

Lecture 1

Introduction to System Programming and Machine Structures

Md Mahbub E Noor

Assistant Professor, Department of CSE
University of Barishal

```
● ● ● bash – lab PC
```

```
$ gcc hello.c -o hello
```

```
$ ./hello
```

```
Hello, System Programming!
```

```
# What happened between  
# typing these two lines and  
# seeing that output?  
# That is this course.
```

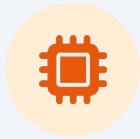
What you will learn today



Tell system software apart from application software, and say where it sits in a computer.



Trace how hello.c becomes a running process: preprocess, compile, assemble, link, load.



Name the parts of a CPU and follow the fetch-decode-execute cycle one step at a time.



Explain user mode and kernel mode, and watch real system calls with strace.

Today's plan (60 min)

0-10

What is system programming?

10-25

The journey of a program

25-45

Machine structures

45-55

User mode, kernel mode

55-60

Checkpoint and homework

Lab 1 (setup + six tasks) runs in the lab period.

Course map: three parts, one story

We follow one program, `hello.c`, from the text you type all the way down to the hardware. Each part of the course explains one stage of that trip.



PART A

Foundations and translation

Lectures 1–6

How source code becomes an executable file: machine structure, assembly language, assemblers, macro processors, linkers, loaders and the ELF file format.



PART B

The Linux kernel

Lectures 7–13

What happens when the program runs: booting, processes, system calls, inter-process communication, signals, memory, the file system and interrupts.



PART C

Modules and device drivers

Lectures 14–15

How we extend the kernel ourselves: loadable modules, the device driver model, and character, block and USB drivers.

`hello.c`

executable

process

kernel

hardware

How each class works

1 Concept

A short explanation on the slides, with one picture.

2 Demo

I run it live in a Linux terminal. You watch the real output.

3 Lab task

You run the same commands and fill in a table with your own results.

4 Homework

You change or extend the program, and explain what changed.



Lab environment

Ubuntu Linux on the lab PCs with `gcc`, `binutils` (as, ld, objdump, readelf, nm), `gdb`, `strace` and `nasm`.

From the kernel lectures on (Part B), risky work runs inside a **virtual machine**, so a crash cannot break the lab PC.



Using AI tools

You may use an AI tool to help you **understand** an error message or a man page.

But every line you submit, you must be able to **explain in the viva**. If you cannot explain it, it does not count.

PART 1

What is system programming?

Two kinds of software, and where each one sits inside a computer.

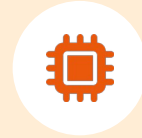
Application software vs system software



Application software

Solves **the user's** problem. The user knows it exists and talks to it directly.

Examples: Web browser, MS Word, a game, the spam detector you built in Python.



System software

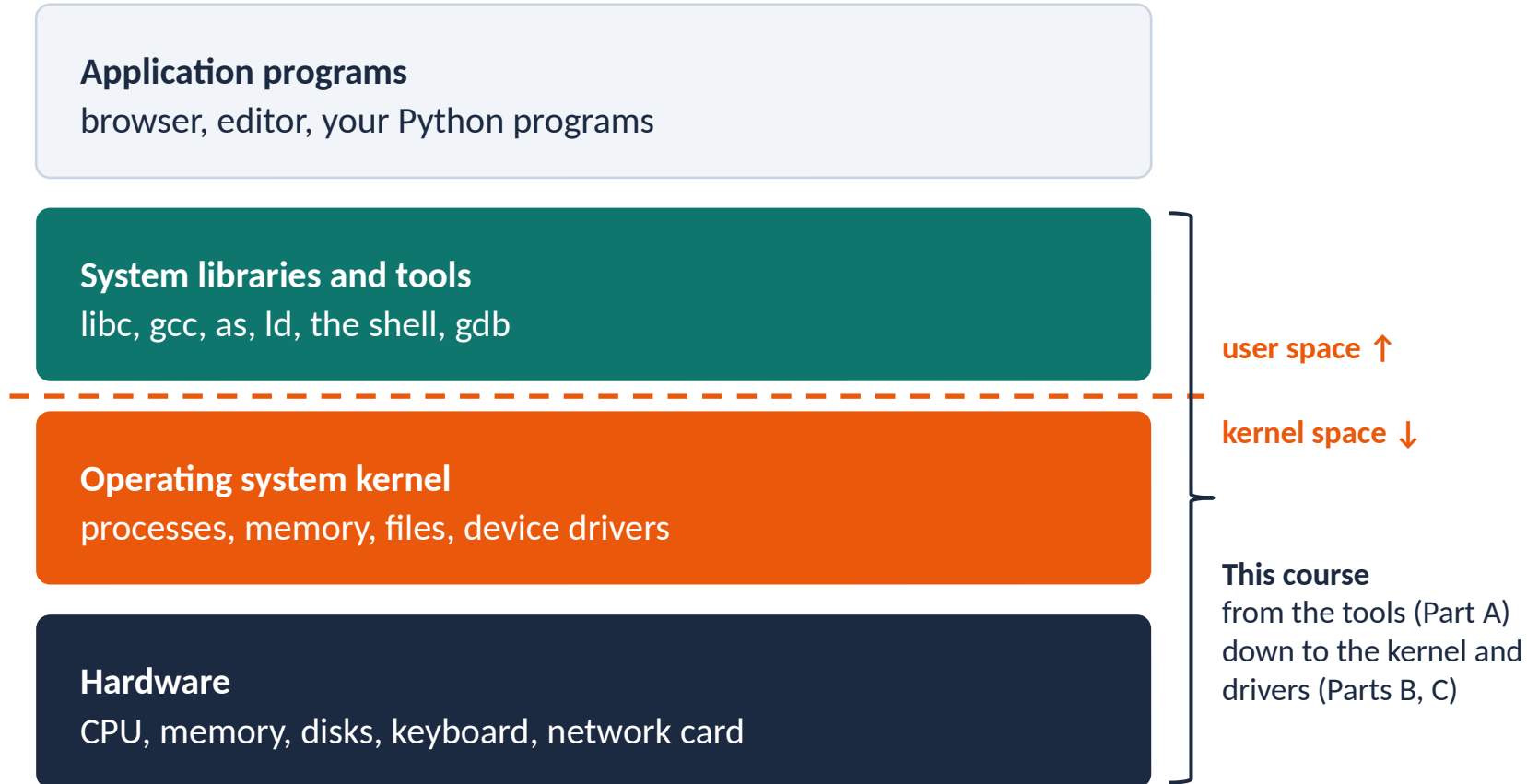
Supports **the operation of the computer itself**. Other programs run on top of it. Most users never notice it.

Examples: Compiler, assembler, linker, loader, operating system kernel, device drivers.

Key idea. System software depends on the machine it runs on. An assembler must know the CPU's **instruction formats and addressing modes**; a loader must know how **memory addresses** work. Some features do not depend on the machine (for example, how an assembler stores its symbol table). We will label every feature as **machine-dependent** or **machine-independent**.

Book: Beck, System Software, §1.1–1.2

Where system software lives: the layered view



Each layer only talks to the layer **below it**. Your program never touches the disk itself. It calls a library function, the library asks the kernel, and the kernel uses a driver to talk to the disk.

The boundary between **user space** and **kernel space** is guarded by the CPU itself. Crossing it needs a special instruction, a **system call** (Lecture 9).

Book: Bryant & O'Hallaron, CS:APP Fig. 1.10; Love, LKD Fig. 1.1

System software you already use without noticing



Compiler

Translates C into assembly language.

today



Assembler

Turns assembly mnemonics into machine-code bytes.

Lecture 3



Macro processor

Expands short macros into full code, like C's #define.

Lecture 4



Linker

Joins object files and libraries into one executable.

Lecture 5



Loader

Copies the executable into memory and starts it.

Lectures 5, 8



OS kernel

Shares the CPU, memory and files between programs.

Lectures 7-13



Device driver

Lets the kernel talk to one kind of hardware.

Lectures 14-15



Debugger

Runs a program step by step so you can inspect it.

every lab (gdb)

PART 2

The journey of a program

What gcc really does when you type `gcc hello.c -o hello`.

From hello.c to a running process: five steps

```
hello.c
#include <stdio.h>

int main(void)
{
    printf("Hello, ...\n");
    return 0;
}
```

gcc is not one program. It is a driver that runs four other programs in turn. The kernel then runs the fifth step when you type `./hello`.

- | | | | | |
|---|--------------------------|------------------------|---|--------|
| 1 | Preprocess
cpp | hello.c → hello.i | Pastes in the #include files and expands macros. Our 7 lines become 820. | L4 |
| 2 | Compile
cc1 | hello.i → hello.s | Translates C into assembly language for this CPU (x86-64). | — |
| 3 | Assemble
as | hello.s → hello.o | Turns each mnemonic into machine-code bytes. Leaves holes for addresses it does not know yet. | L3 |
| 4 | Link
ld | hello.o + libc → hello | Fills the holes, attaches the C library, writes an executable ELF file. | L5, L6 |
| 5 | Load
kernel | hello → process | Creates a process, maps the file into memory, jumps to its first instruction. | L5, L8 |

Book: Bryant & O'Hallaron, CS:APP Fig. 1.3 (the compilation system)

Demo: stop gcc after every step

```
bash — ~/lab1
$ gcc -E hello.c -o hello.i # stop after preprocessing
$ gcc -S hello.c -o hello.s # stop after compiling
$ gcc -c hello.s -o hello.o # stop after assembling
$ gcc hello.o -o hello      # link
$ ./hello
Hello, System Programming!
$ wc -l hello.c hello.i hello.s
$ ls -l hello* ; file hello*
```

File	Lines	Bytes	What the file command says
hello.c	7	97	C source, ASCII text
hello.i	820	21,329	C source, ASCII text
hello.s	45	681	assembler source, ASCII text
hello.o	—	1,512	ELF 64-bit LSB relocatable, x86-64
hello	—	15,960	ELF 64-bit LSB pie executable, x86-64, dynamically linked

Look closely. hello.i is **120 times bigger** than hello.c: that is stdio.h pasted in. The last two files are both **ELF** files, but only **hello** can run. **hello.o** is "relocatable": its addresses are not final yet.

Question: what exactly is missing from hello.o? (Next two slides, and Lectures 5–6.)

Inside hello.s: your first look at assembly

hello.s (main only, directives removed)

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    call    puts@PLT
    movl    $0, %eax
    popq    %rbp
    ret
```

pushq / movq

Set up a **stack frame** for main, so it has its own space on the stack.

leaq .LC0(%rip), %rax

Put the **address of the string** "Hello, ..." into register rax.

movq %rax, %rdi

Copy it to **rdi**: on x86-64 Linux the first argument always travels in rdi.

call puts@PLT

Jump to the library function **puts**, through the PLT (Lecture 5.2).

movl \$0, %eax

The return value goes in **eax**: this is **return 0**;

Puzzle. We wrote **printf**, but the compiler called **puts**. Why is that allowed? (Hint: look at what our string ends with, and whether it has any % signs.)

Book: CS:APP §3.2–3.4 (x86-64 registers and calling convention)

Inside hello.o: machine code with holes in it

```
objdump -d hello.o
```

```
0000000000000000 <main>:
```

```
 0: f3 0f 1e fa      endbr64
 4: 55              push    %rbp
 5: 48 89 e5        mov     %rsp,%rbp
 8: 48 8d 05 00 00 00 00 lea     0x0(%rip),%rax
f: 48 89 c7        mov     %rax,%rdi
12: e8 00 00 00 00  call    17 <main+0x17>
17: b8 00 00 00 00  mov     $0x0,%eax
1c: 5d              pop     %rbp
1d: c3              ret
```

```
nm hello.o
```

```
0000000000000000 T main
                 U puts
```

Read a line

Left: the **offset** inside main. Middle: the **machine-code bytes**. Right: the same instruction in assembly.

Find the holes

The orange lines end in **00 00 00 00**. The assembler did not know the address of the string or of puts, so it wrote zeros and left a note, a **relocation entry**, for the linker.

Read nm

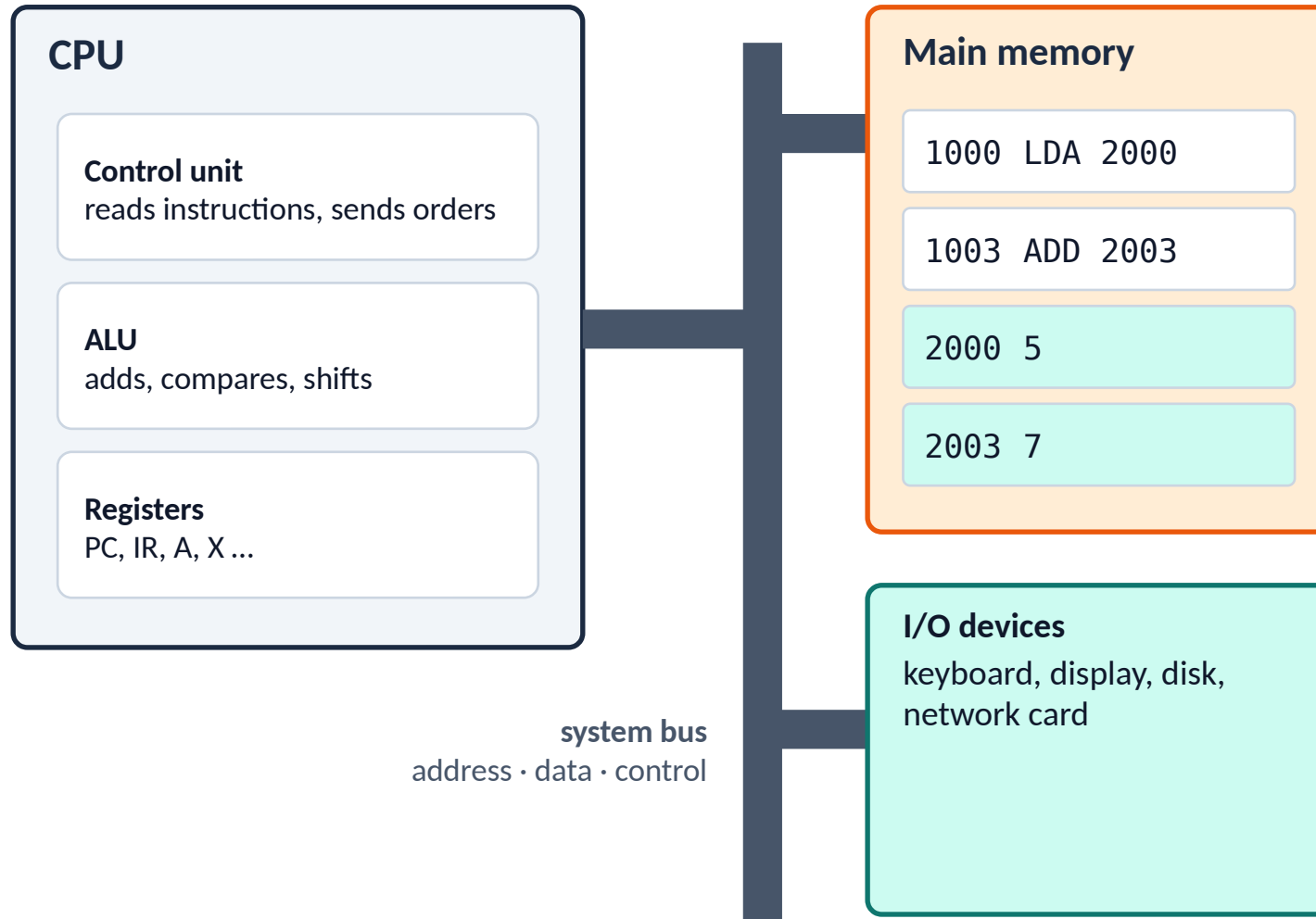
T main: main is defined here, in the code ("text") section. **U puts**: puts is used but **undefined** here. The linker must find it in libc.

PART 3

Machine structures

What is inside the box that runs our program: CPU, registers, memory and the cycle that drives them.

Inside a computer: the stored-program machine



Stored-program idea. Instructions and data sit in the **same memory**, both as plain numbers. Look at memory 1000: the number there is an instruction only because the CPU **fetches** it as one.

The CPU never touches a device directly by itself. Everything travels over the **bus**: the CPU puts an address on it, and memory or a device answers.

This design is called the **von Neumann architecture**. Nearly every computer today, from a phone to a server, follows it.

Book: CS:APP Fig. 1.4; Beck §1.3 (SIC machine architecture)

Registers: the CPU's own tiny, fast memory

A register holds one word inside the CPU itself. Every CPU gives its registers the same jobs; only the names change. We use **SIC** for the assembler theory on paper (Lectures 2–5) and **x86-64** for the real labs.

SIC and SIC/XE (Beck)

Name	No.	Job
A	0	Accumulator: arithmetic results
X	1	Index: stepping through arrays
L	2	Linkage: return address of a JSUB call
B	3	Base: base-relative addressing (XE)
S, T	4, 5	General working registers (XE)
F	6	Floating point, 48 bits (XE)
PC	8	Program counter: next instruction
SW	9	Status word: condition code

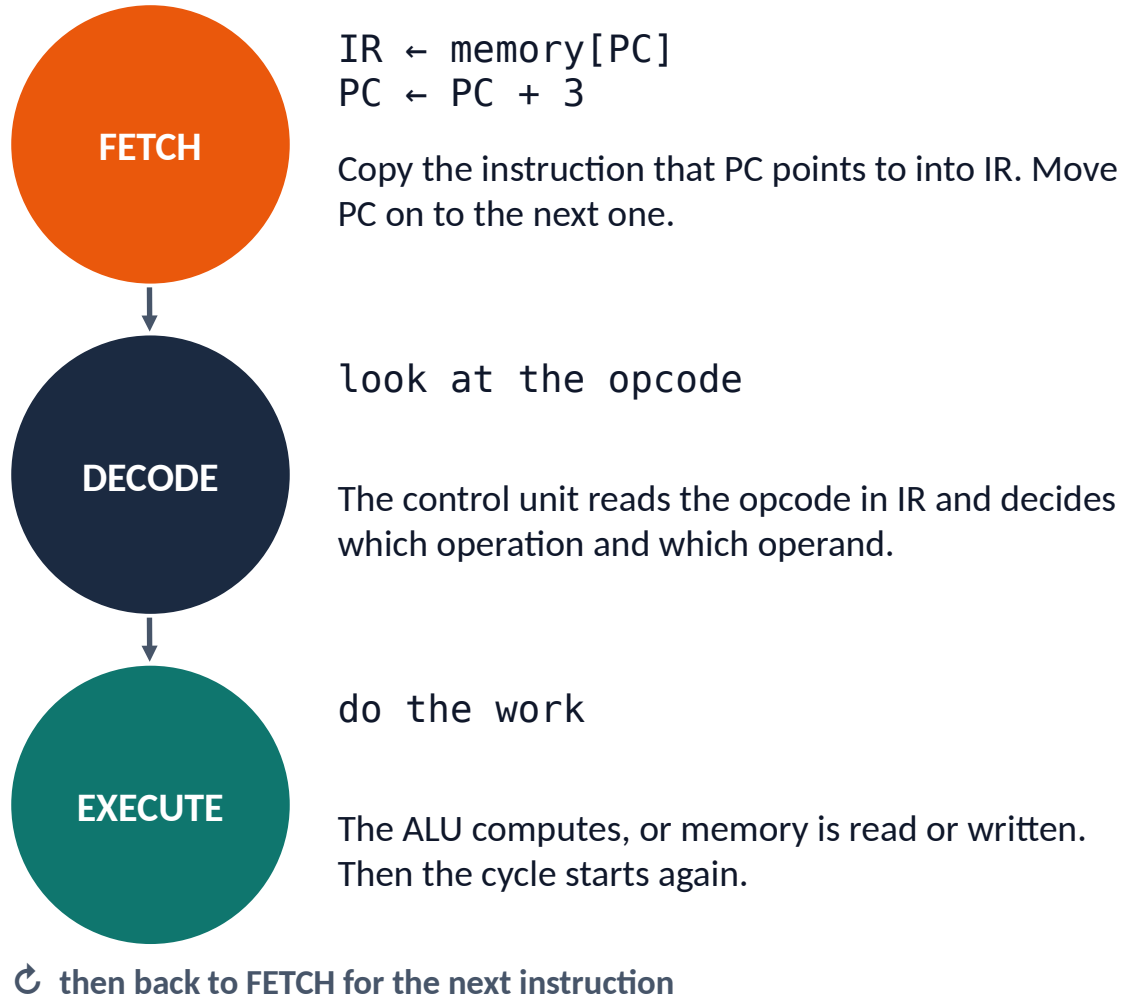
Orange rows: the program counter and the status register. Both machines have them, under different names.

x86-64 (your lab PC)

Name	Job
rax	Return value of a function
rdi, rsi, rdx	Arguments 1, 2, 3
rcx, r8, r9	Arguments 4, 5, 6
rsp	Stack pointer: top of the stack
rbp	Base (frame) pointer
rip	Instruction pointer: same job as PC
rflags	Flags: same job as SW
rbx, r10–r15	General purpose

Book: Beck §1.3.1–1.3.2; CS:APP §3.4

The fetch-decode-execute cycle



● ● ● a tiny SIC program

addr	instruction	meaning
1000	LDA ALPHA	$A \leftarrow \text{ALPHA}$
1003	ADD BETA	$A \leftarrow A + \text{BETA}$
1006	STA GAMMA	$\text{GAMMA} \leftarrow A$
...	data ...	
2000	ALPHA WORD 5	
2003	BETA WORD 7	
2006	GAMMA RESW 1	

Why PC + 3? Every SIC instruction is exactly **3 bytes**: 1 byte opcode, 2 bytes for flags and address. So the next instruction always starts 3 bytes later: 1000, 1003, 1006.

(x86-64 instructions are 1 to 15 bytes, so its fetch step must decode the length first.)

Book: Beck §1.3.1 (SIC instruction format); CS:APP §1.4.1

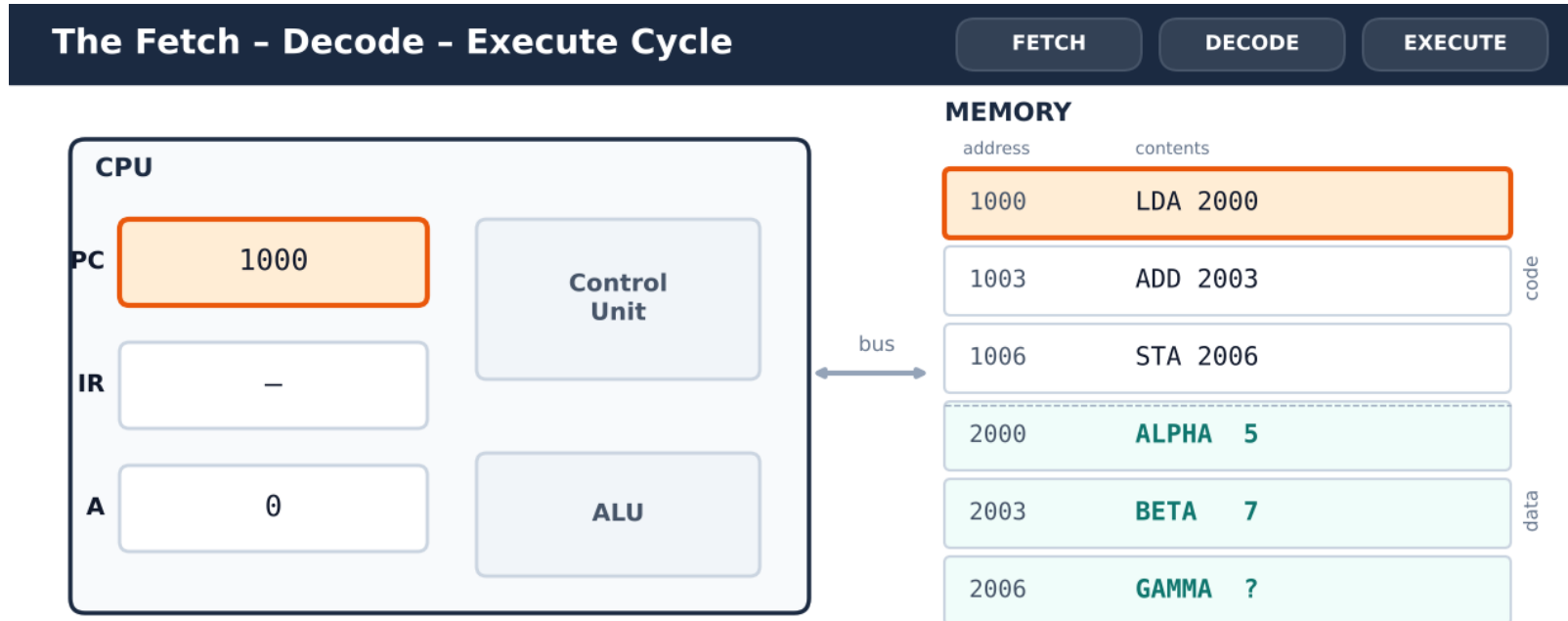
Trace it step by step

Predict each row before it appears. Values after each step. Addresses in hex.

Step	Phase	PC	IR	A	GAMMA	What happened
0	start	1000	—	0	?	Program loaded. PC points at the first instruction.
1	fetch	1003	LDA 2000	0	?	IR ← memory[1000]; PC moves on by 3.
2	decode	1003	LDA 2000	0	?	Opcode LDA: load A from address 2000.
3	execute	1003	LDA 2000	5	?	A ← memory[2000] = 5.
4	fetch	1006	ADD 2003	5	?	IR ← memory[1003]; PC moves on by 3.
5	decode	1006	ADD 2003	5	?	Opcode ADD: add memory[2003] to A.
6	execute	1006	ADD 2003	12	?	ALU: 5 + 7 = 12, result in A.
7	fetch	1009	STA 2006	12	?	IR ← memory[1006]; PC moves on by 3.
8	decode	1009	STA 2006	12	?	Opcode STA: store A at address 2006.
9	execute	1009	STA 2006	12	12	memory[2006] ← A. GAMMA is now 12.

Orange = the value that changed in this step.

Watch the cycle run



A tiny program sits in memory. Goal: $GAMMA = ALPHA + BETA$. The PC (program counter) holds 1000, the address of the first instruction.

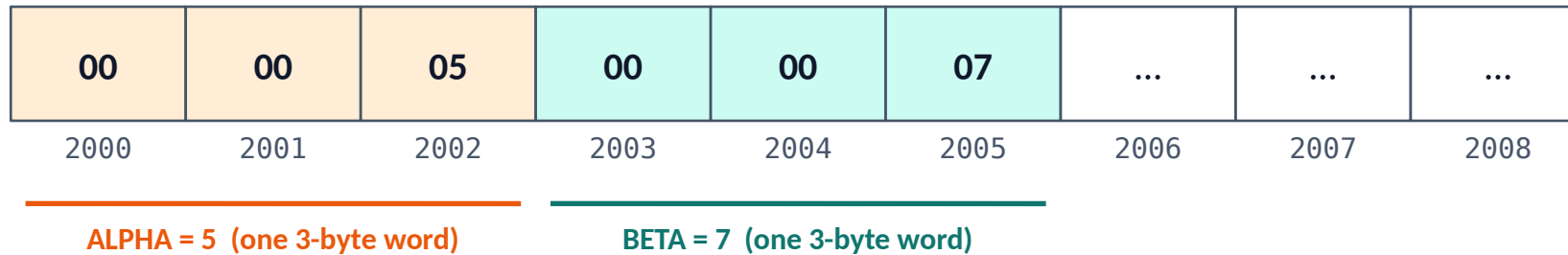
Click the video to play or pause. 55 seconds, no sound.

Watch for

1. The **phase light** at the top right: FETCH, DECODE, EXECUTE.
2. PC jumps by 3 at every fetch.
3. IR changes only during FETCH.
4. Only EXECUTE touches the **data** area of memory.
5. After STA, GAMMA becomes 12, but A keeps 12 too.

Memory: bytes, words and addresses

Memory is one long row of **bytes**. Each byte has a number, its **address**. A **word** is the group of bytes the CPU moves in one go.



```
● ● ● sizes.c → ./sizes
```

```
# sizeof each C type
char   : 1 byte
short  : 2 bytes
int     : 4 bytes
long   : 8 bytes
pointer: 8 bytes
```

```
# 8-byte pointer =
# 64-bit addresses
```

SIC

word = **3 bytes** (24 bits); memory = **32,768 bytes** (2^{15}), so a 15-bit address is enough.

SIC/XE

same 3-byte word; memory up to **1 MB** (2^{20}), so addresses need 20 bits (format 4).

x86-64

every byte has its own address; **pointers are 8 bytes**, as the program on the right shows.

Book: Beck §1.3.1–1.3.2; CS:APP §2.1

Byte order: which byte comes first?

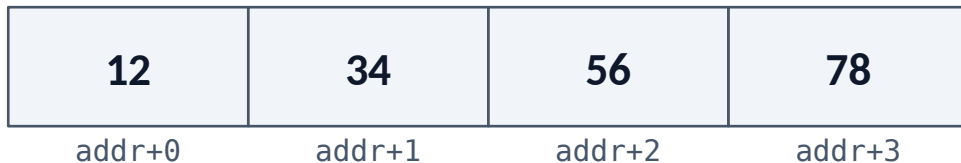
A 4-byte integer `0x12345678` needs four memory cells. CPUs disagree about the order in which to store them.

Little-endian · x86-64, most ARM phones



lowest address holds the **least** significant byte

Big-endian · network byte order, many old CPUs



lowest address holds the **most** significant byte

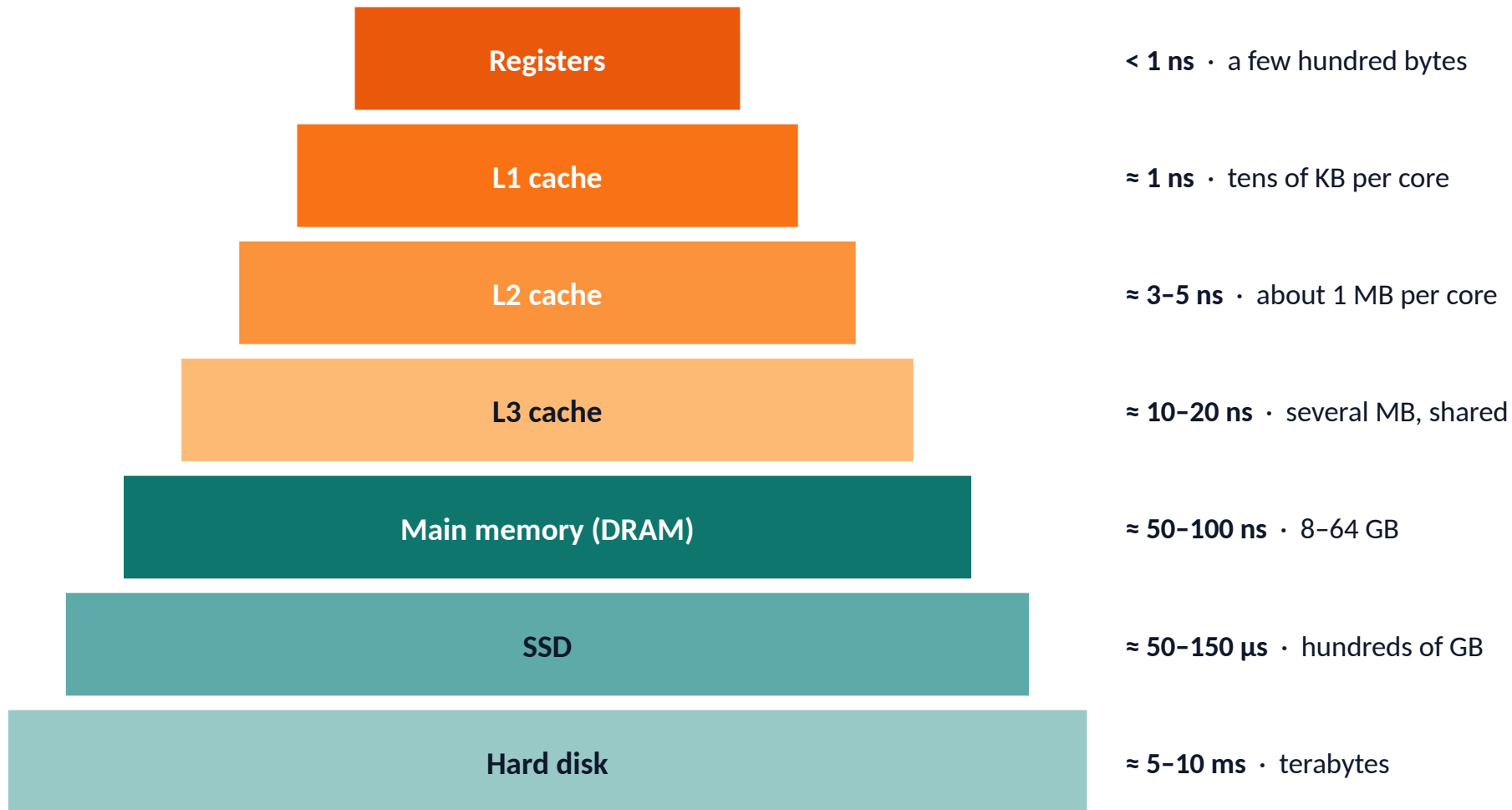
```
endian.c → ./endian
Value of x      : 0x12345678
Size of x       : 4 bytes
Address ...988  : 78
Address ...989  : 56
Address ...98a  : 34
Address ...98b  : 12
```

Predict first: our lab PC is x86-64. Which byte will the program print first? Then click to see the real output.

Why care? You will read raw bytes in `objdump`, `hexdump` and ELF files (Lecture 6). Read them backwards on x86-64.

Book: CS:APP §2.1.3 (addressing and byte ordering)

The memory hierarchy: fast and small at the top



Going up: faster, smaller, more expensive per byte.

access time · typical size

Human scale. Stretch time so that **1 ns = 1 second**:

L1 hit: **1 second**

DRAM read: **1½ minutes**

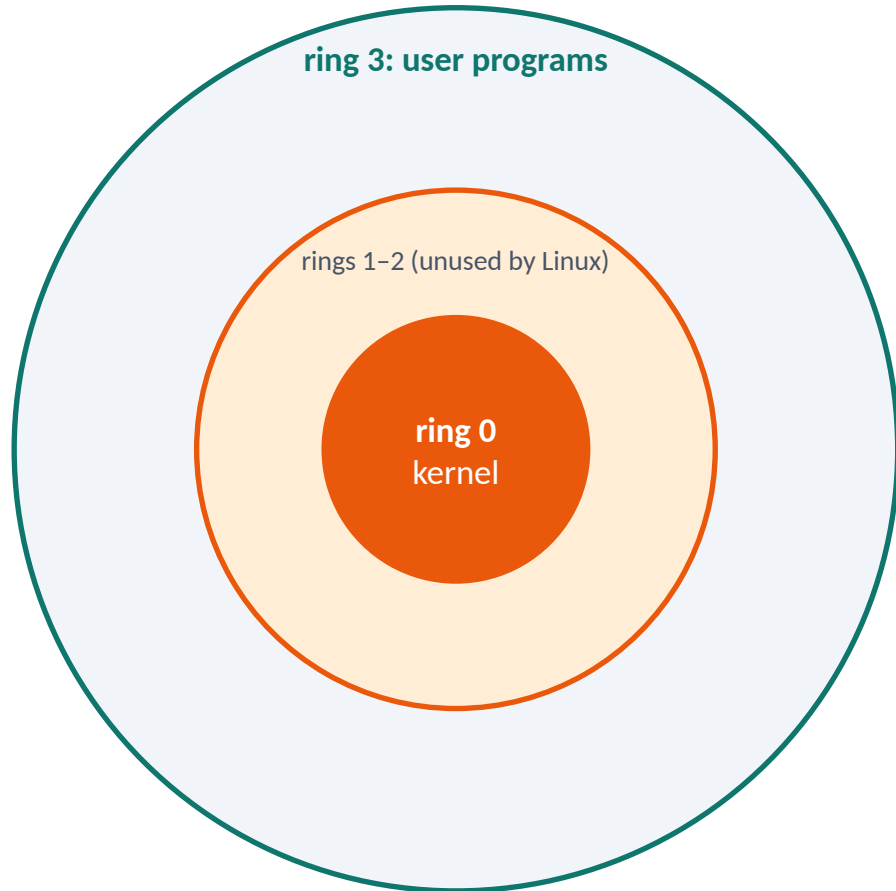
SSD read: **about a day**

Hard-disk read: **about 3 months**

This is why the kernel caches files in RAM (Lecture 12).

Book: CS:APP Fig. 1.9 (an example memory hierarchy)

Two modes of the CPU: user mode and kernel mode



	User mode	Kernel mode
Who runs here	your programs, libc, the shell	the Linux kernel, drivers
Privileged instructions (hlt, cli, load page table)	not allowed	allowed
Memory it can reach	only its own	all of it
Talk to devices directly	no, must ask the kernel	yes
If it crashes	only that program dies	the whole machine can stop

The mode is a bit inside the CPU, not a setting in software. The CPU checks it **before every privileged instruction**. x86 has four rings; Linux uses only ring 0 and ring 3.

Book: Bovet & Cesati, UTLK Ch. 1; Love, LKD Ch. 1

Demo: a user program tries a kernel-only instruction

```
privileged.c
#include <stdio.h>

int main(void)
{
    printf("User mode: about to run ...\n");
    fflush(stdout);
    __asm__ volatile ("hlt");
    printf("This line is never printed.\n");
    return 0;
}
```

```
bash
$ gcc privileged.c -o privileged && ./privileged
User mode: about to run a privileged instruction...
Segmentation fault
$ echo $?
139
```

1. hlt in ring 3

The program asks the CPU to halt. The CPU sees it is in user mode.



2. CPU raises a fault

A General Protection exception. The CPU switches to kernel mode and jumps to the kernel's handler.



3. Kernel decides

The handler sees a user program broke the rules and sends it the signal SIGSEGV (signal 11).



4. Program is killed

The shell reports exit status 139 = 128 + 11. The rest of the machine keeps running.

System calls: the legal doorway into the kernel

```
strace ./hello (shortened)

execve("./hello", ["./hello"], ...) = 0
brk(NULL) = 0x55d3b96a2000
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", ...) = 3
read(3, "\177ELF\2\1\1\3...", 832) = 832
mmap(NULL, 2174352, PROT_READ, ...) = 0x7f0646000000
... 25 more calls to set up the process ...
write(1, "Hello, System Programming!\n", 27) = 27
exit_group(0) = ?
```

34

system calls
(normal build)

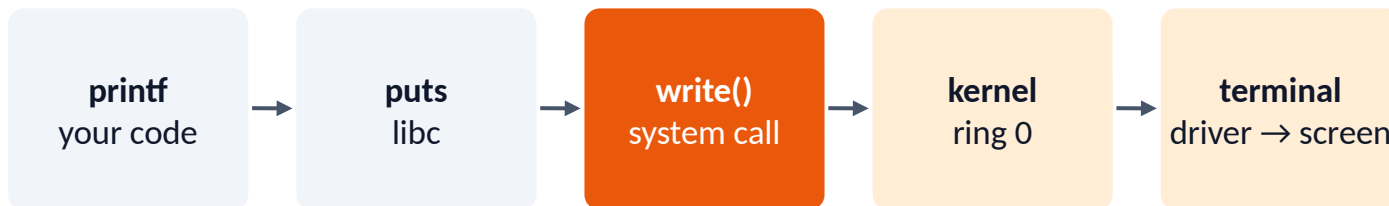
16

system calls
(gcc -static)

A system call is a **deliberate request** from a user program for something only the kernel may do: open a file, write to the screen, create a process.

Even "hello world" crosses into the kernel **34 times**. Most of those calls load the shared C library. A static build carries libc inside the file, so it needs only 16.

The path of our printf, from your code to the screen:



Book: Love, LKD Ch. 5; Kerrisk, TLPI Ch. 3

Three ways the CPU enters kernel mode



System call

The program **asks** on purpose.

Examples: write, open, fork, execve

Lecture 9



Exception

The program does something the CPU **cannot finish alone**.

Examples: divide by zero, page fault, the hlt we just tried

Lectures 11, 13



Interrupt

A **hardware device** needs attention, at any moment.

Examples: key pressed, disk finished, timer tick

Lecture 13

The same four steps every time: the CPU (1) switches to kernel mode, (2) saves where the program was, (3) runs a kernel handler, and (4) returns to the program, or kills it, as we saw with hlt.

LAB 1

Tools, the gcc pipeline and your machine

Run everything you saw today on your own lab PC and record your own numbers.

Lab 1 setup: check your tools

```
● ● ● bash – check each tool prints a version

$ gcc --version
gcc (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0
$ as --version
GNU assembler (GNU Binutils for Ubuntu) 2.42
$ gdb --version
GNU gdb (Ubuntu 15.1-1ubuntu1~24.04.1) 15.1
$ strace -V
strace -- version 6.8
$ nasm -v
NASM version 2.16.01
# your version numbers may differ – that is fine
```

A tool is missing? Ask the lab assistant. The lab PCs have no internet, so tools are installed from a local copy of the packages:

```
sudo dpkg -i /lab/debs/*.deb
```

On a PC with internet the same is done by:

```
sudo apt install build-essential \
gdb strace nasm
```

Make a folder for every lab

```
mkdir -p ~/sysprog/lab1
cd ~/sysprog/lab1
```

Keep a **lab notebook** (a plain text file is fine). Paste each command and its output, then write one line saying what it shows.

Lab 1 tasks, step by step

1

Write hello.c

`nano hello.c`, type the program, save with Ctrl+O, exit with Ctrl+X.

2

Run the four stages

`gcc -E, -S, -c`, then link. Fill a table with `wc -l` and `ls -l` for all five files.

3

Ask file what each file is

`file hello*`. Which files are ELF? Which one is executable?

4

Find the holes

`nm hello.o` and `objdump -d hello.o`. Copy the line with `00 00 00 00`.

5

Count system calls

`strace -c ./hello`. Then build with `-static` and count again.

6

Know your machine

Run `endian.c` and `sizes.c`. Then `lscpu | grep -i cache` and record the cache sizes.

Submit: your lab notebook with every command, its output, and one sentence per task. Bonus: why does the static file need fewer system calls?

When things go wrong: read the message

Error messages tell you **where** (file:line:column) and **what**. Read the first one first; later errors are often caused by it.

Missing semicolon

```
bash
$ gcc mistake.c -o mistake
mistake.c:3:22: error:
expected ';'
    before 'return'
```

Line 3, column 22. Add the ; at the end of the printf line.

Forgot #include

```
bash
$ gcc mistake.c -o mistake
warning: implicit
declaration of
    function 'printf'
note: include '<stdio.h>'
```

gcc even tells you the fix. A **warning** still builds, but never ignore it.

Forgot ./

```
bash
$ hello
hello: command not found
$ echo $?
127
```

The shell searches only the PATH folders. Type ./hello to run the file in this folder.

Not executable

```
bash
$ ./hello
./hello: Permission denied
$ echo $?
126
```

The file lost its x permission. Fix: `chmod +x hello`. (Trying to run hello.o gives the same kind of error.)

Habit: scroll up to the **first** error, read file:line:column, and look at the line **before** it too. A missing ; is reported on the next line.

Checkpoint: test yourself

Q1

Which program turns hello.s into hello.o?

→ the assembler (as)

Q2

In SIC, the CPU fetches the instruction at address 1006. What is PC afterwards?

→ 1009 (every SIC instruction is 3 bytes)

Q3

nm prints "U puts" for hello.o. What does U mean?

→ undefined: puts is used here but defined elsewhere (in libc)

Q4

On a little-endian PC, 0x0A0B0C0D is stored at addresses 100–103. Which byte is at 100?

→ 0D, the least significant byte

Q5

A user program runs the privileged instruction cli. What happens?

→ the CPU faults; the kernel kills it with SIGSEGV

Key ideas from today



System software supports the machine itself, and much of it depends on the machine's architecture.



gcc runs four programs (cpp, cc1, as, ld); the kernel does the fifth step, loading.



An object file is machine code with holes. The linker fills them; nm shows what is missing.



The CPU repeats fetch → decode → execute. PC says where; IR holds what.



Memory is bytes with addresses. x86-64 stores multi-byte values little-endian.



User mode cannot run privileged instructions. System calls, exceptions and interrupts are the only ways into the kernel.

Homework 1 (due before Lecture 2)

1. Trace

Add `SUB DELTA` after `ADD BETA` in our SIC program, with `DELTA = 6`. Write the new memory listing and the full trace table (Step, Phase, PC, IR, A) for `ALPHA = 9`, `BETA = 4`. What is `GAMMA`?

2. Registers

Compile `int add(int a, int b) { return a + b; }` with `gcc -S -O0` and again with `-O2`. In which registers do `a` and `b` arrive? Where does the result go? Why is the `-O2` version so much shorter?

3. Byte order

Predict, then check with a program: the bytes of `0xCAFEBAFE` (int) and `0x0102` (short) in address order on your PC.

4. Buffering

Write a program with three `printf` lines. Run `strace -e trace=write ./a.out`, then `strace -e trace=write ./a.out > out.txt`. Count the write calls each time. Explain the difference.

5. Think

In two or three sentences: what could go wrong on a lab PC with 40 users if any program could run `hlt` or `cli`?

Next lecture and reading

NEXT: LECTURE 2

Machine and assembly language

SIC and SIC/XE instruction formats and addressing modes.

Reading gcc's x86-64 output with confidence.

Your first hand-written assembly program, assembled with nasm and run.

Main text, Part A

L. L. Beck, **System Software: An Introduction to Systems Programming**, 3rd ed., Addison-Wesley, 1997. Today: Ch. 1 (§1.1–1.3).

Machine view

R. E. Bryant & D. R. O'Hallaron, **Computer Systems: A Programmer's Perspective**, 3rd ed., Pearson, 2016. Today: Ch. 1, §2.1.

Main text, Part B

D. P. Bovet & M. Cesati, **Understanding the Linux Kernel**, 3rd ed., O'Reilly, 2005; R. Love, **Linux Kernel Development**, 3rd ed., Addison-Wesley, 2010. Today: Ch. 1 of each.

User-space side

M. Kerrisk, **The Linux Programming Interface**, No Starch Press, 2010. Today: Ch. 3.