photo: Intel CPU Wafers

COMP201

Computer Systems & Programming

Lecture #13 – Introduction to x86-64 Assembly

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Spring 2021

Good news, everyone!

Image: Professor Farnsworth (Futurama)

- Assignment 3 is out! (due April 14, 23:59)
  - Farzin will announce an extra office hour to answer your questions
- No labs this week!

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?*
5. **Working with Multiple Files** – *What really happens in GCC? How to write your own Makefiles?*

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. **Working with Multiple Files** – *What really happens in GCC? How to write your own Makefiles?*

---

6. **Assembly** - *How does a computer interpret and execute C programs?*
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?*
10. **Heap Allocators** - *How do core memory-allocation operations like malloc and free work?*

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

Learning Assembly

Moving data around	→	Arithmetic and logical operations	→	Control flow	→	Function calls
Lecture 13		Lecture 14		Lecture 15		Lecture 16

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

```c
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

```c
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
```

```
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                 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:    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                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 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
                                         p    4005cb <sum_array+0x15>
                                      vslq %edx,%rcx
                                         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
```

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

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:   4
  4005c5:   0                                 4),%eax
  4005c8:   8
  4005cb:   39 f2                 cmp    %esi,%edx
  4005cd:   7c f3                 jl     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    mov    $0x0,%edx
  4005bb:    b8 00 00 00 00    mov    $0x0,%eax
  4005c0:    eb 09             jmp    4005cb <sum_array+0x15>
                               movslq %edx,%rcx
                               add    (%rdi,%rcx,4),%eax
                               add    $0x1,%edx
  4005cb:    39 f2             cmp    %esi,%edx
  4005cd:    7c f3             jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3             repz retq
```

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

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00        mov    $0x0,%edx
  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                 repz retq
```

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

```
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                   jl     4005c2 <sum_array+0xc>
  4005cf:      f3 c3                   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                 jl     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          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                   jl     4005c2 <sum_array+0xc>
  4005cf:     f3 c3
```

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

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                jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                repz retq
```

$[number] means a constant value, or "immediate" (e.g. 1 here).

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                   jl     4005c2 <sum_array+0xc>
  4005cf:     f3 c3                   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

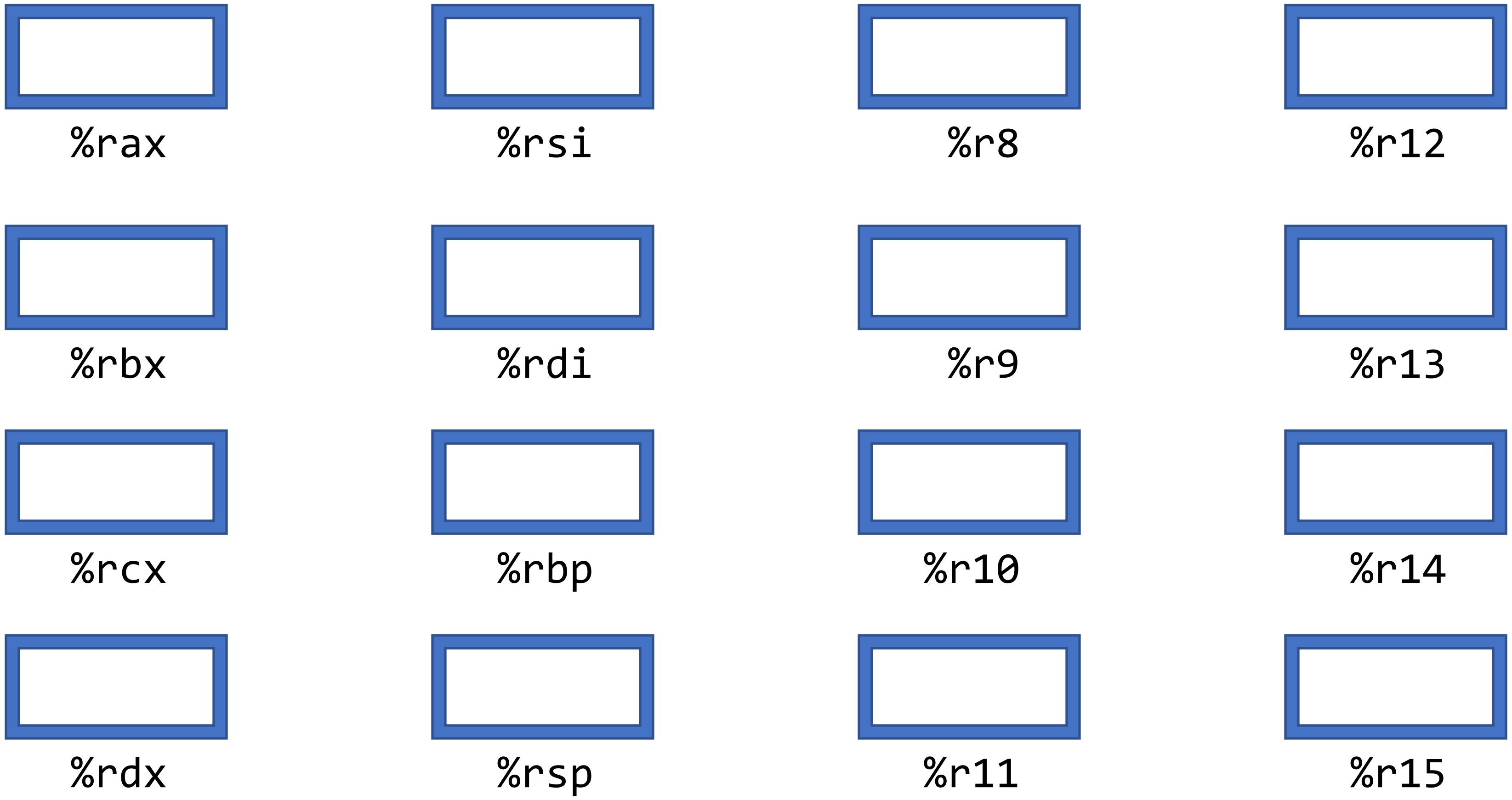
Assembly Abstraction

- 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



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!

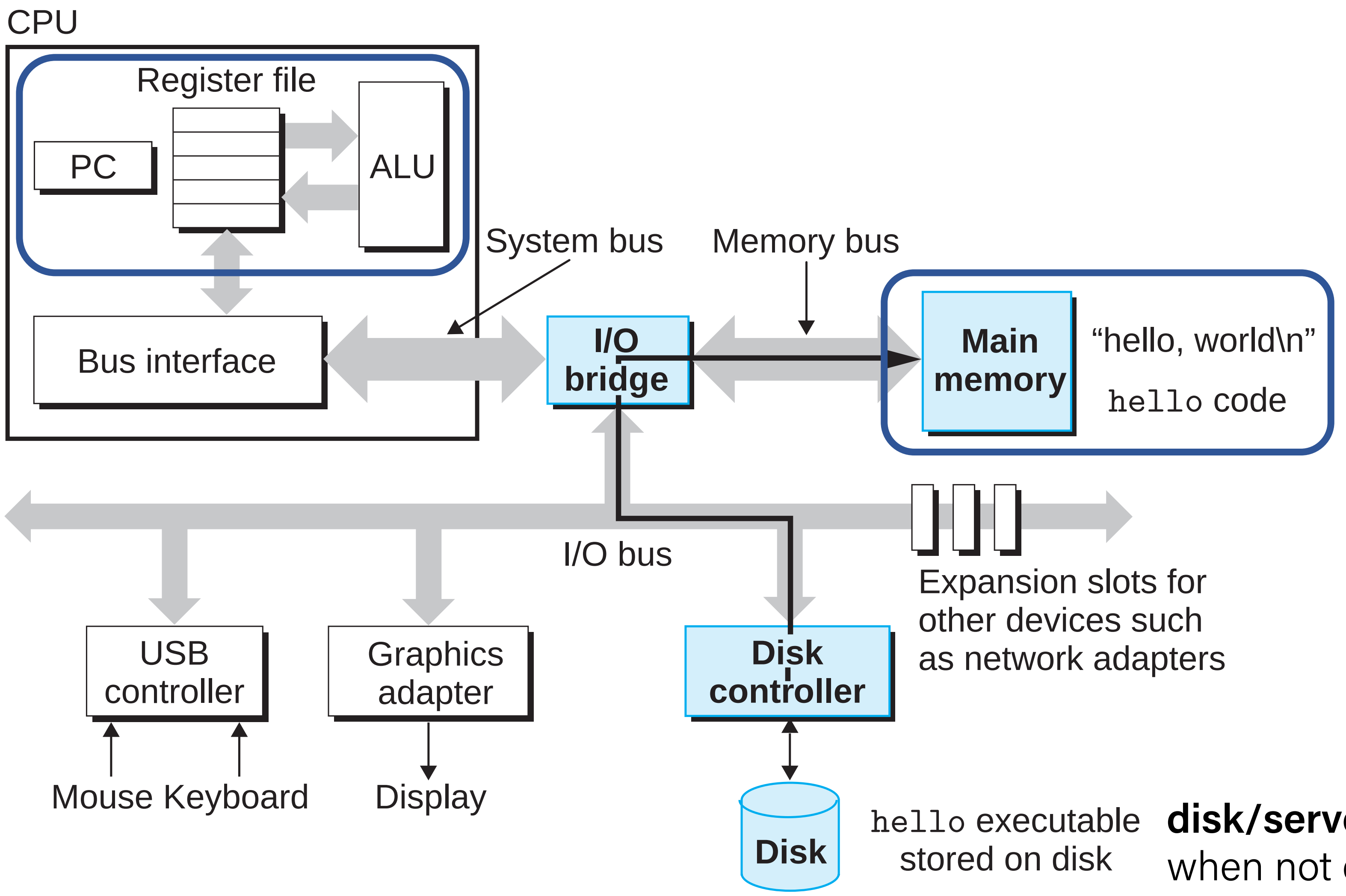
Machine-Level Code

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

registers accessed by name
ALU is main workhorse of CPU



memory needed for program execution (stack, heap, etc.) accessed by address

disk/server stores program when not executing

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;`	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*

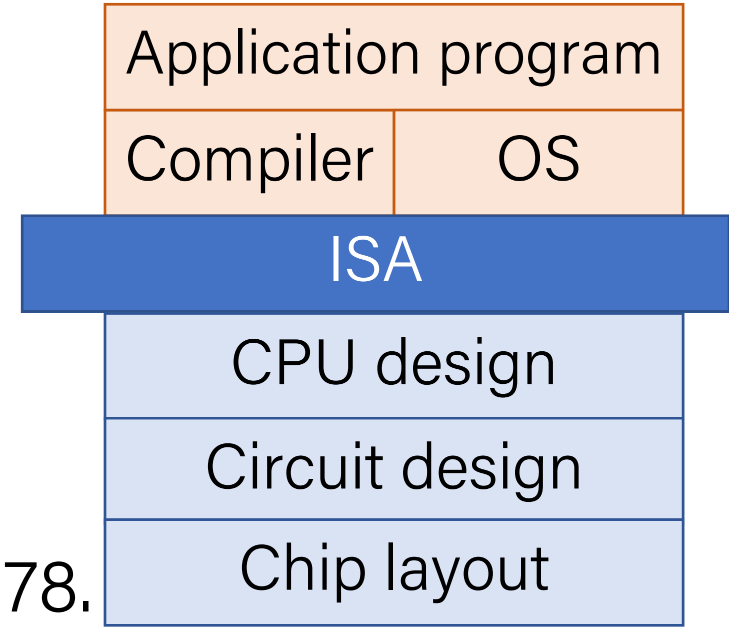
Assembly

- 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)

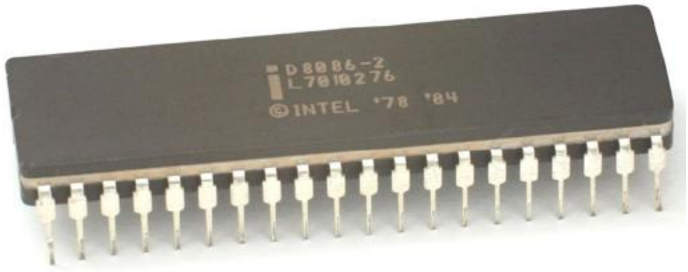
A contract between program/compiler and hardware:

- 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 <u>copies</u> 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

Copy the value in register %rbx into some destination.

```
mov     %rbx,____
```

```
mov     ____,%rbx
```

Copy the value from some source into register %rbx.

Operand Forms: Absolute Addresses

Copy the value at address 0x104 into some destination.

```
mov     0x104,_____

mov     _____,0x104
```

Copy the value from some source into the 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 address 0x55

Operand Forms: Indirect

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

```
mov        (%rbx),______
```

```
mov        ______,(%rbx)
```

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

Operand Forms: Base + Displacement

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

```
mov     0x10(%rax),__________
```

```
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 ***<u>0x10 plus</u>*** *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 ***<u>0x10 plus</u>*** *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

Imm(r_b, r_i) is equivalent to address **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),________
```

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

```
mov    ________,(,%rdx,4)
```

*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 (**<u>the value in register %rax</u>** 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 (**<u>the value in register %rax</u>** plus 2 times the value in register %rdx).*

Operand Forms: Scaled Indexed

*Copy the value at the address which is (**<u>0x4 plus</u>** 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 (**<u>0x4 plus</u>** the value in register %rax plus 2 times the value in register %rdx).*

Most General Operand Form

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

is equivalent to...

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

Most General Operand Form

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

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)

Memory Location Syntax

Syntax	Meaning
`0x104`	Address `0x104` (no `$`)
`(%rax)`	What's in `%rax`
`4(%rax)`	What's in `%rax`, plus `4`
`(%rax, %rdx)`	Sum of what's in `%rax` and `%rdx`
`4(%rax, %rdx)`	Sum of values in `%rax` and `%rdx`, plus 4
`(, %rcx, 4)`	What's in `%rcx`, times 4 (multiplier can be 1, 2, 4, 8)
`(%rax, %rcx, 2)`	What's in `%rax`, plus 2 times what's in `%rcx`
`8(%rax, %rcx, 2)`	What's in `%rax`, plus 2 times what's in `%rcx`, plus 8

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

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

Practice #3: Operand Forms

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
the value *0x11* is stored at address *0x10C*.

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

Move 0x42 to memory address 0x100

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

Move 0x11 into %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)

Goals of indirect addressing: C

Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!

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;
}
```

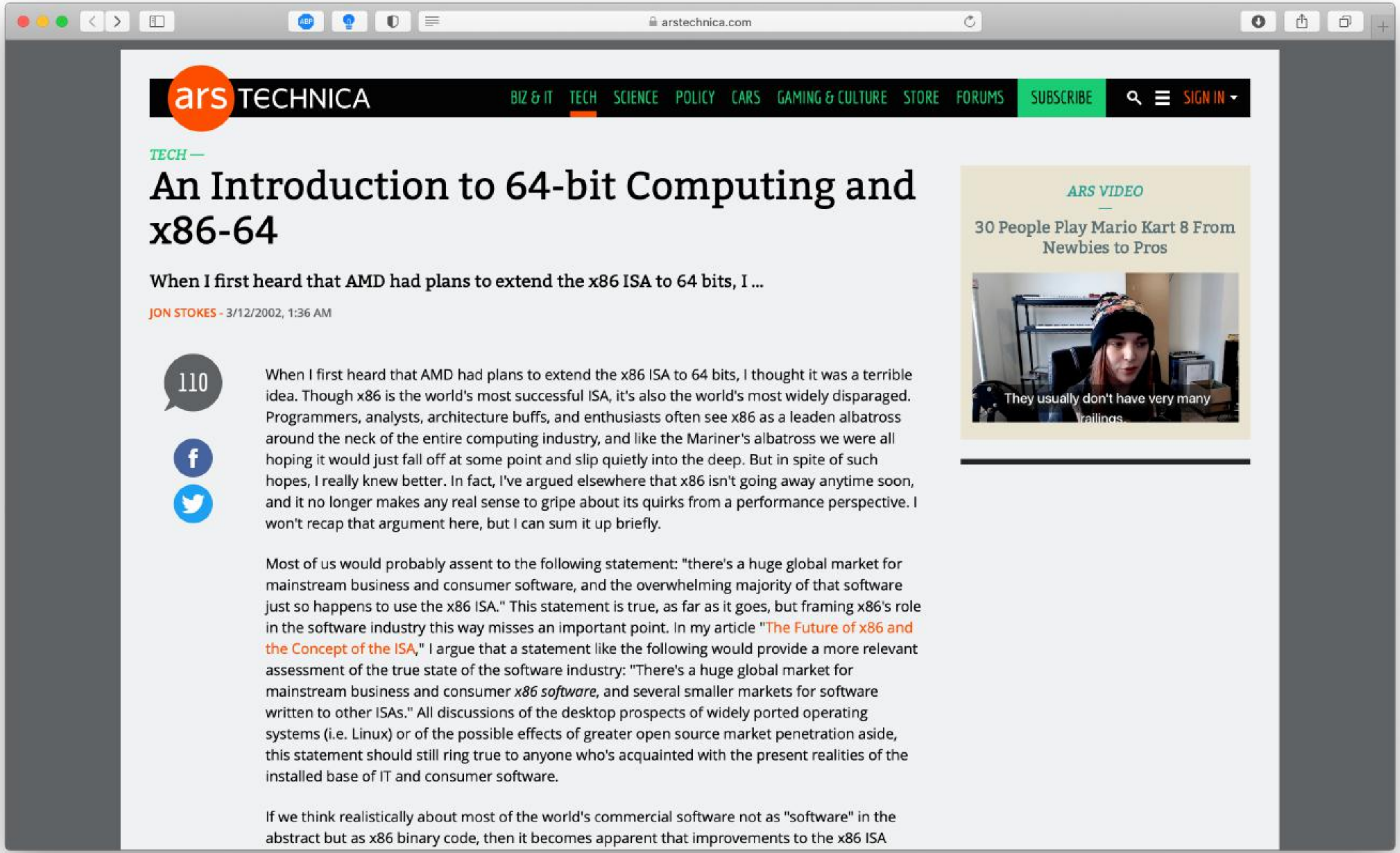
We're 1/4th of the way to understanding assembly!
What looks understandable right now?

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          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
                                     cmp    %esi,%edx
                                     jl     4005c2 <sum_array+0xc>
                                     repz retq
```

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

Additional Reading



arstechnica.com

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

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

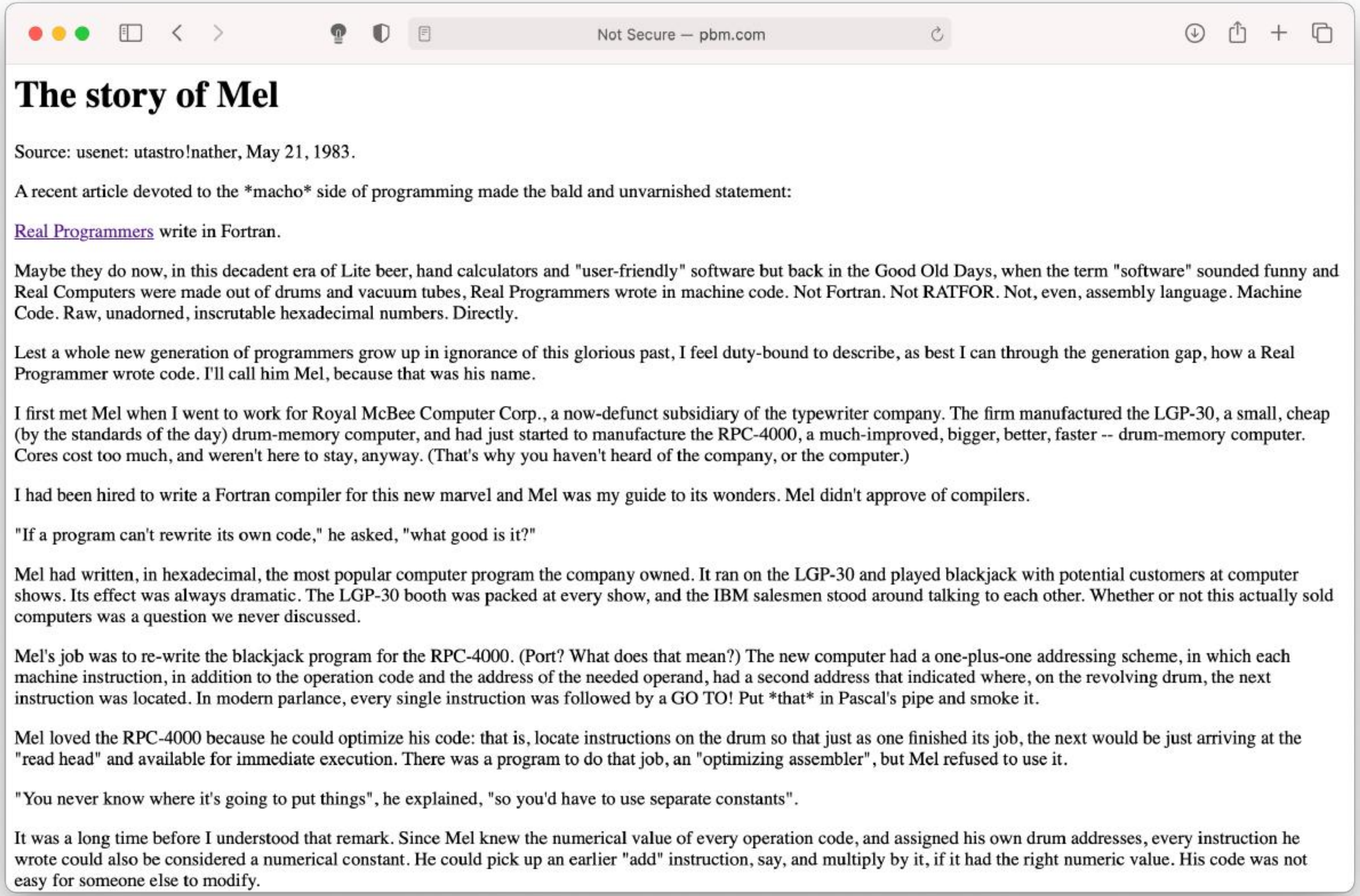
ARS VIDEO

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/

Additional Reading



The story of Mel

Source: usenet: utastro!nather, May 21, 1983.

A recent article devoted to the *macho* side of programming made the bald and unvarnished statement:

Real Programmers write in Fortran.

Maybe they do now, in this decadent era of Lite beer, hand calculators and "user-friendly" software but back in the Good Old Days, when the term "software" sounded funny and Real Computers were made out of drums and vacuum tubes, Real Programmers wrote in machine code. Not Fortran. Not RATFOR. Not, even, assembly language. Machine Code. Raw, unadorned, inscrutable hexadecimal numbers. Directly.

Lest a whole new generation of programmers grow up in ignorance of this glorious past, I feel duty-bound to describe, as best I can through the generation gap, how a Real Programmer wrote code. I'll call him Mel, because that was his name.

I first met Mel when I went to work for Royal McBee Computer Corp., a now-defunct subsidiary of the typewriter company. The firm manufactured the LGP-30, a small, cheap (by the standards of the day) drum-memory computer, and had just started to manufacture the RPC-4000, a much-improved, bigger, better, faster -- drum-memory computer. Cores cost too much, and weren't here to stay, anyway. (That's why you haven't heard of the company, or the computer.)

I had been hired to write a Fortran compiler for this new marvel and Mel was my guide to its wonders. Mel didn't approve of compilers.

"If a program can't rewrite its own code," he asked, "what good is it?"

Mel had written, in hexadecimal, the most popular computer program the company owned. It ran on the LGP-30 and played blackjack with potential customers at computer shows. Its effect was always dramatic. The LGP-30 booth was packed at every show, and the IBM salesmen stood around talking to each other. Whether or not this actually sold computers was a question we never discussed.

Mel's job was to re-write the blackjack program for the RPC-4000. (Port? What does that mean?) The new computer had a one-plus-one addressing scheme, in which each machine instruction, in addition to the operation code and the address of the needed operand, had a second address that indicated where, on the revolving drum, the next instruction was located. In modern parlance, every single instruction was followed by a GO TO! Put *that* in Pascal's pipe and smoke it.

Mel loved the RPC-4000 because he could optimize his code: that is, locate instructions on the drum so that just as one finished its job, the next would be just arriving at the "read head" and available for immediate execution. There was a program to do that job, an "optimizing assembler", but Mel refused to use it.

"You never know where it's going to put things", he explained, "so you'd have to use separate constants".

It was a long time before I understood that remark. Since Mel knew the numerical value of every operation code, and assigned his own drum addresses, every instruction he wrote could also be considered a numerical constant. He could pick up an earlier "add" instruction, say, and multiply by it, if it had the right numeric value. His code was not easy for someone else to modify.

http://www.pbm.com/~lindahl/mel.html

Extra Practice

Extra Practice

slido

Fill in the blank to complete the code that generated the assembly below.

```
long arr[5];
...
long num = ____???___;
```

```
// %rdi stores arr, %rcx stores 3, and %rax stores num
mov (%rdi, %rcx, 8),%rax
```

Extra Practice

Fill in the blank to complete the code that generated the assembly below.

```
long arr[5];
...
long num = arr[3];
```

```
// %rdi stores arr, %rcx stores 3, and %rax stores num
mov (%rdi, %rcx, 8),%rax
```

Extra Practice

Fill in the blank to complete the code that generated the assembly below.

```
int x = ...
int *ptr = malloc(…);
___???___ = x;
```

```
// %ecx stores x, %rax stores ptr
mov %ecx,(%rax)
```

Extra Practice

Fill in the blank to complete the code that generated the assembly below.

```
int x = ...
int *ptr = malloc(…);
*ptr = x;
```

```
// %ecx stores x, %rax stores ptr
mov %ecx,(%rax)
```

Extra Practice

slido

Fill in the blank to complete the code that generated the assembly below.

```
char str[5];
...
___???___ = 'c';
```

```
// %rcx stores str, %rdx stores 2
mov $0x63,(%rcx,%rdx,1)
```

Extra Practice

Fill in the blank to complete the code that generated the assembly below.

```
char str[5];
...
str[2] = 'c';
```

```
// %rcx stores str, %rdx stores 2
mov $0x63,(%rcx,%rdx,1)
```

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