Assembly Crash Course

Memory

Yan Shoshitaishvili
Arizona State University

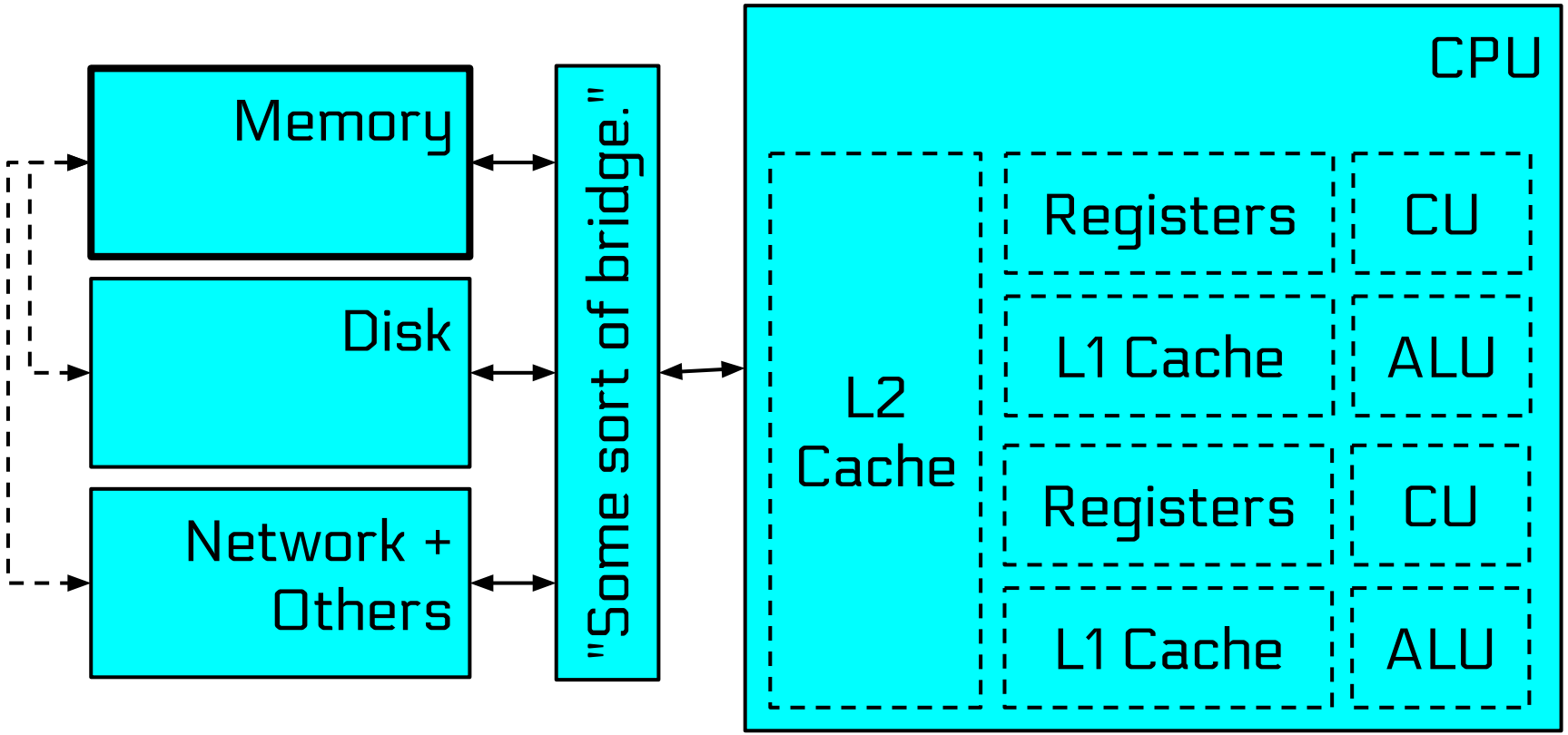
The Need for "Memory"

Registers are *expensive*, and we have a limited number of them.

We need a place to store lots of data and have *fairly fast* access to it when needed.

This place is system Memory.

Reminder: Computer Architecture



Memory: Process Perspective

Your process memory is used for A LOT:

Memory ↔ Registers
Memory ↔ Disk
Memory ↔ Network
Memory ↔ Video Card

There is too much memory to name every location (unlike registers).

Process memory is *addressed* linearly.
From: `0x10000` (for security reasons)
To: `0x7fffffffffff` (for architecture / OS purposes)

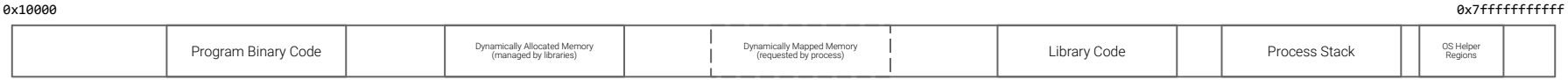
Each memory *address* references **one byte** in memory.
This means 127 *terabytes* of addressable RAM!

A Process' Memory

You don't have 127 TB of RAM... But that's okay, cause it's all ~~fake~~ ~~pretend~~ virtual!

Your process' memory starts out partially filled in by the Operating System.

Your process can ask for more memory from the Operating System (more on this later)!



Memory (stack)

The stack has several uses. For now, we'll talk about *temporary data storage*.

Registers and immediates can be **push**ed onto the stack to save values:

```
mov rax, 0xc001ca75
push rax
push 0xb0bacafe # WARNING: even on 64-bit x86, you can only push 32-bit immediates...
push rax
```

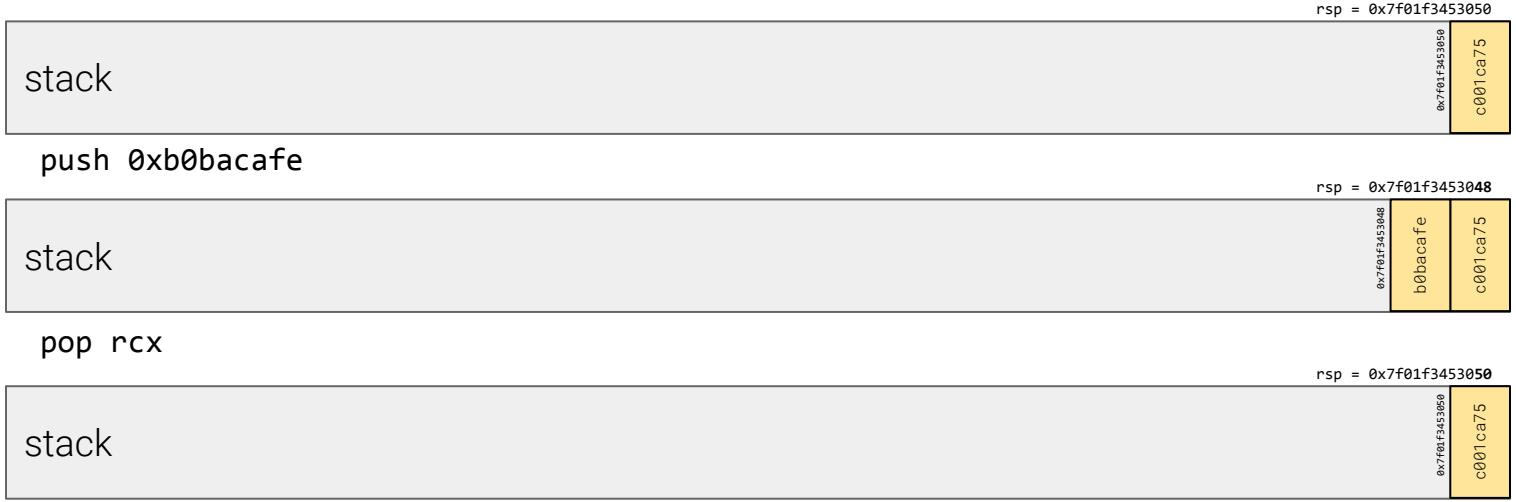
(Like mov, push leaves the value in the src register intact.)

Values can be **pop**ped back off of the stack (to any register!).

```
pop rbx # sets rbx to 0xc001ca75
pop rcx # sets rcx to 0xb0bacafe
```

Addressing the Stack

The CPU knows where the stack is because its address is stored in **`rsp`**.



Historical oddity: the stack grows backwards toward smaller memory addresses!

`push` *decreases* rsp, **`pop`** *increases it*.

Accessing Memory

You can also move data between registers and memory with ... `mov`!

This will load the 64-bit value stored at memory address `0x12345` into `rbx`:

```
mov rax, 0x12345
mov rbx, [rax]
```

This will store the 64-bit value in `rbx` into memory at address `0x133337`:

```
mov rax, 0x133337
mov [rax], rbx
```

This is equivalent to push `rcx`:

```
sub rsp, 8
mov [rsp], rcx
```

Each addressed memory location contains one byte.

An 8-byte write at address `0x133337` will write to addresses `0x133337` through `0x13333f`.

Controlling Write Sizes

You can use partials to store/load fewer bits!

Load 64 bits from addr `0x12345` and store the lower 32 bits to addr `0x133337`.

```
mov rax, 0x12345
mov rbx, [rax]
mov rax, 0x133337
mov [rax], ebx
```

Load 8 bits from addr `0x12345` to `bh`.

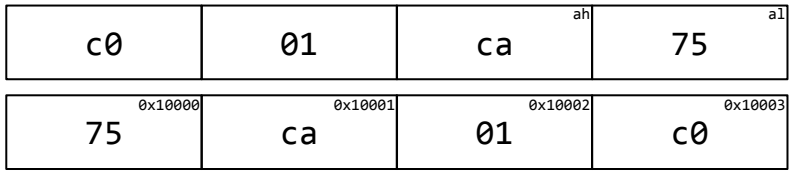
```
mov rax, 0x12345
mov bh, [rax]
```

Don't forget: changing 32-bit partials (e.g., by loading from memory) zeroes out the whole 64-register. Storing 32-bits to memory has no such problems, though.

Memory Endianess

Data on most modern systems is stored *backwards*, in *little endian*.

```
mov eax, 0xc001ca75 # sets rax to
mov rcx, 0x10000
mov [rcx], eax   # stores data as
mov bh, [rcx] # reads 0x75
```



Bytes are *only* shuffled for multi-byte stores and loads of registers to memory!
Individual bytes *never* have their bits shuffled.
Yes, writes to the stack behave just like any other write to memory.

Why little endian?

Intel created the 8008 for a company called Datapoint in 1972.
Datapoint used little endian for easier implementation of *carry* in arithmetic!
Intel used little endian in 8008 for compatibility with Datapoint's processes!
Every step in the evolution between 8008 and modern x86 maintained some level of binary compatibility with its predecessor.

https://en.wikipedia.org/wiki/Endianness

Address Calculation

You can do some limited calculation for memory addresses.

Use **`rax`** as an offset off some base address (in this case, the stack).

```
mov rax, 0
mov rbx, [rsp+rax*8] # read a qword right at the stack pointer
inc rax
mov rcx, [rsp+rax*8] # read the qword to the right of the previous one
```

You can get the calculated address with Load Effective Address (**`lea`**).

```
mov rax, 1
pop rcx
lea rbx, [rsp+rax*8+5] # rbx now holds the computed address for double-checking
mov rbx, [rbx]
```

Address calculation has limits.
`reg+reg*(2 or 4 or 8)+value` is as good as it gets.

RIP-Relative Addressing

`lea` is one of the few instructions that can directly access the rip register!

```
lea rax, [rip] # load the address of the next instruction into rax
lea rax, [rip+8] # the address of the next instruction, plus 8 bytes
```

You can also use `mov` to read directly from those locations!

```
mov rax, [rip] # load 8 bytes from the location pointed to by the address of the next instruction
```

Or even *write* there!

```
mov [rip], rax # write 8 bytes over the next instruction (CAVEATS APPLY)
```

This is useful for working with data embedded near your code!

This is what makes certain security features on modern machines *possible*.

Writing Immediate Values

You can also write immediate values. However, you must specify their size!

This writes a 32-bit 0x1337 (padded with 0 bits) to address 0x133337.

```
mov rax, 0x133337
mov DWORD PTR [rax], 0x1337
```

Depending on your assembler, it might expect `DWORD` instead of `DWORD PTR`.

Other Memory Regions

Other regions might be mapped in memory!

We previously talked about regions loaded due to directives in the ELF headers, but functionality such as `mmap` and `malloc` can cause other regions to be mapped as well.

These will feature prominently (and be discussed) in future modules.