Road map

- **Midterm a'comin**
  - Friday in class
  - Exams page on web site has info + practice problems
- **Today's lecture**
  - A first look at assembly
  - Where is our data stored?
  - The mov instruction and addressing modes

It’s bits all the way down…

- **Data representation so far**
  - Integer (unsigned, 2’s complement signed)
  - Char (ASCII)
  - Address (unsigned long)
  - Float/double (IEEE floating point)
  - Aggregates (arrays, structs)

- **The code itself is binary too!**
  - Instructions (machine encoding)

Compiling code, what happens?

simple.c

```
int find_max(int arr[], size_t n)
{
    int max = arr[0];
    for (size_t i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}
...
```

```
myth> make
gcc simple.c -o simple
```

```
^ELF^B^A^A^@^@^@^@^@^@^@^@^@^B^@
>^@^A^@^@^@\300^D@^@^@^@^@^@^@^@
^@^@^@^@^@^@\370\225^@^@^@^@^@^@
^@^@^@^@^@^@8^@^@^@^@&^@#^@^F^@^
@^@^E^@^@^@^@^@^@^@^@^@^@^@^@^@^
...
```

simple

Source file (in text form)
Compiler parses input
validates language rules,
generates assembly instructions
writes object file (in binary form)

What’s in an object file?

```
objdump -d simple
```

```
0000000004005b6 <find_max>:
   4005b6:   8b 07
   4005b8:   ba 01 00 00 00        mov    $0x1,%edx
   4005bd:   eb 0d                 jmp    4005cc <find_max+0x16>
   4005bf:                         mov    (%rdi,%rdx,4),%ecx
   4005c2:                         cmp    %ecx,%eax
   4005c4:                         jge    4005c8 <find_max+0x12>
   4005c6:   89 c8                 mov    %ecx,%eax
   4005c8:   48 83 c2 01           add    $0x1,%rdx
   4005cc:   48 39 f2              cmp    %rsi,%rdx
   4005cf:   72 ee                 jb     4005bf <find_max+0x9>
   4005d1:   f3 c3                 repz retq
```

Name of function, memory address of code (function pointer)

Sequential instructions are at sequential addresses

machine code
each instruction encoded in binary

each machine instruction decoded into human-readable **assembly**

What is an assembly instruction?

```
4005c6:  89 c8          mov  %ecx,%eax
4005c8:  48 83 c2 01    add  $0x1,%rdx
4005cc:  48 39 f2       cmp  %rsi,%rdx
4005cf:  72 ee          jb   4005bf <find_max+0x9>
```

%eax
is register name,
(storage location on CPU)

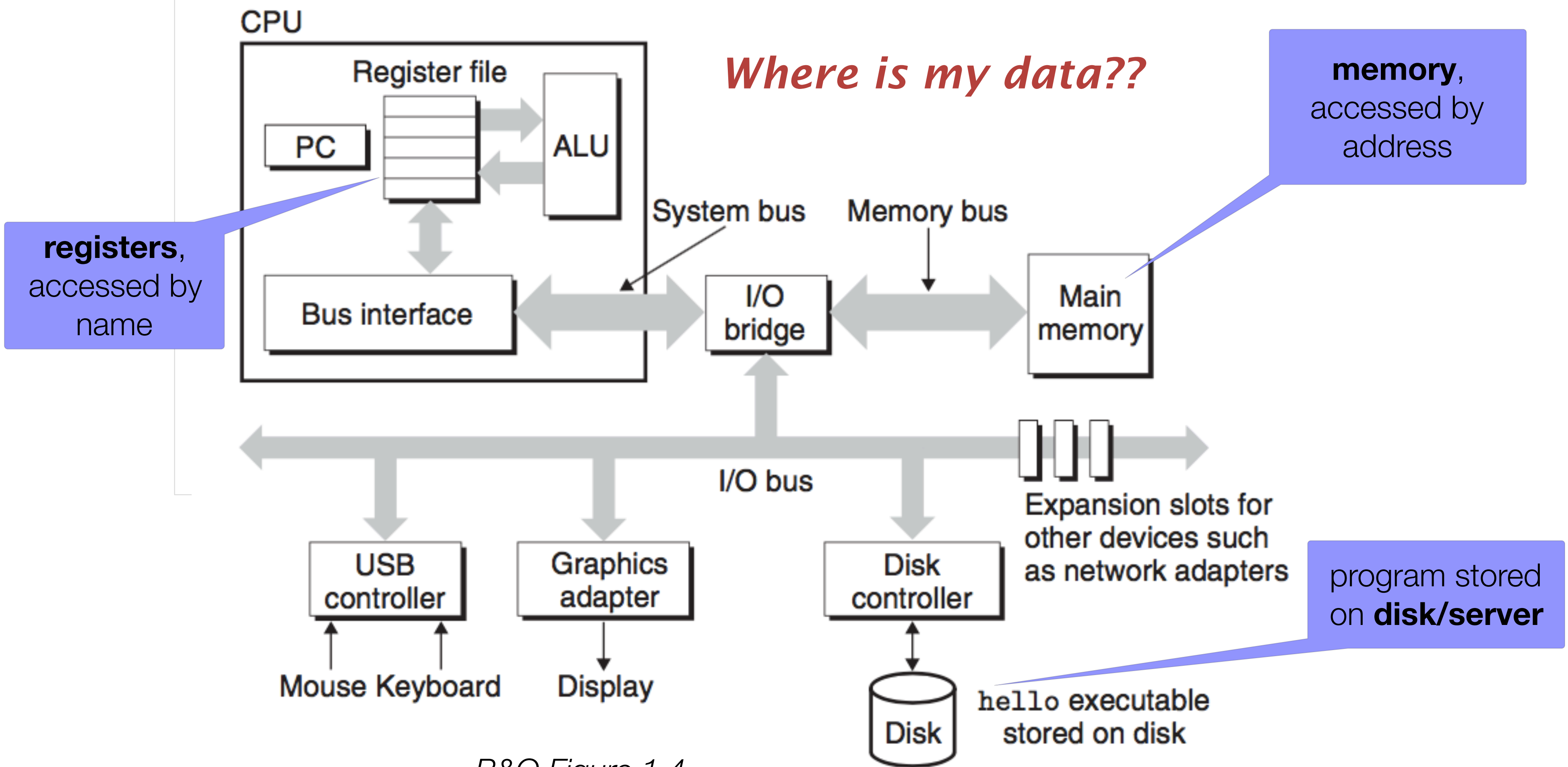
$0x1
is constant value
("immediate")

opcode
(instruction name/type)

operands
(arguments to instruction)

4005bf
is direct address

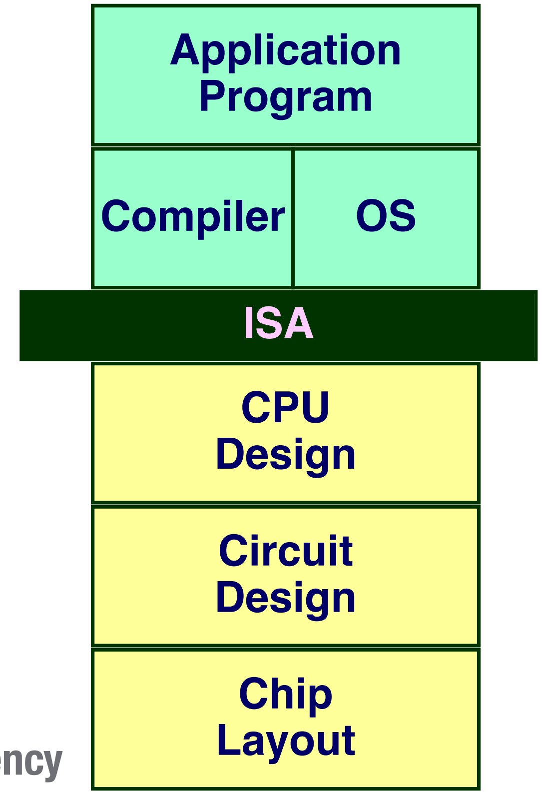
Computer anatomy



B&O Figure 1.4

Instruction set architecture

- **The ISA defines**
  - Operations that the processor can execute
  - Data transfer operations, how to access data
  - Control mechanisms like branch, jump (think loops and if-else)
  - Contract between programmer/compiler and hardware
- **Layer of abstraction**
  - Above: programmer/compiler emits instructions as allowed in ISA
  - Below: hardware implements what is described in ISA
- **ISAs have incredible inertia!**
  - Legacy support is a huge issue for x86-64
- **CISC vs RISC**
  - (CISC, x86) Large set of specialized/expressive instructions, slower frequency
  - (RISC, ARM) Small set of simple instructions, higher frequency
    - Pres. Hennessy Turing Award!



Assembly characteristics

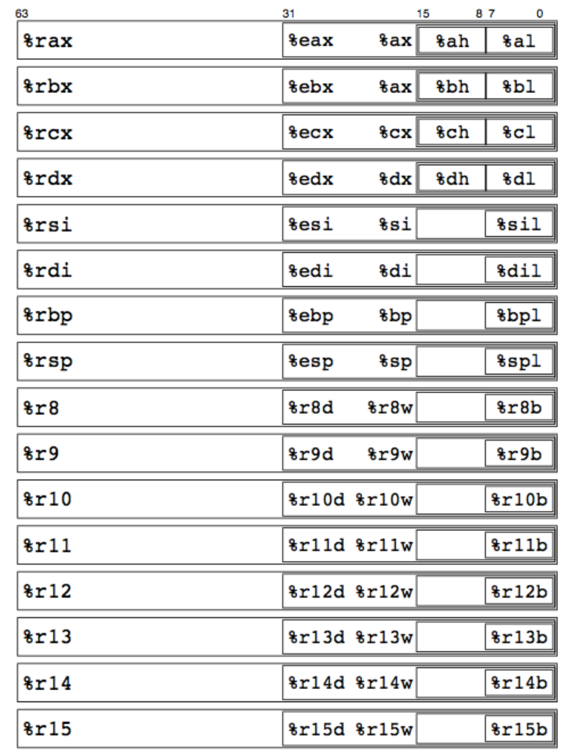
- **Data**
  - **"integer" data, 1/2/4/8 bytes**
    - Char, int, long, pointer, signed/unsigned
  - **Floating point data, 4/8/(10) bytes**
    - Special-purpose registers and instructions
  - **No aggregates**
    - Arrays and structs are just contiguously located bytes in memory
  - **No names, no types**
    - Refer to data by where stored (register/memory), size in bytes
- **Operations**
  - **Perform arithmetic/logical ops on register or memory data**
  - **Transfer data between memory and register**
    - Load/store
  - **Control flow**
    - Unconditional jump to/from other functions
    - Conditional branch

The almighty mov instruction

- **Programs manipulate <u>data</u>**
  - Where is that data stored? registers, memory (also: disk, server, network, …)
- **`mov` instruction is the assembly equivalent of assignment**
  - Most common instruction of all
- **Key insight: no access to variables by name/type**
  - High-level language had descriptive names, type information
  - Assembly accesses variable by identifying where it is stored (register/memory)
- **General form: `movx src, dst`**
  - Copy bytes from one place to another
    - Source can be memory, registers, constants
    - Destination can be memory, registers

Mov operands: imm/reg

Op	Src	Dst	Comments
`movl`	`$0,`	`%eax`	src is immediate
`movb`	`$0x41,`	`%al`	Virtual sub-register
`mov`	`%rax,`	`%rdx`	Register to register



movx suffix is how many bytes to move

- `b` for byte (1), `w` for word (2), `l` for long (4), `q` for quad (8)
  - (suffixes show legacy…)
- Elided if can be inferred from operands

Mov operands: direct/indirect

Op	Src	Dst	Comments
`movl`	`$0,`	`0x605428`	Store, **direct** address (Note no prefix on address literal)
`movl`	`$0,`	`(%rsp)`	Store, **indirect** address (address in register, dereference)
`movl`	`0x605428,`	`%edx`	Load
`movl`	`(%rsp),`	`%edx`	Load

Load = read from memory location

Store = write to memory location

No mem-to-mem transfer

Either src or dst is memory, not both

Direct: Data at fixed location

Indirect: Register holds pointer

Addressing modes

Op	Src	Dst	Comments
`movl`	`$0,`	`0x605428`	**Direct** address
`movl`	`$0,`	`(%rsp)`	**Indirect** address
`movl`	`$0,`	`20(%rsp)`	Indirect with **displacement**

Displacement
is any constant
(negative or positive)

Base

Target address = base + displacement

Addressing modes

Op	Src	Dst	Comments
`movl`	`$0,`	`0x605428`	**Direct** address
`movl`	`$0,`	`(%rsp)`	**Indirect** address
`movl`	`$0,`	`20(%rsp)`	Indirect with **displacement**
`movl`	`$0,`	`20(%rsp, %rax, 4)`	Indirect with **scaled-index**

Displacement
is any constant
(negative or positive)
If missing, =0

Base
register
if missing, = 0

Index
register

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

Target address =
base + displacement + index*scale

Load effective address

- `lea` = "load effective address"
  - Basically a `mov` without the dereference
  - Used for address calculation, e.g. &arr[x]
  - Also arithmetic expressions of form x + ky (faster than sequenced mul/add)
    - where k = 1, 2, 4, 8

- Examples

```
leal (%rax, %rsi, 4), %rax
```

  - Computes base + scaled-index, e.g address of array elem

```
leal 7(%rdx, %rdx, 4), %rdx
```

  - Computes x = 5x + 7 (assuming x stored in %rdx)