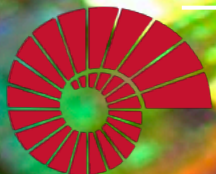


COMP201

Computer Systems & Programming

Lecture #18 – Introduction to x86-64 Assembly

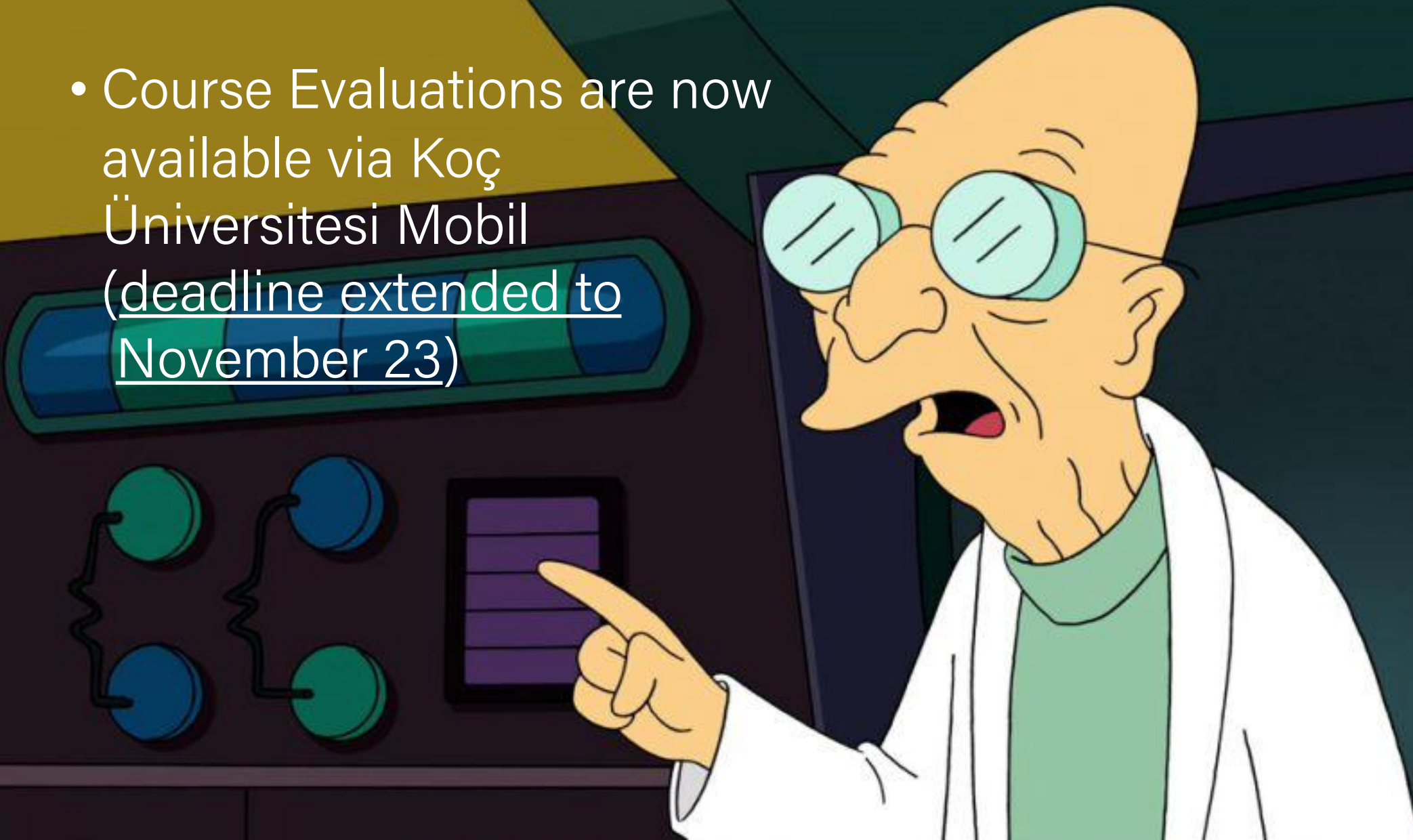


KOÇ
UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

Good news, everyone!

- Course Evaluations are now available via Koç Üniversitesi Mobil ([deadline extended to November 23](#))



Recap

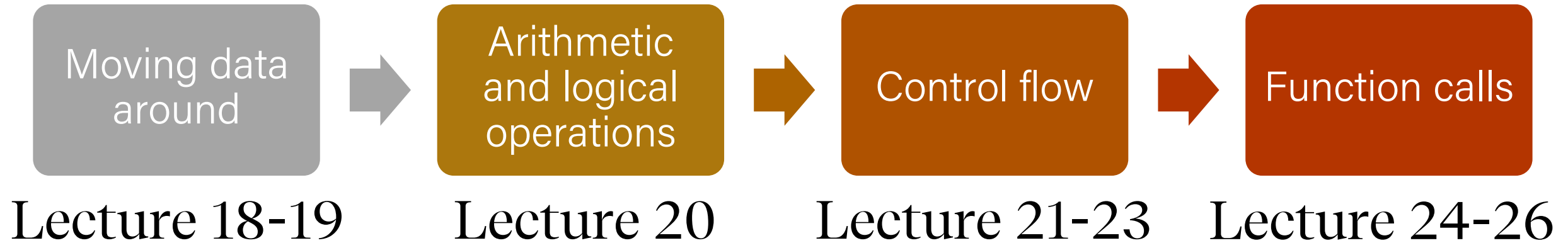
1. **Bits and Bytes** - *How can a computer represent numbers?*
2. **Chars and C-Strings** - *How can a computer represent and manipulate more complex data like text?*
3. **Pointers, Stack and Heap** - *How can we effectively manage all types of memory in our programs?*
4. **Generics** - *How can we use our knowledge of memory and data representation to write code that works with any data type?*

Course Overview

1. **Bits and Bytes** - *How can a computer represent numbers?*
 2. **Chars and C-Strings** - *How can a computer represent and manipulate more complex data like text?*
 3. **Pointers, Stack and Heap** - *How can we effectively manage all types of memory in our programs?*
 4. **Generics** - *How can we use our knowledge of memory and data representation to write code that works with any data type?*
-
5. **Assembly** - *How does a computer interpret and execute C programs?*
 6. **Heap Allocators** - *How do core memory-allocation operations like malloc and free work?*
 7. **The Memory Hierarchy** - *How to improve the performance of application programs by improving their temporal and spatial locality?*
 8. **Code Optimization** - *How write C code so that a compiler can then generate efficient machine code?*
 9. **Linking** - *How static and dynamic linking work?*

COMP201 Topic 6: How does
a computer interpret and
execute C programs?

Learning Assembly



Learning Goals

- Learn what assembly language is and why it is important
- Become familiar with the format of human-readable assembly and x86
- Learn the **mov** instruction and how data moves around at the assembly level

Plan for Today

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The **mov** instruction

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class

Lecture Plan

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The **mov** instruction

Bits all the way down

Data representation so far

- Integer (unsigned `int`, 2's complement signed `int`)
- Floating Points (IEEE single (`float`) and double (`double`) precision)
- `char` (ASCII)
- Address (unsigned long)
- Aggregates (arrays, `structs`)

The code itself is binary too!

- Instructions (machine encoding)

GCC

- **GCC** is the compiler that converts your human-readable code into machine-readable instructions.
- C, and other languages, are high-level abstractions we use to write code efficiently. But computers don't really understand things like data structures, variable types, etc. Compilers are the translator!
- Pure machine code is 1s and 0s – everything is bits, even your programs! But 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're going to go behind the curtain to see what the assembly code for our programs looks like.

Lecture Plan

- Overview: GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The `mov` instruction

Demo: Looking at an Executable (objdump -d)



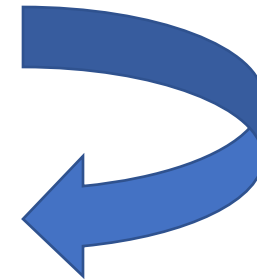
Our First Assembly

```
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;  
}
```



make
objdump -d sum

0000000004005b6 <sum_array>:

4005b6:	ba 00 00 00 00	mov	\$0x0,%edx
4005bb:	b8 00 00 00 00	mov	\$0x0,%eax
4005c0:	eb 09	jmp	4005cb <sum_array+0x15>
4005c2:	48 63 ca	movslq	%edx,%rcx
4005c5:	03 04 8f	add	(%rdi,%rcx,4),%eax
4005c8:	83 c2 01	add	\$0x1,%edx
4005cb:	39 f2	cmp	%esi,%edx
4005cd:	7c f3	j1	4005c2 <sum_array+0xc>
4005cf:	f3 c3	repz	retq

Our First Assembly

00000000004005b6 <sum_array>:

```
4005b6:    ba 00 00 00 00
4005bb:    b8 00 00 00 00
4005c0:    eb 09
4005c2:    48 63 ca
4005c5:    03 04 8f
4005c8:    83 c2 01
4005cb:    39 f2
4005cd:    7c f3
4005cf:    f3 c3
```

```
mov     $0x0,%edx
mov     $0x0,%eax
jmp     4005cb <sum_array+0x15>
movslq %edx,%rcx
add     (%rdi,%rcx,4),%eax
add     $0x1,%edx
cmp     %esi,%edx
jl     4005c2 <sum_array+0xc>
repz   retq
```


Our First Assembly

0000000004005b6 <sum_array>;

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

```
4005b6: 4a 00 00 00 00 00 mov     $0x0,%edx
4005bb: 4a 00 00 00 00 00 mov     $0x0,%eax
4005c0: 4a 00 00 00 00 00 pshlq   $0,%eax
4005c6: 4a 00 00 00 00 00 vsiq    %edx,%rcx
4005cb: 4a 00 00 00 00 00 d        (%rdi,%rcx,4),%eax
4005c8: 83 c2 01          add     $0x1,%edx
4005cb: 39 f2            cmp     %esi,%edx
4005cd: 7c f3            jl     4005c2 <sum_array+0xc>
4005cf: f3 c3            repz   retq
```

Our First Assembly

0000000004005b6 <sum_array>:

4005b6:	ba 00 00 00 00	mov	\$0x0,%edx
4005bb:	b8 00 00 00 00	mov	\$0x0,%eax
4005c0:	eb 09	jmp	4005cb <sum_array+0x15>
4005c2:	44 44 44 44 44	mov	0x44(4),%eax
4005c5:	00 00 00 00 00	mov	\$0,%eax
4005c8:	8b 00 00 00 00	mov	%eax,%eax
4005cb:	39 f2	cmp	%esi,%edx
4005cd:	7c f3	j1	4005c2 <sum_array+0xc>
4005cf:	f3 c3	repz	retq

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

Our First Assembly

00000000004005b6 <sum_array>:

4005b6: ba 00 00 00 00

4005bb: b8 00 00 00 00

4005c0: eb 09

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

4005c5: 59 12

4005cd: 7c f3

4005cf: f3 c3

```
mov    $0x0,%edx
mov    $0x0,%eax
jmp    4005cb <sum_array+0x15>
movslq %edx,%rcx
add    (%rdi,%rcx,4),%eax
add    $0x1,%edx
cmp    %esi,%edx
jl     4005c2 <sum_array+0xc>
repz  retq
```

Our First Assembly

0000000004005b6 <sum_array>:

```
4005b6:  ba 00 00 00 00
4005bb:  b8 00 00 00 00
4005c0:  eb 09
4005c2:  48 63 ca
4005c5:  03 04 8f
4005c8:  83 c2 01
4005cb:  39 f2
4005cd:  7c f3
4005cf:  f3 c3
```

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

Our First Assembly

0000000004005b6 <sum_array>:

```
4005b6:    ba 00 00 00 00
4005bb:    b8 00 00 00 00
4005c0:    eb 09
4005c2:    48 63 ca
4005c5:    03 04 8f
4005c8:    83 c2 01
4005cb:    39 f2
4005cd:    7c f3
4005cf:    f3 c3
```

```
mov     $0x0,%edx
mov     $0x0,%eax
jmp     4005cb <sum_array+0x15>
movslq  %edx,%rcx
add     (%rdi,%rcx,4),%eax
add     $0x1,%edx
cmp     %esi,%edx
jl      4005c2 <sum_array+0xc>
repz   retq
```

Our First Assembly

00000000004005b6 <sum_array>:

4005b6:	ba 00 00 00 00	mov	\$0x0,%edx
4005bb:	b8 00 00 00 00	mov	\$0x0,%eax
4005c0:	eb 09	jmp	4005cb <sum_array+0x15>
4005c2:	48 63 ca	movslq	%edx,%rcx
4005c5:	03 04 8f	add	(%rdi,%rcx,4),%eax
4005c8:	83 c2 01	add	\$0x1,%edx
4005cb:	39 f2	cmp	%esi,%edx
4005cd:	7c f3	jle	4005c2 <sum_array+0xc>
4005cf:	f3 c3	repz	retq

Each instruction has an operation name ("opcode").

Our First Assembly

00000000004005b6 <sum_array>:

```
4005b6:    ba 00 00 00 00
4005bb:    b8 00 00 00 00
4005c0:    eb 09
4005c2:    48 63 ca
4005c5:    03 04 8f
4005c8:    83 c2 01
4005cb:    39 f2
4005cd:    7c f3
4005cf:    f3 c3
```

```
mov     $0x0,%edx
mov     $0x0,%eax
jmp     4005cb <sum_array+0x15>
movslq  %edx,%rcx
add     (%rdi,%rcx,4),%eax
add     $0x1,%edx
cmp     %esi,%edx
jl     4005c2 <sum_array+0xc>
```


Each instruction can also have arguments ("operands").

Our First Assembly

0000000004005b6 <sum_array>:

```
4005b6:  ba 00 00 00 00
4005bb:  b8 00 00 00 00
4005c0:  eb 09
4005c2:  48 63 ca
4005c5:  03 04 8f
4005c8:  83 c2 01
4005cb:  39 f2
4005cd:  7c f3
4005cf:  f3 c3
```

```
mov    $0x0,%edx
mov    $0x0,%eax
jmp    4005cb <sum_array+0x15>
movslq %edx,%rcx
add    (%rdi,%rcx,4),%eax
add    $0x1,%edx
cmp    %eax,%edx
jl    4005c2 <sum_array+0xc>
repz  retq
```



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

Our First Assembly

0000000004005b6 <sum_array>:

```
4005b6:  ba 00 00 00 00
4005bb:  b8 00 00 00 00
4005c0:  eb 09
4005c2:  48 63 ca
4005c5:  03 04 8f
4005c8:  83 c2 01
4005cb:  39 f2
4005cd:  7c f3
4005cf:  f3 c3
```

```
mov    $0x0,%edx
mov    $0x0,%eax
jmp    4005cb <sum_array+0x15>
movslq %edx,%rcx
add    (%rdi,%rcx,4),%eax
add    $0x1,%edx
cmp    %esi,%edx
jl     4005c2 <sum_array+0xc>
repz  retq
```

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

Lecture Plan

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- **Registers and The Assembly Level of Abstraction**
- The **mov** instruction

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`

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

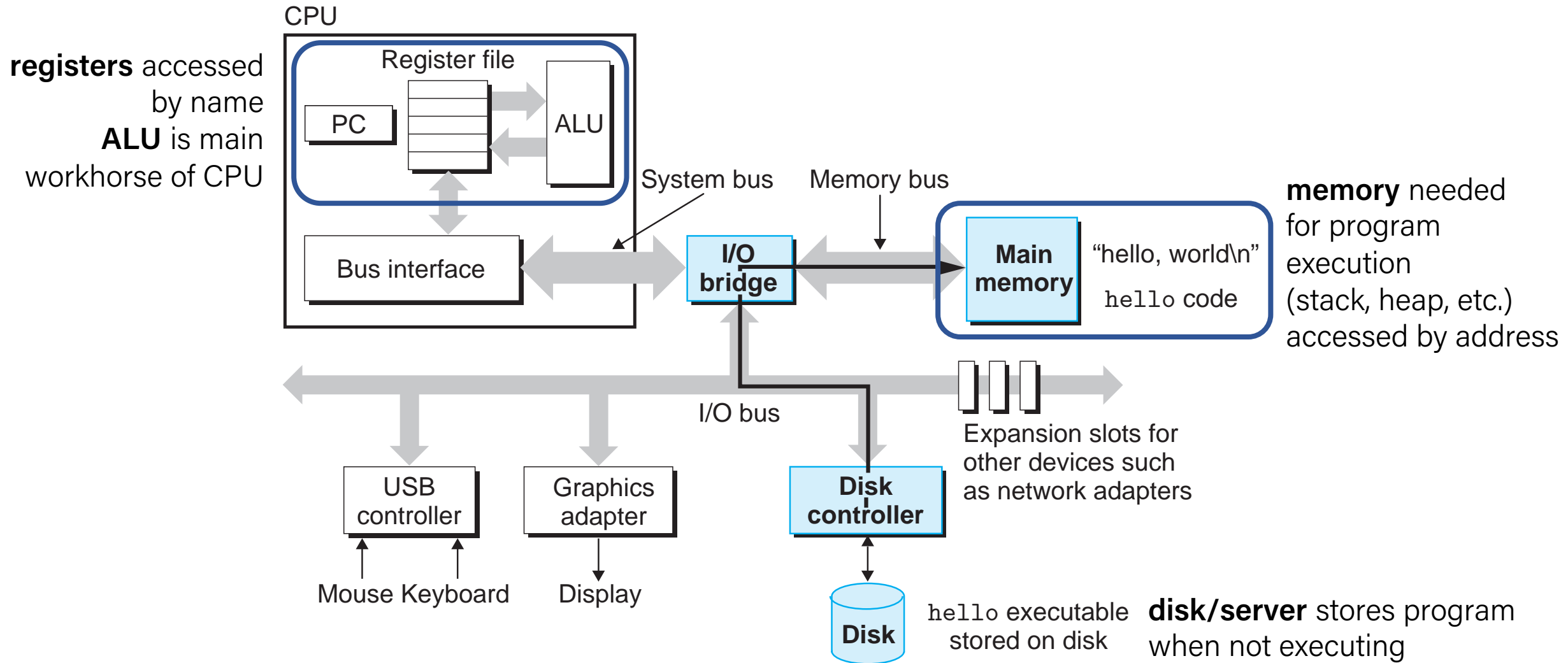
- A **register** is a 64-bit space inside the processor.
- There are 16 registers available, each with a unique name.
- Registers are like “scratch paper” for the processor. Data being calculated or manipulated is moved to registers first. Operations are performed on registers.
- Registers also hold 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. This is the level of logic your program must be in to execute!

Machine-Level Code

Assembly instructions manipulate these registers. For example:

- One instruction adds two numbers in registers
- One instruction transfers data from a register to memory
- One instruction transfers data from memory to a register

Computer architecture



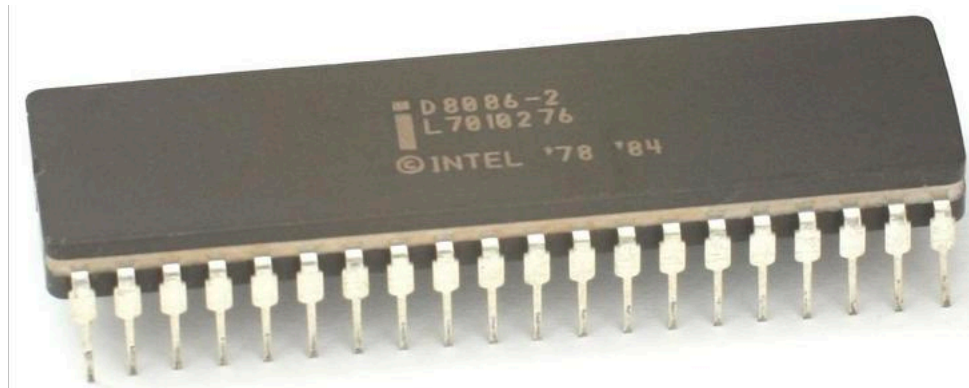
GCC And Assembly

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

C	Assembly Abstraction
int sum = x + y;	<ol style="list-style-type: none">1) <i>Copy x into register 1</i>2) <i>Copy y into register 2</i>3) <i>Add register 2 to register 1</i>4) <i>Write register 1 to memory for sum</i>

Assembly

- We are going to learn the **x86-64** instruction set architecture. This instruction set is used by Intel and AMD processors.
- There are many other instruction sets: ARM, MIPS, etc.



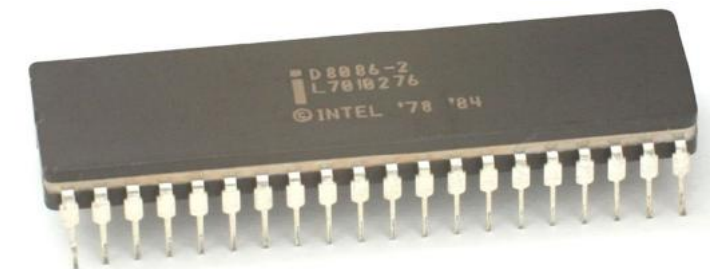
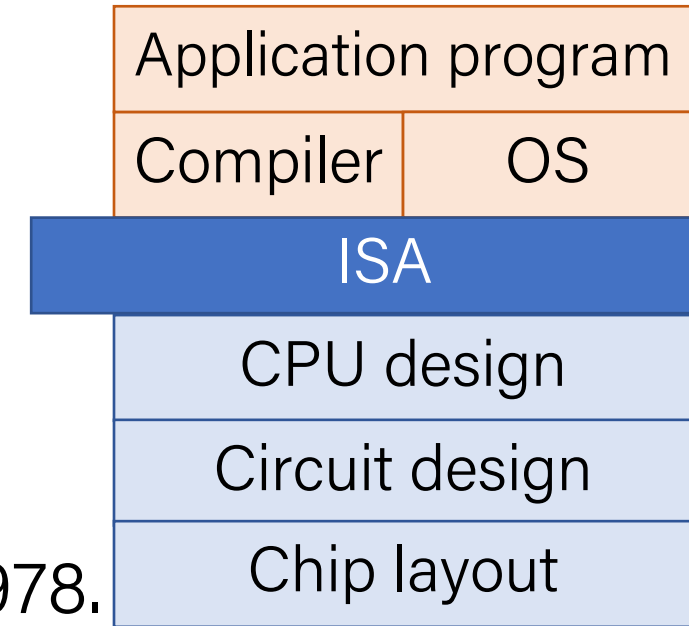
Instruction set architecture (ISA)

A contract between program/compiler and hardware:

- Defines operations that the processor (CPU) can execute
- Data read/write/transfer operations
- Control mechanisms

Intel originally designed their instruction set back in 1978.

- Legacy support is a huge issue for x86-64
- Originally 16-bit processor, then 32 bit, now 64 bit. These design choices dictated the register sizes (and even register/instruction names).



Lecture Plan

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- 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.

Operand Forms: Absolute Addresses

mov **0x104**, _____

Copy the value at address 0x104 into some destination.

mov _____, **0x104**

Copy the value from some source into the memory at address 0x104.

Practice #1: Operand Forms

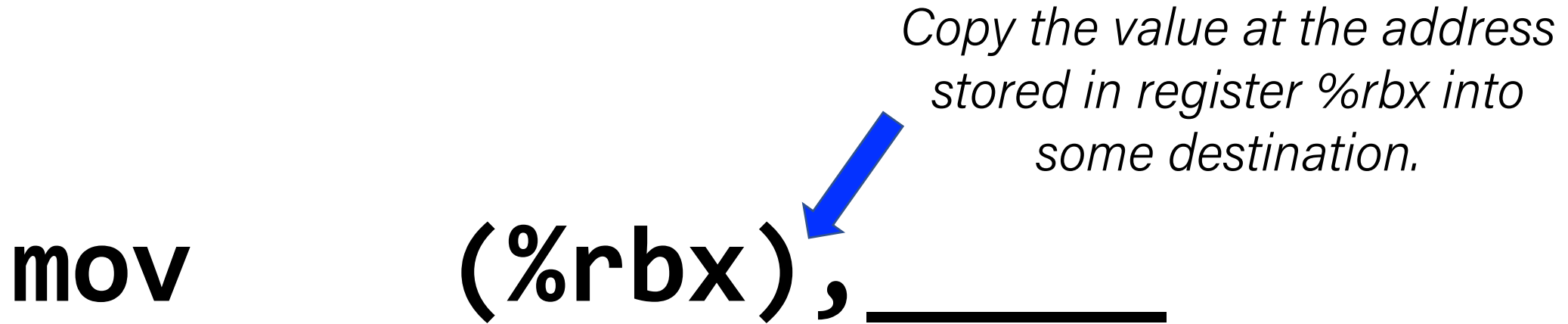
What are the results of the following move instructions (executed separately)? For this problem, assume the value 5 is stored at address 0x42, and the value 8 is stored in %rbx.

- 1. mov \$0x42,%rax** Move 0x42 into %rax
- 2. mov 0x42,%rax** Move 5 into %rax
- 3. mov %rbx,0x55** Move 8 to the address %rax

Operand Forms: Indirect

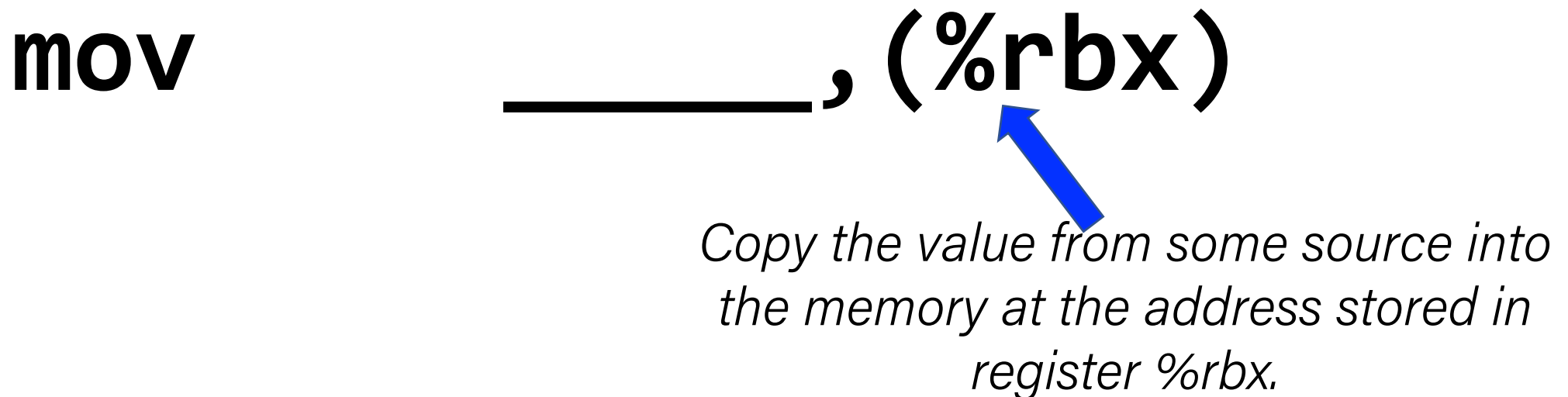
mov **(%rbx), _____**

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

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

mov

(%rax, %rdx), _____

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).*

Practice #2: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume

the value $0x11$ is stored at address $0x10C$,
the value $0xAB$ is stored at address $0x104$,
 $0x100$ is stored in register `%rax` and $0x3$ is stored in `%rdx`.

1. `mov $0x42, (%rax)` Move $0x42$ to memory address $0x100$
2. `mov 4(%rax), %rcx` Move $0xAB$ into `%rcx`
3. `mov 9(%rax,%rdx), %rcx` Move $0x11$ into `%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)

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;  
}
```

We're 1/4th of the way to understanding assembly!
What looks understandable right now?

Some notes:

- Registers store addresses and values
- `mov src, dst` **copies** value into `dst`
- `sizeof(int)` is 4
- Instructions executed sequentially

00000000004005b6 <sum_array>:

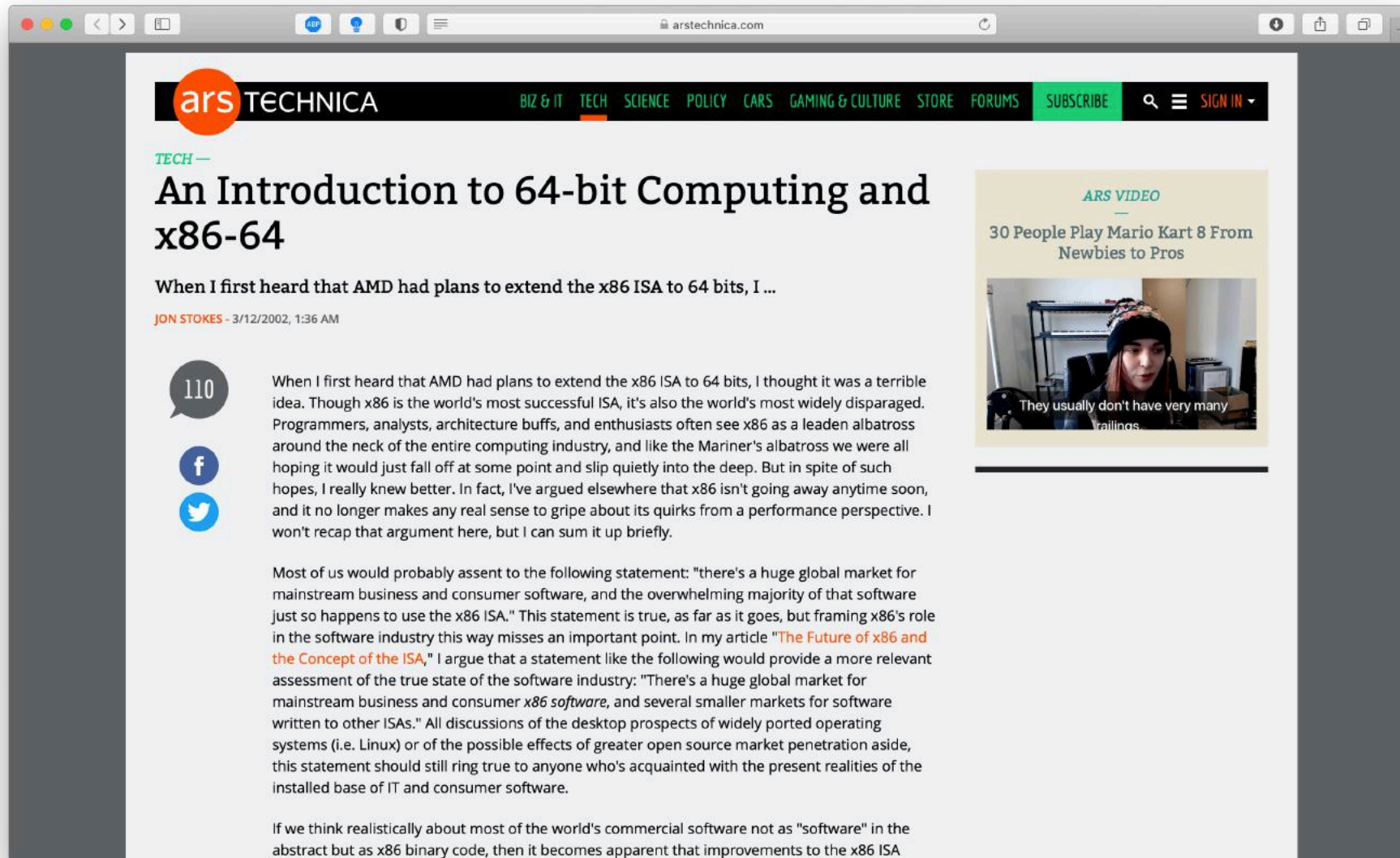
```
4005b6:  ba 00 00 00 00  
4005bb:  b8 00 00 00 00  
4005c0:  eb 09  
4005c2:  48 63 ca  
4005c5:  03 04 8f  
4005c8:  82 c2 01
```

```
mov    $0x0,%edx  
mov    $0x0,%eax  
jmp    4005cb <sum_array+0x15>  
movslq %edx,%rcx  
add    (%rdi,%rcx,4),%eax  
add    $0x1,%edx  
cmp    %esi,%edx  
jl     4005c2 <sum_array+0xc>  
repz  retq
```

We'll come back to this example in future lectures!



Additional Reading



The screenshot shows a web browser window displaying an article on the Ars Technica website. The browser's address bar shows the URL arstechnica.com. The website's navigation bar includes links for 'BIZ & IT', 'TECH', 'SCIENCE', 'POLICY', 'CARS', 'GAMING & CULTURE', 'STORE', 'FORUMS', 'SUBSCRIBE', and 'SIGN IN'. The article is categorized under 'TECH' and has a main title 'An Introduction to 64-bit Computing and x86-64'. The author is identified as 'JON STOKES' with a date of '3/12/2002, 1:36 AM'. The article text discusses the author's initial skepticism about AMD's plans to extend the x86 ISA to 64 bits, arguing that while x86 is widely disparaged, it remains a dominant ISA. A video player on the right side of the page shows a person playing Mario Kart 8, with a caption that reads 'They usually don't have very many railings'.

ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE STORE FORUMS SUBSCRIBE SIGN IN

TECH —

An Introduction to 64-bit Computing and x86-64

When I first heard that AMD had plans to extend the x86 ISA to 64 bits, I ...

JON STOKES - 3/12/2002, 1:36 AM

110

f

Twitter

When I first heard that AMD had plans to extend the x86 ISA to 64 bits, I thought it was a terrible idea. Though x86 is the world's most successful ISA, it's also the world's most widely disparaged. Programmers, analysts, architecture buffs, and enthusiasts often see x86 as a leaden albatross around the neck of the entire computing industry, and like the Mariner's albatross we were all hoping it would just fall off at some point and slip quietly into the deep. But in spite of such hopes, I really knew better. In fact, I've argued elsewhere that x86 isn't going away anytime soon, and it no longer makes any real sense to gripe about its quirks from a performance perspective. I won't recap that argument here, but I can sum it up briefly.

Most of us would probably assent to the following statement: "there's a huge global market for mainstream business and consumer software, and the overwhelming majority of that software just so happens to use the x86 ISA." This statement is true, as far as it goes, but framing x86's role in the software industry this way misses an important point. In my article "[The Future of x86 and the Concept of the ISA](#)," I argue that a statement like the following would provide a more relevant assessment of the true state of the software industry: "There's a huge global market for mainstream business and consumer *x86 software*, and several smaller markets for software written to other ISAs." All discussions of the desktop prospects of widely ported operating systems (i.e. Linux) or of the possible effects of greater open source market penetration aside, this statement should still ring true to anyone who's acquainted with the present realities of the installed base of IT and consumer software.

If we think realistically about most of the world's commercial software not as "software" in the abstract but as x86 binary code, then it becomes apparent that improvements to the x86 ISA

ARS VIDEO

30 People Play Mario Kart 8 From Newbies to Pros

They usually don't have very many railings

<https://arstechnica.com/gadgets/2002/03/an-introduction-to-64-bit-computing-and-x86-64/>

Recap

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The **mov** instruction

Next time: diving deeper into assembly