Logistics

- Discord: 108/128 students
- Go to OH and check-in with the TA. This is graded. Deadline: 1/27/25
  - 43/129 students
- Zybooks: check the instructions on the website
  - If you cannot access the textbook yet, please email me
  - 125/129 students. Average Score 94%
- Work on Sim1 even though you cannot submit on Gradescope yet.
  - Use the test cases provided

Recap

01-Numbers Representation and addition

- Everything in the computer is written as ones are zeros
  - Binary (**0b**) and Hexadecimal (**0x**)
- Computers have a finite number of bits to store integers (bit, nibble, byte, half-word, word)
- Modern computers use 2’s complement to represent integers
- Adders update the sum result bits, but also compute other bits like carryout and overflow.
- With 2’s complement, we do not need to build a circuit for subtraction
  - a - b = a + (-b)

Today

- Sign extension
- Overflow bit
- Subtraction

Let’s Jump to Assembly

- Intro to MIPS
- Registers
- Basic Operations

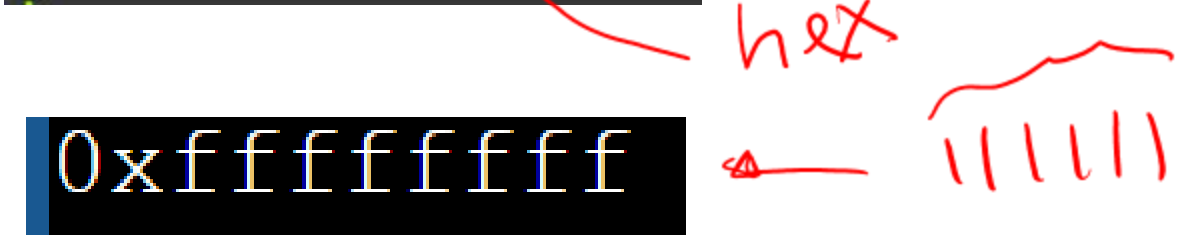
Sign Extension

What is the output?

```c
#include <stdio.h>

int main()
{
    int x = 1;
    printf("%d", x);
    return 0;
}
```

Why?

```c
#include <stdio.h>

int main()
{
    int x = -1;
    printf("%#x",x);

    return 0;
}
```



Sign Extension

- We often want to convert from a small format to a larger one.
  - This is trivial for positive numbers.

All of these numbers are equal to 6_{ten}:

8-bit							0000	0110
16-bit					0000	0000	0000	0110
32-bit	0000	0000	0000	0000	0000	0000	0000	0110

Sign Extension

- Use **sign extension** (pad with 1)
  - All of these numbers are equal to -8_{ten}

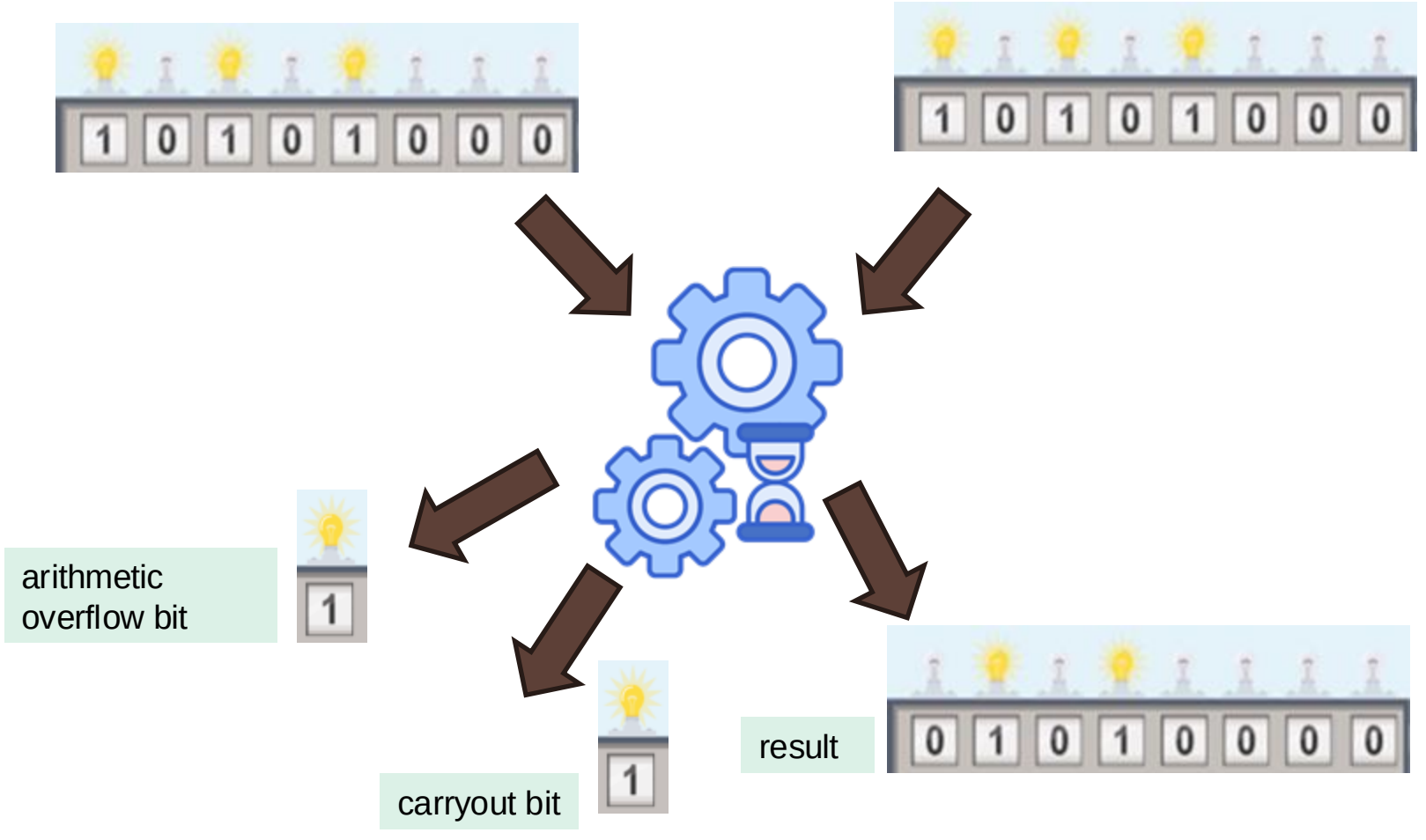
8-bit							1111	1000
16-bit					1111	1111	1111	1000
32-bit	1111	1111	1111	1111	1111	1111	1111	1000

Sign Extension

- Generally, to convert a **signed integer** to a larger signed integer, simply copy the sign bit (MSB) into all of the new bits
  - Positive numbers get more zeros
  - Negative numbers get more ones
- This is called **sign extension**

Let’s sum INTEGER numbers in the computer

Intro to the Arithmetic Overflow bit



Arithmetic Overflow

Adding two positives
May overflow

0010 + 0011 = 0101 (ok)
2 + 3 = 5

0111 + 0001 = 1000 (overflow)
7 + 1 ≠ -8

Adding two negatives
May overflow

1111 + 1111 = (1)1110 (ok)
-1 + -1 = -2

1000 + 1111 = (1)0111 (overflow)
-8 + -1 ≠ 7

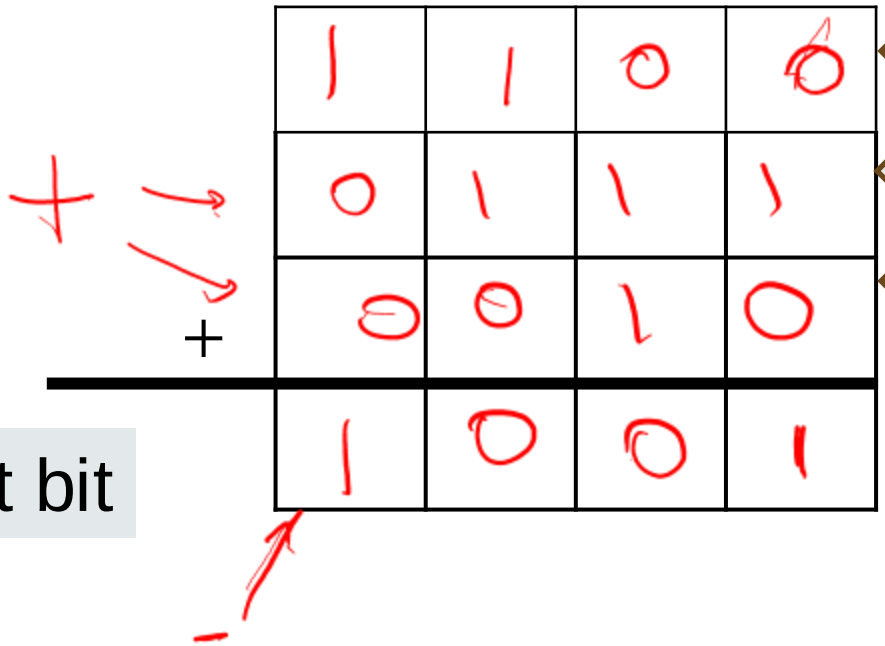
Adding positive and negative
Cannot overflow

1000 + 0111 = 1111 (ok)
-8 + 7 = -1

0010 + 1000 = 1010 (ok)
2 + -8 = -6

ICA question 3

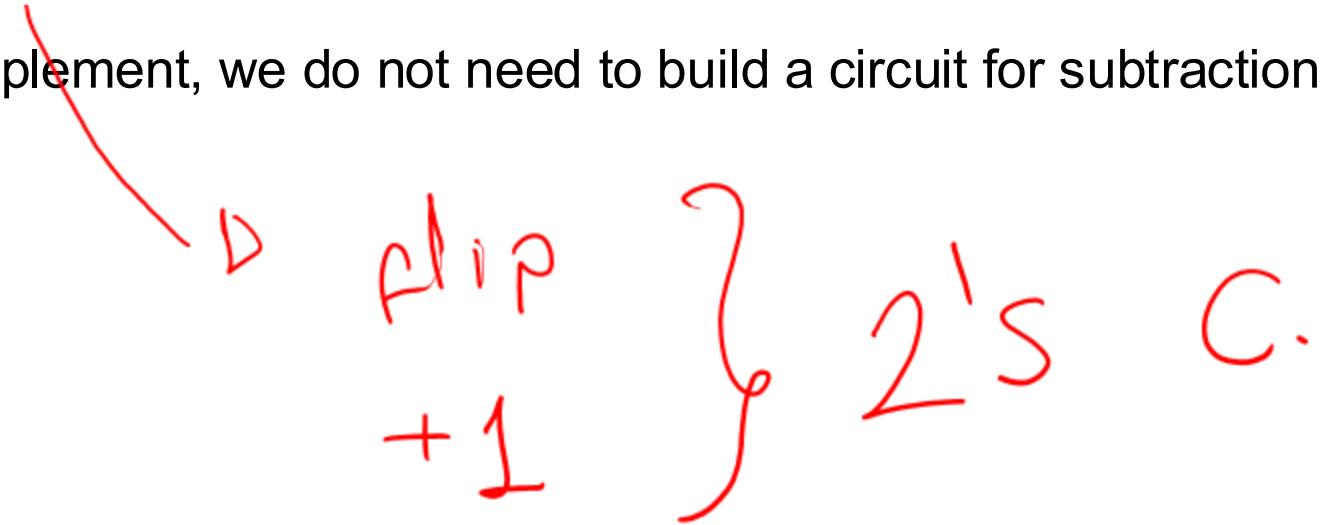
Show how a computer adds adds the 4-bit integers 0x7 + 0x2
Fill in the blanks:



Subtraction

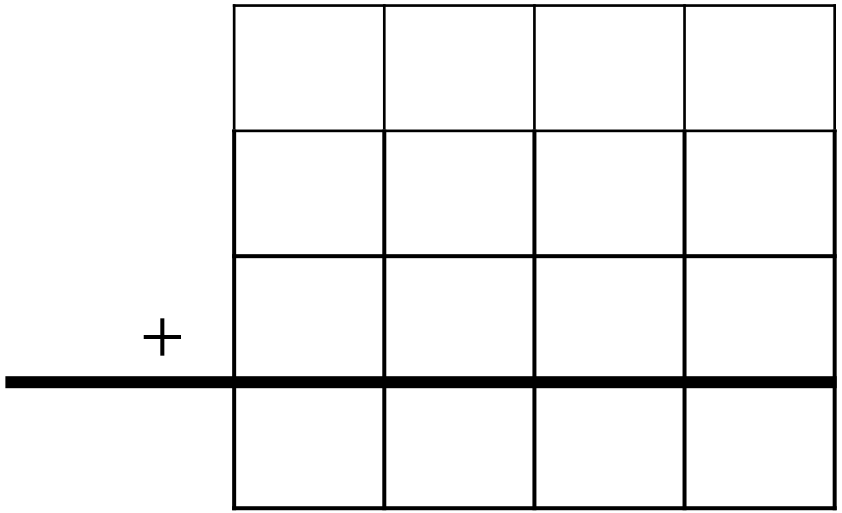
A - B = A+ (-B)

- With 2's complement, we do not need to build a circuit for subtraction



ICA question 4

Show how a computer adds adds the 4-bit integers 0x4 - 0x2
Fill in the blanks:



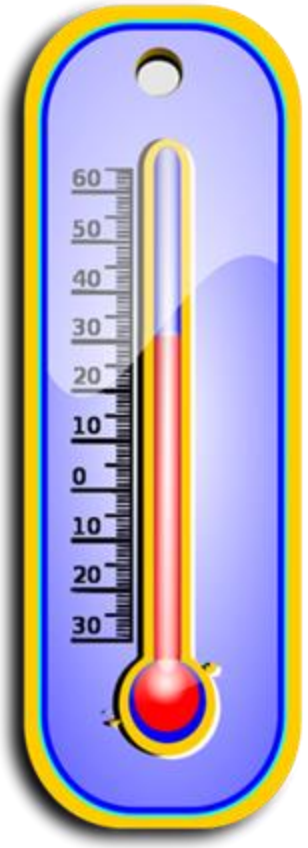
carryout bit

overflow bit

Temperature check

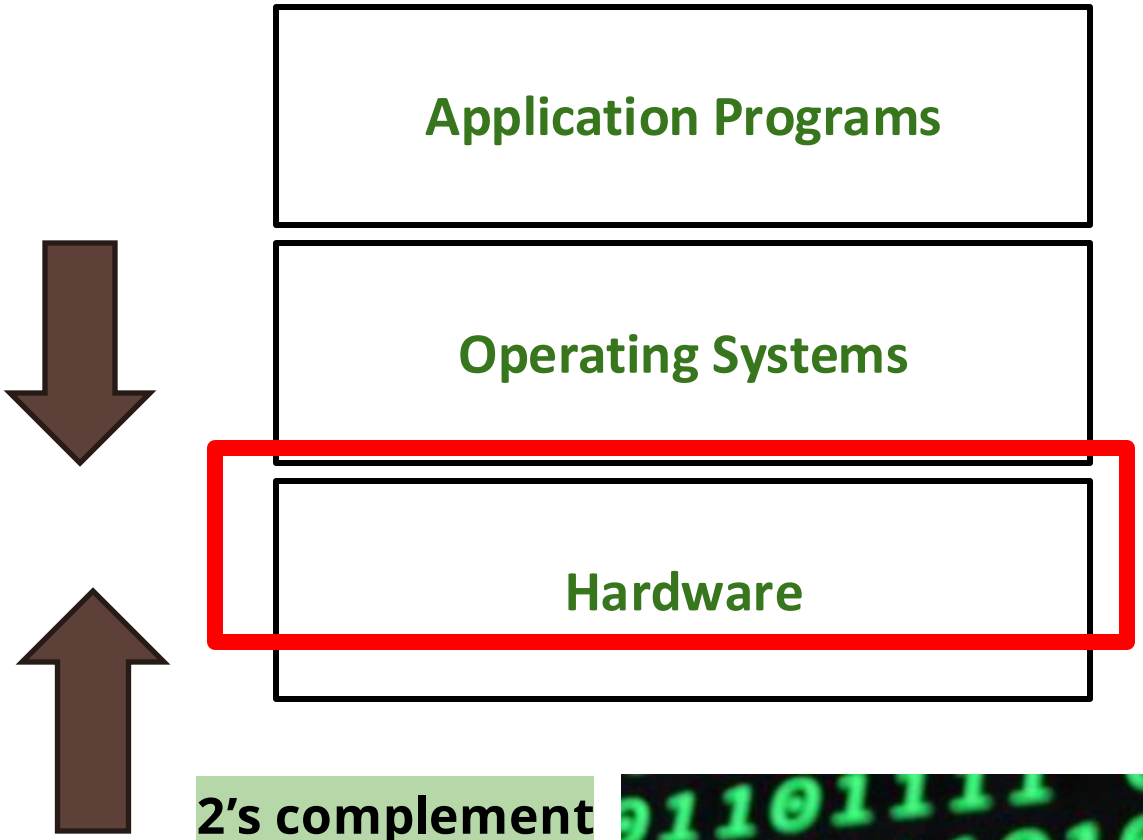
How are you feeling?

A. Very confused
B. Need a lot more practice
C. Need a little more practice
D. Just have a couple of questions
E. Feeling good

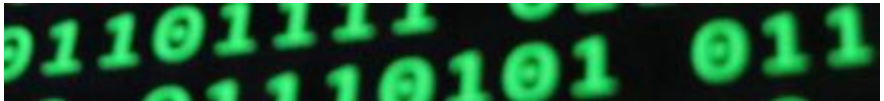


Overview

In this class



2's complement



Creating an executable program

source code

> vi hello.c

hello.c

```c
#include <stdio.h>

/**
 * main - Entry point, prints
 *
 * Return: Always 0 (Success)
 */
int main(void)
{
        puts("Hello, World");
        return (0);
}
```

compiler

> gcc -S hello.c

hello.s

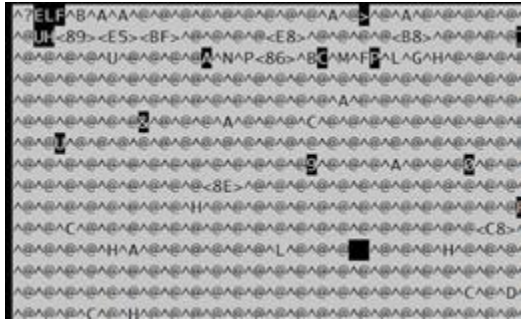
```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
```

assembly code

assembler and linker

> gcc hello.c

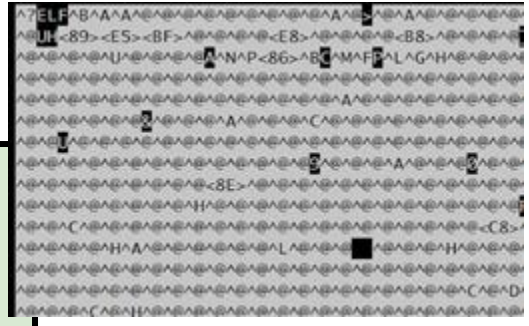
hello (or a.out)



object code

Let’s run a program

hello

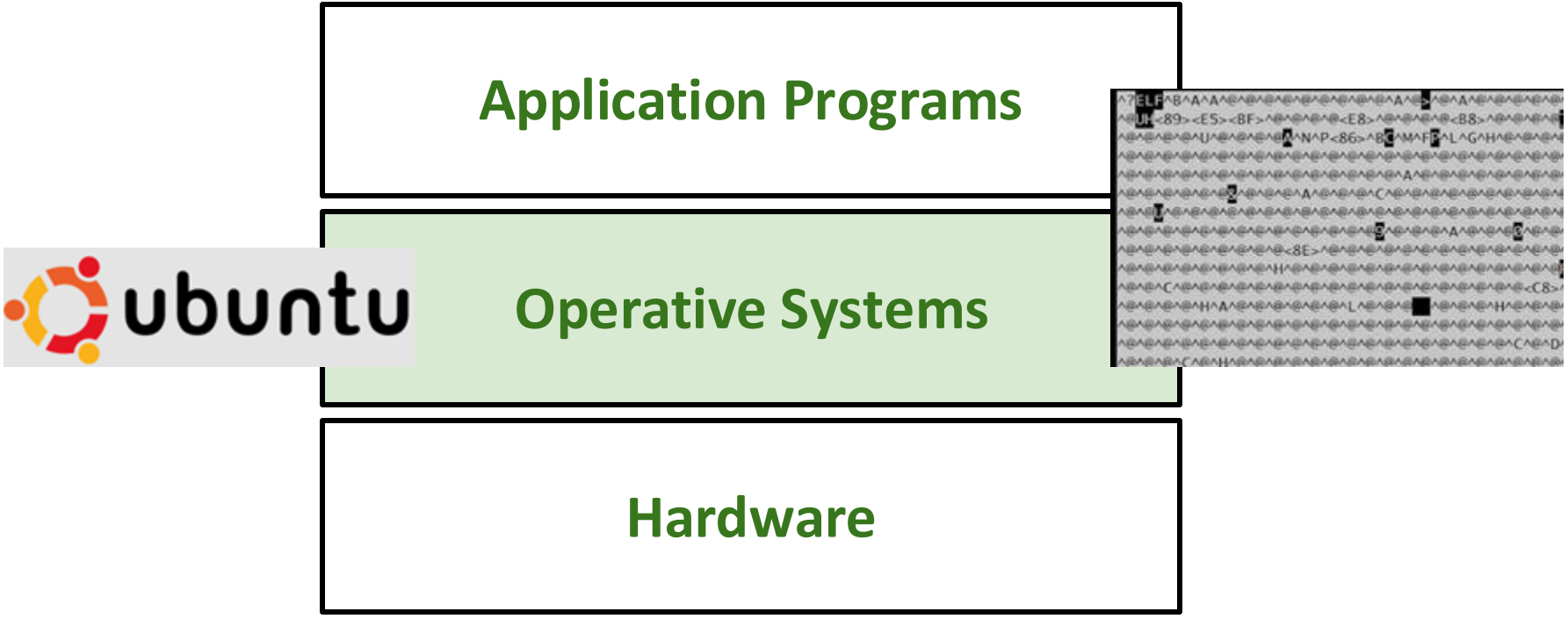


Application Programs

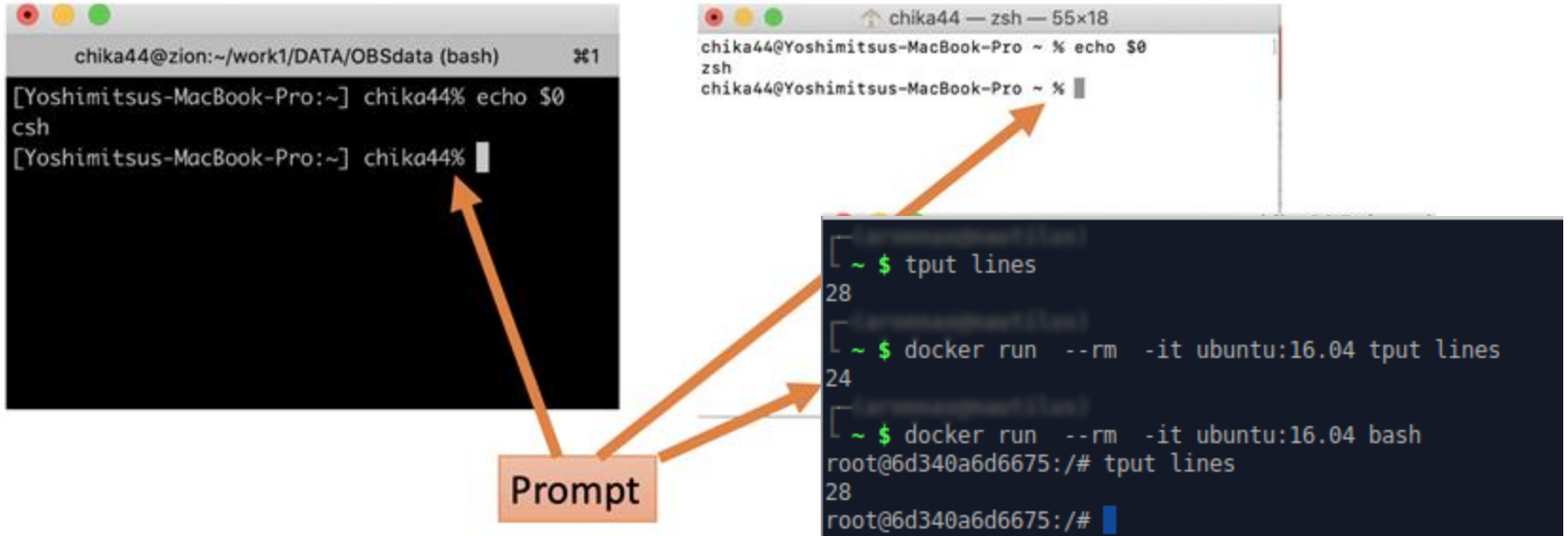
Operative Systems

Hardware

Let’s run a program



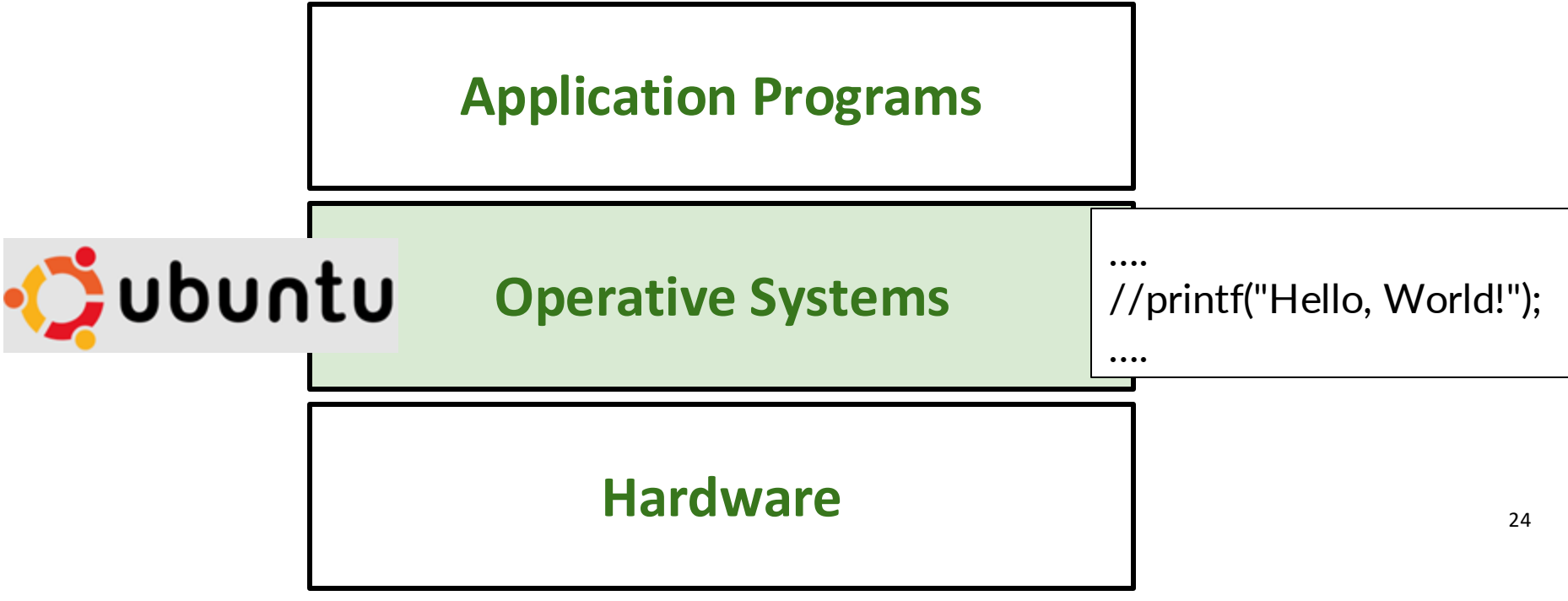
UNIX Shell



Command	Meaning	Options
$ ls	list	-la
$ mkdir	make directory	-p
$ cd	change directory	
$ cp	copy	-r
$ mv	move or rename	
$ rm	remove	-r

Basic Commands: UNIX Shell — UNIX 0.1 documentation

Let’s run a program



Let’s run a program

```
> ./hello
```

hello

hello

Application Programs

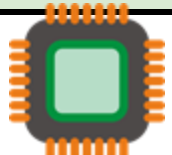
Operative Systems

Hardware

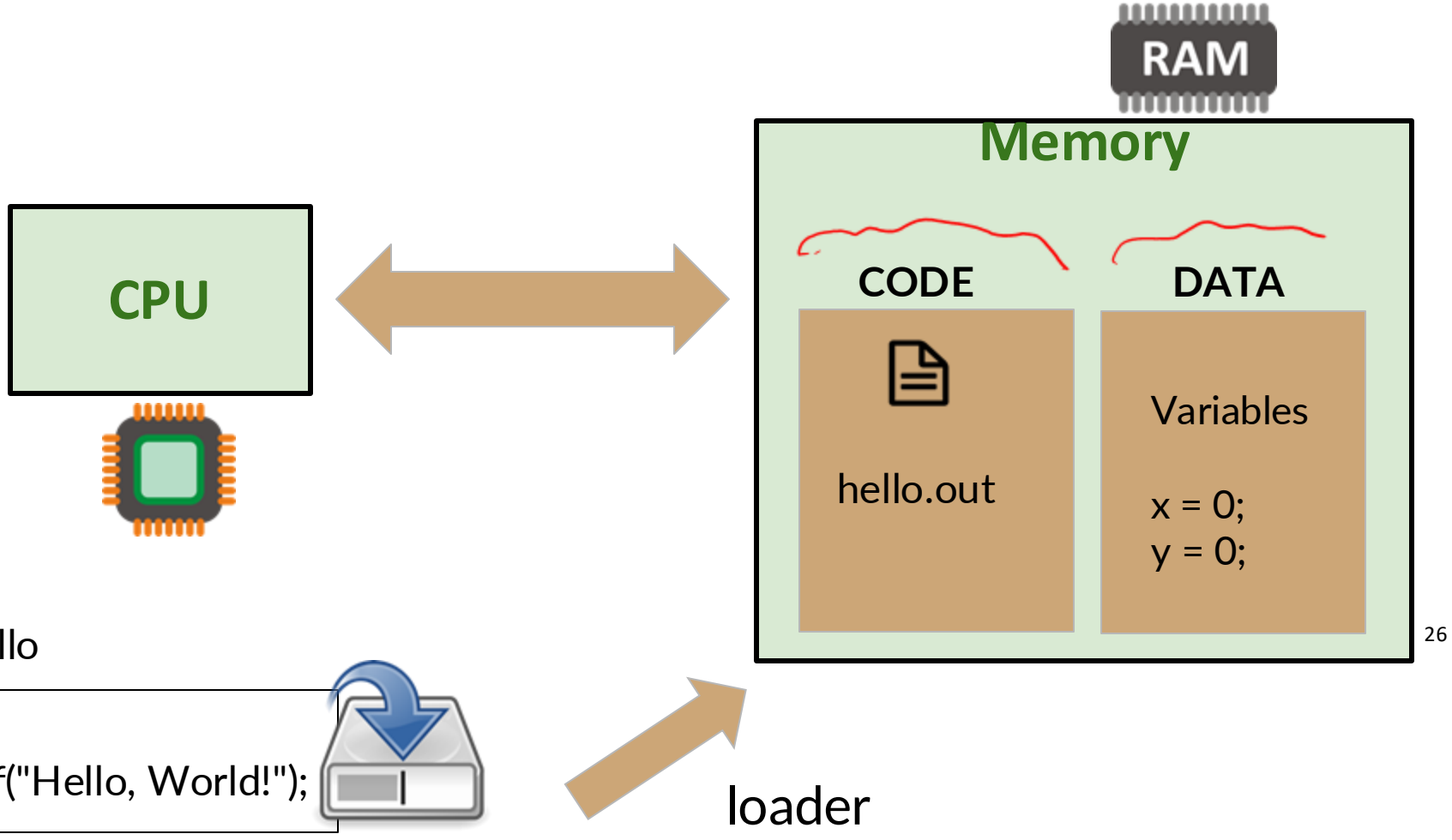
Memory

CPU

```
....
//printf("Hello, World!");
....
```



Let’s run a program



Assembly

Creating an executable program

source code

> vi hello.c

hello.c

```c
#include <stdio.h>

/**
 * main - Entry point, prints
 *
 * Return: Always 0 (Success)
 */
int main(void)
{
        puts("Hello, World");
        return (0);
}
```

compiler

> gcc -S hello.c

hello.s

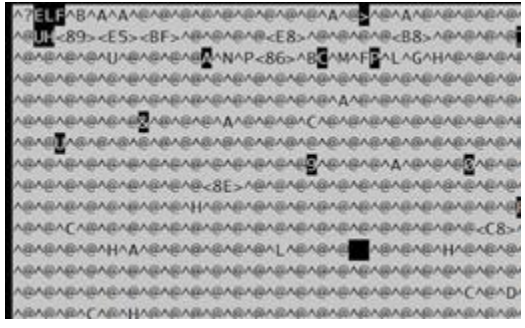
```
        .file   "main.c"
        .section        .rodata
.LC0:
        .string "Hello, World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
```

assembly code

assembler and linker

> gcc hello.c

hello (or a.out)

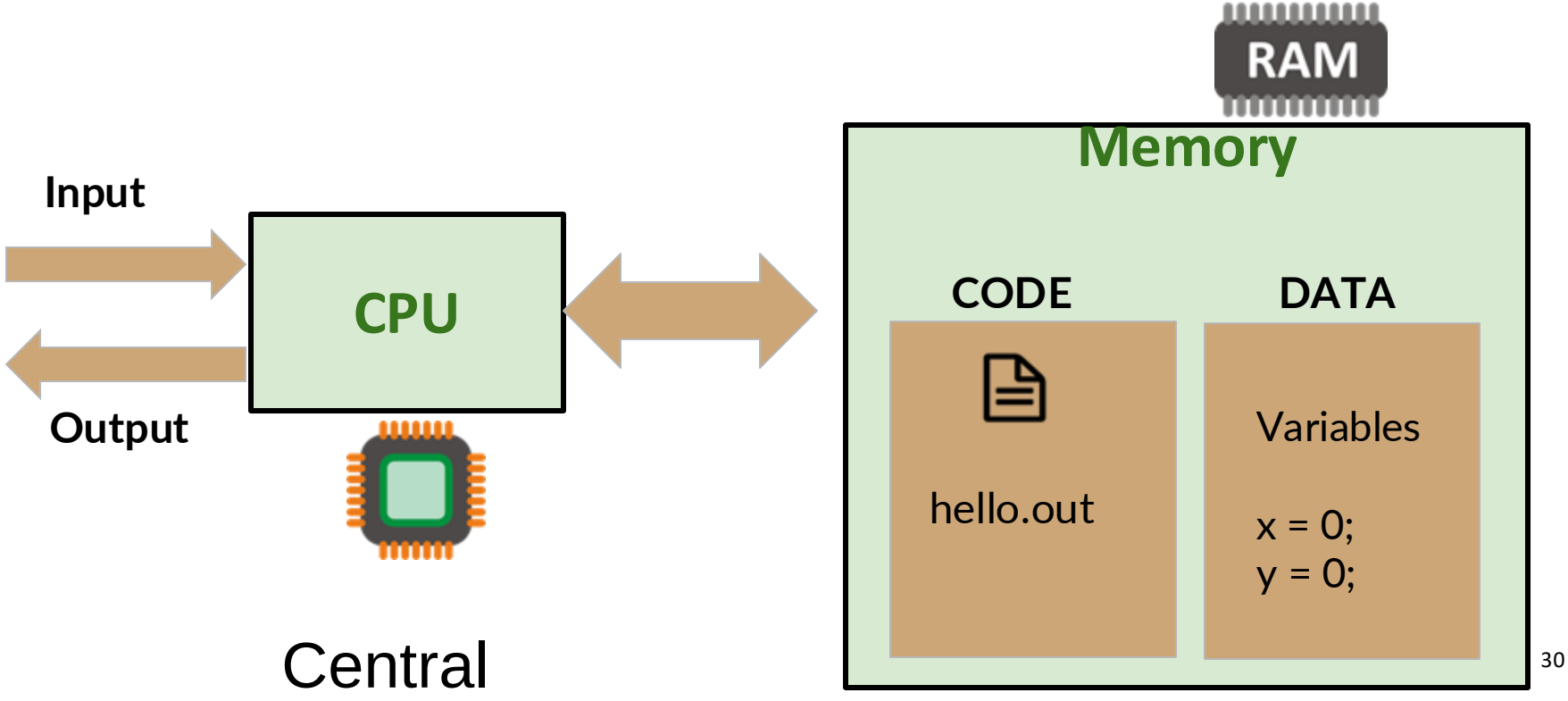


object code

Instruction Set Architecture (ISA)

- **Instruction Set Architecture** – An abstract model of a computer. It describes what a computer does, not how the computer does it.
- **Machine Code** – The binary instructions that a computer executes.
- **Assembly Code** – The human-readable form of the machine code, but more than that, too. Typically, there is a one-to-one match between an assembly language instruction and a machine language instruction.

Structure of CPU (simplified)



Central Processing Unit

Instruction Set Architecture

- CPU is actually very limited:
  - Read/write registers
  - Read/write memory
  - Do simple math (add, sub, logic)
  - Do simple comparisons (eq, less than)
  - Only one operation at a time (per CPU*)

Types of Assembly language (in general)

1. RISC Reduced Instruction Set Computer
2. DSP Digital Signal Processor
3. CISC Complex Instruction Set Computer
4. VLIW Very Long Instruction Word

Examples of Assembly Languages for some architectures

- x86 Assembly: Used for Intel and AMD processors
- ARM Assembly: Used for ARM processors, in mobile devices
- MIPS Assembly: Used for MIPS often utilized in academia
- RISC-V Assembly: Newish open ISA that is gaining popularity
- PowerPC Assembly: Used for PowerPC, in older Apple
- Z80 Assembly: Zilog Z80 microprocessor, in early PC
- PIC Assembly: Microchip's PIC microcontrollers

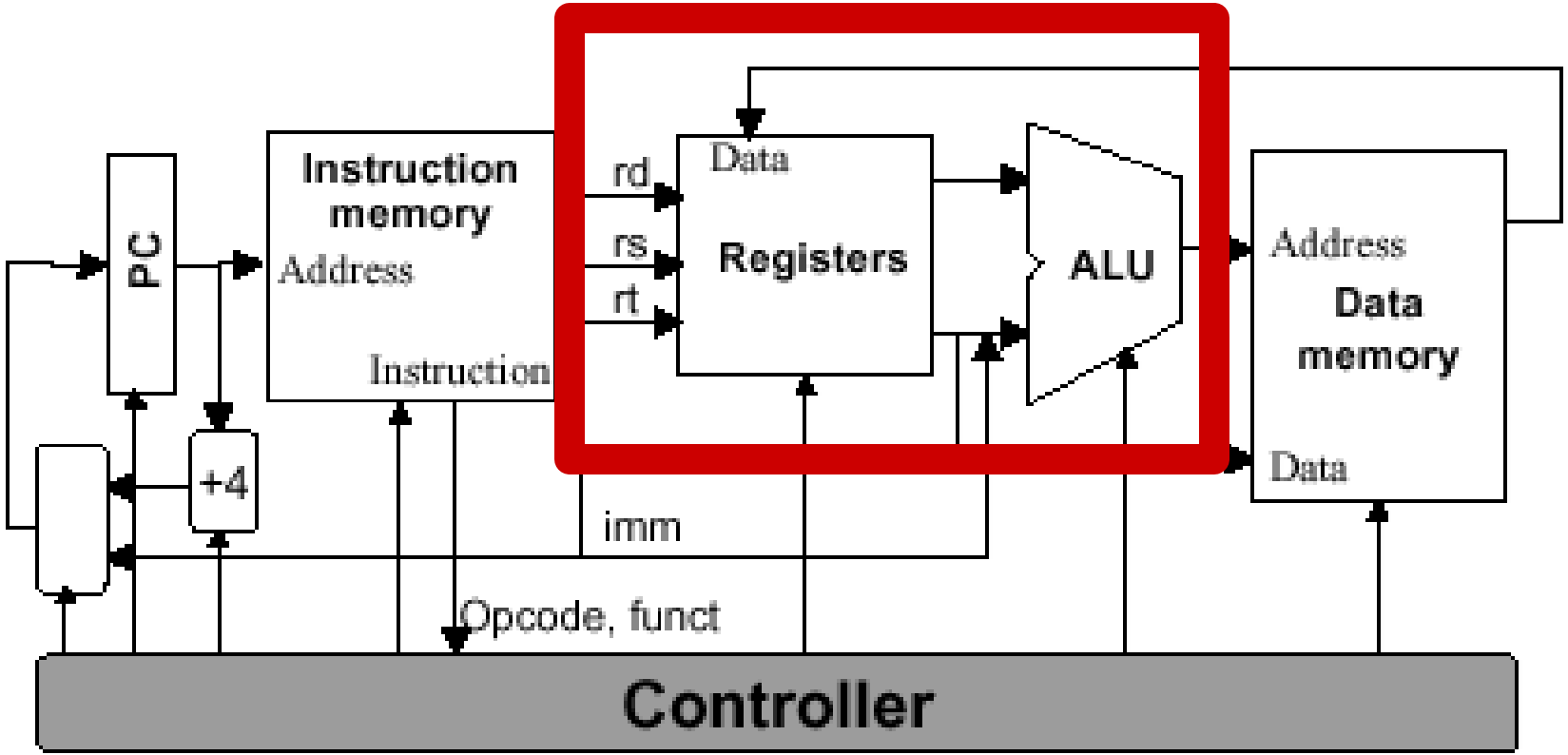
MIPS

What is MIPS (Microprocessor without Interlocked Pipeline Stages) ?

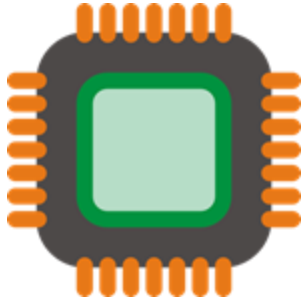
- MIPS is a RISC ISA
  - Reduced Instruction Set
  - Simpler, easier to understand
  - Used in real hardware (but rarely)

- CISC alternative: x86 (Intel/AMD)
  - Complex Instruction Set

Schematic diagram of MIPS architecture



What is a Register?



CPU can only operate on data stored in registers.

Each register has 32 bits

Identified with a Number (machine code)
and a Name (Assembly)

Example:

Register name **$zero** Register number: **0**

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

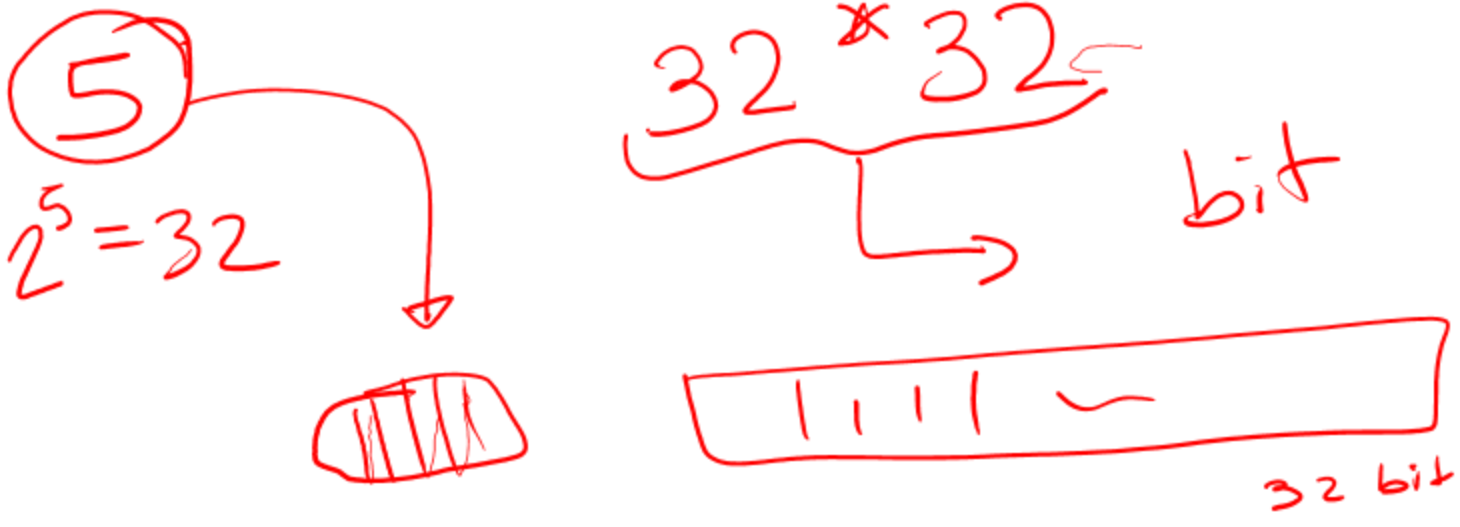
Registers

A few others:
PC
Floating-point
Hi/Lo

Name (**assembly**)	Number (**machine language**)	Purpose	Must Preserve
$zero	0	Always zero; readonly	n/a
$at	1	Reserved for assembler	n/a
$v0-$v1	2-3	Return values	no
$a0-$a3	4-7	Parameters	no*
$t0-$t7	**8-15**	**Temporaries**	**no**
$s0-$s7	**16-23**	**Saved values**	**yes**
$t8-$t9	**24-25**	**Temporaries**	**no**
$k0-$k1	26-27	Reserved for OS	n/a
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

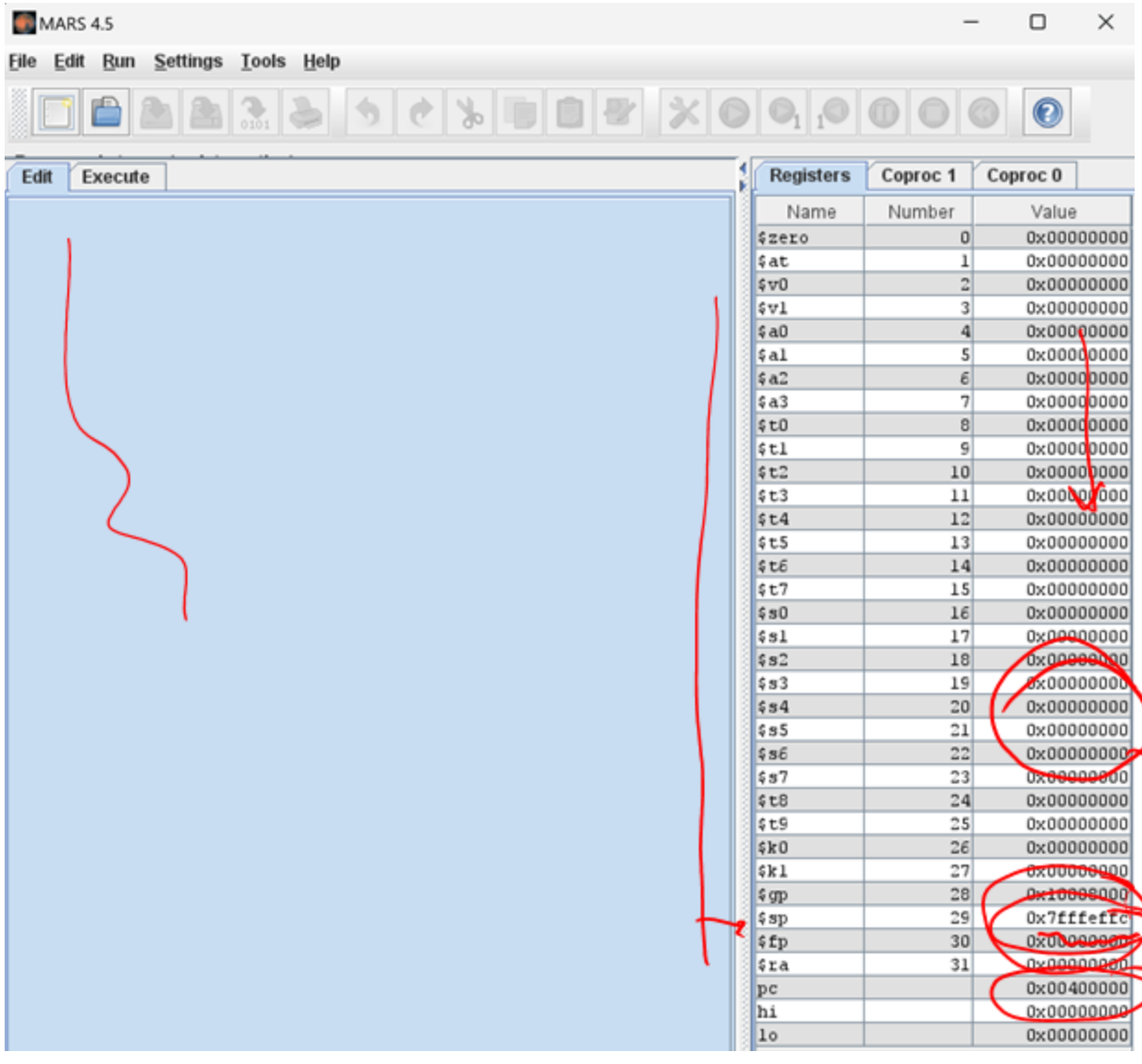
Given that the MIPS CPU has 32 registers, How many bits do we need to identify a register?

i.e. to store



Registers You Can Use

- `$sX` general purpose
  (later: will be “saved” registers)
- `$tX` general purpose
  (later: will be “temporary” registers)
- `$zero` constant
- `$a0/$v0` syscall parameters
  (later: arguments, return values)



How many nibbles can you store in a register?

32/4

hex

8

Logic and arithmetic operations

Available Operations

- Basic 3-operand instructions:

add	32-bi
sub	32-bi
and	bitwise AND
or inclusive OR	bitwise
nor	bitwise OR, negated
xor	bitwise exclusive OR

How would you perform 8-bit or 16-bit addition?

Why aren't there any logical AND/OR operations?

Where is the NOT operator?

Basic Instructions

```
add $s0,$t1,$t2          # s0 = t1 + t2
```

- Basic arithmetic instructions read from two registers, write to a third

- `add $s0,$s0,$s0          # s0 *= 2`
- OK to use the same register 2 or 3 times

Building Complex Operations

- Build complex operations using a sequence of simpler steps

```
add $t0,$s1,$s2      # t0=s1+s2
add $s0,$t0,$s3      # s0=s1+s2+s3
```

- Use registers to hold temporary values
  - Be careful not to overwrite important values!

Building Complex Operations

.When writing a complex expression, it's OK to use the **destination register** as a temporary:

```
add     $s0,$s1,$s2     # s0=s1+s2
add     $s0,$s0,$s3     # s0=s1+s2+s3
```

.**WARNING:** Never overwrite any register before it is used the last time. How would you write this?

```
$s0 *= 3;
```

ICA Question 5

Assume that the following variables are in registers:

```
alpha        $s0
bravo        $s1
charlie      $s2
```

Write a sequence of instructions which will calculate the following values. You may use any `$tX` registers, but **do not modify** any `$sX` registers (except the one you're required to modify):

```
charlie = alpha + bravo + charlie
alpha   = - bravo - charlie
bravo   = charlie - (alpha + bravo)
alpha = bravo*2 + charlie*3
```

zero − bravo

→ $t1

A solution

- alpha $s0
- bravo $s1
- charlie $s2

```
# charlie = alpha + bravo + Charlie

add $t0,$s0,$s1 # t0 = alpha+bravo
add $s2,$t0,$s2 # s2 = alpha+bravo+charlie
```

A solution

- alpha $s0
- bravo $s1
- charlie $s2

```
# alpha  = - bravo – charlie

sub $t0,$zero,$s1 # t0 = 0-bravo
sub $s0,$t0,$s2 # s0 = -bravo-charlie
```

A solution

- alpha $s0
- bravo $s1
- charlie $s2

```
#bravo = charlie - (alpha + bravo)
```

```
add $t0,$s0,$s1 # t0 = alpha+bravo
sub $s1,$s2,$t0 # s1 = charlie-(alpha+bravo)
```

A solution

- alpha $s0
- bravo $s1
- charlie $s2

```
# alpha = bravo*2 + charlie*3

add $t0,$s1,$s1 # t0 = 2*bravo
add $t1,$s2,$s2 # t1 = 2*charlie
add $t1,$t1,$s2 # t1 = 3*charlie
add $s0,$t0,$t1 # s0 = 2*bravo+3*charlie
```

Immediate Operations

- No `subi`

`addi`'s immediate is **signed** (sign extend to 32 bits)

```
addi $t0,$t0,-1     # t0--
```

Immediate Operations

- Most (not all) 3-register instructions have an "immediate" alternative
  - Replace 2nd input with a 16-bit constant
  - Hint: `$zero+constant` is often useful

```
addi $t0,$t0,1
andi $s0,$s3,0x3
ori  $t7,$s0,0xff
addi $s5,$zero,11
```

Group Exercise:

Convert each of these operations to **pseudocode.**
Use the register names like variable names.

EXIT POLL