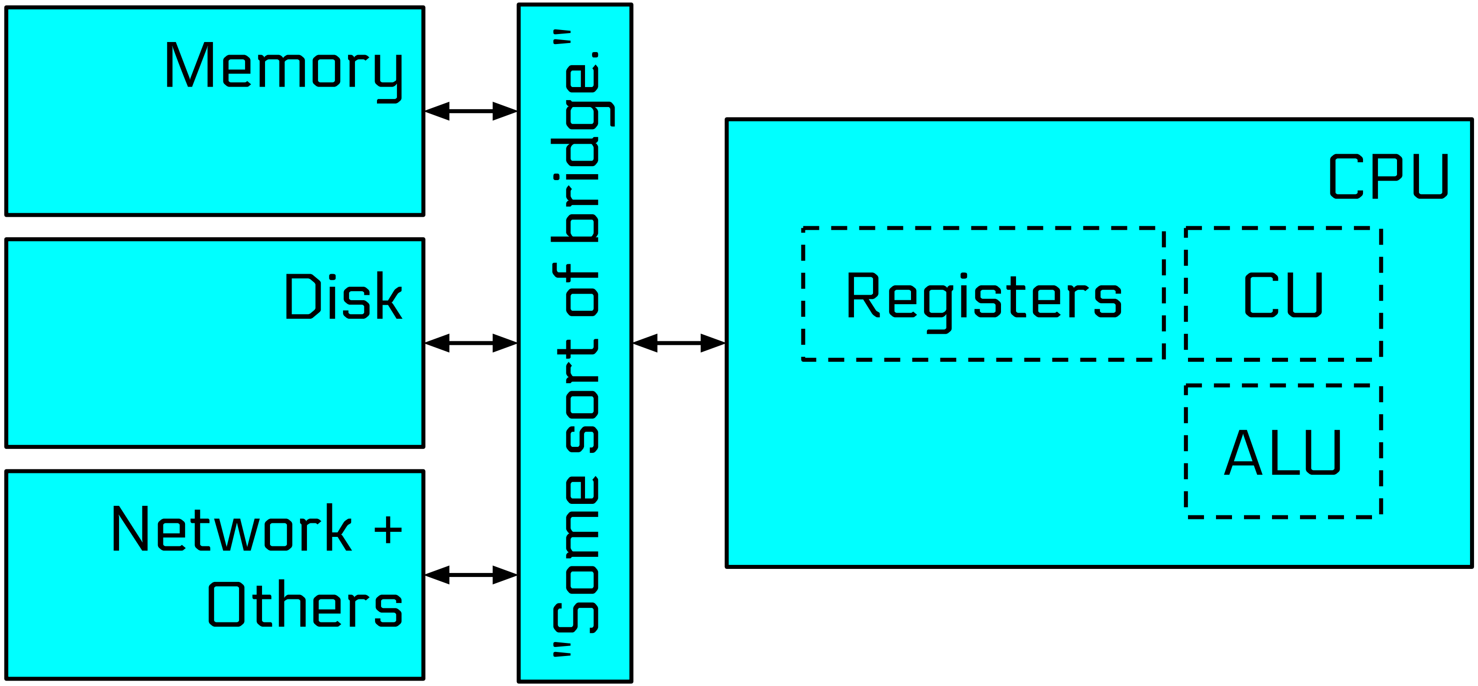
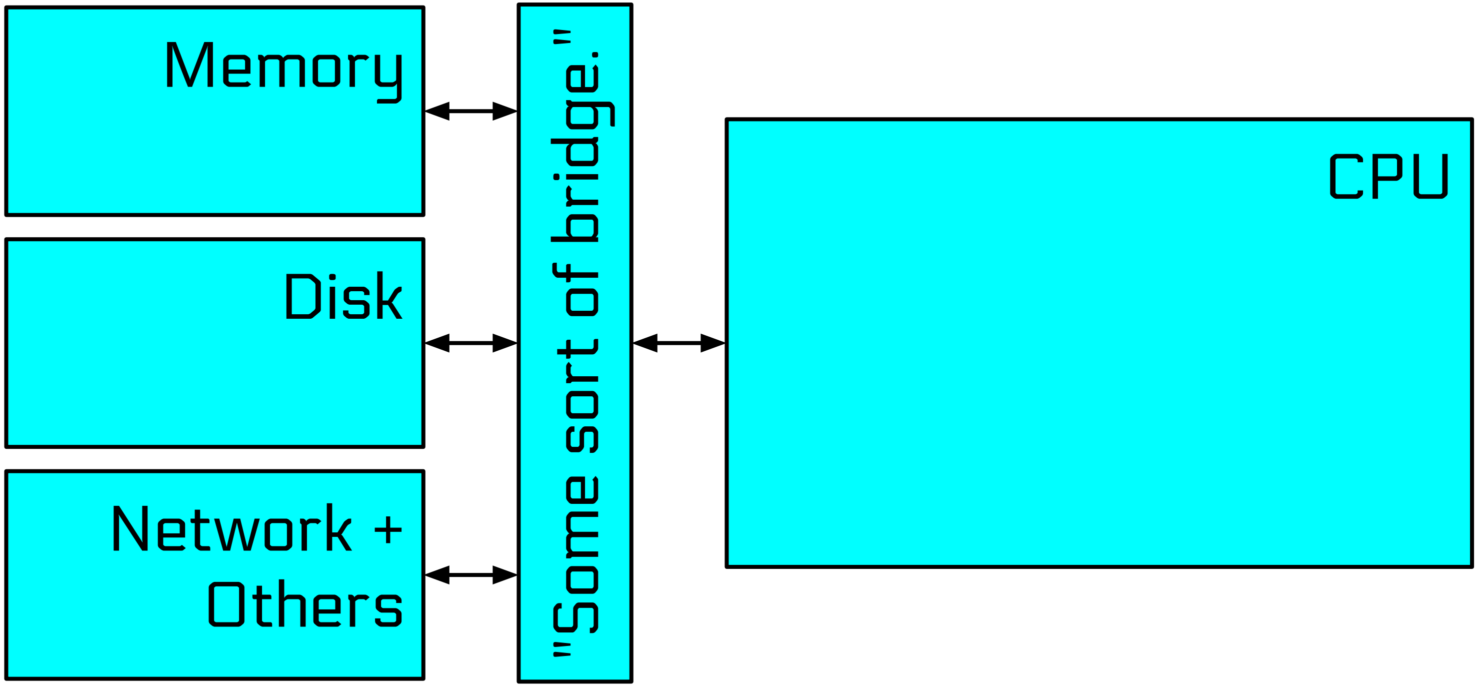
Assembly Crash Course

Computer Architecture

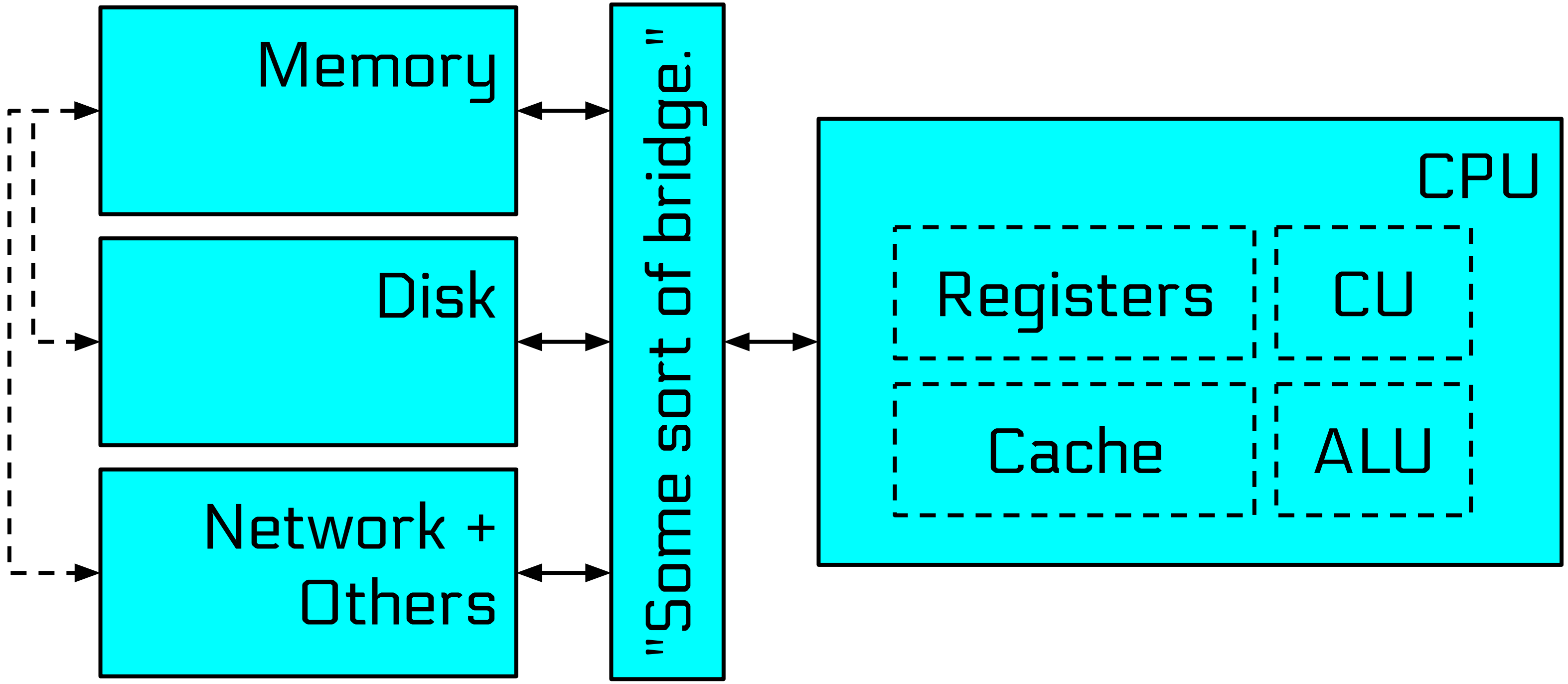
Computer Architecture (drilling down)



Computer Architecture (very high level)



Computer Architecture (further down!)



Assembly

The only *true* programming language, as far as a CPU is concerned.

Concepts:

- instructions
  - data manipulation instructions
  - comparison instructions
  - control flow instructions
  - system calls
- registers
- memory
  - program
  - stack
  - other mapped mem

Registers

Registers are very fast, temporary stores for data.

You get several "general purpose" registers:

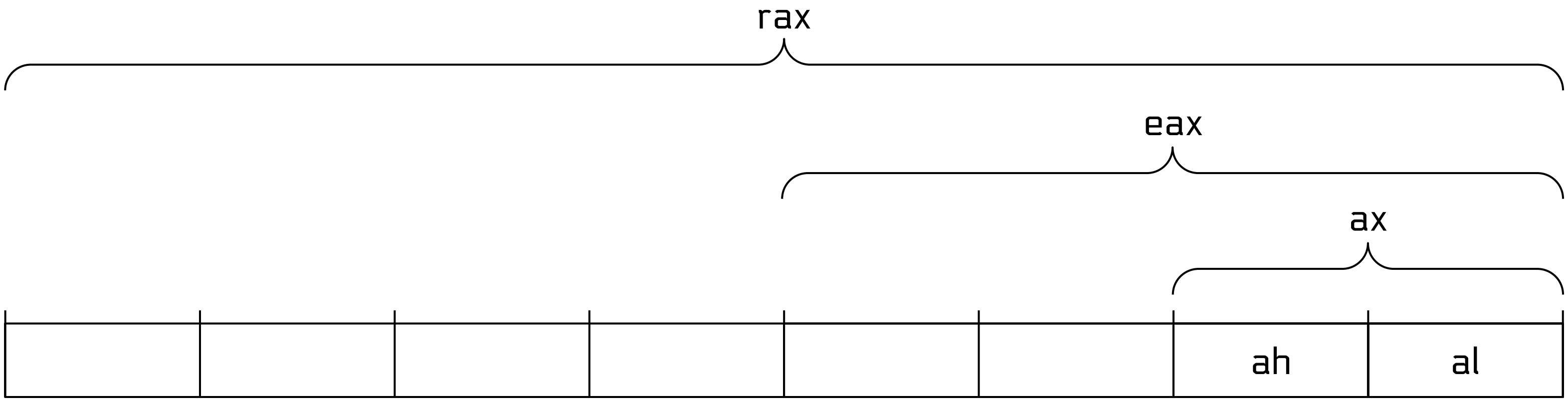
- 8085: a, c, d, b, e, h, l
- 8086: ax, cx, dx, bx, **sp**, **bp**, si, di
- x86: eax, ecx, edx, ebx, **esp**, **ebp**, esi, edi
- amd64: rax, rcx, rdx, rbx, **rsp**, **rbp**, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15
- arm: r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, **r13**, **r14**

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).

Partial Register Access



Registers can be accessed *partially*.

Due to a historical oddity, accessing `eax` will sign-extend out the rest of `rax`.
Other partial access preserve untouched parts of the register.

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

Instructions

General form:

```
OPCODE OPERAND OPERAND, ...
```

`OPCODE` - what to do

`OPERANDS` - what to do it on/with

```
mov rax, rbx
add rax, 1
cmp rax, rbx
jb some_location
```

`Useful reference:` http://ref.x86asm.net

Instructions (data manipulation)

Instructions can move and manipulate data in registers and memory.

```
mov rax, rbx
mov rax, [rbx+4]
add rax, rbx
mul rsi
inc rax
inc [rax]
```

Instructions (control flow)

Control flow is determined by conditional and unconditional jumps.

Unconditional: call, jmp, ret

Conditional:

je	jump if equal
jne	jump if not equal
jg	jump if greater
jl	jump if less
jle	jump if less than or equal
jge	jump if greater than or equal
ja	jump if above (unsigned)
jb	jump if below (unsigned)
jae	jump if above or equal (unsigned)
jbe	jump if below or equal (unsigned)
js	jump if signed
jns	jump if not signed
jo	jump if overflow
jno	jump if not overflow
jz	jump if zero
jnz	jump if not zero

```
cmp rax, rbx
jb some_location 🤔
```

Instructions (conditionals)

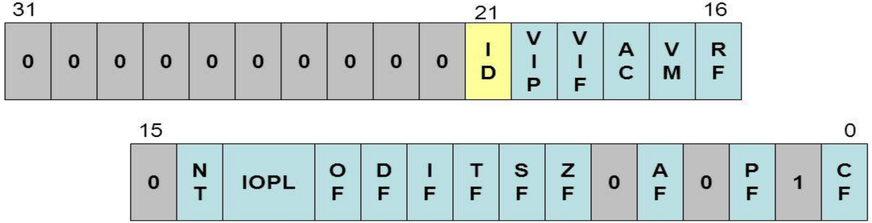
Conditionals key off of the "flags" register:

- eflags (x86), rflags (amd64), aspr (arm).

Updated by (x86/amd64):

- arithmetic operations
- cmp - subtraction (`cmp rax, rbx`)
- test - and (`test rax, rax`)

je	jump if equal	ZF=1
jne	jump if not equal	ZF=0
jg	jump if greater	ZF=0 and SF=OF
jl	jump if less	SF!=OF
jle	jump if less than or equal	ZF=1 or SF!=OF
jge	jump if greater than or equal	SF=OF
ja	jump if above (unsigned)	CF=0 and ZF=0
jb	jump if below (unsigned)	CF=1
jae	jump if above or equal (unsigned)	CF=0
jbe	jump if below or equal (unsigned)	CF=1 or ZF=1
js	jump if signed	SF=1
jns	jump if not signed	SF=0
jo	jump if overflow	OF=1
jno	jump if not overflow	OF=0
jz	jump if zero	ZF=1
jnz	jump if not zero	ZF=0



Instructions (system calls)

Almost all programs have to interact with the outside world!

This is primarily done via *system calls* (`man syscalls`). Each system call is well-documented in section 2 of the man pages (i.e., `man 2 open`).

System calls (on amd64) are triggered by:

1. set `rax` to the *system call number*
2. store arguments in `rdi`, `rsi`, etc (more on this later)
3. call the `syscall` instruction

We can trace process system calls using `strace`.

System Calls

System calls have very well-defined interfaces that very rarely change.

There are over 300 system calls in Linux. Here are some examples:

`int open(const char *pathname, int flags)` - returns a file new file descriptor of the open file (also shows up in /proc/self/fd!)
`ssize_t read(int fd, void *buf, size_t count)` - reads data from the file descriptor
`ssize_t write(int fd, void *buf, size_t count)` - writes data to the file descriptor
`pid_t fork()` - forks off an *identical* child process. Returns 0 if you're the child and the PID of the child if you're the parent.
`int execve(const char *filename, char **argv, char **envp)` - *replaces* your process.
`pid_t wait(int *wstatus)` - wait child termination, return its PID, write its status into *wstatus.

Typical signal combinations:

- fork, execve, wait (think: a shell)
- open, read, write (cat)

Critical resource: https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

Memory (stack)

The stack fulfils four main uses:

1. Track the "callstack" of a program.
    a. return values are "`pushed`" to the stack during a call and "`popped`" during a ret
2. Contain local variables of functions.
3. Provide scratch space (to alleviate register exhaustion).
4. Pass function arguments (always on x86, only for functions with "many" arguments on other architectures).

Relevant registers (amd64): rsp, rbp

Relevant instructions (amd64): push, pop

Calling Conventions

Callee and caller functions must agree on argument passing.

Linux x86: push arguments (in reverse order), then call (which pushes return address), return value in eax

Linux amd64: rdi, rsi, rdx, rcx, r8, r9, return value in rax

Linux arm: r0, r1, r2, r3, return value in r0

Registers are *shared* between functions, so calling conventions should agree on what registers are protected.

Linux amd64: rbx, rbp, r12, r13, r14, r15 are "callee-saved"