Architecture of Digital Systems II

1 Introduction
Designing Embedded Systems

Embedded Computer Systems

Embedding computers into applications is not a new idea.

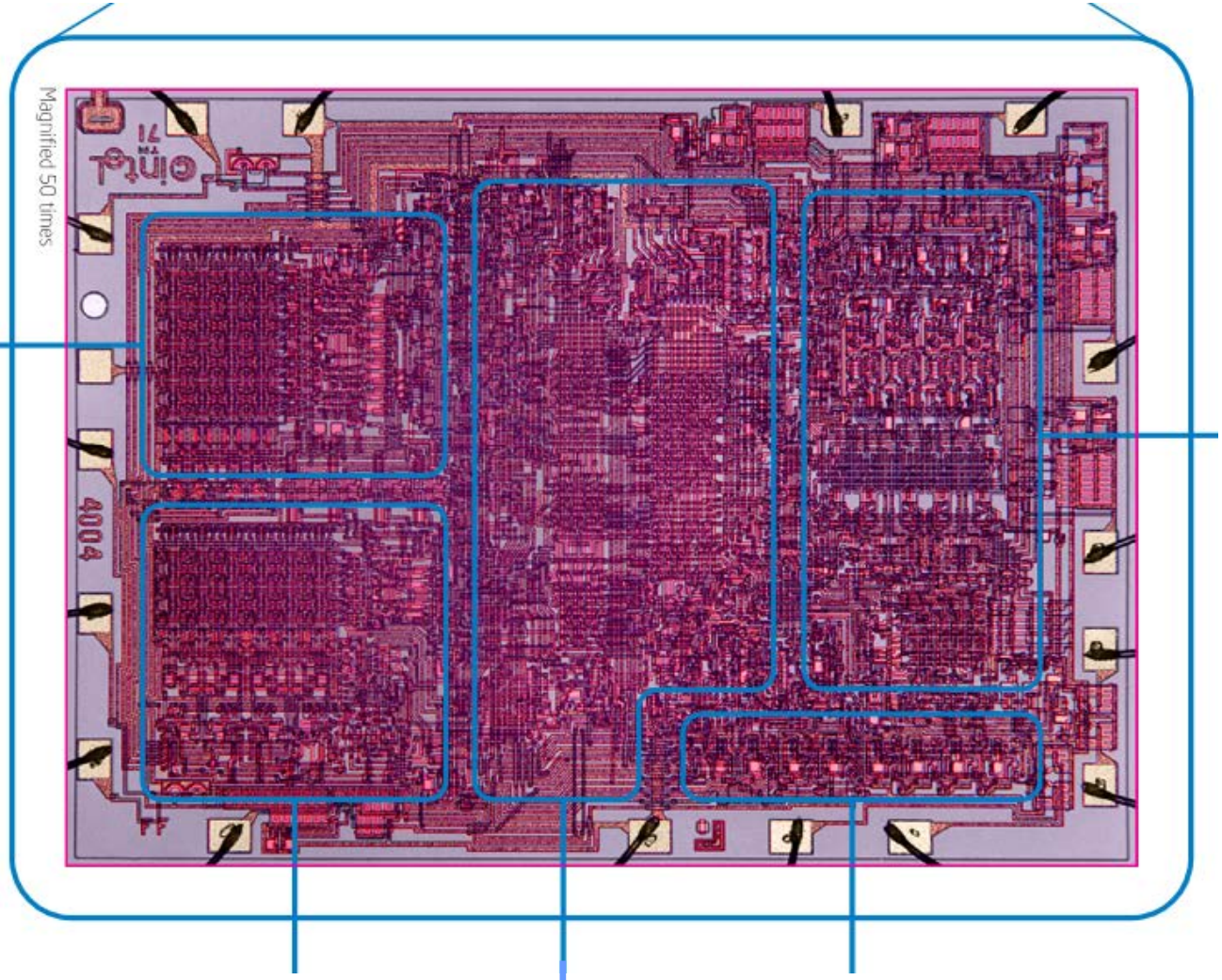
Examples:

- 1949: Whirlwind (designed at MIT),
  - a computer to control an aircraft simulator,
  - supported real-time operation.

- 1971: Intel 4004, the first microprocessor
  - (A microprocessor is a CPU on a single chip.)
  - The 4004 was designed for a desktop calculator.

© Intel

Intel 4004 explained



index registers

arithmetic unit

stack and program counter

instruction decoder

8-phase clock generator

Automotive Embedded Systems

After the microprocessor was available, automobile makers started using it almost immediately:

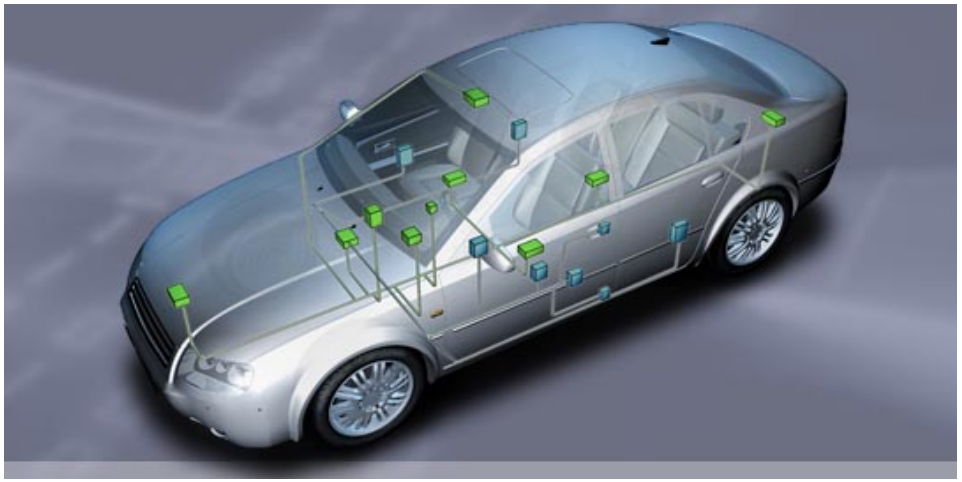
- device is programmable → hardware designs can be re-used
- large production volumes → low hardware cost
- complex control tasks can be solved
  (engine control, emission control)

Today, electronics in cars constitute 30% - 40% of the overall production cost!

Automotive Embedded Systems

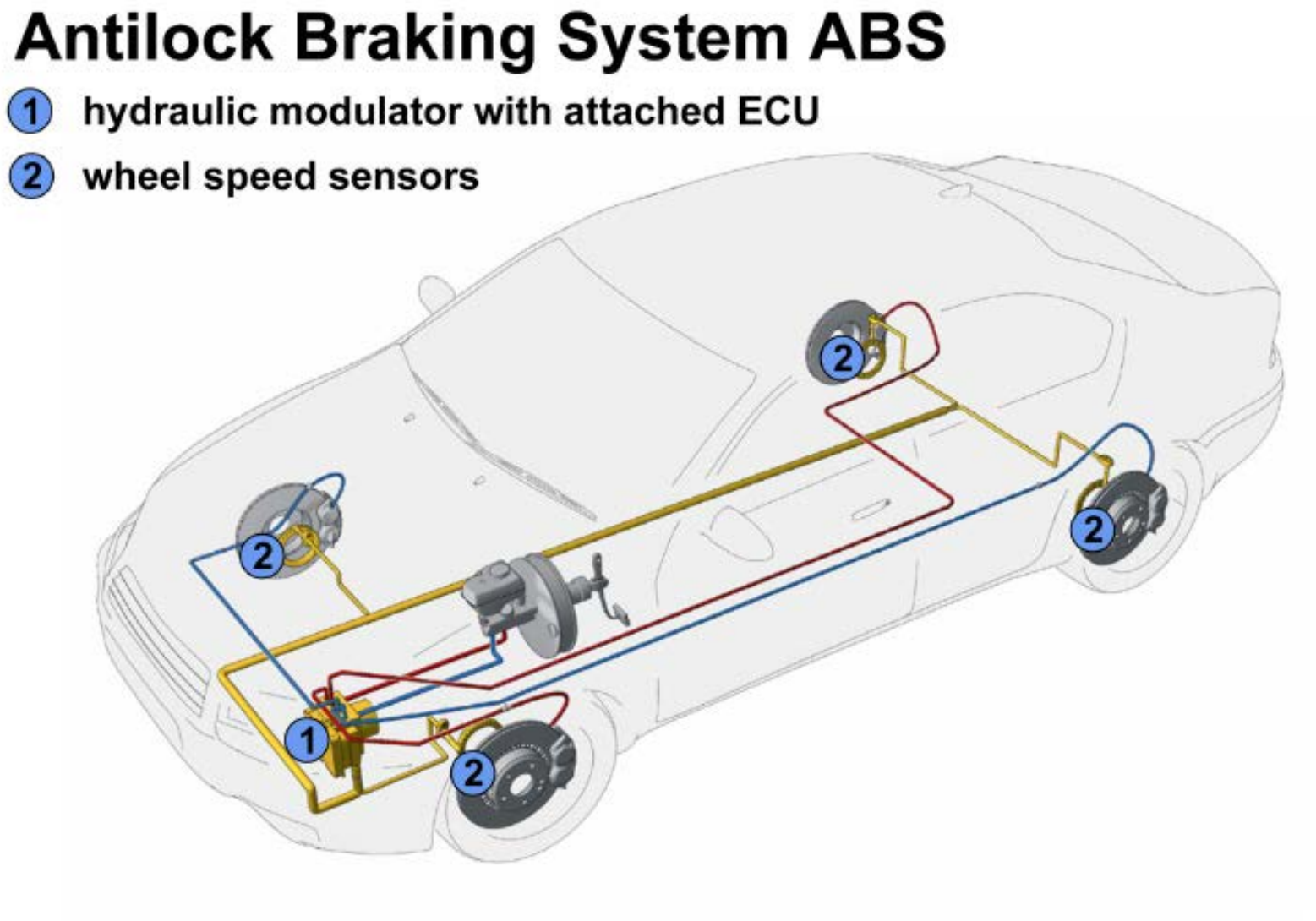
In a modern car, there can be up to 100 ECUs (Electronic Control Units), implementing a large variety of functions. Examples:

- Engine management
- Emission control
- Anti-lock Brake System (ABS)
- Electronic Stability Program (ESP)
- Airbag
- Air conditioning
- Navigation
- Communication
- Entertainment systems

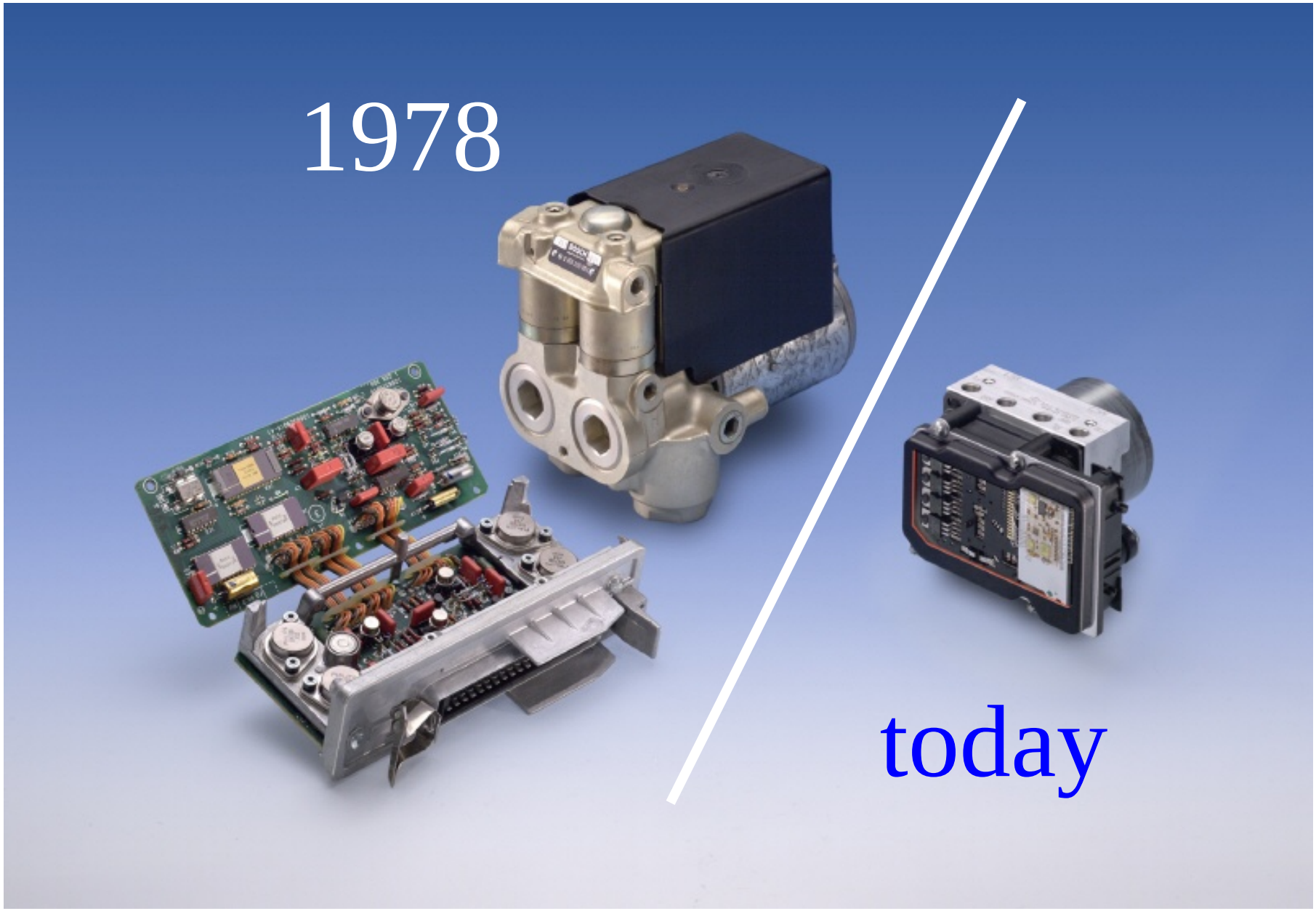


Many of these systems need to communicate with each other over on-vehicle networks.

Automotive example



Automotive Example: ABS

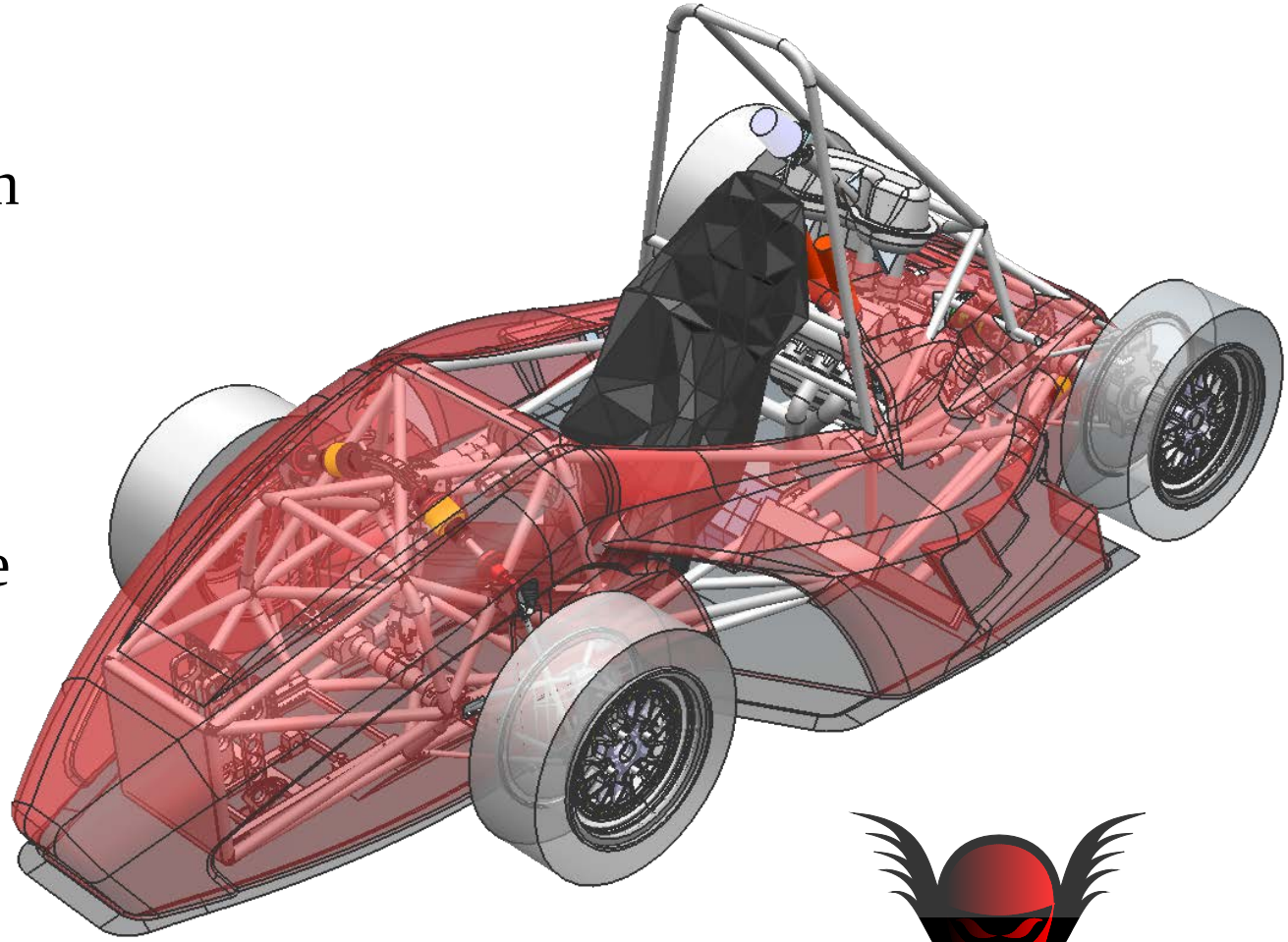


© Robert Bosch GmbH

Automotive Example – KaRaT (karat-racing.de)

Built-in sensors:

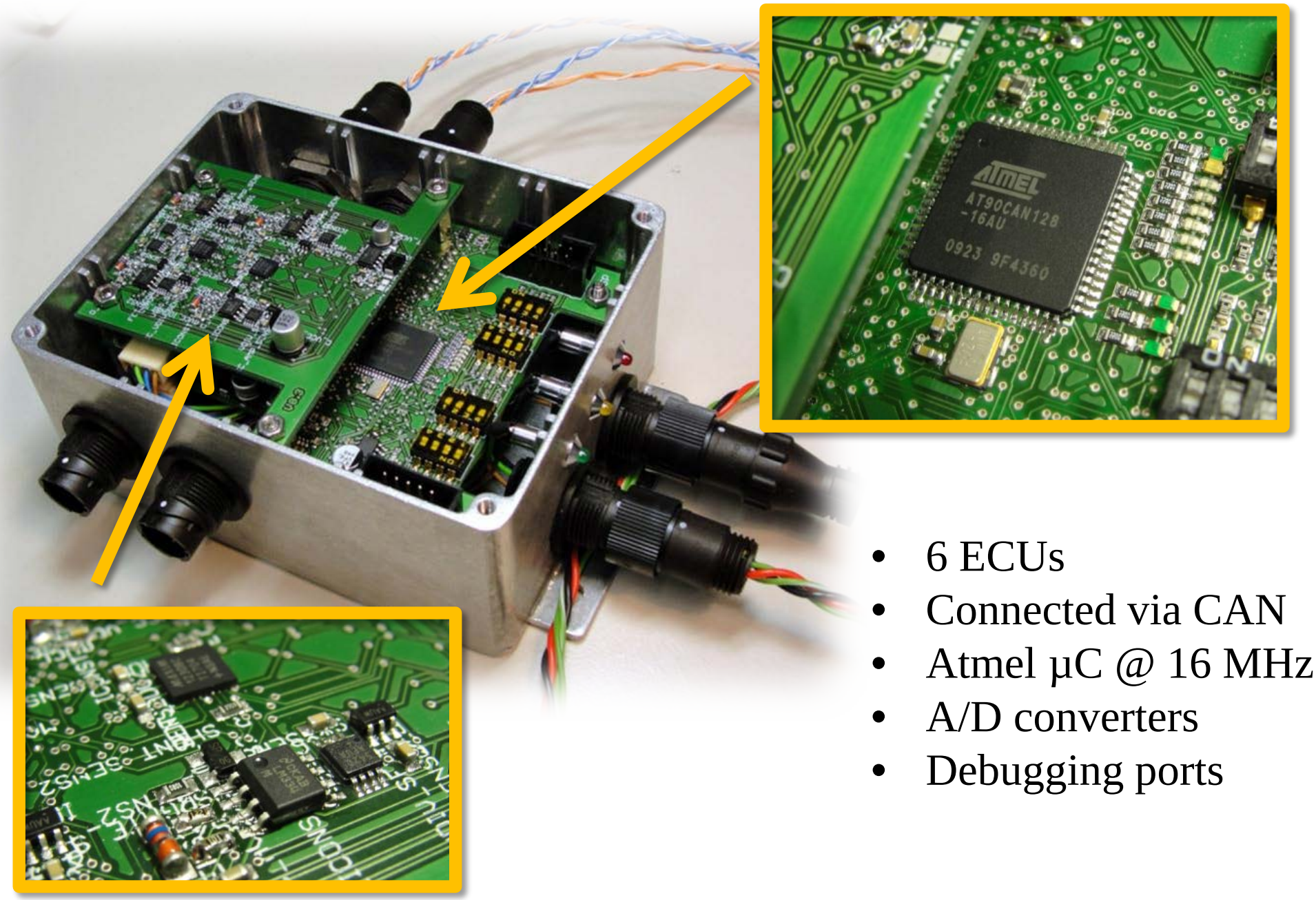
- 4x wheel rotation
- 4x spring deflection
- Steer angle
- Acceleration
- Gyration
- GPS location
- Engine temperature
- Driver's heart rate



Controlled by several ECUs
Connected via 1 Mbit/s CAN

KaRaT
KAISERSLAUTERN RACING TEAM e.V.

Automotive Example – KaRaT ECU (karat-racing.de)



- 6 ECUs
- Connected via CAN
- Atmel µC @ 16 MHz
- A/D converters
- Debugging ports

Embedded Systems Are Virtually Everywhere

- Household appliances
- Telecommunications/networking
- Consumer electronics (DVD players, MP3 players, mobile phones, video games, cameras, internet access points, toys)
- Building control (heating, A/C)
- Avionics ("fly-by-wire")
- Space (spacecraft, satellites)
- Industrial automation (production control, robots)
- Military
- Medical care
- ...
- “Internet of Things”

Characteristics of Embedded Systems

Embedded computing systems have to fulfill demands different from those of general-purpose computing:

- specialized functionality and user interfaces, performance
- real-time constraints
- safety constraints
- power constraints
- cost constraints
- others

Real-Time Constraints (1)

Many embedded computing systems need to perform in real time. The correct output must be produced by a certain deadline, otherwise the system does not function properly:

Hard real-time constraints:
if violated, the system fails completely.

Soft real-time constraints:
if violated, the system does not fail completely, but performance is degraded.

Examples:

- SRS (airbag) must react within a few milliseconds after a crash (hard real-time constraint)
- video decoder must provide frames in time, otherwise presentation "stutters" (soft real-time constraint)

Real-Time Constraints (2)

Some systems have several real-time activities going on simultaneously, with repeated deadlines at various rates (multirate behavior). Example:

- multimedia applications (both video and audio streams)

The requirements for the design of an embedded system may contain

- maximum latency
  (maximum time until system responds)
- minimum throughput
  (minimum amount of data processed per second)

In many applications, real-time constraints are safety-critical!

Safety-Critical Systems

Many embedded applications directly influence the safety of the systems and of the people using them (for example, avionic systems, automotive systems, traffic control, bio/medical systems). Ensuring the safety of such systems significantly affects design.

Design considerations for safety-critical systems:

Design correctness

- Verification of design hardware (simulation, formal tools)
- System simulation (HW/SW)
- Software testing

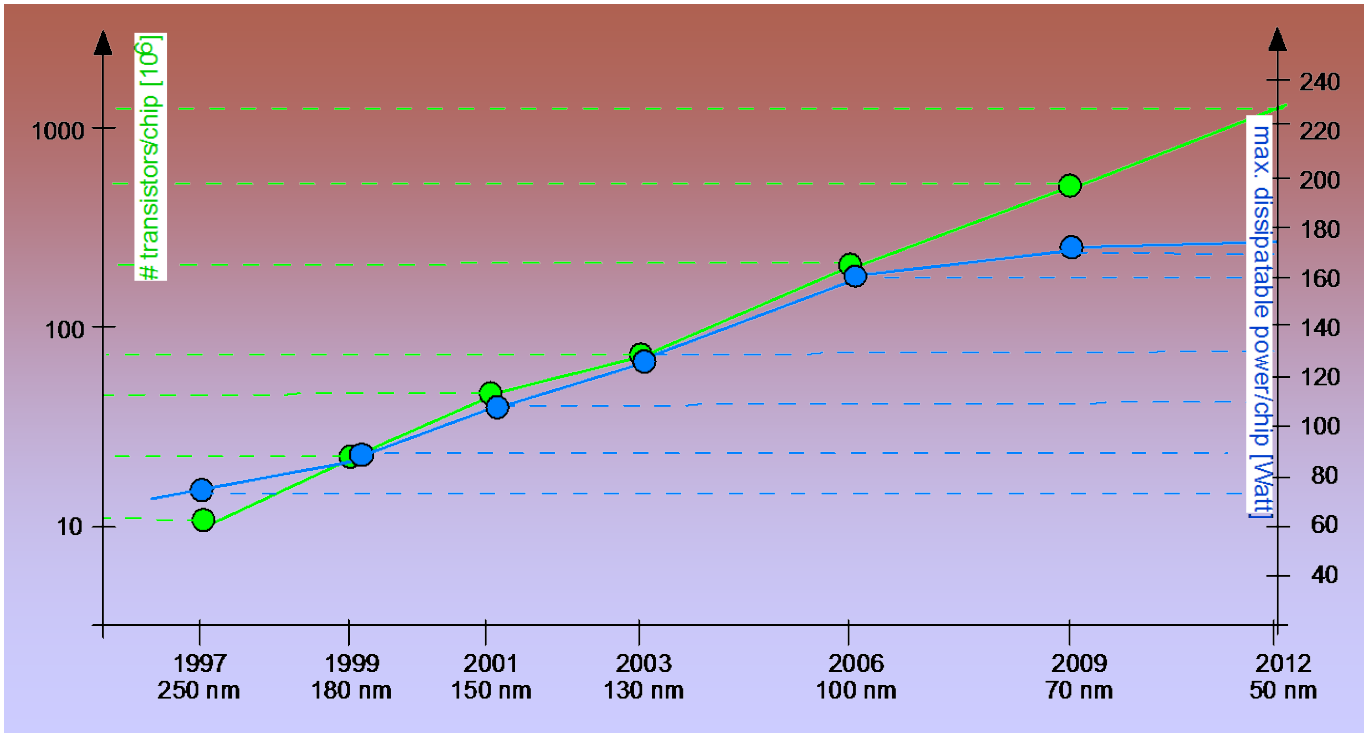
Fault tolerance/reliability/robustness

- fault-tolerant design (redundancy)
- built-in self-test
- handling of exceptions
- graceful degradation

Power Constraints

Mobile applications (hand-held devices, IoT devices, etc.) demand an efficient usage of the available energy.

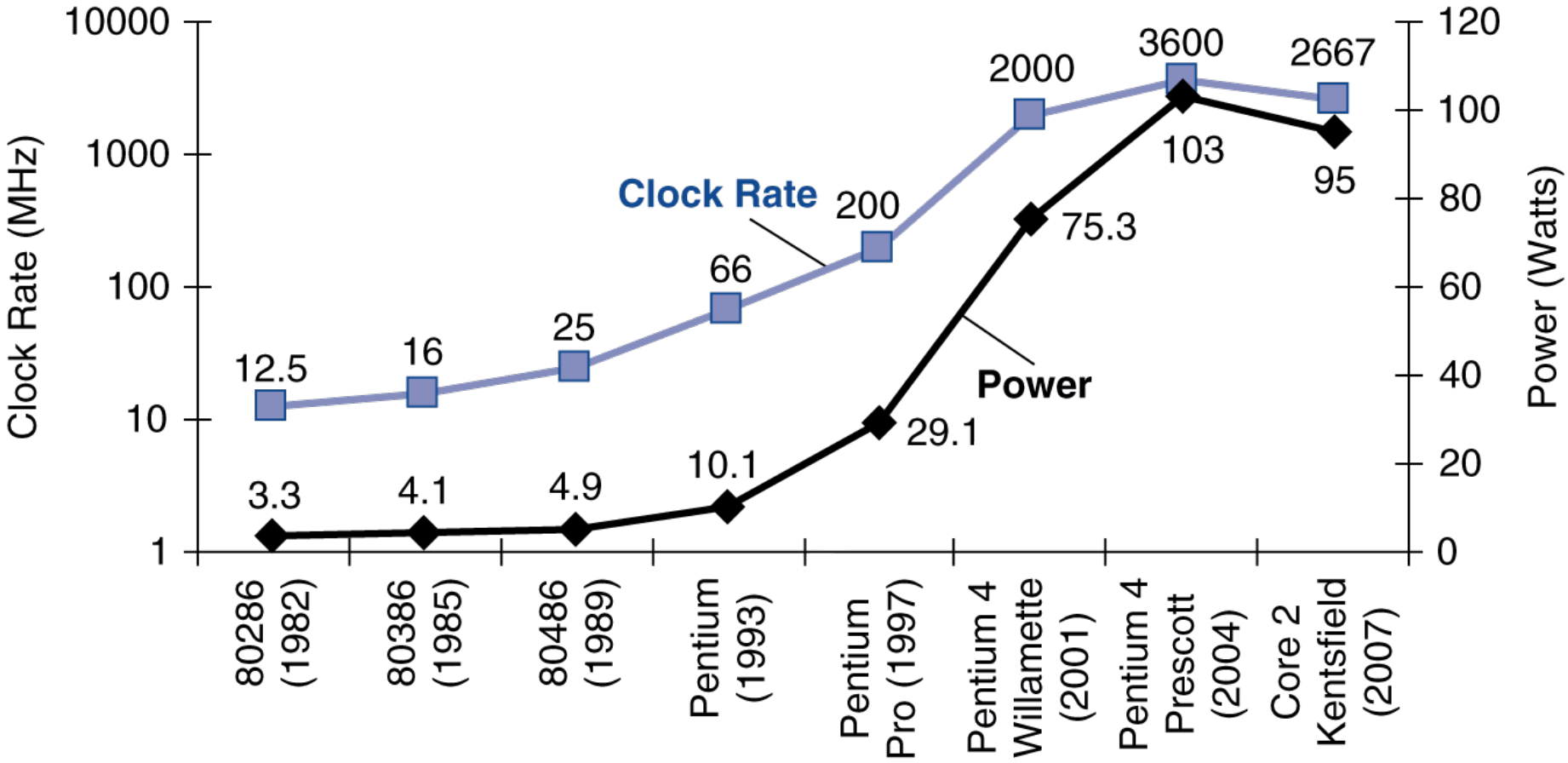
But also when power supply is not an issue, the heat dissipation can be an important constraint (as is in general-purpose computing). The increasing transistor density is in conflict with the available heat dissipation.



Source: SIA roadmap

The “Power Wall“

Source: [2] Patterson/Hennessy: *Computer Organization and Design - The HW/SW Interface*



Clock rate and power consumption of microprocessors are correlated. Maximum power consumption is limited by practical heat dissipation.

Cost Constraints

The total cost of making a product is one important factor for success in the market. The individual components of cost are affected by many factors:

Non-recurring Engineering (NRE) cost (development cost), influenced by

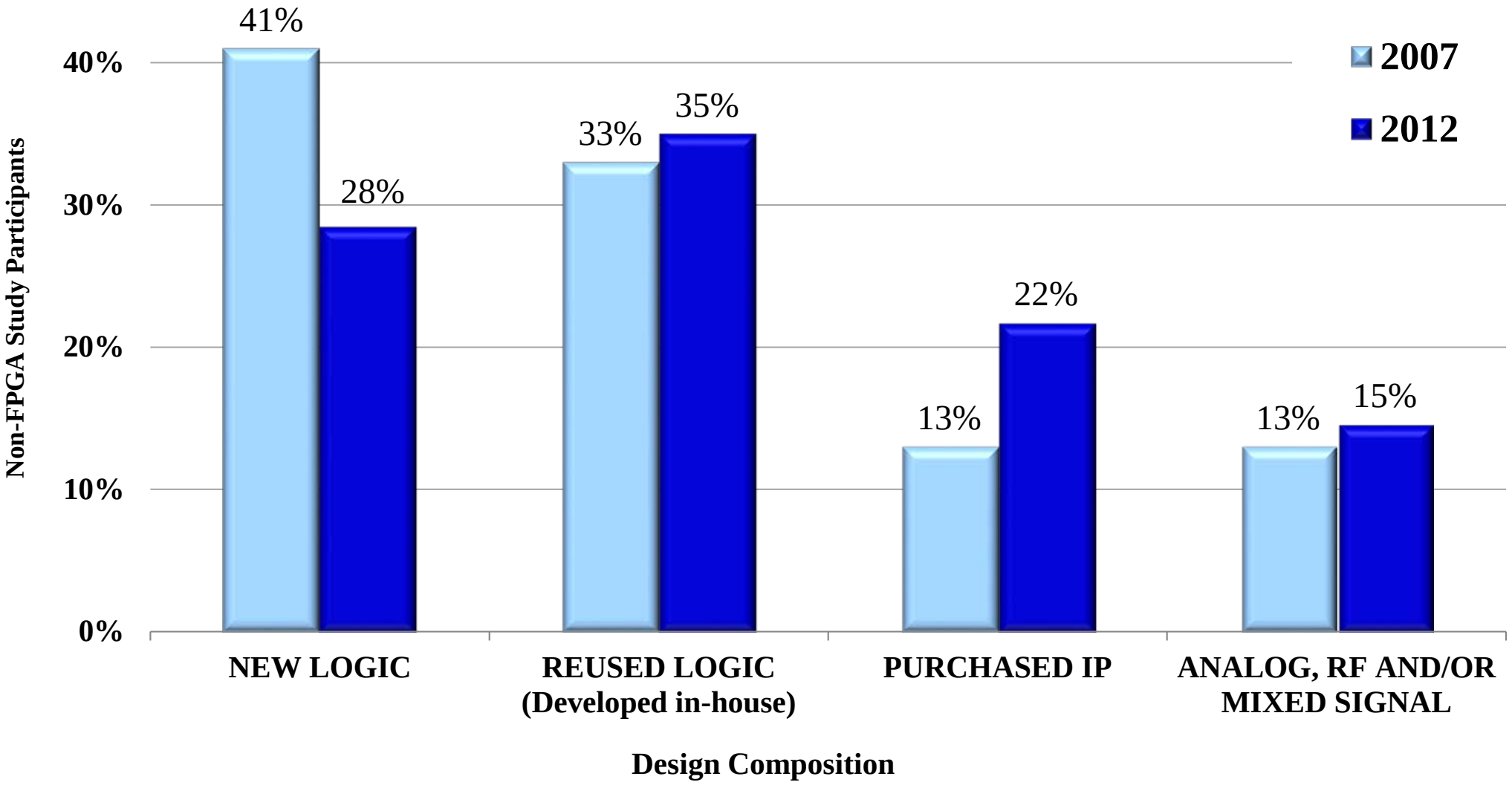
- degree of design re-use, e.g., by re-using microprocessors, software, FPGA, IP modules (IP = "intellectual property")
- design automation
- ...

Manufacturing cost, influenced by

- amount of memory, I/O, types of microprocessors used
- number of components (integration, testing)
- system size (packaging)
- ...

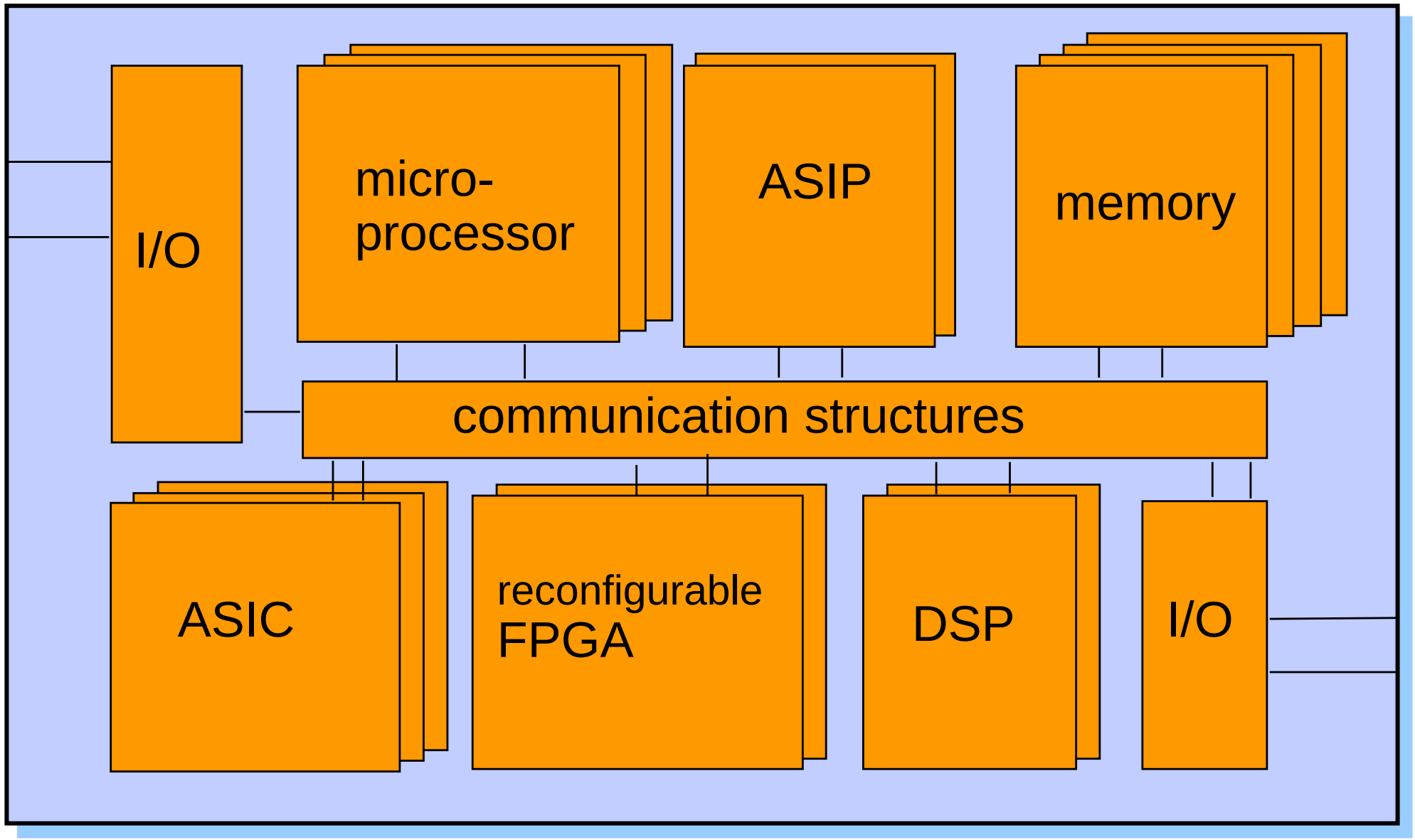
For some systems, also life-cycle costs (e.g., maintenance) may play an important role.

Lowering Design Cost: Reuse Trends



Source: Wilson Research Group and Mentor Graphics.

Hardware Components for Embedded Systems



Component Choice: Microprocessor

Reasons for using a microprocessor:

- leads to efficient implementations (area, performance)
- programmability
  - software is easier to maintain and update than hardware
  - families of products can be built to provide various feature sets at different price points
  - software can be developed independently of the hardware implementation
  - software can be re-used even if the microprocessor hardware is changed.

Component Choice: Customized Hardware

Reasons for using an Application-Specific Integrated Circuit (ASIC):

- ASICs allow to meet performance constraints that otherwise cannot be met

ASIC production is rather expensive and feasible only at higher production volumes. Also, changes in functionality require re-design of the hardware ("re-spin"). Bug fixes are expensive!

Reasons for using a Field-Programmable Gate Array (FPGA):

- allows to meet performance constraints that cannot be met with a microprocessor-based design
- can be re-configured (bugfixes possible!)

Main cost factor is development cost which may be higher than for a microprocessor-based design.

Component Choice: ASIP

An ASIP is an Application-Specific Instruction Set Processor. It is a special form of IP (Intellectual Property), i.e., a piece of RTL code in a hardware description language describing a special-purpose CPU. The code is generated by ASIP design tools. This software usually also generates compilers/assemblers and simulators along with the CPU.

Reasons for using an ASIP:

- generated hardware is optimized for performance and area
- (weak) programmability allows for (restricted) updates and bug fixes
- tool-generated hardware can be quickly re-configured and re-used.

In contrast to a microprocessor, an ASIP is often only *weakly programmable*, and only special software can be executed.

ASIPs are usually components of Systems-On-Chip that are implemented as ASICs or FPGAs.

Challenges of Embedded System Design (1)

In the process of designing an embedded system, several questions need to be answered:

How much hardware do we need?

- design space is huge!
- “too much” hardware→ higher development cost, higher production cost
- “not enough” hardware → performance constraints violated

How do we meet deadlines and performance constraints?

- invest in hardware (raises cost)
- speed up the system (may fail power constraints)

How do we minimize power consumption?

- slow the system down (may violate performance constraints)
- “low-power design”

Challenges of Embedded System Design (2)

How do we design for the future?

The hardware platform may be re-used for several product versions or generations.

- design for upgradeability
- design for re-use

How do we ensure safety, reliability and robustness?

- safety-critical systems must not fail!
- unreliable systems make for unhappy customers!
- find bugs early!

Challenges of Embedded System Design (3)

The design of embedded computing systems is more difficult than desktop computer programming:

Development is difficult

- development environments more limited
- software is usually developed on a PC or workstation and then downloaded into the embedded device

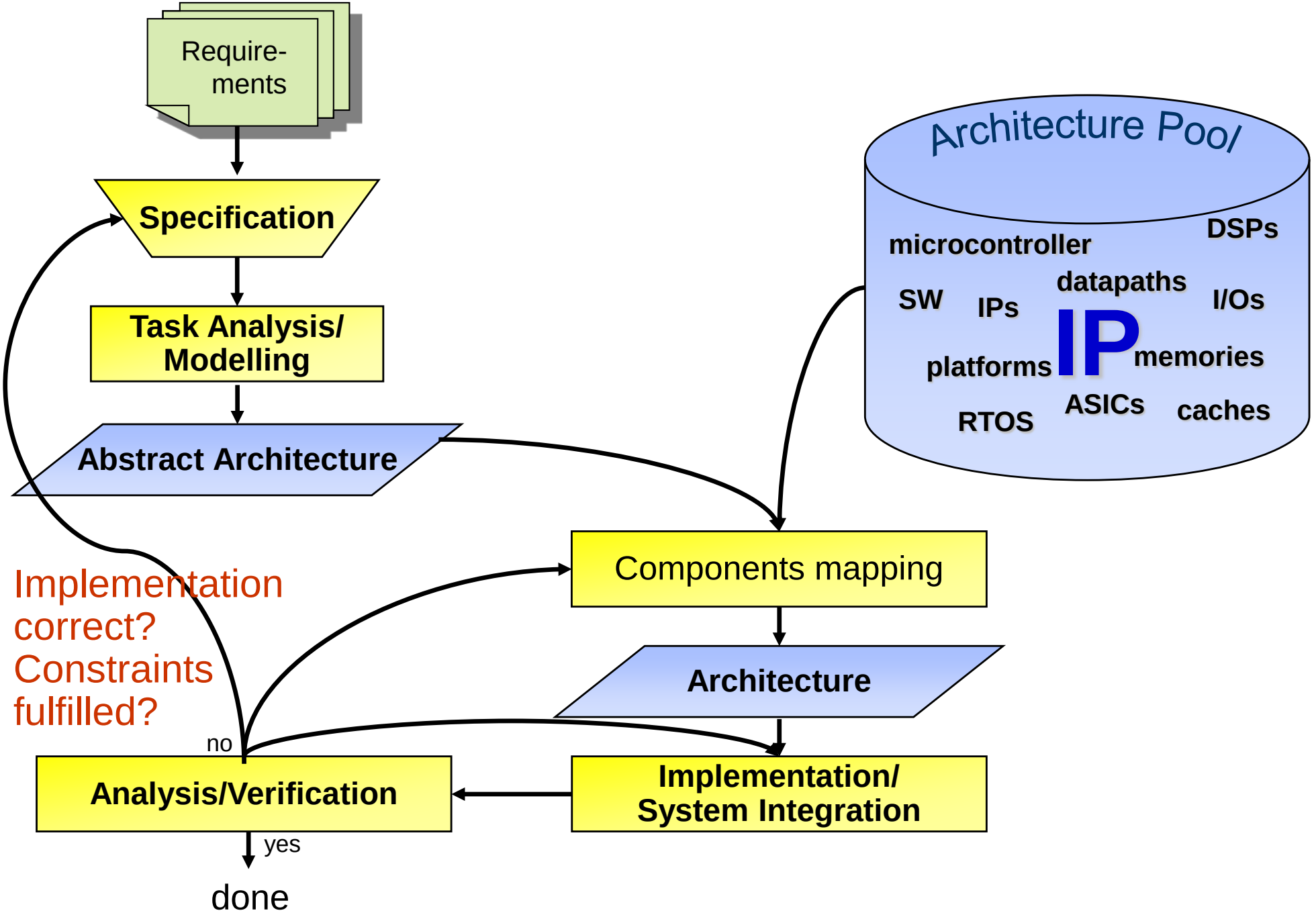
Debugging is hard

- controllability and observability is limited (often: no keyboards, no displays)
- system state must be inferred from input/output signals
- environment of the embedded system may be hard to control

Testing is complex

- the application environment may need to be exercised (e.g., a controlled machine, production line)
- timing is important (real-time constraints)

Basic Embedded System Design Flow



Design Phases (1): Requirements Definition

Before starting the design process, we need to collect the requirements for our system. Usually, we collect them as an informal description from the customers. The requirements serve as a starting point for specification.

Requirements may be functional or nonfunctional.

Functional requirements capture the intended system behavior and user interface.

Nonfunctional requirements include:

- performance ("soft" and "hard" timing constraints)
- cost (NRE cost, manufacturing cost)
- physical size and weight
- power consumption
- ...

Design Phases (2): Specification

The specification is a more precise description of the system using technical terms. It is used as a reference during the design process.

- The specification should reflect the customer's requirements.
- It must be written carefully and unambiguously to avoid problems during design.

Usually, specifications are written in informal languages as, for example, English.

There are also formal system description languages like UML, SDL or Statecharts that allow capturing intended design behavior in a formal manner.

Design Phases (3): Architecture Design (1)

The specification says *what* the system does. The architecture explains *how* the system implements those functions. Typically, the architecture is specified as a block diagram, in which blocks define functions performed and connections between the blocks define control operations and data flow.

Defining the architecture is an iterative refinement process. A first abstract architecture definition identifies the individual tasks and functions performed by the system, without giving too much implementation detail. For some blocks in the abstract architecture it may remain open whether the function is implemented in hardware or in software.

Refinement leads to the final architecture definition given in two block diagrams: one for the hardware architecture and another for the software architecture.

Design Phases (3): Architecture Design (2)

Architectural descriptions must satisfy

- functional requirements
- non-functional requirements (performance, cost, power, etc.)

Ensuring that non-functional requirements are met is difficult. On the architectural level we can only estimate the properties of the individual components and the complete system. Estimation can be based on

- experience
- simplified models
- model simulation and analysis

Accurate estimation is crucial!

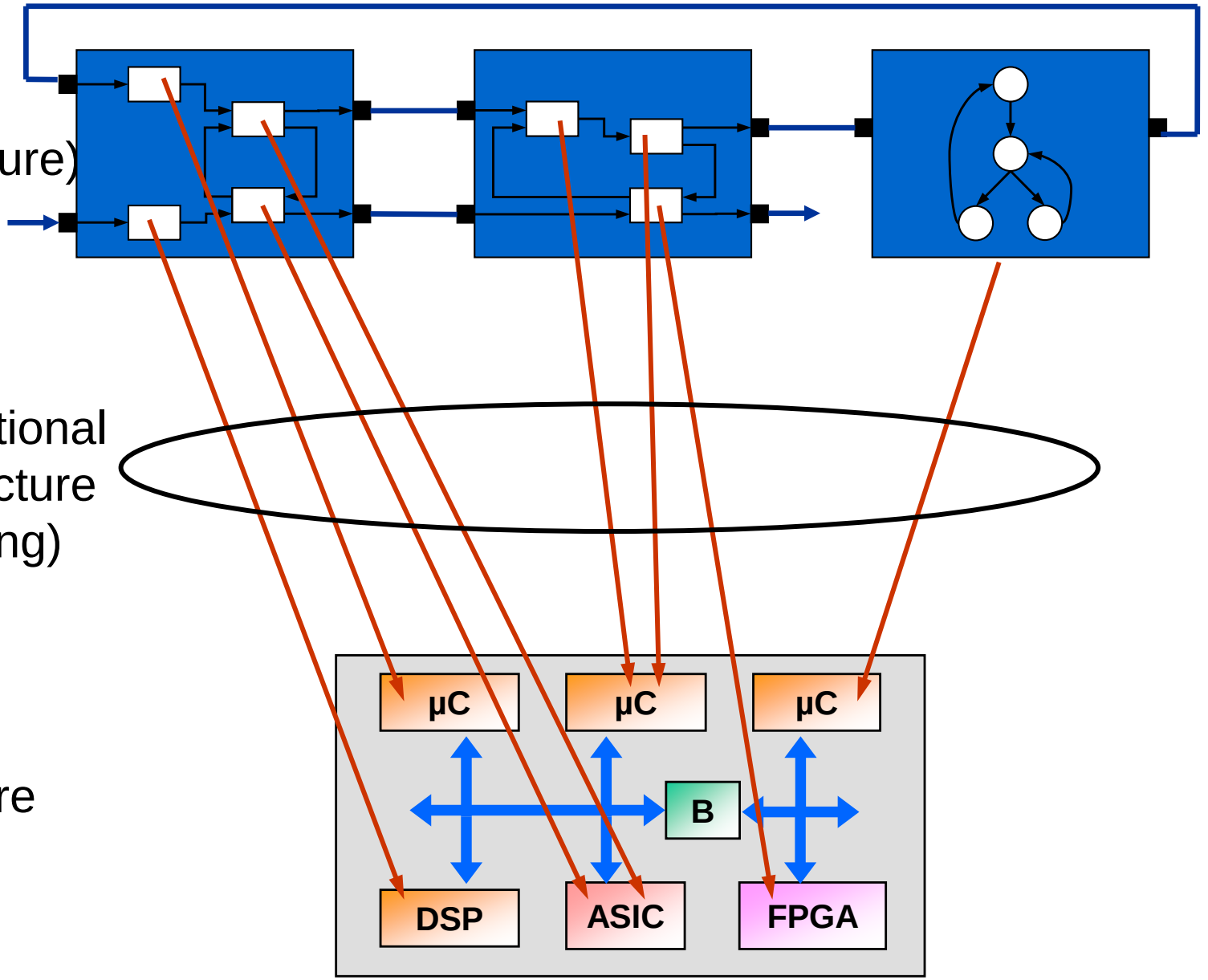
Bad design decisions due to wrongly estimated data may not show up until after implementation!

Design Phases (3): Architecture Design (3)

Functional Model
(Abstract Architecture)

Mapping of Functional
Blocks to Architecture
Blocks (Partitioning)

Target Architecture



Design Phases (4): Designing HW and SW Components

Each component block in the architecture needs to be *implemented*.

Hardware components are described on the Register-Transfer Level (RTL) using Hardware Description Languages (HDLs) like VHDL or Verilog.

Hardware components may be pre-designed (e.g., third-party products such as microprocessors, memory chips, communication modules, other IP), or may need to be custom designed.

Software components consist of libraries, software interfaces and application programs, written in higher-level programming languages or in assembler code. Often, also software components may be standard pre-designed modules, ready for use. Others you may need to design yourself.

Design Phases (5): System Integration

The final step in design is system integration: putting all components together and bringing the system to work.

System integration is difficult because often in this phase, bugs appear. Finding the cause of these bugs is a tedious and complex task because the complete system consisting of hardware and software needs to be taken into consideration.

Debugging facilities for embedded systems are limited.

Simulation of a detailed model may be impossible due to system complexity.

It is much more effective and efficient to find bugs already during the design of the individual modules. Verification and analysis should therefore be carried out already during the component implementation phase (module-level verification).

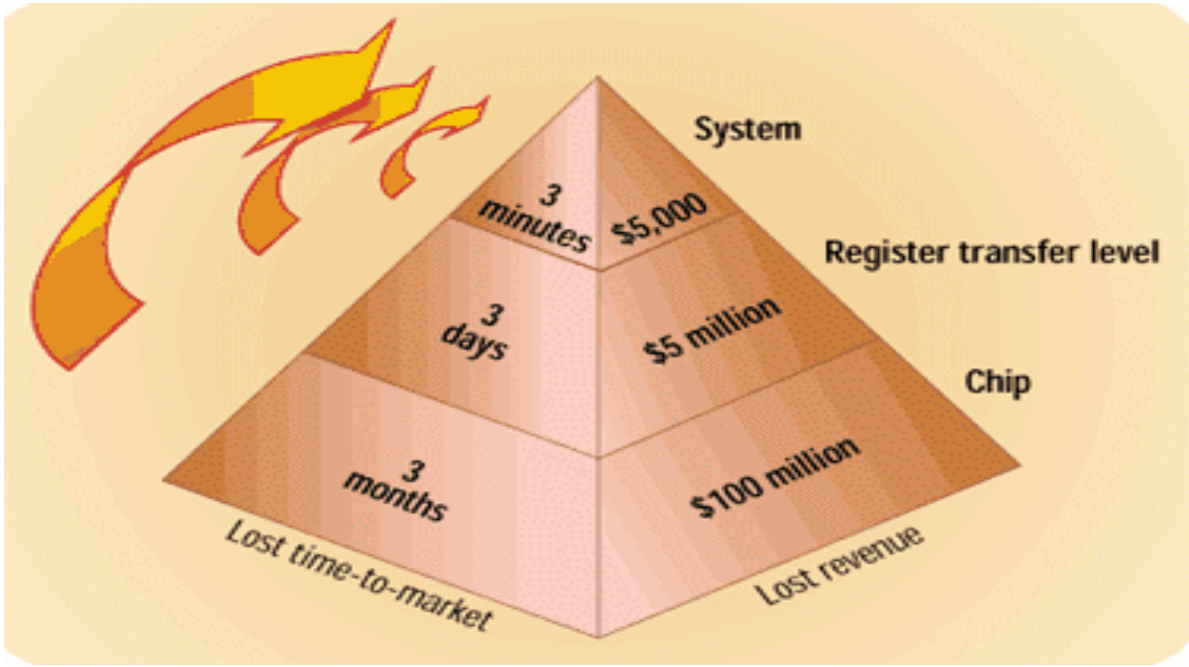
System Analysis and Verification

Making errors in architecture design can be very expensive.

The same holds for errors made during component design or system integration.

The earlier an error is found the better.

Re-spin cost per design phase



Source: Integrated Communications Design May, 2001

System Analysis, Validation and Verification of the system's functional and non-functional properties are important at every stage of design!

→ see course "Verification of Digital Systems"

Embedded System Design – Tool Chain

Today, an efficient design flow integrates a number of design tools:

- *architectural-level simulators* for exploring the design space and for making architectural choices,
- *virtual prototyping systems* for early development of the software and for testing it with virtual hardware models
- *integrated development environments* for hardware development
- *synthesis tools* for generating the hardware description,
- *simulators* and *estimators* to validate and verify the implementation,
- *formal verification tools* for verifying implementation correctness and standards compliance,
- *compilers/assemblers/linkers* for generating machine code,
- *profilers* for gaining statistical information about the software,
- *debuggers* for fixing bugs in software and hardware,
- *analysis tools* for checking timing and other constraints,
- ...