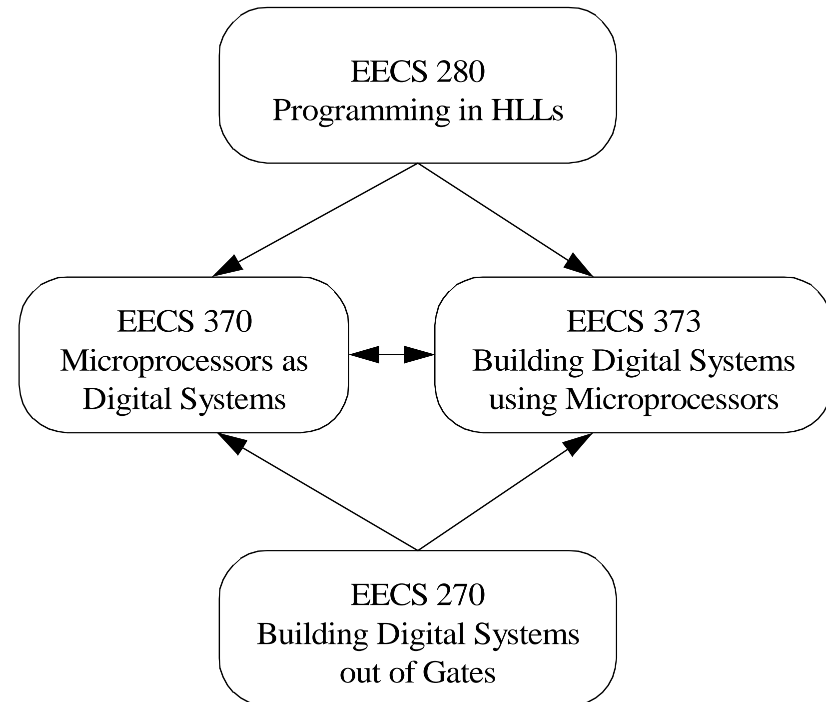
EECS 373

Design of Microprocessor-based Systems

Fall 1998 Course Notes

Prof. Steven K. Reinhardt

Where does this course fit in?



© 1998 Steven K. Reinhardt

Overview

What is this course about?

- how to design and build systems using microprocessors

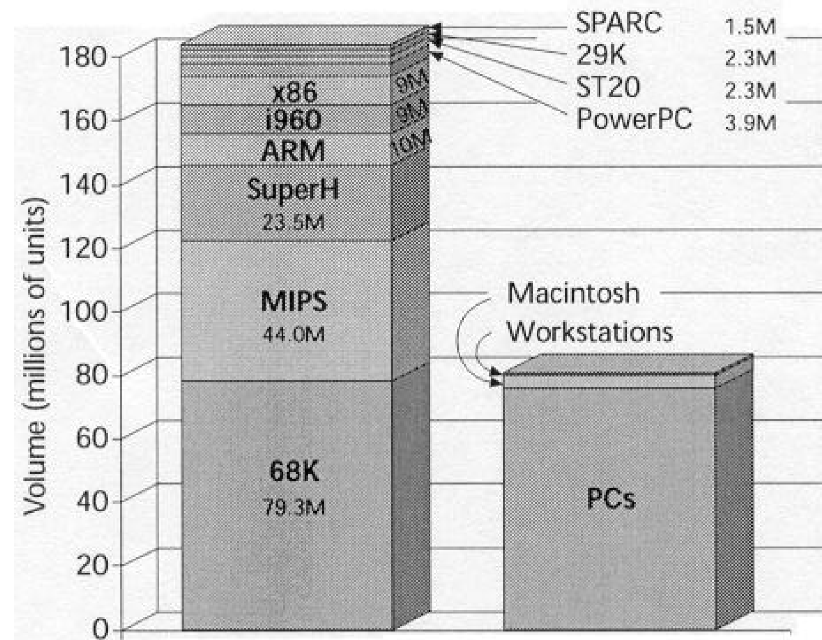
What are some examples of microprocessor-based systems?

Why are more and more systems using microprocessors?

- microprocessors are very cheap (< $1 at the low end)
- an off-the-shelf microprocessor + software can replace a lot of application-specific logic
- software enables flexible, sophisticated features that would be difficult or impossible otherwise
- software is typically easier to debug & fix than hardware
- more and more information is being stored and transmitted in digital form

Embedded vs. general-purpose systems

A microprocessor-based system that is dedicated to a specific task or tasks as part of a larger product is referred to as an *embedded system*. Although embedded systems get much less publicity, they vastly outnumber general-purpose systems in the market.

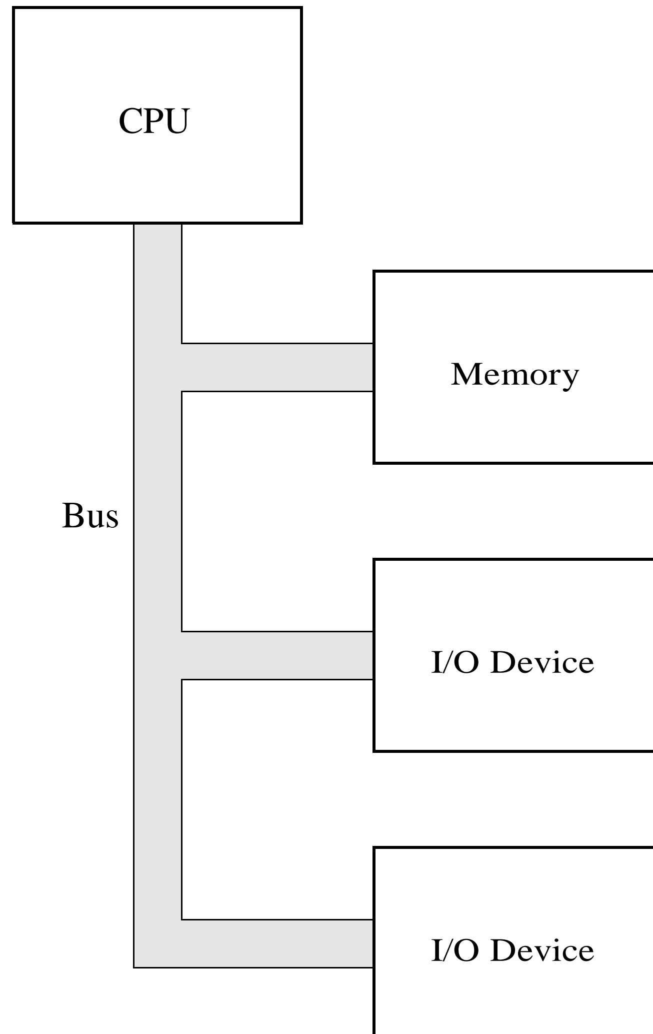


Source: *Microprocessor Report*

This chart shows the number of 32-bit microprocessors shipped in 1997. On the left are processors shipped in embedded systems; on the right are processors shipped in general-purpose systems. Note that this chart does not include hundreds of millions of 8-bit and 16-bit embedded processors.

 © 1998 Steven K. Reinhardt

Generic microprocessor-based system



CPU

The central processing unit (CPU) is the core of the processor where instructions get executed.

- *Registers* are storage locations within the CPU. PowerPC processors have 32 general-purpose registers (called r0, r1, ... r31) plus a handful of special-purpose registers.
- *Function units* are pieces of mostly combinational logic that manipulate data values. The most common function unit is the arithmetic logic unit (ALU), which typically adds, subtracts, and does boolean operations on integer values.
- *Control logic* repeatedly fetches the next instruction from memory, decodes it, and executes it. Typical instruction: read two values from registers, send them to a function unit to produce a new value, write the new value to a register.

The CPU executes instructions encoded in *machine language*, which maps every instruction to a binary value. We will primarily use *assembly language* in this class, which is a human-readable encoding of the same set of instructions.

high-level language: a = b + c
assembly language: add r1, r2, r3
machine language: 01111100001000100001101000010100

Some instructions read or write (get values from or send values to) memory locations or I/O devices. For PowerPC processors, these are the *load* and *store* instructions respectively. These transfers occur over the system bus.

 © 1998 Steven K. Reinhardt

Busses

A *bus* is a group of signals (wires) that communicates among several devices.

A typical system bus is composed of three smaller busses:

1. address
2. data
3. control

Address & data busses each carry a single value, i.e., an *n*-wire bus carries values in the range 0 ... 2^n-1

Control bus is more a collection of independent signals: read/write, address valid/not valid, data valid/not valid, etc.

Bus width (e.g. 32-bit bus, 64-bit bus) refers to the number of *data* wires; not necessarily the same as CPU "width" (size of internal CPU registers).

- To read a memory location or I/O device, CPU puts an address on the address bus; the appropriate memory module or device sends data back via the data bus.
- To write a memory location or I/O device, CPU puts an address on the address bus and data on the data bus; the appropriate memory module or device takes the data the data bus.

Memory

- Stores instructions (programs) & data
- Several types (read-only, non-volatile, etc.)
- Organized as a linear array of locations (addresses), each storing one byte (8 bits).
- Typically accessed one, two, four, or sometimes eight bytes at a time (8, 16, 32, or 64 bits). These are generally called *byte*, *halfword*, *word*, and *doubleword* accesses.
- Fundamental properties:
  1. Read returns value of last write
  2. Accesses have no side effects

 © 1998 Steven K. Reinhardt

I/O devices

- Let system interact with the world
- *Device interface* (a.k.a. *controller* or *adapter*) is digital logic that connects device to bus
- Often loosely say "device" to mean device + interface
- examples?

- *I/O device registers* look a lot like memory to the CPU: bunch of locations that can be read from & written to
- Key difference: I/O device registers are connected to other things (external wires, device control logic, etc.)
- Result:
  1. reads usually don't return last value written (e.g., may be value of last key pressed on keyboard instead)
  2. writes usually have side effects (e.g., display character on screen)
- Hardware must decode bus control & address signals to make I/O devices and memory modules do the right thing

Course topics (not in order)

- PowerPC architecture & assembly language programming
- Bus designs, protocols, and interfacing
- Standard busses (PCI, USB, etc) & bus bridging
- Common I/O devices:
  - Timers
  - Analog-to-digital, digital-to-analog conversion
  - Serial I/Ol
  - Video
- Interrupts: I/O devices demand attention
- DMA: I/O devices talk directly to memory
- Memory technologies: SRAM, DRAM (page mode, EDO, synchronous, Rambus), Flash, etc.
- FPGAs (lab implementation technology)

 © 1998 Steven K. Reinhardt