CPE 100 Digital logic Design

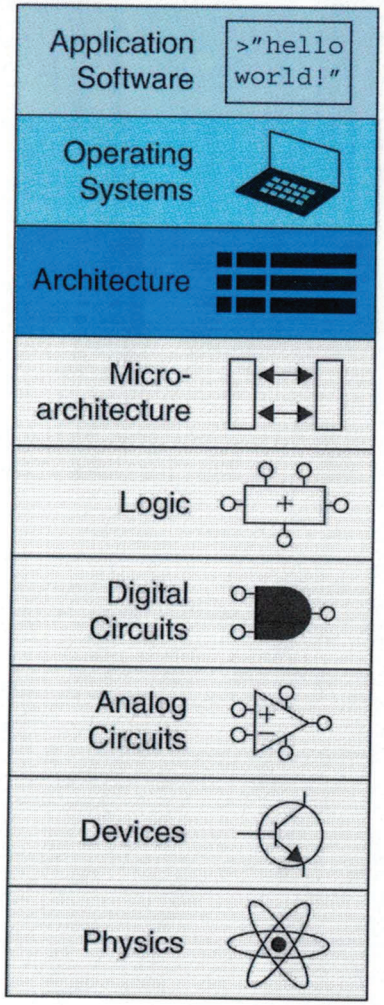
Lecture 26

MAY 3, 2021

COMPUTER ARCHITECTURE

Chapter 6 :: Topics

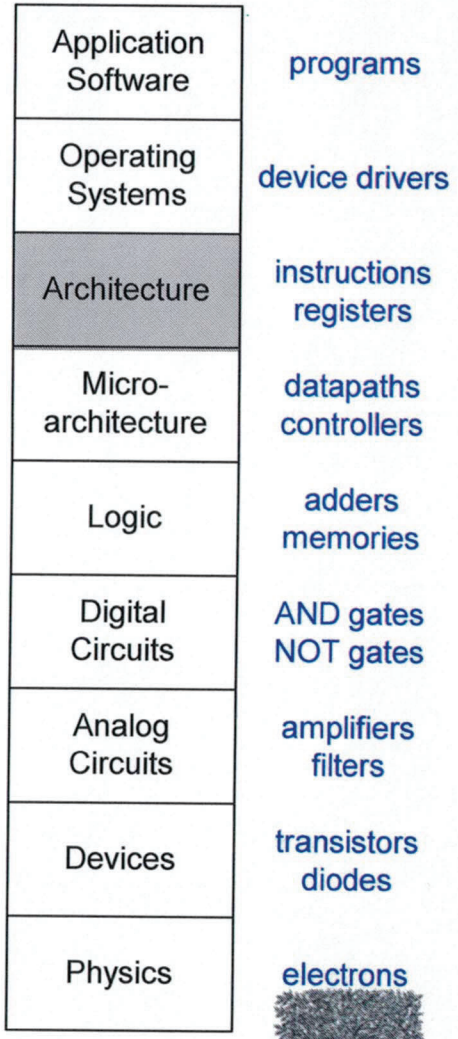
- Introduction
- Assembly Language
- Machine Language
- Programming
- Addressing Modes
- Lights, Camera, Action: Compiling, Assembling, & Loading
- Odds and Ends



ELSEVIER

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)
- **MIPS** architecture:
  - Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
  - Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco

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

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**

Instructions: Addition

C Code

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

MIPS 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;
```

MIPS 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 MIPS instructions.

C Code

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

MIPS assembly code

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

Design Principle 2

Make the common case fast

- MIPS 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
- MIPS 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)**

Operands

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

Operands: Registers

- MIPS has 32 32-bit registers
- Registers are faster than memory
- MIPS called "32-bit architecture" because it operates on 32-bit data

MIPS Register Set

Name	Register Number	Usage
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	Function return values
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved variables
`$t8-$t9`	24-25	more temporaries
`$k0-$k1`	26-27	OS temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	Function return address

Operands: Registers

- **Registers:**
  - `$` before name
  - Example: `$0`, "register zero", "dollar zero"
- Registers used for **specific purposes**:
  - `$0` always holds the **constant value 0**.
  - the ***saved registers***, `$s0`–`$s7`, used to hold variables
  - the ***temporary registers***, `$t0` - `$t9`, used to hold intermediate values during a larger computation
  - Discuss others later

Instructions with Registers

- Revisit add instruction

C Code

```
a = b + c
```

MIPS assembly code

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

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

Word-Addressable Memory

- Each 32-bit data word has a unique address

Word Address	Data	
⋮	⋮	⋮
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

Note: MIPS uses **byte-addressable** memory, which we'll talk about next.

Reading Word-Addressable Memory

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

```
lw $s0, 5($t1)
```

- **Address calculation:**
  - add *base address* (`$t1`) to the *offset* (5)
  - address = (`$t1` + 5)
- **Result:**
  - `$s0` holds the value at address (`$t1` + 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($0)   # read memory word 1 into $s3
```

Word Address	Data	
⋮	⋮	⋮
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