Assembly Crash Course

Registers

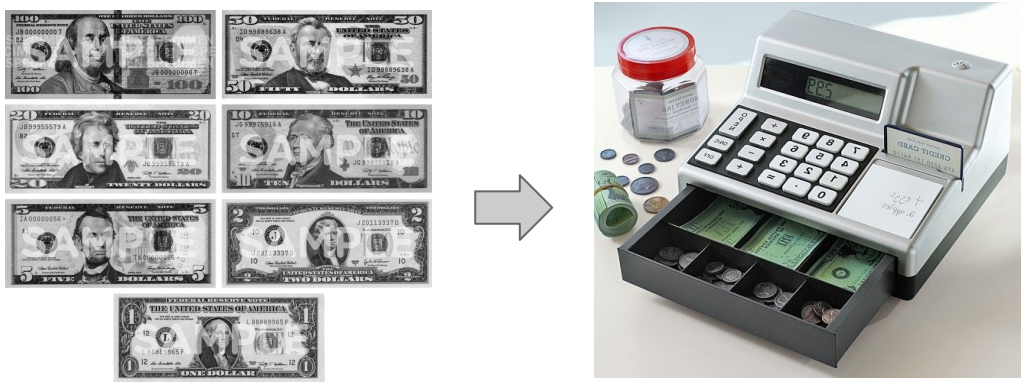
Yan Shoshitaishvili
Arizona State University

The Need for "Registers"

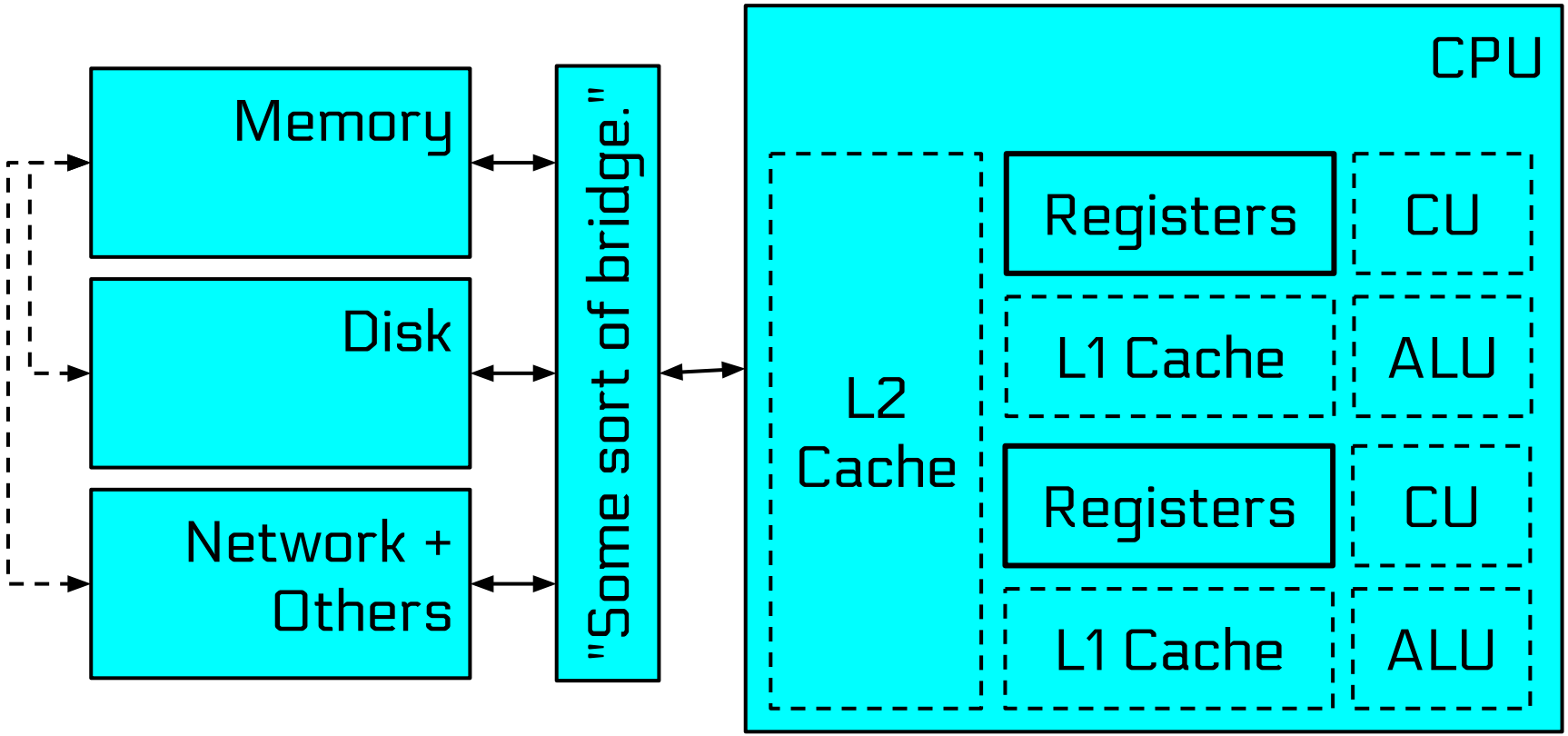
CPUs need to be *fast*.

To be fast, CPUs need rapid access to data they're working on.

This is done via the *Register File*.



Reminder: Computer Architecture



Registers

Registers are very fast, temporary stores for data.

You get several "general purpose" registers:

- 8085: a, c, d, b, e, h, l
- 8086: ax, cx, dx, bx, **sp**, **bp**, si, di
- x86: eax, ecx, edx, ebx, **esp**, **ebp**, esi, edi
- amd64: rax, rcx, rdx, rbx, **rsp**, **rbp**, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15
- arm: r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, **r13**, **r14**

The address of the next instruction is in a register:

eip (x86), rip (amd64), r15 (arm)

Various extensions add other registers (x87, MMX, SSE, etc).

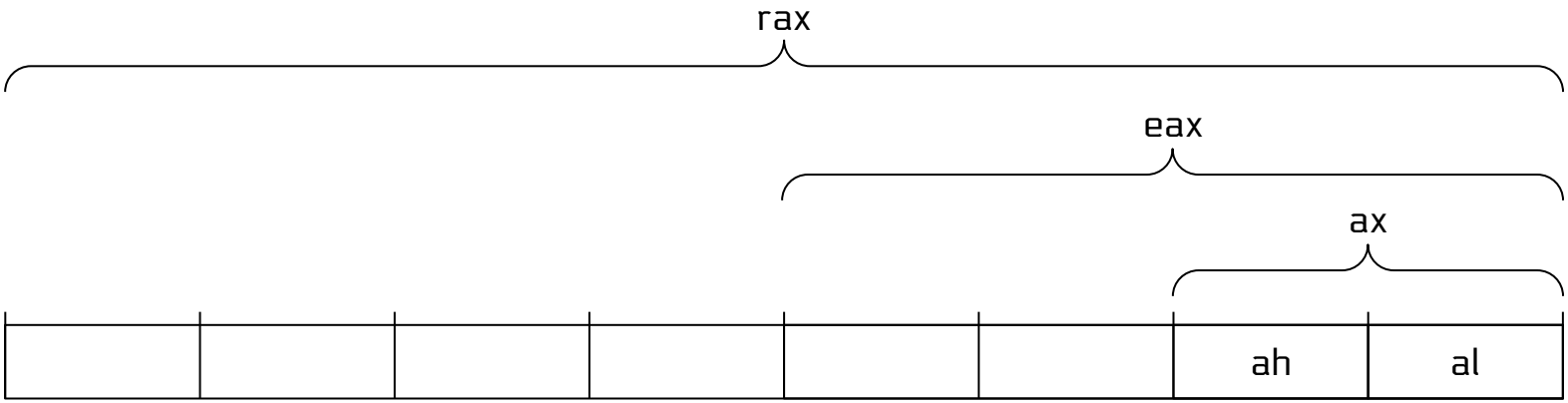
Register Size

Registers are (typically) the same size as the word width of the architecture.

On a 64-bit architecture (most) registers will hold 64 bits (8 bytes).

10110110	11011110	01111101	00000110	10110000	00111100	11110000	01000101

Partial Register Access



Registers can be accessed *partially*.

All partial accesses on amd64 (that I know of)

64	32	16	8H	8L
rax	eax	ax	ah	al
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rbx	ebx	bx	bh	bl
rsp	esp	sp		spl
rbp	ebp	bp		bpl
rsi	esi	si		sil
rdi	edi	di		dil
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

Setting Registers

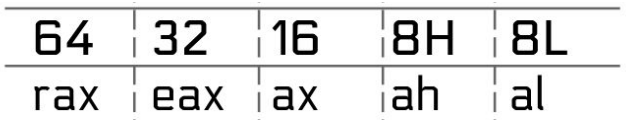
You load data into registers with... assembly! "**`mov`**" means "move".

```
mov rax, 0x539
mov rbx, 1337
```

Data specified directly in the instruction like this is called an **Immediate Value**.

You can also load data into partial registers:

```
mov ah, 0x5
mov al, 0x39
```



32-bit CAVEAT!

If you write to a 32-bit partial (e.g., **`eax`**), the CPU will *zero out* the rest of the register!
This was done for (believe it or not) performance reasons.

This sets **`rax`** to **`0xfffffffffffff0539`**:

```
mov rax, 0xffffffffffffffff
mov ax, 0x539
```

This sets **`rax`** to **`0x0000000000000539`**:

```
mov rax, 0xffffffffffffffff
mov eax, 0x539
```

Shunting Data Around

You can also `mov` data between registers!

LINGUISTIC CAVEAT!
"mov" doesn't move the data, it copies it.

This sets both `rax` and `rbx` to 0x539 (1337).

```
mov rax, 0x539
mov rbx, rax
```

You can, of course, `mov` partials (32-bit clobber caveat applies)!
This sets rax to 0x539 and rbx to 0x39.

```
mov rax, 0x539
mov rbx, 0
mov bl, al
```

Extending Data...

Consider:

```
mov eax, -1
```

eax is now `0xffffffff` (both `4294967295` and `-1`) but...
rax is now `0x00000000ffffffff` (*only* `4294967295`)!

What if you wanted the operate on that `-1` in 64-bit land?

```
mov eax, -1
movsx rax, eax
```

`movsx` does a *sign-extending* move, preserving the Two's Complement value (i.e., copies the top bit to the rest of the register).

eax is now `0xffffffff` (both `4294967295` and `-1`) but...
rax is now `0xffffffffffffffff` (both `4294967295` and `-1`)!

Register Arithmetic

Once you have data in registers, you can *compute*!

For most arithmetic instructions, the first specified register stores the result.

Instruction	C / Math equivalent	Description
`add rax, rbx`	`rax = rax + rbx`	add rax to rbx
`sub ebx, ecx`	`ebx = ebx - ecx`	subtract ecx from ebx
`imul rsi, rdi`	`rsi = rsi * rdi`	multiple rsi to rdi, truncate to 64-bits
`inc rdx`	`rdx = rdx + 1`	increment rdx
`dec rdx`	`rdx = rdx - 1`	decrement rdx
`neg rax`	`rax = 0 - rax`	negate rax in terms of numerical value
`not rax`	`rax = ~rax`	negate each bit of rax
`and rax, rbx`	`rax = rax & rbx`	bitwise AND between the bits of rax and rbx
`or rax, rbx`	`rax = rax | rbx`	bitwise OR between the bits of rax and rbx
`xor rcx, rdx`	`rcx = rcx ^ rdx`	bitwise XOR (don't confuse ^ for exponent!)
`shl rax, 10`	`rax = rax << 10`	shift rax's bits left by 10, filling with 10 zeroes on the right
`shr rax, 10`	`rax = rax >> 10`	shift rax's bits right by 10, filling with 10 zeroes on the left
`sar rax, 10`	`rax = rax >> 10`	shift rax's bits right by 10, *with sign-extension to fill the now "missing" bits!*
`ror rax, 10`	`rax = (rax >> 10) | (rax << 54)`	rotate the bits of rax right by 10
`rol rax, 10`	`rax = (rax << 10) | (rax >> 54)`	rotate the bits of rax left by 10

Curious how these work? Play around with the rappel tool (https://github.com/yrp604/rappel)!

Some Registers are Special

You cannot directly read from or write to `rip`.
Contains the memory address of the next instruction to be executed (ip = Instruction Pointer).

You should be careful with `rsp`.
Contains the address of an region of memory to store temporary data (sp = Stack Pointer).

Some other registers are, by convention, used for important things.

More on this later in this module!

Other Registers Exist!

Modern x86 processors have a lot of other registers!

Registers for use by the Operating System itself (stay tuned for Kernel Security!).

Registers for *floating point* computation.

Registers for crunching large data fast.
32 512-bit "zmm" registers!