# Embedded Systems

## 1 - Introduction

© Lothar Thiele

Computer Engineering and Networks Laboratory

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Lecture Organization

**Herbstsemester 2021**

**227-0124-00L Embedded Systems**

**Daten der Belegungseinschränkung**

| Platzzahl | Aktuelle Belegung |
|---|---|
| 200 | 200 |

**Anzahl Studierende in der Warteliste zum Zeitpunkt 14.09.2021 13:23:37: 122**

# Organization

WWW: https://www.tec.ee.ethz.ch/education/lectures/embedded-systems.html

Lecture: Lothar Thiele, thiele@ethz.ch; Michele Magno <michele.magno@pbl.ee.ethz.ch>

Coordination: Seonyeong Heo (ETZ D97.7) <seoheo@ethz.ch>

References:

- *P. Marwedel:* Embedded System Design, Springer, ISBN 978-3-319-85812-8/978-3-030-60909-2, 2018/2021.
- *G.C. Buttazzo:* Hard Real-Time Computing Systems. Springer Verlag, ISBN 978-1-4614-0676-1, *2011.*
- *Edward A. Lee and Sanjit A. Seshia:* Introduction to Embedded Systems, A Cyber-Physical Systems Approach, Second Edition, MIT Press, ISBN 978-0-262-53381-2*, 2017.*

Sources: The slides contain ideas and material of J. Rabaey, K. Keuzer, M. Wolf, P. Marwedel, P. Koopman, E. Lee, P. Dutta, S. Seshia, and from the above cited books.

# Organization Summary

- **Lectures** are held on Mondays from 14:15 to 16:00 in ETF C1 until further notice. Life streaming and slides are available via the web page of the lecture. In addition, you find audio and video recordings of most of the slides as well as recordings of this years and last years life streams on the web page of the lecture.
- **Exercises** take place on Wednesdays and Fridays from 16:15 to 17:00 via Zoom. On Wednesdays the lecture material is summarized, hints on how to approach the solution are given and a sample question is solved. On Fridays, the correct solutions are discussed.
- **Laboratories** take place on Wednesdays and Fridays from 16:15 to 18:00 (the latest). On Wednesdays the session starts with a short introduction via Zoom and then questions can be asked via Zoom. Fridays are reserved for questions via Zoom.

# Further Material via the Web Page

## Lecture Slides

All lecture slides are available for download as a bundle:

- Embedded Systems lecture slides [single page format] ⤓
- Embedded Systems lecture slides [4on1 page format] ⤓

## Lecture Recordings

### Life Recordings Autumn 2021

The life recordings of the lectures in Autumn Semester are available at the following link: Embedded Systems Life Recordings AS 2021.

### Life Recordings Autumn 2020

The life recordings of last years lecture are available at the following links:

1. Lecture 1: Chapters 1. Introduction and 2. Software Development
2. Lecture 2: Chapters 2. Software Development and 3. Hardware-Software Interface

### Audio and Videos of Selected Chapters

Some of the chapters are documented via carefully recoreded videos. They contain some of the slides as well as audio explanations.

- 1. Introduction
- 2. Software Development
- 3. Hardware Software Interface

## Exercises and Laboratory

| **Generic Documents** | |
|---|---|
| Embedded System Companion | Supplementary Material |
| Remote Installation Instructions | |
| **Documents for Lab 0** | |
| Handout | Source (code) |
| Slides and videos | Solution (code and handout) |
| **Documents for Lab 1** | |
| Handout | Source (code) |
| Slides and videos | Solution (code and handout) |
| **Documents for Lab 2** | |
| Handout | Source (code) |
| Slides and videos | Solution (code and handout) |
| **Documents for Lab 3** | |

# When and where?

## Schedule

| | When | Where |
|---|---|---|
| Lectures | Monday 14:15 - 16:00 | ETF C1 |
| Exercises | Wednesday 16:15 - 17:00<br>Friday 16:15 - 17:00 | Zoom<br>Zoom |
| Labs | Wednesday 16:15 - 18:00<br>Friday 16:15 - 18:00 | Zoom<br>Zoom |

## Timetable

| Date | Lecture | Exercice | Lab |
|---|---|---|---|
| 27.09.2021 | 1. Introduction<br>2. Software Development | | |
| 29.09./01.10.2021 | | | 0. Prelab [MM] |
| 04.10.2021 | 3. Hardware-Software Interface | | |
| 06./08.10.2021 | | | 1. Bare Metal Program- |

# What will you learn?

- Theoretical foundations and principles of the analysis and design of embedded systems.
- Practical aspects of embedded system design, mainly software design.

The course has three components:

- *Lecture:* Communicate principles and practical aspects of embedded systems.
- *Exercise*: Use paper and pencil to deepen your understanding of analysis and design principles .
- *Laboratory (ES-Lab)*: Introduction into practical aspects of embedded systems design. Use of state-of-the-art hardware and design tools.

# Please read carfully!!

- https://www.tec.ee.ethz.ch/education/lectures/embedded-systems.html

Exercises and Laboratory

We urgently ask all students to do the laboratory on their own hardware. For this, we provide you with a virtual machine that has all the necessary software already pre-installed. You can find the installation instructions on GitLab. We have tested this setup on PCs and Laptops with an USB port that run Windows 10, macOS Catalina, as well as Linux Mint and Linux Ubuntu 18.04 and 20.04; in general, all platforms which can run VirtualBox should work. In exceptional circumstances where this is not possible, students are allowed to use the computers in ETZ D61.1 or ETZ D96.1 during the regular laboratory hours (Wednesday or Friday 16.15 – 18.00). In such a case, please send an email with your name and Legi number to the lecture coordinator. You will receive a time slot and room allocation that guarantees that the maximum occupation of the computer rooms is respected. You are not allowed to enter ETZ D61.1 or ETZ D96.1 during the laboratory hours if you do not have an allocated slot.

# What you got already…


# Be careful and please do not …

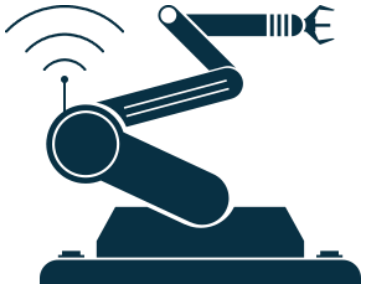
# You have to return the board at the end!

# Embedded Systems - Impact

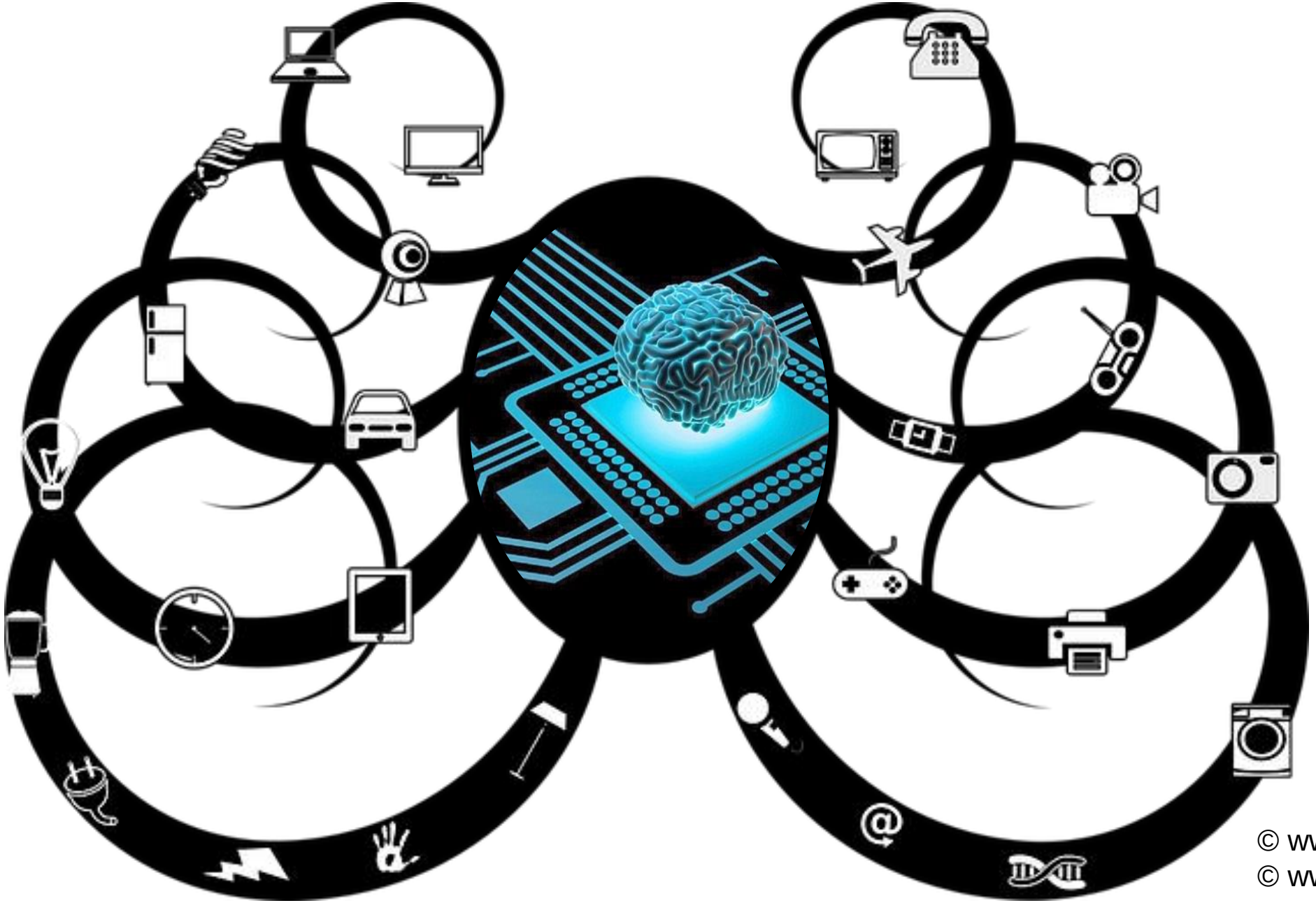
# Embedded Systems

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

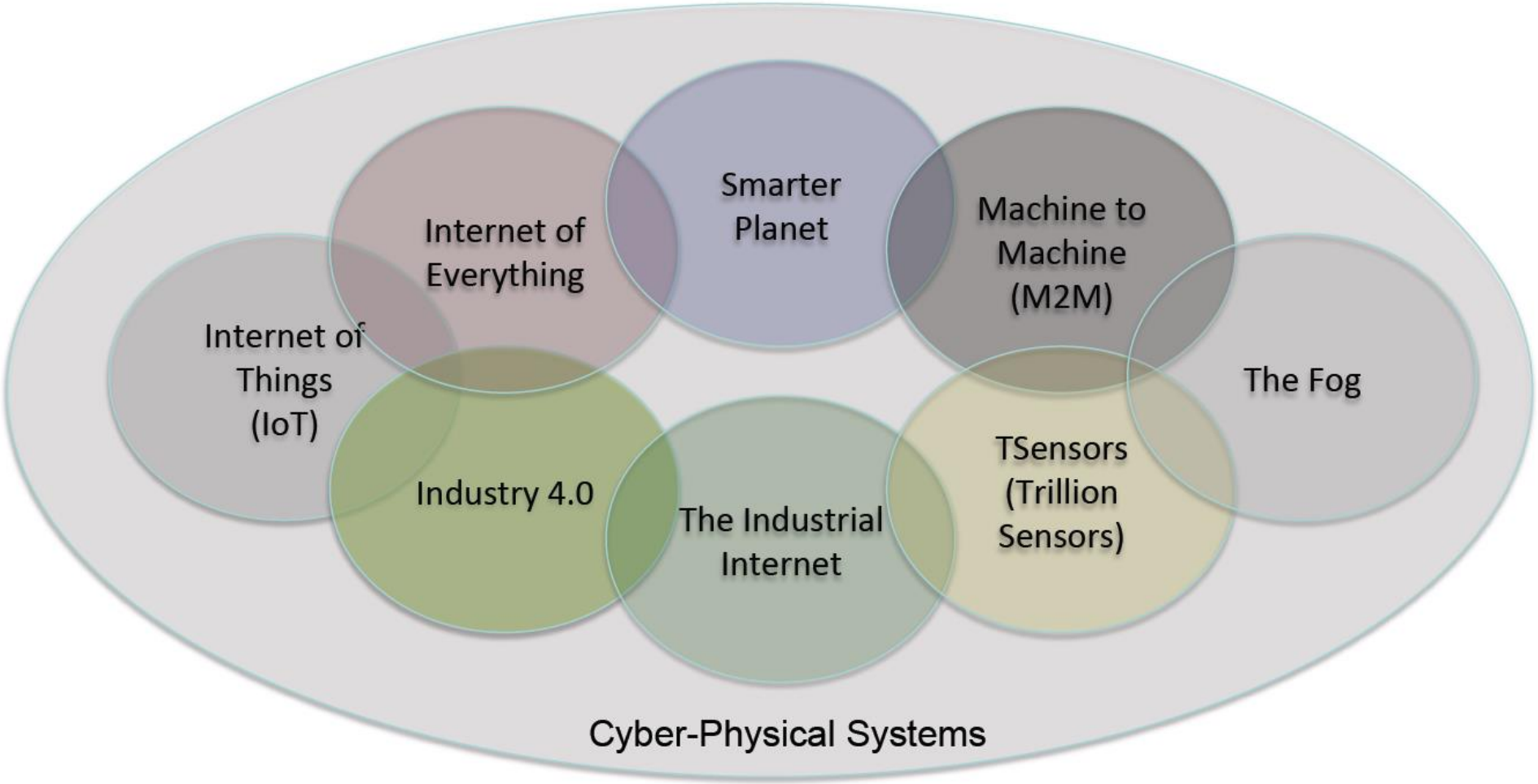
Examples:

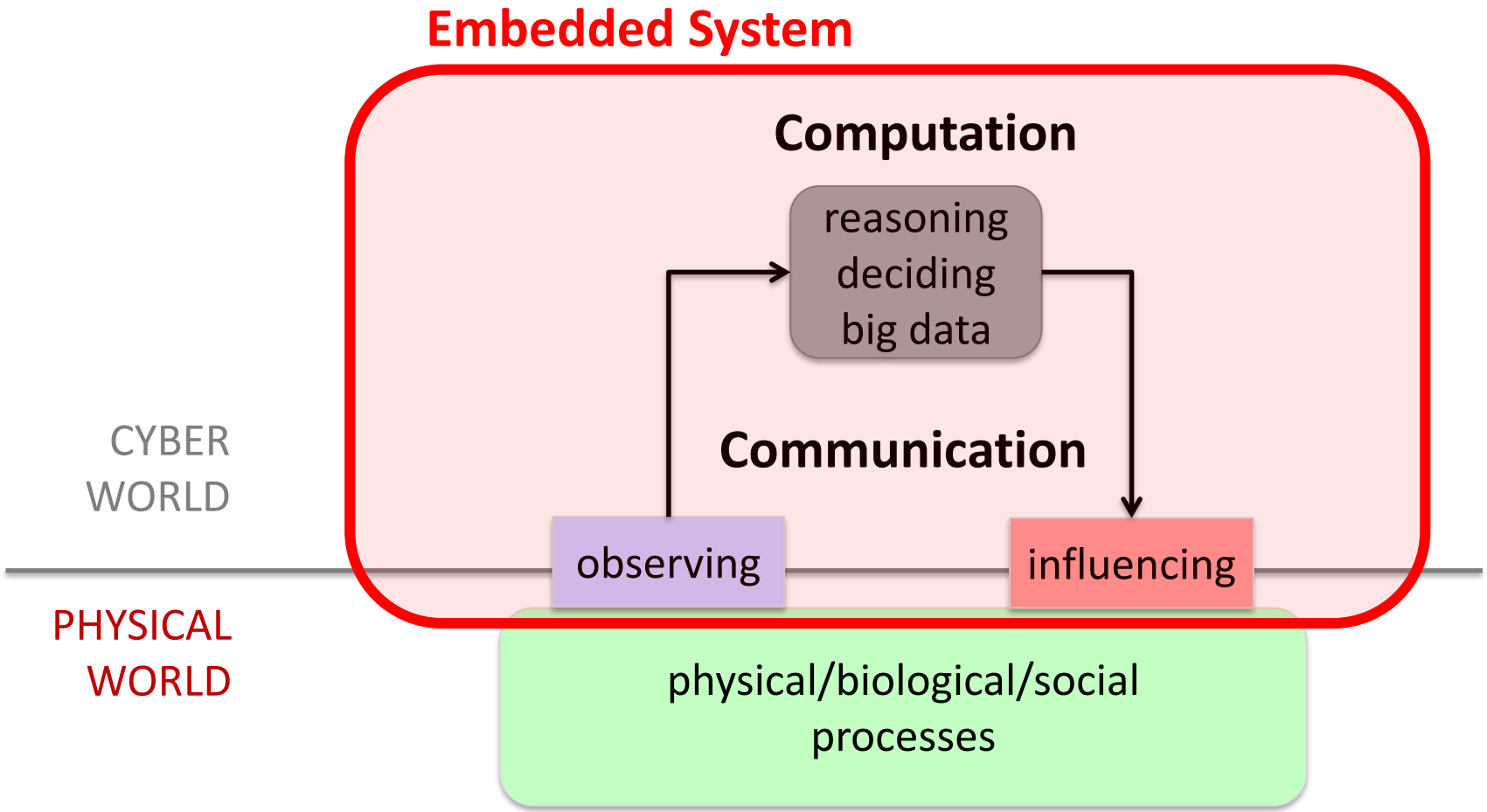

Often, the main reason for buying is not information processing



# Many Names – Similar Meanings



# Embedded System




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

# Reactivity & Timing

Embedded systems are often reactive:

- Reactive systems must react to stimuli from the system environment :

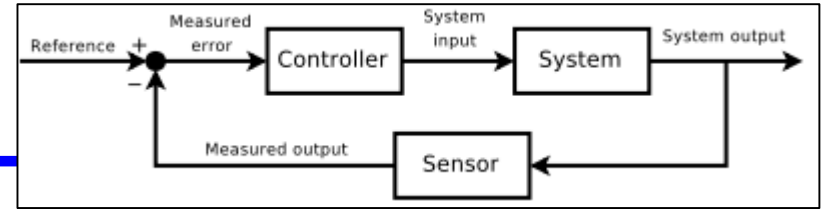
> *„A reactive system is one which is in continual interaction with is environment and executes at a pace determined by that environment“ [Bergé, 1995]*

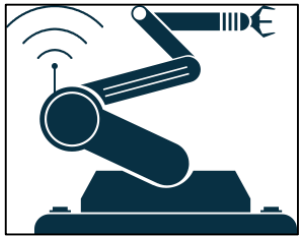
Embedded systems often must meet *real-time constraints:*

- For hard real-time systems, right answers arriving too late are wrong. All other time-constraints are called soft. A *guaranteed system response* has to be explained without statistical arguments.

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

# Predictability & Dependability



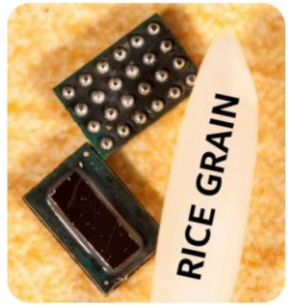
CPS = cyber-physical system

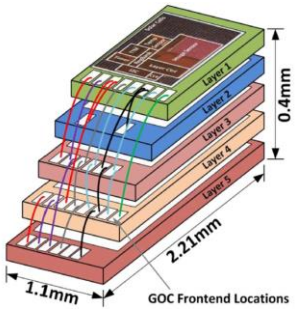
"It is essential to *predict* how a CPS is going to behave under any circumstances […] *before* it is deployed."[Maj14]

"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.

# Efficiency & Specialization




- Embedded systems must be *efficient:*
  - *Energy* efficient
  - *Code-size* and *data memory* efficient
  - *Run-time* efficient
  - *Weight* efficient
  - *Cost* efficient

Embedded Systems are often *specialized* towards a certain application or application domain:

- Knowledge about the expected behavior and the system environment at design time is exploited to *minimize resource usage* and to *maximize predictability and reliability*.

# Comparison

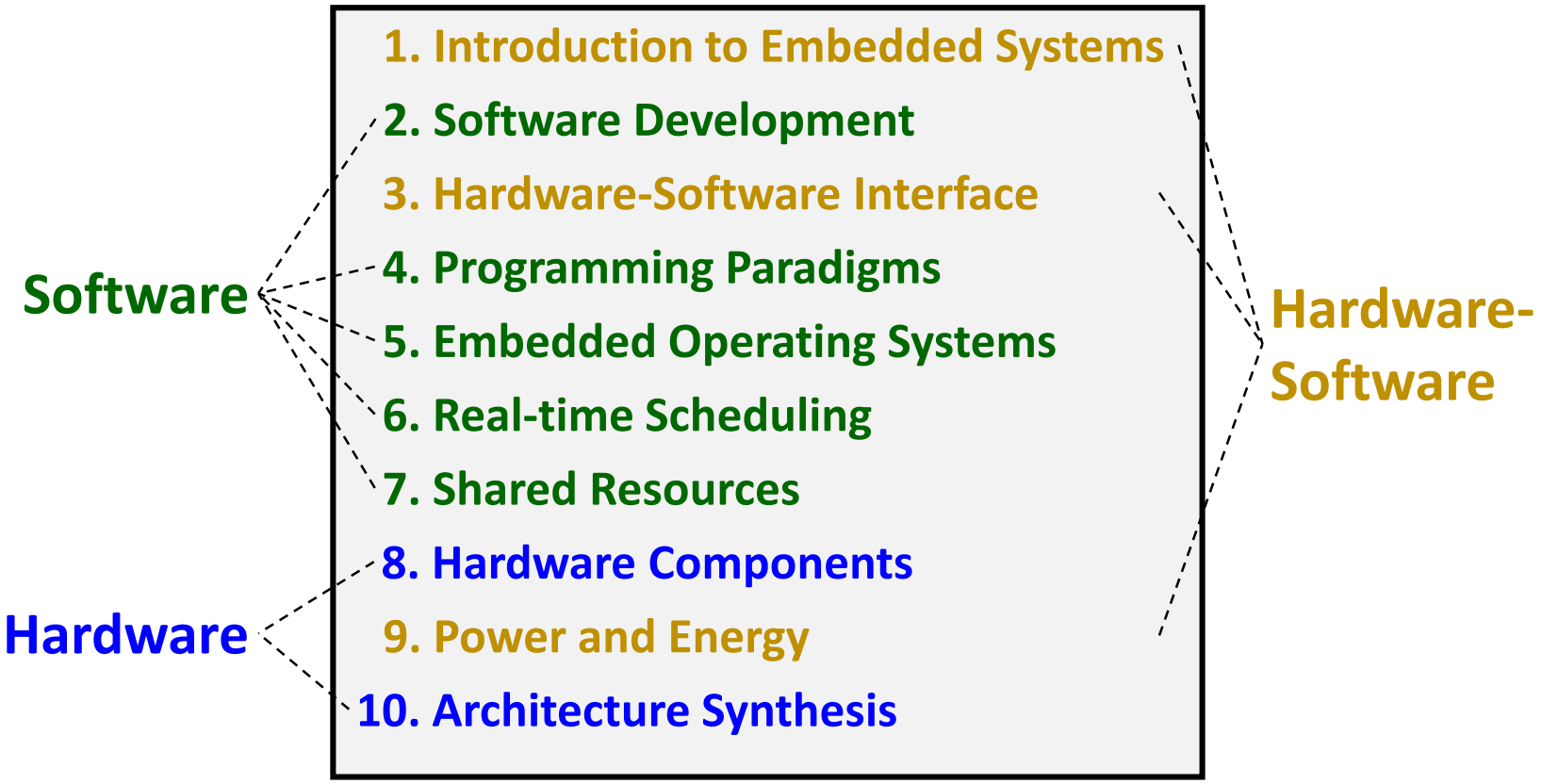
## *Embedded Systems:*

- Few applications that are known at design-time.
- Not programmable by end user.
- Fixed run-time requirements (additional computing power often not useful).
- Typical criteria:
  - cost
  - power consumption
  - size and weight
  - dependability
  - worst-case speed

## *General Purpose Computing*

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

# Lecture Overview



# Components and Requirements by Example

# Components and Requirements by Example

## - Hardware System Architecture -

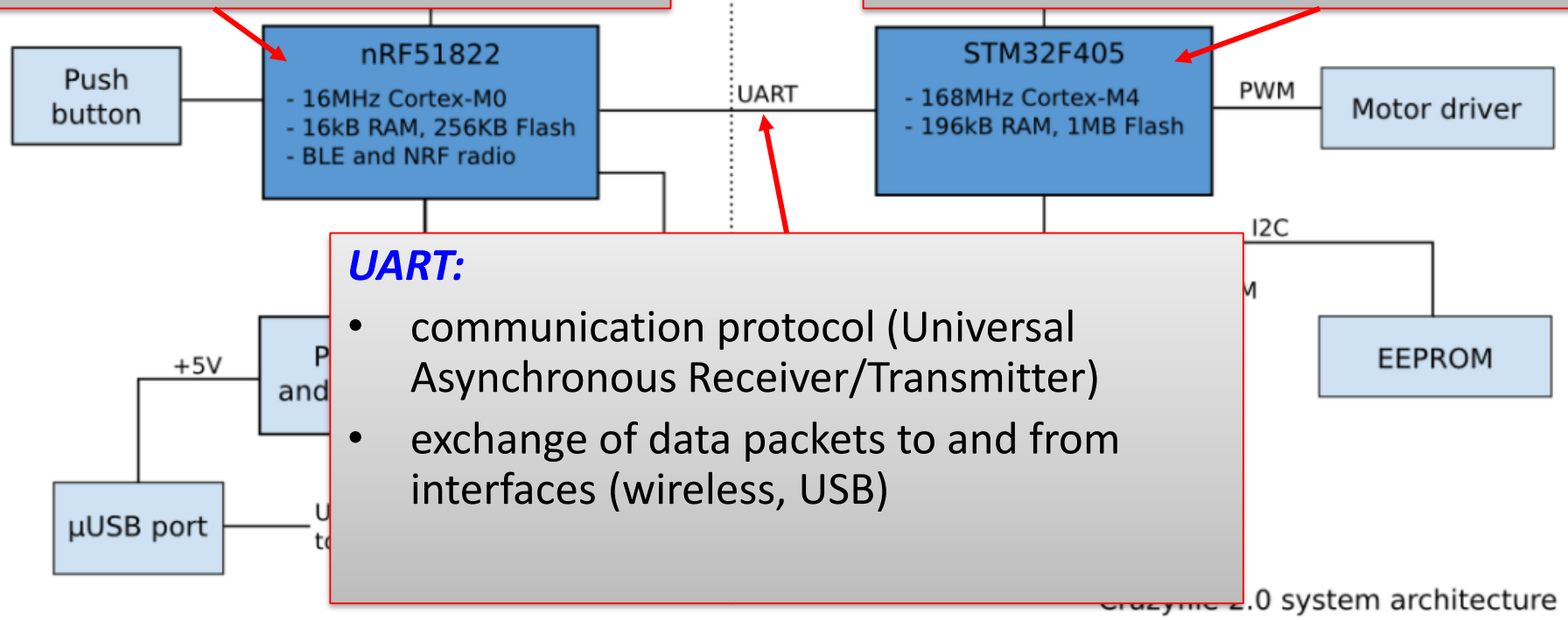
# High-Level Block Diagram View

***low power CPU***

- enabling power to the rest of the system
- battery charging and voltage measurement
- wireless radio (boot and operate)
- detect and check expansion boards

***higher performance CPU***

- sensor reading and motor control
- flight control
- telemetry (including the battery voltage)
- additional user development
- USB connection




***UART:***

- communication protocol (Universal Asynchronous Receiver/Transmitter)
- exchange of data packets to and from interfaces (wireless, USB)

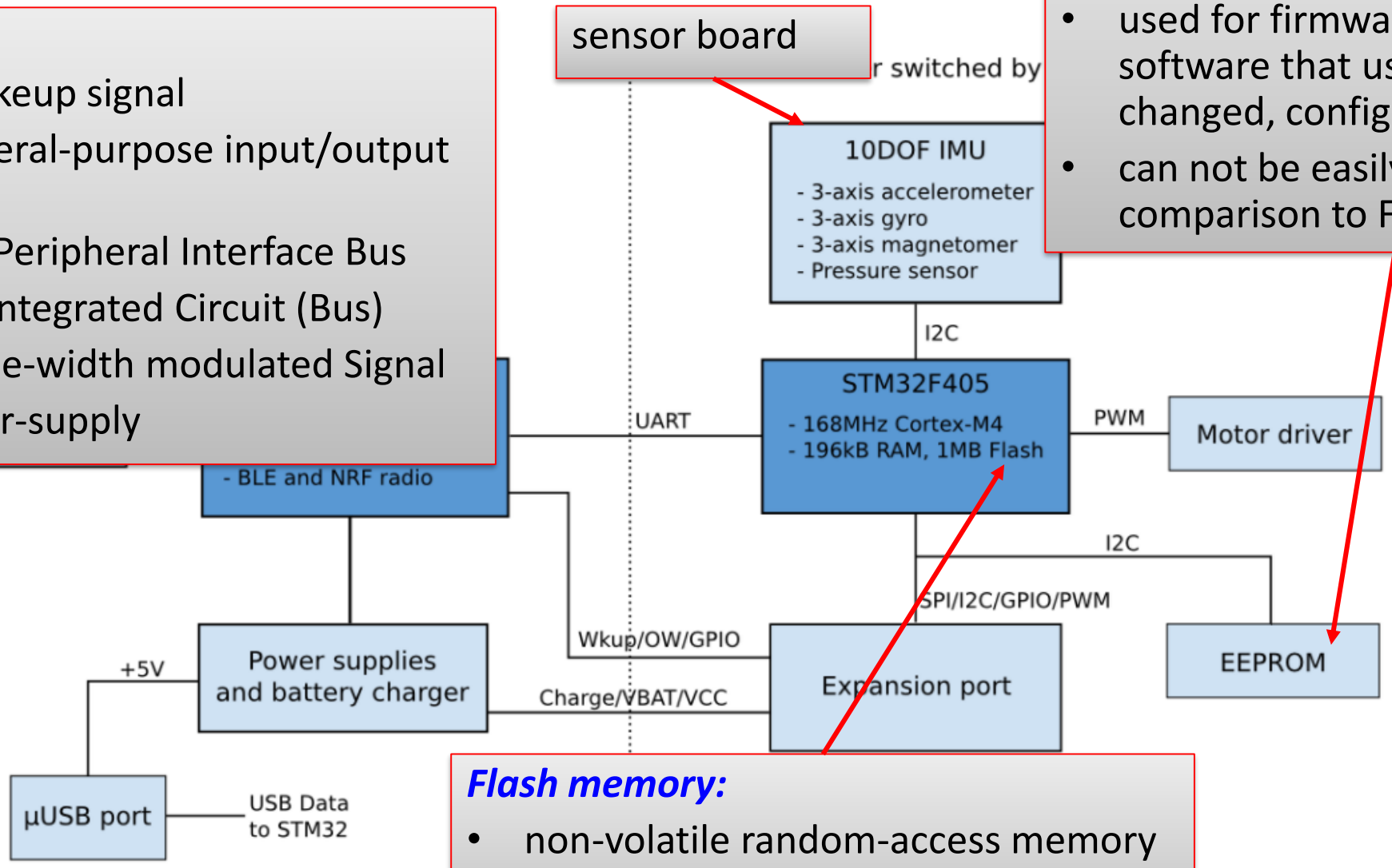
# High-Level Block Diagram View

***Acronyms:***

- Wkup: Wakeup signal
- GPIO: General-purpose input/output signal
- SPI: Serial Peripheral Interface Bus
- I2C: Inter-Integrated Circuit (Bus)
- PWM: Pulse-width modulated Signal
- VCC: power-supply

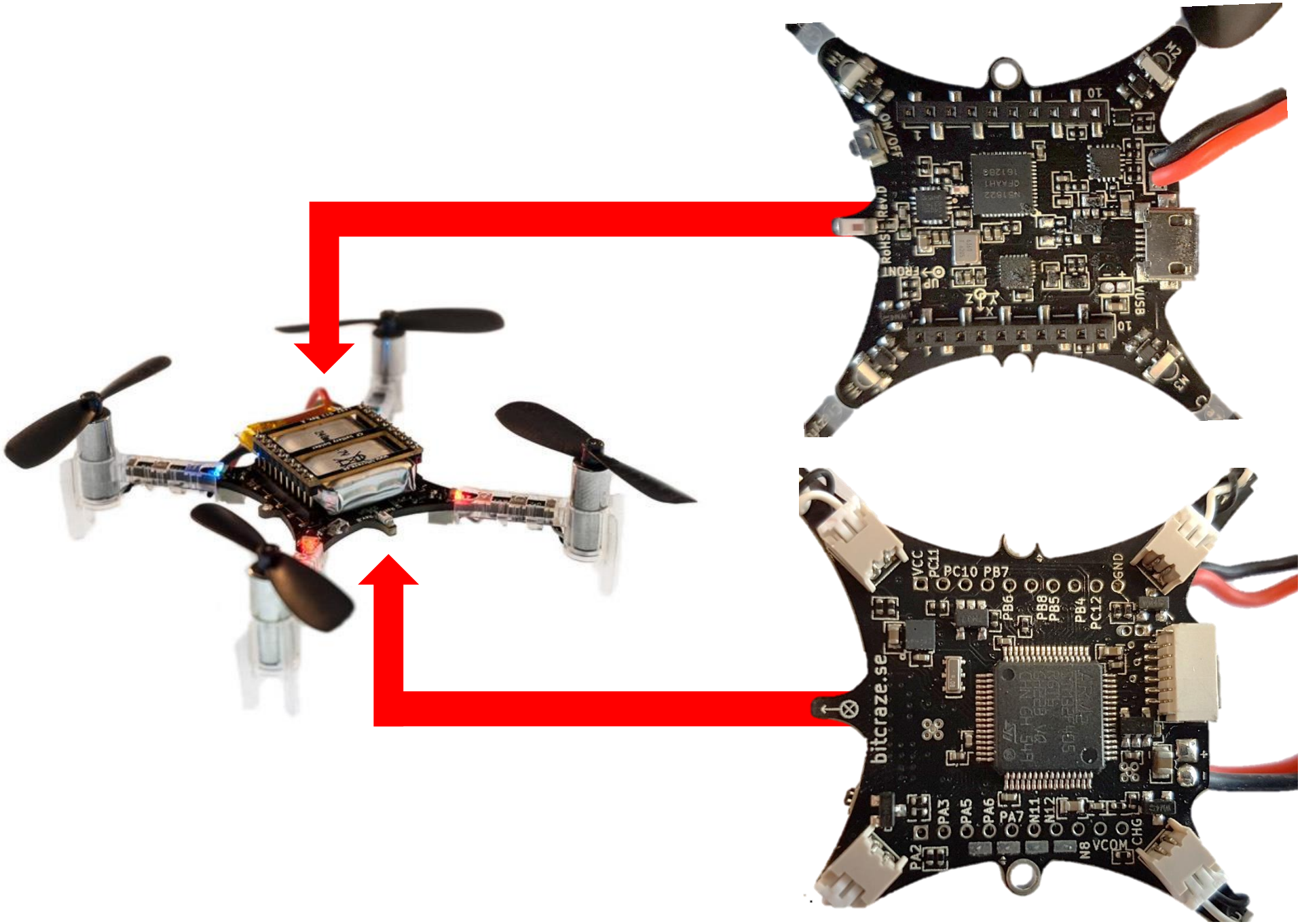
***EEPROM:***

- electrically erasable programmable read-only memory
- used for firmware (part of data and software that usually is not changed, configuration data)
- can not be easily overwritten in comparison to Flash

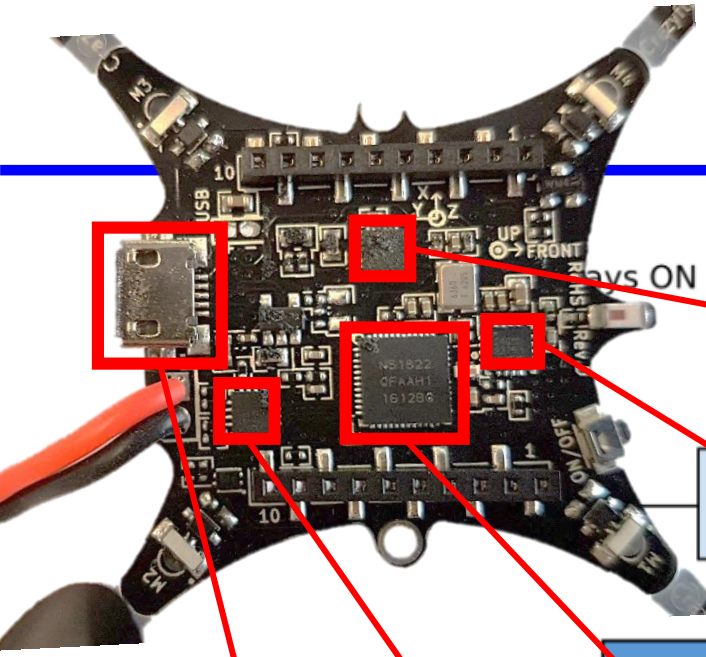



***Flash memory:***

- non-volatile random-access memory for program and data


ON/OFF
RoHS
REV.D
FRONT
USB
bitcraze.se
VCC
PC11
PC10
PB7
GND
PB6
PB8
PB5
PB4
PC12
PA2
PA3
PA5
PA6
PA7
CN11
CN12
N8
VCOM
CHG

# High-Level Physical View



ays ON power domain

Power switched by nRF51 (VCC)

RF power amplifier

10DOF IMU
- 3-axis accelerometer
- 3-axis gyro
- 3-axis magnetomer
- Pressure sensor

I2C

Push button

nRF51822
- 16MHz Cortex-M0
- 16kB RAM, 256KB Flash
- BLE and NRF radio

UART

STM32F405
- 168MHz Cortex-M4
- 196kB RAM, 1MB Flash

PWM

Motor driver

I2C

SPI/I2C/GPIO/PWM

Wkup/OW/GPIO

Power supplies and battery charger

+5V

Charge/VBAT/VCC

Expansion port

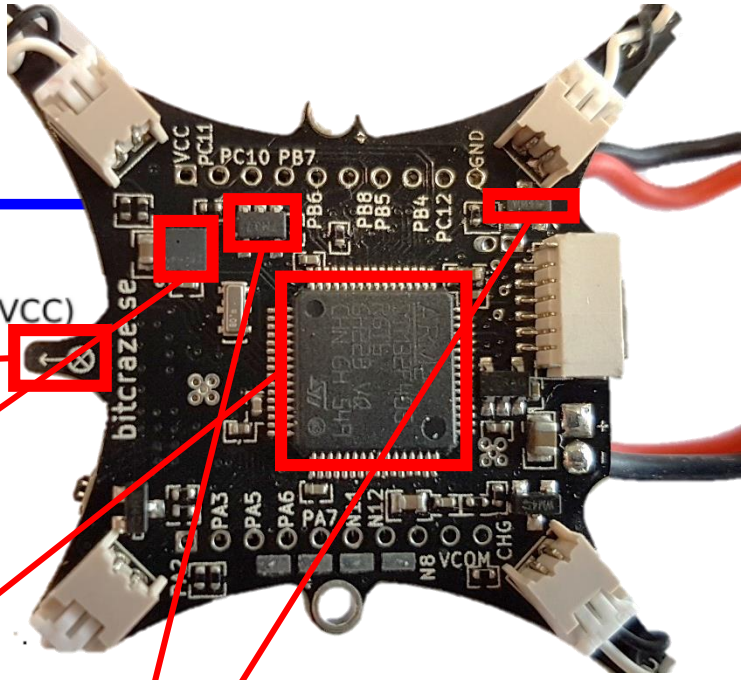
EEPROM

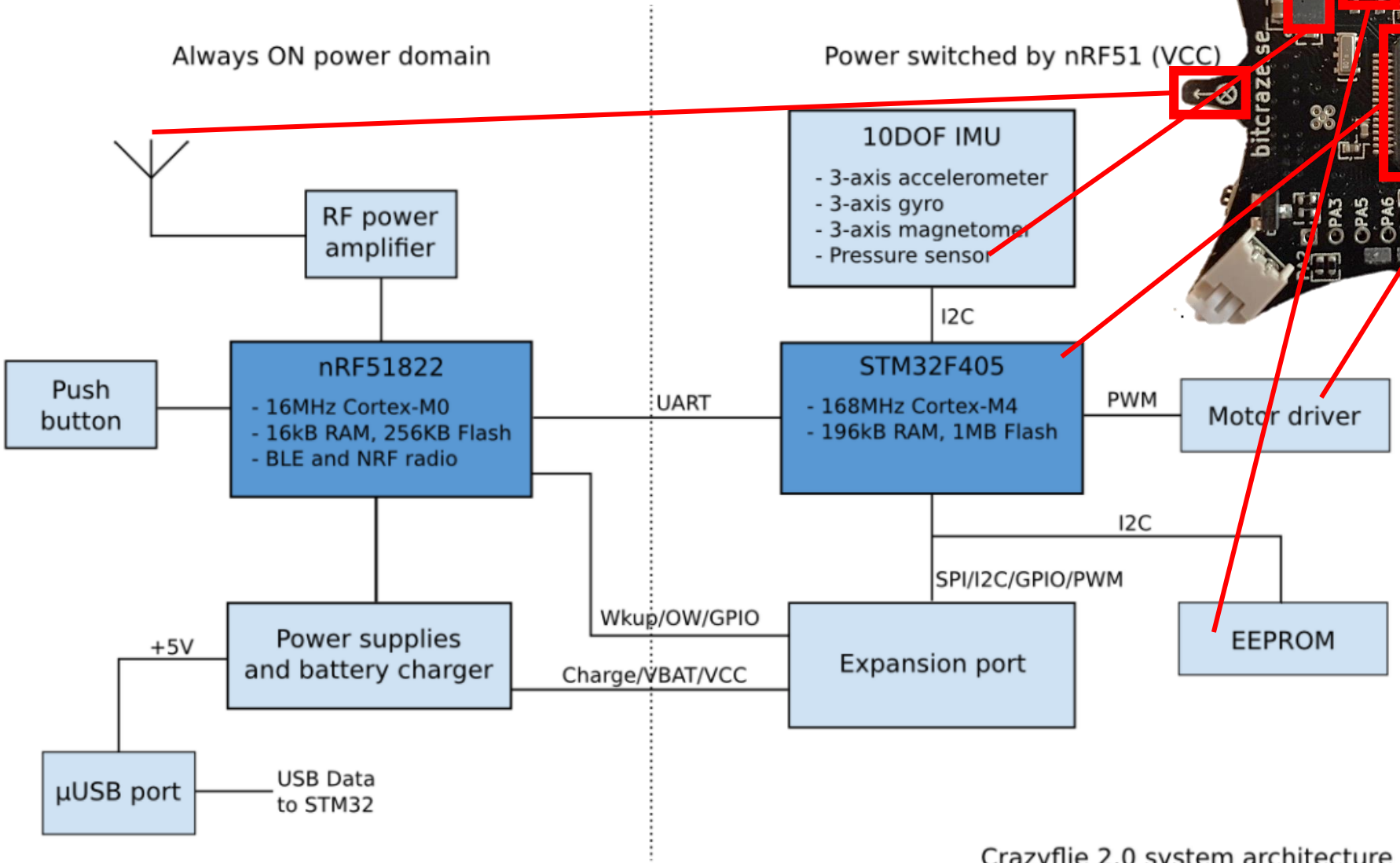
µUSB port

USB Data to STM32

Crazyflie 2.0 system architecture

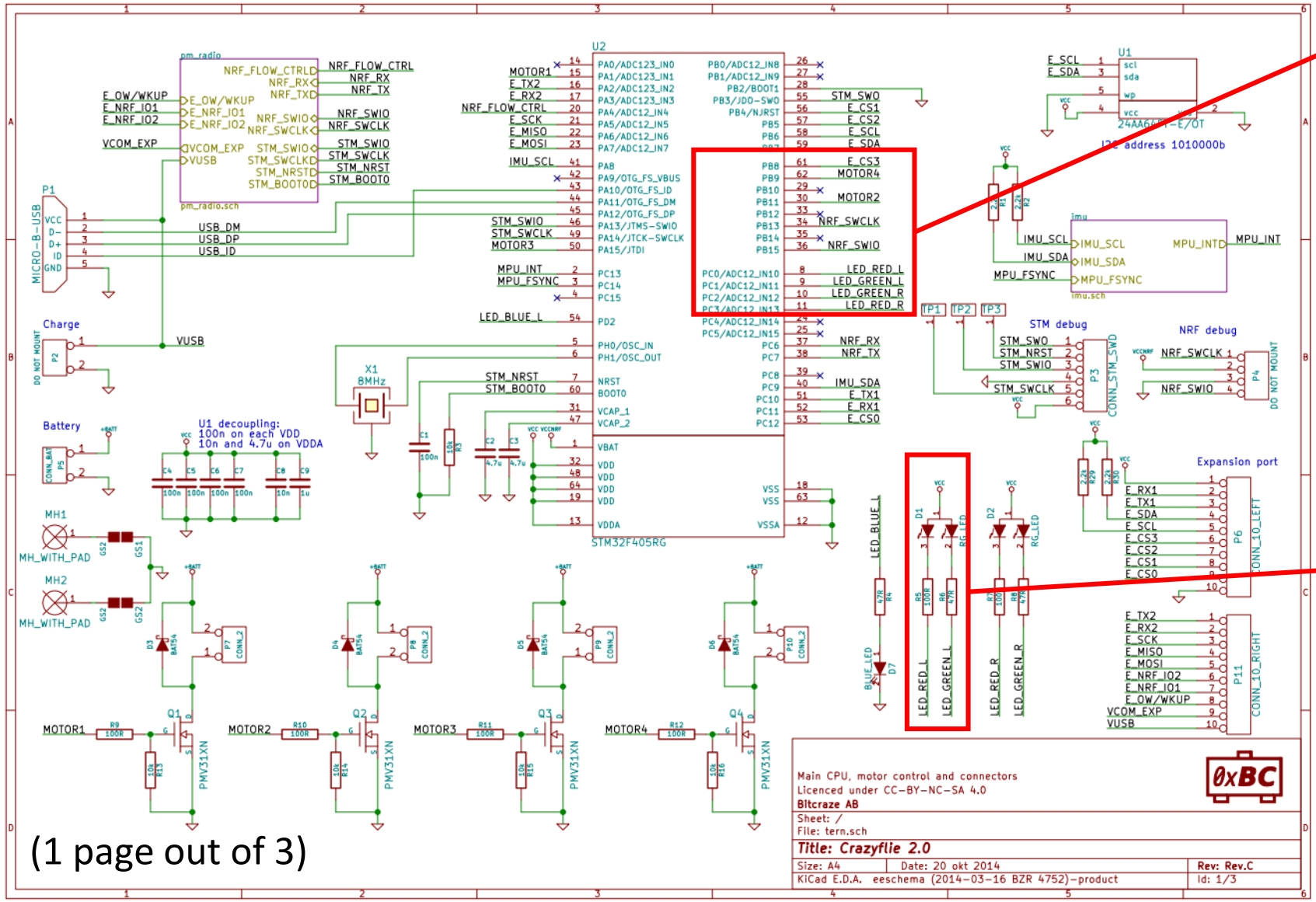
# High-Level Physical View



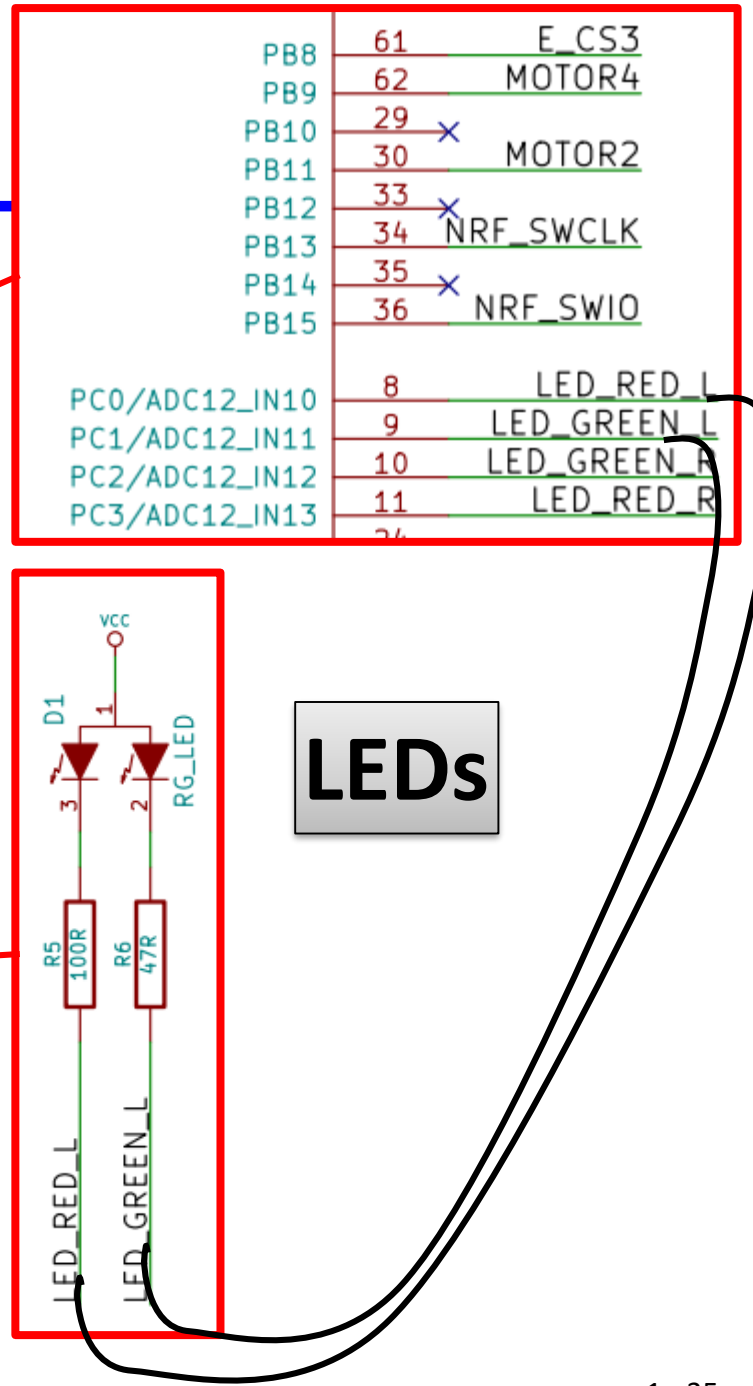




Crazyflie 2.0 system architecture

# Low-Level Schematic Diagram View

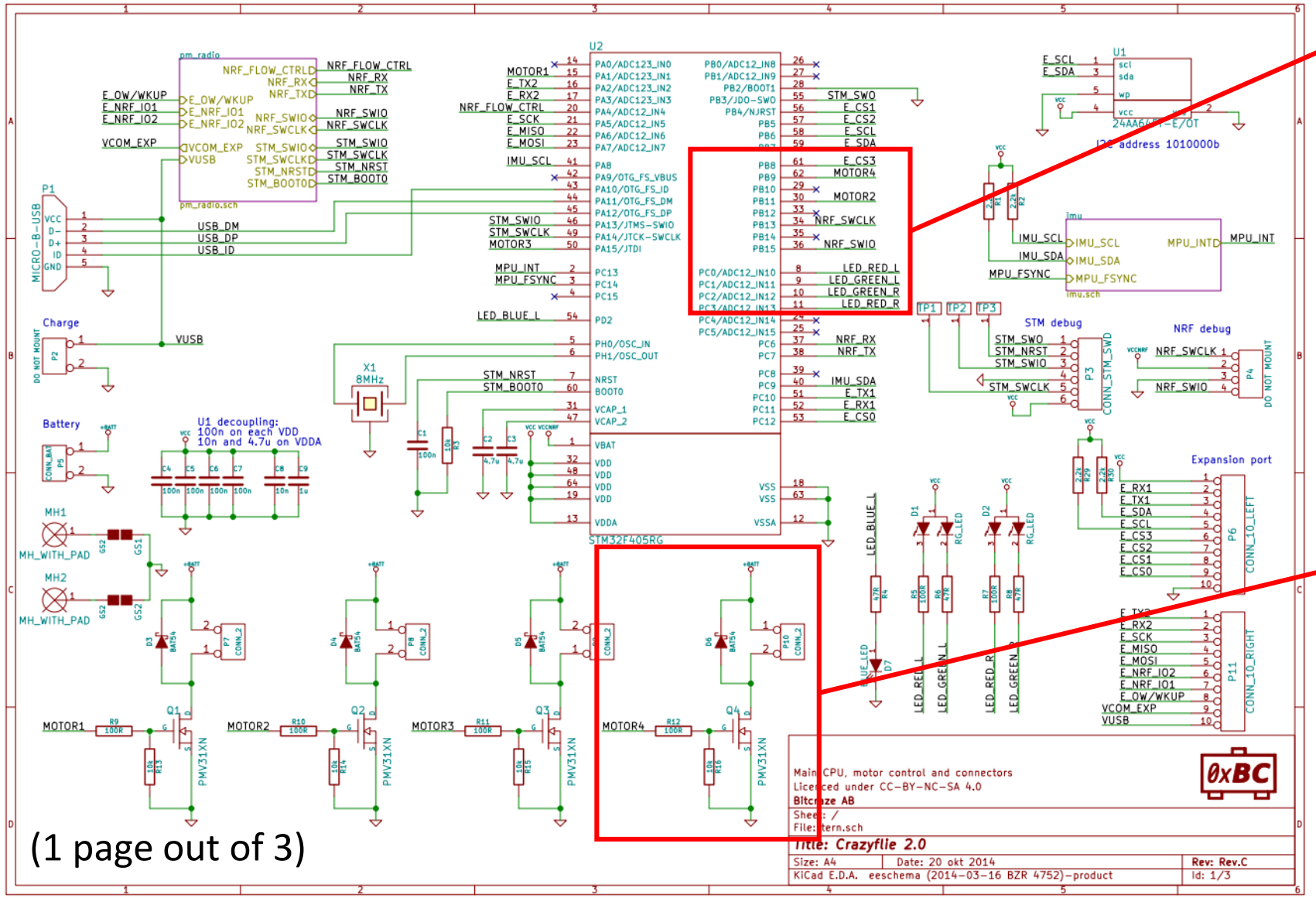



(1 page out of 3)

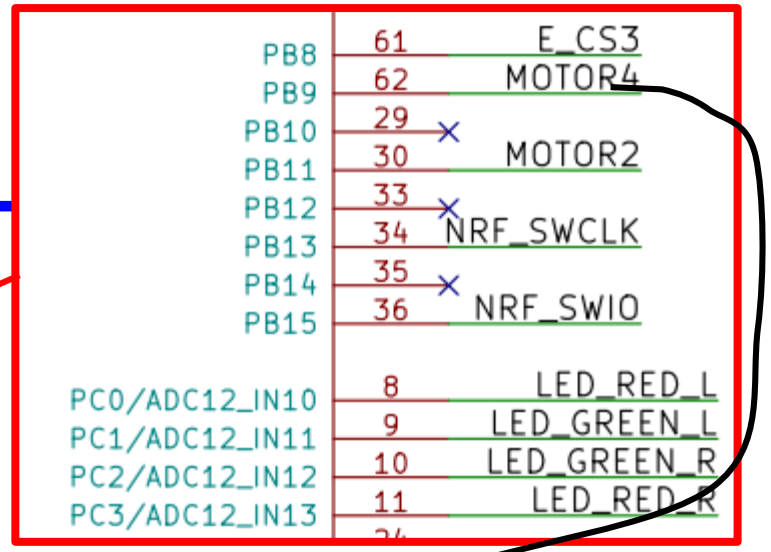


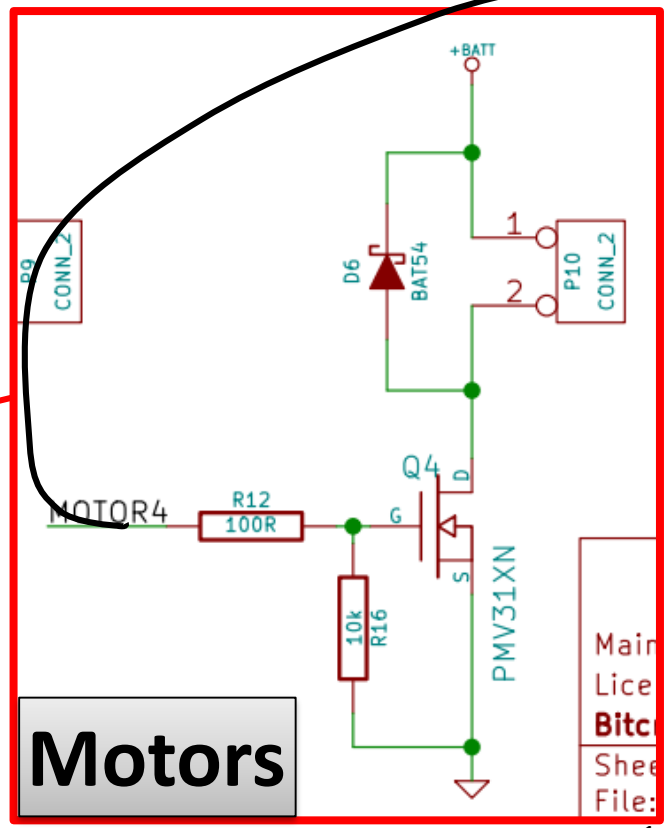

**LEDs**

# Low-Level Schematic Diagram View



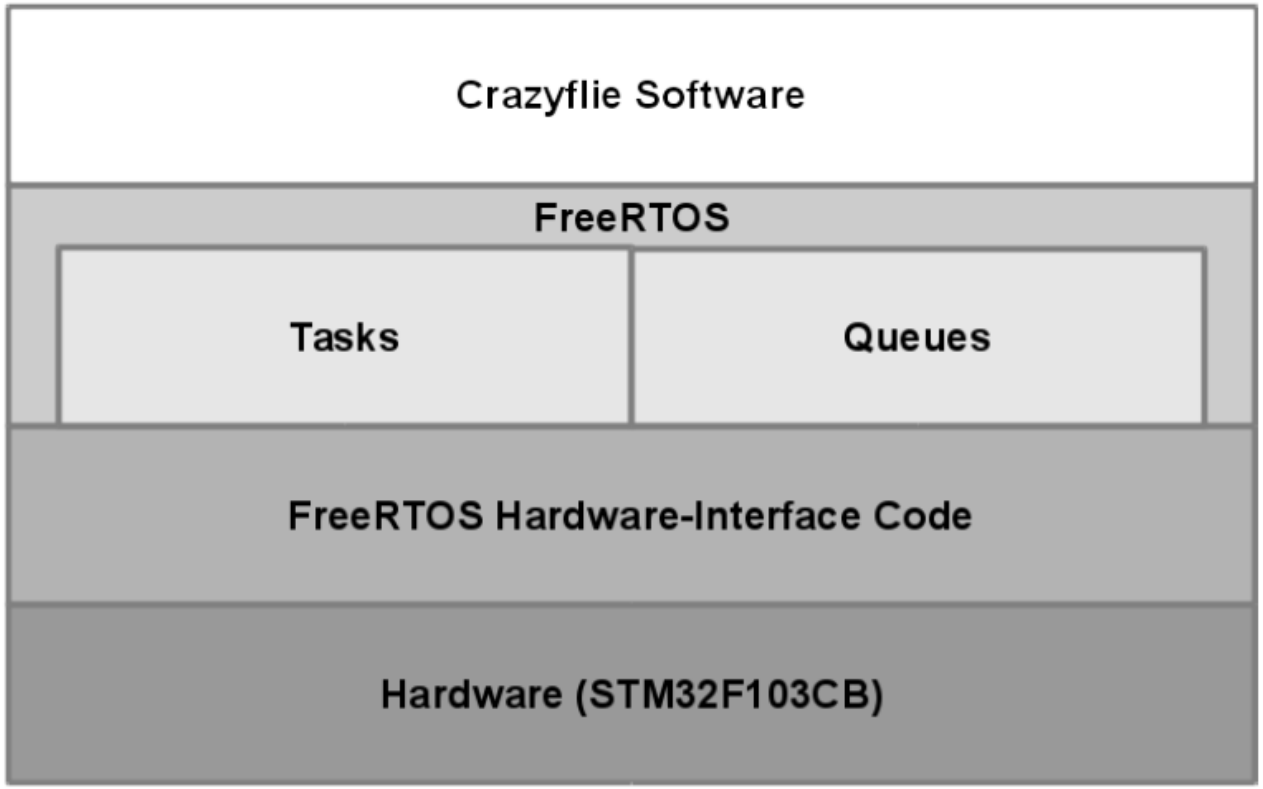

(1 page out of 3)



# High-Level Software View

- The software is built on top of a *real-time operating system* “FreeRTOS”.
- We will use the same operating system in the ES-Lab … .



# High-Level Software View

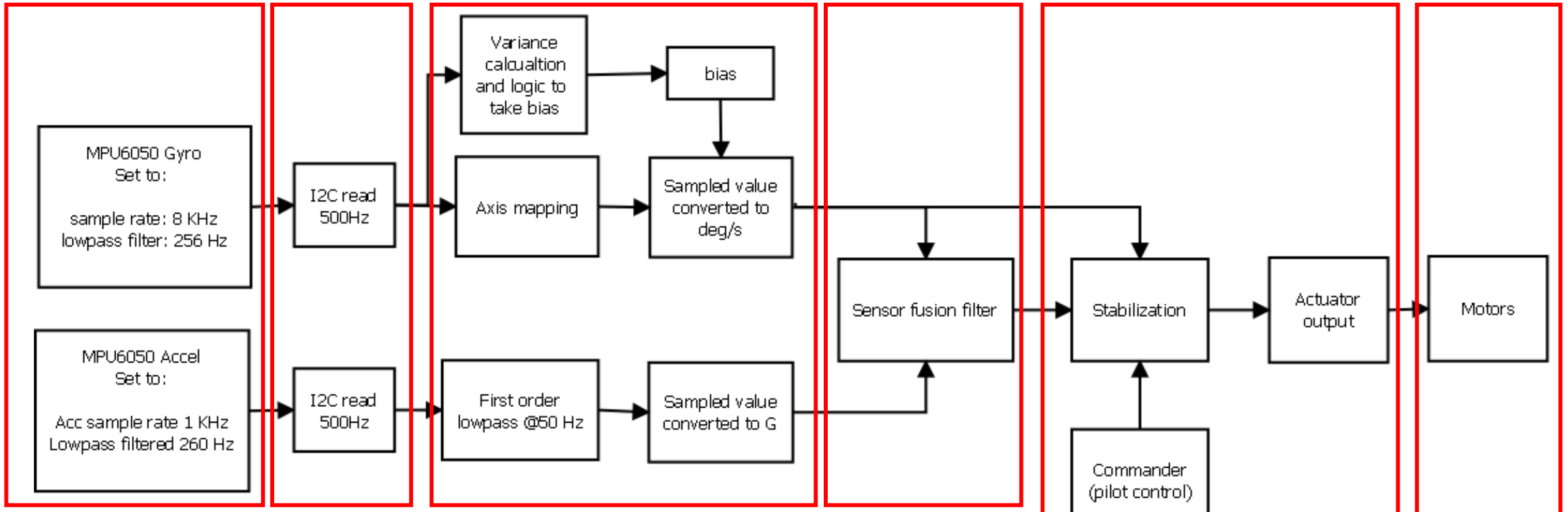
The *software architecture* supports

- *real-time tasks* for motor control (gathering sensor values and pilot commands, sensor fusion, automatic control, driving motors using PWM (pulse width modulation, … ) but also

- *non-real-time tasks* (maintenance and test, handling external events, pilot commands, … ).

# High-Level Software View

*Block diagram* of the stabilization system:

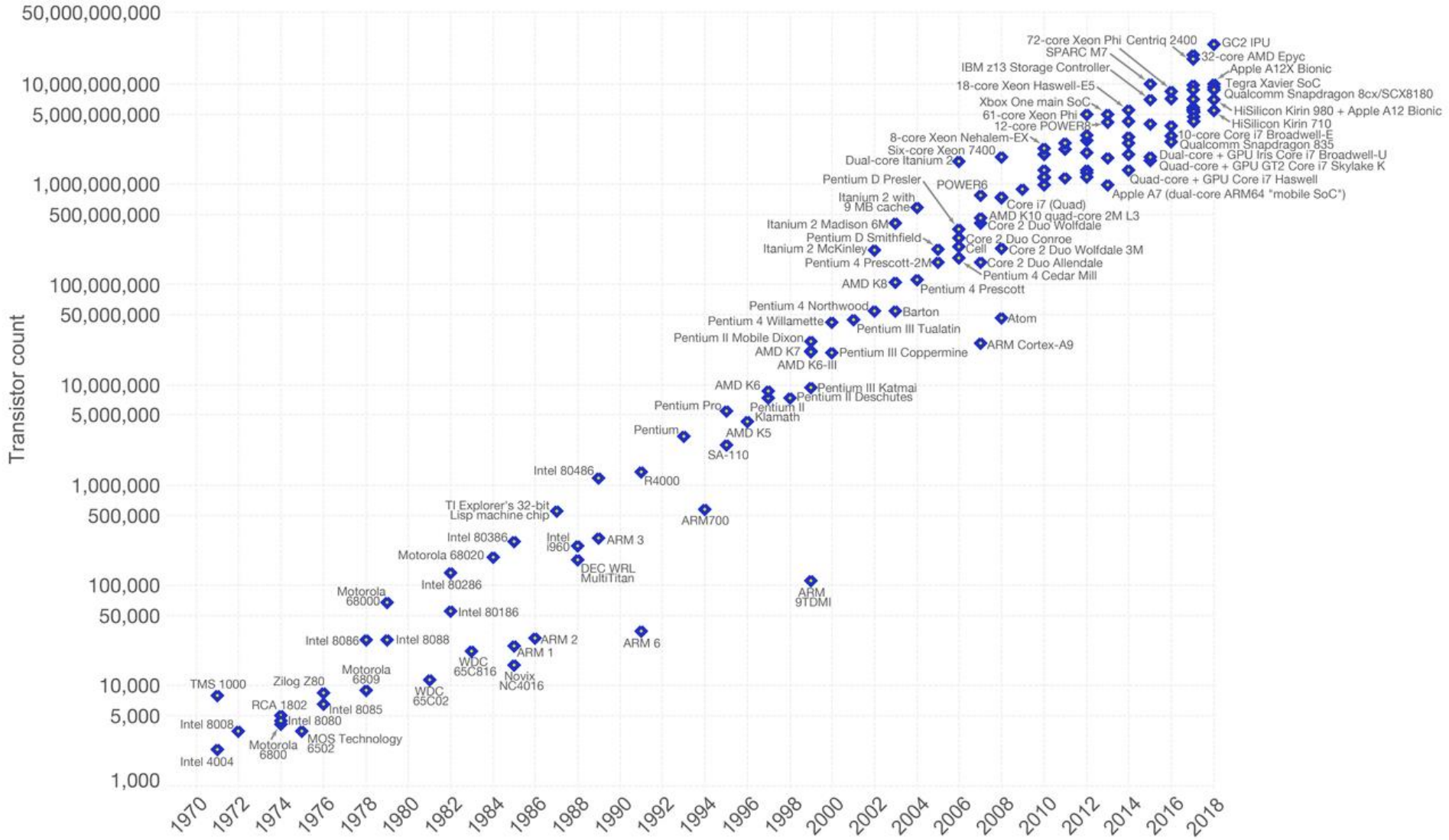


# Components and Requirements by Example

## - Processing Elements -

# What can you do to increase performance?

# From Computer Engineering

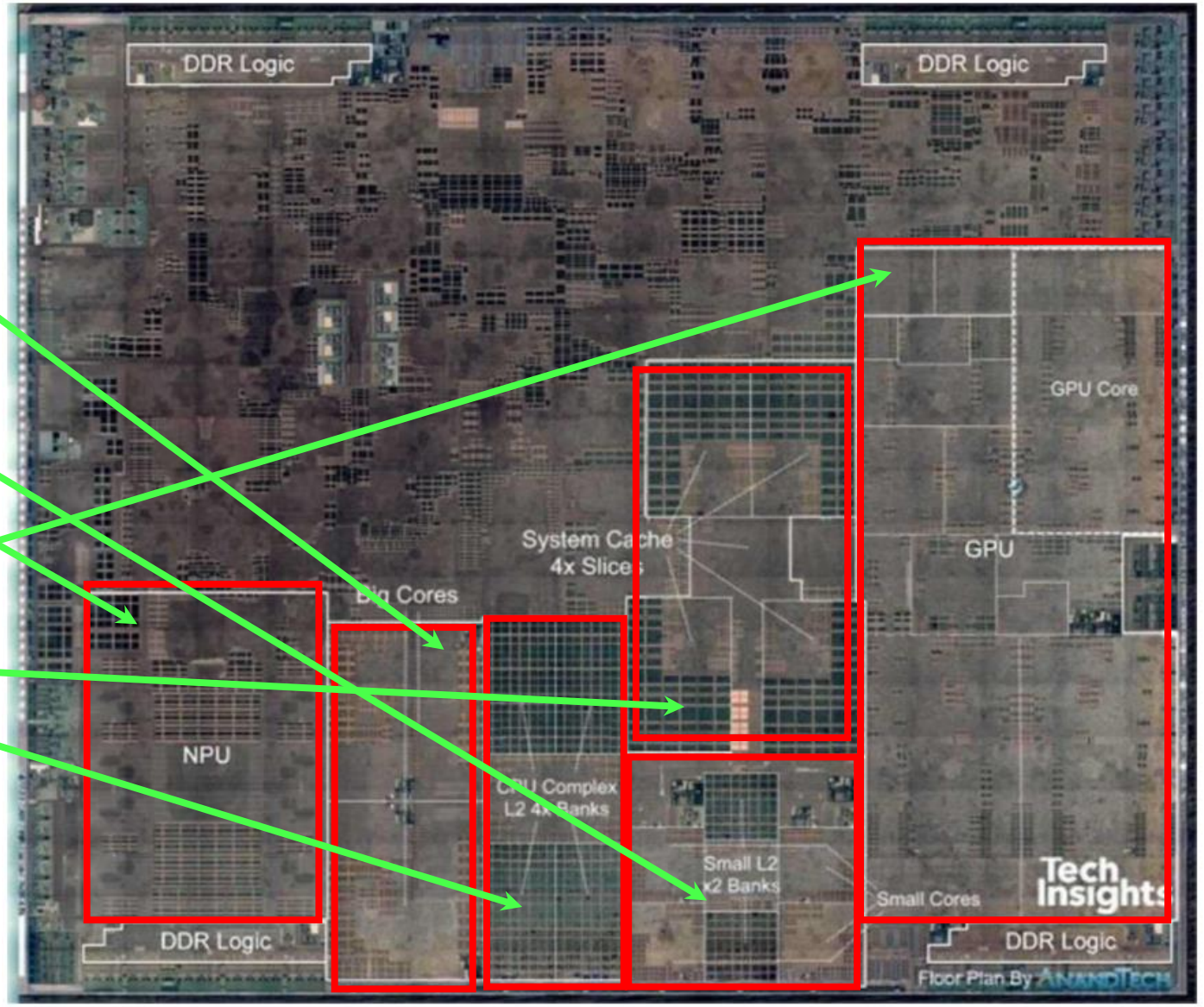



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

# From Computer Engineering

***iPhone Prozessor A12***

- 2 processor cores - high performance
- 4 processor cores - less performant
- Acceleration for Neural Networks
- Graphics processor
- Caches



# What can you do to decrease power consumption?

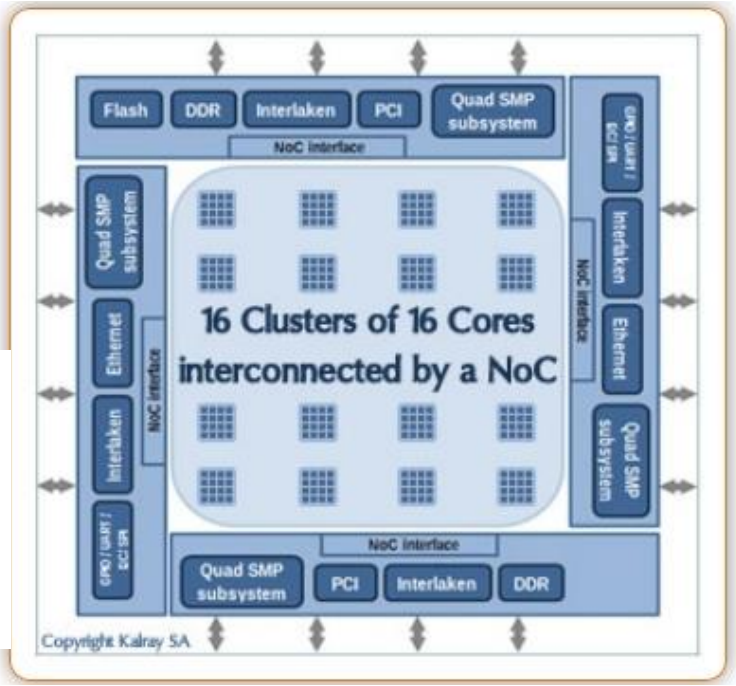
# Embedded Multicore Example

***Trends:***

- Specialize multicore processors towards real-time processing and low power consumption (parallelism can decrease energy consumption)
- Target domains:

IMAGE & AUDIO · SIGNAL PROCESSING · DATA SECURITY · SCIENTIFIC COMPUTING · CONTROL COMMAND


| 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 |

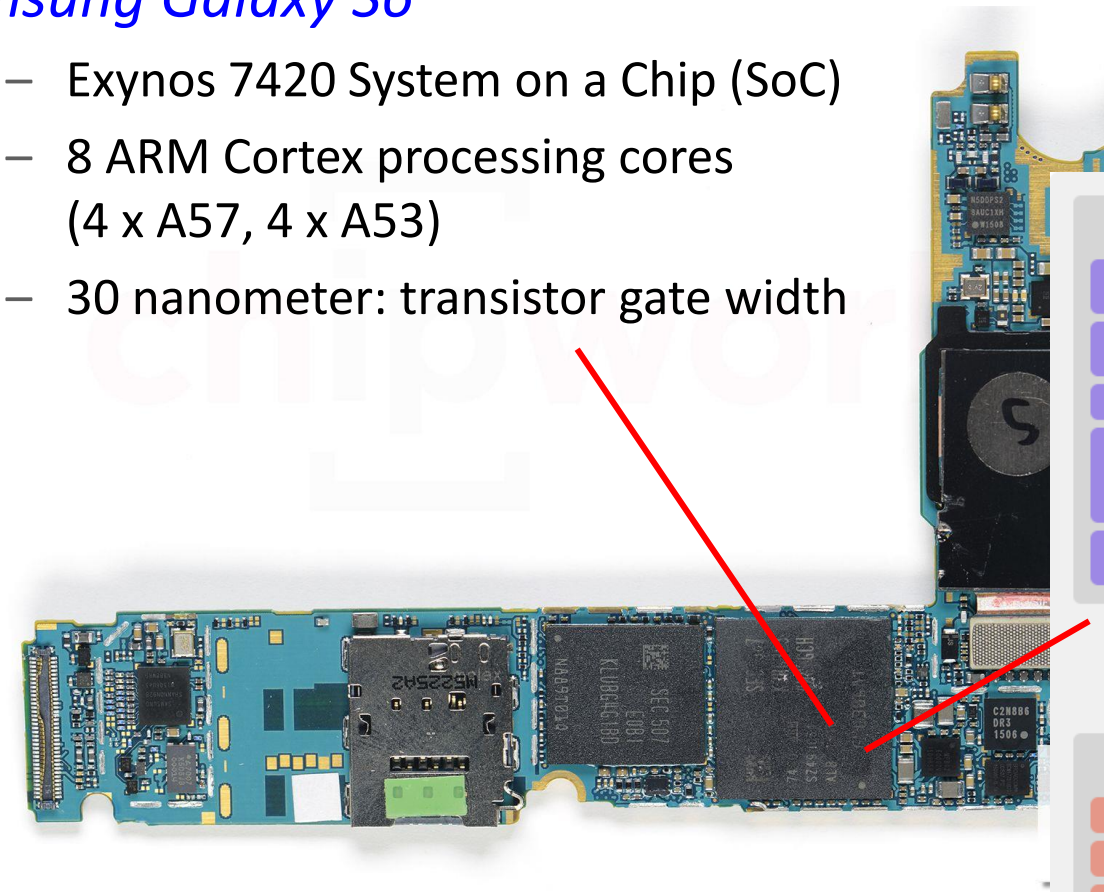


# Why does higher parallelism help in reducing power?

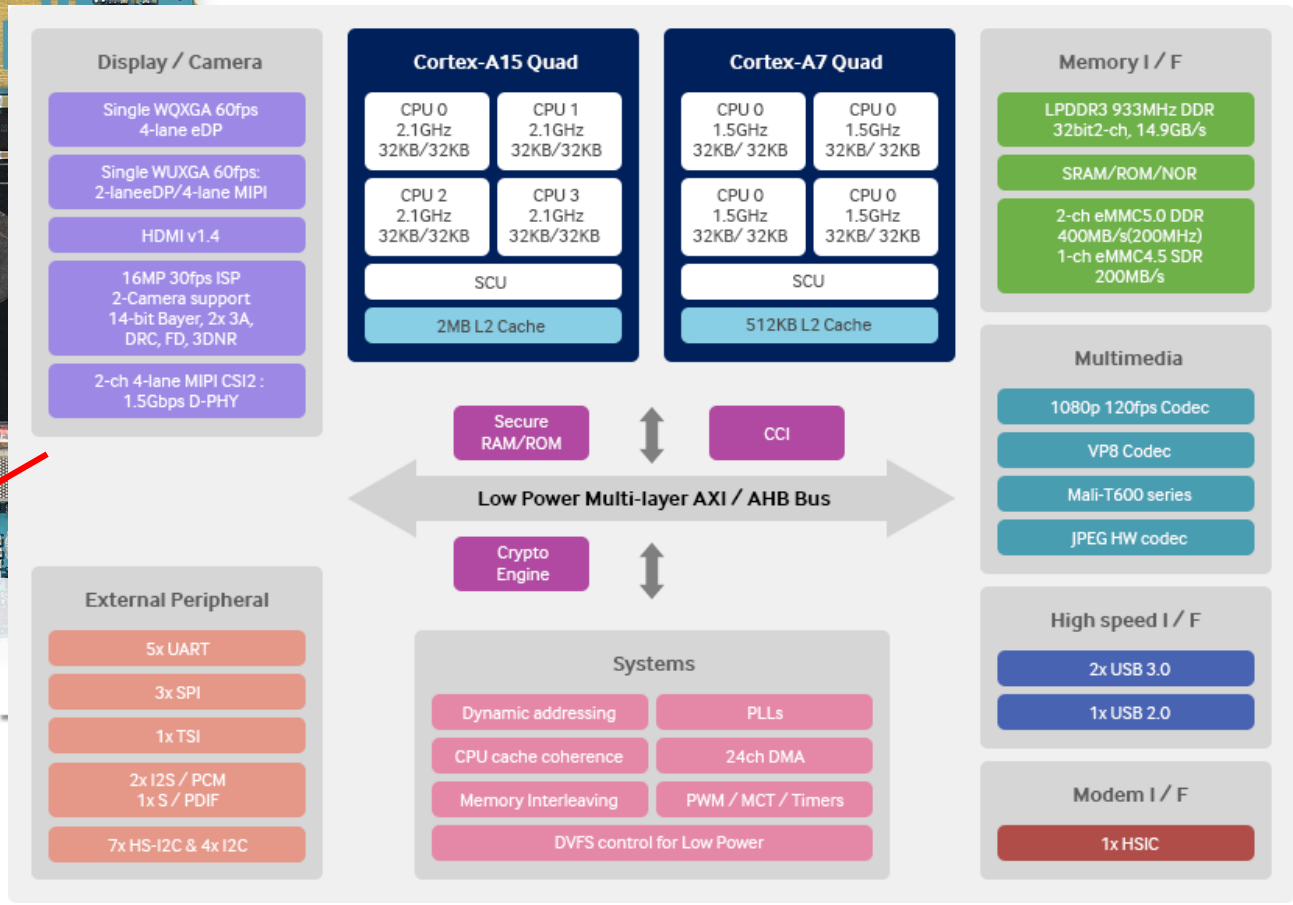
# System-on-Chip

## *Samsung Galaxy S6*

- Exynos 7420 System on a Chip (SoC)
- 8 ARM Cortex processing cores (4 x A57, 4 x A53)
- 30 nanometer: transistor gate width



Exynos 5422

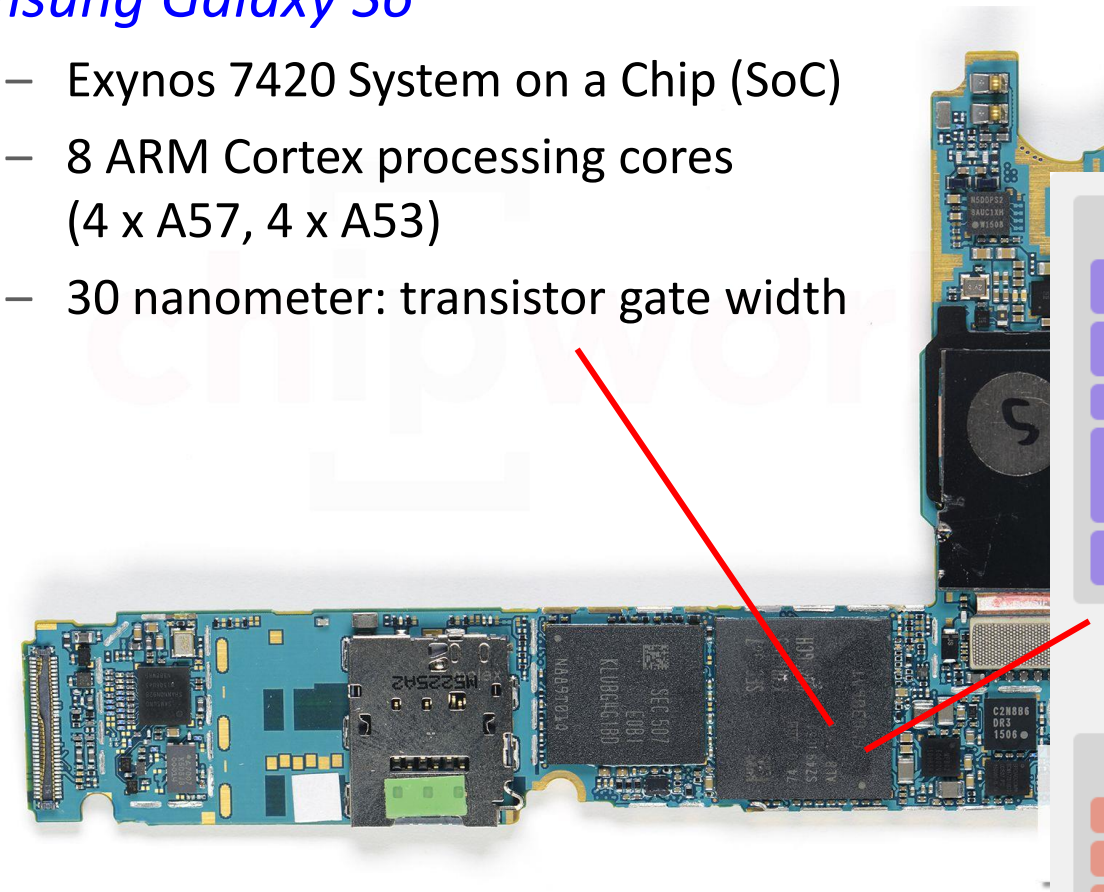


# How to manage extreme workload variability?

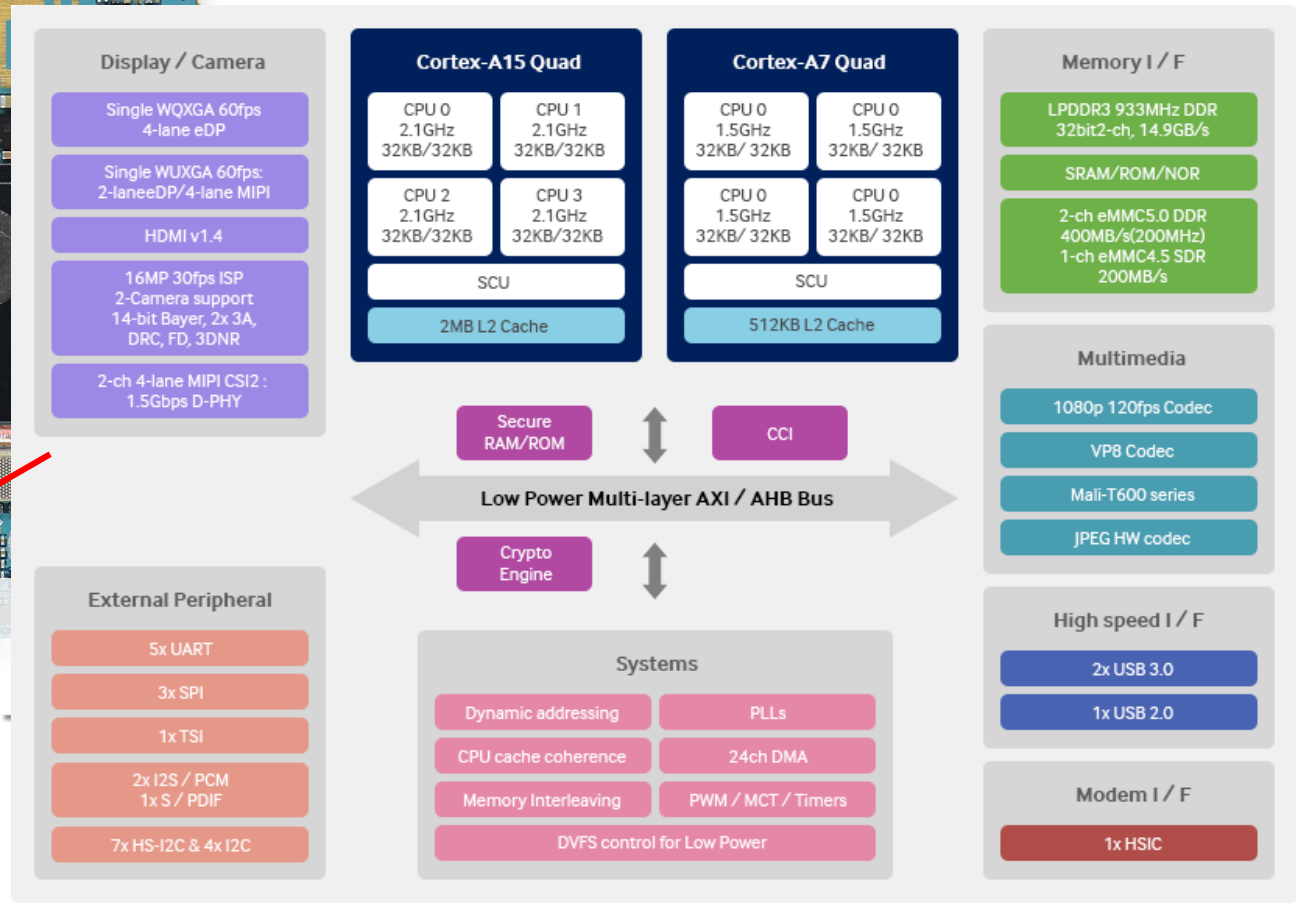
# System-on-Chip

## *Samsung Galaxy S6*

- Exynos 7420 System on a Chip (SoC)
- 8 ARM Cortex processing cores (4 x A57, 4 x A53)
- 30 nanometer: transistor gate width



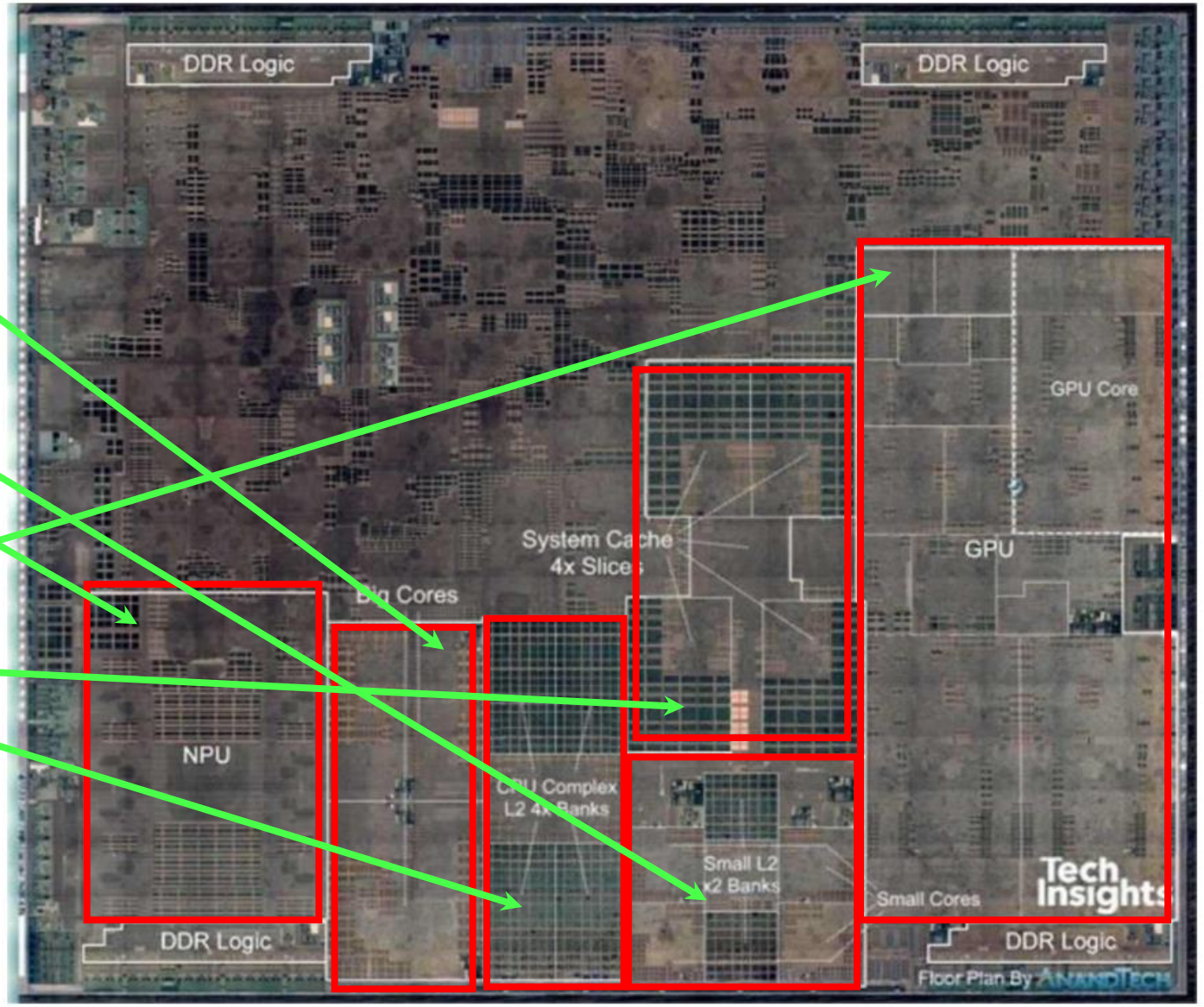
Exynos 5422



# From Computer Engineering

***iPhone Prozessor A12***

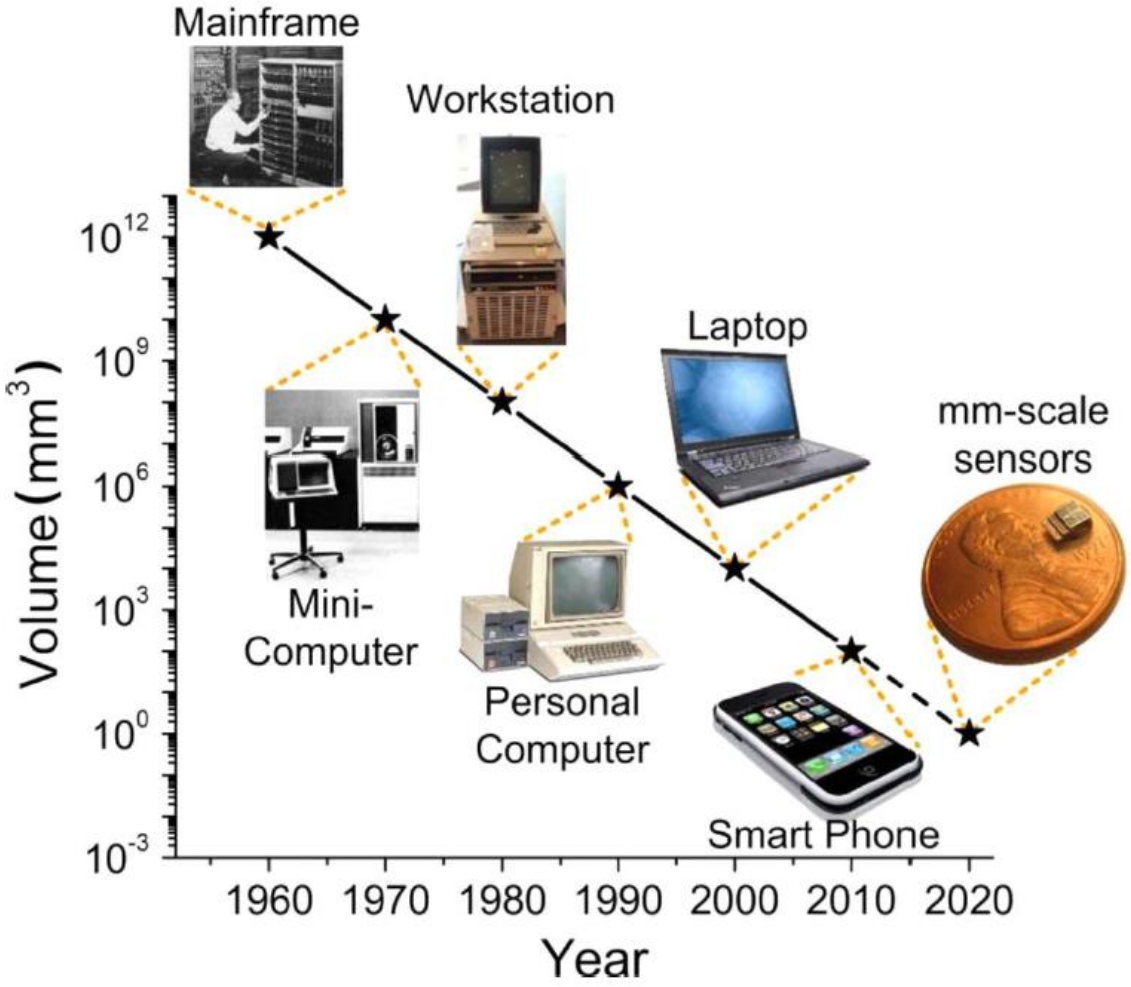
- 2 processor cores - high performance
- 4 processor cores - less performant
- Acceleration for Neural Networks
- Graphics processor
- Caches



# Components and Requirements by Example

## - Systems -

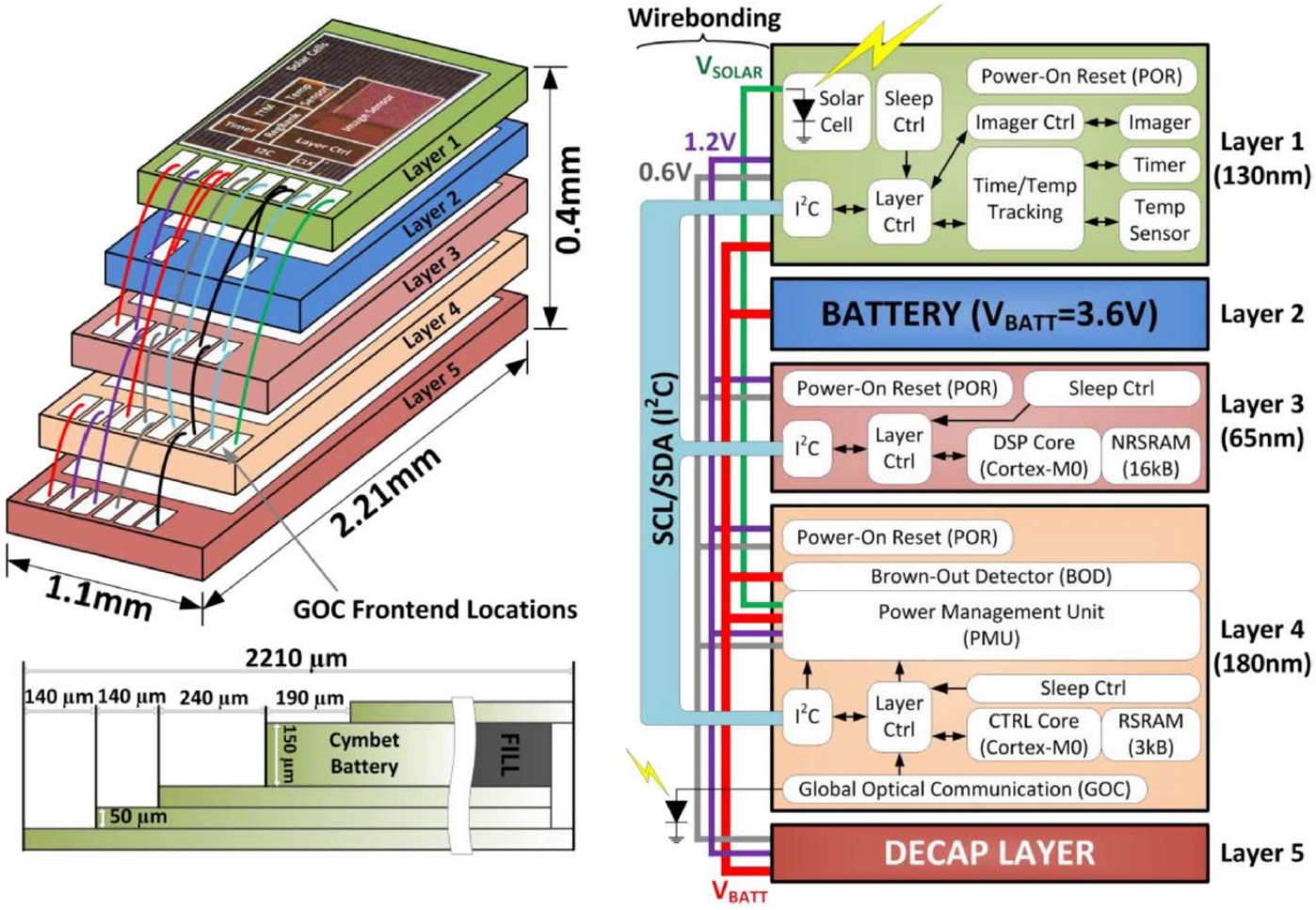
# Zero Power Systems and Sensors



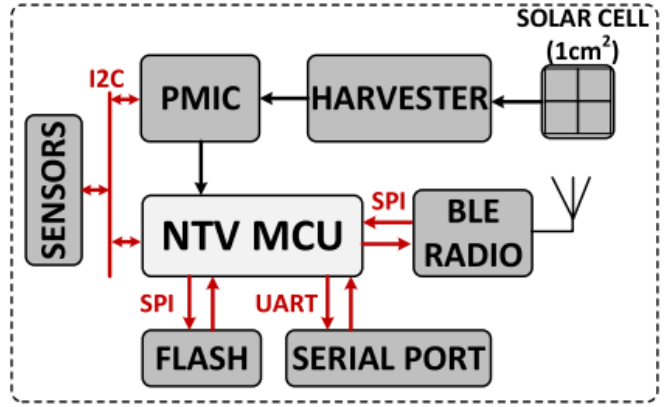

Streaming information to and from the physical world:

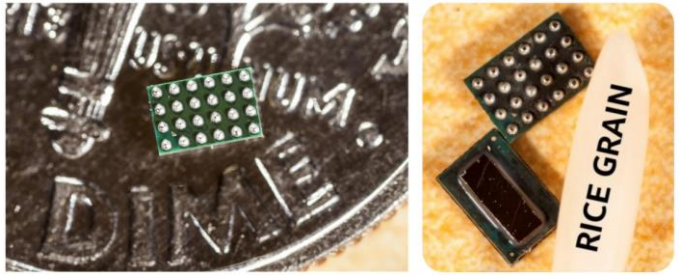
- "Smart Dust"
- Sensor Networks
- Cyber-Physical Systems
- Internet-of-Things (IoT)

# Zero Power Systems and Sensors




IEEE Journal of Solid-State Circuits, Jan 2013, 229-243.




IEEE Journal of Solid-State Circuits, April 2017, 961-971.

# Trends …

- *Embedded systems are communicating with each other,* with servers or with the cloud. Communication is increasingly wireless.

- *Higher degree of integration* on a single chip or integrated components:
  - Memory + processor + I/O-units + (wireless) communication.
  - Use of networks-on-chip for communication between units.
  - Use of homogeneous or heterogeneous multiprocessor systems on a chip (MPSoC).
  - Use of integrated microsystems that contain energy harvesting, energy storage, sensing, processing and communication ("zero power systems").
  - The complexity and amount of software is increasing.

- *Low power and energy constraints* (portable or unattended devices) are increasingly important, as well as temperature constraints (overheating).
- There is increasing interest in *energy harvesting* to achieve long term autonomous operation.

# Embedded Systems

## 2. Software Development

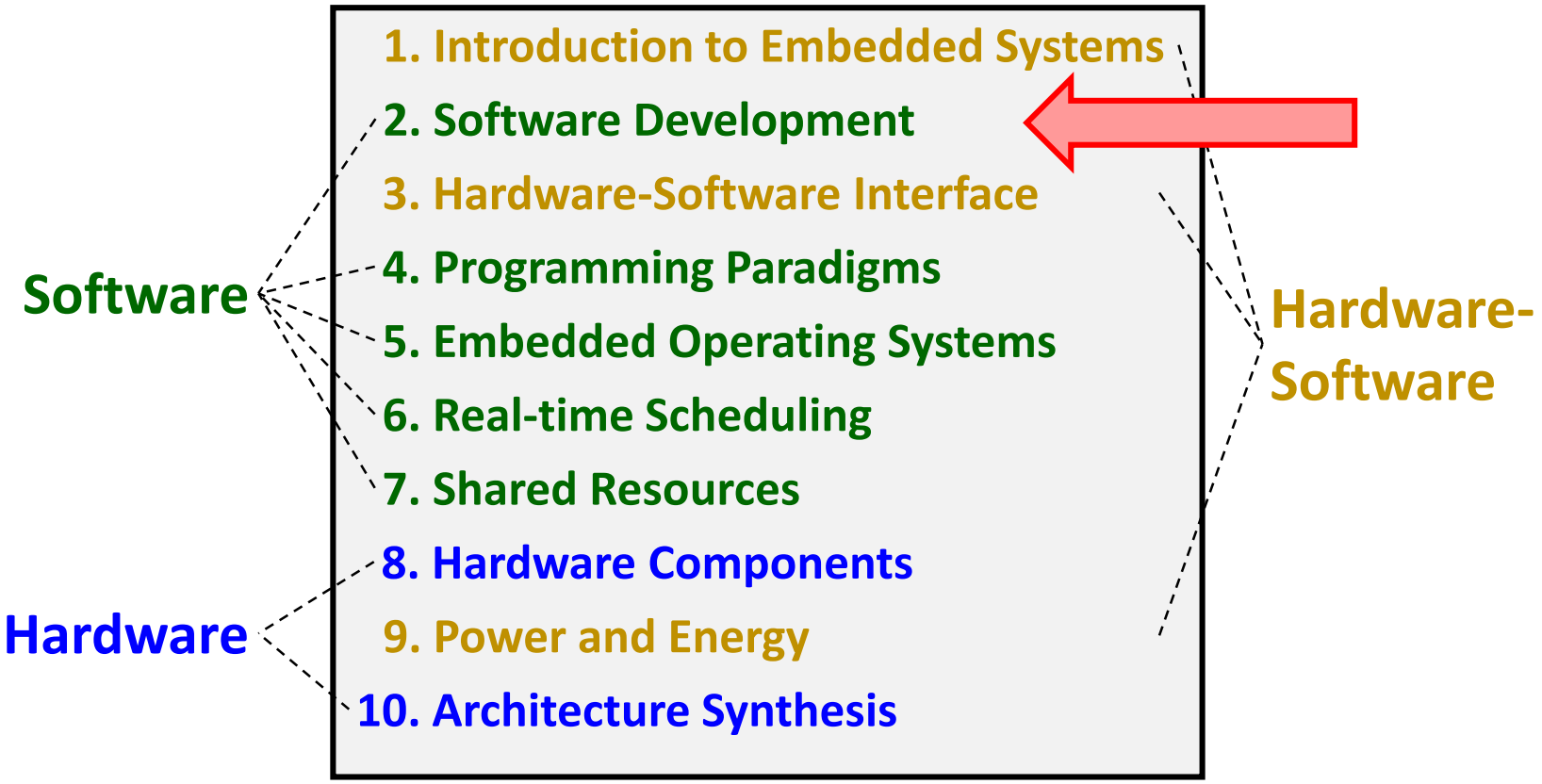
© Lothar Thiele

Computer Engineering and Networks Laboratory

ETH

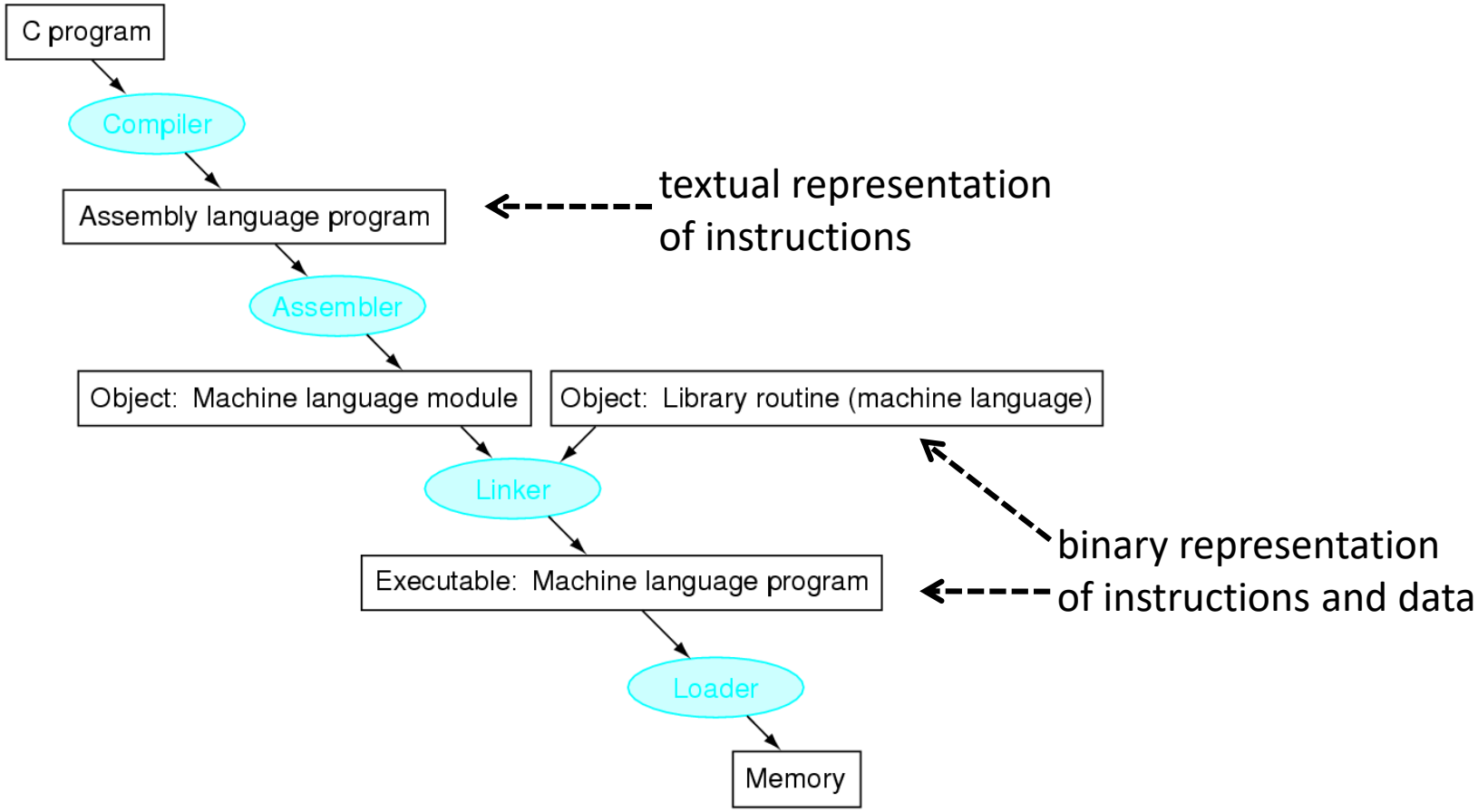
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Where we are …

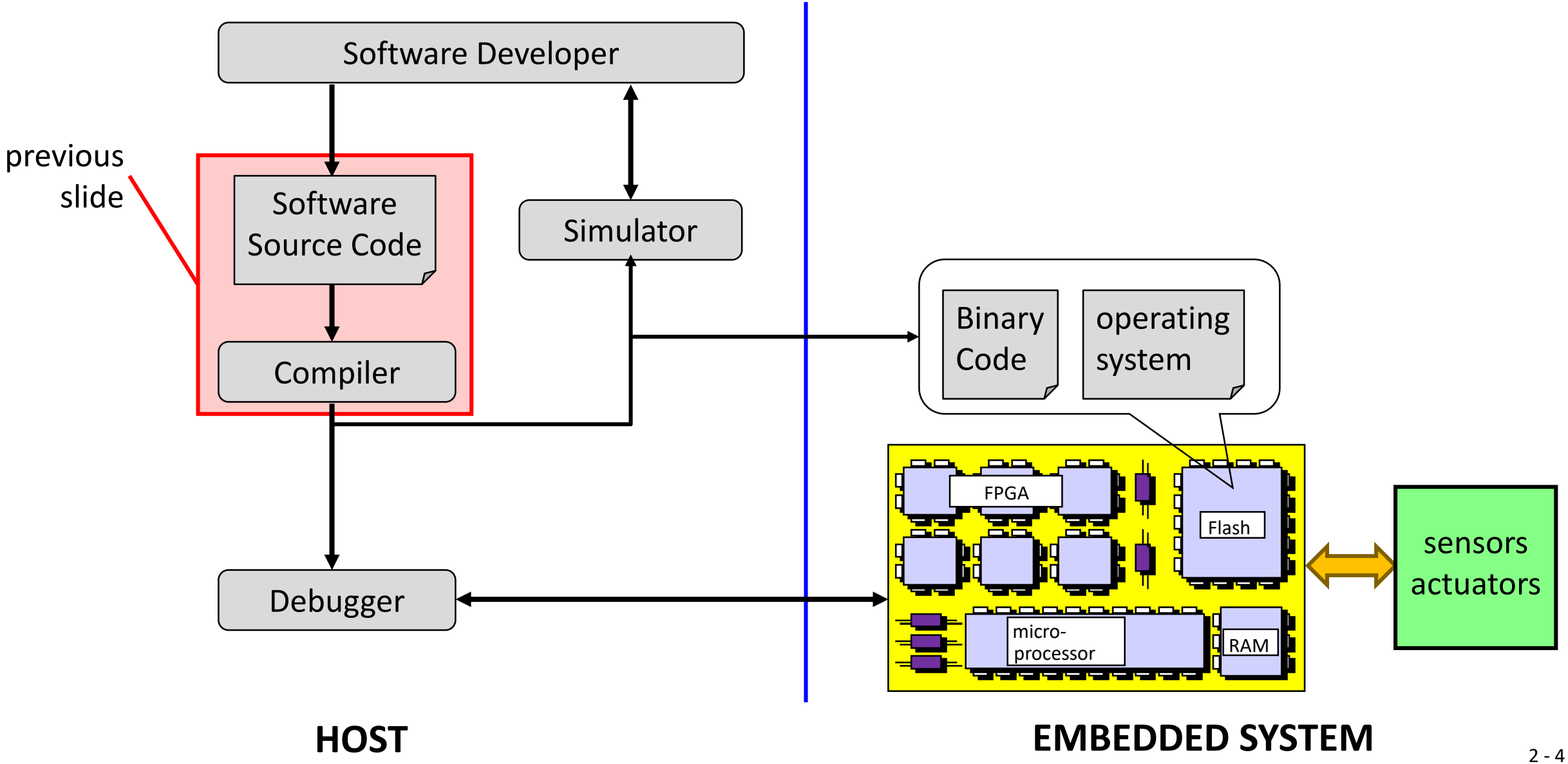

1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis
Software
Hardware
Hardware-
Software

# Remember: Computer Engineering I

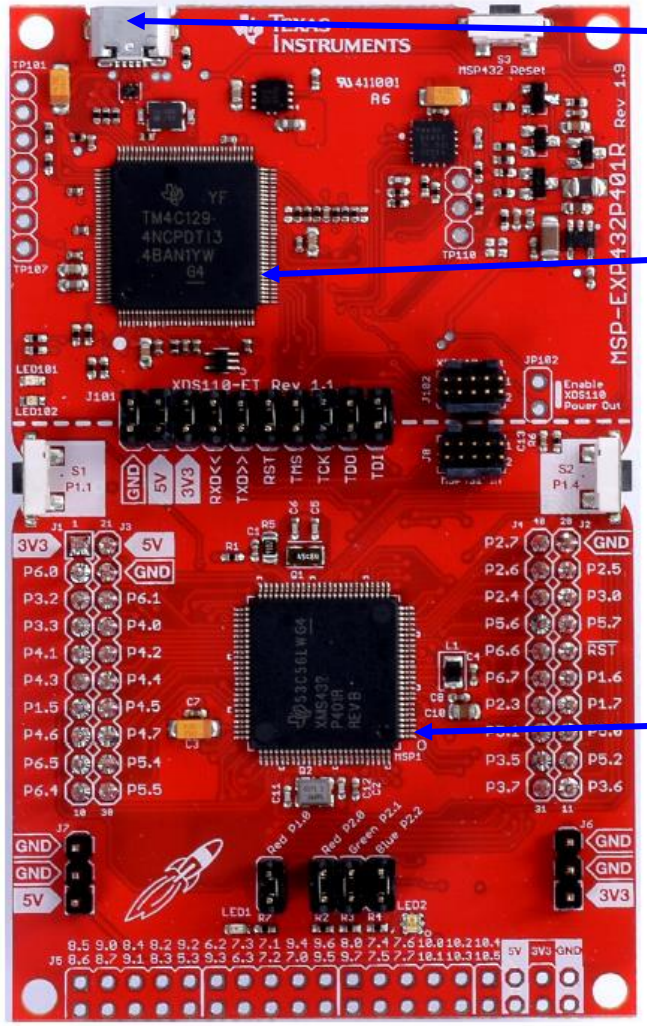
Compilation of a C program to machine language program:

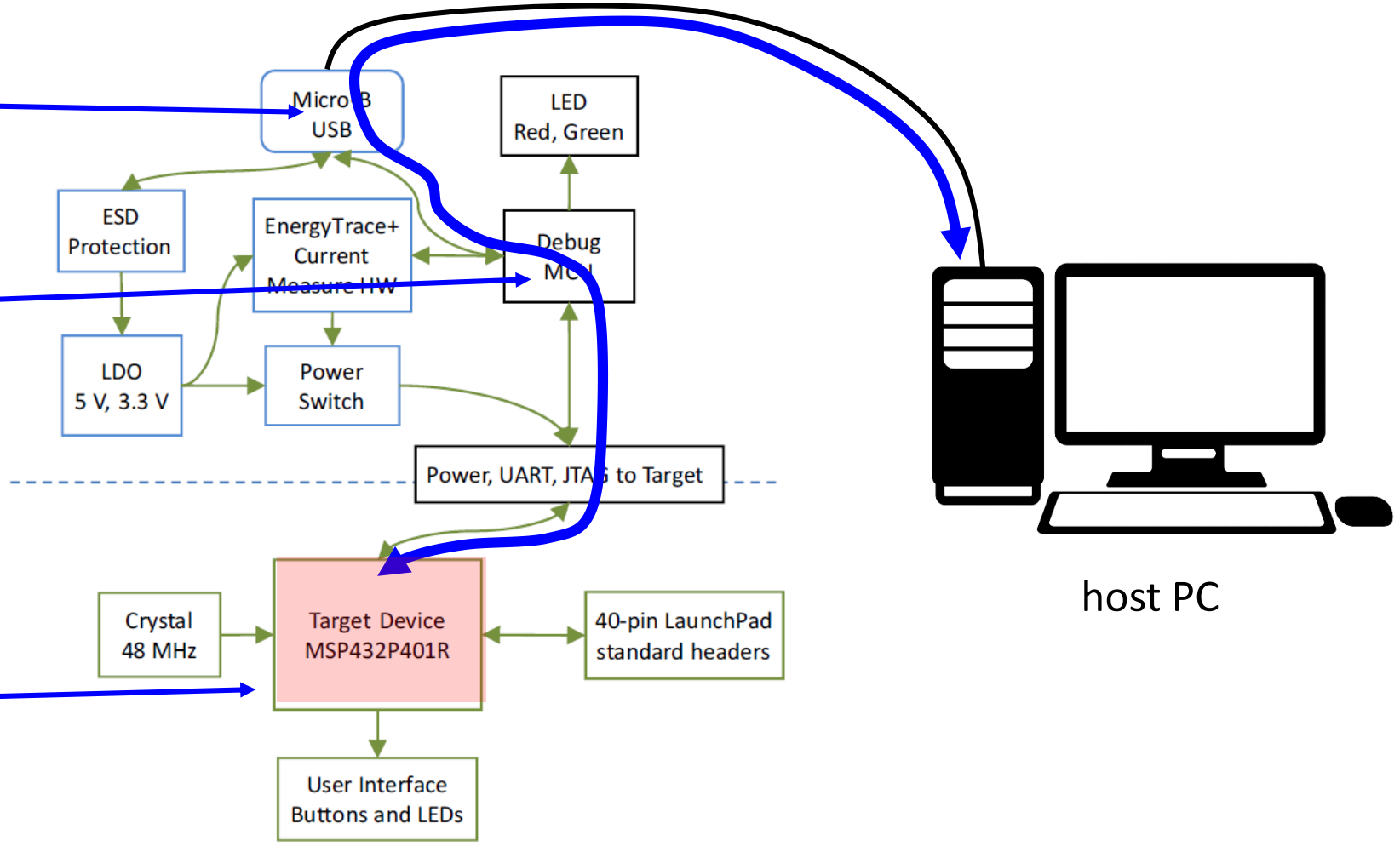


# Embedded Software Development



# Software Development with MSP432 (ES-Lab)

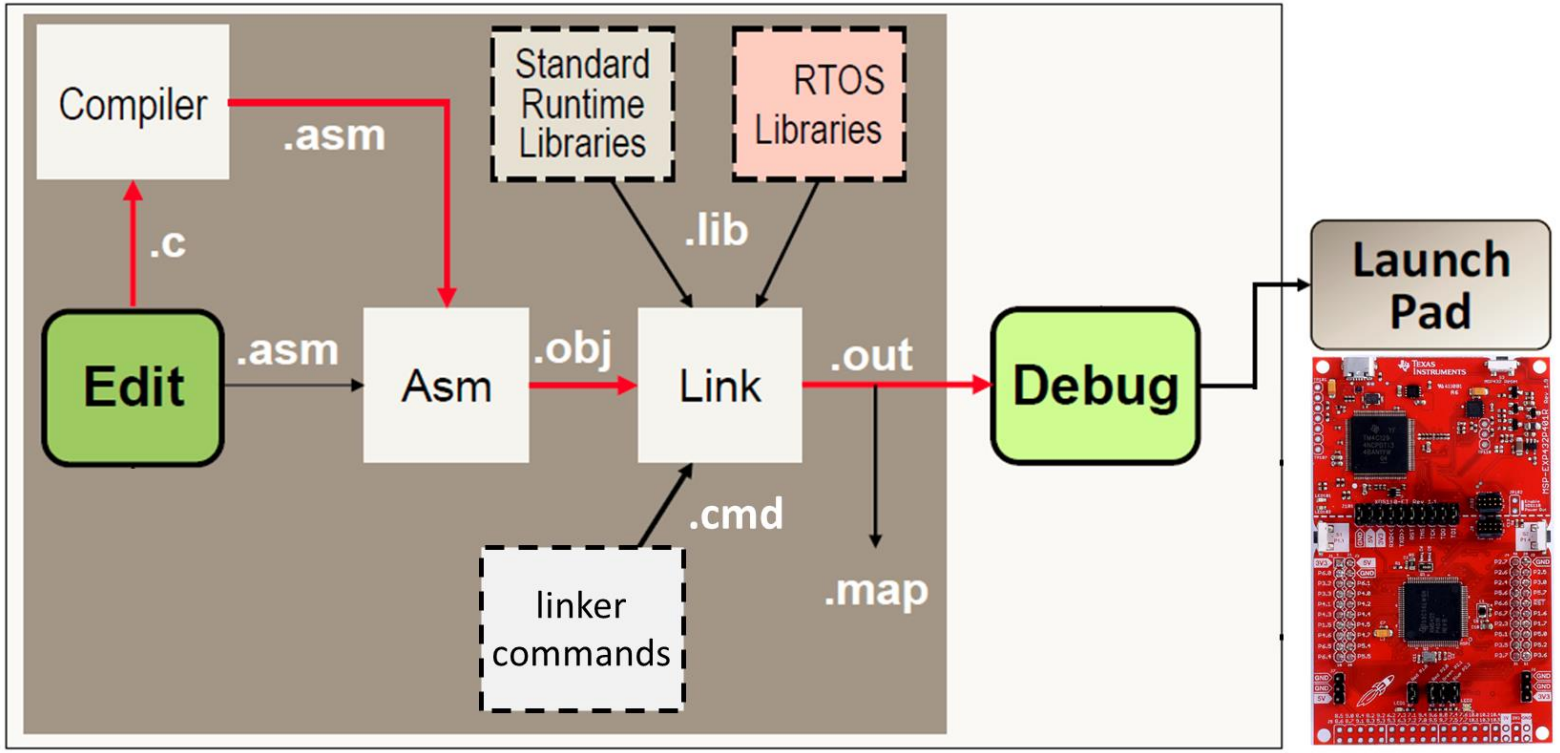




# Software Development (ES-Lab)

Software development is nowadays usually done with the support of an IDE (Integrated Debugger and Editor / Integrated Development Environment)

- edit and build the code
- debug and validate



# Software Development (ES-Lab)

assembly code

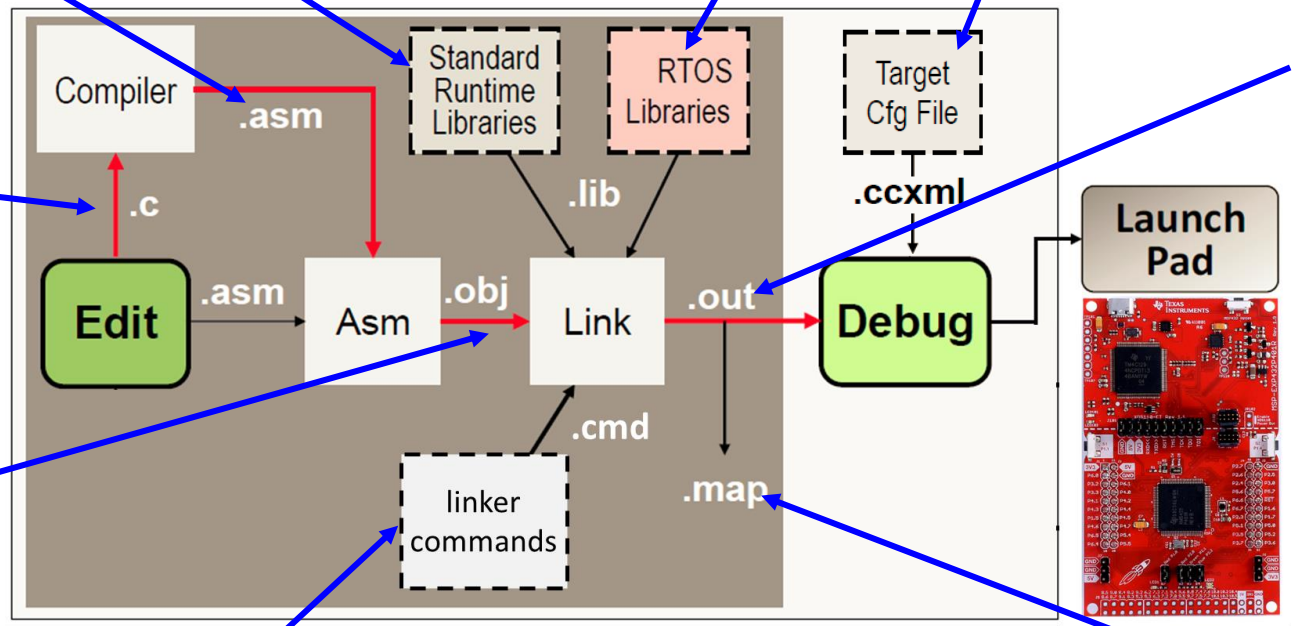
object libraries that are referenced in the code

object libraries that contain the operating system (if any)

target configuration file specifies the connection to the target (e.g. USB) and the target device

source code file in C

the executable output file that is loaded into flash memory on the processor




relocatable object file

Linker command file that tells the linker how to allocate memory and to stitch the object files and libraries together.

report created by the linker describing where the program and data sections are located in memory.

# Software Development (ES-Lab)

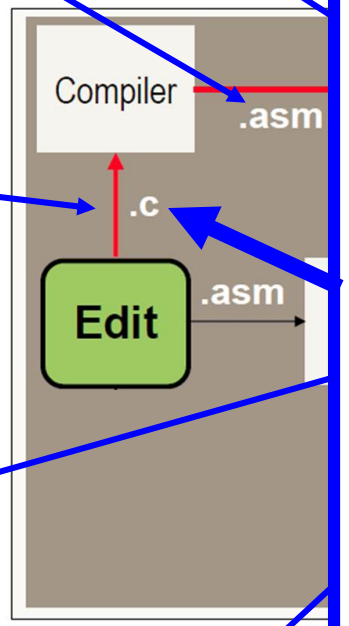
assembly code

object libraries that are refere in the code

source code file in C

relocatable object file

Linker command file tha how to allocate memory the object files and libra




```
...

/*
 * Main function
 */
int main(void)
{
    /* Halting WDT and disabling master interrupts */
    MAP_WDT_A_holdTimer();
    MAP_Interrupt_disableMaster();

    /* Seed the pseudo random num generator */
    srand(TLV->RANDOM_NUM_1);

    /* Set the core voltage level to VCORE1 */
    MAP_PCM_setCoreVoltageLevel(PCM_VCORE1);

    /* Set 2 flash wait states for Flash bank 0 and 1*/
    MAP_FlashCtl_setWaitState(FLASH_BANK0, 2);
    MAP_FlashCtl_setWaitState(FLASH_BANK1, 2);

    /* Default SysTick period for all 4 color states = 0.5s */
    periods[0] = 1500000;
    periods[1] = 1500000;
    periods[2] = 1500000;
    periods[3] = 1500000;

...
```

to the target device

e output file into flash he processor

g

s

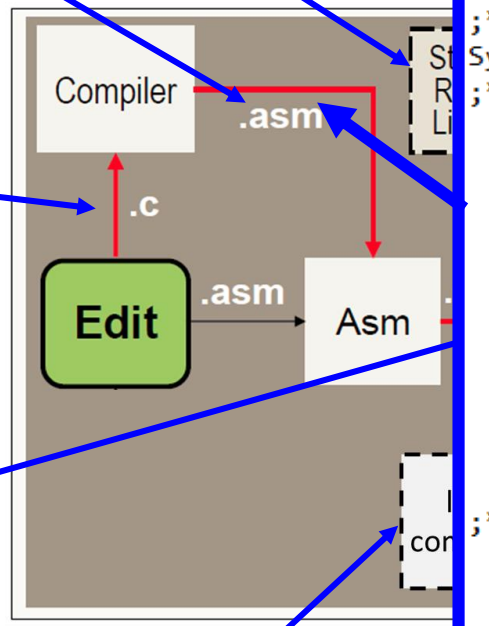
# Software Development

assembly code

object libraries that are referenced in the code

source code file in C

relocatable object file



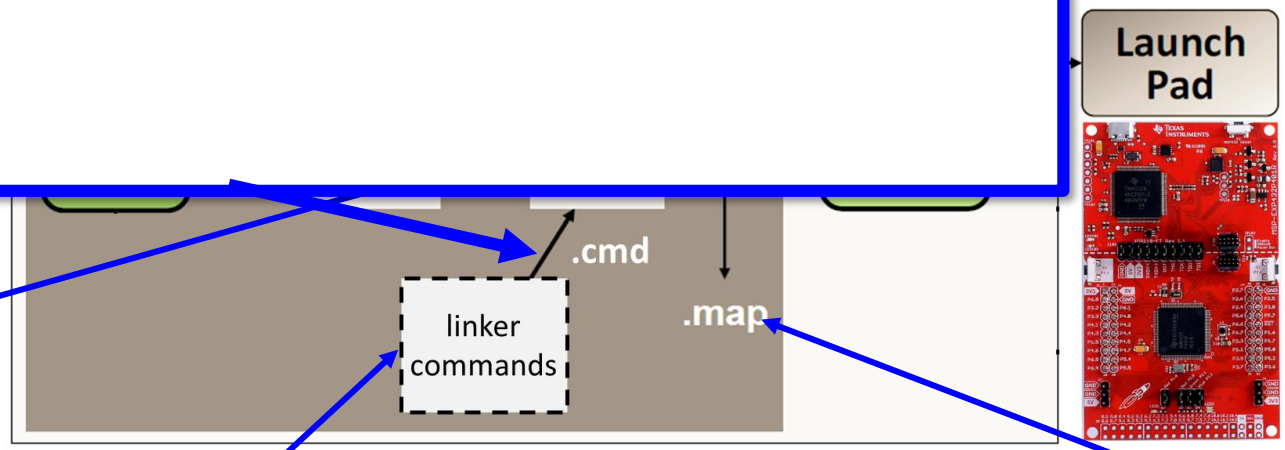

Linker command file that tells how to allocate memory and t the object files and libraries to

```
...

;*****************************************************************************
;* FUNCTION NAME: SysTick_Handler                                            *
;*                                                                           *
;*   Regs Modified     : A1,A2,A3,A4,V9,SP,LR,SR,D0,D0_hi,D1,D1_hi,D2,D2_hi, *
;*                       D3,D3_hi,D4,D4_hi,D5,D5_hi,D6,D6_hi,D7,D7_hi,       *
;*                       FPEXC,FPSCR                                         *
;*   Regs Used         : A1,A2,A3,A4,V9,SP,LR,SR,D0,D0_hi,D1,D1_hi,D2,D2_hi, *
;*                       D3,D3_hi,D4,D4_hi,D5,D5_hi,D6,D6_hi,D7,D7_hi,       *
;*                       FPEXC,FPSCR                                         *
;*   Local Frame Size  : 0 Args + 0 Auto + 4 Save = 4 byte                   *
;*****************************************************************************
SysTick_Handler:
;* ---------------------------------------------------------------------------*
        .dwcfi  cfa_offset, 0
        PUSH      {A4, LR}              ; [DPU_3_PIPE]
        .dwcfi  cfa_offset, 8
        .dwcfi  save_reg_to_mem, 14, -4
        .dwcfi  save_reg_to_mem, 3, -8
        .dwpsn  file "../main.c",line 374,column 5,is_stmt,isa 1
        LDR       A1, $C$CON64          ; [DPU_3_PIPE] |374|
        LDR       A1, [A1, #0]          ; [DPU_3_PIPE] |374|
        CMP       A1, #1                ; [DPU_3_PIPE] |374|
        BNE       ||$C$L20||            ; [DPU_3_PIPE] |374|
        ; BRANCHCC OCCURS {||$C$L20||}   ; [] |374|
;* ---------------------------------------------------------------------------*
        .dwpsn  file "../main.c",line 375,column 9,is_stmt,isa 1
        LDR       A2, $C$CON65          ; [DPU_3_PIPE] |375|
        LDR       A1, [A2, #0]          ; [DPU_3_PIPE] |375|
        ADDS      A1, A1, #1            ; [DPU_3_PIPE] |375|
        STR       A1, [A2, #0]          ; [DPU_3_PIPE] |375|
;* ---------------------------------------------------------------------------*

...
```

```
...

MEMORY
{
    MAIN       (RX) : origin = 0x00000000, length = 0x00040000
    INFO       (RX) : origin = 0x00200000, length = 0x00004000
#ifdef __TI_COMPILER_VERSION__
#if     __TI_COMPILER_VERSION__ >= 15009000
    ALIAS
    {
    SRAM_CODE  (RWX): origin = 0x01000000
    SRAM_DATA  (RW) : origin = 0x20000000
    } length = 0x00010000
#else
    /* Hint: If the user wants to use ram functions, please observe that SRAM_CODE          */
    /* and SRAM_DATA memory areas are overlapping. You need to take measures to separate    */
    /* data from code in RAM. This is only valid for Compiler version earlier than 15.09.0.STS.*/
    SRAM_CODE  (RWX): origin = 0x01000000, length = 0x00010000
    SRAM_DATA  (RW) : origin = 0x20000000, length = 0x00010000
#endif
#endif
}

...
```

target configuration file specifies the connection to the target (e.g. USB) and the target device

the executable output file that is loaded into flash memory on the processor




relocatable object file

Linker command file that tells the linker how to allocate memory and to stitch the object files and libraries together.

report created by the linker describing where the program and data sections are located in memory.

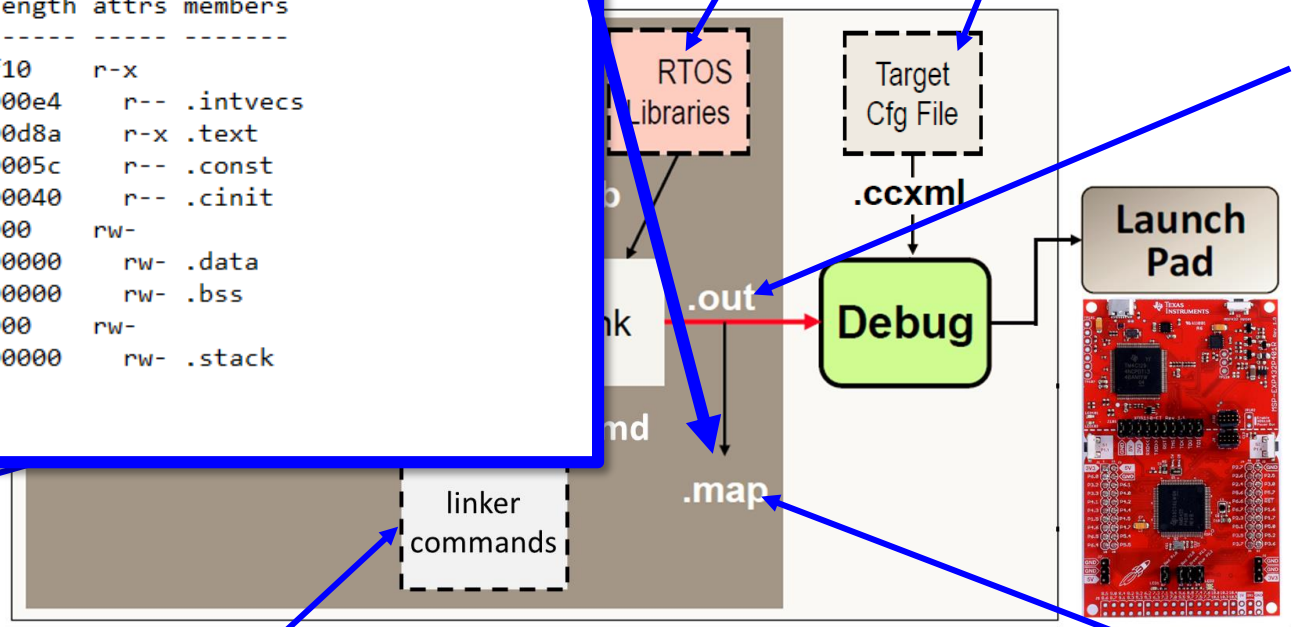
# Lab)

```
...

MEMORY CONFIGURATION

         name             origin    length      used     unused   attr    fill
----------------------  --------  ---------  --------  --------  ----  --------
  MAIN                  00000000   00040000  00000f0a  0003f0f6  R  X
  INFO                  00200000   00004000  00000000  00004000  R  X
  SRAM_CODE             01000000   00010000  00000268  0000fd98  RW X
  SRAM_DATA             20000000   00010000  00000268  0000fd98  RW


SEGMENT ALLOCATION MAP

run origin  load origin   length   init length attrs members
----------  -----------  ---------- ----------- ----- -------
00000000    00000000     00000f10   00000f10    r-x
  00000000    00000000     000000e4   000000e4    r-- .intvecs
  000000e4    000000e4     00000d8a   00000d8a    r-x .text
  00000e70    00000e70     0000005c   0000005c    r-- .const
  00000ed0    00000ed0     00000040   00000040    r-- .cinit
20000000    20000000     00000068   00000000    rw-
  20000000    20000000     00000050   00000000    rw- .data
  20000050    20000050     00000018   00000000    rw- .bss
2000fe00    2000fe00     00000200   00000000    rw-
  2000fe00    2000fe00     00000200   00000000    rw- .stack

...
```

ct libraries that
tain the operating
em (if any)

target configuration file specifies the connection to the target (e.g. USB) and the target device

the executable output file that is loaded into flash memory on the processor




relocatable object file

Linker command file that tells the linker how to allocate memory and to stitch the object files and libraries together.

report created by the linker describing where the program and data sections are located in memory.

# Software Development (ES-Lab)

assembly code

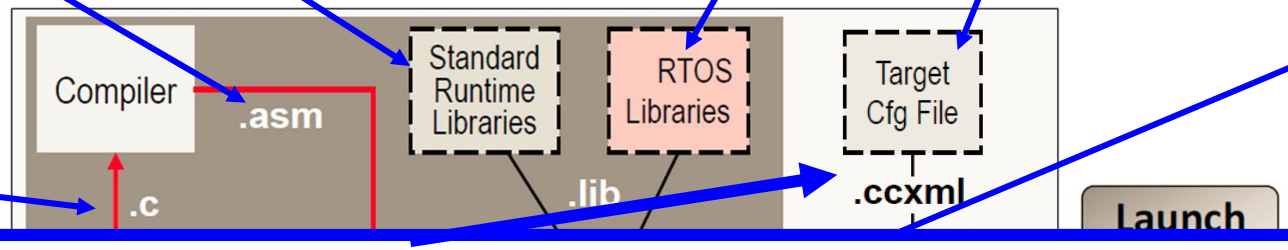
object libraries that are referenced in the code

object libraries that contain the operating system (if any)

target configuration file specifies the connection to the target (e.g. USB) and the target device

source code file in C

the executable output file that is loaded into flash memory on the processor




```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configurations XML_version="1.2" id="configurations_0">
    <configuration XML_version="1.2" id="configuration_0">
        <instance XML_version="1.2" desc="Texas Instruments XDS110 USB Debug Probe" href="connections/ ...
        <connection XML_version="1.2" id="Texas Instruments XDS110 USB Debug Probe">
            <instance XML_version="1.2" href="drivers/tixds510cs_dap.xml" id="drivers" xml= ...
            <instance XML_version="1.2" href="drivers/tixds510cortexM.xml" id="drivers" xml= ...
            <property Type="choicelist" Value="2" id="SWD Mode Settings">
                <choice Name="SWD Mode - Aux COM port is target TDO pin" value="nothing"/>
            </property>
            <platform XML_version="1.2" id="platform_0">
                <instance XML_version="1.2" desc="MSP432P401R" href="devices/msp432p401r.xml" id= ...
            </platform>
        </connection>
    </configuration>
</configurations>
```

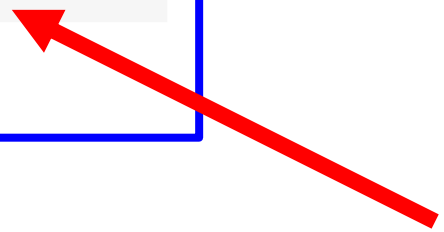
describing a sections

# Much more in the ES-PreLab …

- The Pre-lab is intended for students with missing background in software development in C and working with an integrated development environment.

## Timetable

| Date | Lecture | Exercice | Lab |
|---|---|---|---|
| 27.09.2021 | 1. Introduction<br>2. Software Development | | |
| 29.09./01.10.2021 | | | 0. Prelab [MM] |
| 04.10.2021 | 3. Hardware-Software In-terface | | |



# Much more in the ES-PreLab …

- The Pre-lab is intended for students with missing background in software development in C and working with an integrated development environment.

## Embedded Systems 1.0.1 – Filling the gaps

### Goals of this Lab

The goal of this lab session is to give a quick crash-course on all necessary background for the following labs. You are expected to have some basic knowledge about programming, but programming an embedded systems is slightly different than Python, Java, or Matlab.

Here are the main topics the pre-lab covers:

- Definitions and keywords – Know what you are talk about
- C programming – Review of the fundamentals
- Embedded systems programming – Specific types and basic operations
- Schematics – Find your way around a processor schematics
- Demo application – If you can make it, you're good to go!

# Embedded Systems

## 3. Hardware Software Interface

© Lothar Thiele

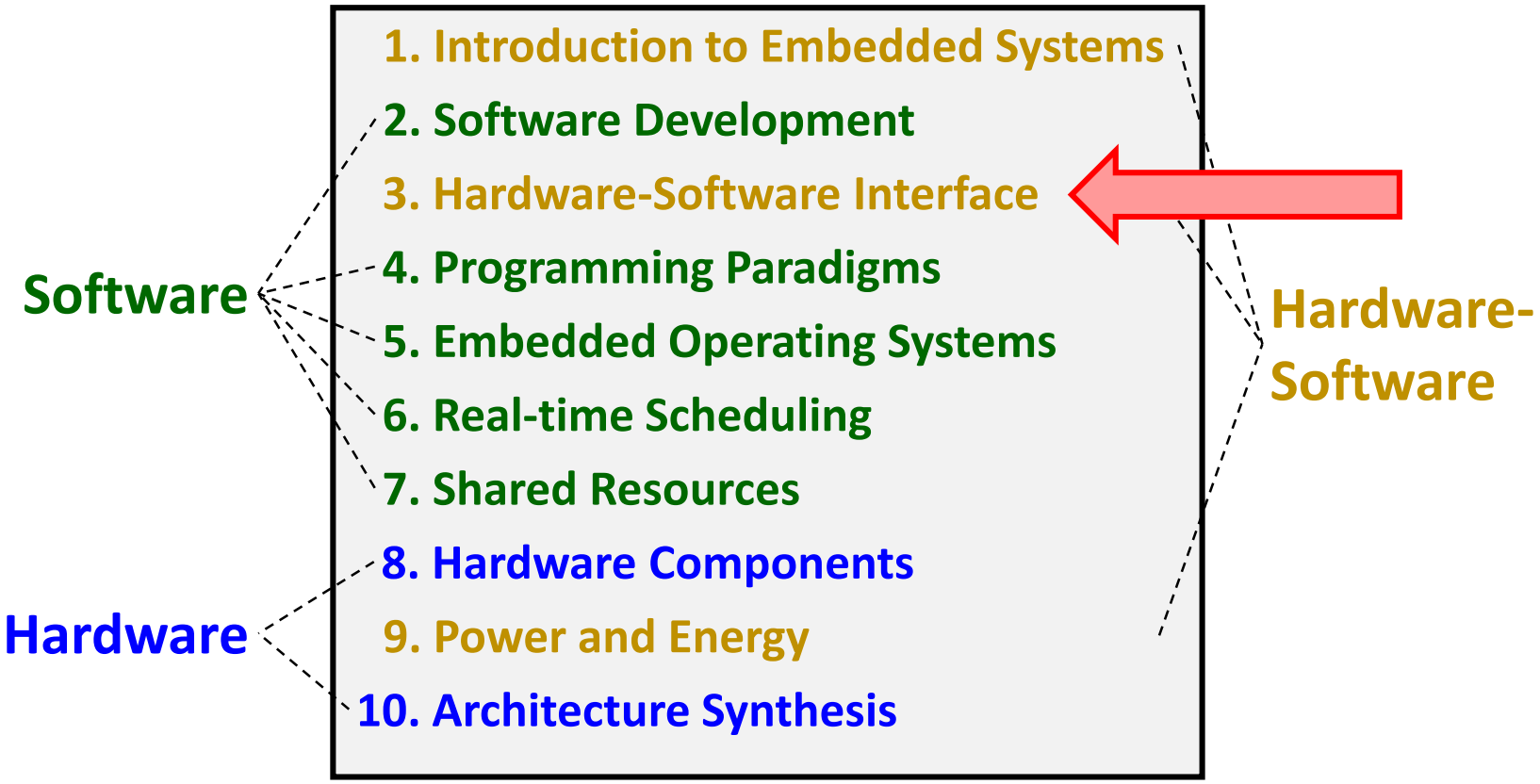
Computer Engineering and Networks Laboratory

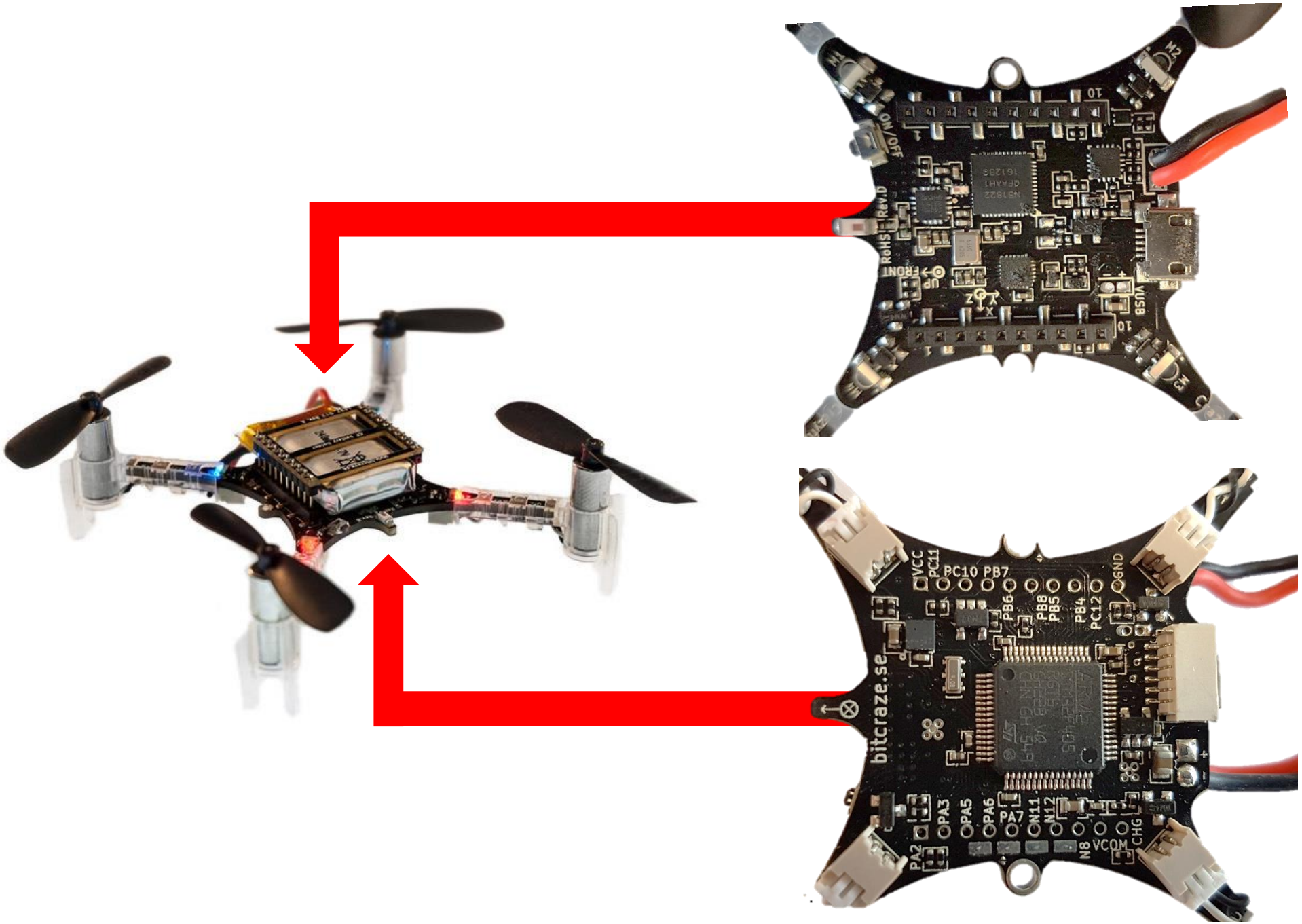
ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

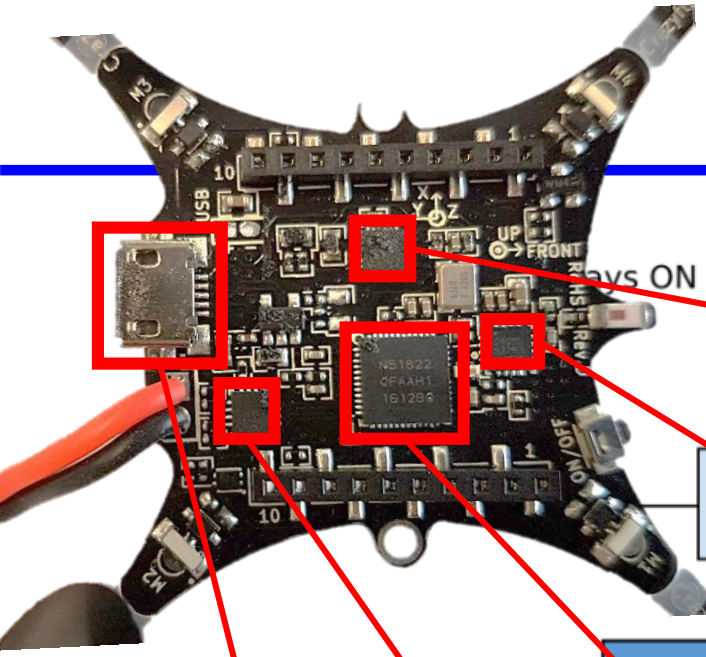
# Do you Remember ?

# Where we are …


1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis
Software
Hardware
Hardware-Software


ON/OFF
REV.D
RoHS
FRONT
USB
bitcraze.se
VCC
PC11
PC10 PB7
GND
PB6
PB8
PB5
PB4
PC12
PA2
PA3
PA5
PA6
PA7
CN11
CN12
N8
VCOM
CHG

# High-Level Physical View



ays ON power domain

Power switched by nRF51 (VCC)

RF power amplifier

10DOF IMU
- 3-axis accelerometer
- 3-axis gyro
- 3-axis magnetomer
- Pressure sensor

I2C

Push button

nRF51822
- 16MHz Cortex-M0
- 16kB RAM, 256KB Flash
- BLE and NRF radio

UART

STM32F405
- 168MHz Cortex-M4
- 196kB RAM, 1MB Flash

PWM

Motor driver

I2C

SPI/I2C/GPIO/PWM

Wkup/OW/GPIO

Power supplies and battery charger

+5V

Charge/VBAT/VCC

Expansion port

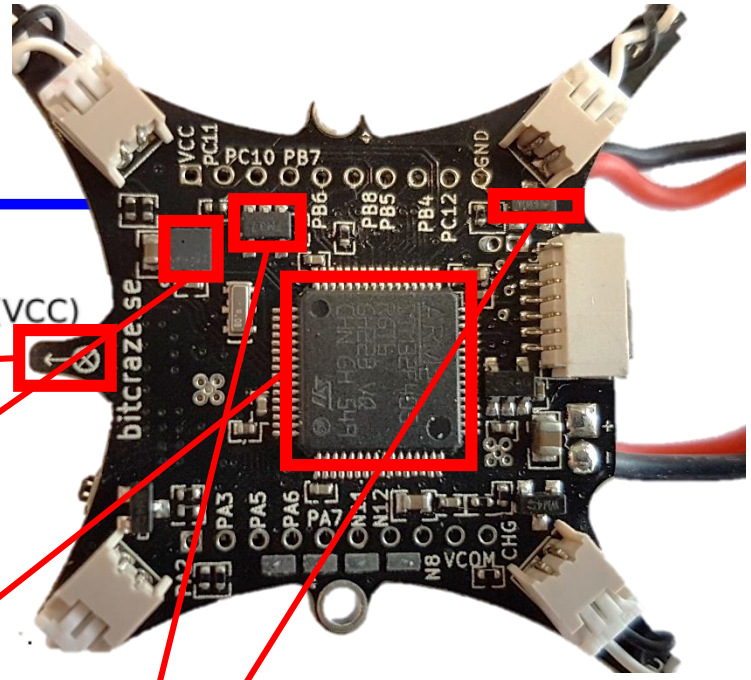
EEPROM

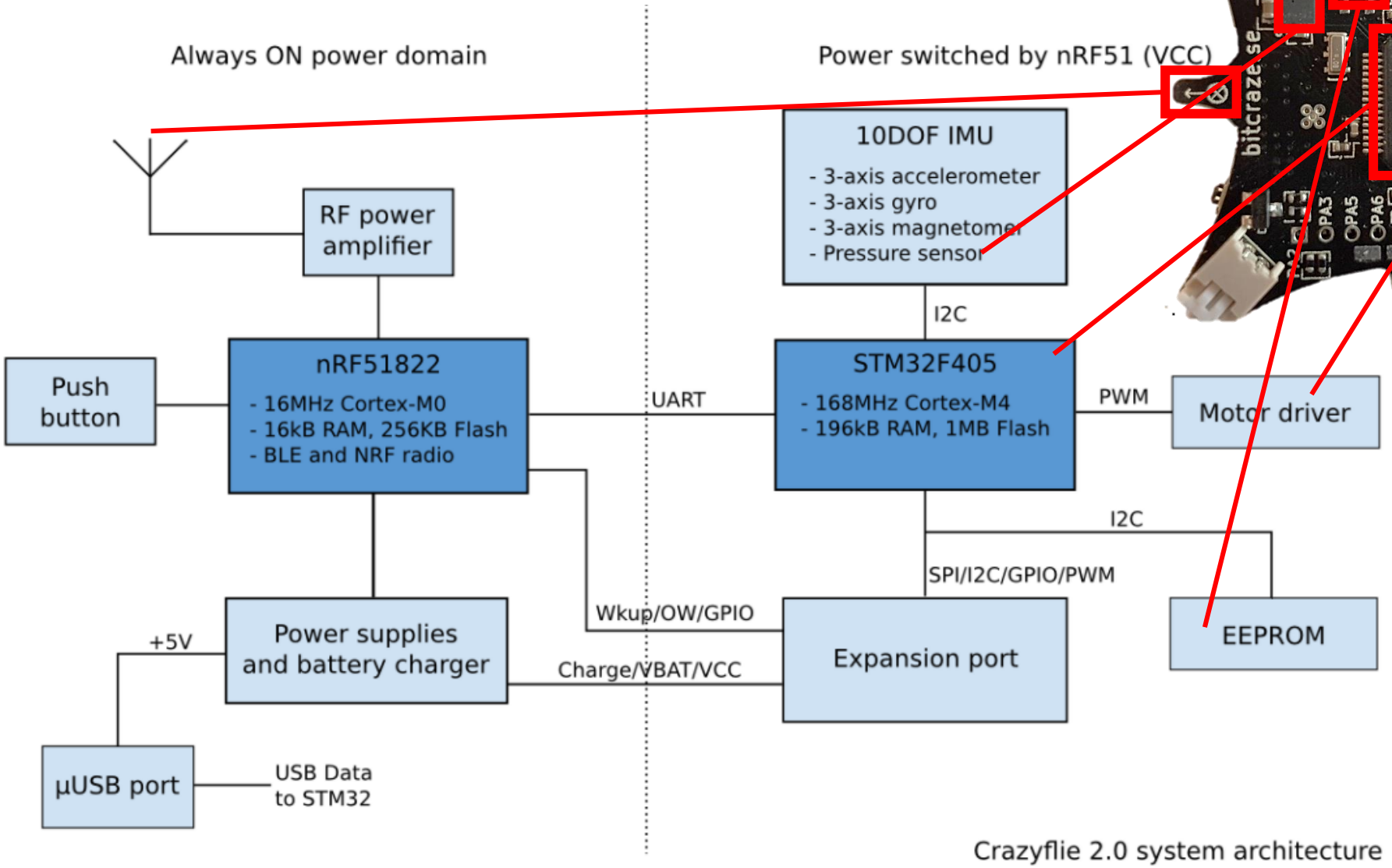
µUSB port

USB Data to STM32

Crazyflie 2.0 system architecture

# High-Level Physical View




Crazyflie 2.0 system architecture

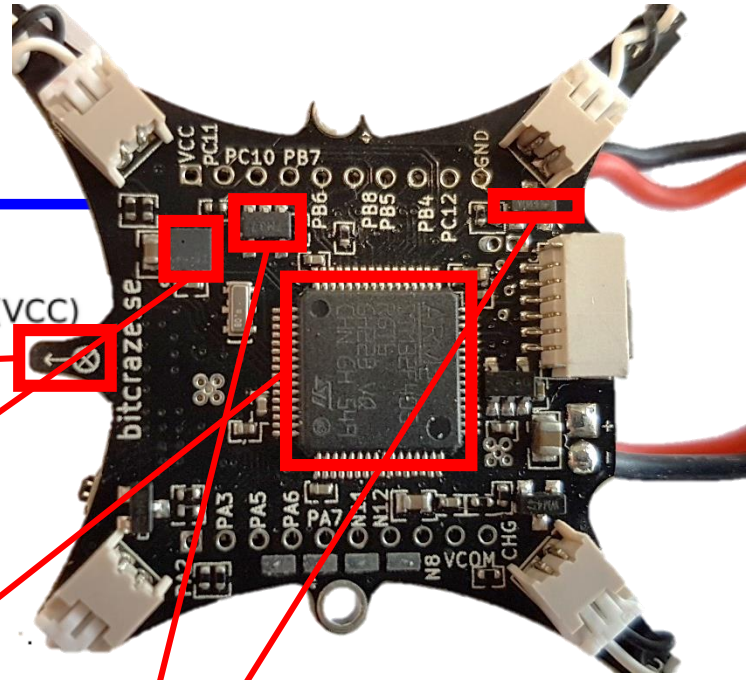
# What you will learn …

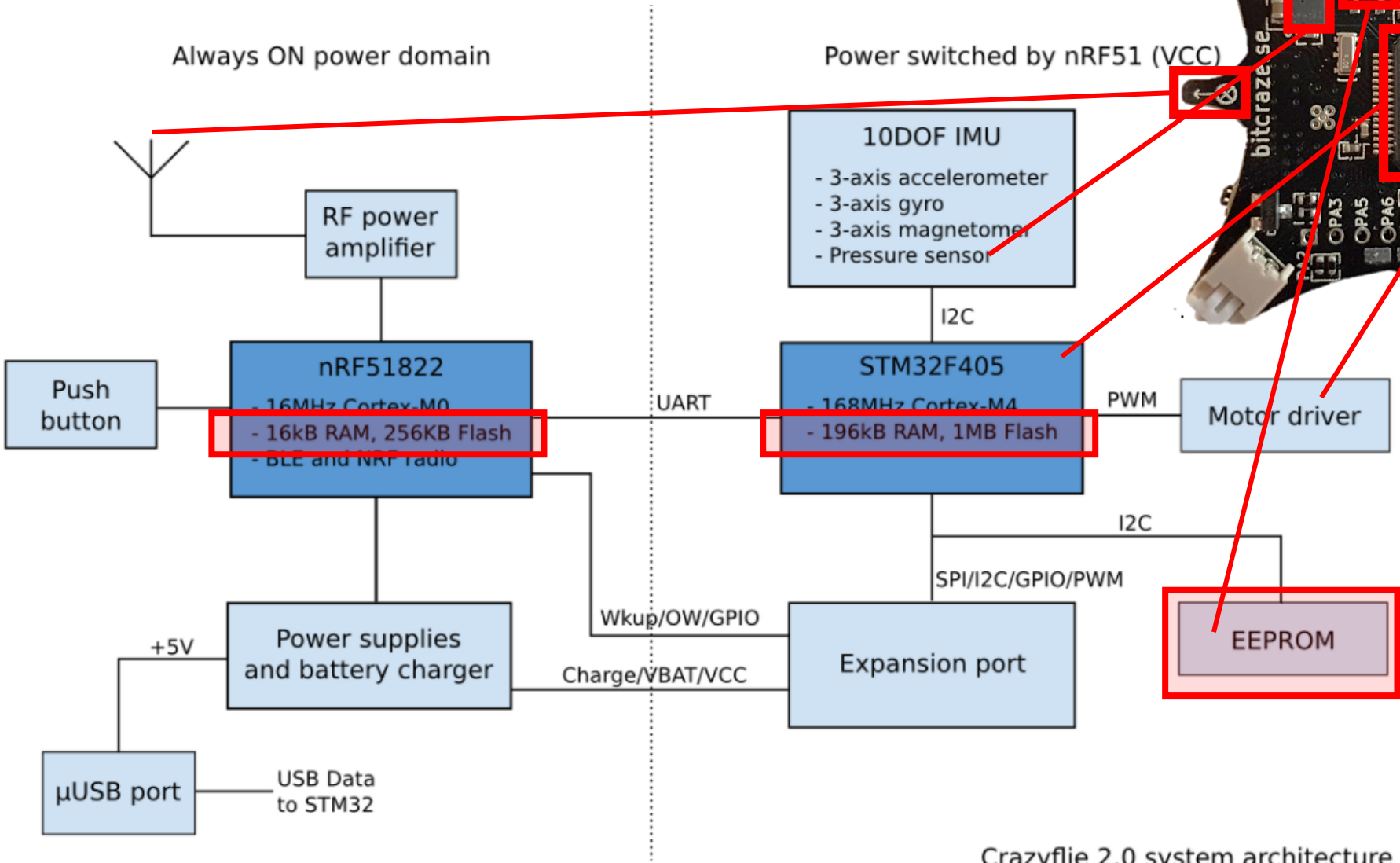
*Hardware-Software Interfaces in Embedded Systems*

- *Storage*
  - SRAM / DRAM / Flash
  - Memory Map
- *Input and Output*
  - UART Protocol
  - Memory Mapped Device Access
  - SPI Protocol
- *Interrupts*
- *Clocks and Timers*
  - Clocks
  - Watchdog Timer
  - System Tick
  - Timer and PWM

# Storage

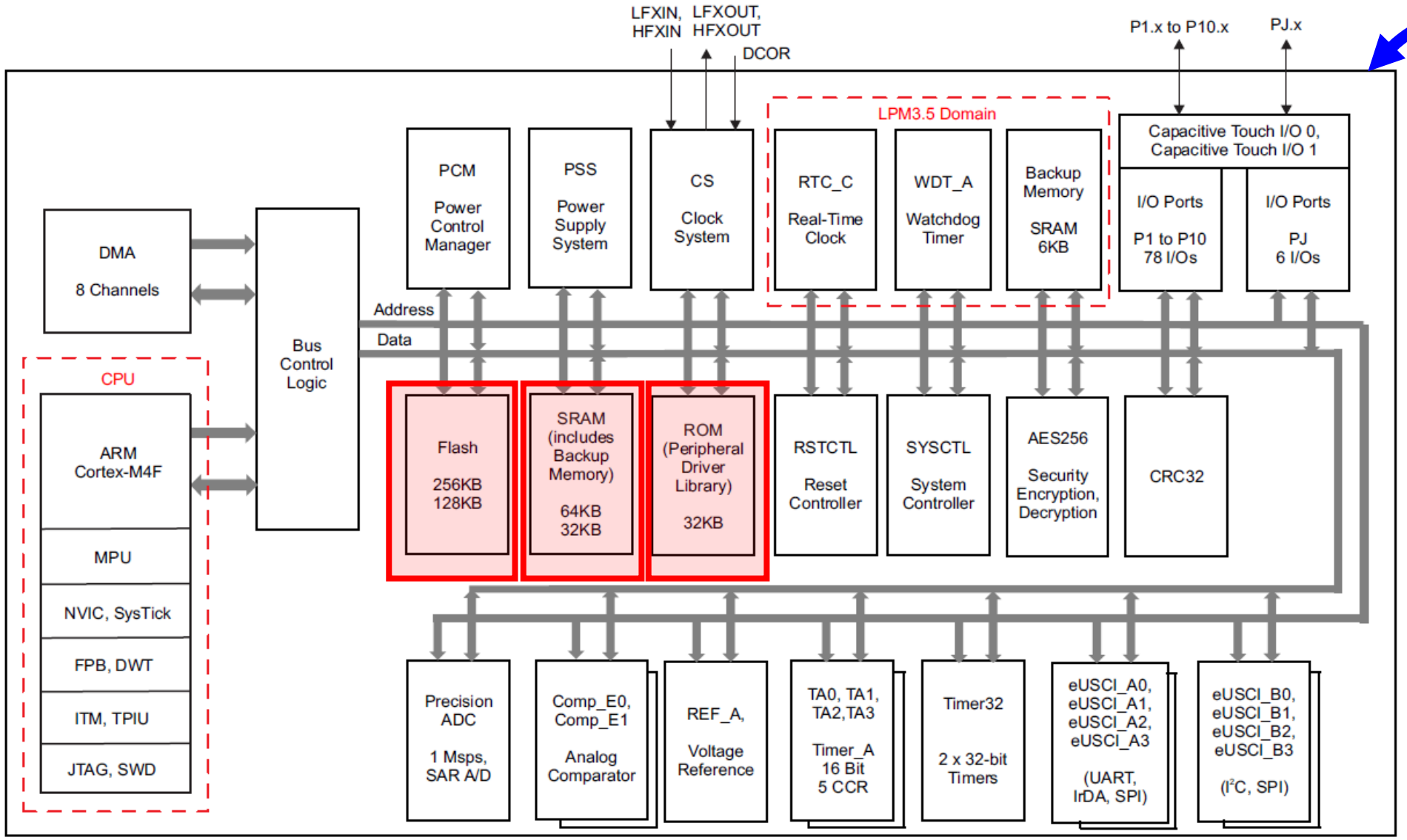
# Remember … ?

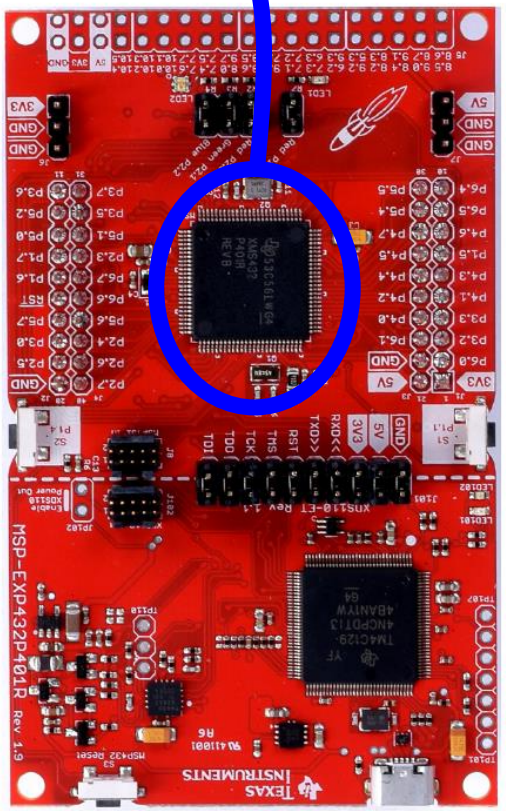



Crazyflie 2.0 system architecture

# MSP432P401R (ES-Lab)



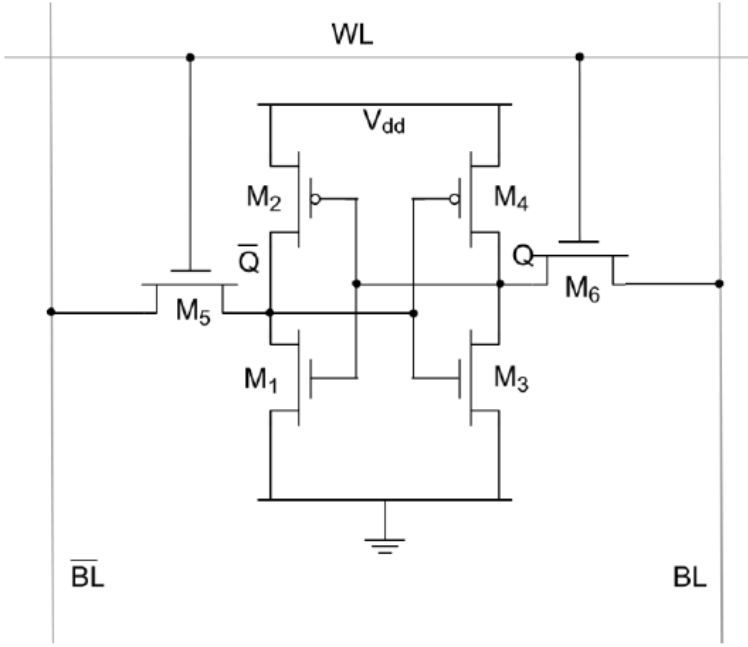



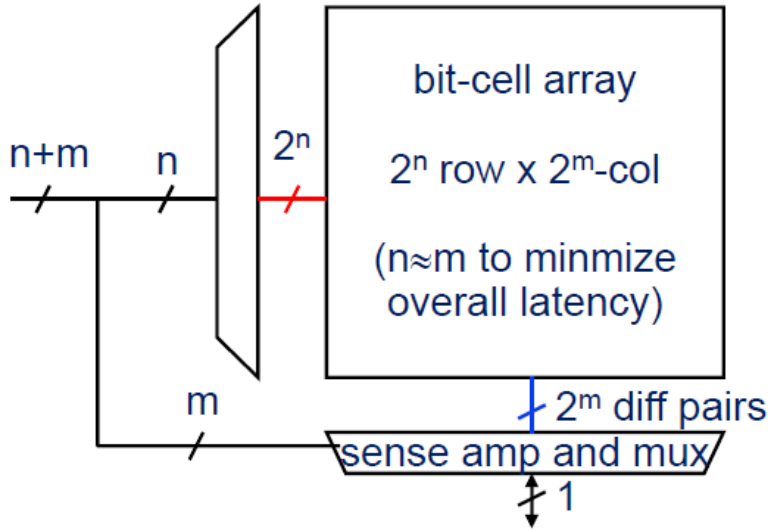
# Storage

## SRAM / DRAM / Flash

# Static Random Access Memory (SRAM)

- *Single bit is stored in a bi-stable circuit*
- *Static Random Access Memory* is used for
  - caches
  - register file within the processor core
  - small but fast memories
- *Read:*
  1. Pre-charge all bit-lines to average voltage
  2. decode address (n+m bits)
  3. select row of cells using n single-bit word lines (WL)
  4. selected bit-cells drive all bit-lines BL ($2^m$ pairs)
  5. sense difference between bit-line pairs and read out
- *Write:*
  - select row and overwrite bit-lines using strong signals



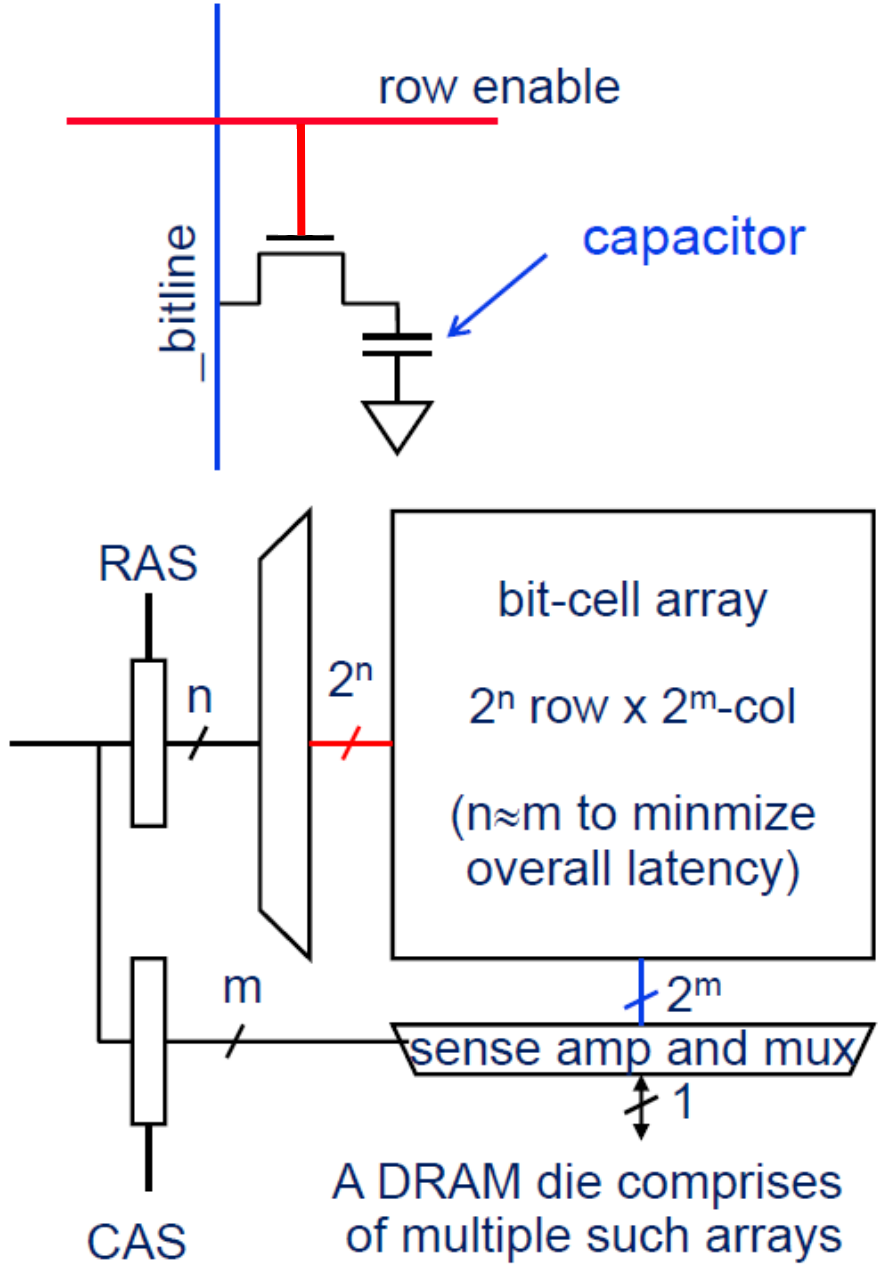
# Dynamic Random Access (DRAM)

*Single bit is stored as a charge in a capacitor*

- Bit cell loses charge when read, bit cell drains over time
- Slower access than with SRAM due to small storage capacity in comparison to capacity of bit-line.
- Higher density than SRAM (1 vs. 6 transistors per bit)

DRAMs require *periodic refresh* of charge

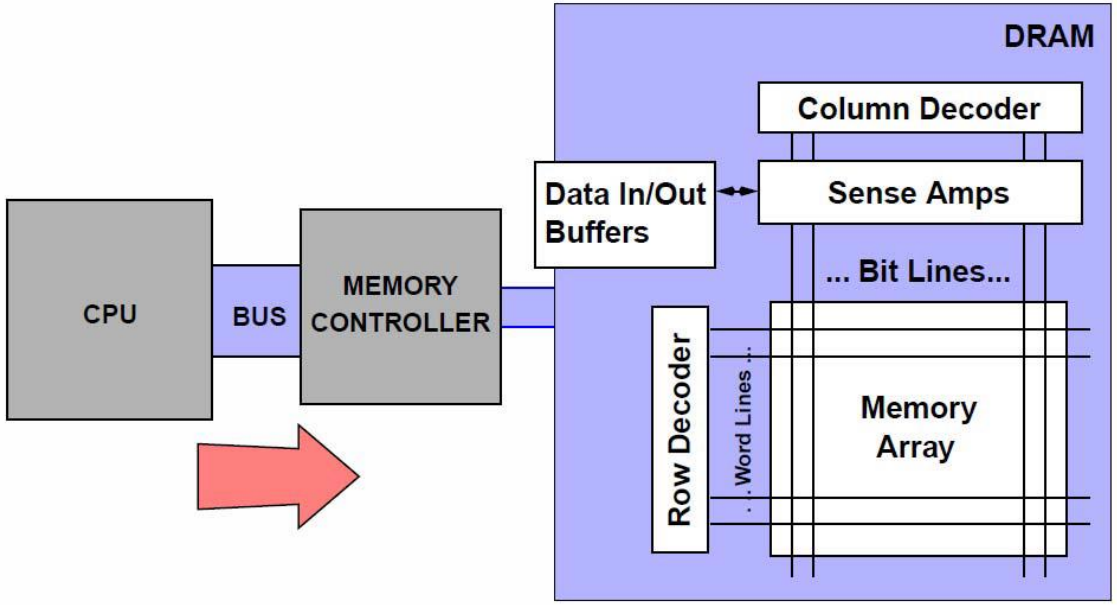
- Performed by the memory controller
- Refresh interval is tens of ms
- DRAM is unavailable during refresh



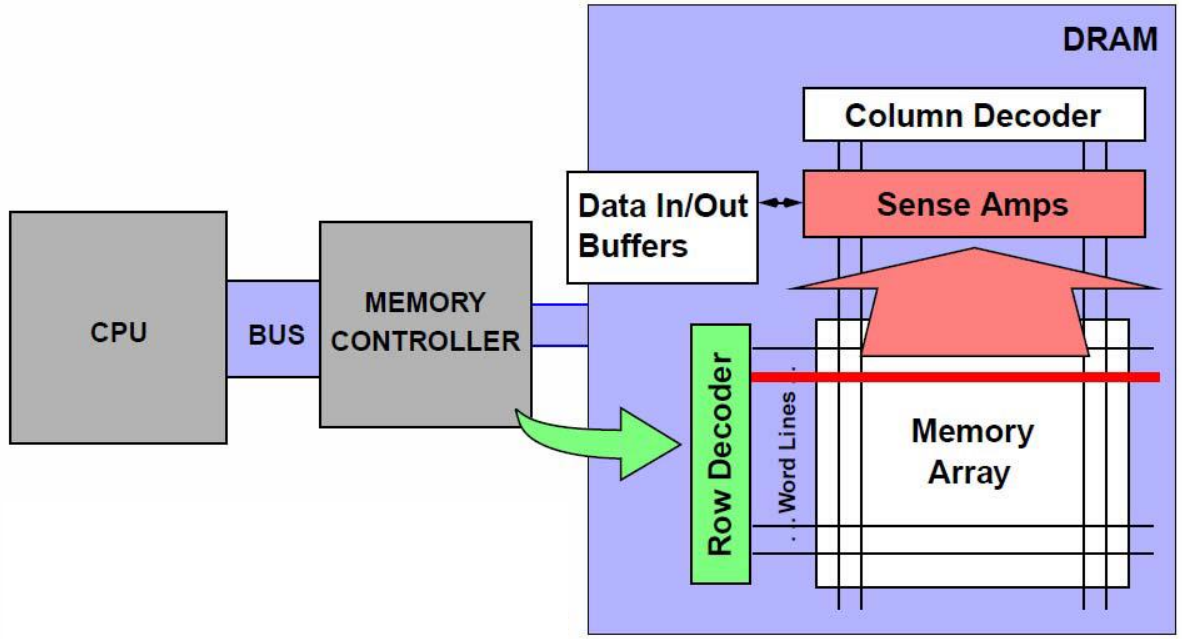

(RAS/CAS = row/column address select)

# DRAM – Typical Access Process

## *1. Bus Transmission*

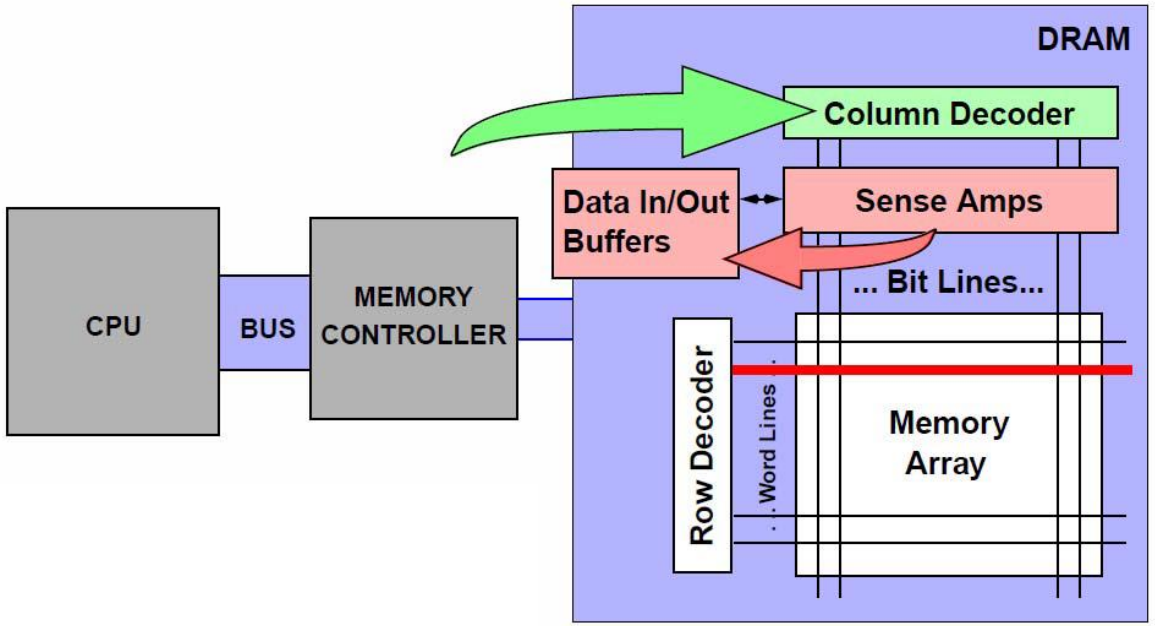



## *2. Precharge and Row Access*

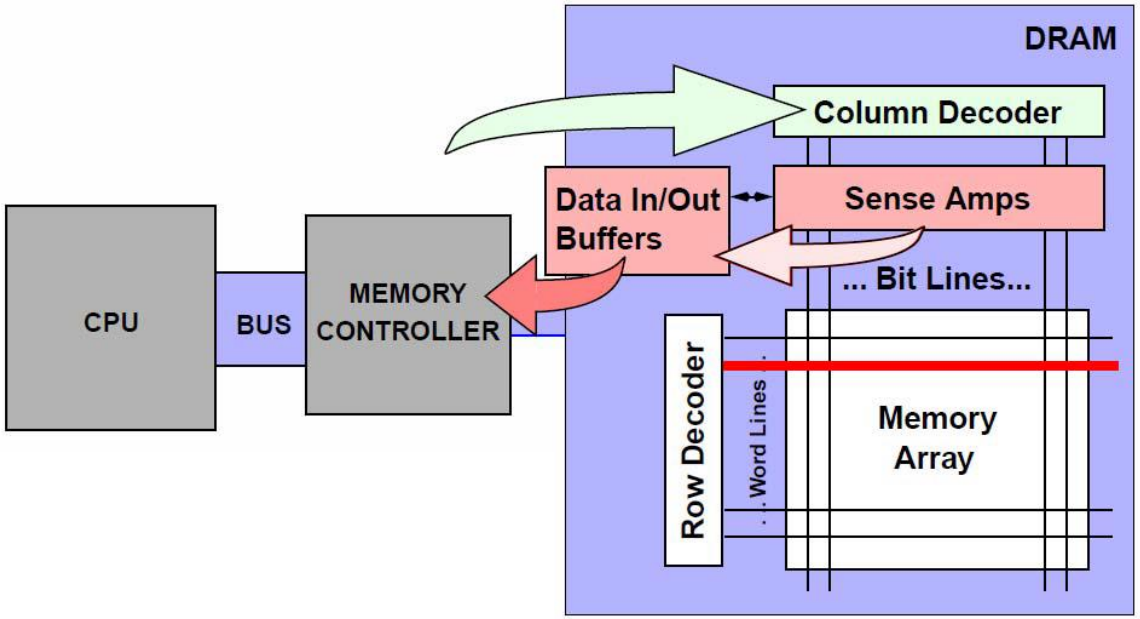


# DRAM – Typical Access Process

## *3. Column Access*




## *4. Data Transfer and Bus Transmission*

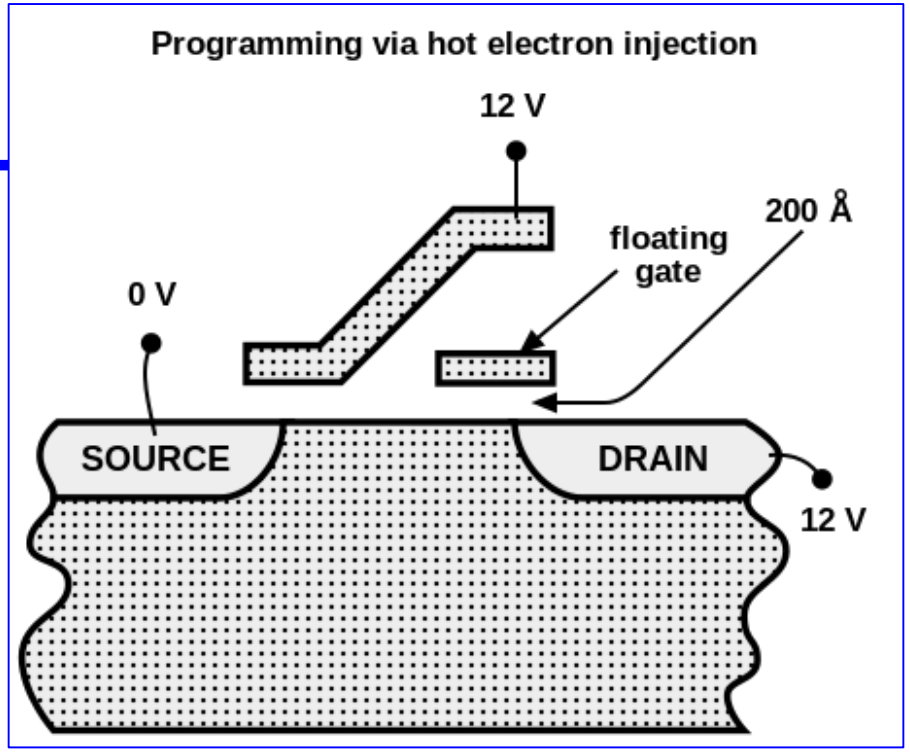


# Flash Memory

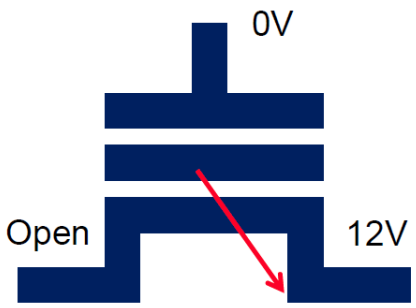
*Electrically modifiable, non-volatile storage*

*Principle* of operation:

- Transistor with a second “floating” gate
- Floating gate can trap electrons
- This results in a detectable change in threshold voltage

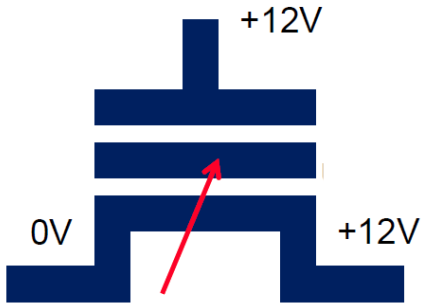



**Erasing to logical “1”**



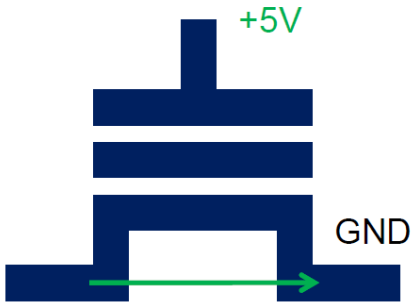

“Quantum tunneling”
Drains charge from FG

**Programming (=writing) to logical “0”**



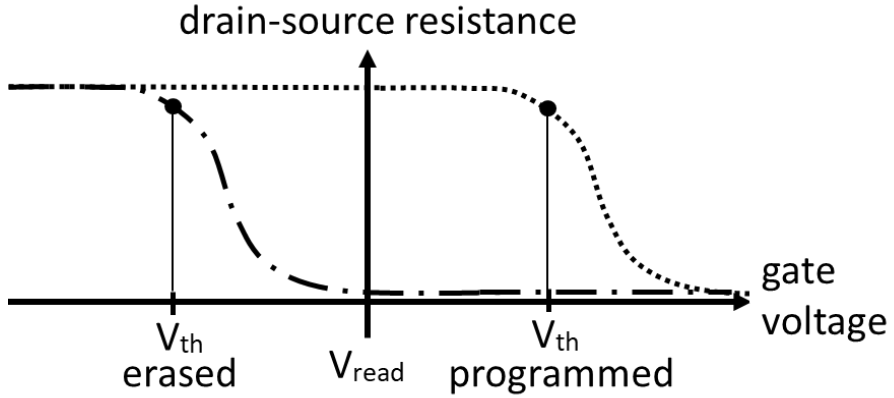

“Hot-electron injection”
traps charge in FG

**Reading**

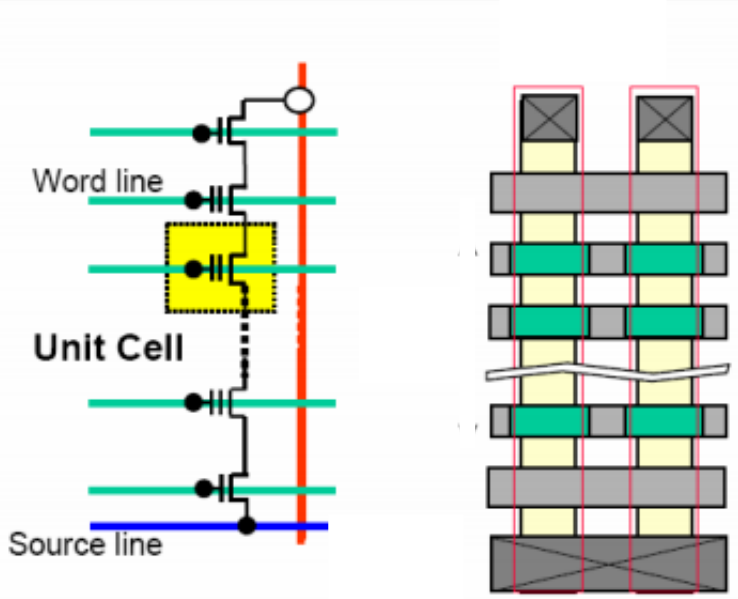
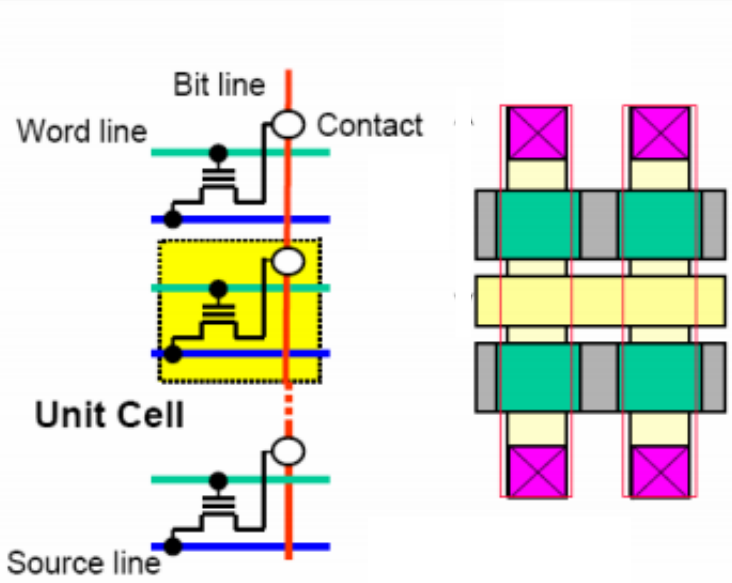



Turn on low Vt or High Vt?

Detect $I_{on}$ to read 0 or 1



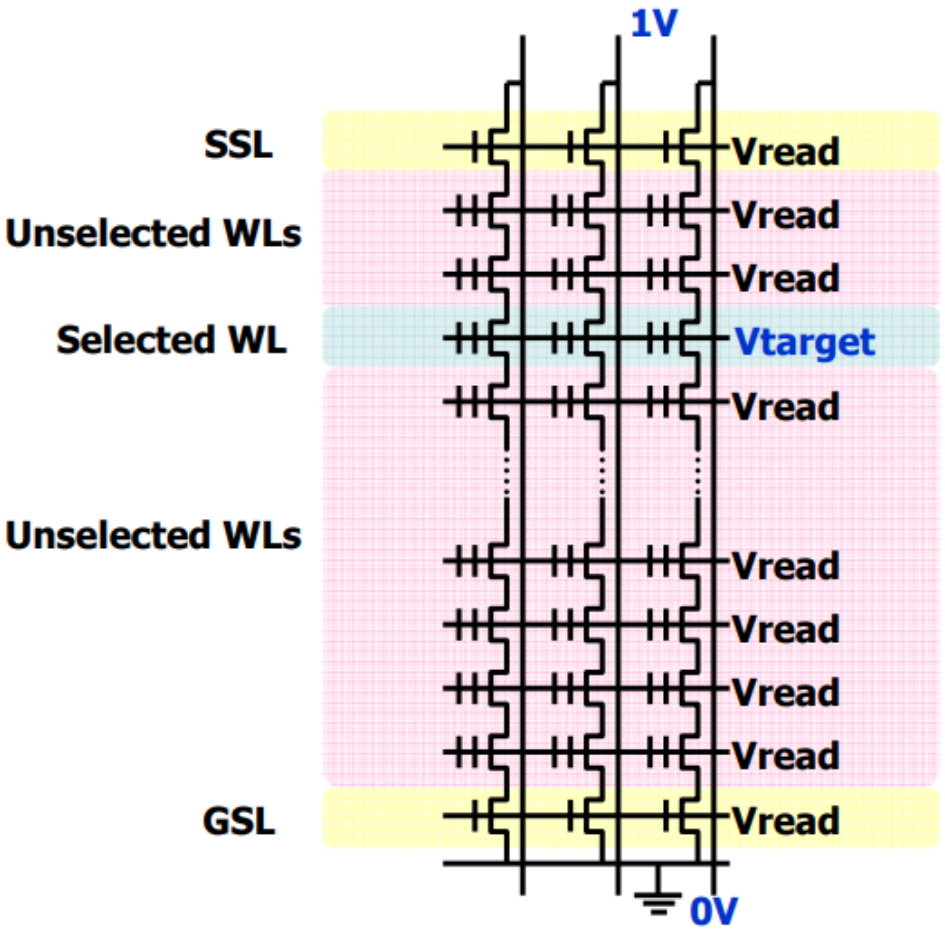
# NAND and NOR Flash Memory

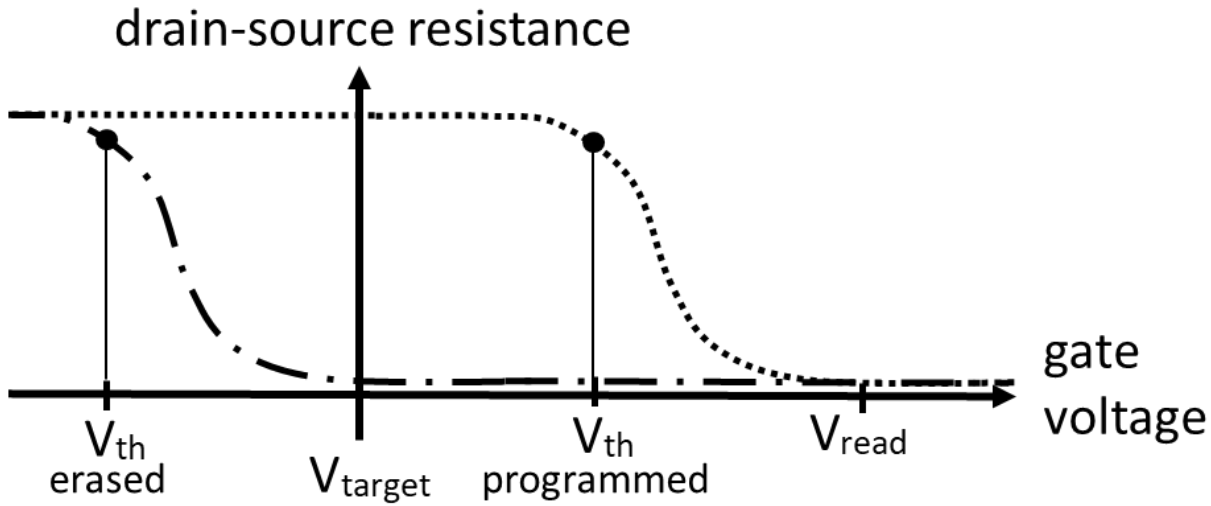
| | NAND | NOR |
|---|---|---|
| Cell Array & Size |  |  |
| Cross-section |  |  |
| Features | Small Cell Size, High Density<br>Low Power<br>➔ *Mass Storage* | **Fast random access**<br>➔ *Code Storage* |

# Example: Reading out NAND Flash

*Selected word-line (WL) :* Target voltage ($V_{target}$)

*Unselected word-lines :* $V_{read}$ is high enough to have a low resistance in all transistors in this row



# Storage

## Memory Map

# Example: Memory Map in MSP432 (ES-Lab)

***Available memory:***

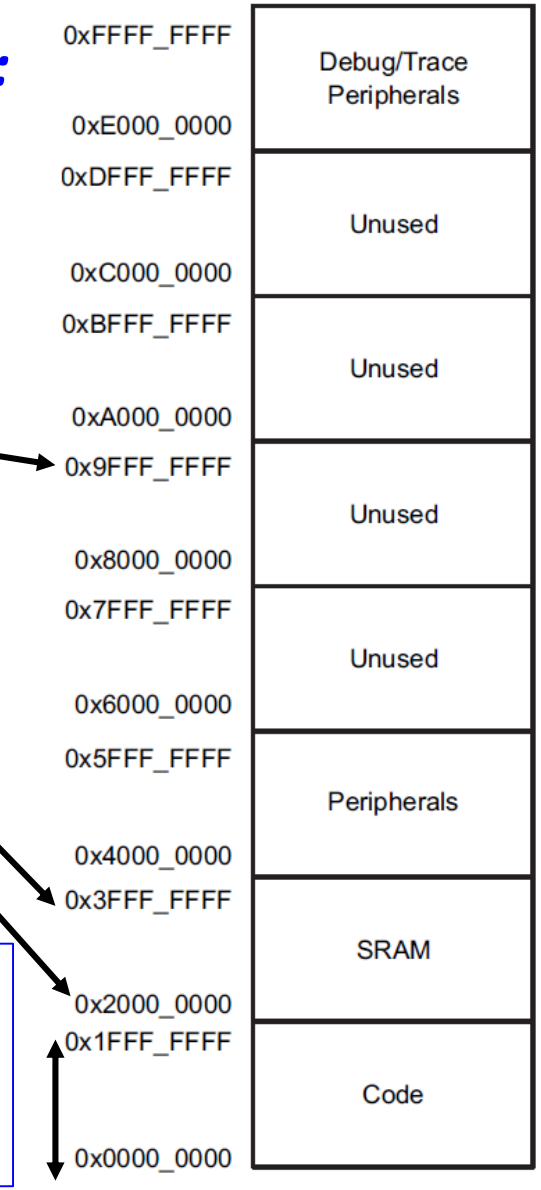
- The processor used in the lab (MSP432P401R) has built in 256kB flash memory, 64kB SRAM and 32kB ROM (Read Only Memory).

***Address space:***

- The processor uses 32 bit addresses. Therefore, the addressable memory space is 4 GByte (= $2^{32}$ Byte) as each memory location corresponds to 1 Byte.
- The address space is used to address the memories (reading and writing), to address the peripheral units, and to have access to debug and trace information (memory mapped microarchitecture).
- The address space is partitioned into zones, each one with a dedicated use. The following is a simplified description to introduce the basic concepts.

# Example: Memory Map in MSP432 (ES-Lab)

***Memory map:***




hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

0011 1111 …. 1111
0010 0000 …. 0000

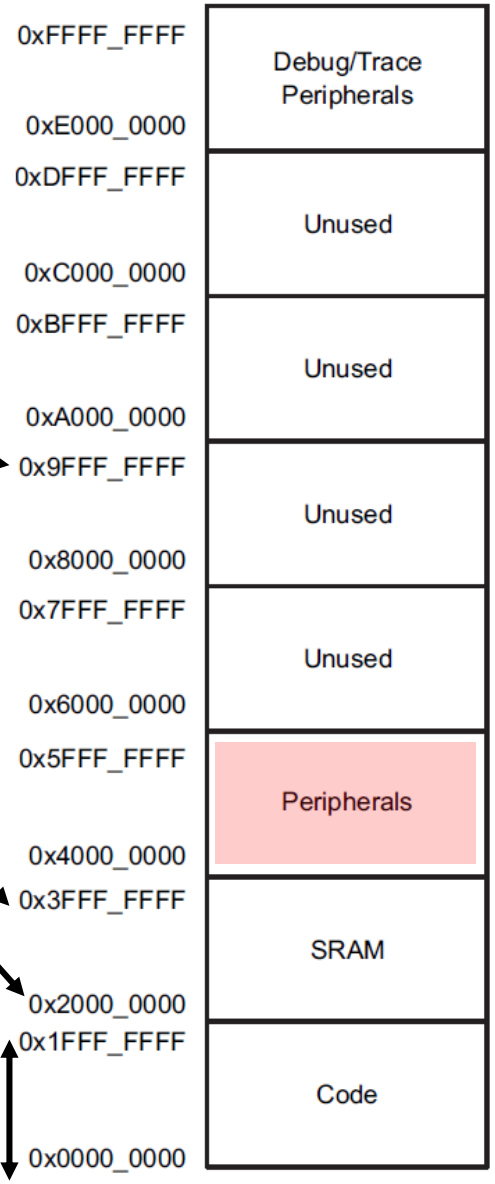
diff. = 0001 1111 …. 1111 → $2^{29}$ different addresses

capacity = $2^{29}$ Byte = 512 MByte

# Example: Memory Map in MSP432 (ES-Lab)

***Memory map:***

hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit




0011 1111 …. 1111

0010 0000 …. 0000

diff. = 0001 1111 …. 1111 → $2^{29}$ different addresses

capacity = $2^{29}$ Byte = 512 MByte

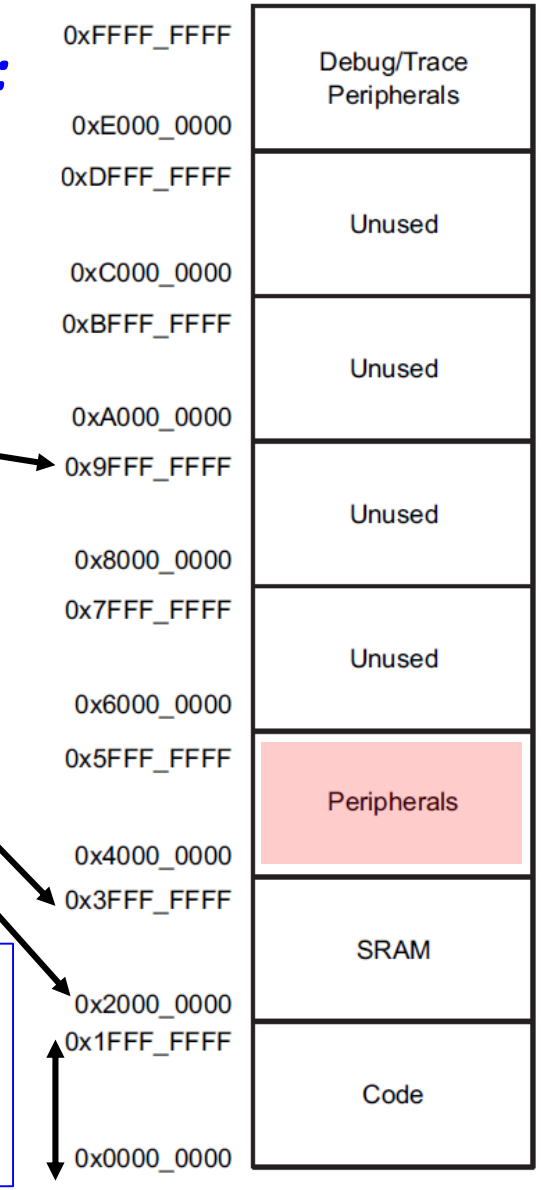
| ADDRESS RANGE | PERIPHERAL |
|---|---|
| 0x4000_0000 to 0x4000_03FF | Timer_A0 |
| 0x4000_0400 to 0x4000_07FF | Timer_A1 |
| 0x4000_0800 to 0x4000_0BFF | Timer_A2 |
| 0x4000_0C00 to 0x4000_0FFF | Timer_A3 |
| 0x4000_1000 to 0x4000_13FF | eUSCI_A0 |
| 0x4000_1400 to 0x4000_17FF | eUSCI_A1 |
| 0x4000_1800 to 0x4000_1BFF | eUSCI_A2 |
| 0x4000_1C00 to 0x4000_1FFF | eUSCI_A3 |
| … | |
| 0x4000_4400 to 0x4000_47FF | RTC_C |
| 0x4000_4800 to 0x4000_4BFF | WDT_A |
| 0x4000_4C00 to 0x4000_4FFF | Port Module |
| … | |

**Table 6-21. Port Registers (Base Address: 0x4000_4C00)**

| REGISTER NAME | ACRONYM | OFFSET from base address |
|---|---|---|
| Port 1 Input | P1IN | 000h |
| Port 2 Input | P2IN | 001h |
| Port 1 Output | P1OUT | 002h |
| Port 2 Output | P2OUT | 003h |

# Example: Memory Map in MSP432 (ES-Lab)

***Memory map:***




hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

0011 1111 …. 1111
0010 0000 …. 0000

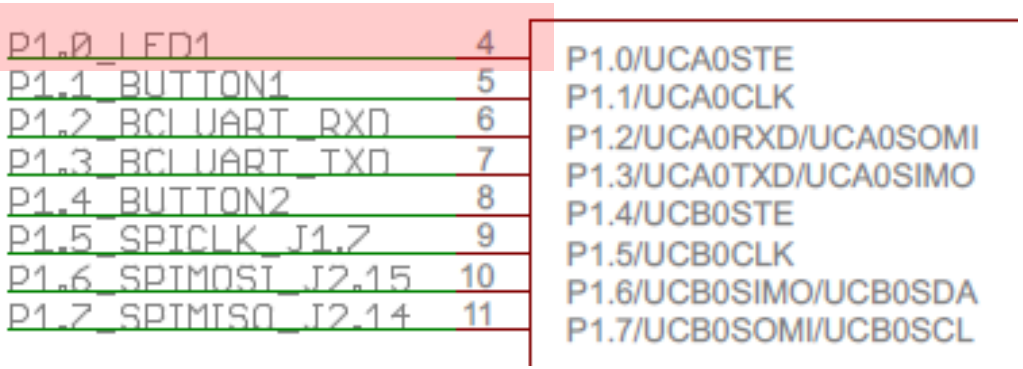
diff. = 0001 1111 …. 1111 → $2^{29}$ different addresses

capacity = $2^{29}$ Byte = 512 MByte

**Table 6-21. Port Registers (Base Address: 0x4000_4C00)**

| REGISTER NAME | ACRONYM | OFFSET |
|---|---|---|
| Port 1 Input | P1IN | 000h |
| Port 2 Input | P2IN | 001h |
| Port 1 Output | P1OUT | 002h |
| Port 2 Output | P2OUT | 003h |

Schematic of LaunchPad as used in the Lab:

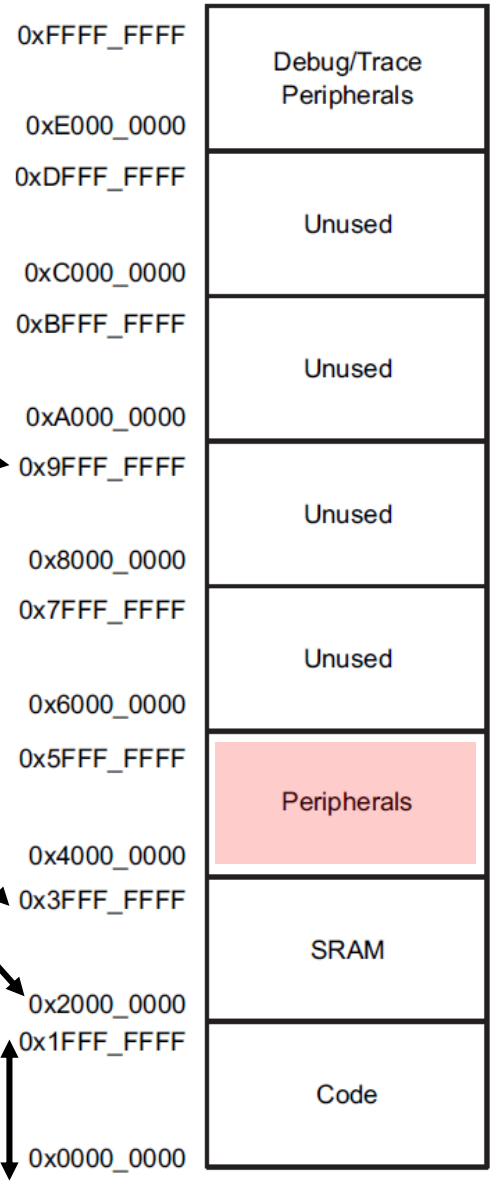



LED1 is connected to Port 1, Pin 0

**How do we toggle LED1 in a C program?**

# Example: Memory Map in MSP432 (ES-Lab)

***Memory map:***




hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

0011 1111 …. 1111

0010 0000 …. 0000

diff. = 0001 1111 …. 1111 → $2^{29}$ different addresses

capacity = $2^{29}$ Byte = 512 MByte

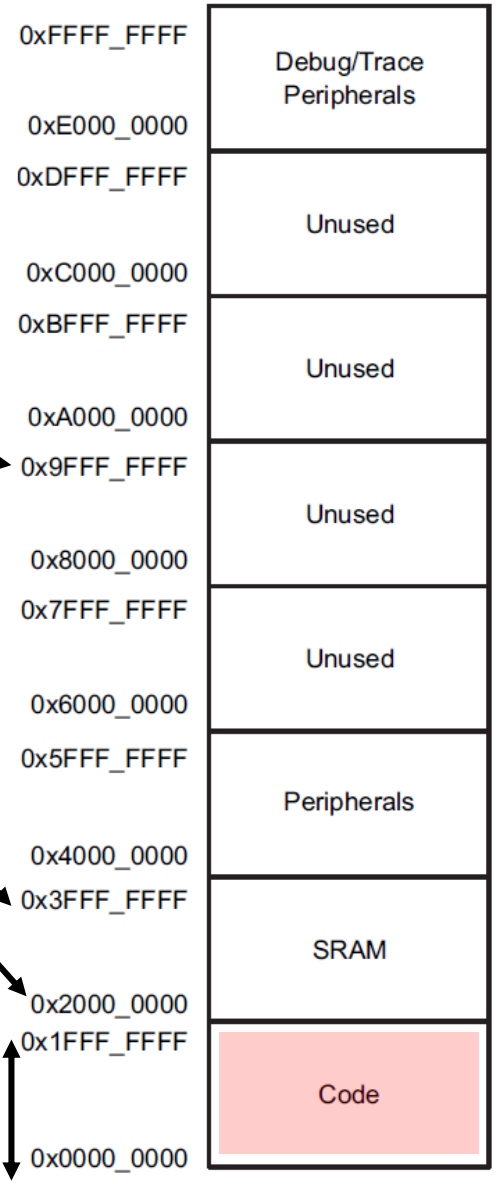
Many necessary elements are missing in the sketch below, in particular the configuration of the port (input or output, pull up or pull down resistors for input, drive strength for output). See lab session.

```
…
//declare p1out as a pointer to an 8Bit integer
volatile uint8_t*  p1out;

//P1OUT should point to Port 1 where LED1 is connected
p1out = (uint8_t*) 0x40004C02;

//Toggle Bit 0 (Signal to which LED1 is connected)
*p1out = *p1out ^ 0x01;
```

^ : XOR

# Example: Memory Map in MSP432 (ES-Lab)

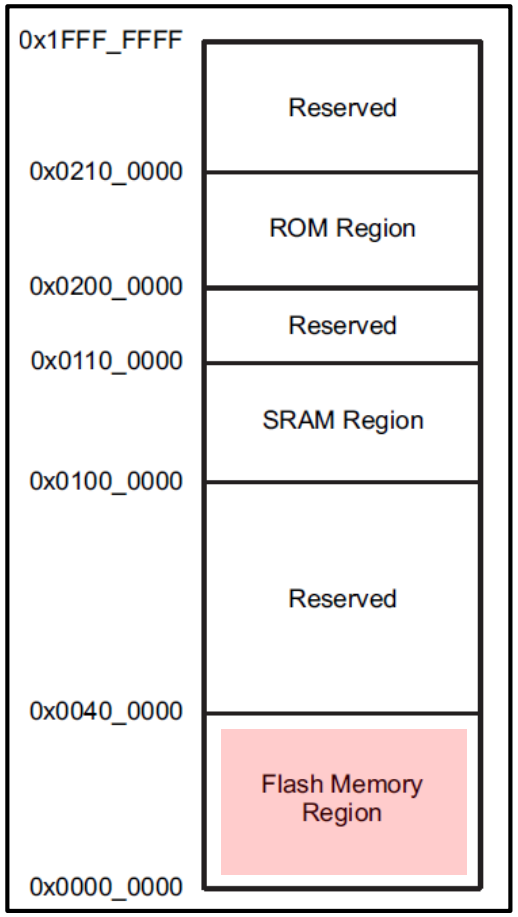
***Memory map:***

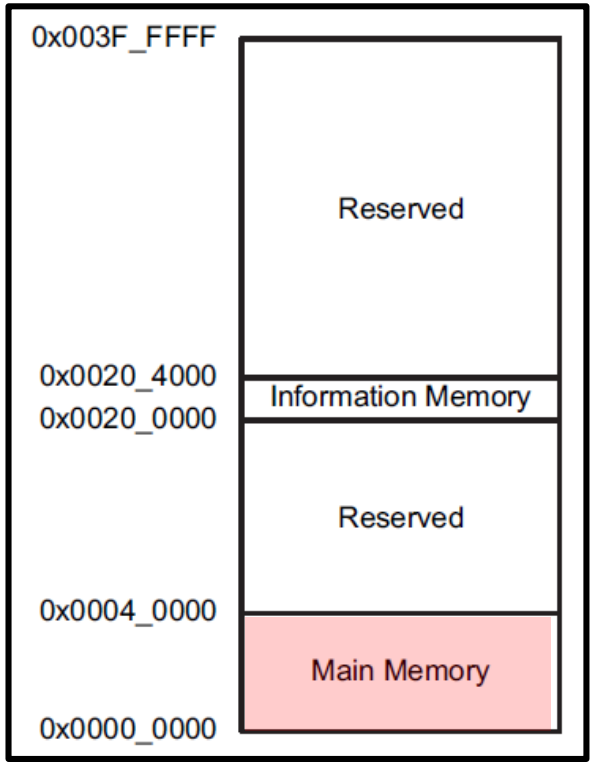



hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

0011 1111 …. 1111
0010 0000 …. 0000

diff. = 0001 1111 …. 1111 → $2^{29}$ different addresses
capacity = $2^{29}$ Byte = 512 MByte



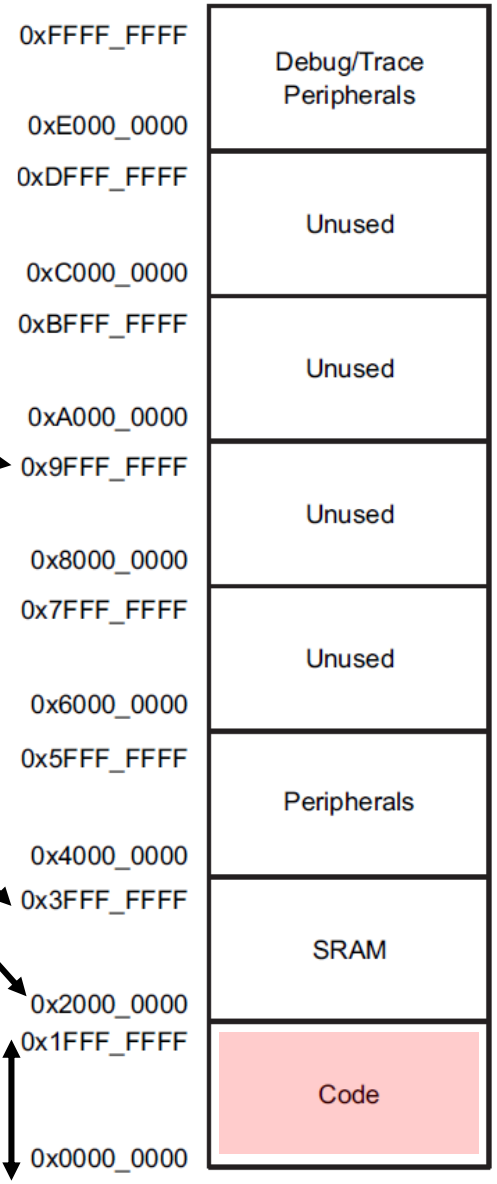




- 0x3FFFF address difference = 4 * $2^{16}$ different addresses → 256 kByte maximal data capacity for Flash Main Memory
- Used for program, data and non-volatile configuration.

# Example: Memory Map in MSP432 (ES-Lab)

***Memory map:***

hexadecimal representation of a 32 bit binary number; each digit corresponds to 4 bit

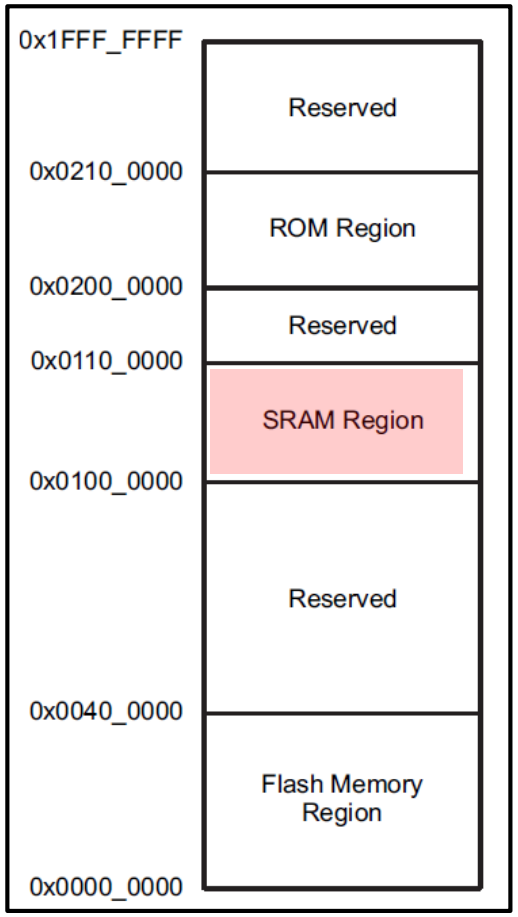


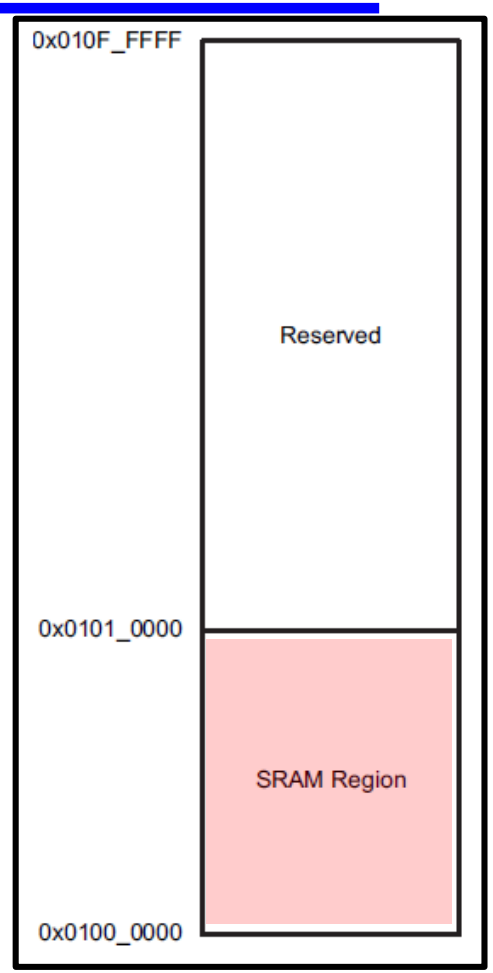

0011 1111 …. 1111

0010 0000 …. 0000

diff. = 0001 1111 …. 1111 → $2^{29}$ different addresses

capacity = $2^{29}$ Byte = 512 MByte




- 0x FFFF address difference = $2^{16}$ different addresses → 64 kByte maximal data capacity for SRAM Region
- Used for program and data.

# Input and Output

# Device Communication

Very often, a processor needs to *exchange information with other processors* or devices. To satisfy various needs, there exists many different *communication protocols*, such as

- ***UART*** (Universal Asynchronous Receiver-Transmitter)
- ***SPI*** (Serial Peripheral Interface Bus)
- ***I2C*** (Inter-Integrated Circuit)
- ***USB*** (Universal Serial Bus)

- As the principles are similar, we will just explain a representative of an asynchronous protocol (*UART,* no shared clock signal between sender and receiver) and one of a synchronous protocol (*SPI* , shared clock signal).

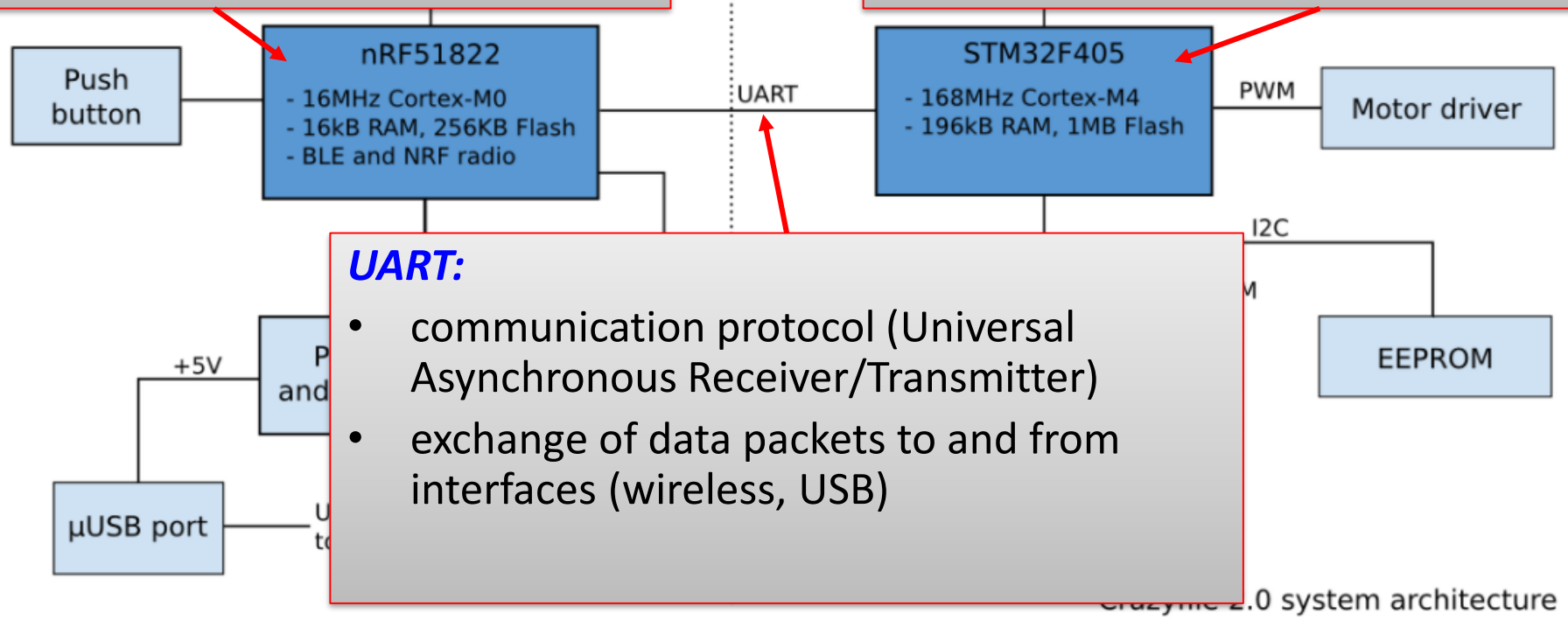
# Remember?

***low power CPU***

- enabling power to the rest of the system
- battery charging and voltage measurement
- wireless radio (boot and operate)
- detect and check expansion boards

***higher performance CPU***

- sensor reading and motor control
- flight control
- telemetry (including the battery voltage)
- additional user development
- USB connection




***UART:***

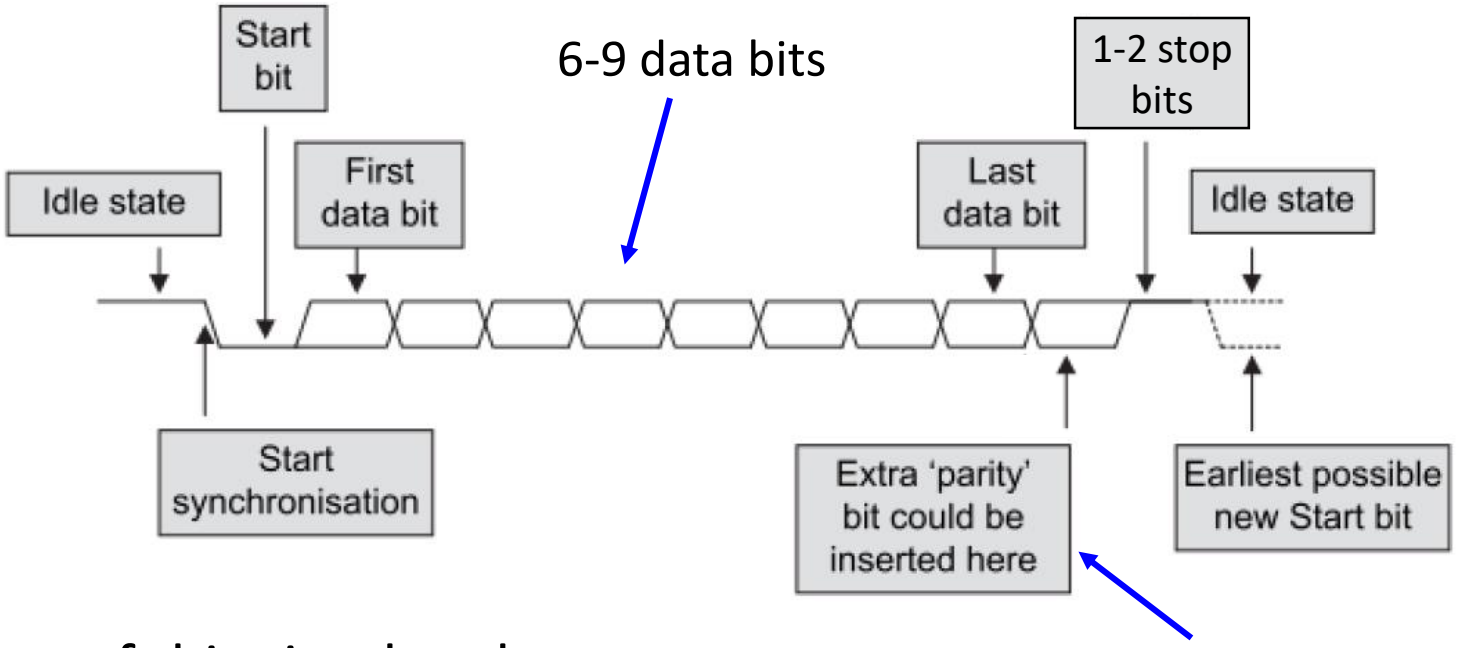
- communication protocol (Universal Asynchronous Receiver/Transmitter)
- exchange of data packets to and from interfaces (wireless, USB)

# Input and Output

## UART Protocol

# UART

- *Serial communication* of bits via a single signal, i.e. UART provides parallel-to-serial and serial-to-parallel conversion.
- Sender and receiver need to *agree on the transmission rate*.
- Transmission of a serial packet starts with a start bit, followed by data bits and finalized using a stop bit:

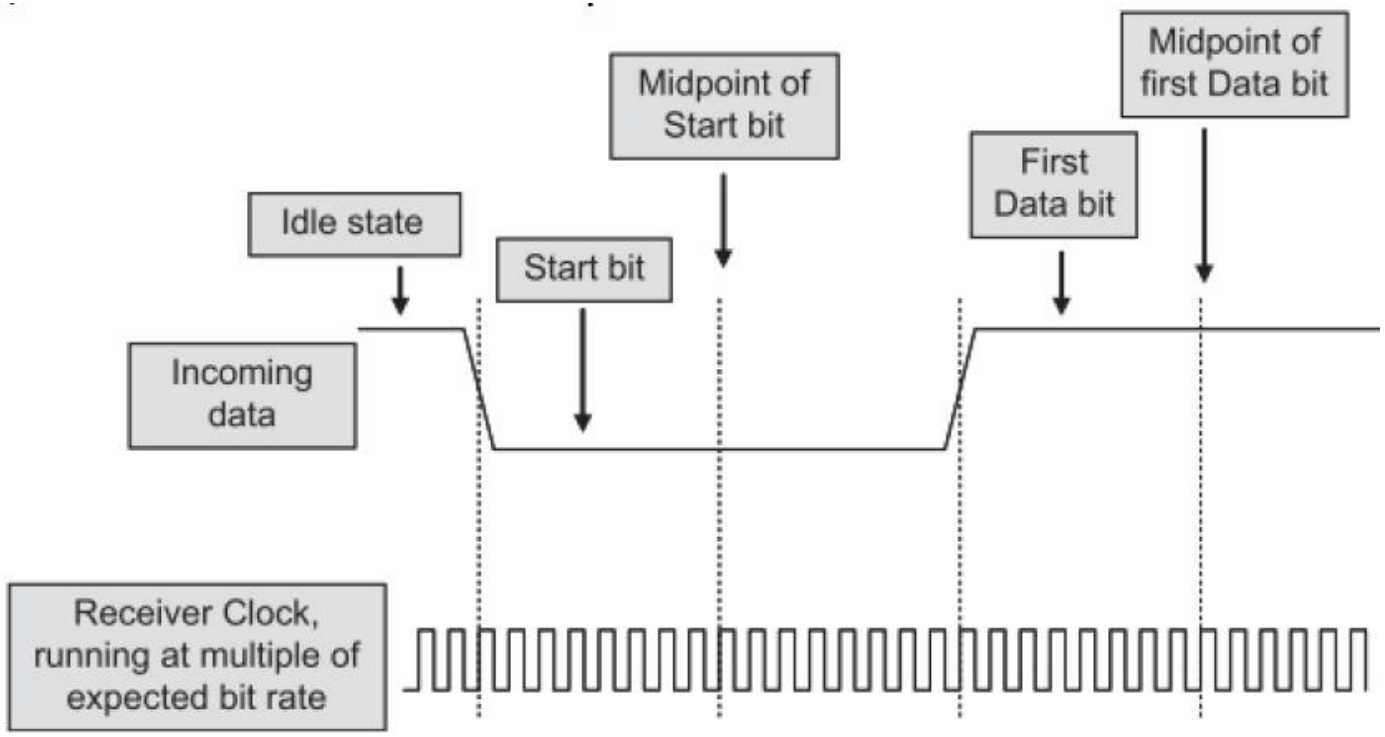



- There exist many variations of this simple scheme.

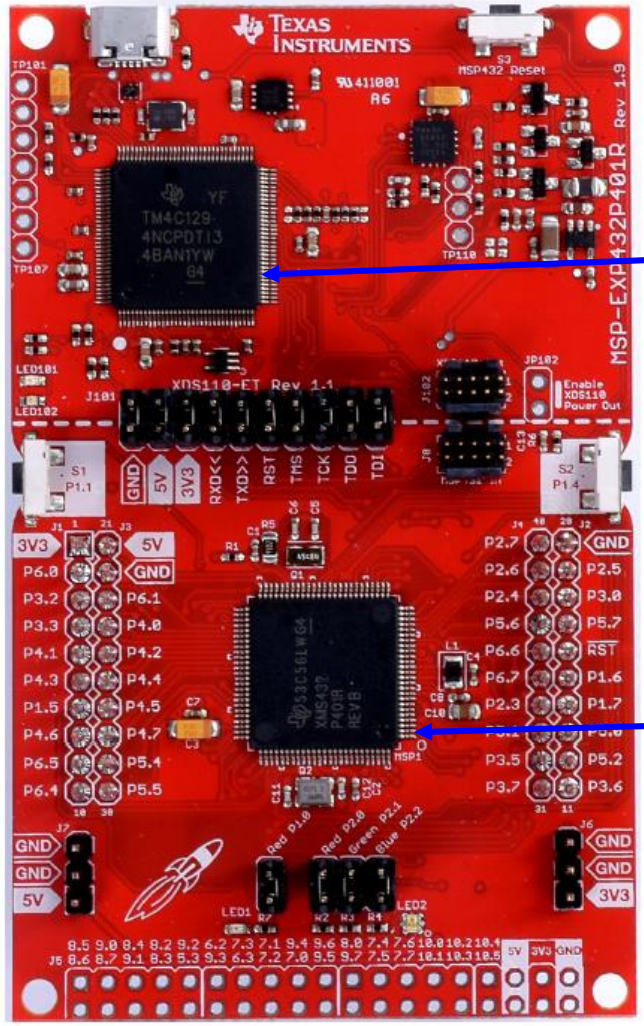
for detecting single bit errors

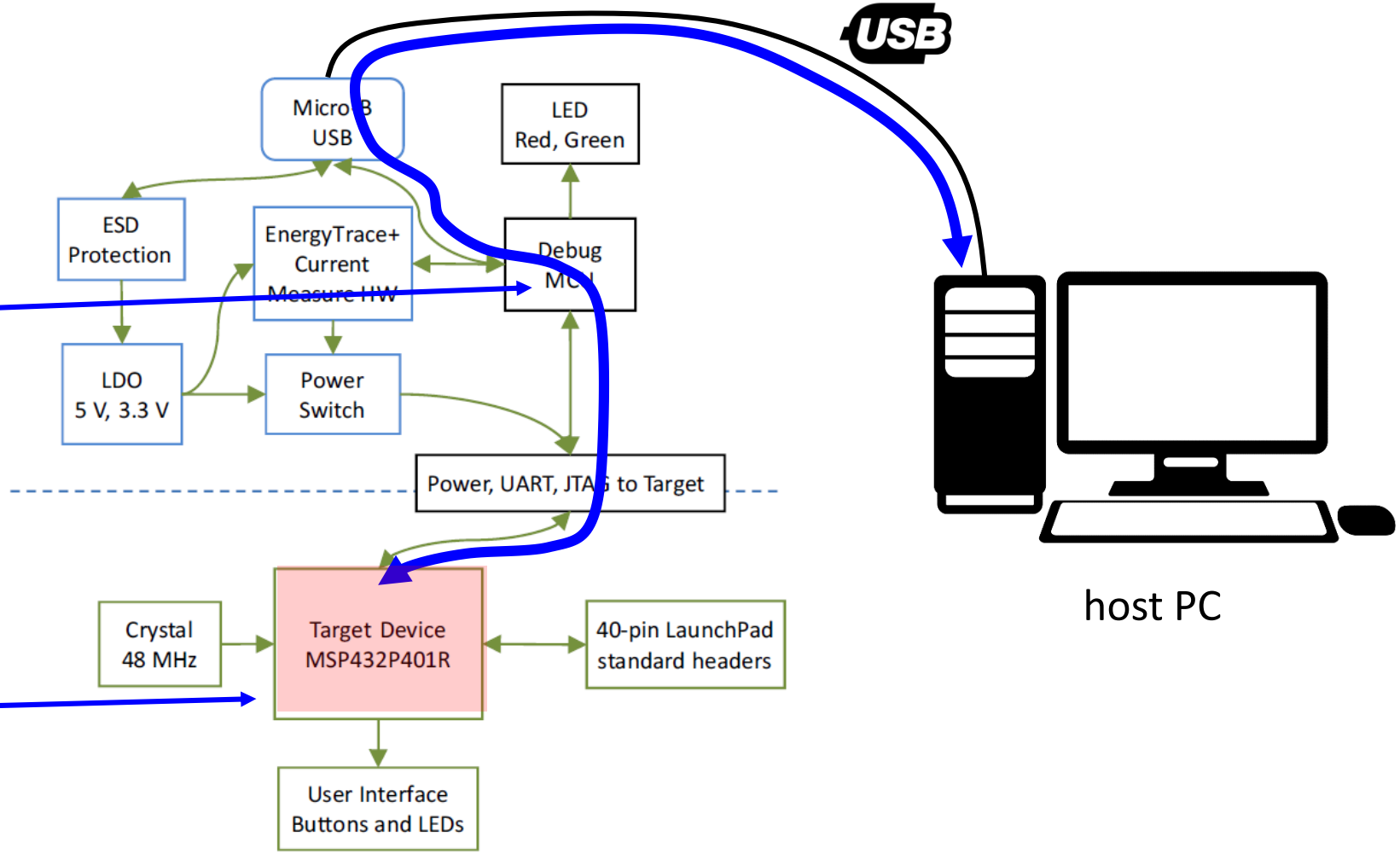
# UART

- The receiver runs an *internal clock* whose frequency is an exact multiple of the expected bit rate.
- When a *Start bit* is detected, a counter begins to count clock cycles e.g. 8 cycles until the midpoint of the anticipated Start bit is reached.
- The clock counter counts a further 16 cycles, to the middle of the first *Data bit*, and so on until the *Stop bit*.

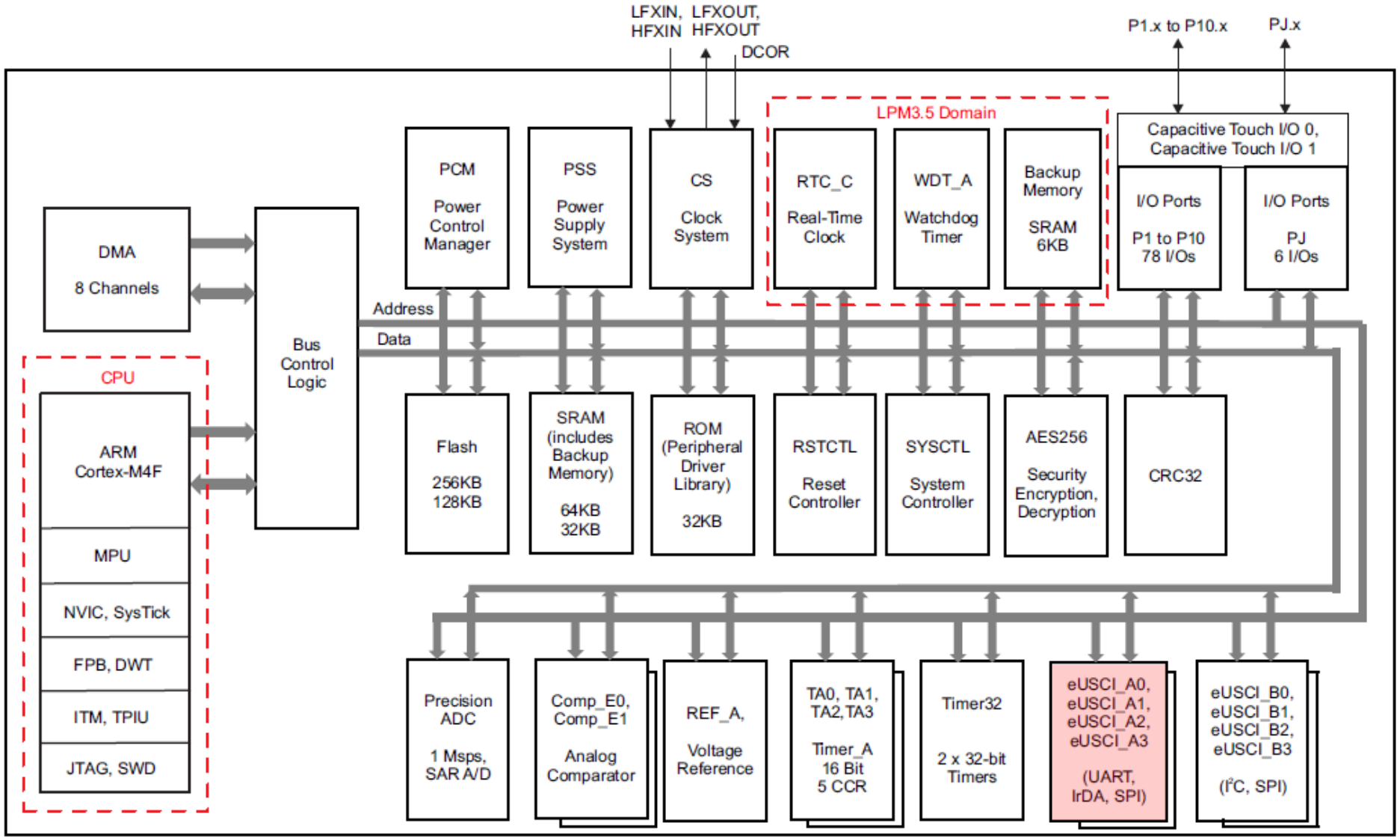


# UART with MSP432 (ES-Lab)



# UART with MSP432 (Lab)



# Input and Output

## Memory Mapped Device Access

# Memory-Mapped Device Access

**eUSCI_A0 Registers (Base Address: 0x4000_1000)**

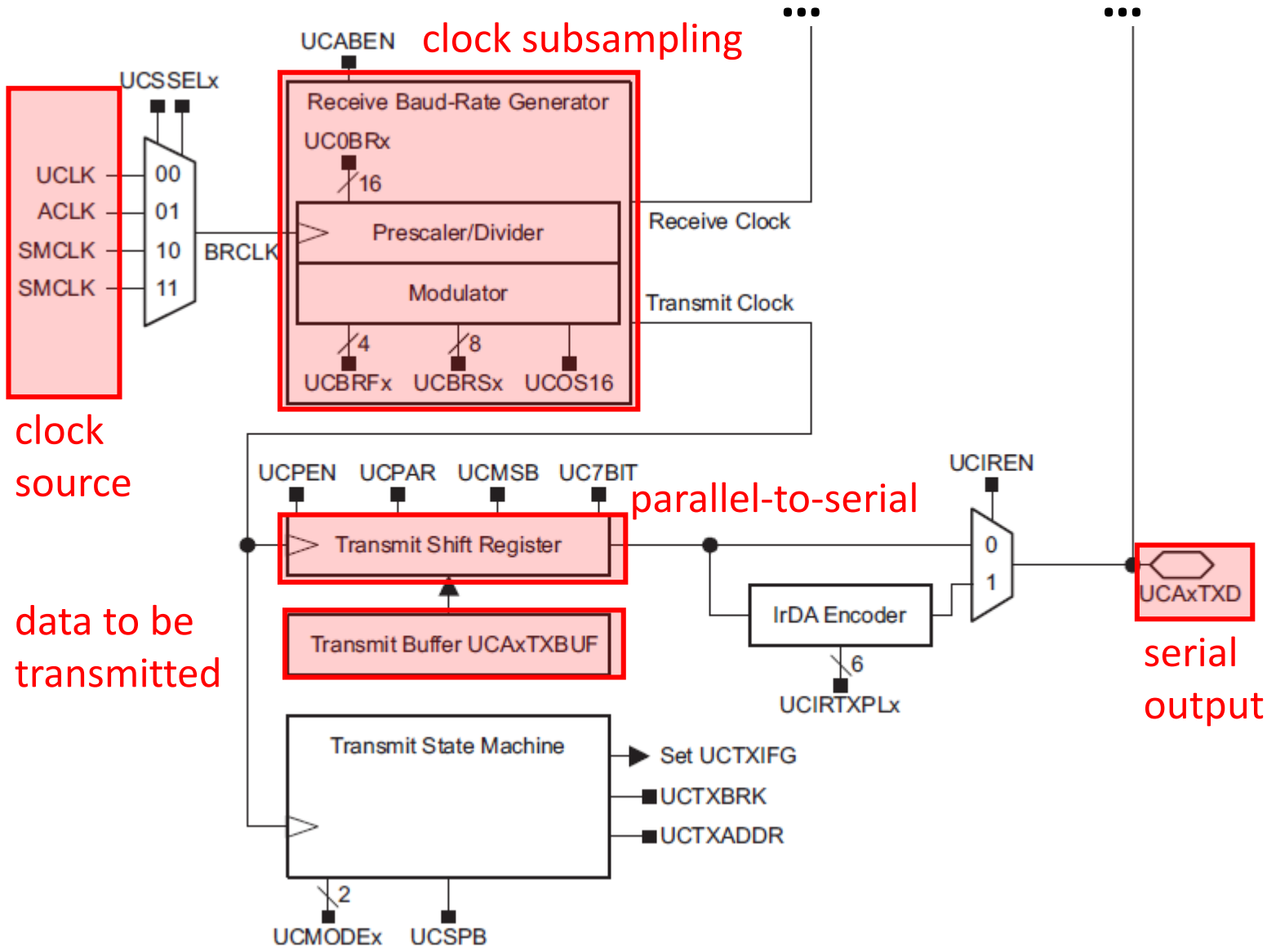
| REGISTER NAME | OFFSET |
|---|---|
| eUSCI_A0 Control Word 0 | 00h |
| eUSCI_A0 Control Word 1 | 02h |
| eUSCI_A0 Baud Rate Control | 06h |
| eUSCI_A0 Modulation Control | 08h |
| eUSCI_A0 Status | 0Ah |
| eUSCI_A0 Receive Buffer | 0Ch |
| eUSCI_A0 Transmit Buffer | 0Eh |
| eUSCI_A0 Auto Baud Rate Control | 10h |
| eUSCI_A0 IrDA Control | 12h |
| eUSCI_A0 Interrupt Enable | 1Ah |
| eUSCI_A0 Interrupt Flag | 1Ch |
| eUSCI_A0 Interrupt Vector | 1Eh |

- *Configuration of Transmitter and Receiver must match*; otherwise, they can not communicate.
- Examples of configuration parameters:
  - transmission rate (baud rate, i.e., symbols/s)
  - LSB or MSB first
  - number of bits per packet
  - parity bit
  - number of stop bits
  - interrupt-based communication
  - clock source

in our case: bit/s

buffer for received bits and bits that should be transmitted

# Transmission Rate




***Clock subsampling:***

- The clock subsampling block is complex, as one tries to match a large set of transmission rates with a fixed input frequency.

***Clock Source:***

- SMCLK in the lab setup = 3MHz
- Quartz frequency = 48 MHz, is divided by 16 before connected to SMCLK

***Example:***

- Transmission rate 4800 bit/s
- 16 clock periods per bit (see 3-26)
- Subsampling factor = 3*10^6 / (4.8*10^3 * 16) = 39.0625

# Software Interface

Part of C program that *prints a character to a UART* terminal on the host PC:

```
...
static const eUSCI_UART_Config uartConfig =
{
  EUSCI_A_UART_CLOCKSOURCE_SMCLK,                  // SMCLK Clock Source
  39,                                              // BRDIV  = 39 , integral part
  1,                                               // UCxBRF =  1 , fractional part * 16
  0,                                               // UCxBRS =  0
  EUSCI_A_UART_NO_PARITY,                          // No Parity
  EUSCI_A_UART_LSB_FIRST,                          // LSB First
  EUSCI_A_UART_ONE_STOP_BIT,                       // One stop bit
  EUSCI_A_UART_MODE,                               // UART mode
  EUSCI_A_UART_OVERSAMPLING_BAUDRATE_GENERATION}; // Oversampling Mode
GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P1,
          GPIO_PIN2 | GPIO_PIN3, GPIO_PRIMARY_MODULE_FUNCTION );     //Configure CPU signals
UART_initModule(EUSCI_A0_BASE, &uartConfig);                 // Configuring UART Module A0
UART_enableModule(EUSCI_A0_BASE);                            // Enable UART module A0

UART_transmitData(EUSCI_A0_BASE,'a');                        // Write character 'a' to UART

...
```

data structure uartConfig contains the configuration of the UART

use uartConfig to write to eUSCI_A0 configuration registers

start UART

base address of A0 (0x40001000), where A0 is the instance of the UART peripheral

# Software Interface

Replacing UART_transmitData(EUSCI_A0_BASE,'a') by a *direct access to registers*:

```
...
volatile uint16_t*  uca0ifg = (uint16_t*) 0x4000101C;
volatile uint16_t*  uca0txbuf = (uint16_t*) 0x4000100E;
...
// Initialization of UART as before
...
while (!((*uca0ifg >> 1) & 0x0001));
*uca0txbuf = (char) 'g'; // Write to transmit buffer
...
```

declare pointers to UART configuration registers

wait until transmit buffer is empty

write character ‘g’ to the transmit buffer

**Table 22-18. UCAxIFG Register Description**

| Bit | Field | Type | Reset | Description |
|---|---|---|---|---|
| 15-4 | Reserved | R | 0h | Reserved |
| 1 | UCTXIFG | RW | 1h | Transmit interrupt flag. UCTXIFG is set when UCAxTXBUF empty.<br>0b = No interrupt pending<br>1b = Interrupt pending |

shift 1 bit to the right

```
!((*uca0ifg >> 1) & 0x0001)
```

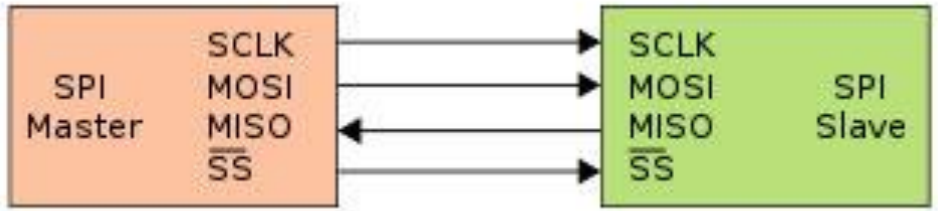
expression is ‘1’ if bit UCTXIFG = 0 (buffer not empty).

# Input and Output

## SPI Protocol

# SPI (Serial Peripheral Interface Bus)

- Typically *communicate across short distances*
- *Characteristics:*
  - 4-wire synchronized (clocked) communications bus
  - supports single master and multiple slaves
  - always full-duplex: Communicates in both directions simultaneously
  - multiple Mbps transmission speeds can be achieved
  - transfer data in 4 to 16 bit serial packets

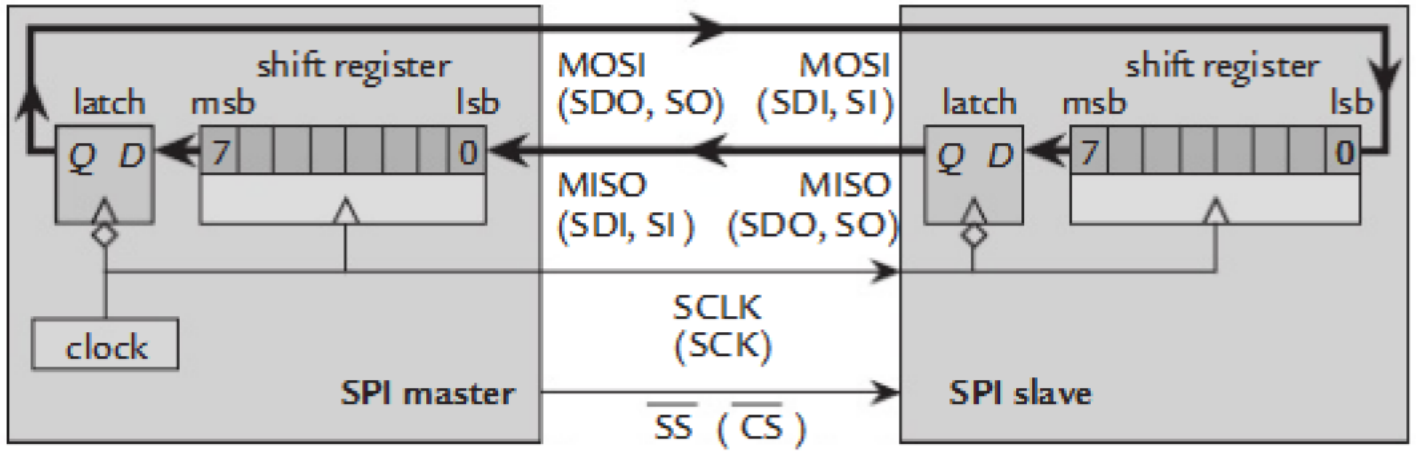



- *Bus wiring:*
  - MOSI (Master Out Slave In) – carries data out of master to slave
  - MISO (Master In Slave Out) – carries data out of slave to master
  - Both MOSI and MISO are active during every transmission
  - $\overline{SS}$ (or CS) – signal to select each slave chip
  - System clock SCLK – produced by master to synchronize transfers

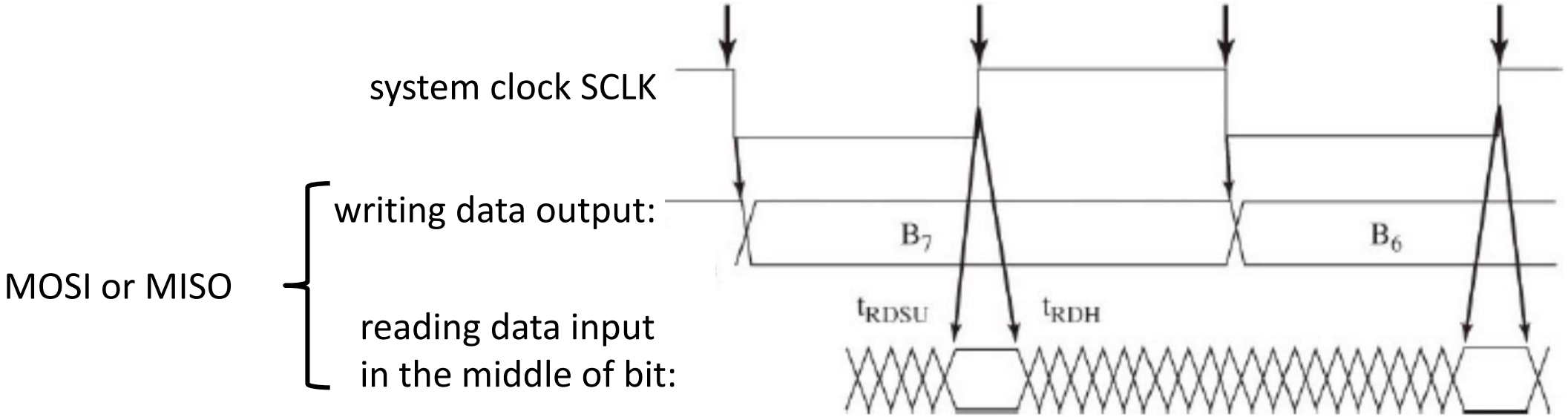
# SPI (Serial Peripheral Interface Bus)

*More detailed circuit diagram:*

- details vary between different vendors and implementations

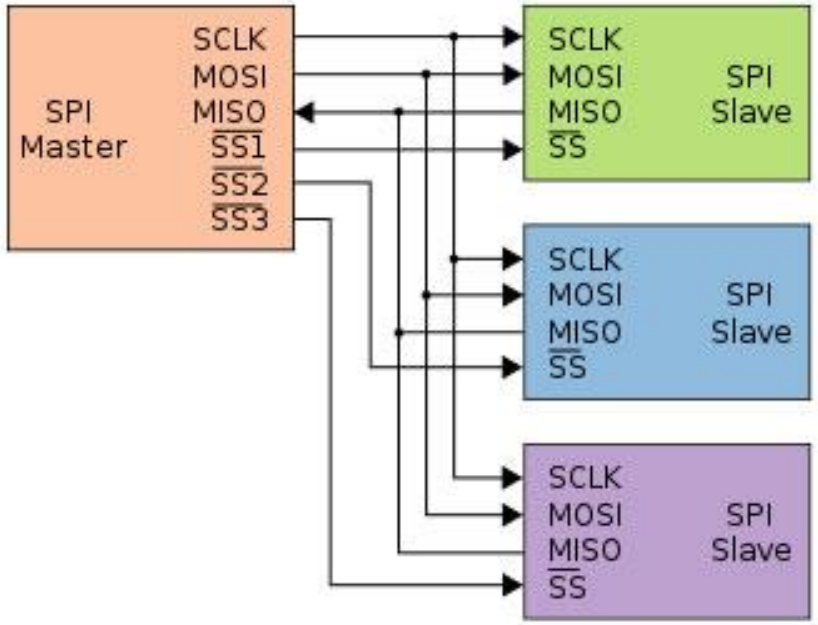



*Timing diagram:*



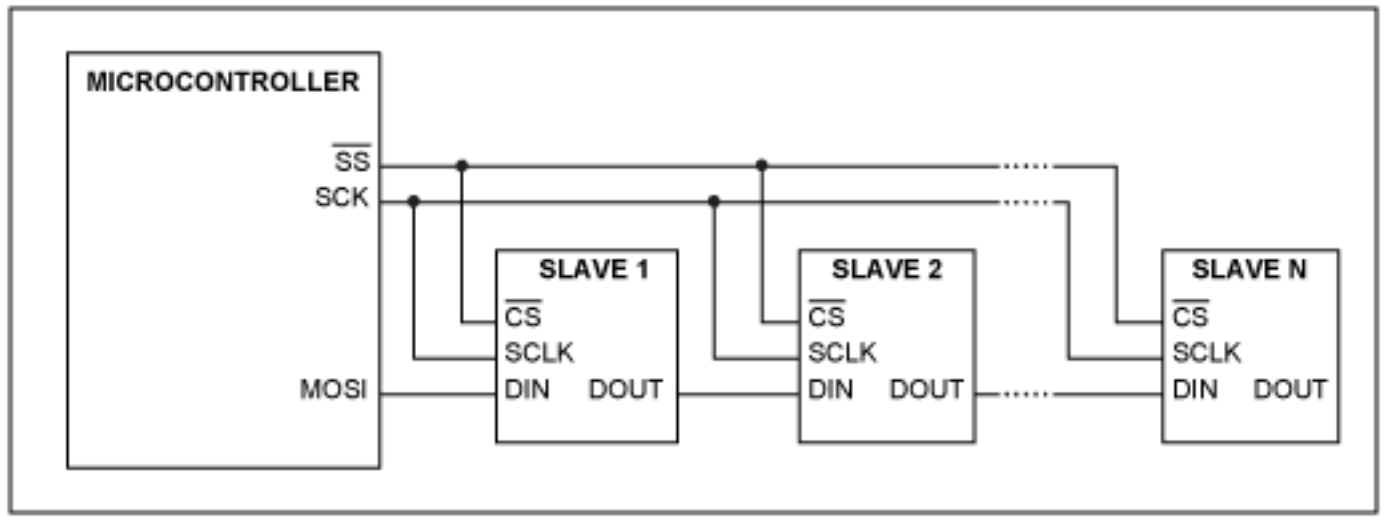
# SPI (Serial Peripheral Interface Bus)

*Two examples of bus configurations:*




Master and multiple independent slaves

http://upload.wikimedia.org/wikipedia/commons/thumb/f/fc/SPI_three_slaves.svg/350px-SPI_three_slaves.svg.png




Master and multiple daisy-chained slaves

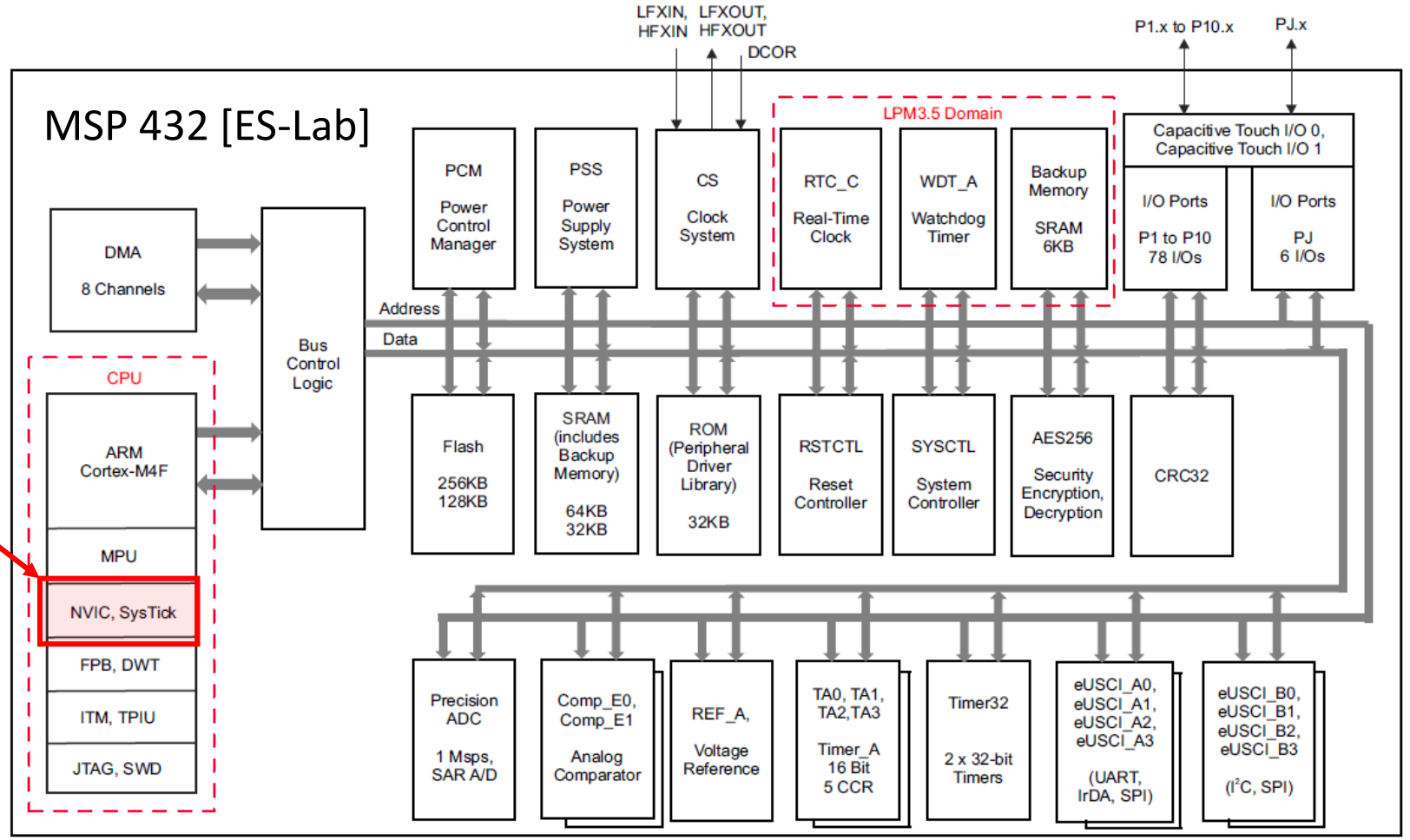
http://www.maxim-ic.com/appnotes.cfm/an_pk/3947

# Interrupts

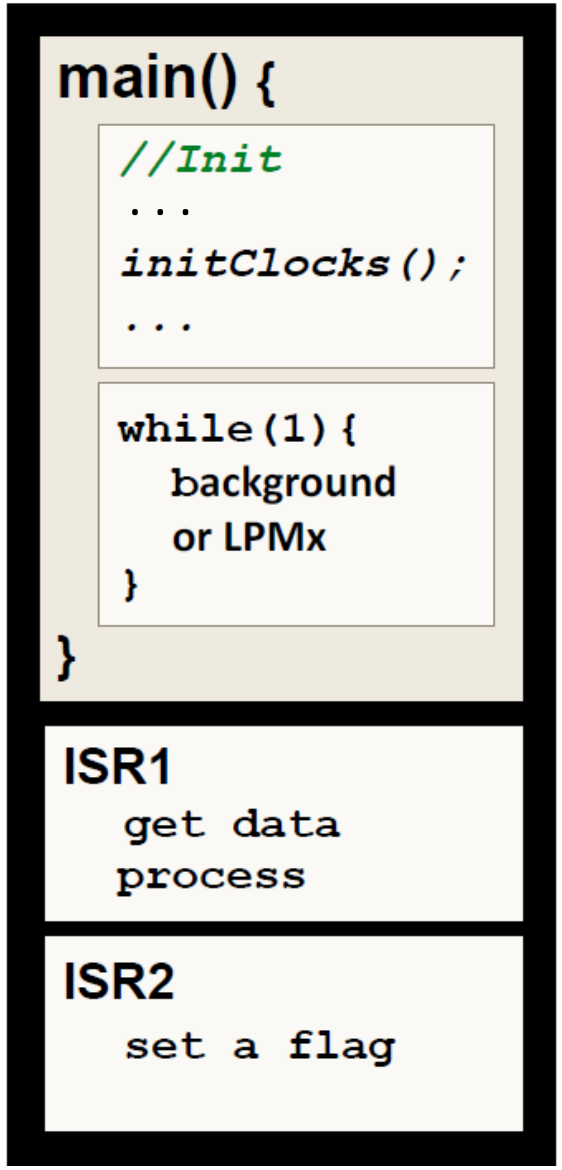
# Interrupts

*A hardware interrupt is an electronic alerting signal sent to the CPU from another component, either from an internal peripheral or from an external device.*

The *Nested Vector Interrupt Controller* (NVIC) handles the processing of interrupts



# Interrupts




**System Initialization**

- The beginning part of main() is usually dedicated to setting up your system

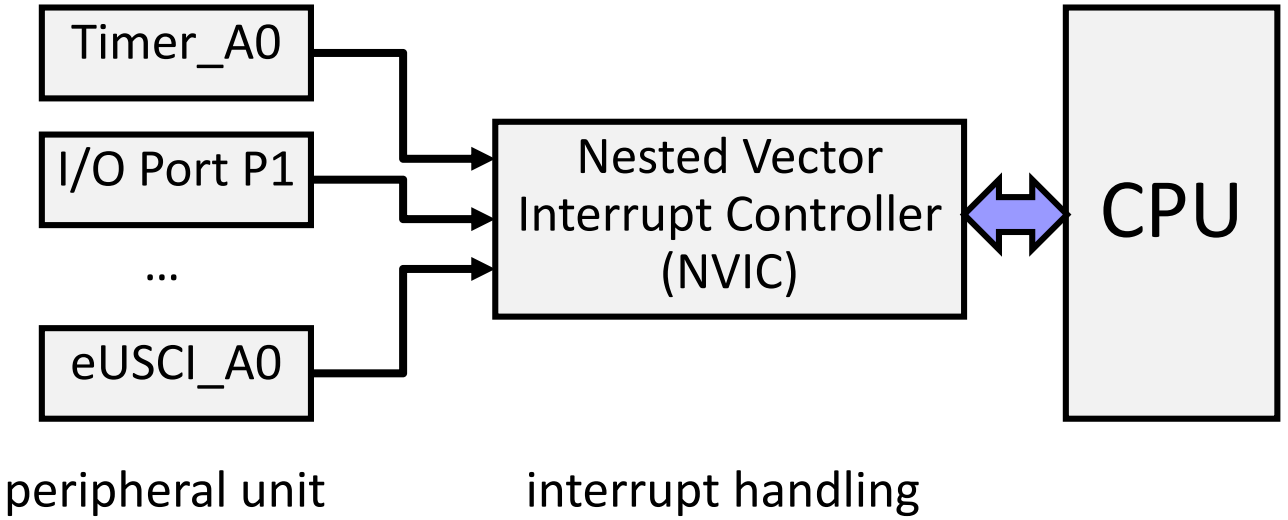
**Background**

- Most systems have an endless loop that runs 'forever' in the background
- In this case, 'Background' implies that it runs at a lower priority than 'Foreground'
- In MSP432 systems, the background loop often contains a Low Power Mode (LPMx) command – this sleeps the CPU/System until an interrupt event wakes it up

**Foreground**

- Interrupt Service Routine (ISR) runs in response to enabled hardware interrupt
- These events may change modes in Background – such as waking the CPU out of low-power mode
- ISR's, by default, are not interruptible
- Some processing may be done in ISR, but it's usually best to keep them short

# Processing of an Interrupt (MSP432 ES-Lab)



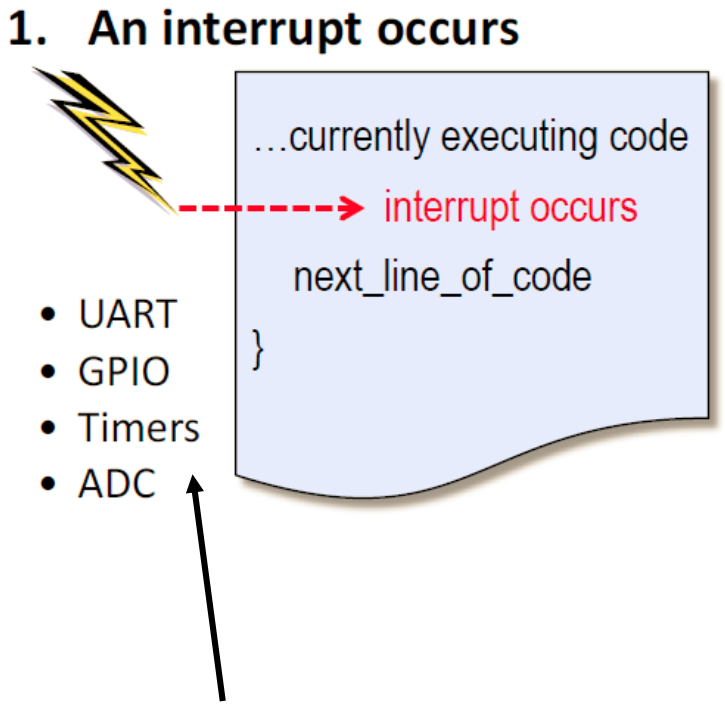

The *vector interrupt controller (NVIC)*

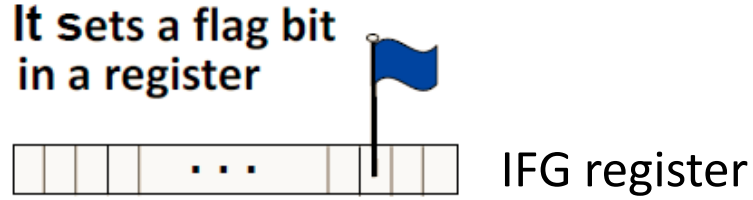
- enables and disables interrupts
- allows to individually and globally *mask interrupts* (disable reaction to interrupt), and
- registers *interrupt service routines* (ISR), sets the priority of interrupts.

*Interrupt priorities* are relevant if

- several interrupts happen at the same time
- the programmer does not mask interrupts in an interrupt service routine (ISR) and therefore, *preemption of an ISR* by another ISR may happen (interrupt nesting).

# Processing of an Interrupt



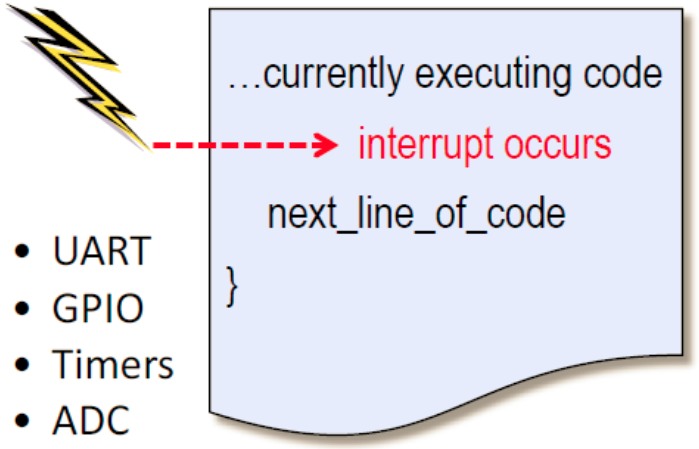




- Most peripherals can generate interrupts to provide status and information.
- Interrupts can also be generated from GPIO pins.

- When an interrupt signal is received, a corresponding bit is set in an IFG register.
- There is an such an IFG register for each interrupt source.
- As some interrupt sources are only on for a short duration, the CPU registers the interrupt signal internally.

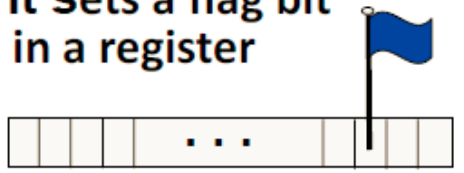
# Processing of an Interrupt

**1. An interrupt occurs**




- UART
- GPIO
- Timers
- ADC
- Etc.

**2. It sets a flag bit in a register**



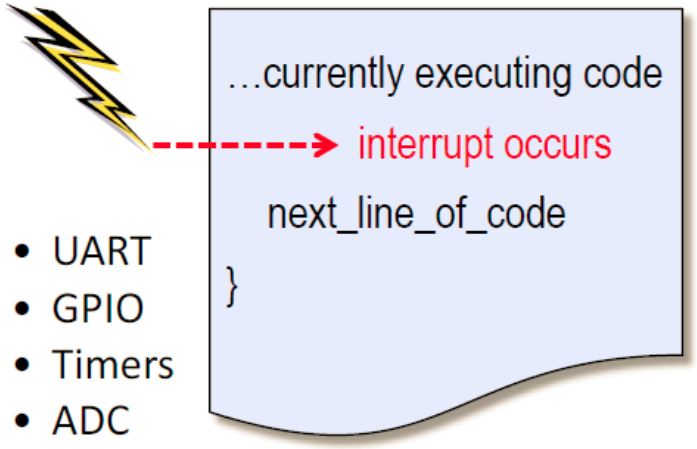
IFG register

**3. CPU/NVIC acknowledges interrupt by:**

- current instruction completes
- saves return-to location on stack
- mask interrupts globally
- determines source of interrupt
- calls interrupt service routine (ISR)

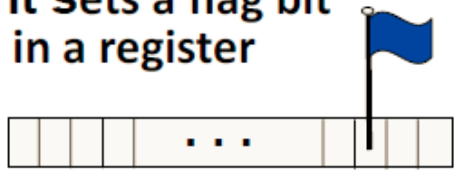
# Processing of an Interrupt

**1. An interrupt occurs**




- UART
- GPIO
- Timers
- ADC
- Etc.

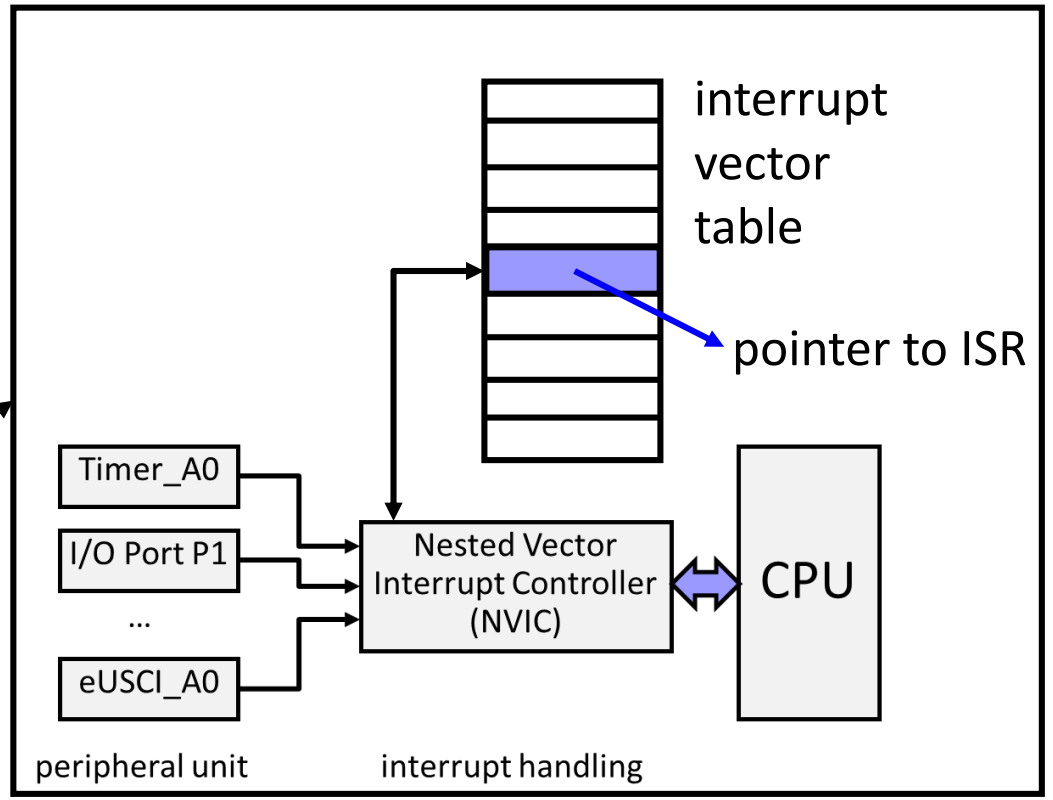
**2. It sets a flag bit in a register**



IFG register

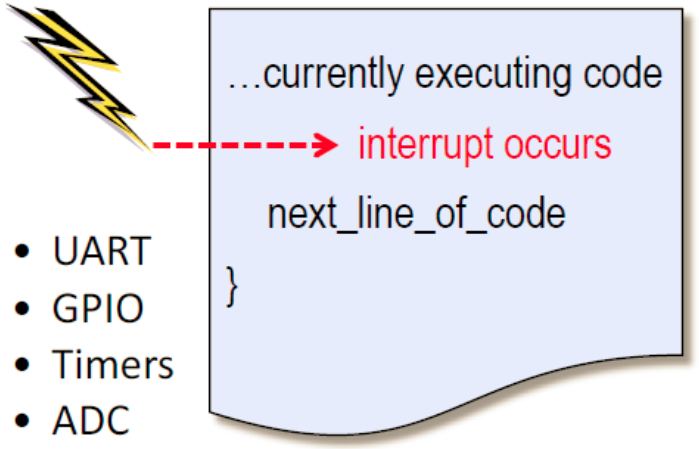
**3. CPU/NVIC acknowledges interrupt by:**

- current instruction completes
- saves return-to location on stack
- mask interrupts globally
- determines source of interrupt
- calls interrupt service routine (ISR)



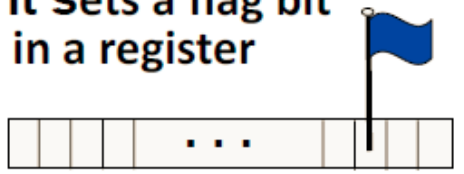
# Processing of an Interrupt

**1. An interrupt occurs**




- UART
- GPIO
- Timers
- ADC
- Etc.

**2. It sets a flag bit in a register**



IFG register

**3. CPU/NVIC acknowledges interrupt by:**

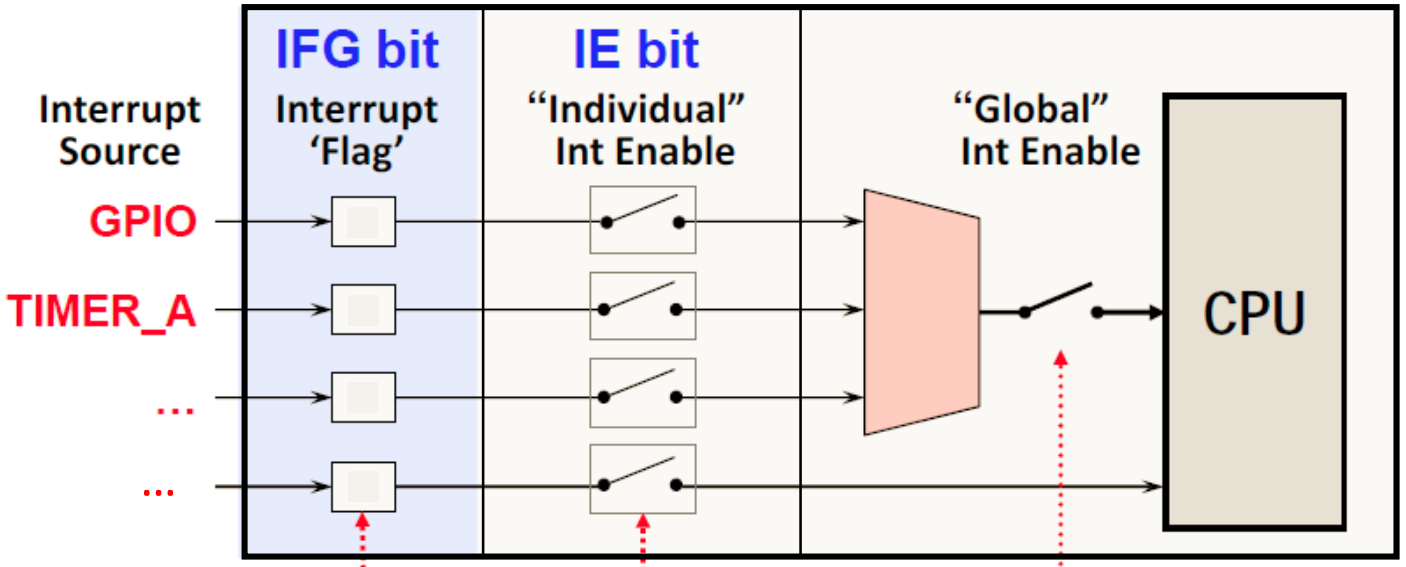
- current instruction completes
- saves return-to location on stack
- mask interrupts globally
- determines source of interrupt
- calls interrupt service routine (ISR)

**4. Interrupt Service Routine (ISR):**

- save context of system
- **run your interrupt's code**
- restore context of system
- (automatically) un-mask interrupts and
- continue where it left off

# Processing of an Interrupt

***Detailed interrupt processing flow:***




get the interrupt status of the selected pin

clears the interrupt status on the selected pin

**Interrupt Flag Reg (IFG) bit set when int occurs; e.g.**
GPIO_getInterruptStatus();
GPIO_clearInterruptFlag();

enable interrupt in the peripheral unit

**Interrupt Enable (IE); e.g.**
GPIO_enableInterrupt();
GPIO_disableInterrupt();

enable interrupt in the interrupt controller

Interrupt_enableInterrupt();

**Global Interrupt Enable**
**Enables ALL maskable interrupts**

Interrupt_enableMaster();
Interrupt_disableMaster();

globally allow / dis-allow the processor to react to interrupts

# Example: Interrupt Processing

- *Port 1, pin 1* (which has a switch connected to it) is configured as an *input* with interrupts enabled and *port 1, pin 0* (which has an LED connected) is configured as an *output*.
- When the *switch is pressed*, the *LED output is toggled*.

clear interrupt flag and enable interrupt in periphery

enable interrupts in the controller (NVIC)

enter low power mode LPM3

```
int main(void)
{
    ...
    GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);
    GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);

    GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN1);
    GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN1);

    Interrupt_enableInterrupt(INT_PORT1);
    Interrupt_enableMaster();

    while (1) PCM_gotoLPM3();
}
```

# Example: Interrupt Processing

- *Port 1, pin 1* (which has a switch connected to it) is configured as an *input* with interrupts enabled and *port 1, pin 0* (which has an LED connected) is configured as an *output*.
- When the *switch is pressed*, the *LED output is toggled*.

predefined name of ISR attached to Port 1

get status (flags) of interrupt-enabled pins of port 1

clear all current flags from all interrupt-enabled pins of port 1

check, whether pin 1 was flagged

```
void PORT1_IRQHandler(void)
{
    uint32_t status;
    status = GPIO_getEnabledInterruptStatus(GPIO_PORT_P1);
    GPIO_clearInterruptFlag(GPIO_PORT_P1, status);

    if(status & GPIO_PIN1)
    {
        GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);
    }
}
```

# Polling vs. Interrupt

*Similar functionality with polling:*

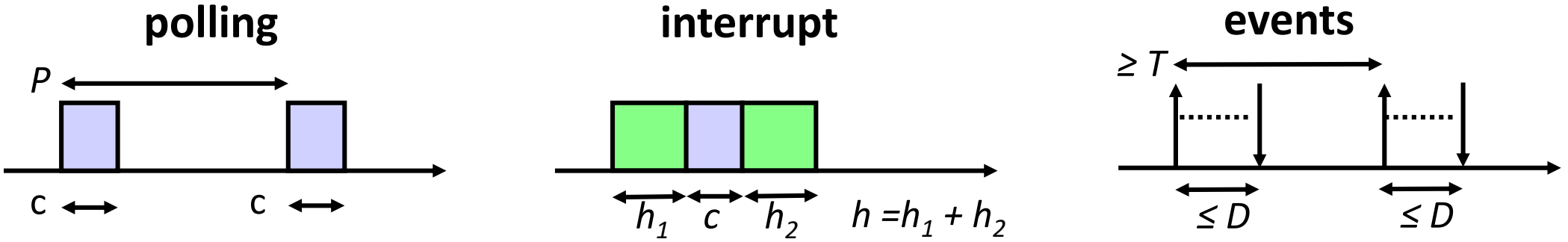
```
int main(void)
{
    uint8_t new, old;
    ...
    GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);
    GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);
    old = GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN1);

    while (1)
    {
        new = GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN1);
        if (!new & old)
        {
            GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0);
        }
        old = new;
    }
}
```

continuously get the signal at pin1 and detect falling edge

# Polling vs. Interrupt

***What are advantages and disadvantages?***

- We *compare polling and interrupt* based on the utilization of the CPU by using a simplified timing model.
- *Definitions:*
  - *utilization u:* average percentage, the processor is busy
  - *computation c:* processing time of handling the event
  - *overhead h:* time overhead for handling the interrupt
  - *period P:* polling period
  - *interarrival time T*: minimal time between two events
  - *deadline D:* maximal time between event arrival and finishing event processing with D ≤ *T.*



# Polling vs. Interrupts

For the following considerations, we suppose that the interarrival time between events is T. This makes the results a bit easier to understand.

Some relations for *interrupt-based* event processing :

- The average utilization is $u_i = (h + c) / T$ .
- As we need at least h+c time to finish the processing of an event, we find the following constraint: $h+c \leq D \leq T$ .

Some relations for *polling-based* event processing:

- The average utilization is $u_p = c / P$ .
- We need at least time $P+c$ to process an event that arrives shortly after a polling took place. The polling period $P$ should be larger than $c$. Therefore, we find the following constraints: $2c \leq c+P \leq D \leq T$

# Polling vs. Interrupts

**Design problem:** *D* and *T* are given by application requirements. *h* and *c* are given by the implementation. When to use interrupt and when polling when considering the resulting system utilization? What is the best value for the polling period P?

**Case 1**: If $D < c + min(c, h)$ then event processing is not possible.

**Case 2**: If $2c \leq D < h+c$ then only polling is possible. The maximal period $P = D-c$ leads to the optimal utilization $u_p = c / (D-c)$ .

**Case 3**: If $h+c \leq D < 2c$ then only interrupt is possible with utilization $u_i = (h + c) / T$ .

**Case 4**: If $c + max(c, h) \leq D$ then both are possible with $u_p = c / (D-c)$ or $u_i = (h + c) / T$ .

*Interrupt gets better in comparison to polling, if the deadline D for processing interrupts gets smaller in comparison to the interarrival time T, if the overhead h gets smaller in comparison to the computation time c, or if the interarrival time of events is only lower bounded by T (as in this case polling executes unnecessarily).*

# Clocks and Timers

# Clocks and Timers

## Clocks

# Clocks

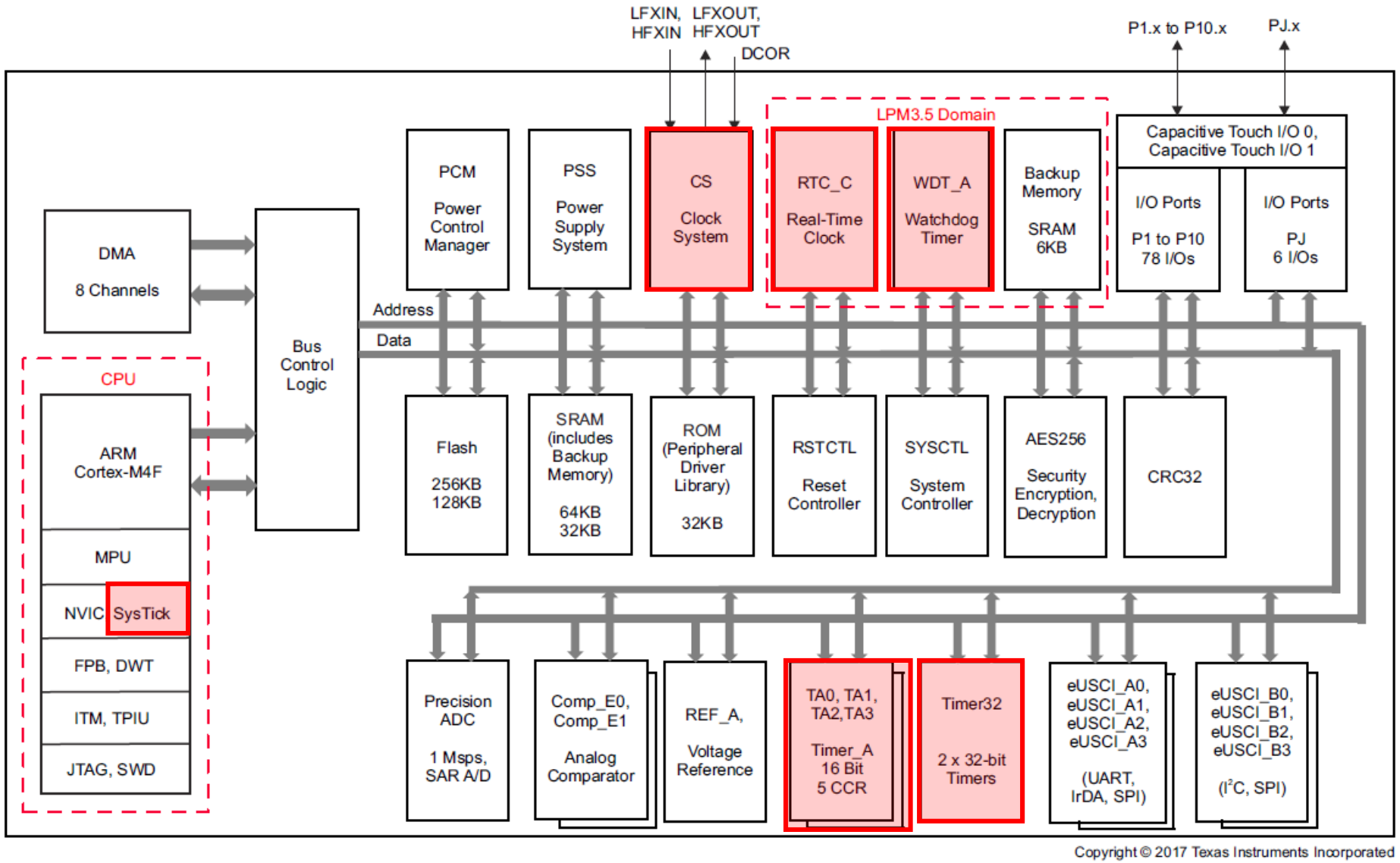
Microcontrollers usually have *many different clock sources* that have different

- frequency (relates to precision)
- energy consumption
- stability, e.g., crystal-controlled clock vs. digitally controlled oszillator

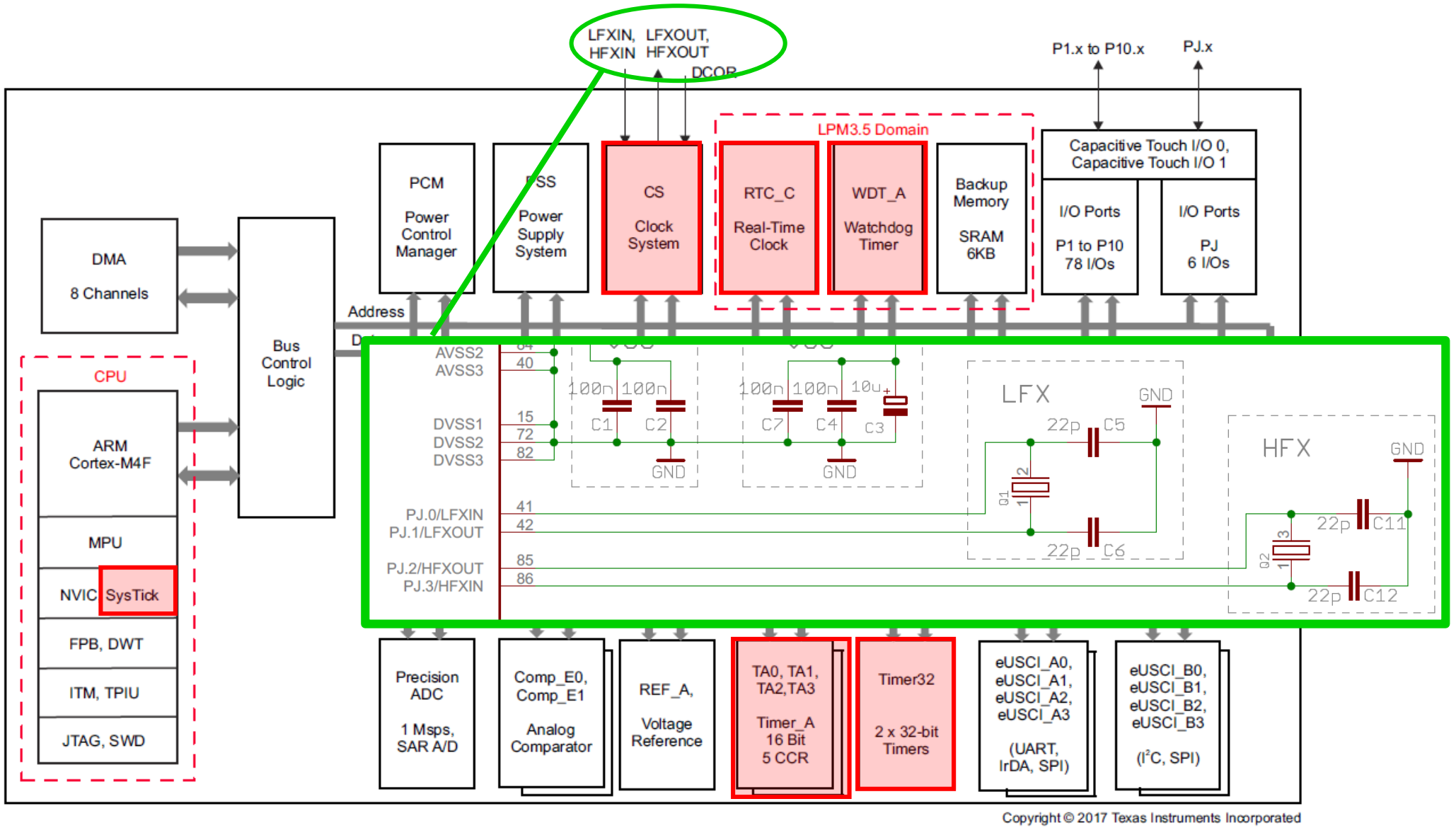
As an example, the MSP432 (ES-Lab) has the following *clock sources*:

| | frequency | precision | current | comment |
|---|---|---|---|---|
| LFXTCLK | 32 kHz | 0.0001% / °C … 0.005% / °C | 150 nA | external crystal |
| HFXTCLK | 48 MHz | 0.0001% / °C … 0.005% / °C | 550 µA | external crystal |
| DCOCLK | 3 MHz | 0.025% / °C | N/A | internal |
| VLOCLK | 9.4 kHz | 0.1% / °C | 50 nA | internal |
| REFOCLK | 32 kHz | 0.012% / °C | 0.6 µA | internal |
| MODCLK | 25 MHz | 0.02% / °C | 50 µA | internal |
| SYSOSC | 5 MHz | 0.03% / °C | 30 µA | internal |

# Clocks and Timers MSP432 (ES-Lab)



# Clocks and Timers MSP432 (ES-Lab)



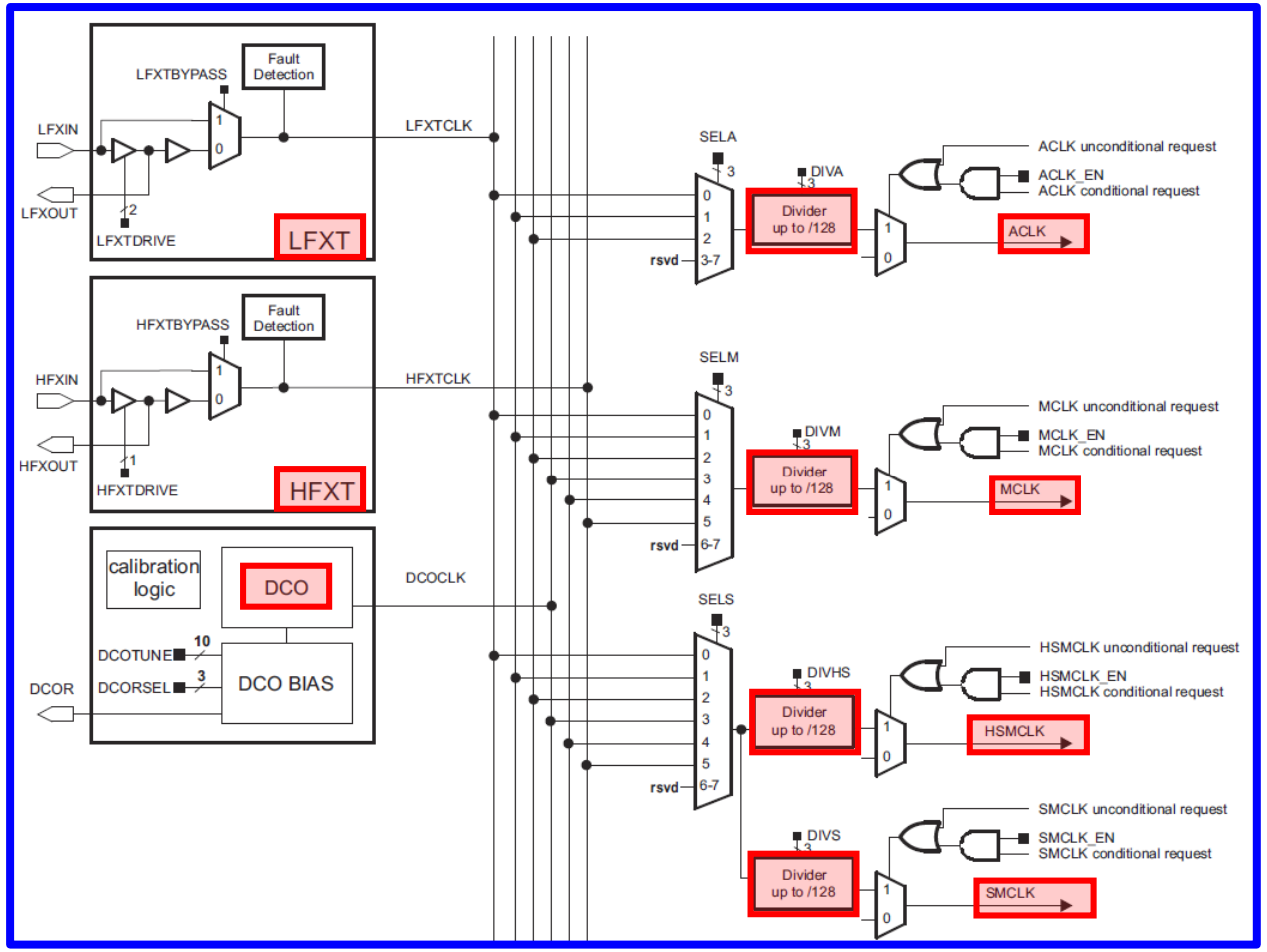
# Clocks

From these basic clocks, *several internally available clock signals* are derived.

They can be used for clocking peripheral units, the CPU, memory, and the various timers.

*Example MSP432 (ES-Lab):*

- only some of the clock generators are shown (LFXT, HFXT, DCO)
- dividers and clock sources for the internally available clock signals can be set by software



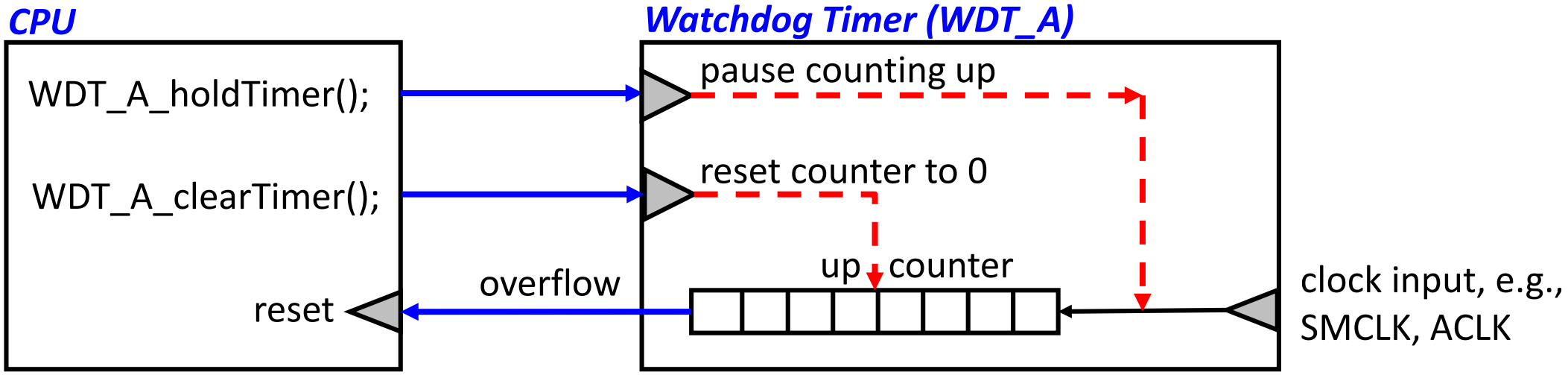
# Clocks and Timers

## Watchdog Timer

# Watchdog Timer

*Watchdog Timers provide system fail-safety*:

- If their counter ever rolls over (back to zero), they *reset the processor*. The goal here is to prevent your system from being inactive (deadlock) due to some unexpected fault.
- To prevent your system from continuously resetting itself, the counter should be reset at appropriate intervals.




**If the count completes without a restart, the CPU is reset.**

# Clocks and Timers

## System Tick

# SysTick MSP432 (ES-Lab)

- ***SysTick*** is a simple *decrementing 24 bit counter* that is part of the NVIC controller (Nested Vector Interrupt Controller). Its clock source is MCLK and it reloads to period-1 after reaching 0.
- It's a *very simple timer*, mainly used for periodic interrupts or measuring time.

```
int main(void) {
     ...
     GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);

     SysTick_enableModule();
     SysTick_setPeriod(1500000);
     SysTick_enableInterrupt();
     Interrupt_enableMaster();

     while (1) PCM_gotoLPM0();
}
void SysTick_Handler(void) {
     MAP_GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0); }
```

if MCLK has a frequency of 3 MHz, an interrupt is generated every 0.5 s.

← go to low power mode LP0 after executing the ISR

# SysTick MSP432 (ES-Lab)

*Example for measuring the execution time* of some parts of a program:

```
int main(void) {
     int32_t start, end, duration;
     ...
     SysTick_enableModule();
     SysTick_setPeriod(0x01000000);
     SysTick_disableInterrupt();

     start = SysTick_getValue();

     ... // part of the program whose duration is measured

     end = SysTick_getValue();
     duration = ((start - end) & 0x00FFFFFF) / 3;

     ...
}
```

if MCLK has frequency of 3 MHz,
the counter rolls over every ~5.6 seconds
as $(2^{24} / (3\ 10^{6}) = 5.59$

the resolution of the duration is one microsecond; the duration must not be longer than ~6 seconds; note the use of modular arithmetic if end > start; overhead for calling SysTick_getValue() is not accounted for;

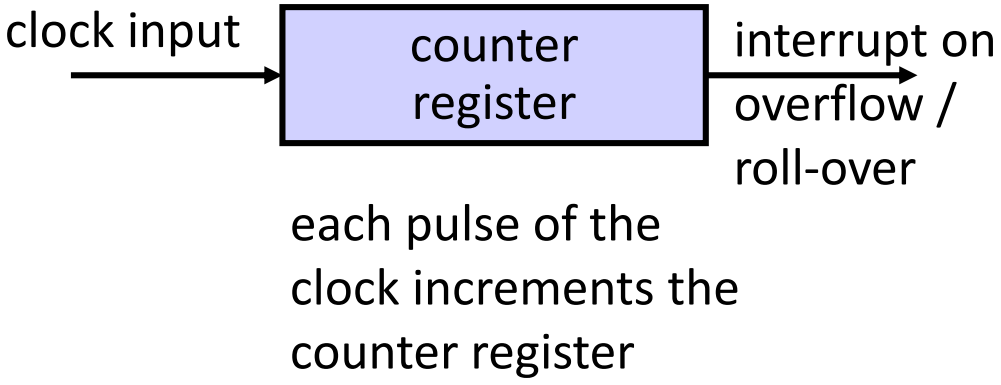
# Clocks and Timers

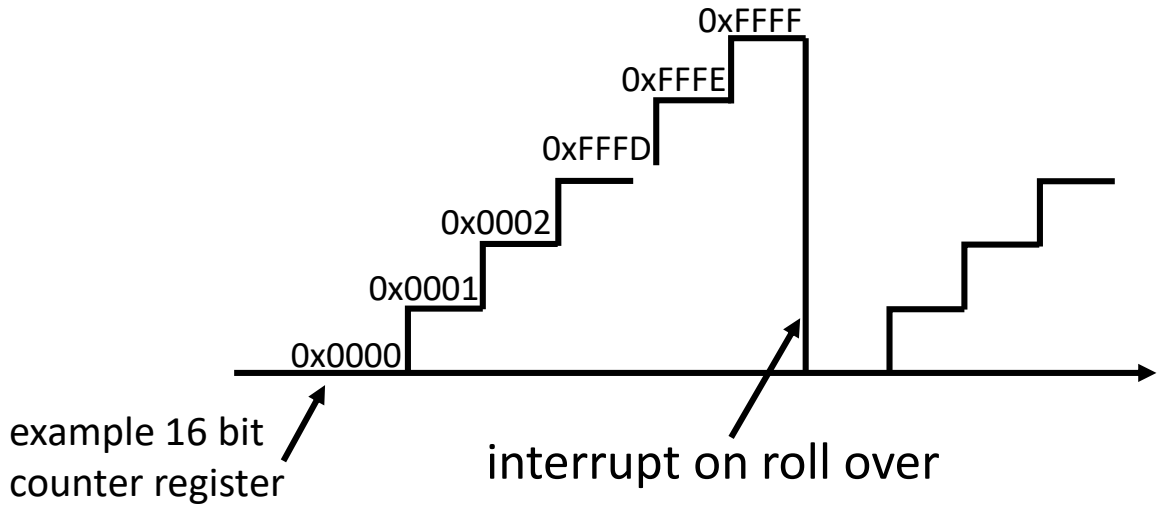
## Timer and PWM

# Timer

Usually, *embedded microprocessors* have *several* elaborate *timers* that allow to

- *capture the current time* or time differences, triggered by hardware or software events,
- generate interrupts when a *certain time is reached* (stop watch, timeout),
- generate interrupts when *counters overflow*,
- generate *periodic interrupts*, for example in order to periodically execute tasks,
- generate *specific output signals*, for example PWM (*pulse width modulation*).

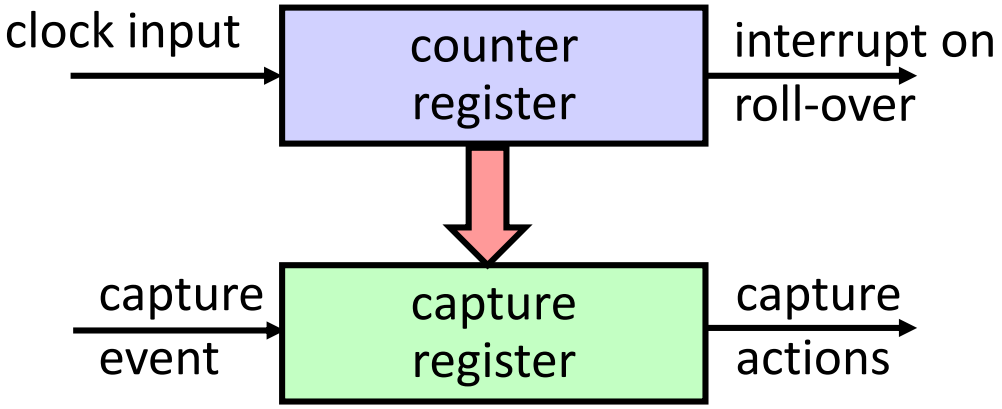


# Timer

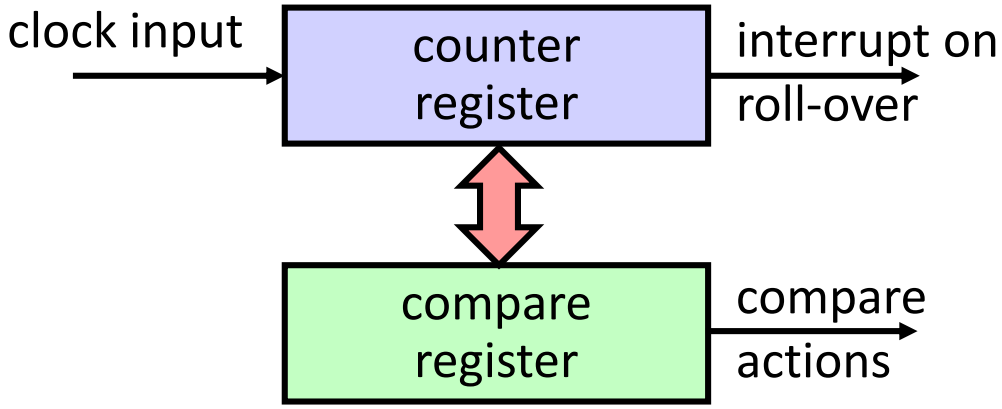
Typically, the mentioned functions are realized via *capture and compare registers*:

**capture**




- the value of *counter register* is stored in *capture register* at the time of the *capture event* (input signals, software)
- the value can be read by software
- at the time of the capture, further actions can be triggered (interrupt, signal)

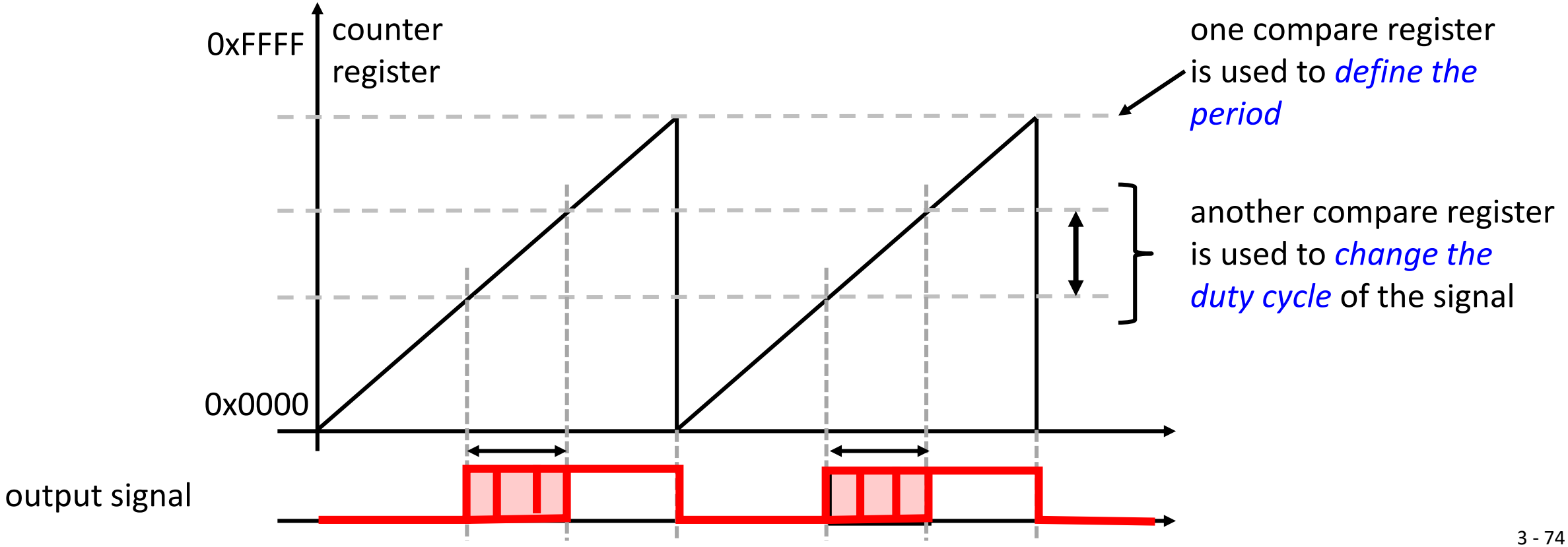
**compare**




- the value of the *compare register* can be set by software
- as soon as the values of the *counter and compare register are equal*, compare actions can be taken such as interrupt, signaling peripherals, changing pin values, resetting the counter register

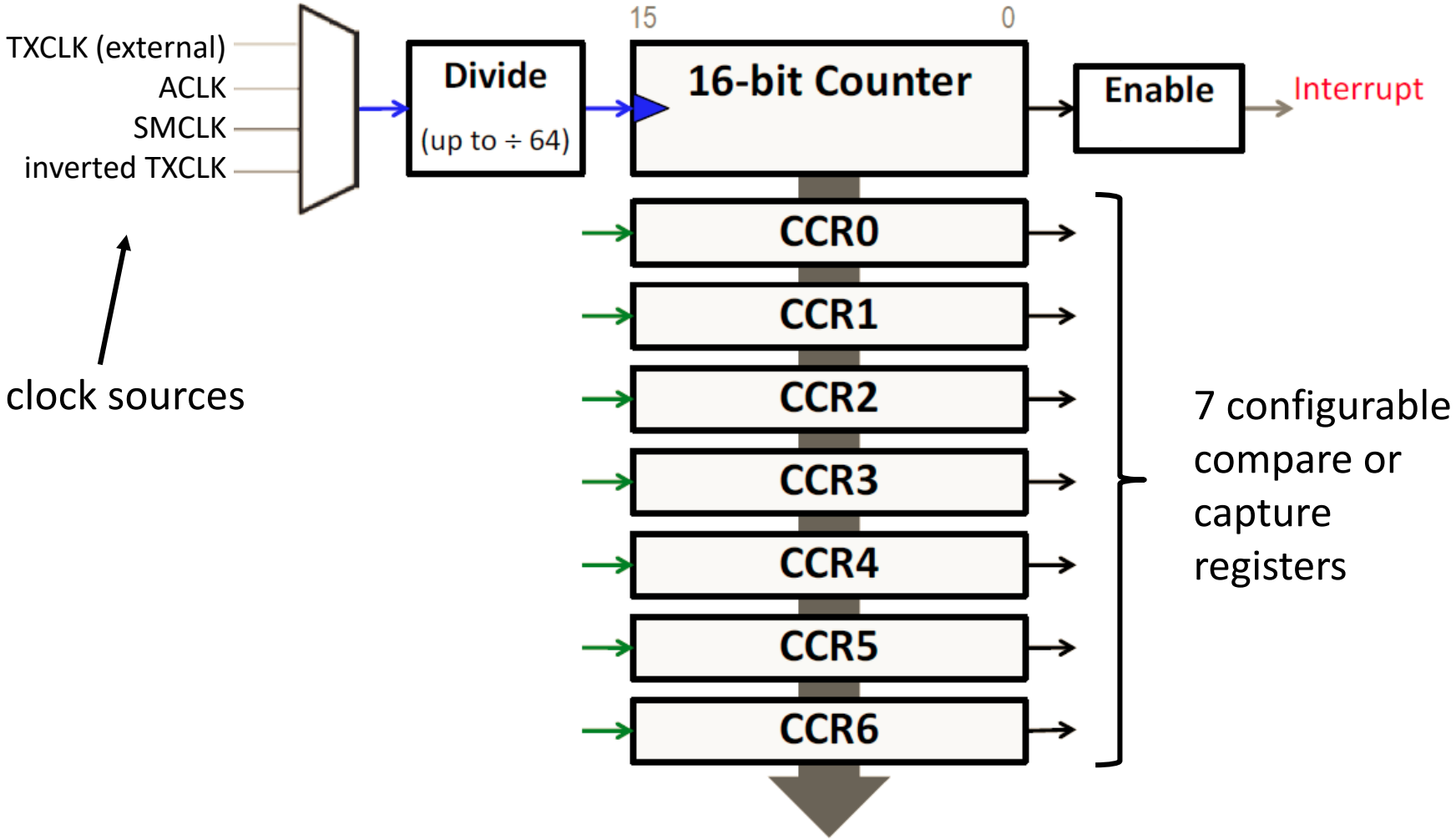
# Timer

- *Pulse Width Modulation (PWM)* can be used to *change the average power* of a signal.
- The use case could be to change the speed of a motor or to modulate the light intensity of an LED.



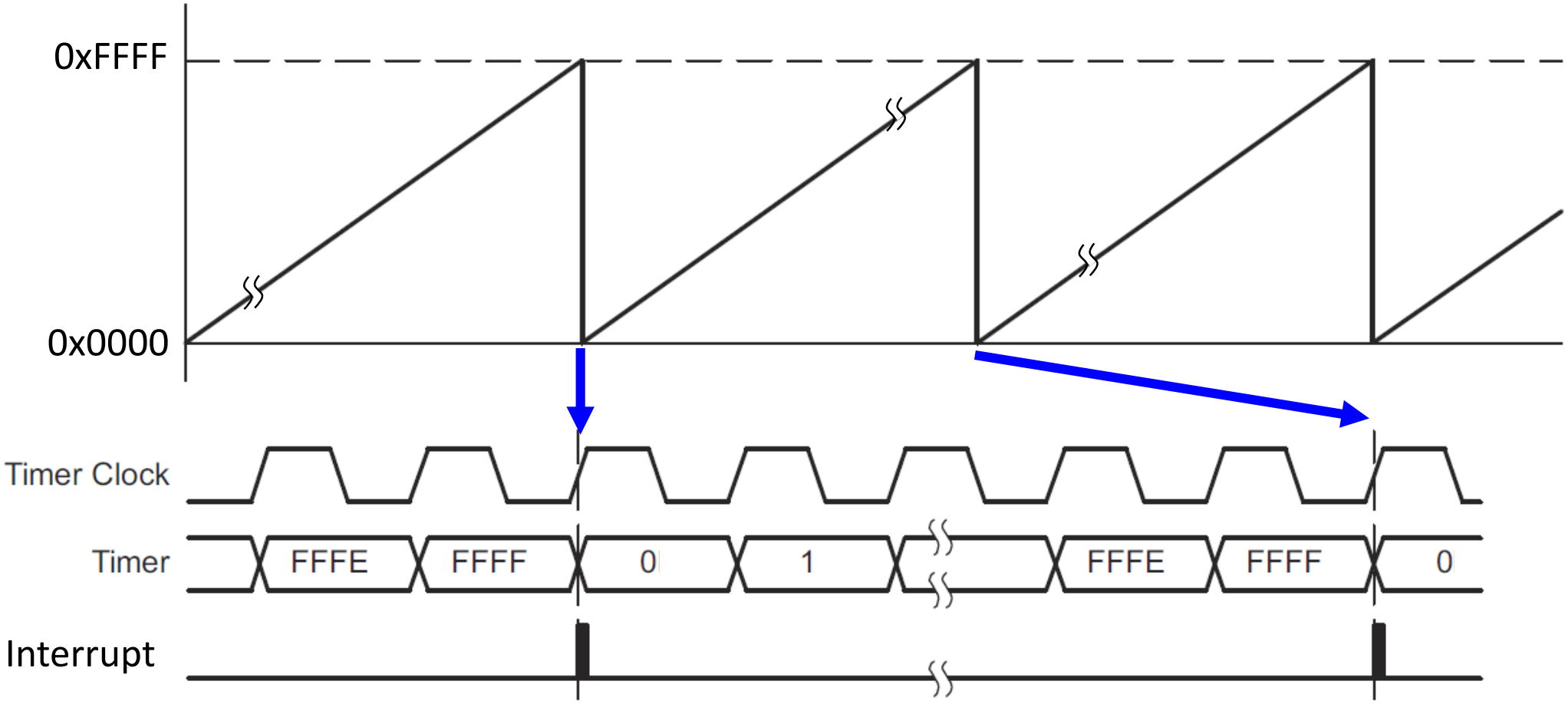
# Timer Example MSP432 (ES-Lab)

***Example:*** Configure Timer in “continuous mode”. *Goal:* generate periodic interrupts.



# Timer Example MSP432 (ES-Lab)

***Example:*** Configure Timer in “continuous mode”. *Goal:* generate periodic interrupts.



# Timer Example MSP432 (ES-Lab)

***Example:*** Configure Timer in "continuous mode". *Goal:* generate periodic interrupts, **but** with configurable periods.

```
int main(void) {
   ...
   const Timer_A_ContinuousModeConfig continuousModeConfig = {
      TIMER_A_CLOCKSOURCE_ACLK,
      TIMER_A_CLOCKSOURCE_DIVIDER_1,
      TIMER_A_TAIE_INTERRUPT_DISABLE,
      TIMER_A_DO_CLEAR};
   ...

   Timer_A_configureContinuousMode(TIMER_A0_BASE, &continuousModeConfig);
   Timer_A_startCounter(TIMER_A0_BASE, TIMER_A_CONTINUOUS_MODE);
   ...

   while(1) PCM_gotoLPM0(); }
```

*clock source is ACLK* (32.768 kHz);
divider is 1 (count frequency 32.768 kHz);
no interrupt on roll-over;

configure *continuous mode* of timer instance A0

*start counter* A0 in continuous mode

so far, nothing happens

only the counter is running

# Timer Example MSP432 (ES-Lab)

***Example:***

- For a *periodic interrupt*, we need to add a *compare register and an ISR*.
- The following code should be added as a definition:

```
#define PERIOD 32768
```

- The following code should be added to main():

```
const Timer_A_CompareModeConfig compareModeConfig = {
    TIMER_A_CAPTURECOMPARE_REGISTER_1,
    TIMER_A_CAPTURECOMPARE_INTERRUPT_ENABLE,
    0,
    PERIOD};
...
Timer_A_initCompare(TIMER_A0_BASE, &compareModeConfig);
Timer_A_enableCaptureCompareInterrupt(TIMER_A0_BASE, TIMER_A_CAPTURECOMPARE_REGISTER_1);
Interrupt_enableInterrupt(INT_TA0_N);
Interrupt_enableMaster();
...
```

a first interrupt is generated *after about one second* as the counter frequency is 32.768 kHz

# Timer Example MSP432 (ES-Lab)

***Example:***

- For a *periodic interrupt*, we need to add a *compare register and an ISR*.
- The following *Interrupt Service Routine (ISR)* should be added. It is called if one of the capture/compare registers CCR1 … CCR6 raises an interrupt

```
void TA0_N_IRQHandler(void) {

   switch(TA0IV) {
      case 0x0002: //flag for register CCR1
         TA0CCR1 = TA0CCR1 + PERIOD;
         ...  // do something every PERIOD
      default: break;
   }
}
```

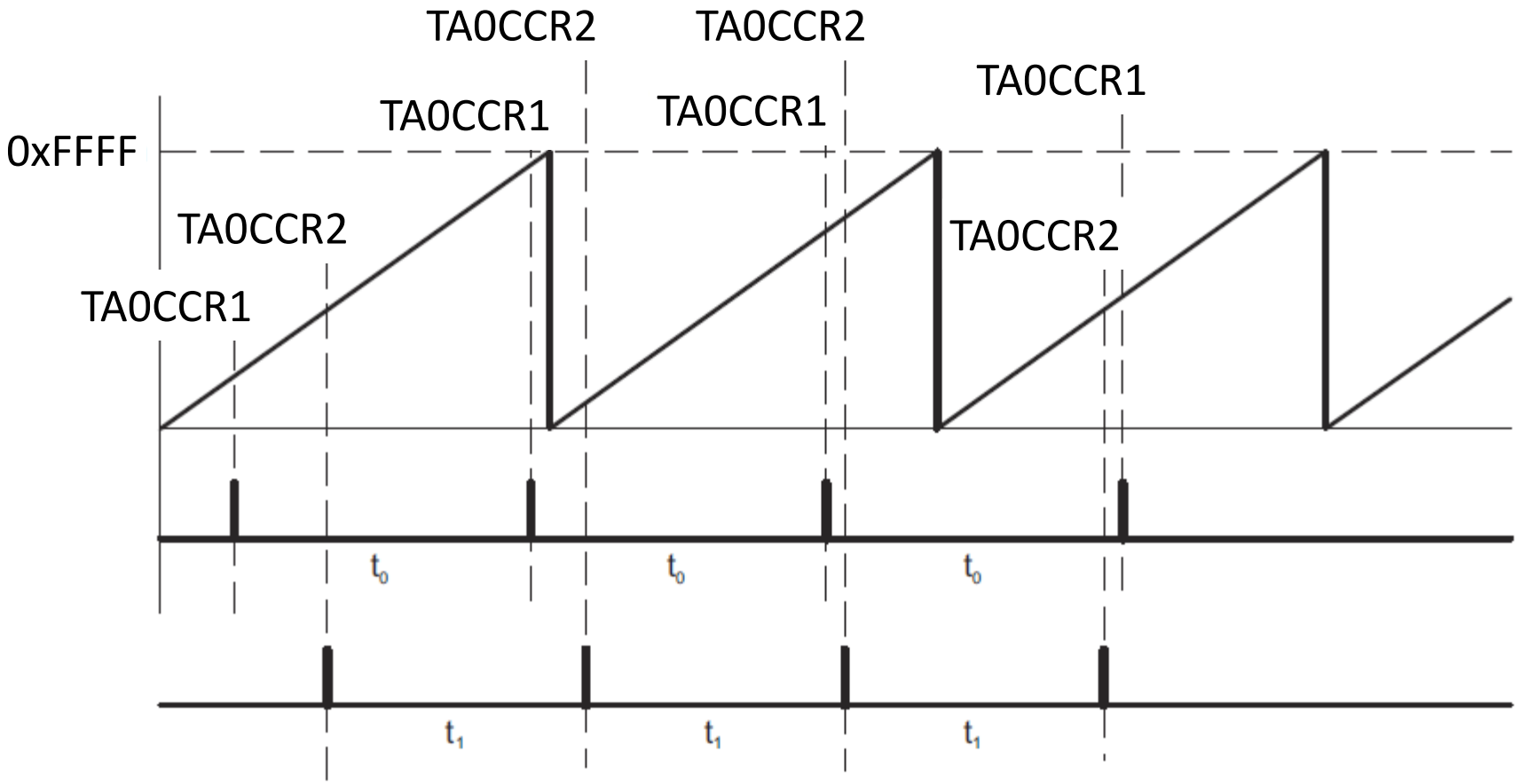
the register TA0IV contains the *interrupt flags* for the registers; after being read, the *highest priority interrupt* (smallest register number) is *cleared automatically*.

the register TA0CCR1 contains the *compare value* of compare register 1.

other cases in the switch statement may be used to handle other capture and compare registers

# Timer Example MSP432 (ES-Lab)

***Example:*** This principle can be used to *generate several periodic interrupts* with *one timer*.



# Embedded Systems

## 4. Programming Paradigms

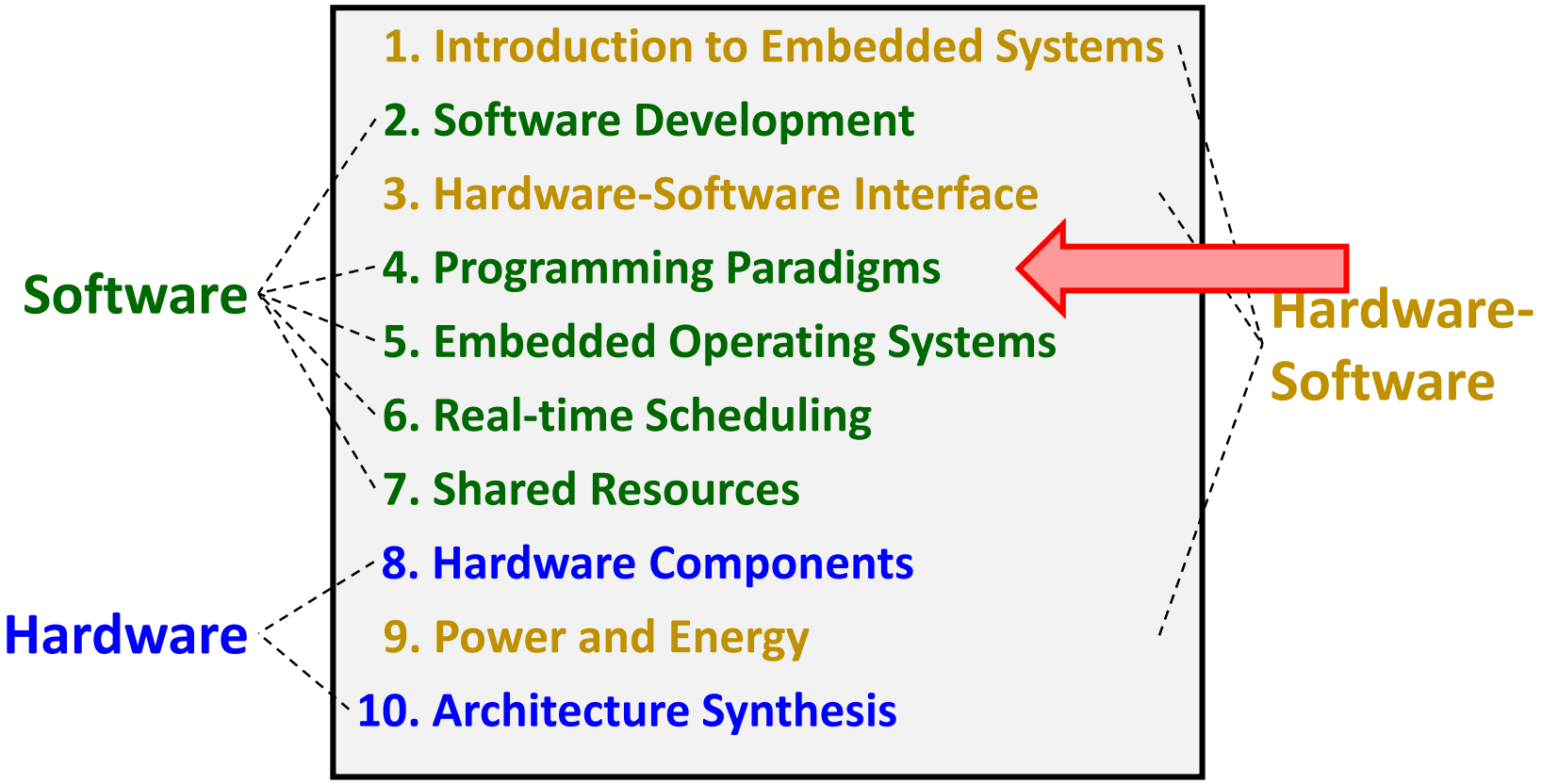
© Lothar Thiele

Computer Engineering and Networks Laboratory

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Where we are …


1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis
Software
Hardware
Hardware-
Software

# Reactive Systems and Timing

# Timing Guarantees

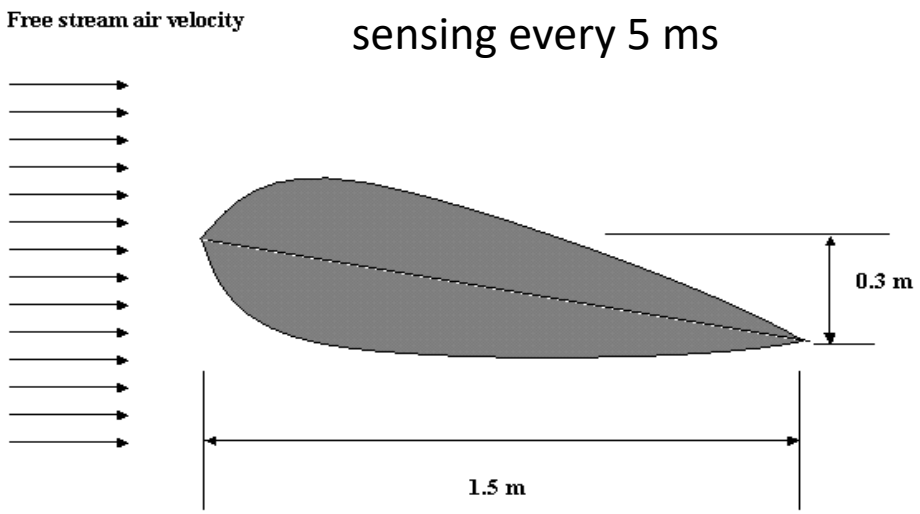
- *Hard real-time systems* can be often found in *safety-critical applications*. They need to provide the result of a computation within a fixed time bound.
- *Typical application domains:*
  - avionics, automotive, train systems, automatic control including robotics, manufacturing, media content production

sideairbag in car,

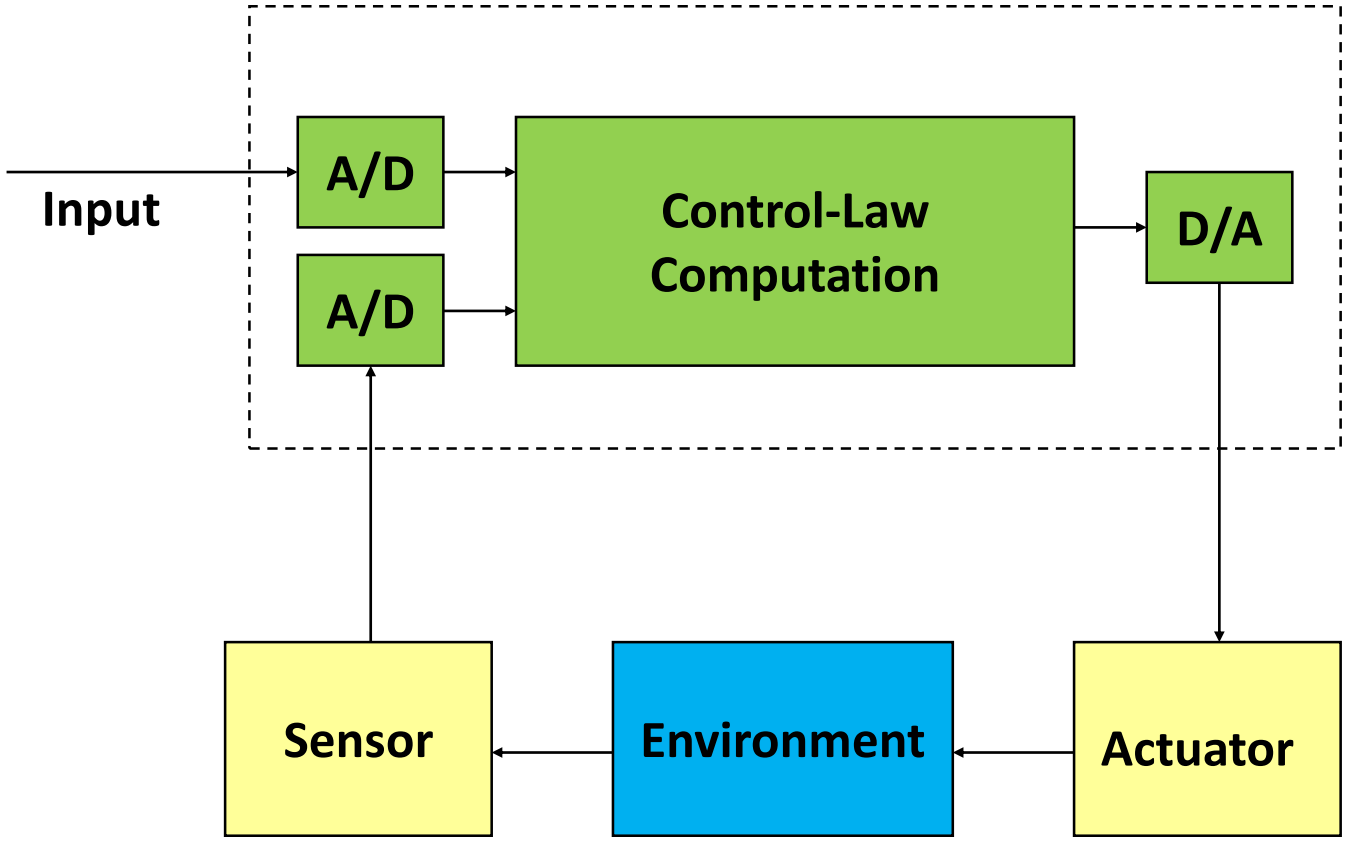
reaction after event in <10 mSec

wing vibration of airplane,

sensing every 5 ms

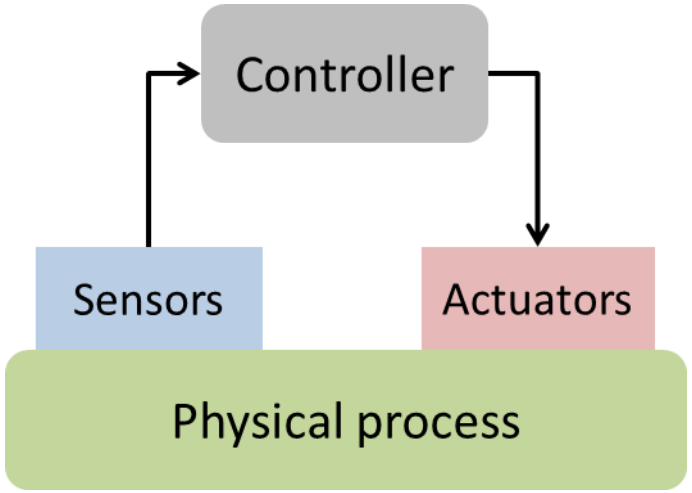


# Simple Real-Time Control System

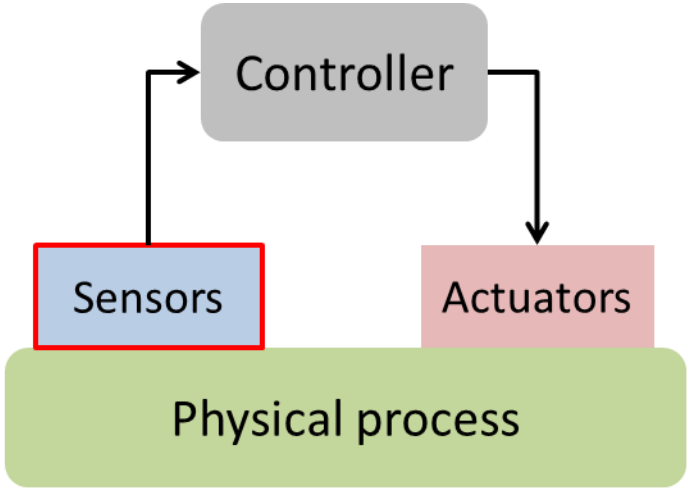


# Real-Time Systems

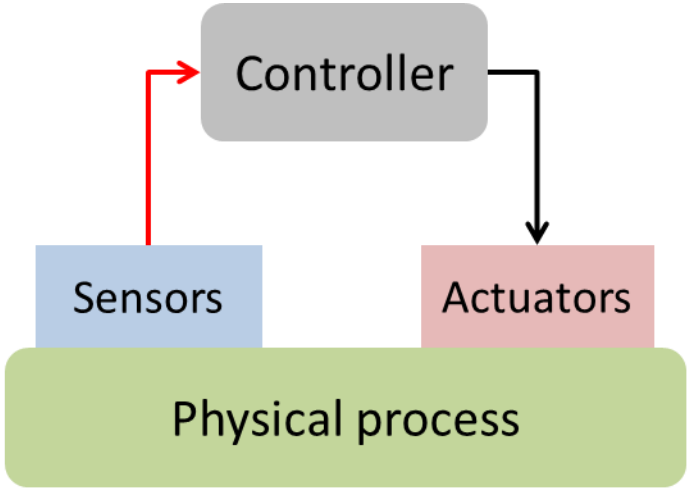
In many *cyber-physical systems (CPSs),* correct timing is a matter of *correctness*, not performance: *an answer arriving too late is consider to be an error.*

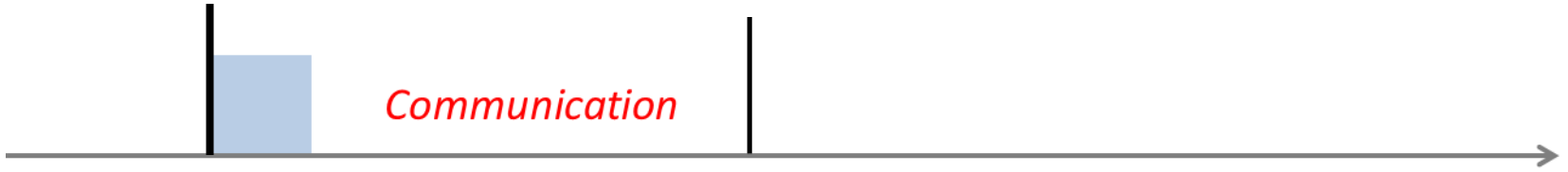


# Real-Time Systems

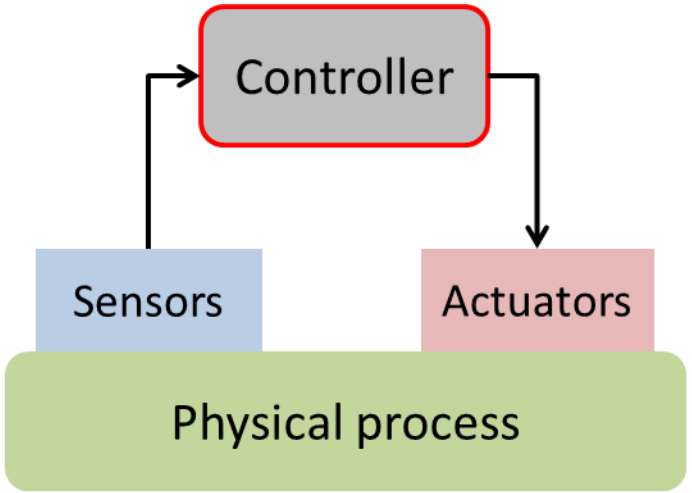


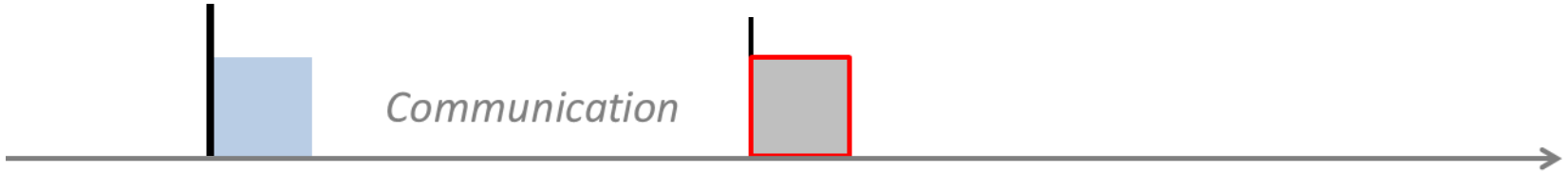

# Real-Time Systems



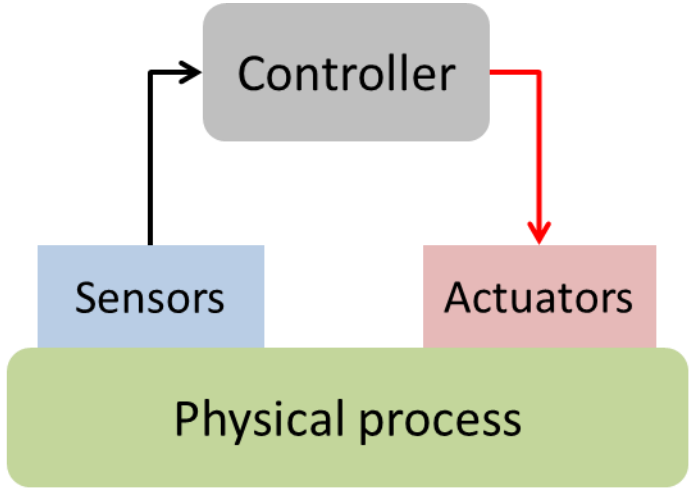



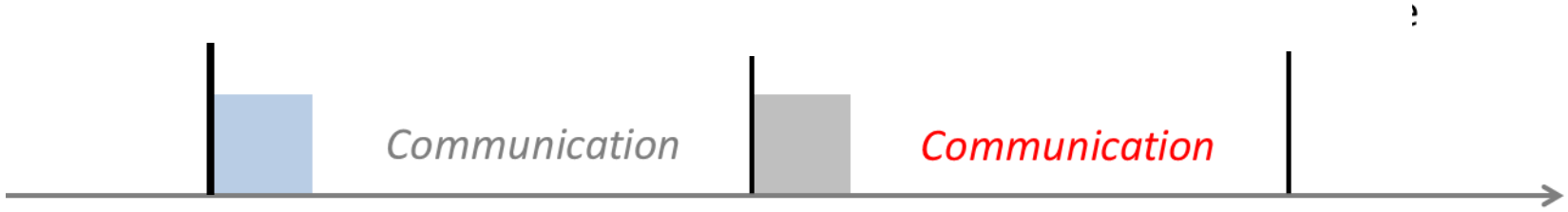
# Real-Time Systems



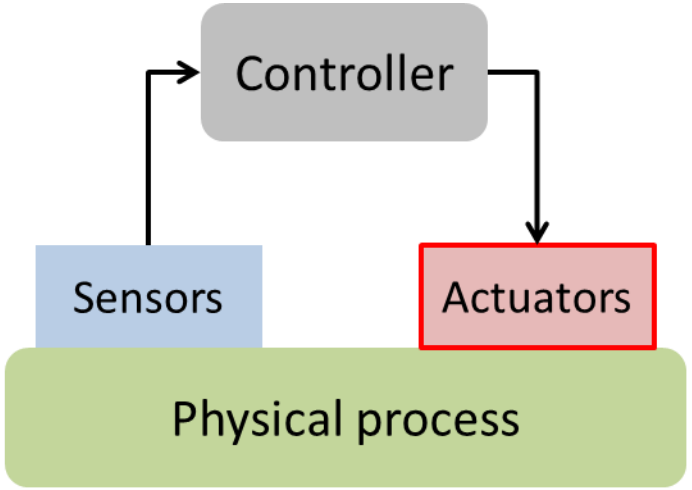



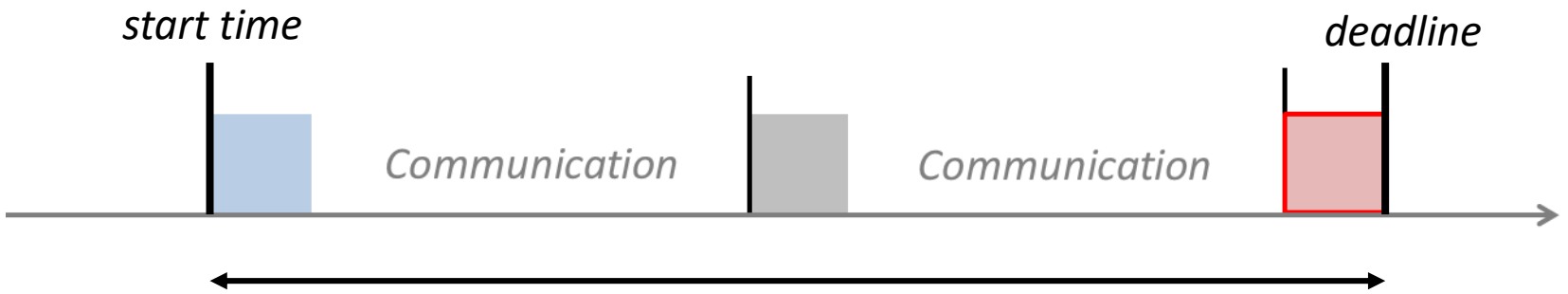
# Real-Time Systems



# Real-Time Systems



# Real-Time Systems

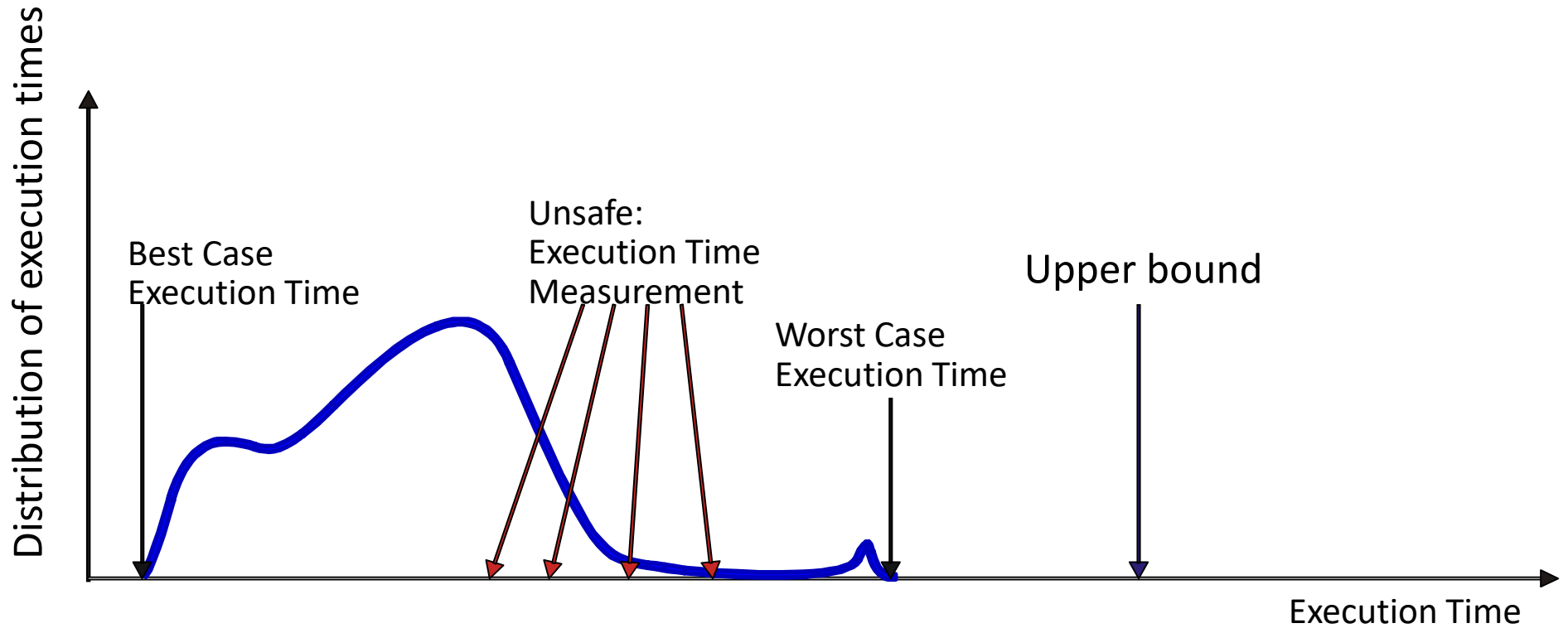
- *Embedded controllers* are often expected to *finish the processing* of data and events reliably *within defined time bounds*. Such a processing may involve sequences of computations and communications.

- Essential for the analysis and design of a real-time system: *Upper bounds on the execution times* of all tasks are statically known. This also includes the communication of information via a wired or wireless connection.

  - This value is commonly called the *Worst-Case Execution Time* (WCET).

  - Analogously, one can define the lower bound on the execution time, the *Best-Case Execution Time* (BCET).

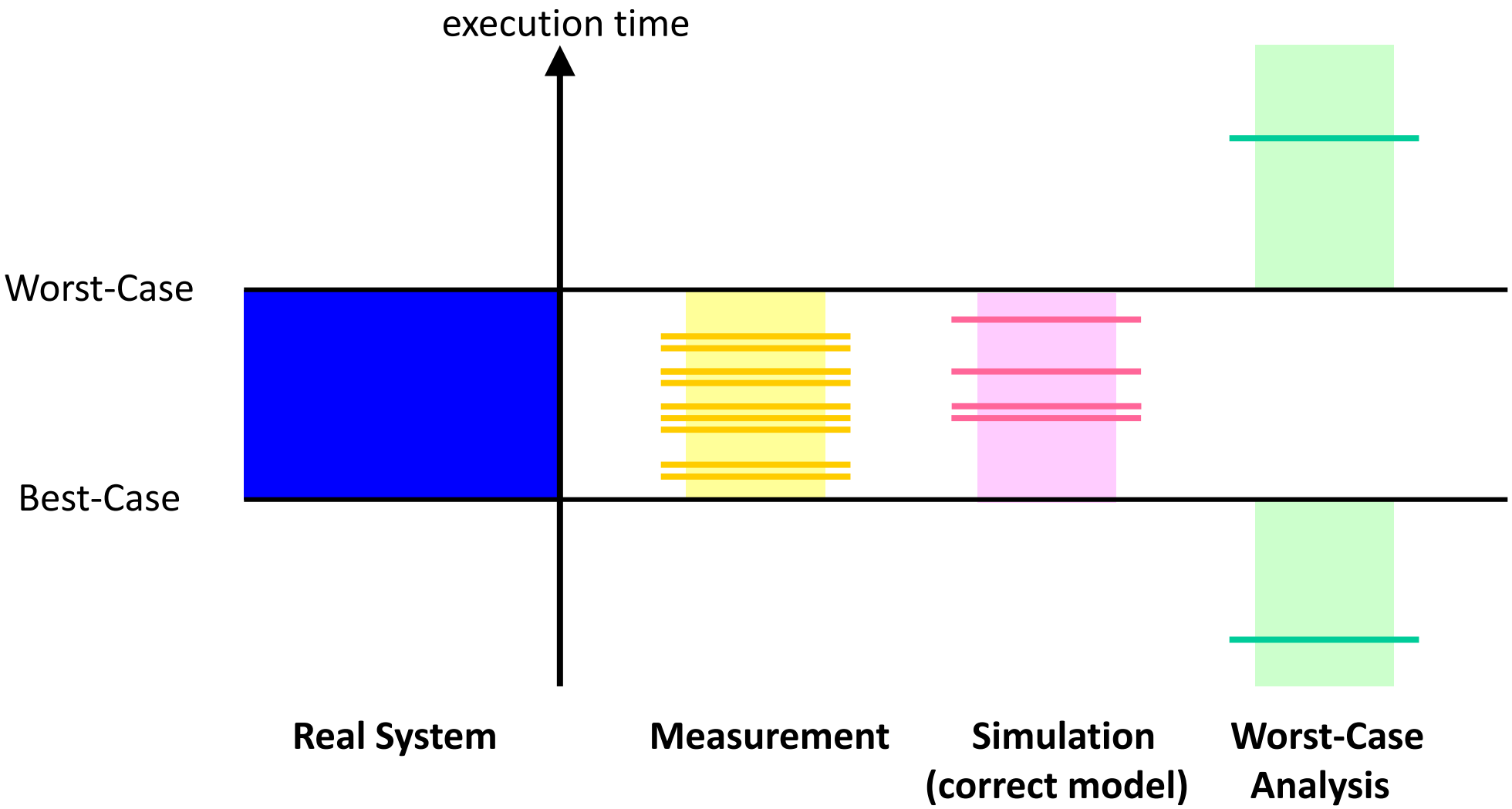
# Distribution of Execution Times



# Modern Hardware Features

- Modern processors *increase the average performance* (execution of tasks) by using *caches, pipelines, branch prediction*, and *speculation* techniques, for example.
- *These features make the computation of the WCET very difficult*: The execution times of single instructions vary widely.
- The microarchitecture has a large *time-varying internal state* that is changed by the execution of instructions and that influences the execution times of instructions.
  - *Best case* - everything goes smoothely: 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.
  - *The span between the best case and worst case may be several hundred cycles.*

# Methods to Determine the Execution Time of a Task



# (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 unless the reaction to all initial system states and all possible inputs is measured.
  - *Exhaustive execution* in general not possible: Too large space of (input domain) x (set of initial execution states).
- ***Simulation*** suffers from the same restrictions.

- ***Compute upper bounds*** along the structure of the program:
  - Programs are *hierarchically* structured: Instructions are "nested" inside statements.
  - Therefore, one may compute the upper execution time bound for a statement from the upper bounds of its constituents, for example of single instructions.
  - *But:* The execution times of individual instructions varies largely!

# Determine the WCET

*Complexity of determining the WCET of tasks:*

- In the general case, it is even *undecidable* whether a finite bound exists.
- For *restricted classes of programs* it is possible, in principle. Computing accurate bounds is *simple for „old“ architectures*, but very *complex for new architectures* with pipelines, caches, interrupts, and virtual memory, for example.

*Analytic (formal) approaches* exist for hardware and software.

- In case of software, it requires the *analysis of the program flow* and the *analysis of the hardware* (microarchitecture). Both are combined in a complex analysis flow, see for example www.absint.de and the lecture “*Hardware/Software Codesign*”.
- *For the rest of the lecture, we assume that reliable bounds on the WCET are available,* for example by means of exhaustive measurements or simulations, or by analytic formal analysis.

# Different Programming Paradigms

# Why Multiple Tasks on one Embedded Device?

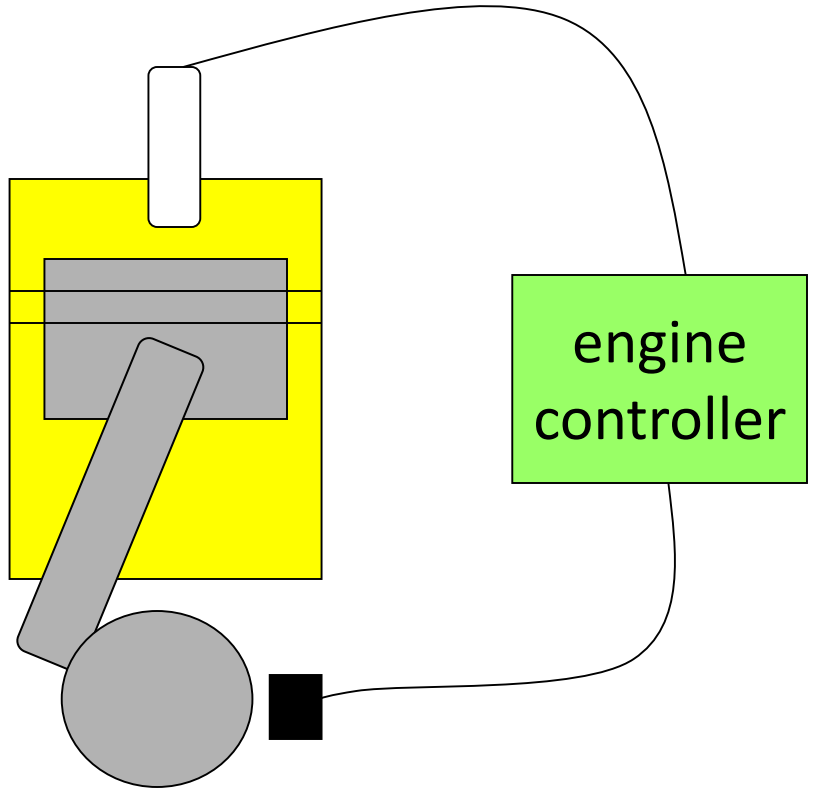
- The concept of *concurrent tasks* reflects our intuition about the *functionality of embedded systems*.

- Tasks help us *manage the complexity of concurrent activities* as happening in the system environment:
  - *Input data* arrive from various *sensors* and input devices.
    - These input streams may have different data rates like in multimedia processing, systems with multiple sensors, automatic control of robots
  - The system may also receive *asynchronous (sporadic) input events*.
    - These input event may arrive from user interfaces, from sensors, or from communication interfaces, for example.

# Example: Engine Control

*Typical Tasks:*

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



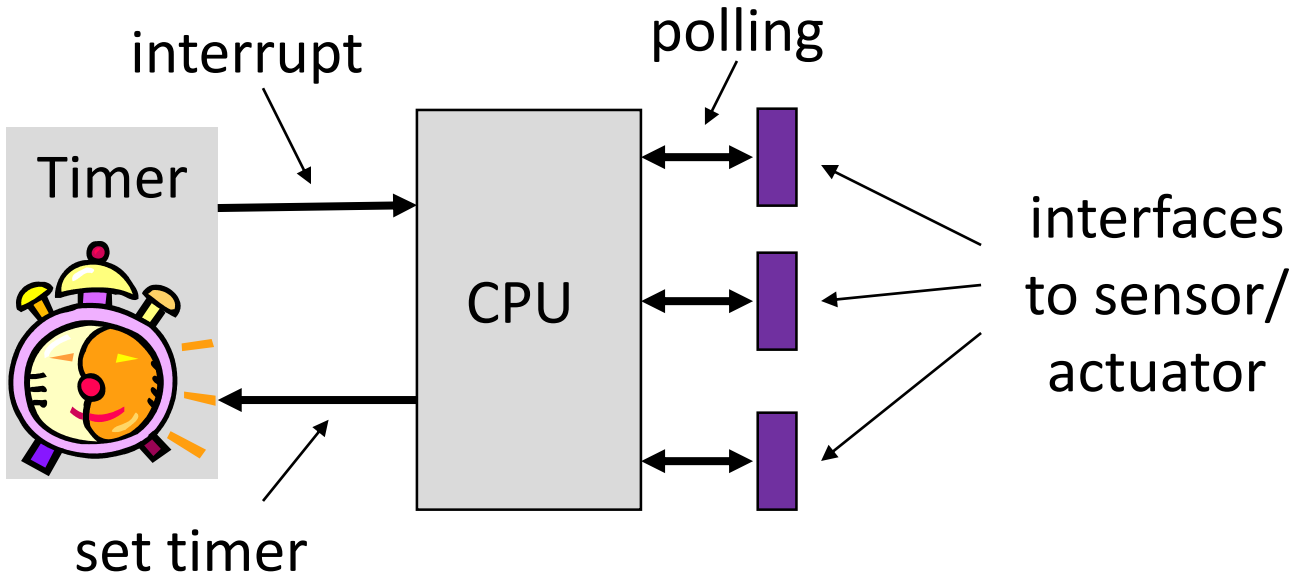
# Overview

- There are many *structured ways of programming an embedded system*.
- In this lecture, only the 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

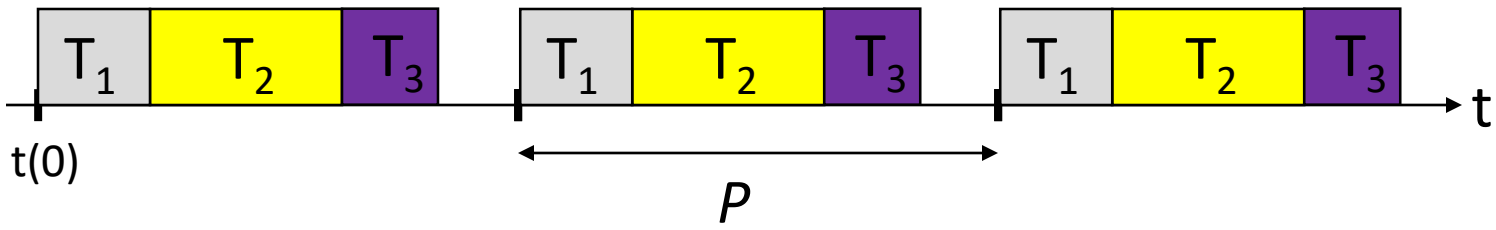
*Pure time-triggered model:*

- *no interrupts* are allowed, except by timers
- the *schedule* of tasks is *computed off-line* and therefore, complex sophisticated algorithms can be used
- the scheduling at run-time is fixed and therefore, it is *deterministic*
- the interaction with environment happens through *polling*



# Simple Periodic TT Scheduler

- A *timer interrupts regularly* with period *P*.
- All tasks have *same period P*.




- *Properties:*
  - later tasks, for example $T_2$ and $T_3$, have unpredictable starting times
  - the communication between tasks or the use of common resources is safe, as there is a static ordering of tasks, for example $T_2$ starts after finishing $T_1$
  - as a necessary precondition, the sum of WCETs of all tasks within a period is bounded by the period *P*:

$$\sum_{(k)} WCET(T_k) < P$$

# Simple Periodic Time-Triggered Scheduler

```
main:
   determine table of tasks (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 task T(k); }
   return;
```

usually done offline

set CPU to low power mode;
processing starts again after interrupt

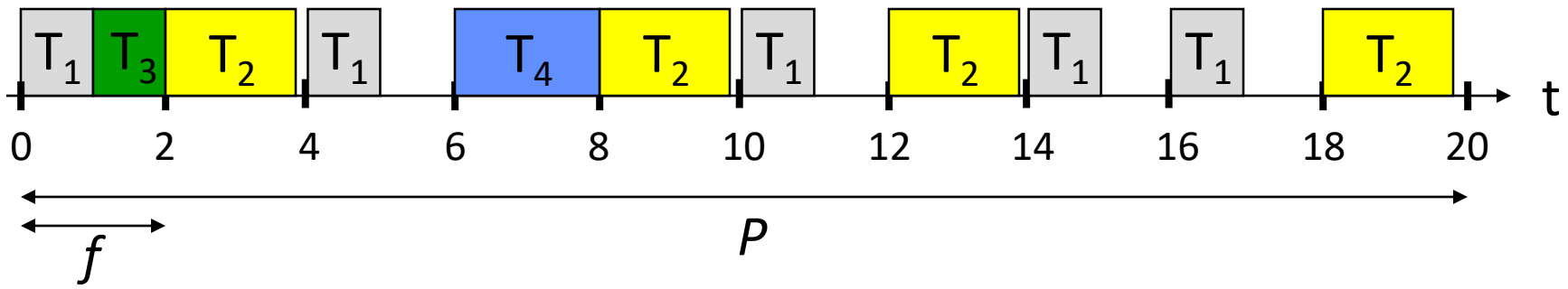
for example using a function pointer in C;
task(= function) returns after finishing.

| k | T(k) |
|---|---|
| 0 | $T_1$ |
| 1 | $T_2$ |
| 2 | $T_3$ |
| 3 | $T_4$ |
| 4 | $T_5$ |

m=5

# Time-Triggered Cyclic Executive Scheduler

- Suppose now, that *tasks may have different periods*.
- To accommodate this situation, the *period P is partitioned into frames of length f*.




- We have a *problem* to determine a feasible schedule, if there are *tasks with a long execution time*.
  - long tasks could be partitioned into a sequence of short sub-tasks
  - but this is tedious and error-prone process, as the local state of the task must be extracted and stored globally

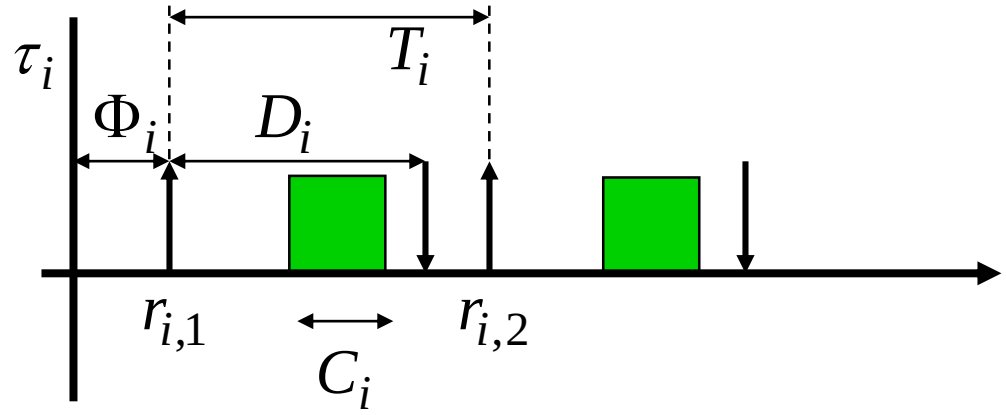
# Time-Triggered Cyclic Executive Scheduling

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

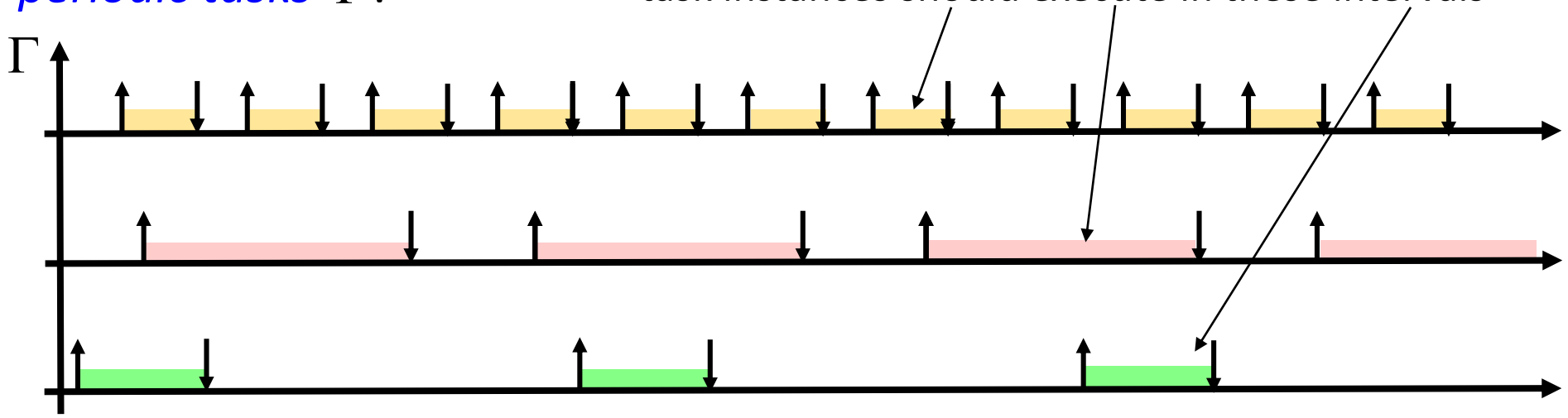
$\Gamma$ : denotes the set of all periodic tasks
$\tau_i$ : denotes a periodic task
$\tau_{i,j}$ : denotes the *j*th instance of task *i*
$r_{i,j}$, $d_{i,j}$ : denote the release time and absolute deadline of the *j*th instance of task *i*
$\Phi_i$ : phase of task *i* (release time of its first instance)
$D_i$ : relative deadline of task *i*

# Time-Triggered Cyclic Executive Scheduling

- *Example* of a single periodic task $\tau_i$:




- *A set of periodic tasks* $\Gamma$:



# Time-Triggered Cyclic Executive Scheduling

- 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* $C_i$. The worst case execution time is also denoted as *WCET*(*i*) .
  - *All instances of a periodic task have the same relative deadline* $D_i$. Therefore, the absolute deadlines satisfy

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

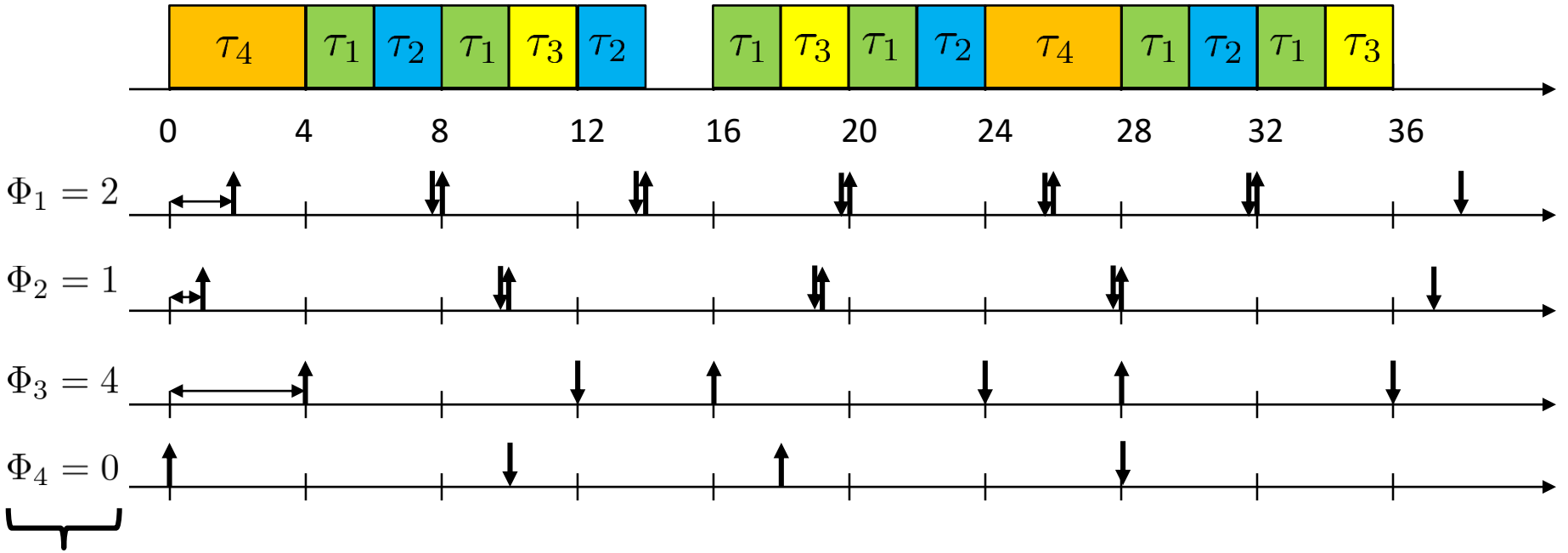
# Time-Triggered Cyclic Executive Scheduling

*Example with 4 tasks:*

- $\tau_1 : T_1 = 6, D_1 = 6, C_1 = 2$ $\quad \tau_2 : T_2 = 9, D_2 = 9, C_2 = 2$
  $\tau_3 : T_3 = 12, D_3 = 8, C_3 = 2$ $\quad \tau_4 : T_4 = 18, D_4 = 10, C_1 = 4$
- $P = 36, f = 4$

requirement

schedule




not given as part of the requirement

# Time-Triggered Cyclic Executive Scheduling

*Some conditions for period P and frame length f:*

- A task executes at most once within a frame:

$$f \leq T_i \ \forall \text{ tasks } \tau_i$$

period of task

- *P* is a multiple of *f*.
- Period *P* is least common multiple of all periods $T_k$ .
- Tasks start and complete within a single frame:

$$f \geq C_i \ \forall \text{ tasks } \tau_i$$

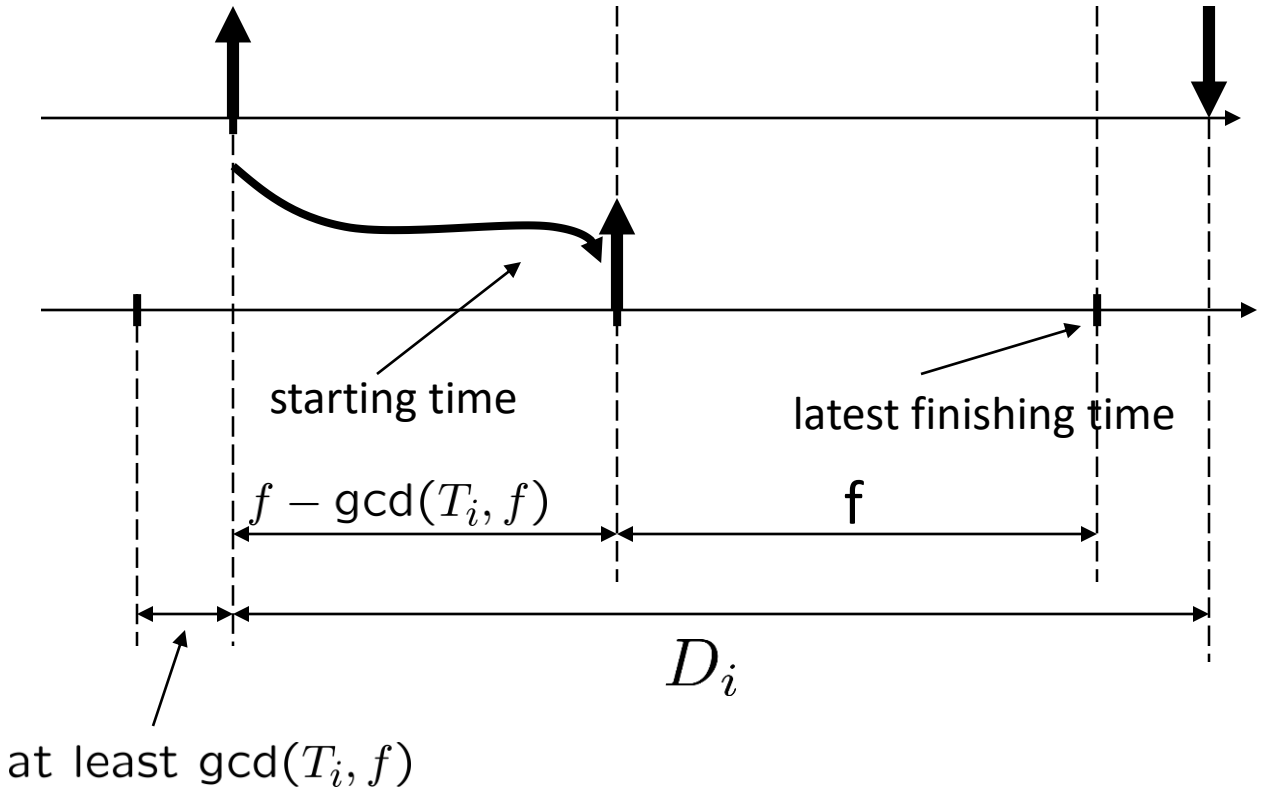
worst case execution time of task

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

$$2f - \gcd(T_i, f) \leq D_i \ \ \forall \text{ tasks } \tau_i$$

relative deadline of task

# Sketch of Proof for Last Condition




release times and deadlines of tasks

frames

# Example: Cyclic Executive Scheduling

*Conditions:*

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

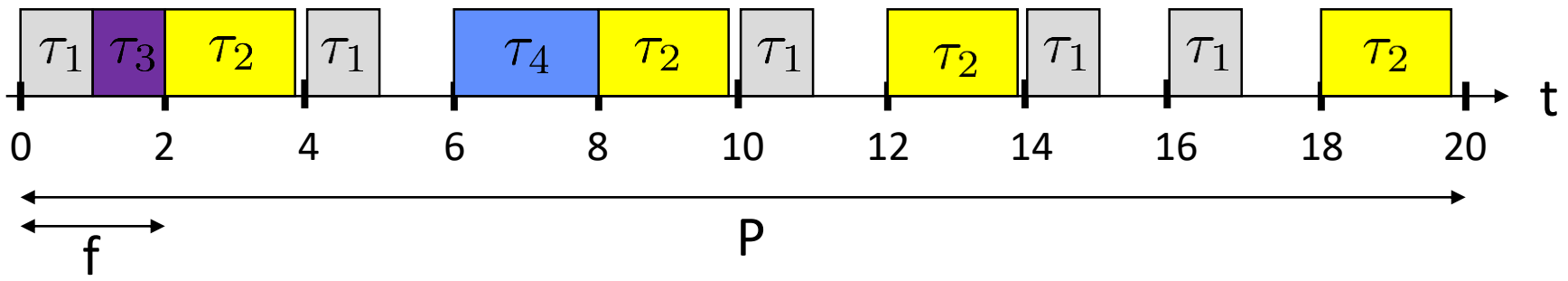
$$f \geq \max\{1.0, 1.0, 1.8, 2.0\} = 2.0$$

$$2f - \gcd(T_i, f) \leq D_i \;\; \forall \text{ tasks } \tau_i$$

possible solution: f = 2

| $\Gamma$ | $T_i$ | $D_i$ | $C_i$ |
|---|---|---|---|
| $\tau_1$ | 4 | 4 | 1.0 |
| $\tau_2$ | 5 | 5 | 1.8 |
| $\tau_3$ | 20 | 20 | 1.0 |
| $\tau_4$ | 20 | 20 | 2.0 |

*Feasible solution (f=2):*



# Time-Triggered Cyclic Executive Scheduling

*Checking for correctness of schedule:*

- $f_{ij}$ denotes the number of the frame in which that instance *j* of task $\tau_i$ executes.
- Is *P* a common multiple of all periods $T_i$ ?
- Is *P* a multiple of *f* ?
- Is the frame sufficiently long?

$$\sum_{\{i \mid f_{ij}=k\}} C_i \leq f \qquad \forall 1 \leq k \leq \frac{P}{f}$$

- Determine offsets such that instances of tasks start after their release time:

$$\Phi_i = \min_{1 \leq j \leq P/T_i} \{(f_{ij} - 1)f - (j-1)T_i\} \qquad \forall \text{ tasks } \tau_i$$

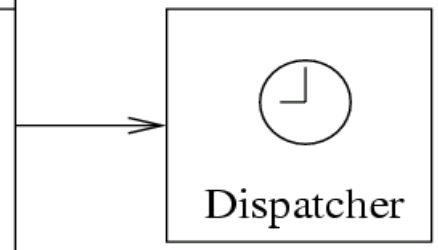
- Are deadlines respected?

$$(j-1)T_i + \Phi_i + D_i \geq f_{ij} f \qquad \forall \text{ tasks } \tau_i \, , \, 1 \leq j \leq P/T_i$$

# 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 a synchronized clock tick.* It looks at the TDL, and then performs the action that has been planned for this instant [Kopetz].

| Time | Action | WCET |
|---|---|---|
| 10 | start T1 | 12 |
| 17 | send M5 | |
| 22 | stop T1 | |
| 38 | start T2 | 20 |
| 47 | send M3 | |



# Simplified Time-Triggered Scheduler

```
main:
   determine static schedule (t(k), T(k)), for k=0,1,…,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 ⌊i/n⌋ * P + t(k);
   execute task T(k_old);
   return;
```

usually done offline

set CPU to low power mode;
processing continues after interrupt

for example using a function pointer in C;
task 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

# Summary Time-Triggered Scheduler

***Properties:***

- *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 the environment
- serious *problems* if there are *long tasks*

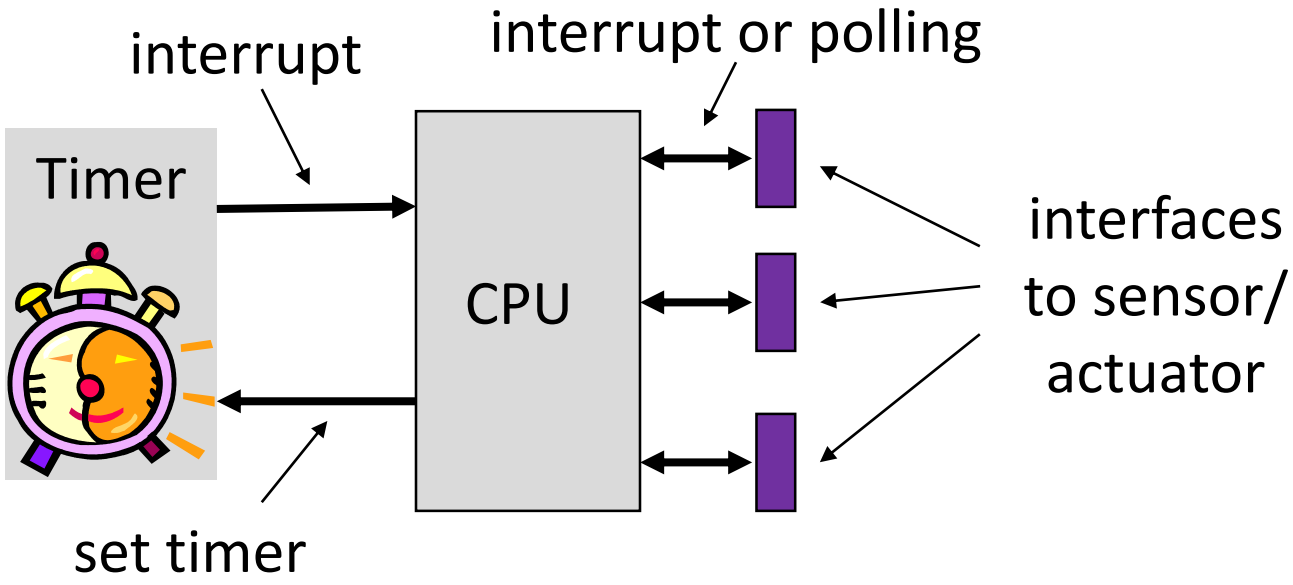
***Extensions:***

- *allow interrupts* → be careful with shared resources and the WCET of tasks!!
- *allow preemptable* background tasks
- *check for task overruns* (execution time longer than WCET) using a watchdog timer

# Event Triggered Systems

*The schedule of tasks is determined by the occurrence of external or internal events:*

- *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 Event-Triggered Scheduling

***Principle:***

- To each event, there is associated a corresponding task that will be executed.
- Events are emitted by (a) external interrupts or (b) by tasks themselves.
- All events are collected in a single queue; depending on the queuing discipline, an event is chosen for execution, i.e., the corresponding task is executed.
- Tasks can not be preempted.

***Extensions:***

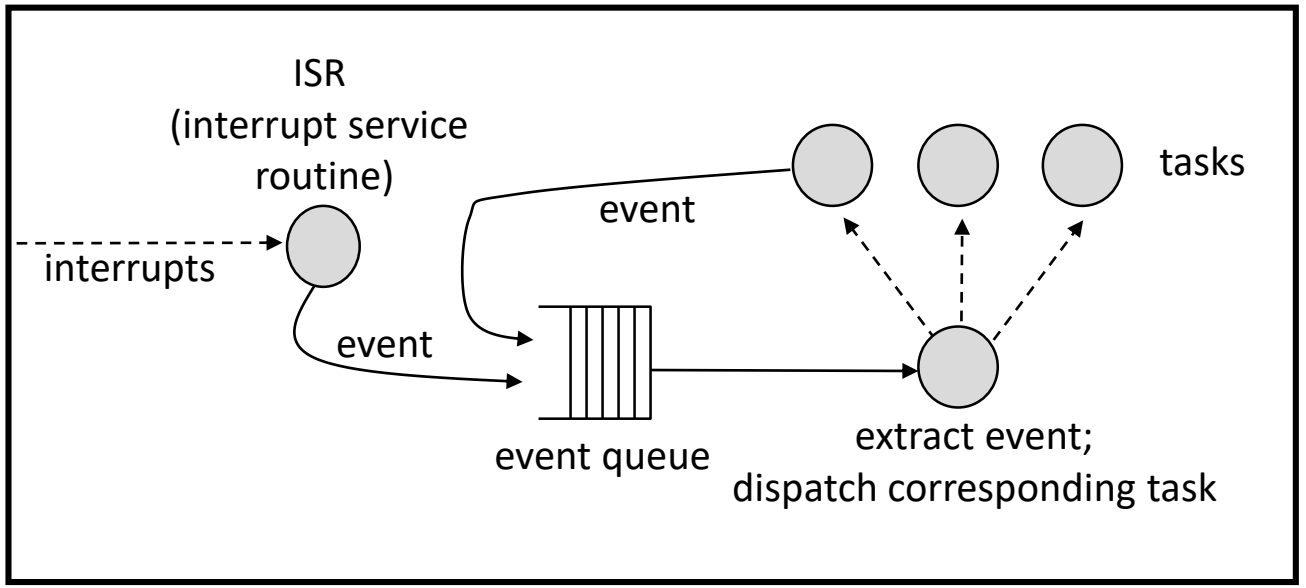
- A *background task* can run if the event queue is empty. It will be preempted by any event processing.
- *Timed events* are ready for execution only after a time interval elapsed. This enables periodic instantiations, for example.

# Non-Preemptive Event-Triggered Scheduling

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

Interrupt:
   put event into event queue;
   return;
```

set the CPU to low power mode;
continue processing after interrupt

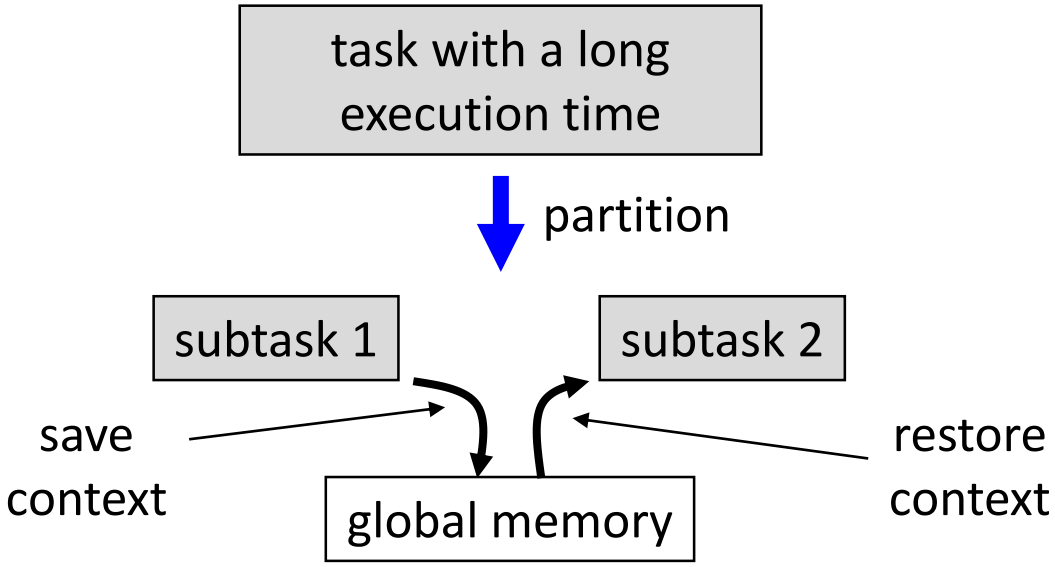
for example using a function pointer in C;
task returns after finishing.



# Non-Preemptive Event-Triggered Scheduling

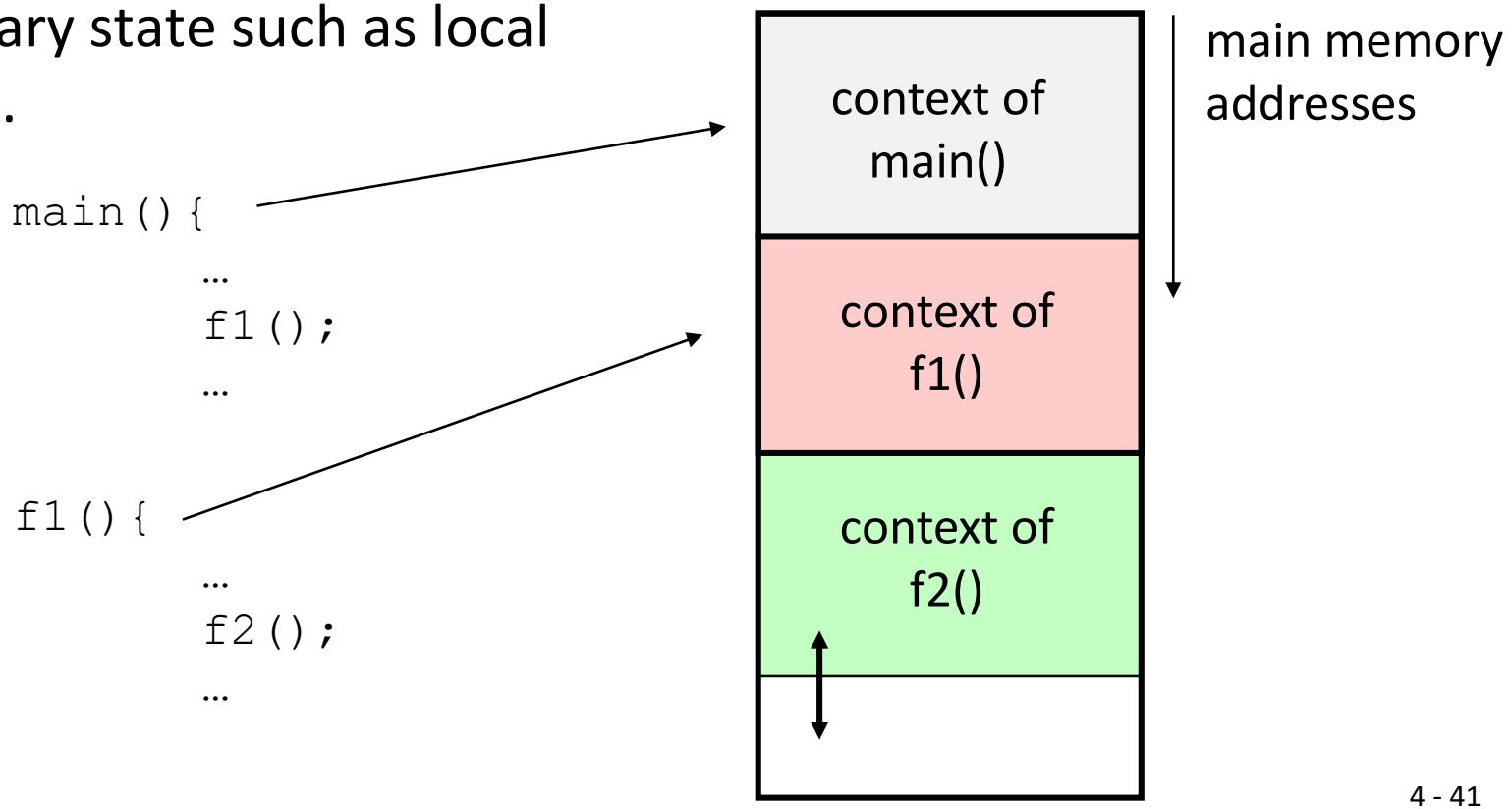
***Properties:***

- *communication between tasks* does not lead to a simultaneous access to shared resources, but interrupts may cause problems as they preempt running tasks
- *buffer overflow* may happen if too many events are generated by the environment or by tasks
- *tasks with a long running time* prevent other tasks from running and may cause buffer overflow as no events are being processed during this time
  - partition tasks into smaller ones
  - but the local context must be stored

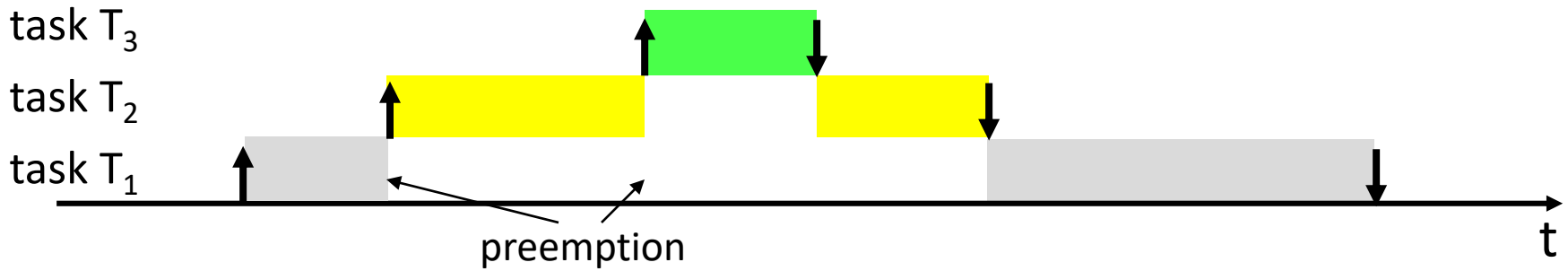


# Preemptive Event-Triggered Scheduling – Stack Policy

- This case is similar to non-preemptive case, but *tasks can be preempted by others*; this resolves partly the problem of tasks with a long execution time.
- If *the order of preemption is restricted*, we can use the usual stack-based context mechanism of function calls. The context of a function contains the necessary state such as local variables and saved registers.



# Preemptive Event-Triggered Scheduling – Stack Policy




- *Tasks must finish in LIFO (last in first out) order* of their instantiation.
  - this restricts flexibility of the approach
  - it is not useful, if tasks wait some unknown time for external events, i.e., they are blocked
- *Shared resources* (communication between tasks!) *must be protected*, for example by disabling interrupts or by the use of semaphores.

# Preemptive Event-Triggered Scheduling – Stack Policy

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

InsertEvent:
   put new event into event queue;
   select event from event queue;
   if (selected task ≠ running task) {
        execute selected task;
        remove selected event from queue;
   }
   return;

Interrupt:
   InsertEvent(…);
   return;
```

set CPU to low power mode; processing continues after interrupt

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

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

# Thread

- *A thread is a unique execution of a program.*
  - Several copies of such a “program” may run simultaneously or at different times.
  - Threads share the same processor and its peripherals.

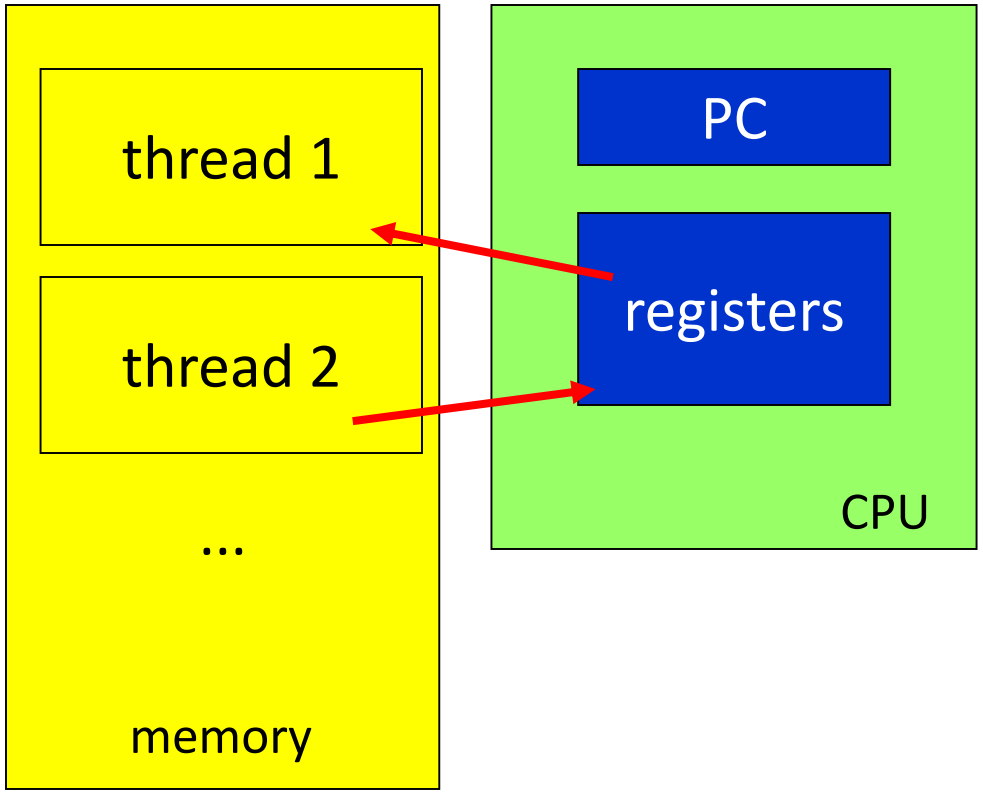
- *A thread has its own local state.* This state consists mainly of:
  - register values;
  - memory stack (local variables);
  - program counter;

- *Several threads may have a shared state* consisting of global variables.

# Threads and Memory Organization

- *Activation record* (also denoted as the thread context) contains the thread local state which includes registers and local data structures.

- *Context switch:*
  - current CPU context goes out
  - new CPU context goes in



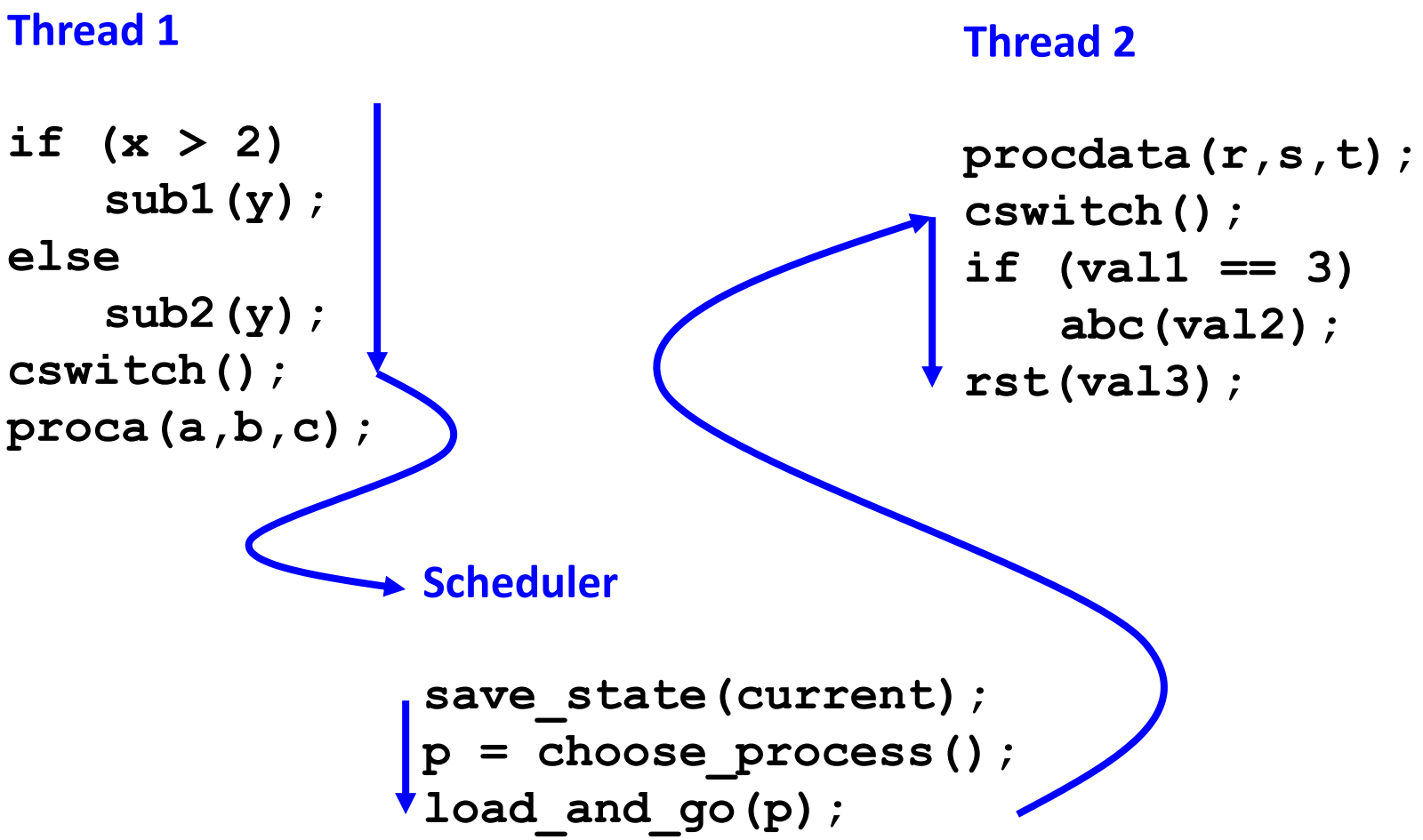
# Co-operative Multitasking

- *Each thread allows a context switch to another thread* at a call to the `cswitch()` function.
  - This function is part of the underlying runtime system (operating system).
  - A *scheduler* within this runtime system chooses which thread will run next.

- ***Advantages***:
  - predictable, where context switches can occur
  - less errors with use of shared resources if the switch locations are chosen carefully

- ***Problems***:
  - programming errors can keep other threads out as a thread may never give up CPU
  - real-time behavior may be at risk if a thread runs too long before the next context switch is allowed

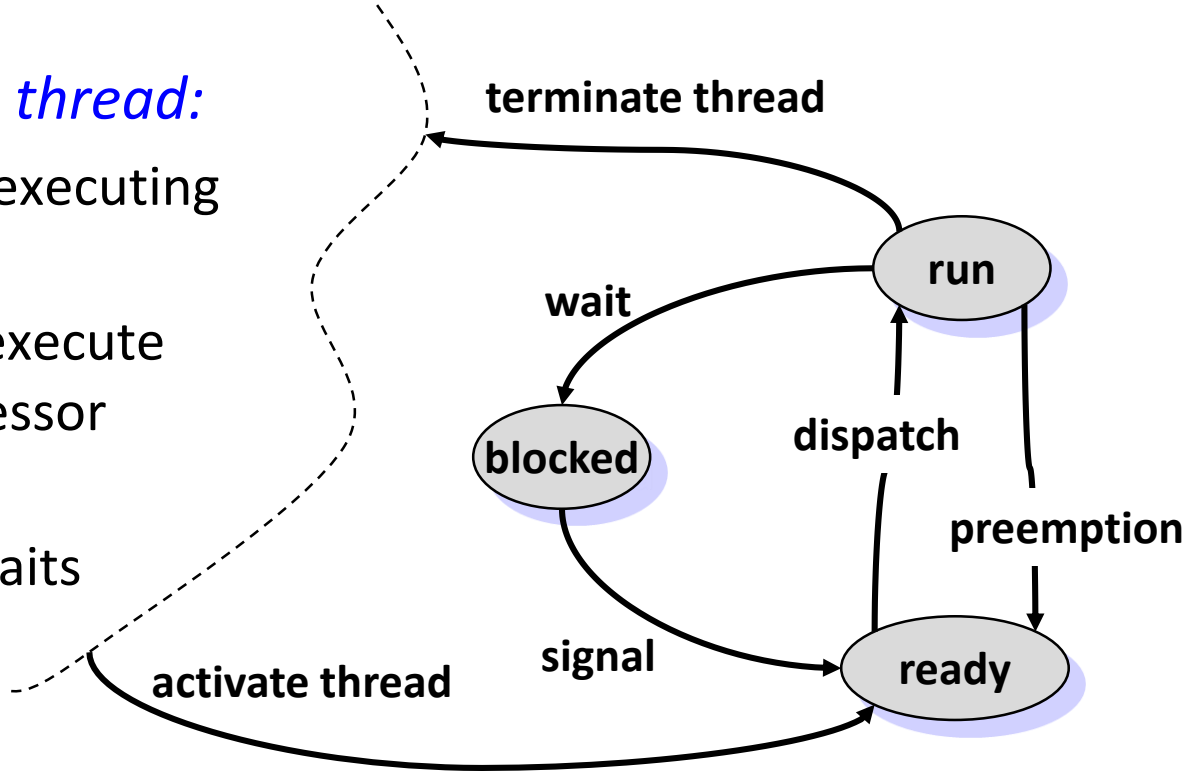
# Example: Co-operative Multitasking



# Preemptive Multitasking

- *Most general form of multitasking:*
  - The scheduler in the runtime system (operating system) controls when contexts switches take place.
  - The scheduler also determines what thread runs next.

- *State diagram corresponding to each single thread:*
  - *Run:* A thread enters this state as it starts executing on the processor
  - *Ready:* State of threads that are ready to execute but cannot be executed because the processor is assigned to another thread.
  - *Blocked*: A task enters this state when it waits for an event.



# Embedded Systems

## 4a. Timing Anomalies

© Lothar Thiele

Computer Engineering and Networks Laboratory

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# 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

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
```

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

```
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

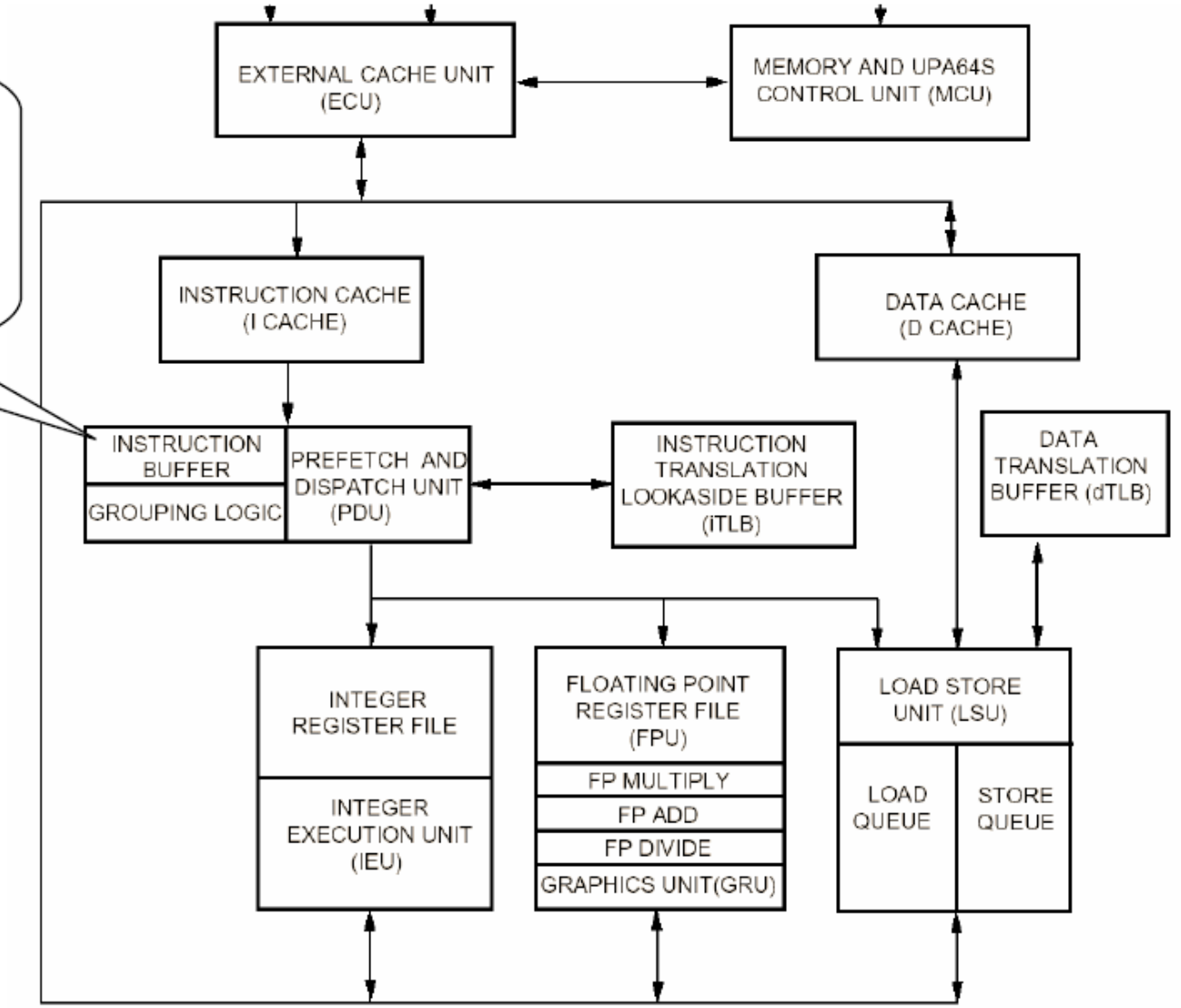
- The CPU is UltraSparc with 360 MHz clock rate.
- Problem 1 (ggt(17,17*97) ):
  - Fast execution: 96*3*1.000.000 / 0.35 = 823 MIPS and 0.35 * 360 / 96 = 1.31 cycles per iteration.
  - Slow execution: 96*3*1.000.000 / 0.65 = 443 MIPS and 0.65 * 360 / 96 = 2.44 cycles per iteration.
  - Therefore, the 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 (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



# User Manual (page 361 …)

## 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 (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.

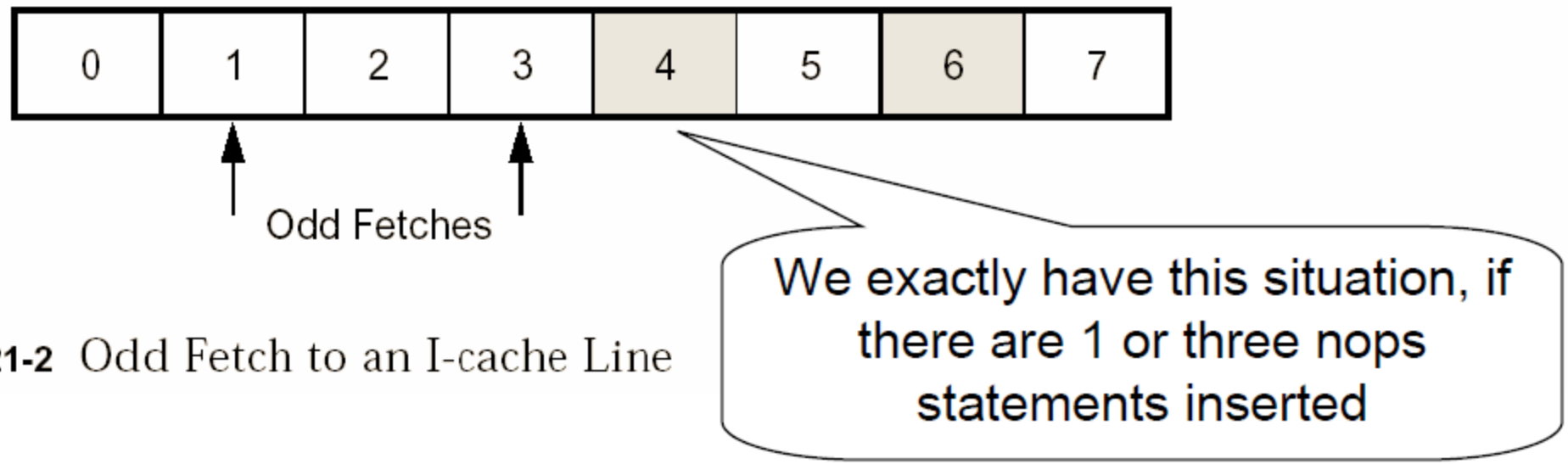



**FIGURE 21-2** Odd Fetch to an I-cache Line

# 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

## 5. Operating Systems

© Lothar Thiele

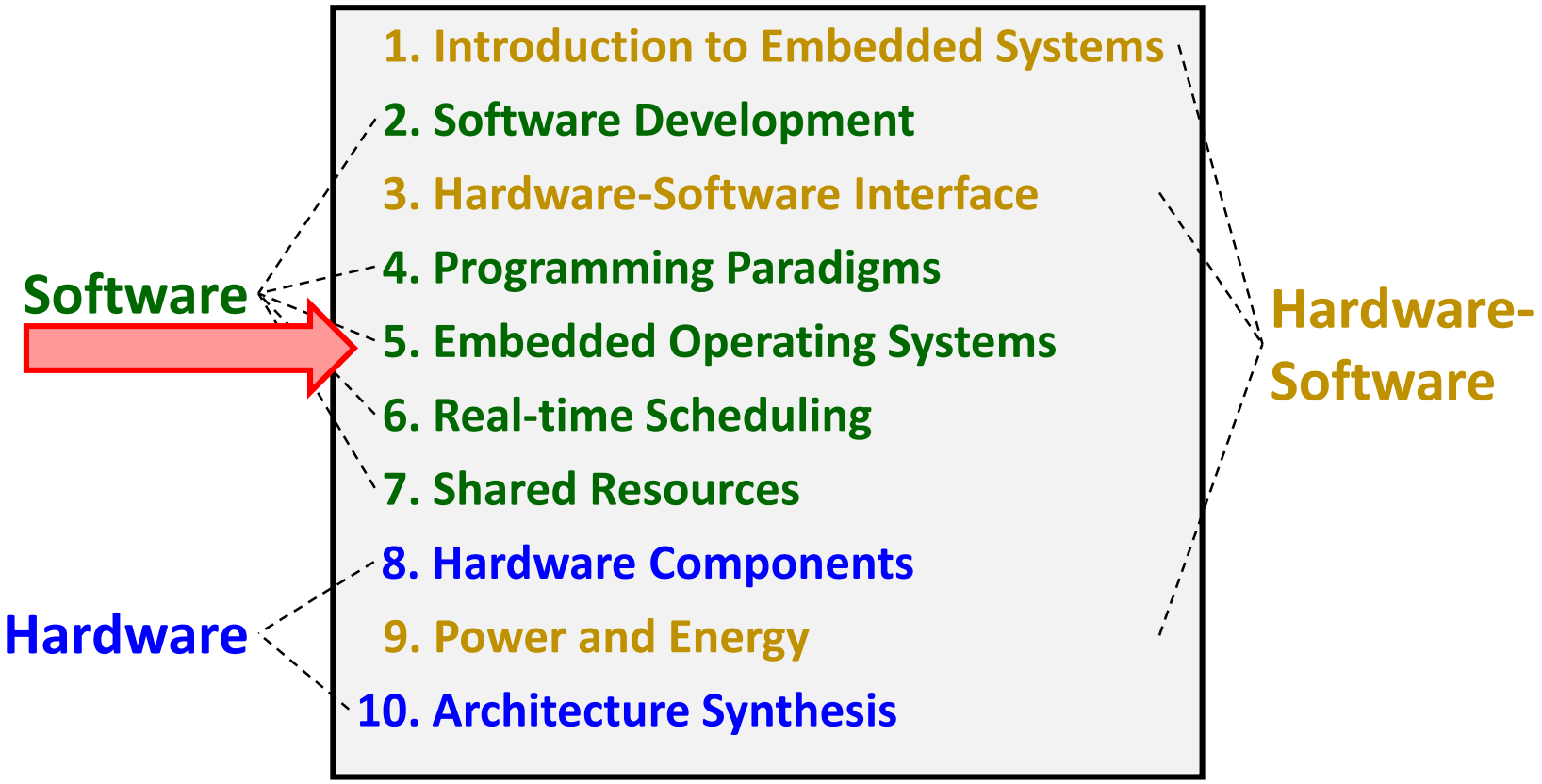
Computer Engineering and Networks Laboratory

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Embedded Operating Systems

# Where we are …


1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis
Software
Hardware
Hardware-
Software

# Embedded Operating System (OS)

- *Why an operating system (OS) at all?*
  - Same reasons why we need one for a traditional computer.
  - Not every devices needs all services.

- In embedded systems we find a *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 broad range of functionality (smart phone, …).

# Embedded Operating System

- *Why is a desktop OS not suited?*
  - The monolithic kernel of a desktop OS offers too many features that take space in memory and consume time.
  - Monolithic kernels are often not modular, fault-tolerant, configurable.
  - Requires too much memory space and is often too ressource hungry in terms of computation time.
  - Not designed for mission-critical applications.
  - The timing uncertainty may be too large for some applications.

# Embedded Operating Systems

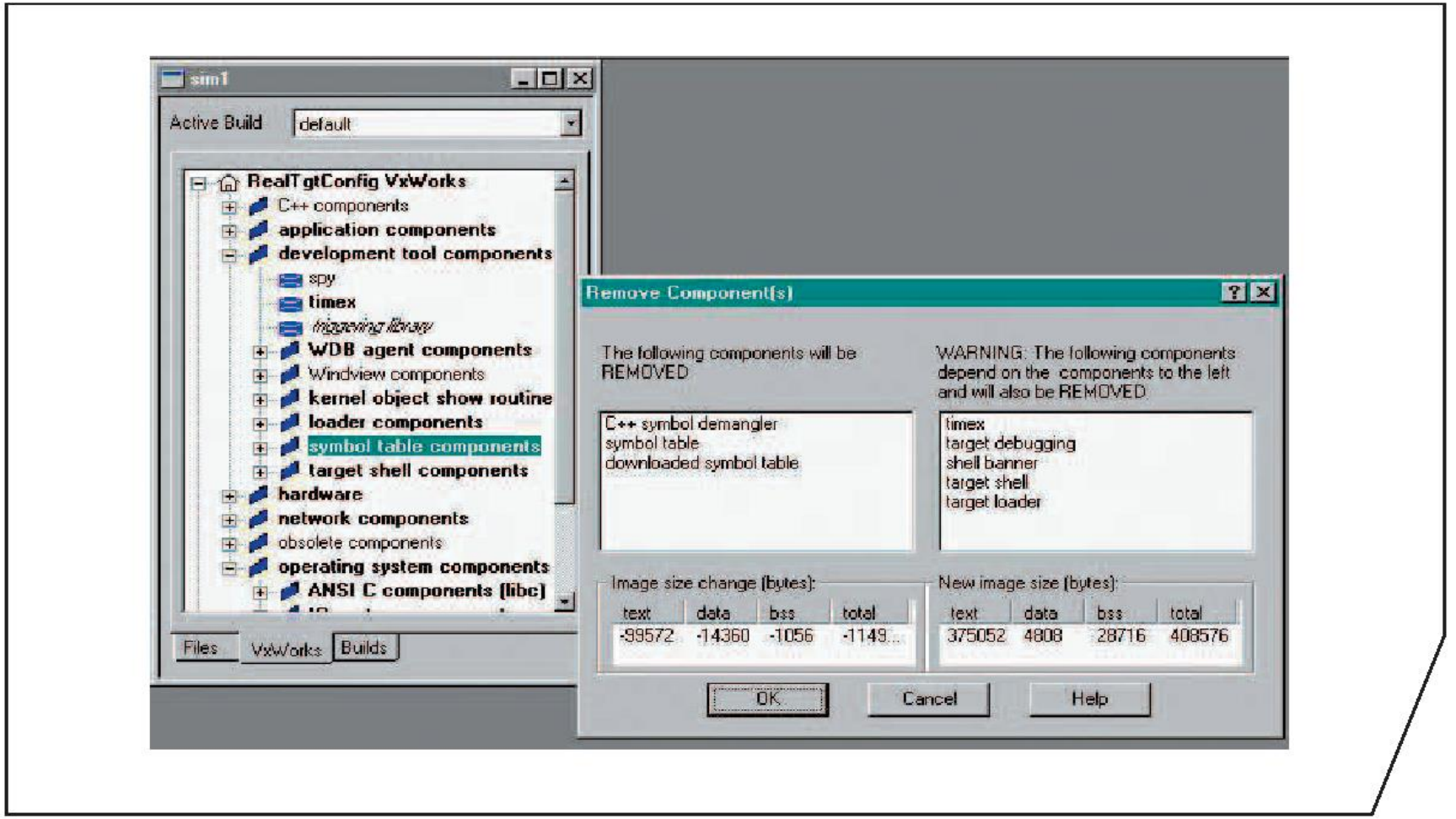
*Essential characteristics of an embedded OS:* Configurability

- *No single operating system will fit all needs*, but often no overhead for unused functions/data is tolerated. Therefore, configurability is needed.
- For example, there are many embedded systems without external memory, a keyboard, a screen or a mouse.

*Configurability examples:*

- *Remove unused functions*/libraries (for example by the linker).
- *Use conditional compilation* (using #if and #ifdef commands in C, for example).
- But deriving a consistent configuration is a potential problem of systems with a large number of derived operating systems. There is the danger of missing relevant components.

# Example: Configuration of VxWorks




**Automatic dependency analysis and size calculations allow users to quickly custom-tailor the VxWORKS operating system.**

© Windriver

http://www.windriver.com/products/development_tools/ide/tornado2/tornado_2_ds.pdf

# Real-time Operating Systems

***A real-time operating system is an operating system that supports the construction of real-time systems.***

*Key requirements:*

1. *The timing behavior of the OS must be predictable.*

   For all services of the OS, an upper bound on the execution time is necessary. For example, for every service upper bounds on blocking times need to be available, i.e. for times during which interrupts are disabled. Moreover, almost all processor activities should be controlled by a real-time scheduler.

2. *OS must manage the timing and scheduling*

   - OS has to be aware of deadlines and should have mechanism to take them into account in the scheduling
   - OS must provide precise time services with a high resolution

# Embedded Operating Systems

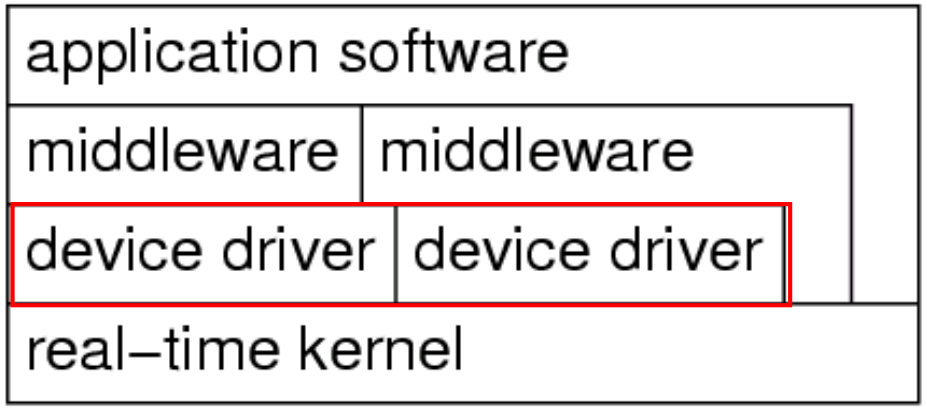
## Features and Architecture

# Embedded Operating System

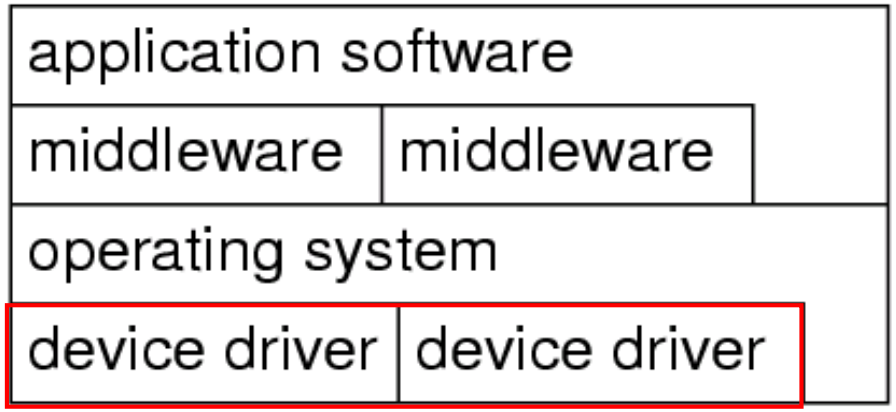
*Device drivers are typically handled directly by tasks* instead of drivers that are managed by the operating system:

- This architecture *improves timing predictability* as access to devices is also handled by the scheduler
- If several tasks use the same external device and the associated driver, then the access must be carefully managed (shared critical resource, ensure fairness of access)

Embedded OS




Standard OS



# Embedded Operating Systems

*Every task can perform an interrupt:*

- For *standard OS*, this would be *serious source of unreliability*. But embedded programs are typically programmed in a controlled environment.
- It is possible to let *interrupts directly start or stop tasks* (by storing the tasks start address in the interrupt table). This approach is more efficient and predictable than going through the operating system's interfaces and services.

*Protection mechanisms* are not always necessary in embedded operating systems:

- Embedded systems are typically designed for a single purpose, untested programs are rarely loaded, software can be considered to be reliable.
- However, protection mechanisms may be needed for *safety and security* reasons.

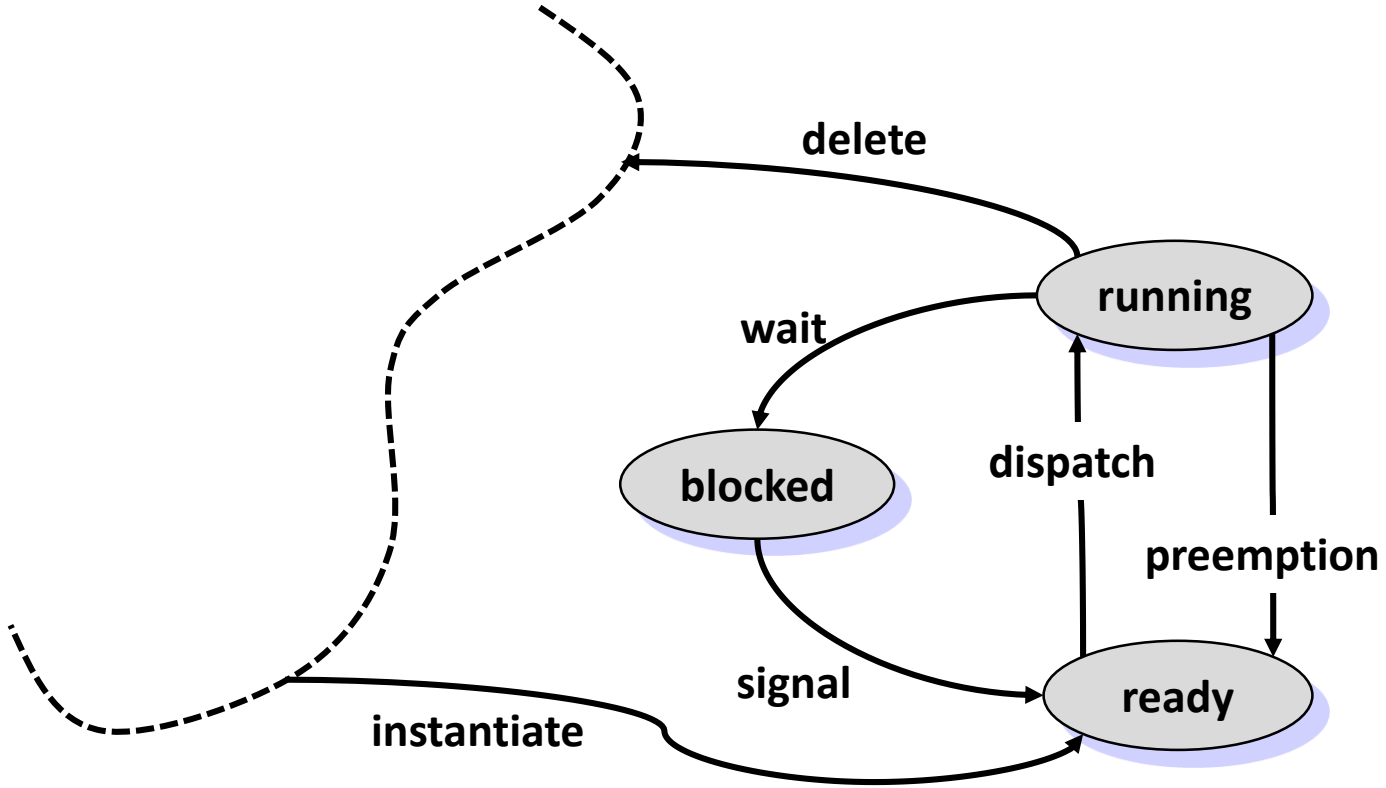
# Main Functionality of RTOS-Kernels

***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)
- *Inter-task communication* (buffering)
- *Support of real-time clocks*
- *Task synchronization* (critical sections, semaphores, monitors, mutual exclusion)
  - In classical operating systems, synchronization and mutual exclusion is performed via semaphores and monitors.
  - In real-time OS, special semaphores and a deep integration of them into scheduling is necessary (for example priority inheritance protocols as described in a later chapter).

# Task States

*Minimal Set of Task States:*



# Task states

*Running:*

- A task enters this state when it starts executing on the processor. There is as most one task with this state in the system.

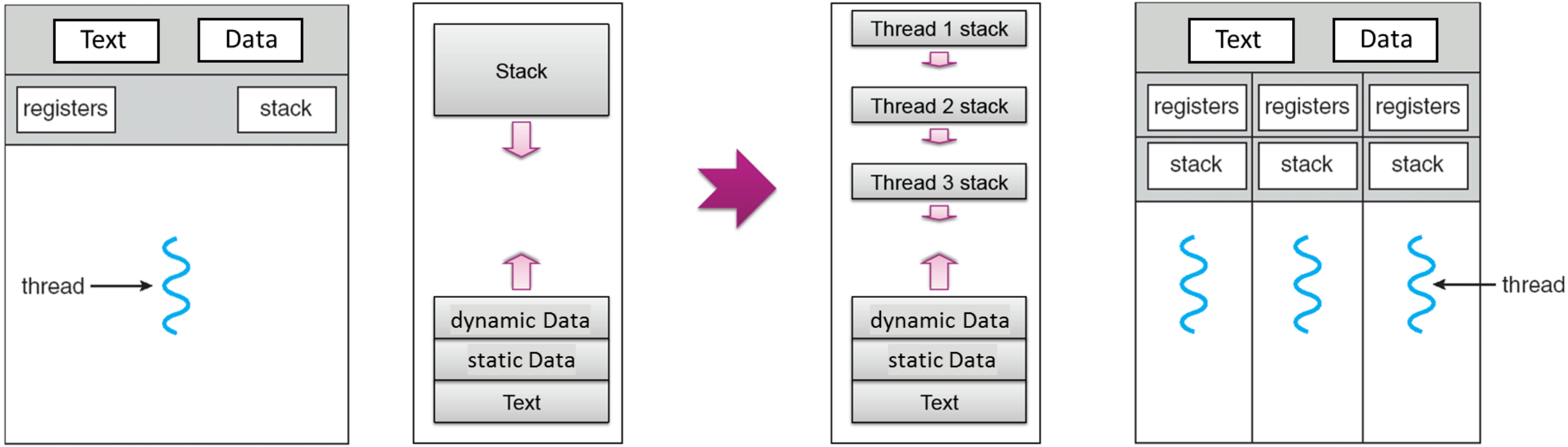
*Ready:*

- State of those tasks that are ready to execute but cannot be run because the processor is assigned to another task, i.e. another task has the state “running”.

*Blocked:*

- A task enters the blocked state when it executes a synchronization primitive to wait for an event, e.g. a wait primitive on a semaphore or timer. In this case, the task is inserted in a queue associated with this semaphore. The task at the head is resumed when the semaphore is unlocked by an event.

# Multiple Threads within a Process




**process with a single thread**

**process with several threads**

# Threads

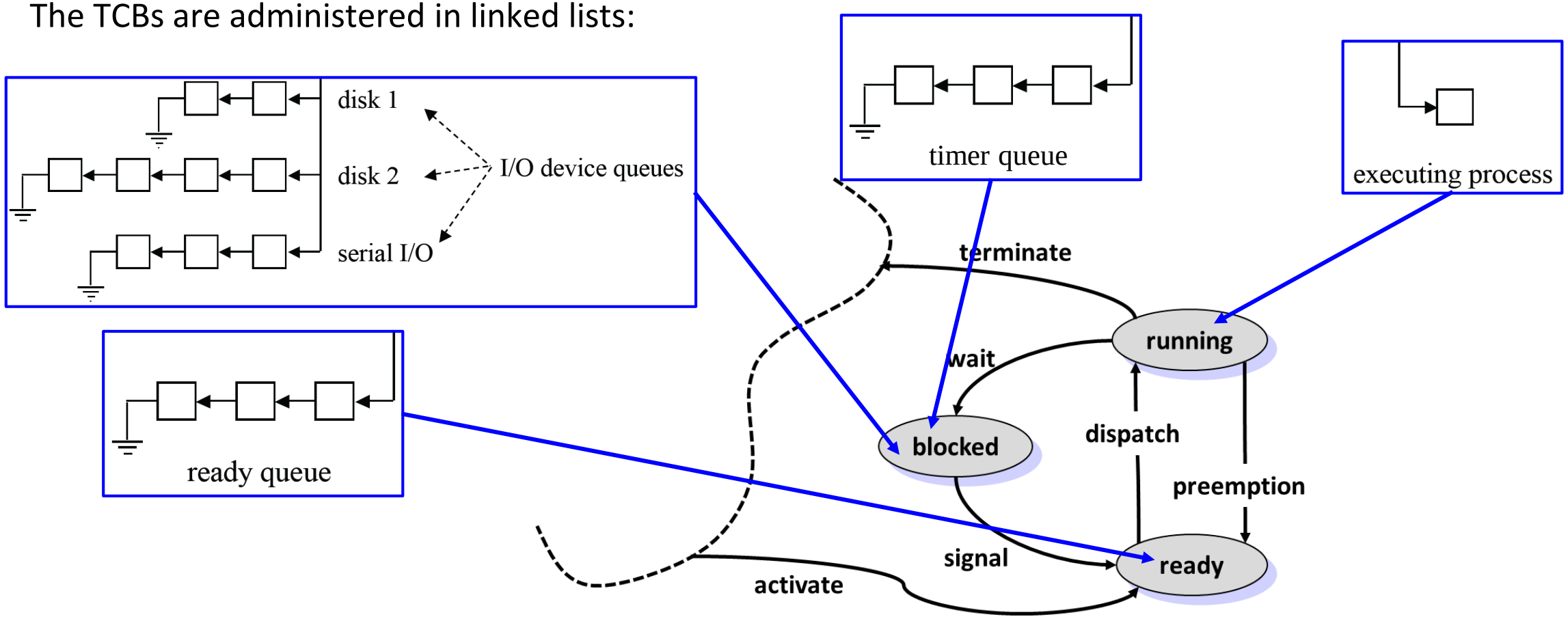
**A thread is the smallest sequence of programmed instructions that can be managed independently by a scheduler; e.g., a thread is a basic unit of CPU utilization.**

- *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.

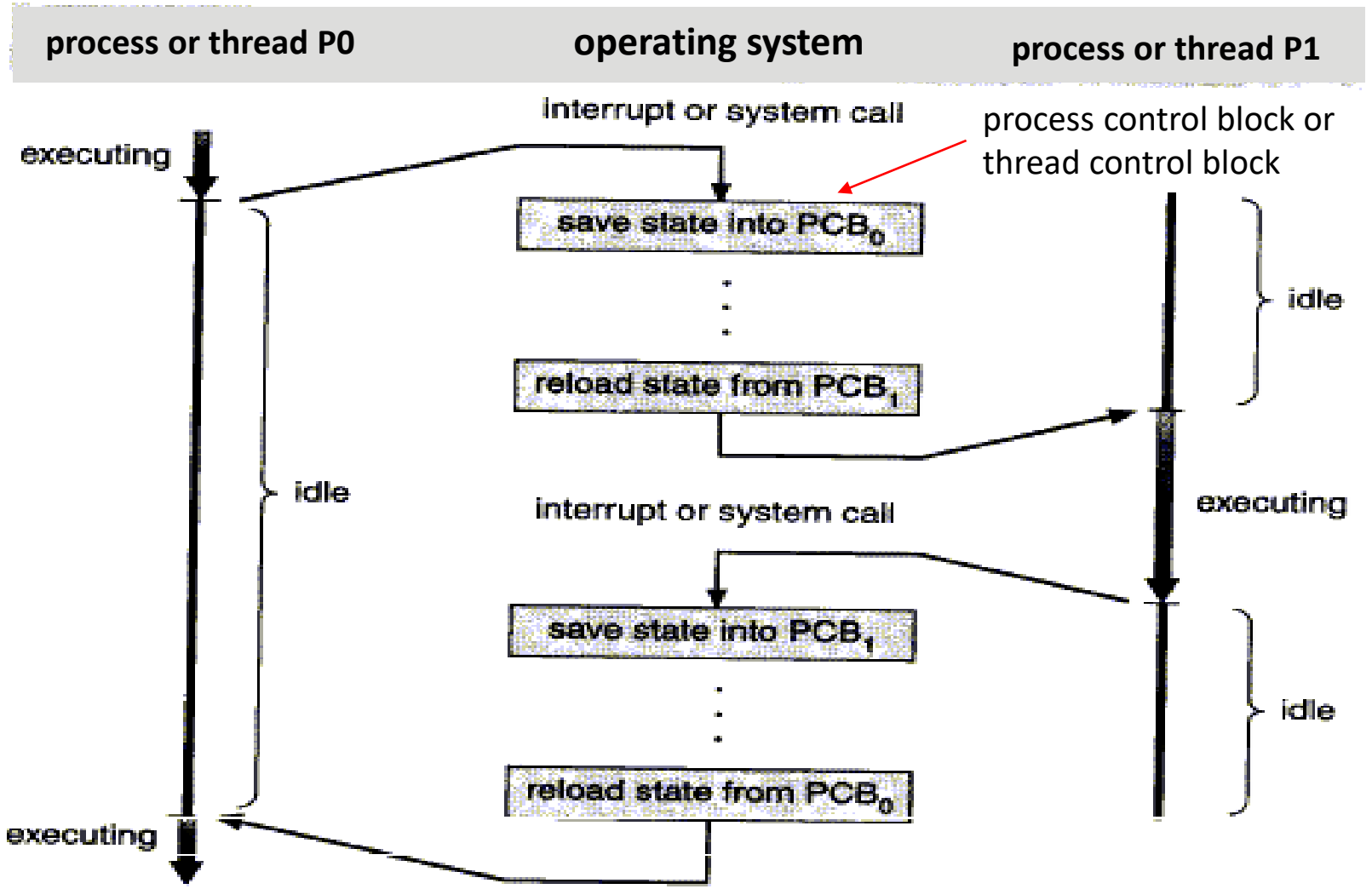
- *Thread advantages and characteristics*:
  - Faster to switch between threads; switching between user-level threads requires no major intervention by the operating system.
  - Typically, an application will have a separate thread for each distinct activity.
  - Thread Control Block (TCB) stores information needed to manage and schedule a thread

# Threads

- The *operating system* maintains for each thread a *data structure (TCB – thread control block)* that contains its current status such as program counter, priority, state, scheduling information, thread name.
- The TCBs are administered in linked lists:



# Context Switch: Processes or Threads



# Embedded Operating Systems

## Classes of Operating Systems

# Class 1: Fast and Efficient Kernels

## *Fast and efficient 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.

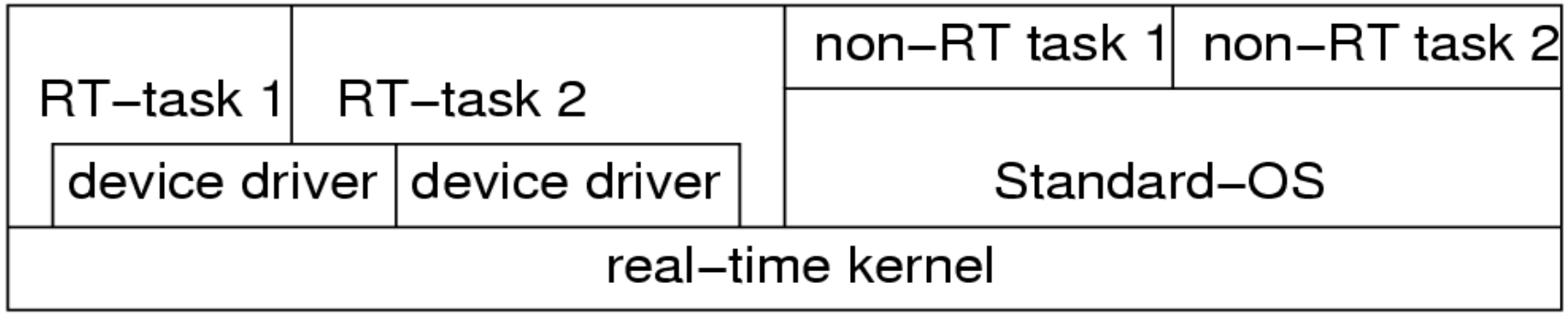
*Examples* include

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

# Class 2: Extensions to Standard OSs

*Real-time extensions to standard OS:*

- Attempt to exploit existing and comfortable main stream operating systems.
- A real-time kernel runs all real-time tasks.
- The standard-OS is executed as one task.

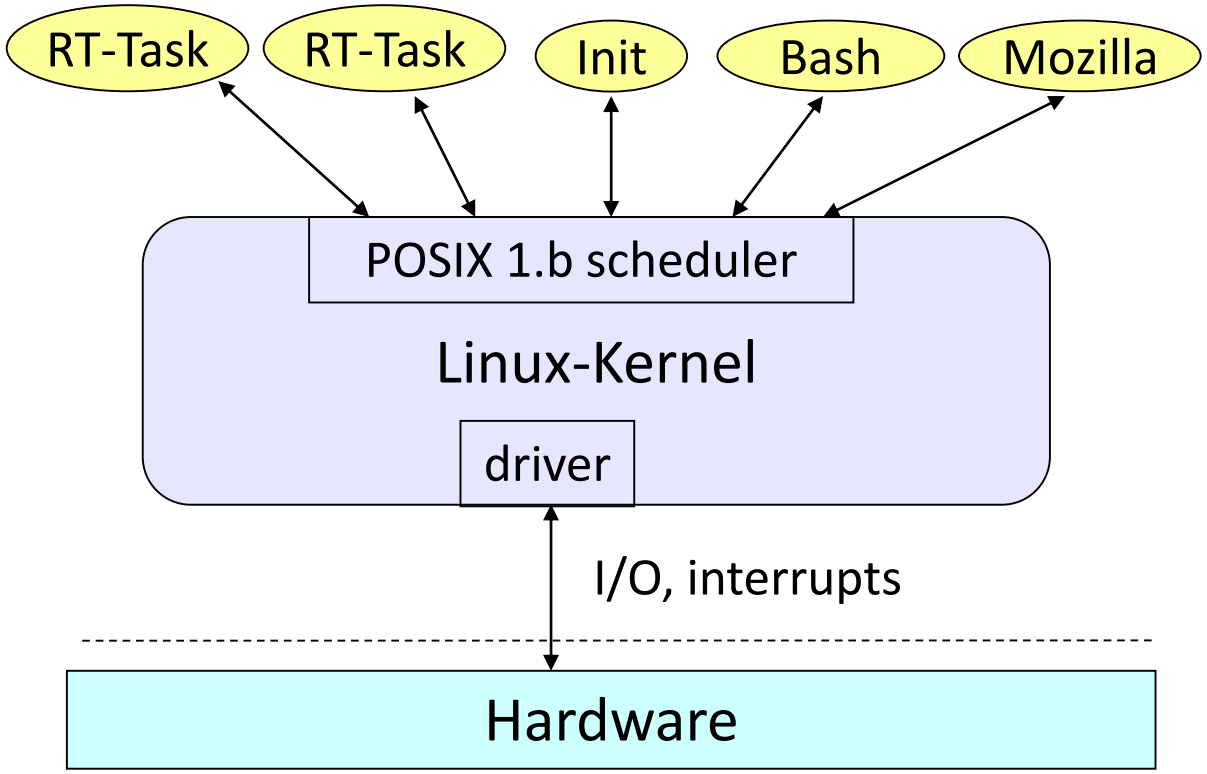



+ Crash of standard-OS does not affect RT-tasks;
- RT-tasks cannot use Standard-OS services;
  less comfortable than expected

revival of the concept: hypervisor

# Example: Posix 1.b RT-extensions to Linux

The standard scheduler of a general purpose operating system can be replaced by a scheduler that exhibits *(soft) real-time properties*.



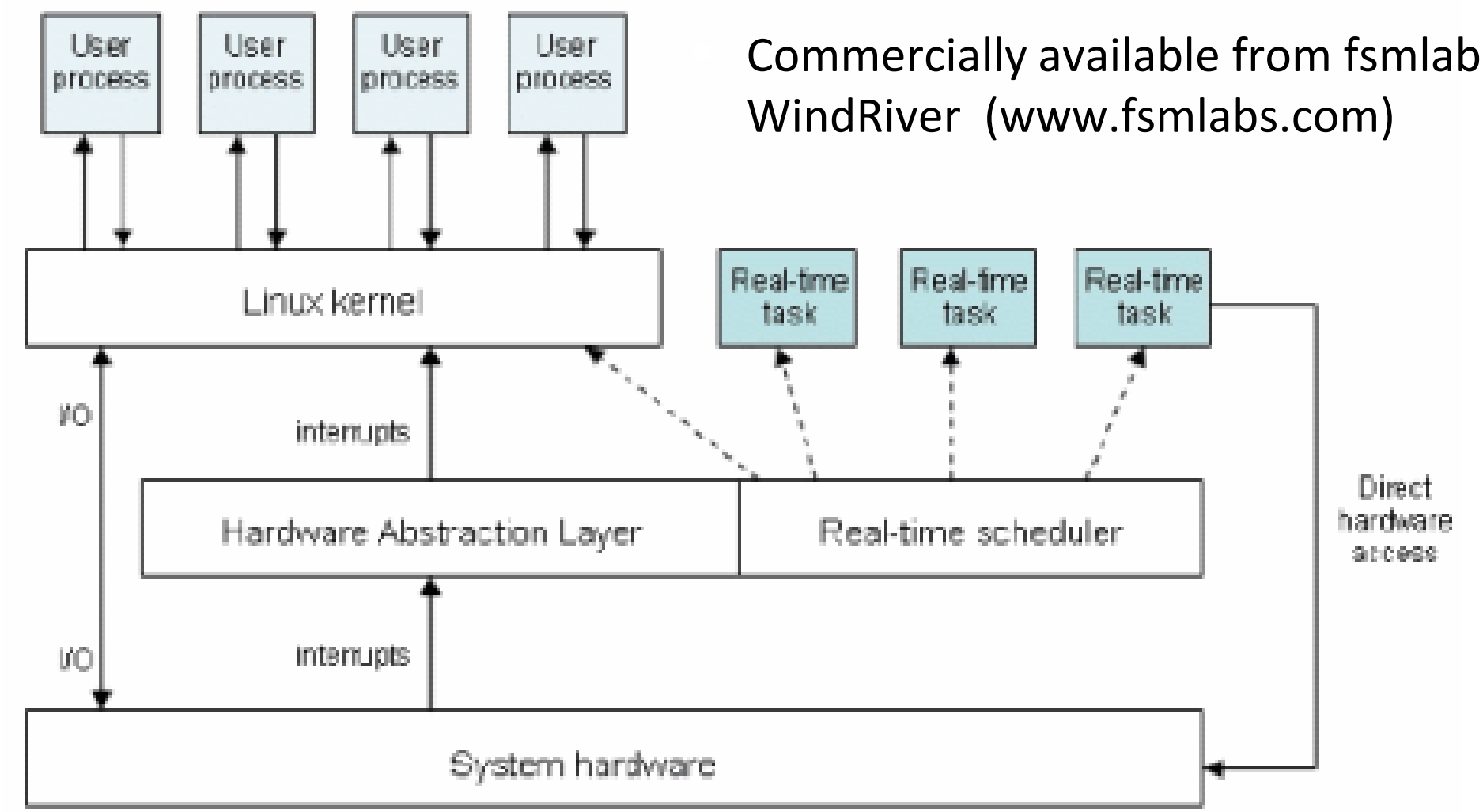

Special calls for real-time as well as standard operating system calls available.

Simplifies programming, but no guarantees for meeting deadlines are provided.

# Example: RT Linux

RT-tasks cannot use standard OS calls.

Commercially available from fsmlabs and WindRiver (www.fsmlabs.com)



# Class 3: Research Systems

***Research systems*** try to avoid limitations of existing real-time and embedded operating systems.

- Examples include L4, seL4, NICTA, ERIKA, SHARK

*Typical Research questions:*

- low overhead memory protection,
- temporal protection of computing resources
- RTOS 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

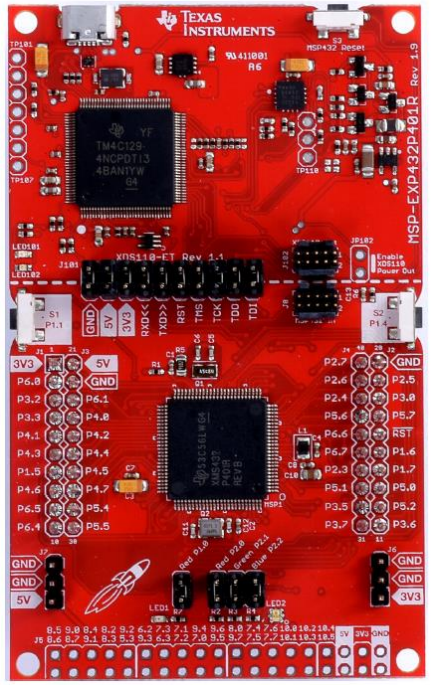
# Embedded Operating Systems

## FreeRTOS in the Embedded Systems Lab (ES-Lab)

# Example: FreeRTOS (ES-Lab)

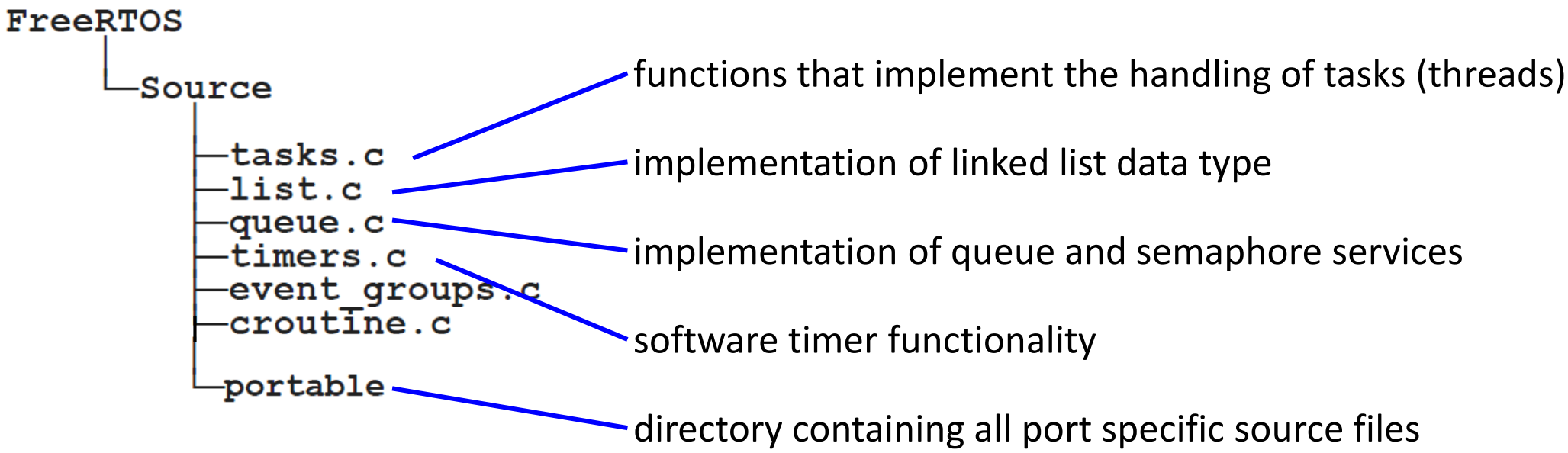
***FreeRTOS*** (http://www.freertos.org/) is a typical embedded operating system. It is available for many hardware platforms, open source and widely used in industry. It is used in the ES-Lab.

- FreeRTOS is a *real-time kernel* (or real-time scheduler).
- Applications are organized as a *collection of independent threads* of execution.
- *Characteristics:* Pre-emptive or co-operative operation, queues, binary semaphores, counting semaphores, mutexes (mutual exclusion), software timers, stack overflow checking, trace recording, … .



# Example: FreeRTOS (ES-Lab)

*Typical directory structure* (excerpts):




- *FreeRTOS is configured* by a header file called `FreeRTOSConfig.h` that determines almost all configurations (co-operative scheduling vs. preemptive, time-slicing, heap size, mutex, semaphores, priority levels, timers, …)

# Embedded Operating Systems

## FreeRTOS Task Management

# Example FreeRTOS – Task Management

***Tasks are implemented as threads.***

- The *functionality of a thread* is implemented in form of a *function:*
  - Prototype:

```
void ATaskFunction( void *pvParameters );
```

some name of task function — pointer to task arguments

  - Task functions are not allowed to return! They can be "killed" by a specific call to a FreeRTOS function, but usually run forever in an infinite loop.
  - Task functions can instantiate other tasks. Each created task is a separate execution instance, with its own stack.
  - *Example:*

```
void vTask1( void *pvParameters ) {
   volatile uint32_t ul; /* volatile to ensure ul is implemented. */
   for( ;; ) {
     ... /* do something repeatedly */
     for( ul = 0; ul < 10000; ul++ ) {  /* delay by busy loop */ }
   }
}
```

# Example FreeRTOS – Task Management

- *Thread instantiation:*

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

returns pdPASS or pdFAIL depending on the success of the thread creation

a pointer to the function that implements the task

a descriptive name for the task

each task has its own unique stack that is allocated by the kernel to the task when the task is created; the usStackDepth value determines the size of the stack (in words)

the priority at which the task will execute; priority 0 is the lowest priority

pxCreatedTask can be used to pass out a handle to the task being created.

task functions accept a parameter of type pointer to void; the value assigned to pvParameters is the value passed into the task.

# Example FreeRTOS – Task Management

***Examples** for changing properties of tasks:*

- Changing the *priority* of a task. In case of preemptive scheduling policy, the ready task with the highest priority is automatically assigned to the “running” state.

```
void vTaskPrioritySet( TaskHandle_t pxTask, UBaseType_t uxNewPriority );
```

handle of the task whose priority is being modified

new priority (0 is lowest priority)

- A task can *delete* itself or any other task. Deleted tasks no longer exist and cannot enter the “running” state again.

```
void vTaskDelete( TaskHandle_t pxTaskToDelete );
```

handle of the task who will be deleted; if NULL, then the caller will be deleted

# Embedded Operating Systems

## FreeRTOS Timers

# Example FreeRTOS – Timers

- The operating system also provides *interfaces to **timers*** of the processor.
- As an example, we use the FreeRTOS timer interface to replace the busy loop by a delay. In this case, the task is put into the “blocked” state instead of continuously running.

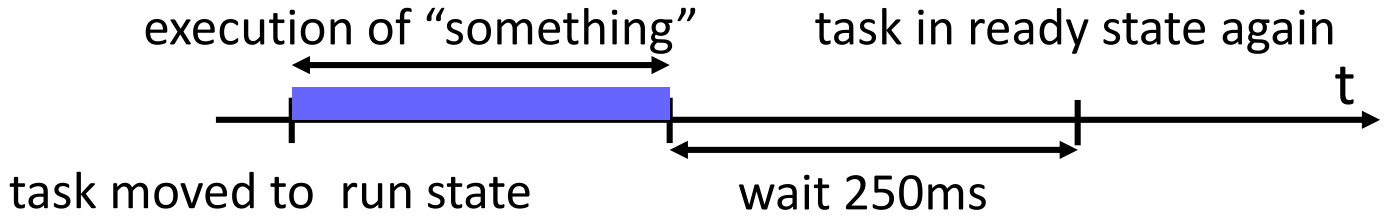
```
void vTaskDelay( TickType_t xTicksToDelay );
```

time is measured in “tick” units that are defined in the configuration of FreeRTOS (`FreeRTOSConfig.h`). The function `pdMS_TO_TICKS()` converts ms to “ticks”.

```
void vTask1( void *pvParameters ) {
   for( ;; ) {
     ... /* do something repeatedly */
     vTaskDelay(pdMS_TO_TICKS(250));  /* delay by 250 ms */
   }
 }
```

# Example FreeRTOS – Timers

- *Problem:* The task *does not execute* strictly *periodically*:




- The parameters to vTaskDelayUntil() specify the exact tick count value at which the calling task should be moved from the “blocked” state into the “ready” state. Therefore, the task is put into the “ready” state periodically.

```
void vTask1( void *pvParameters ) {
   TickType_t xLastWakeTime = xTaskGetTickCount();
   for( ;; ) {
     ... /* do something repeatedly */
     vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(250));
   }
}
```

automatically updated when task is unblocked

time to next unblocking

The xLastWakeTime variable needs to be initialized with the current tick count. Note that this is the only time the variable is written to explicitly. After this xLastWakeTime is automatically updated within vTaskDelayUntil().

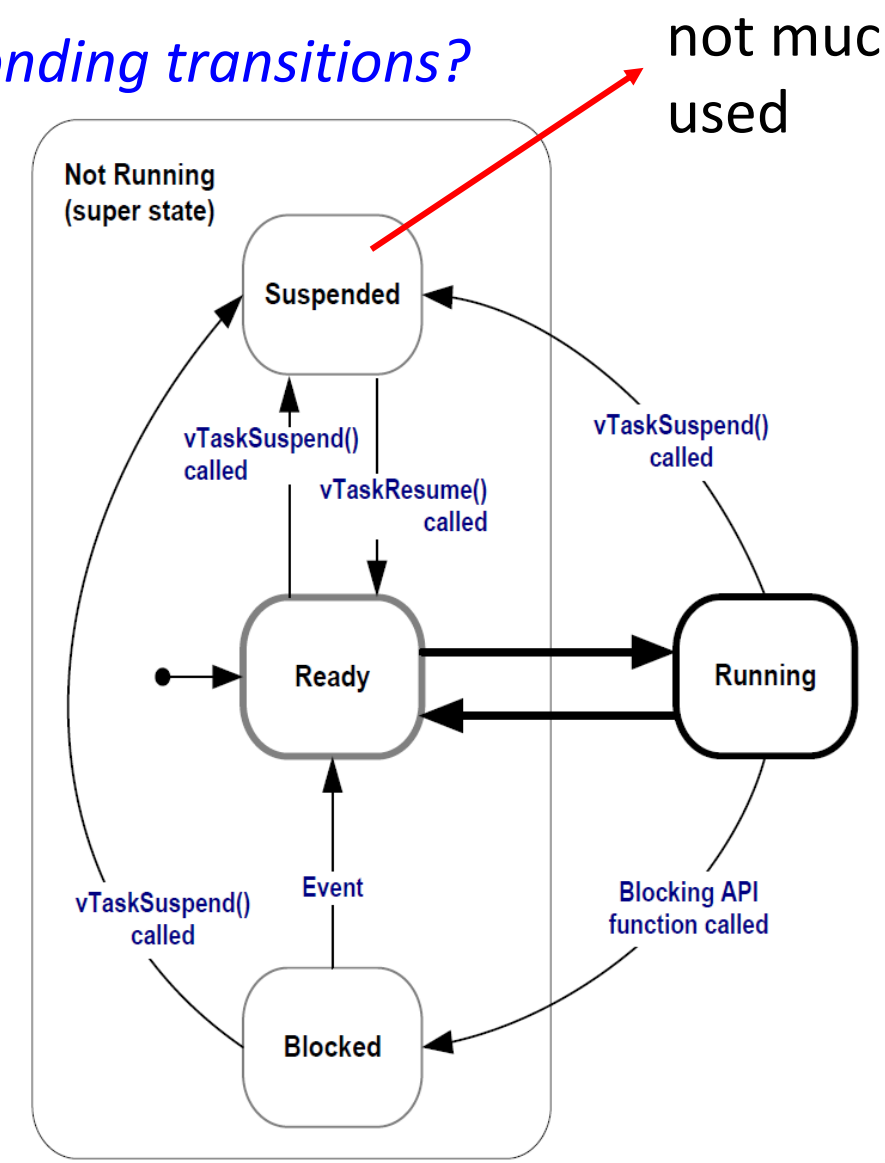
# Embedded Operating Systems

## FreeRTOS Task States

# Example FreeRTOS – Task States

*What are the task states in FreeRTOS and the corresponding transitions?*

- A task that is waiting for an event is said to be in the *"Blocked" state*, which is a sub-state of the *"Not Running" state*.
- Tasks can enter the "Blocked" state to wait for two different types of event:
  - *Temporal (time-related) events*—the event being either a delay period expiring, or an absolute time being reached.
  - *Synchronization events*—where the events originate from another task or interrupt. For example, queues, semaphores, and mutexes, can be used to create synchronization events.



# Example FreeRTOS – Task States

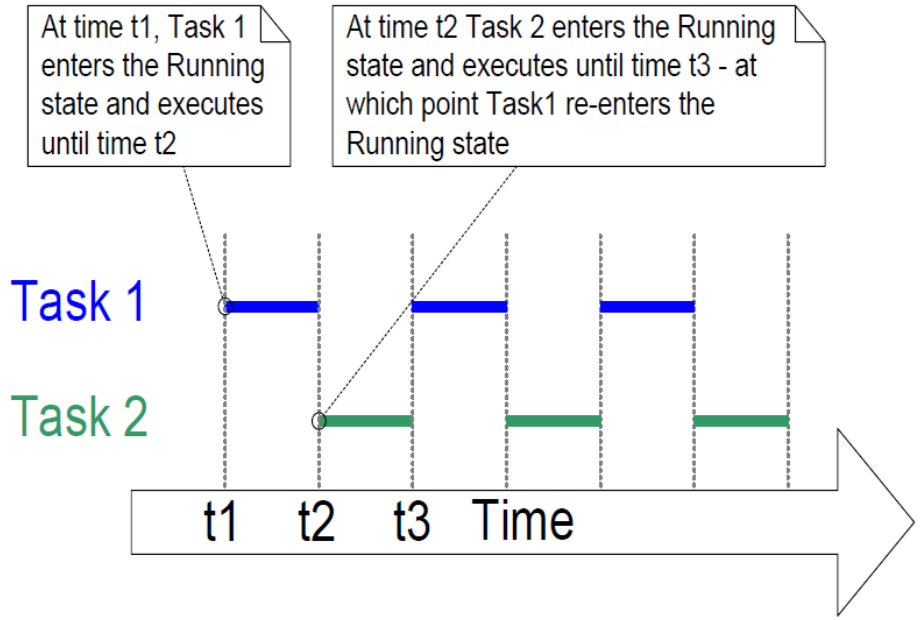
***Example 1:*** *Two threads with equal priority.*

```
void vTask1( void *pvParameters ) {
   volatile uint32_t ul;
   for( ;; ) {
     ... /* do something repeatedly */
     for( ul = 0; ul < 10000; ul++ ) { }
   }
}
```

```
void vTask2( void *pvParameters ) {
   volatile uint32_t u2;
   for( ;; ) {
     ... /* do something repeatedly */
     for( u2 = 0; u2 < 10000; u2++ ) { }
   }
}
```

```
int main( void ) {
   xTaskCreate(vTask1, "Task 1", 1000, NULL, 1, NULL);
   xTaskCreate(vTask2, "Task 2", 1000, NULL, 1, NULL);
   vTaskStartScheduler();
   for( ;; );
}
```

Both tasks have priority 1. In this case, FreeRTOS uses time slicing, i.e., every task is put into “running” state in turn.



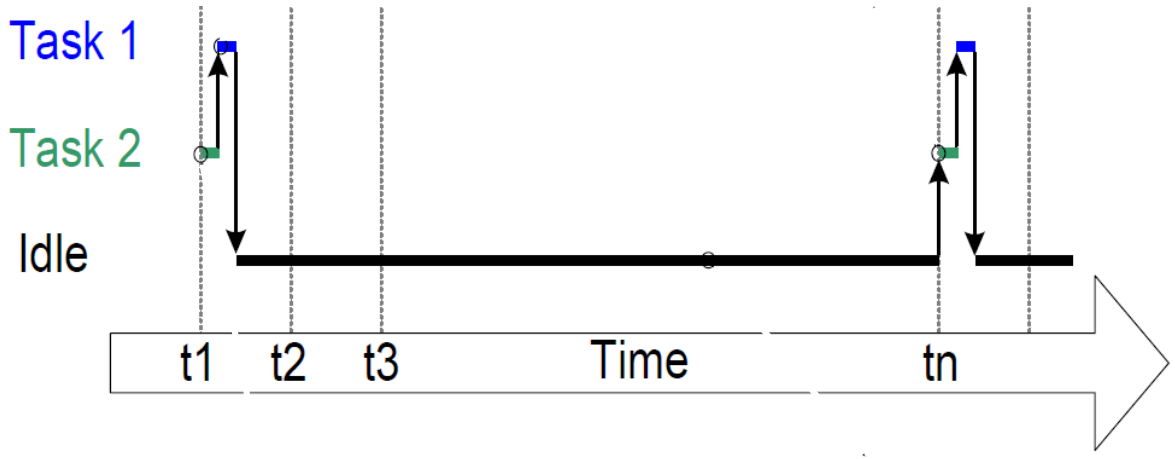
# Example FreeRTOS – Task States

***Example 2:*** *Two threads with delay timer.*

```
void vTask1( void *pvParameters ) {
   TickType_t xLastWakeTime = xTaskGetTickCount();
   for( ;; ) {
     ... /* do something repeatedly */
     vTaskDelayUntil(&xLastWakeTime,pdMS_TO_TICKS(250));
   }
}
```

```
void vTask2( void *pvParameters ) {
   TickType_t xLastWakeTime = xTaskGetTickCount();
   for( ;; ) {
     ... /* do something repeatedly */
     vTaskDelayUntil(&xLastWakeTime,pdMS_TO_TICKS(250));
   }
}
```

```
int main( void ) {
  xTaskCreate(vTask1,"Task 1",1000,NULL,1,NULL);
  xTaskCreate(vTask2,"Task 2",1000,NULL,2,NULL);
  vTaskStartScheduler();
  for( ;; );
}
```




If no user-defined task is in the running state, FreeRTOS chooses a built-in Idle task with priority 0. One can associate a function to this task, e.g., in order to go to low power processor state.

# Embedded Operating Systems

## FreeRTOS Interrupts

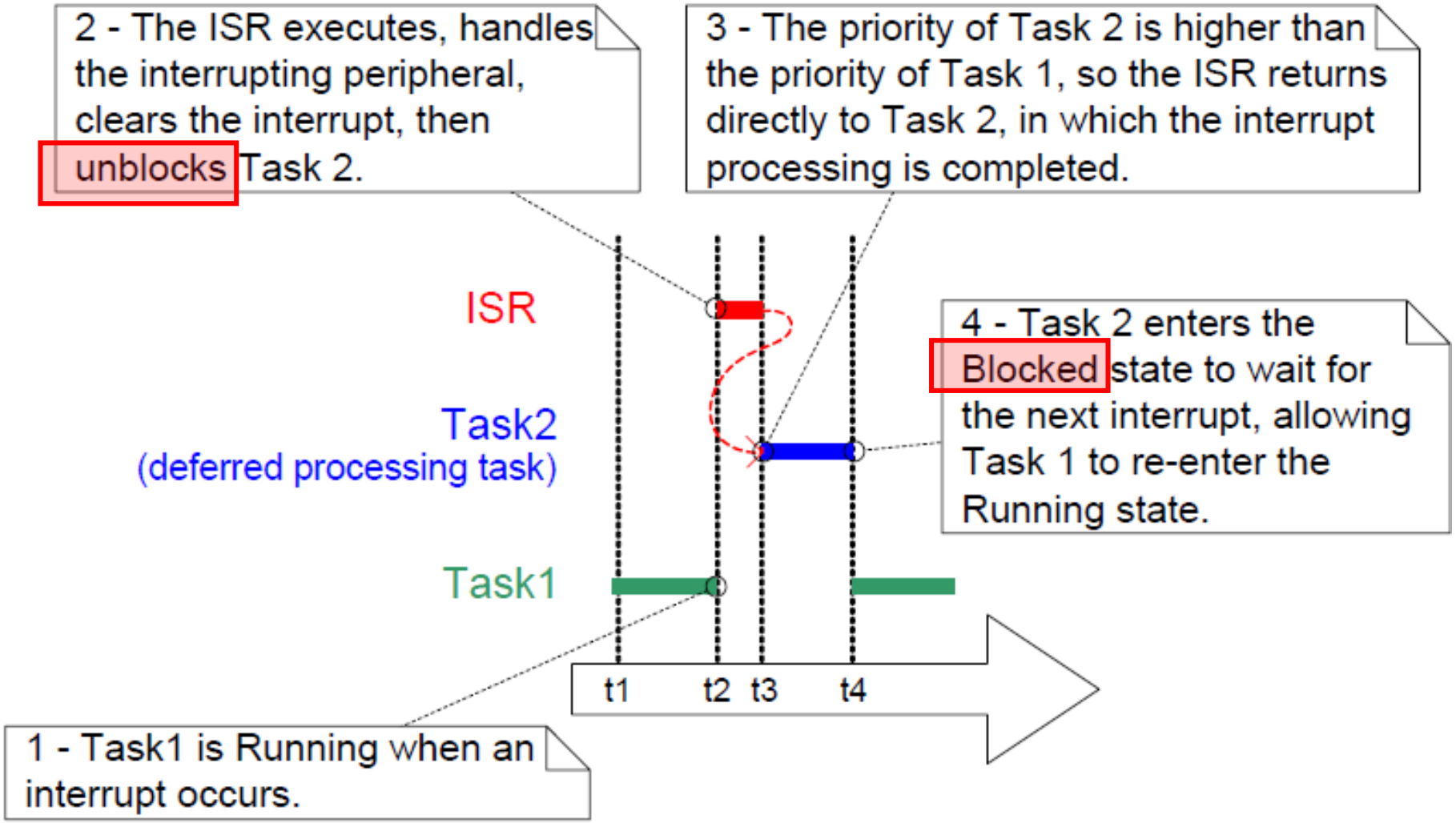
# Example FreeRTOS – Interrupts

*How are tasks (threads) and hardware interrupts scheduled jointly?*

- Although written in software, an *interrupt service routine (ISR)* is a hardware feature because the hardware controls which interrupt service routine will run, and when it will run.
- *Tasks will only run when there are no ISRs running*, so the lowest priority interrupt will interrupt the highest priority task, and there is no way for a task to pre-empt an ISR. In other words, ISRs have always a higher priority than any other task.

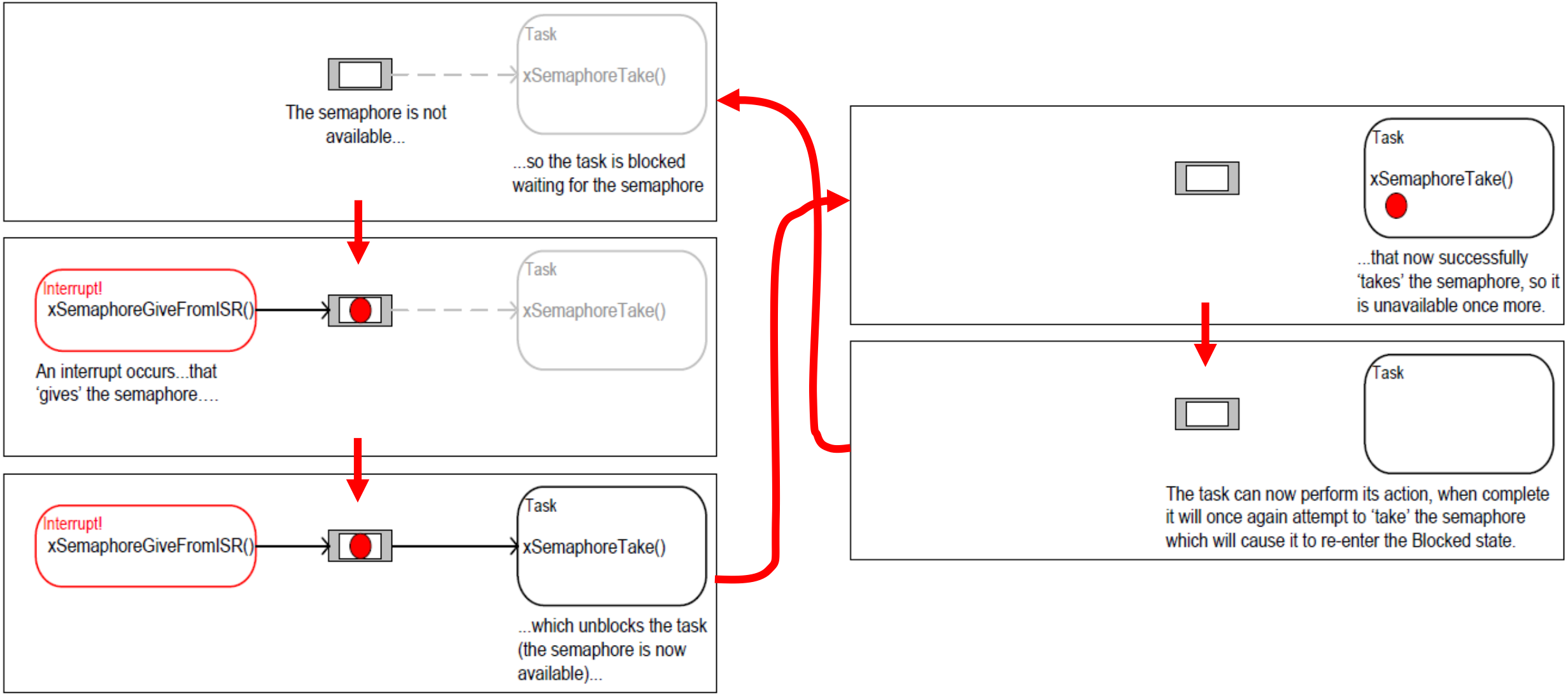
- *Usual pattern:*
  - ISRs are usually very short. They find out the reason for the interrupt, clear the interrupt flag and determine what to do in order to handle the interrupt.
  - Then, they unblock a regular task (thread) that performs the necessary processing related to the interrupt.
  - For blocking and unblocking, usually semaphores are used.

# Example FreeRTOS – Interrupts




blocking and unblocking is typically implemented via semaphores

# Example FreeRTOS – Interrupts



# Embedded Systems

## 6. Aperiodic and Periodic Scheduling

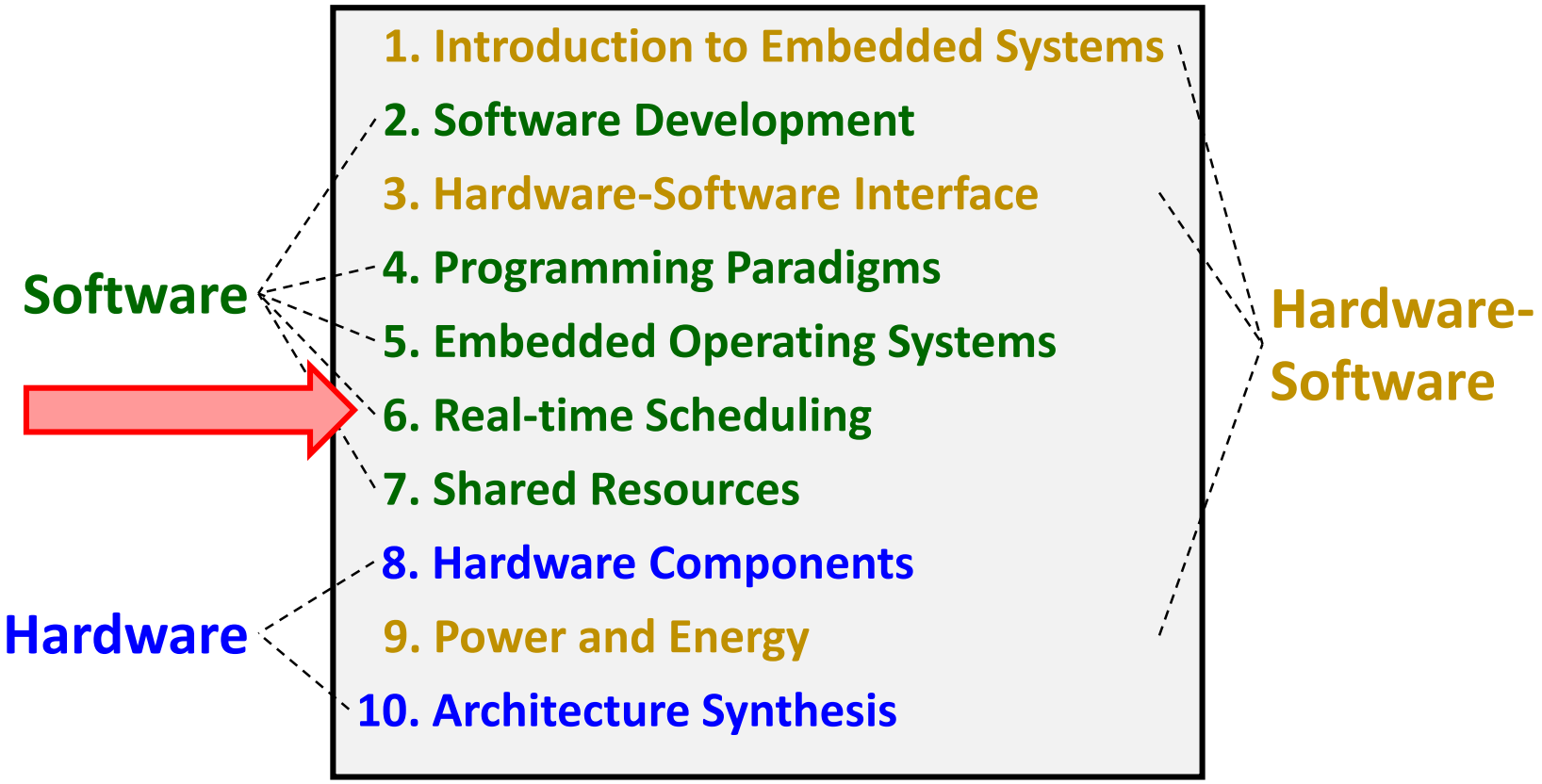
© Lothar Thiele

Computer Engineering and Networks Laboratory

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Where we are …


1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis
Software
Hardware
Hardware-
Software

# Basic Terms and Models

# 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

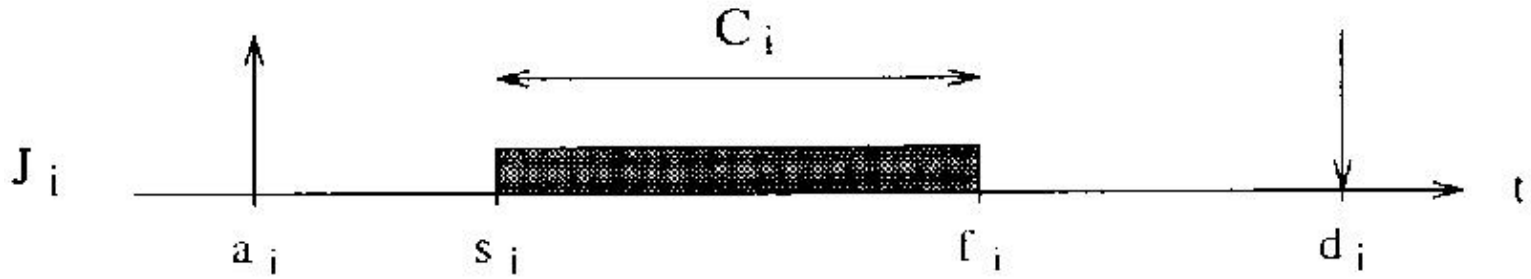
Given a *set of tasks* $J = \{J_1, J_2, \ldots\}$:

- 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.

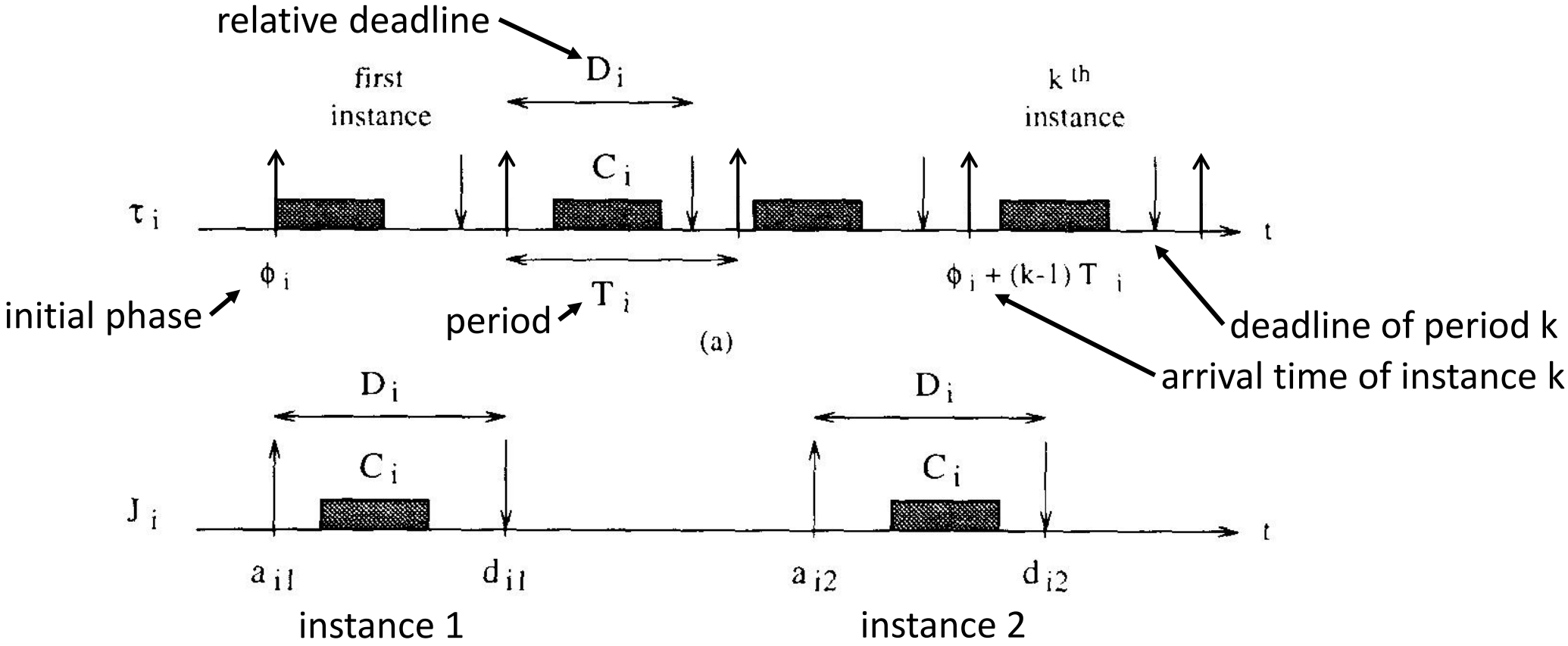
# Schedule and Timing



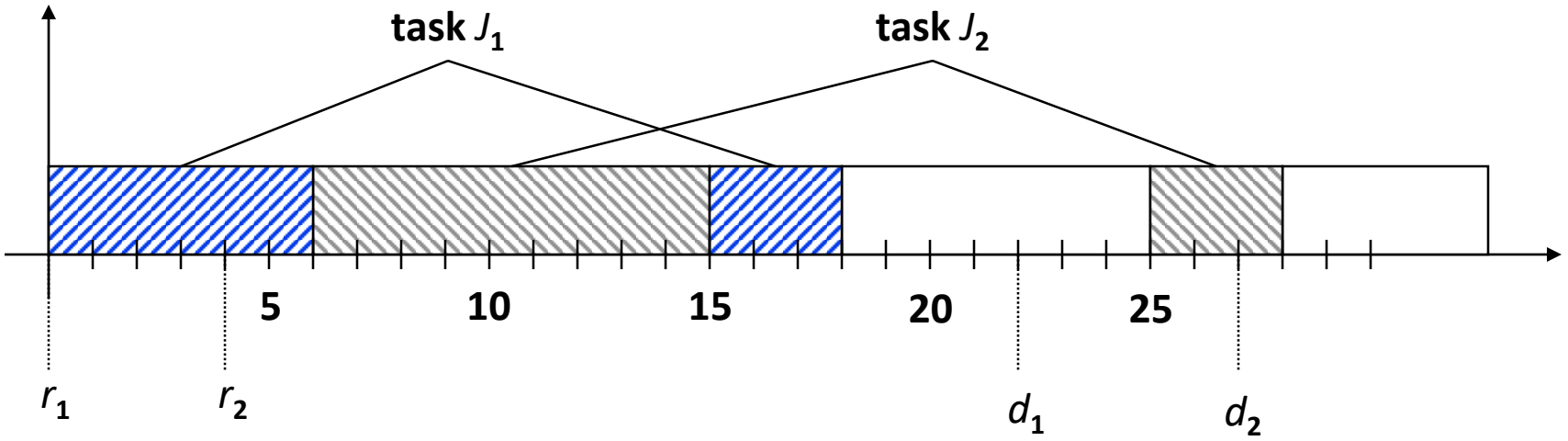

- Using the above definitions, we have $d_i \geq r_i + C_i$
- *Lateness* $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline; note that if a task completes before the deadline, its lateness is negative.
- *Tardiness or exceeding time* $E_i = \max(0, L_i)$ is the time 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* $\tau_i$ : 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* $\Phi_i$ .



# Example for Real-Time Model




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

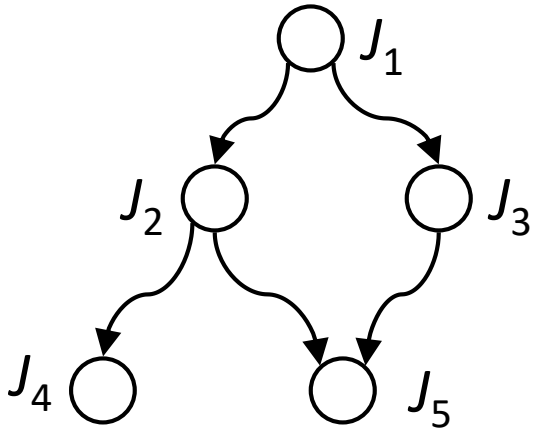
Tardiness: $E_1 = 0$, $E_2 = 1$

Laxity: $X_1 = 13$, $X_2 = 11$

# Precedence Constraints

- *Precedence relations* between tasks 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.

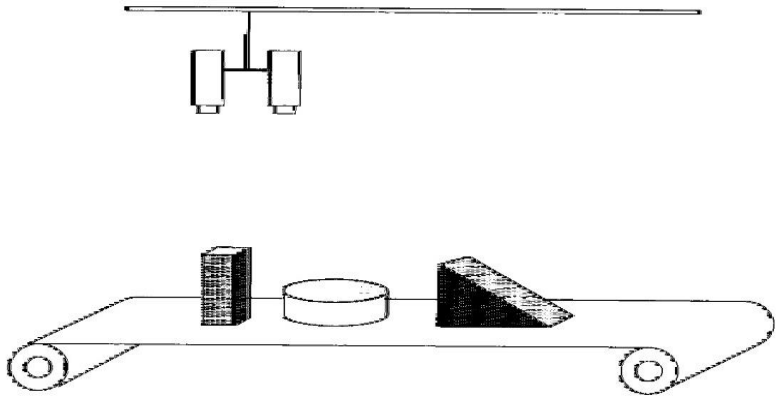
- There are different *interpretations* possible:
  - All successors of a task are activated (*concurrent task execution*). We will use this interpretation in the lecture.
  - One successor of a task is activated: *non-deterministic choice*.

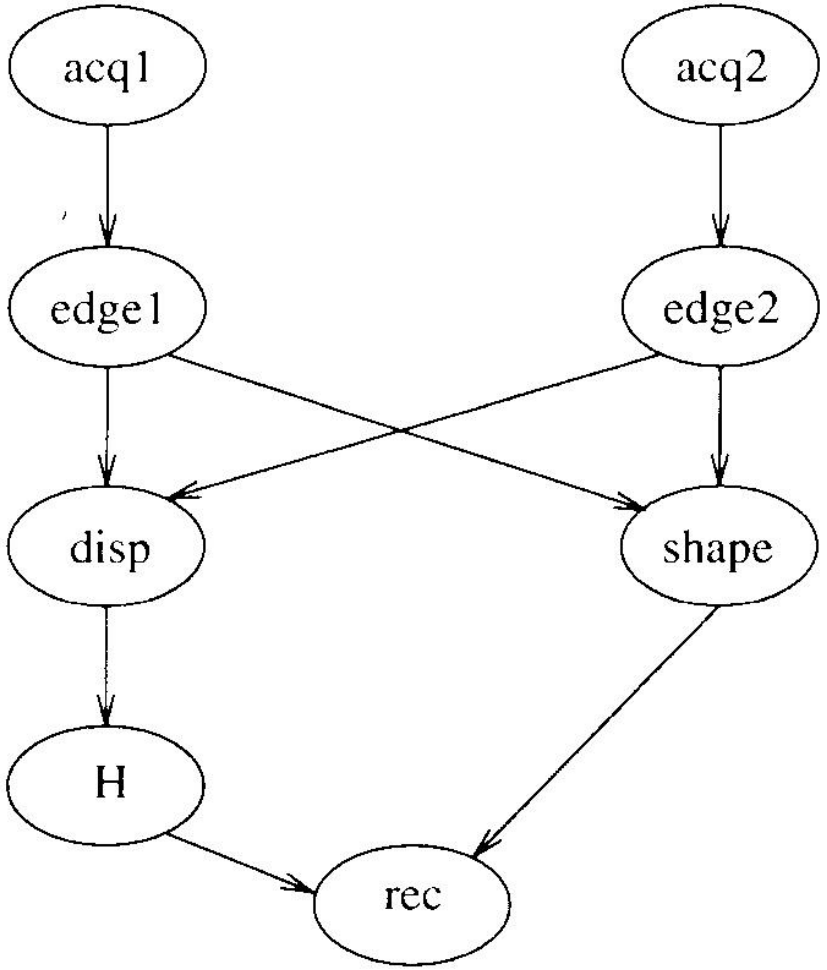


# Precedence Constraints

*Example for concurrent activation:*

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





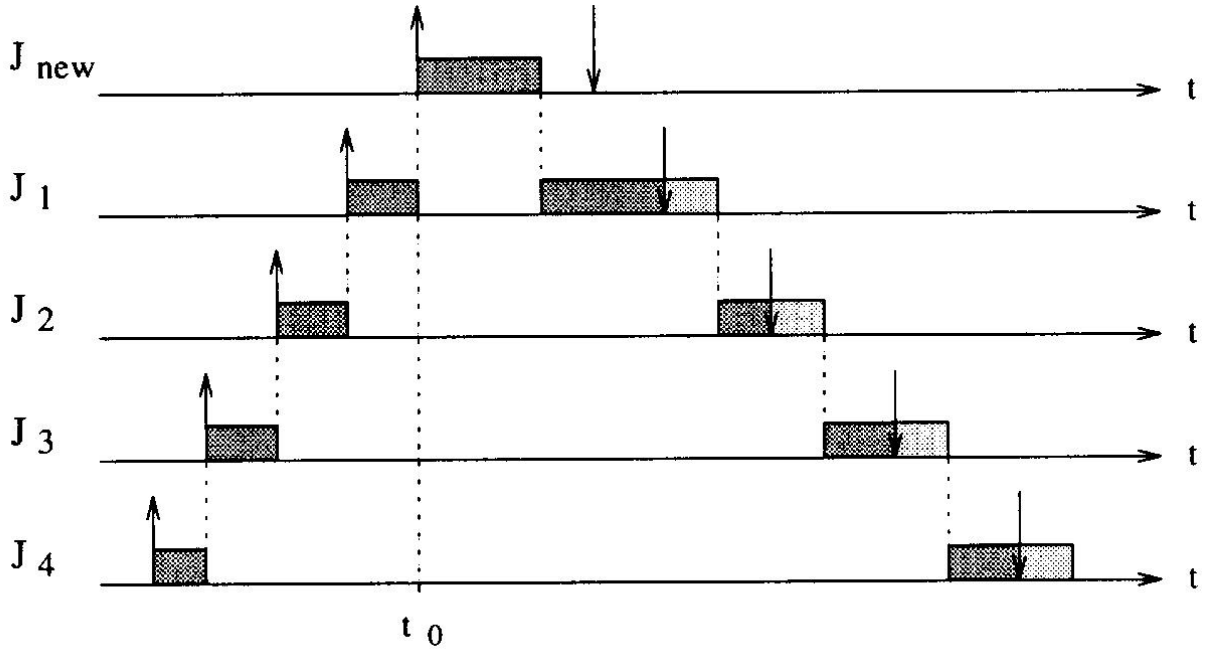
# Classification of Scheduling Algorithms

- 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.
- *Acceptance Test:* The runtime system decides whenever a task is added to the system, whether it can schedule the whole task set without deadline violations.




Example for the „*domino effect*“, if an acceptance test wrongly accepted a new task.

# Metrics to Compare Schedules

- Average response time:

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

- Total completion time:

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

- Weighted sum of response time:

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

- Maximum lateness:

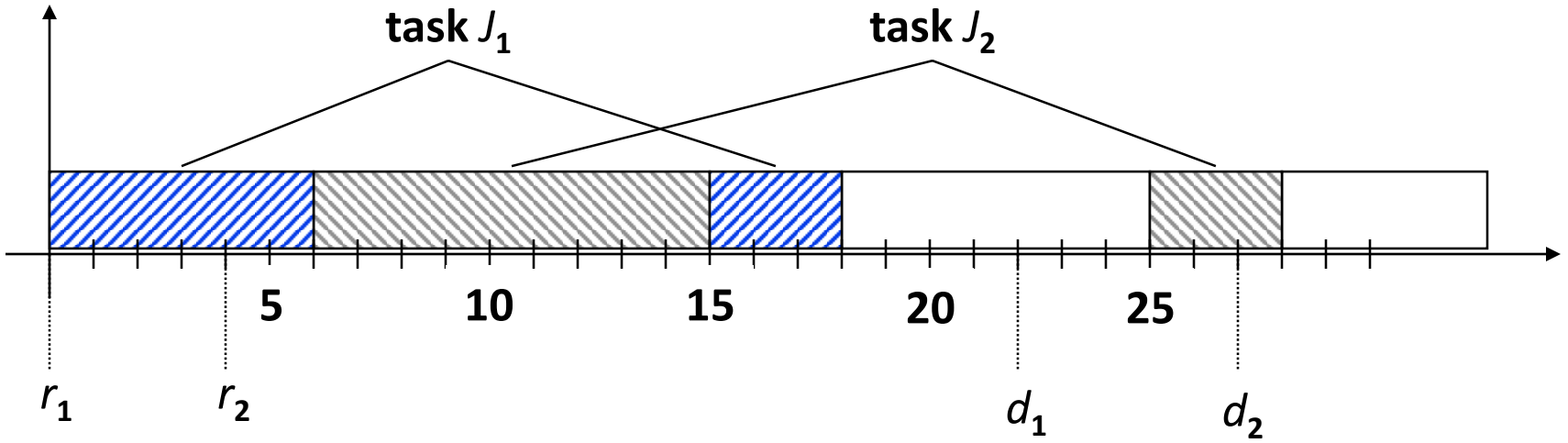
$$L_{\max} = \max_i(f_i - d_i)$$

- Number of late tasks:

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

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

# Metrics Example




Average response time: $\overline{t_r} = \frac{1}{2}(18+24) = 21$

Total completion time: $t_c = 28 - 0 = 28$

Weighted sum of response times: $w_1 = 2, w_2 = 1: \; t_w = \frac{2 \cdot 18 + 24}{3} = 20$

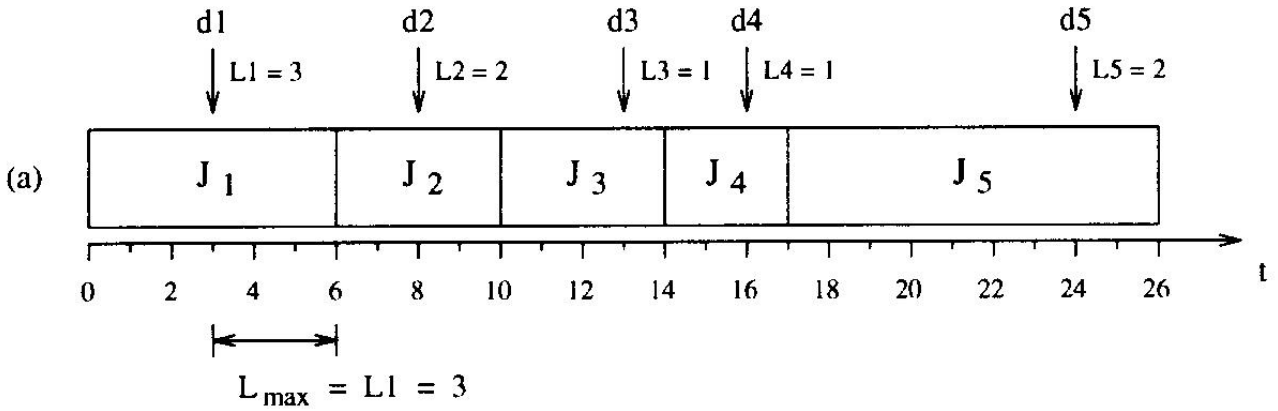
Number of late tasks: $N_{\text{late}} = 1$

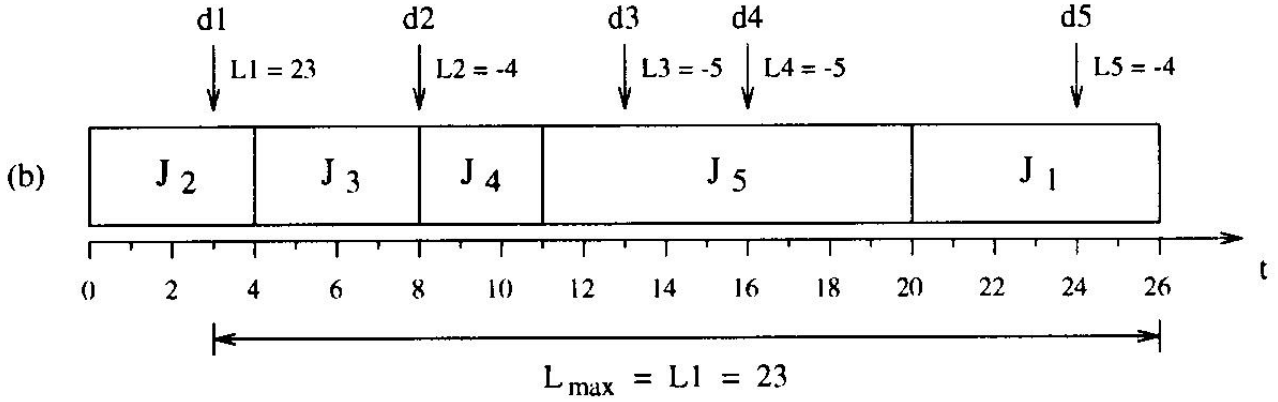
Maximum lateness: $L_{\max} = 1$

# Metrics and Scheduling Example

In schedule (a), the *maximum lateness is minimized*, but all tasks miss their deadlines.

In schedule (b), the maximal lateness is larger, but only one *task misses* its deadline.



# Real-Time Scheduling of Aperiodic Tasks

# Overview Aperiodic Task Scheduling

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) |

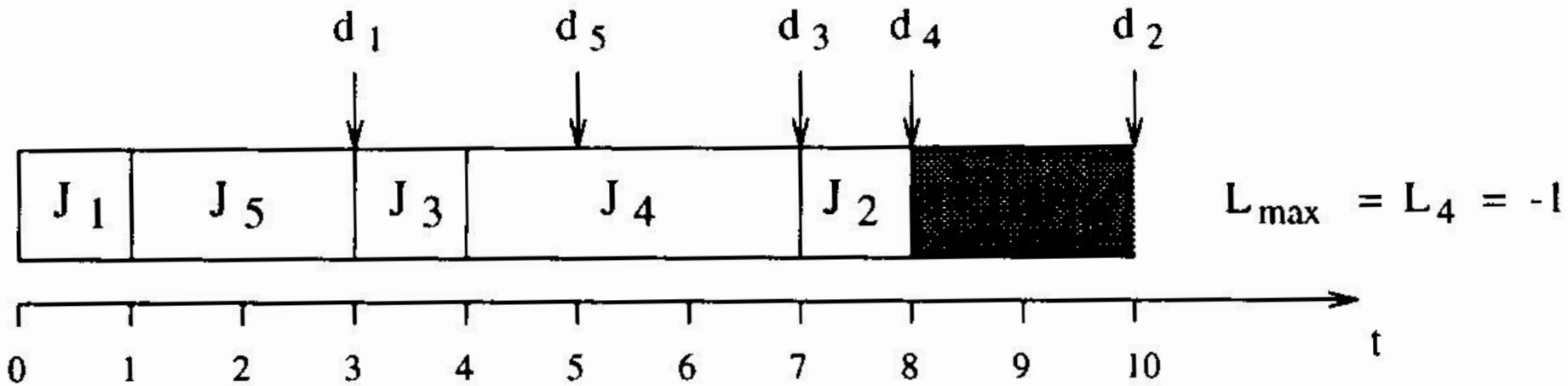
# Earliest Deadline Due (EDD)

*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.

# Earliest Deadline Due (EDD)

***Example 1:***

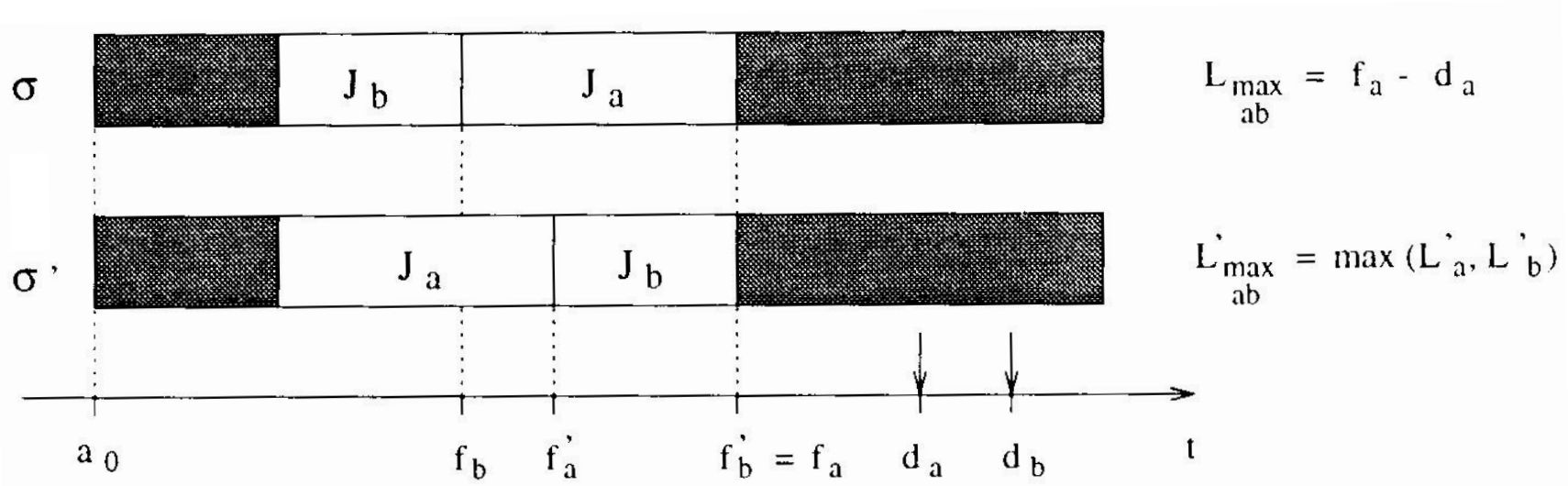
| | $J_1$ | $J_2$ | $J_3$ | $J_4$ | $J_5$ |
|---|---|---|---|---|---|
| $C_i$ | 1 | 1 | 1 | 3 | 2 |
| $d_i$ | 3 | 10 | 7 | 8 | 5 |



# Earliest Deadline Due (EDD)

*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:*




$L_{max}_{ab} = f_a - d_a$

$L'_{max}_{ab} = \max(L'_a, L'_b)$

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

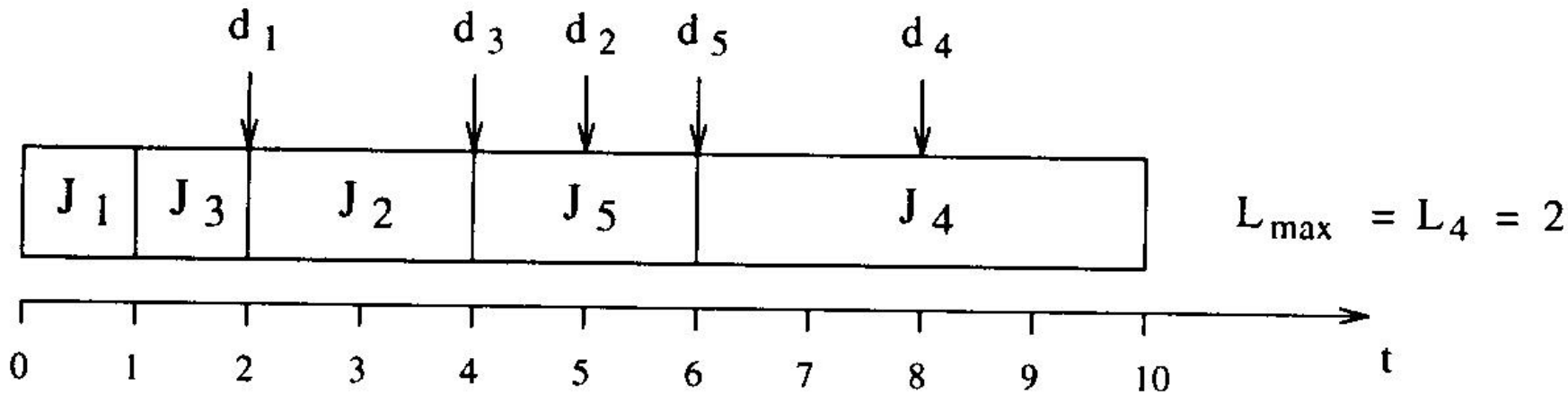
if $(L'_a \leq L'_b)$ then $L'_{max}_{ab} = f'_b - d_b < f_a - d_a$

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

# Earliest Deadline Due (EDD)

***Example 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 |



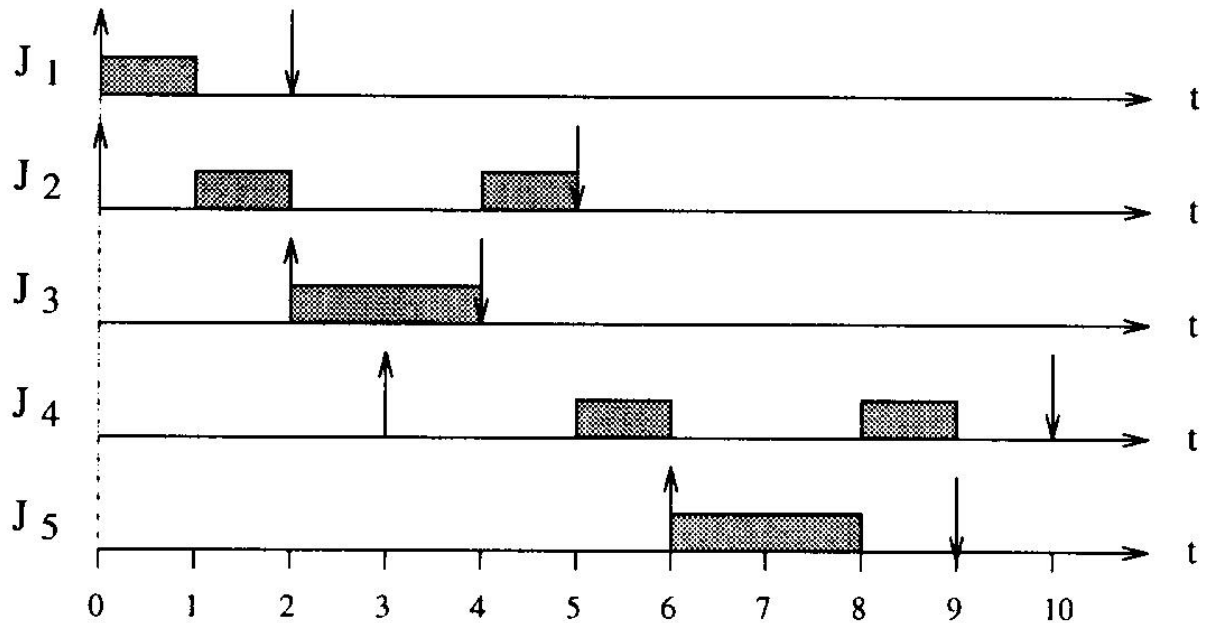
# Earliest Deadline First (EDF)

*Horn's rule:* Given a set of *n* independent tasks with arbitrary arrival times, any algorithm that at any instant executes a task with the earliest absolute deadline among the ready tasks is optimal with respect to minimizing the maximum lateness.

# 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)

*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)

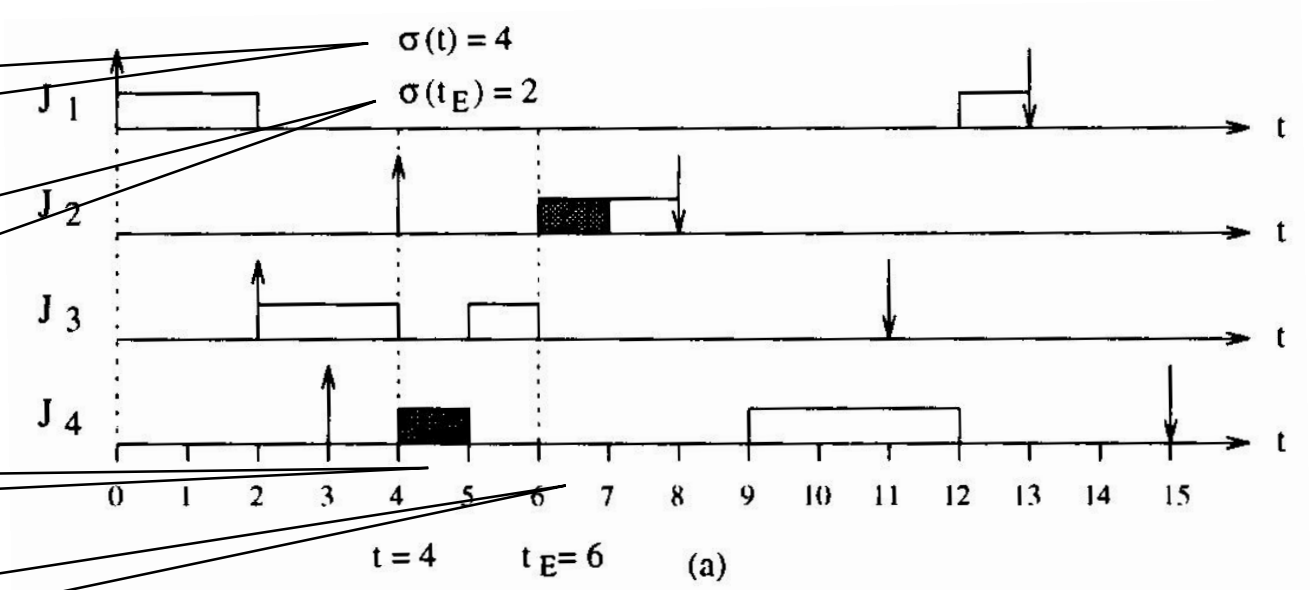
which task is executing ?

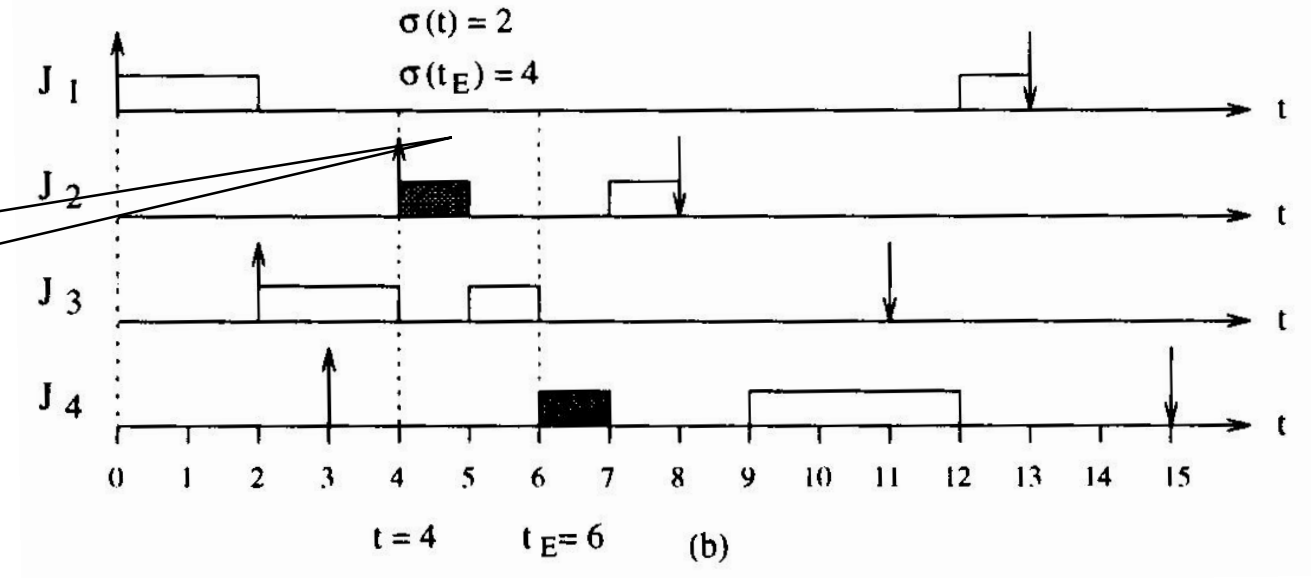
which task has earliest deadline ?

time slice

slice for interchange

situation after interchange



# Earliest Deadline First (EDF)

remaining worst-case execution time of task k

***Acceptance test:***

- worst case finishing time of task i:

$$f_i = t + \sum_{k=1}^{i} c_k(t)$$

- EDF guarantee condition:

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

- algorithm:

```
Algorithm: EDF_guarantee (J, J_new)
{       J'=J∪{J_new};  /* ordered by deadline */
        t = current_time();
        f_0 = t;
        for (each J_i∈J') {
                f_i = f_i-1 + c_i(t);
                if (f_i > d_i) return(INFEASIBLE);
        }
        return(FEASIBLE);
}
```

# 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

# EDF*

# EDF*

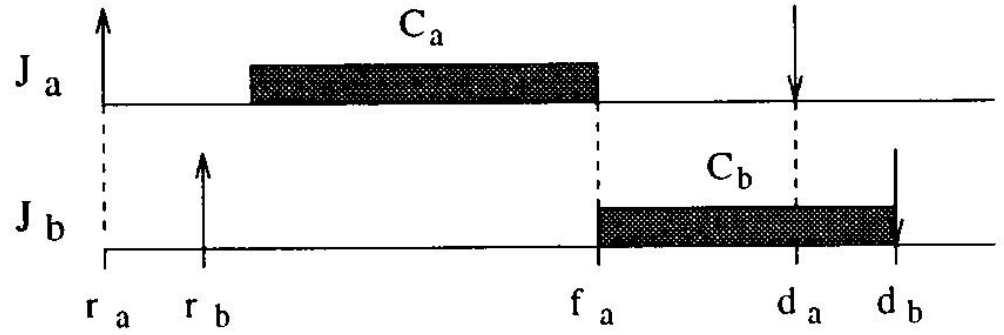
# Earliest Deadline First (EDF*)

*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.

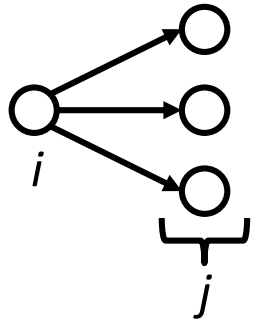
**task b depends on task a:** $J_a \rightarrow J_b$

$$\begin{cases} f_a \le d_a \\ f_a \le d_b - C_b \end{cases}$$




- *Solution:*

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



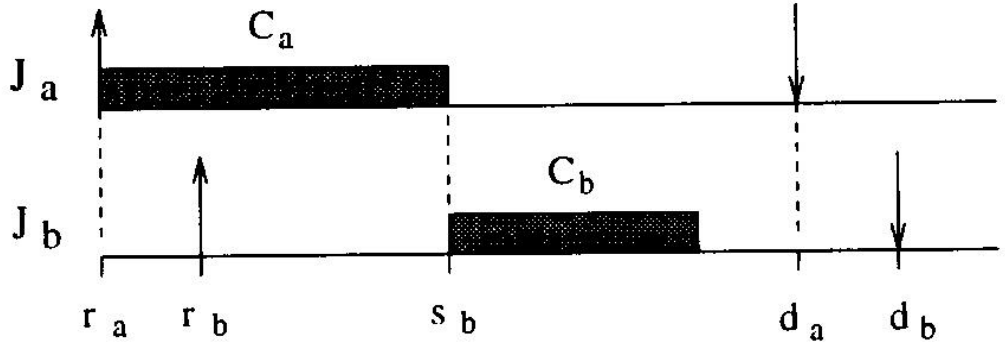
# Earliest Deadline First (EDF*)

*Modification of release times:*

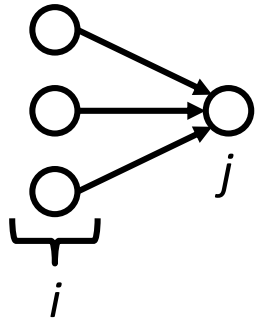
- Task must start the execution not earlier than its release time.
- Task must not start the execution earlier than the minimum finishing time of its predecessor.

**task b depends on task a:** $J_a \rightarrow J_b$

$$\begin{cases} s_b \geq r_b \\ s_b \geq r_a + C_a \end{cases}$$




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



# 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 $r_j^* = \max\left(r_j, \max\left(r_i^* + C_i : J_i \rightarrow J_j\right)\right)$
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 $d_i^* = \min\left(d_i, \min\left(d_j^* - C_j : J_i \rightarrow J_j\right)\right)$
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. To this end, show that the original task set meets the timing constraints also. This can be done by using $r_i^* \geq r_i$ , $d_i^* \leq d_i$ ; we only made the constraints stricter.
- Show that if there exists a schedule for the original task set, then also for the modified one. We can show the following: If there exists no schedule for the modified task set, then there is none for the original task set. This can be done by showing that no feasible schedule was excluded by changing the deadlines and release times.
- 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.

# Real-Time Scheduling of Periodic Tasks

# Overview

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

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

# Model of Periodic Tasks

- *Examples:* sensory data acquisition, low-level actuation, control loops, action planning and system monitoring.
- When an *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:

$\Gamma$ : denotes a set of periodic tasks
$\tau_i$ : denotes a periodic task
$\tau_{i,j}$ : denotes the jth instance of task i
$r_{i,j}$, $s_{i,j}$, $f_{i,j}$, $d_{i,j}$ : denote the release time, start time, finishing time, absolute deadline of the jth instance of task i
$\Phi_i$ : denotes the phase of task i (release time of its first instance)
$D_i$ : denotes the relative deadline of task i
$T_i$ : denotes the period 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

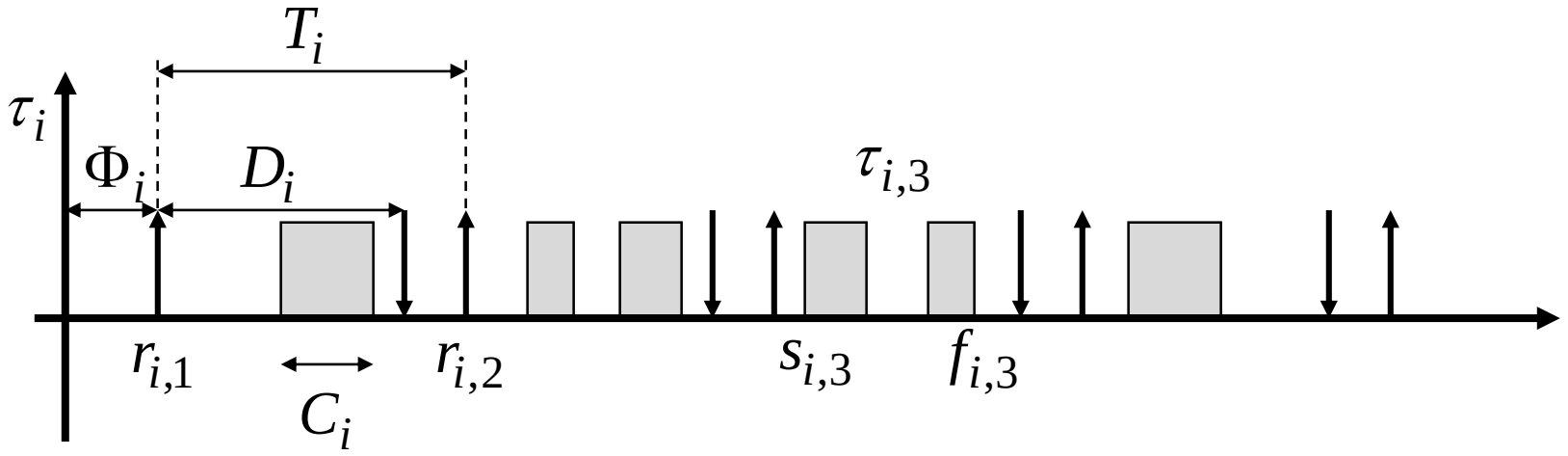
$$d_{i,j} = \Phi_i + (j-1)T_i + D_i$$

  - Often, the relative deadline equals the period $D_i = T_i$ (*implicit deadline*), and therefore

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

# Model of Periodic Tasks

- *The following hypotheses are assumed on the tasks (continued):*
  - 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*.
  - *Example:*



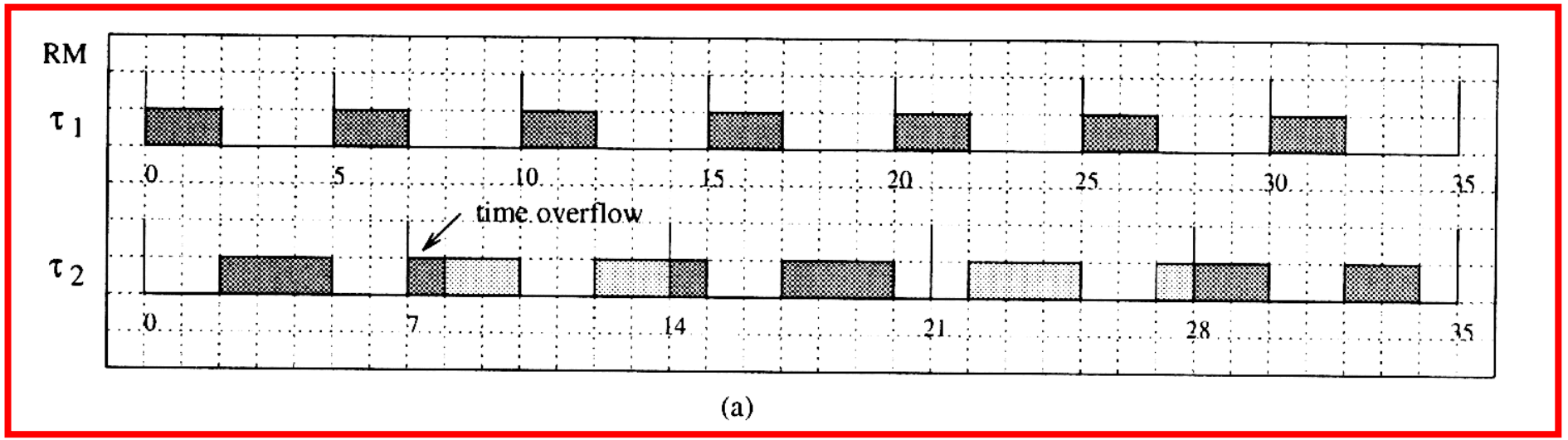
# Rate Monotonic Scheduling (RM)

- *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 job is preempted by a job of a task with higher priority.
  - Deadlines equal the periods $D_i = T_i$ .

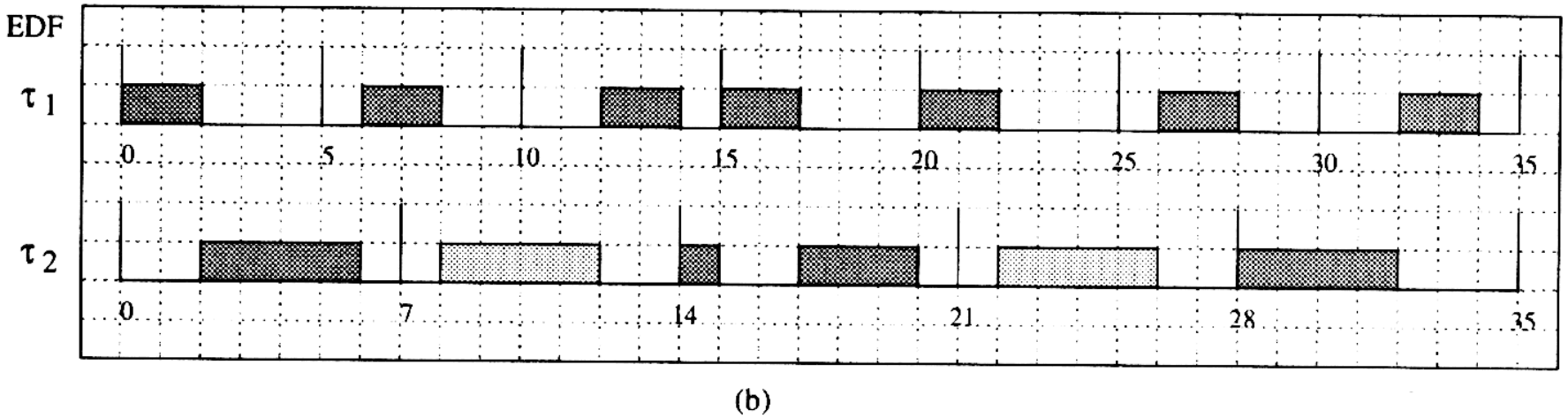
Rate-Monotonic Scheduling Algorithm: Each task is assigned a priority. Tasks with higher request rates (that is with shorter periods) will have higher priorities. Jobs of tasks with higher priority interrupt jobs of tasks with lower priority.

# Periodic Tasks

***Example:*** 2 tasks, deadlines = periods, utilization = 97%




(a)




(b)

# Rate Monotonic Scheduling (RM)

*Optimality:* RM is optimal among all fixed-priority assignments in the sense that no 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 instants. If all tasks are feasible at their critical instant, 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.

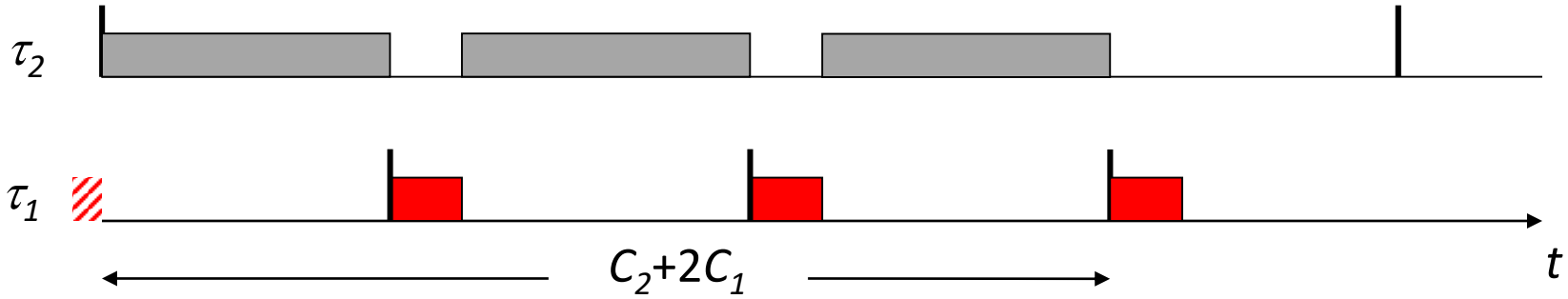
# Proof of Critical Instance

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

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

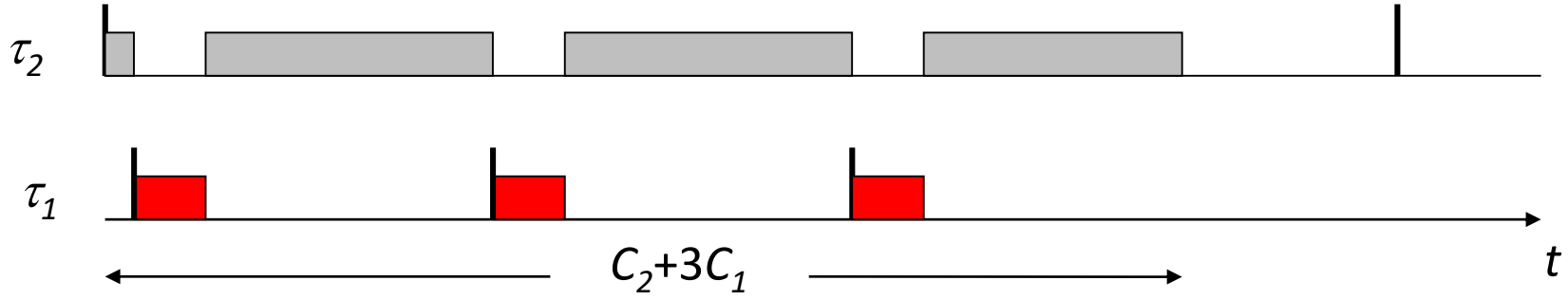
*Proof sketch:* Start with 2 tasks $\tau_1$ and $\tau_2$ .

Response time of a job of $\tau_2$ is delayed by jobs of $\tau_1$ of higher priority:



# Proof of Critical Instance

Delay may increase if $\tau_1$ starts earlier:




Maximum delay achieved if $\tau_2$ and $\tau_1$ start simultaneously.

Repeating the argument for all higher priority tasks of some task $\tau_2$ :

> The worst case response time of a job occurs when it is released simultaneously with all higher-priority jobs.

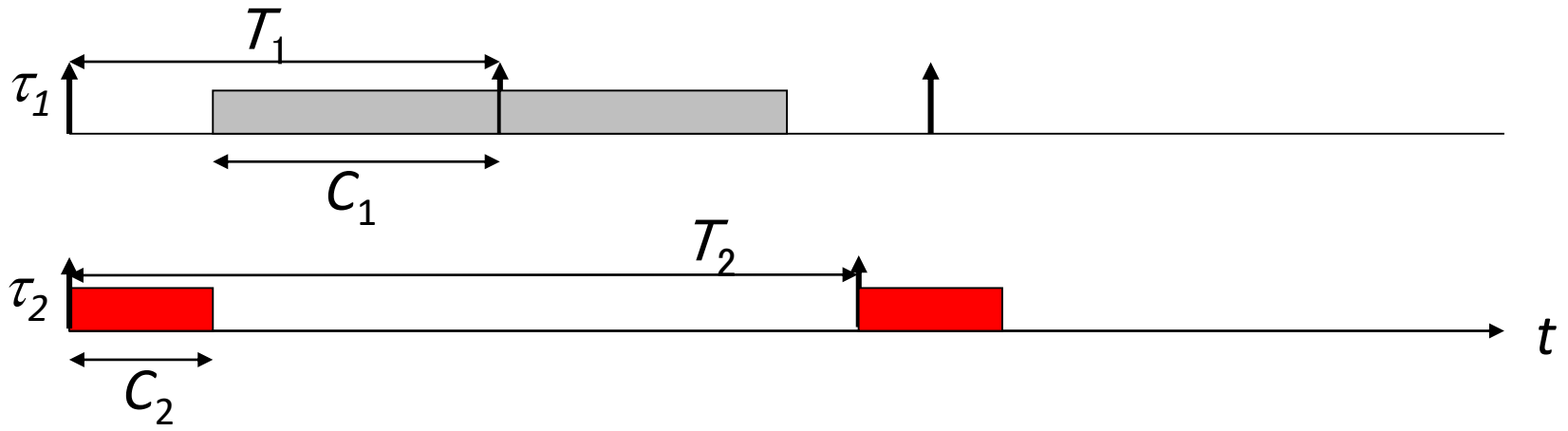
# Proof of RM Optimality (2 Tasks)

We have two tasks $\tau_1$, $\tau_2$ with periods $T_1 < T_2$.

Define $F = \lfloor T_2/T_1 \rfloor$: the number of periods of $\tau_1$ **fully** contained in $T_2$

Consider two cases A and B:

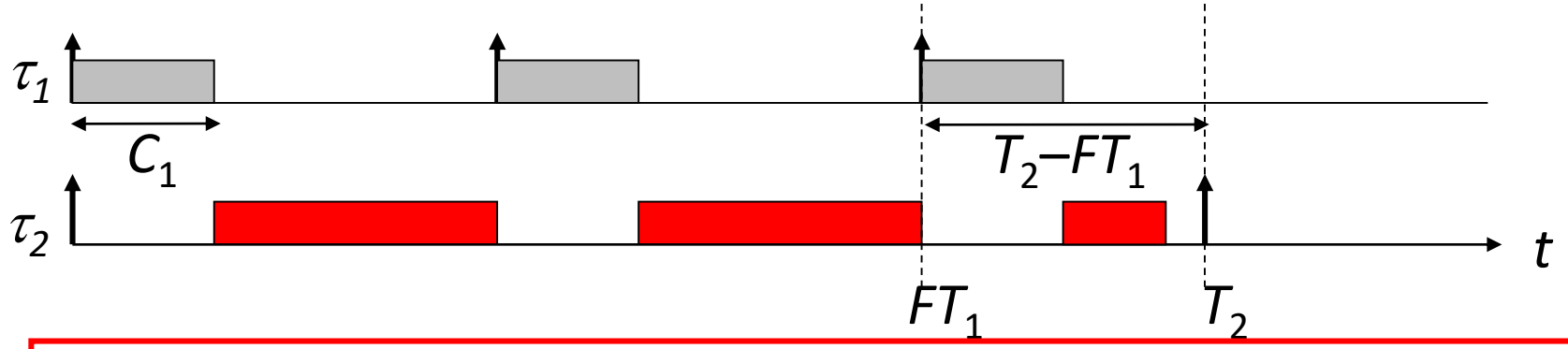
*Case A:* Assume RM is not used → prio($\tau_2$) is highest:




Schedule is feasible if $C_1+C_2 \leq T_1$ and $C_2 \leq T_2$ (A)

# Proof of RM Optimality (2 Tasks)

*Case B:* Assume RM **is** used → prio($\tau_1$) is highest:




Schedulable is feasible if

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

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

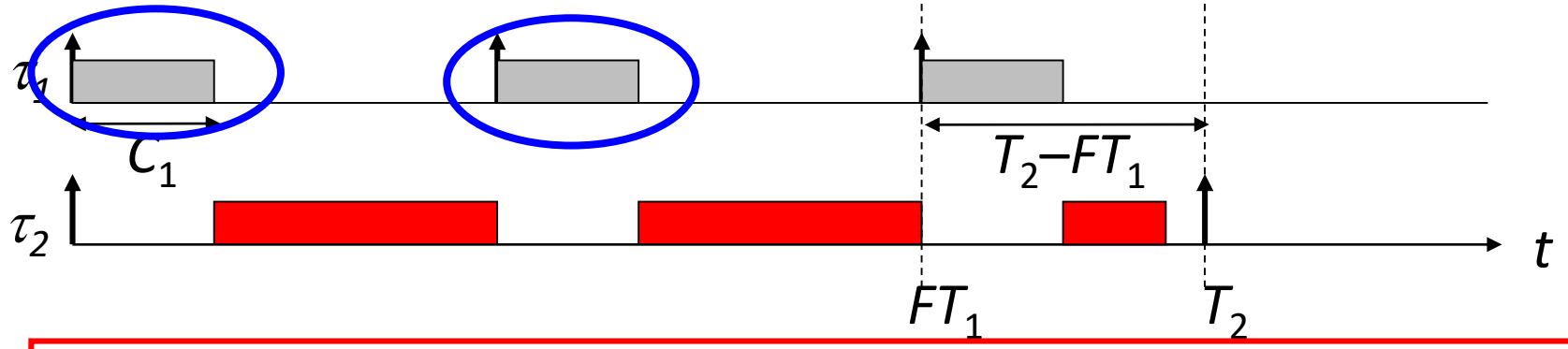
$$C_1+C_2 \le T_1 \Rightarrow FC_1+C_2 \le FC_1+FC_2 \le F T_1 \Rightarrow$$

$$FC_1+C_2+\min(T_2-FT_1, C_1) \le F T_1 +\min(T_2-FT_1, C_1) \le \min(T_2, C_1+FT_1) \le T_2$$

Given tasks $\tau_1$ and $\tau_2$ with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

# Proof of RM Optimality (2 Tasks)

*Case B:* Assume RM **is** used → prio($\tau_1$) is highest:




Schedulable is feasible if

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

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

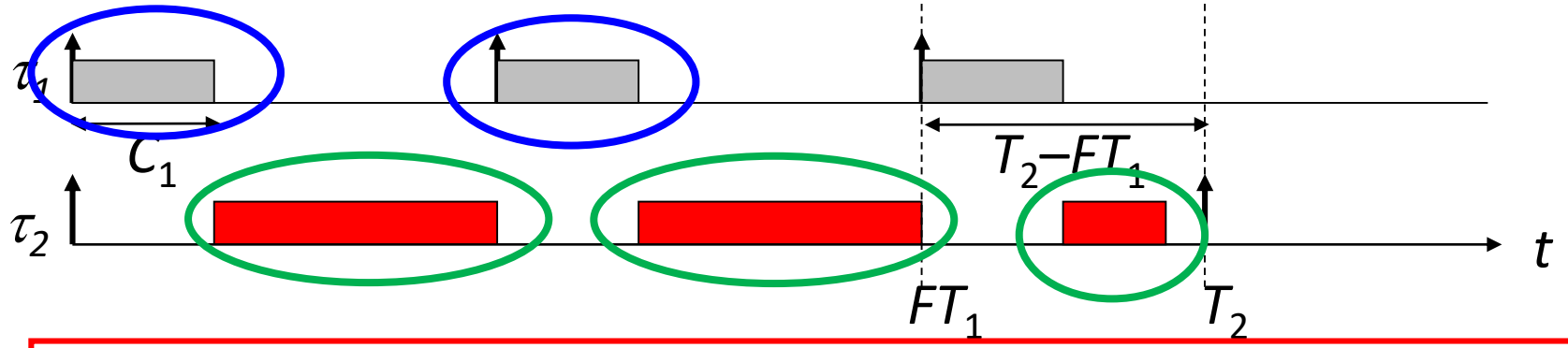
$$C_1+C_2 \le T_1 \Rightarrow FC_1+C_2 \le FC_1+FC_2 \le F T_1 \Rightarrow$$

$$FC_1+C_2+\min(T_2-FT_1, C_1) \le F T_1 +\min(T_2-FT_1, C_1) \le \min(T_2, C_1+FT_1) \le T_2$$

Given tasks $\tau_1$ and $\tau_2$ with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

# Proof of RM Optimality (2 Tasks)

*Case B:* Assume RM **is** used → prio($\tau_1$) is highest:




Schedulable is feasible if

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

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

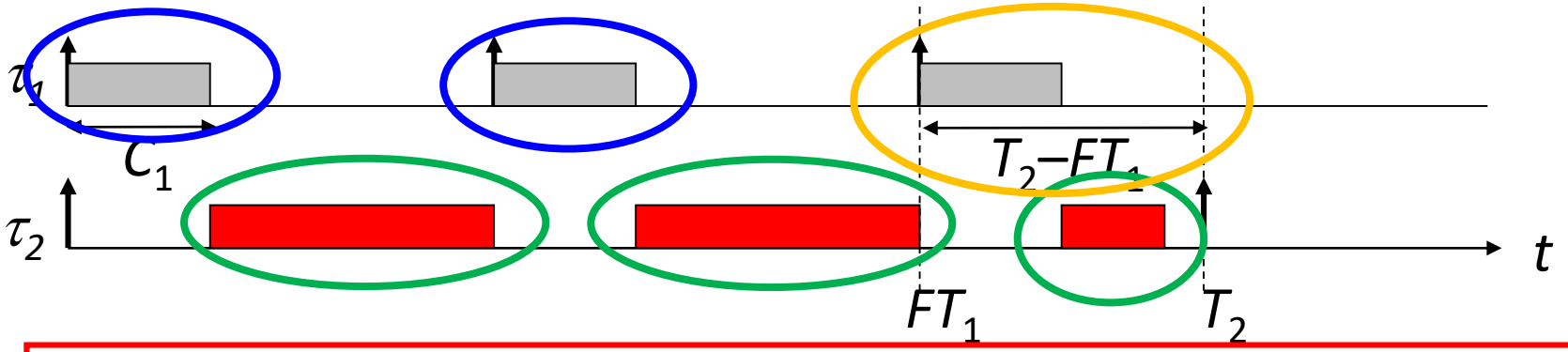
$$C_1+C_2 \le T_1 \Rightarrow FC_1+C_2 \le FC_1+FC_2 \le F T_1 \Rightarrow$$

$$FC_1+C_2+\min(T_2-FT_1, C_1) \le F T_1 +\min(T_2-FT_1, C_1) \le \min(T_2, C_1+FT_1) \le T_2$$

Given tasks $\tau_1$ and $\tau_2$ with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

# Proof of RM Optimality (2 Tasks)

*Case B:* Assume RM **is** used → prio($\tau_1$) is highest:




Schedulable is feasible if

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

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

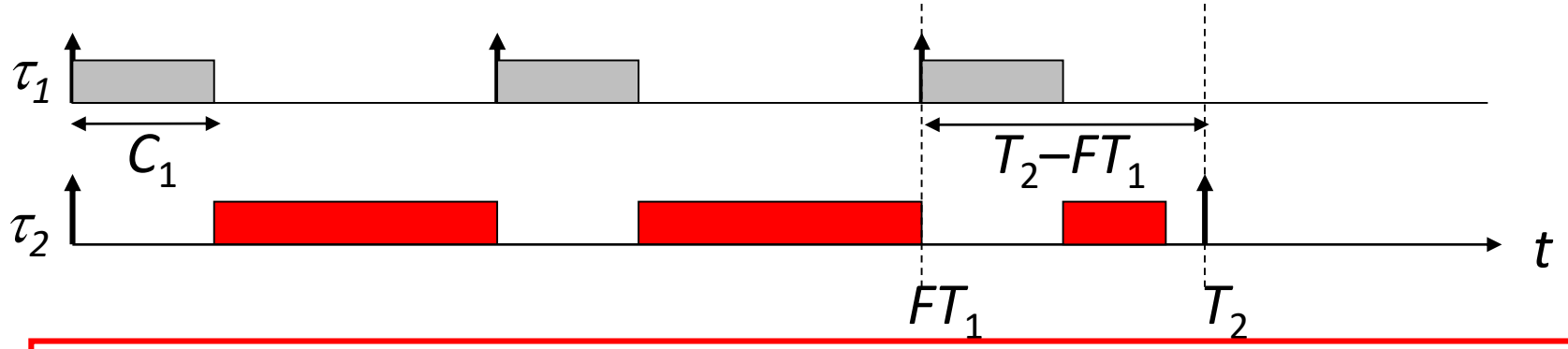
$$C_1+C_2 \le T_1 \Rightarrow FC_1+C_2 \le FC_1+FC_2 \le F T_1 \Rightarrow$$

$$FC_1+C_2+\min(T_2-FT_1, C_1) \le F T_1 +\min(T_2-FT_1, C_1) \le \min(T_2, C_1+FT_1) \le T_2$$

Given tasks $\tau_1$ and $\tau_2$ with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

# Proof of RM Optimality (2 Tasks)

*Case B:* Assume RM **is** used → prio($\tau_1$) is highest:




Schedulable is feasible if

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

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

$$C_1+C_2 \le T_1 \Rightarrow FC_1+C_2 \le FC_1+FC_2 \le F T_1 \Rightarrow$$

$$FC_1+C_2+\min(T_2-FT_1, C_1) \le F T_1 +\min(T_2-FT_1, C_1) \le \min(T_2, C_1+FT_1) \le T_2$$

Given tasks $\tau_1$ and $\tau_2$ with $T_1 < T_2$, then if the schedule is feasible by an arbitrary fixed priority assignment, it is also feasible by RM.

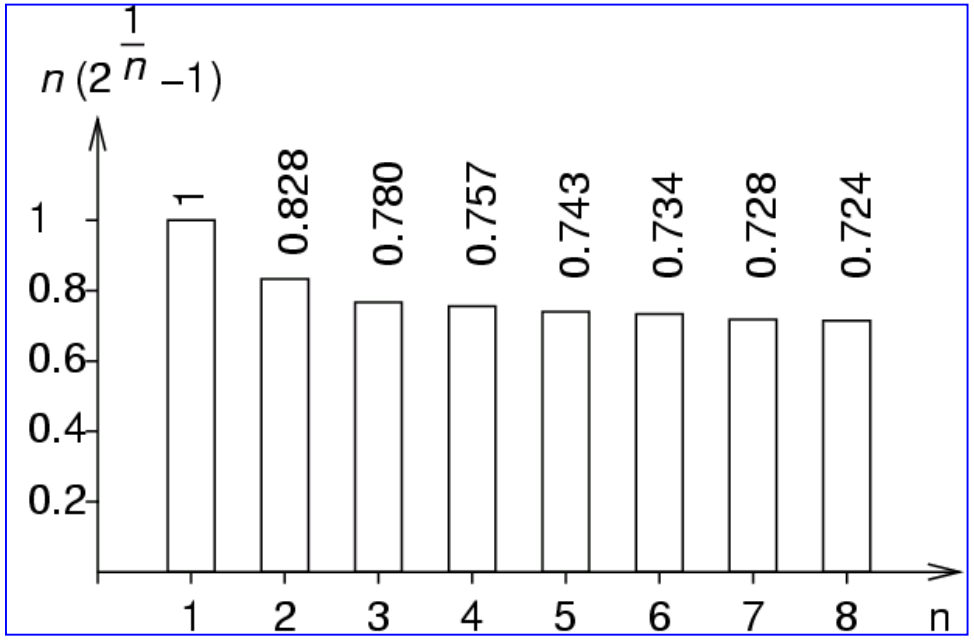
# Admittance Test

# Rate Monotonic Scheduling (RM)

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

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

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)

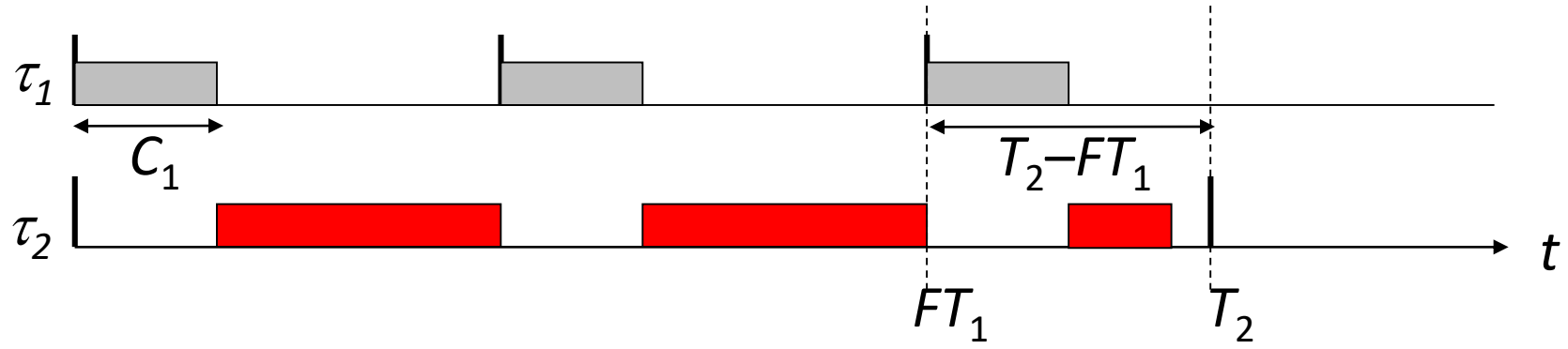
We have two tasks $\tau_1$, $\tau_2$ with periods $T_1 < T_2$.
Define $F = \lfloor T_2/T_1 \rfloor$: number of periods of $\tau_1$ **fully** contained in $T_2$

*Proof Concept:* 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 increasing the computation time $C_2$ to just meet the deadline of $\tau_2$; we will determine this limit of $C_2$ using the results of the RM optimality proof.
- minimize upper bound with respect to other task parameters in order to find the utilization below which the system is definitely schedulable.

# Proof of Utilization Bound (2 Tasks)

*As before:*




Schedulable if $FC_1+C_2+\min(T_2-FT_1, C_1) \leq T_2$ and $C_1 \leq T_1$

*Utilization:*

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} = \frac{C_1}{T_1} + \frac{T_2 - FC_1 - \min\{T_2 - FT_1, C_1\}}{T_2}$$

$$= 1 + \frac{C_1(T_2 - FT_1) - T_1 \min\{T_2 - FT_1, C_1\}}{T_1 T_2}$$

# Proof of Utilization Bound (2 Tasks)

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

# 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 $C_1 = T_2 - FT_1$ :

$$U = 1 + \frac{(T_2 - FT_1)^2 - T_1(T_2 - FT_1)\}}{T_1 T_2}$$

$$= 1 + \frac{T_1}{T_2}\left(\left(\frac{T_2}{T_1} - F\right)^2 - \left(\frac{T_2}{T_1} - F\right)\right)$$

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 $G = T_2/T_1$. Then we obtain

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

# Proof of Utilization Bound (2 Tasks)

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

$$G=\frac{T_2}{T_1}=\sqrt{2} \qquad \boxed{U=2(\sqrt{2}-1)}$$

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 period*, 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. Jobs with higher priority interrupt jobs with lower priority.

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

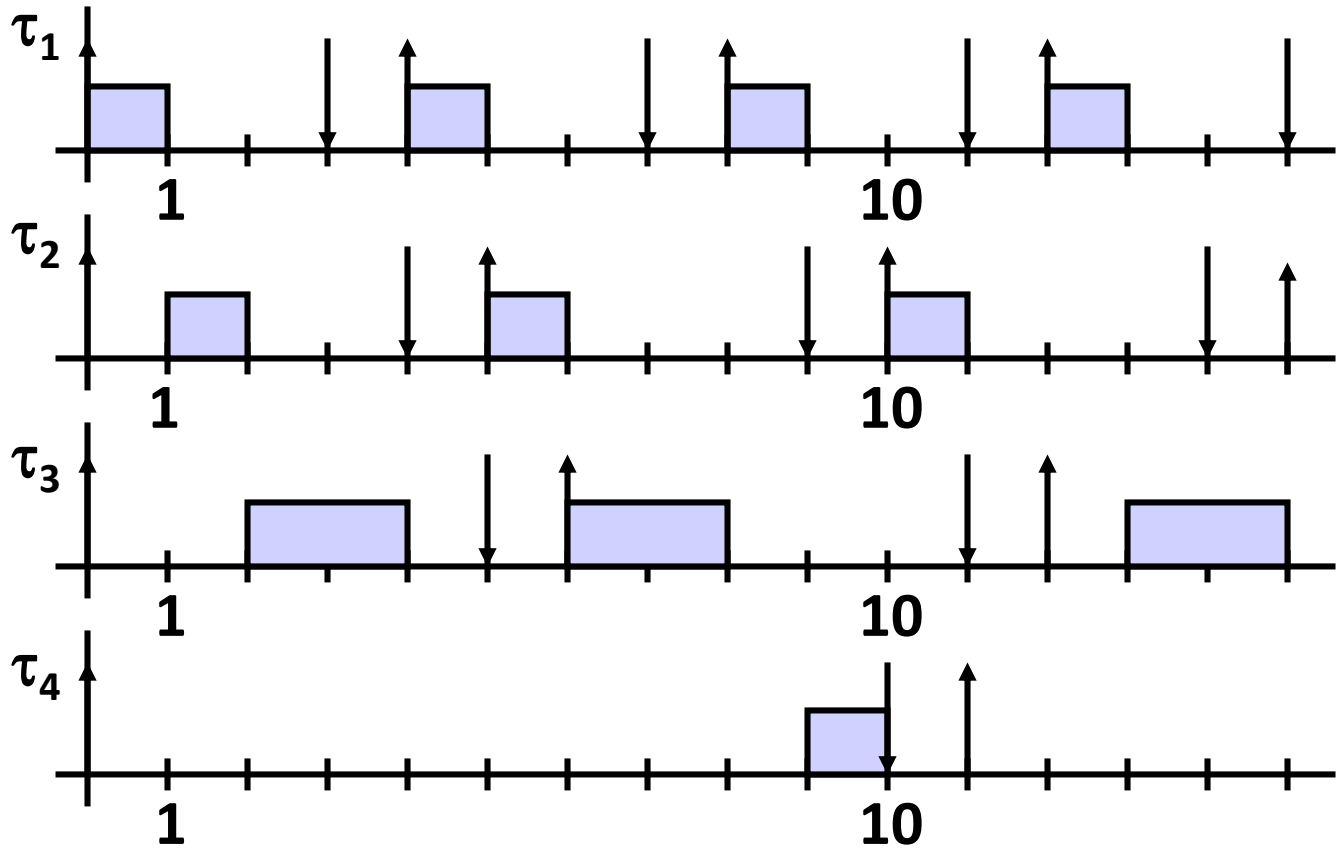
$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{1/n} - 1\right)$$

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

# Deadline Monotonic Scheduling (DM) - Example

$U = 0.874$

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



# 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* 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 job 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 (Γ)
{       for (each τi∈Γ){
                I = 0;
                do {
                        R = I + Ci;
                        if (R > Di) return(UNSCHEDULABLE);
                        I = ∑j=1,…,(i-1)⌈R/Tj⌉ Cj;
                } while (I + Ci > R);
        }
        return(SCHEDULABLE);
}
```

# DM Example

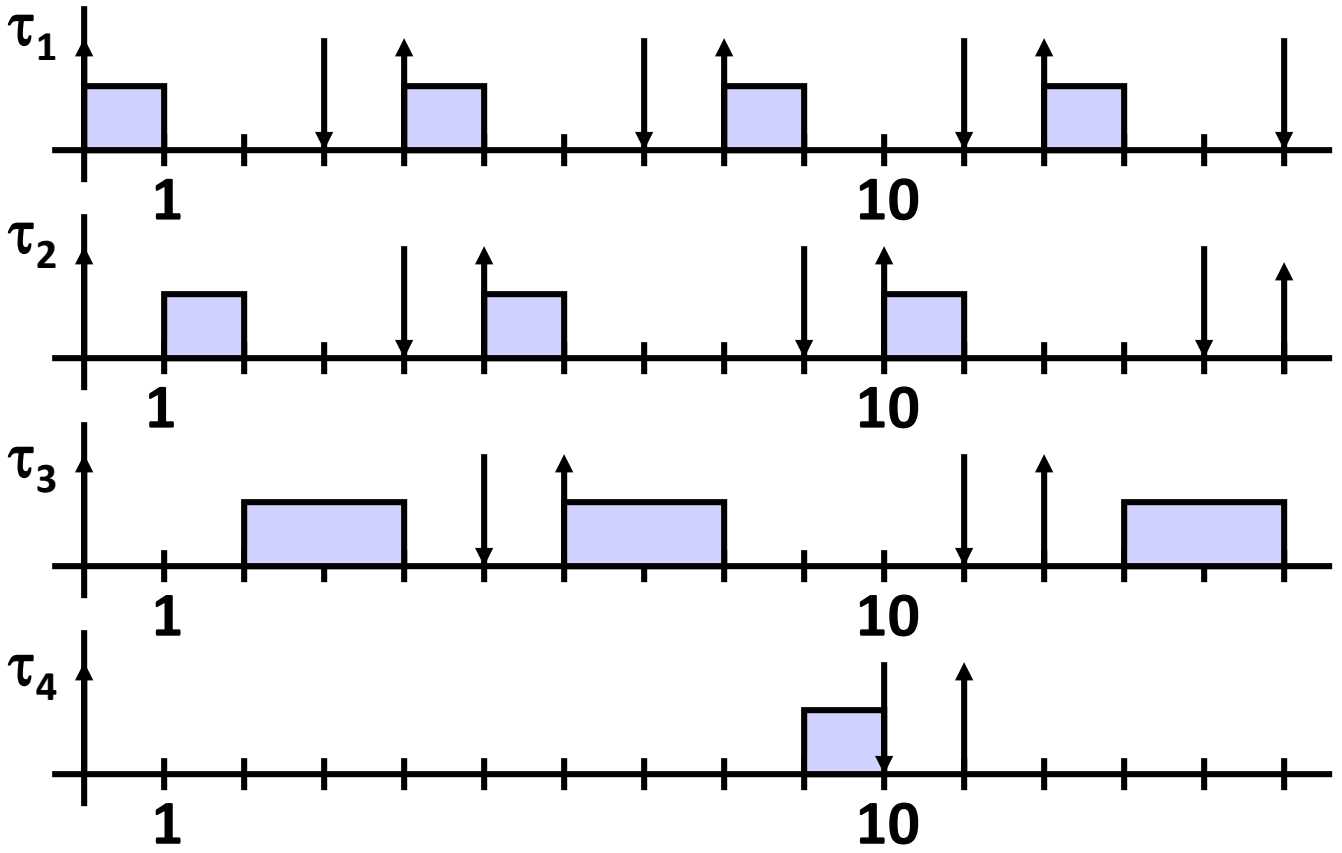
***Example:***

- 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 the schedulability test 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

$U = 0.874$

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



# EDF Scheduling (earliest deadline first)

- *Assumptions:*
  - dynamic priority assignment
  - intrinsically preemptive

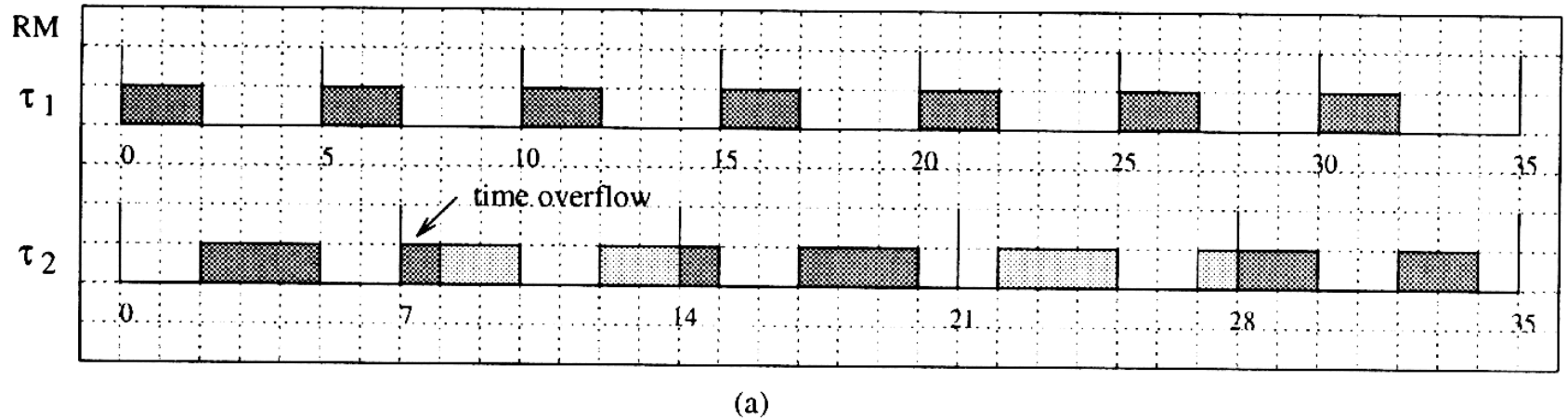
- *Algorithm:* The currently executing task is preempted whenever another periodic instance with earlier deadline becomes active.

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

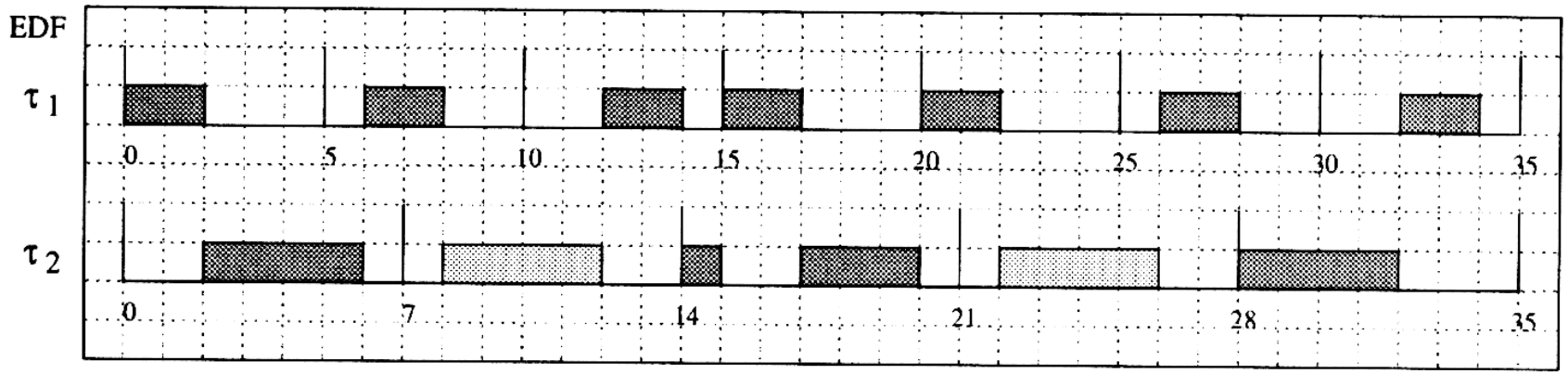
- *Optimality:* No other algorithm can schedule a set of periodic tasks if the set that can not be scheduled by EDF.
- The proof is simple and follows that of the aperiodic case.

# Periodic Tasks

***Example:*** 2 tasks, deadlines = periods, utilization = 97%




(a)




(b)

# EDF Scheduling

A *necessary and sufficient schedulability test* for $D_i = T_i$ :

> 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 $T = T_1 \cdot T_2 \cdot \ldots \cdot T_n$ is

$$\sum_{i=1}^{n} \frac{C_i}{T_i} T = UT > T$$

and therefore, it exceeds the available processor time in this interval.

- If the utilization satisfies $U \leq 1$ , then there is a valid schedule.

We will proof this fact 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 a time before $t_2$ such that (a) the processor is continuously busy in [$t_1$, $t_2$ ] and (b) the processor only executes tasks that have their arrival time AND their 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 such that shortly before the processor works on a task with deadline after $t_2$ or the processor is idle, then we found $t_1$: We know that there is no execution on a task with deadline after $t_2$ .
    - But it could be in principle, that a task that arrived before $t_1$ is executing in [$t_1$, $t_2$ ].
    - If the processor is idle before $t_1$, then this is clearly not possible due to EDF (the processor is not idle, if there is a ready task).
    - If the processor is not idle before $t_1$, this is not possible as well. Due to EDF, the processor will always work on the task with the closest deadline and therefore, once starting with a task with deadline after $t_2$ all task with deadlines before $t_2$ are finished.

# EDF Scheduling

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

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

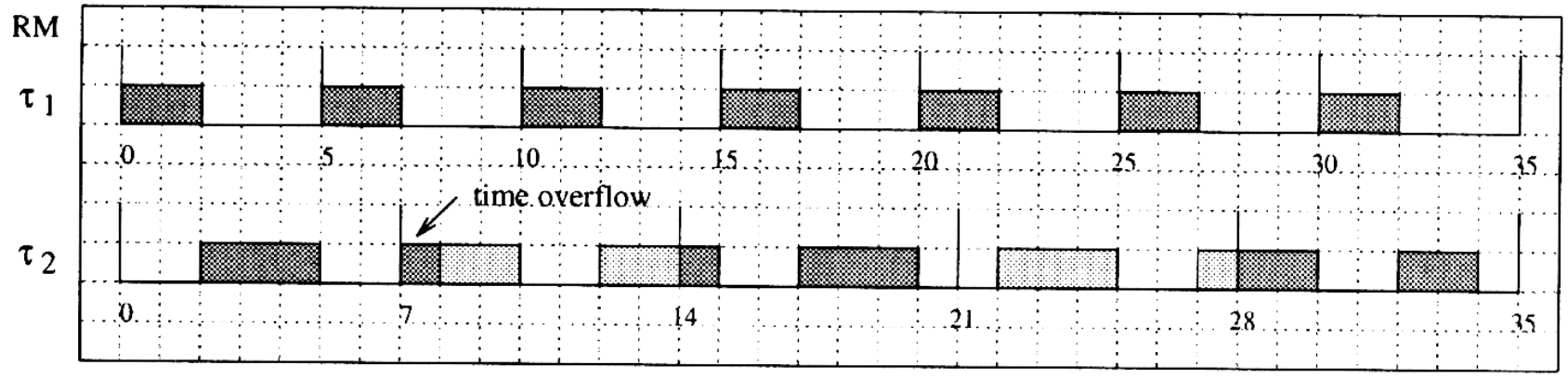
number of complete periods of task i in the interval

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

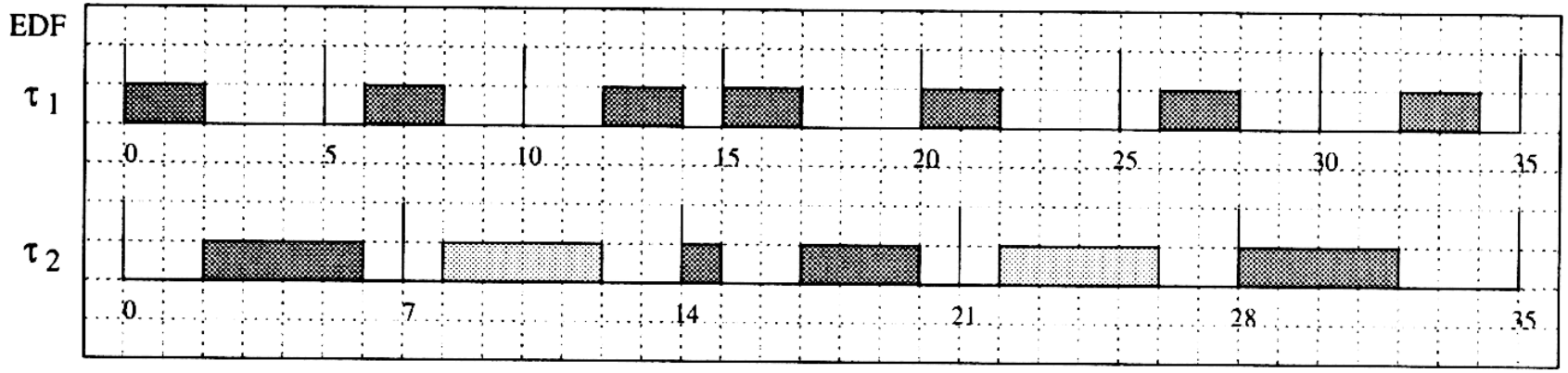
$$t_2 - t_1 < C_p(t_1, t_2) \leq (t_2 - t_1)U \Rightarrow U > 1$$

# Periodic Task Scheduling

***Example:*** 2 tasks, deadlines = periods, utilization = 97%




(a)




(b)

# Real-Time Scheduling of Mixed Task Sets

# Problem of Mixed Task Sets

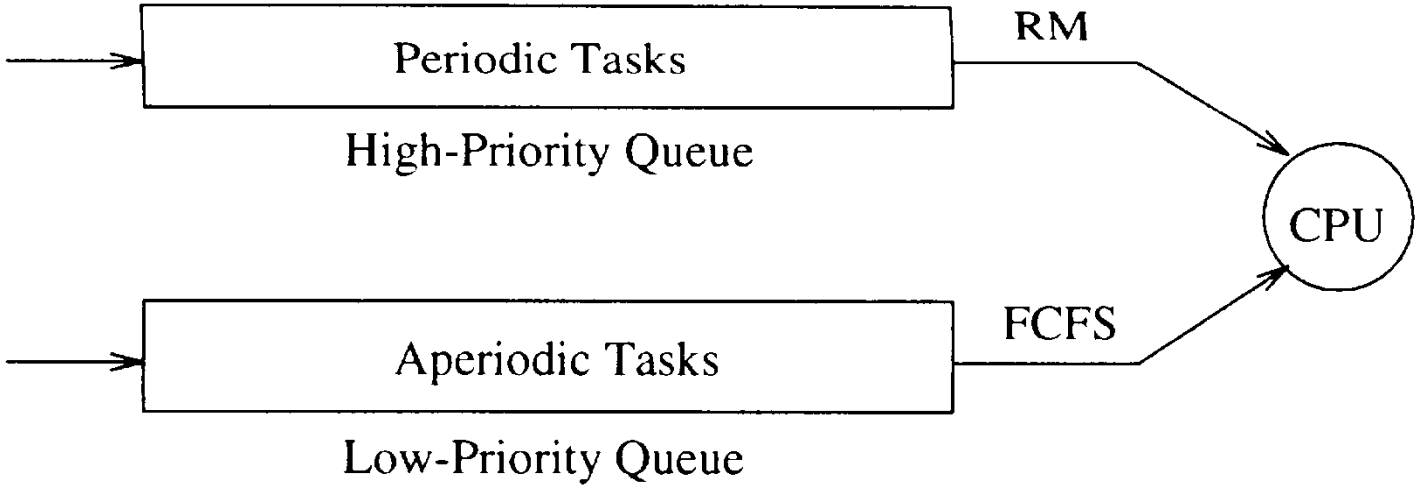
In many applications, there are aperiodic as well as periodic tasks.

- *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

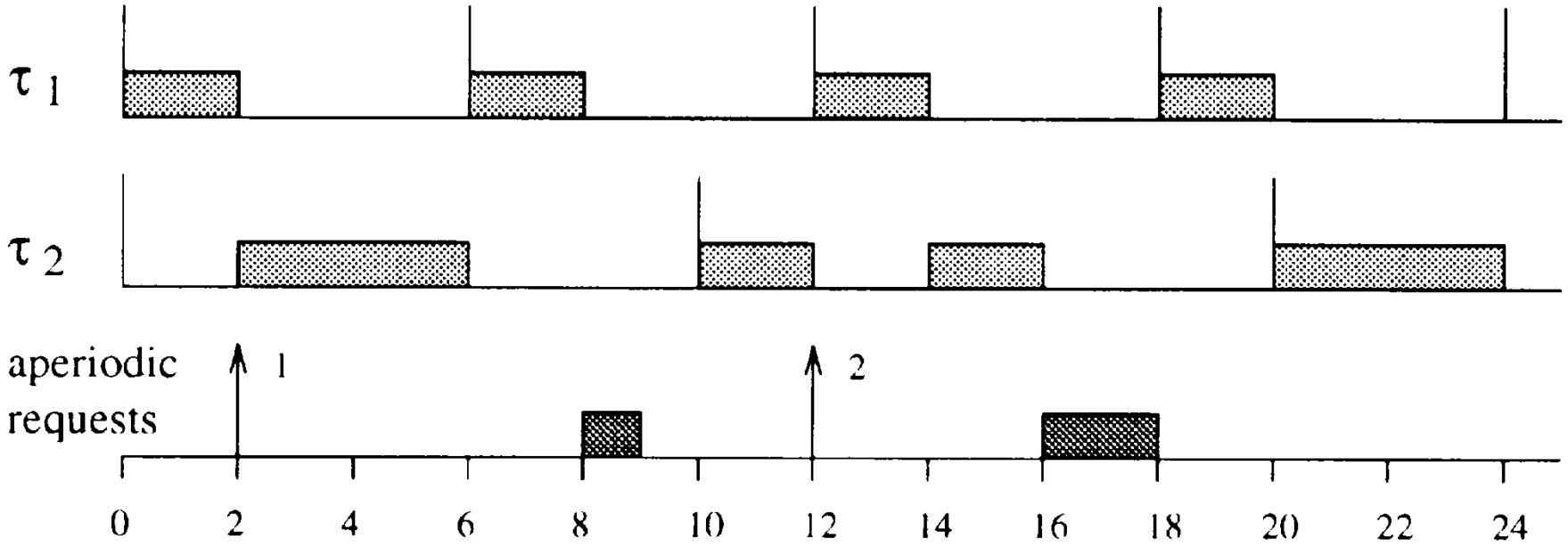
*Background scheduling* is a simple solution for RM and EDF:

- Processing of aperiodic tasks in the background, i.e. execute if there are no pending periodic requests.
- 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.
- Example:



# Background Scheduling

*Example* (rate monotonic periodic schedule):



# Rate-Monotonic Polling Server

- ***Idea:*** Introduce an artificial periodic task whose purpose is to service aperiodic requests as soon as possible (therefore, “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.
  - Its priority (period!) can be chosen to match the response time requirement for the aperiodic tasks.
- *Disadvantage:* If an aperiodic requests arrives just after the server has suspended, it must wait until the beginning of the next polling period.

# Rate-Monotonic Polling Server

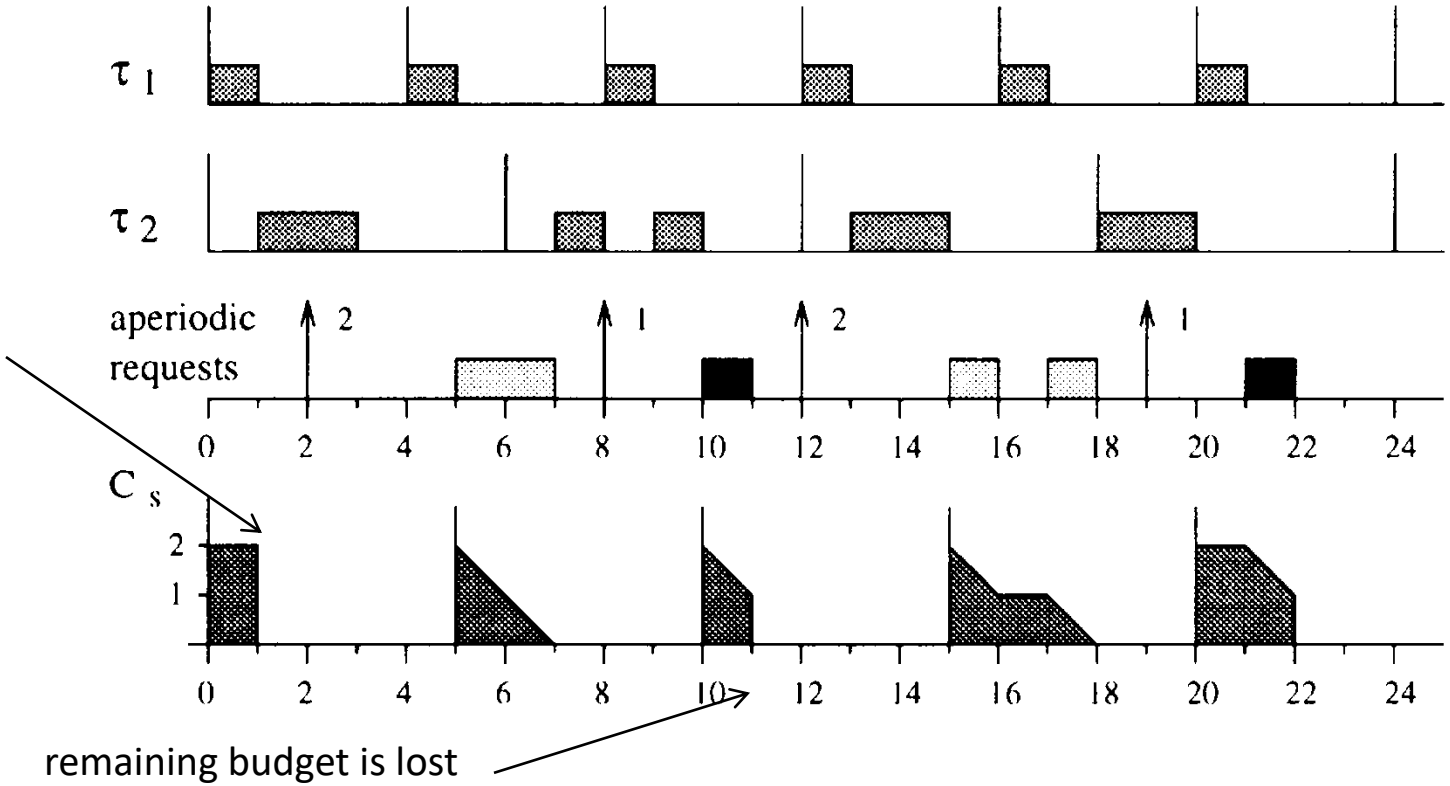
*Example:*

| | $C_i$ | $T_i$ |
|---|---|---|
| $\tau_1$ | 1 | 4 |
| $\tau_2$ | 2 | 6 |

Server

$C_s = 2$

$T_s = 5$




server has current highest priority and checks the queue of tasks

remaining budget is lost

# Rate-Monotonic Polling Server

*Schedulability analysis* of periodic tasks:

- The interference by a server task is the same as the one introduced by an equivalent periodic task in rate-monotonic fixed-priority scheduling.
- 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.

# Rate-Monotonic Polling Server

*Guarantee the response time of aperiodic requests*:

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

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

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

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

Maximal number of necessary server periods.

# EDF – Total Bandwidth Server

***Total Bandwidth Server:***

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

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

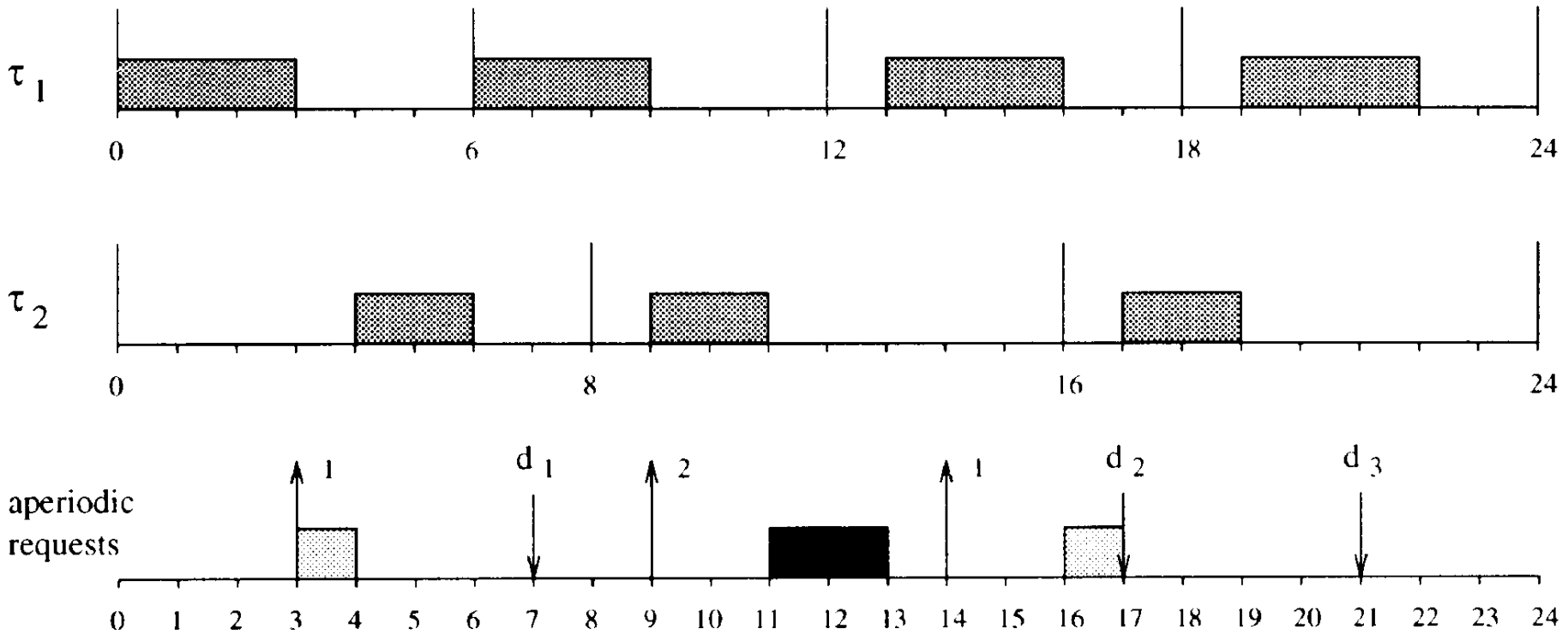
where $C_k$ is the execution time of the request and $U_s$ is the server utilization factor (that is, its bandwidth). By definition, $d_0=0$.

- 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:*

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

# Embedded Systems

# 6a. Example Network Processor

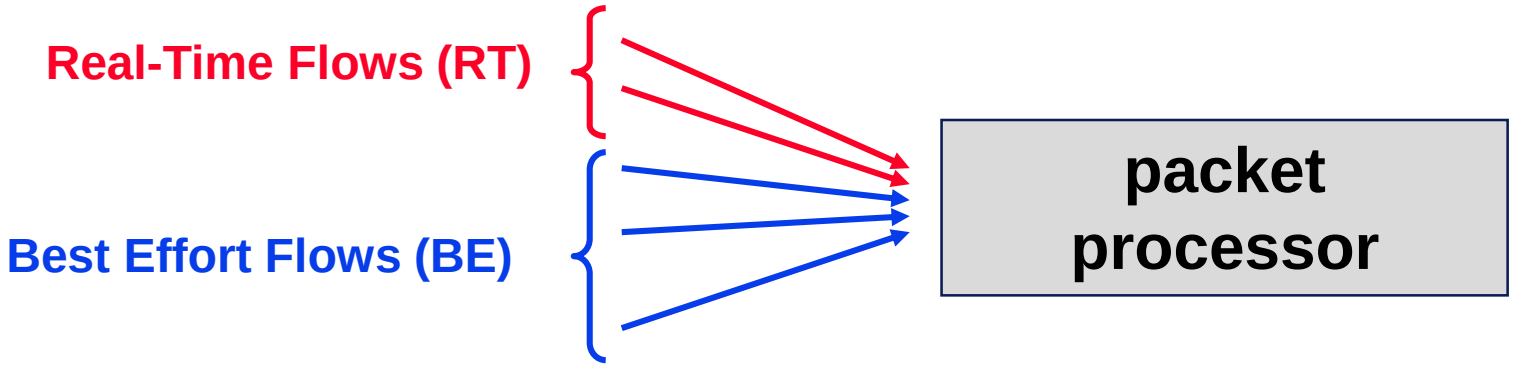
Lothar Thiele

# 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

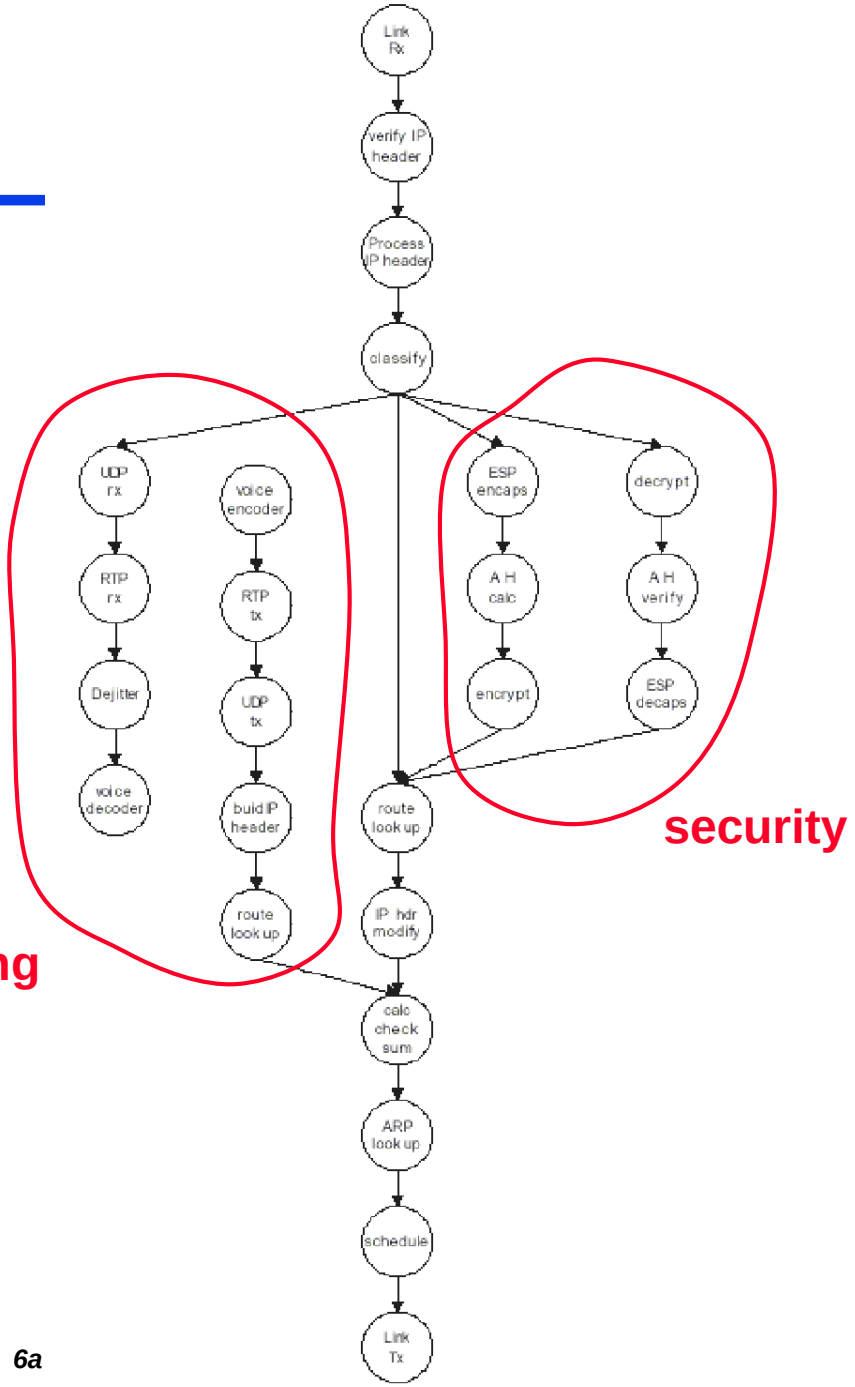
# Our Model – Simple NP



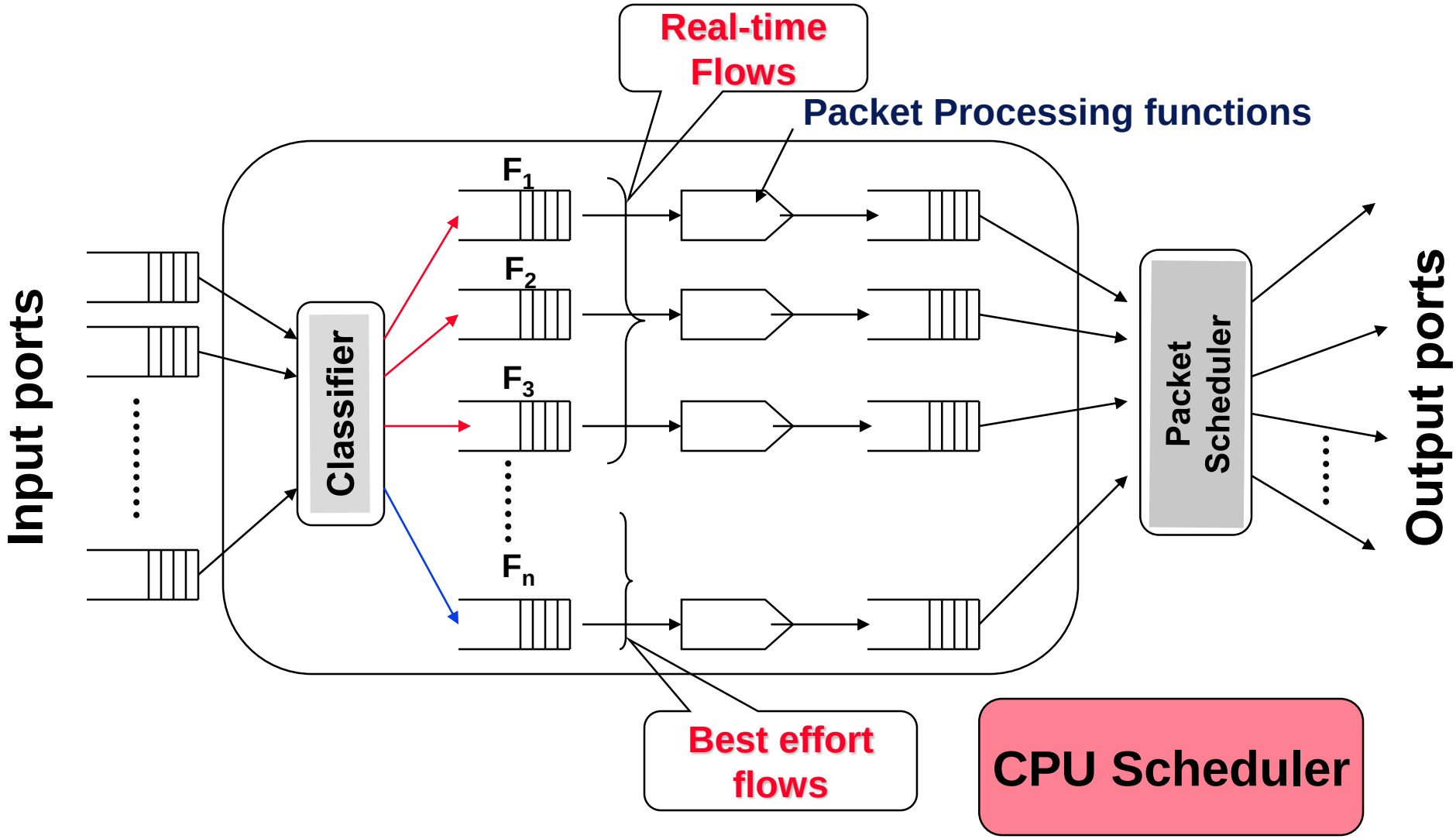

- Real-time flows have deadlines which must be met
- Best effort flows may have several QoS classes and should be served to achieve maximum throughput

# Task Model

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



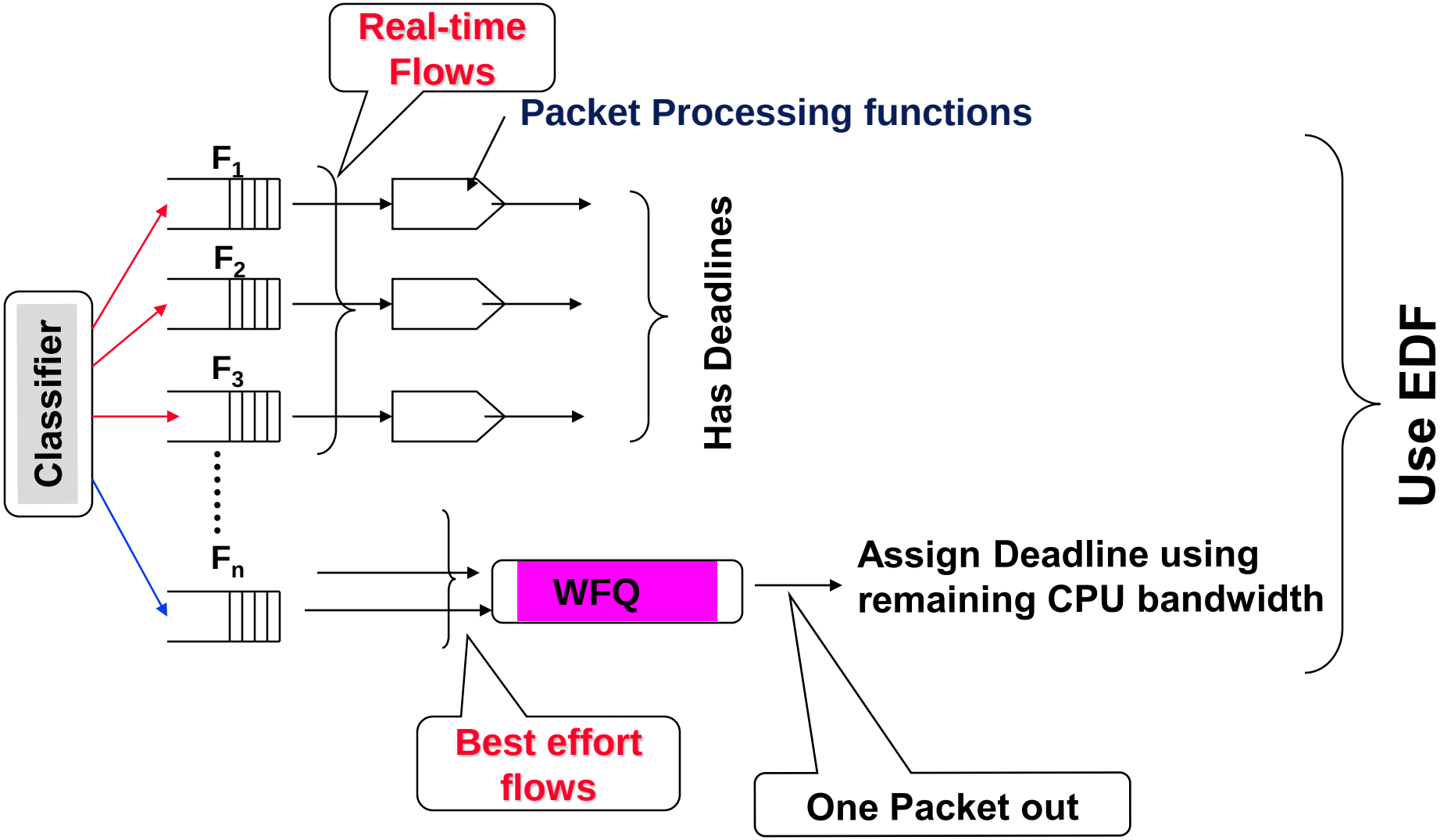
# Architecture



# CPU Scheduling

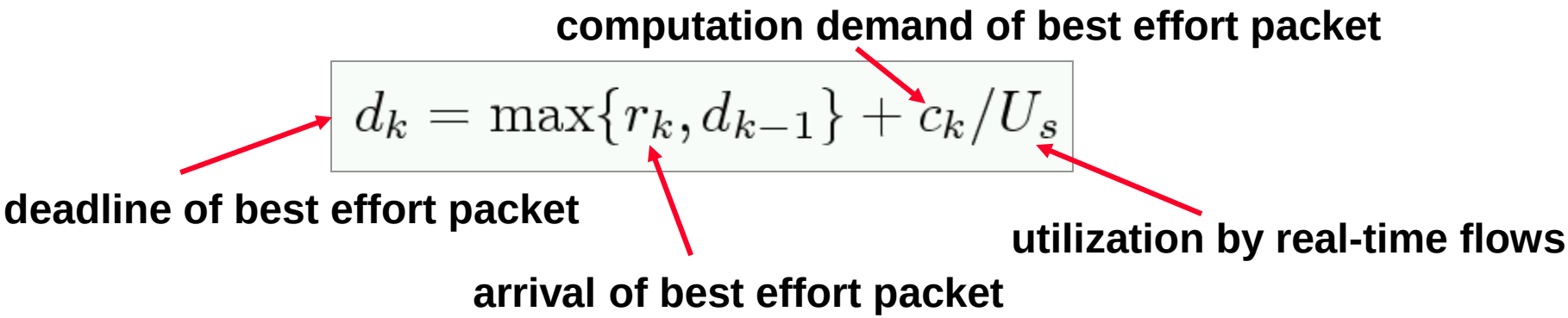
- First Schedule RT, then BE (background scheduling)
  - Overly pessimistic
- Use ***EDF Total Bandwidth Server***
  - EDF for Real-Time tasks
  - Use the remaining bandwidth to server Best Effort Traffic
  - WFQ (weighted fair queuing) to determine which best effort flow to serve; not discussed here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

# CPU Scheduling



# CPU Scheduling

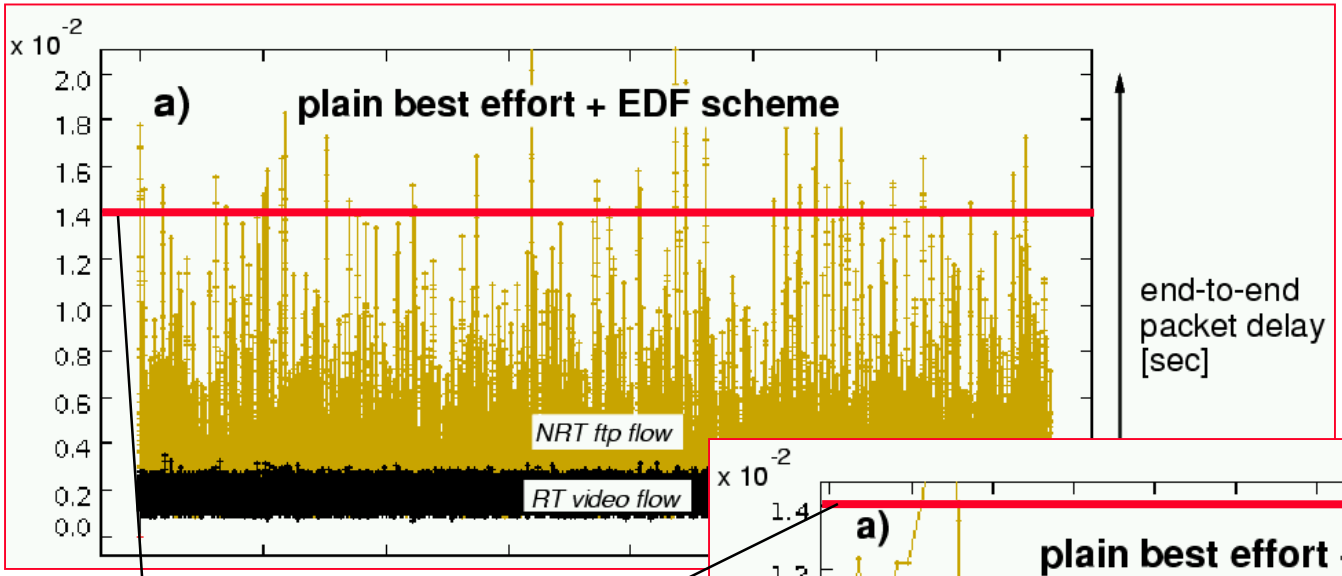
- As discussed, the ***basis is the TBS***:



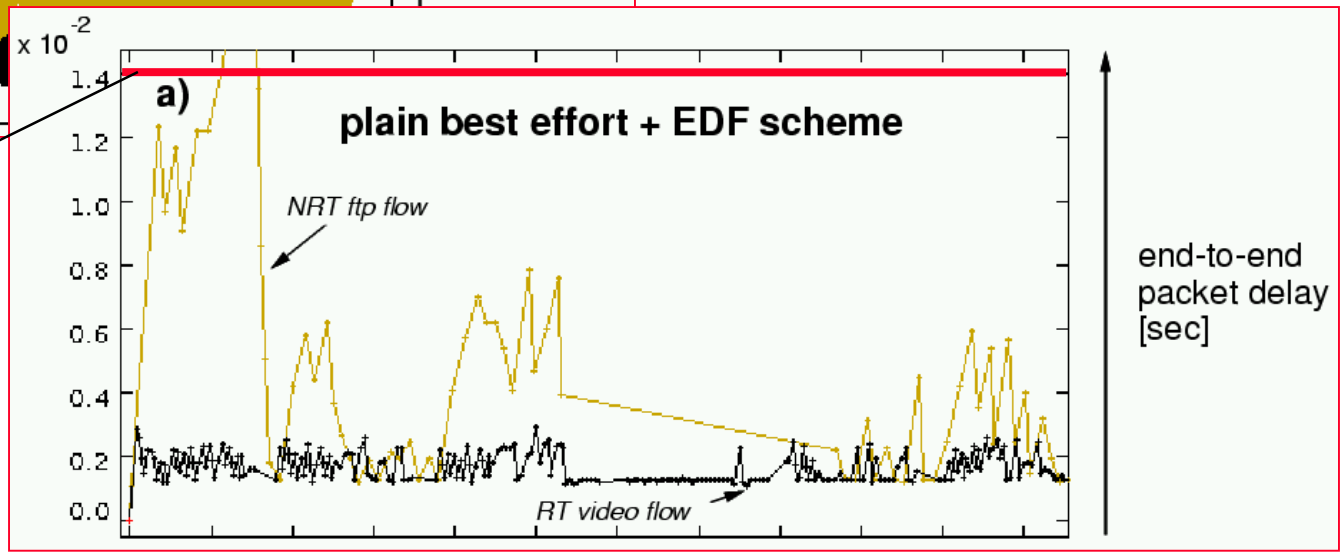

- ***But***: utilization depends on time (packet streams) !
  - Just taking upper bound is too pessimistic
  - Solution with time dependent utilization is (much) more complex – BUT IT HELPS …

# CPU Scheduling

- Before

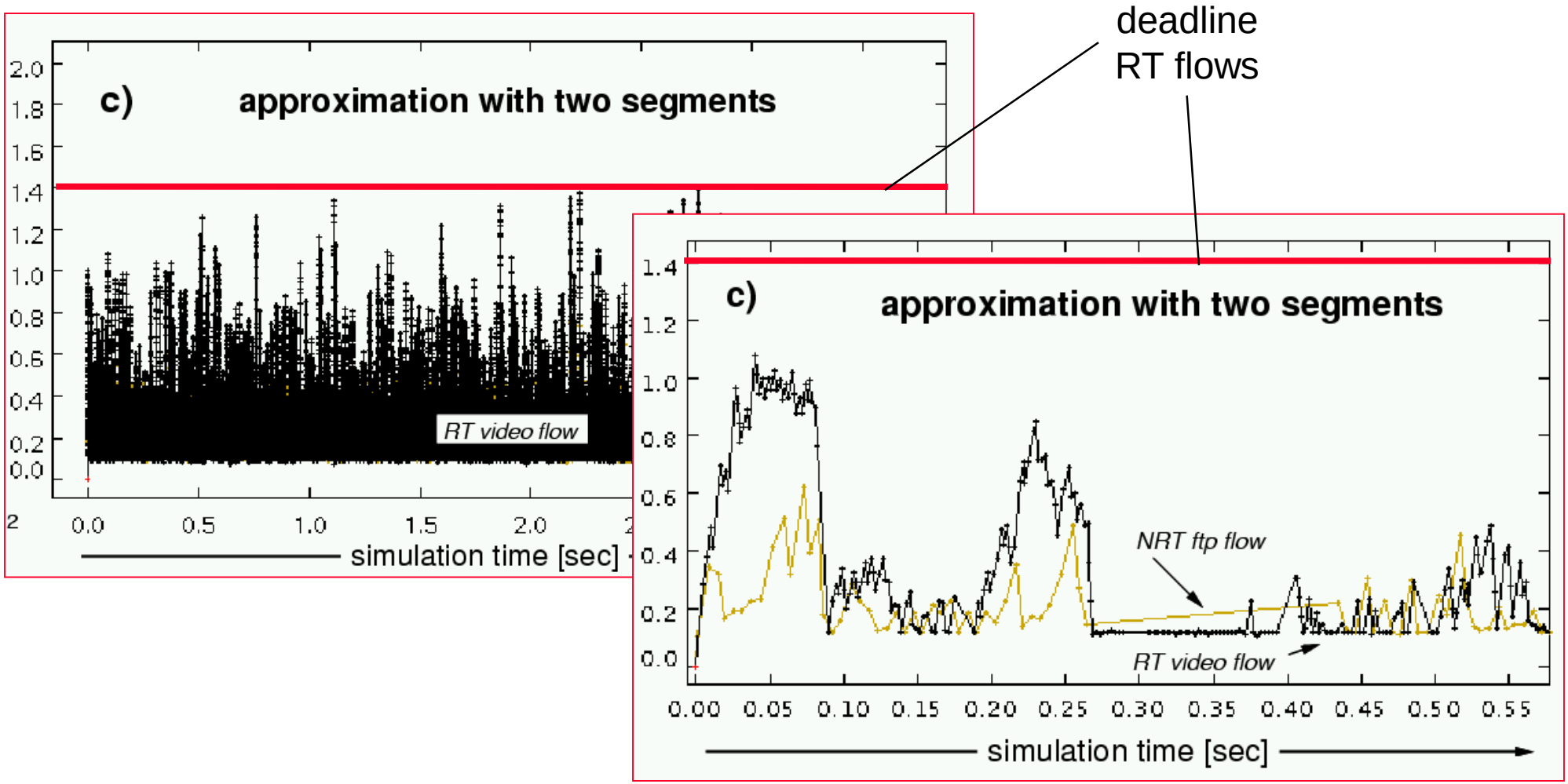



deadline
RT flows



# CPU Scheduling

- After



# Embedded Systems

## 7. Shared Resources

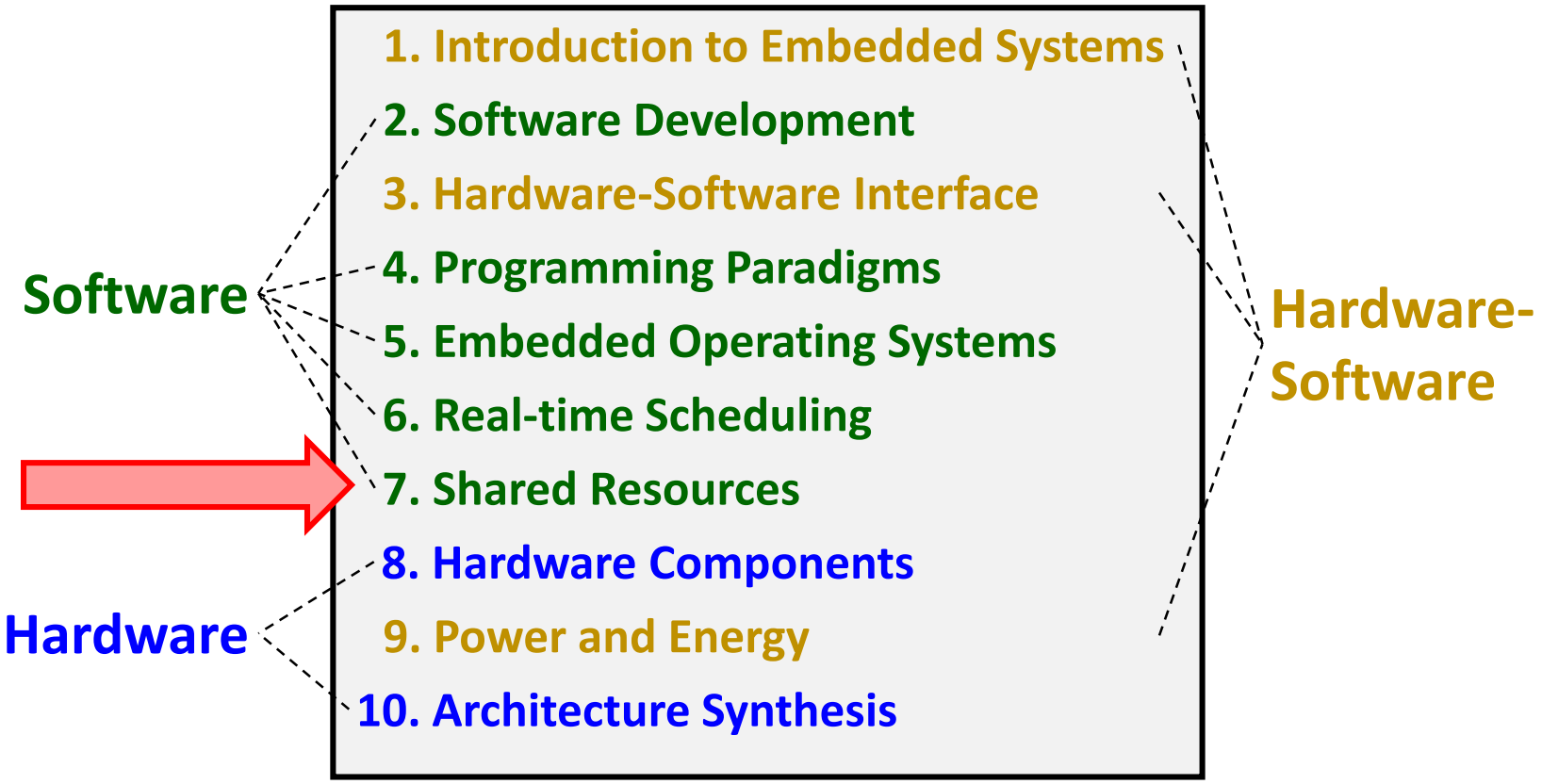
© Lothar Thiele

Computer Engineering and Networks Laboratory

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Where we are …

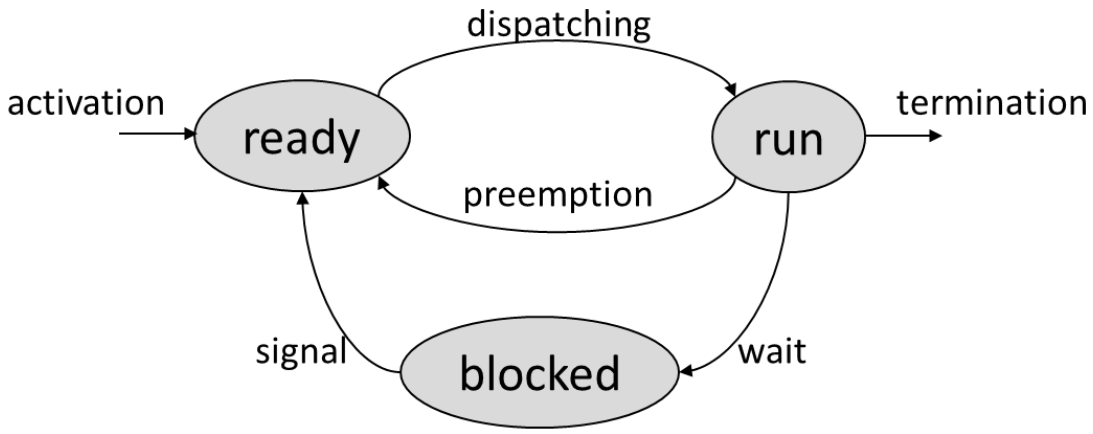

1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis
Software
Hardware
Hardware-Software

# Ressource Sharing

# Resource Sharing

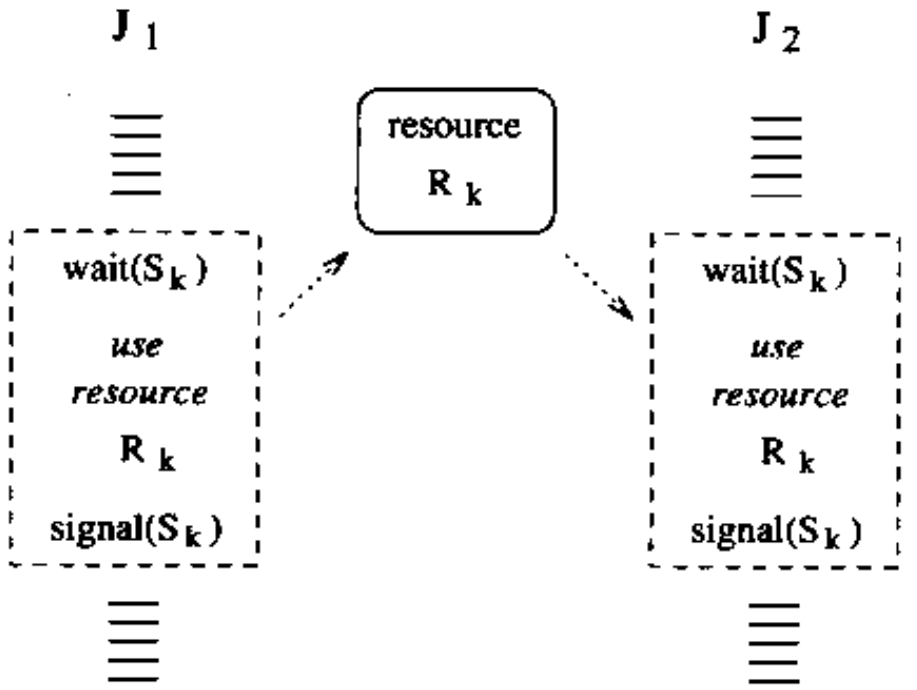
- Examples of *shared resources*: data structures, variables, main memory area, file, set of registers, I/O unit, … .
- Many shared resources do not allow simultaneous accesses but require *mutual exclusion*. These resources are called *exclusive resources*. In this case, no two threads are allowed to operate on the resource at the same time.

- There are several methods available to *protect exclusive resources*, for example
  - *disabling interrupts* and preemption or
  - using concepts like *semaphores and mutex* that put threads into the blocked state if necessary.



# Protecting Exclusive Resources using Semaphores

- Each *exclusive resource* $R_i$ must be protected by a different *semaphore* $S_i$ . Each critical section operating on a resource must begin with a `wait(`$S_i$`)` primitive and end with a `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 *blocked state*, until another tasks executes a *signal* primitive that *unlocks the semaphore*.

# Example FreeRTOS (ES-Lab)

To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a ‘mutual exclusion’ technique.

*One possibility is to disable all interrupts:*

```
...
taskENTER_CRITICAL();
   ... /* access to some exclusive resource */
taskEXIT_CRITICAL();
...
```

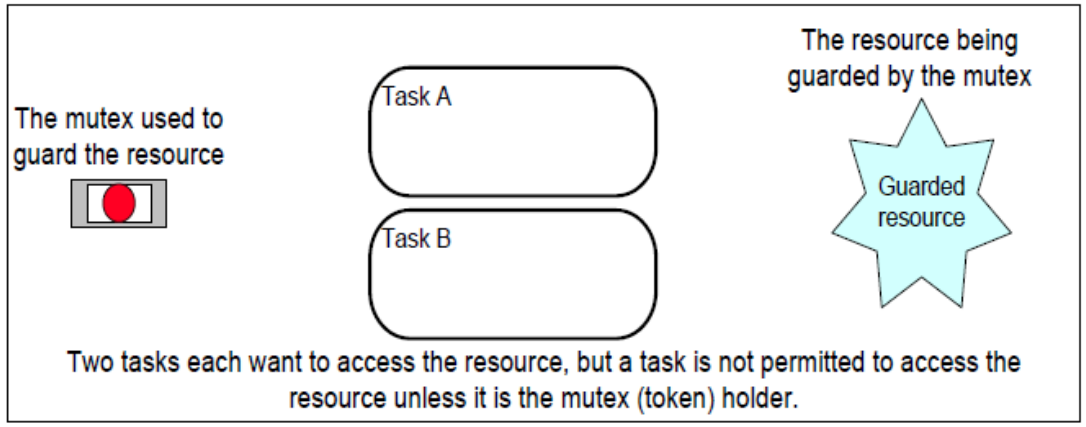
This kind of critical sections must be kept very short, otherwise they will adversely affect interrupt response times.

# Example FreeRTOS (ES-Lab)

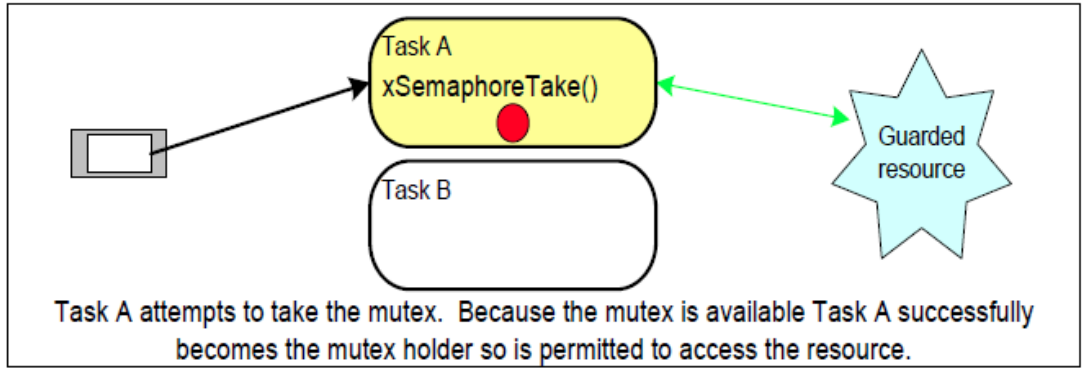
*Another possibility is to use mutual exclusion:* In FreeRTOS, a *mutex* is a special type of *semaphore* that is used to *control access* to a resource that is shared between two or more tasks. *A semaphore that is used for mutual exclusion must always be returned:*

- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared.
- For a task to access the resource legitimately, it must first successfully 'take' the token (be the token holder). When the token holder has finished with the resource, it must 'give' the token back.
- Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource.

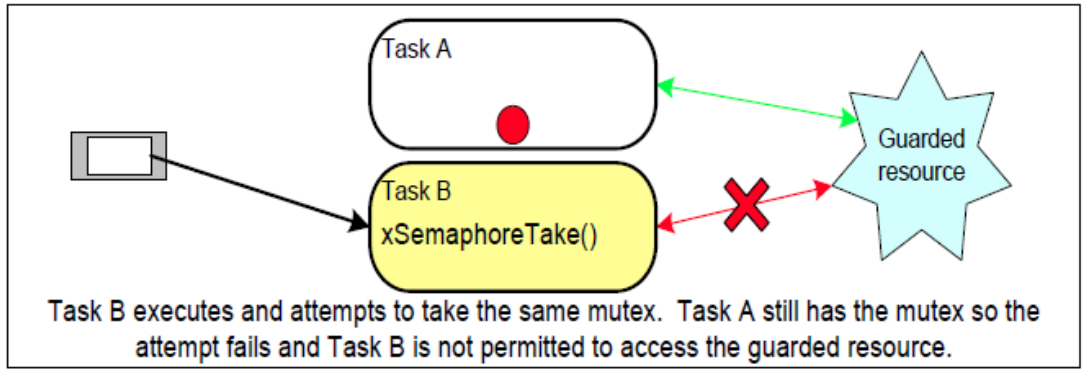
# Example FreeRTOS (ES-Lab)



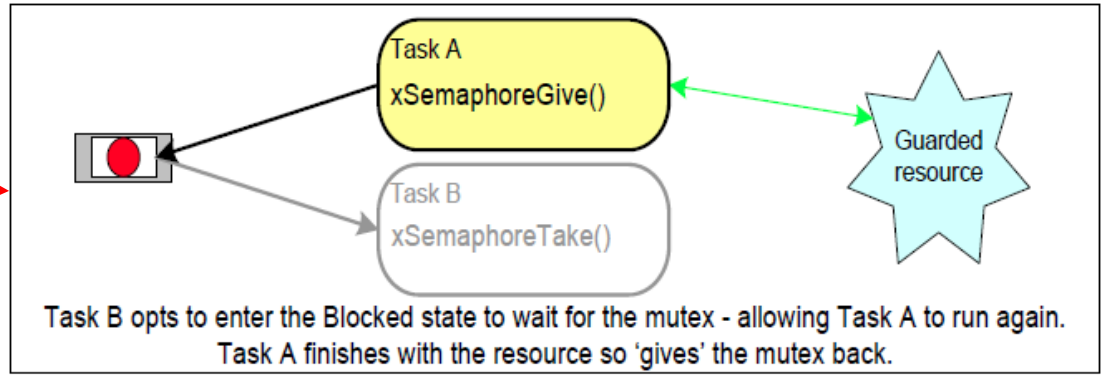

Two tasks each want to access the resource, but a task is not permitted to access the resource unless it is the mutex (token) holder.



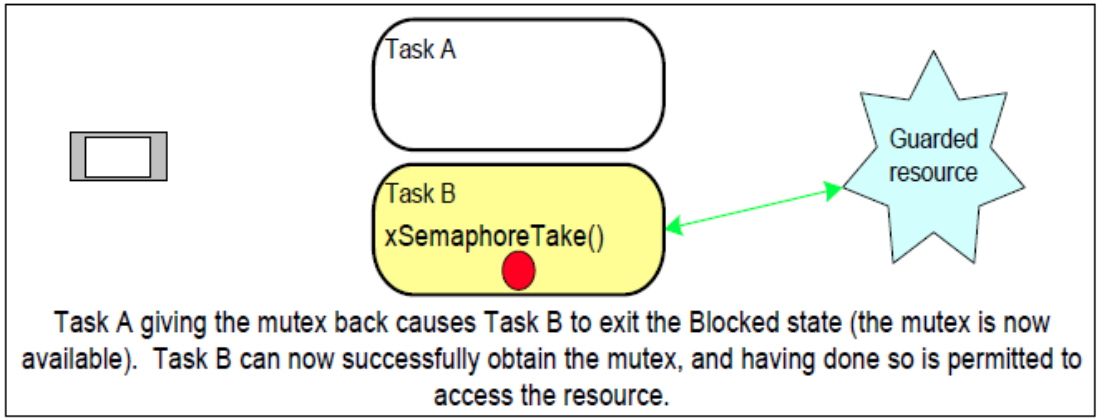

Task A attempts to take the mutex. Because the mutex is available Task A successfully becomes the mutex holder so is permitted to access the resource.



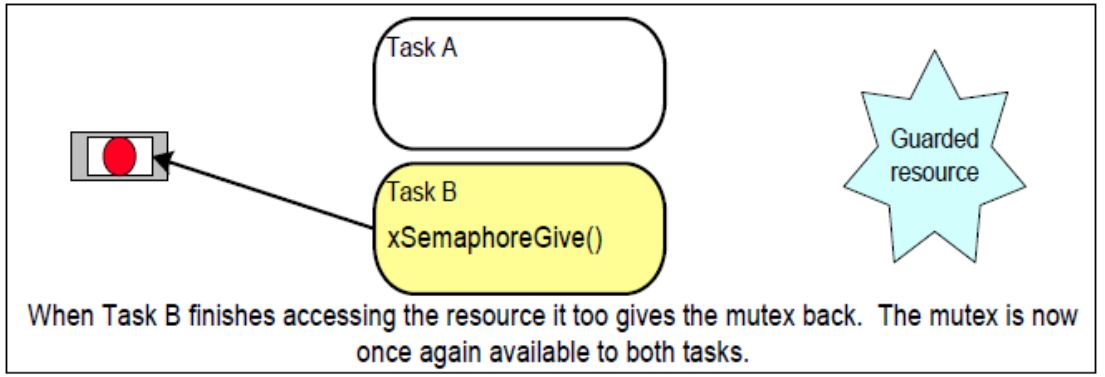

Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.




Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again. Task A finishes with the resource so 'gives' the mutex back.




Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.




When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

# Example FreeRTOS (ES-Lab)

***Example:***

create mutex semaphore

```
SemaphoreHandle_t xMutex;

int main( void ) {
   xMutex = xSemaphoreCreateMutex();
   if( xMutex != NULL ) {
      xTaskCreate(vTask1,"Task1",1000,NULL,1,NULL);
      xTaskCreate(vTask2,"Task2",1000,NULL,2,NULL);
      vTaskStartScheduler();
   }
   for( ;; );
}
```

some defined constant for infinite timeout; otherwise, the function would return if the mutex was not available for the specified time

```
void vTask1( void *pvParameters ) {
   for( ;; ) {
      ...
      xSemaphoreTake(xMutex,portMAX_DELAY);
         ... /* access to exclusive resource */
      xSemaphoreGive(xMutex);
      ... }
}
```

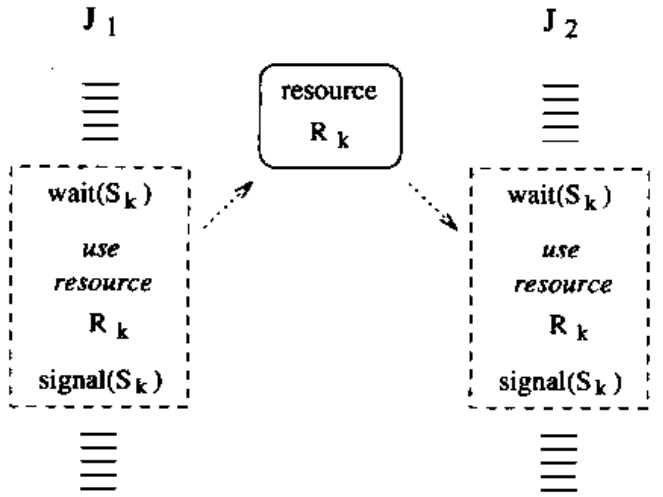
```
void vTask2( void *pvParameters ) {
   for( ;; ) {
      ...
      xSemaphoreTake(xMutex,portMAX_DELAY);
         ... /* access to exclusive resource */
      xSemaphoreGive(xMutex);
      ... }
}
```

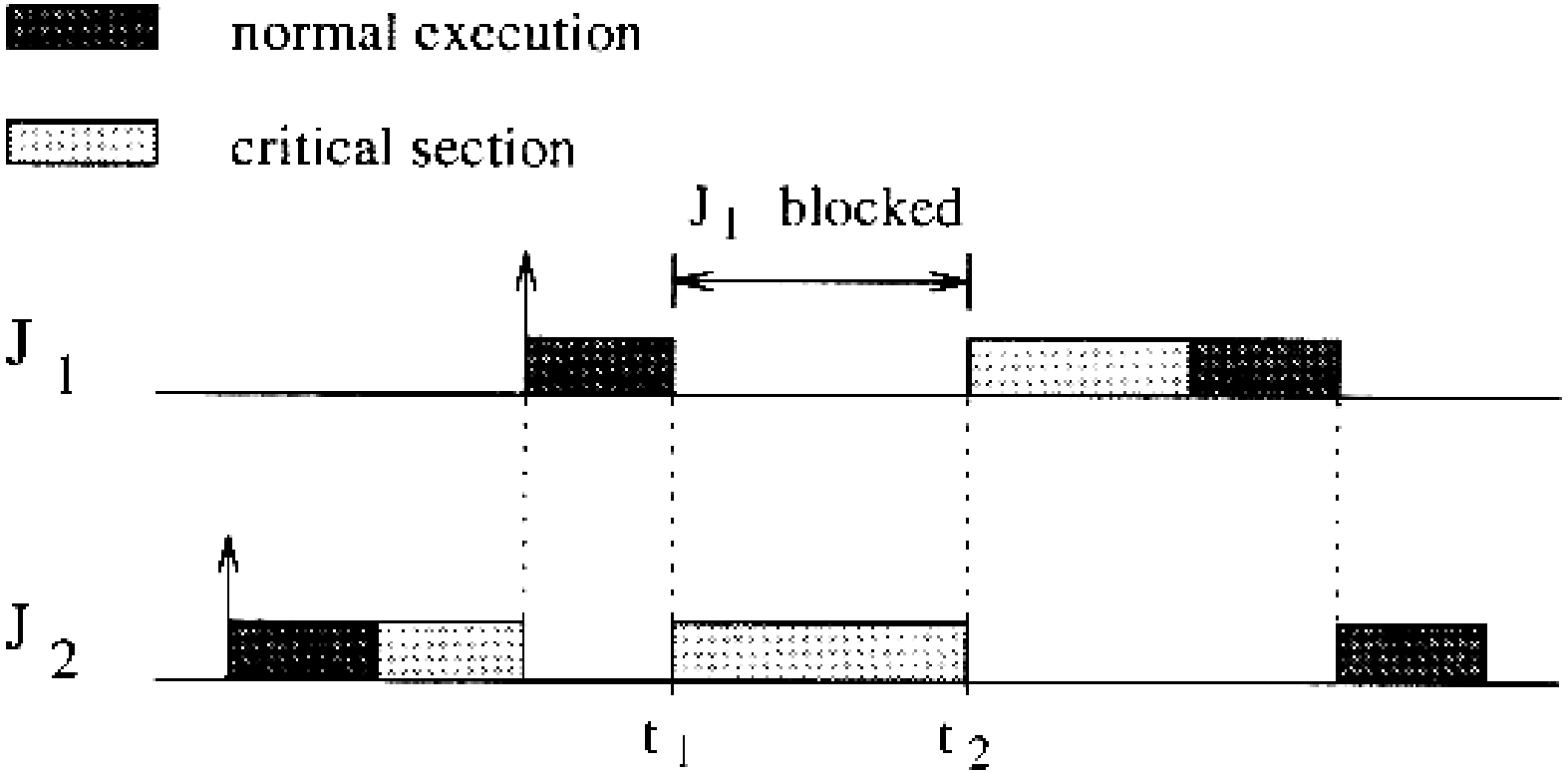
# Ressource Sharing

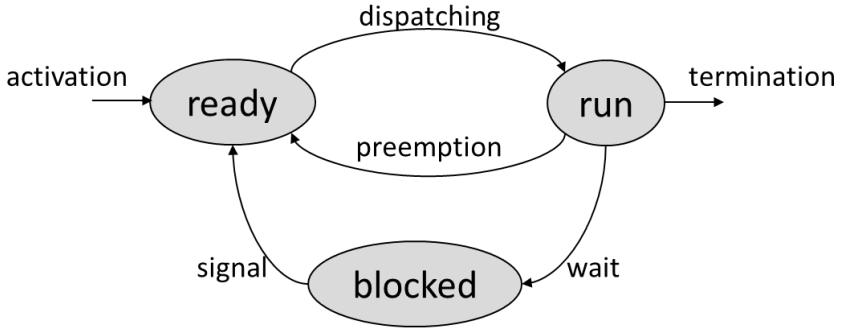
## Priority Inversion

# Priority Inversion (1)

*Unavoidable blocking:*

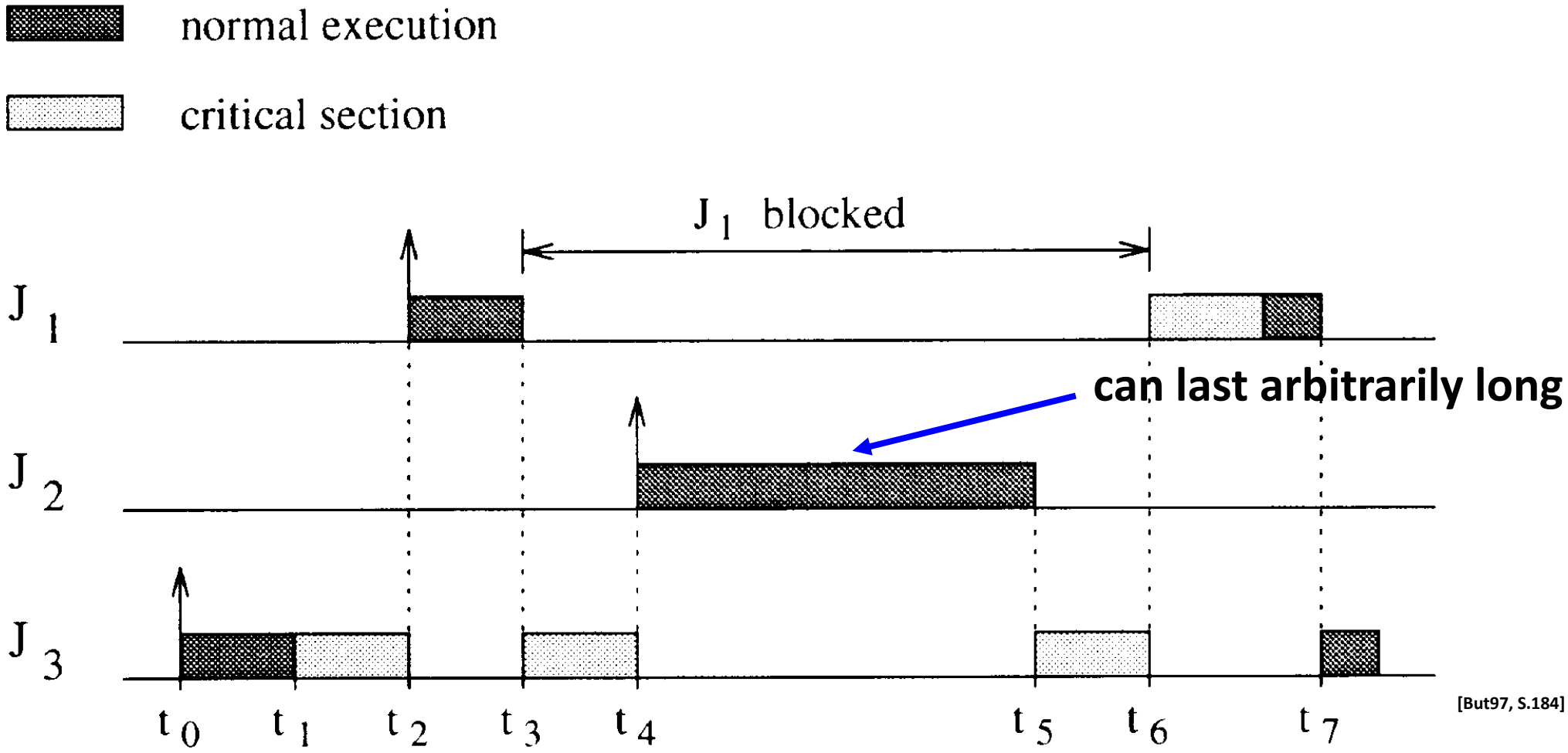


# Priority Inversion (2)

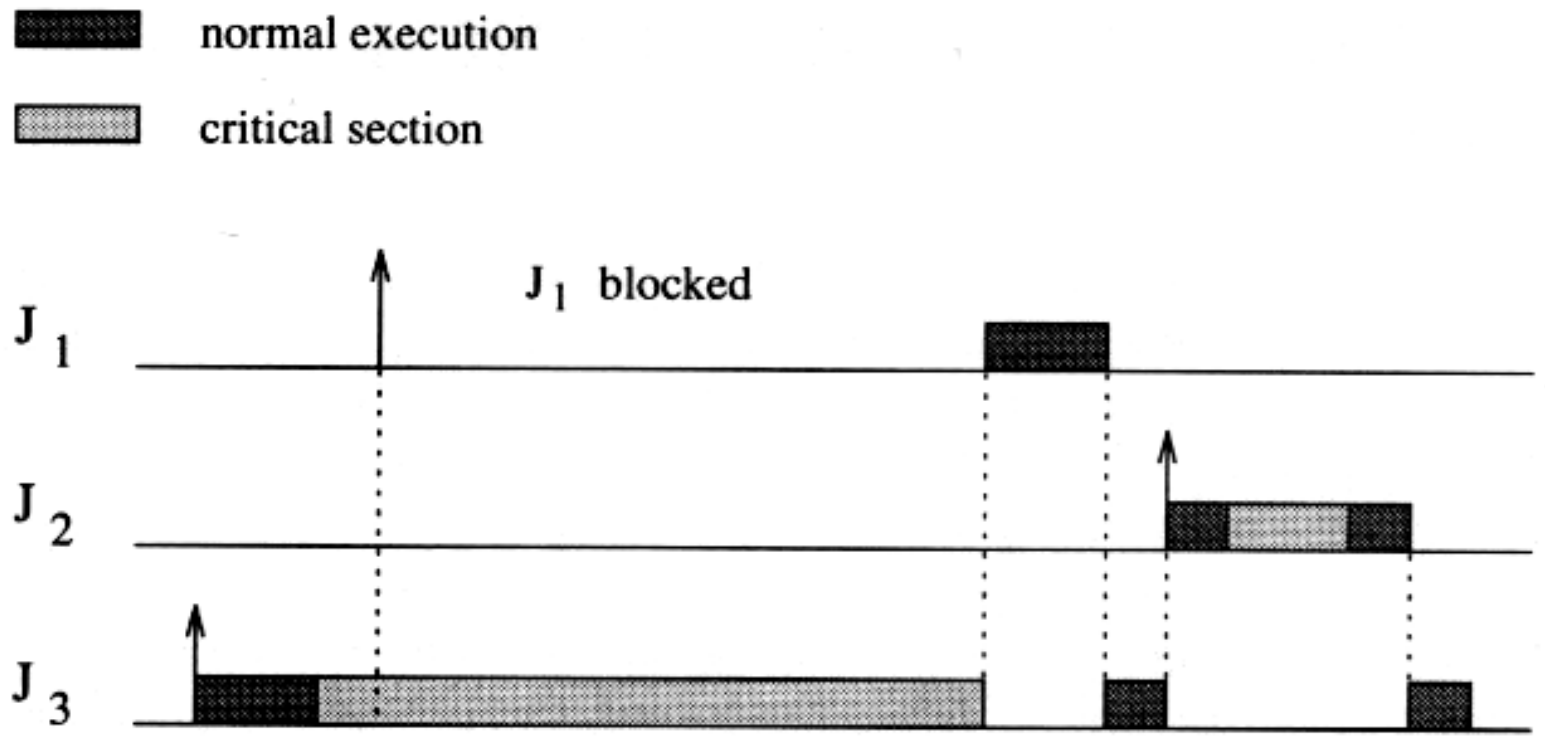
***Priority Inversion:***




[But97, S.184]

# Solutions to Priority Inversion

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



# 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.

*Specific 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 `wait(`$S_i$`)` and end with a `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 $J_1$, …$J_n$ are ordered with respect to nominal priority where $J_1$ has *highest priority*. Jobs do not suspend themselves.

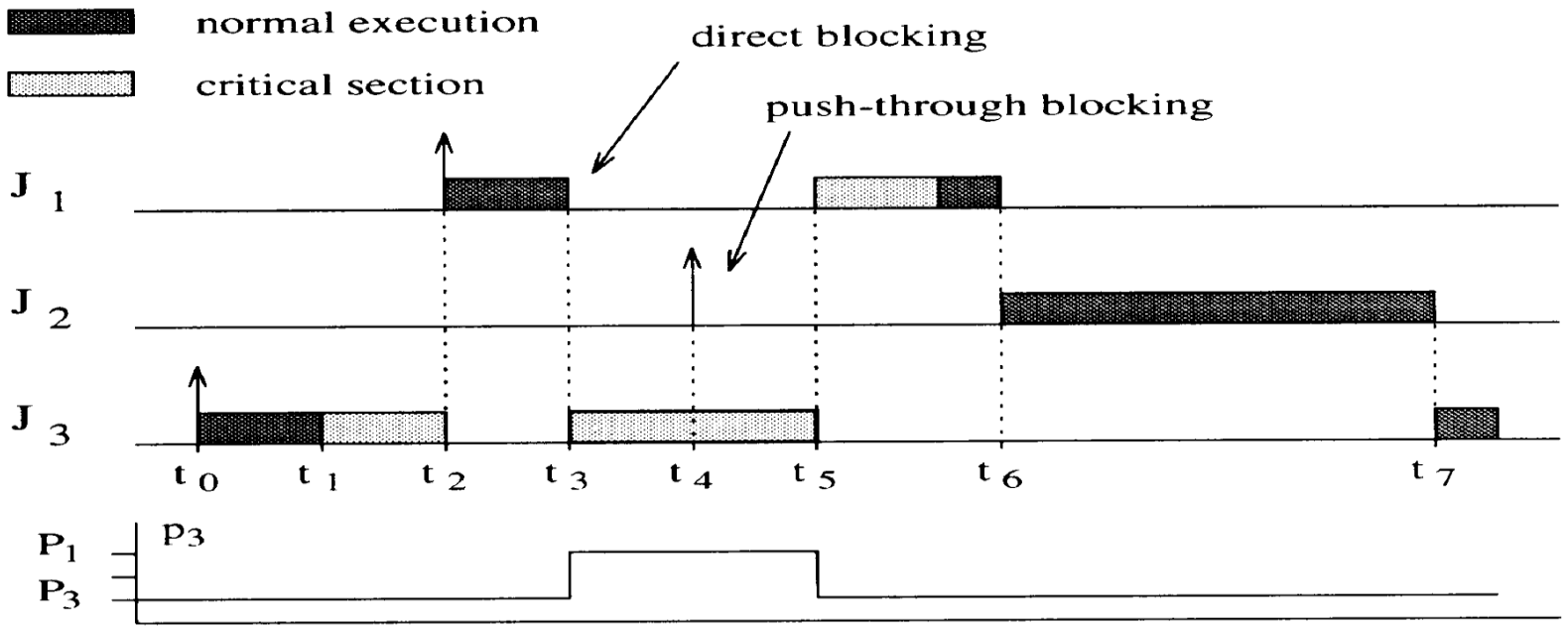
# Priority Inheritance Protocol (PIP)

*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:***

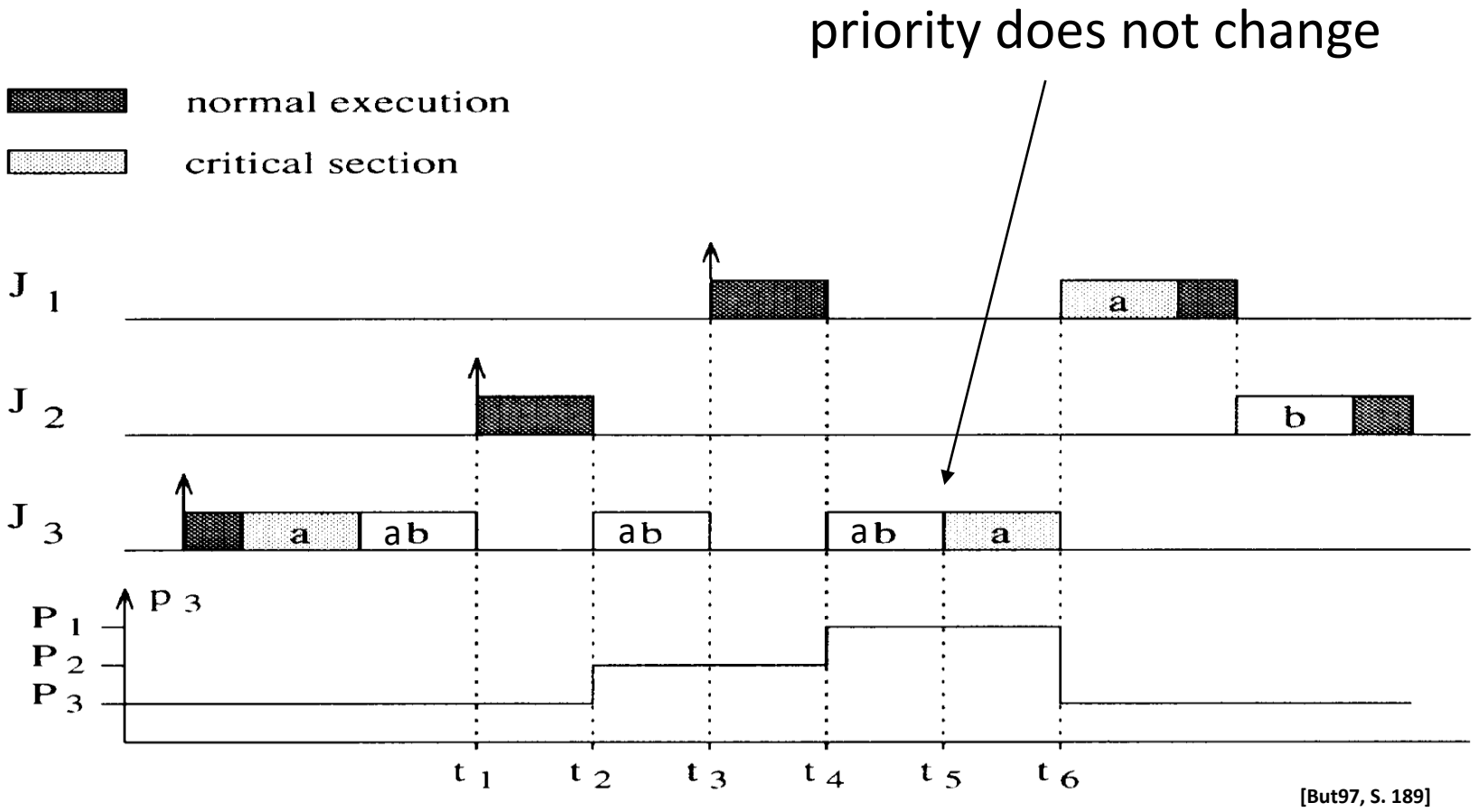



*Direct Blocking*: higher-priority job tries to acquire a resource held by a lower-priority job

*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)

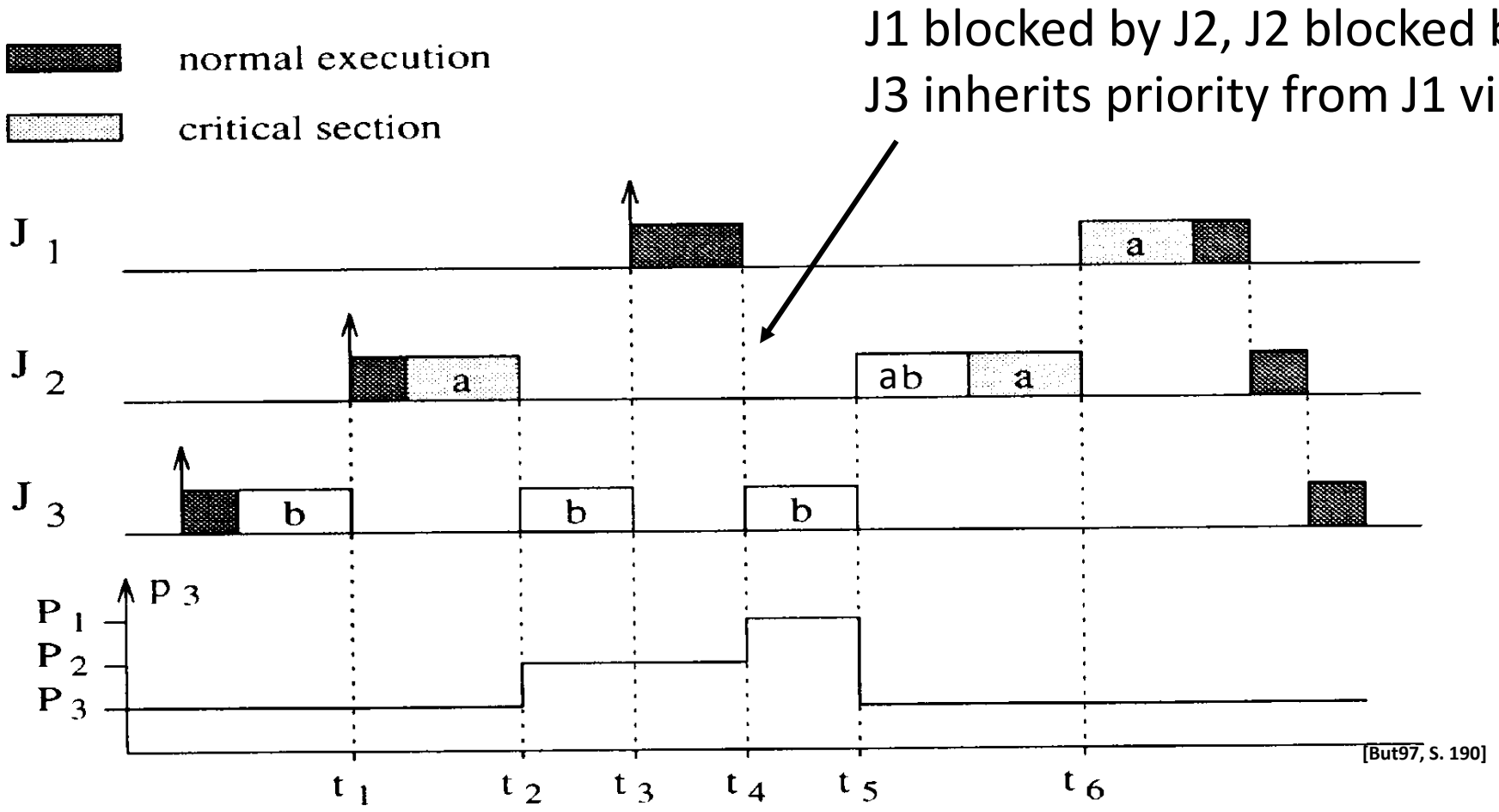
*Example with nested critical sections:*




[But97, S. 189]

# Priority Inheritance Protocol (PIP)

*Example of transitive priority inheritance:*

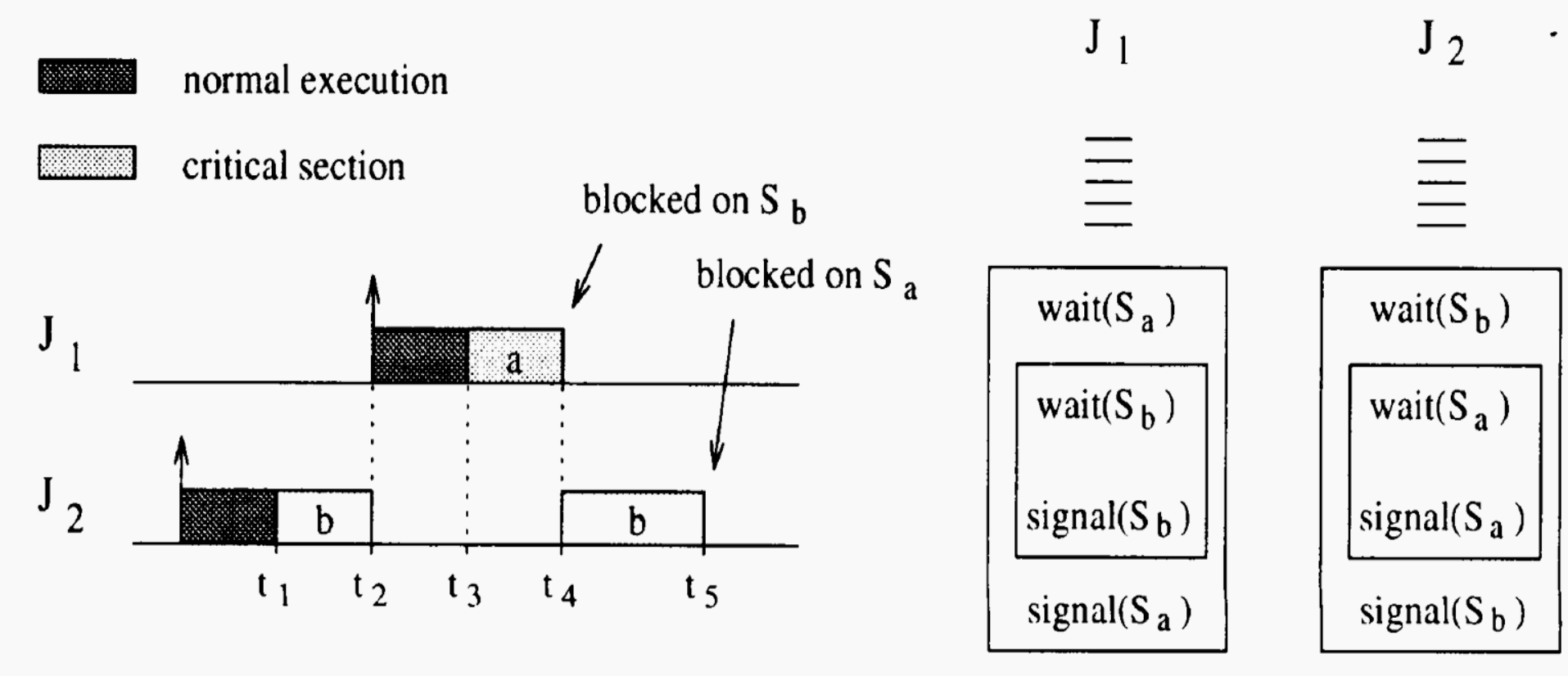



[But97, S. 190]

# Priority Inheritance Protocol (PIP)

*Still a Problem: Deadlock*

…. but there are other protocols like the Priority Ceiling Protocol …




[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.

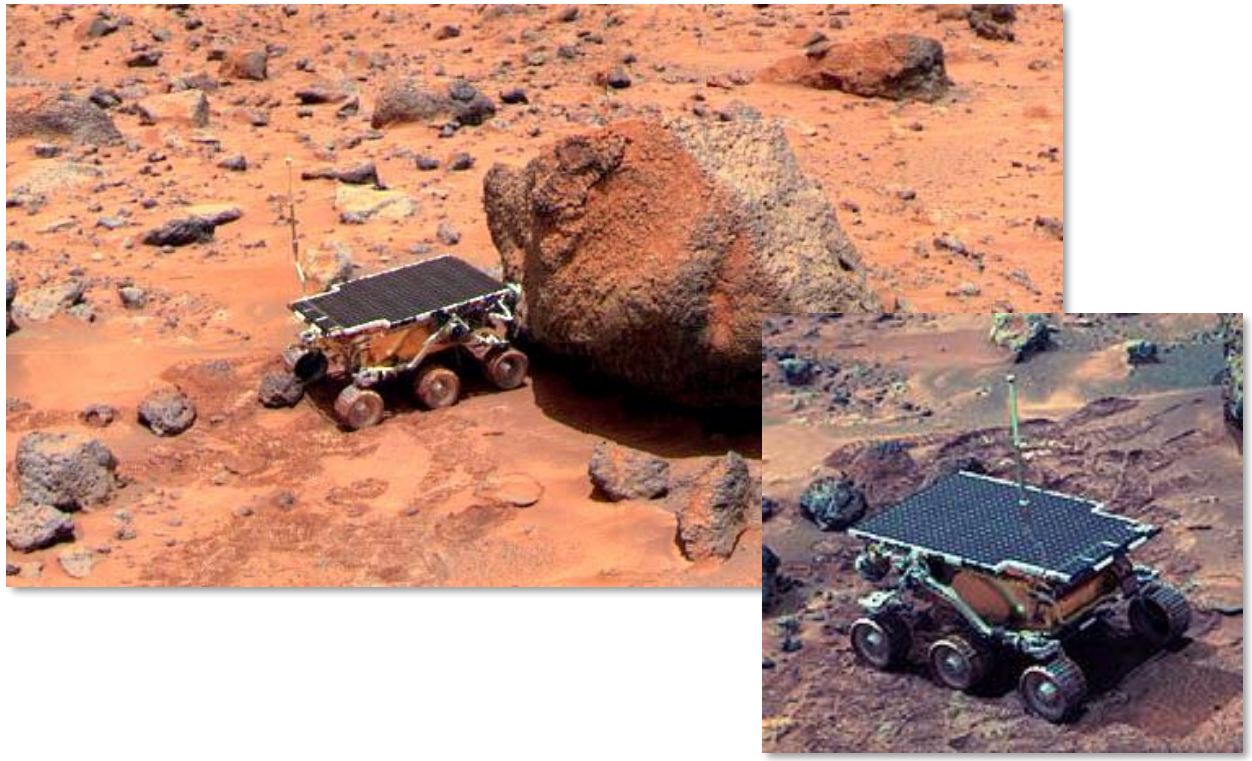
However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the 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].



# Timing Anomalies

# Timing Anomaly

Suppose, a real-time system works correctly with a given processor architecture. Now, you replace the processor with a faster one.

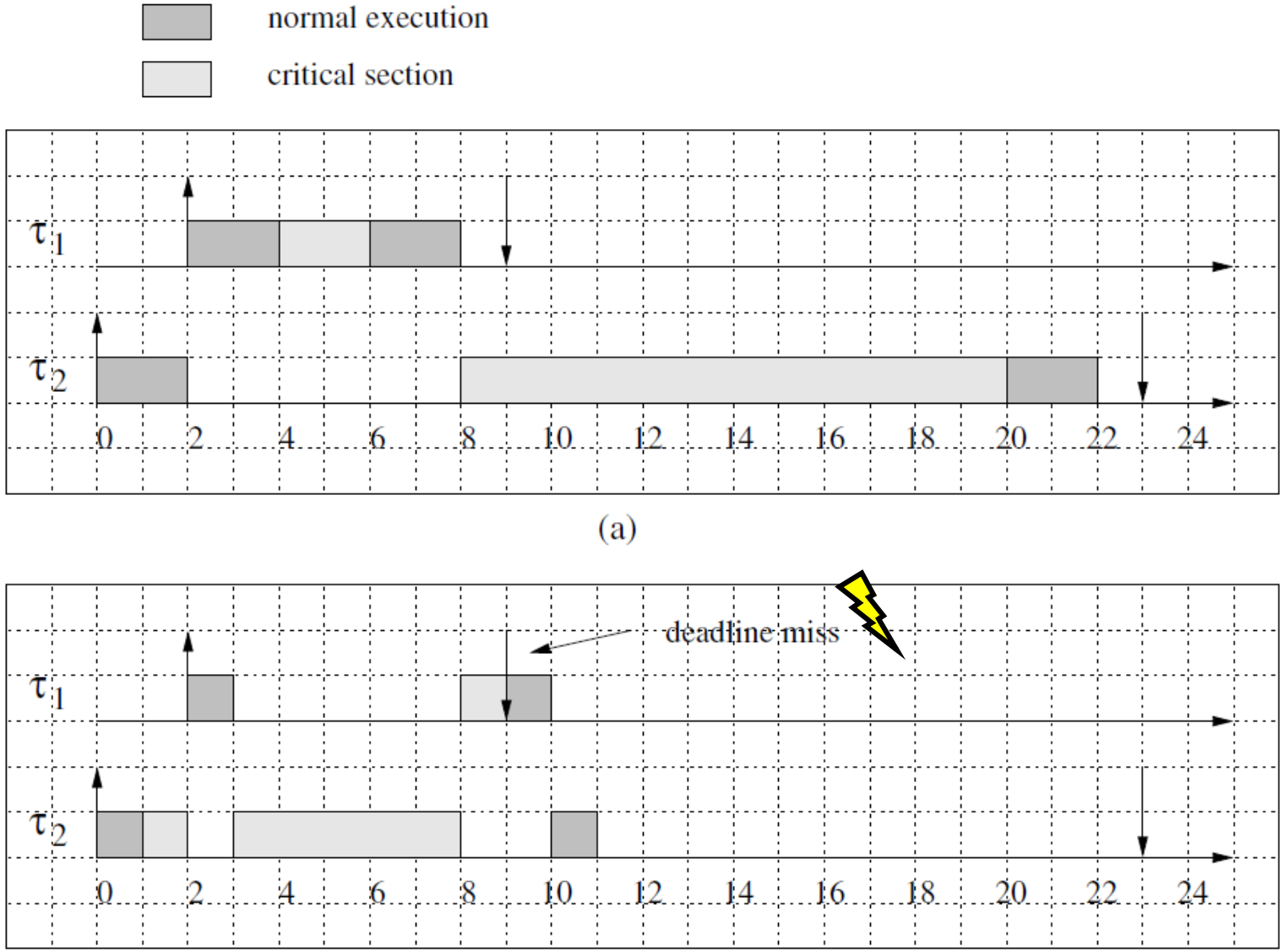
Are real-time constraints still satisfied?

Unfortunately, this is not true in general. *Monotonicity* does not hold in general, i.e., making a part of the system operate faster does not lead to a faster system execution. In other words, *many software and systems architectures are fragile.*

There are usually many timing anomalies in a system, starting from the microarchitecture (caches, pipelines, speculation) via single processor scheduling to multiprocessor scheduling.

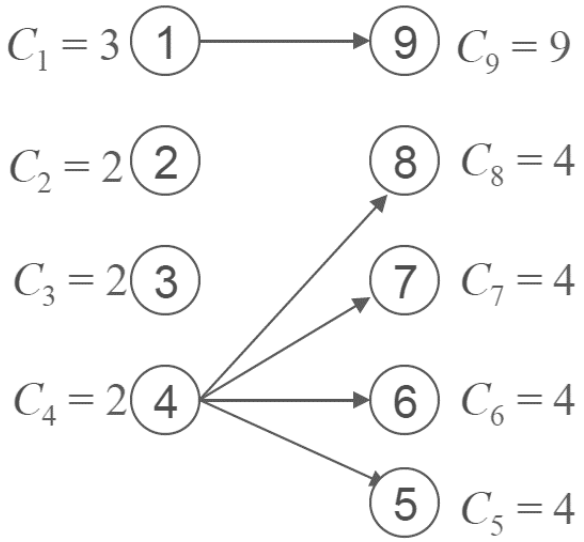
# Single Processor with Critical Sections

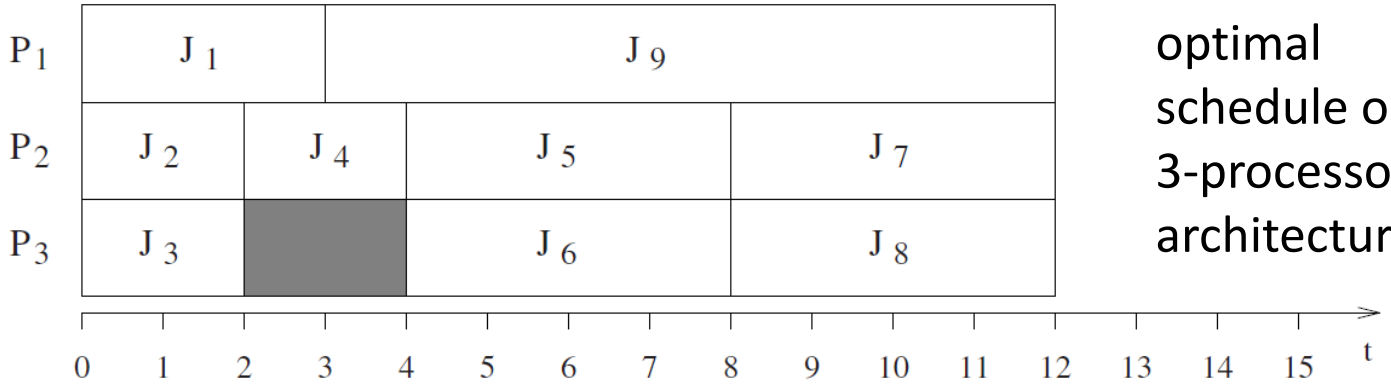
*Example:* Replacing the processor with one that is twice as fast leads to a deadline miss.



# Multiprocessor Example (Richard's Anomalies)

*Example:* 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.

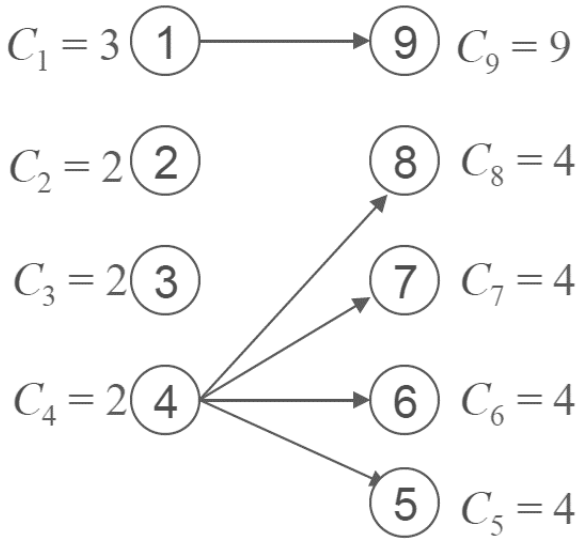


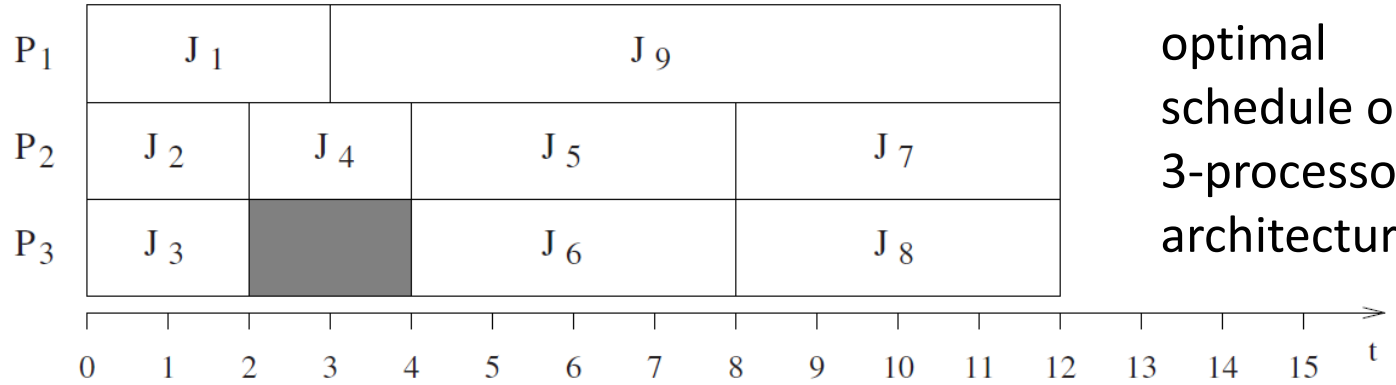




optimal schedule on a 3-processor architecture

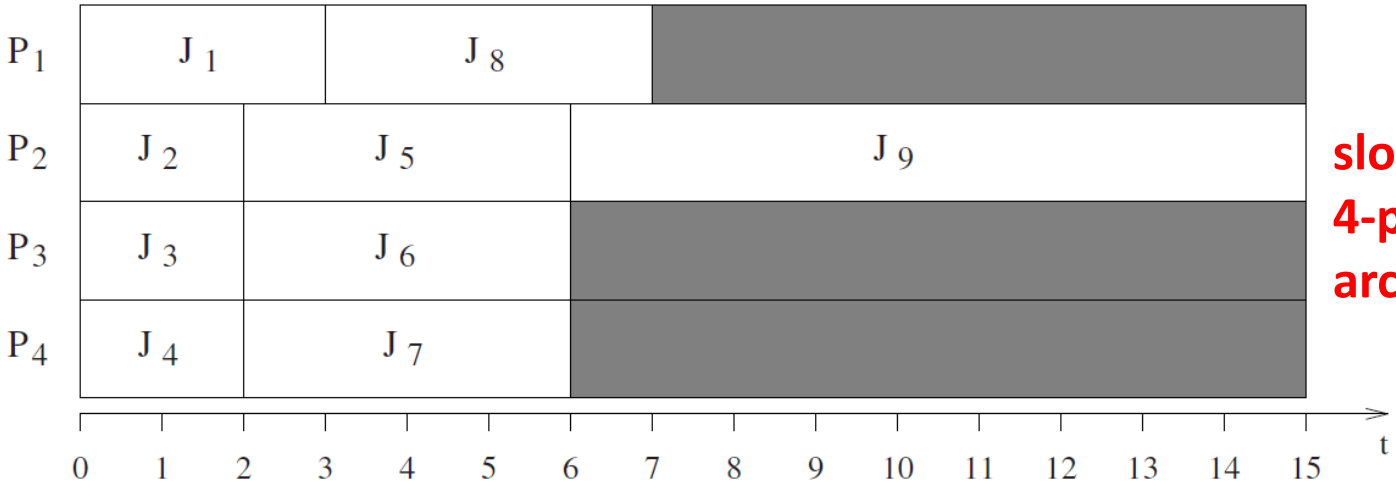
# Multiprocessor Example (Richard's Anomalies)

*Example:* 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.



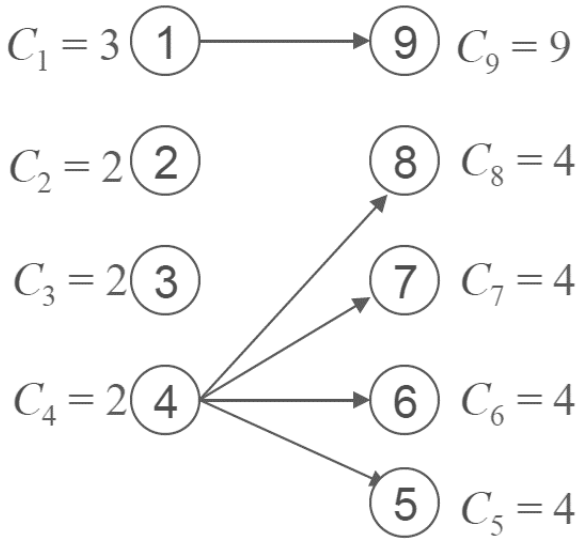

optimal schedule on a 3-processor architecture

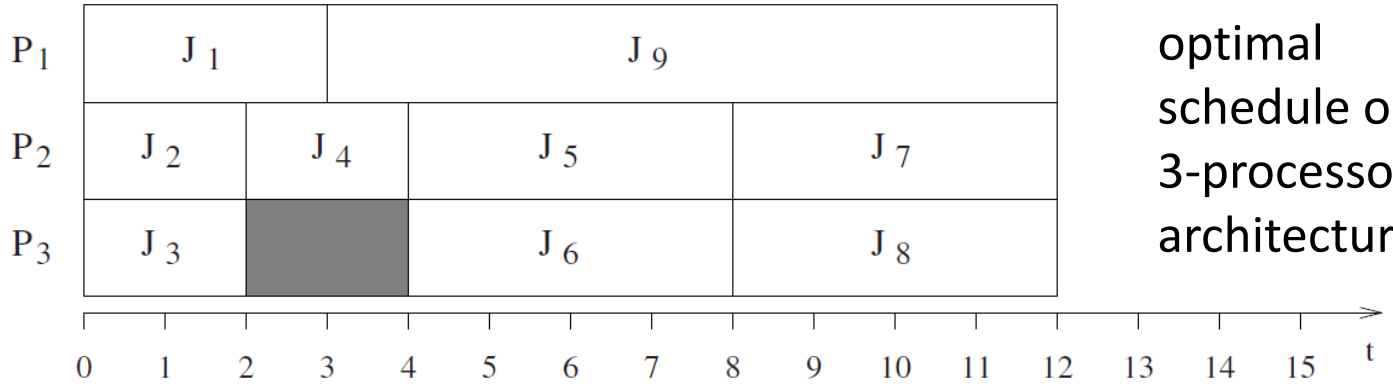



**slower on a 4-processor architecture!**

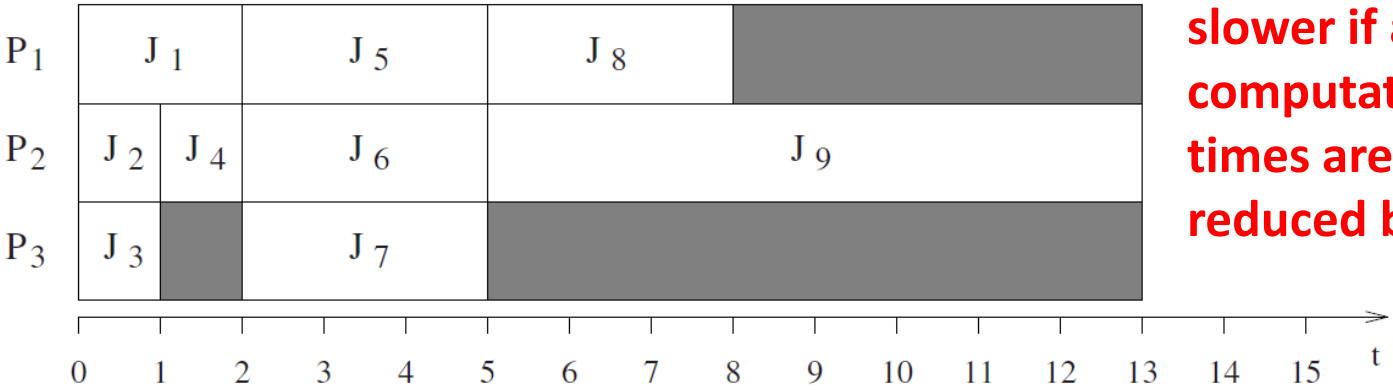
# Multiprocessor Example (Richard's Anomalies)

*Example:* 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.



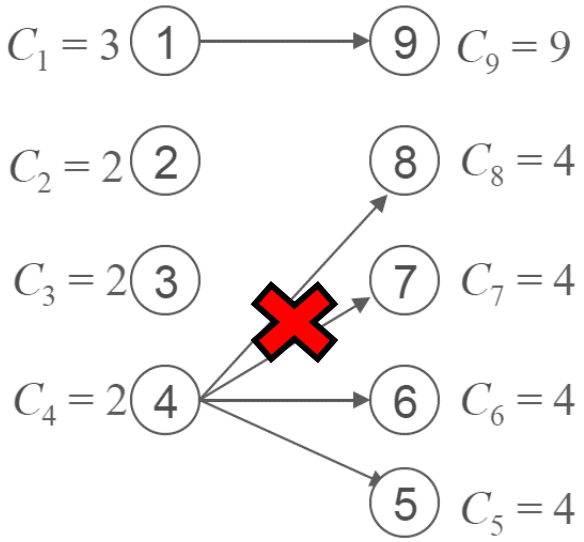

optimal schedule on a 3-processor architecture

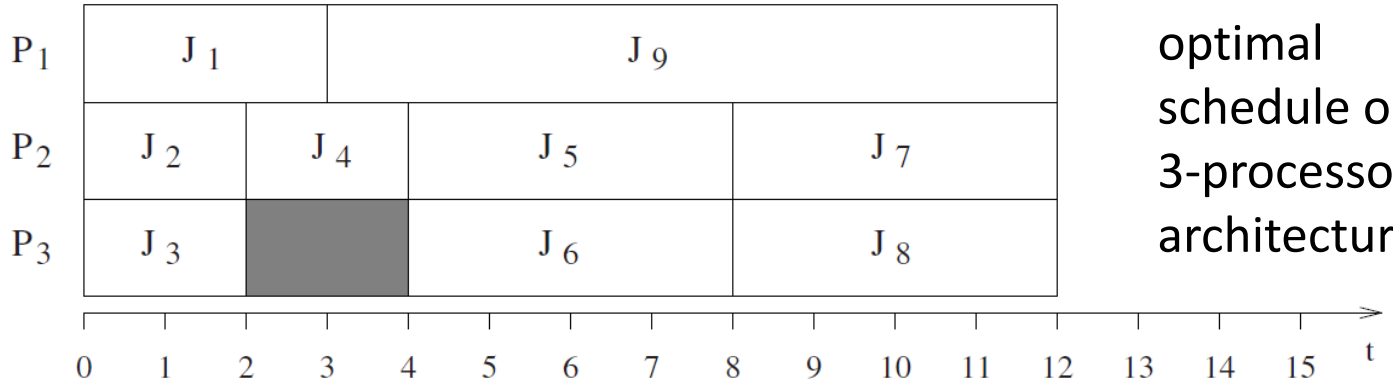



**slower if all computation times are reduced by 1!**

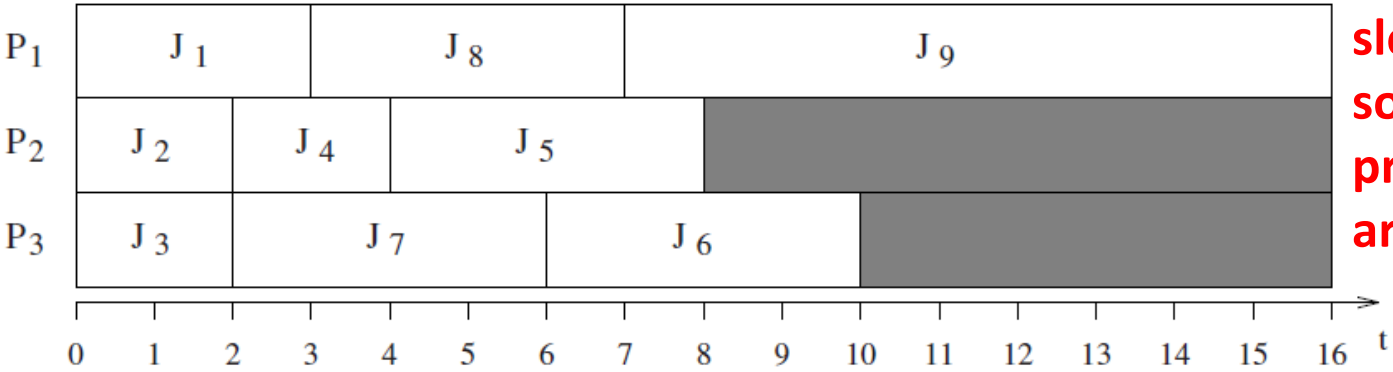
# Multiprocessor Example (Richard's Anomalies)

*Example:* 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower numbered tasks have higher priority than higher numbers. Assignment of tasks to processors is greedy.




optimal schedule on a 3-processor architecture



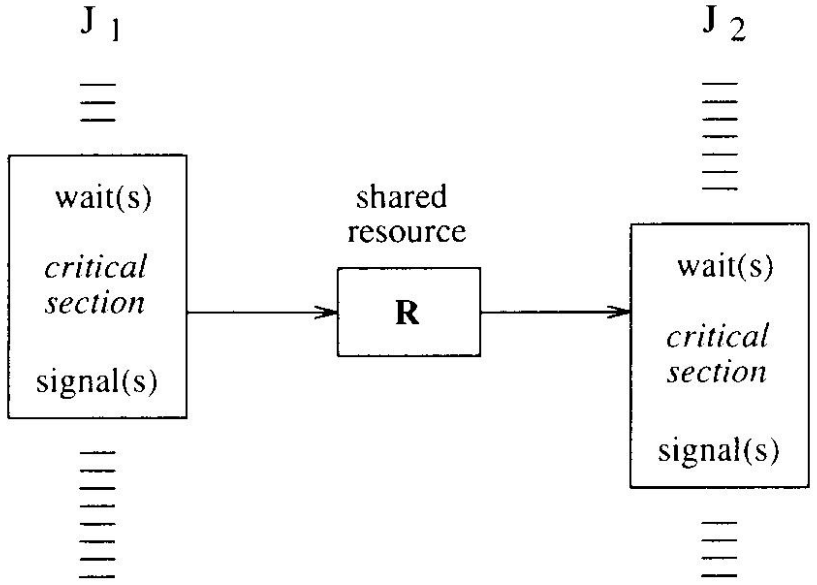

**slower if some precedences are removed!**

# Communication and Synchronization

# Communication Between Tasks

*Problem:* the use of shared memory for implementing communication between tasks may cause priority inversion and blocking.

Therefore, either the implementation of the shared medium is "thread safe" or the data exchange must be *protected by critical sections*.



# Communication Mechanisms

*Synchronous communication:*

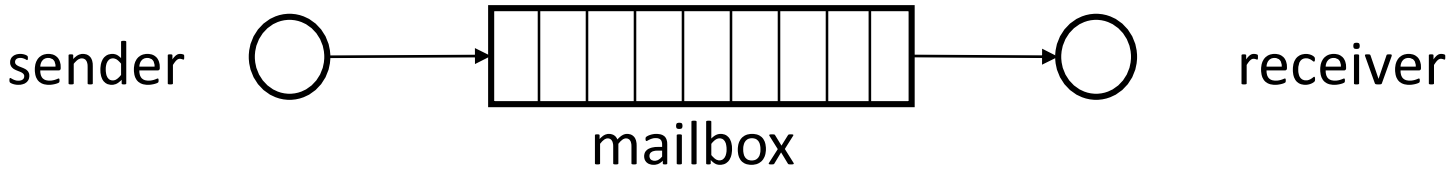
- Whenever two tasks want to communicate they must be synchronized for a message transfer to take place (rendez-vous).
- They have to wait for each other, i.e. both must be at the same time ready to do the data exchange.

- *Problem:*
  - In case of dynamic real-time systems, estimating the maximum blocking time for a process rendez-vous is difficult.
  - Communication always needs synchronization. Therefore, the timing of the communication partners is closely linked.

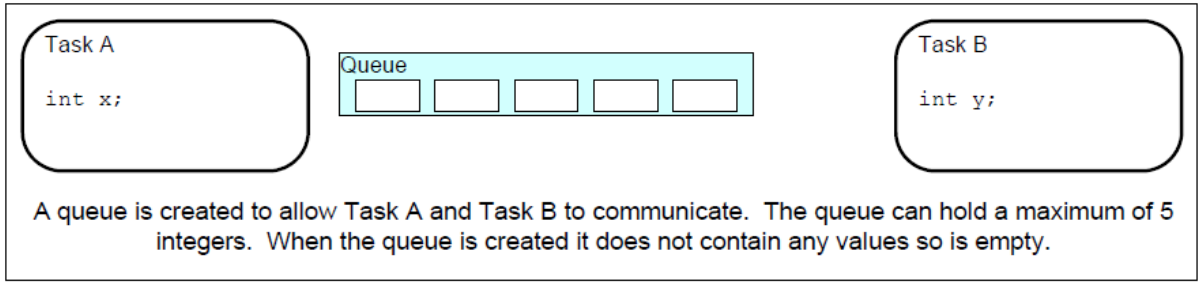
# Communication Mechanisms

*Asynchronous communication:*

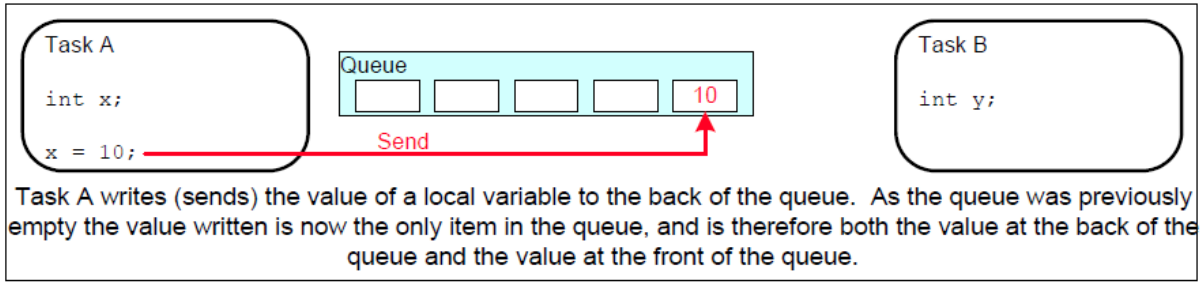
- Tasks do not necessarily have to wait for each other.
- The sender just deposits its message into a channel and continues its execution; similarly the receiver can directly access the message if at least a message has been deposited into the channel.
- More suited for real-time systems than synchronous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



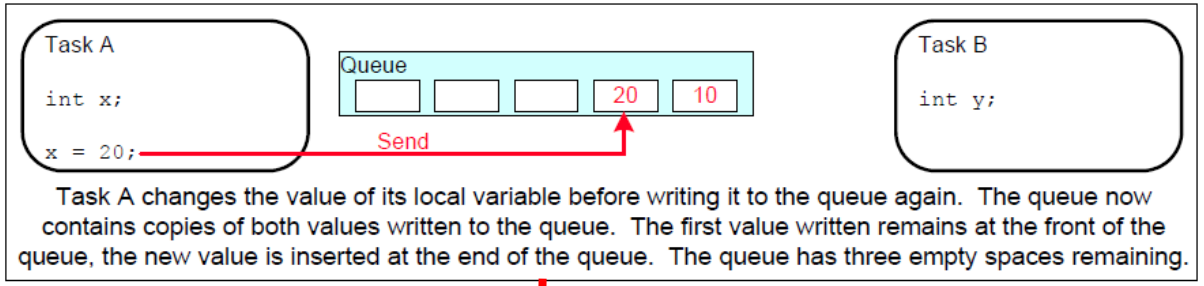
# Example: FreeRTOS (ES-Lab)



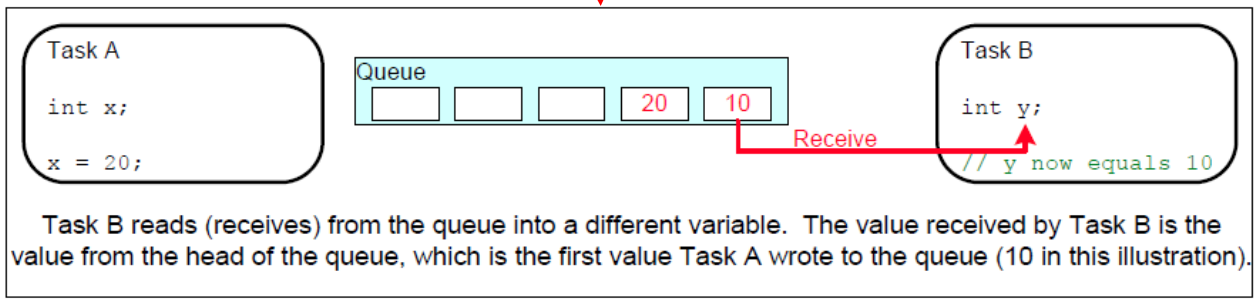

A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



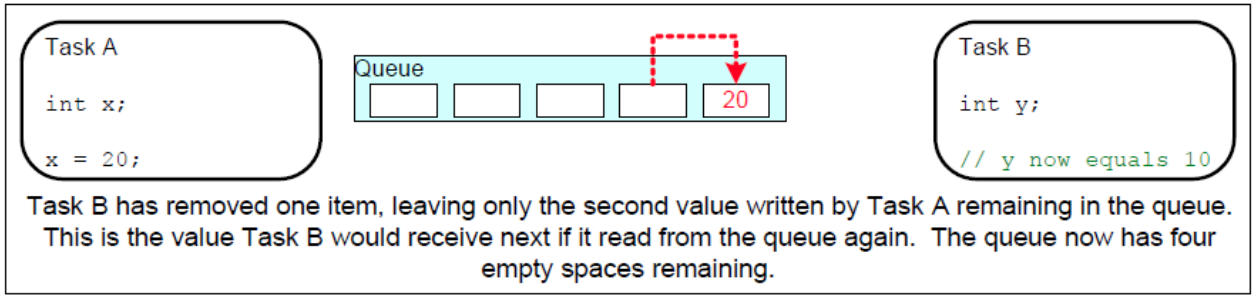

Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.




Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.




Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).




Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

# Example: FreeRTOS (ES-Lab)

*Creating a queue:*

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

returns handle to created queue

the maximum number of items that the queue being created can hold at any one time

the size in bytes of each data item

*Sending item to a queue:*

```
BaseType_t xQueueSend( QueueHandle_t xQueue,
                       const void * pvItemToQueue,
                       TickType_t xTicksToWait );
```

a pointer to the data to be copied into the queue

returns pdPASS if item was successfully added to queue

the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue

# Example: FreeRTOS (ES-Lab)

*Receiving item from a queue:*

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,
                          void * const pvBuffer,
                          TickType_t xTicksToWait );
```

returns pdPASS if data was successfully read from the queue

the maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue

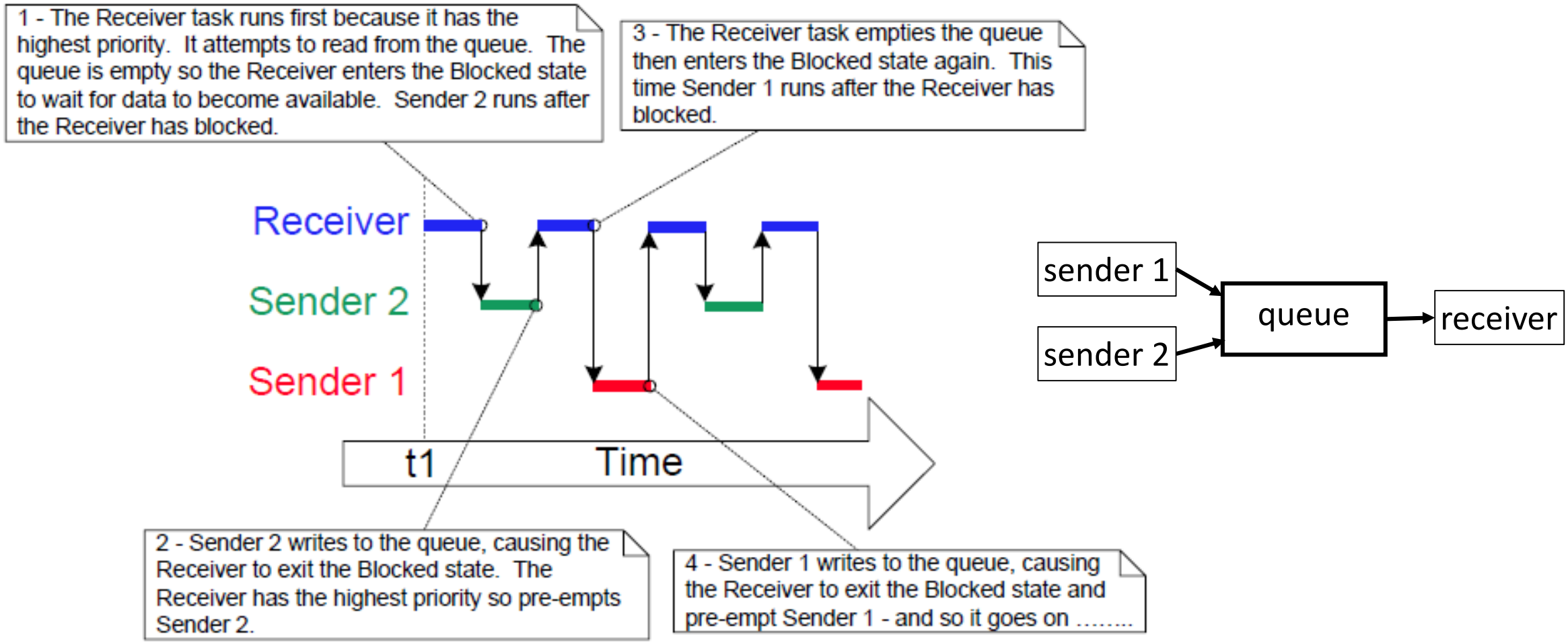
a pointer to the memory into which the received data will be copied

***Example:***

- Two sending tasks with equal priority 1 and one receiving task with priority 2.
- FreeRTOS schedules tasks with equal priority in a round-robin manner: A blocked or preempted task is put to the end of the ready queue for its priority. The same holds for the currently running task at the expiration of the time slice.

# Example: FreeRTOS (ES-Lab)

*Example cont.:*



# Communication Mechanisms

*Cyclical Asynchronous Buffers (CAB):*

- *Non-blocking communication between tasks*.
- A reader gets the most recent message put into the CAB. A message is not consumed (that is, extracted) by a receiving process but is maintained until overwritten by a new message.
- As a consequence, once the first message has been put in a CAB, a task can never be blocked during a receive operation. Similarly, since a new message overwrites the old one, a sender can never be blocked.
- Several readers can simultaneously read a single message from the CAB.

**writing**

```
buf_pointer = reserve(cab_id);
<copy message in *buf_pointer>
putmes(buf_pointer, cab_id);
```

**reading**

```
mes_pointer = getmes(cab_id);
<use message>
unget(mes_pointer, cab_id);
```

# Embedded Systems

## 8. Hardware Components

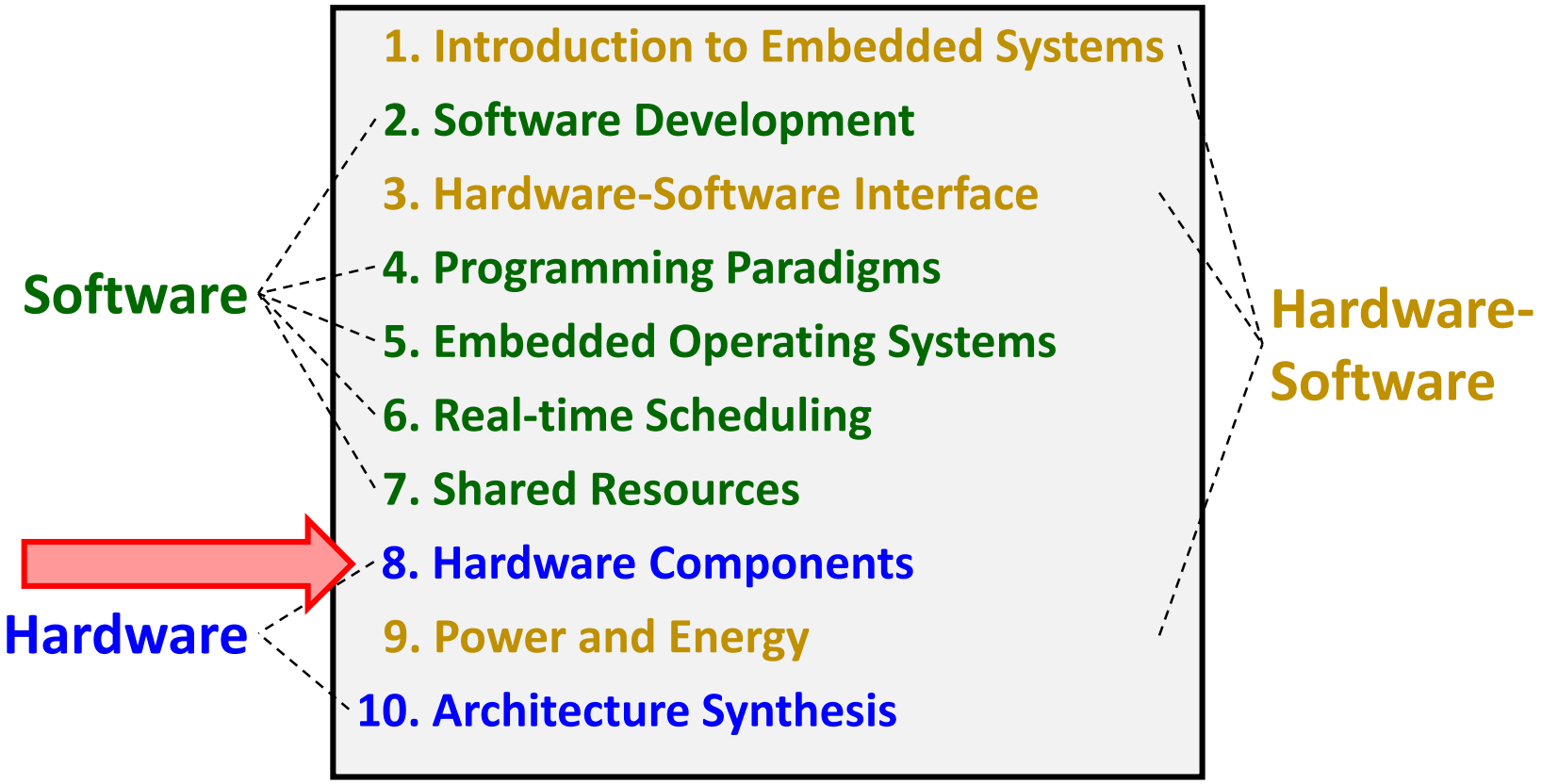
© Lothar Thiele

Computer Engineering and Networks Laboratory

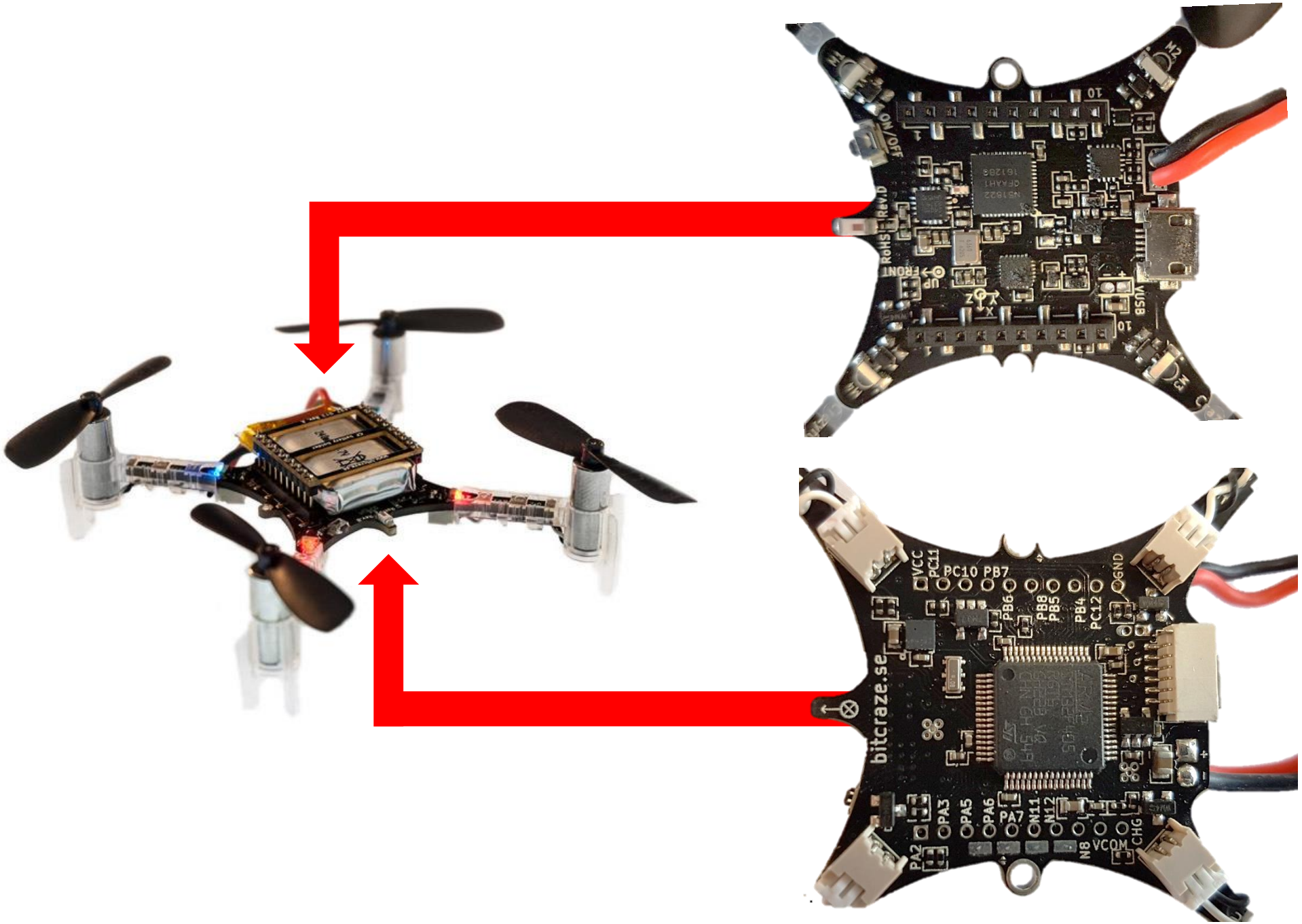
ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

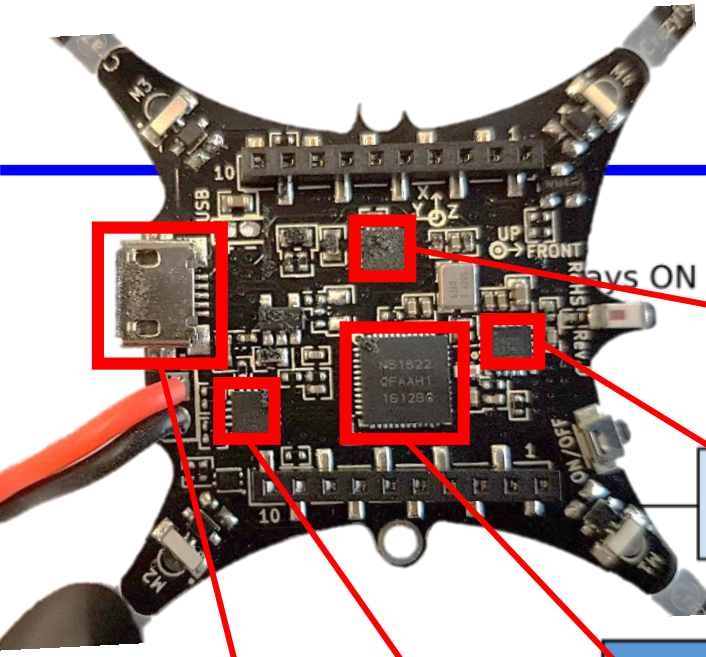
# Where we are …


1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis
Software
Hardware
Hardware-Software

# Do you Remember ?


ON/OFF
RoHS
REV.D
FRONT
USB
bitcraze.se
VCC
PC11
PC10
PB7
GND
PB6
PB8
PB5
PB4
PC12
PA2
PA3
PA5
PA6
PA7
CN11
N12
N8
VCOM
CHG

# High-Level Physical View



ays ON power domain

Power switched by nRF51 (VCC)

10DOF IMU
- 3-axis accelerometer
- 3-axis gyro
- 3-axis magnetomer
- Pressure sensor

I2C

RF power amplifier

Push button

nRF51822
- 16MHz Cortex-M0
- 16kB RAM, 256KB Flash
- BLE and NRF radio

UART

STM32F405
- 168MHz Cortex-M4
- 196kB RAM, 1MB Flash

PWM

Motor driver

I2C

SPI/I2C/GPIO/PWM

Wkup/OW/GPIO

Power supplies and battery charger

+5V

Charge/VBAT/VCC

Expansion port

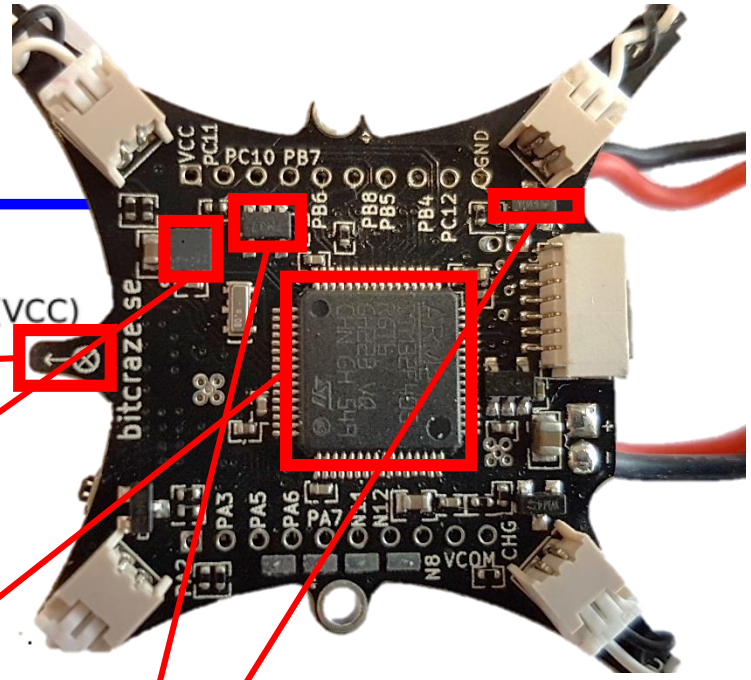
EEPROM

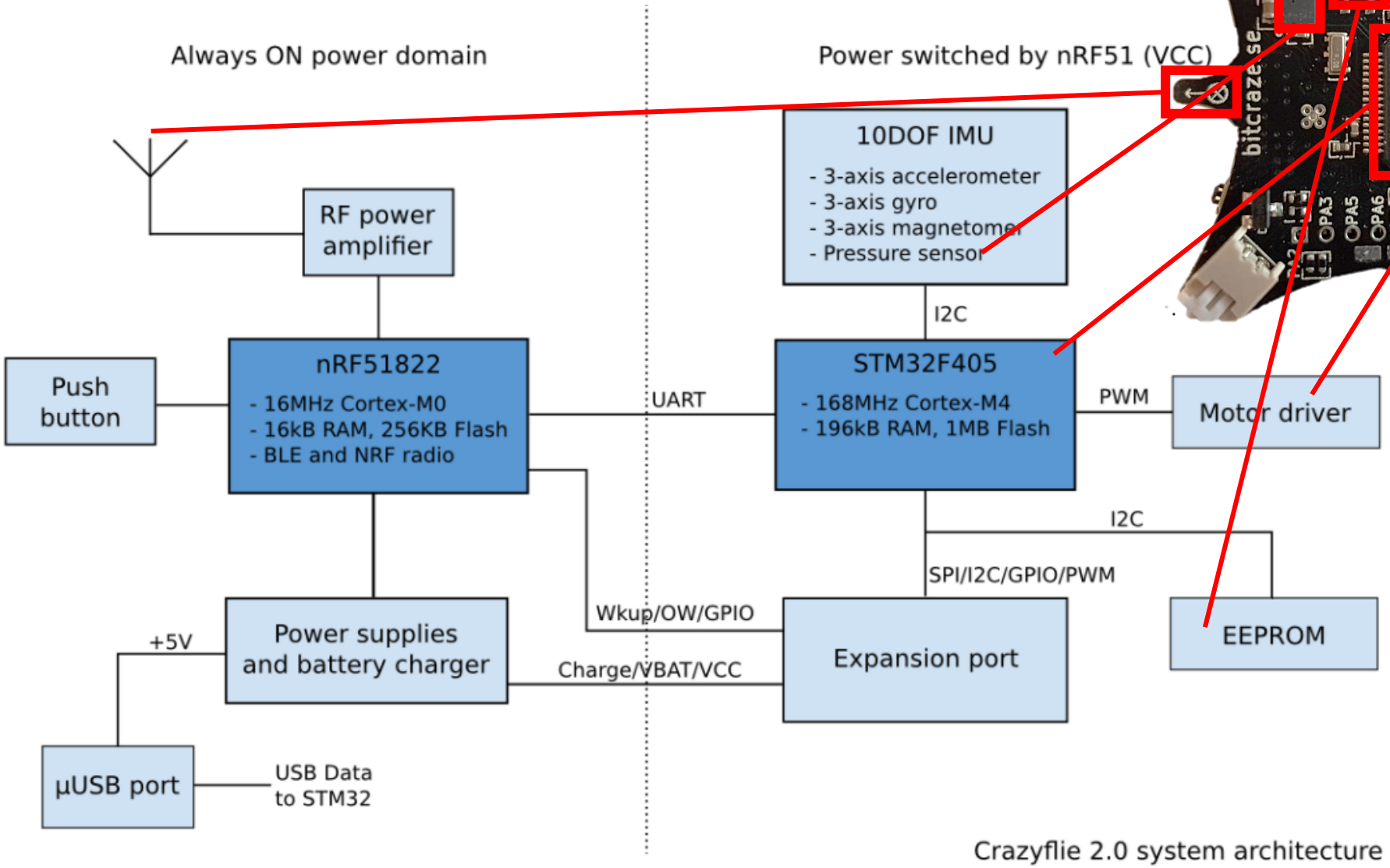
µUSB port

USB Data to STM32

Crazyflie 2.0 system architecture

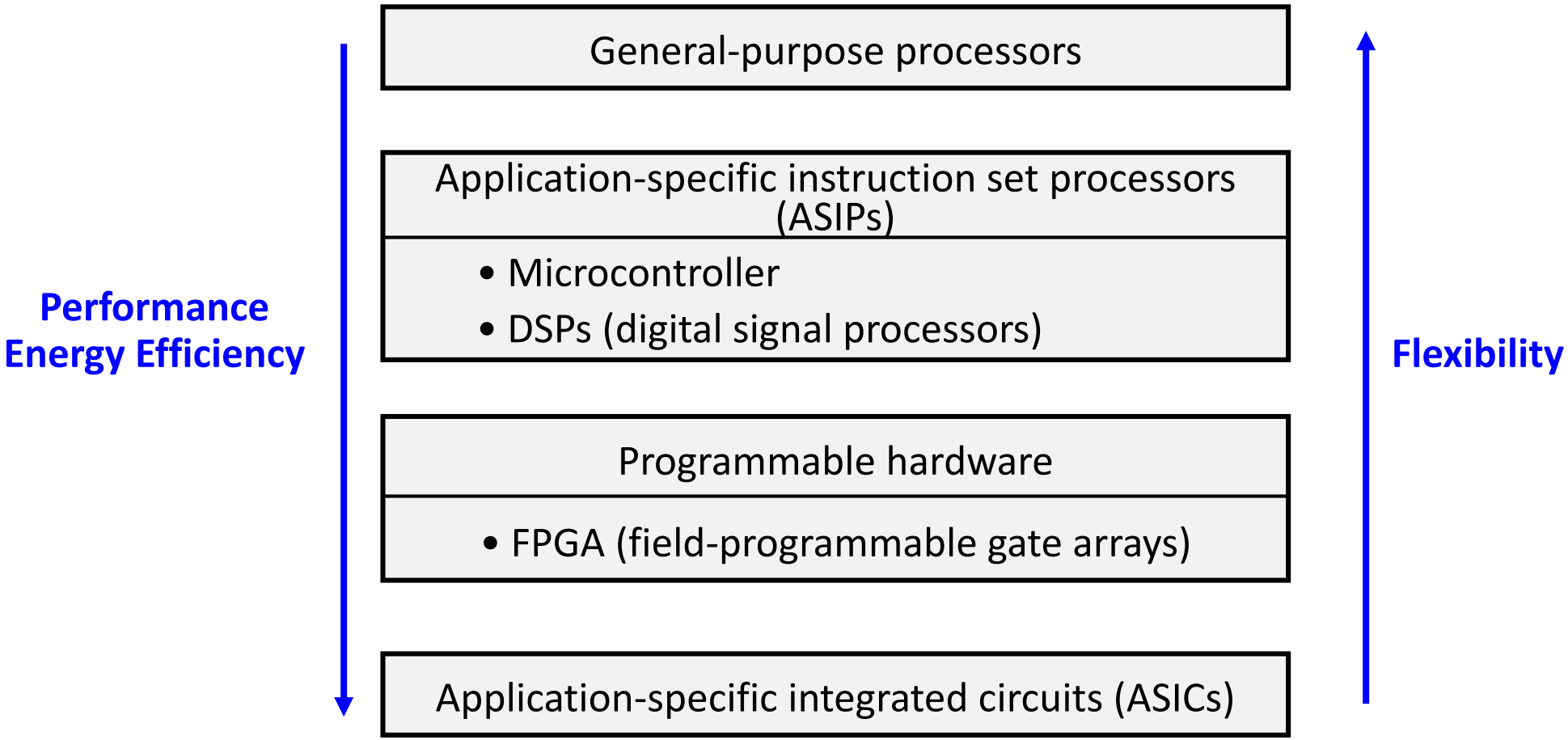
# High-Level Physical View



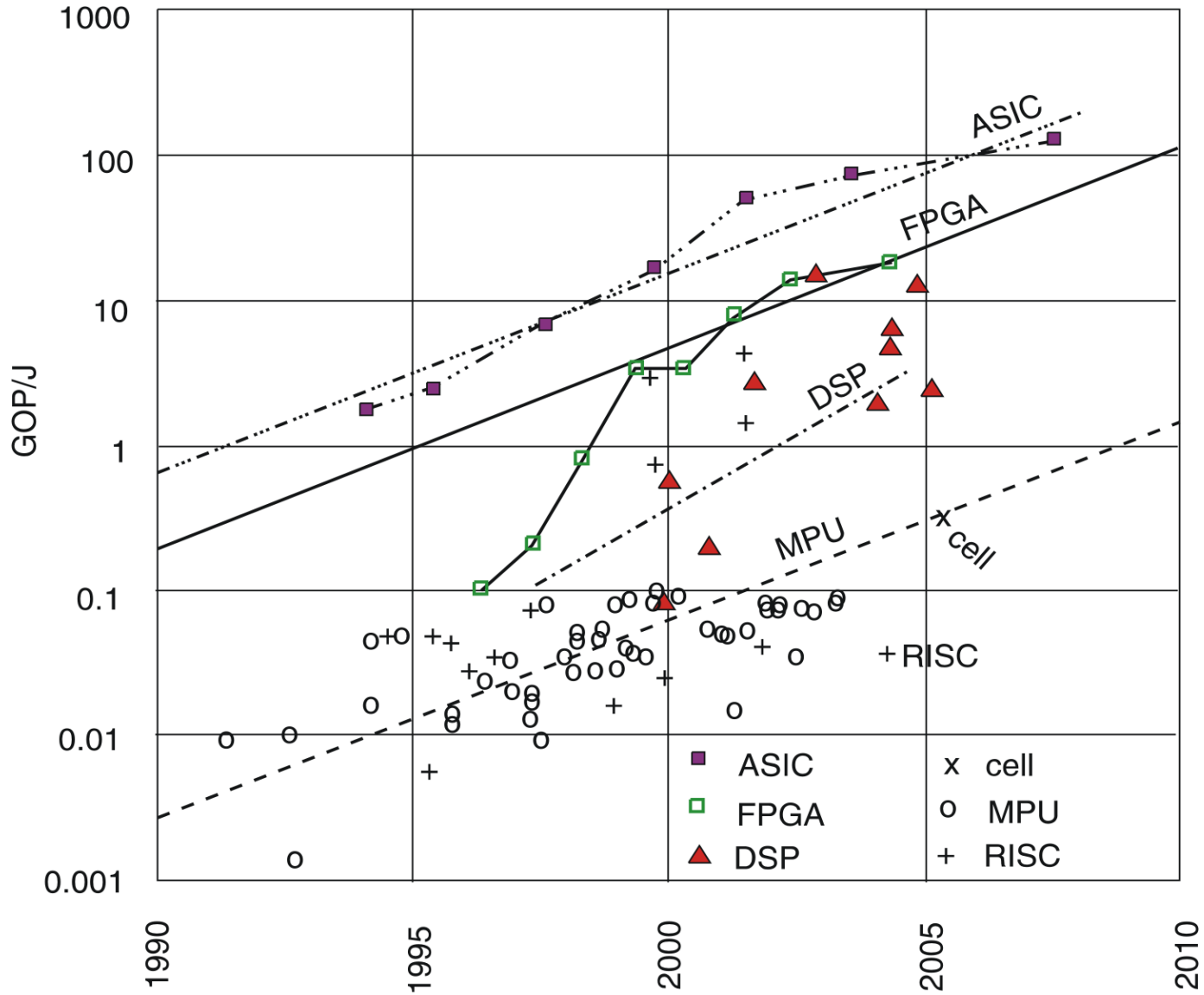




Crazyflie 2.0 system architecture

# Implementation Alternatives



# Energy Efficiency



# Topics

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

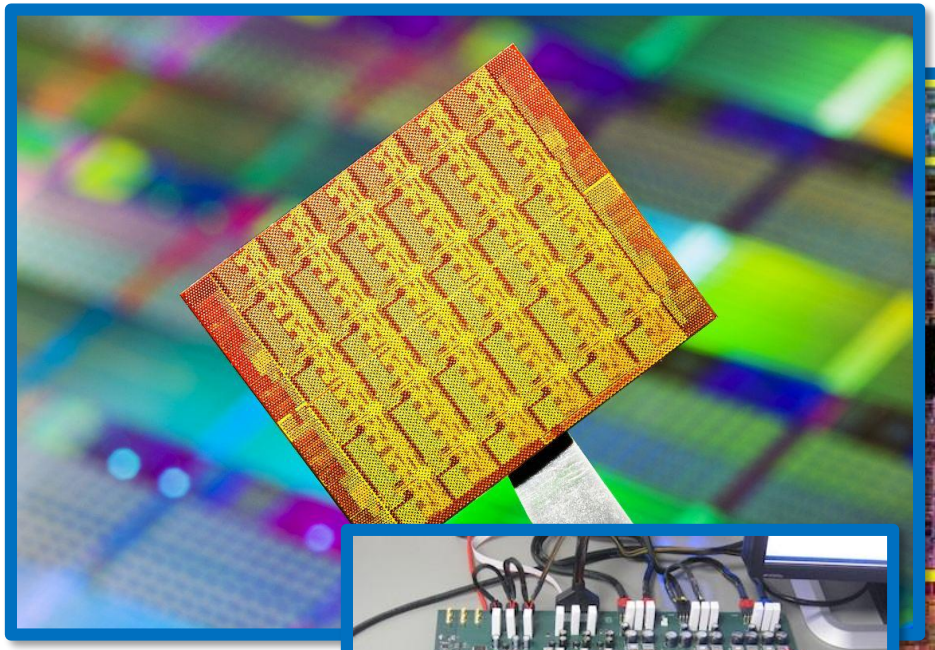
# General-Purpose Processors

- *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

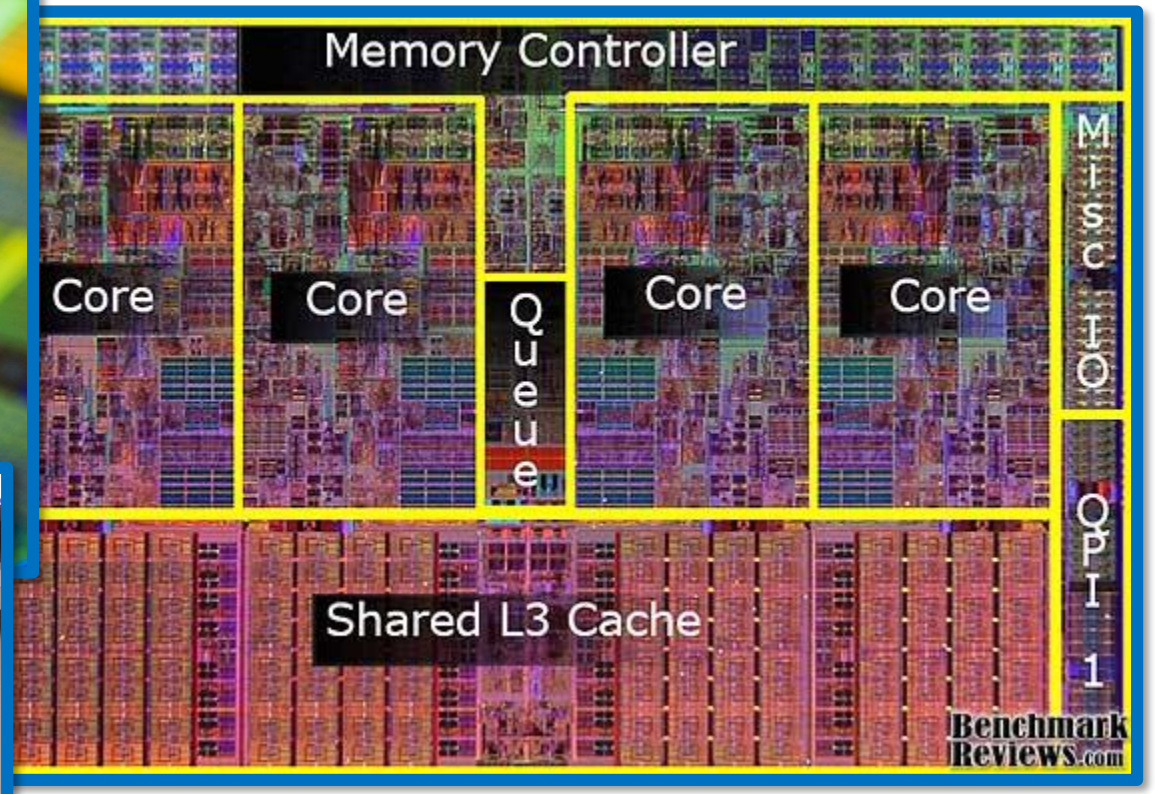
# General-Purpose Processors

- *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

# Multicore Examples

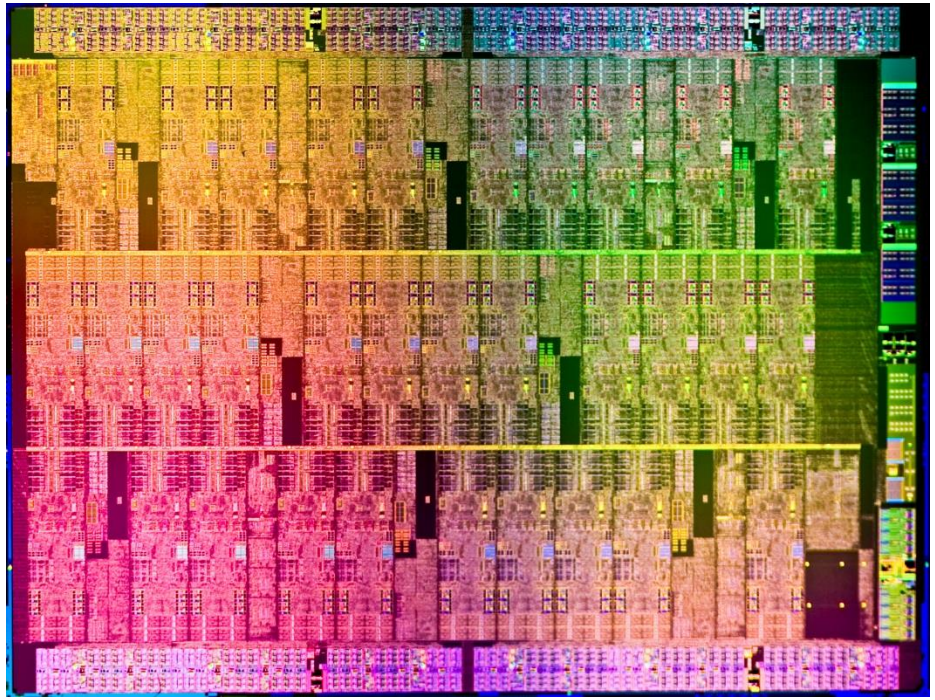


48 cores

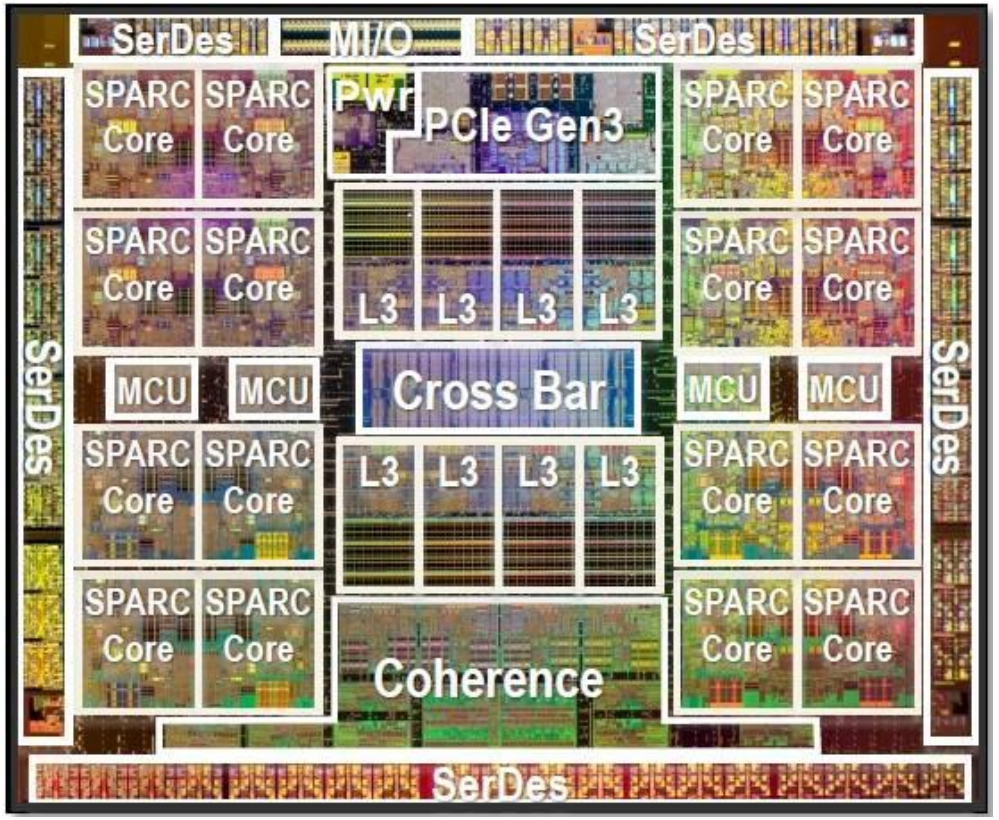



4 cores

# Multicore Examples

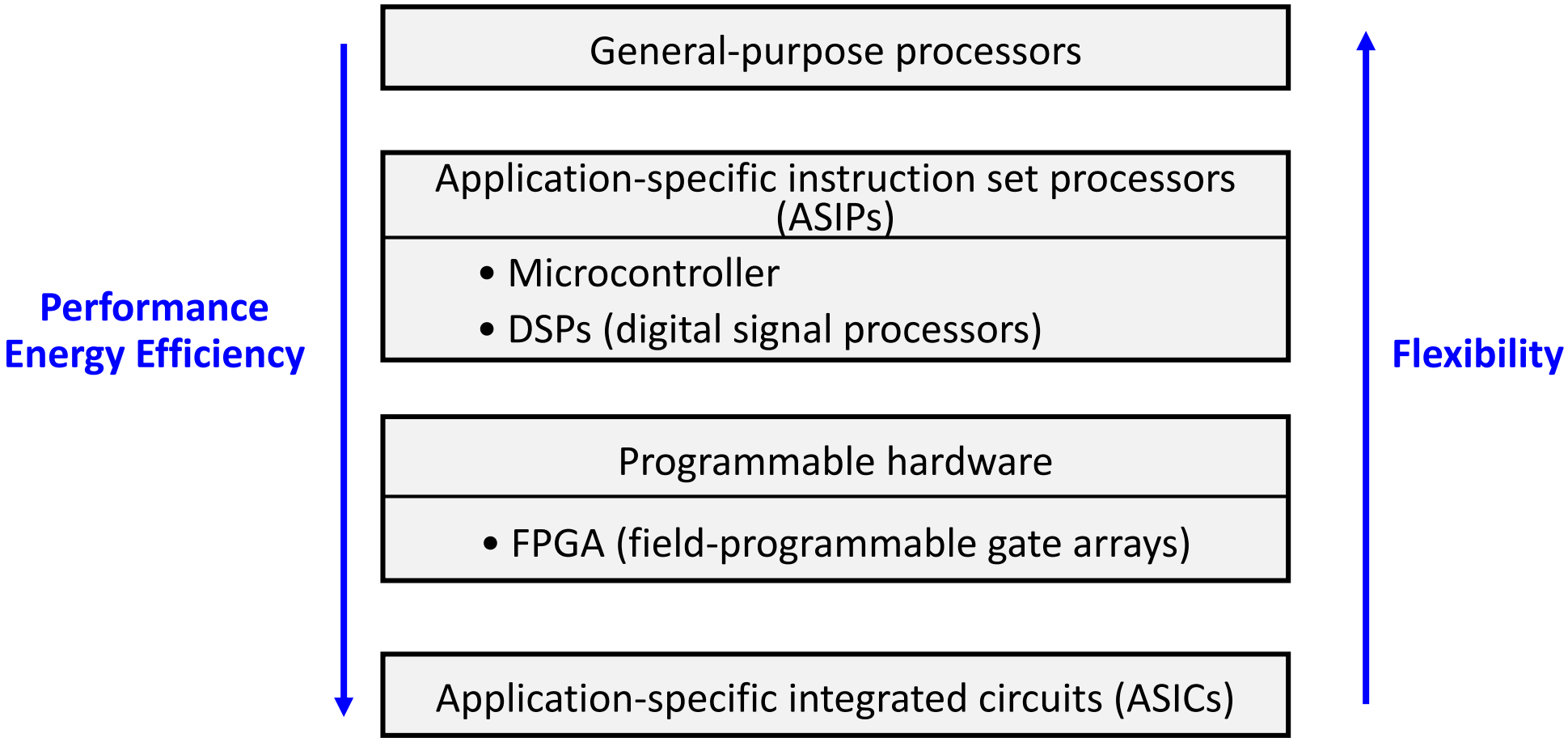


Intel Xeon Phi
(5 Billion transistors,
22nm technology,
350mm$^2$ area)




Oracle Sparc T5

# Implementation Alternatives



# Topics

- General Purpose Processors
- **System Specialization**
- Application Specific Instruction Sets
  - Micro Controller
  - Digital Signal Processors and VLIW
- Programmable Hardware
- ASICs
- Heterogeneous Architectures

# 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

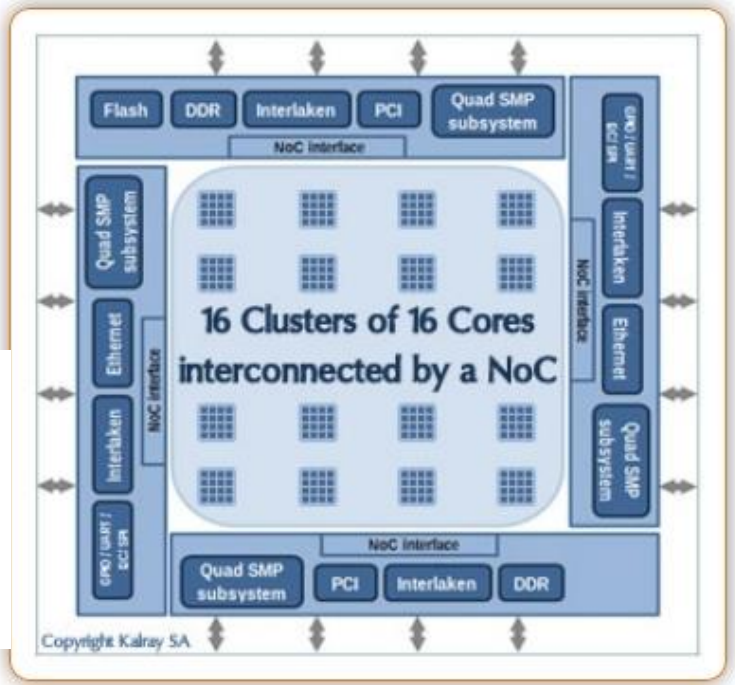
# Embedded Multicore Example

*Recent development:*

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

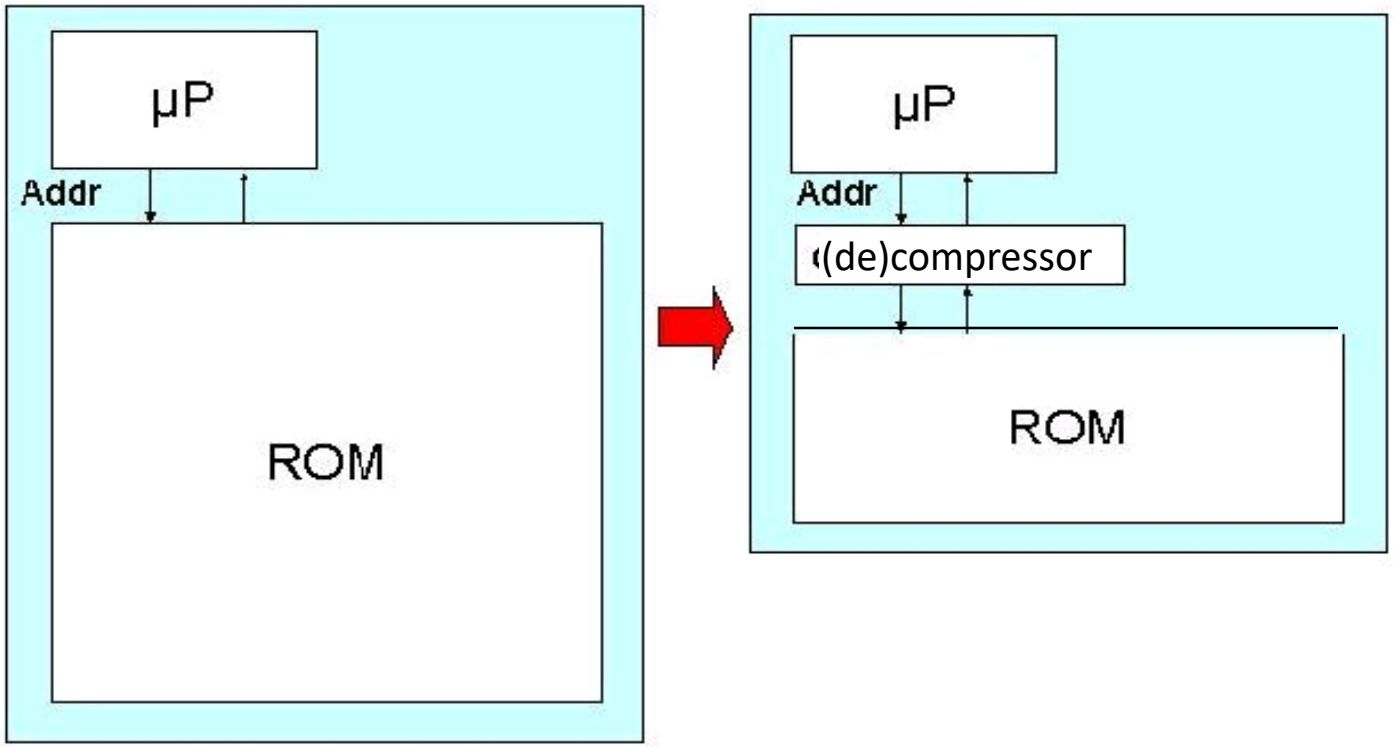


| Core Generation | Number of Processing Cores | GFLOPS/W | GOPS/W |
|---|---|---|---|
| Andey | 256 | 25 | 75 |
| Bostan (2014) | 256 | 50 | 80 |
| Coolidge (2015) | 64/256/1024 | 75 | 115 |



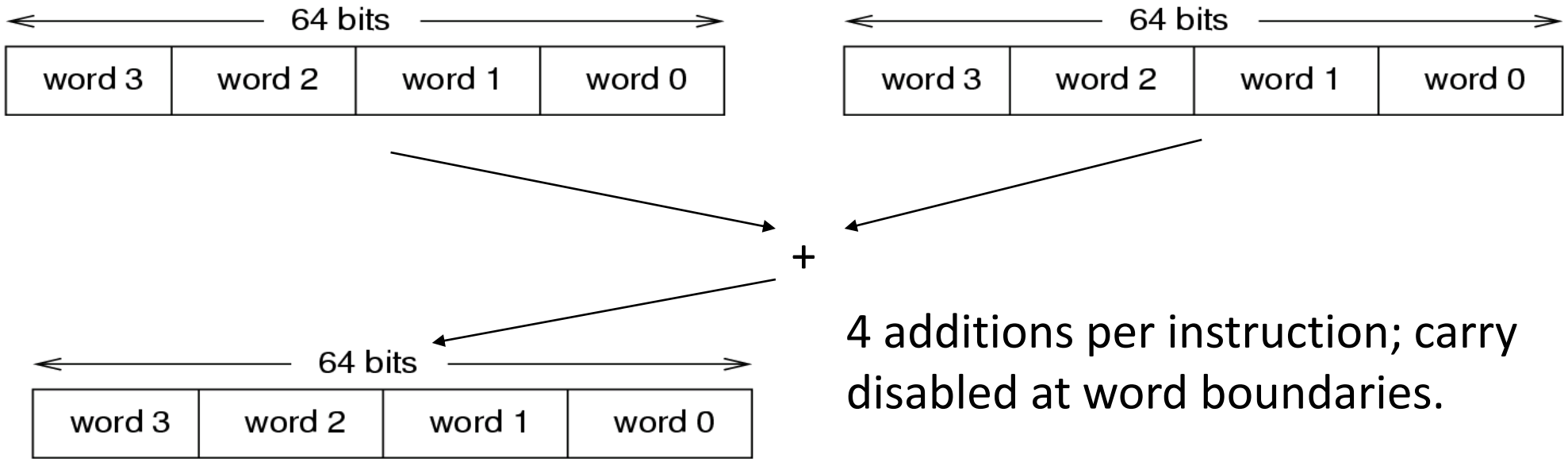
# Example: Code-size Efficiency

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



# Example: Multimedia-Instructions

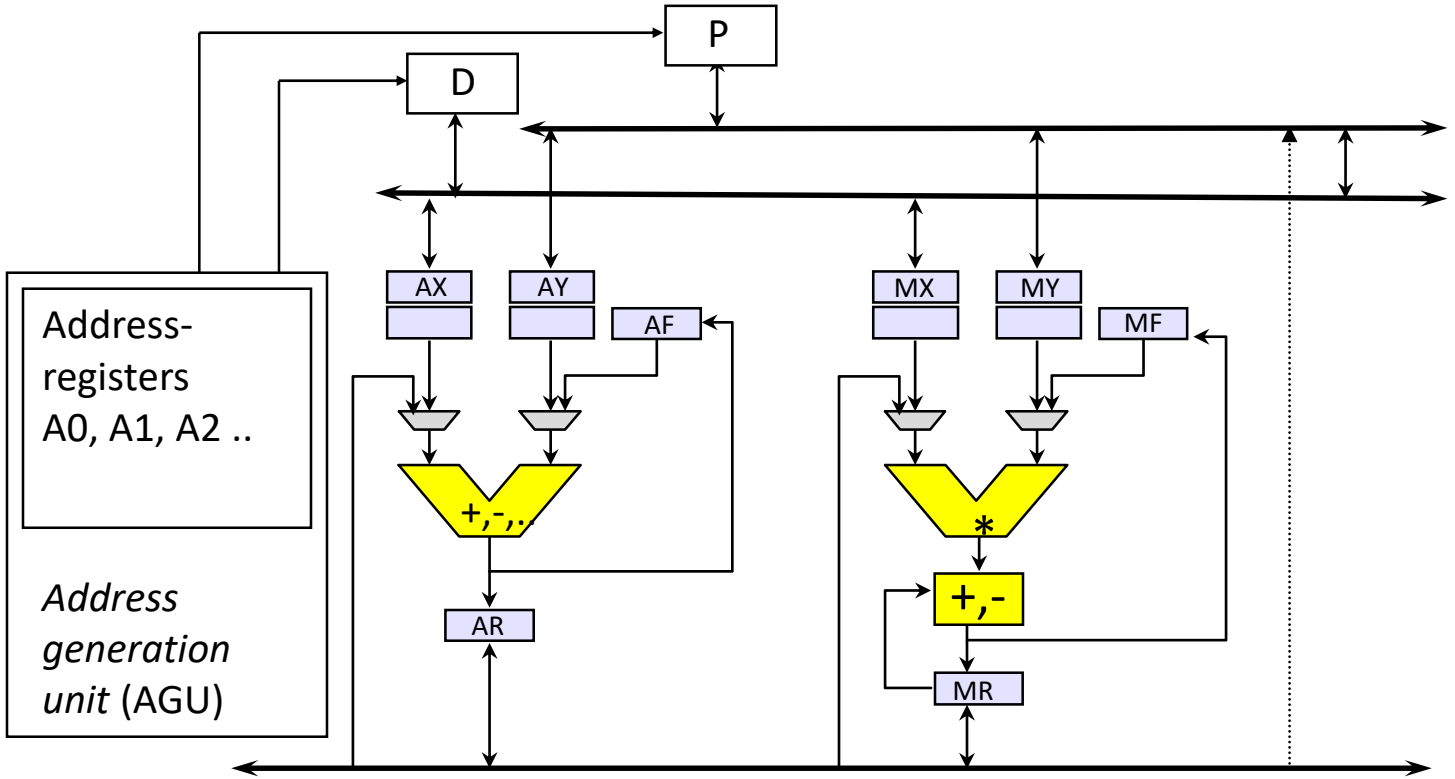
- Multimedia instructions exploit that many registers, adders etc. are quite wide (32/64 bit), whereas most multimedia data types are narrow (e.g. 8 bit per color, 16 bit per audio sample per channel).
- Idea: Several values can be stored per register and added in parallel.




4 additions per instruction; carry disabled at word boundaries.

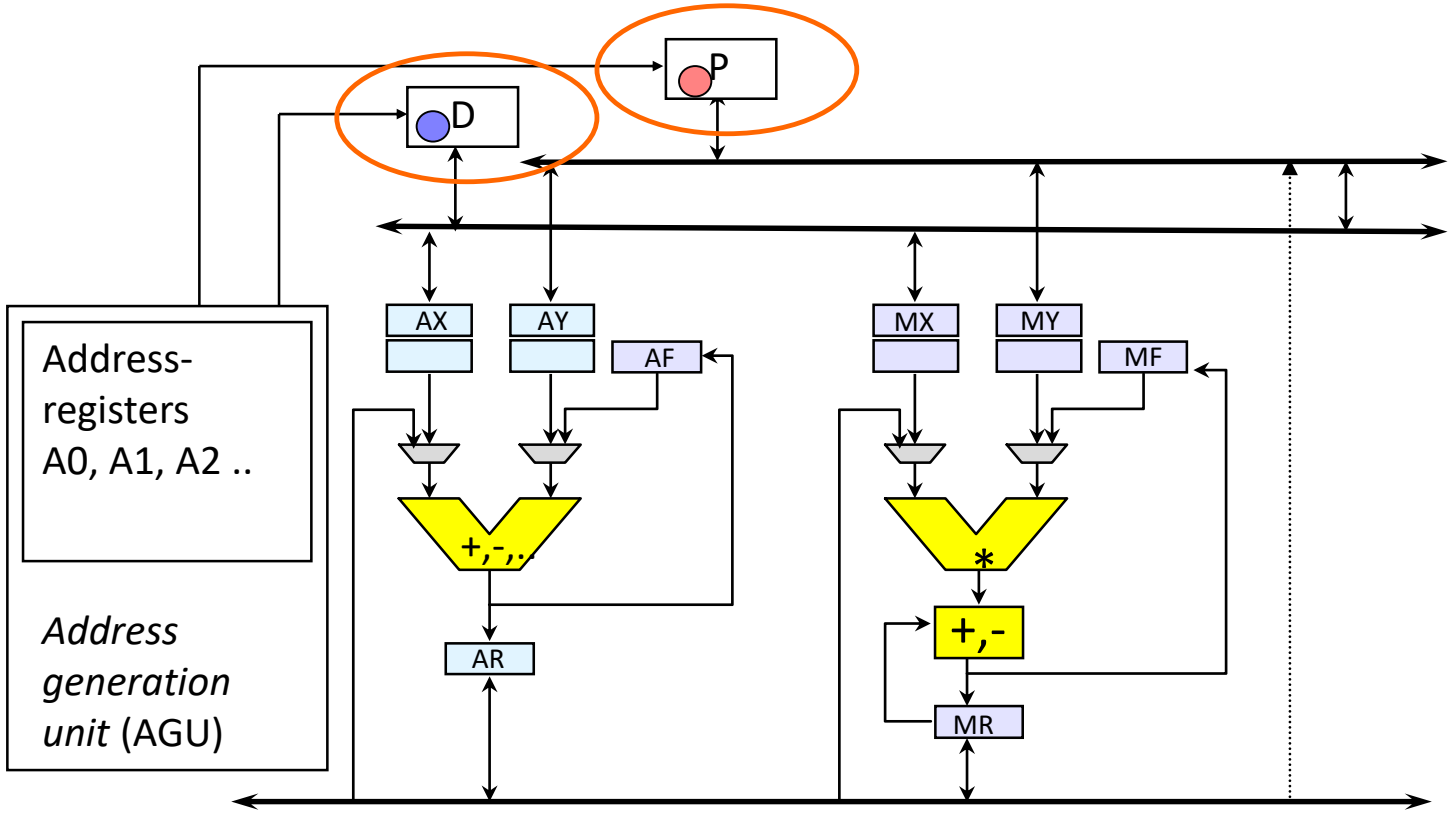
# Example: Heterogeneous Processor Registers

*Example (ADSP 210x):*




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

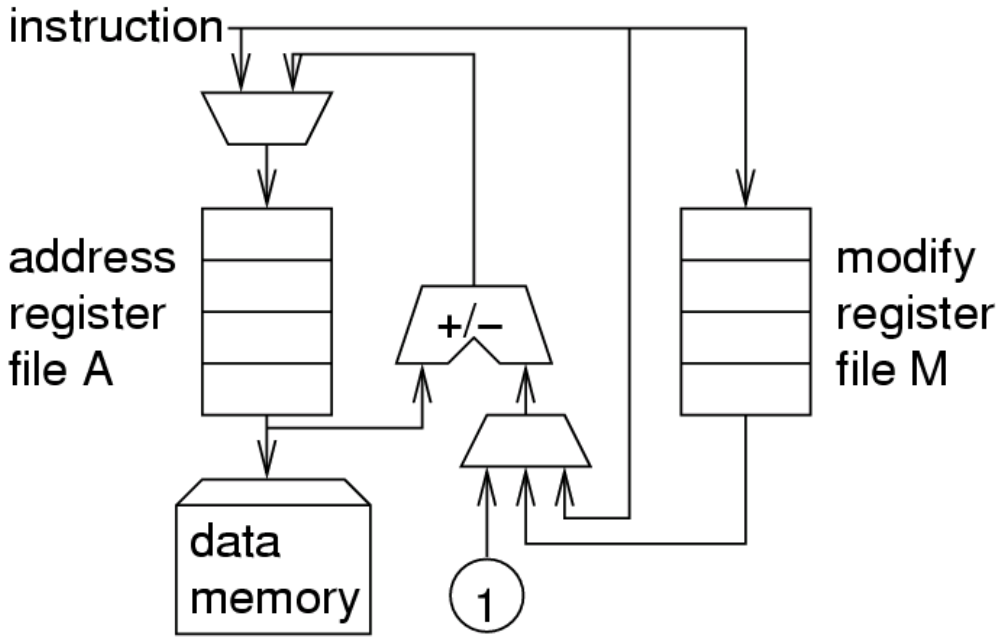
# Example: Multiple Memory Banks




Enables parallel fetches for some operations

# 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:

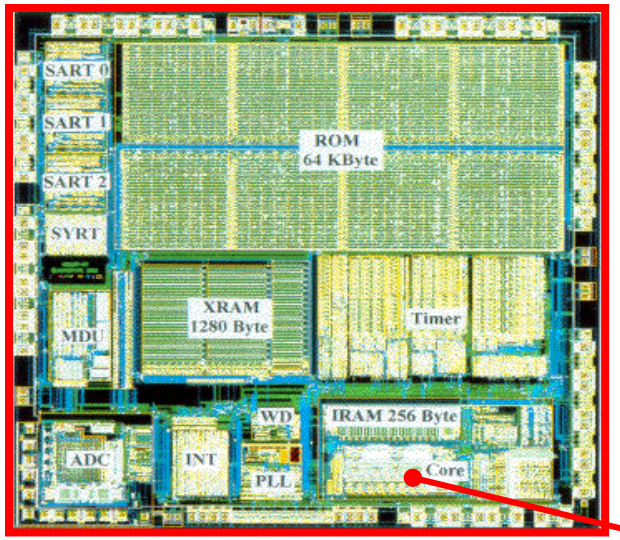
  M[j] := ‘immediate’
  A[i] := A[i] ± M[j]
  A[i] := A[i] ± 1
  A[i] := A[i] ± ‘immediate’
  A[i] := ‘immediate’

# Topics

- System Specialization
- **Application Specific Instruction Sets**
  - **Micro Controller**
  - Digital Signal Processors and VLIW
- Programmable Hardware
- ASICs
- Heterogeneous Architectures

# Microcontroller

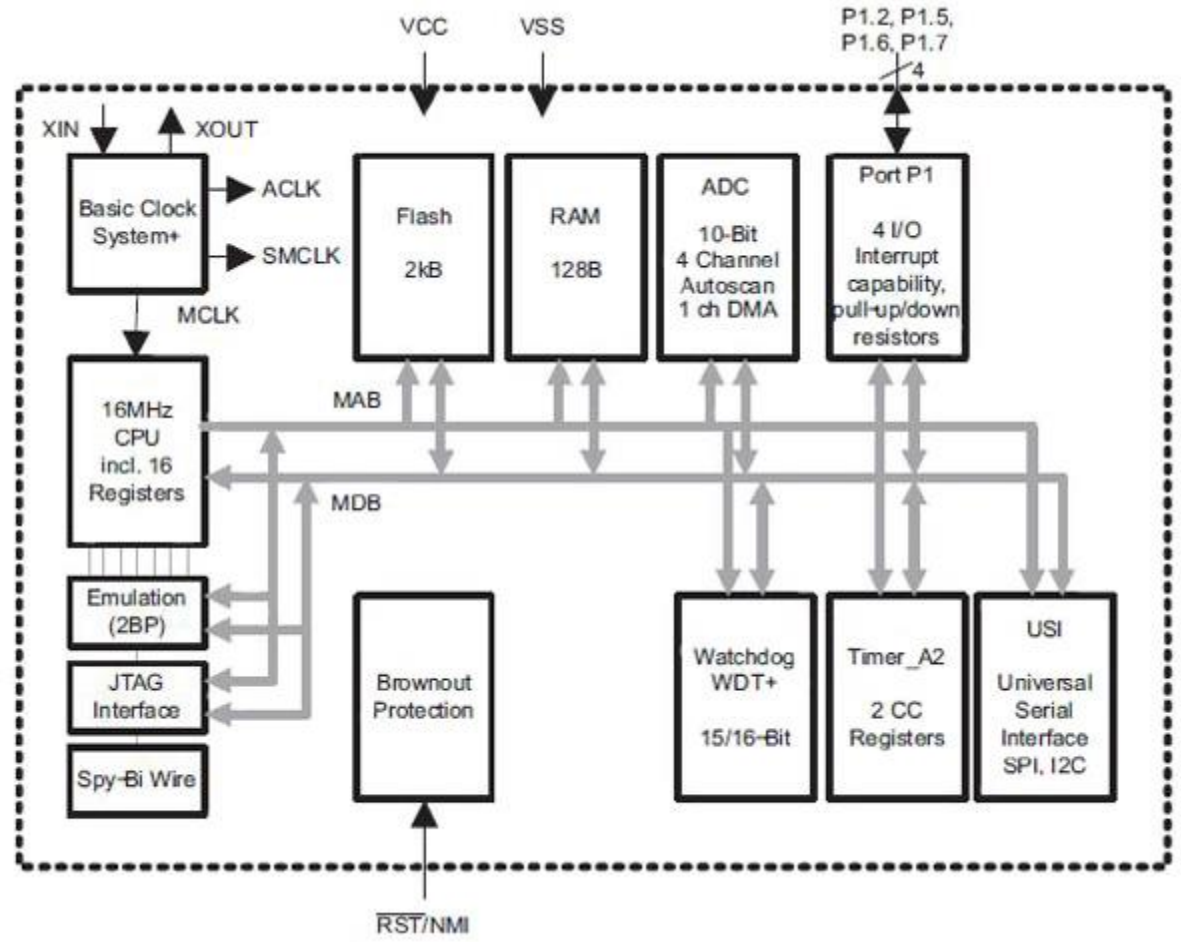
- *Control-dominant applications*
  - supports process scheduling and synchronization
  - preemption (interrupt), context switch
  - short latency times
- *Low power consumption*
- Peripheral units often integrated
- Suited for real-time applications




**8051 core**

SIECO51 (Siemens)

# Microcontroller as a System-on-Chip




**MSP 430 RISC Processor (Microchip)**

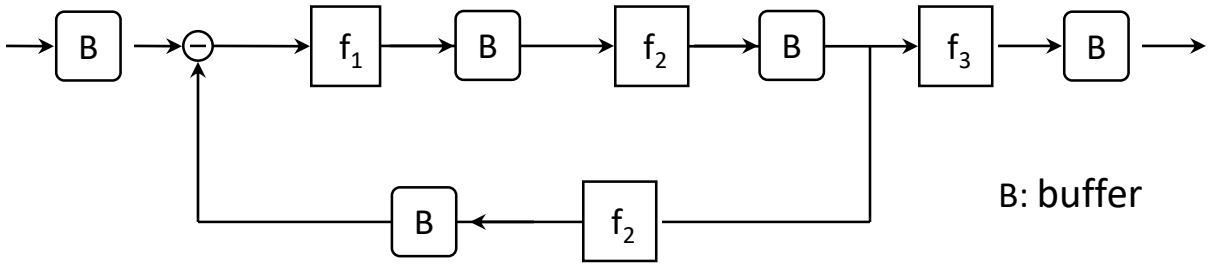
- complete system
- timers
- $I^2C$-bus and par./ser. interfaces for communi-cation
- A/D converter
- watchdog (SW activity timeout): safety
- on-chip memory (volatile/non-volatile)
- interrupt controller

# Topics

- System Specialization
- **Application Specific Instruction Sets**
  - Micro Controller
  - **Digital Signal Processors and VLIW**
- Programmable Hardware
- ASICs
- Heterogeneous Architectures

# Data Dominated Systems

- *Streaming oriented systems* with mostly periodic behavior
- Underlying *model of computation* is often a signal flow graph or data flow graph:



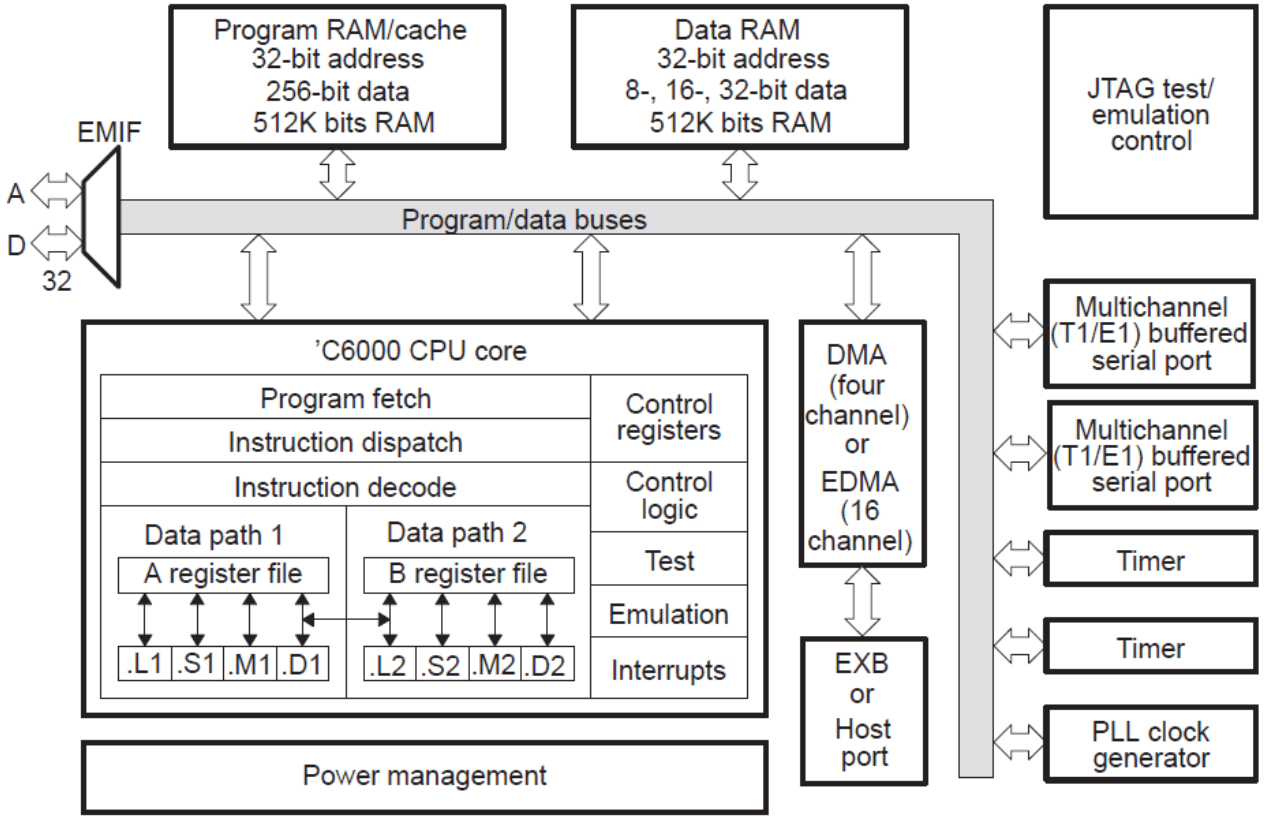

- Typical *application examples*:
  - signal processing
  - multimedia processing
  - automatic control

# 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*

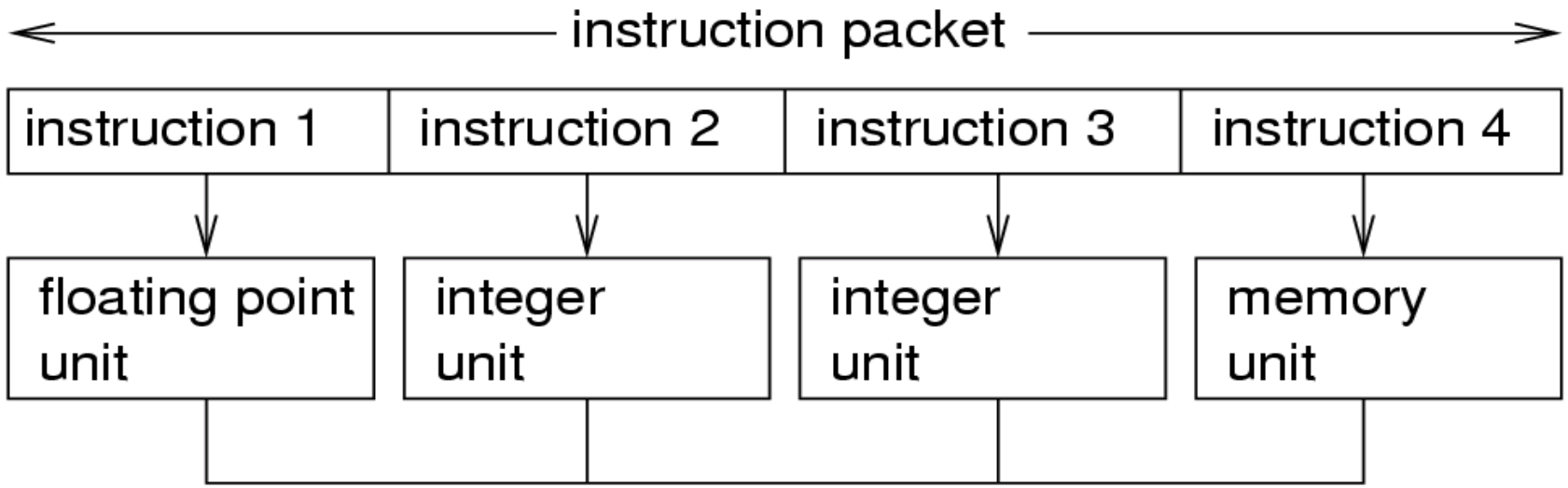
*Figure 2–1. TMS320C62x/C67x Block Diagram*



# Very Long Instruction Word (VLIW)

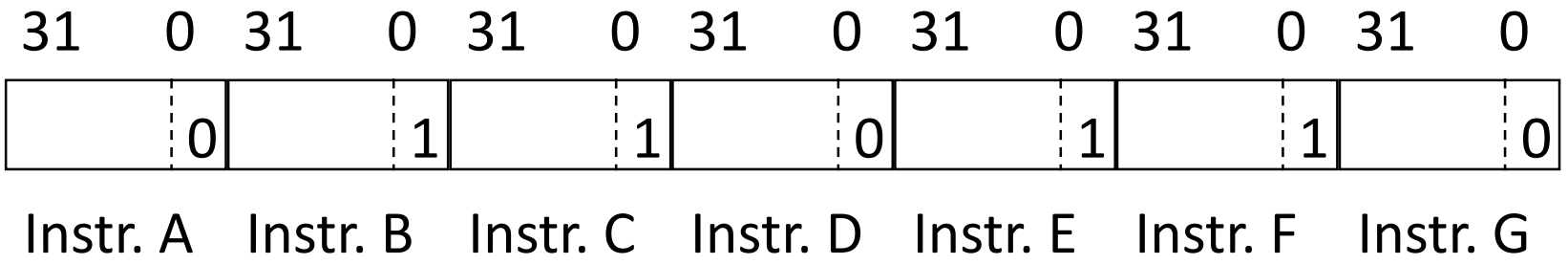
*Key idea:* detection of possible parallelism to be done by compiler, not by hardware at run-time (inefficient).

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



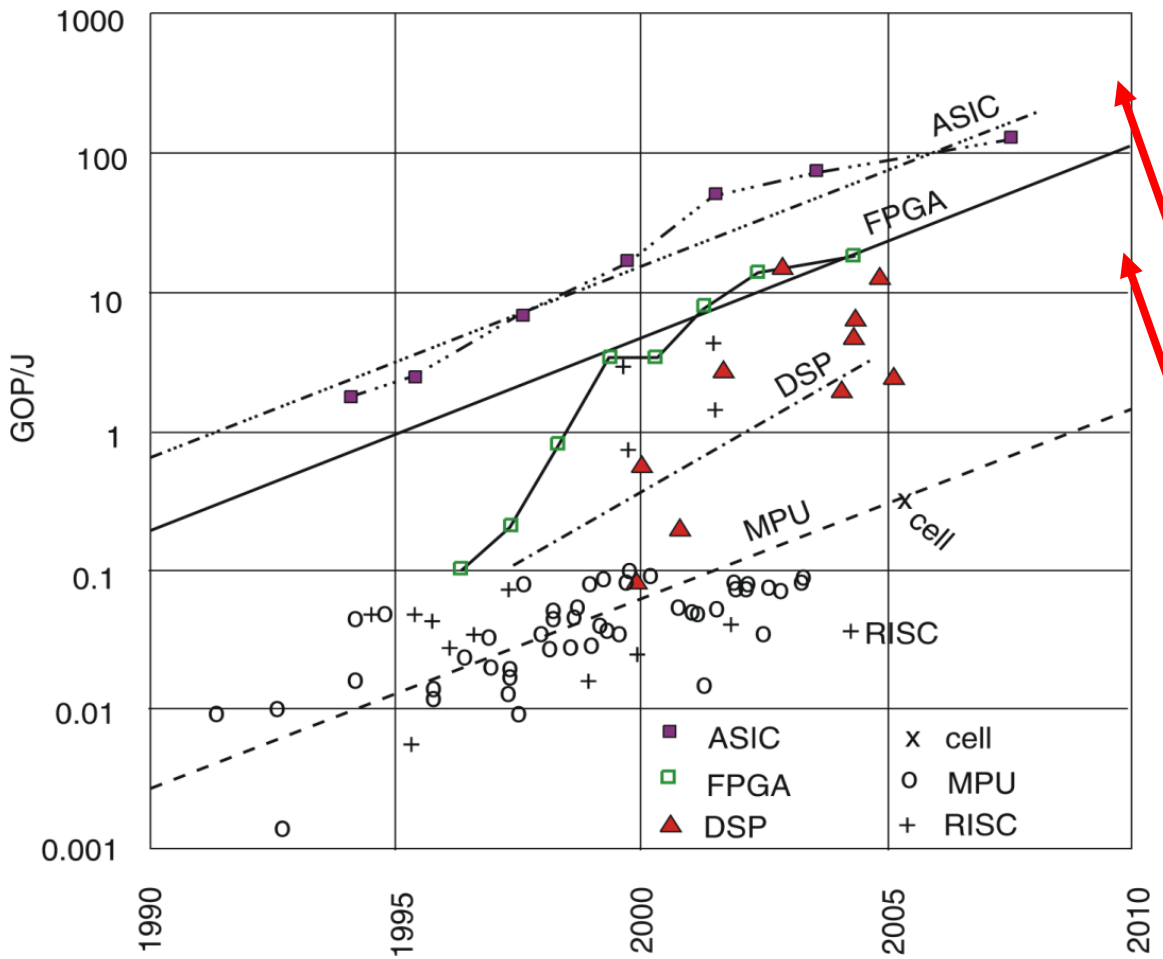
# Explicit Parallelism Instruction Computers (EPIC)

The TMS320C62xx VLIW Processor as an example of EPIC:

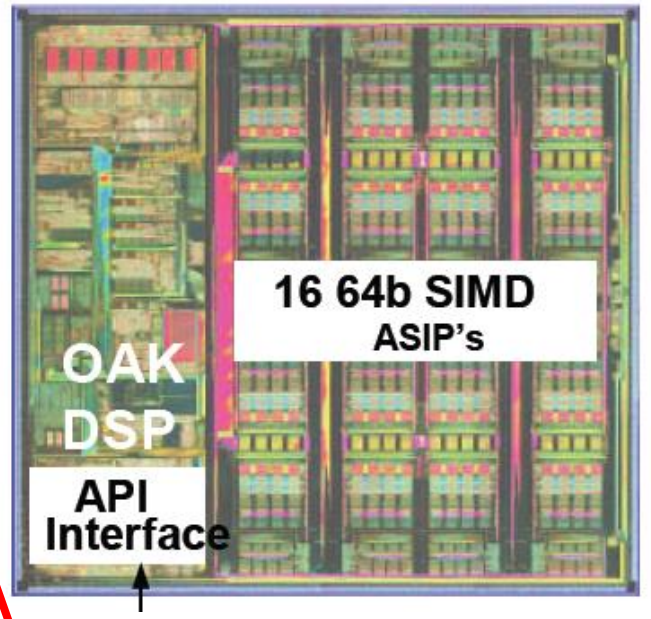



| Cycle | Instruction | | |
|---|---|---|---|
| 1 | A | | |
| 2 | B | C | D |
| 3 | E | F | G |

# Example Infineon

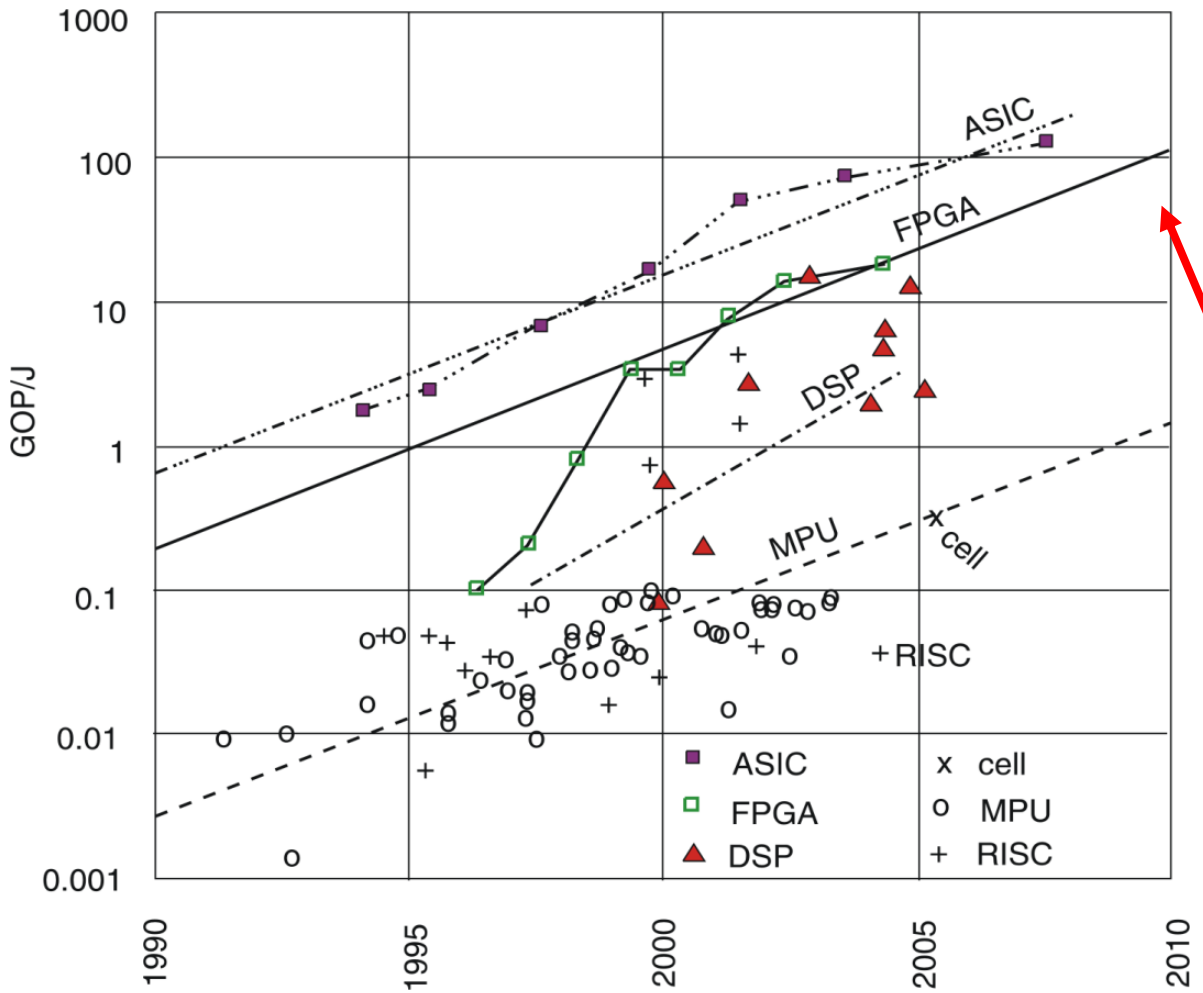



**Processor core for car mirrors**
**Infineon**



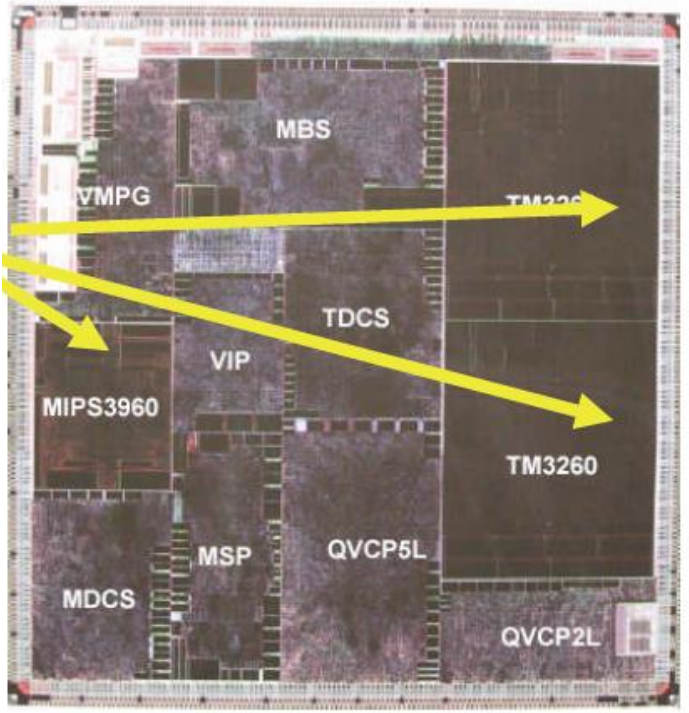

**200MHz , 0.76 Watt**
**100Gops @ 8b**
**25Gops @ 32b**

# Example NXP Trimedia VLIW




**Nexperia Digital Video Platform**
**NXP**

VLIW
MIPS



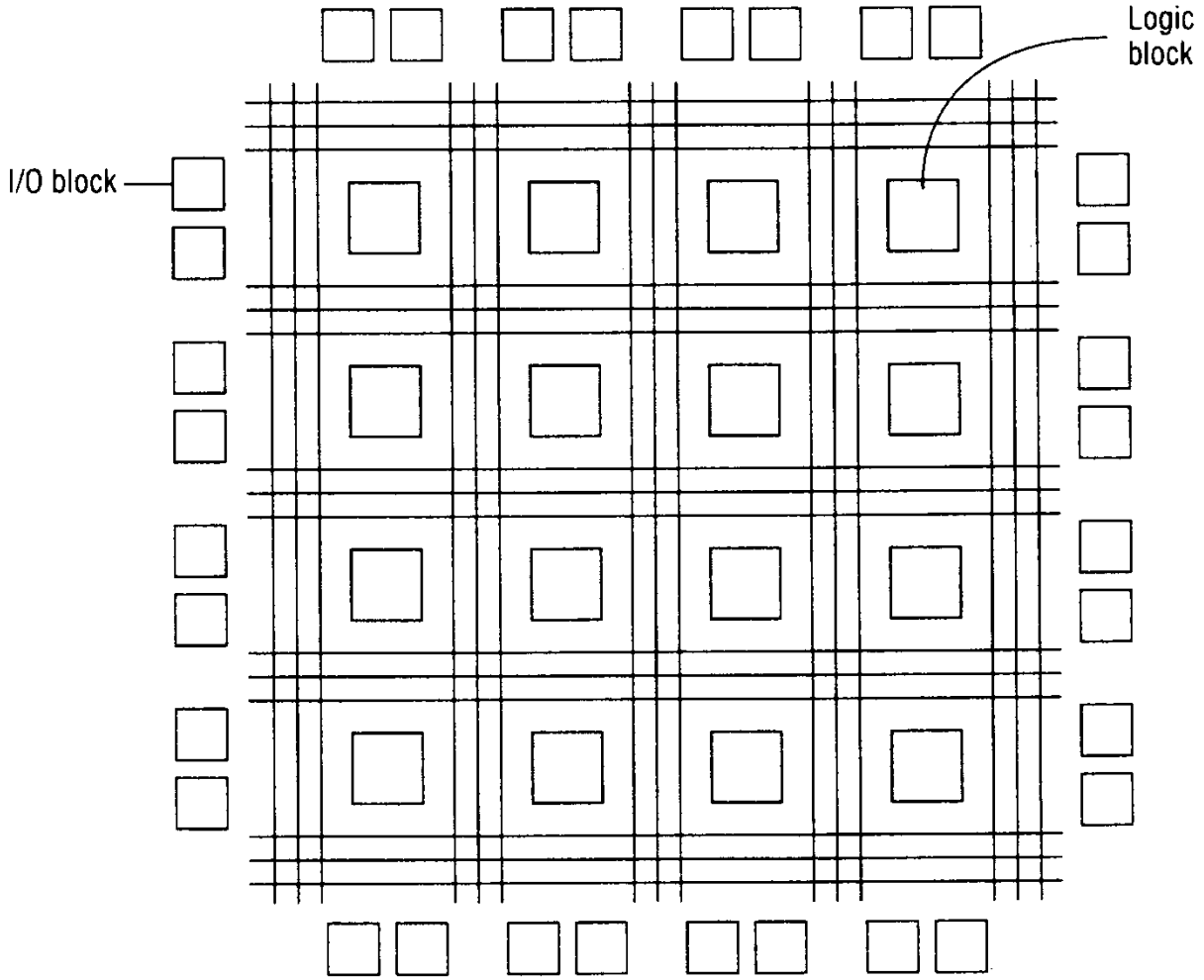

**1 MIPS, 2 Trimedia**
**60 coproc,**
**266MHz, 1.5 watt 100 Gops**

# Topics

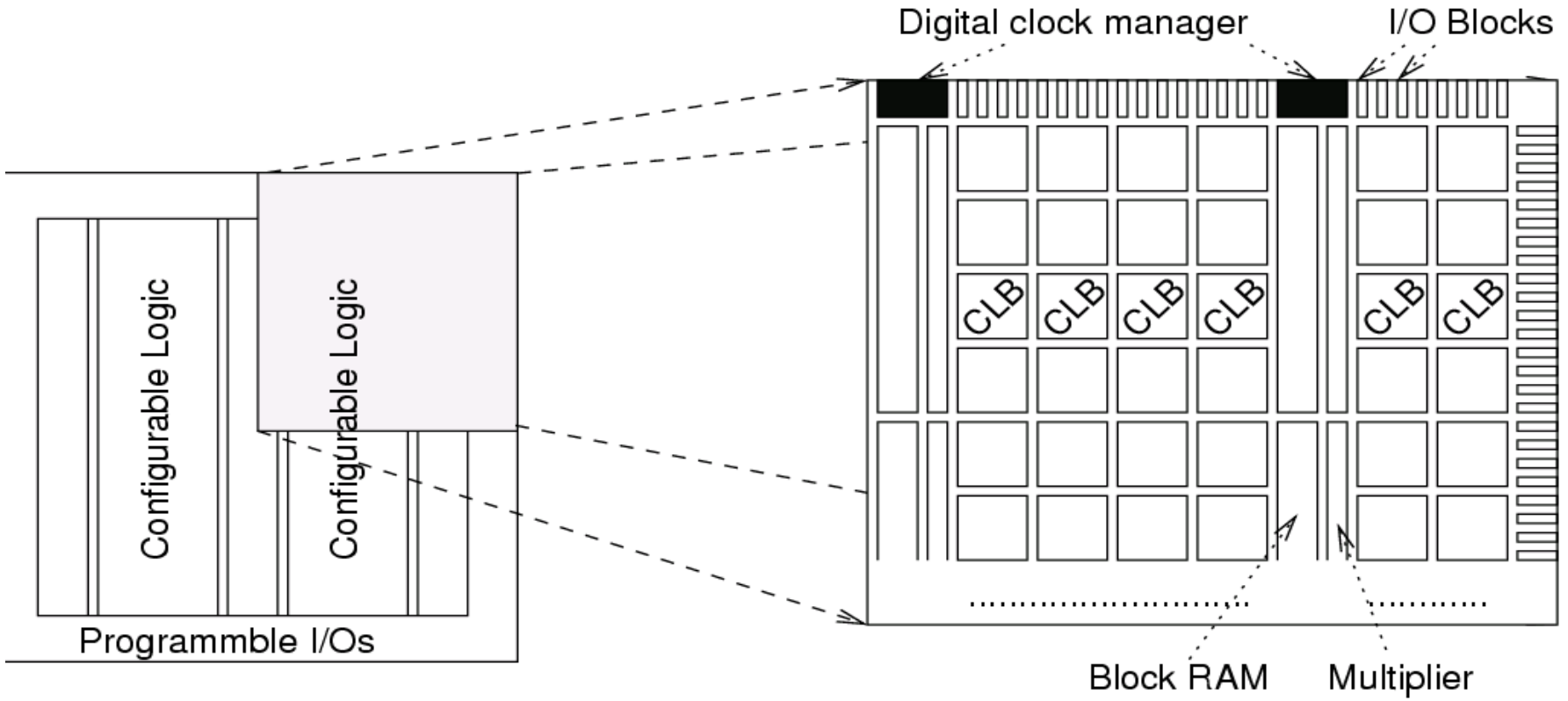
- System Specialization
- Application Specific Instruction Sets
  - Micro Controller
  - Digital Signal Processors and VLIW
- **Programmable Hardware**
- ASICs
- System-on-Chip

# FPGA – Basic Strucutre

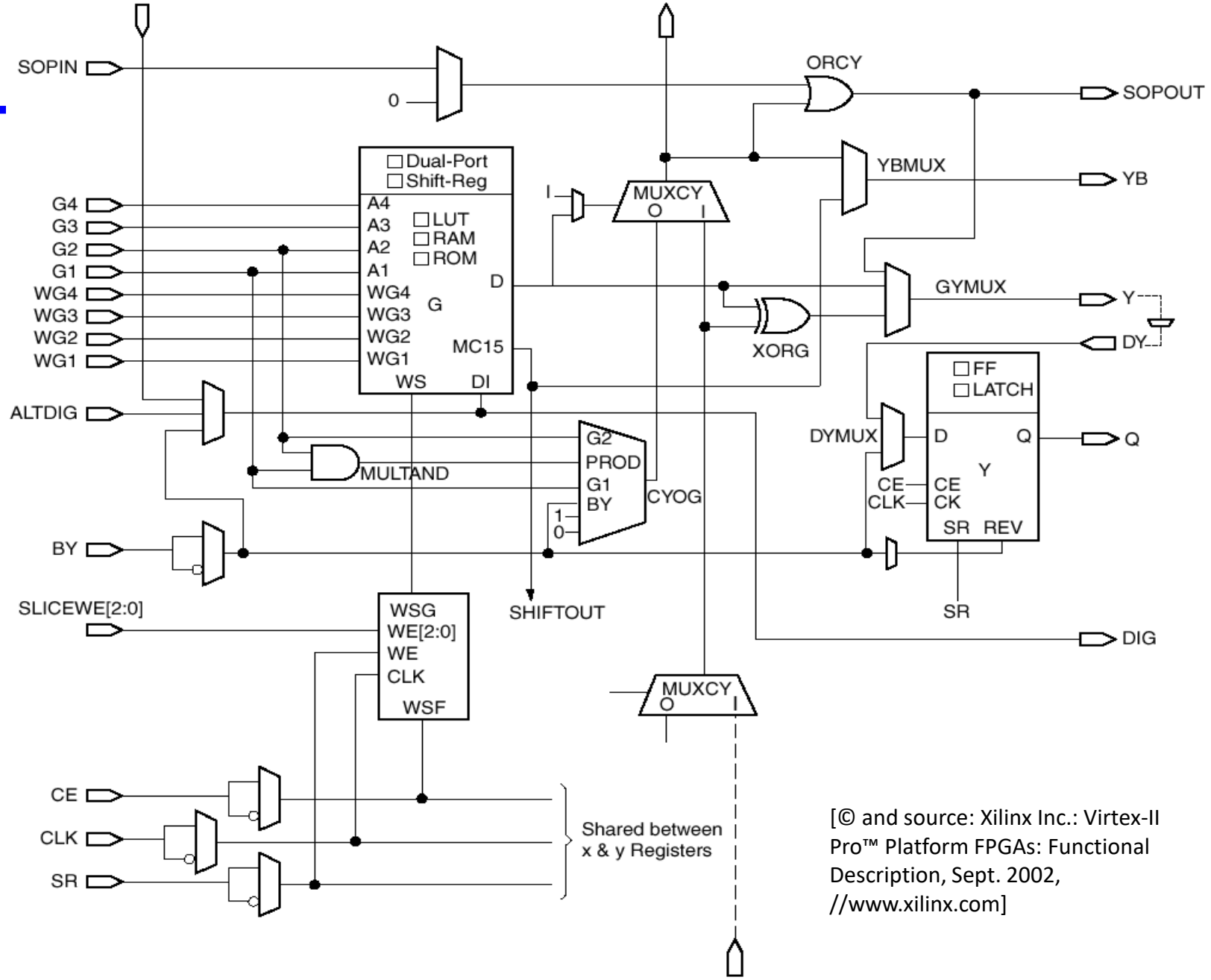
- Logic Units
- I/O Units
- Connections



# Floor-plan of VIRTEX II FPGAs



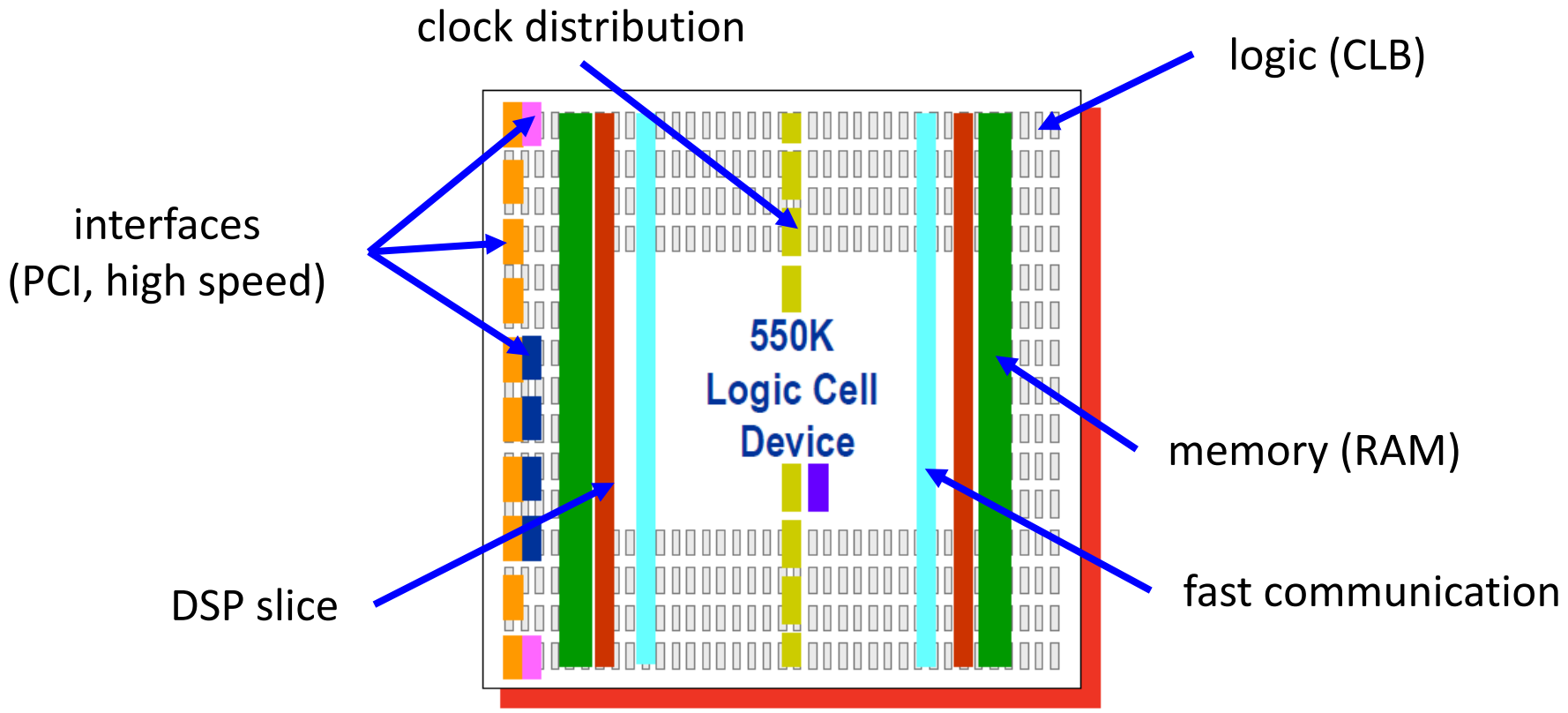
# Virtex Logic Cell




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

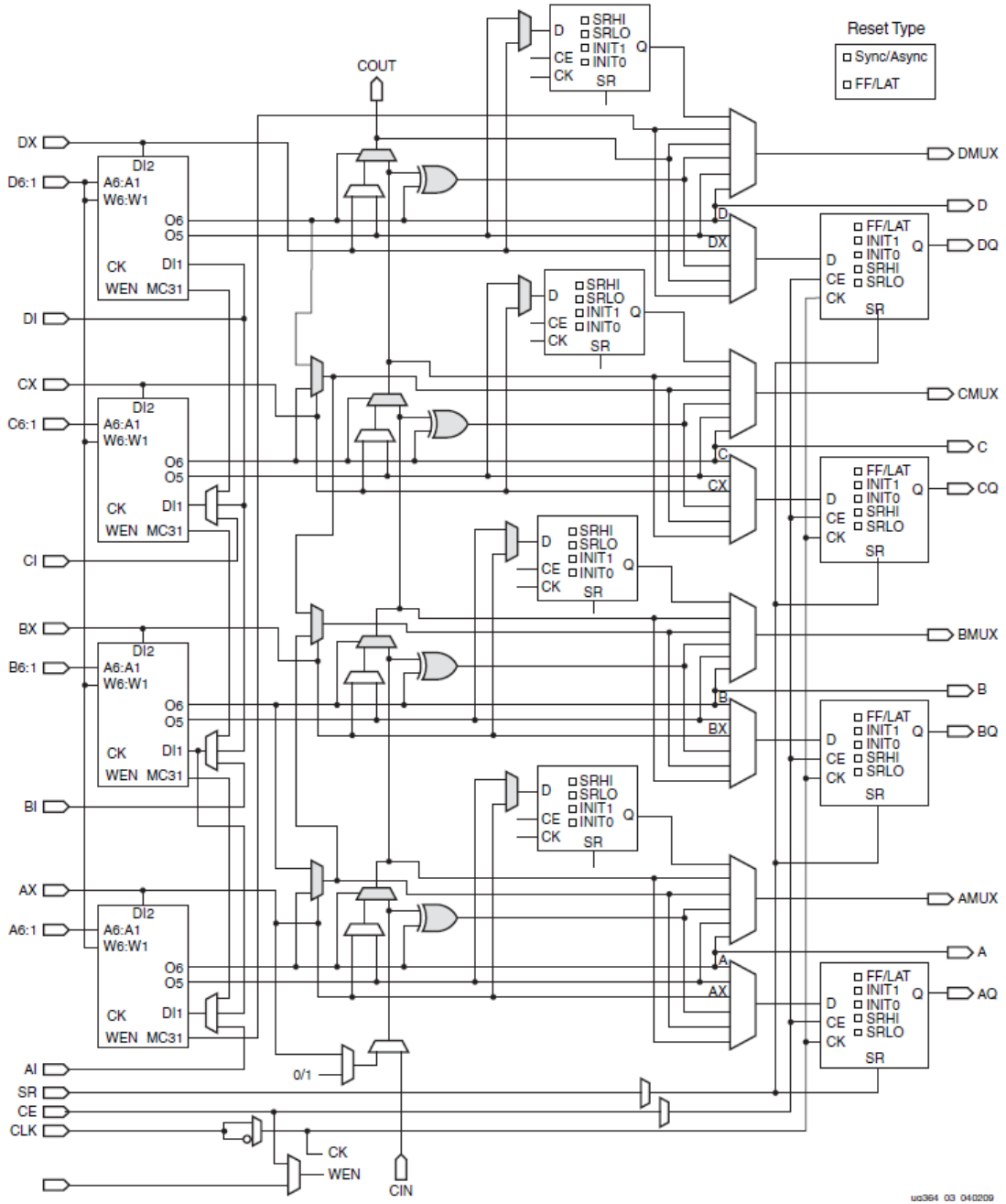
# Example Virtex-6

- 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**
- Heterogeneous Architectures

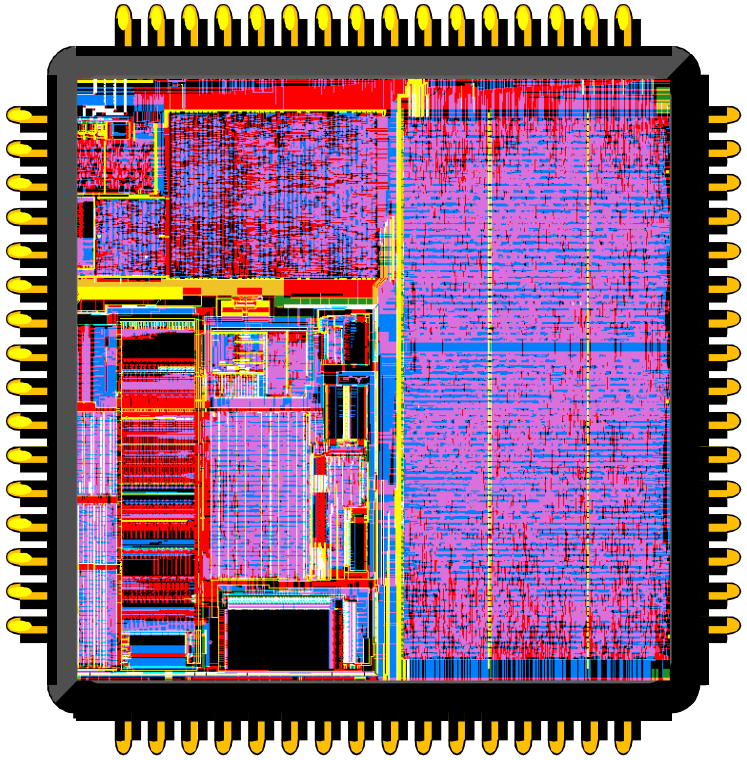
# Application Specific Circuits (ASICS)

*Custom-designed circuits* are 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).



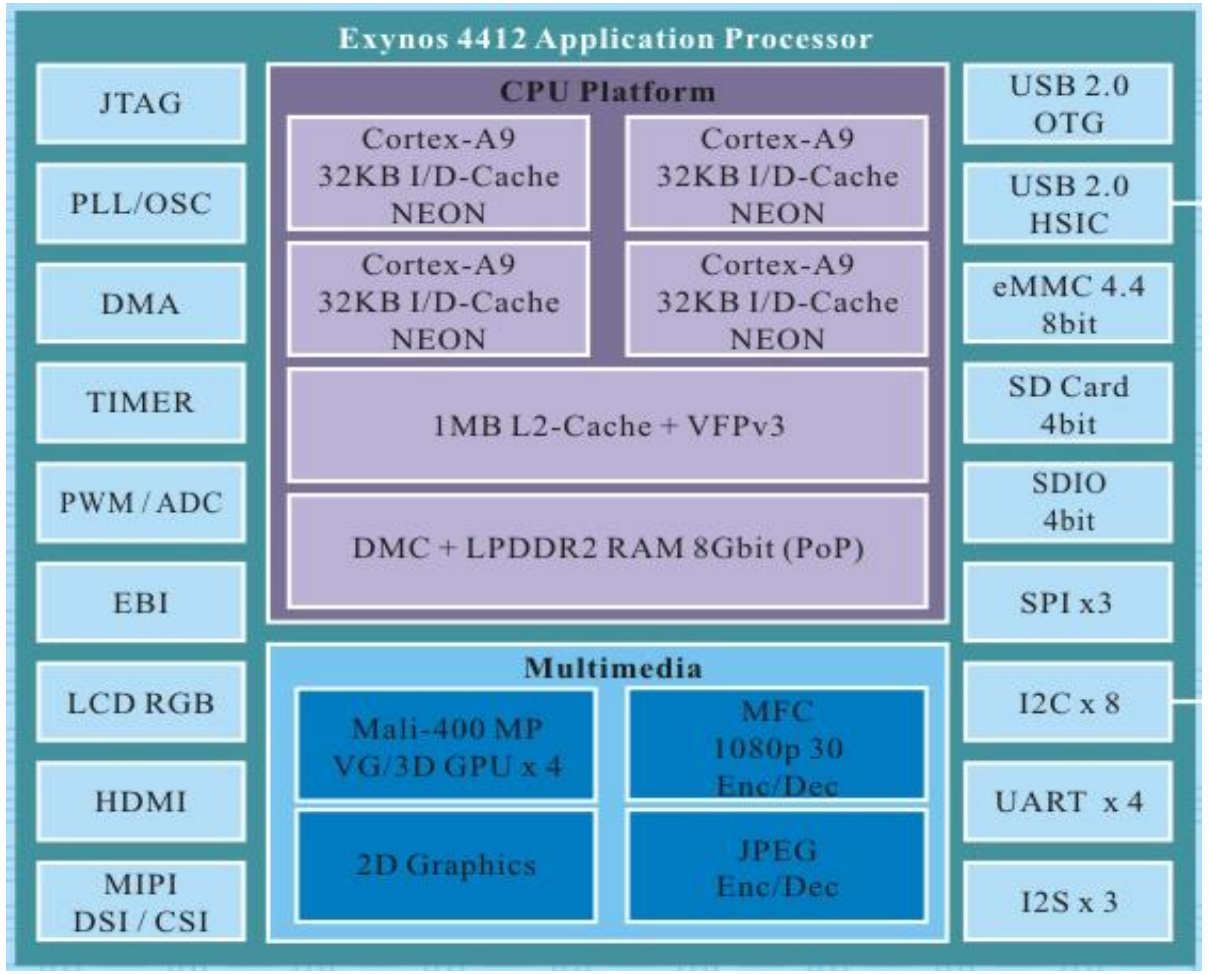
# Topics

- System Specialization
- Application Specific Instruction Sets
  - Micro Controller
  - Digital Signal Processors and VLIW
- Programmable Hardware
- ASICs
- **Heterogeneous Architectures**

# Example: Heterogeneous Architecture

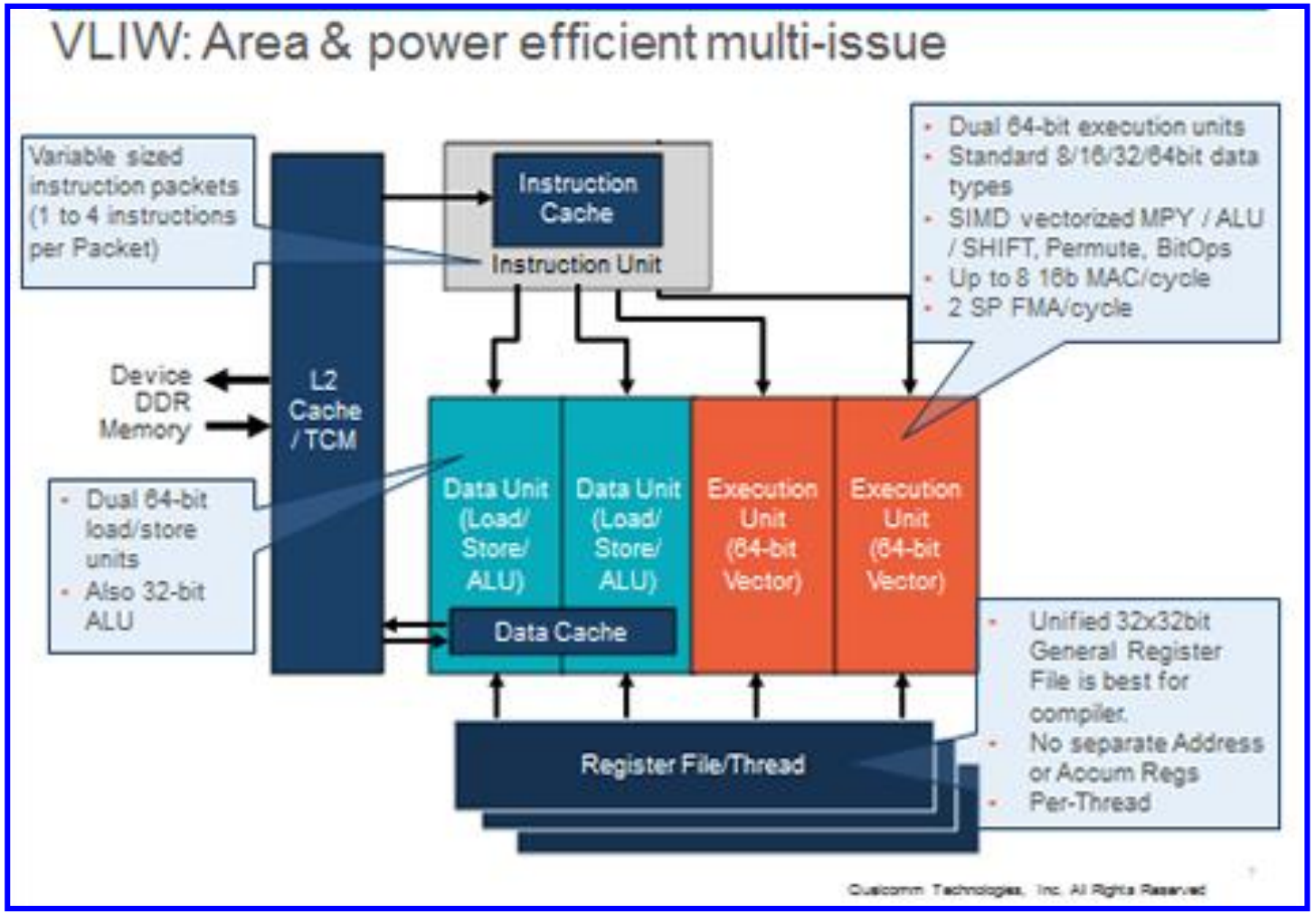
Samsung Galaxy Note II

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



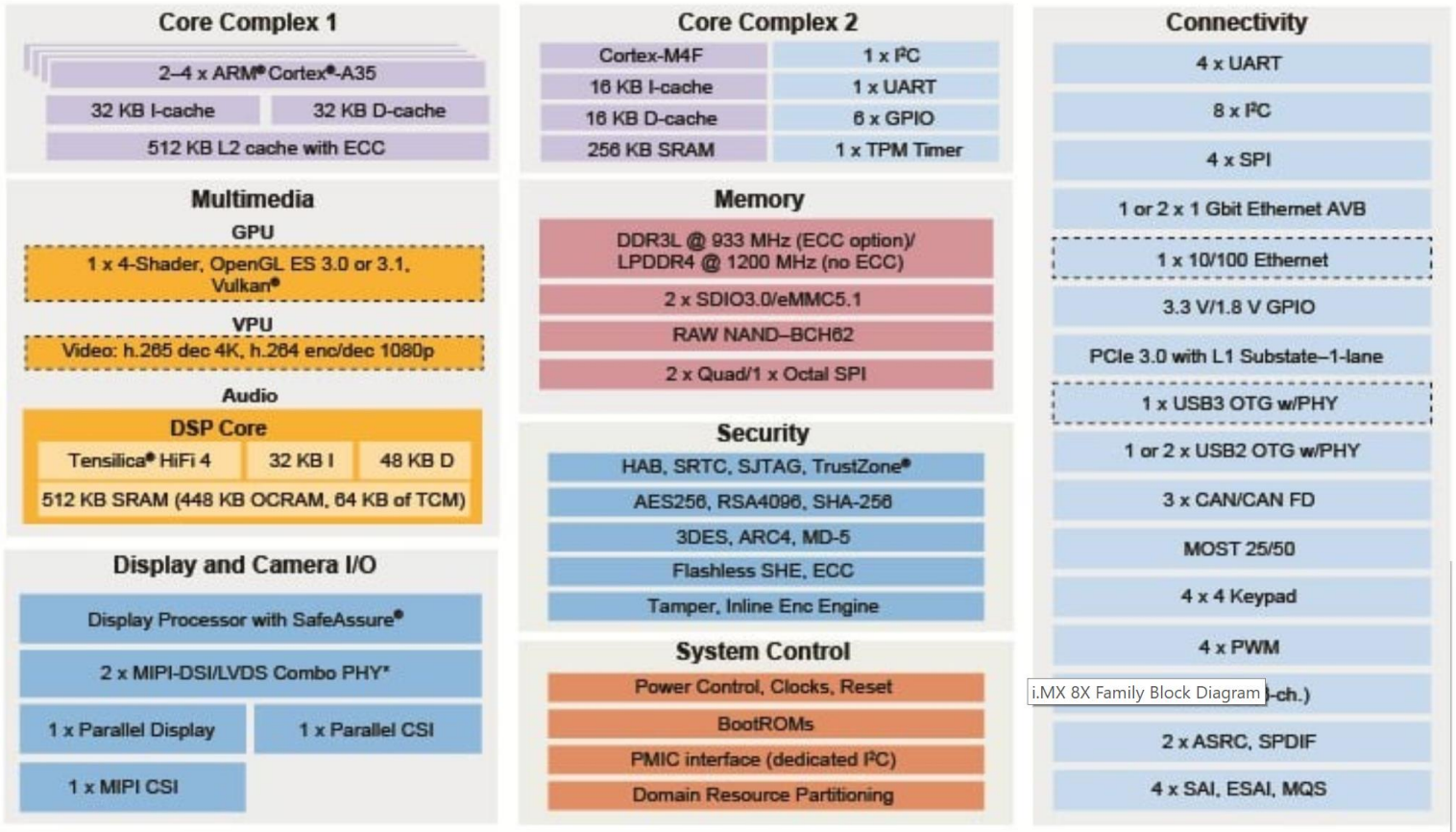
# Example: Heterogeneous Architecture

Hexagon DSP



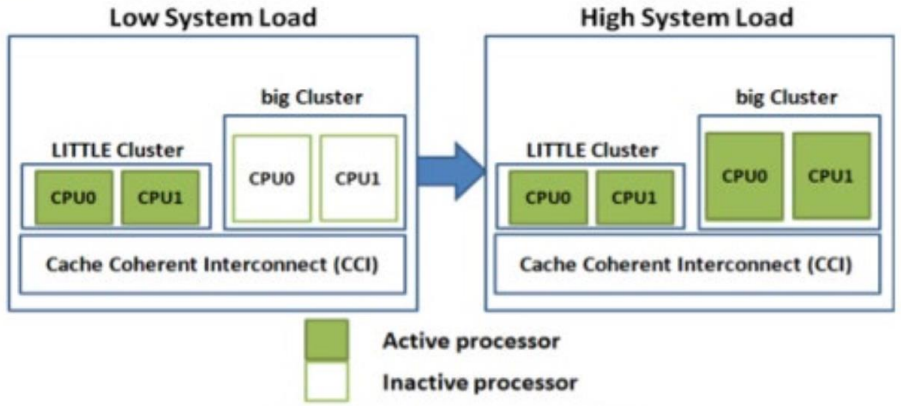

Snapdragon 835
(Galaxy S8)

# Example: ARM big.LITTLE Architecture





Toradex Colibri Compute-on-Module



# Embedded Systems

## 9. Power and Energy

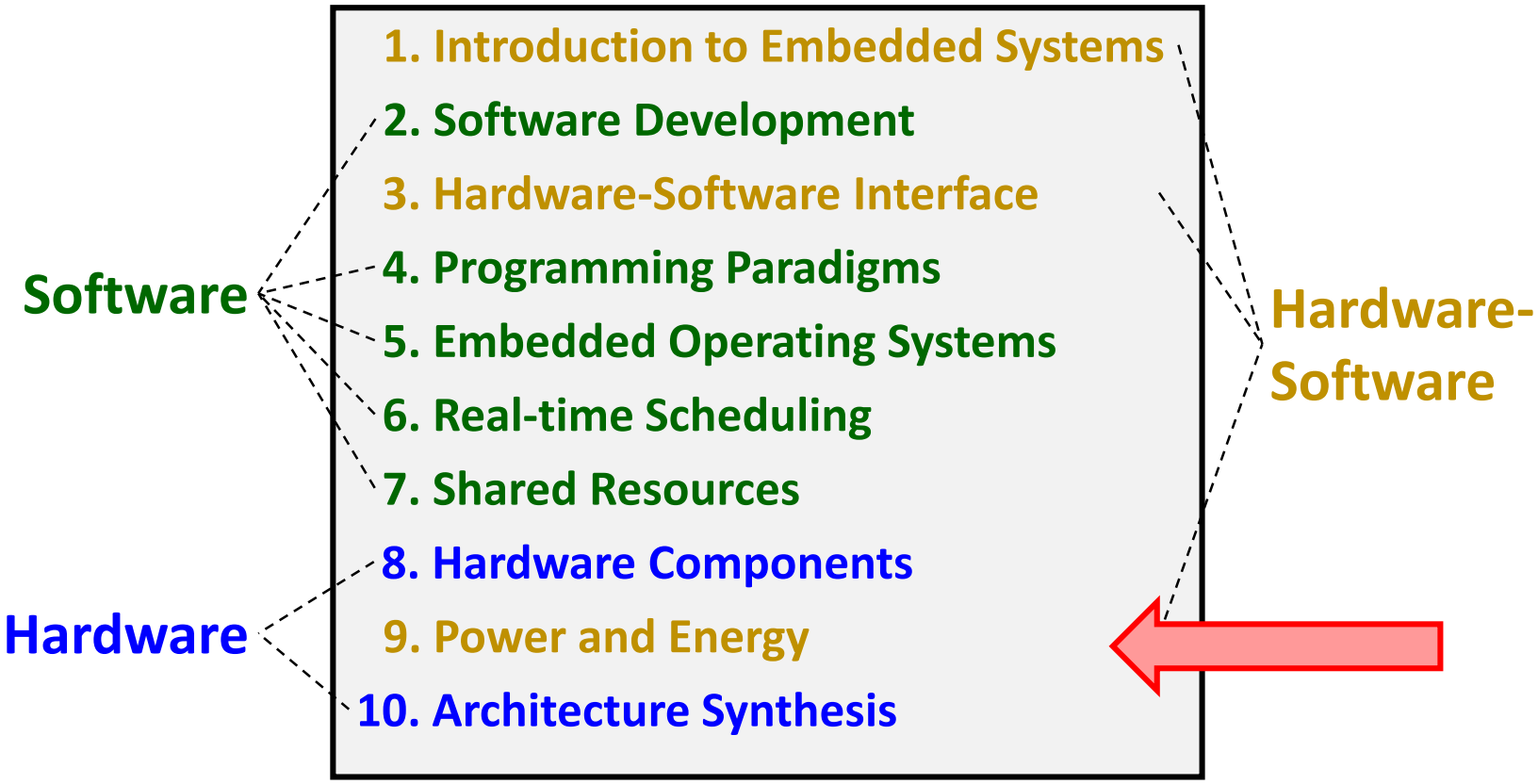
© Lothar Thiele

Computer Engineering and Networks Laboratory

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Lecture Overview



# General Remarks

# Power and Energy Consumption

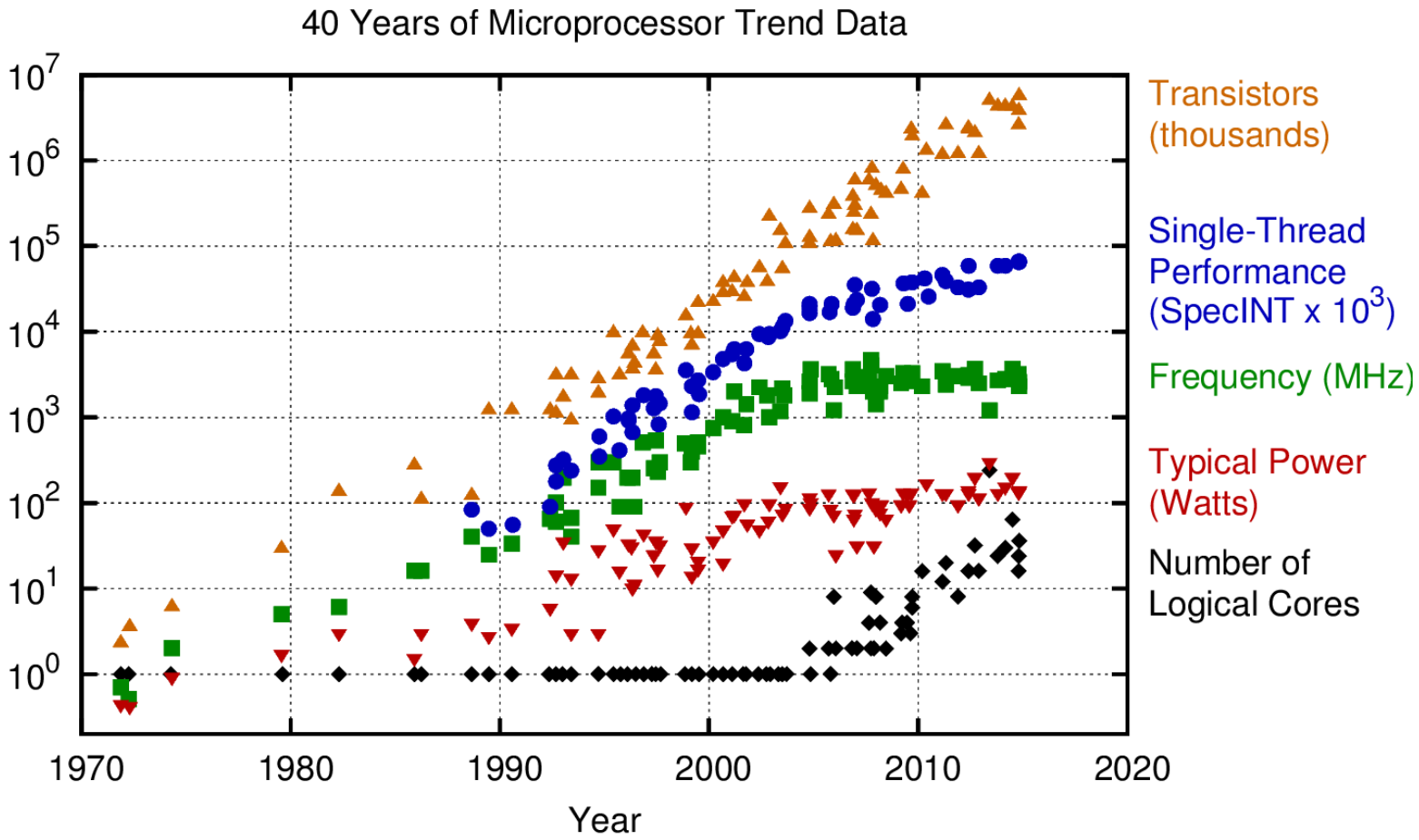
- Statements that are true since a decade or longer:

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

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

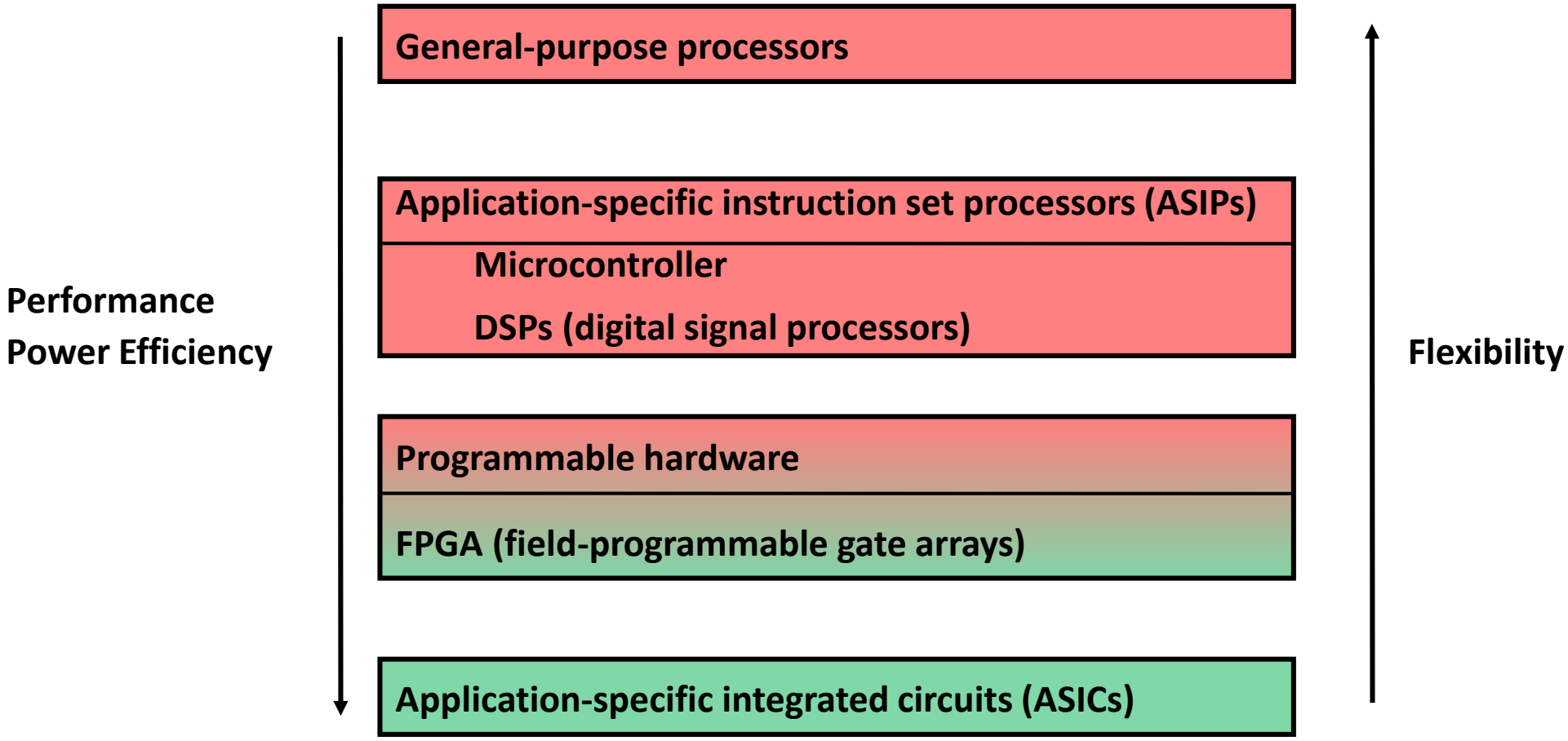

- Main ***reasons*** are:
  - power provisioning is expensive
  - battery capacity is growing only slowly
  - devices may overheat
  - energy harvesting (e.g. from solar cells) is limited due to the relatively low energy available density

# Some Trends



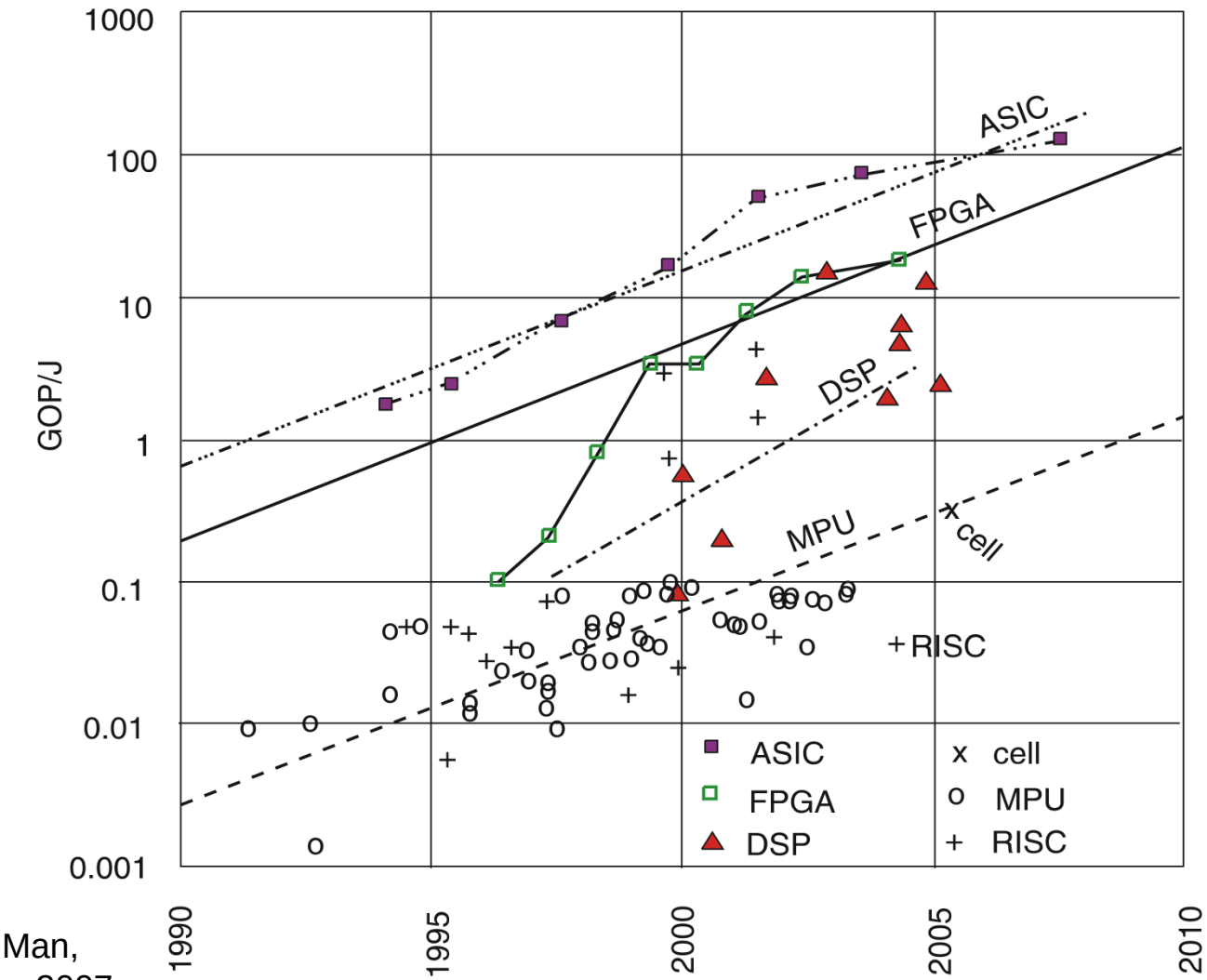

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

# Implementation Alternatives



# Energy Efficiency

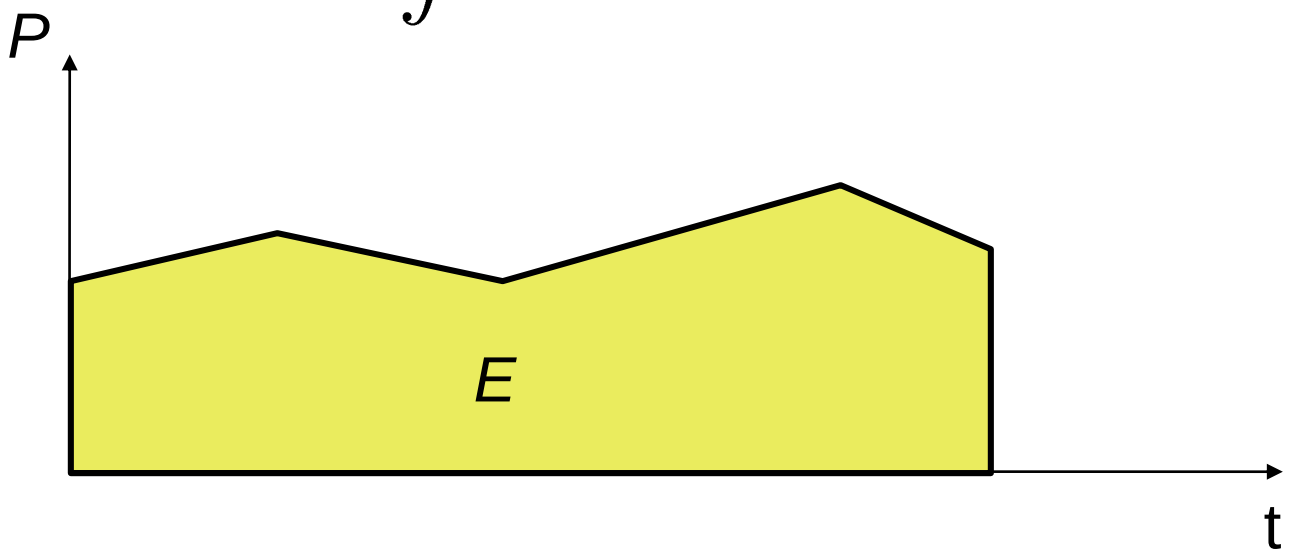
- It is necessary to ***optimize HW and SW.***
- Use ***heterogeneous architectures*** in order to adapt to required performance and to class of application.
- Apply ***specialization techniques***.



# Power and Energy

# Power and Energy

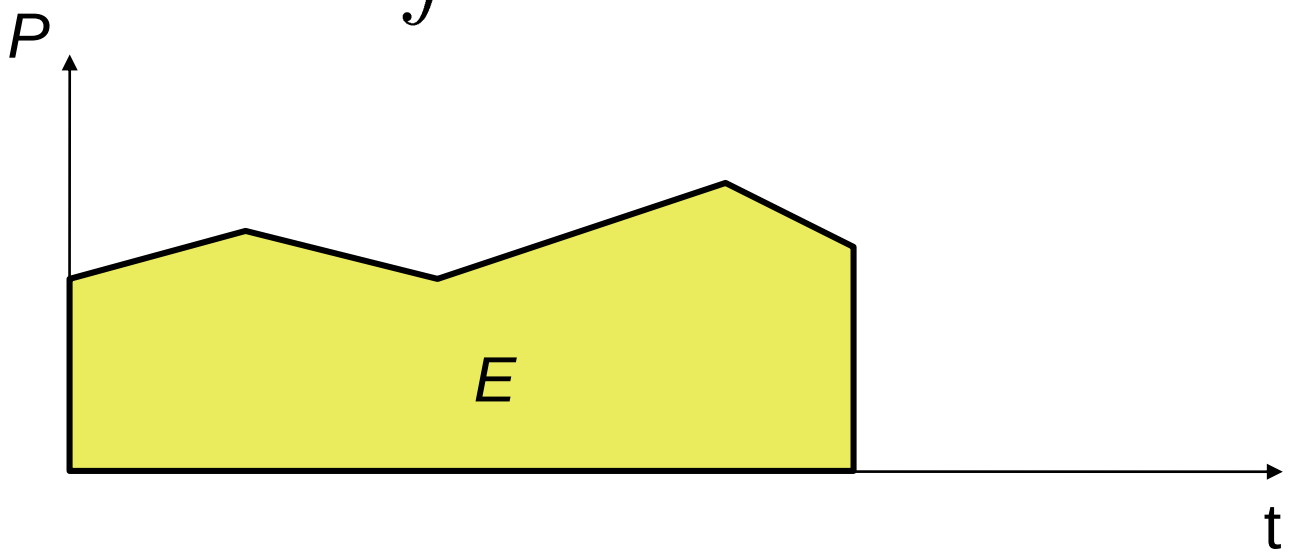
$$E = \int P(t)dt$$




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

# Power and Energy

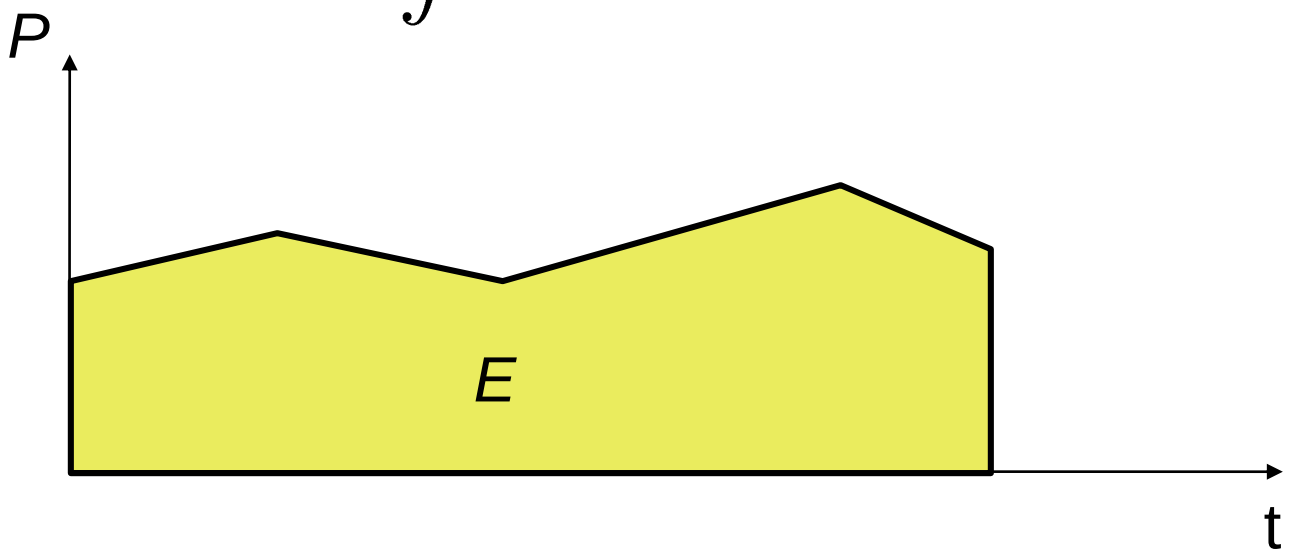
$$E = \int P(t)dt$$




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

# Power and Energy

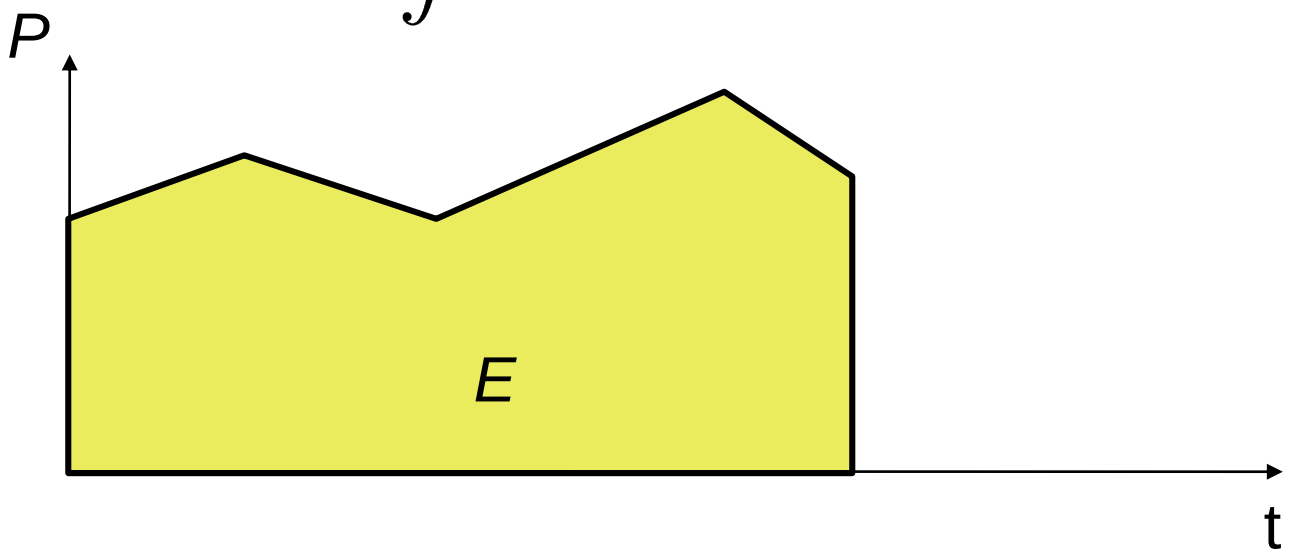
$$E = \int P(t)dt$$




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

# Power and Energy

$$E = \int P(t)dt$$

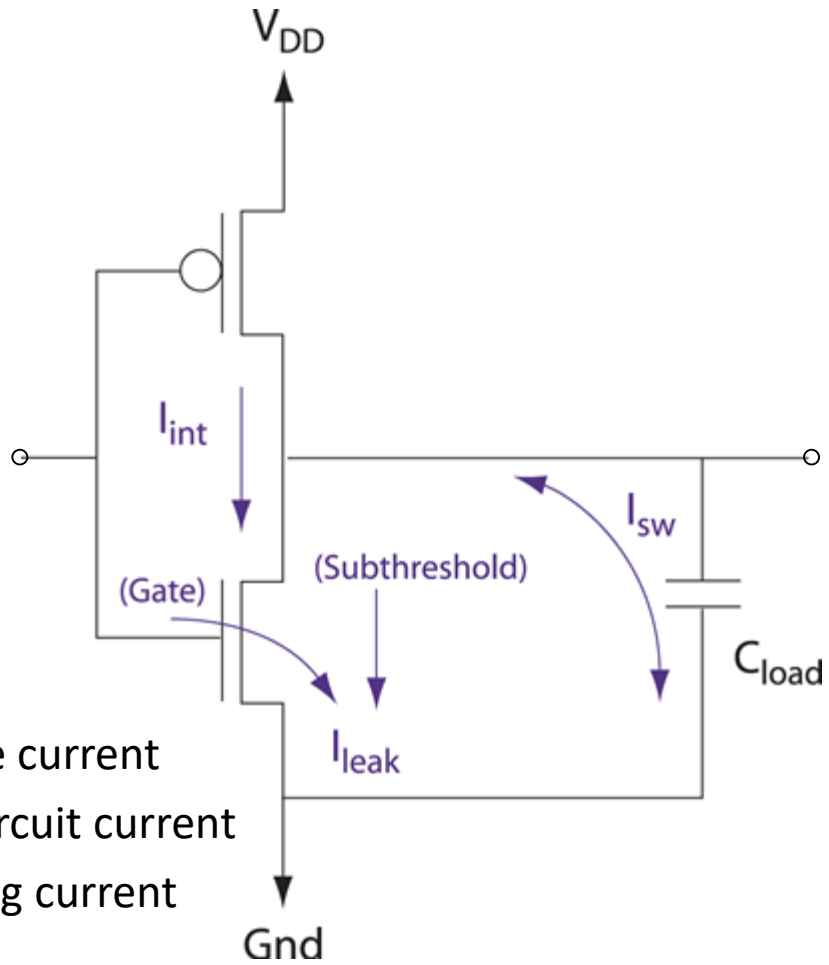



In some cases, faster execution also means less energy, but the opposite may be true if power has to be increased to allow for a faster execution.

# Low Power vs. Low Energy

- Minimizing the ***power consumption*** *(voltage * current)* is important for
  - the design of the power supply and voltage regulators
  - the dimensioning of interconnect between power supply and components
  - cooling (short term cooling)
    - high cost
    - 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 (energy harvesting, solar panels, maintenance/batteries)
  - long lifetimes, low temperatures

# Power Consumption of a CMOS Gate

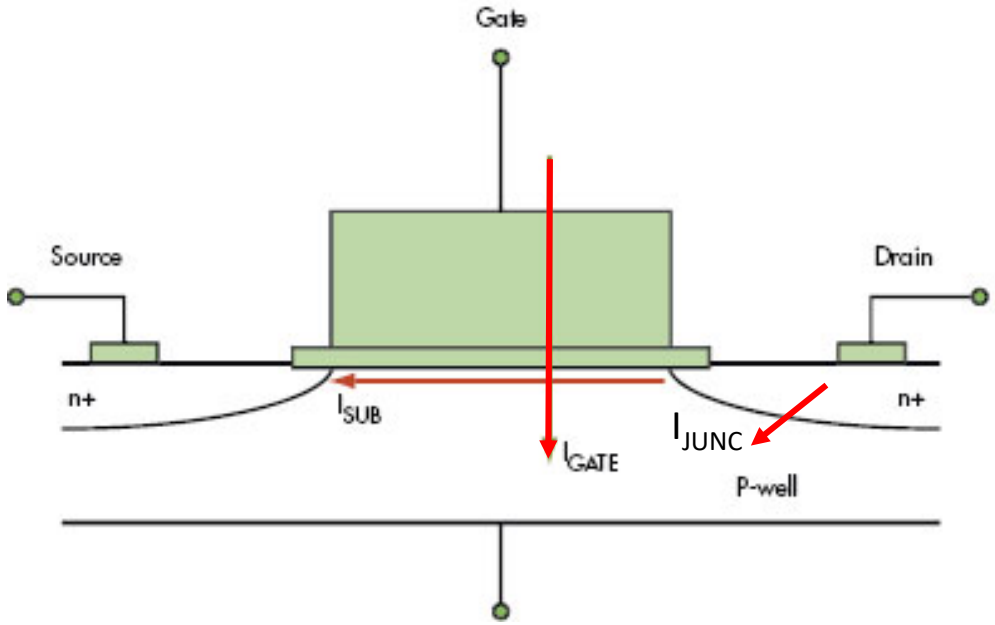



$I_{leak}$ : leakage current

$I_{int}$ : short circuit current

$I_{sw}$ : switching current

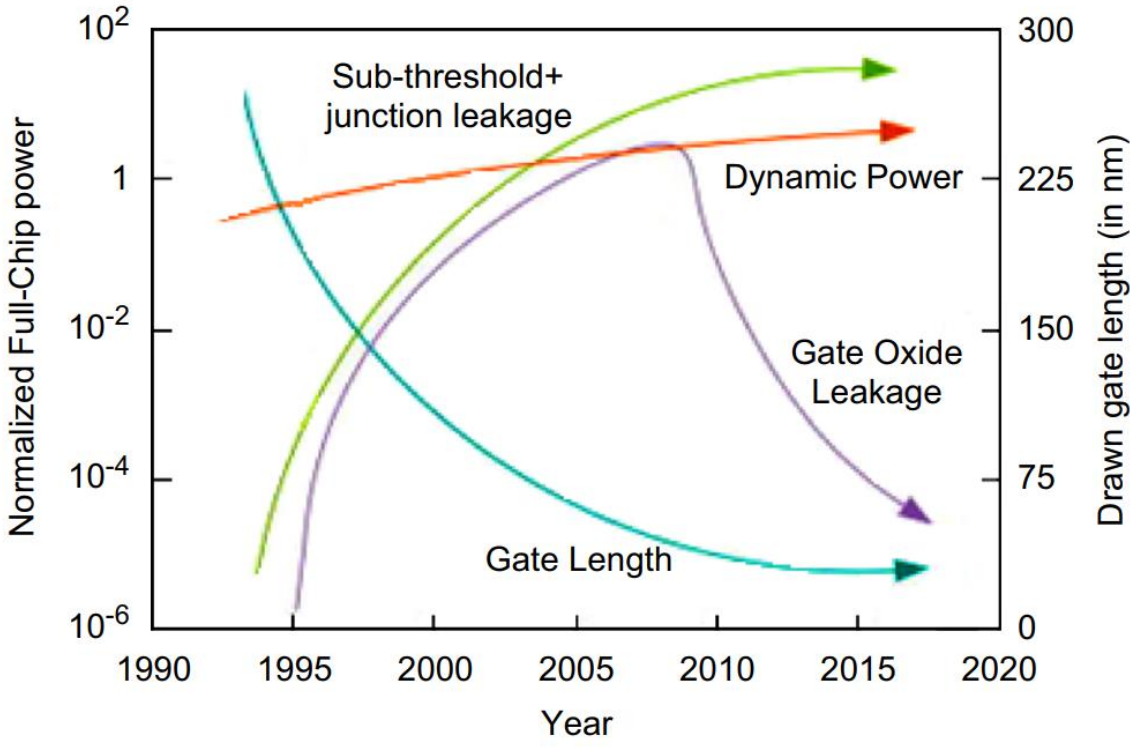
subthreshold ($I_{SUB}$), junction ($I_{JUNC}$) and gate-oxide ($I_{GATE}$) leakage



# Power Consumption of a CMOS Processors

***Main sources***:

- Dynamic power consumption
  - charging and discharging capacitors
  - Short circuit power consumption: short circuit path between supply rails during switching
- Leakage and static power
  - gate-oxide/subthreshold/junction leakage
  - becomes one of the major factors due to shrinking feature sizes in semiconductor technology

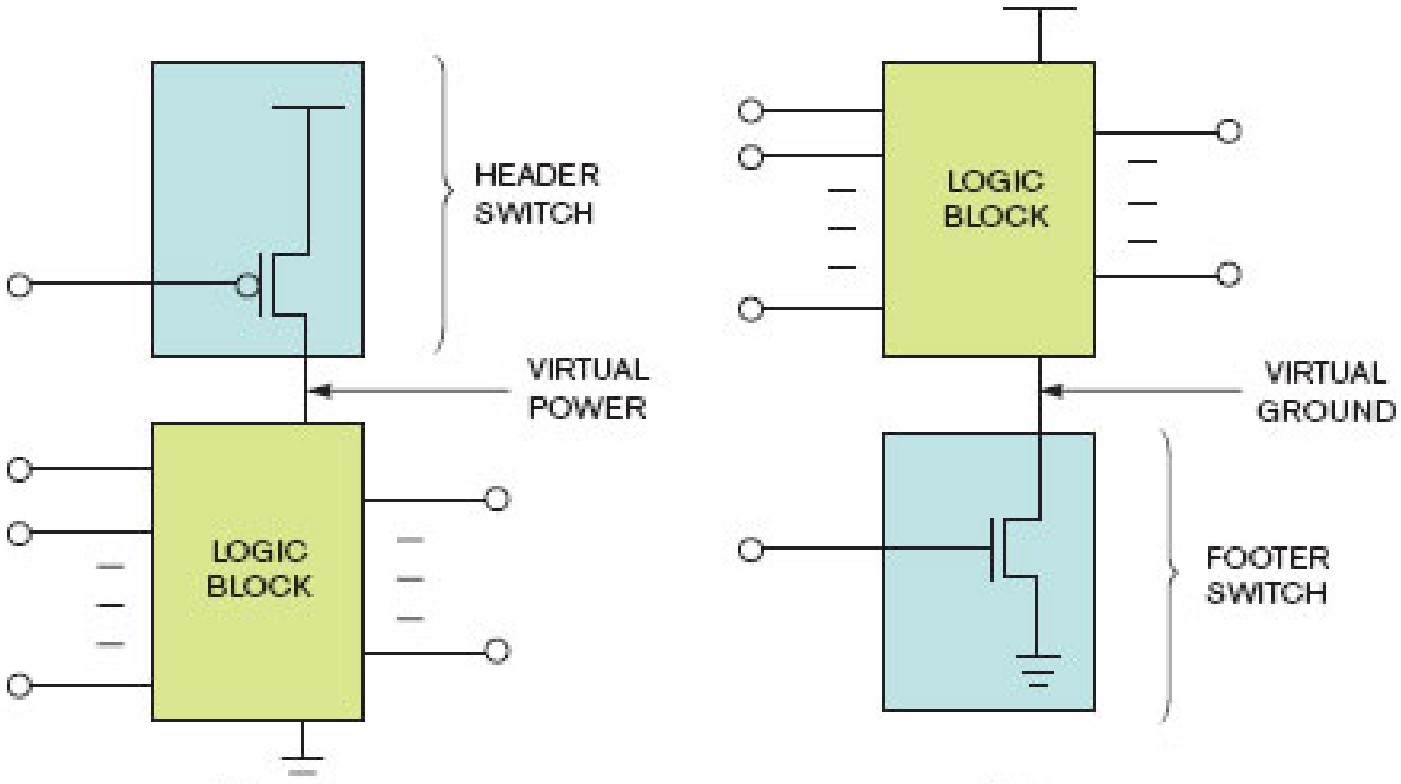



[J. Xue, T. Li, Y. Deng, Z. Yu, Full-chip leakage analysis for 65 nm CMOS technology and beyond, Integration VLSI J. 43 (4) (2010) 353–364]

# Reducing Static Power - 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



# Dynamic Voltage Scaling (DVS)

***Average power consumption of CMOS circuits (ignoring leakage):***

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

$V_{dd}$ : supply voltage
$\alpha$ : switching activity
$C_L$ : load capacity
$f$ : clock frequency

***Delay of 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 $V_{dd}$ .

# Dynamic Voltage Scaling (DVS)

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

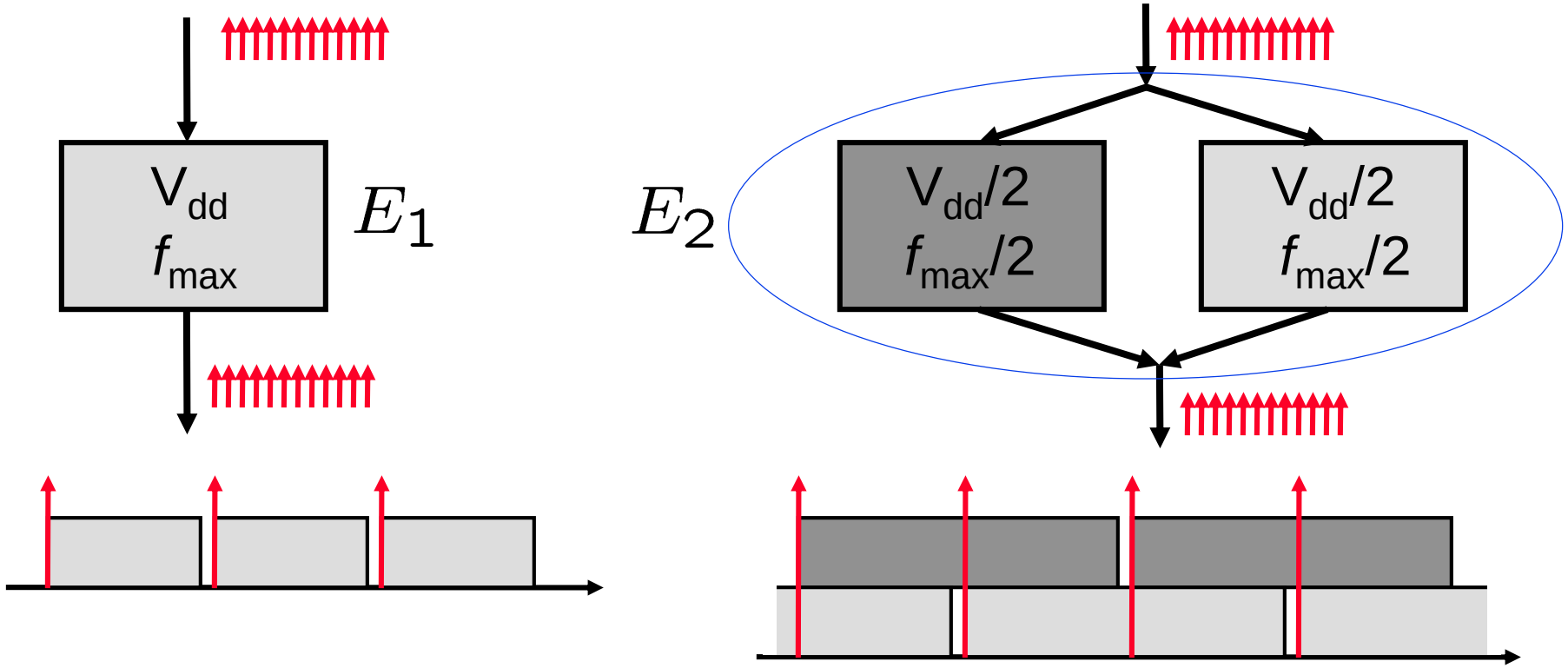
$$E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 \ (\#\text{cycles})$$

Saving energy for a given task:

- reduce the supply voltage $V_{dd}$
- reduce switching activity $\alpha$
- reduce the load capacitance $C_L$
- reduce the number of cycles *#cycles*

# Techniques to Reduce Dynamic Power

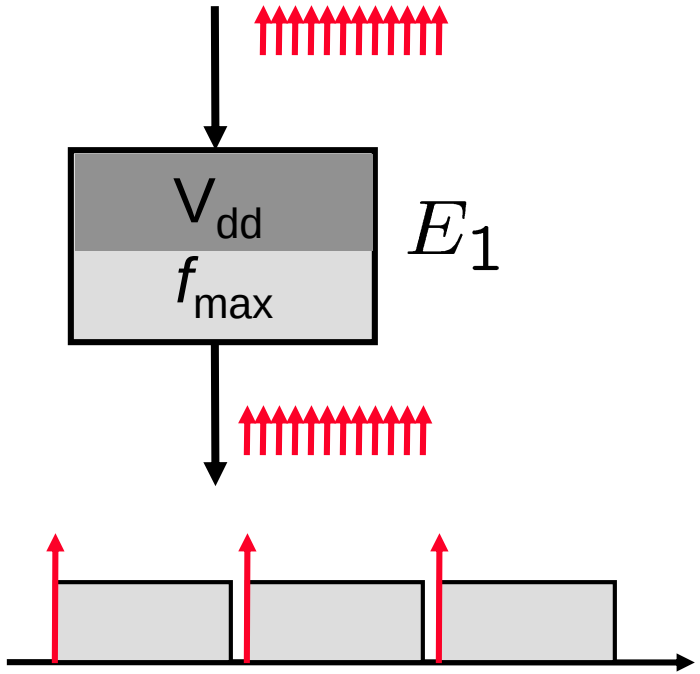
# Parallelism

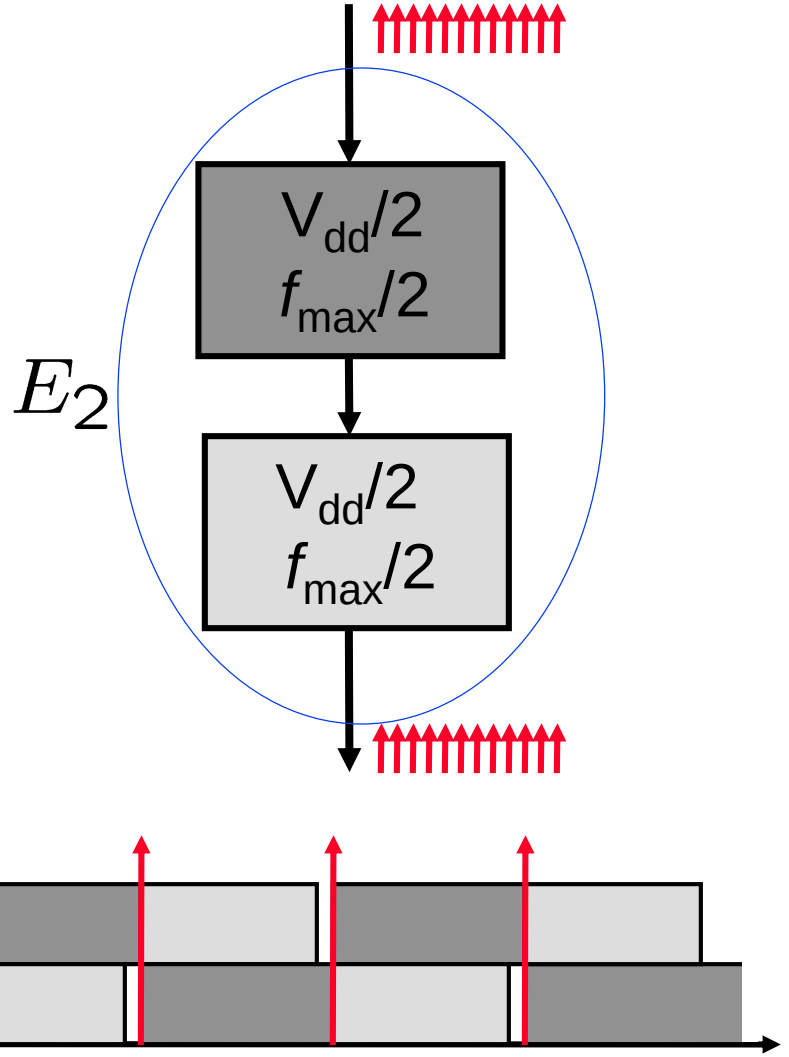



$$E \sim V_{dd}^2 \,(\#\text{cycles})$$

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

# Pipelining



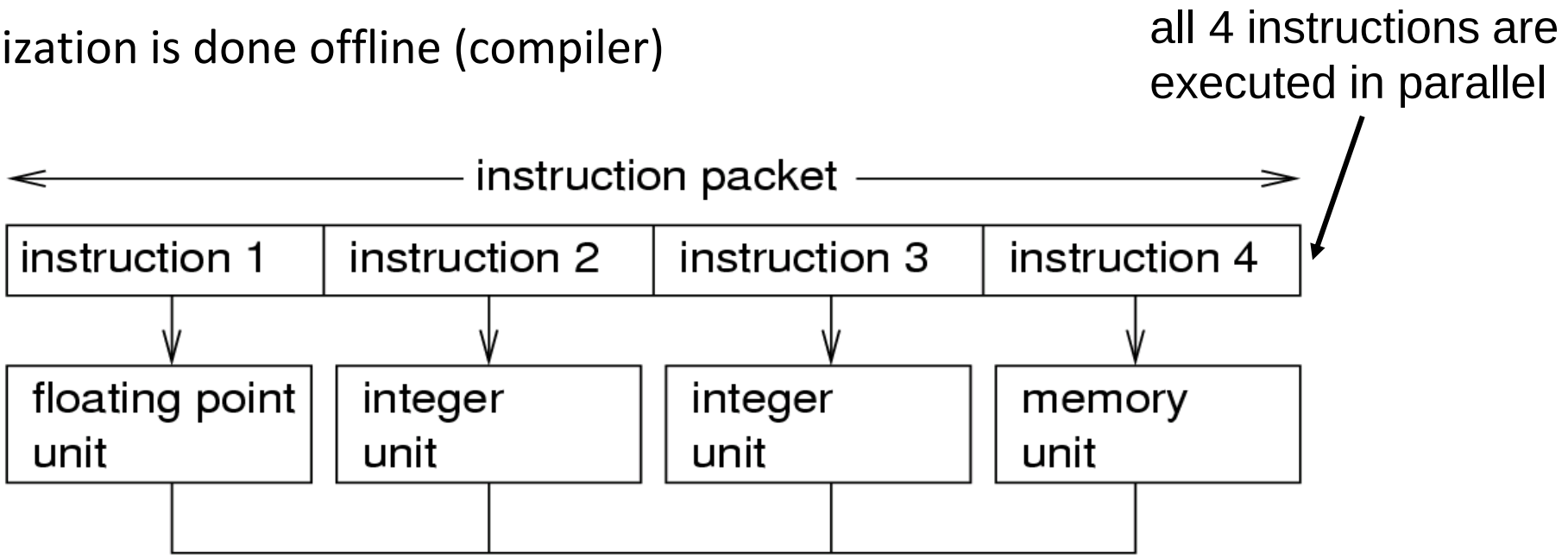




$$E \sim V_{dd}^2 \ (\#\text{cycles})$$

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

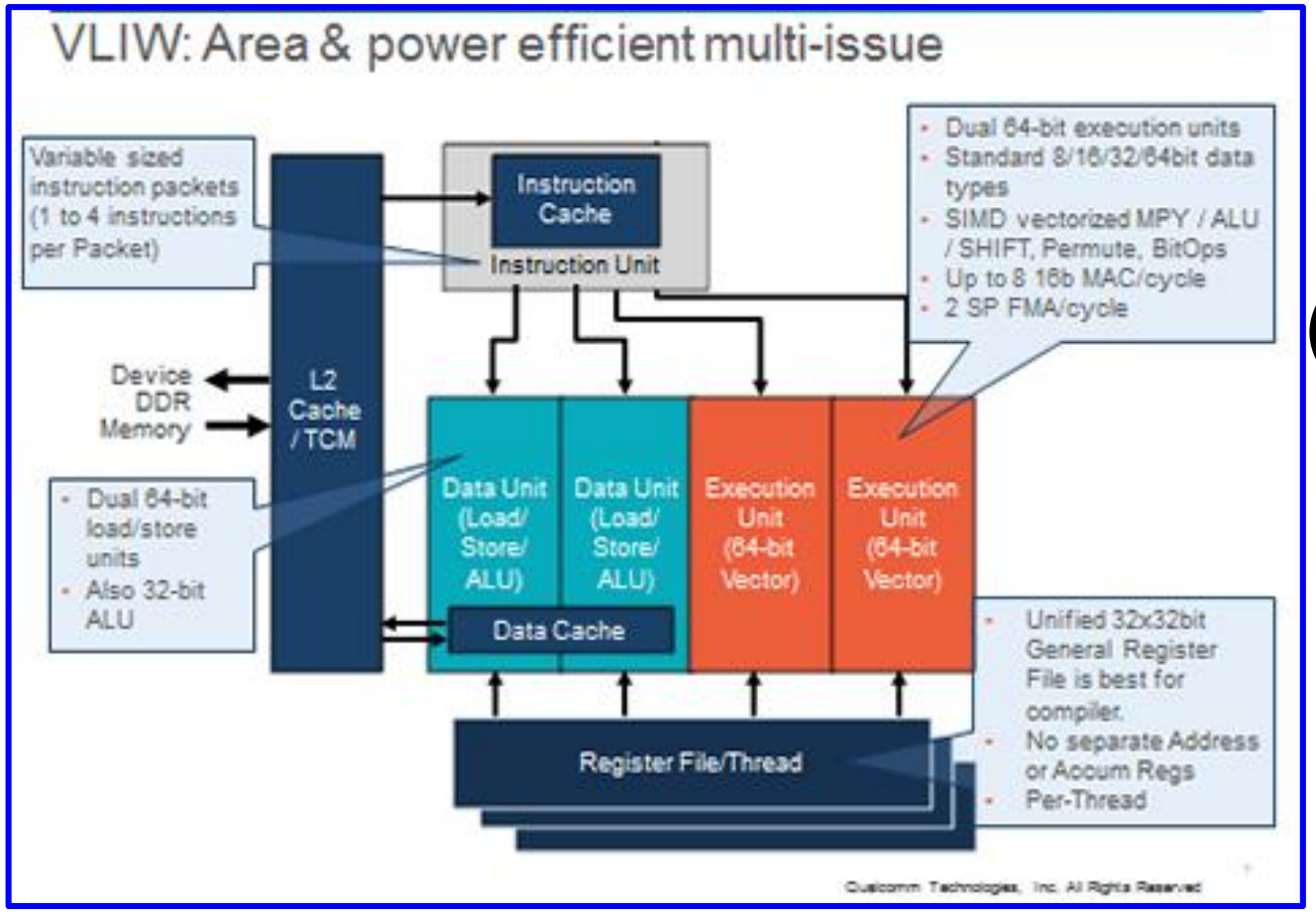
# VLIW (Very Long Instruction Word) Architectures

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

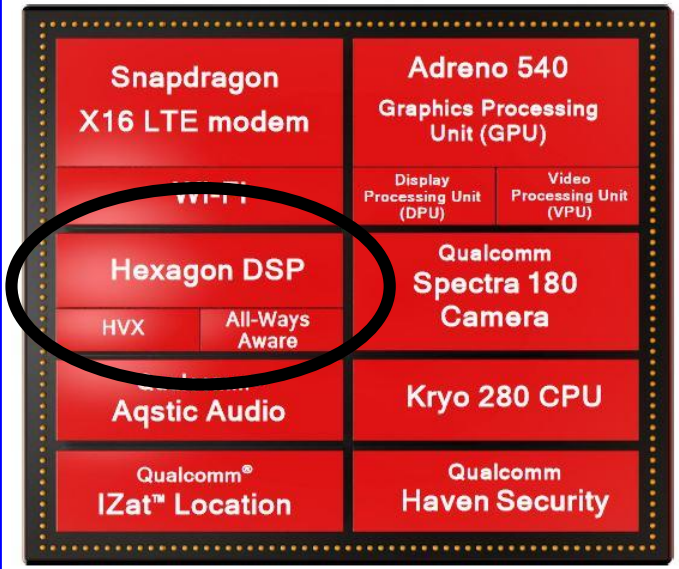


# Example: Qualcomm Hexagon

Hexagon DSP




Snapdragon 835 (Galaxy S8)



# Dynamic Voltage and Frequency Scaling - Optimization

# Dynamic Voltage and Frequency Scaling (DVFS)

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

energy per cycle

reduce voltage -> reduce energy per task

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

$$f \sim \frac{1}{\tau} \sim V_{dd}$$

reduce voltage -> reduce clock frequency

maximum frequency of operation

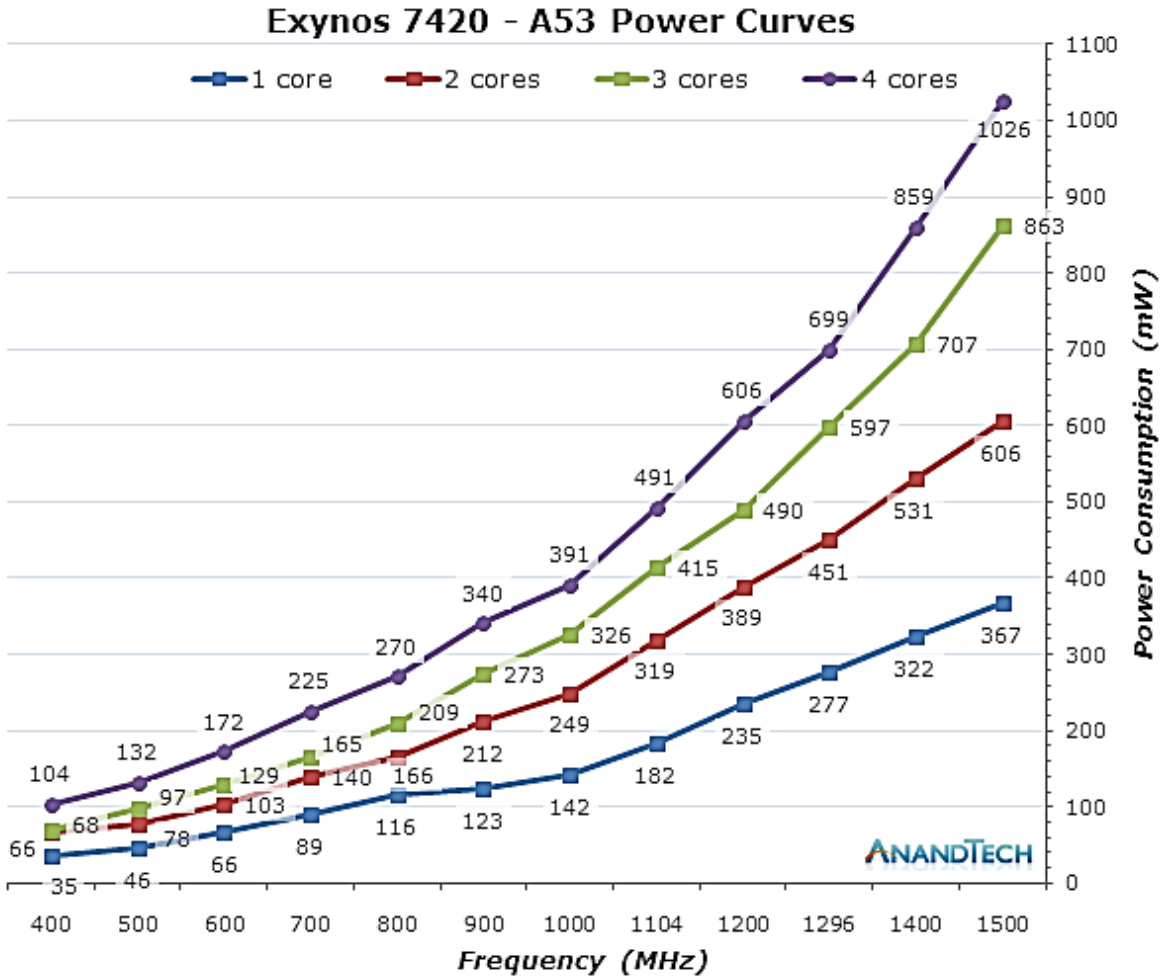
gate delay

Saving energy for a given task:

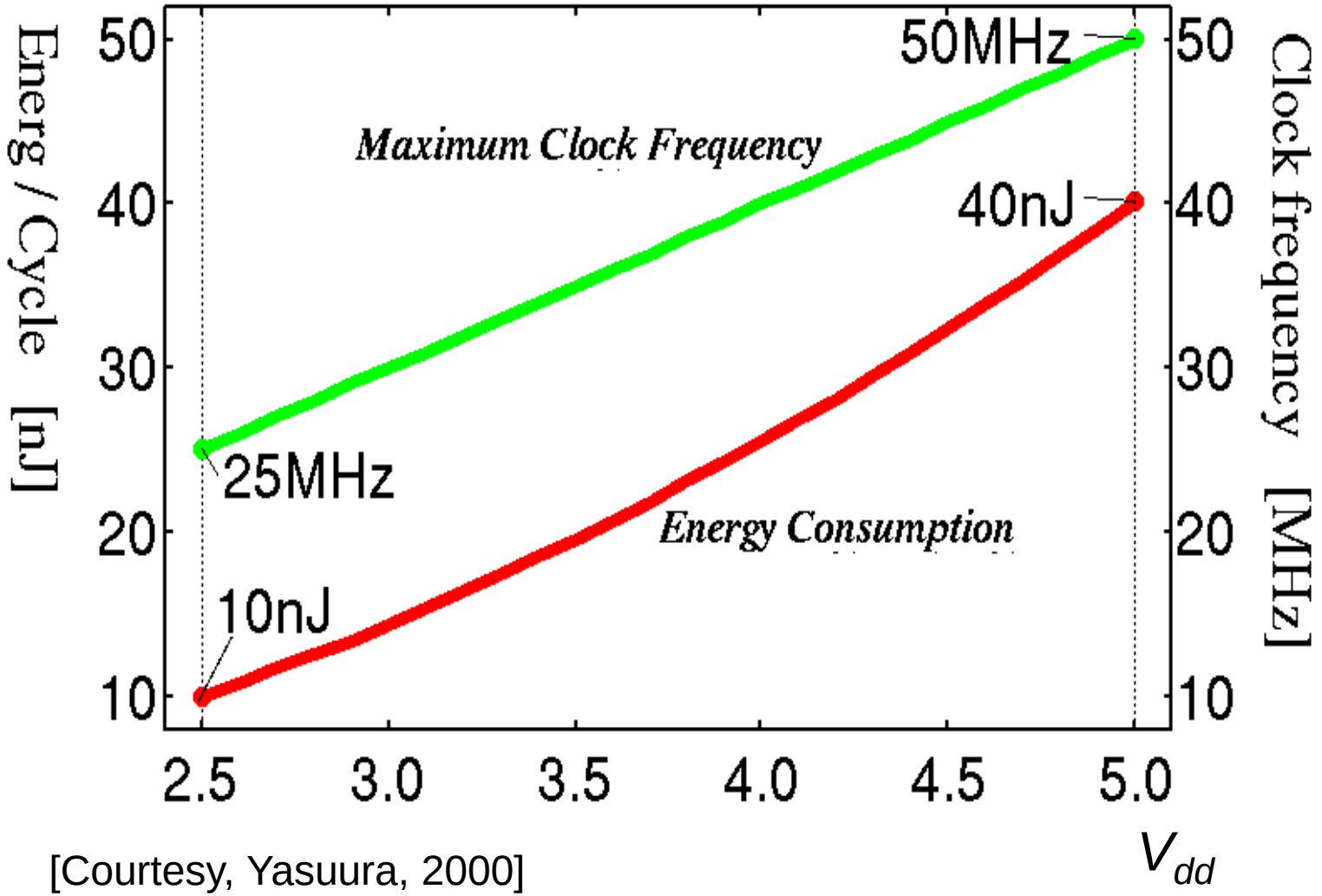
- reduce the supply voltage $V_{dd}$
- reduce switching activity $\alpha$
- reduce the load capacitance $C_L$
- reduce the number of cycles *#cycles*

# Example DVFS: Samsung Exynos (ARM processor)

ARM processor core A53 on the Samsung Exynos 7420 (used in mobile phones, e.g. Galaxy S6)



# Example: Dynamic Voltage and Frequency Scaling

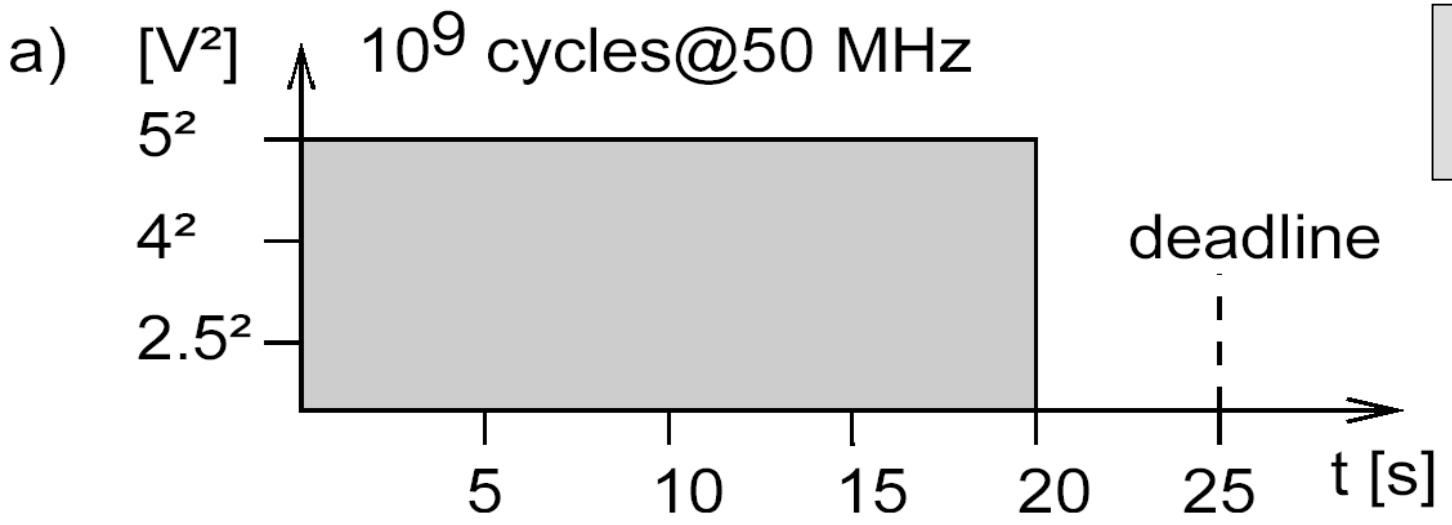



[Courtesy, Yasuura, 2000]

# Example: DVFS – Complete Task as Early as Possible

| $V_{dd}$ [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 |

We suppose a task that needs $10^9$ cycles to execute within 25 seconds.

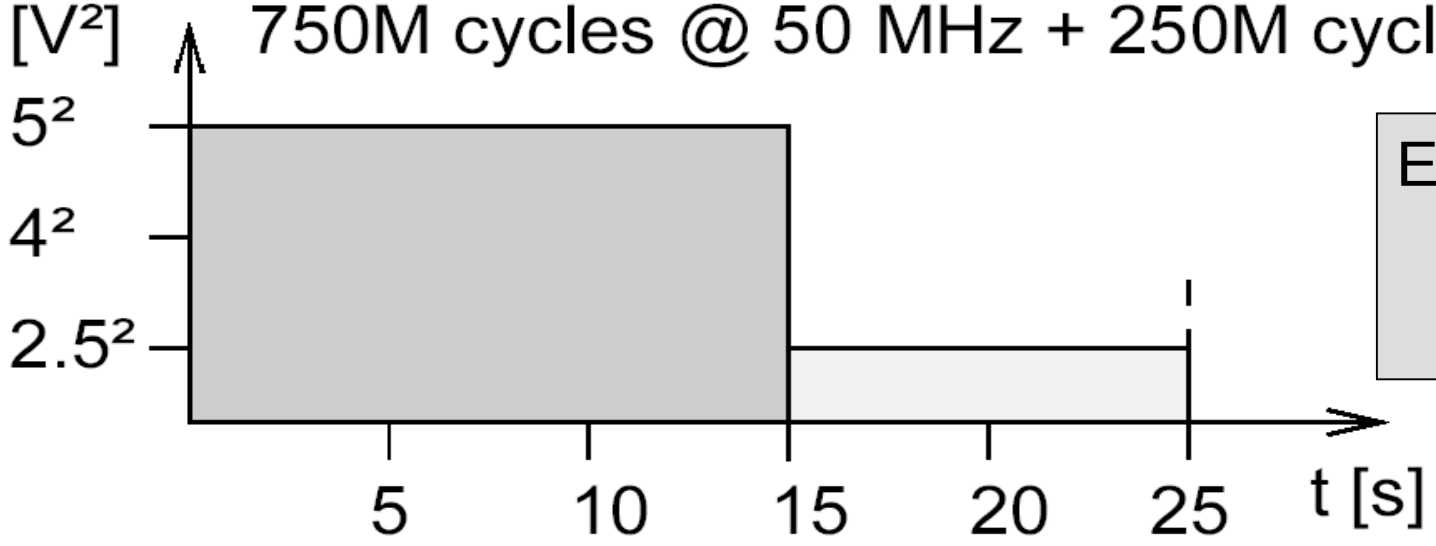



$E_a = 10^9 \times 40 \times 10^{-9}$
$= 40$ [J]

# Example: DVFS – Use 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) 750M cycles @ 50 MHz + 250M cycles @ 25 MHz



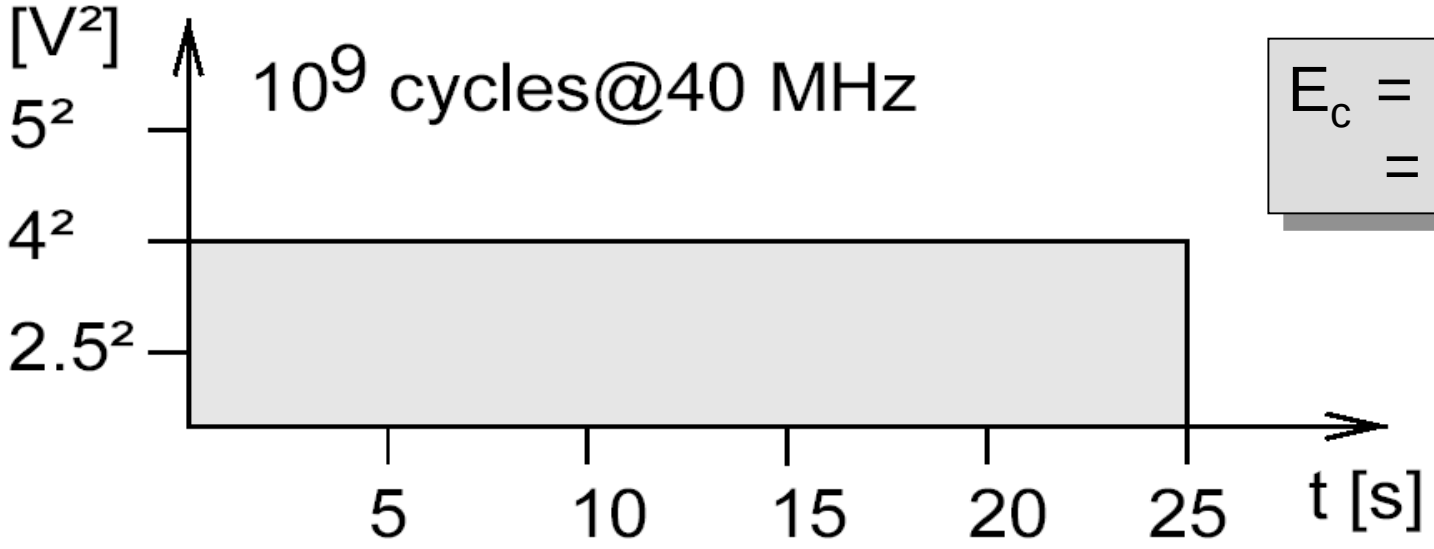

$E_b = 750\ 10^6 \times 40 \times 10^{-9} + 250\ 10^6 \times 10 \times 10^{-9}$

$= 32.5$ [J]

# Example: DVFS – Use One Voltage

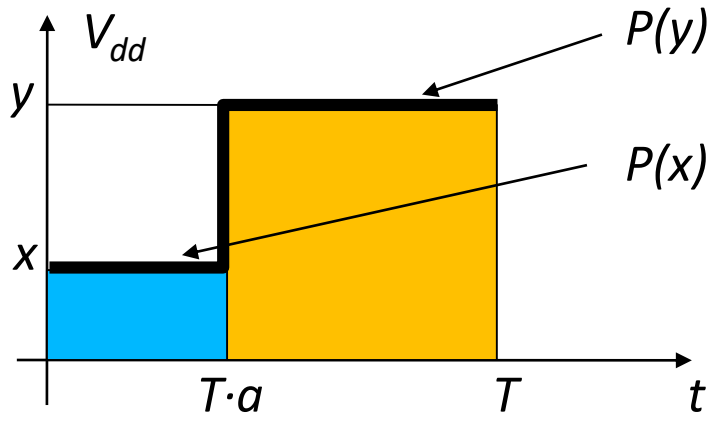
| $V_{dd}$ [V] | 5.0 | 4.0 | 2.5 |
|---|---|---|---|
| Energy per cycle [nJ] | 40 | 25 | 10 |
| $f_{max}$ [MHz] | 50 | 40 | 25 |
| cycle time [ns] | 20 | 25 | 40 |

c)




$E_c = 10^9 \times 25 \times 10^{-9}$
$= 25$ [J]

# DVFS: Optimal Strategy




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

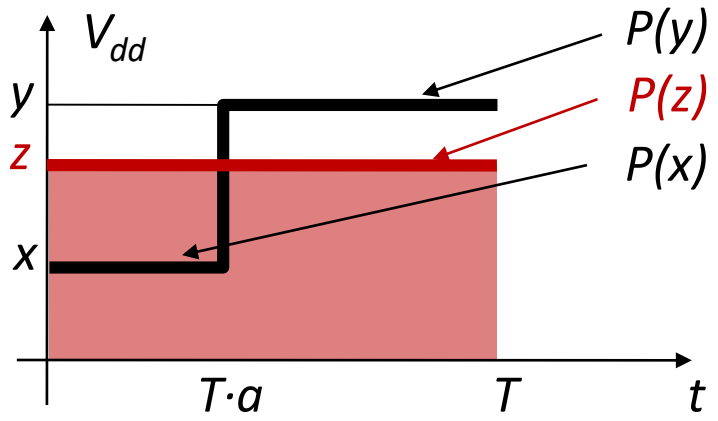
gate delay: $\tau \sim \frac{1}{V_{dd}}$

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 $(1\text{-}a) \cdot T$ time units;
  energy consumption: $T \cdot ( P(x) \cdot a + P(y) \cdot (1\text{-}a) )$

# DVFS: Optimal Strategy




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

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

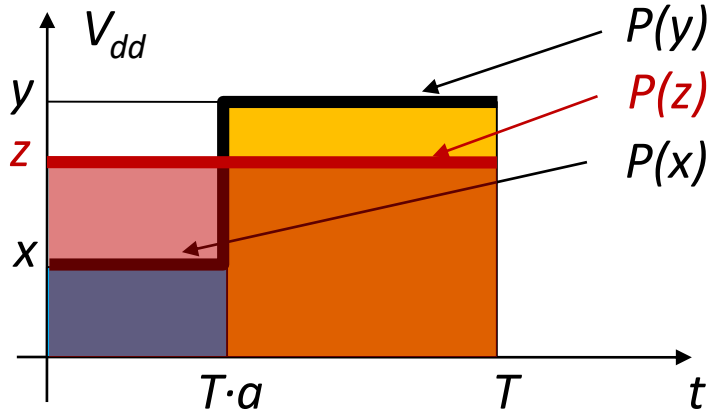
execution rate: $f(t) \sim V_{dd}(t)$

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 $(1\text{-}a) \cdot T$ time units;
  energy consumption: $T \cdot ( P(x) \cdot a + P(y) \cdot (1\text{-}a) )$

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

# DVFS: Optimal Strategy




$z \cdot T = a \cdot T \cdot x + (1\text{-}a) \cdot T \cdot y$

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

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

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

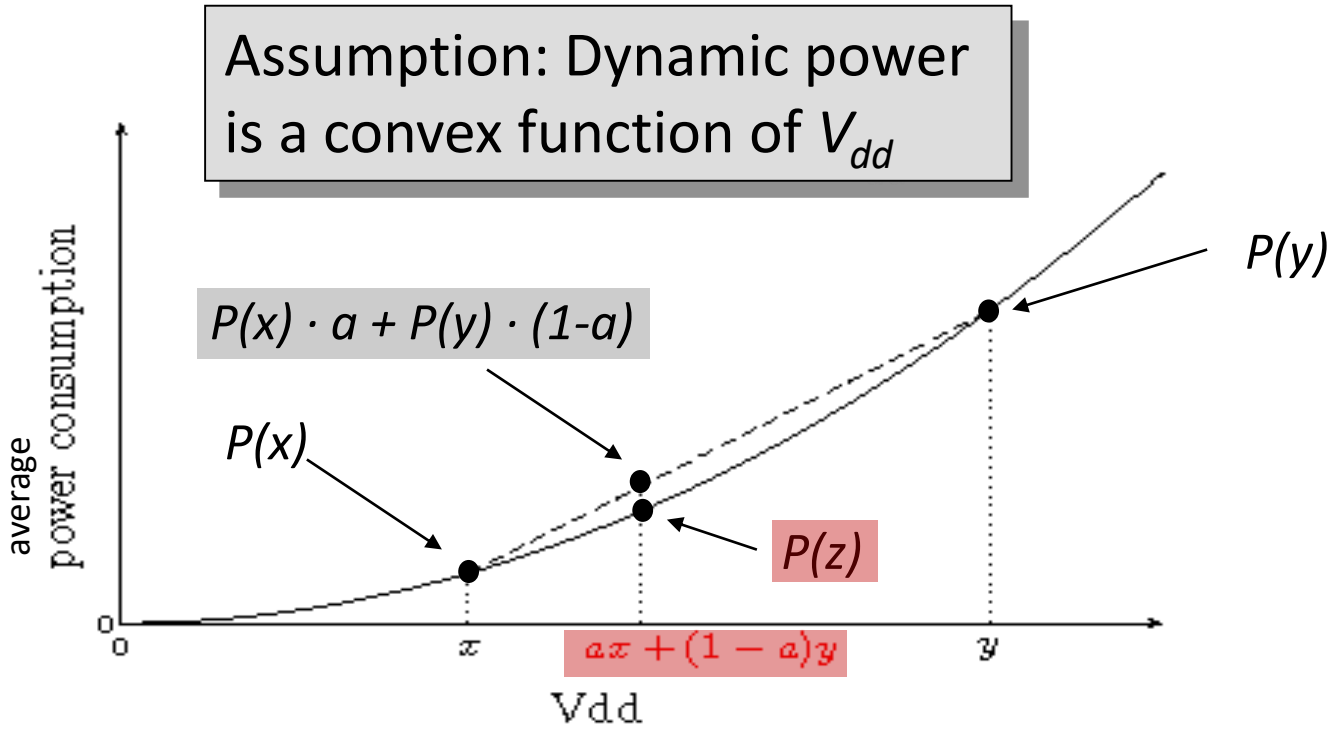
execution rate: $f(t) \sim V_{dd}(t)$

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 $(1\text{-}a) \cdot T$ time units;
  energy consumption: $T \cdot ( P(x) \cdot a + P(y) \cdot (1\text{-}a) )$

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

# DVFS: 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

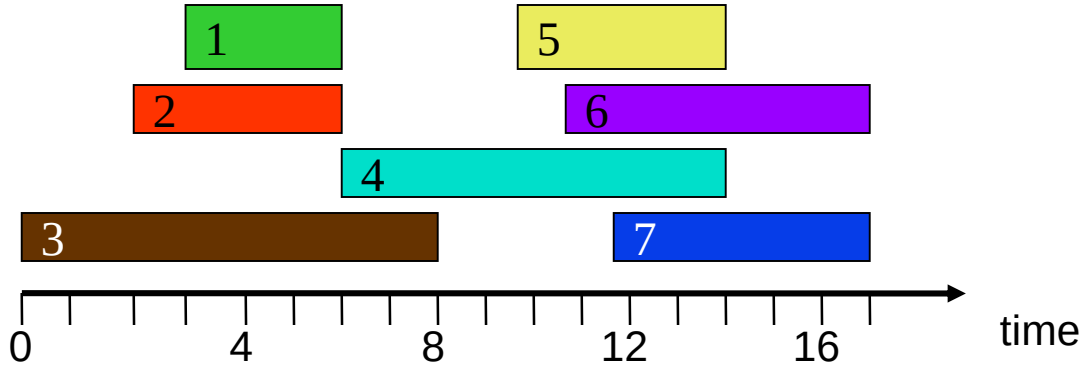
# DVFS: Real-Time Offline Scheduling on One Processor

- Let us model a set of independent tasks as follows:
  - We suppose that a task $v_i \epsilon V$
    - 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 Optimal DVFS Algorithm for Offline Scheduling




3,6,5

2,6,3

0,8,2

6,14,6

10,14,6

11,17,2

12,17,2

$a_i$,$d_i$,$c_i$

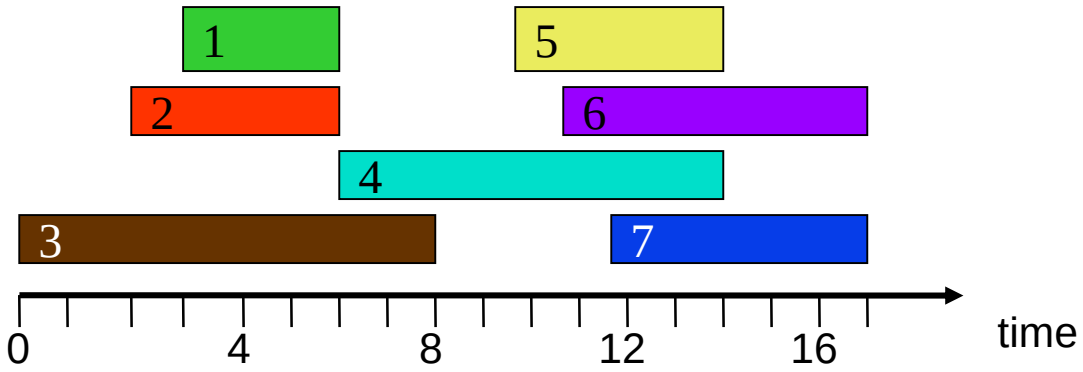
- Define ***intensity*** G([*z, z'*]) in some time interval [*z, z'*]:
  - average accumulated execution time of all tasks that have arrival and deadline in [*z, z'*] relative to the length of the interval *z'-z*

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

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

# YDS Optimal DVFS 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

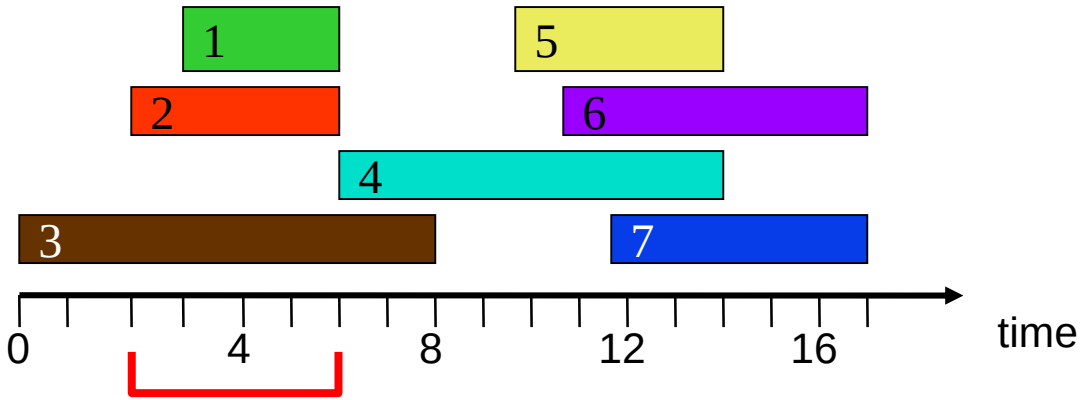
11,17,2

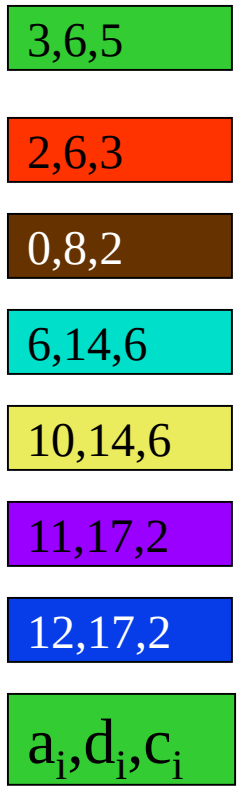
12,17,2

$a_i,d_i,c_i$

# YDS Optimal DVFS 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

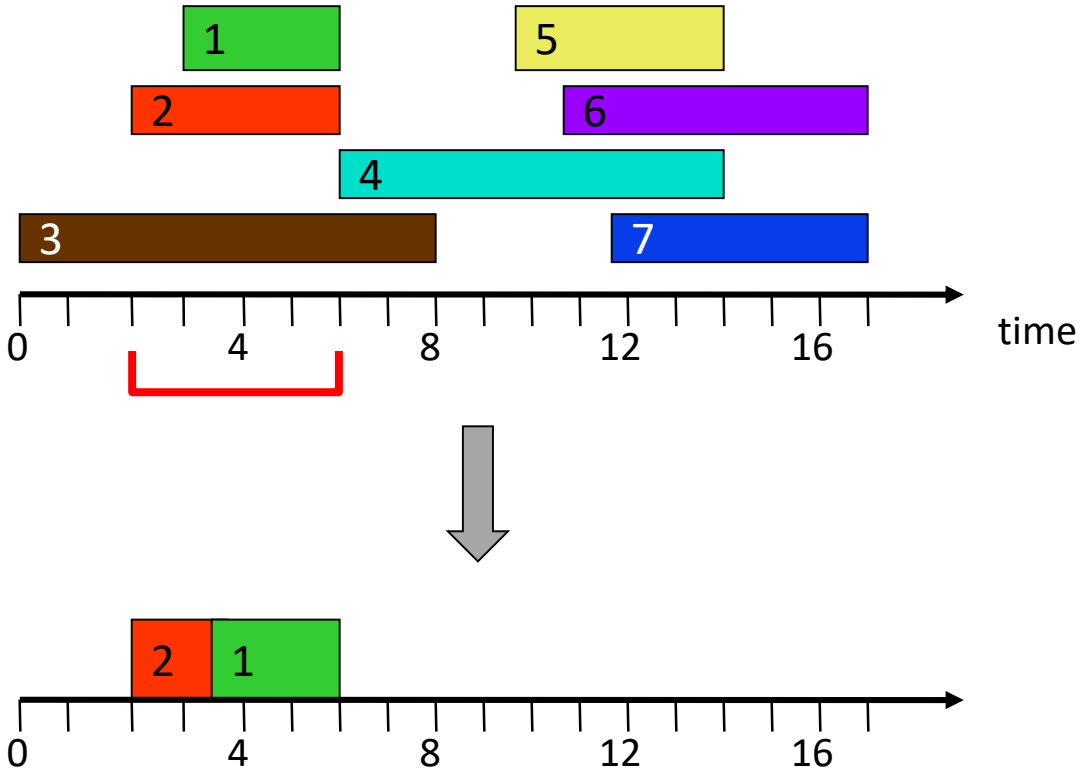
G([3,6]) =5/3, G([3,14]) = (5+6+6)/(14-3) = 17/11, G([3,17])=(5+6+6+2+2)/14=21/14

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

# YDS Optimal DVFS 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.




3,6,5

2,6,3

0,8,2

6,14,6

10,14,6

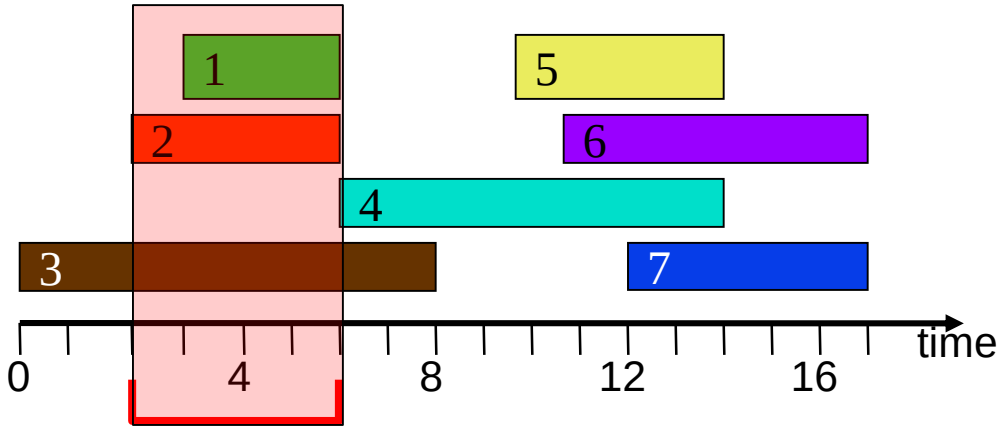
11,17,2

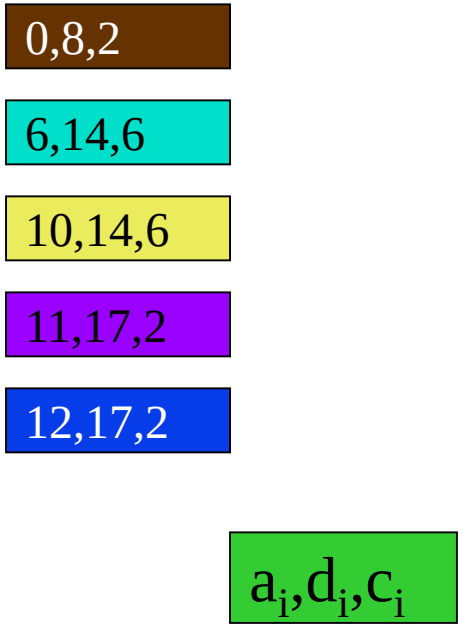
12,17,2

$a_i,d_i,c_i$

# YDS Optimal DVFS Algorithm for Offline Scheduling

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

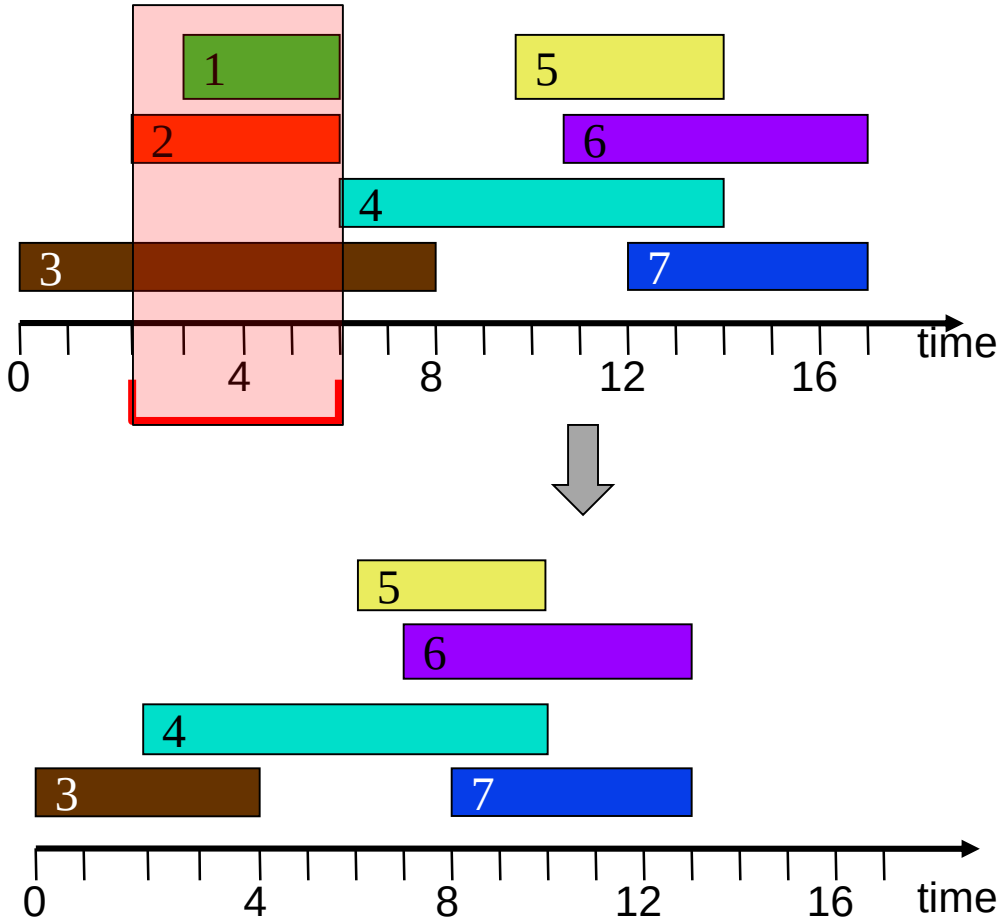


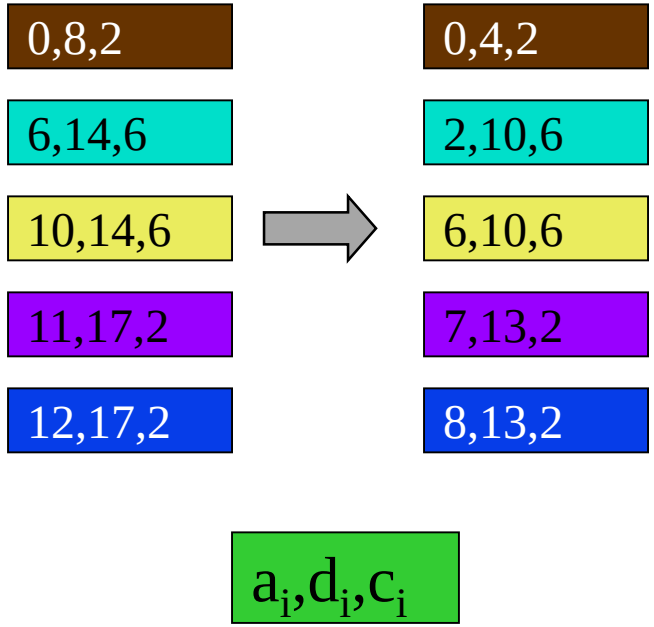




$a_i,d_i,c_i$

# YDS Optimal DVFS Algorithm for Offline Scheduling

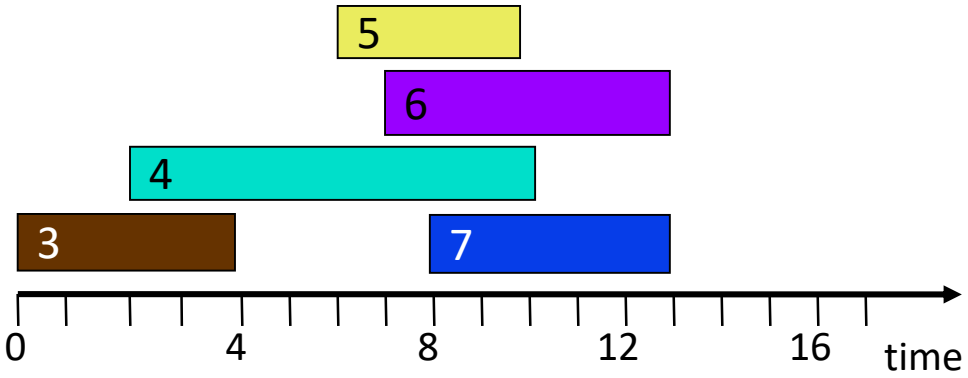
***Step 2:*** Adjust the arrival times and deadlines by excluding the possibility to execute at the previous critical intervals.



# YDS Optimal DVFS 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

0,4,2

2,10,6

6,10,6

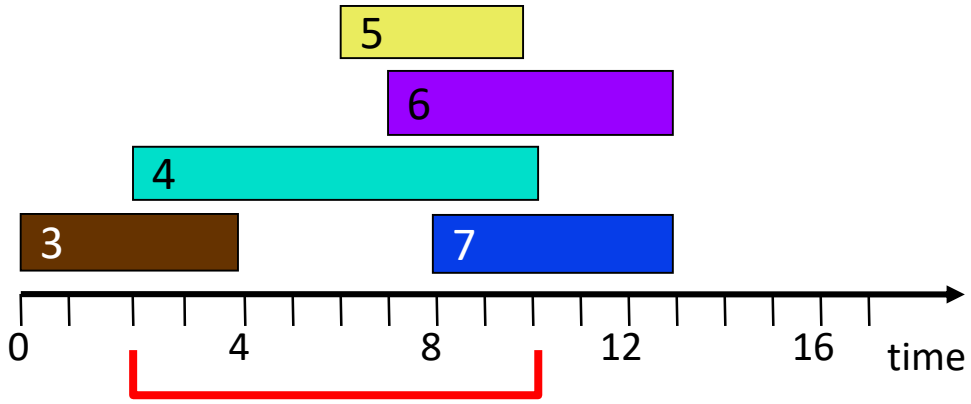
7,13,2

8,13,2

$a_i, d_i, c_i$

# YDS Optimal DVFS 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

0,4,2

2,10,6

6,10,6

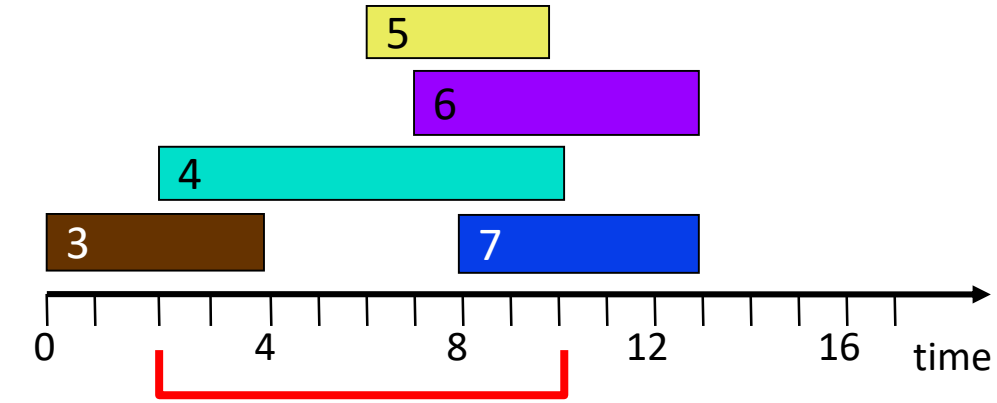
7,13,2

8,13,2

$a_i, d_i, c_i$

# YDS Optimal DVFS Algorithm for Offline Scheduling

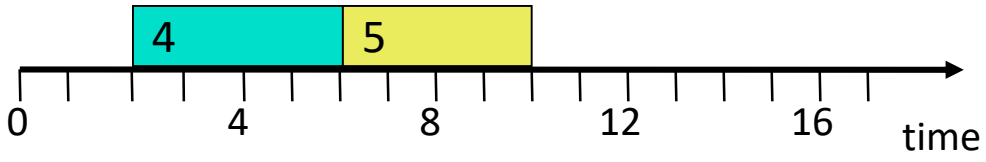
***Step 3:*** Run the algorithm for the revised input again




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




0,4,2

2,10,6

6,10,6

7,13,2

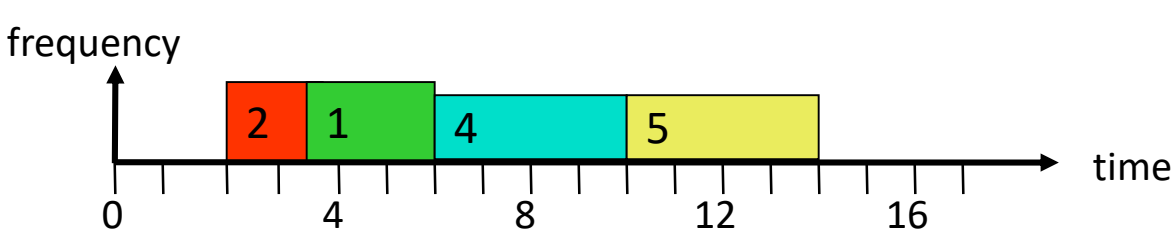
8,13,2

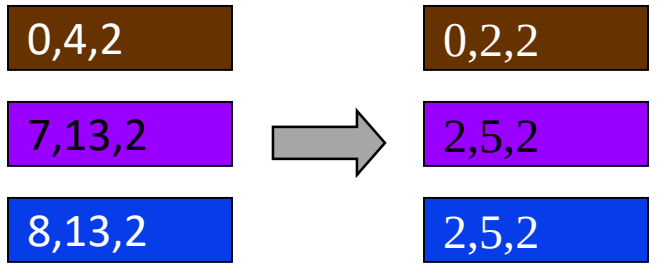
$a_i, d_i, c_i$

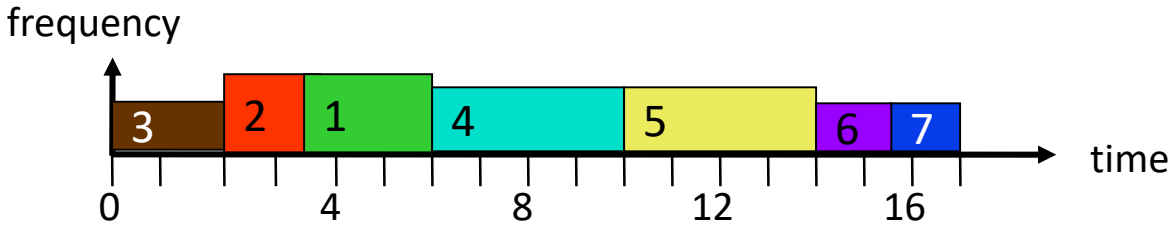
# YDS Optimal DVFS Algorithm for Offline Scheduling

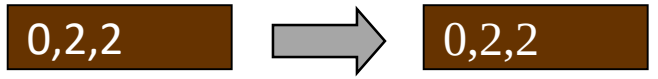
***Step 3:*** Run the algorithm for the revised input again

***Step 4:*** Put pieces together



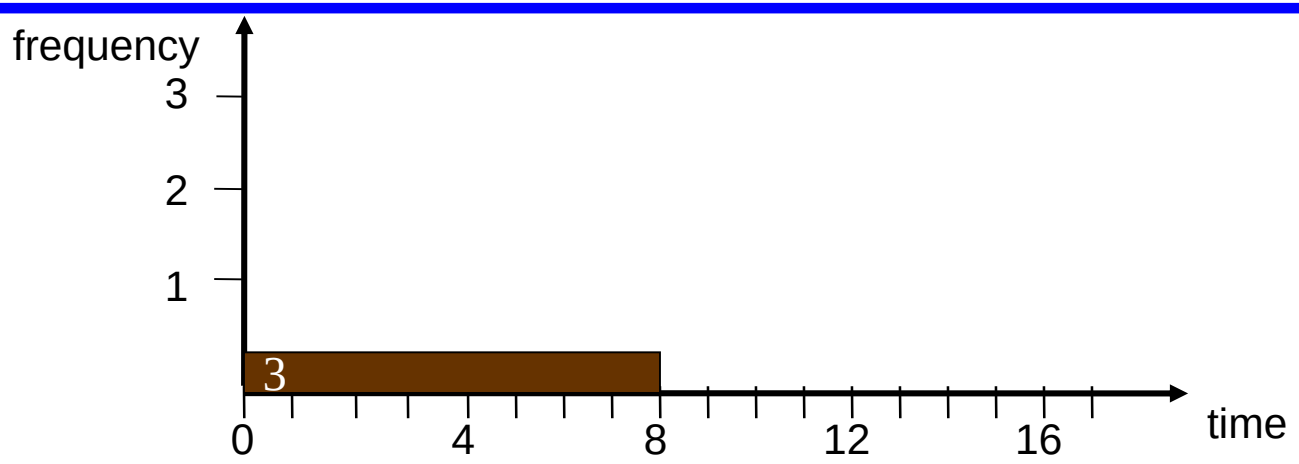




| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ |
|---|---|---|---|---|---|---|---|
| frequency | 2 | 2 | 1 | 1.5 | 1.5 | 4/3 | 4/3 |

# YDS Optimal DVFS Algorithm for Online Scheduling



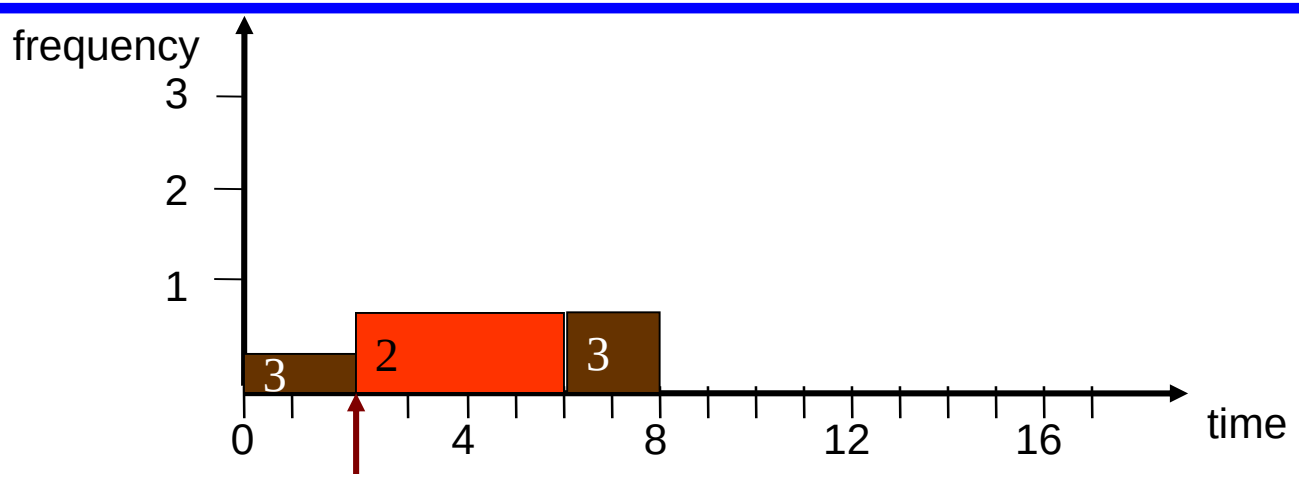

0,8,2

**Continuously update to the best schedule for all arrived tasks:**

Time 0: task $v_3$ is executed at 2/8

$a_i,d_i,c_i$

# YDS Optimal DVFS Algorithm for Online Scheduling




2,6,3

0,8,2

**Continuously update to the best schedule for all arrived tasks:**

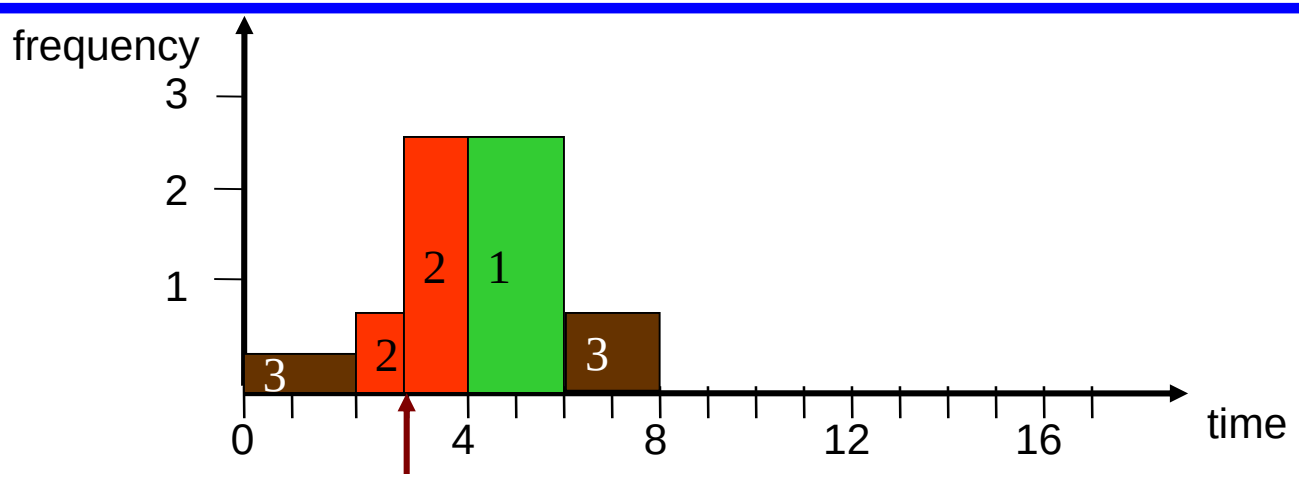
Time 0: task $v_3$ is executed at 2/8

Time 2: task $v_2$ arrives

- G([2,6]) = ¾, G([2,8]) = 4.5/6=3/4 => execute $v_8$ , $v_2$ at ¾

$a_i,d_i,c_i$

# YDS Optimal DVFS Algorithm for Online Scheduling




3,6,5

2,6,3

0,8,2

**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

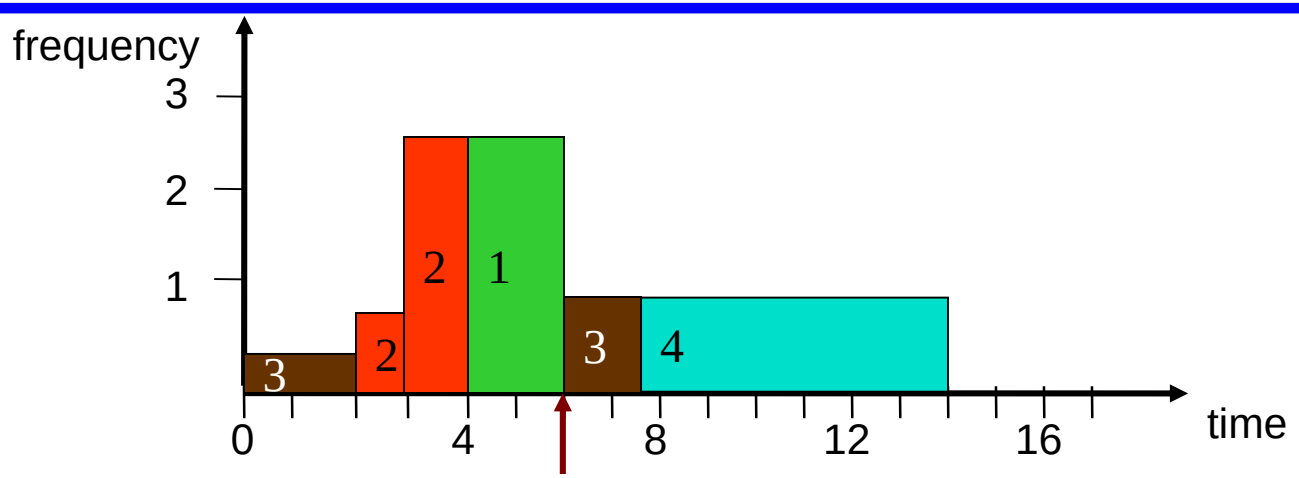
- G([2,6]) = ¾, G([2,8]) = 4.5/6=3/4 => execute $v_8$ , $v_2$ at ¾

Time 3: task $v_1$ arrives

- G([3,6]) = (5+3-3/4)/3=29/12, G([3,8]) < G([3,6]) => execute $v_2$ and $v_1$ at 29/12

$a_i,d_i,c_i$

# YDS Optimal DVFS Algorithm for Online Scheduling




3,6,5

2,6,3

0,8,2

6,14,6

**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]) = ¾, G([2,8]) = 4.5/6=3/4 => execute $v_8$ , $v_2$ at ¾

Time 3: task $v_1$ arrives

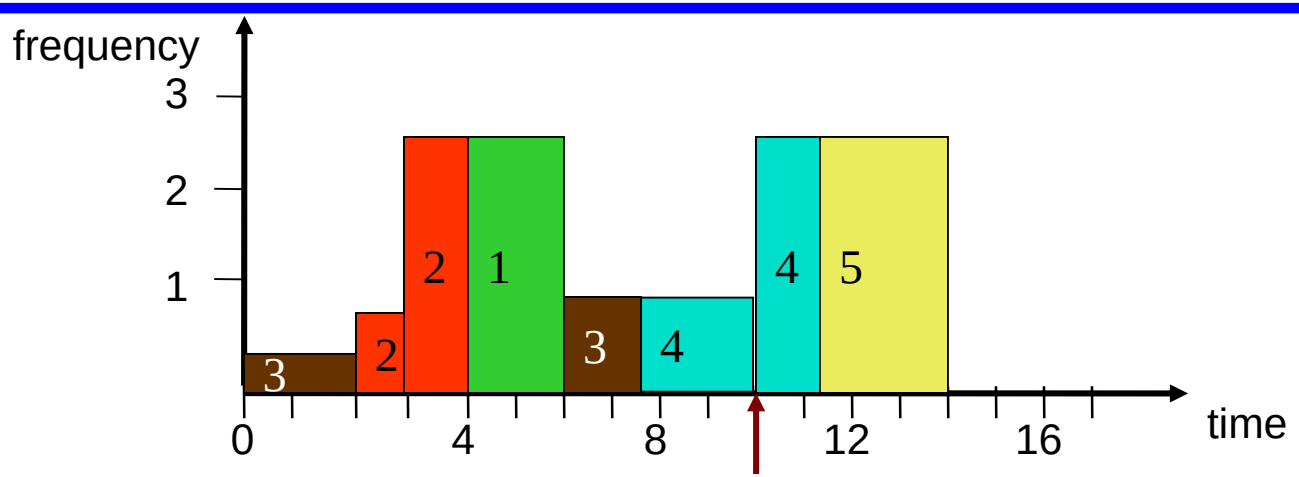
- G([3,6]) = (5+3-3/4)/3=29/12, G([3,8]) < G([3,6]) => execute $v_2$ and $v_1$ at 29/12

Time 6: task $v_4$ arrives

- G([6,8]) = 1.5/2, G([6,14]) = 7.5/8 => execute $v_3$ and $v_4$ at 15/16

$a_i,d_i,c_i$

# YDS Optimal DVFS Algorithm for Online Scheduling




3,6,5

2,6,3

0,8,2

6,14,6

10,14,6

**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]) = ¾, G([2,8]) = 4.5/6=3/4 => execute $v_8$ , $v_2$ at ¾

Time 3: task $v_1$ arrives

- G([3,6]) = (5+3-3/4)/3=29/12, G([3,8]) < G([3,6]) => execute $v_2$ and $v_1$ at 29/12

Time 6: task $v_4$ arrives

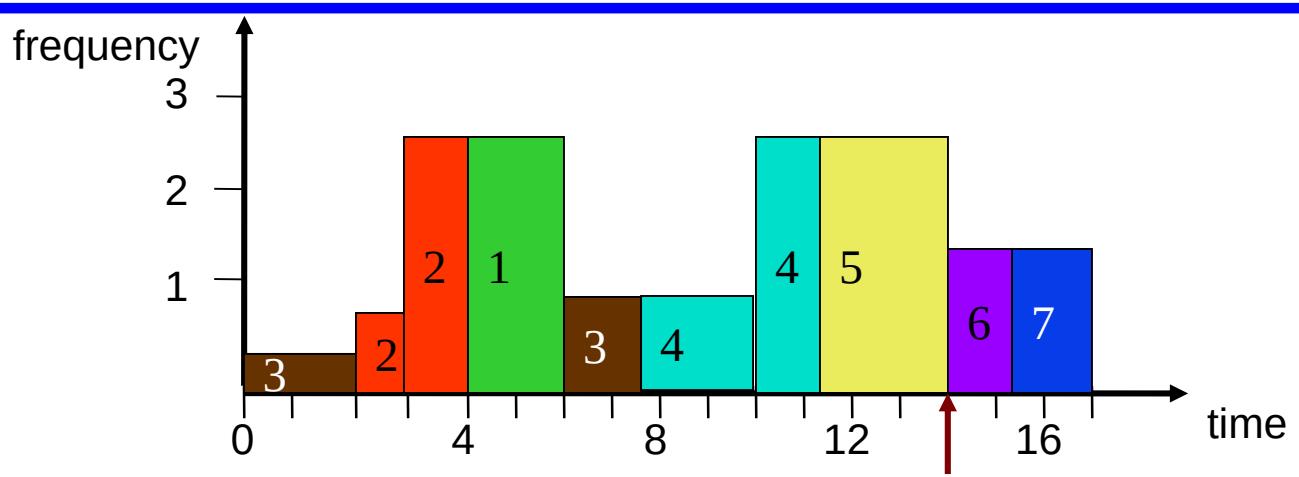
- G([6,8]) = 1.5/2, G([6,14]) = 7.5/8 => execute $v_3$ and $v_4$ at 15/16

Time 10: task $v_5$ arrives

- G([10,14]) = 39/16 => execute $v_4$ and $v_5$ at 39/16

$a_i,d_i,c_i$

# YDS Optimal DVFS Algorithm for Online Scheduling




3,6,5

2,6,3

0,8,2

6,14,6

10,14,6

11,17,2

12,17,2

$a_i,d_i,c_i$

**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]) = ¾, G([2,8]) = 4.5/6=3/4 => execute $v_8$ , $v_2$ at ¾

Time 3: task $v_1$ arrives

- G([3,6]) = (5+3-3/4)/3=29/12, G([3,8]) < G([3,6]) => 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 => execute $v_3$ and $v_4$ at 15/16

Time 10: task $v_5$ arrives

- G([10,14]) = 39/16 => 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 => execute $v_6$ and $v_7$ at 4/3

# Remarks on the 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.

# Dynamic Power Management

# Dynamic Power Management (DPM)

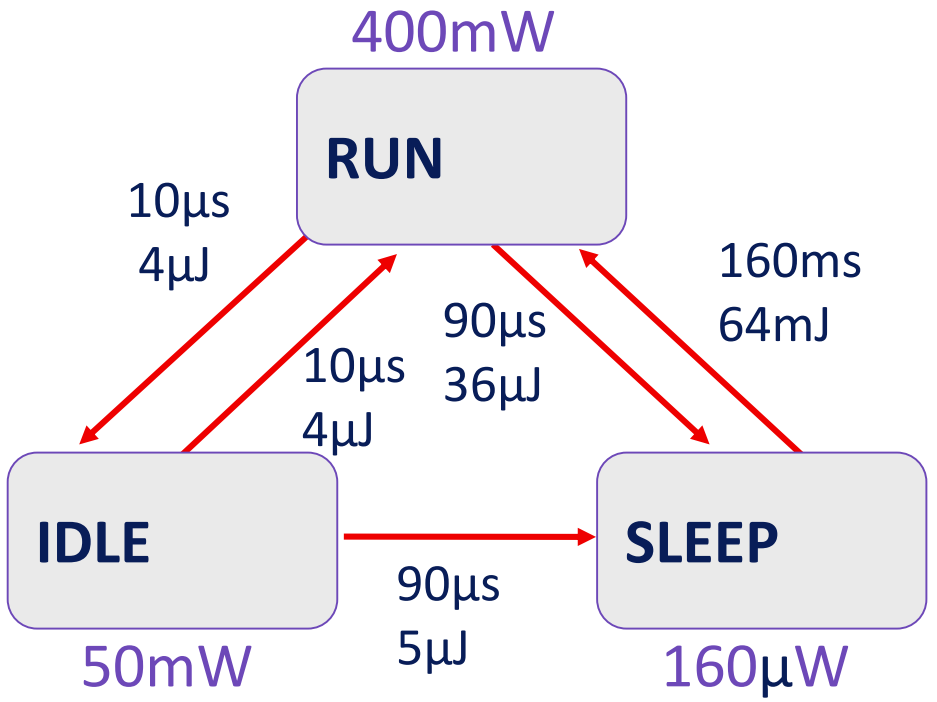
- Dynamic power management tries to assign optimal **power saving states during program execution**
- DPM requires hardware and software support

Example: StrongARM SA1100

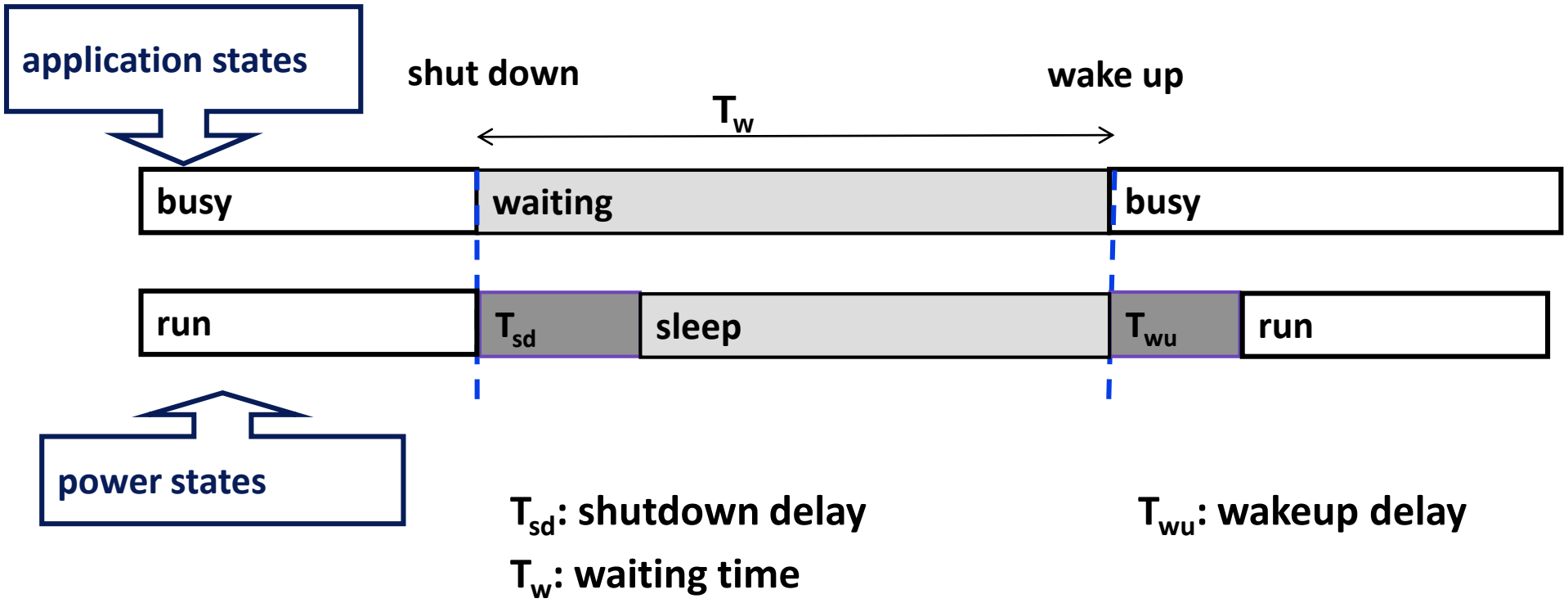
***RUN***: operational

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

***SLEEP***: Shutdown of on-chip activity



# Dynamic Power Management (DPM)

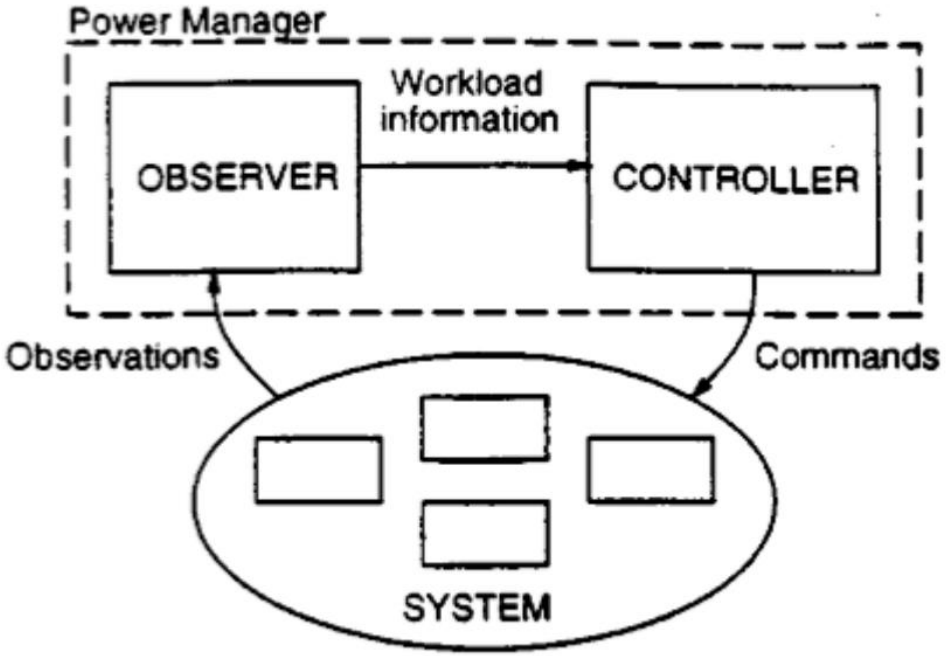



Desired: Shutdown only during long waiting times. This leads to a tradeoff between energy saving and overhead.

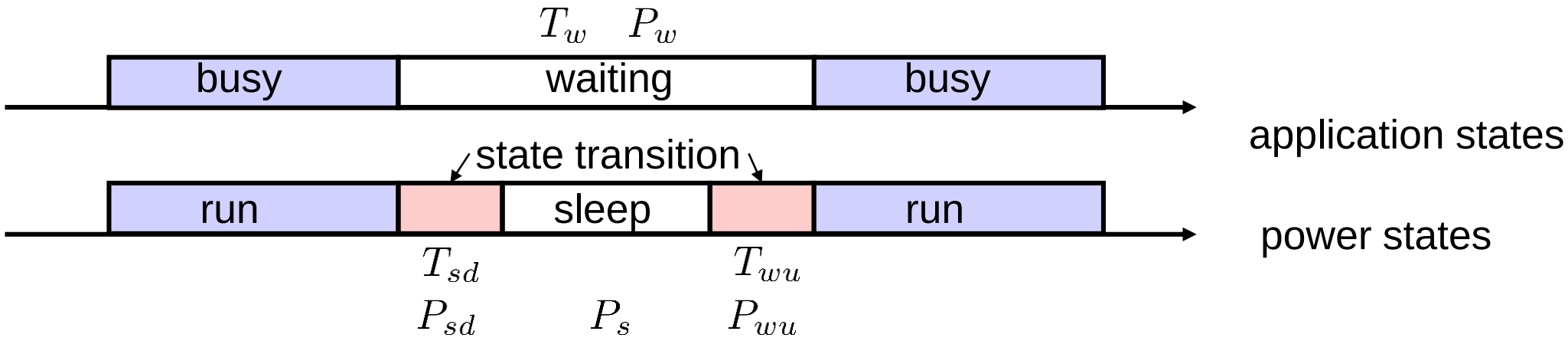
# Break-Even Time

***Definition:*** The minimum waiting time required to compensate the cost of entering an inactive (sleep) state.

- Enter an inactive state is beneficial only if the waiting time is longer than the break-even time
- Assumptions for the calculation:
  - No performance penalty is tolerated.
  - An ideal power manager that has the *full* knowledge of the future workload trace. On the previous slide, we supposed that the power manager has *no* knowledge about the future.



# Break-Even Time




Scenario 1 (no transition): $E_1 = T_w \cdot P_w$

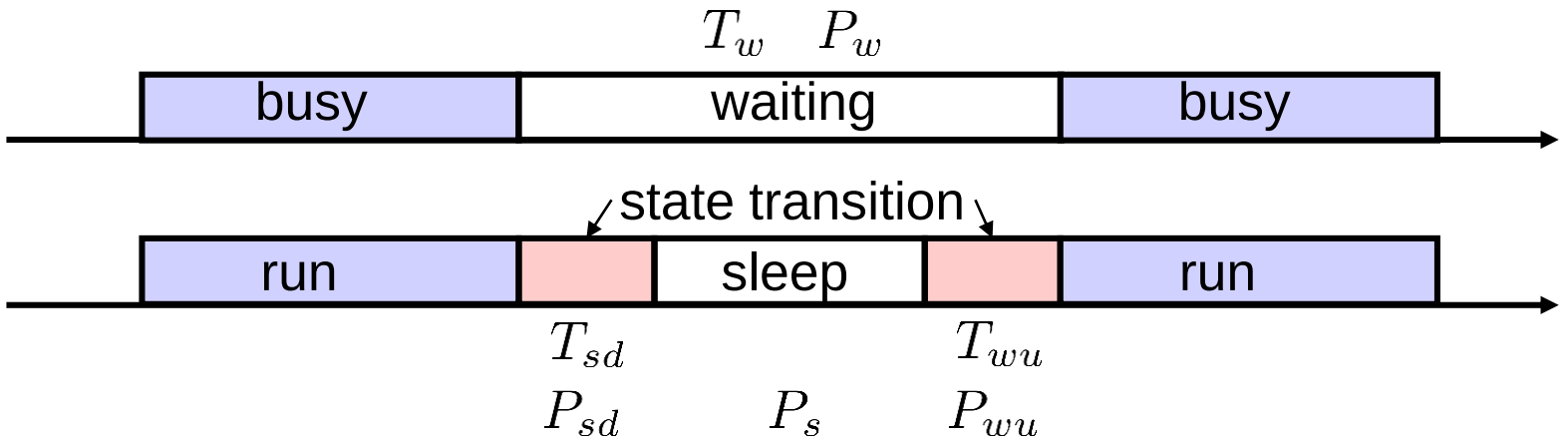
Scenario 2 (state transition): $E_2 = T_{sd} \cdot P_{sd} + T_{wu} \cdot P_{wu} + (T_w - T_{sd} - T_{wu}) \cdot P_s$

Break-even time: Limit for $T_w$ such that $E_2 \leq E_1$

Break-even constraint: $T_w \geq \dfrac{T_{sd} \cdot (P_{sd} - P_s) + T_{wu} \cdot (P_{wu} - P_s)}{P_w - P_s}$ ← break-even time

Time constraint: $T_w \geq T_{sd} + T_{wu}$

# Break-Even Time




Scenario 1 (no transition): $E_1 = T_w \cdot P_w$

Scenario 2 (state transition): $E_2 = T_{sd} \cdot P_{sd} + T_{wu} \cdot P_{wu} + (T_w - T_{sd} - T_{wu}) \cdot P_s$

Break-even time: Limit for $T_w$ such that $E_2 \leq E_1$

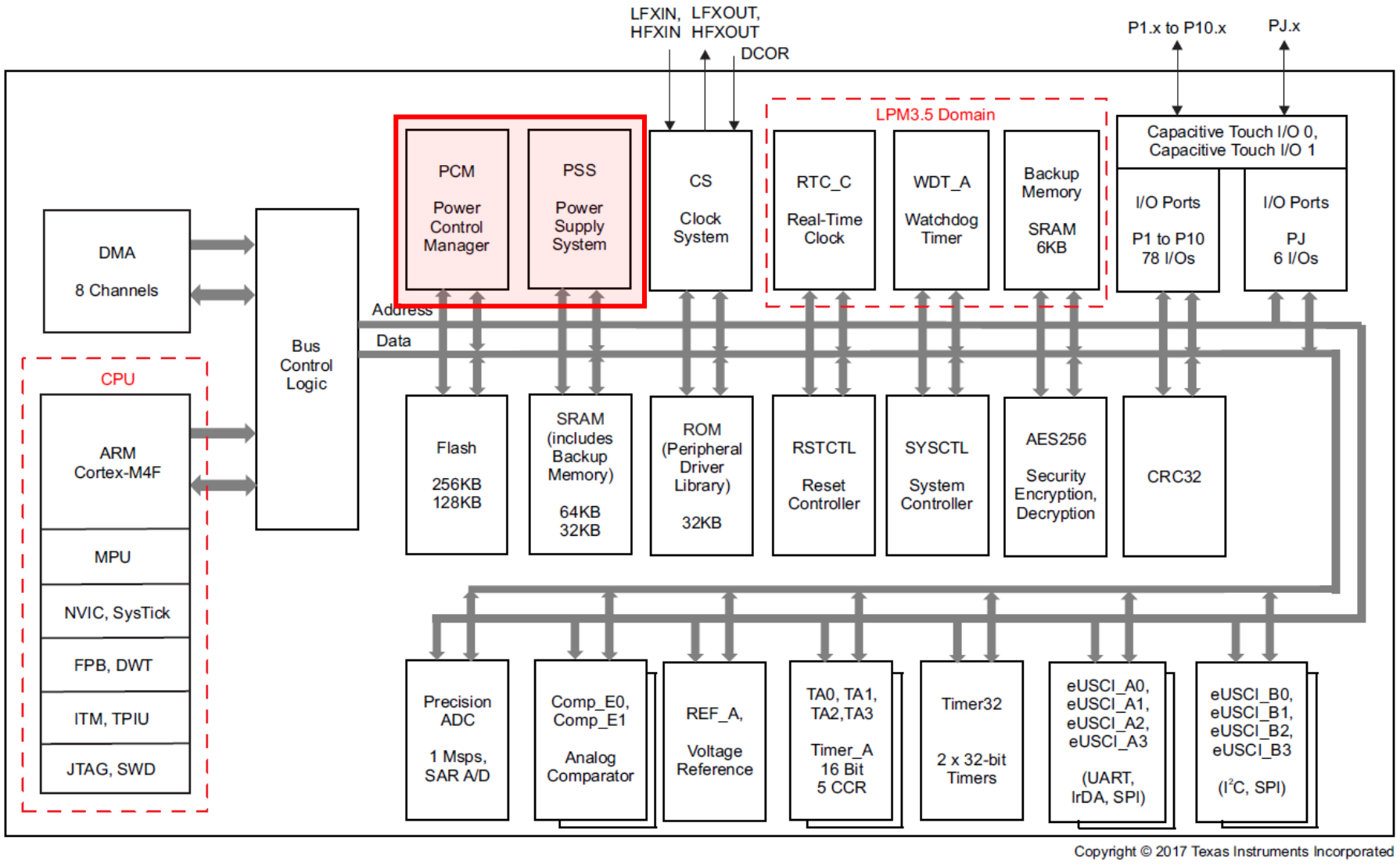
Break-even constraint: $T_w \geq \dfrac{T_{sd} \cdot (P_{sd} - P_s) + T_{wu} \cdot (P_{wu} - P_s)}{P_w - P_s}$

Time constraint: $T_w \geq T_{sd} + T_{wu}$

remove, if power manager has no knowledge about future

break-even time

# Power Modes in MSP432 (Lab)




The MSP432 has one active mode in 6 different configurations which all allow for execution of code.

It has 5 major low power modes (LP0, LP3, LP4, LP3.5, LP4.5), some of them can be in one of several configurations.

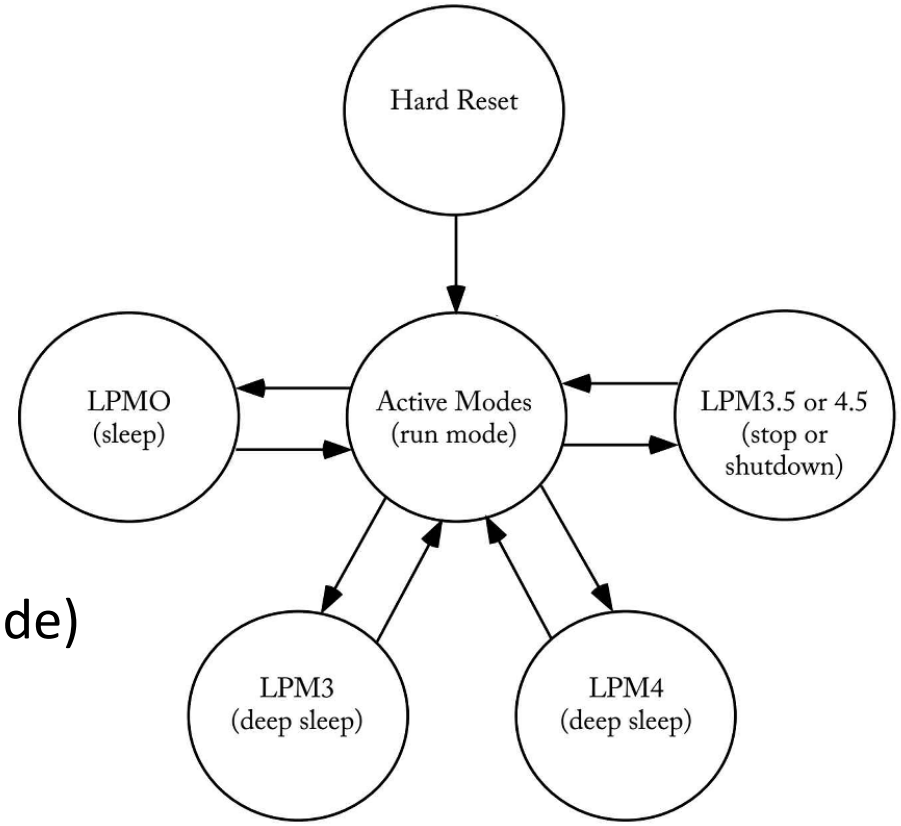
In total, the MSP432 can be in 18 different low power configurations.

active mode (32MHz): 6 - 15 mW ; low power mode (LP4): 1.5 – 2.1 µW

# Power Modes in MSP432 (Lab)

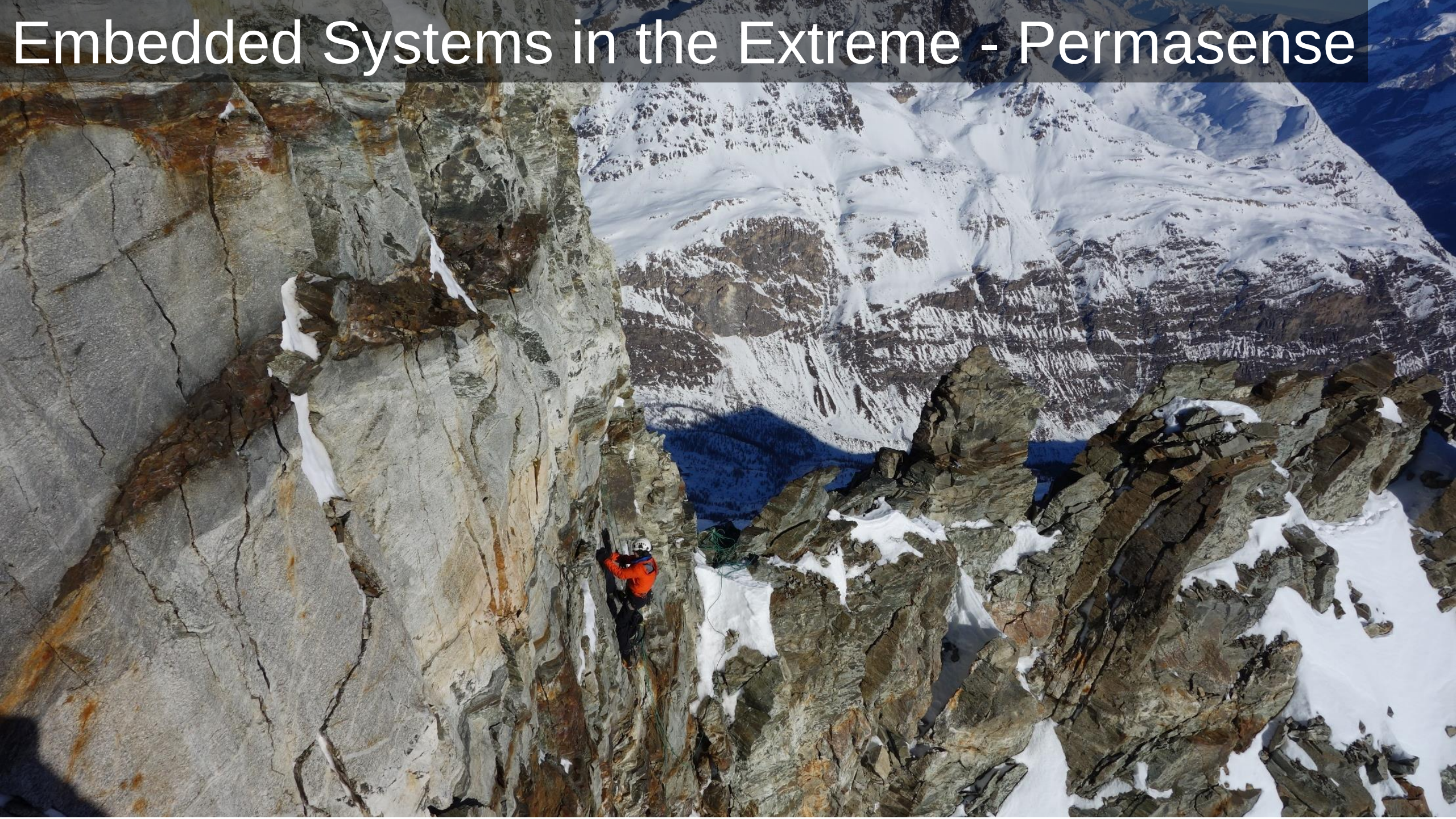
- Transition between modes can be handled using C-level interfaces to the power control manger.

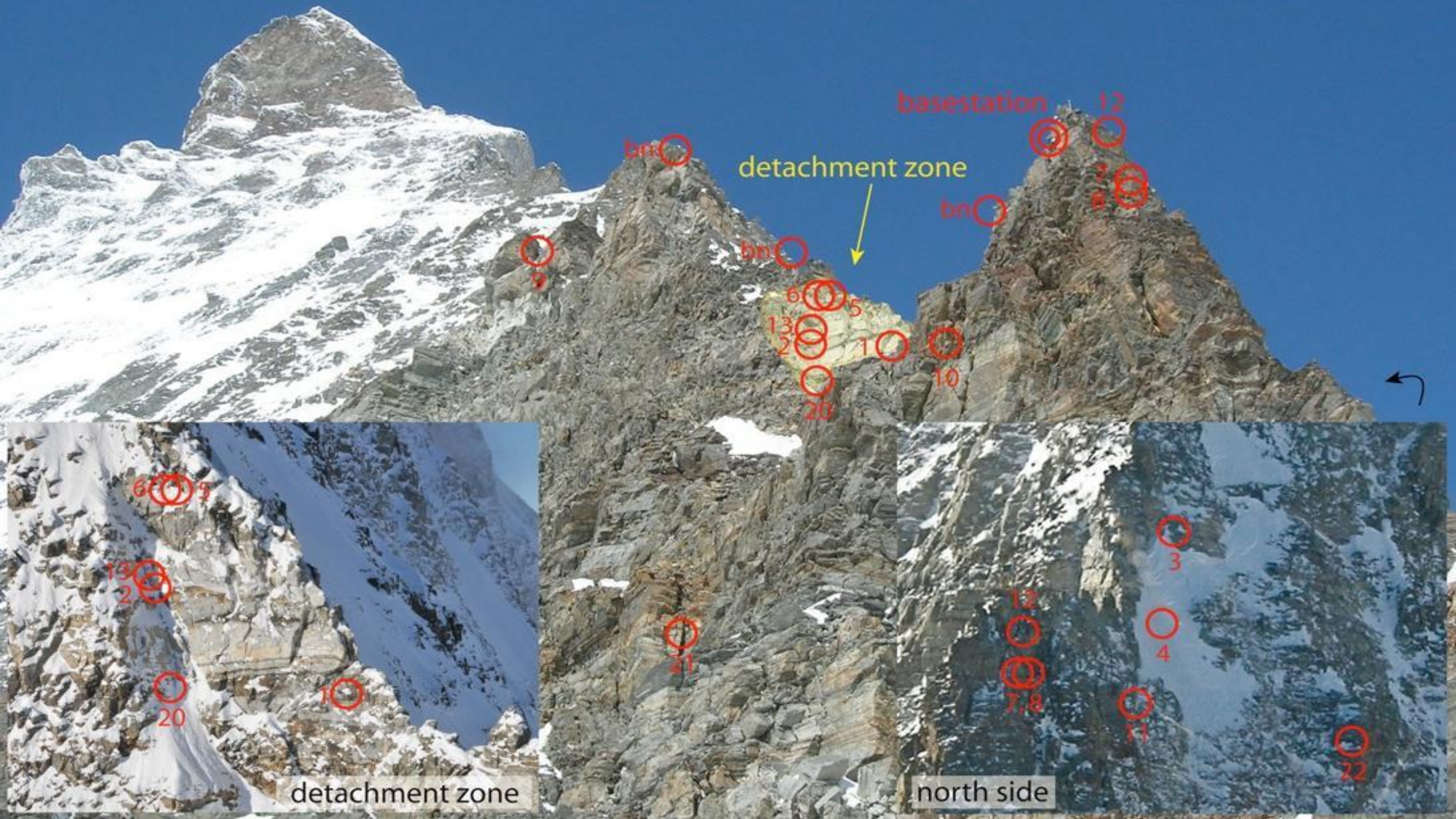
- Examples of interface functions:
  - uint8_t PCM_getPowerState (void)
  - bool PCM_gotoLPM0 (void)
  - bool PCM_gotoLPM3 (void)
  - bool PCM_gotoLPM4 (void)
  - bool PCM_shutdownDevice (uint32_t shutdownMode)



# Battery-Operated Systems and Energy Harvesting

# Embedded Systems in the Extreme - Permasense

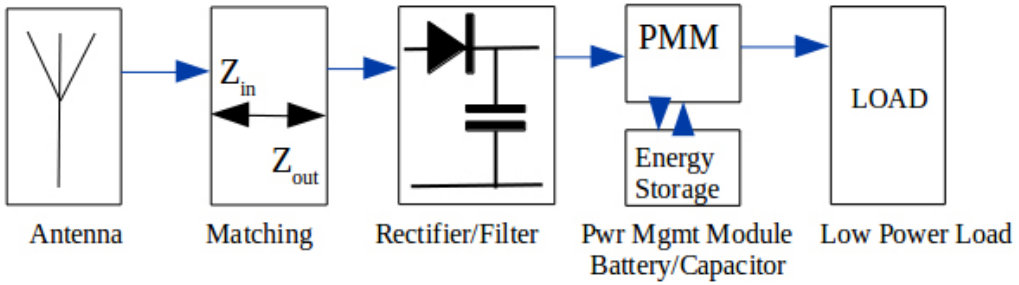



basestation
12
bm
detachment zone
bm
bm
9
6
5
13
2
1
10
20
21
3
4
12
7,8
11
22
detachment zone
north side

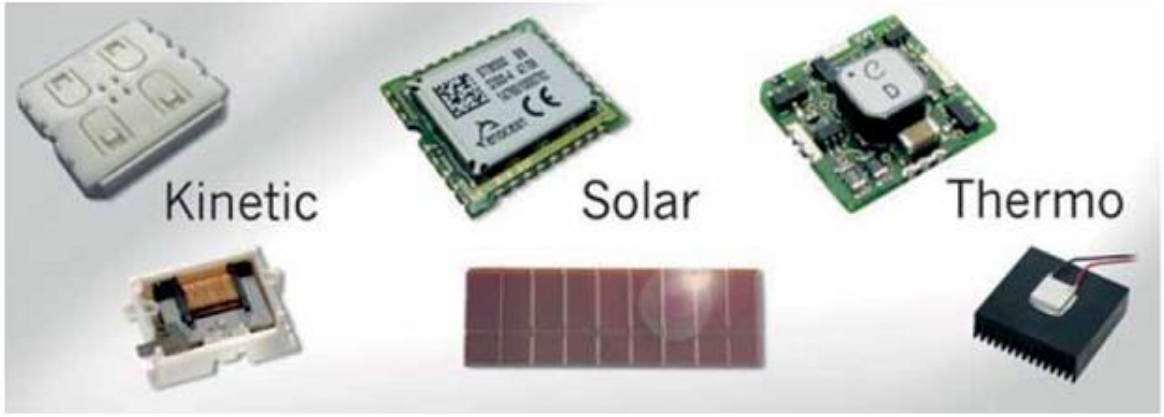
# Reasons for Battery-Operated Devices and Harvesting

- Battery operation:
  - no continuous power source available
  - mobility

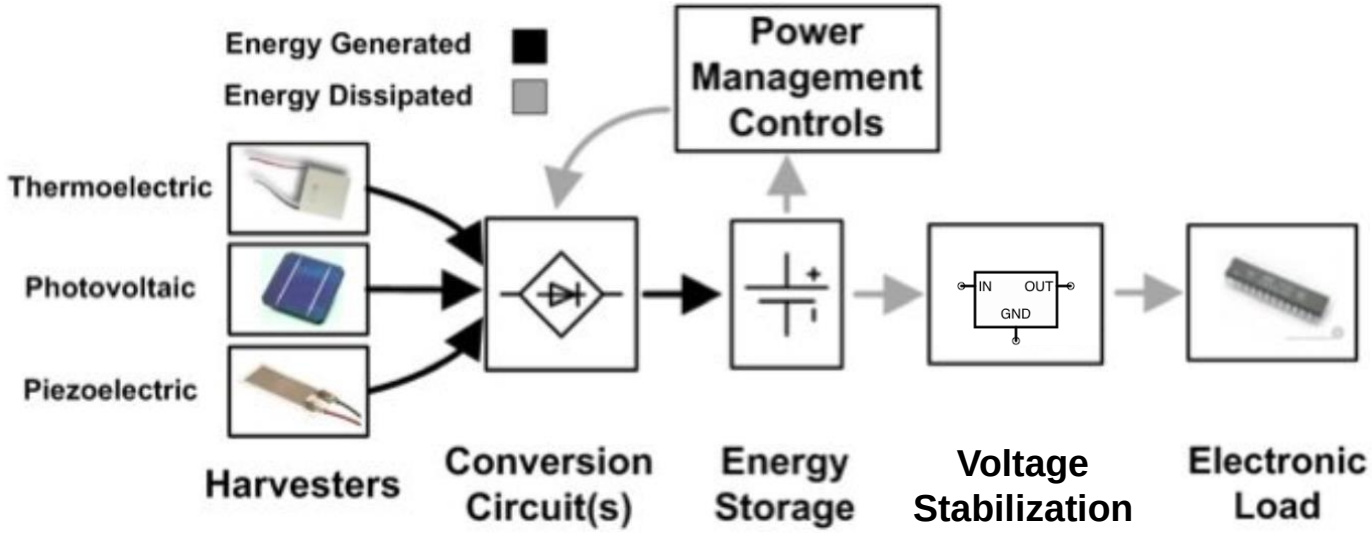

- Energy harvesting:
  - prolong lifetime of battery-operated devices
  - infinite lifetime using rechargeable batteries
  - autonomous operation




radio frequency (RF) harvesting



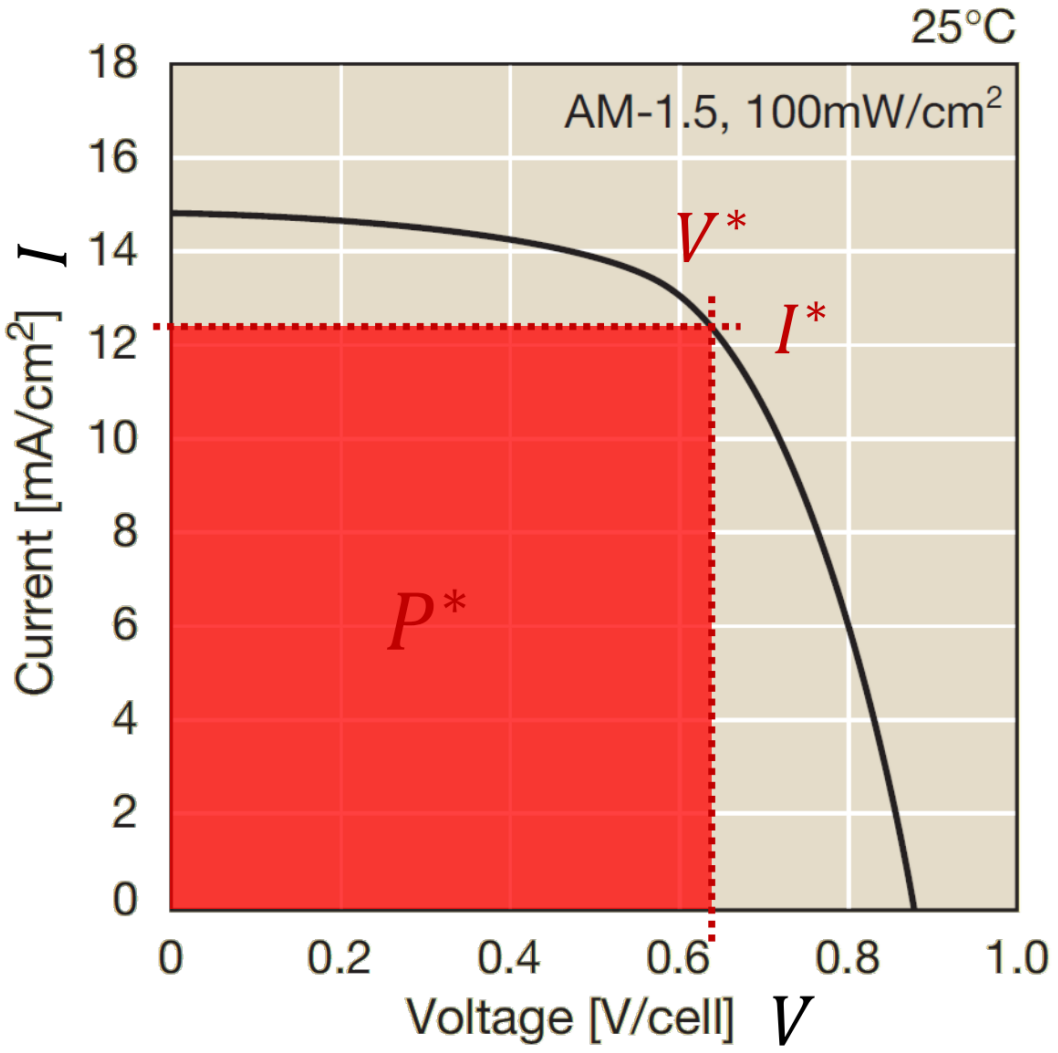
# Typical Power Circuitry – Power Point Tracking




power point tracking / impedance matching; conversion to voltage of energy storage

rechargeable battery or supercapacitor

# Solar Panel Characteristics

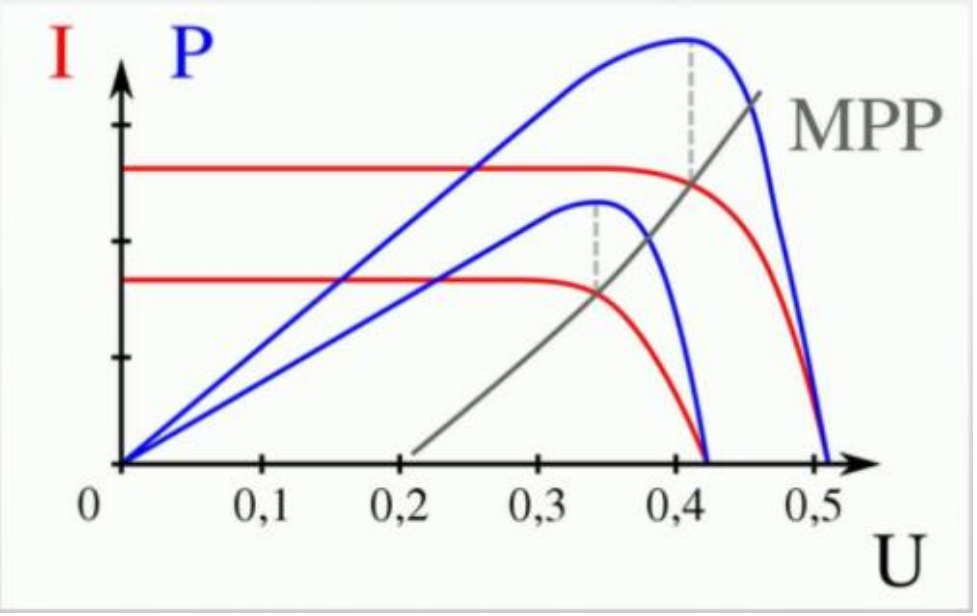



- Variable output power
  - Illuminance level
  - Electrical operation point
  - (Temperature, age, …)
- I-V-Characteristics
  - Non-linear
  - Dependent on ambient
- Maximum Power Point Tracking
  - Dynamic algorithm to find $P^*$

Diagram: *Amorton Amorphous Silicon Solar Cells Datasheet,* © Panasonic

# Typical Power Circuitry – Maximum Power Point Tracking

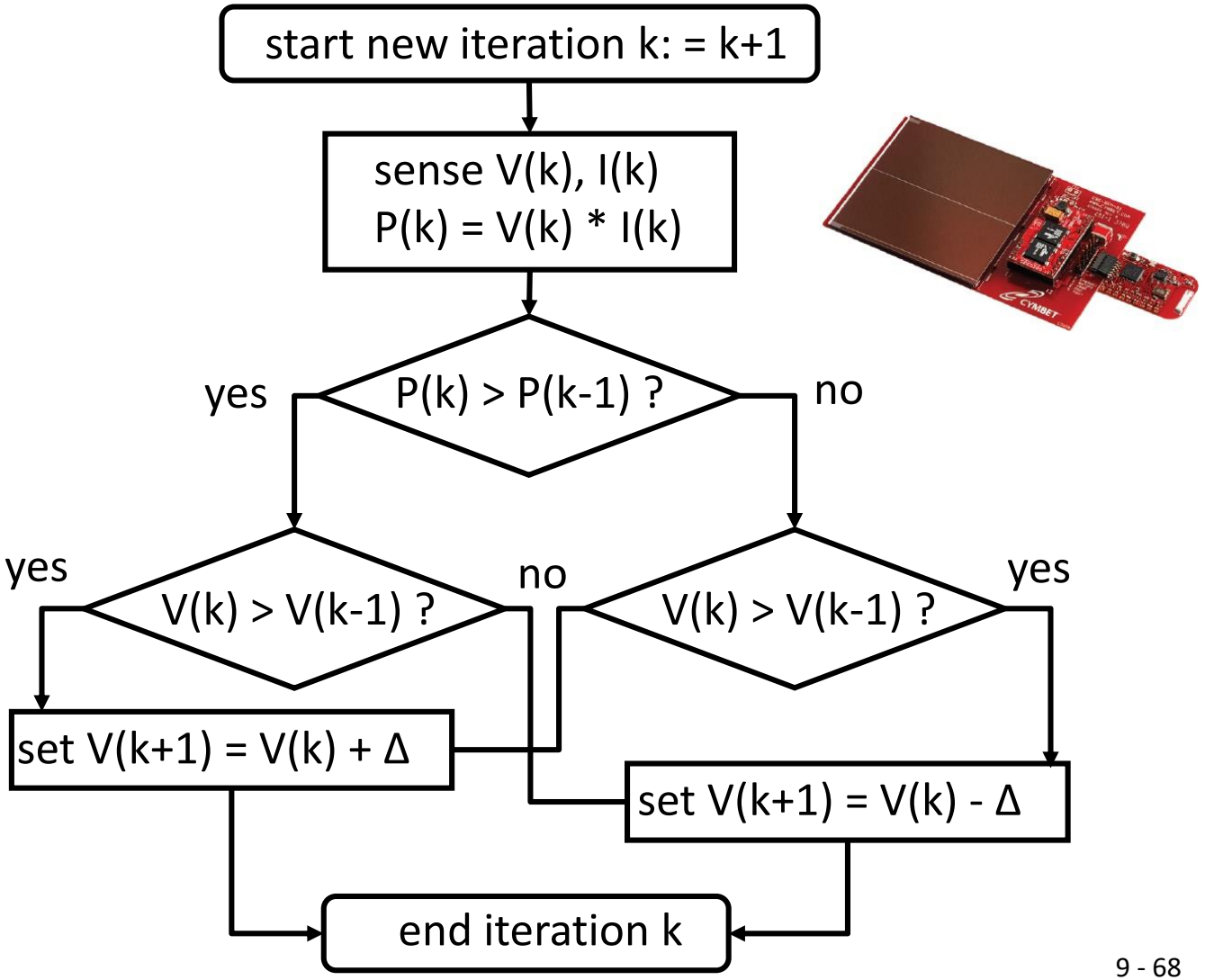
U/I curves of a typical solar cell:



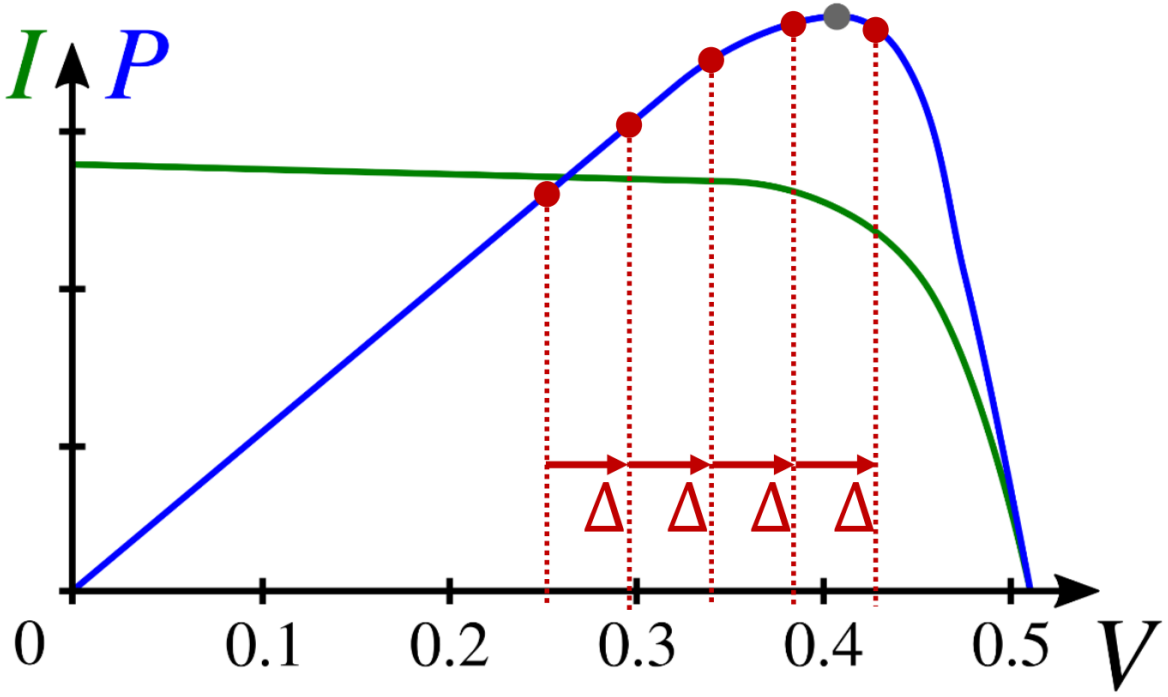

**red:** current for different light intensities
**blue:** power for different light intensities
**grey:** maximal power

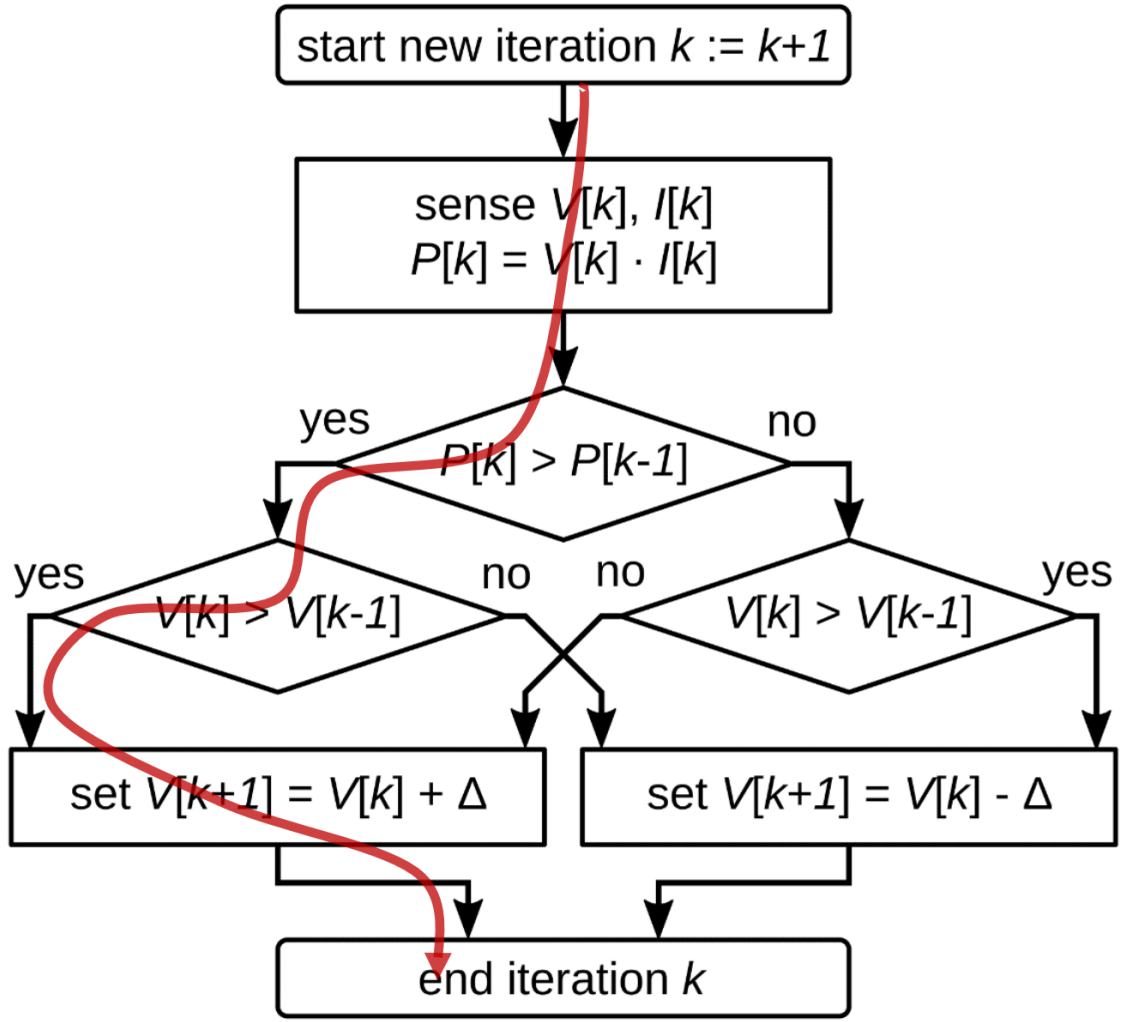
**tracking**: determine optimal impedance seen by the solar panel

simple tracking algorithm (assume constant illumination) :

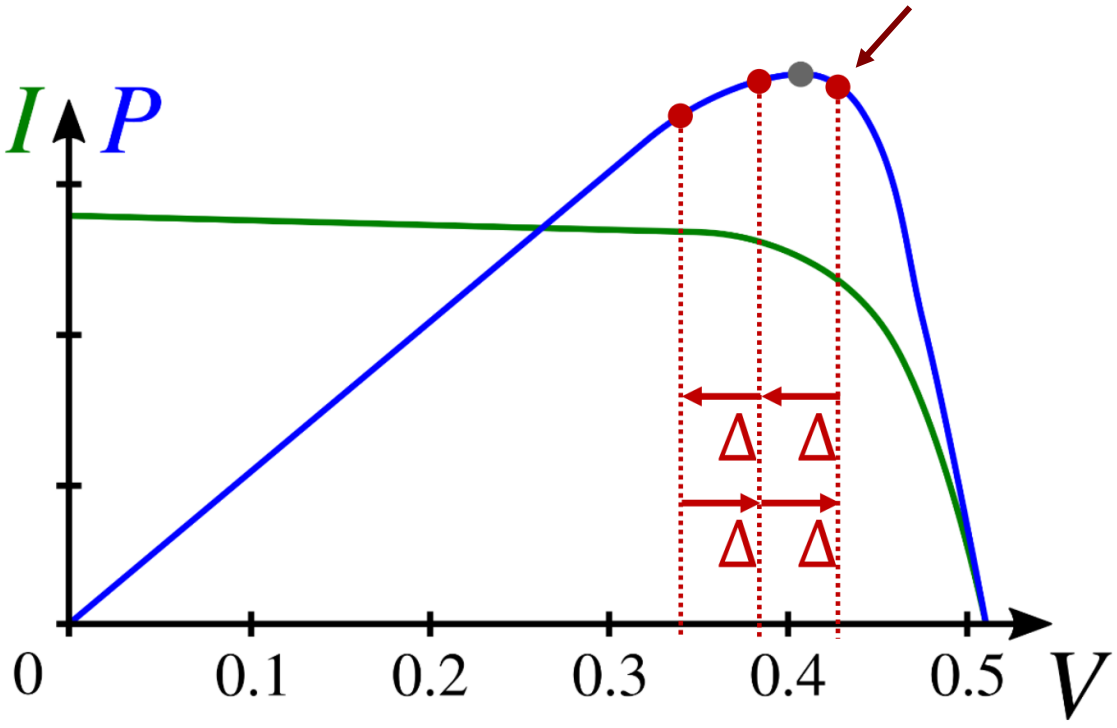


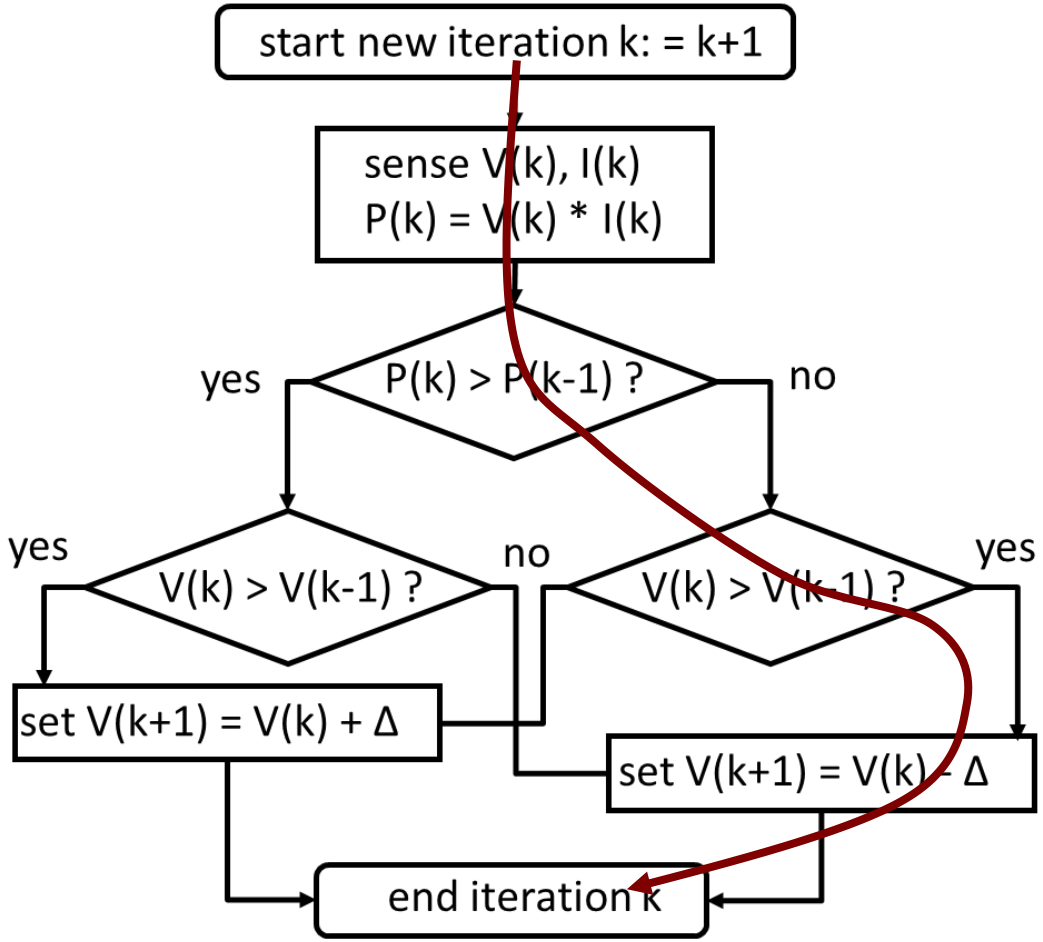
# Maximal Power Point Tracking



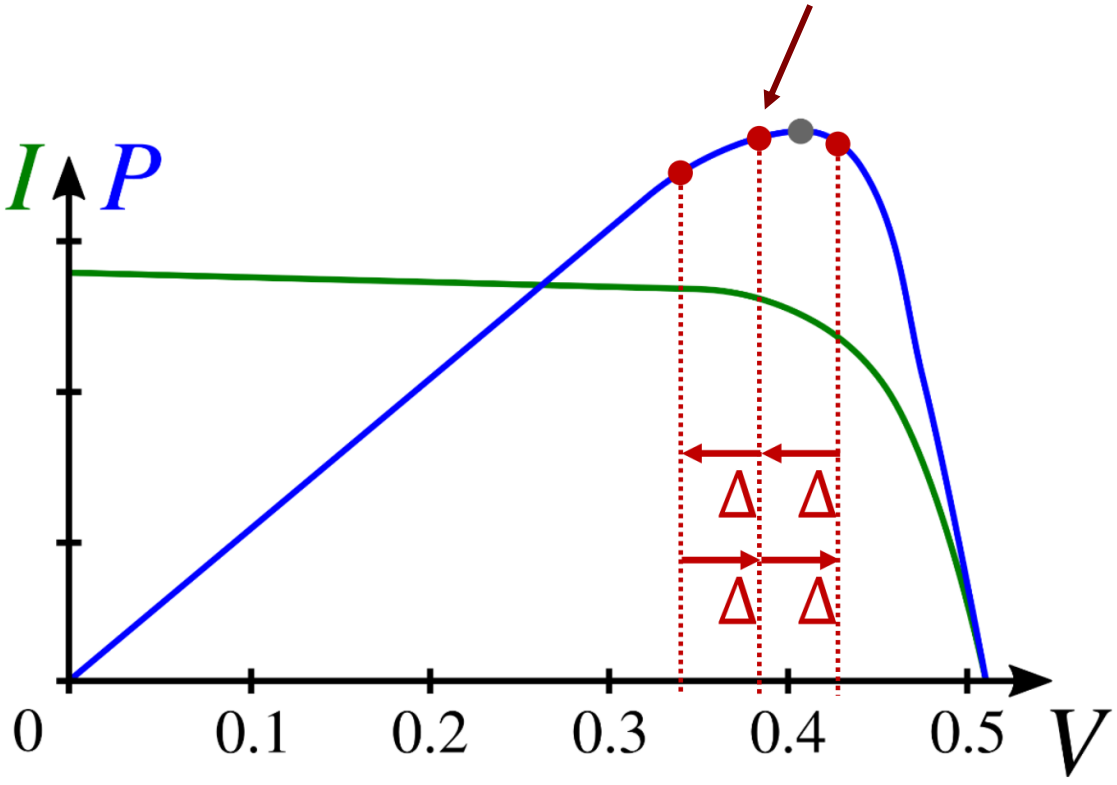



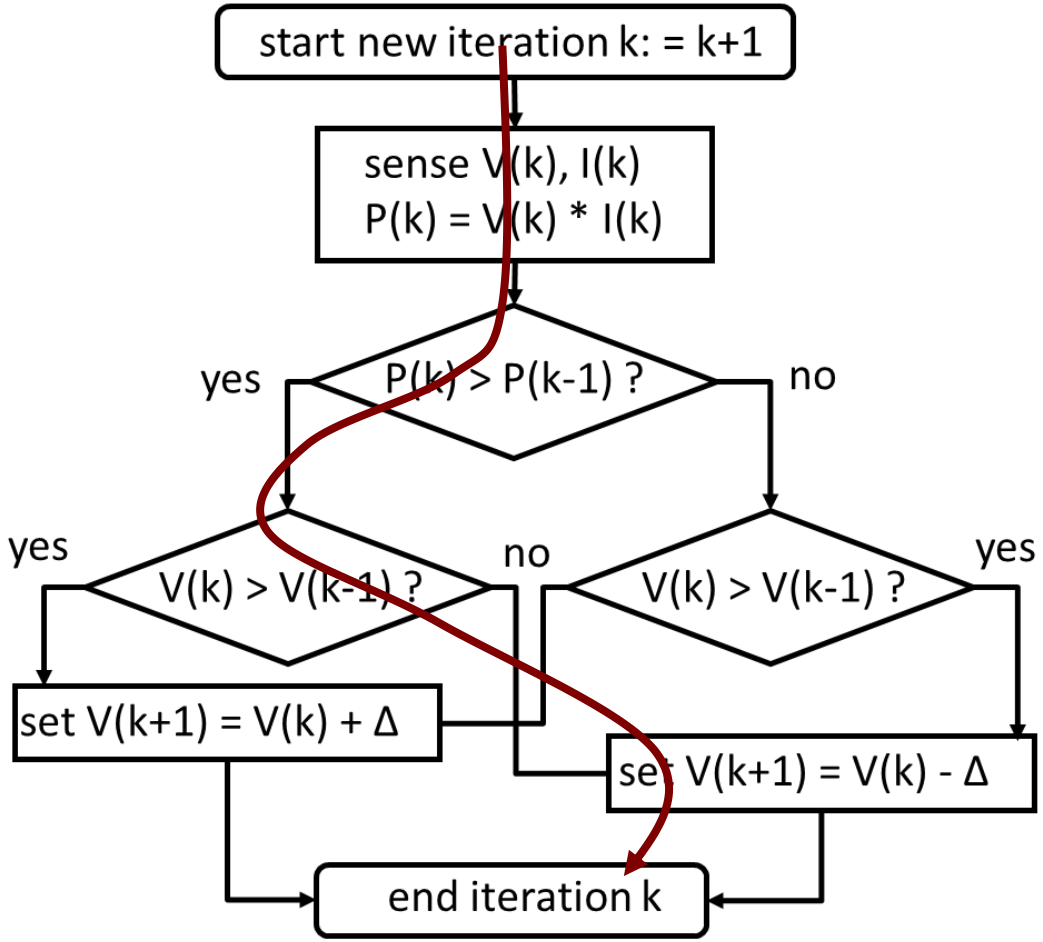
# Maximal Power Point Tracking



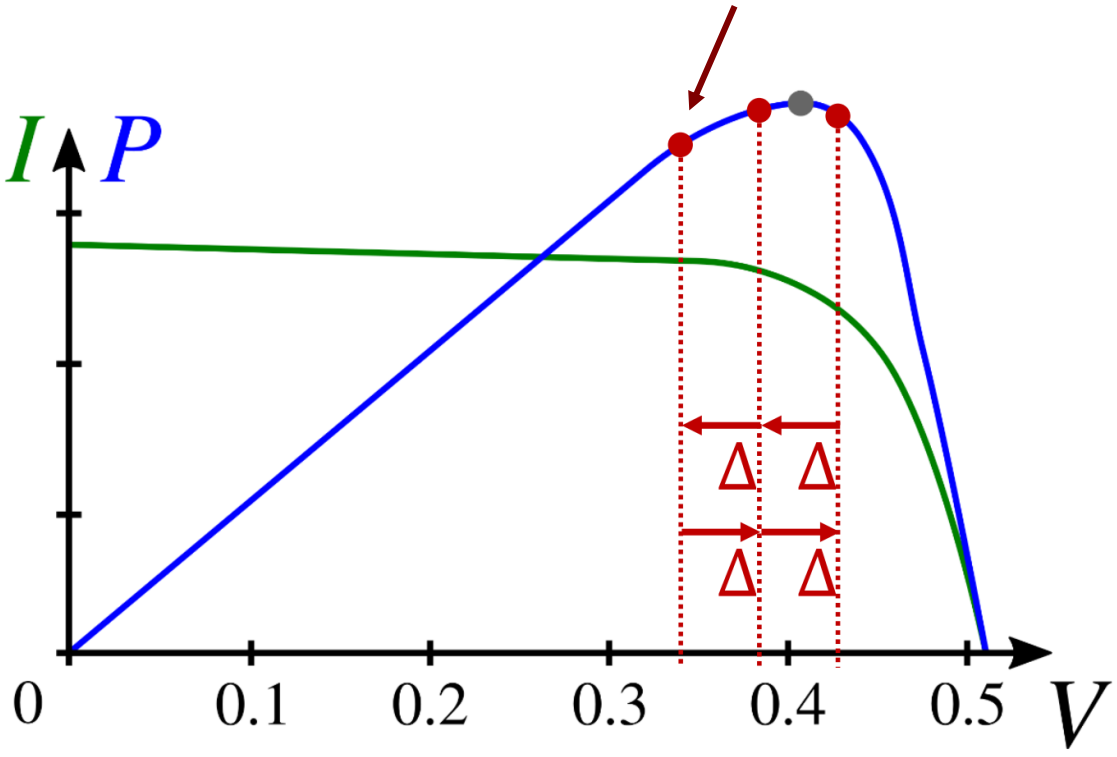



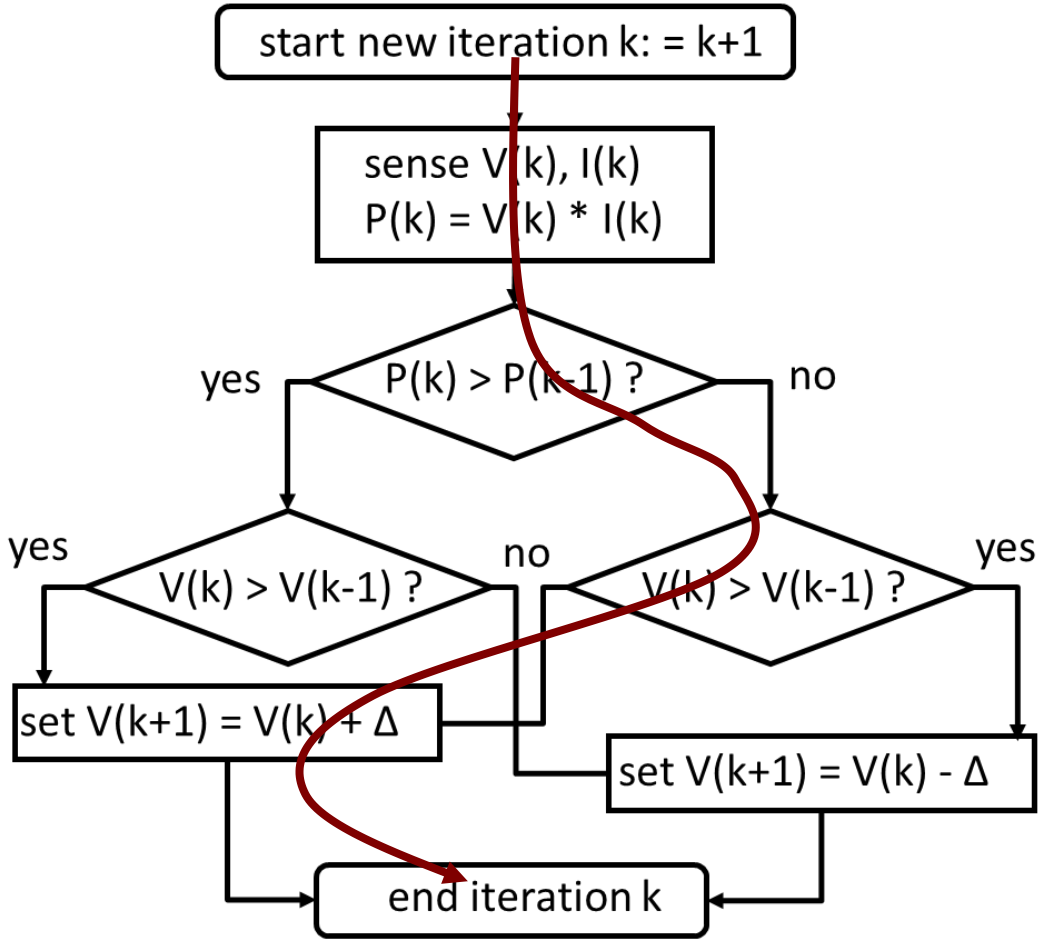
# Maximal Power Point Tracking



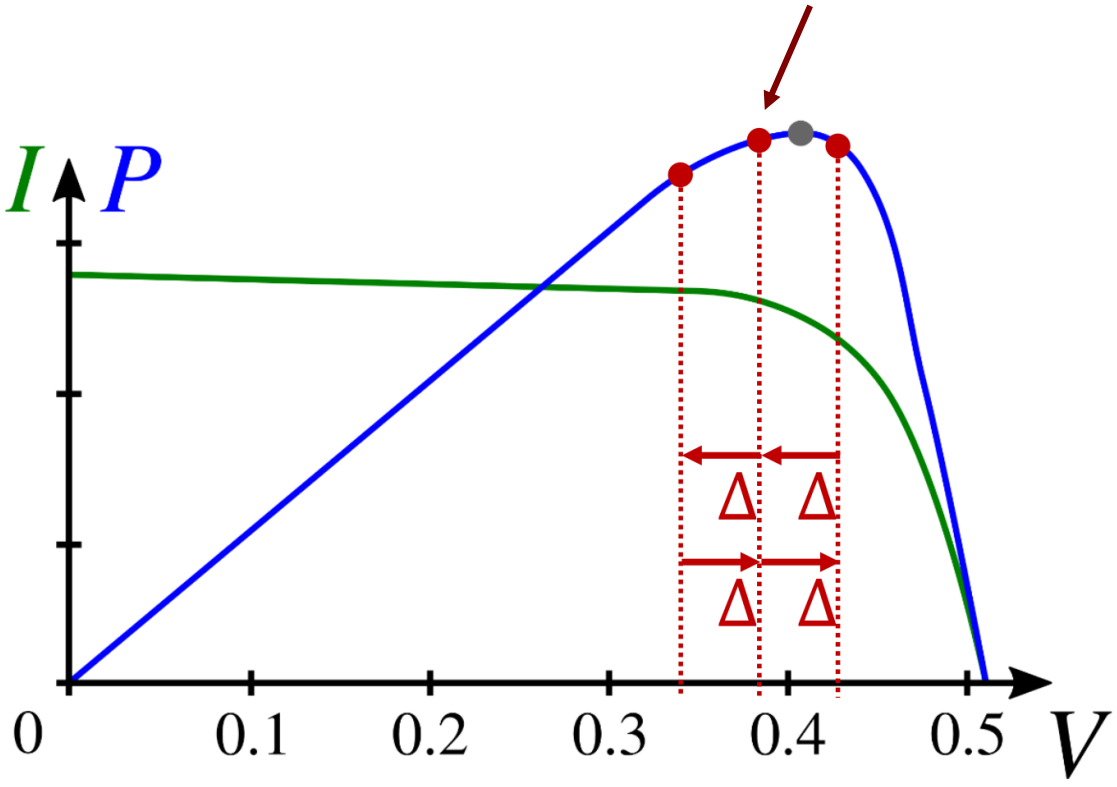



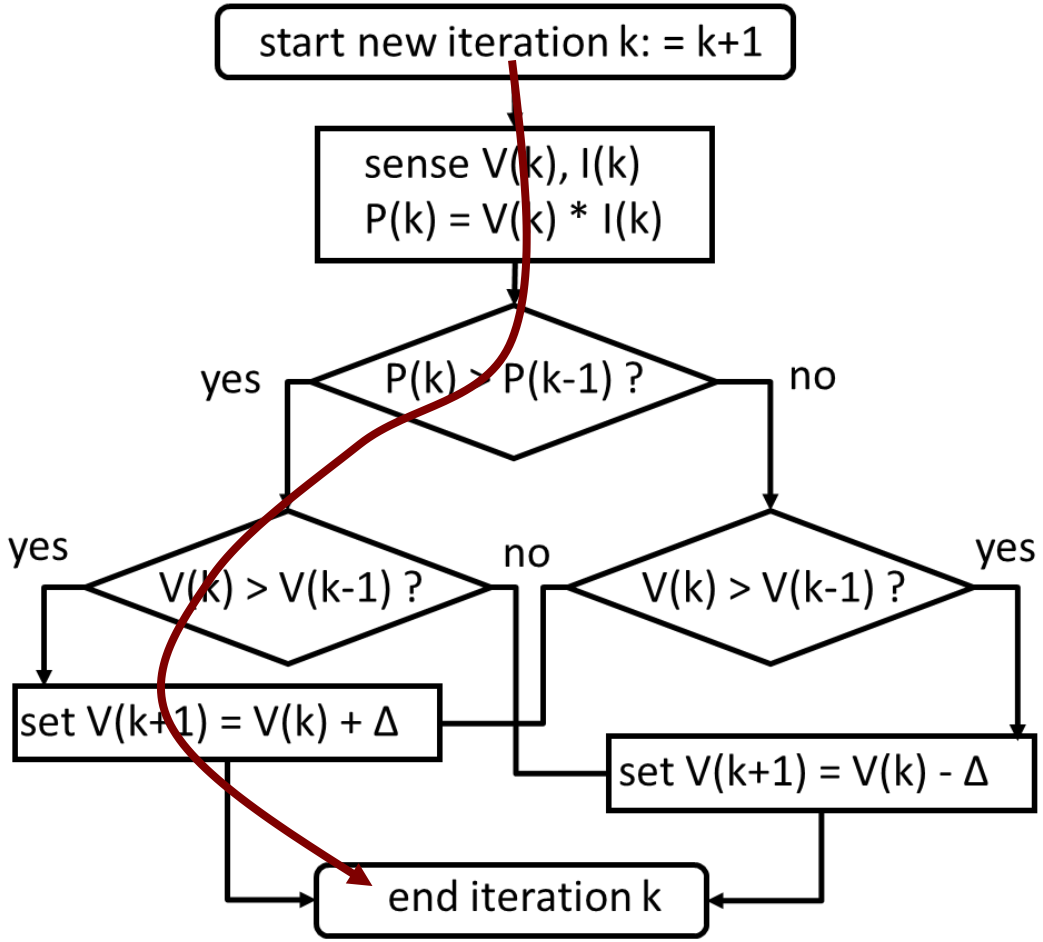
# Maximal Power Point Tracking



# Maximal Power Point Tracking



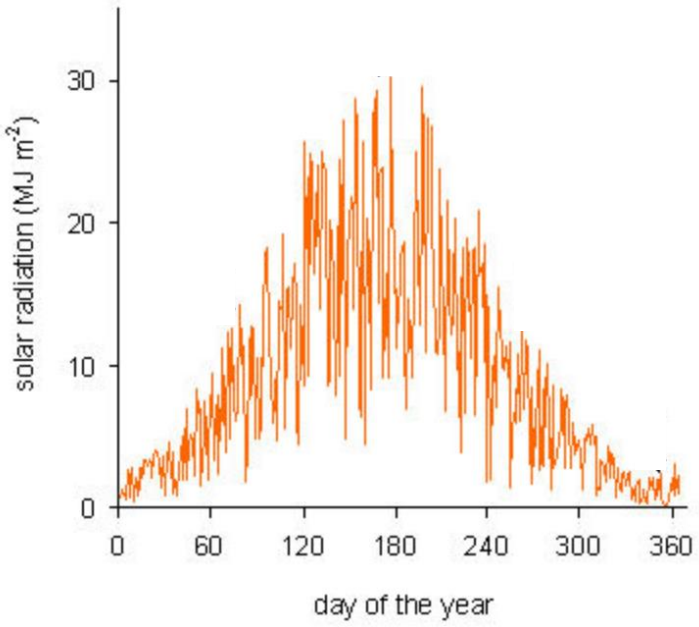



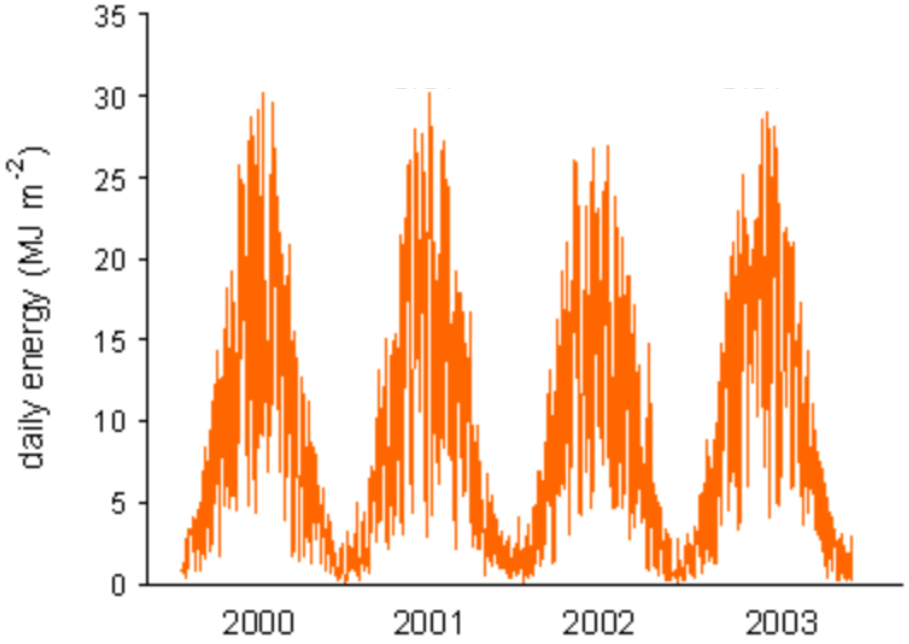
# Typical Challenge in (Solar) Harvesting Systems

**Challenges:**

- What is the optimal maximum capacity of the battery?
- What is the optimal area of the solar cell?
- How can we control the application such that a continuous system operation is possible, even under a varying input energy (summer, winter, clouds)?

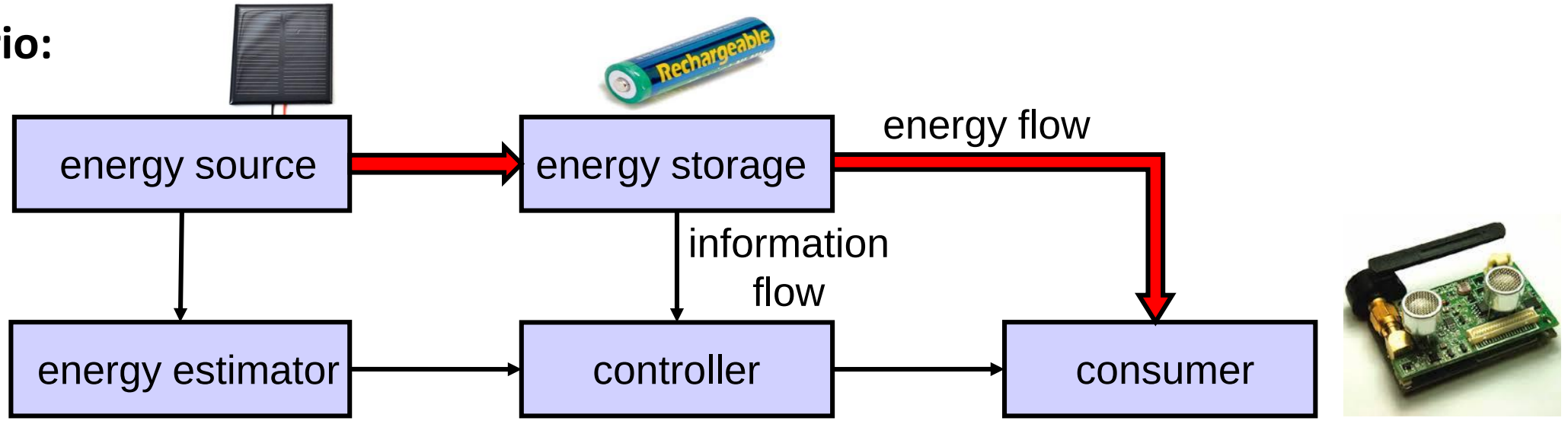
Example of a solar energy trace:



# Example: Application Control

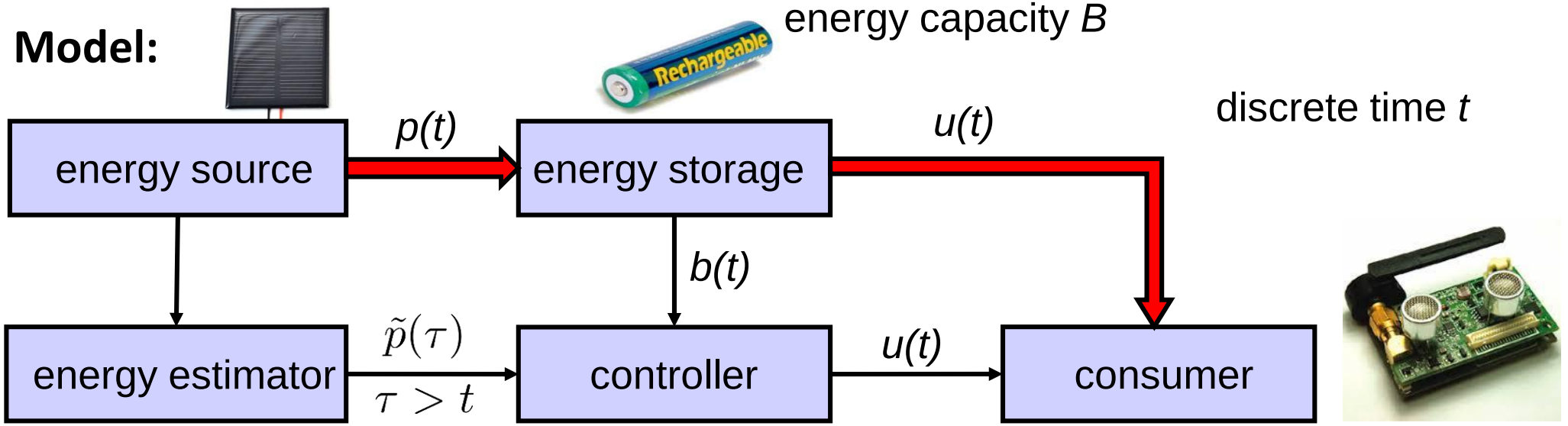
**Scenario:**




- The controller can adapt the service of the consumer device, for example the sampling rate for its sensors or the transmission rate of information. As a result, the power consumption changes proportionally.
- **Precondition for correctness** of application control: Never run out of energy.
- **Example for optimality criterion**: Maximize the lowest service of (or equivalently, the lowest energy flow to) the consumer.

# Application Control

**Formal Model:**




- harvested and used energy in *[t, t+1)*: *p(t), u(t)*
- battery model: $b(t+1) = \min\{b(t) + p(t) - u(t), B\}$
- failure state: $b(t) + p(t) - u(t) < 0$
- utility:

$$U(t_1, t_2) = \sum_{t_1 \leq \tau < t_2} \mu(u(\tau))$$

$\mu$ is a strictly concave function; higher used energy gives a reduced reward for the overall utility.

# Application Control

- **What do we want?** We would like to determine an optimal control *u*(t)* for time interval *[t, t+1)* for all *t in [0, T)* with the following properties:
  - $\forall\ 0 \leq t < T\ :\ b^*(t) + p(t) - u^*(t) \geq 0$
  - There is no feasible use function u(t) with a larger minimal energy:

    $$\forall u\ :\ \min_{0 \leq t < T}\{u(t)\} \leq \min_{0 \leq t < T}\{u^*(t)\}$$

  - The use function maximizes the utility *U(0, T).*
  - We suppose that the battery has the same or better state at the end than at the start of the time interval, i.e., *b*(T) ≥ b*(0).*
- We would like to answer two questions:
  - Can we say something about the characteristics of *u*(t)* ?
  - How does an algorithm look like that efficiently computes *u*(t)* ?

# Application Control

*Theorem:* Given a use function u*(t), $t \in [0, T)$ such that the system never enters a failure state. If u*(t) is optimal with respect to maximizing the minimal used energy among all use functions and maximizes the utility U(t, T), then the following relations hold for all $\tau \in (0, T)$:

$$u^*(\tau - 1) < u^*(\tau) \Longrightarrow b^*(\tau) = 0$$

empty battery

$$u^*(\tau - 1) > u^*(\tau) \Longrightarrow b^*(\tau) = B$$

full battery

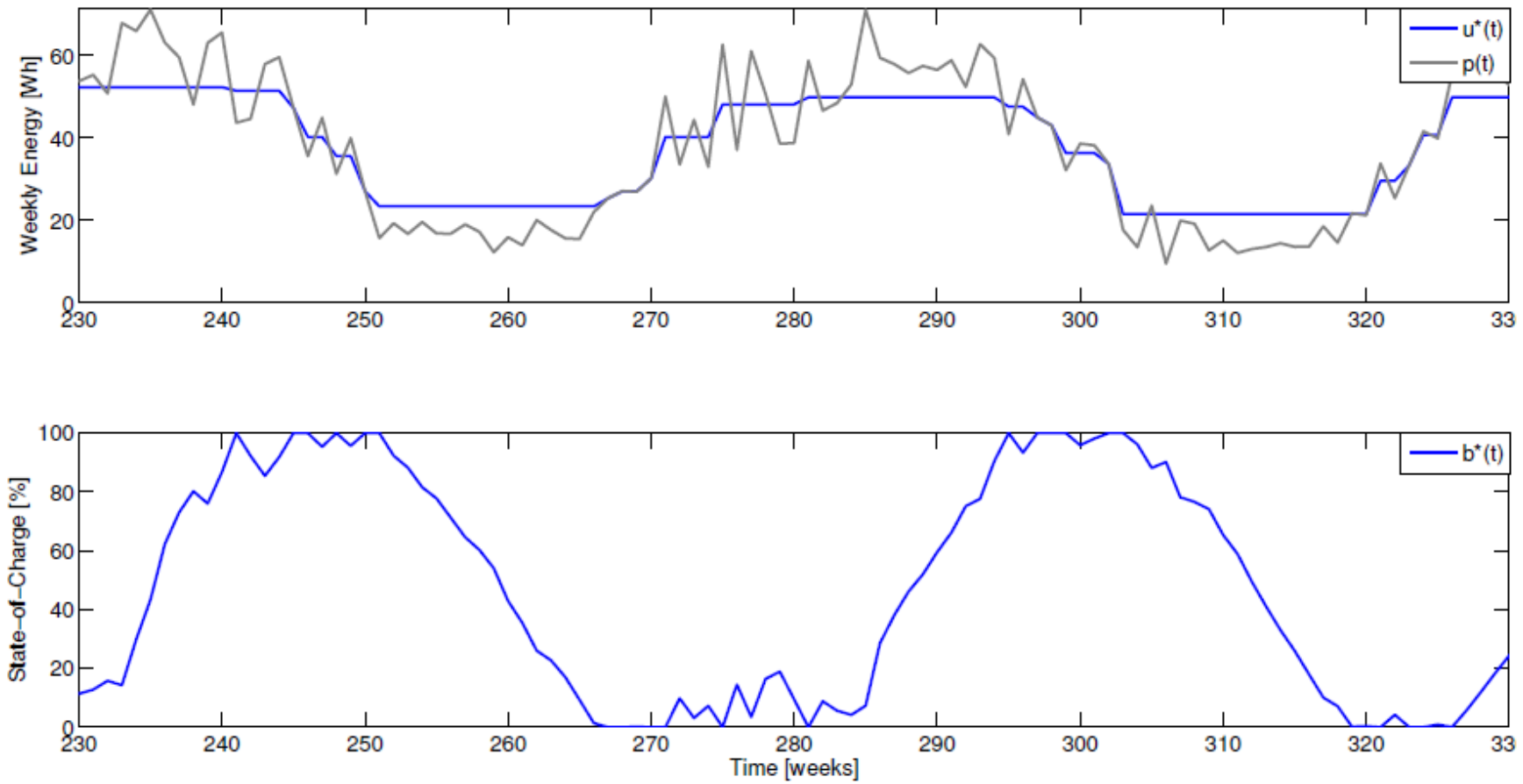
*Sketch of a proof*: First, let us show that a consequence of the above theorem is true (just reverting the relations):

$$\forall \tau \in (s, t] \ : \ 0 < b^*(\tau) < B \implies \forall \tau \in [s, t] \ : \ u^*(\tau) = u^*(t)$$

In other words, as long as the battery is neither full nor empty, the optimal use function does not change.

# Application Control

- Proof sketch cont.:

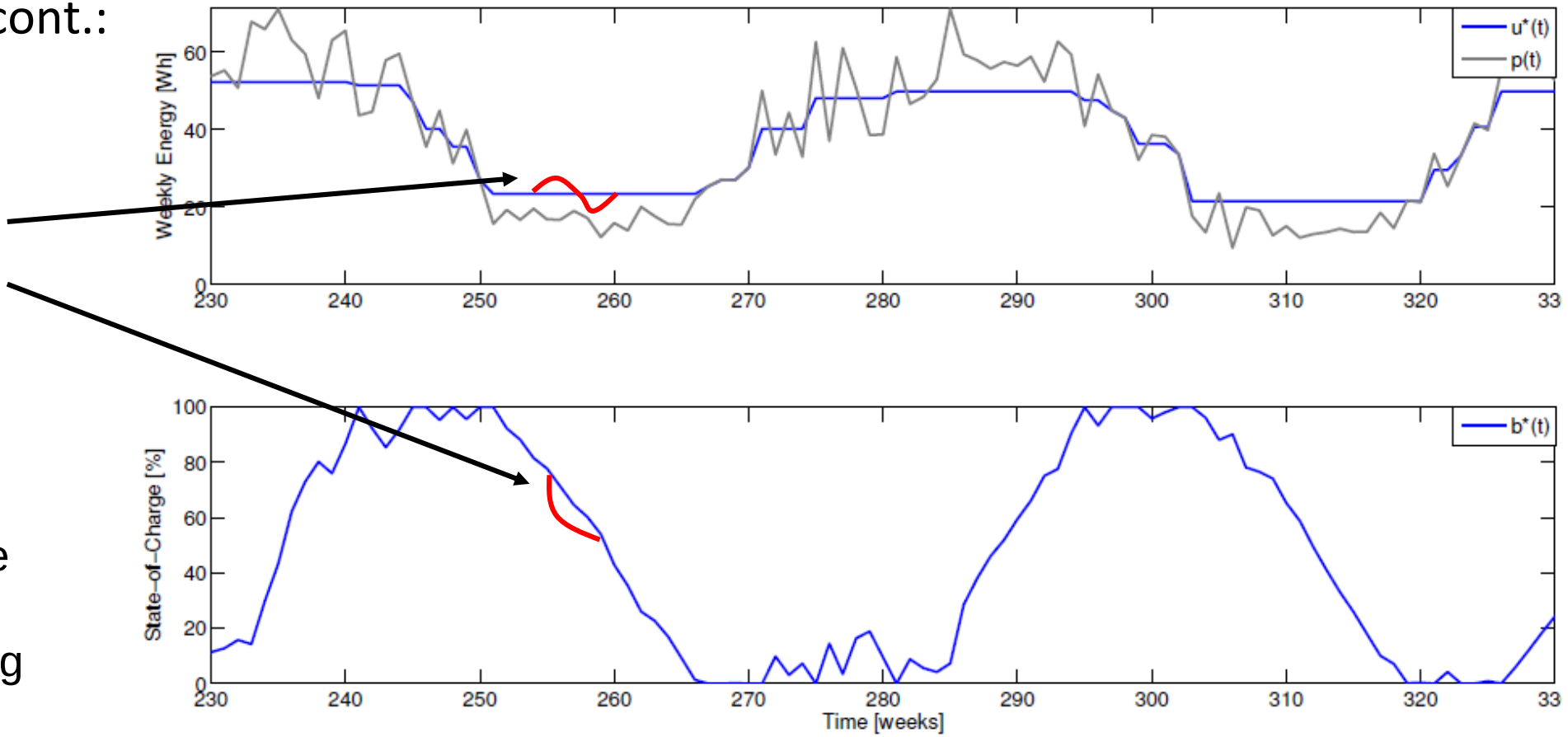



(top) Example of an optimal use function $u^*(t)$ for a given harvest function $p(t)$ and (bottom) the corresponding stored energy $b^*(t)$.

# Application Control

- Proof sketch cont.:

suppose we change the use function locally from being constant such that the overall battery state does not change




(top) Example of an optimal use function $u^*(t)$ for a given harvest function $p(t)$ and (bottom) the corresponding stored energy $b^*(t)$.

then the utility is worse due to the concave function $\mu$: diminishing reward for higher use function values; and the minimal use function is potentially smaller

# Application Control

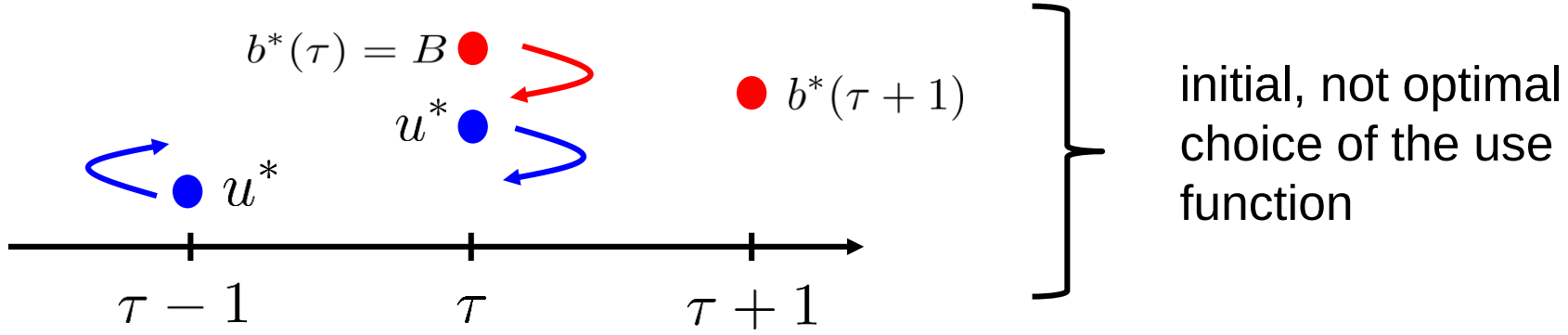
- Proof sketch cont.: Now we show that for all $\tau \in (t, T)$

$$u^*(\tau - 1) < u^*(\tau) \Longrightarrow b^*(\tau) = 0$$

or equivalently

$$b^*(\tau) > 0 \Longrightarrow u^*(\tau - 1) \geq u^*(\tau)$$

We already have shown this for $0 < b^*(\tau) < B$. Therefore, we only need to show that $b^*(\tau) = B \Longrightarrow u^*(\tau - 1) \geq u^*(\tau)$. Suppose now that we have $u^*(\tau - 1) < u^*(\tau)$ if the battery is full at $\tau$. Then we can increase the use at time $\tau - 1$ and decrease it at time $\tau$ by the same amount without changing the battery level at time $\tau + 1$. This again would increase the overall utility and potentially increase the minimal use function.



# Application Control

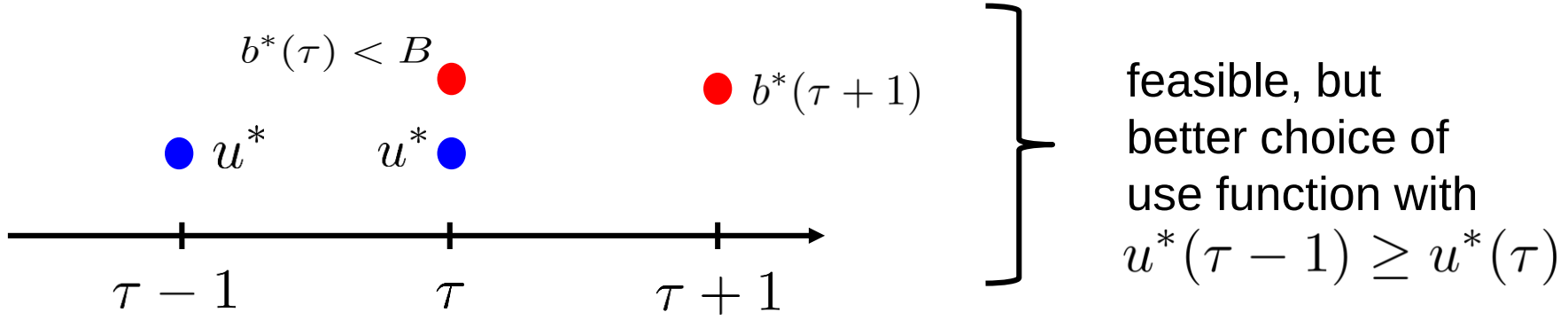
- Proof sketch cont.: Now we show that for all $\tau \in (t, T)$

$$u^*(\tau - 1) < u^*(\tau) \Longrightarrow b^*(\tau) = 0$$

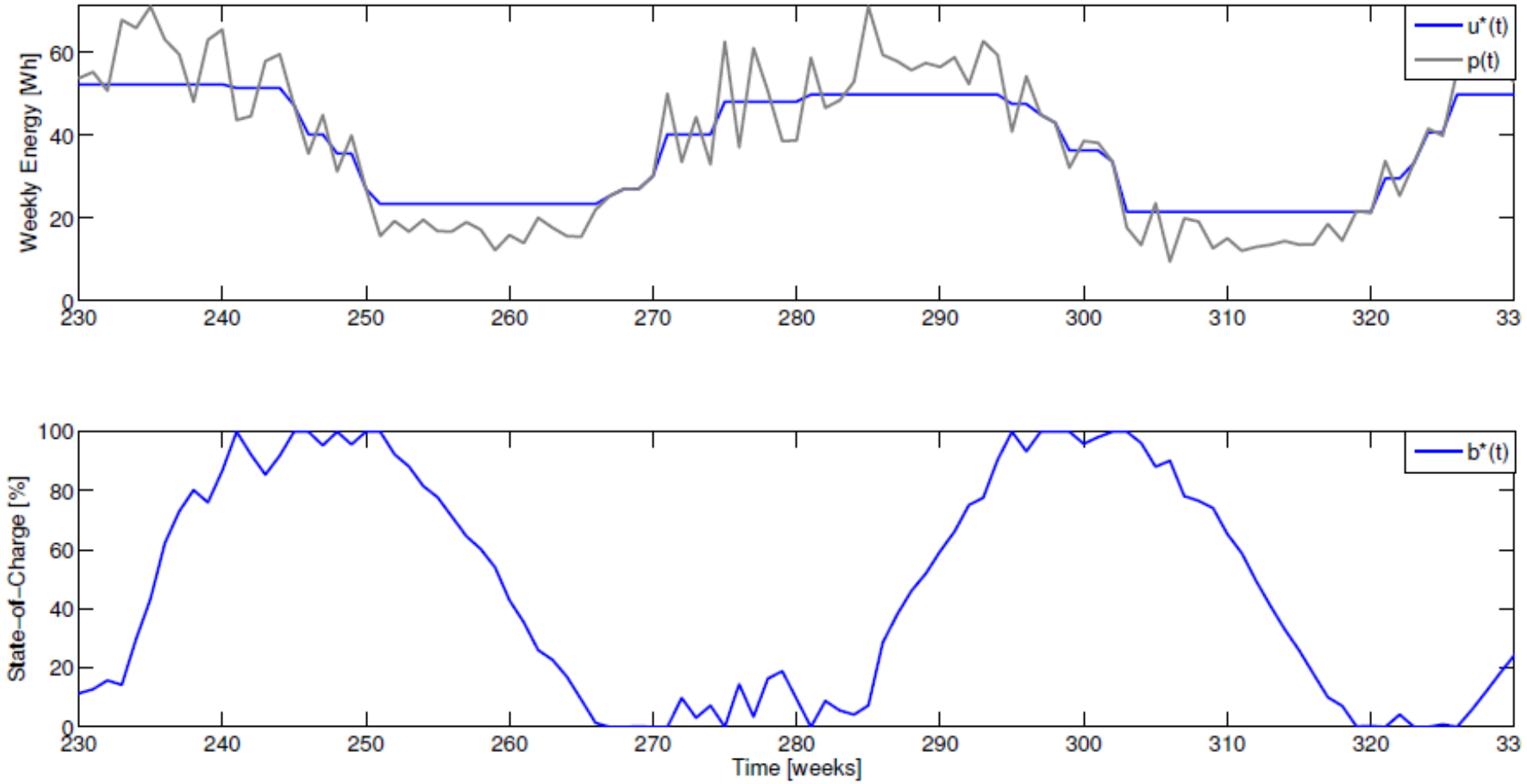
or equivalently

$$b^*(\tau) > 0 \Longrightarrow u^*(\tau - 1) \geq u^*(\tau)$$

We already have shown this for $0 < b^*(\tau) < B$. Therefore, we only need to show that $b^*(\tau) = B \Longrightarrow u^*(\tau - 1) \geq u^*(\tau)$. Suppose now that we have $u^*(\tau - 1) < u^*(\tau)$ if the battery is full at $\tau$. Then we can increase the use at time $\tau - 1$ and decrease it at time $\tau$ by the same amount without changing the battery level at time $\tau + 1$. This again would increase the overall utility and potentially increase the minimal use function.



# Application Control




(top) Example of an optimal use function $u^*(t)$ for a given harvest function $p(t)$ and (bottom) the corresponding stored energy $b^*(t)$.

# Application Control

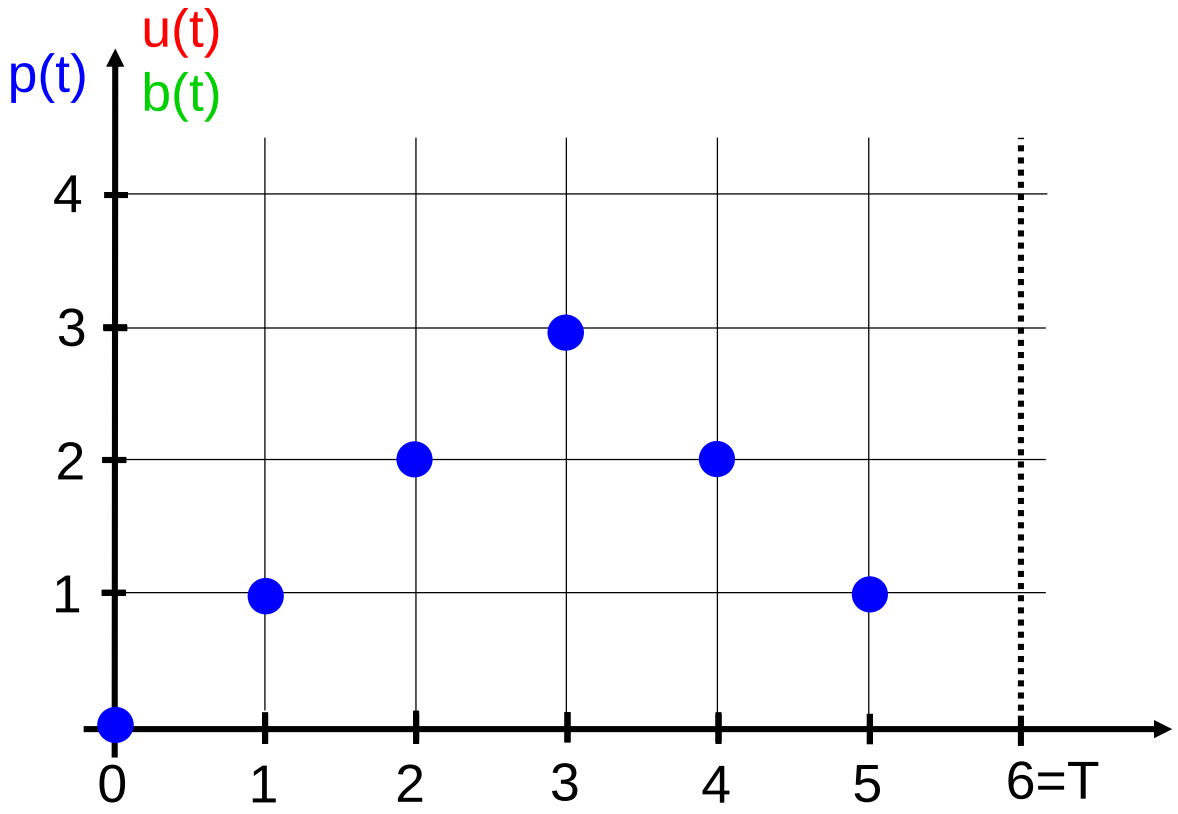
- How can we efficiently compute an optimal use function?
  - There are several options available as we just need to solve a convex optimization problem.
  - A simple but inefficient possibility is to convert the problem into a linear program. At first suppose that the utility is simply

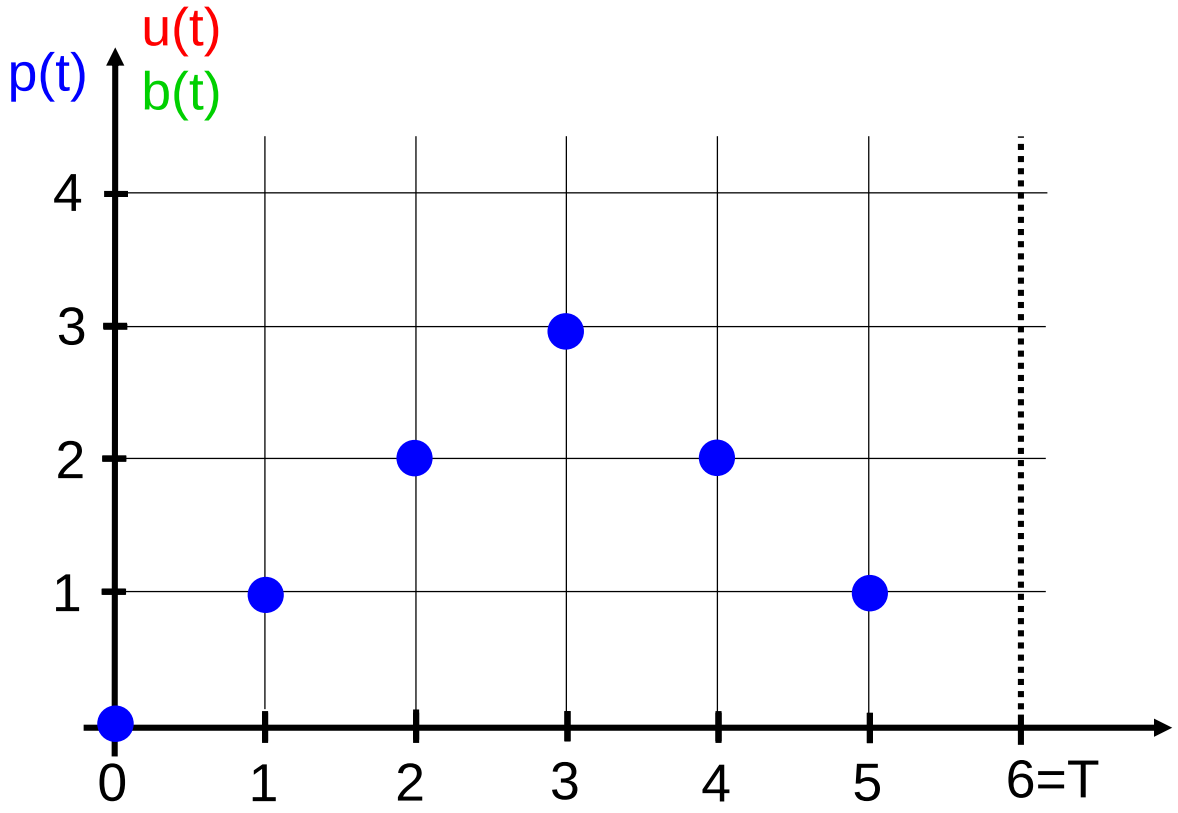
$$U(0,T) = \sum_{0 \le \tau < T} u(\tau)$$

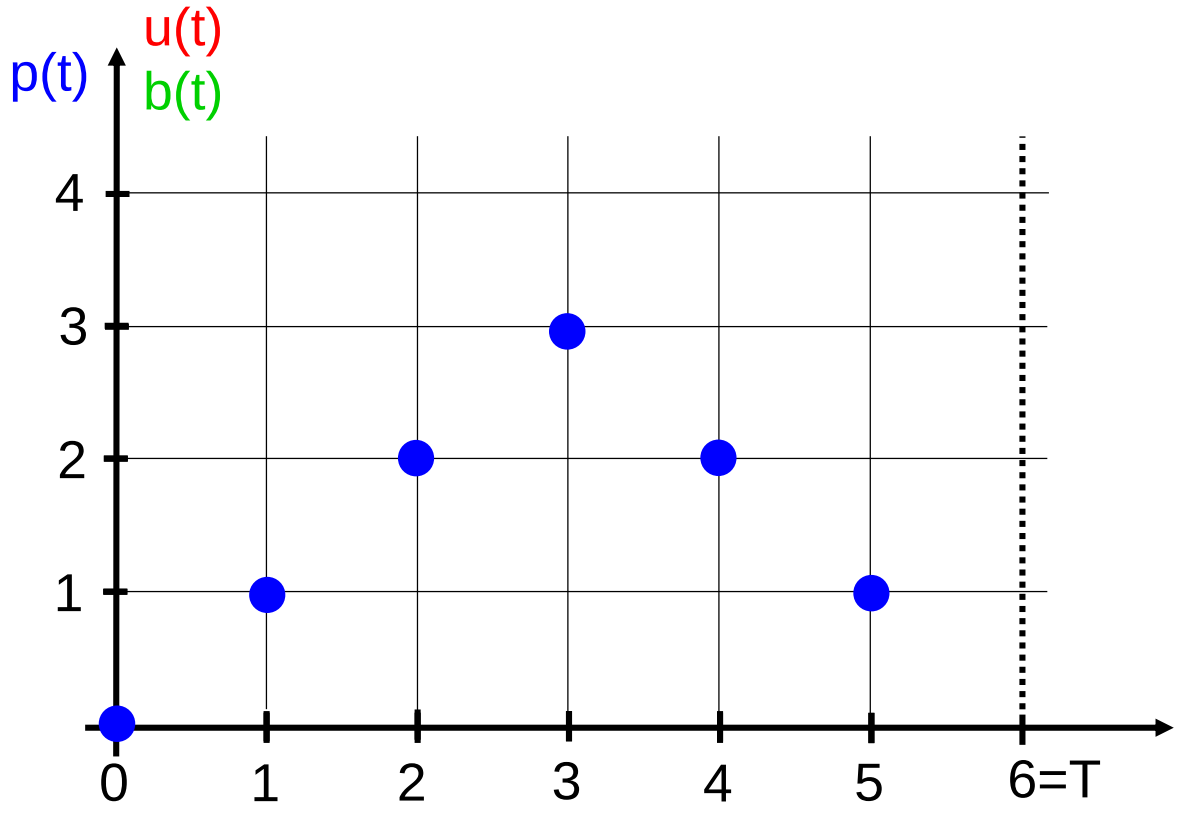
Then the linear program has the form:

[Concave functions $\mu$ could be piecewise linearly approximated. This is not shown here.]

$$\begin{aligned}
&\text{maximize} \quad \sum_{0 \le \tau < T} u(\tau) \\
&\forall \tau \in [0,T) \ : \ b(\tau+1) = b(\tau) - u(\tau) + p(\tau) \\
&\forall \tau \in [0,T) \ : \ 0 \le b(\tau+1) \le B \\
&\forall \tau \in [0,T) \ : \ u(\tau) \ge 0 \\
&b(T) = b(0) = b_0
\end{aligned}$$


u(t)
p(t)
b(t)
4
3
2
1
0
1
2
3
4
5
6=T


u(t)
b(t)
p(t)
4
3
2
1
0
1
2
3
4
5
6=T

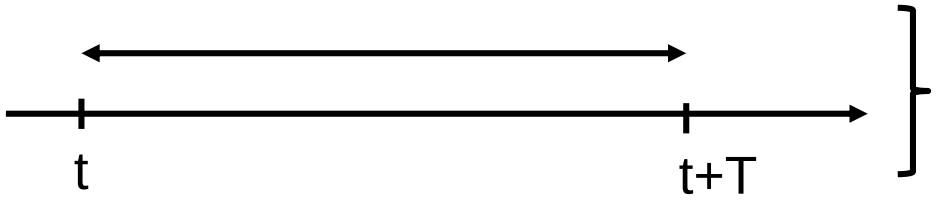

u(t)
b(t)
p(t)
4
3
2
1
0
1
2
3
4
5
6=T

# Application Control

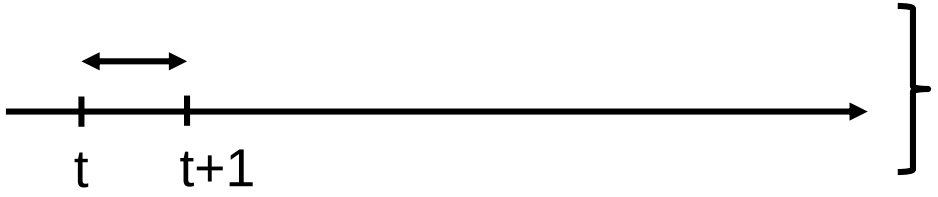
- But what happens if the estimation of the future incoming energy is not correct?
  - If it would be correct, then we would just compute the whole future application control now and would not change anything anymore.
  - This will not work as errors will accumulate and we will end up with many infeasible situations, i.e., the battery is completely empty and we are forced to stop the application.
  - **Possibility**: Finite horizon control
    - At time t, we compute the optimal control (see previous slides) using the currently available battery state b(t) with predictions $\tilde{p}(\tau)$ for all $t \leq \tau < t+T$ and $b(t+T) = b(t)$ .
    - From the computed optimal use function $u(\tau)$ for all $t \leq \tau < t+T$ we just take the first use value *u(t)* in order to control the application.
    - At the next time step, we take as initial battery state the actual state; therefore, we take mispredictions into account. For the estimated future energy, we also take the new estimations.

# Application Control

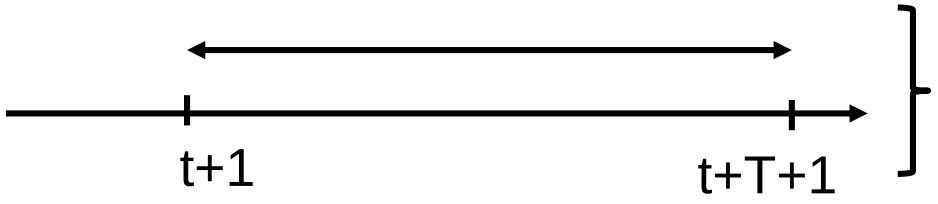
- **Finite horizon control**:




compute the optimal use function in [t, t+T)
using the actual battery state at time t

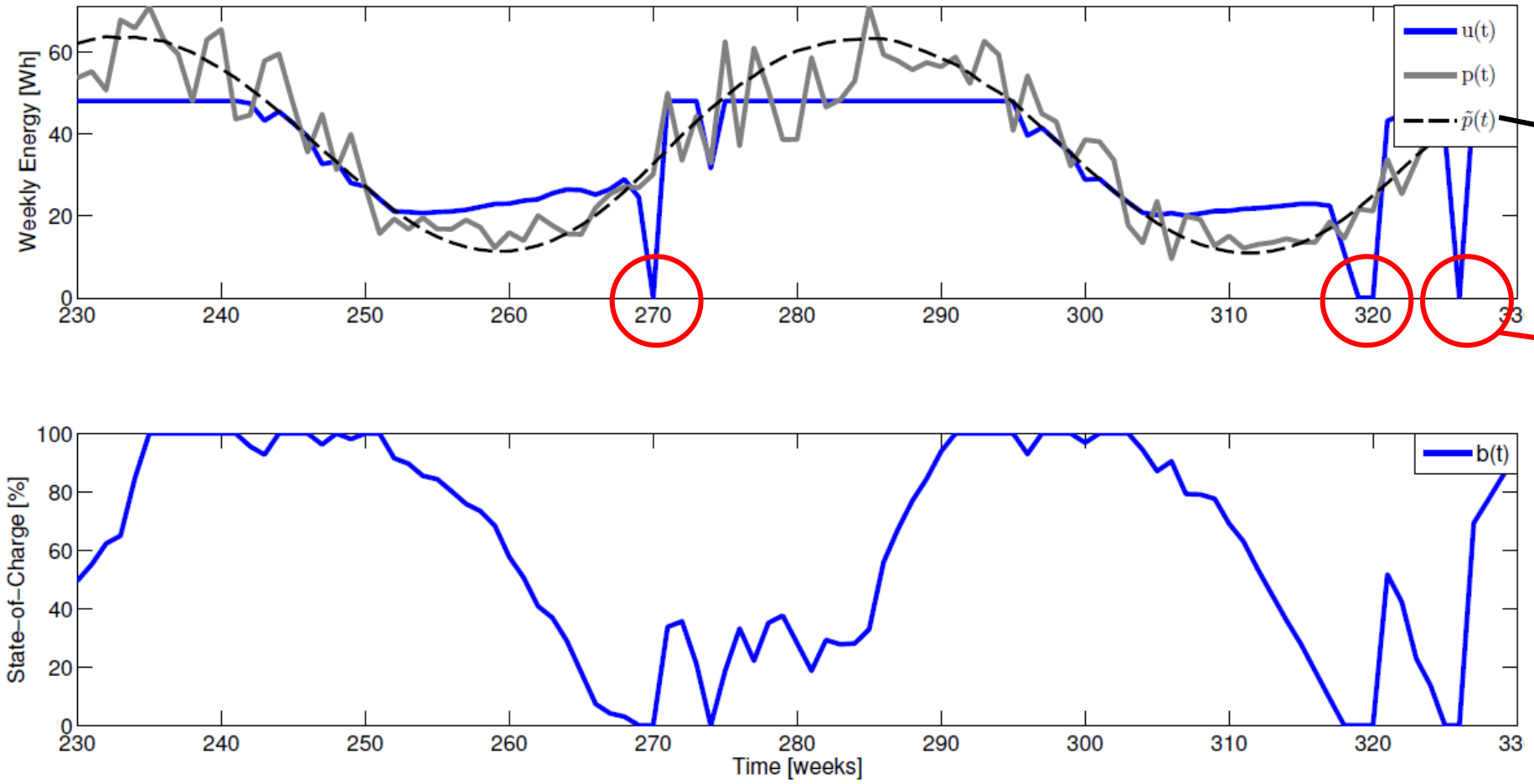



apply this use function in the interval [t, t+1).




compute the optimal use function in [t+1, t+T+1)
using the actual batter state at time t+1

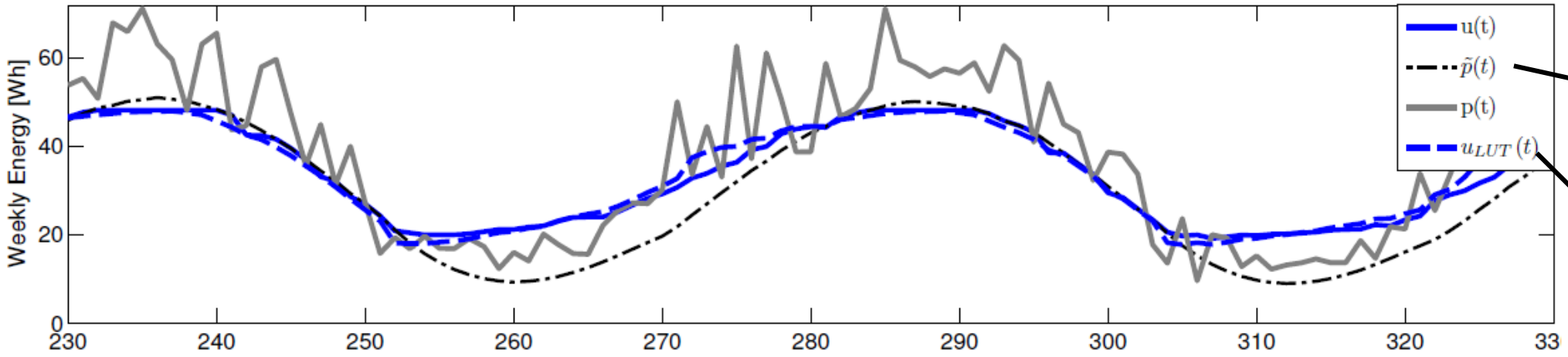
# Application Control using Finite Horizon




estimated input energy

still energy breakdown due to misprediction

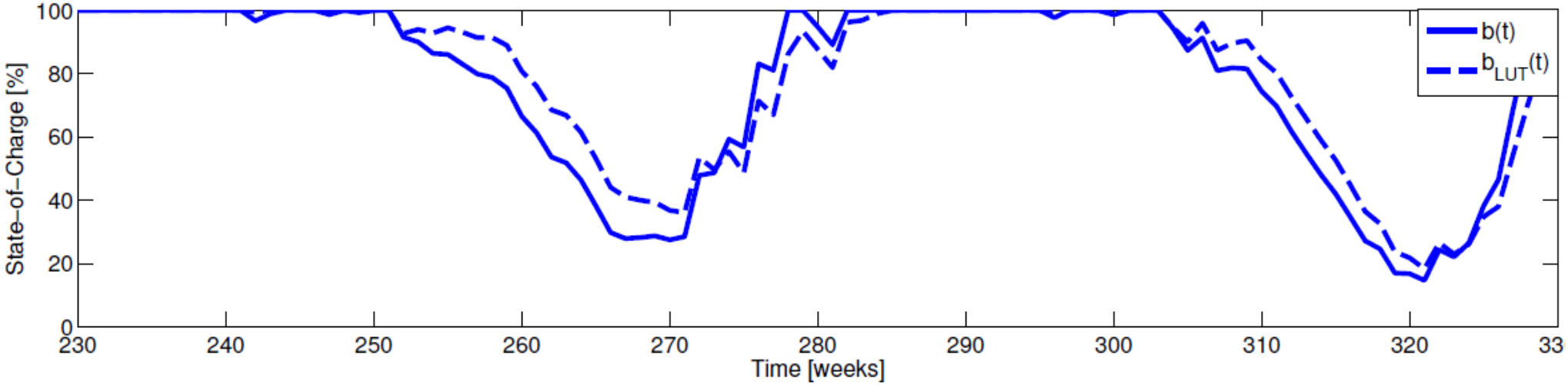
# Application Control using Finite Horizon




more pessimistic prediction

simplified optimization using a look-up-table

[not covered]



# Remember: What you got some time ago …


# What we told you: Be careful and please do not …


**Return the boards at the embedded systems exam!**

# Embedded Systems

## 10. Architecture Synthesis

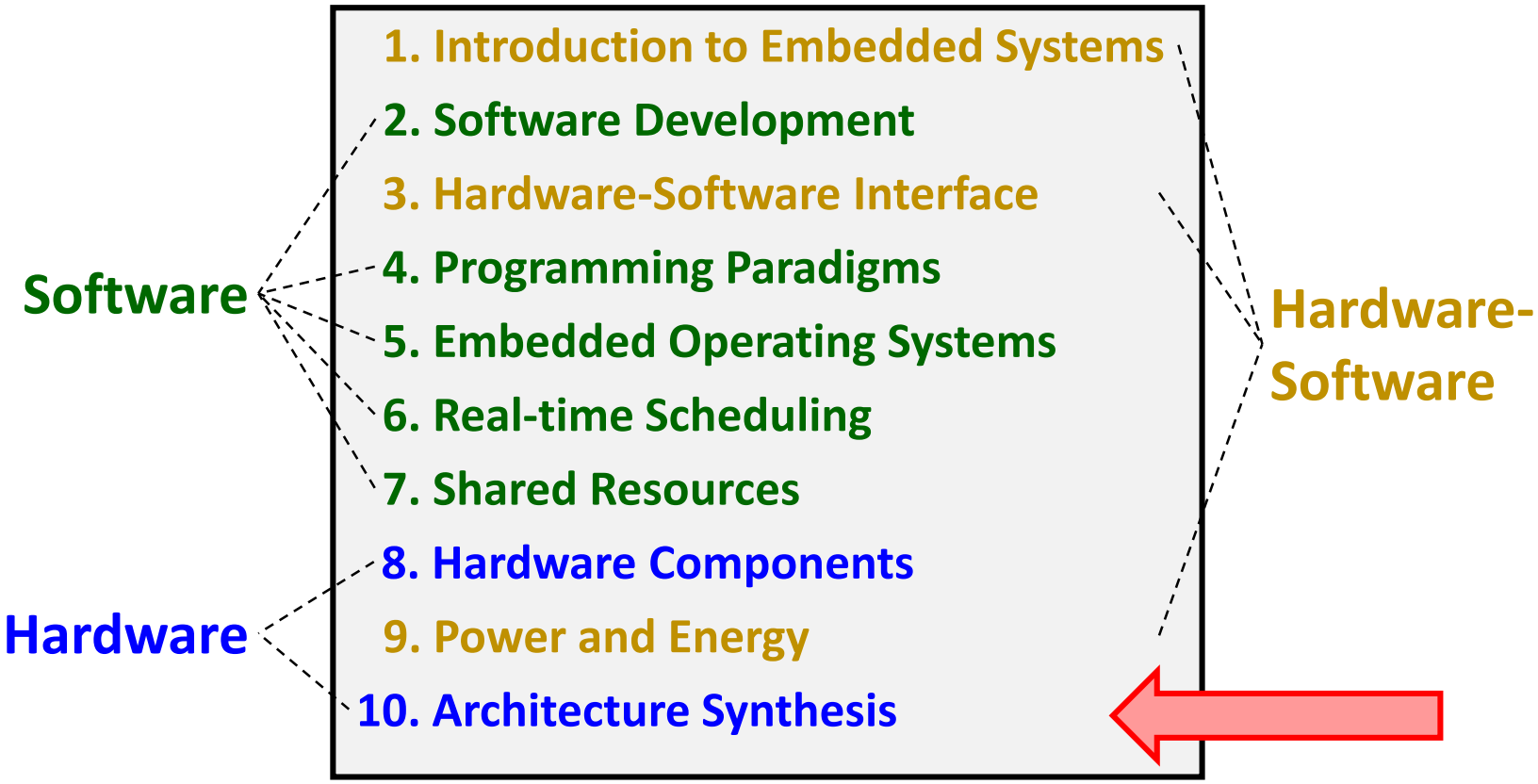
© Lothar Thiele

Computer Engineering and Networks Laboratory

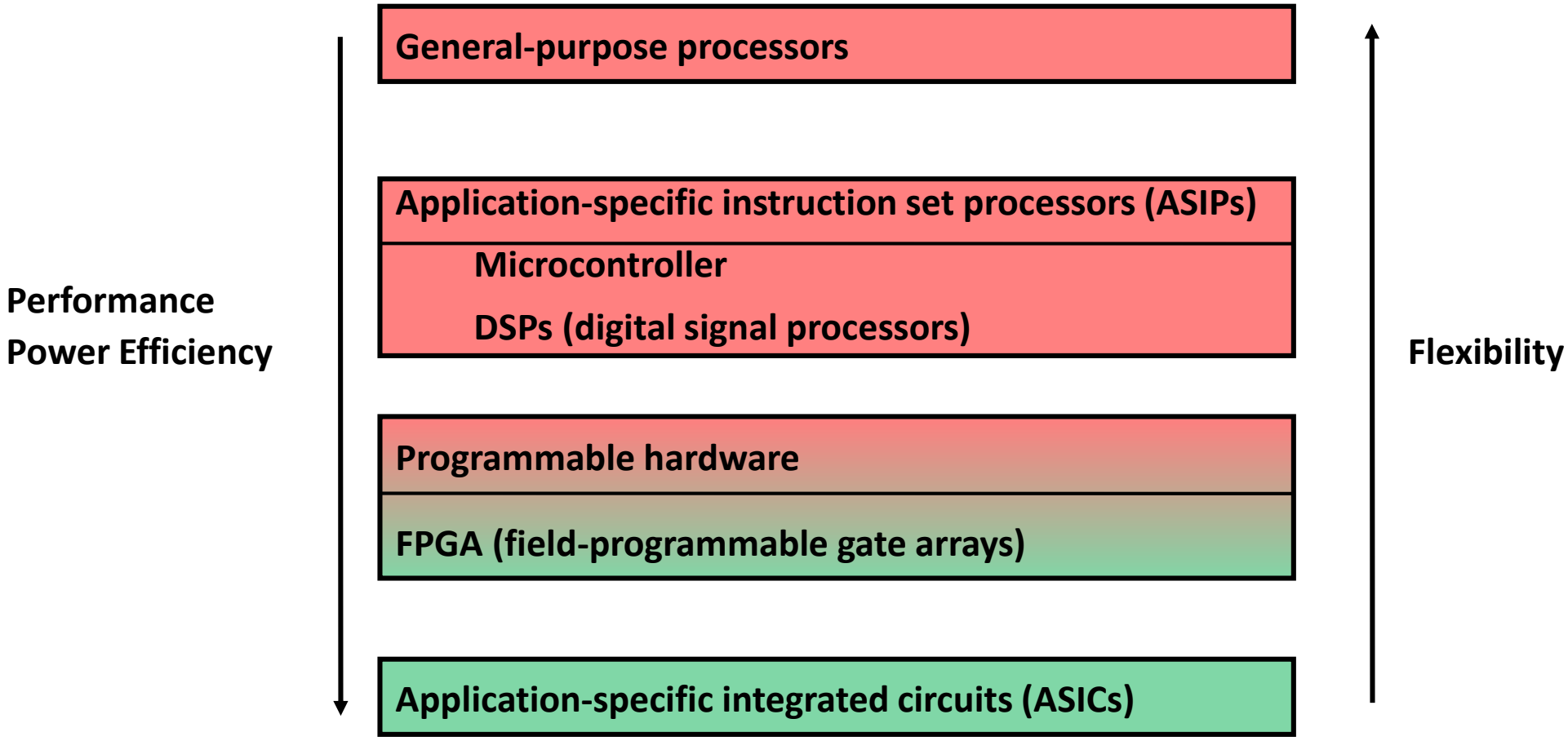
ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Lecture Overview



# Implementation Alternatives



# 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.

# Specification Models

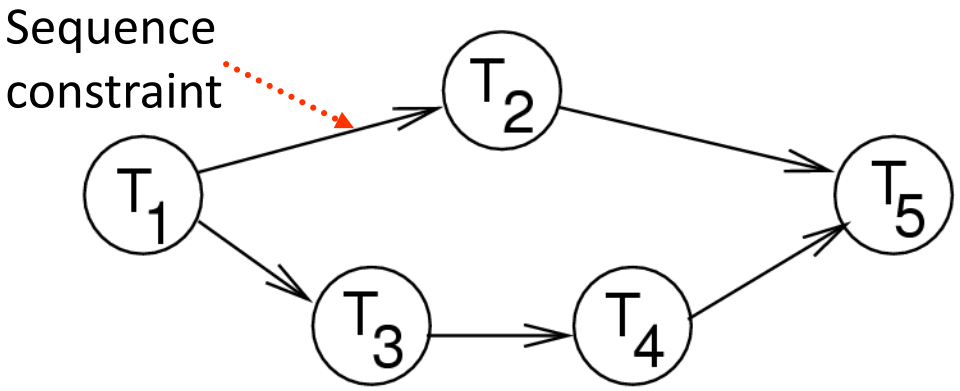
# Specification

- *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 and general specification models for the whole system.

- Now, we will introduce some relevant models for architecture level (hardware) synthesis.

# Task Graph or Dependence Graph (DG)




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

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

If $(v1, v2) \in E$, then *v1* is called an **immediate predecessor** of *v2* and *v2* is called an **immediate successor** of *v1*.

Suppose $E^*$ is the transitive closure of $E$. If $(v1, v2) \in E^*$, then *v1* is called a **predecessor** of *v2* and *v2* is called a **successor** of *v1*.

# Dependence Graph

- *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

# Dependence Graph and Single Assignment Form

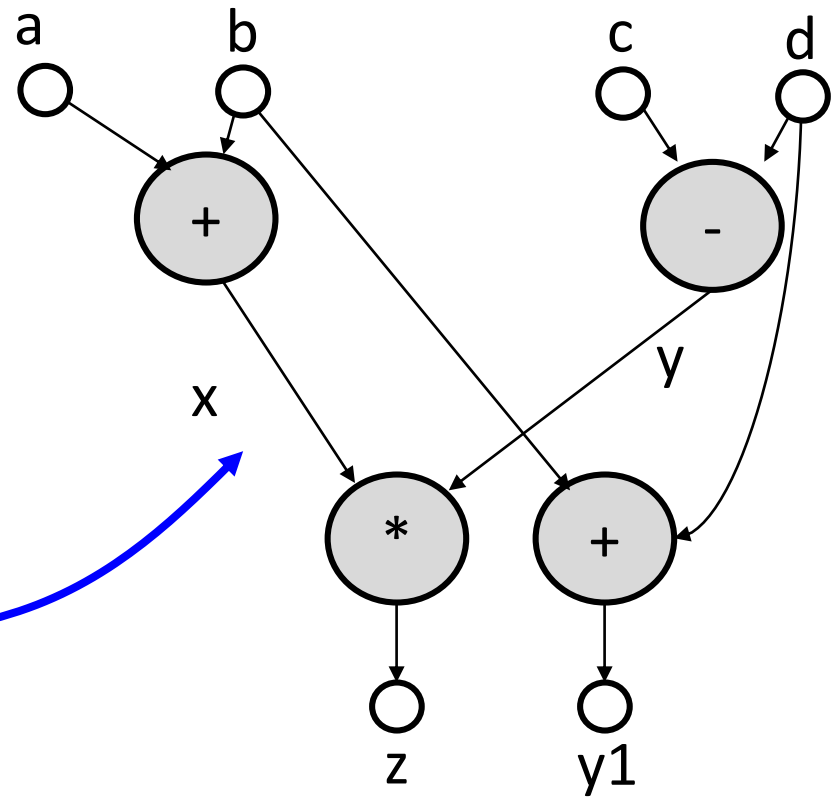
**given basic block**:

```
x = a + b;
y = c - d;
z = x * y;
y = b + d;
```

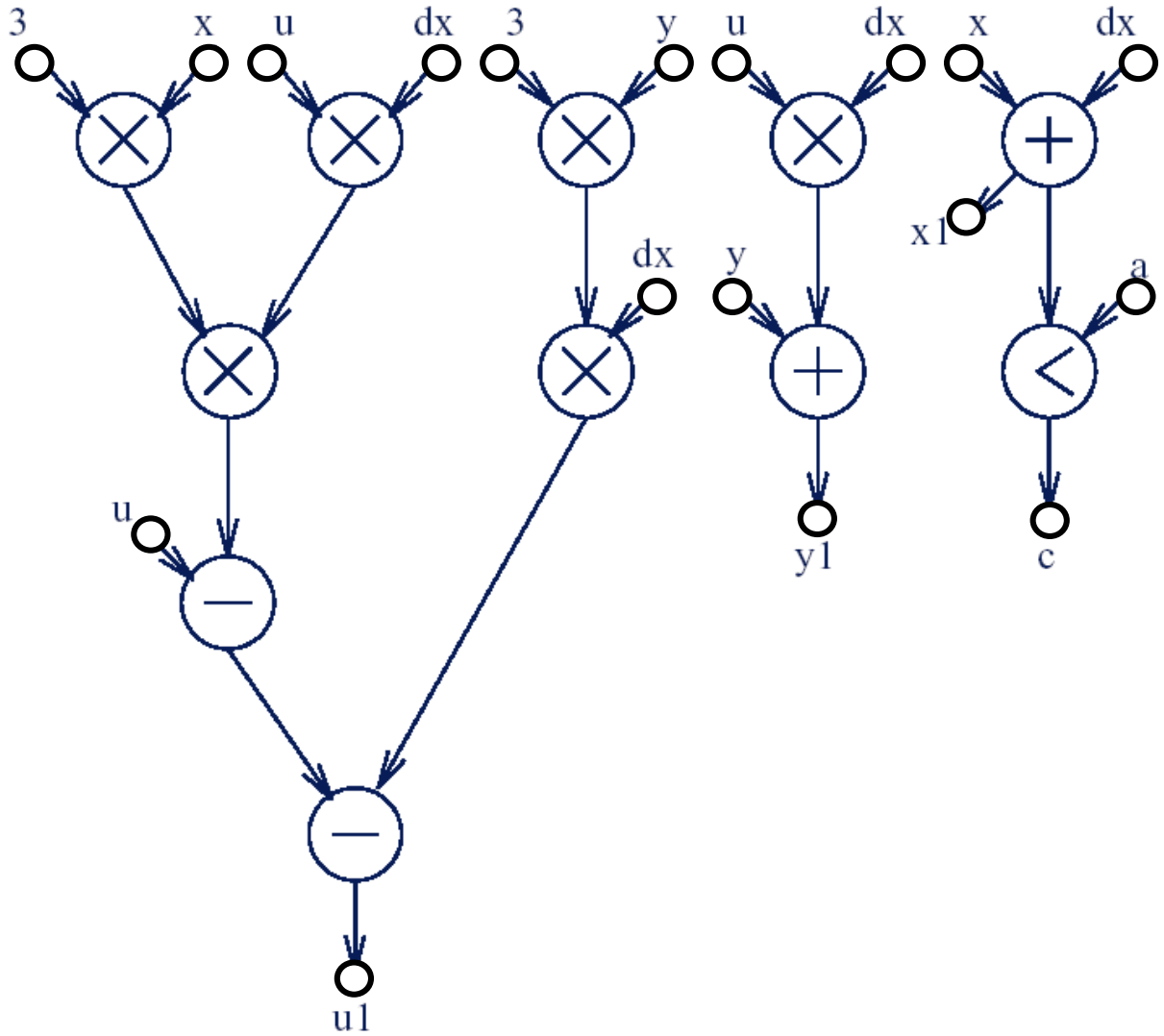
**single assignment form:**

```
x = a + b;
y = c - d;
z = x * y;
y1 = b + d;
```

dependence graph



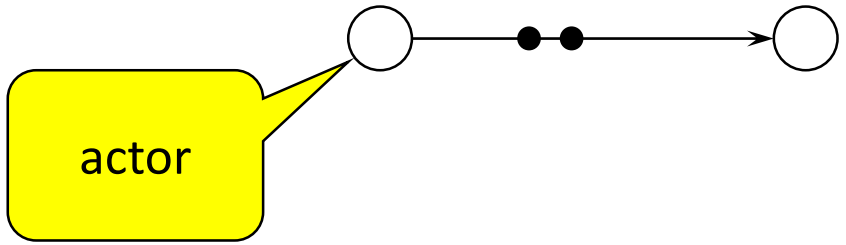
# Example of a Dependence Graph

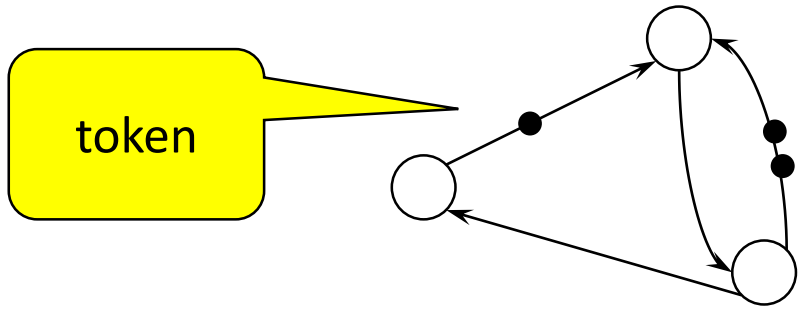


# Marked Graph (MG)

- *A marked graph* $G = (V, A, del)$ consists of
  - nodes (actors) $v \in V$
  - edges $a = (v_i, v_j) \in A, \quad A \subseteq V \times V$
  - number of initial tokens (or marking) on edges $del : A \rightarrow \mathbf{Z}^{\geq 0}$

- The *marking* is often represented in form of a vector: $del = \begin{pmatrix} del_1 \\ \cdots \\ del_i \\ \cdots \\ del_{|A|} \end{pmatrix}$



# Marked Graph

- 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.

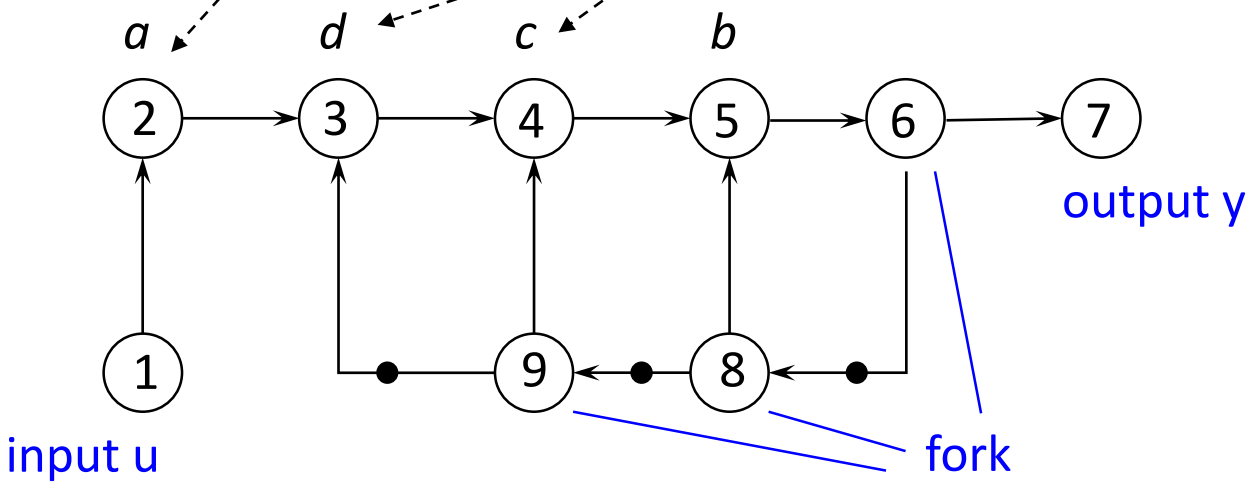
# Marked Graph

## Example (model of a digital filter with infinite impulse response IIR)

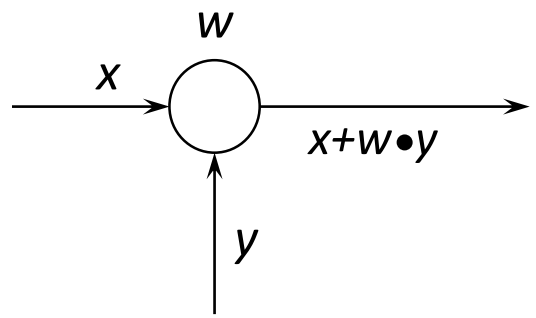
- Filter equation:

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

- Possible model as a *marked graph*:



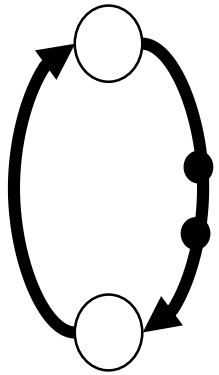

nodes 3-5:

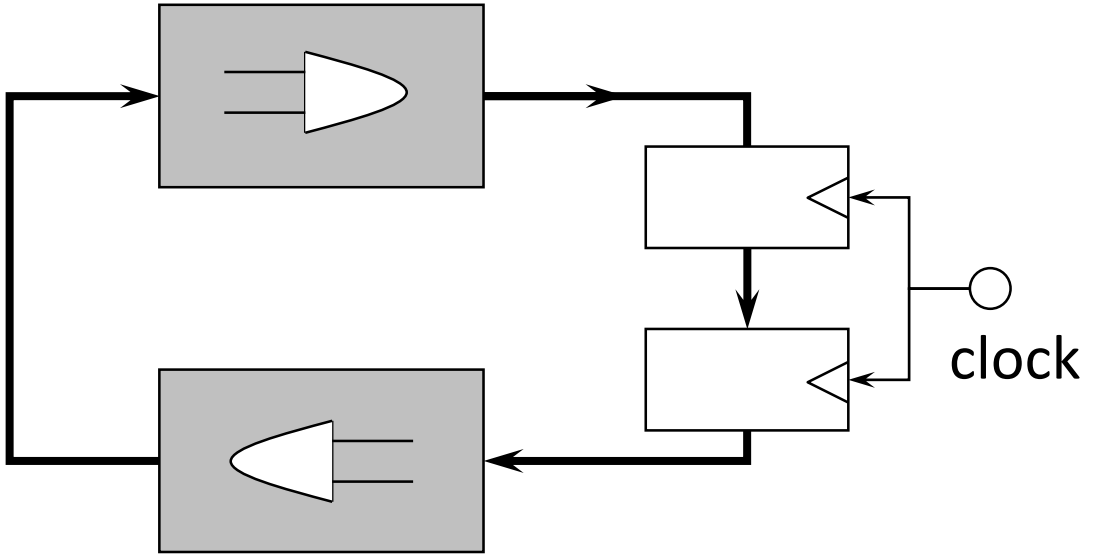



node 2: *x=0*

# Implementation of Marked Graphs

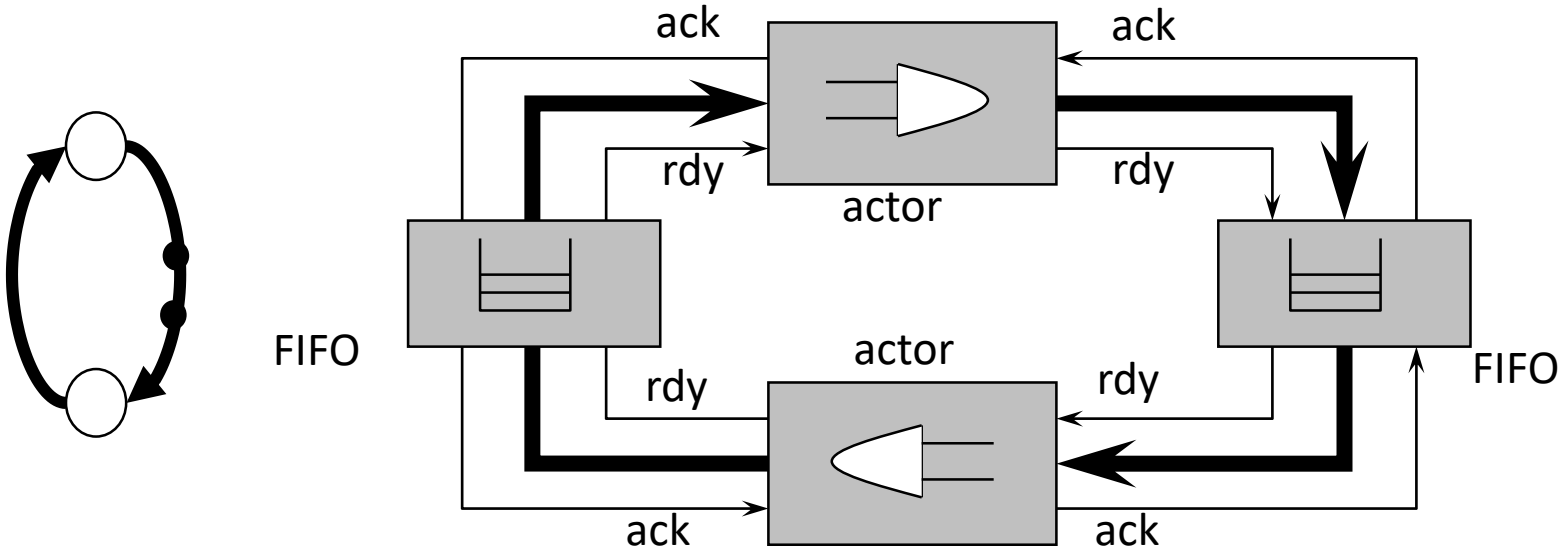
- There are *different possibilities to implement marked graphs* in hardware or software directly. Only the most simple possibilities are shown here.
- *Hardware implementation* as a synchronous digital circuit:
  - Actors are implemented as combinatorial circuits.
  - Edges correspond to synchronously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

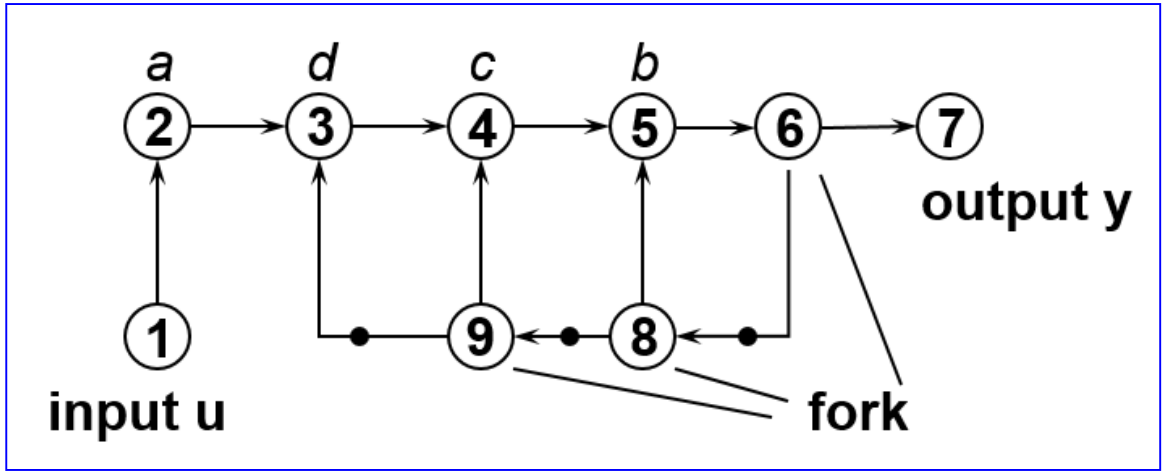


# Implementation of Marked Graphs

- *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.

# Models for Architecture Synthesis

- *A sequence graph* $G_S = (V_S, E_S)$ is a dependence graph with a single start node (no incoming edges) and a single end node (no outgoing edges).
  $V_S$ denotes the operations of the algorithm and $E_S$ denotes the dependence relations.

- *A resource graph* $G_R = (V_R, E_R),\ V_R = V_S \cup V_T$ models resources and bindings.
  $V_T$ 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* $w : E_R \rightarrow \mathbf{Z}^{\geq 0}$ 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 for Architecture Synthesis - Example

*Example sequence graph:*

- Algorithm (differential equation):

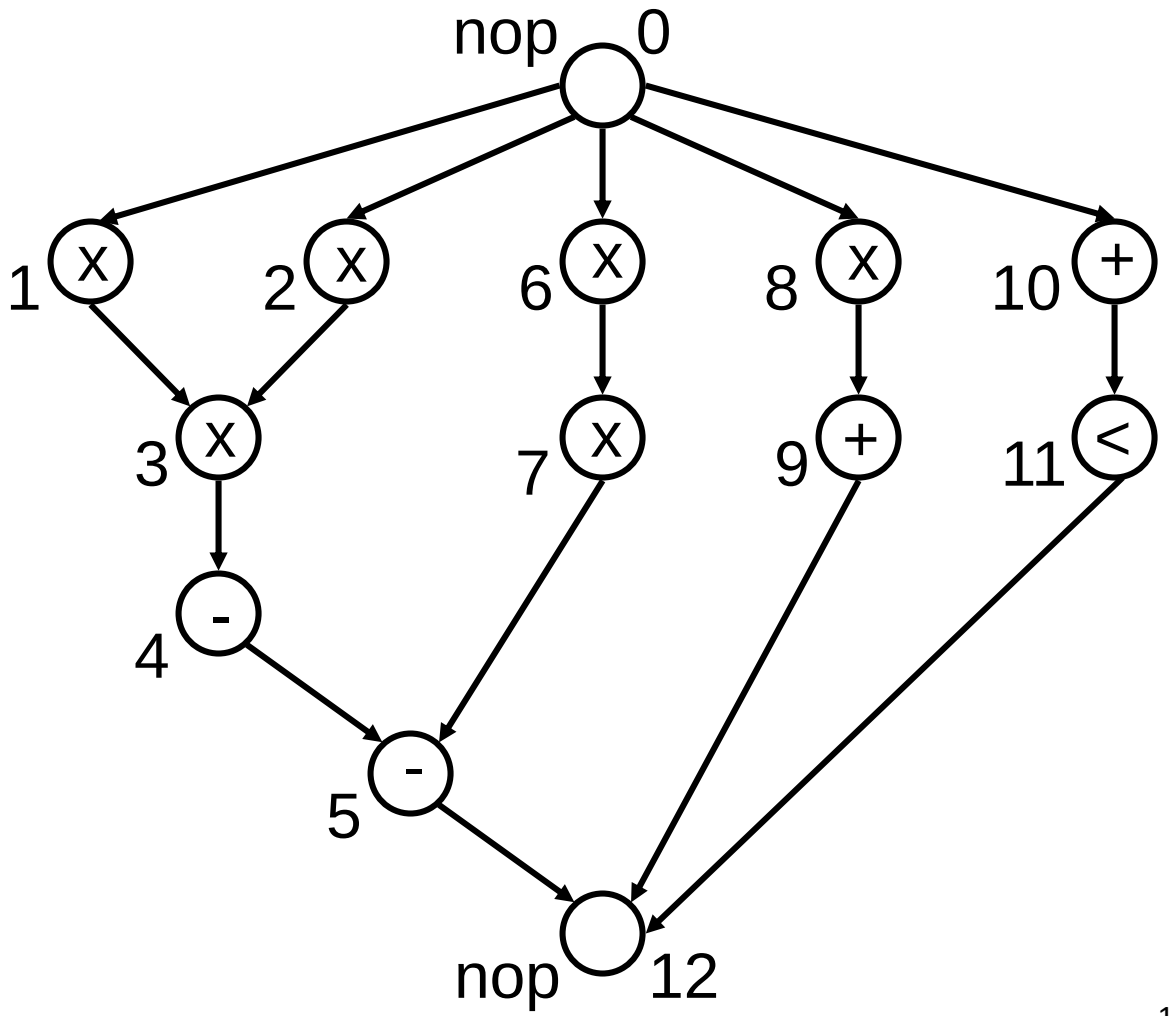
```
int diffeq(int x, int y, int u, int dx, int a) {
  int x1, u1, y1;
  while ( x < a ) {
    x1 = x + dx;
    u1 = u - (3 * x * u * dx) - (3 * y * dx);
    y1 = y + u * dx;
    x = x1;
    u = u1;
    y = y1;
  }
  return y;
}
```

# Models for Architecture Synthesis - Example

- *Corresponding sequence graph:*

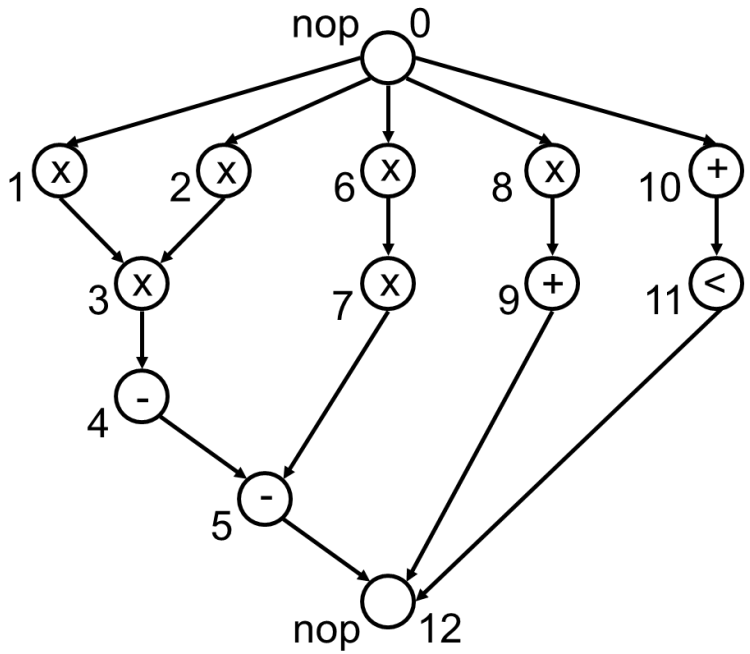
$$G_S = (V_S, E_S)$$

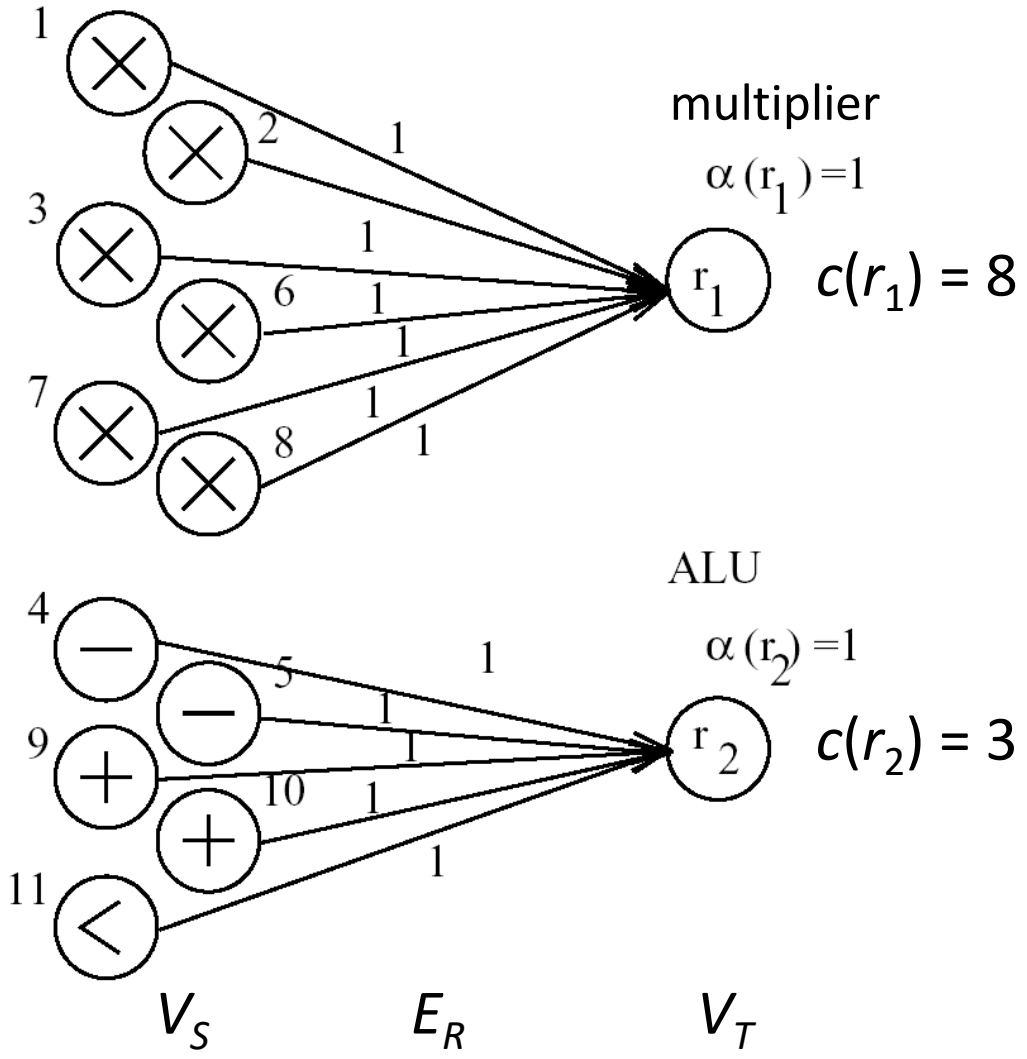
```
int diffeq(int x, int y, int u, int dx, int a) {
  int x1, u1, y1;
  while ( x < a ) {
    x1 = x + dx;
    u1 = u - (3 * x * u * dx) - (3 * y * dx);
    y1 = y + u * dx;
    x = x1;
    u = u1;
    y = y1;
  }
  return y;
}
```



# Models for Architecture Synthesis - Example

- *Corresponding resource graph* with one instance of a multiplier (cost 8) and one instance of an ALU (cost 3):




$$G_S = (V_S, E_S)$$

$$G_R = (V_R, E_R),\ V_R = V_S \cup V_T$$

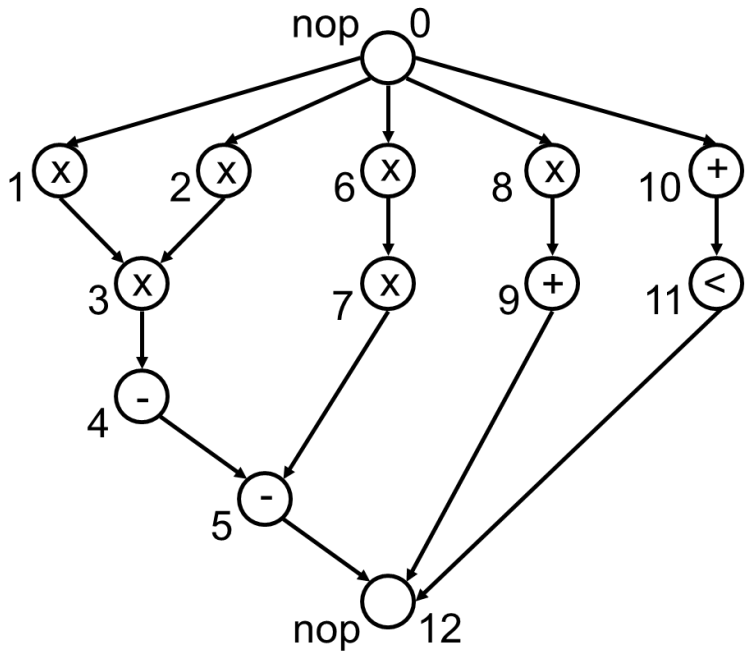
# Allocation and Binding

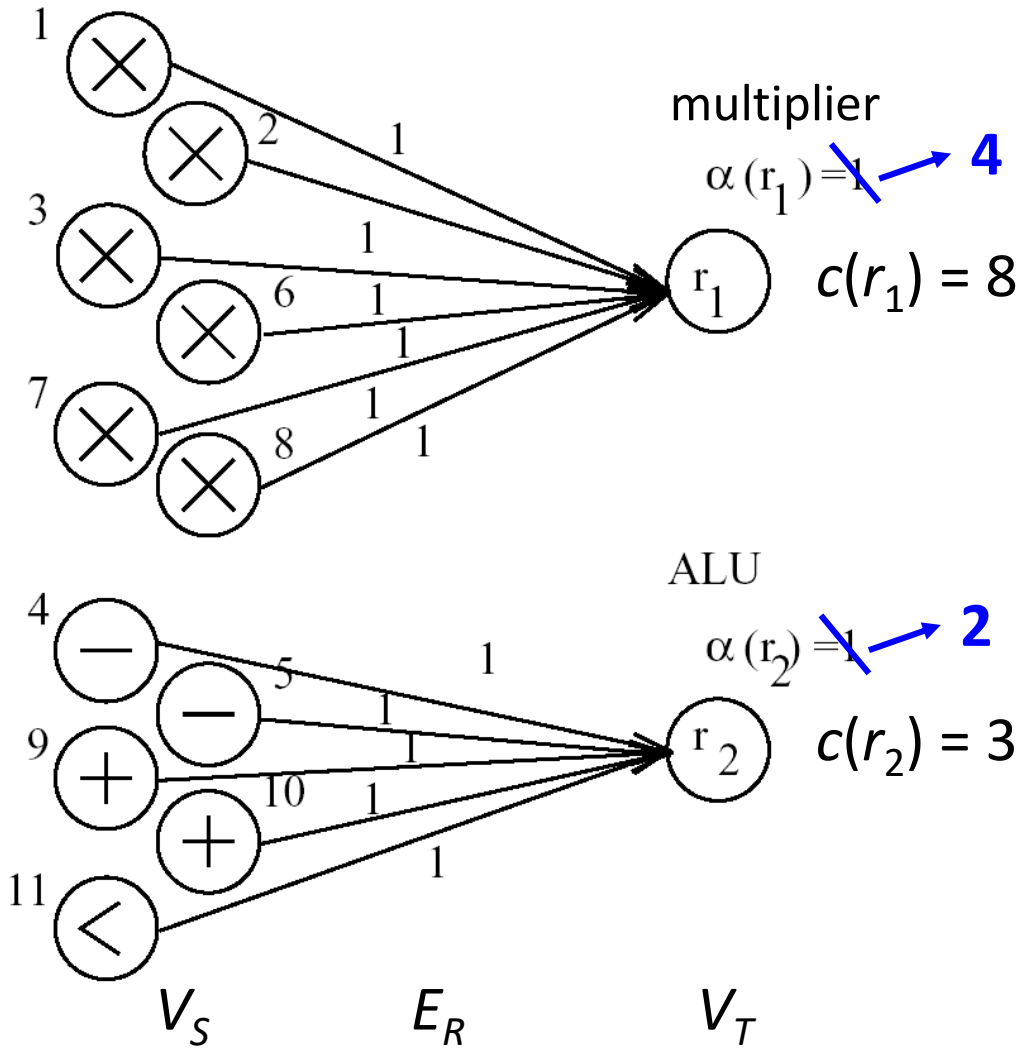
An allocation is a function $\alpha : V_T \rightarrow \mathbf{Z}^{\geq 0}$ that assigns to each resource type $v_t \in V_T$ the number $\alpha(v_t)$ of available instances.

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 $v_t \in V_T$.

# Models for Architecture Synthesis - Example

- *Corresponding resource graph* with 4 instances of a multiplier (cost 8) and two instance of an ALU (cost 3):



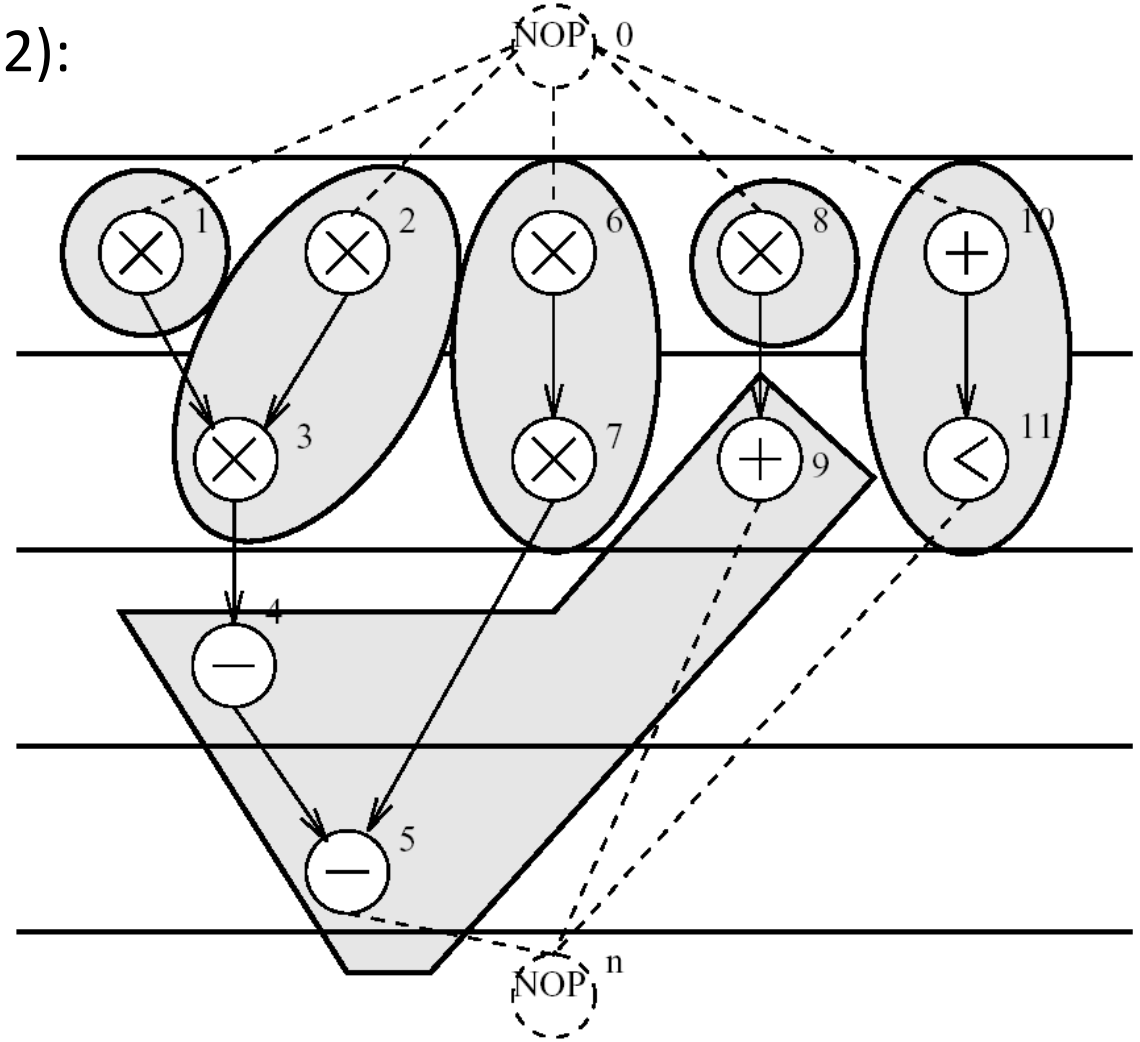

$$G_S = (V_S, E_S)$$

$$G_R = (V_R, E_R),\ V_R = V_S \cup V_T$$

# Models for Architecture Synthesis - Example

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

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



# 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

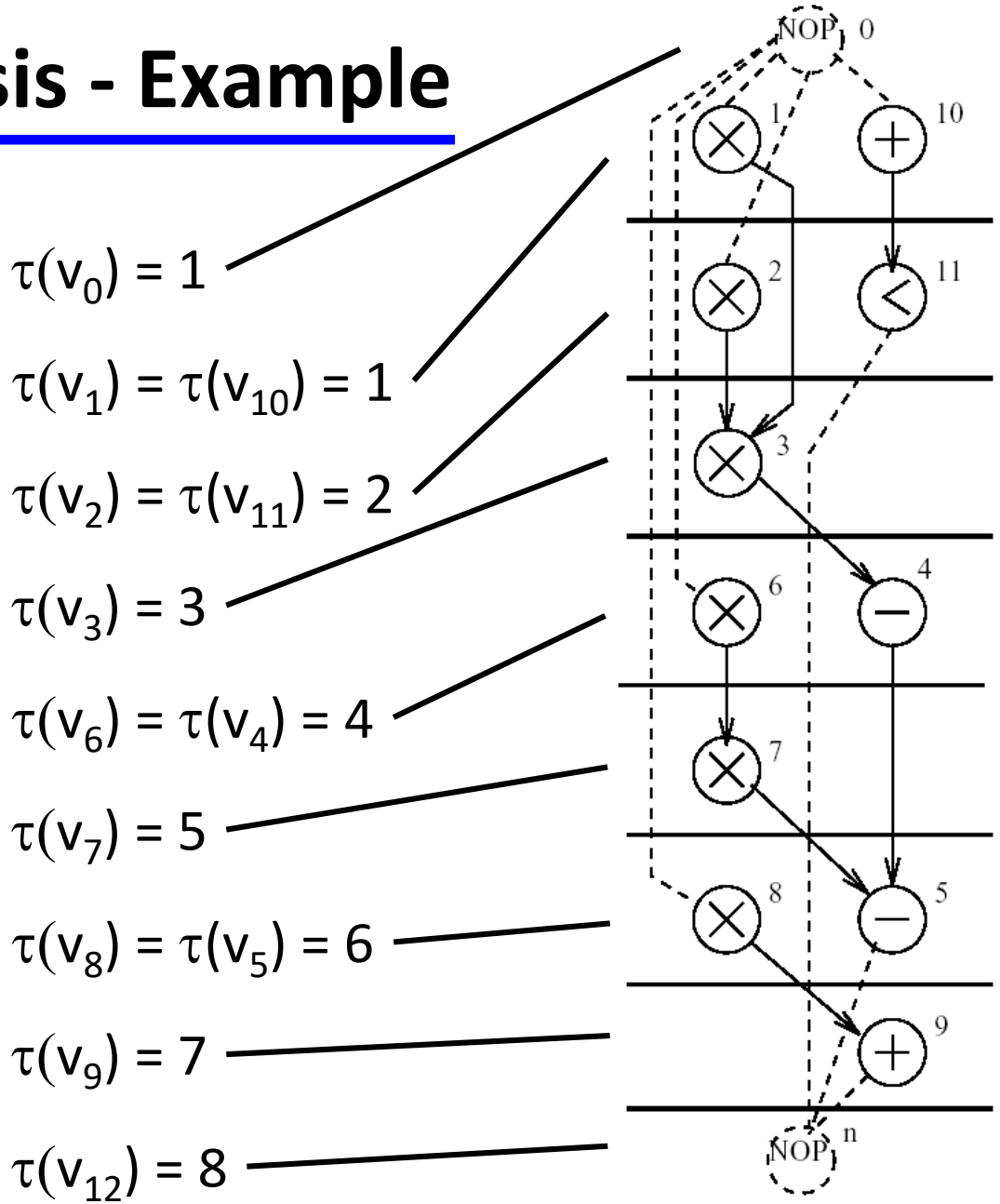
$$\tau(v_j) - \tau(v_i) \geq w(v_i) \quad \forall (v_i, v_j) \in E_S$$

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

# Models for Architecture Synthesis - Example

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



# Multiobjective Optimization

# 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”.

- Synthesis or design problems are typically multiobjective.

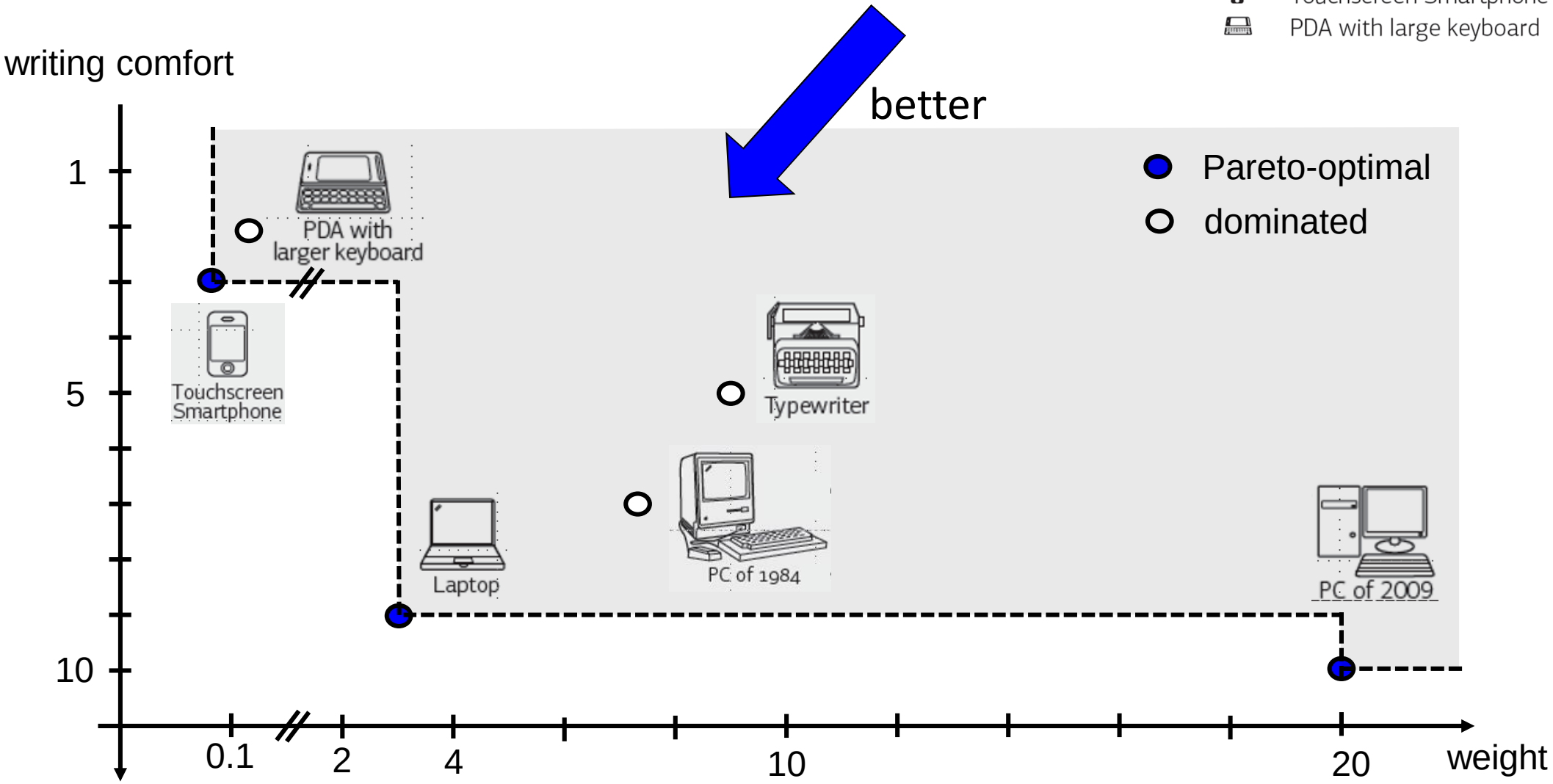
# Multiobjective Optimization

- 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 2020 | 20.00 | 10 |
|  | PC of 1984 | 7.50 | 7 |
|  | Laptop | 3.00 | 9 |
|  | Typewriter | 9.00 | 5 |
|  | Touchscreen Smartphone | 0.09 | 3 |
|  | PDA with large keyboard | 0.11 | 2 |

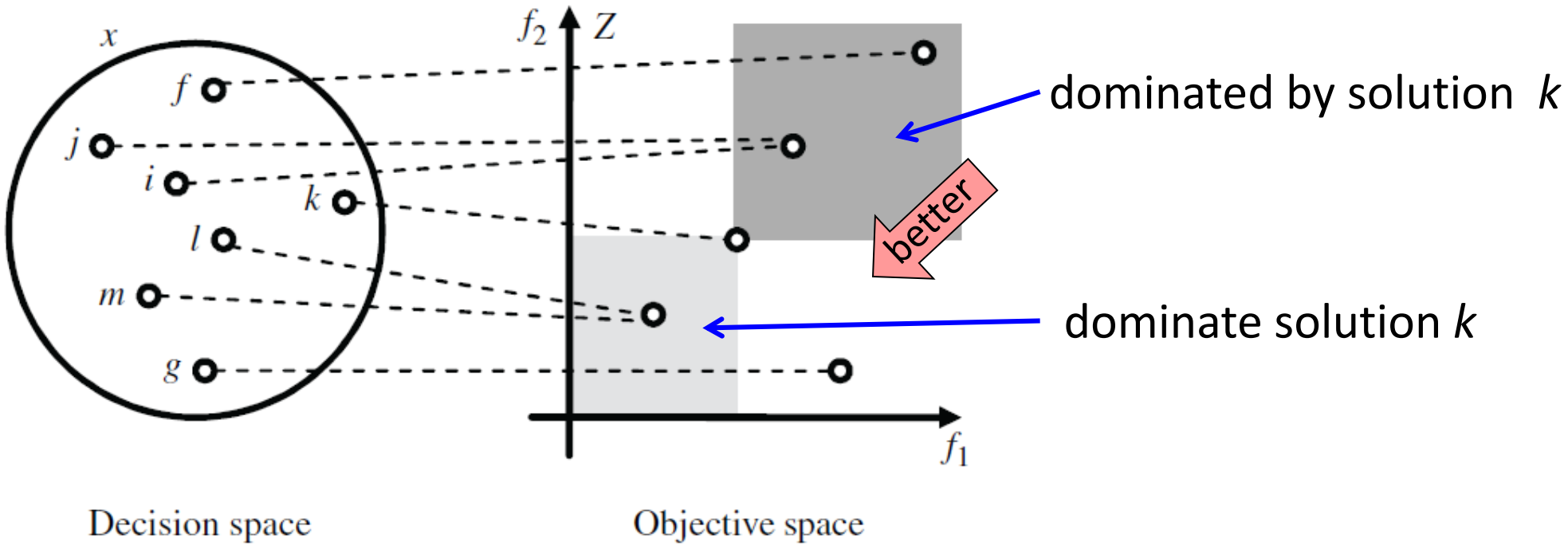
# Multiobjective Optimization

| Icon | Device | weight (kg) | comfort rating |
|---|---|---|---|
| | PC of 2020 | 20.00 | 10 |
| | PC of 1984 | 7.50 | 7 |
| | Laptop | 3.00 | 9 |
| | Typewriter | 9.00 | 5 |
| | Touchscreen Smartphone | 0.09 | 3 |
| | PDA with large keyboard | 0.11 | 2 |



# Pareto-Dominance

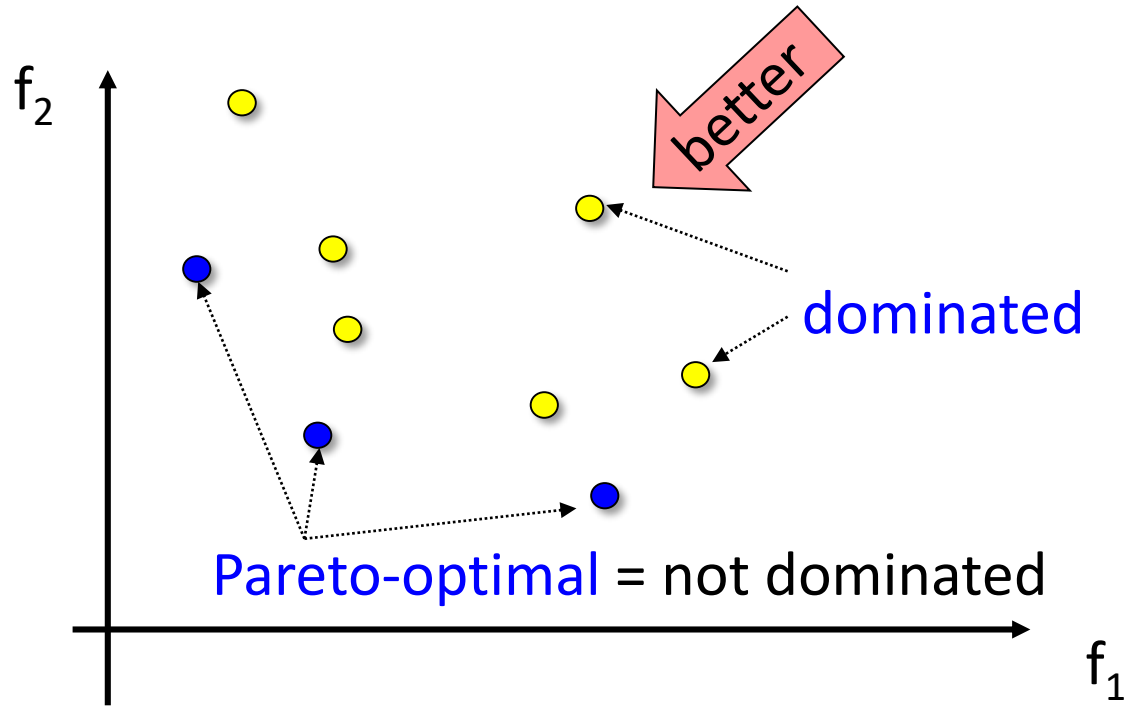
**Definition** : *A solution* $a \in \mathsf{X}$ *weakly Pareto-dominates a solution* $b \in \mathsf{X}$*, denoted as* $a \preceq b$*, if it is as least as good in all objectives, i.e.,* $f_i(a) \leq f_i(b)$ *for all* $1 \leq i \leq n$*. Solution* $a$ *is better then* $b$*, denoted as* $a \prec b$*, iff* $(a \preceq b) \wedge (b \npreceq a)$*.*



# 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*.

objective space **Z:**



# Architecture Synthesis without Resource Constraints

# Synthesis Algorithms

## Classification

- *unlimited resources:*
  - no constraints in terms of the available resources are defined.
- *limited resources:*
  - constrains are given in terms of the number and type of available resources.

## Classes of synthesis algorithms

- *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.

# Synthesis/Scheduling Without Resource Constraints

The corresponding 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
with $\alpha(v_i) \rightarrow \infty$ for all $v_i \in V_T$ is defined as

$$L = \min\{\tau(v_n) - \tau(v_0) \; : \; \tau(v_j) - \tau(v_i) \geq w(v_i, \beta(v_i)) \; \forall (v_i, v_j) \in E_S\}$$

# ASAP Algorithm

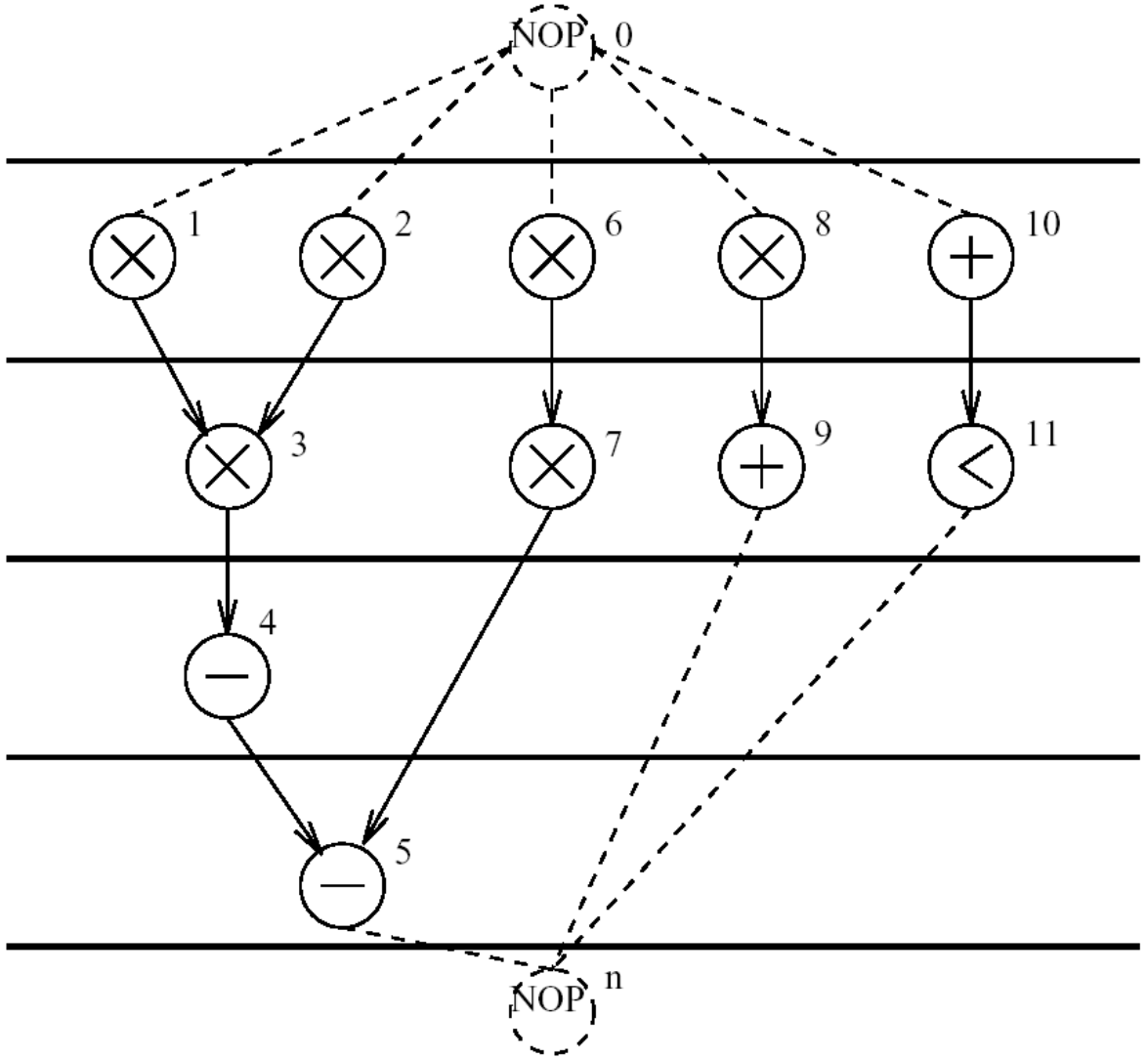
***ASAP = As Soon As Possible***

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

*Example:*

$w(v_i) = 1$



# ALAP Algorithm

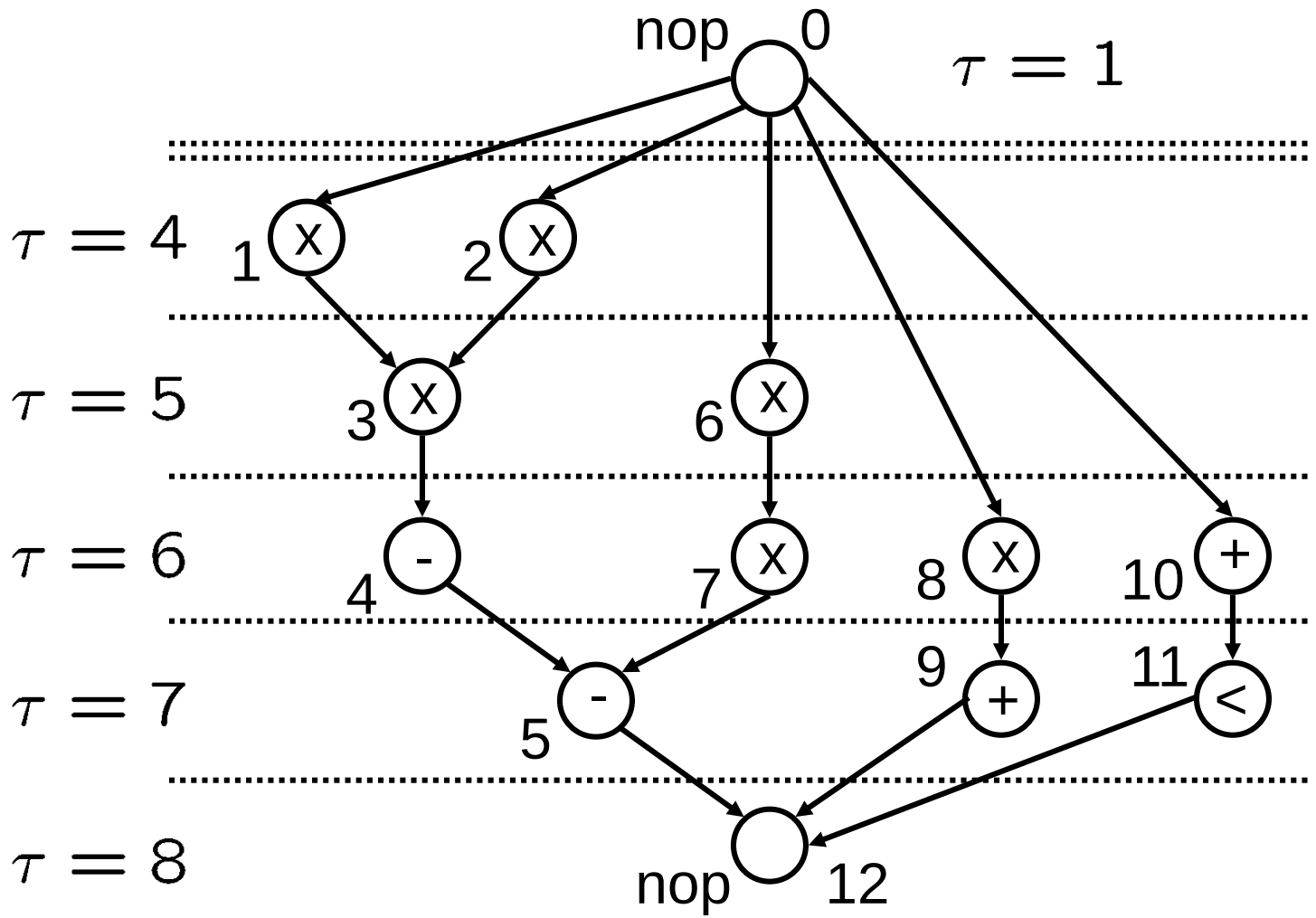
***ALAP = As Late As Possible***

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

# ALAP Algorithm - Example

*Example:*

$L_{max} = 7$
$w(v_i) = 1$



# Scheduling with Timing Constraints

There are 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:*

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

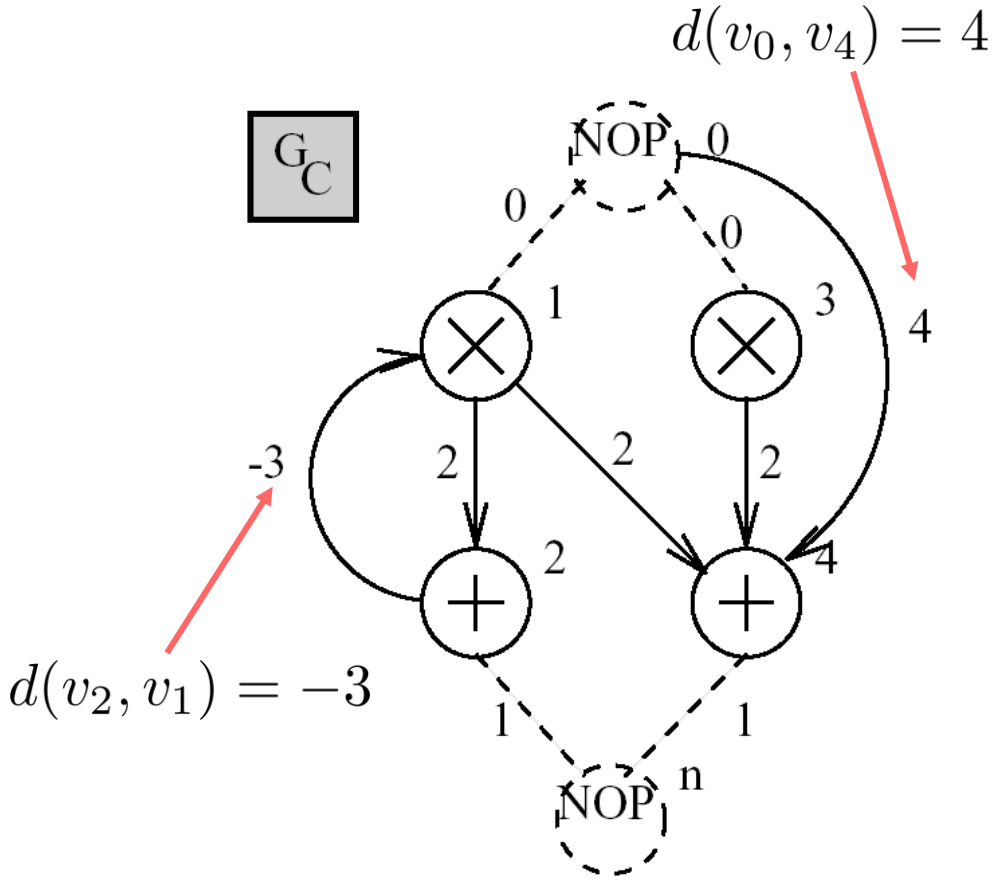
- *Equality constraint:*

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

# 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 $O(|V_C|\ |E_C|)$ ("iterative ASAP"):

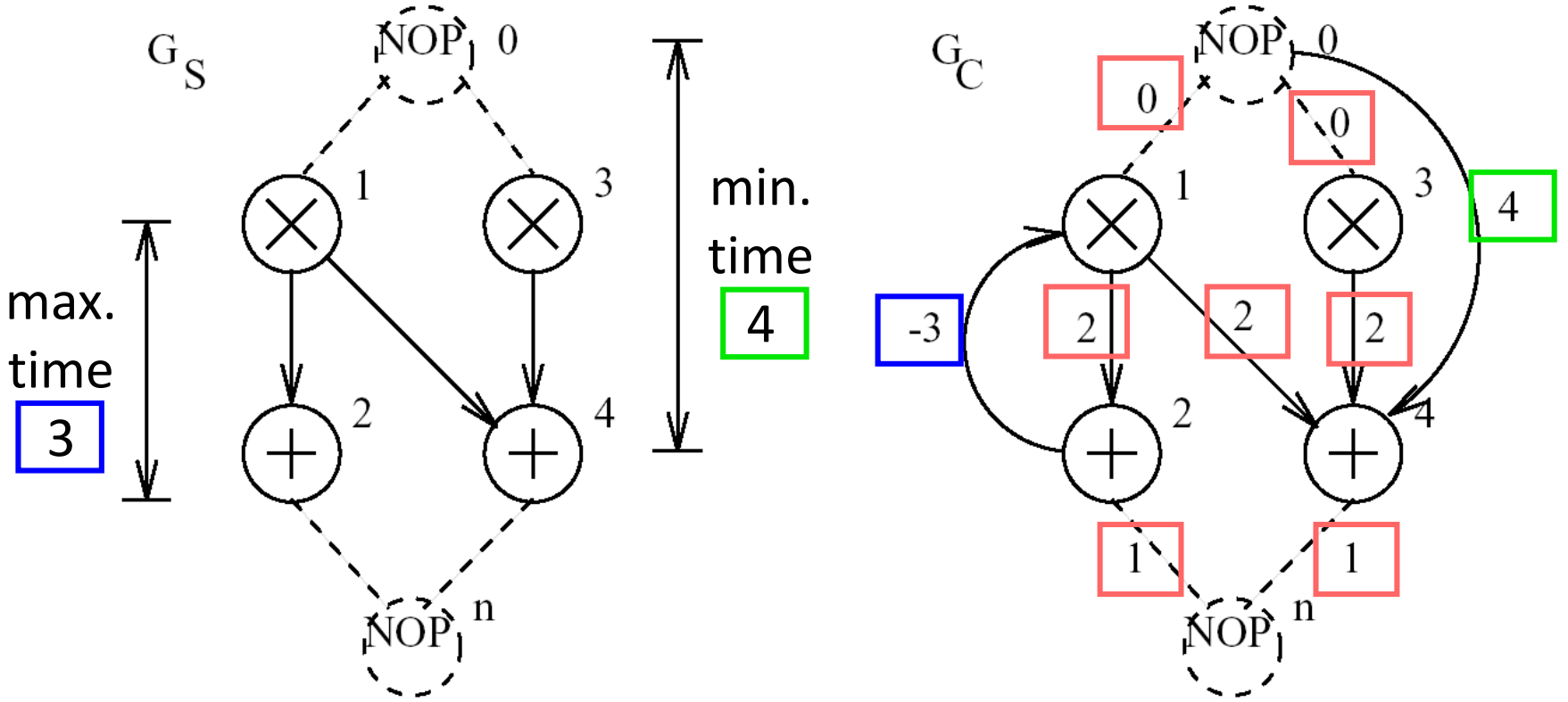
> Iteratively set $\tau(v_j) := \max\{\tau(v_j), \tau(v_i)+d(v_i, v_j) : (v_i, v_j) \in E_C\}$ for all $v_j \in V_C$ starting from $\tau(v_i) = -\infty$ for $v_i \in V_C \backslash \{v_0\}$ and $\tau(v_0) = 1$.

# Weighted Constraint Graph - Example

*Example:*

$w(v_1) = w(v_3) = 2$ $\quad$ $w(v_2) = w(v_4) = 1$

$$\tau(v_0) = \tau(v_1) = \tau(v_3) = 1,\ \tau(v_2) = 3,$$
$$\tau(v_4) = 5,\ \tau(v_n) = 6,\ L = \tau(v_n) - \tau(v_0) = 5$$



# Architecture Synthesis with Resource Constraints

# 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 $\alpha$ and binding $\beta$.

Then the minimal latency is defined as

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

dependencies are respected

there are not more than the available resources in use at any moment in time and for any resource type

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

# List Scheduling

*List scheduling is 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 (b) whose predecessors finished.
- $T_k$ denotes the currently running operations mapped to resource $v_k$ .

# 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 $|S_k| + |T_k| \leq \alpha(v_k)$;
$\tau(v_i) = t \ \forall v_i \in S_k;$ $\}$
$t = t + 1;$
$\}$ UNTIL ($v_n$ planned)
RETURN ($\tau$); $\}$

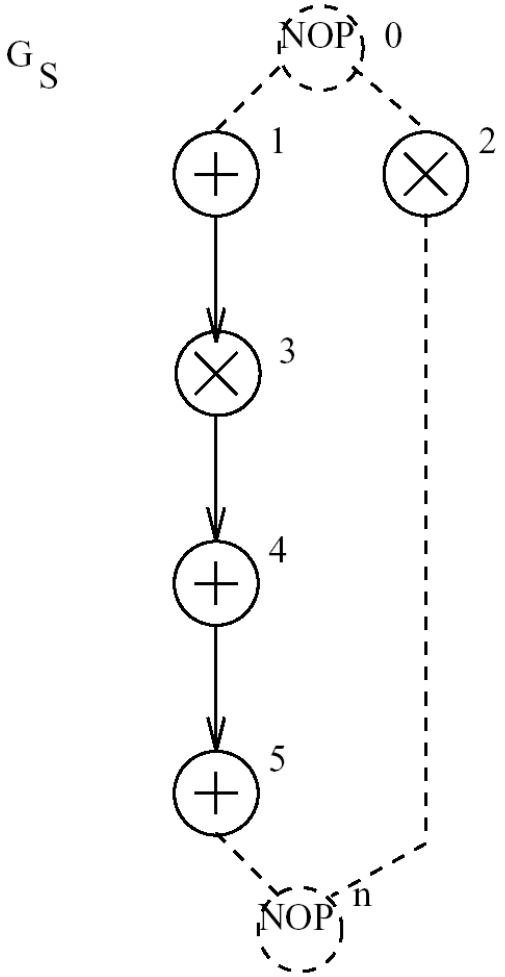
resource types

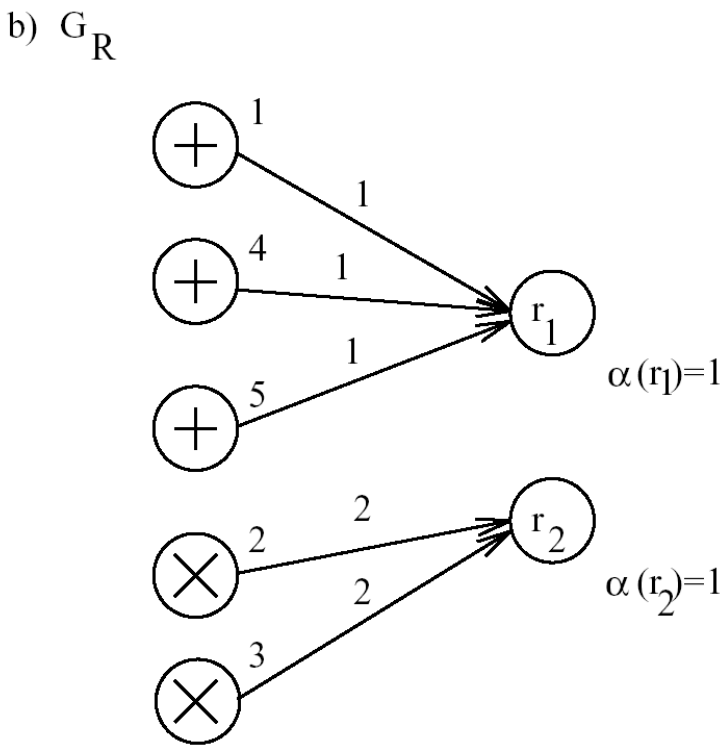
$v \in V_S$ with $\beta(v) = v_k$

# List Scheduling - Example

*Example:*

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 $|S_k| + |T_k| \leq \alpha(v_k)$;
      $\tau(v_i) = t \ \ \forall v_i \in S_k$; }
    $t = t + 1$;
  } UNTIL ($v_n$ planned)
  RETURN ($\tau$); }

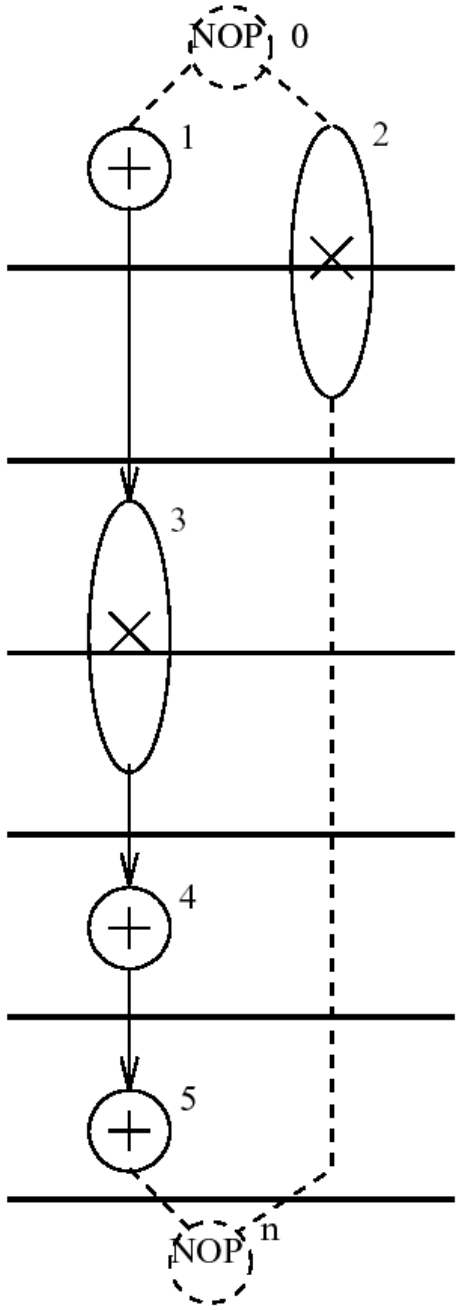


# List Scheduling - Example

*Solution via list scheduling:*

- In the example, the solution is independent of the chosen priority function.
- Because of the greedy selection principle, all resource are occupied in the first time step.
- List scheduling is a heuristic algorithm:
  In this example, it does not yield the minimal latency!

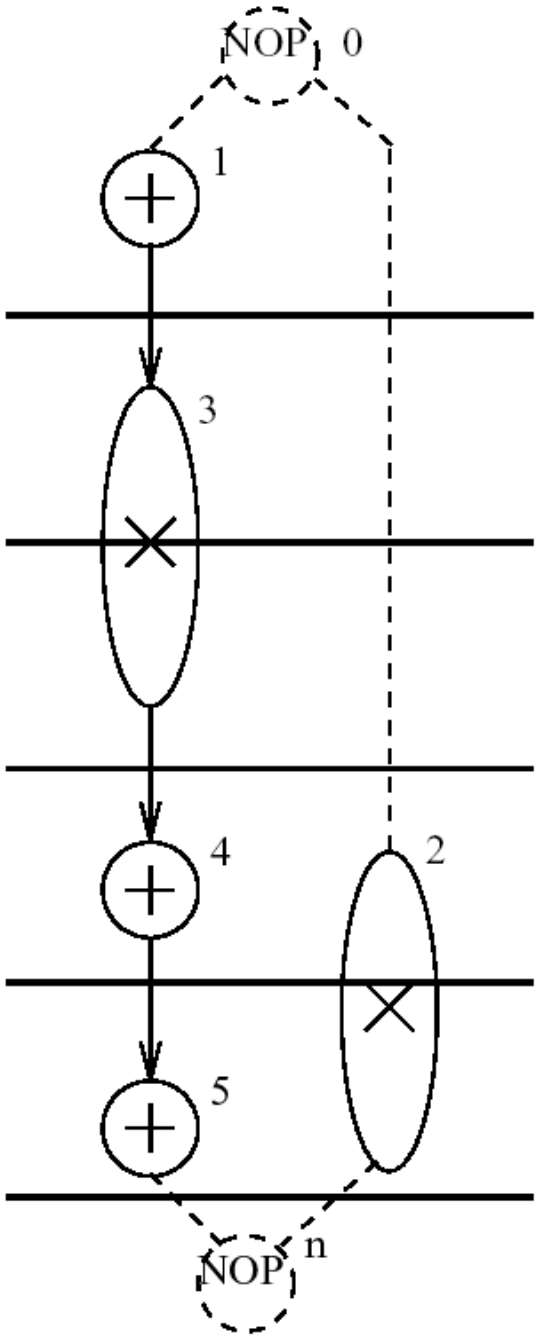


# List Scheduling

*Solution via an optimal method:*

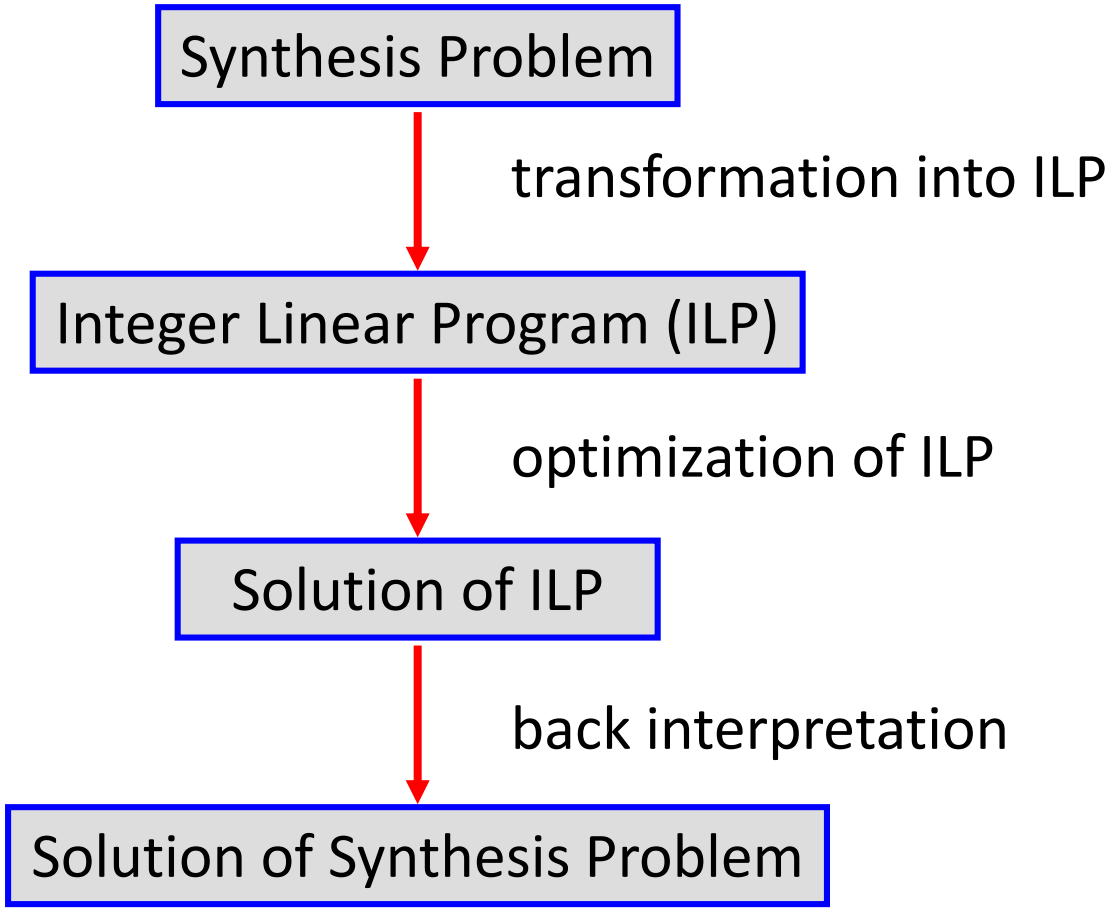
- Latency is smaller than with list scheduling.

- An example of an optimal algorithm is the transformation into an integer linear program as described next.



# 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 higher computational complexity 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Integer Linear Program

*Explanations:*

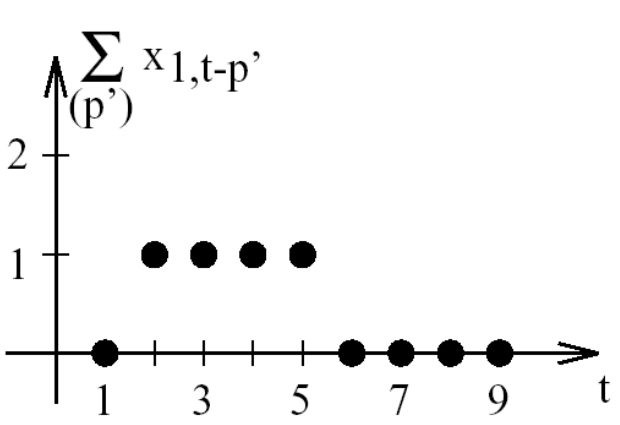
- (1) declares variables x to be binary .
- (2) makes sure that exactly one variable $x_{i,t}$ for all t has the value 1, all others are 0.
- (3) determines the relation between variables $x$ and starting times of operations $\tau$. In particular, if $x_{i,t} = 1$ then the operation $v_i$ starts at time $t$, 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

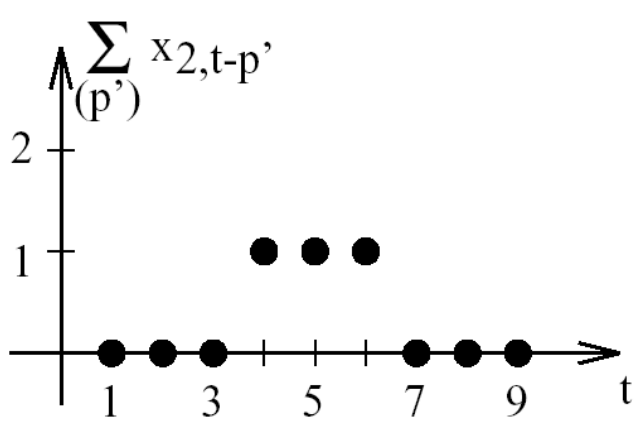
*Explanations:*

- (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 $v_k$ :

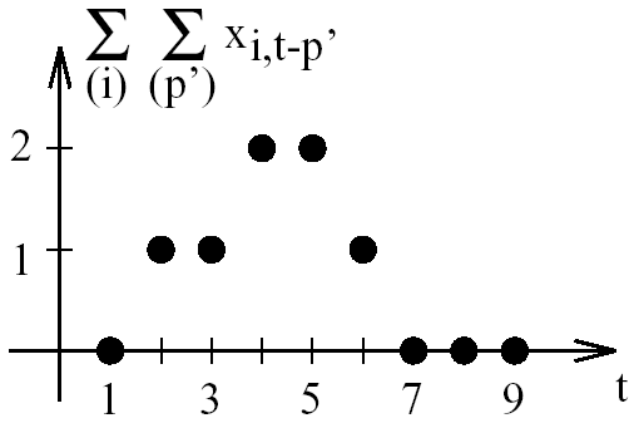
$$\sum_{p'=0}^{w(v_i)-1} x_{i,t-p'} = \begin{cases} 1 & : \quad \forall t : \tau(v_i) \leq t \leq \tau(v_i) + w(v_i) - 1 \\ 0 & : \quad \text{sonst} \end{cases}$$




$w(v_1) = 4\ ,\ \tau(v_1) = 2$




$w(v_2) = 3\ ,\ \tau(v_2) = 4$



# Architecture Synthesis for Iterative Algorithms and Marked Graphs

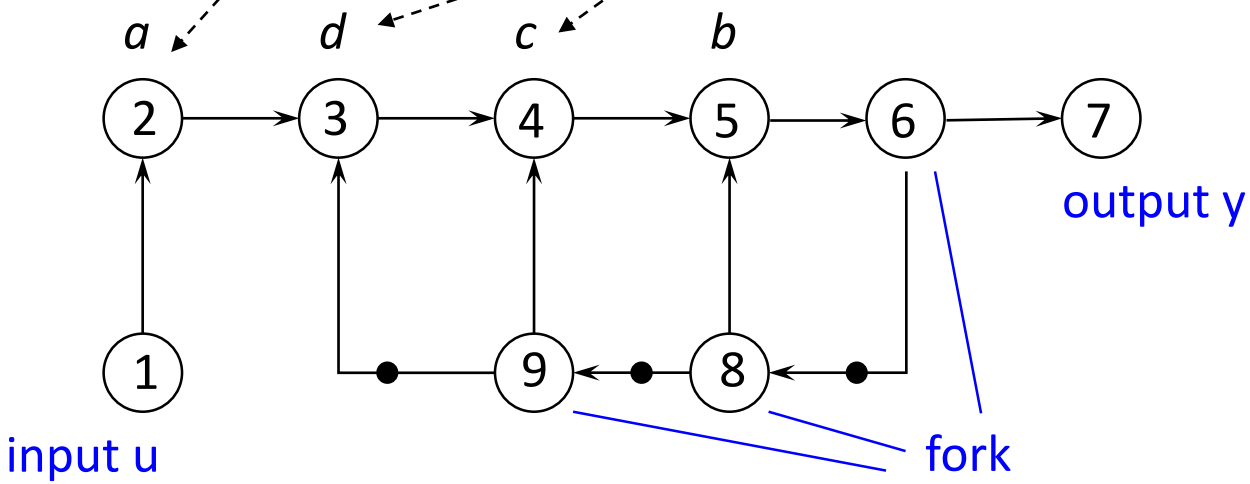
# Remember … : Marked Graph

## Example (model of a digital filter with infinite impulse response IIR)

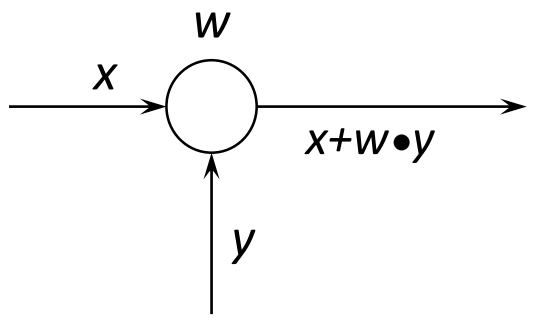
- Filter equation:

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

- Possible model as a *marked graph*:




nodes 3-5:




node 2: *x=0*

# Iterative Algorithms

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

$$x_i[l] = \mathbf{F}_i[\ldots, x_j[l - d_{ji}], \ldots] \quad \forall l \;\; \forall i \in I$$

  Here, $x_i$ denote a set of indexed variables, $\boldsymbol{F}_i$ 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:

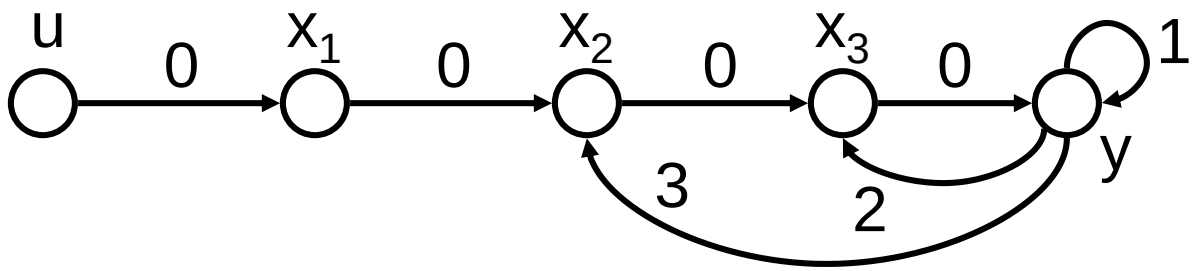
$$y[l] = au[l] + by[l-1] + cy[l-2] + dy[l-3] \quad \forall l$$

- Equivalent set of indexed equations:

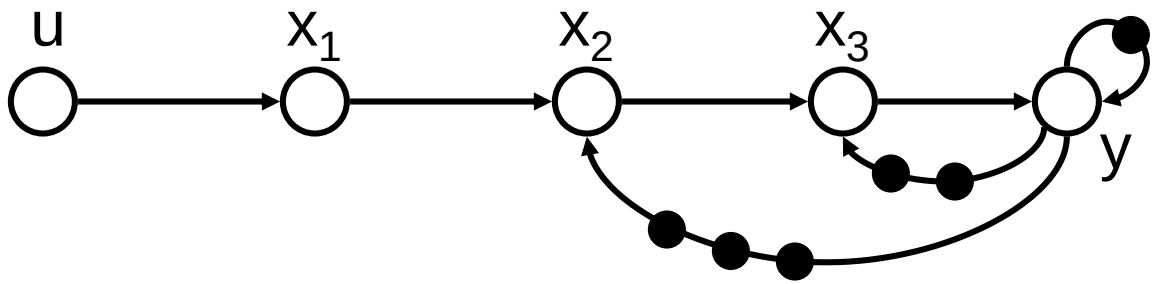
$$\begin{aligned} x_1[l] &= au[l] && \forall l \\ x_2[l] &= x_1[l] + dy[l-3] && \forall l \\ x_3[l] &= x_2[l] + cy[l-2] && \forall l \\ y[l] &= x_3[l] + by[l-1] && \forall l \end{aligned}$$

# 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 $d_{ij}$.

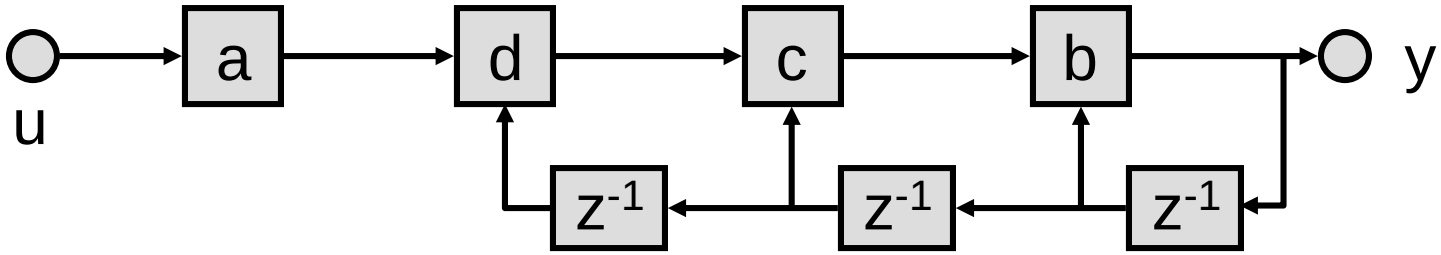



Equivalent *marked graph:*



# 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);}
```

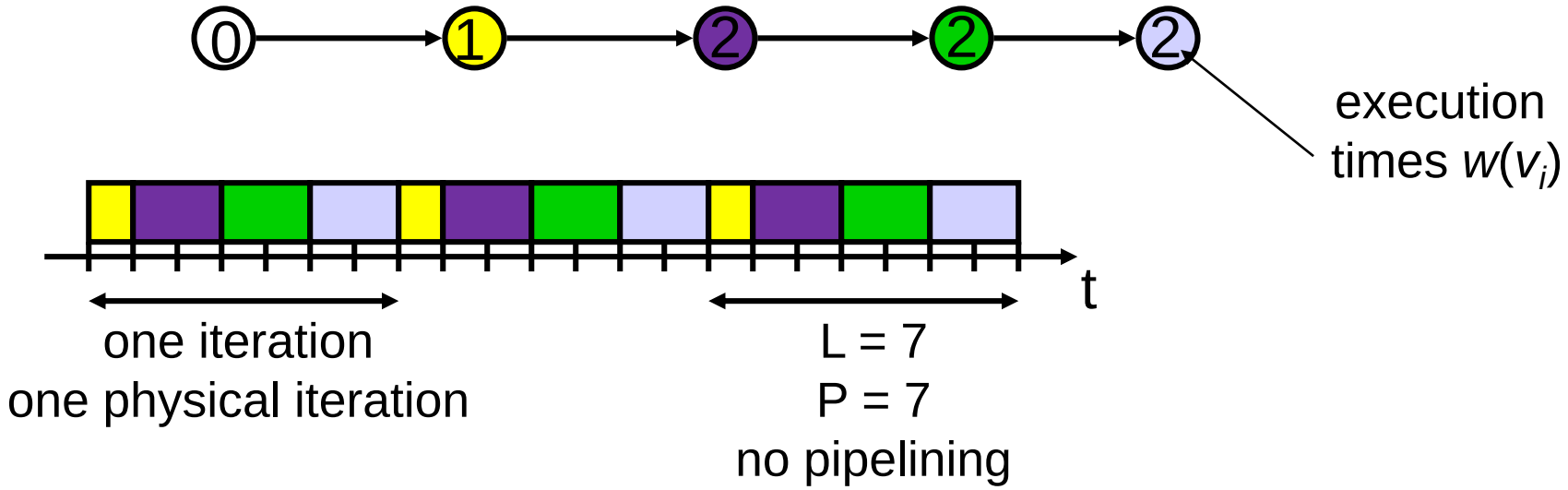
# Iterative Algorithms

- 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 $l$ are executed simultaneously.

# 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:

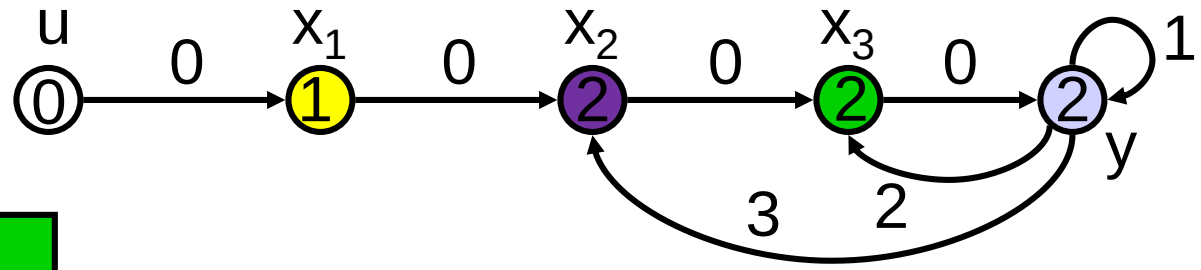


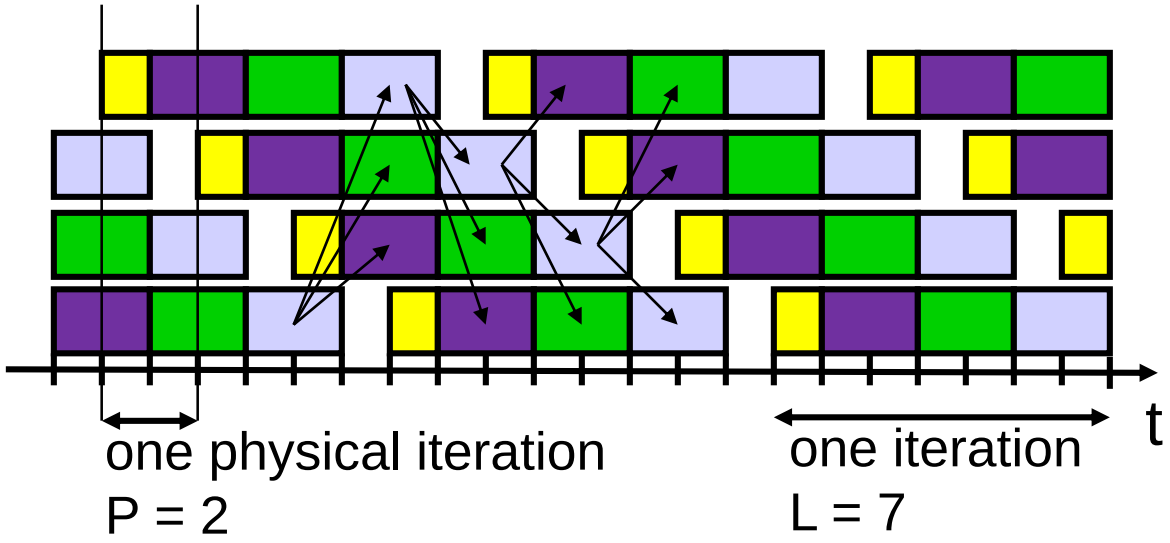
# Iterative Algorithms

## *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!)




- 4 resources
- functional pipelining

# 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.

# Integer Linear Program

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

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

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

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

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

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

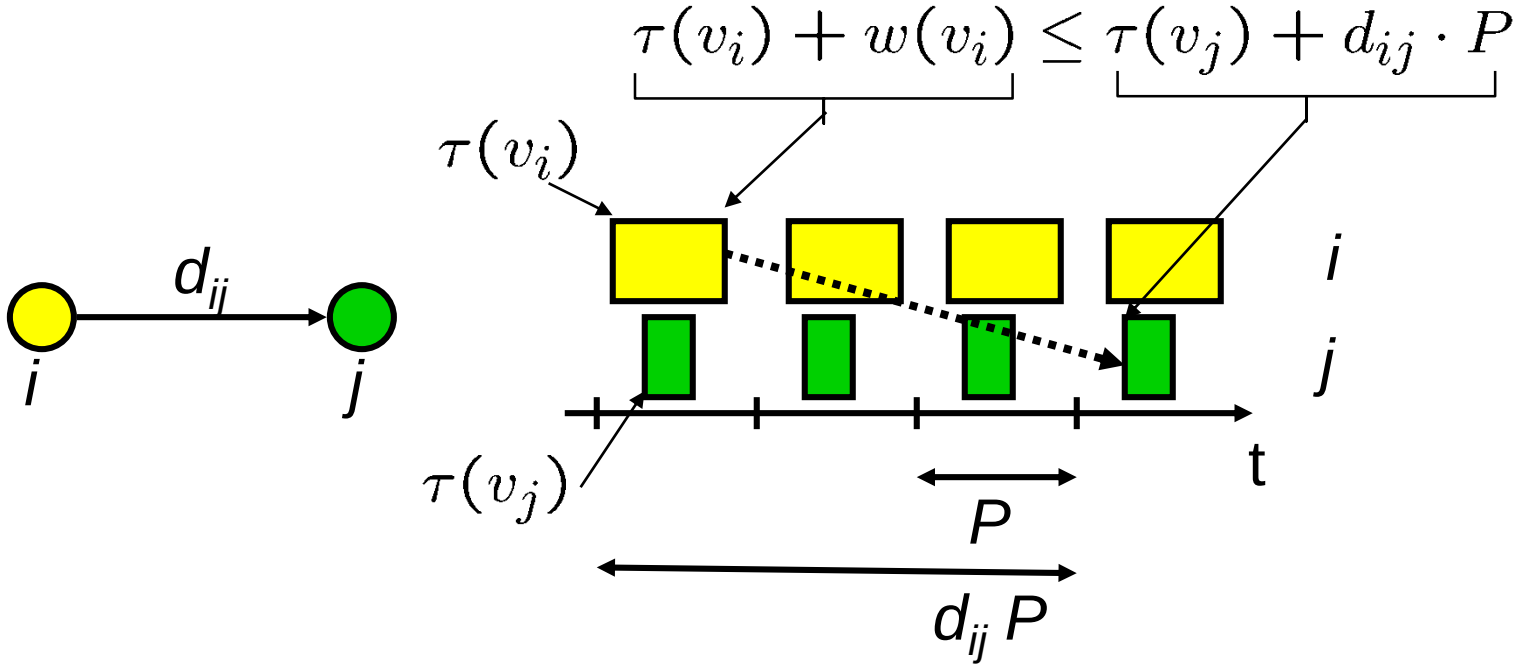
$$\forall v_k \in V_T \;\; \forall t : 1 \leq t \leq \max\{h_i : v_i \in V_S\} \qquad (5)$$

# Iterative Algorithms

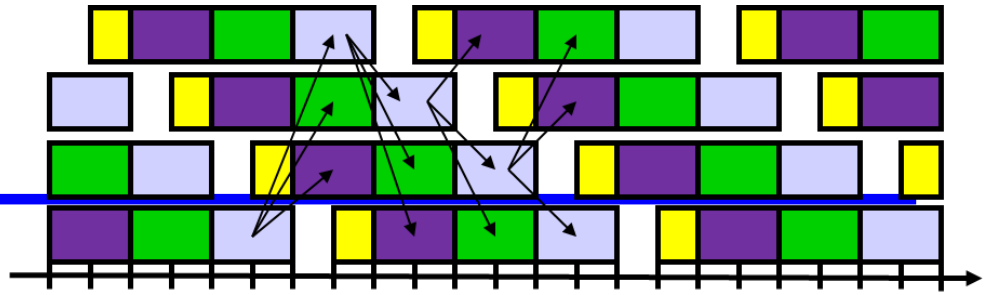
Eqn.(4) is replaced by:

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

*Proof of correctness:*



# Iterative Algorithms



Eqn. (5) is replaced by

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

$$\forall 1 \le t \le P,\ \forall v_k \in V_T$$

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

$$t = \tau(v_i) + p' - p \cdot P$$

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

Therefore, we obtain

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

# 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*.

Shows how one can consider binding in an ILP.

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

minimize: $\sum_{k \in K} \sum_{v_i \in V_S} y_{ik} \cdot e_k(v_i)$

subject to:

$$y_{ik} \in \{0, 1\} \quad \forall v_i \in V_S, k \in K \tag{1}$$

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

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

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

# Dynamic Voltage Scaling

minimize: $\sum_{k \in K} \sum_{v_i \in V_S} y_{ik} \cdot e_k(v_i)$

subject to:

$$y_{ik} \in \{0, 1\} \quad \forall v_i \in V_S, k \in K \tag{1}$$

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

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

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

# Dynamic Voltage Scaling

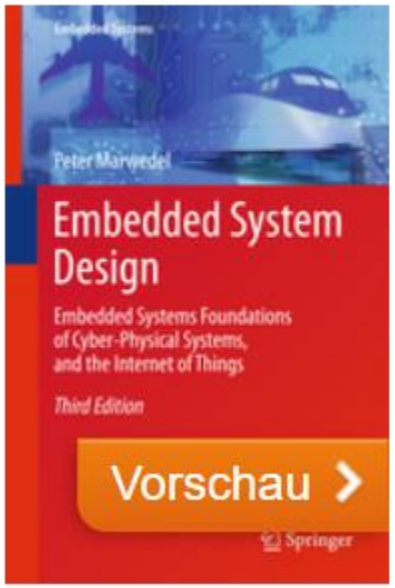
*Explanations*:

- 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.

# Chapter 8

- Not covered this semester.
- Not covered in exam.

- If interested: Read




© 2018

Embedded System Design

Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things

Autoren: **Marwedel**, Peter

» Zeige nächste Auflage

# Remember: What you got some time ago …


# What we told you: Be careful and please do not …


**Return the boards at the embedded systems exam!**