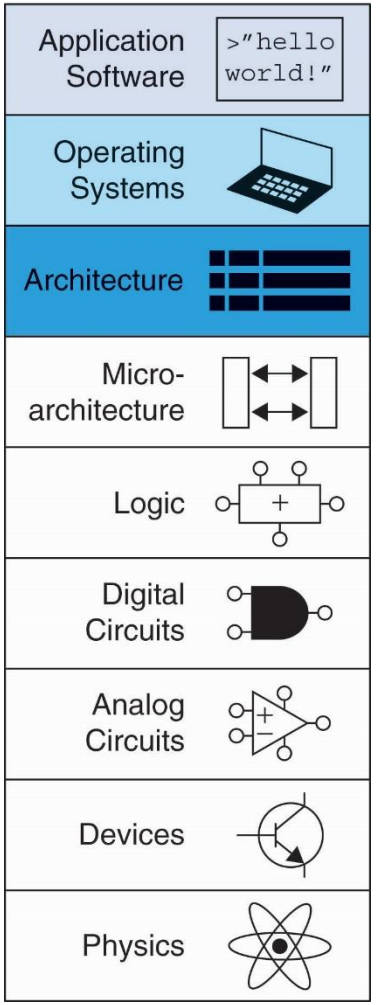
# Digital Design & Computer Architecture

Sarah Harris & David Harris

# Chapter 6: Architecture

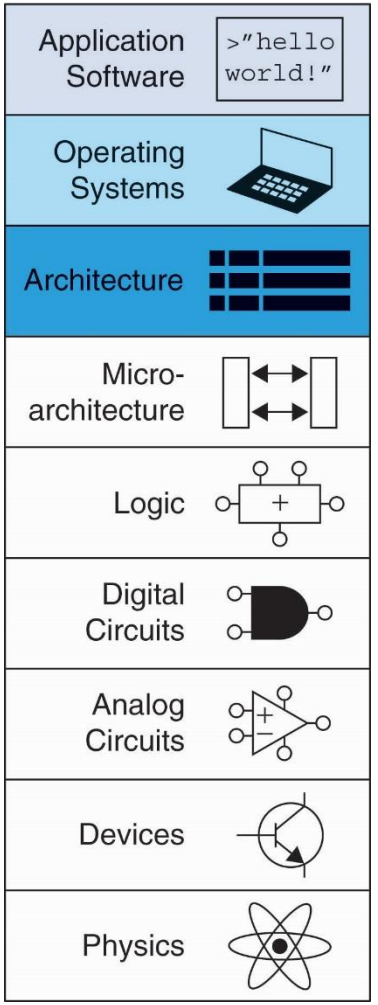
# Chapter 6 :: Topics

- Introduction
- Assembly Language
- Programming
- Machine Language
- Addressing Modes
- Lights, Camera, Action: Compiling, Assembly, & Loading
- Odds & Ends



# Introduction

- Jumping up a few levels of abstraction
- **Architecture:** programmer's view of computer
  - Defined by instructions & operand locations
- **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)



# Assembly Language

- **Instructions:** commands in a computer's language
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- **RISC-V architecture:**
  - Developed by Krste Asanovic, David Patterson and their colleagues at UC Berkeley in 2010.
  - First widely accepted open-source computer architecture

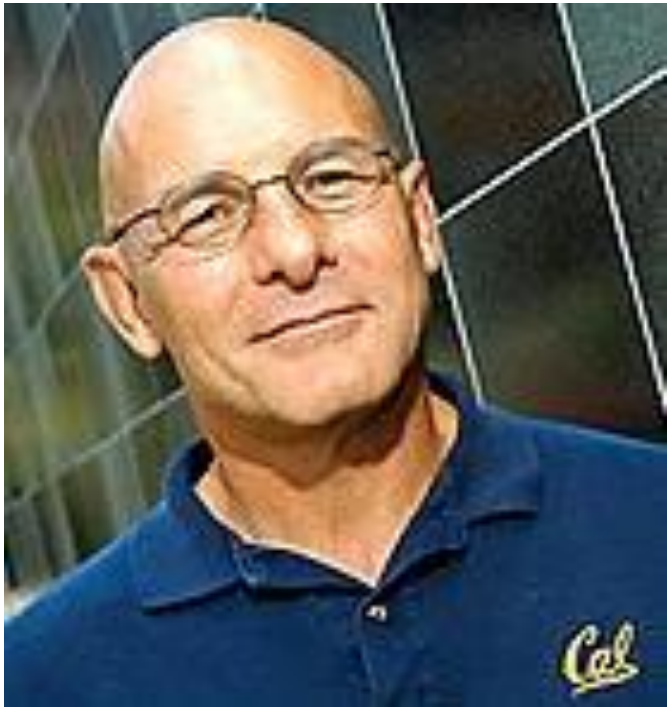
Once you've learned one architecture, it's easier to learn others

# Kriste Asanovic

- Professor of Computer Science at the University of California, Berkeley
- Developed RISC-V during one summer
- Chairman of the Board of the RISC-V Foundation
- Co-Founder of SiFive, a company that commercializes and develops supporting tools for RISC-V

# David Patterson

- Professor of Computer Science at the University of California, Berkeley since 1976
- Coinvented the Reduced Instruction Set Computer (**RISC**) with John Hennessy in the 1980's
- Founding member of RISC-V team.
- Was given the Turing Award (with John Hennessy) for pioneering a quantitative approach to the design and evaluation of computer architectures.



# John Hennessy

- President of Stanford University from 2000 - 2016.
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Coinvented the Reduced Instruction Set Computer (**RISC**) with David Patterson in the 1980's
- Was given the Turing Award (with David Patterson) for pioneering a quantitative approach to the design and evaluation of computer architectures.

# Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

1. **Simplicity favors regularity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

Chapter 6: Architecture

# Instructions

# Instructions: Addition

**C Code**

```
a = b + c;
```

**RISC-V assembly code**

```
add a, b, c
```

- **`add:`** mnemonic indicates operation to perform
- **`b, c:`** source operands (on which the operation is performed)
- **`a:`** destination operand (to which the result is written)

# Instructions: Subtraction

Similar to addition - only **mnemonic** changes

**C Code**

```
a = b - c;
```

**RISC-V assembly code**

```
sub a, b, c
```

- `sub:` mnemonic
- `b, c:` source operands
- `a:` destination operand

# Design Principle 1

## Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware

# Multiple Instructions

More complex code is handled by multiple RISC-V instructions.

**C Code**

```
a = b + c - d;
```

**RISC-V assembly code**

```
add t, b, c  # t = b + c
sub a, t, d  # a = t - d
```

# Design Principle 2

## Make the common case fast

- RISC-V includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- RISC-V is a ***reduced instruction set computer*** **(RISC)**, with a small number of simple instructions
- Other architectures, such as Intel's x86, are ***complex instruction set computers*** **(CISC)**

Chapter 6: Architecture

# Operands

# Operands

- **Operand location:** physical location in computer
  - Registers
  - Memory
  - Constants (also called *immediates*)

# Operands: Registers

- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V called “32-bit architecture” because it operates on 32-bit data

# Design Principle 3

## Smaller is Faster

- RISC-V includes only a small number of registers

# RISC-V Register Set

| Name | Register Number | Usage |
|---|---|---|
| `zero` | x0 | Constant value 0 |
| `ra` | x1 | Return address |
| `sp` | x2 | Stack pointer |
| `gp` | x3 | Global pointer |
| `tp` | x4 | Thread pointer |
| `t0-2` | x5-7 | Temporaries |
| `s0/fp` | x8 | Saved register / Frame pointer |
| `s1` | x9 | Saved register |
| `a0-1` | x10-11 | Function arguments / return values |
| `a2-7` | x12-17 | Function arguments |
| `s2-11` | x18-27 | Saved registers |
| `t3-6` | x28-31 | Temporaries |

# Operands: Registers

- **Registers:**
  - Can use either name (i.e., `ra`, `zero`) or `x0`, `x1`, etc.
  - Using name is preferred
- Registers used for **specific purposes**:
  - `zero` always holds the **constant value 0**.
  - the ***saved registers***, `s0-s11`, used to hold variables
  - the ***temporary registers***, `t0-t6`, used to hold intermediate values during a larger computation
  - Discuss others later

# Instructions with Registers

- Revisit `add` instruction

**C Code**

```
a = b + c;
```

**RISC-V assembly code**

```
# s0 = a, s1 = b, s2 = c
add s0, s1, s2
```

**#** indicates a single-line comment

# Instructions with Constants

- `addi` instruction

**C Code**

```
a = b + 6;
```

**RISC-V assembly code**

```
# s0 = a, s1 = b
addi s0, s1, 6
```

Chapter 6: Architecture

# Memory Operands

# Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers

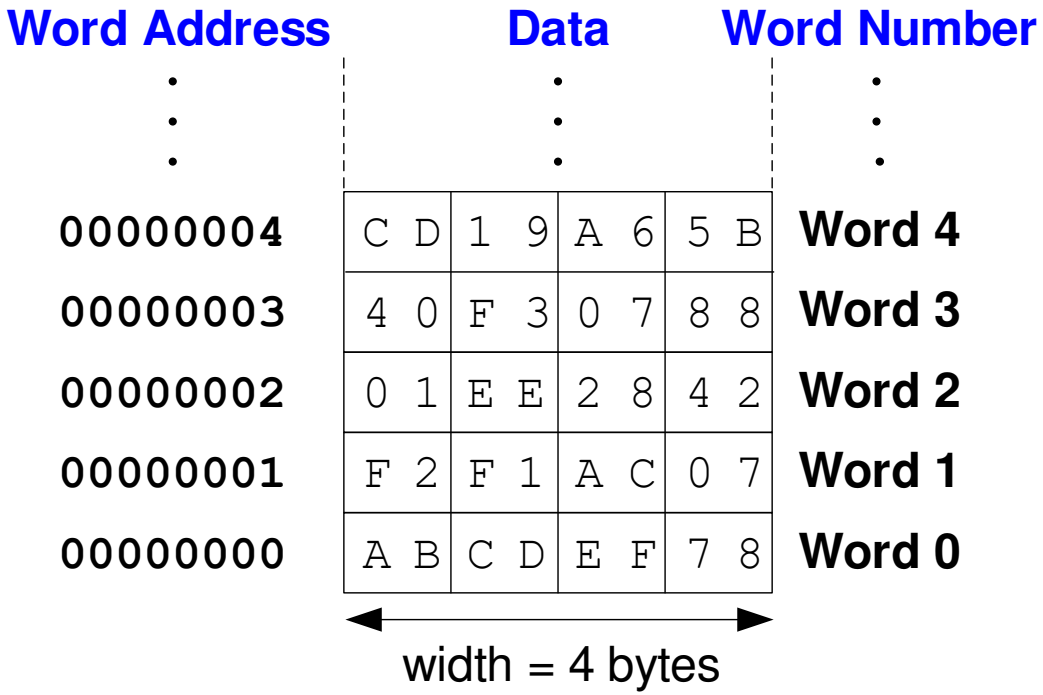
# Memory

- First, we'll discuss **word-addressable** memory
- Then we'll discuss **byte-addressable** memory

RISC-V is **byte-addressable**

# Word-Addressable Memory

- Each 32-bit data word has a unique address




RISC-V uses **byte-addressable** memory, which we'll talk about next.

# Reading Word-Addressable Memory

- Memory read called ***load***
- **Mnemonic:** *load word* (`lw`)
- **Format:**

```
lw t1, 5(s0)
lw destination, offset(base)
```

- **Address calculation:**
  - add *base address* (`s0`) to the *offset* (5)
  - address = (`s0` + 5)
- **Result:**
  - `t1` holds the data value at address (`s0` + 5)

**Any register** may be used as base address

# Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into `s3`
  - address = (0 + 1) = 1
  - `s3` = 0xF2F1AC07 after load

**Assembly code**

```
lw s3, 1(zero) # read memory word 1 into s3
```

| Word Address | Data | Word Number |
|---|---|---|
| 00000004 | C D 1 9 A 6 5 B | Word 4 |
| 00000003 | 4 0 F 3 0 7 8 8 | Word 3 |
| 00000002 | 0 1 E E 2 8 4 2 | Word 2 |
| 00000001 | **F 2 F 1 A C 0 7** | Word 1 |
| 00000000 | A B C D E F 7 8 | Word 0 |

# Writing Word-Addressable Memory

- Memory write is called a ***store***
- **Mnemonic:** *store word* (`sw`)

# Writing Word-Addressable

- **Example:** Write (store) the value in `t4` into memory address 3
  - add the base address (`zero`) to the offset (0x3)
  - address: (0 + 0x3) = 3
  - for example, if `t4` holds the value 0xFEEDCABB, then after this instruction completes, word 3 in memory will contain that value

Offset can be written in **decimal** (default) or **hexadecimal**

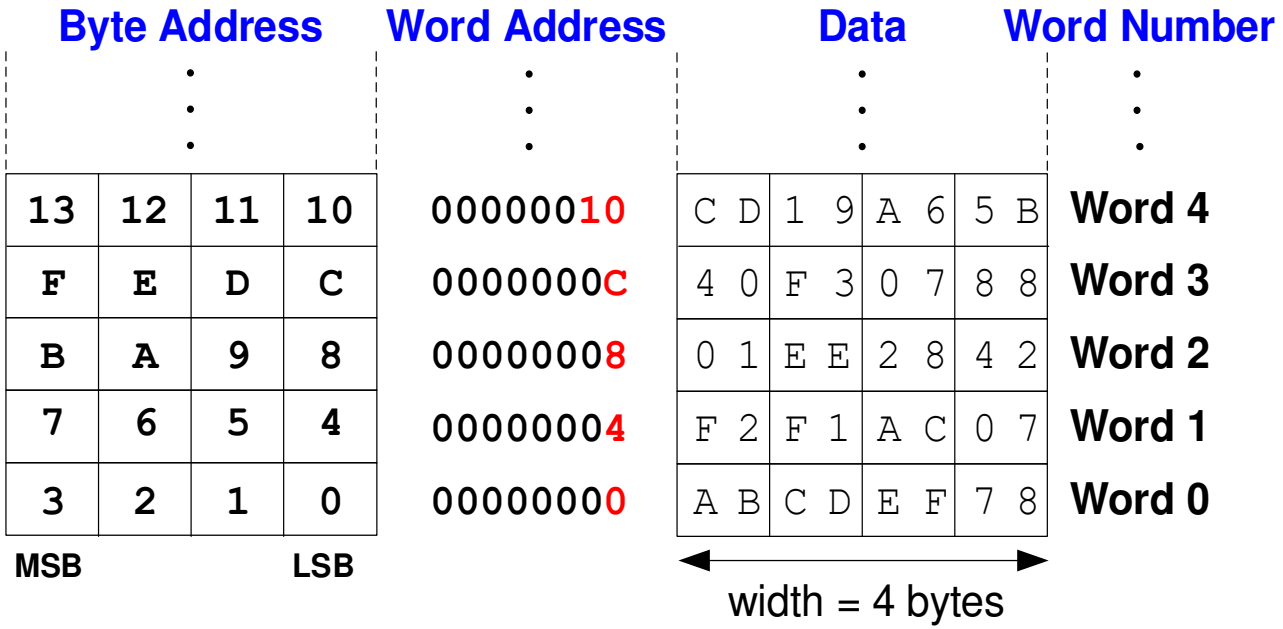
**Assembly code**

```
sw t4, 0x3(zero)   # write the value in t4
                   # to memory word 3
```

| Word Address | Data | Word Number |
|---|---|---|
| 00000004 | C D 1 9 A 6 5 B | Word 4 |
| 00000003 | **F E E D C A B B** | Word 3 |
| 00000002 | 0 1 E E 2 8 4 2 | Word 2 |
| 00000001 | F 2 F 1 A C 0 7 | Word 1 |
| 00000000 | A B C D E F 7 8 | Word 0 |

# Byte-Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- 32-bit word = 4 bytes, so word address **increments by 4**



# Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is $2 \times 4 = 8$
  - the address of memory word 10 is $10 \times 4 = 40$ (0x28)

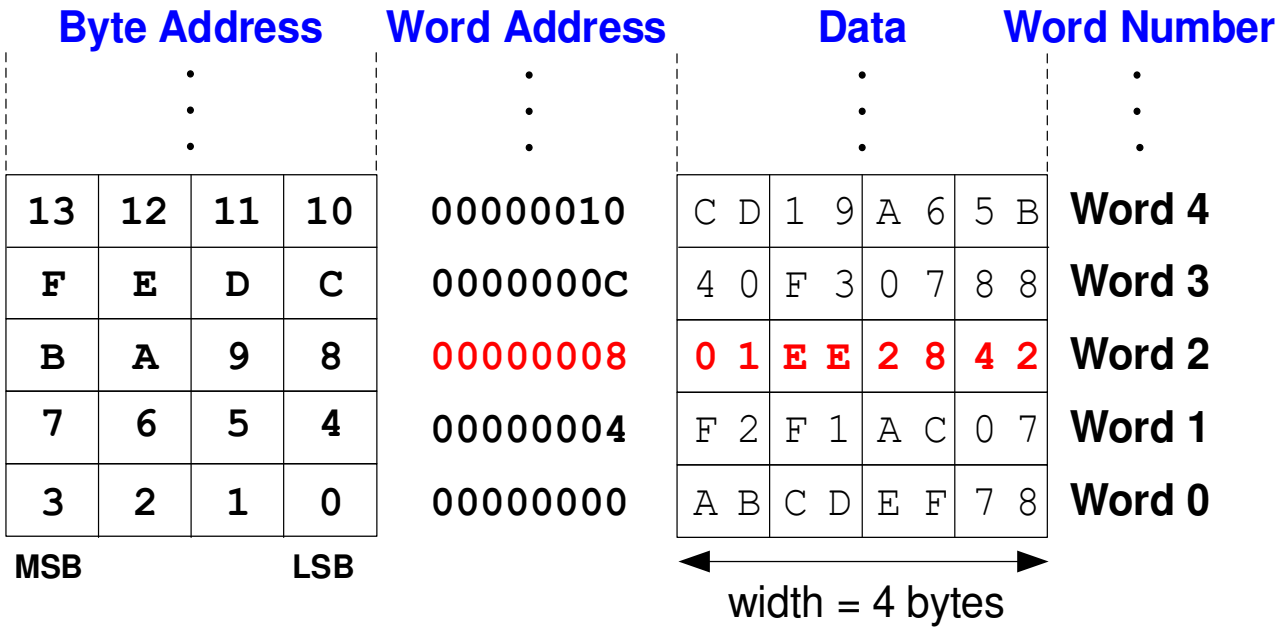
- RISC-V is **byte-addressed**, not word-addressed

# Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 8 into `s3`.
- `s3` holds the value 0x1EE2842 after load

**RISC-V assembly code**

```
lw s3, 8(zero)  # read word at address 8 into s3
```

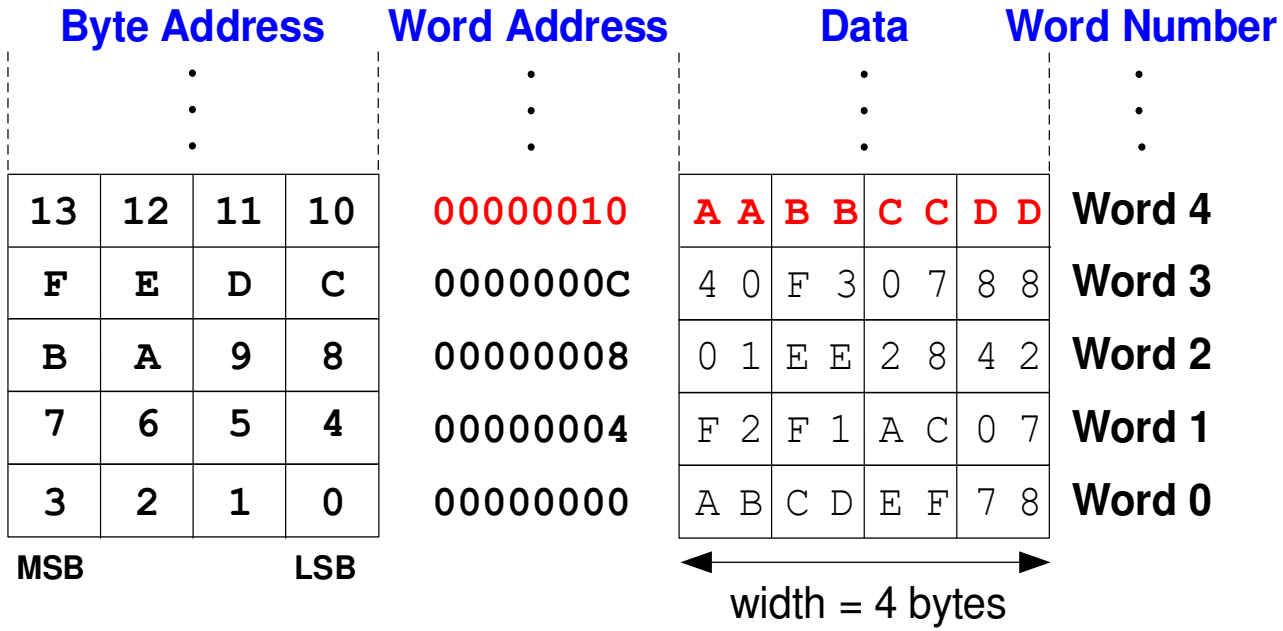


# Writing Byte-Addressable Memory

- **Example:** store the value held in `t7` into memory address 0x10 (16)
  - if `t7` holds the value 0xAABBCCDD, then after the `sw` completes, word 4 (at address 0x10) in memory will contain that value

**RISC-V assembly code**

```
sw t7, 0x10(zero)  # write t7 into address 16
```



Chapter 6: Architecture

# Generating Constants

# Generating 12-Bit Constants

- 12-bit signed constants (immediates) using `addi`:

**C Code**

```
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

**RISC-V assembly code**

```
# s0 = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

Any immediate that needs **more than 12 bits** cannot use this method.

# Generating 32-bit Constants

- Use load upper immediate (`lui`) and `addi`
- `lui`: puts an immediate in the upper 20 bits of destination register and 0's in lower 12 bits

**C Code**

```
int a = 0xFEDC8765;
```

**RISC-V assembly code**

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

Remember that `addi` **sign-extends** its 12-bit immediate

# Generating 32-bit Constants

- If **bit 11** of 32-bit constant is **1**, increment upper 20 bits by **1** in `lui`

**C Code**

```
int a = 0xFEDC8EAB;
```

**Note:** -341 = 0xEAB

**RISC-V assembly code**

```
# s0 = a
lui  s0, 0xFEDC9     # s0 = 0xFEDC9000
addi s0, s0, -341    # s0 = 0xFEDC9000 + 0xFFFFFEAB
                     #    = 0xFEDC8EAB
```