

Embedded Systems

1. Introduction

Lothar Thiele

Organization

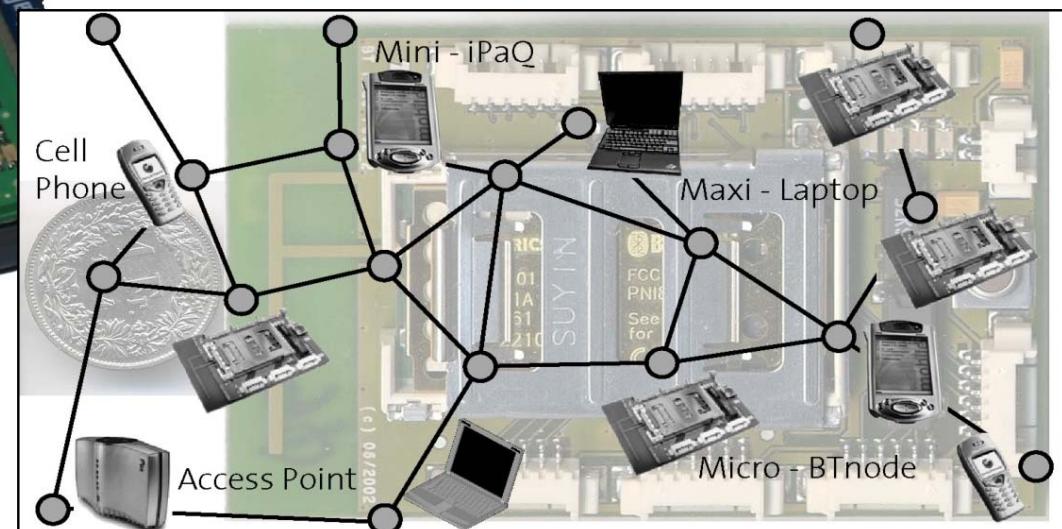
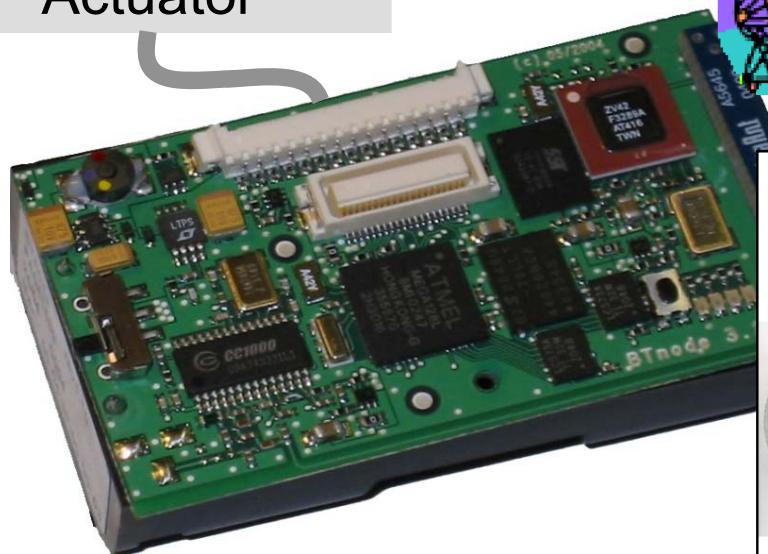
- ▶ **WWW:** <http://www.tik.ee.ethz.ch/tik/education/lectures/ES/>
- ▶ **Lecture:** Lothar Thiele, thiele@ethz.ch
- ▶ **Coordination:** Olga Saukh, olga.saukh@tik.ee.ethz.ch
- ▶ **References:**
 - **P. Marwedel: *Embedded System Design (paperback), Springer Verlag, December 2011, ISBN: 978-94-007-0256-1.***
 - **G.C. Buttazzo: *Hard Real-Time Computing Systems. Springer Verlag, 2011.***
 - W. Wolf: Computers as Components – Principles of Embedded System Design. Morgan Kaufman Publishers, 2012.
 - J. Teich: Digitale Hardware/Software Systeme, Springer Verlag, 2007.
- ▶ The slides contain material of J. Rabaey, K. Keutzer, Wayne Wolf, Peter Marwedel, Philip Koopman and from the above books of J. Teich, G.C. Buttazzo, W. Wolf and P. Marwedel.

Communicating Embedded Systems

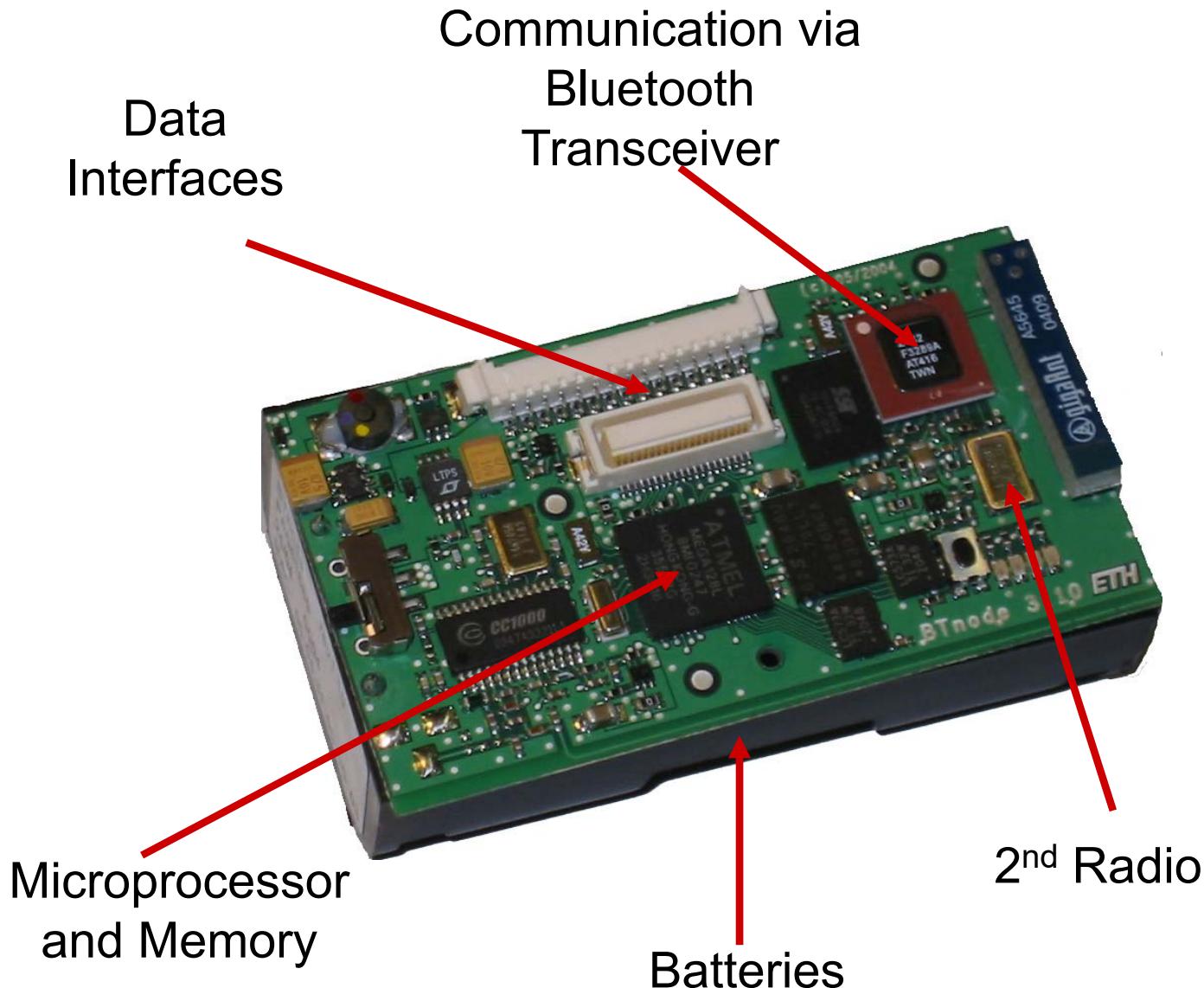
- ▶ Example: BTnodes
 - complete platform including OS
 - especially suited for pervasive computing applications



Sensor
Actuator



BTnode Platform



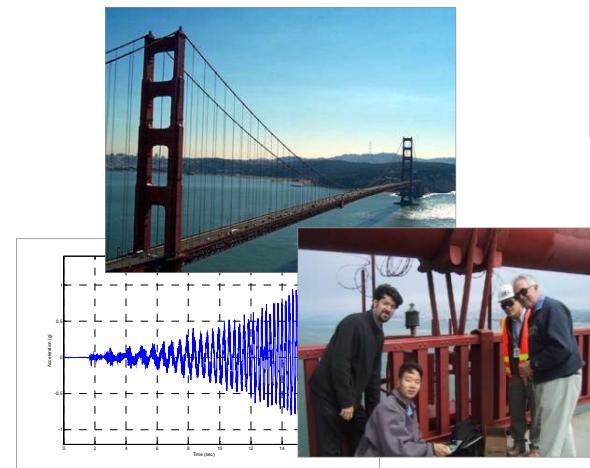
- ▶ generic platform for wireless distributed embedded computing
- ▶ complete platform including OS
- ▶ especially suited for pervasive computing applications (IoT)

Where are sensor nodes used?

Logistics



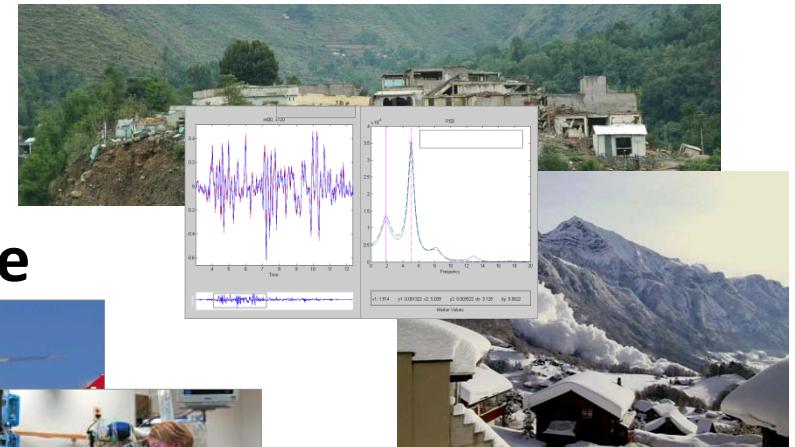
Maintenance



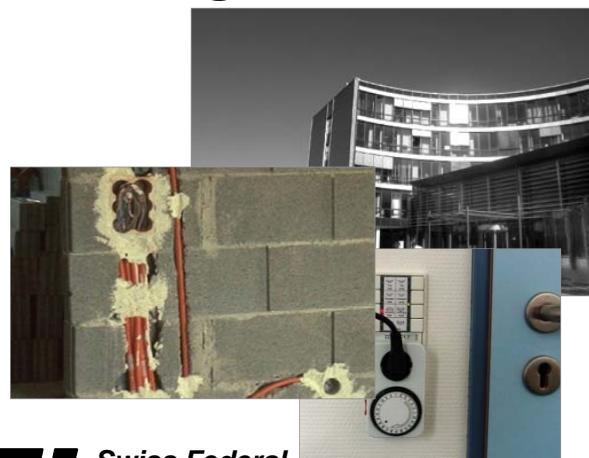
Factory Automation



Natural Hazards



Building Automation

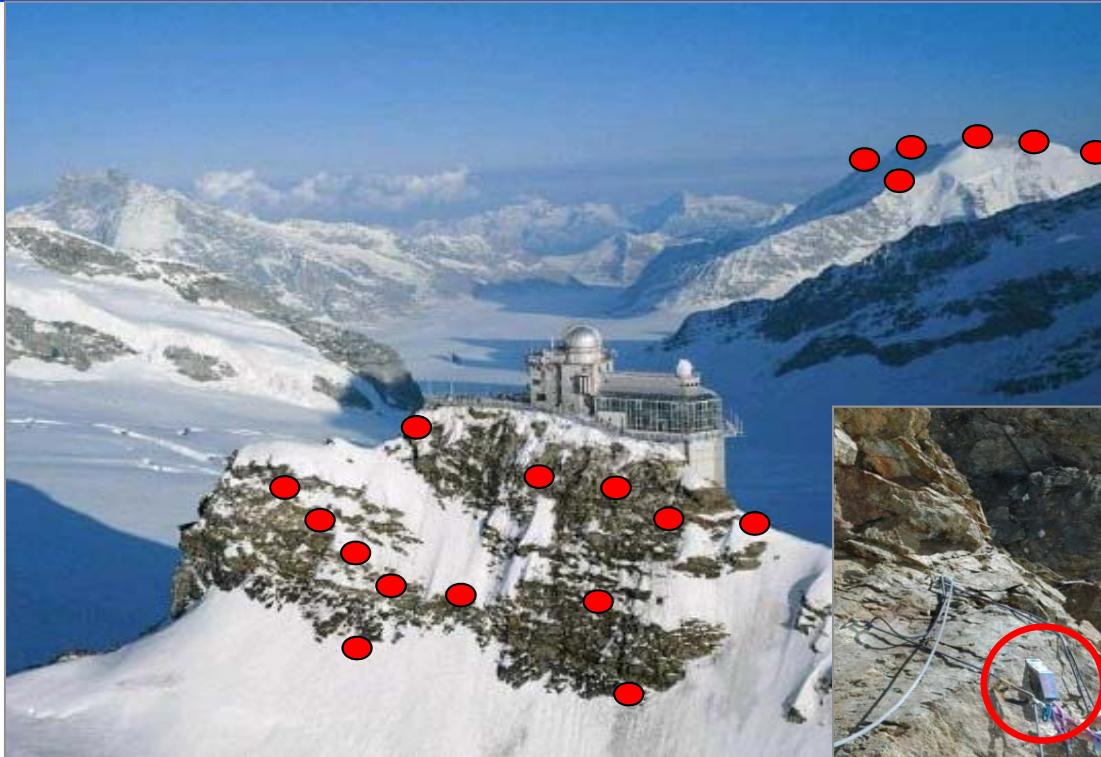


Health Care



1 - 5

PermaSense Project



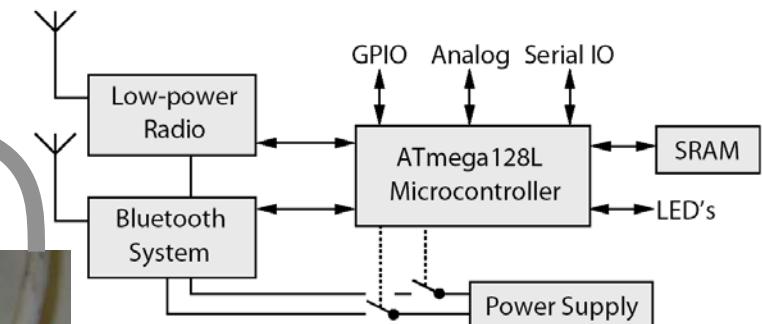
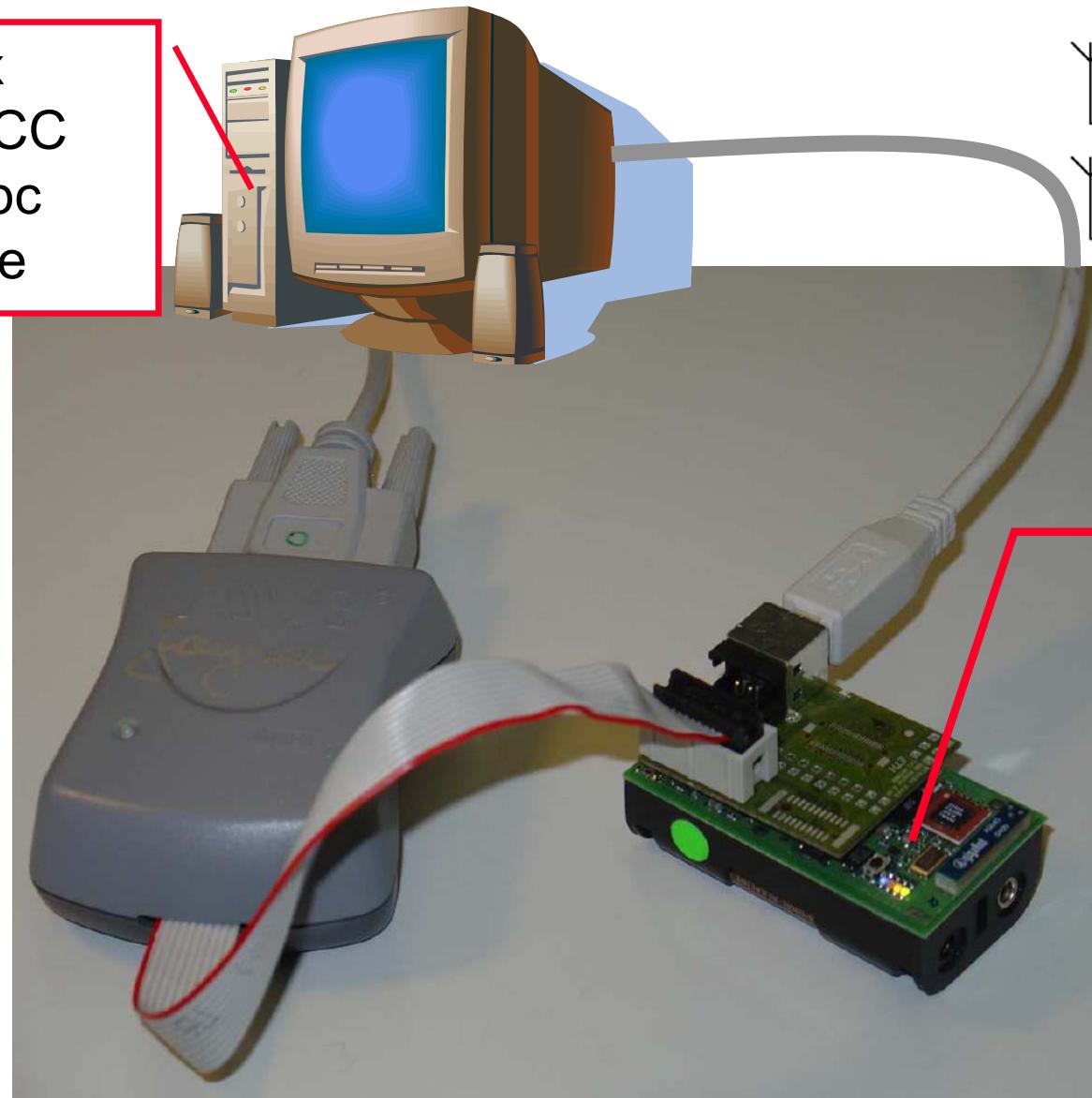
▶ Jan Beutel – ETH Zurich



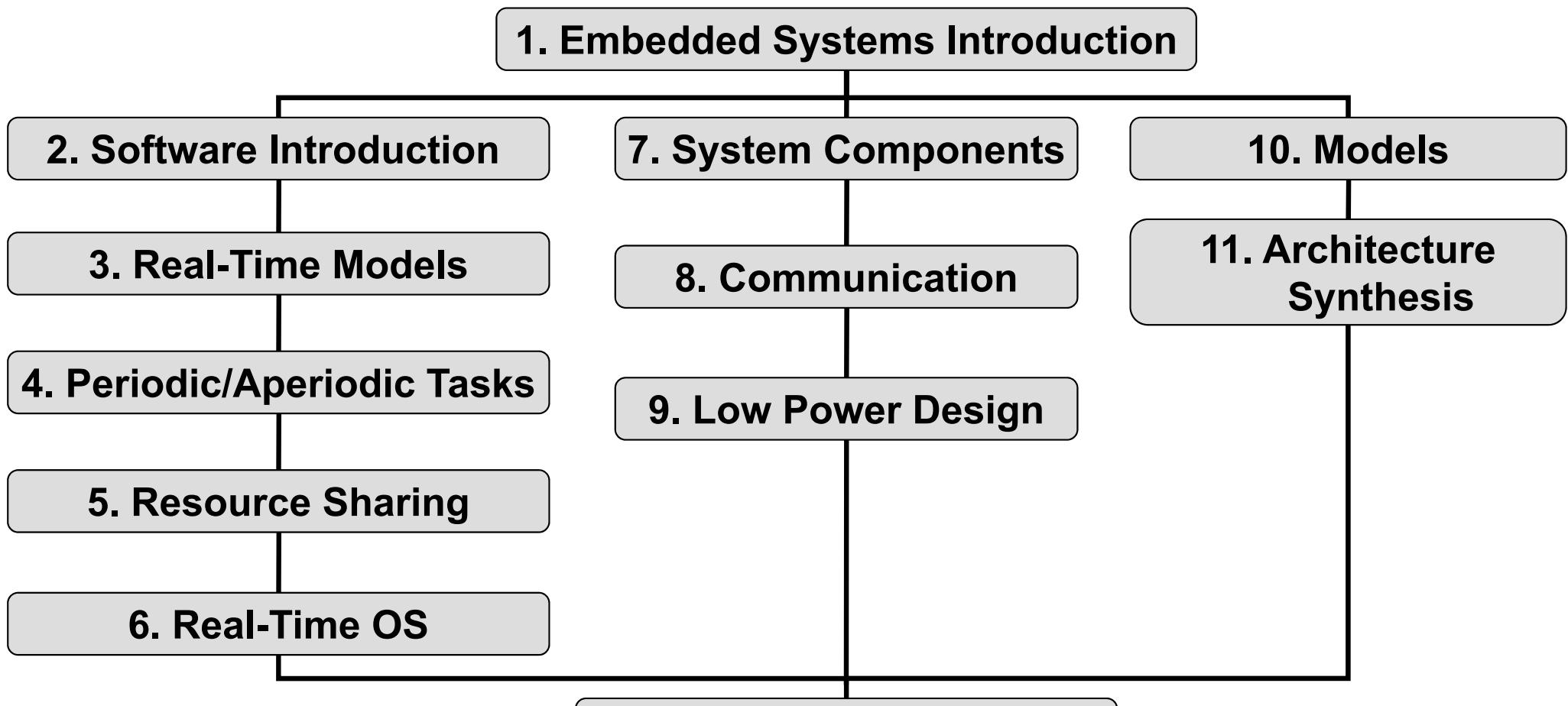


Development in ES Exercise

Linux
GNU GCC
AVR libc
Eclipse



Contents of Course

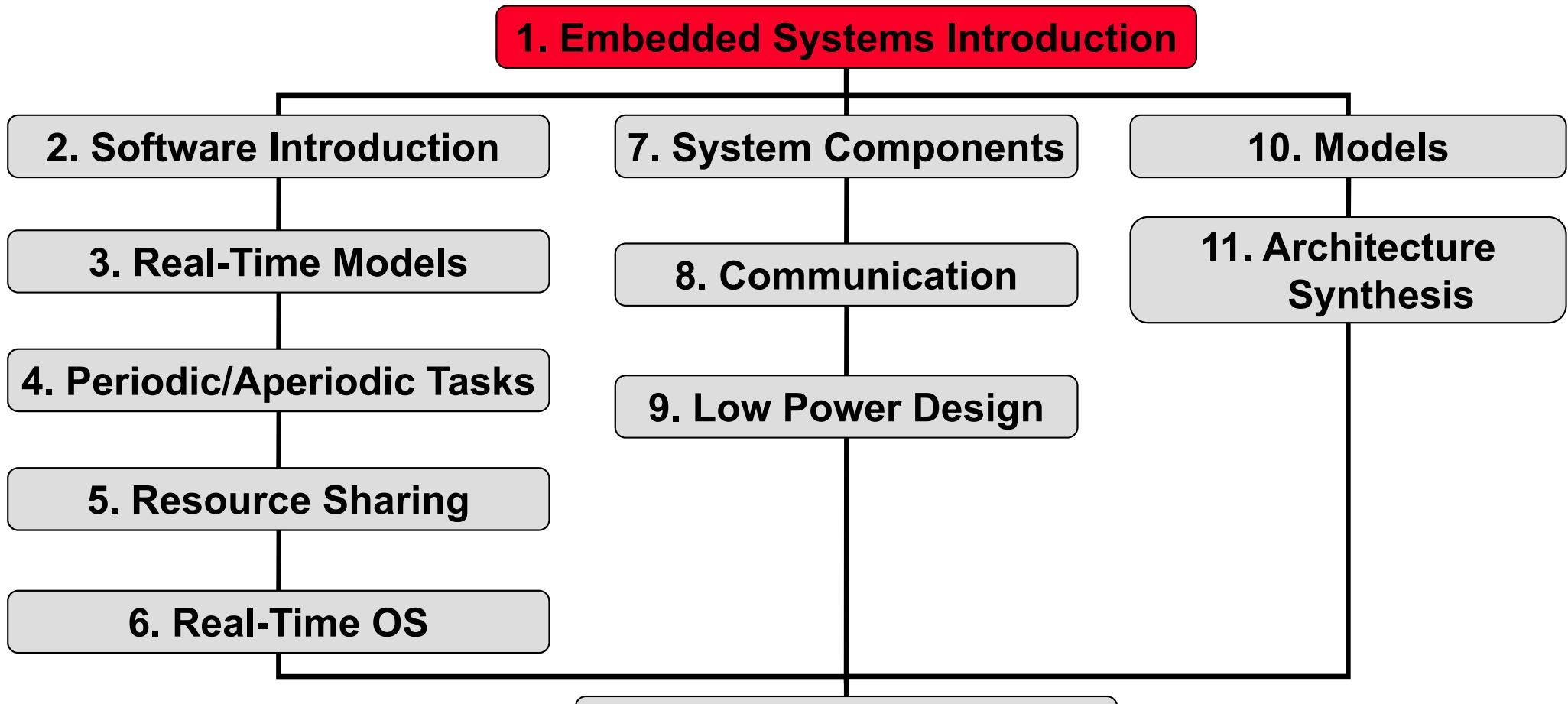


*Software and
Programming*

*Processing and
Communication*

Hardware

Contents of Course

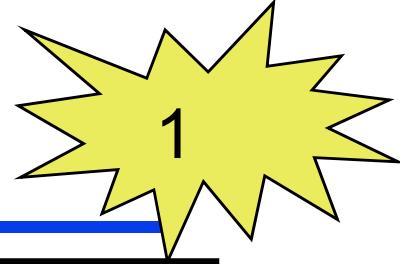


*Software and
Programming*

*Processing and
Communication*

Hardware

Embedded Systems



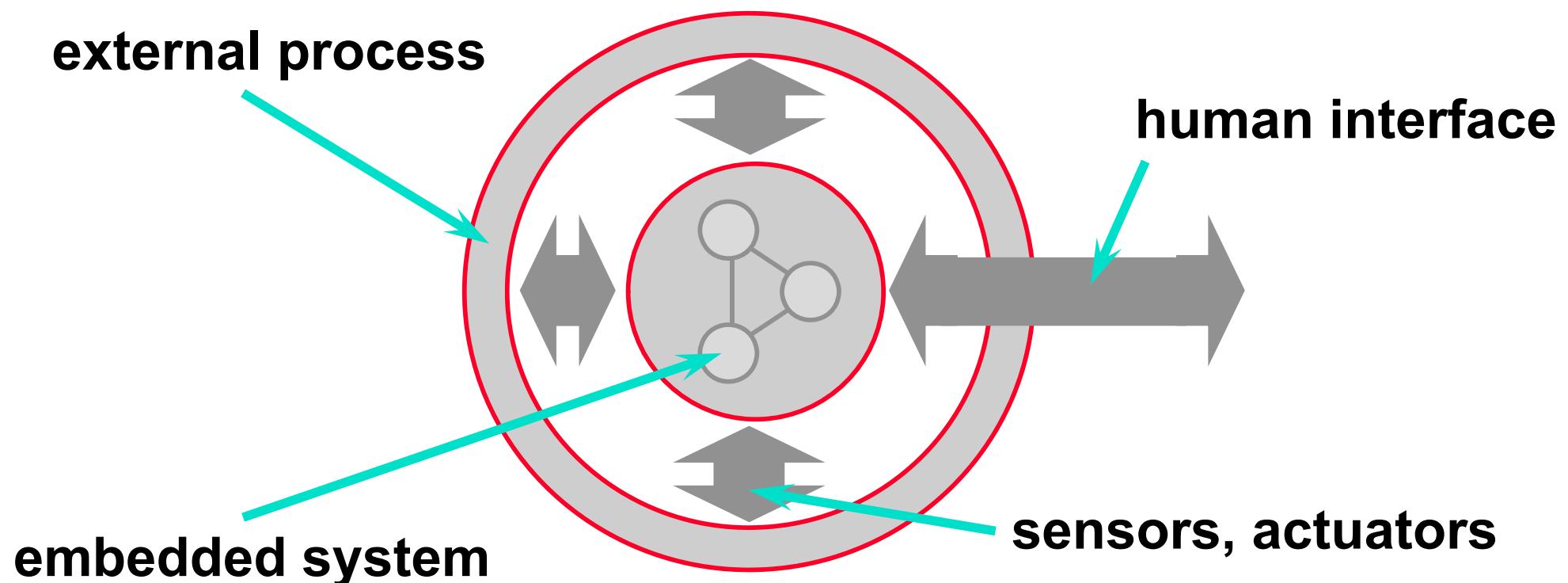
Embedded systems (ES) = **information processing systems embedded into a larger product**

Examples:



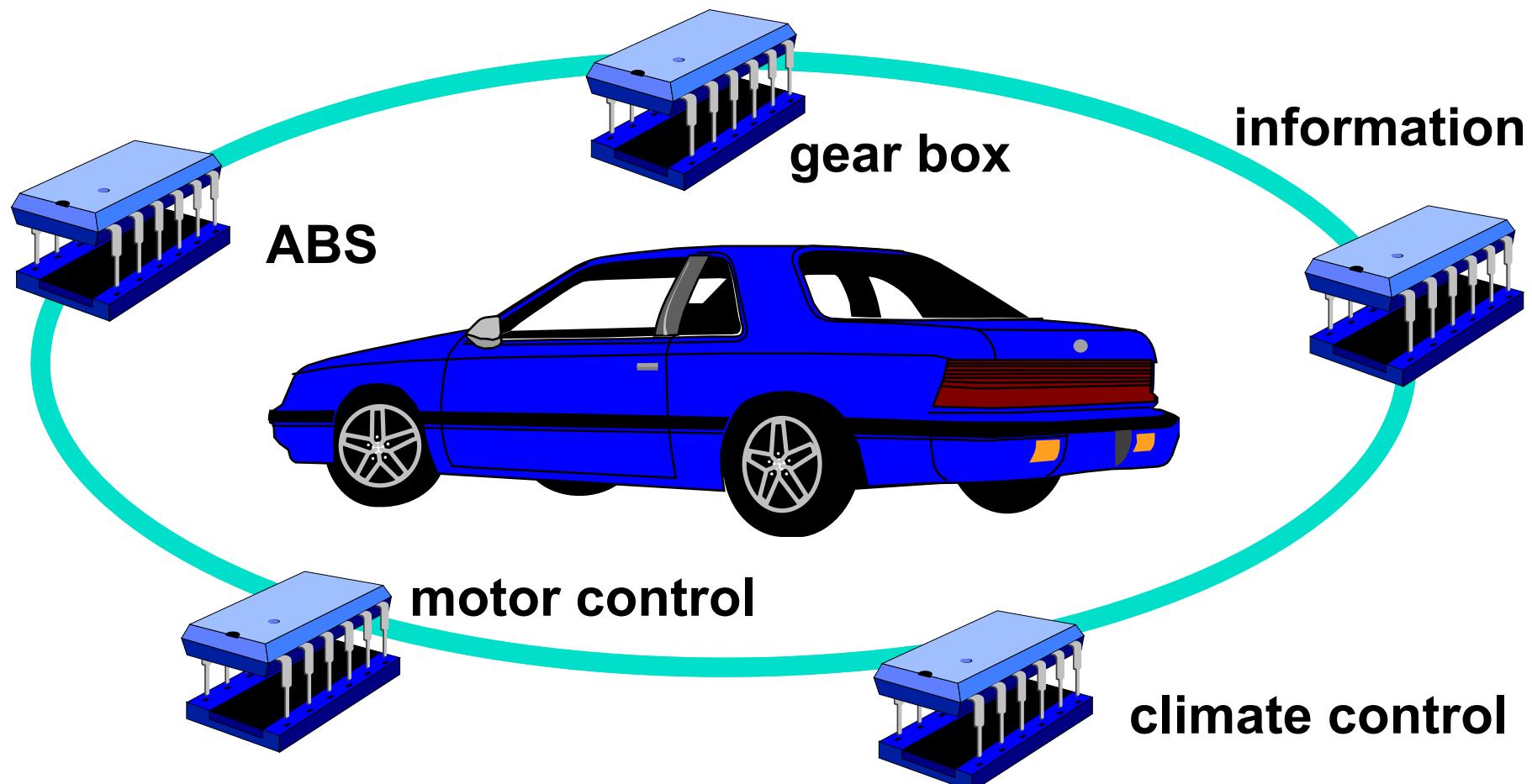
Main reason for buying often is **not** information processing

Embedded Systems



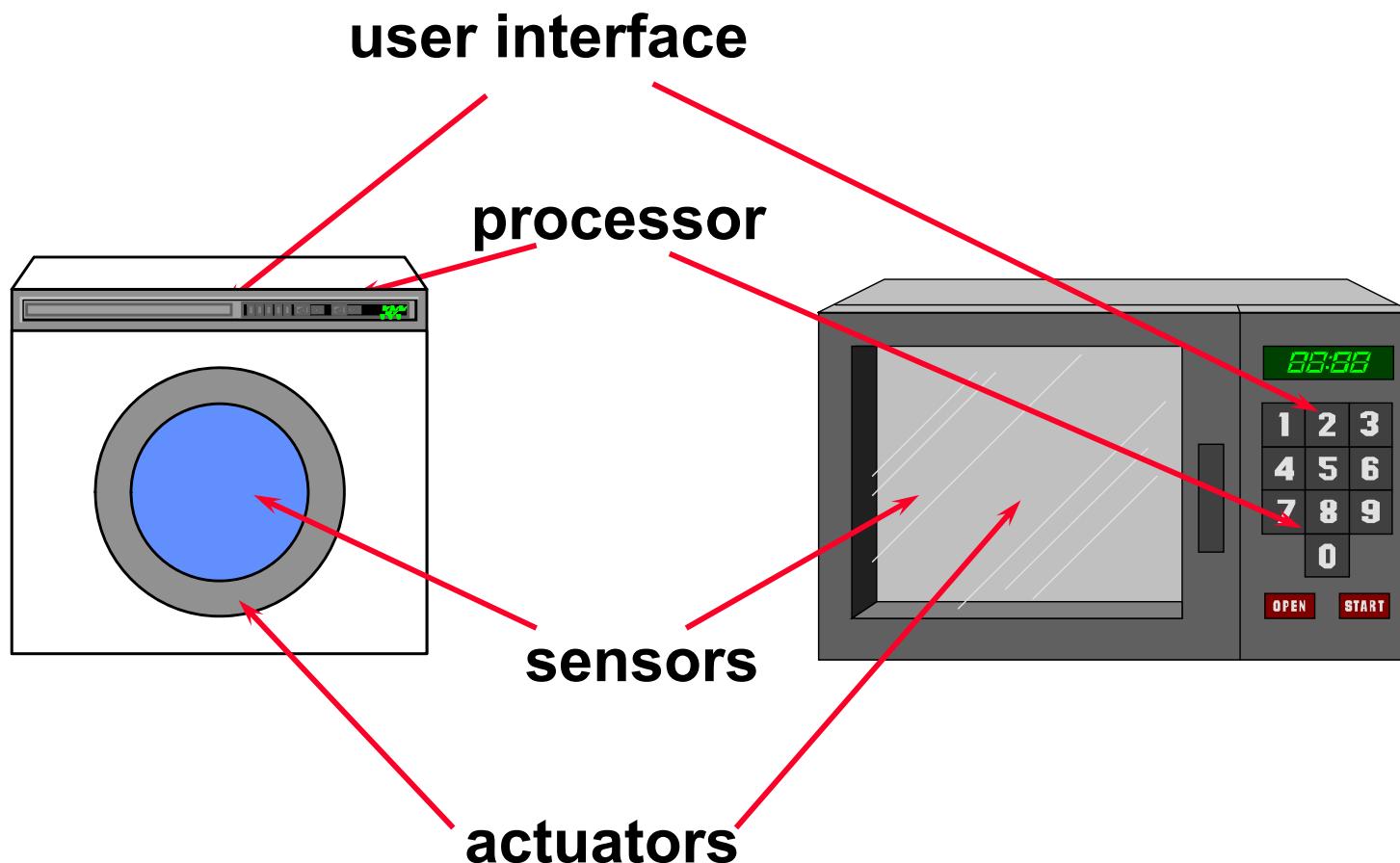
Examples of Embedded Systems

Car as an integrated control-, communication and information system.



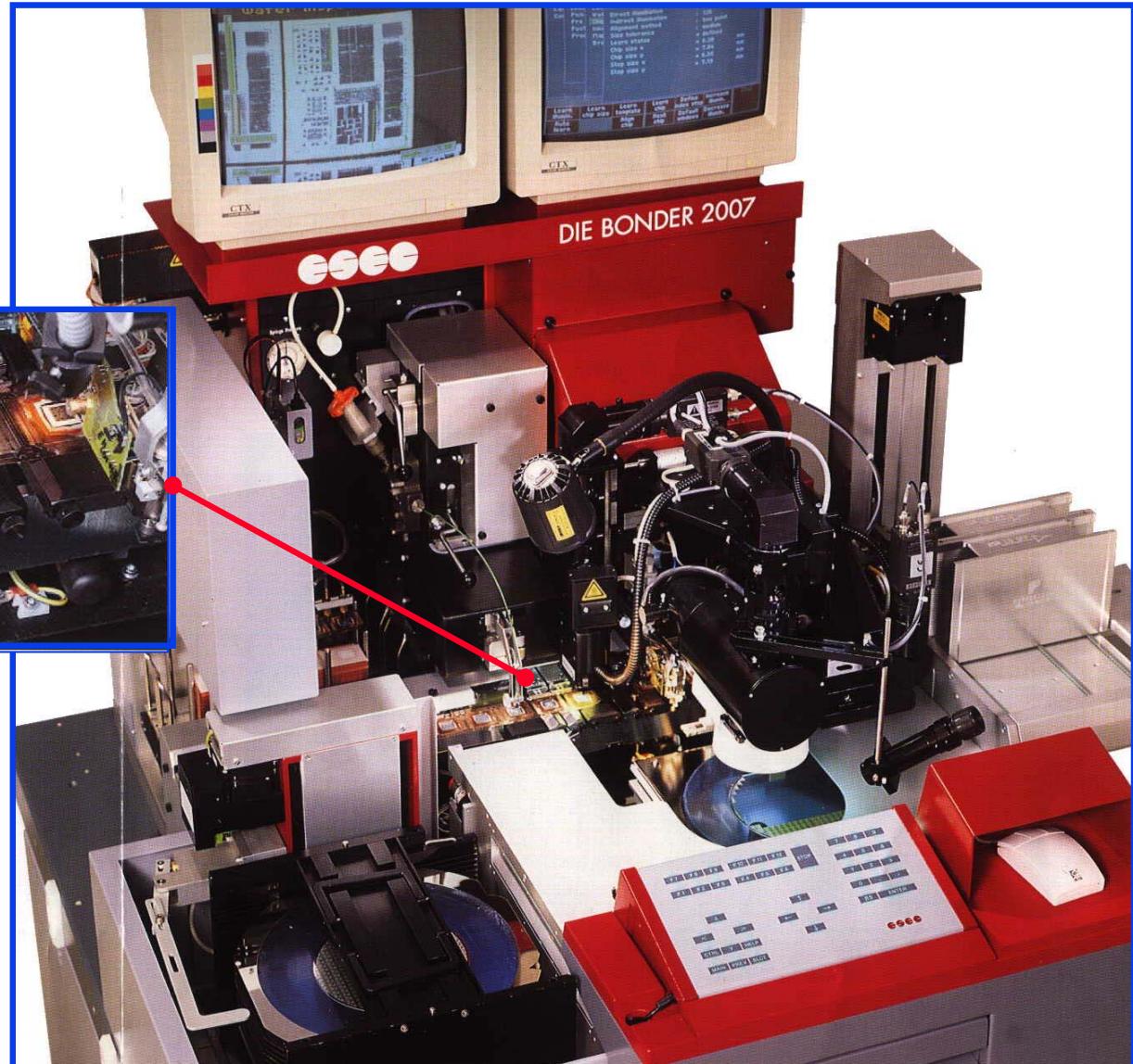
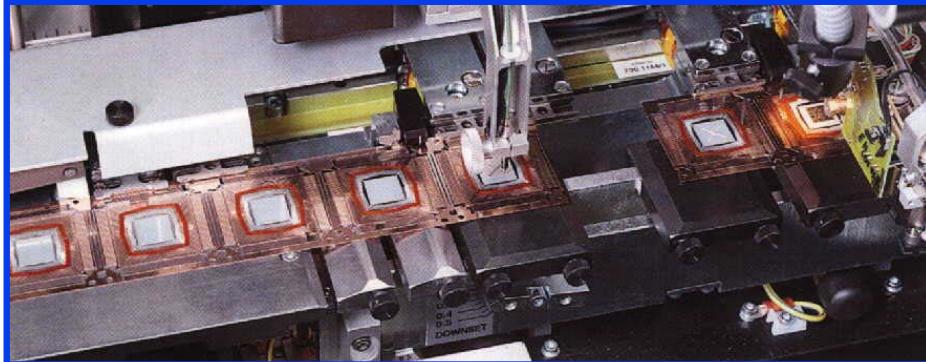
Examples of Embedded Systems

Consumer electronics, for example MP3 Audio, digital camera, home electronics, . . .



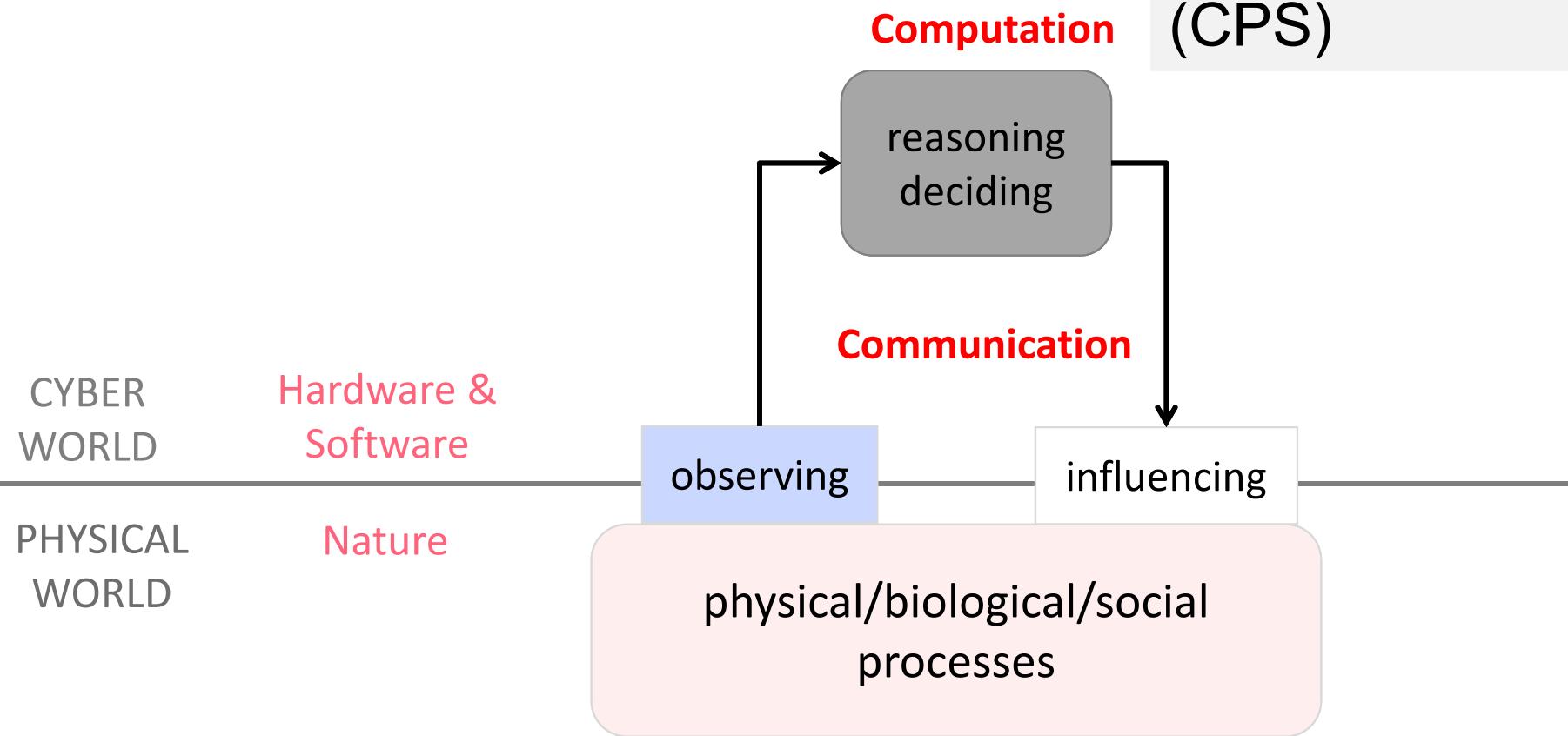
Examples of Embedded Systems

Production systems

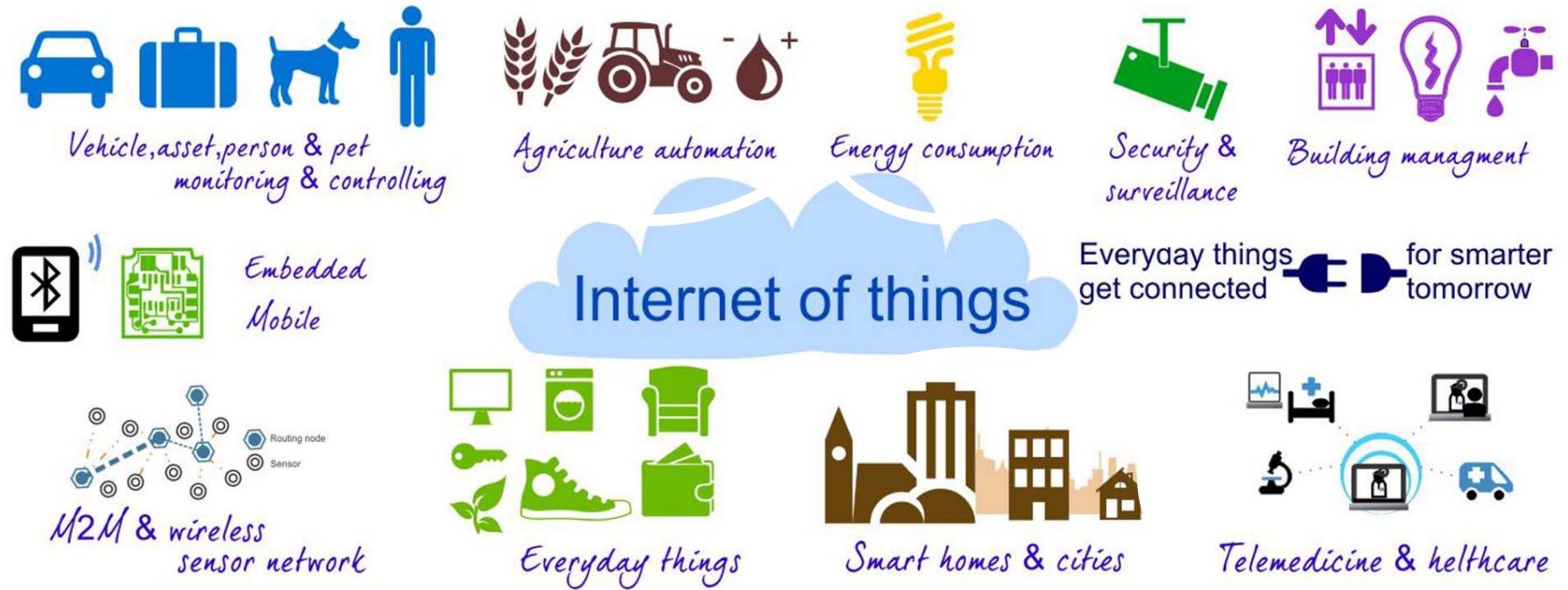


Smart World

Sometimes denoted as:
cyber-physical system
(CPS)



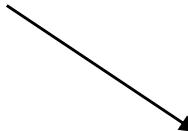
Use feedback to influence the dynamics of the physical world by taking smart decisions in the cyber world



Internet of Things (IOT) infrastructure will be omnipresent.

Predictability & Dependability

CPS = cyber-physical system



“It is essential to **predict** how a CPS is going to behave under any circumstances [...] **before** it is deployed.”^{Maj14}

“CPS must **operate dependably**, safely, securely, efficiently and in real-time.”^{Raj10}

^{Maj14} R. Majumdar & B. Brandenburg (2014). Foundations of Cyber-Physical Systems.

^{Raj10} R. Rajkumar et al. (2010). Cyber-Physical Systems: The Next Computing Revolution.

Characteristics of Embedded Systems (1)

- ▶ Must be **dependable**:
 - **Reliability:** $R(t)$ = probability of system working correctly provided that it was working at $t=0$
 - **Maintainability:** $M(d)$ = probability of system working correctly d time units after error occurred.
 - **Availability:** probability of system working at time t
 - **Safety:** no harm to be caused
 - **Security:** confidential and authentic communication

Making the system dependable must not be an after-thought, it must be considered from the very beginning.

Characteristics of Embedded Systems (2)

- ▶ Must be **efficient**:
 - **Energy** efficient
 - **Code-size** and **data memory** efficient
 - **Run-time** efficient
 - **Weight** efficient
 - **Cost** efficient
- ▶ **Dedicated** towards a certain **application**: Knowledge about behavior at design time can be used to minimize resources and to maximize robustness.

Characteristics of Embedded Systems (3)

- ▶ Many ES must meet ***real-time constraints***:
 - A real-time system must ***react to stimuli*** from the controlled object (or the operator) within the time interval dictated by the environment.
 - For real-time systems, right answers arriving too late are wrong.

„A real-time constraint is called hard, if not meeting that constraint could result in a catastrophe“ [Kopetz, 1997].

- All other time-constraints are called soft.
- A ***guaranteed system response*** has to be explained without statistical arguments.

Characteristics of Embedded Systems (4)

- ▶ Typically, ES are **reactive systems**:

„A reactive system is one which is in continual interaction with its environment and executes at a pace determined by that environment“ [Bergé, 1995]

- ▶ Frequently **connected to physical environment** through sensors and actuators (CPS).
- ▶ In these cases, the analog and digital system aspects need to be considered: **hybrid systems**.

Comparison

▶ Embedded Systems

- Few applications that are known at design-time.
- Not programmable by end user.
- Fixed run-time requirements (additional computing power not useful).
- Criteria:
 - cost
 - power consumption
 - predictability
 - ...

▶ General Purpose Computing

- Broad class of applications.
- Programmable by end user.
- Faster is better.
- Criteria:
 - cost
 - power consumption
 - average speed

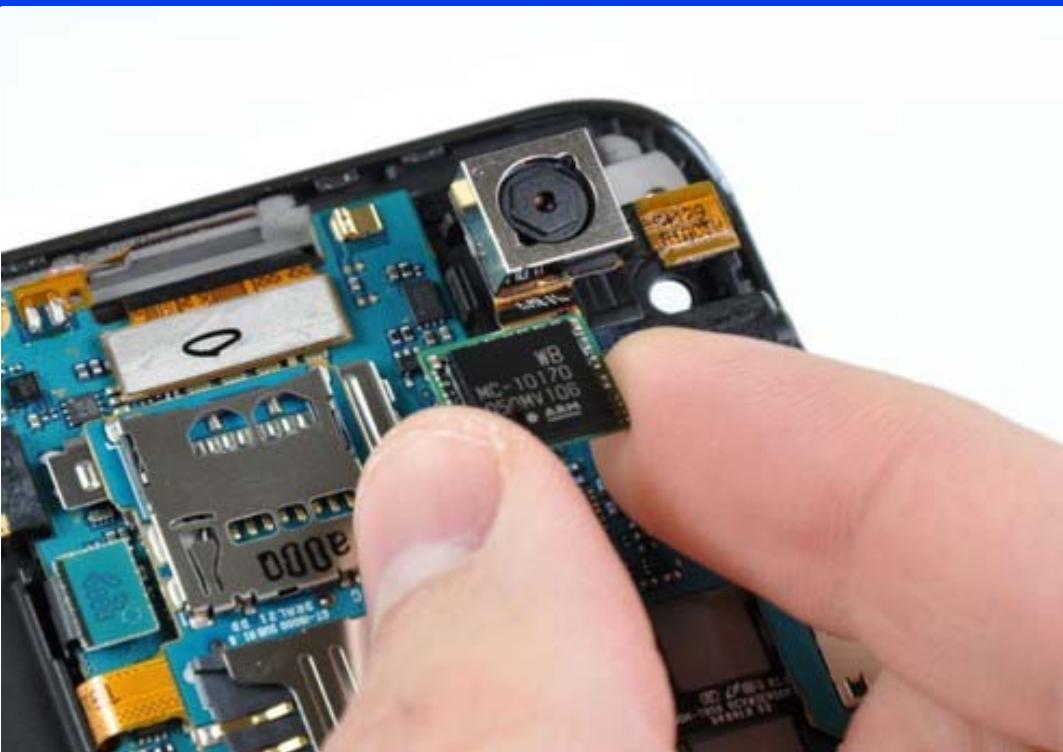
Trends ...

- ▶ Since long time, embedded systems overtook the market of PCs and Laptops.
- ▶ Ubiquitous and pervasive computing, Internet of Things:
 - Information anytime, anywhere; building ambient intelligence into our environment; internet of things:
 - Wearable computers
 - “Smart Labels” on consumer products
 - Intelligent buildings
 - Environmental Monitoring
 - Traffic control and communicating automobiles
 - Embedded systems provide the basic technology.

Trends ...

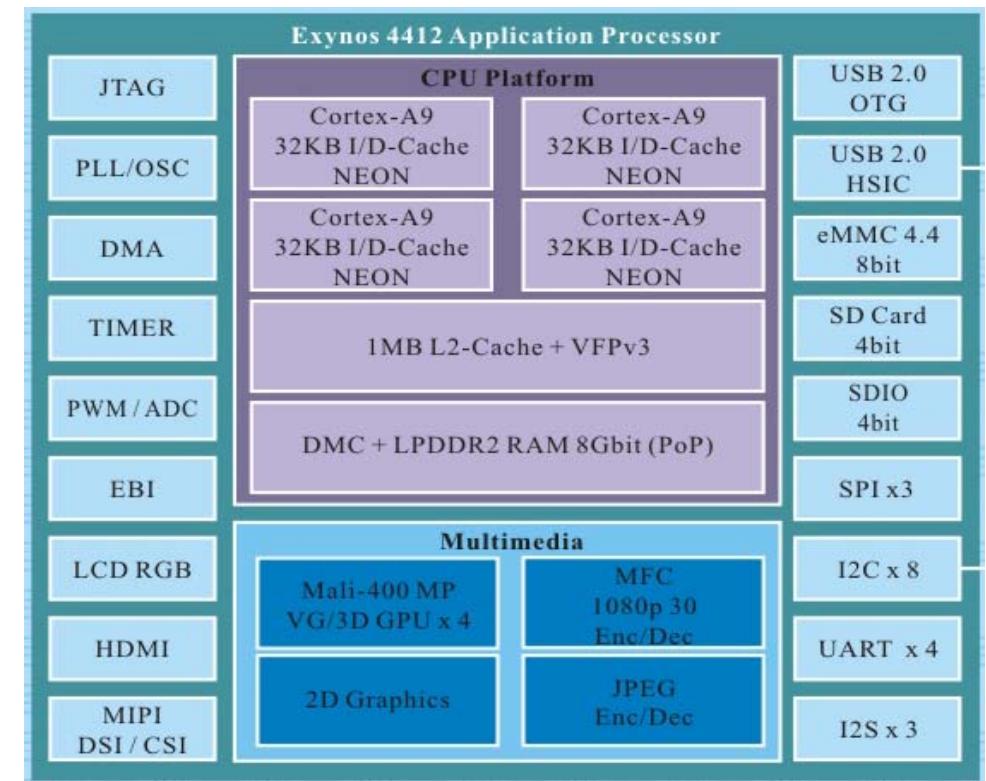
- ▶ Embedded systems are communicating, very often wireless.
- ▶ Higher degree of integration on a single chip:
 - Memory + processor + I/O-units + (wireless) communication.
 - Network on chip for communication between units.
 - Multiprocessor Systems on a Chip (MPSoC).
 - Microsystems that contain energy harvesting, energy storage, sensing, processing and communication (“zero power systems”).
 - Software increasing (amount and complexity).
- ▶ Low power and energy constraints (portable or unattended devices) are increasingly important, as well as temperature constraints (overheating). Increased interest in energy harvesting to achieve long term autonomous operation.

Multiprocessor systems-on-a-chip (MPSoCs)

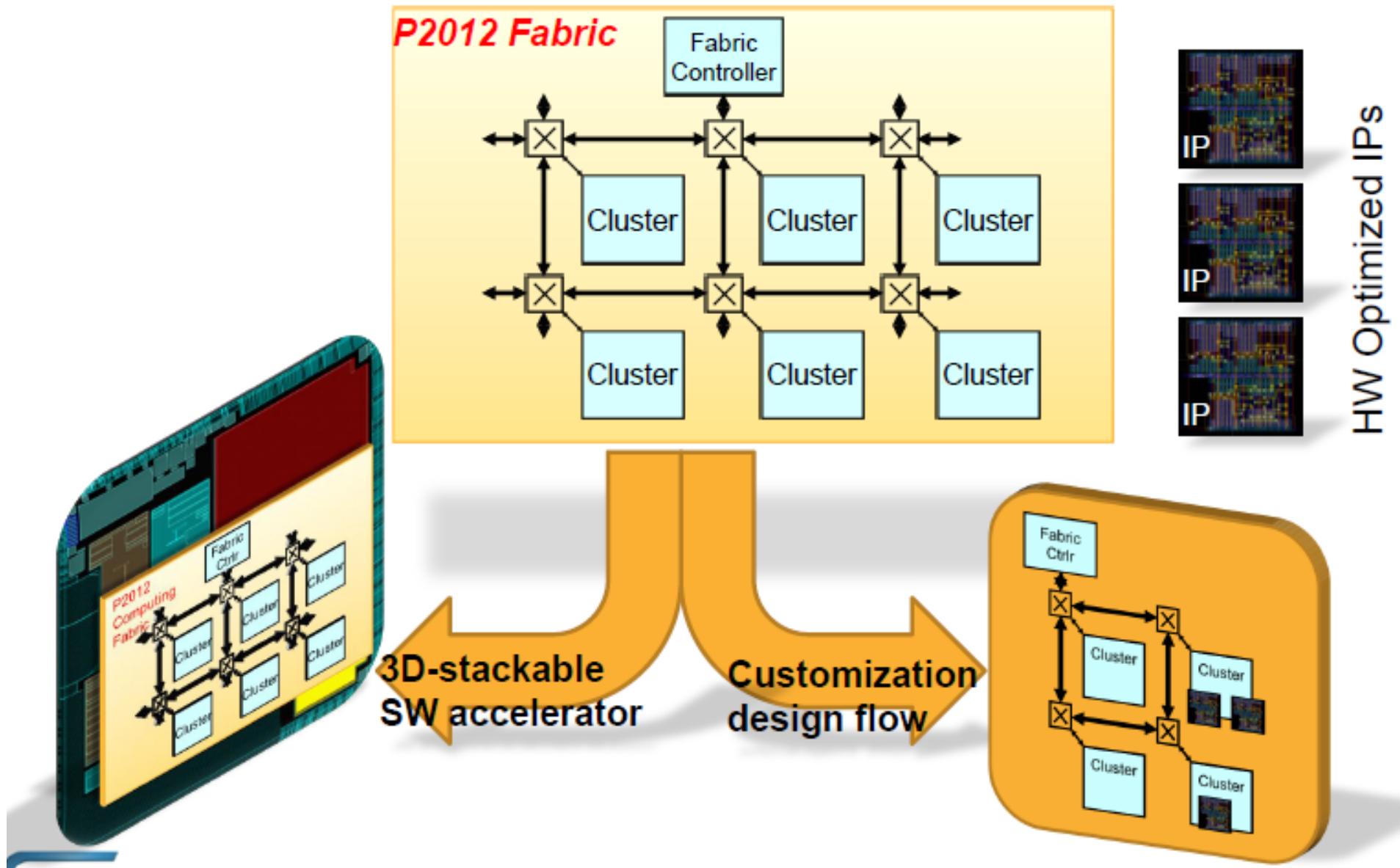


Samsung Galaxy Note II

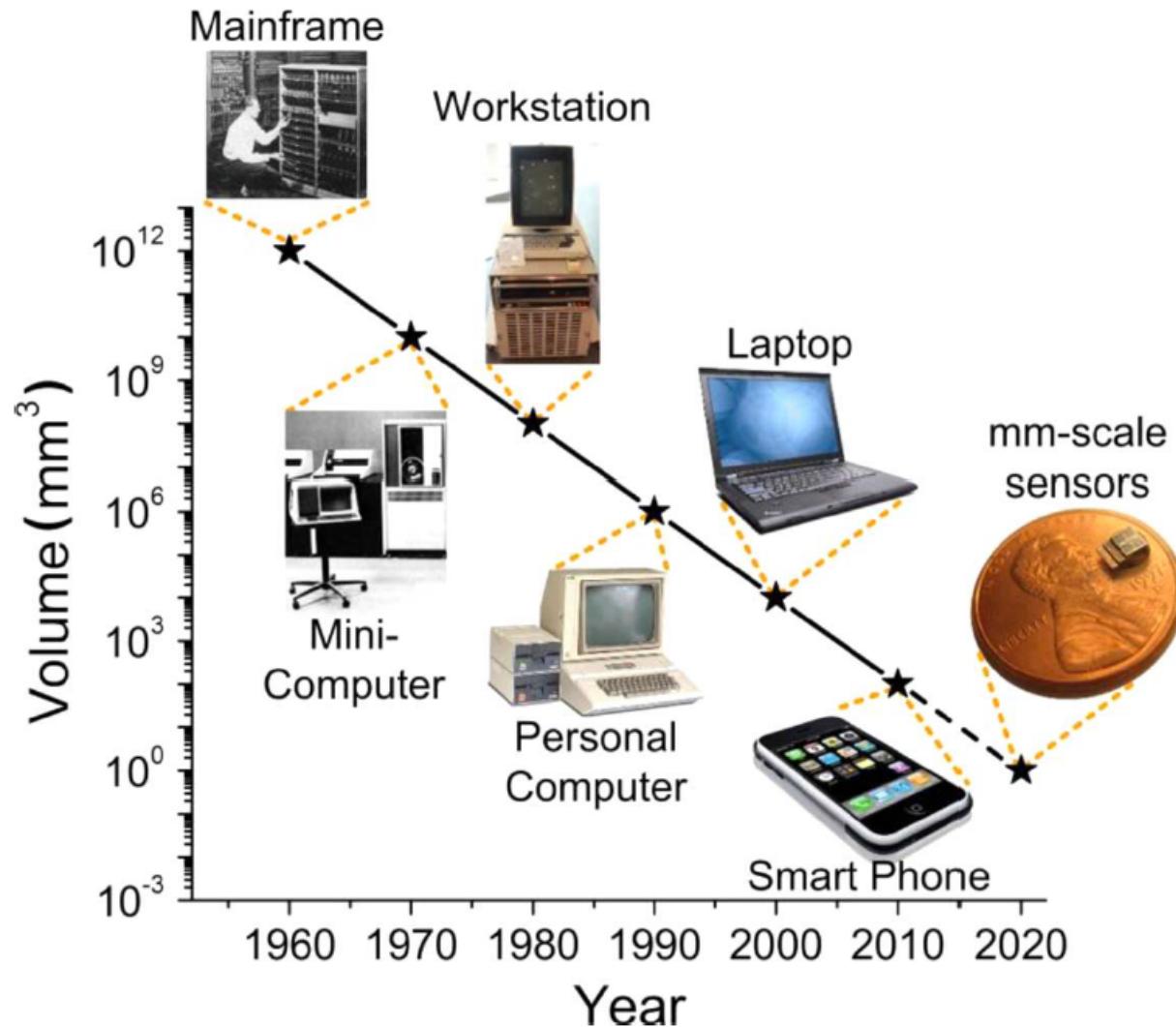
- Eynos 4412 System on a Chip (SoC)
- ARM Cortex-A9 processing core
- 32 nanometer: transistor gate width
- Four processing cores



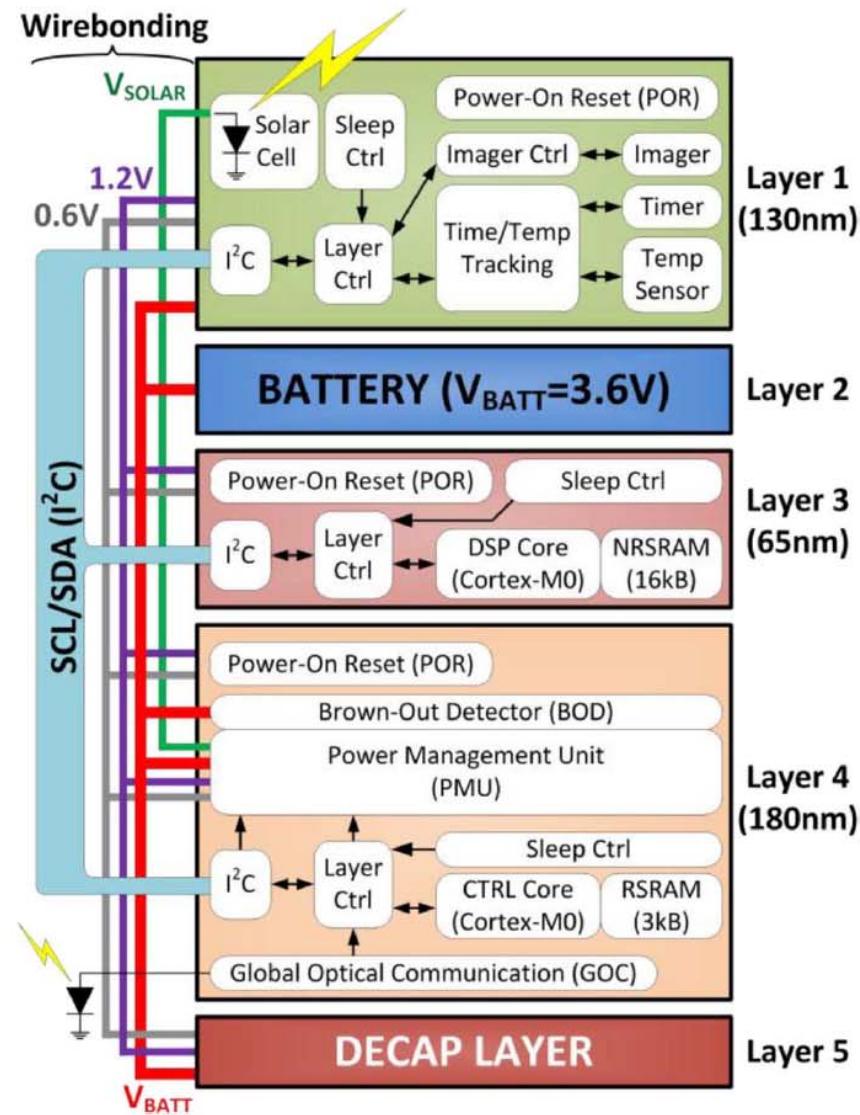
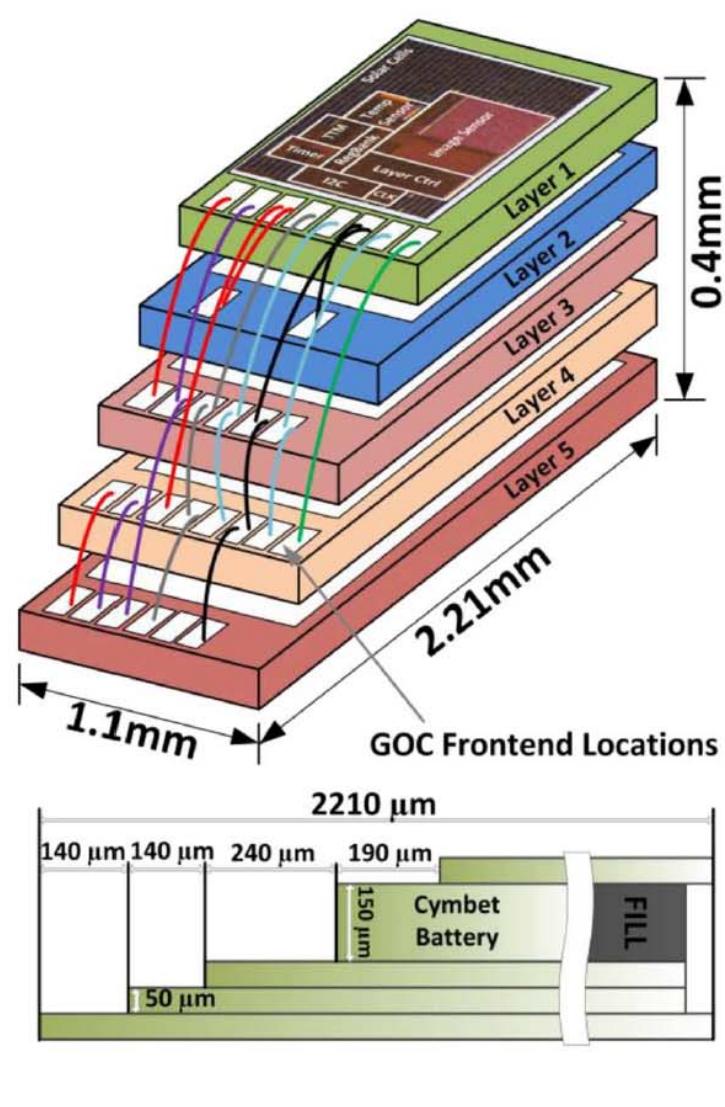
Example ST2012/STHORM



Zero Power Systems and Sensors

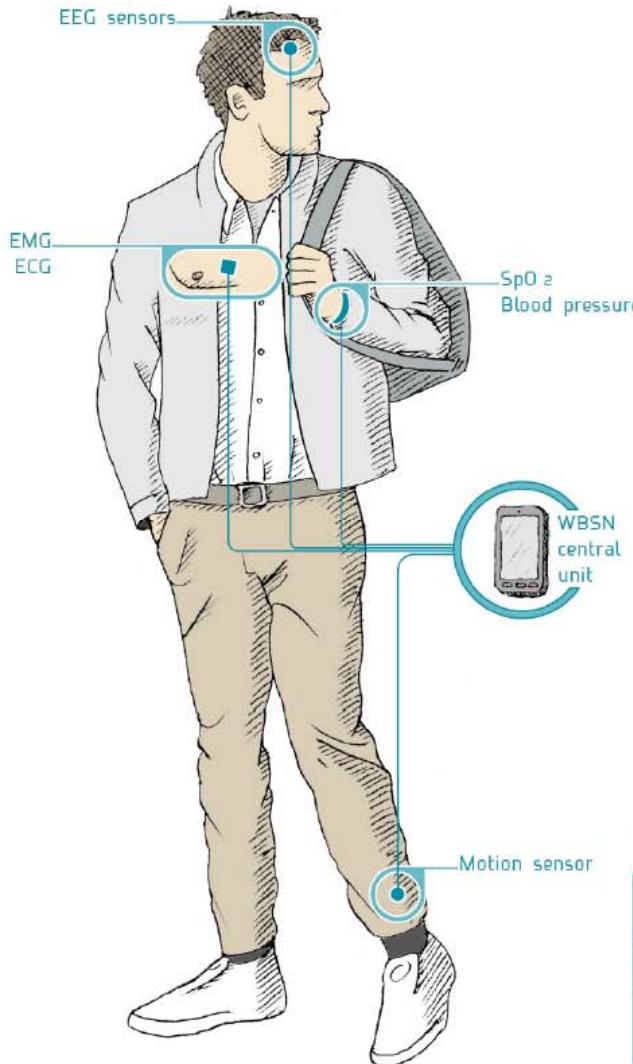


Zero Power Systems and Sensors



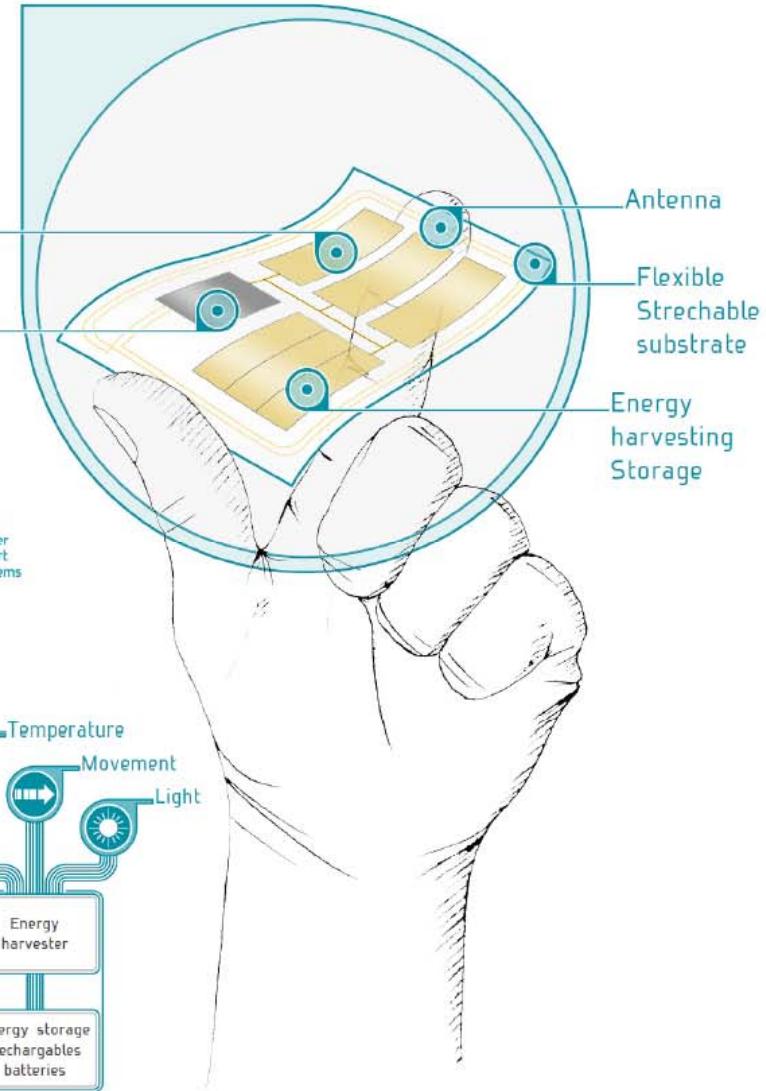
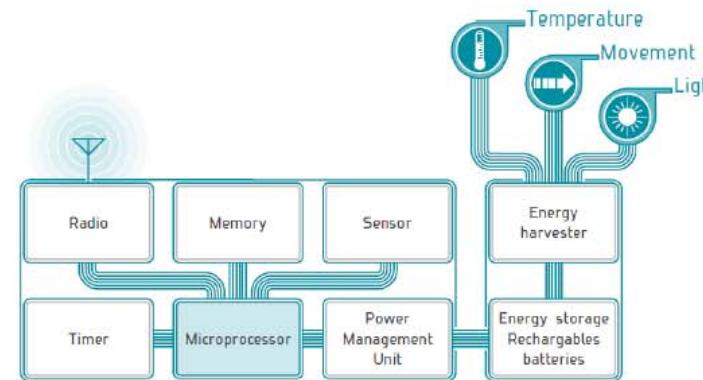
IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 48, NO. 1, JANUARY 2013

Zero Power Systems and Sensors



Sensors:
ECG
EEG
Bio sensors
Accelerometer

Microcontroller with communications capabilities



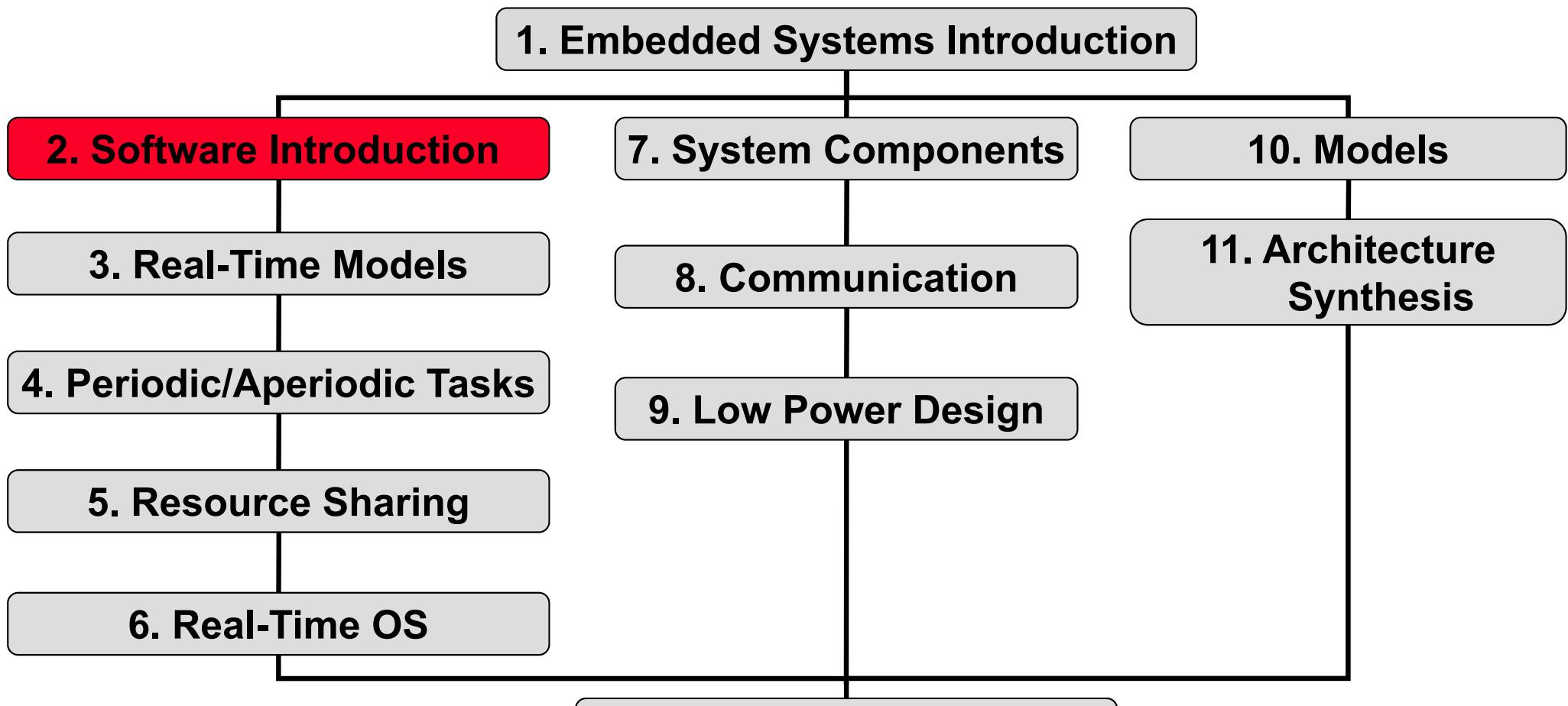
Z-Sys and Guardian Angel Research Proposals 2013

Embedded Systems

2. Software Introduction

Lothar Thiele

Contents of Course

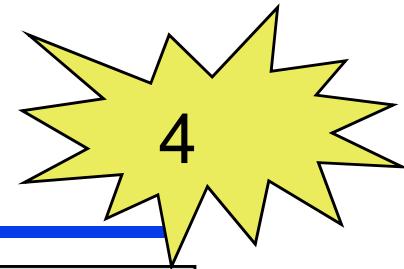


*Software and
Programming*

*Processing and
Communication*

Hardware

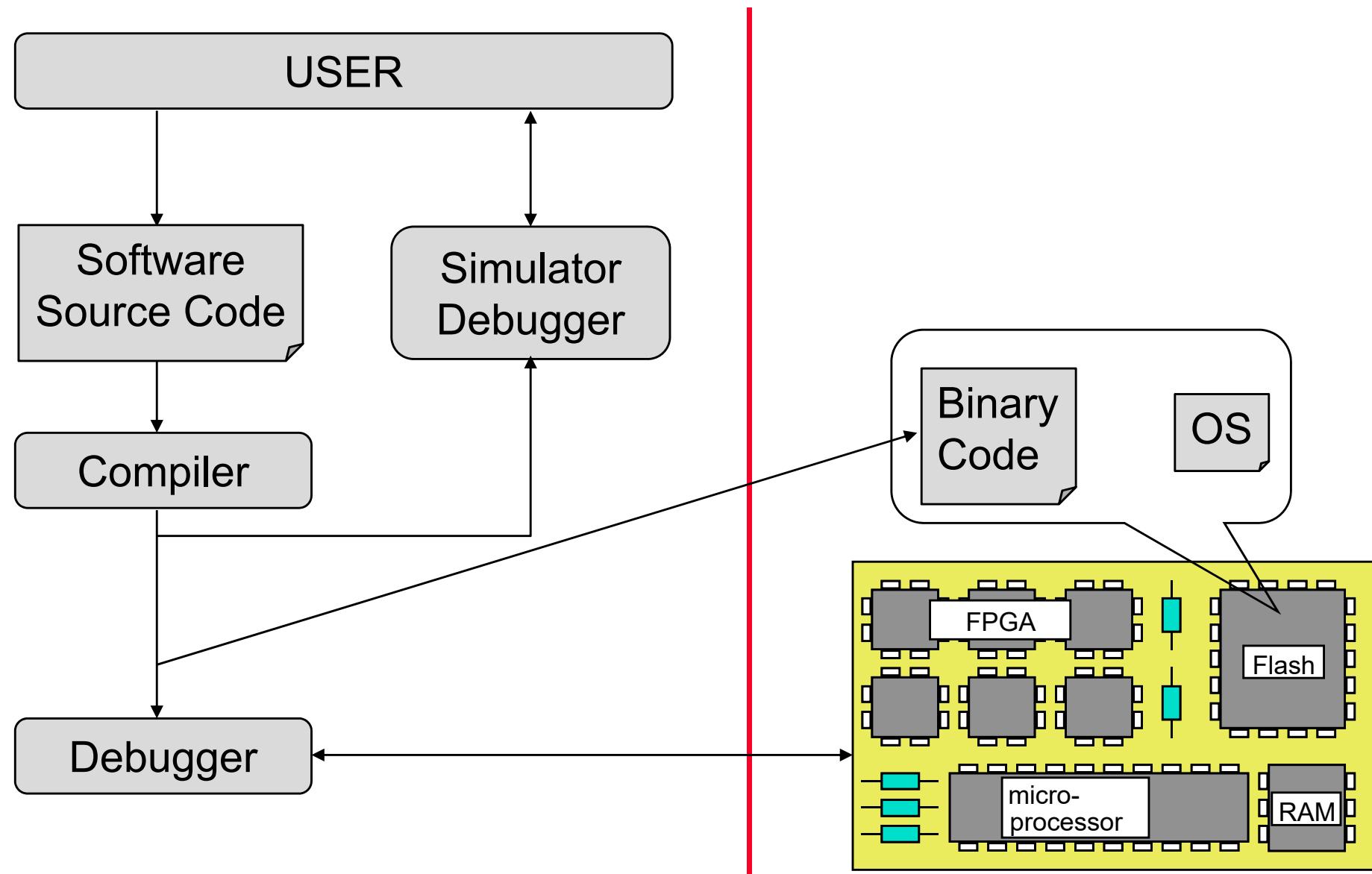
Subtopics



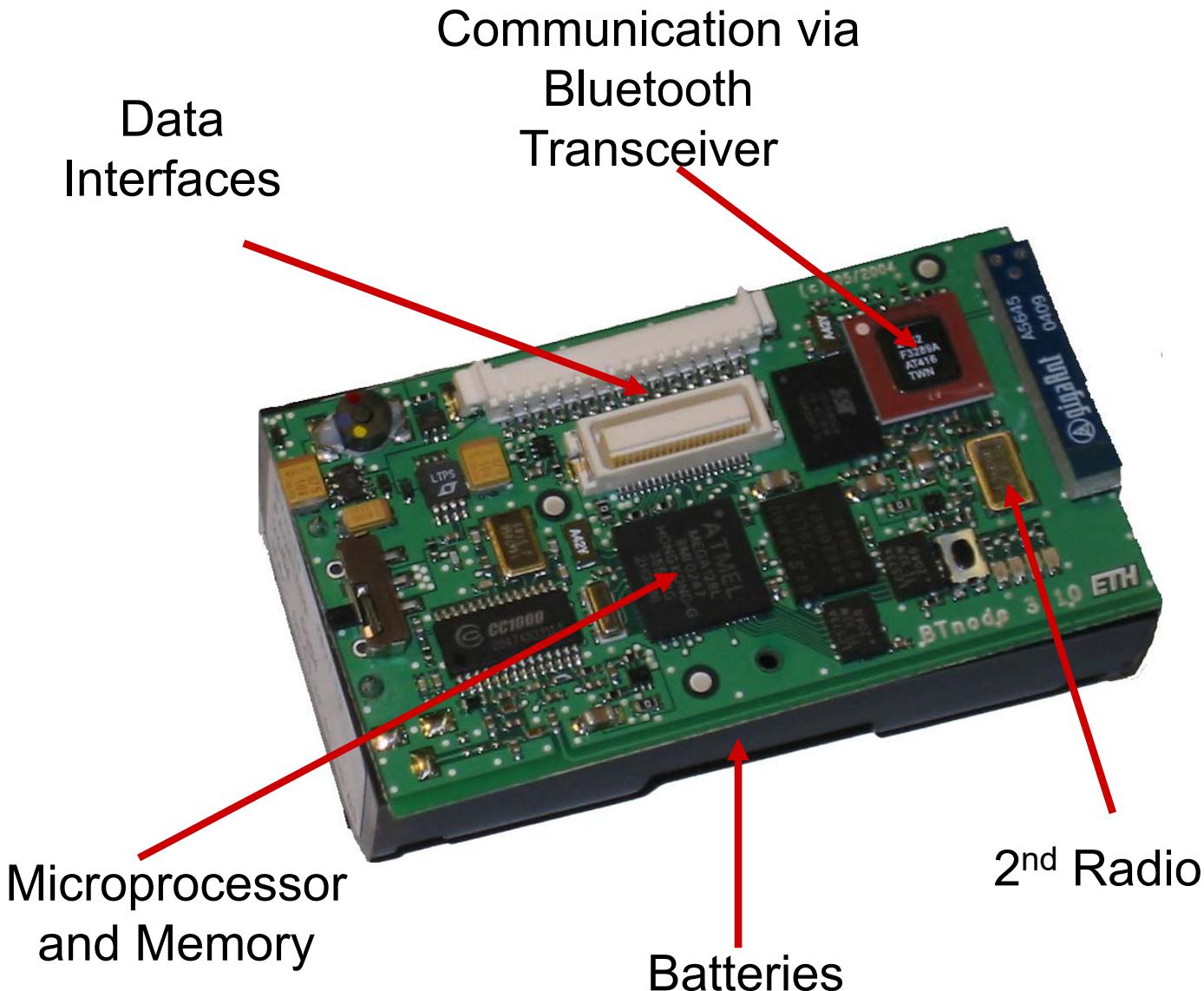
- ▶ A few introductory remarks.

- ▶ Different programming paradigms.

Embedded Software Development



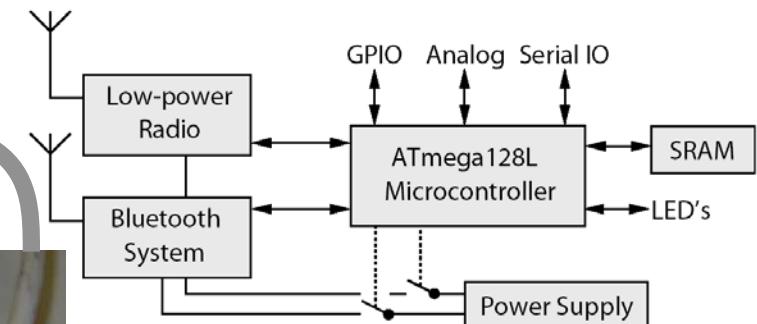
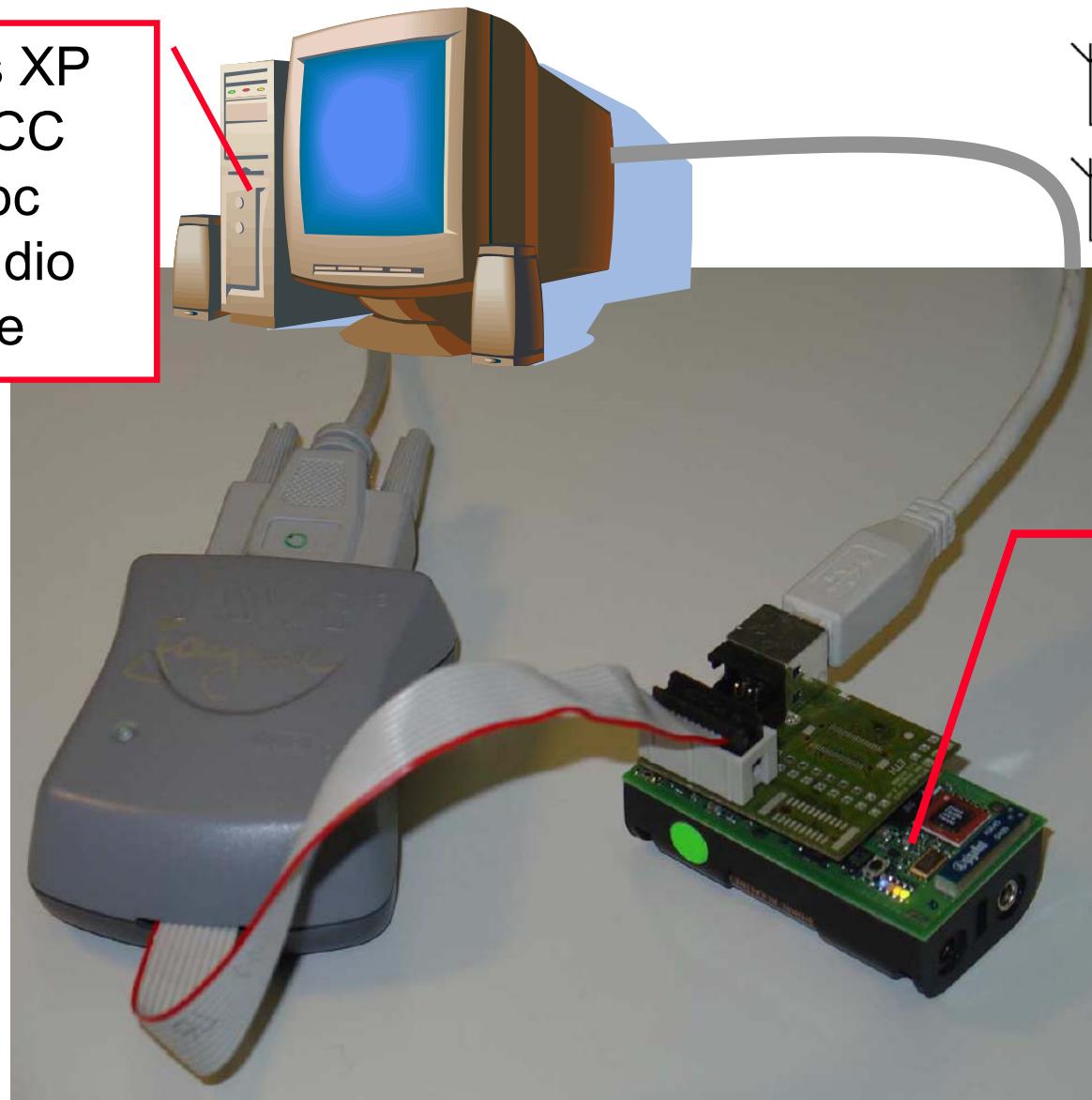
BTnode Platform



- ▶ generic platform for wireless distributed embedded computing
- ▶ complete platform including OS
- ▶ especially suited for pervasive computing applications (IoT)

Development in ES Exercise

Windows XP
GNU GCC
AVR libc
AVR Studio
Eclipse



Timing Guarantees

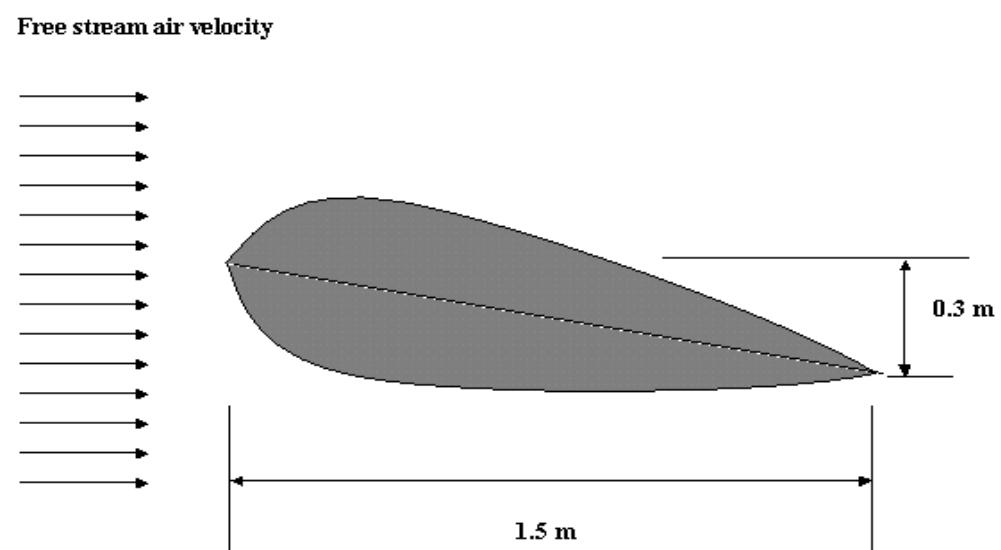


- ▶ **Hard real-time systems**, often in safety-critical applications abound
 - Aeronautics, automotive, train industries, manufacturing control

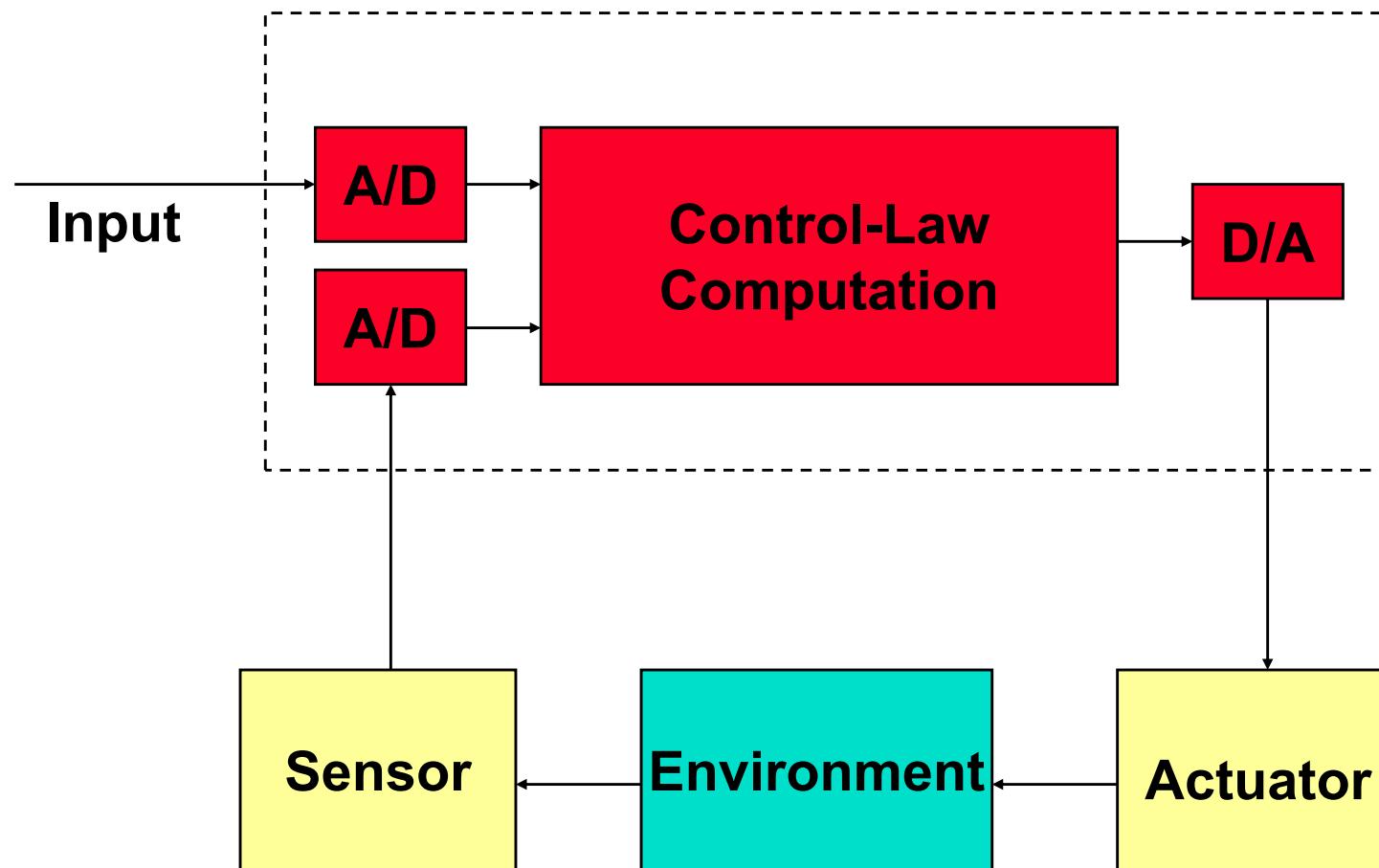
Sideairbag in car,
Reaction in <10 mSec



Wing vibration of airplane,
sensing every 5 mSec

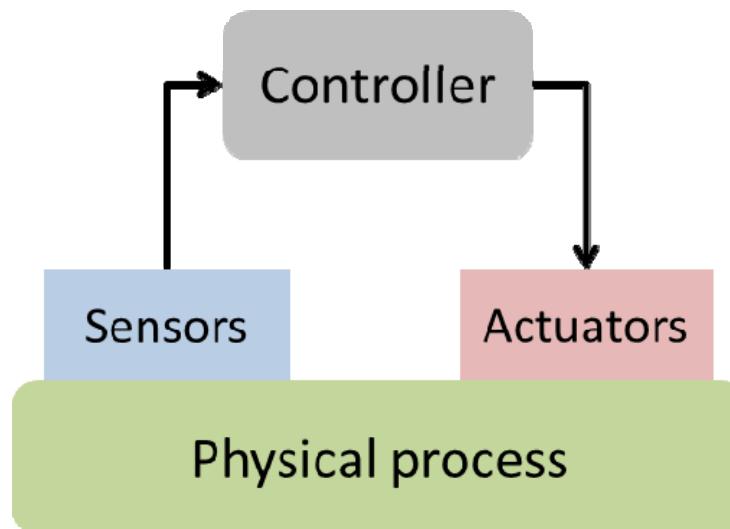


Simple Real-Time Control System

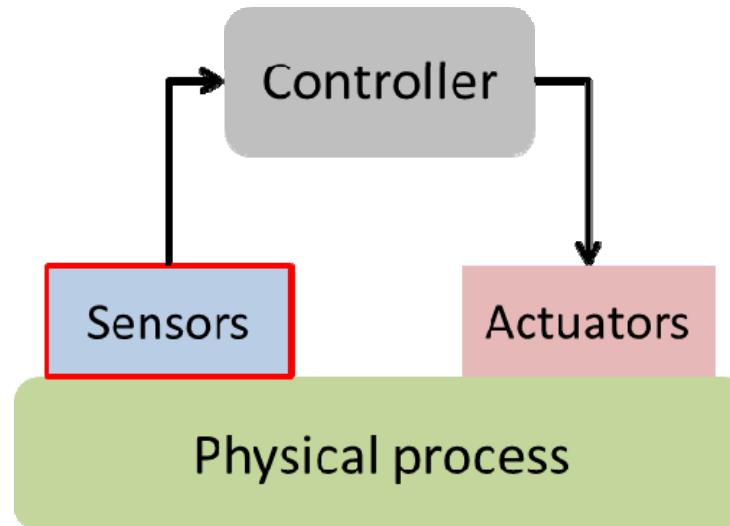


Real-Time Systems

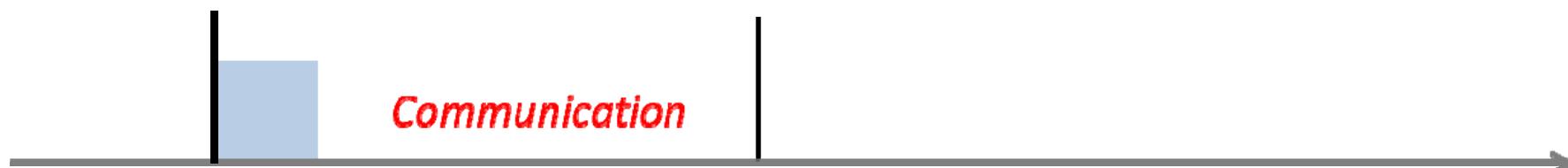
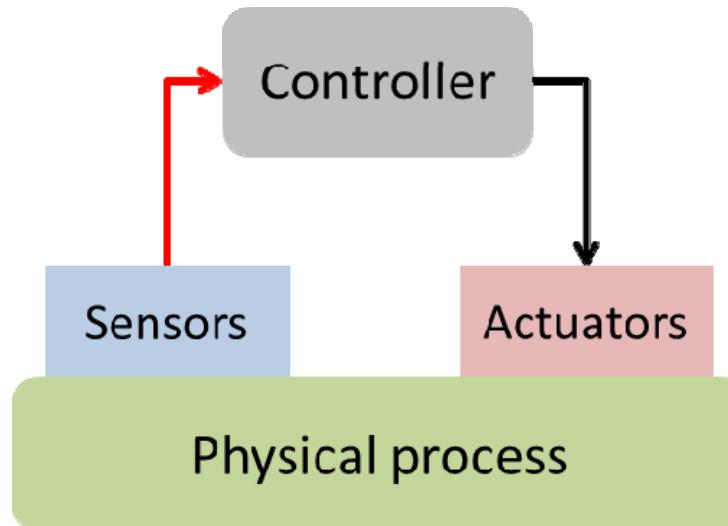
In many cyber-physical systems (CPSs) correct timing is a matter of **correctness**, not performance.



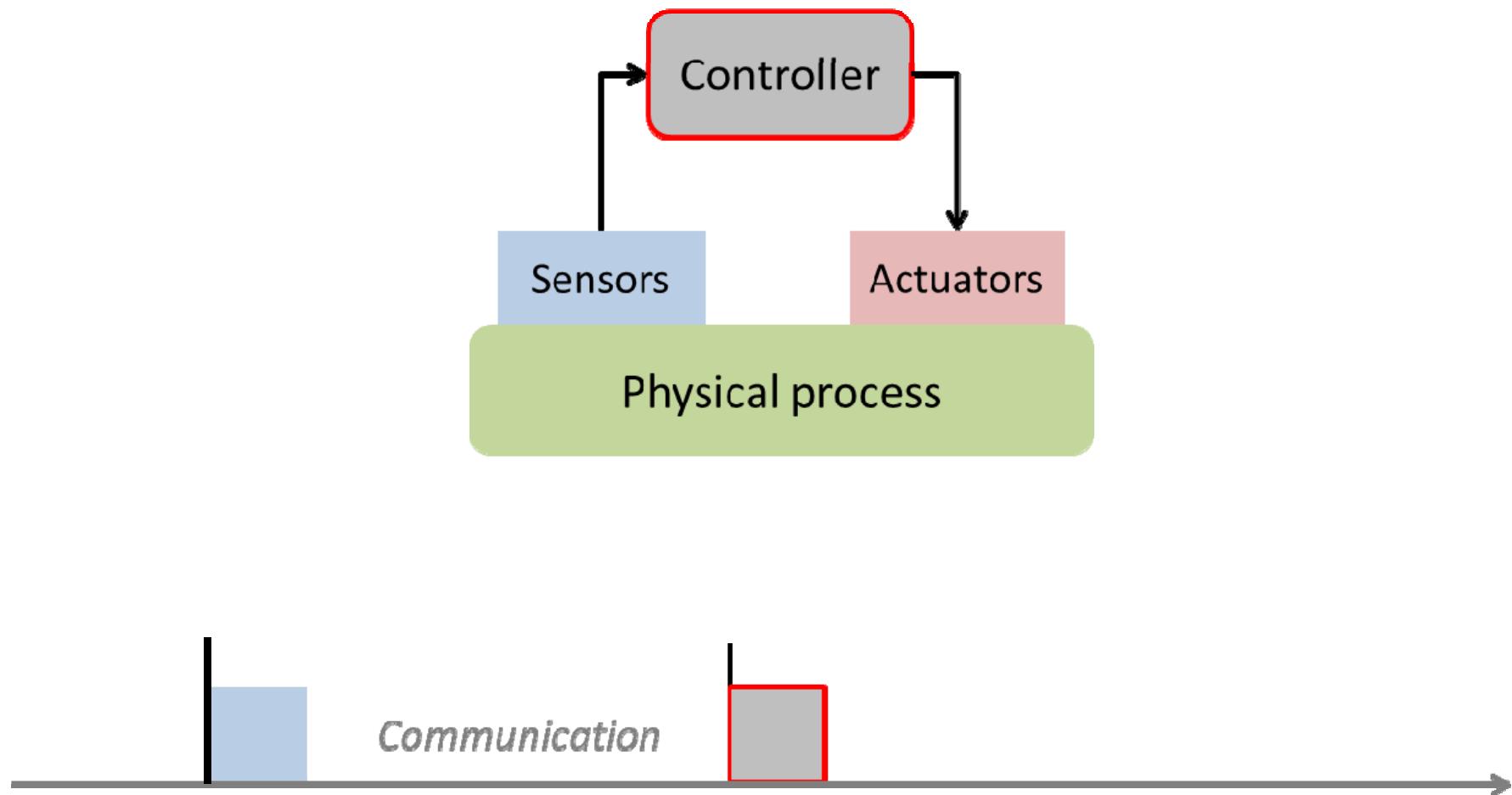
Real-Time Systems



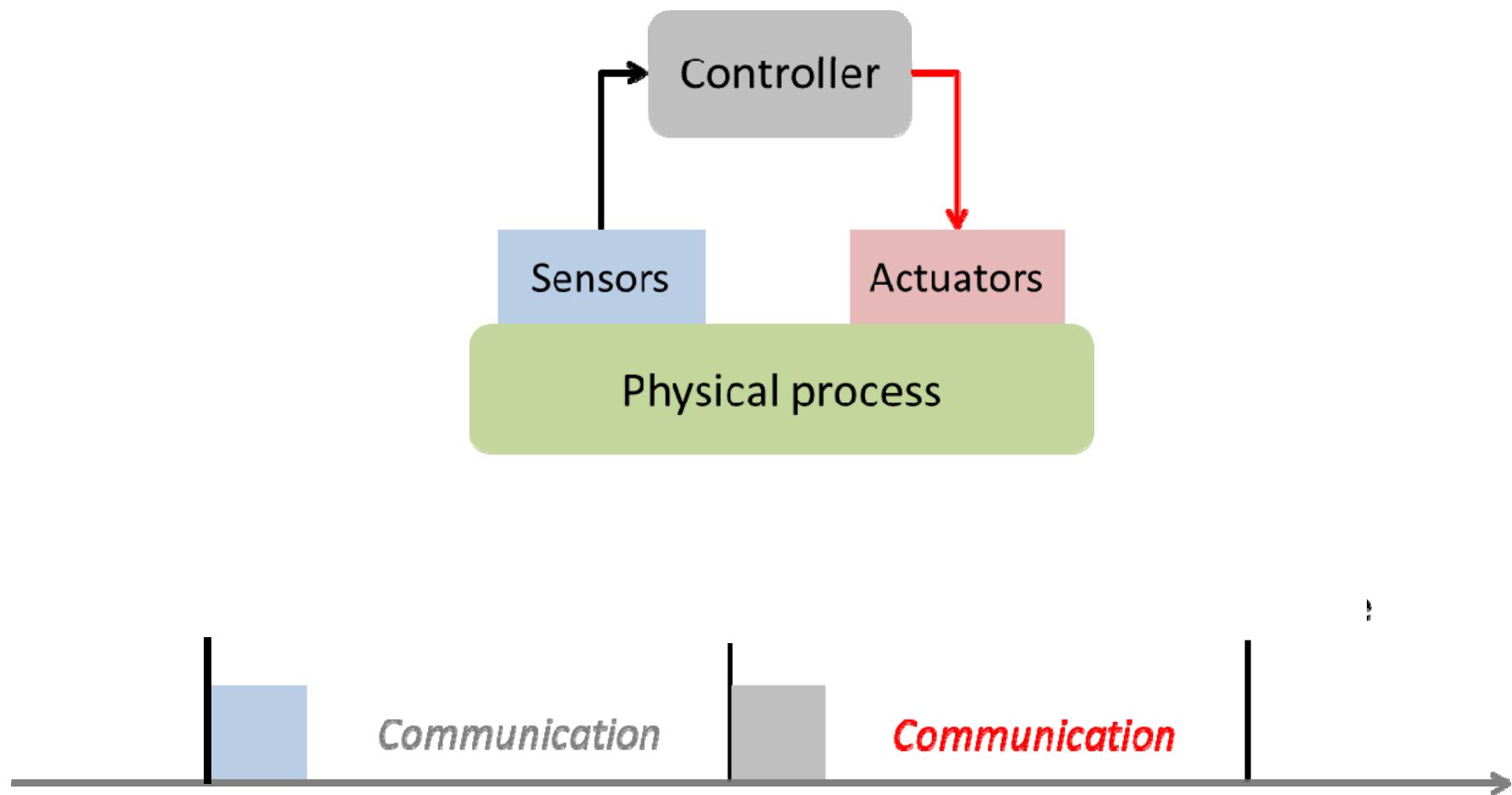
Real-Time Systems



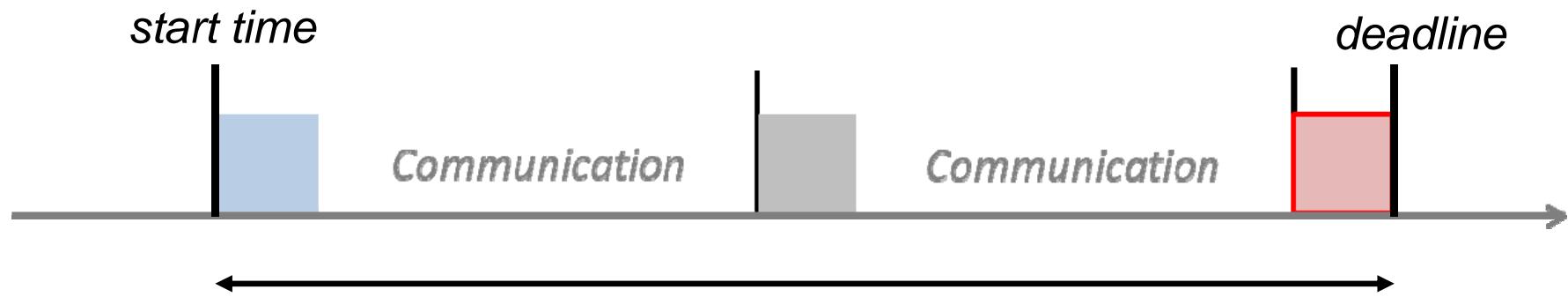
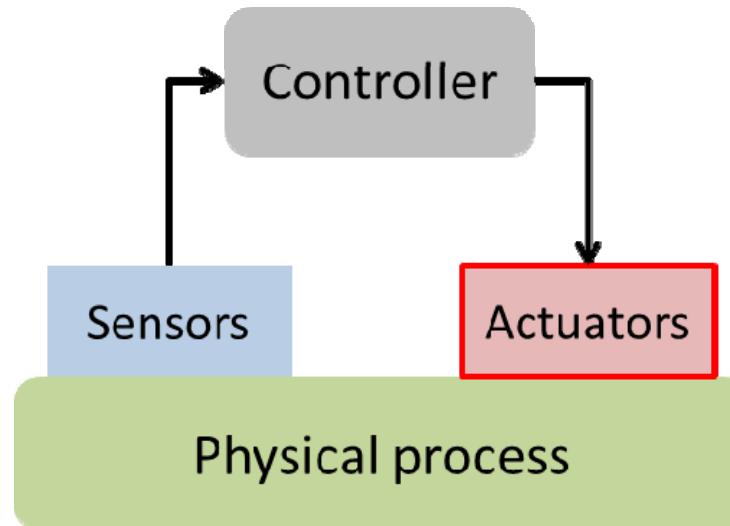
Real-Time Systems



Real-Time Systems



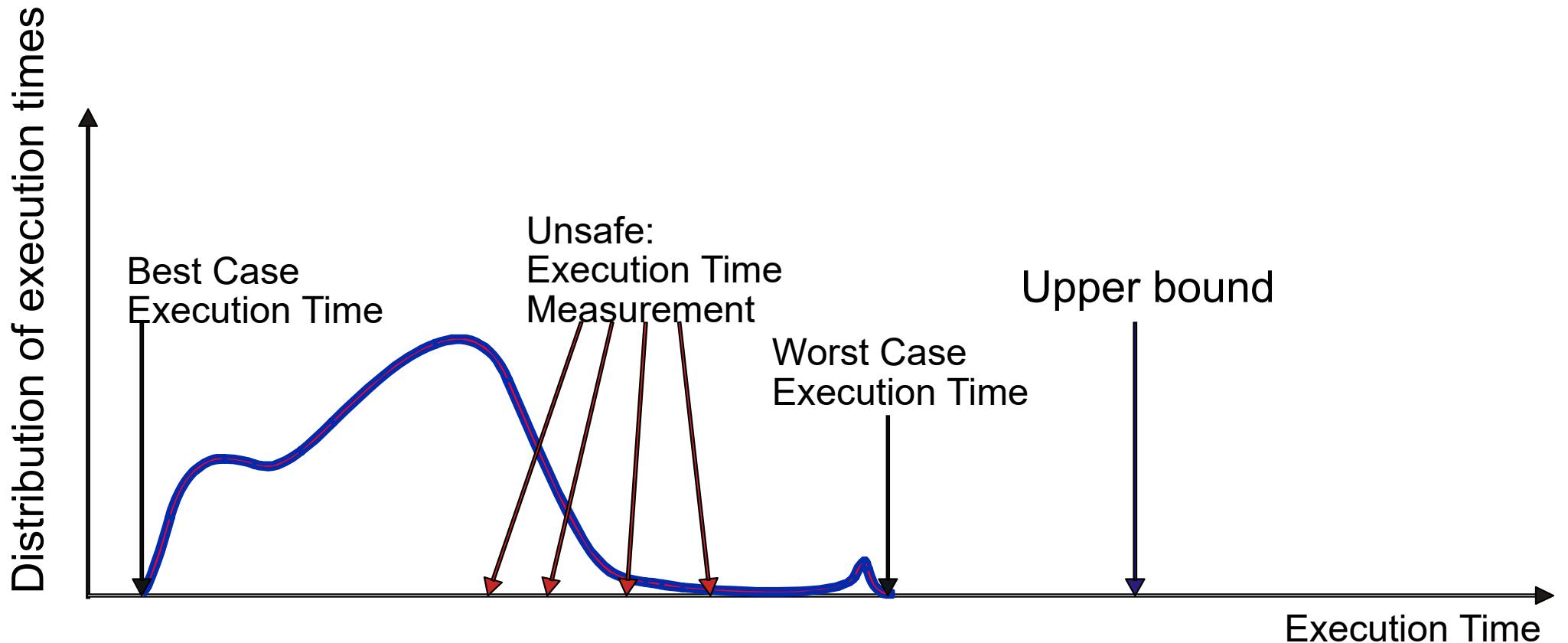
Real-Time Systems



Real-Time Systems

- ▶ Embedded controllers are often expected to finish their tasks reliably within time bounds.
- ▶ Essential: *upper bound on the execution times* of all tasks statically known.
- ▶ Commonly called the *Worst-Case Execution Time* (WCET)
- ▶ Analogously, *Best-Case Execution Time* (BCET)

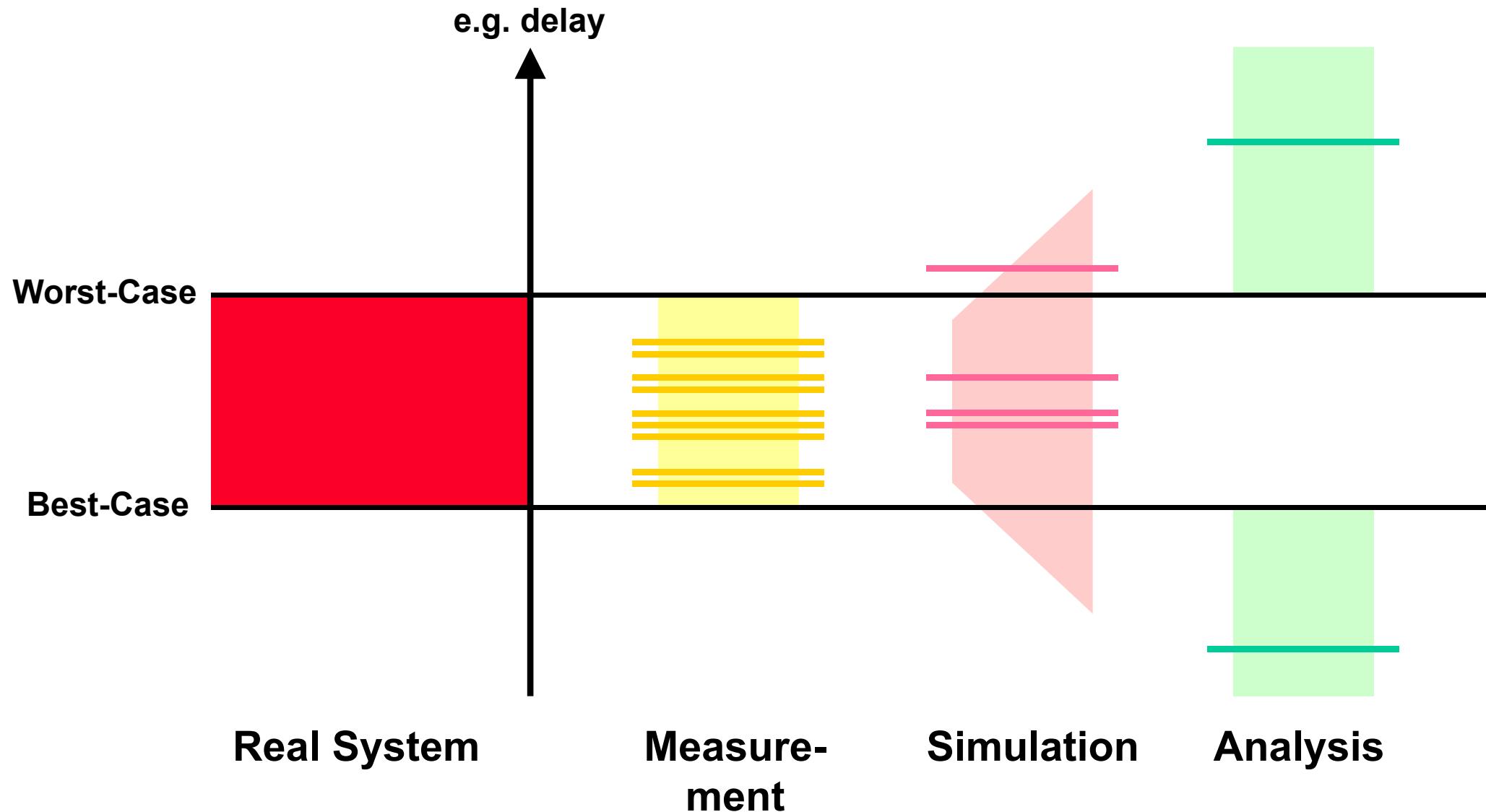
Distribution of Execution Times



Modern Hardware Features

- ▶ Modern processors ***Increase performance*** by using:
Caches, Pipelines, Branch Prediction, Speculation
- ▶ These features make ***WCET computation difficult***.
Execution times of instructions vary widely.
 - ***Best case*** - everything goes smoothly: no cache miss, operands ready, needed resources free, branch correctly predicted.
 - ***Worst case*** - everything goes wrong: all loads miss the cache, resources needed are occupied, operands are not ready.
 - ***Span may be several hundred cycles.***

System-Level Performance Methods



(Most of) Industry's Best Practice

- ▶ **Measurements**: determine execution times directly by observing the execution or a simulation on a set of inputs.
 - Does **not guarantee** an upper bound to all executions.
- ▶ **Exhaustive execution** in general **not possible!**
 - Too large space of (input domain) \times (set of initial execution states).
- ▶ **Compute upper bounds** along the **structure** of the program:
 - Programs are hierarchically structured.
 - Instructions are “nested” inside statements.
 - So, compute the upper bound for a statement from the upper bounds of its constituents

Determine the WCET

Complexity:

- in the general case: undecidable whether a bound exists.
- for restricted programs: simple for „old“ architectures, very complex for new architectures with pipelines, caches, interrupts, virtual memory, etc.

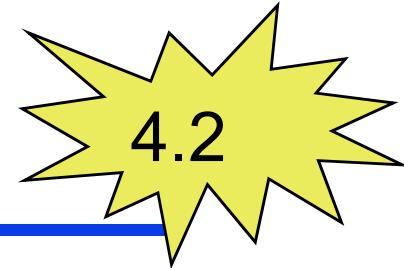
Analytic (formal) approaches:

- for hardware: typically requires hardware synthesis
- for software: requires availability of machine programs; complex analysis (see, e.g., www.absint.de); requires precise machine (hardware) model [see lecture “hardware software codesign”].

Subtopics

- ▶ A few introductory remarks.
- ▶ Different programming paradigms.

Why Multiple Processes?

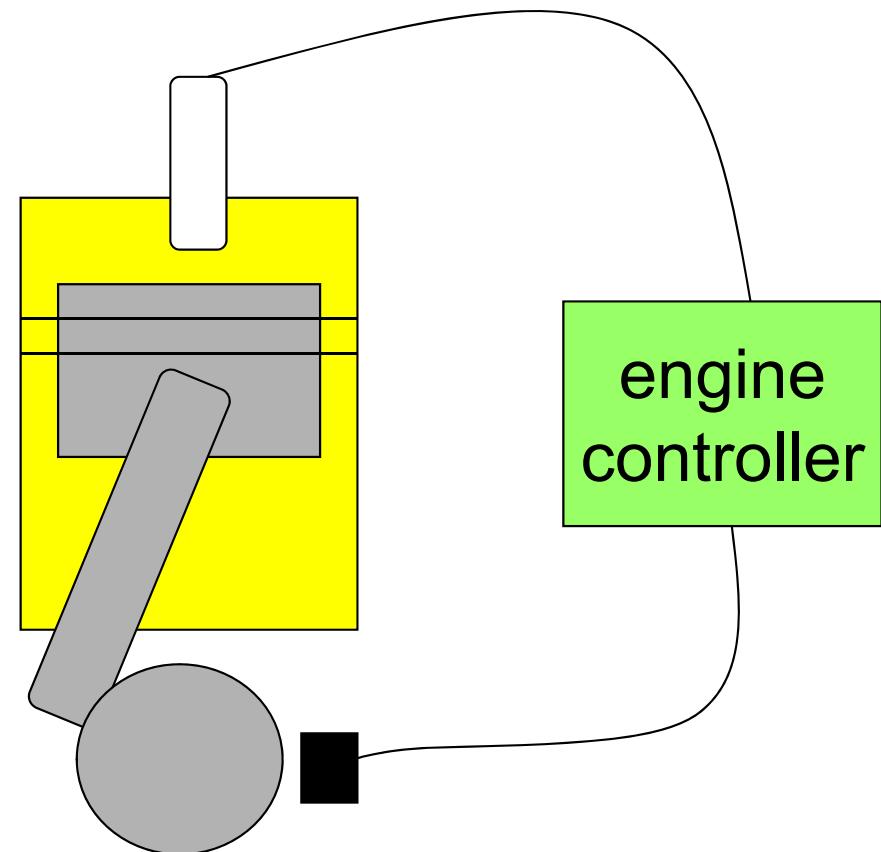


- ▶ The concept of ***concurrent processes*** reflects the intuition about the functionality of embedded systems.
- ▶ Processes help us ***manage timing complexity***:
 - multiple rates
 - multimedia
 - automotive
 - asynchronous input
 - user interfaces
 - communication systems

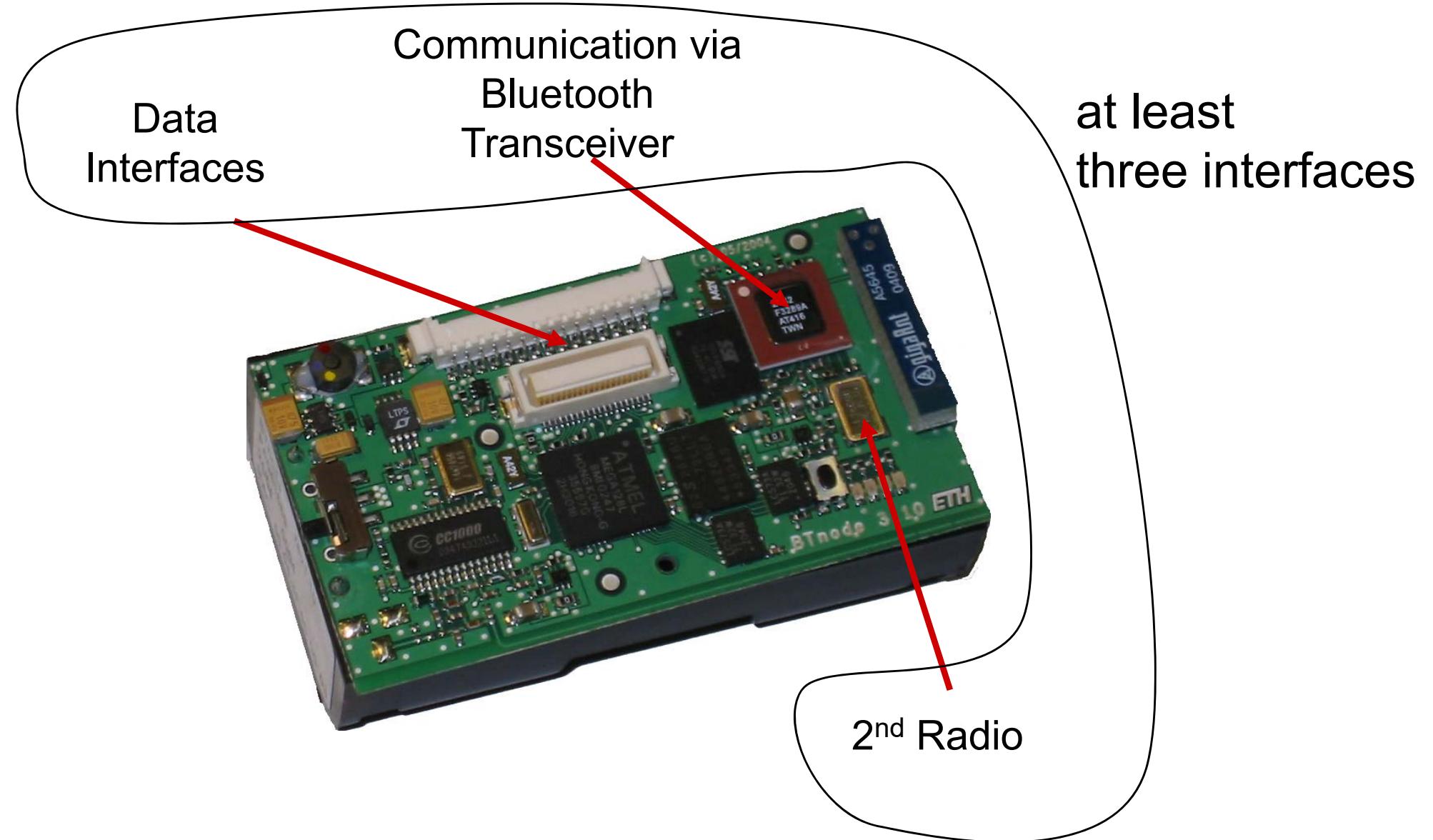
Example: Engine Control

► *Processes:*

- spark control
- crankshaft sensing
- fuel/air mixture
- oxygen sensor
- Kalman filter – control algorithm



BTnode Platform



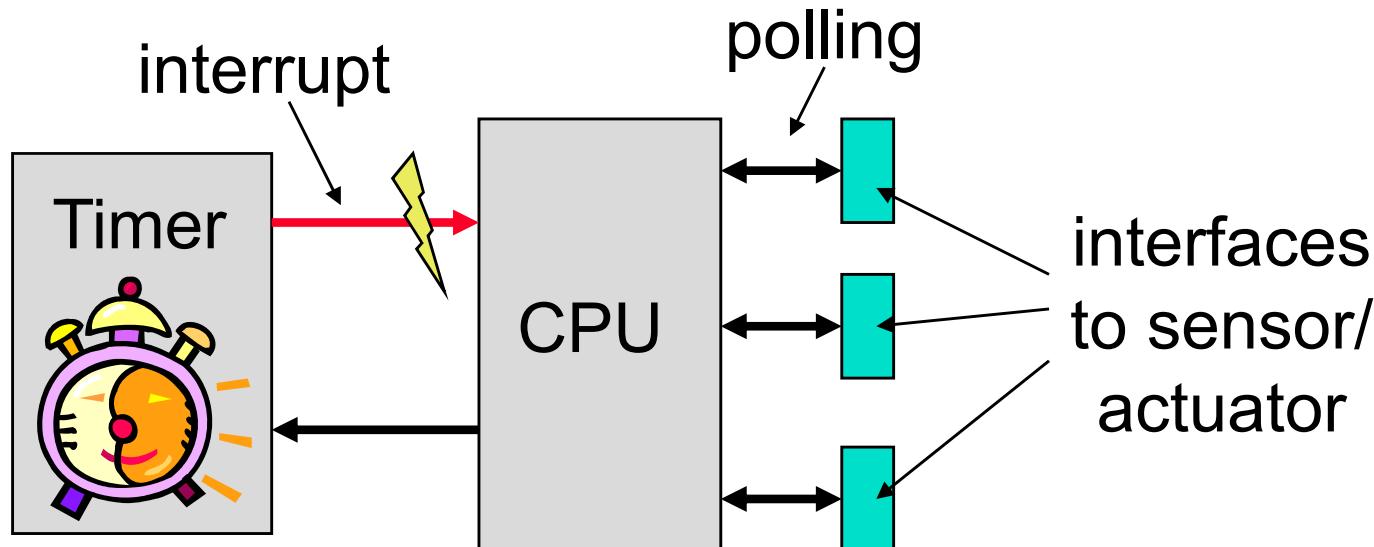
Overview

- ▶ There are **MANY** structured ways of programming an embedded system.
- ▶ Only **main principles** will be covered:
 - ***time triggered approaches***
 - periodic
 - cyclic executive
 - generic time-triggered scheduler
 - ***event triggered approaches***
 - non-preemptive
 - preemptive – stack policy
 - preemptive – cooperative scheduling
 - preemptive - multitasking

Time-Triggered Systems

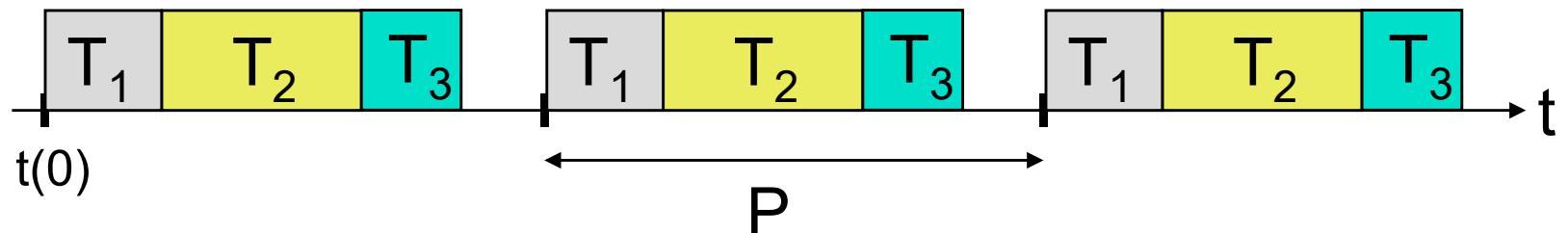
► *Pure model:*

- no interrupts except by timer
- schedule computed off-line → complex sophisticated algorithms can be used
- deterministic behavior at run-time
- interaction with environment through polling



Simple Periodic TT Scheduler

- ▶ Timer interrupts regularly with period P.
- ▶ All processes have same period P.



- ▶ **Properties:**
 - later processes (T_2 , T_3) have unpredictable starting times
 - no problem with communication between processes or use of common resources, as there is a static ordering
 - $\sum_{(k)} WCET(T_k) < P$

Simple Periodic TT Scheduler

main:

```
determine table of processes (k, T(k)), for k=0,1,...,m-1;  
i=0; set the timer to expire at initial phase t(0);  
while (true) sleep();
```

Timer Interrupt:

```
i=i+1;  
set the timer to expire at i*P + t(0);  
for (k=0,...,m-1){ execute process T(k); }  
return;
```

usually done offline

set CPU to low power mode;
returns after interrupt

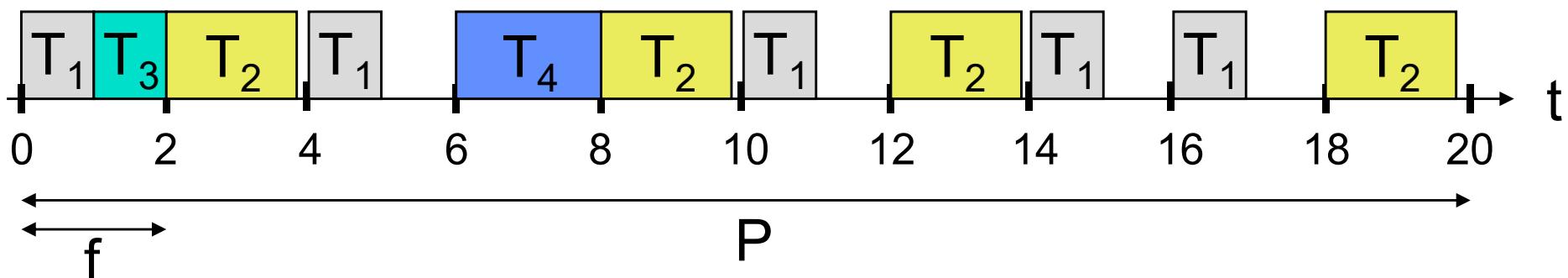
for example using a
function pointer in C;
task returns after finishing.

k	T(k)
0	T ₁
1	T ₂
2	T ₃
3	T ₄
4	T ₅

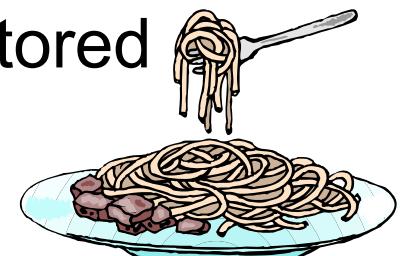
m=5

TT Cyclic Executive Scheduler

- ▶ Processes may have different periods.
- ▶ The period P is partitioned into frames of length f .



- ▶ Problem, if there are long processes; they need to be partitioned into a sequence of small processes; this is TERRIBLE, as local state must be extracted and stored globally:



TT Cyclic Executive Scheduler

- ▶ Some conditions:

- A process executes at most once ~~within a frame~~:
period of process k

$$f \leq p(k) \quad \forall k$$

- Period P is least common multiple of all periods $p(k)$.
 - Processes start and complete within a single frame:

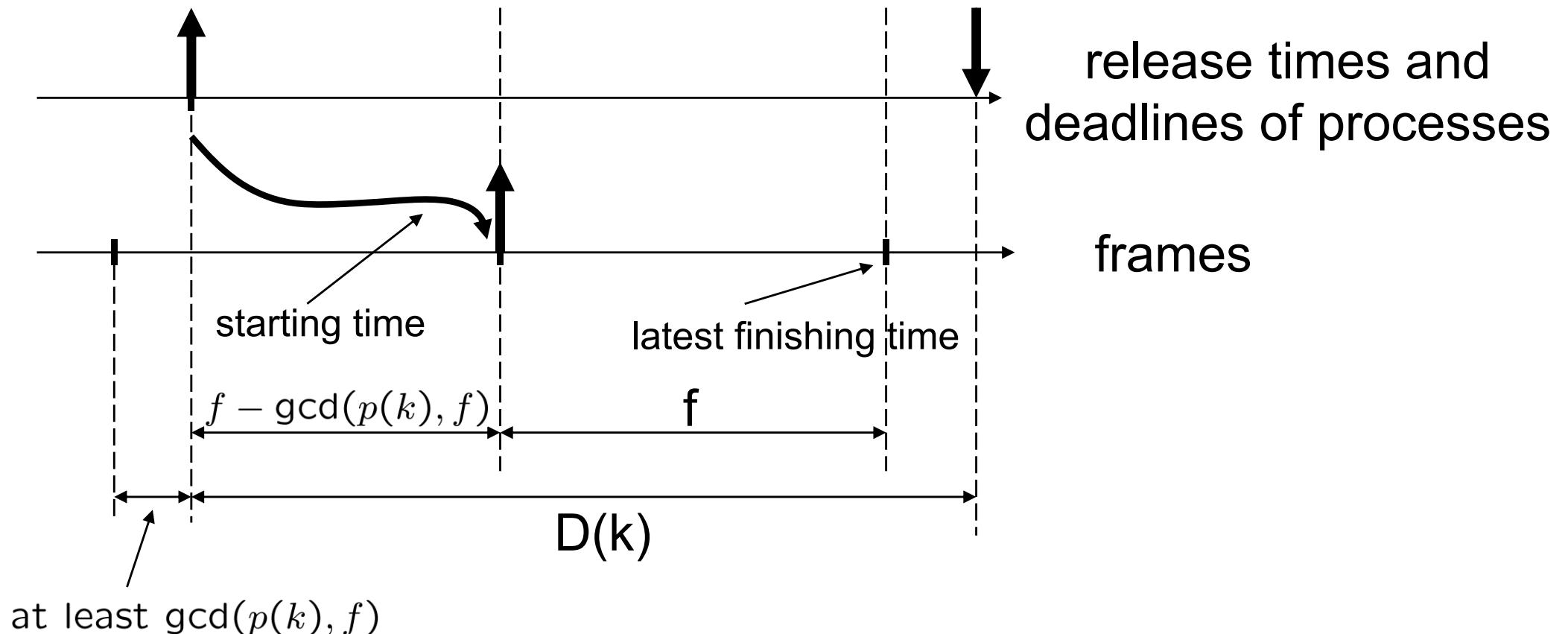
$$f \geq WCET(k) \quad \forall k$$

- Between release time and deadline of every task there is at least one frame boundary:

$$2f - \gcd(p(k), f) \leq D(k) \quad \forall k$$

relative deadline of process k

Sketch of Proof for Last Condition



Example: Cyclic Executive Scheduler

- ▶ Conditions:

$$f \leq \min\{4, 5, 20\} = 4$$

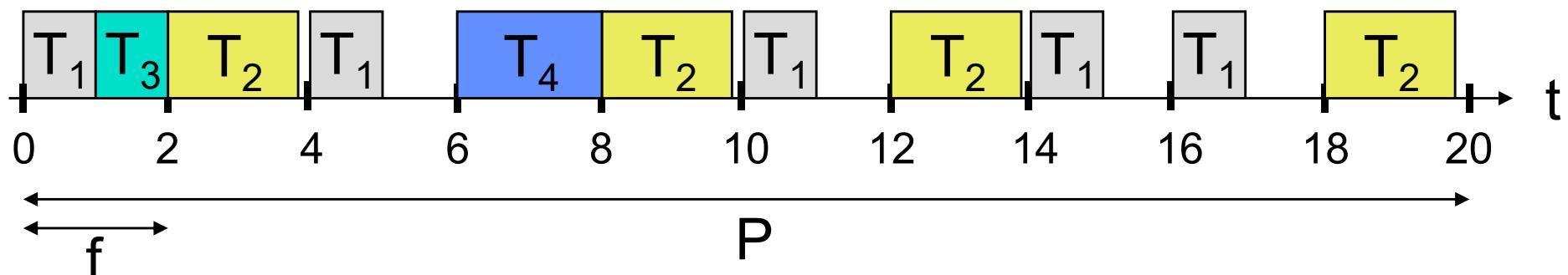
$$f \geq \max\{1.0, 1.0, 1.8, 2.0\} = 2.0$$

$$2f - \gcd(p(k), f) \leq D(k) \quad \forall k$$

possible solution: $f = 2$

- ▶ Feasible solution ($f=2$):

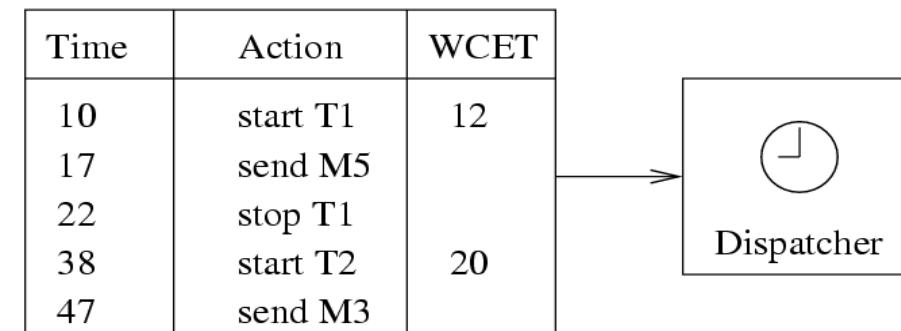
T(k)	D(k)	p(k)	WCET(k)
T ₁	4	4	1.0
T ₂	5	5	1.8
T ₃	20	20	1.0
T ₄	20	20	2.0



Generic Time-Triggered Scheduler

*In an entirely time-triggered system, the temporal control structure of all tasks is established **a priori** by off-line support-tools. This temporal control structure is encoded in a **Task-Descriptor List (TDL)** that contains the cyclic schedule for all activities of the node. This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary. ..*

The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant [Kopetz].



Simplified Time-Triggered Scheduler

main:

```
determine static schedule  $(t(k), T(k))$ , for  $k=0, 1, \dots, n-1$ ;  
determine period of the schedule  $P$ ;  
set  $i=k=0$  initially; set the timer to expire at  $t(0)$ ;  
while (true) sleep();
```

Timer Interrupt:

```
k_old := k;  
i := i+1; k := i mod n;  
set the timer to expire at  $\lfloor i/n \rfloor * P + t(k)$ ;  
execute process  $T(k_{\text{old}})$ ;  
return;
```

usually done offline

set CPU to low power mode;
returns after interrupt

for example using a
function pointer in C;
process returns after finishing.

k	$t(k)$	$T(k)$
0	0	T_1
1	3	T_2
2	7	T_1
3	8	T_3
4	12	T_2

$n=5, P = 16$

possible extensions: execute aperiodic background tasks if system is idle; check for task overruns (WCET too long)

Summary Time-Triggered Scheduler

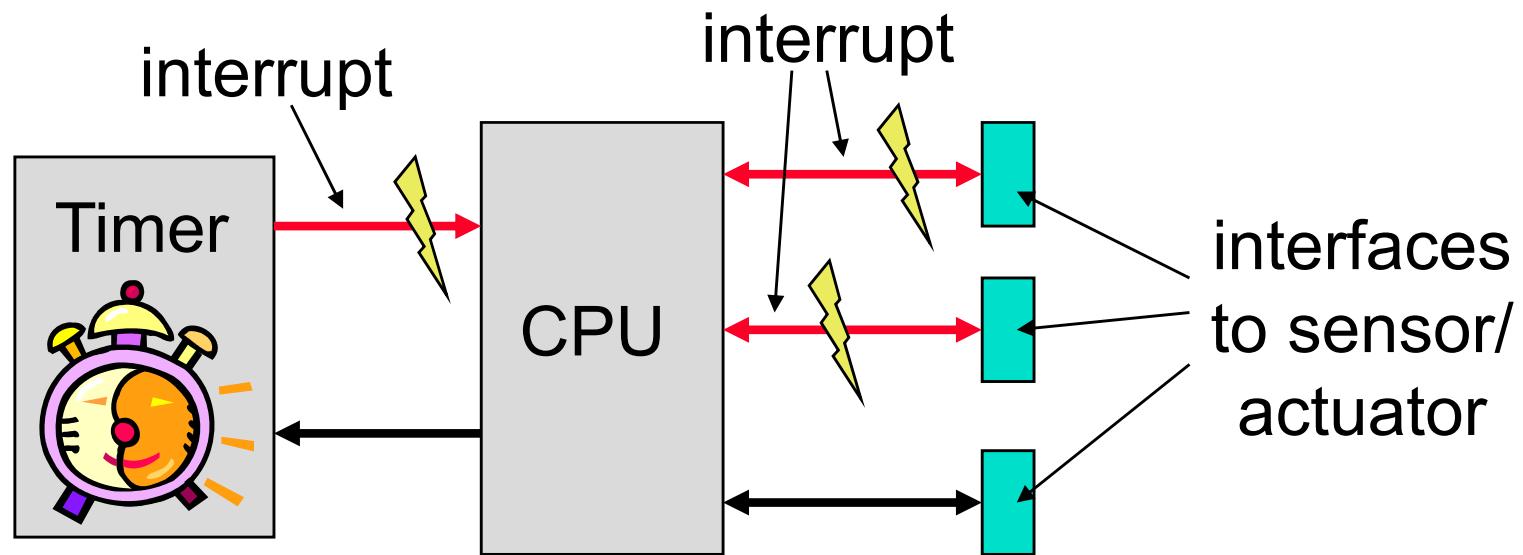
- ▶ **deterministic** schedule; conceptually simple (static table); relatively easy to validate, test and certify
- ▶ no problems in using **shared resources**

- ▶ external communication **only via polling**
- ▶ **inflexible** as no adaptation to environment
- ▶ serious **problems** if there are **long processes**

- ▶ **Extensions:**
 - allow interrupts (shared resources ? WCET ?) → **be careful!!**
 - allow preemptable background processes

Event Triggered Systems

- ▶ The schedule of processes is determined by the occurrence of external interrupts:
 - **dynamic and adaptive**: there are possible problems with respect to timing, the use of shared resources and buffer over- or underflow
 - **guarantees** can be given either off-line (if bounds on the behavior of the environment are known) or during run-time



Non-Preemptive ET Scheduling

► *Principle:*

- To each event, there is associated a corresponding process that will be executed.
- Events are emitted by (a) external interrupts and (b) by processes themselves.
- Events are collected in a queue; depending on the queuing discipline, an event is chosen for running.
- Processes can not be preempted.

► *Extensions:*

- A background process can run (and preempted!) if the event queue is empty.
- Timed events enter the queue only after a time interval elapsed. This enables periodic instantiations for example.

Non-Preemptive ET Scheduling

main:

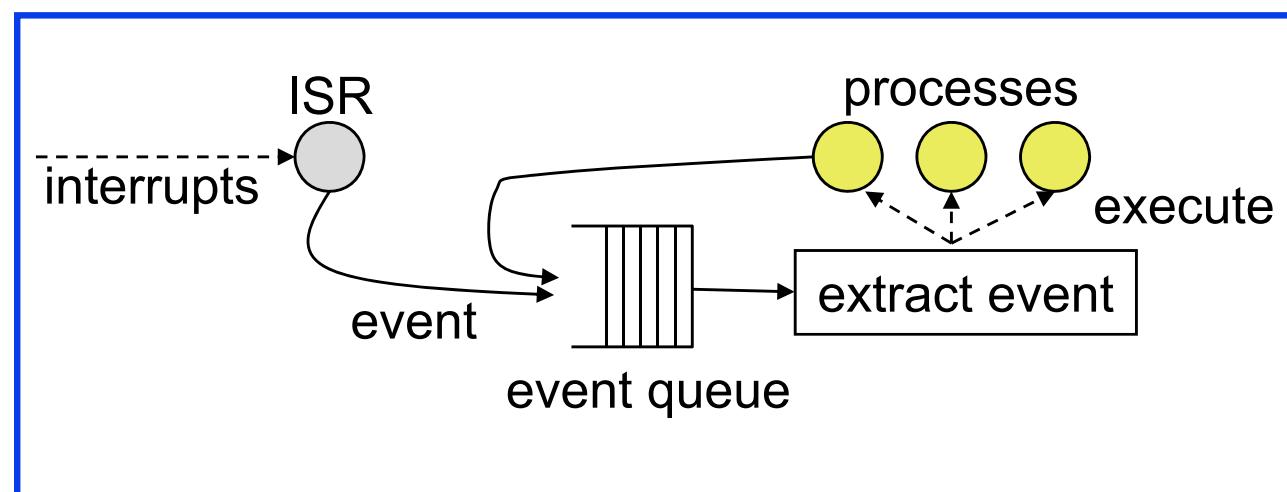
```
while (true) {  
    if (event queue is empty) {  
        sleep();  
    } else {  
        extract event from event queue;  
        execute process corresponding to event;  
    }  
}
```

set CPU to low power mode;
returns after interrupt

for example using a
function pointer in C;
process returns after
finishing.

Interrupt:

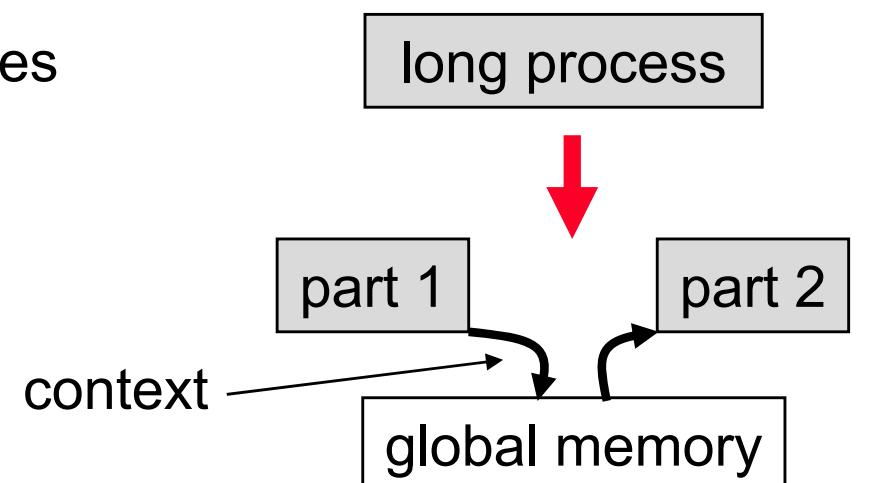
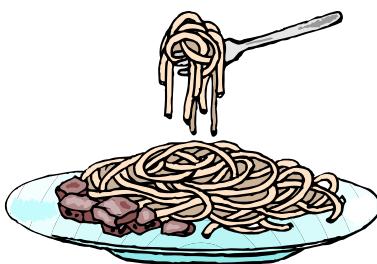
```
put event into event queue;  
return;
```



Non-Preemptive ET Scheduling

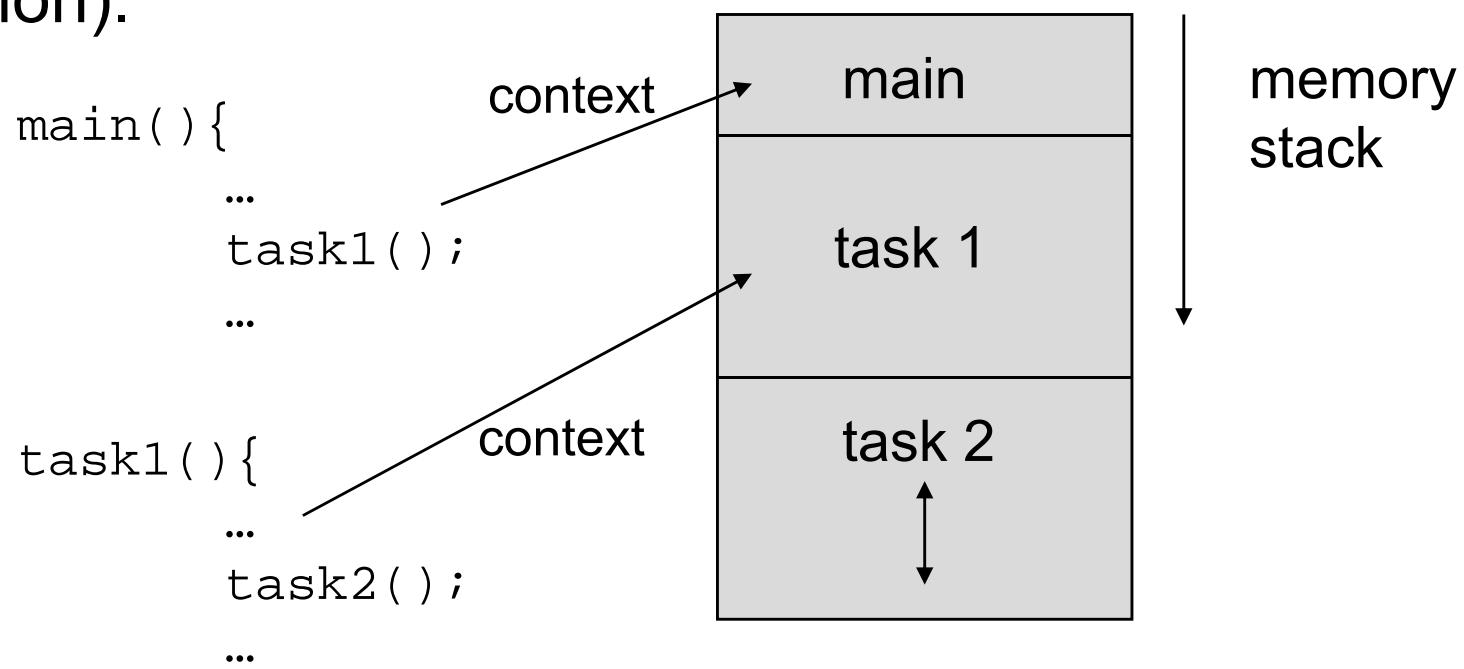
► *Properties:*

- communication between processes is simple (no problems with shared resources); interrupts may cause problems with shared resources
- buffer overflow if too many events are generated by environment or processes
- long processes prevent others from running and may cause buffer overflow
 - partition processes into smaller ones
 - local context must be stored

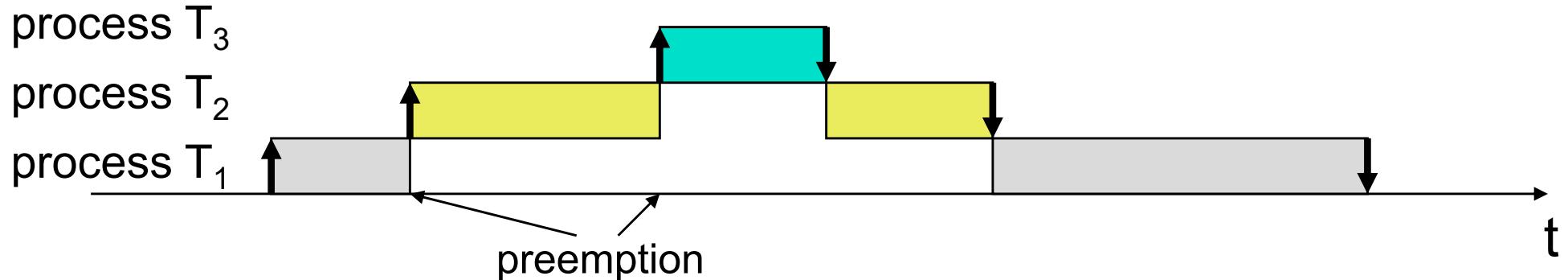


Preemptive ET Scheduling – Stack Policy

- ▶ Similar to non-preemptive case, but **processes can be preempted** by others; this resolves partly the problem of long tasks.
- ▶ If the order of preemption is restricted, we can use the usual **stack-based context mechanism of function calls** (process = function).



Preemptive ET Scheduling – Stack Policy



- ▶ Processes must finish in **LIFO order** of their instantiation.
 - restricts flexibility
 - not useful, if several processes wait unknown time for external events
- ▶ **Shared resources** (communication between processes!) must be **protected**, for example: disabling interrupts, use of semaphores.

Preemptive ET Scheduling – Stack Policy

main:

```
while (true) {
    if (event queue is empty) {
        sleep();
    } else {
        select event from event queue;
        execute selected process; →
        remove selected event from queue;
    }
}
```

set CPU to low power mode;
returns after interrupt

for example using a
function pointer in C;
process returns after finishing.

InsertEvent:

```
put new event into event queue;
select event from event queue;
if (sel. process ≠ running process) {
    execute selected process;
    remove selected event from queue;
}
return;
```

Interrupt:

```
InsertEvent(...);
return;
```

may be called by
interrupt service
routines (ISR)
or processes

Process

- ▶ **A process is a unique execution of a program.**
 - Several copies of a “program” may run simultaneously or at different times.
- ▶ A **process has its own state**. In case of a thread, this state consists mainly of:
 - register values;
 - memory stack;

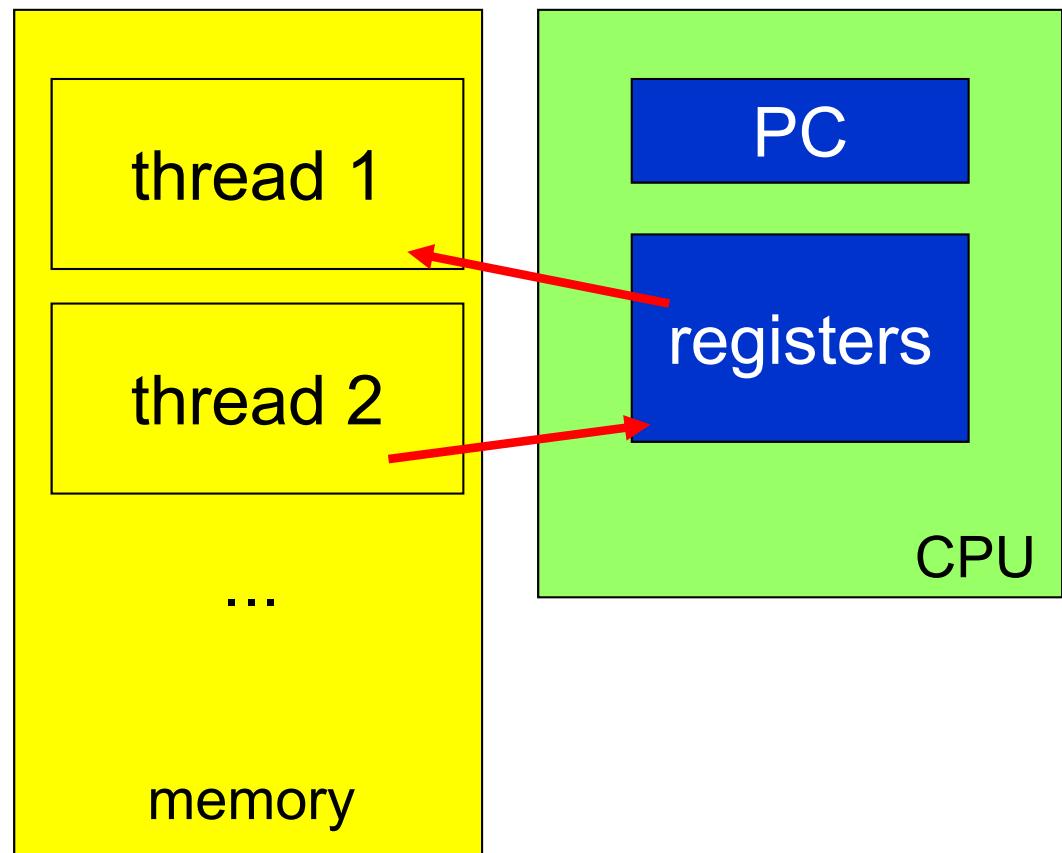
Processes and CPU

- ▶ ***Activation record:***

- copy of process state
- includes registers and local data structures

- ▶ ***Context switch:***

- current CPU context goes out
- new CPU context goes in



Co-operative Multitasking

- ▶ Each process allows a context switch at `cswitch()` call.
 - ▶ Separate scheduler chooses which process runs next.
-
- ▶ **Advantages:**
 - predictable, where context switches can occur
 - less errors with use of shared resources
 - ▶ **Problems:**
 - programming errors can keep other threads out, thread never gives up CPU
 - real-time behavior at risk if it takes too long before context switch allowed

Example: co-operative multitasking

Process 1

```
if (x > 2)
    sub1(y);
else
    sub2(y);
cswitch();
proca(a,b,c);
```

Process 2

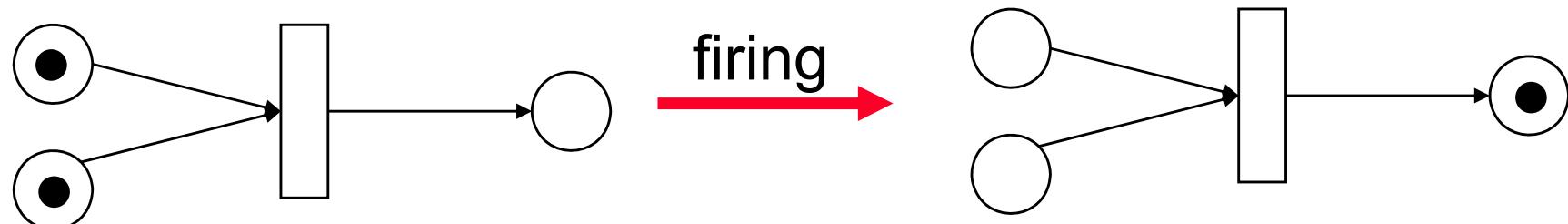
```
procdata(r,s,t);
cswitch();
if (val1 == 3)
    abc(val2);
rst(val3);
```

Scheduler

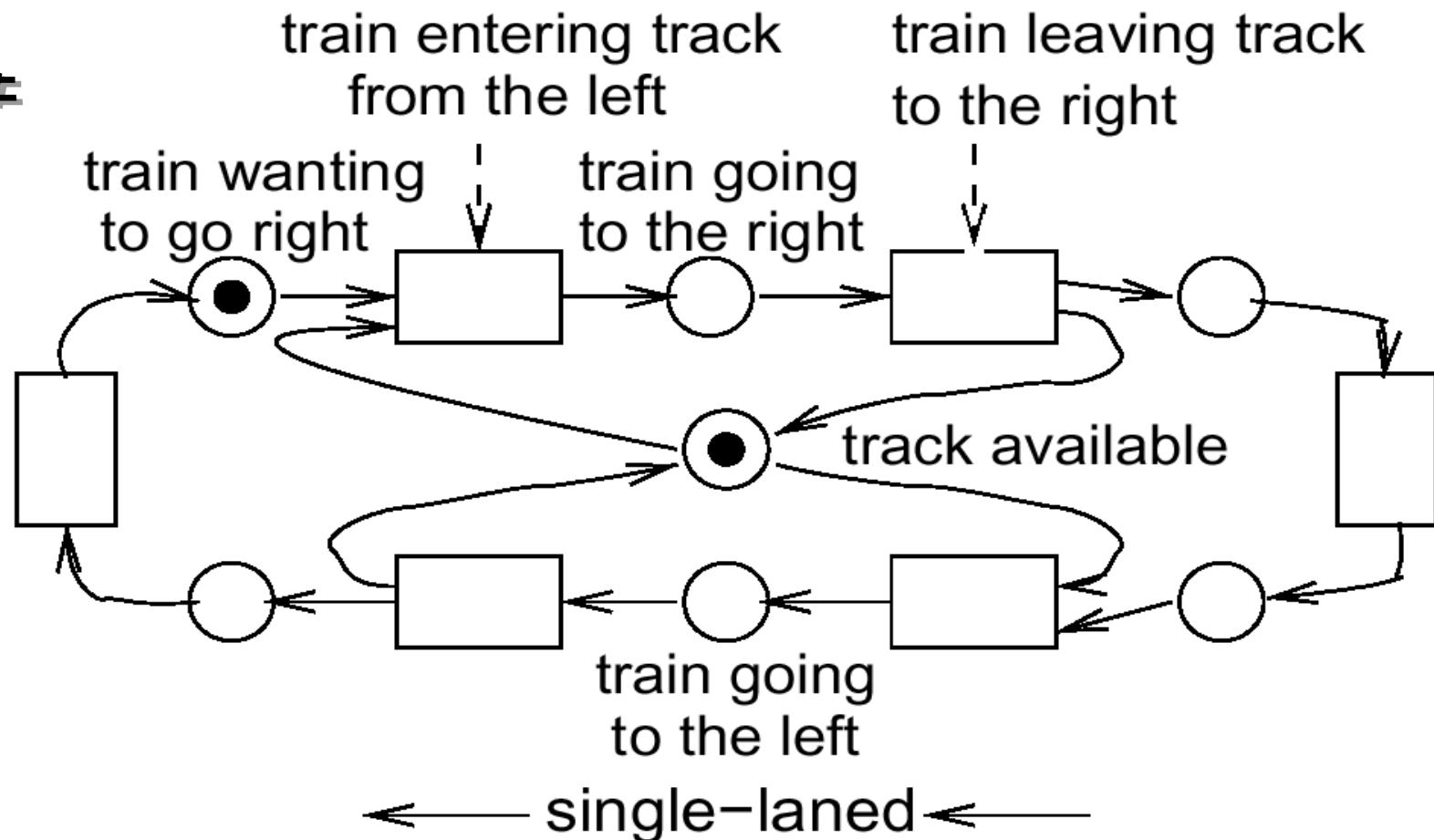
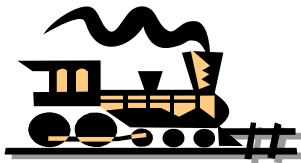
```
save_state(current);
p = choose_process();
load_and_go(p);
```

A Typical Programming Interface

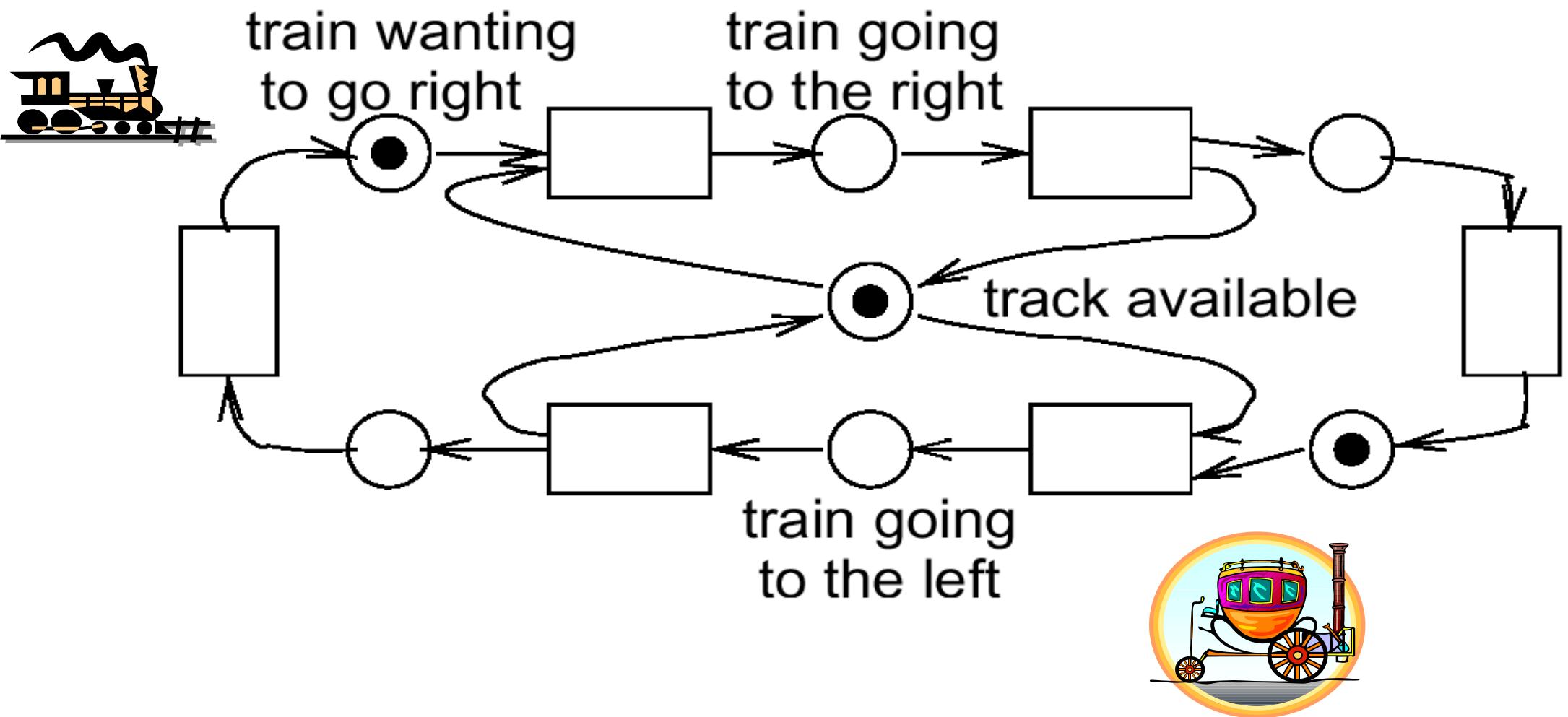
- ▶ Example of a co-operative multitasking OS for small devices: **NutOS** (as used in the practical exercises).
- ▶ Semantics of the calls is expressed using **Petri Nets**
 - Bipartite graph consisting of **places** and **transitions**.
 - Data and control are represented by moving **token**.
 - Token are moved by transitions according to **rules**: A transition can **fire** (is enabled) if there is at least one token in every input place. After firing, one token is removed from each input place and one is added to each output place.



Example: Single Track Rail Segment



Example: Conflict for Resource „Track“



API for Co-Operative Scheduling

▶ *Creating a Thread*

```
THREAD(my_thread, arg) {  
    for (;;) {  
        // do something  
    }  
}
```

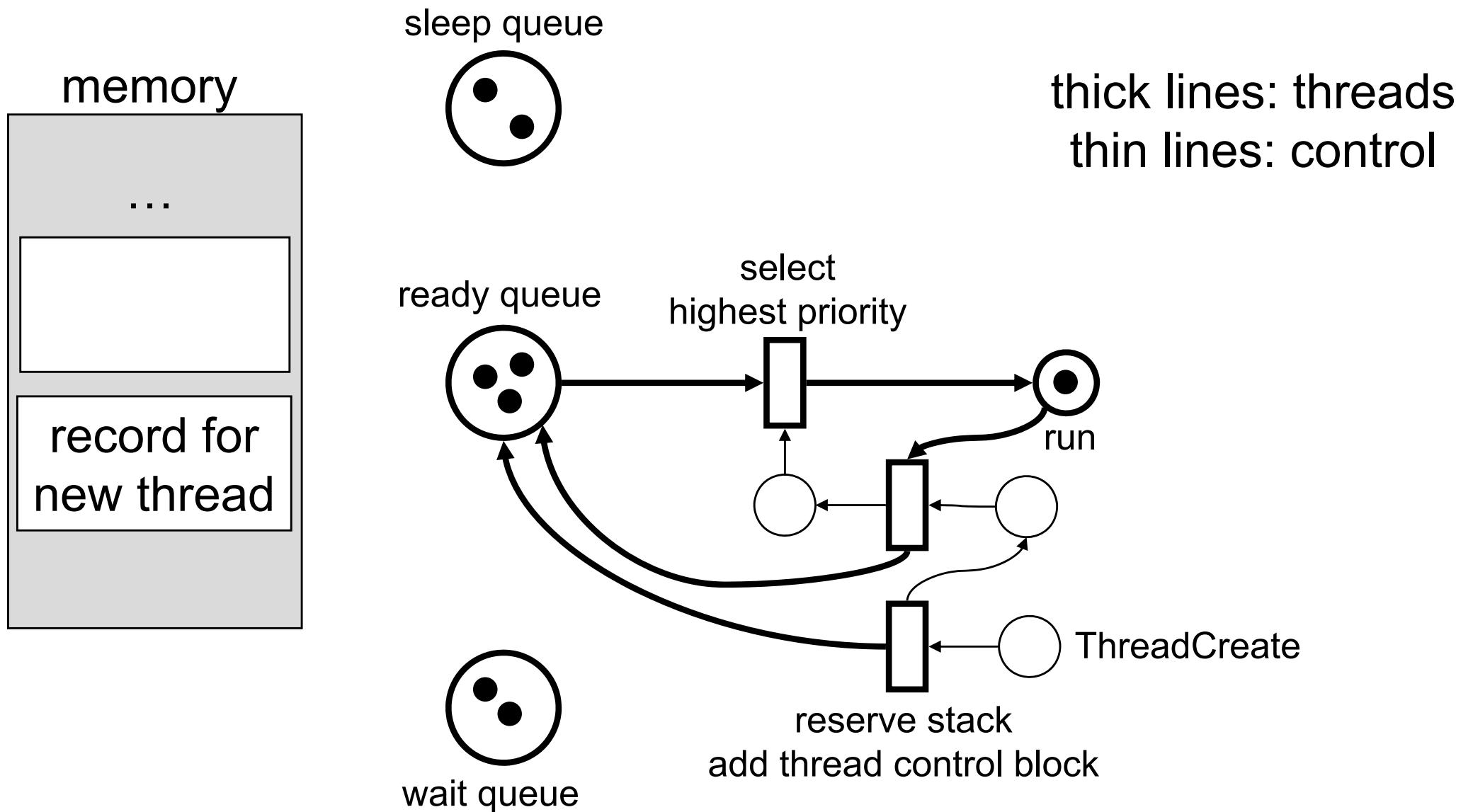
a thread looks like a function that never returns

the thread is put into life

```
int main(void) {  
    if (0 == NutThreadCreate("My Thread", my_thread, 0, 192)) {  
        // Creating the thread failed  
    }  
    for (;;) {  
        // do something  
    }  
}
```

stack size

API for Co-Operative Scheduling



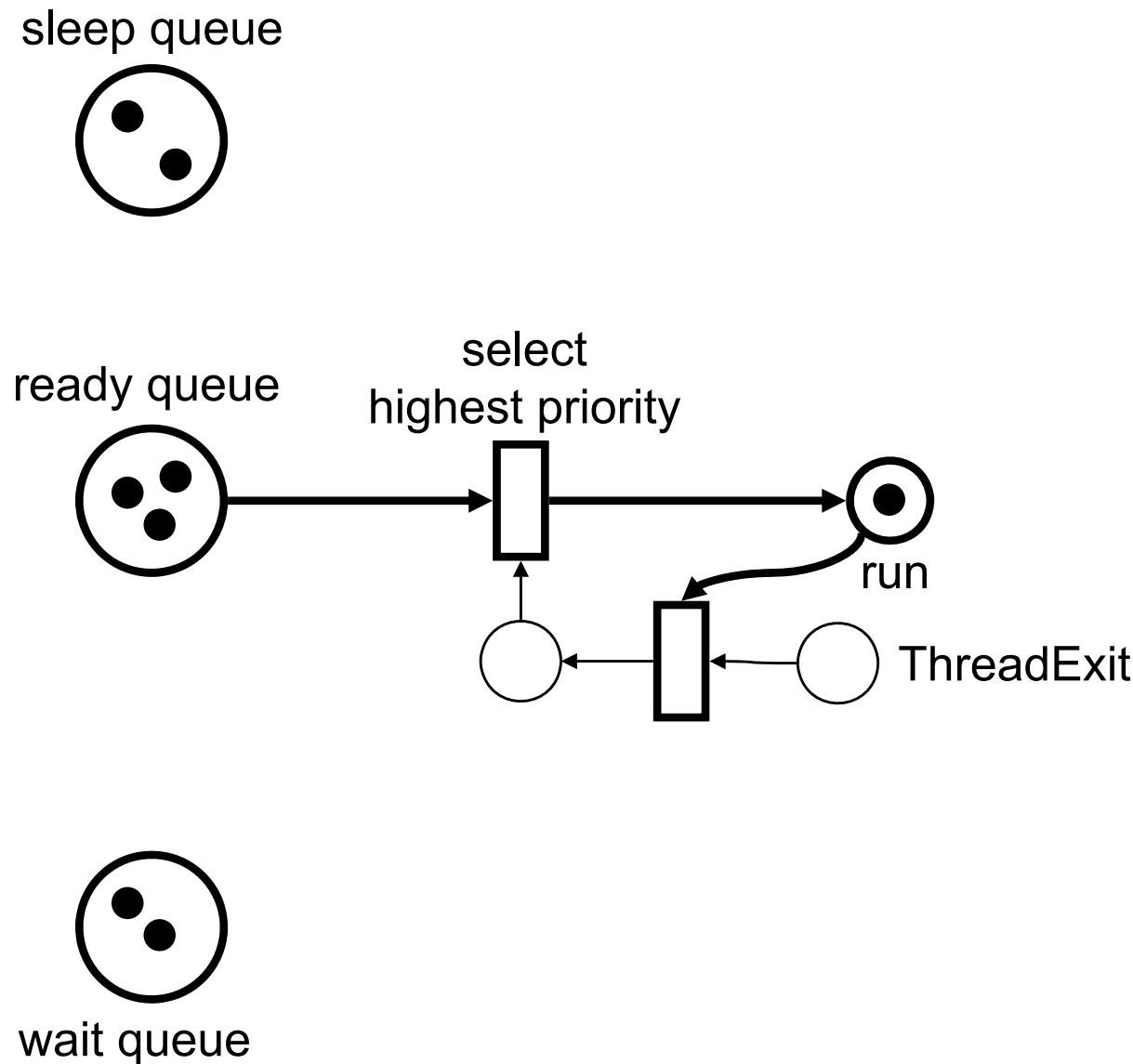
API for Co-Operative Scheduling

- ▶ **Terminating**

```
THREAD(my_thread, arg) {  
    for (;;) {  
        // do something  
        if (some condition)  
            NutThreadExit()  
    }  
}
```

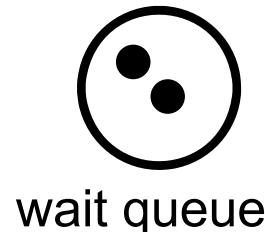
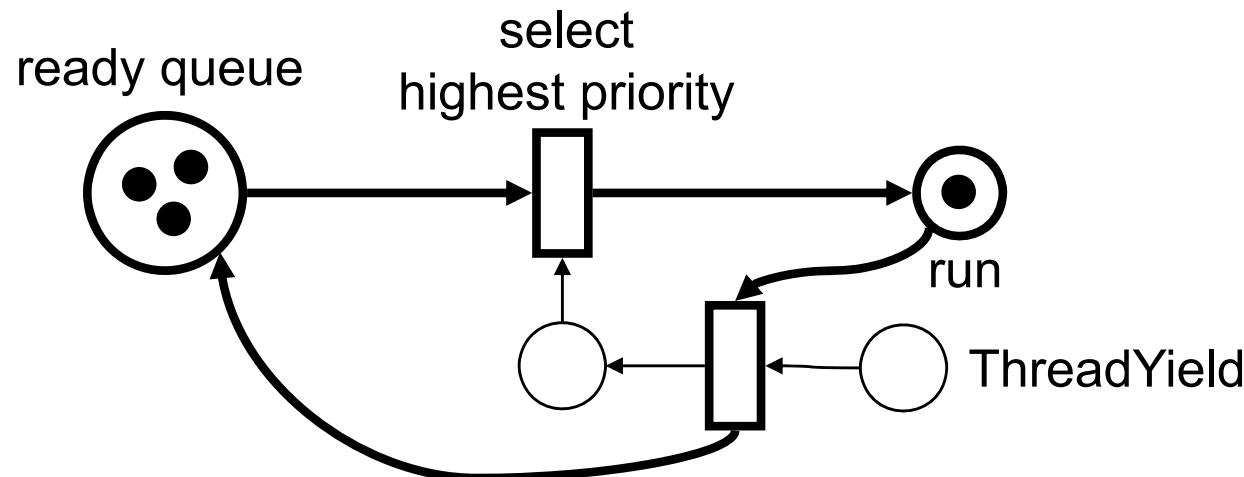
can only kill itself

API for Co-Operative Scheduling



API for Co-Operative Scheduling

- ▶ ***Yield access to another thread:***



wait queue

```
THREAD(my_thread, arg) {  
    for (;;) {  
        NutThreadSetPriority(20);  
        // do something  
    }  
}
```

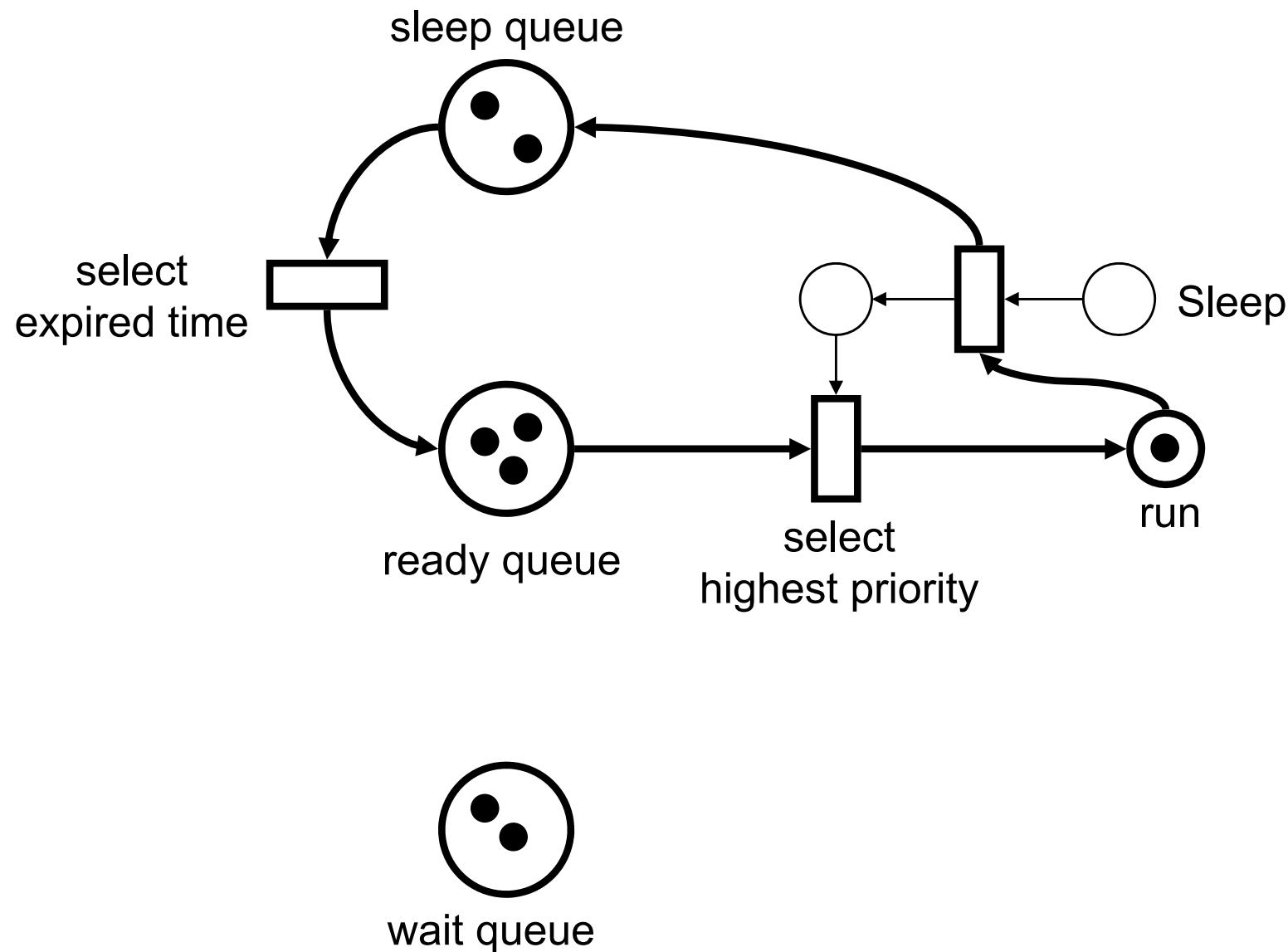
- ▶ ***Same structure for SetPriority:***

API for Co-Operative Scheduling

- ▶ *Sleep*

```
THREAD(my_thread, arg) {  
    for (;;) {  
        // do something  
        NutSleep(1000);  
    }  
}
```

API for Co-Operative Scheduling



API for Co-Operative Scheduling

► *Posting and waiting for events:*

```
#include <sys/event.h>

HANDLE my_event;

THREAD(thread_A, arg) {
    for (;;) {
        // some code
        NutEventWait(&my_event, NUT_WAIT_INFINITE);
        // some code
    }
}
```

wait for event, but only limited time

```
THREAD(thread_B, arg) {
    for (;;) {
        // some code
        NutEventPost(&my_event);
        // some code
    }
}
```

post event

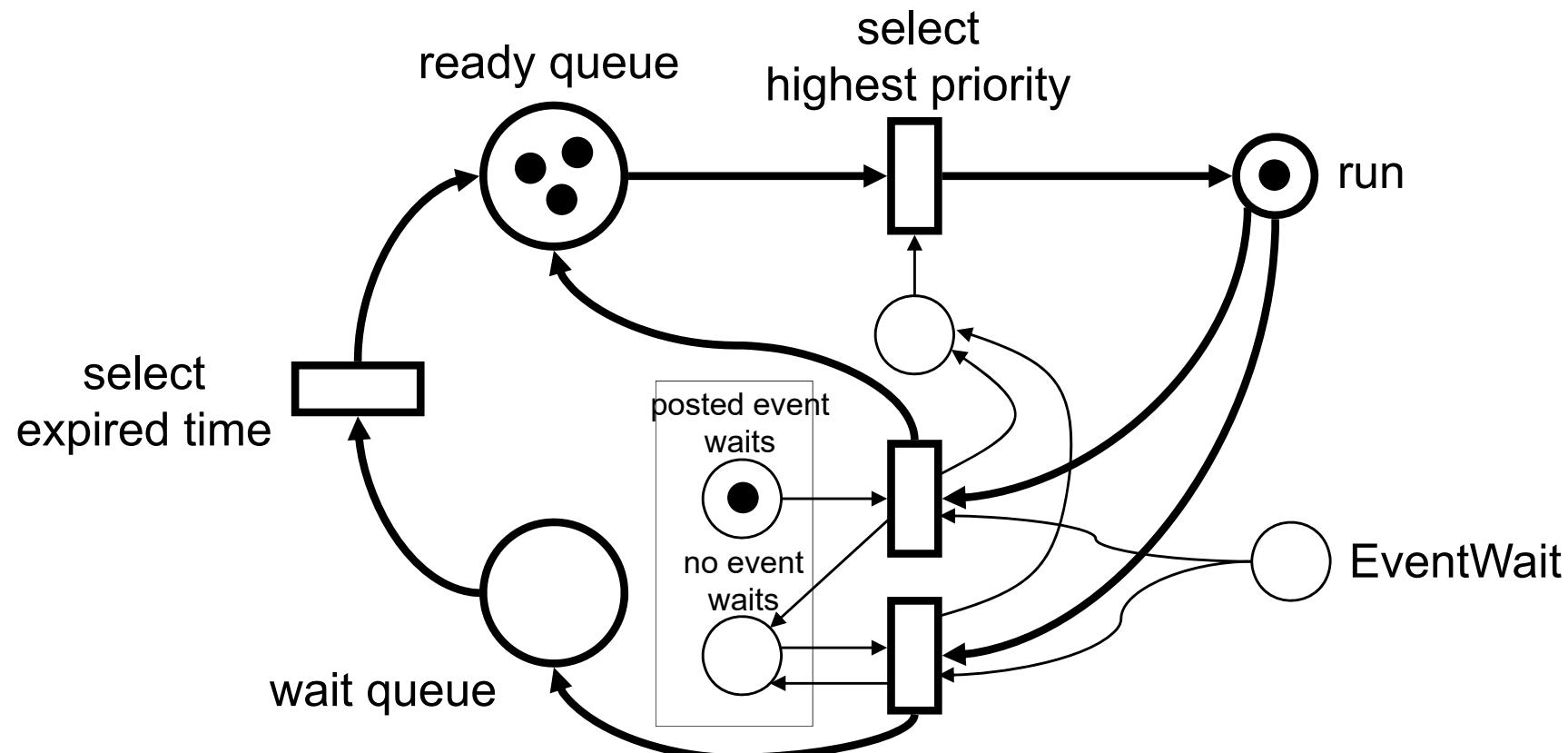
API for Co-Operative Scheduling

EventWait

sleep queue



there is one
event queue for
each event type



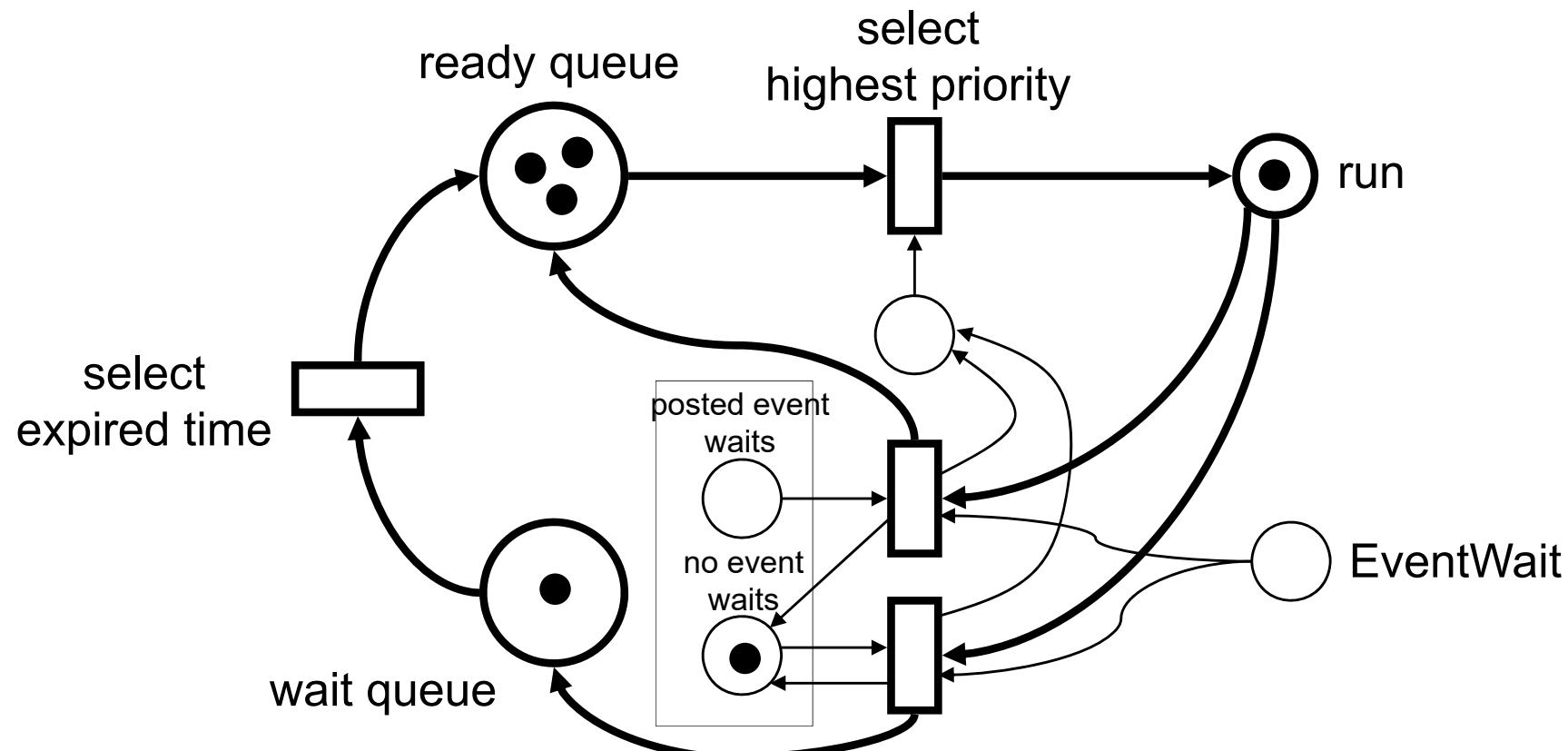
API for Co-Operative Scheduling

EventWait

sleep queue

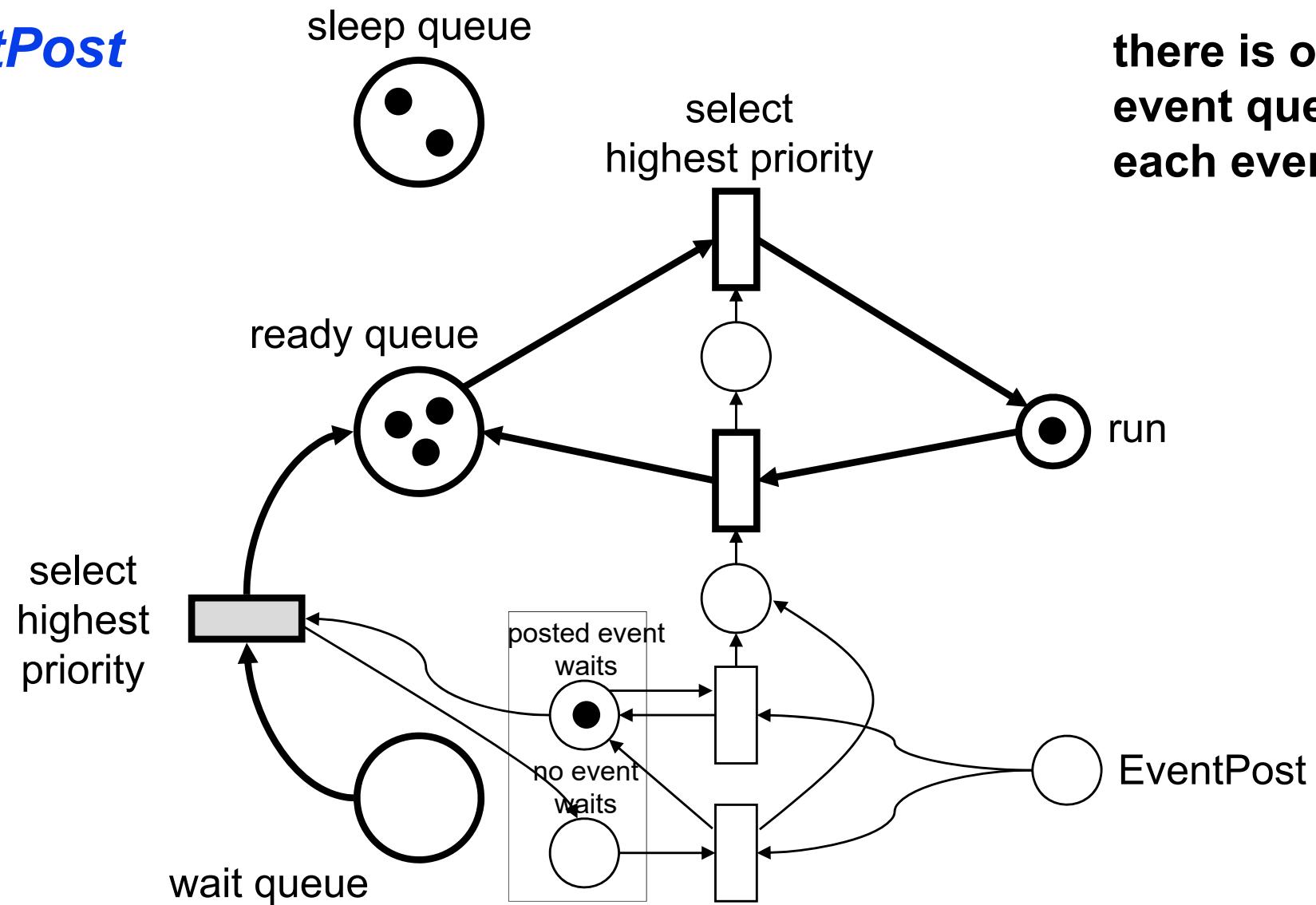


there is one event queue for each event type



API for Co-Operative Scheduling

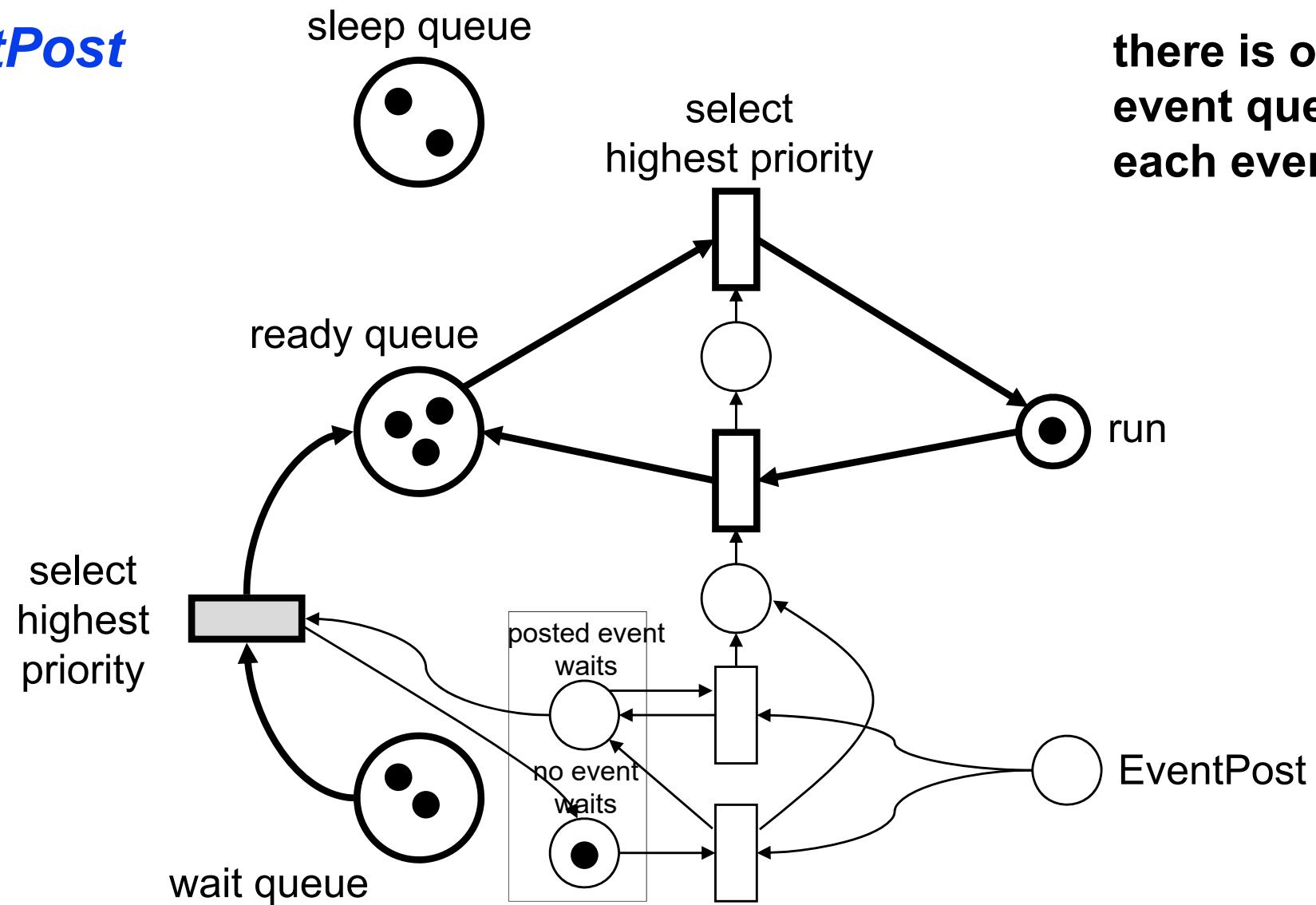
EventPost



there is one
event queue for
each event type

API for Co-Operative Scheduling

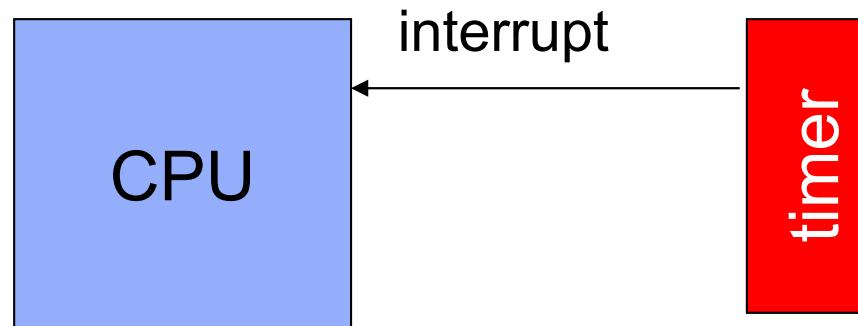
EventPost



there is one
event queue for
each event type

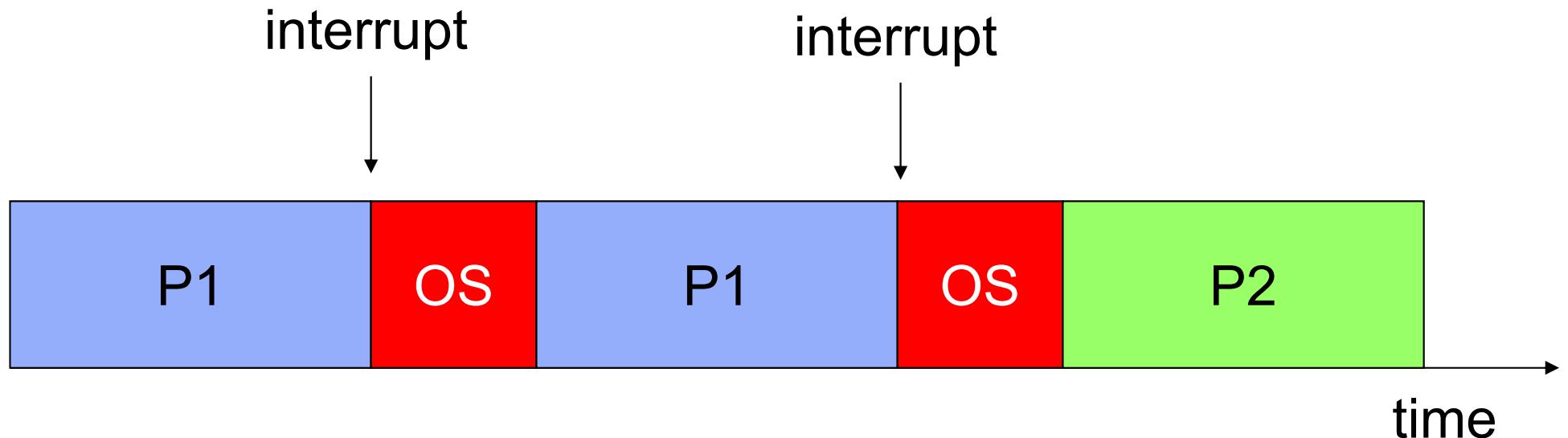
Preemptive Multitasking

- ▶ **Most powerful form of multitasking:**
 - Scheduler (OS) controls when contexts switches;
 - Scheduler (OS) determines what process runs next.
- ▶ Use of **timers** to call OS and switch contexts:



- ▶ Use **hardware or software interrupts**, or direct calls to OS routines to switch context.

Flow of Control with Preemption



Timing Peculiarities in Modern Computer Architectures

- The following example is taken from an exercise in “Systemprogrammierung”.
- It was not! constructed for challenging the timing predictability of modern computer architectures; the strange behavior was found by chance.
- A straightforward GCD algorithm was executed on an UltraSparc (Sun) architecture and timing was measured.
- ***Goal in this lecture:*** Determine the cause(s) for the strange timing behavior.

Program

- Only the relevant assembler program is shown (and the related C program); the calling *main* function just jumps to label *ggt* 1.000.000 times.

```
.text  
.global ggt  
.align 32
```

Here, we will introduce nop statements; there are NOT executed.

```
ggt:           ! %o0:= x, %o1 := y  
    cmp    %o0, %o1  
    blu,a ggt          ! if (%o0 < %o1) {goto ggt;}  
    sub    %o1, %o0, %o1 ! %o1 = %o1 - %o0  
    bgu,a ggt          ! if (%o0 > %o1) {goto ggt;}  
    sub    %o0, %o1, %o0 ! %o0 = %o1 - %o0  
    retl  
    nop
```

```
int ggt_c (int x, int y) {  
    while (x != y) {  
        if (x < y) { y -= x; }  
        else { x -= y; }  
    }  
    return (x);  
}
```

Observation

- Depending on the number of nop statements before the **ggt** label, the execution time of **ggt(17, 17*97)** varies by a factor of almost 2. The execution time of **ggt(17*97, 17)** varies by a factor of more than 4.
- This behavior is periodic in the number of nop statements, i.e. it repeats after 8 nop statements.
- Measurements:

nop	time[s] ggt(17,17*97)	time[s] ggt(17*97,17)
0	0.36	0.62
1	0.35	2.78
2	0.36	0.64
3	0.35	2.79

nop	time[s] ggt(17,17*97)	time[s] ggt(17*97,17)
4	0.37	0.63
5	0.35	0.62
6	0.65	0.64
7	0.64	0.63

Simple Calculations

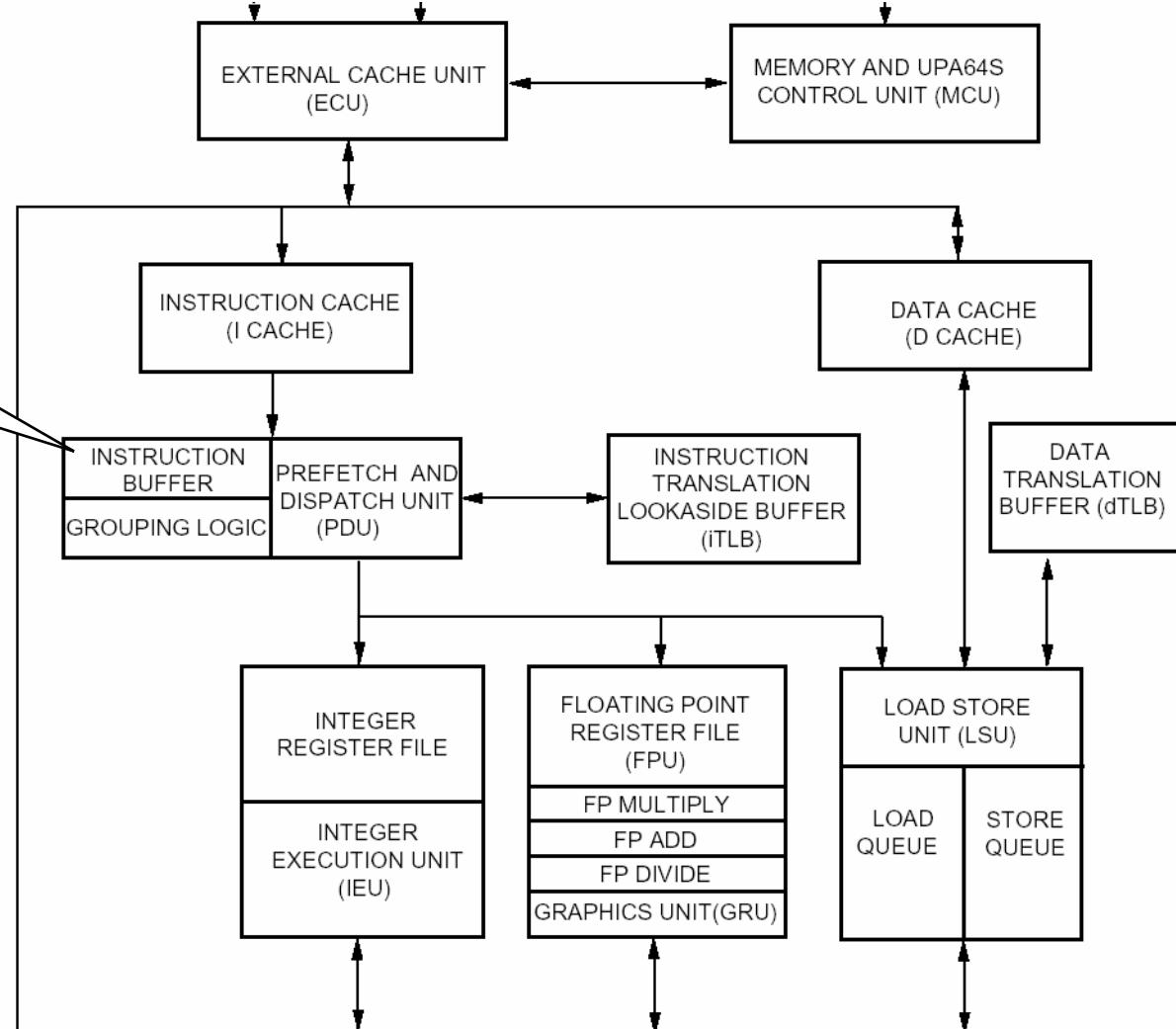
- The CPU is UltraSparc with 360 MHz clock rate.
- Problem 1 (ggt(17,17*97)):
 - Fast execution: $96*3*1.000.000 / 0.35 = 823$ MIPS and $0.35 * 360 / 96 = 1.31$ cycles per iteration.
 - Slow execution: $96*3*1.000.000 / 0.65 = 443$ MIPS and $0.65 * 360 / 96 = 2.44$ cycles per iteration.
 - Therefore, the difference is about 1 cycle per iteration.
- Problem 2 (ggt(17*97, 17)):
 - Fast execution: $96*4*1.000.000 / 0.63 = 609$ MIPS and $0.63 * 360 / 96 = 2.36$ cycles per iteration.
 - Slow execution: $96*4*1.000.000 / 2.78 = 138$ MIPS and $2.78 * 360 / 96 = 10.43$ cycles per iteration.
 - Therefore, the difference is about 8 cycles per iteration.

Explanations

- **Problem 1 ($\text{ggt}(17, 17*97)$):**
 - The first three instructions (cmp, blu, sub) are called 96 times before *ggt* returns. The timing behavior depends on the location of the program in address space.
 - The reason is most probably the implementation of the 4 word instruction buffer between the instruction cache and the pipeline: The instruction buffer can not be filled by different cache lines in one cycle.
 - In the slow execution, one needs to fill the instruction buffer twice for each iteration. This needs at least two cycles (despite of any parallelism in the pipeline).

Block Diagram of UltraSparc

Instruction buffer
for hiding latency
to cache



Instruction Availability

Instruction dispatch is limited to the number of instructions available in the instruction buffer. Several factors limit instruction availability. UltraSPARC-II*i* fetches up to four instructions per clock from an aligned group of eight instructions.

When the fetch address (modulo 32) is equal to 20, 24, or 28, then three, two, or one instruction(s) respectively are added to the instruction buffer. The next cache line and set are predicted using a next field and set predictor for each aligned four instructions in the instruction cache. When a set or next field mispredict occurs, instructions are not added to the instruction buffer for two clocks.

Address Alignment

0 nop

Cache line:

cmp	blu	sub	...				
-----	-----	-----	-----	--	--	--	--

Instruction buffer:

cmp	blu	sub	...
-----	-----	-----	-----

5 nop

Cache line:

nop	nop	nop	nop	nop	cmp	blu	sub
-----	-----	-----	-----	-----	-----	-----	-----

Instruction buffer:

cmp	blu	sub	
-----	-----	-----	-------------------------------------------------------------------------------------

6 nop

Cache lines:

nop	nop	nop	nop	nop	nop	cmp	blu
sub

Instruction buffer:

cmp	blu		
-----	-----	--------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------

2 fetches are necessary
as sub is missing

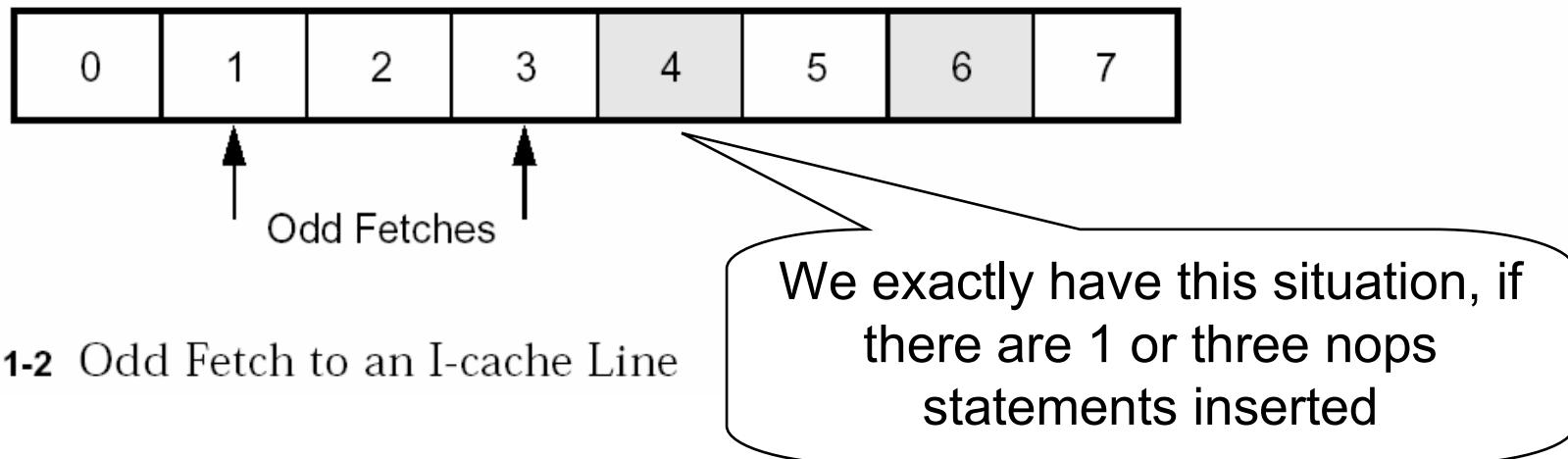
Explanations

- **Problem 2 ($\text{ggt}(17*97, 17)$):**
 - The loop is executed (cmp, blu, sub, bgu, sub) 96 times, where the first sub instruction is not executed (since blu is used with ',a' suffix, which means, that instruction in the delay slot is not executed if branch is not taken). Therefore, there are four instructions to be executed, but the loop has 5 instructions in total.
 - The main reason for this behavior is most probably due to the branch prediction scheme used in the architecture.
 - In particular, there is a prediction ***of the next block of 4 instructions*** to be fetched into the instruction buffer. This scheme is based on a two bit predictor and is also used to control the pipeline and to prevent stalls.
 - But there is a problem due to the optimization of the state information that is stored (prediction for blocks of instructions and single instructions):

User Manual (page 342 ...)

The following cases represent situations when the prediction bits and/or the next field do not operate optimally:

1. When the target of a branch is word 1 or word 3 of an I-cache line (FIGURE 21-2) and the fourth instruction to be fetched (instruction 4 and 6 respectively) is a branch, the branch prediction bits from the wrong pair of instructions are used.



Conclusions

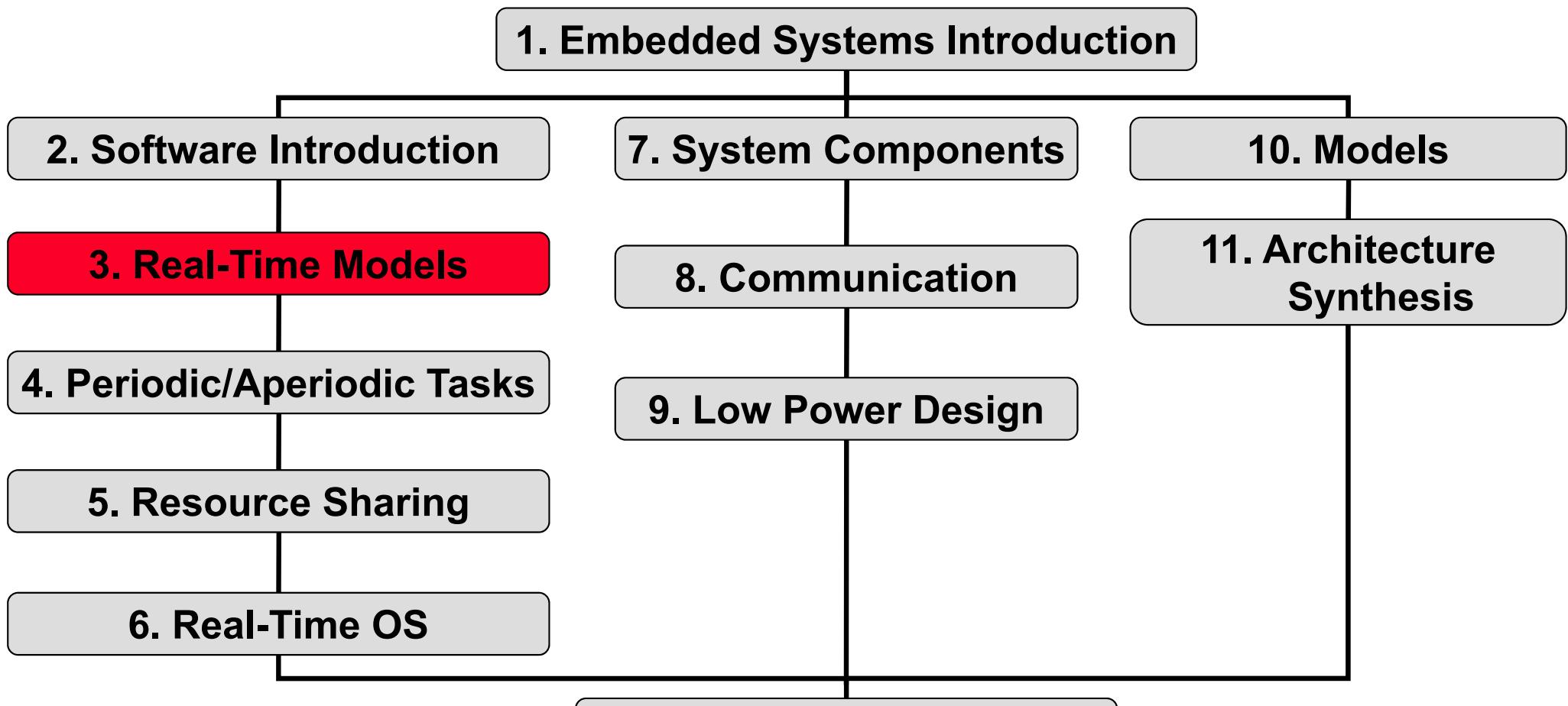
- Innocent changes (just moving code in address space) can easily change the timing by a factor of 4.
- In our example, the timing oddities are caused by two different architectural features of modern superscalar processors:
 - branch prediction
 - instruction buffer
- It is hard to predict the timing of modern processors; this is bad in all situations, where timing is of importance (embedded systems, hard real-time systems).
- What is a proper approach to predictable system design ?

Embedded Systems

3. Real-Time Models

Lothar Thiele

Contents of Course



*Software and
Programming*

*Processing and
Communication*

Hardware

Basic Terms

► *Real-time systems*

- **Hard**: A real-time task is said to be hard, if missing its deadline may cause catastrophic consequences on the environment under control. Examples are sensory data acquisition, detection of critical conditions, actuator servoing.
- **Soft**: A real-time task is called soft, if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior. Examples are command interpreter of the user interface, displaying messages on the screen.

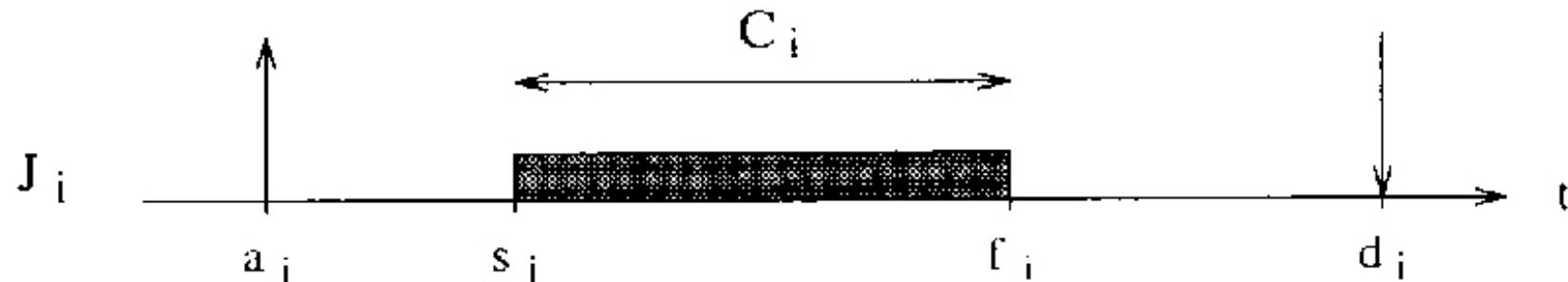
Schedule

- ▶ Given a set of **tasks** $J = \{J_1, J_2, \dots\}$:
 - A **schedule** is an assignment of tasks to the processor, such that each task is executed until completion.
 - A **schedule** can be defined as an integer step function $\sigma : R \rightarrow N$ where $\sigma(t)$ denotes the task which is executed at time t . If $\sigma(t) = 0$ then the processor is called **idle**.
 - If $\sigma(t)$ changes its value at some time, then the processor performs a **context switch**.
 - Each interval, in which $\sigma(t)$ is constant is called a **time slice**.
 - A **preemptive schedule** is a schedule in which the running task can be arbitrarily suspended at any time, to assign the CPU to another task according to a predefined scheduling policy.

Schedule and Timing

- ▶ A schedule is said to be **feasible**, if all task can be completed according to a set of specified constraints.
- ▶ A set of tasks is said to be **schedulable**, if there exists at least one algorithm that can produce a feasible schedule.
- ▶ **Arrival time** a_i or **release time** r_i is the time at which a task becomes ready for execution.
- ▶ **Computation time** C_i is the time necessary to the processor for executing the task without interruption.
- ▶ **Deadline** d_i is the time at which a task should be completed.
- ▶ **Start time** s_i is the time at which a task starts its execution.
- ▶ **Finishing time** f_i is the time at which a task finishes its execution.

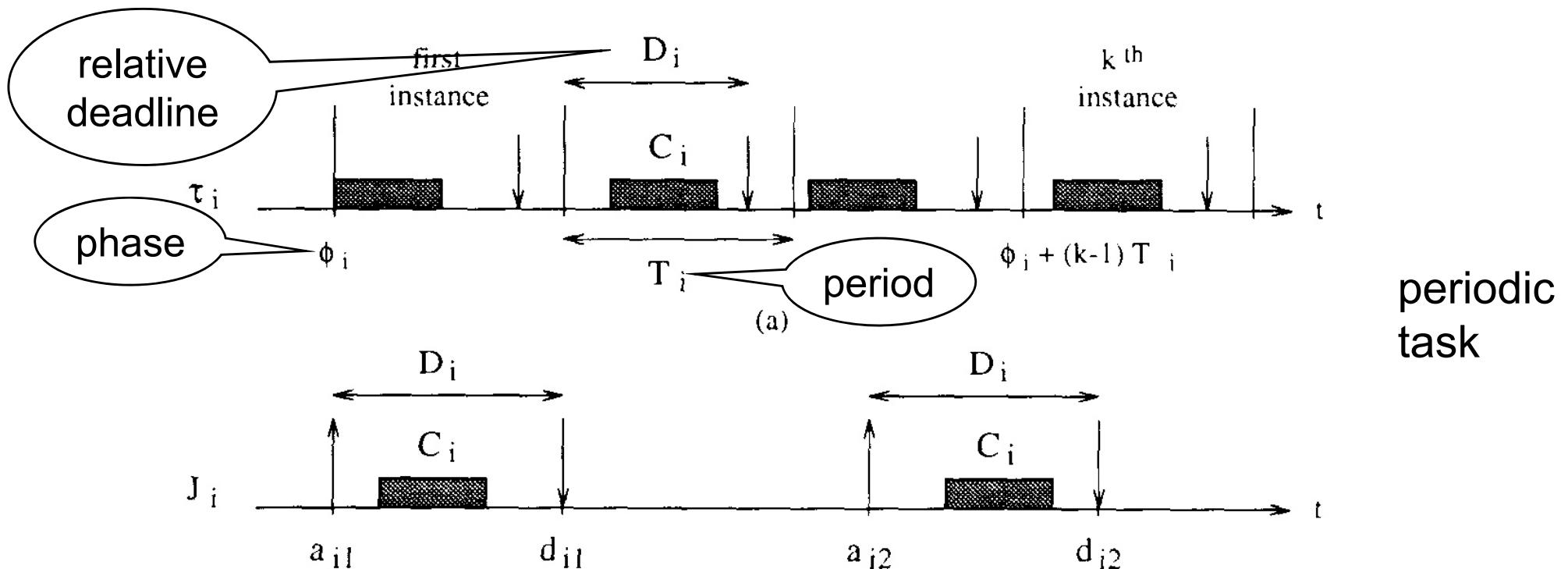
Schedule and Timing



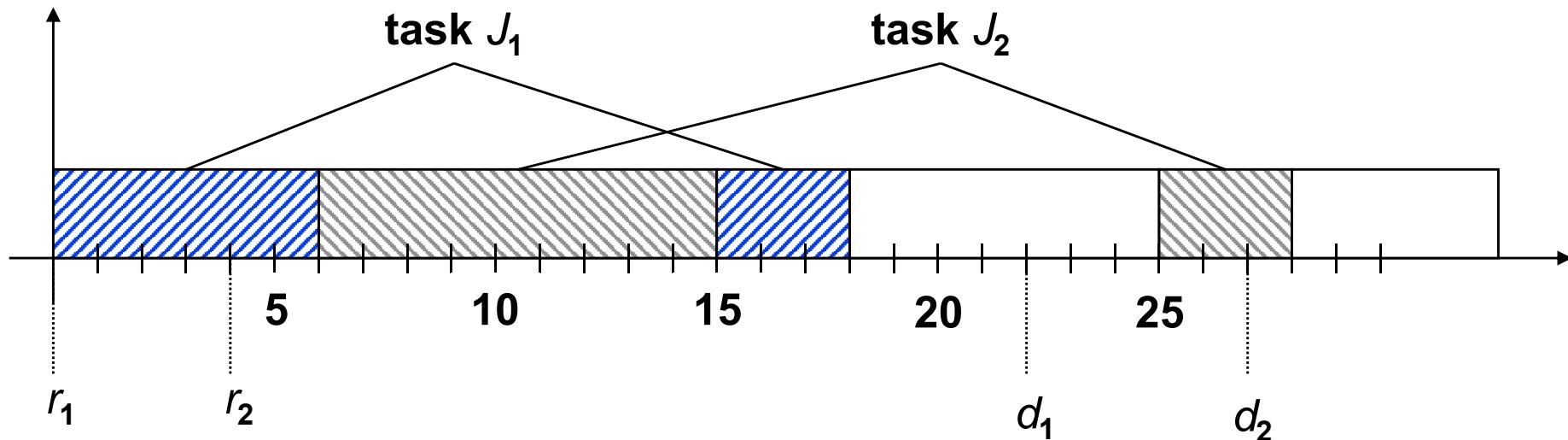
- ▶ Using the above definitions, we have $d_i \geq r_i + C_i$
- ▶ **Lateness** $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline; note that if a task completes before the deadline, its lateness is negative.
- ▶ **Tardiness or exceeding time** $E_i = \max(0, L_i)$ is the time a task stays active after its deadline.
- ▶ **Laxity or slack time** $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its activation to complete within its deadline.

Schedule and Timing

- ▶ **Periodic task** τ_i : infinite sequence of identical activities, called instances or jobs, that are regularly activated at a constant rate with period T_i . The activation time of the first instance is called phase Φ_i .



Example



Computation times: $C_1 = 9, C_2 = 12$

Start times: $s_1 = 0, s_2 = 6$

Finishing times: $f_1 = 18, f_2 = 28$

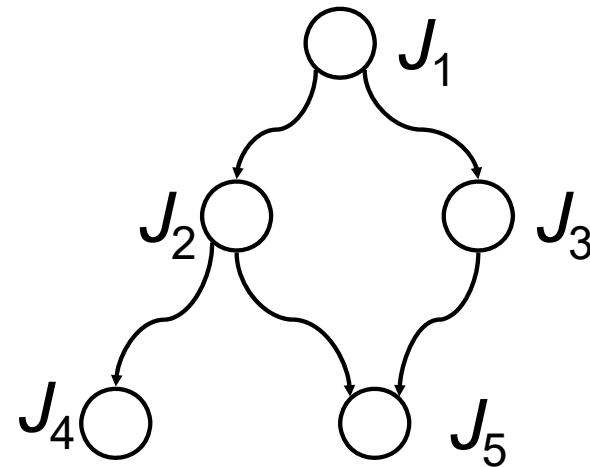
Lateness: $L_1 = -4, L_2 = 1$

Tardiness: $E_1 = 0, E_2 = 1$

Laxity: $X_1 = 13, X_2 = 11$

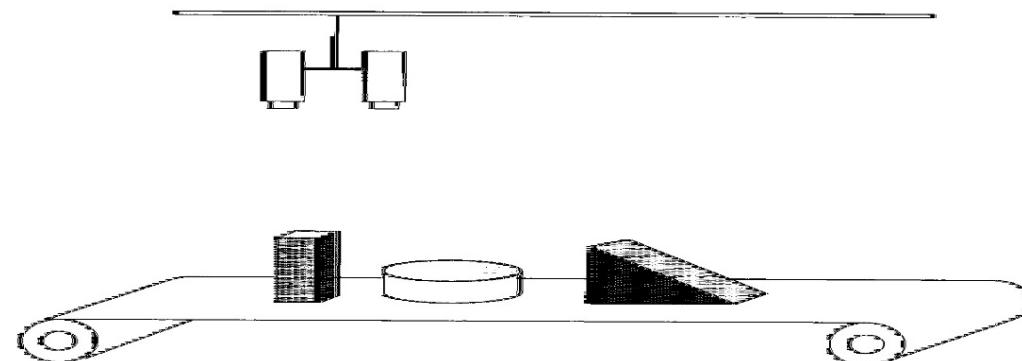
Precedence Constraints

- ▶ **Precedence relations** between graphs can be described through an acyclic directed graph G where tasks are represented by nodes and precedence relations by arrows. G induces a partial order on the task set.
- ▶ There are different interpretations possible:
 - All successors of a task are activated (concurrent task execution).
 - One successor of a task is activated (non-deterministic choice).

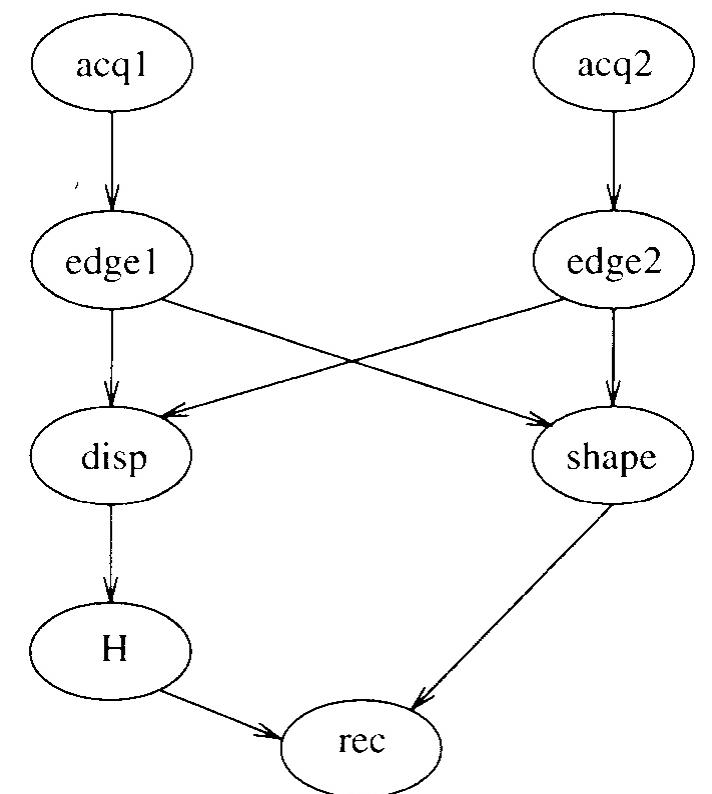


Precedence Constraints

- ▶ Example (concurrent activation):



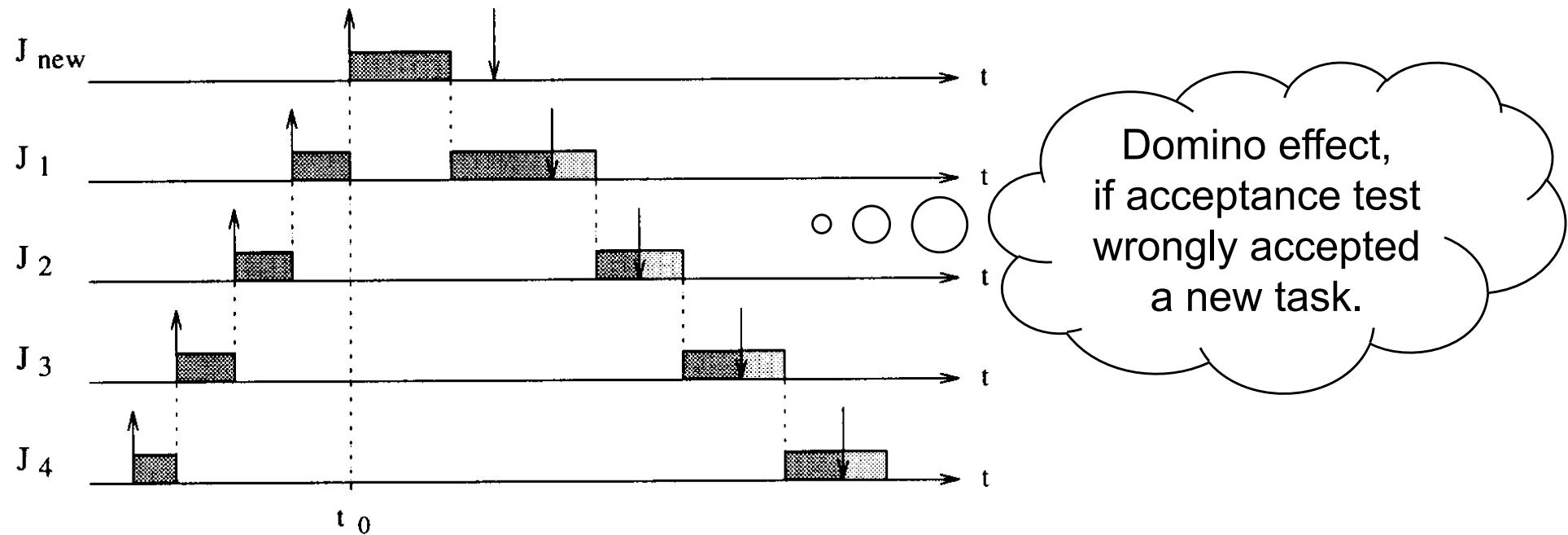
- Image acquisition $acq1$ $acq2$
- Low level image processing $edge1$ $edge2$
- Feature/contour extraction $shape$
- Pixel disparities $disp$
- Object size H
- Object recognition rec



Classification of Scheduling Algorithms

- ▶ With ***preemptive algorithms***, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy.
- ▶ With a ***non-preemptive algorithm***, a task, once started, is executed by the processor until completion.
- ▶ ***Static algorithms*** are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.
- ▶ ***Dynamic algorithms*** are those in which scheduling decisions are based on dynamic parameters that may change during system execution.

Classification of Scheduling Algorithms



- ▶ An algorithm is said **optimal** if it minimizes some given cost function defined over the task set.
- ▶ An algorithm is said to be **heuristic** if it tends toward but does not guarantee to find the optimal schedule.

Metrics

- ▶ Average response time:
- ▶ Total completion time:
- ▶ Weighted sum of response time:
- ▶ Maximum lateness:
- ▶ Number of late tasks:

$$\bar{t}_r = \frac{1}{n} \sum_{i=1}^n (f_i - r_i)$$

$$t_c = \max_i(f_i) - \min_i(r_i)$$

$$t_w = \frac{\sum_{i=1}^n w_i (f_i - r_i)}{\sum_{i=1}^n w_i}$$

$$L_{\max} = \max_i(f_i - d_i)$$

$$N_{\text{late}} = \sum_{i=1}^n miss(f_i)$$

$$miss(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$$

Metrics

- ▶ Average response time:
- ▶ Total completion time:
- ▶ Weighted sum of response time:
- ▶ **Maximum lateness:**
- ▶ **Number of late tasks:**

$$\bar{t}_r = \frac{1}{n} \sum_{i=1}^n (f_i - r_i)$$

$$t_c = \max_i(f_i) - \min_i(r_i)$$

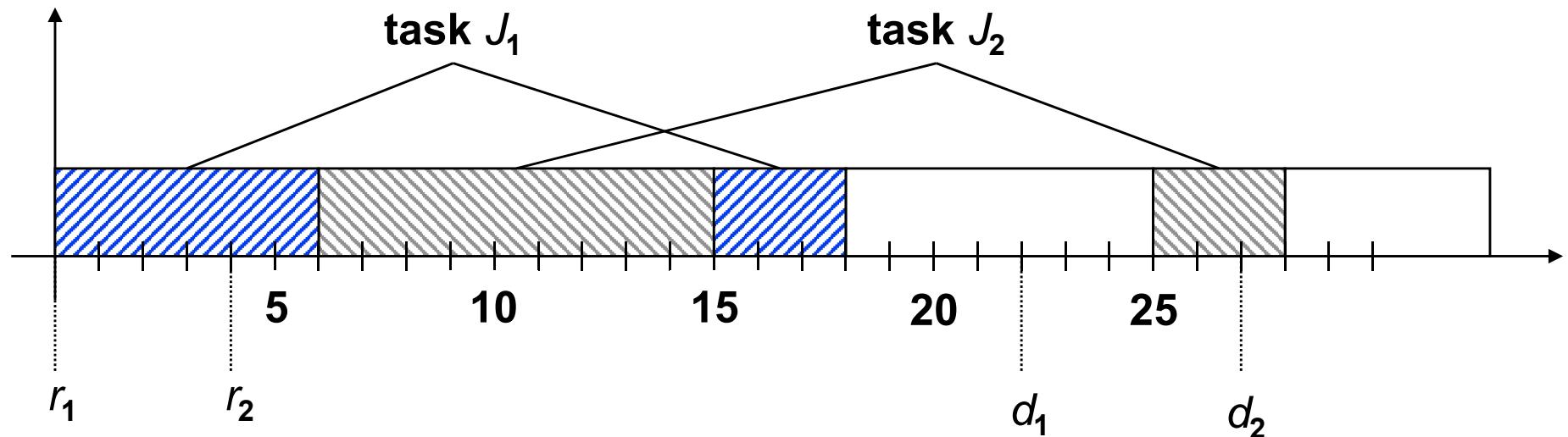
$$t_w = \frac{\sum_{i=1}^n w_i (f_i - r_i)}{\sum_{i=1}^n w_i}$$

$$L_{\max} = \max_i(f_i - d_i)$$

$$N_{\text{late}} = \sum_{i=1}^n miss(f_i)$$

$$miss(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$$

Metrics Example



Average response time:

$$\bar{t}_r = \frac{1}{2}(18 + 24) = 21$$

Total completion time:

$$t_c = 28 - 0 = 28$$

Weighted sum of response times:

$$w_1 = 2, w_2 = 1: t_w = \frac{2 \cdot 18 + 24}{3} = 20$$

Number of late tasks:

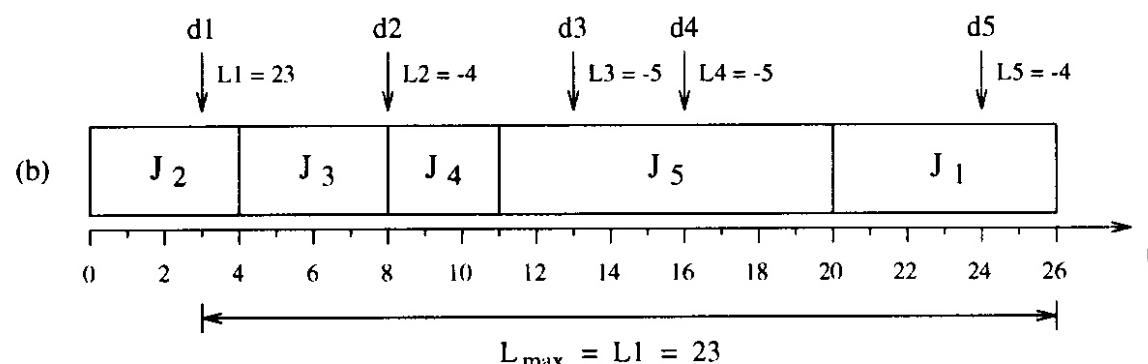
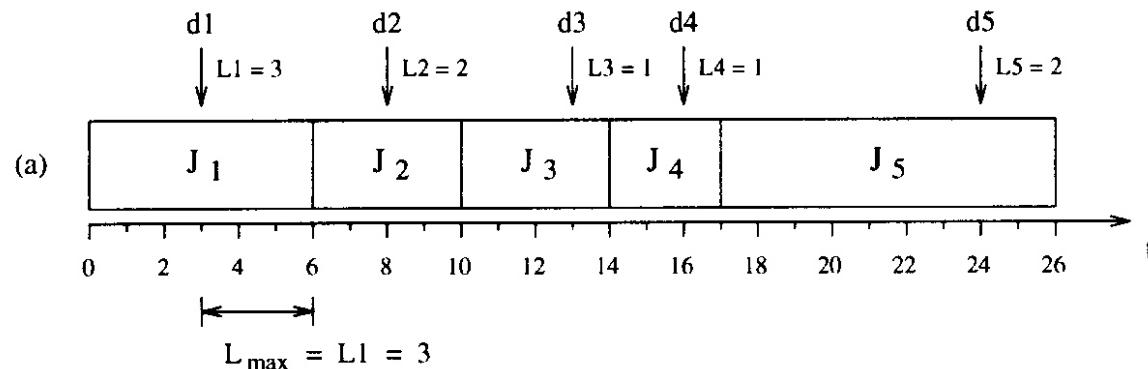
$$N_{\text{late}} = 1$$

Maximum lateness:

$$L_{\max} = 1$$

Scheduling Example

- ▶ In (a), the maximum lateness is minimized, but all tasks miss their deadlines.
- ▶ In (b), the maximal lateness is larger, but only one task misses its deadline.

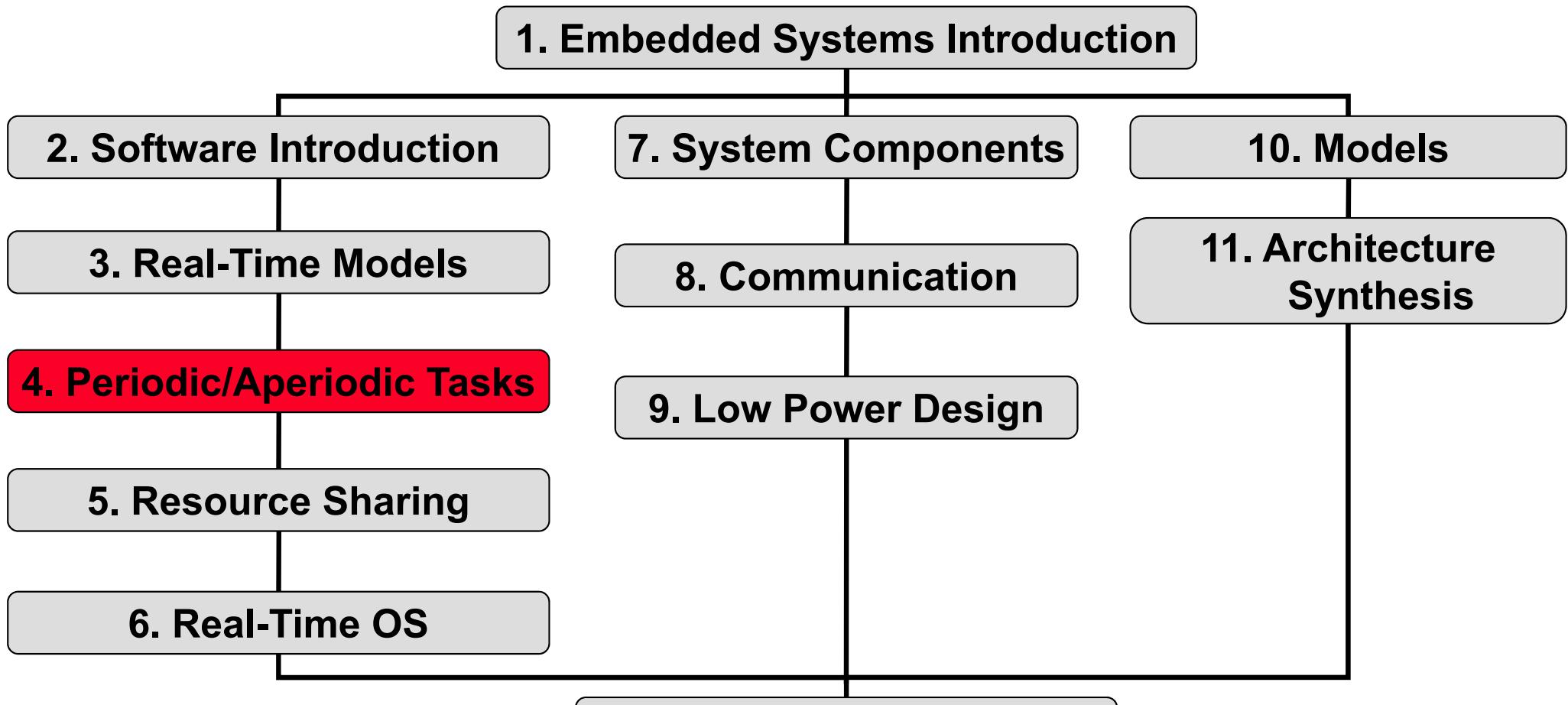


Embedded Systems

4. Aperiodic and Periodic Tasks

Lothar Thiele

Contents of Course



*Software and
Programming*

*Processing and
Communication*

Hardware

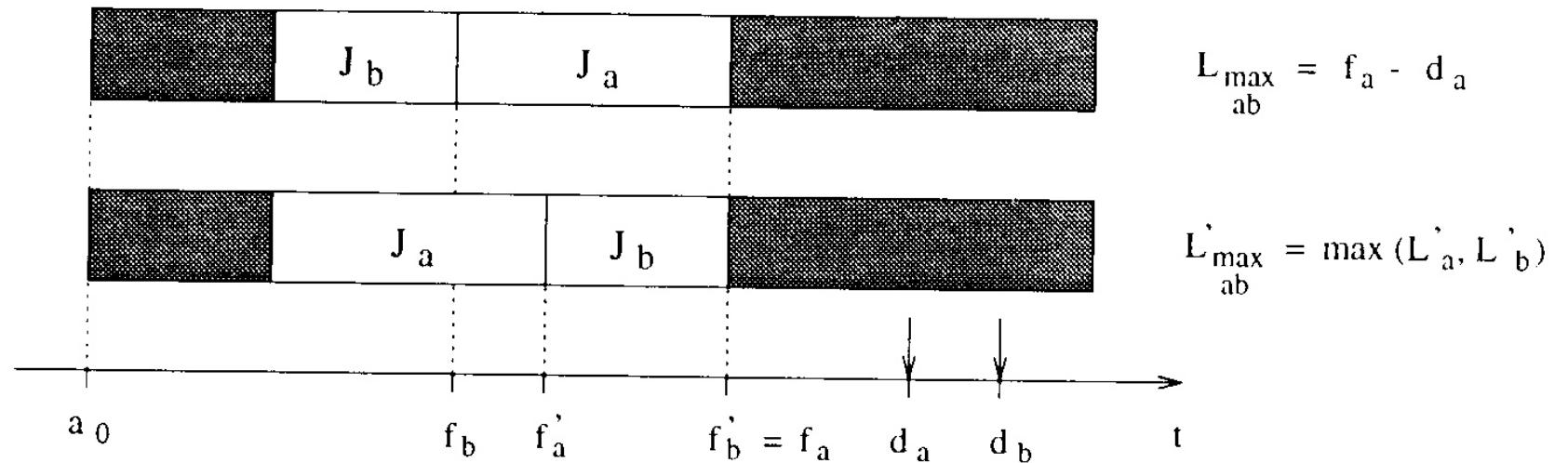
Overview

- ▶ Scheduling of *aperiodic tasks* with real-time constraints:
 - Table with some known algorithms:

	Equal arrival times non preemptive	Arbitrary arrival times preemptive
Independent tasks	EDD (Jackson)	EDF (Horn)
Dependent tasks	LDF (Lawler)	EDF* (Chetto)

Earliest Deadline Due (EDD)

- ▶ **Jackson's rule:** Given a set of n tasks. Processing in order of non-decreasing deadlines is optimal with respect to minimizing the maximum lateness.
- ▶ **Proof concept:**



if ($L'_a \geq L'_b$) then $L'_{\max}^{ab} = f'_a - d_a < f_a - d_a$

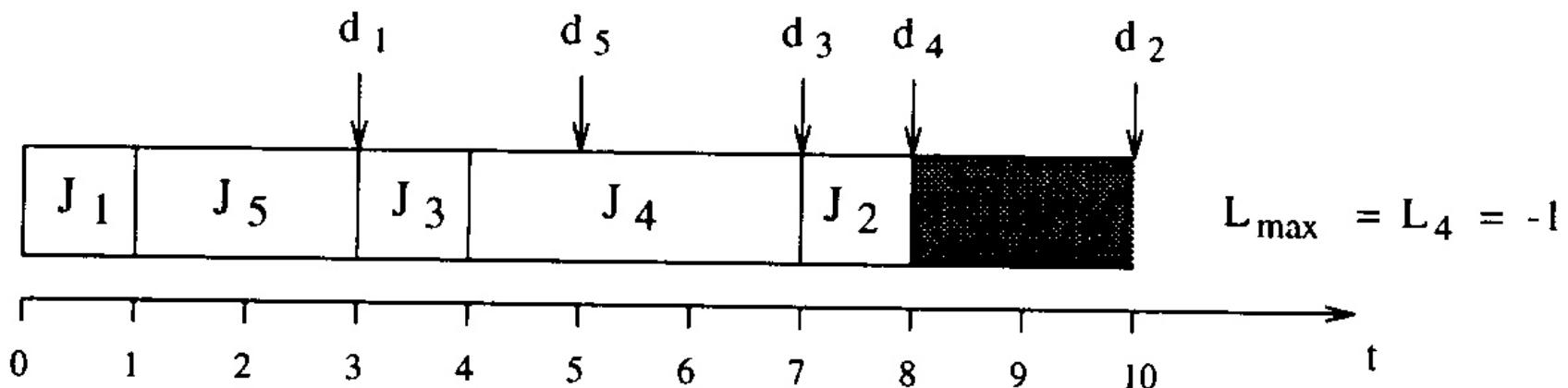
if ($L'_a \leq L'_b$) then $L'_{\max}^{ab} = f'_b - d_b < f_a - d_a$

in both cases: $L'_{\max}^{ab} < L_{\max}^{ab}$

Earliest Deadline Due (EDD)

► Example 1:

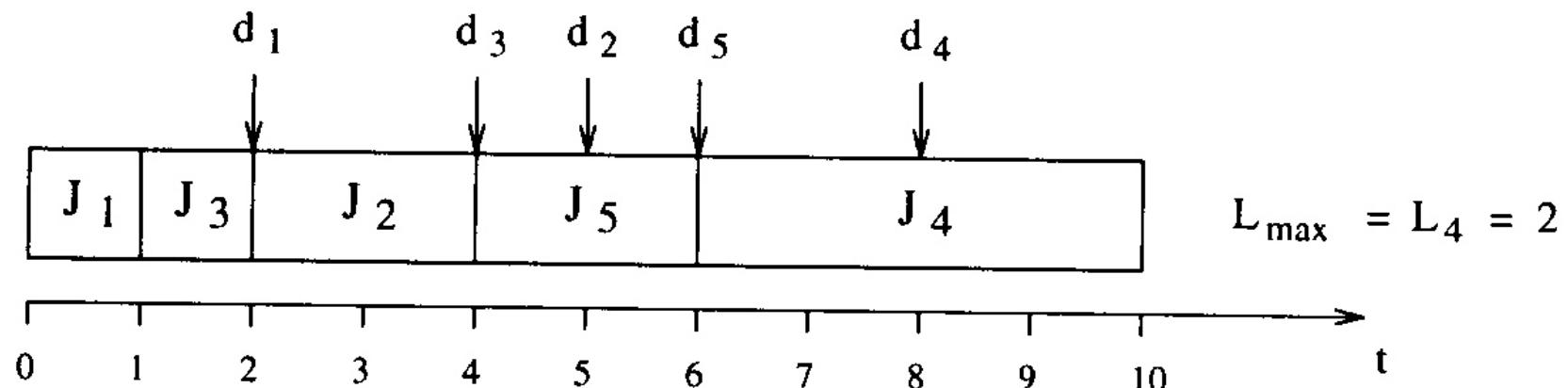
	J ₁	J ₂	J ₃	J ₄	J ₅
C _i	1	1	1	3	2
d _i	3	10	7	8	5



Earliest Deadline Due (EDD)

► Example 2:

	J ₁	J ₂	J ₃	J ₄	J ₅
C _i	1	2	1	4	2
d _i	2	5	4	8	6



Earliest Deadline First (EDF)

- ▶ **Horn's rule:** Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any instant executes the task with the earliest absolute deadline among the ready tasks is optimal with respect to minimizing the maximum lateness.
- ▶ **Concept of proof:** For each time interval $[t, t + 1]$ it is verified, whether the actual running task is the one with the earliest absolute deadline. If this is not the case, the task with the earliest absolute deadline is executed in this interval instead. This operation cannot increase the maximum lateness.

Earliest Deadline First (EDF)

- ▶ Used quantities and terms:
 - $\sigma(t)$ identifies the task executing in the slice $[t, t + 1)$
 - $E(t)$ identifies the ready task that, at time t , has the earliest deadline
 - $t_E(t)$ is the time ($\geq t$) at which the next slice of task $E(t)$ begins its execution in the current schedule

Earliest Deadline First (EDF)

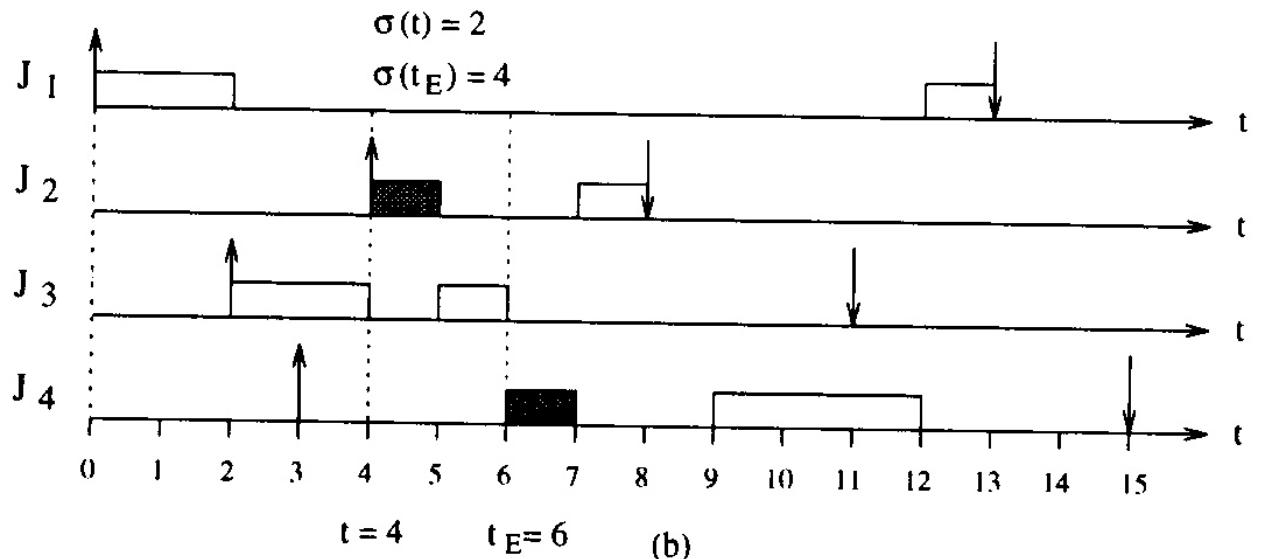
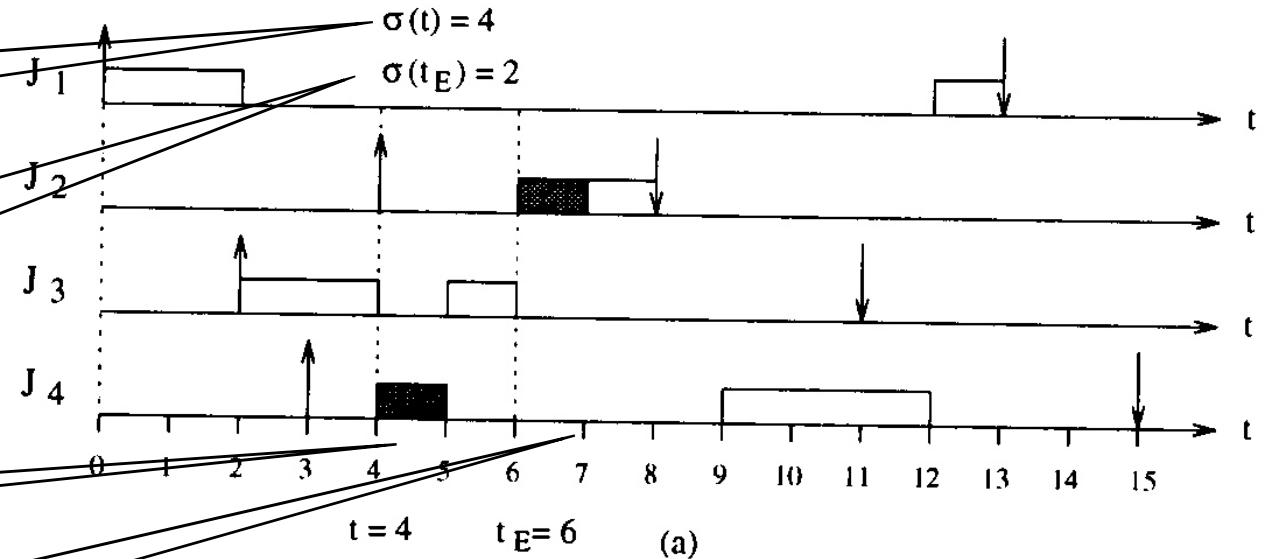
which task is executing ?

which task has earliest deadline ?

time slice

slice for interchange

situation after interchange



Earliest Deadline First (EDF)

remaining worst-case execution time of task k

► **Guarantee:**

- worst case finishing time of task i: $f_i = t + \sum_{k=1}^i c_k(t)$

- EDF guarantee condition:

$$\forall i = 1, \dots, n \quad t + \sum_{k=1}^i c_k(t) \leq d_i$$

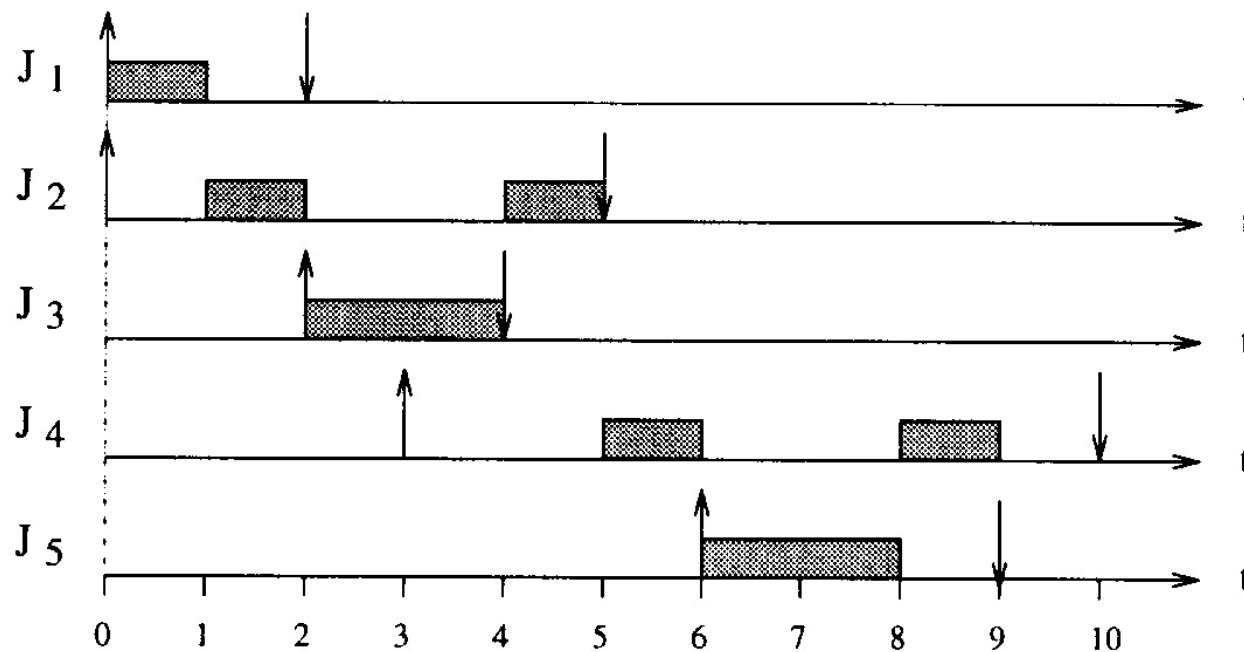
- algorithm:

```
Algorithm: EDF_guarantee (J, Jnew)
{
    J' = J ∪ {Jnew}; /* ordered by deadline */
    t = current_time();
    f0 = t;
    for (each Ji ∈ J') {
        fi = fi-1 + ci(t);
        if (fi > di) return(INFEASIBLE);
    }
    return(FEASIBLE);
}
```

Earliest Deadline First (EDF)

► Example:

	J ₁	J ₂	J ₃	J ₄	J ₅
a _i	0	0	2	3	6
C _i	1	2	2	2	2
d _i	2	5	4	10	9



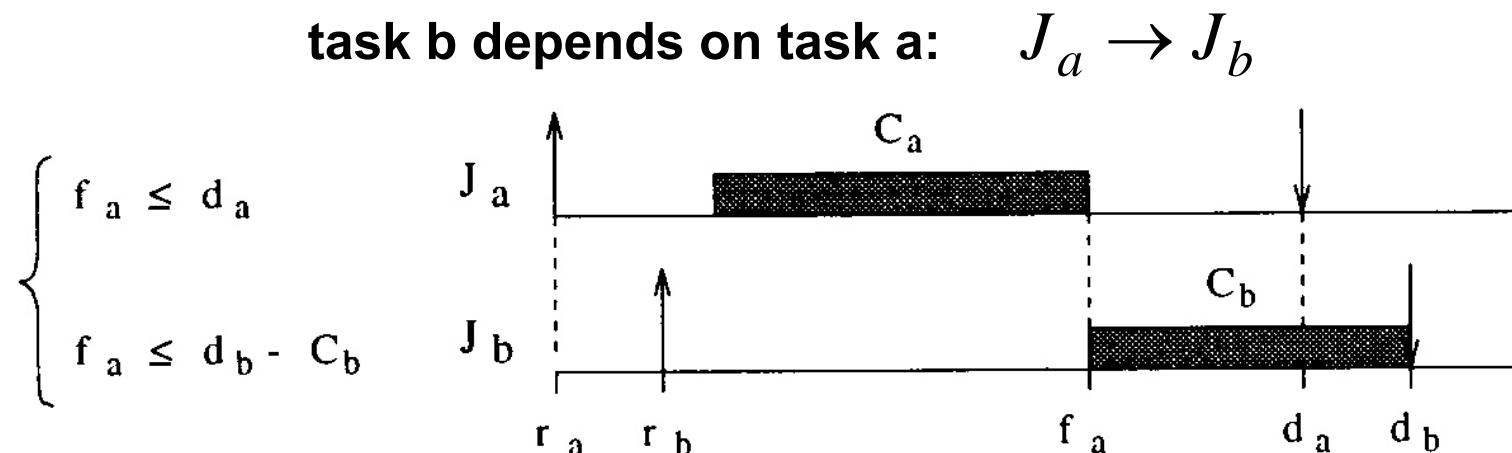
Earliest Deadline First (EDF*)

- ▶ The problem of scheduling a set of *n tasks with precedence constraints* (concurrent activation) can be solved in polynomial time complexity if tasks are preemptable.
- ▶ The *EDF* algorithm* determines a feasible schedule in the case of tasks with precedence constraints if there exists one.
- ▶ By the modification it is guaranteed that if there exists a valid schedule at all then
 - a task starts execution not earlier than its release time and not earlier than the finishing times of its predecessors (a task cannot preempt any predecessor)
 - all tasks finish their execution within their deadlines

Earliest Deadline First (EDF*)

► *Modification of deadlines:*

- Task must finish the execution time within its deadline
- Task must not finish the execution later than the maximum start time of its successor

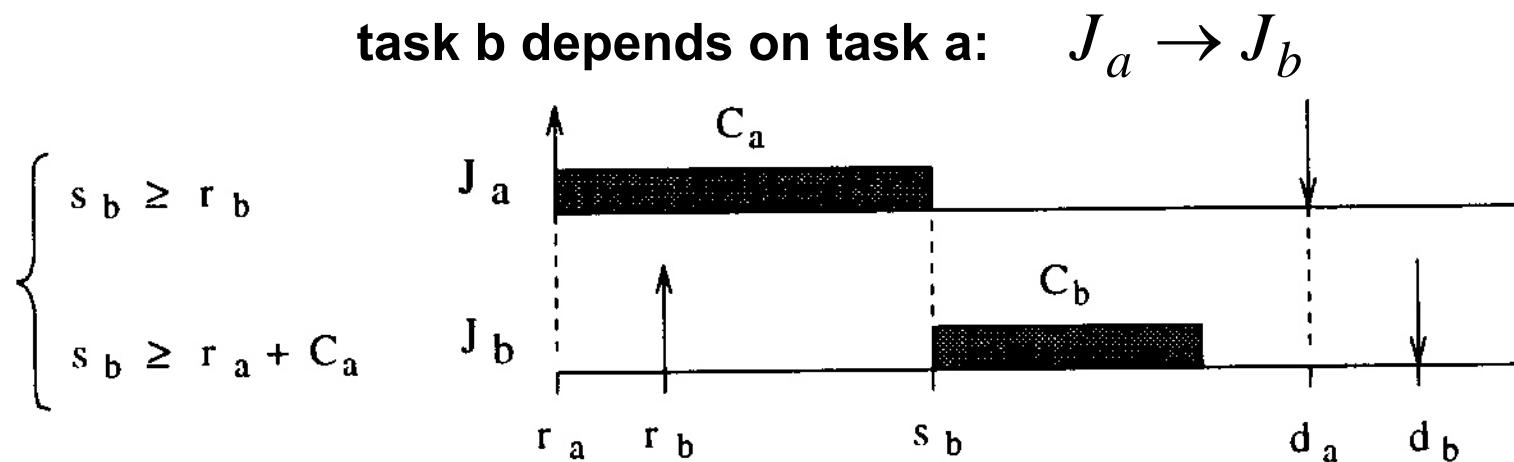


- Solution: $d_i^* = \min(d_i, \min(d_j^* - C_j : J_i \rightarrow J_j))$

Earliest Deadline First (EDF*)

► *Modification of release times:*

- Task must start the execution not earlier than its release time.
- Task must not start the execution earlier than the minimum finishing time of its predecessor.



- Solution: $r_j^* = \max(r_j, \max(r_i^* + C_i : J_i \rightarrow J_j))$

Earliest Deadline First (EDF*)

- **Algorithm** for modification of release times:
 1. For any initial node of the precedence graph set $r_i^* = r_i$
 2. Select a task j such that its release time has not been modified but the release times of all immediate predecessors i have been modified. If no such task exists, exit.
 3. Set $r_j^* = \max(r_j, \max(r_i^* + C_i : J_i \rightarrow J_j))$
 4. Return to step 2
- **Algorithm** for modification of deadlines:
 1. For any terminal node of the precedence graph set $d_i^* = d_i$
 2. Select a task i such that its deadline has not been modified but the deadlines of all immediate successors j have been modified. If no such task exists, exit.
 3. Set $d_i^* = \min(d_i, \min(d_j^* - C_j : J_i \rightarrow J_j))$
 4. Return to step 2

Earliest Deadline First (EDF*)

► *Proof concept*

- Show that if there exists a feasible schedule for the modified task set under EDF then the original task set is also schedulable (ignoring precedence relations). To this end, show that the original task set meets the timing constraints also. This can be done by using $d_i^* \leq d_i$, $r_i^* \geq r_i$.
- Show the reverse also.
- In addition, show that the precedence relations in the original task set are not violated. In particular, show that
 - a task cannot start before its predecessor and
 - a task cannot preempt its predecessor.

Overview

- ▶ Table of some known *preemptive scheduling* algorithms for *periodic tasks*:

	Deadline equals period	Deadline smaller than period
static priority	RM (rate-monotonic) EDF	DM (deadline-monotonic) EDF*
dynamic priority		

Model of Periodic Tasks

- ▶ **Examples:** sensory data acquisition, low-level servoing, control loops, action planning and system monitoring. When a control application consists of several concurrent periodic tasks with individual timing constraints, the OS has to guarantee that each periodic instance is regularly activated at its proper rate and is completed within its deadline.

- ▶ **Definitions:**

Γ : denotes a set of periodic tasks

τ_i : denotes a generic periodic task

$\tau_{i,j}$: denotes the j th instance of task i

$r_{i,j}, s_{i,j}, f_{i,j}, d_{i,j}$:
denotes the release time, start time, finishing time,
absolute deadline of the j th instance of task i

Φ_i : phase of task i (release time of its first instance)

D_i : relative deadline of task i

Model of Periodic Tasks

- ▶ The following ***hypotheses*** are assumed on the tasks:
 - The instances of a periodic task are ***regularly activated*** at a constant rate. The interval T_i between two consecutive activations is called period. The release times satisfy

$$r_{i,j} = \Phi_i + (j-1)T_i$$

- All instances have the ***same worst case execution time*** C_i
- All instances of a periodic task have the ***same relative deadline*** D_i . Therefore, the absolute deadlines satisfy

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$

Model of Periodic Tasks

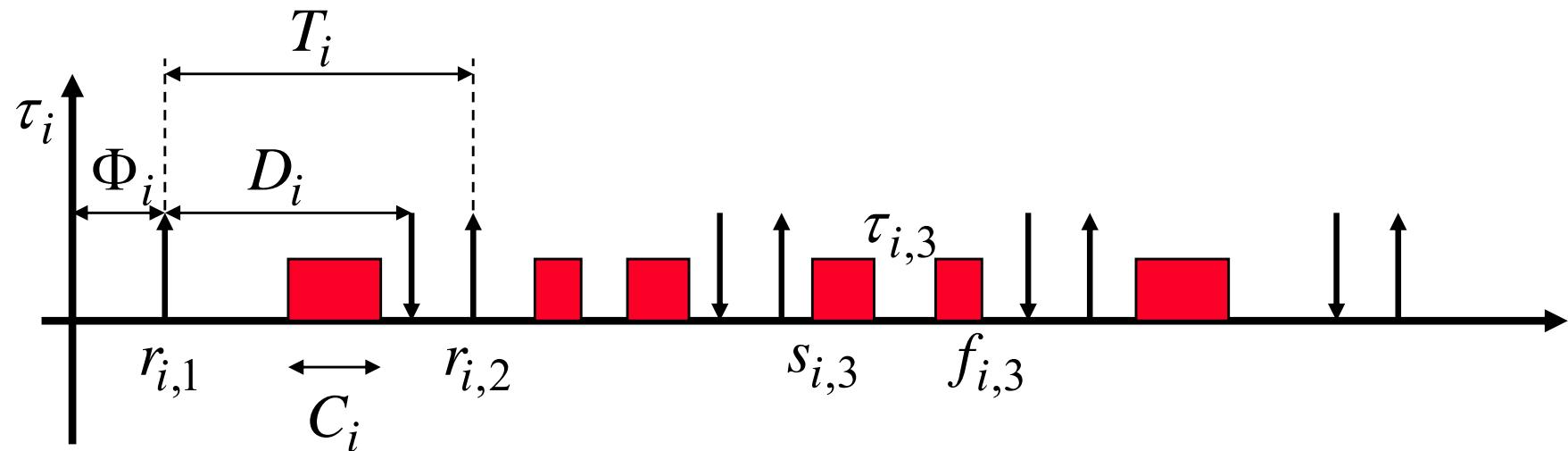
- ▶ The following ***hypotheses*** are assumed on the tasks cont':
 - Often, the relative deadline equals the period $D_i = T_i$ and therefore

$$d_{i,j} = \Phi_i + jT_i$$

- All periodic tasks are ***independent***; that is, there are no precedence relations and no resource constraints.
- No task can suspend itself, for example on I/O operations.
- All tasks are released as soon as they arrive.
- All overheads in the OS kernel are assumed to be zero.

Model of Periodic Tasks

- ▶ *Example:*



Rate Monotonic Scheduling (RM)

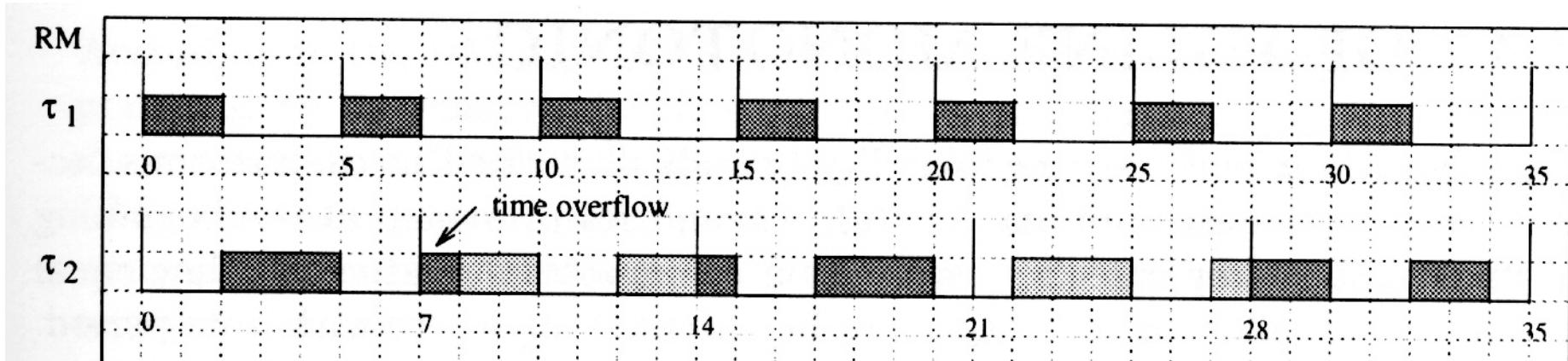
- ▶ **Assumptions:**

- Task priorities are assigned to tasks before execution and do not change over time (**static priority assignment**).
- RM is intrinsically **preemptive**: the currently executing task is preempted by a task with higher priority.
- **Deadlines** equal the periods $D_i = T_i$.

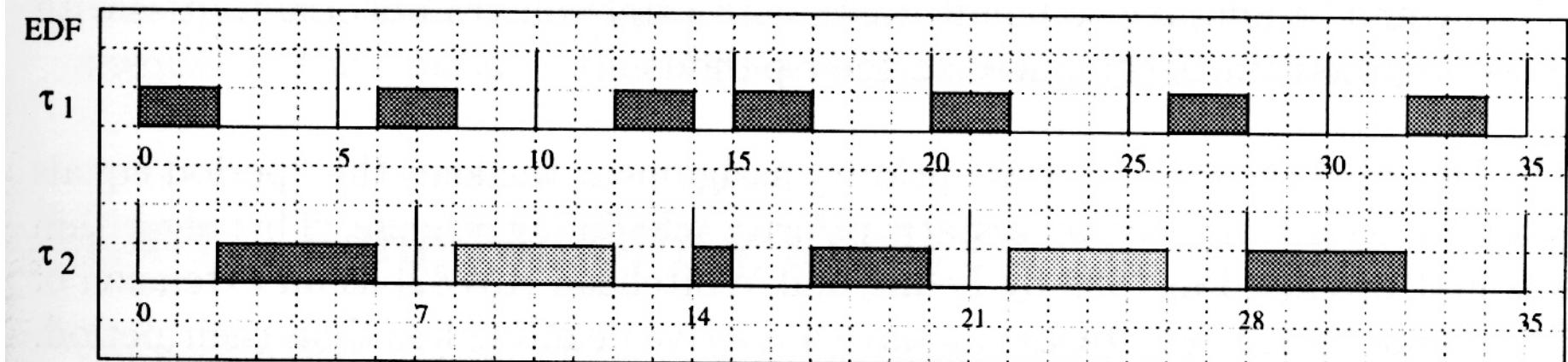
- ▶ **Algorithm:** Each task is assigned a priority. Tasks with higher request rates (that is with shorter periods) will have higher priorities. Tasks with higher priority interrupt tasks with lower priority.

Periodic Tasks

- ▶ **Example:** 2 tasks, deadline = periods, $U = 97\%$



(a)



(b)

Rate Monotonic Scheduling (RM)

- ▶ ***Optimality***: RM is optimal among all fixed-priority assignments in the sense that not other fixed-priority algorithm can schedule a task set that cannot be scheduled by RM.
- ▶ The ***proof*** is done by considering several cases that may occur, but the main ideas are as follows:
 - ***A critical instant for any task occurs whenever the task is released simultaneously with all higher priority tasks.***
The tasks schedulability can easily be checked at their critical instances. If all tasks are feasible at their critical instants, then the task set is schedulable in any other condition.
 - Show that, given two periodic tasks, if the schedule is feasible by an arbitrary priority assignment, then it is also feasible by RM.
 - Extend the result to a set of n periodic tasks.

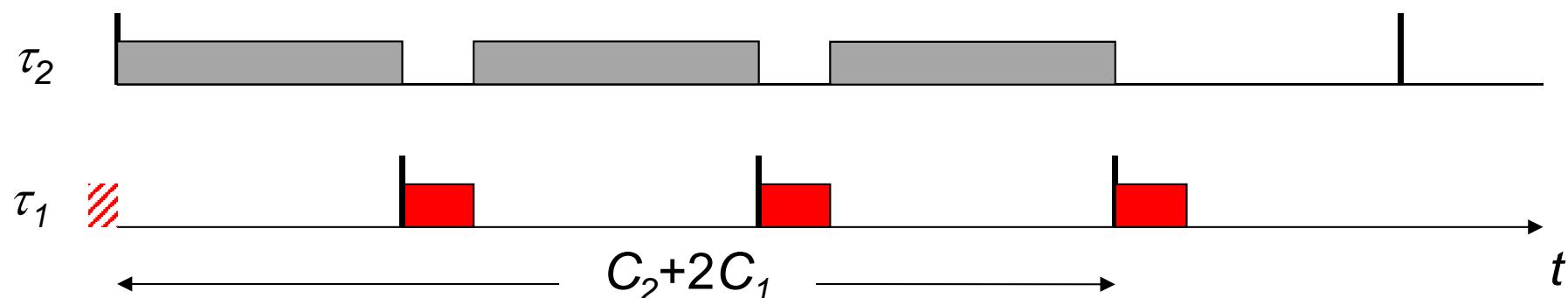
Proof of Critical Instance

Definition: A **critical instant** of a task is the time at which the release of a task will produce the largest response time.

Lemma: For any task, the **critical instant** occurs if that task is simultaneously released with all higher priority tasks.

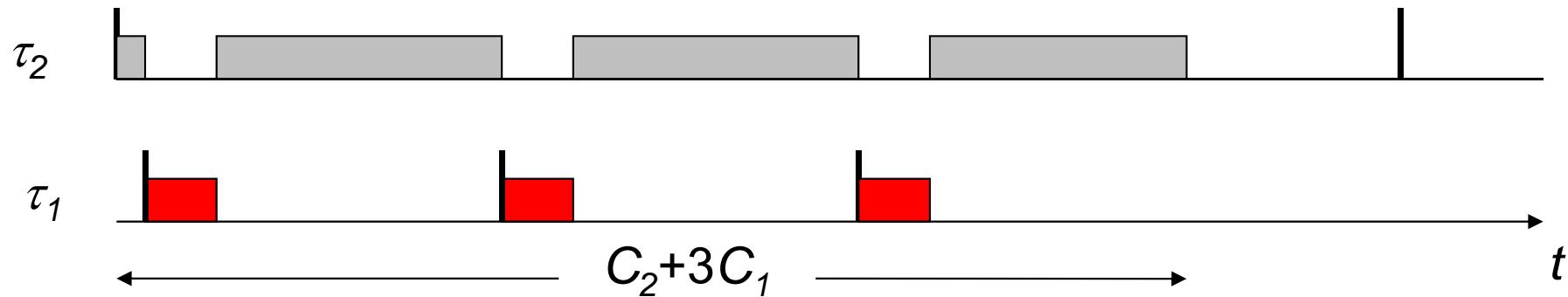
Proof sketch: Start with 2 tasks τ_1 and τ_2 .

Response time of τ_2 is delayed by tasks τ_1 of higher priority:



Proof of Critical Instance

Delay may increase if τ_1 starts earlier:



Maximum delay achieved if τ_2 and τ_1 start simultaneously.

Repeating the argument for all *higher priority* tasks of some task τ_2 :

The worst case response time of a task occurs when it is released simultaneously with all higher-priority tasks.

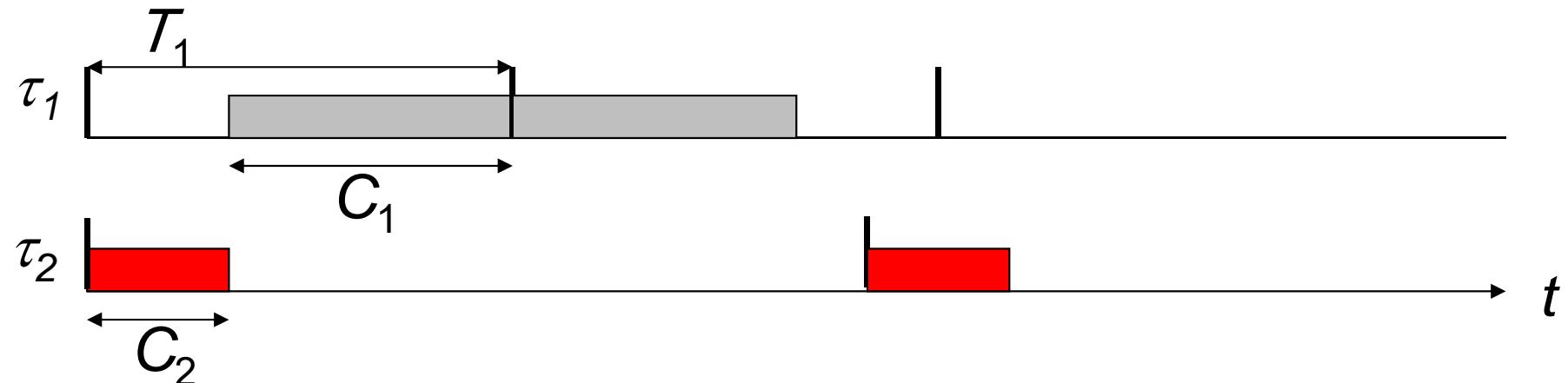
Proof of RM Optimality (2 Tasks)

We have two tasks τ_1, τ_2 with periods $T_1 < T_2$.

Define $F = \lfloor T_2/T_1 \rfloor$: number of periods of τ_1 **fully contained** in T_2

Consider two cases A and B:

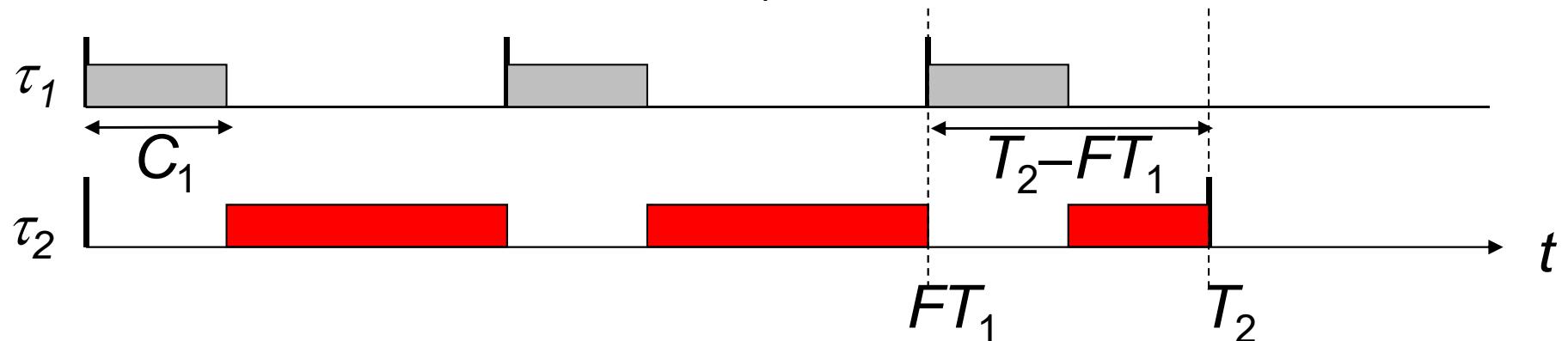
A: Assume RM is **not** used \rightarrow $\text{prio}(\tau_2)$ is highest:



Schedule is feasible if $C_1 + C_2 \leq T_1$ (A)

Proof of RM Optimality (2 Tasks)

B: Assume RM **is** used $\rightarrow \text{prio}(\tau_1)$ is highest:



Schedulable if

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2 \text{ and } C_1 \leq T_1 \quad (\text{B})$$

We need to show that (A) \Rightarrow (B): $C_1 + C_2 \leq T_1 \Rightarrow C_1 \leq T_1$

$$C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1 \Rightarrow$$

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq FT_1 + \min(T_2 - FT_1, C_1) \leq \min(T_2, C_1 + FT_1) \leq T_2$$

Given tasks τ_1 and τ_2 with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

Rate Monotonic Scheduling (RM)

- ▶ **Schedulability analysis:** A set of periodic tasks is schedulable with RM if

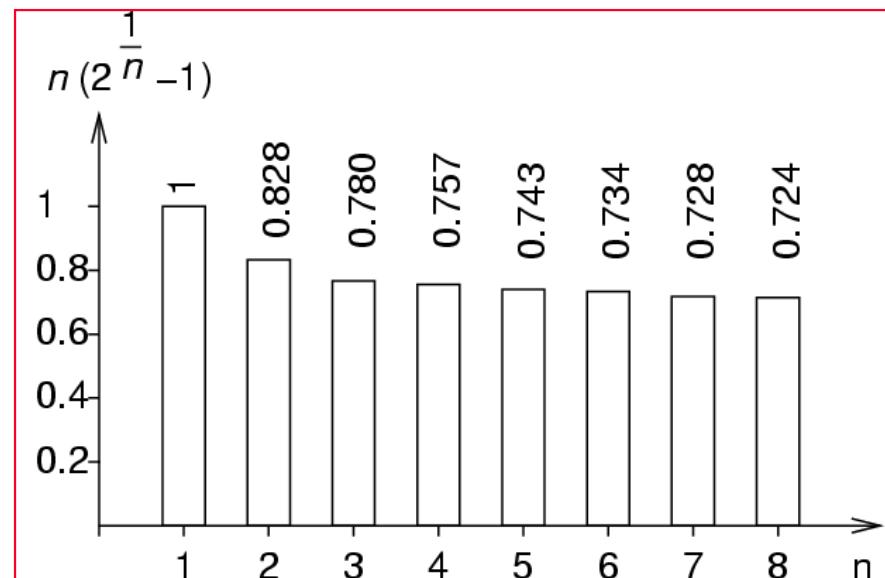
$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

This condition is sufficient but not necessary.

- ▶ The term

$$U = \sum_{i=1}^n \frac{C_i}{T_i}$$

denotes the **processor utilization factor** U which is the fraction of processor time spent in the execution of the task set.



Proof of Utilization Bound (2 Tasks)

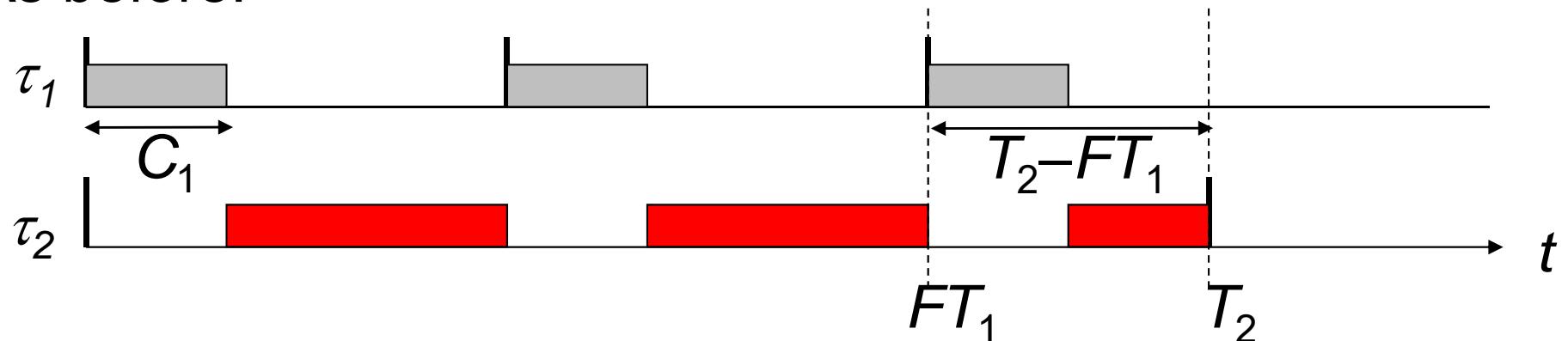
We have two tasks τ_1, τ_2 with periods $T_1 < T_2$.

Define $F = \lfloor T_2/T_1 \rfloor$: number of periods of τ_1 **fully** contained in T_2

- ▶ Proof procedure: Compute upper bound on utilization U such that the task set is still schedulable.
 - assign priorities according to RM;
 - compute upper bound U_{up} by setting computation times to fully utilize processor (C_2 adjusted to fully utilize processor);
 - minimize upper bound with respect to other task parameters.

Proof of Utilization Bound (2 Tasks)

- As before:



Schedulable if

$$FC_1 + C_2 + \min(T_2 - FT_1, C_1) \leq T_2 \text{ and } C_1 \leq T_1$$

- Utilization:

$$\begin{aligned} U &= \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{T_2 - FC_1 - \min\{T_2 - FT_1, C_1\}}{T_2} \\ &= 1 + \frac{C_1(T_2 - FT_1) - T_1 \min\{T_2 - FT_1, C_1\}}{T_1 T_2} \end{aligned}$$

Proof of Utilization Bound (2 Tasks)

- ▶ Minimize utilization bound w.r.t C_1 :
 - If $C_1 \leq T_2 - FT_1$ then U decreases with increasing C_1
 - If $T_2 - FT_1 \leq C_1$ then U decreases with decreasing C_1
 - Therefore, minimum U is obtained with $C_1 = T_2 - FT_1$:

$$\begin{aligned} U &= 1 + \frac{(T_2 - FT_1)^2 - T_1(T_2 - FT_1)}{T_1 T_2} \\ &= 1 + \frac{T_1}{T_2} \left(\left(\frac{T_2}{T_1} - F \right)^2 - \left(\frac{T_2}{T_1} - F \right) \right) \end{aligned}$$

- ▶ We now need to minimize w.r.t. $G = T_2/T_1$ where $F = \lfloor T_2/T_1 \rfloor$ and $T_1 < T_2$. As F is integer, we first suppose that it is independent of $G = T_2/T_1$. We obtain

Proof of Utilization Bound (2 Tasks)

$$U = \frac{T_1}{T_2} \left(\left(\frac{T_2}{T_1} - F \right)^2 + F \right) = \frac{(G-F)^2 + F}{G}$$

Minimizing U with respect to G yields

$$2G(G - F) - (G - F)^2 - F = G^2 - (F^2 + F) = 0$$

If we set $F = 1$, then we obtain

$$G = \frac{T_2}{T_1} = \sqrt{2}$$

$$U = 2(\sqrt{2} - 1)$$

It can easily be checked, that all other integer values for F lead to a larger upper bound on the utilization.

Deadline Monotonic Scheduling (DM)

- ▶ **Assumptions** are as in rate monotonic scheduling, but
 - deadlines may be smaller than the periodic, i.e.

$$C_i \leq D_i \leq T_i$$

- ▶ **Algorithm:** Each task is assigned a priority. Tasks with smaller relative deadlines will have higher priorities. Tasks with higher priority interrupt tasks with lower priority.

- ▶ **Schedulability analysis:** A set of periodic tasks is schedulable with DM if

$$\sum_{i=1}^n \frac{C_i}{D_i} \leq n(2^{1/n} - 1)$$

This condition is sufficient but not necessary (in general).

Deadline Monotonic Scheduling (DM)

- ▶ There is also a **necessary and sufficient schedulability test** which is computationally more involved. It is based on the following observations:
 - The worst-case processor demand occurs when all tasks are released simultaneously; that is, at their **critical instances**.
 - For each task i , the sum of its processing time and the interference (preemption) imposed by higher priority tasks must be less than or equal to D_i .
 - A measure of the **worst case interference** for task i can be computed as the sum of the processing times of all higher priority tasks released before some time t where tasks are ordered according to $m < n \Leftrightarrow D_m < D_n$:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j$$

Deadline Monotonic Scheduling (DM)

- ▶ The **longest response time** R_i of a periodic task i is computed, at the critical instant, as the sum of its computation time and the interference due to preemption by higher priority tasks

$$R_i = C_i + I_i$$

- ▶ Hence, the **schedulability test** needs to compute the smallest R_i that satisfies

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

for all tasks i . Then, $R_i \leq D_i$ must hold for all tasks i .

- ▶ It can be shown that this condition is necessary and sufficient.

Deadline Monotonic Scheduling (DM)

- ▶ The **longest response times** R_i of the periodic tasks i can be computed iteratively by the following algorithm:

```
Algorithm: DM_guarantee ( $\Gamma$ )
{
    for (each  $\tau_i \in \Gamma$ ){
        I = 0;
        do {
            R = I + Ci;
            if (R > Di) return(UNSCHEDULABLE);
            I =  $\sum_{j=1, \dots, (i-1)} \lceil R/T_j \rceil C_j;$ 
        } while (I + Ci > R);
    }
    return(SCHEDULABLE);
}
```

DM Example

► ***Example:***

- Task 1: $C_1 = 1; T_1 = 4; D_1 = 3$
- Task 2: $C_2 = 1; T_2 = 5; D_2 = 4$
- Task 3: $C_3 = 2; T_3 = 6; D_3 = 5$
- Task 4: $C_4 = 1; T_4 = 11; D_4 = 10$

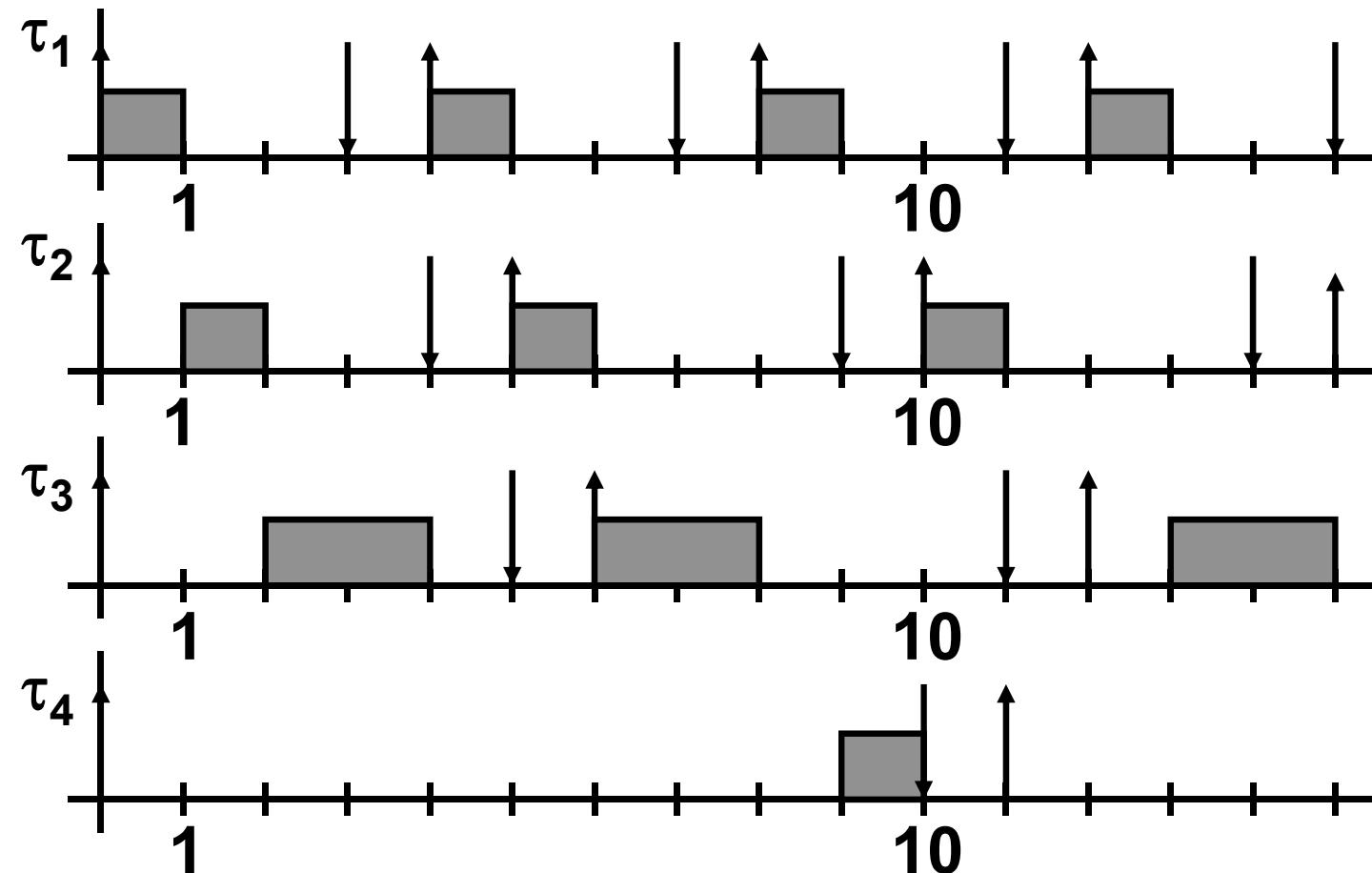
► ***Algorithm*** for task 4:

- Step 0: $R_4 = 1$
- Step 1: $R_4 = 5$
- Step 2: $R_4 = 6$
- Step 3: $R_4 = 7$
- Step 4: $R_4 = 9$
- Step 5: $R_4 = 10$

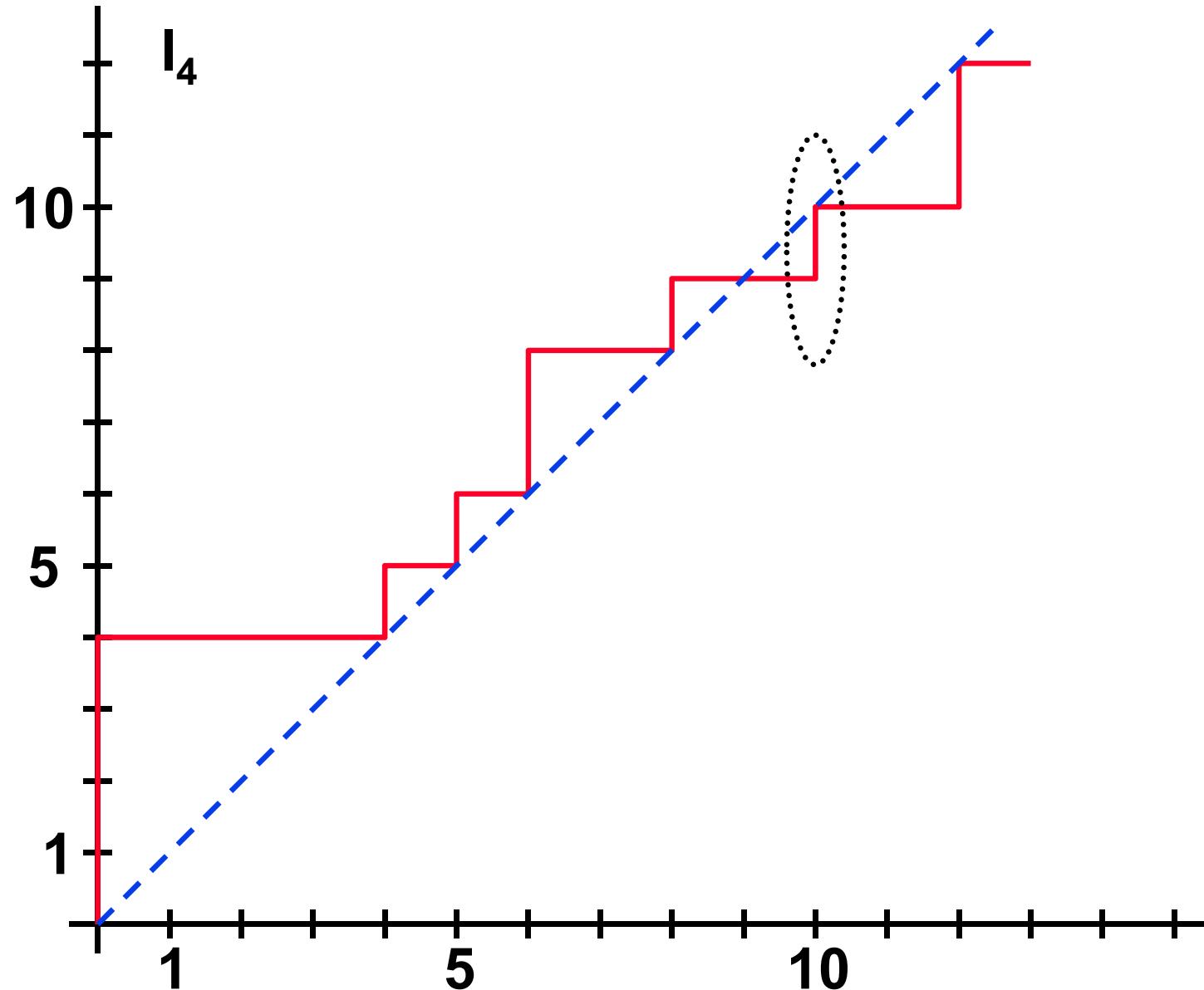
DM Example

$$U = 0.874$$

$$\sum_{i=1}^n \frac{C_i}{D_i} = 1.08 > n(2^{1/n} - 1) = 0.757$$



DM Example



EDF Scheduling (earliest deadline first)

- ▶ **Assumptions:**
 - dynamic priority assignment
 - intrinsically preemptive
 - $D_i \leq T_i$
- ▶ **Algorithm:** The currently executing task is preempted whenever another periodic instance with earlier deadline becomes active.
$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$
- ▶ **Optimality:** No other algorithm can schedule a set of periodic tasks if the set that can not be scheduled by EDF.
- ▶ The **proof** is simple and follows that of the aperiodic case.

EDF Scheduling

- ▶ A necessary and sufficient **schedulability test** if $D_i = T_i$:
 - A set of periodic tasks is schedulable with EDF if and only if

$$\sum_{i=1}^n \frac{C_i}{T_i} = U \leq 1$$

- ▶ The term

$$U = \sum_{i=1}^n \frac{C_i}{T_i}$$

denotes the **average processor utilization**.

EDF Scheduling

- ▶ If the utilization satisfies $U > 1$, then there is no valid schedule: The total demand of computation time in interval $T = T_1 \cdot T_2 \cdot \dots \cdot T_n$ is
$$\sum_{i=1}^n \frac{C_i}{T_i} T = UT > T$$
and therefore, it exceeds the available processor time.
- ▶ If the utilization satisfies $U \leq 1$, then there is a valid schedule.
 - We will proof this by contradiction: Assume that deadline is missed at some time t_2 . Then we will show that the utilization was larger than 1.

EDF Scheduling

- ▶ If the deadline was missed at t_2 then define t_1 as the maximal time before t_2 where
 - the processor is continuously busy in $[t_1, t_2]$ and
 - the processor only executes tasks that have their arrival time AND deadline in $[t_1, t_2]$.
- ▶ Why does such a time t_1 exist?
 - We find such a t_1 by starting at t_2 and going backwards in time, always ensuring that the processor only executed tasks that have their deadline before or at t_2 :
 - Because of EDF, the processor will be busy shortly before t_2 and it executes on the task that has deadline at t_2 .
 - Suppose that we reach a time when the processor gets idle, then we found t_1 : There is a task arrival at t_1 and the task queue is empty shortly before.
 - Suppose that we reach a time such that shortly before the processor works on a task with deadline after t_2 , then we also found t_1 : Because of EDF, all tasks the processor processed in $[t_1, t_2]$ arrived at or after t_1 (otherwise, the processor would not have operated before t_1 on a task with deadline after t_2).

EDF Scheduling

- Within the interval $[t_1, t_2]$ the total computation time demanded by the periodic tasks is bounded by

$$C_p(t_1, t_2) = \sum_{i=1}^n \left\lfloor \frac{t_2 - t_1}{T_i} \right\rfloor C_i \leq \sum_{i=1}^n \frac{t_2 - t_1}{T_i} C_i = (t_2 - t_1)U$$

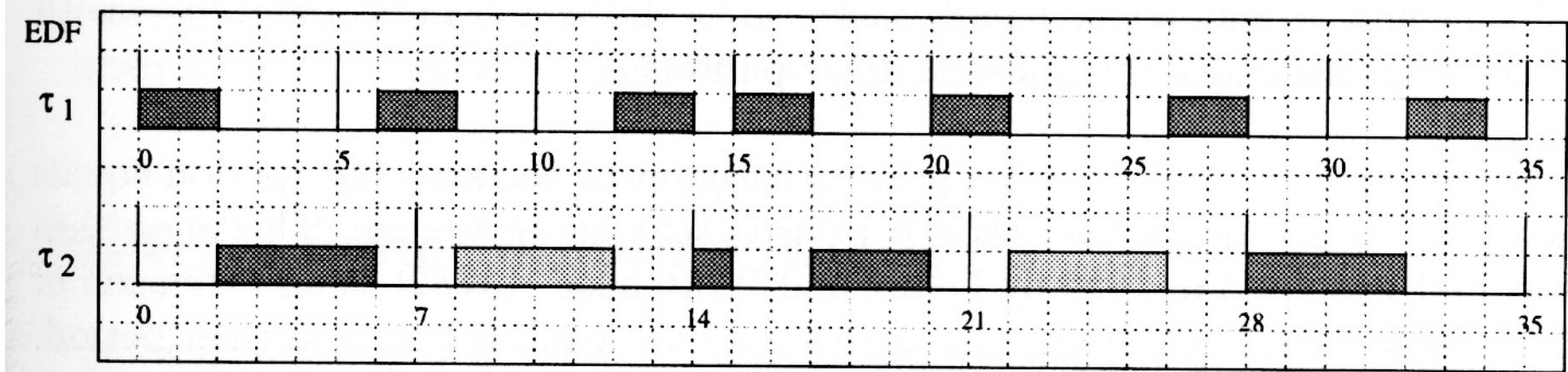
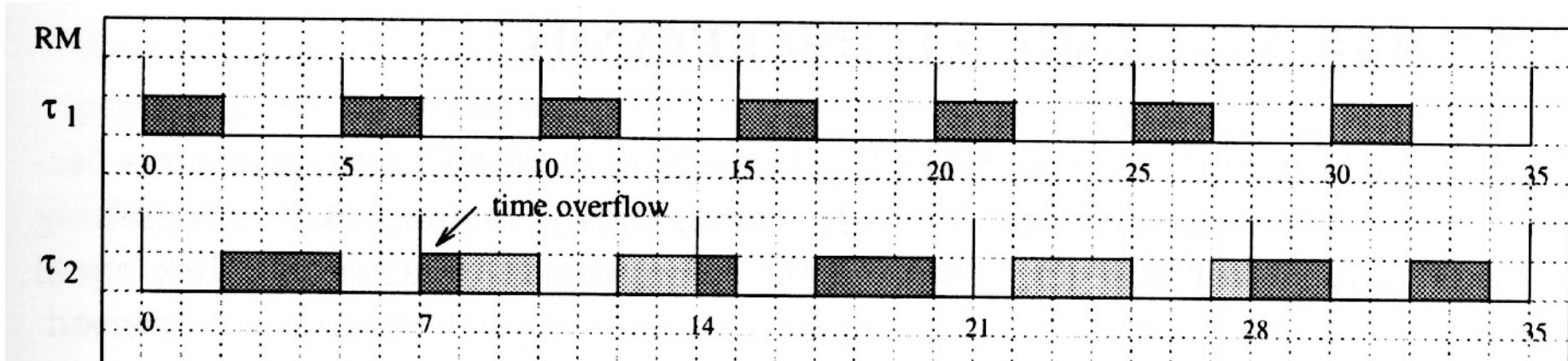
number of complete periods
of task I in the interval

- Since the deadline at time t_2 is missed, we must have:

$$t_2 - t_1 < C_p(t_1, t_2) \leq (t_2 - t_1)U \Rightarrow U > 1$$

Periodic Tasks

- ▶ **Example:** 2 tasks, deadline = periods, $U = 97\%$

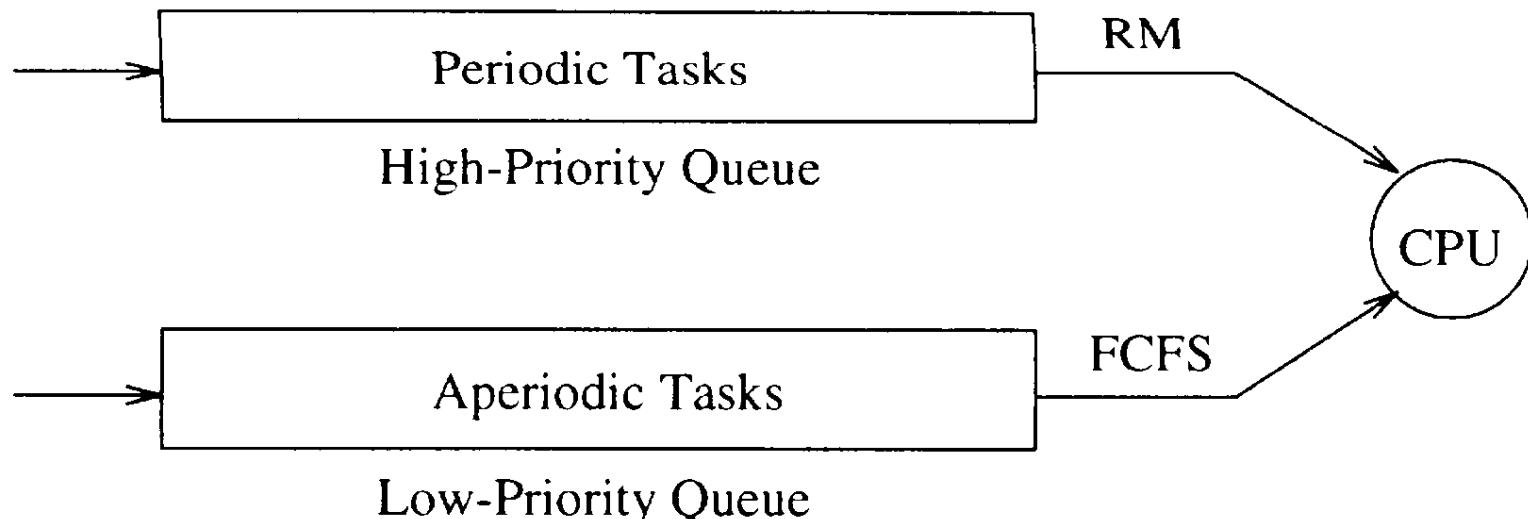


Problem of Mixed Task Sets

- ▶ In many applications, there are as well aperiodic as periodic tasks.
- ▶ **Periodic tasks**: time-driven, execute critical control activities with hard timing constraints aimed at guaranteeing regular activation rates.
- ▶ **Aperiodic tasks**: event-driven, may have hard, soft, non-real-time requirements depending on the specific application.
- ▶ **Sporadic tasks**: Offline guarantee of event-driven aperiodic tasks with critical timing constraints can be done only by making proper assumptions on the environment; that is by assuming a maximum arrival rate for each critical event. Aperiodic tasks characterized by a minimum interarrival time are called **sporadic**.

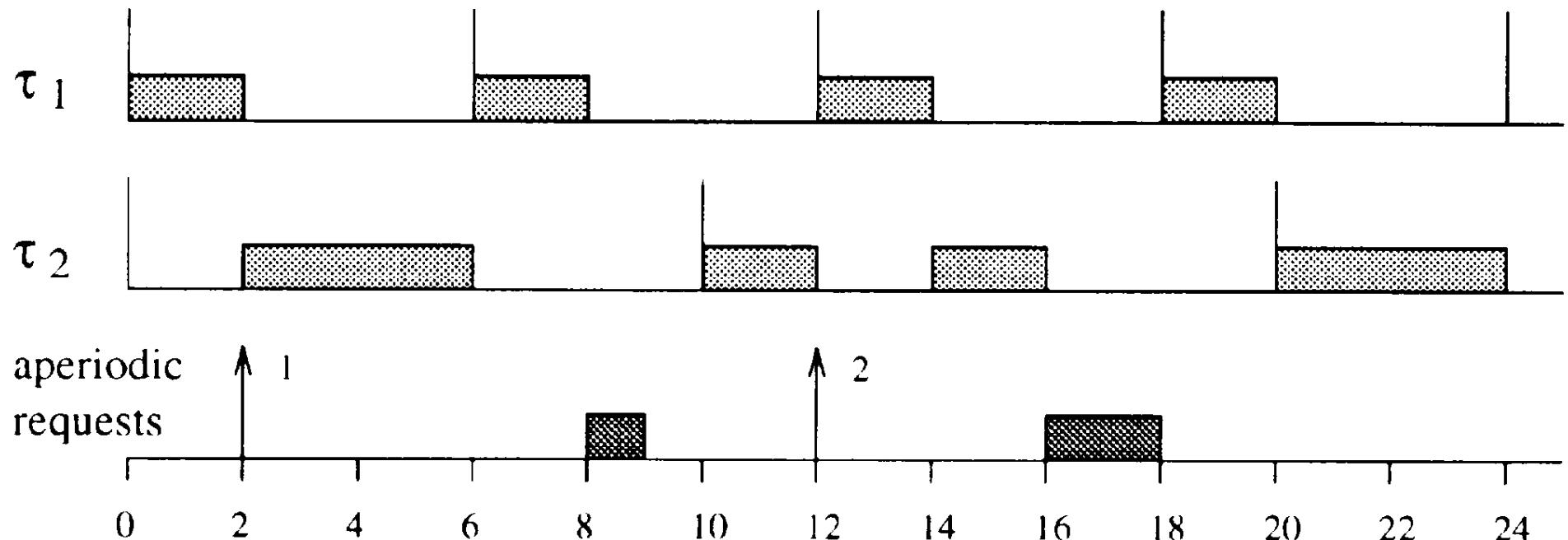
Background Scheduling

- ▶ ***Simple solution*** for RM and EDF scheduling of periodic tasks:
 - Processing of aperiodic tasks in the background, i.e. if there are no periodic request.
 - Periodic tasks are not affected.
 - Response of aperiodic tasks may be prohibitively long and there is no possibility to assign a higher priority to them.



Background Scheduling

- ▶ **Example** (rate monotonic periodic schedule):



RM - Polling Server

- ▶ **Idea:** Introduce an *artificial periodic task* whose purpose is to service aperiodic requests as soon as possible (therefore, “server”).
 - Like any periodic task, a server is characterized by a period T_s and a computation time C_s .
 - The server is scheduled with the same algorithm used for the periodic tasks and, once active, it serves the aperiodic requests within the limit of its server capacity.
 - Its priority (period!) can be chosen to match the response time requirement for the aperiodic tasks.

RM - Polling Server

- ▶ ***Function of polling server (PS)***

- At regular intervals equal to T_s , a PS task is instantiated. When it has the highest current priority, it serves any pending aperiodic requests within the limit of its capacity C_s .
- If no aperiodic requests are pending, PS suspends itself until the beginning of the next period and the time originally allocated for aperiodic service is ***not preserved for aperiodic execution.***

- ▶ ***Disadvantage:*** If an aperiodic request arrives just after the server has suspended, it must wait until the beginning of the next polling period.

RM - Polling Server

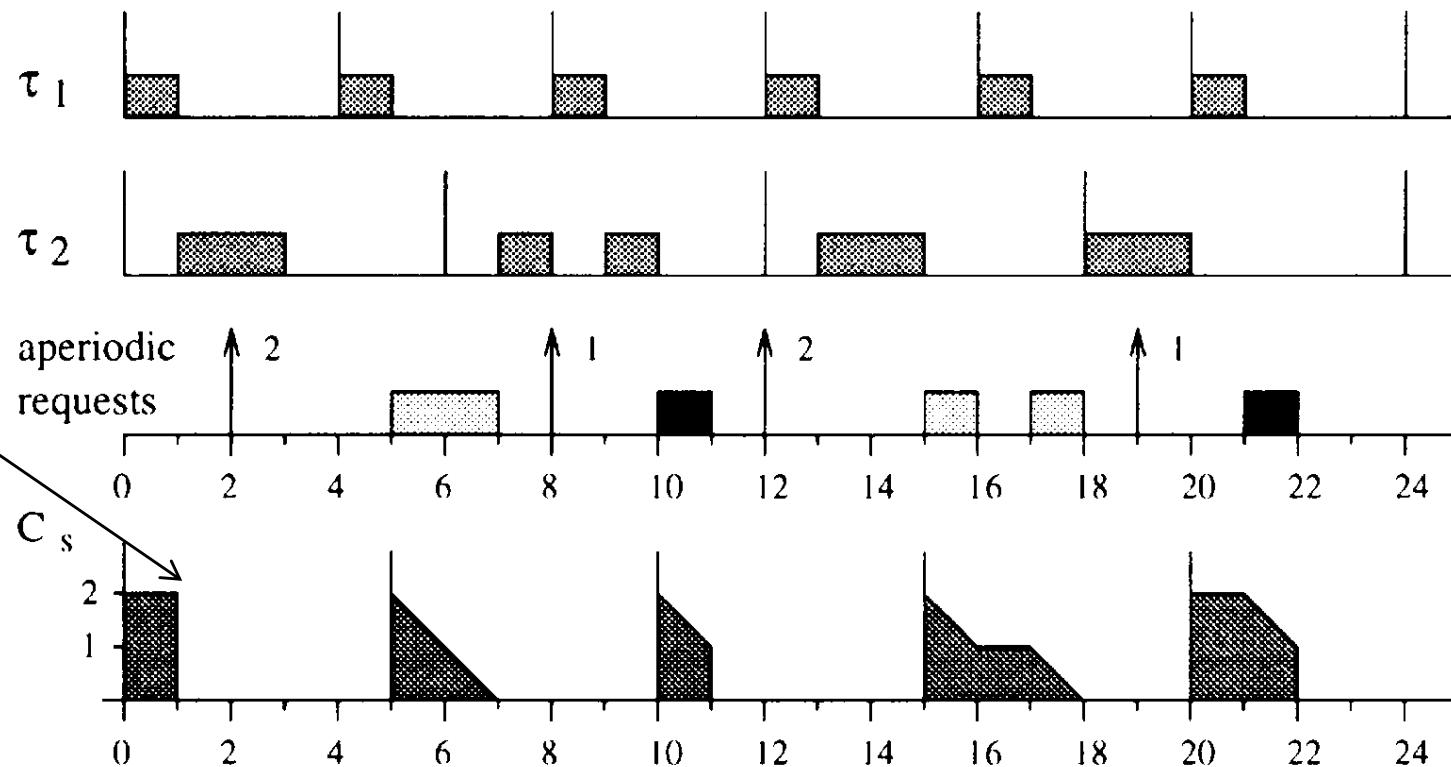
► Example

	C_i	T_i
τ_1	1	4
τ_2	2	6

Server

$$\begin{aligned}C_s &= 2 \\T_s &= 5\end{aligned}$$

server has current highest priority and checks the queue of tasks



RM - Polling Server

- ▶ **Schedulability analysis** of periodic tasks
 - As in the case of RM as the interference by a server task is the same as the one introduced by an equivalent periodic task.
 - A set of periodic tasks and a server task can be executed within their deadlines if

$$\frac{C_s}{T_s} + \sum_{i=1}^n \frac{C_i}{T_i} \leq (n+1) \left(2^{1/(n+1)} - 1 \right)$$

- Again, this test is sufficient but not necessary.

RM - Polling Server

- ▶ **Aperiodic guarantee** of aperiodic activities.
- ▶ **Assumption:** An aperiodic task is finished before a new aperiodic request arrives.
 - Computation time C_a , deadline D_a .
 - **Sufficient schedulability test:**

$$(1 + \left\lceil \frac{C_a}{C_s} \right\rceil)T_s \leq D_a$$

The aperiodic task arrives shortly after the activation of the server task.

Maximal number of necessary server periods.

If the server task has the highest priority there is a necessary test also.

EDF – Total Bandwidth Server

► *Total Bandwidth Server:*

- When the k th aperiodic request arrives at time $t = r_k$, it receives a deadline

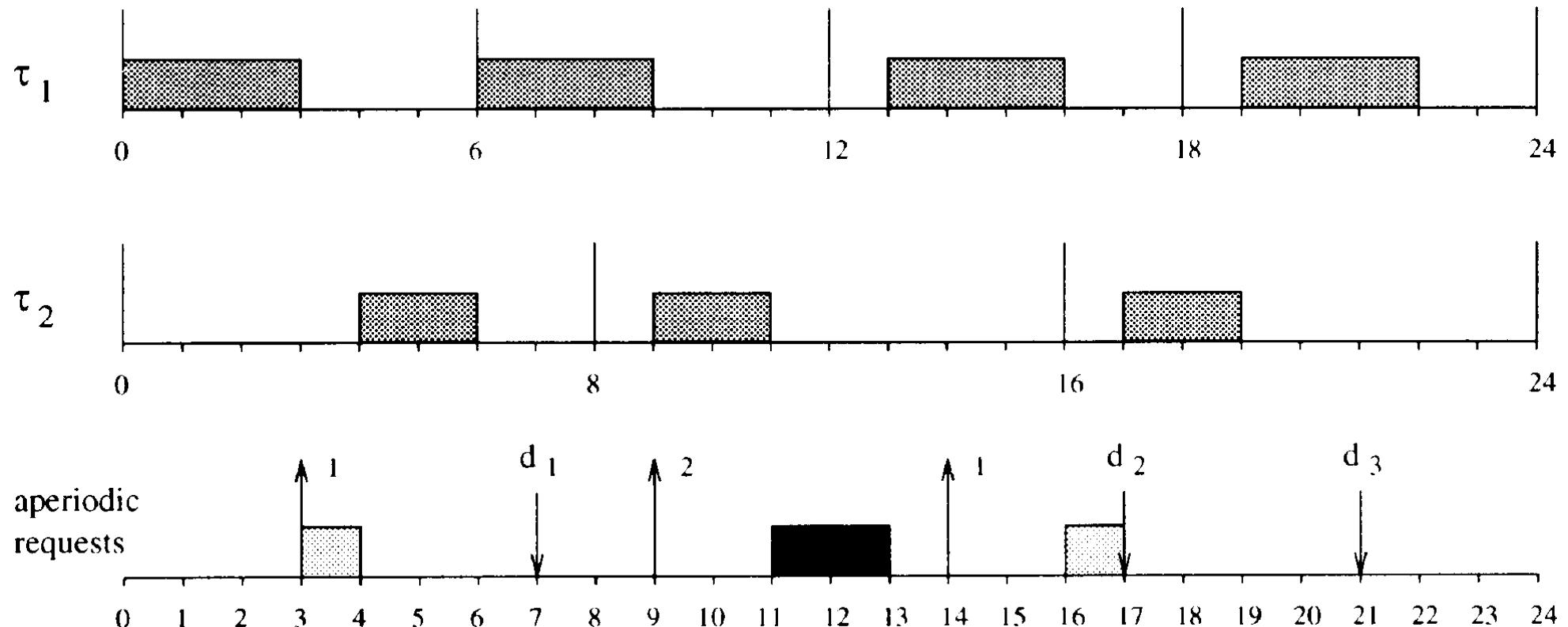
$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

where C_k is the execution time of the request and U_s is the server utilization factor (that is, its bandwidth). By definition, $d_0=0$.

- Once a deadline is assigned, the request is inserted into the ready queue of the system as any other periodic instance.

EDF – Total Bandwidth Server

- ▶ **Example:** $U_p = 0.75$, $U_s = 0.25$, $U_p + U_s = 1$



EDF – Total Bandwidth Server

- ▶ **Schedulability test:**

- Given a set of n periodic tasks with processor utilization U_p and a total bandwidth server with utilization U_s , the whole set is schedulable by EDF if and only if

$$U_p + U_s \leq 1$$

- ▶ **Proof:**

- In each interval of time $[t_1, t_2]$, if C_{ape} is the total execution time demanded by aperiodic requests arrived at t_1 or later and served with deadlines less or equal to t_2 , then

$$C_{ape} \leq (t_2 - t_1)U_s$$

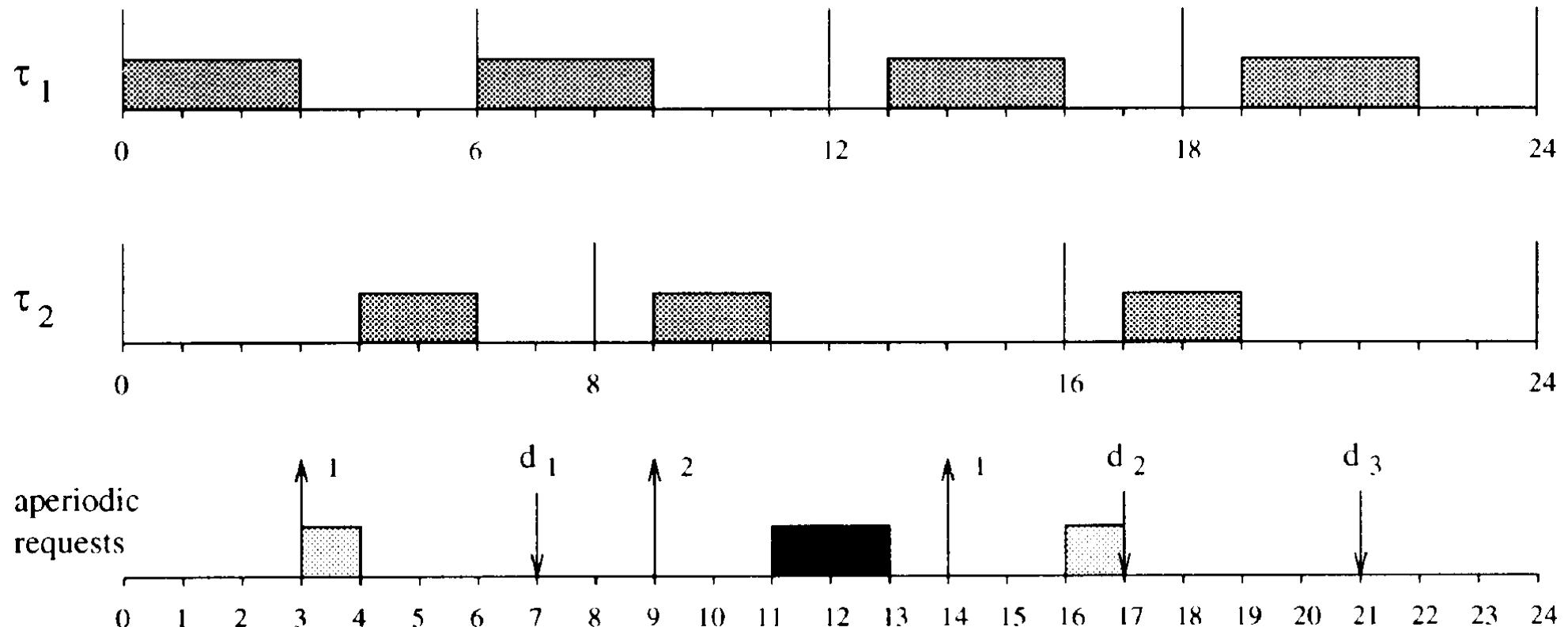
EDF – Total Bandwidth Server

- ▶ If this has been proven, the proof of the schedulability test follows closely that of the periodic case.
- ▶ Proof of lemma:

$$\begin{aligned} C_{ape} &= \sum_{k=k_1}^{k_2} C_k \\ &= U_s \sum_{k=k_1}^{k_2} (d_k - \max(r_k, d_{k-1})) \\ &\leq U_s (d_{k_2} - \max(r_{k_1}, d_{k_1-1})) \\ &\leq U_s (t_2 - t_1) \end{aligned}$$

EDF – Total Bandwidth Server

- ▶ **Example:** $U_p = 0.75$, $U_s = 0.25$, $U_p + U_s = 1$

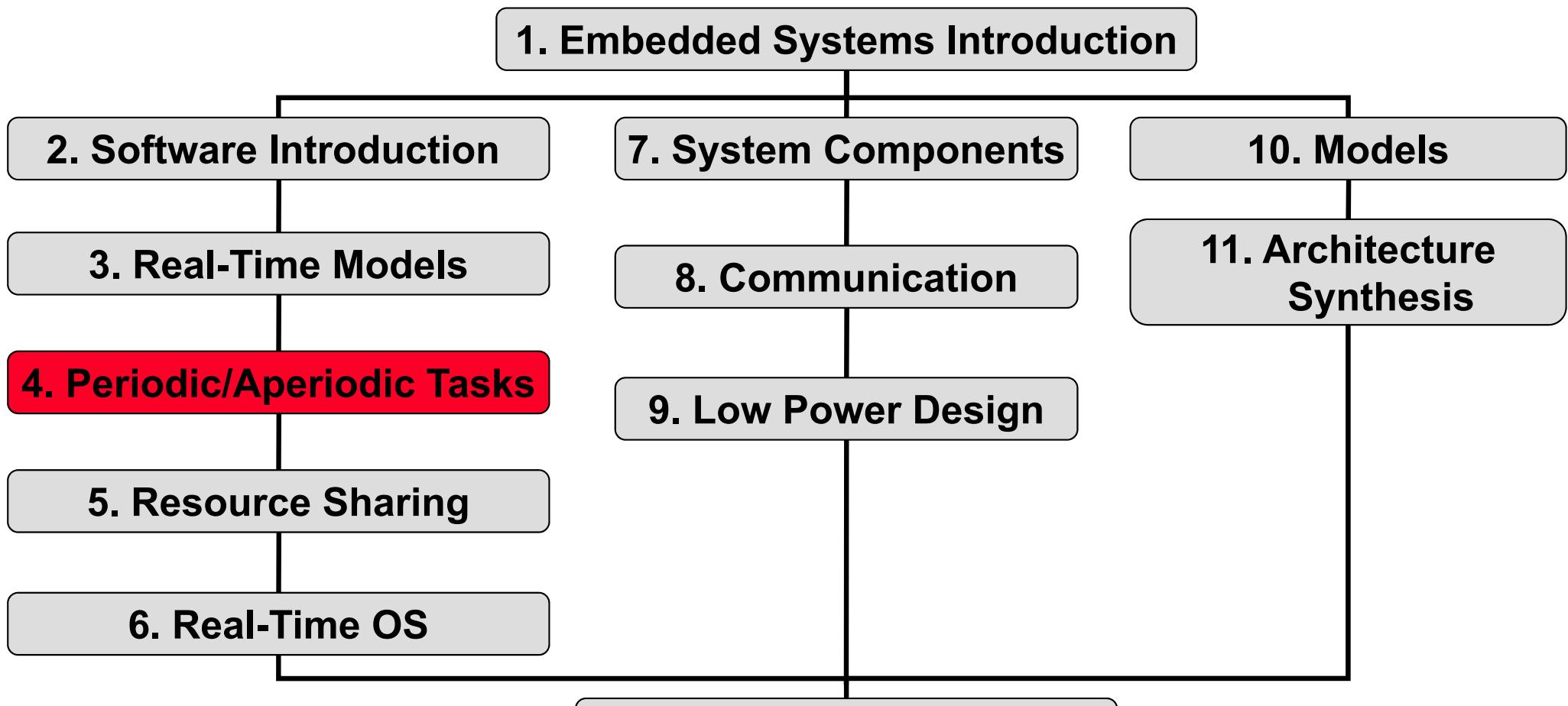


Embedded Systems

4a. Example Network Processor

Lothar Thiele

Contents of Course



*Software and
Programming*

*Processing and
Communication*

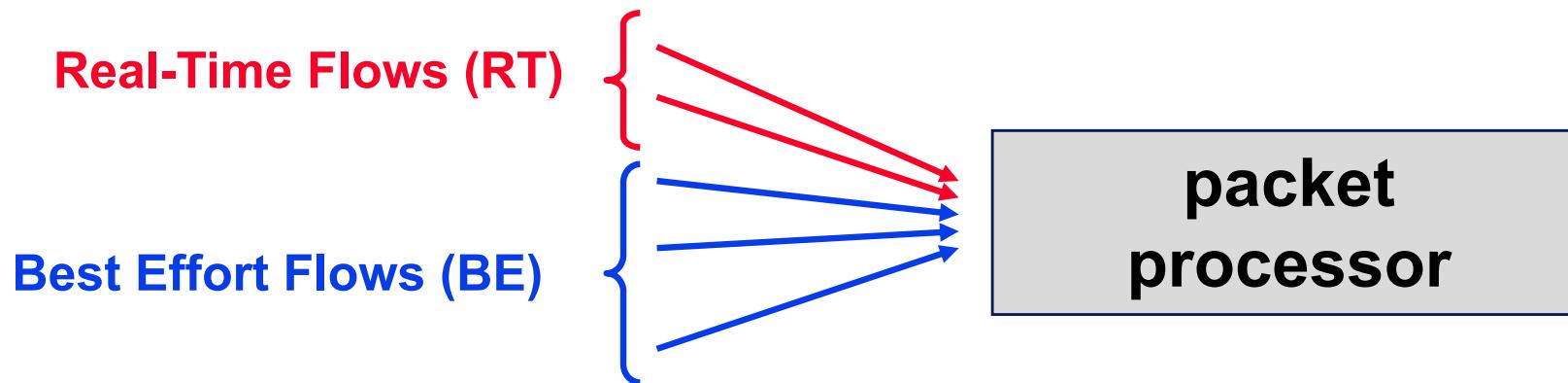
Hardware

Software-Based NP

**Network Processor:
Programmable Processor Optimized to
Perform Packet Processing**

- ▶ How to Schedule the CPU cycles meaningfully?
 - Differentiating the level of service given to different flows
 - Each flow being processed by a different processing function

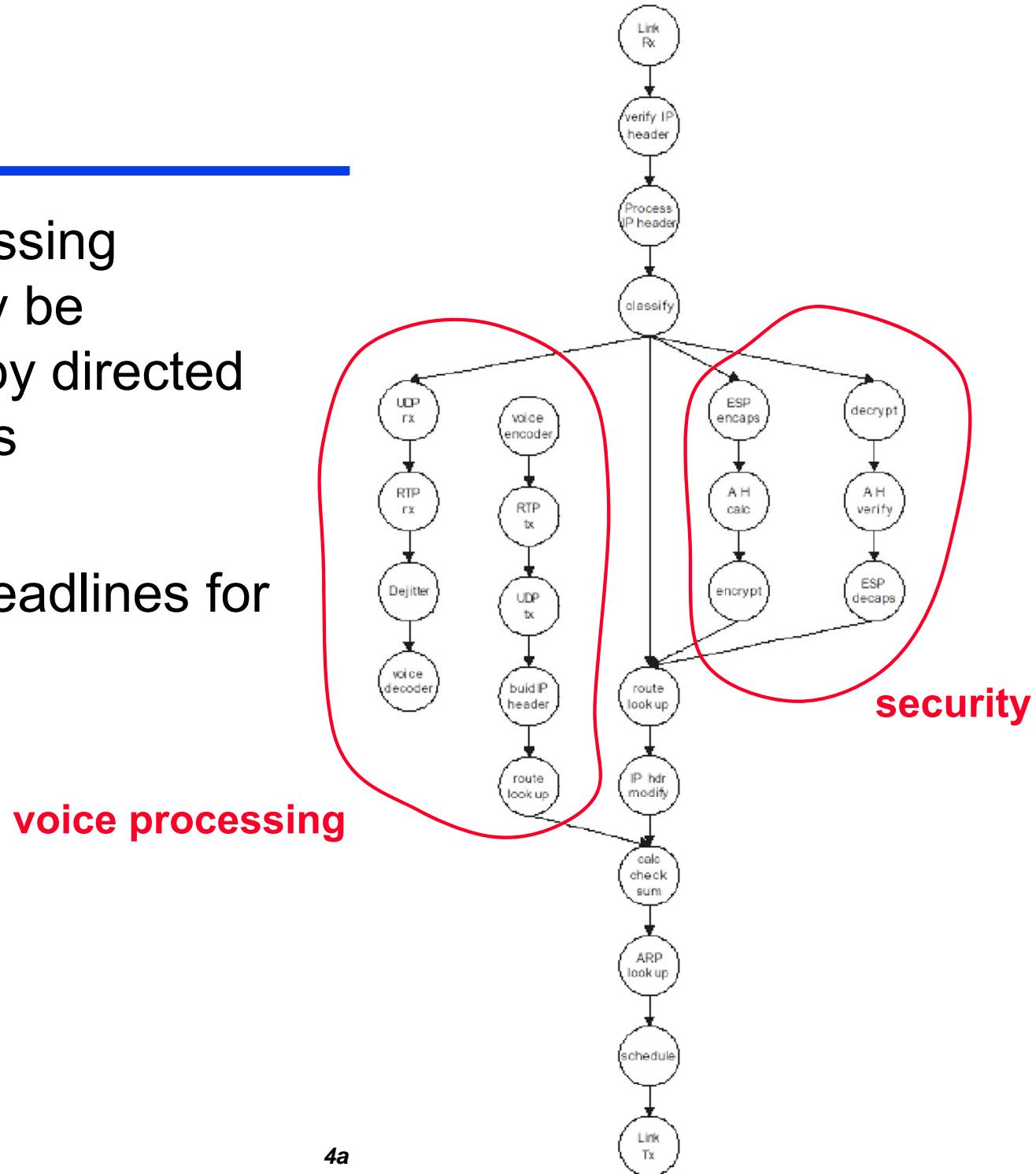
Our Model – Simple NP



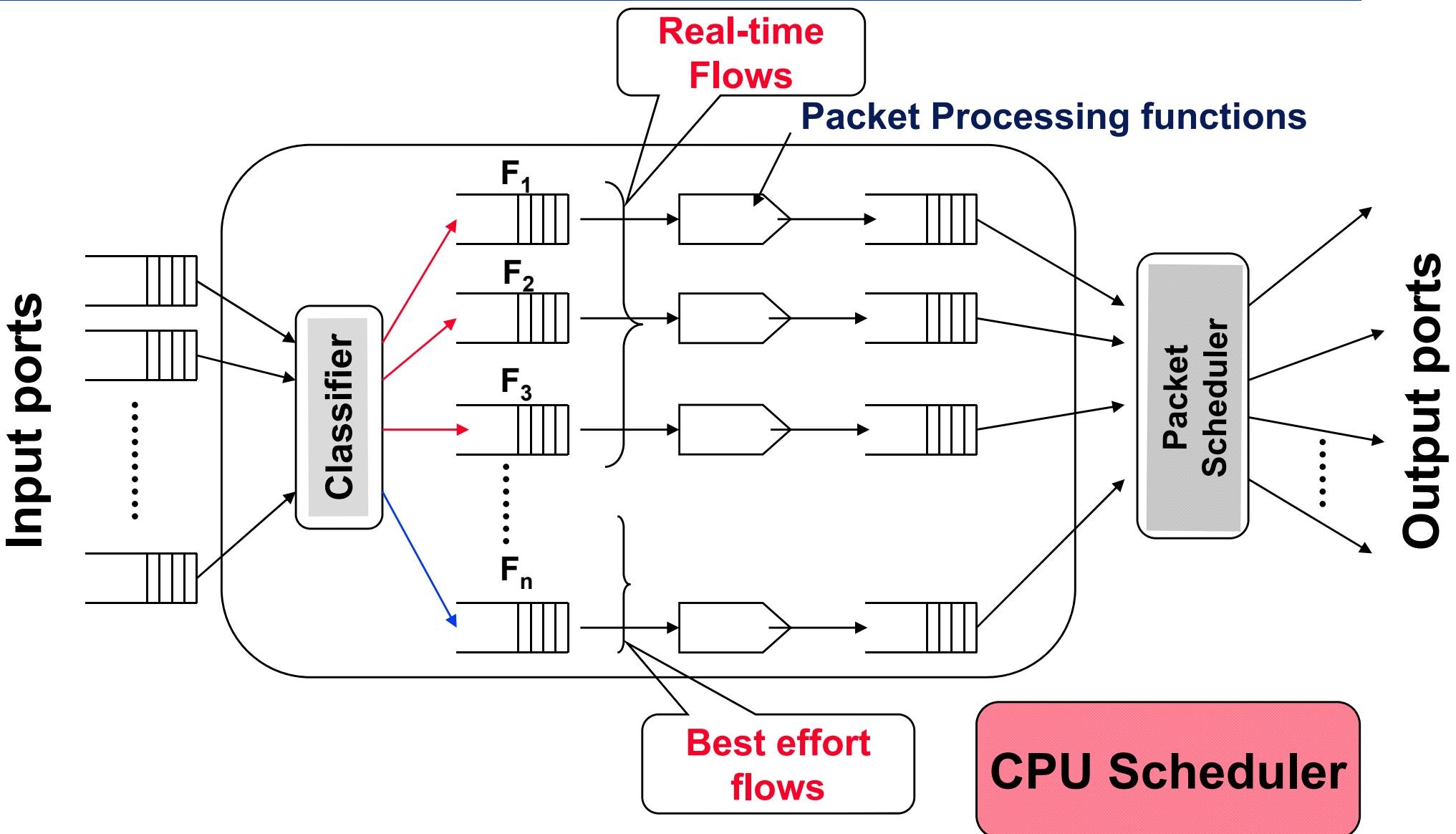
- ▶ Real-time flows have deadlines which must be met
- ▶ Best effort flows may have several QoS classes and should be served to achieve maximum throughput

Task Model

- ▶ Packet processing functions may be represented by directed acyclic graphs
- ▶ End-to-end deadlines for RT packets



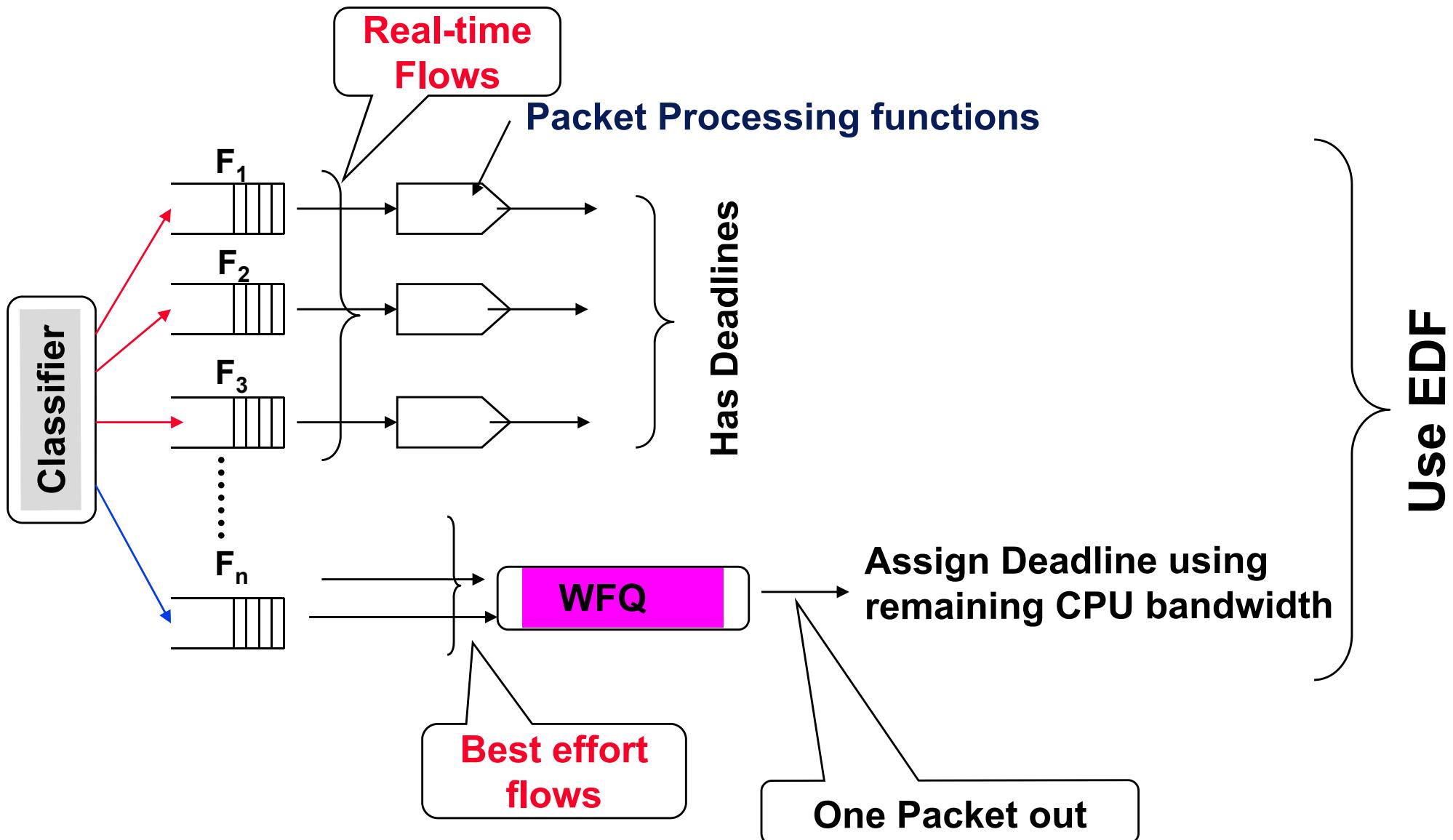
Architecture



CPU Scheduling

- ▶ First Schedule RT, then BE (background scheduling)
 - Overly pessimistic
- ▶ Use ***EDF Total Bandwidth Server***
 - EDF for Real-Time tasks
 - Use the remaining bandwidth to serve Best Effort Traffic
 - WFQ (weighted fair queuing) to determine which best effort flow to serve; not discussed here ...

CPU Scheduling



CPU Scheduling

- ▶ As discussed, the **basis is the TBS**:

$$d_k = \max\{r_k, d_{k-1}\} + c_k/U_s$$

computation demand of best effort packet

deadline of best effort packet arrival of best effort packet utilization by real-time flows

The diagram illustrates the formula for calculating the deadline of a best-effort packet. The formula is enclosed in a light gray box:

$$d_k = \max\{r_k, d_{k-1}\} + c_k/U_s$$

Four red arrows point to specific components of the formula:

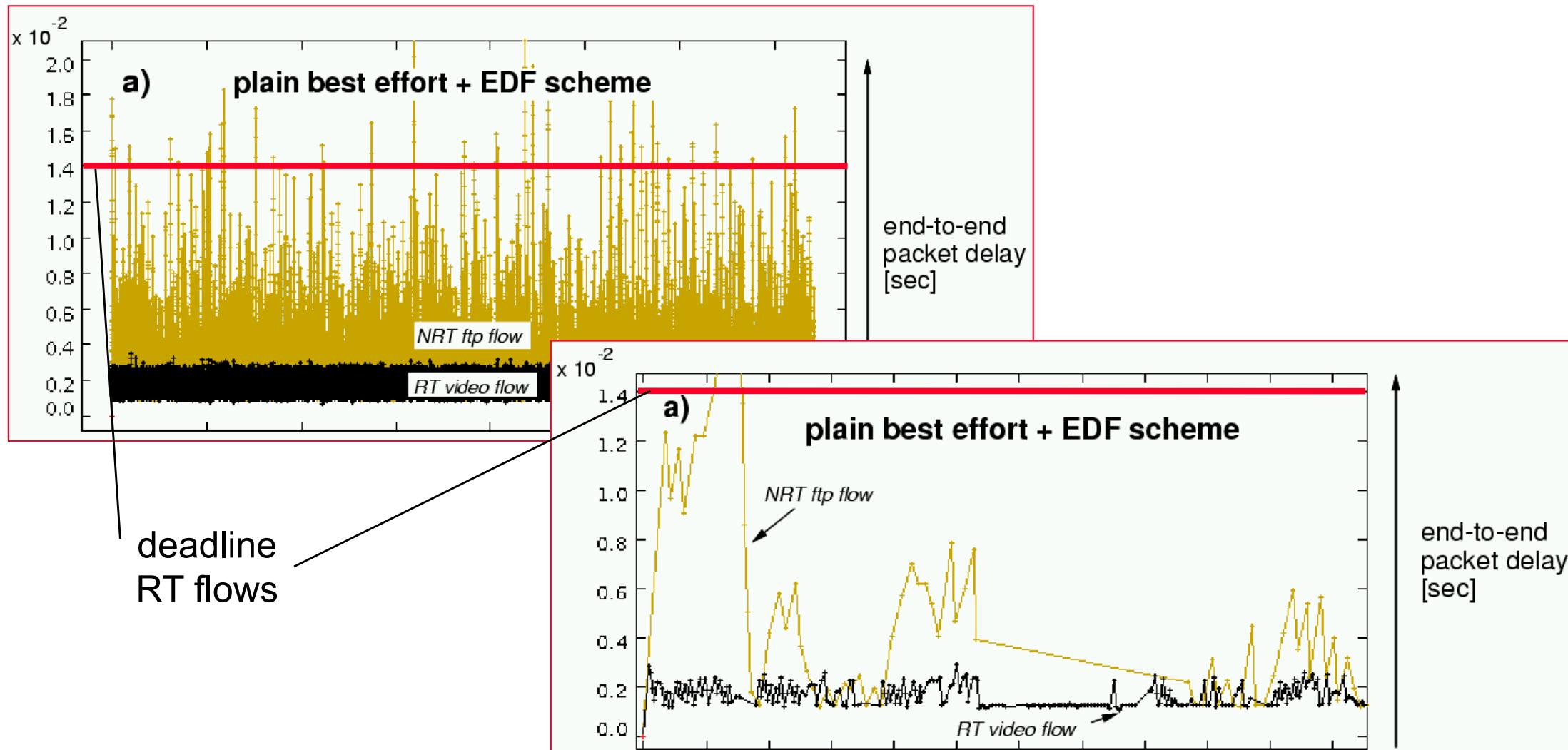
- An arrow points to r_k from the left.
- An arrow points to d_{k-1} from the bottom.
- An arrow points to c_k/U_s from the top.
- An arrow points to U_s from the right.

Labels above the formula identify its purpose: "computation demand of best effort packet". Labels below the formula identify the components: "deadline of best effort packet", "arrival of best effort packet", and "utilization by real-time flows".

- ▶ **But**: utilization depends on time (packet streams) !
 - Just taking upper bound is too pessimistic
 - Solution with time dependent utilization is (much) more complex – BUT IT HELPS ...

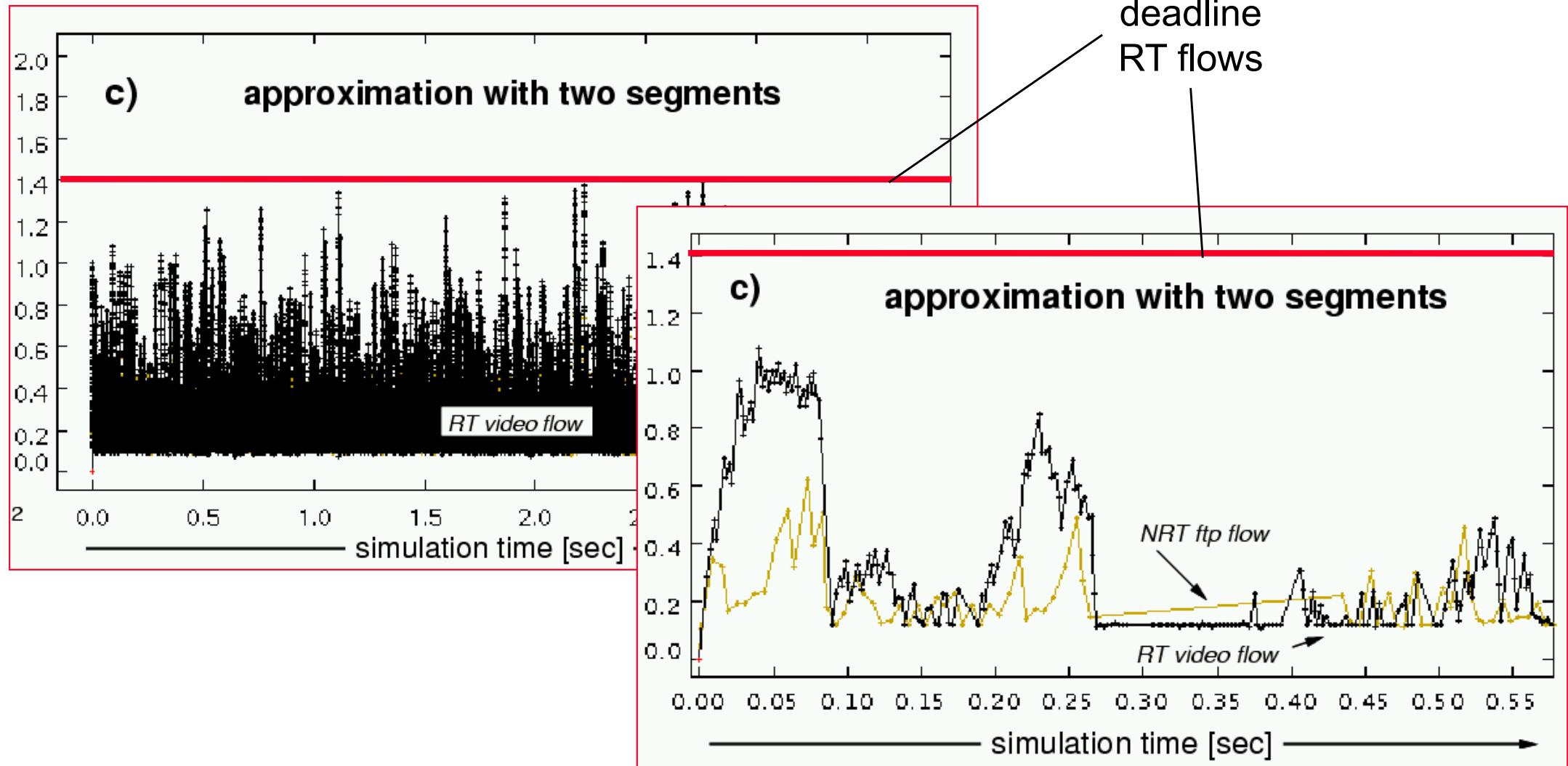
CPU Scheduling

► Before



CPU Scheduling

► After

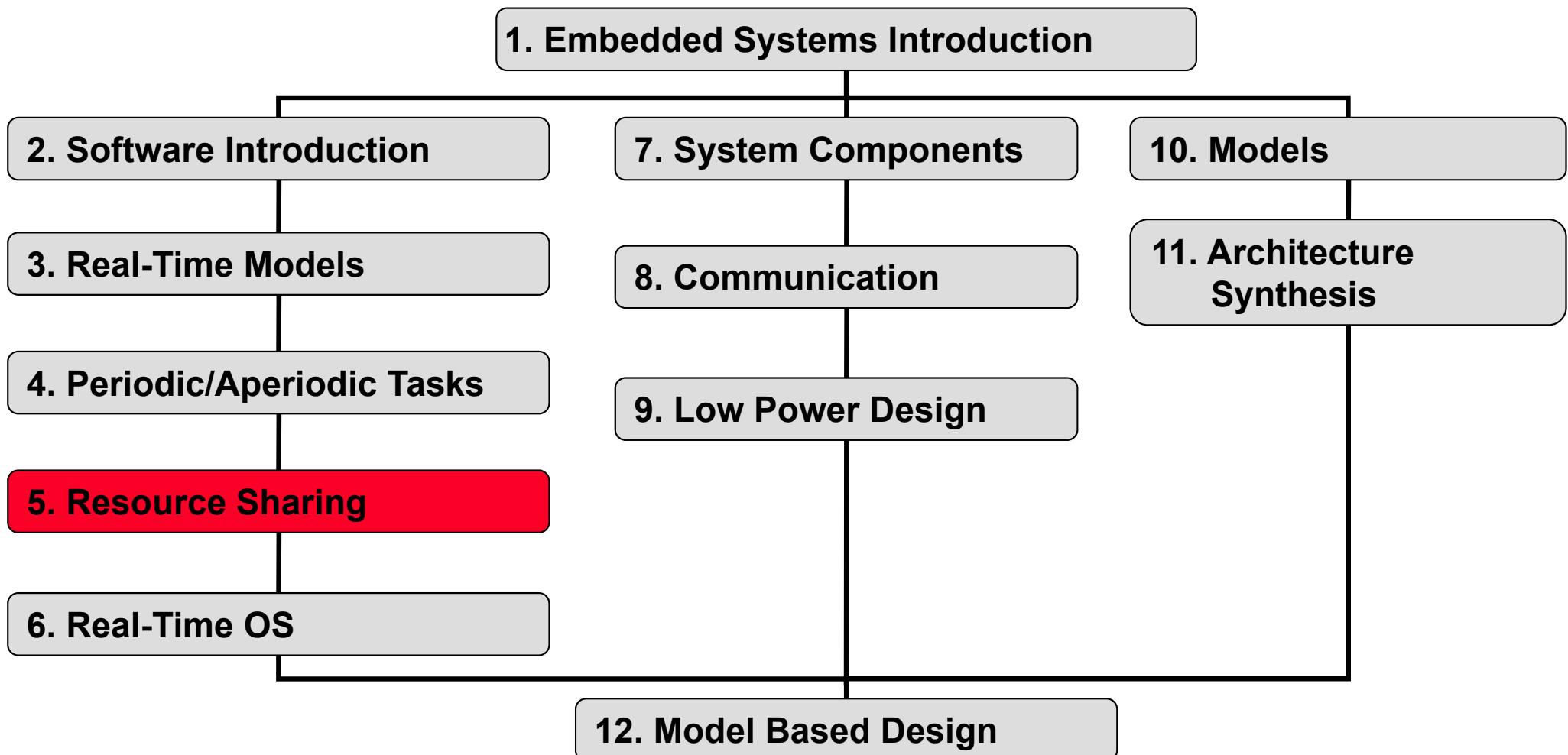


Embedded Systems

5. Resource Sharing

Lothar Thiele

Contents of Course



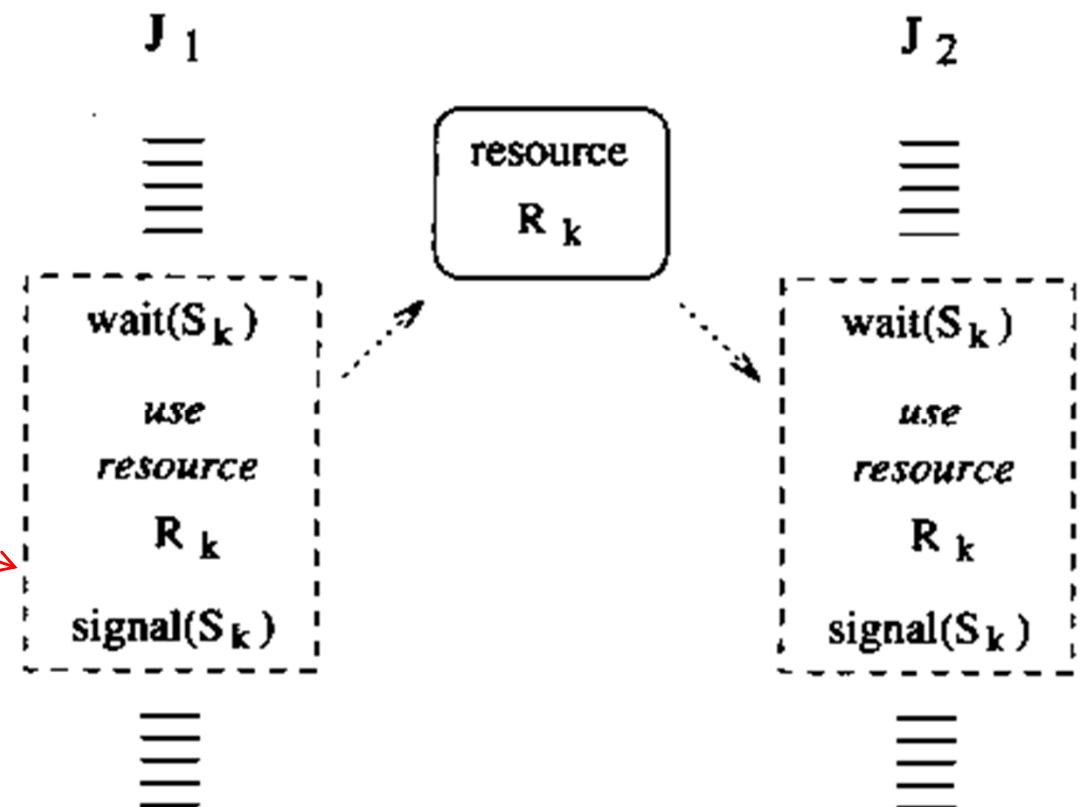
*Software and
Programming*

*Processing and
Communication*

Hardware

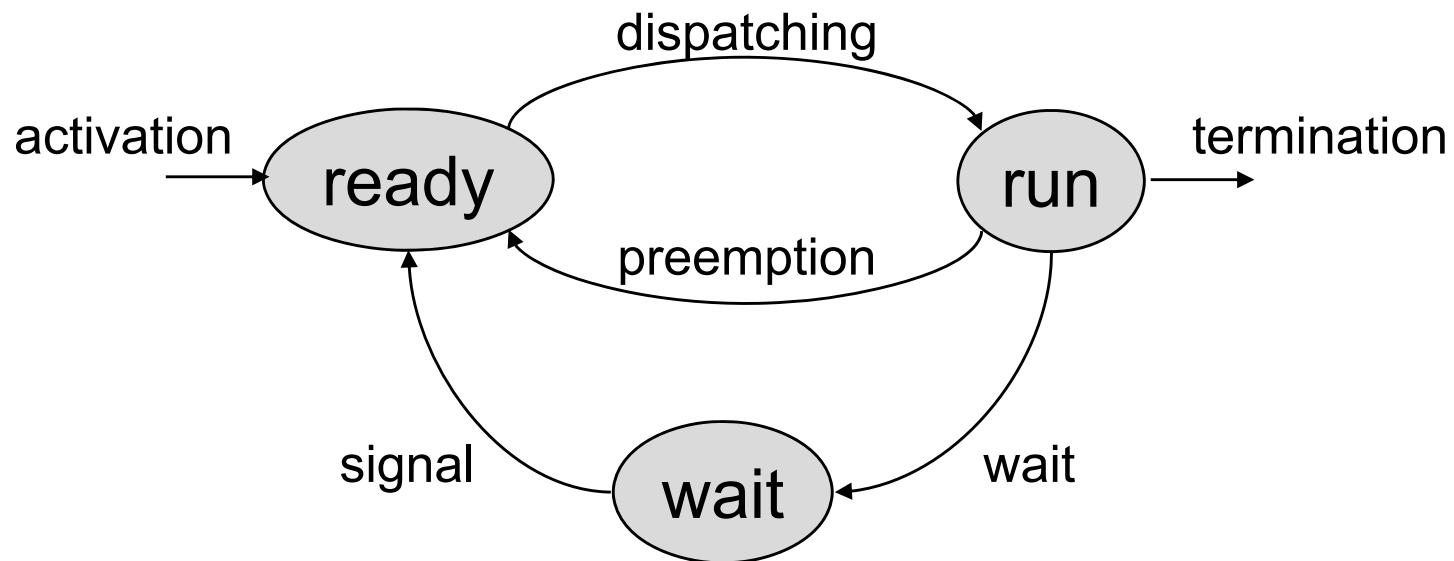
Resource Sharing

- ▶ **Examples** of common resources: data structures, variables, main memory area, file, set of registers, I/O unit,
- ▶ Many shared resources do not allow simultaneous accesses but require **mutual exclusion (exclusive resources)**. A piece of code executed under mutual exclusion constraints is called a **critical section**.



Terms

- ▶ A task waiting for an exclusive resource is said to be **blocked** on that resource. Otherwise, it proceeds by entering the **critical section** and **holds** the resource. When a task leaves a critical section, the associated resource becomes **free**.
- ▶ Waiting state caused by resource constraints:



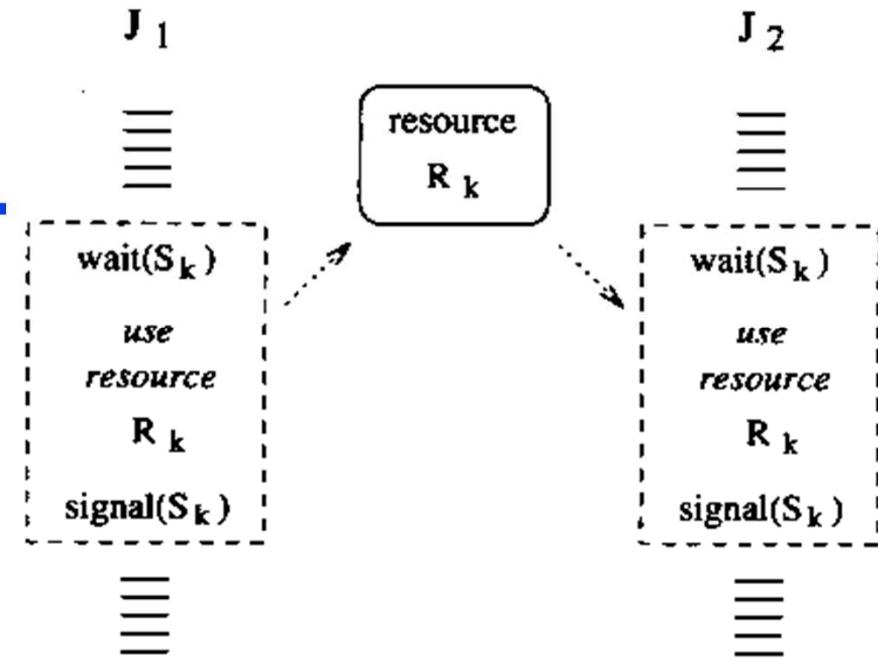
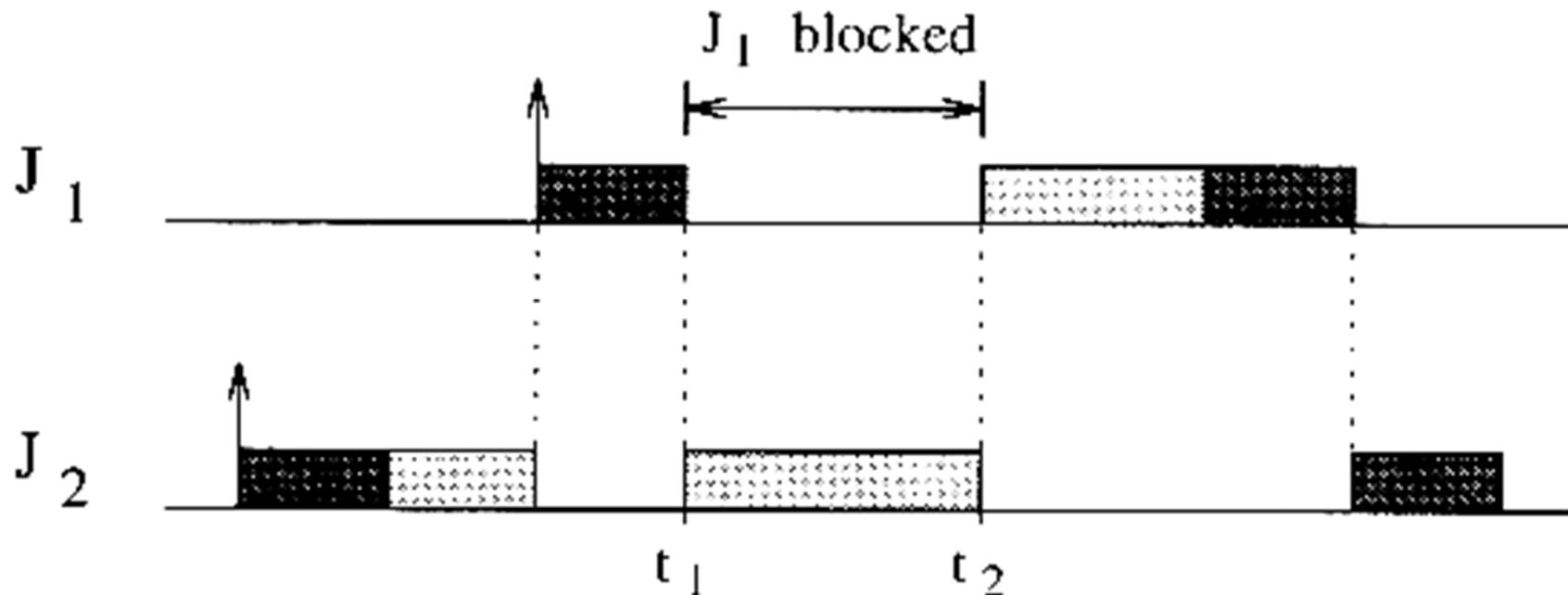
Terms

- ▶ Each **exclusive resource** R_i must be protected by a different **semaphore** S_i , and each critical section operating on a resource must begin with a $\text{wait}(S_i)$ primitive and end with a $\text{signal}(S_i)$ primitive.
- ▶ All tasks blocked on the same resource are kept in a queue associated with the semaphore. When a running task executes a **wait** on a **locked semaphore**, it enters a **waiting state**, until another task executes a **signal** primitive that **unlocks the semaphore**.

Priority Inversion (1)

- ▶ Unavoidable blocking

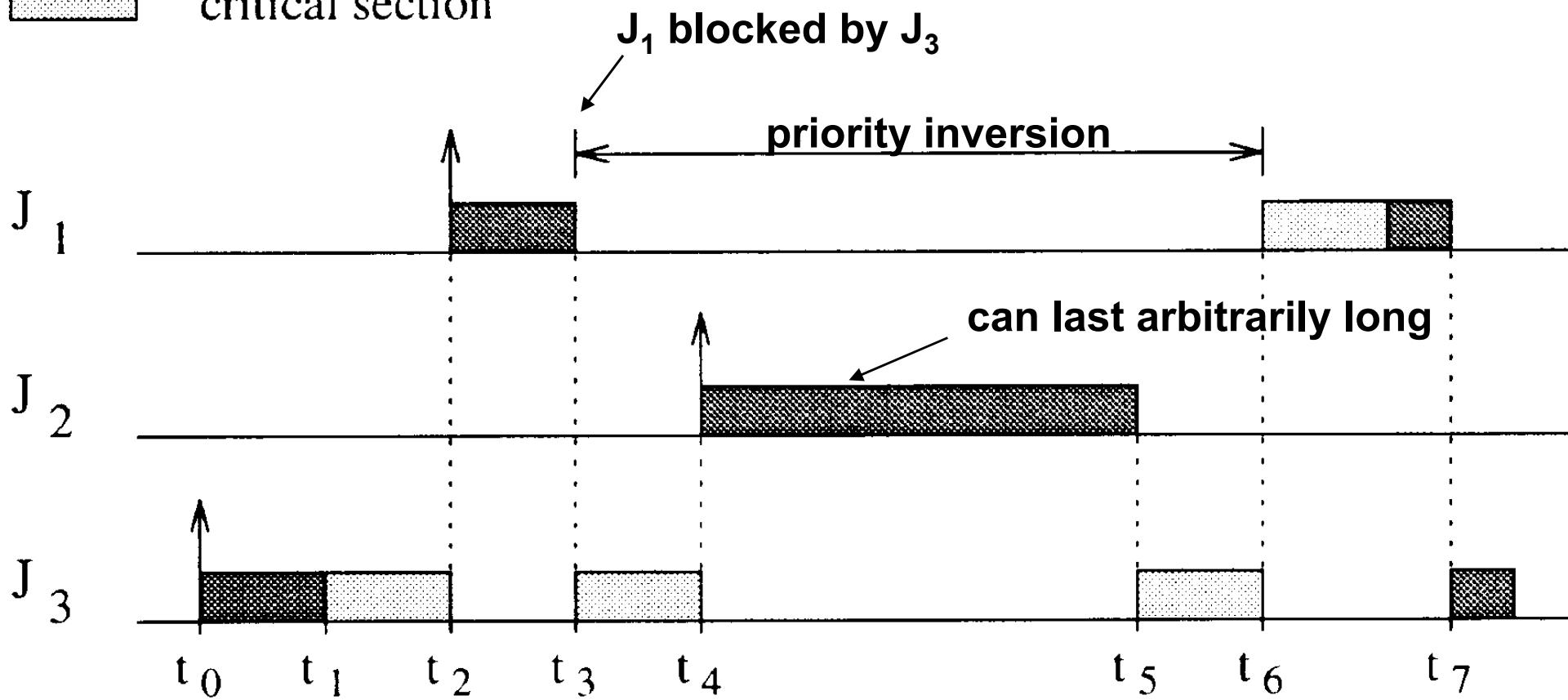
 normal execution
 critical section



Priority Inversion (2)

 normal execution

 critical section

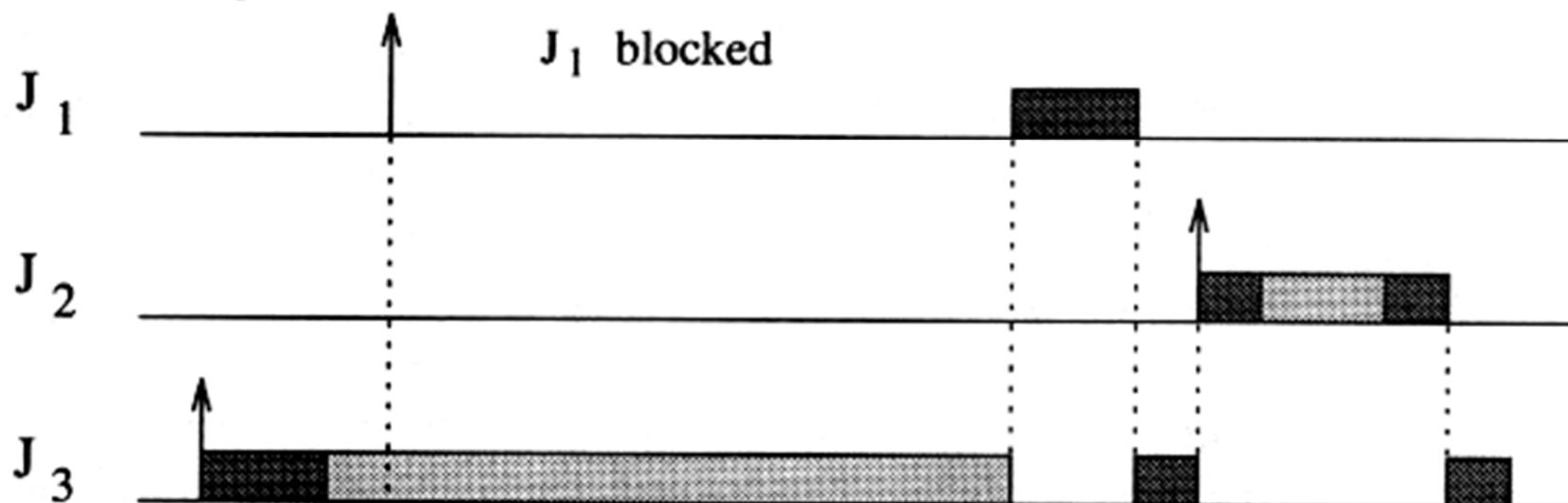


[But97, S.184]

Solutions

- ▶ ***Disallow preemption*** during the execution of all critical sections. Simple, but creates unnecessary blocking as unrelated tasks may be blocked.

 normal execution
 critical section



Resource Access Protocols

- ▶ **Basic idea:** Modify the priority of those tasks that cause blocking. When a task J_i blocks one or more higher priority tasks, it temporarily assumes a higher priority.
- ▶ **Methods:**
 - **Priority Inheritance Protocol** (PIP), for static priorities
 - Priority Ceiling Protocol (PCP), for static priorities
 - Stack Resource Policy (SRP),
for static and dynamic priorities
 - others ...

Priority Inheritance Protocol (PIP)

- ▶ **Assumptions:**

n tasks which cooperate through m shared resources; fixed priorities, all critical sections on a resource begin with a $\text{wait}(S_i)$ and end with a $\text{signal}(S_i)$ operation.

- ▶ **Basic idea:**

When a task J_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.

- ▶ **Terms:**

We distinguish a fixed ***nominal priority*** P_i and an ***active priority*** p_i , larger or equal to P_i . Jobs J_1, \dots, J_n are ordered with respect to nominal priority where J_1 has ***highest priority***. Jobs do not suspend themselves.

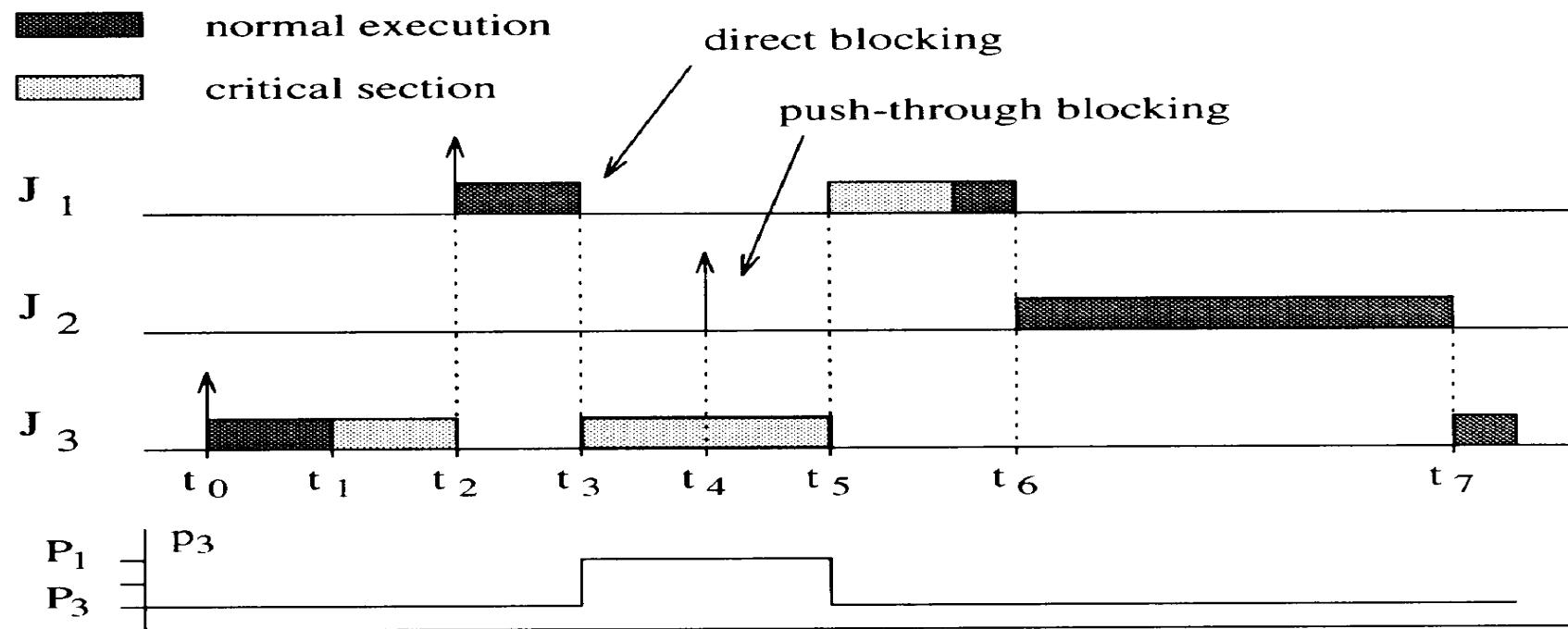
Priority Inheritance Protocol (PIP)

► **Algorithm:**

- Jobs are scheduled based on their **active priorities**. Jobs with the same priority are executed in a FCFS discipline.
- When a job J_i tries to **enter a critical section** and the resource is blocked by a lower priority job, the job J_i is blocked. Otherwise it enters the critical section.
- When a job J_i is **blocked**, it transmits its active priority to the job J_k that holds the semaphore. J_k resumes and executes the rest of its critical section with a priority $p_k = p_i$ (it **inherits** the priority of the highest priority of the jobs blocked by it).
- When J_k exits a critical section, it **unlocks** the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by J_k , then p_k is set to P_k , otherwise it is set to the highest priority of the jobs blocked by J_k .
- Priority inheritance is **transitive**, i.e. if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.

Priority Inheritance Protocol (PIP)

► Example:

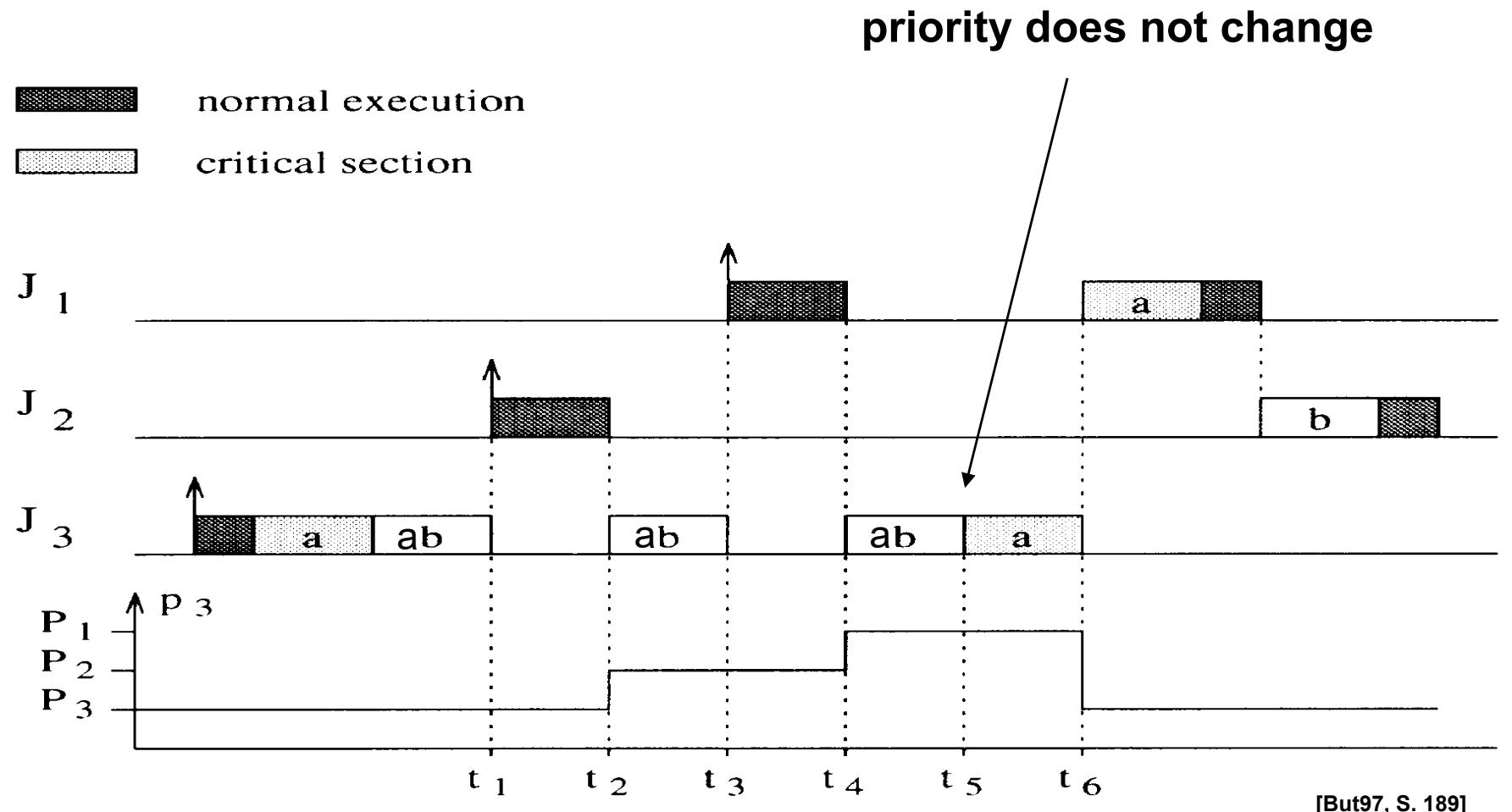


Direct Blocking: higher-priority job tries to acquire a resource held by a lower-priority job

Push-through Blocking: medium-priority job is blocked by a lower-priority job that has inherited a higher priority from a job it directly blocks

Priority Inheritance Protocol (PIP)

- ▶ Example with nested critical sections:



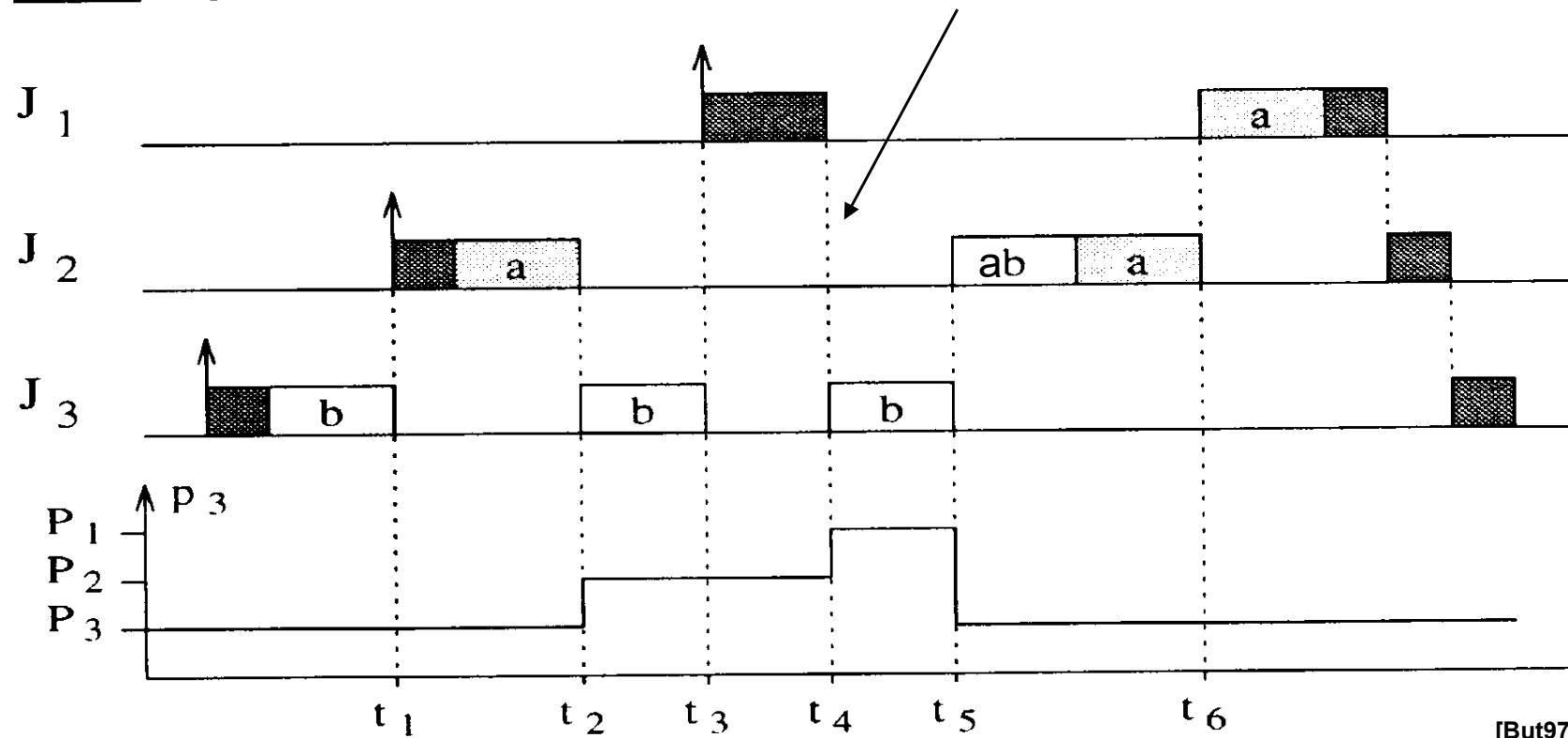
[But97, S. 189]

Priority Inheritance Protocol (PIP)

- ▶ Example of transitive priority inheritance:

 normal execution
 critical section

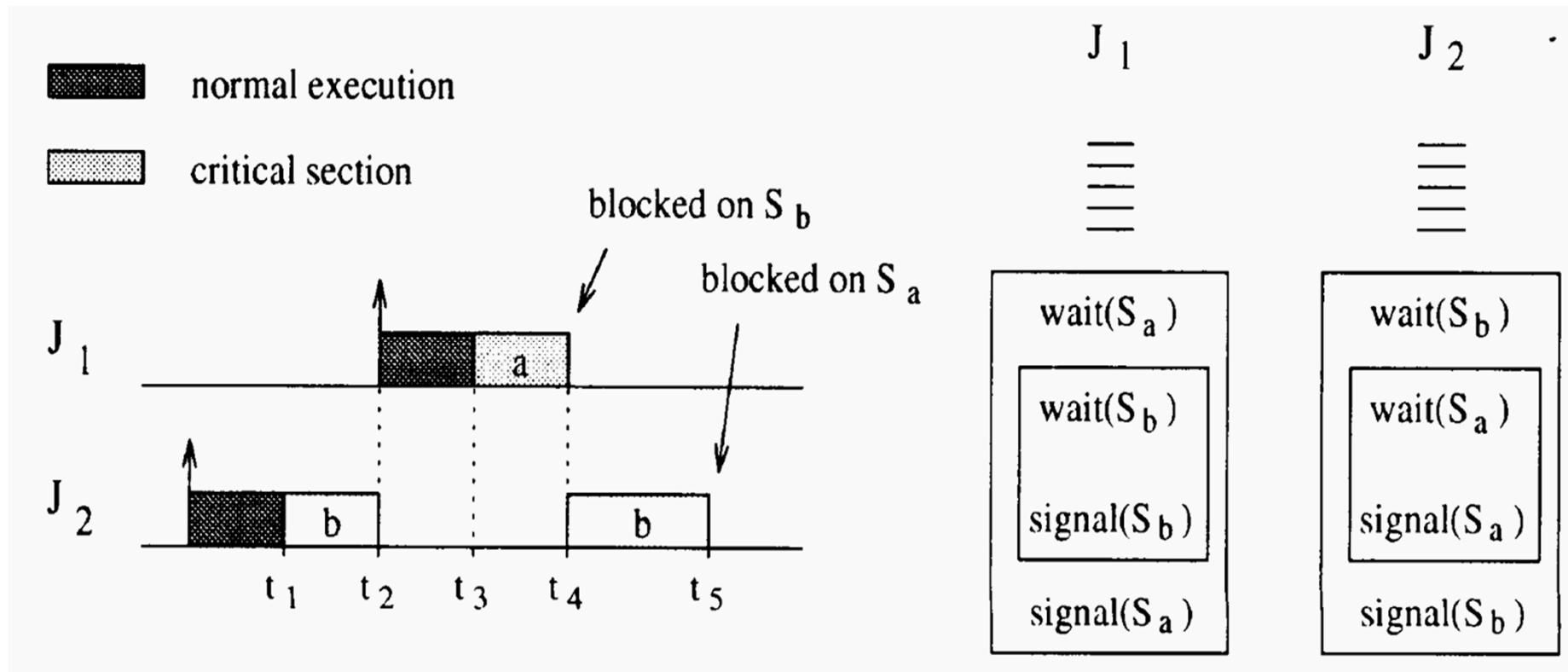
J1 blocked by J2, J2 blocked by J3.
J3 inherits priority from J1 via J2.



[But97, S. 190]

Priority Inheritance Protocol (PIP)

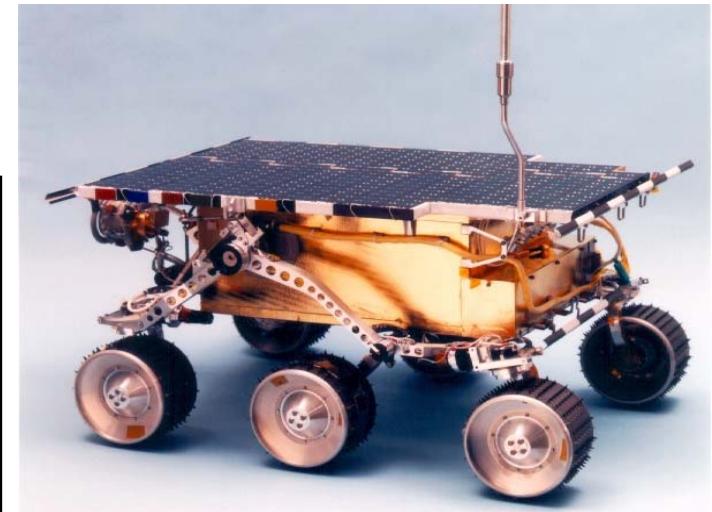
- ▶ Problem: *Deadlock*



[But97, S. 200]

The MARS Pathfinder problem (1)

“But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data.



The MARS Pathfinder problem (2)

“VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.”

“Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft.”

- A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).”

The MARS Pathfinder problem (3)

- The meteorological data gathering task ran as an infrequent, low priority thread, ... When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex.
- The spacecraft also contained a communications task that ran with medium priority.”



High priority: retrieval of data from shared memory

Medium priority: communications task

Low priority: thread collecting meteorological data

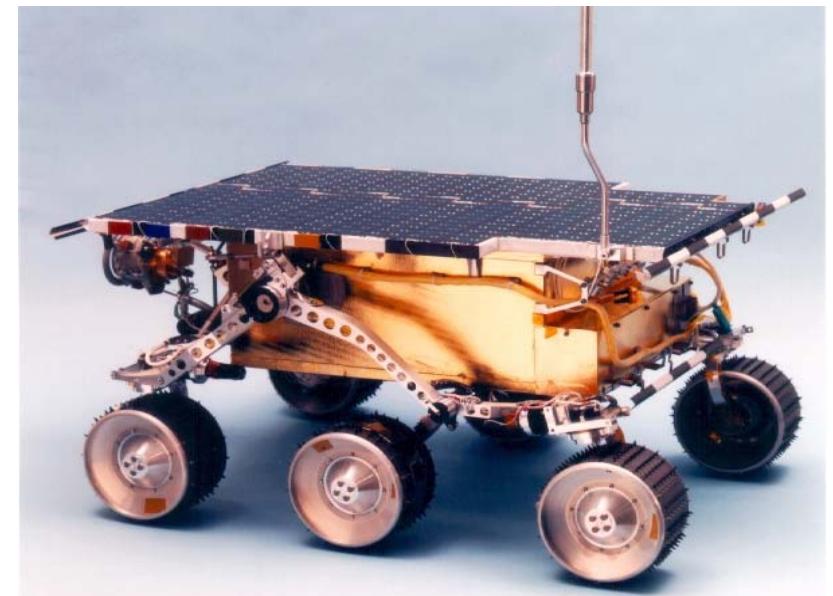
The MARS Pathfinder problem (4)

“Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset. This scenario is a classic case of priority inversion.”

Priority inversion on Mars

Priority inheritance also solved the Mars Pathfinder problem: the VxWorks operating system used in the pathfinder implements a flag for the calls to mutex primitives. This flag allows priority inheritance to be set to “on”. When the software was shipped, it was set to “off”.

The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to “on”, while the Pathfinder was already on the Mars [Jones, 1997].

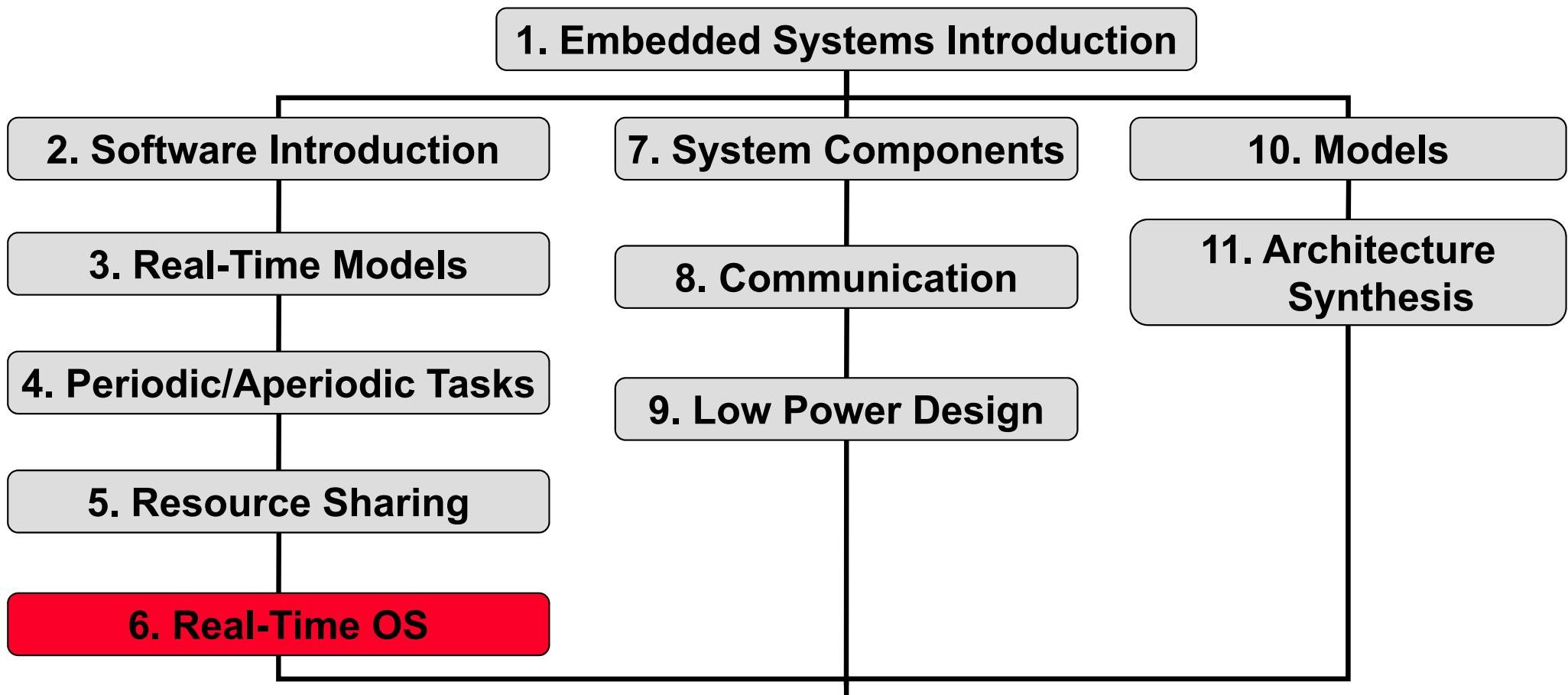


Embedded Systems

6. Real-Time Operating Systems

Lothar Thiele

Contents of Course



*Software and
Programming*

*Processing and
Communication*

Hardware

Embedded OS

- ▶ **Why an OS at all?**
 - Same reasons why we need one for a traditional computer.
 - Not all services are needed for any device.
- ▶ Large variety of **requirements** and environments:
 - Critical applications with high functionality (medical applications, space shuttle, process automation, ...).
 - Critical applications with small functionality (ABS, pace maker, ...)
 - Not very critical applications with varying functionality (smart phone, smart card, microwave oven, ...)

Embedded OS

- ▶ Why is a ***desktop OS not suited?***
 - Monolithic kernel is too feature reach.
 - Monolithic kernel is not modular, fault-tolerant, configurable, modifiable,
 - Takes too much memory space.
 - It is often too ressource hungry in terms of computation time.
 - Not designed for mission-critical applications.
 - Timing uncertainty too large.

Embedded Operating Systems

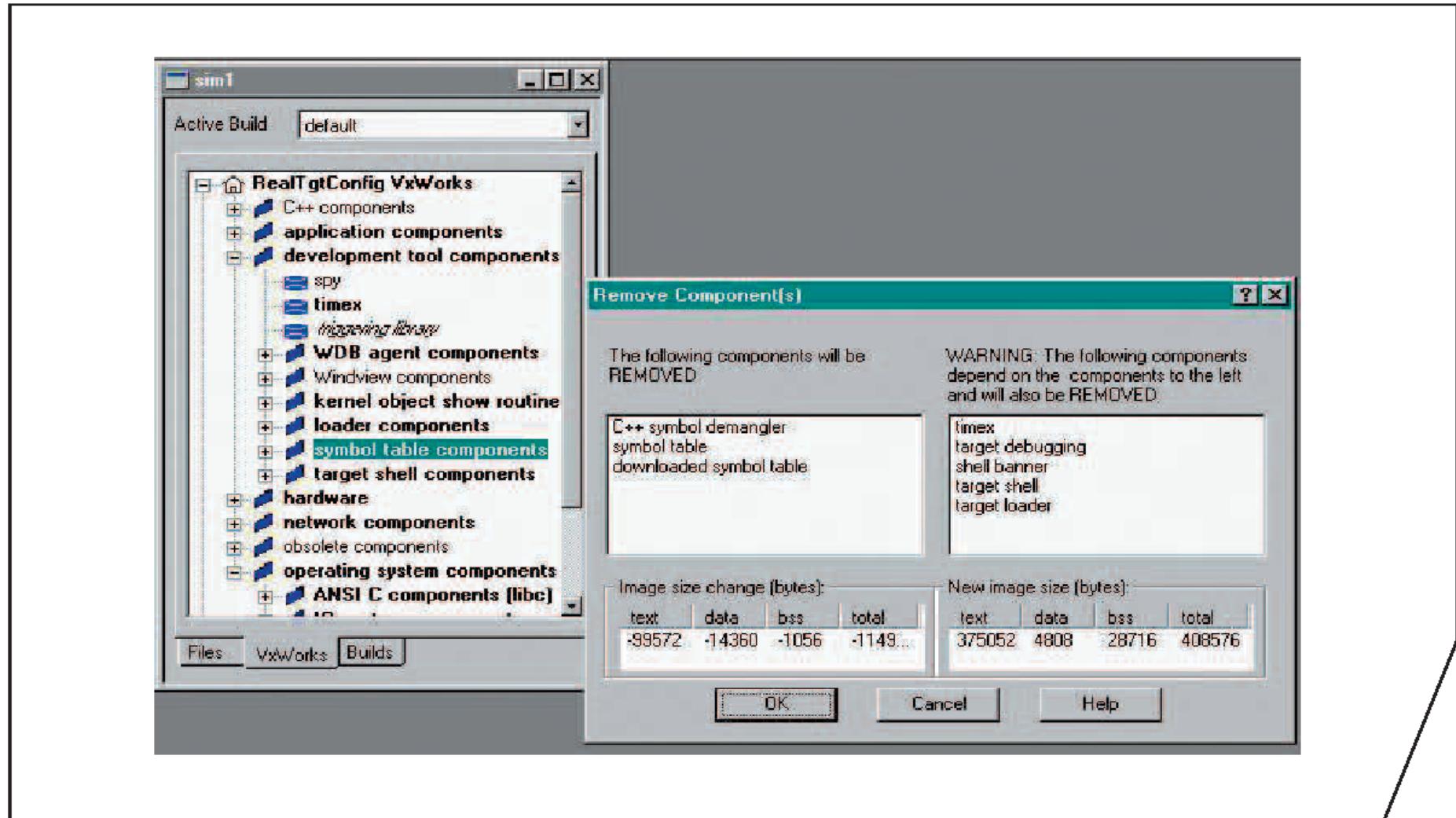
Configurability

- No single RTOS will fit all needs, no overhead for unused functions/data tolerate: configurability is needed.
- For example, there are many embedded systems without external memory, a keyboard, a screen or a mouse.

Configurability examples:

- Simplest form: remove unused functions (by linker for example).
- Conditional compilation (using #if and #ifdef commands).
- Validation is a potential problem of systems with a large number of derived operating systems:
 - each derived operating system must be tested thoroughly;
 - for example, eCos (open source RTOS from Red Hat) includes 100 to 200 configuration points.

Example: Configuration of VxWorks



Automatic dependency analysis and size calculations allow users to quickly customize the VxWORKS operating system.

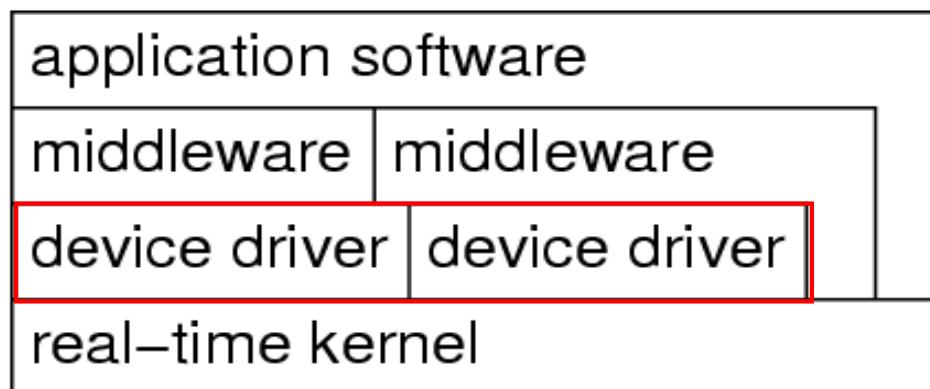
© Windriver

Embedded operating systems

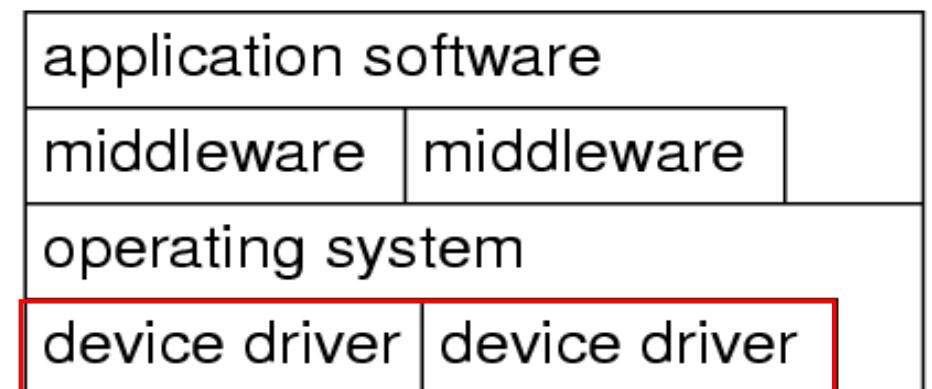
Device drivers handled by tasks instead of hidden integrated drivers:

- Improve predictability; everything goes through scheduler
- Effectively no device that needs to be supported by all versions of the OS, except maybe the system timer.

RTOS



Standard OS



Embedded Operating Systems

Interrupts can be employed by any process

- For standard OS: this would be serious source of unreliability.
- But embedded programs can be considered to be tested . . .
- It is possible to let interrupts directly start or stop tasks (by storing the tasks start address in the interrupt table). More efficient and predictable than going through OS interfaces and services.
- However, compositability suffers: if a specific task is connected to some interrupt, it may be difficult to add another task which also needs to be started by the same event.
- If real-time processing is of concern, time to handle interrupts need to be considered. For example, interrupts may be handled by the scheduler.

Embedded Operating Systems

Protection mechanisms are not always necessary:

- Embedded systems are typically designed for a single purpose, untested programs rarely loaded, software considered reliable.
- *Privileged I/O* instructions not necessary and tasks can do their own I/O.

Example: Let `switch` be the address of some switch. Simply use

```
load register,switch
```

instead of a call to the underlying operating system.

- However, protection mechanisms may be needed for safety and security reasons.

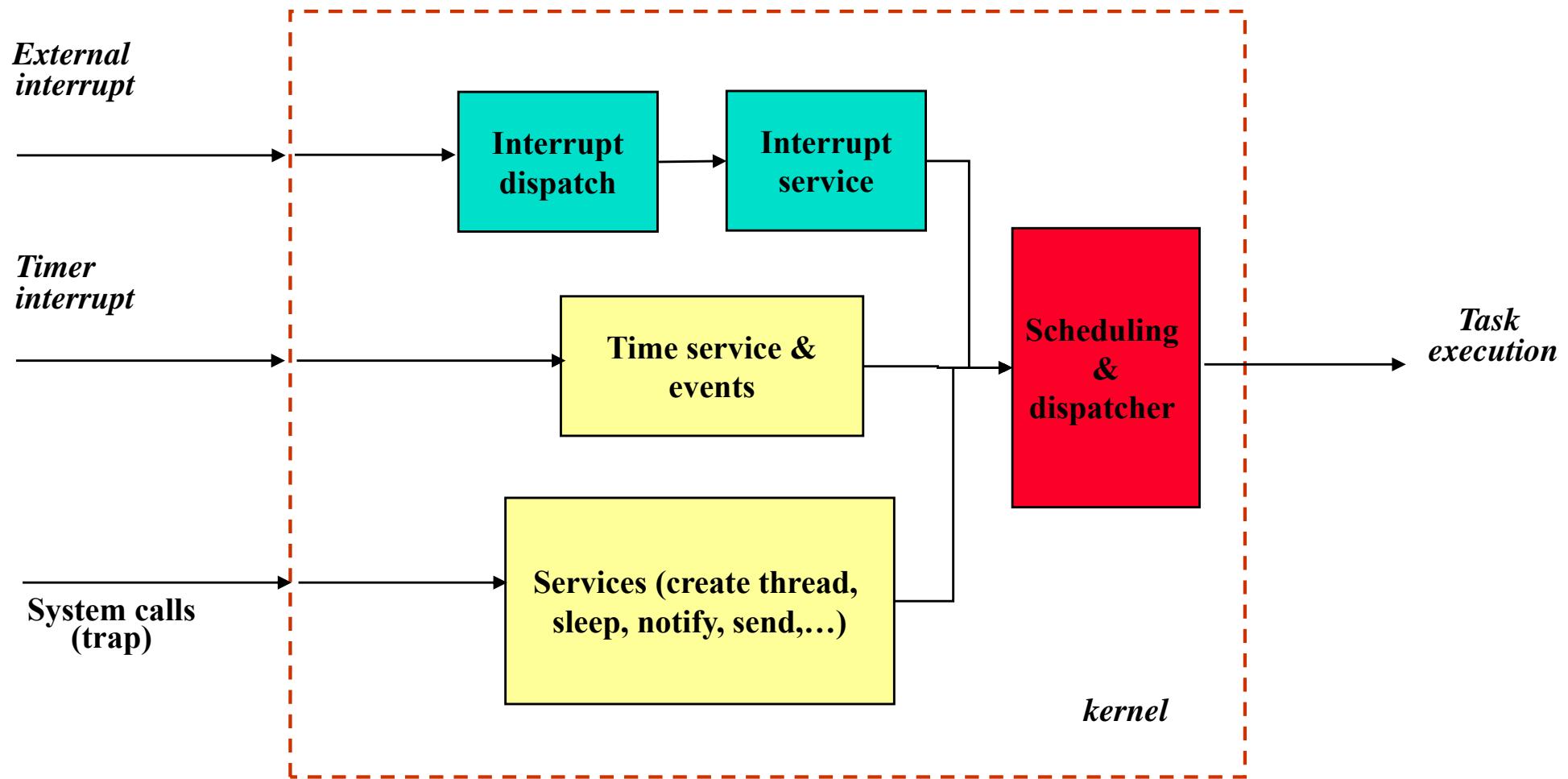
Real-time Operating Systems

- ▶ **A real-time operating system is an operating system that supports the construction of real-time systems.**
- ▶ **Three key requirements:**
 1. The timing behavior of the OS must be *predictable*.
 \forall services of the OS: Upper bound on the execution time!

RTOSs must be deterministic (unlike standard Java for example):

- upper bounds on blocking times need to be available, i.e. during which interrupts are disabled,
- almost all activities are controlled by a real-time scheduler.

Task Management Services



Real-time Operating Systems

2. OS must *manage the timing and scheduling*

- OS possibly has to be aware of deadlines;
(unless scheduling is done off-line).
- OS must provide precise time services with high resolution.

3. The OS must be *fast*

- Practically important.

Main Functionality of RTOS-Kernels

► *Task management*:

- Execution of **quasi-parallel tasks** on a processor using processes or threads (lightweight process) by
 - maintaining process states, process queuing,
 - allowing for preemptive tasks (fast context switching) and quick interrupt handling
- CPU **scheduling** (guaranteeing deadlines, minimizing process waiting times, fairness in granting resources such as computing power)
- Process **synchronization** (critical sections, semaphores, monitors, mutual exclusion)
- Inter-process **communication** (buffering)
- Support of a **real-time clock** as an internal time reference

Task Management

- ▶ ***Task synchronization:***

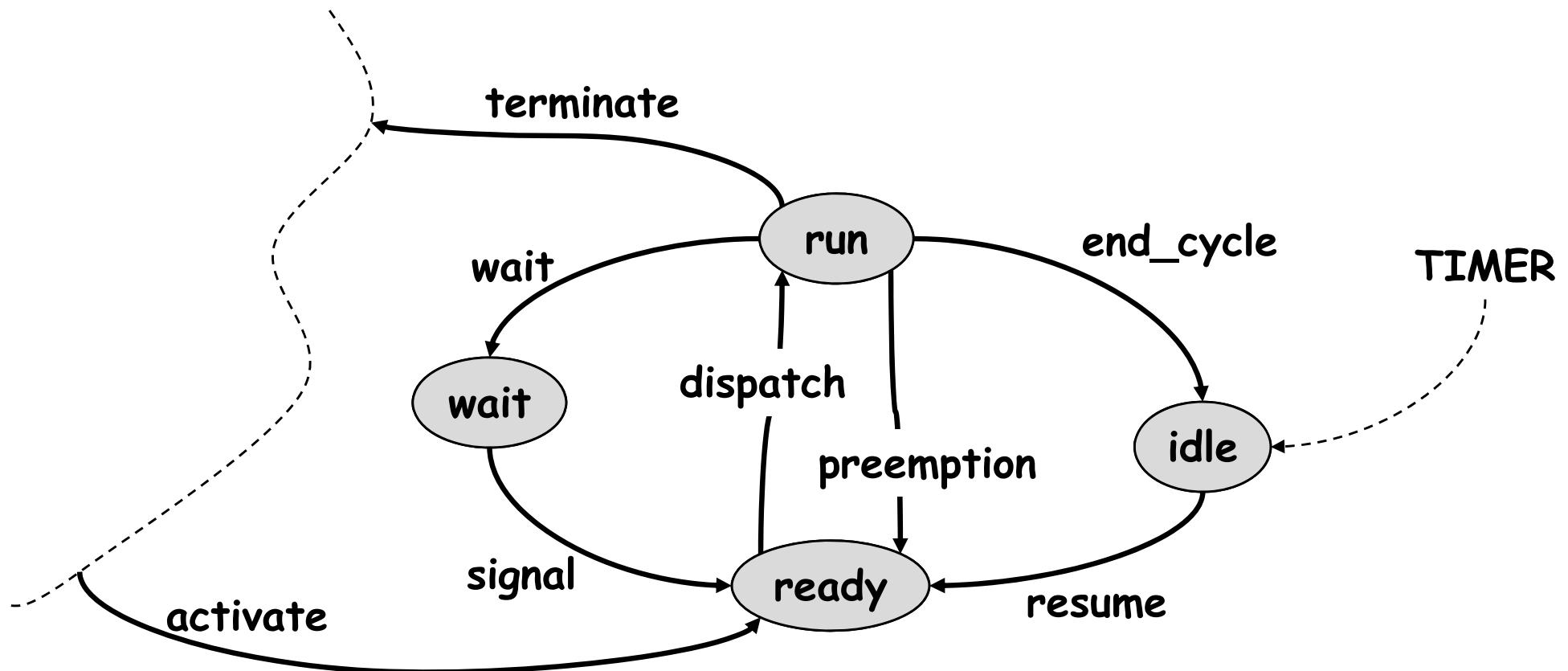
- In classical operating systems, synchronization and mutual exclusion is performed via semaphores and monitors.
- In real-time OS, special semaphores and a deep integration into scheduling is necessary (priority inheritance protocols,).

- ▶ ***Further responsibilities:***

- Initializations of internal data structures (tables, queues, task description blocks, semaphores, ...)

Task States

- ▶ *Minimal Set of Task States:*



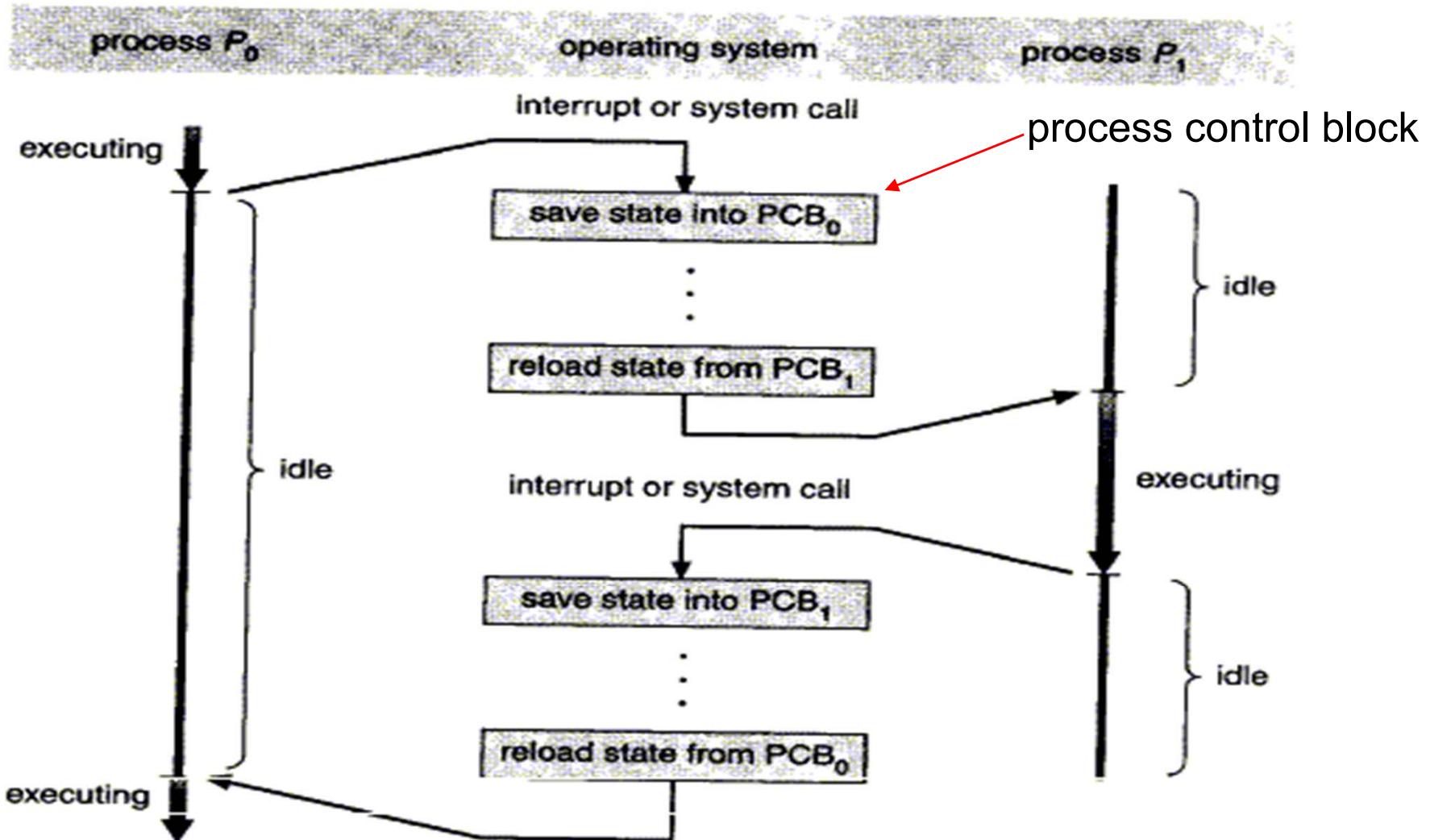
Task states

- ▶ **Run:**
 - A task enters this state as it starts executing on the processor
- ▶ **Ready:**
 - State of those tasks that are ready to execute but cannot be executed because the processor is assigned to another task.
- ▶ **Wait:**
 - A task enters this state when it executes a synchronization primitive to wait for an event, e.g. a wait primitive on a semaphore. In this case, the task is inserted in a queue associated with the semaphore. The task at the head is resumed when the semaphore is unlocked by a signal primitive.
- ▶ **Idle:**
 - A periodic job enters this state when it completes its execution and has to wait for the beginning of the next period.

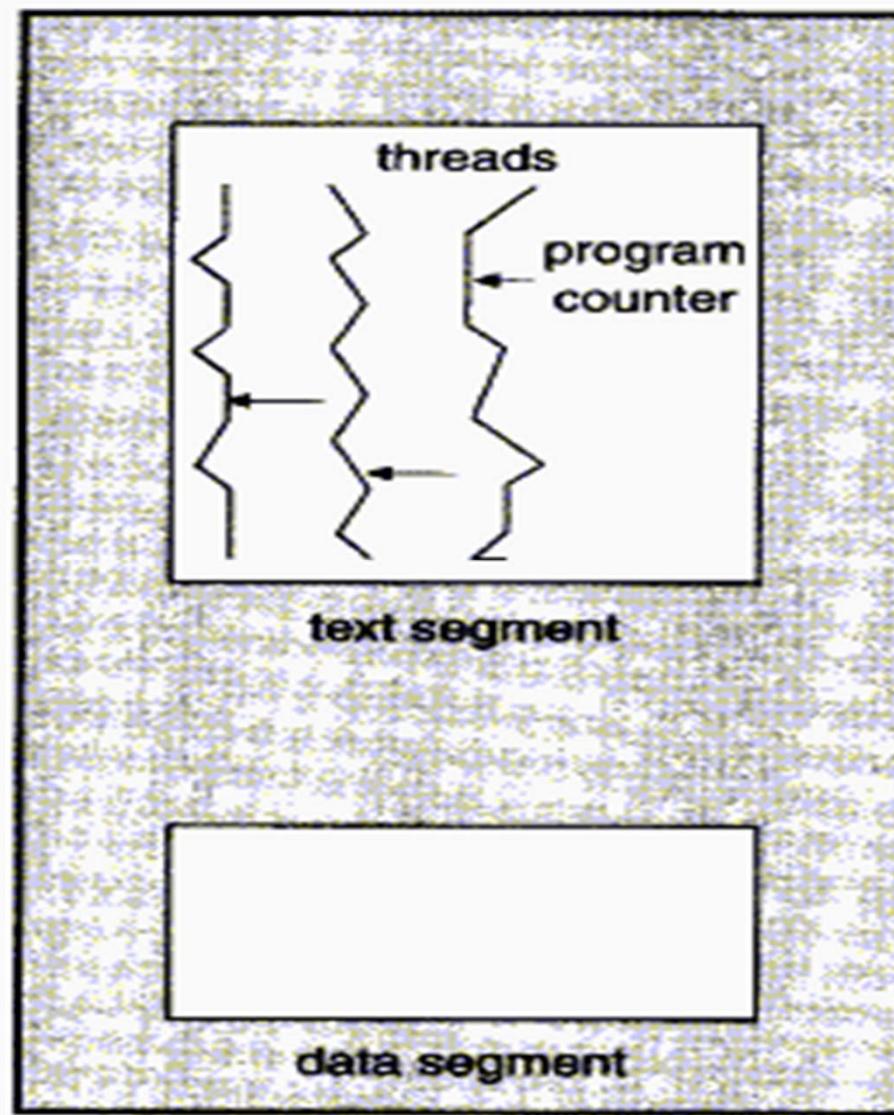
Threads

- ▶ A **thread** is the smallest sequence of programmed instructions that can be managed independently by a scheduler; e.g., a thread is a basic unit of CPU utilization.
- ▶ Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources:
 - Typically **shared by threads**: memory.
 - Typically **owned by threads**: registers, stack.
- ▶ **Thread** advantages and characteristics:
 - Faster to switch between threads; switching between user-level threads requires no major intervention by the operating system.
 - Typically, an application will have a separate thread for each distinct activity.
 - Thread Control Block (TCB) stores information needed to manage and schedule a thread

Context Switching

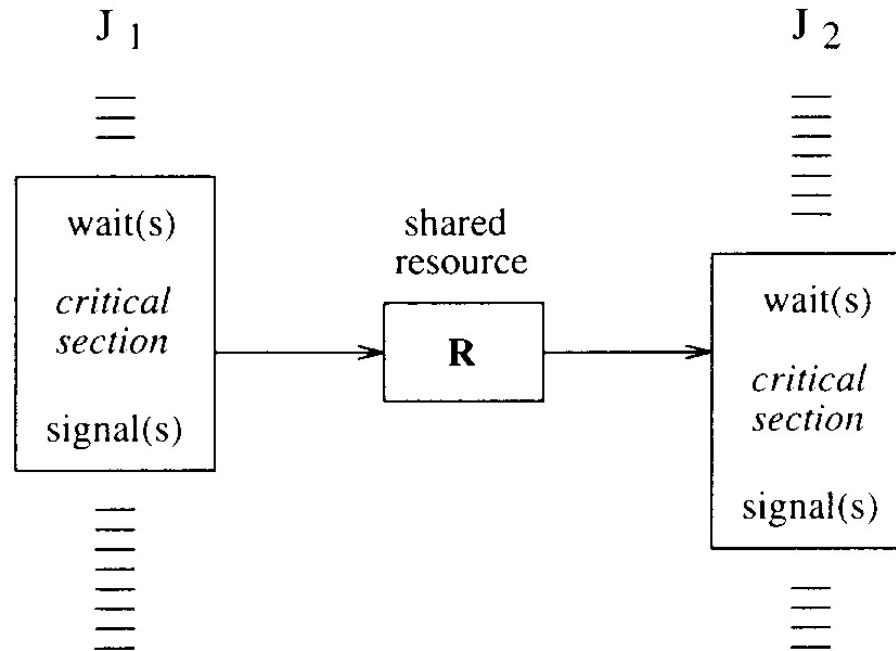


Multiple Threads within a Process



Communication Mechanisms

- ▶ **Problem:** the use of shared resources for implementing message passing schemes may cause priority inversion and blocking.



Communication mechanisms

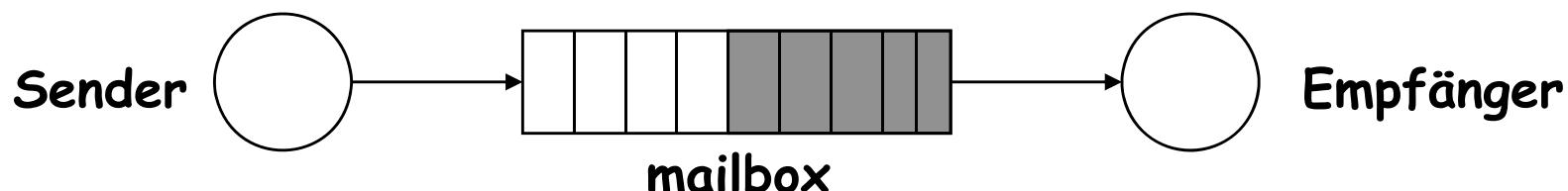
- ▶ **Synchronous communication:**

- Whenever two tasks want to communicate they must be synchronized for a message transfer to take place (**rendez-vous**)
- They have to wait for each other.
- **Problem** in case of dynamic real-time systems: Estimating the maximum blocking time for a process rendez-vous.
- In a **static** real-time environment, the problem can be solved off-line by transforming all synchronous interactions into precedence constraints.

Communication mechanisms

► **Asynchronous communication:**

- Tasks do not have to wait for each other
- The sender just deposits its message into a channel and continues its execution; similarly the receiver can directly access the message if at least a message has been deposited into the channel.
- More suited for real-time systems than synchronous comm.
- **Mailbox**: Shared memory buffer, FIFO-queue, basic operations are send and receive, usually has fixed capacity.
- **Problem**: Blocking behavior if channel is full or empty; alternative approach is provided by cyclical asynchronous buffers.



Class 1: Fast Proprietary Kernels

Fast proprietary kernels

For hard real-time systems, these kernels are questionable, because they are designed to be fast, rather than to be predictable in every respect

Examples include

FreeRTOS, QNX, eCOS, RT-LINUX, VxWORKS, LynxOS.

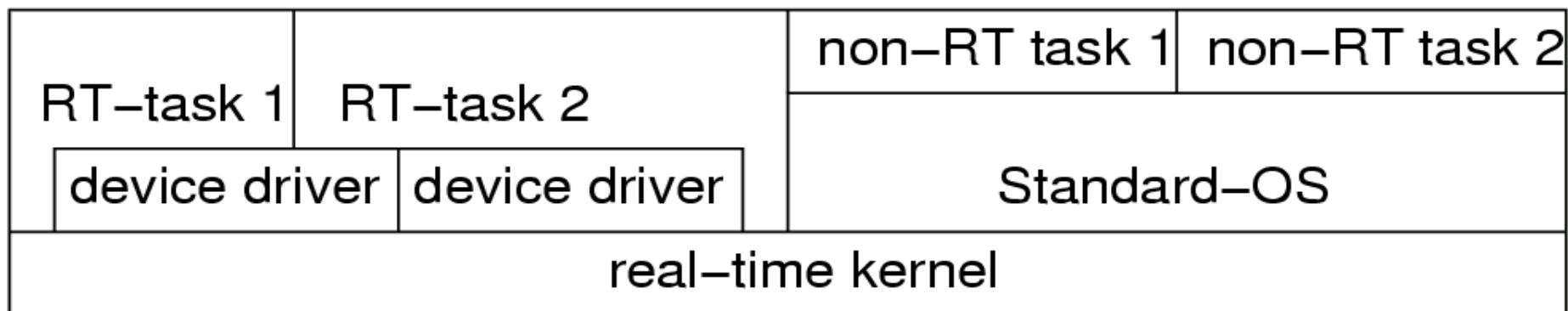
Class 2: Extensions to Standard OSs

Real-time extensions to standard OS:

Attempt to exploit comfortable main stream OS.

RT-kernel running all RT-tasks.

Standard-OS executed as one task.

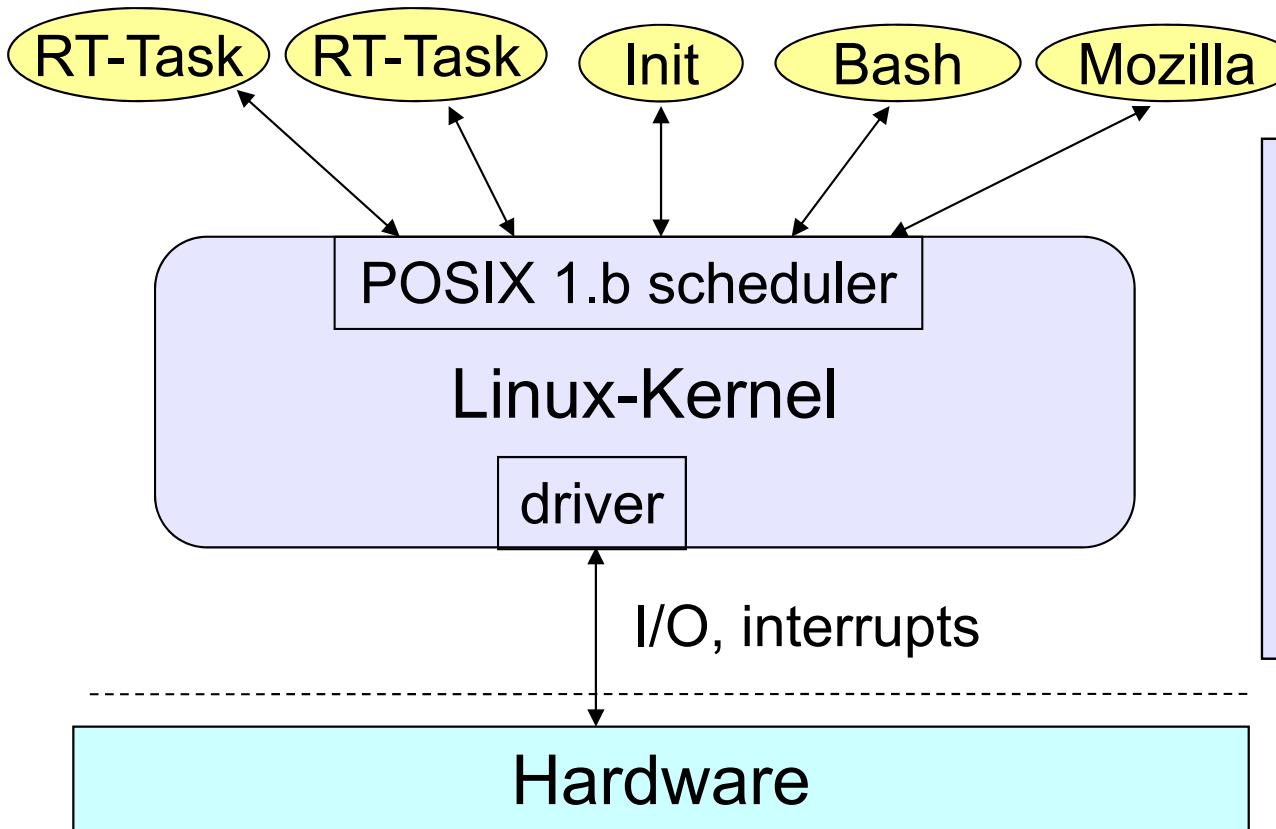


- + Crash of standard-OS does not affect RT-tasks;
- RT-tasks cannot use Standard-OS services;
less comfortable than expected

revival of the concept:
hypervisor

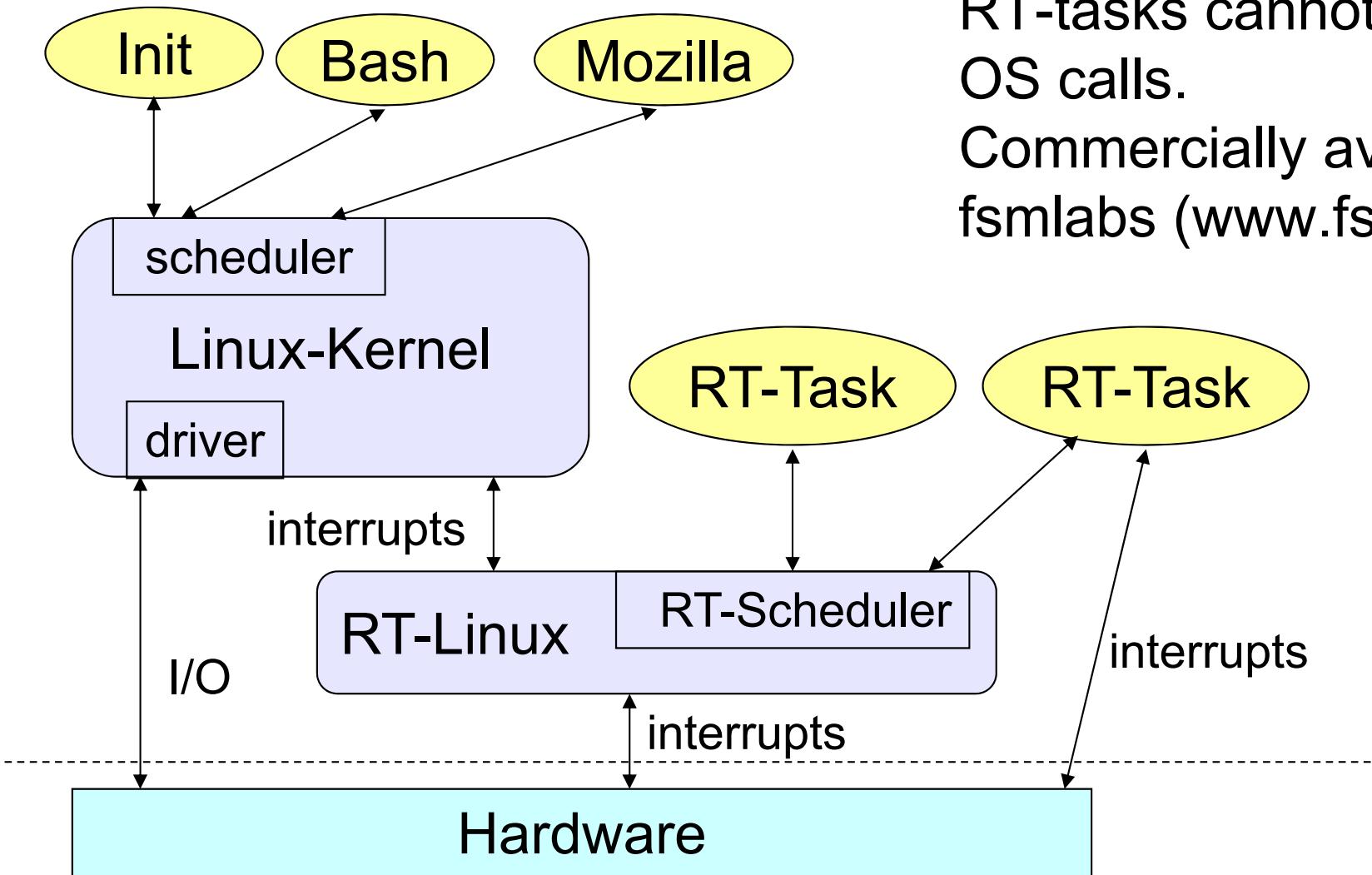
Example: Posix 1.b RT-extensions to Linux

Standard scheduler can be replaced by POSIX scheduler implementing priorities for RT tasks



Special RT-calls and standard OS calls available.
Easy programming, no guarantee for meeting deadline

Example: RT Linux



RT-tasks cannot use standard OS calls.
Commercially available from fsm labs (www.fsmlabs.com)

Class 3: Research Systems

Research systems trying to avoid limitations:

- Include L4, seL4, NICTA, ERIKA, SHARK

Research issues:

- low overhead memory protection,
- temporal protection of computing resources
- RTOSes for on-chip multiprocessors
- quality of service (QoS) control (besides real-time constraints)
- formally verified kernel properties

List of current real-time operating systems:

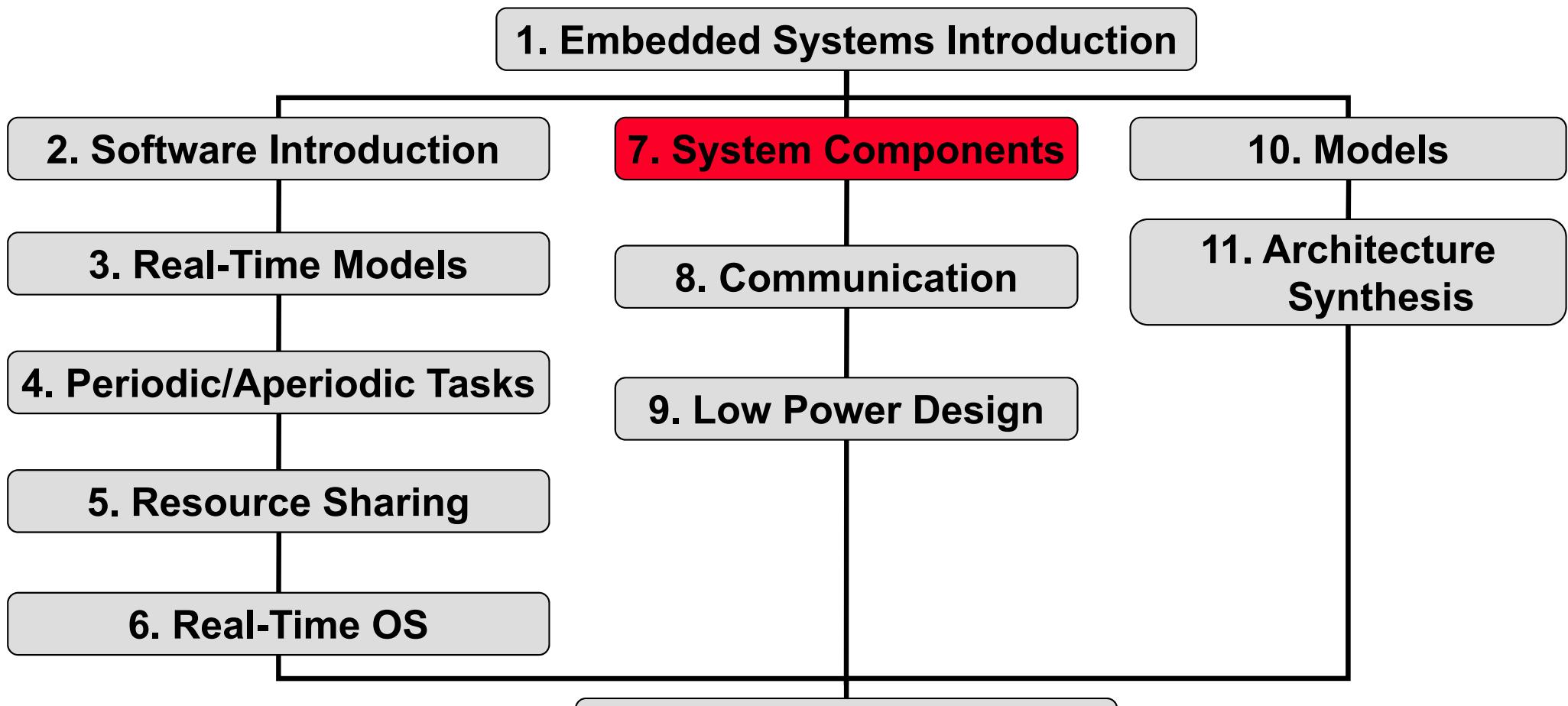
http://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems

Embedded Systems

7. System Components

Lothar Thiele

Contents of Course



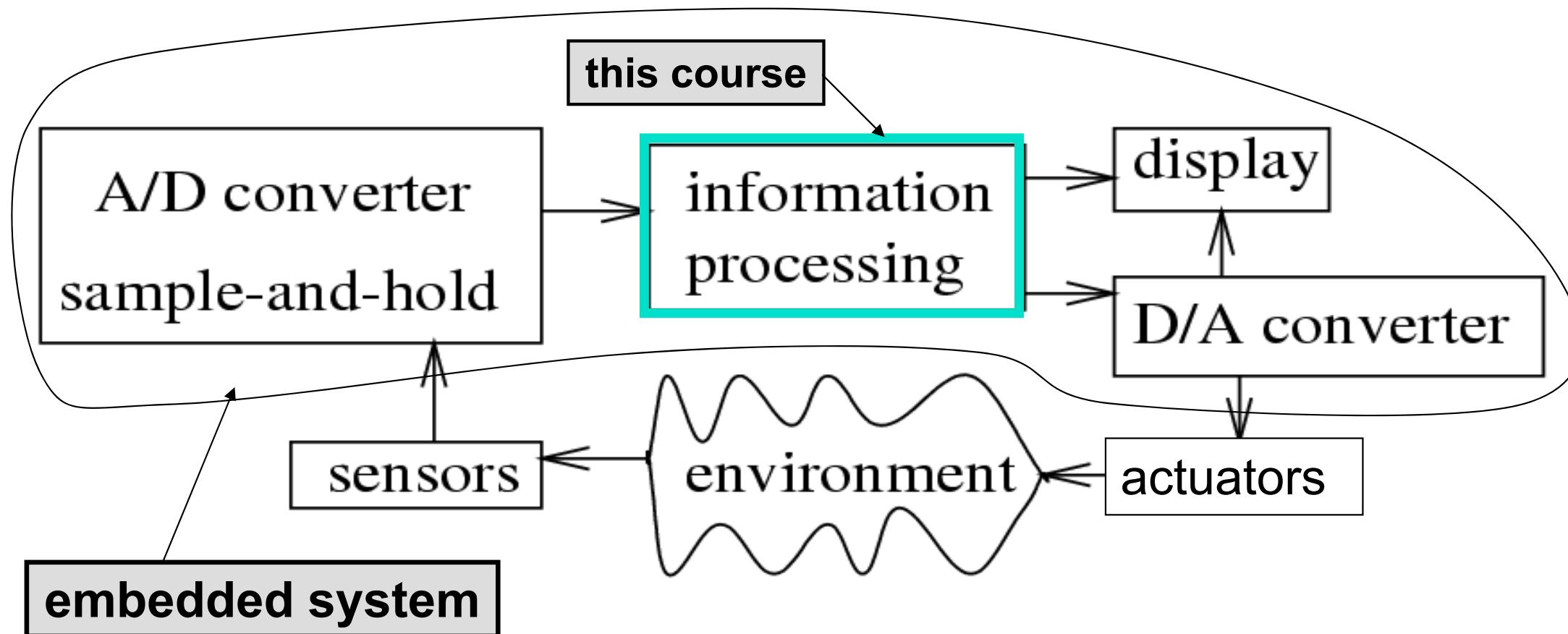
*Software and
Programming*

*Processing and
Communication*

Hardware

Embedded System Hardware

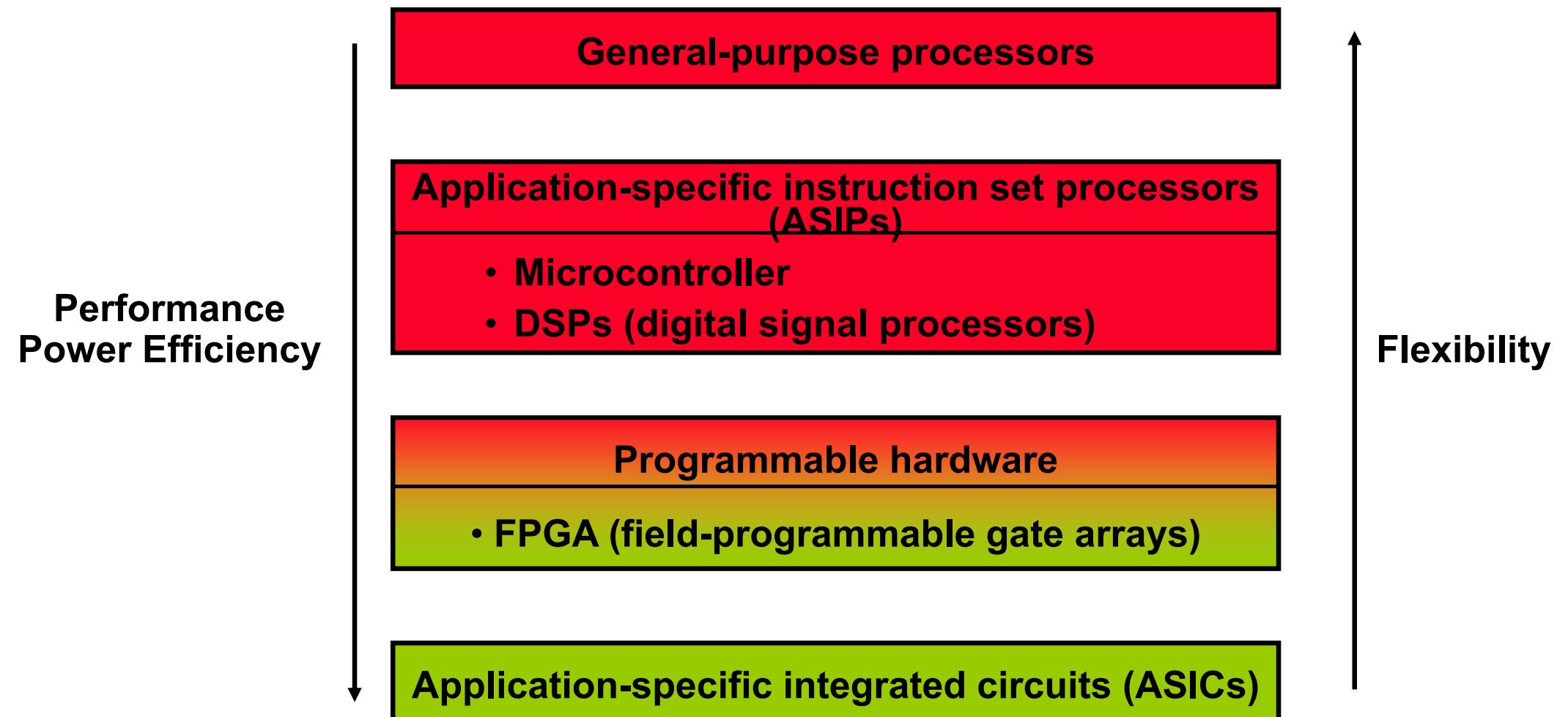
Embedded system hardware is frequently used in a loop
(*„hardware in a loop“*):



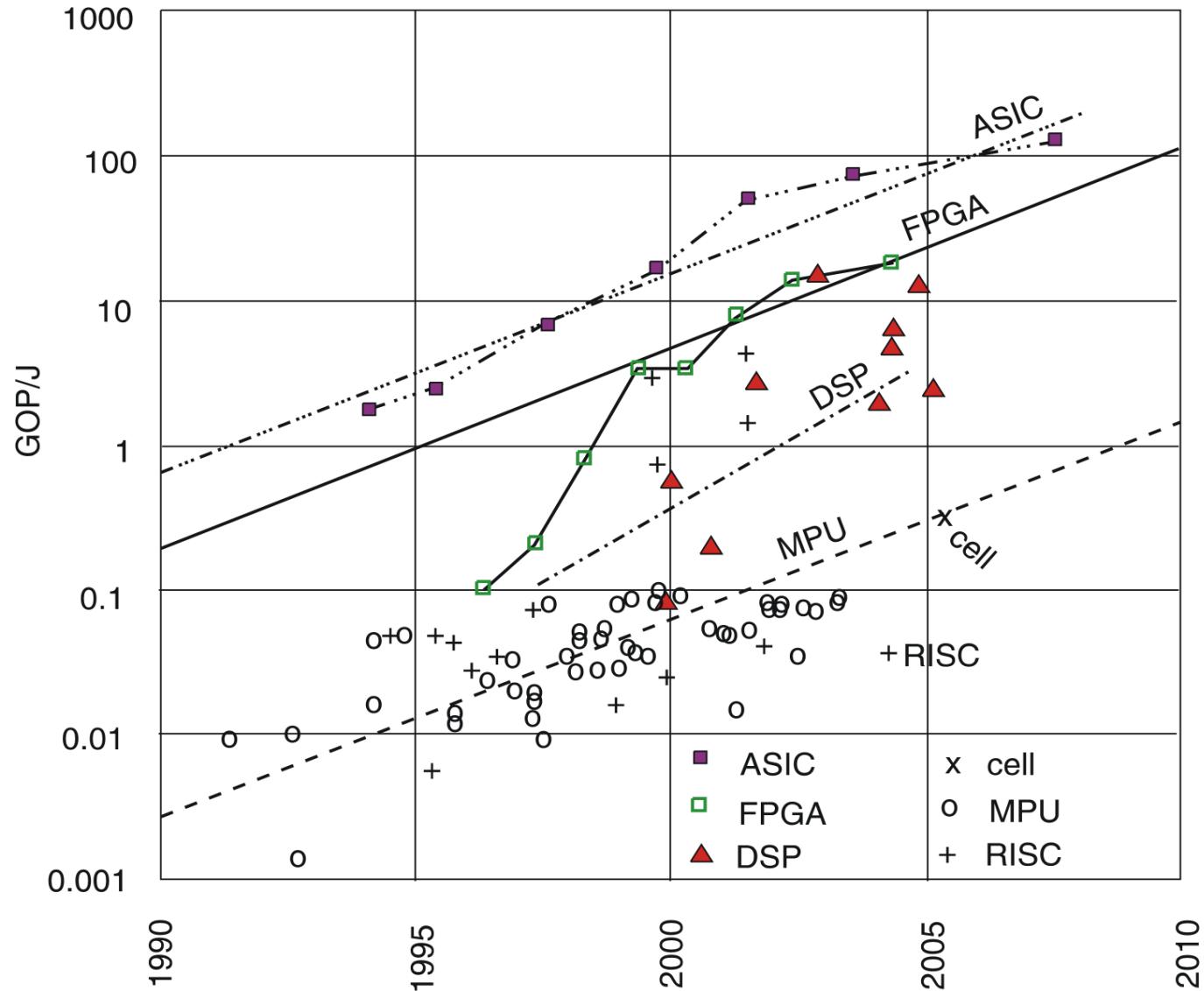
Topics

- ▶ ***System Specialization***
- ▶ Application Specific Instruction Sets
 - Micro Controller
 - Digital Signal Processors and VLIW
- ▶ Programmable Hardware
- ▶ ASICs
- ▶ System-on-Chip

Implementation Alternatives



Energy Efficiency



© Hugo De Man,
IMEC, Philips, 2007

General-purpose Processors

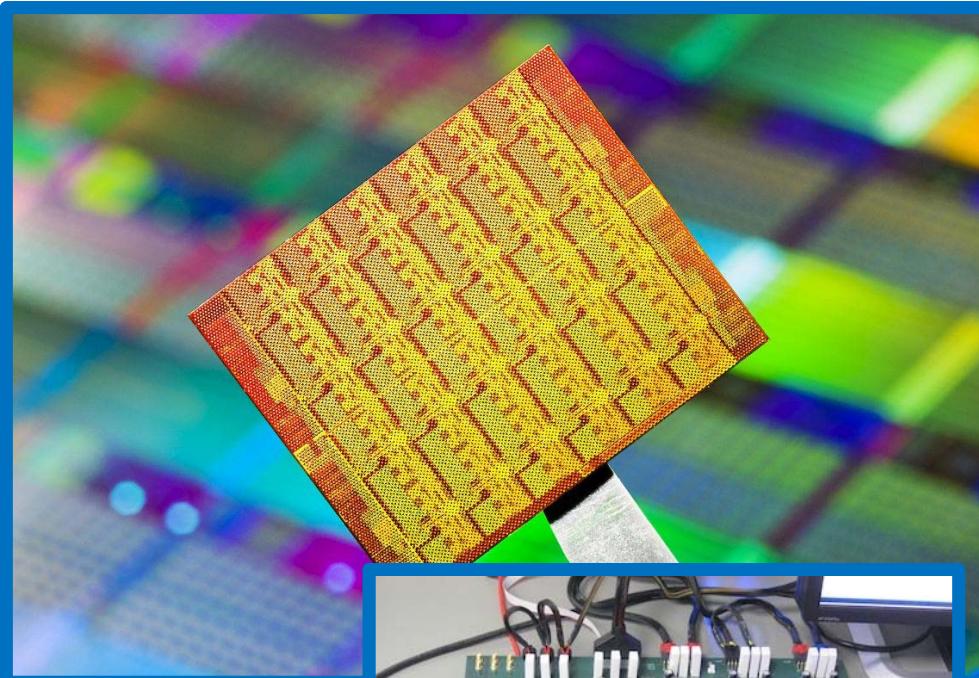
- ▶ ***High performance***
 - Highly optimized circuits and technology
 - Use of parallelism
 - superscalar: dynamic scheduling of instructions
 - super-pipelining: instruction pipelining, branch prediction, speculation
 - complex memory hierarchy
- ▶ ***Not suited for real-time applications***
 - Execution times are highly unpredictable because of intensive resource sharing and dynamic decisions
- ▶ ***Properties***
 - Good average performance for large application mix
 - High power consumption

General-purpose Processors

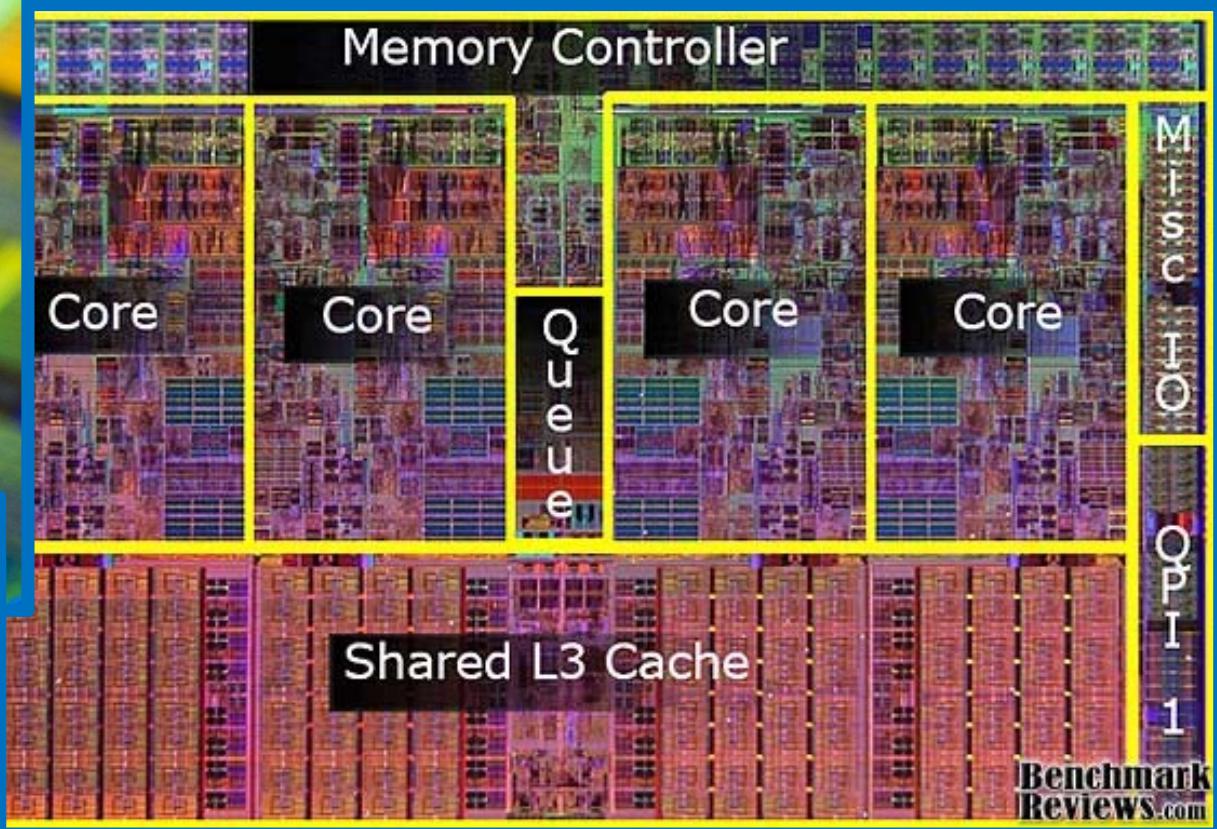
► *Multicore Processors*

- Potential of providing higher execution performance by exploiting parallelism
- Especially useful in high-performance embedded systems, e.g. autonomous driving
- Disadvantages and problems for embedded systems:
 - Increased interference on shared resources such as buses and shared caches
 - Increased timing uncertainty
 - Often, there is limited parallelism in embedded applications

Multicore Examples

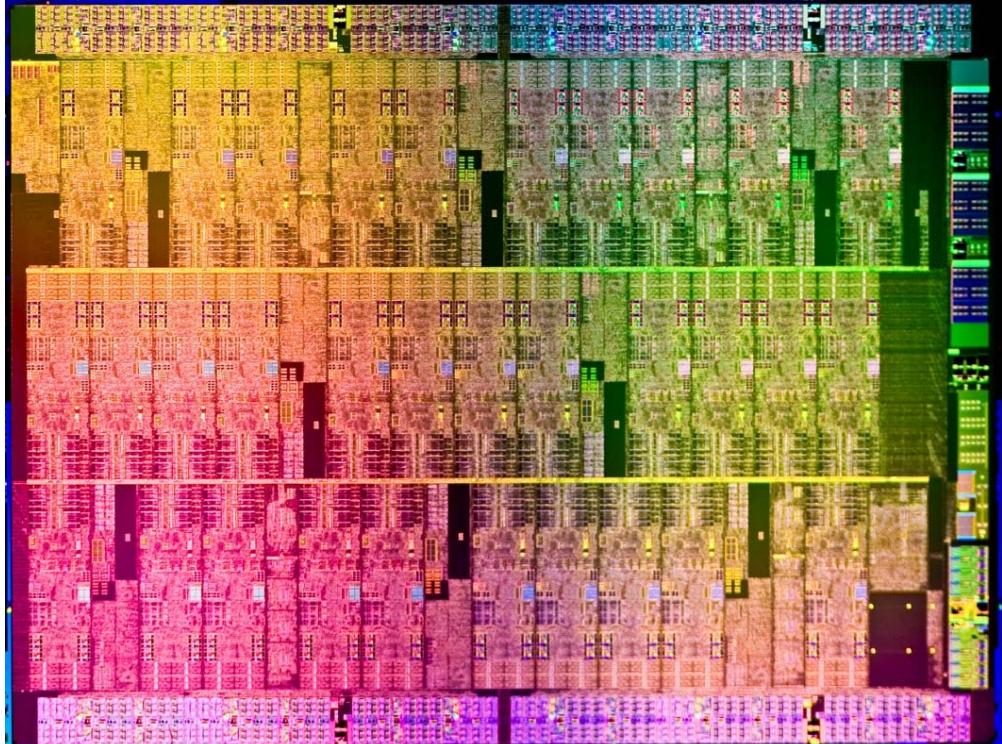


48 cores

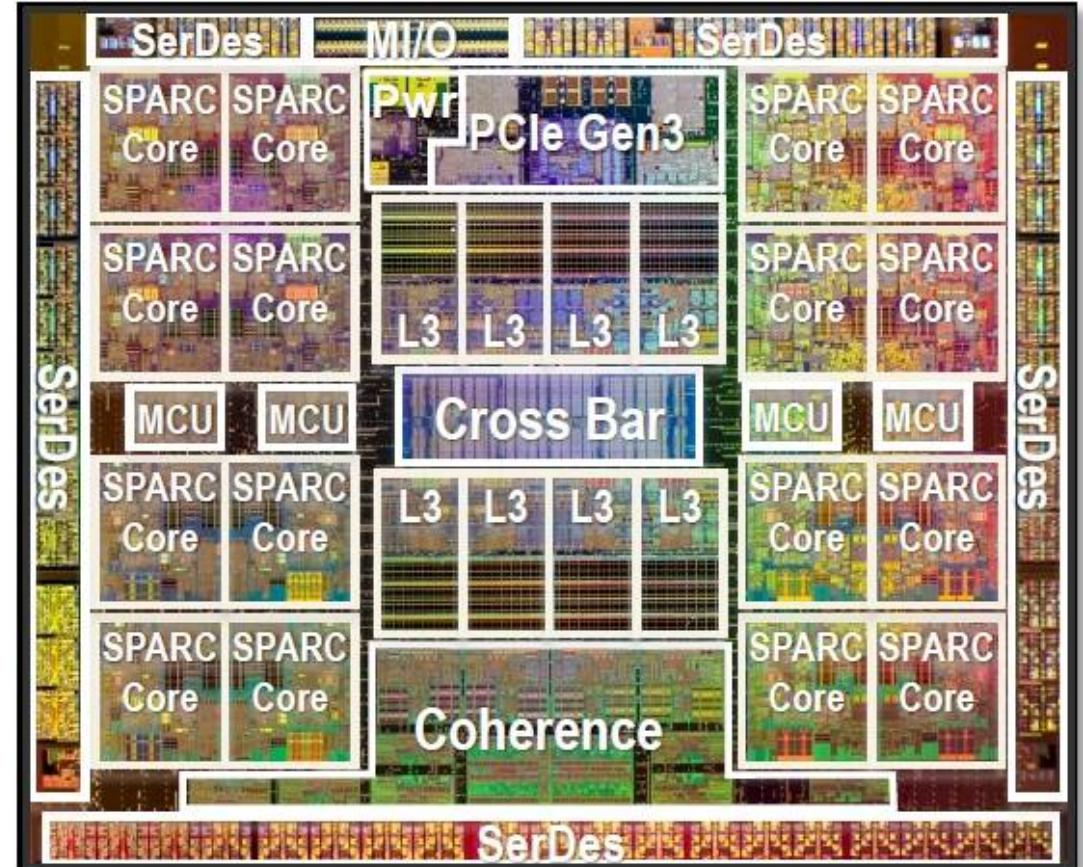


4 cores

Multicore Examples



Intel Xeon Phi
(5 Billion transistors,
22nm technology,
350mm² area)



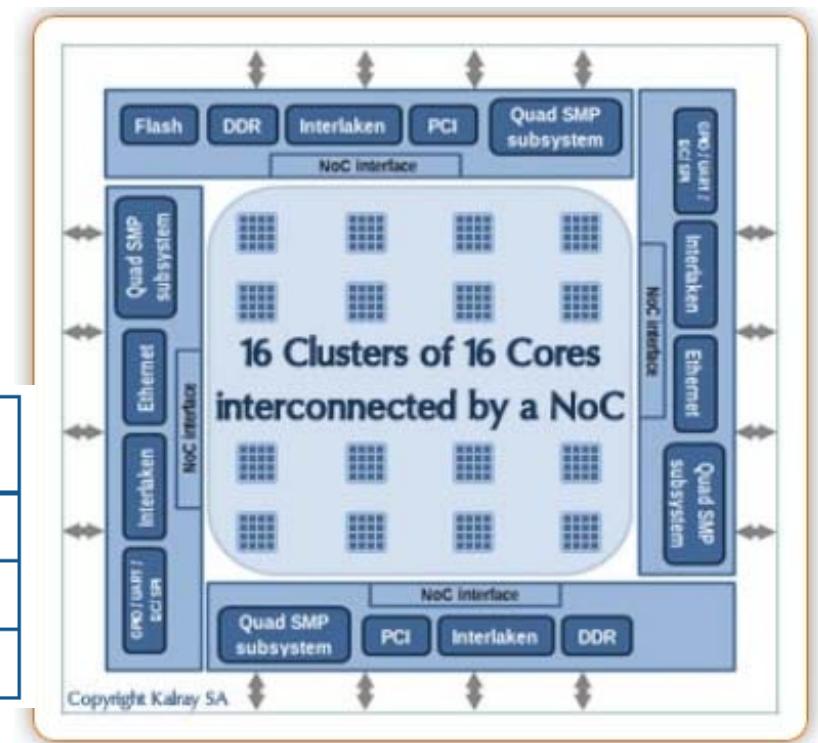
Oracle Sparc T5

Embedded Multicore Example

- ▶ Recent development:
 - Specialize multicore processors towards real-time processing and low power consumption
 - Target domains:



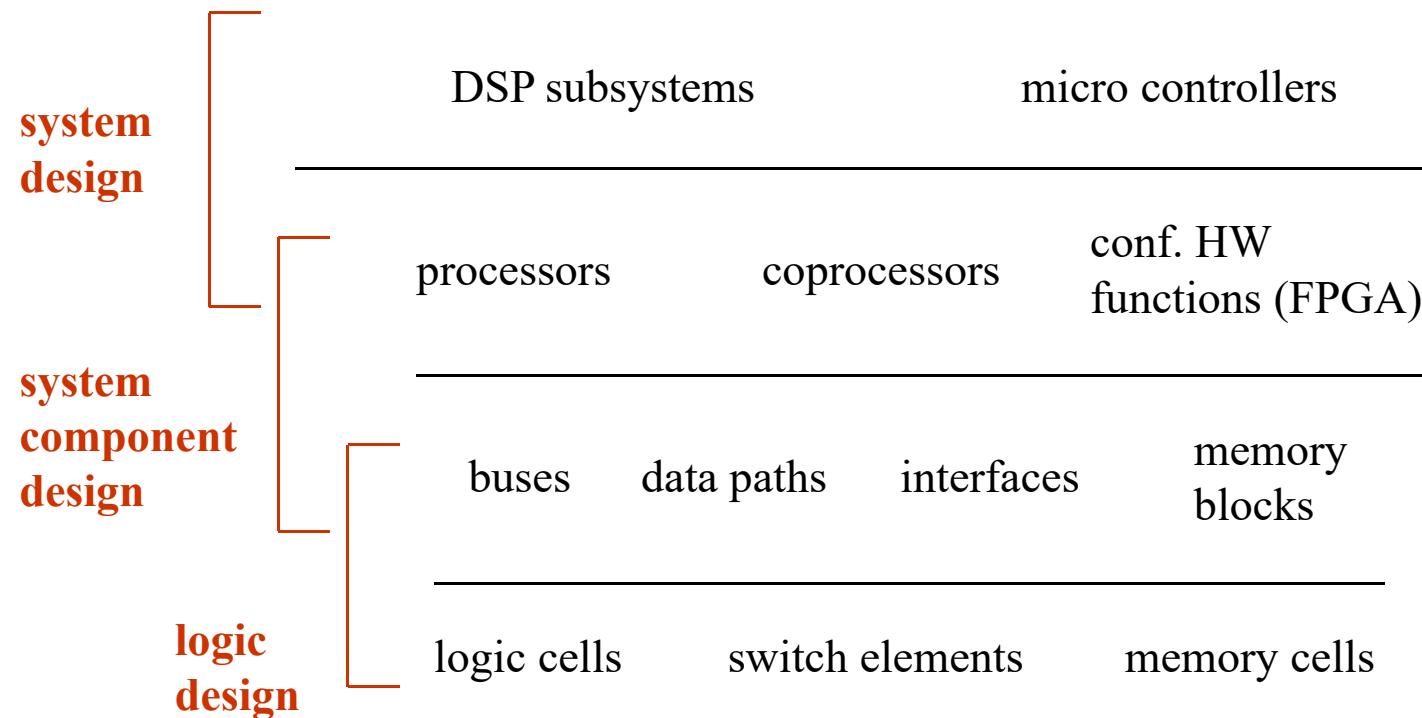
Core Generation	Number of Processing Cores	GFLOPS/W	GOPS/W
Andey	256	25	75
Bostan (2014)	256	50	80
Coolidge (2015)	64/256/1024	75	115



System Specialization

- ▶ The main difference between general purpose highest volume microprocessors and embedded systems is ***specialization***.
- ▶ ***Specialization should respect flexibility***
 - application domain specific systems shall cover a class of applications
 - some flexibility is required to account for late changes, debugging
- ▶ ***System analysis required***
 - identification of application properties which can be used for specialization
 - quantification of individual specialization effects

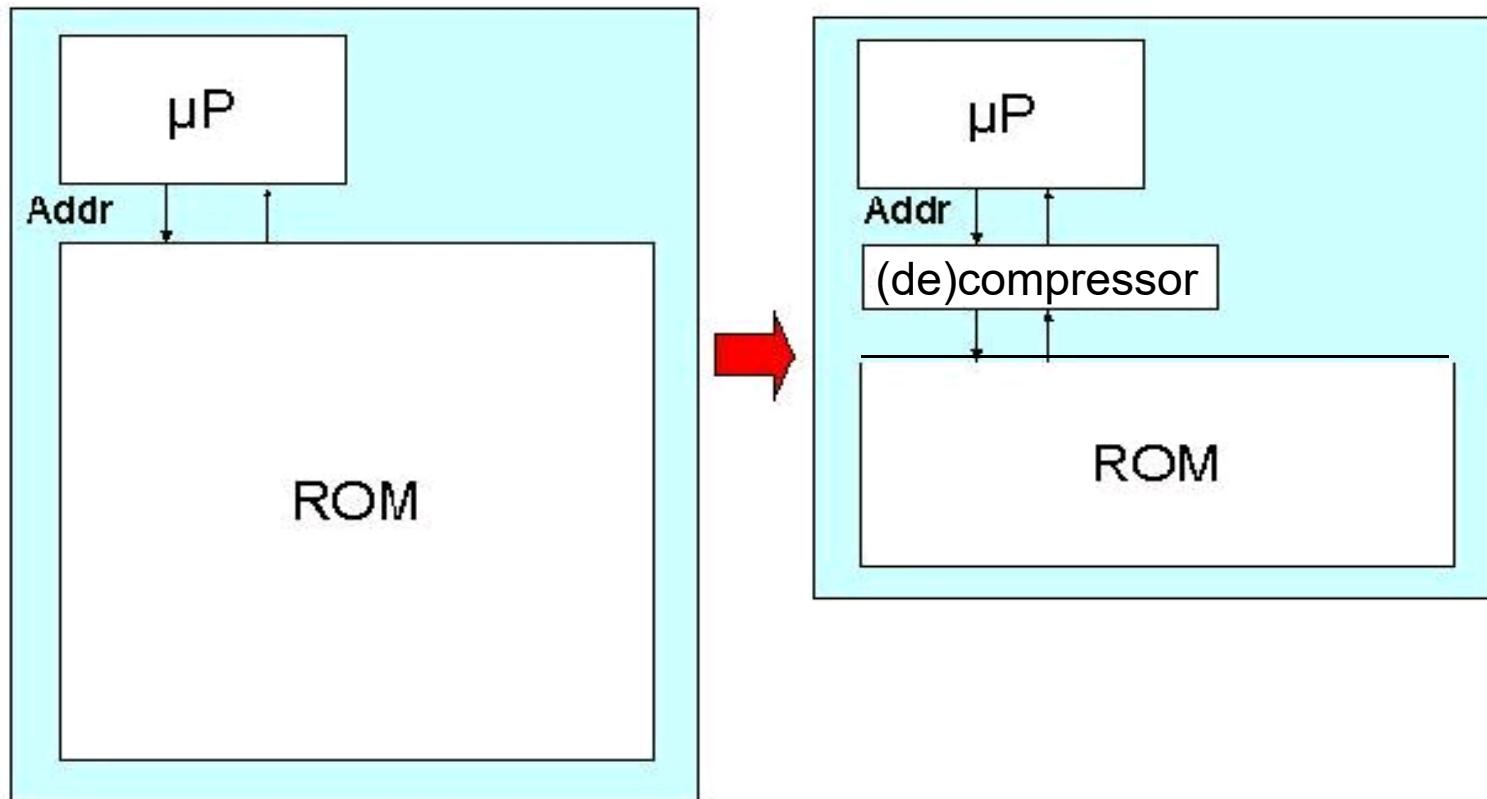
Architecture Specialization Techniques



A simple system design classification

Example: Code-size Efficiency

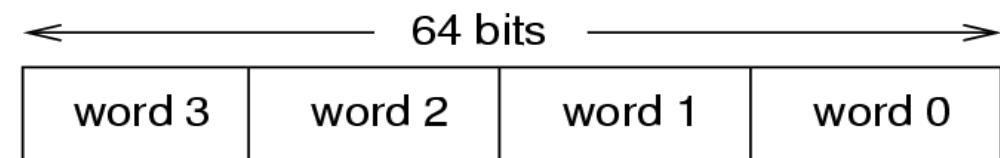
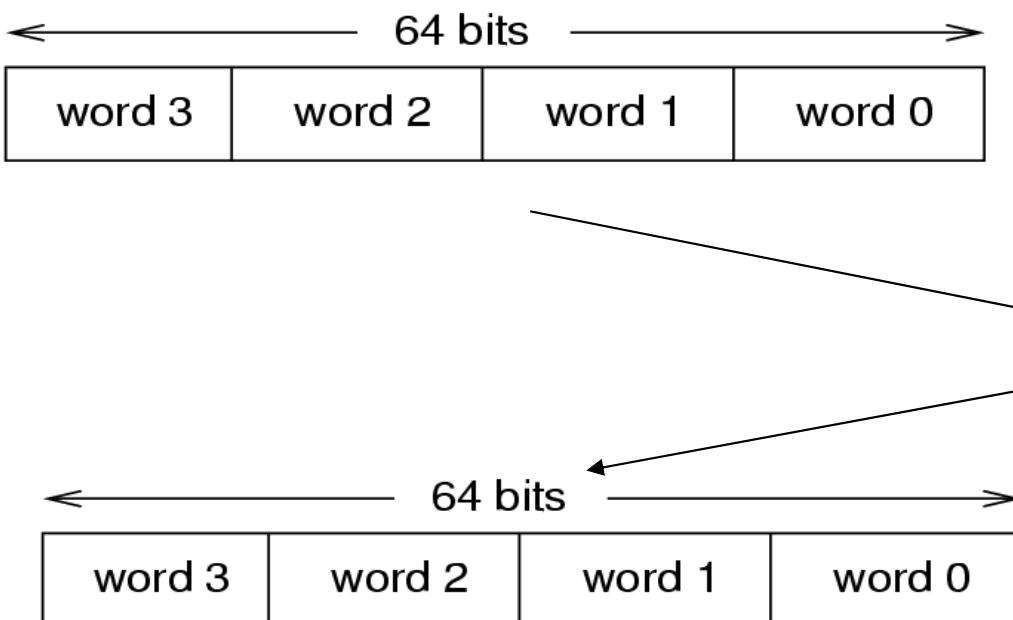
- ▶ RISC (Reduced Instruction Set Computers) machines designed for run-time-, not for code-size-efficiency.
- ▶ ***Compression techniques***: key idea



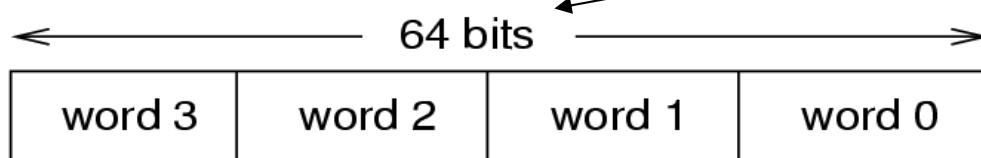
Example: Multimedia-Instructions

Multimedia instructions exploit that many registers, adders etc are quite wide (32/64 bit), whereas most multimedia data types are narrow (e.g. 8 bit per color, 16 bit per audio sample per channel)

☞ 2-8 values can be stored per register and added.



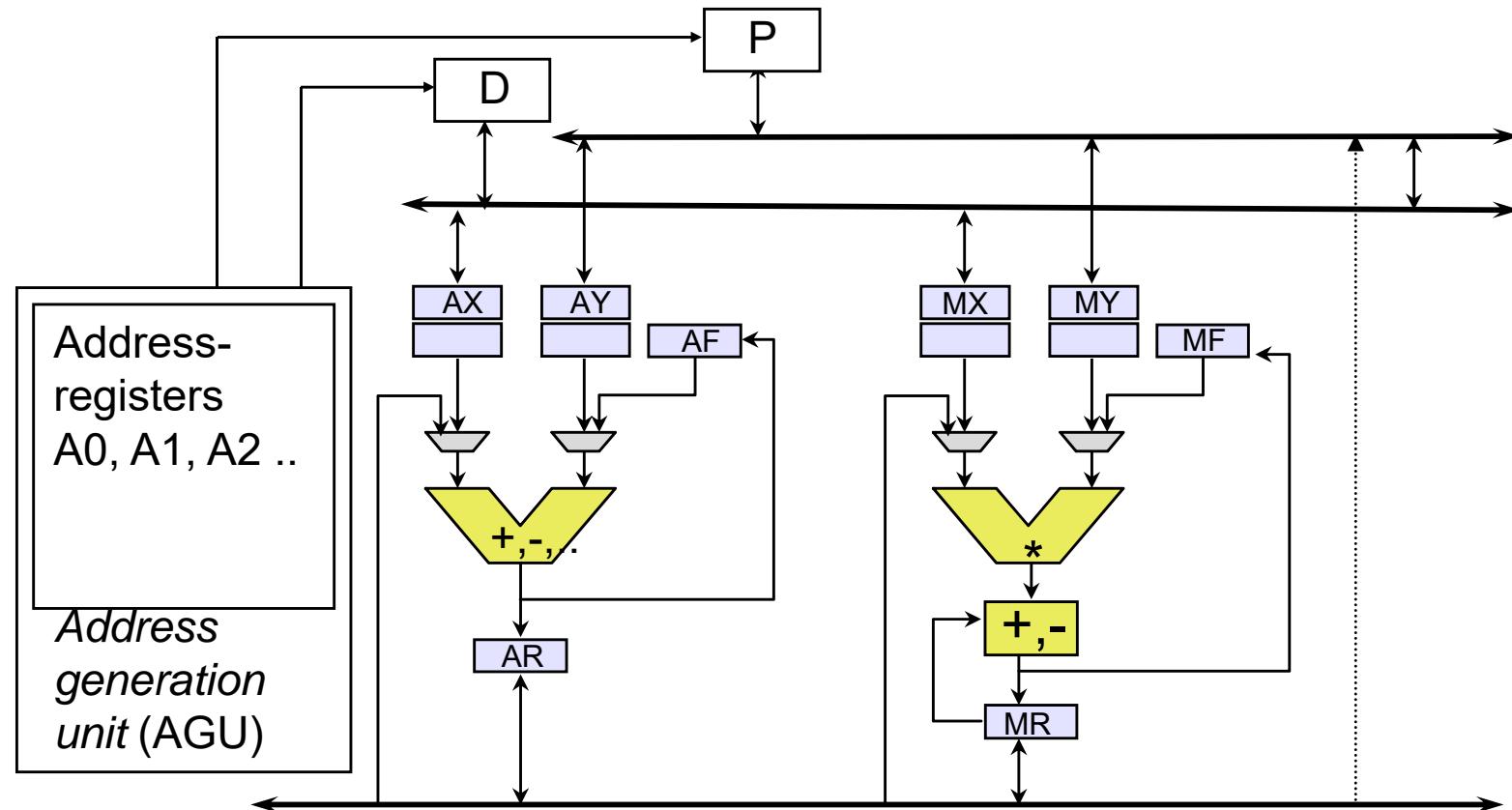
$$\begin{matrix} & & & + \\ & \searrow & \swarrow & \\ \end{matrix}$$



4 additions per instruction;
carry disabled at word
boundaries.

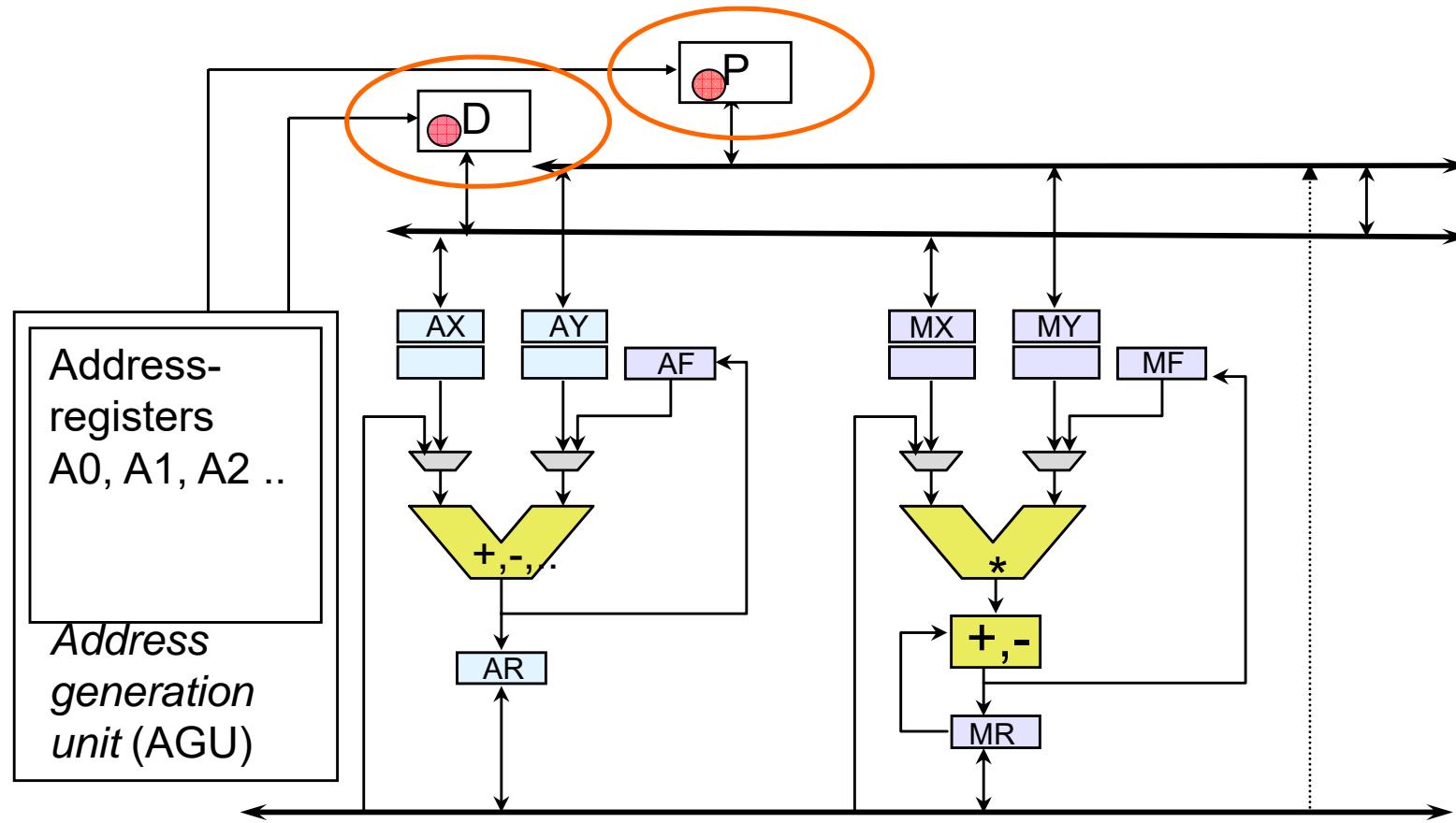
Example: Heterogeneous registers

Example (ADSP 210x):



Different functionality of registers AR, AX, AY, AF, MX, MY, MF, MR

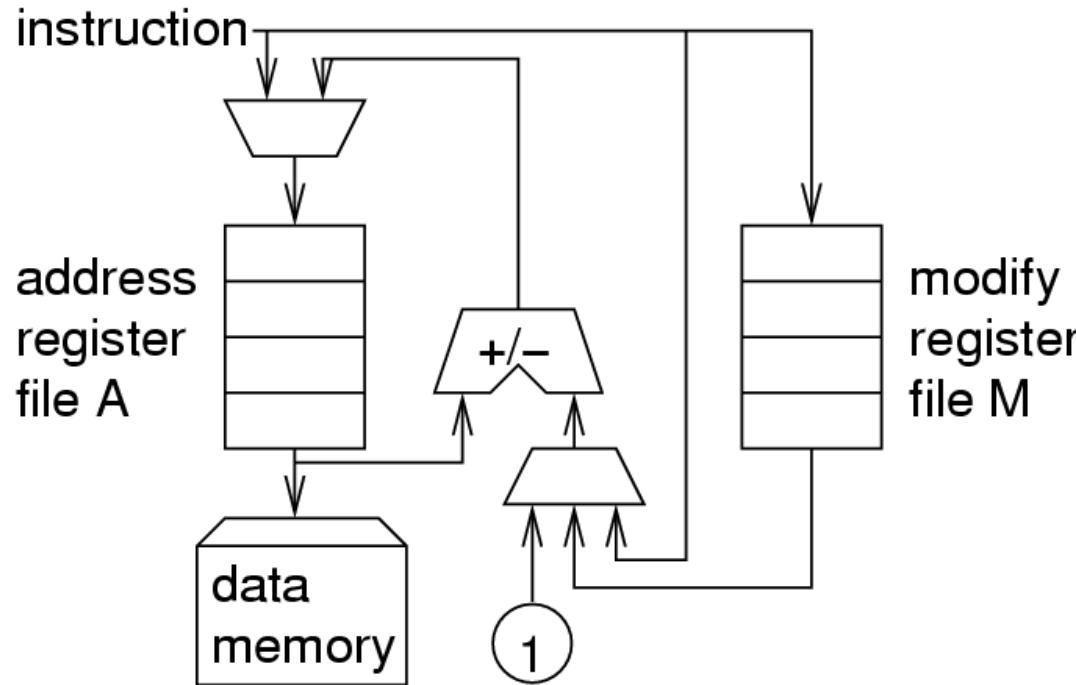
Example: Multiple memory banks or memories



Simplifies parallel fetches

Example: Address generation units

Example (ADSP 210x):



- Data memory can only be fetched with address contained in register file A, but its update can be done in parallel with operation in main data path (**takes effectively 0 time**).
- Register file A contains several precomputed addresses $A[i]$.
- There is another register file M that contains modification values $M[j]$.
- Possible updates:
 - $M[j] := \text{'immediate'}$
 - $A[i] := A[i] \pm M[j]$
 - $A[i] := A[i] \pm 1$
 - $A[i] := A[i] \pm \text{'immediate'}$
 - $A[i] := \text{'immediate'}$

Example: Modulo addressing

Modulo addressing:

$$A_{m++} \equiv A_m := (A_m + 1) \bmod n$$

(implements ring or circular buffer in memory)

$x[t]$: value accessed at time t

\dots

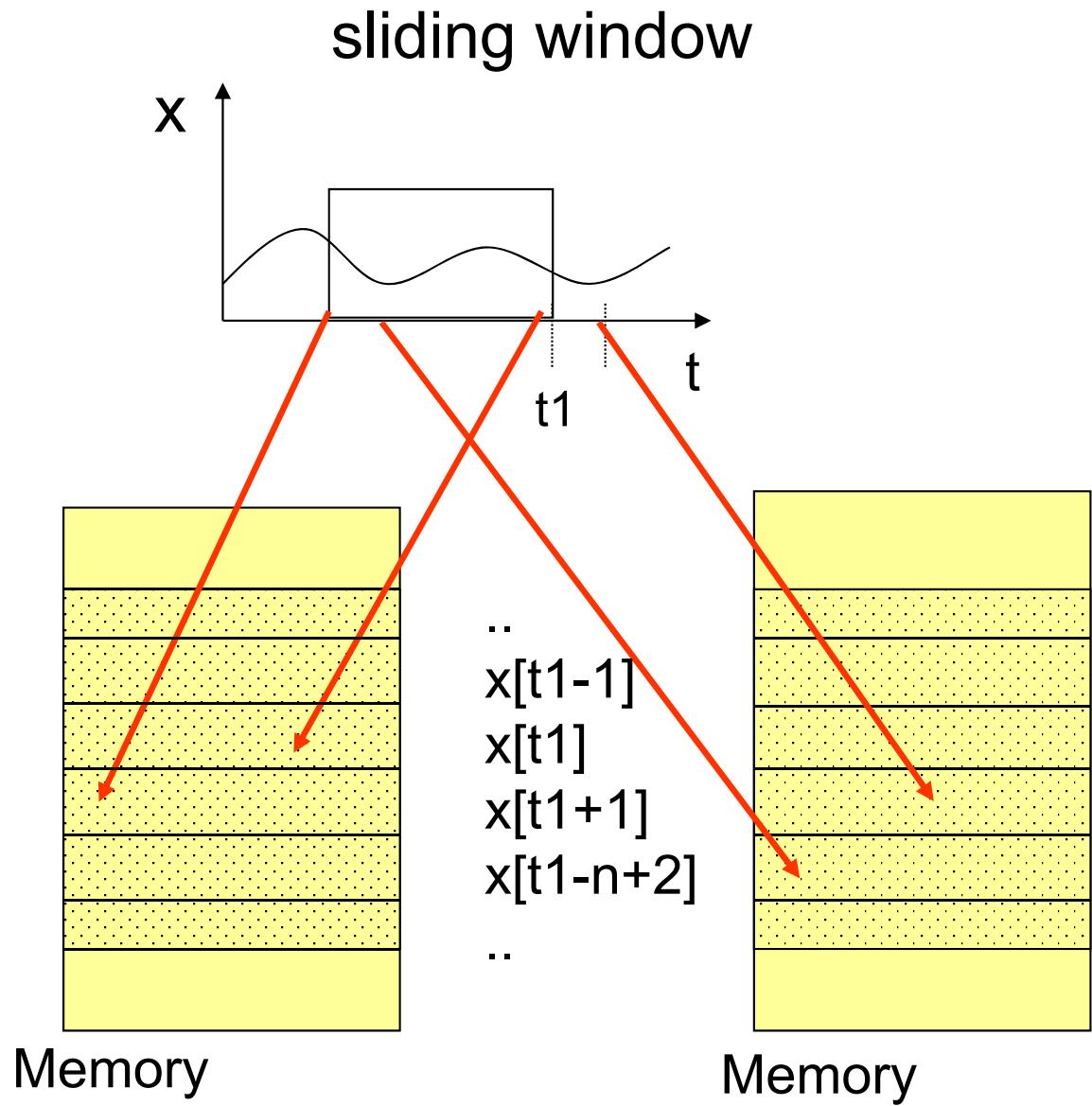
$x[t_1-1]$

$x[t_1]$

$x[t_1-n+1]$

$x[t_1-n+2]$

\dots

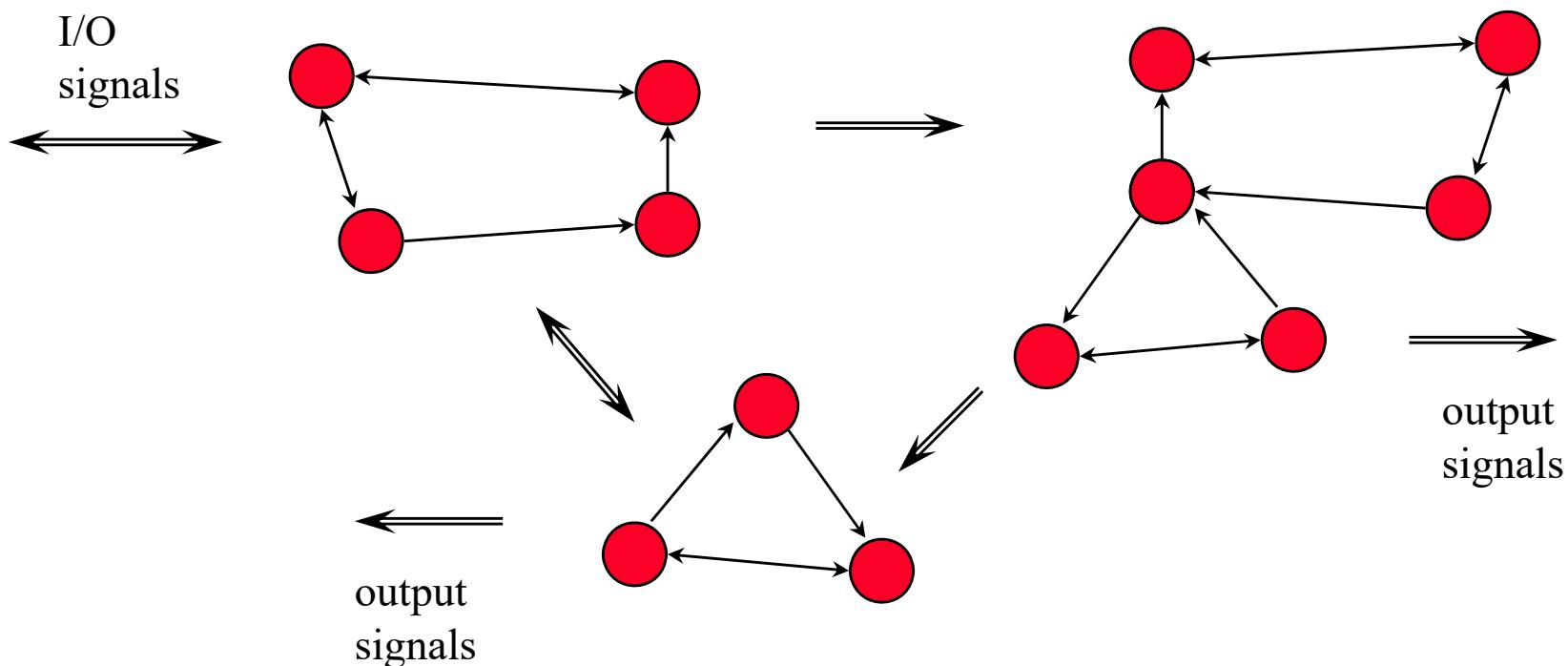


Topics

- ▶ System Specialization
- ▶ Application Specific Instruction Sets
 - *Micro Controller*
 - Digital Signal Processors and VLIW
- ▶ Programmable Hardware
- ▶ ASICs
- ▶ System-on-Chip

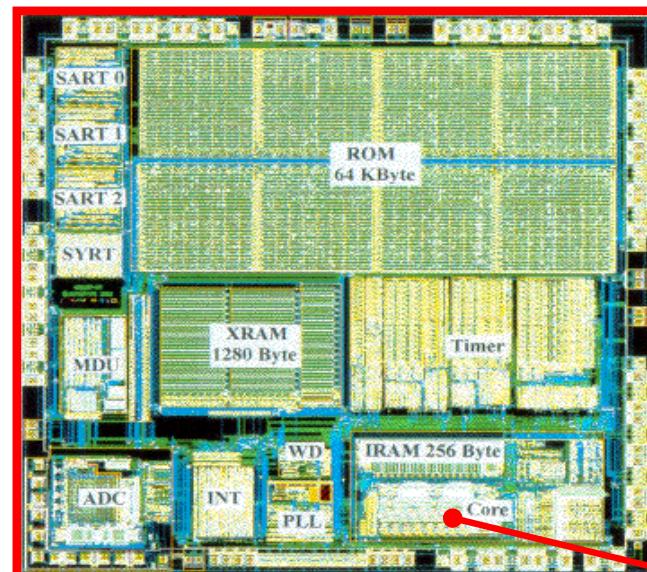
Control Dominated Systems

- ▶ Reactive systems with ***event driven behavior***
- ▶ Underlying semantics of system description (“input model of computation”) typically (coupled) Finite State Machines or Petri Nets



Microcontroller

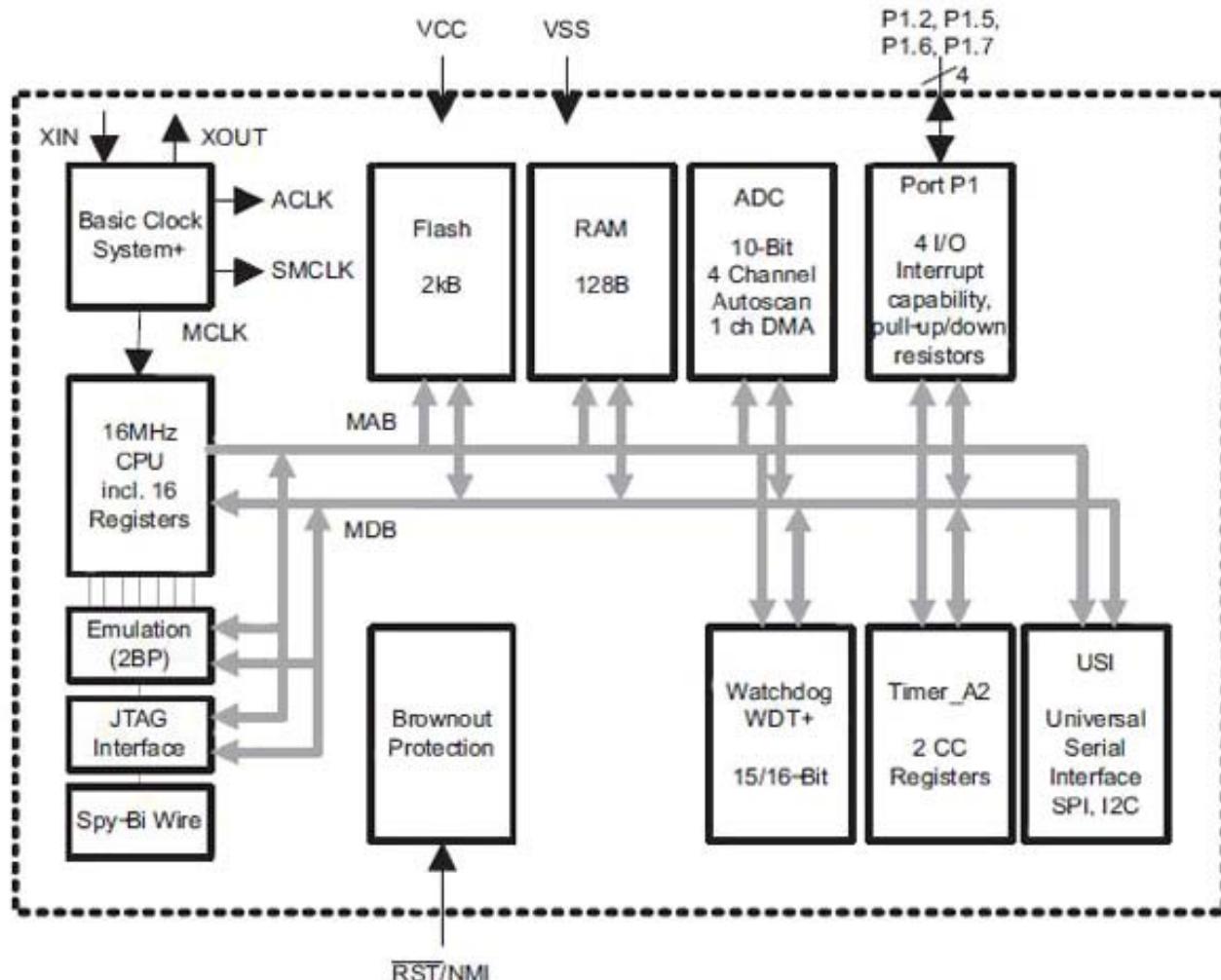
- ▶ control-dominant applications
 - supports process scheduling and synchronization
 - preemption (interrupt), context switch
 - short latency times
- ▶ low power consumption
- ▶ peripheral units often integrated
- ▶ suited for real-time applications



8051 core

SIECO51 (Siemens)

Microcontroller as a System-on-Chip



MSP 430 RISC Processor (Microchip)

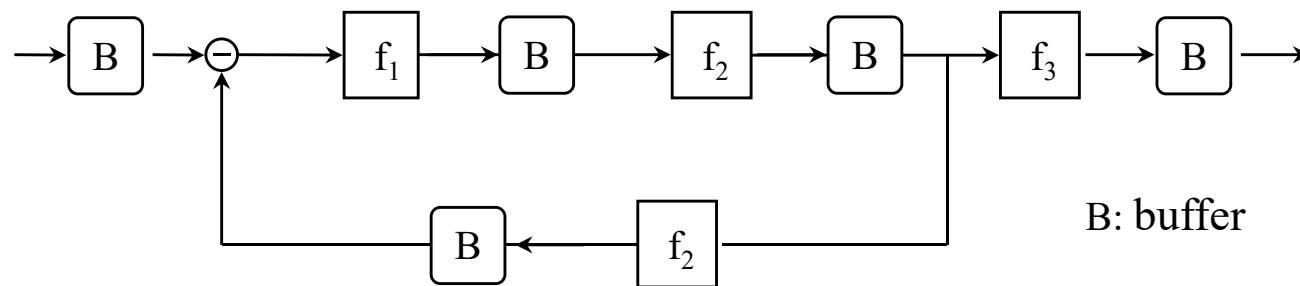
- complete system
- timers
- I²C-bus and par./ser. interfaces for communication
- A/D converter
- watchdog (SW activity timeout): safety
- on-chip memory (volatile/non-volatile)
- interrupt controller

Topics

- ▶ System Specialization
- ▶ Application Specific Instruction Sets
 - Micro Controller
 - *Digital Signal Processors and VLIW*
- ▶ Programmable Hardware
- ▶ ASICs
- ▶ System-on-Chip

Data Dominated Systems

- ▶ ***Streaming oriented systems*** with mostly periodic behavior
- ▶ Underlying semantics of input description e.g. ***flow graphs*** (“input model of computation”)

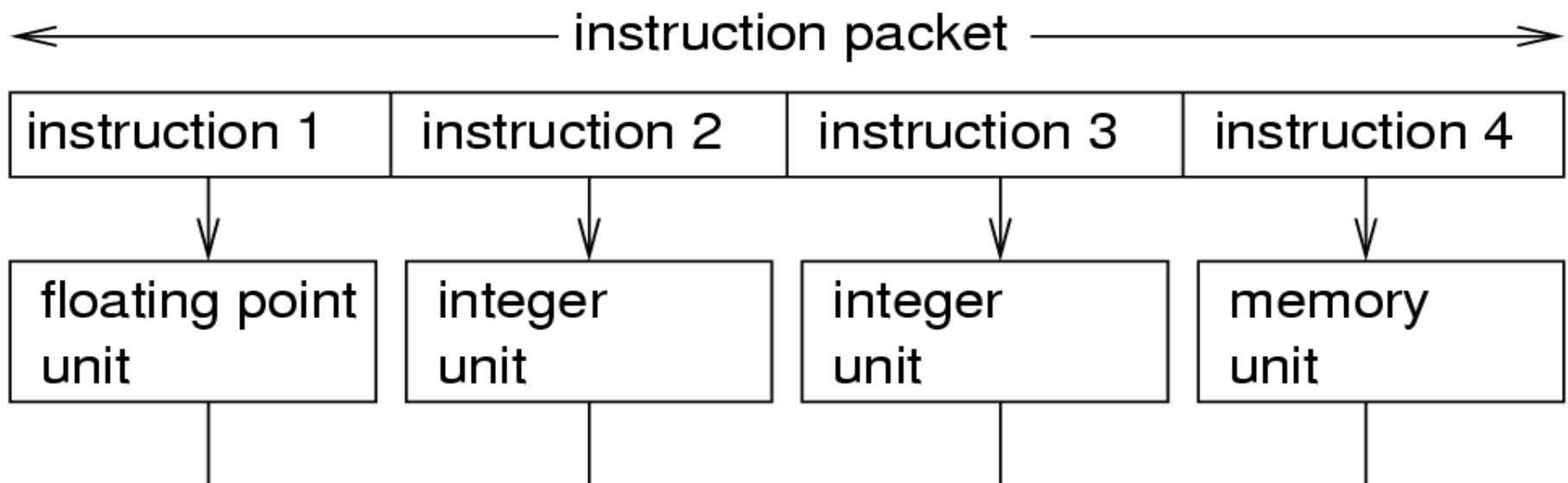


- ▶ ***Application examples:*** signal processing, control engineering

Very Long Instruction Word (VLIW)

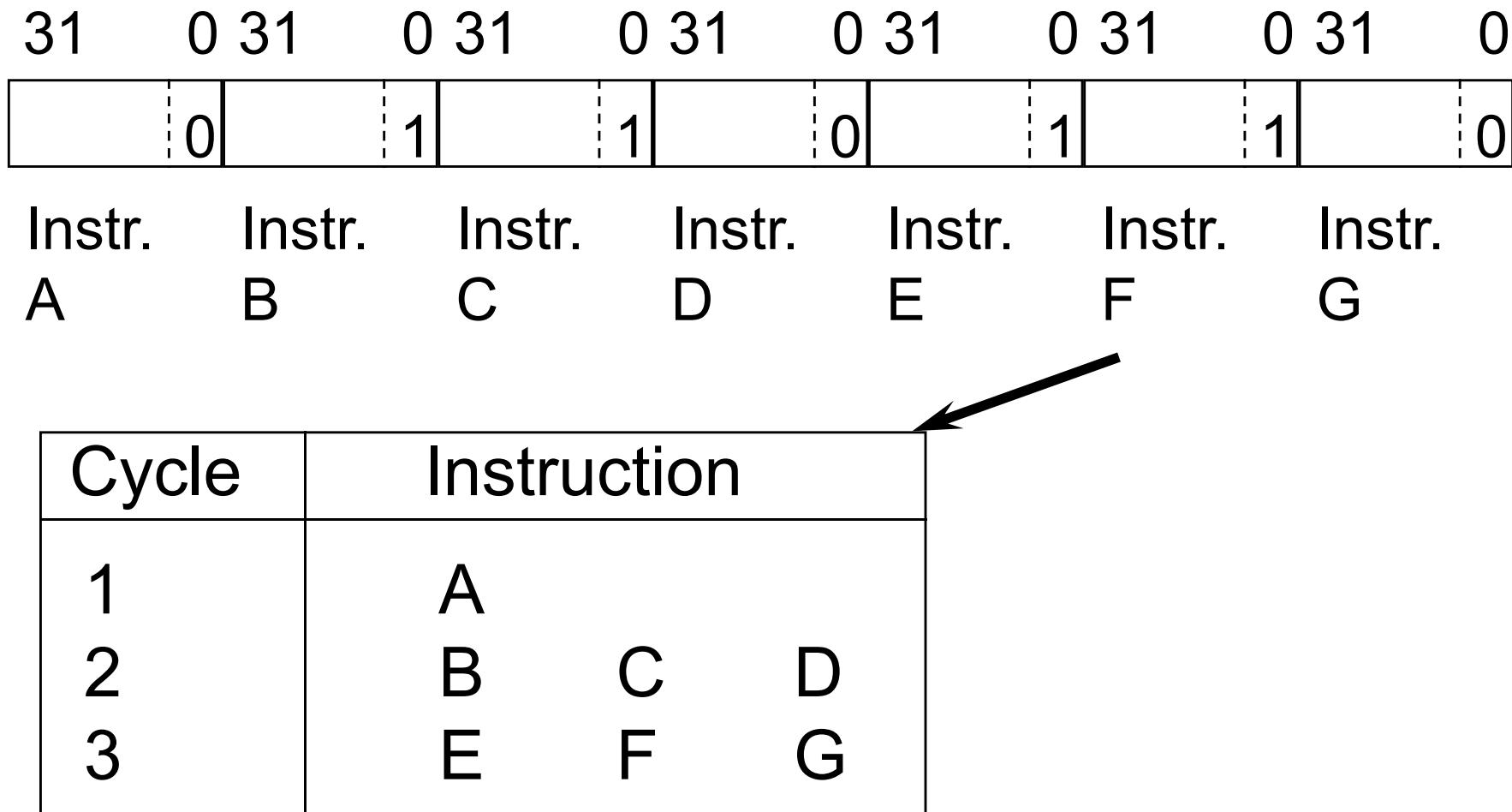
Key idea: detection of possible parallelism to be done by compiler, not by hardware at run-time (inefficient).

VLIW: parallel operations (instructions) encoded in one long word (instruction packet), each instruction controlling one functional unit. E.g.:



Explicit Parallelism Instruction Computers

The TMS320C62xx VLIW Processor as an example of EPIC:

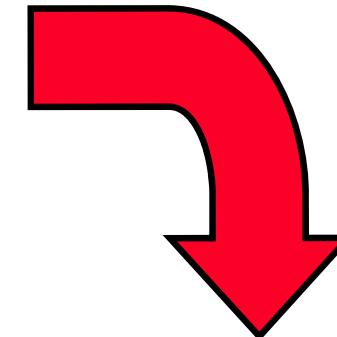


MAC (multiply & accumulate)

```
sum = 0.0;  
for (i=0; i<N; i++)  
    sum = sum + a[i]*b[i];
```

zero-overhead loop
(repeat next instruction N times)

MAC - Instruktion



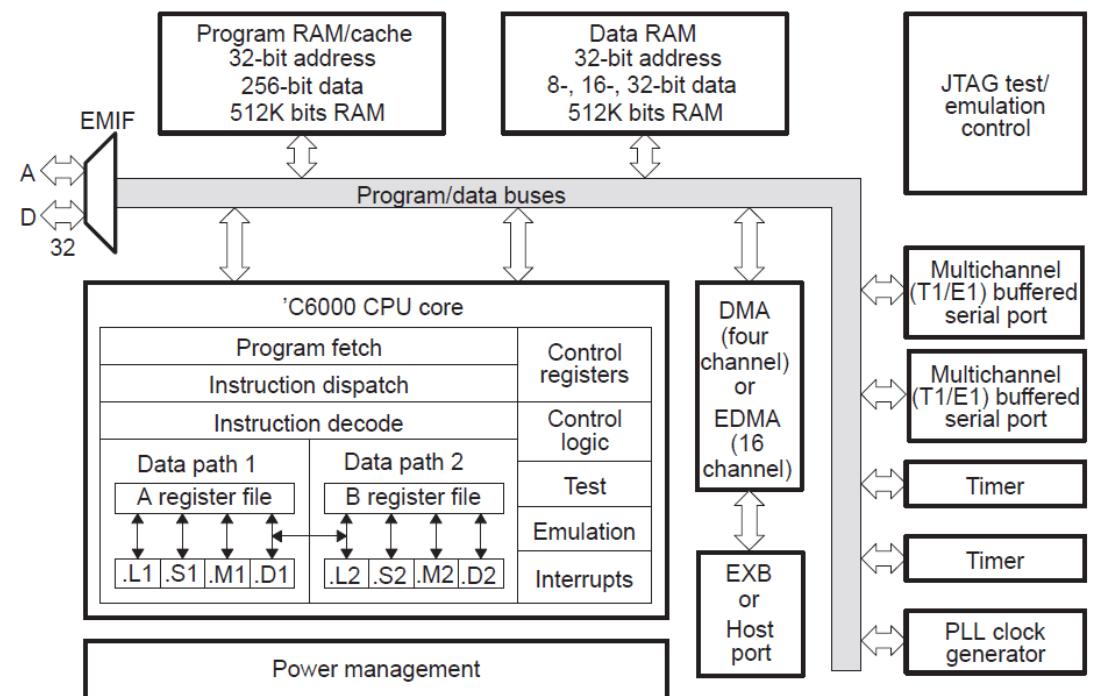
LDF	0, R0
LDF	0, R1
RPTS	N
MPYF3	* (AR0)++, * (AR1)++, R0
ADDF3	R0, R1, R1

TMS320C3x Assembler
(Texas Instruments)

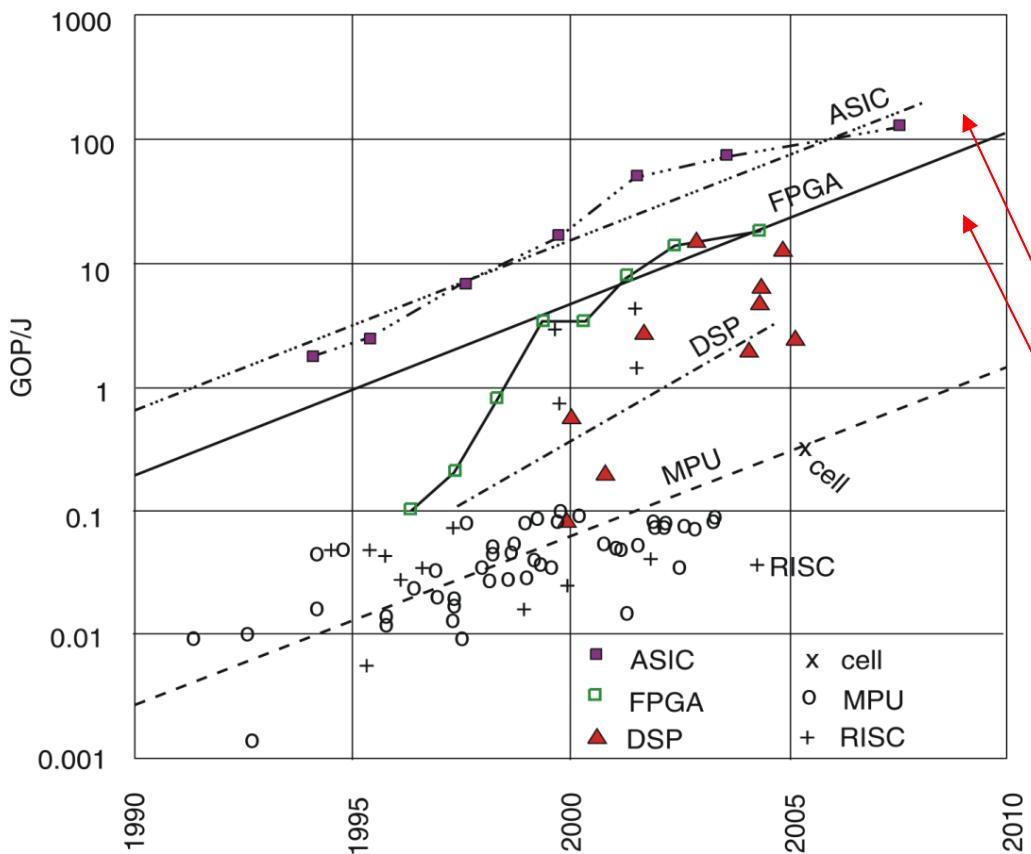
Digital Signal Processor

- ▶ optimized for data-flow applications
- ▶ suited for simple control flow
- ▶ parallel hardware units (VLIW)
- ▶ specialized instruction set
- ▶ high data throughput
- ▶ zero-overhead loops
- ▶ specialized memory
- ▶ suited for real-time applications

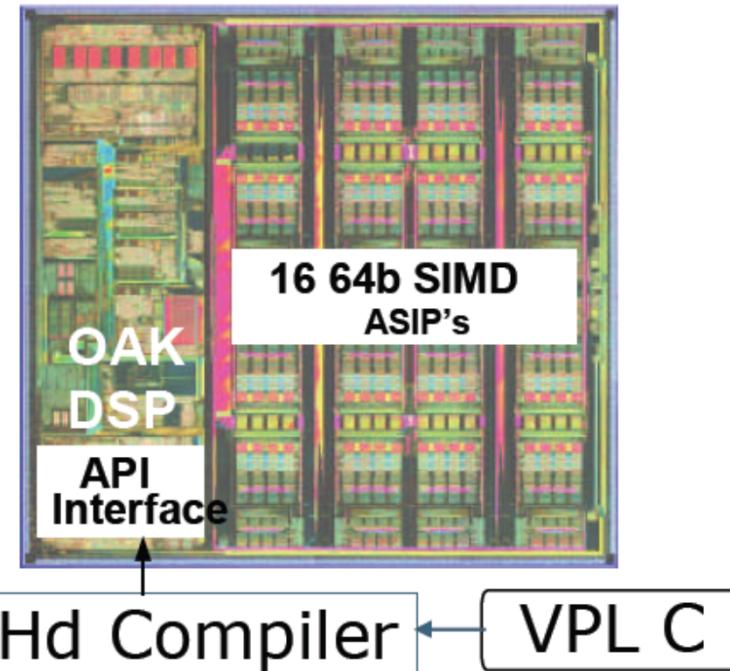
Figure 2-1. TMS320C62x/C67x Block Diagram



Example Infineon

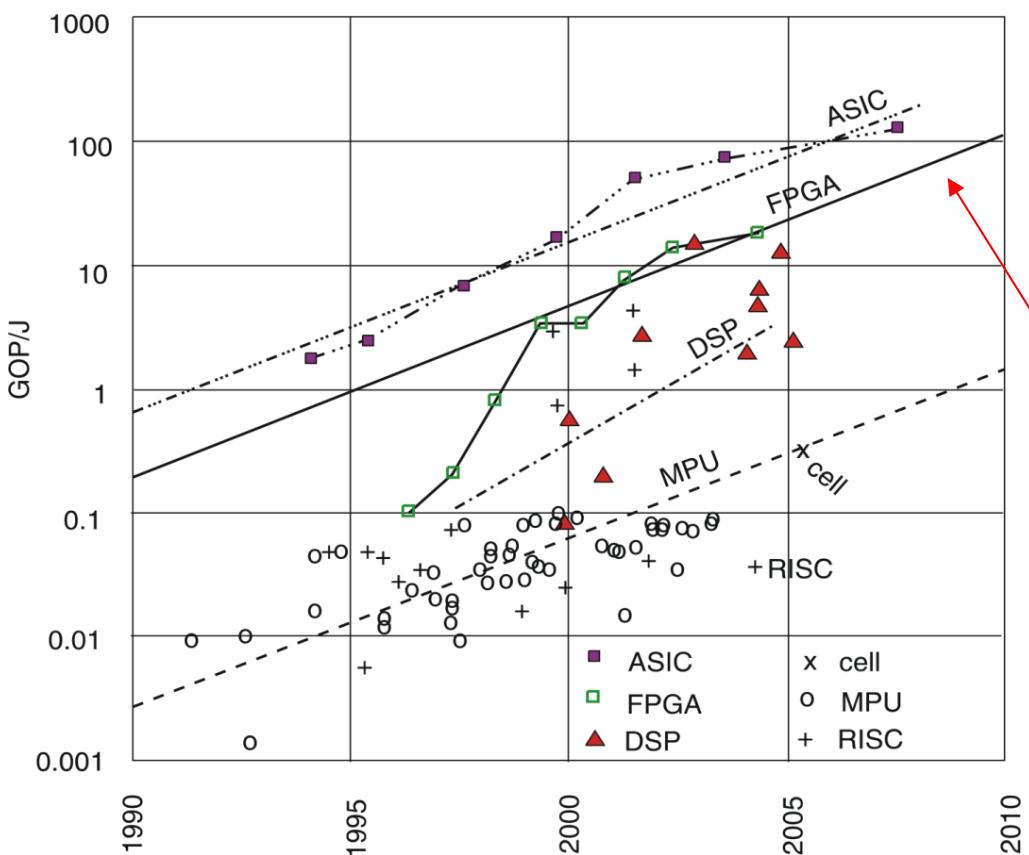


**VIP for car mirrors
Infineon**

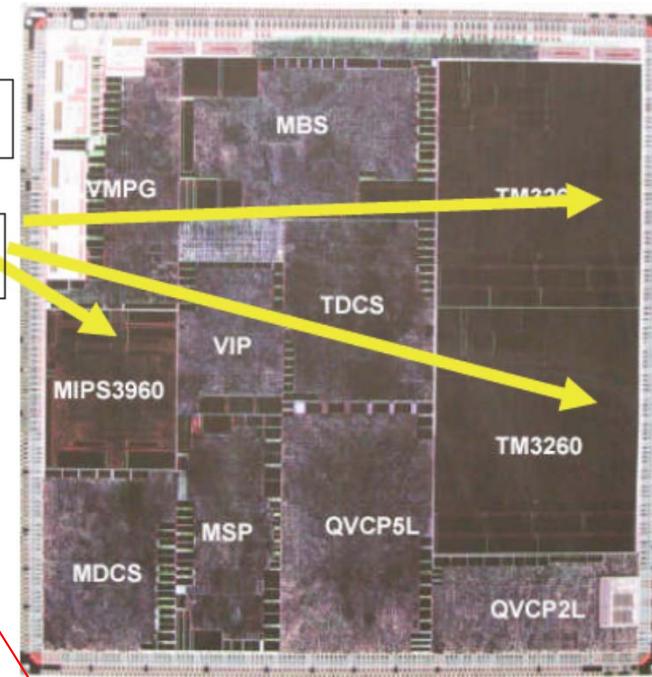


200MHz , 0.76 Watt
100Gops @ 8b
25Gops @ 32b

Example NXP Trimedia VLIW



Nexperia Digital Video Platform
NXP



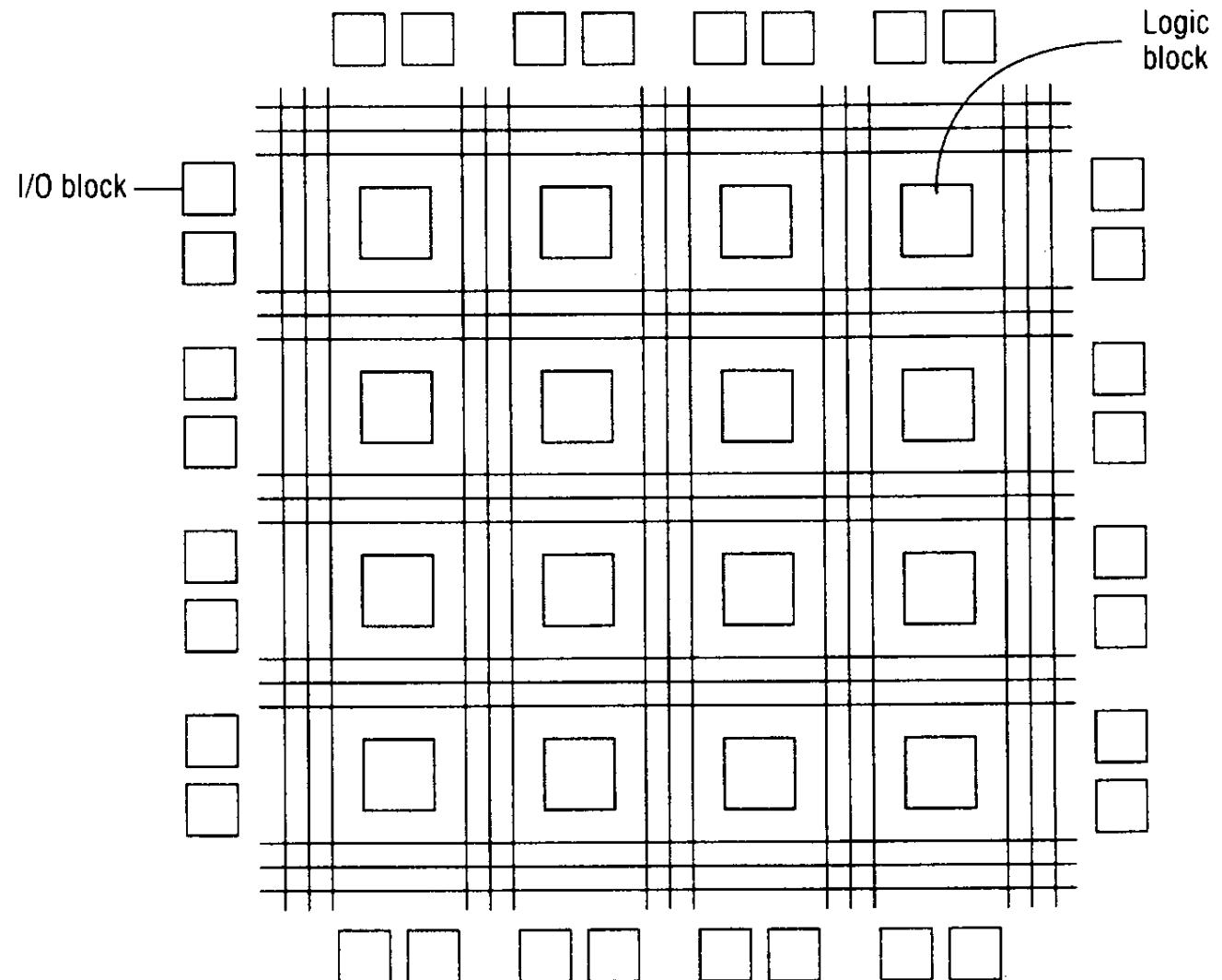
**1 MIPS, 2 Trimedia
60 coproc, 250 RAM's
266MHz, 1.5 watt 100 Gops**

Topics

- ▶ System Specialization
- ▶ Application Specific Instruction Sets
 - Micro Controller
 - Digital Signal Processors and VLIW
- ▶ ***Programmable Hardware***
- ▶ ASICs
- ▶ System-on-Chip

FPGA – Basic Structure

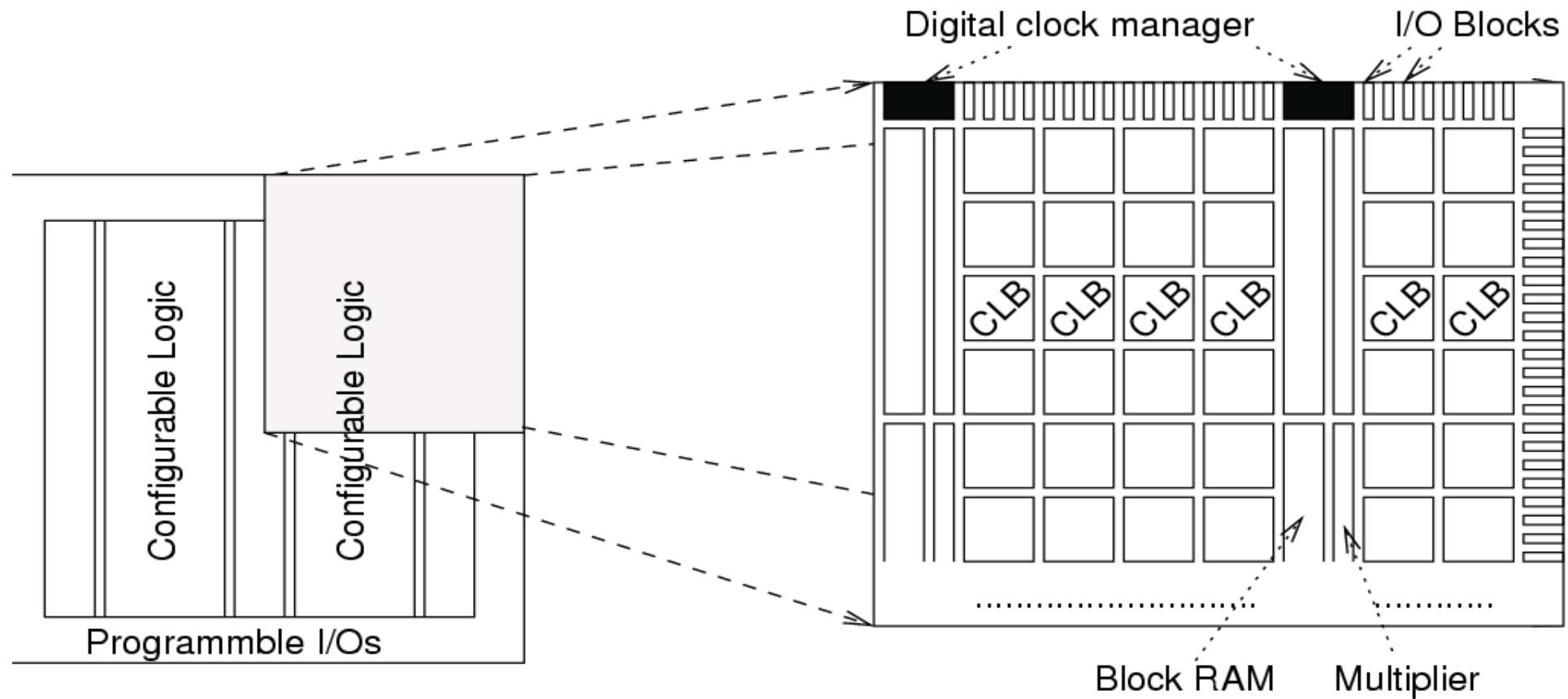
- ▶ Logic Units
- ▶ I/O Units
- ▶ Connections



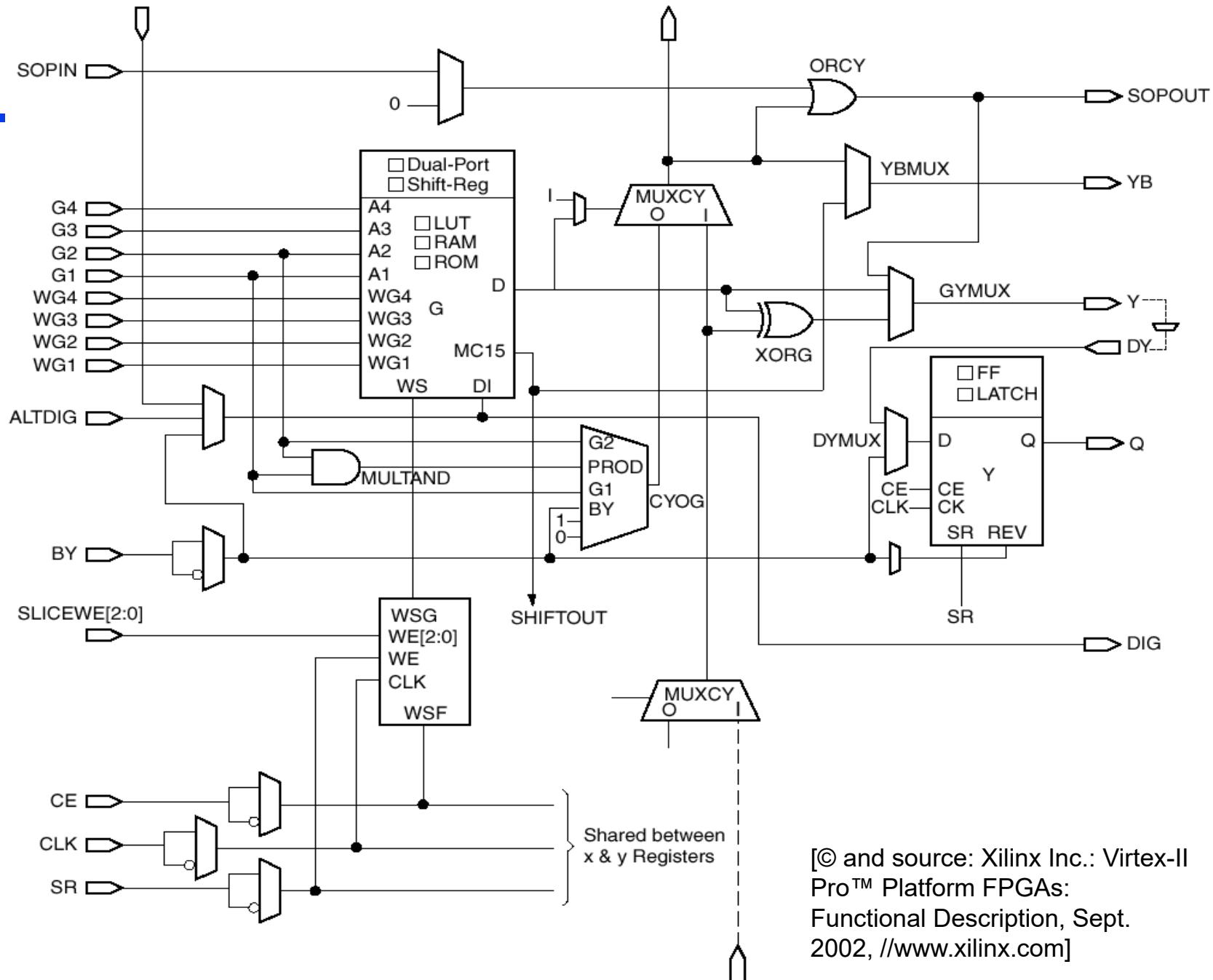
FPGA - Classification

- ▶ ***Granularity of logic units:***
 - Gate, tables, memory, functional blocks (ALU, control, data path, processor)
- ▶ ***Communication network:***
 - Crossbar, hierarchical mesh, tree
- ▶ ***Reconfiguration:***
 - fixed at production time, once at design time, dynamic during run-time

Floor-plan of VIRTEX II FPGAs



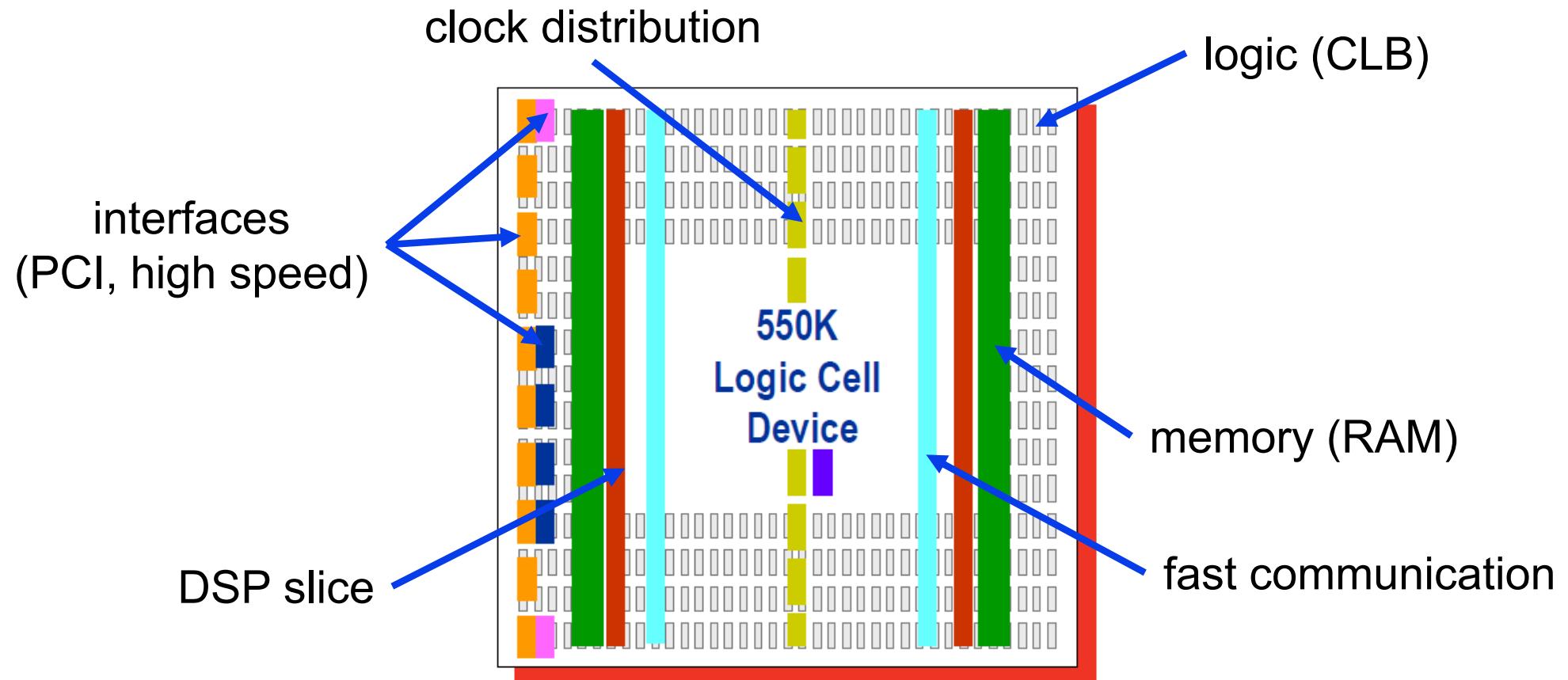
Virtex Logic Cell



[© and source: Xilinx Inc.: Virtex-II Pro™ Platform FPGAs:
Functional Description, Sept. 2002, //www.xilinx.com]

Example Virtex-6

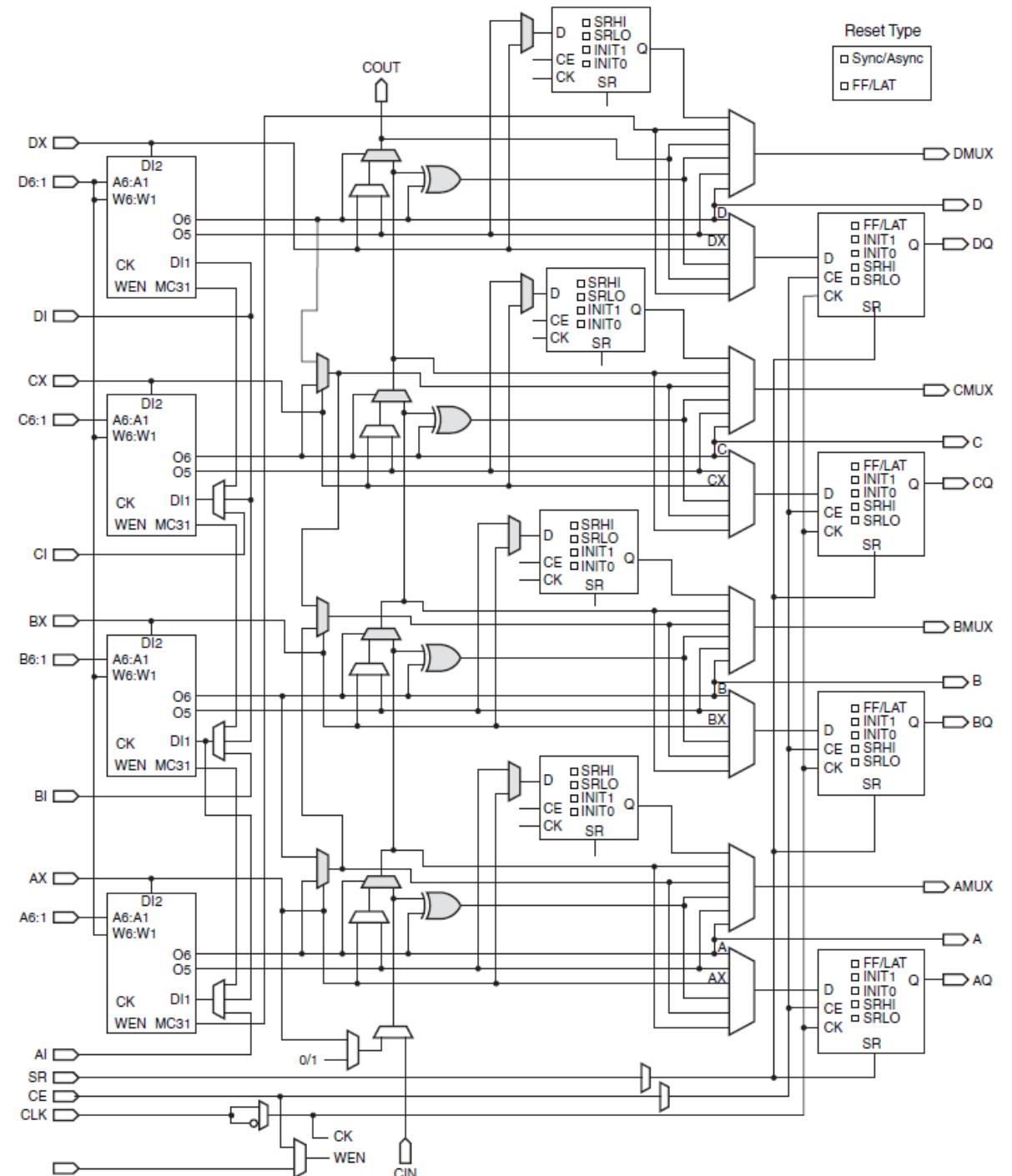
- ▶ Combination of flexibility (CLB's), Integration and performance (heterogeneity of hard-IP Blocks)



XILINX Virtex UltraScale

Effective LEs (K)	3,435
Logic Cells (K)	2,863
UltraRAM (Mb)	432.0
Block RAM (Mb)	94.5
DSP Slices	11,904
I/O Pins	832

Virtex-6 CLB Slice



Topics

- ▶ System Specialization
- ▶ Application Specific Instruction Sets
 - Micro Controller
 - Digital Signal Processors and VLIW
- ▶ Programmable Hardware
- ▶ **ASICs**
- ▶ System-on-Chip

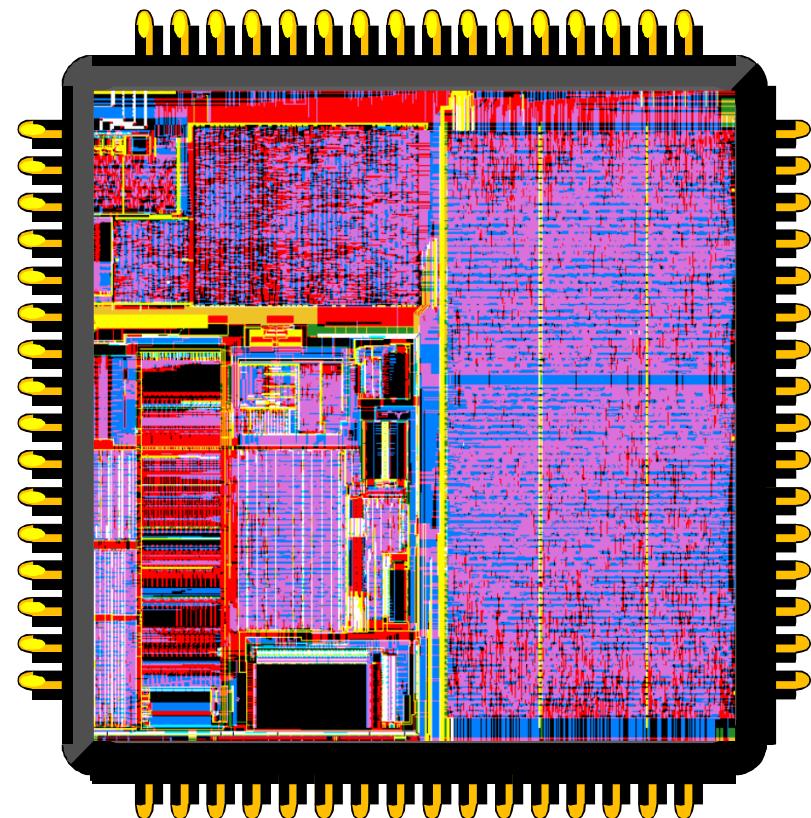
Application Specific Circuits (ASICs)

Custom-designed circuits necessary

- if ultimate speed or
- energy efficiency is the goal and
- large numbers can be sold.

Approach suffers from

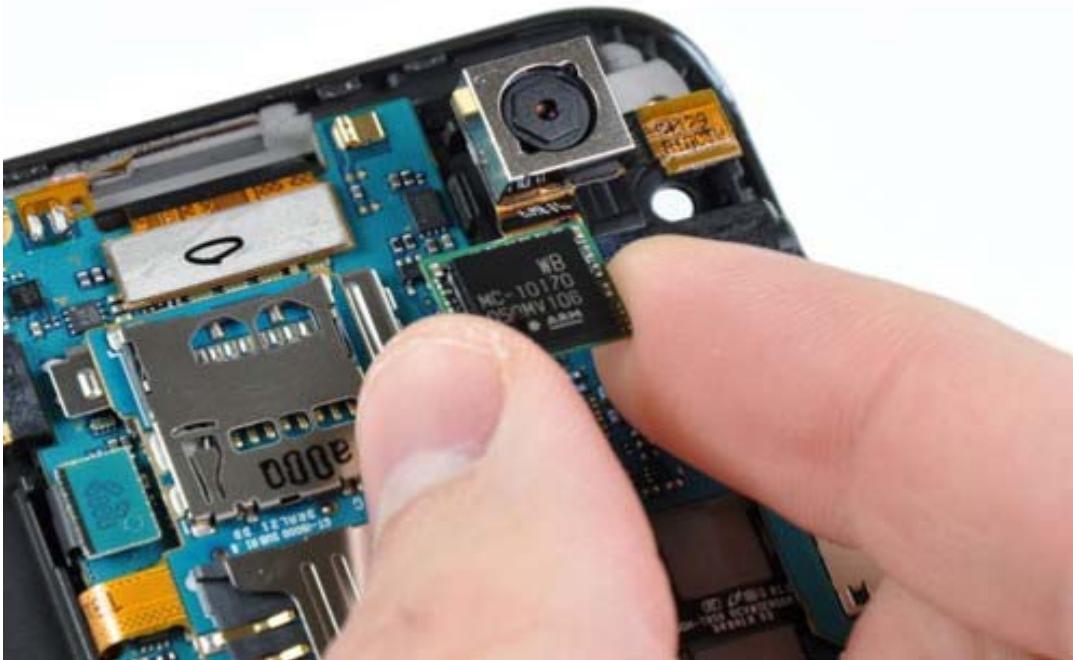
- long design times,
- lack of flexibility
(changing standards) and
- high costs
(e.g. Mill. \$ mask costs).



Topics

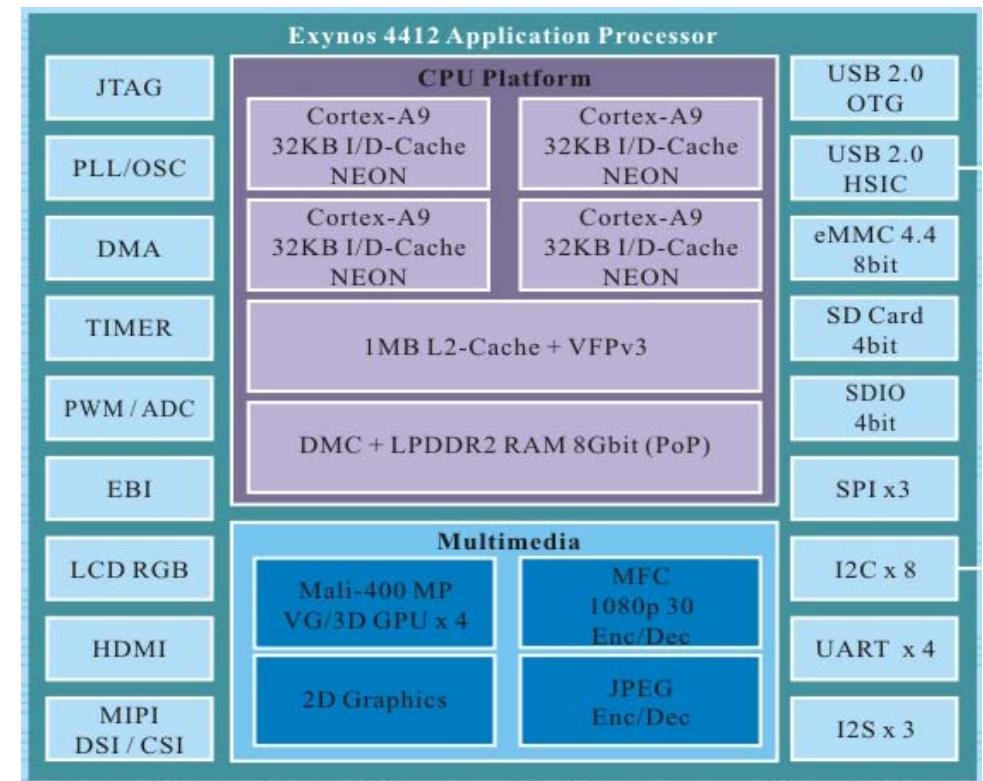
- ▶ System Specialization
- ▶ Application Specific Instruction Sets
 - Micro Controller
 - Digital Signal Processors and VLIW
- ▶ Programmable Hardware
- ▶ ASICs
- ▶ ***System-on-Chip***

System-on-Chip



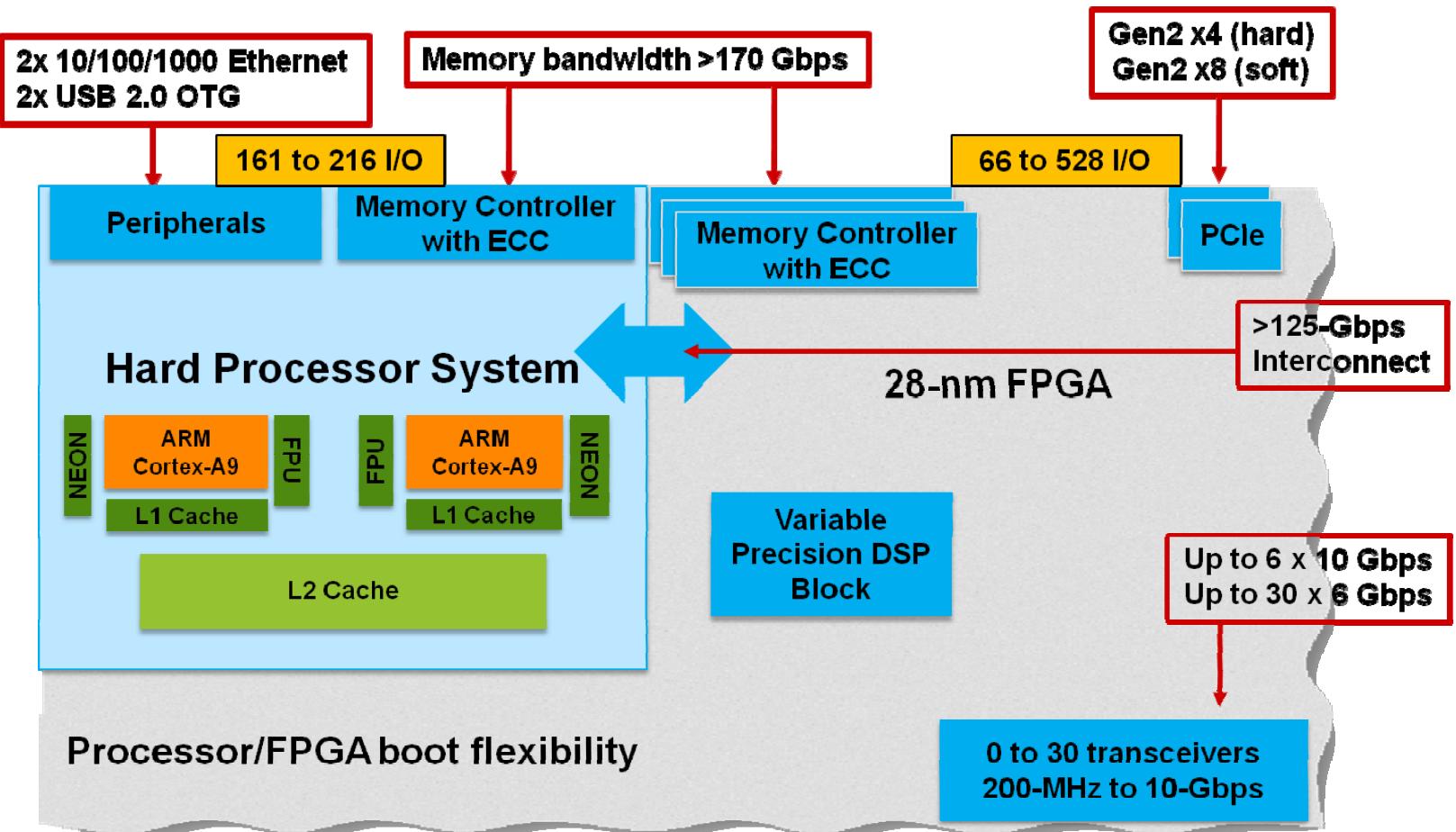
Samsung Galaxy Note II

- Eynos 4412 System on a Chip (SoC)
- ARM Cortex-A9 processing core
- 32 nanometer: transistor gate width
- Four processing cores



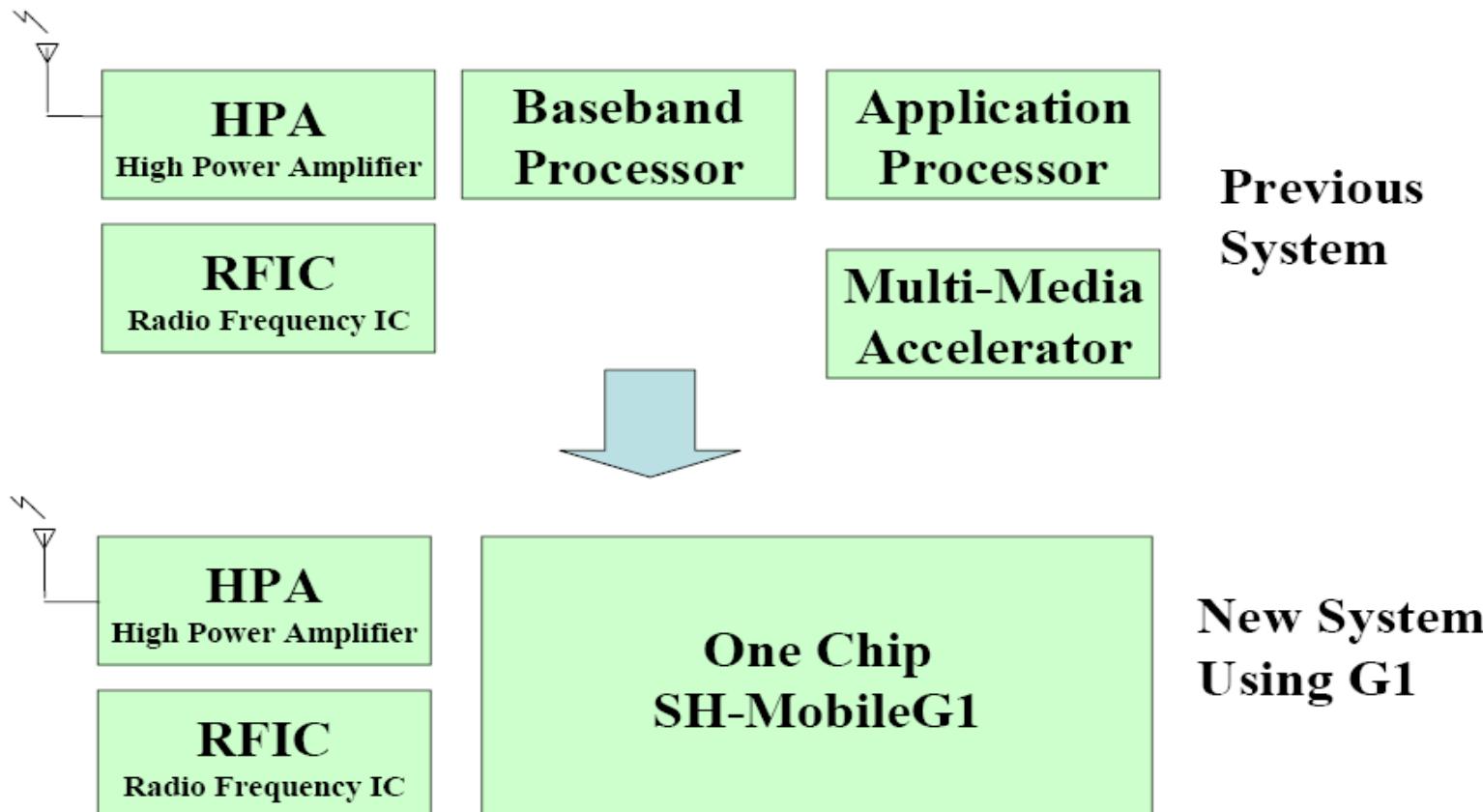
Configurable System-On-Chip

Example:
Altera's SoC
FPGA integrates a
dual-core ARM
Cortex-A9
processor system
with a low power
FPGA fabrics



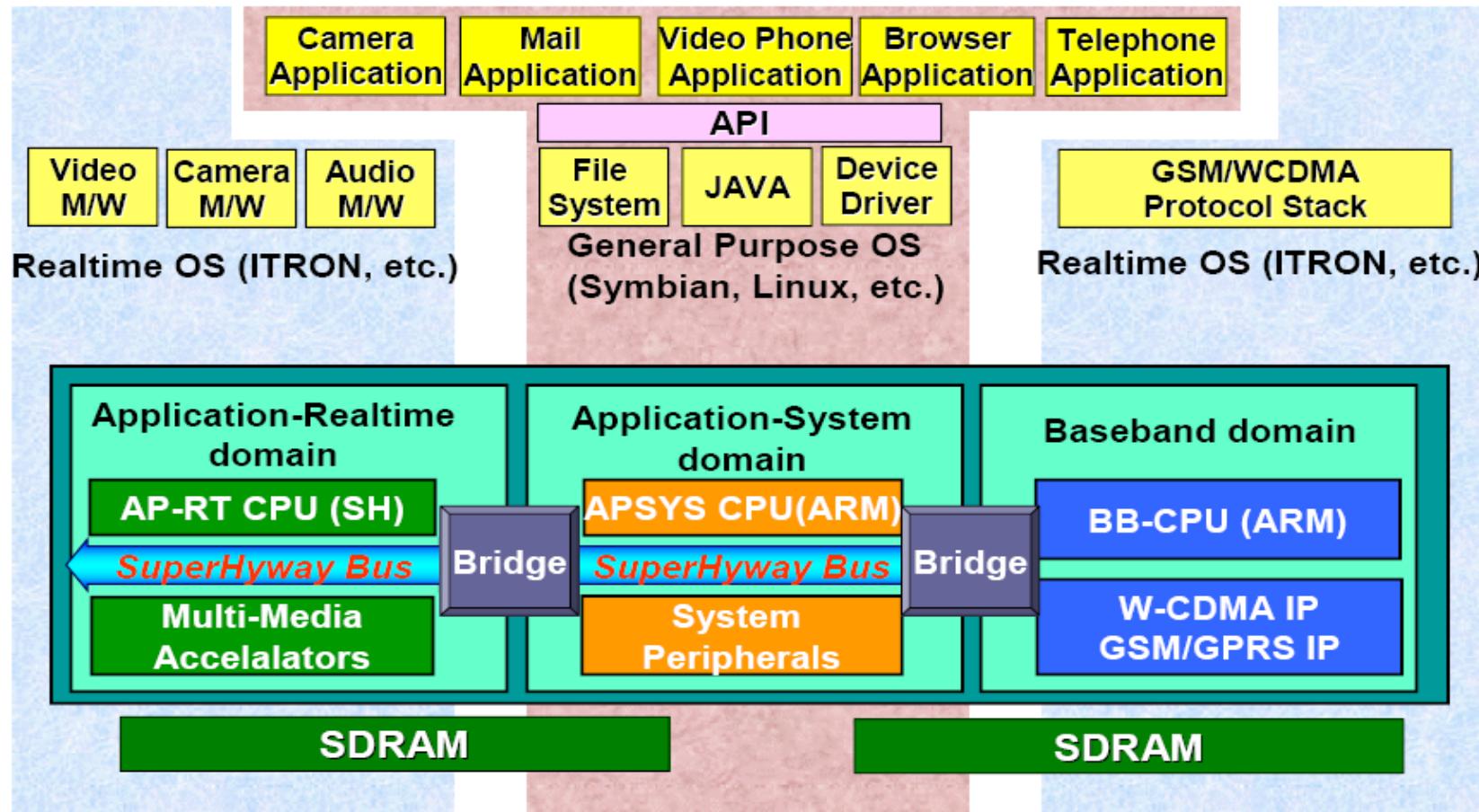
Trend: multiprocessor systems-on-a-chip (MPSoCs)

3G Multi-Media Cellular Phone System



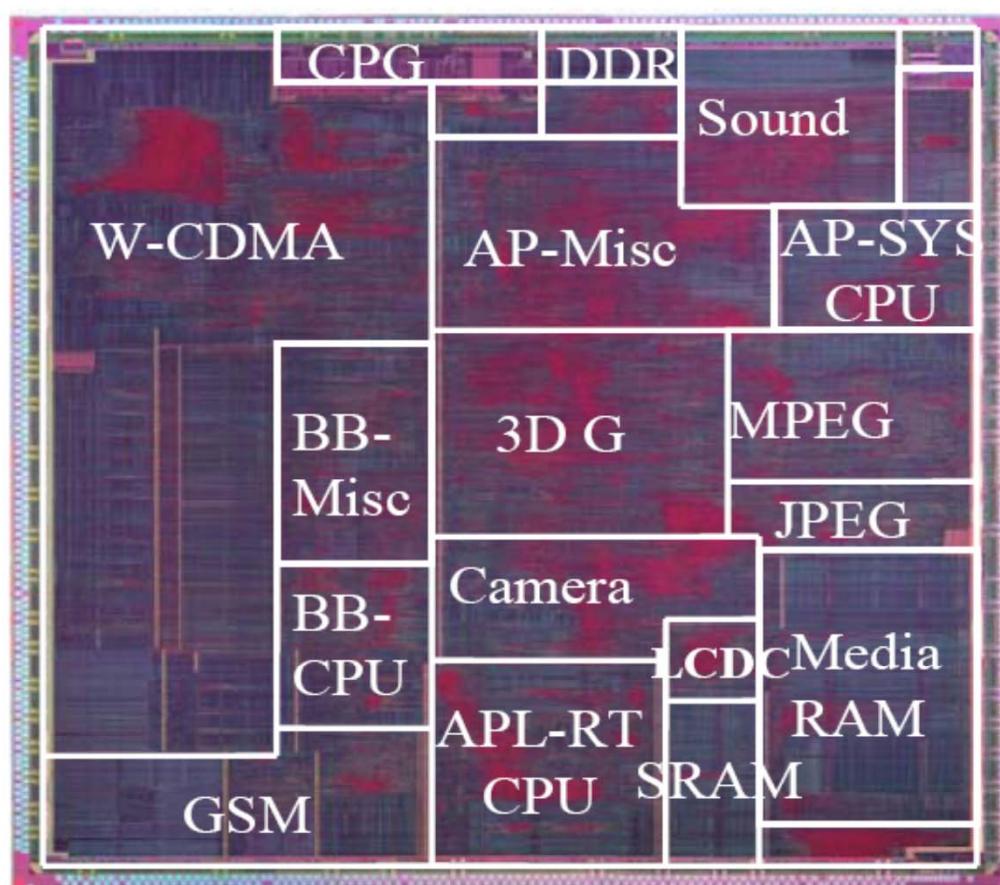
Multiprocessor systems-on-a-chip (MPSoCs)

A Sample of System Architecture using G1



Multiprocessor systems-on-a-chip (MPSoCs)

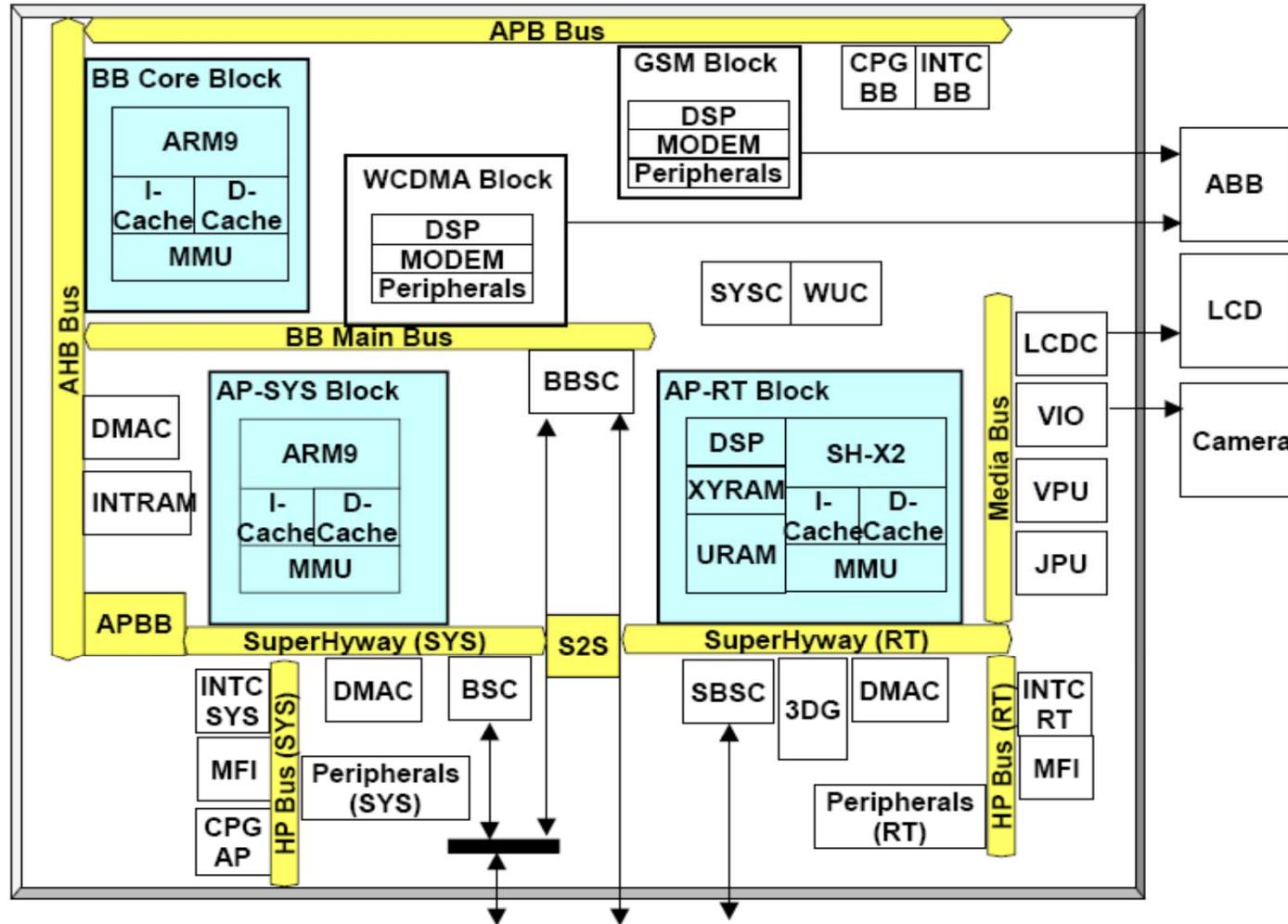
SH-MobileG1: Chip Overview



Die size	11.15mm x 11.15mm
Process	90nm LP 8M(7Cu+1Al) CMOS dual-Vth
Supply voltage	1.2V(internal), 1.8/2.5/3.3V(I/O)
# of TRs, gate, memory	181M TRs, 13.5M Gate 20.2 Mbit mem

Multiprocessor systems-on-a-chip (MPSoCs)

G1 Module Diagram

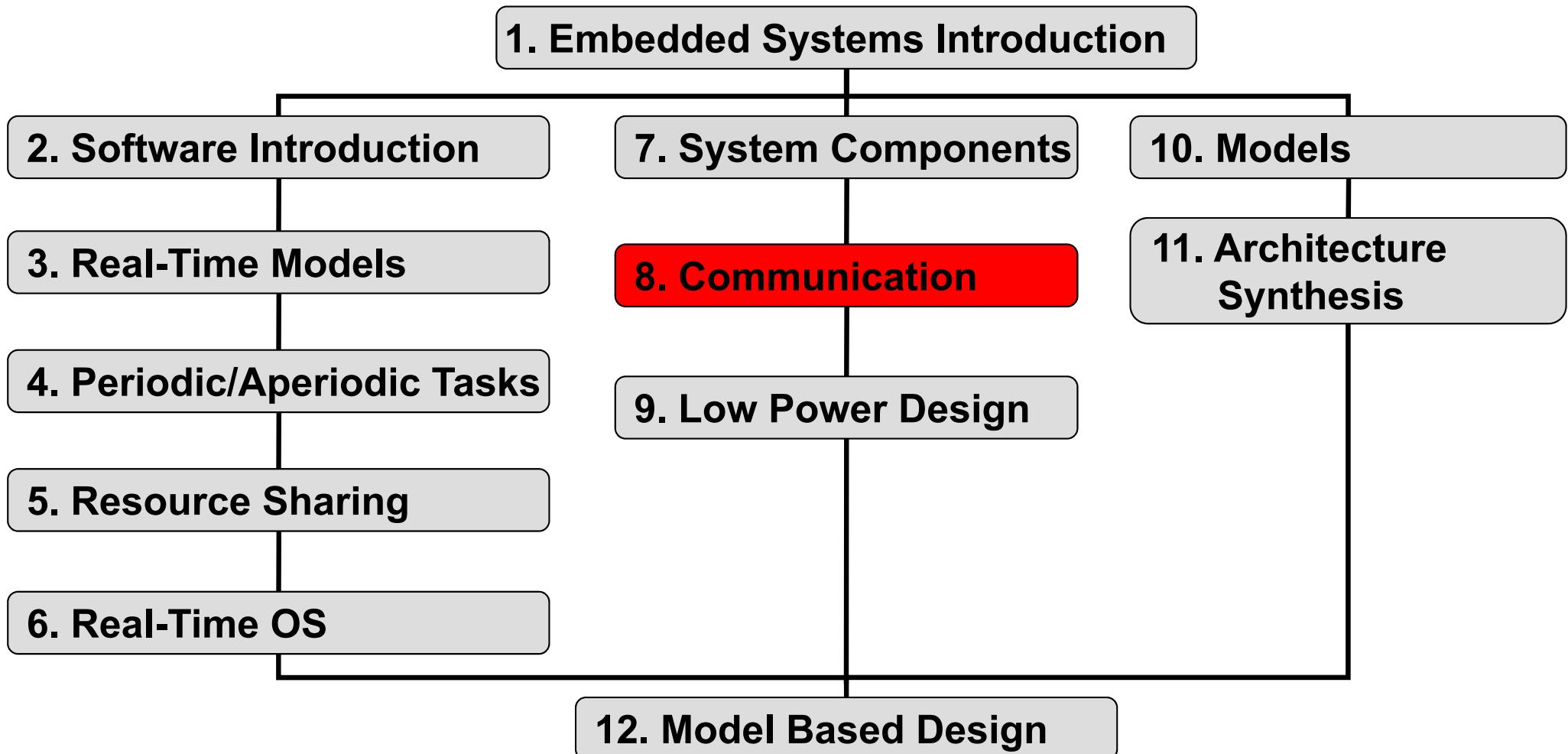


Embedded Systems

8. Communication

Lothar Thiele

Contents of Course



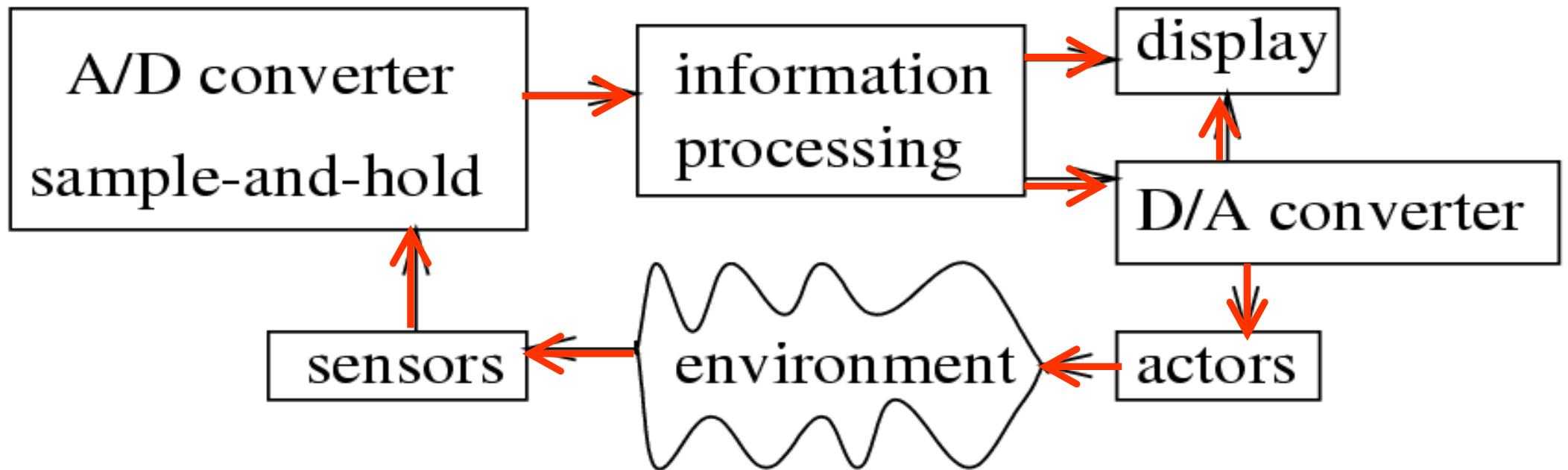
*Software and
Programming*

*Processing and
Communication*

Hardware

Communication

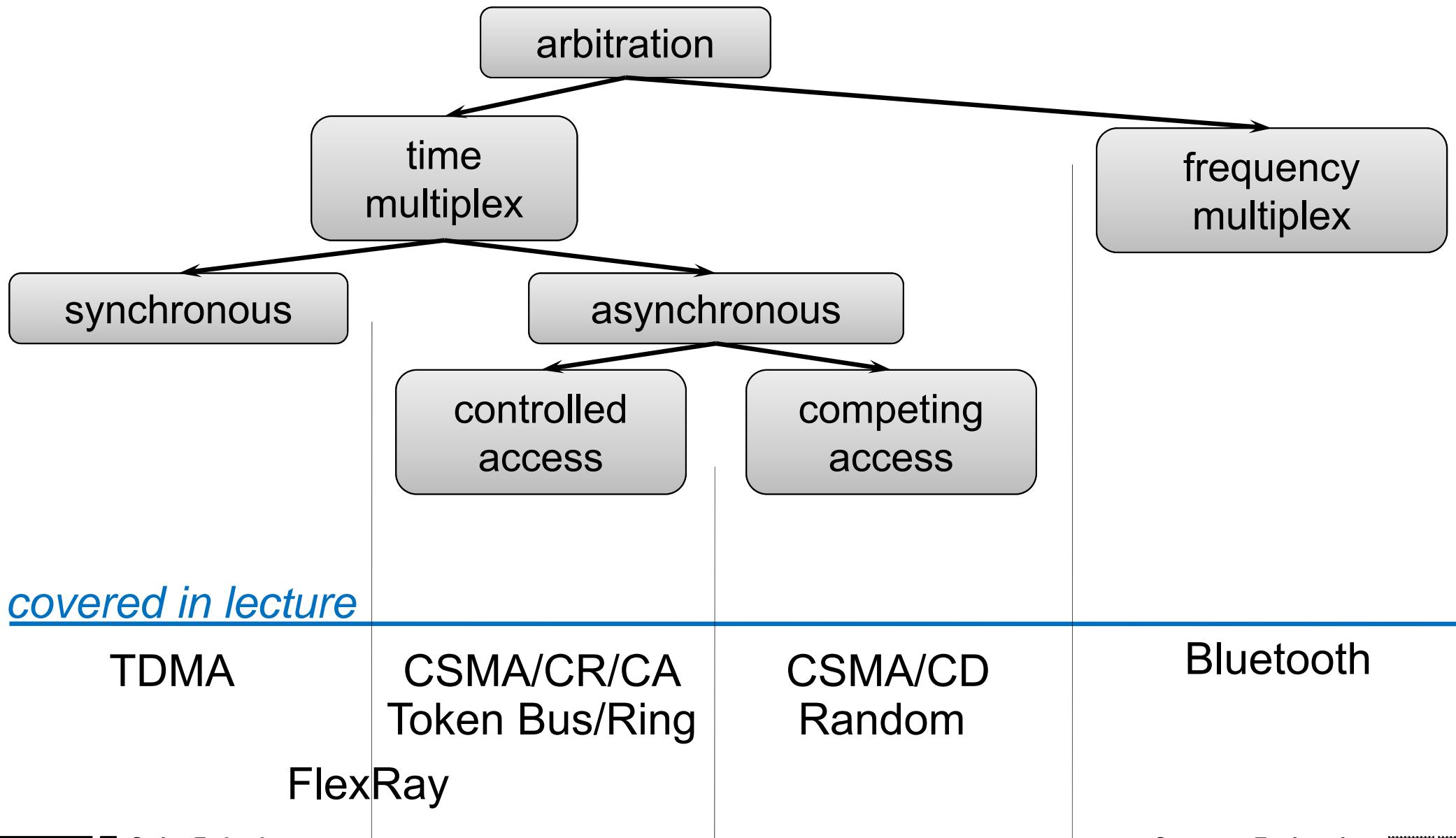
is everywhere ...



Communication: Requirements

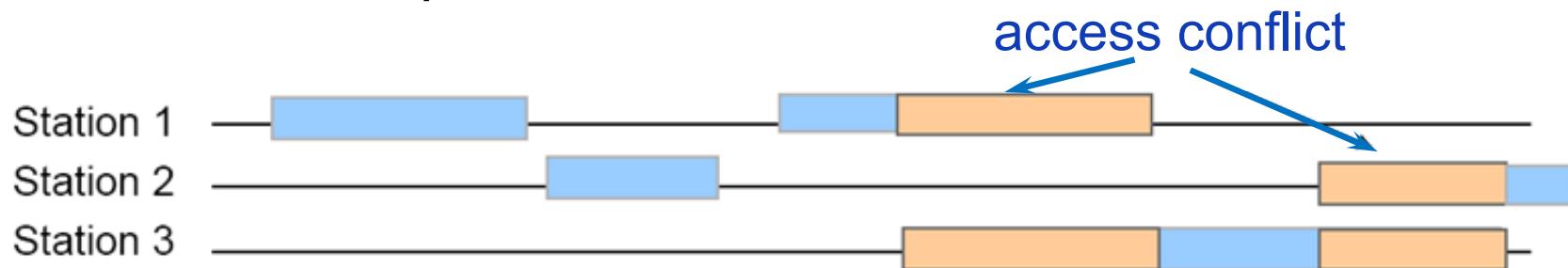
- Performance
 - bandwidth and latency
 - guaranteed behavior (real-time)
- Efficiency
 - cost (material, installation, maintenance)
 - low power
- Robustness
 - fault tolerance
 - maintainability, diagnose-ability
 - security, safety

Protocol Classification

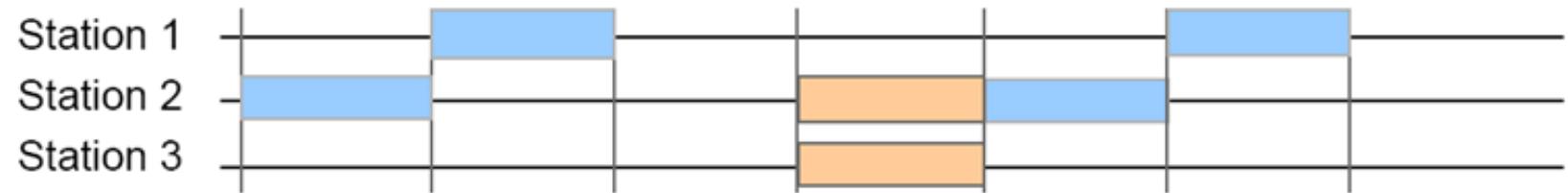


Random Access

- ▶ Random access to communication medium
 - no access control; requires low medium utilization



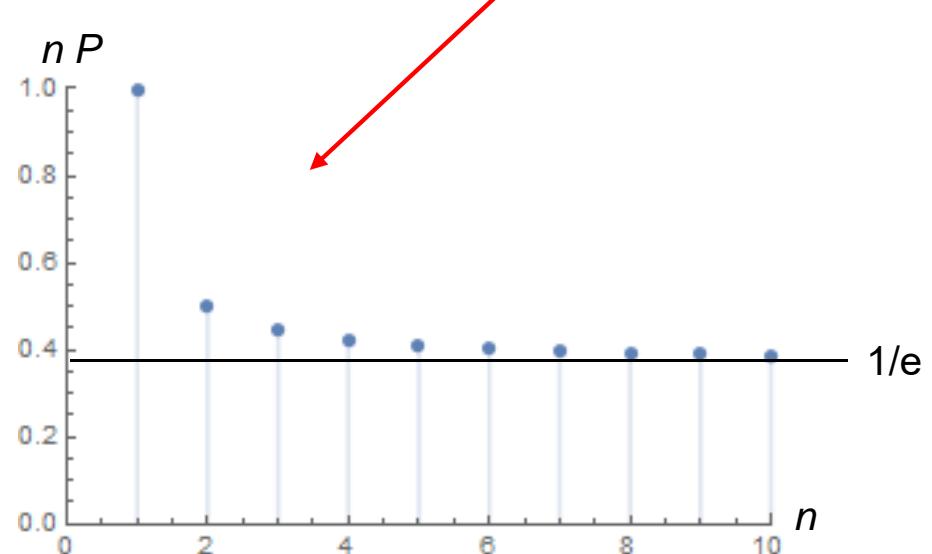
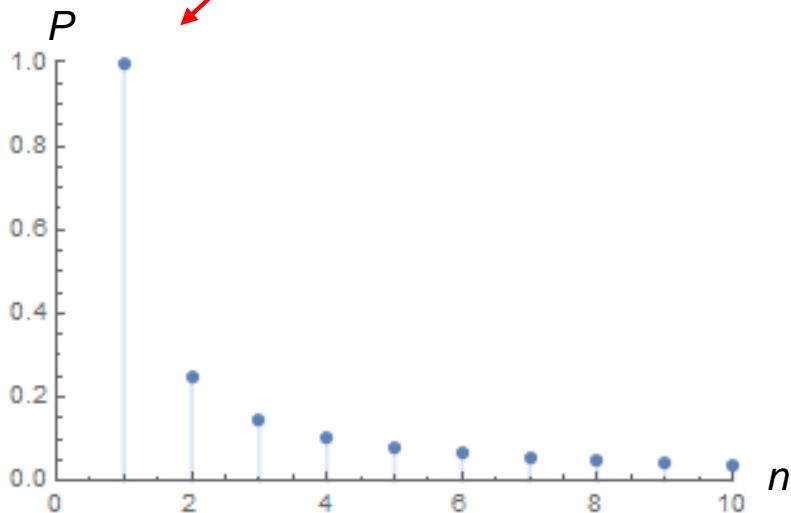
- improved variant: slotted random access



- What is the optimal sending rate p in case of n stations?
 - probability that a slot is not taken by others: $(1 - p)^{n-1}$
 - probability that a station transmits successfully: $P = p \cdot (1 - p)^{n-1}$
 - determine maximum with respect to p : $dP/dp = 0 \rightarrow p = 1/n$

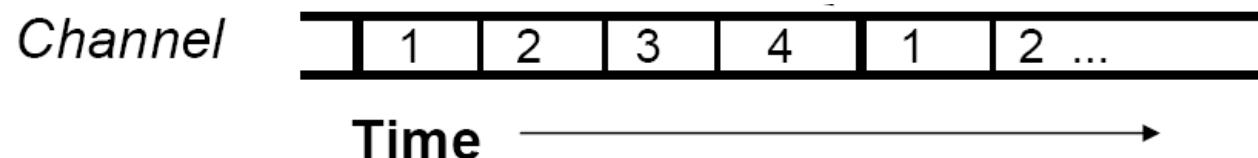
Random Access

- ▶ Random access to communication medium
 - What is the optimal sending rate p in case of n stations?
 - probability that a slot is not taken by others: $(1 - p)^{n-1}$
 - probability that a station transmits successfully: $P = p \cdot (1 - p)^{n-1}$
 - determine maximum with respect to p : $dP/dp = 0 \rightarrow p = 1/n$
 - optimal probability that a station can successfully transmit: P
 - optimal probability that a slot contains useful data: $n P$

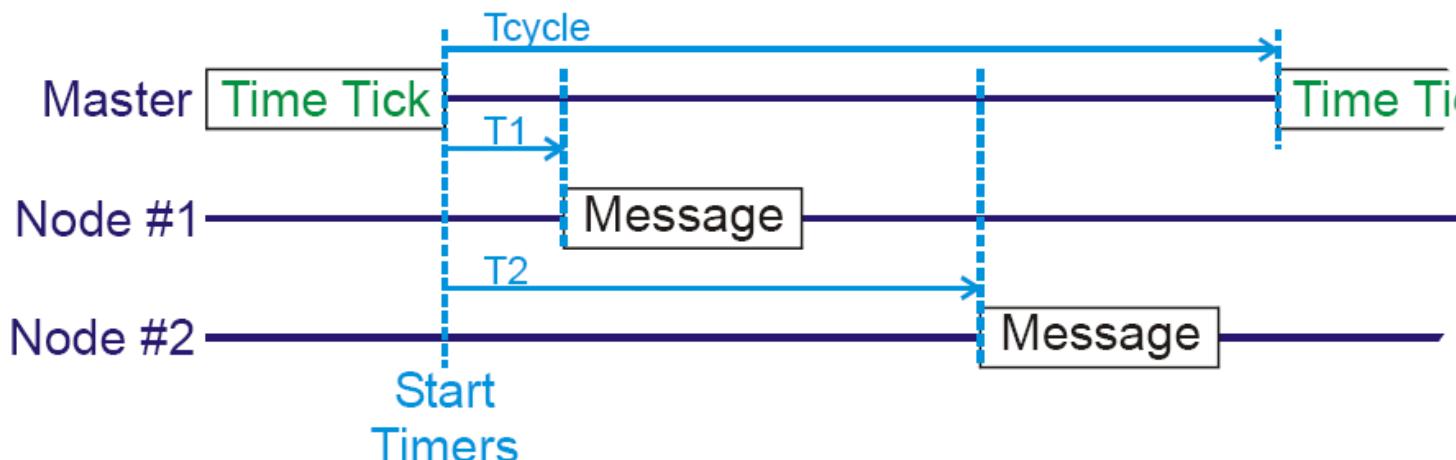


TDMA

- ▶ Communication in statically allocated time slots
- ▶ Synchronization among all nodes necessary:
 - periodic repetition of communication frame or

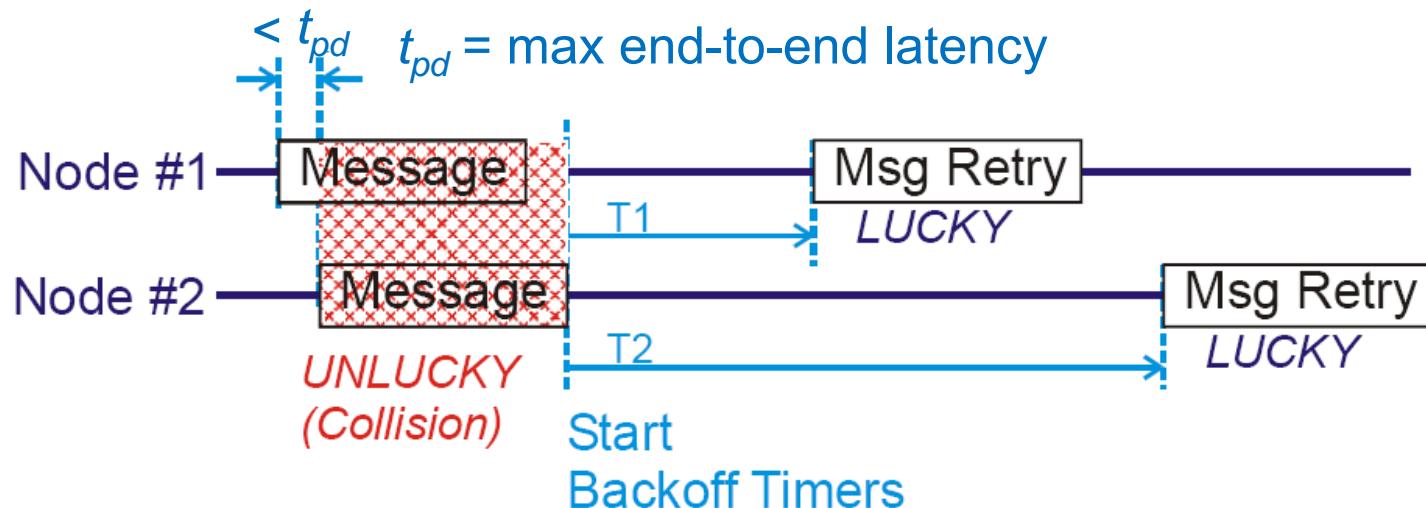


- master node sends out a synchronization frame
- ▶ Examples: TTP, static portion of FlexRay, satellite networks



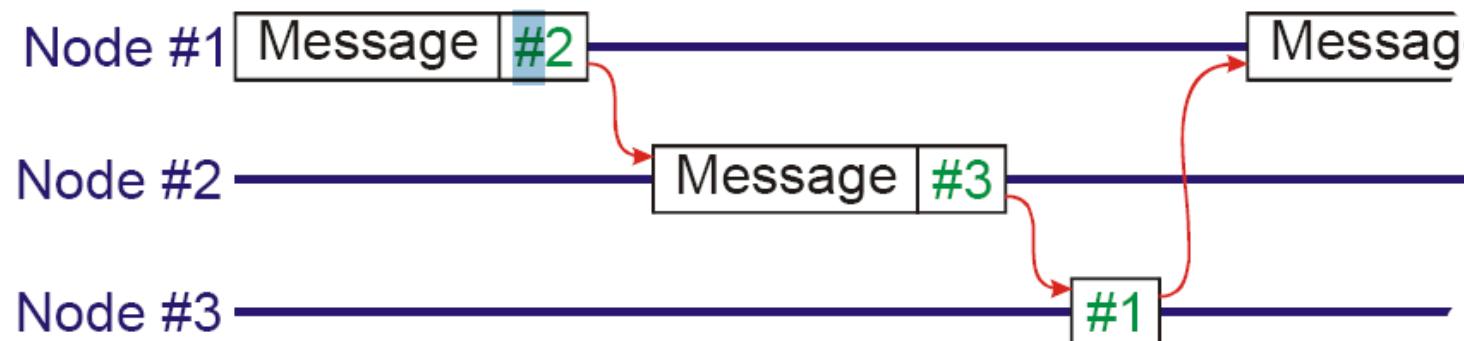
CSMA/CD

- ▶ Carrier Sense Multiple Access / Collision Detection
- ▶ Try to avoid and detect collisions:
 - before starting to transmit, check whether the channel is idle
 - if a collision is detected (several nodes started almost simultaneously), wait for some time (backoff timer)
 - repeated collisions result in increasing backoff times
- ▶ Examples: Ethernet, IEEE 802.3



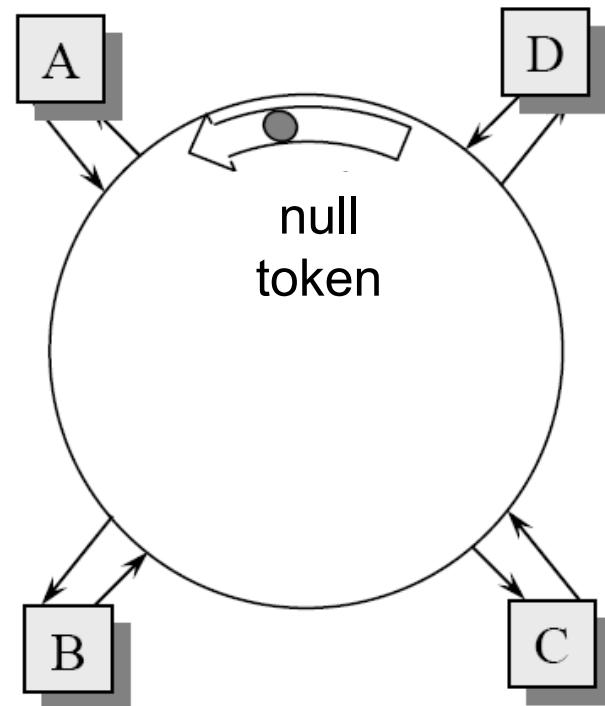
Token Protocols

- ▶ Token value determines which node is transmitting and/or should transmit next
 - Only the token holder may transmit
 - Master/slave polling is a special form
 - Null messages with tokens must be passed to prevent network from going idle
- ▶ Examples: IEEE 802.4, Profibus, TokenRing

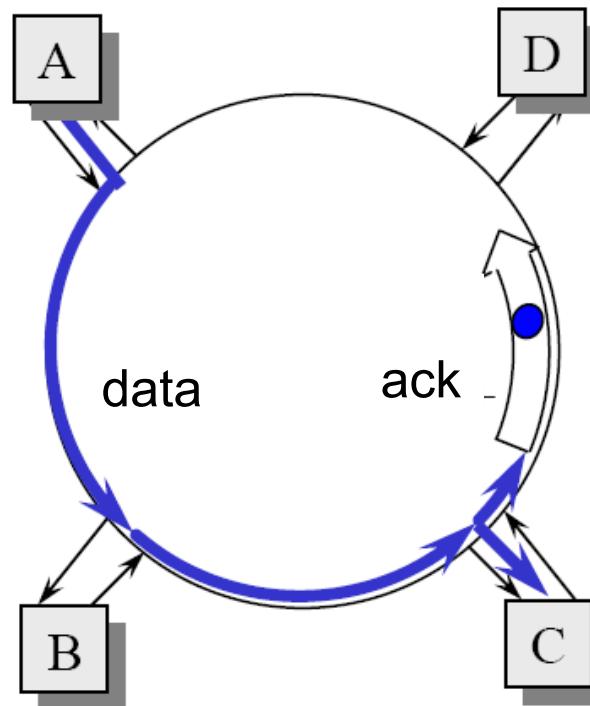


Token passes to next node according to # field.

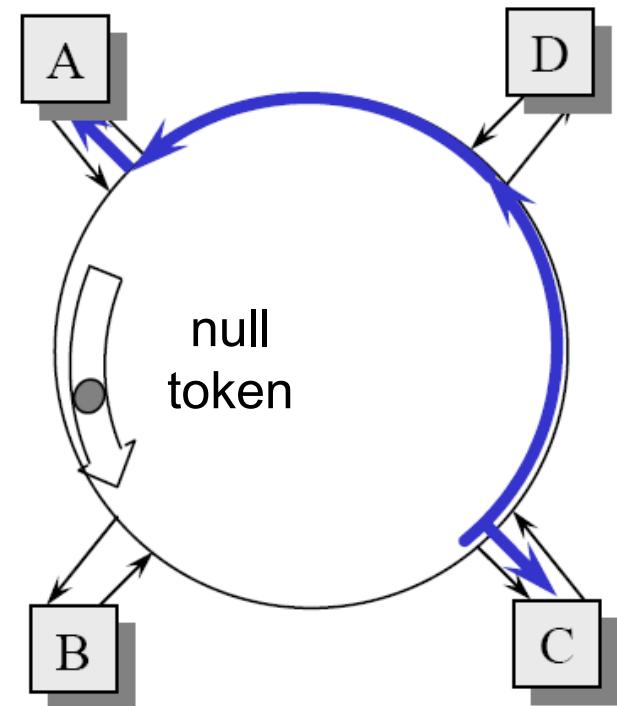
Token Ring



A requests transmission



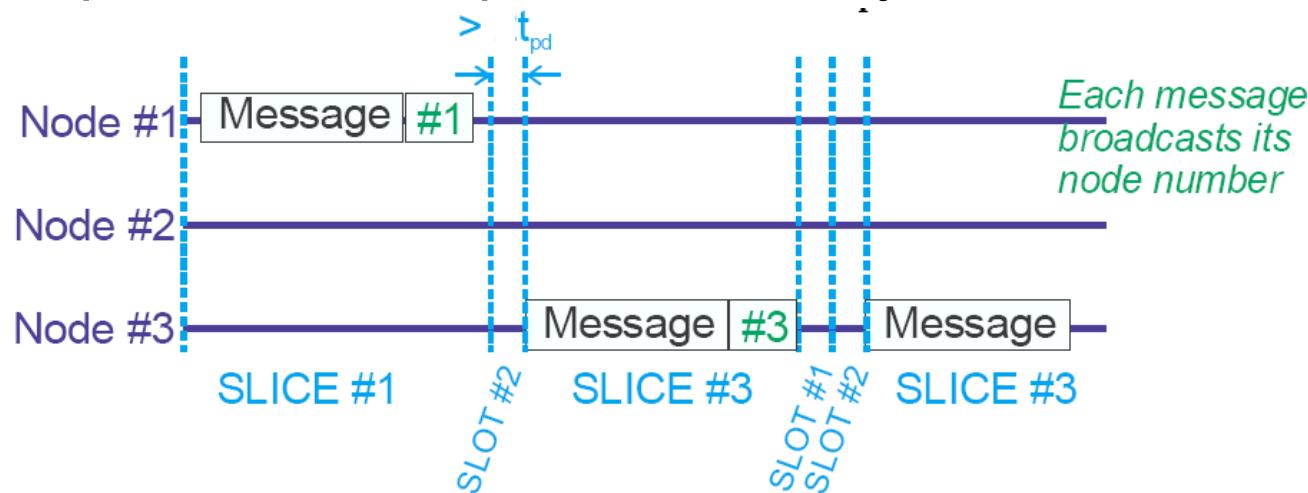
A is token owner
A sends data to C
C sends
acknowledge



A sends null token

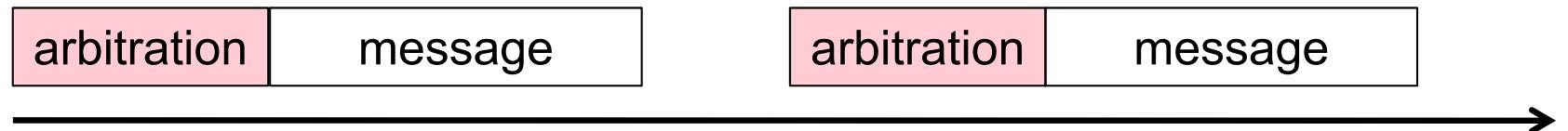
CSMA/CA – Flexible TDMA (FTDMA)

- ▶ Carrier Sense Multiple Access / Collision Avoidance
- ▶ Operation:
 - reserve s **slots** for n nodes; note: slots are normally idle – they are (short) time intervals, not signals; if slot is used it becomes a **slice**.
 - nodes keep track of global communication state by sensing
 - nodes start transmitting a message only during the assigned slot
 - If $s=n$, no collisions; if $s < n$, collision may occur (see random access)
 - Examples: 802.11, part of FlexRay



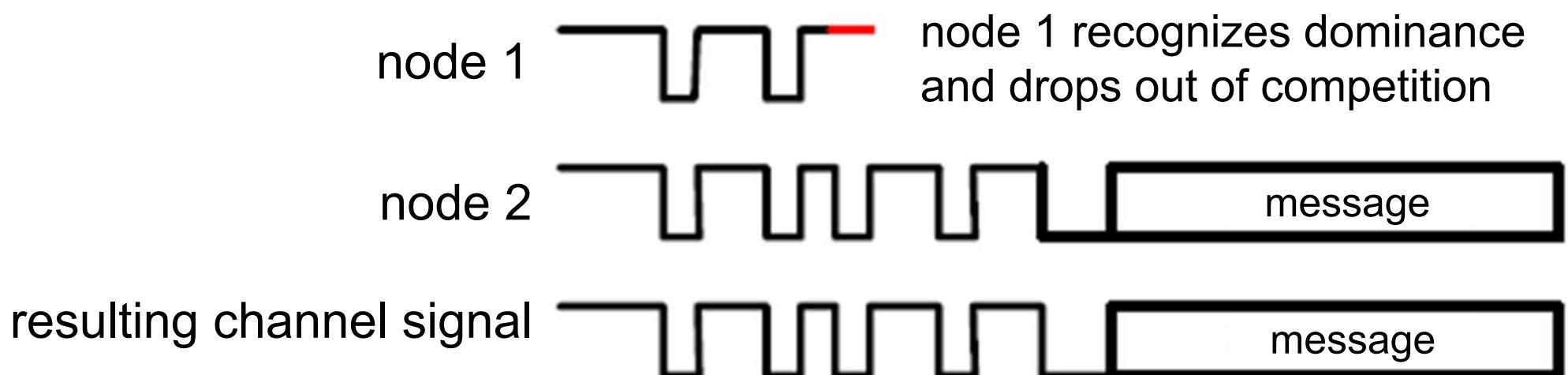
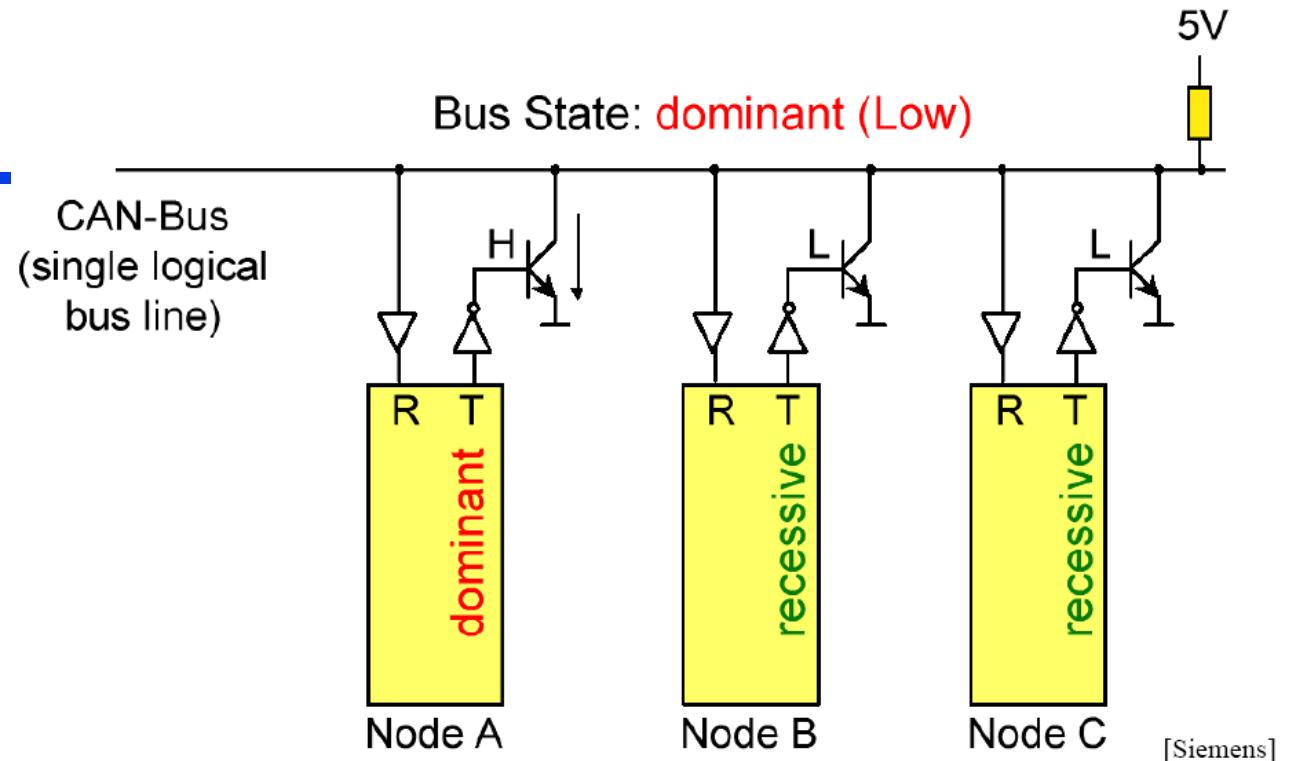
CSMA/CR

- ▶ Carrier Sense Multiple Access / Collision Resolution
- ▶ Operation:
 - Before any message transmission, there is a global arbitration



- Each node is assigned a unique identification number
- All nodes wishing to transmit compete by transmitting a binary signal based on their identification value
- A node drops out the competition if it detects a dominant state while transmitting a passive state
- Thus, the node with the lowest identification value wins
- ▶ Example: CAN Bus

CSMA/CR

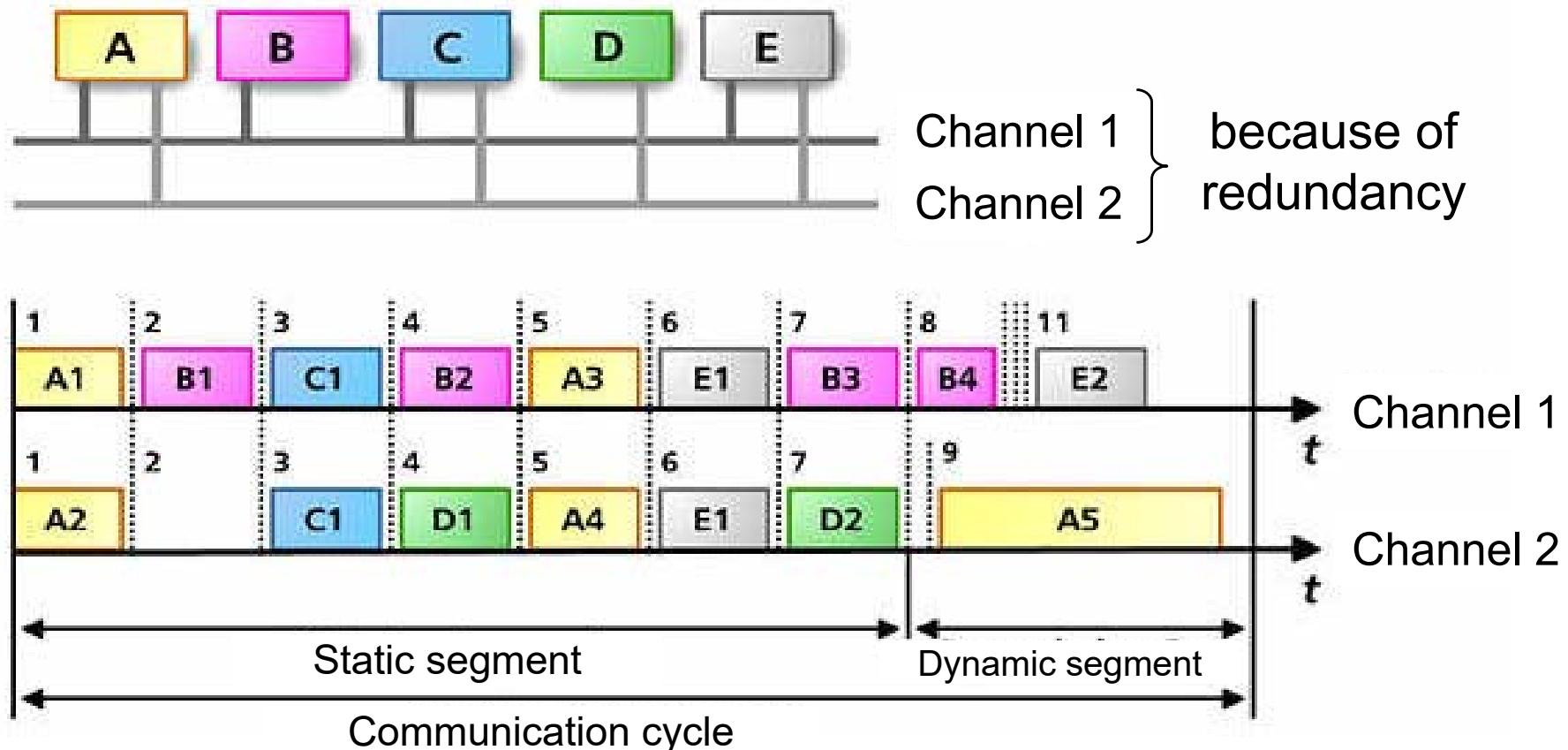


FlexRay

- ▶ FlexRay:
 - Developed by the FlexRay consortium (BMW, Ford, Bosch, DaimlerChrysler, General Motors, Motorola, Philips).
 - Combination of a TDMA and the Byteflight [Byteflight Consortium, 2003] (Flexible TDMA, close to CSMA/CA) protocol.
 - High data rates can be achieved:
 - initially targeted for ~ 10Mbit/sec;
 - design allows much higher data rates
- ▶ Operation principle:
 - Cycle is subdivided into a static and a dynamic segment.
 - Static segment is based on a fixed allocation of time slots to nodes.
 - Dynamic segment for transmission of ad-hoc communication with variable bandwidth requirements.

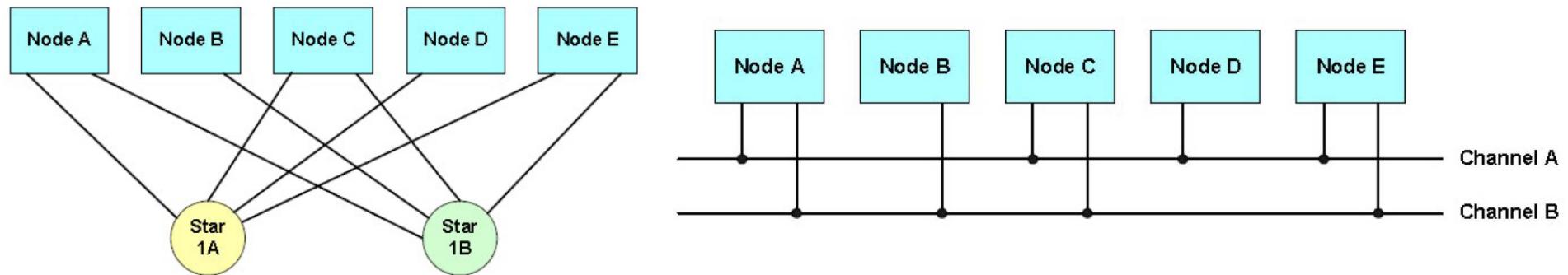
FlexRay

- ▶ Use of two independent channels to eliminate single-point failures

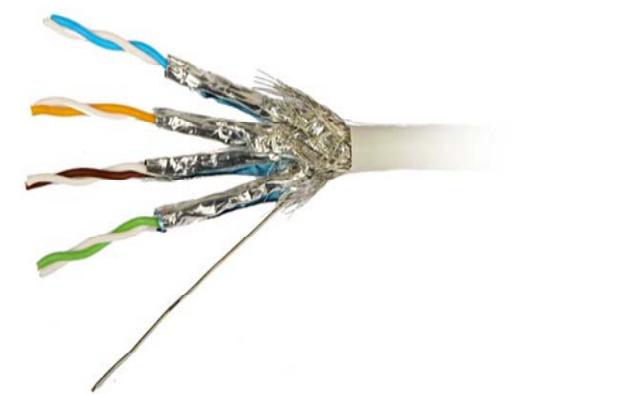
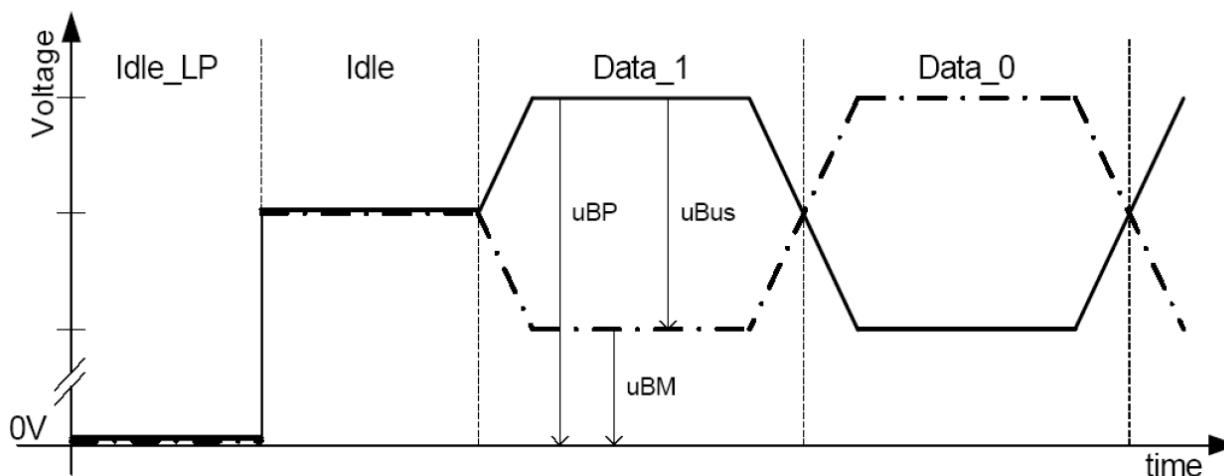


FlexRay

- ▶ Basic topologies (any combination also possible):



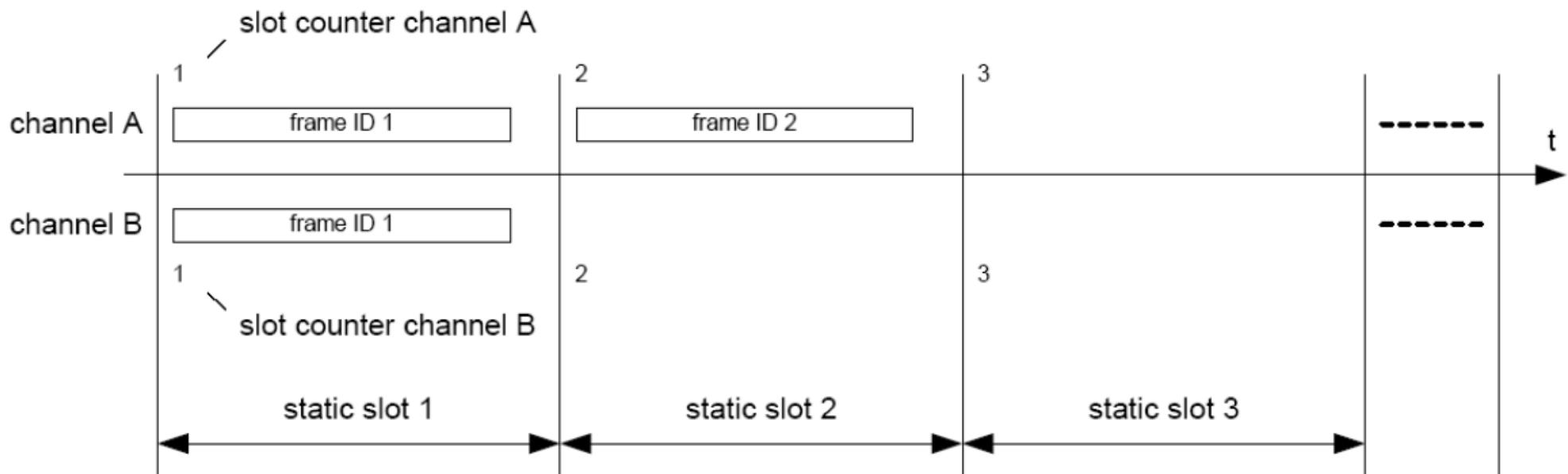
- ▶ Typical physical layer (twisted pairs and differential encoding to reduce sensitivity to electromagnetic coupling):



FlexRay

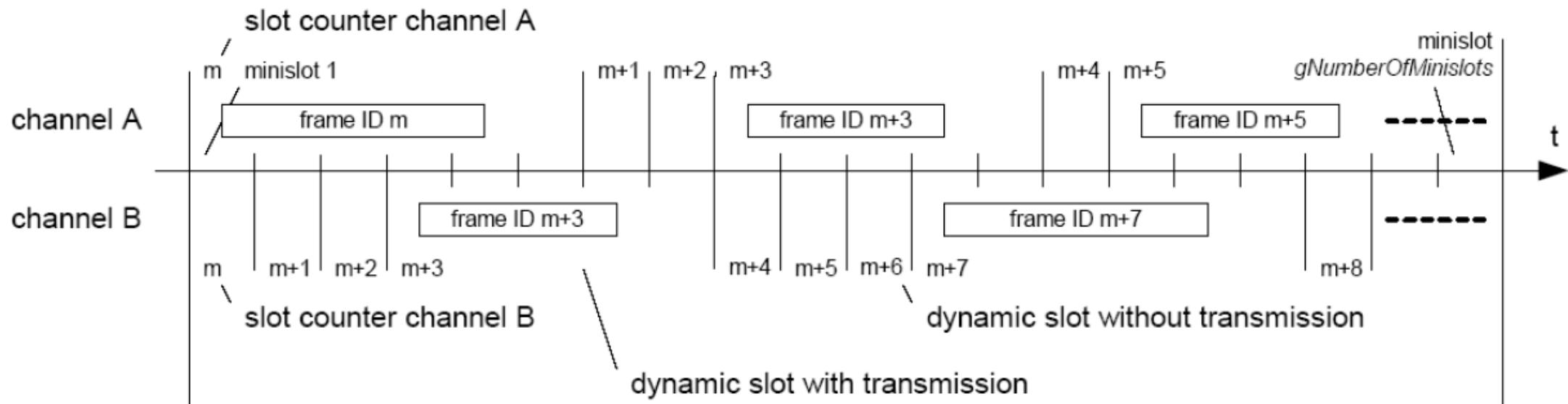
- ▶ TDMA

- all static slots are the same length whether used or not
- all slots are repeated in order every communication cycle
- slots are lock-stepped in order on both channels

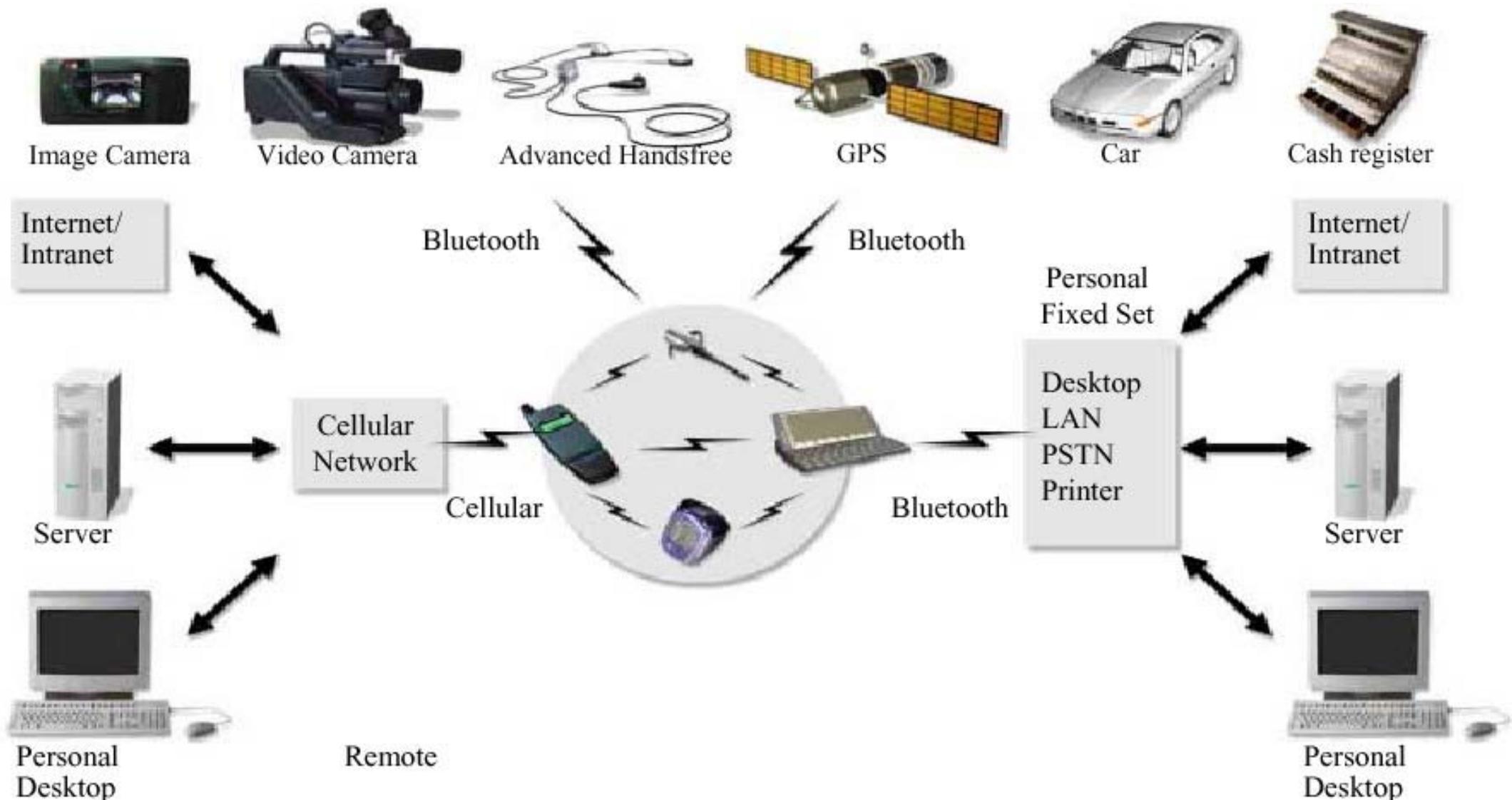


FlexRay

- ▶ Flexible TDMA:
 - each minislot is an **opportunity** to send a message
 - if message isn't sent, minislot elapses unused (short idle period)
 - all nodes watch whether a message is sent so they can count minislots



Example Bluetooth



Who was Bluetooth?

- ▶ Wikinger
- ▶ King of Denmark 940-981
- ▶ Christianized, unified and controlled Denmark and Norway



Bluetooth Overview

- ▶ ***Design Goals***
 - small size, low cost, low energy
 - secure transmission (encryption, authentication)
 - robust transmission (interference with wireless LAN)
- ▶ ***Technical Data***
 - 2.4 GHz Band (open band, spectral bandwidth 79 MHz, frequency hopping and time multiplex)
 - 10-100 m transmission range, 1 Mbit/s bandwidth for each connection
 - simultaneous transmission of multimedia streams (synchronous) and data (asynchronous)
 - ad hoc network (spontaneous connections to neighbor nodes, dynamic network topologies, no centralized coordination, multi-hop communication)

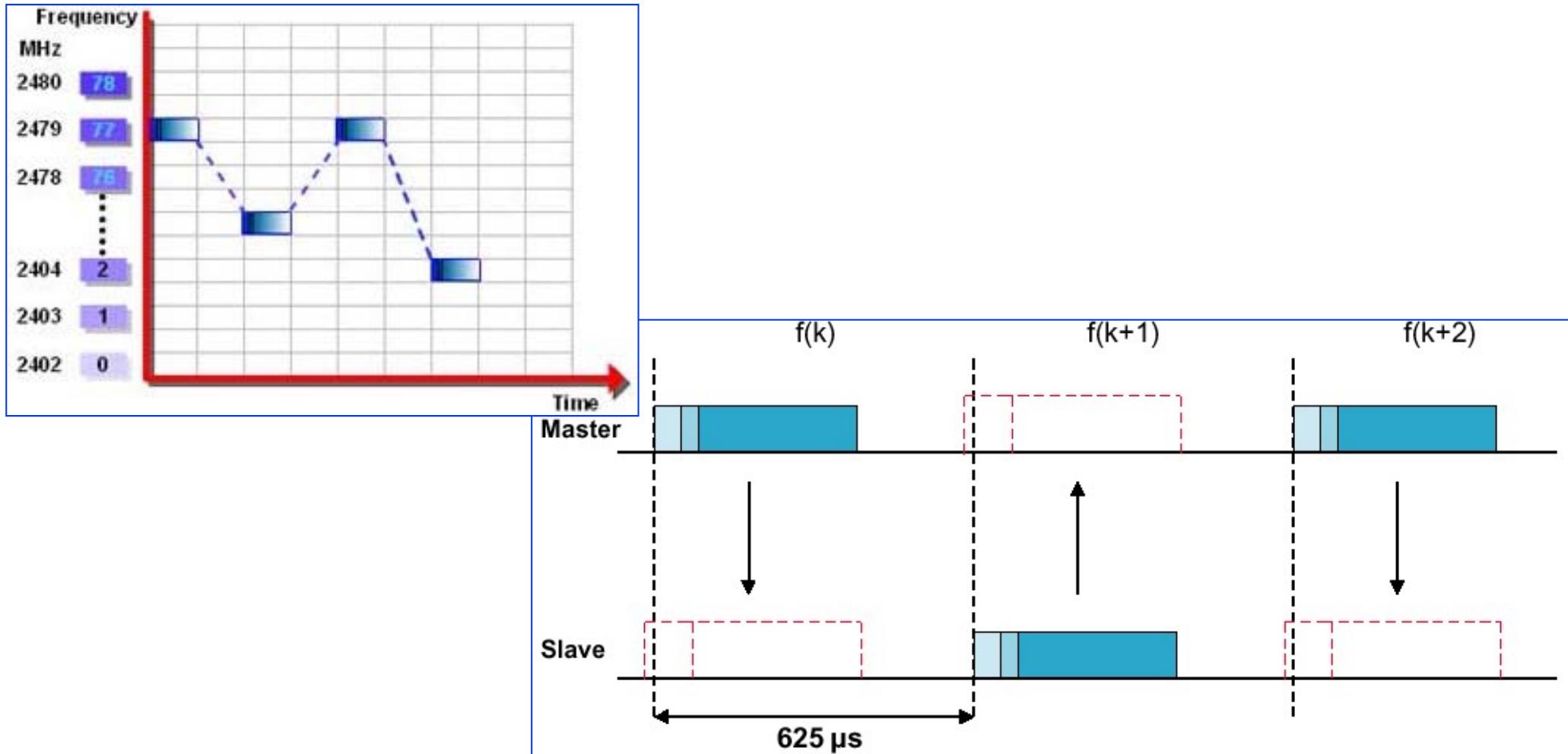
Bluetooth Overview

► *Frequency Hopping*

- Transmitter jumps from one frequency to another with a fixed rate (1600 hops/s). The ordering (channel sequence) is determined by a pseudo random sequence of length $2^{27}-1$.
- Frequency range $(2402 + k)$ MHz, $k = 0 \dots 78$.
- The data transmission is partitioned into time windows of length 0.625 ms; each packet is transmitted by means of a different frequency.

Bluetooth Overview

Example frequency hopping:



**SEARCH** The Web CNN.com**Search**[Home Page](#)[World](#)[U.S.](#)[Weather](#)[Business at CNNMoney](#)[Sports at SI.com](#)[Politics](#)[Law](#)**Technology**[Science & Space](#)[Health](#)[Entertainment](#)[Travel](#)[Education](#)[Special Reports](#)[Autos with Ecomundo.com](#)**SERVICES**[Video](#)[E-mail Newsletters](#)[Your E-mail Alerts](#)[RSS](#)[CNNtoGO](#)[TV Commercials](#)[Contact Us](#)**SEARCH**

TECHNOLOGY

'Master' and 'slave' computer labels unacceptable, officials say

Wednesday, November 26, 2003 Posted: 3:24 PM EST (2024 GMT)

LOS ANGELES, California

(Reuters) -- Los Angeles officials have asked that manufacturers, suppliers and contractors stop using the terms "master" and "slave" on computer equipment, saying such terms are unacceptable and offensive.

The request -- which has some suppliers furious and others busy re-labeling components -- came after an unidentified worker spotted a videotape machine carrying devices labeled "master" and "slave" and filed a discrimination complaint with the county's Office of Affirmative Action Compliance.

In the computer industry, "master" and "slave" are used to refer to primary and secondary hard disk drives. The terms are also used in other industries.

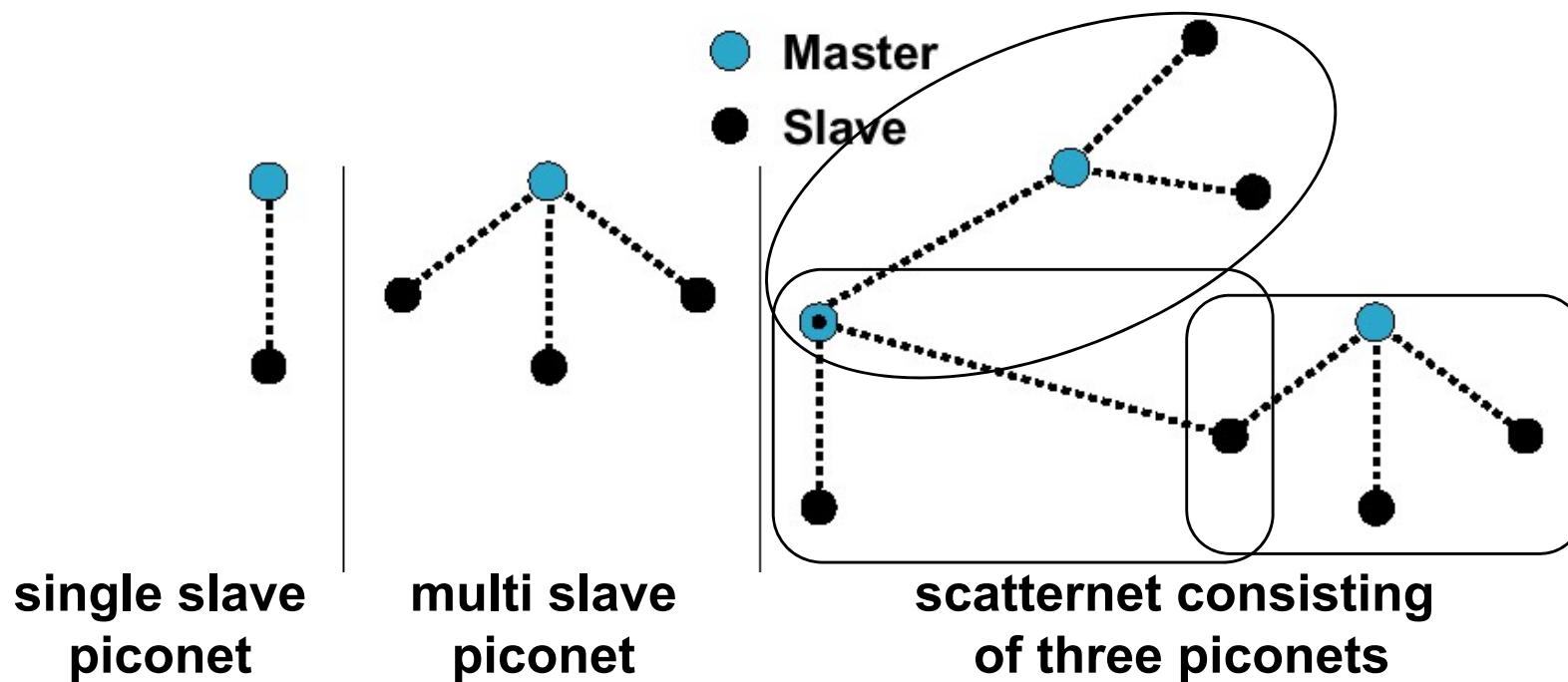
"Based on the cultural diversity and sensitivity of Los Angeles County, this is not an acceptable identification label," Joe Sandoval, division manager of purchasing and contract services, said in a memo sent to County vendors.

"We would request that each manufacturer, supplier and contractor review, identify and remove/change any identification or labeling of equipment components that could be interpreted as discriminatory or offensive in nature," Sandoval said in the memo, which was distributed last week and made available to Reuters.

Network Topologies

► *Ad-hoc networks*

- all nodes are potentially mobile
- dynamic emergence of connections
- hierarchical structure (scatternet) of small nets (piconet)



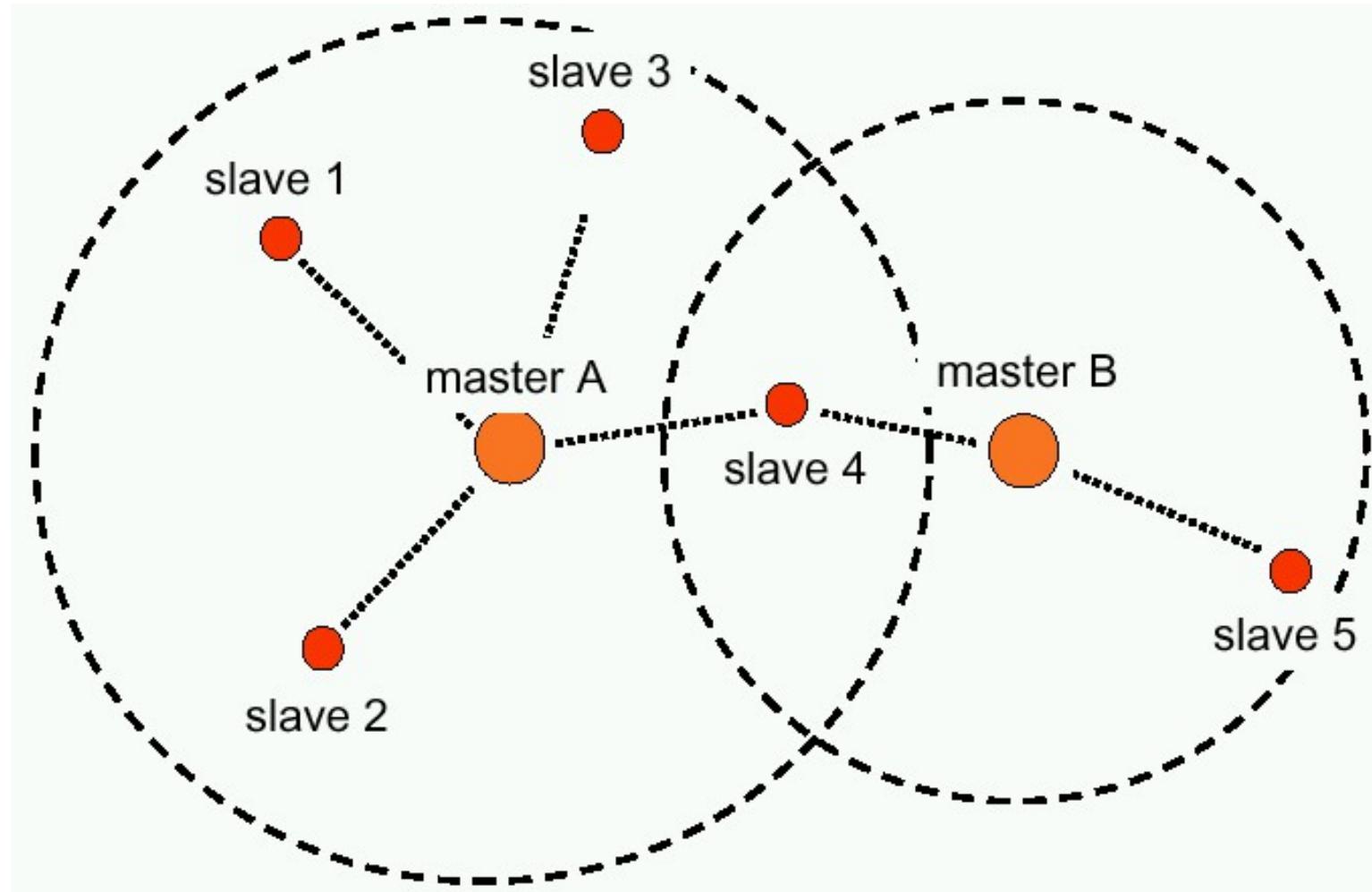
Network Topologies

► *Piconet*

- A piconet contains 1 master and maximally 7 slaves
- All nodes in a piconet use the same frequency hopping scheme (channel sequence) which is determined by
 - the device address of the master BD_ADDR and
 - phase which is determined by the system clock of the master.
- Connections are either one-to-one or between the master and all slaves (broadcast).
- The following connection types are possible:
 - 432 kBit/s (duplex) or 721/56 kBit/s (asymmetric) or
 - 3 audio channels or
 - a combination of data and audio.

Netzwerktopologien

► Scatternet

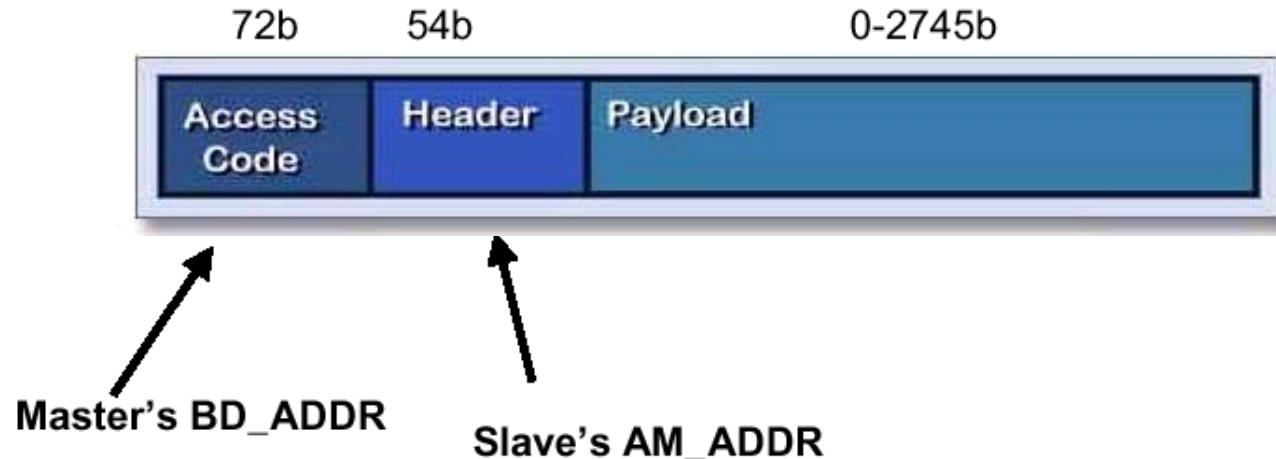


Network Topologies

▶ *Scatternet*

- Several piconets with overlapping nodes form a scatternet.
- A node can simultaneously have the roles of slaves in several piconets and the role of a master in at most one piconet.
- The channel sequences of the different piconets are not synchronized.
- As a result, large network structures can emerge and multi-hop communication is possible.

Packet Format



The access code identifies all packets between Bluetooth devices.

Packet Header identifies and characterizes the connection between master and slave.

Addressing

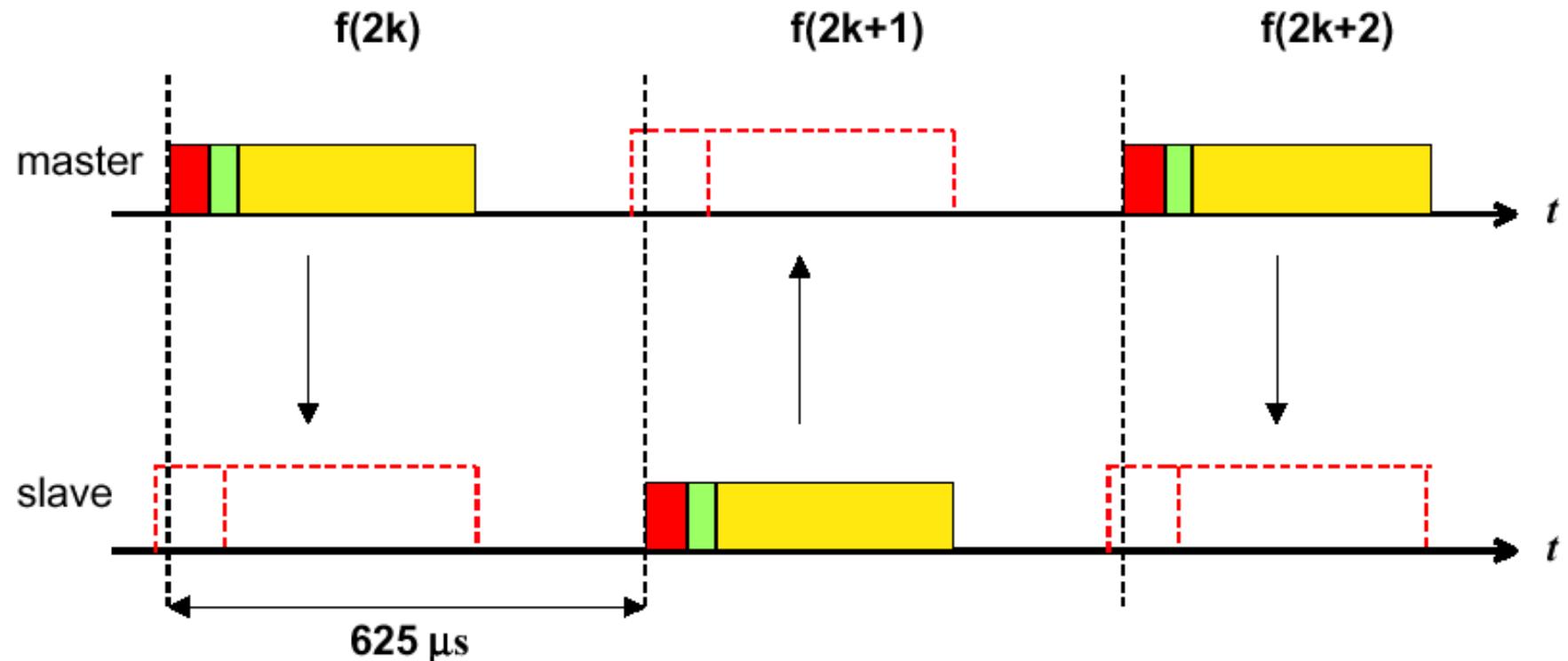
- ▶ ***Bluetooth Device Address: BD_ADDR***
 - 48 Bit
 - Unique address for each device
- ▶ ***Active Member Address AM_ADDR***
 - 3 Bit for maximally 7 active Slaves in a piconet.
 - Address “Null“ is a broadcast to all slaves.
- ▶ ***Parked Member Address PM_ADDR***
 - 8 Bit for parked slaves.

Connection Types

- ▶ Mixed transmission of data and audio.
- ▶ ***Synchronous Connection-Oriented*** (SCO)
 - Point to point full duplex connection between master & slaves
 - Master reserves slots to allow transmission of packets in regular intervals.
- ▶ ***Asynchronous Connection-Less*** (ACL)
 - Asynchronous service
 - No reservation of slots
 - The master transmits spontaneously, the addressed slave answers in the following interval.

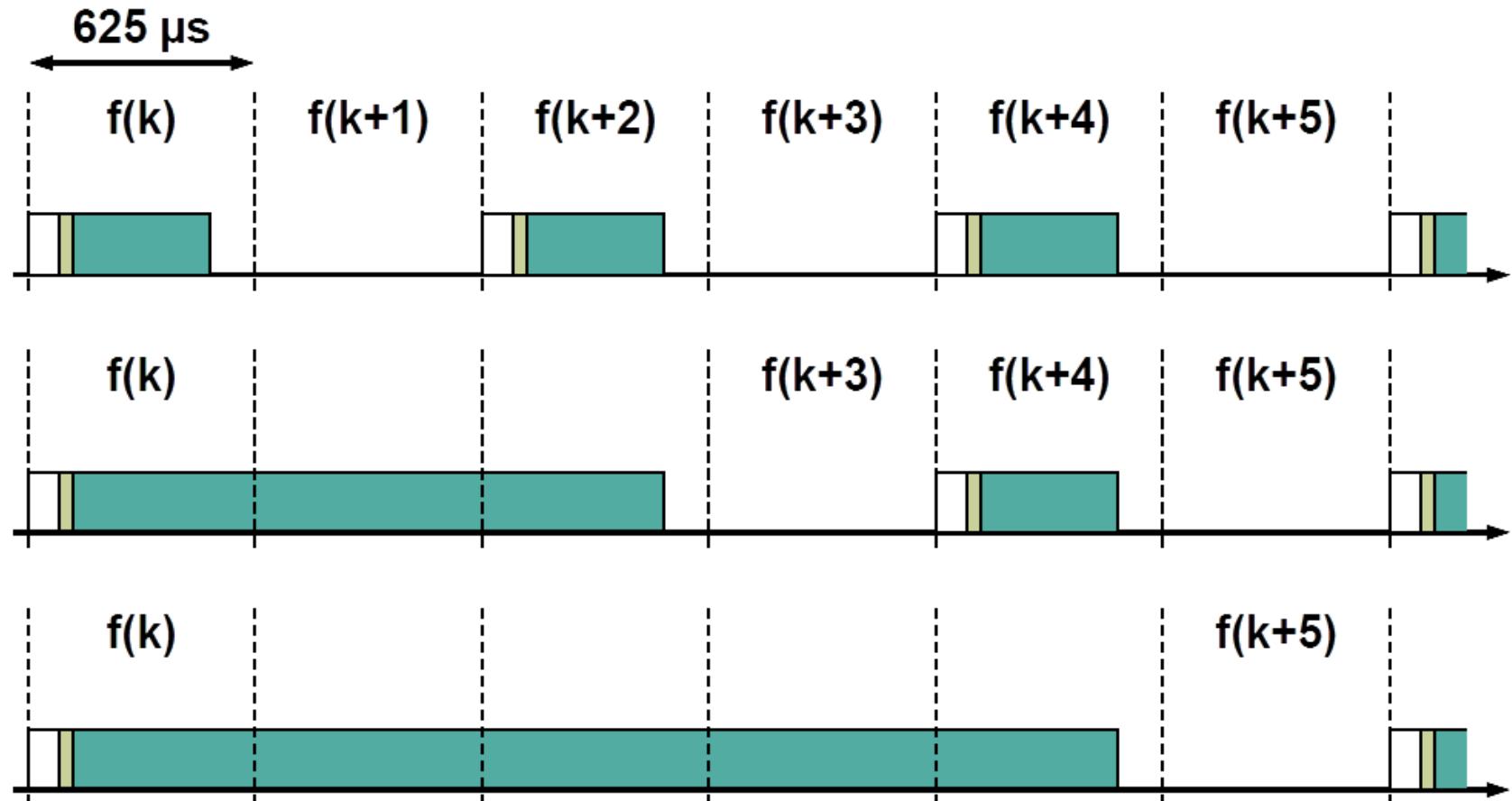
Frequency Hopping Time Multiplex

- ▶ A packet of the master is followed by a slave packet.
- ▶ After each packet, the channel (frequency) is switched.

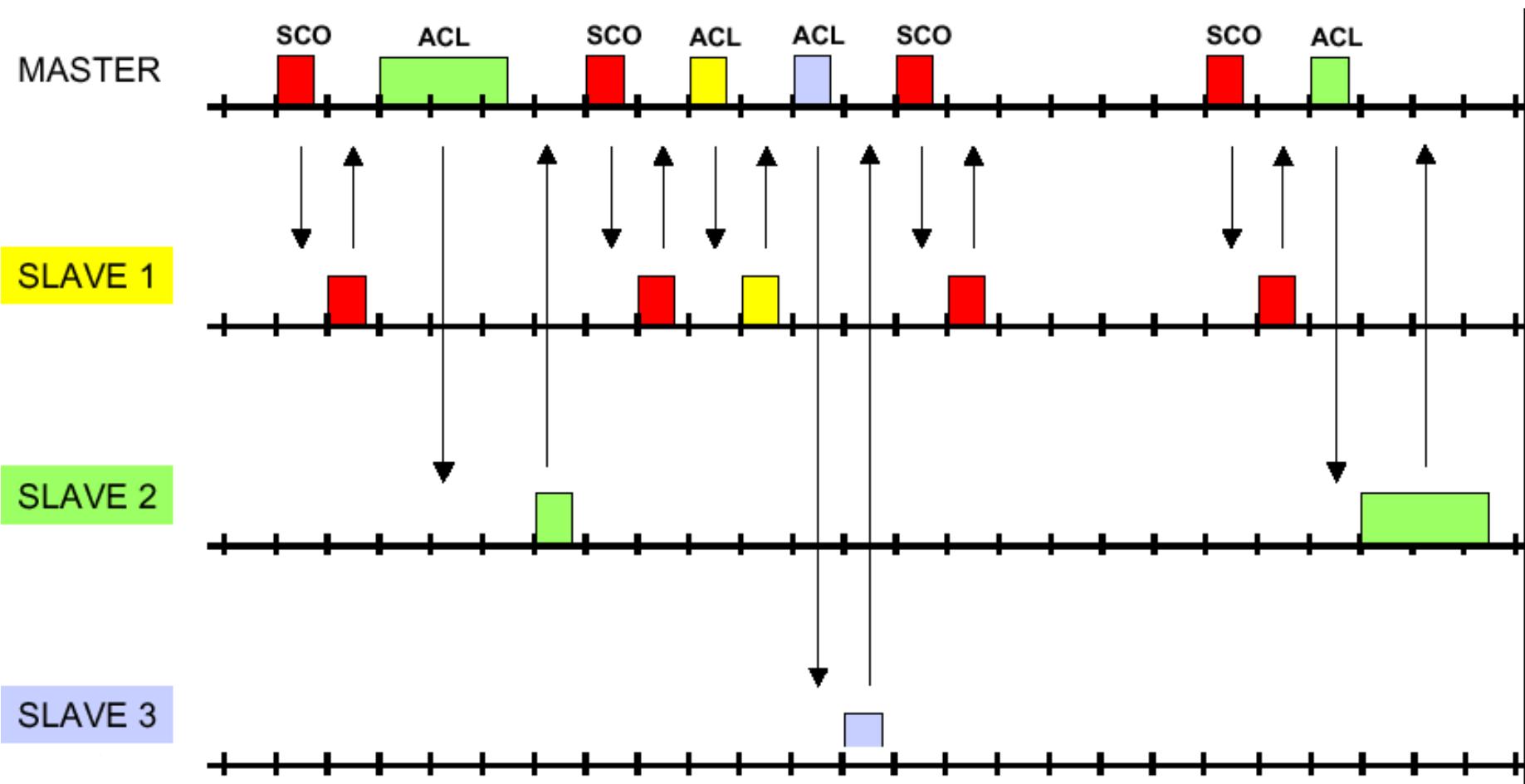


Multi-Slot Communication

- ▶ Master can only start sending in even slot numbers.
- ▶ Packets from master or slave have length of 1, 3 or 5 slots.



ACL and SCO Connections

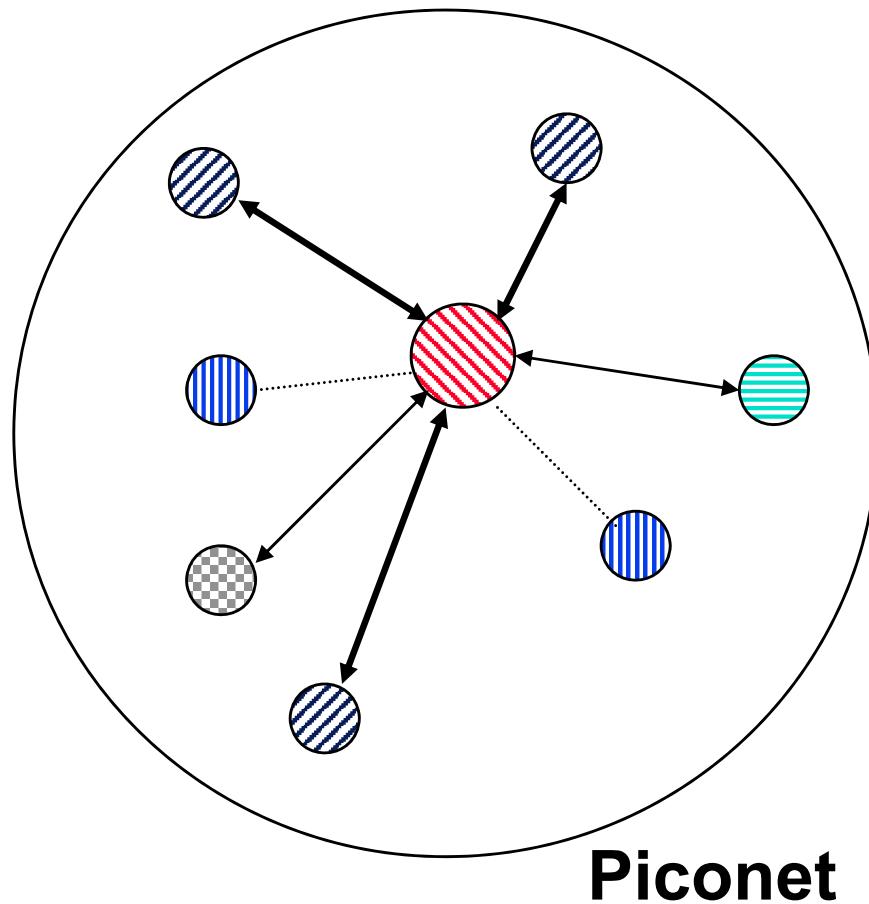


Modes and States

- ▶ **Modes of operation:**
 - **Inquiry** (master identifies addresses of neighboring nodes)
 - **Page** (master attempts connection to a slave whose address BD_ADDR is known)
 - **Connected** (connection between master and slave is established)
- ▶ **States in connection mode** (sorted in decreasing order of power consumption)
 - **active** (active in a connection to a master)
 - **hold** (does not process data packets)
 - **sniff** (awakens in regular time intervals)
 - **park** (passive, in *no connection* with master but still synchronized)

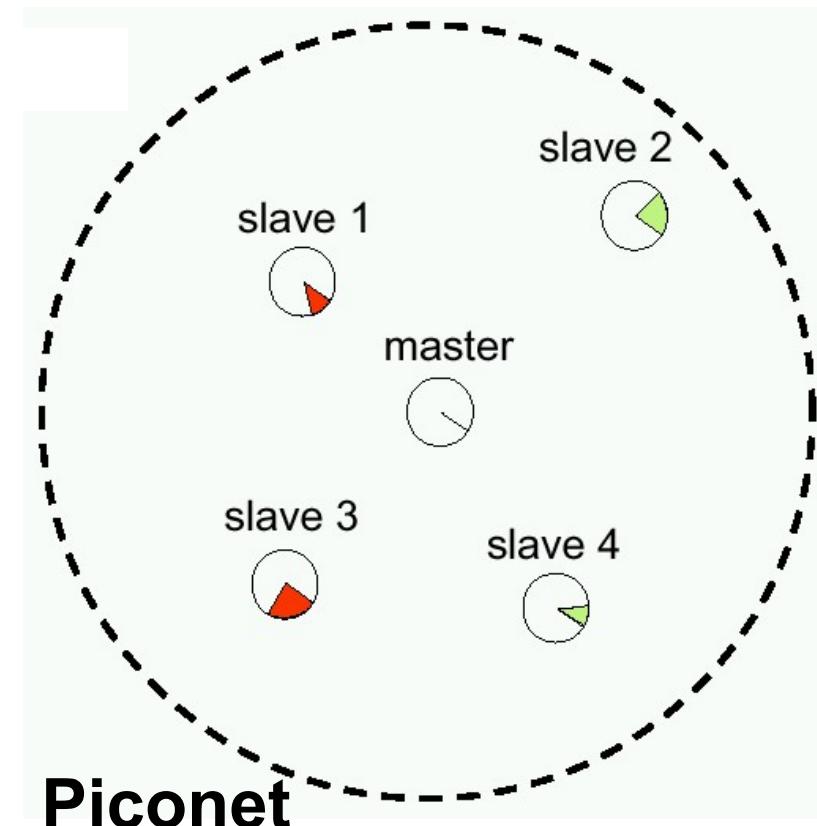
States in Connection Mode

-  master
-  active
-  hold
-  sniff
-  park

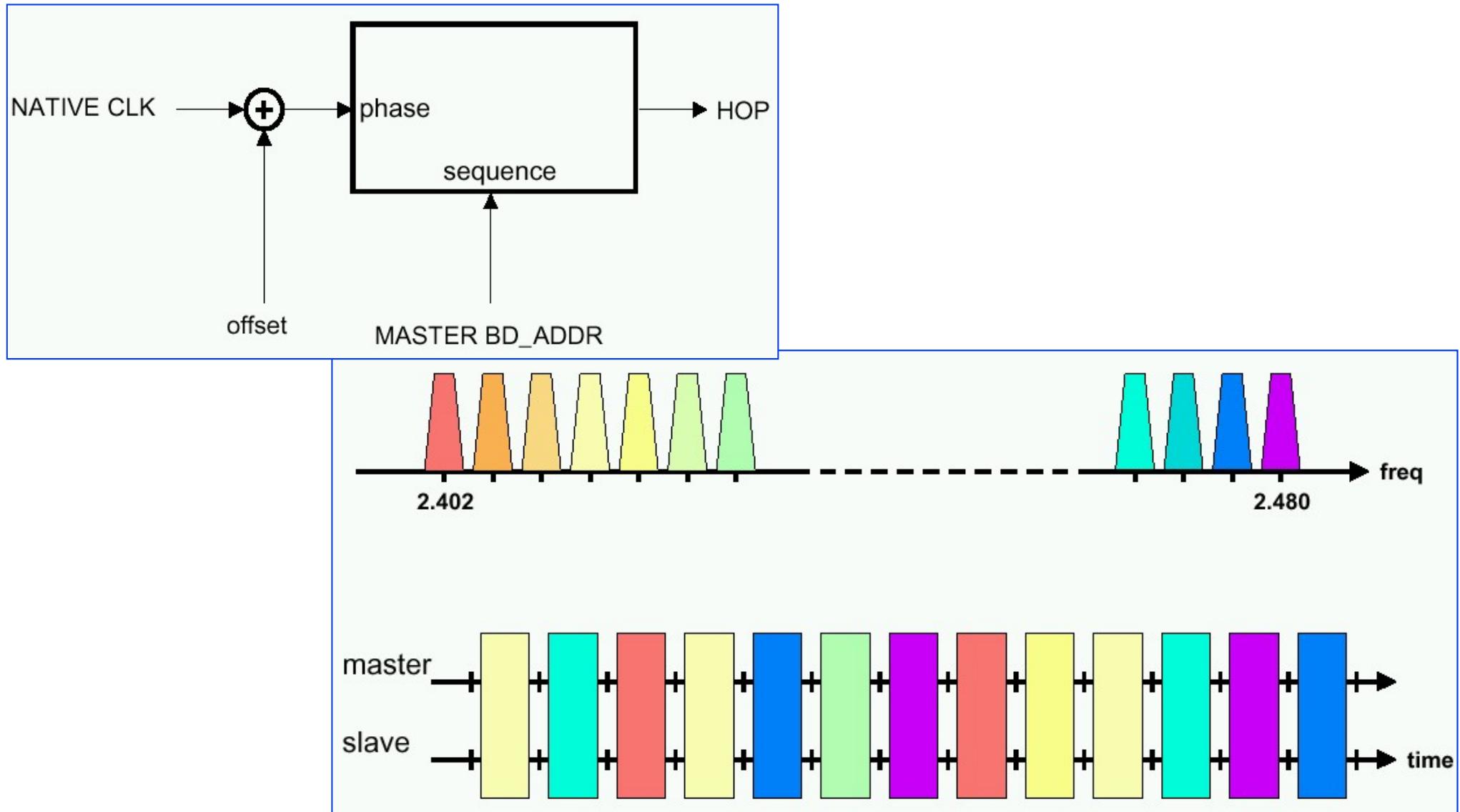


Synchronization in Connection Mode

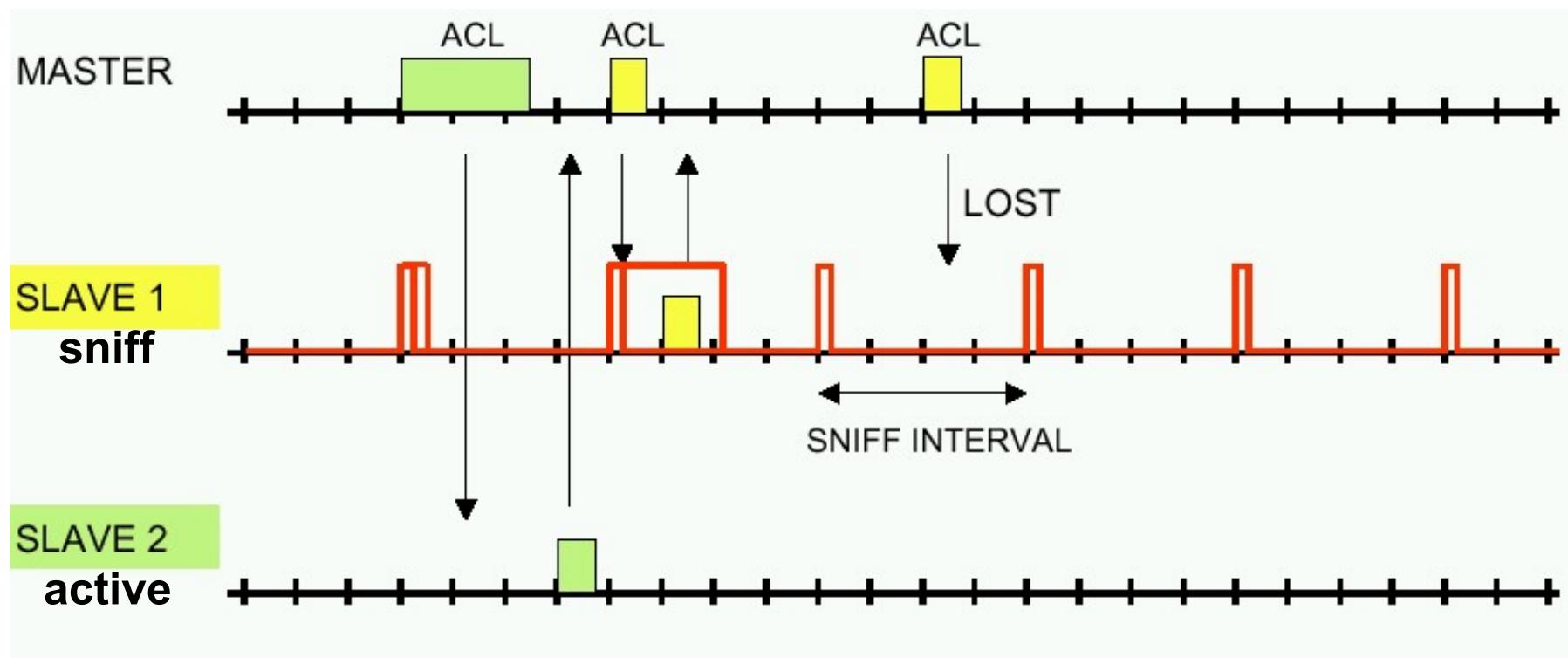
- ▶ The **channel sequence** of a piconet is determined by the BD_ADDR of the master.
- ▶ The **phase** within the sequence is also determined by the master; all slaves follow.



Synchronization in Connection Mode



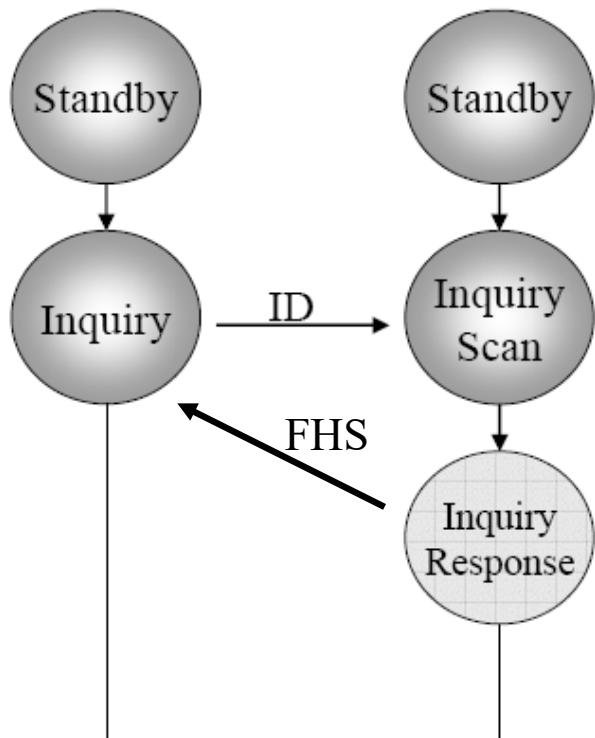
The Sniff State



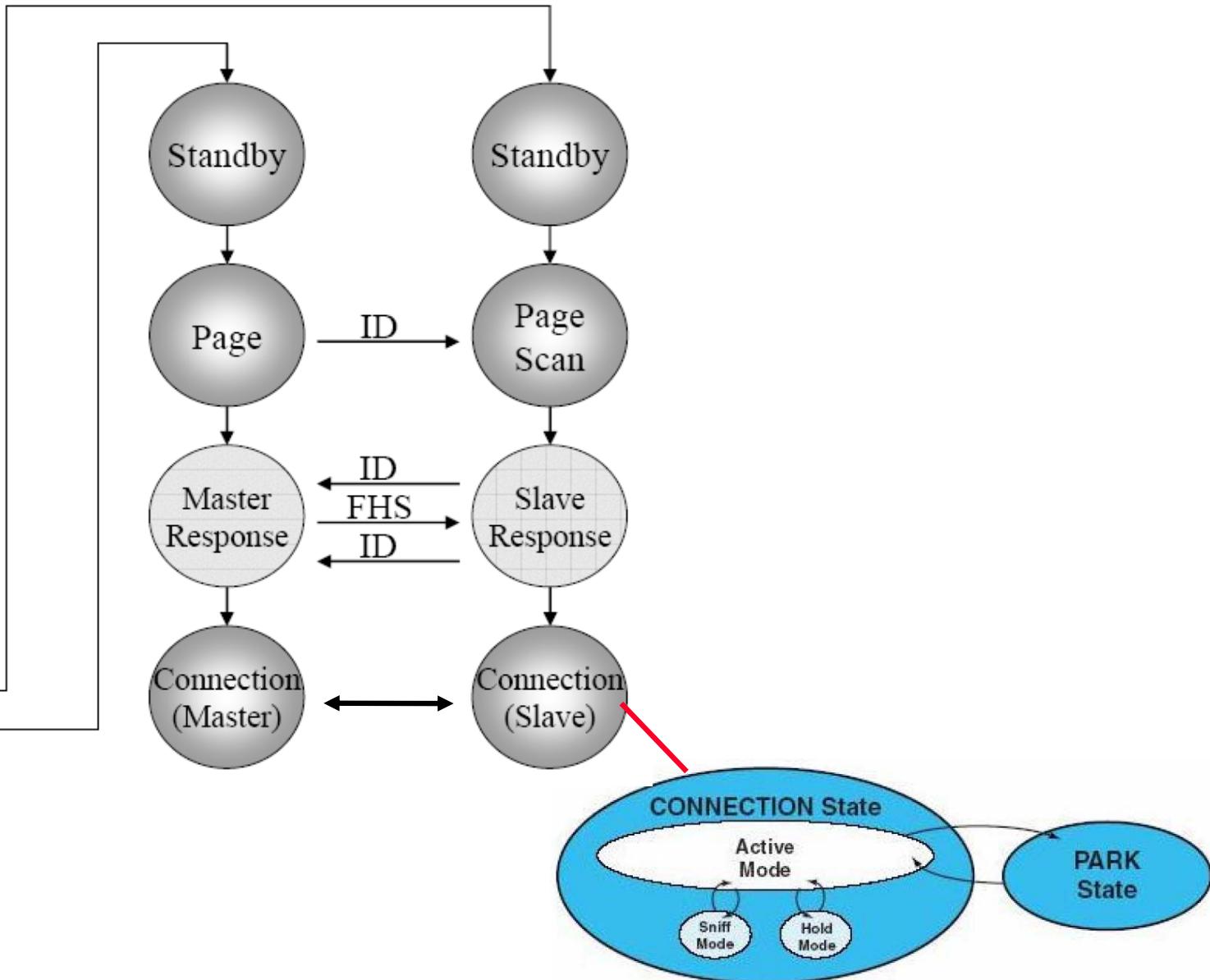
A slave in sniff state listens in regular time intervals whether there is a packet with its address. If yes, it answers.

From Standby to Connection

Master



Slave



The Page Mode

Synchronization between master and slave.
It is a prerequisite for establishing a connection.

page

page scan

master page
response

slave page
response

Master transmits its own and slave address to slave (it uses a special channel sequence)

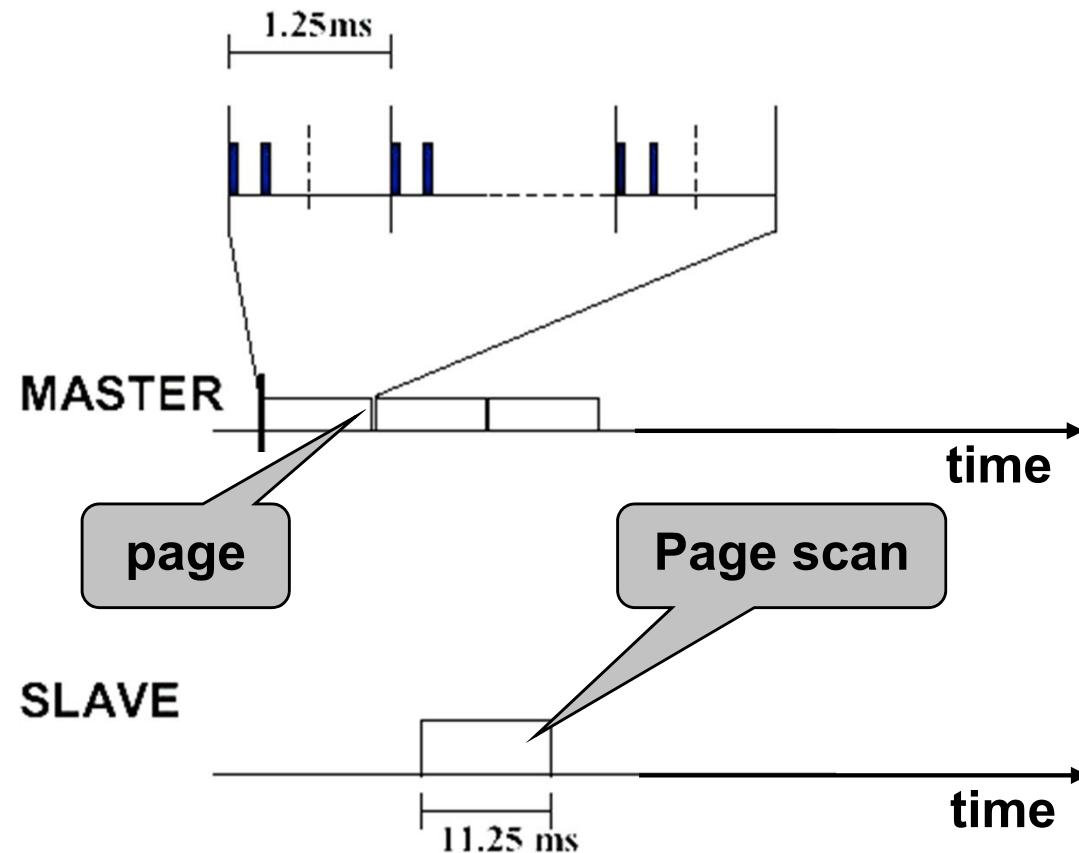
Slave listens, whether its own address is sent from a master.

Slave answers the master with its own address.

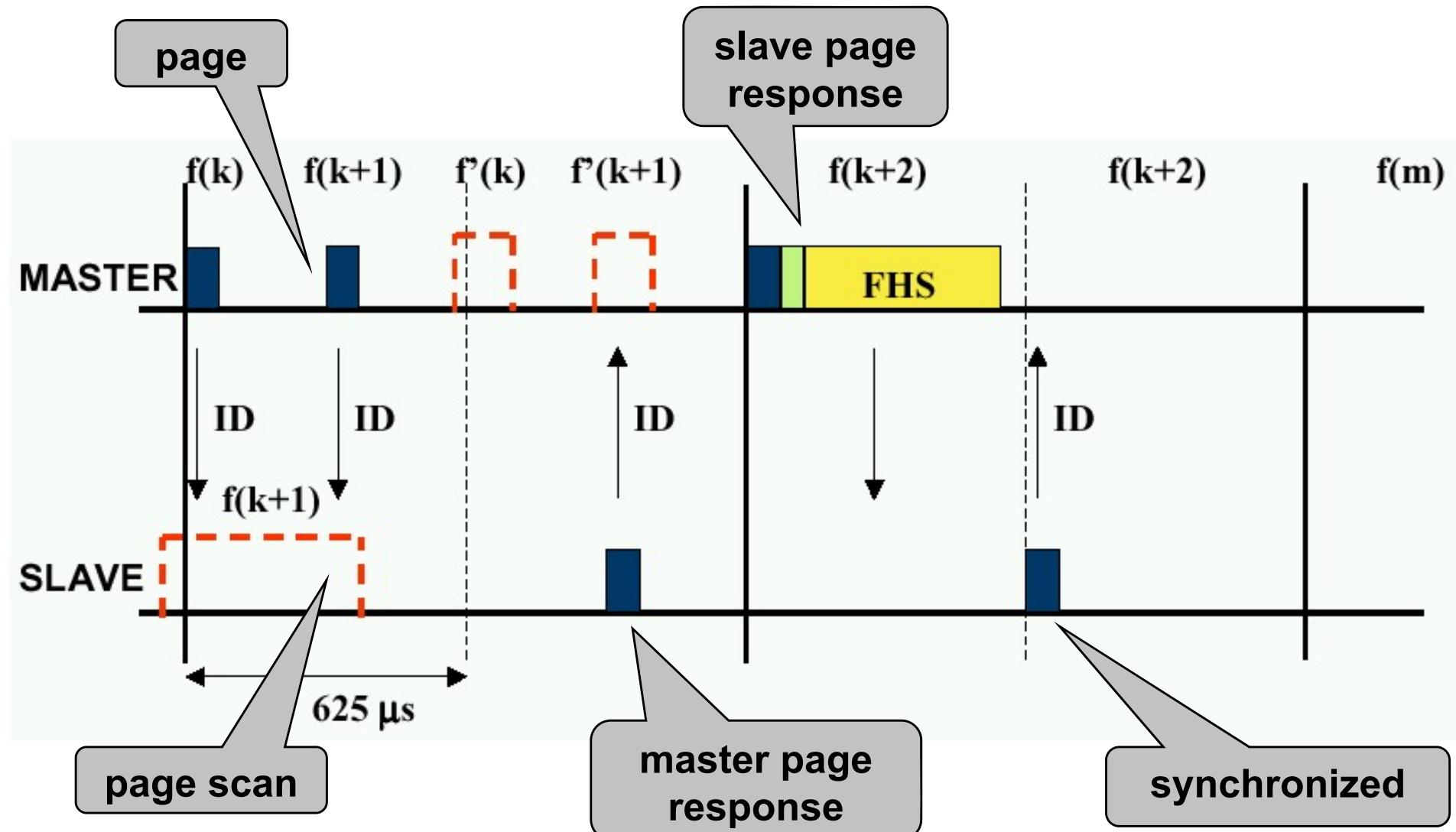
Master sends FHS-packet (frequency hop synchronization) to slave. It contains the channel sequence and the phase of the piconet.

Problem:
Synchronization

The Page Mode



The Page Mode



Protocol Hierarchy

- ▶ The **baseband specification** defines the packet formats, the physical and logical channels, the error correction, the synchronization between receiver and transmitter, and the different modes of operation and states that allow the transmission of data and audio.
- ▶ The **audio specification** defines the transmission of audio signals, in particular the coding and decoding methods.
- ▶ The **link manager** (LM) covers the authentication of a connection and the encryption, the management of a piconet (synchronous/asynchronous connection), the initiation of a connection (asynchronous/synchronous packet types, exchange of name and ID) and the transition between different modes of operation and states.



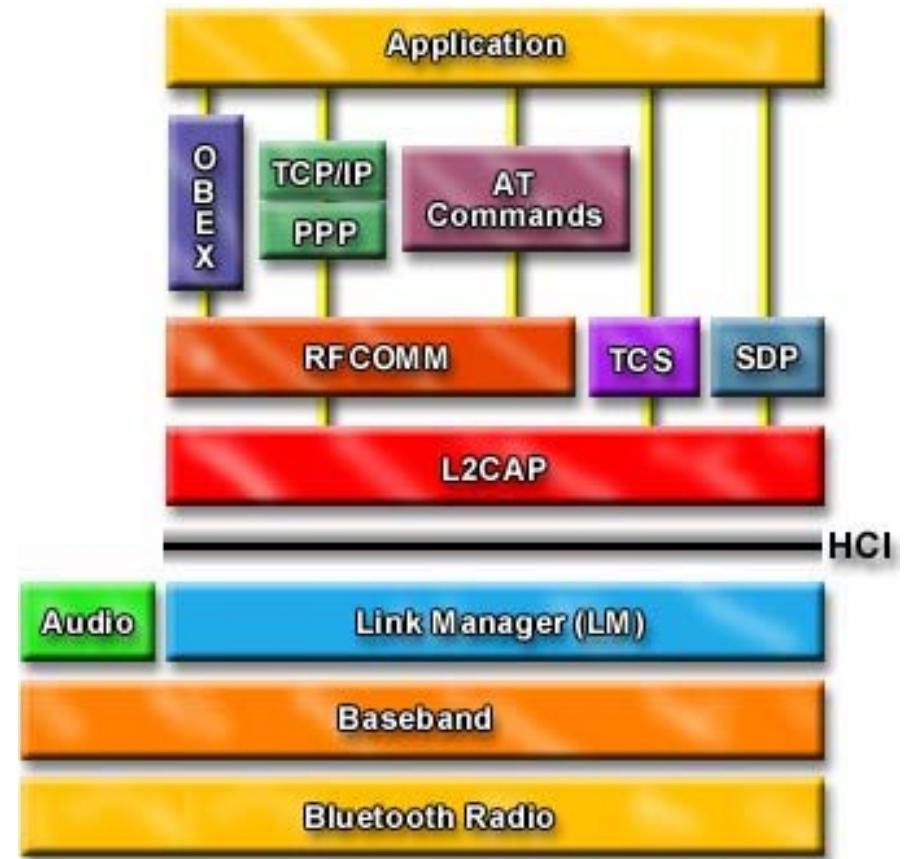
Protocol Hierarchy

- ▶ The **host controller interface (HCI)** defines a common standardized interface between a host and a bluetooth node; it is specified for several physical interconnections (USB, RS232, PCI, ...).
- ▶ The **link layer control and adaptation layer (L2CAP)** provides an abstract interface for data communication. It segments packets (up to 64kByte) and assembles them again, it allows the multiplexing of connections (simultaneous use of several protocols and connections) and allows the exchange of quality of service information between two nodes (packet rate, packet size, latency, delay variations, maximal rate).



Protocol Hierarchy

- ▶ **RFCOMM** is a simple transport protocol that simulates a serial connection (~RS 232).
- ▶ There are several other protocols that are defined such as the **telephony control protocol specification** (TCS), the **service discovery protocol** (SDP), the **OBEX** (Object Exchange Protocol), and **TCP/IP**.
- ▶ Finally, the **application** can use the top layers of the protocol stack.

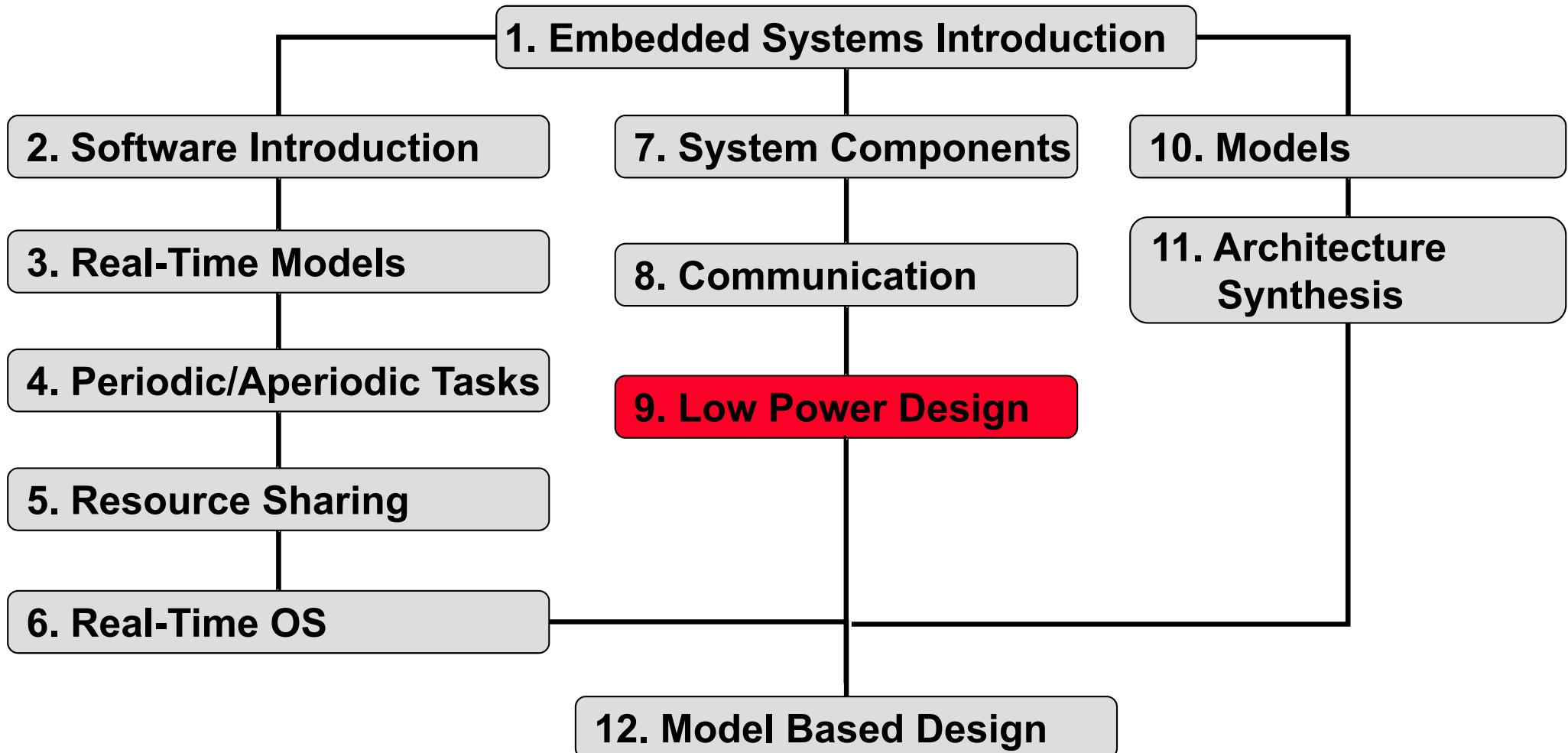


Embedded Systems

9. Low Power Design

Lothar Thiele

Contents of Course



*Software and
Programming*

*Processing and
Communication*

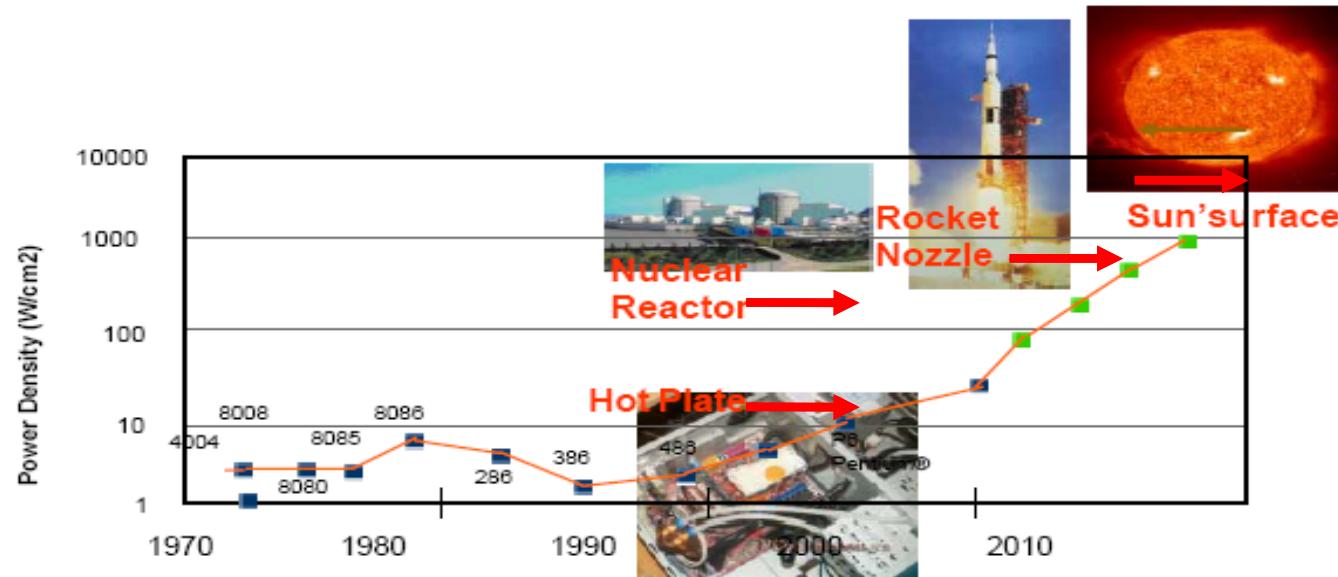
Hardware

Topics

- ▶ **General Remarks**
- ▶ Power and Energy
- ▶ Basic Techniques
 - Parallelism
 - VLIW (parallelism and reduced overhead)
 - Dynamic Voltage Scaling
 - Dynamic Power Management

Power and Energy Consumption

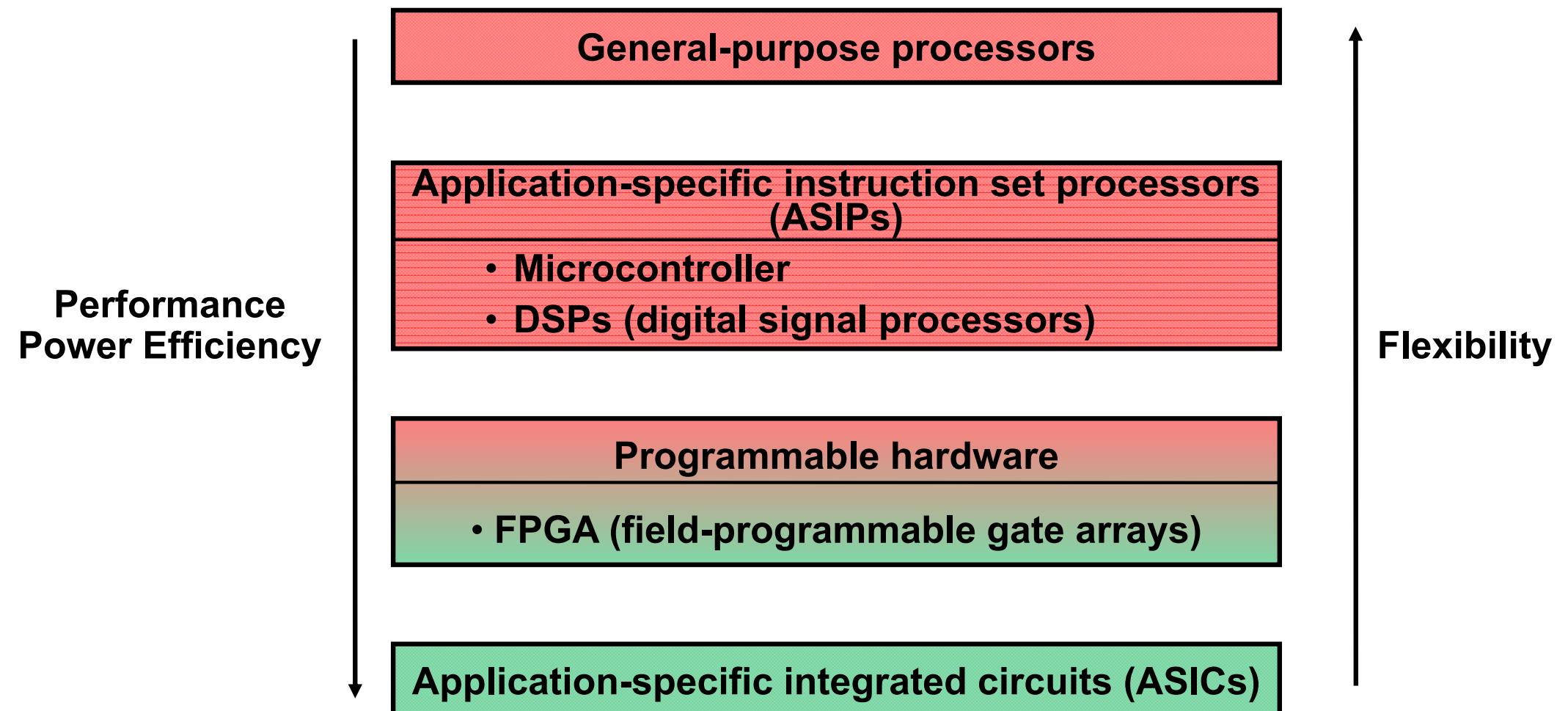
Need for efficiency (power and energy):



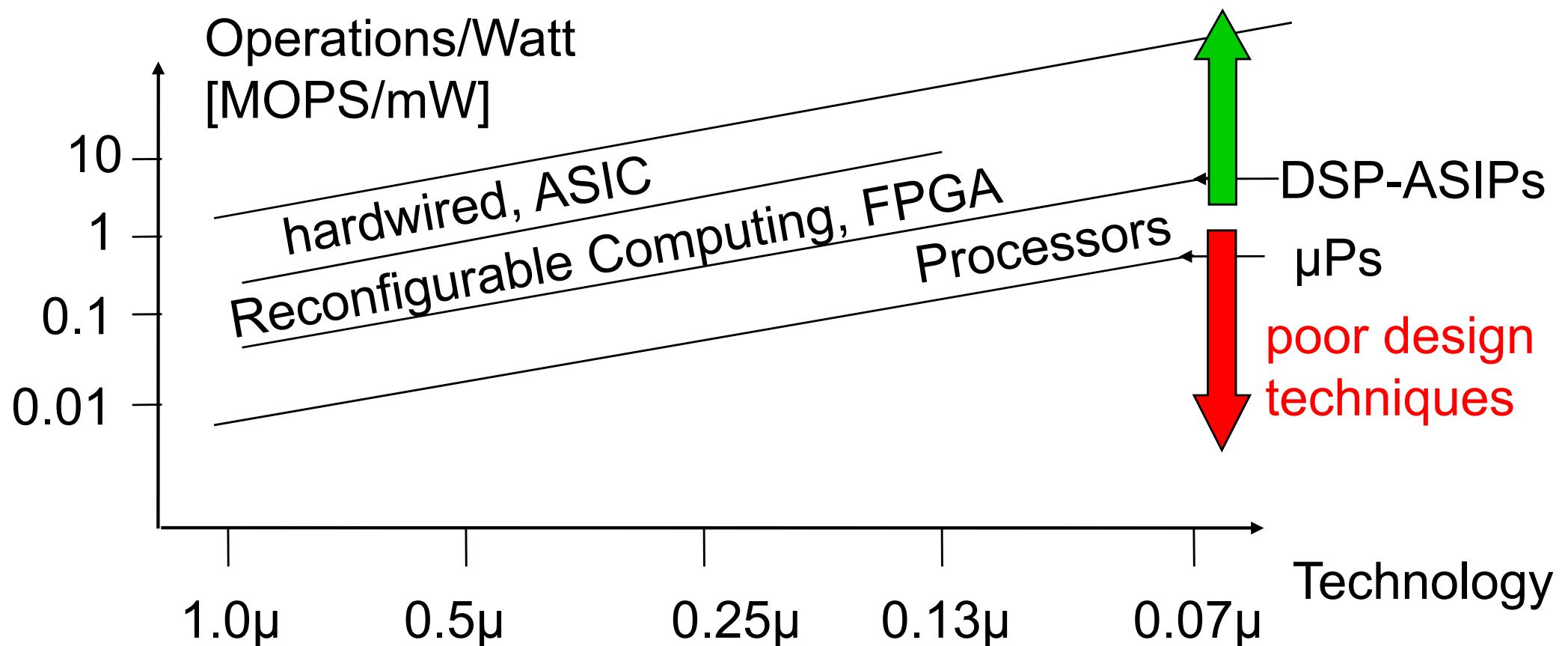
„Power is considered as the most important constraint in embedded systems.“ [in: L. Eggemont (ed): Embedded Systems Roadmap 2002, STW]

“Power demands are increasing rapidly, yet battery capacity cannot keep up.” [in Ditzel et al.: Power-Aware Architecting for data-dominated applications, 2007, Springer]

Implementation Alternatives

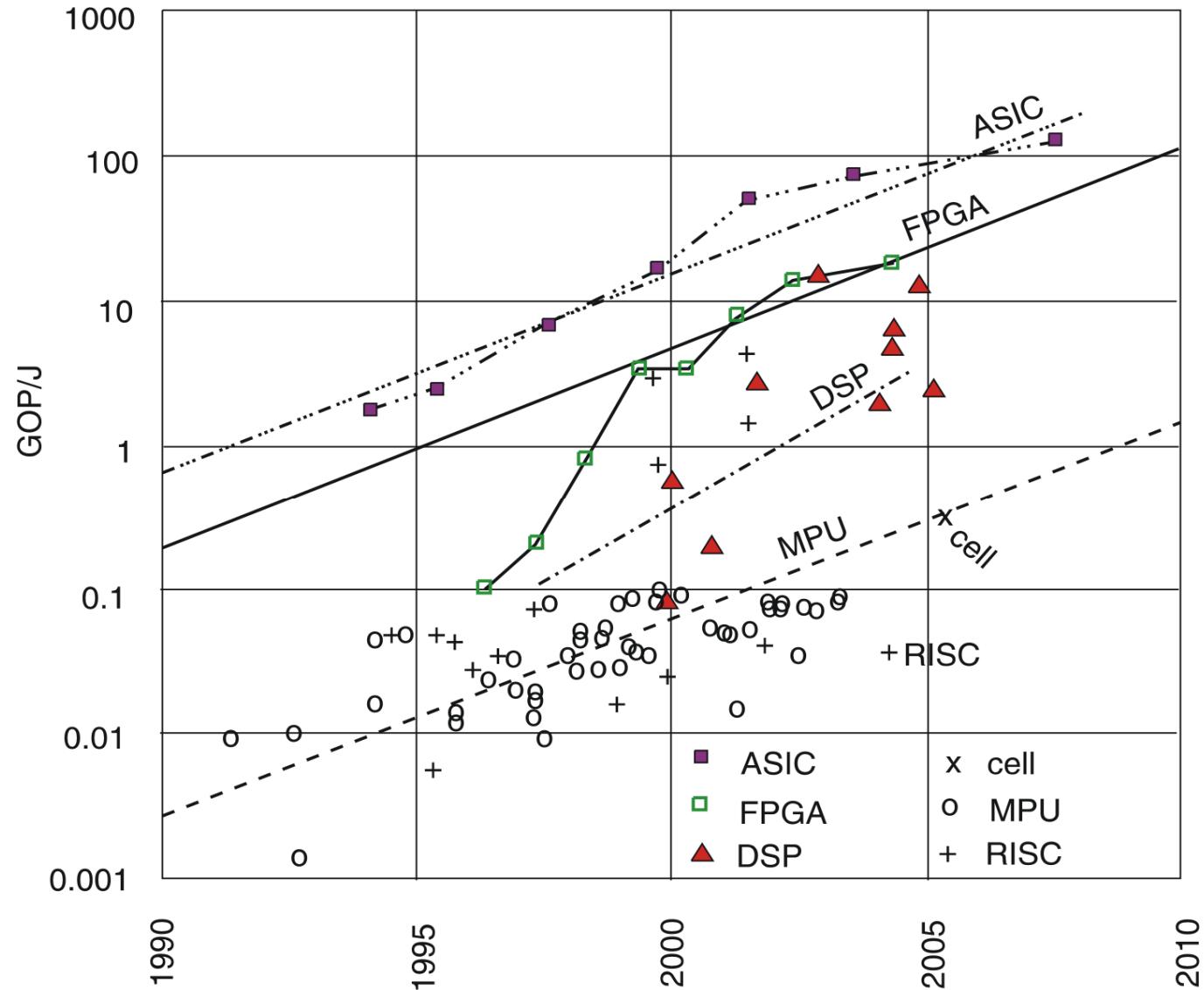


The Power/Flexibility Conflict



Necessary to *optimize HW and SW*.
Use *heterogeneous architectures*.
Apply *specialization techniques*.

Energy Efficiency

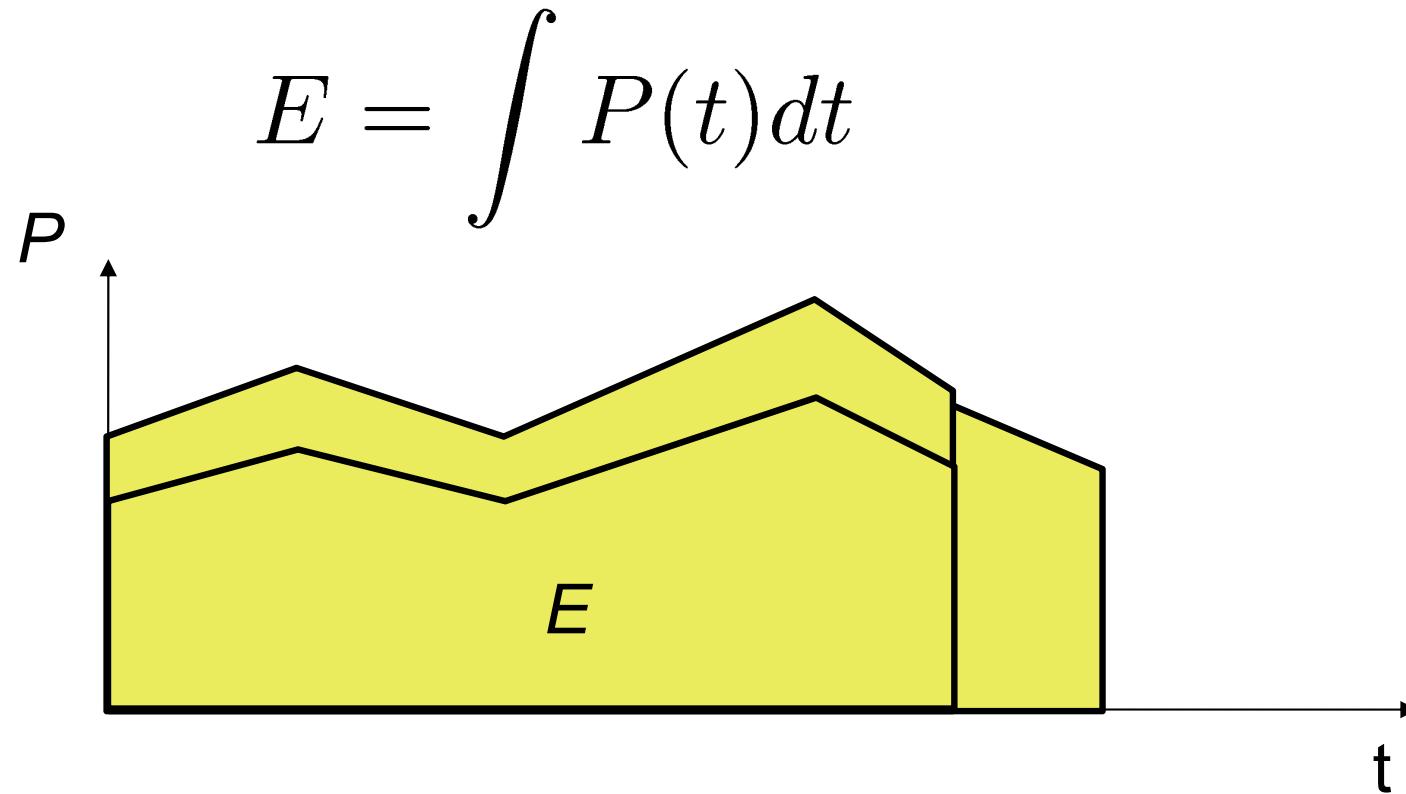


© Hugo De Man,
IMEC, Philips, 2007

Topics

- ▶ General Remarks
- ▶ *Power and Energy*
- ▶ Basic Techniques
 - Parallelism
 - VLIW (parallelism and reduced overhead)
 - Dynamic Voltage Scaling
 - Dynamic Power Management

Power and Energy are Related

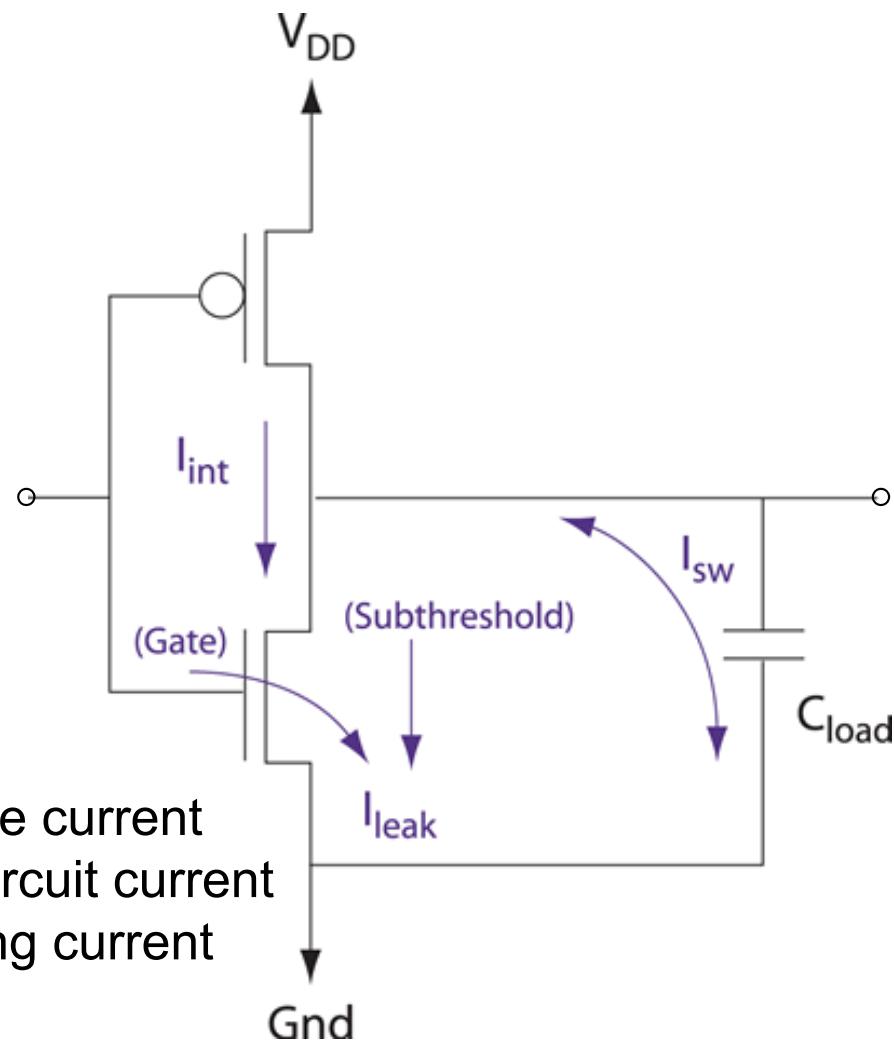


In many cases, faster execution also means less energy, but the opposite may be true if power has to be increased to allow faster execution.

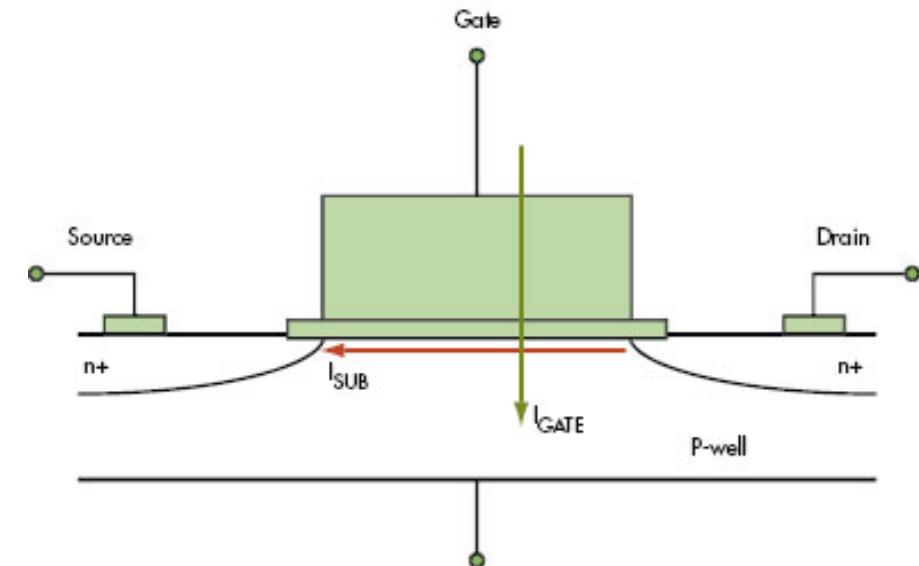
Low Power vs. Low Energy

- ▶ Minimizing the ***power consumption*** is important for
 - the design of the power supply
 - the design of voltage regulators
 - the dimensioning of interconnect
 - cooling (short term cooling)
 - high cost (estimated to be rising at \$1 to \$3 per Watt for heat dissipation [Skadron et al. ISCA 2003])
 - limited space
- ▶ Minimizing the ***energy consumption*** is important due to
 - restricted availability of energy (mobile systems)
 - limited battery capacities (only slowly improving)
 - very high costs of energy (solar panels, in space)
 - long lifetimes, low temperatures

Power Consumption of a CMOS Gate



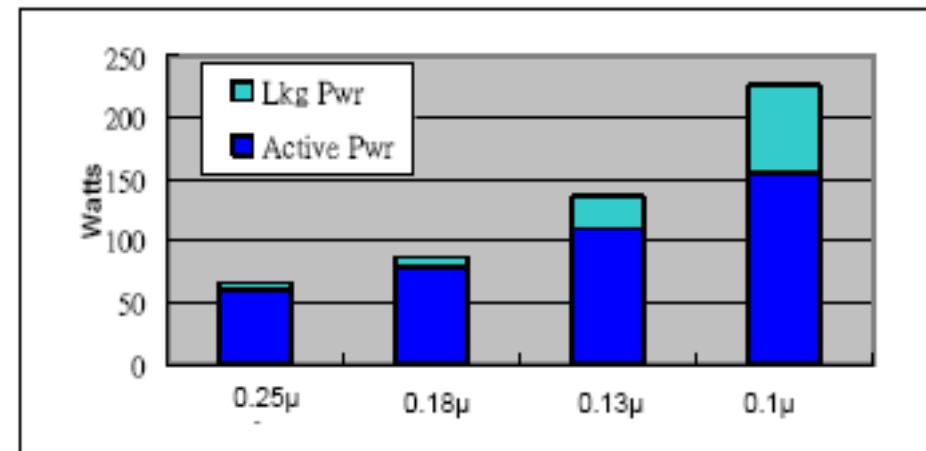
subthreshold and gate-oxide leakage



Power Consumption of CMOS Processors

► *Main sources:*

- Dynamic power consumption
 - charging and discharging capacitors
- Short circuit power consumption
 - short circuit path between supply rails during switching
- Leakage
 - leaking diodes and translators
 - becomes one of the major factors due to shrinking feature sizes in semiconductor technology



(Micro32 Keynotes by Fred Pollack)

Dynamic Voltage Scaling (DVS)

Power consumption of CMOS circuits (ignoring leakage):

$$P \sim \alpha C_L V_{dd}^2 f$$

V_{dd} : supply voltage

α : switching activity

C_L : load capacity

f : clock frequency

Delay for CMOS circuits:

$$\tau \sim C_L \frac{V_{dd}}{(V_{dd} - V_T)^2}$$

V_{dd} : supply voltage

V_T : threshold voltage

$$V_T \ll V_{dd}$$

Decreasing V_{dd} reduces P quadratically (f constant).

The gate delay increases reciprocally with decreasing V_{dd} .

Maximal frequency f_{\max} decreases linearly with decreasing V_{dd} .

Potential for Energy Optimization: DVS

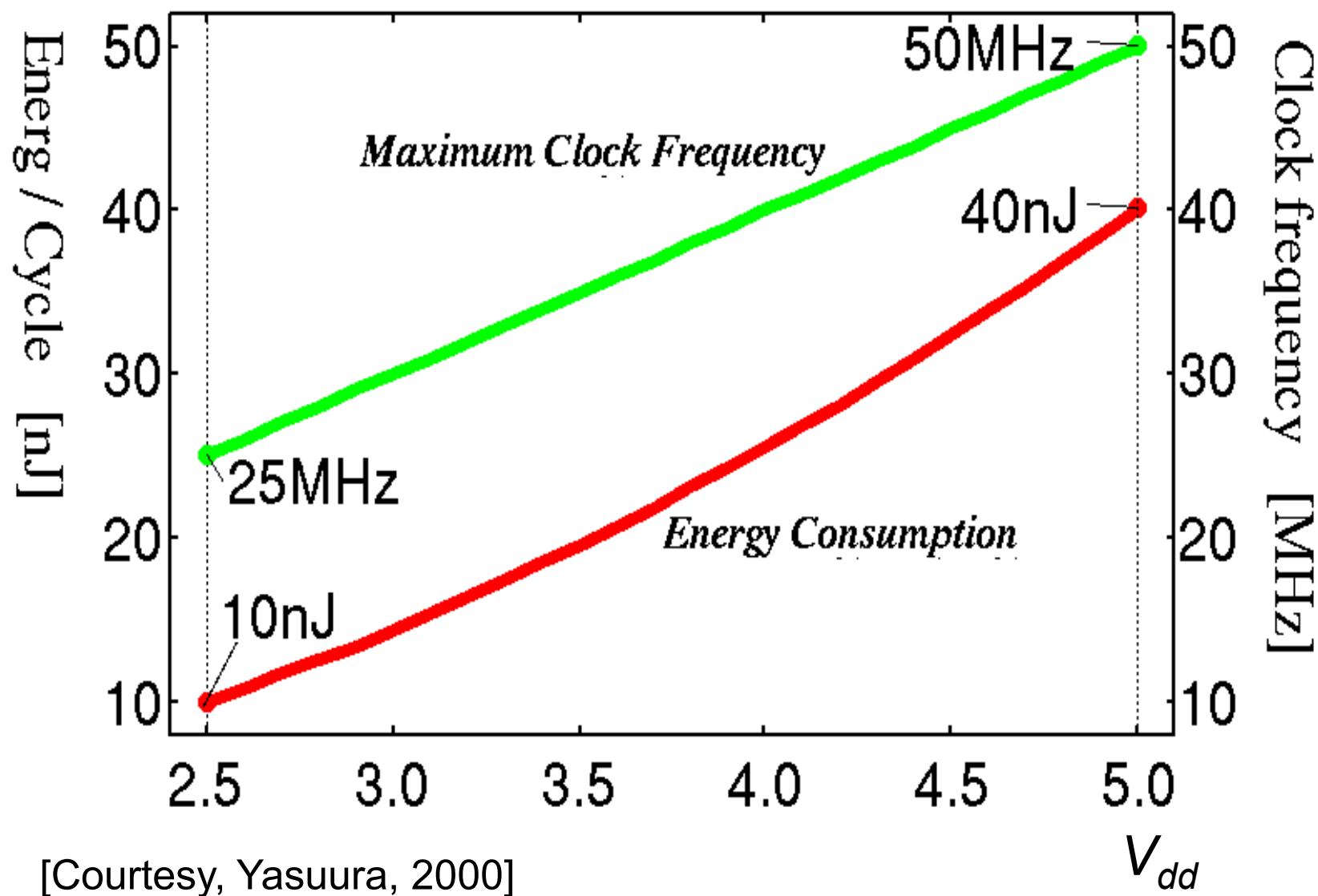
$$P \sim \alpha C_L V_{dd}^2 f$$

$$E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 (\#cycles)$$

Saving energy for a given task:

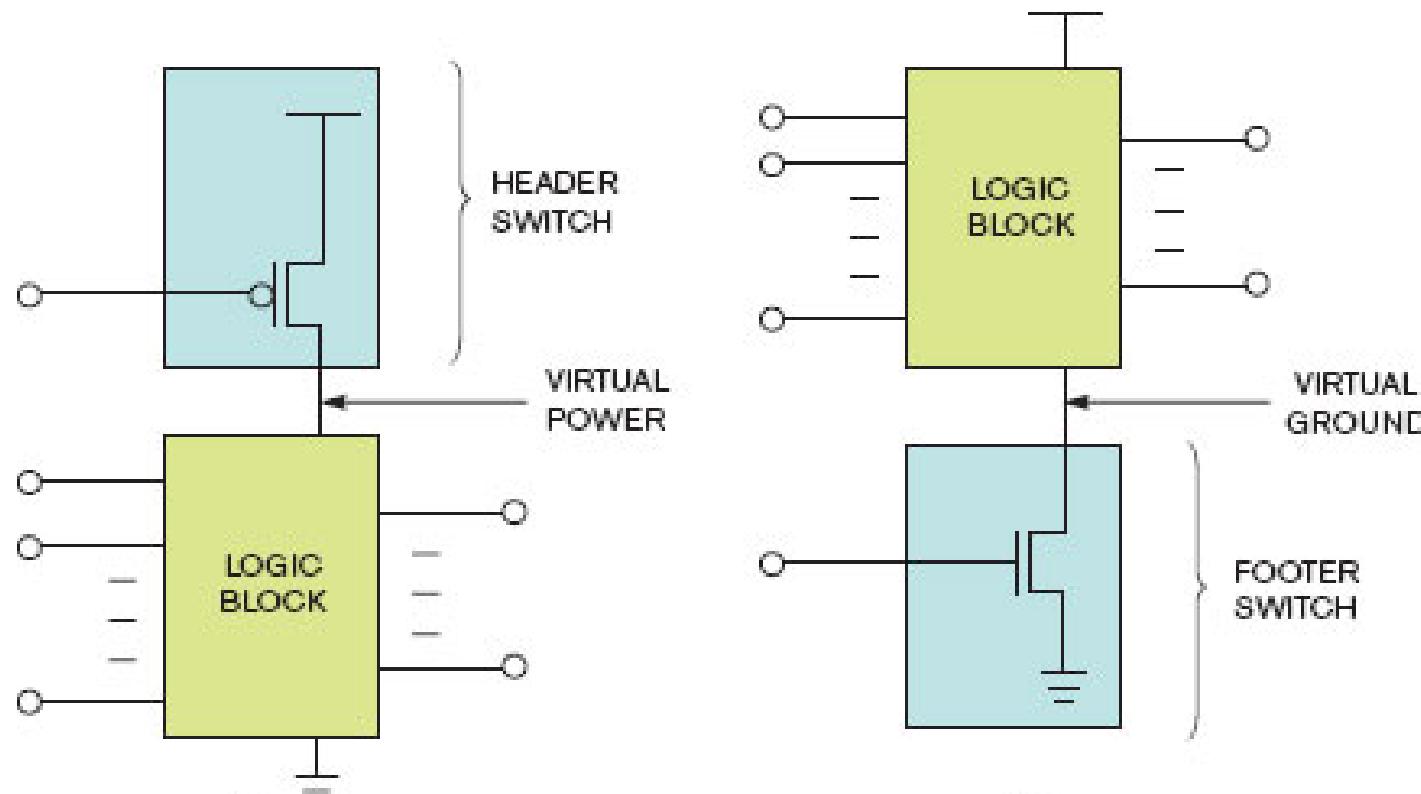
- Reduce the supply voltage V_{dd}
- Reduce switching activity α
- Reduce the load capacitance C_L
- Reduce the number of cycles $\#cycles$

Example: Voltage Scaling



Power Supply Gating

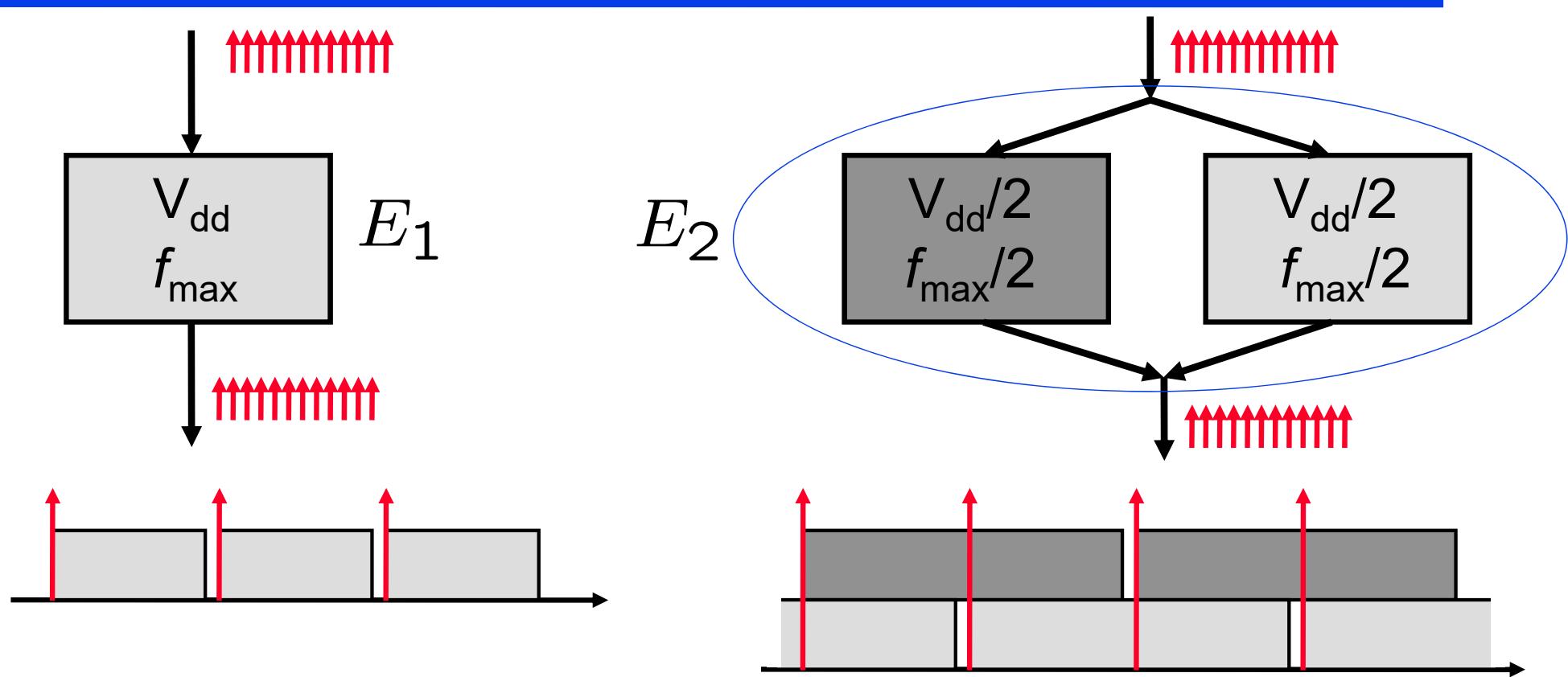
- ▶ Power gating is one of the most effective ways of minimizing static power consumption (leakage)
 - Cut-off power supply to inactive units/components
 - Reduces leakage



Topics

- ▶ General Remarks
- ▶ Power and Energy
- ▶ Basic Techniques
 - ***Parallelism***
 - VLIW (parallelism and reduced overhead)
 - Dynamic Voltage Scaling
 - Dynamic Power Management

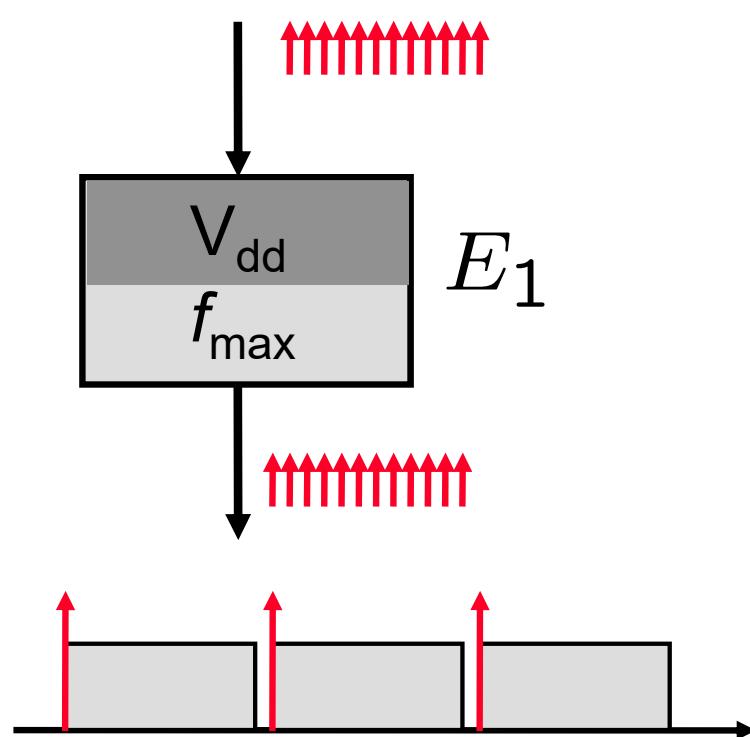
Use of Parallelism



$$E \sim V_{dd}^2 (\# \text{cycles})$$

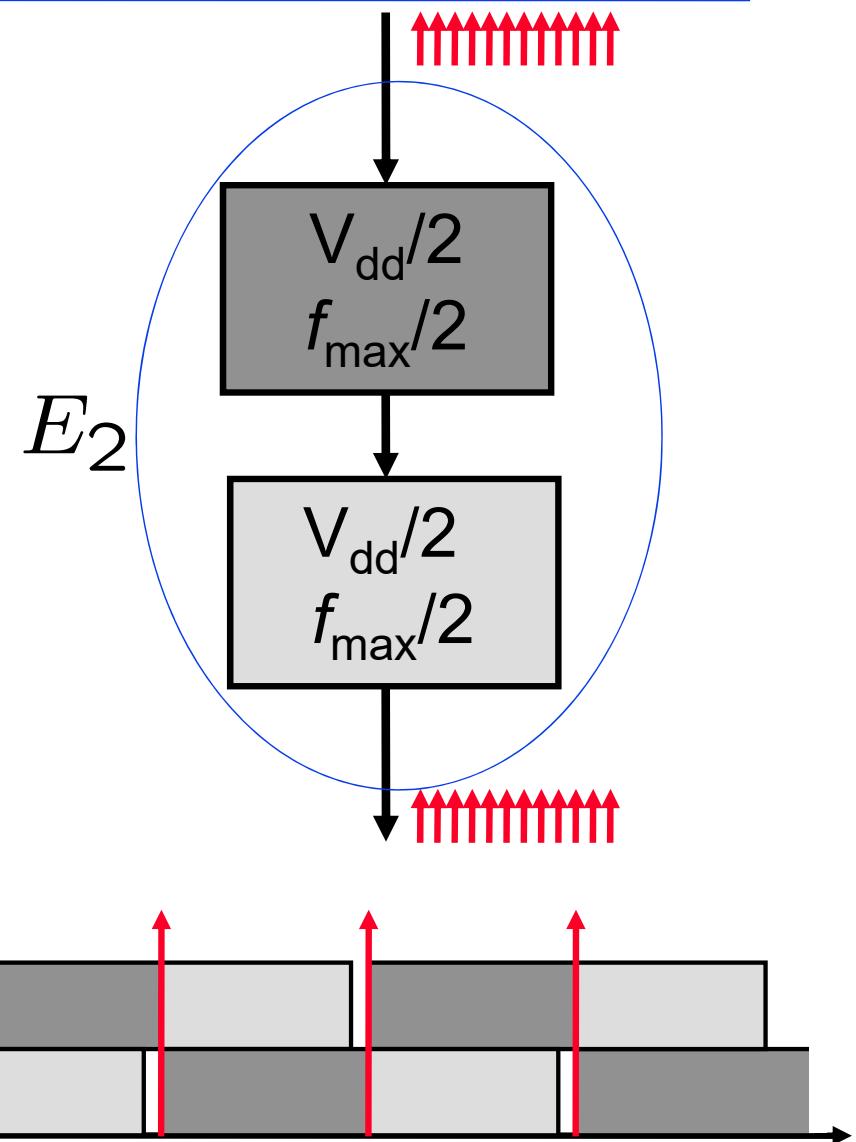
$$E_2 = \frac{1}{4} E_1$$

Use of Pipelining



$$E \sim V_{dd}^2 (\# \text{cycles})$$

$$E_2 = \frac{1}{4} E_1$$

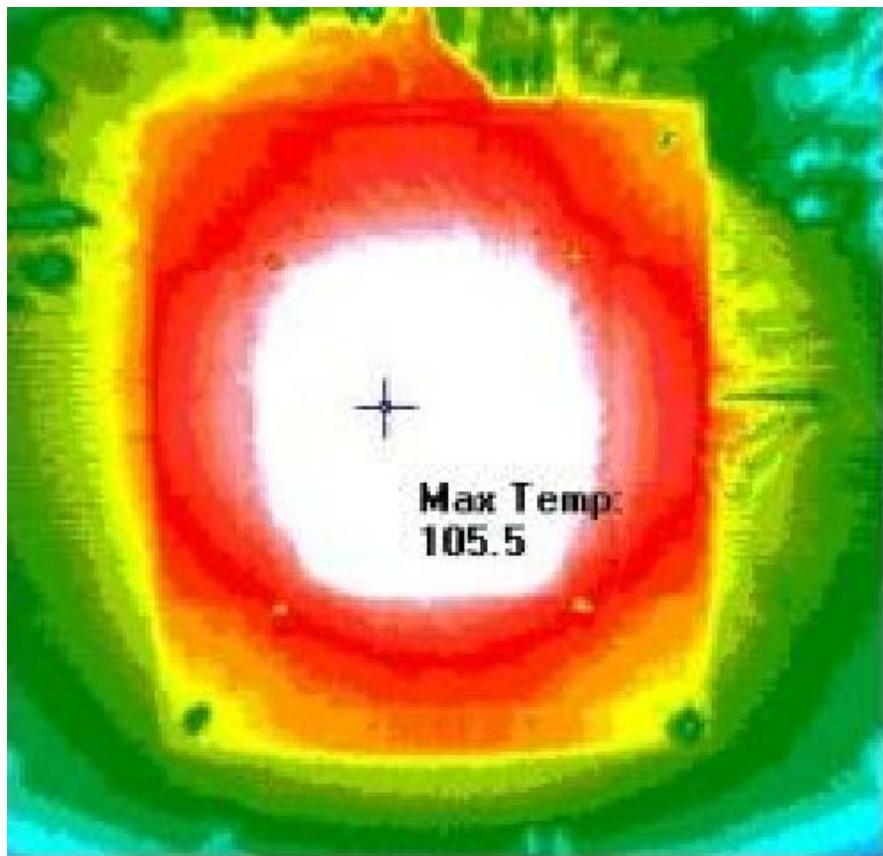


Topics

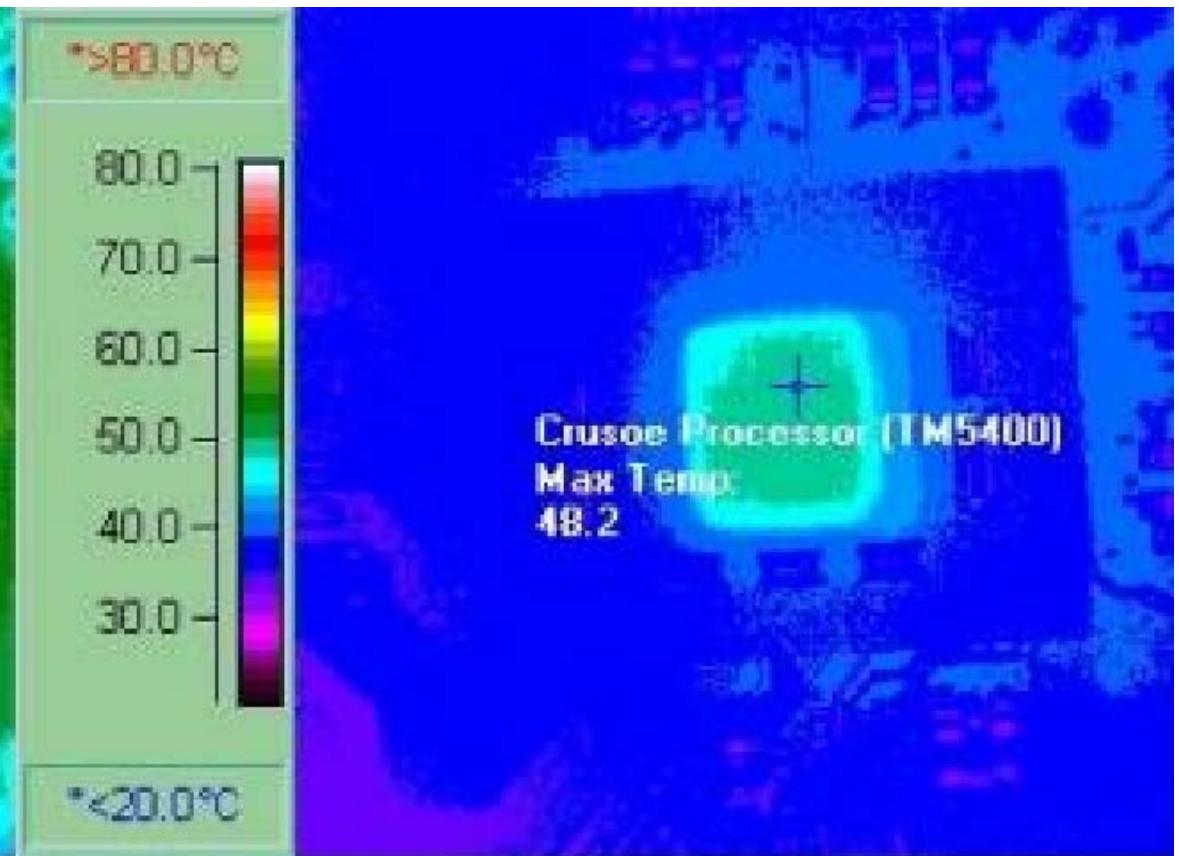
- ▶ General Remarks
- ▶ Power and Energy
- ▶ Basic Techniques
 - Parallelism
 - **VLIW (*parallelism and reduced overhead*)**
 - Dynamic Voltage Scaling
 - Dynamic Power Management

New ideas help ...

Pentium



Crusoe

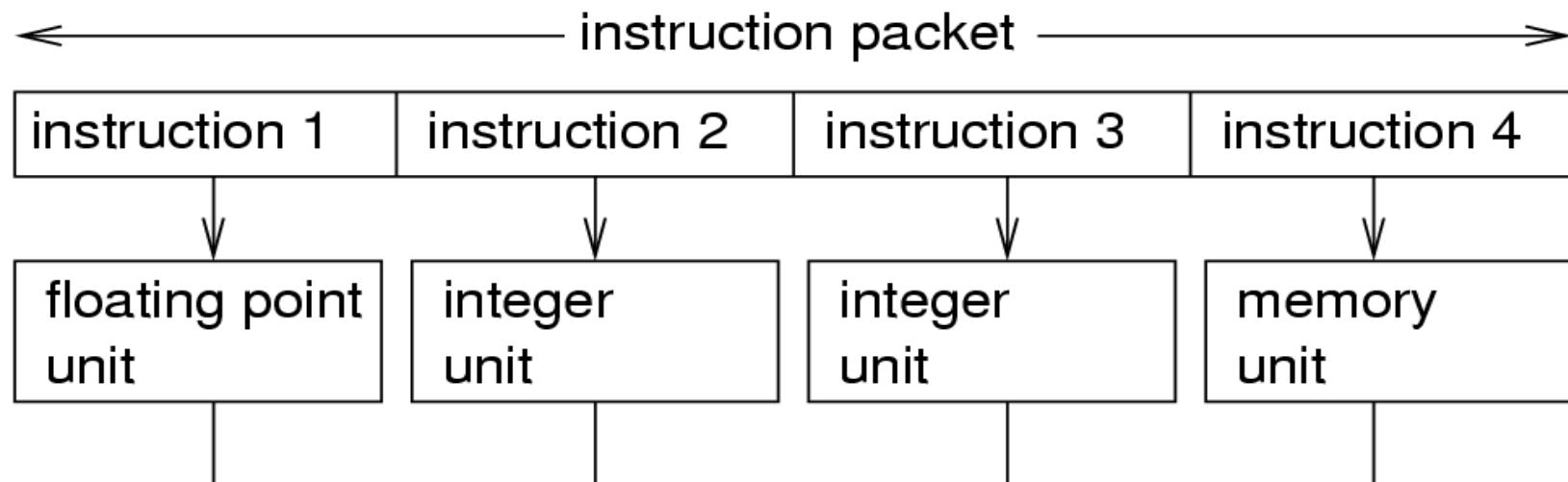


Running the same multimedia application.

As published by Transmeta [www.transmeta.com]

VLIW Architectures

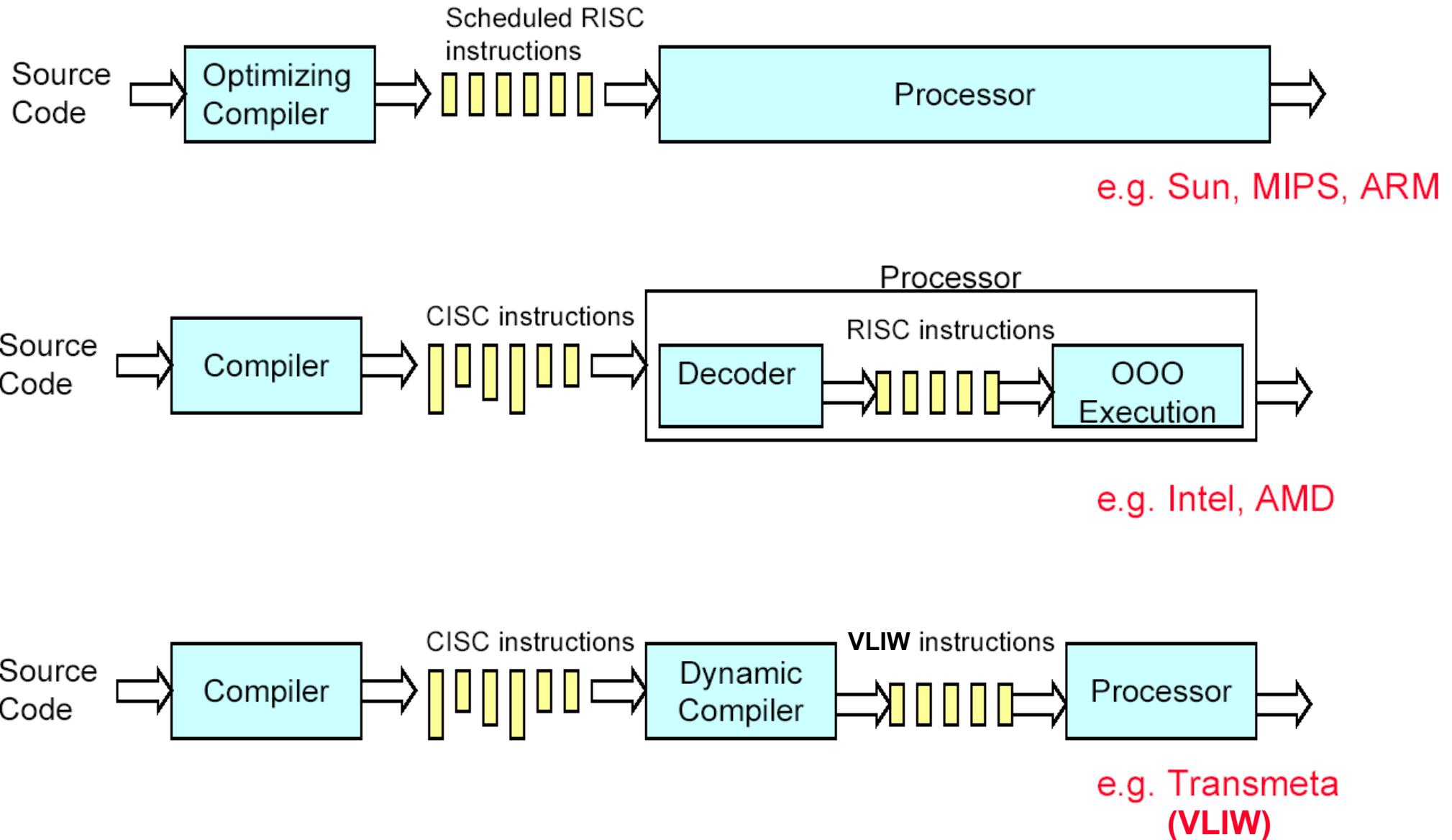
- ▶ ***Large degree of parallelism***
 - many computational units, (deeply) pipelined
- ▶ ***Simple hardware architecture***
 - explicit parallelism (parallel instruction set)
 - parallelization is done offline (compiler)



Transmeta was a typical VLIW Architecture

- 128-bit instructions (bundles):
 - 4 operations per instruction
 - 2 combinations of instructions allowed
- Register files
 - 64 integer, 32 floating point
- Some interesting features
 - 6 stage pipeline (2x fetch, decode, register read, execute, write)
 - x86 ISA execution using software techniques
 - Skip the binary compatibility problem!!
 - Interpretation and just-in-time binary translation
 - Speculation support

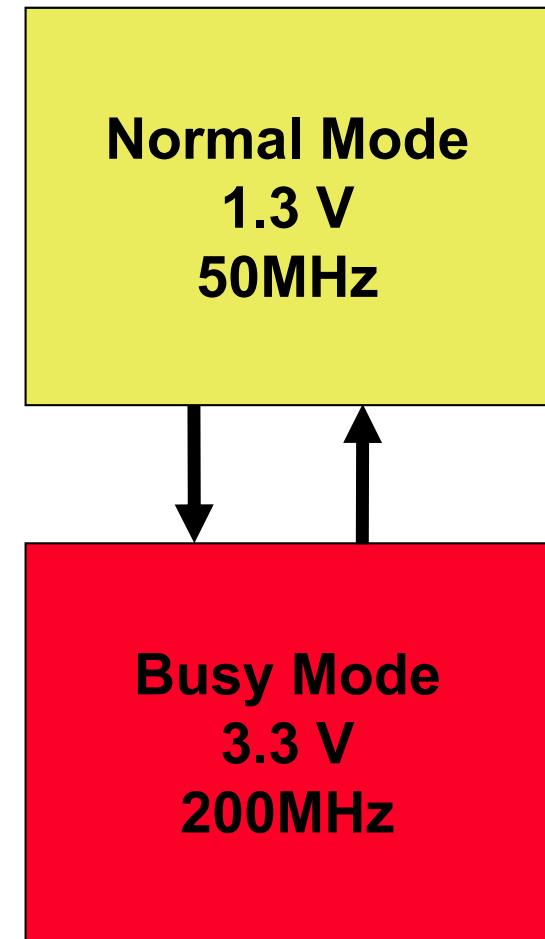
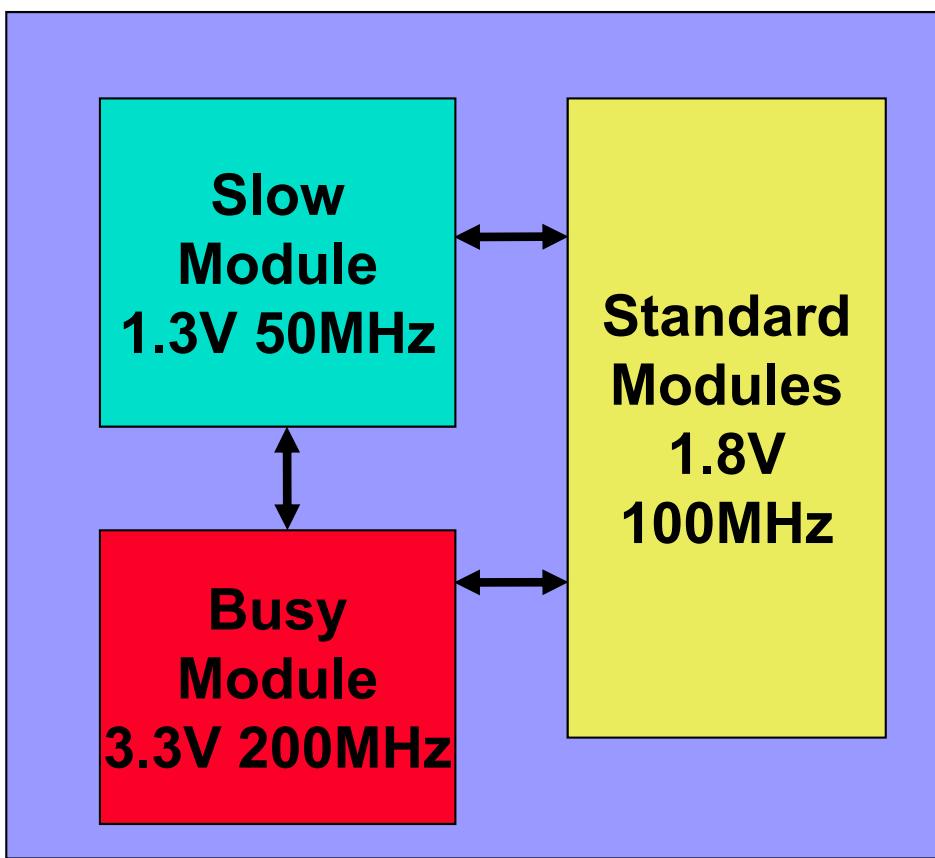
Transmeta



Topics

- ▶ General Remarks
- ▶ Power and Energy
- ▶ Basic Techniques
 - Parallelism
 - VLIW (parallelism and reduced overhead)
 - ***Dynamic Voltage Scaling***
 - Dynamic Power Management

Spatial vs. Dynamic Voltage Management



Not all components require same performance.

Required performance may change over time

Potential for Energy Optimization: DVS

$$P \sim \alpha C_L V_{dd}^2 f$$

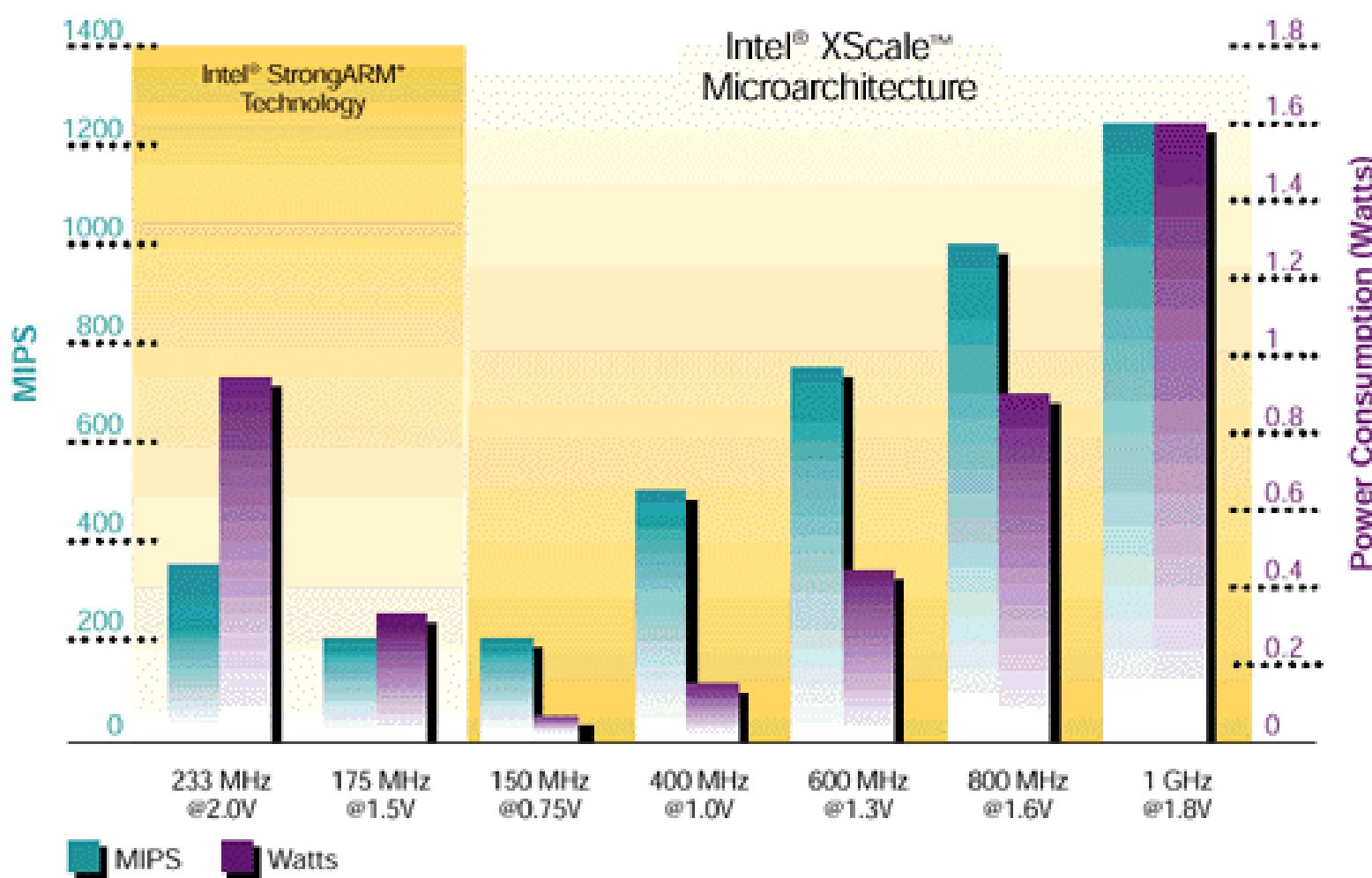
$$E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 (\#cycles)$$

Saving energy for a given task:

- Reduce the supply voltage V_{dd}
- Reduce switching activity α
- Reduce the load capacitance C_L
- Reduce the number of cycles $\#cycles$

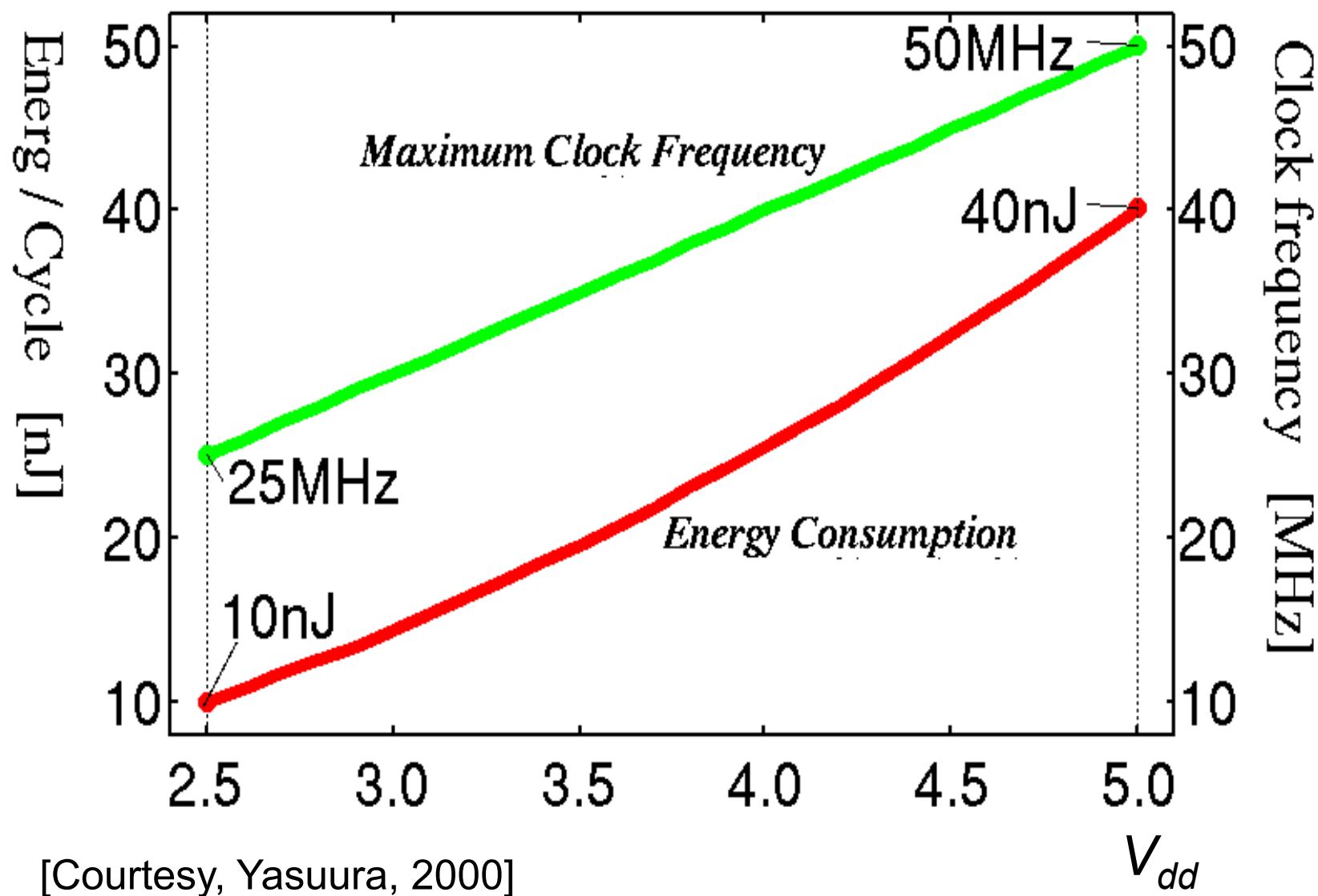
Example: INTEL Xscale

POWER-PERFORMANCE COMPARISON



OS should schedule distribution of the energy budget.

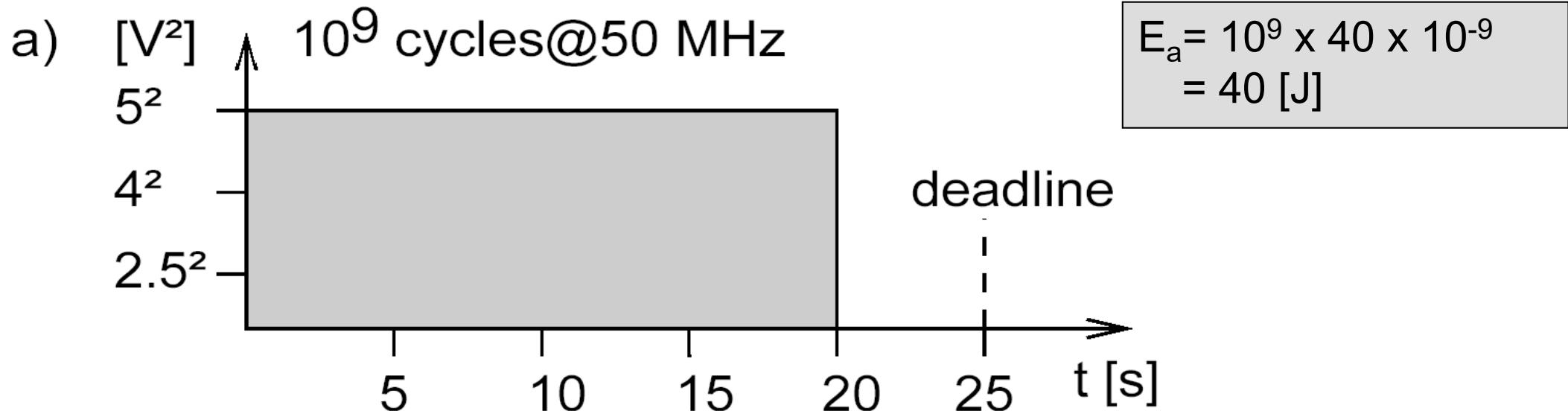
Example: Voltage Scaling



DVS Example: a) Complete task ASAP

V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

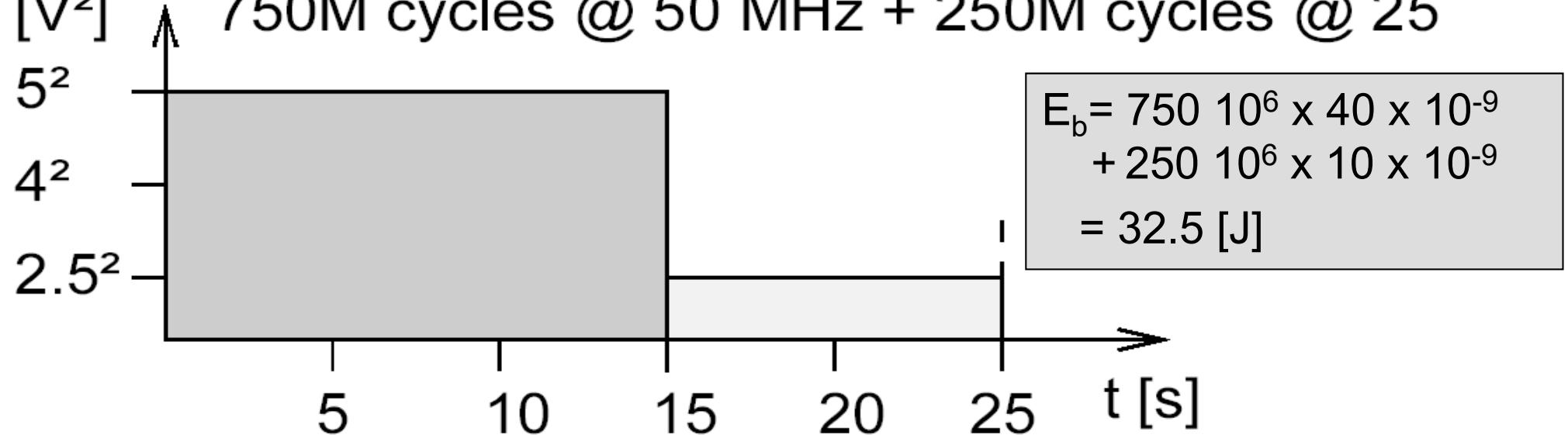
Task that needs to execute 10^9 cycles within 25 seconds.



DVS Example: b) Two voltages

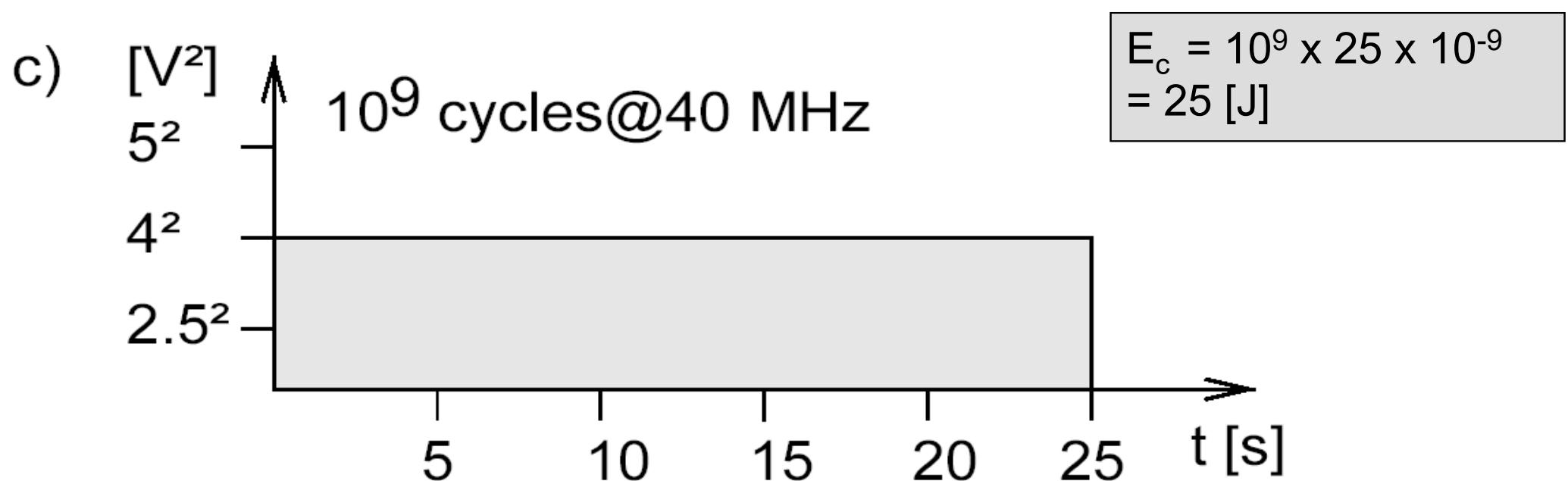
V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40

b) $[V^2]$ 750M cycles @ 50 MHz + 250M cycles @ 25

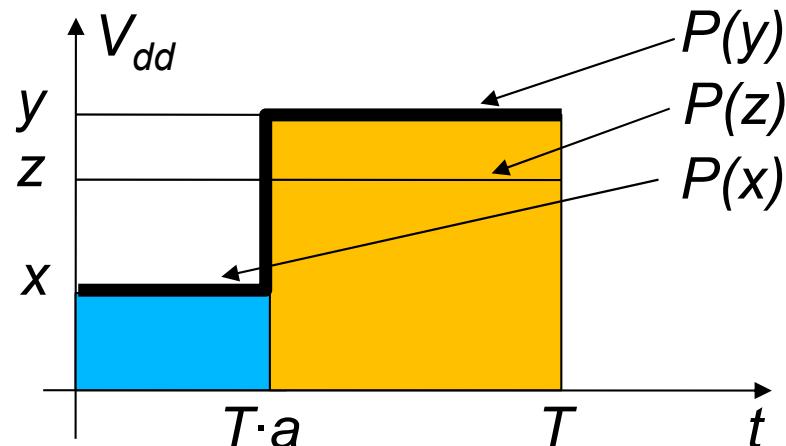


DVS Example: c) Optimal Voltage

V_{dd} [V]	5.0	4.0	2.5
Energy per cycle [nJ]	40	25	10
f_{max} [MHz]	50	40	25
cycle time [ns]	20	25	40



DVS: Optimal Strategy



$$z = a \cdot x + (1-a) \cdot y$$

Execute task in fixed time T with variable voltage $V_{dd}(t)$:

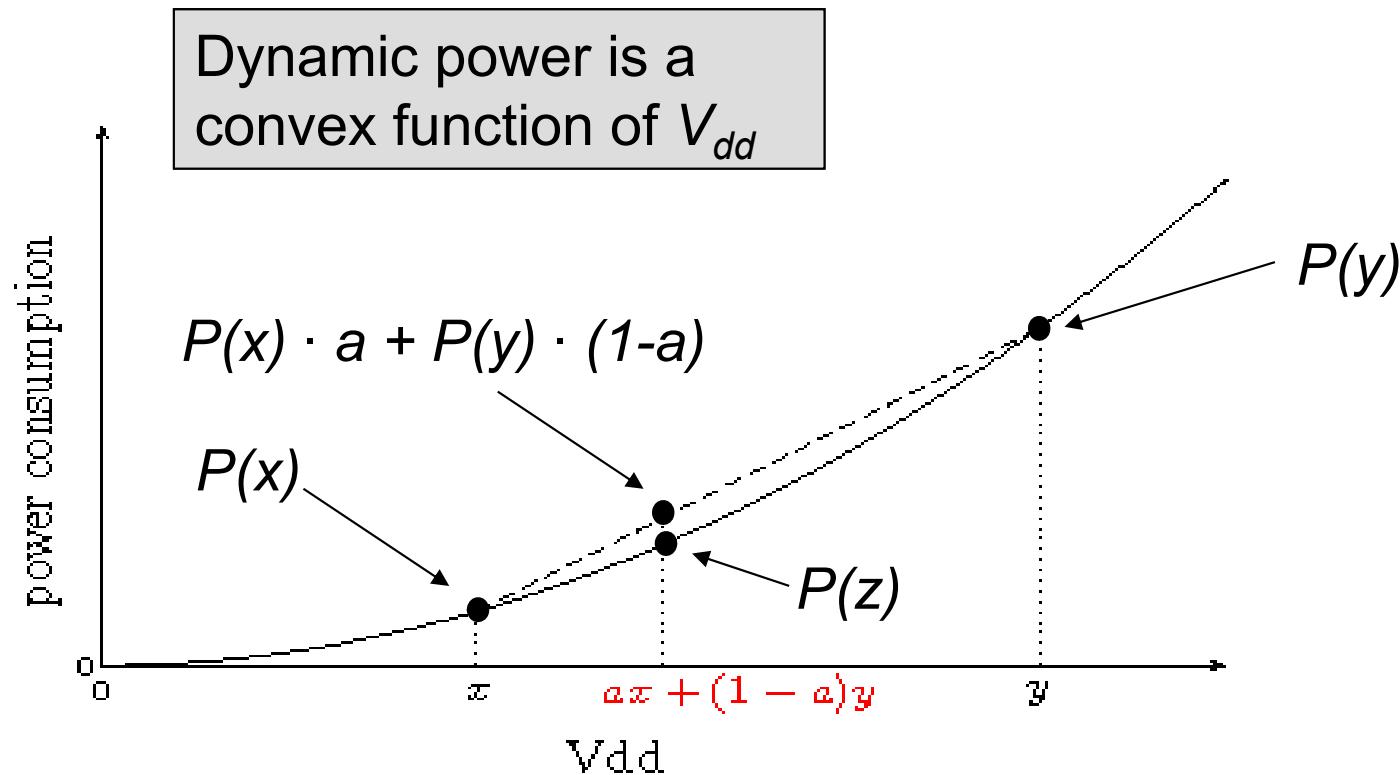
gate delay: $\tau \sim \frac{1}{V_{dd}}$

execution rate: $f(t) \sim V_{dd}(t)$

invariant: $\int V_{dd}(t) dt = \text{const.}$

- ▶ **case A:** execute at voltage x for $T \cdot a$ time units and at voltage y for $(1-a) \cdot T$ time units;
energy consumption $T \cdot (P(x) \cdot a + P(y) \cdot (1-a))$
- ▶ **case B:** execute at voltage $z = a \cdot x + (1-a) \cdot y$ for T time units;
energy consumption $T \cdot P(z)$

DVS: Optimal Strategy



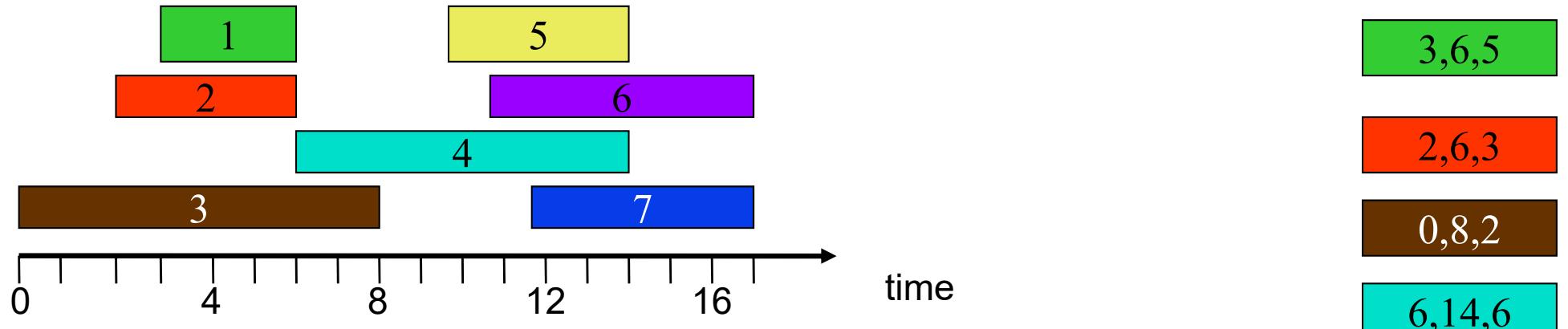
- ▶ If possible, running at a constant frequency (voltage) minimizes the energy consumption for dynamic voltage scaling:
 - **case A** is always worse if the power consumption is a convex function of the supply voltage

DVS: Offline Scheduling on One Processor

- ▶ Let us model a set of independent tasks as follows:
 - We suppose that a task $v_i \in V$
 - requires c_i computation time at normalized processor frequency 1
 - arrives at time a_i
 - has (absolute) deadline constraint d_i
- ▶ How do we schedule these tasks such that all these tasks can be finished ***no later than their deadlines*** and the energy consumption is ***minimized***?
 - YDS Algorithm from “A Scheduling Model for Reduce CPU Energy”, Frances Yao, Alan Demers, and Scott Shenker, FOCS 1995.”

If possible, running at a constant frequency (voltage) minimizes the energy consumption for dynamic voltage scaling.

YDS Algorithm for Offline Scheduling



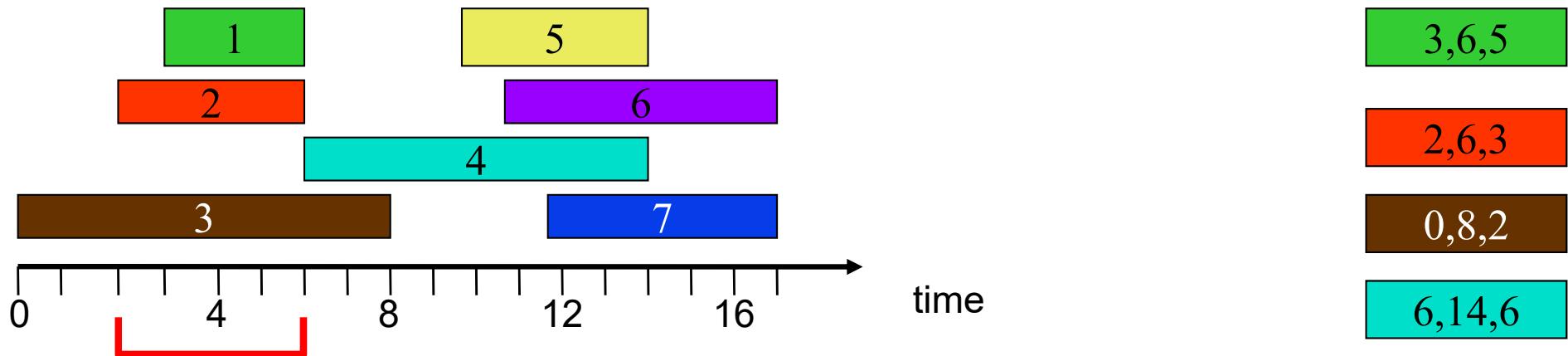
- ▶ Define **intensity** $G([z, z'])$ in some time interval $[z, z']$:
 - average accumulated execution time of all tasks that have arrival and deadline in $[z, z']$ relative to the length of the interval $z'-z$

$$V'([z, z']) = \{v_i \in V : z \leq a_i < d_i \leq z'\}$$

$$G([z, z']) = \sum_{v_i \in V'([z, z'])} c_i / (z' - z)$$

YDS Algorithm for Offline Scheduling

- ▶ **Step 1:** Execute jobs in the interval with the highest intensity by using the earliest-deadline first schedule and running at the intensity as the frequency.



$$G([0,6]) = (5+3)/6 = 8/6, G([0,8]) = (5+3+2)/(8-0) = 10/8,$$

$$G([0,14]) = (5+3+2+6+6)/14 = 11/7, G([0,17]) = (5+3+2+6+6+2+2)/17 = 26/17$$

$$G([2, 6]) = (5+3)/(6-2) = 2, G([2, 14]) = (5+3+6+6)/(14-2) = 5/3,$$

$$G([2, 17]) = (5+3+6+6+2+2)/15 = 24/15$$

$$G([3, 6]) = 5/3, G([3, 14]) = (5+6+6)/(14-3) = 17/11, G([3, 17]) = (5+6+6+2+2)/14 = 21/14$$

$$G([6, 14]) = 12/(14-6) = 12/8, G([6, 17]) = (6+6+2+2)/(17-6) = 16/11$$

$$G([10, 14]) = 6/4, G([10, 17]) = 10/7, G([11, 17]) = 4/6, G([12, 17]) = 2/5$$

3,6,5

2,6,3

0,8,2

6,14,6

10,14,6

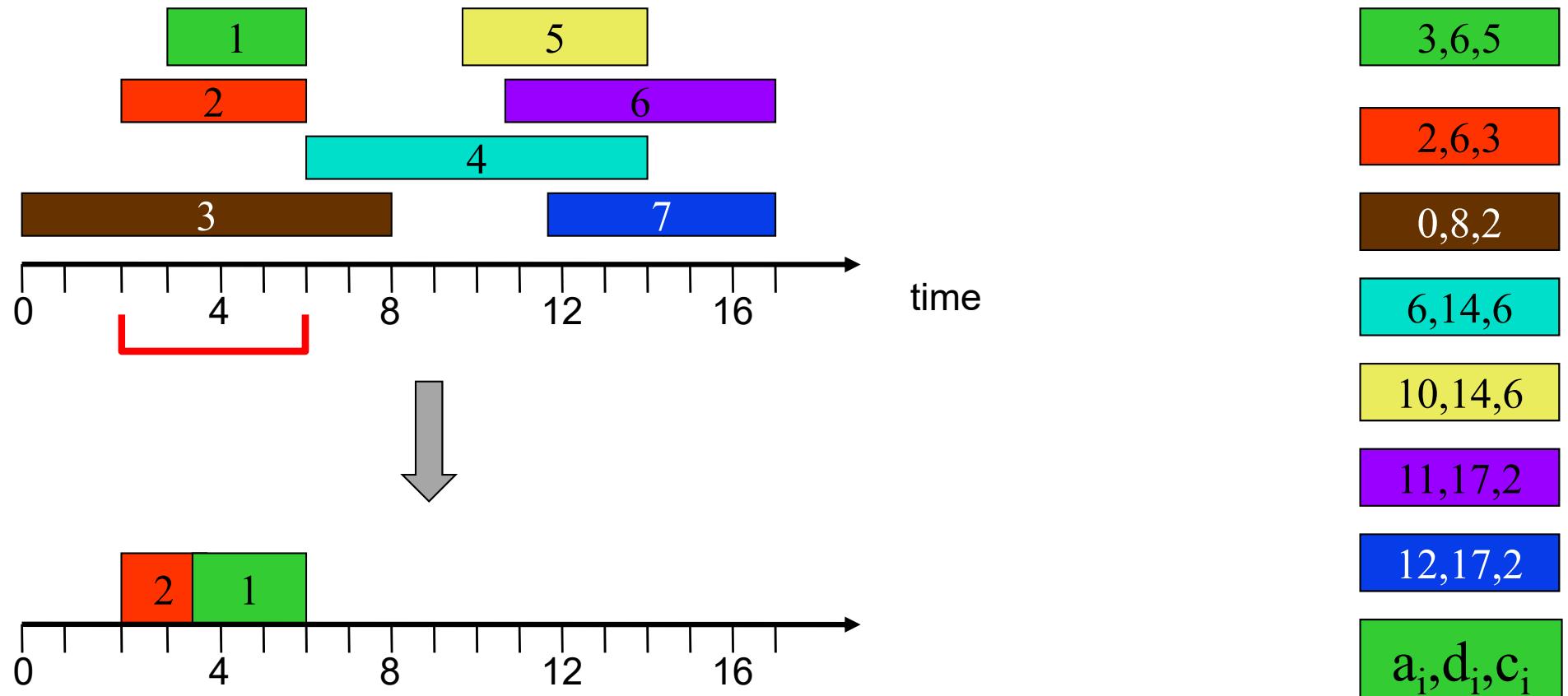
11,17,2

12,17,2

a_i, d_i, c_i

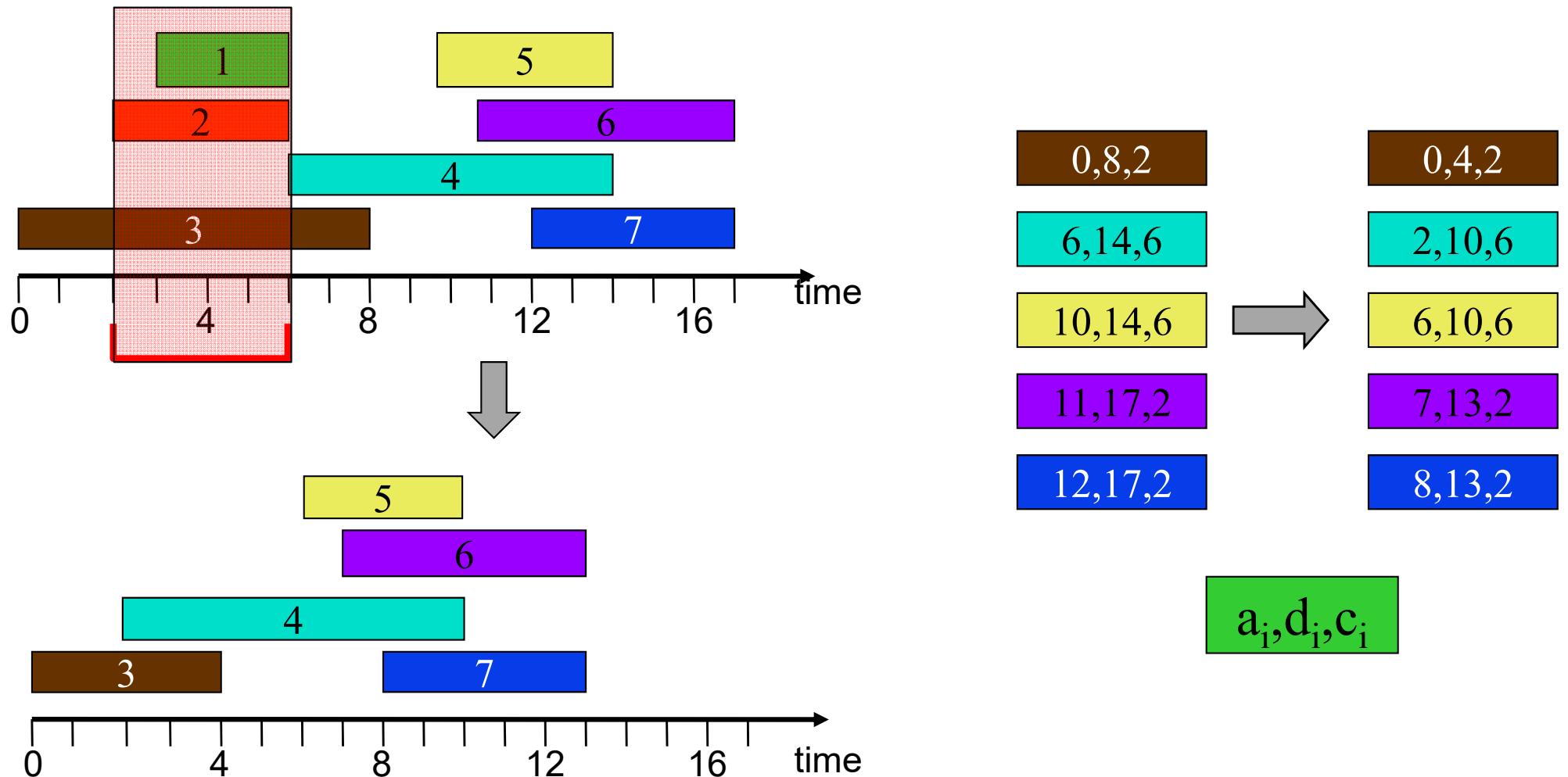
YDS Algorithm for Offline Scheduling

- ▶ **Step 1:** Execute jobs in the interval with the highest intensity by using the earliest-deadline first schedule and running at the intensity as the frequency.



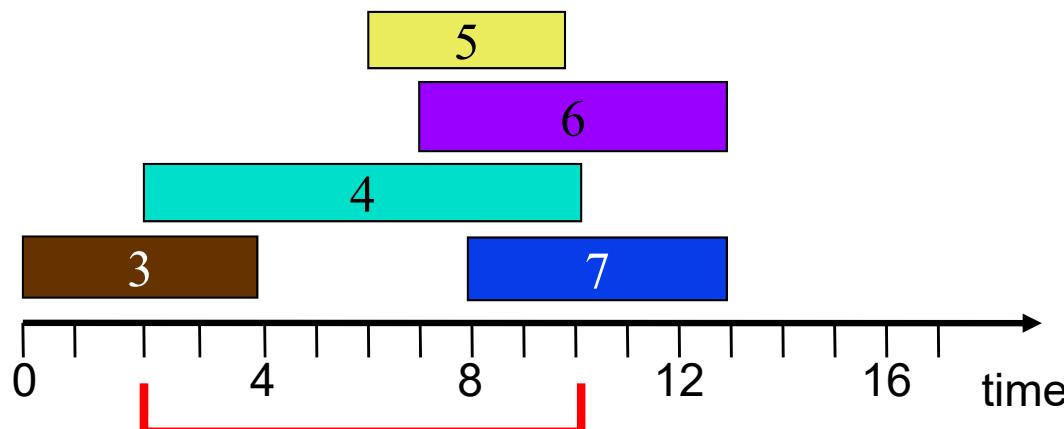
YDS Algorithm for Offline Scheduling

- ▶ **Step 2:** Adjust the arrival times and deadlines by excluding the possibility to execute at the previous critical intervals.



YDS Algorithm for Offline Scheduling

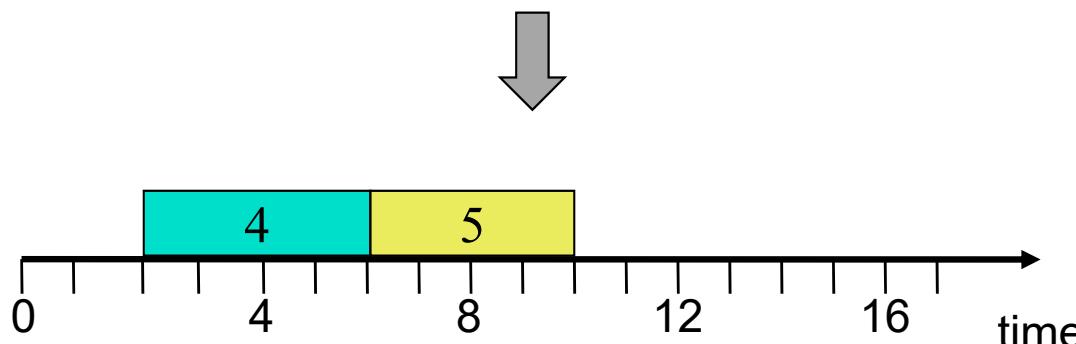
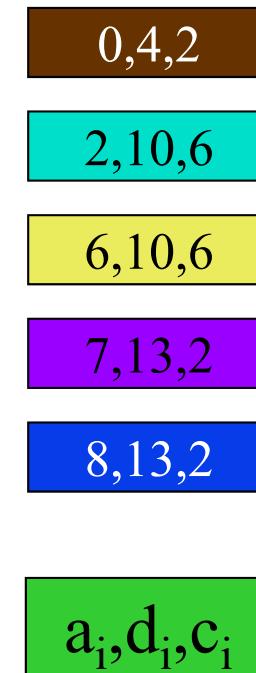
- ▶ **Step 3:** Run the algorithm for the revised input again



$$G([0,4]) = 2/4, G([0,10]) = 14/10, G([0,13]) = 18/13$$

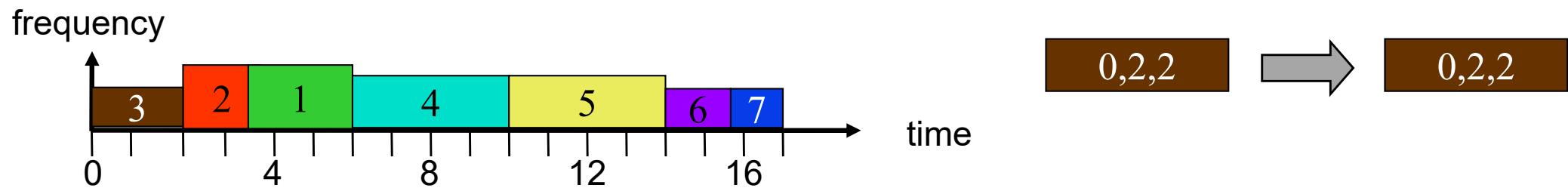
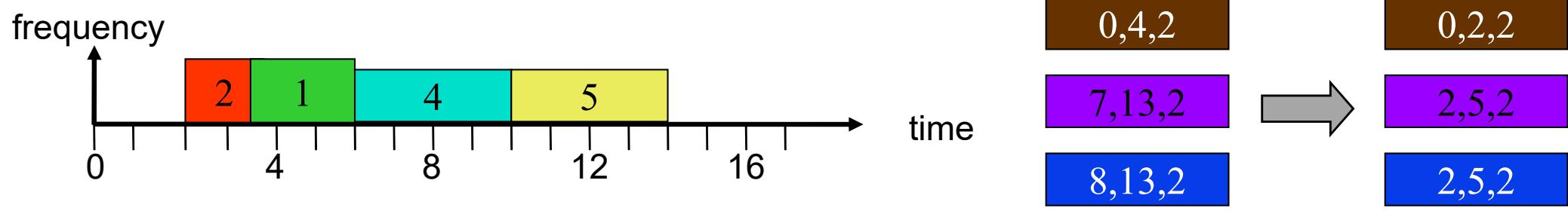
$$G([2,10]) = 12/8, G([2,13]) = 16/11, G([6,10]) = 6/4$$

$$G([6,13]) = 10/7, G([7,13]) = 4/6, G([8,13]) = 4/5$$



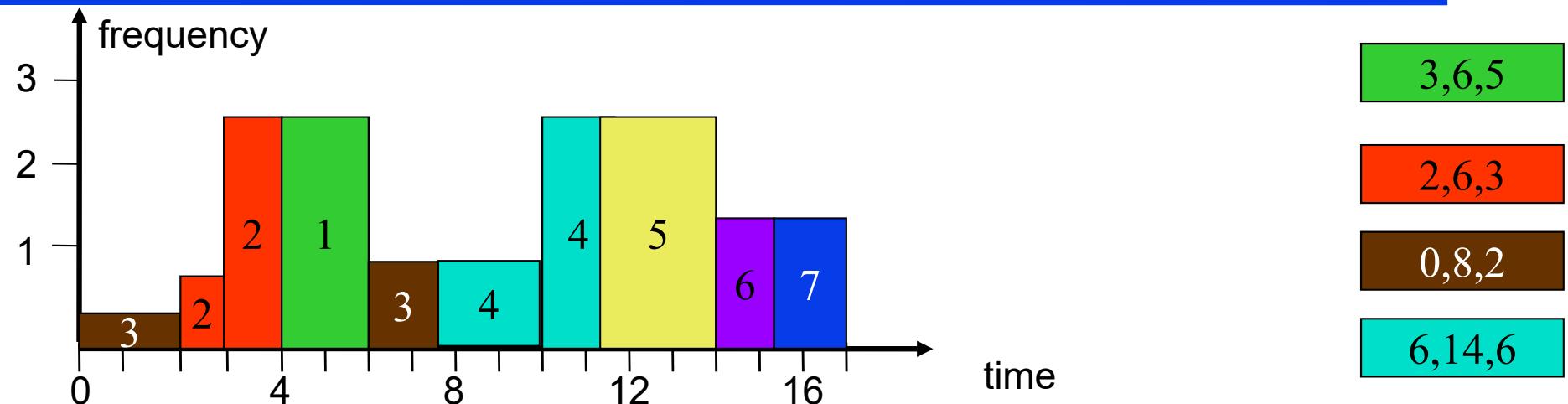
YDS Algorithm for Offline Scheduling

- ▶ **Step 3:** Run the algorithm for the revised input again
- ▶ **Step 4:** Put pieces together



	v_1	v_2	v_3	v_4	v_5	v_6	v_7
frequency	2	2	1	1.5	1.5	$\frac{4}{3}$	$\frac{4}{3}$

DVS: Online Scheduling on One Processor



► Continuously update to the best schedule for all arrived tasks

- Time 0: task v_3 is executed at 2/8
- Time 2: task v_2 arrives
 - $G([2,6]) = \frac{3}{4}$, $G([2,8]) = 4.5/6=3/4 \Rightarrow$ execute v_2 at $\frac{3}{4}$
- Time 3: task v_1 arrives
 - $G([3,6]) = (5+3-3/4)/3=29/12$, $G([3,8]) < G([3,6]) \Rightarrow$ execute v_2 and v_1 at $29/12$
- Time 6: task v_4 arrives
 - $G([6,8]) = 1.5/2$, $G([6,14]) = 7.5/8 \Rightarrow$ execute v_3 and v_4 at $15/16$
- Time 10: task v_5 arrives
 - $G([10,14]) = 39/16 \Rightarrow$ execute v_4 and v_5 at $39/16$
- Time 11 and Time 12
 - The arrival of v_6 and v_7 does not change the critical interval
- Time 14:
 - $G([14,17]) = 4/3 \Rightarrow$ execute v_6 and v_7 at $4/3$

Remarks on YDS Algorithm

► *Offline*

- The algorithm guarantees the minimal energy consumption while satisfying the timing constraints
- The time complexity is $O(N^3)$, where N is the number of tasks in V
 - Finding the critical interval can be done in $O(N^2)$
 - The number of iterations is at most N
- Exercise:
 - For periodic real-time tasks with deadline=period, running at ***constant speed with 100% utilization*** under EDF has minimum energy consumption while satisfying the timing constraints.

► *Online*

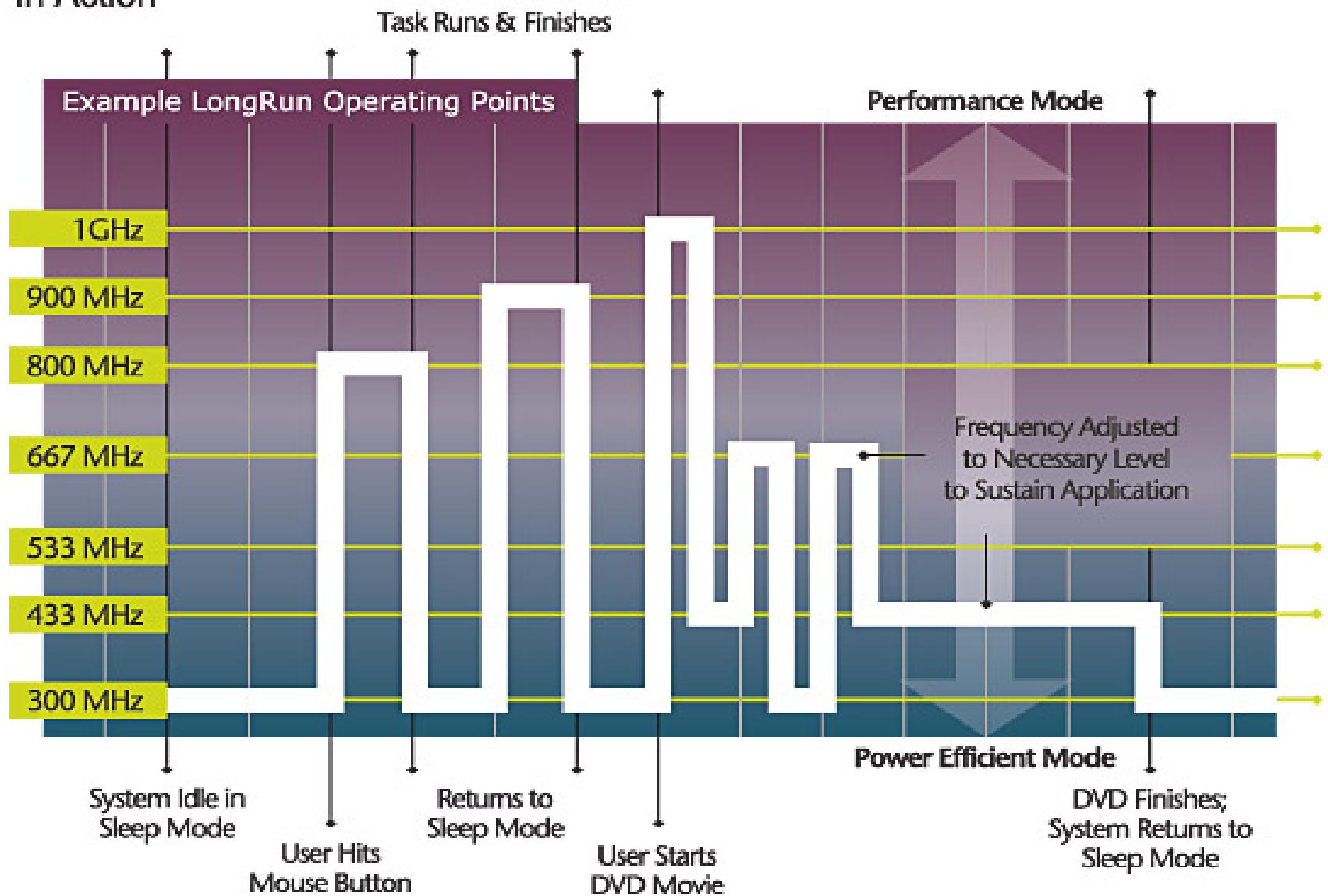
- Compared to the optimal offline solution, the on-line schedule uses at most 27 times of the minimal energy consumption.

Topics

- ▶ General Remarks
- ▶ Power and Energy
- ▶ Basic Techniques
 - Parallelism
 - VLIW (parallelism and reduced overhead)
 - Dynamic Voltage Scaling
 - ***Dynamic Power Management***

Transmeta™ LongRun™ Power Management

In Action



Dynamic Power Management (DPM)

Dynamic Power management tries to assign optimal **power saving states**

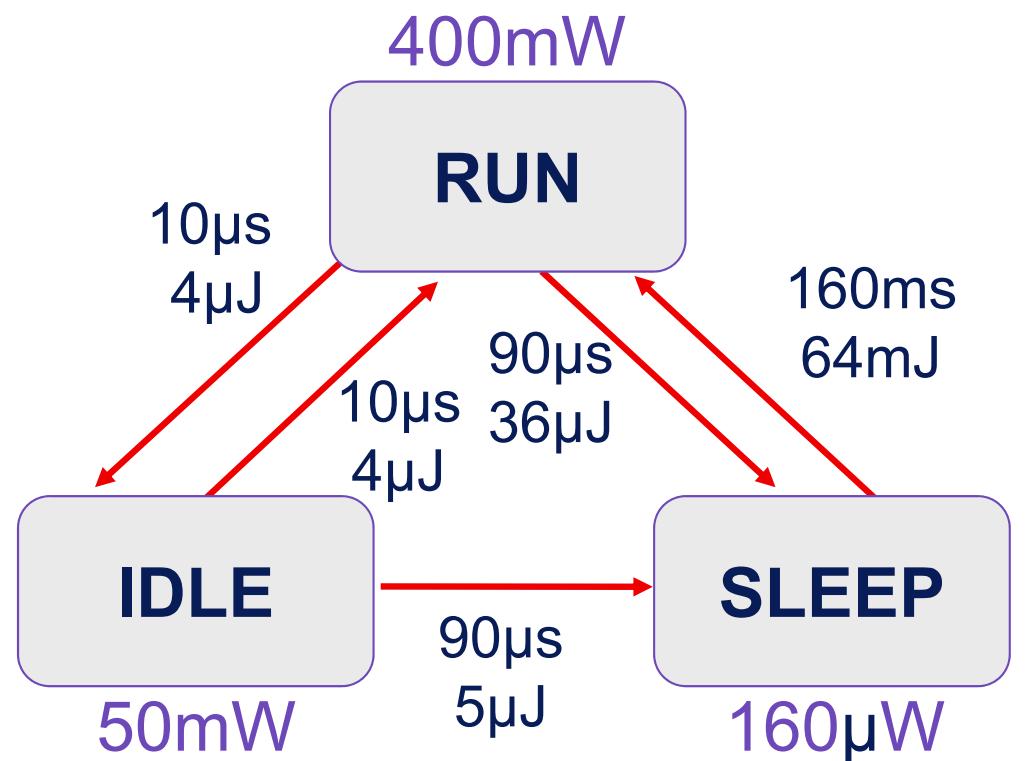
Requires Hardware Support

Example: StrongARM SA1100

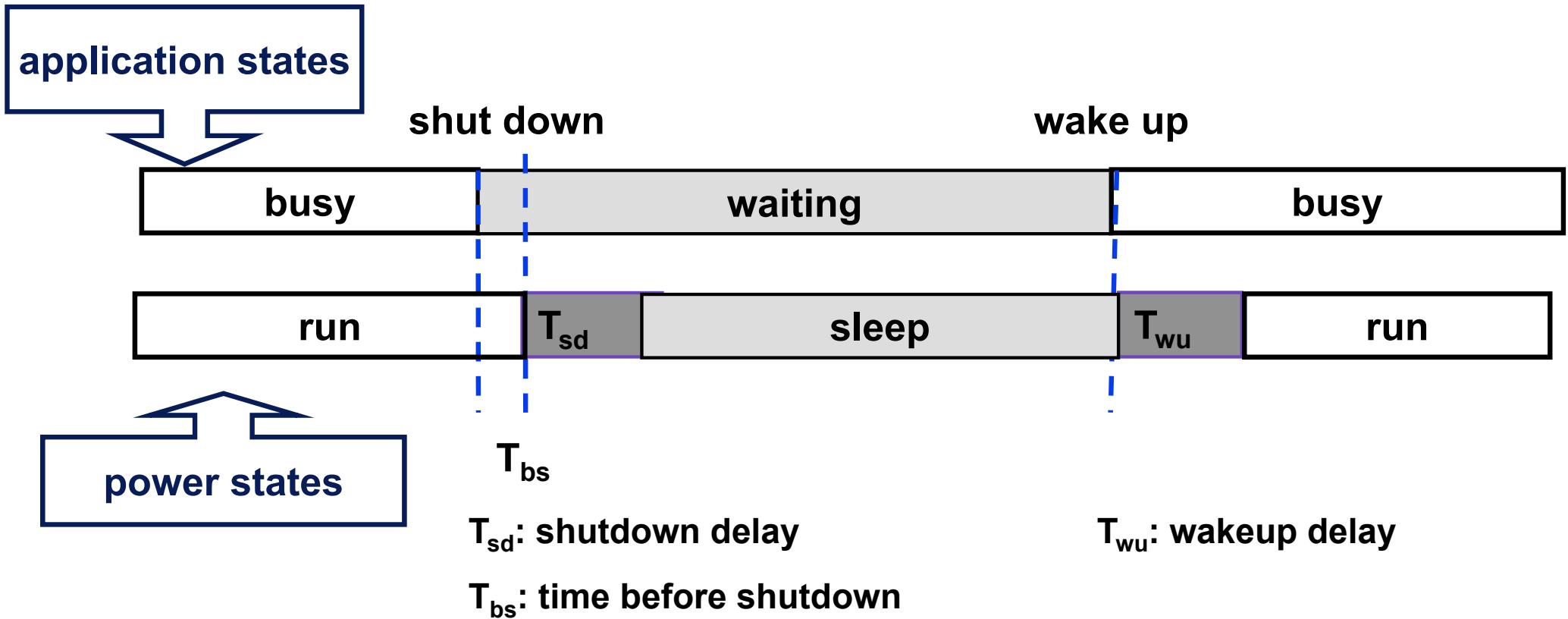
RUN: operational

IDLE: a SW routine may stop the CPU when not in use, while monitoring interrupts

SLEEP: Shutdown of on-chip activity



Reduce Power According to Workload



Desired: Shutdown only during long idle times
→ Tradeoff between savings and overhead

The Challenge

Questions:

- When to go to a power-saving state?
- Is an idle period long enough for shutdown?

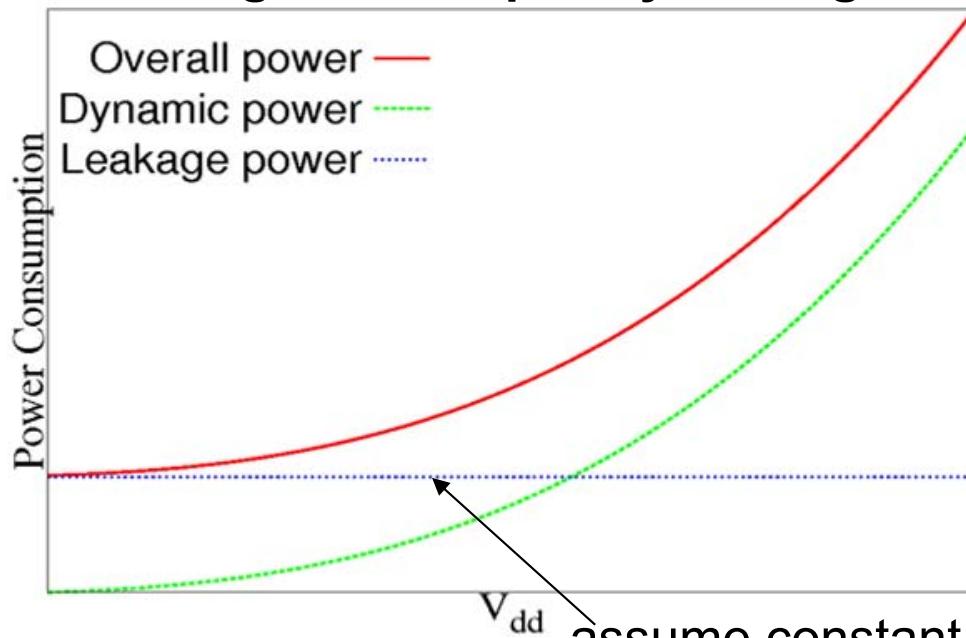
→ Predicting the future

Combining DVFS and DPM

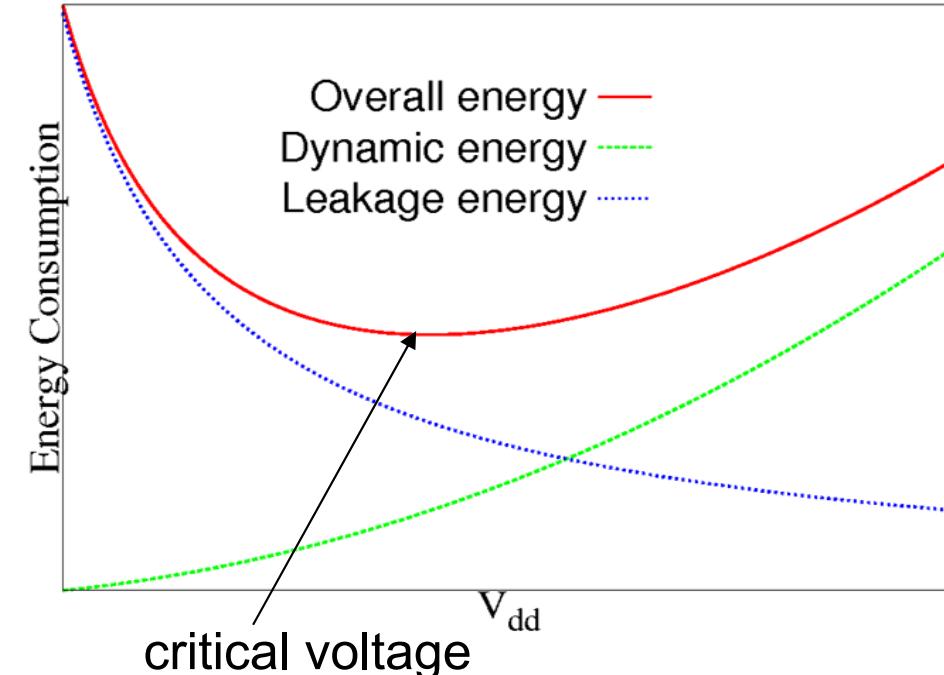
DVS Critical frequency (voltage):

- Running at any frequency/voltage lower than this frequency is not worthwhile for execution.

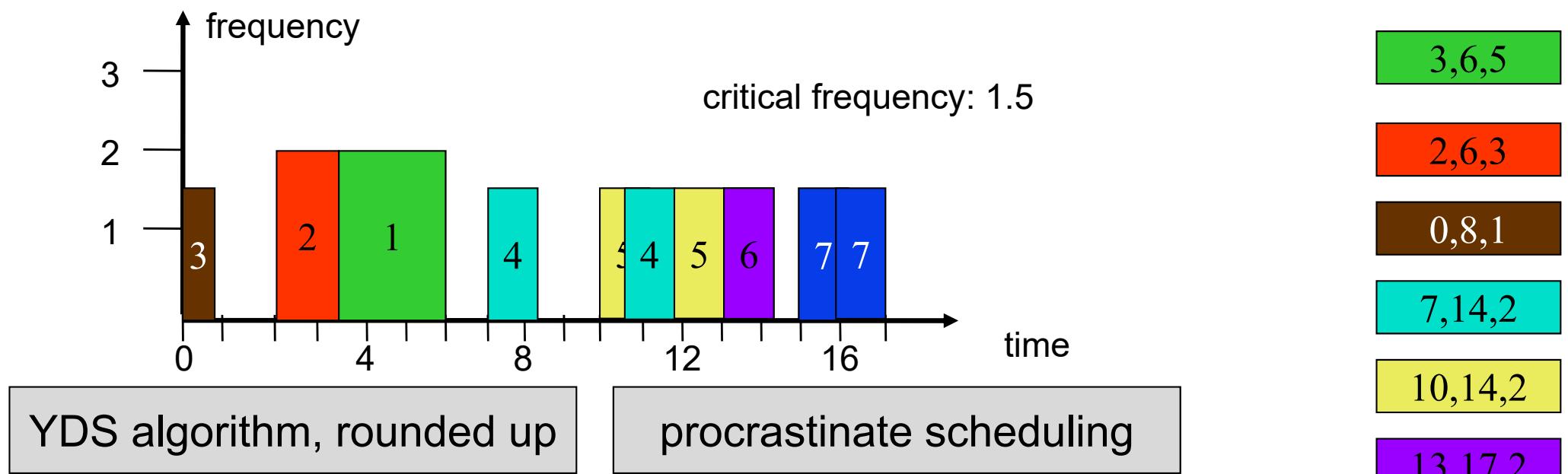
power during „run task“ using voltage and frequency scaling



energy for executing task



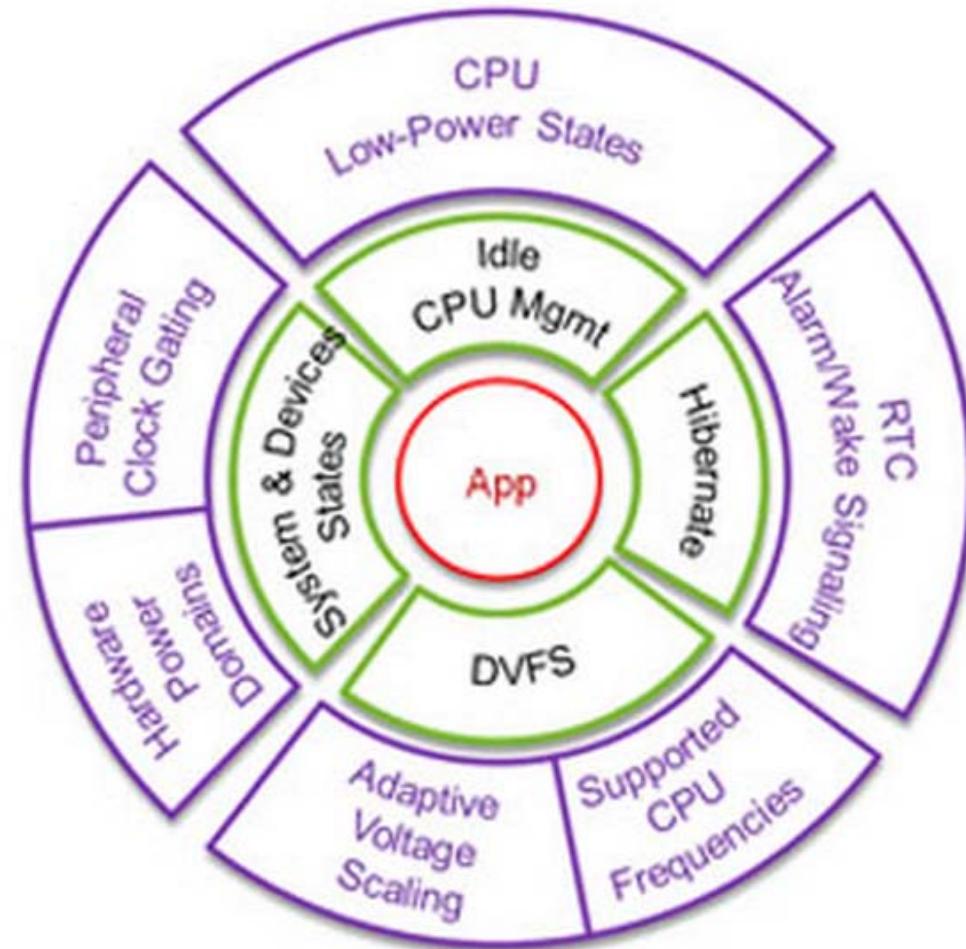
Procrastination Schedule



- ▶ Execute by using voltages higher or equal to the critical voltage only
 - apply YDS algorithm
 - round up voltages lower than the critical voltage
- ▶ Procrastinate the execution of tasks to aggregate enough time for sleeping
 - Try to reduce the number of times to turn on/off
 - Sleep as long as possible

Operating System Services

- = Hardware power management
- = Application Software
- = RTOS Power Management Framework

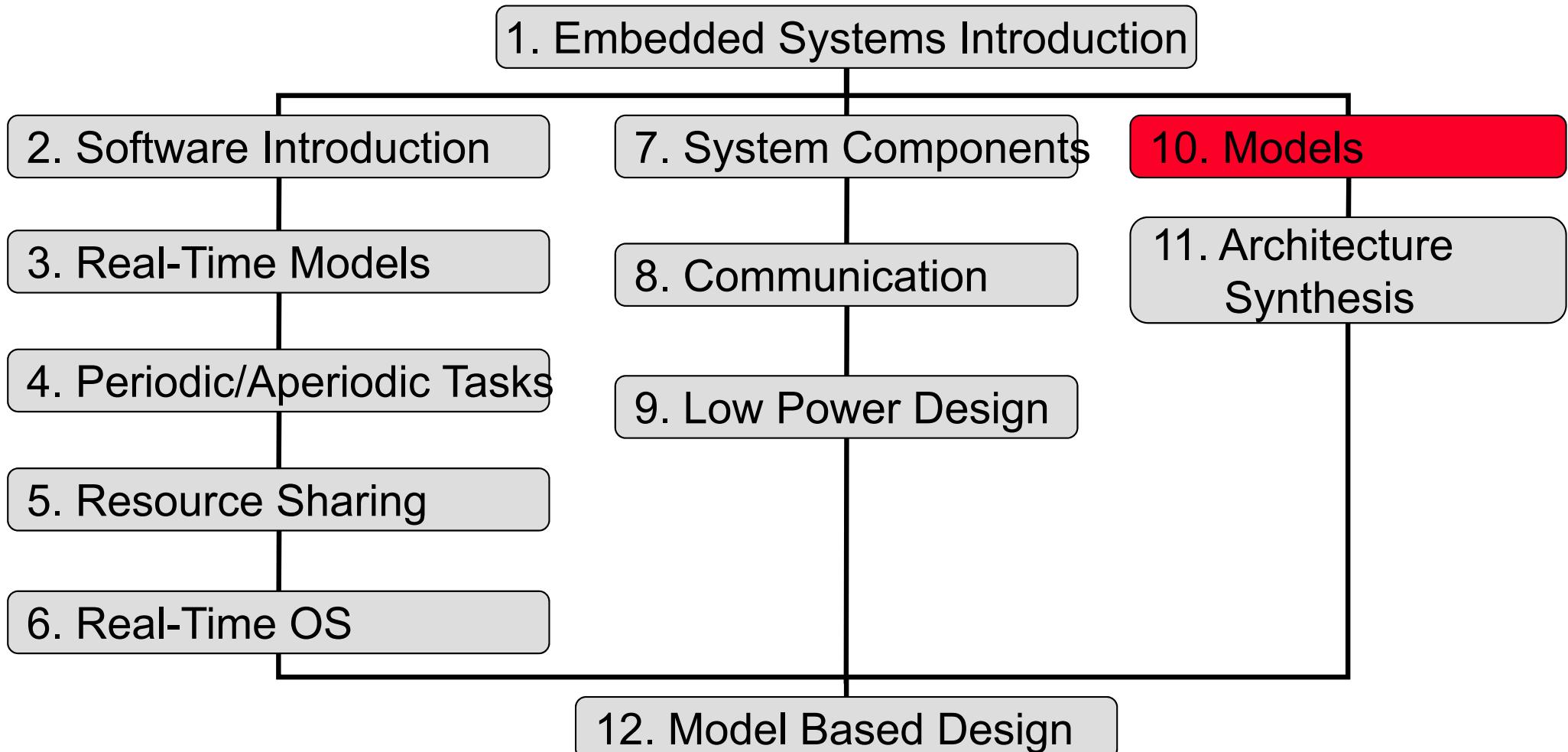


Embedded Systems

10. Architecture Design – Models

Lothar Thiele

Contents of Course



*Software and
Programming*

*Processing and
Communication*

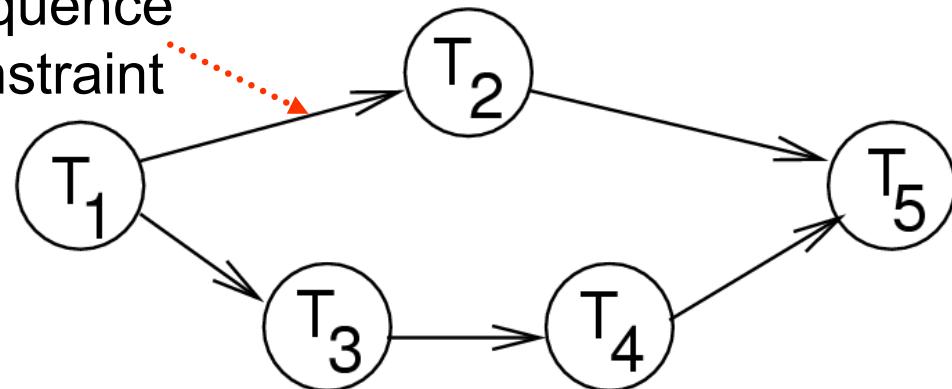
Hardware

Specification

- ▶ **Formal specification** of the desired functionality and the structure (architecture) of an embedded systems is a **necessary step** for using computer aided design methods.
- ▶ There exist **many different formalisms** and models of computation, see also the models used for real-time software (chapter 3) and general specification models for the whole system.
- ▶ **Now:**
 - Relevant models for the architecture level (hardware).

Task Graph or Dependence Graph (DG)

Sequence constraint



Nodes are assumed to be a „program“ described in some programming language, e.g. C or Java; or just a single operation.

Def.: A **dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a partial order.

If $(v_1, v_2) \in E$, then v_1 is called an **immediate predecessor** of v_2 and v_2 is called an **immediate successor** of v_1 .

Suppose E^* is the transitive closure of E .

If $(v_1, v_2) \in E^*$, then v_1 is called a **predecessor** of v_2 and v_2 is called a **successor** of v_1 .

Dependence Graph (DG)

- ▶ A dependence graph describes ***order relations*** for the execution of single operations or tasks. ***Nodes*** correspond to ***tasks or operations***, ***edges*** correspond to ***relations*** („executed after“).
- ▶ Usually, a dependence graph describes a ***partial order*** between operations and therefore, leaves freedom for scheduling (parallel or sequential). It represents ***parallelism*** in a program ***but no branches*** in control flow.
- ▶ A dependence graph is acyclic.
- ▶ Often, there are additional quantities associated to edges or nodes such as
 - execution times, deadlines, arrival times
 - communication demand

Single Assignment Form

given basic block:

$x = a + b;$

$y = c - d;$

$z = x * y;$

$y = b + d;$

single assignment
form:

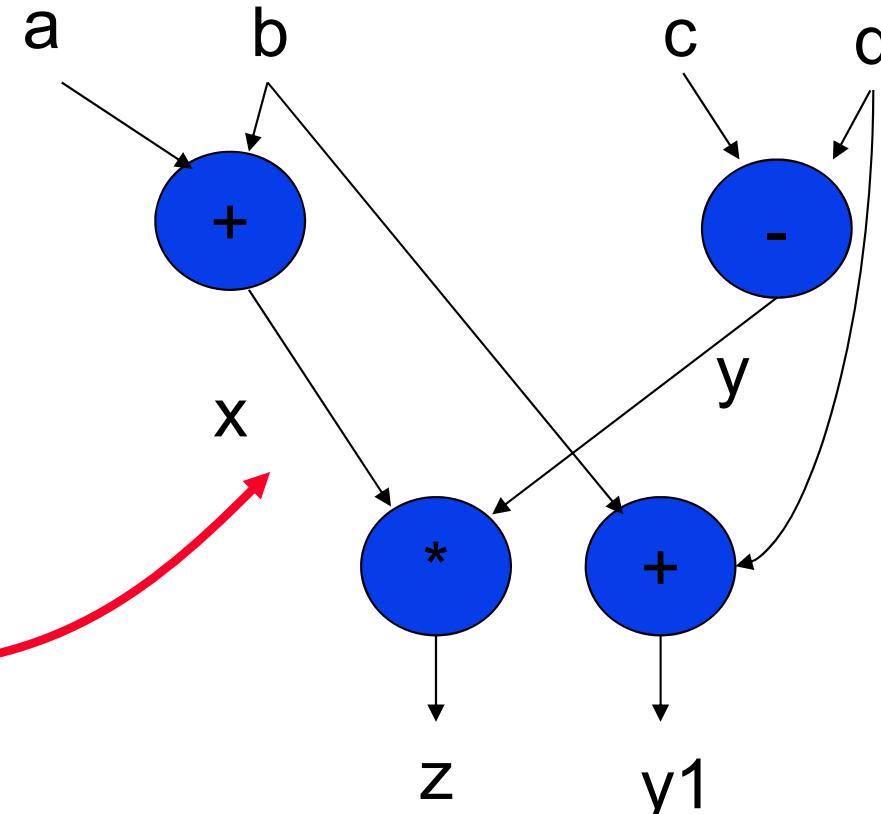
$x = a + b;$

$y = c - d;$

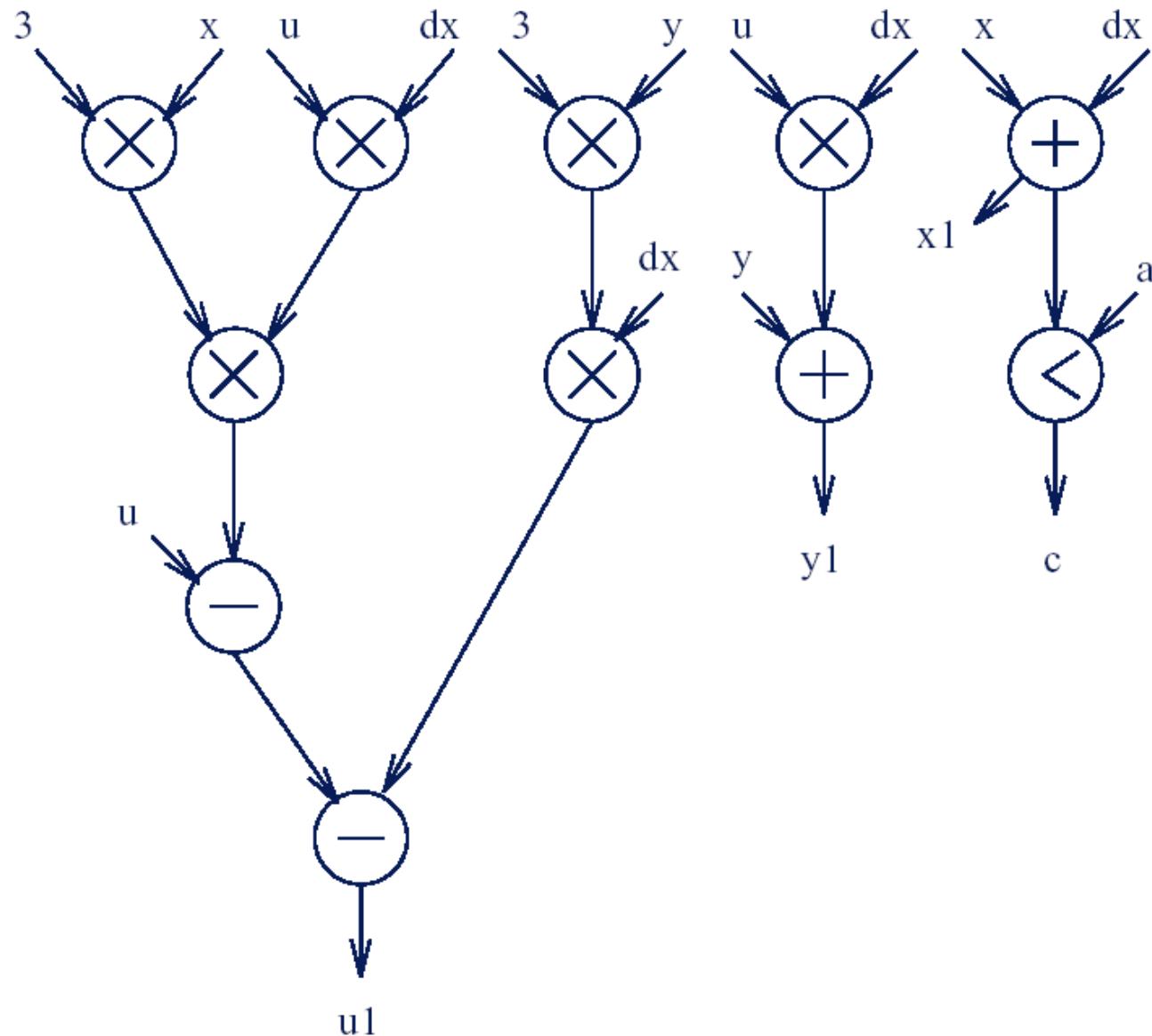
$z = x * y;$

$y1 = b + d;$

dependence graph



Dependence Graph (DG)



Control-Data Flow Graph (CDFG)

- ▶ ***Goal:***

- Description of control structures (for example branches) and data dependencies.

- ▶ ***Applications:***

- Describing the semantics of programming languages.
 - Internal representation in compilers for hardware and software.

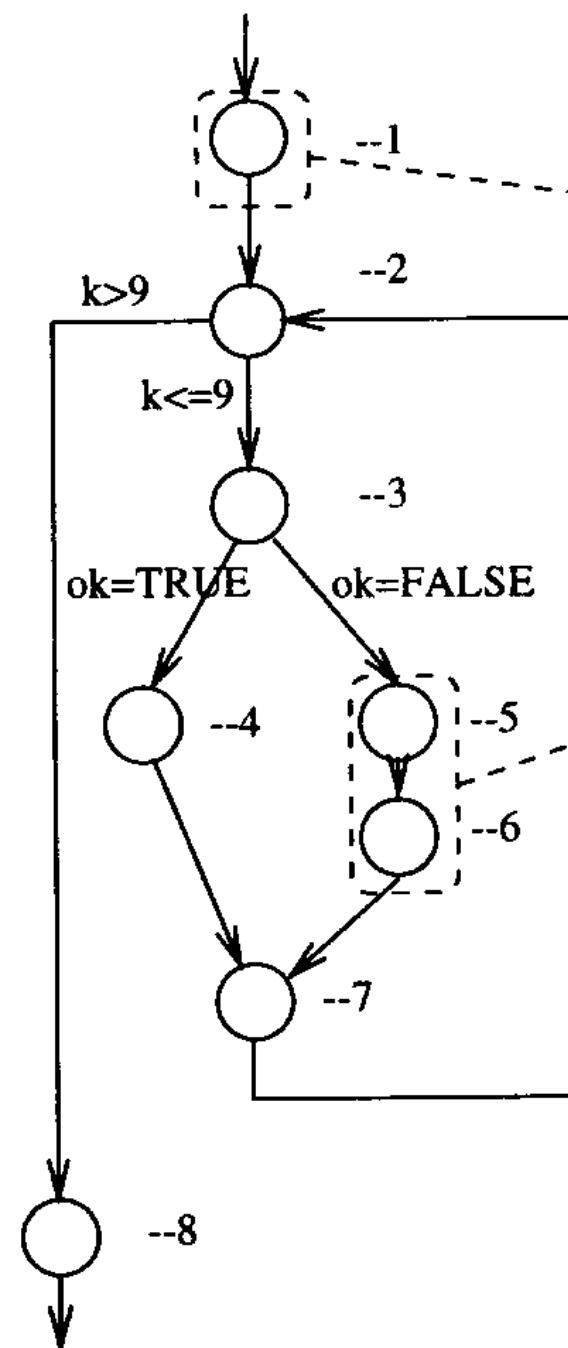
- ▶ ***Representation:***

- Combination of control flow (sequential state machine) and dependence representation.
 - Many variants exist.

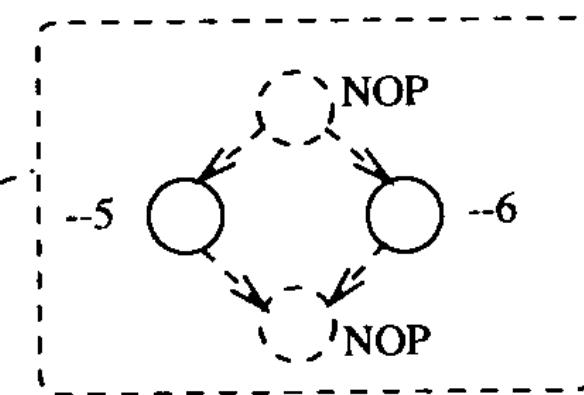
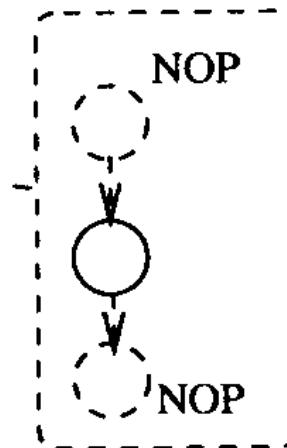
a) VHDL-Code:

```
...  
s := k; --1  
LOOP  
  EXIT WHEN k>9; --2  
  IF (ok = TRUE) --3  
    j:=j+1; --4  
  ELSE  
    j:= 0; --5  
    ok:= TRUE; --6  
  END IF;  
  k:=k+1; --7  
END LOOP;  
r := j; --8  
...
```

b) CDFG: CFG



+ DFGs



Control-Data Flow Graph (CDFG)

- ▶ ***Control Flow Graph:***

- It corresponds to a ***finite state machine***, which represents the sequential control flow in a program.
- ***Branch conditions*** are very often associated to the outgoing edges of a node.
- The operations to be executed within a state (node) are associated in form of a ***dependence graph***.

- ▶ ***Dependence Graph (also called Data Flow Graph DFG):***

- NOP (no operation) operations represent the start point and end point of the execution. This form of a graph is called a ***polar graph***: it contains two distinguished nodes, one without incoming edges, the other one without outgoing edges.

Sequence Graph (SG)

- ▶ A **sequence graph** is a **hierarchy** of directed graphs. A generic element of the graph is a **dependence graph** with the following properties:
 - It contains **two kinds** of nodes: (a) **operations or tasks** and (b) **hierarchy nodes**.
 - Each graph is **acyclic and polar** with two distinguished nodes: the start node and the end node. No operation is assigned to them (NOP).
 - There are the following **hierarchy nodes**: (a) module call (**CALL**) (b) branch (**BR**) and (c) iteration (**LOOP**).

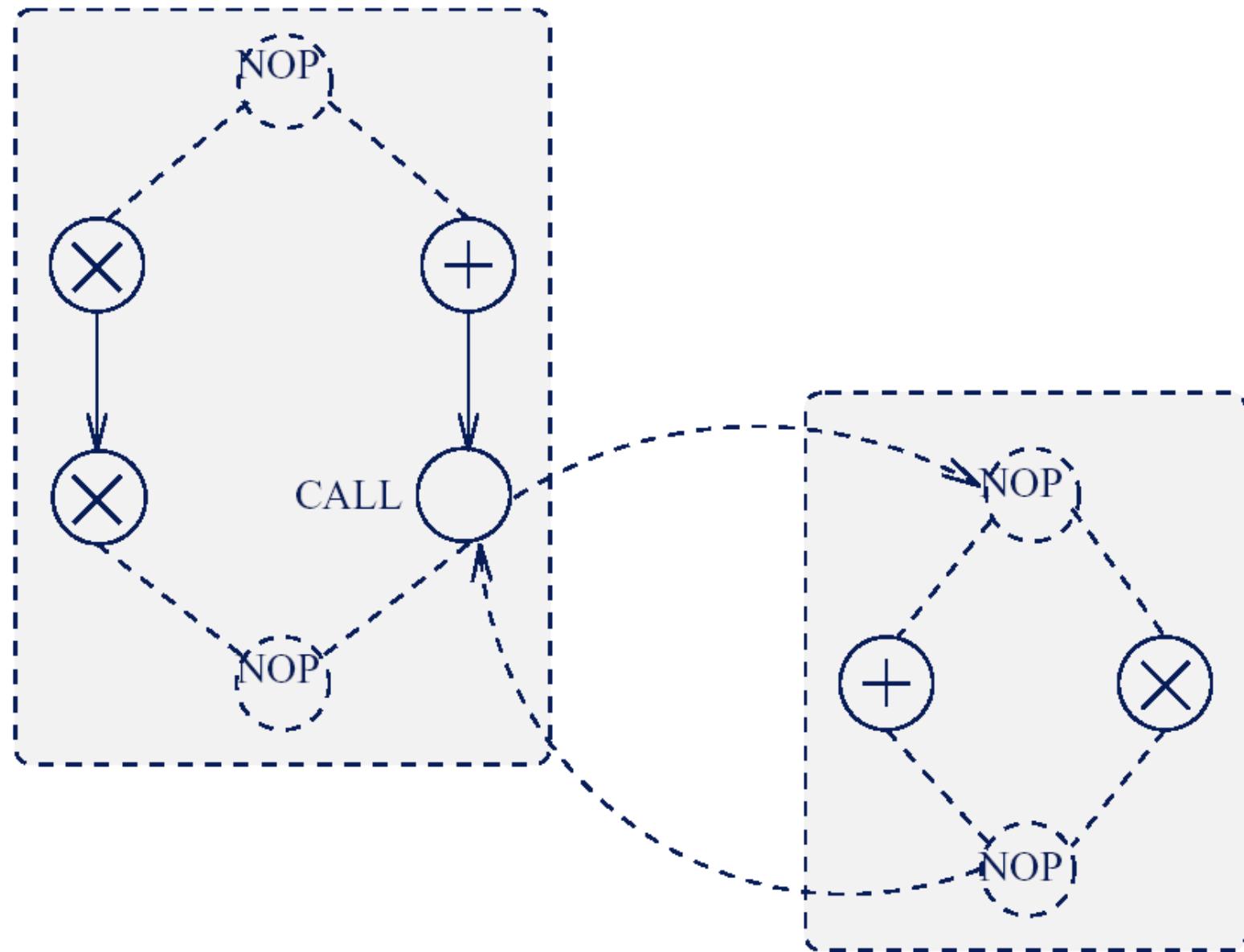
Sequence Graph (SG)

- ▶ *Example:*

```
x := a * b;  
y := x * c;  
z := a + b;  
submodul(a, z);
```

```
PROCEDURE submodul(m, n) IS  
    p := m + n;  
    q := m * n;  
END submodul
```

Sequence Graph (SG)

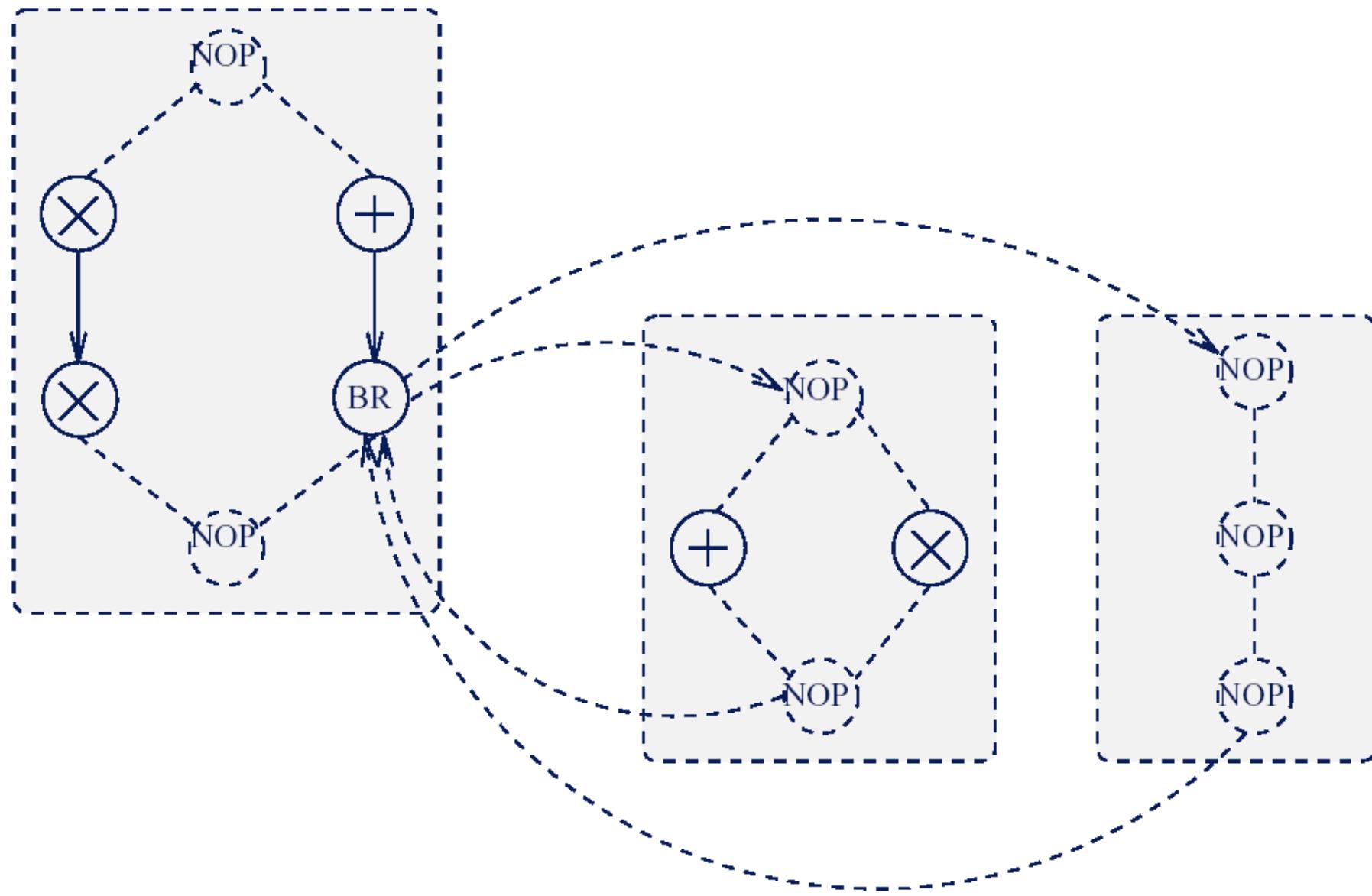


Sequence Graph (SG)

- ▶ *Example:*

```
x := a * b;  
y := x * c;  
z := a + b;  
IF z > 0 THEN  
    p := m + n;  
    q := m * n;  
END IF
```

Sequence Graph (SG)

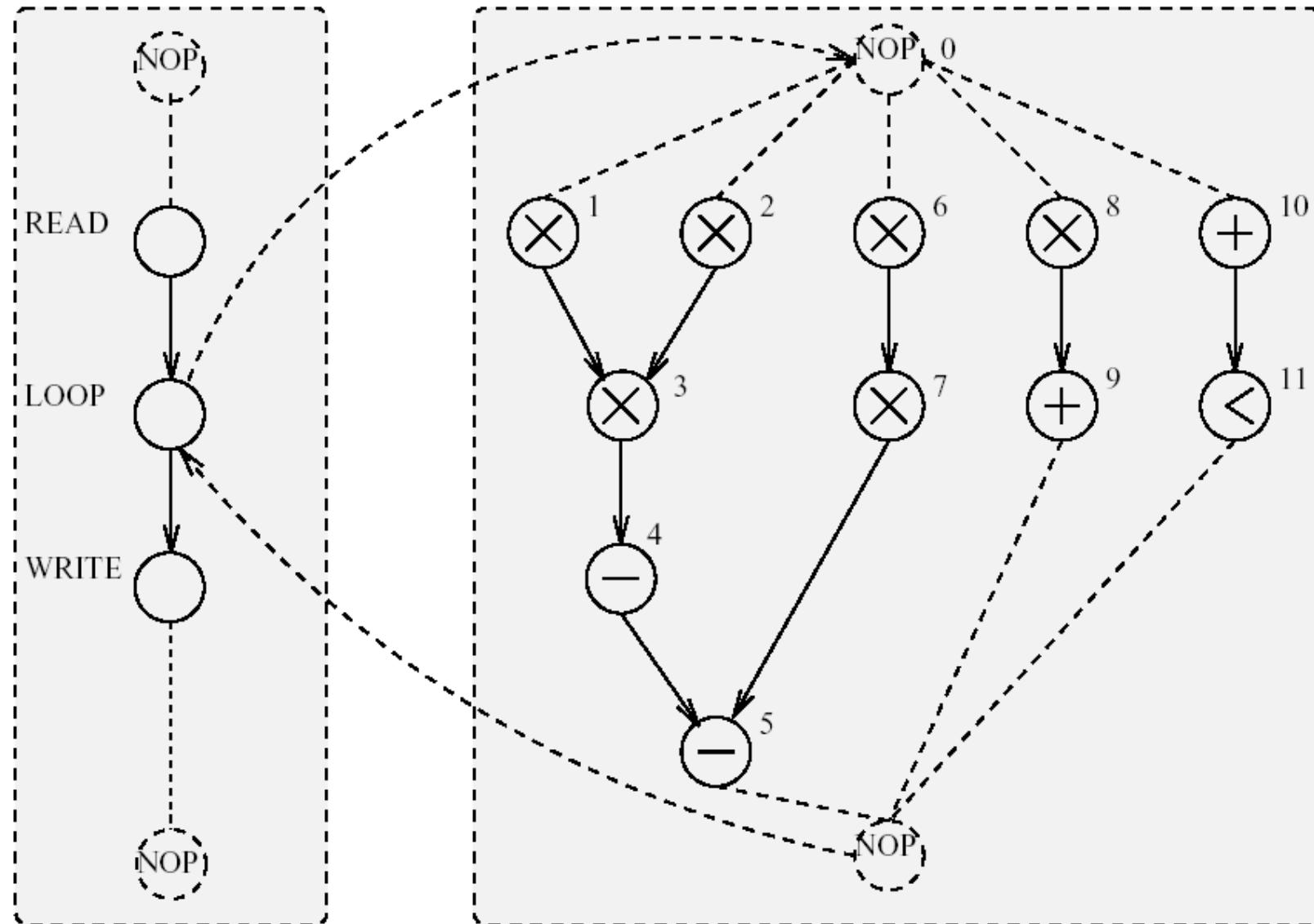


Sequence Graph (SG)

- ▶ **Example** iteration (differential equation):

```
int diffeq(int x, int y, int u, int dx, int a)
{ int x1, u1, y1;
while ( x < a ) {
    x1 = x + dx;
    u1 = u - (3 * x * u * dx) - (3 * y * dx);
    y1 = y + u * dx;
    x = x1; u = u1; y = y1;
}
return y;
}
```

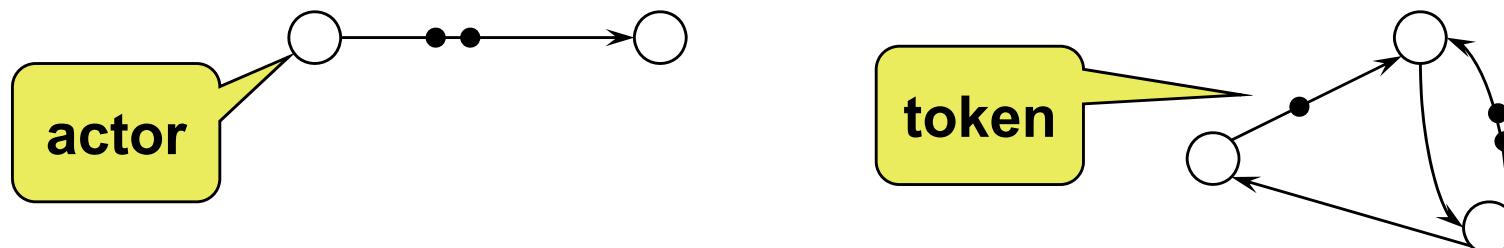
Sequence Graph (SG)



Marked Graphs (MG)

- ▶ A **marked graph** $G = (V, A, del)$ consists of
 - nodes (**actors**) $v \in V$
 - edges $a = (v_i, v_j) \in A, A \subseteq V \times V$
 - number of initial tokens on edges $del : A \rightarrow \mathbb{N}$
- ▶ The **marking** (distribution of tokens) is often represented in form of a vector:

$$del = (\dots del_k \dots) \in \mathbb{Z}^{1 \times |A|}$$



Marked Graphs (MG)

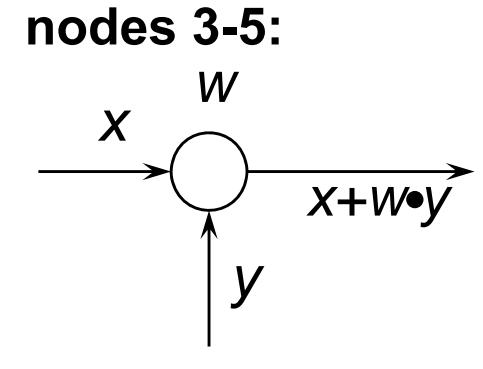
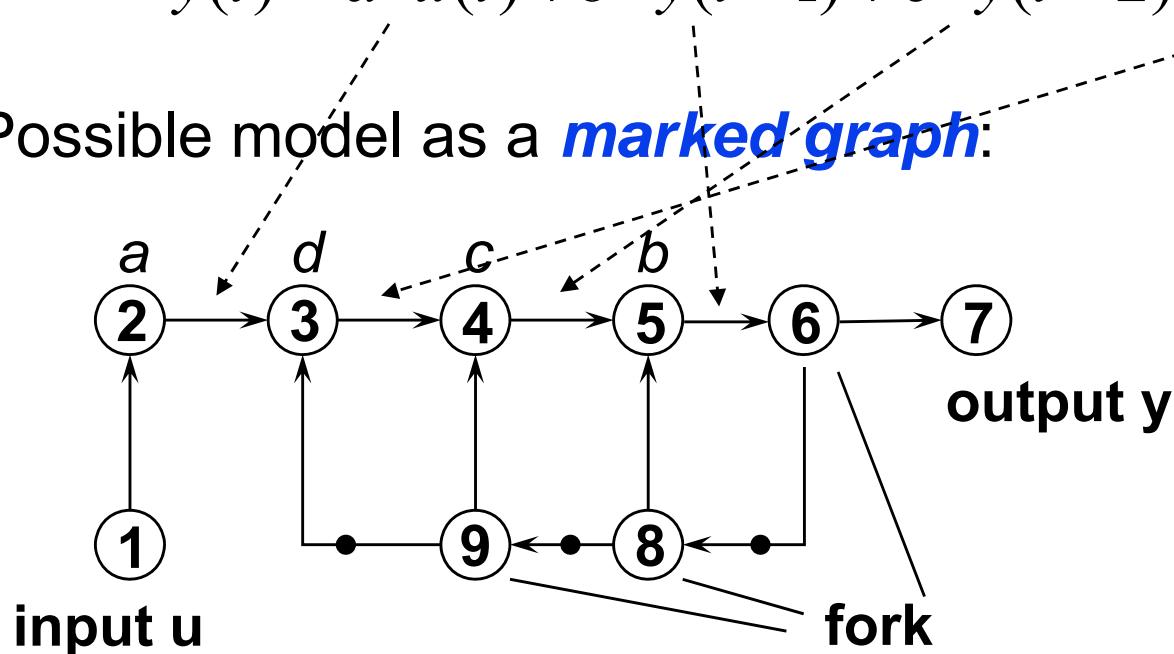
- ▶ The token on the edges correspond to data that are stored in **FIFO queues**.
- ▶ A node (actor) is called **activated** if on every input edge there is at least one token.
- ▶ A node (actor) can **fire** if it is activated.
- ▶ The **firing** of a node v_i (actor operates on the first tokens in the input queues) removes from each input edge a token and adds a token to each output edge. The output token correspond to the processed data.
- ▶ Marked graphs are mainly used for modeling **regular computations**, for example signal flow graphs.

Marked Graphs (MG)

- ▶ **Example** (model of a digital filter with infinite impulse response IIR)
 - **Filter equation:**

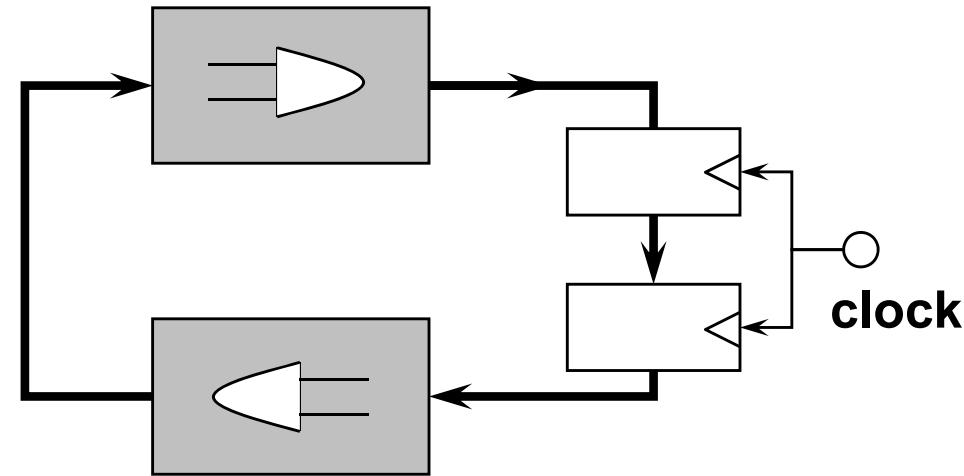
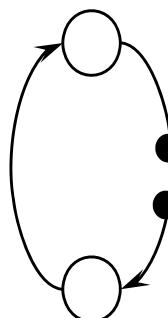
$$y(l) = a \cdot u(l) + b \cdot y(l-1) + c \cdot y(l-2) + d \cdot y(l-3)$$

- Possible model as a **marked graph**:



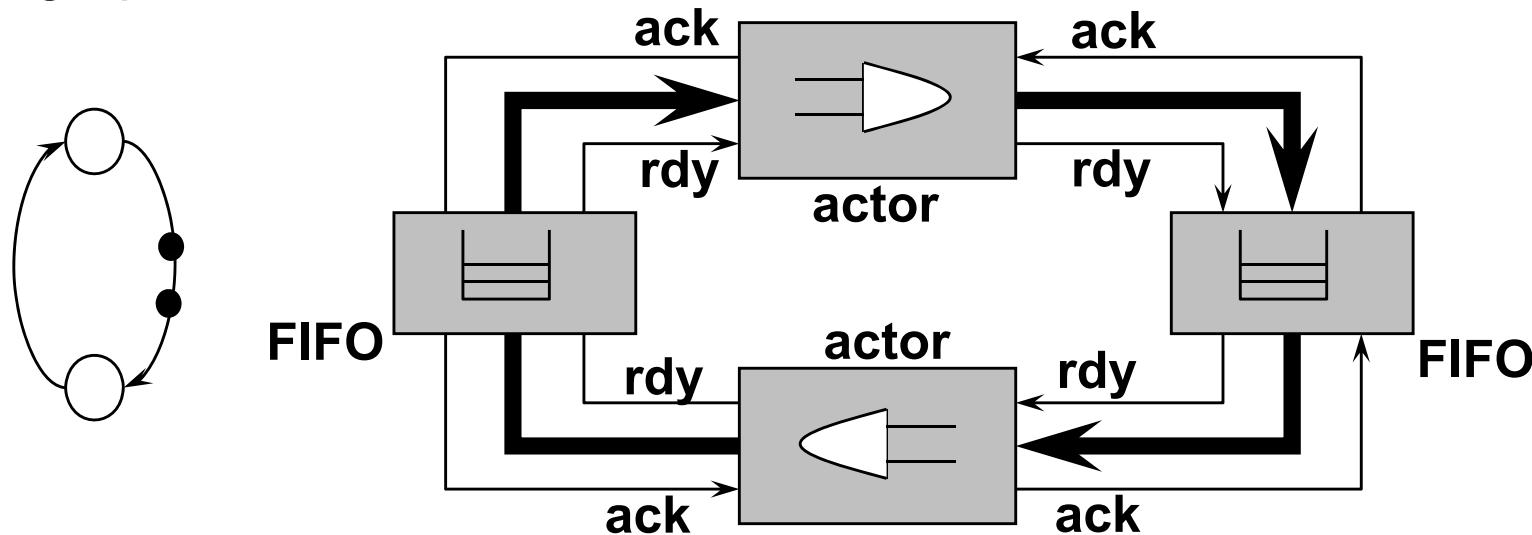
Implementation of Marked Graphs

- ▶ There are ***different possibilities*** to implement marked graphs in hardware or software directly. Only the most simple possibilities are shown here.
- ▶ Hardware implementation as a ***synchronous digital circuit***.
 - Actors are implemented as combinatorial circuits.
 - Edges correspond to synchronously clocked shift registers (FIFOs).



Implementation of Marked Graphs

- Hardware implementation as a ***self-timed asynchronous circuit***.
 - Actors and FIFO registers are implemented as independent units.
 - The coordination and synchronization of firings is implemented using a ***handshake protocol***.
 - Delay insensitive direct implementation of the semantics of marked graphs.



Implementation of Marked Graphs

- **Software** implementation with **static scheduling**:
 - At first, a **feasible sequence** of actor firings is determined which ends in the starting state (initial distribution of tokens).
 - This sequence is implemented directly in software.
 - **Example** digital filter:

feasible sequence: (1, 2, 3, 9, 4, 8, 5, 6, 7)

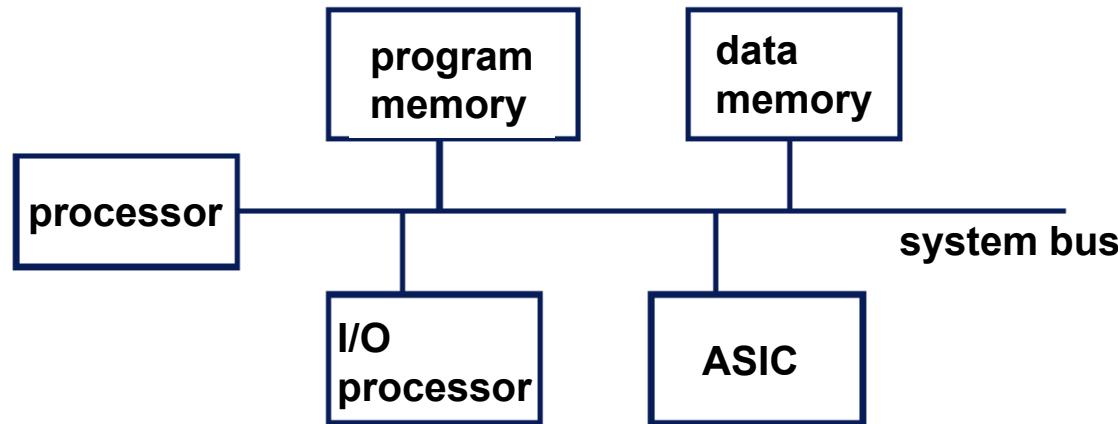
program:

```
while(true) {
    t1 = read(u);
    t2 = a*t1;
    t3 = t2+d*t9;
    t9 = t8;
    t4 = t3+c*t9;
    t8 = t6;
    t5 = t4+b*t8;
    t6 = t5;
    write(y, t6); }
```

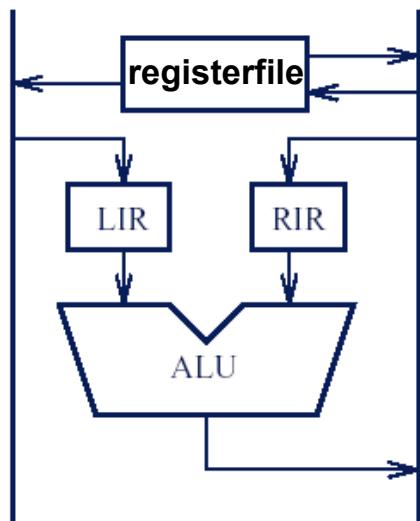
Implementation of Marked Graphs

- **Software** implementation with **dynamic scheduling**:
 - Scheduling is done using a (real-time) operating system.
 - **Actors** correspond to **threads** (or tasks).
 - **After firing** (finishing the execution of the corresponding thread) the thread is removed from the set of ready threads and put into **wait state**.
 - It is put into the **ready state** if all necessary input data are present.
 - This mode of execution directly corresponds to the semantics of marked graphs. It can be compared with the self-timed hardware implementation.

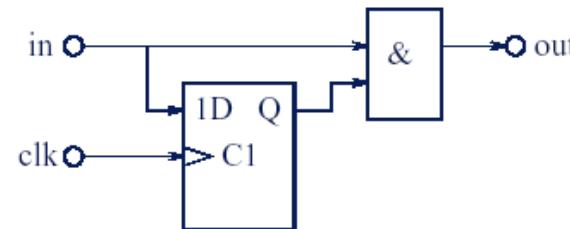
Block Diagrams



a) System Level



b) Architecture Level



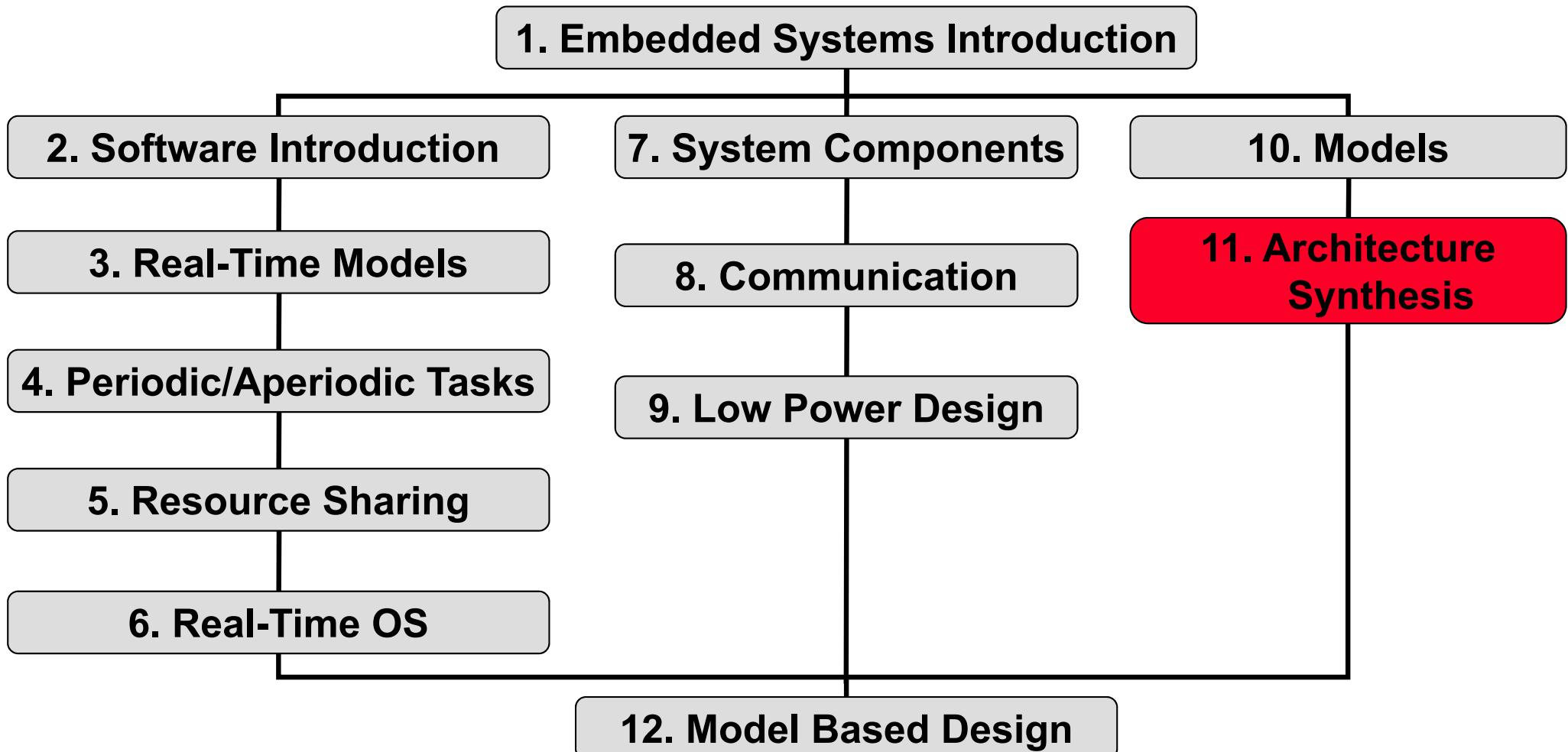
c) Logic Level

Embedded Systems

11. Architecture Synthesis

Lothar Thiele

Contents of Course



*Software and
Programming*

*Processing and
Communication*

Hardware

Contents

- ▶ **Models**
- ▶ Scheduling without resource constraints
 - ASAP
 - ALAP
 - Timing Constraints
- ▶ Scheduling with resource constraints
 - List Scheduling
 - Integer Linear Programming
- ▶ Iterative Algorithms
- ▶ Dynamic Voltage Scaling

Architecture Synthesis

Determine a hardware architecture that efficiently executes a given algorithm.

- ▶ **Major tasks** of architecture synthesis:
 - **allocation** (determine the necessary hardware resources)
 - **scheduling** (determine the timing of individual operations)
 - **binding** (determine relation between individual operations of the algorithm and hardware resources)
- ▶ **Classification** of synthesis algorithms:
 - **heuristics** or **exact methods**
- ▶ Synthesis methods can often be applied **independently of granularity** of algorithms, e.g. whether operation is a whole complex task or a single operation.

Models

- ▶ **Sequence graph** $G_S = (V_S, E_S)$
where V_S denotes the operations of the algorithm and E_S the dependence relations.
- ▶ **Resource graph** $G_R = (V_R, E_R)$, $V_R = V_S \cup V_T$
where V_T denote the resource types of the architecture and G_R is a bipartite graph. An edge $(v_s, v_t) \in E_R$ represents the availability of a resource type v_t for an operation v_s .
- ▶ **Cost function** $c : V_T \rightarrow \mathbf{Z}$
- ▶ **Execution times** $w : E_R \rightarrow \mathbf{Z}^{>0}$
are assigned to each edge $(v_s, v_t) \in E_R$ and denote the execution time of operation $v_s \in V_S$ on resource type $v_t \in V_T$.

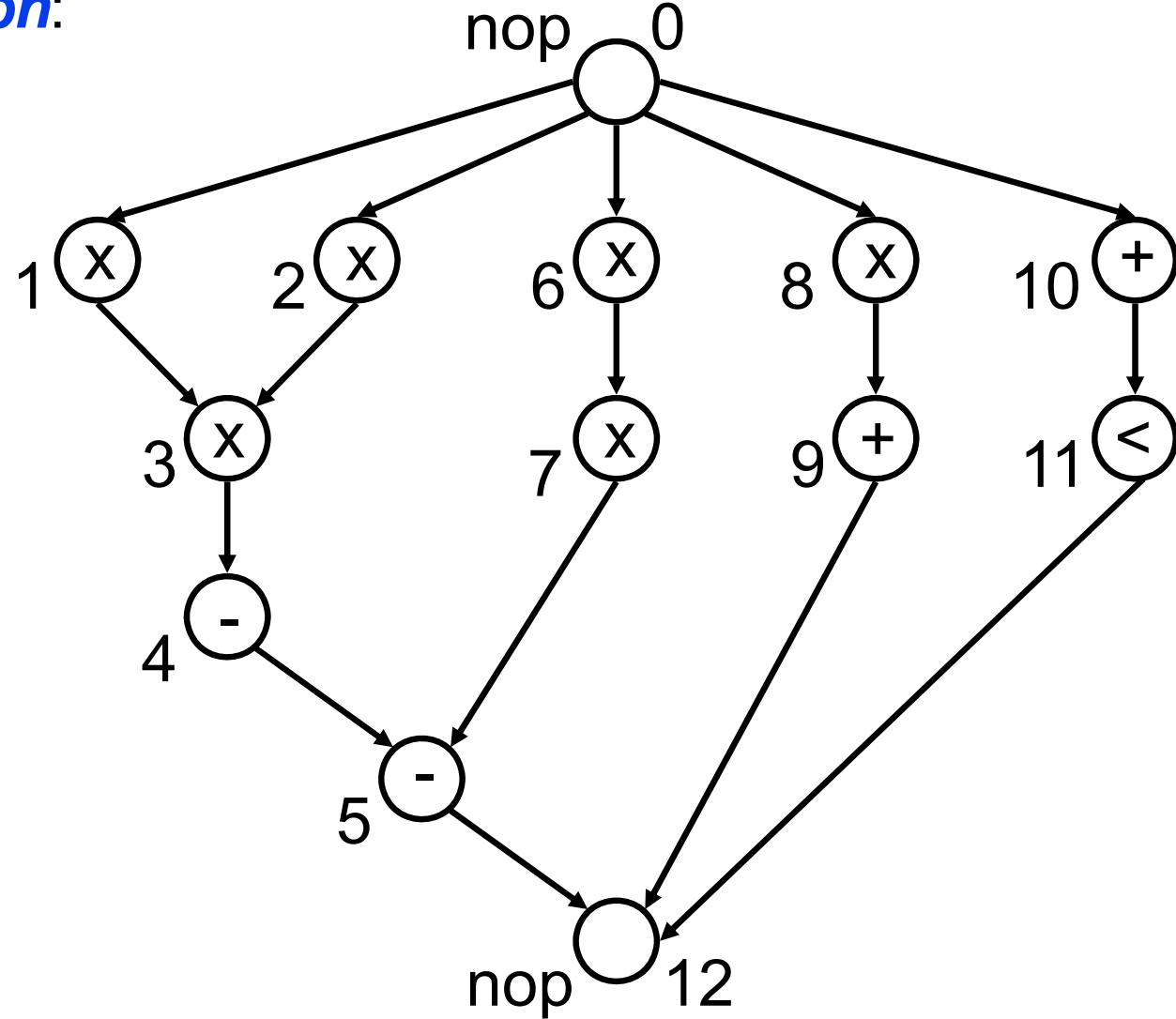
Models

- ▶ **Example** sequence graph:
 - Given **algorithm** (differential equation):

```
int diffeq(int x, int y, int u, int dx, int a) {  
    int x1, u1, y1;  
    while ( x < a ) {  
        x1 = x + dx;  
        u1 = u - ( 3 * x * u * dx ) - ( 3 * y * dx );  
        y1 = y + u * dx;  
        x = x1;  
        u = u1;  
        y = y1;  
    }  
    return y;  
}
```

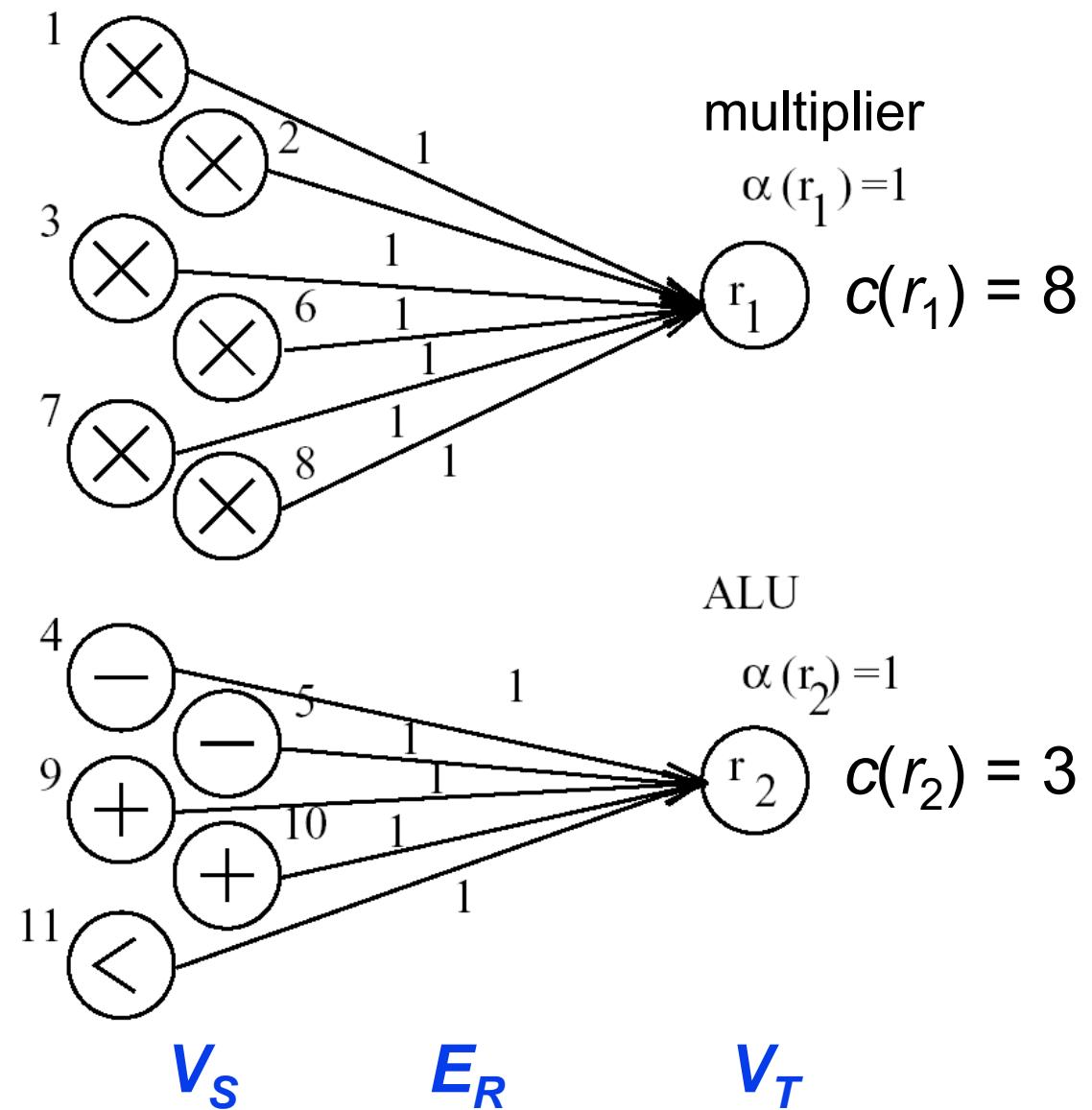
Models

- **Sequence graph:**



Models

- *Resource graph:*



Allocation and Binding

An allocation is a function $\alpha : V_T \rightarrow \mathbf{Z}^{\geq 0}$ that assigns to each resource type $v_t \in V_T$ the number $\alpha(v_t)$ of available instances.

A binding is defined by functions $\beta : V_S \rightarrow V_T$ and $\gamma : V_S \rightarrow \mathbf{Z}^{>0}$. Here, $\beta(v_s) = v_t$ and $\gamma(v_s) = r$ denote that operation $v_s \in V_S$ is implemented on the r th instance of resource type $v_t \in V_T$.

Scheduling

A schedule is a function $\tau : V_S \rightarrow \mathbf{Z}^{>0}$ that determines the starting times of operations. A schedule is feasible if the conditions

$$\tau(v_j) - \tau(v_i) \geq w(v_i) \quad \forall (v_i, v_j) \in E_S$$

are satisfied. $w(v_i) = w(v_i, \beta(v_i))$ denotes the execution time of operation v_i .

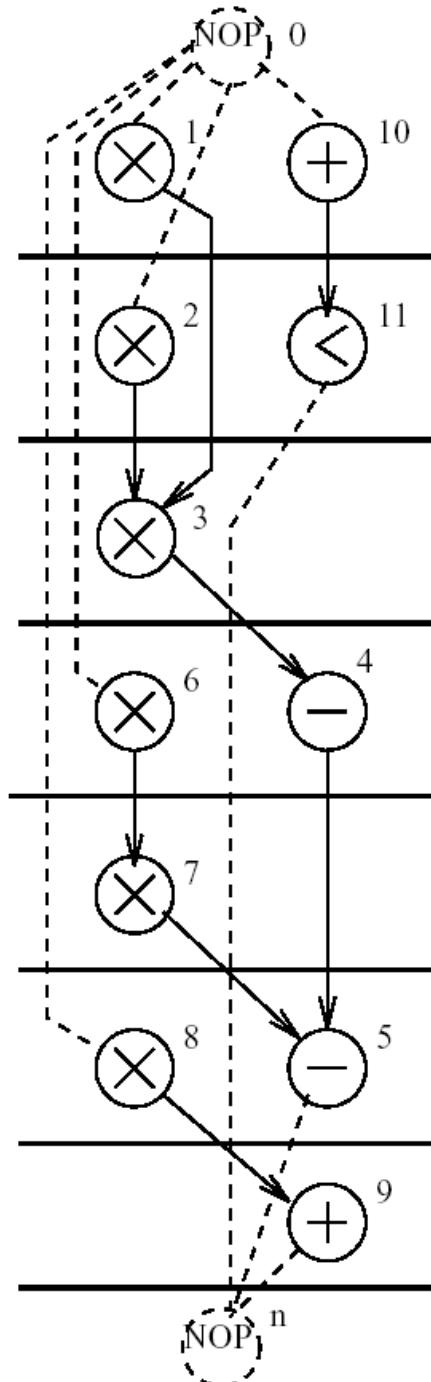
The latency L of a schedule is the time difference between start node v_0 and end node v_n :

$$L = \tau(v_n) - \tau(v_0)$$

Scheduling

► Example:

$$L = \tau(v_{12}) - \tau(v_0) = 7$$



$$\tau(v_0) = 1$$

$$\tau(v_1) = \tau(v_{10}) = 1$$

$$\tau(v_2) = \tau(v_{11}) = 2$$

$$\tau(v_3) = 3$$

$$\tau(v_6) = \tau(v_4) = 4$$

$$\tau(v_7) = 5$$

$$\tau(v_8) = \tau(v_5) = 6$$

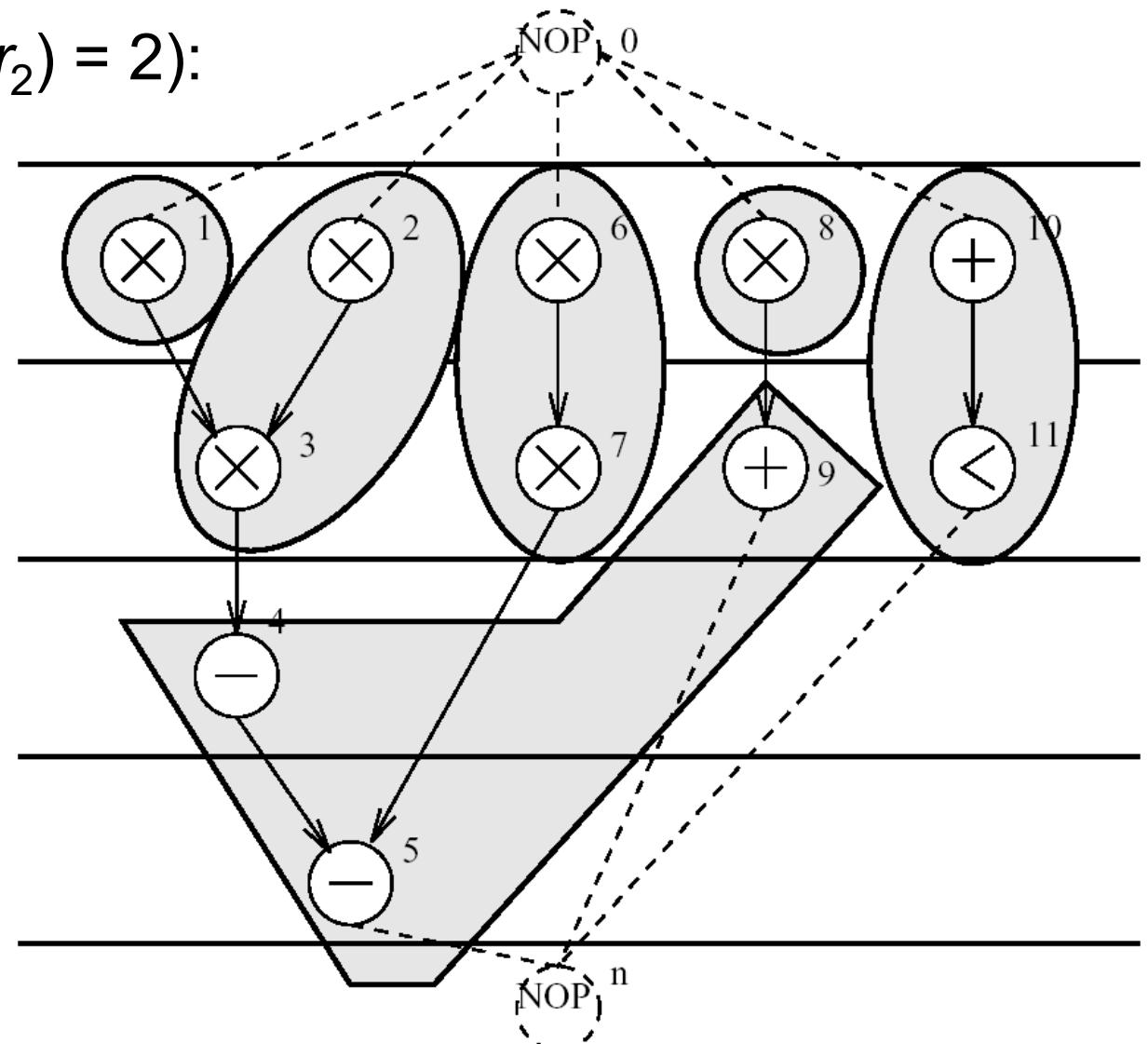
$$\tau(v_9) = 7$$

$$\tau(v_{12}) = 8$$

Binding

- ▶ Example ($\alpha(r_1) = 4$, $\alpha(r_2) = 2$):

$\beta(v_1) = r_1, \gamma(v_1) = 1,$
 $\beta(v_2) = r_1, \gamma(v_2) = 2,$
 $\beta(v_3) = r_1, \gamma(v_3) = 2,$
 $\beta(v_4) = r_2, \gamma(v_4) = 1,$
 $\beta(v_5) = r_2, \gamma(v_5) = 1,$
 $\beta(v_6) = r_1, \gamma(v_6) = 3,$
 $\beta(v_7) = r_1, \gamma(v_7) = 3,$
 $\beta(v_8) = r_1, \gamma(v_8) = 4,$
 $\beta(v_9) = r_2, \gamma(v_9) = 1,$
 $\beta(v_{10}) = r_2, \gamma(v_{10}) = 2,$
 $\beta(v_{11}) = r_2, \gamma(v_{11}) = 2$



Multiobjective Optimization

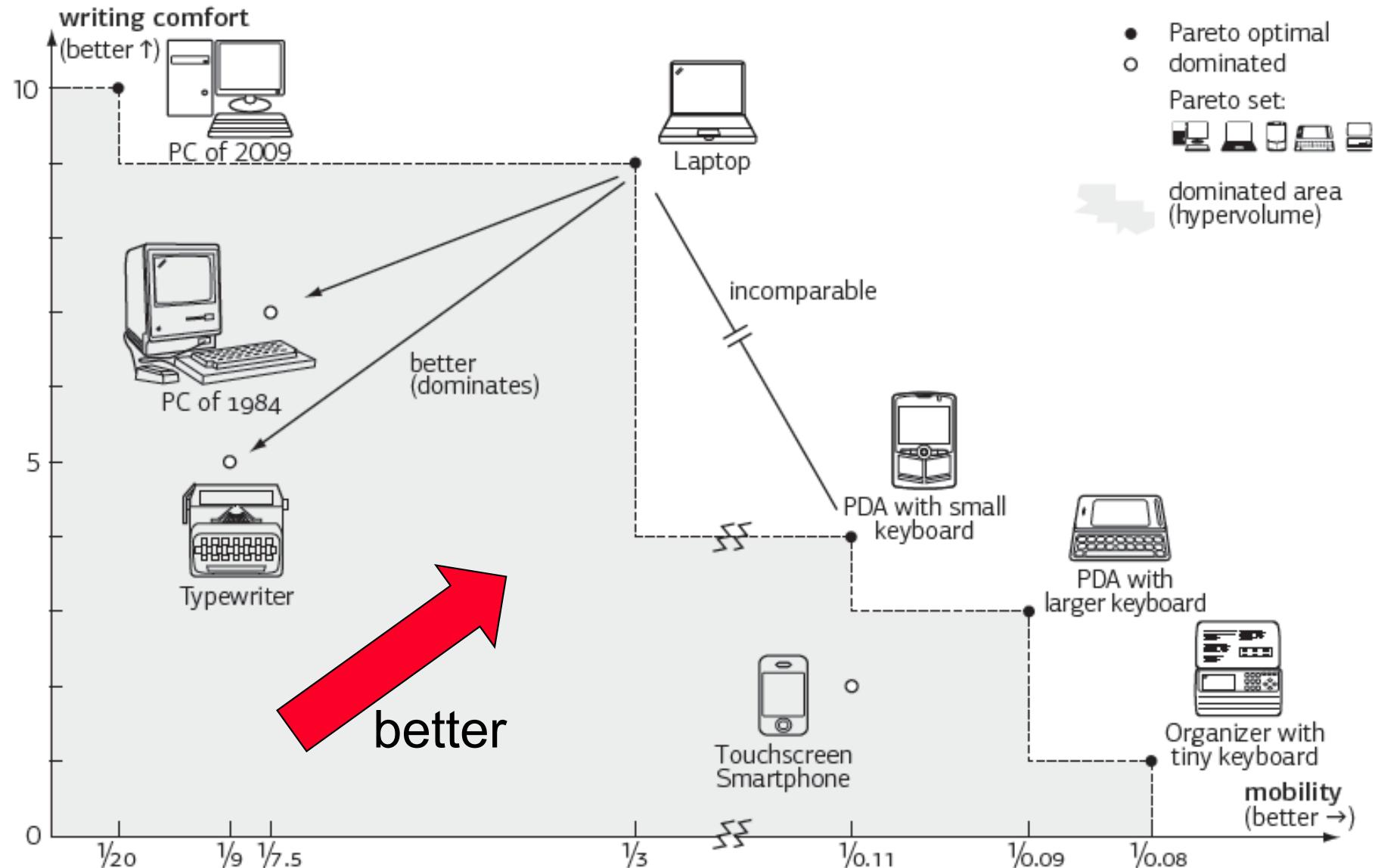
- ▶ Architecture Synthesis is an optimization problem with more than one objective:
 - Latency of the algorithm that is implemented
 - Hardware cost (memory, communication, computing units, control)
 - Power and energy consumption
- ▶ Optimization problems with several objectives are called “multiobjective optimization problems”.

Multiobjective Optimization

- ▶ Let us suppose, we would like to select a typewriting device. Criteria are
 - mobility (related to weight)
 - comfort (related to keyboard size and performance)

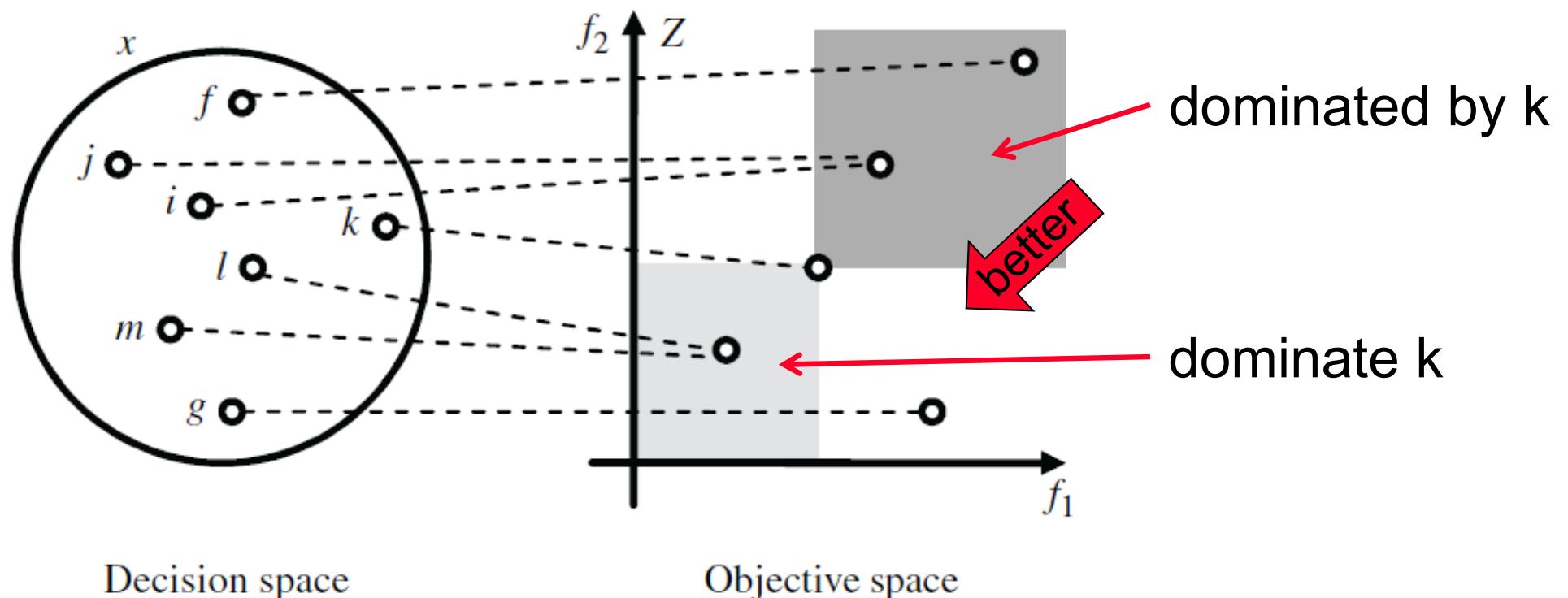
Icon	Device	weight (kg)	comfort rating
	PC of 2009	20.00	10
	PC of 1984	7.50	7
	Laptop	3.00	9
	Typewriter	9.00	5
	Touchscreen Smartphone	0.11	2
	PDA with large keyboard	0.09	3
	PDA with small keyboard	0.11	4
	Organizer with tiny keyboard	0.08	1

Multiobjective Optimization



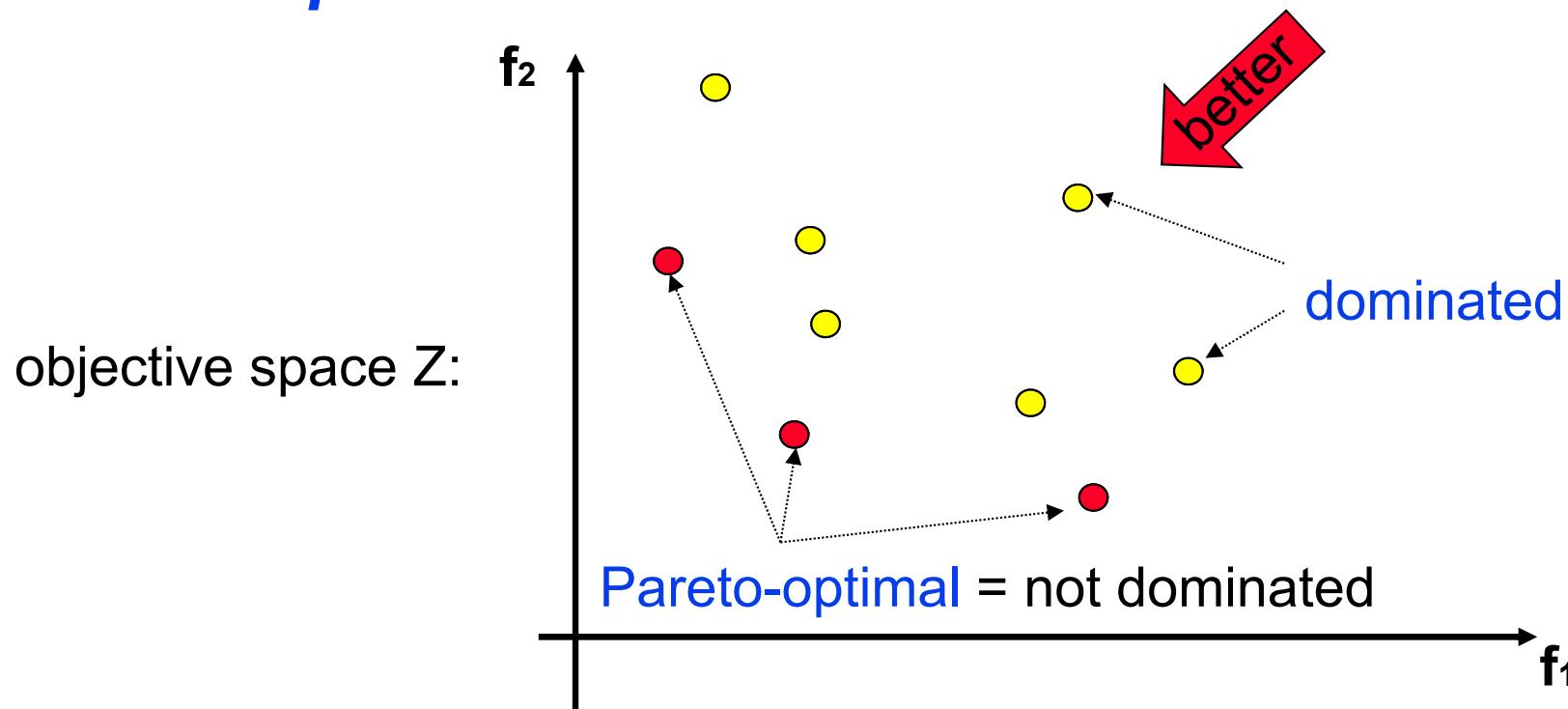
Pareto-Dominance

Definition : A solution $a \in X$ weakly Pareto-dominates a solution $b \in X$, denoted as $a \preceq b$, if it is as least as good in all objectives, i.e., $f_i(a) \leq f_i(b)$ for all $1 \leq i \leq n$. Solution a is better then b , denoted as $a \prec b$, iff $(a \preceq b) \wedge (b \not\preceq a)$.



Pareto-optimal Set

- ▶ A solution is named **Pareto-optimal**, if it is not Pareto-dominated by any other solution in X .
- ▶ The set of all Pareto-optimal solutions is denoted as the Pareto-optimal set and its image in objective space as the **Pareto-optimal front**.



Contents

- ▶ Models
- ▶ ***Scheduling without resource constraints***
 - ASAP
 - ALAP
 - Timing Constraints
- ▶ Scheduling with resource constraints
 - List Scheduling
 - Integer Linear Programming
- ▶ Iterative Algorithms
- ▶ Dynamic Voltage Scaling

Scheduling Algorithms

► *Classification*

- ***unlimited resources***: no constraints in terms of the available resources are defined.
- ***limited resources***: constraints are given in terms of the number a types of available resources.

- ***iterative algorithms***: an initial solution to the architecture synthesis is improved step by step.
- ***constructive algorithms***: the synthesis problem is solved in one step.
- ***transformative algorithms***: the initial problem formulation is converted into a (classical) optimization problem.

Scheduling Without Resource Constraints

- ▶ The scheduling method can be used
 - as a **preparatory step** for the general synthesis problem
 - to **determine bounds** on feasible schedules in the general case
 - if there is a **dedicated resource** for each operation.

Given is a sequence graph $G_S = (V_S, E_S)$ and a resource graph $G_R = (V_R, E_R)$. Then the latency minimization without resource constraints is defined as

$$L = \min\{\tau(v_n) : \tau(v_j) - \tau(v_i) \geq w(v_i) \quad \forall (v_i, v_j) \in E_S\}$$

Contents

- ▶ Models
- ▶ Scheduling without resource constraints
 - **ASAP**
 - ALAP
 - Timing Constraints
- ▶ Scheduling with resource constraints
 - List Scheduling
 - Integer Linear Programming
- ▶ Iterative Algorithms
- ▶ Dynamic Voltage Scaling

The ASAP Algorithm

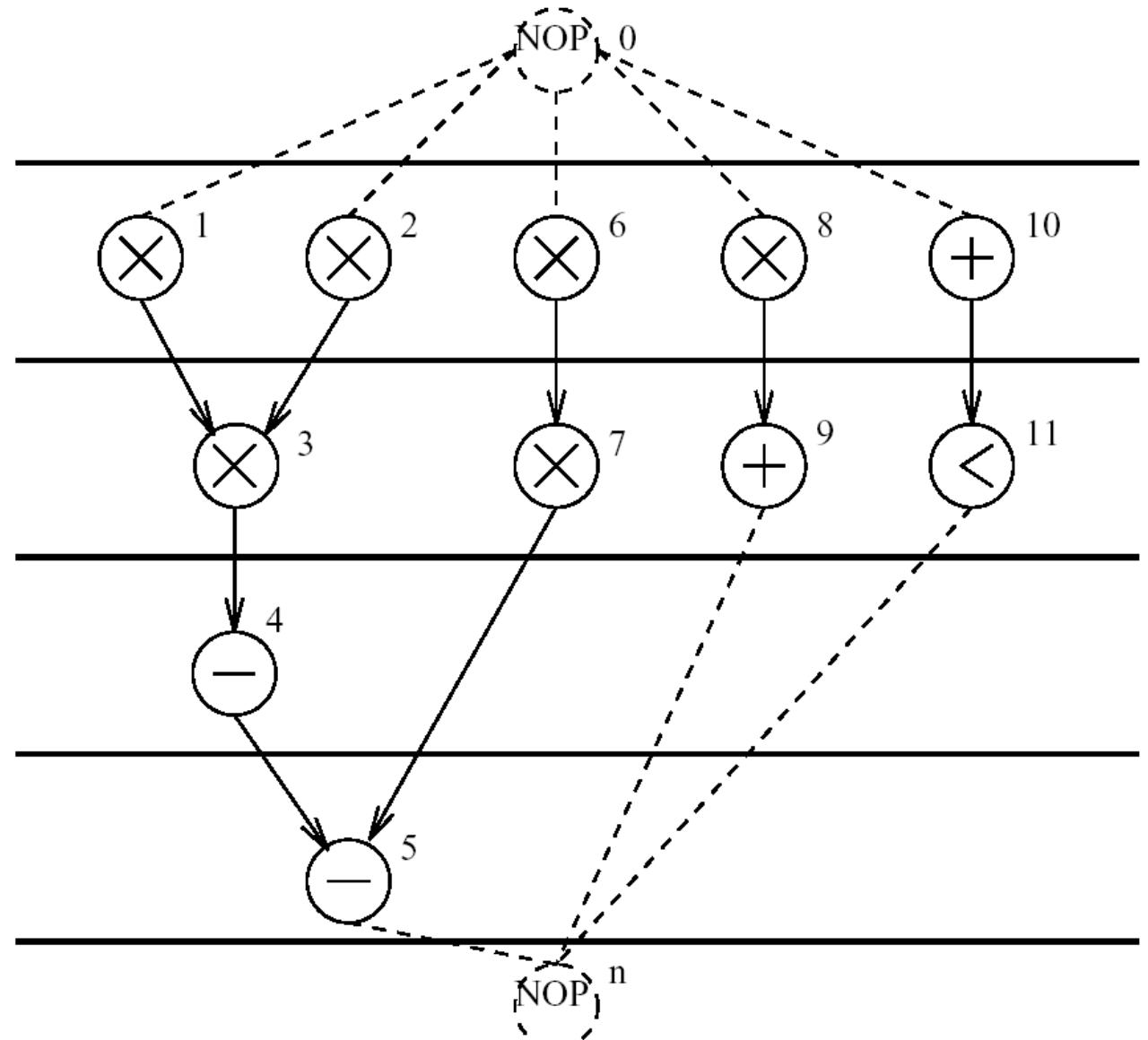
- ▶ **ASAP = As Soon As Possible**

```
ASAP( $G_S(V_S, E_S), w$ ) {  
     $\tau(v_0) = 1$ ;  
    REPEAT {  
        Determine  $v_i$  whose predec. are planed;  
         $\tau(v_i) = \max\{\tau(v_j) + w(v_j) \ \forall (v_j, v_i) \in E_S\}$   
    } UNTIL ( $v_n$  is planned);  
    RETURN ( $\tau$ );  
}
```

The ASAP Algorithm

- ▶ **Example:**

$$w(v_i) = 1$$



Contents

- ▶ Models
- ▶ Scheduling without resource constraints
 - ASAP
 - **ALAP**
 - Timing Constraints
- ▶ Scheduling with resource constraints
 - List Scheduling
 - Integer Linear Programming
- ▶ Iterative Algorithms
- ▶ Dynamic Voltage Scaling

The ALAP Algorithm

- ▶ **ALAP = As Late As Possible**

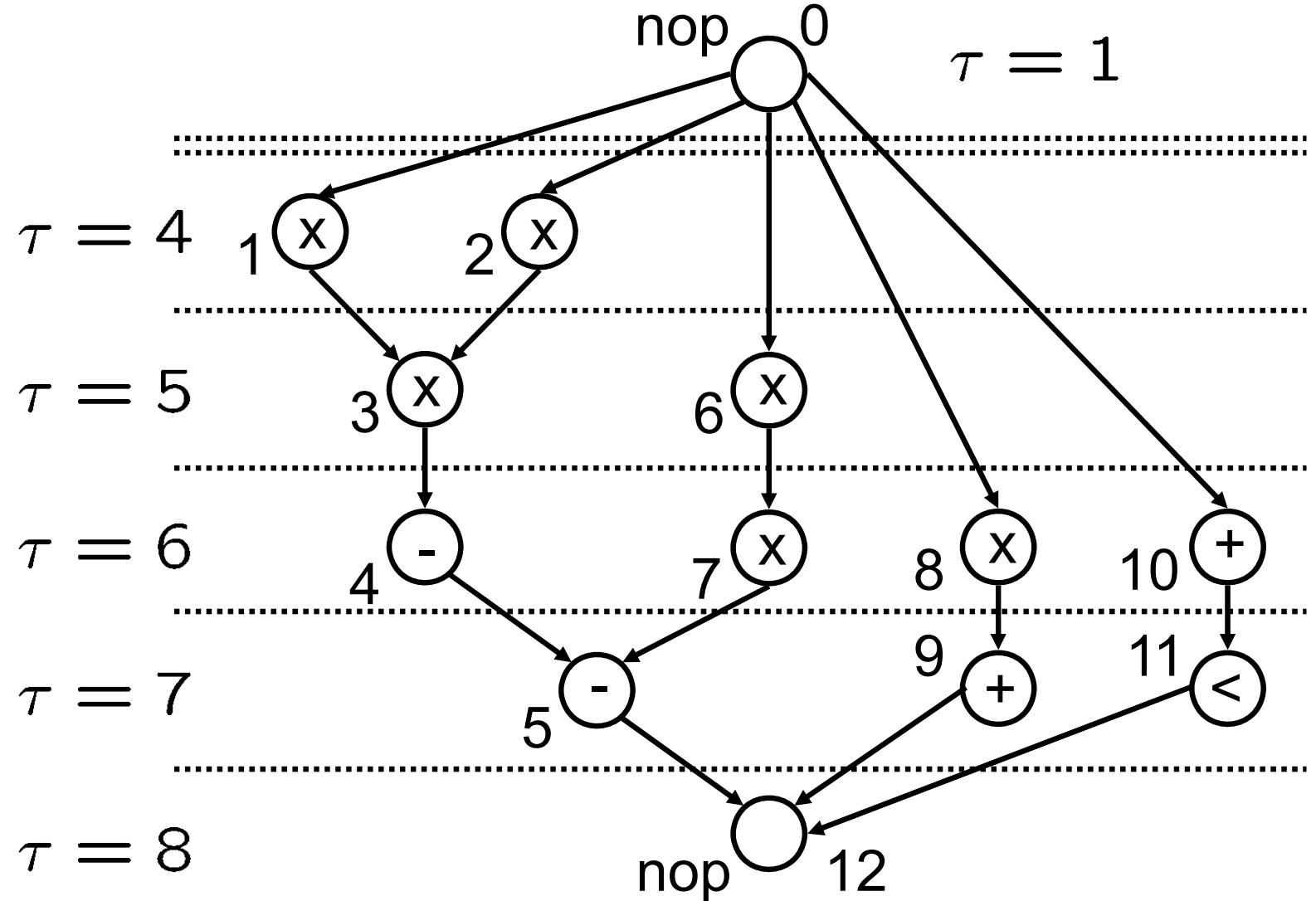
```
ALAP( $G_S(V_S, E_S)$ ,  $w$ ,  $L_{max}$ ) {  
     $\tau(v_n) = L_{max} + 1$ ;  
    REPEAT {  
        Determine  $v_i$  whose succ. are planned;  
         $\tau(v_i) = \min\{\tau(v_j) \forall (v_i, v_j) \in E_S\} - w(v_i)$   
    } UNTIL ( $v_0$  is planned);  
    RETURN ( $\tau$ );  
}
```

The ALAP Algorithm

- ▶ **Example:**

$$L_{\max} = 7$$

$$w(v_i) = 1$$



Contents

- ▶ Models
- ▶ Scheduling without resource constraints
 - ASAP
 - ALAP
 - ***Timing Constraints***
- ▶ Scheduling with resource constraints
 - List Scheduling
 - Integer Linear Programming
- ▶ Iterative Algorithms
- ▶ Dynamic Voltage Scaling

Scheduling with Timing Constraints

- ▶ Different classes of timing constraints:
 - **deadline** (latest finishing times of operations), for example

$$\tau(v_2) + w(v_2) \leq 5$$

- **release times** (earliest starting times of operations), for example

$$\tau(v_3) \geq 4$$

- **relative constraints** (differences between starting times of a pair of operations), for example

$$\tau(v_6) - \tau(v_7) \geq 4$$

$$\tau(v_4) - \tau(v_1) \leq 2$$

Scheduling with Timing Constraints

- ▶ We will model all timing constraints using **relative constraints**. Deadlines and release times are defined relative to the start node v_0 .
- ▶ Minimum, maximum and equality constraints can be converted into each other:

- **Minimum constraint:**

$$\tau(v_j) \geq \tau(v_i) + l_{ij} \longrightarrow \tau(v_j) - \tau(v_i) \geq l_{ij}$$

- **Maximum constraint:**

$$\tau(v_j) \leq \tau(v_i) + l_{ij} \longrightarrow \tau(v_i) - \tau(v_j) \geq -l_{ij}$$

- **Equality constraint:**

$$\begin{aligned}\tau(v_j) = \tau(v_i) + l_{ij} \longrightarrow \tau(v_j) - \tau(v_i) &\leq l_{ij} \wedge \\ &\tau(v_j) - \tau(v_i) \geq l_{ij}\end{aligned}$$

Weighted Constraint Graph

- ▶ Timing constraints can be represented in form of a weighted constraint graph:

A weighted constraint graph $G_C = (V_C, E_C, d)$ related to a sequence graph $G_S = (V_S, E_S)$ contains nodes $V_C = V_S$ and a weighted edge for each timing constraint. An edge $(v_i, v_j) \in E_C$ with weight $d(v_i, v_j)$ denotes the constraint $\tau(v_j) - \tau(v_i) \geq d(v_i, v_j)$.

Weighted Constraint Graph

- ▶ In order to represent a **feasible schedule**, we have one edge corresponding to each precedence constraint with

$$d(v_i, v_j) = w(v_i)$$

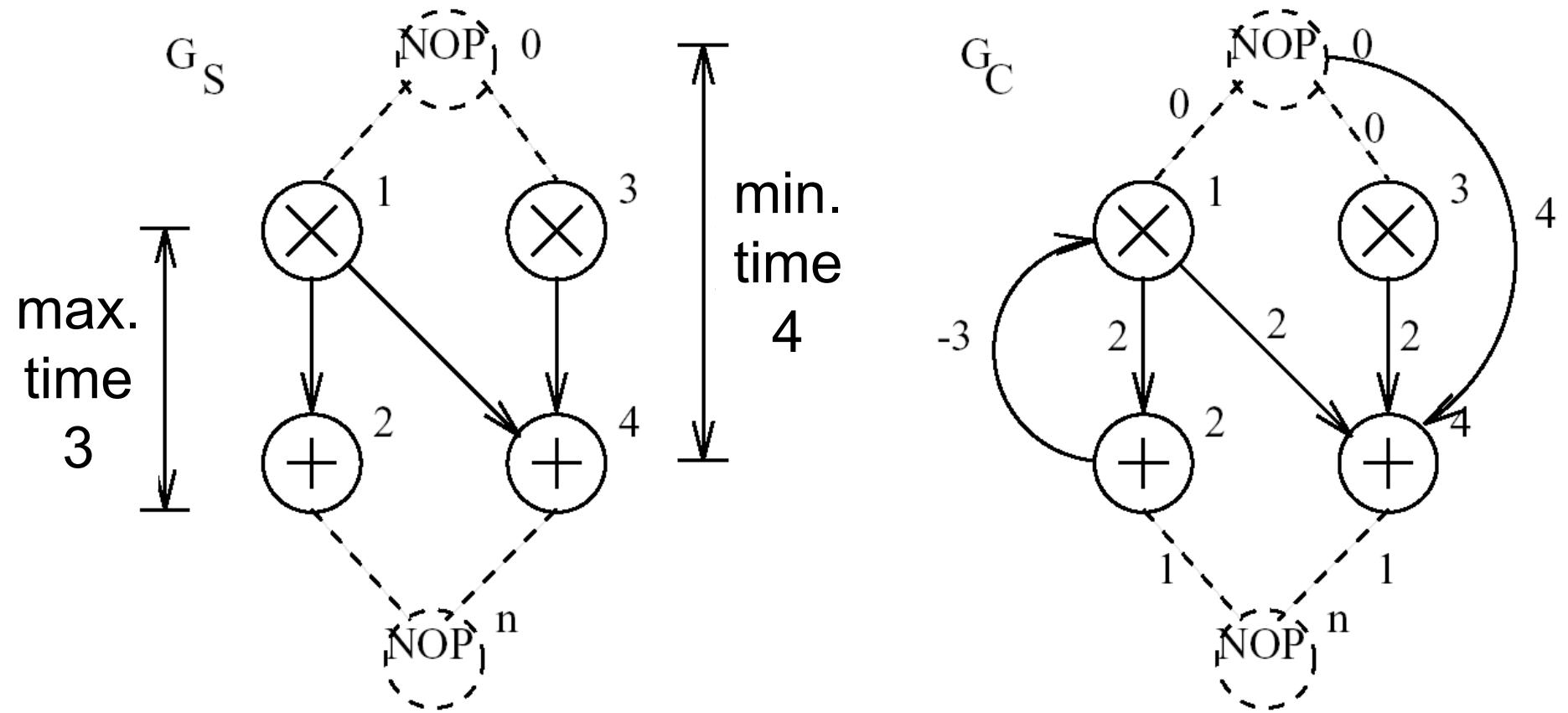
where $w(v_i)$ denotes the execution time of v_i .

- ▶ A consistent assignment of starting times $\tau(v_i)$ to all operations can be done by solving a **single source longest path** problem.
- ▶ A possible algorithm (**Bellman-Ford**) has complexity $O(|V_C| |E_C|)$:

Iteratively set $\tau(v_j) := \max\{\tau(v_j), \tau(v_i) + d(v_i, v_j) : (v_i, v_j) \in E_C\}$ for all $v_j \in V_C$ starting from $\tau(v_i) = -\infty$ for $v_i \in V_C \setminus \{v_0\}$ and $\tau(v_0) = 1$.

Weighted Constraint Graph

- ▶ **Example:** $w(v_1) = w(v_3) = 2 \quad w(v_2) = w(v_4) = 1$
 $\tau(v_0) = \tau(v_1) = \tau(v_3) = 1, \tau(v_2) = 3,$
 $\tau(v_4) = 5, \tau(v_n) = 6, L = \tau(v_n) - \tau(v_0) = 5$



Contents

- ▶ Models
- ▶ Scheduling without resource constraints
 - ASAP
 - ALAP
 - Timing Constraints
- ▶ ***Scheduling with resource constraints***
 - List Scheduling
 - Integer Linear Programming
- ▶ Iterative Algorithms
- ▶ Dynamic Voltage Scaling

Scheduling With Resource Constraints

Given is a sequence graph $G_S = (V_S, E_S)$, a resource graph $G_R = (V_R, E_R)$ and an associated allocation α and binding β .

Then the minimal latency is defined as

$$\begin{aligned} L = \min\{ & \tau(v_n) : \\ & (\tau(v_j) - \tau(v_i) \geq w(v_i, \beta(v_i)) \quad \forall (v_i, v_j) \in E_S) \wedge \\ & (|\{v_s : \beta(v_s) = v_t \wedge \tau(v_s) \leq t < \tau(v_s) + w(v_s, v_t)\}| \leq \alpha(v_t) \\ & \forall v_t \in V_T, \forall 1 \leq t \leq L_{max}) \} \end{aligned}$$

where L_{max} denotes an upper bound on the latency.

Contents

- ▶ Models
- ▶ Scheduling without resource constraints
 - ASAP
 - ALAP
 - Timing Constraints
- ▶ Scheduling with resource constraints
 - *List Scheduling*
 - Integer Linear Programming
- ▶ Iterative Algorithms
- ▶ Dynamic Voltage Scaling

List Scheduling

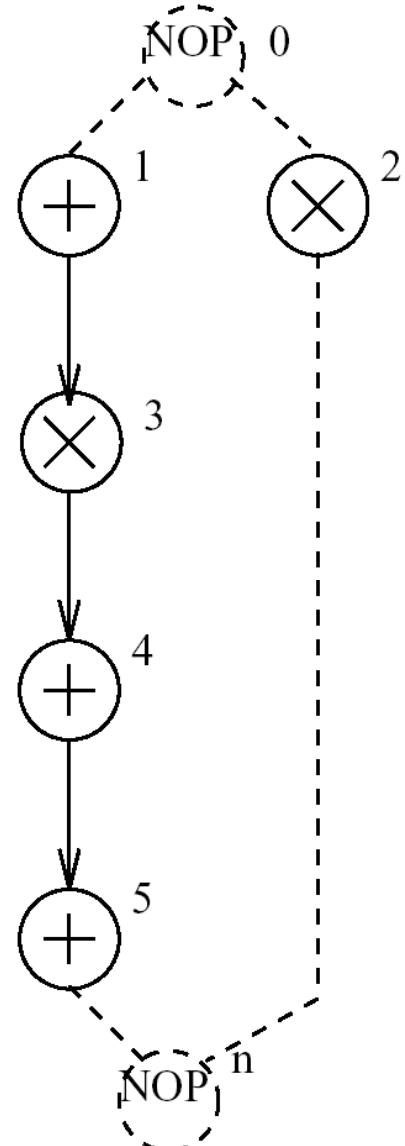
```
LIST( $G_S(V_S, E_S), G_R(V_R, E_R), \alpha, \beta, priorities$ ) {
     $t = 1;$ 
    REPEAT {
        FORALL  $v_k \in V_T$  {
            determine candidates to be scheduled  $U_k$ ;
            determine running operations  $T_k$ ;
            choose  $S_k \subseteq U_k$  with maximal priority
                and  $|S_k| + |T_k| \leq \alpha(v_k)$ ;
             $\tau(v_i) = t \quad \forall v_i \in S_k;$  }
         $t = t + 1;$ 
    } UNTIL ( $v_n$  planned)
    RETURN ( $\tau$ );
}
```

List Scheduling

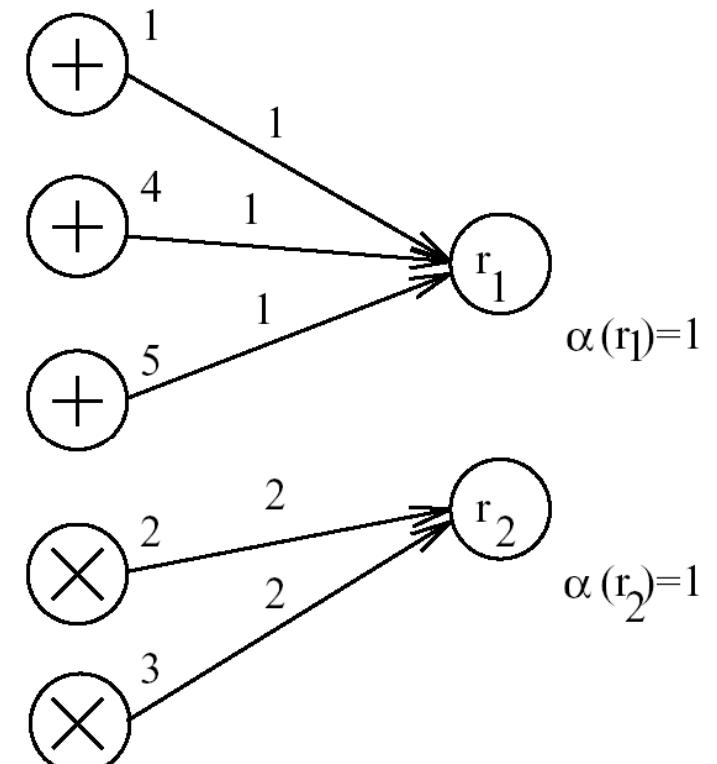
- ▶ One of the most **widely used** algorithms for scheduling under resource constraints.
- ▶ **Principles:**
 - To each operation there is a **priority** assigned which denotes the urgency of being scheduled. This **priority is static**, i.e. determined before the List Scheduling.
 - The algorithm schedules **one time step after the other**.
 - U_k denotes the set of operations that (a) are mapped onto resource v_k and whose predecessors finished.
 - T_k denotes the currently running operations mapped to resource v_k .

List Scheduling

► **Example:** G_S

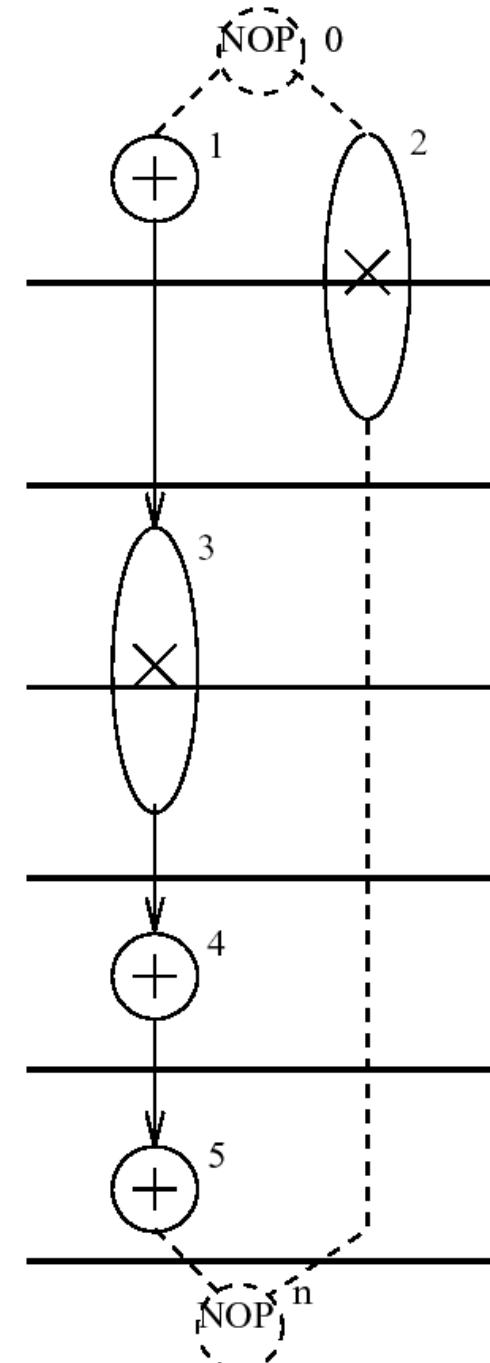


b) G_R



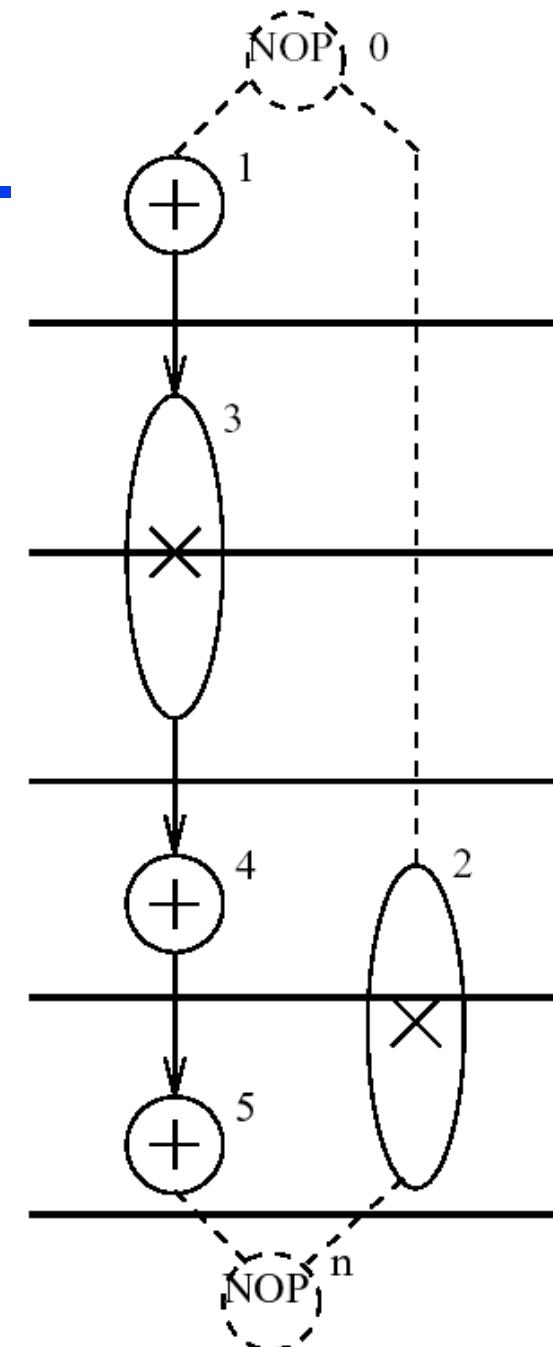
List Scheduling

- ▶ Solution via *list scheduling*:
 - In the example, the solution is independent of priority.
 - Because of the *greedy* principle, all resources are directly occupied.
 - List scheduling is a *heuristic algorithm*.
In this example, it does not yield the minimal latency!



List Scheduling

- ▶ Solution via an *optimal method*:
 - *Latency is smaller* than with list scheduling.
 - An example of an optimal algorithm is the transformation into an *integer linear program*.

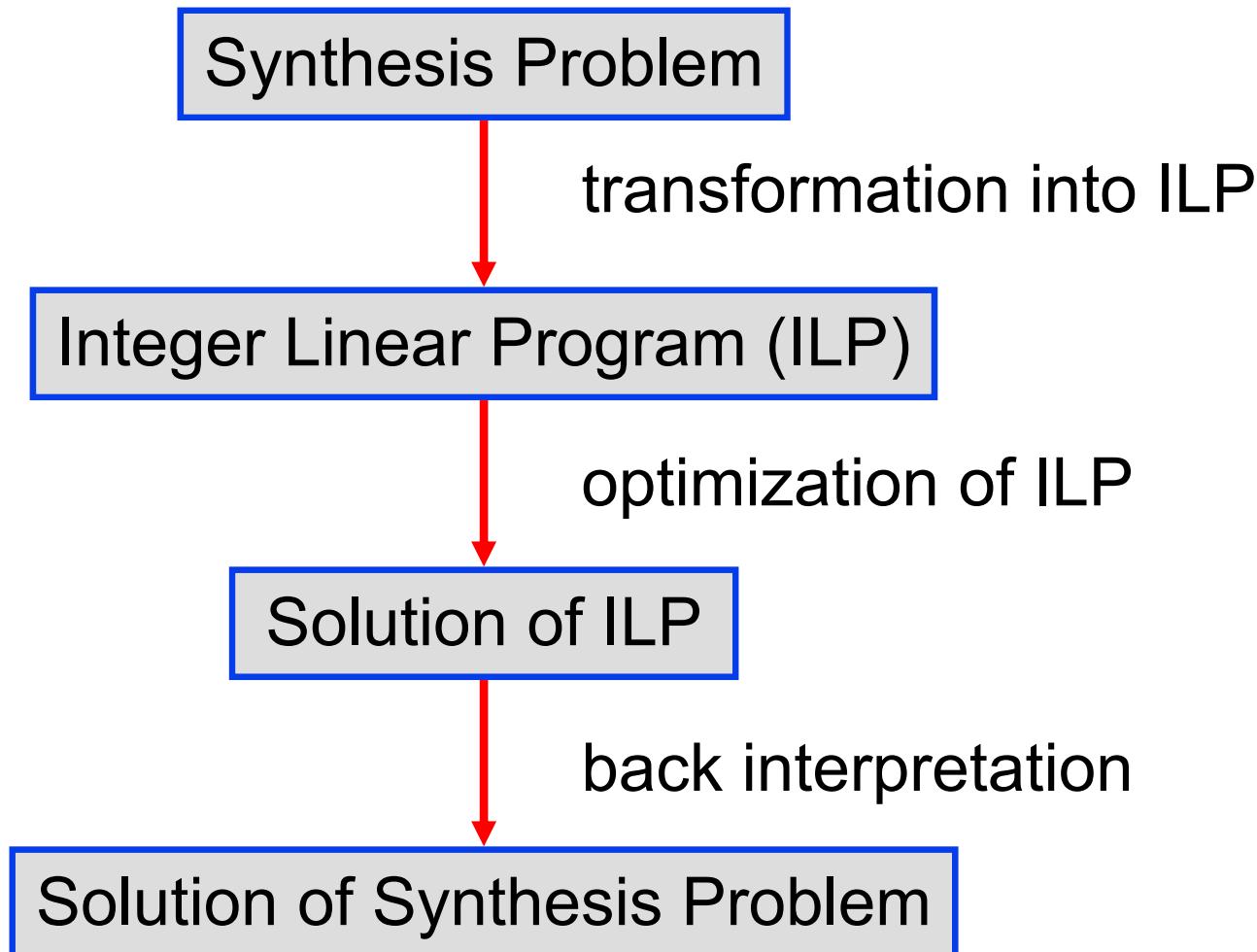


Contents

- ▶ Models
- ▶ Scheduling without resource constraints
 - ASAP
 - ALAP
 - Timing Constraints
- ▶ Scheduling with resource constraints
 - List Scheduling
 - *Integer Linear Programming*
- ▶ Iterative Algorithms
- ▶ Dynamic Voltage Scaling

Integer Linear Programming

► *Principle:*



Integer Linear Program

- ▶ Yields ***optimal solution*** to synthesis problems as it is based on an exact mathematical description of the problem.
- ▶ Solves ***scheduling, binding and allocation*** simultaneously.
- ▶ ***Standard optimization*** approaches (and software) are available to solve integer linear programs:
 - in addition to linear programs (linear constraints, linear objective function) some variables are forced to be integers.
 - much more complex than solving linear program
 - efficient methods are based on (a) branch and bound methods and (b) determining additional hyperplanes (cuts).

Integer Linear Program

- ▶ Many **variants** exist, depending on available information, constraints and objectives, e.g. minimize latency, minimize resources, minimize memory. Just an example is given here!!
- ▶ For the following example, we use the **assumptions**:
 - The **binding is determined** already, i.e. every operation v_i has a unique execution time $w(v_i)$.
 - We have determined the **earliest and latest starting times** of operations v_i as l_i and h_i , respectively. To this end, we can use the ASAP and ALAP algorithms that have been introduced earlier. The maximal latency L_{\max} is chosen such that a feasible solution to the problem exists.

Integer Linear Program

minimize: $\tau(v_n) - \tau(v_0)$

subject to $x_{i,t} \in \{0, 1\} \quad \forall v_i \in V_S \quad \forall t : l_i \leq t \leq h_i \quad (1)$

$$\sum_{t=l_i}^{h_i} x_{i,t} = 1 \quad \forall v_i \in V_S \quad (2)$$

$$\sum_{t=l_i}^{h_i} t \cdot x_{i,t} = \tau(v_i) \quad \forall v_i \in V_S \quad (3)$$

$$\tau(v_j) - \tau(v_i) \geq w(v_i) \quad \forall (v_i, v_j) \in E_S \quad (4)$$

$$\sum_{\forall i: (v_i, v_k) \in E_R} \sum_{p'=\max\{0, t-h_i\}}^{\min\{w(v_i)-1, t-l_i\}} x_{i,t-p'} \leq \alpha(v_k) \quad \forall v_k \in V_T \quad \forall t : 1 \leq t \leq \max\{h_i : v_i \in V_S\} \quad (5)$$

Integer Linear Program

► *Explanations:*

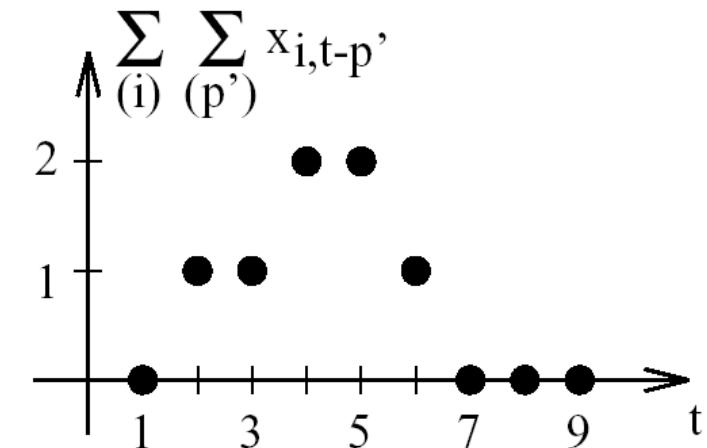
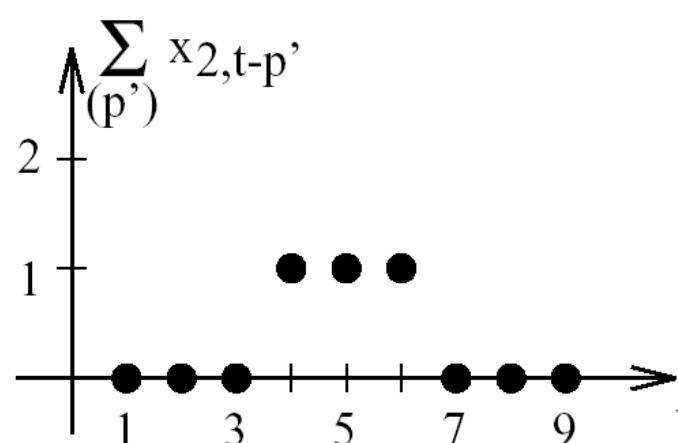
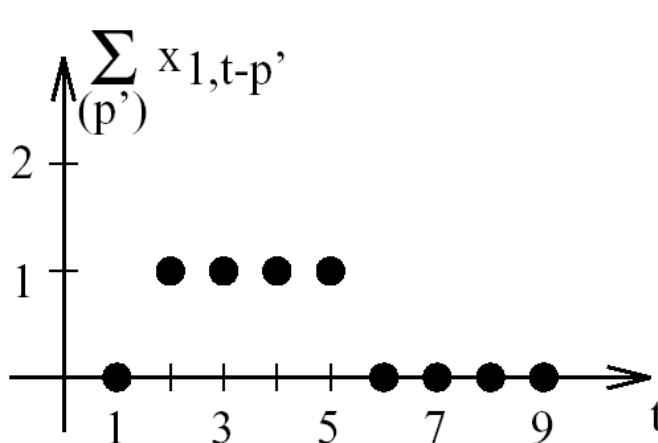
- (1) declares variables x to be binary .
- (2) makes sure that exactly one variable $x_{i,t}$ for all t has the value 1, all others are 0.
- (3) determines the relation between variables x and starting times of operations τ . In particular, if $x_{i,t} = 1$ then the operation v_i starts at time t , i.e. $\tau(v_i) = t$.
- (4) guarantees, that all precedence constraints are satisfied.
- (5) makes sure, that the resource constraints are not violated. For all resource types $v_k \in V_T$ and for all time instances t it is guaranteed that the number of active operations does not increase the number of available resource instances.

Integer Linear Program

► *Explanations:*

- (5) The first sum selects all operations that are mapped onto resource type v_k . The second sum considers all time instances where operation v_i is occupying resource type v_k :

$$\sum_{p'=0}^{w(v_i)-1} x_{i,t-p'} = \begin{cases} 1 & : \quad \forall t : \tau(v_i) \leq t \leq \tau(v_i) + w(v_i) - 1 \\ 0 & : \quad \text{sonst} \end{cases}$$



Contents

- ▶ Models
- ▶ Scheduling without resource constraints
 - ASAP
 - ALAP
 - Timing Constraints
- ▶ Scheduling with resource constraints
 - List Scheduling
 - Integer Linear Programming
- ▶ ***Iterative Algorithms***
- ▶ Dynamic Voltage Scaling

Iterative Algorithms

- ▶ **Iterative algorithms** consist of a set of indexed equations that are evaluated for all values of an index variable l :

$$x_i[l] = F_i[\dots, x_j[l - d_{ji}], \dots] \quad \forall l \quad \forall i \in I$$

Here, x_i denote a set of indexed variables, F_i denote arbitrary functions and d_{ji} are constant index displacements.

- ▶ Examples of well known representations are **signal flow graphs** (as used in signal and image processing and automatic control), **marked graphs** and special forms of **loops**.

Iterative Algorithms

- ▶ **Several representations** of the same iterative algorithm:
 - One **indexed equation** with constant index dependencies:

$$y[l] = au[l] + by[l - 1] + cy[l - 2] + dy[l - 3] \quad \forall l$$

- Equivalent set of indexed equations:

$$x_1[l] = au[l] \quad \forall l$$

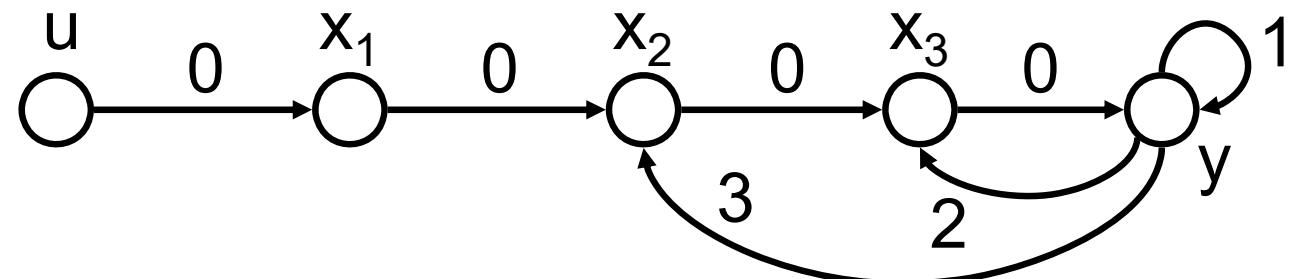
$$x_2[l] = x_1[l] + dy[l - 3] \quad \forall l$$

$$x_3[l] = x_2[l] + cy[l - 2] \quad \forall l$$

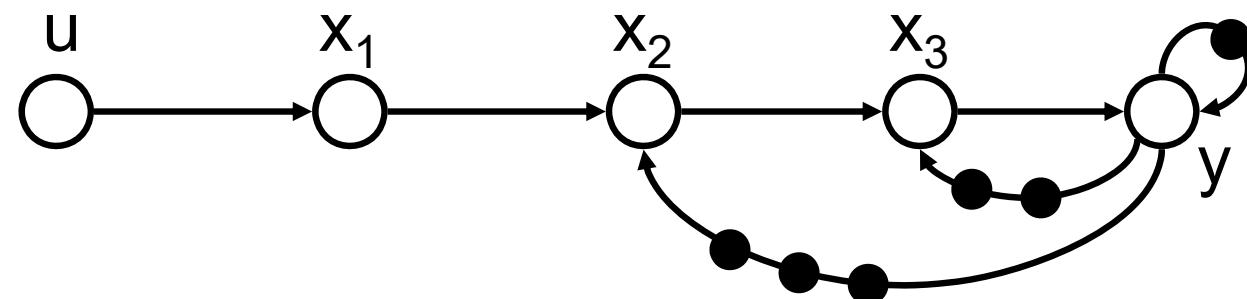
$$y[l] = x_3[l] + by[l - 1] \quad \forall l$$

Iterative Algorithms

- **Extended sequence graph** $G_S = (V_S, E_S, d)$: To each edge $(v_i, v_j) \in E_S$ there is associated the **index displacement** d_{ij} . An edge $(v_i, v_j) \in E_S$ denotes that the variable corresponding to v_j depends on variable corresponding to v_i with displacement d_{ij} .

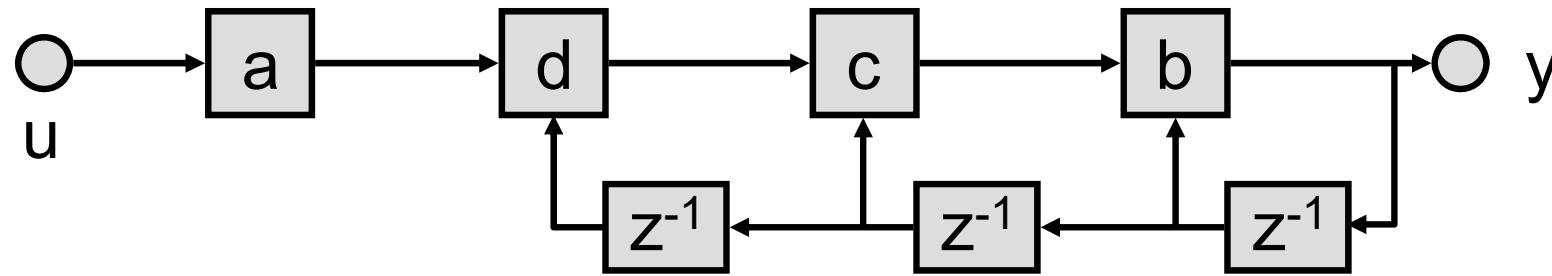


- Equivalent **marked graph**:



Iterative Algorithms

- ▶ Equivalent *signal flow graph*:



- ▶ Equivalent *loop program*:

```
while(true) {
    t1 = read(u);
    t5 = a*t1 + d*t2 + c*t3 + b*t4;
    t2 = t3;
    t3 = t4;
    t4 = t5;
    write(y, t5);}
```

Iterative Algorithms

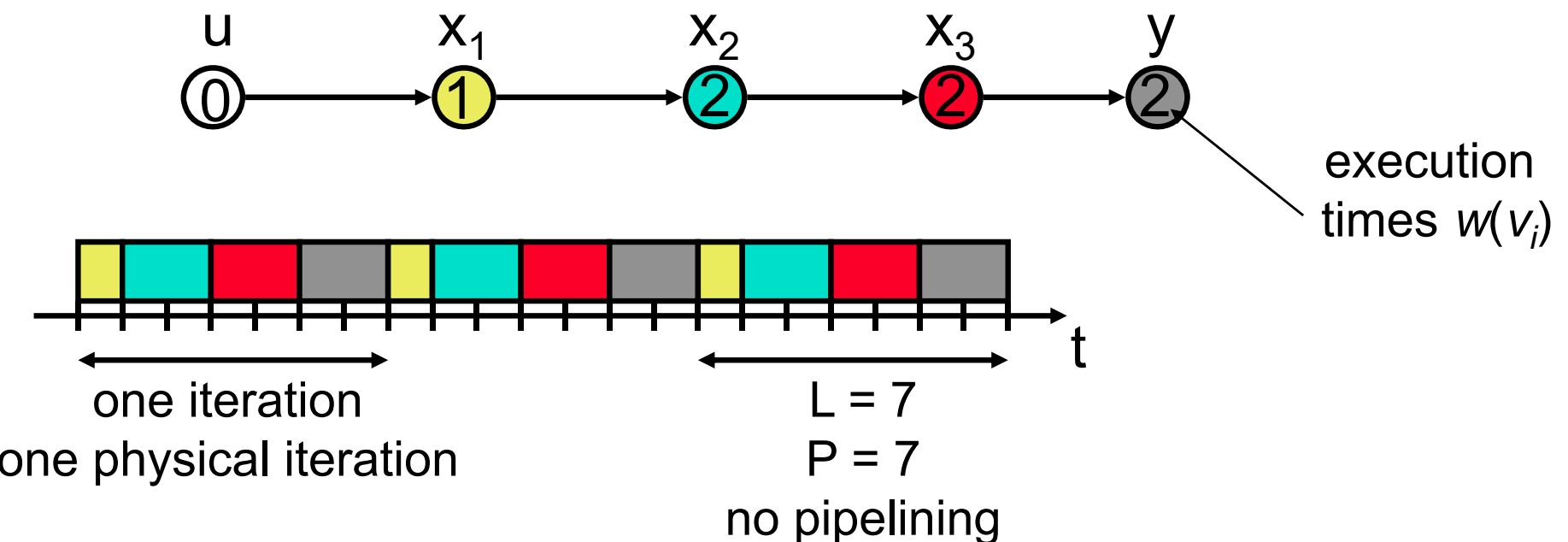
- ▶ An **iteration** is the set of all operations necessary to compute all variables $x_i[l]$ for a fixed index l .
- ▶ The **iteration interval** P is the time distance between two successive iterations of an iterative algorithm. $1/P$ denotes the **throughput** of the implementation.
- ▶ The **latency** L is the maximal time distance between the starting and the finishing times of operations belonging to one iteration.
- ▶ In a pipelined implementation (**functional pipelining**), there exist time instances where the operations of different iterations / are executed simultaneously.
- ▶ In case of **loop folding**, starting and finishing times of an operation are in different physical iterations.

Iterative Algorithms

► *Implementation principles*

- A **simple possibility**, the edges with $d_{ij} > 0$ are removed from the extended sequence graph. The resulting simple sequence graph is implemented using **standard methods**.

Example with unlimited resources:

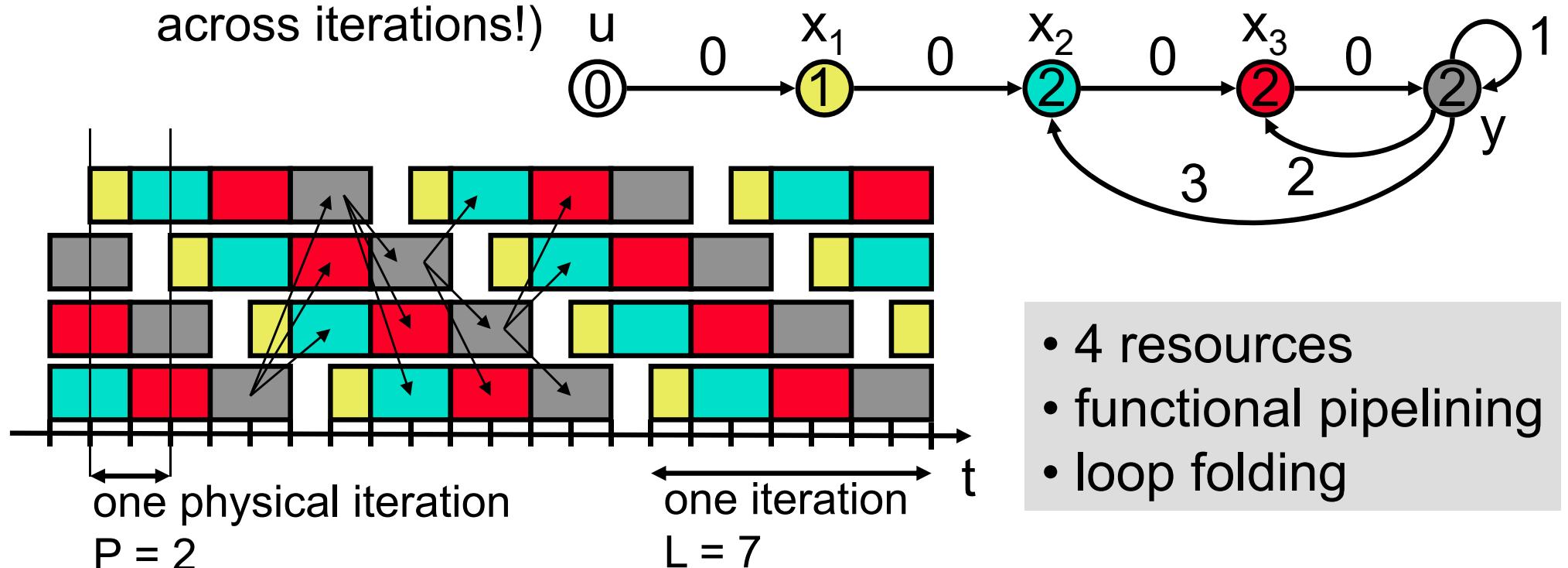


Iterative Algorithms

► *Implementation principles*

- Using ***functional pipelining***: Successive iterations overlap and a higher throughput ($1/P$) is obtained.

Example with unlimited resources (note data dependencies across iterations!)



Iterative Algorithms

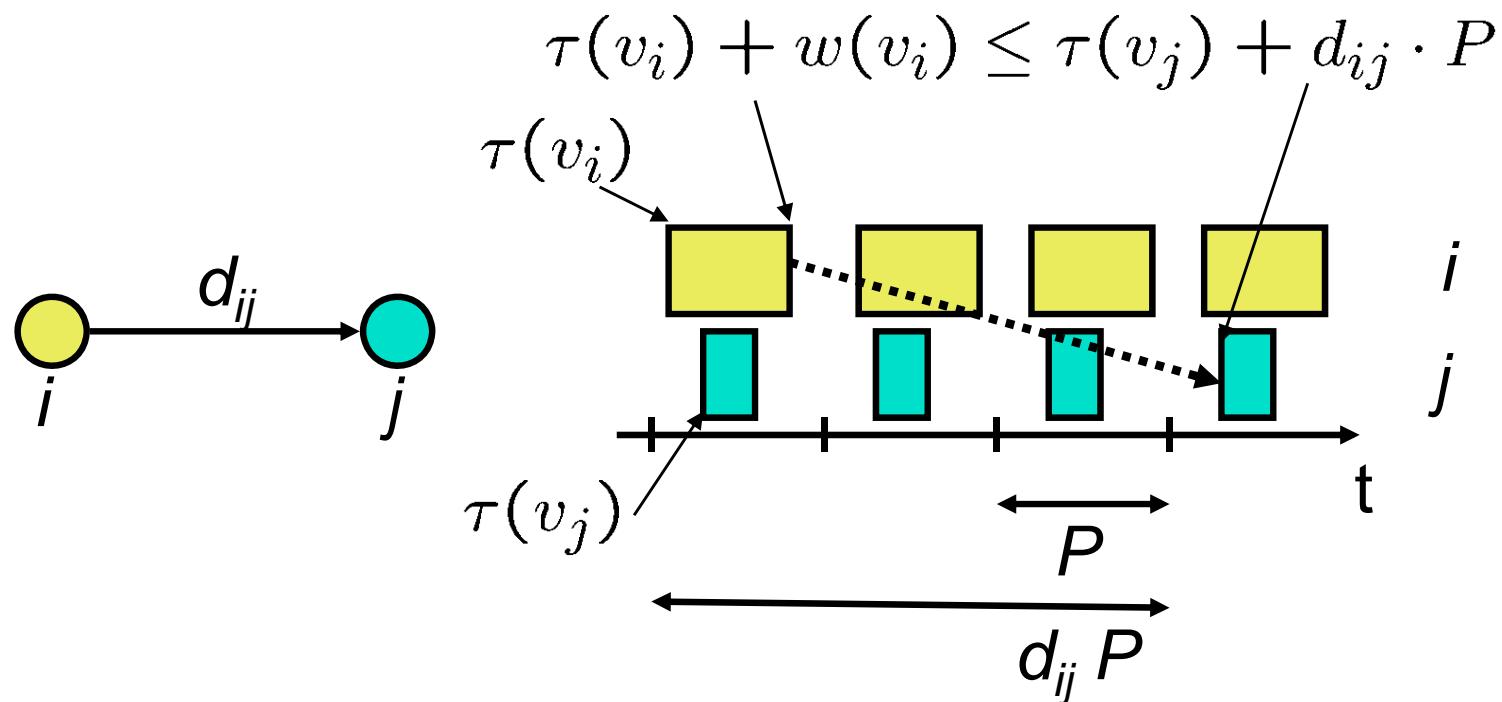
- ▶ Solving the synthesis problem using ***integer linear programming***:
 - Starting point is the ILP formulation given for simple sequence graphs.
 - Now, we use the ***extended sequence graph*** (including displacements d_{ij}).
 - **ASAP** and **ALAP** scheduling for upper and lower bounds h_i and l_i use only edges with $d_{ij} = 0$ (remove dependencies across iterations).
 - We suppose, that a suitable ***iteration interval*** P is chosen beforehand. If it is too small, no feasible solution to the ILP exists and P needs to be increased.

Iterative Algorithms

- Eqn.(4) is replaced by:

$$\tau(v_j) - \tau(v_i) \geq w(v_i) - d_{ij} \cdot P \quad \forall (v_i, v_j) \in E_S$$

Proof of correctness:



Iterative Algorithms

- Eqn. (5) is replaced by

$$\sum_{\forall i: (v_i, v_k) \in E_R} \sum_{p'=0}^{w(v_i)-1} \sum_{\forall p: l_i \leq t - p' + p \cdot P \leq h_i} x_{i,t-p'+p \cdot P} \leq \alpha(v_k)$$
$$\forall 1 \leq t \leq P, \forall v_k \in V_T$$

Sketch of **Proof**: An operation v_i starting at $\tau(v_i)$ uses the corresponding resource at time steps t with

$$t = \tau(v_i) + p' - p \cdot P$$
$$\forall p', p : 0 \leq p' < w(v_i) \wedge l_i \leq t - p' + p \cdot P \leq h_i$$

Therefore, we obtain

$$\sum_{p'=0}^{w(v_i)-1} \sum_{\forall p: l_i \leq t - p' + p \cdot P \leq h_i} x_{i,t-p'+p \cdot P}$$

Contents

- ▶ Models
- ▶ Scheduling without resource constraints
 - ASAP
 - ALAP
 - Timing Constraints
- ▶ Scheduling with resource constraints
 - List Scheduling
 - Integer Linear Programming
- ▶ Iterative Algorithms
- ▶ ***Dynamic Voltage Scaling***

Dynamic Voltage Scaling

- If we transform the DVS problem into an integer linear program optimization: we can **optimize the energy** in case of **dynamic voltage scaling**.
- As an **example**, let us model a set of tasks with dependency constraints.
 - We suppose that a task $v_i \in V_S$ can use one of the execution times $w_k(v_i) \forall k \in K$ and corresponding energy $e_k(v_i)$. There are $|K|$ different voltage levels.
 - We suppose that there are **deadlines** $d(v_i)$ for each operation v_i .
 - We suppose that there are no resource constraints, i.e. all tasks can be executed in parallel.

Dynamic Voltage Scaling

$$\begin{aligned} \text{minimize: } & \sum_{k \in K} \sum_{v_i \in V_S} y_{ik} \cdot e_k(v_i) \\ \text{subject to: } & y_{ik} \in \{0, 1\} \quad \forall v_i \in V_S, k \in K \end{aligned} \quad (1)$$

$$\sum_{k \in K} y_{ik} = 1 \quad \forall v_i \in V_S \quad (2)$$

$$\tau(v_j) - \tau(v_i) \geq \sum_{k \in K} y_{ik} \cdot w_k(v_i) \quad \forall (v_i, v_j) \in E_S \quad (3)$$

$$\tau(v_i) + \sum_{k \in K} y_{ik} \cdot w_k(v_i) \leq d(v_i) \quad \forall v_i \in V_S \quad (4)$$

Dynamic Voltage Scaling

► *Explanations:*

- The objective functions just sums up all individual energies of operations.
- Eqn. (1) makes decision variables y_{ik} binary.
- Eqn. (2) guarantees that exactly one implementation (voltage) $k \in K$ is chosen for each operation v_i .
- Eqn. (3) implements the precedence constraints, where the actual execution time is selected from the set of all available ones.
- Eqn. (4) guarantees deadlines.