

x86 ISA

PC hardware

x86 instruction set architecture

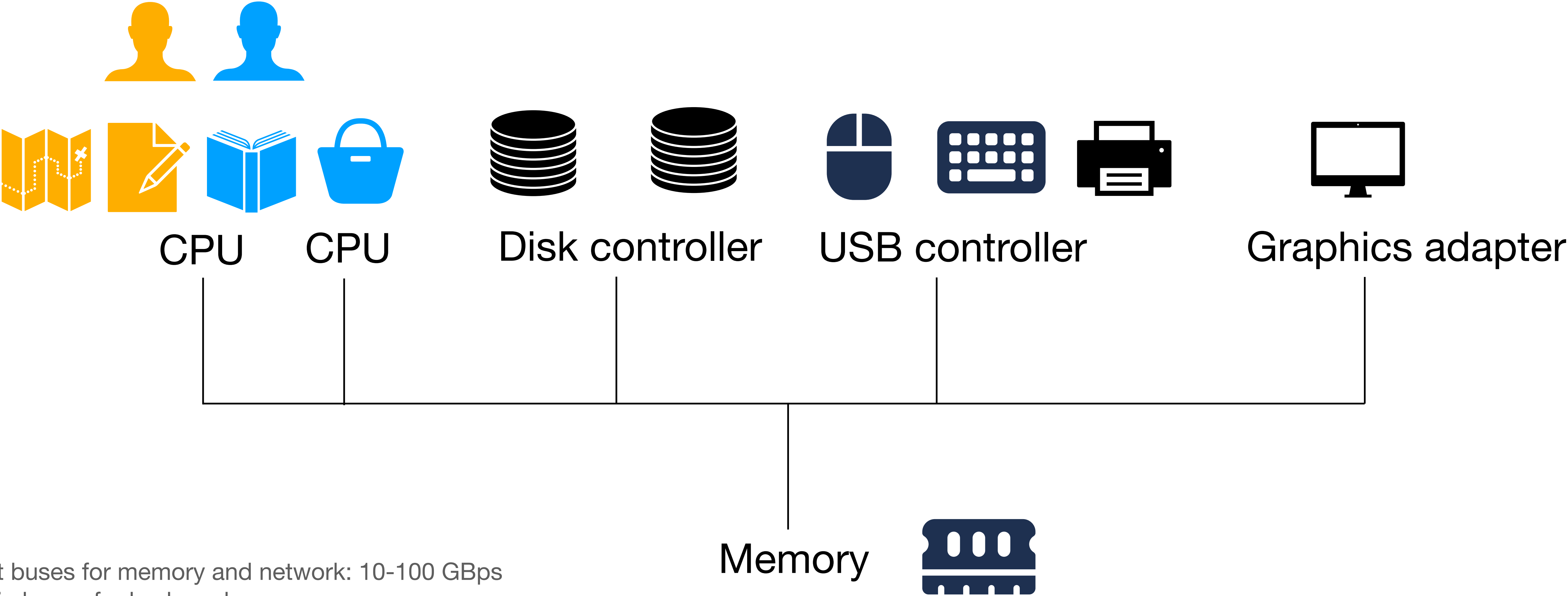
gcc calling convention

ELF (Executable and Linkable Format)

Agenda

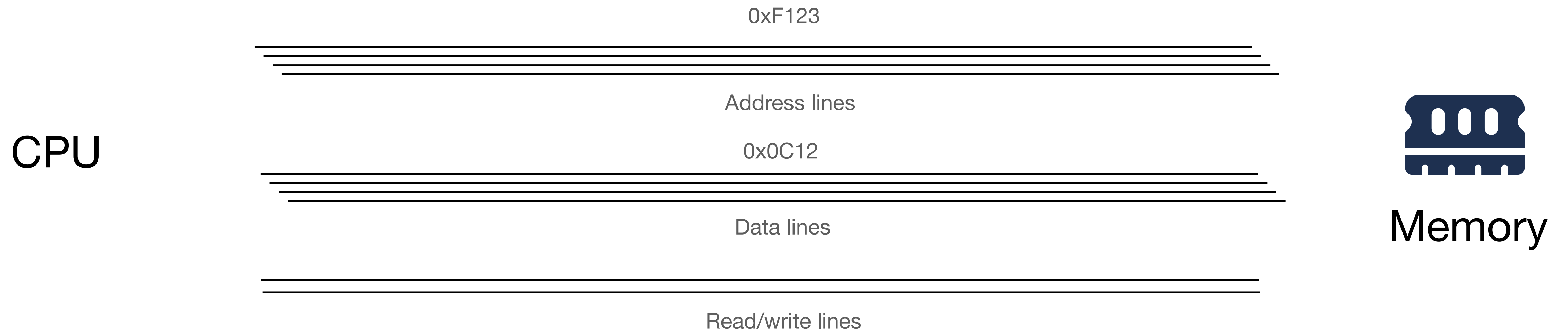
- Build a mental model of how hardware components (e.g., CPU and memory, CPU and IO) interact with one another
- x86 instruction set architecture: software-hardware interface defined by Intel in early 1980s. Has become a standard.
 - CPU may have internal state (pipeline registers) and optimizations (branch prediction, out-of-order execution) not part of ISA, i.e., not visible to the software
 - Understand x86 instruction set so that we can read and write x86 assembly. Assembly programs are sometimes required by OS to get fine-grained control of the hardware
- Understand gcc calling convention so that we can call C programs from assembly and vice-versa
- How are programs loaded from disk and run? Our boot loader will load and run our OS

Computer organization



Fat buses for memory and network: 10-100 GBps
Thin buses for keyboard, mouse

CPU-memory interaction



- Each read/write takes ~100 cycles
- Faster memory: on-chip registers ~1 cycle.

Registers

- **General purpose registers.**
 - **%eax, %ebx, %ecx, %edx**
 - **%edi: destination index, %esi: source index**
- Flags register. %eflags
- Instruction pointer. %eip
- Stack registers. %ebp: base pointer, %esp: stack pointer
- Special registers.
 - Control registers %cr0, %cr2, %cr3, %cr4;
 - Segment registers %cs, %ds, %es, %fs, %gs, %ss
 - Table registers: global descriptor table %gdtr, local descriptor table %ldtr, interrupt descriptor table %idtr
- Other registers not used in xv6: 8 80-bit floating point registers, debug registers

mov instructions

Intel SDM Vol 1 7.3.1.1

Assembly	“C” equivalent
movl %eax, %edx	edx = eax
movl \$0x123, %edx	edx=0x123
movl 0x123, %edx	edx = *(int32_t*)0x123
movl (%ebx), %edx	edx=*(int32_t*) ebx
movl 4(%ebx), %edx	edx=*(int32_t*)(ebx+4)

Assembly	“C” equivalent
movsb	*edi = *esi; edi++; esi++;

Other instruction variants

General-Purpose Registers						
31	16	15	8	7	0	
			AH		AL	16-bit
			BH		BL	AX
			CH		CL	BX
			DH		DL	CX
			BP			DX
			SI			32-bit
			DI			EAX
			SP			EBX
						ECX
						EDX
						EBP
						ESI
						EDI
						ESP

Figure 3-5. Alternate General-Purpose Register Names

- movw: moves 2 bytes (%ax)
- movb: moves 1 byte (%al, %ah)

Many other instructions: ADD, SUB, MUL, DIV, ...

Registers

- General purpose registers.
 - `%eax`, `%ebx`, `%ecx`, `%edx`
 - `%edi`: destination index, `%esi`: source index
- **Flags register. `%eflags`**
- Instruction pointer. `%eip`
- Stack registers. `%ebp`: base pointer, `%esp`: stack pointer
- Special registers.
 - Control registers `%cr0`, `%cr2`, `%cr3`, `%cr4`;
 - Segment registers `%cs`, `%ds`, `%es`, `%fs`, `%gs`, `%ss`
 - Table registers: global descriptor table `%gdtr`, local descriptor table `%ldtr`, interrupt descriptor table `%idtr`
- Other registers not used in xv6: 8 80-bit floating point registers, debug registers

EFLAGS

- Carry flag: Most significant bit overflowed.

```
movl $0xFFFFFFFF %eax
addl %eax, %eax
```

```
eax = 0xFFFFFFFF
eax = eax + eax
```

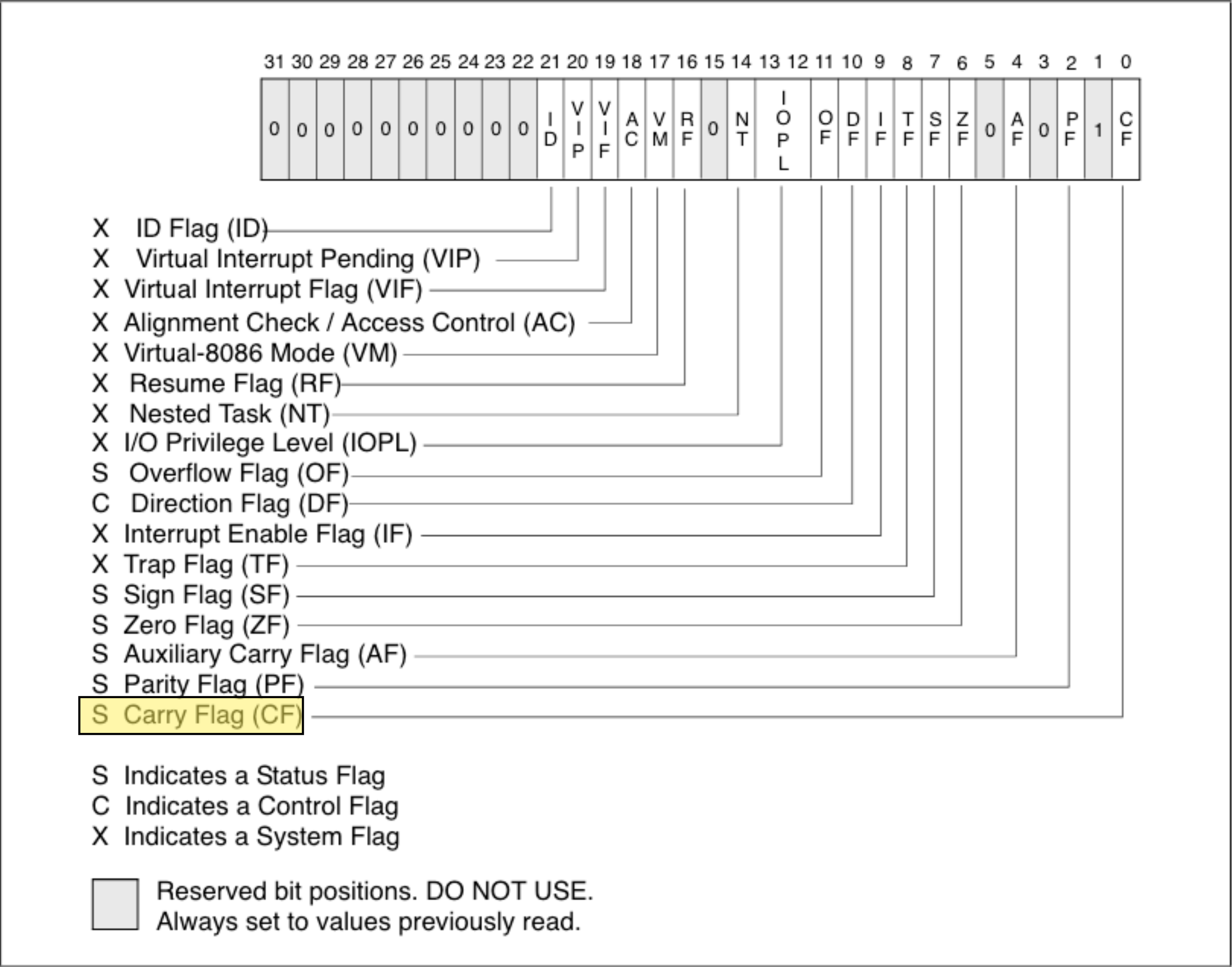


Figure 3-8. EFLAGS Register

EFLAGS (2)

- Zero flag: Set if result is zero.

```
xorl %eax, %eax
```

```
eax = eax xor eax
```

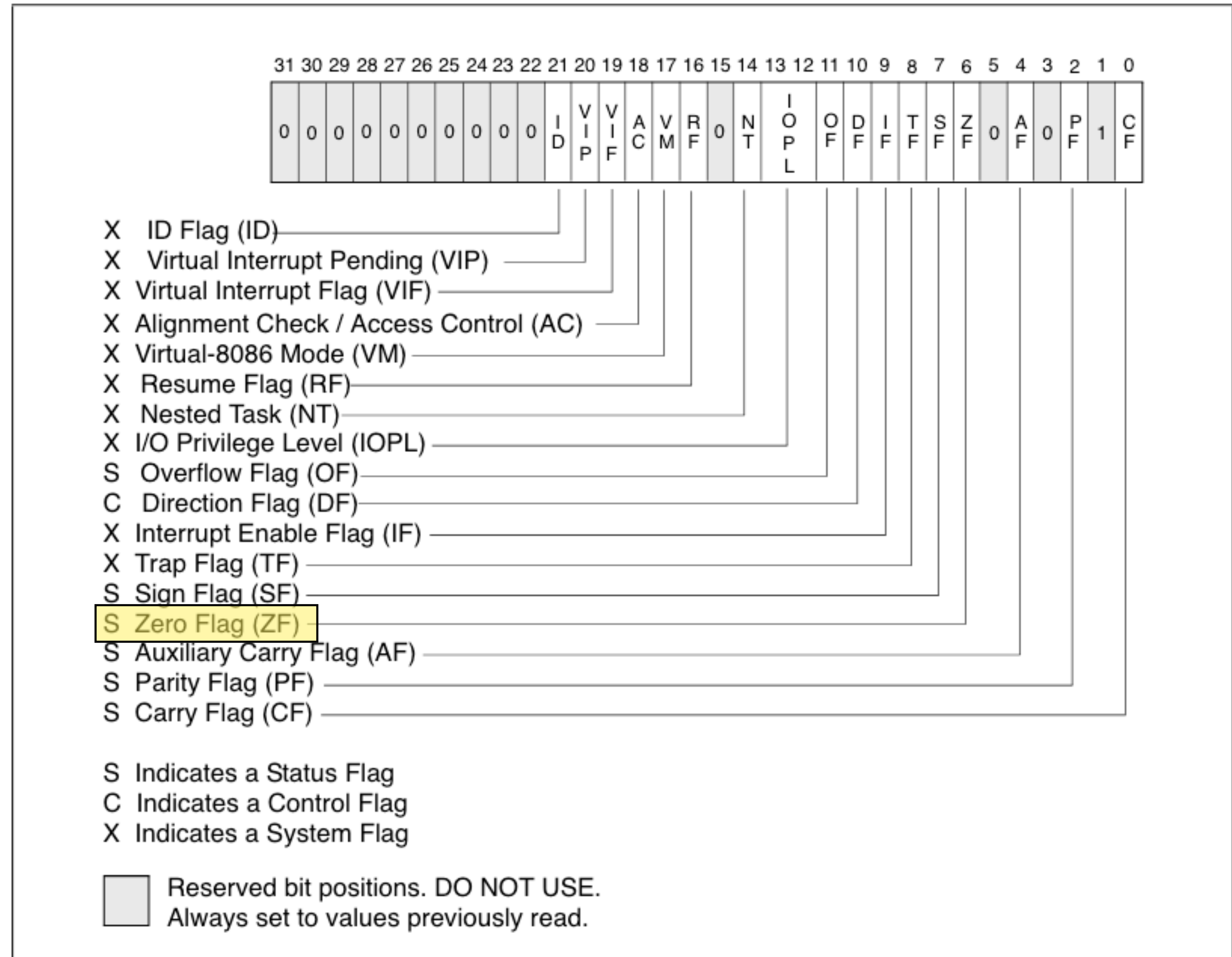


Figure 3-8. EFLAGS Register

EFLAGS (3)

- Sign flag: Equal to the most significant bit of the result (which is the sign bit of a signed integer)

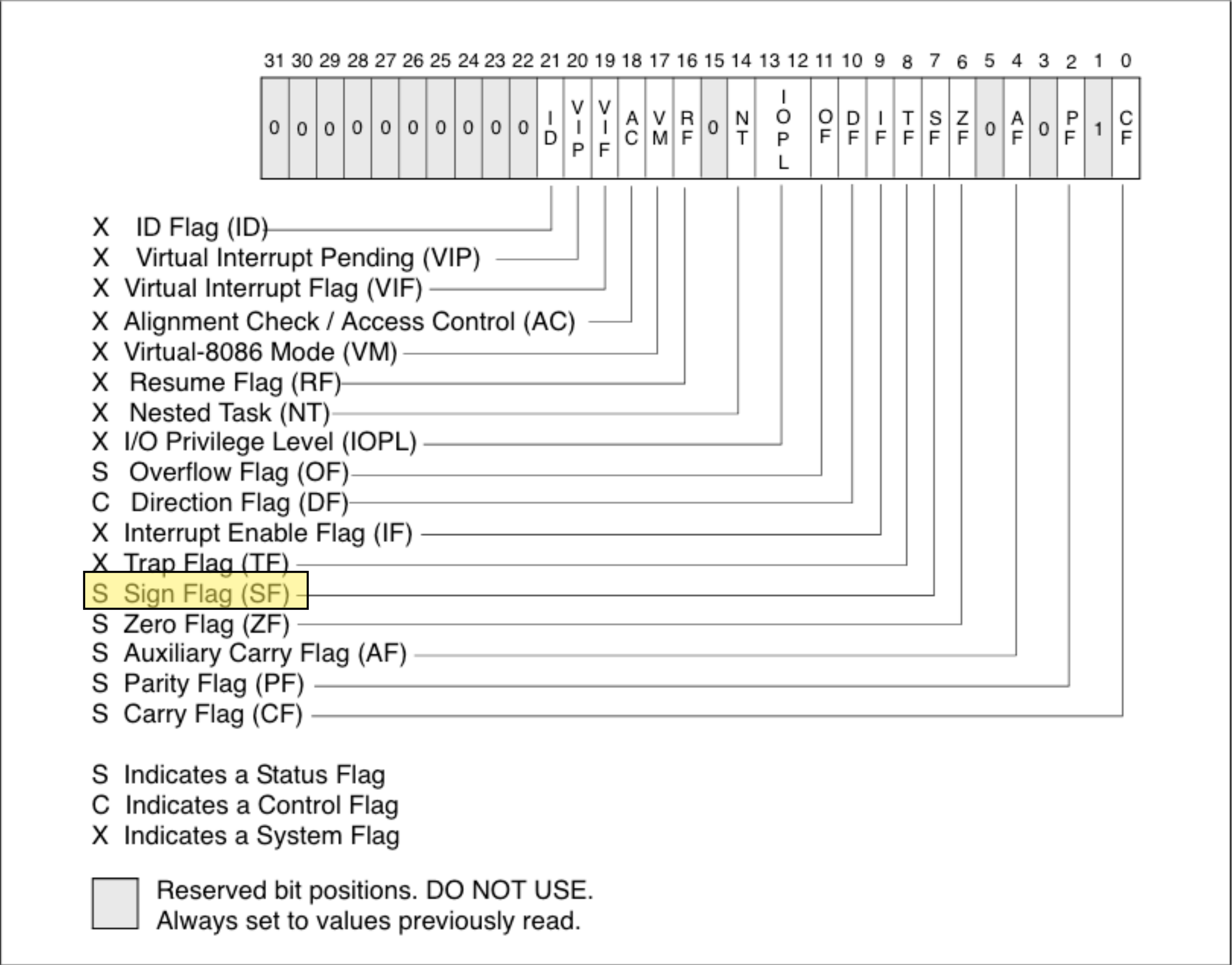



Figure 3-8. EFLAGS Register

Registers in action

02.flags.c

```
int foo(int x, int y) {  
    int z = x + y;  
    if(z % 2 == 0)  
        return x;  
    return y;  
}
```

gcc -m32 -S -O1 02.flags.c



02.flags.s

```
foo:  
    movl 4(%esp), %eax    # eax = x  
    movl %eax, %edx      # edx = eax (z = x)  
    addl 8(%esp), %edx    # edx += y  
    andl $1, %edx        # edx = (edx & 1). ZF if edx is even.  
    cmovne 8(%esp), %eax  # eax = y if !ZF (conditional move)  
    ret
```

Function parameters are put on stack:
(%esp): contains return address
4(%esp): contains x
8(%esp): contains y

%eax shall contain the return value

Registers

- General purpose registers.
 - `%eax`, `%ebx`, `%ecx`, `%edx`
 - `%edi`: destination index, `%esi`: source index
- Flags register. `%eflags`
- **Instruction pointer. `%eip`**
- Stack registers. `%ebp`: base pointer, `%esp`: stack pointer
- Special registers.
 - Control registers `%cr0`, `%cr2`, `%cr3`, `%cr4`;
 - Segment registers `%cs`, `%ds`, `%es`, `%fs`, `%gs`, `%ss`
 - Table registers: global descriptor table `%gdtr`, local descriptor table `%ldtr`, interrupt descriptor table `%idtr`
- Other registers not used in xv6: 8 80-bit floating point registers, debug registers

Instruction pointer

- Next instruction is pointed to by instruction pointer %eip

```
for(;;){  
    run next instruction  
}
```

- %eip is simply incremented in most cases
- Except special instructions
 - JMP 0x1234: changes %eip to 0x1234 e.g., while loop
 - JZ, JNZ, etc: jump if last result was zero, non-zero, etc. This uses bits from EFLAGS register. e.g, while(x != 0) { .. }
 - CALL 0x1234: Similar to JMP, additionally saves the current instruction pointer on stack e.g., function call
 - RET: returns back to caller. Changes %eip to address in stack

Registers in action (2)

02.eip.c

```
int exponent(int x, int y) {  
    int z = x;  
    while(y > 0) {  
        z = z * x;  
        y --;  
    }  
    return z;  
}
```

gcc -m32 -S -O1 02.eip.c

exponent:

```
    movl 4(%esp), %ecx  
    movl 8(%esp), %eax  
    movl %ecx, %edx  
    testl %eax, %eax
```

```
    jle .L1
```

.L3:

```
    imull %ecx, %edx  
    subl $1, %eax  
    jne .L3
```

.L1:

```
    movl %edx, %eax  
    ret
```

ecx = x

eax = y

edx = ecx (z = x)

bitwise and eax with eax.

SF if eax<0. ZF if eax=0.

Jump if SF or ZF (y <= 0)

z = z*x

eax-- (y--). ZF if eax=0 (y=0)

Jump back to loop if !ZF

eax = edx (return z)

Registers

- General purpose registers.
 - `%eax`, `%ebx`, `%ecx`, `%edx`
 - `%edi`: destination index, `%esi`: source index
- Flags register. `%eflags`
- Instruction pointer. `%eip`
- **Stack registers. `%ebp`: base pointer, `%esp`: stack pointer**
- Special registers.
 - Control registers `%cr0`, `%cr2`, `%cr3`, `%cr4`;
 - Segment registers `%cs`, `%ds`, `%es`, `%fs`, `%gs`, `%ss`
 - Table registers: global descriptor table `%gdtr`, local descriptor table `%ldtr`, interrupt descriptor table `%idtr`
- Other registers not used in xv6: 8 80-bit floating point registers, debug registers

Stack pointers

- Stack grows downwards
- `%ebp` points to return address
- `%esp` points to top of stack

<code>pushl %eax</code>	<code>subl \$4, %esp</code> <code>movl %eax, (%esp)</code>
<code>popl %eax</code>	<code>movl (%esp), %eax</code> <code>addl \$4, %esp</code>

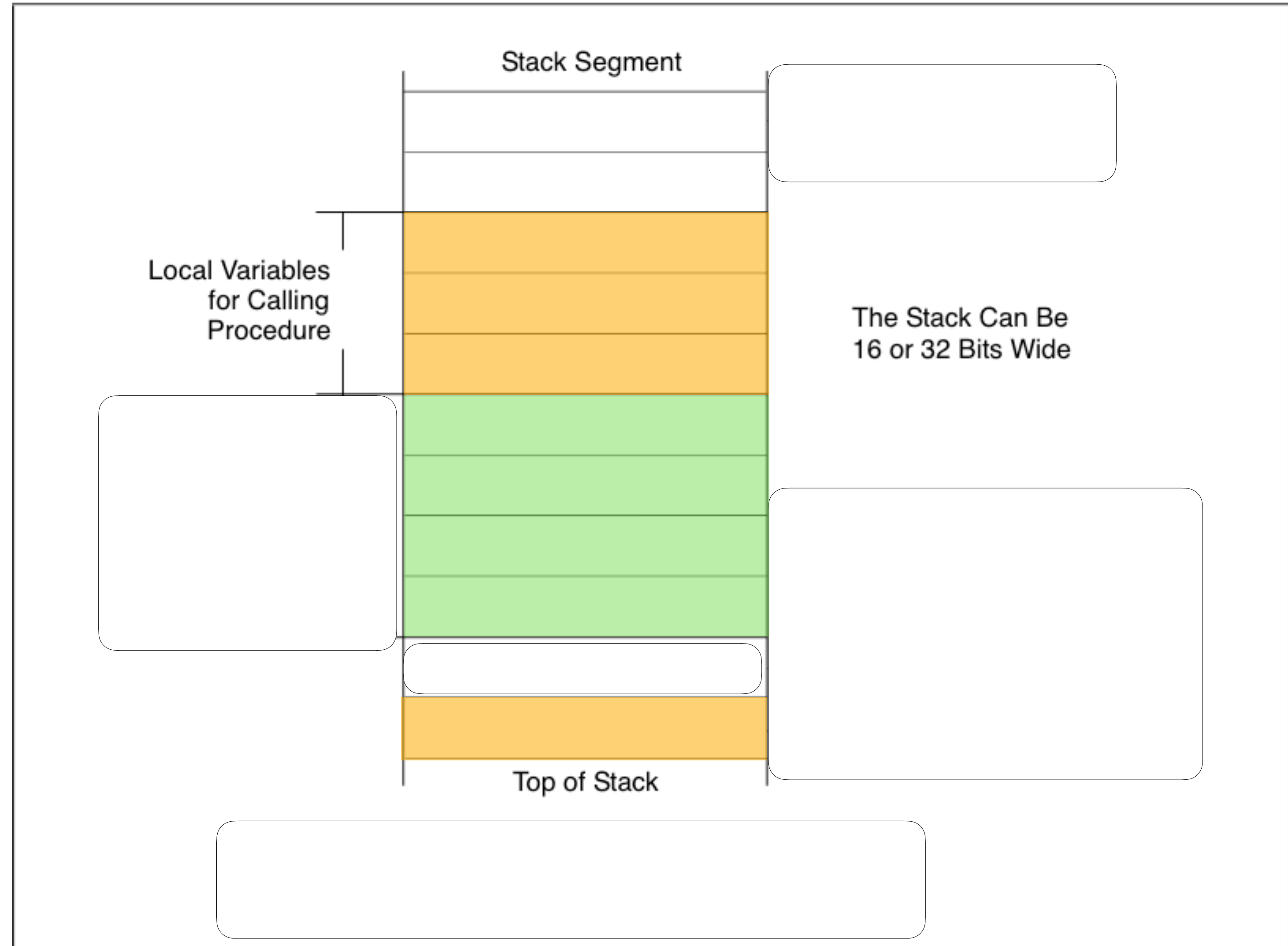


Figure 6-1. Stack Structure

Calling a function

main -> foo(x, y) -> bar(z)

- main pushes foo's parameters (x, y) on the stack
- Executes CALL instruction to save return address on the stack and jump %eip to first instruction of foo
 - foo reads parameters from the stack into registers, does computation on them
 - foo pushes bar's parameters (z) on the stack, executes CALL instruction
 - bar reads z from the stack into registers, does computation on them
 - Executes RET instruction to jump %eip to return address in the function foo
- foo executes RET instruction

Function calling in action

02.c

```
int foo(int x, int y) {
    return x + y;
}

int main() {
    return foo(41, 42);
}
```

gcc -m32 -S 02.c

pushl %eax	subl \$4, %esp movl %eax, (%esp)
popl %eax	movl(%esp), %eax addl \$4, %esp

02.s

```
_foo:
    pushl %ebp                # Save caller's base pointer
    movl %esp, %ebp          # ebp = esp
    movl 8(%ebp), %eax         # eax = *(ebp + 8)
    addl 12(%ebp), %eax        # eax = eax + *(ebp + 12)
    popl %ebp                 # Restore caller's base pointer
    retl                      # change eip to return address

    .globl _main              ## -- Begin function main
    .p2align 4, 0x90
_main:
    pushl %ebp                # Save caller's base pointer
    movl %esp, %ebp          # ebp = esp
    subl $24, %esp           # esp = esp - 24
    movl $0, -4(%ebp)         # *(ebp-4) = 0
    movl $41, (%esp)          # *(esp) = 41
    movl $42, 4(%esp)         # *(esp+4) = 42
    calll _foo                # Push current eip on to stack, jump to foo
    addl $24, %esp            # esp = esp + 24 (Restore caller's esp)
    popl %ebp                 # Restore caller's ebp
    retl
```


Function calling in action

Stack

02.s

eip

→

_foo:

pushl %ebp

movl %esp, %ebp

movl 8(%ebp), %eax

addl 12(%ebp), %eax

popl %ebp

retl

.globl _main

.p2align 4, 0x90

_main:

pushl %ebp

movl %esp, %ebp

subl \$24, %esp

movl \$0, -4(%ebp)

movl \$41, (%esp)

movl \$42, 4(%esp)

calll _foo

addl \$24, %esp

popl %ebp

retl

Save caller's base pointer

ebp = esp

eax = *(ebp + 8)

eax = eax + *(ebp + 12)

Restore caller's base pointer

change eip to return address

-- Begin function main

Save caller's base pointer

ebp = esp

esp = esp - 0x18

*(ebp-4)=0

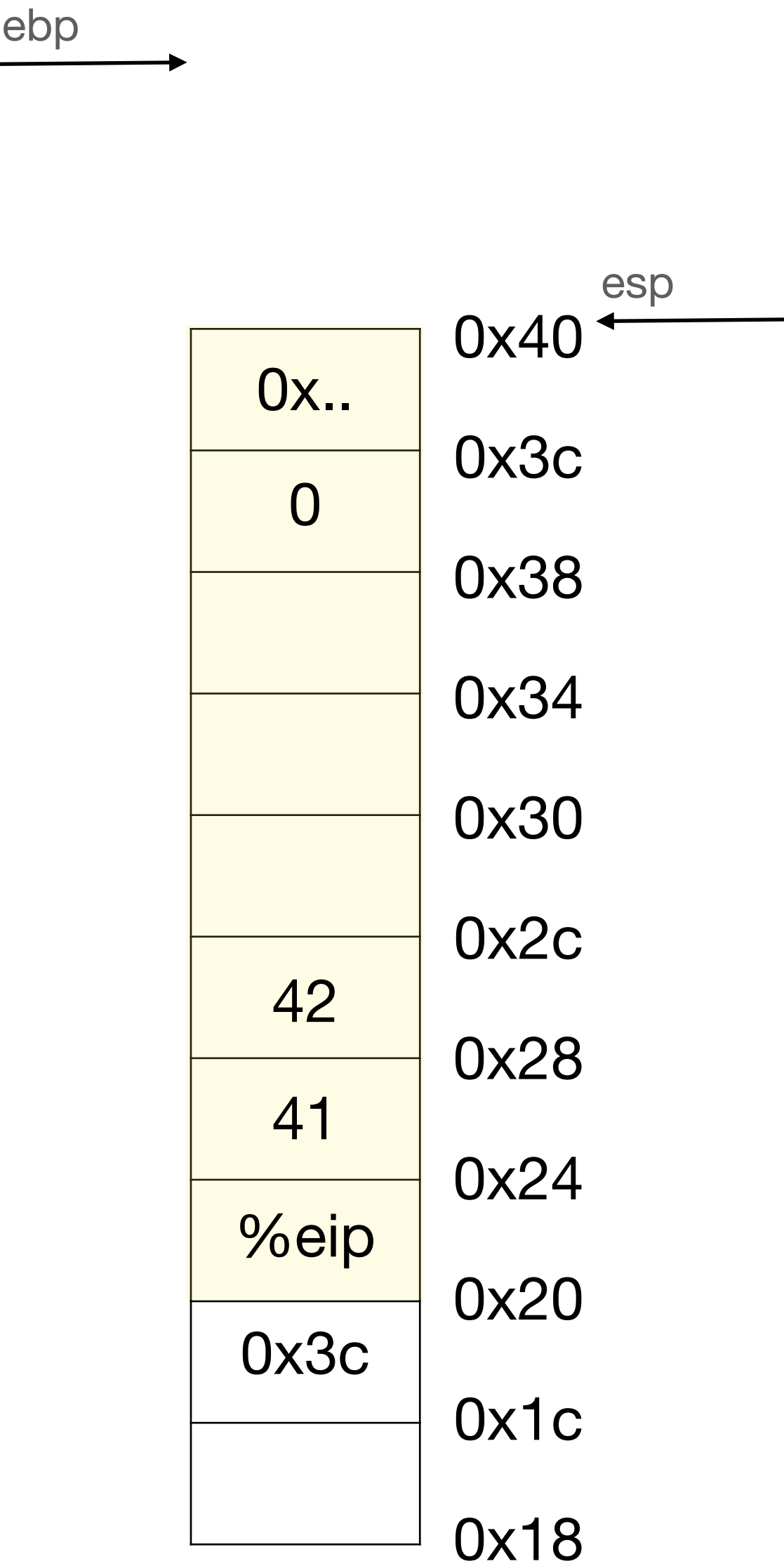
*(esp) = 41

*(esp+4) = 42

Push current eip on to stack, jump to foo

esp = esp + 24 (Restore caller's esp)

Restore caller's ebp



Function calling in action

Generating backtrace

02.s

eip→

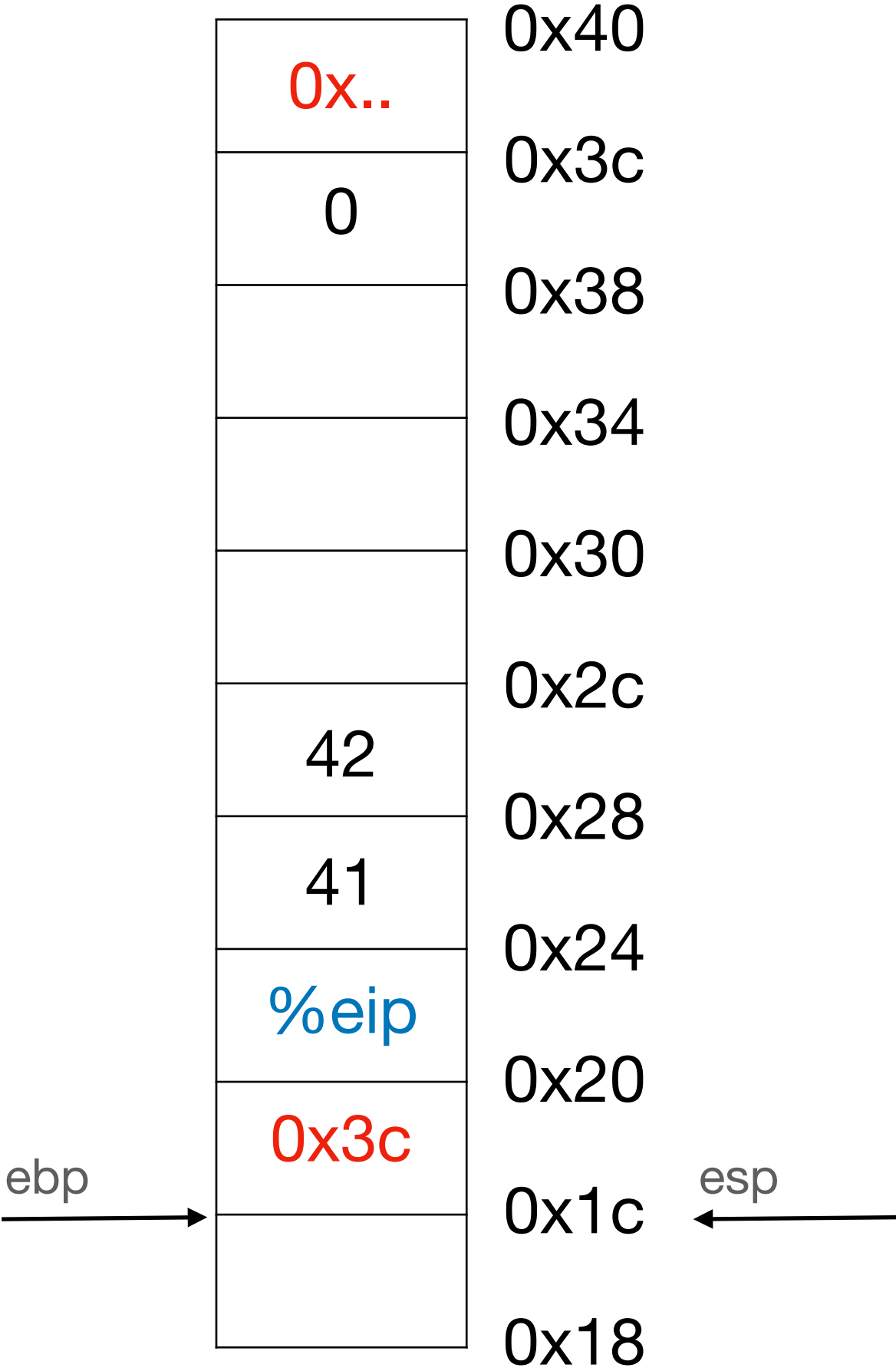
`_foo:
 pushl %ebp
 movl %esp, %ebp
 movl 8(%ebp), %eax
 addl 12(%ebp), %eax
 popl %ebp
 retl

 .globl _main
 .p2align 4, 0x90
_main:
 pushl %ebp
 movl %esp, %ebp
 subl $24, %esp
 movl $0, -4(%ebp)
 movl $41, (%esp)
 movl $42, 4(%esp)
 calll _foo
 addl $24, %esp
 popl %ebp
 retl`

Save caller's base pointer
ebp = esp
eax = *(ebp + 8)
eax = eax + *(ebp + 12)
Restore caller's base pointer
change eip to return address

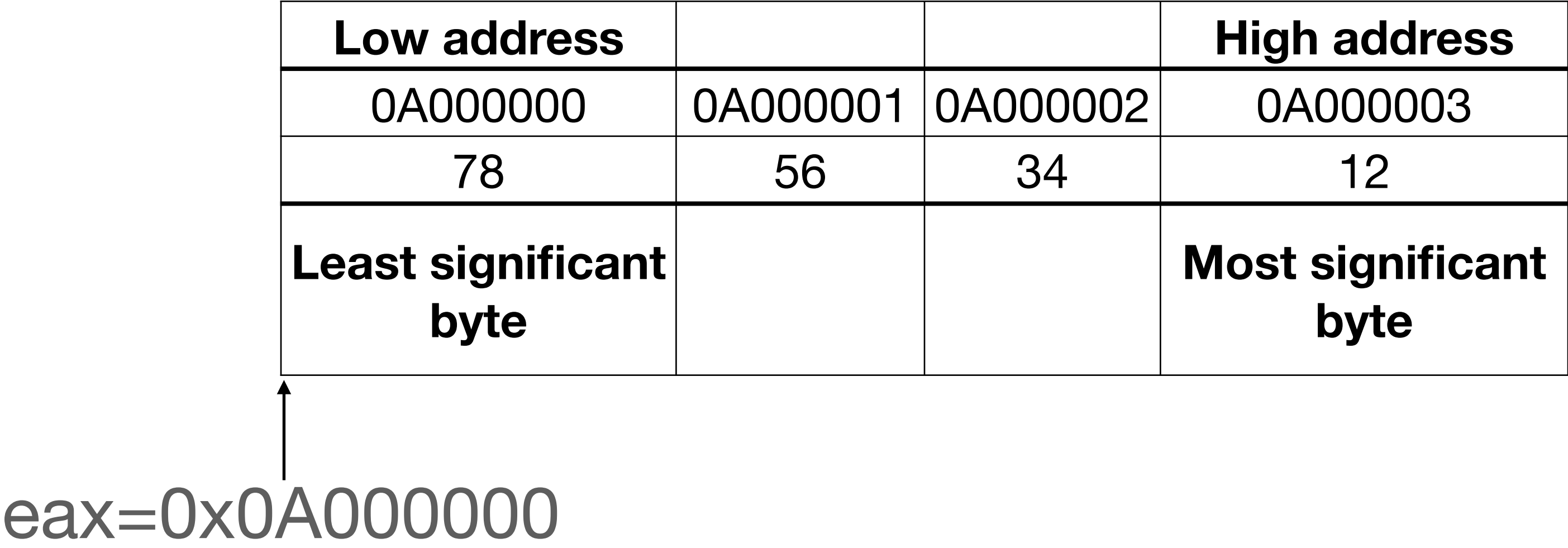
-- Begin function main

Save caller's base pointer
ebp = esp
esp = esp - 0x18
*(ebp-4)=0
*(esp) = 41
*(esp+4) = 42
Push current eip on to stack, jump to foo
esp = esp + 24 (Restore caller's esp)
Restore caller's ebp



Little endian

- Example: storing 0x12345678
- `movl (%eax), %ebx`
 - `ebx = 0x12345678`
- `movw (%eax), %bx`
 - `ebx = 0x????5678`
- `movb (%eax), %bl`
 - `ebx = 0x??????78`



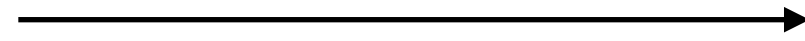
Instructions are in memory!

02.s

```
_foo:
    pushl %ebp
    movl  %esp, %ebp
    movl  8(%ebp), %eax
    addl  12(%ebp), %eax
    popl  %ebp
    retl

    .globl _main
    .p2align 4, 0x90
_main:
    pushl %ebp
    movl  %esp, %ebp
    subl  $24, %esp
    movl  $0, -4(%ebp)
    movl  $41, (%esp)
    movl  $42, 4(%esp)
    calll _foo
    addl  $24, %esp
    popl  %ebp
    retl
```

```
gcc -m32 -c 02.s -o 02.o
vim 02.o
:%!xxd
```



Instructions are in memory!

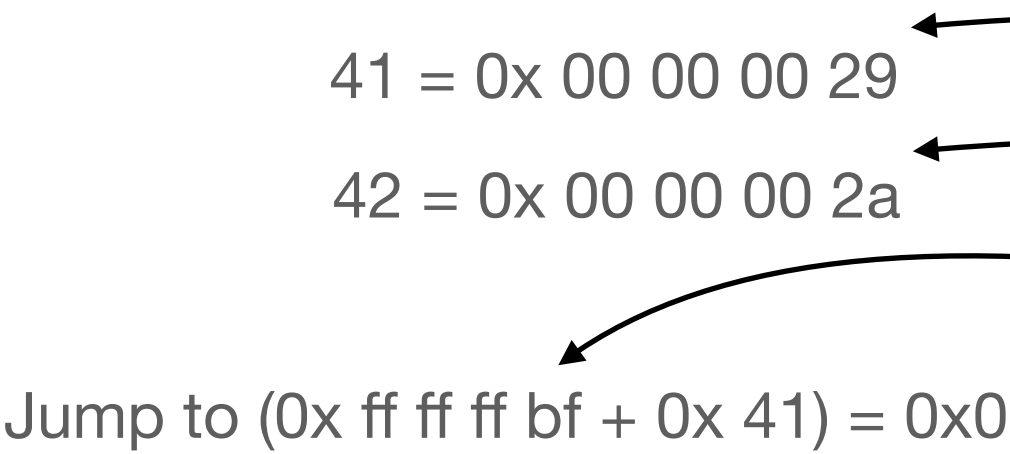
```
02.s

_foo:
    pushl %ebp
    movl  %esp, %ebp
    movl  8(%ebp), %eax
    addl  12(%ebp), %eax
    popl  %ebp
    retl

.globl _main
.p2align 4, 0x90
_main:
    pushl %ebp
    movl  %esp, %ebp
    subl  $24, %esp
    movl  $0, -4(%ebp)
    movl  $41, (%esp)
    movl  $42, 4(%esp)
    calll _foo
    addl  $24, %esp
    popl  %ebp
    retl
```

```
gcc -m32 -c 02.s -o 02.o
objdump -d 02.o > 02.dump
```

call 0x0123	pushl %eip (*) movl \$0x123, %eip (*)
ret	popl %eip (*)



00000000 <_foo>:
0: 55 pushl %ebp
1: 89 e5 movl %esp, %ebp
3: 8b 45 0c movl 12(%ebp), %eax
6: 8b 45 08 movl 8(%ebp), %eax
9: 8b 45 08 movl 8(%ebp), %eax
c: 03 45 0c addl 12(%ebp), %eax
f: 5d popl %ebp
10: c3 retl

00000020 <_main>:
20: 55 pushl %ebp
21: 89 e5 movl %esp, %ebp
23: 83 ec 18 subl \$24, %esp
26: c7 45 fc 00 00 00 00 movl \$0, -4(%ebp)
2d: c7 04 24 29 00 00 00 movl \$41, (%esp)
34: c7 44 24 04 2a 00 00 00 movl \$42, 4(%esp)
3c: e8 bf ff ff ff calll 0x0 <_foo>
41: 83 c4 18 addl \$24, %esp
44: 5d popl %ebp
45: c3 retl

* fake instructions
call saves eip of next instruction

Compiling, linking, loading

- *Preprocessor* takes C source code (ASCII text), expands #include, removes comments etc, produces C source code
- *Compiler* takes C source code (ASCII text), does compile-time optimizations, produces assembly program (also ASCII text) `.c -> .s`
- *Assembler* takes assembly program (ASCII text), produces *.o file* (binary, relocatable object file) `.s -> .o`
- *Linker* takes multiple *‘.o’s*, does link-time optimizations, produces a single *executable object a.out* (binary) `*.o -> a.out`
- *Loader* loads the program image into memory at run-time and starts executing it. `./a.out`

Compiling and linking example

- $02.yy.c \text{ — compiler } \rightarrow 02.yy.s \text{ — assembler } \rightarrow 02.yy.o$
where $yy = \text{main, func, eip, flags}$
- $02.main.o, 02.func.o, 02.eip.o, 02.flags.o \text{ — linker } \rightarrow 02.main$
- Load and run `02.main`
- A *dynamic linker* is also involved for calling `libc`'s `printf` etc. whose code is not linked in the executable. Not used in `xv6`
- When compiler compiles `02.main.c` to `02.main.s`, it may not see/have the code for `02.func.c` => Need a calling convention

gcc calling convention

at entry to a function (i.e. just after call):

- %eip points at first instruction of function
- %esp points at return address
- %esp+4 points at first argument

after ret instruction:

- %eip contains return address
- %esp points at arguments pushed by caller

called function may have trashed arguments

- %eax contains return value (or trash if function is void)
- %eax, %edx, and %ecx may be trashed (caller save)
- %ebp, %ebx, %esi, %edi must contain contents from time of call (callee save)

Function calling in action

gcc calling convention

02.s

```
_foo:
    pushl %ebp                Save caller's base pointer
    movl  %esp, %ebp         ebp = esp
    movl  8(%ebp), %eax       eax = *(ebp + 8)
    addl  12(%ebp), %eax      eax = eax + *(ebp + 12)
    popl  %ebp               Restore caller's base pointer
    retl                     change eip to return address

    .globl _main              ## -- Begin function main
    .p2align 4, 0x90
_main:
    pushl %ebp                Save caller's base pointer
    movl  %esp, %ebp         ebp = esp
    subl  $24, %esp          esp = esp - 0x18
    movl  $0, -4(%ebp)        *(ebp-4)=0
    movl  $41, (%esp)         *(esp) = 41
    movl  $42, 4(%esp)        *(esp+4) = 42
    calll _foo                Push current eip on to stack, jump to foo
    addl  $24, %esp           esp = esp + 24 (Restore caller's esp)
    popl  %ebp               Restore caller's ebp
    retl
```

- Functions save and restore callee-saved registers (ebp)
- Assume that 8(%ebp) points to the first argument, etc
- Return value is put in eax
- Can assume that the function call will return after the call instruction

Loader

- Loads program's executable file from disk (containing instructions, global variables, etc) to memory
- Jump eip into the program
- Our OS will also just be an executable which will be loaded and run by our boot loader

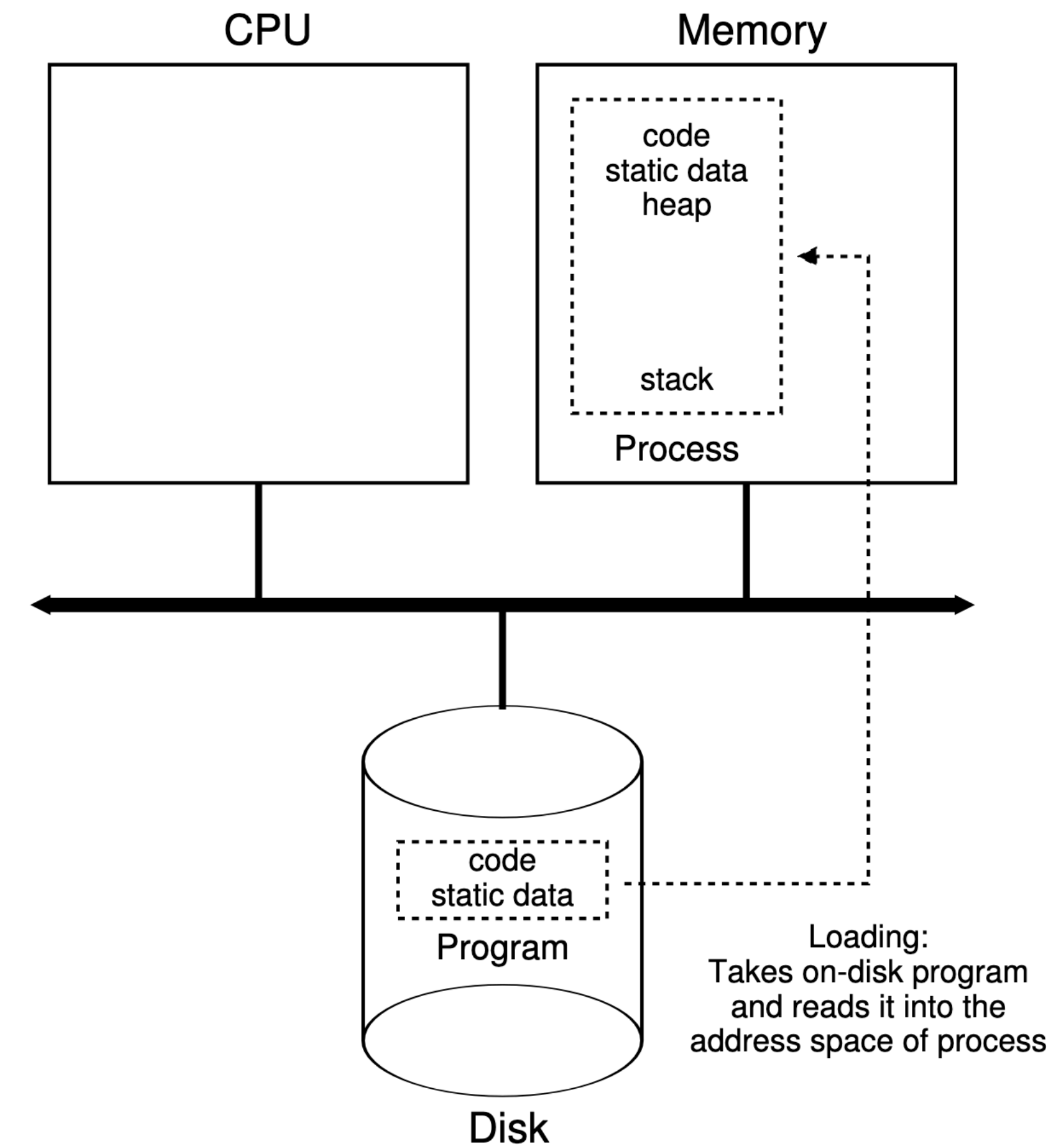


Figure 4.1: Loading: From Program To Process

Executable and Linkable Format (ELF)

- Port executables from one machine to another¹, one OS to another²

1: Same architecture

2. Same system calls

gcc v10.0

gcc v12.0

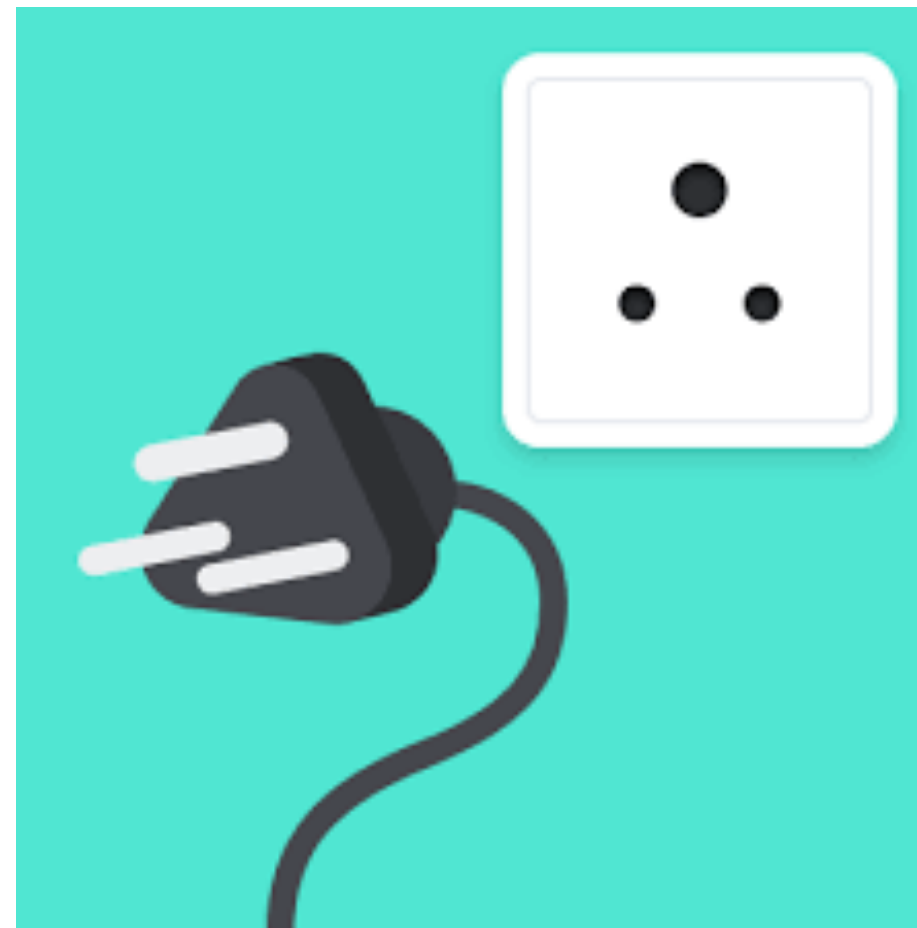
clang

ELF format

Loader

Design principle

Use interfaces, not implementation



ELF¹⁰¹ a Linux executable walk-through

ANGE ALBERTINI
CORKAMI.COM

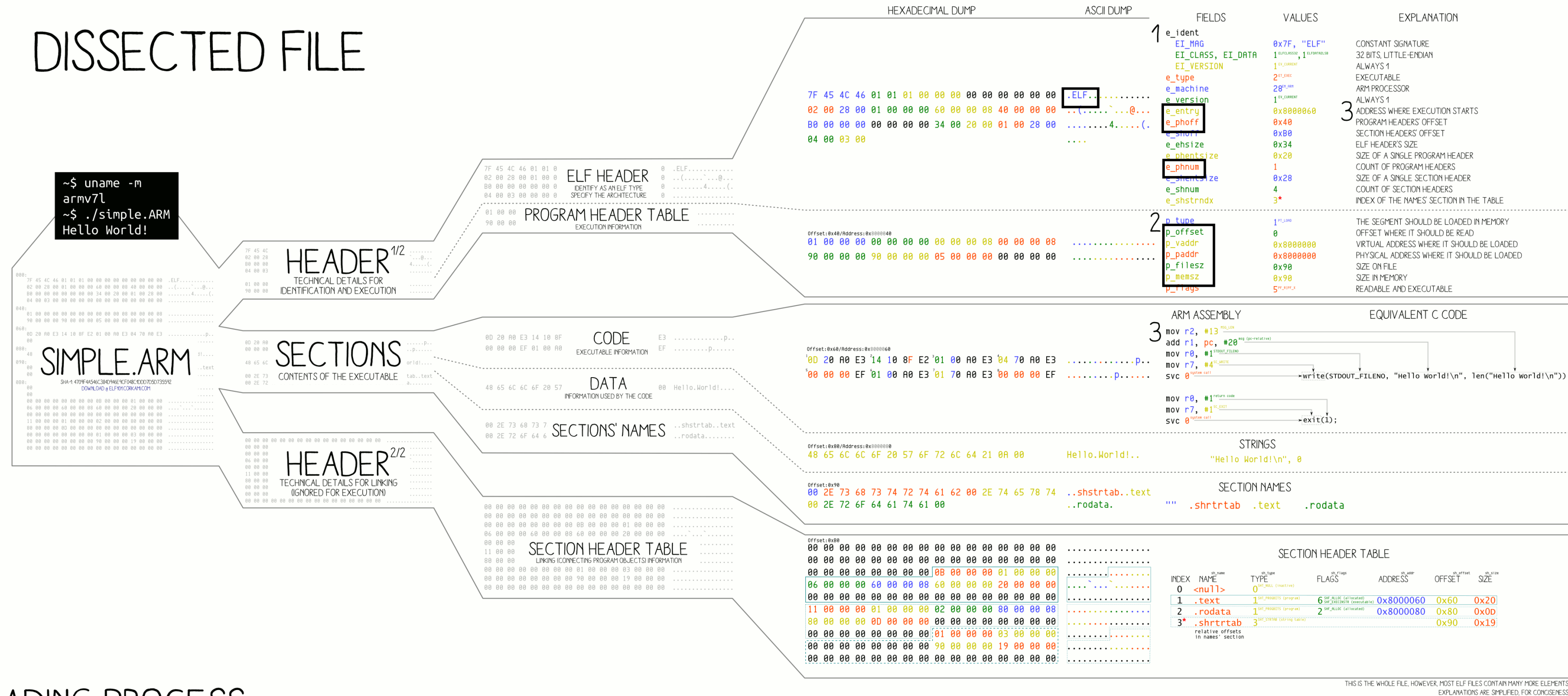


ELF

- ELF header: magic constant, program header offset, count of program headers, entry

- Program header: offset, size in file and in memory, address where to load

DISSECTED FILE



LOADING PROCESS

1 HEADER

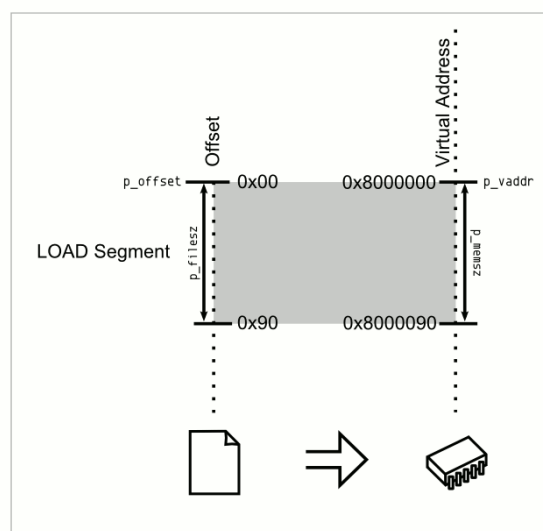
THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)

2 MAPPING

THE FILE IS MAPPED IN MEMORY
ACCORDING TO ITS SEGMENT(S)

3 EXECUTION

ENTRY IS CALLED
SYSCALLS ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE R7 REGISTER
- CALLING INSTRUCTION SVC



TRIVIA

THE ELF WAS FIRST SPECIFIED BY U.S. L. AND U.I.
FOR UNIX SYSTEM V, IN 1989

THE ELF IS USED, AMONG OTHERS, IN:

- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

ELF in action

Elf headers

```
$ readelf -l 02.main
Elf file type is EXEC (Executable file)
Entry point 0x8049050
There are 11 program headers, starting at offset 52
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  ...
  LOAD           0x001000 0x08049000 0x08049000 0x00284 0x00284 R E 0x1000
  LOAD           0x002000 0x0804a000 0x0804a000 0x00120 0x00120 R    0x1000

Section to Segment mapping:
Segment Sections...
...
03      .init .plt .text .fini
04      .rodata .eh_frame_hdr .eh_frame
```

After loading, jump EIP to 0x8049050

Copy 0x284 bytes from 0x1000 offset in the file and copy it to virtual address 0x08049000.

This section is readable and executable

Data section is not marked as executable

ELF in action

ELF file

```
$ vim 02.main (:%!xxd)
000000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF.....      Magic header
000000ff0: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00001000: f30f 1efb 5383 ec08 e893 0000 0081 c3f3  ....S.....
00001010: 2f00 008b 83fc ffff ff85 c074 02ff d083  /. ....t....
00001020: c408 5bc3 0000 0000 0000 0000 0000 0000  .. [.....
00001280: c408 5bc3 0000 0000 0000 0000 0000 0000  .. [ .....
00001290: 0000 0000 0000 0000 0000 0000 0000 0000  .....
```

0x284 bytes of Executable section starting at 0x1000

ELF in action

Disassembly

```
$ vim 02.main.dump
```

```
Disassembly of section .init:
```

```
08049000 <_init>:
```

```
8049000: f3 0f 1e fb
```

```
8049004: 53
```

```
8049005: 83 ec 08
```

```
endbr32
```

```
push    %ebx
```

```
sub     $0x8,%esp
```

Matches the executable
section of ELF as
expected

```
Disassembly of section .text:
```

```
08049050 <_start>:
```

```
8049050: f3 0f 1e fb
```

```
8049054: 31 ed
```

```
8049056: 5e
```

```
endbr32
```

```
xor     %ebp,%ebp
```

```
pop     %esi
```

ELF header marked this
as entry point

Revisit concurrency/volatile

Each thread has its own registers. Memory is common.

- ./threads 100000
- threads.c
- threads.s, threads.pseudo.c

```
while ...                # eax != loops
    movl counter %eax    # eax = counter
    addl $1 %eax         # eax ++
    movl %eax counter    # counter = eax
```

Thread 1	Thread 2
Read counter = 0	
...	
Write counter = 100	
	Read counter = 100
	..
Read counter = 199	
..	
Writer counter = 300	
	Write counter = 200
	Read counter = 200
	...

Revisit concurrency/volatile (2)

Each thread has its own registers. Memory is common.

- ./threads 10
- threads.c
- threads.s, threads.pseudo.c

```
while ...                # eax != loops
    movl counter %eax    # eax = counter
    addl $1 %eax         # eax ++
    movl %eax counter    # counter = eax
```

Thread 1	Thread 2
Read counter = 0	
Write counter = 1	
Read counter = 1	
...	
Writer counter = 10	
	Read counter = 10
	Writer counter = 11
	Read counter = 11
	Writer counter = 12
	...

Revisit concurrency/volatile (3)

Each thread has its own registers. Memory is common.

- ./threads-notv 100000
- threads-notv.c
- threads-notv.s, threads-notv.pseudo.c

```
movl counter %ecx      # ecx = counter
movl loops %edx        # edx = loops
while ..               # eax != edx
    addl $1 %eax        # eax ++
addl %ecx %edx         # edx += ecx
movl %edx, counter     # counter = edx
```

Thread 1	Thread 2
Read counter = 0	
....	
	Read counter = 0

....	

Writer counter = 100000	

	Writer counter = 100000

Revisit concurrency/volatile (4)

Each thread has its own registers. Memory is common.

- ./threads-notv-O3 100000
- threads-notv-O3.c
- threads-notv.O3.s,
threads-notv.pseudo.O3.c

```
movl loops %eax      # eax = loops
addl %eax, counter    # counter += eax
```

Thread 1	Thread 2
Read counter = 0	
Writer counter = 100000	
	Read counter = 100000
	Writer counter = 200000

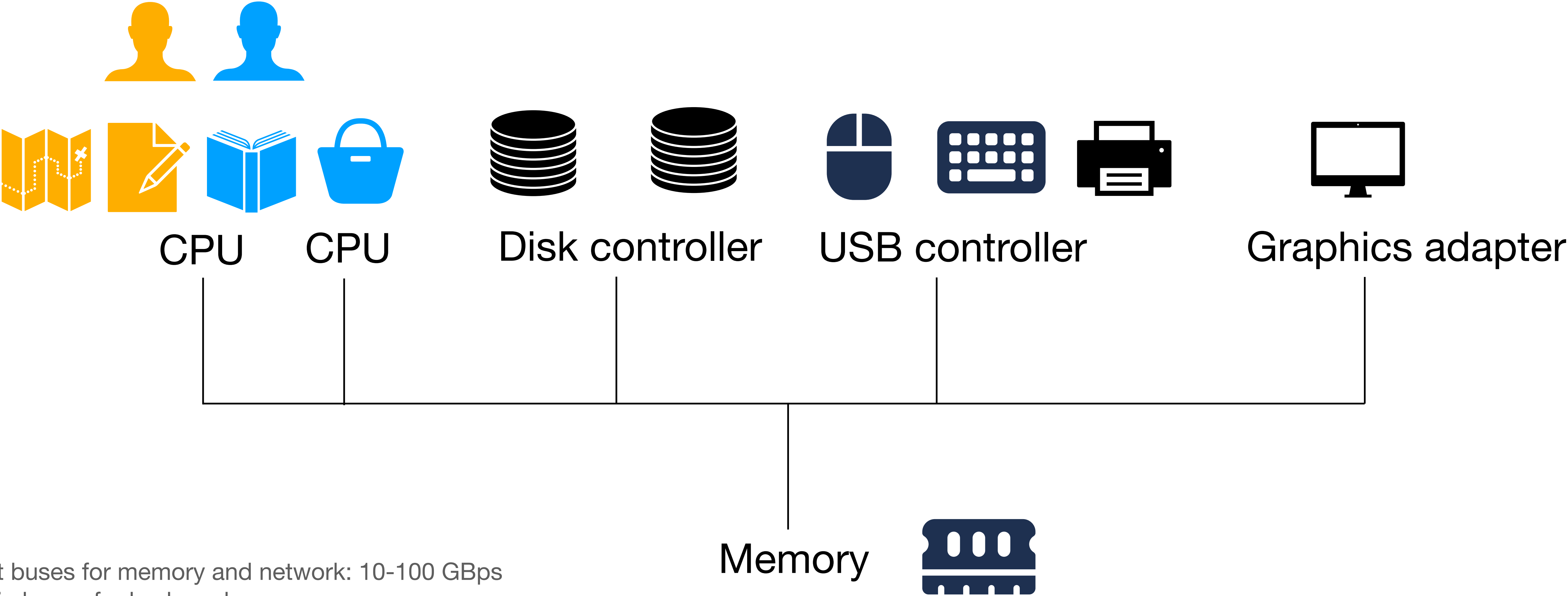
Memory access hierarchy: caches

- Registers are limited in size.
- Main memory is slow.
- Recently accessed data lives on on-chip caches.
- Mostly transparent to OS

Intel Core i7 Xeon 5500 at 2.4 GHz		
Memory	Access time	Size
register	1 cycle	64 bytes
L1 cache	~4 cycles	64 kilobytes
L2 cache	~10 cycles	4 megabytes
L3 cache	~40-75 cycles	8 megabytes
remote L3	~100-300 cycles	
Local DRAM	~60 nsec	
Remote DRAM	~100 nsec	

Figure A-1. Latency numbers for an Intel i7 Xeon system, based on http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

Computer organization



Fat buses for memory and network: 10-100 GBps
Thin buses for keyboard, mouse

I/O devices

Port-mapped IO

- Similar to reading from (writing to) memory locations
- Special instructions:
 - inb (outb) reads (writes) a byte to port
- Only 1024 ports

Writing a byte to line printer

```
#define DATA_PORT    0x378
#define STATUS_PORT   0x379
#define CONTROL_PORT  0x37A
#define    BUSY 0x80
#define    STROBE 0x01
void
lpt_putc(char c)
{
    /* wait for printer to consume previous byte */
    while((inb(STATUS_PORT) & BUSY));

    /* put the byte on the data lines */
    outb(DATA_PORT, c);

    /* tell the printer to look at the data */
    outb(CONTROL_PORT, STROBE);
    outb(CONTROL_PORT, 0);
}
```


I/O devices

Memory-mapped IO

- Regular memory access instructions
- Reads and writes are routed to appropriate device
 - Writes to VGA memory appear on the screen
- Power-on jumps %eip to 0x000F000
- Careful! Does not behave like memory!
 - Reading same location twice can change due to external events (declare volatile)
- Example: `sudo cat /proc/iomem`

