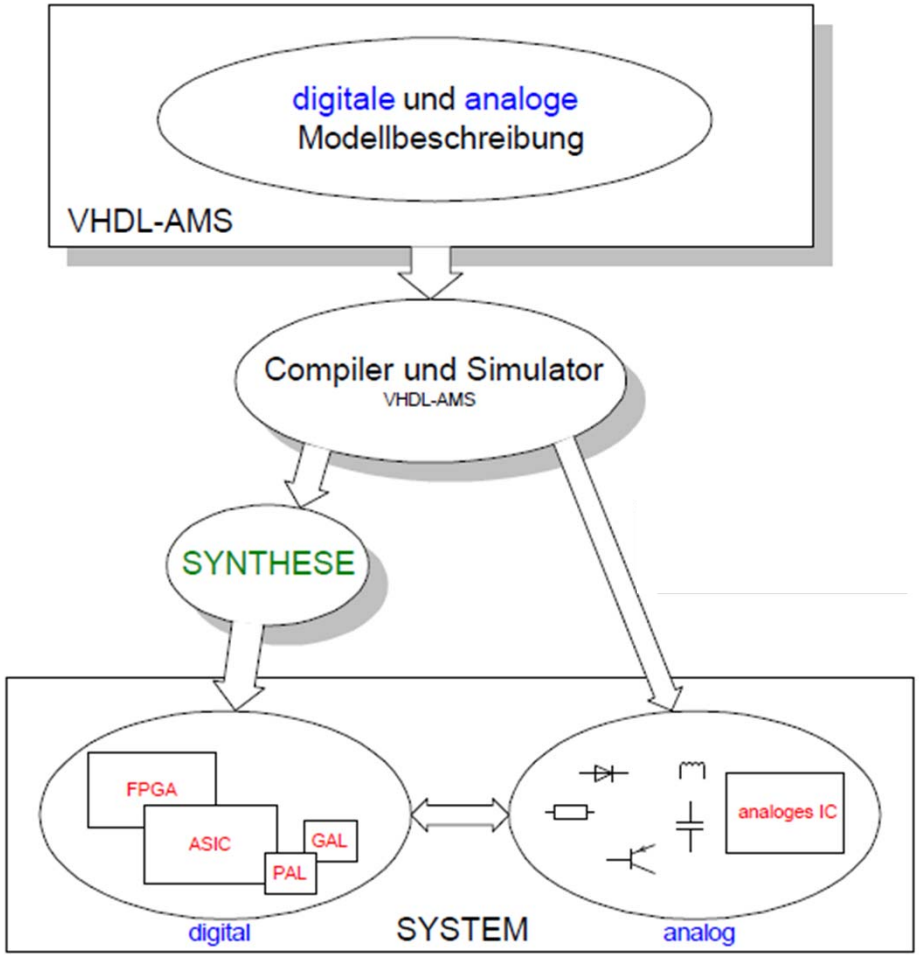
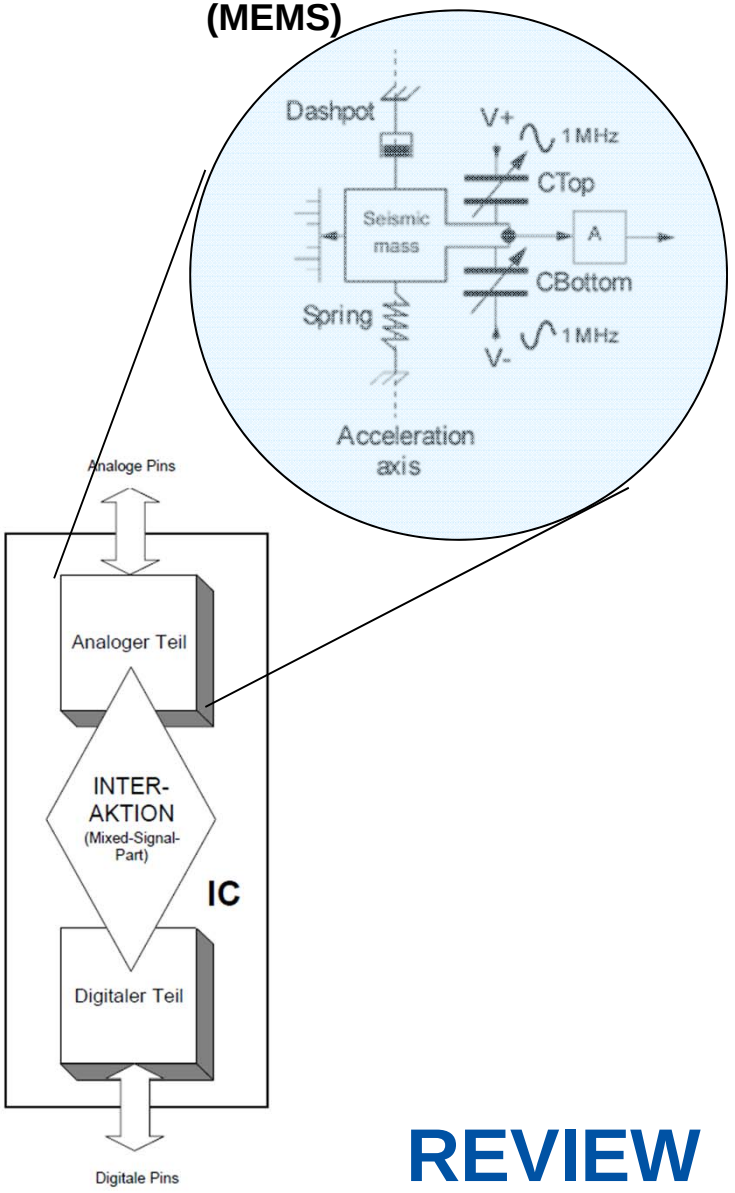
Embedded Systems

The world isn‘t digital



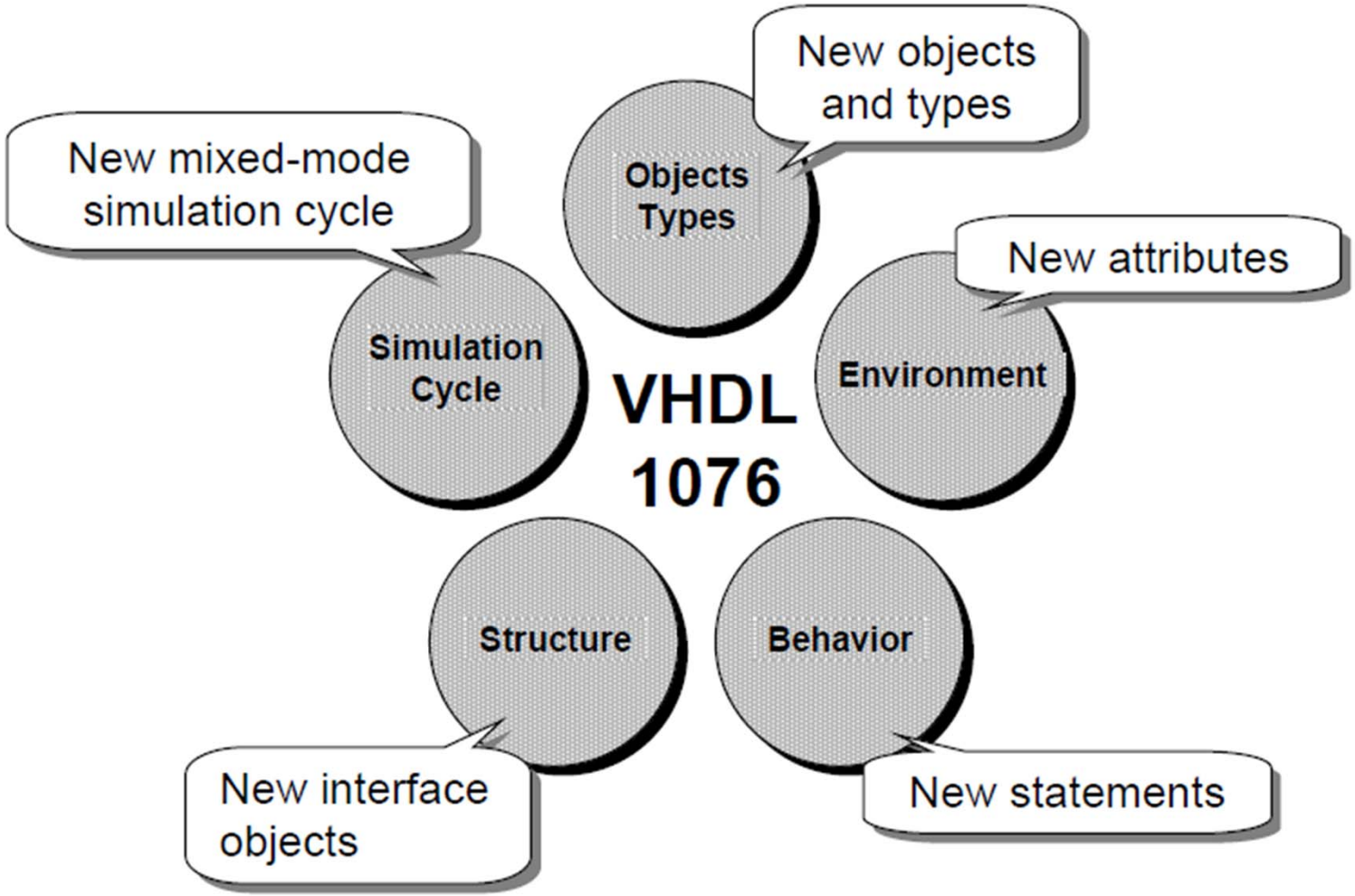
Micro-Electro-Mechanical Systems (MEMS)



Mixed-Signal-IC

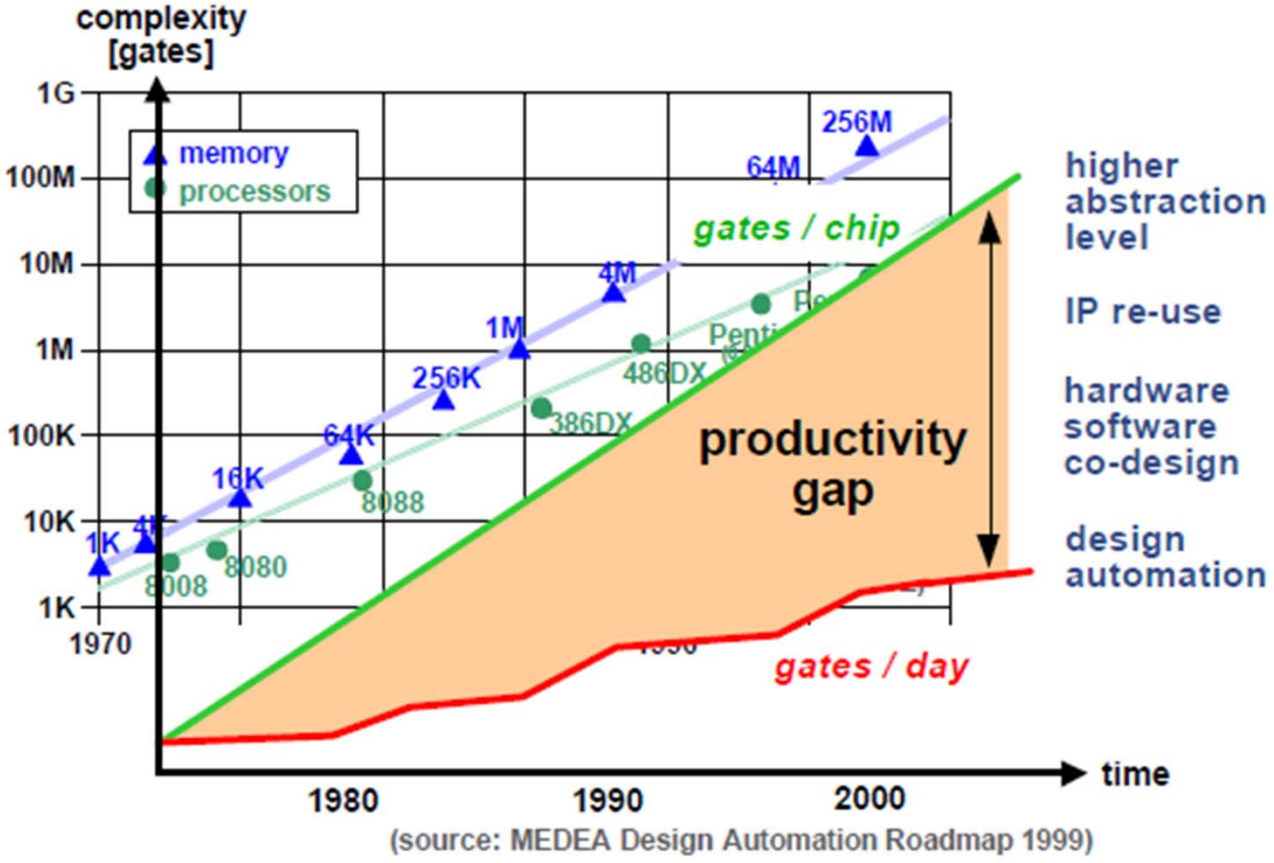
REVIEW

VHDL-AMS Language Architecture




SYSTEMC™

Productivity Gap



(source: MEDEA Design Automation Roadmap 1999)

REVIEW

Introduction

The followings are not supported by native C/C++

- Hardware style communication
  - Signals, protocols, etc.
- Notion of time
  - Cycle/clock, delay, time sequenced op., etc.
- Concurrency
  - HW operates in parallel.
- Reactivity
  - HW responds to stimuli.
- Hardware data types
  - Bits, bit-vector types, multi-valued logic type, and so on.

SystemC is modeling platform consisting of a set of C++ class library, plus a simulation kernel that supports hardware modeling concepts at the system level, behavioral level and register transfer level.

SystemC features

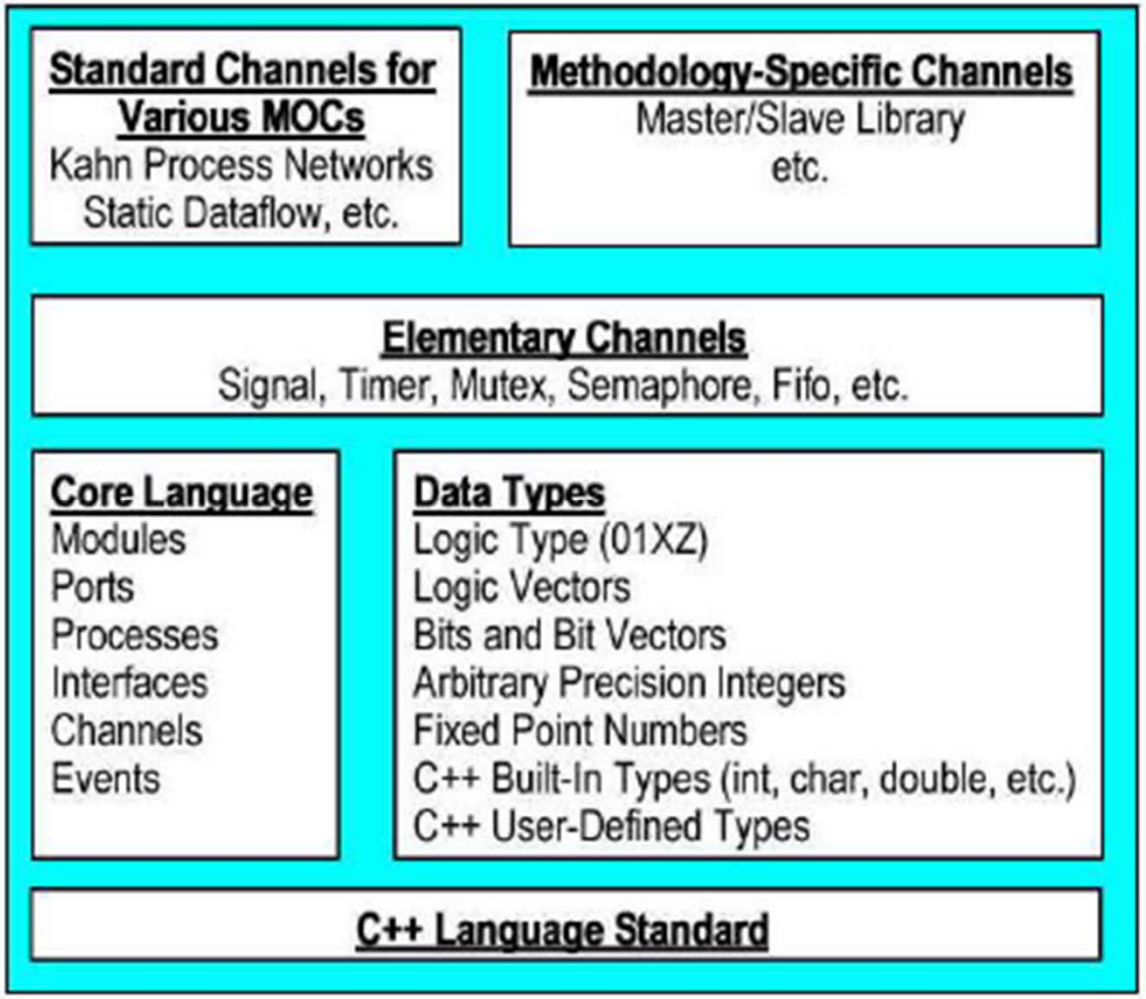
- Processes → for concurrency
- Clocks → for time
- Hardware data types → bit vectors, 4-valued logic, ….
- Waiting and watching →for reactivity
- Modules, ports, and signals →for hierarchy
- Channel, interface, and event → abstract communications

- Simulation support
- Support of multiple abstraction levels and iterative refinement

SystemC V2.0 language structure

Upper layers are built cleanly on lower layers.

Lower layers can be used without upper layers.



Comparison: SystemC - VHDL

	VHDL	SystemC
➢ Hierarchy	entity	module
➢ Connection	port	port
➢ Communication	signal	signal
➢ Functionality	process	process
➢ Test Bench		object orientation
➢ System Level		channel, interface, event abstract data types
➢ I/O	simple file I/O	C++ I/O capabilities

SystemC design flow

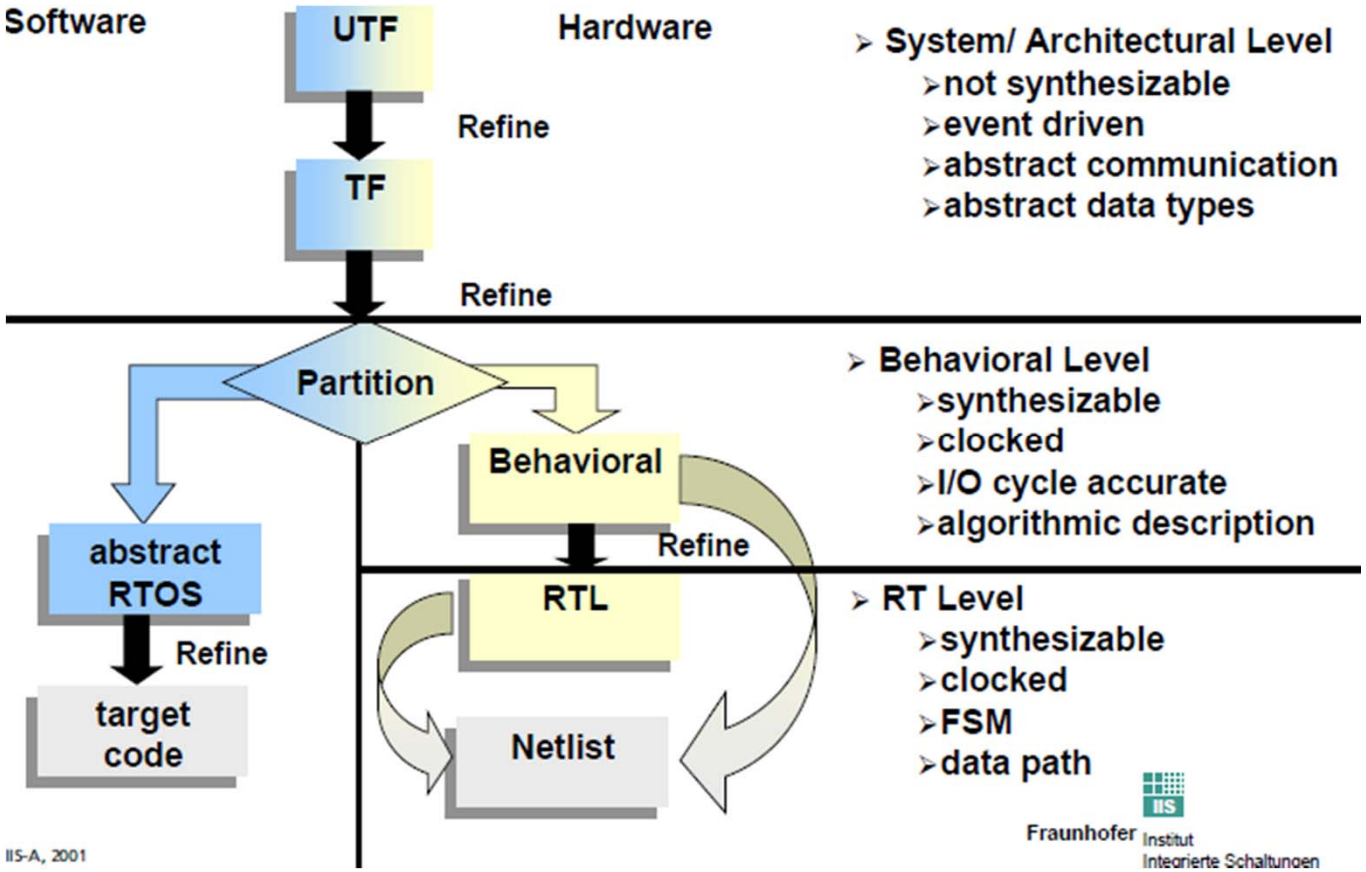
REVIEW

Untimed functional (UTF)

A network of HW/SW neutral modules executing in zero times and communicating with abstract channels.

Timed Functional (TF)

A network of modules executing in some defined times and communicating with abstract channels. Allows allocation of time to behavioral blocks (using wait(delay))



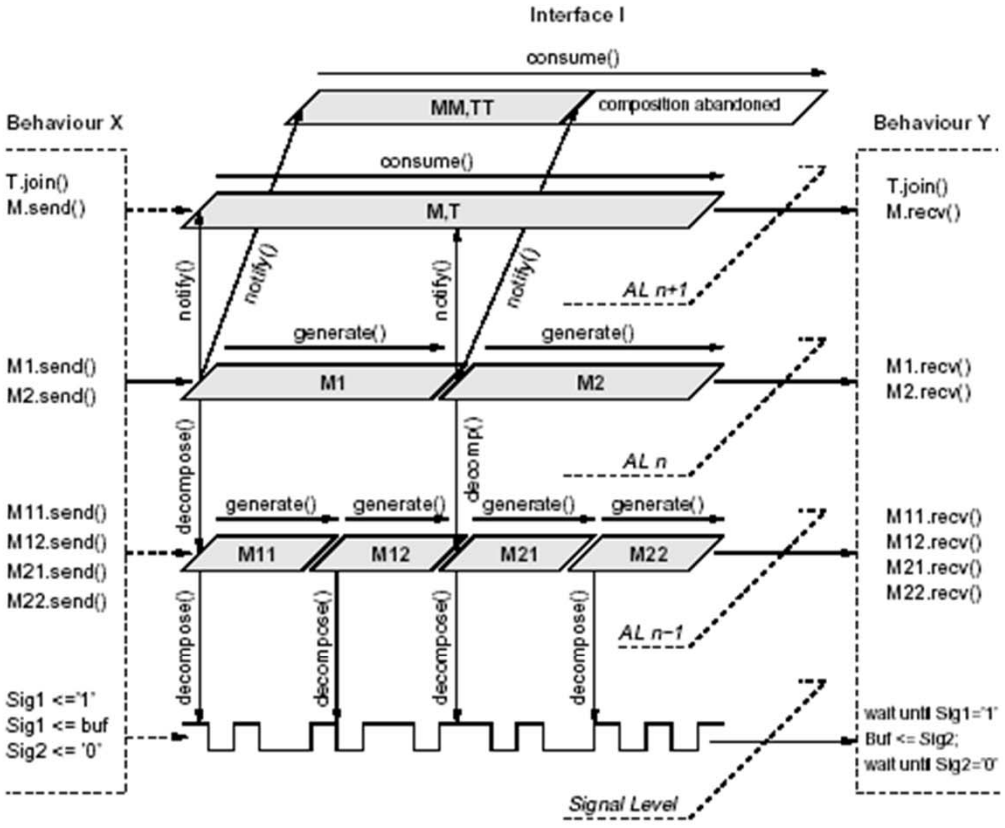
SystemC – Transaction level modeling

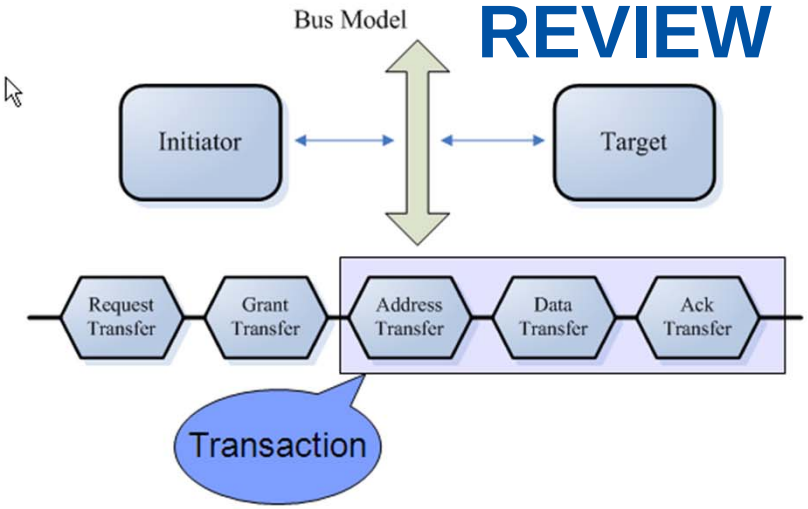
Concept of TLM

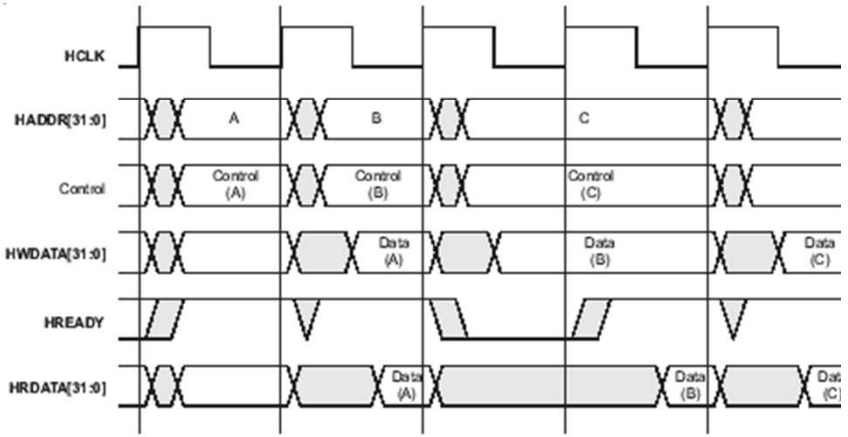
- Event-driven simulation style
  - That’s the name “Transaction”
  - Simulation triggered by data communication
  - Bus events
- Goal
  - High speed simulation
  - Simplify modeling
  - Early system analysis

- In a transaction-level model (TLM), the details of communication among computation components are separated from the details of computation components.

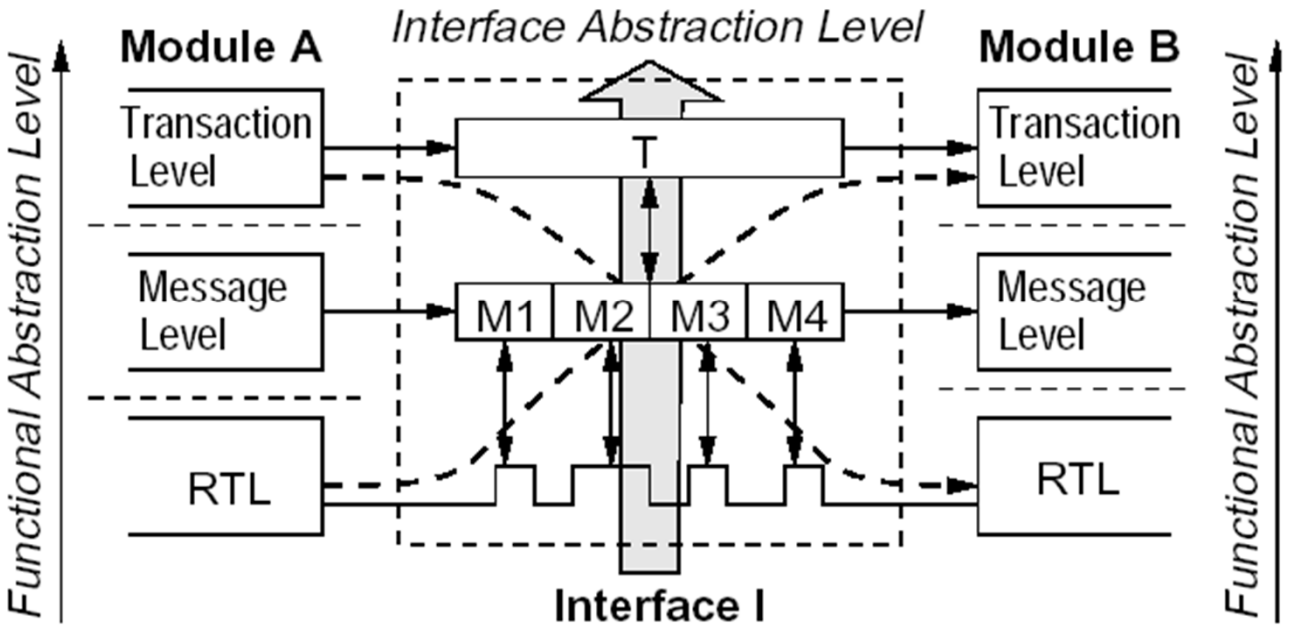
TLM- refinement



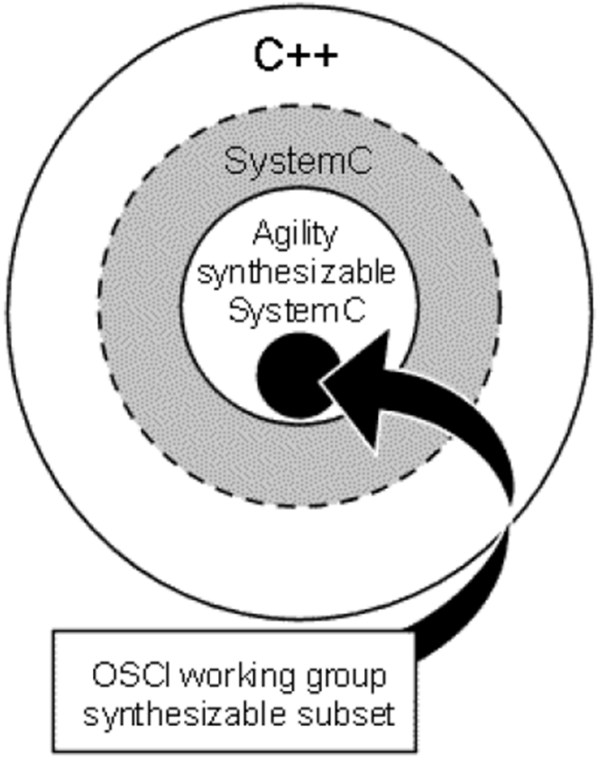




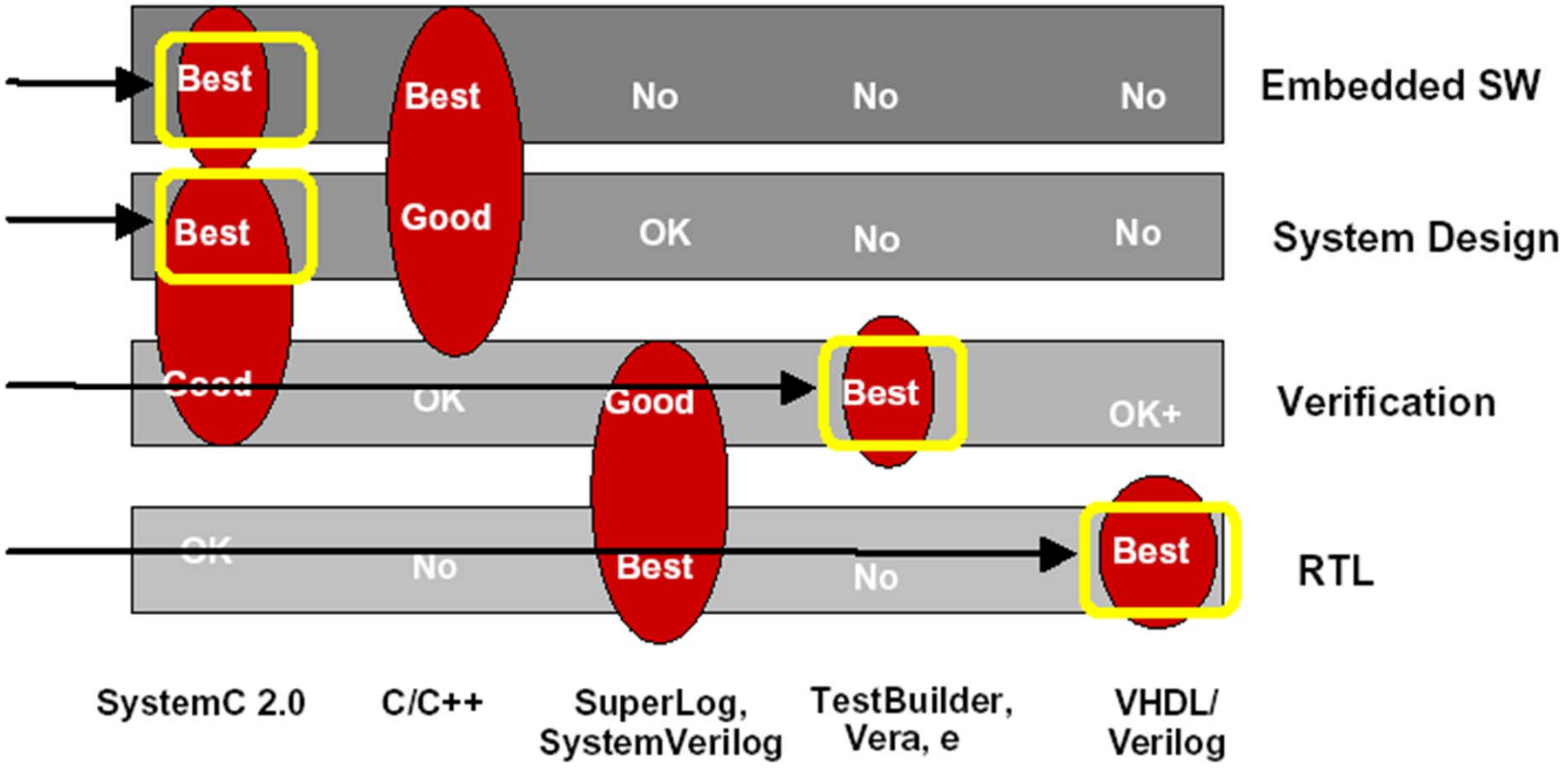
TLM- different abstraction layers



Synthesizable SystemC



Hardware description languages



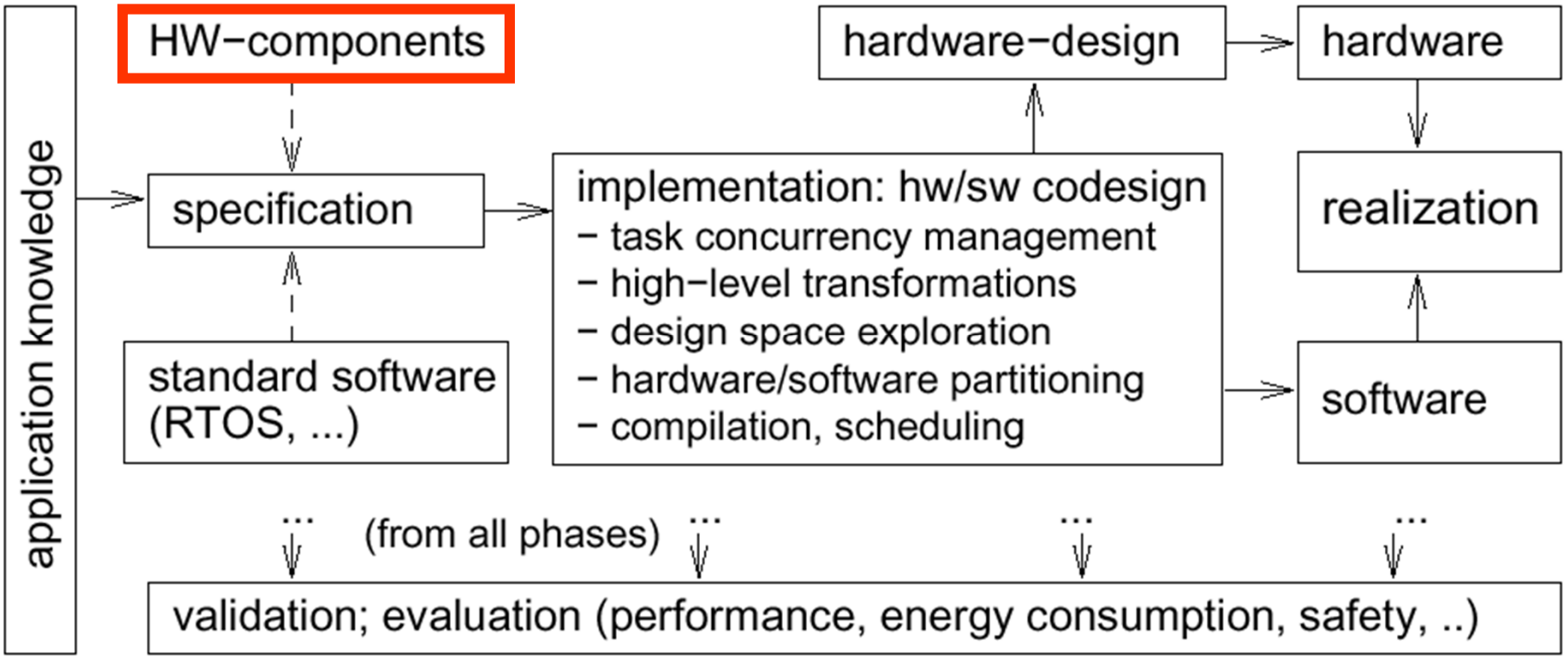
Importance of High-Level Design Methods

System Verification Processing Speeds

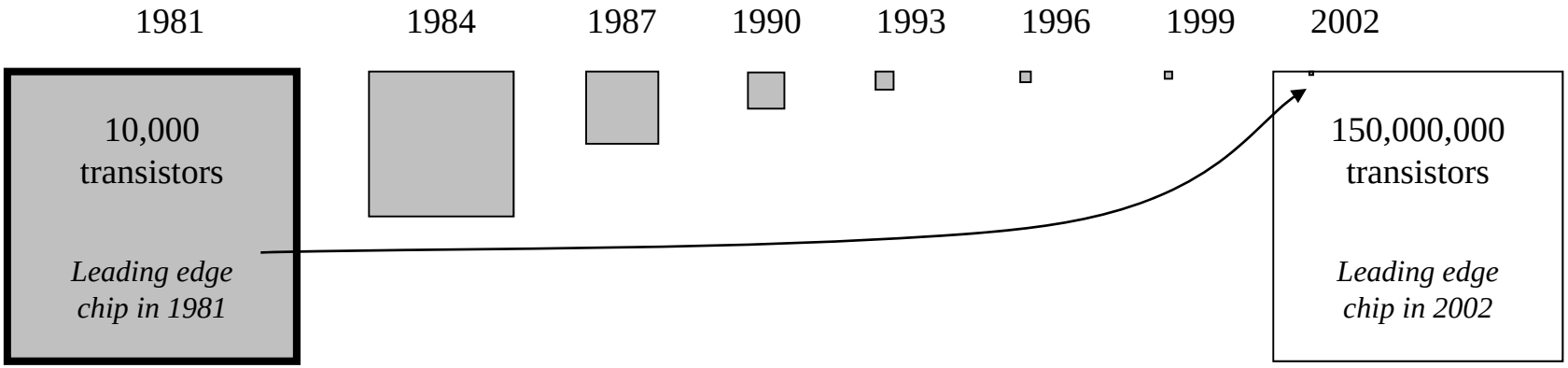
System implementation	Processing time (s/frame)
Behavioral model	1 200 (20 min/frame)
RTL model	144 000 (1.6 days/frame)
Gate model	228 000 (2.6 days/frame)
Gate model on hardware accelerator	1 200
Rapid prototype	0.5
Target hardware	0.05

Source: Paul Clemente, Ron Crevier, Peter Runstadler "RTL and Behavioral Synthesis, A Case Study", VHDL Times, vol. 5, no. 1.

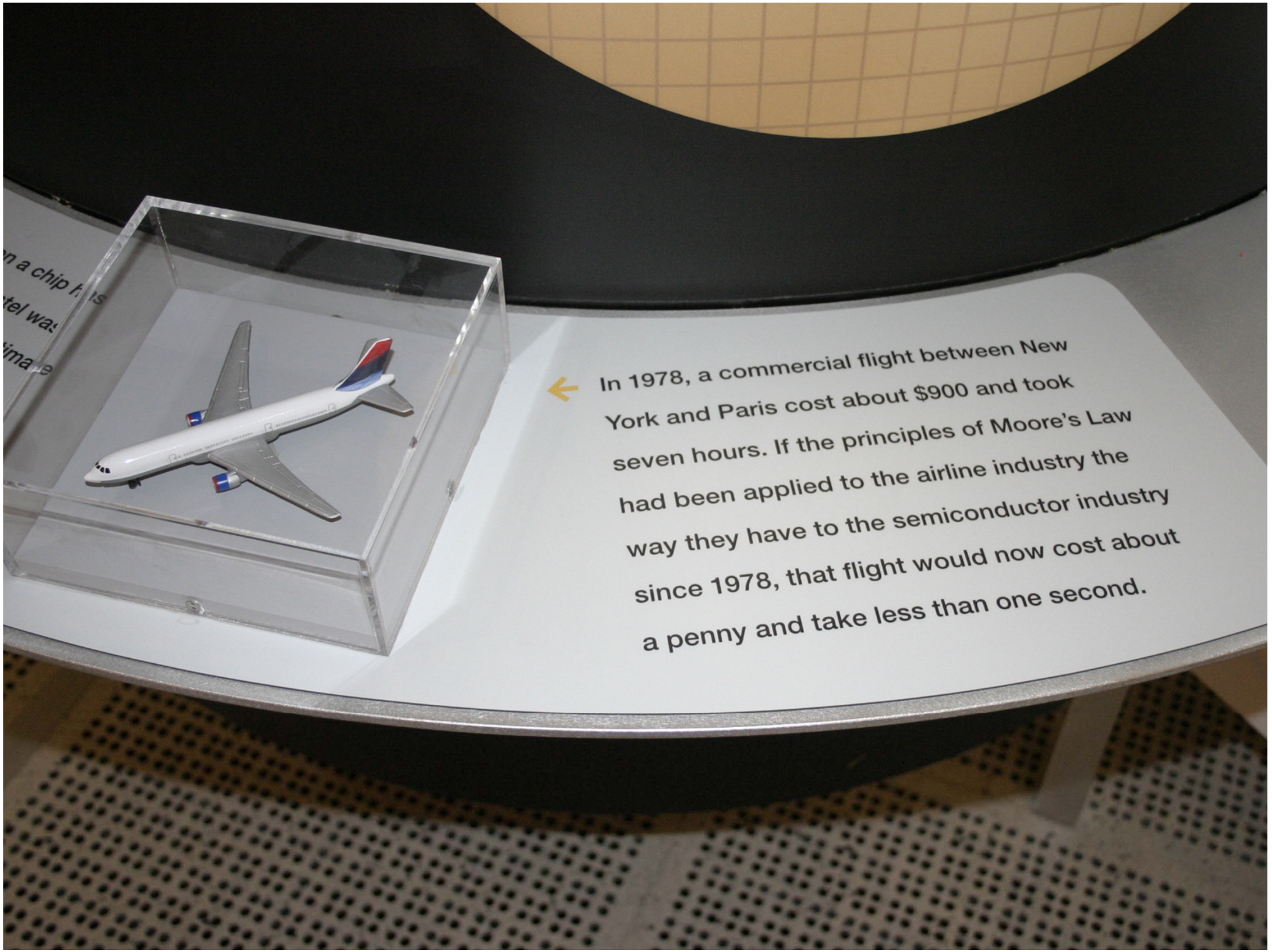
Overview of embedded systems design



Graphical illustration of Moore's law

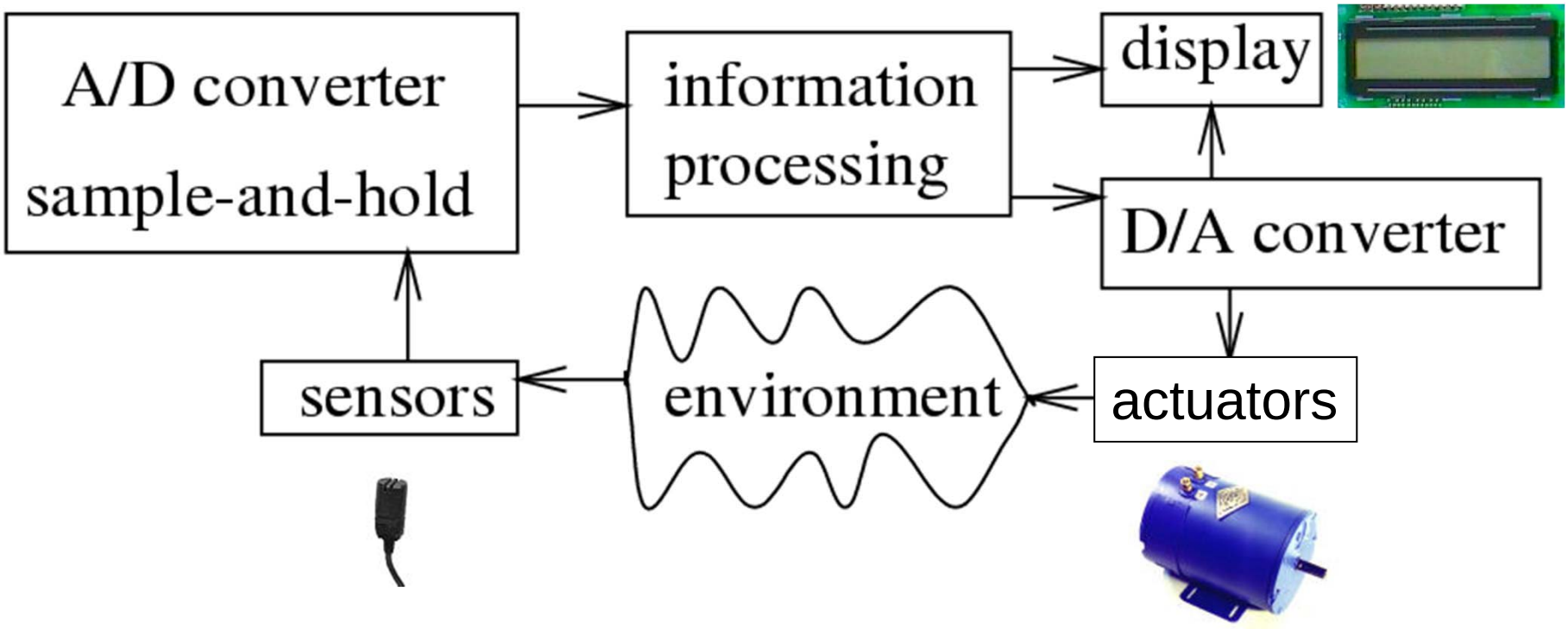


- Something that doubles frequently grows more quickly than most people realize!


In 1978, a commercial flight between New York and Paris cost about $900 and took seven hours. If the principles of Moore's Law had been applied to the airline industry the way they have to the semiconductor industry since 1978, that flight would now cost about a penny and take less than one second.

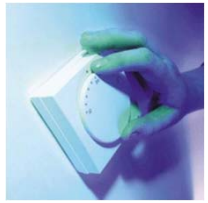
Embedded System Hardware

- Embedded system hardware is frequently used in a loop (***„hardware in a loop“***):



Many examples of such loops

- Heating
- Lights
- Engine control
- Power supply
- …
- Robots

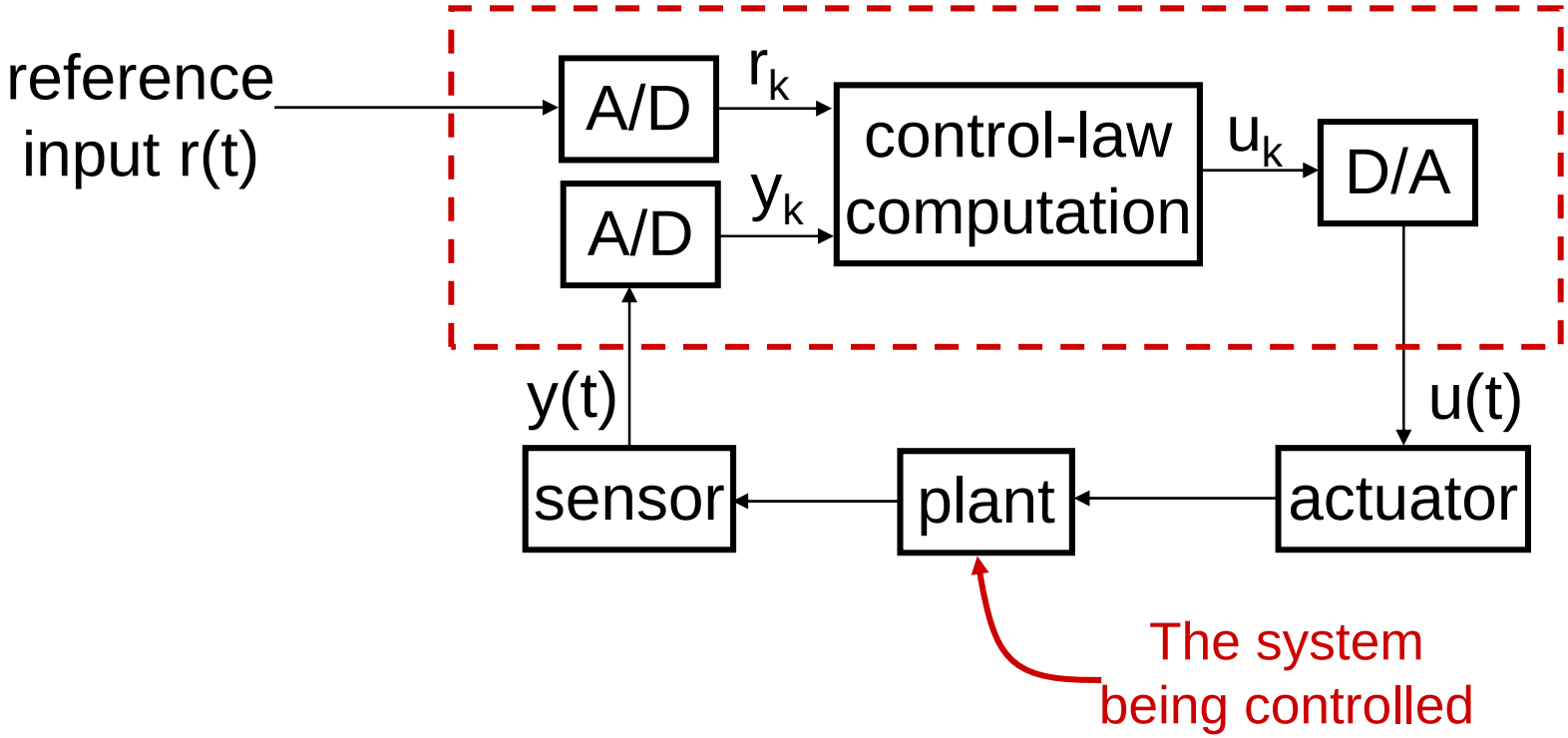


Heating: www.masonsplumbing.co.uk/images/heating.jpg
Robot:: Courtesy and ©: H.Ulbrich, F. Pfeiffer, TU München

Example Real-Time Applications

Many real-time systems are **control systems**.

Example 1: A simple one-sensor, one-actuator control system.



Simple Control System (cont'd)

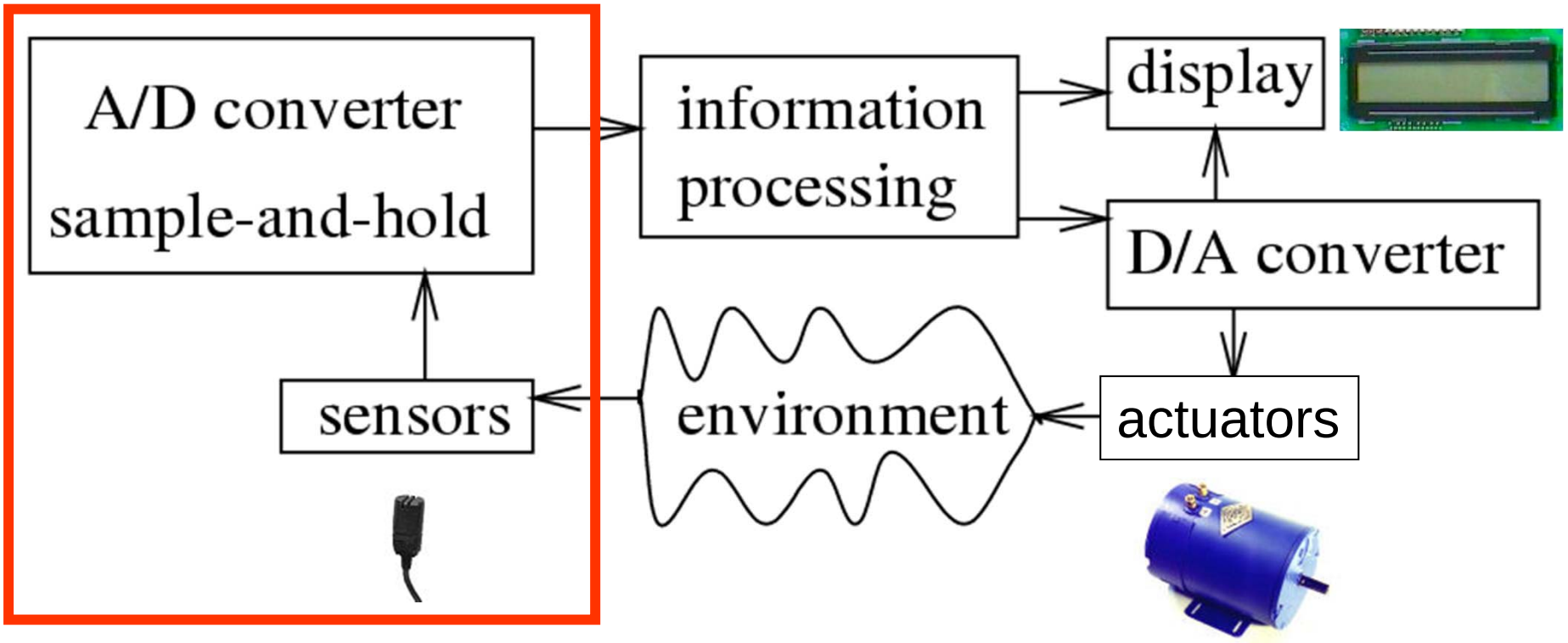
Pseudo-code for this system:

```
set timer to interrupt periodically with period T;
at each timer interrupt do
    do analog-to-digital conversion to get y;
    compute control output u;
    output u and do digital-to-analog conversion;
end do
```

T is called the **sampling period**. *T* is a key design choice.
*T*ypical range for *T*: seconds to milliseconds.

Embedded System Hardware

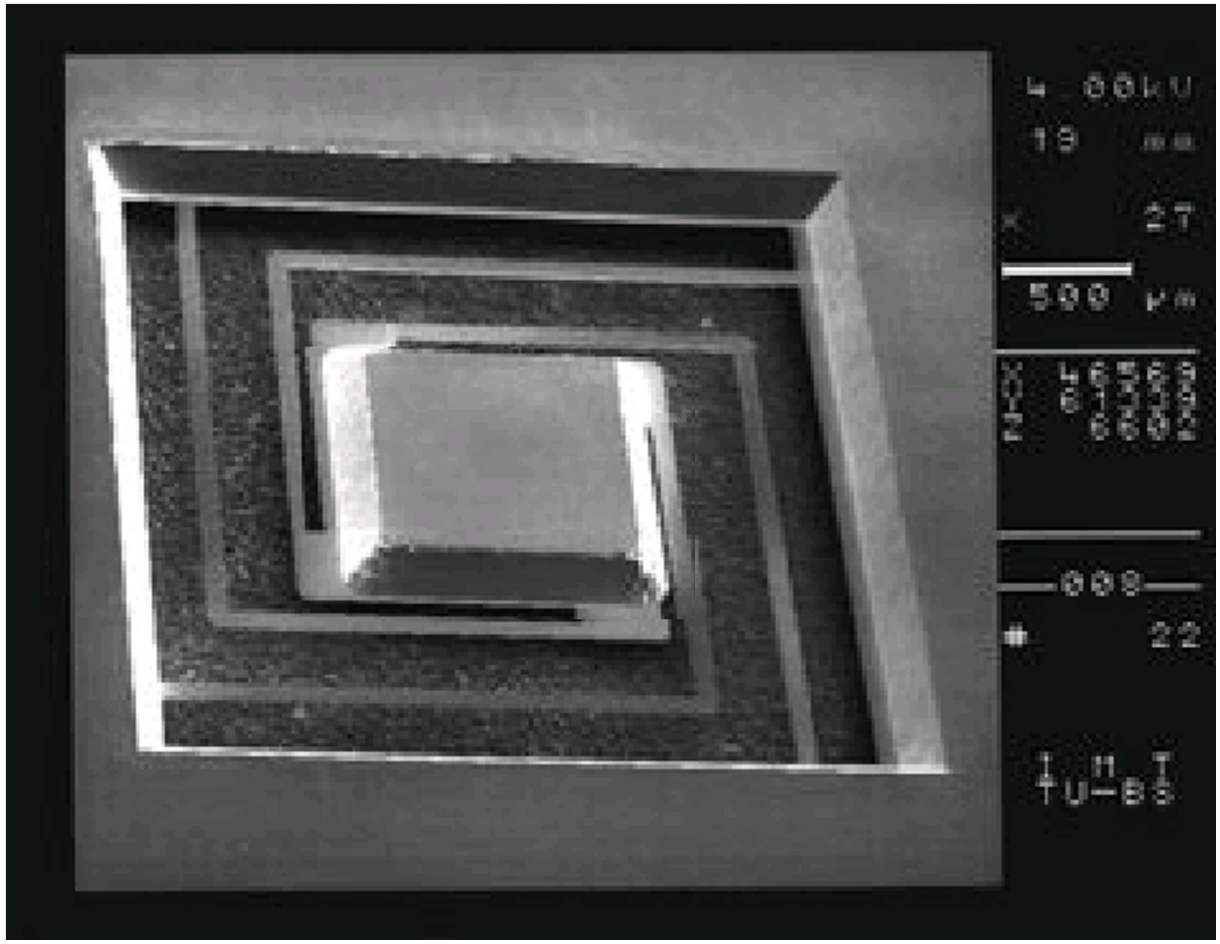
- Embedded system hardware is frequently used in a loop (***„hardware in a loop“***):



Sensors

- Processing of physical data starts with capturing this data.
- Sensors can be designed for virtually every physical and chemical quantity
  - including weight, velocity, acceleration, electrical current, voltage, temperatures etc.
  - chemical compounds.
- Many physical effects used for constructing sensors.
- Examples:
  - law of induction (generation of voltages in an electric field),
  - light-electric effects.
- Huge amount of sensors designed in recent years.

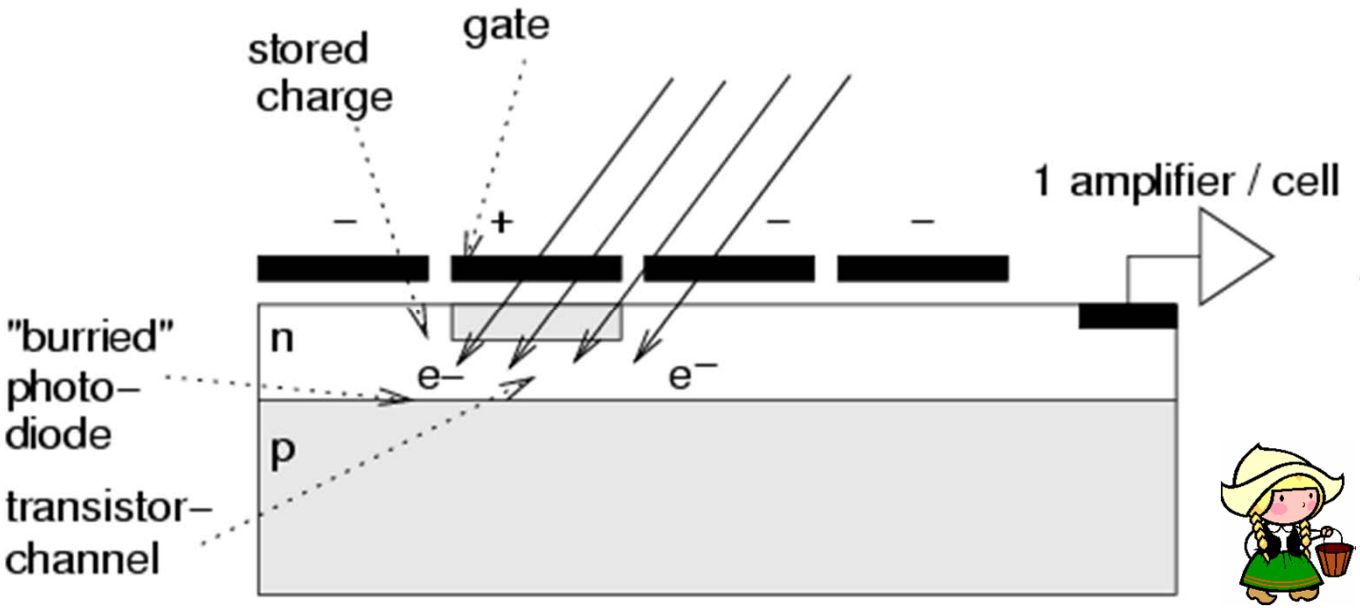
Example: Acceleration Sensor



Courtesy & ©: S. Bütgenbach, TU Braunschweig

Charge-coupled devices (CCD) image sensors

Based on charge transfer to next pixel cell



Corresponding to “bucket brigade device”
(German: “*Eimerkettenschaltung*”)

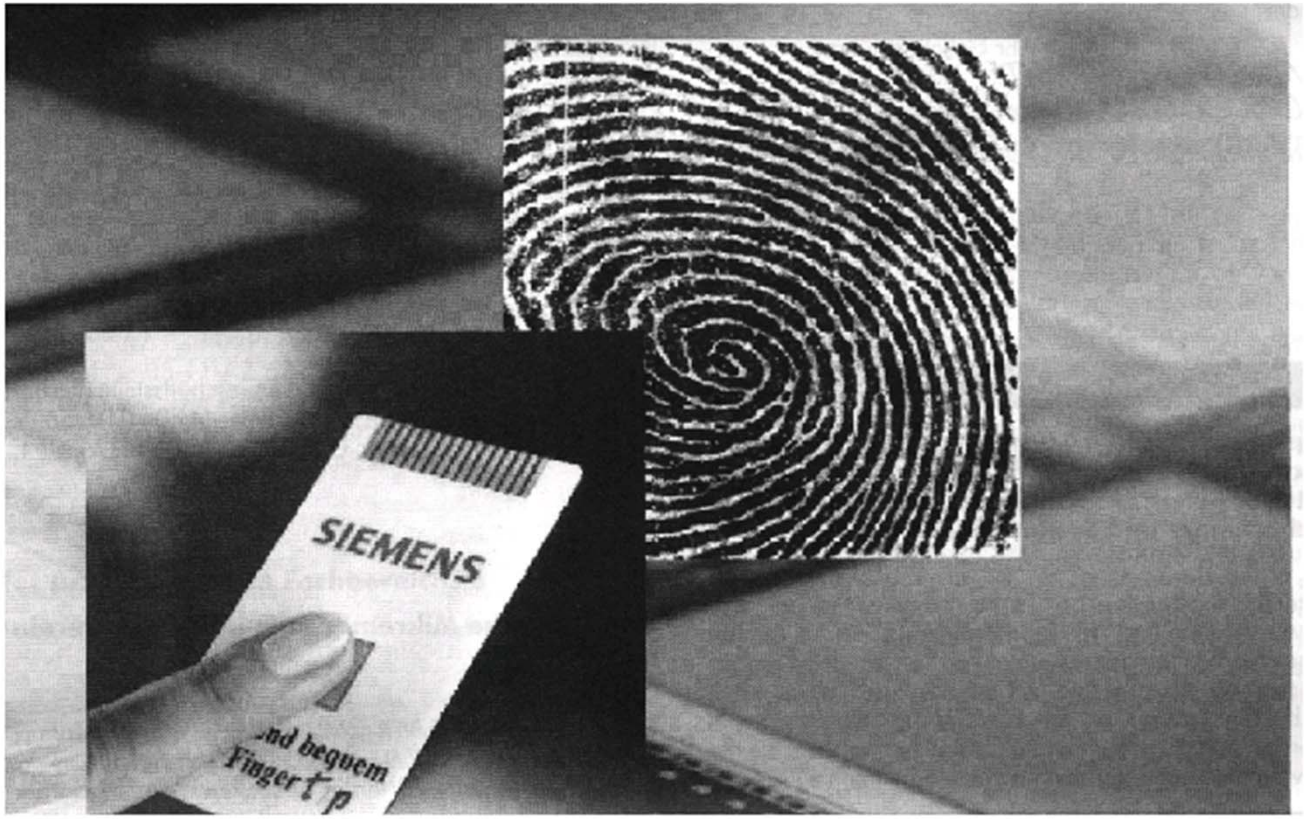
Comparison CCD/CMOS sensors

Property	CCD	CMOS
Technology optimized for	Optics	VLSI technology
Technology	Special	Standard
Smart sensors	No, no logic on chip	Logic elements on chip
Access	Serial	Random
Size	Limited	Can be large
Power consumption	Low	Larger
Applications	Compact cameras	Low cost devices

See also B. Diericks: CMOS image sensor concepts. Photonics West 2000 Short course (Web)

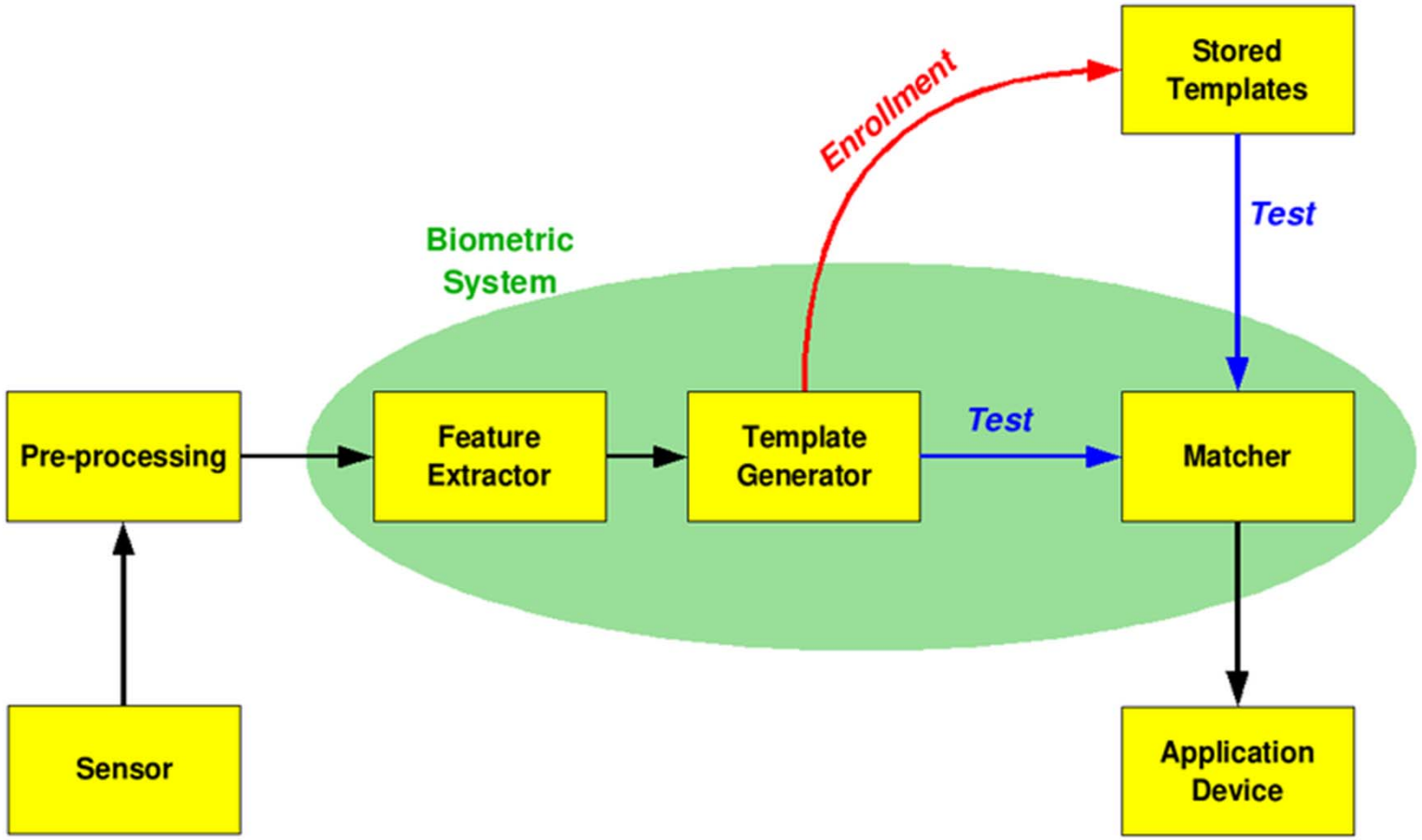
Example: Biometrical Sensors

Example: Fingerprint sensor (© Siemens, VDE):



Matrix of 256 x 256 elem. Voltage ~ distance. Resistance also computed. No fooling by photos and wax copies.

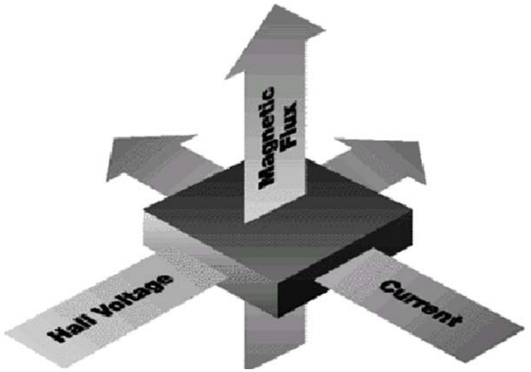
Example: Biometrical System



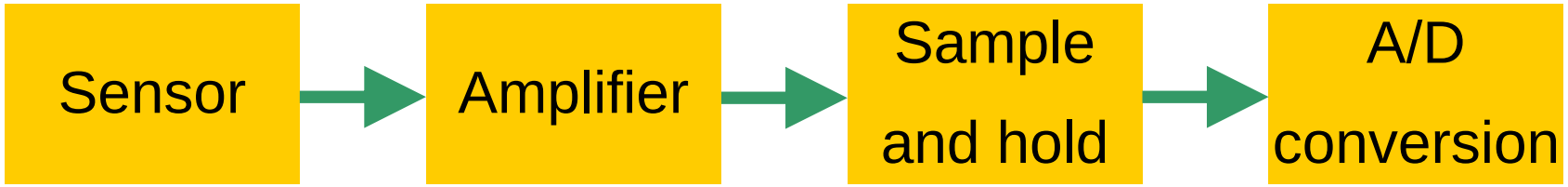
e.g.; Integrated into ID mouse.

Other sensors

- Rain sensors for wiper control
  („Sensors multiply like rabbits“ [ITT automotive])
- Pressure sensors
- Proximity sensors
- Engine control sensors
- Hall effect sensors



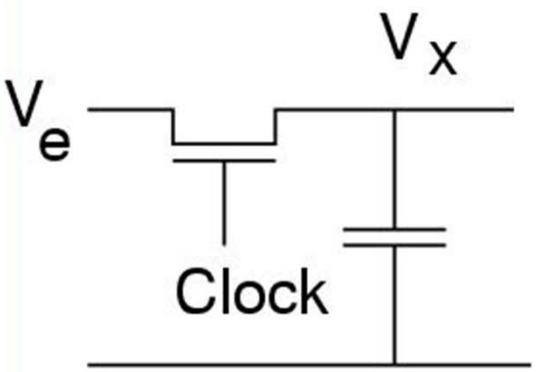
Standard layout of sensor systems for contin. entities

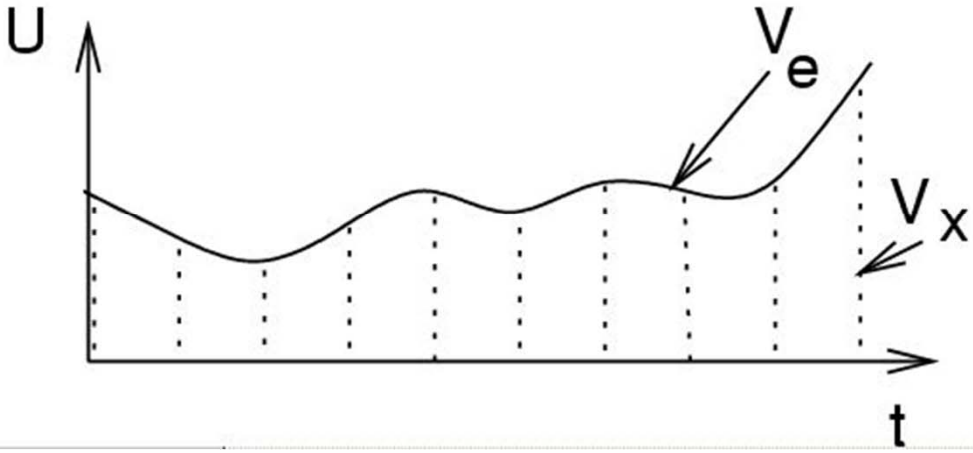


- Sensor: detects/measures entity and converts it to electrical domain
  - May entail ES-controllable actuation: e.g. charge transfer in CCD
- Amplifier: adjusts signal to the dynamic range of the A/D conversion
  - Often dynamically adjustable gain: e.g. ISO settings at digital cameras, input gain for microphones (sound or ultrasound), extremely wide dynamic ranges in seismic data logging
- Sample + hold: samples signal at discrete time instants
- A/D conversion: converts samples to digital domain

Discretization of time

V_e is a mapping $\mathbb{R} \rightarrow \mathbb{R}$





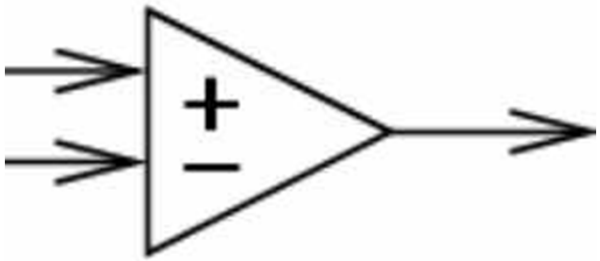
V_x is a **sequence** of values or a mapping $\mathbb{Z} \rightarrow \mathbb{R}$

Discrete time: sample and hold-devices.

Ideally: width of clock pulse -> 0

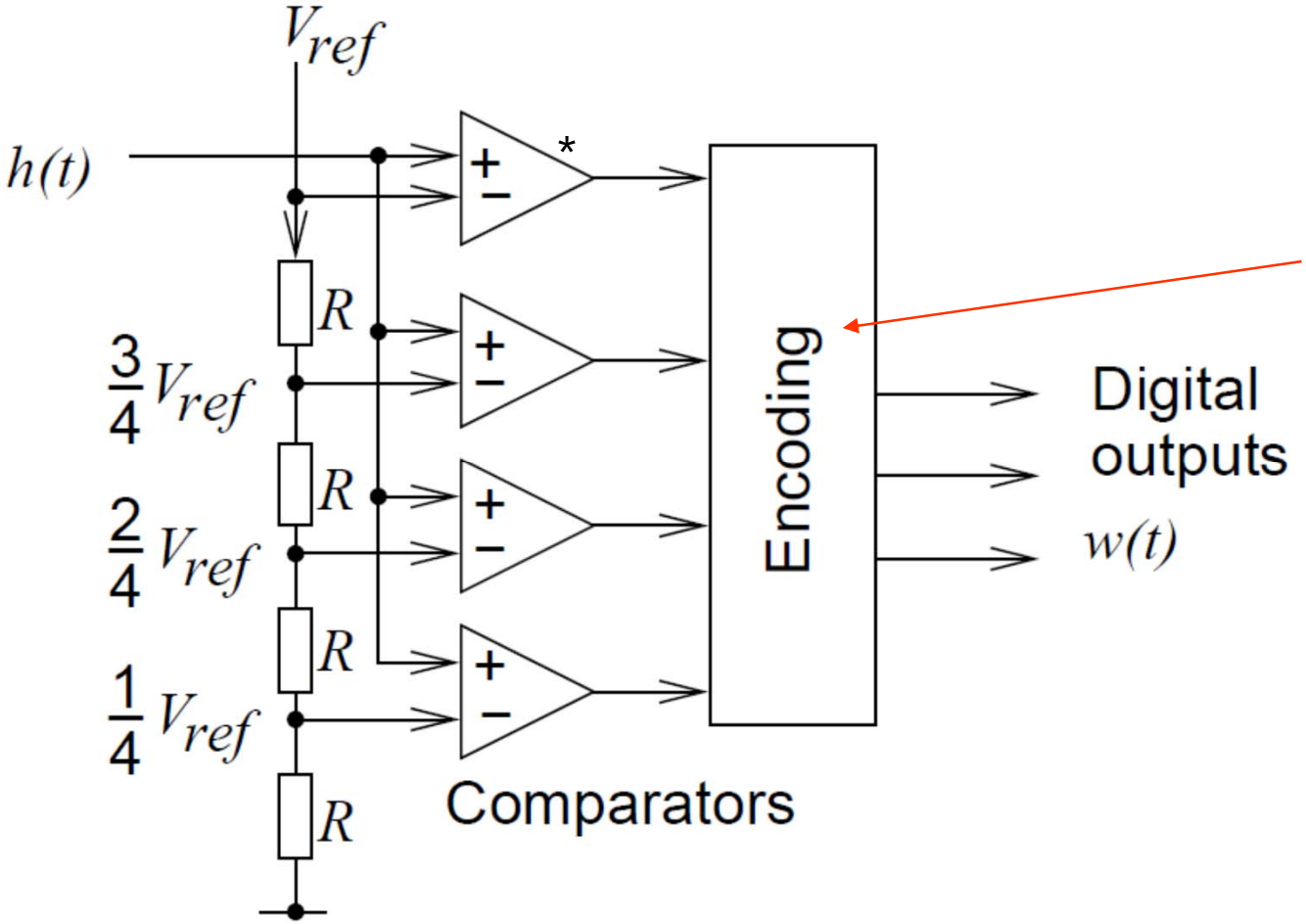
Discretization of values: A/D-converters
1. Flash A/D converter (1)

- Basic element: analog comparator



  - Output = ´1´ if voltage at input + exceeds that at input -.
  - Output = ´0´ if voltage at input - exceeds that at input +.
- Idea:
  - Generate *n* different voltages by voltage divider (resistors), e.g. V_{ref}, ¾ V_{ref}, ½ V_{ref}, ¼ V_{ref}.
  - Use *n* comparators for parallel comparison of input voltage V_x to these voltages.
  - Encoder to compute digital output.

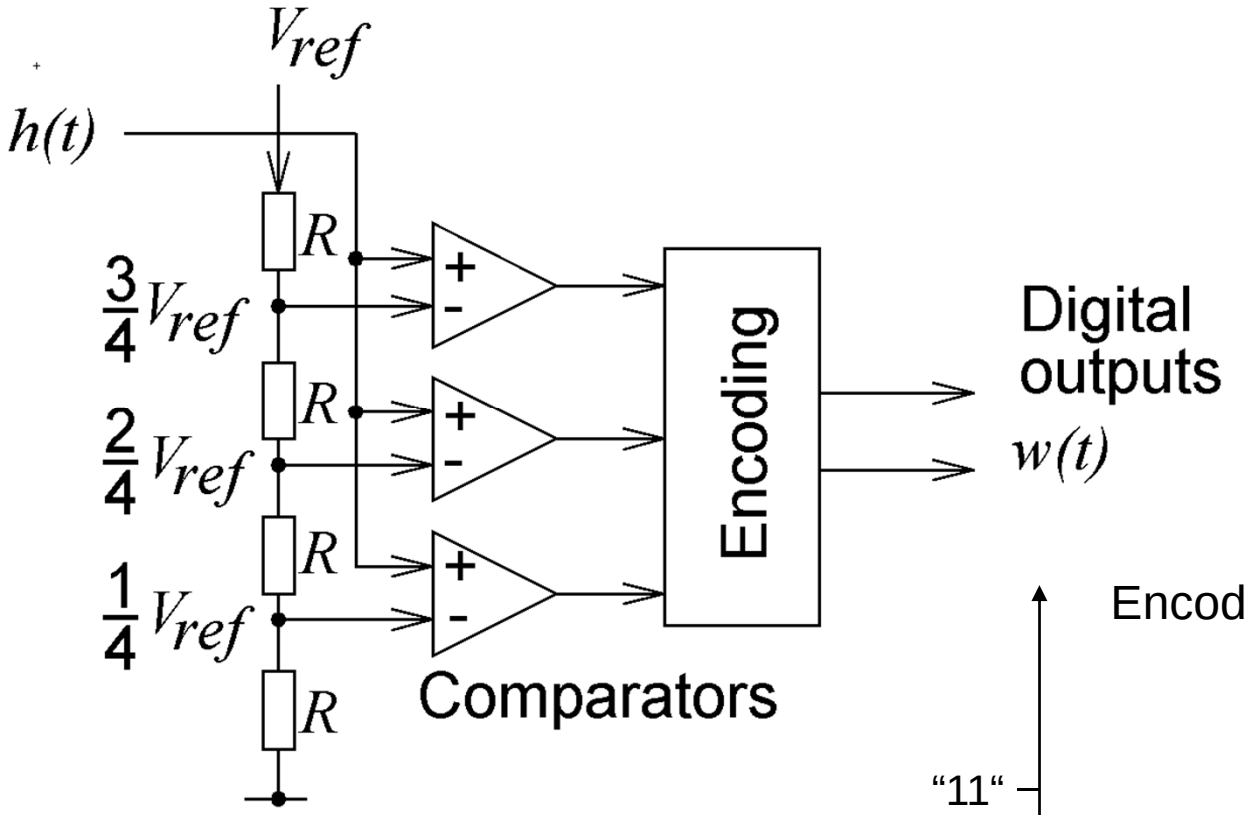
Flash A/D converter

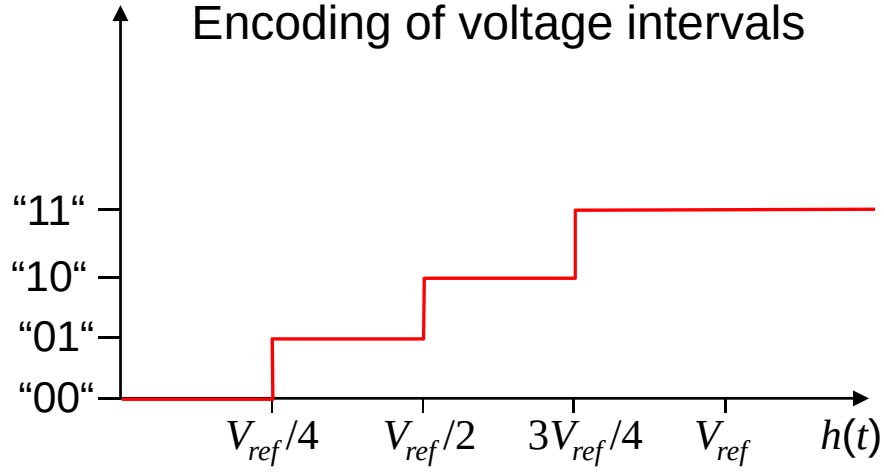


- Encodes input number of most significant ‘1’ as an unsigned number, e.g. “1111” -> “100”, “0111” -> “011”, “0011” -> “010”, “0001” -> “001”, “0000” -> “000” (Priority encoder).

* Frequently, the case $h(t) > V_{ref}$ would not be decoded

Assuming $0 \leq h(t) \leq V_{ref}$





Resolution and speed of Flash A/D-converter

- Resolution (in bits): number of bits produced
- Resolution Q (in volts): difference between two input voltages causing the output to be incremented by 1

$$Q = \frac{V_{FSR}}{n}$$ with

Q: resolution in volts per step
V_{FSR}: difference between largest and smallest voltage
n: number of voltage intervals

Example:
$Q = V_{ref}/4$ for the previous slide, assuming * to be absent

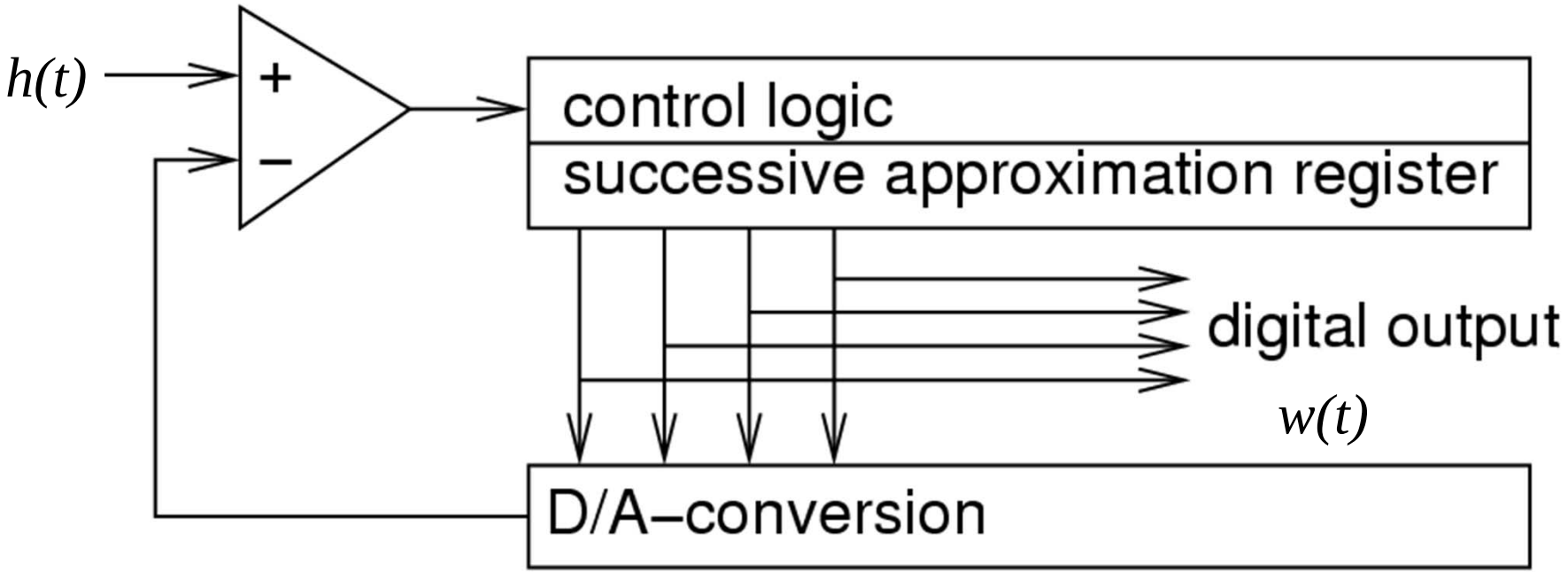
- **Parallel comparison with reference voltage**

  Speed: $O(1)$

  Hardware complexity: $O(n)$

  Applications: e.g. in video processing

Higher resolution:
Successive approximation



Key idea: binary search:

Set MSB='1'

if too large: reset MSB

Set MSB-1='1'

if too large: reset MSB-1

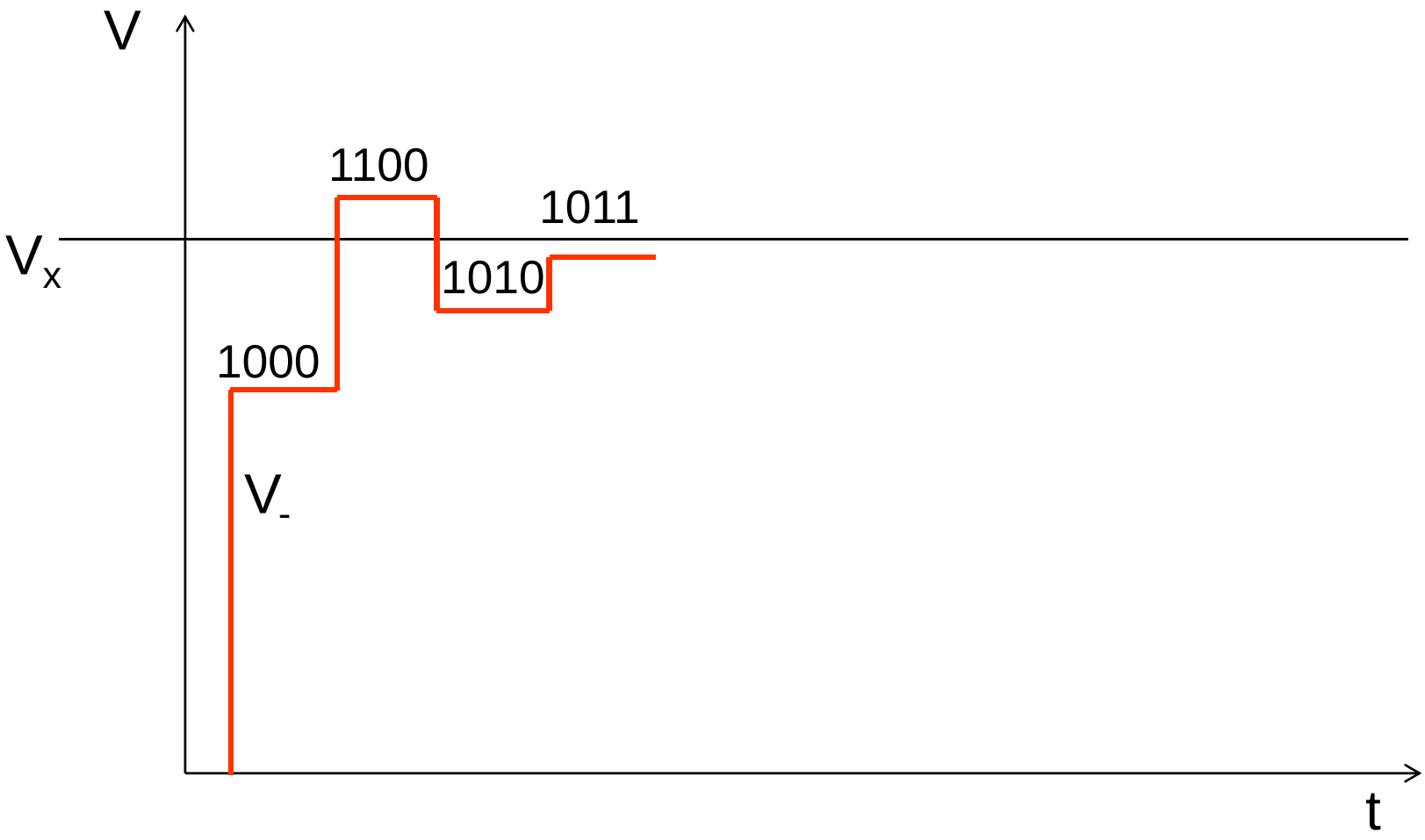
Speed: *O(ld(n))*

Hardware complexity: *O(ld(n))*

with n= # of distinguished voltage levels;

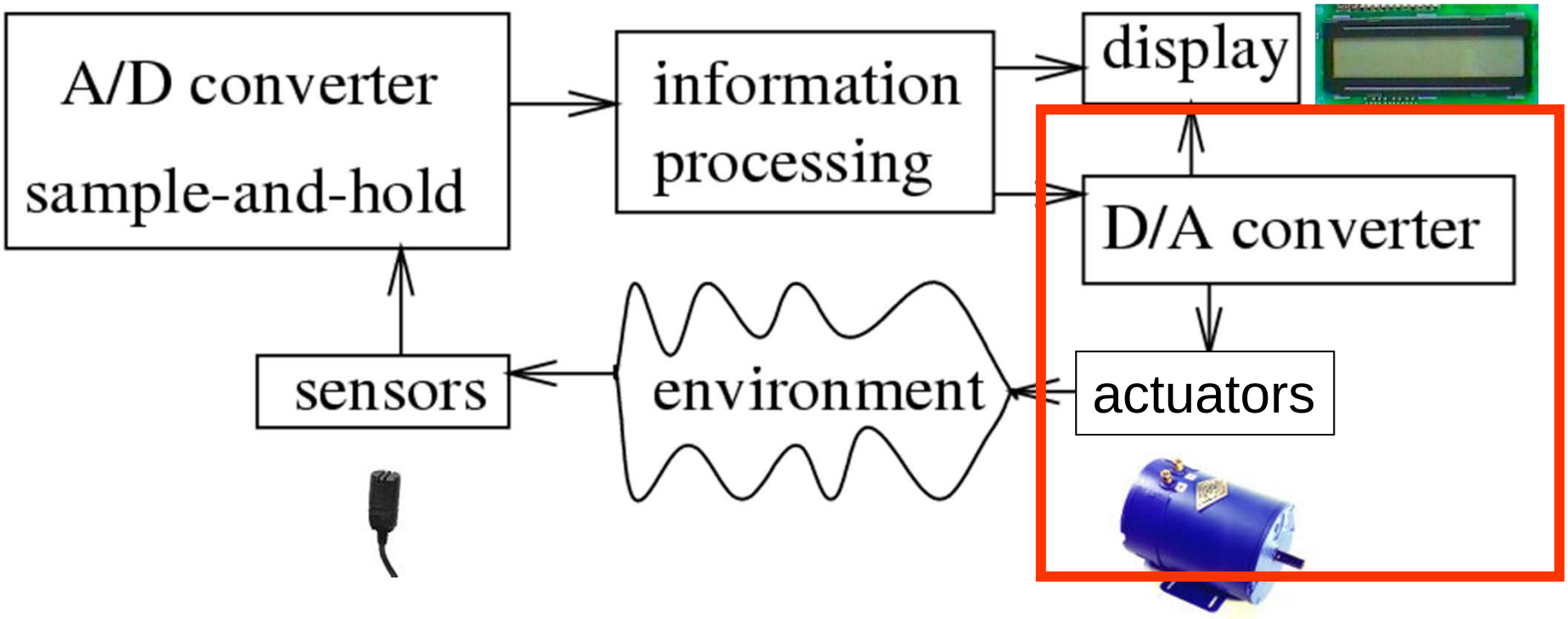
slow, but high precision possible.

Successive approximation (2)

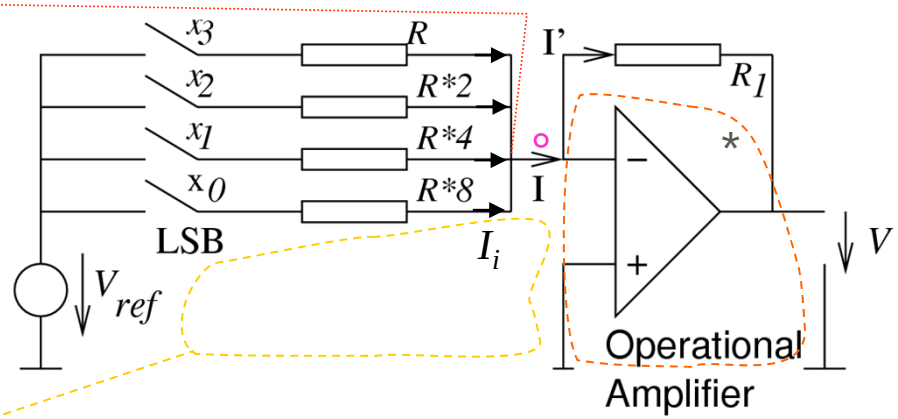


Embedded System Hardware

- Embedded system hardware is frequently used in a loop (***„hardware in a loop“***):



Digital-to-Analog (D/A) Converters



Junction rule: $I = \sum_i I_i$

Loop rule: $I_i = x_i \times \frac{V_{ref}}{2^{3-i} \times R}$

☞ $$I = x_3 \times \frac{V_{ref}}{R} + x_2 \times \frac{V_{ref}}{2 \times R} + x_1 \times \frac{V_{ref}}{4 \times R} + x_0 \times \frac{V_{ref}}{8 \times R} = \frac{V_{ref}}{8 \times R} \times \sum_{i=0}^{3} x_i \times 2^i$$

Loop rule*: $V + R_1 \times I' = 0$

$I \sim nat\,(x)$, where $nat(x)$: natural number represented by x;

Junction rule°: $I = I'$

Hence: $V + R_1 \times I = 0$

Finally: $$-V = V_{ref} \times \frac{R_1}{8 \times R} \sum_{i=0}^{3} x_i \times 2^i = V_{ref} \times \frac{R_1}{8 \times R} \times nat(x)$$

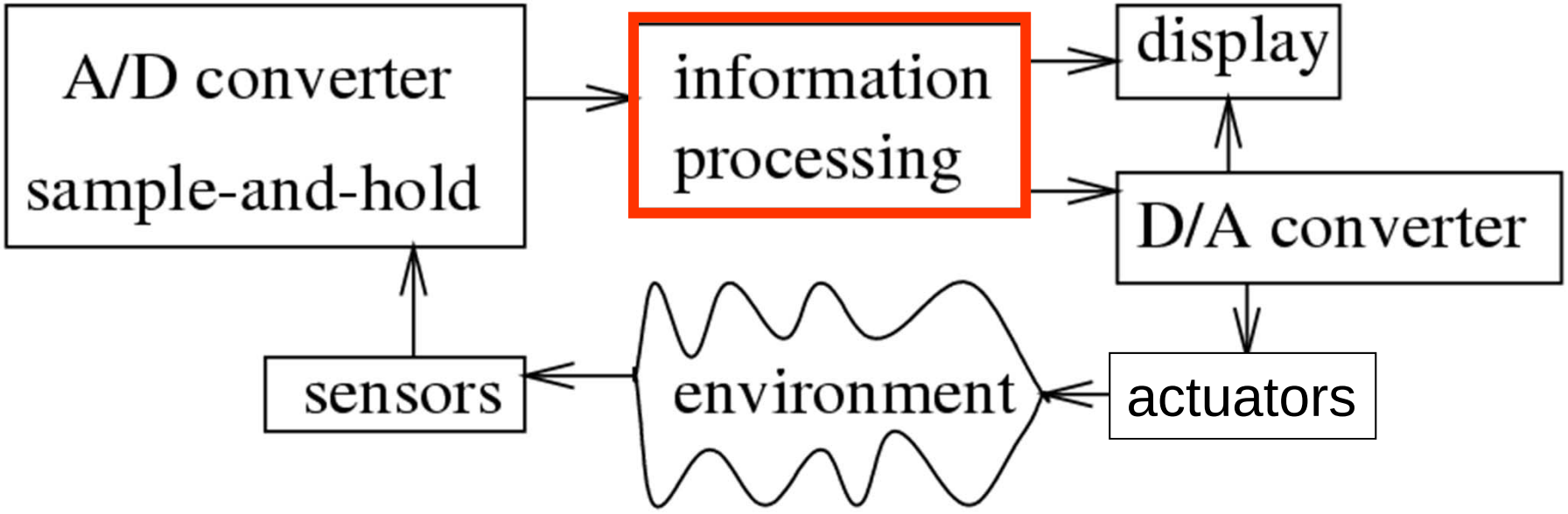
Op-amp turns current $I \sim nat\,(x)$ into a voltage $\sim nat\,(x)$

Actuators and output

- Huge variety of actuators and outputs, impossible to represent
- Two base types:
  - analogue drive (requires D/A conversion)
    - speakers, electrical motors with collector
    - electromagnetic (e.g., coils) or electrostatic drives
    - piezo drives
  - digital drive (requires amplification only)
    - LEDs
    - stepper motors
    - relais, electromagnetic valve

Embedded System Hardware

- Embedded system hardware is frequently used in a loop (***„hardware in a loop“***):

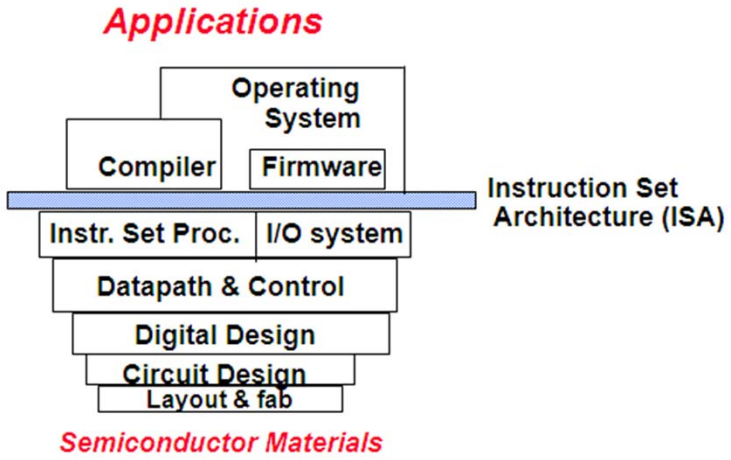


Instruction Set Architecture

Instruction Set Architecture

Is the interface between hardware and software.

- allows easy programming (compilers, OS, ..);
  - Provides convenient functionality to higher levels
- allows efficient implementations (hardware);
  - Permits an efficient implementation at lower levels
- has a long lifetime (survives many HW generations) - portability



Instruction Set Architecture (ISA)

- How is data represented?
  - Set of machine-recognized data types
    - bytes, words, integers, floating point, strings, . . .
- Where can data be stored?
  - Programmable storage
    - regs, PC, memory
- How can data be accessed?
  - Methods of identifying and obtaining data referenced by instructions (addressing modes)
    - reg., absolute, relative, reg + offset, …
- What operations can be done on data?
  - Operations performed on those data types
    - Add, sub, mul, div, xor, move, ….
- How are instructions encoded?

  Format (encoding) of the instructions
    - Op code, operand fields, …

How is data represented?
- Data formats

- The ISA supports several data formats by providing representations for integers, characters, floating-point, multimedia, etc.
- **Integer data formats** can be signed or unsigned
  (e.g., in DEC Alpha there is byte, 16-bit word, 32-bit longword, and 64-bit quadword).
- There are two ways of **ordering byte addresses** within a word
  - big-endian: most significant byte first, and
  - little-endian: least significant byte first.
- There are also packed and unpacked BCD numbers, and ASCII characters.
- **Floating-point data formats** (ANSI/IEEE 754-1985):
  standard, basic or extended, each having two widths: single or double.
- **Multimedia data formats** are 32-, 64-, and 128-bit words (soon perhaps also 256-bit) concluding several 8- or 16-bit pixel representations or 32-bit (single precision) floating-point numbers used for 3D graphics.

Where can data be stored?
- Address space

- Several **address spaces** are distinguished by the (assembly language) programmer, such as register space, stack space, heap space, text space, I/O space, and control space.
- Except for the registers, all other address spaces are mapped onto a single contiguous memory address space.
- A RISC ISA additionally contains a register file, which consists of a relatively large number of general-purpose CPU registers
  - early RISC processors: MIPS: 32 32-bit general purpose registers,
  RISC I: register windowing
- Contemporary RISC processors: additionally 32 64-bit floating-point and multimedia registers.

How can data be accessed? - Addressing modes

- **Register mode**: the operand is stored in one of the registers.
- **Immediate** (or literal) mode: the operand is a part of the instruction.
- **Direct** (or absolute) mode: the address of the operand in memory is stored in the instruction.
- **Register indirect** (or register deferred) mode: the address of the operand in memory is stored in one of the registers.
- **Autoincrement** (or register indirect with postincrement) mode: like the register indirect, except that the content of the register is incremented after the use of the address.
  - This mode offers automatic address increment useful in loops and in accessing byte, half-word, or word arrays of operands.
- **Autodecrement** (register indirect with predecrement) mode: the content of the register is decremented and is then used as a register indirect address.
  - This mode can be used to scan an array in the direction of decreasing indices.
- ....

Addressing Modes

Addressing mode	Example instruction / Meaning
Register	`load Reg1,Reg2` `Reg1 ← (Reg2)`
Immediate	`load Reg1,#const` `Reg1 ← const`
Direct	`load Reg1,(const)` `Reg1 ← Mem[const]`
Register indirect	`load Reg1,(Reg2)` `Reg1 ← Mem[(Reg2)]`
Autoincrement	`load Reg1,(Reg2)+` `Reg1 ← Mem[(Reg2)], Reg2 ← (Reg2) + step`
Autodecrement	`load Reg1,-(Reg2)` `Reg2 ← (Reg2) - step, Reg1 ← Mem[(Reg2)]`
Displacement	`load Reg1,displ(Reg2)` `Reg1 ← Mem[displ + (Reg2)]`
Indexed and scaled indexed	`load Reg1,(Reg2*scale)` `Reg1 ← Mem[(Reg2)*scale]`
Indirect scaled indexed	`load Reg1,(Reg2,Reg3*scale)` `Reg1 ← Mem[(Reg2) + (Reg3)*scale]`
Indirect scaled indexed with displacement	`load Reg1,displ(Reg2,Reg3*scale)` `Reg1 ← Mem[displ + (Reg2) + (Reg3)*scale]`
PC-relative	`branch displ` `PC ← PC + displ` (if branch taken)

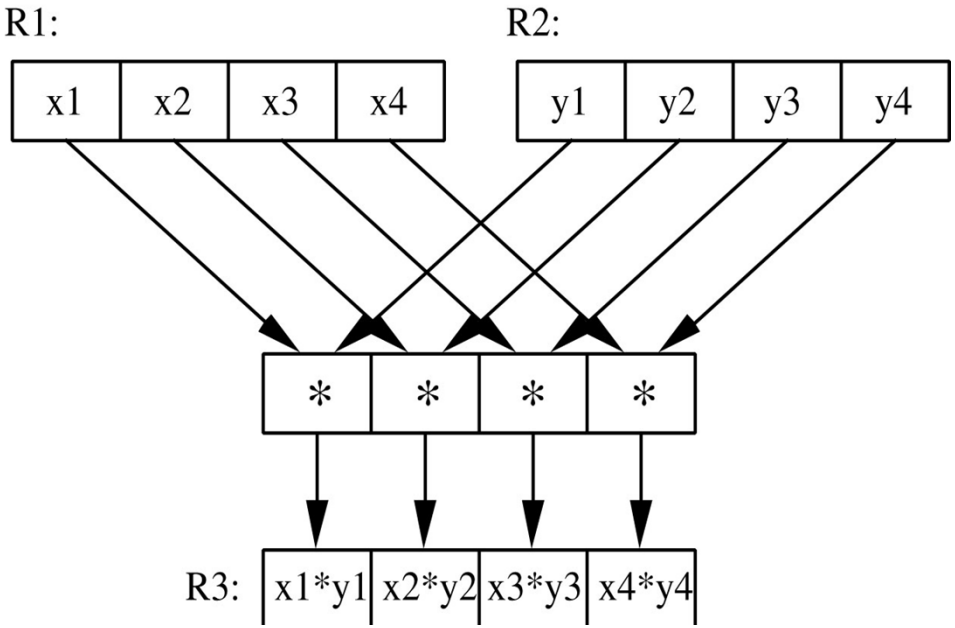
`const,displ`	...	decimal, hexadecimal, octal or binary numbers
`step`	...	e.g., 4 in systems with 4-byte uniform instruction size
`scale`	...	scaling factor, e.g., 1, 2, 4, 8, 16

What operations can be done on data? - Instruction set

- **Data movement instructions**: transfer data from one location to another.
  - When there is a separate I/O address space, these instructions also include special I/O instructions.
  - Stack manipulation instructions (e.g. push, pop) also fall into this category.
- **Integer arithmetic and logical instructions**: can be one-operand (e.g. complement), two-operand or three-operand instructions.
  - In some processors, different instructions are used for different data formats of their operands.
    There may be separate signed and unsigned multiply/divide instructions.
- **Shift and rotate instructions**: left or right shifts and rotations.
  - There are two types of shifts: logical and arithmetic.
- **Bit manipulation instructions**: operate on specified fields of bits. The field is specified by its width and offset from the beginning of the word. Instructions usually include test (affecting certain flags), set, clear, and possibly others.

Instruction Set (continued)

- **Multimedia instructions**:
  - process multiple sets of small operands and obtain multiple results by a single instruction
  - Utilization of subword parallelism (data parallel instructions, SIMD)
  - Saturation arithmetic
  - Additional arithmetic, masking and selection, reordering and conversion instructions



Instruction Set (continued)

- **Floating-point instructions**: floating-point data movement, arithmetic, comparison, square root, absolute value, transcendental functions, and others.
- **Control transfer instructions**: consist primarily of jumps, branches, procedure calls, and procedure returns. We assume that jumps are unconditional and branches are conditional. Some systems may also have return from exception instructions.
- **System control instructions**: allow the user to influence directly the operation of the processor and other parts of the computer system.
- **Special function unit instructions**: perform particular operations on special function units (e.g. graphic units).
  Another type of special instructions are atomic instructions for controlling the access to critical sections in multiprocessors.

- Depending on the way of specifying its operands an instruction can be one of the following types:
  - register-register, memory-register, register-memory, or memory-memory.

How are instructions encoded?
- Instruction and addressing formats

- **3-address instruction format**: opcode | Dest | Src1 | Scr2;
  typically used by register-register (also called load/store) machines.
- **2-address instruction format**: opcode | Dest/Src1 | Src2 ;
  often supported register-memory machines.
- **1-address instruction format**: opcode | Src;
  supported by the accumulator machine.
- **0-address instruction format**: only opcode;
  supported by the stack machine.

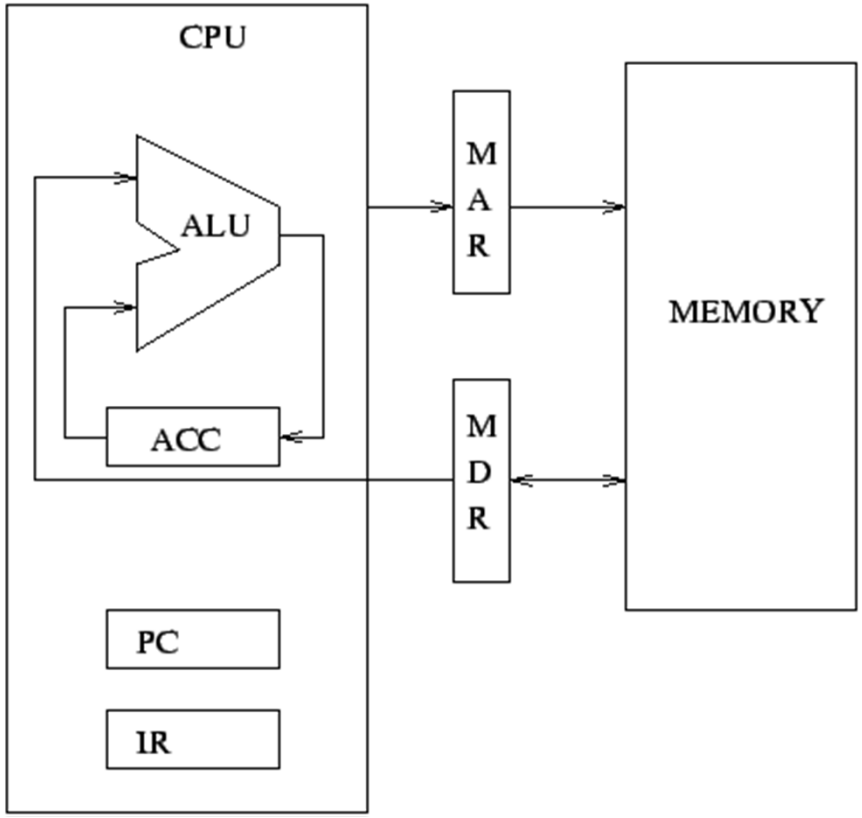
- Most RISC ISAs use a 3-address instruction format where all instructions have a fixed length of 32 bits.
- CISC ISAs often use register-memory with variable instruction lengths.
- Accumulator machines are today mostly found in microcontrollers.
- Also stack machines use variable instruction lengths, today exemplified in JAVA processors.

Example - accumulator machine

C = A + B

D = C - B

```
load A
add B
store C
load C
sub B
store D
```

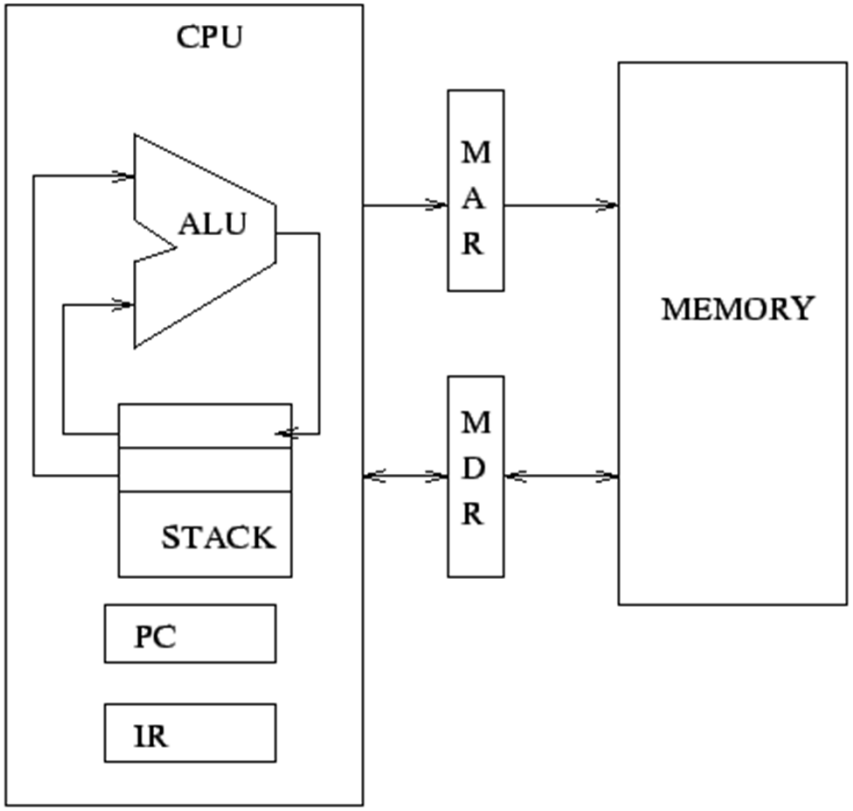


Example – stack machine

C = A + B

D = C - B

push B
push A
add
pop C
push B
push C
sub
pop D

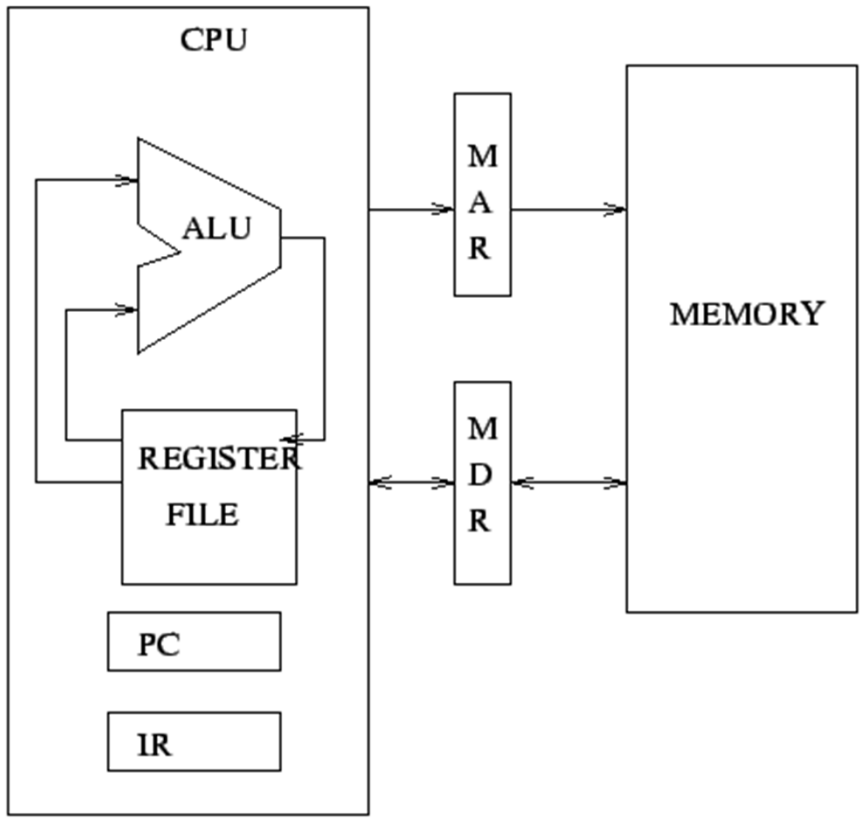


Example – register machine

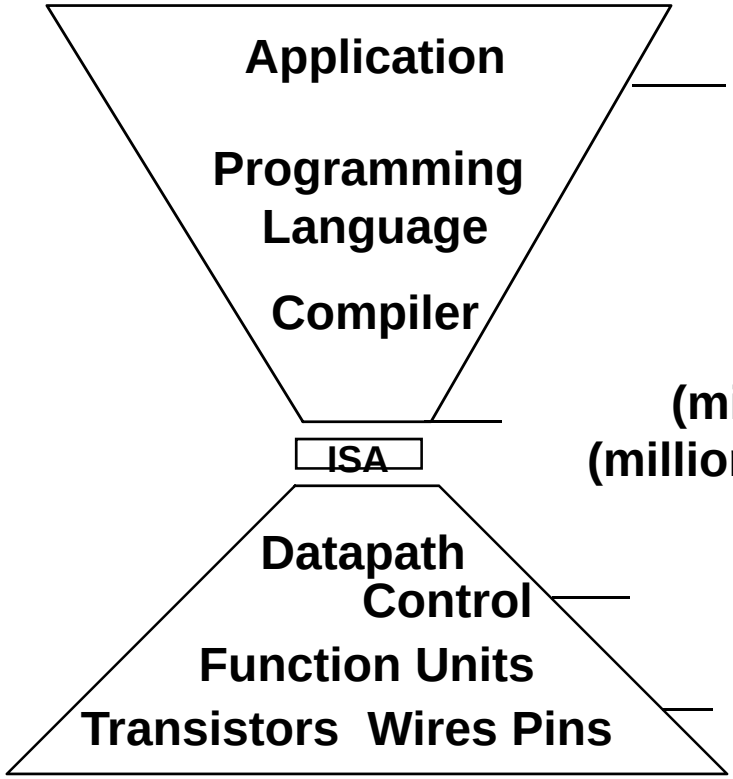
C = A + B

D = C - B

```
load Reg1,A
load Reg2,B
add Reg3,Reg1,Reg2
store C,Reg3
load Reg1,C
load Reg2,B
sub Reg3,Reg1,Reg2
store D,Reg3
```



Metrics of Performance



Answers per day/month

(millions) of Instructions per second: MIPS
(millions) of (FP) operations per second: MFLOP/s

Megabytes per second

Cycles per second (clock rate)

CISC vs. RISC

What is CISC?

- CISC is an acronym for Complex Instruction Set Computer and are chips that are easy to program and which make efficient use of memory
  - earliest machines were programmed in assembly language and memory was slow and expensive,
- Most common microprocessor designs such as the Intel 80x86 and Motorola 68K series followed the CISC philosophy.
- But recent changes in software and hardware technology have forced a re-examination of CISC and many modern CISC processors are hybrids, implementing many RISC principles.

CISC Attributes

- 2-operand format, where instructions have a source and a destination. Register to register, register to memory, and memory to register commands. Multiple addressing modes for memory, including specialized modes for indexing through arrays
- Variable length instructions where the length often varies according to the addressing mode
- Instructions which require multiple clock cycles to execute.

E.g. Pentium is considered a modern CISC processor

- Complex instruction-decoding logic, driven by the need for a single instruction to support multiple addressing modes.

- A small number of general purpose registers. This is the direct result of having instructions which can operate directly on memory and the limited amount of chip space not dedicated to instruction decoding, execution, and microcode storage.

- Several special purpose registers. Many CISC designs set aside special registers for the stack pointer, interrupt handling, and so on. This can simplify the hardware design somewhat, at the expense of making the instruction set more complex.

- A 'Condition code'' register which is set as a side-effect of most instructions. This register reflects whether the result of the last operation is less than, equal to, or greater than zero and records if certain error conditions occur.

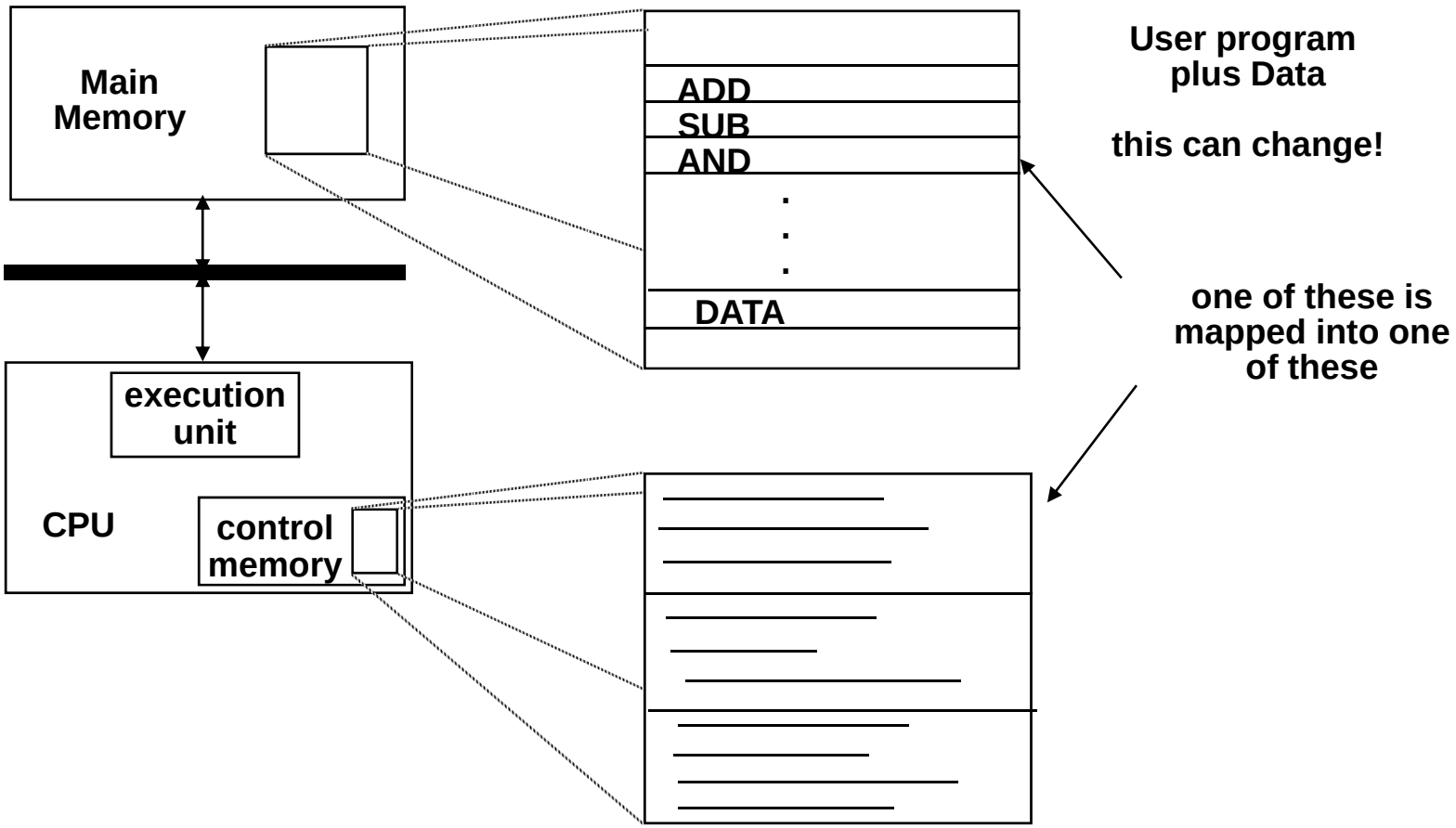
At the time of their initial development, CISC machines used available technologies to optimize computer performance.

- Microprograming is as easy as assembly language to implement, and much less expensive than hardwiring a control unit.

- The ease of microcoding new instructions allowed designers to make CISC machines upwardly compatible: a new computer could run the same programs as earlier computers because the new computer would contain a superset of the instructions of the earlier computers.

- Because microprogram instruction sets can be written to match the constructs of high-level languages, the compiler does not have to be as complicated.

Microprogramming



Supported complex instructions a sequence of simple micro-inst

CISC Disadvantages

- Earlier generations of a processor family generally were contained as a subset in every new version - so instruction set & chip hardware become more complex with each generation of computers.

- So that as many instructions as possible could be stored in memory with the least possible wasted space, individual instructions could be of almost any length - this means that different instructions will take different amounts of clock time to execute, slowing down the overall performance of the machine.

- Many specialized instructions aren't used frequently enough to justify their existence -approximately 20% of the available instructions are used in a typical program.

- CISC instructions typically set the condition codes as a side effect of the instruction. Not only does setting the condition codes take time, but programmers have to remember to examine the condition code bits before a subsequent instruction changes them.

What is RISC?

- RISC, or *Reduced Instruction Set Computer*. is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

- About 80% of the computations of a typical program required only about 20% of the instructions in a processor's instruction set. The most frequently used instructions were simple instructions such as load, store and add.

- Certain design features have been characteristic of most RISC processors:
  - ***one cycle execution time***: RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called PIPELINING
  - ***pipelining*: a techique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;**
  - ***large number of registers*:** the RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

RISC Attributes

The main characteristics of CISC microprocessors are:

- Extensive instructions.
- Complex and efficient machine instructions.
- Microencoding of the machine instructions.
- Extensive addressing capabilities for memory operations.
- Relatively few registers.

In comparison, RISC processors are more or less the opposite of the above:

- Reduced instruction set.
- Less complex, simple instructions.
- Hardwired control unit and machine instructions.
- Few addressing schemes for memory operands with only two basic instructions, LOAD and STORE
- Many symmetric registers which are organized into a register file.

CISC versus RISC

CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers