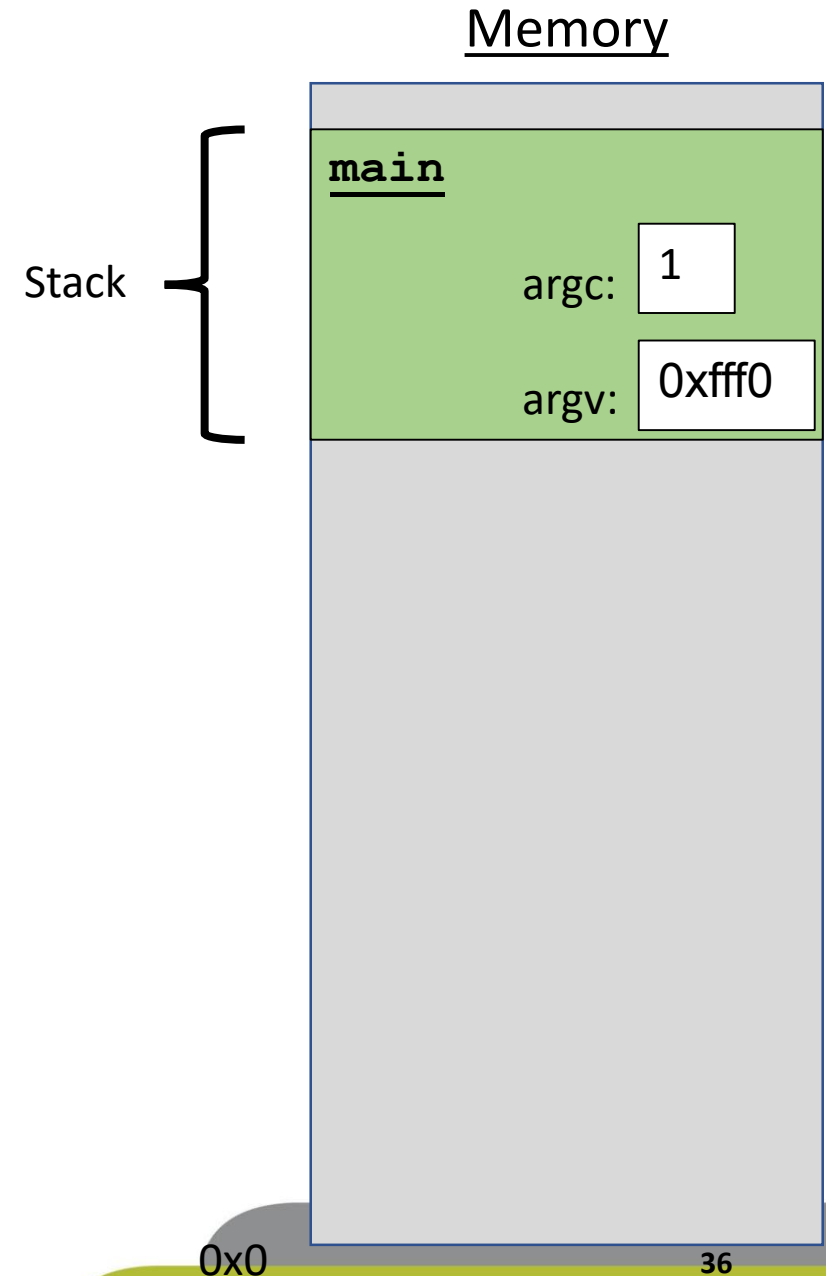


The Stack

- Each function call has its own stack frame to store its local variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

```
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```

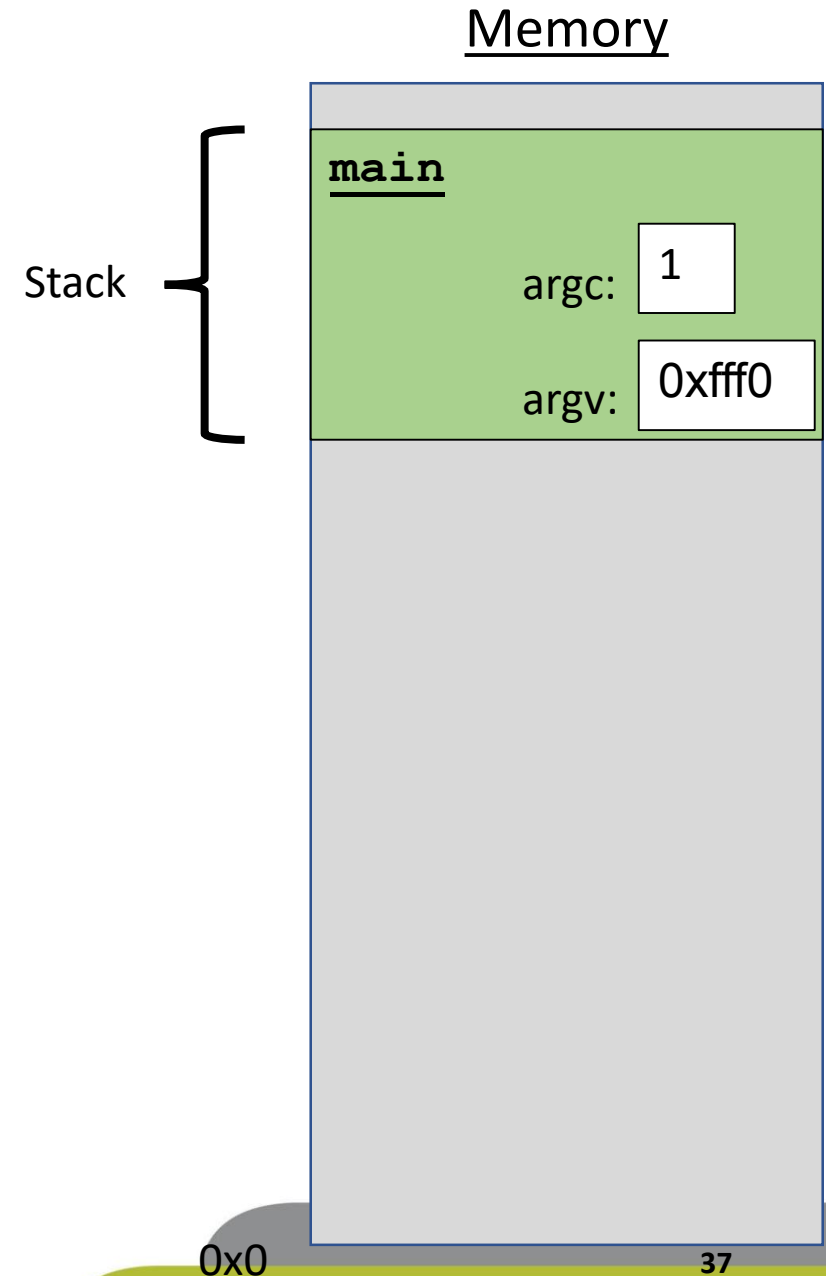


The Stack

- Each function call has its own stack frame to store its local variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

```
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



Memory

The Stack

- Each function call has its own stack frame to store its local variables.

```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}
```

```
int main(int argc, char *argv[]) {  
    factorial(4);  
    return 0;  
}
```



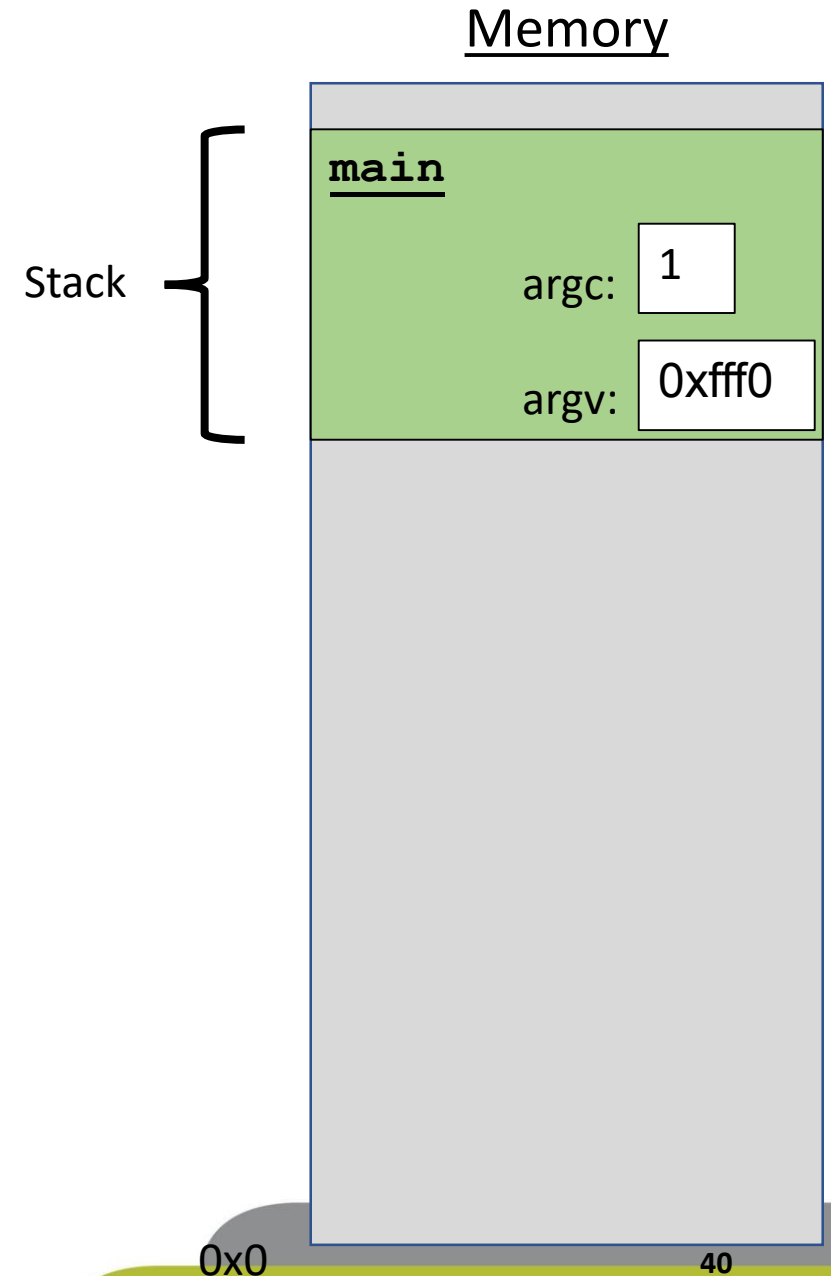
DOUGLAS COLLEGE

The Limitations of Stack

The Stack

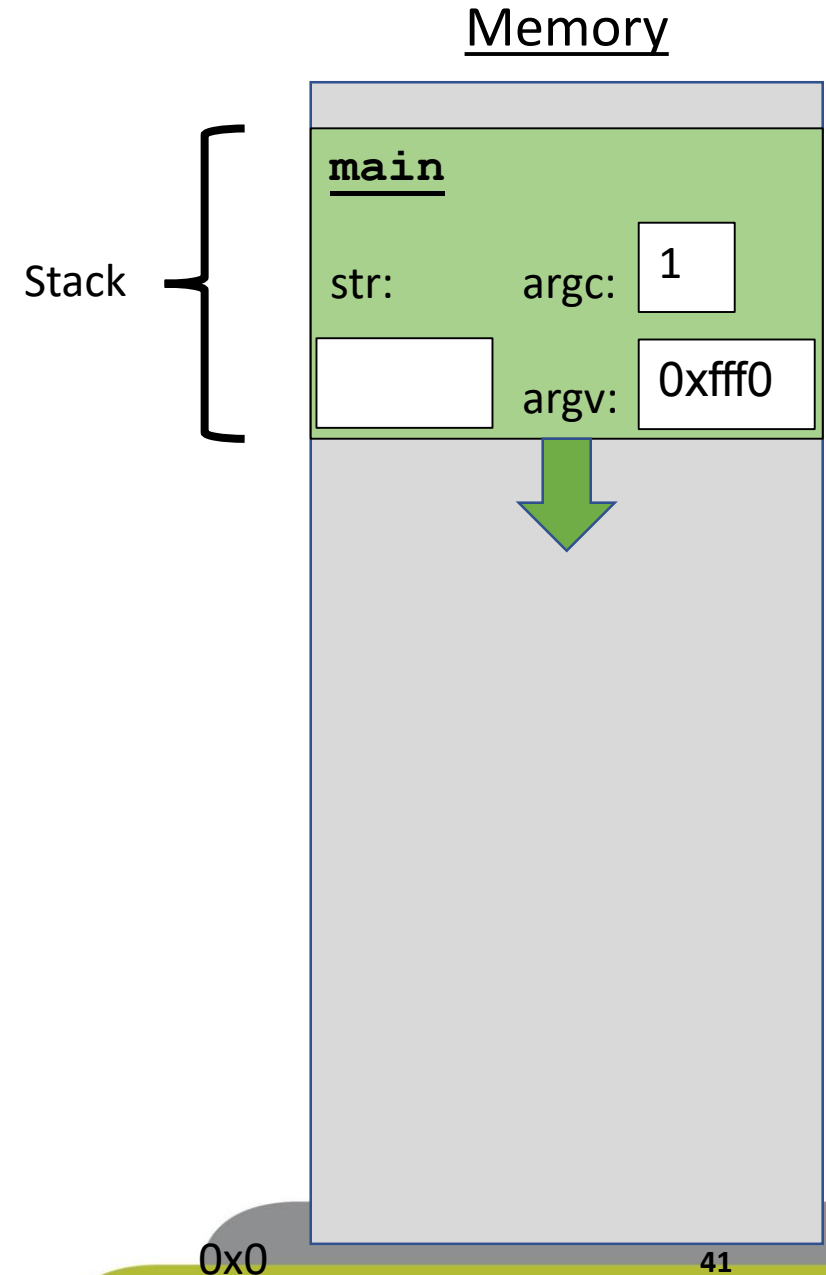
```
char* create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}
```

```
int main(int argc, char *argv[]) {  
    char *str = create_string('r', 4);  
    printf("%s", str); // want "rrrr"  
    return 0;  
}
```



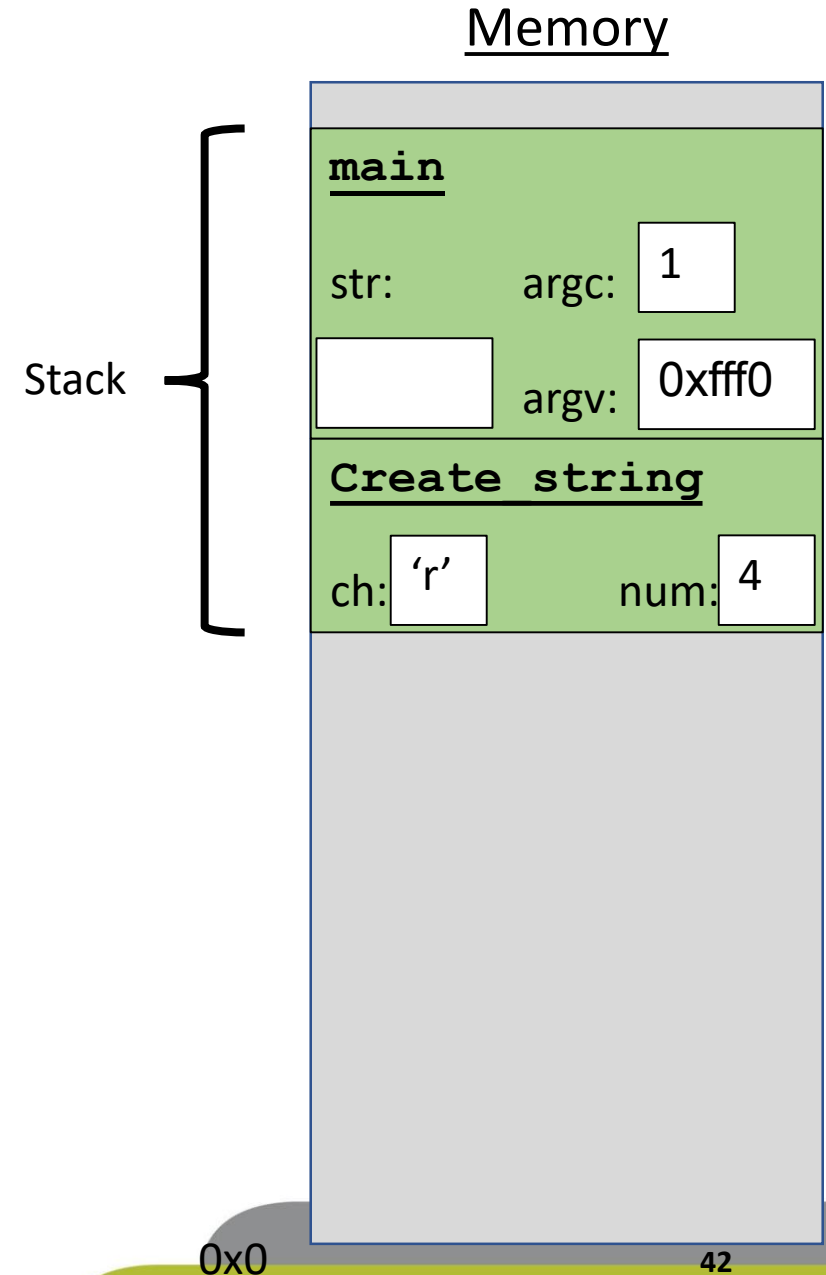
The Stack

```
char* create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('r', 4);  
    printf("%s", str); // want "rrrr"  
    return 0;  
}
```



The Stack

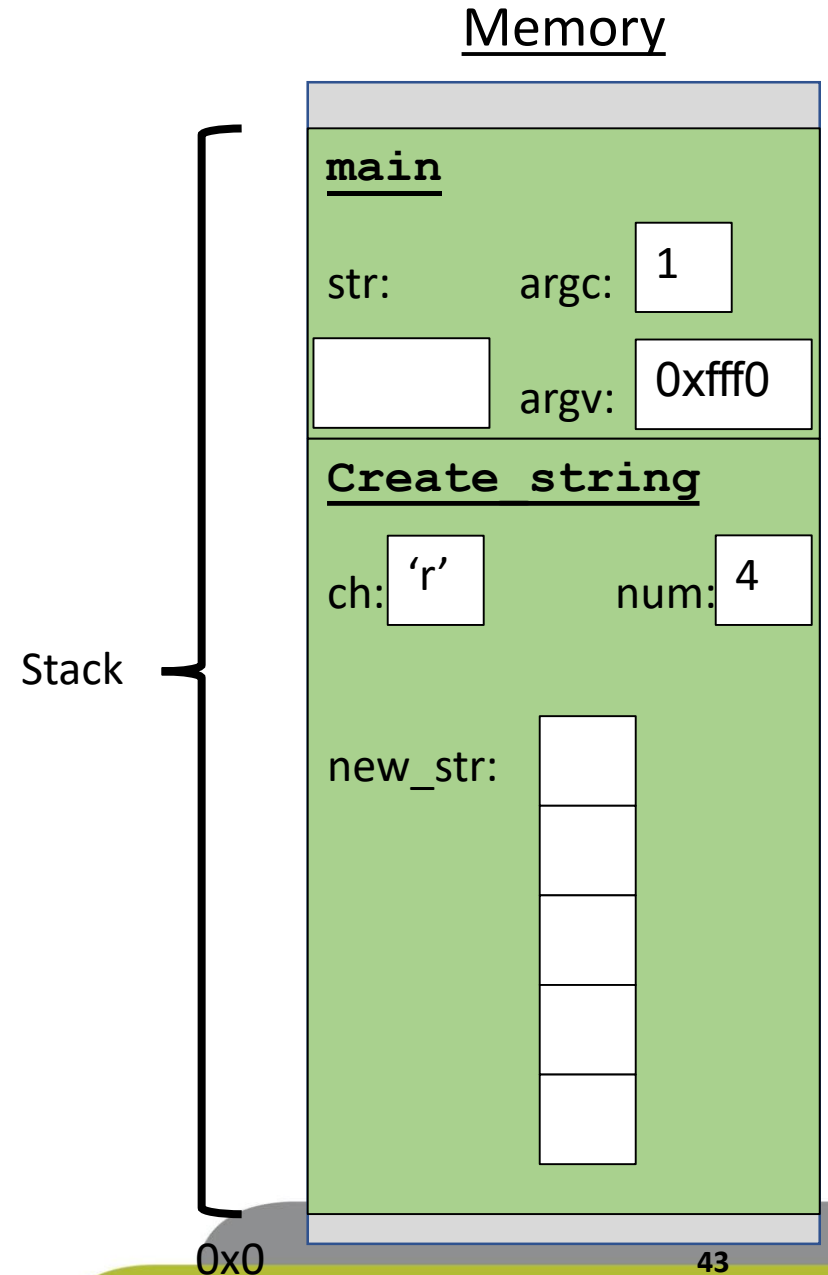
```
char* create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('r', 4);  
    printf("%s", str); // want "rrrr"  
    return 0;  
}
```



The Stack

```
char* create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('r', 4);
    printf("%s", str); // want "rrrr"
    return 0;
}
```

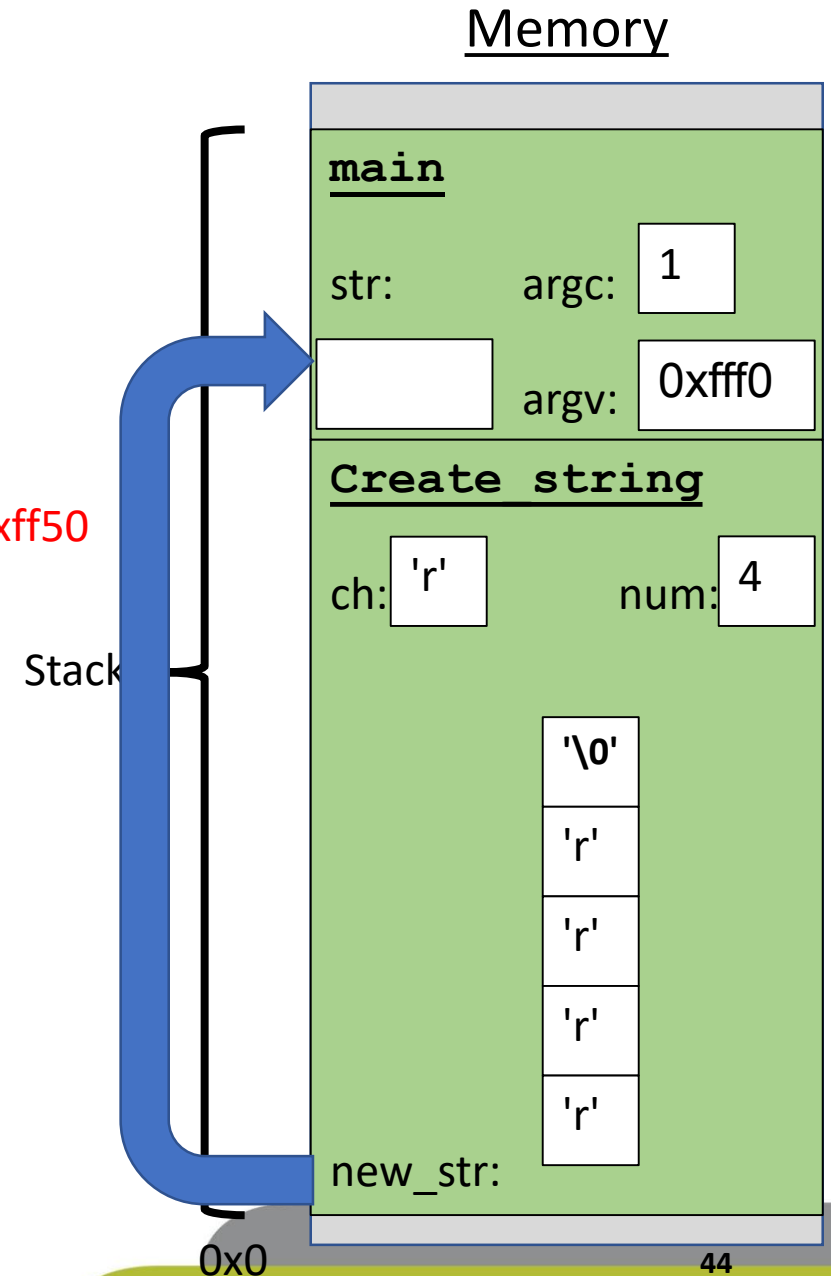


The Stack

```
char* create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('r', 4);
    printf("%s", str); // want "rrrr"
    return 0;
}
```

Returns 0xff50

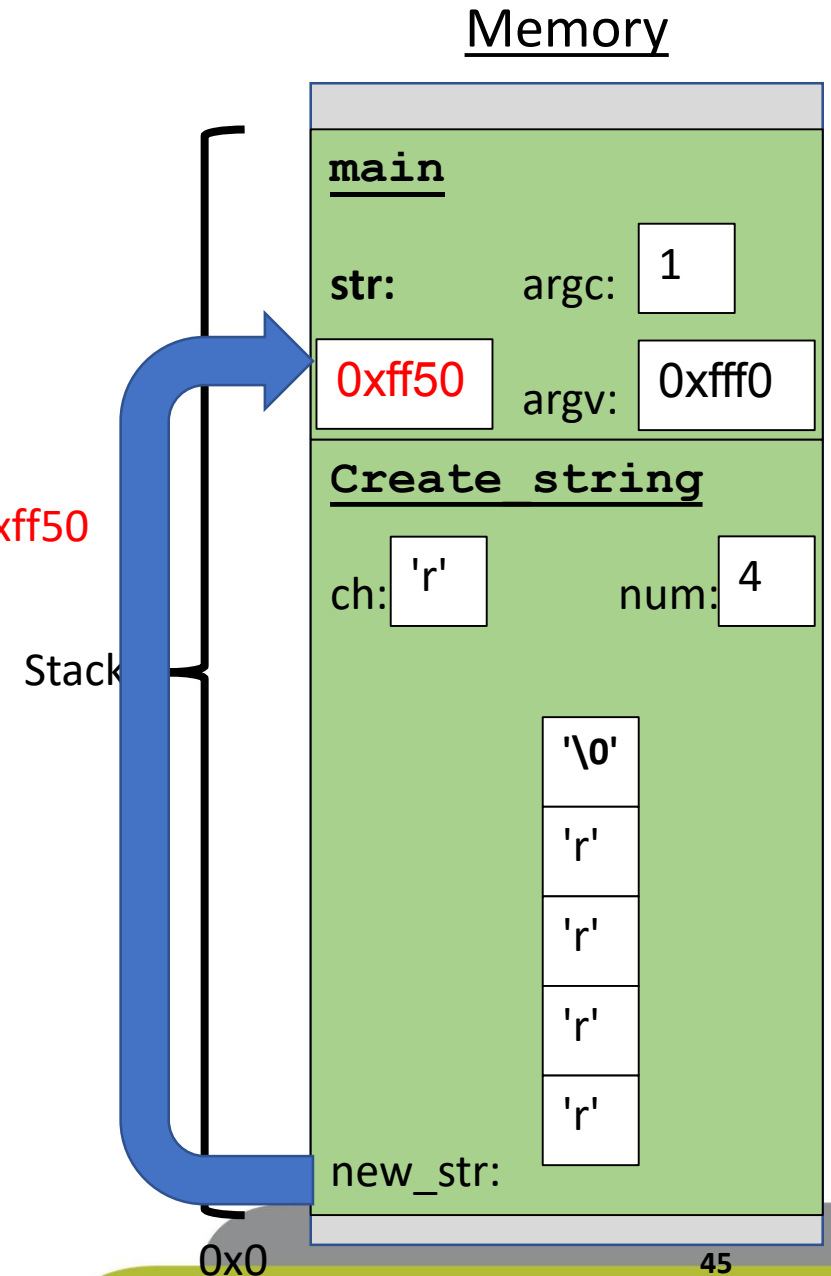


The Stack

```
char* create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('r', 4);
    printf("%s", str); // want "rrrr"
    return 0;
}
```

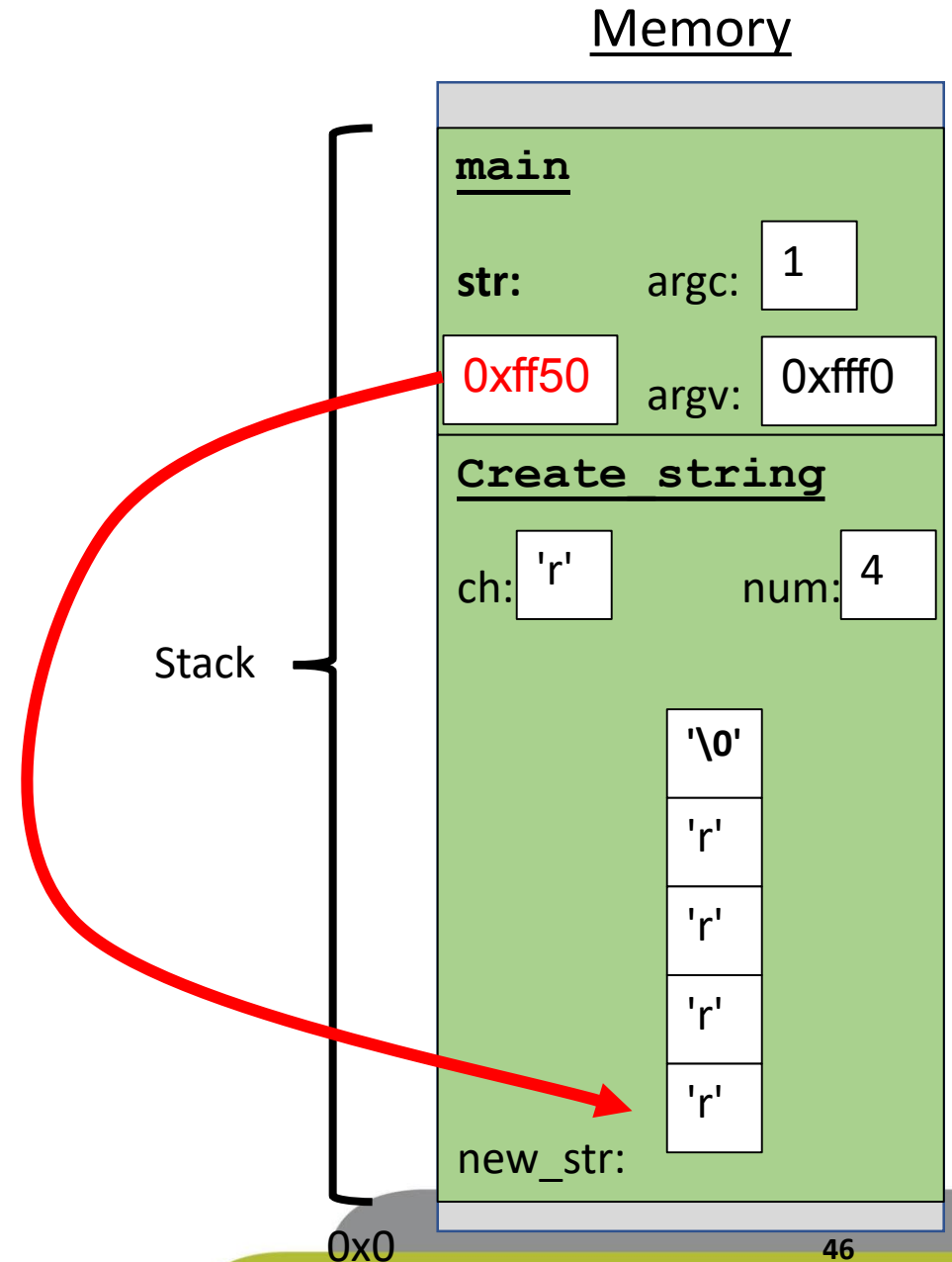
Returns 0xff50



The Stack

```
char* create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('r', 4);
    printf("%s", str); // want "rrrr"
    return 0;
}
```

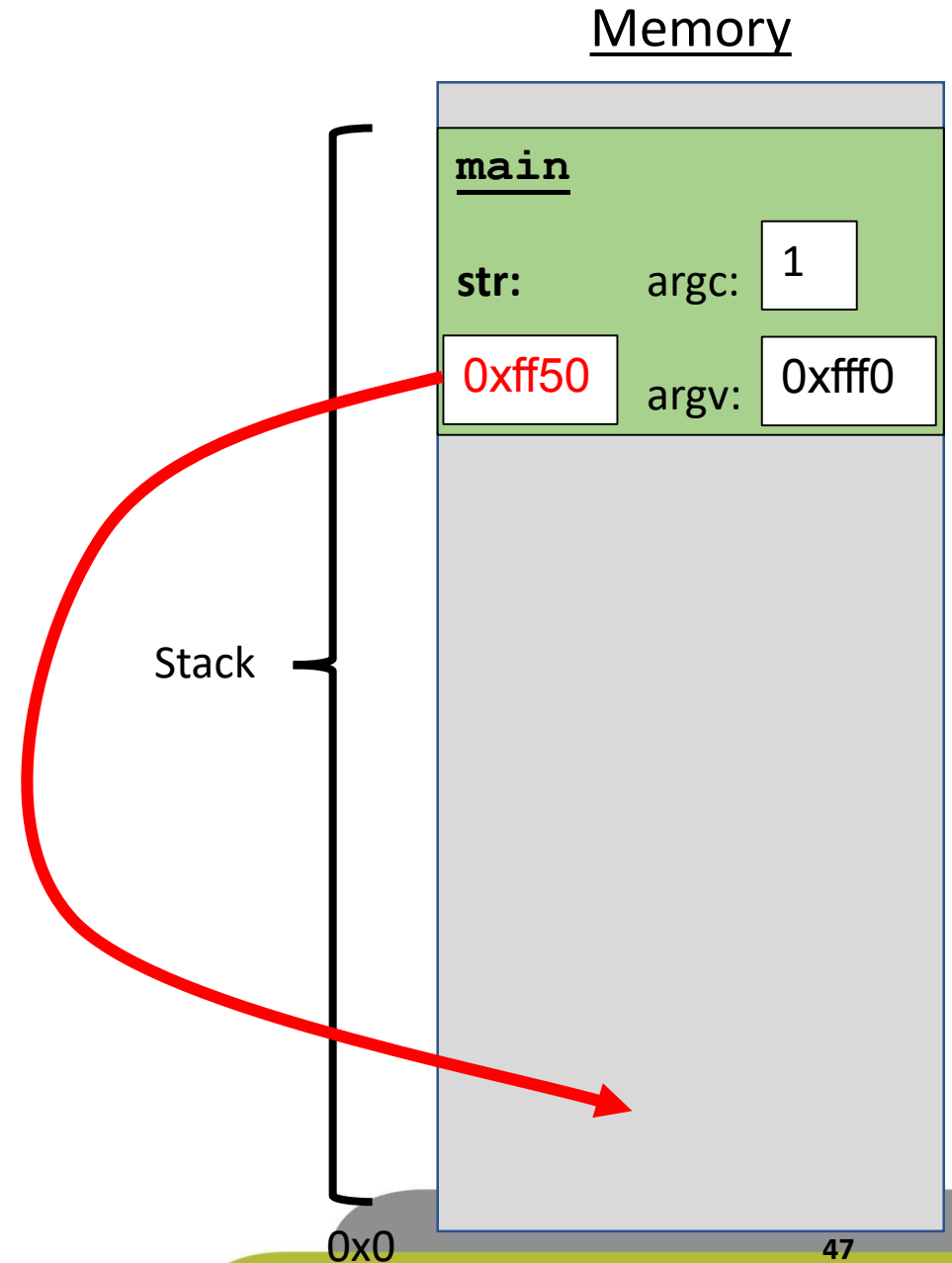


The Stack

```
char* create_string(char ch, int num) {
    char new_str[num + 1];
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}
```

```
}
```

```
int main(int argc, char *argv[]) {
    char *str = create_string('r', 4);
    printf("%s", str); // want "rrrr"
    return 0;
}
```

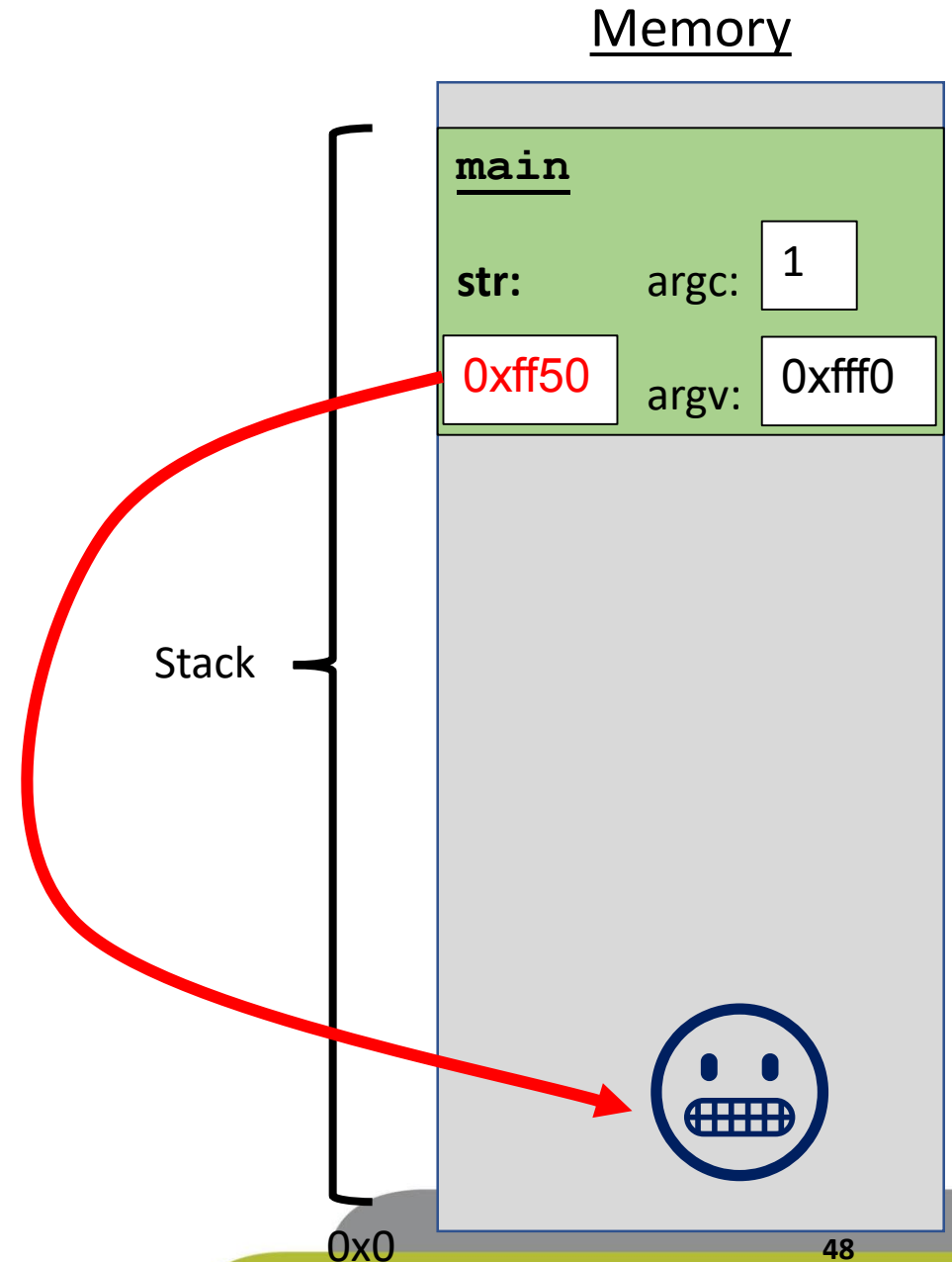


The Stack

```
char* create_string(char ch, int num) {  
    char new_str[num + 1];  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\0';  
    return new_str;  
}
```

```
}
```

```
int main(int argc, char *argv[]) {  
    char *str = create_string('r', 4);  
    printf("%s", str); // want "rrrr"  
    return 0;  
}
```



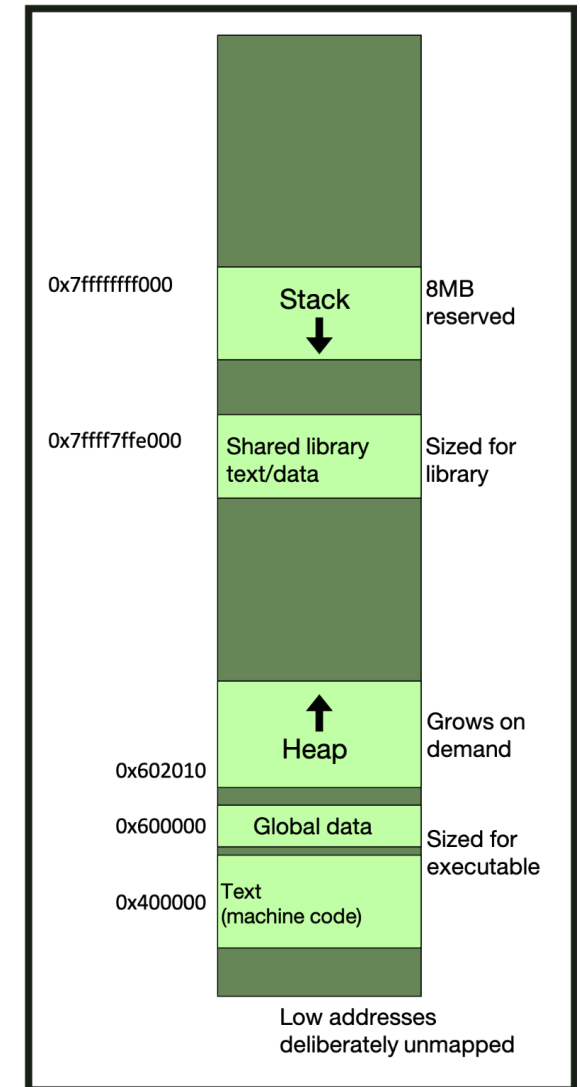


DOUGLAS COLLEGE

The Heap

The Heap

- The **heap** is a part of memory below the stack that we can manage ourselves.
- Unlike the stack, the memory only goes away when we release it ourselves.
- Unlike the stack, the heap grows upwards as more memory is allocated.
- The heap is dynamic memory – memory that can be allocated, resized, and freed during program execution (as opposed to compilation).



The `malloc()` Function

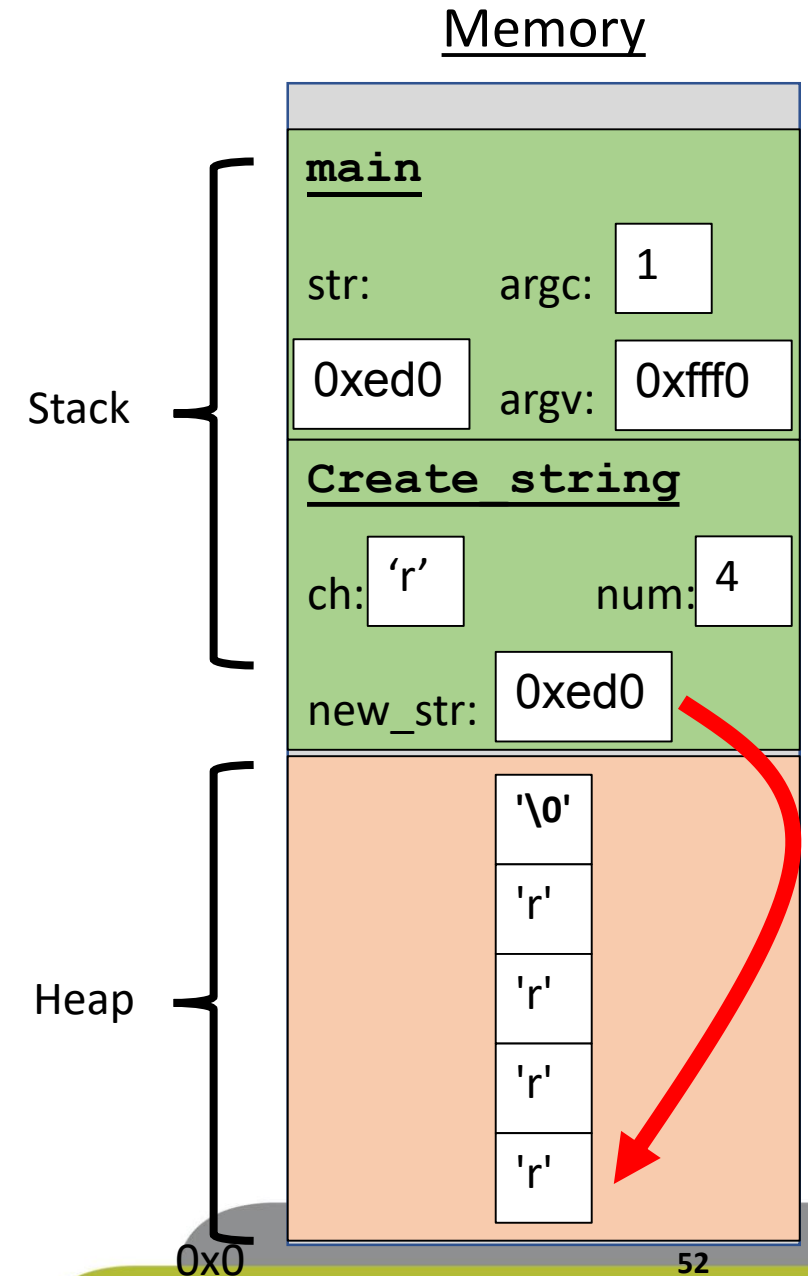
`void*malloc(size_t size);`

- To allocate memory on the heap, we can use the `malloc()` function (“memory allocate”) and specify the number of bytes we need.
- This function **returns a pointer to the starting address of the new memory**.
- It does not know whether it will be used as an array, a single block of memory, or a structure.
- **`void*`** means a pointer to generic memory.
- The memory is **not** cleared out before being getting allocated.
- If `malloc()` returns **`NULL`**, it means there was not enough memory for the allocation request.

The Heap

```
char* create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char)*(num+1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

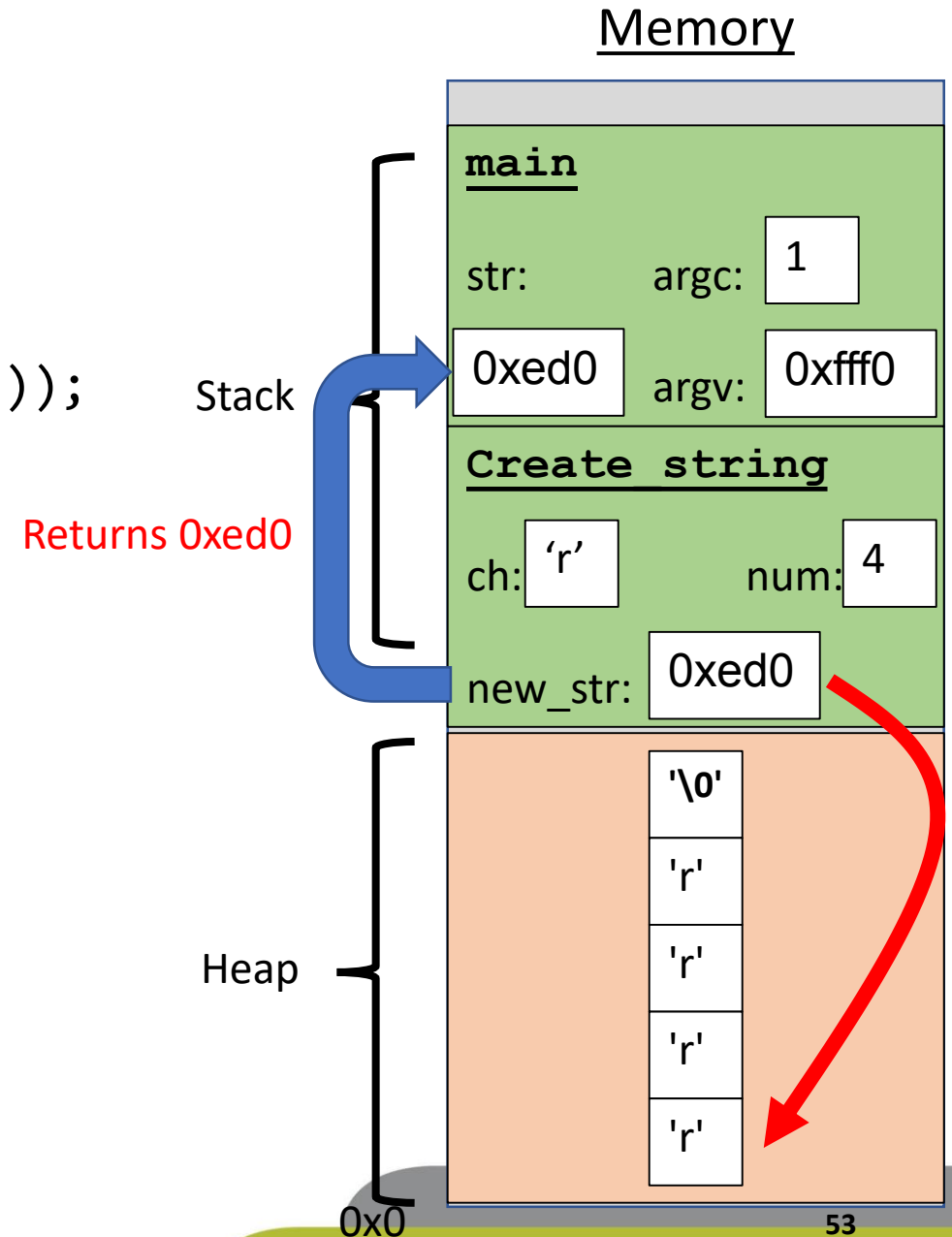
int main(int argc, char *argv[]) {
    char *str = create_string('r', 4);
    printf("%s", str); // want "rrrr"
    return 0;
}
```



The Heap

```
char* create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char)*(num+1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('r', 4);
    printf("%s", str); // want "rrrr"
    return 0;
}
```

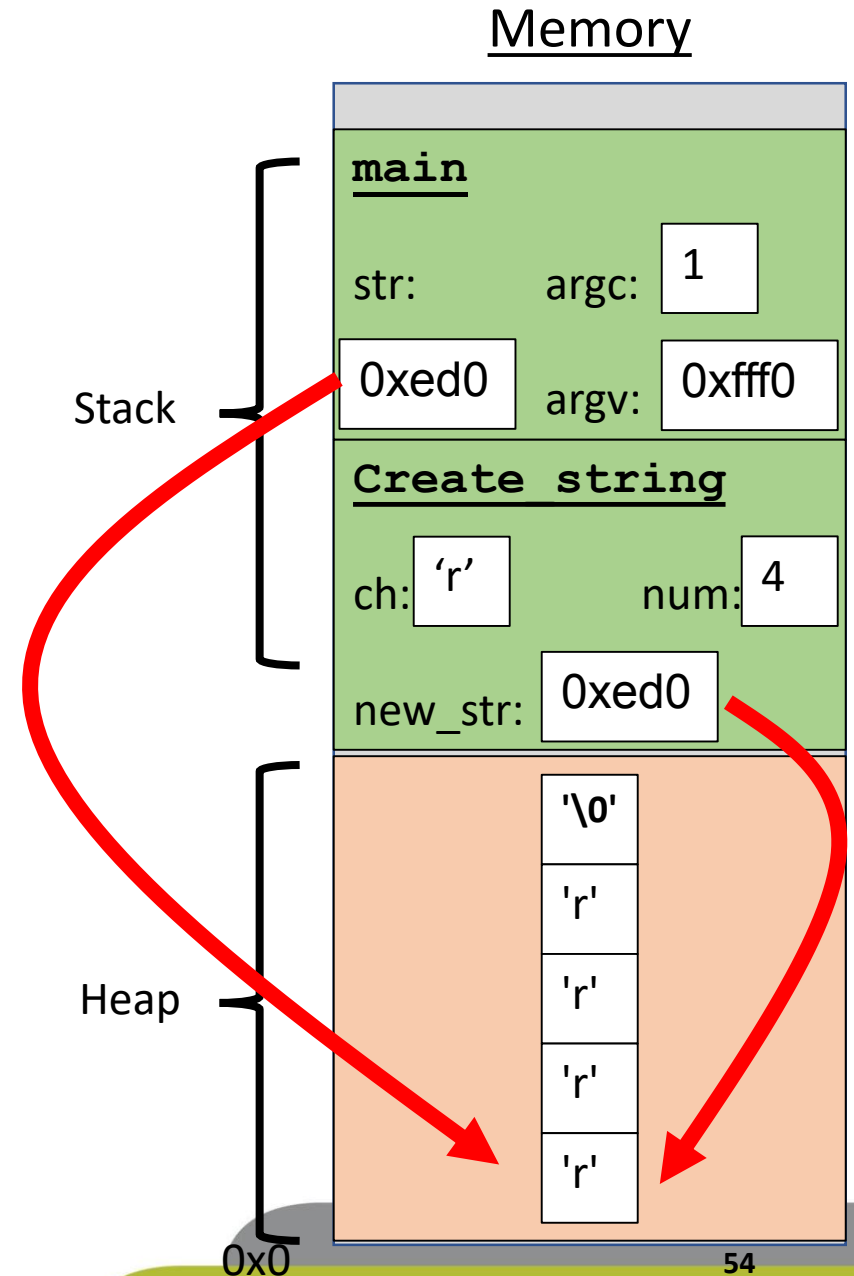


The Heap

```
char* create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char)*(num+1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}
```

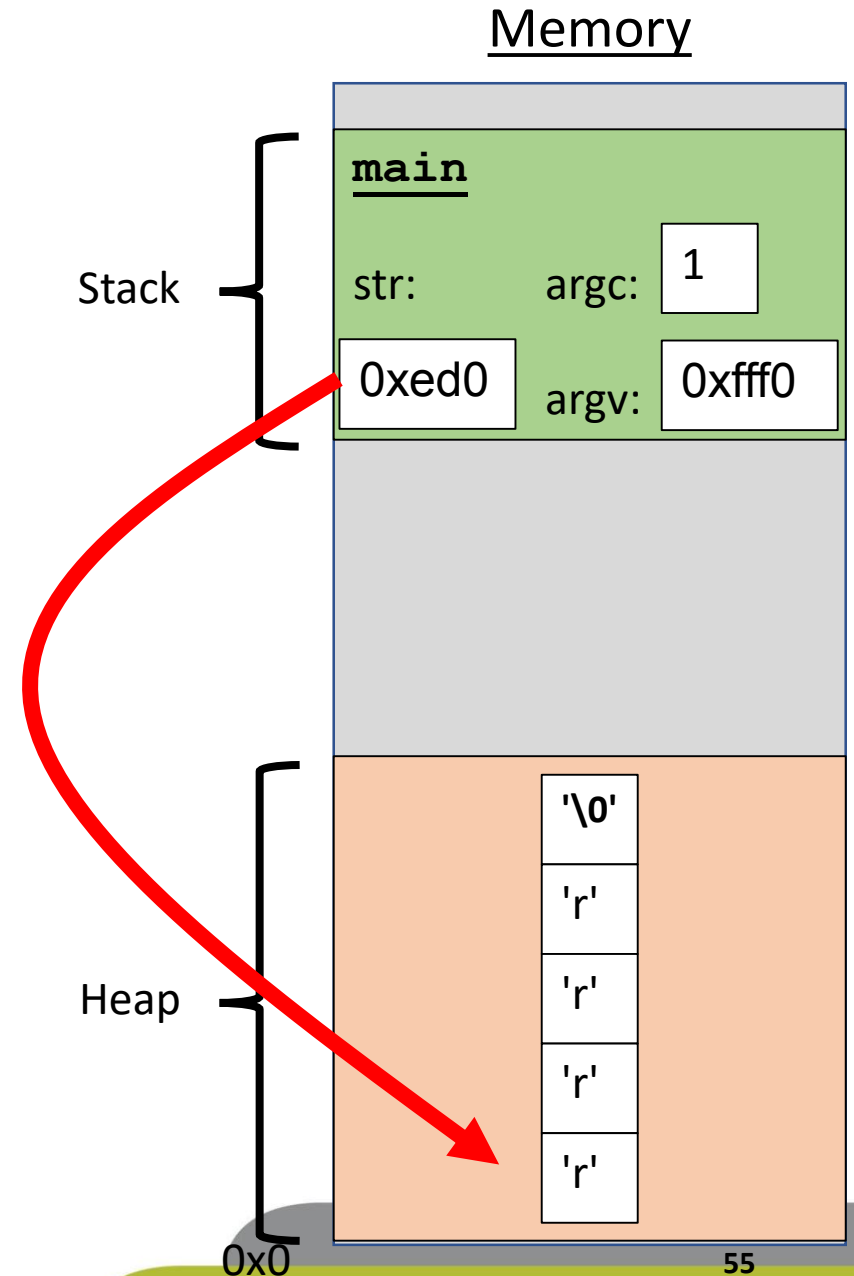
```
}
```

```
int main(int argc, char *argv[]) {
    char *str = create_string('r', 4);
    printf("%s", str); // want "rrrr"
    return 0;
}
```



The Heap

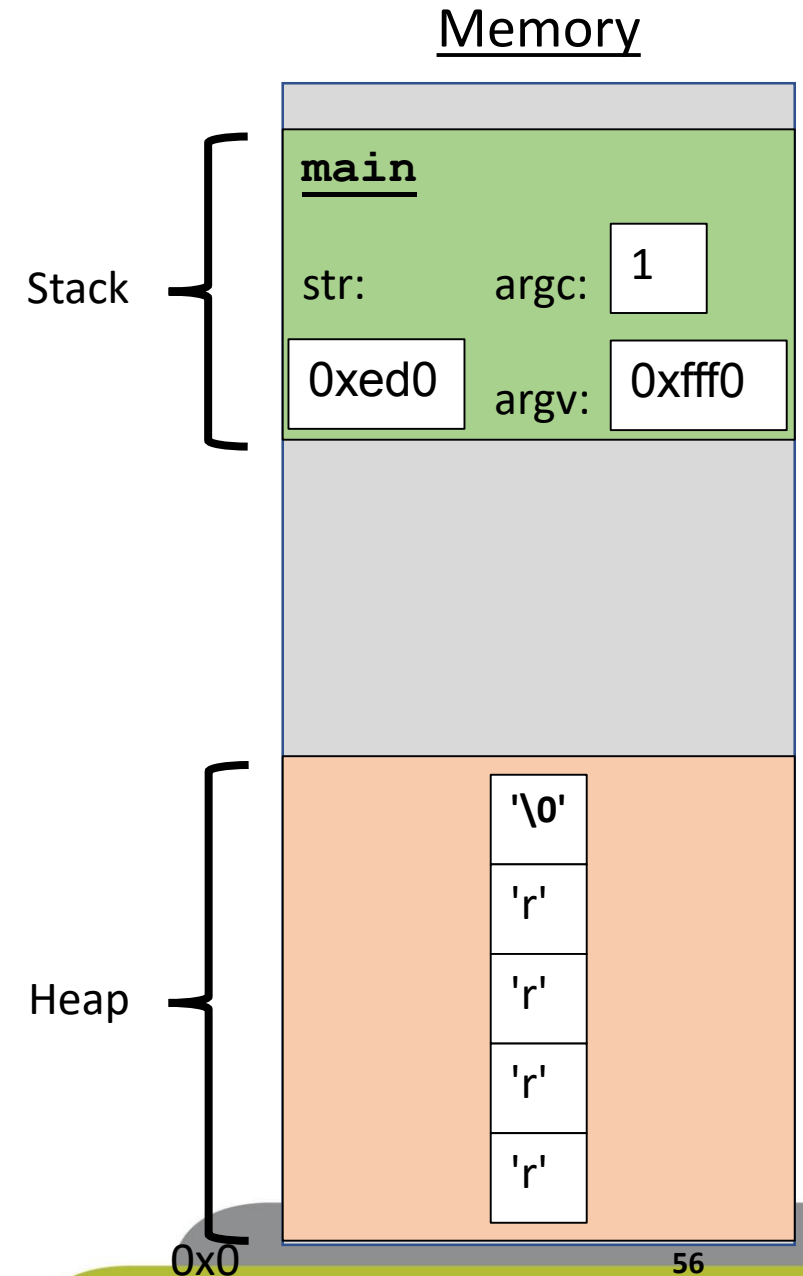
```
char* create_string(char ch, int num) {  
    char *new_str = malloc(sizeof(char)*(num+1));  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('r', 4);  
    printf("%s", str); // want "rrrr"  
    return 0;  
}
```



The Heap

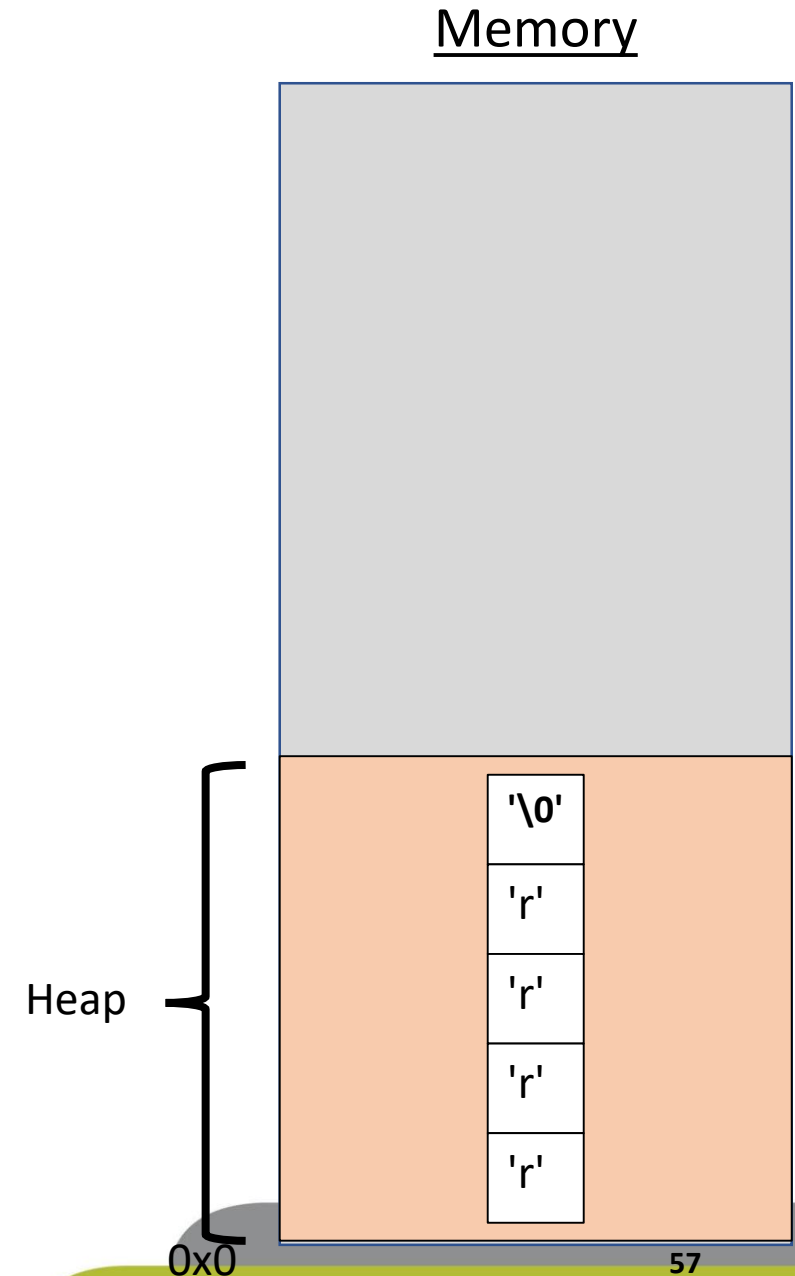
```
char* create_string(char ch, int num) {
    char *new_str = malloc(sizeof(char)*(num+1));
    for (int i = 0; i < num; i++) {
        new_str[i] = ch;
    }
    new_str[num] = '\0';
    return new_str;
}

int main(int argc, char *argv[]) {
    char *str = create_string('r', 4);
    printf("%s", str); // want "rrrr"
    return 0;
}
```



The Heap

```
char* create_string(char ch, int num) {  
    char *new_str = malloc(sizeof(char)*(num+1));  
    for (int i = 0; i < num; i++) {  
        new_str[i] = ch;  
    }  
    new_str[num] = '\\0';  
    return new_str;  
}  
  
int main(int argc, char *argv[]) {  
    char *str = create_string('r', 4);  
    printf("%s", str); // want "rrrr"  
    return 0;  
}
```



The calloc() Function

- The **calloc()** function allocates contiguous multiple blocks of memory of the same size, and **initializes the allocated bytes to zero.**
- Similar to the **malloc()** function, **calloc()** also returns a pointer to the first byte of allocated memory.
- The **calloc()** function is more expensive than **malloc()** because it zeros out memory.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdlib.h>
#include <stdio.h>
int main()
{
    int* ptr, i;
    printf("Enter an integer:\n");
    scanf("%d", &i);

    ptr = (int*)calloc(i, sizeof(int));
    if (ptr != NULL)
        for (int n = 0; n < i; n++)
        {
            printf("%d ", *(ptr + n));
        }
}
```

The `realloc()` Function

- The **`realloc()`** function allows us to expand (or reduce) the size of a previously allocated block of memory while retaining (or truncating) the original data.
- There is no guarantee that the newly allocated memory will be in the same location as the previously allocated memory.
- It is therefore important to keep track of the pointer returned by the `realloc()` function.
- If the reallocation of memory is unsuccessful, a **`NULL`** pointer is returned

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char* ptr;
    ptr = (char*)malloc(7 * sizeof(char));
    strcpy(ptr, "Hello!");
    printf(" %s, Address = %p\n", ptr, ptr);

    ptr = (char*)realloc(ptr, 15 * sizeof(char));
    strcat(ptr, " World!");
    printf(" %s, Address = %p\n", ptr, ptr);
    free(ptr);
    return 0;
}
```

The free() Function

- When memory is allocated dynamically using **malloc()** or **calloc()**, it must be manually released using the **free()** function (in the reverse order of creation).
- Failing to release memory could lead to segmentation faults or insufficient memory.

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    int* ptr = (int*)malloc(10 * sizeof(int));
    if (ptr != NULL) {
        *(ptr + 2) = 50;
        printf("%d\n\n", *(ptr + 2));
    }
    free(ptr);
}
```

The free() Function

Even if we have multiple pointers to the same block of memory, each memory block should only be freed **once**.

```
char *bytes = malloc(4);  
char *ptr = bytes;
```

```
...  
free(bytes);
```



```
...  
free(ptr);
```



We must free the address we received in the previous allocation call; we cannot free part of a previous allocation.

```
char *bytes = malloc(4);  
char *ptr = malloc(10);
```

```
...  
free(bytes);
```



```
...  
free(ptr + 1);
```



Dynamic Memory Allocation of 2D Arrays

- Here **rows** is the number of rows, and **cols** is the number of columns for our 2D array.
- Expression ***(arr + i*cols + j)** should remind us of how C interprets multidimensional arrays...

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

int main()
{
    int rows = 3, cols = 7;
    int* arr = (int*)malloc(rows * cols * sizeof(int));

    int i, j, count = 0;
    for (i = 0; i < rows; i++)
        for (j = 0; j < cols; j++)
            *(arr + i * cols + j) = ++count;

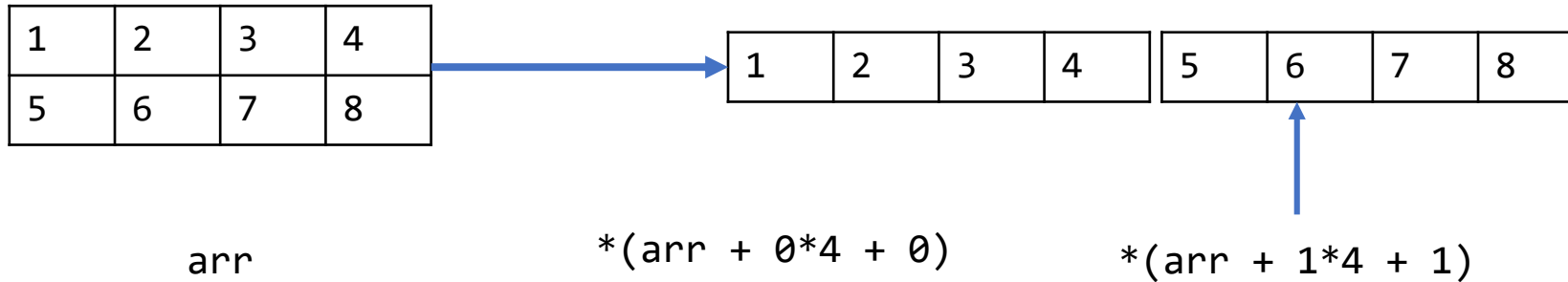
    for (i = 0; i < rows; i++)
        for (j = 0; j < cols; j++)
            printf("%d\n", *(arr + i * cols + j));

    return 0;
}
```

Dynamic Memory Allocation of 2D Arrays

- What is happening here?

$*(arr + i*cols + j)$



References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2395

Intro to Computer Architecture

Lecture 7

Plan for Today

- Memory Technologies
 - Static Random Access Memory (SRAM)
 - Dynamic Random Access Memory (DRAM)
 - Read and Write Transactions between CPU and DRAM
 - Disk Storage
 - Solid State Disks (SSD)
 - The Gap between CPU and Memory Performance
- Locality to the Rescue
- How the Principle of Locality Leads to the Concept of a Memory Hierarchy
- Memory Hierarchies
- Caching
 - Direct-mapped and set associative

Static RAM and Dynamic RAM

Random-Access Memory (RAM)

- **Key features of RAM:**
 - RAM is traditionally packaged as a chip.
 - Basic storage unit is normally a **cell** (one bit per cell).
 - Multiple RAM chips form a memory module.
- **RAM comes in two varieties:**
 - **SRAM** (Static RAM): used for cache memories, both on and off the CPU chip.
 - **DRAM** (Dynamic RAM): DRAM is used for the main memory plus the frame buffer of a graphics system.
- A desktop system will typically have no more than a few tens of megabytes of SRAM, but hundreds or thousands of megabytes of DRAM.

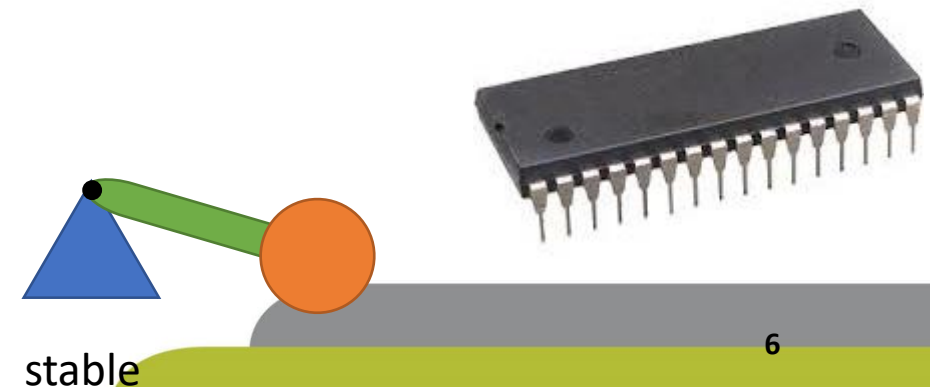
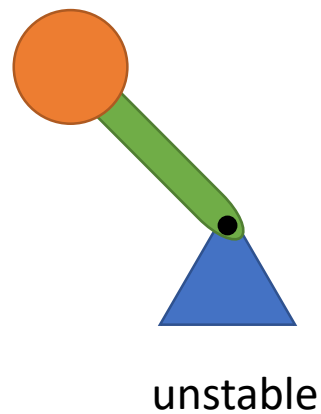
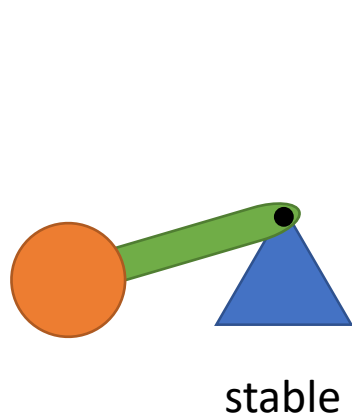
SRAM

- SRAM stores each bit in a **bistable** memory cell. Each cell is a six-transistor circuit.
- This circuit has the property that it can stay indefinitely in either of two different voltage configurations, or states.
- All other states are unstable because starting from any other voltage configuration, the circuit will quickly move toward one of the stable states.
- Such a memory cell is analogous to the inverted pendulum.
- SRAM memory cells will retain their value indefinitely for as long as power is on.
- Even when a disturbance perturbs the voltages, the circuit will return to the stable value when the disturbance is removed.



SRAM

- SRAM stores each bit in a **bistable** memory cell. Each cell is a six-transistor circuit.
- This circuit has the property that it can stay indefinitely in either of two different voltage configurations, or states.
- All other states are unstable because starting from any other voltage configuration, the circuit will quickly move toward one of the stable states.
- Such a memory cell is analogous to the inverted pendulum.
- SRAM memory cells will retain their value indefinitely for as long as power is on.
- Even when a disturbance perturbs the voltages, the circuit will return to the stable value when the disturbance is removed.



DRAM

- DRAM stores each bit as charge on a capacitor. This capacitor is very small.
- DRAM storage can be made very dense as each cell consists of a capacitor and a single access transistor.
- DRAM memory cell is very sensitive to any disturbance. If the capacitor voltage is disturbed, it will never recover.
- Exposure to light will cause the capacitor voltages to change. In fact, the sensors in digital cameras are essentially arrays of DRAM cells.
- A DRAM cell loses its charge within a time period of around 10 to 100 milliseconds.
- This retention time is still quite long since processor clock times are on the order of nanoseconds.



SRAM vs. DRAM Summary

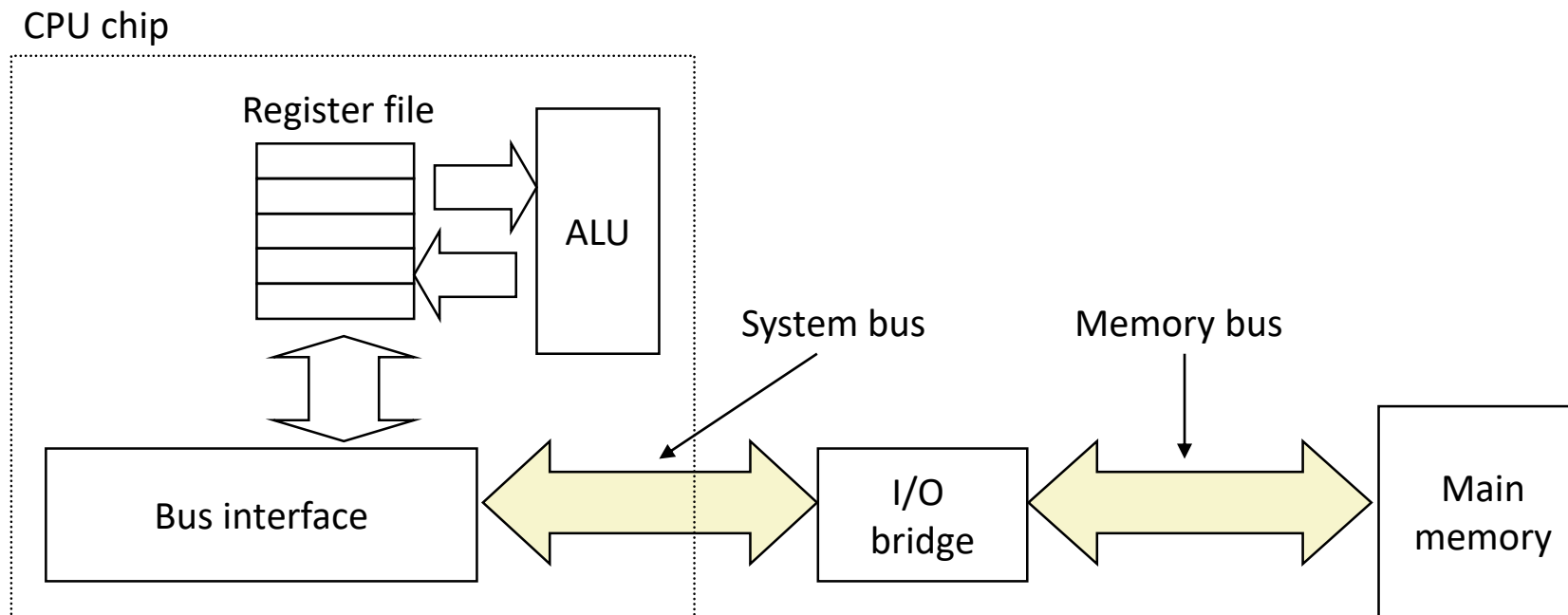
	Transistors per bit	Access Time	Refreshing Required?	Error Correction Required?	Cost	Application
SRAM	4 or 6	1	No	Not typically	100	Cache
DRAM	1	10	Yes	Yes	1	Main memory

Non-volatile Memory

- DRAM and SRAM are volatile memories, which means they lose information if powered off.
- Nonvolatile memories retain value even if powered is turned off.
 - **Read-only memory (ROM)**: programmed during production
 - **Programmable ROM (PROM)**: can be programmed once
 - **Erasable PROM (EPROM)**: can be bulk erased (UV, X-Ray)
 - **Electrically erasable PROM (EEPROM)**: electronic erase capability
 - **Flash memory**: EEPROMs with partial (block-level) erase capability
 - Wears out after about 100,000 erases.
- Uses for Nonvolatile Memories
 - Firmware programs stored in a ROM (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems, etc.)
 - Solid state disks (replace rotating disks in thumb drives, smart phones, mp3 players, tablets, laptops, etc.)
 - Disk caches

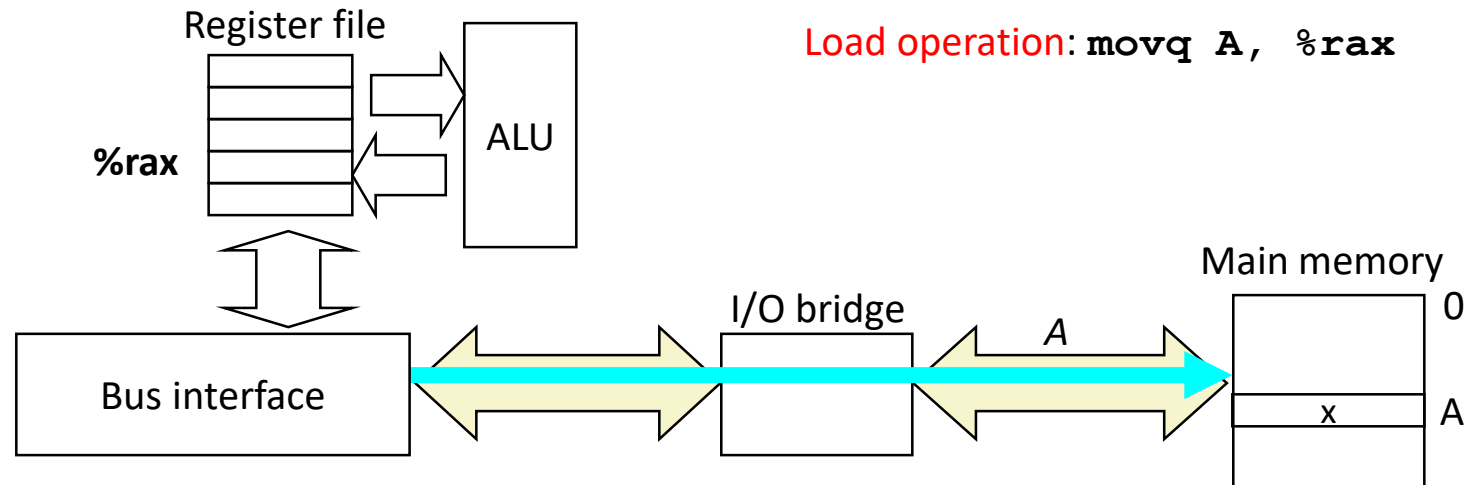
Bus Structure Connecting CPU and Memory

- A bus is a collection of parallel wires that carry address, data, and control signals.
- Buses are typically shared by multiple devices.



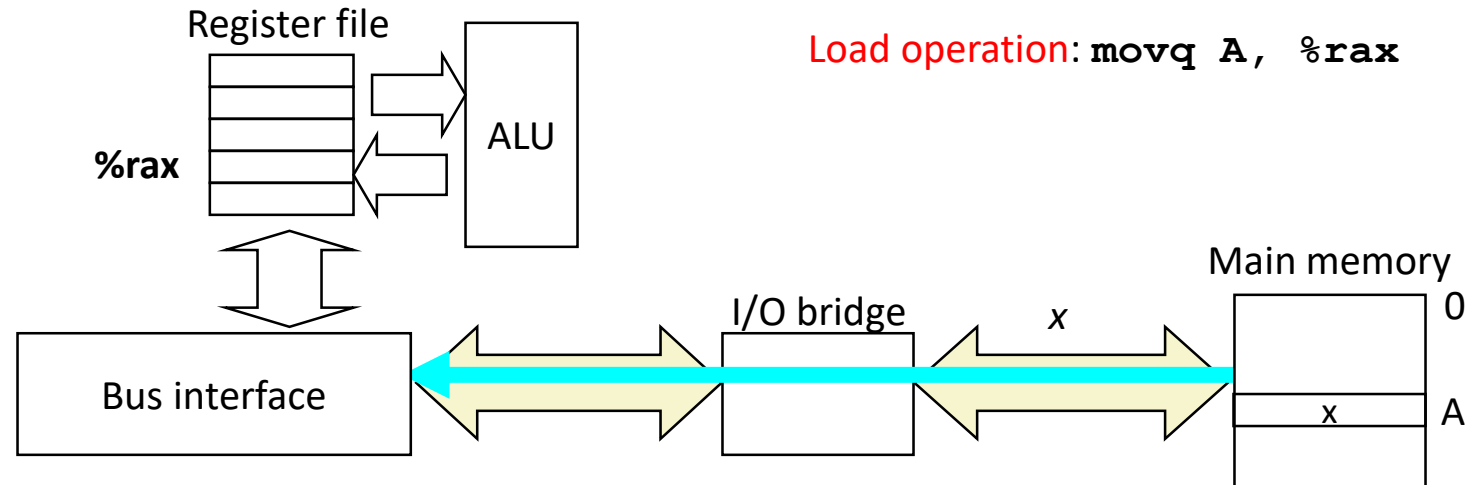
Memory Read Transaction (1)

- CPU places address A on the memory bus.



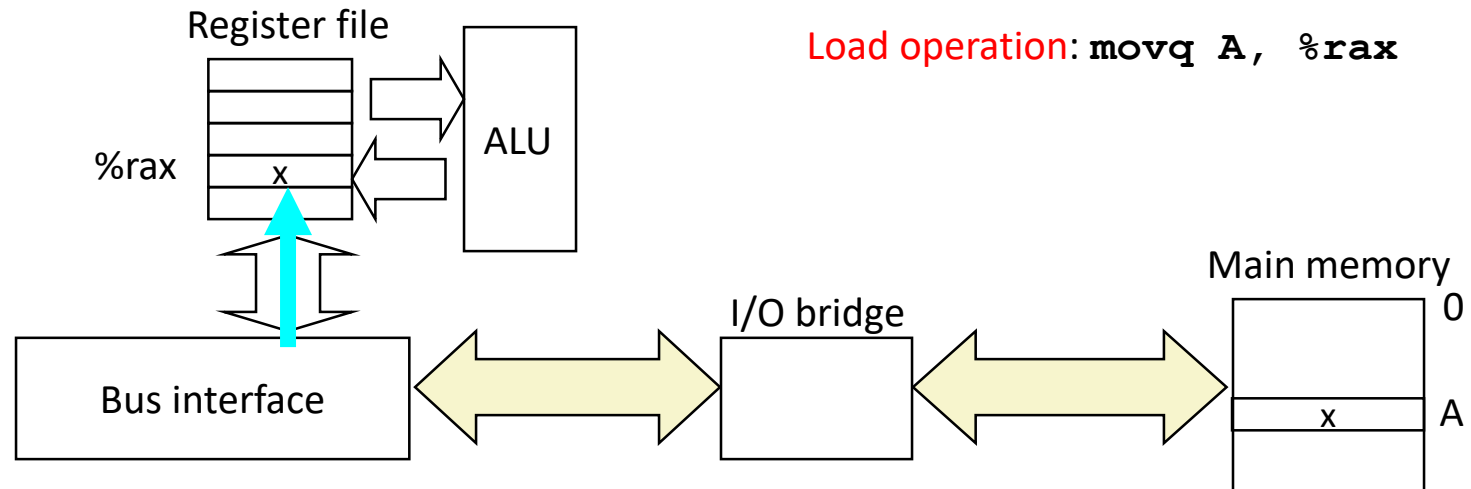
Memory Read Transaction (2)

- Main memory reads A from the memory bus, retrieves word x, and places it on the bus.



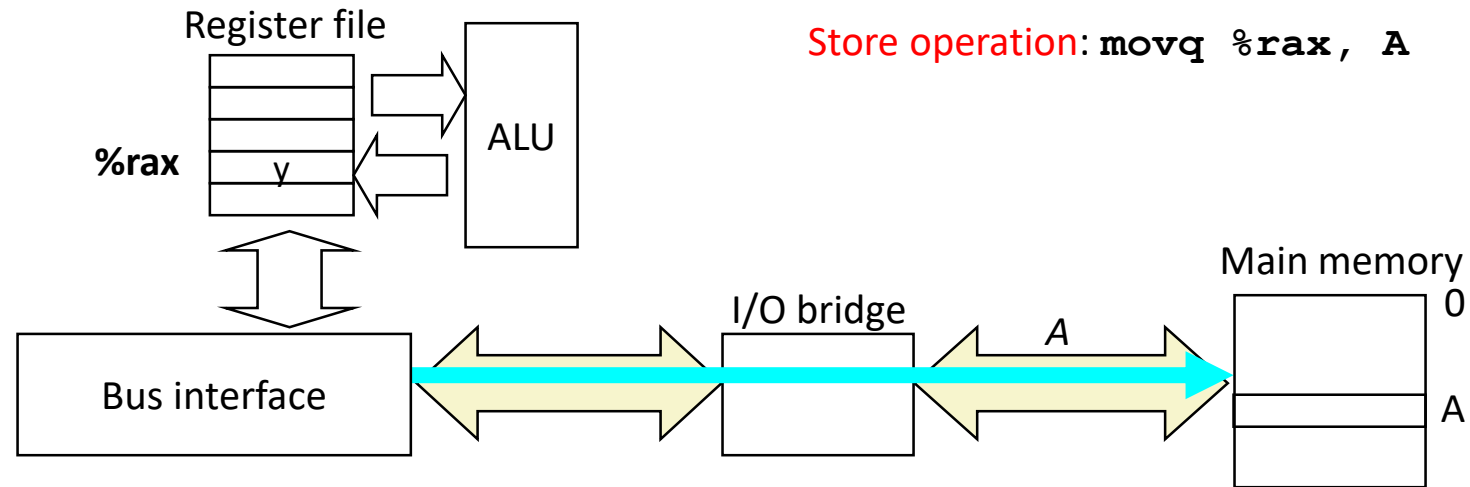
Memory Read Transaction (3)

- CPU read word x from the bus and copies it into register %rax.



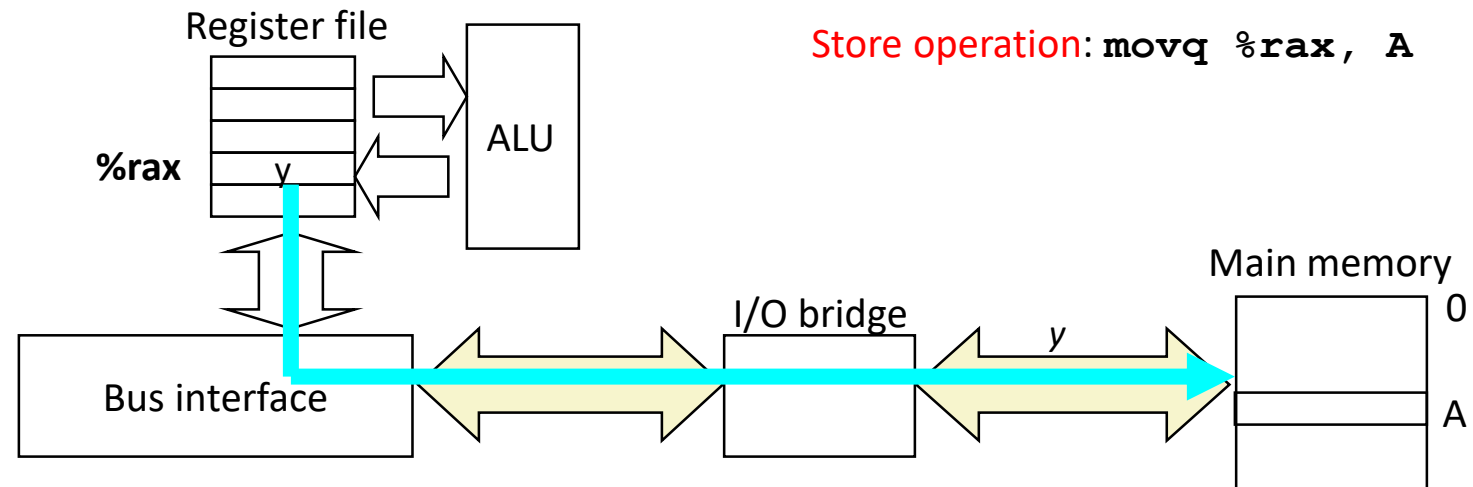
Memory Write Transaction (1)

- CPU places address A on bus. Main memory reads it and waits for the corresponding data word to arrive.



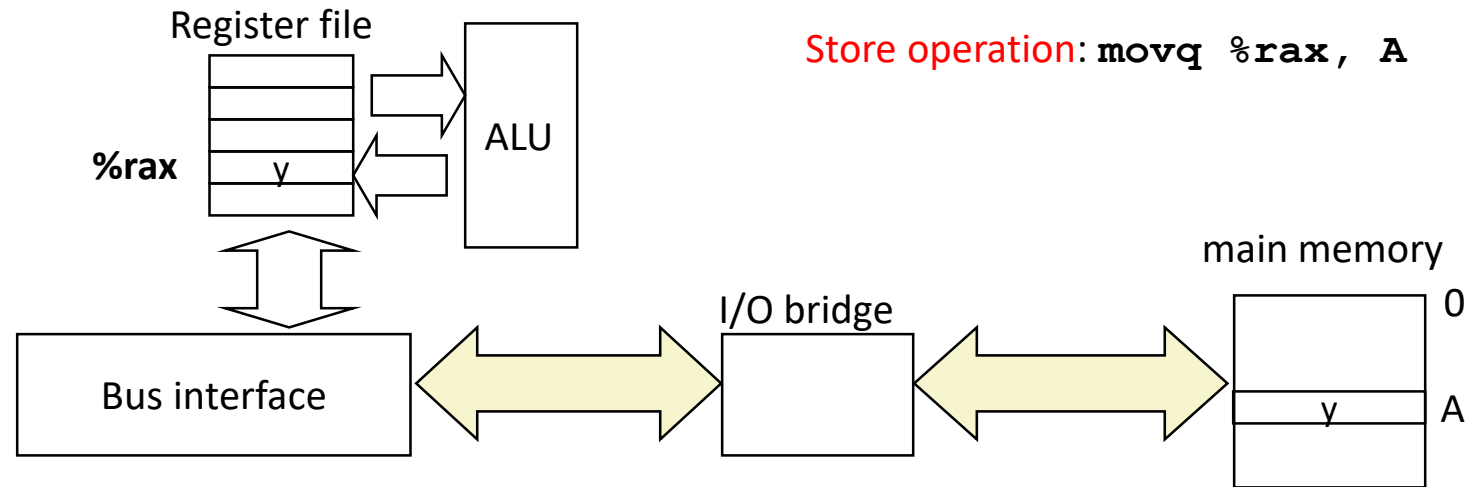
Memory Write Transaction (2)

- CPU places data word y on the bus.



Memory Write Transaction (3)

- Main memory reads data word y from the bus and stores it at address A .

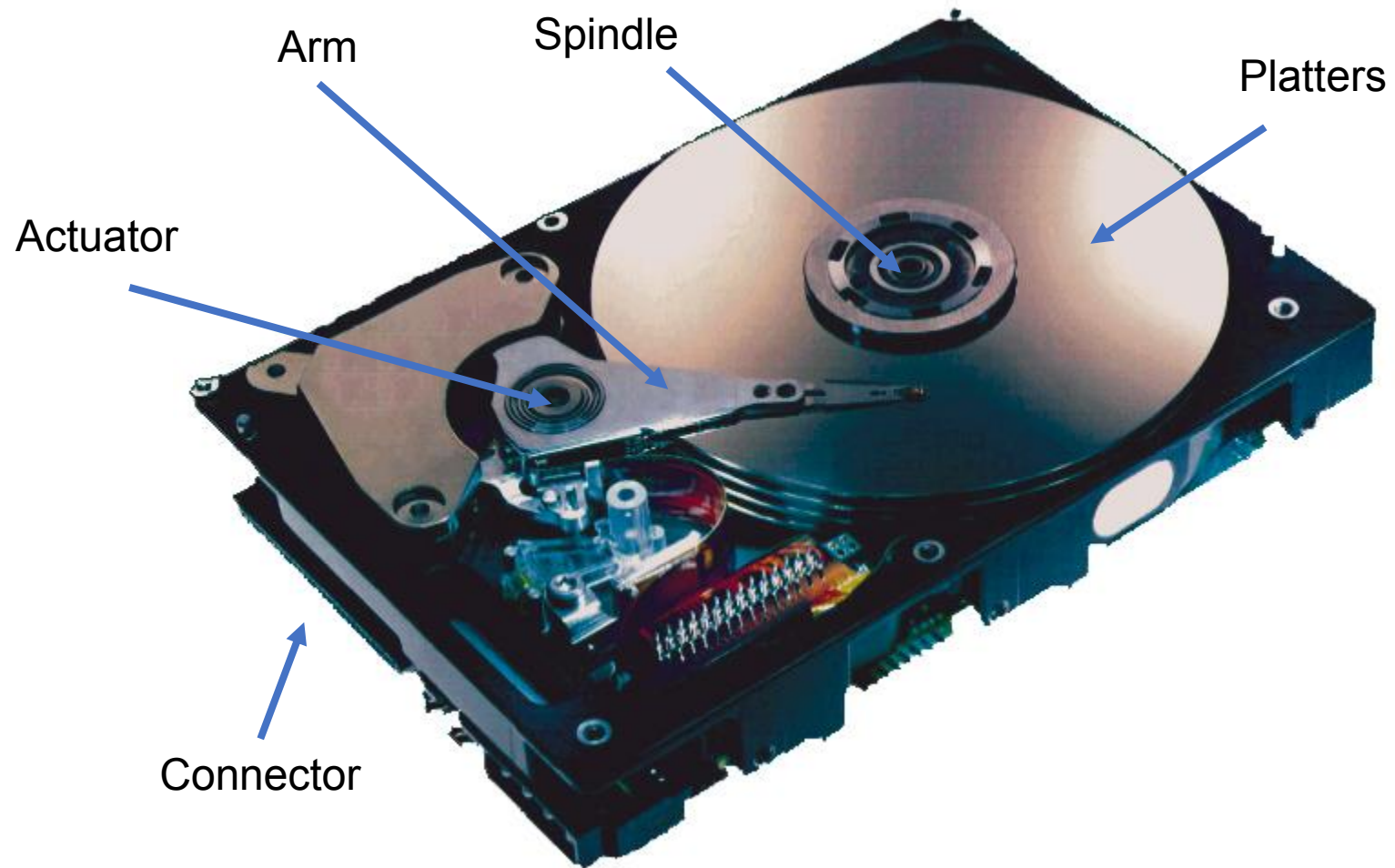


Disk Storage

Disks

- **Disks** are workhorse storage devices that hold enormous amounts of data, on the order of hundreds to thousands of gigabytes, as opposed to the hundreds or thousands of megabytes in a RAM-based memory.
- However, it takes on the order of milliseconds to read information from a disk, a hundred thousand times longer than from DRAM and a million times longer than from SRAM.

What Is inside A Disk Drive?



Disk Geometry

- Disks consist of platters...

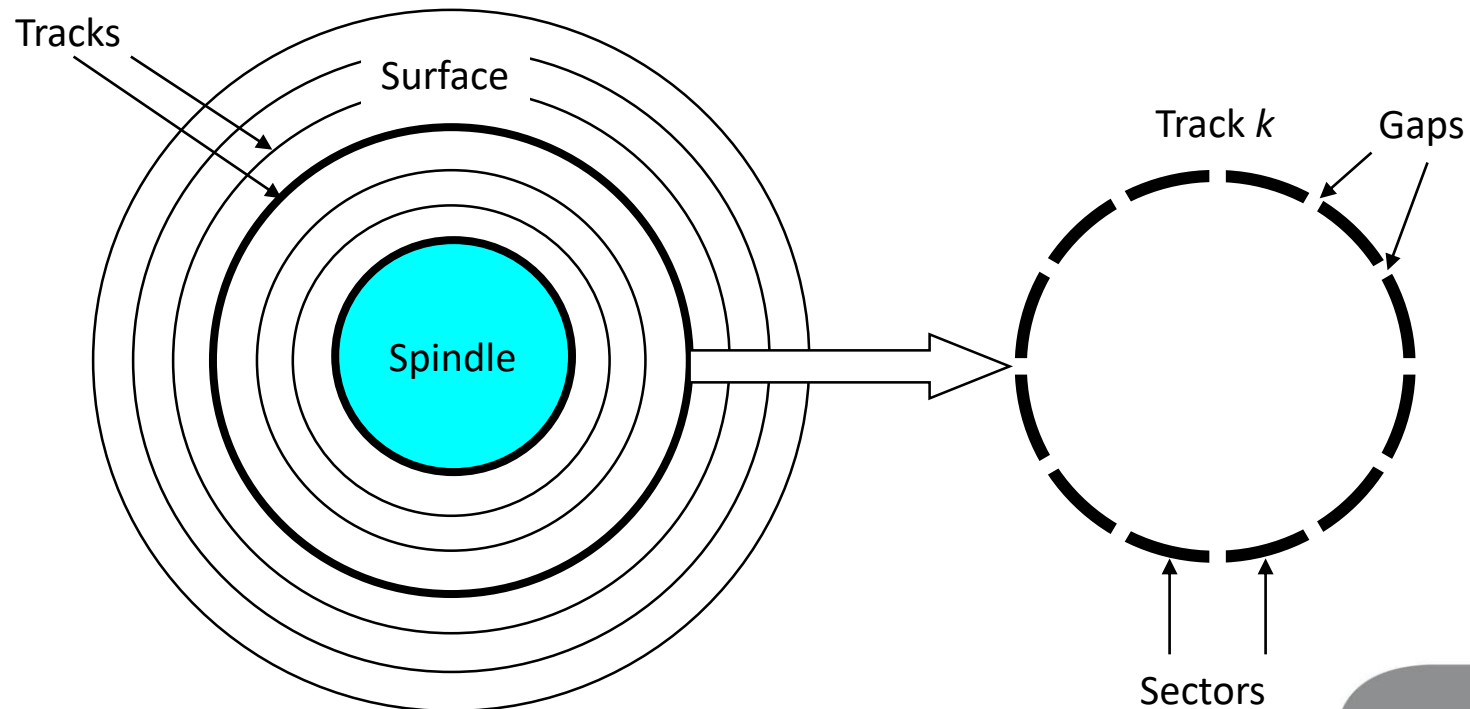
Disk Geometry

- Disks consist of platters...



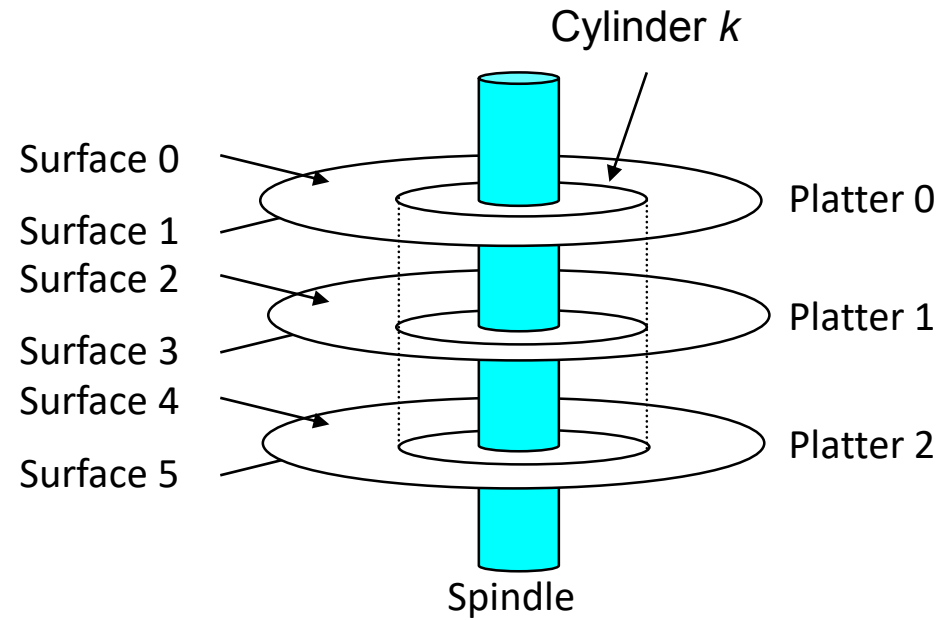
Disk Geometry

- Disks consist of platters, each with two surfaces.
- Each surface consists of concentric rings called tracks.
- Each track consists of sectors separated by gaps.



Disk Geometry

- Aligned tracks form a cylinder.

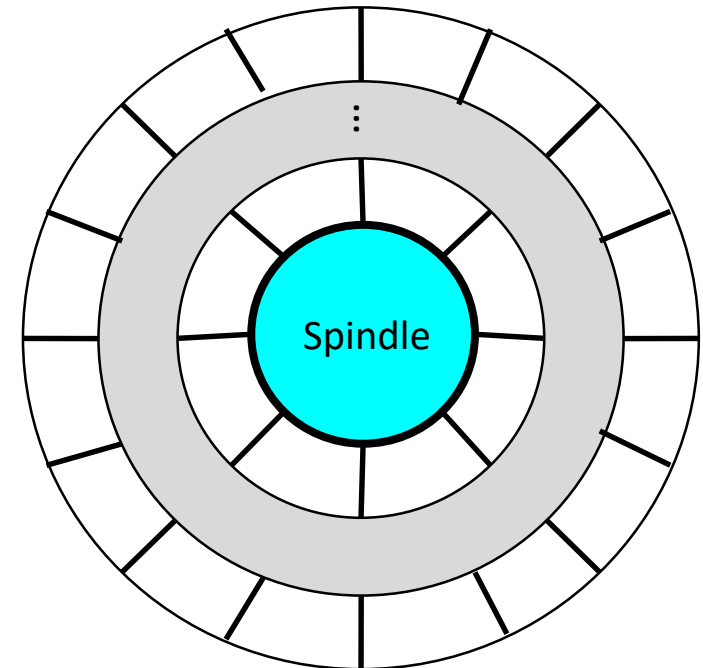


Disk Capacity

- **Capacity:** maximum number of bits that can be stored.
 - Vendors express capacity in units of Giga Bytes (GB), where $1 \text{ GB} \approx 10^9 \text{ Bytes}$.
- Capacity is determined by these factors:
 - **Recording density (bits/in):** number of bits that can be stored into a 1 inch segment of a track.
 - **Track density (tracks/in):** number of tracks that can be stored into a 1 inch radial segment.
 - **Areal density (bits/in²):** multiplication product of recording and track density.

Recording Zones

- Modern disks partition tracks into disjoint subsets called **recording zones**.
- Each track in a zone has the same number of sectors, determined by the circumference of innermost track.
- Each zone has a different number of sectors/track, outer zones have more sectors/track than inner zones.
- So we use **the average number of sectors/track** when computing capacity.



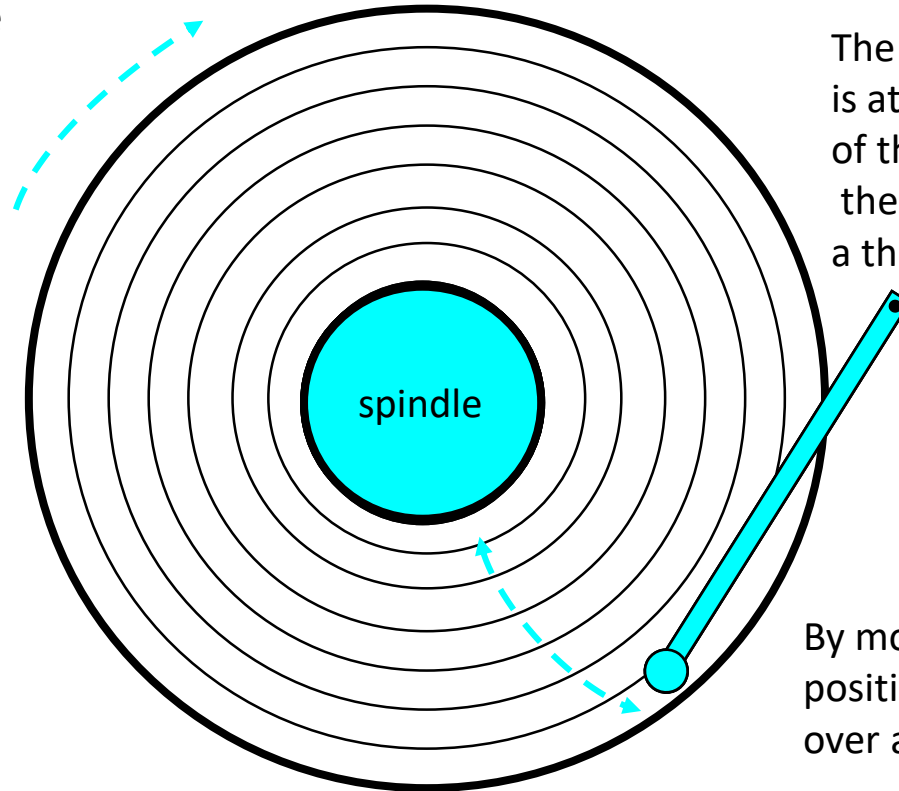
Calculating Disk Capacity

- **Capacity =** (# bytes/sector) x (avg. # sectors/track) x (# tracks/surface) x (# surfaces/platter) x (# platters/disk)
- Example:
 - 512 bytes/sector
 - 300 sectors/track (on average)
 - 20,000 tracks/surface
 - 2 surfaces/platter
 - 5 platters/disk

$$\begin{aligned}\text{Capacity} &= 512 \times 300 \times 20000 \times 2 \times 5 \\ &= 30,720,000,000 \\ &= 30.72 \text{ GB}\end{aligned}$$

Disk Operation

The disk surface spins at a fixed rotational rate

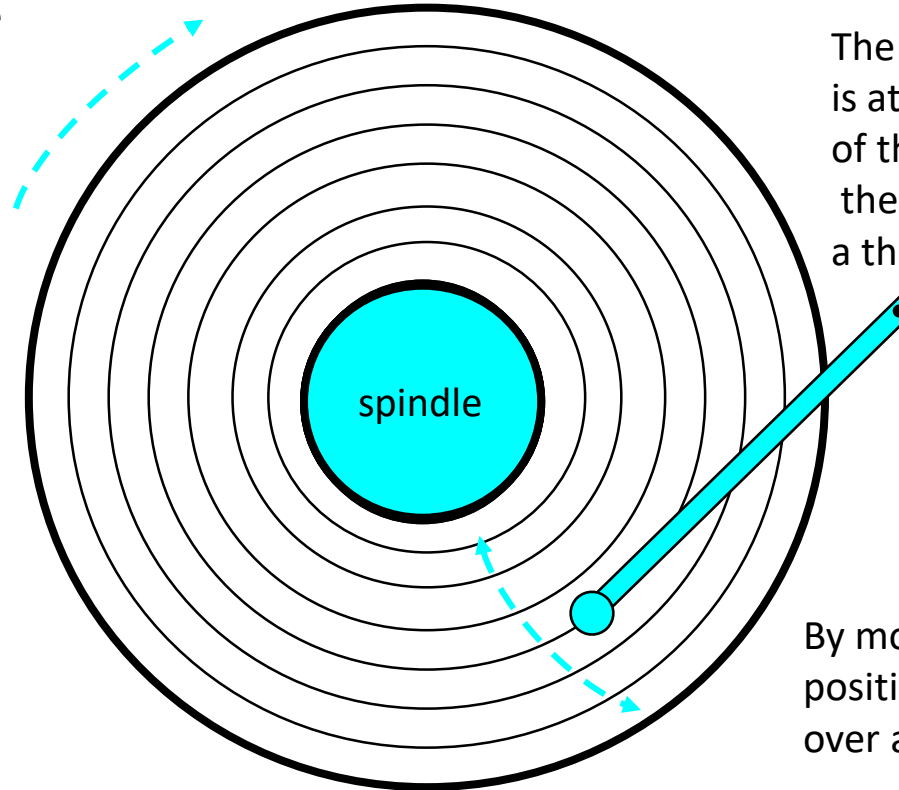


The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

Disk Operation

The disk surface spins at a fixed rotational rate

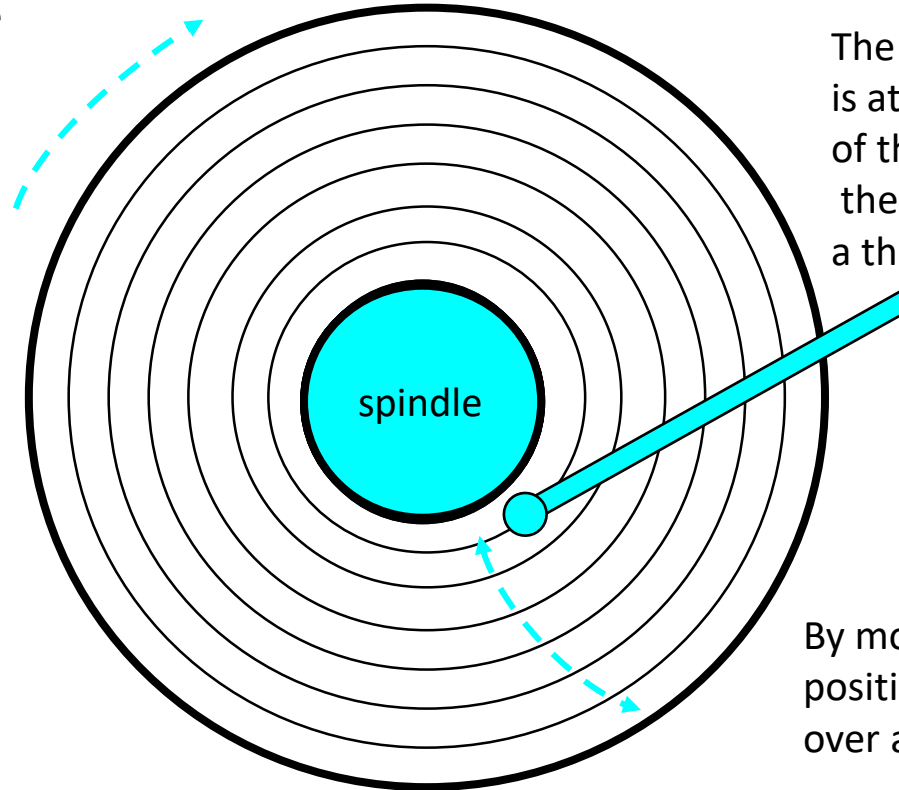


The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

Disk Operation

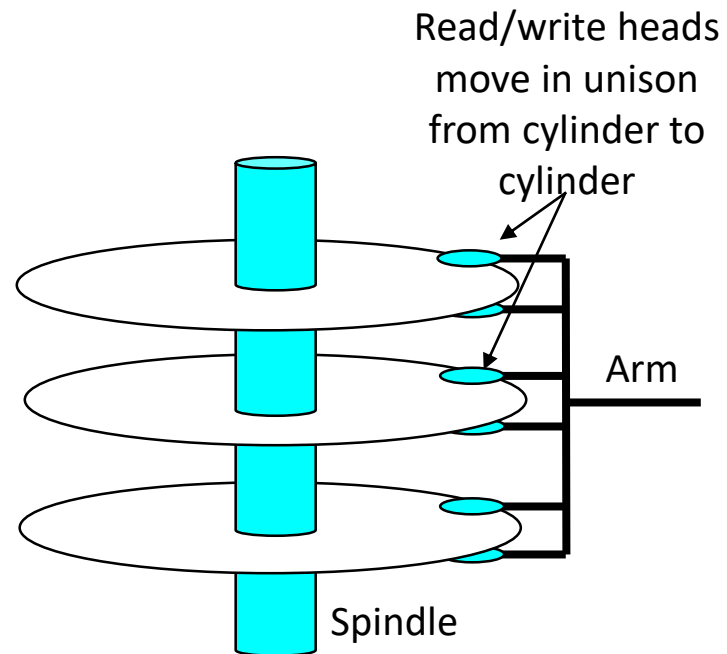
The disk surface spins at a fixed rotational rate



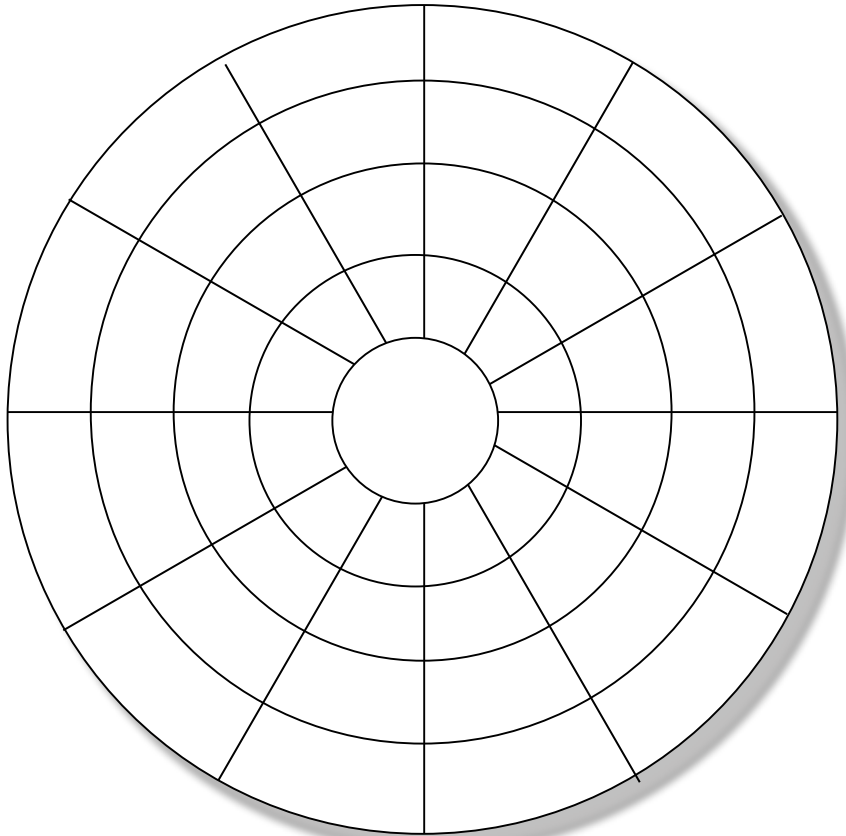
The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

Disk Operation



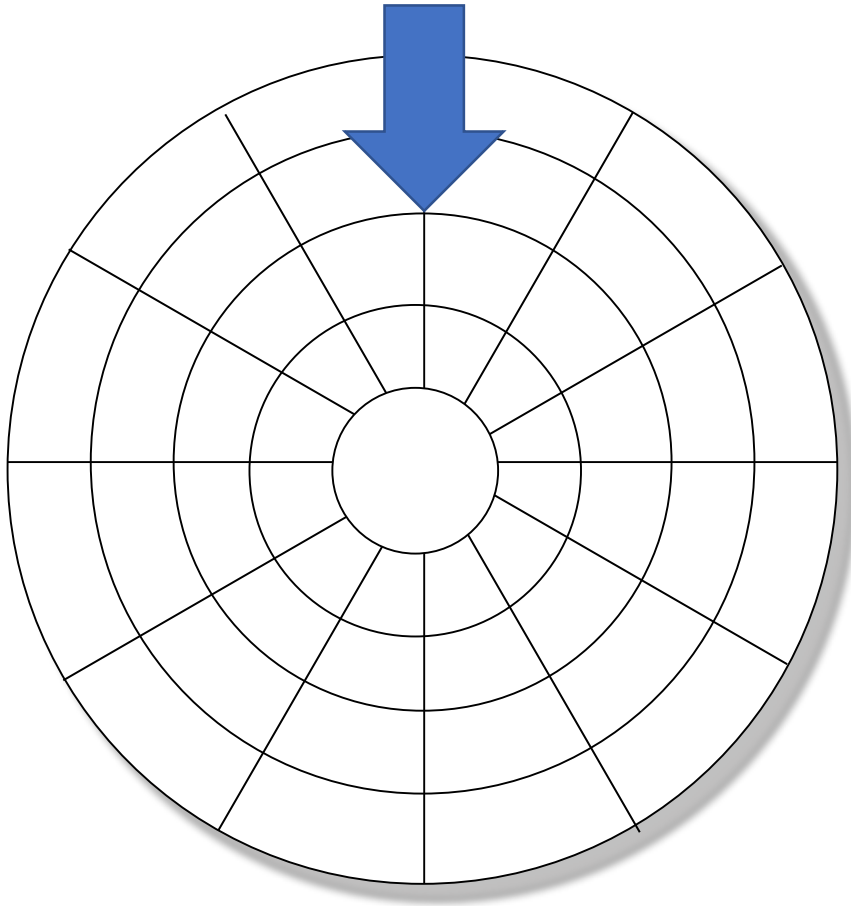
Disk Structure



Surface organized into tracks

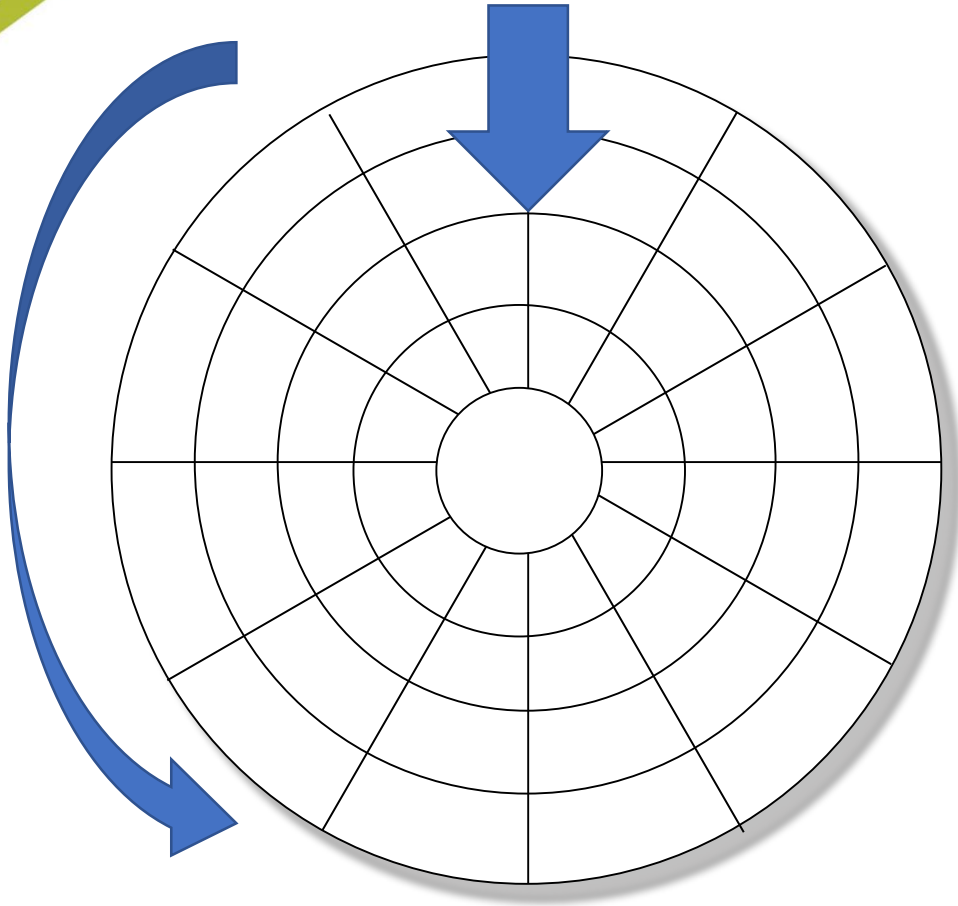
Tracks divided into sectors

Disk Access



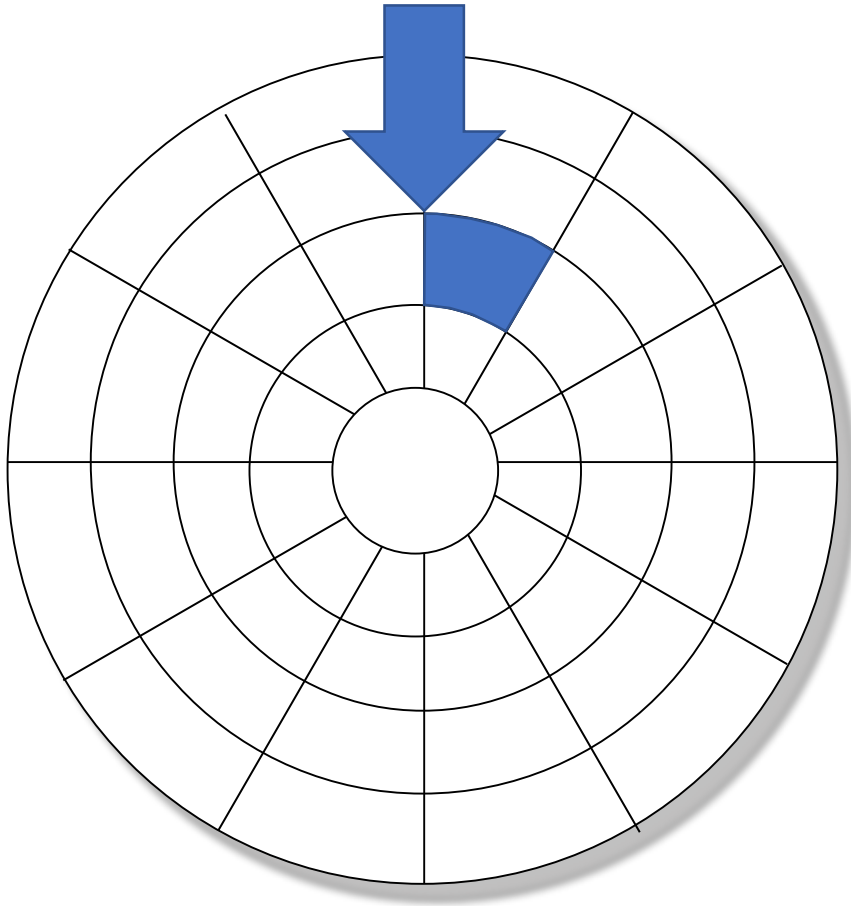
Head in position above track

Disk Access



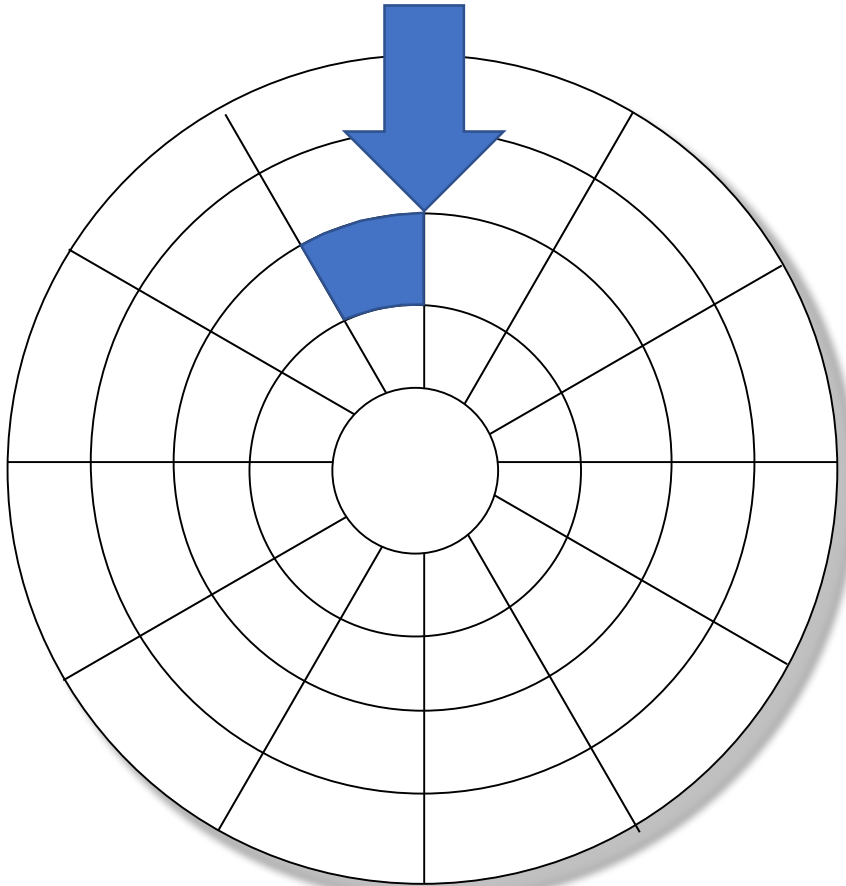
Counter-clockwise rotation

Disk Access – Read



About to read blue sector

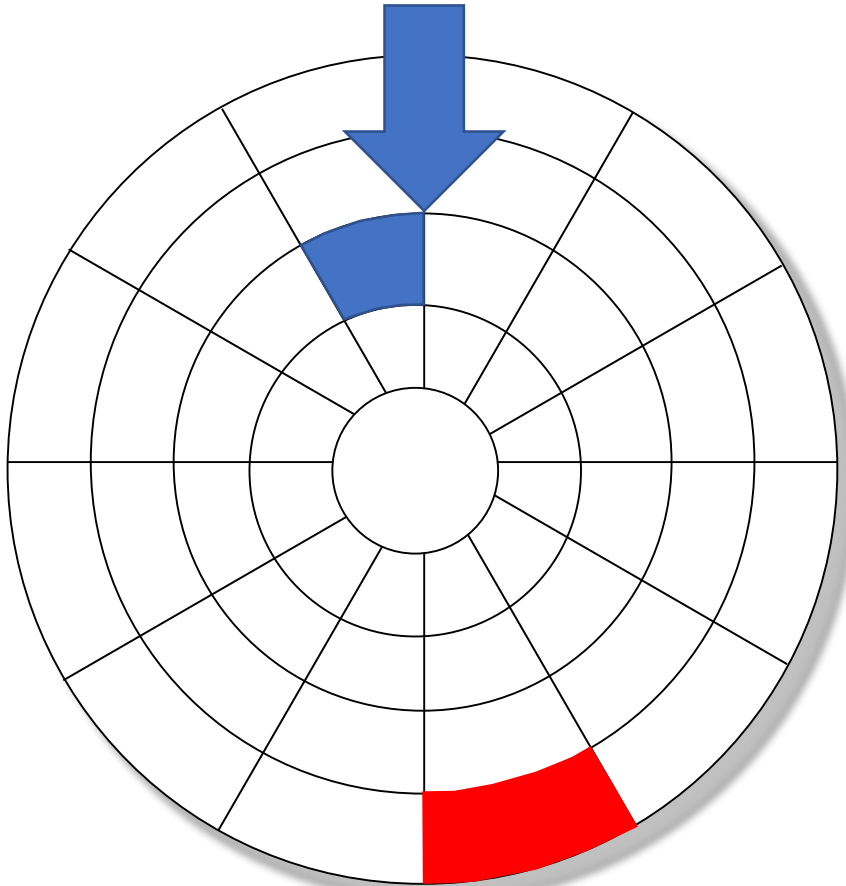
Disk Access – Read



After reading blue sector

After **BLUE** read

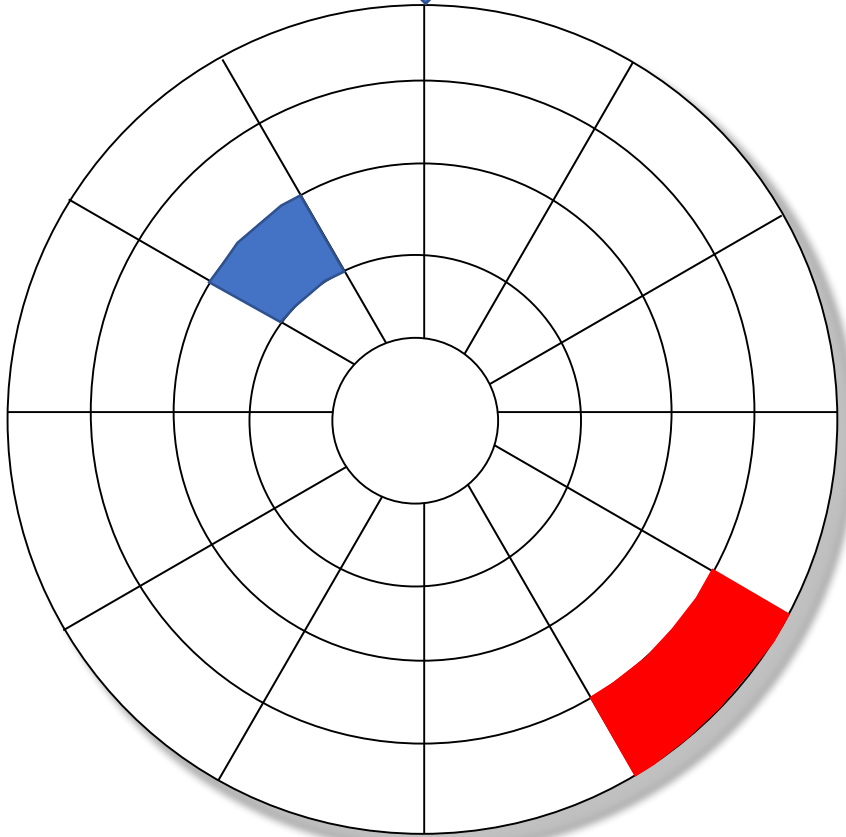
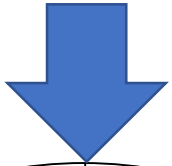
Disk Access – Read



After BLUE read

Red read request scheduled next

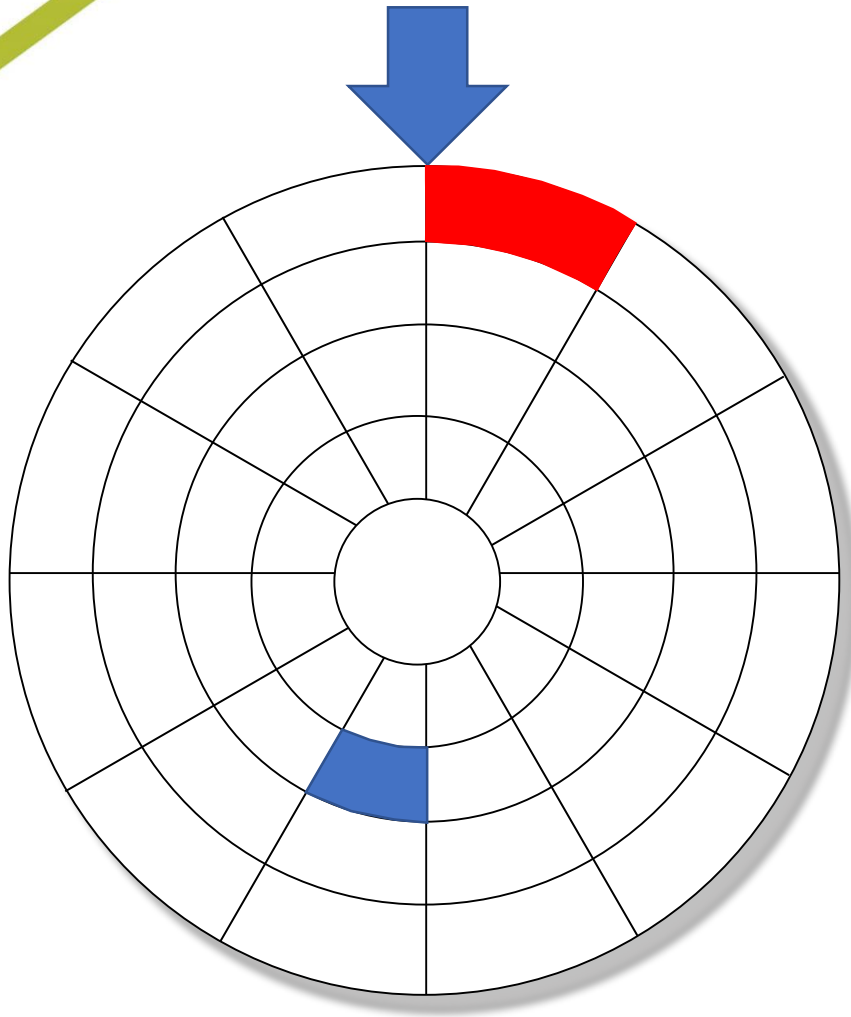
Disk Access – Seek



Seek to Red's Track

After **BLUE** read

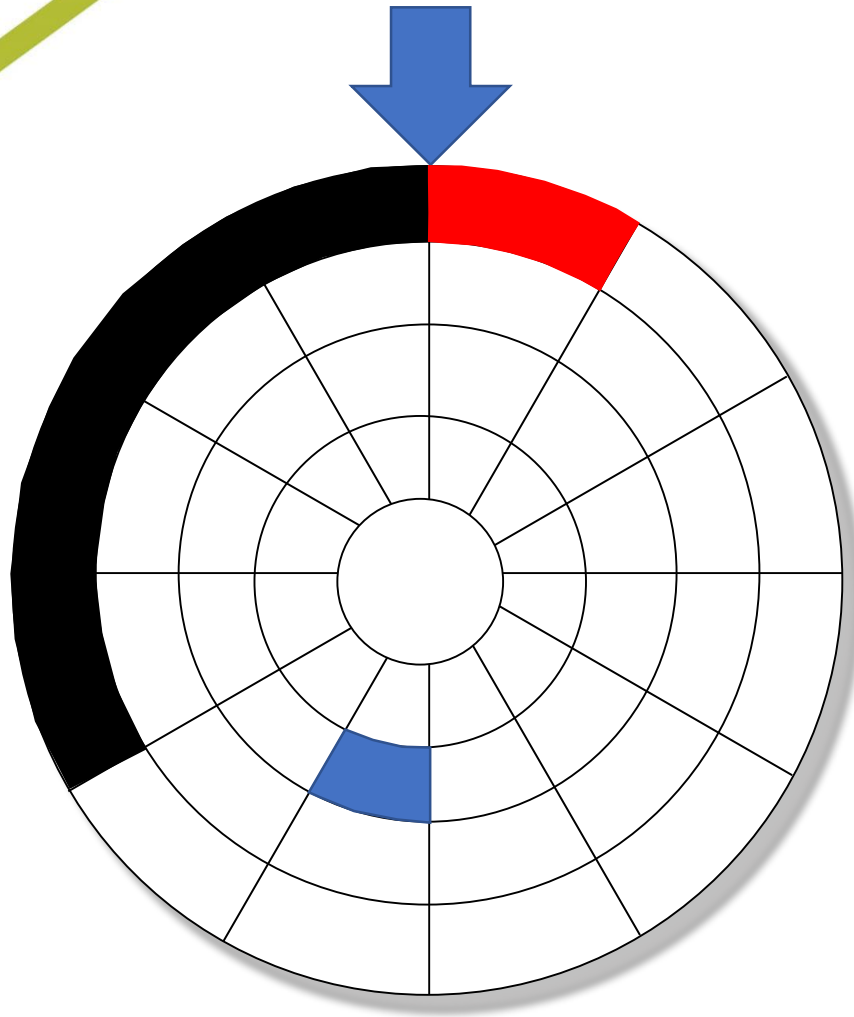
Disk Structure



Wait for sector to rotate

After BLUE read

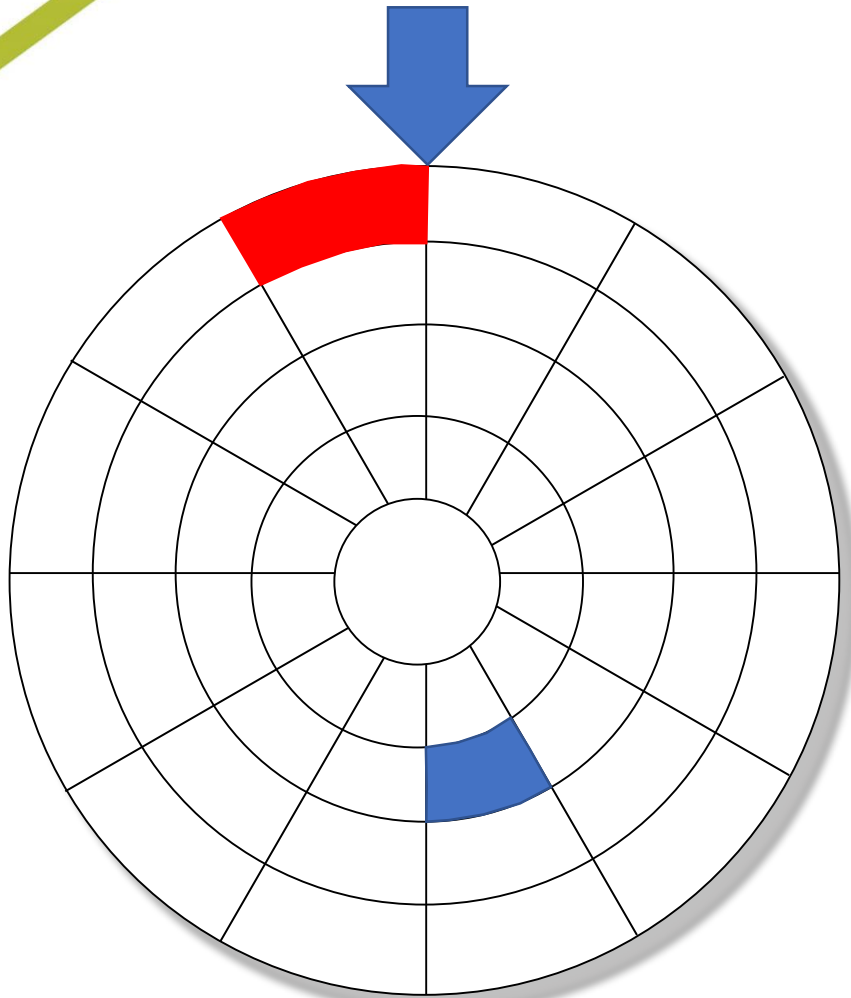
Disk Access – Rotation Latency



Rotational Latency

After **BLUE** read

Disk Access – Read



Complete read of red

After RED read

Disk Access Time

- Average time to access some target sector approximated by:

$$T_{access} = T_{avg\ seek} + T_{avg\ rotation} + T_{avg\ transfer}$$

- **Seek time ($T_{avg\ seek}$)**

- Time to position heads over cylinder containing target sector.
- Typical $T_{avg\ seek}$ is 3-9 ms

- **Rotational latency ($T_{avg\ rotation}$)**

- Time waiting for first bit of target sector to pass under r/w head.

$$T_{avg\ rotation} = 0.5 \times \frac{1}{RPM} \times \frac{60\ sec}{1\ min}$$

-

Typical rotation = 7200 RPMs

- **Transfer time ($T_{avg\ transfer}$)**

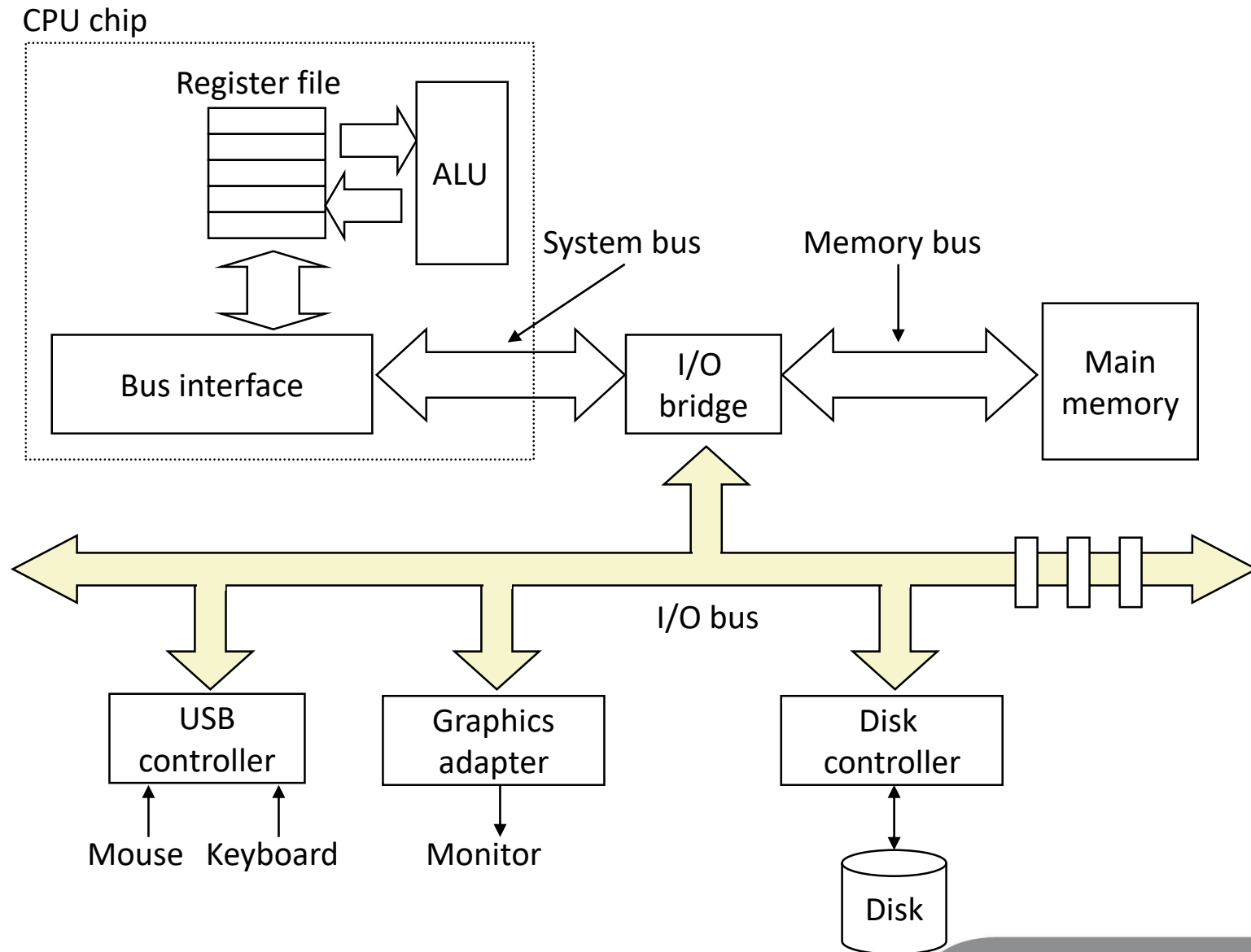
- Time to read the bits in the target sector.

$$T_{avg\ transfer} = \frac{1}{RPM} \times \frac{1}{\text{average \# of sectors per track}} \times \frac{60\ sec}{1\ min}$$

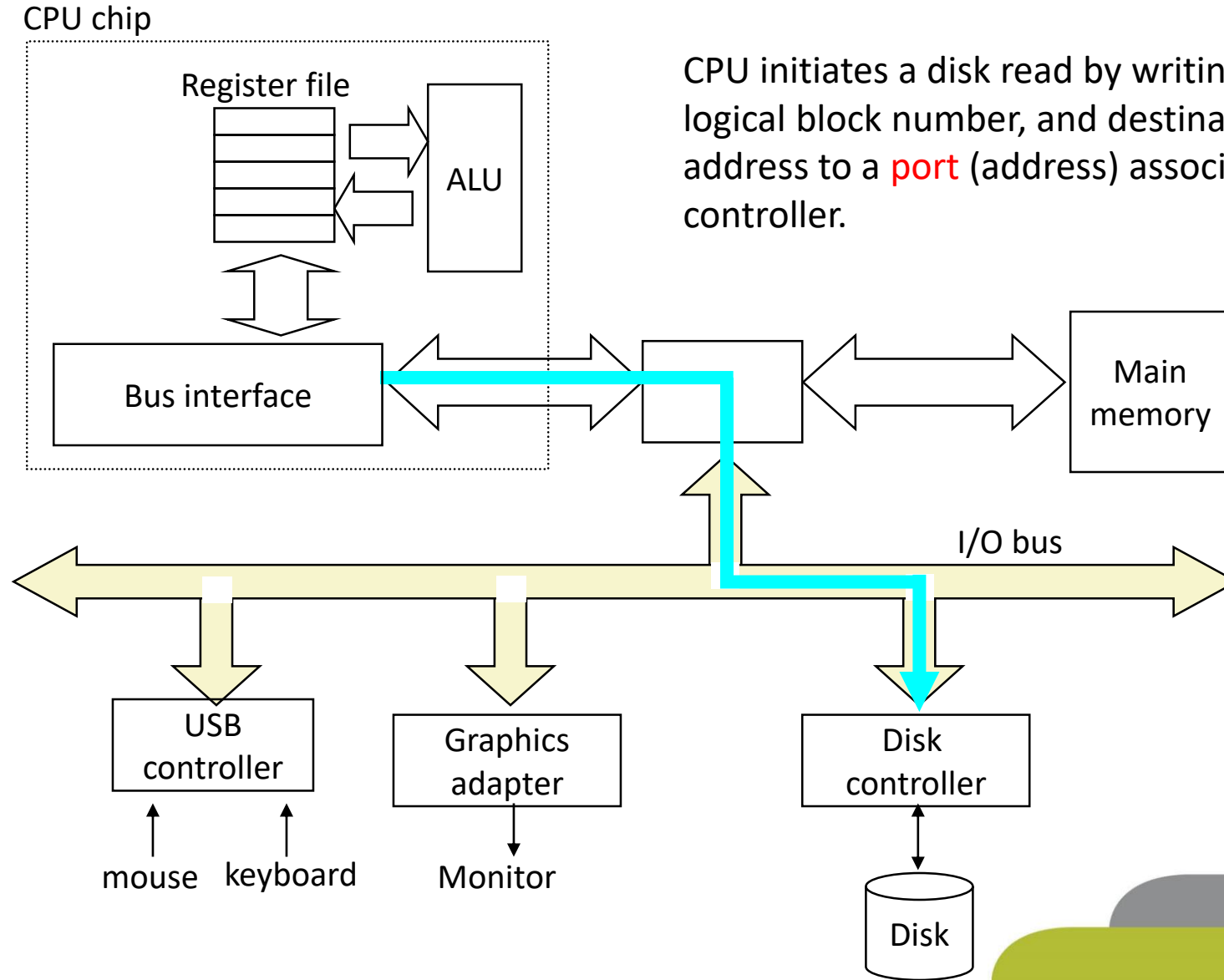
Disk Access Time - Example

- Given:
 - Rotational rate = 7,200 RPM
 - Average seek time = 9 ms.
 - Avg # sectors/track = 400.
- Derived:
 - $T_{\text{avg rotation}} = 1/2 \times (60 \text{ secs}/7200 \text{ RPM}) \times 1000 \text{ ms/sec} = 4 \text{ ms}.$
 - $T_{\text{avg transfer}} = 60/7200 \text{ RPM} \times 1/400 \text{ secs/track} \times 1000 \text{ ms/sec} = 0.02 \text{ ms}$
 - $T_{\text{access}} = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms}$
- Important points:
 - **Access time dominated by seek time and rotational latency.**
 - First bit in a sector is the most expensive, the rest are essentially free.
 - SRAM access time is about 4 ns/double-word, DRAM about 60 ns/double-word
 - **Disk is about 40,000 times slower than SRAM, and 2,500 times slower than DRAM.**

I/O Bus

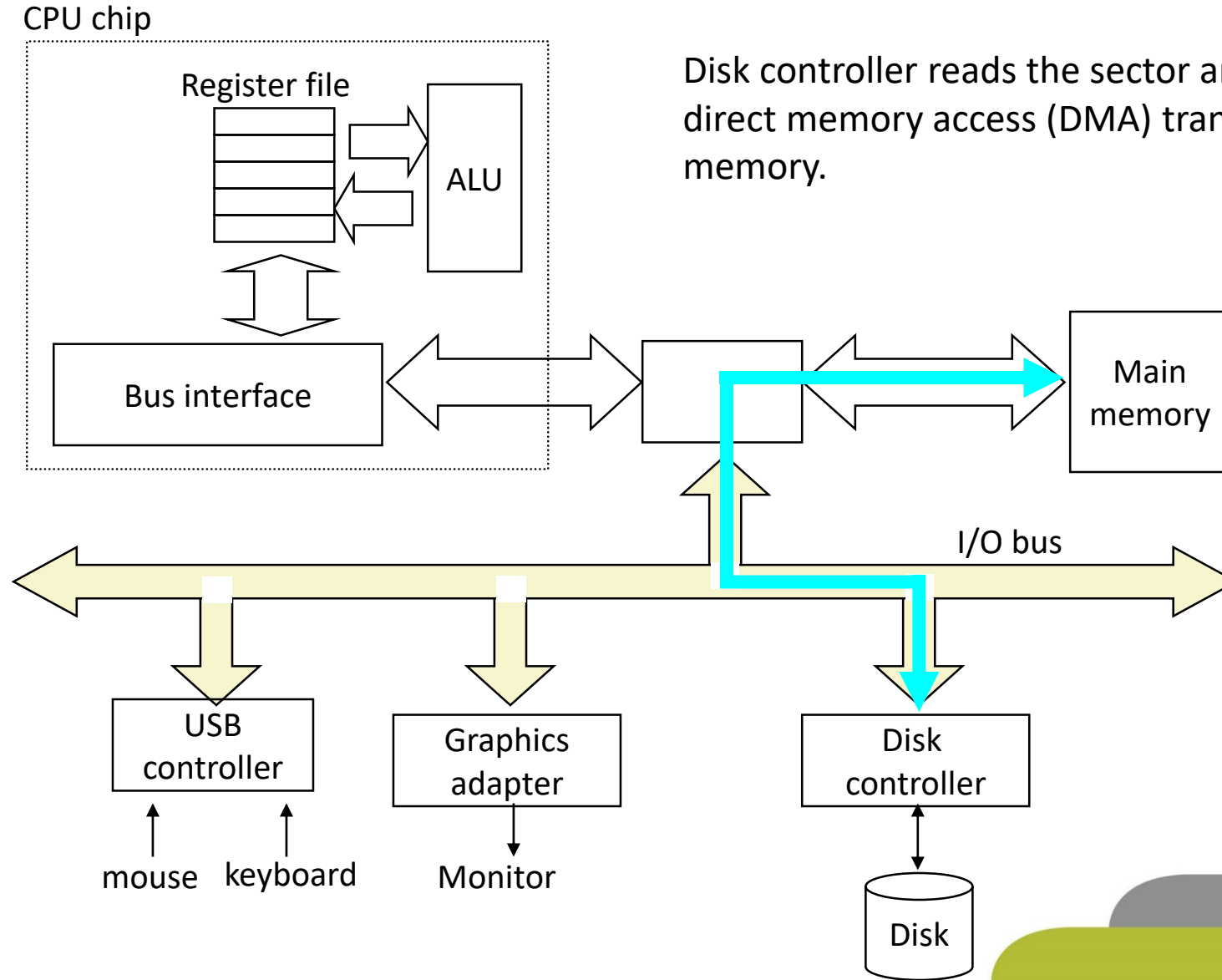


Reading a Disk Sector (1)

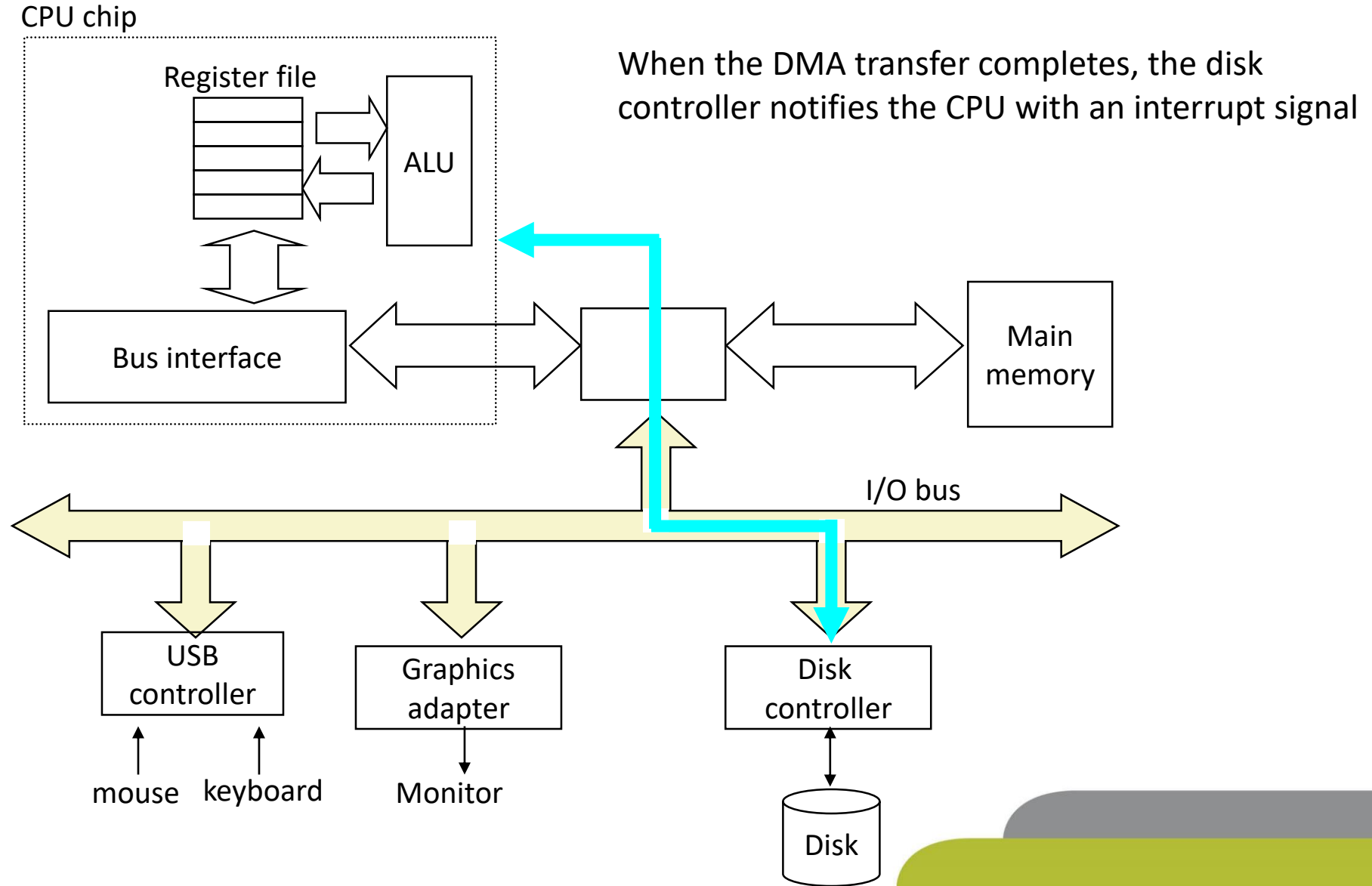


CPU initiates a disk read by writing a command, logical block number, and destination memory address to a **port** (address) associated with disk controller.

Reading a Disk Sector (2)



Reading a Disk Sector (3)





DOUGLAS COLLEGE

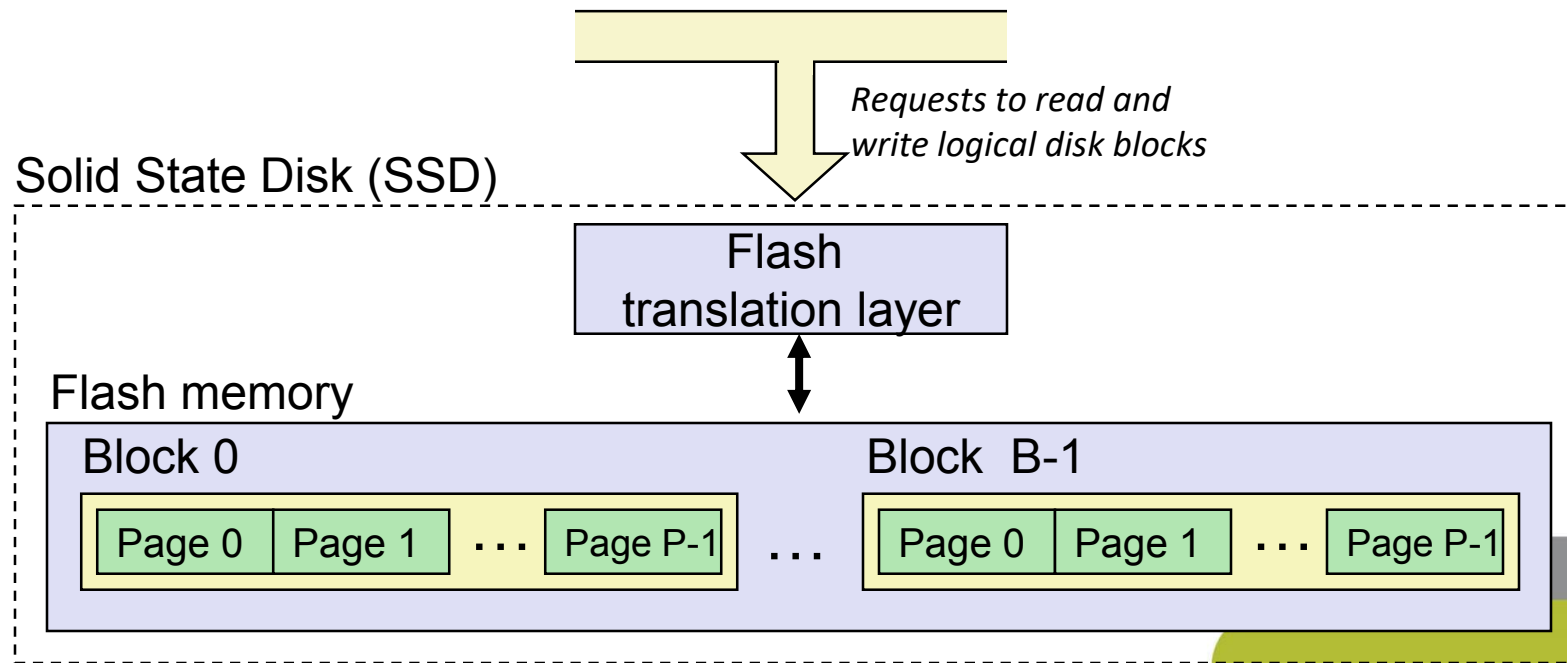
Solid State Disks

Solid State Disks (SSDs)

- A solid state disk (SSD) is a storage technology, based on flash memory that in some situations is an attractive alternative to the conventional rotating disk.
- An SSD package plugs into a standard disk slot on the bus (typically USB or SATA) and behaves like any other disk, processing requests from the CPU to read and write logical disk blocks.
- An SSD package consists of one or more flash memory chips, which replace the mechanical drive in a conventional rotating disk, and a **flash translation layer**, which is a hardware/firmware device that plays the same role as a disk controller, translating requests for logical blocks into accesses of the underlying physical device.

Solid State Disks (SSDs)

- A flash memory consists of a sequence of B blocks, where each block consists of P pages.
- Typically, pages are 512 bytes to 4 KB in size, and a block consists of 32- 128 pages, with total block sizes ranging from 16KB to 512 KB. Data are read and written in units of **pages**.
- **A Page can be written only after its block has been erased.**
- A block wears out after about 100,000 repeated writes.

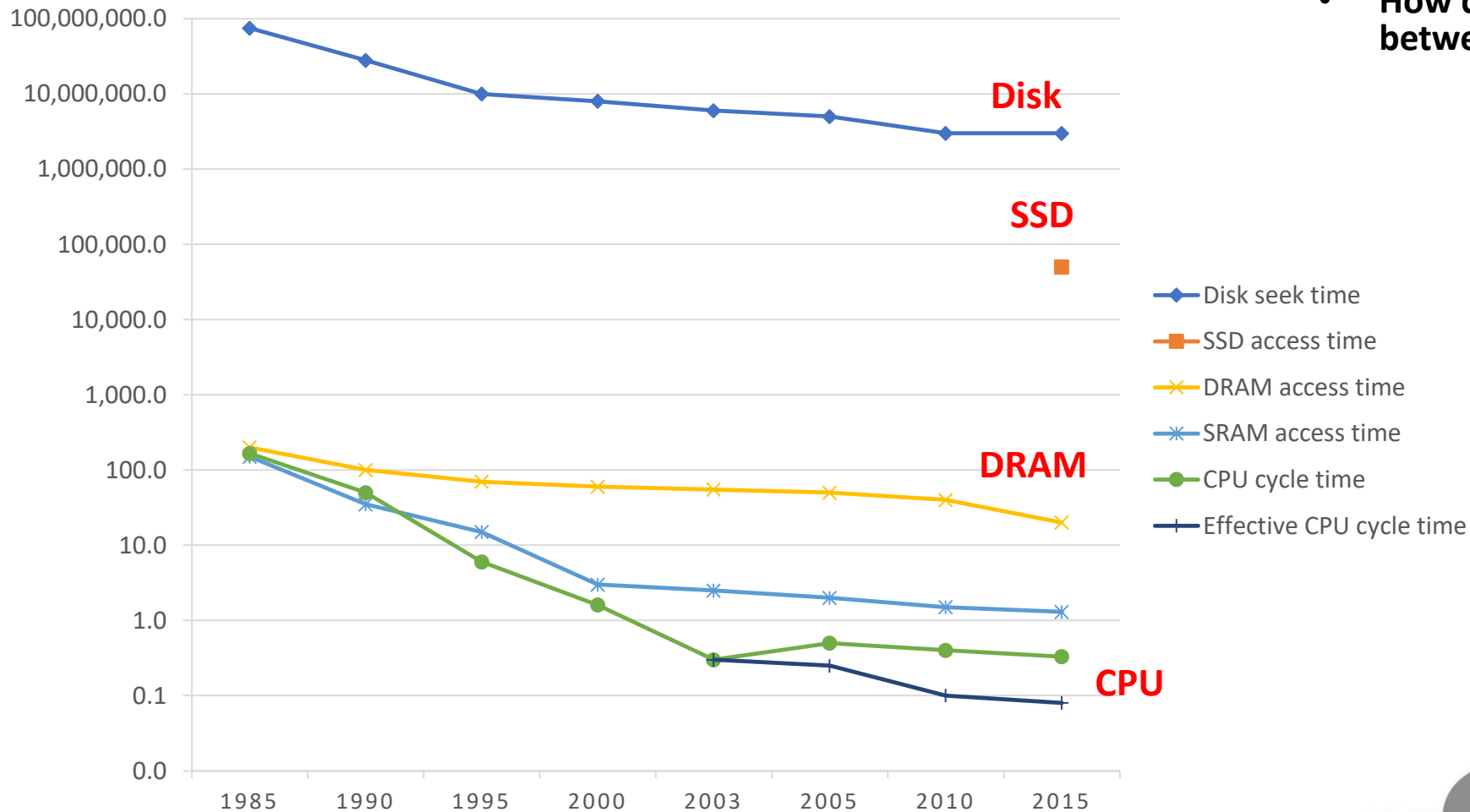


SSD vs. Rotating Disk

- **Advantages**
 - No moving parts → faster, less power, more rugged
- **Disadvantages**
 - Have the potential to wear out
 - Mitigated by “wear leveling logic” in flash translation layer
 - E.g. Intel SSD 730 guarantees 128 petabyte (128×10^{15} bytes) of writes before they wear out
 - In 2015, about 30 times more expensive per byte
- **Applications**
 - MP3 players, smart phones, laptops
 - Beginning to appear in desktops and servers

The CPU-Memory Gap

- The gap widens between DRAM, disk, and CPU speeds.
- How do we bridge the gap between memory and the CPU?

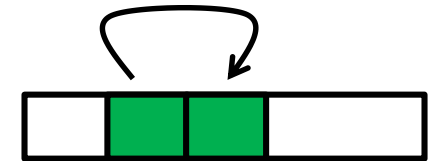
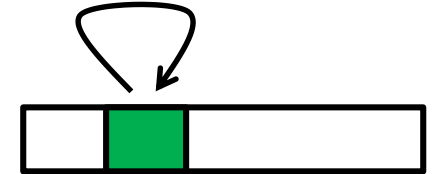


Locality of Reference

(Principle of Locality)

Locality

- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently.
- **Temporal locality:**
 - Recently referenced items are likely to be referenced again in the near future.
- **Spatial locality:**
 - Items with nearby addresses tend to be referenced close together in time.



Locality

- Locality reflects the observation that during the course of execution of a program, memory references by the processor tend to cluster.
- Locality is based on three assertions:
 - During any interval of time, a program references memory location non-uniformly.
 - As a function of time, the probability that a given unit of memory is referenced tends to change slowly.
 - The correlation between immediate past and immediate future memory reference patterns is high and tapers off as the time interval increases.

Spatial vs. Temporal Locality

- **Temporal locality:**
 - Refers to the tendency of a program to reference in the near future those units of memory referenced in the recent past.
 - Constants, temporary variables, are constructs that lead to this principle.
- **Spatial locality:**
 - Refers to the tendency of a program to reference units of memory whose addresses are near one another.
 - Also reflects the tendency of a program to access data locations sequentially, such as when processing a table of data.

Spatial vs. Temporal Locality – Analogy



Locality

- Programmers should understand the principle of locality because, in general, programs with good locality run faster than programs with poor locality.
- All levels of modern computer systems are designed to exploit locality.
- At the hardware level, the principle of locality allows computer designers to speed up main memory accesses by introducing small fast memories known as **cache memories** that hold blocks of the most recently referenced instructions and data items.
- At the operating system level, the principle of locality allows the system to use the main memory as a cache of the most recently referenced chunks of the virtual address space.
- Similarly, the operating system uses main memory to cache the most recently used disk blocks in the disk file system.

Locality

- The principle of locality also plays a crucial role in the design of application programs.
- For example, Web browsers exploit temporal locality by caching recently referenced documents on a local disk.
- High-volume Web servers store recently requested documents in front-end disk caches that satisfy requests for these documents without requiring any intervention from the server.

Qualitative Locality Example

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```

- Data references
 - Reference array elements in succession → Spatial locality
 - Reference variable sum each iteration. → Temporal locality
- Instruction references
 - Reference instructions in sequence. → Spatial locality
 - Cycle through loop repeatedly. → Temporal locality

Qualitative Locality Example

- A function such as **sumvec()** that visits each element of a vector sequentially is said to have a **stride-1 reference pattern** (with respect to the element size).
- Visiting every kth element of a contiguous vector is called a **stride-k reference pattern**.
- Stride-1 reference patterns are a common and important source of spatial locality in programs.
- In general, **as the stride increases, the spatial locality decreases.**

Poll 1 (Extra Credit)

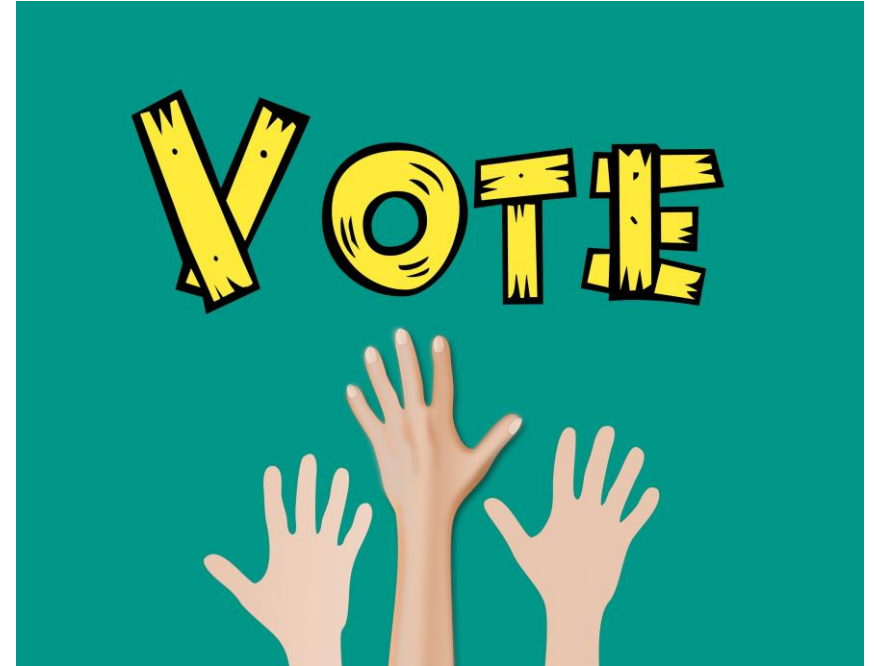
The following function enjoys good overall locality.

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

- a) True
- b) False

Please use the “Poll” window to participate for extra credit!



Qualitative Locality Example (M=2, N=3)

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access Order	1	3	5	2	4	6

Qualitative Locality Example (M=2, N=3)

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access Order	1	2	3	4	5	6

Summary of Locality

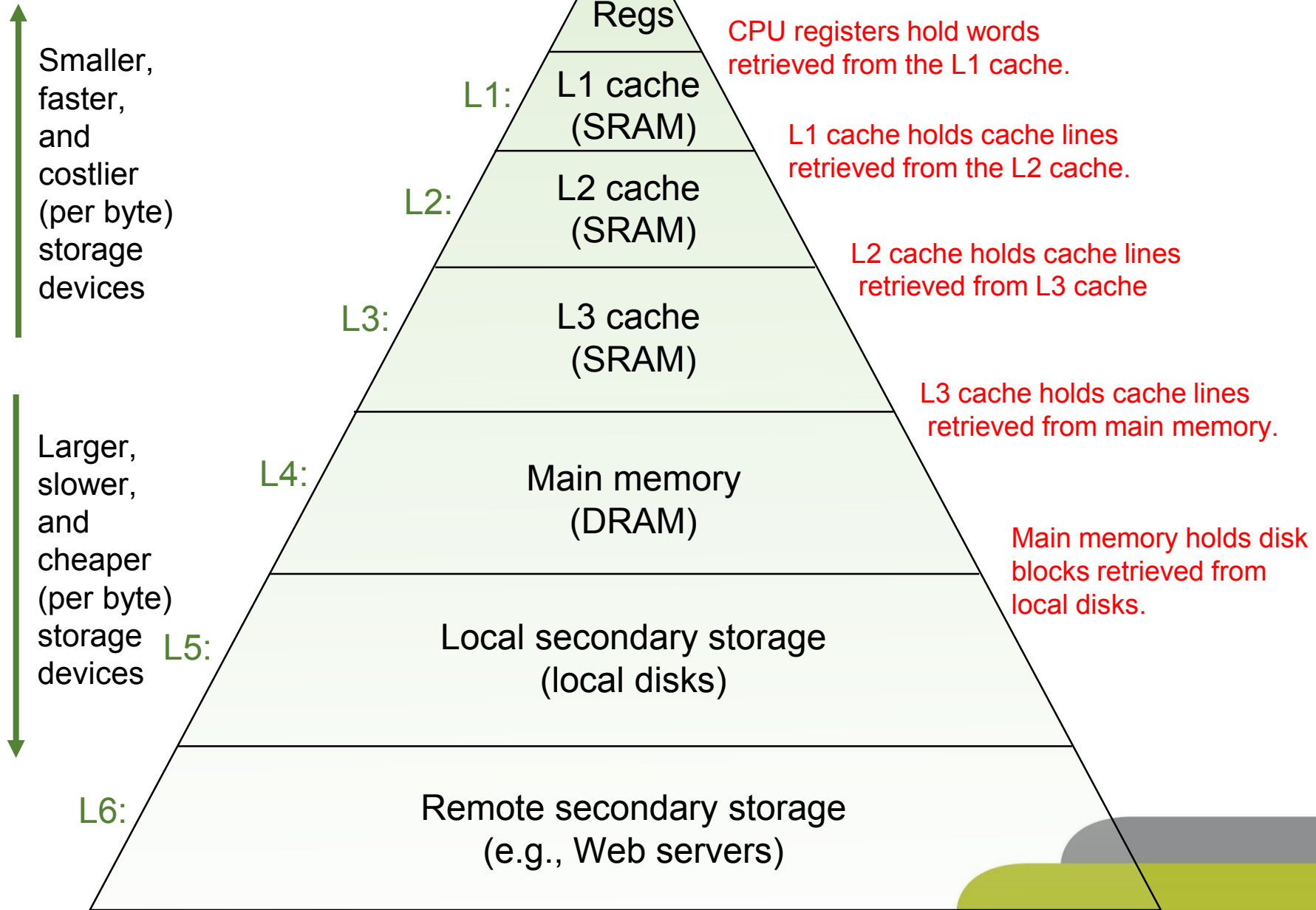
- Programs that repeatedly reference the same variables enjoy good temporal locality.
- For programs with stride-k reference patterns, the smaller the stride, the better the spatial locality.
- Programs with stride-1 reference patterns have good spatial locality.
- Programs that hop around memory with large strides have poor spatial locality.
- Loops have good temporal and spatial locality with respect to instruction fetches.
- The smaller the loop body and the greater the number of loop iterations, the better the locality.

Memory Hierarchies

Memory Hierarchy

- Some fundamental and enduring properties of hardware and software:
 - Fast storage technologies cost more per byte, have less capacity, and require more power.
 - The gap between CPU and main memory speed is widening.
 - Well-written programs tend to exhibit good locality.
- They suggest an approach for organizing memory and storage systems known as a **memory hierarchy**.

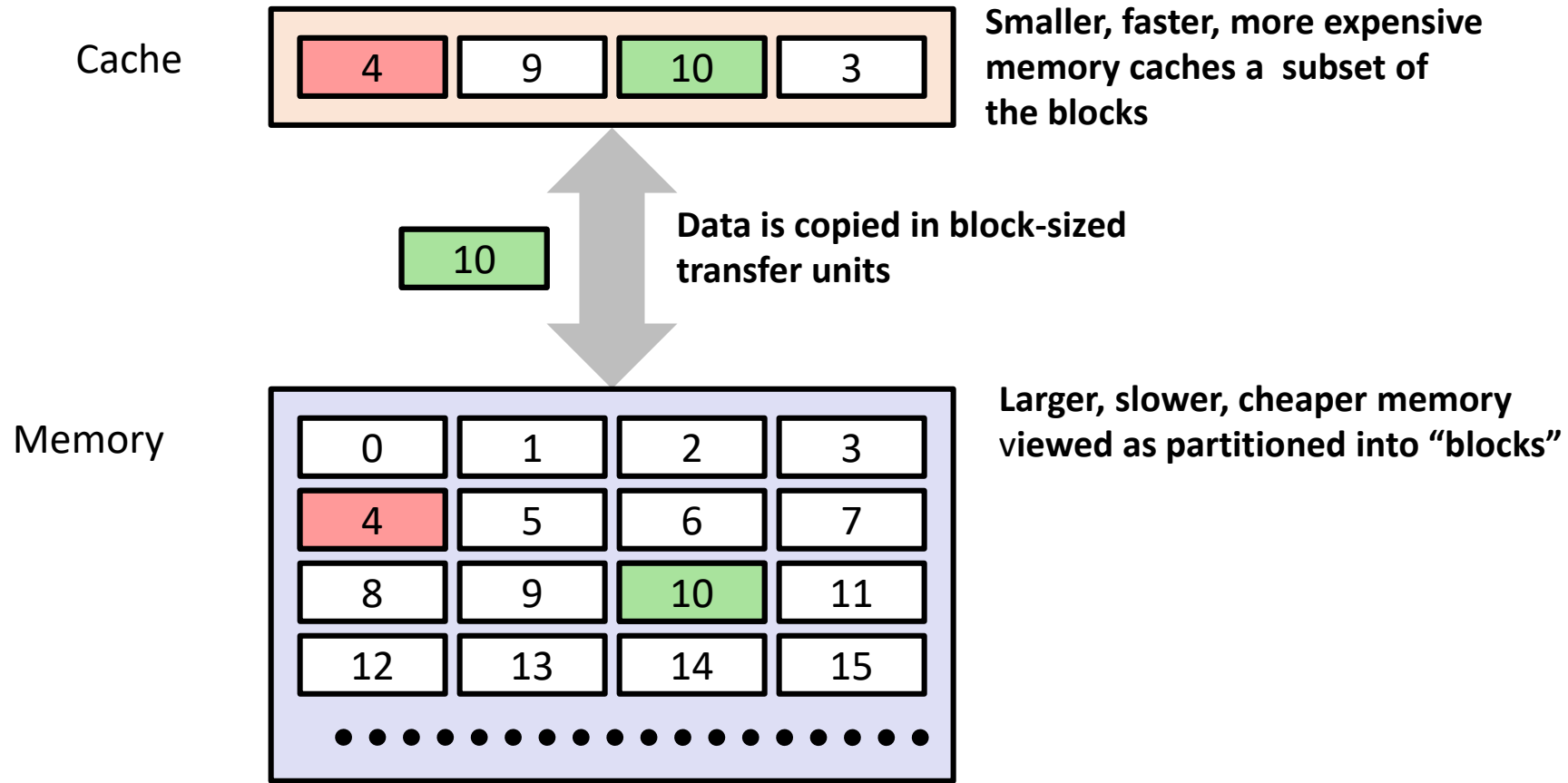
Memory Hierarchy



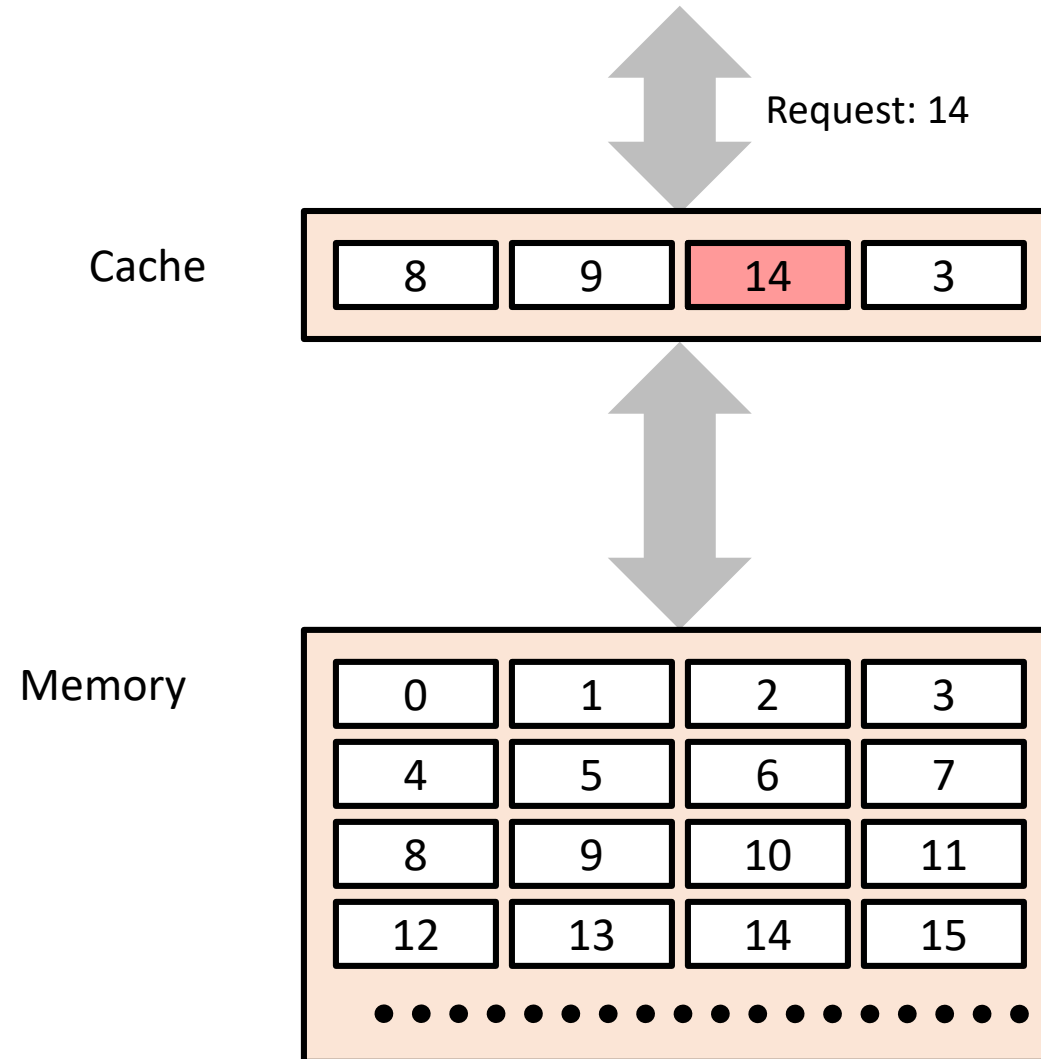
Caches

- **Cache:** A smaller, faster storage device that acts as a **staging area for a subset of the data in a larger, slower device.**
- **Fundamental idea of a memory hierarchy:**
 - For each k , the faster, smaller device at level k serves as a cache for the larger, slower device at level $k+1$.
 - Because of locality, programs tend to access the data at level k more often than they access the data at level $k+1$.
 - Thus, the storage at level $k+1$ can be slower, and thus larger and cheaper per bit.
- **Main idea:** the memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

General Cache Concepts



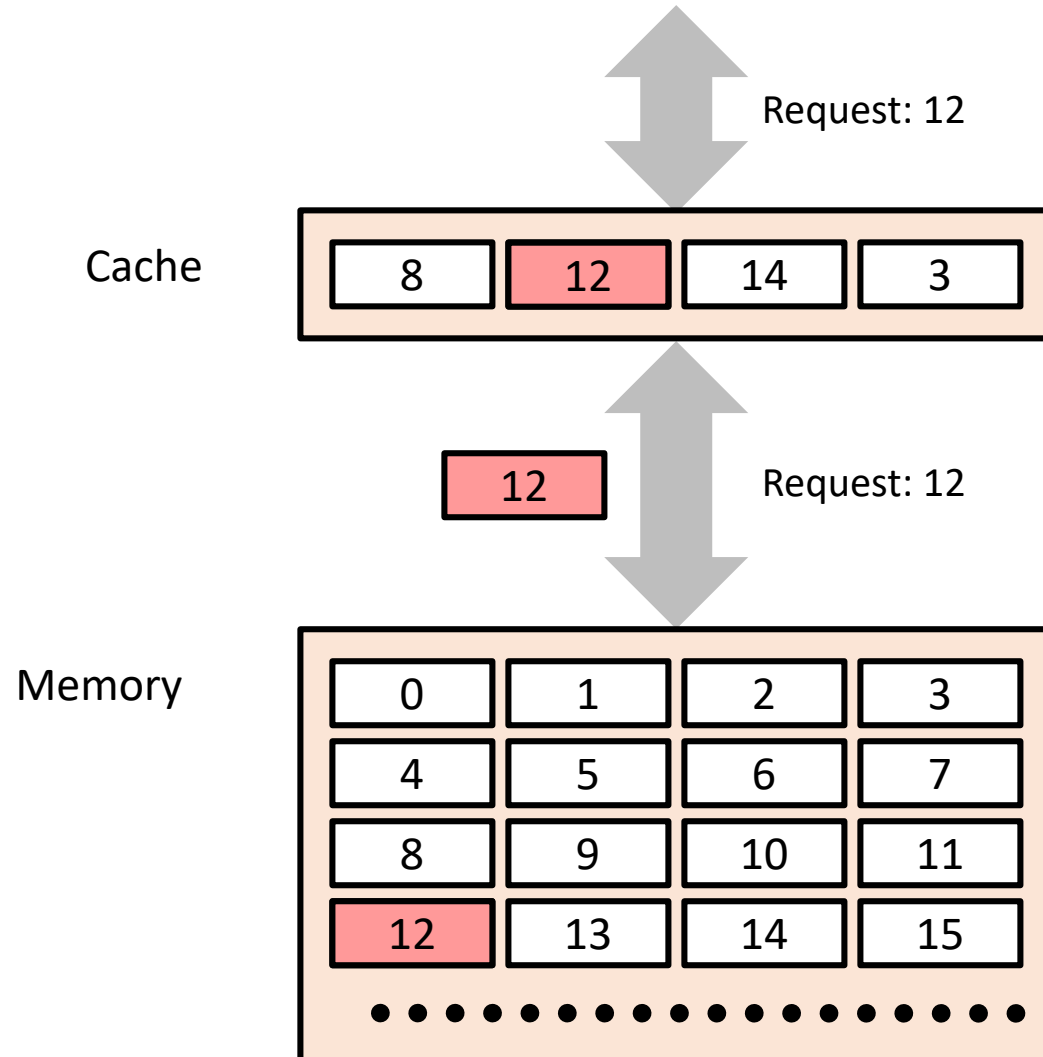
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

*Block b is fetched from
memory*

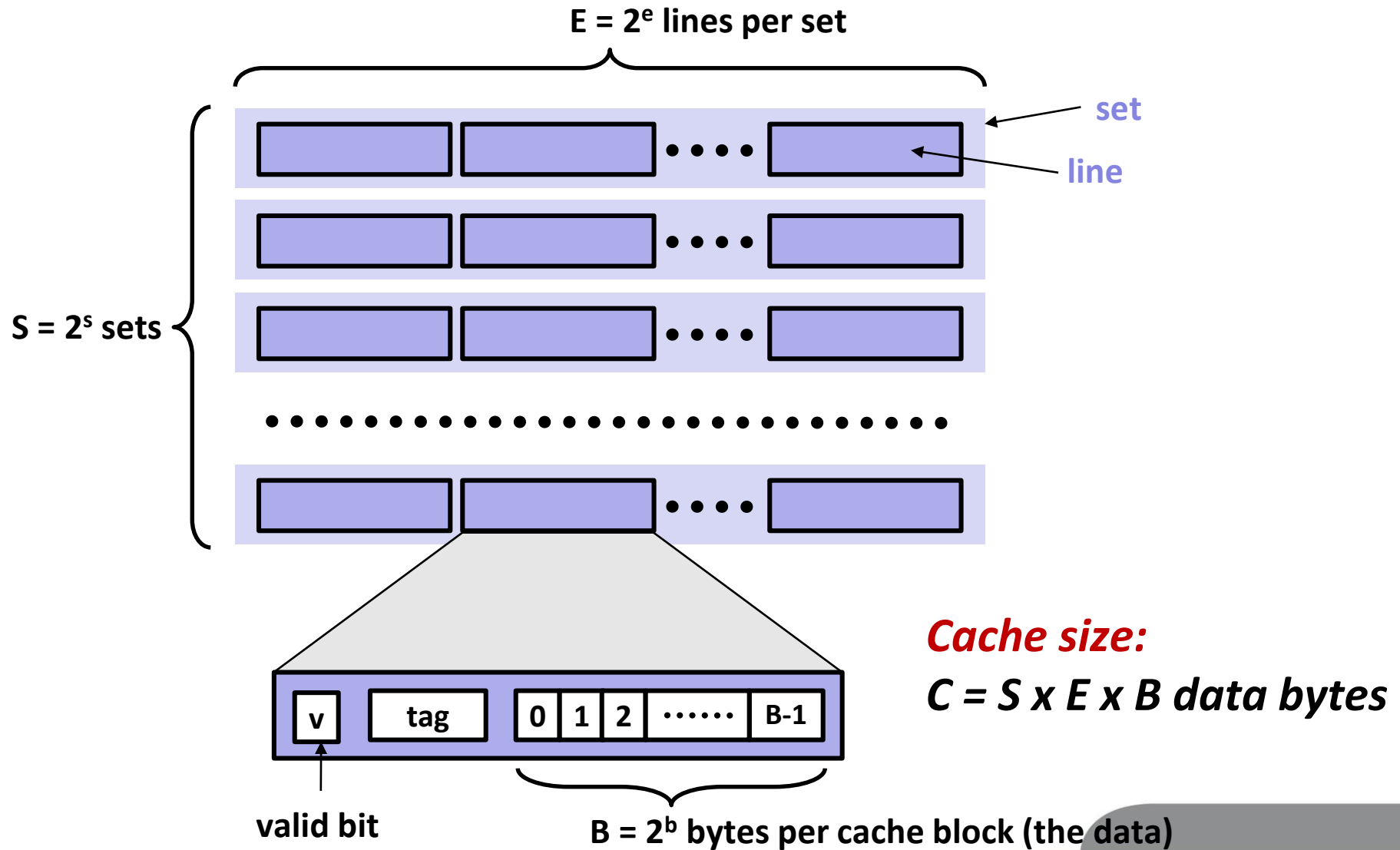
Block b is stored in cache

- **Placement policy:**
determines where b goes
- **Replacement policy:**
determines which block
gets evicted (victim)

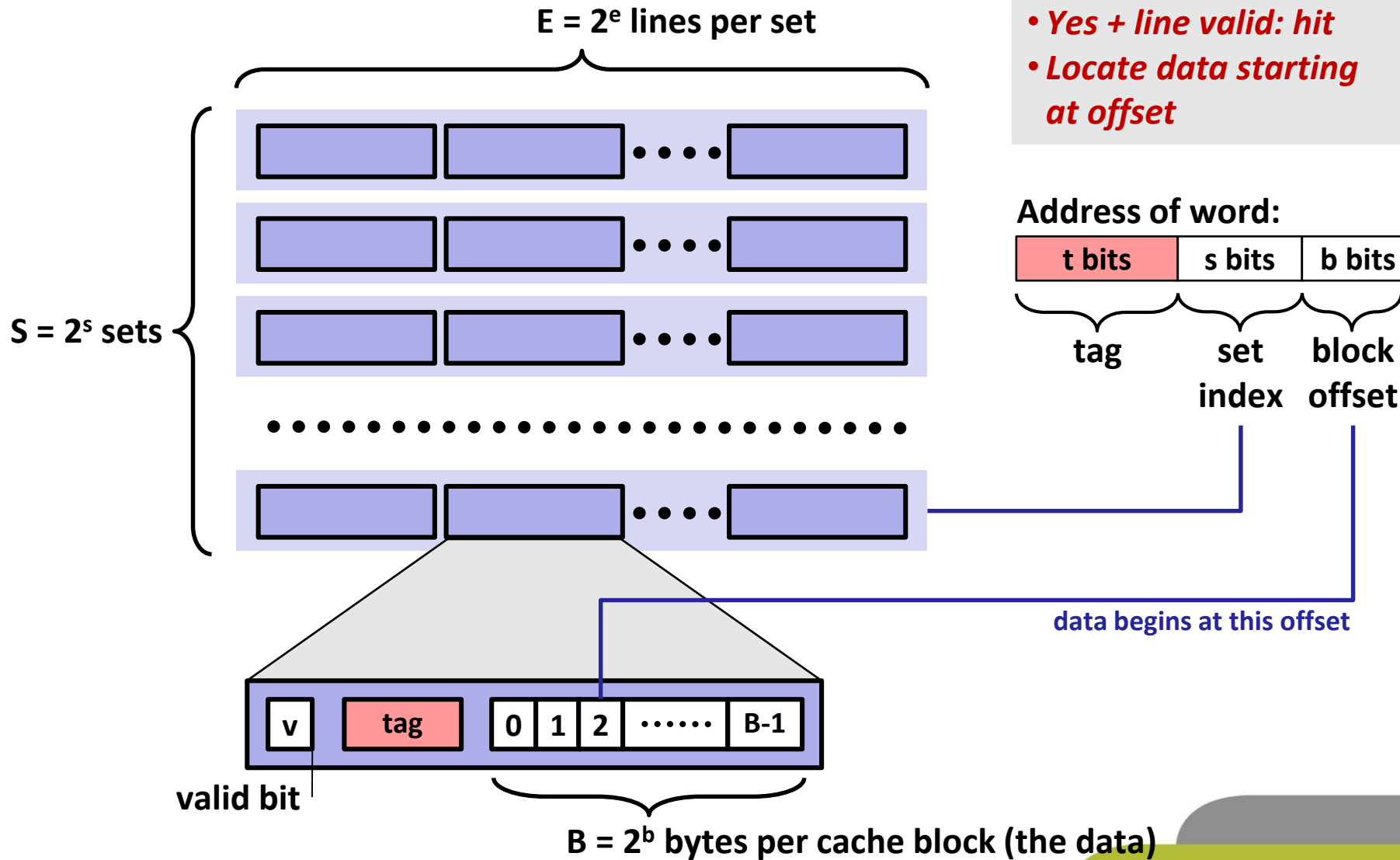
Types of Cache Misses

- **Cold (compulsory) miss**
 - Cold misses occur because the cache is empty.
- **Conflict miss**
 - Most caches limit blocks at level $k+1$ to a small subset of the block positions at level k .
 - E.g. Block i at level $k+1$ must be placed in block $(i \bmod 4)$ at level k .
 - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
 - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **Capacity miss**
 - Occurs when the set of active cache blocks (**working set**) is larger than the cache.

General Cache Organization



Cache Read

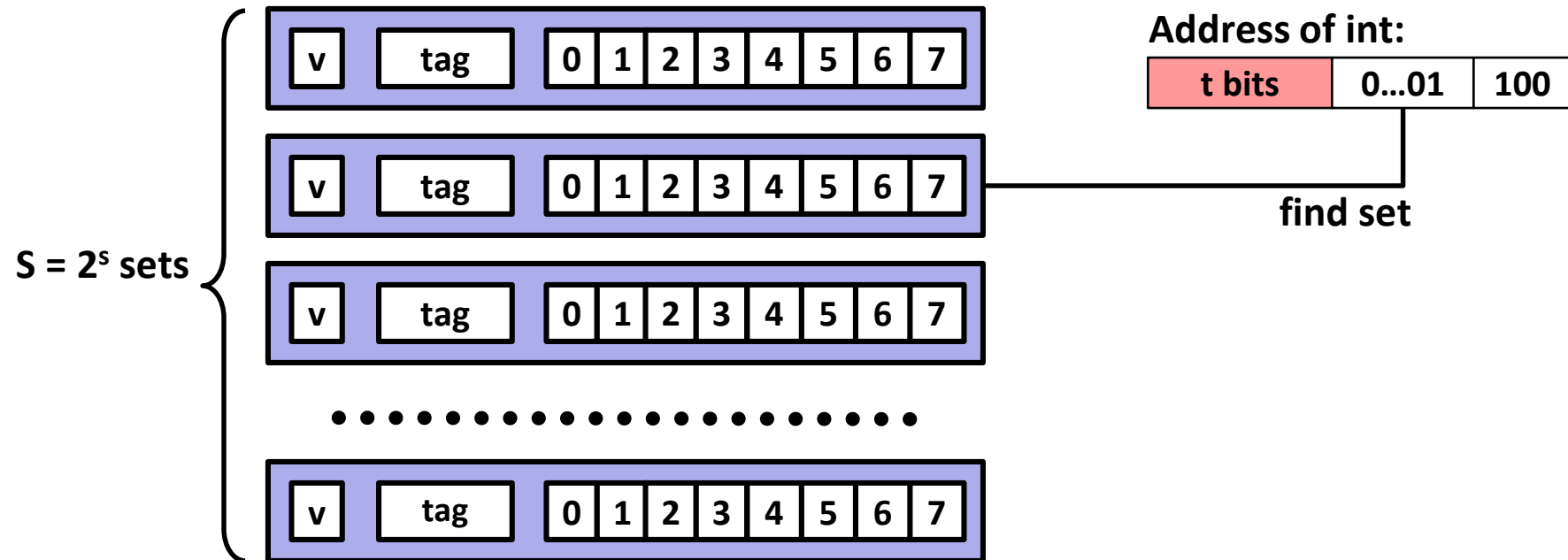


- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

Direct Mapped Cache (E = 1)

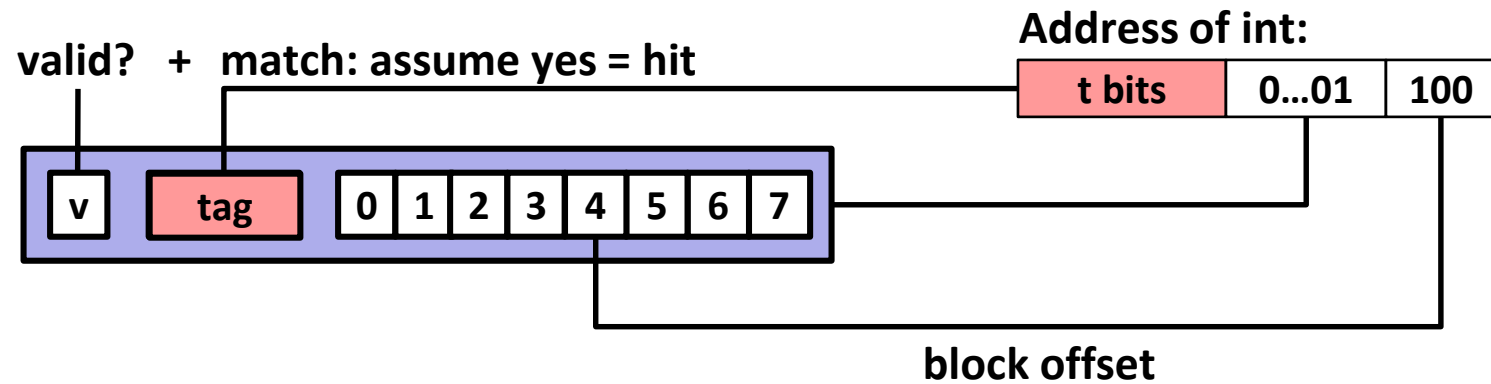
Direct mapped: One line per set

Assume: cache block size 8 bytes



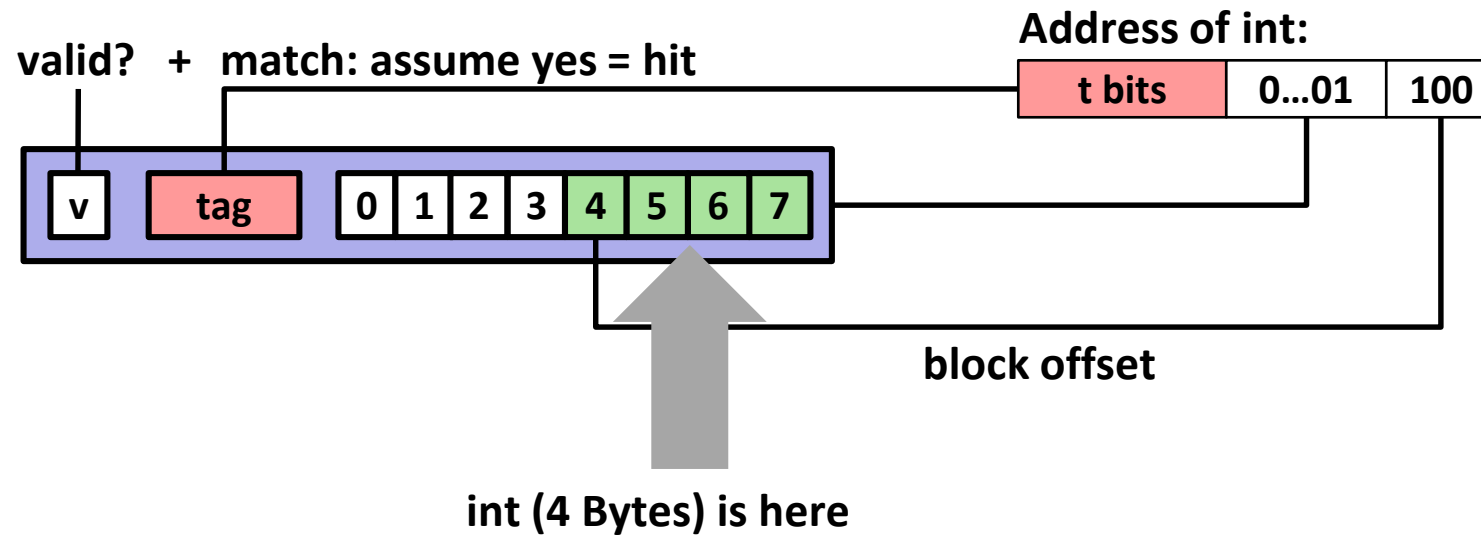
Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



If tag doesn't match: old line is evicted and replaced

Direct Mapped Cache Simulation Example

t=1	s=2	b=1
x	xx	x

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 Blocks/set

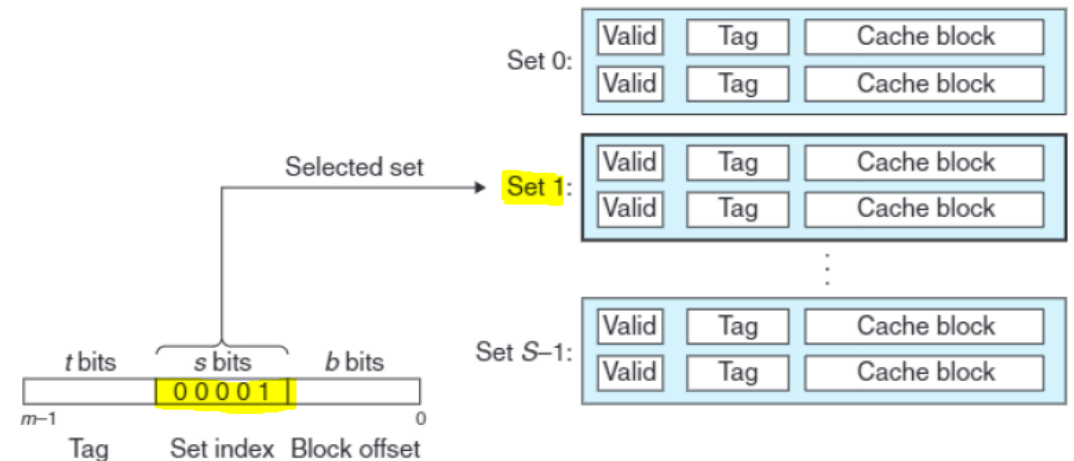
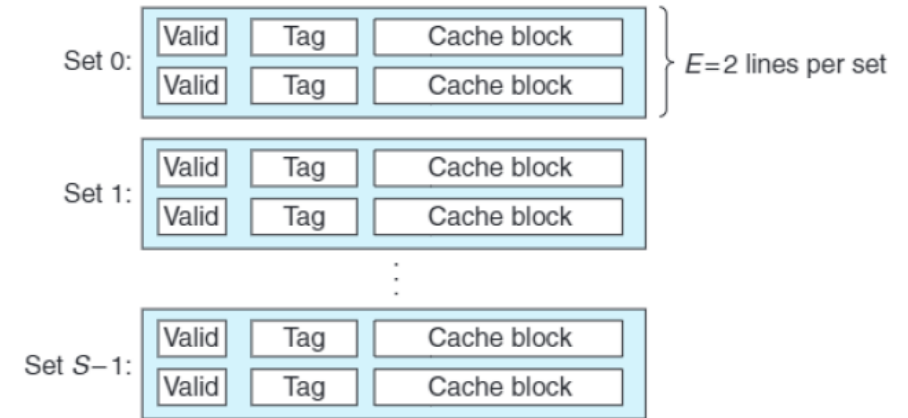
Address trace (reads, one byte per read):

0	[0000] ₂ ,	miss
1	[0001] ₂ ,	hit
7	[0111] ₂ ,	miss
8	[1000] ₂ ,	miss
0	[0000] ₂	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

E-way Set Associative Cache

- The problem with conflict misses in direct-mapped caches stems from the constraint that each set has exactly one line (or in our terminology, $E = 1$).
- A **set associative cache** relaxes this constraint so that each set holds more than one cache line.
- A cache with $1 < E < C/B$ is called an **E-way set associative cache**.



E-way Set Associative Cache (Here: $E = 2$)

- Set selection is identical to a direct-mapped cache, with the set index bits identifying the specific cache set.

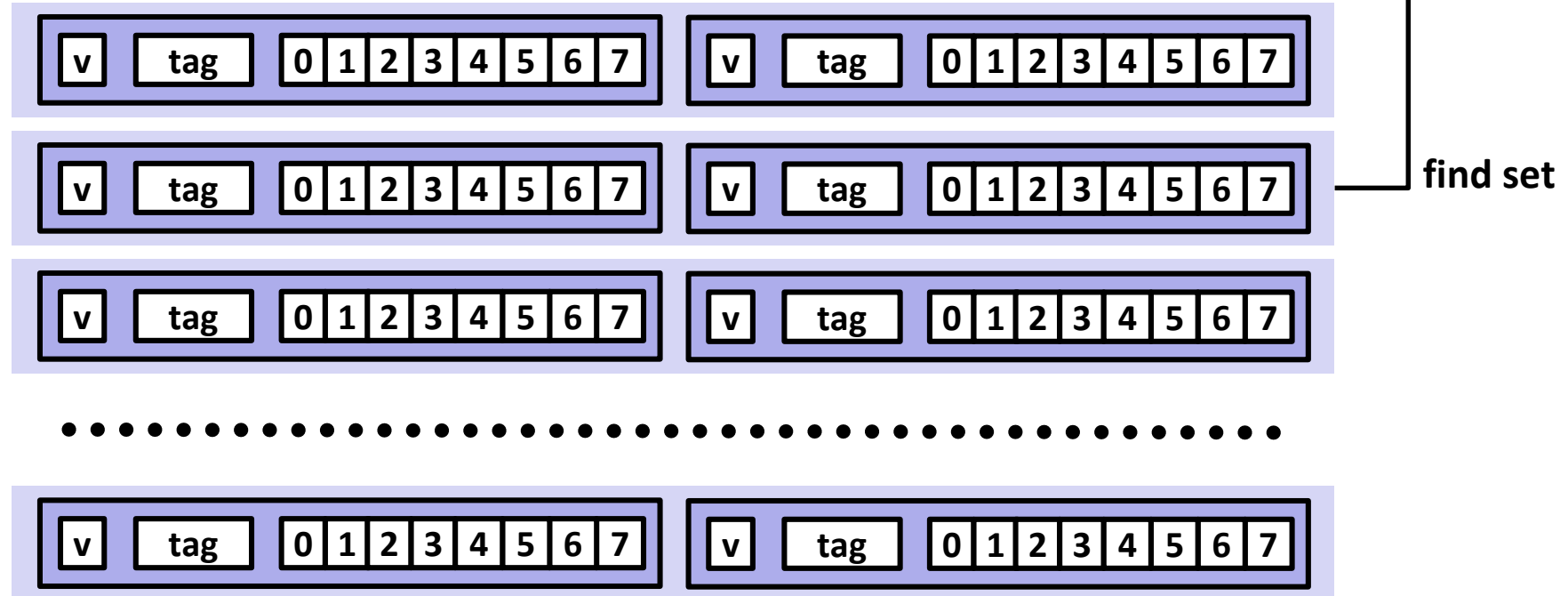
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:

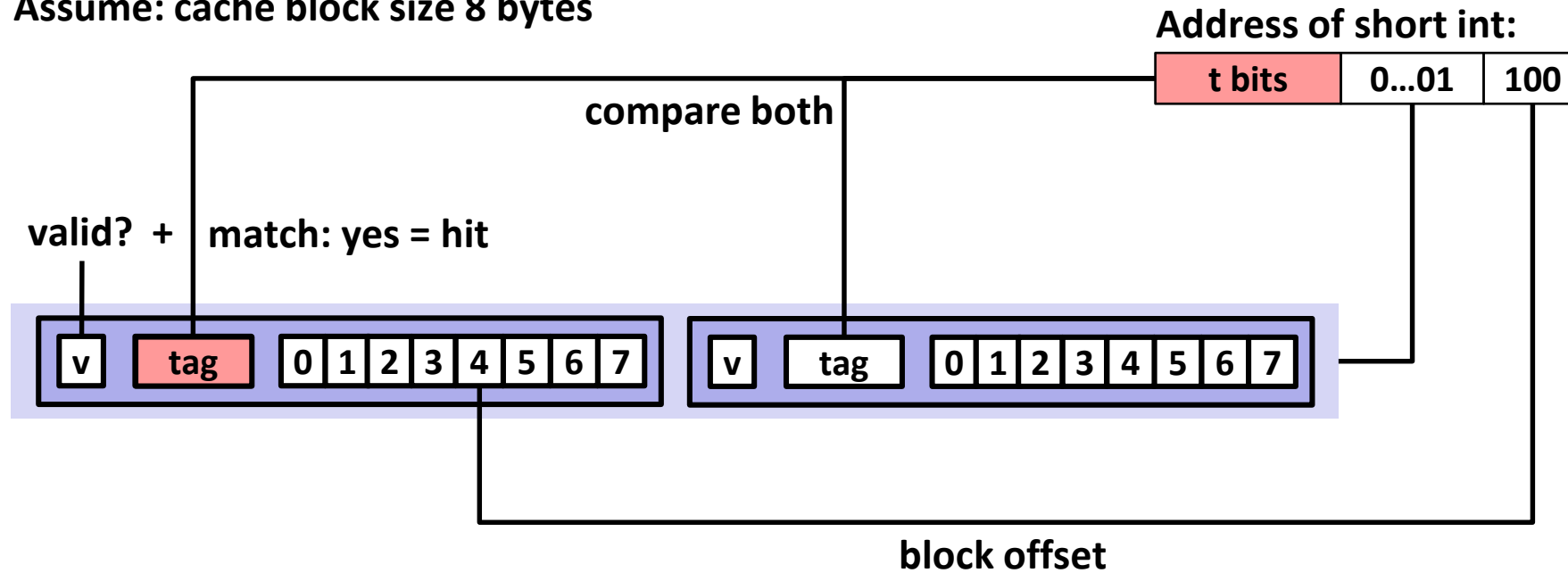
t bits	0...01	100



Line Matching and Word Selection in Set Associative Caches

E = 2: Two lines per set

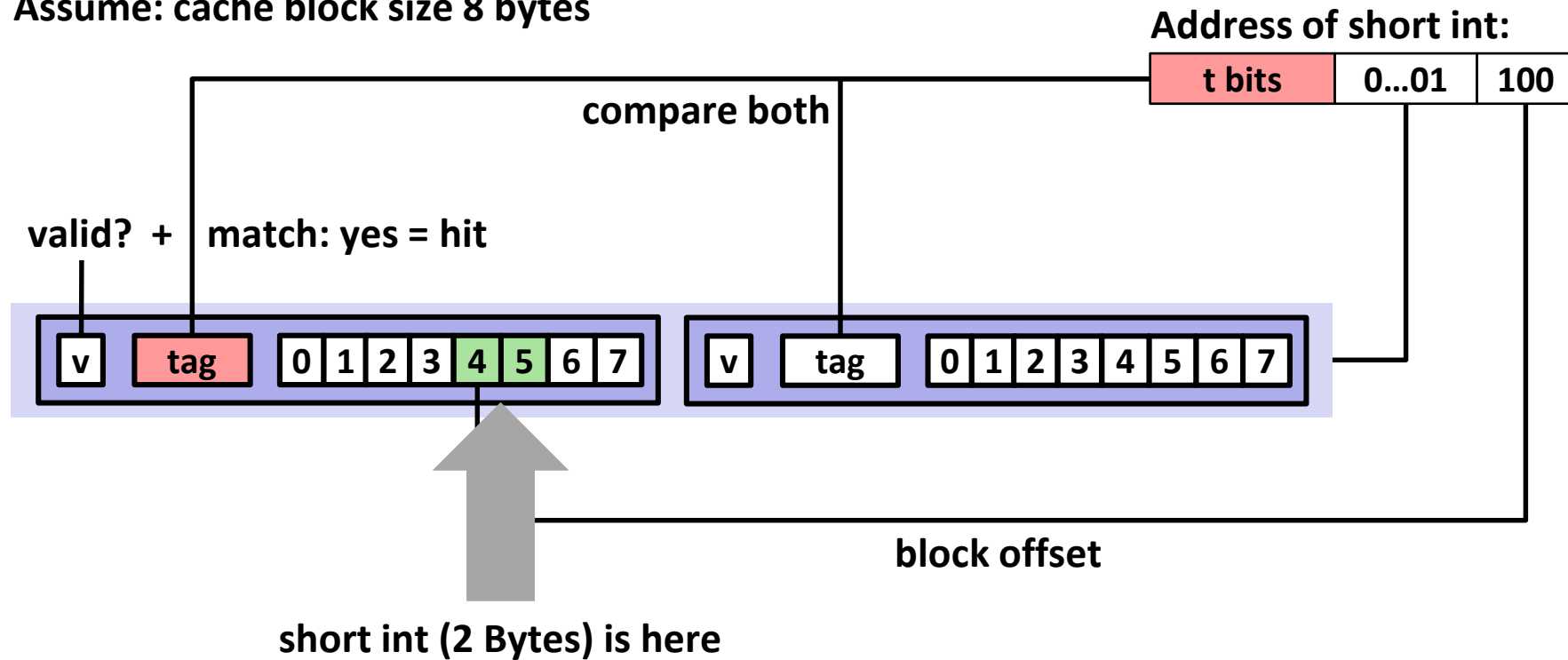
Assume: cache block size 8 bytes



Line Matching and Word Selection in Set Associative Caches

E = 2: Two lines per set

Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

E-way Set Associative Cache Simulation (Here: E = 2)

t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	[00 <u>00</u> ₂],	miss
1	[00 <u>01</u> ₂],	hit
7	[01 <u>11</u> ₂],	miss
8	[10 <u>00</u> ₂],	miss
0	[00 <u>00</u> ₂]	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

Replacement Algorithms

- Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced.
- For direct mapping, there is only one possible line for any particular block and no choice is possible.
- For set-associative techniques, a replacement algorithm is needed.

Most Common Replacement Algorithms

- **Least recently used (LRU)**
 - Most effective
 - Replace that block in the set that has been in the cache longest with no reference to it.
 - Because of its simplicity of implementation, LRU is the most popular replacement algorithm.
- **First-in-first-out (FIFO)**
 - Replace that block in the set that has been in the cache longest.
 - Easily implemented as a round-robin or circular buffer technique.
- **Least frequently used (LFU)**
 - Replace that block in the set that has experienced the fewest references.
 - Could be implemented by associating a counter with each line.

What about Write Transactions?

When a block that is resident in the cache is to be replaced there are two cases to consider:



If the old block in the cache has not been altered then it may be overwritten with a new block without first writing out the old block



If at least one write operation has been performed on a word in that line of the cache then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block

There are two problems to contend with:



More than one device may have access to main memory



A more complex problem occurs when multiple processors are attached to the same bus and each processor has its own local cache - if a word is altered in one cache it could conceivably invalidate a word in other caches

What about Write Transactions?

- As we have seen, the operation of a cache with respect to reads is simple.
 - First, look for a copy of the desired word w in the cache. If there is a hit, return w immediately.
 - If there is a miss, fetch the block that contains w from the next lower level of the memory hierarchy, store the block in some cache line (possibly evicting a valid line), and then return w .
- The situation for writes is a little more complicated.
- Suppose we write (i.e. modify) a word w that is already cached (**a write hit**).
- What should the cache do now?
 1. Immediately write w 's cache block to the next lower level (**write-through**).
 2. Defer the update by writing the updated block to the next lower level only when it is evicted from the cache by the replacement algorithm (**write-back**).

Write Hits

- **Write-through**
 - Simplest technique
 - All write operations are made to main memory as well as to the cache.
 - The main disadvantage of this technique is that it generates substantial memory traffic and may create a bottleneck.
- **Write back**
 - Minimizes memory writes.
 - Updates are made only in the cache.
 - When an update occurs, a dirty bit, or use bit, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set.
 - Portions of main memory are invalid and hence accesses by I/O modules can be allowed only through the cache.
 - This makes for complex circuitry and a potential bottleneck.

Write Misses

- How do we deal with write misses?
 - The block containing the word to be written is fetched from main memory (or next level cache) into the cache and the processor proceeds with the write cycle (**write-allocate**).
 - The block containing the word to be written is modified in the main memory and not loaded into the cache, bypassing the cache (**no-write-allocate**).
- **Write-through** caches are typically **no-write-allocate**.
 - The reasoning is that even if locality holds and a write will be made to the same block in the near future, the write-through policy will generate a write to main memory anyway, so bringing the block into the cache does not seem efficient.
- **Write-back** caches are typically **write-allocate**.
 - The reasoning is that subsequent writes to the same block, if the block originally caused a miss, will hit in the cache next time, setting the dirty bit for the block. That will eliminate extra memory accesses and result in efficient execution. The write back, no write allocate option takes advantage of locality of reference.
- The implementation details vary from system to system and are often proprietary and poorly documented.

Cache Miss Rate

- **Miss Rate**
 - Fraction of memory references not found in cache (misses / accesses)
= 1 – hit rate
 - Typical numbers (in percentages):
 - 3-10% for L1
 - can be quite small (e.g., < 1%) for L2, depending on size.
- **Hit Time**
 - Time to deliver a line in the cache to the processor
 - includes time to determine whether the line is in the cache
 - Typical numbers:
 - 4 clock cycles for L1
 - 10 clock cycles for L2
- **Miss Penalty**
 - Additional time required because of a miss
 - typically 50-200 cycles for main memory (Trend: increasing!)

Cache Miss Rate or Hit Rate?

- Huge difference between a hit and a miss
 - Could be 100x, if just L1 and main memory
- 99% hits is twice as good as 97%!!!
 - Consider:
 - cache hit time of 1 cycle
 - miss penalty of 100 cycles
 - Average access time:
 - 97% hits: $1 \text{ cycle} + 0.03 * 100 \text{ cycles} = \mathbf{4 \text{ cycles}}$
 - 99% hits: $1 \text{ cycle} + 0.01 * 100 \text{ cycles} = \mathbf{2 \text{ cycles}}$
- **This is why miss rate is used instead of hit rate!!!**

Qualitative Rules for Writing Cache Friendly Code

- **Make the common case go as fast as possible**
 - Focus on the inner loops of the core functions
- **Minimize the misses in the inner loops**
 - Repeated references to variables are good (temporal locality)
 - Stride-1 reference patterns are good (spatial locality)

Matrix Multiplication Example

- Description:
 - Multiply $N \times N$ matrices
 - Matrix elements are doubles (8 bytes)
 - N reads per source element
 - N values summed per destination
 - but may be able to hold in register
- Assume:
 - Block size = 32B (big enough for four doubles)
 - Matrix dimension (N) is very large
 - Approximate $1/N$ as 0.0
 - Cache is not big enough to hold multiple rows

Variable sum held in register

```
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        sum = 0.0;   
        for (k=0; k<n; k++)  
            sum += a[i][k] * b[k][j];  
        c[i][j] = sum;  
    }  
}
```

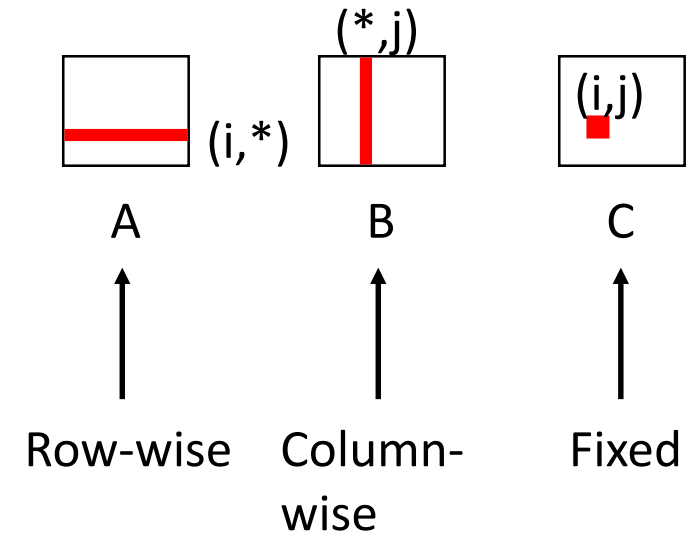
Matrix Multiplication Example

```

for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}

```

Inner loop:



Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2395

Intro to Computer Architecture

Lecture 8

Plan for Today

- Virtual Memory
 - Physical Addressing
 - Virtual Addressing
 - Address Spaces
 - VM as A Tool for Caching
 - Locality to the Rescue!
 - Virtual Memory as a Tool for Memory Management
 - Virtual Memory as a Tool for Memory Protection



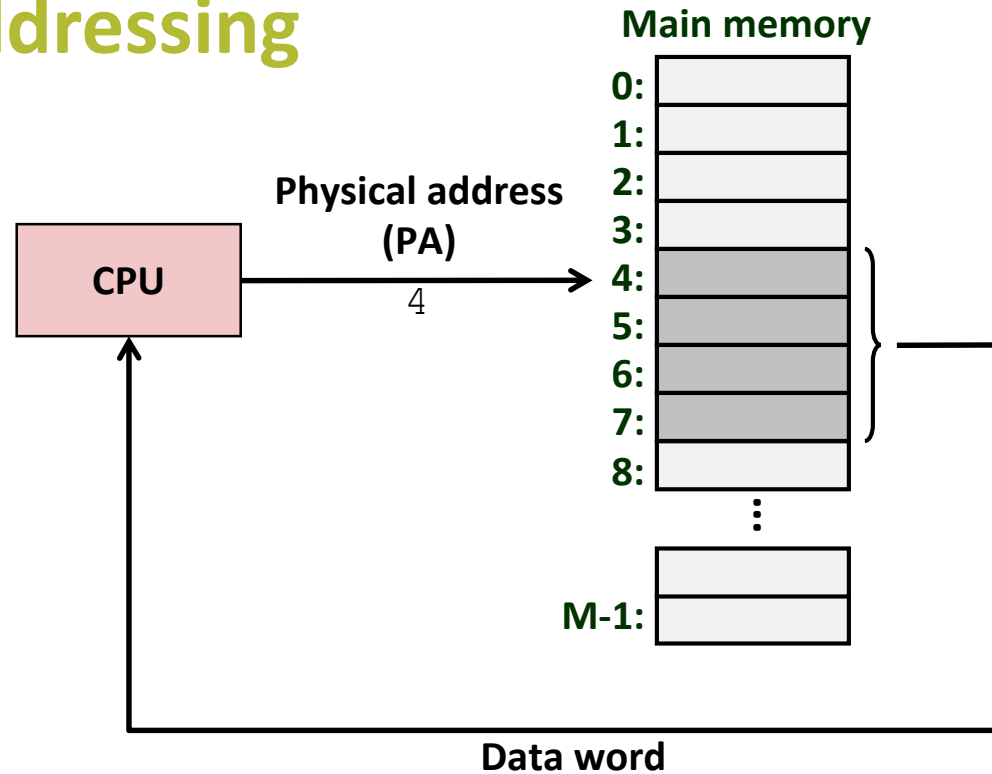
DOUGLAS COLLEGE

Physical Addressing

Physical Addressing

- The main memory of a computer system is organized as an array of M contiguous byte-size cells.
- Each byte has a unique physical address.
- The first byte has an address of 0, the next byte an address of 1, the next byte an address of 2, and so on.
- Given this simple organization, the most natural way for a CPU to access memory would be to use physical addresses.
- We call this approach **physical addressing**.
- The next slide shows an example of physical addressing in the context of a load instruction that reads the 4-byte word starting at physical address 4.

Physical Addressing



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

Physical Addressing

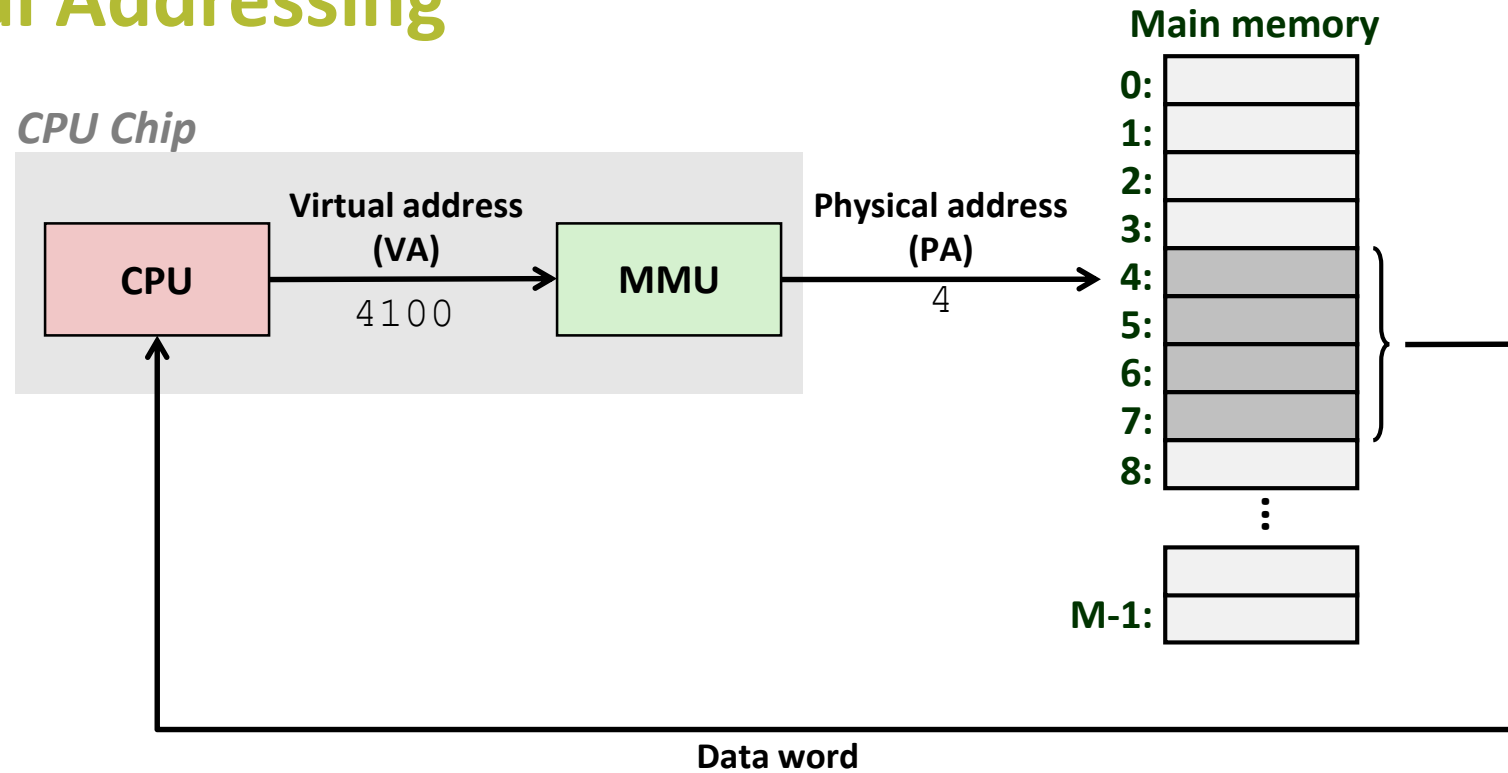
- The main memory of a computer system is organized as an array of M contiguous byte-size cells.
- Each byte has a unique physical address.
- The first byte has an address of 0, the next byte an address of 1, the next byte an address of 2, and so on.
- Given this simple organization, the most natural way for a CPU to access memory would be to use physical addresses.
- We call this approach **physical addressing**.
- The next slide shows an example of physical addressing in the context of a load instruction that reads the 4-byte word starting at physical address 4.
- When the CPU executes the load instruction, it generates an effective physical address and passes it to main memory over the memory bus.
- The main memory fetches the 4-byte word starting at physical address 4 and returns it to the CPU, which stores it in a register.

Virtual Addressing

Virtual Addressing

- With **virtual addressing**, the CPU accesses main memory by generating a virtual address, which is converted to the appropriate physical address before being sent to main memory (DRAM).
- The task of converting a virtual address to a physical one is known as **address translation**.
- Address translation requires close cooperation between the CPU hardware and the operating system.
- Dedicated hardware on the CPU chip called the **memory management unit (MMU)** translates virtual addresses on the fly, using a lookup table stored in main memory whose contents are managed by the operating system.

Virtual Addressing



- Used in all modern servers, laptops, and smart phones.
- One of the great ideas in computer science.

Address Spaces

Address Spaces

- An **address space** is an ordered set of nonnegative integer addresses
$$\{0, 1, 2, 3, \dots\}$$
- If the integers in the address space are consecutive, then we say that it is a **linear address space**. In this course, we will always assume linear address spaces.
- In a system with virtual memory, the CPU generates virtual addresses from an address space of $N = 2^n$ addresses called the virtual address space:
$$\{0, 1, 2, 3, \dots, N-1\}$$
- The size of an address space is characterized by the number of bits that are needed to represent the largest address.
- A virtual address space with $N = 2^n$ addresses is called an **n-bit address space**.
- Modern systems typically support either 32-bit or 64-bit virtual address spaces.

Address Spaces

- A system also has a physical address space that corresponds to the $M=2^m$ bytes of physical memory in the system:

$$\{0, 1, 2, 3, \dots, M-1\}$$

- M is not required to be a power of 2, but to simplify our discussion, we will assume that $M = 2^m$.
- The concept of an address space is important because it makes a distinction between data objects (bytes) and their attributes (addresses).
- Once we recognize this distinction, then we can generalize and **allow each data object to have multiple independent addresses**, each chosen from a different address space.
- **This is the basic idea of virtual memory:**
 - Each byte of main memory has a virtual address chosen from the virtual address space, and a physical address chosen from the physical address space.

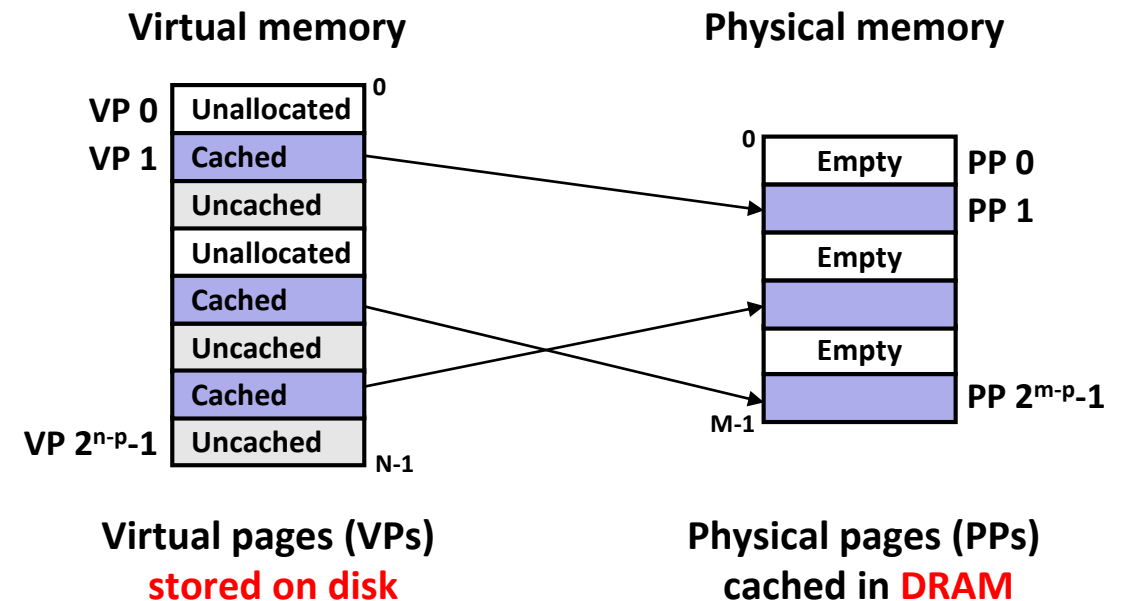
Virtual Memory: A Tool for Caching

VM as a Tool for Caching

- Conceptually, a virtual memory is organized as an array of N contiguous byte-size cells stored on disk.
- Each byte has a unique virtual address that serves as an index into the array.
- The contents of the array on disk are cached in main memory (DRAM).
- The data on disk (the lower level) is partitioned into blocks that serve as the transfer units between the disk and the main memory (the upper level).
- VM systems partition the virtual memory into fixed-size blocks called **virtual pages (VPs)**.
- Each virtual page is $P = 2^p$ bytes in size.
- Similarly, physical memory is partitioned into **physical pages (PPs)**, also P bytes in size. (Physical pages are also referred to as **page frames**.)

VM as a Tool for Caching

- At any point in time, the set of virtual pages is partitioned into three subsets:
 - Unallocated:** Pages that have not yet been allocated by the VM system.
 - Unallocated blocks do not have any data associated with them, and thus do not occupy any space on disk.
 - Cached:** Allocated pages that are currently cached in physical memory.
 - Uncached:** Allocated pages that are not cached in physical memory.



VM as a Tool for Caching

■ DRAM cache organization driven by the huge miss penalty

- DRAM is about **10x** slower than SRAM.
- Disk is about **10,000x** slower than DRAM.

■ Consequences for VM caching

- Large page (block) size: typically 4 KB, sometimes 4 MB
- **Fully associative mapping:**
 - Any VP (from disk) can be placed in any PP in DRAM
 - Requires a “large” mapping function – different from cache memories
- Highly sophisticated, expensive replacement algorithms
 - Too complicated and open-ended to be implemented in hardware
- Use Write-back rather than write-through because of the large access time of disk

VM as a Tool for Caching

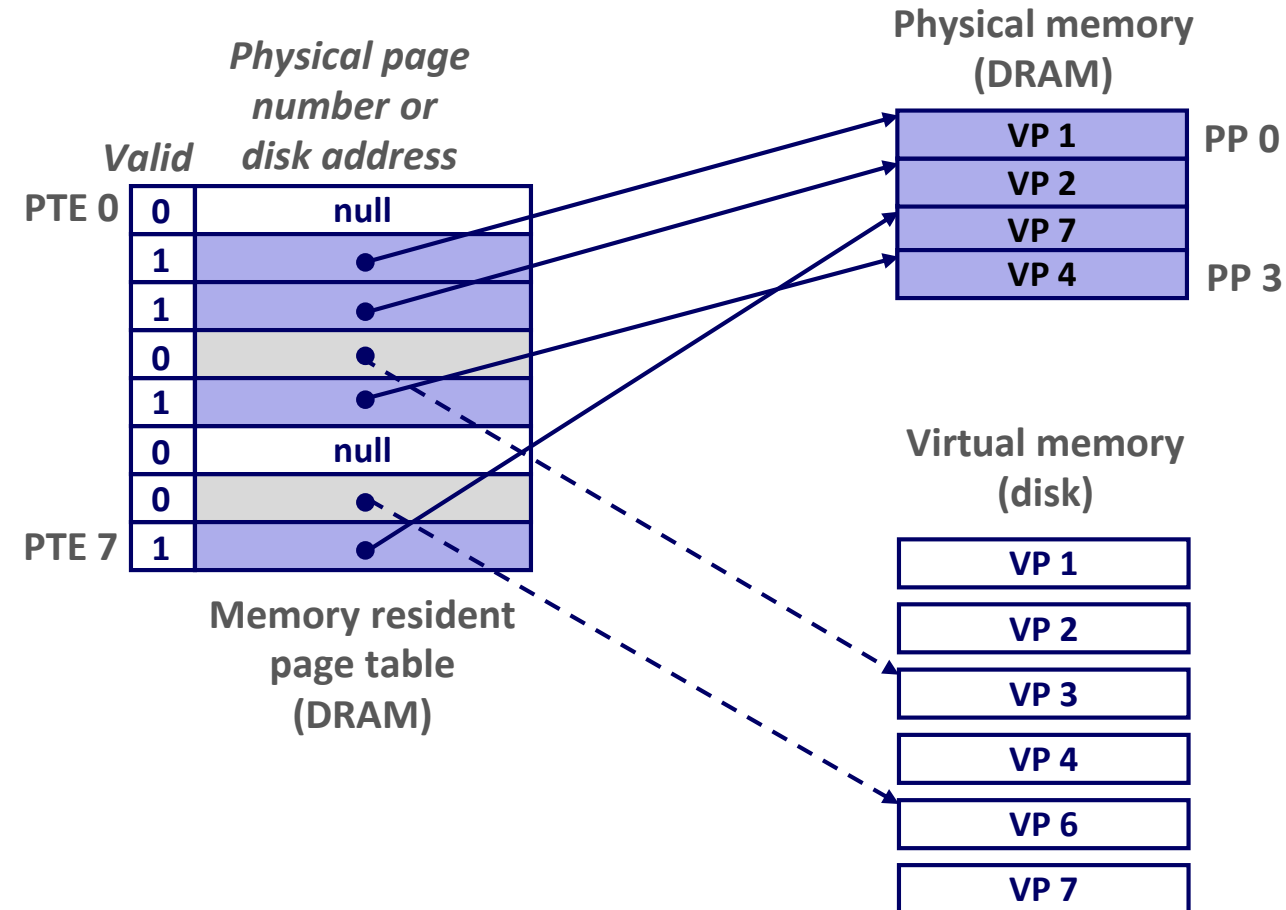
- As with any cache, the VM system must have some way to determine if a virtual page is cached somewhere in DRAM.
- If a virtual page is cached in DRAM, the system must determine which physical page in DRAM it is cached in.
- If there is a miss, the system must determine where the virtual page is stored on disk, select a **victim page** in DRAM, and copy the virtual page from disk to DRAM, replacing the victim page.
- These capabilities are provided by the **operating system, address translation hardware in the MMU (memory management unit)**, and a data structure stored in DRAM known as **a page table that maps virtual pages to physical pages**.
- The address translation hardware reads the page table each time it converts a virtual address to a physical address.
- The operating system maintains the contents of the page table and transfers pages back and forth between disk and DRAM.

The Page Table

- A **page table** is an array of **page table entries (PTEs)** that maps virtual pages to physical pages.
- Each page in the virtual address space has a PTE in the page table.
- Each PTE consists of a **valid bit** and an **n-bit address field**.
- The valid bit indicates whether the virtual page is currently cached in DRAM.
- If the valid bit is set to 1, the address field indicates the start of the corresponding physical page in DRAM where the virtual page is cached.
- If the valid bit is **not** set, then a null address indicates that the virtual page has not yet been allocated.
- Otherwise, the address points to the start of the virtual page on disk.

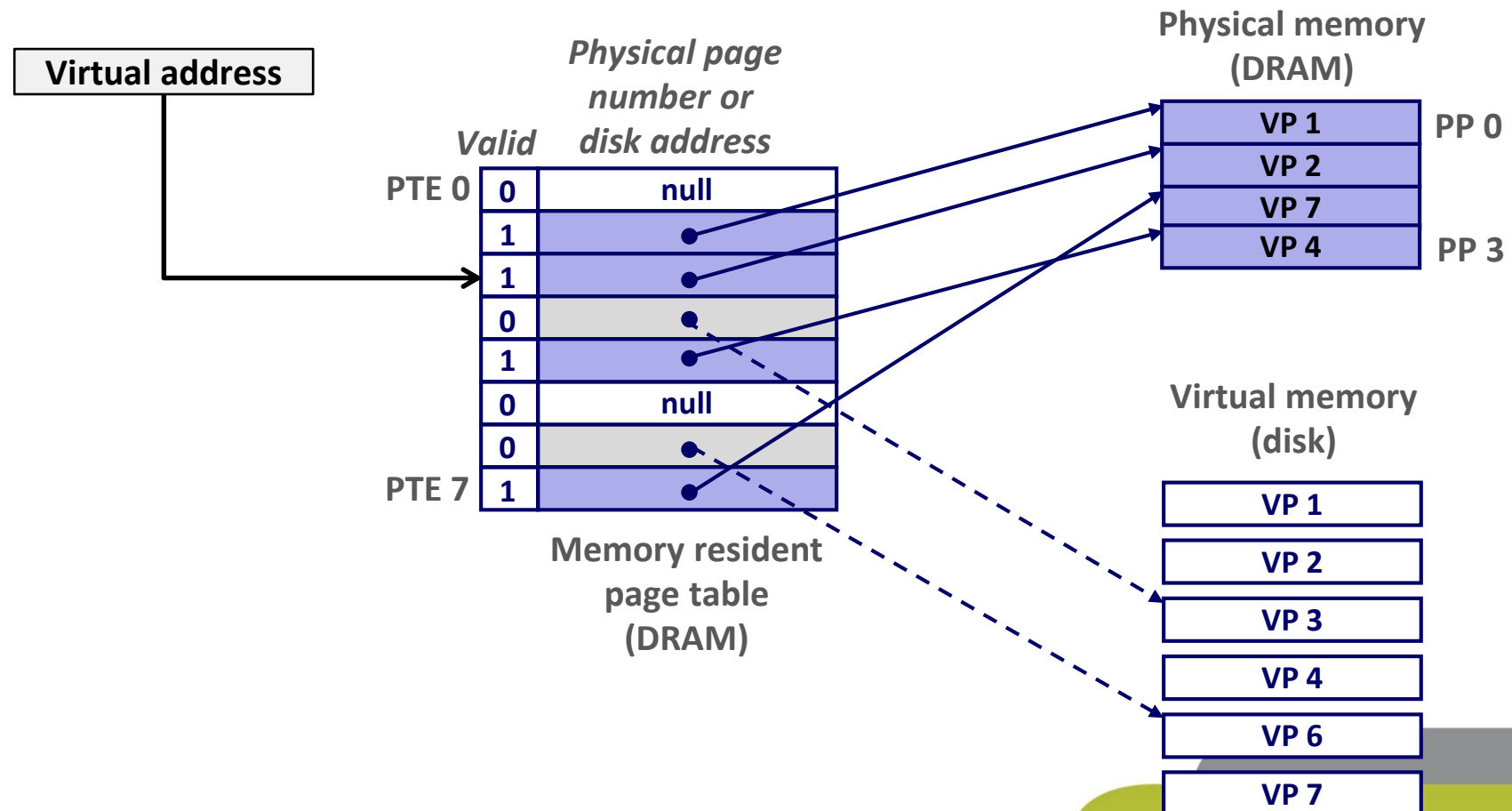
The Page Table

- Here is a page table for a system with eight virtual pages and four physical pages. Four virtual pages (VP 1, VP 2, VP 4, and VP 7) are currently cached in DRAM.
- Two pages (VP 0 and VP 5) have not yet been allocated, and the rest (VP 3 and VP 6) have been allocated but are not currently cached.
- Since the DRAM cache is fully associative, any physical page can contain any virtual page.



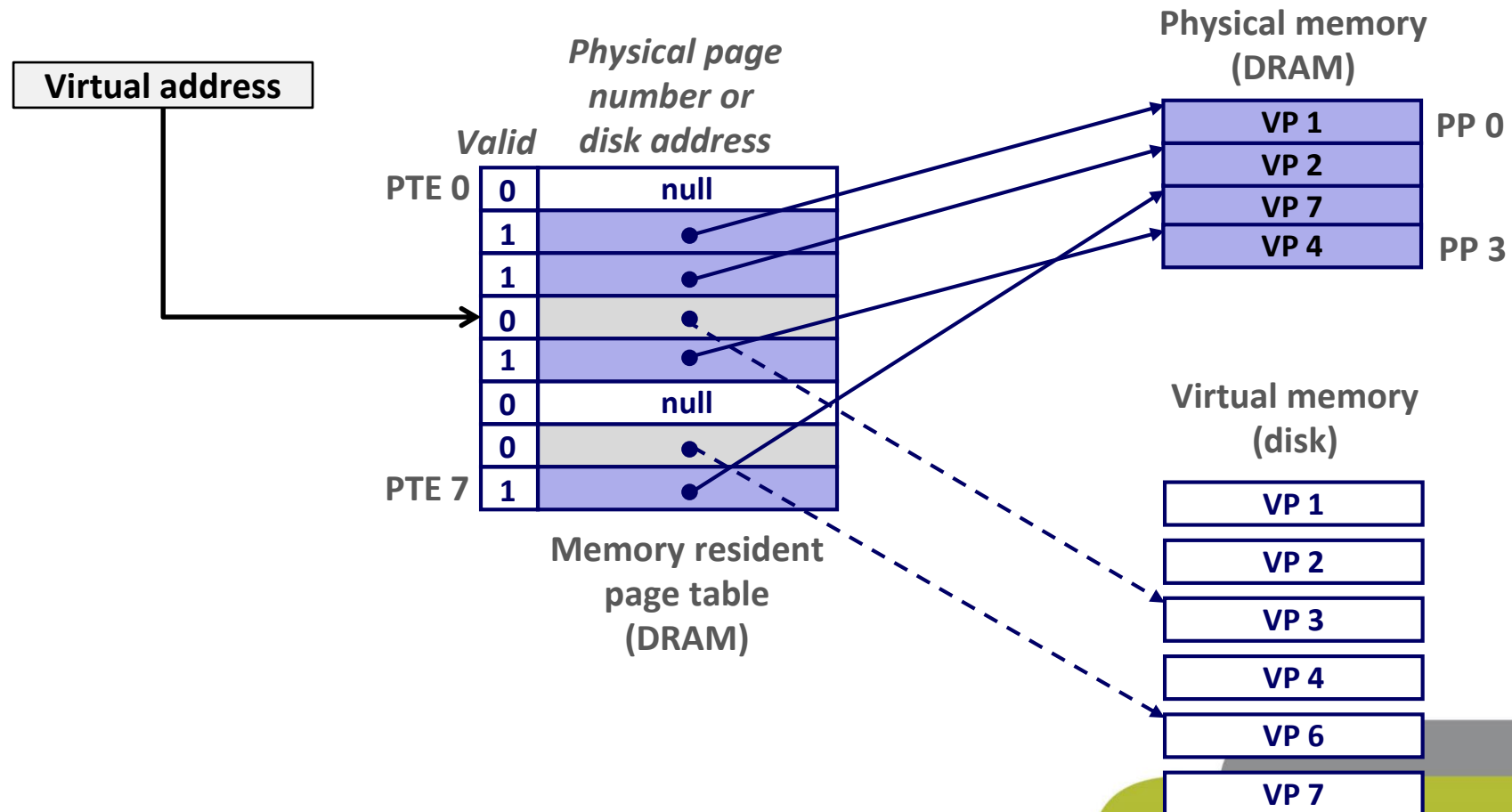
Page Hit (DRAM Cache Hit)

- When the CPU reads a word of virtual memory contained in VP 2 (cached in DRAM), the address translation hardware uses the virtual address to locate PTE 2 and read it from memory. Since the valid bit is set, the address translation hardware uses the DRAM address in the PTE (which points to the start of the cached page in PP 1) to construct the physical address of the word.



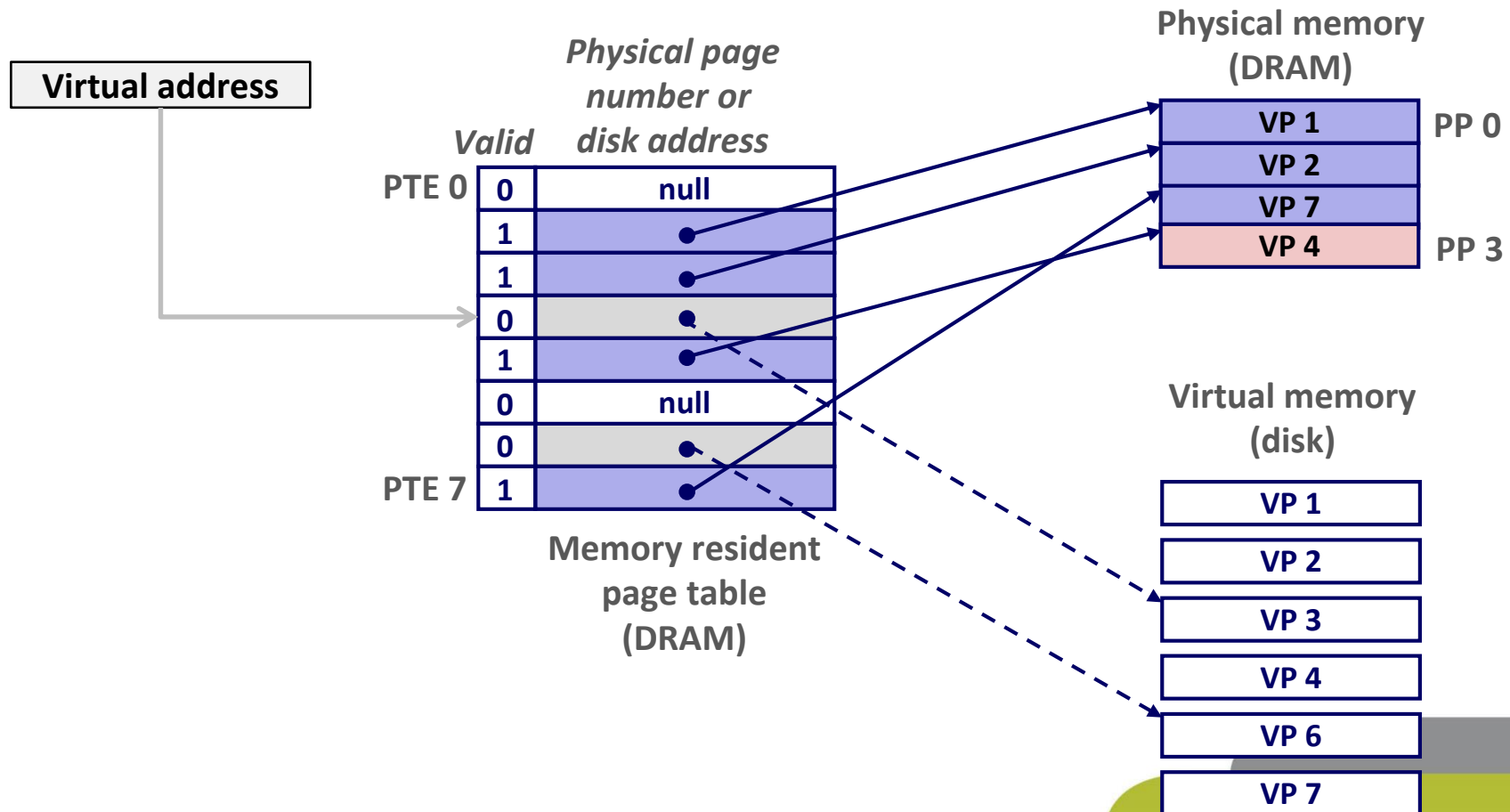
Page Fault (DRAM Cache Miss)

- The CPU has referenced a word in VP 3, which is not cached in DRAM. The address translation hardware reads PTE 3 from DRAM, infers from the valid bit that VP 3 is not cached, and triggers a **page fault exception**.



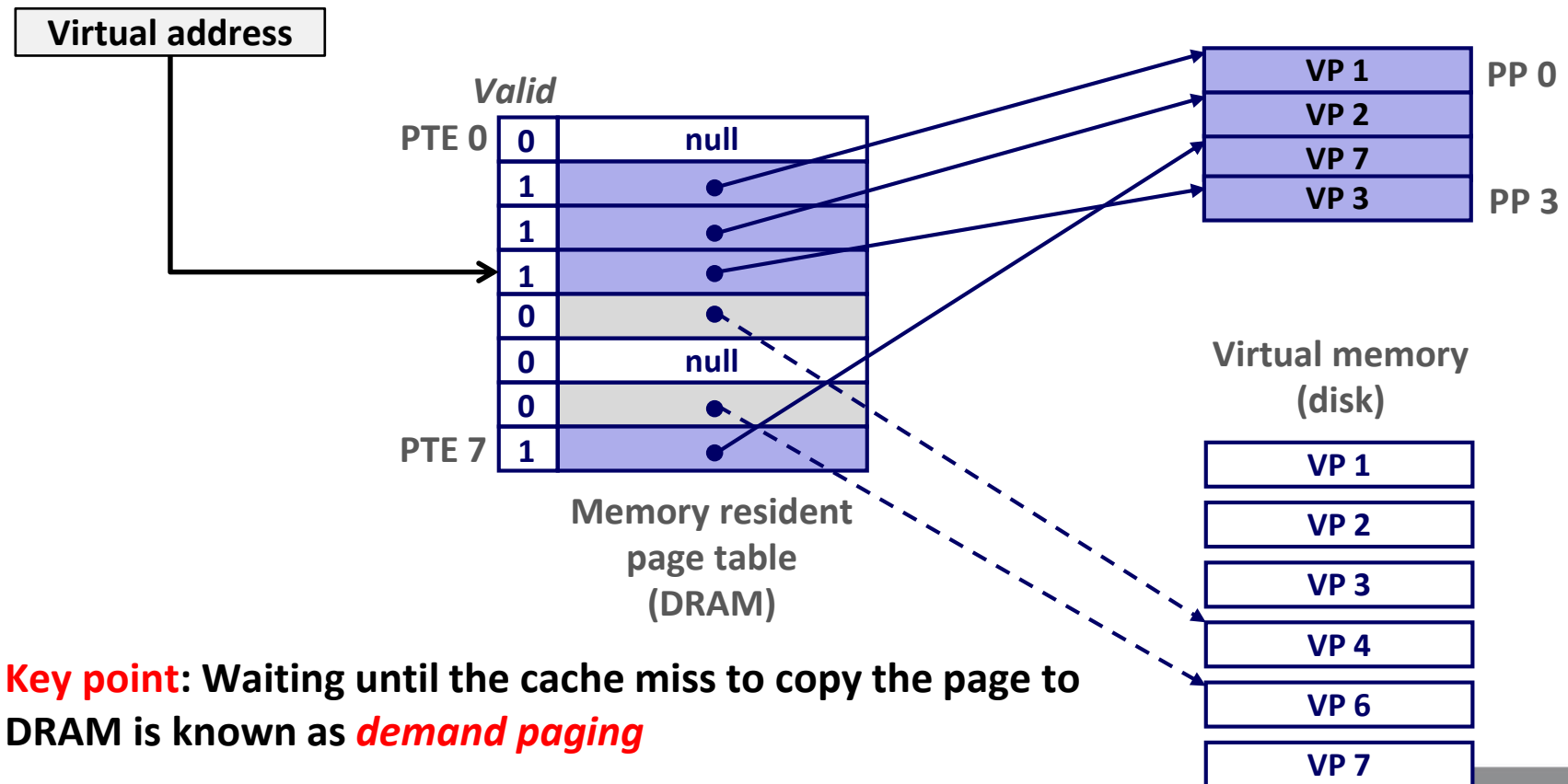
Page Fault (DRAM Cache Miss)

- Page miss causes page fault (an exception).
- Page fault handler selects a **victim** to be **evicted** (here VP 4).



Page Fault (DRAM Cache Miss)

- Now, VP 3 is cached and VP 4 is evicted.
- Exception-causing instruction is restarted → page hit.

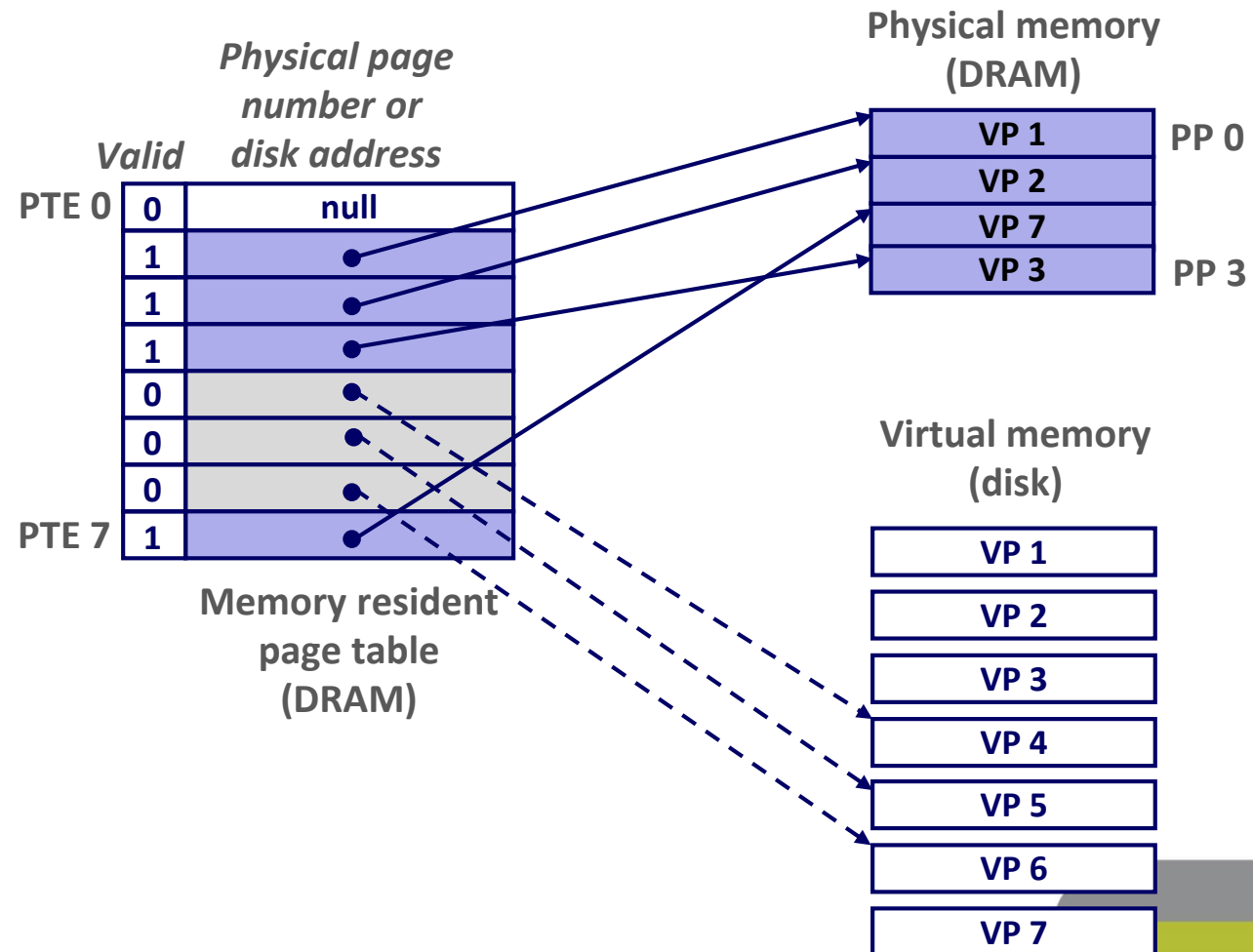


Page Fault (DRAM Cache Miss)

- Virtual memory was invented in the 1960s, long before the widening CPU-memory gap spawned SRAM caches.
- As a result, virtual memory systems use a different terminology from SRAM caches, even though many of the ideas are similar.
- In virtual memory terminology, **blocks** are known as **pages**.
- Transferring a page between disk and memory is known as **swapping or paging**.
- Pages are **swapped in (paged in)** from disk to DRAM, and **swapped out (paged out)** from DRAM to disk.
- The strategy of waiting until the last moment to swap in a page, when a miss occurs, is known as **demand paging**.

Allocating Pages

- Allocating a new page (VP 5) of virtual memory, for example, as a result of calling **malloc()** or **calloc()**.





DOUGLAS COLLEGE

Virtual Memory and Locality

Locality in the Context of Virtual Memory

- Virtual memory seems terribly inefficient, but it works because of locality.
- Although the total number of distinct pages that programs reference during an entire run might exceed the total size of physical memory, the principle of locality promises that at any point in time, a program will tend to work on a smaller set of active pages known as the **working set** or **resident set**.
- After an initial overhead where the working set is paged into DRAM, subsequent references to the working set result in hits, with no additional disk traffic.
- As long as our programs have good temporal locality, virtual memory systems work well. But of course, not all programs exhibit good temporal locality.

Locality in the Context of Virtual Memory

- If the working set size exceeds the size of physical memory, then the program can produce an unfortunate phenomenon known as **thrashing**, where pages are swapped in and out continuously.
- **If (working set size < main memory size)**
 - Good performance for one process after compulsory misses.
- **If (SUM(working set sizes) > main memory size)**
 - Thrashing: Performance meltdown where pages are swapped (copied) in and out continuously.

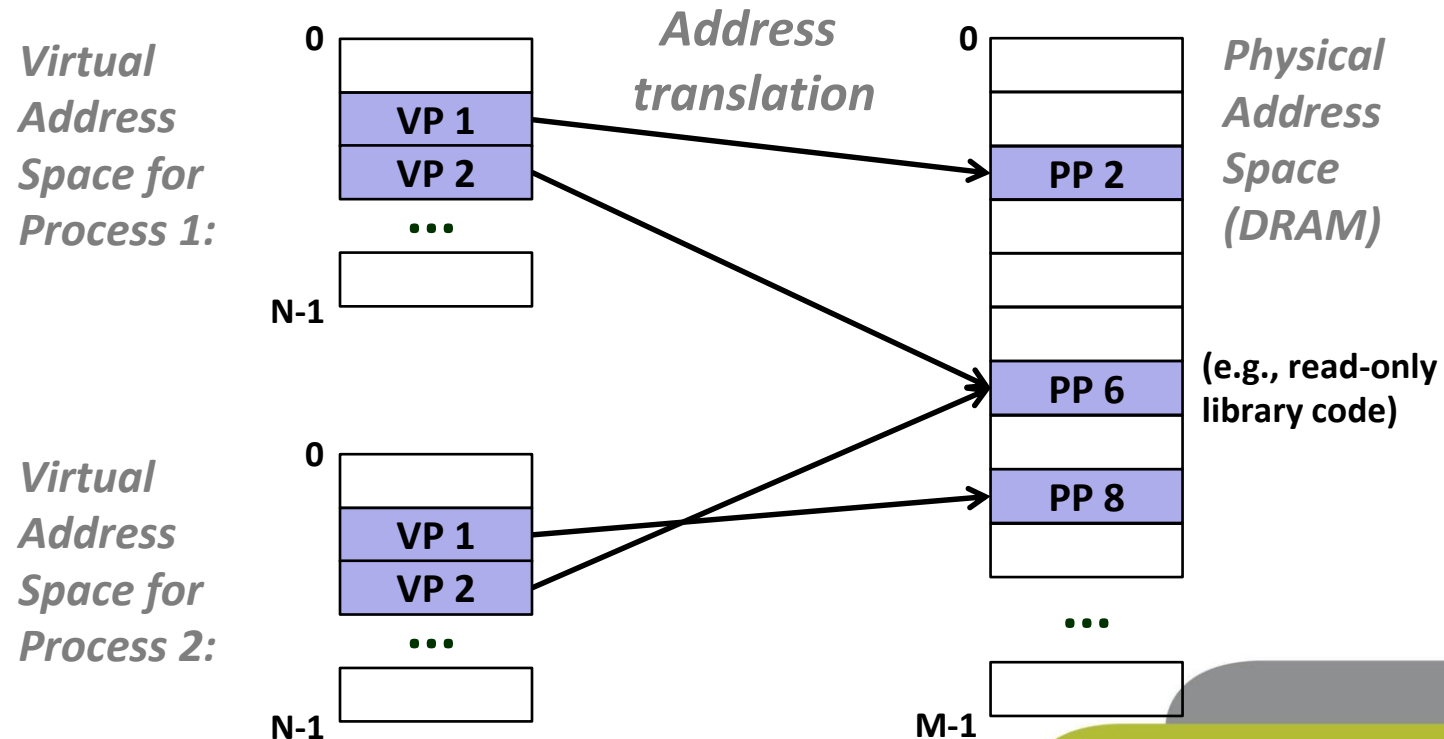
Virtual Memory: A Tool for Memory Management

VM as a Tool for Memory Management

- We have seen how virtual memory provides a mechanism for using the DRAM to cache pages from a typically larger virtual address space.
- However, some early systems supported a virtual address space that was smaller than the available DRAM.
- Yet virtual memory was still a useful mechanism because it greatly simplified memory management and provided a natural way to protect memory.
- We have so far assumed that there is a single page table in DRAM that maps a single virtual address space to the physical address space.
- In fact, operating systems provide a separate page table, and thus a separate virtual address space, **for each process**.

VM as a Tool for Memory Management

- **Key idea: each process has its own virtual address space**
 - It can view memory as a simple linear array
 - Mapping function scatters addresses through DRAM
 - Well-chosen mappings can improve locality



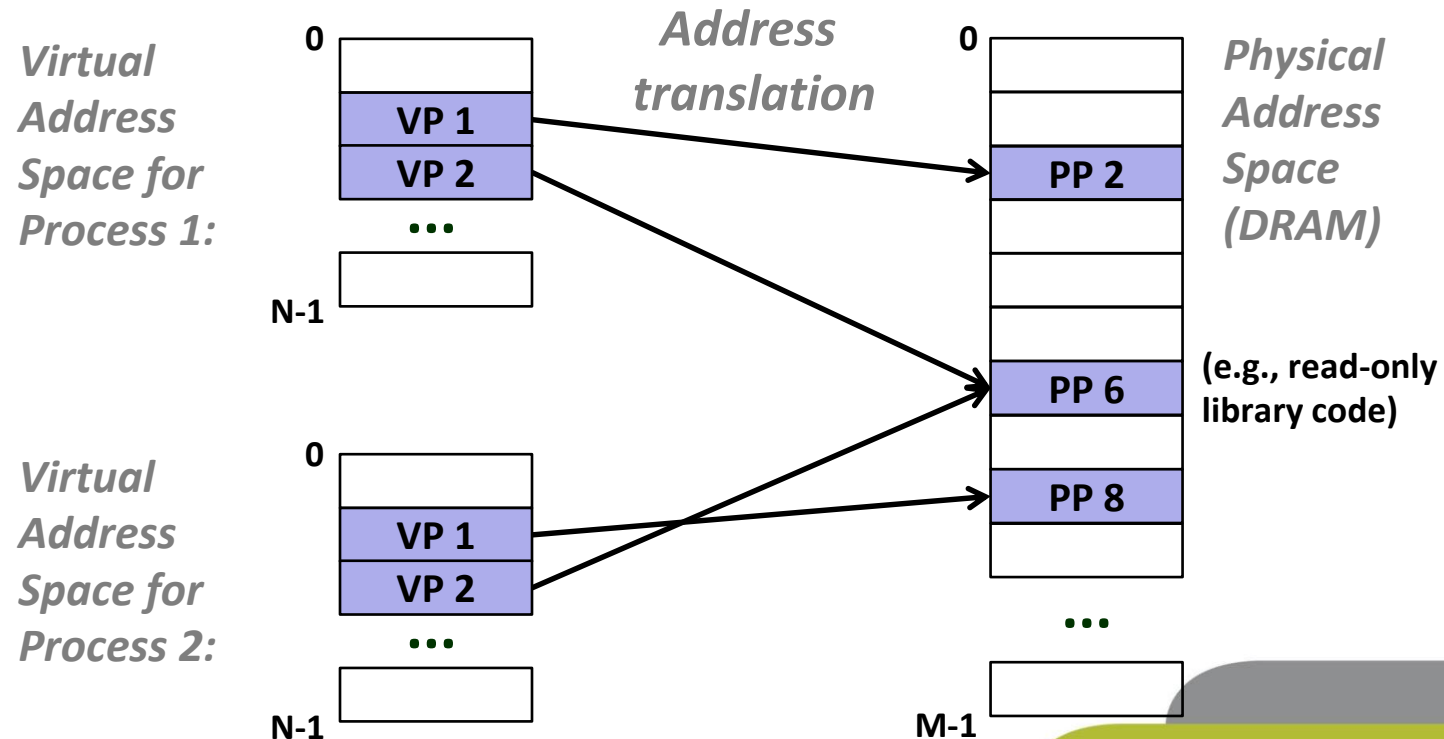
VM as a Tool for Memory Management

■ Simplifying memory allocation

- Each virtual page can be mapped to any physical page
- A virtual page can be stored in different physical pages at different times

■ Sharing code and data among processes

- Map virtual pages to the same physical page (here: PP 6)



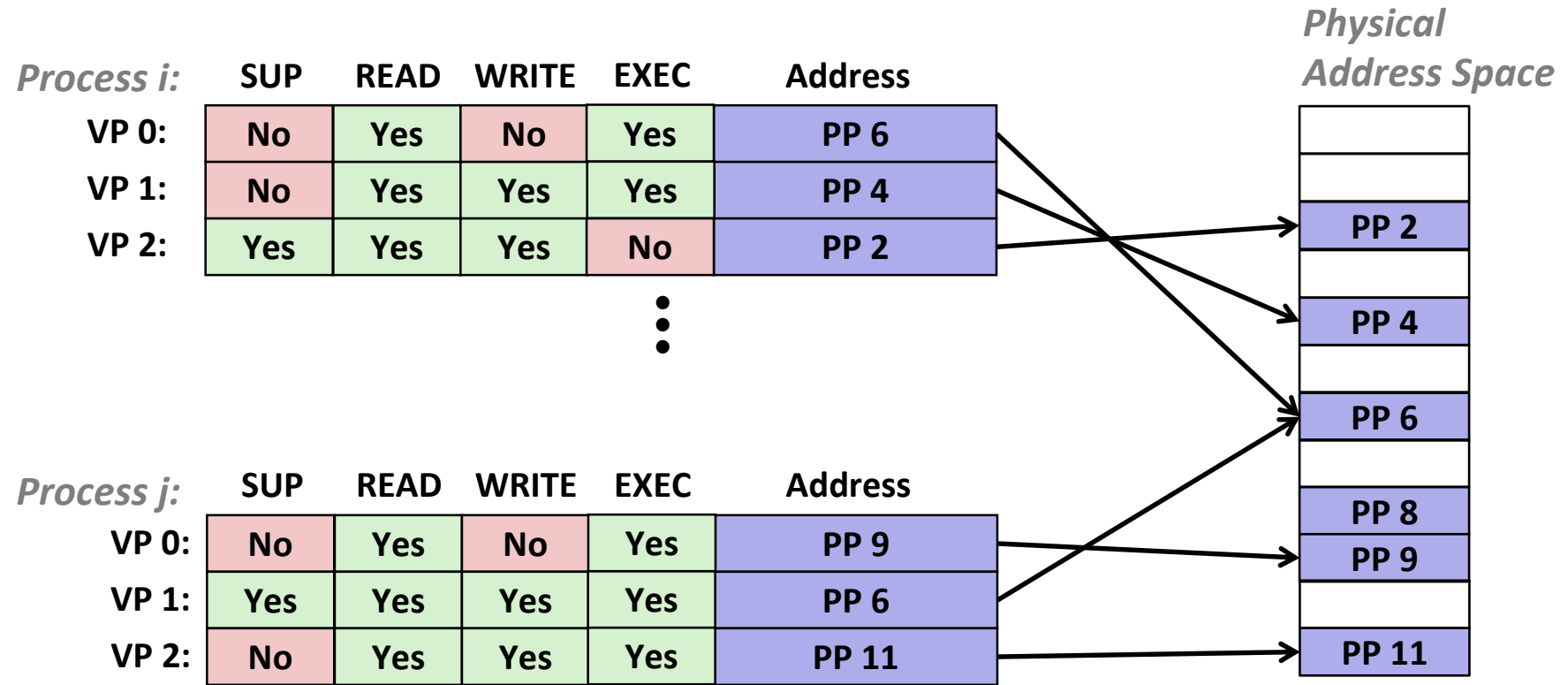
Virtual Memory: A Tool for Memory Protection

VM as a Tool for Memory Protection

- Any modern computer system must provide the means for the operating system to control access to the memory system.
- A user process should not be allowed to modify its read-only code section.
- Nor should it be allowed to read or modify any of the code and data structures in the **kernel** (the portion of the operating system code that is always resident in memory).
- It should not be allowed to read or write the private memory of other processes.
- It should not be allowed to modify any virtual pages that are shared with other processes, unless all parties explicitly allow it.
- Providing separate virtual address spaces makes it easy to isolate the private memories of different processes.

VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- MMU checks these bits on each access



VM as a Tool for Memory Protection

- In our example, there are three permission bits to each PTE.
 - **The SUP bit** indicates whether processes must be running in kernel (supervisor) mode to access the page.
 - Processes running in kernel mode can access any page, but processes running in user mode are only allowed to access pages for which SUP is 0.
 - **The READ and WRITE bits** control read and write access to the page. For example, if process i is running in user mode, then it has permission to read VP 0 and to read or write VP 1. However, it is not allowed to access VP 2.
- If an instruction violates these permissions, then the CPU triggers a general protection fault that transfers control to an exception handler in the kernel, which sends a SIGSEGV signal to the offending process.
- Linux shells typically report this exception as a "**segmentation fault**."

Virtual Memory: Address Translation

VM Address Translation

■ Virtual Address Space

- $V = \{0, 1, \dots, N-1\}$

■ Physical Address Space

- $P = \{0, 1, \dots, M-1\}$

■ Address Translation

- $MAP: V \rightarrow P \cup \{\emptyset\}$

- For virtual address a :

- $MAP(a) = a'$ if data at virtual address a is at physical address a' in P

- $MAP(a) = \emptyset$ if data at virtual address a is not in physical memory

- Either invalid or stored on disk

VM Address Translation Parameters and Symbols

■ Basic Parameters

- $N = 2^n$: Number of addresses in virtual address space
- $M = 2^m$: Number of addresses in physical address space
- $P = 2^p$: Page size (bytes)

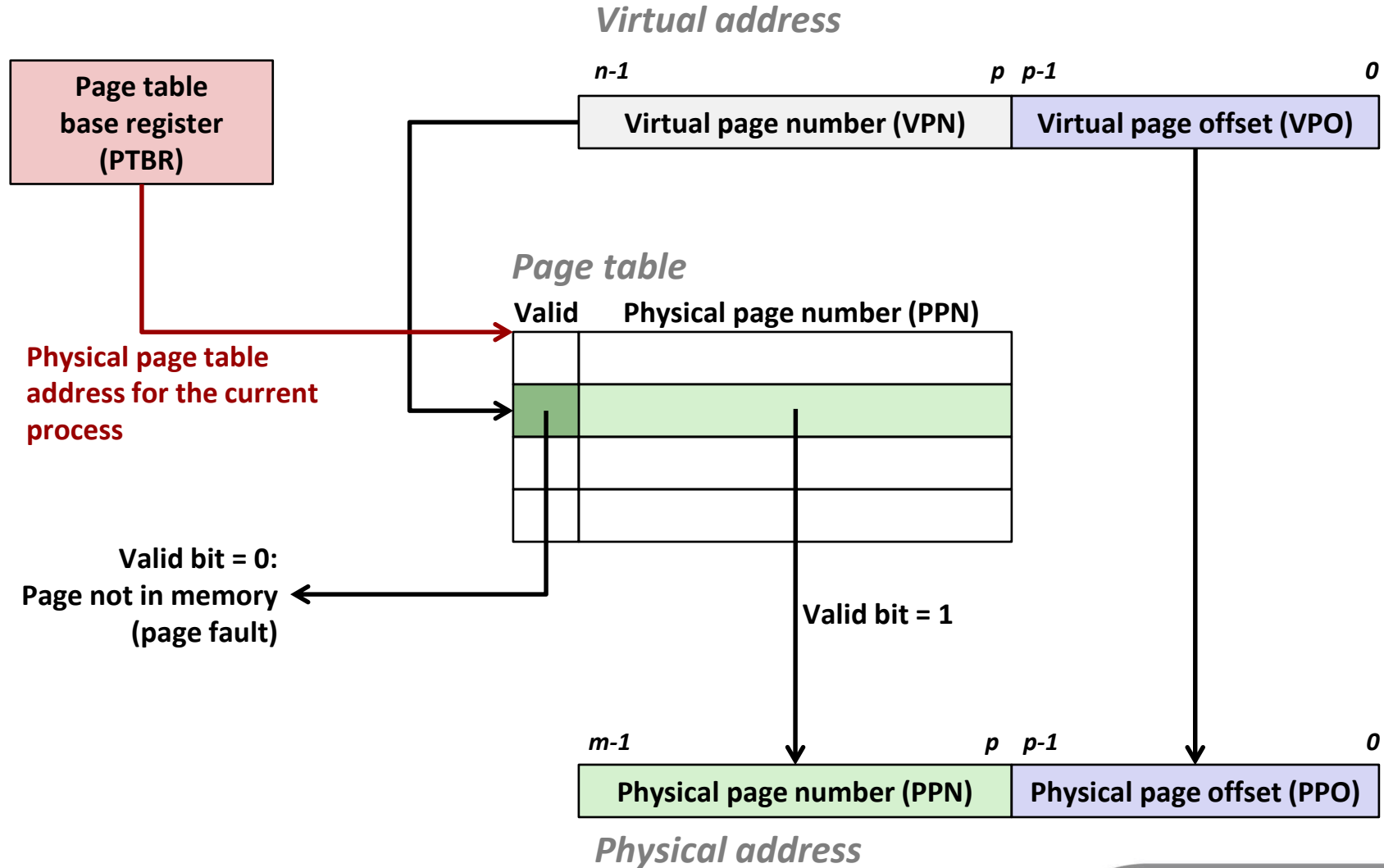
■ Components of the virtual address (VA)

- **VPO**: Virtual page offset
- **VPN**: Virtual page number

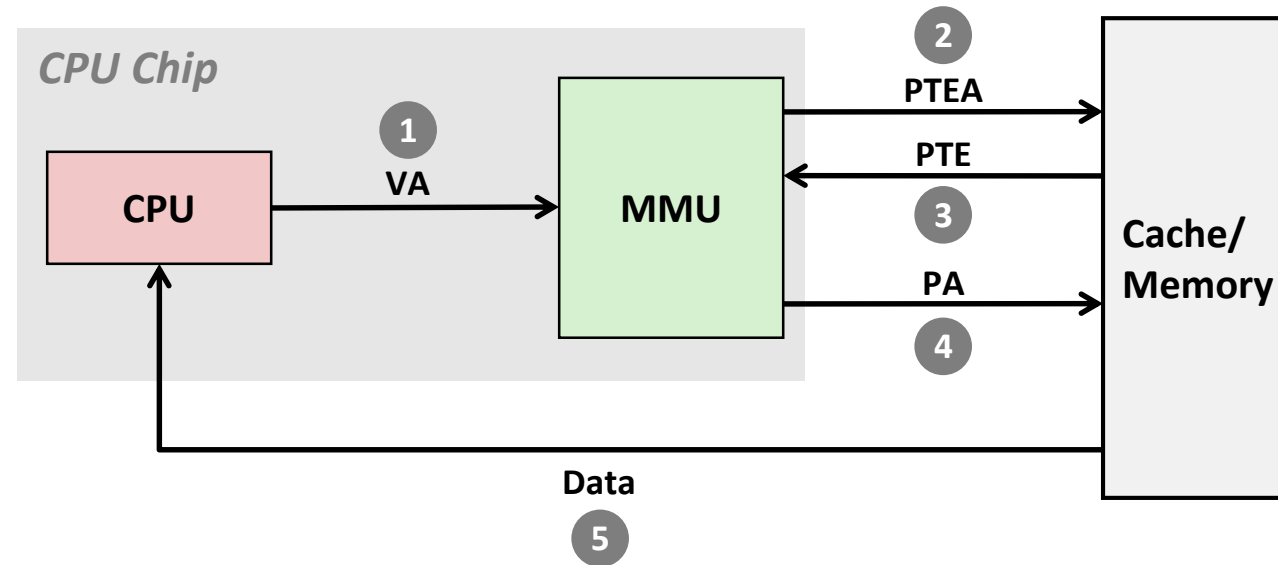
■ Components of the physical address (PA)

- **PPO**: Physical page offset (same as VPO)
- **PPN**: Physical page number

VM Address Translation Using a Page Table

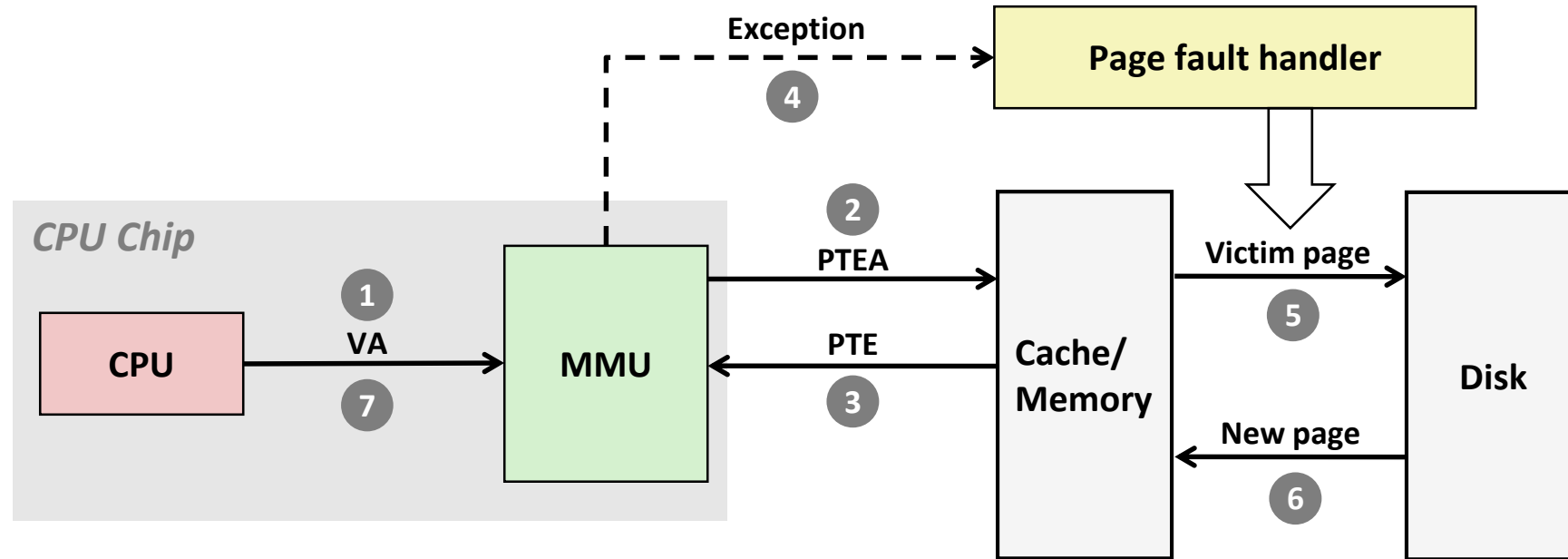


VM Address Translation: Page Hit



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) MMU sends physical address to cache/memory
- 5) Cache/memory sends data word to processor

VM Address Translation: Page Fault



- 1) Processor sends virtual address to MMU
- 2-3) MMU fetches PTE from page table in memory
- 4) Valid bit is zero, so MMU triggers page fault exception
- 5) Handler identifies victim (and, if dirty, pages it out to disk)
- 6) Handler pages in new page and updates PTE in memory
- 7) Handler returns to original process, restarting faulting instruction

Multi-Level Page Tables

■ Suppose:

- 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE

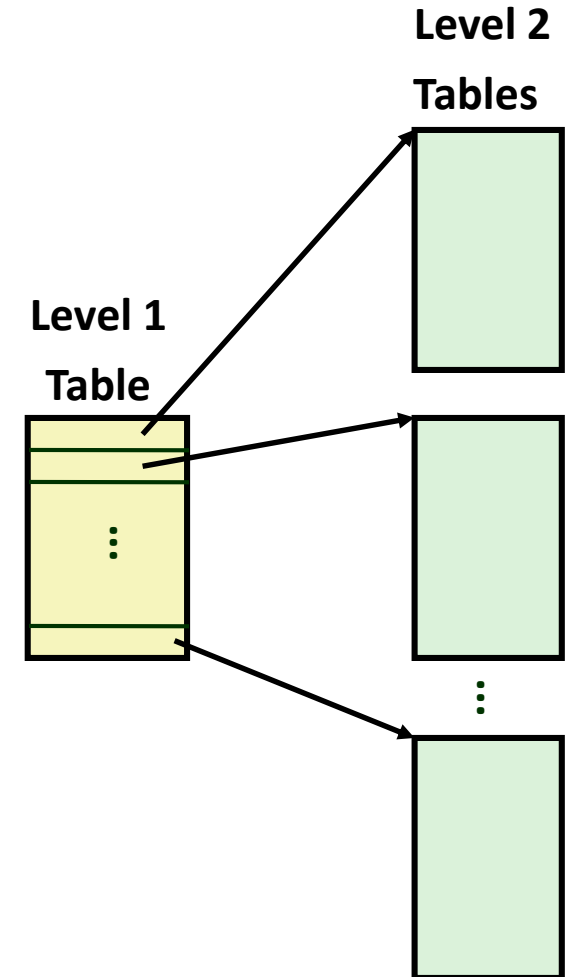
■ Problem:

- Would need a 512 GB page table!
 - $2^{48} * 2^{-12} * 2^3 = 2^{39}$ bytes

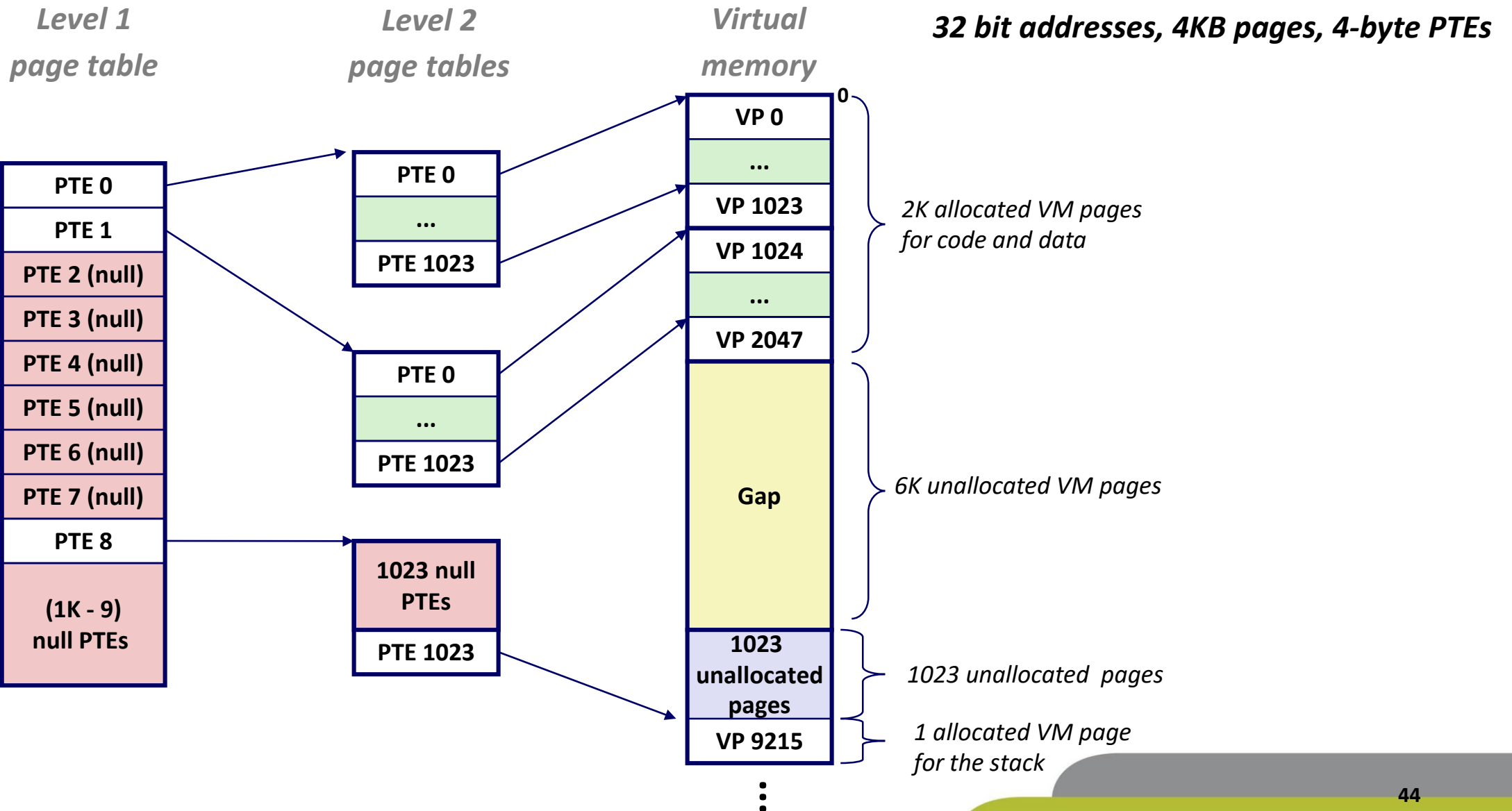
■ Common solution: Multi-level page table

■ Example: 2-level page table

- Level 1 table: each PTE points to a page table (always memory resident)
- Level 2 table: each PTE points to a page (paged in and out like any other data)



Multi-Level Page Tables : Example



VM Address Translation Parameters and Symbols

- Each PTE in the level 1 table is responsible for mapping a 4 MB chunk of the virtual address space, where each chunk consists of 1,024 contiguous pages.
- If every page in chunk *i* is unallocated, then level 1 PTE *i* is null.
- If at least one page in chunk *i* is allocated, then level 1 PTE *i* points to the base of a level 2 page table.
- Each PTE in a level 2 page table is responsible for mapping a 4-KB page of virtual memory, just as before when we looked at single-level page tables.
- If a PTE in the level 1 table is null, then the corresponding level 2 page table does not even have to exist → potential savings, since most of the 4 GB virtual address space for a typical program is unallocated.
- Second, only the level 1 table needs to be in main memory at all times.
- The level 2 page tables can be created and paged in and out by the VM system as they are needed.

References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2395

Intro to Computer Architecture

Lecture 9

Plan for Today

- History of Intel processors and architectures
- The C Programming Language and Assembly
- Our First Assembly
- Registers and The Assembly Level of Abstraction
- The **mov** instruction
- Operand Forms for the **mov** instruction
- The **lea** instruction

History of Intel Processors and Architectures

Intel x86 Processors

- **Dominate laptop/desktop/server market**
- **Evolutionary design**
 - Backwards compatible up until 8086, introduced in 1978
 - Added more features as time goes on
- The list that follows shows some models of Intel processors and some of their key features.
- **We use the number of transistors required to implement the processors as an indication of how they have evolved in complexity.**

Intel x86 Evolution

<i>Name</i>	<i>Date</i>	<i>Transistors</i>	<i>MHz</i>
■ 8086	1978	29K	5-10
■ First 16-bit Intel processor. Basis for IBM PC & DOS ■ 1MB address space			
■ 386	1985	275K	16-33
■ First 32 bit Intel processor, referred to as IA32 ■ Added “flat addressing”, capable of running Unix			
■ Pentium 4E	2004	125M	2800-3800
■ First 64-bit Intel x86 processor, referred to as x86-64			
■ Core 2	2006	291M	1060-3500
■ First multi-core Intel processor			
■ Core i7	2008	731M	1700-3900
■ Four cores			

Intel x86 Evolution

■ Machine Evolution

■ i386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2001	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M
■ Core i9	2017	???

■ Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores

State of the Art (2022)

- **Core i9 Alder Lake (Q1 2022)**

- **Desktop Model:**

- 8 cores
- Integrated graphics
- 3.4-5.3 GHz
- 150W – 241W

x86 Competitors: 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: serious competitor to Pentium 4.
- Developed x86-64, their own extension to 64 bits.

■ Recent Years

- Intel caught up again!
 - Leads the world in semiconductor technology.
- AMD has fallen behind.
 - Relies on external semiconductor manufacturer.

Intel's 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
 - Totally different architecture (Itanium).
 - Executes IA32 code only as legacy.
 - Performance was disappointing.
- **2003: AMD Steps in with Evolutionary Solution**
 - x86-64 (now called “AMD64”).
- **Intel Felt Obligated to Focus on IA64**
 - Hard to admit mistake or that AMD is better.
- **2004: Intel Announces EM64T extension to IA32**
 - Extended Memory 64-bit Technology.
 - Almost identical to x86-64!
- **All but low-end x86 processors support x86-64**
 - But, lots of code still runs in 32-bit mode.

The C Programming Language and Assembly

GCC and Assembly

- C, and other languages, are high-level abstractions we use to write code efficiently.
- But computers do not understand data structures, variable types, etc. Compilers are the translator.
- Pure machine code is comprised of 1s and 0s – everything is bits, even our programs!
- However, we can read it in a human-readable form called **assembly**. (Engineers used to write code in assembly before C).
- There may be multiple assembly instructions needed to encode a single C instruction.
- We will attempt to behind the scenes to see what the assembly code for our programs looks like.

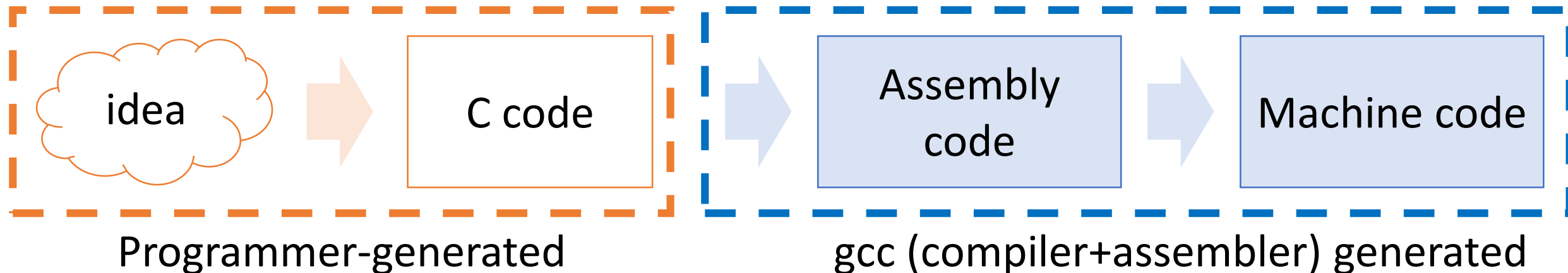
GCC and Assembly

- GCC compiles our program – it lays out memory on the stack and heap and generates assembly instructions to access and do calculations on those memory locations.
- Below is what the “assembly-level abstraction” of C code might look like:

C	Assembly Abstraction
int sum = x + y;	1) Copy x into register 1 2) Copy y into register 2 3) Add register 2 to register 1 4) Write register 1 to memory for sum

Why Are We Learning to Read Assembly?

- **Main goal:**
 - Information retrieval.
- We will not be writing assembly code because that is the compiler's job!
- Instead, we would like to translate the assembly back into our C code.
- Knowing how our C code is converted into machine instructions will hopefully give us insight into how to write more efficient code.



Examining Assembly with objdump -d

Our First Assembly

```
int sum_array(int arr[], int nelems) {  
    int sum = 0;  
    for (int i = 0; i < nelems; i++) {  
        sum += arr[i];  
    }  
    return sum;  
}
```

What does this look like in assembly?

Our First Assembly

```
int sum_array(int arr[], int nelems) {  
    int sum = 0;  
    for (int i = 0; i < nelems; i++) {  
        sum += arr[i];  
    }  
    return sum;  
}
```

```
gcc -g -Og -std=gnu99 -Wall -fcf-protection=none -fno-pic -no-pie example_00.c -o example_00.exe  
objdump.exe -d example_00.exe > asm.txt
```


Our First Assembly

0000000100401080 <sum_array>:

100401080:	49 89 c9	mov	%rcx,%r9
100401083:	41 89 d0	mov	%edx,%r8d
100401086:	b8 00 00 00 00	mov	\$0x0,%eax
10040108b:	ba 00 00 00 00	mov	\$0x0,%edx
100401090:	eb 0a	jmp	10040109c <sum_array+0x1c>
100401092:	48 63 c8	movslq	%eax,%rcx
100401095:	41 03 14 89	add	(%r9,%rcx,4),%edx
100401099:	83 c0 01	add	\$0x1,%eax
10040109c:	44 39 c0	cmp	%r8d,%eax
10040109f:	7c f1	j1	100401092 <sum_array+0x12>
1004010a1:	89 d0	mov	%edx,%eax
1004010a3:	c3	ret	

Our First Assembly

0000000100401080 <sum_array>:

100401080: 49 89 c9

This is the name of the function (same as C) and the memory address where the code for this function starts.

100401092: 48 63 c8

100401095: 41 03 14 89

100401099: 83 c0 01

10040109c: 44 39 c0

10040109f: 7c f1

1004010a1: 89 d0

1004010a3: c3

```

mov    %rcx,%r9
mov    %edx,%r8d
mov    $0x0,%eax
mov    $0x0,%edx
jmp     10040109c <sum_array+0x1c>
movslq %eax,%rcx
add     (%r9,%rcx,4),%edx
add     $0x1,%eax
cmp     %r8d,%eax
jl      100401092 <sum_array+0x12>
mov     %edx,%eax
ret

```

Our First Assembly

0000000100401080 <sum_array>:

100401080:	49 89 c9	mov	%rcx,%r9
100401083:	41 89 d0	mov	%edx,%r8d
100401086:	18 00 00 00 00	mov	\$0x0,%eax
10040108b:	1a 00 00 00 00	mov	\$0x0,%edx
100401090:	eb 0a	mov	0xa,%ecx
100401092:	48 63	mov	0x63,%eax
100401095:	41 03	mov	0x3,%eax
100401099:	83 c0	cmpl	%eax,%ecx
10040109c:	44 39 c0	cmpl	%eax,%ecx
10040109f:	7c f1	jle	100401092 <sum_array+0x12>
1004010a1:	89 d0	mov	%edx,%eax
1004010a3:	13	ret	

These are the memory addresses where each of the instructions live. Sequential instructions are sequential in memory.

Our First Assembly

0000000100401080 <sum_array>:

```
100401080: 49 89 c9
100401083: 41 89 d0
100401086: b8 00 00 00 00
10040108b: ba 00 00 00 00
```

This is the assembly code:
“human-readable” versions of
each machine code instruction.

```
10040109c: 44 39 c0
10040109f: 7c f1
1004010a1: 89 d0
1004010a3: c3
```

```
mov    %rcx,%r9
mov    %edx,%r8d
mov    $0x0,%eax
mov    $0x0,%edx
jmp    10040109c <sum_array+0x1c>
movslq %eax,%rcx
add    (%r9,%rcx,4),%edx
add    $0x1,%eax
cmp    %r8d,%eax
jl     100401092 <sum_array+0x12>
mov    %edx,%eax
ret
```

Our First Assembly

0000000100401080 <sum_array>:

```

100401080: 49 89 c9
100401083: 41 89 d0
100401086: b8 00 00 00 00
10040108b: ba 00 00 00 00
100401090: eb 0a
100401092: 48 63 c8
100401095: 41 03 14 89
100401099: 83 c0 01
10040109c: 44 39 c0
10040109f: 7c f1
1004010a1: 89 d0
1004010a3: c3

```



```

mov    %rcx,%r9
mov    %edx,%r8d
mov    $0x0,%eax

```

This is the machine code: raw hexadecimal instructions, representing binary as read by the computer. Different instructions may be different byte lengths.

```

jl     100401092 <sum_array+0x12>
mov    %edx,%eax
ret

```

Our First Assembly

0000000100401080 <sum_array>:

100401080:	49 89 c9	mov	%rcx,%r9
100401083:	41 89 d0	mov	%edx,%r8d
100401086:	b8 00 00 00 00	mov	\$0x0,%eax
10040108b:	ba 00 00 00 00	mov	\$0x0,%edx
100401090:	eb 0a	jmp	10040109c <sum_array+0x1c>
100401092:	48 63 c8	movslq	%eax,%rcx
100401095:	41 03 14 89	add	(%r9,%rcx,4),%edx
100401099:	83 c0 01	add	\$0x1,%eax
10040109c:	44 39 c0	cmp	%r8d,%eax
10040109f:	7c f1	j1	100401092 <sum_array+0x12>
1004010a1:	89 d0	mov	%edx,%eax
1004010a3:	c3	ret	

Our First Assembly

0000000100401080 <sum_array>:

```

100401080: 49 89 c9
100401083: 41 89 d0
100401086: b8 00 00 00 00
10040108b: ba 00 00 00 00
100401090: eb 0a
100401092: 48 63 c8
100401095: 41 03 14 89
100401099: 83 c0 01
10040109c: 44 39 c0
10040109f: 7c f1
1004010a1: 89 d0
1004010a3: c3

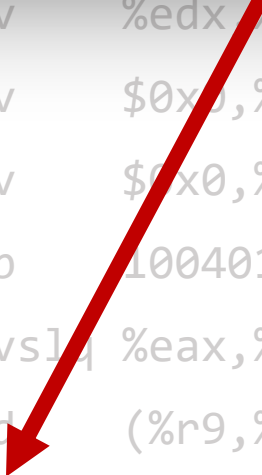
```

Each instruction has an operation name (“opcode”).

```

mov     %edx,%r8d
mov     $0x0,%eax
mov     $0x0,%edx
jmp     10040109c <sum_array+0x1c>
movslq  %eax,%rcx
add     (%r9,%rcx,4),%edx
add     $0x1,%eax
cmp     %r8d,%eax
jl      100401092 <sum_array+0x12>
mov     %edx,%eax
ret

```



Our First Assembly

0000000100401080 <sum_array>:

100401080:	49 89 c9	mov	%rcx,%r9
100401083:	41 89 d0	mov	%edx,%r8d
100401086:	b8 00 00 00 00	mov	\$0x0,%eax
10040108b:	ba 00 00 00 00	mov	\$0x0,%edx
100401090:	eb 0a	jmp	10040109c <sum_array+0x1c>
100401092:	48 63 c8	movslq	%eax,%rcx
100401095:	41 03 14 89	add	(%r9,%rcx,4),%edx
100401099:	83 c0 01	add	<u>\$0x1,%eax</u>
10040109c:	44 39 c0	cmp	%r8d,%eax
10040109f:	7c f1		
1004010a1:	89 d0		
1004010a3:	c3		

Each instruction can also have arguments (“operands”).

Our First Assembly

0000000100401080 <sum_array>:

100401080:	49 89 c9	mov	%rcx,%r9
100401083:	41 89 d0	mov	%edx,%r8d
100401086:	b8 00 00 00		
10040108b:	ba 00 00 00		
100401090:	eb 0a		
100401092:	48 63 c8	movslq	%eax,%rcx
100401095:	41 03 14 89	add	(%r9,%rcx,4),%edx
100401099:	83 c0 01	add	\$0x1,%eax
10040109c:	44 39 c0	cmp	%r8d,%eax
10040109f:	7c f1	j1	100401092 <sum_array+0x12>
1004010a1:	89 d0	mov	%edx,%eax
1004010a3:	c3	ret	

\$(number) means a constant value, or “immediate” (e.g. 1 here).

Our First Assembly

0000000100401080 <sum_array>:

100401080:	49 89 c9	mov	%rcx,%r9
100401083:	41 89 d0	mov	%edx,%r8d
100401086:	b8 00 00 00		
10040108b:	ba 00 00 00		
100401090:	eb 0a		
100401092:	48 63 c8	movslq	%eax,%rcx
100401095:	41 03 14 89	add	(%r9,%rcx,4),%edx
100401099:	83 c0 01	add	\$0x1,%eax
10040109c:	44 39 c0	cmp	%r8d,%eax
10040109f:	7c f1	j1	100401092 <sum_array+0x12>
1004010a1:	89 d0	mov	%edx,%eax
1004010a3:	c3	ret	

%[name] means a register, a storage location on the CPU (e.g. eax here).

Registers and the Assembly Level of Abstraction

Assembly Abstraction

- C abstracts away the low-level details of machine code. It lets us work using variables, variable types, and other higher-level abstractions.
- C and other languages let us write code that works on most machines.
- Assembly code is just bytes. No variable types, no type checking, etc.
- Assembly/machine code is processor-specific.
- **What is the level of abstraction for assembly code?**

Registers



%rax



%rsi



%r8



%r12



%rbx



%rdi



%r9



%r13



%rcx



%rbp



%r10



%r14



%rdx



%rsp



%r11



%r15

Registers

What is a register?

A register is a fast read/write memory slot right on the CPU that can hold variable values.

Registers are **not** located in memory.

Registers

- A register is a 64-bit space inside the processor.
- There are 16 registers available, each with a unique name.
- Registers are like “scratch paper” for the processor. Data being calculated or manipulated is moved to registers first. Operations are performed on registers.
- Registers also hold parameters and return values for functions.
- Registers are extremely fast memory.
- Processor instructions consist mostly of moving data into/out of registers and performing arithmetic on them.

x86-64 Registers

- An x86-64 CPU contains a set of 16 general-purpose registers storing 64-bit values.
- These registers are used **to store integer data as well as pointers**.
- Their names all begin with %r, but otherwise follow multiple different naming conventions, owing to the historical evolution of the **instruction set**.

64-bit register	32-bit sub-register	16-bit sub-register	8-bit sub-register
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

x86-64 Registers

- The original 8086 had eight 16-bit registers, **%ax** through **%bp**.
- Each had a specific purpose, and hence they were given names that reflected how they were to be used.
- With the extension to IA32, these registers were expanded to 32-bit registers, labeled **%eax** through **%ebp**.
- In the extension to x86-64, the original eight registers were expanded to 64 bits, labeled **%rax** through **%rbp**.
- In addition, eight new registers were added, and these were given labels according to a new naming convention: **%r8** through **%r15**.

64-bit register	32-bit sub-register	16-bit sub-register	8-bit sub-register
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

8086 Registers

- As the nested boxes indicate, instructions can operate on data of different sizes stored in the low-order bytes of the 16 registers.
- Byte-level operations can access the least significant byte, 16-bit operations can access the least significant 2 bytes, 32-bit operations can access the least significant 4 bytes, and 64-bit operations can access entire registers

64-bit register	32-bit sub-register	16-bit sub-register	8-bit sub-register
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

8086 Registers

- Different registers serve different roles in typical programs.
- Most unique among them is the stack pointer, **%rsp**, used to indicate the end position in the run-time stack.
- Some instructions specifically read and write this register.
- The other 15 registers have more flexibility in their uses. A small number of instructions make specific use of certain registers.

64-bit register	32-bit sub-register	16-bit sub-register	8-bit sub-register
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

The mov Instruction

mov

- The **mov** instruction copies bytes from one place to another; it is similar to the assignment operator (=) in C.

mov src, dst

- The **src** and **dst** can each be one of:

- Immediate (constant value, like a number) (*only src*)

\$0x104

- Register

%rbx

- Memory Location
(*at most one of src, dst*)

Direct address **0x6005c0**

Operand Forms: Immediate

mov \$0x104, _____



*Copy the value 0x104
into some
destination.*

Operand Forms: Registers

mov

%rbx, _____

*Copy the value in
register %rbx into
some destination.*



mov

_____, %rbx

*Copy the value from
some source into
register %rbx.*



In-Class Practice #1

What are the results of the following move instructions (executed separately)? For this problem, assume the value **5** is stored at address **0x42**, and the value **8** is stored in **%rbx**.

1. `mov $0x42,%rax`

2. `mov 0x42,%rax`

3. `mov %rbx,0x55`

Operand Forms: Indirect

mov

(%rbx), _____

Copy the value at the address stored in register %rbx into some destination.

mov

_____, (%rbx)

Copy the value from some source into the memory at the address stored in register %rbx.

Operand Forms: Base + Displacement

mov

$0x10(\%rax),$ _____

*Copy the value at the address
($0x10$ plus what is stored in
register $\%rax$) into some
destination.*

mov

_____, $0x10(\%rax)$

*Copy the value from some source into
the memory at the address ($0x10$ plus
what is stored in register $\%rax$).*

Operand Forms: Indexed

mov

(%rax,%rdx), _____

Copy the value at the address which is (the sum of the values in registers %rax and %rdx) into some destination.

mov

_____, (%rax,%rdx)

Copy the value from some source into the memory at the address which is (the sum of the values in registers %rax and %rdx).

Operand Forms: Indexed

Copy the value at the address which is (the sum of 0x10 plus the values in registers %rax and %rdx) into some destination.

mov **0x10(%rax,%rdx), _____**

mov **_____, 0x10(%rax,%rdx)**

Copy the value from some source into the memory at the address which is (the sum of 0x10 plus the values in registers %rax and %rdx).

In-Class Practice #2

What are the results of the following move instructions (executed separately)? For this problem, assume the value *0x11* is stored at address *0x10C*, *0xAB* is stored at address *0x104*, *0x100* is stored in register *%rax* and *0x3* is stored in *%rdx*.

1. `mov $0x42, (%rax)`
2. `mov 4(%rax), %rcx`
3. `mov 9(%rax, %rdx), %rcx`

$\text{Imm}(r_b, r_i)$ is equivalent to address $\text{Imm} + R[r_b] + R[r_i]$

Displacement: positive or negative constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Operand Forms: Scaled Indexed

Copy the value at the address which is (4 times the value in register %rdx) into some destination.

mov (**,%rdx,4**), _____

mov _____, (**,%rdx,4**)

The *scaling factor* (e.g. 4 here) must be hardcoded to be either 1, 2, 4 or 8.

Copy the value from some source into the memory at the address which is (4 times the value in register %rdx).

Operand Forms: Scaled Indexed

Copy the value at the address which is (4 times the value in register %rdx, plus 0x4), into some destination.

mov

0x4(, %rdx, 4), _____

mov

_____, 0x4(, %rdx, 4)

Copy the value from some source into the memory at the address which is (4 times the value in register %rdx, plus 0x4).

Operand Forms: Scaled Indexed

Copy the value at the address which is (the value in register %rax plus 2 times the value in register %rdx) into some destination.

mov

(%rax,%rdx,2), _____

mov

_____, (%rax,%rdx,2)

Copy the value from some source into the memory at the address which is (the value in register %rax plus 2 times the value in register %rdx).

Operand Forms: Scaled Indexed

Copy the value at the address which is (0x4 plus the value in register %rax plus 2 times the value in register %rdx) into some destination.

mov

0x4(%rax,%rdx,2), _____

mov

_____, 0x4(%rax,%rdx,2)

Copy the value from some source into the memory at the address which is (0x4 plus the value in register %rax plus 2 times the value in register %rdx).

Most General Operand Form

$\text{Imm}(r_b, r_i, s)$

is equivalent to...

$\text{Imm} + R[r_b] + R[r_i] * s$

Most General Operand Form

$\text{Imm}(r_b, r_i, s)$ is equivalent to
address **$\text{Imm} + R[r_b] + R[r_i]*s$**

Displacement:
pos/neg constant
(if missing, = 0)

Base: register (if
missing, = 0)

Index: register
(if missing, = 0)

Scale must be
1,2,4, or 8
(if missing, = 1)

Summary of Operand Forms

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

In-Class Practice #3

What are the results of the following move instructions (executed separately)? For this problem, assume the value *0x1* is stored in register *%rcx*, the value *0x100* is stored in register *%rax*, the value *0x3* is stored in register *%rdx*, and value *0x11* is stored at address *0x10C*.

1. **mov \$0x42,0xfc(,%rcx,4)**

2. **mov (%rax,%rdx,4),%rbx**

Imm(*r_b*, *r_i*, *s*) is equivalent to
address **Imm** + **R[*r_b*]** + **R[*r_i*]**s***
 Displacement Base Index Scale
 (1,2,4,8)

Data Sizes

- Data sizes in assembly have slightly different terminology:
 - A **byte** is 1 byte.
 - A **word** is 2 bytes.
 - A **double word** is 4 bytes.
 - A **quad word** is 8 bytes.
- Assembly instructions can have suffixes to refer to these sizes:
 - **b** means byte
 - **w** means word
 - **l** means double word
 - **q** means quad word

Register Sizes

64-bit register	32-bit sub-register	16-bit sub-register	8-bit sub-register
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

Register Responsibilities

- Some registers take on special responsibilities during program execution.
- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

mov Instruction Variants

- **mov** can take an optional suffix (**b,w,l,q**) that specifies the size of data to move: **movb**, **movw**, **movl**, **movq**
- **mov** only updates the specific register bytes or memory locations indicated.
 - **Exception:** **movl** writing to a register will also set high order 4 bytes to 0.
- In essence:
 - Instructions that generate 1 or 2-byte quantities leave the remaining bytes unchanged.
 - Those that generate 4-byte quantities set the upper 4 bytes of the register to zero.
 - adopted as part of the expansion from IA32 to x86-64.

mov

- The **movabsq** instruction is used to write a 64-bit Immediate (constant) value.
 - The regular **movq** instruction can only take 32-bit immediates.
- 64-bit immediate as source, only register as destination.

```
movabsq $0x0011223344556677, %rax
```

In-Class Practice #3

For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. **movb**, **movw**, **movl** or **movq**).

mov__ %eax, (%rsp)

mov__ (%rax), %dx

mov__ \$0xff, %b1

mov__ (%rsp,%rdx,4), %d1

mov__ (%rdx), %rax

mov__ %dx, (%rax)

movz and movs

- There are two **mov** instructions that can be used to copy a smaller source to a larger destination: **movz** and **movs**.
 - **movz** fills the remaining bytes with zeros
 - **movs** fills the remaining bytes by sign-extending the most significant bit in the source.
- The source must be from memory or a register, and the destination is a register.

MOVZ

MOVZ S, R

$R \leftarrow \text{ZeroExtend}(S)$

Instruction	Description
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

In-Class Practice #5

Why is a **movz1q** instruction absent in the previous table?



movs

MOVS S, R

$R \leftarrow \text{SignExtend}(S)$

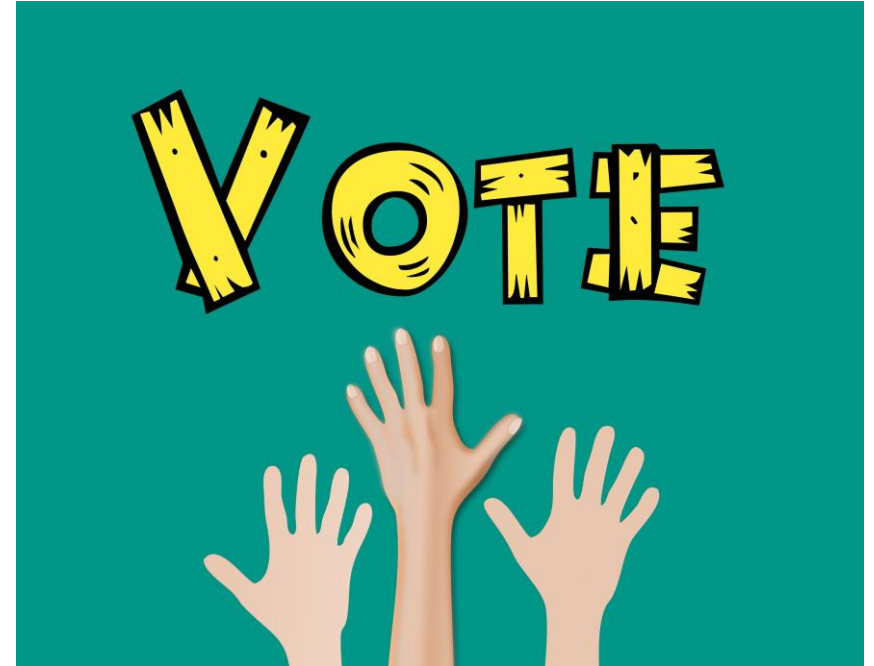
Instruction	Description
movsbw	Move sign-extended byte to word
movsbl	Move sign-extended byte to double word
movswl	Move sign-extended word to double word
movsbq	Move sign-extended byte to quad word
movswq	Move sign-extended word to quad word
movslq	Move sign-extended double word to quad word
cltq	Sign-extend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

Poll 1 (Extra Credit)

The `cltq` instruction is equivalent to which of the following?

- a) `movzq %eax, %rax`
- b) `movslq %eax, %rax`
- c) `movzq %rax, %eax`
- d) `movslq %rax, %eax`

Please use the “Poll” window to participate for extra credit!



mov Instruction Variants

- The following example illustrates how different data movement instructions either do or do not change the high-order bytes of the destination.

<code>movabsq \$0x0011223344556677, %rax</code>	<code>%rax = 0011223344556677</code>
<code>movb \$0xAA, %dl</code>	<code>%dl = AA</code>
<code>movb %dl, %al</code>	<code>%rax = 00112233445566AA</code>
<code>movsbq %dl, %rax</code>	<code>%rax = FFFFFFFFFFFFFFFFAA</code>
<code>movzbq %dl, %rax</code>	<code>%rax = 00000000000000AA</code>

movq Operand Forms

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

The lea Instruction (Load Effective Address)

The `lea` Instruction

- The **lea** instruction copies an “effective address” from one place to another.

lea **src, dst**

- Unlike **mov**, which copies data at the address **src** to the destination, **lea** copies the value of **src** *itself* to the destination.

The syntax for the destinations is the same as **mov**. The difference is in how it handles the **src**.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
6(%rax), %rdx	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
(%rax, %rcx), %rdx	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
(%rax, %rcx, 4), %rdx	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.
7(%rax, %rax, 8), %rdx	Go to the address (7 + %rax + 8 * %rax) and copy data there into %rdx.	Copy (7 + %rax + 8 * %rax) into %rdx.

Unlike **mov**, which copies data at the address src to the destination, **lea** copies the value of src *itself* to the destination.

References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2395

Intro to Computer Architecture

Lecture 10

Extra Credit Poll (1)

Why is a **movz1q** instruction absent in this table?

Instruction	Description
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

- a) Because we never have to sign-extend from double to quad word.
- b) Because movl will do that already even though it is missing the suffixes.
- c) Do not remember...



Plan for Today

- **lea** Instruction Examples
- Logical and Arithmetic Operations
- Executing Instructions using the Program Counter
- Intro to Control Mechanics in Assembly

- **lea Instruction Examples**

In-Class Practice #1

- Consider the following C function:

```
long scale(long x, long y, long z)
{
    long t = x + (4 * y) + (12 * z);
    return t;
}
```

- When compiled, the arithmetic operations are carried out by a series of **lea** instructions

0000000100401080 <scale>:

100401080:	48 8d 04 91	lea	(%rcx,%rdx,4),%rax
100401084:	4b 8d 14 40	lea	(%r8,%r8,2),%rdx
100401088:	48 8d 04 90	lea	(%rax,%rdx,4),%rax
10040108c:	c3	ret	

- Decode each of the lea instructions to explain how the compiler carries out the operation.

In-Class Practice #1

- Consider the following C function:

```
long scale(long x, long y, long z)
{
    long t = x + (4 * y) + (12 * z);
    return t;
}
```

- When compiled, the arithmetic operations are carried out by a series of **lea** instructions (**x** is in **%rcx**, **y** is in **%rdx**, and **z** is in **%r8**)

0000000100401080 <scale>:

100401080:	48 8d 04 91	lea	(%rcx,%rdx,4),%rax	# t = 4*y + x
100401084:	4b 8d 14 40	lea	(%r8,%r8,2),%rdx	# y = 2*z + Z = 3*z
100401088:	48 8d 04 90	lea	(%rax,%rdx,4),%rax	# t = 4*y + t = 12*z + 4*y + x
10040108c:	c3	ret		

In-Class Practice #2

- Consider the following C function:

```
short scale2(short x, short y, short z)
{
    short t = _____;
    return t;
}
```

- When compiled, a series of **lea** assembly instructions are generated as follow:

0000000100401080 <scale2>:

100401080:	8d 0c 49	lea	(%rcx,%rcx,2),%ecx
100401083:	8d 14 91	lea	(%rcx,%rdx,4),%edx
100401086:	42 8d 04 02	lea	(%rdx,%r8,1),%eax
10040108a:	c3	ret	

- Decode each of the lea instructions to complete the function.

In-Class Practice #2

- Consider the following C function:

```
short scale2(short x, short y, short z)
{
    short t = z + 4 * y + 3 * x;
    return t;
}
```

- When compiled, a series of **lea** assembly instructions are generated as follow:

0000000100401080 <scale2>:

100401080:	8d 0c 49	lea	(%rcx,%rcx,2),%ecx
100401083:	8d 14 91	lea	(%rcx,%rdx,4),%edx
100401086:	42 8d 04 02	lea	(%rdx,%r8,1),%eax
10040108a:	c3	ret	

- Decode each of the lea instructions to complete the function.

- **Logical and Arithmetic Operations**

Unary Instructions

The following instructions operate on a single operand (register or memory):

Instruction	Effect	Description
<code>inc D</code>	$D \leftarrow D + 1$	Increment
<code>dec D</code>	$D \leftarrow D - 1$	Decrement
<code>neg D</code>	$D \leftarrow -D$	Negate
<code>not D</code>	$D \leftarrow \sim D$	Complement

Examples:

```
incq 16(%rax)
dec %rdx
not %rcx
```

Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). **Both cannot be memory locations.** Read it as “Subtract S from D”:

Instruction	Effect	Description
add S, D	$D \leftarrow D + S$	Add
sub S, D	$D \leftarrow D - S$	Subtract
imul S, D	$D \leftarrow D * S$	Multiply
xor S, D	$D \leftarrow D \wedge S$	Exclusive-or
or S, D	$D \leftarrow D \mid S$	Or
and S, D	$D \leftarrow D \& S$	And

Examples:

```
addq %rcx, (%rax)
xorq $16, (%rax, %rdx, 8)
subq %rdx, 8(%rax)
```

Large Multiplication

Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?

If you specify two operands to **imul**, it multiplies them together and truncates until it fits in a 64-bit register.

$$\text{imul } S, D \qquad D \leftarrow D * S$$

If you specify one operand, it multiplies that by **%rax**, and splits the product across **2** registers. It puts the high-order 64 bits in **%rdx** and the low-order 64 bits in **%rax**.

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

Division and Remainder

Instruction	Effect	Description
<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
<code>cqto</code>	$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	Convert to oct word

- Terminology: **dividend / divisor = quotient + remainder**
- **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
- The high-order 64 bits of the dividend are in **%rdx**, and the low-order 64 bits are in **%rax**. The divisor is the operand to the instruction.
- The quotient is stored in **%rax**, and the remainder in **%rdx**.
- Most divisions use only 64-bit dividends. The **cqto** instruction sign-extends the 64-bit value in **%rax** into **%rdx** to fill both registers with the dividend, as the division instruction expects.

In-Class Practice #3

- Consider the following C function:

```
void remdiv(long x, long y, long* q, long* r)
{
    long quotient = x / y;
    long remainder = x % y;
    *q = quotient;
    *r = remainder;
}
```

- When compiled, we might get the following assembly:

0000000100401080 <remdiv>:

100401080:	48 89 c8	mov	%rcx,%rax
100401083:	48 89 d1	mov	%rdx,%rcx
100401086:	48 99	cqto	
100401088:	48 f7 f9	idiv	%rcx
10040108b:	49 89 00	mov	%rax, (%r8)
10040108e:	49 89 11	mov	%rdx, (%r9)
100401091:	c3	ret	

- Decode the assembly instructions to explain how the function is carrying out the division.**

In-Class Practice #3

- Consider the following C function:

```
void remdiv(long x, long y, long* q, long* r)
{
    long quotient = x / y;
    long remainder = x % y;
    *q = quotient;
    *r = remainder;
}
```

- When compiled, we might get the following assembly:

0000000100401080 <remdiv>:

100401080:	48 89 c8	mov	%rcx,%rax	#copy lower-bits of x into %rax
100401083:	48 89 d1	mov	%rdx,%rcx	#copy y into %rcx
100401086:	48 99	cqto		#sign-extend %rax into %rdx
100401088:	48 f7 f9	idiv	%rcx	#divide %rdx:%rax by %rcx
10040108b:	49 89 00	mov	%rax, (%r8)	#copy quotient into *q
10040108e:	49 89 11	mov	%rdx, (%r9)	#copy remainder into *r
100401091:	c3	ret		

- Decode the assembly instructions to explain how the function is carrying out the division.**

Shift Instructions

The following instructions have two operands: the shift amount **k** and the destination to shift, **D**. **k** can be either an immediate value, or the byte register **%cl** (**and only that register!**)

Instruction	Effect	Description
<code>sal k, D</code>	$D \leftarrow D \ll k$	Left shift
<code>shl k, D</code>	$D \leftarrow D \ll k$	Left shift (same as <code>sal</code>)
<code>sar k, D</code>	$D \leftarrow D \gg_A k$	Arithmetic right shift
<code>shr k, D</code>	$D \leftarrow D \gg_L k$	Logical right shift

Examples:

```
shll $3, (%rax)
shr1 %cl, (%rax, %rdx, 8)
sar1 $4, 8(%rax)
```

Shift Amount

Instruction	Effect	Description
<code>sal k, D</code>	$D \leftarrow D \ll k$	Left shift
<code>shl k, D</code>	$D \leftarrow D \ll k$	Left shift (same as <code>sal</code>)
<code>sar k, D</code>	$D \leftarrow D \gg_A k$	Arithmetic right shift
<code>shr k, D</code>	$D \leftarrow D \gg_L k$	Logical right shift

- When using **%cl**, the width of what you are shifting determines what portion of **%cl** is used.
- For **w** bits of data, it looks at the low-order **log₂(w)** bits of **%cl** to know how much to shift.
 - If **%cl** = 0xff, then: **shlb** shifts by 7 because it considers only the low-order $\log_2(8) = 3$ bits, which represent 7. **shlw** shifts by 15 because it considers only the low-order $\log_2(16) = 4$ bits, which represent 15.

In-Class Practice #4

- Consider the following C function:

```
long shift_left4_rightn(long x, long n)
{
    x <<= 4;
    x >>= n;
    return x;
}
```

- When compiled, we might get the following partial assembly:

0000000100401080 <shift_left4_rightn>:

100401080:	48 89 c8	mov %rcx,%rax
100401083:	48 c1 e0 04	-----
100401087:	89 d1	mov %edx,%ecx
100401089:	48 d3 f8	-----
10040108c:	c3	ret

- Fill in the blanks if x is stored in %rcx and n is stored in %edx.

In-Class Practice #4

- Consider the following C function:

```
long shift_left4_rightn(long x, long n)
{
    x <<= 4;
    x >>= n;
    return x;
}
```

- When compiled, we might get the following partial assembly:

0000000100401080 <shift_left4_rightn>:

100401080:	48 89 c8	mov	%rcx,%rax
100401083:	48 c1 e0 04	shl	\$0x4,%rax
100401087:	89 d1	mov	%edx,%ecx
100401089:	48 d3 f8	sar	%cl,%rax
10040108c:	c3	ret	

- Fill in the blanks if x is stored in %rcx and n is stored in %edx.**



DOUGLAS COLLEGE

Reverse Engineering Example

In-Class Practice #5

- Consider the following C function:

```
int add_to(int x, int arr[], int i)
{
    int sum = -----;
    sum += -----;
    return -----;
}
```

- When compiled, we might get the following partial assembly:

0000000100401080 <add_to>:

100401080:	4d 63 c0	movslq %r8d,%r8
100401083:	89 c8	mov %ecx,%eax
100401085:	42 03 04 82	add (%rdx,%r8,4),%eax
100401089:	c3	ret

- Fill in the blanks if `i` is stored in `%r8d`, `x` is stored in `%ecx`, and `arr[0]` is stored in `%rdx`.

In-Class Practice #5

- Consider the following C function:

```
int add_to(int x, int arr[], int i)
{
    int sum = x;
    sum += arr[i];
    return sum;
}
```

- When compiled, we might get the following partial assembly:

0000000100401080 <add_to>:

100401080:	4d 63 c0	movslq %r8d,%r8
100401083:	89 c8	mov %ecx,%eax
100401085:	42 03 04 82	add (%rdx,%r8,4),%eax
100401089:	c3	ret

- Fill in the blanks if `i` is stored in `%r8d`, `x` is stored in `%ecx`, and `arr[0]` is stored in `%rdx`.

Extra Credit Poll (2)

What do the following instructions do?

```
imul    %ecx,%ecx  
movslq  %ecx,%rax  
ret
```

- a. `long square(int num){
 return num * num;
}`
- b. `long square(int num) {
 return num + num;
}`
- c. `long square(int num) {
 return num * 17;
}`



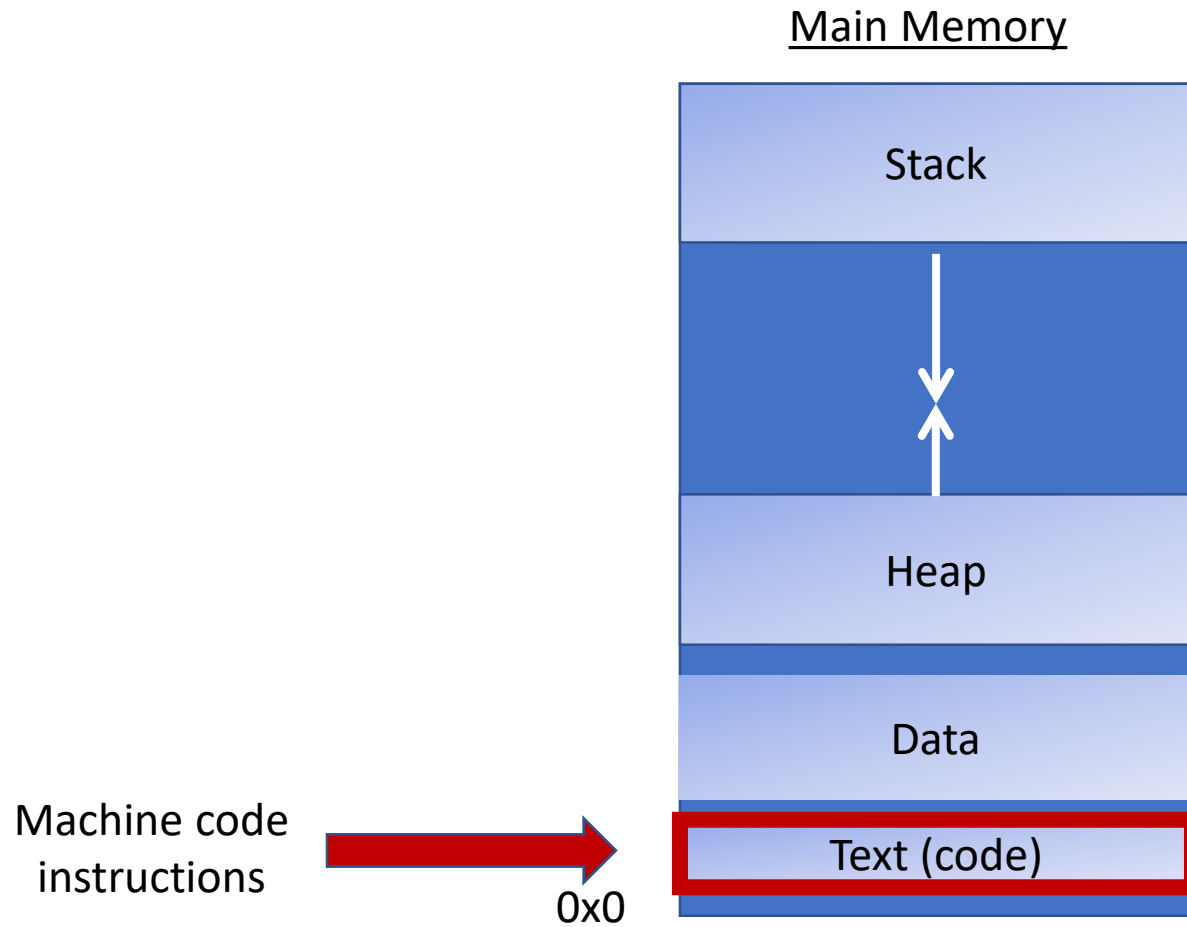
Executing Instructions using the Program Counter

Executing Instructions

So far:

- Program values can be stored in memory or registers.
- Assembly instructions read/write values back and forth between registers (on the CPU) and memory.
- Assembly instructions are also stored in memory.
- **Who controls the instructions?**
 - How do we know what to do now or next?
- **Answer:**
- The **program counter** (PC), %rip.

Instructions Are Just Bytes



Instructions Are Just Bytes

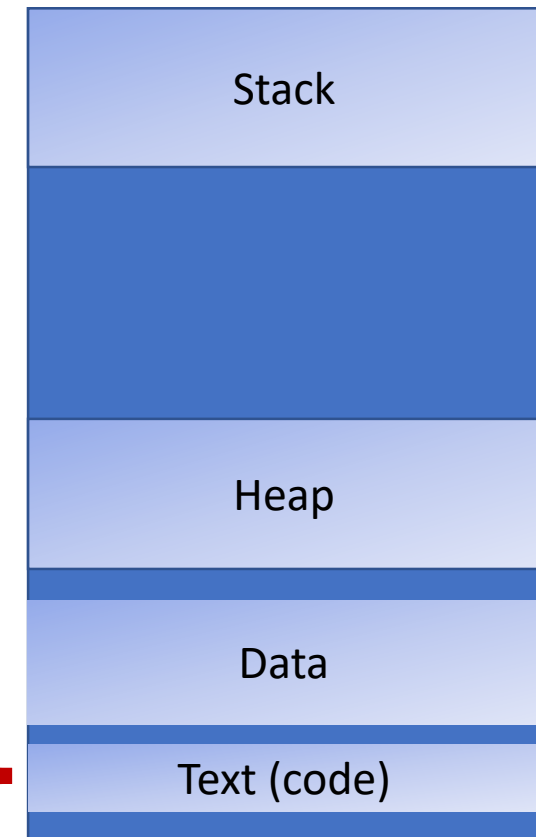
00000000004004ed <loop>:

```

4004ed: 55          push    %rbp
4004ee: 48 89 e5    mov     %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00  movl    $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01  addl    $0x1,-0x4(%rbp)
4004fc: eb fa      jmp     4004f8 <loop+0xb>
  
```

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Main Memory



%rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the *next instruction* to be executed.

0x4004ed

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the *next instruction* to be executed.

0x4004ee

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the *next instruction* to be executed.

0x4004f1

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the *next instruction* to be executed.

0x4004f8

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the *next instruction* to be executed.

0x4004fc

%rip

4004fd fa

4004fc eb

4004fb 01

4004fa fc

4004f9 45

4004f8 83

4004f7 00

4004f6 00

4004f5 00

4004f4 00

4004f3 fc

4004f2 45

4004f1 c7

4004f0 e5

4004ef 89

4004ee 48

4004ed 55

%rip

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

4004fd fa

4004fc eb

4004fb 01

4004fa fc

4004f9 45

4004f8 83

4004f7 00

4004f6 00

4004f5 00

4004f4 00

4004f3 fc

4004f2 45

4004f1 c7

4004f0 e5

4004ef 89

4004ee 48

4004ed 55

0x4004fc

%rip

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction

Jump!

- How can we use this representation of execution to represent e.g. a **loop**?
- **Key Idea:** we can “interfere” with **%rip** and set it back to an earlier instruction!

The jmp Instruction

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

```
push    %rbp
mov     %rsp,%rbp
movl    $0x0,-0x4(%rbp)
addl    $0x1,-0x4(%rbp)
jmp     4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

0x4004fc

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

The jmp Instruction

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

```
push    %rbp
mov     %rsp,%rbp
movl    $0x0,-0x4(%rbp)
addl    $0x1,-0x4(%rbp)
jmp     4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

0x4004f8

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

The jmp Instruction

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

```
push    %rbp
mov     %rsp,%rbp
movl    $0x0,-0x4(%rbp)
addl    $0x1,-0x4(%rbp)
jmp     4004f8 <loop+0xb>
```

The **jmp** instruction is an **unconditional jump** that sets the program counter to the **jump target** (the operand).

0x4004fc

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

The jmp Instruction

00000000004004ed <loop>:

4004ed: 55

4004ee: 48 89 e5

4004f1: c7 45 fc 00 00 00 00

4004f8: 83 45 fc 01

4004fc: eb fa

push %rbp

mov %rsp,%rbp

movl \$0x0,-0x4(%rbp)

addl \$0x1,-0x4(%rbp)

jmp 4004f8 <loop+0xb>

This assembly represents an infinite loop in C!

while (true) {...}

0x4004f8

%rip

4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

- **Control Mechanics in Assembly**

The `jmp` Instruction

- The **`jmp`** instruction jumps to another instruction in the assembly code (“Unconditional Jump”).

`jmp Label` (Direct Jump)

`jmp *Operand` (Indirect Jump)

- The destination can be hardcoded into the instruction (direct jump):

`jmp 404f8 <loop+0xb> # jump to instruction at 0x404f8`

- The destination can also be one of the usual operand forms (indirect jump):

`jmp *%rax # jump to instruction at address in %rax`

The jmp Instruction

1. How do we repeat instructions in a loop?

`jmp [target]`

- A 1-step unconditional jump (always jump when we execute this instruction)

What if we want a **conditional jump**?

Control Flow Mechanics

- In C, we have control flow statements (e.g., **if**, **else**, **while**, **for**) to write programs that are more expressive than just one instruction following another.
- This is *conditional execution of statements*: executing statements if one condition is true, executing other statements if one condition is false.
- How is this represented in assembly?

Control Flow Mechanics

```
if (x > y)
{
    // a
}
else
{
    // b
}
```

In Assembly:

1. Calculate the condition result
2. Based on the result, go to a or b

Conditional Jumps

- In assembly, it takes more than one instruction to do these two steps.
- Most often: 1 instruction to calculate the condition, 1 to conditionally jump

Common Pattern:

1. **cmp S1, S2** // compare two values

2. **je [target]** or **jne [target]** or **jl [target]** or ... // conditionally jump

“jump if
equal”

“jump if
not equal”

“jump if
less than”

Conditional Jumps

- There are variants of **jmp** that jump only if certain conditions are true (“Conditional Jump”). The jump location for these must be hardcoded into the instruction.

Instruction	Synonym	Set Condition
<code>je Label</code>	<code>jz</code>	Equal / zero
<code>jne Label</code>	<code>jnz</code>	Not equal / not zero
<code>js Label</code>		Negative
<code>jns Label</code>		Nonnegative
<code>jg Label</code>	<code>jnl</code>	Greater (signed >)
<code>jge Label</code>	<code>jnl</code>	Greater or equal (signed >=)
<code>jl Label</code>	<code>jnge</code>	Less (signed <)
<code>jle Label</code>	<code>jng</code>	Less or equal (signed <=)
<code>ja Label</code>	<code>jnb</code>	Above (unsigned >)
<code>jae Label</code>	<code>jnb</code>	Above or equal (unsigned >=)
<code>jb Label</code>	<code>jnae</code>	Below (unsigned <)
<code>jbe Label</code>	<code>jna</code>	Below or equal (unsigned <=)

Control

Read **cmp S1, S2** as “compare S2 to S1”:

// Jump if %edi > 2

```
cmp $2, %edi  
jg [target]
```

// Jump if %edi != 3

```
cmp $3, %edi  
jne [target]
```

// Jump if %edi == 4

```
cmp $4, %edi  
je [target]
```

// Jump if %edi <= 1

```
cmp $1, %edi  
jle [target]
```

References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



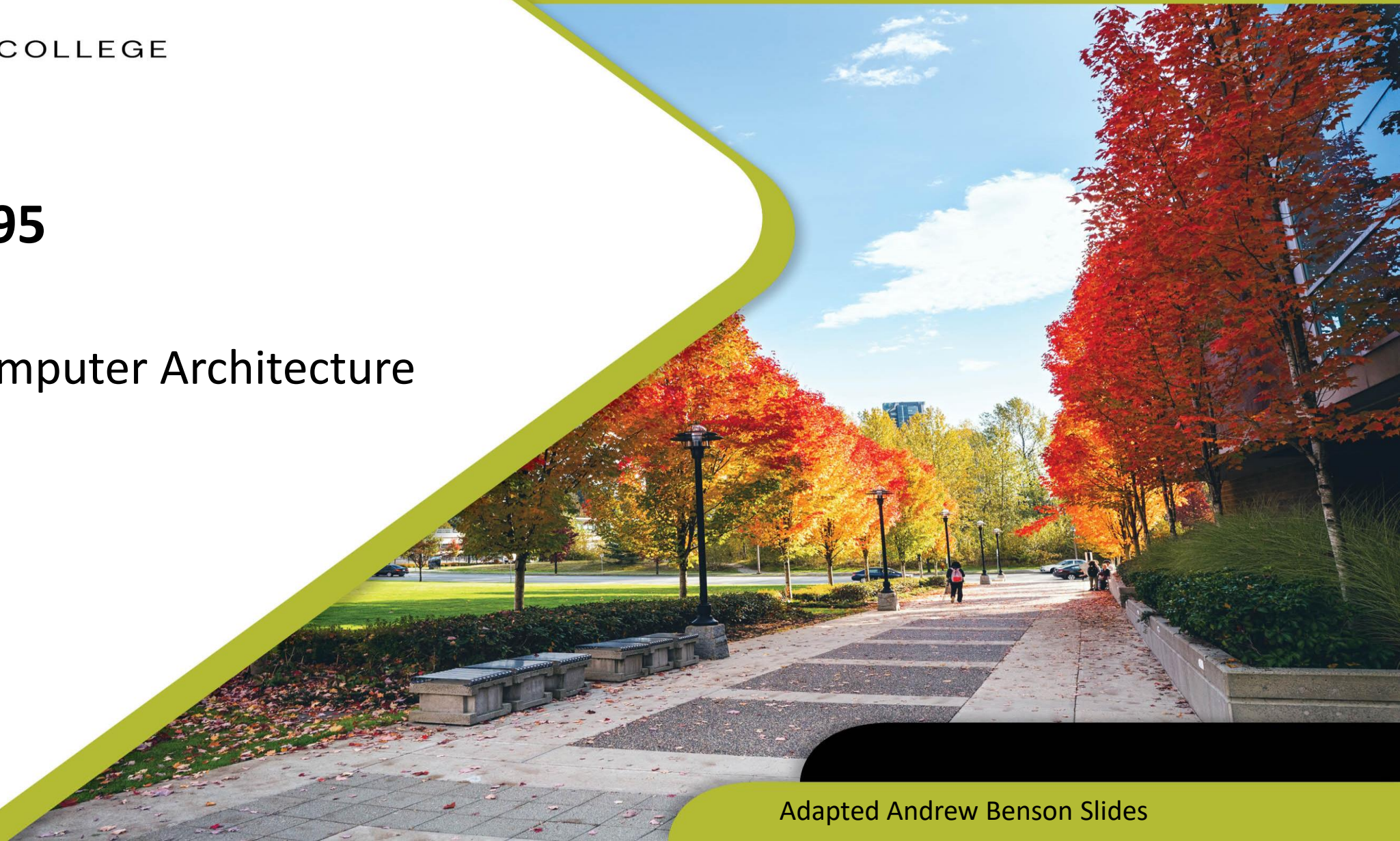
Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2395

Intro to Computer Architecture

Lecture 11



Plan for Today

- Flow Control
- Condition Codes
- If-Statements
- While-Loops
- For-Loops
- Other Instructions
- Revisiting %rip



DOUGLAS COLLEGE

Flow Control

Control

Read **cmp S1, S2** as “compare S2 to S1”:

// Jump if %edi > 2

```
cmp $2, %edi  
jg [target]
```

// Jump if %edi != 3

```
cmp $3, %edi  
jne [target]
```

// Jump if %edi == 4

```
cmp $4, %edi  
je [target]
```

// Jump if %edi <= 1

```
cmp $1, %edi  
jle [target]
```

Control

Read **cmp S1, S2** as “compare S2 to S1”:

// Jump if %edi > 2

```
cmp $2, %edi
jg [target]
```

// Jump if %edi == 4

```
cmp $4, %edi
je [target]
```

// Jump if

```
cmp $3, %edi
jne [target]
```

How does the jump instruction know anything about the compared values in the earlier instruction?

// Jump if %edi <= 1

```
$1, %edi
[target]
```



Condition Codes

Control

- The CPU has special registers called **condition codes** that are like “global variables”.
- They automatically keep track of information about the most recent arithmetic or logical operation.
 - **cmp** compares via calculation (subtraction) stores information in the condition codes
 - Conditional jump instructions look at these condition codes to know whether to jump
- What exactly are the condition codes, and how do they store this information?

Condition Codes

- Alongside normal registers, the CPU also has single-bit condition code registers.
- They store the results of the most recent arithmetic or logical operation.
- Most common condition codes:
 - **CF: Carry flag.** The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
 - **ZF: Zero flag.** The most recent operation yielded zero.
 - **SF: Sign flag.** The most recent operation yielded a negative value.
 - **OF: Overflow flag.** The most recent operation caused a two's-complement overflow (negative or positive).

Condition Codes

- Alongside normal registers, the CPU also has single-bit condition code registers. They store the results of the most recent arithmetic or logical operation.
- Example: if we calculate $t = a + b$, condition codes are set according to:
 - CF**: Carry flag (Unsigned Overflow): `(unsigned) t < (unsigned) a`
 - ZF**: Zero flag (Zero): `(t == 0)`
 - SF**: Sign flag (Negative): `(t < 0)`
 - OF**: Overflow flag (Signed Overflow): `(a < 0 == b < 0) && (t < 0 != a < 0)`

Setting Condition Codes

The **cmp** instruction is like the subtraction instruction, but it does not store the result anywhere. It just sets condition codes. (**Note** the operand order!)

`CMP S1, S2` $S2 - S1$

Instruction	Description
<code>cmpb</code>	Compare byte
<code>cmpw</code>	Compare word
<code>cmpd</code>	Compare double word
<code>cmpq</code>	Compare quad word

Control

Read **cmp S1,S2** as “*compare S2 to S1*”. It calculates $S2 - S1$ and updates the condition codes with the result.

```
// Jump if %edi > 2
```

```
cmp $2, %edi
```

```
jg [target]
```

```
// Jump if %edi != 3
```

```
cmp $3, %edi
```

```
jne [target]
```

```
// Jump if %edi == 4
```

```
cmp $4, %edi
```

```
je [target]
```

```
// Jump if %edi <= 1
```

```
cmp $1, %edi
```

```
jle [target]
```

Conditional Jumps

Conditional jumps can look at subsets of the condition codes in order to check their condition of interest.

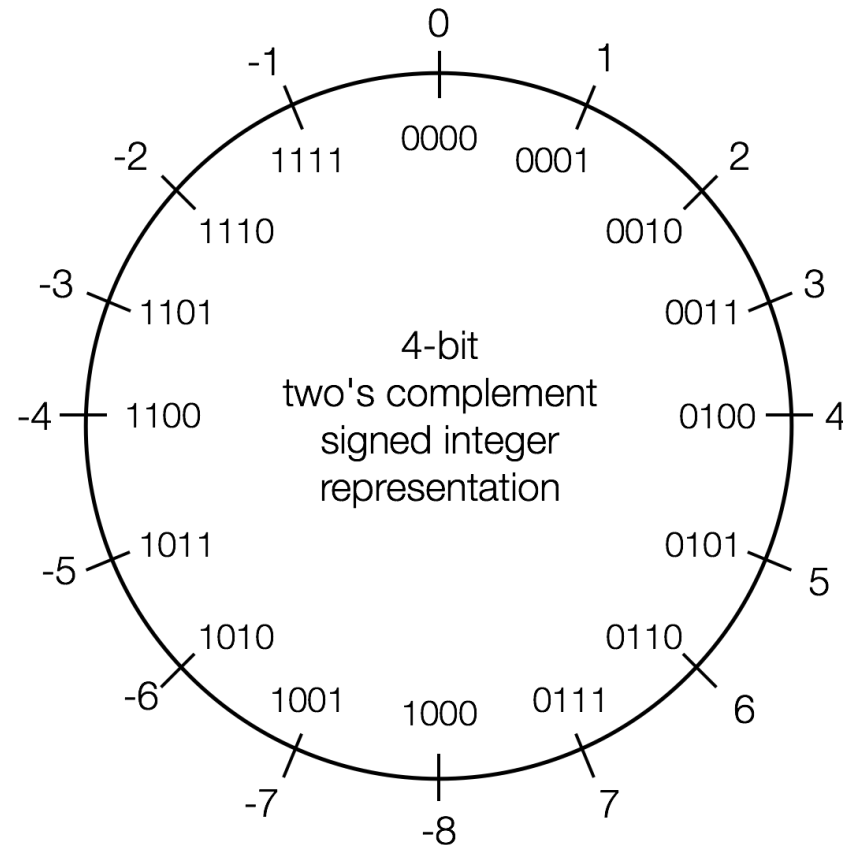
Instruction	Synonym	Set Condition
<code>je Label</code>	<code>jz</code>	Equal / zero (ZF = 1)
<code>jne Label</code>	<code>jnz</code>	Not equal / not zero (ZF = 0)
<code>js Label</code>		Negative (SF = 1)
<code>jns Label</code>		Nonnegative (SF = 0)
<code>jg Label</code>	<code>jnle</code>	Greater (signed >) (ZF = 0 and SF = OF)
<code>jge Label</code>	<code>jnl</code>	Greater or equal (signed >=) (SF = OF)
<code>jl Label</code>	<code>jnge</code>	Less (signed <) (SF != OF)
<code>jle Label</code>	<code>jng</code>	Less or equal (signed <=) (ZF = 1 or SF != OF)
<code>ja Label</code>	<code>jnbe</code>	Above (unsigned >) (CF = 0 and ZF = 0)
<code>jae Label</code>	<code>jnb</code>	Above or equal (unsigned >=) (CF = 0)
<code>jb Label</code>	<code>jnae</code>	Below (unsigned <) (CF = 1)
<code>jbe Label</code>	<code>jna</code>	Below or equal (unsigned <=) (CF = 1 or ZF = 1)

In-Class Discussion

- The **j1** instruction examines the **SF** and **OF** condition codes to see if **SF != OF**.
- Why does this determine if **s2** is less than **s1** in **cmp s1, s2**?

In-Class Discussion

- The **j1** instruction examines the **SF** and **OF** condition codes to see if **SF != OF**.
- Why does this determine if **s2** is less than **s1** in **cmp s1, s2**?



Setting Condition Codes

The **test** instruction is like **cmp**, but for **bitwise AND**. It does not store the & result anywhere. It just sets condition codes.

TEST S1, S2

S2 & S1

Instruction	Description
testb	Test byte
testw	Test word
testl	Test double word
testq	Test quad word

In-Class Practice #1

je target

jump if ZF is 1

Let **%edi** store **0x10**. Will we jump in the following cases?

%edi

0x10

1. `cmp     $0x10, %edi`
 `je       40056f`
 `add     $0x1, %edi`

2. `test     $0x10, %edi`
 `je       40056f`
 `add     $0x1, %edi`

In-Class Practice #1

je target

jump if ZF is 1

Let **%edi** store **0x10**. Will we jump in the following cases? %edi

0x10

1. **cmp** \$0x10, %edi
 je 40056f
 add \$0x1, %edi

$S2 - S1 = 0$, so jump

2. **test** \$0x10, %edi
 je 40056f
 add \$0x1, %edi

$S2 \& S1 \neq 0$, so don't jump!

Extra Credit Poll (1)

```
00000000004004d6 <if_then>:                                %edi 0x5
4004d6: 83 ff 06      cmp    $0x6, %edi
4004d9: 75 03        jne    4004de <if_then+0x8>
4004db: 83 c7 01      add    $0x1, %edi
4004de: 8d 04 3f      lea    (%rdi,%rdi,1), %eax
4004e1: c3           retq
```

What is the value of **%rip** after executing the **jne** instruction?

1. **4004d9**
2. **4004db**
3. **4004de**

Extra Credit Poll (1)

```

00000000004004d6 <if_then>:                                %edi  0x5
4004d6:  83 ff 06      cmp    $0x6, %edi
4004d9:  75 03          jne    4004de <if_then+0x8>
4004db:  83 c7 01      add    $0x1, %edi
4004de:  8d 04 3f      lea    (%rdi,%rdi,1), %eax
4004e1:  c3            retq

```

What is the value of **%rip** after executing the **jne** instruction?

1. **4004d9**
2. **4004db**
3. **4004de**

$0x5 - 0x6 = -1 \neq 0$

%rip = 4004de (i.e., do not add 1)

Extra Credit Poll (2)

```
00000000004004d6 <if_then>:                                %edi 0x5
4004d6: 83 ff 06      cmp    $0x6, %edi
4004d9: 75 03         jne    4004de <if_then+0x8>
400rdb: 83 c7 01      add    $0x1, %edi
4004de: 8d 04 3f      lea    (%rdi,%rdi,1), %eax
4004e1: c3           retq
```

What is the value of **%eax** when we hit the **retq** instruction?

1. **4004e1**
2. **0x2**
3. **0xa**

Extra Credit Poll (2)

```

00000000004004d6 <if_then>:                                %edi  0x5
4004d6:  83 ff 06      cmp    $0x6, %edi
4004d9:  75 03         jne    4004de <if_then+0x8>
400rdb:  83 c7 01      add    $0x1, %edi
4004de:  8d 04 3f      lea    (%rdi,%rdi,1), %eax
4004e1:  c3           retq

```

What is the value of **%eax** when we hit the **retq** instruction?

1. 4004e1
2. 0x2
3. 0xa

$$5 + 1 * 5 = 10 = 0xa$$

If-Statements

If-Statements

- How can we use instructions like `cmp` and conditional jumps to implement if statements in assembly?

In-Class Practice #2 – Fill in the Blanks

0000000100401080 <if_then>:

100401080:	83 f9 04	cmp	\$0x4,%ecx
100401083:	74 04	je	100401089 <if_then+0x9>
100401085:	8d 04 09	lea	(%rcx,%rcx,1),%eax
100401088:	c3	ret	
100401089:	83 c1 01	add	\$0x1,%ecx
10040108c:	eb f7	jmp	100401085 <if_then+0x5>

```
int if_then(int param1)
{
    if (----?----)
        ----?----;

    return ----?----;
}
```

In-Class Practice #2 – Fill in the Blanks

0000000100401080 <if_then>:

100401080:	83 f9 04	cmp	\$0x4,%ecx
100401083:	74 04	je	100401089 <if_then+0x9>
100401085:	8d 04 09	lea	(%rcx,%rcx,1),%eax
100401088:	c3	ret	
100401089:	83 c1 01	add	\$0x1,%ecx
10040108c:	eb f7	jmp	100401085 <if_then+0x5>

```
int if_then(int param1)
{
    if (param1 == 4)
        param1++;

    return param1 * 2;
}
```


In-Class Practice #3 – Fill in the Blanks

0000000100401080 <if_then>:

100401080:	83 f9 03	cmp	\$0x3,%ecx
100401083:	7f 06	jg	10040108b <if_then+0xb>
100401085:	b8 00 00 00 00	mov	\$0x0,%eax
10040108a:	c3	ret	
10040108b:	b8 0a 00 00 00	mov	\$0xa,%eax
100401090:	eb f8	jmp	10040108a <if_then+0xa>

```

int if_then(int param1)
{
    int result;
    if (---?---)
        ---?---;
    else
        ---?---;

    return ---?---;
}

```

In-Class Practice #3 – Fill in the Blanks

0000000100401080 <if_then>:

100401080:	83 f9 03	cmp	\$0x3,%ecx
100401083:	7f 06	jg	10040108b <if_then+0xb>
100401085:	b8 00 00 00 00	mov	\$0x0,%eax
10040108a:	c3	ret	
10040108b:	b8 0a 00 00 00	mov	\$0xa,%eax
100401090:	eb f8	jmp	10040108a <if_then+0xa>

```

int if_then(int param1)
{
    int result;
    if (param1 > 3)
        result = 10;
    else
        result = 0;

    return result++;
}

```

While-Loops

Loops and Control Flow

```
void loop()  
{  
    int i = 0;  
    while (i < 100)  
        i++;  
}
```

```
00000000100401080 <loop>:  
100401080: b8 00 00 00 00  
100401085: eb 03  
100401087: 83 c0 01  
10040108a: 83 f8 63  
10040108d: 7e f8  
10040108f: c3
```

```
mov    $0x0,%eax  
jmp     10040108a <loop+0xa>  
add     $0x1,%eax  
cmp     $0x63,%eax  
jle     100401087 <loop+0x7>  
ret
```

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

```
00000000100401080 <loop>:
    100401080:    b8 00 00 00 00
    100401085:    eb 03
    100401087:    83 c0 01
    10040108a:    83 f8 63
    10040108d:    7e f8
    10040108f:    c3
```

```
mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret
```

Set %eax (i) to 0.

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

```
00000000100401080 <loop>:
    100401080:    b8 00 00 00 00
    100401085:    eb 03
    100401087:    83 c0 01
    10040108a:    83 f8 63
    10040108d:    7e f8
    10040108f:    c3
```

```
mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret
```

Jump to another instruction.

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

00000000100401080 <loop>:

```
100401080:  b8 00 00 00 00
100401085:  eb 03
100401087:  83 c0 01
10040108a:  83 f8 63
10040108d:  7e f8
10040108f:  c3
```

```
mov    $0x0,%eax
jmp    10040108a <loop+0xa>
add    $0x1,%eax
cmp    $0x63,%eax
jle    100401087 <loop+0x7>
ret
```

Jump to another instruction.

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

```
00000000100401080 <loop>:
```

```
100401080: b8 00 00 00 00
100401085: eb 03
100401087: 83 c0 01
10040108a: 83 f8 63
10040108d: 7e f8
10040108f: c3
```

```
mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret
```

Compare %eax (i) to 0x63 (99) by calculating %eax – 0x63. This is 0 – 99 = -99, so it sets the Sign Flag to 1.

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

```
00000000100401080 <loop>:
```

```
100401080:  b8 00 00 00 00
100401085:  eb 03
100401087:  83 c0 01
10040108a:  83 f8 63
10040108d:  7e f8
10040108f:  c3
```

```
mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret
```

jle means “jump if less than or equal”. This jumps if %eax ≤ 0x63. The flags indicate this is true, so we jump.

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

```
00000000100401080 <loop>:
```

```
100401080: b8 00 00 00 00
```

```
100401085: eb 03
```

```
100401087: 83 c0 01
```

```
10040108a: 83 f8 63
```

```
10040108d: 7e f8
```

```
10040108f: c3
```

```
mov    $0x0,%eax
```

```
jmp     10040108a <loop+0xa>
```

```
add     $0x1,%eax
```

```
cmp     $0x63,%eax
```

```
jle     100401087 <loop+0x7>
```

```
ret
```

jle means “jump if less than or equal”. This jumps if %eax ≤ 0x63. The flags indicate this is true, so we jump.

Loops and Control Flow

```
void loop()  
{  
    int i = 0;  
    while (i < 100)  
        i++;  
}
```

```
00000000100401080 <loop>:  
100401080: b8 00 00 00 00  
100401085: eb 03  
100401087: 83 c0 01  
10040108a: 83 f8 63  
10040108d: 7e f8  
10040108f: c3
```

```
mov    $0x0,%eax  
jmp     10040108a <loop+0xa>  
add     $0x1,%eax  
cmp     $0x63,%eax  
jle     100401087 <loop+0x7>  
ret
```

Add 1 to %eax (i).

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

```
00000000100401080 <loop>:
```

```
100401080: b8 00 00 00 00
100401085: eb 03
100401087: 83 c0 01
10040108a: 83 f8 63
10040108d: 7e f8
10040108f: c3
```

```
mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret
```

Compare %eax (i) to 0x63 (99) by calculating %eax – 0x63. This is 1 – 99 = -98, so it sets the Sign Flag to 1.

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

```
00000000100401080 <loop>:
```

```
100401080:  b8 00 00 00 00
100401085:  eb 03
100401087:  83 c0 01
10040108a:  83 f8 63
10040108d:  7e f8
10040108f:  c3
```

```
mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret
```

jle means “jump if less than or equal”. This jumps if %eax ≤ 0x63. The flags indicate this is true, so we jump.

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

```
00000000100401080 <loop>:
```

```
100401080:  b8 00 00 00 00
100401085:  eb 03
100401087:  83 c0 01
10040108a:  83 f8 63
10040108d:  7e f8
10040108f:  c3
```

```
mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret
```

We continue in this pattern until we do not make this conditional jump. When will that be?

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

```
00000000100401080 <loop>:
    100401080:    b8 00 00 00 00
    100401085:    eb 03
    100401087:    83 c0 01
    10040108a:    83 f8 63
    10040108d:    7e f8
    10040108f:    c3
```

```
mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret
```

We will stop looping when this comparison says that $\%eax - 0x63 > 0$!

Loops and Control Flow

```
void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}
```

```
00000000100401080 <loop>:
```

```
100401080:  b8 00 00 00 00
100401085:  eb 03
100401087:  83 c0 01
10040108a:  83 f8 63
10040108d:  7e f8
10040108f:  c3
```

```
mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret
```

Then, we return from the function.



DOUGLAS COLLEGE

For-Loops

Loops and Control Flow

```
int loop()
{
    int result = 10;
    for (int i = 0; i < 100; i++)
        result += 10;
    return result;
}
```

```
00000000100401080 <loop>:
100401080:  b8 00 00 00 00    mov     $0x0,%eax
100401085:  ba 0a 00 00 00    mov     $0xa,%edx
10040108a:  eb 06             jmp     100401092 <loop+0x12>
10040108c:  83 c2 0a          add     $0xa,%edx
10040108f:  83 c0 01          add     $0x1,%eax
100401092:  83 f8 63          cmp     $0x63,%eax
100401095:  7e f5             jle     10040108c <loop+0xc>
100401097:  89 d0             mov     %edx,%eax
100401099:  c3               ret
```



Other Instructions

Condition Code-Dependent Instructions

- There are three common instruction types that use condition codes:
 - **jmp** instructions conditionally jump to a different next instruction.
 - **set** instructions conditionally set a byte to 0 or 1.
 - New versions of **mov** instructions conditionally move data.

The set Instruction

set instructions conditionally **set a byte to 0 or 1**.

- Reads current state of flags.
- Destination is a single-byte register (e.g., **%a1**) or single-byte memory location.
- Does not perturb other bytes of register.
- Typically followed by **movzbl** to zero those bytes.

```

int small(int x)
{
    return x < 16;
}
00000000100401080 <small>:
    100401080:  83 f9 0f    cmp     $0xf,%ecx
    100401083:  0f 9e c0    setle  %a1
    100401086:  0f b6 c0    movzbl %a1,%eax
    100401089:  c3                ret

```

The `cmov` Instruction (Conditional Move)

`cmovx src,dst` conditionally moves data in `src` to data in `dst`.

- Move `src` to `dst` if condition `x` holds; no change otherwise.
- `src` is memory address/register, `dst` is register.
- May be more efficient than branch (i.e., jump).
- Often seen with C **ternary operator**: `result = test ? then: else;`

```
int max(int x, int y)
{
    return x > y ? x : y;
}
```

```
0000000100401080 <max>:
    100401080:  39 ca  cmp    %ecx,%edx
    100401082:  89 c8  mov    %ecx,%eax
    100401084:  0f 4d  c2     cmovge %edx,%eax
    100401087:  c3     ret
```

In-Class Practice #4 – Fill in the Blanks

0000000100401080 <loop>:

100401080:	b8 01 00 00 00	mov	\$0x1,%eax
100401085:	eb 0c	jmp	100401093 <loop+0x13>
100401087:	4c 8d 04 11	lea	(%rcx,%rdx,1),%r8
10040108b:	49 0f af c0	imul	%r8,%rax
10040108f:	48 83 c1 01	add	\$0x1,%rcx
100401093:	48 39 d1	cmp	%rdx,%rcx
100401096:	7c ef	j1	100401087 <loop+0x7>
100401098:	c3	ret	

```

long loop(long a, long b)
{
    long result = 1;
    while (---?---)
    {
        result = ---?---;
        a = ---?---;
    }
    return result;
}

```

In-Class Practice #4 – Fill in the Blanks

0000000100401080 <loop>:

100401080:	b8 01 00 00 00	mov	\$0x1,%eax
100401085:	eb 0c	jmp	100401093 <loop+0x13>
100401087:	4c 8d 04 11	lea	(%rcx,%rdx,1),%r8
10040108b:	49 0f af c0	imul	%r8,%rax
10040108f:	48 83 c1 01	add	\$0x1,%rcx
100401093:	48 39 d1	cmp	%rdx,%rcx
100401096:	7c ef	j1	100401087 <loop+0x7>
100401098:	c3	ret	

```

long loop(long a, long b)
{
    long result = 1;
    while (a < b)
    {
        result = result * (a + b);
        a = a + 1;
    }
    return result;
}

```




DOUGLAS COLLEGE

Revisiting the Program Counter

Condition Code-Dependent Instructions

- **%rip** is a special register that points to the next instruction to execute.
- Let us dive deeper into how **%rip** works, and how jumps modify it.

%rip

```

void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}

```

00000000100401080 <loop>:

100401080: b8 00 00 00 00
 100401085: eb 03
 100401087: 83 c0 01
 10040108a: 83 f8 63
 10040108d: 7e f8
 10040108f: c3

mov \$0x0,%eax
 jmp 10040108a <loop+0xa>
 add \$0x1,%eax
 cmp \$0x63,%eax
 jle 100401087 <loop+0x7>
 ret

100401080
 100401080 + 5
 100401080 + 7
 100401080 + 10
 100401080 + 15

These are 0-based offsets in bytes for each instruction relative to the start of this function.

%rip

```

void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}

```

```

00000000100401080 <loop>:

```

```

100401080: b8 00 00 00 00
100401085: eb 03
100401087: 83 c0 01
10040108a: 83 f8 63
10040108d: 7e f8
10040108f: c3

```

```

mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret

```

These are bytes for the machine code instructions. Instructions are variable length.

%rip

```

void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}

```

00000000100401080 <loop>:

100401080: b8 00 00 00 00

100401085: eb 03

100401087: 83 c0 01

10040108a: 83 f8 63

10040108d: 7e f8

10040108f: c3

mov \$0x0,%eax

jmp 10040108a <loop+0xa>

add \$0x1,%eax

cmp \$0x63,%eax

jle 100401087 <loop+0x7>

ret

0xeb means jmp.

%rip

```

void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}

```

00000000100401080 <loop>:

100401080: b8 00 00 00 00

100401085: **eb 03**

100401087: 83 c0 01

10040108a: 83 f8 63

10040108d: 7e f8

10040108f: c3

mov \$0x0,%eax

jmp 10040108a <loop+0xa>

add \$0x1,%eax

cmp \$0x63,%eax

jle 100401087 <loop+0x7>

ret

0x03 is the number of instruction bytes to jump relative to %rip.

With no jump, **%rip** would advance to the next line. This **jmp** says to then go **3** bytes further.

%rip

```

void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}

```

```

00000000100401080 <loop>:
100401080:  b8 00 00 00 00
100401085:  eb 03
100401087:  83 c0 01
10040108a:  83 f8 63
10040108d:  7e f8
10040108f:  c3

```

```

mov    $0x0,%eax
jmp     10040108a <loop+0xa>
add     $0x1,%eax
cmp     $0x63,%eax
jle     100401087 <loop+0x7>
ret

```

0x03 is the number of instruction bytes to jump relative to %rip.

With no jump, **%rip** would advance to the next line. This **jmp** says to then go **3** bytes further.

%rip

```

void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}

```

00000000100401080 <loop>:

100401080: b8 00 00 00 00

100401085: eb 03

100401087: 83 c0 01

10040108a: 83 f8 63

10040108d: 7e f8

10040108f: c3

mov \$0x0,%eax

jmp 10040108a <loop+0xa>

add \$0x1,%eax

cmp \$0x63,%eax

jle 100401087 <loop+0x7>

ret

0x7e means jle.

%rip

```

void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}

```

```

00000000100401080 <loop>:

```

```

100401080:  b8 00 00 00 00

```

```

100401085:  eb 03

```

```

100401087:  83 c0 01

```

```

10040108a:  83 f8 63

```

```

10040108d:  7e f8

```

```

10040108f:  c3

```

```

mov    $0x0,%eax

```

```

jmp    10040108a <loop+0xa>

```

```

add    $0x1,%eax

```

```

cmp    $0x63,%eax

```

```

jle    100401087 <loop+0x7>

```

```

ret

```

0xf8 is the number of instruction bytes to jump relative to **%rip**. This is -8 (in two's complement).

With no jump, **%rip** would advance to the next line. This **jmp** says to then go 8 bytes back.

%rip

```

void loop()
{
    int i = 0;
    while (i < 100)
        i++;
}

```

00000000100401080 <loop>:

100401080: b8 00 00 00 00

100401085: eb 03

100401087: 83 c0 01

10040108a: 83 f8 63

10040108d: 7e f8

10040108f: c3

mov \$0x0,%eax

jmp 10040108a <loop+0xa>

add \$0x1,%eax

cmp \$0x63,%eax

jle 100401087 <loop+0x7>

ret

0xf8 is the number of instruction bytes to jump relative to **%rip**. This is -8 (in two's complement).

With no jump, **%rip** would advance to the next line. This **jmp** says to then go 8 bytes back.

Summary (Instruction Pointer)

- Machine code instructions live in main memory, just like stack and heap data.
- **%rip** is a register that stores a number (an address) of the next instruction to execute. It marks our place in the program's instructions.
- To advance to the next instruction, special hardware adds the size of the current instruction in bytes.
- **jmp** instructions work by adjusting **%rip** by a specified amount.

References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



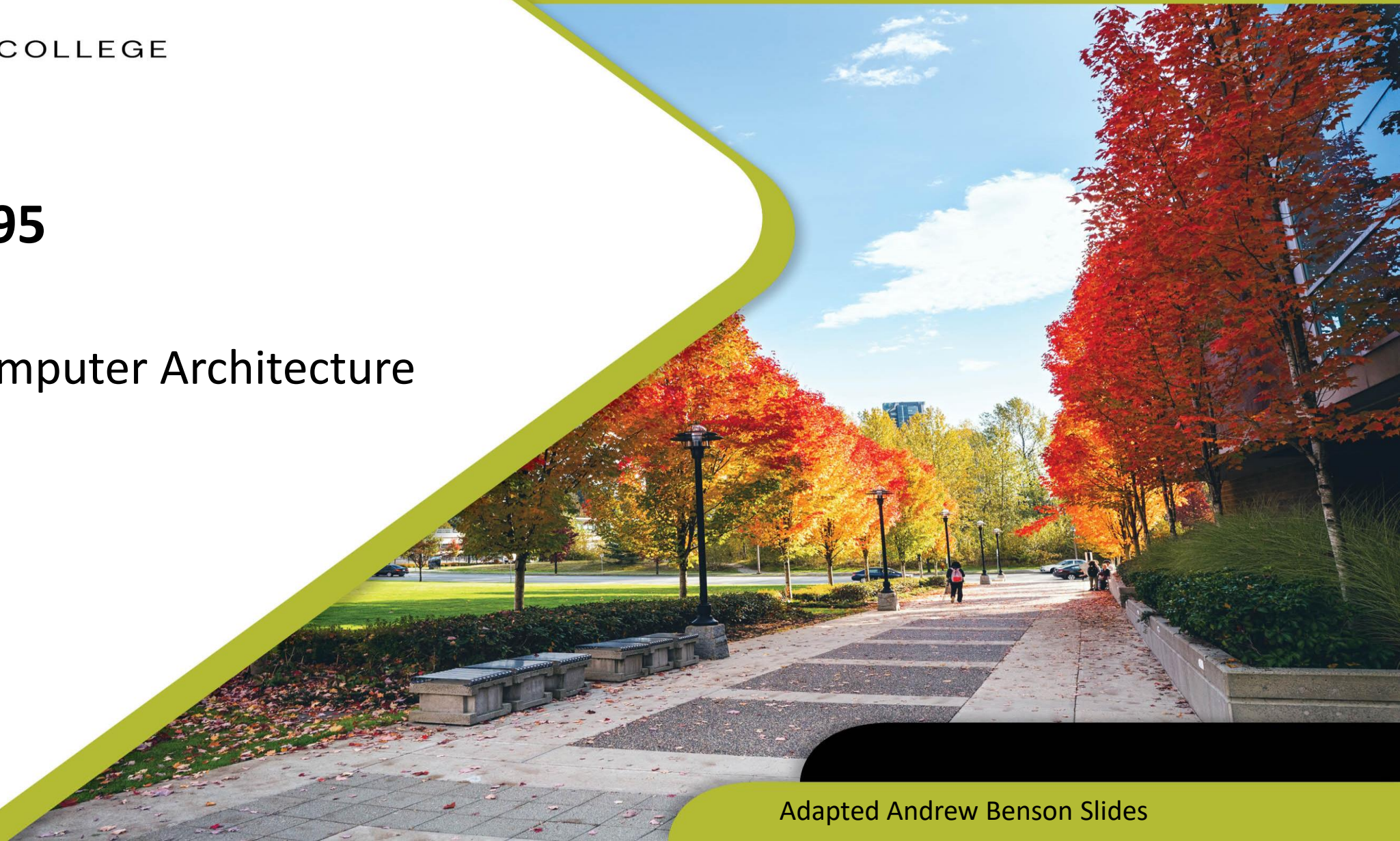
Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

CMPT 2395

Intro to Computer Architecture

Lecture 12



Plan for Today

- Calling a Function in Assembly
- Floating Point Representation
- Parallel Programming



DOUGLAS COLLEGE

Calling a Function in Assembly

Calling a Function in Assembly

To call a function in assembly, we must do a few things:

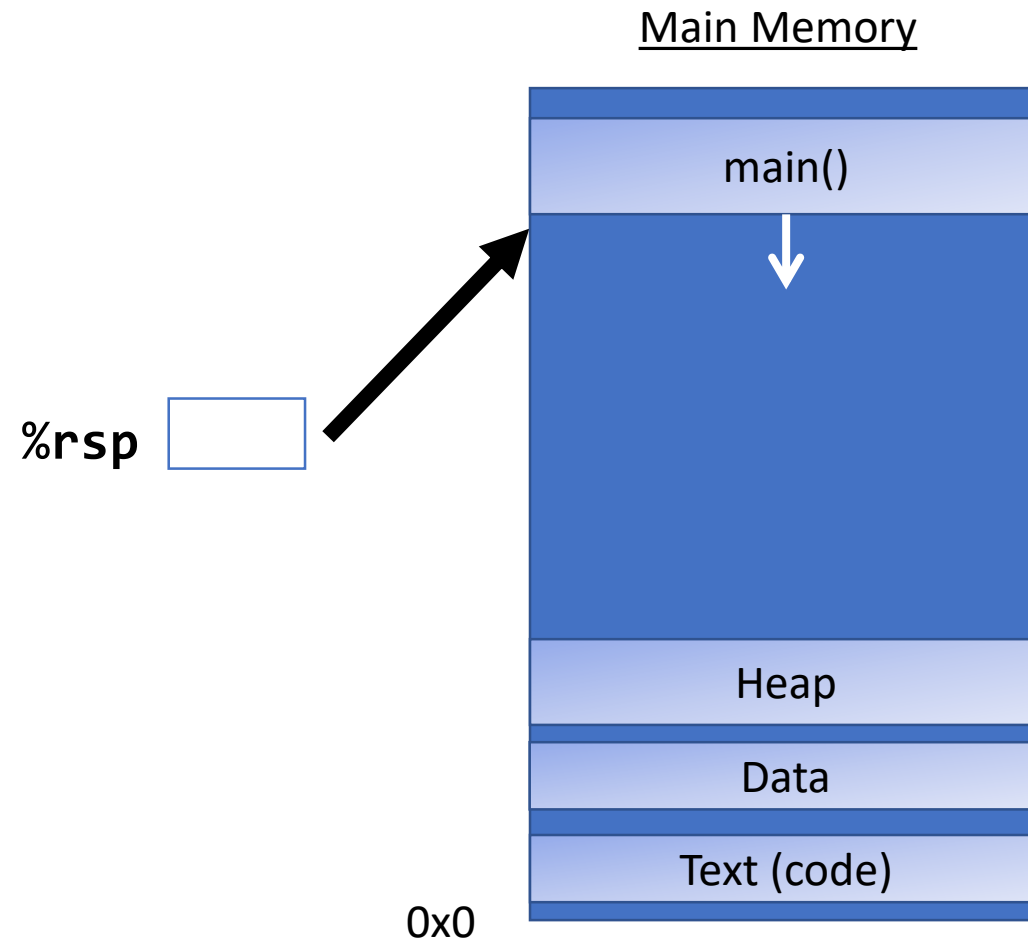
- **Pass Control** – `%rip` must be adjusted to execute the callee's instructions, and then resume the caller's instructions afterwards.
- **Pass Data** – we must pass any parameters and receive any return value.
- **Manage Memory** – we must handle any space needs of the callee on the stack.

How does assembly
interact with the stack?

Terminology: **caller** function calls the **callee** function.

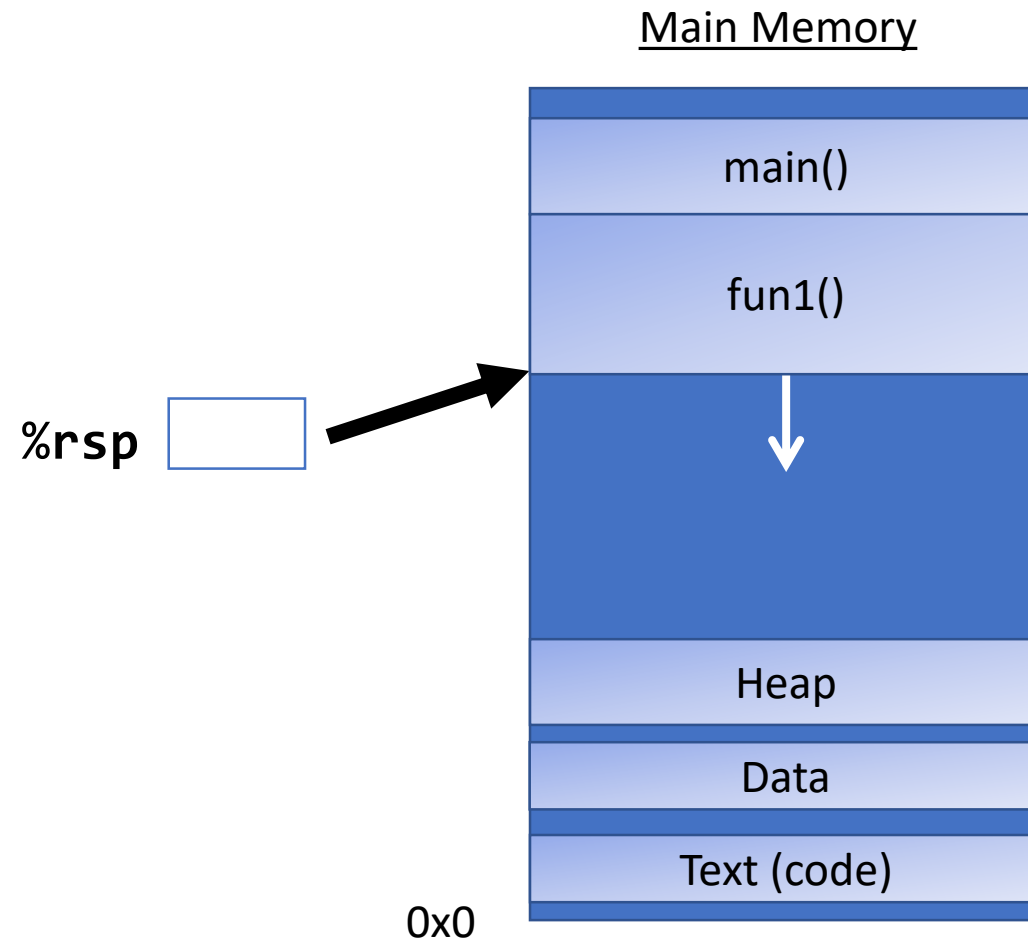
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



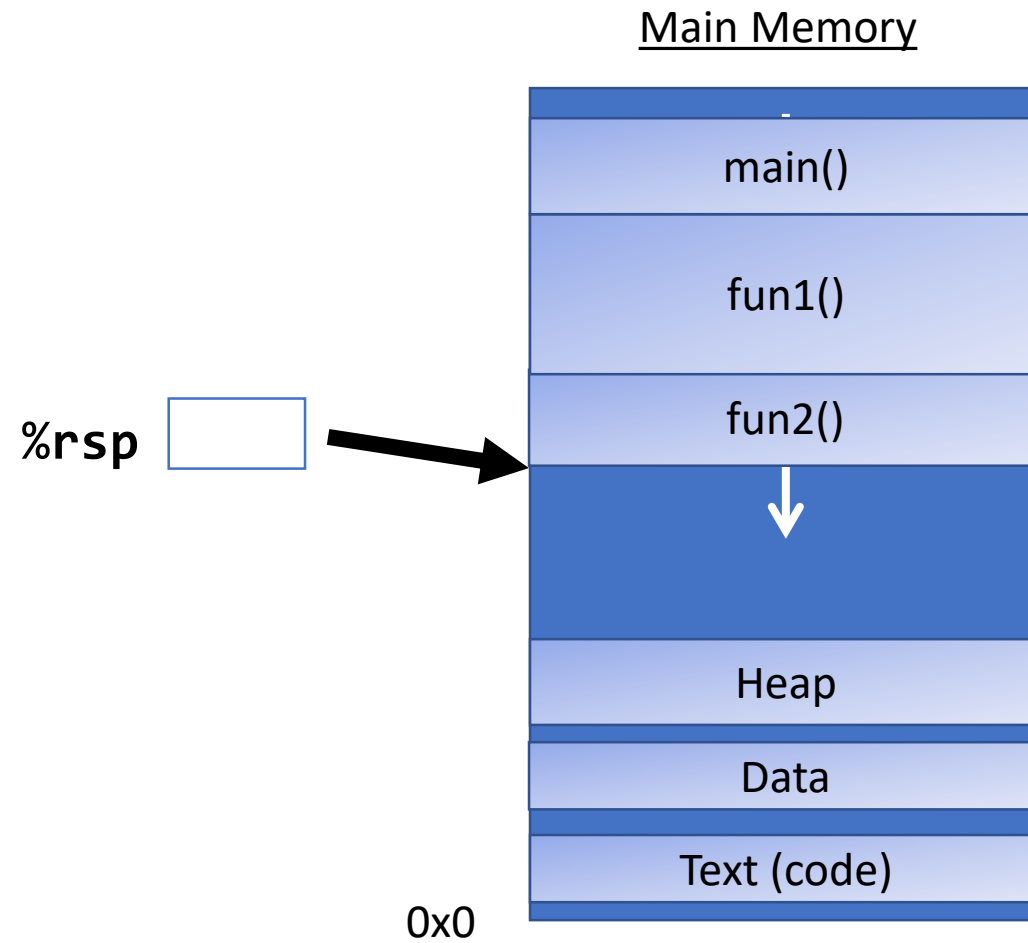
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



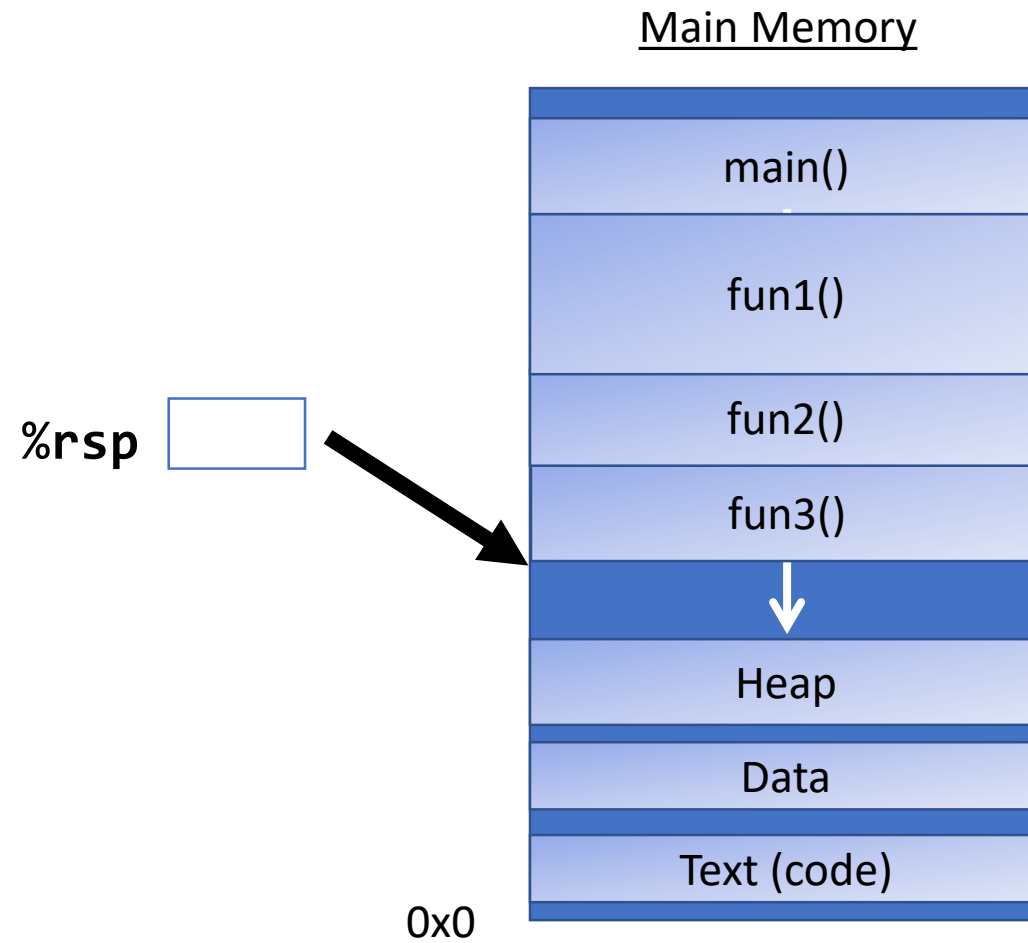
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



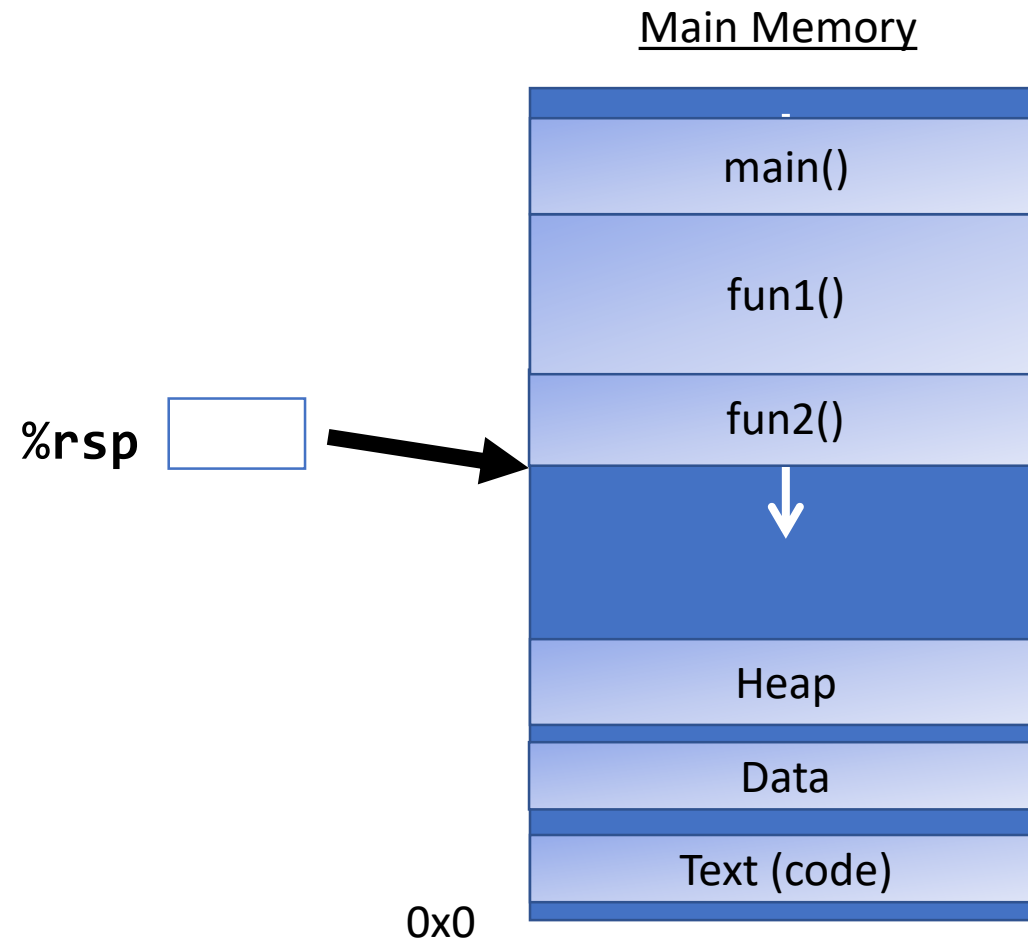
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



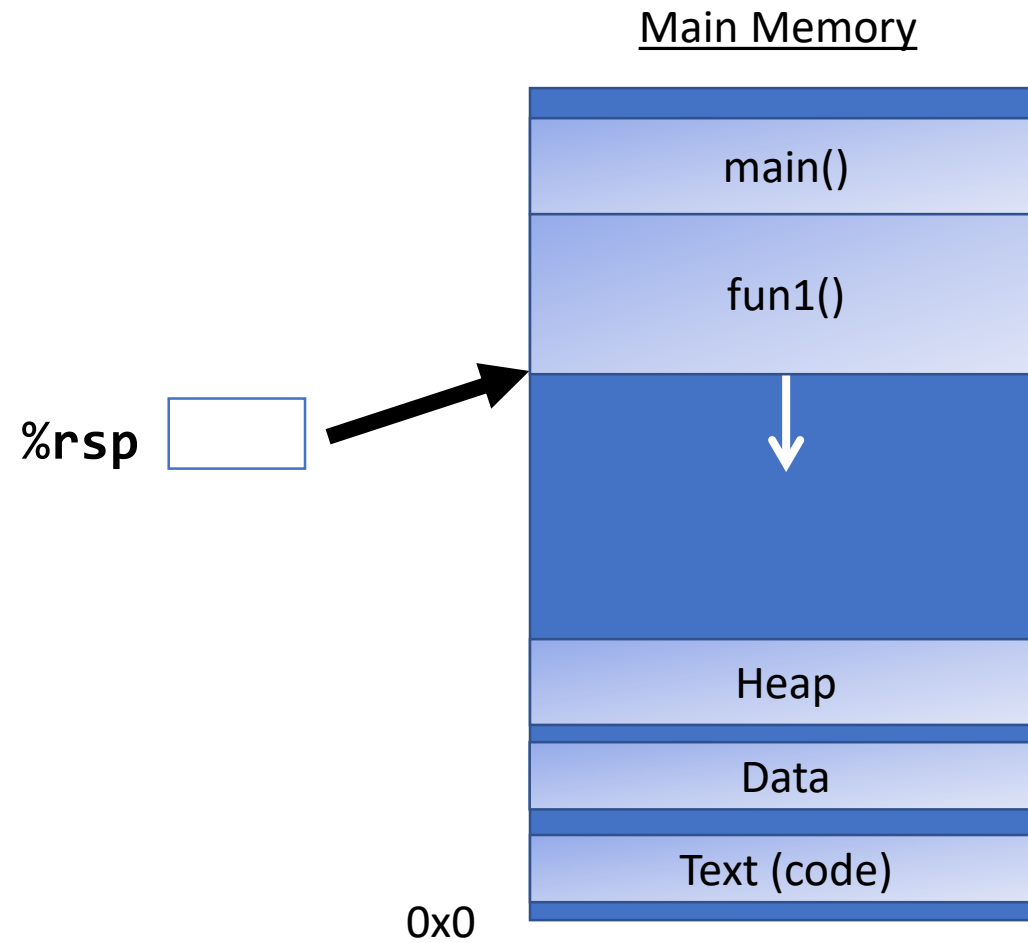
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



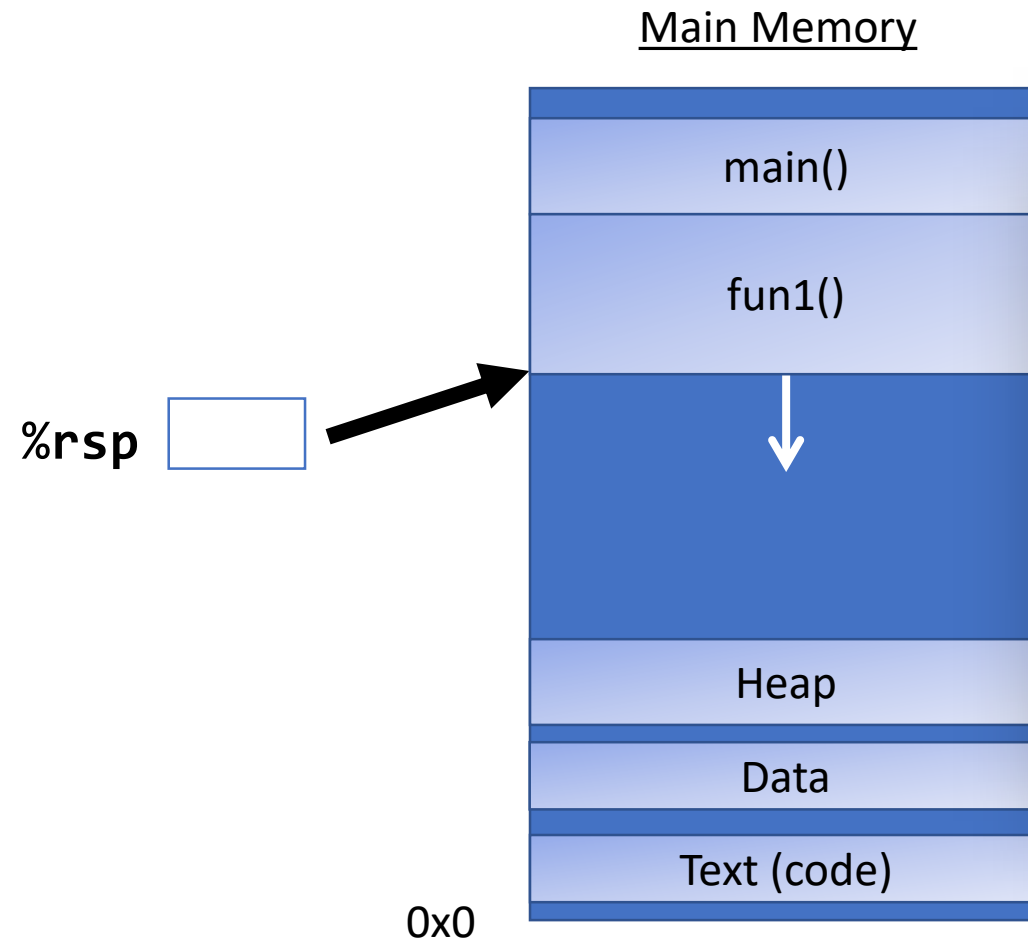
%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



%rsp

- **%rsp** is a special register that stores the address of the current “top” of the stack (the bottom in our diagrams, since the stack grows downwards).



Key idea: %rsp must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.

The push Instruction

- The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
<code>pushq S</code>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$

- This behavior is equivalent to the following:
`subq $8, %rsp`
`movq S, (%rsp)`
- Sometimes, we will see instructions just explicitly decrement the stack pointer to make room for future data.

The pop Instruction

- The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **%rsp** accordingly.

Instruction	Effect
popq D	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8;$

- Note:** this does not remove/clear out the data! It just increments **%rsp** to indicate the next **push** can overwrite that location.

The pop Instruction

- The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **%rsp** accordingly.

Instruction	Effect
<code>popq D</code>	$D \leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8;$

- This behavior is equivalent to the following:

```
movq (%rsp), D
addq $8, %rsp
```

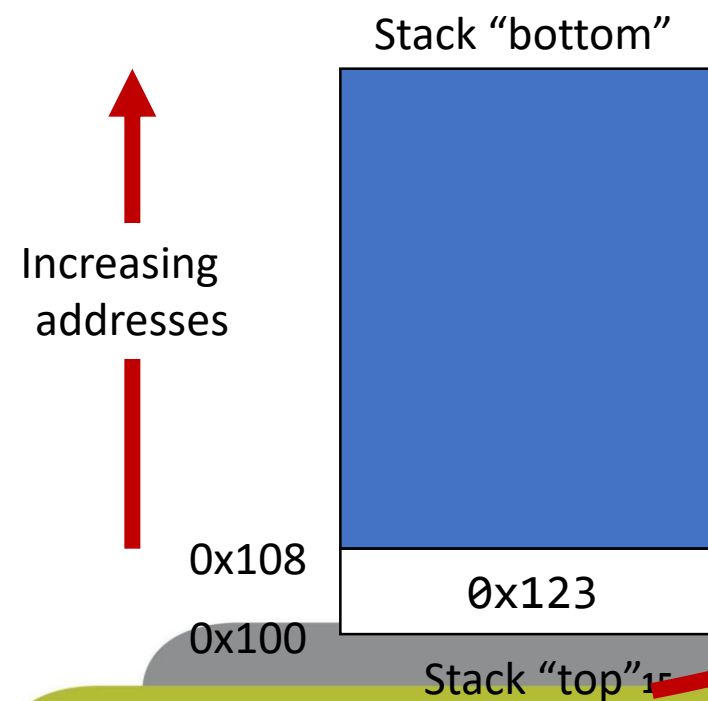
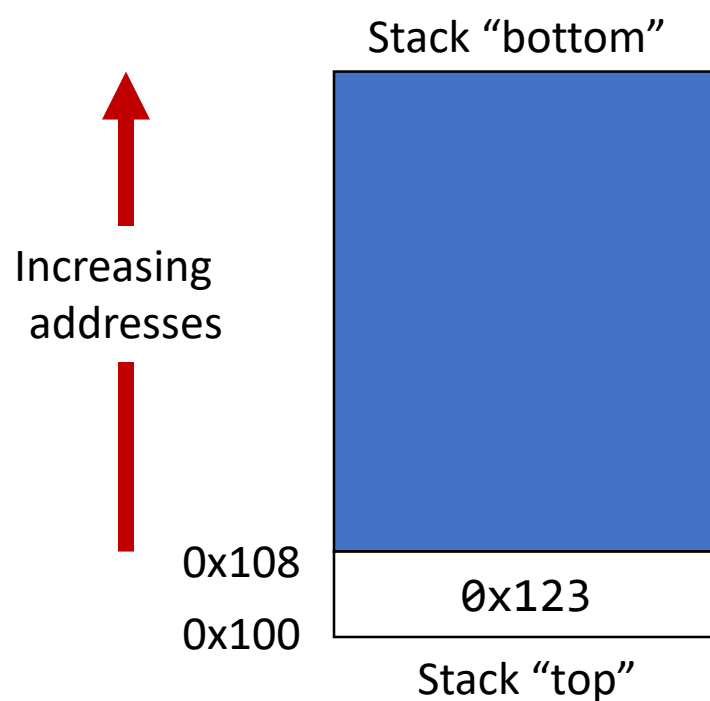
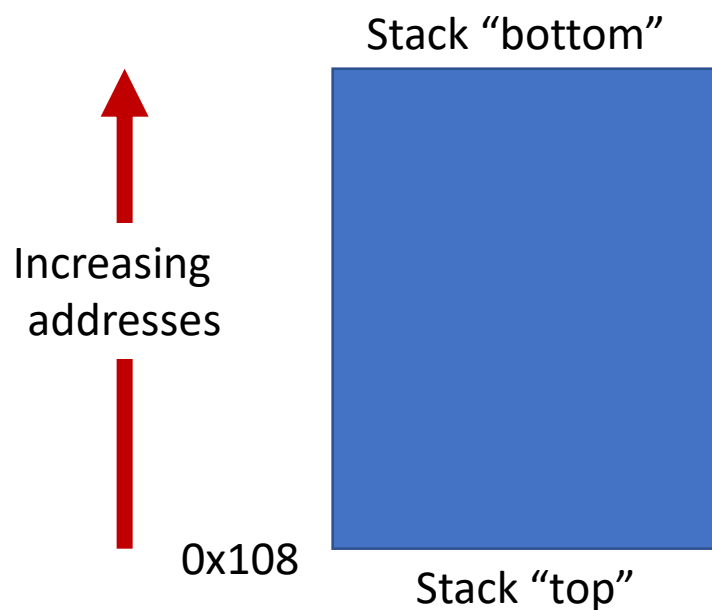
- Sometimes, you'll see instructions just explicitly increment the stack pointer to pop data.

Stack Example

Initial Condition	
%rax	0x123
%rdx	0
%rsp	0x108

pushq %rax	
%rax	0x123
%rdx	0
%rsp	0x100

popq %rdx	
%rax	0x123
%rdx	0x123
%rsp	0x108



Remembering where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the next caller instruction to go back to after the function ends.

Solution: push the next value of `%rip` onto the stack. Then, call the function. When function execution is finished, put this value back into `%rip` and continue executing.

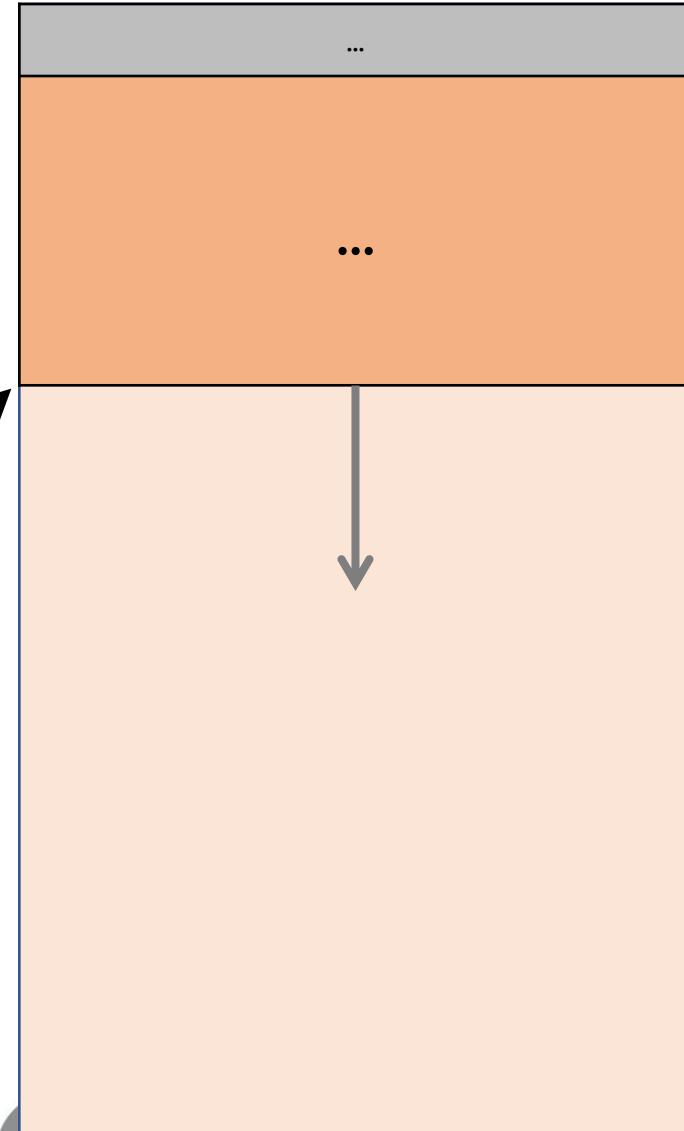
e.g., `main()` calls `fun()`:

`main()`

`%rsp` 0xff20

`%rip` 0x3021

Stack



Remembering where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the next caller instruction to go back to after the function ends.

Solution: push the next value of `%rip` onto the stack. Then, call the function. When function execution is finished, put this value back into `%rip` and continue executing.

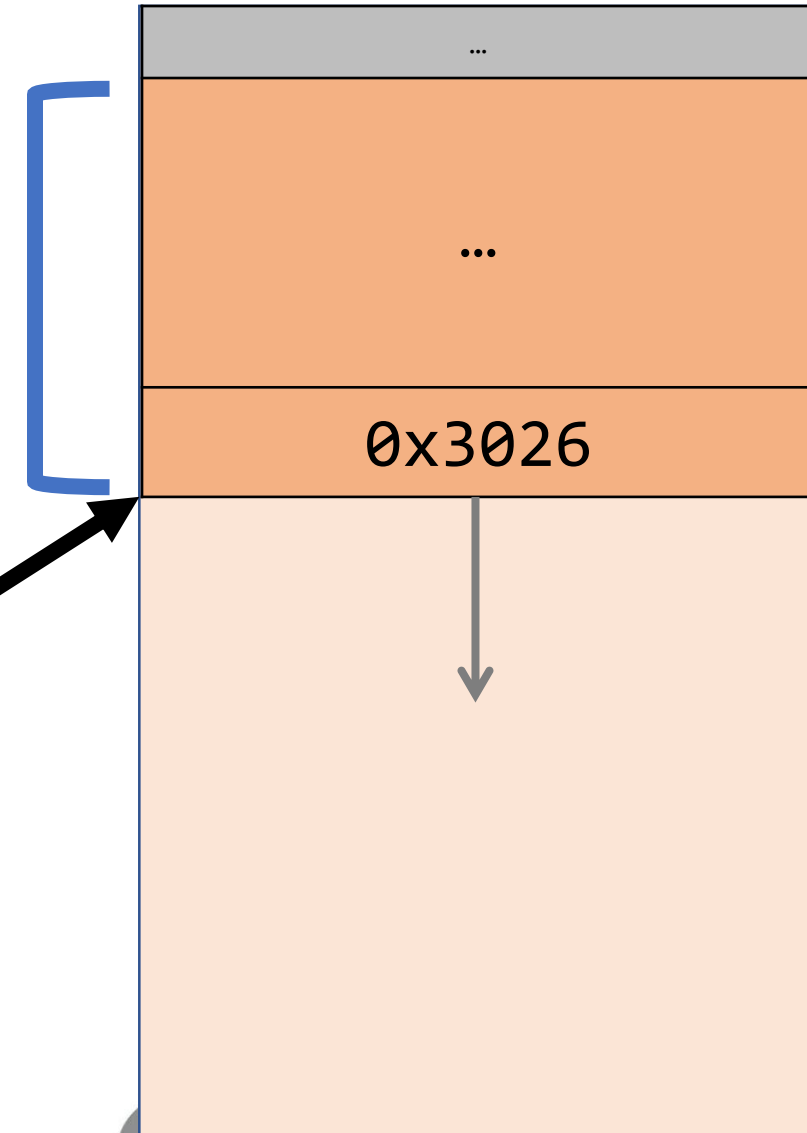
e.g., `main()` calls `fun()`:

`main()`

`%rsp` 0xff18

`%rip` 0x3021

Stack



Remembering where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the next caller instruction to go back to after the function ends.

Solution: push the next value of `%rip` onto the stack. Then, call the function. When function execution is finished, put this value back into `%rip` and continue executing.

e.g., `main()` calls `fun()`:

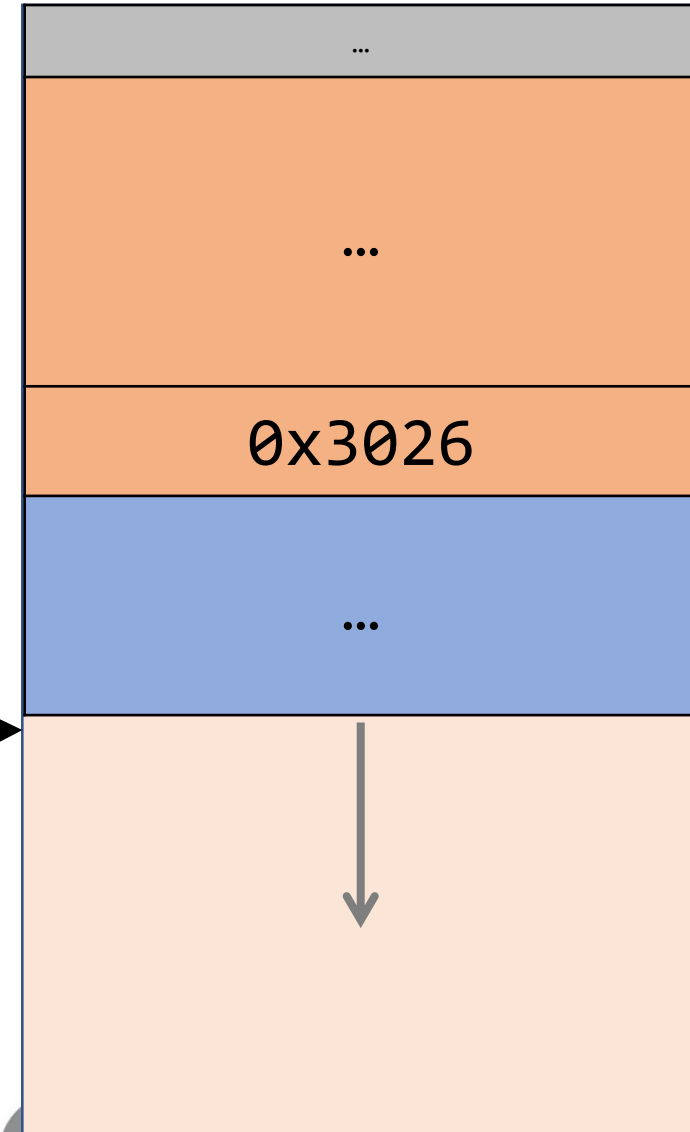
`main()`

`fun()`

`%rsp` 0xff08

`%rip` 0x4058

Stack



Remembering where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the next caller instruction to go back to after the function ends.

Solution: push the next value of `%rip` onto the stack. Then, call the function. When function execution is finished, put this value back into `%rip` and continue executing.

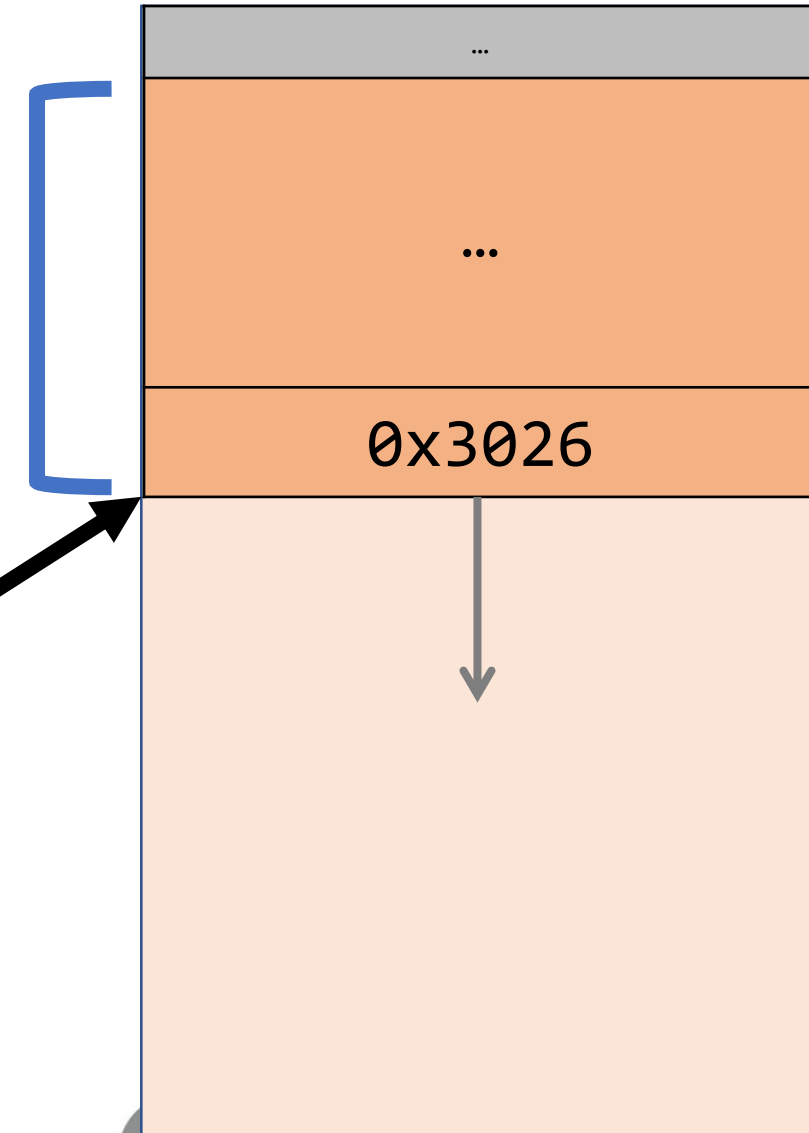
e.g., `main()` calls `fun()`:

`main()`

`%rsp` 0xff18

`%rip` 0x3021

Stack



Remembering where We Left Off

Problem: `%rip` points to the next instruction to execute. To call a function, we must remember the next caller instruction to go back to after the function ends.

Solution: push the next value of `%rip` onto the stack. Then, call the function. When function execution is finished, put this value back into `%rip` and continue executing.

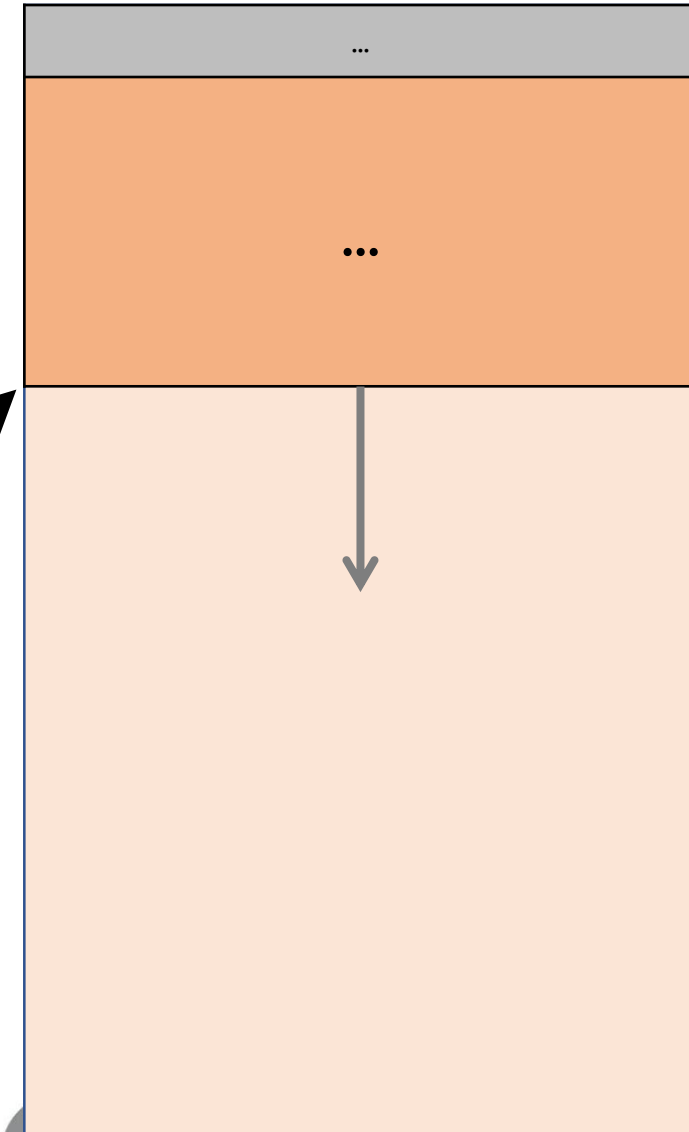
e.g., `main()` calls `fun()`:

`main()`

`%rsp` 0xff20

`%rip` 0x3026

Stack



Call and Return Instructions

- The **call** instruction pushes the address of the instruction immediately following the **call** instruction onto the stack and sets **%rip** to point to the beginning of the specified function's instructions.

```
call Label  
call *Operand
```

The **ret** instruction pops this instruction address from the stack and stores it in **%rip**.

```
ret
```

The stored **%rip** value for a function is called its **return address**. It is the address of the instruction at which to resume the function's execution (not to be confused with **return value**, which is the value returned from a function).

Parameters and Return (gcc-specific)

- There are special registers that store parameters and the return value.
- To call a function, we must put any parameters we are passing into the correct registers (i.e., `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`).
- Parameters beyond the first 6 are put on the stack.
- If the **caller** expects a return value, it looks in `%rax` after the **callee** completes.

Local Storage

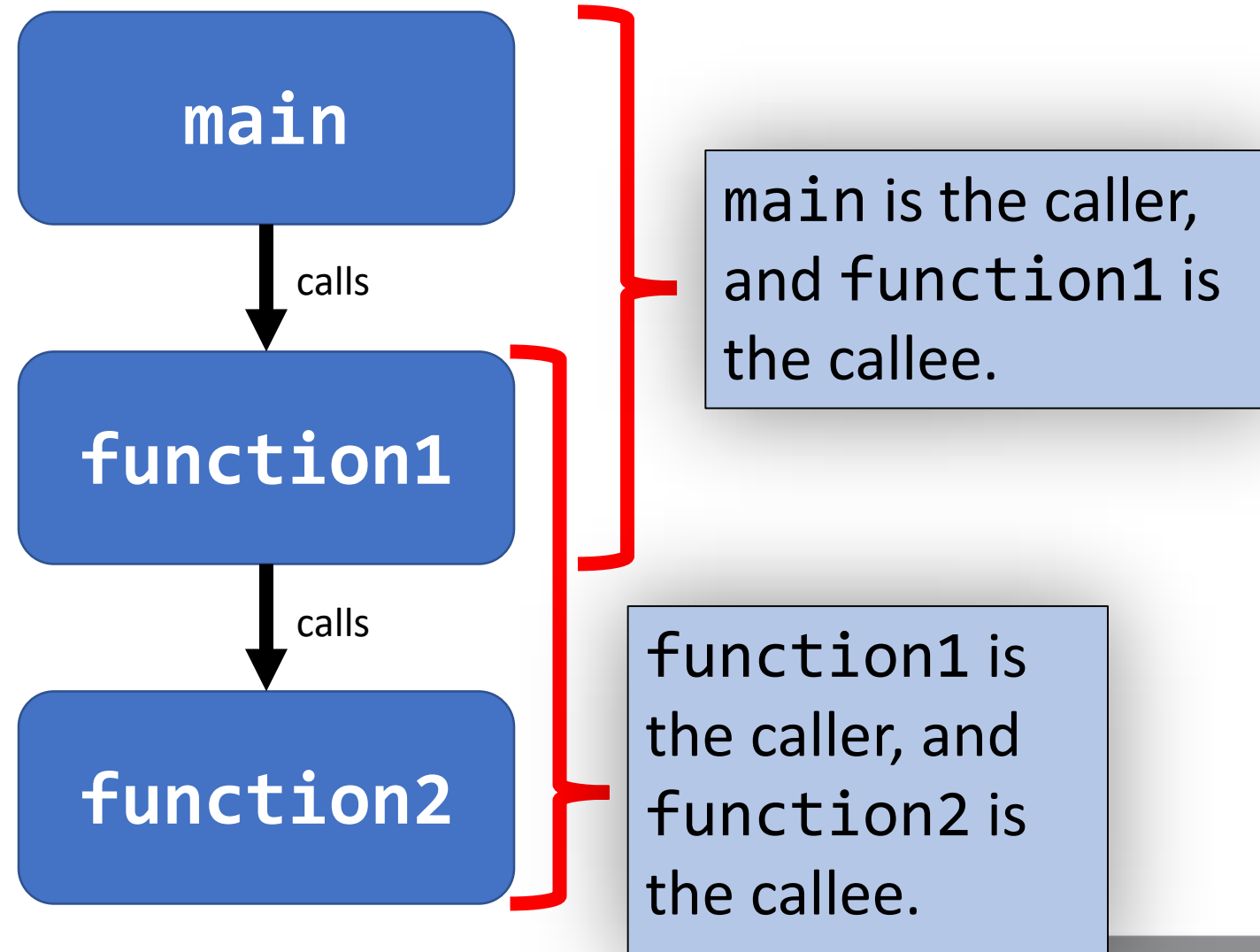
- We have often seen local variables stored directly in registers, rather than on the stack as we would expect. This is for optimization reasons.
- There are **three** common reasons that local data must be in memory:
 1. We've run out of registers.
 2. The '&' operator is used on it, so we must generate an address for it.
 3. They are arrays or structs (need to use address arithmetic).

Register Restrictions

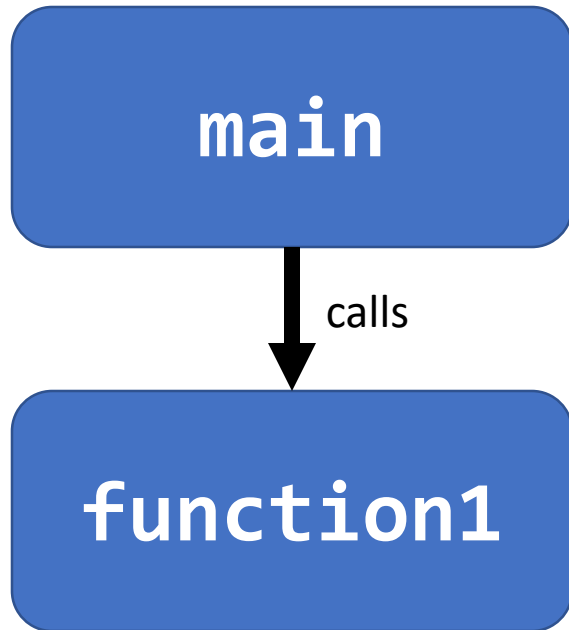
- There is only one copy of registers for all programs and functions.
- **Problem:** what if *fun1* is building up a value in register `%r10`, and calls *fun2* in the middle, which also has instructions that modify `%r10`? *fun1*'s value will be overwritten!
- **Solution:** make some agreement that callers and callees must follow when using registers so they do not interfere with one another.
- These rules define two types of registers: **caller-owned** and **callee-owned**

Caller – Callee

Caller/callee is terminology that refers to a pair of functions. A single function may be both a caller and callee simultaneously (e.g. `function1` at right).



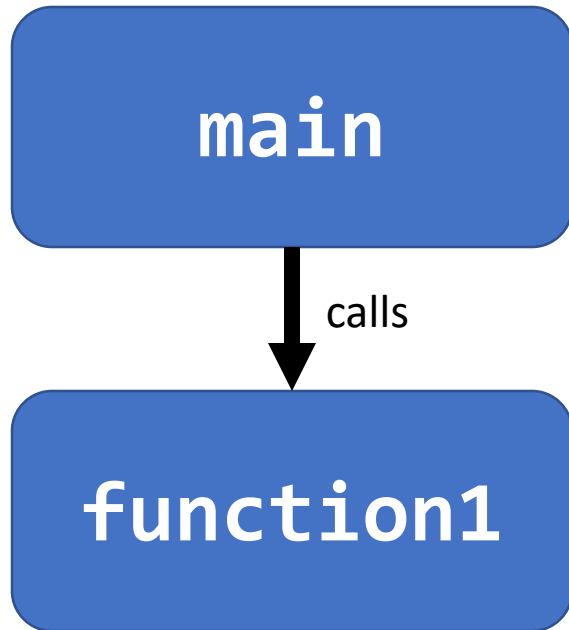
Caller-Owned Registers



`main` can use caller-owned registers and know that `function1` will not permanently modify their values.

If `function1` wants to use any caller-owned registers, it must save the existing values and restore them before returning.

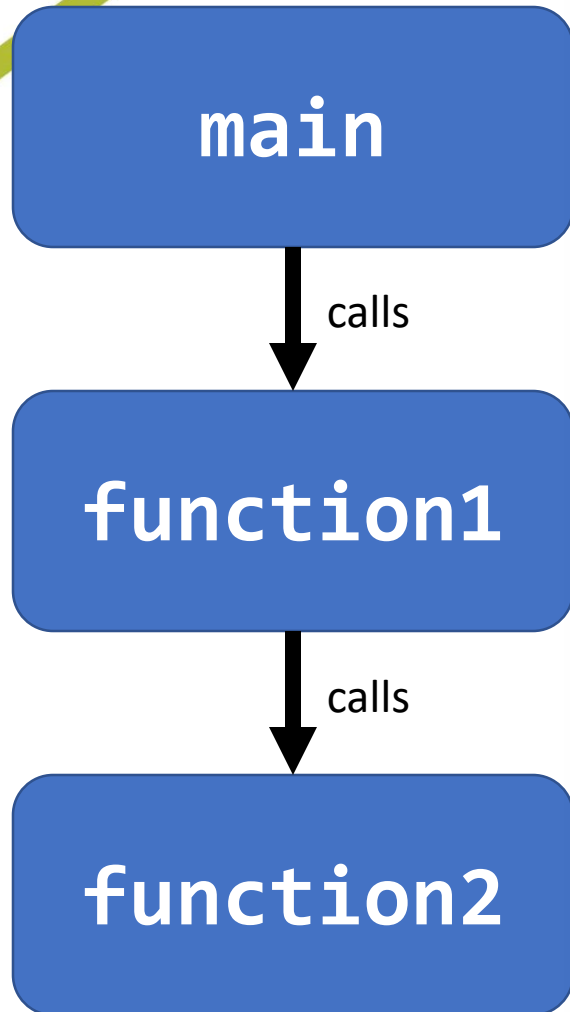
Callee-Owned Registers



`main` can use callee-owned registers but calling `function1` may permanently modify their values.

If `function1` wants to use any callee-owned registers, it can do so without saving the existing values.

The Rules of the Road



Caller-owned registers:

- **function1** must save/restore existing values of any it wants to use.
- **function1** can assume that calling **function2** will not permanently change their values.

Callee-owned registers:

- **function1** does not need to save/restore existing values of any it wants to use.
- calling **function2** may permanently change their values.



DOUGLAS COLLEGE

Floating Point Representation

Base 2 conversion

Convert the following number/fractions to base 10 (decimal) and base 2.

	<u>Number/fraction</u>	<u>Base 10</u>	<u>Base 2</u>
1.	1/2	0.5	0.1
2.	5		
3.	9/8		
4.	1/3		
5.	1/10 (bonus)		

Base 2 conversion

Convert the following number/fractions to base 10 (decimal) and base 2.

	<u>Number/fraction</u>	<u>Base 10</u>	<u>Base 2</u>
1.	$1/2$	0.5	0.1
2.	5	5.0	101.0
3.	$9/8$	1.125	1.001
4.	$1/3$		
5.	$1/10$ (bonus)		

Base 2 conversion

Convert the following number/fractions to base 10 (decimal) and base 2.

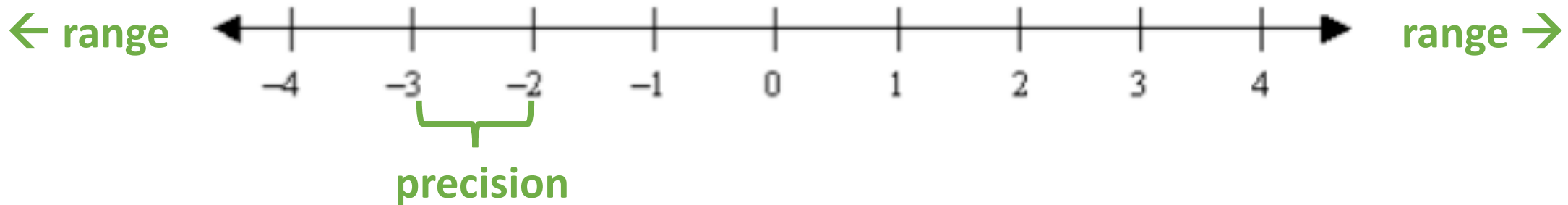
	<u>Number/fraction</u>	<u>Base 10</u>	<u>Base 2</u>
1.	1/2	0.5	0.1
2.	5	5.0	101.0
3.	9/8	1.125	1.001
4.	1/3	0.3333...	0.01010101...
5.	1/10 (bonus)	0.1	0.00011001100...

Conceptual goal: How can we represent real numbers with a ***fixed*** number of bits?

Learning goal: Appreciate the IEEE floating point format!

Approximating Real Numbers

How can we represent real numbers with a ***fixed*** number of bits?



In the world of real numbers:

- The real number line extends forever (infinite **range**).
- Real numbers have infinite resolution (infinite **precision**).

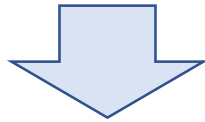
In the discrete world of computing, we must **approximate**:

- Each variable type is fixed width (`float`: 32 bits, `double`: 64 bits).
- Compromises are inevitable (**range** and **precision**). Like with `int`, we need to make choices about which numbers make the cut and which don't.

Approximating Real Numbers

- Base 10, decimal case:

5934.216123121...₁₀

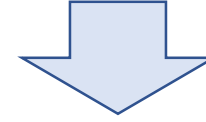


5 9 3 4 . 2 1 6 7

10^3 10^2 10^1 10^0 10^{-1} 10^{-2} 10^{-3} 10^{-4}

- Base 2, binary case:

1011.0101010101...₂



1 0 1 1 . 0 1 0 1

2^3 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- Decide on fixed granularity, e.g., 1/16
- Assign bits to represent powers from 2^3 to 2^{-4}

Local Storage

- **Problem:** We must fix where the decimal point is in our representation. This fixes our **precision**.

$$\begin{array}{ccc}
 \text{(base 10)} & 6.022e23 = & \underbrace{11 \dots 0.0}_{79 \text{ bits}} \text{ (base 2)}
 \end{array}$$

$$\begin{array}{ccc}
 6.626e-34 = & \underbrace{0.0 \dots 01}_{111 \text{ bits}}
 \end{array}$$

To store both these numbers in the same fixed-point representation, the bit width of the type would need to be at least 190 bits wide!

Scientific Notation to The Rescue

We have a need for **relative** rather than absolute precision.

How much error/approximation is tolerable? Radius of atom, football, planet?

Consider for decimal values:

3,650,123

→

3.65 x 10⁶

0.0000072491

→

7.25 x 10⁻⁶



1 digit for
exponent

3 digits for mantissa (and round)

As a datatype, store mantissa and exponent separately.

Allocations of bits to exponent and mantissa (respectively) determines range and precision.



IEEE Floating Point Standard

IEEE Standard 754

Established in 1985 as a uniform standard for floating point arithmetic .

Supported by all major systems today.

Hardware: specialized co-processor vs. integrated into main chip.

Driven by numerical concerns

Behavior defined in mathematical terms.

Clear standards for rounding, overflow, underflow.

Support for transcendental functions (roots, trig, exponentials, logs).

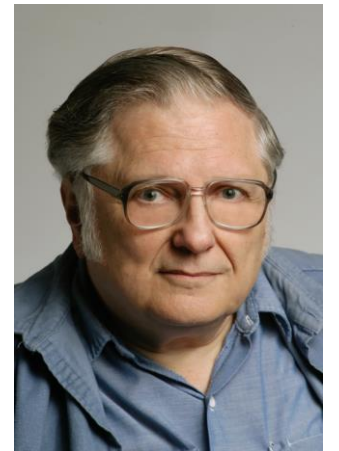
Hard to make fast in hardware.

Numerical analysts predominated over hardware designers in defining standard.



I think that it is nice to have at least one example—and the floating-point standard is one—where sleaze did not triumph.

— Will Kahan
(chief architect of standard)



IEEE Floating Point Standard

- Let's aim to represent numbers of the following scientific-notation-like format:

$$V = (-1)^S \times M \times 2^E$$

Sign bit, s :
negative ($s = 1$)
positive ($s = 0$)

Mantissa, M :
Significant digits,
also called
significand

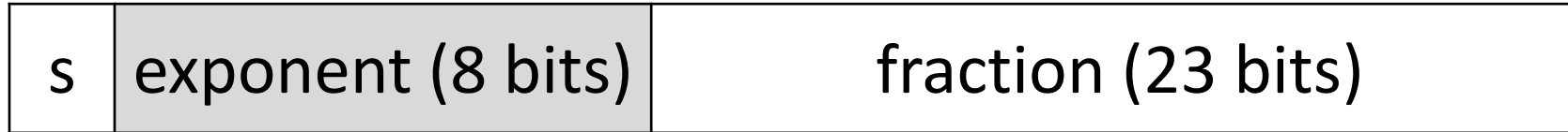
Exponent, E :
Scales value by
(possibly negative)
power of 2

Example for the Next Few Slides

What is the number represented by the following 32-bit float?

s	exponent (8 bits)	fraction (23 bits)
0	1000 0000	0100 0000 0000 0000 0000 000

Exponent



exponent (Binary)	<i>E</i> (Base 10)
11111111	RESERVED
11111110	127
11111101	126
11111100	125
...	...
00000011	-124
00000010	-125
00000001	-126
00000000	RESERVED

special

normalized

denormalized

Exponent



exponent (Binary)	<i>E</i> (Base 10)
11111111	RESERVED
11111110	127
11111101	126
11111100	125
...	...
00000011	-124
00000010	-125
00000001	-126
00000000	RESERVED

Not 2's complement

$E = \text{exponent} - \text{bias}$,
 where **float bias** $= 2^{8-1} - 1 = 127$

Exponents are sequentially represented starting from 000...1 (most negative) to 111...10 (most positive).

Bit-level comparison is fast!

Fraction: Normalized Values



We could just encode whatever M is in the fraction field (f). But there's a trick we can use to make the most out of the bits we have...

$$M = 1. [\text{fraction bits}]$$



Scientific Notation to the Rescue Again!

Correct scientific notation:

In the mantissa, always keep one non-zero digit to the left of the decimal point.

For base 10:

$$42.4 \times 10^5 \rightarrow 4.24 \times 10^6$$

$$324.5 \times 10^5 \rightarrow 3.245 \times 10^7$$

$$0.624 \times 10^5 \rightarrow 6.24 \times 10^4$$

For base 2:

$$10.1 \times 2^5 \rightarrow 1.01 \times 2^6$$

$$1011.1 \times 2^5 \rightarrow 1.0111 \times 2^8$$

$$0.110 \times 2^5 \rightarrow 1.10 \times 2^4$$

Observation: in base 2, this means there is *always* a 1 to the left of the decimal point!

Fraction: Normalized Values

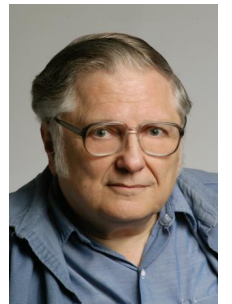


We could just encode whatever M is in the fraction field (f). But there's a trick we can use to make the most out of the bits we have...

$$M = 1. [\text{fraction bits}]$$

An “implied leading 1” representation

Means: we get one additional bit of precision for free!



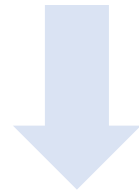
Thanks, Velvel!

Example

What is the number represented by the following 32-bit float?

s	exponent (8 bits)	fraction (23 bits)
0	1000 0000	0100 0000 0000 0000 0000 000

Subtract bias
($2^{8-1} - 1 = 127$)



$$E = 128 - 127 = 1$$

Add implicit 1



$$\begin{aligned} M &= (1.01)_2 \\ &= 1 + 0 \times 2^{-1} + 1 \times 2^{-2} = 1.25 \end{aligned}$$

$$V = (-1)^0 \times 1.25 \times 2^1 = 2.5$$

In-Class Practice #1

s	exponent (8 bits)	fraction (23 bits)
0	0111 1110	0000 0000 0000 0000 0000 000

1. Is this number:
A. Greater than 0?
B. Less than 0?
2. Is this number:
A. Less than -1?
B. Between -1 and 1?
C. Greater than 1?
3. What is the number?

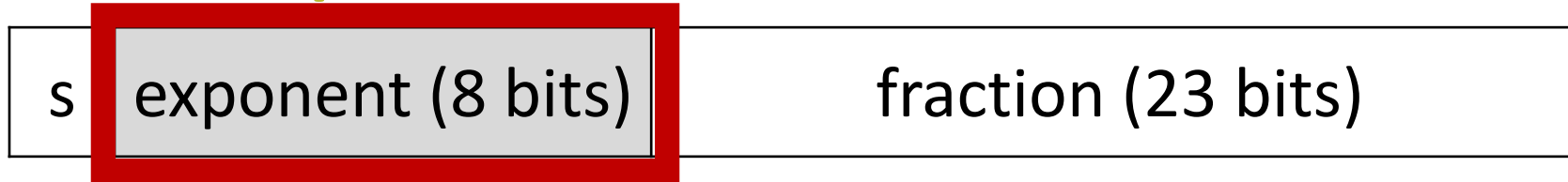


In-Class Practice #1

s	exponent (8 bits)	fraction (23 bits)
0	0111 1110	0000 0000 0000 0000 0000 000

- Is this number:
 - Greater than 0?**
 - Less than 0?
- Is this number:
 - Less than -1?
 - Between -1 and 1?**
 - Greater than 1?
- What is the number? $(-1)^0 \times 1.0 \times 2^{-1} = 0.5$

Reserved Exponent Values



exponent (Binary)	E (Base 10)
11111111	RESERVED
11111110	127
11111101	126
11111100	125
...	...
00000011	-124
00000010	-125
00000001	-126
00000000	RESERVED

special

normalized

denormalized

All Zeros: Zero and Denormalized Floats $V = (-1)^s \times M \times 2^E$

Zero (+0, -0)

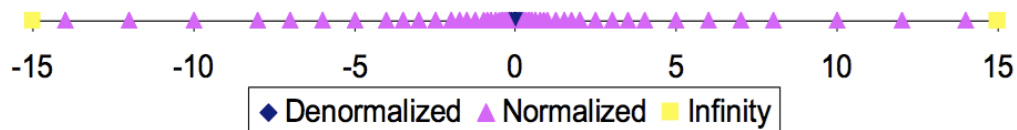
s	exponent (8 bits)	fraction (23 bits)
any	0000 0000	all zeros

Denormalized floats:

s	exponent (8 bits)	fraction (23 bits)
any	0000 0000	any nonzero

Smallest normalized exponent: $E = 1 - \text{bias} = -126$

Mantissa has **no leading one**: $M = 0.$ [fraction bits]



Denormalized values enable gradual **underflow** (too-small-to-represent floats).

All Ones: Infinity and NaN

Infinity (+inf, -inf)

s	exponent (8 bits)	fraction (23 bits)
any	1111 1111	all zeros

Floats have built-in handling of overflow:
infinity + anything = infinity.

Not a number (**NaN**):

s	exponent (8 bits)	fraction (23 bits)
any	1111 1111	any nonzero

Computation results in an
invalid mathematical real number.

Examples: $\text{sqrt}(x)$ (i.e., $\sqrt[2]{x}$), where x is negative, $\frac{\infty}{\infty}$, $\infty + (-\infty)$, etc.

Example Numbers (Blackboard Table)

- The table on Blackboard shows some examples for a hypothetical 8-bit floating-point format having $k = 4$ exponent bits and $n = 3$ fraction bits.
- The bias is $2^{k-1} - 1 = 7$. The table is divided into three regions representing the three classes of numbers.
- The different columns show how the exponent field encodes the exponent E , while the fraction field encodes the significand M , and together they form the represented value $V = 2^E \times M$.
- Note the smooth transition between the largest denormalized number $7/512$ and the smallest normalized number $8/512$.
- This smoothness is due to the definition of E for denormalized values.
- By making E equal to $1 - \text{Bias}$ (rather than $-\text{Bias}$), we compensate for the fact that the significand of a denormalized number does not have an implied leading 1.

float vs. double

float (32 bits)

- 8-bit exponent ranges from -126 to +127,
 $2^{127} = 10^{37}$

s	exponent (8 bits)	Fraction (23 bits)

double (64 bits)

- 11-bit exponent ranges from -1022 to +1023,
 $2^{1023} = 10^{308}$

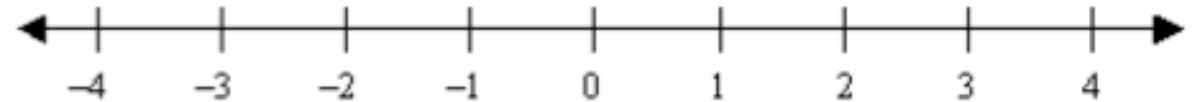
s	exponent (11 bits)	Fraction (52 bits)

float vs. int

32-bit integer (type **int**): :
-2,147,483,648 to 2147483647

64-bit integer (type **long**):
-9,223,372,036,854,775,808
to 9,223,372,036,854,775,807

All integers in these ranges can be represented.

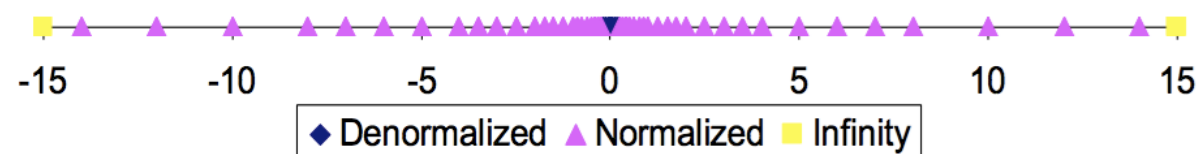


32-bit floating point (type **float**):
 $\sim 1.7 \times 10^{-38}$ to $\sim 3.4 \times 10^{38}$ (+ negative range)

64-bit floating point (type **double**):
 $\sim 9 \times 10^{-307}$ to $\sim 1.8 \times 10^{308}$ (+ negative range)

Not all numbers can be represented.
Gaps can get quite large: larger the exponent, larger the gap between successive fraction values.

(normalized float/double ranges)



Important Notes

- Approximation and rounding in floating point computations are inevitable.
- Single operations are commutative, but sequence is not associative.

$(a + b)$ equals $(b + a)$
But $(a + b) + c$ may **not** equal $a + (b + c)$

- Equality comparison operations are (often) inaccurate.

Important Notes

- Approximation and rounding in floating point computations are inevitable.
- Single operations are commutative, but sequence is not associative.

$(a + b)$ equals $(b + a)$
But $(a + b) + c$ may not equal $a + (b + c)$

- Equality comparison operations are (often) inaccurate.

Example: “minifloat”

For a more compact example representation, we will use an 8 bit “minifloat” with a 4 bit exponent, 3 bit fraction and bias of 7 (minifloat hypothetical and is not a real datatype).



Example: “minifloat”

With minifloat, $128 + 4$ **would not equal 132**. Why not?

$$\begin{array}{r} 128 \\ + \quad 4 \\ \hline 128? \end{array}$$

Let's step through the calculations to add these two numbers (note: this is just for understanding; real float calculations are more efficient).

Example: “minifloat”

128	0	1110	000
+ 4	0	1001	000
144?	0	1110	001

We don't have enough bits to represent values between 128 and 144!

The *next-largest minifloat* that can be represented after 128 is **144**.
132 isn't representable!

144:	0	1110	001

Key Idea: the smallest float “hop” we can take is to adjust the fractional component by 1.

Important Notes

- Approximation and rounding in floating point computations are inevitable.
- Single operations are commutative, but sequence is not associative.

$(a + b)$ equals $(b + a)$
But $(a + b) + c$ may **not** equal $a + (b + c)$

- Equality comparison operations are (often) unwise.

Floating Point Arithmetic

Is this just over/underflowing? It turns out it's more subtle.

```
float a = 3.14;
float b = 1e20;
printf("(3.14 + 1e20) - 1e20 = %g\n", (a + b) - b); // 0
printf("3.14 + (1e20 - 1e20) = %g\n", a + (b - b)); // 3.14
```

Floating point arithmetic is not associative. The order of operations matters!

The first line loses precision when first adding 3.14 and 1e20 (as we have seen).

The second line first evaluates $1e20 - 1e20 = 0$, and then adds 3.14.

%e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
%E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
%g	Use the shortest representation: %e or %f	

Important Notes

- Approximation and rounding in floating point computations are inevitable.
- Single operations are commutative, but sequence is not associative.

$(a + b)$ equals $(b + a)$
But $(a + b) + c$ may not equal $a + (b + c)$

- Equality comparison operations are (often) inaccurate. See example.



DOUGLAS COLLEGE

Parallel Programming

Why Parallel Programming

- It may speed up our code.
- Today, our computers have one or more CPUs that have multiple processing cores (Multi-core processor).
- This helps with desktop computing tasks like multitasking (running multiple programs, plus the operating system, simultaneously).
- For scientific computing, this means you have the ability in principle of splitting up our computations into groups and running each group on its own processor.

Types of Parallel Tasks

- We can separate a computation into three groups depending on how it can be parallelized:
 - **embarrassingly parallel**: problem is one for which it is very easy to separate it into some number of independent tasks that then may be run in parallel.
 - An example of an embarrassingly parallel problem is when we need to run a preprocessing pipeline on datasets collected for 15 subjects.
 - Each subject's data can be processed independently of the others.
 - In other words, the computations involved in processing one subject's data do not in any way depend on the results of the computations for processing some other subject's data.

Types of Parallel Tasks

- We can separate a computation into three groups depending on how it can be parallelized:
 - **Serial**: problem cannot be split into independent sub-problems.
 - The computation at one stage does depend on the results of a computation at an earlier stage, and so it is not so easy to parallelize across independent processing units.
 - In these kinds of problems, there is a need for some communication or coordination between sub-tasks.
 - Time dependent differential equations are examples of serial problems.

Types of Parallel Tasks

- We can separate a computation into three groups depending on how it can be parallelized:
 - **Mixtures**: problems that has both embarrassingly parallel properties as well as serial dependency properties.
 - Example: the computations involved in training and running an artificial neural network (ANN).
 - An ANN is made up of several layers of neuron-like processing units, each layer having many (even hundreds or thousands) of these units.
 - If the ANN is a pure feedforward architecture, then computations within each layer are embarrassingly parallel, while computations between layers are serial.

The Threads Model for Parallel Programming

- The threads model of parallel programming is one in which a single process (a single program) can spawn multiple, concurrent "threads" (sub-programs).
- Each thread runs independently of the others, although they can all access the same shared memory space (and hence they can communicate with each other if necessary).
- Threads can be spawned and killed as required, by the main program.
- Threads are a way that a program can spawn concurrent units of processing that can then be delegated by the operating system to multiple processing cores.
- The advantage of a multithreaded program is that you can achieve big speedups, as all cores of your CPU (and all CPUs if you have more than one) are used at the same time.

OpenMP: A Tool for Parallel Programming

- OpenMP is an API that implements a multi-threaded, shared memory form of parallelism.
- It uses a set of compiler directives (statements that we add to our C code) that are incorporated at compile-time to generate a multi-threaded version of your code.

```
#include <omp.h>
main () {
int var1, var2, var3;
Serial code
    .
    .
    .

Beginning of parallel section. Fork a team of threads.
Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
{
    Parallel section executed by all threads
        .
        .
        .

    All threads join master thread and disband
}
Resume serial code
    .
    .
    .
}
```

Private vs. Shared variables

- By using the **private()** and **shared()** directives, we can specify variables within the parallel region as being:
 - shared, i.e. visible and accessible by all threads simultaneously, or
 - private, i.e. private to each thread, meaning each thread will have its own local copy.
- In the code example next for parallelizing a for loop, we specify the **thread_id** and **nloops** variables as **private**.
- By default, loop iteration counters in OpenMP loop constructs (in this case, the **i** variable) in the **for**-loop are set to **private** variables.

```
// gcc -fopenmp parallel.c -o parallel.exe
#include <stdio.h>
#include <omp.h>

int main(int argc, char **argv)
{
    int i, thread_id, nloops;

#pragma omp parallel private(thread_id, nloops)
    {
        nloops = 0;

#pragma omp for
        for (i=0; i<1000; ++i)
        {
            ++nloops;
        }

        thread_id = omp_get_thread_num();

        printf("Thread %d performed %d iterations of the loop.\n",
            thread_id, nloops );
    }

    return 0;
}
```

Critical Code

- In OpenMP, we can specify something called a "critical" section of code.
- This is code that is performed by all threads, but is only performed one thread at a time (i.e. in serial).
- This provides a convenient way of letting us update a global variable with local results from each thread.
- We don't have to worry about other threads writing to that global variable at the same time (a collision).

```

#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{
    int i, thread_id;
    int glob_nloops, priv_nloops;
    glob_nloops = 0;

    // parallelize this chunk of code
    #pragma omp parallel private(priv_nloops, thread_id)
    {
        priv_nloops = 0;
        thread_id = omp_get_thread_num();

        // parallelize this for loop
        #pragma omp for
        for (i=0; i<1000000; ++i)
        {
            ++priv_nloops;
        }

        // make this a "critical" code section
        #pragma omp critical
        {
            printf("Thread %d is adding its iterations (%d) to sum (%d), ",
                   thread_id, priv_nloops, glob_nloops);
            glob_nloops += priv_nloops;
            printf(" total nloops is now %d.\n", glob_nloops);
        }
    }
    printf("Total # loop iterations is %d\n",
           glob_nloops);
    return 0;
}

```

Reduction

- Reduction refers to the process of combining the results of several sub-calculations into a final result.
- This is a very common paradigm (also known as map-reduce).
- We use this paradigm in the code example next, where we use the "critical code" directive to accomplish this.
- The map-reduce paradigm is so common that OpenMP has a specific directive that allows us to more easily implement this.

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

int main(int argc, char *argv[])
{
    int i, thread_id;
    int glob_nloops, priv_nloops;
    glob_nloops = 0;

    // parallelize this chunk of code
    #pragma omp parallel private(priv_nloops, thread_id) reduction(+:glob_nloops)
    {
        priv_nloops = 0;
        thread_id = omp_get_thread_num();

        // parallelize this for loop
        #pragma omp for
        for (i=0; i<100000; ++i)
        {
            ++priv_nloops;
        }
        glob_nloops += priv_nloops;
    }
    printf("Total # loop iterations is %d\n",
        glob_nloops);
    return 0;
}
```


References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.



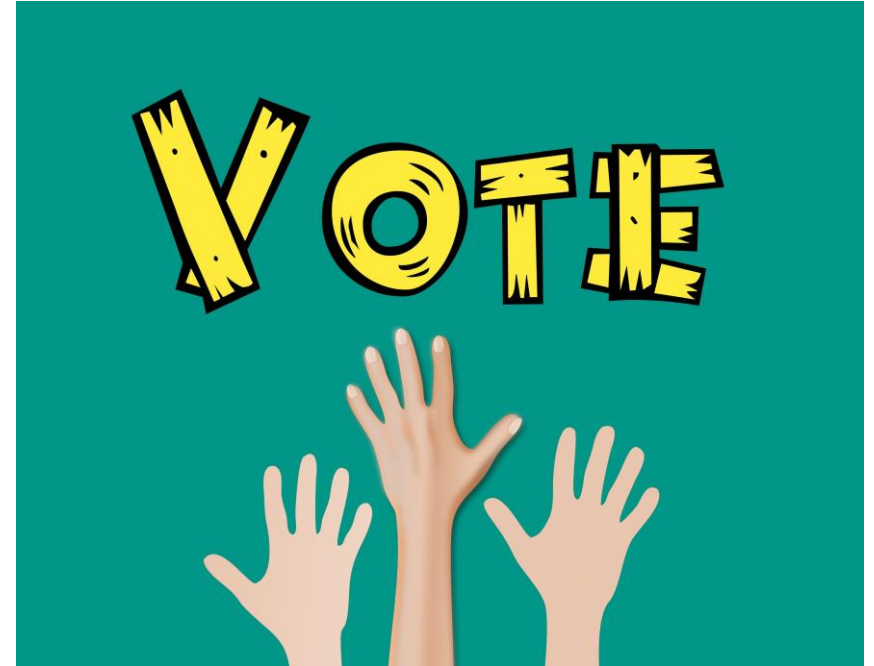
Thank you.
DOUGLAS COLLEGE

DOUGLAS COLLEGE

Poll 1 (Extra Credit)

Which one of the following choices is not part of the von Neumann's computer architecture design?

- a) Data and instructions must be stored in separate memory units.
- b) The main memory's contents must be addressable.
- c) Execution of instructions is sequential unless there are jump conditions.



Please use the “Poll” window to participate for extra credit! One answer only please!

Plan for Today

- Bits and Bytes
- The Binary Number System
- The Hexadecimal Number System
- Integer Representations
- Unsigned Integers
- Signed Integers
- Overflow
- Expansion and Truncation of Bit Representations

Bits and Bytes

A “Bit” Confused?

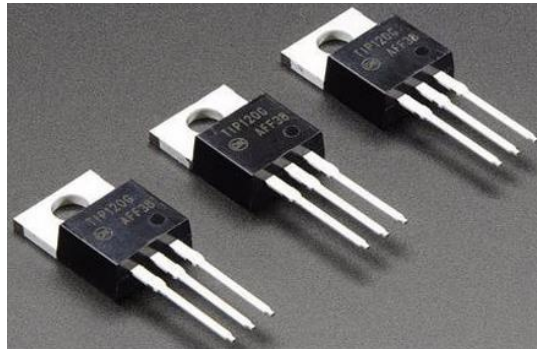
0

Bit = Binary-digit

1

Bits

- Computer systems are built based on the concept of two states: “on” and “off”.
- This concept is represented in computer hardware by transistors, which can be described in the simplest form as switches that either allow current to pass through (on) or not (off).



- Binary-digits or bits represent the same concept in computer software: 0 means “off”, and 1 means “on”.

A Little “Bit” More...

- Bits can be combined together to represent more data.
- A **byte** is collection of 8 **bits**.
- In computer architecture, memory is just a very long array of bytes.
- We say computer memory is **byte-addressable**; this means that we cannot address (i.e., store the location of) a bit; **we can only store the location of a byte**.
- In other words, a byte is the smallest unit of memory that is addressable.
- Nevertheless, computers still operate on the basis of bits.
- Many different ways of representing data as bits have emerged:
 - Audio
 - Images
 - Text
 - Video
 - Models
 - ...



DOUGLAS COLLEGE

The Binary Number System

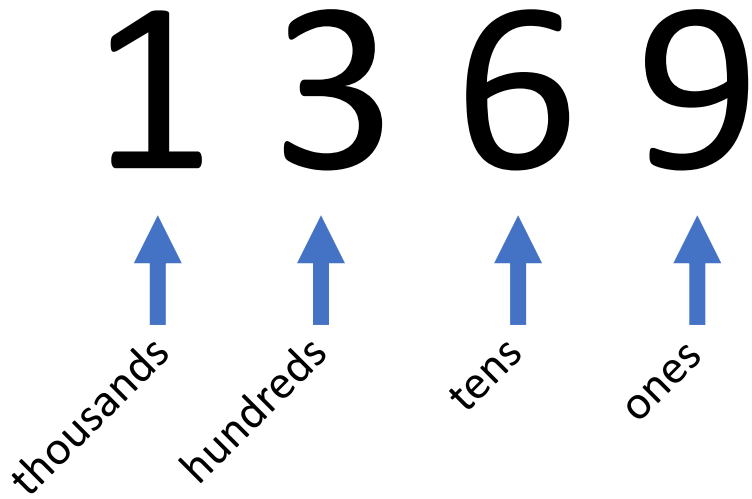
Base 10

1 3 6 9

Base 10 means numbers are represented with the following ten digits:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Base 10



$$= 1 \times 1000 + 3 \times 100 + 6 \times 10 + 9 \times 1$$

Base 10

1 3 6 9



10^3



10^2



10^1



10^0

Base 10

1 3 6 9

10^x :  3  2  1  0

Base 2

$$\begin{array}{cccc} 1 & 0 & 1 & 1 \\ \uparrow & \uparrow & \uparrow & \uparrow \\ 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

Base 2 (i.e., binary) means numbers are represented with the following 2 digits:
0 and 1

Base 10

Most significant bit

Least significant bit

1 0 1 1

eights

fours

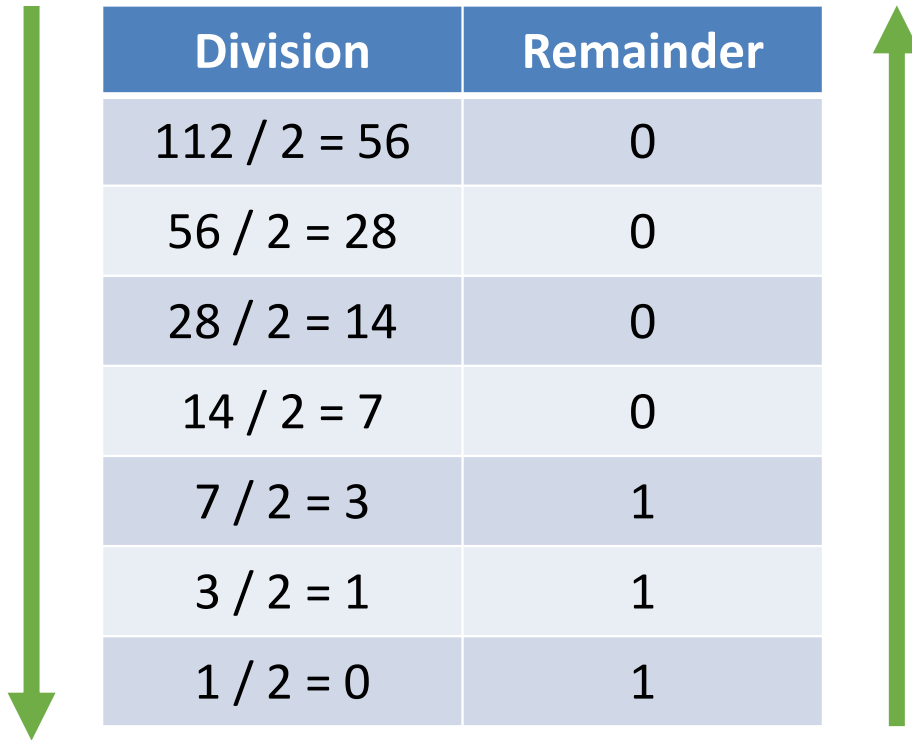
twos

ones

$$= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 11_{10}$$

How to Convert from Decimal to Binary

- Here is an example for 112:

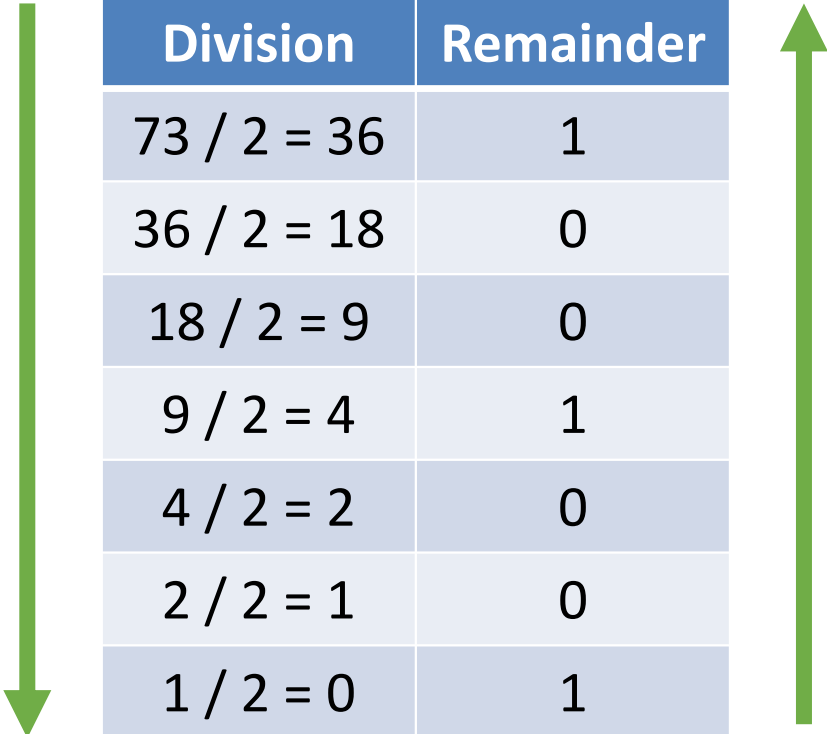


Division	Remainder
$112 / 2 = 56$	0
$56 / 2 = 28$	0
$28 / 2 = 14$	0
$14 / 2 = 7$	0
$7 / 2 = 3$	1
$3 / 2 = 1$	1
$1 / 2 = 0$	1

$$112_{10} = 1110000_2$$

How to Convert from Binary to Decimal

- Here is an example for 73:



Division	Remainder
$73 / 2 = 36$	1
$36 / 2 = 18$	0
$18 / 2 = 9$	0
$9 / 2 = 4$	1
$4 / 2 = 2$	0
$2 / 2 = 1$	0
$1 / 2 = 0$	1

$$73_{10} = 1001001_2$$

Byte Values

- What are the minimum and maximum base-10 values a single byte (8 bits) can store?

Minimum = 0

Maximum = ?

Byte Values

- What are the minimum and maximum base-10 values a single byte (8 bits) can store?

Minimum = 0

Maximum = ?

	1	1	1	1	1	1	1	
2^x :	7	6	5	4	3	2	1	0

$$1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 255$$

$$\text{Or more simply } 2^8 - 1 = 255$$

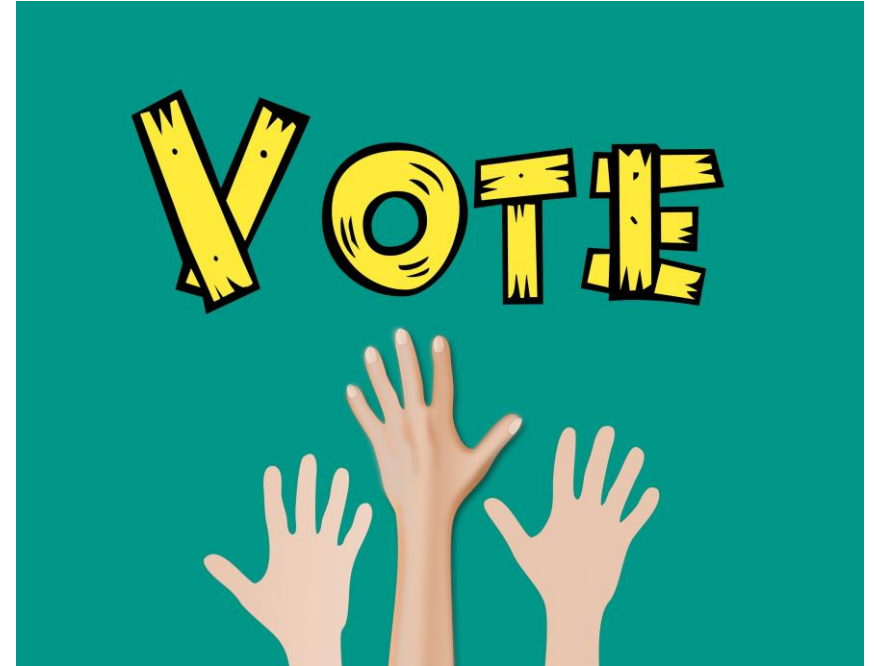
Poll 1 (Extra Credit)

Assuming an integer is represented by 1 byte, the following expression will lead to a compilation error:

$$11111111_2 + 00000001_2$$

- a) True
- b) False
- c) Not sure

Please use the “Poll” window to participate for extra credit! One answer only please!



Multiplying by Base

$$1450 \times 10 = 14500$$

$$1100_2 \times 2 = 11000_2$$

The Trick: inserting 0 at the end multiplies the value by the base.

Dividing by Base

$$1450 / 10 = 145$$

$$1100_2 / 2 = 110_2$$

The Trick: removing 0 at the end divides the value by the base.

Arithmetic Examples

$$\begin{array}{r} 10010001 \\ + 01001111 \\ \hline \end{array}$$

$$\begin{array}{r} 11010001 \\ - 10001110 \\ \hline \end{array}$$



DOUGLAS COLLEGE

The Hexadecimal Number System

The Hexadecimal Number System

- In computer architecture, we often work with large numbers that have 32 or 64 bits.
- To simplify the math (as well as the representation), we can use base-16 instead of base-10.
- Base-16 is called hexadecimal.
- Let us take the binary number 011010100011 as an example.

0110 1010 0011

0-15 0-15 0-15

- **The Trick:** each four-digit part represents a base-16 (hexadecimal) digit.

The Hexadecimal Number System

- Hexadecimal is base-16, so the digits we need to use are 0 – 15, but how can we do that?

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
										10	11	12	13	14	15

The Hexadecimal Number System

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

The Hexadecimal Number System

- In our course, we will use a convention to distinguish hexadecimal numbers.
- We will prefix hexadecimal numbers by prefixing them with **0x**, and binary numbers with **0b**.
- For instance, **0xF5** is **0b11110101**

0x F 5

 { {

 1111 0101

Hexadecimal to Binary Example

- Convert **0x173A** to binary.

Hexadecimal to Binary Example

- Convert **0x173A** to binary.

Hexadecimal	1	7	3	A
Binary	0001	0111	0011	1010

Hexadecimal to Binary Example

- What is **0b1111001010** in hexadecimal?

Hexadecimal to Binary Example

- What is **0b1111001010** in hexadecimal?

Binary	11	1100	1010
Hexadecimal	3	C	A

Arithmetic Examples

0x4490
+ 0x3333

0x1ACB
+ 0x2BCA

0x20A
*** 0x32**



DOUGLAS COLLEGE

Integer Representations

Number Representations

- **Unsigned Integers:** positive and 0 integers. (e.g. 0, 1, 2, ... 99999)
- **Signed Integers:** negative, positive, and 0 integers. (e.g. ...-2, -1, 0, 1,... 9999)
- **Floating Point Numbers:** real numbers (e.g., 0.1, -12.2, 1.5×10^{12})

Number Representations

C Declaration	Size (Bytes)
<code>int</code>	4
<code>double</code>	8
<code>float</code>	4
<code>char</code>	1
<code>char *</code>	8
<code>short</code>	2
<code>long</code>	8

What is 32-bit vs. 64-bit?

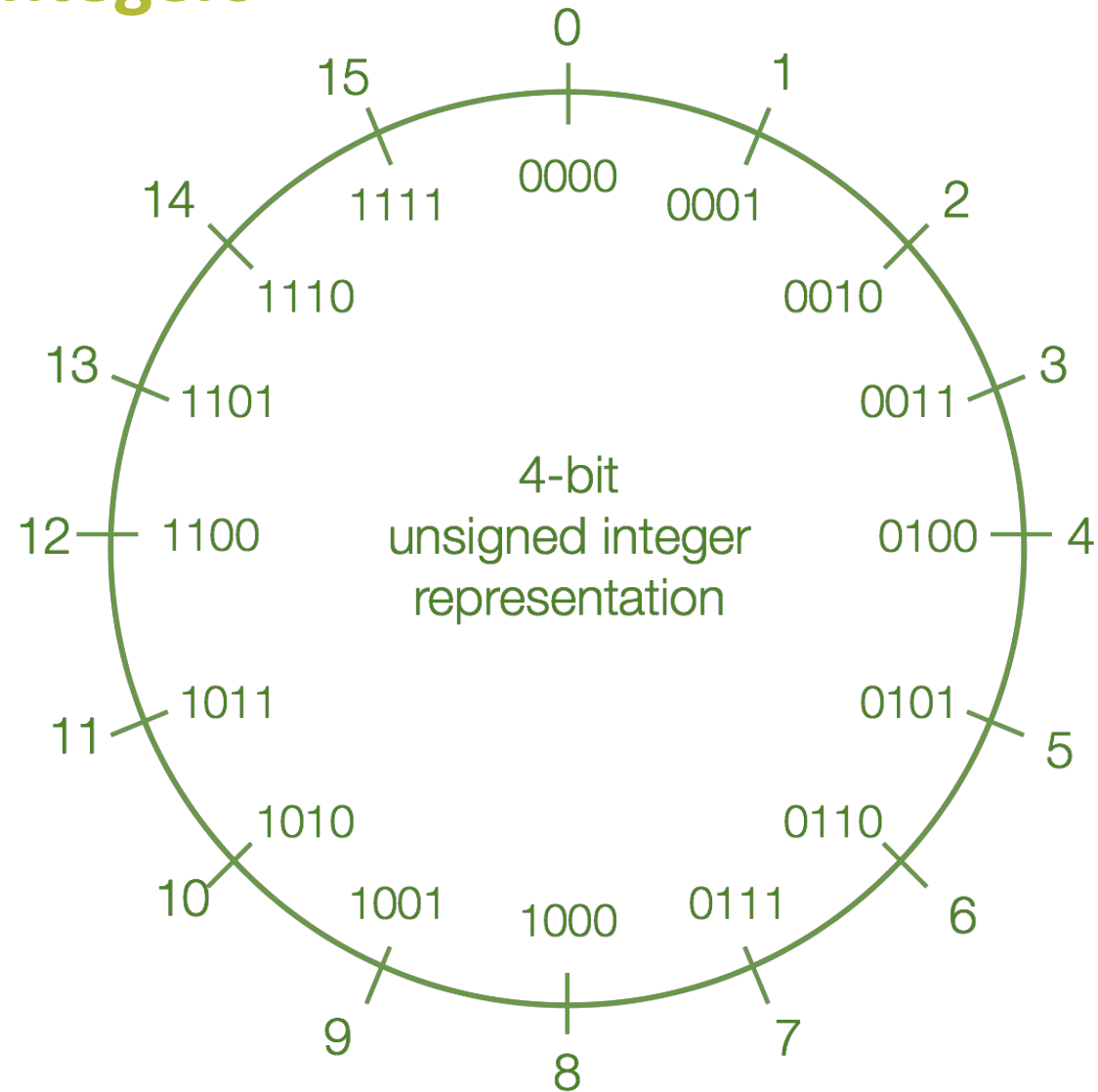
- Most computers in early 2000's were 32-bit. This means that pointers were 4 bytes (32 bits).
- 32-bit pointers store a memory address from 0 to $2^{32}-1$, which is equal to 2^{32} bytes of addressable memory. This equals 4 Gigabytes, meaning that 32-bit computers could have at most 4GB of memory (RAM).
- As a result, computers transitioned to 64-bit. This means that pointers in programs became 64 bits long.
- 64-bit pointers store memory addresses from 0 to $2^{64}-1$, which is equal to 2^{64} bytes of addressable memory. This equals $1024 \times 1024 \times 1024 \times 16$ GB of memory (RAM)!

Unsigned Integers

Unsigned Integers

- An unsigned integer is 0 or a positive integer (no negatives).
- We have already seen how to convert between decimal and binary. Examples:
 - $0b0001 = 1$
 - $0b0101 = 5$
 - $0b1011 = 11$
 - $0b1111 = 15$
- The range of an unsigned number is $0 \rightarrow 2^w - 1$, where w is the number of bits used to represent this an unsigned integer.
- As an example, a 32-bit integer can represent 0 to $2^{32} - 1$ (4,294,967,295).

Unsigned Integers



Signed Integers

How to Represent Negative Integers?

$$\begin{array}{r} 0101 \\ + \text{????} \\ \hline 0000 \end{array}$$

How to Represent Negative Integers?

$$\begin{array}{r} 0101 \\ + 1011 \\ \hline 0000 \end{array}$$

How to Represent Negative Integers?

$$\begin{array}{r} 0011 \\ + \text{????} \\ \hline 0000 \end{array}$$

How to Represent Negative Integers?

$$\begin{array}{r} 0011 \\ + 1101 \\ \hline 0000 \end{array}$$

How to Represent Negative Integers?

$$\begin{array}{r} 0000 \\ + \text{????} \\ \hline 0000 \end{array}$$

How to Represent Negative Integers?

$$\begin{array}{r} 0000 \\ + 0000 \\ \hline 0000 \end{array}$$

The Pattern

$$\begin{array}{r} 0101 \\ + 1011 \\ \hline 0000 \end{array}$$

Positive

0101



1010



1011

Invert Digits

Negative

Add 1

The Pattern

$$\begin{array}{r} 0011 \\ + 1101 \\ \hline 0000 \end{array}$$

Positive

0011



1100



1101

Invert Digits

Negative

Add 1

The Pattern

$$\begin{array}{r} 0000 \\ + 0000 \\ \hline 0000 \end{array}$$

Positive

0000



1111



0000

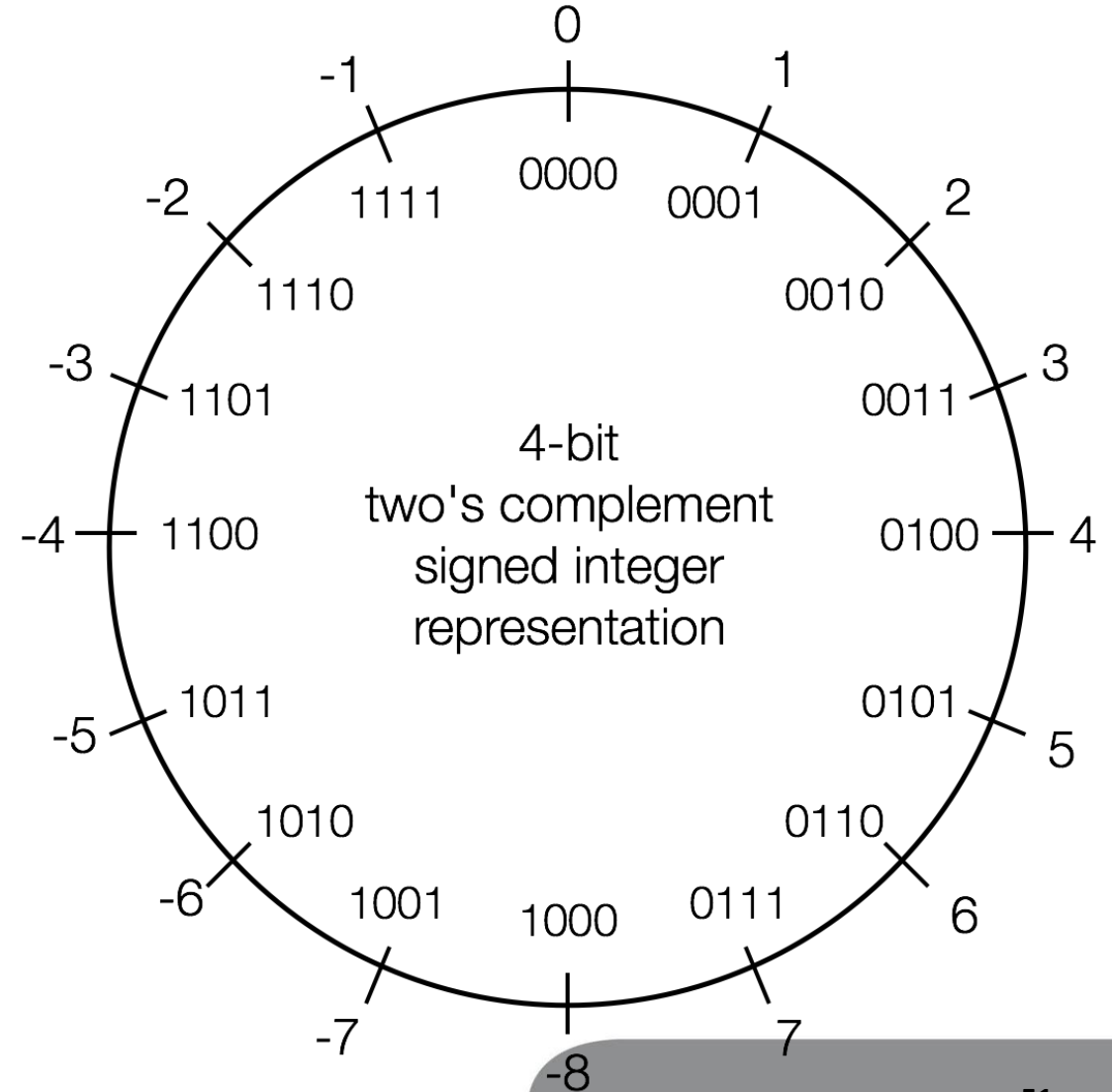
Invert Digits

Negative

Add 1

The Two's Complement Method for Signed Representation

- In two's complement, the negative equivalent of a positive integer is represented by the two's complement of that integer.
- **The two's complement of a number is the binary digits inverted, plus 1.**
- This works to convert from positive to negative, and back from negative to positive.
- **The most significant digit (i.e., the left-most digit) represents the sign of the number.**



The Two's Complement

- What is $2 + (-5)$?

$$\begin{array}{r} 0010 \\ + 1011 \\ \hline 1101 \end{array}$$

2
-5
-3

The Two's Complement

- How about $4 - 5$?

$$\begin{array}{r}
 0100 \quad 4 \\
 - 0101 \quad 5 \\
 \hline
 \text{????}
 \end{array}$$



$$\begin{array}{r}
 0100 \quad 4 \\
 + 1011 \quad -5 \\
 \hline
 1111 \quad -1
 \end{array}$$



DOUGLAS COLLEGE

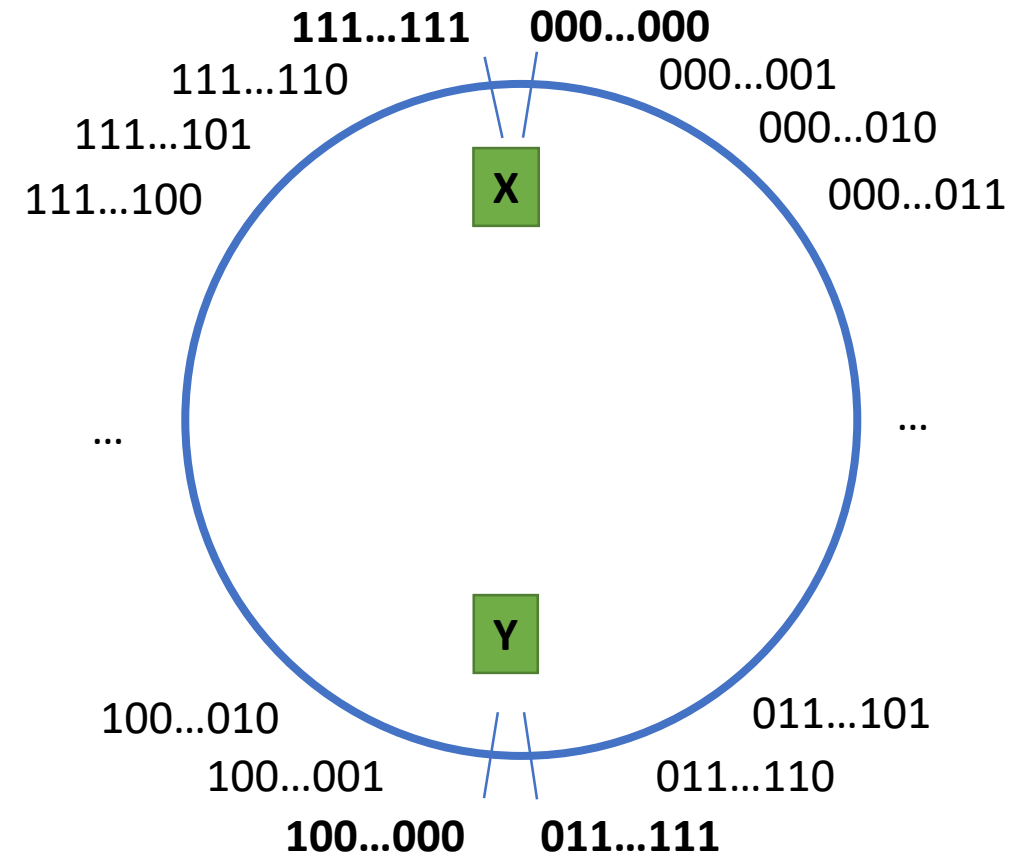
Overflow

Poll 1 (Extra Credit)

At which points can overflow occur for signed and unsigned int? (assume binary values shown are all 32 bits)

- A. Signed and unsigned can both overflow at points X and Y
- B. Signed can overflow only at X, unsigned only at Y
- C. Signed can overflow only at Y, unsigned only at X
- D. Signed can overflow at X and Y, unsigned only at X
- E. Other

Please use the “Poll” window to participate for extra credit! One answer only please!



Overflow

- If we exceed the maximum value of our bit representation, we overflow back to the smallest bit representation:

$$0b1111 + 0b1 = 0b0000$$

- If we go below the minimum value of our bit representation, we overflow back to the largest bit representation.

$$0b0000 - 0b1 = 0b1111$$

Minimum and Maximum Integer Values

Type	Bytes	Minimum	Maximum
char	1	-128	127
unsigned char	1	0	255
short	2	-32768	32767
unsigned short	2	0	65535
unsigned int	4	0	4294967295
long	8	-9223372036854775808	9223372036854775807
unsigned long	8	0	18446744073709551615

Overflow Example: Restart Your B787 Every 248 Days!





DOUGLAS COLLEGE

Expansion and Truncation

Bit Expansion and Truncation

- Sometimes, we want to convert between two integers of different sizes (e.g. **short** to **int**, or **int** to **long**).
- Conversion from a larger data type to a smaller one may lead to loss of data!
- We should, however, be able to convert from a smaller data type to a larger data type no matter what.
- For unsigned values, we can add leading zeros to the representation ("zero extension").
- For signed values, we can repeat the sign of the value for new digits ("sign extension").
- When using the `<`, `>`, `<=`, `>=` comparison operators between different size types, the C compiler will promote the smaller data type to the larger data type.
- If we attempt to cast a larger data type into a smaller one, the C compiler will drop the most significant bits from the left.

Bit Expansion Examples

```
short s = 4;
```

```
// short is a 16-bit format, so
```

```
s = 0000 0000 0000 0100b
```

```
int i = s;
```

```
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

```
short s = -4;
```

```
// short is a 16-bit format, so
```

```
s = 1111 1111 1111 1100b
```

```
int i = s;
```

```
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```



Thank you.
DOUGLASCOLLEGE

DOUGLASCOLLEGE

References

- Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Ha, 1988.
- Hanly, Jeri R. and Elliot B. Koffman, *C Program Design for Engineers*, Addison Wesley, 2001.
- Lafore, Robert, *Object-Oriented Programming in C++*, SAMS, 2001.
- Bryant, Randal and David R. O'Halloran, *Computer Systems: A Programmer's Perspective*, Pearson, 2019.
- Stallings, William, *Computer Organization and Architecture Designing for Performance*, Pearson, 2016.
- Benson, Andrew, *Computer Architecture Notes*, Stanford University, 2020.