

Introduction to Embedded Systems

1 - Introduction

Prof. Dr. Marco Zimmerling



Organization

Titel ↑	Nummer	Veranstaltungsart	Durchführende/r	Belegwünsche	Zulassungen	Auslastung
▼ <i>Einführung in Embedded Systems / Introduction to Embedded Systems</i>	11LE13Ü-910	Übung				
● Einführung in Embedded Systems - Übung digital (1. Grp.)			■ Zimmerling, Marco	0	134	67% (134 / 200)
▼ <i>Einführung in Embedded Systems / Introduction to Embedded Systems</i>	11LE13V-910	Vorlesung				
● Einführung in Embedded Systems - Vorlesung (-)			■ Zimmerling, Marco	0	155	77% (155 / 200)

Organization

Responsible: Marco Zimmerling (zimmerling@cs.uni-freiburg.de)

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.
- E. A. Lee and S. A. Seshia: *Introduction to Embedded Systems – A Cyber-Physical Systems Approach*, Second Edition, MIT Press, ISBN 978-0-262-53381-2, 2017.

Sources: Slides are based on material from L. Thiele, J. Rabaey, K. Keutzer, M. Wolf, P. Marwedel, P. Koopman, E. Lee, P. Dutta, S. Seshia, and the above-cited books.

What will you learn?

- Requirements and problems arising in embedded systems applications
- Theoretical foundations and principles of the analysis and design of embedded systems

The course has two components:

- **Lectures:** Communicate principles and practical aspects of embedded systems
- **Exercises:** Use paper and pencil to deepen your understanding of analysis and design principles

Lectures and Exercises

Lectures:

- Who: Marco Zimmerling (zimmerling@cs.uni-freiburg.de)
- When: Tuesday, 8:00 – 10:00
- Physical attendance: Georges-Köhler-Allee 101, HS 00-036 (in English)
- Virtual attendance: Live streaming via Zoom, recordings

Exercises:

- Who: Jürgen Mattheis (matthejue@gmail.com)
Pascal Walter (pascal.walter2008@web.de)
- When: Tuesday, 12:00 – 14:00
- Physical attendance: Georges-Köhler-Allee 101, HS 00-036 (in **English**)
Georges-Köhler-Allee 101, Kinohörsaal (in **German**)
- Virtual attendance: Live streaming via Big Blue Button, recordings

Join the Course on ILIAS!

Access via: <https://nes-lab.org/>

- Courses
- Introduction to Embedded Systems
- ILIAS
- Login: RZ username + password
- Course password: **es-0x8af**



Resources:

- Forum, course schedule, Zoom/BBB links, important announcements
- Slides and recording **after** each lecture
- Exercise sheets **before** each exercise
- Slides, recordings, and exercise solutions **after** each exercise

Exercises: Modus Operandi

Before the exercise:

- Download exercise sheet and recap corresponding lecture material
- Bring paper and pencil

During the exercise:

- Teaching assistant briefly summarizes the lecture material required to solve the exercise questions and provides a few hints on how to approach a solution
- For each exercise question:
 - Solve it yourself and ask questions
 - Teaching assistant presents correct solution

After the exercise:

- Check out exercise solution and recording
- Further questions? → Forum → Teaching Assistants

Schedule and Exam

Schedule:

- Today (October 18): **No exercise**
- Next week (October 25): Lecture and **maybe** no exercise
- In two weeks (November 1): **All Saints' Day**
- **Check regularly on ILIAS for updates!**

Exam:

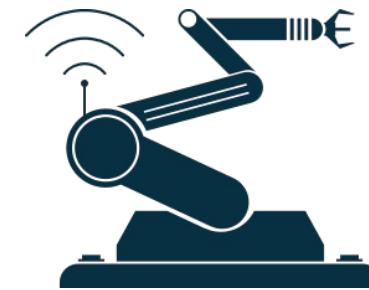
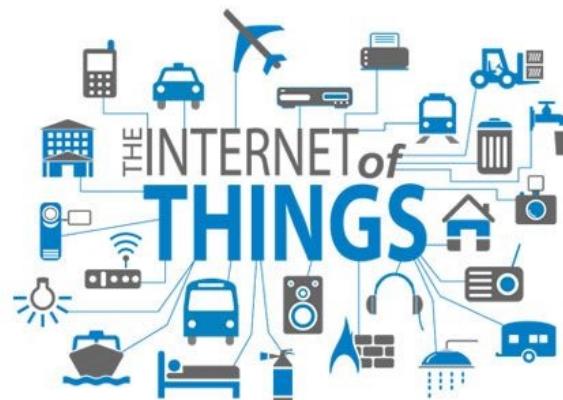
- Format: Written, 120 minutes
- Language: Questions in both English and German
- When and where: TBD (between February 13 and March 31, 2023)
- **We will discuss an example exam in the last exercise session!**

Embedded Systems - Impact

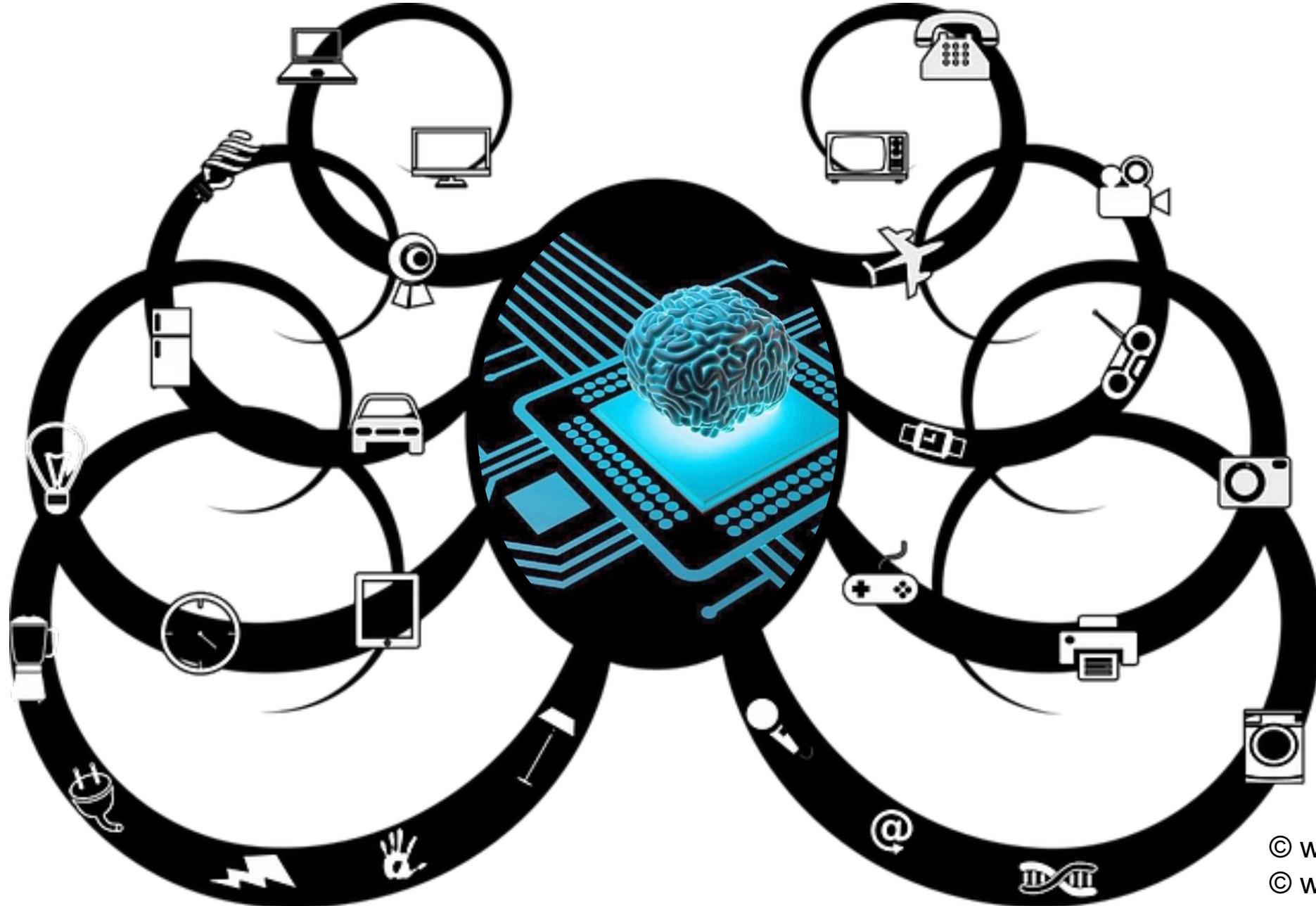
Embedded Systems

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

Examples:

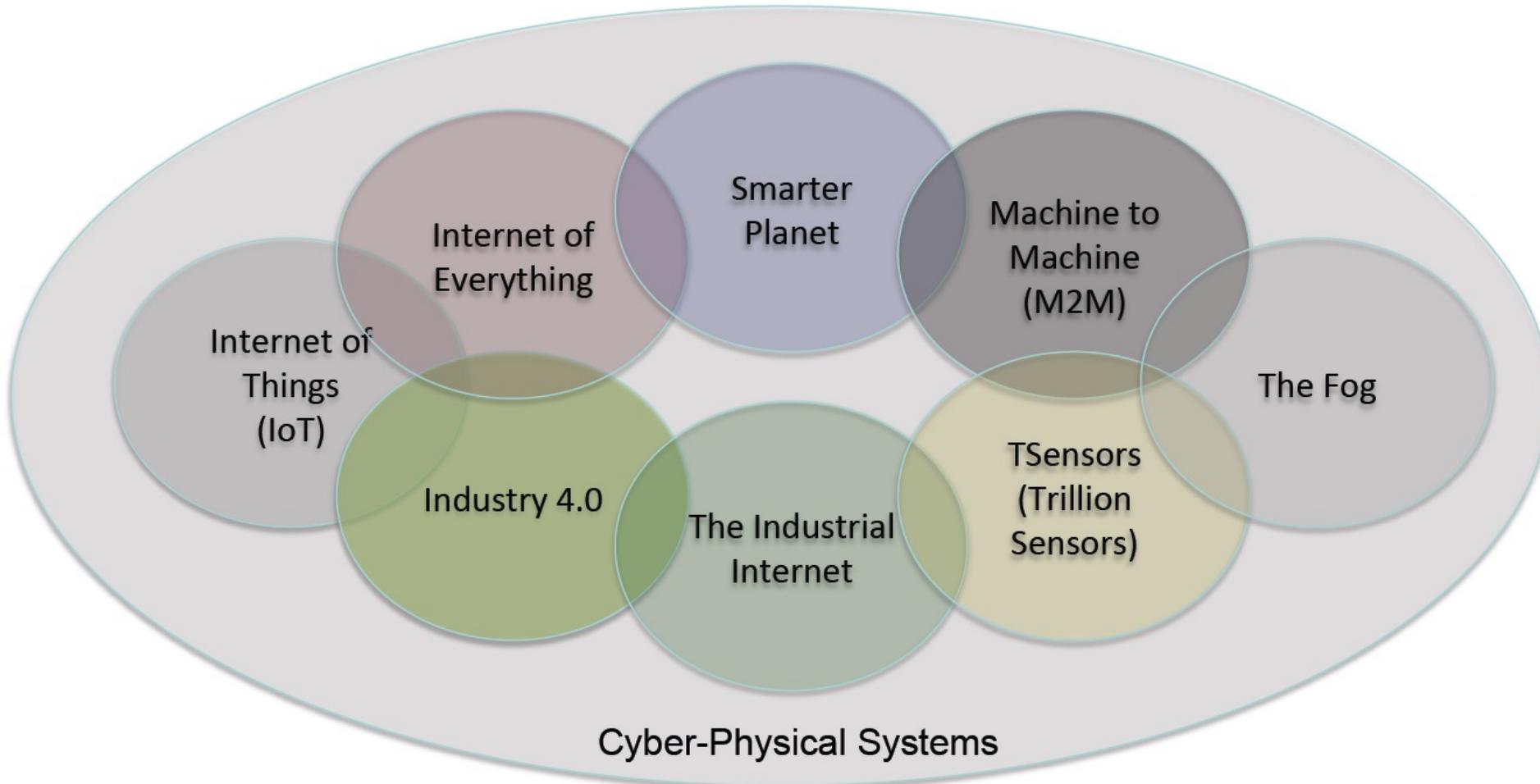


Often, the main reason for buying is not information processing



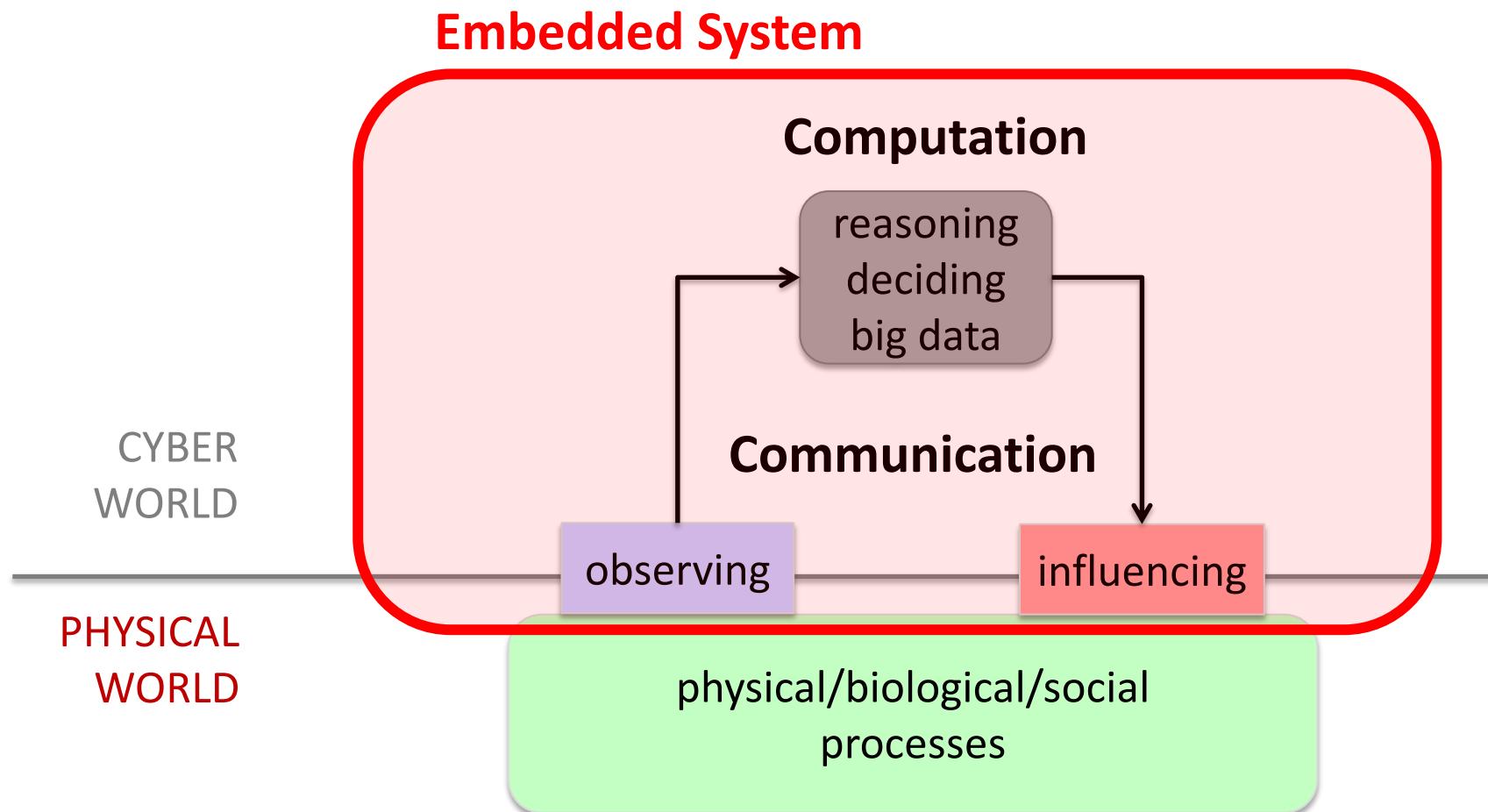
© www.braingrid.org
© www.openpr.com

Many Names – Similar Meanings



© Edward Lee

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 :

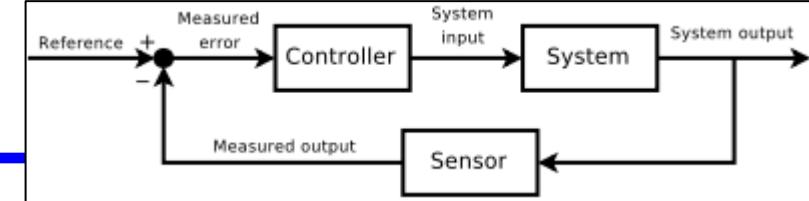
„A reactive system is one which is in continual interaction with its environment and executes at a pace determined by that environment“ [Bergé, 1995]

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

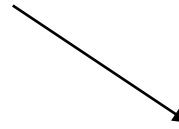
- For hard real-time systems, right answers arriving too late are wrong. All other time-constraints are called soft. A **guaranteed system response** has to be explained without statistical arguments.

„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}

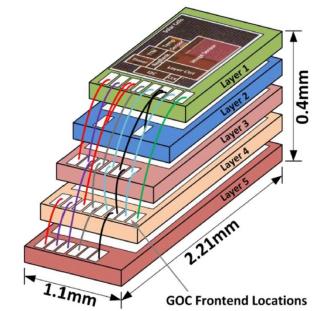
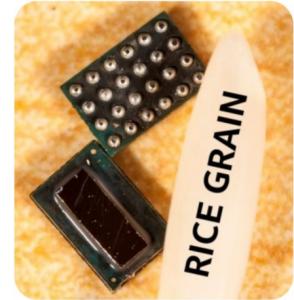
“CPS must *operate dependably*, safely, securely, efficiently and in real-time.”^{Raj10}

^{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

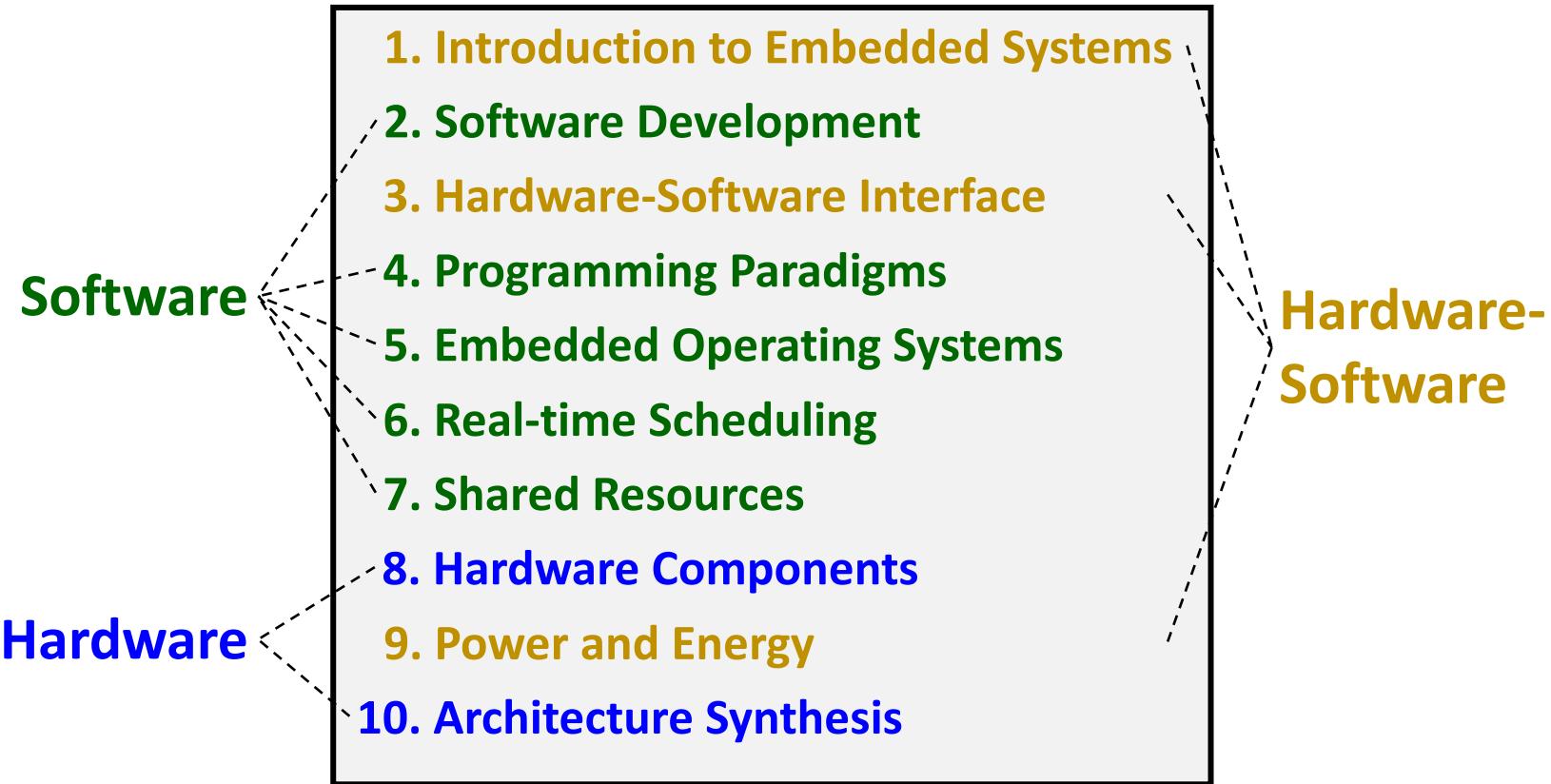
Embedded Systems:

- 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 -



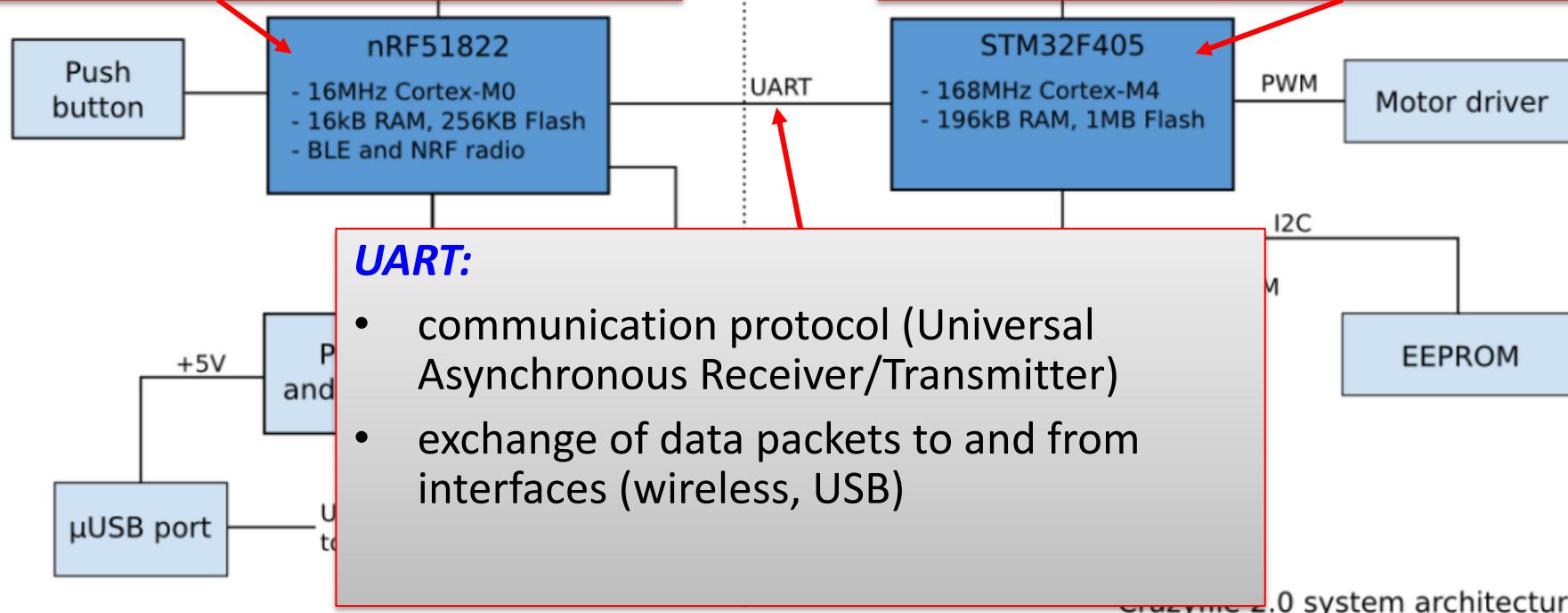
High-Level Block Diagram View

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

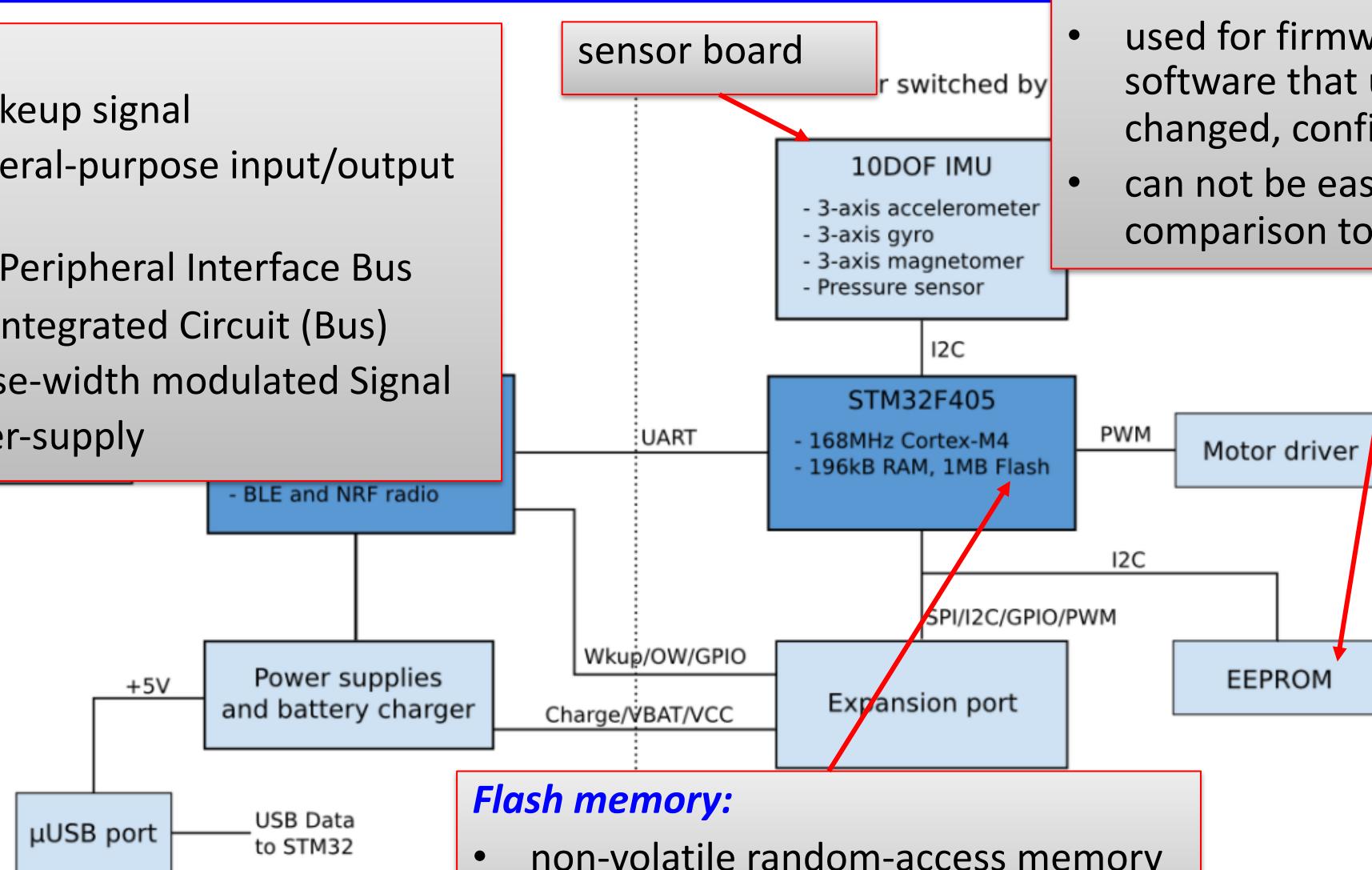
- sensor reading and motor control
- flight control
- telemetry (including the battery voltage)
- additional user development
- USB connection



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



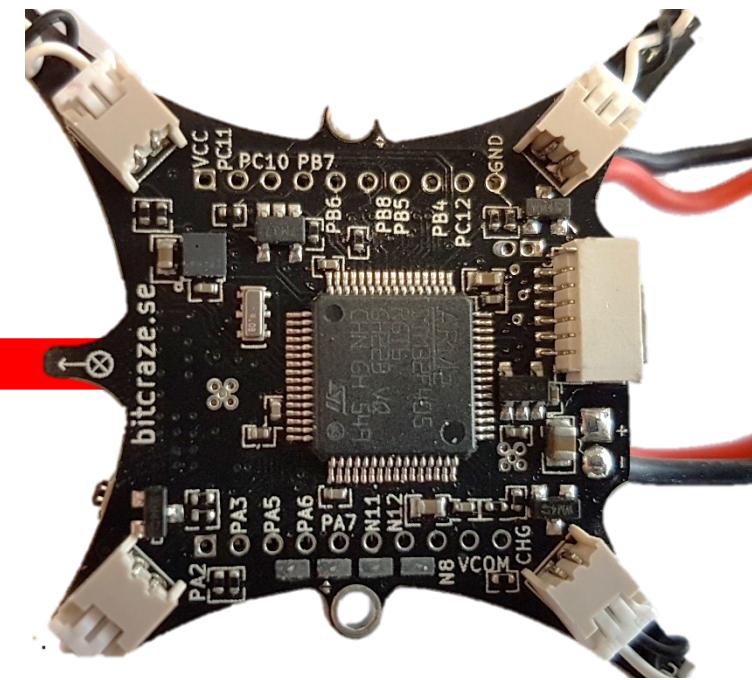
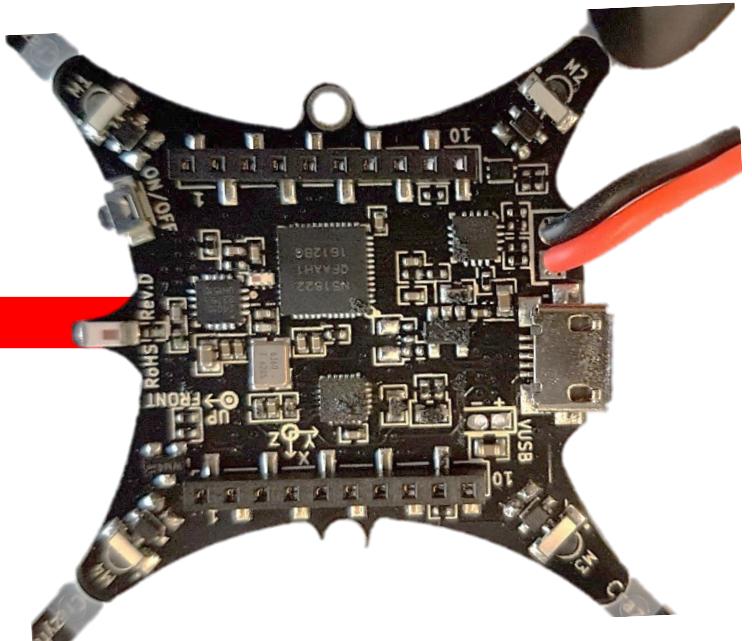
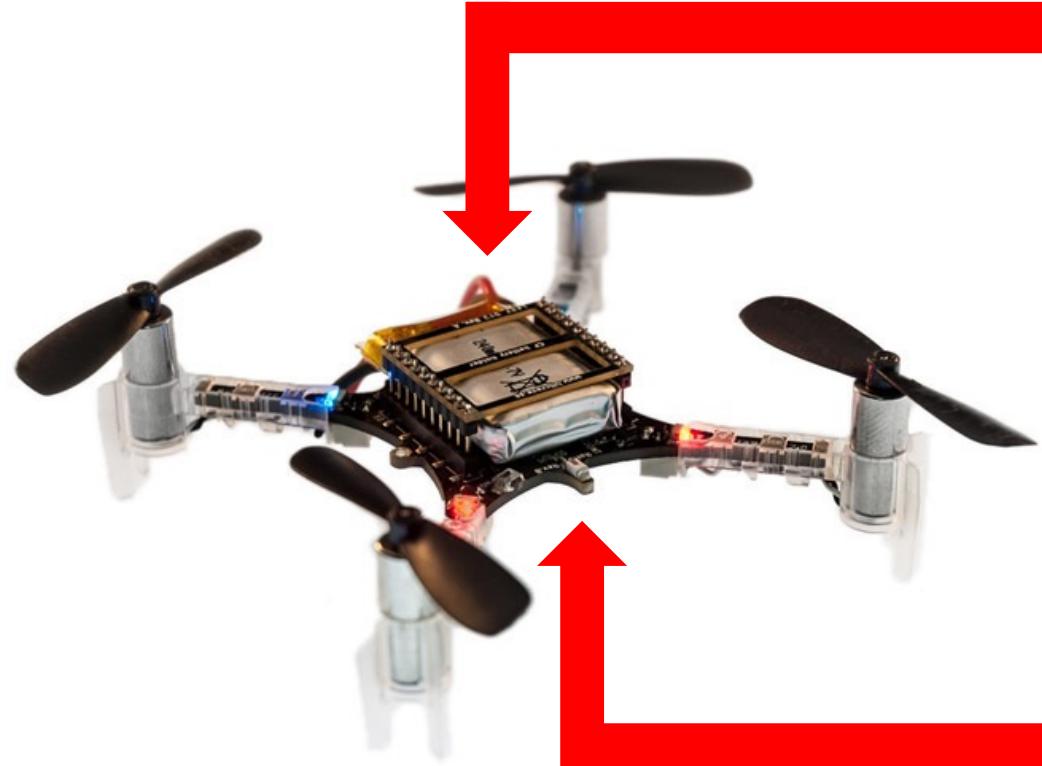
EEPROM:

- 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

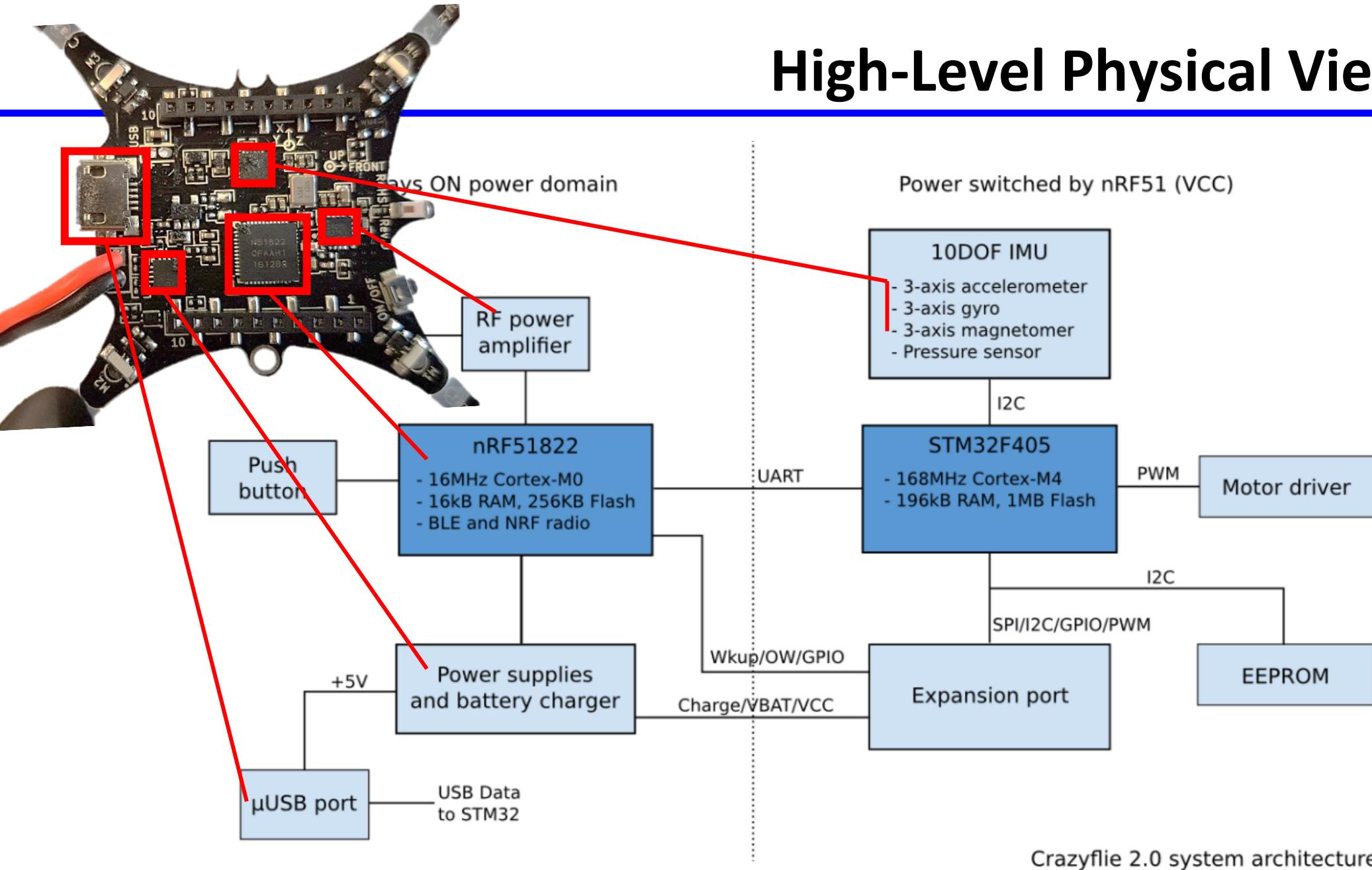
Flash memory:

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

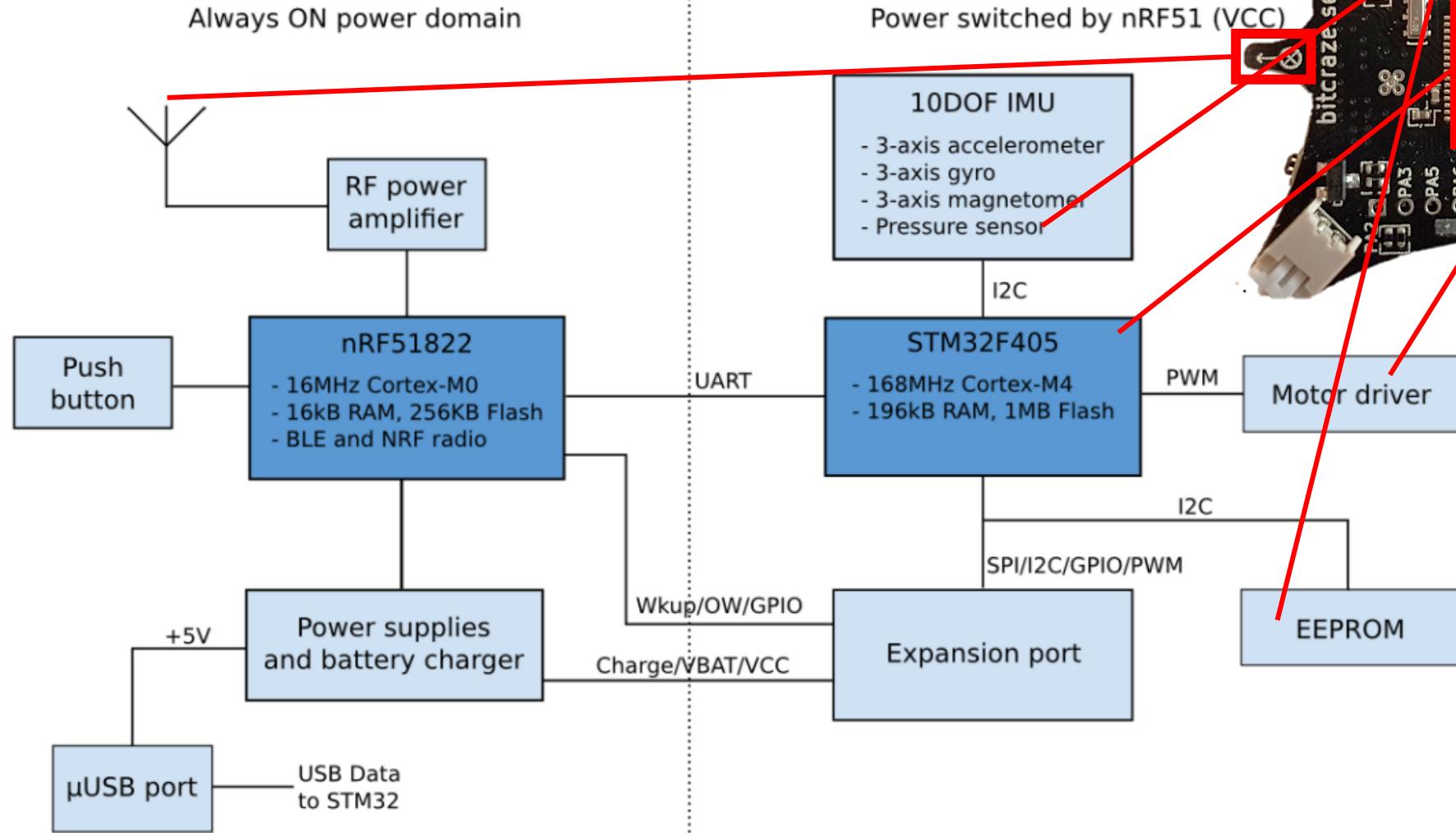
System architecture



High-Level Physical View

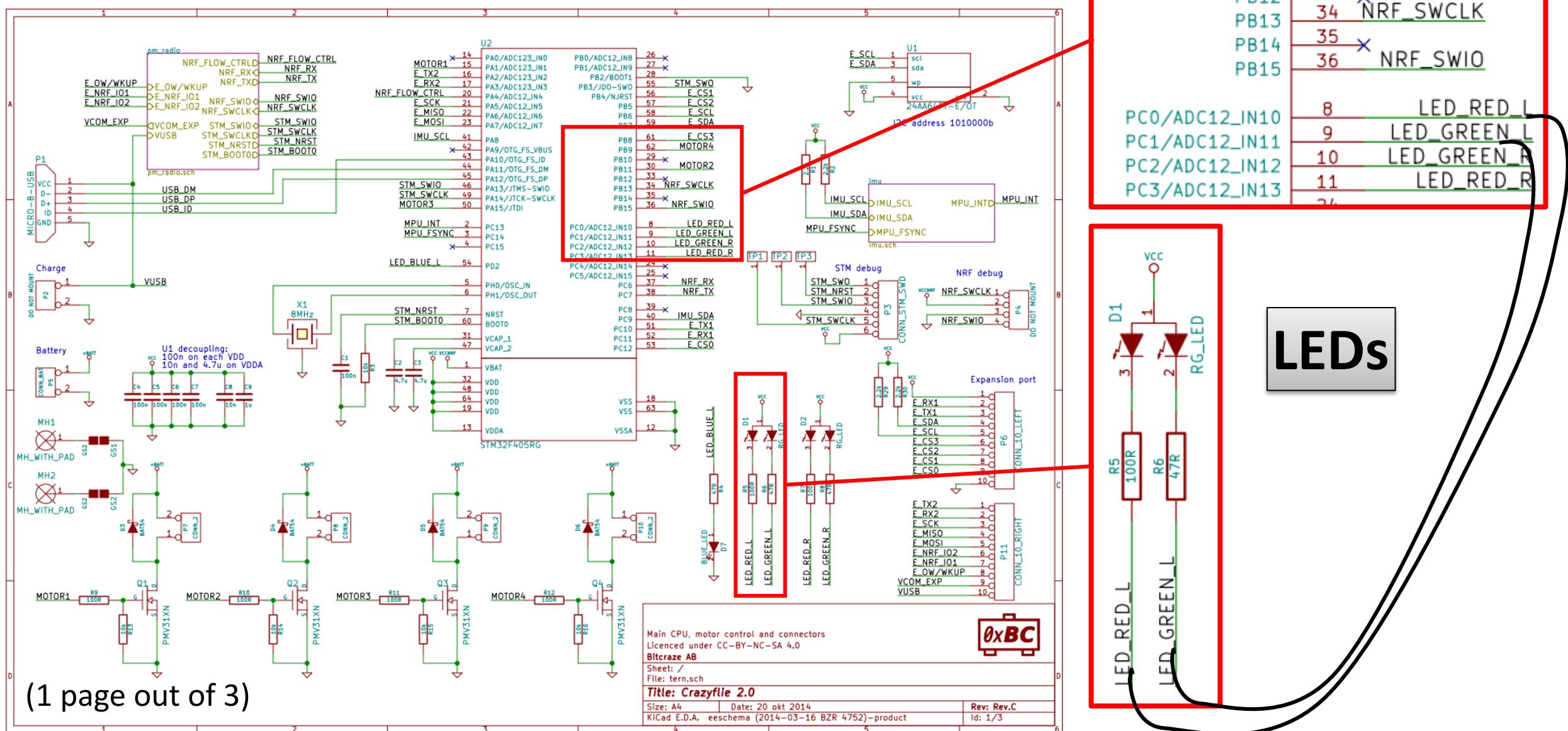


High-Level Physical View



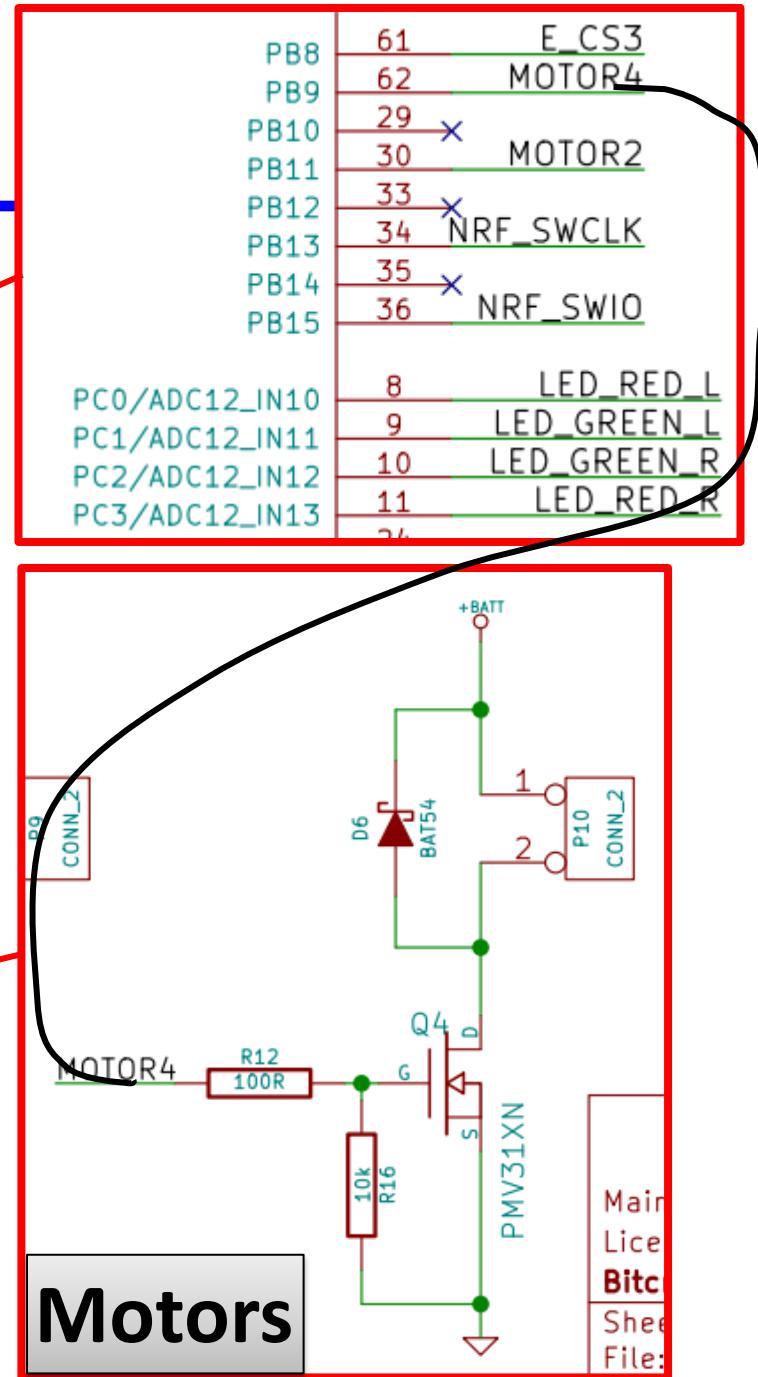
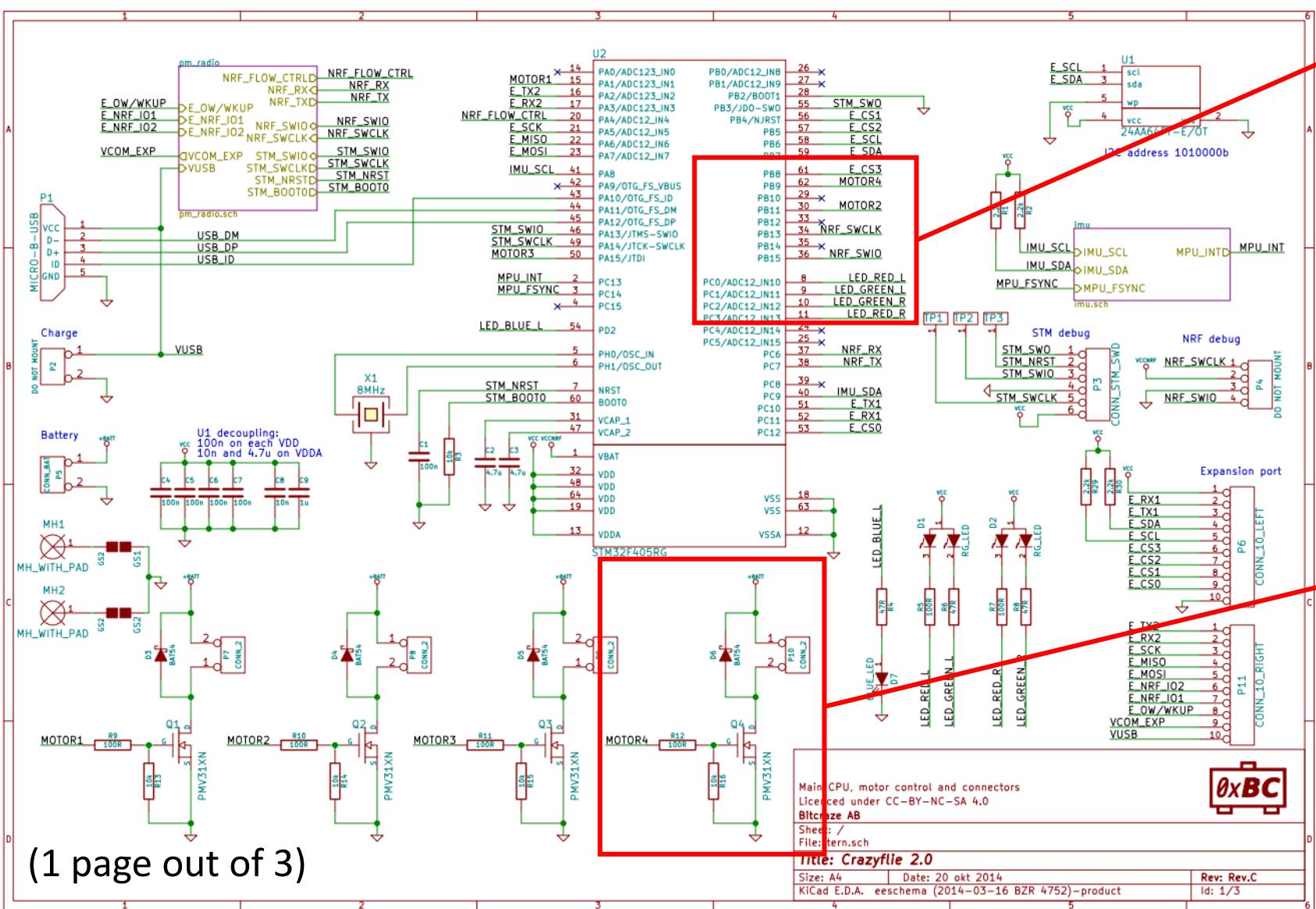
Crazyflie 2.0 system architecture

Low-Level Schematic Diagram View



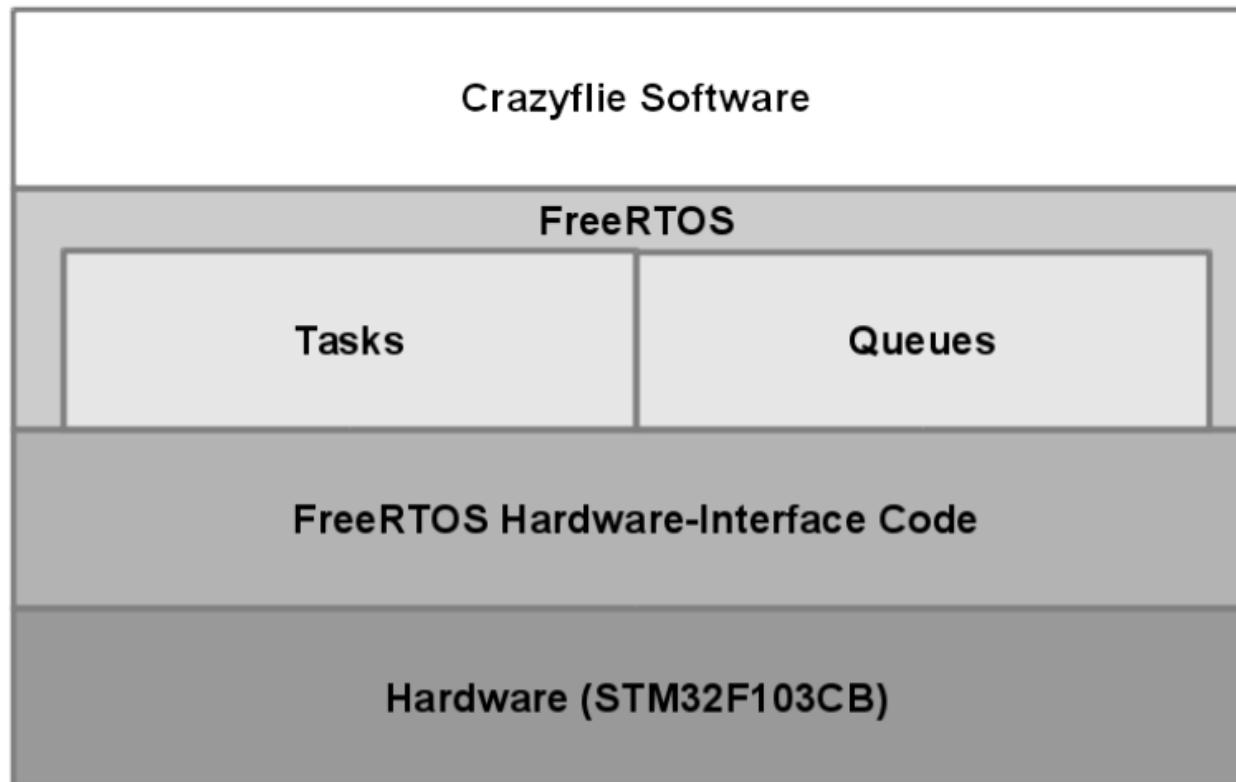
(1 page out of 3)

Low-Level Schematic Diagram View



High-Level Software View

The software is built on top of a *real-time operating system* “FreeRTOS”.



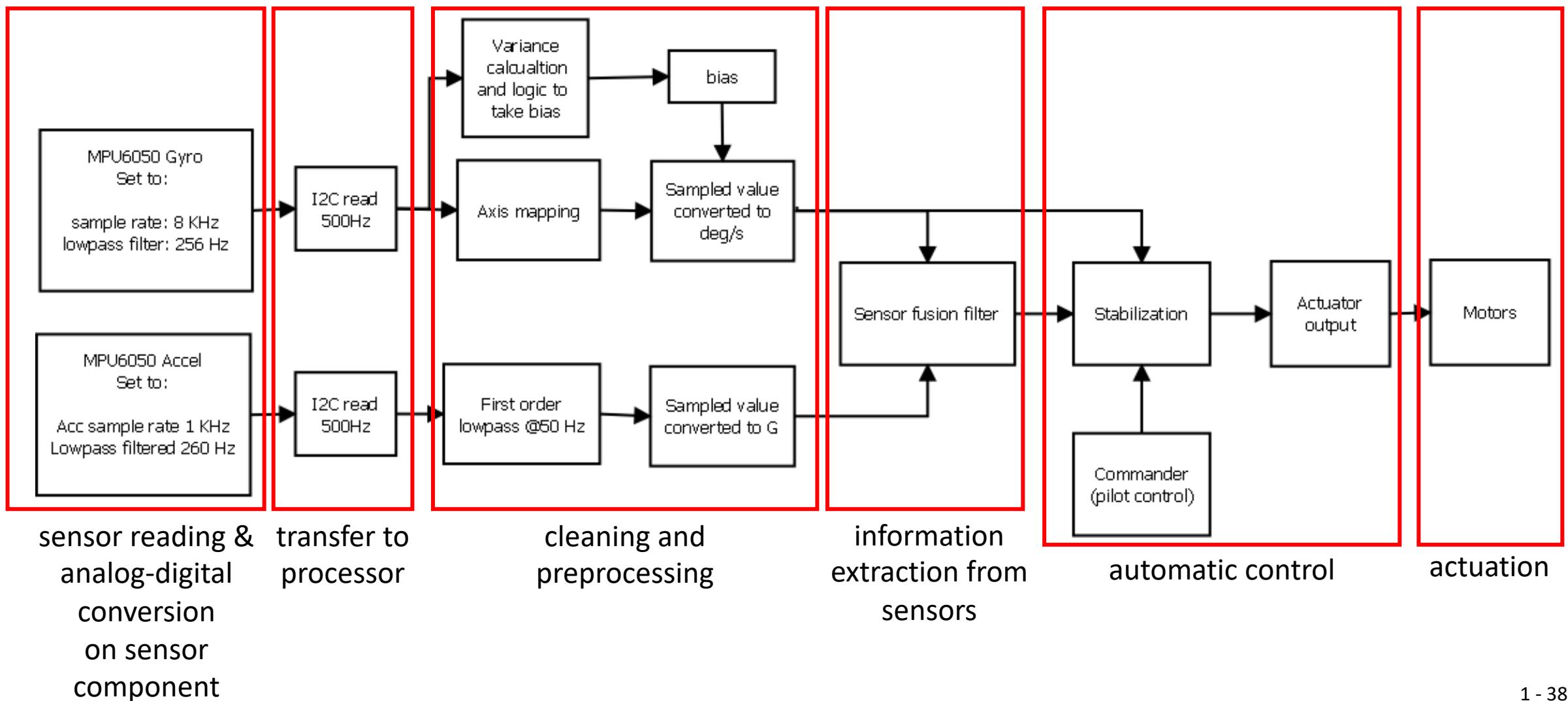
High-Level Software View

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:



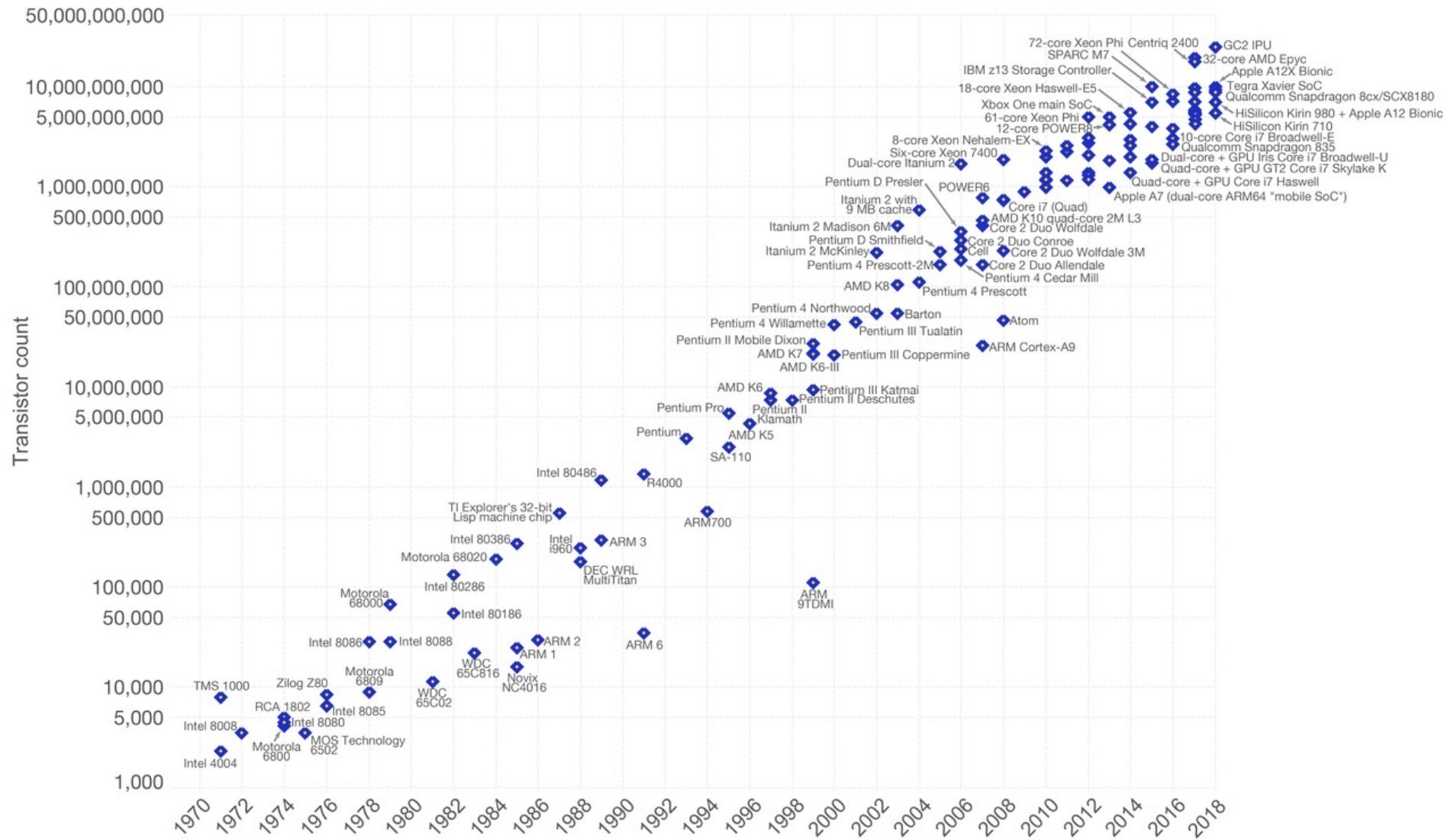
Components and Requirements by Example

- Processing Elements -



What can you do to increase performance?

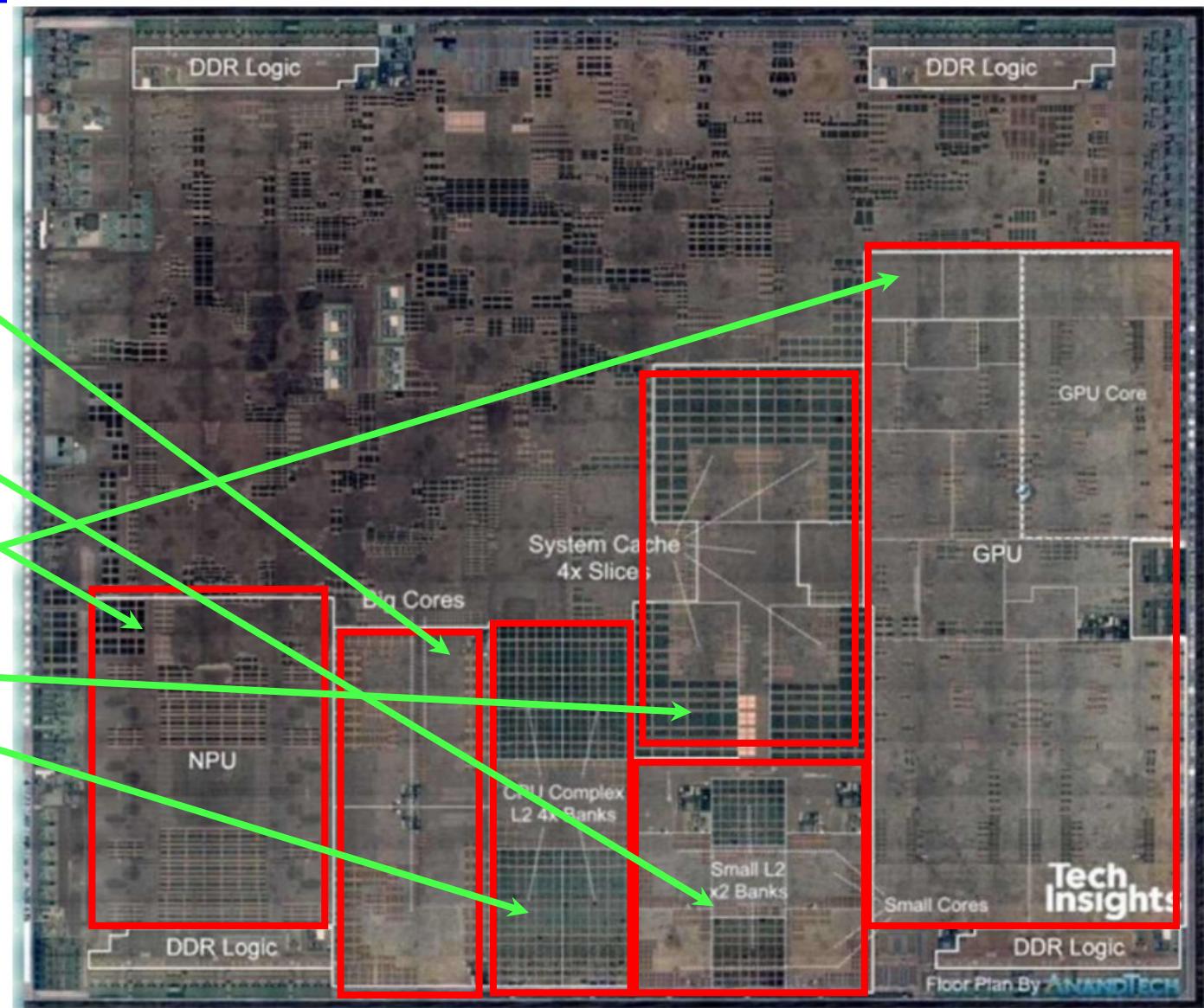
Faster and More Complex Processors



Faster and More Complex Processors

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?

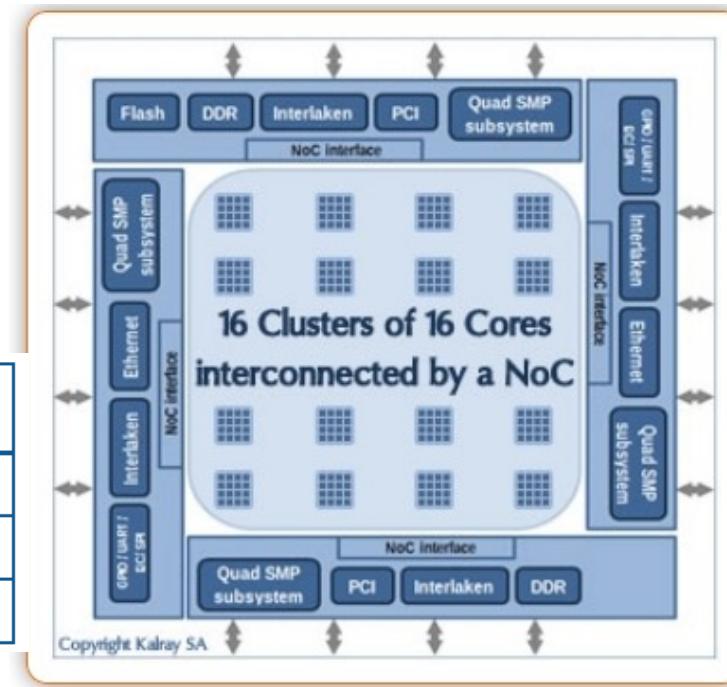
Embedded Multicore Example

Trends:

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



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

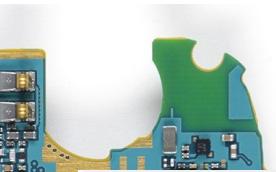
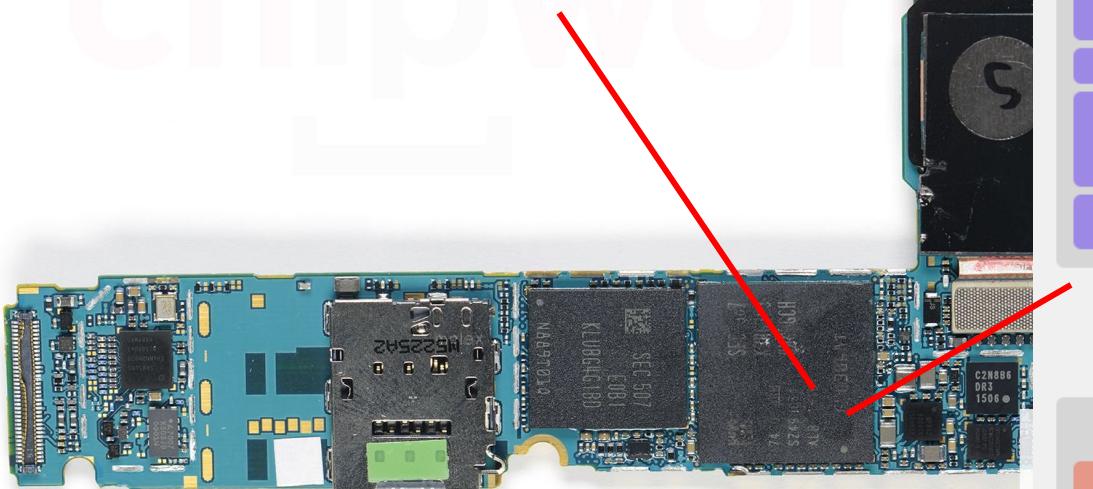


Why does higher parallelism help in reducing power?

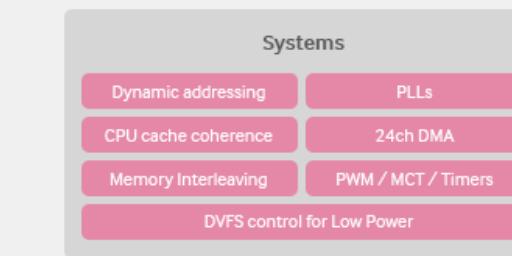
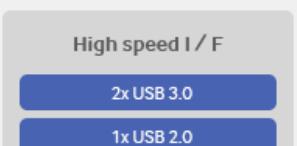
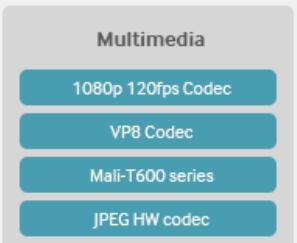
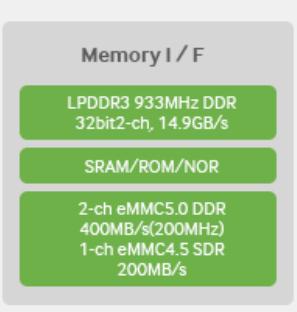
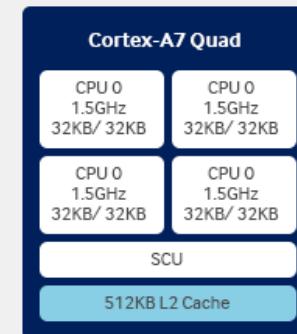
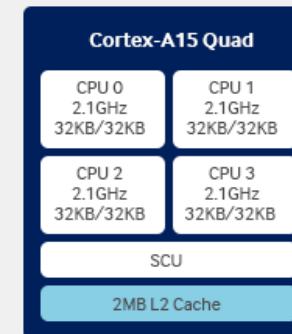
System-on-Chip

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?

ARM big.LITTLE Architecture

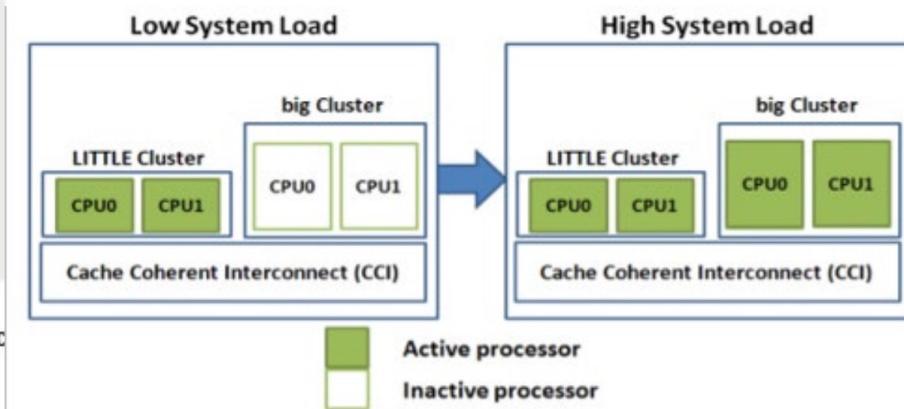
Core Complex 1	Core Complex 2	Connectivity
2–4 x ARM® Cortex®-A35	Cortex-M4F	4 x UART
32 KB I-cache	16 KB I-cache	8 x I²C
32 KB D-cache	16 KB D-cache	4 x SPI
512 KB L2 cache with ECC	256 KB SRAM	1 or 2 x 1 Gbit Ethernet AVB
Multimedia	Memory	1 x 10/100 Ethernet
GPU 1 x 4-Shader, OpenGL ES 3.0 or 3.1, Vulkan®	DDR3L @ 933 MHz (ECC option)/ LPDDR4 @ 1200 MHz (no ECC)	3.3 V/1.8 V GPIO
VPU	2 x SDIO3.0/eMMC5.1	PCIe 3.0 with L1 Substate–1-lane
Video: h.265 dec 4K, h.264 enc/dec 1080p	RAW NAND–BCH62	1 x USB3 OTG w/PHY
Audio	2 x Quad/1 x Octal SPI	1 or 2 x USB2 OTG w/PHY
DSP Core	Security	3 x CAN/CAN FD
Tensilica® HiFi 4	HAB, SRTC, SJTAG, TrustZone®	MOST 25/50
32 KB I	AES256, RSA4096, SHA-256	4 x 4 Keypad
48 KB D	3DES, ARC4, MD-5	4 x PWM
512 KB SRAM (448 KB OCRAM, 64 KB of TCM)	Flashless SHE, ECC	2 x ASRC, SPDIF
Display and Camera I/O	Tamper, Inline Enc Engine	4 x SAI, ESAI, MQS
Display Processor with SafeAssure®		
2 x MIPI-DSI/LVDS Combo PHY*	System Control	
1 x Parallel Display	Power Control, Clocks, Reset	
1 x Parallel CSI	BootROMs	
1 x MIPI CSI	PMIC interface (dedicated I²C)	
	Domain Resource Partitioning	

Available on certain product families

Note: Accessing muxable controller's full capabilities is dependent upon board component choice



Toradex Colibri Compute-on-Module

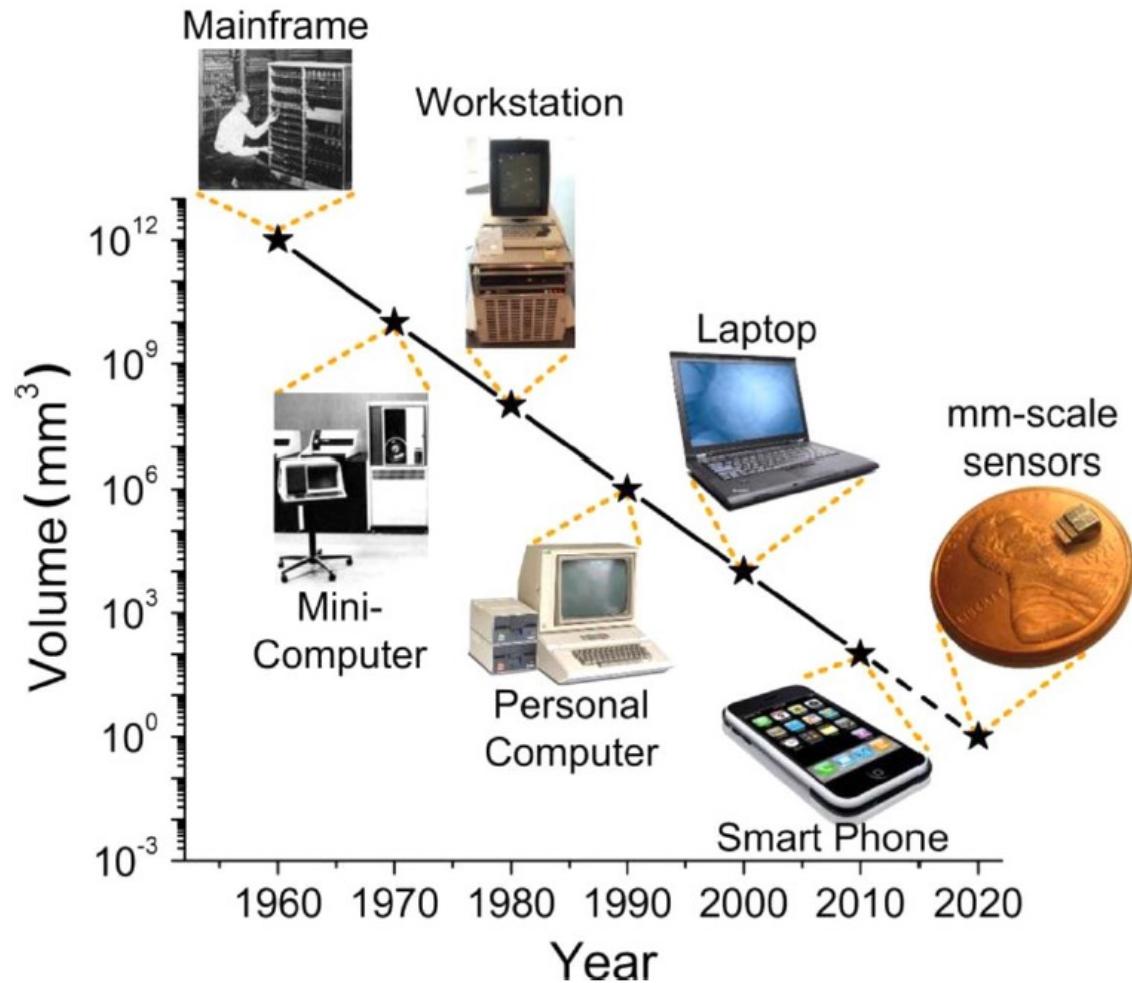


Components and Requirements by Example

- Systems -



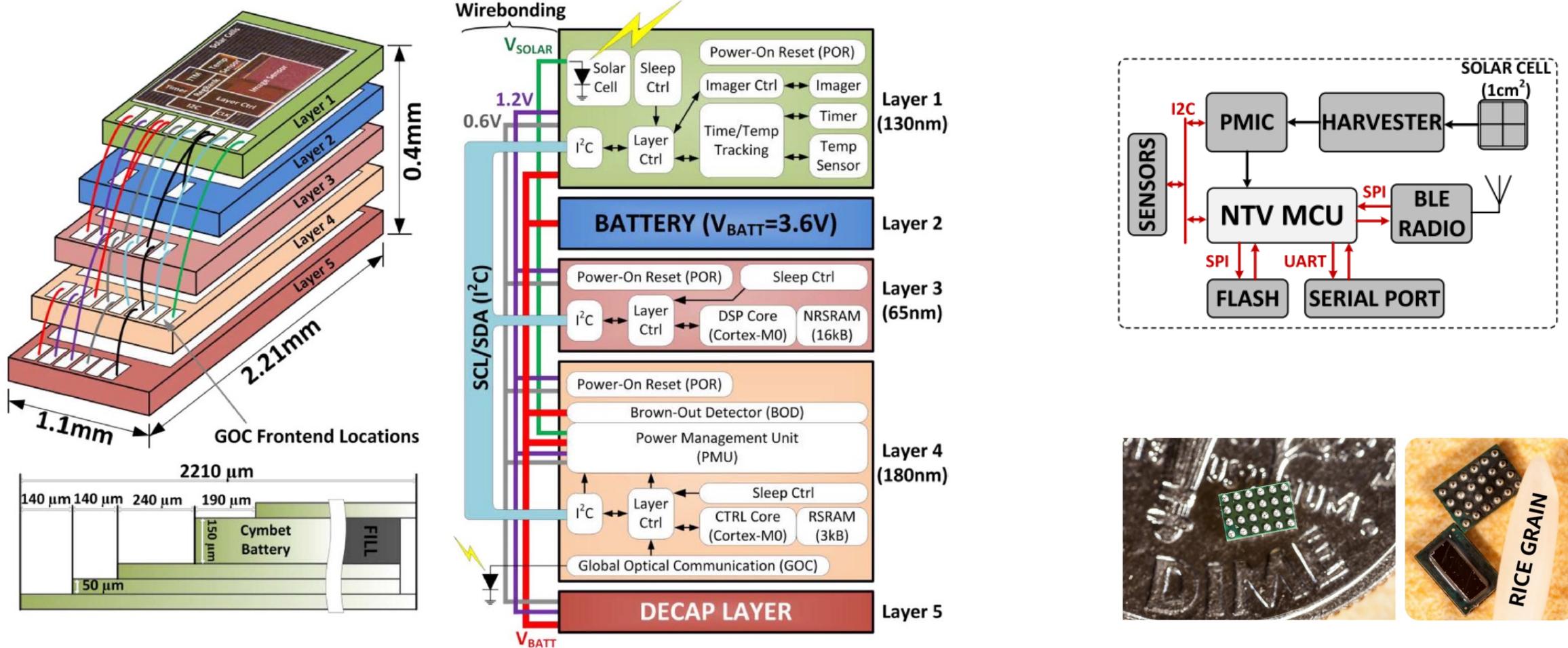
Zero Power Systems and Sensors



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* and *battery-free systems* to achieve long-term autonomous operation.

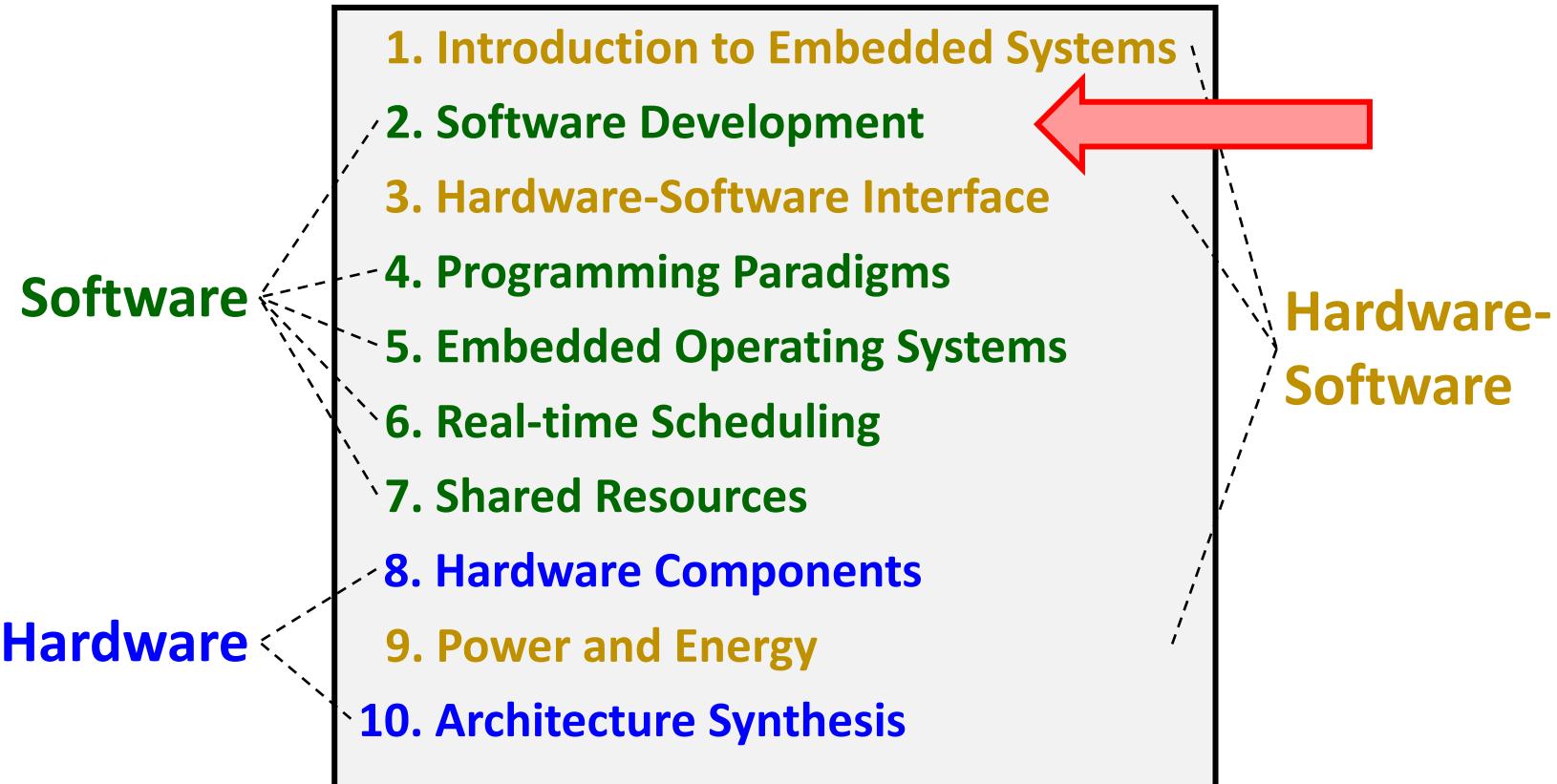
Introduction to Embedded Systems

2. Software Development

Prof. Dr. Marco Zimmerling

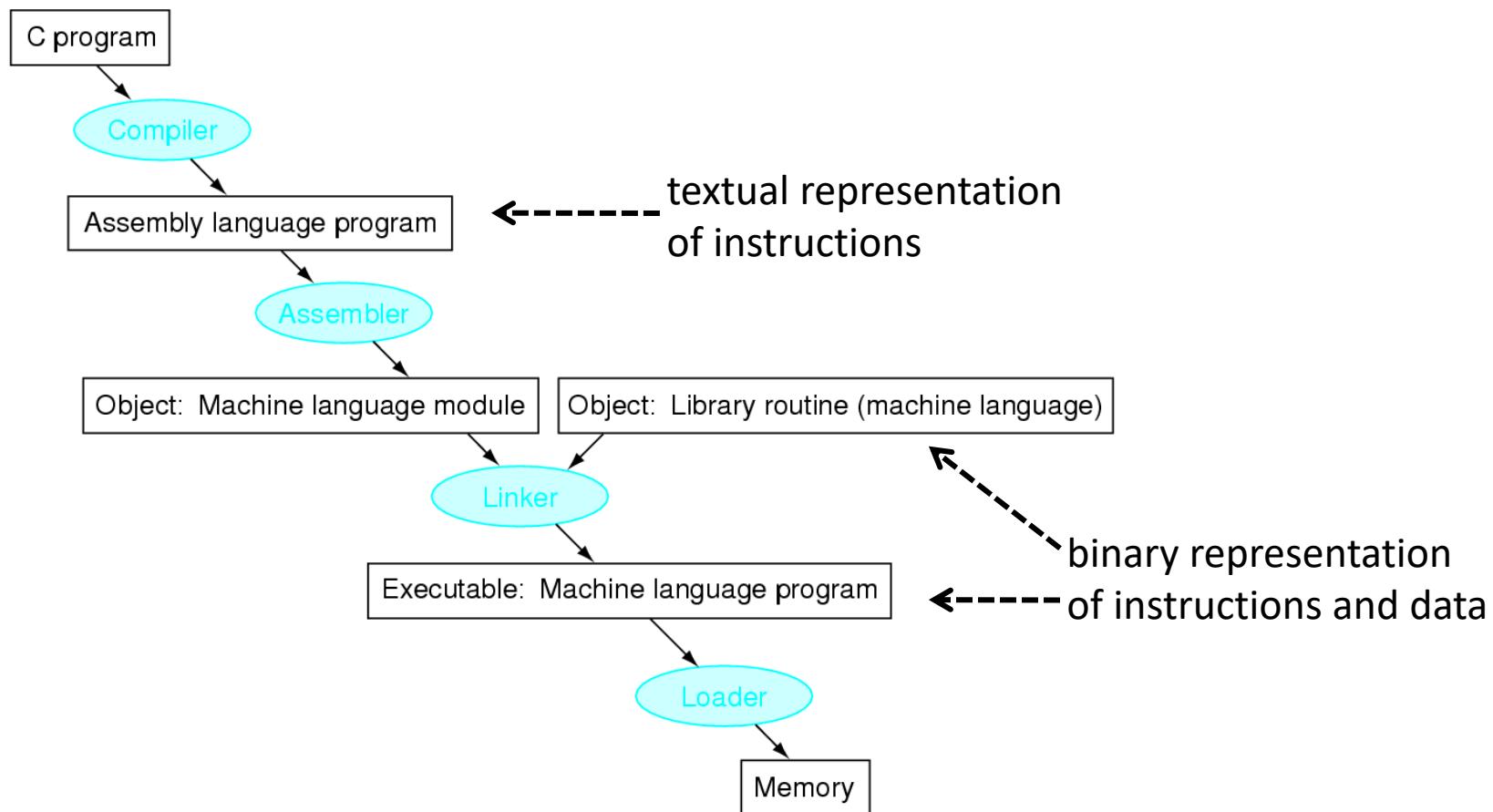


Where we are ...



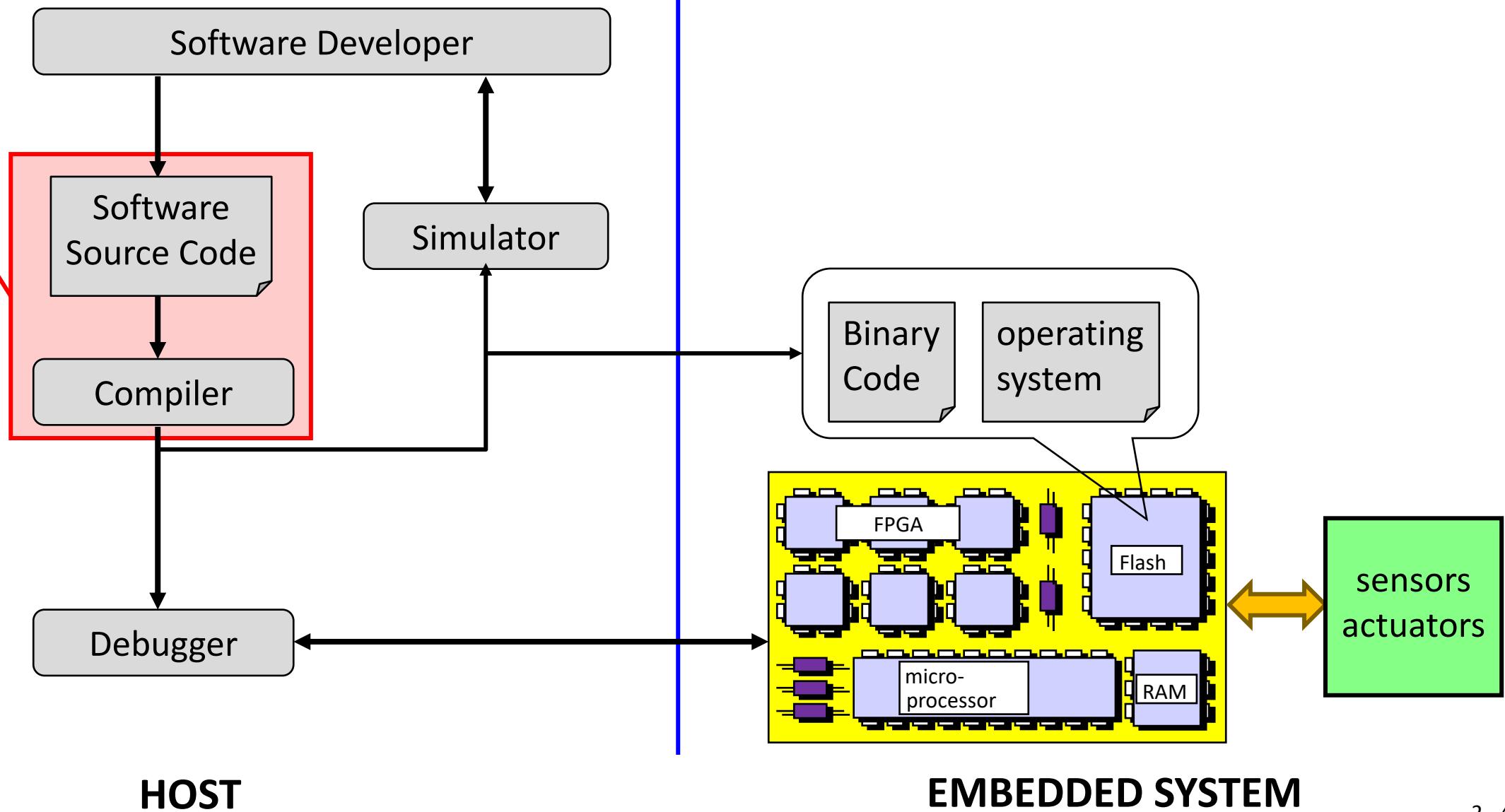
Compilation Process

Compilation of a C program to a machine language program:

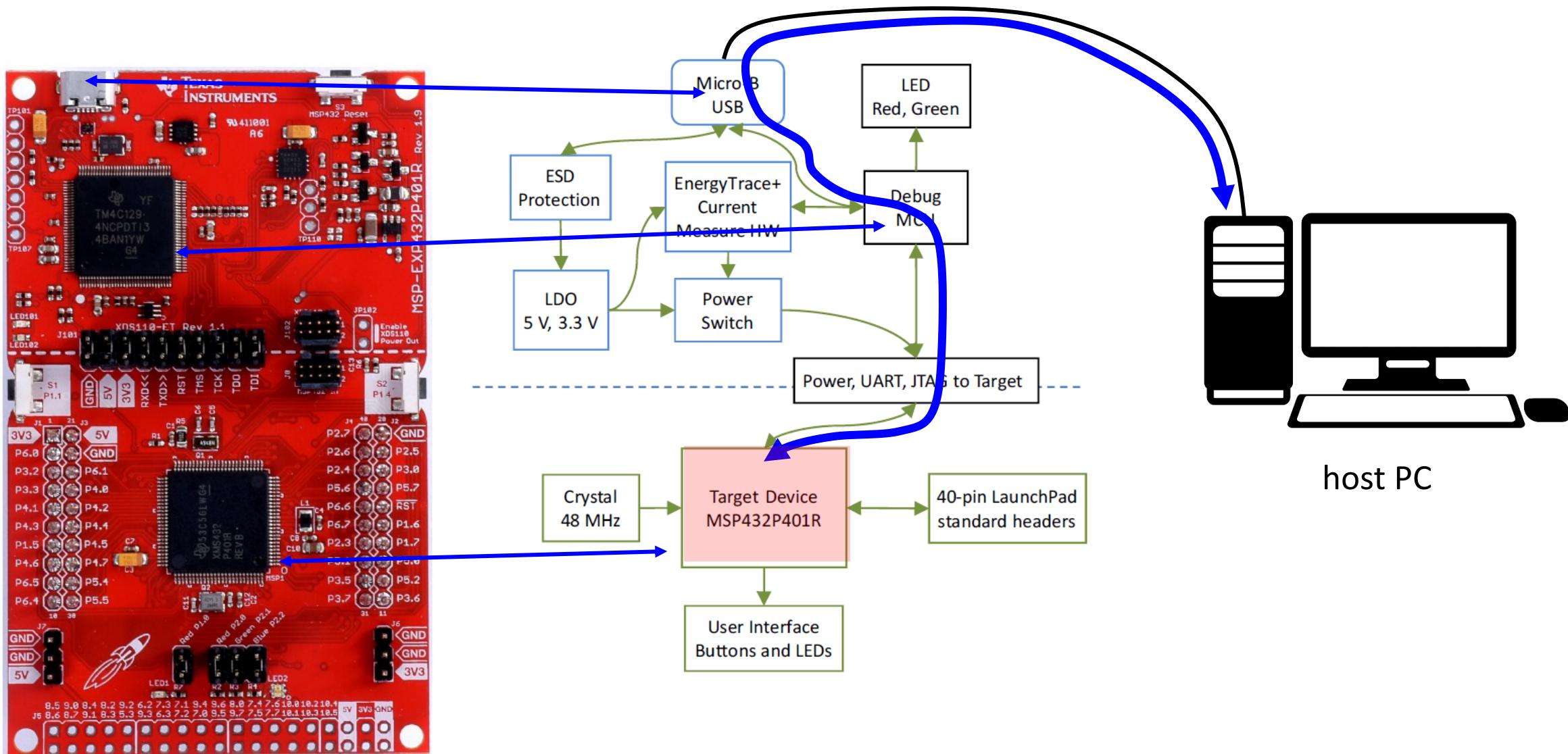


Embedded Software Development

previous slide



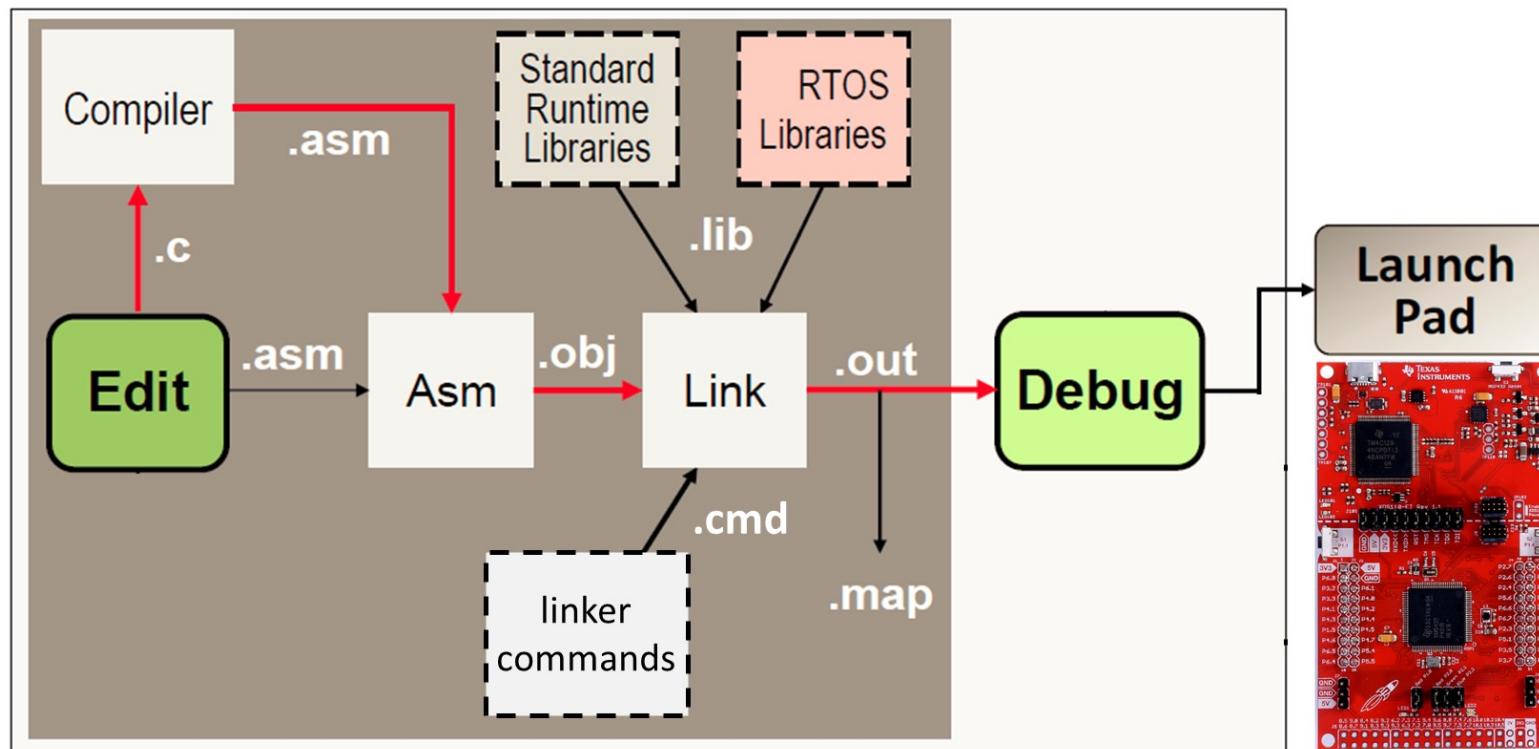
Software Development with the TI LaunchPad



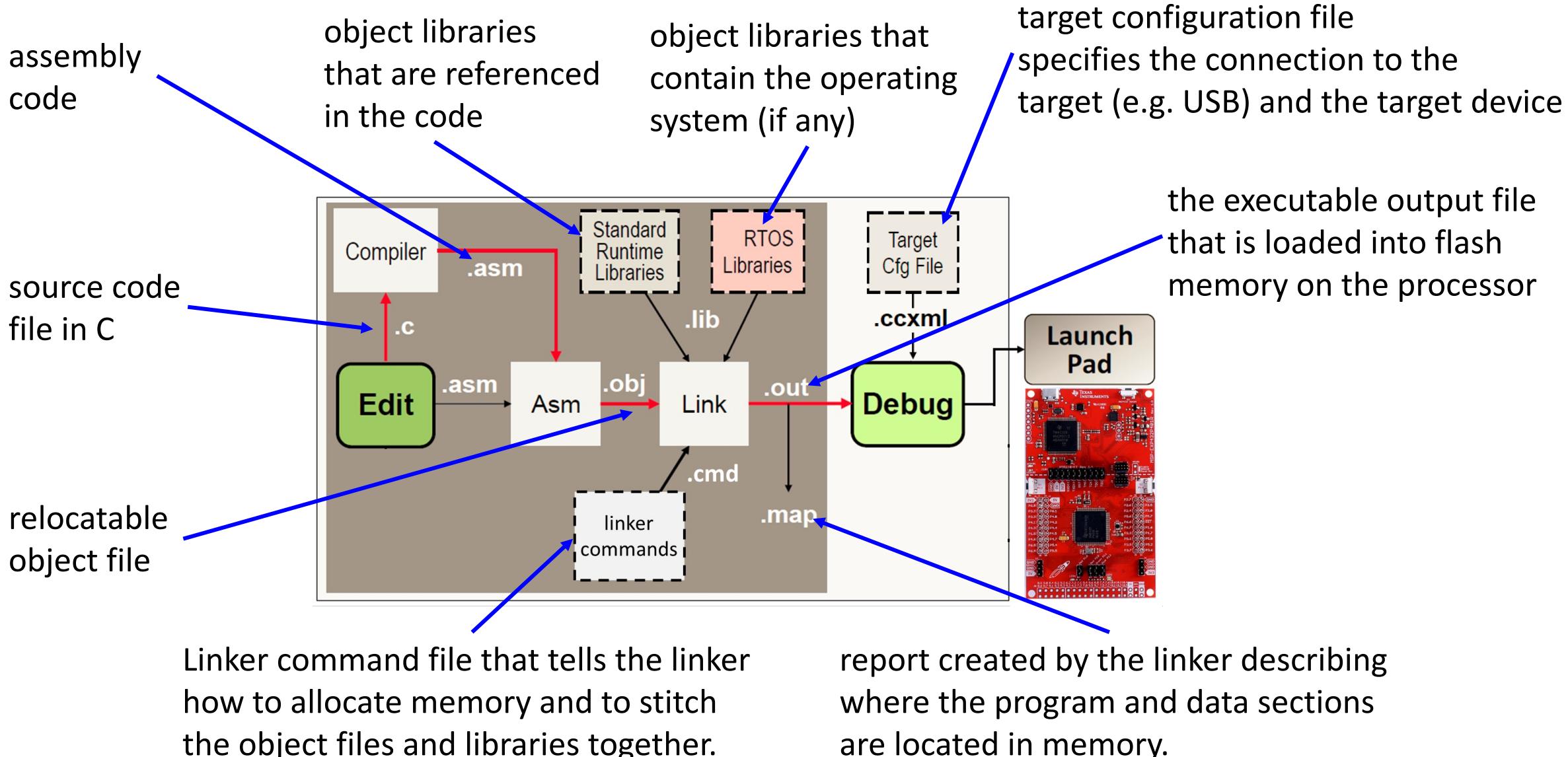
Software Development

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



Software Development

assembly code

source code file in C

relocatable object file

object libraries that are referred in the code

Compiler .asm

Edit .asm

Linker command file that specifies how to allocate memory for the object files and libraries

```
...
/*
 * 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);

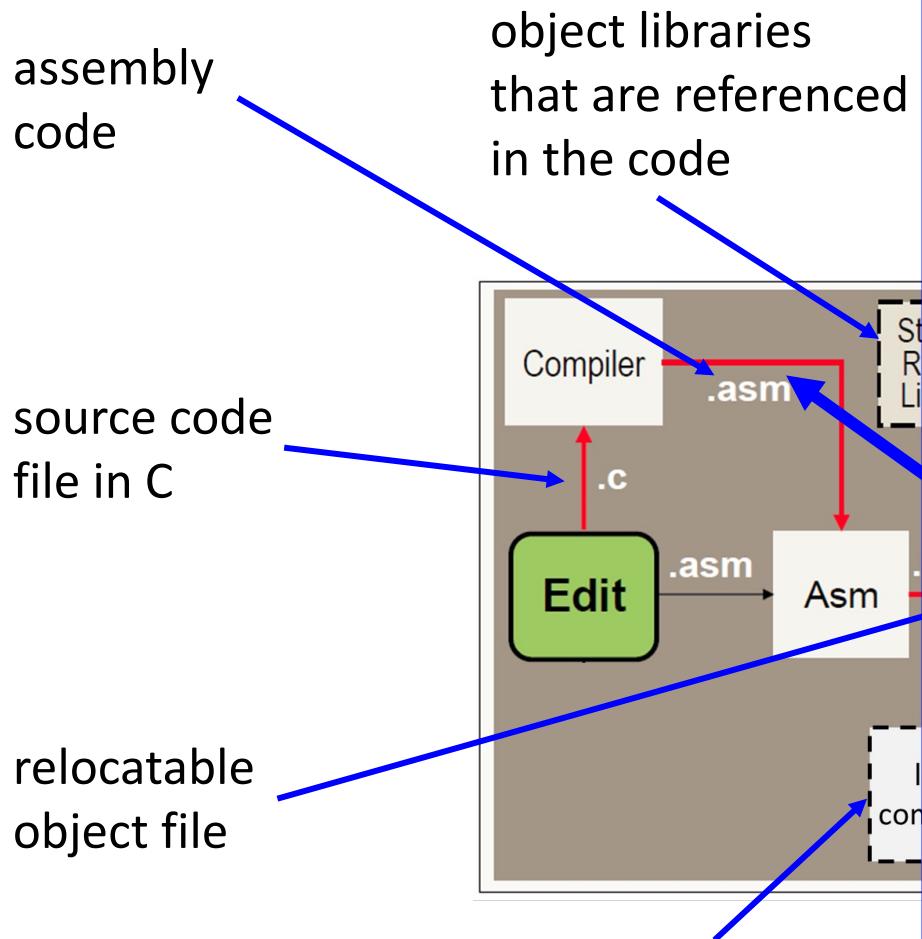
    /* Set the core voltage level to VCORE1 */
    MAP_PCM_setCoreVoltageLevel(PCM_VCORE1);

    /* 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
the output file into flash
the processor

Software Development



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

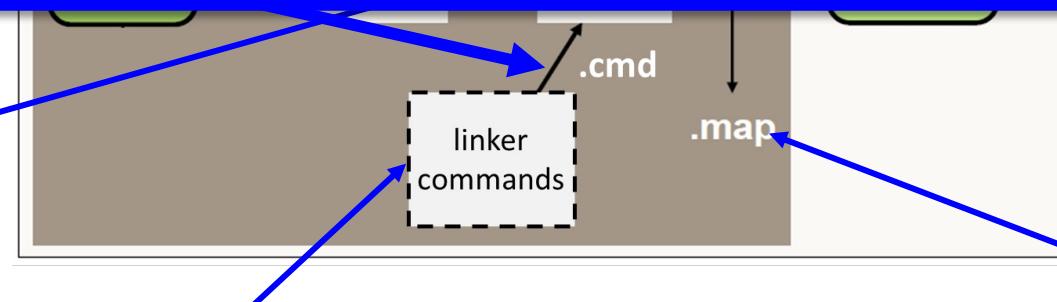
```
...
; **** FUNCTION NAME: SysTick_Handler ****
;
; * Regs Modified      : A1,A2,A3,A4,V9,SP,LR,SR,D0,D0_hi,D1,D1_hi,D2,D2_hi,
; *                      D3,D3_hi,D4,D4_hi,D5,D5_hi,D6,D6_hi,D7,D7_hi,
; *                      FPEXC,FPSCR
;
; * Regs Used          : A1,A2,A3,A4,V9,SP,LR,SR,D0,D0_hi,D1,D1_hi,D2,D2_hi,
; *                      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
; **** FUNCTION NAME: SysTick_Handler ****
;
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
#define __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
}
```

...

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.



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

...

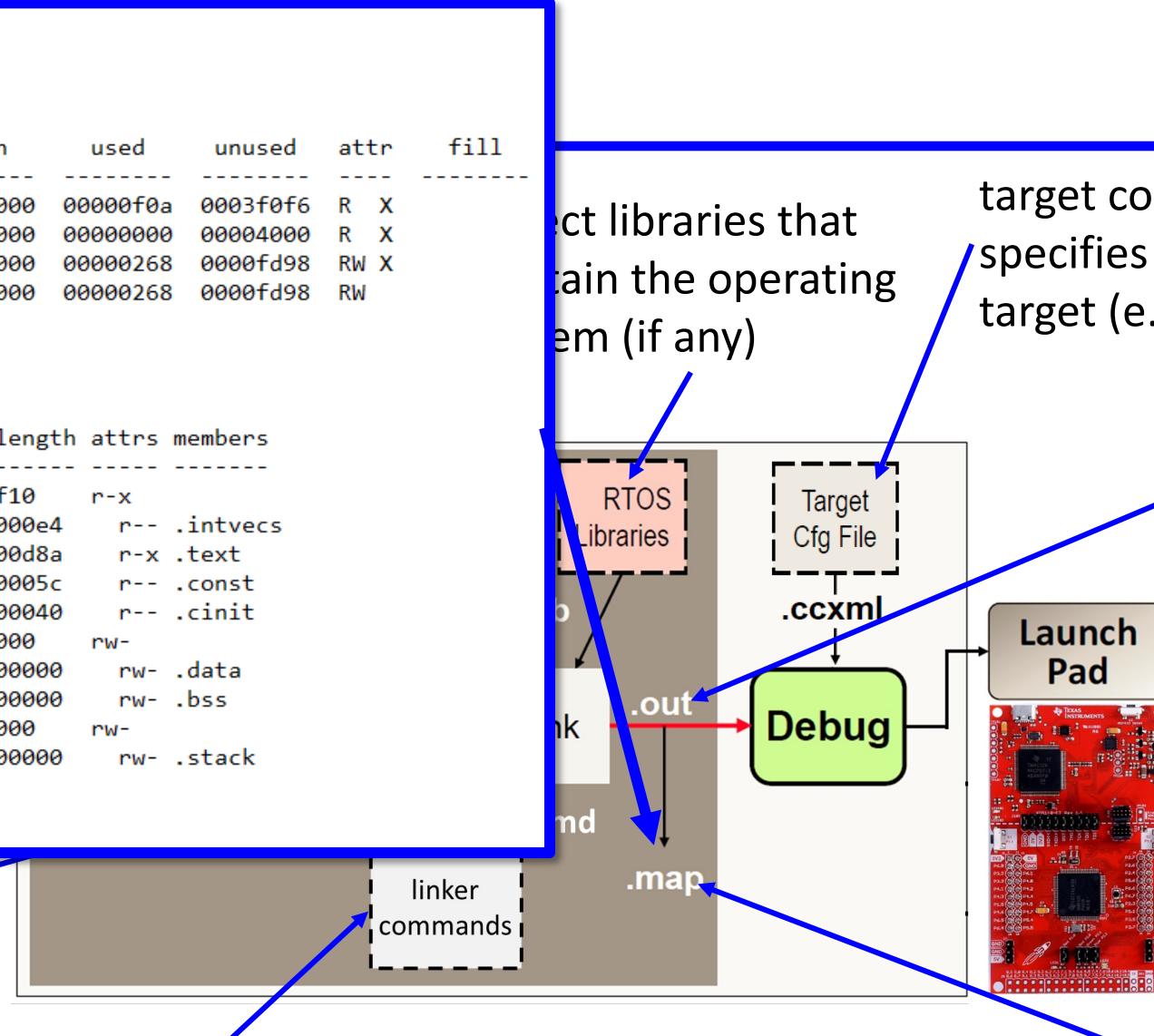
MEMORY CONFIGURATION

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
...						

relocatable object file



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

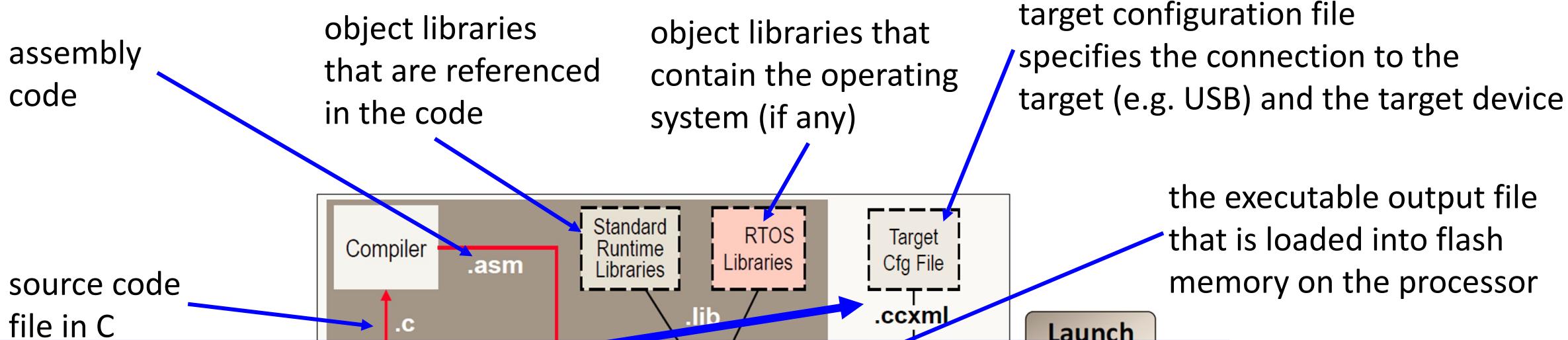
report created by the linker describing where the program and data sections are located in memory.

select libraries that contain the operating system (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

Software Development

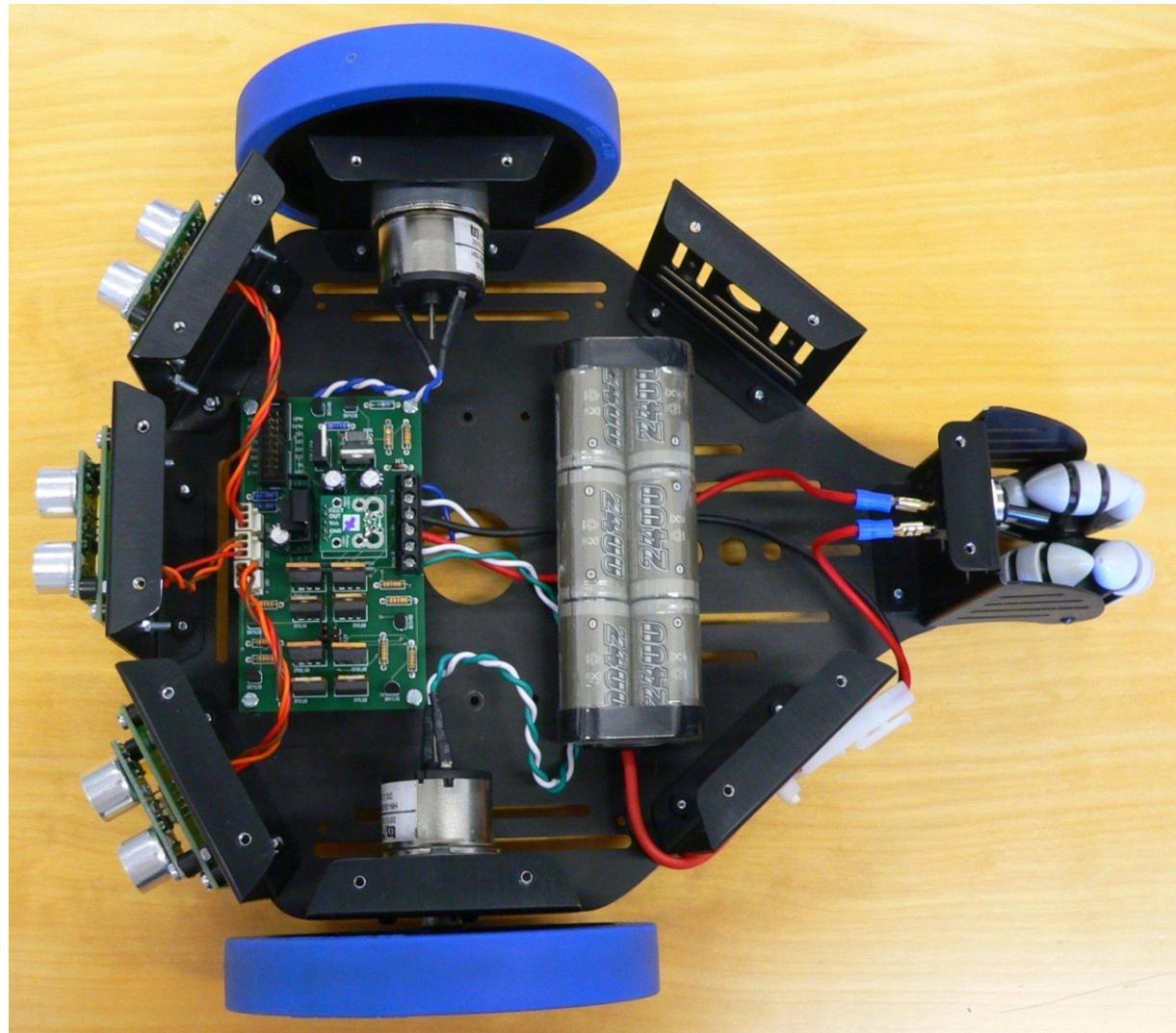


```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configurations XML_version="1.2" id="configurations_0">
  <configuration XML_version="1.2" id="configuration_0">
    <instance XML_version="1.2" desc="Texas Instruments XDS110 USB Debug Probe" href="connections/ ...
    <connection XML_version="1.2" 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 ...

- Mikrocontroller-Praktikum
- Hardware-Praktikum (summer)
 - Microcontroller programming using Arduino IDE
 - FPGA programming
 - *From the schematics to a real embedded application*



Introduction to Embedded Systems

3. Hardware Software Interface

Prof. Dr. Marco Zimmerling



Join the Course on ILIAS!

Access via: <https://nes-lab.org/>

- Courses
- Introduction to Embedded Systems
- ILIAS
- Login: RZ username + password
- Course password: **es-0x8af**



Resources:

- Forum, course schedule, Zoom/BBB links, important announcements
- Slides and recording **after** each lecture
- Exercise sheets **before** each exercise
- Slides, recordings, and exercise solutions **after** each exercise

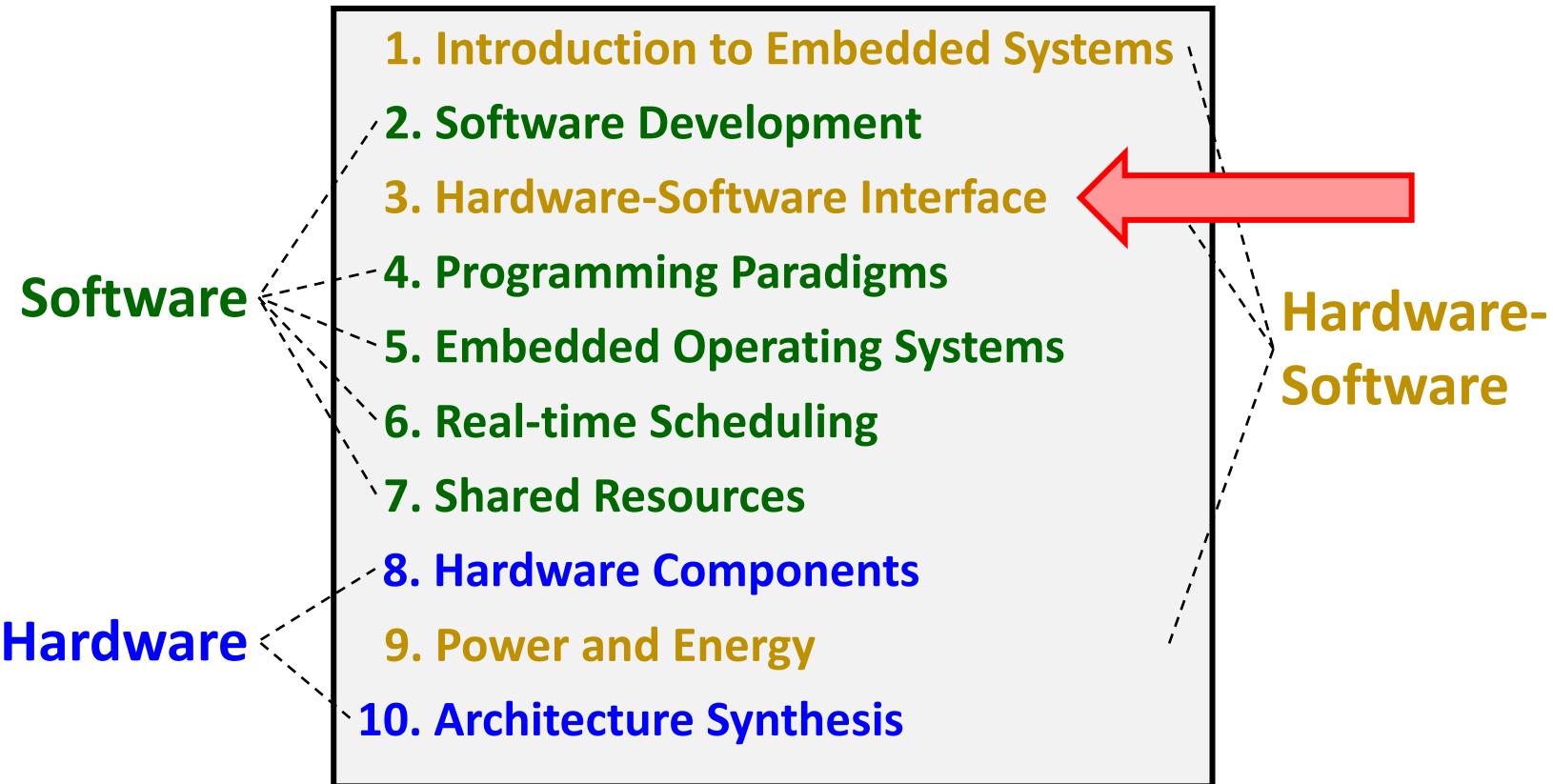
Schedule

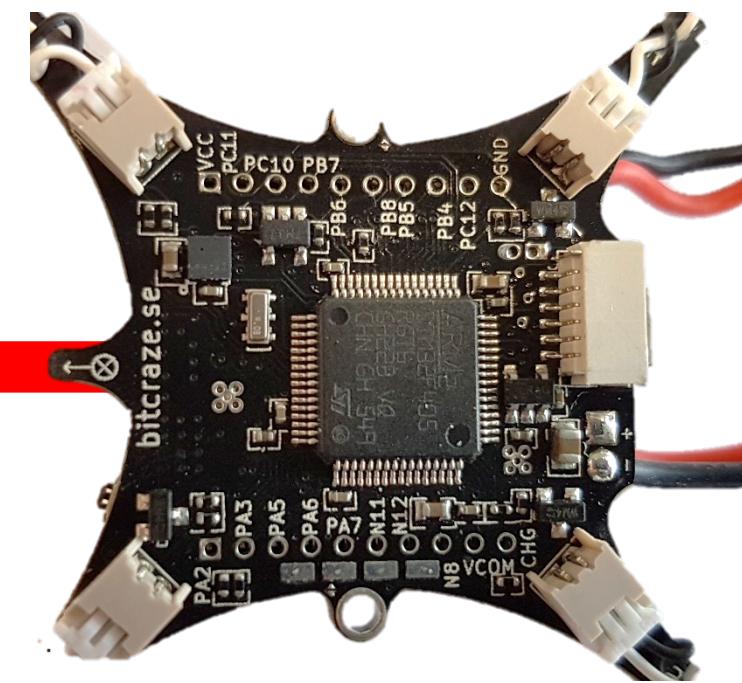
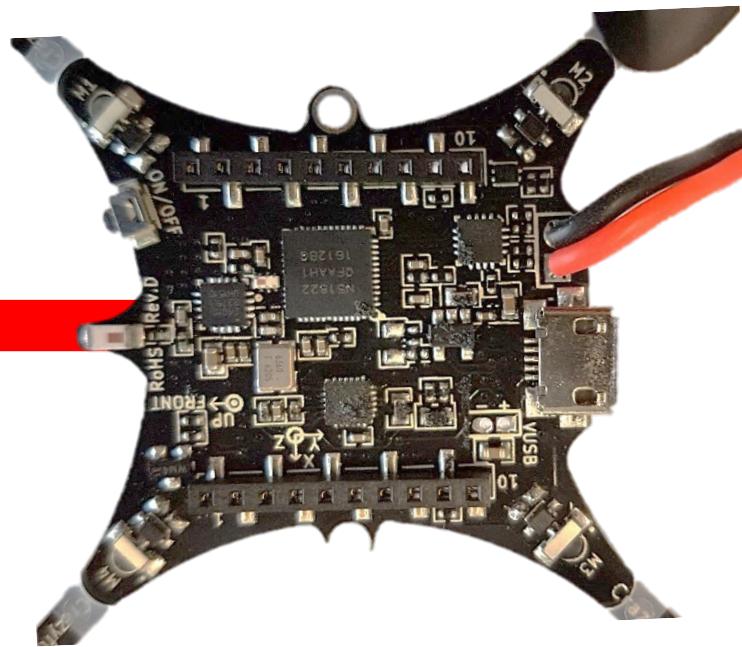
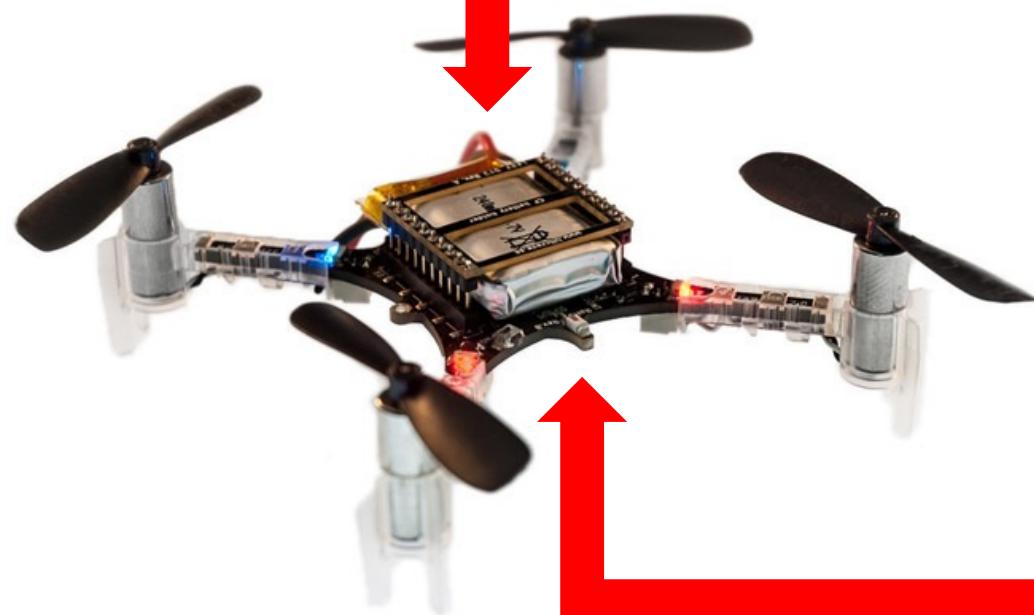
Schedule:

- Today (October 25): **No exercise**
- Next week (November 1): **All Saints' Day**
- In two weeks (November 8): **Lecture and exercise**
- **Check regularly on ILIAS for updates!**

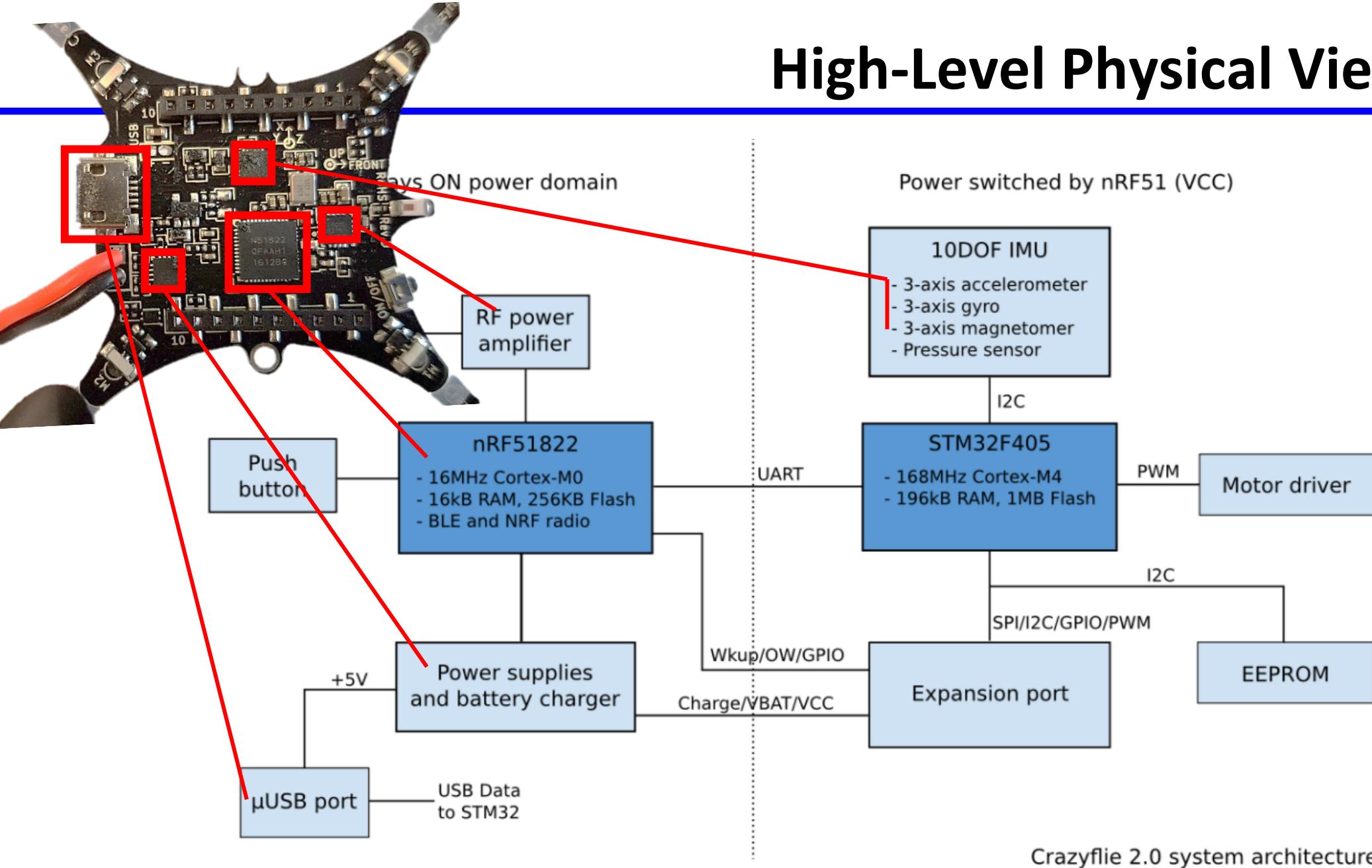
Do you remember?

Where we are ...

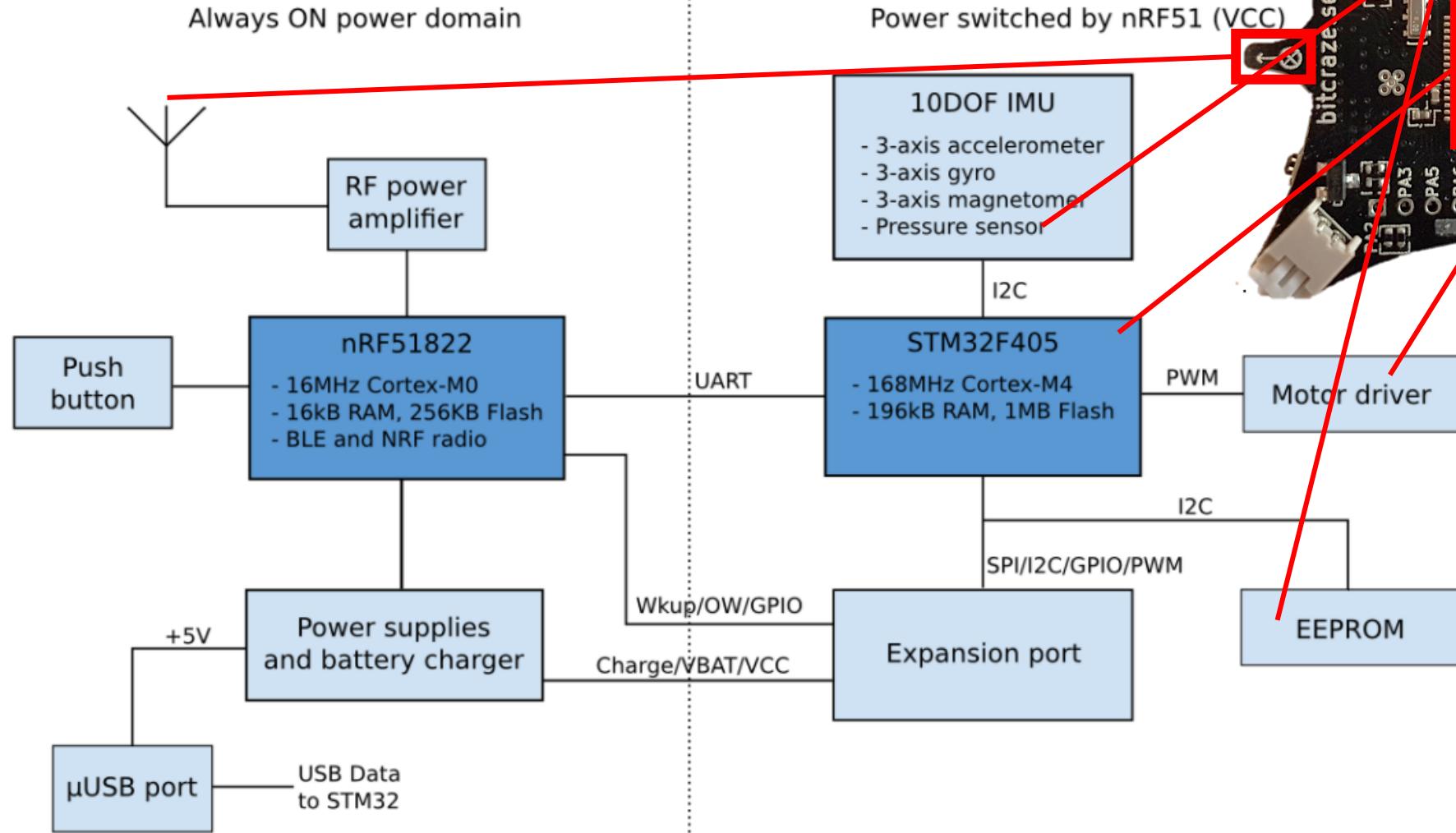




High-Level Physical View



High-Level Physical View



Crazyflie 2.0 system architecture

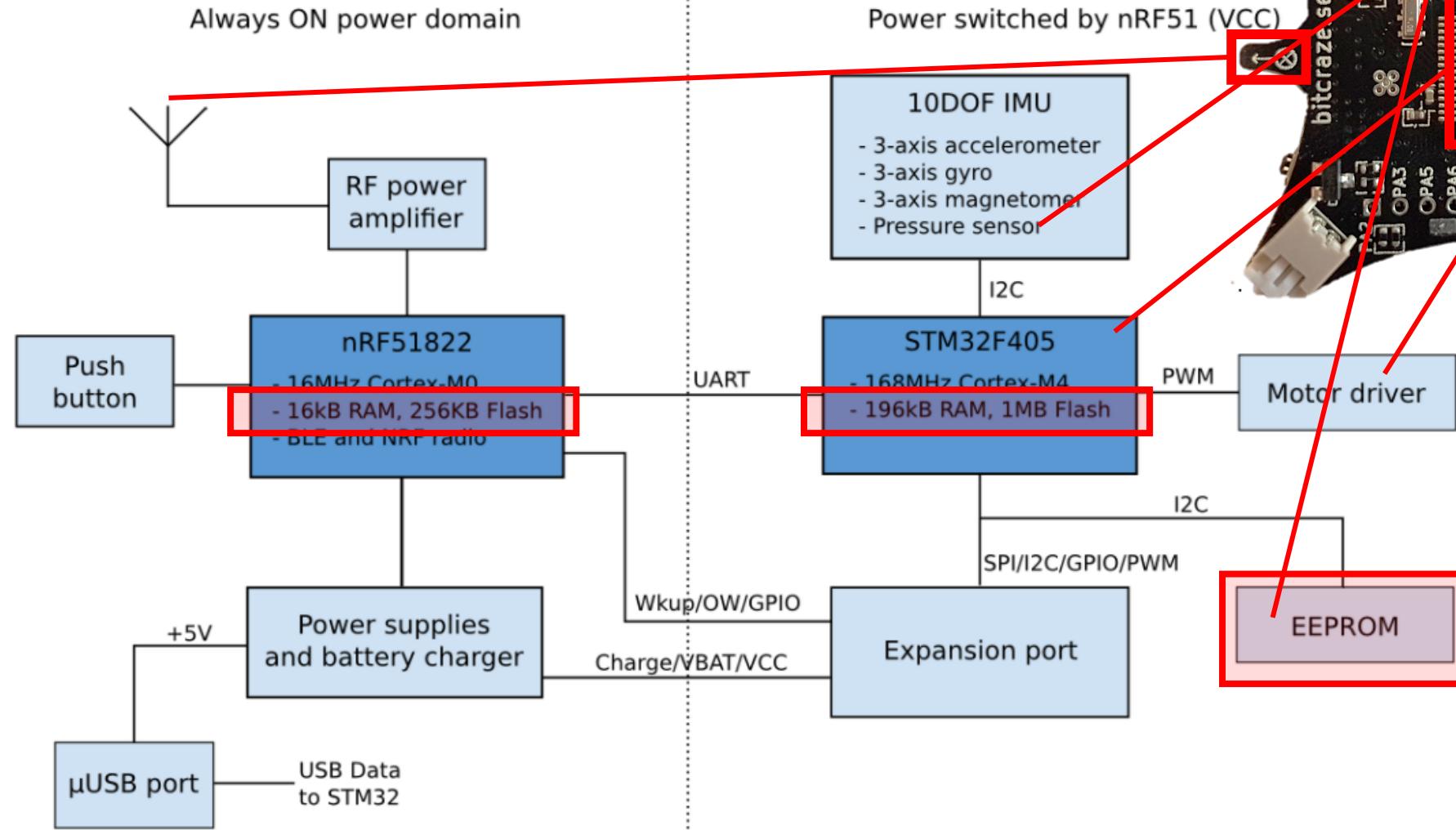
What you will learn ...

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

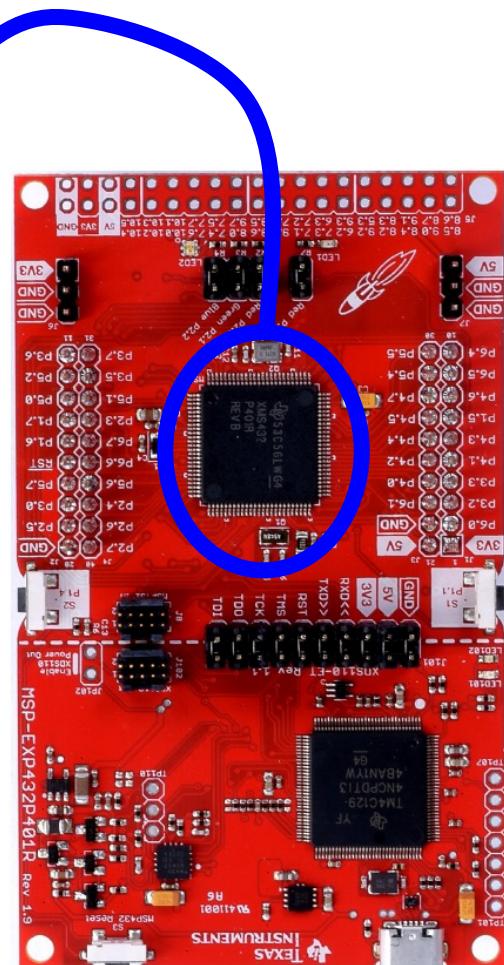
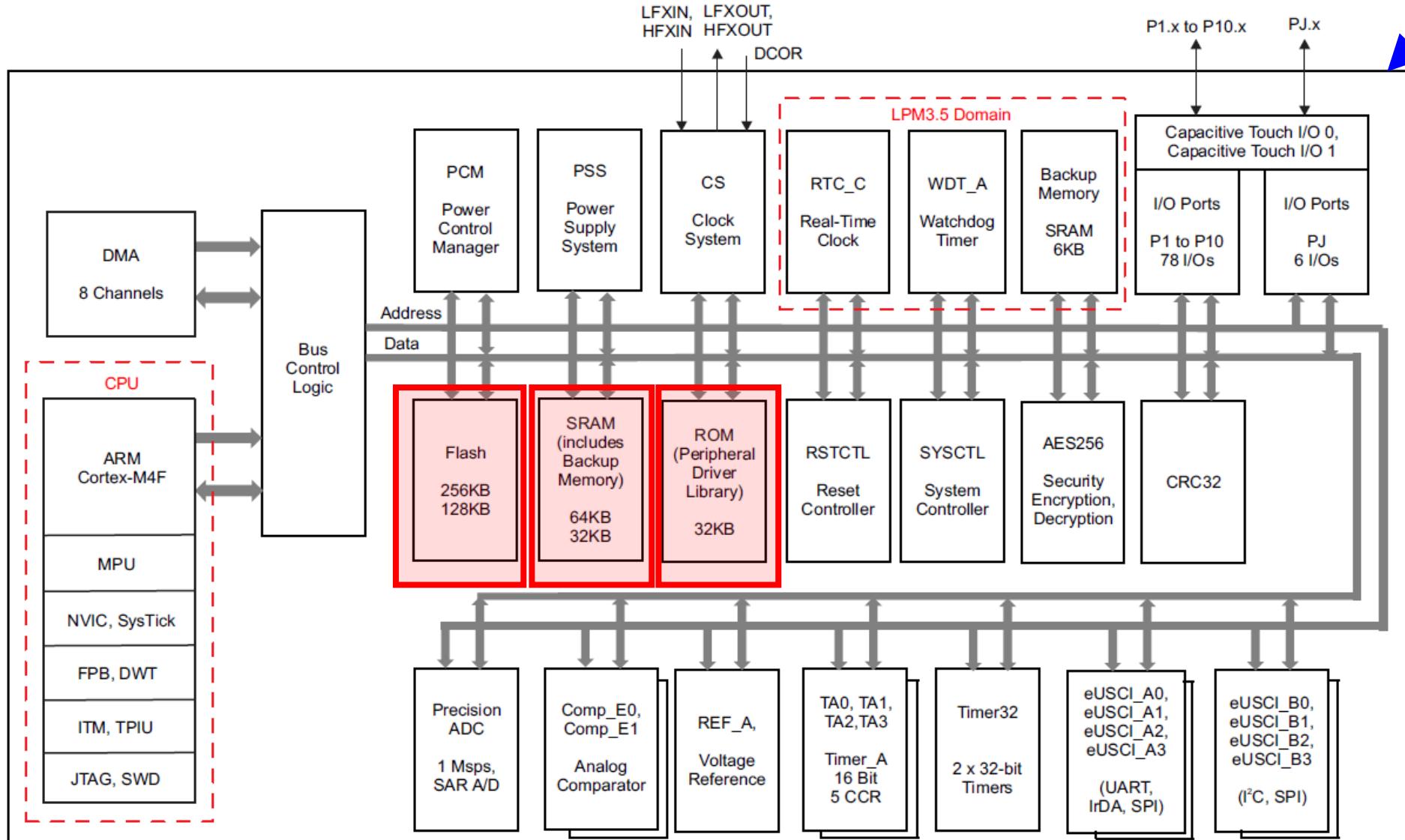
Storage

Remember ... ?



Crazyflie 2.0 system architecture

MSP432P401R

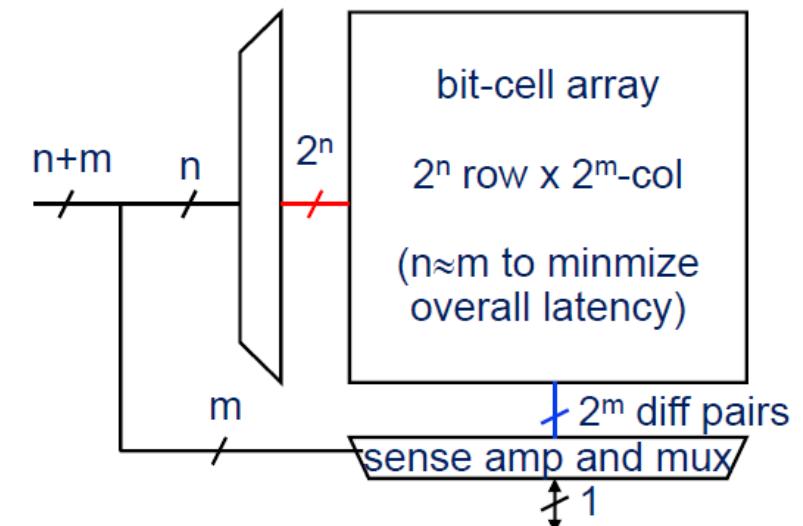
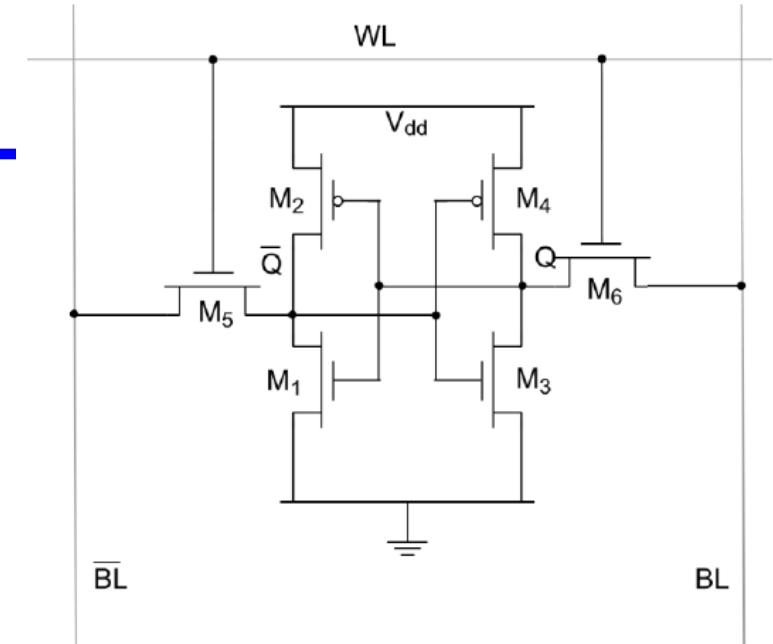


Storage

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 2^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



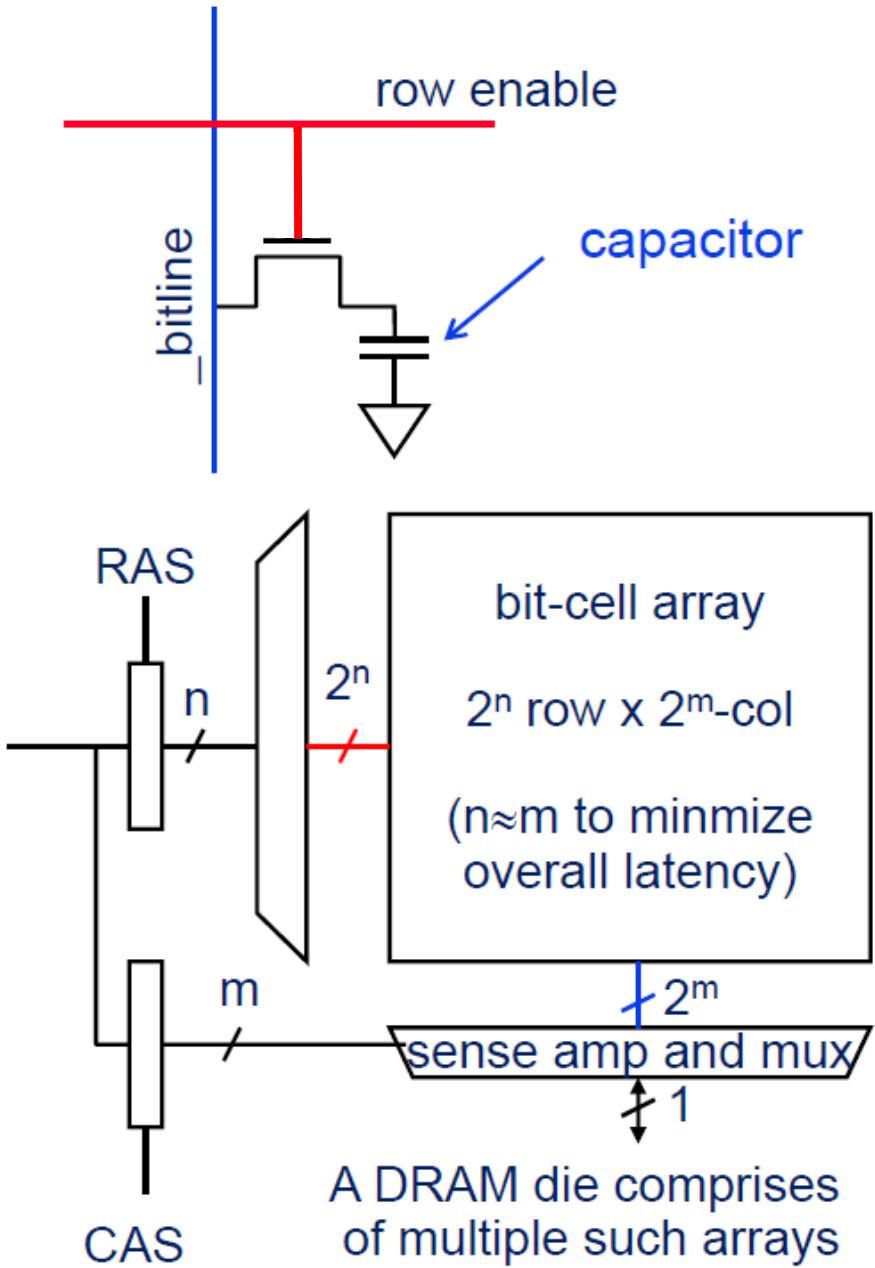
Dynamic Random Access (DRAM)

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

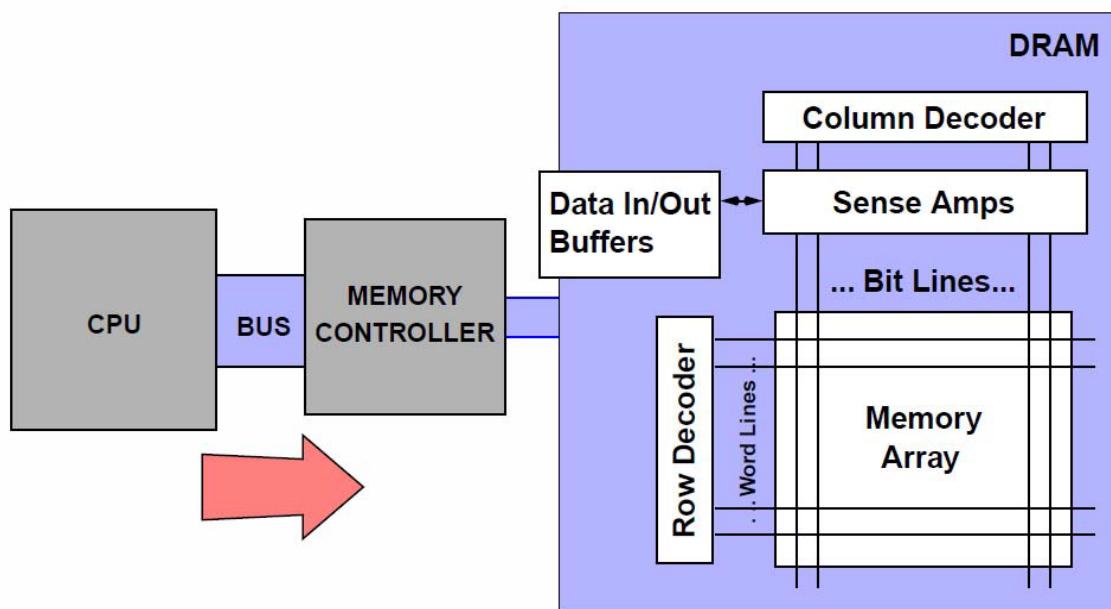
- Performed by the memory controller
- Refresh interval is tens of ms
- DRAM is unavailable during refresh



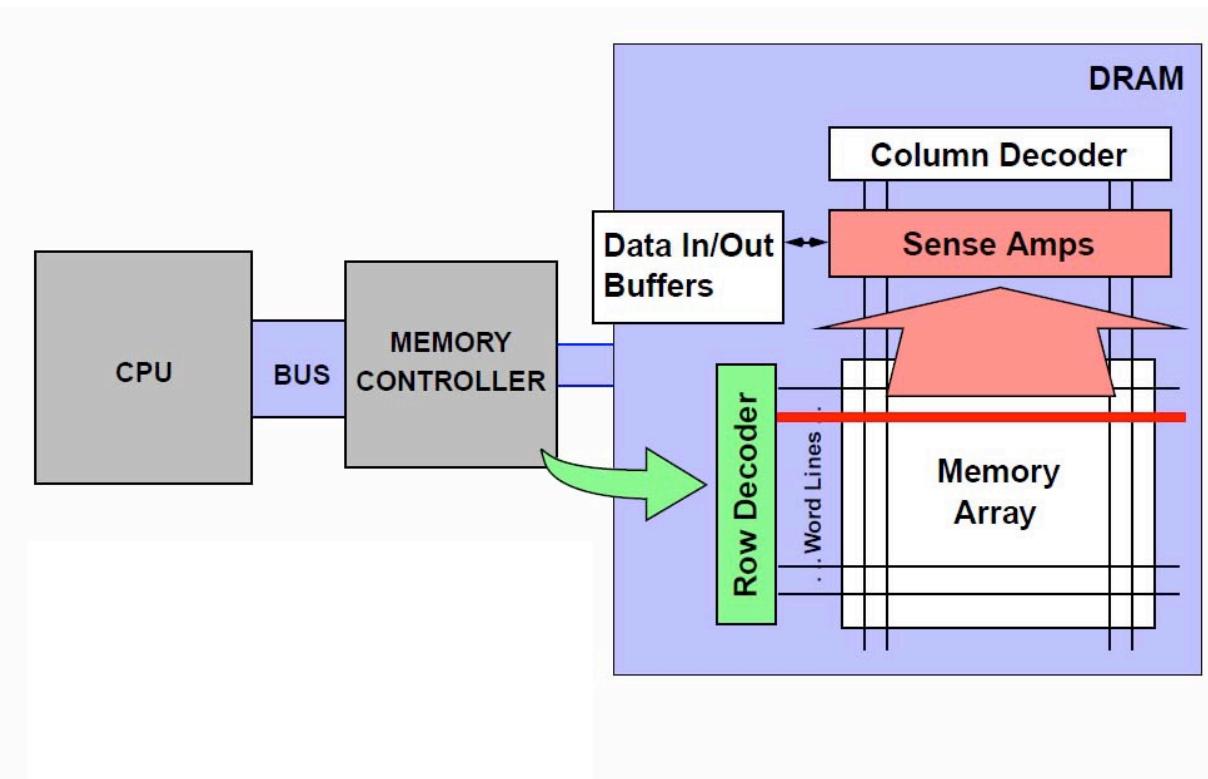
(RAS/CAS = row/column address select)

DRAM – Typical Access Process

1. Bus Transmission

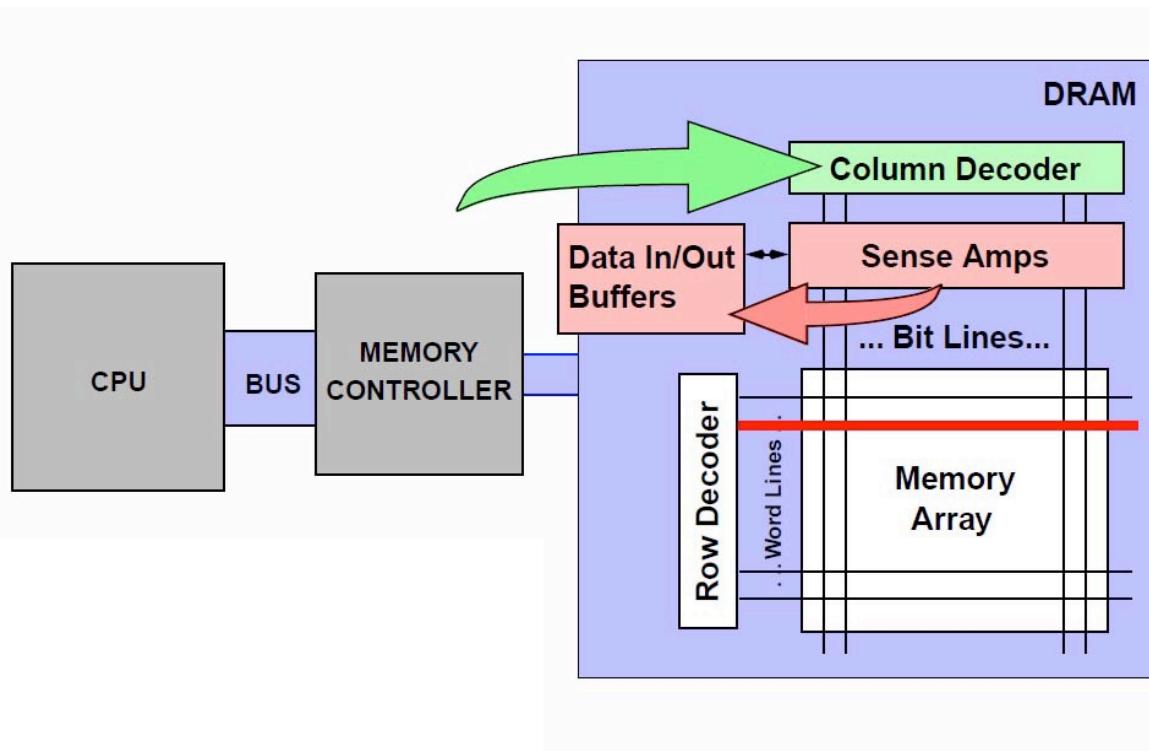


2. Precharge and Row Access

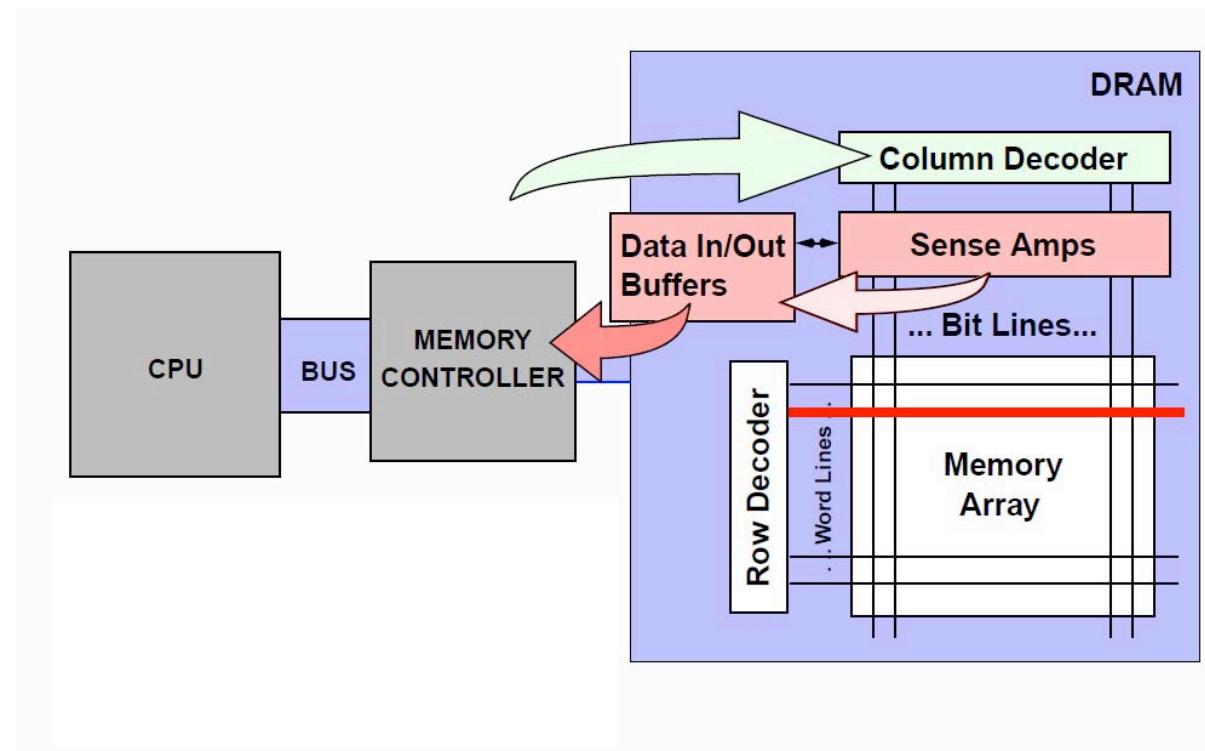


DRAM – Typical Access Process

3. Column Access



4. Data Transfer and Bus Transmission

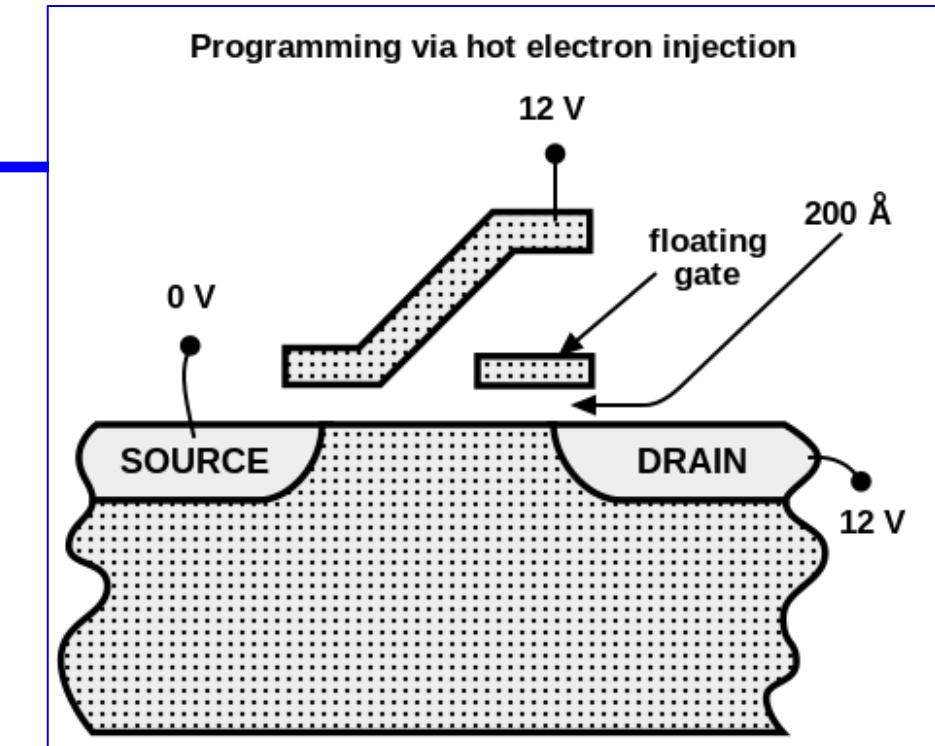


Flash Memory

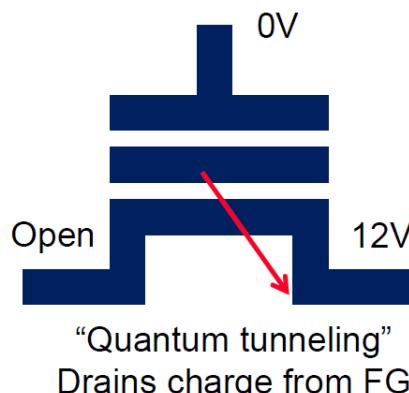
Electrically modifiable, non-volatile storage

Principle of operation:

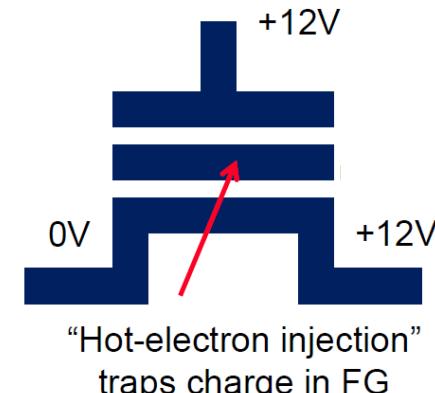
- Transistor with a second “floating” gate
- Floating gate can trap electrons
- This results in a detectable change in threshold voltage



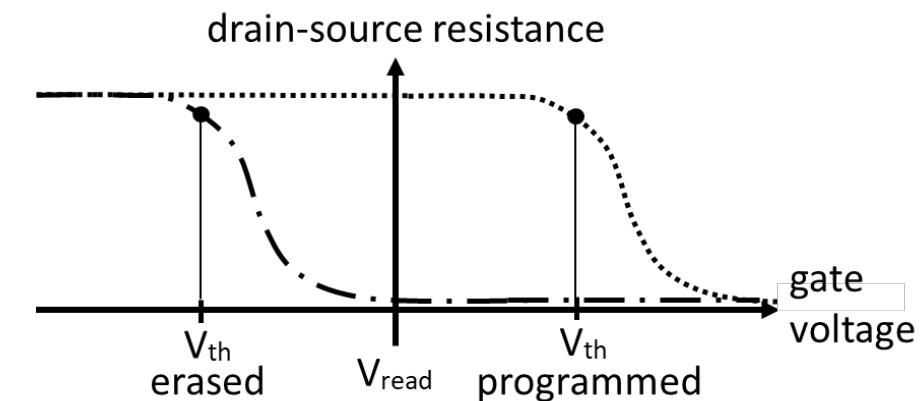
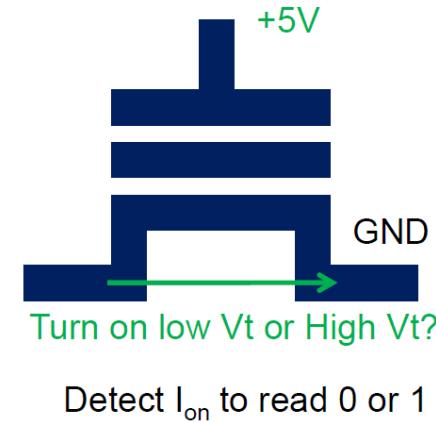
Erasing
to logical “1”



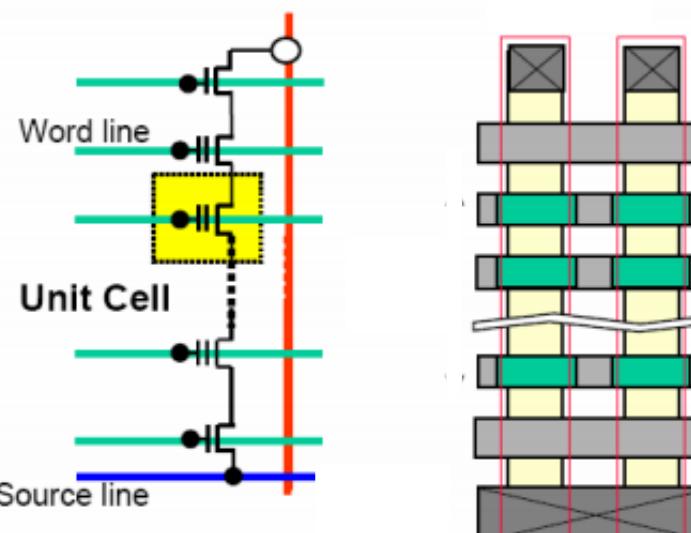
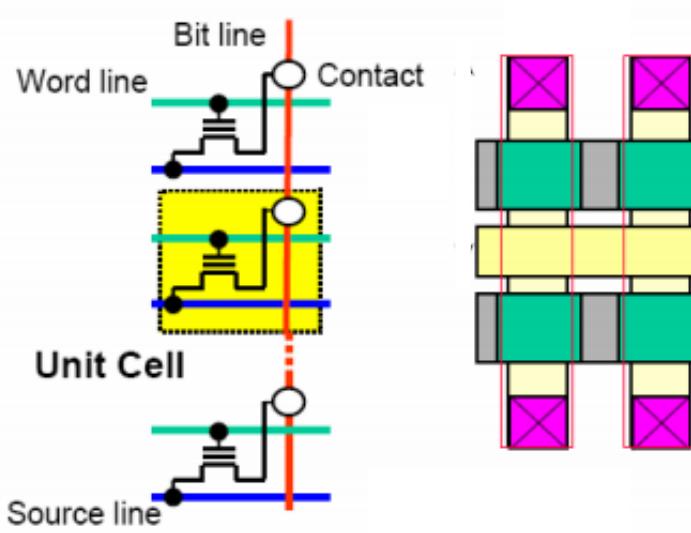
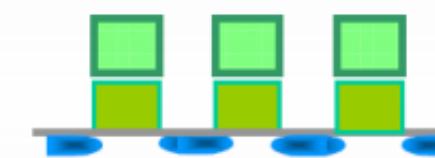
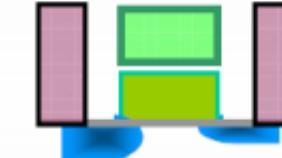
Programming (=writing)
to logical “0”



Reading



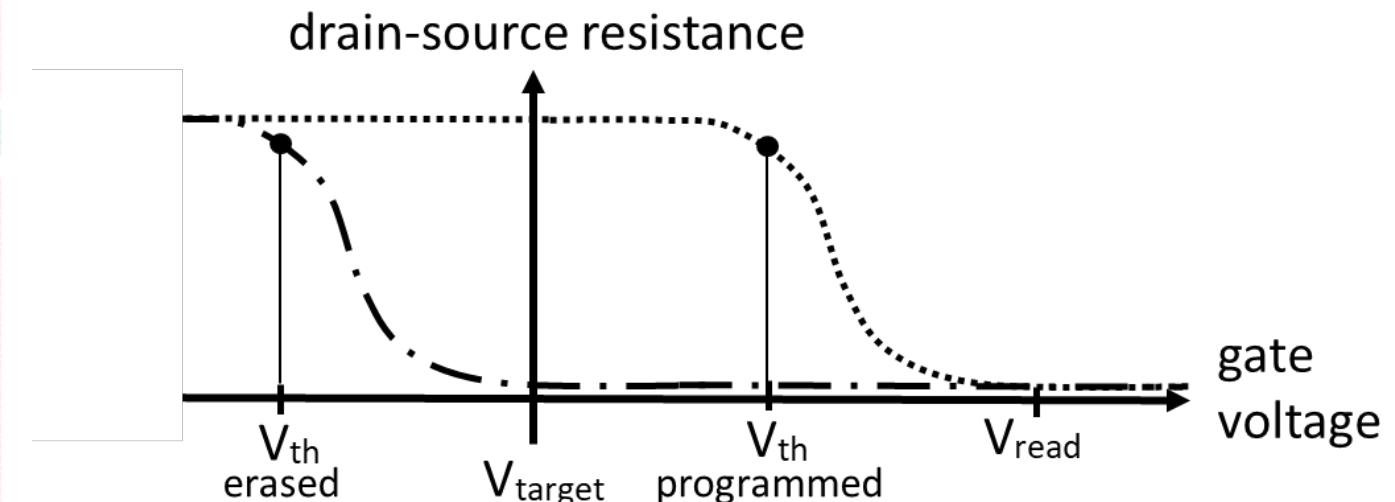
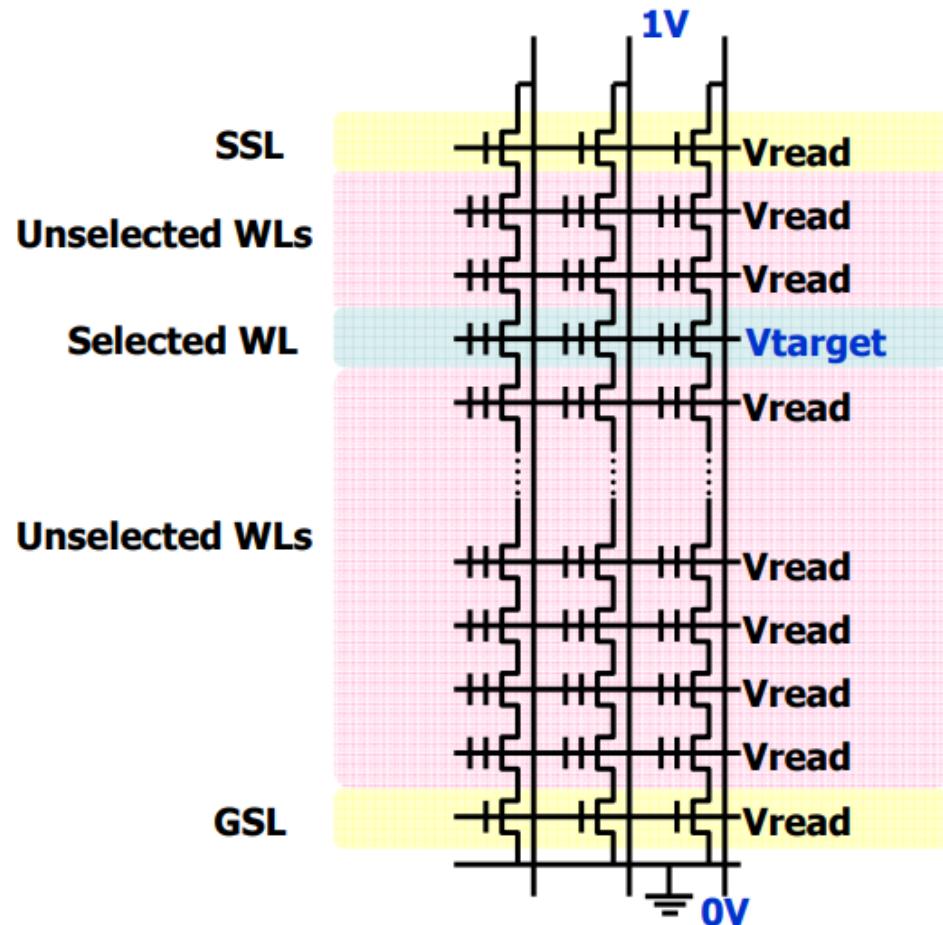
NAND and NOR Flash Memory

	NAND	NOR
Cell Array & Size		
Cross-section		
Features	<p>Small Cell Size, High Density Low Power → Mass Storage</p>	<p>Fast random access → Code Storage</p>

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

Available memory:

- The MSP432P401R processor 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

Memory map:

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

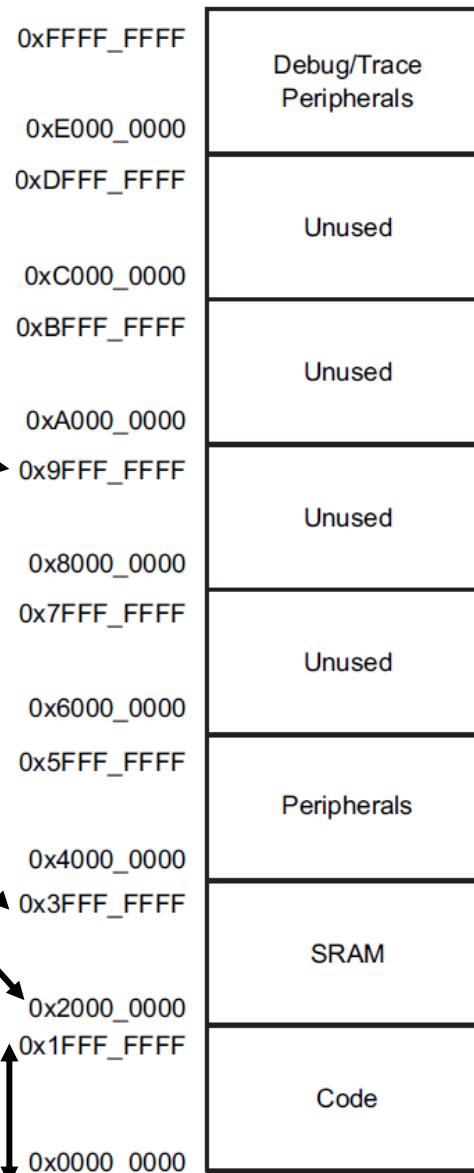
0011 1111 ... 1111

0010 0000 0000

diff. = 0001 1111 ... 1111 →

2^{29} different addresses

capacity = 2^{29} Byte =
512 MByte



Example: Memory Map in MSP432

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

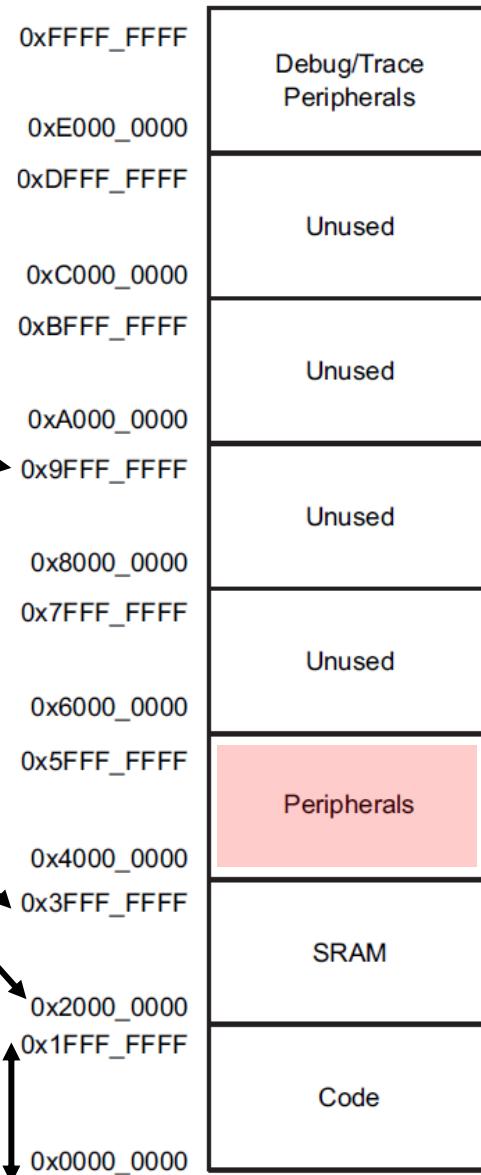


Table 6-21. Port Registers (Base Address: 0x4000_4C00)

REGISTER NAME	ACRONYM	OFFSET 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

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

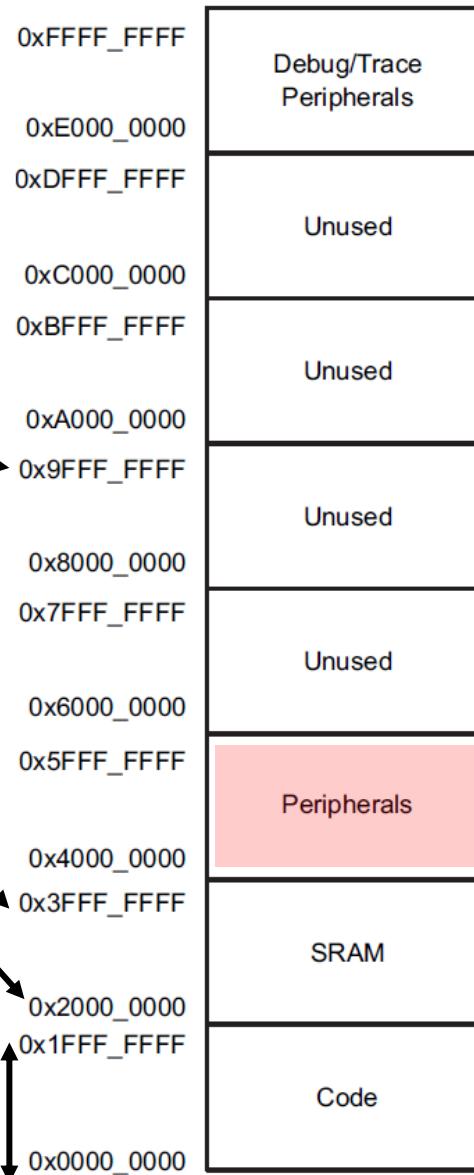


Table 6-21. Port Registers (Base Address: 0x4000_4C00)

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:

P1.0_LED1	4	P1.0/UCA0STE
P1.1_BUTTON1	5	P1.1/UCA0CLK
P1.2_BCI_UART_RXD	6	P1.2/UCA0RXD/UCA0SOMI
P1.3_BCI_UART_TXD	7	P1.3/UCA0TXD/UCA0SIMO
P1.4_BUTTON2	8	P1.4/UCB0STE
P1.5_SPICLK_J1.7	9	P1.5/UCB0CLK
P1.6_SPTMOSI_J2.15	10	P1.6/UCB0SIMO/UCB0SDA
P1.7_SPTMISO_J2.14	11	P1.7/UCB0SOMI/UCB0SCL

LED1 is connected to Port 1, Pin 0

How do we toggle LED1 in a C program?

Example: Memory Map in MSP432

Memory map:

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

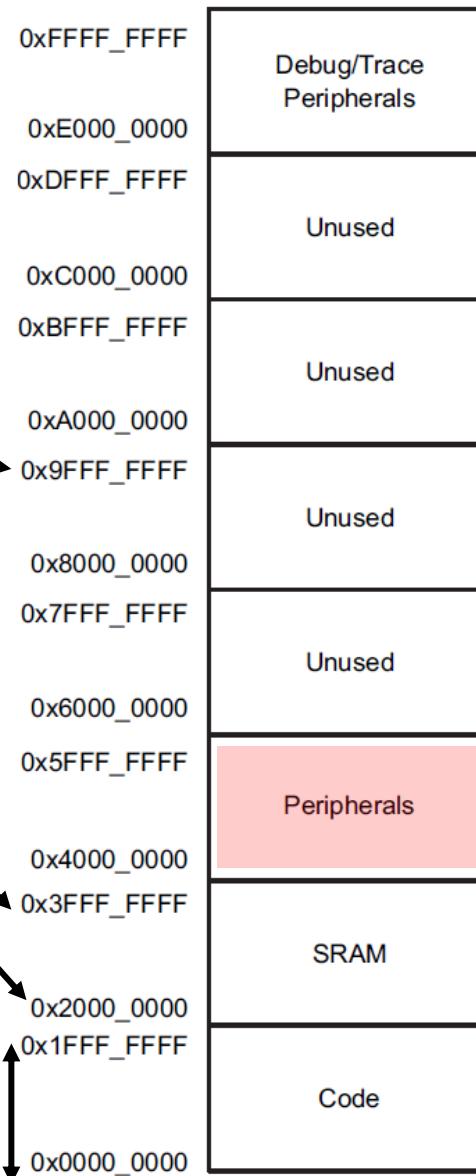
0011 1111 ... 1111

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).

```
...
//declare plout as a pointer to an 8Bit integer
volatile uint8_t* plout;

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

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

^ : XOR

Example: Memory Map in MSP432

Memory map:

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

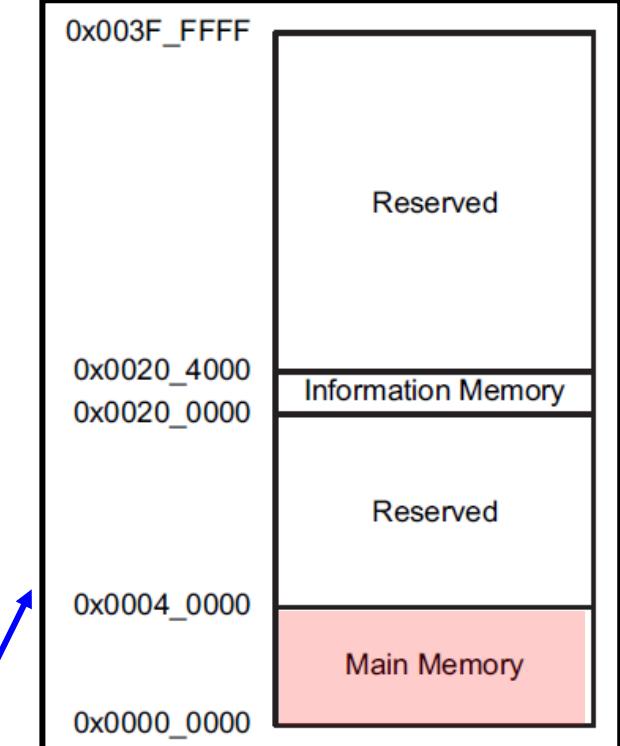
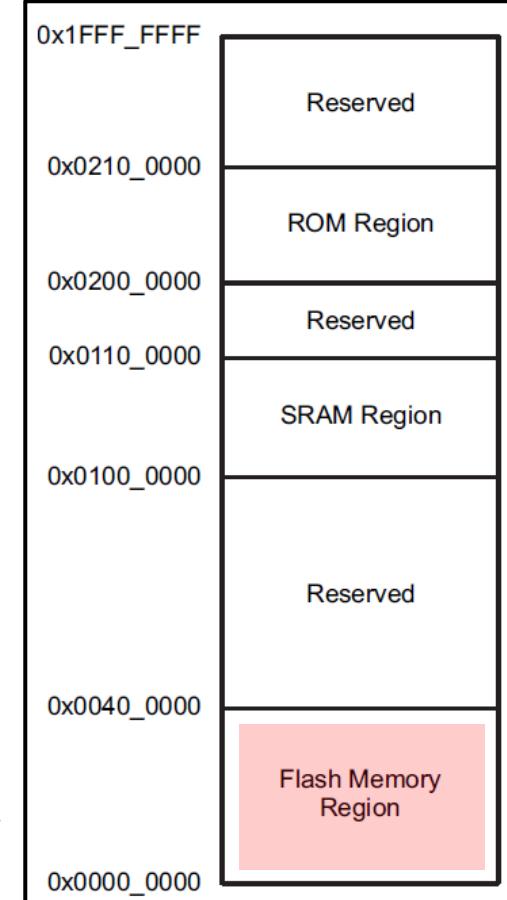
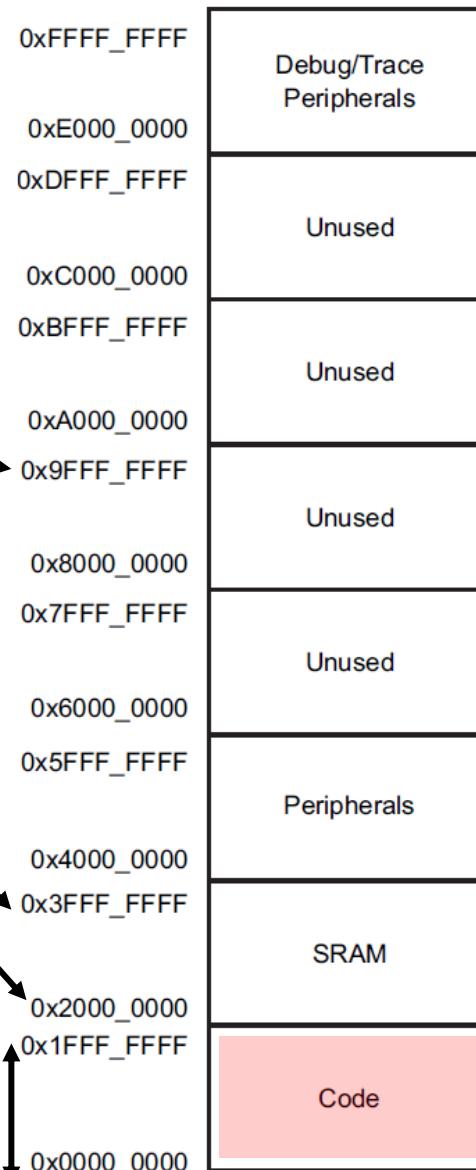
0011 1111 ... 1111

0010 0000 ... 0000

diff. = 0001 1111 ... 1111 →

2^{29} different addresses

capacity = 2^{29} Byte = 512 MByte



- 0x3FFF address difference = $4 * 2^{16}$ different addresses → 256 kByte maximal data capacity for Flash Main Memory
- Used for program, data and non-volatile configuration.

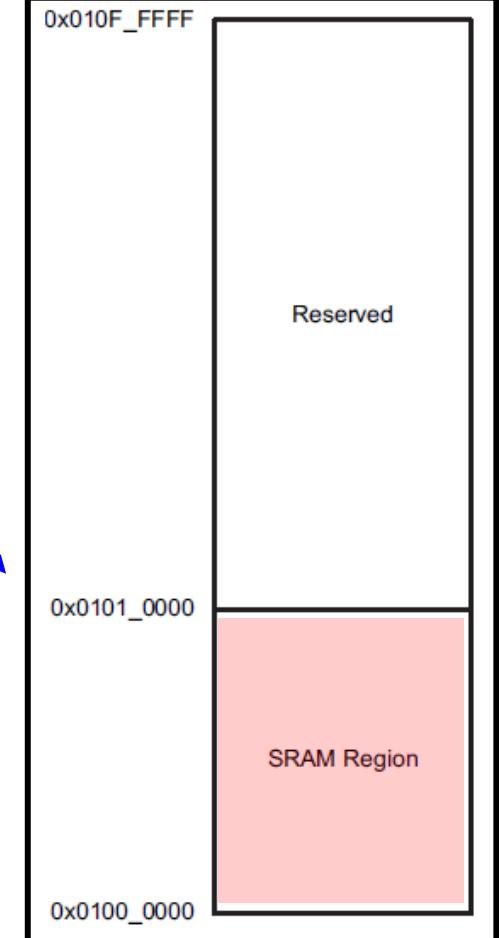
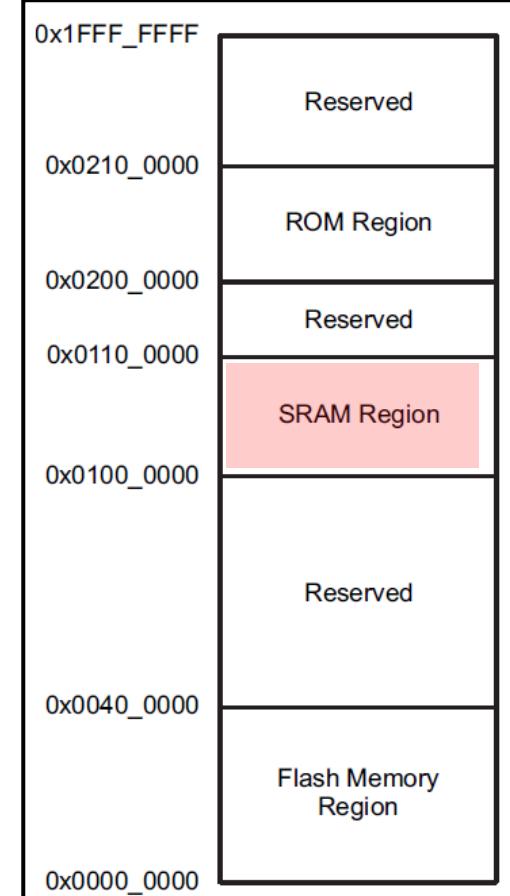
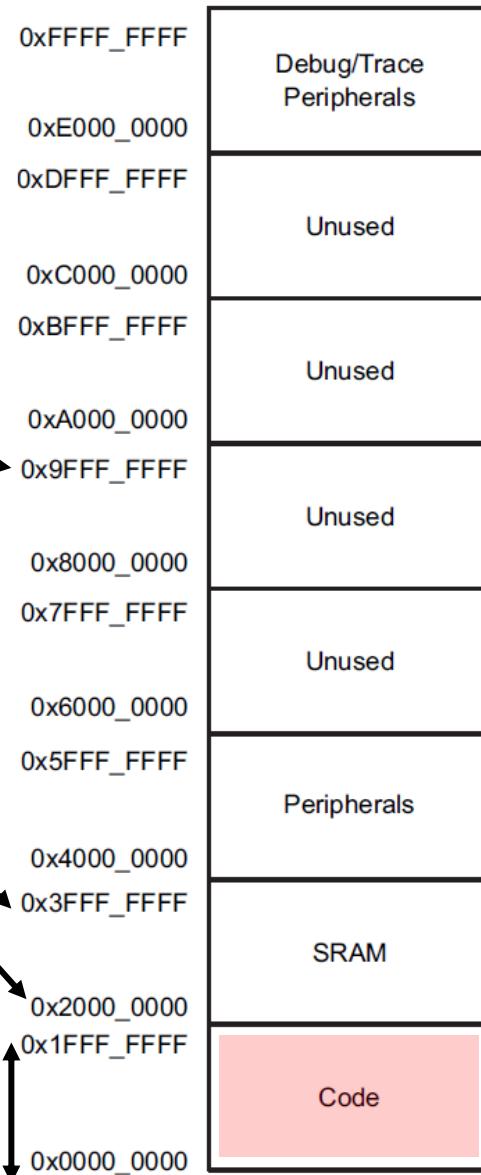
Example: Memory Map in MSP432

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).

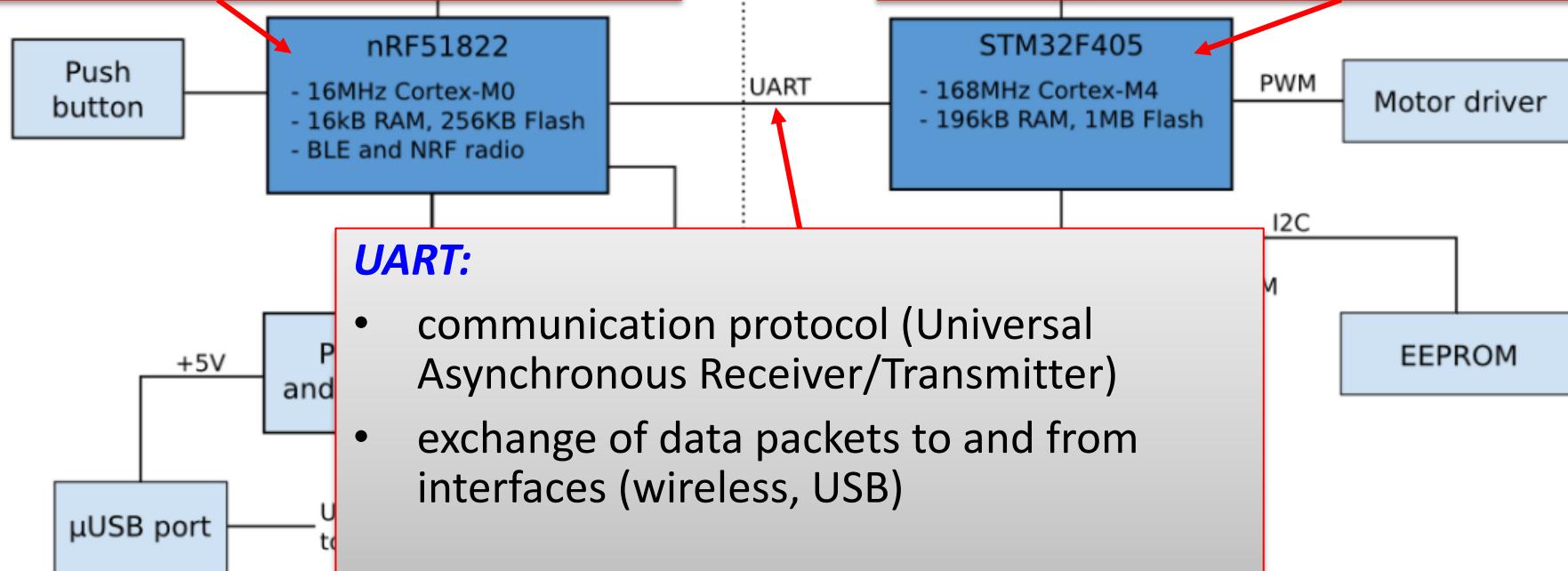
Remember?

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



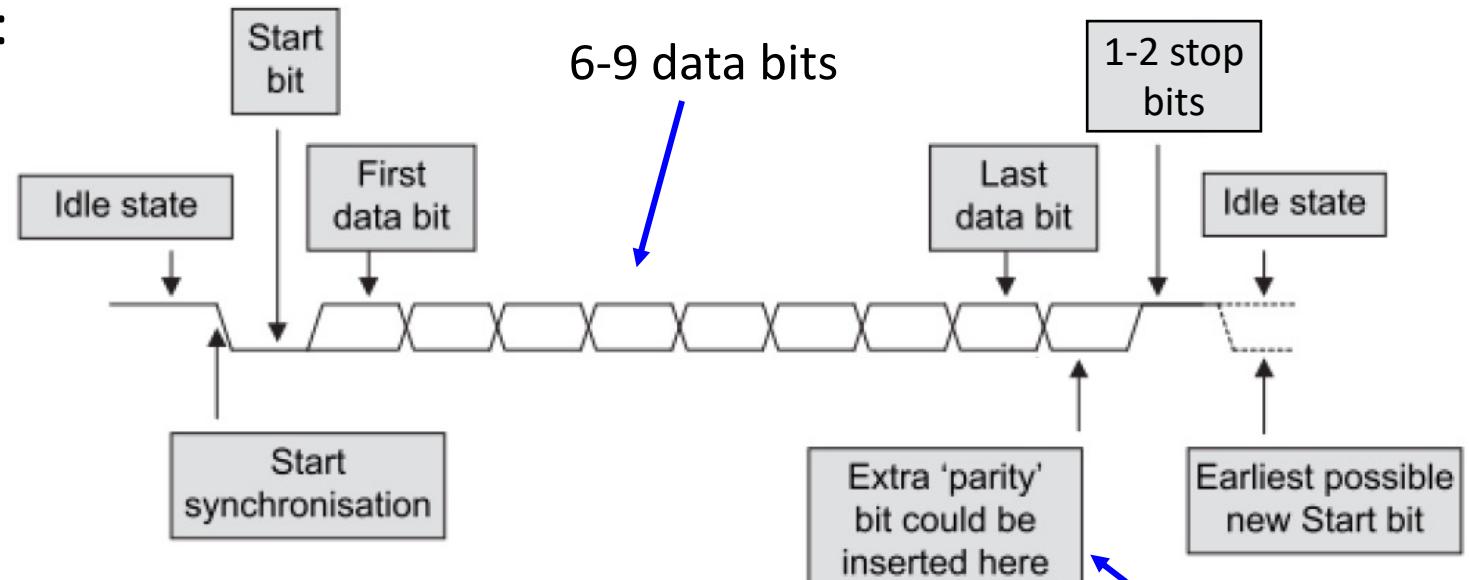
Crazyflie 2.0 system architecture

Input and Output

UART Protocol

UART

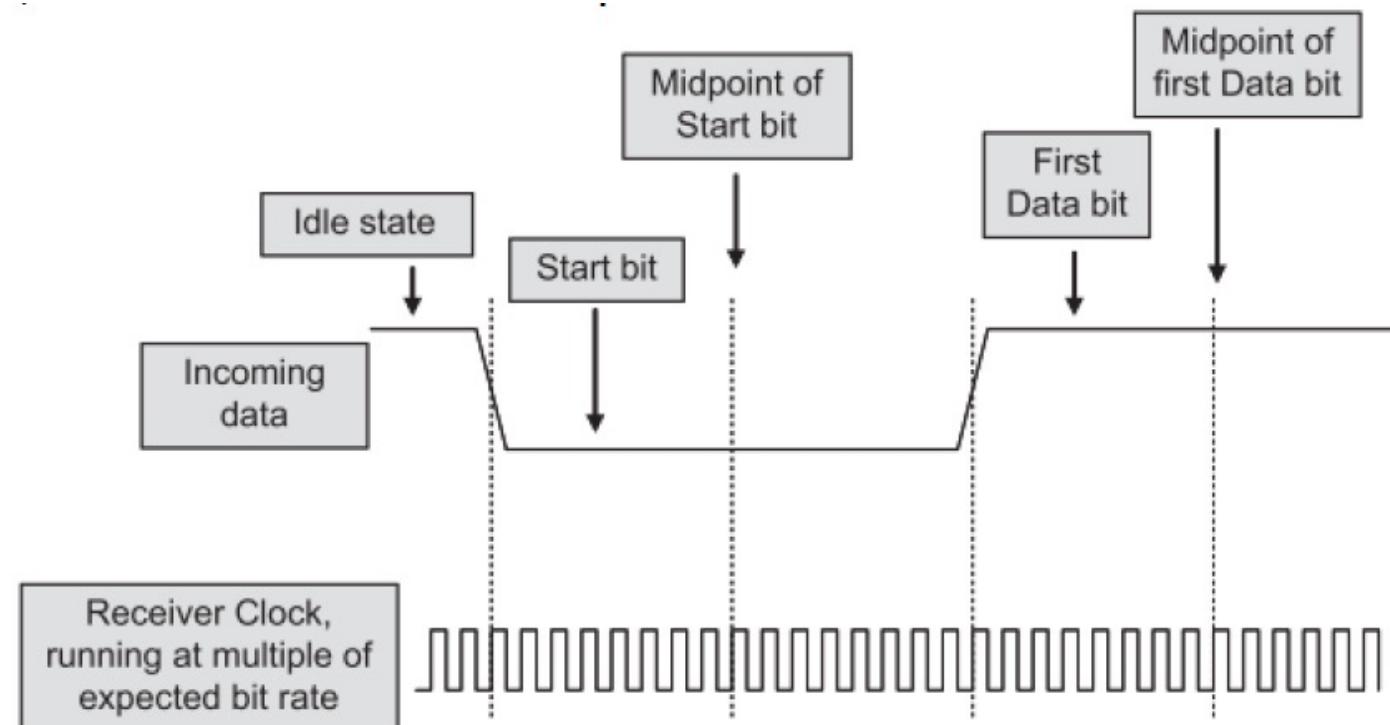
- *Serial communication* of bits via a single signal, i.e. UART provides parallel-to-serial and serial-to-parallel conversion.
- Sender and receiver need to *agree on the transmission rate*.
- Transmission of a serial packet starts with a start bit, followed by data bits and finalized using a stop bit:



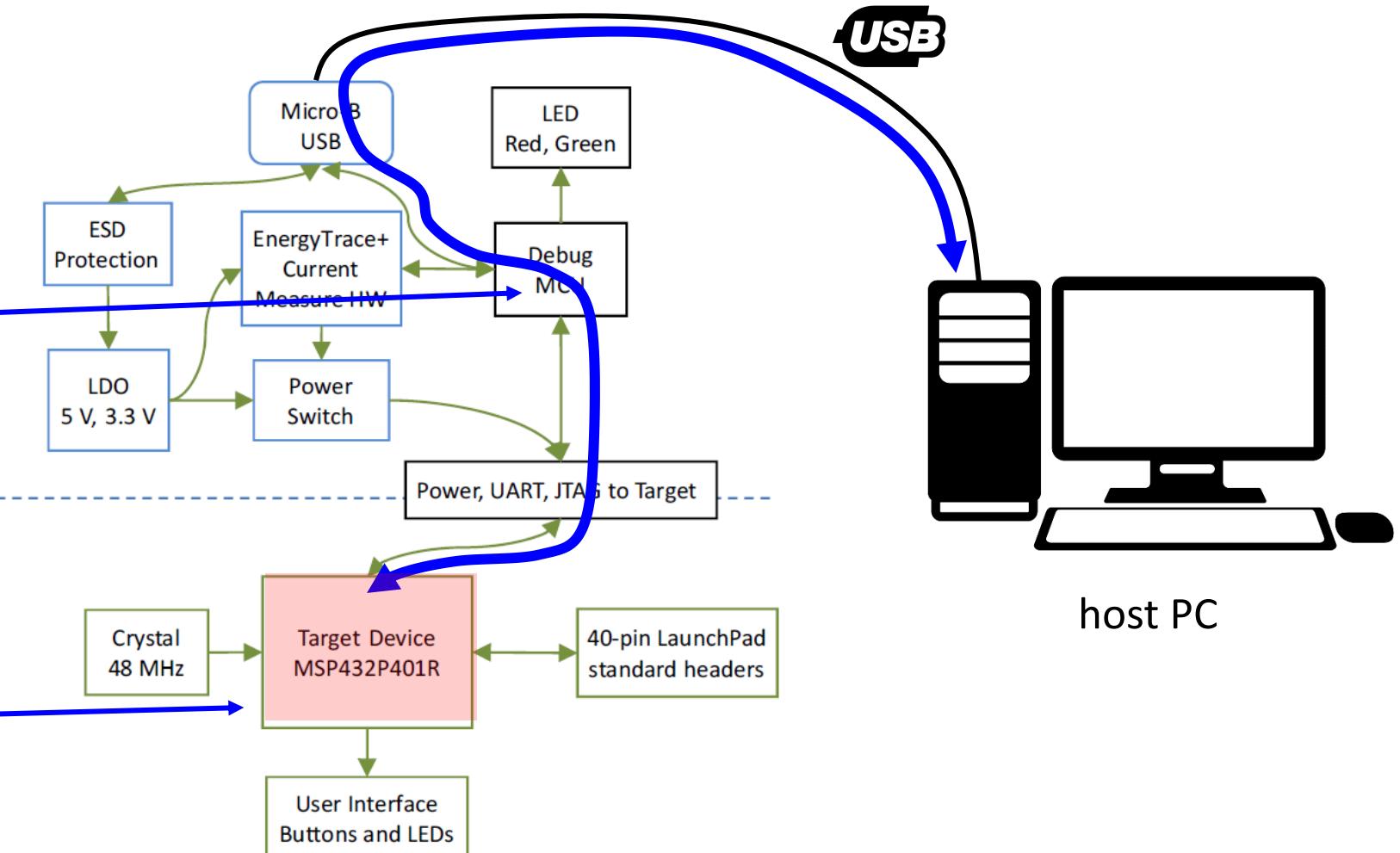
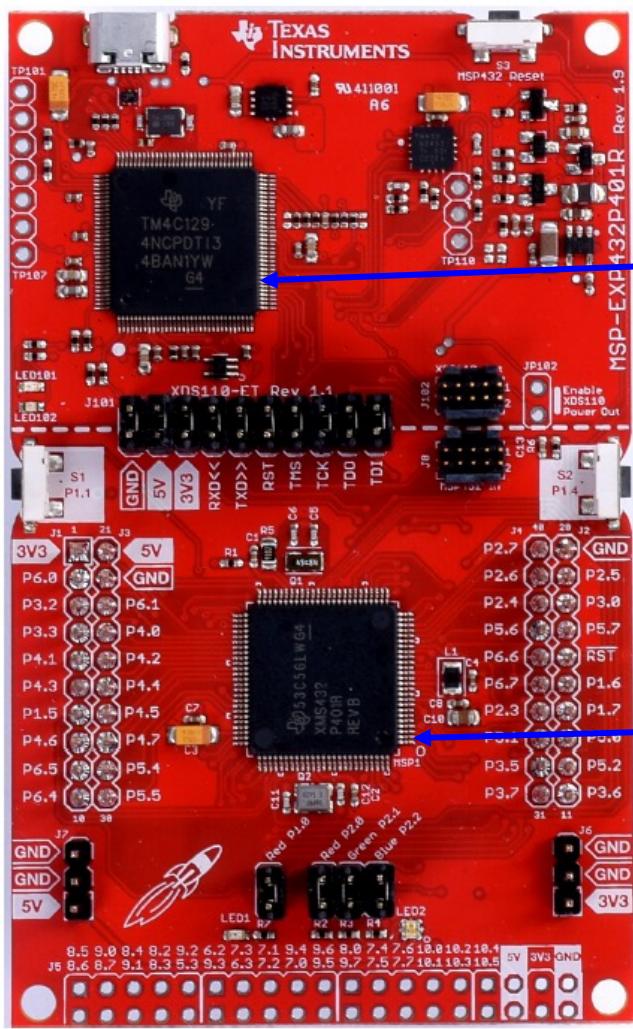
- There exist many variations of this simple scheme.
for detecting single bit errors

UART

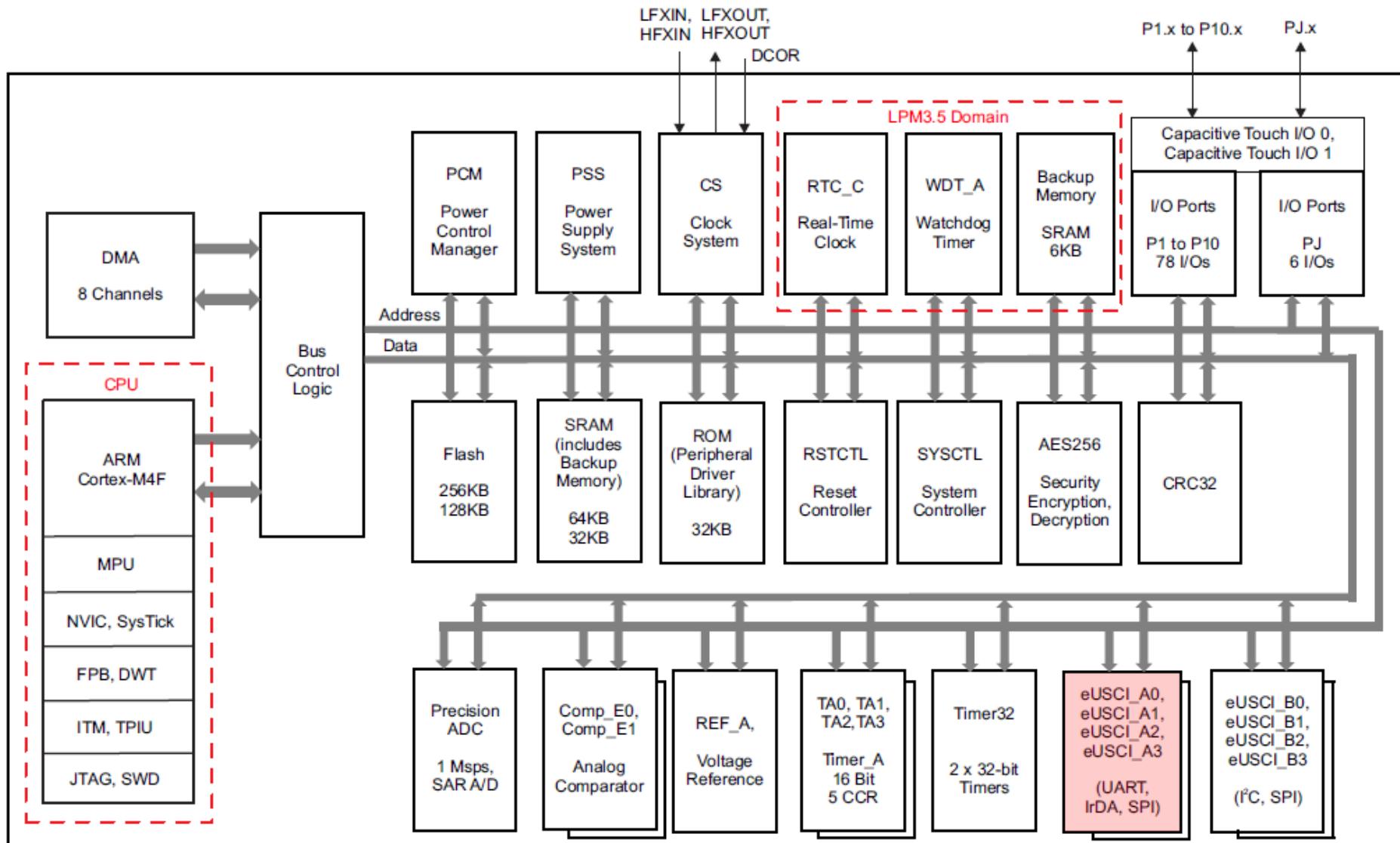
- The receiver runs an *internal clock* whose frequency is an exact multiple of the expected bit rate.
- When a *Start bit* is detected, a counter begins to count clock cycles e.g. 8 cycles until the midpoint of the anticipated Start bit is reached.
- The clock counter counts a further 16 cycles, to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



UART with MSP432



UART with MSP432



Input and Output

Memory Mapped Device Access

Memory-Mapped Device Access

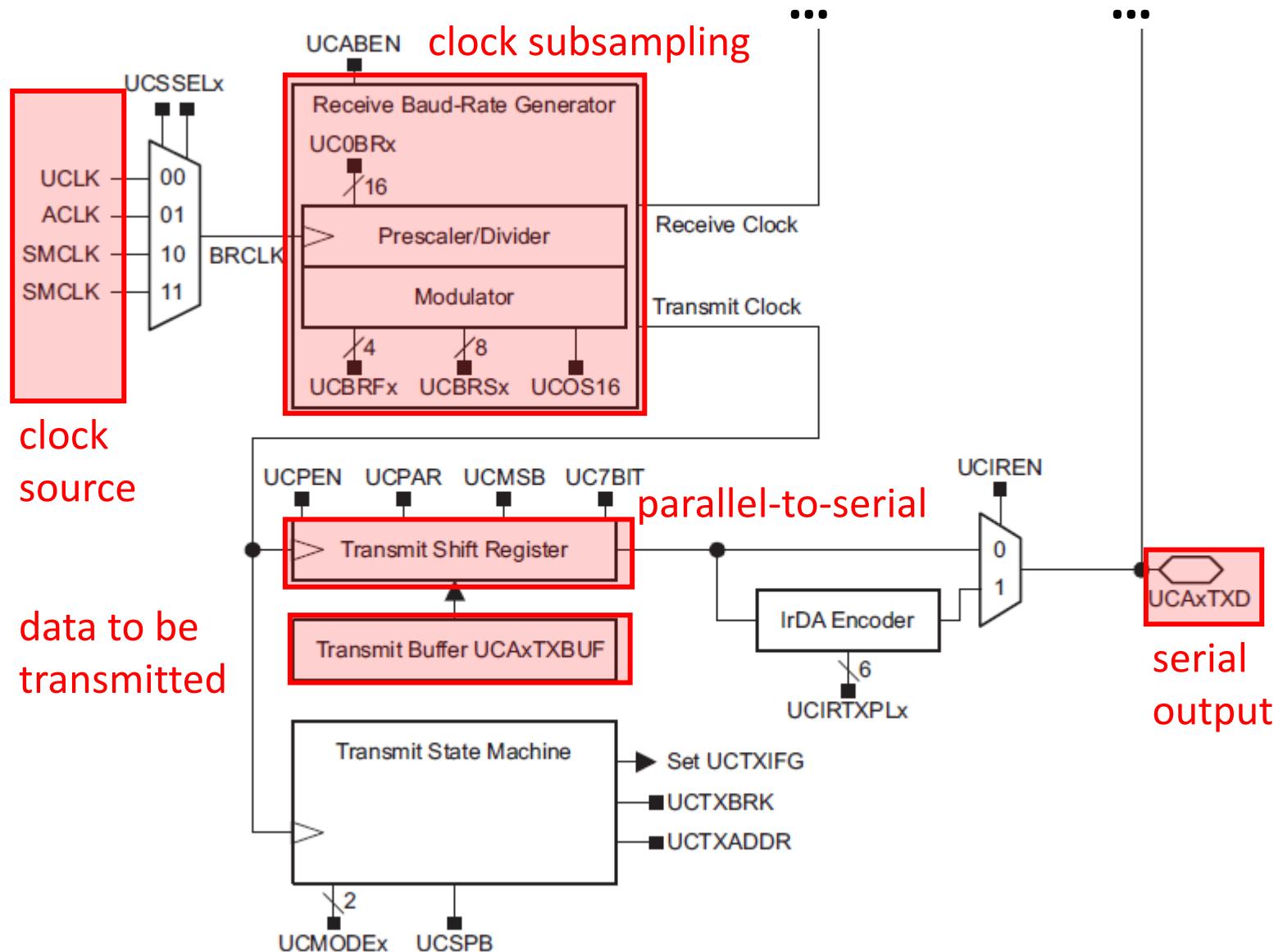
eUSCI_A0 Registers (Base Address: 0x4000_1000)

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)
in our case: bit/s
 - LSB or MSB first
 - number of bits per packet
 - parity bit
 - number of stop bits
 - interrupt-based communication
 - clock source

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:

- Let us assume $SMCLK = 3MHz$
- Quartz frequency = 48 MHz, is divided by 16 before connected to $SMCLK$

Example:

- Transmission rate 4800 bit/s
- 16 clock periods per bit (see 3-34)
- 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. 0b = No interrupt pending 1b = Interrupt pending

shift 1 bit to the right

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

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

Introduction to Embedded Systems

3. Hardware Software Interface

Prof. Dr. Marco Zimmerling



Organization

Join ILIAS course:

- Login: RZ username + password
- Course password: **es-0x8af**

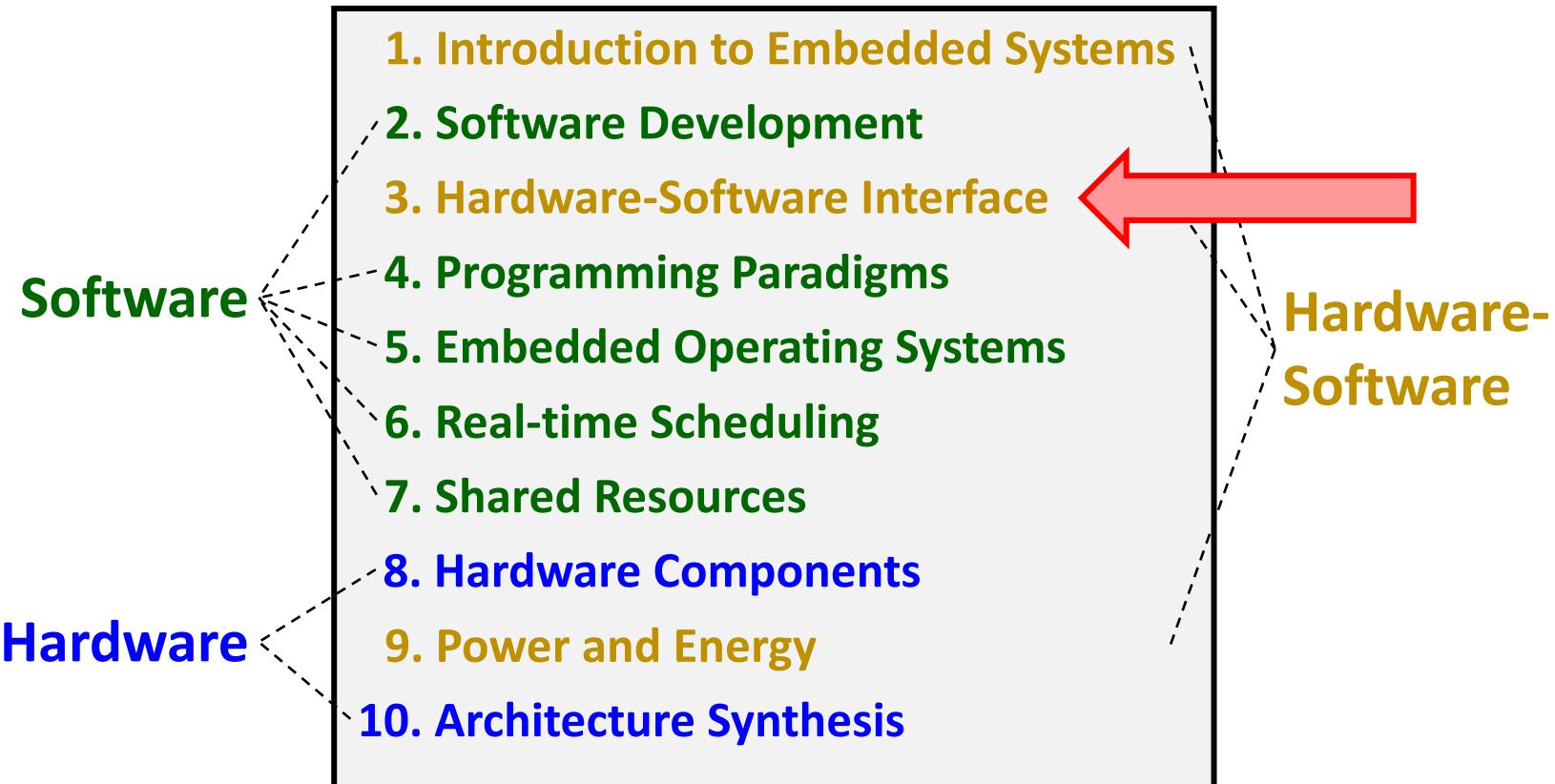
Exercises:

- From 12:00 to 14:00 in HS 00-038 (German) and Kinohörsaal (English)
- Today (November 8):
 - First exercise sheet [released](#) + [overview](#) of tasks given during exercise session
- Next week (November 15):
 - [Solutions](#) of first exercise sheet presented during exercise session
 - Second exercise sheet [released](#) + [overview](#) of tasks given during exercise session

Exam: [March 4, 2023](#) from 10:00 to 12:00, room TBD



Where we are ...



What you will learn ...

Hardware-Software Interfaces in Embedded Systems

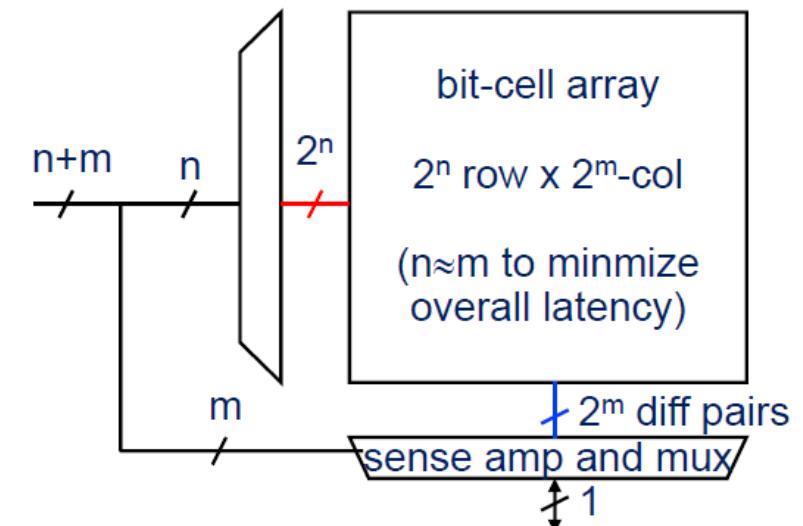
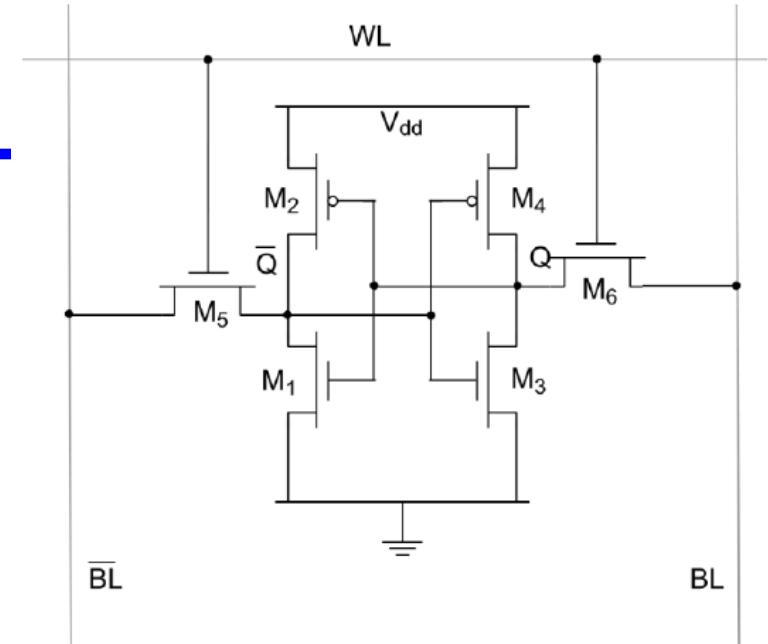
- *Storage*
 - SRAM / DRAM / Flash
 - Memory Map
- *Input and Output*
 - UART Protocol
 - Memory Mapped Device Access
 - SPI Protocol
- *Interrupts*
- *Clocks and Timers*
 - Clocks
 - Watchdog Timer
 - System Tick
 - Timer and PWM

Storage

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 2^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



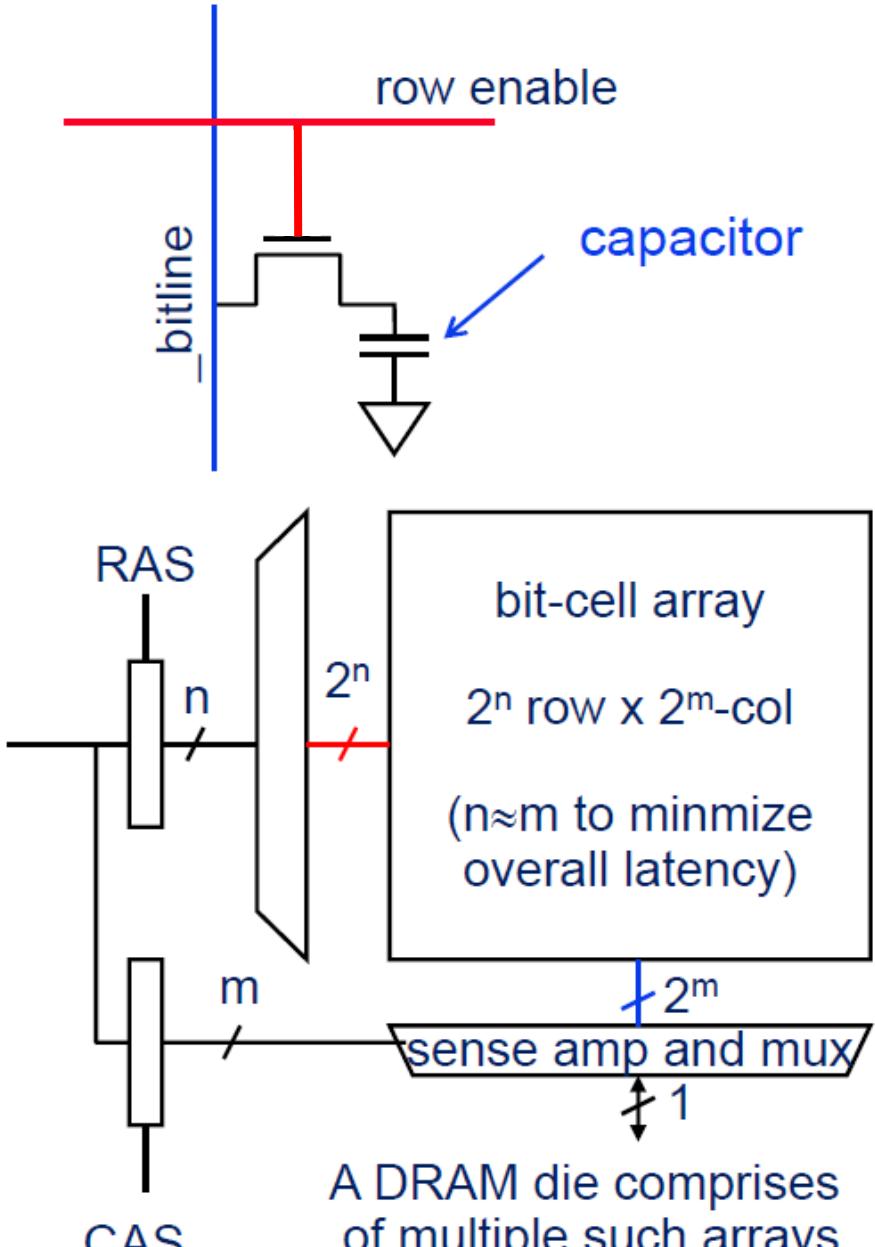
Dynamic Random Access (DRAM)

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



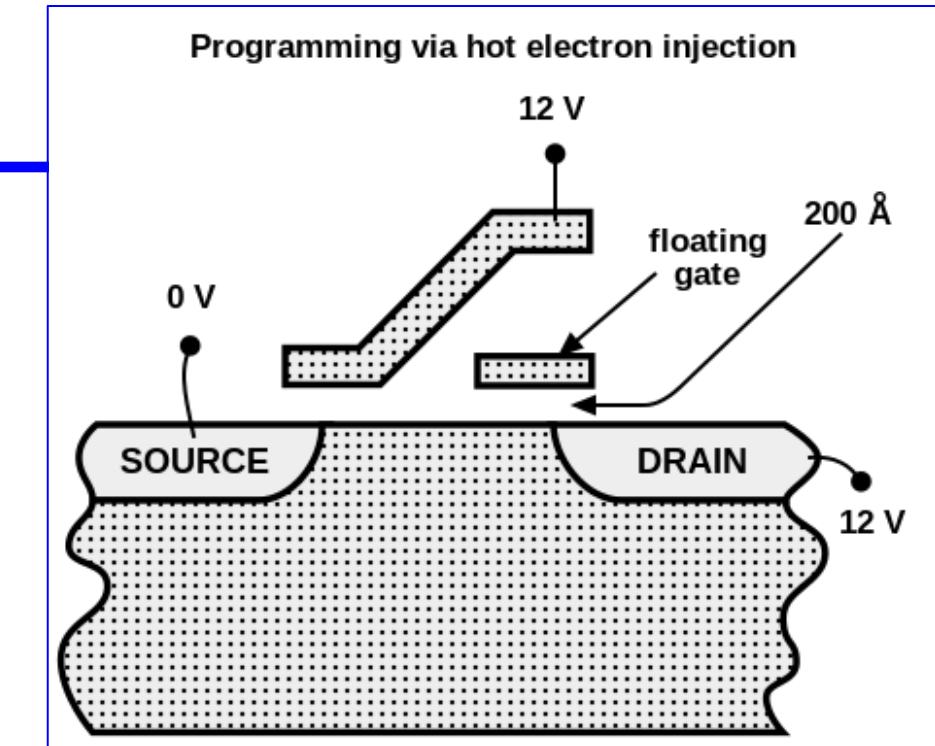
(RAS/CAS = row/column address select)

Flash Memory

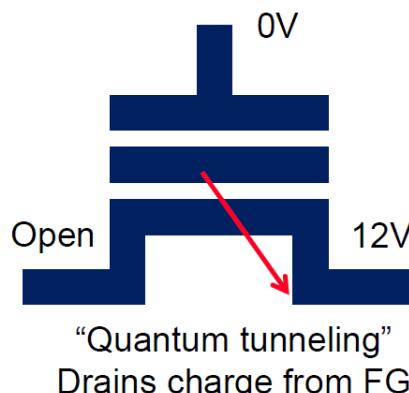
Electrically modifiable, non-volatile storage

Principle of operation:

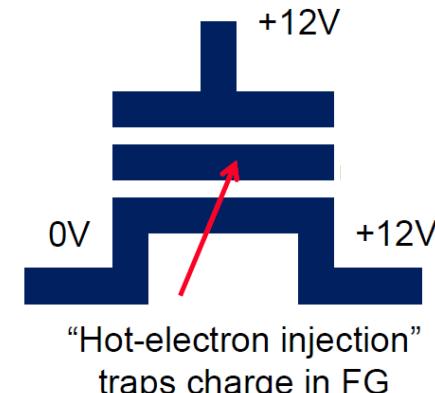
- Transistor with a second “floating” gate
- Floating gate can trap electrons
- This results in a detectable change in threshold voltage



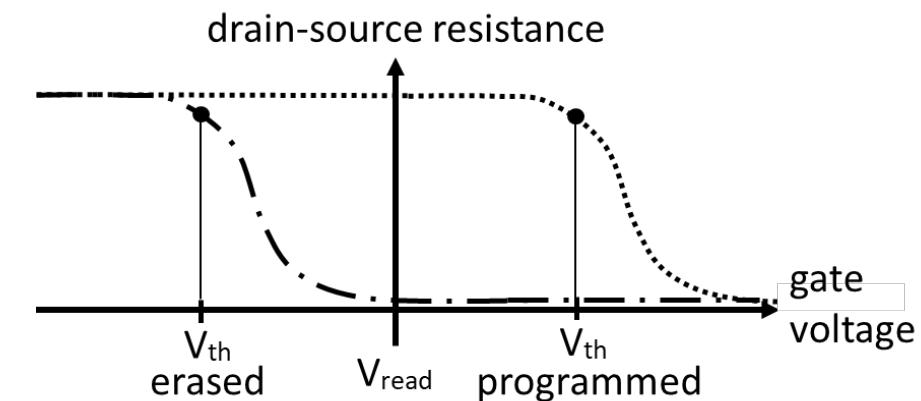
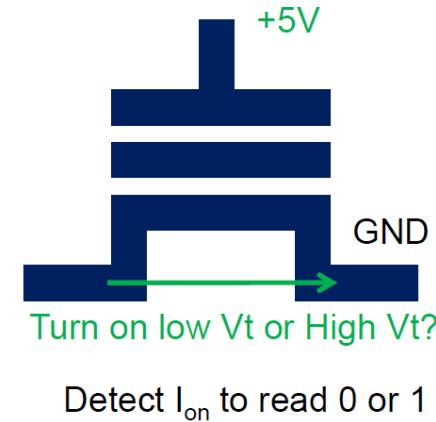
Erasing
to logical “1”



Programming (=writing)
to logical “0”



Reading



Example: Memory Map in MSP432

Memory map:

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

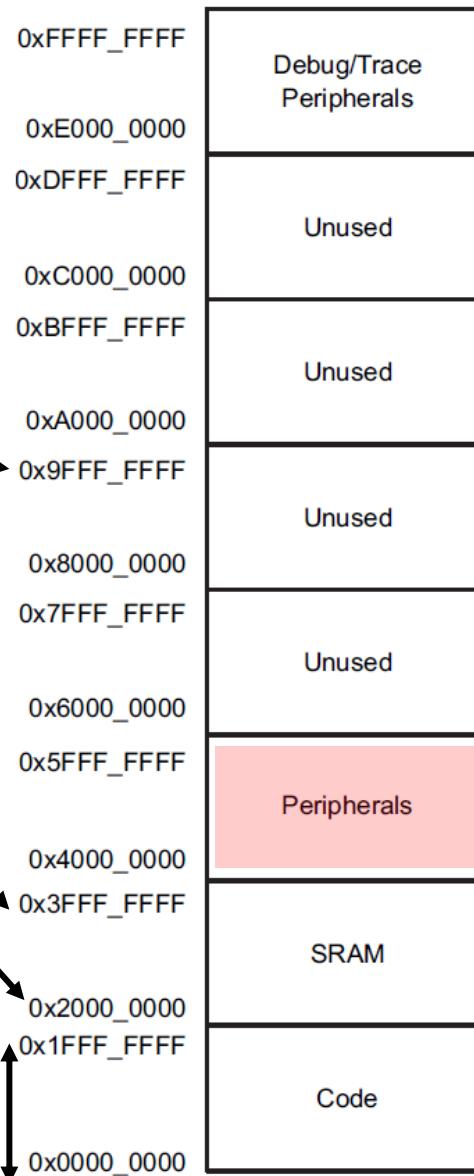
0011 1111 ... 1111

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).

```
...
//declare plout as a pointer to an 8Bit integer
volatile uint8_t* plout;

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

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

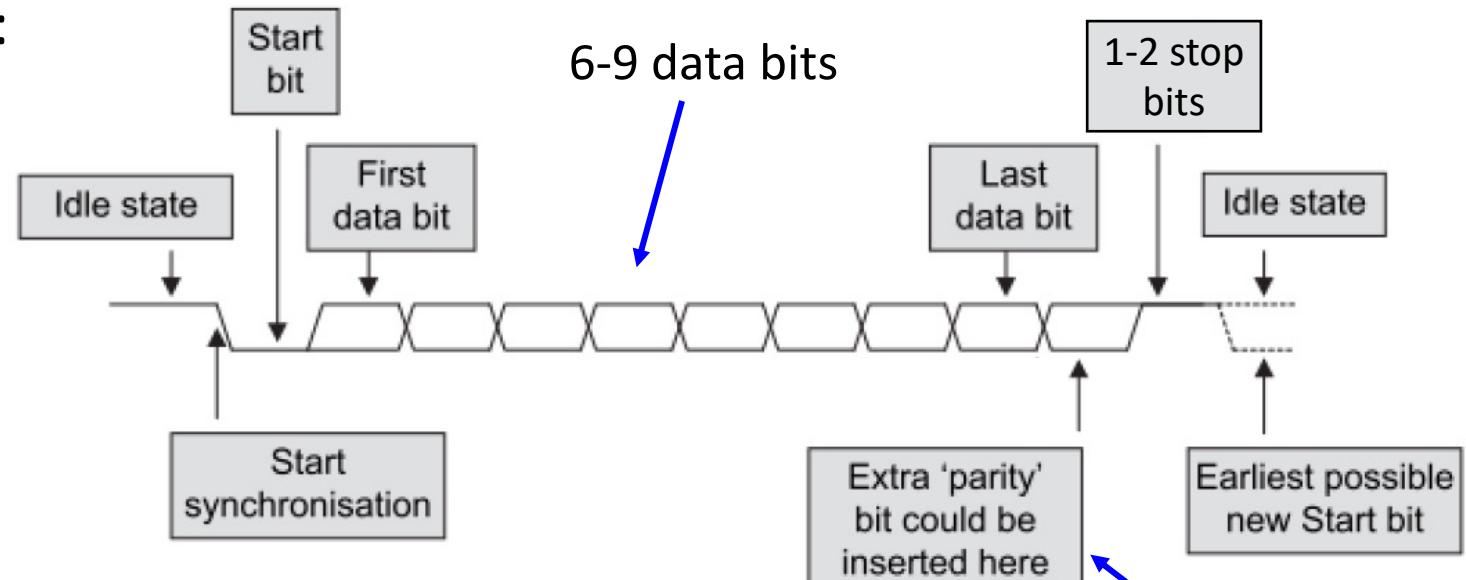
^ : XOR

Input and Output

UART Protocol

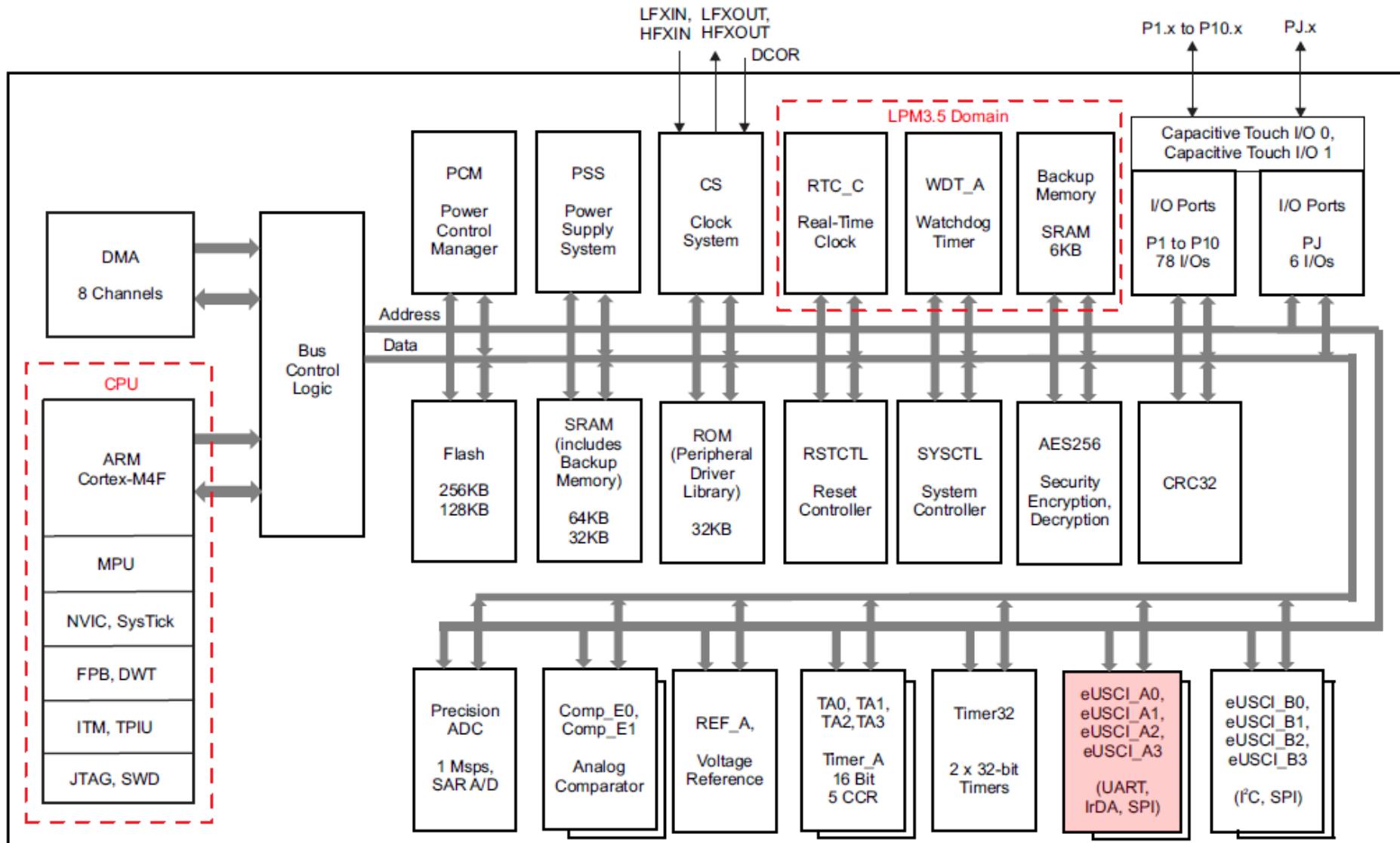
UART

- *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 with MSP432

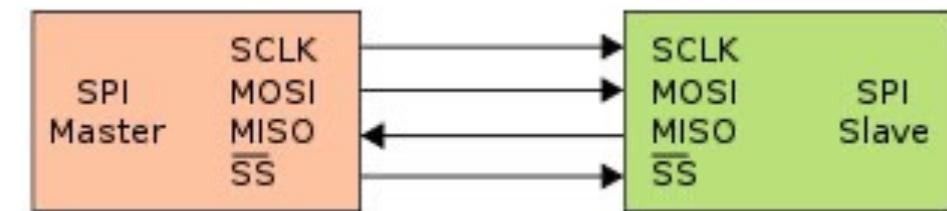


Input and Output

SPI Protocol

SPI (Serial Peripheral Interface Bus)

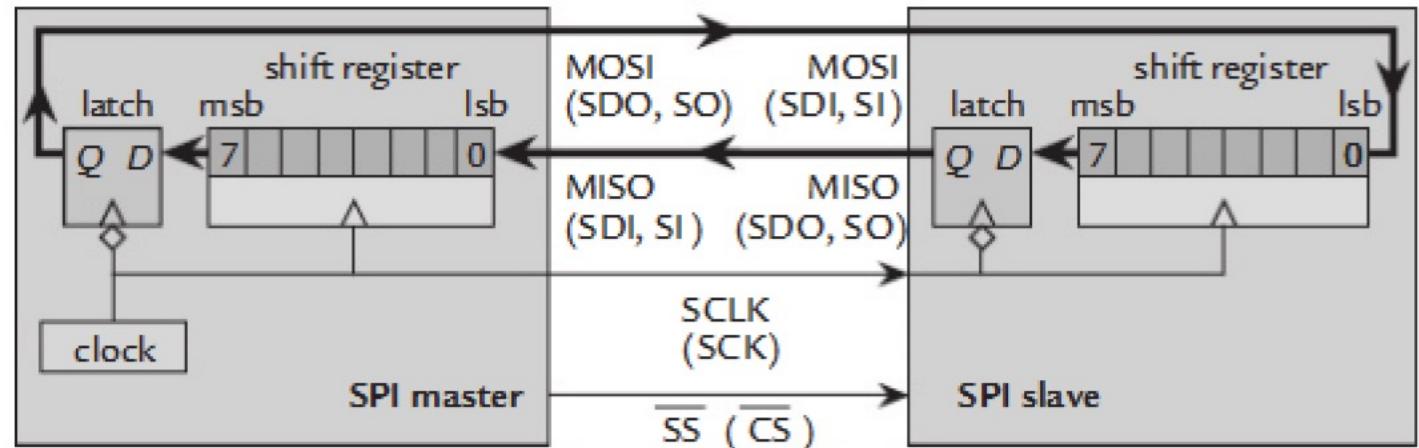
- Typically *communicate across short distances*
- *Characteristics:*
 - 4-wire synchronized (clocked) communications bus
 - supports single master and multiple slaves
 - always full-duplex: Communicates in both directions simultaneously
 - multiple Mbps transmission speeds can be achieved
 - transfer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



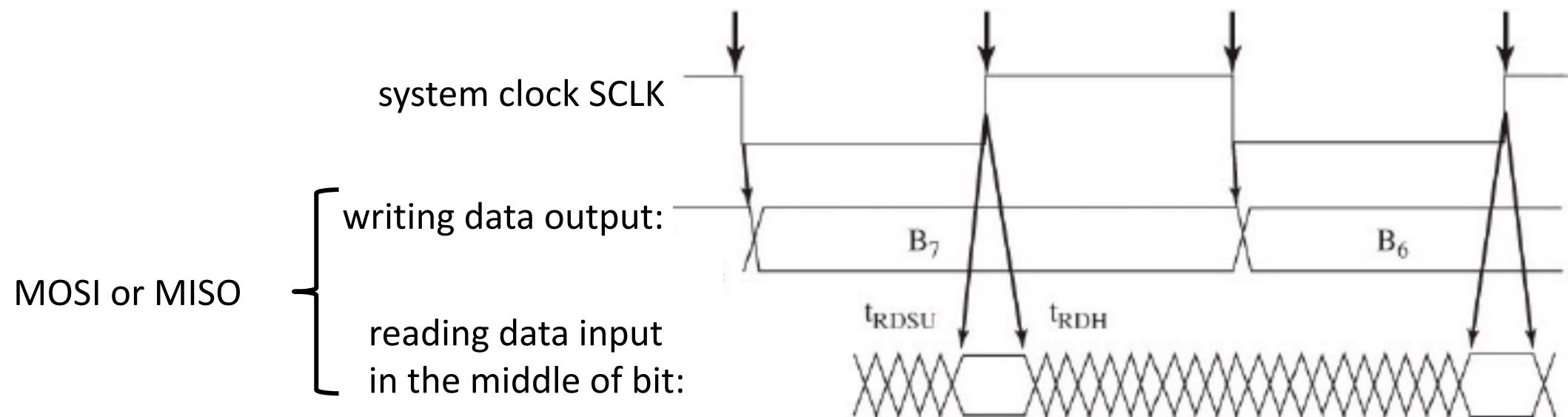
SPI (Serial Peripheral Interface Bus)

More detailed circuit diagram:

- details vary between different vendors and implementations

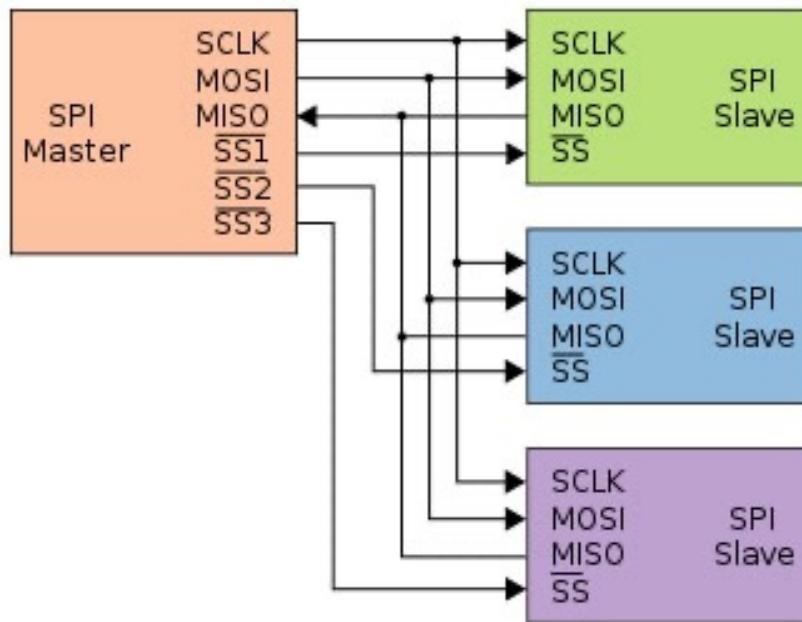


Timing diagram:



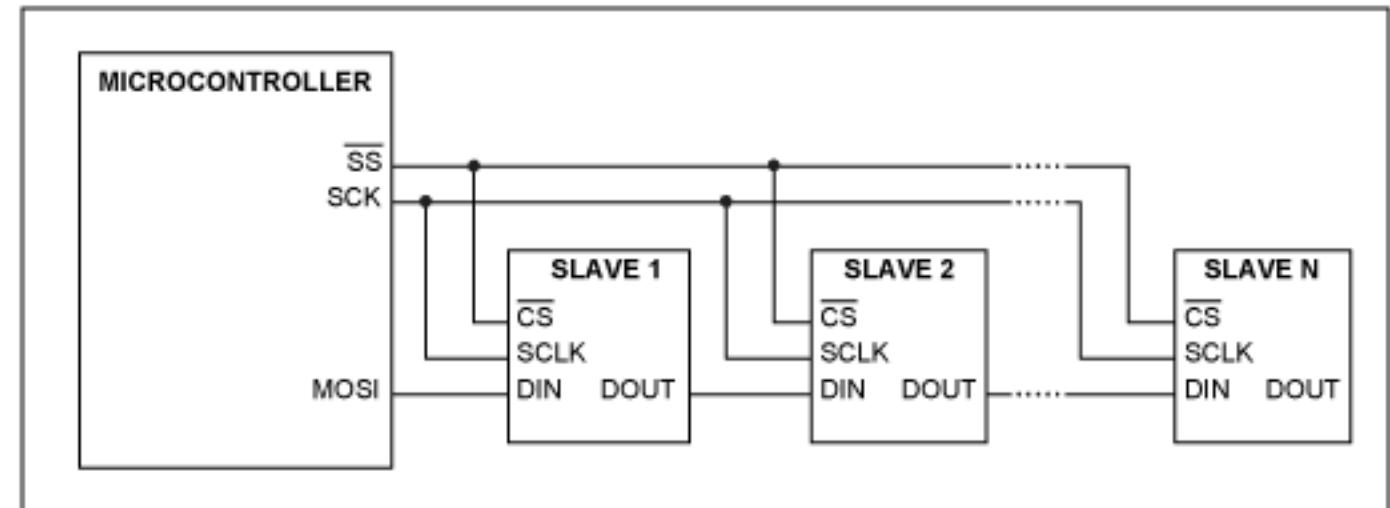
SPI (Serial Peripheral Interface Bus)

Two examples of bus configurations:



Master and multiple independent slaves

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



Master and multiple daisy-chained slaves

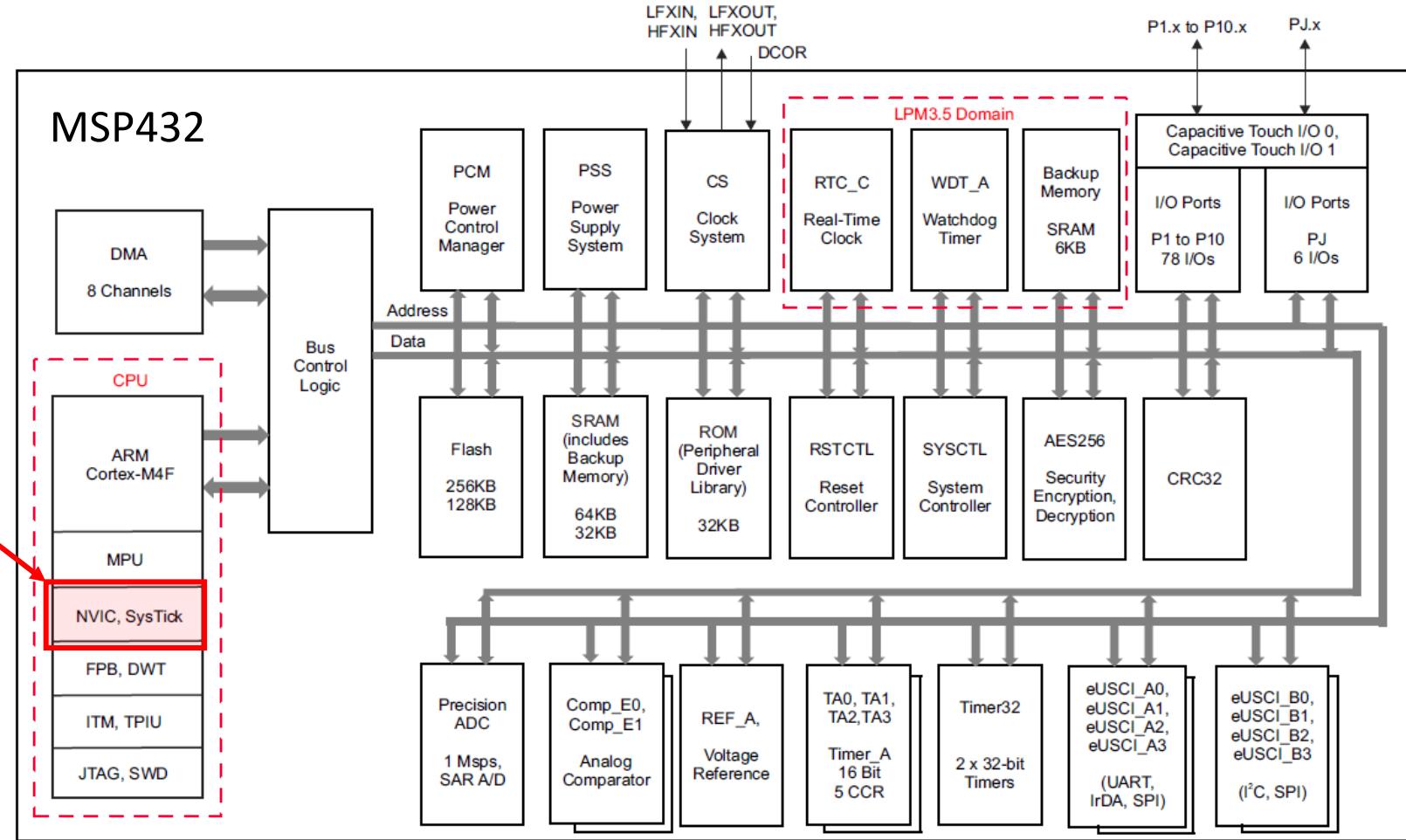
http://www.maxim-ic.com/appnotes.cfm/an_pk/3947

Interrupts

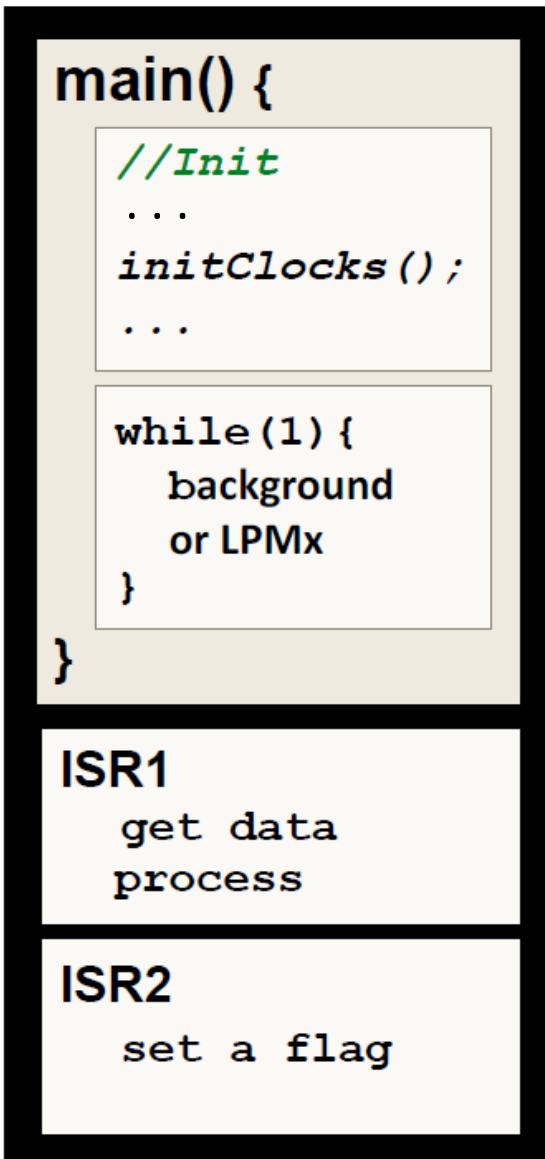
Interrupts

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

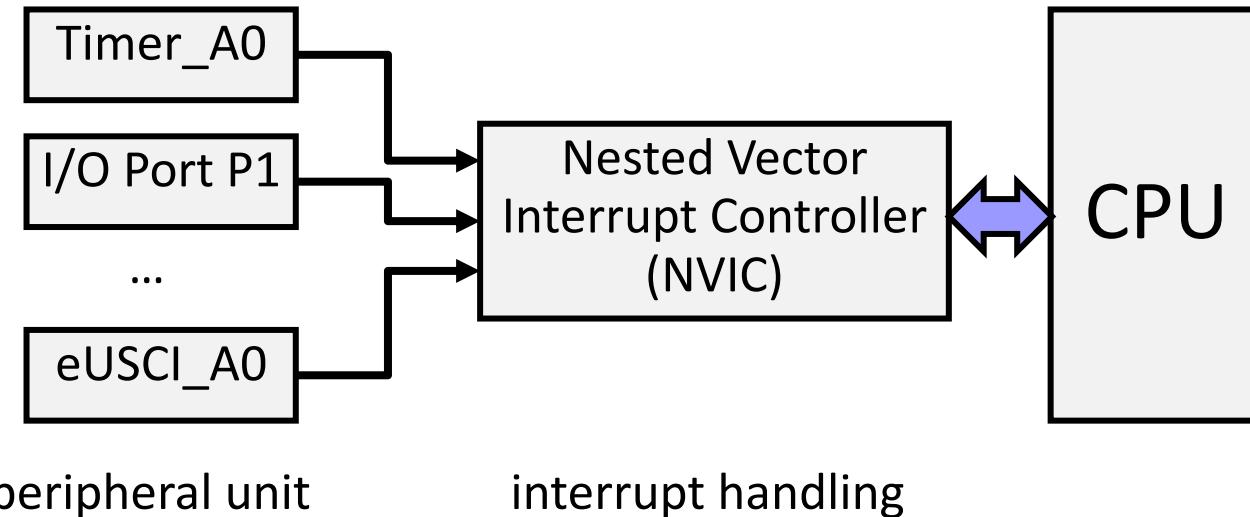
Background

- ◆ 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)



The *vector interrupt controller (NVIC)*

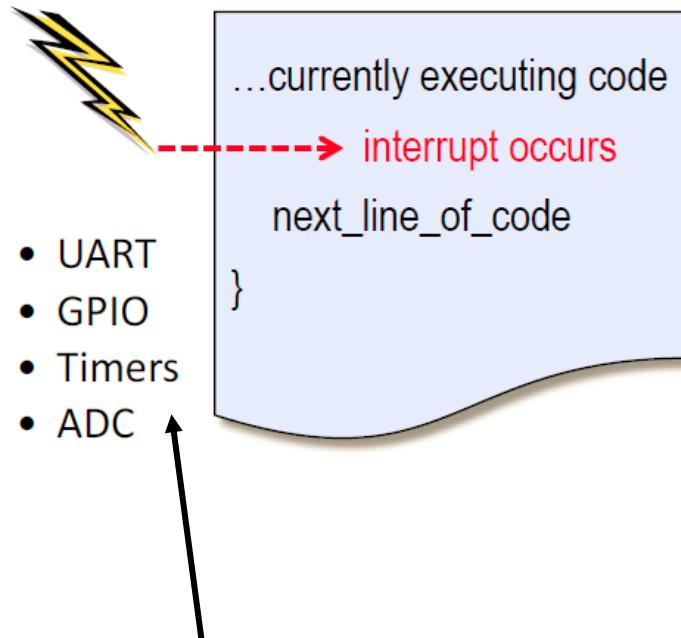
- enables and disables interrupts
- allows to individually and globally *mask interrupts* (disable reaction to interrupt), and
- registers *interrupt service routines* (ISR), sets the priority of interrupts.

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).

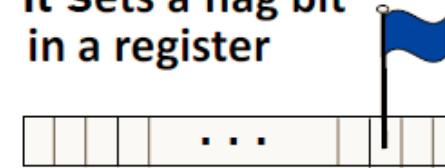
Processing of an Interrupt

1. An interrupt occurs



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

2. It sets a flag bit in a register



- When an interrupt signal is received, a corresponding bit is set in an IFG register.
- There is such an IFG register for each interrupt source.
- As some interrupt sources are only on for a short duration, the IFG register stores the fact that an interrupt has happened.

Processing of an Interrupt

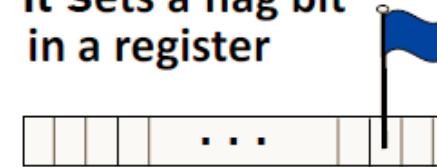
1. An interrupt occurs



...currently executing code
-----> interrupt occurs
next_line_of_code
}

- UART
- GPIO
- Timers
- ADC
- Etc.

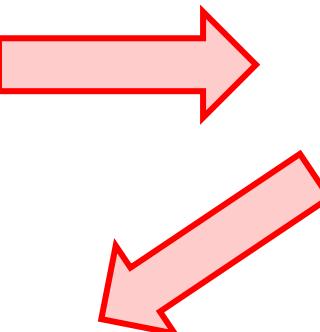
2. It sets a flag bit in a register



interrupt flag (IFG)
register

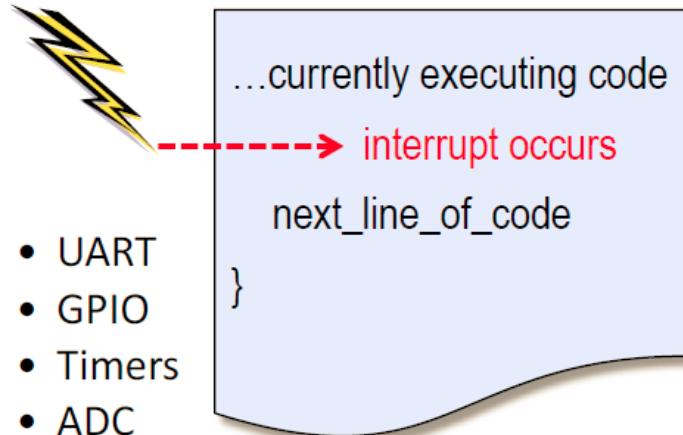
3. CPU/NVIC acknowledges interrupt by:

- 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

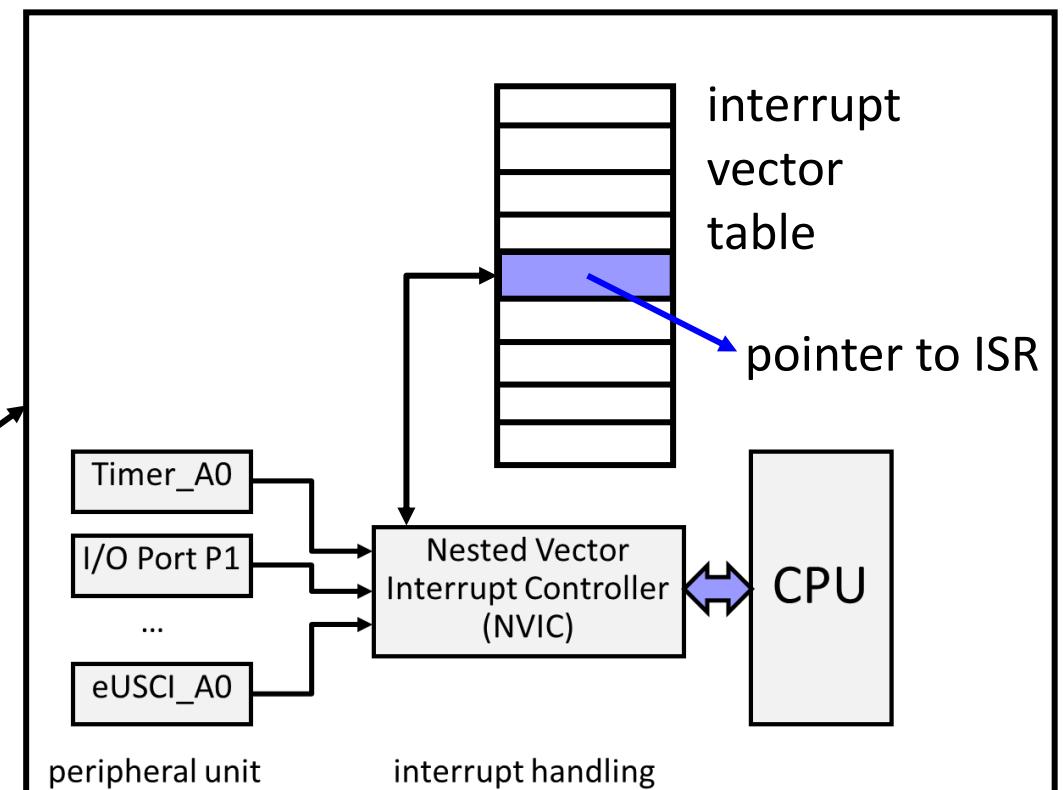
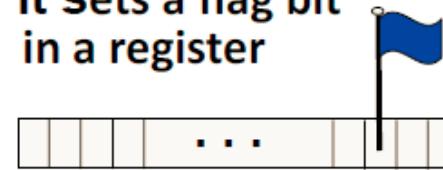


- UART
- GPIO
- Timers
- ADC
- Etc.

3. CPU/NVIC acknowledges interrupt by:

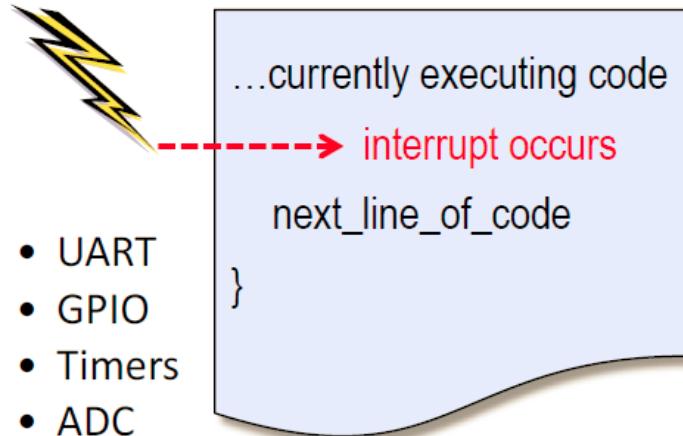
- current instruction completes
- saves return-to location on stack
- mask interrupts globally
- determines source of interrupt
- calls interrupt service routine (ISR)

2. It sets a flag bit in a register

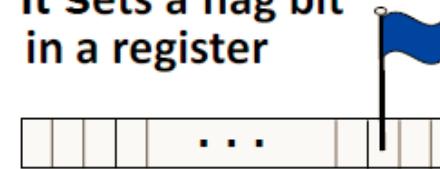


Processing of an Interrupt

1. An interrupt occurs

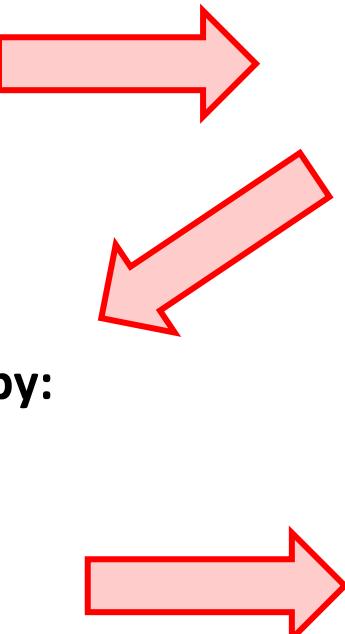


2. It sets a flag bit in a register



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)

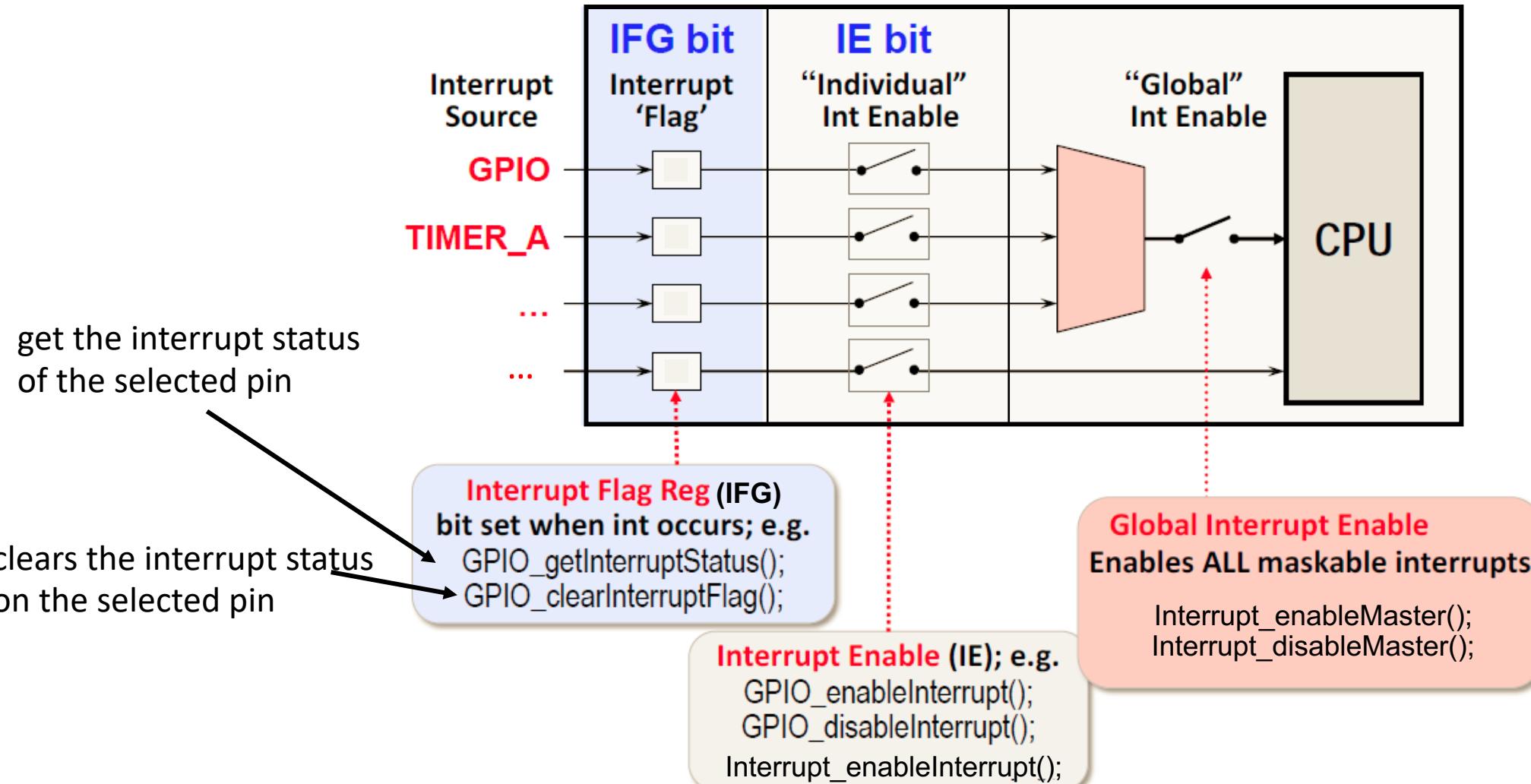


4. Interrupt Service Routine (ISR):

- save context of system (on stack of the ISR)
- **run your interrupt's code**
- restore context of system (from stack of the ISR)
- (automatically) un-mask interrupts and
- continue where it left off (return-to location)

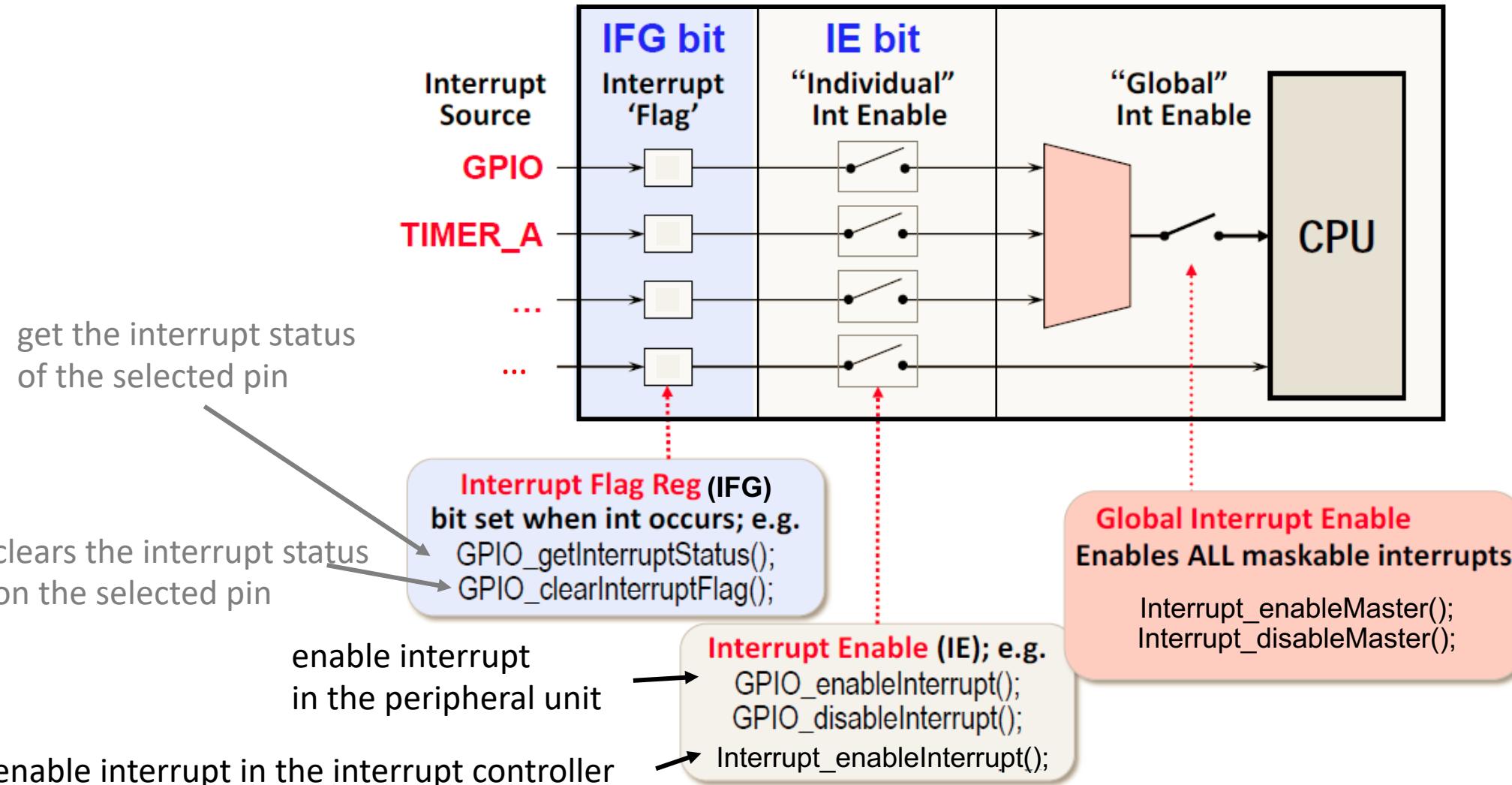
Processing of an Interrupt

Detailed interrupt processing flow:



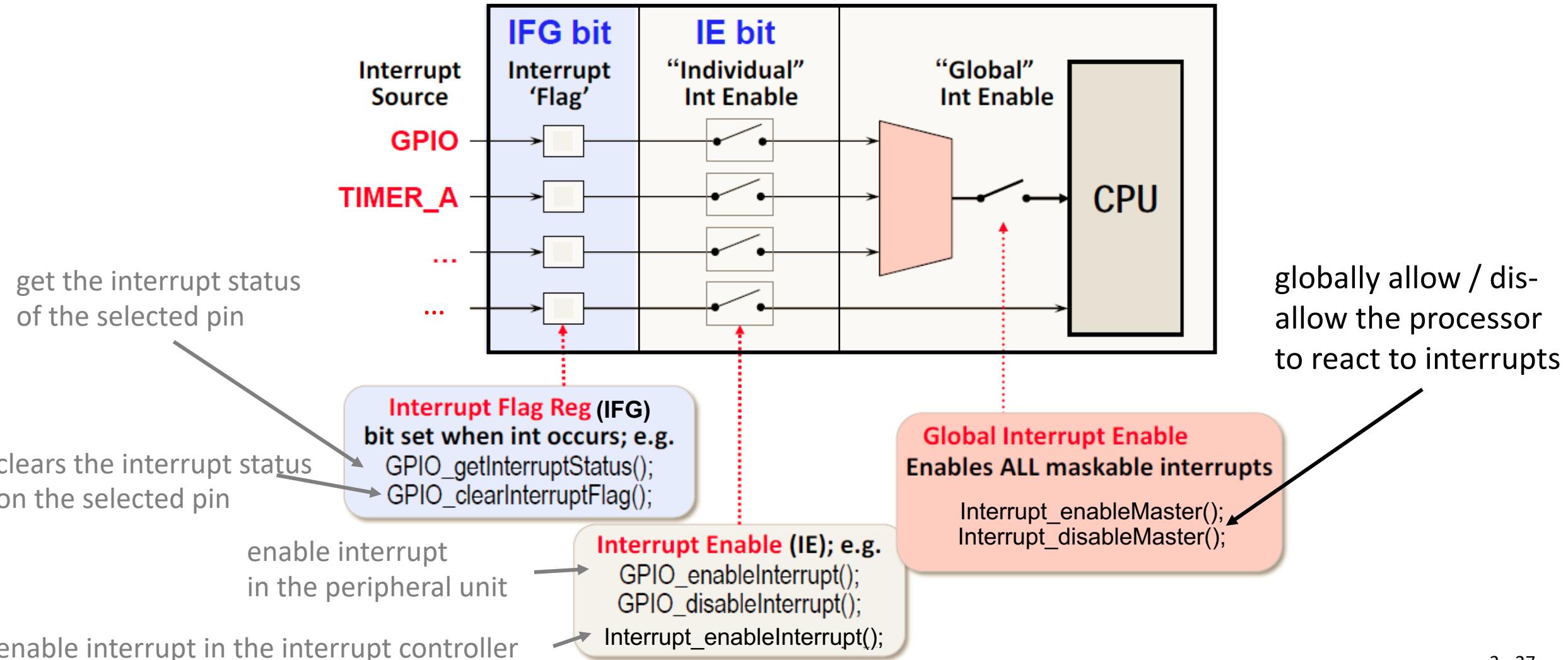
Processing of an Interrupt

Detailed interrupt processing flow:



Processing of an Interrupt

Detailed interrupt processing flow:



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*.

configure input
and output pins

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:*

continuously get the
signal at pin 1 and
detect falling edge

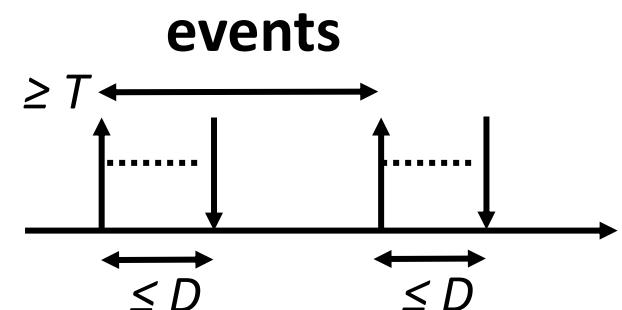
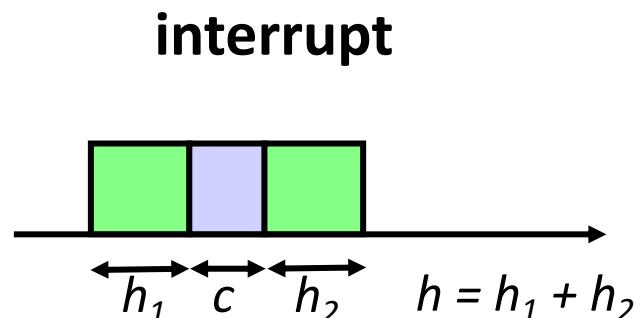
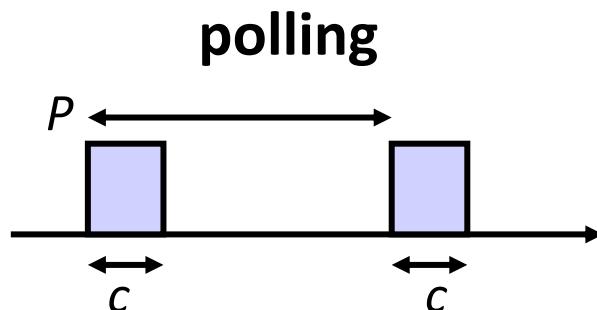
```
int main(void)
{
    uint8_t new, old;
    ...
    GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);
    GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);
    old = GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN1);

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

Polling vs. Interrupt

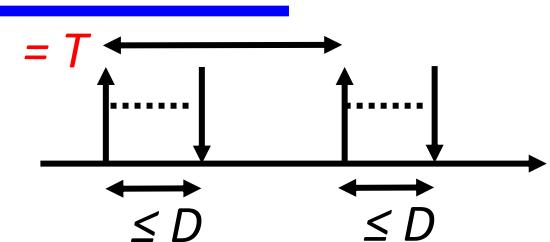
What are advantages and disadvantages?

- 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 for handling the event
 - *overhead h*: time overhead for handling the interrupt
 - *period P*: polling period
 - *inter-arrival time T*: minimal time between two events
 - *deadline D*: maximal time between event arrival and finishing event processing with $D \leq T$.



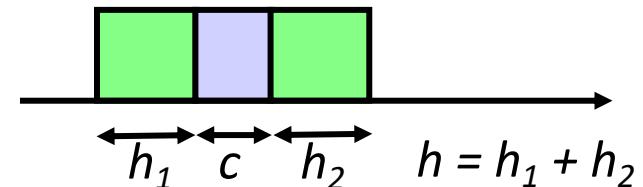
Polling vs. Interrupts

In the following, we suppose that the inter-arrival time between events is exactly T . This makes the results a bit easier to understand.



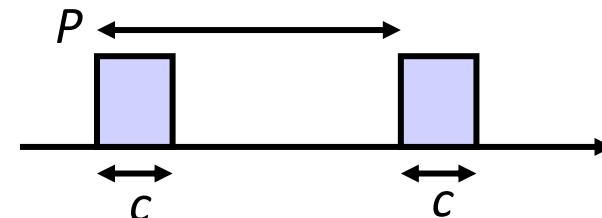
Some relations for *interrupt-based* event processing :

- The average utilization is $u_i = (h + c) / T$.
- As we need $h+c$ 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 most $P+c$ 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 polling period $P = D-c$ leads to the optimal (i.e., minimal) 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 inter-arrival time T , if the overhead h gets smaller in comparison to the computation time c , or if the inter-arrival 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 oscillator

As an example, the MSP432 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

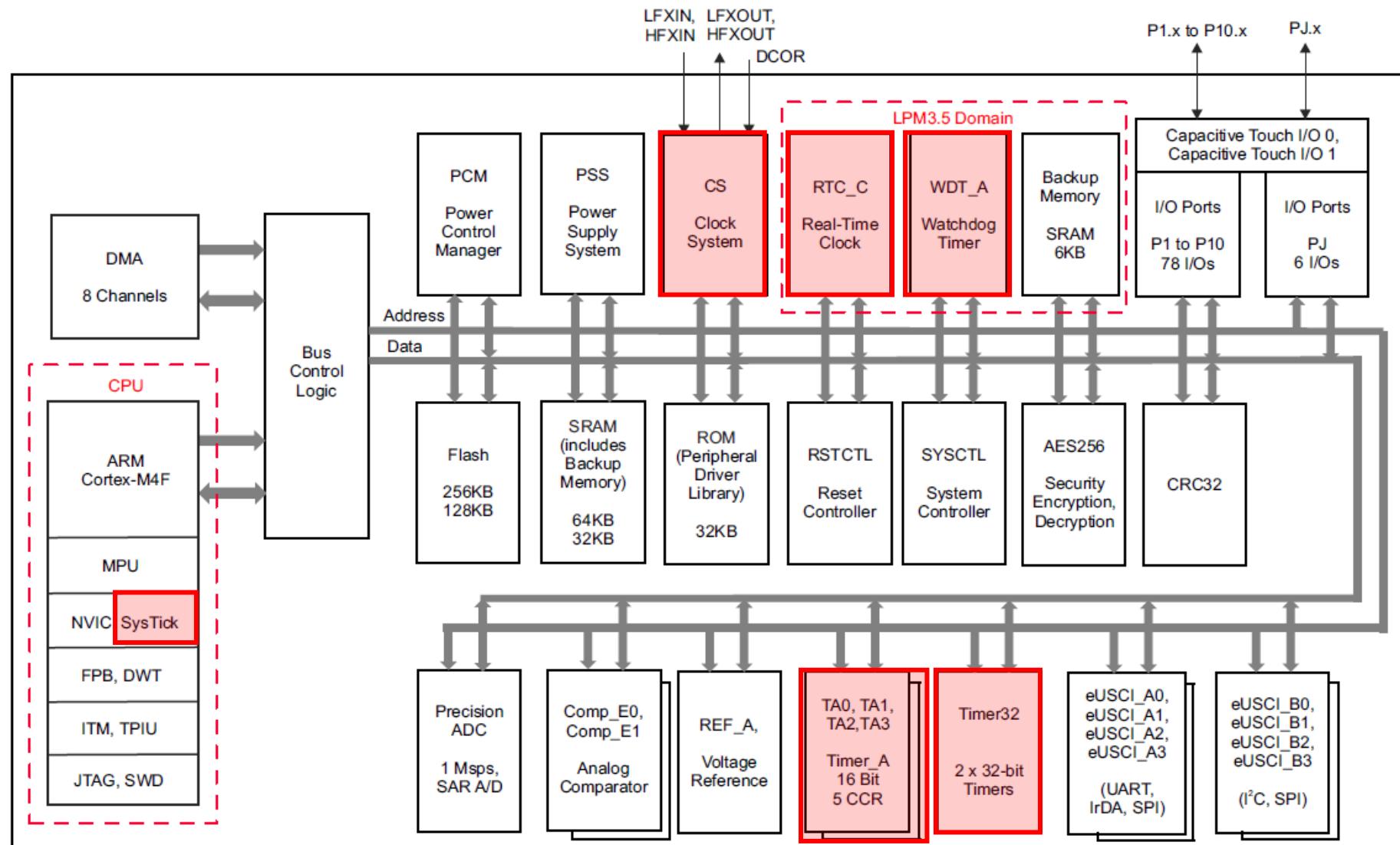
}

crystal-controlled
(more stable)

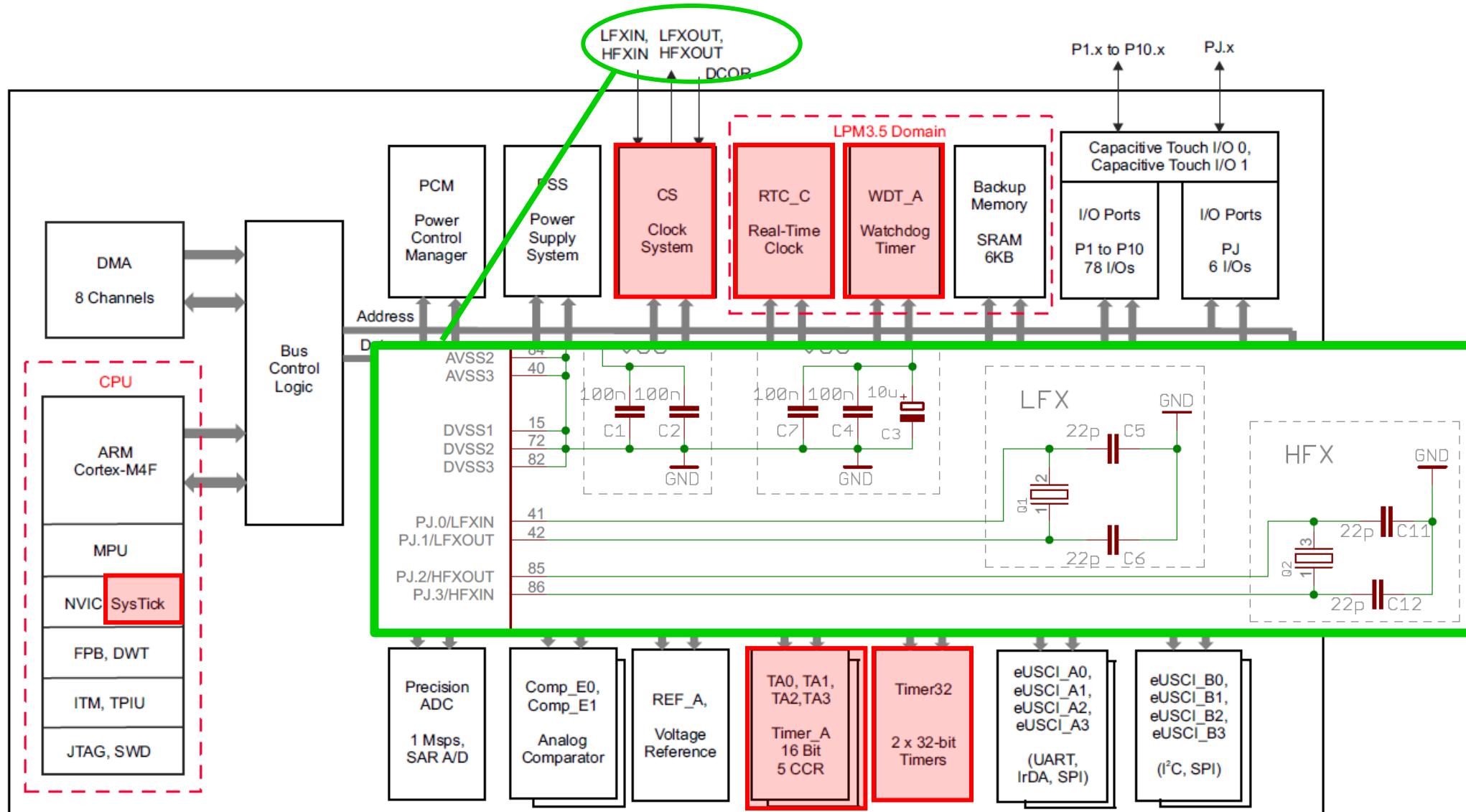
}

digitally-controlled
(less stable)

Clocks and Timers MSP432



Clocks and Timers MSP432



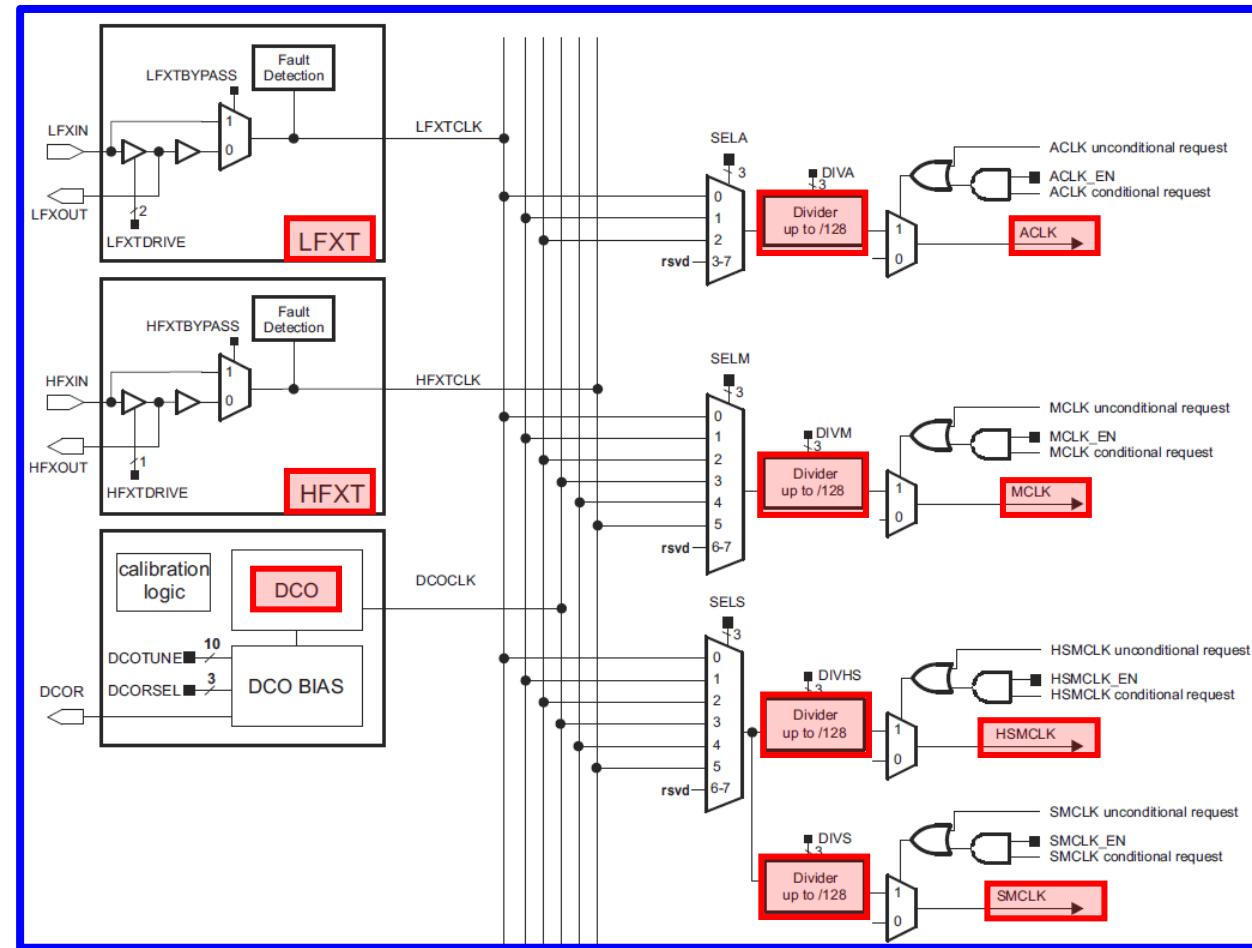
Clocks

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:

- 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



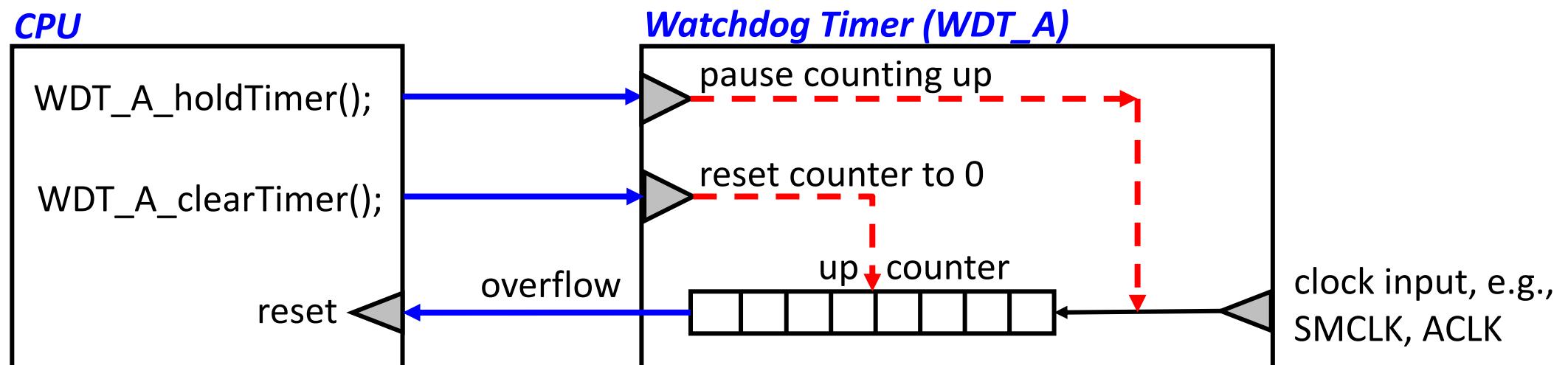
Clocks and Timers

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 an overflow occurs because the counter was not reset to 0,
then the CPU is reset.

Clocks and Timers

System Tick

SysTick MSP432

- **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(); ← go to low power mode LPM0 after executing the ISR  
}  
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.

SysTick MSP432

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 $2^{24} / (3 \times 10^6) = 5.59\dots$

} the resolution of the duration is one
microsecond; the duration must not be
longer than ~5.6 seconds; note the use of
modular arithmetic if end > start;
overhead for calling SysTick_getValue()
is not accounted for;

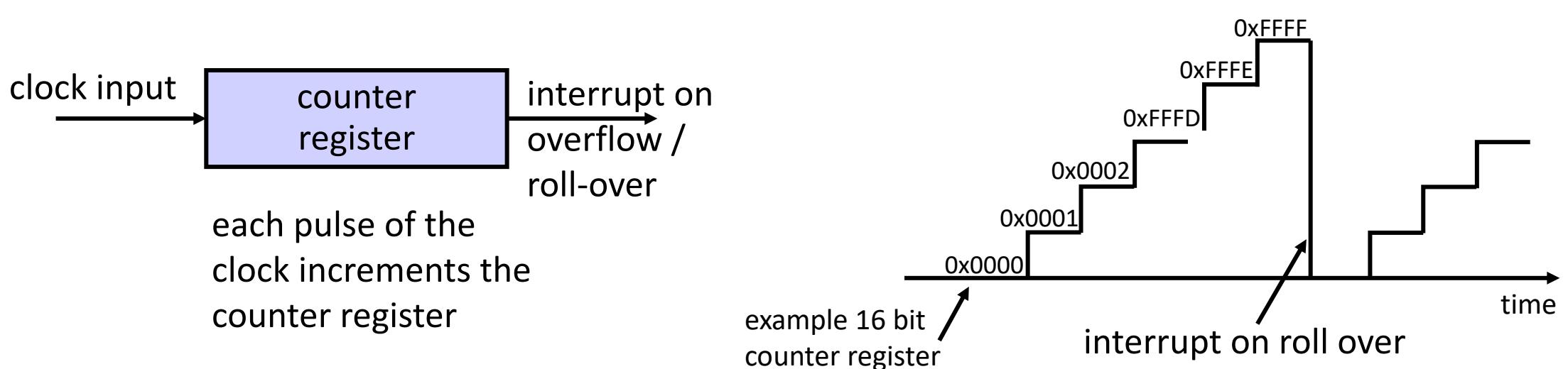
Clocks and Timers

Timer and PWM

Timer

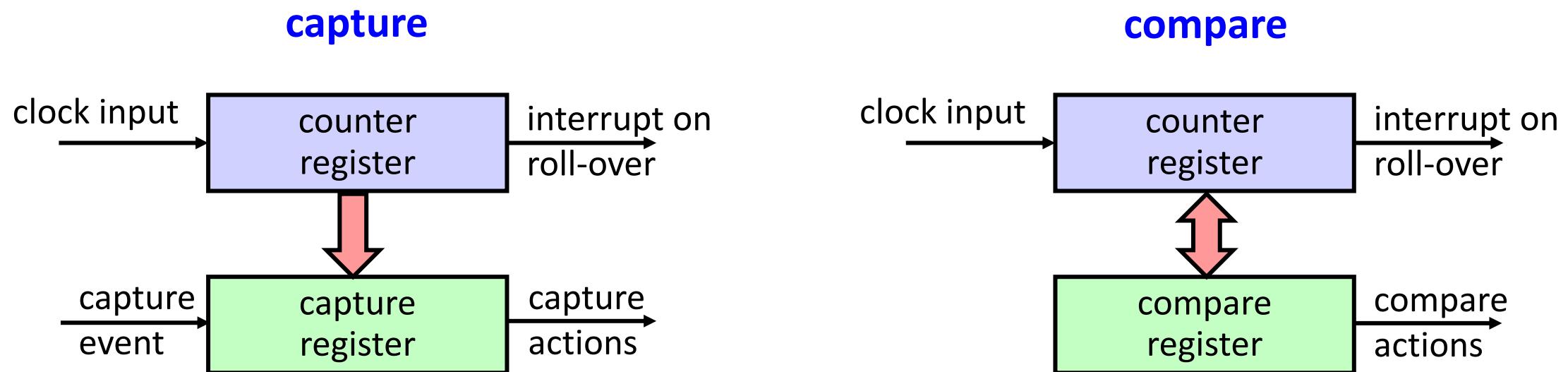
Usually, *embedded microprocessors* have *several* elaborate *timers* that allow to

- *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*:

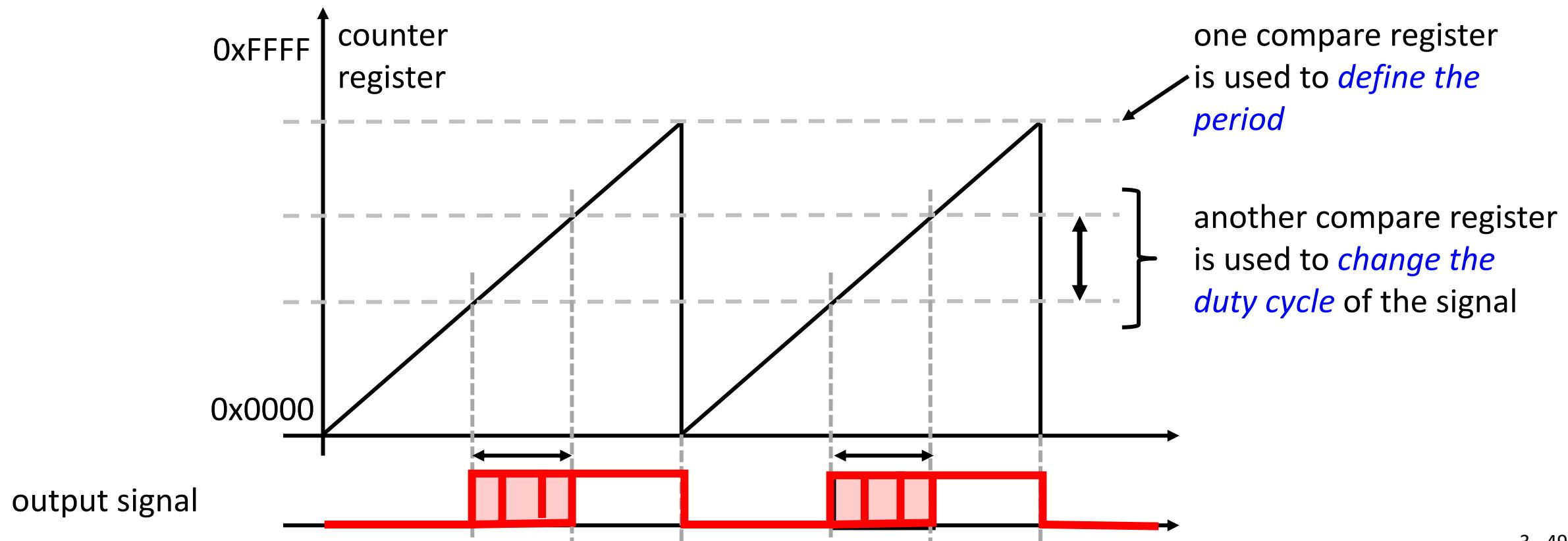


- the value of *counter register* is stored in *capture register* at the time of the *capture event* (input signals, software)
- the value can be read by software
- at the time of the capture, further actions can be triggered (interrupt, signal)

- 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

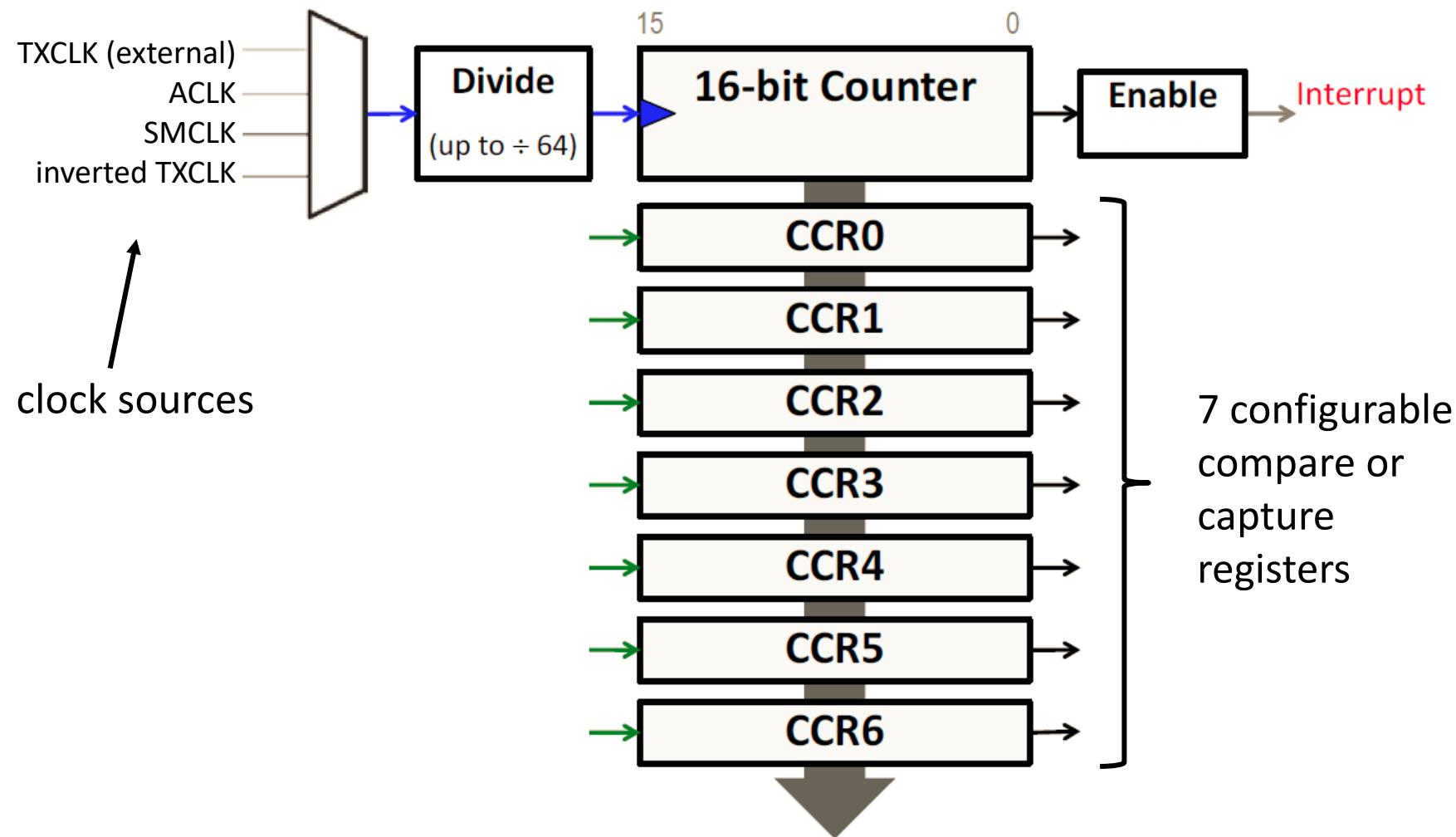
Timer

- *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.



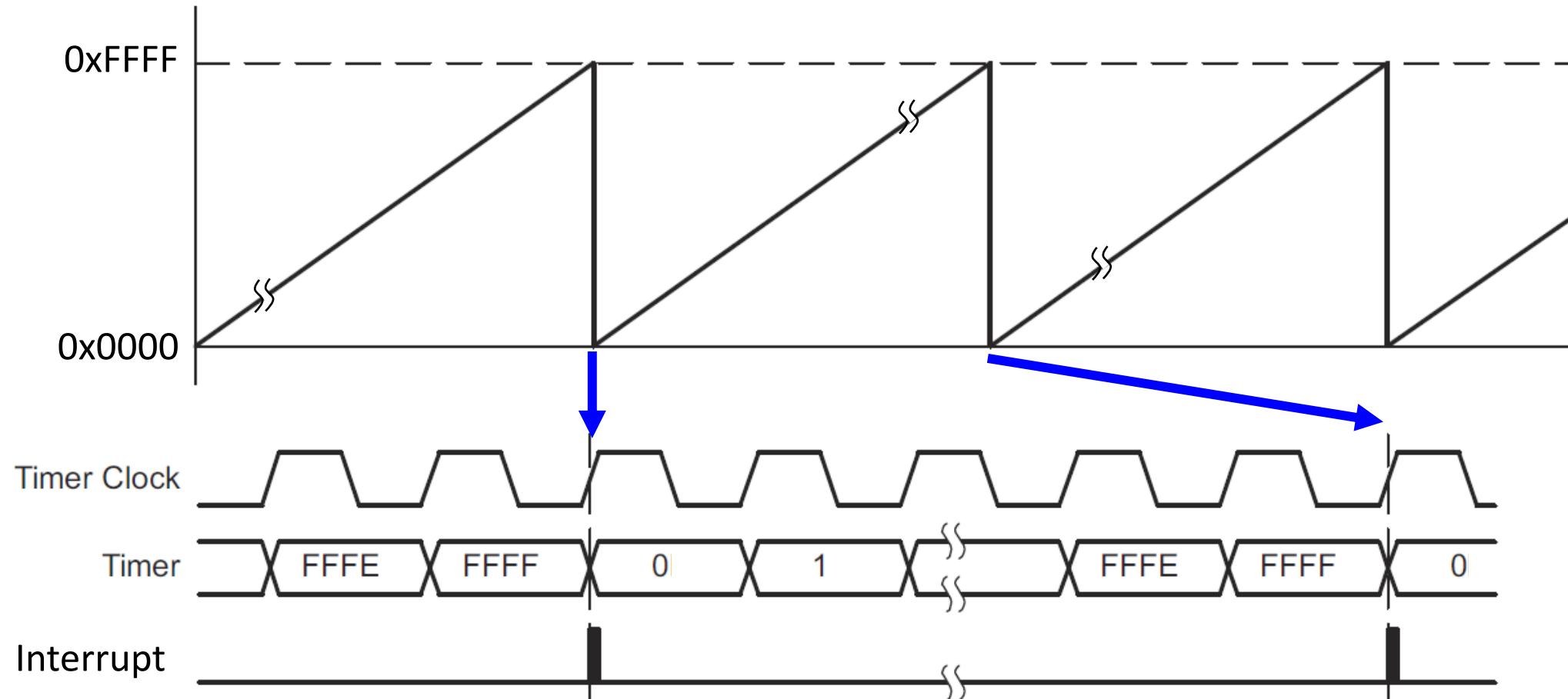
Timer Example MSP432

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



Timer Example MSP432

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



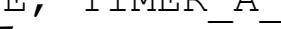
Timer Example MSP432

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

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_CaptureCompareModeConfig 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

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 CCRO ... 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;  
    }  
}
```

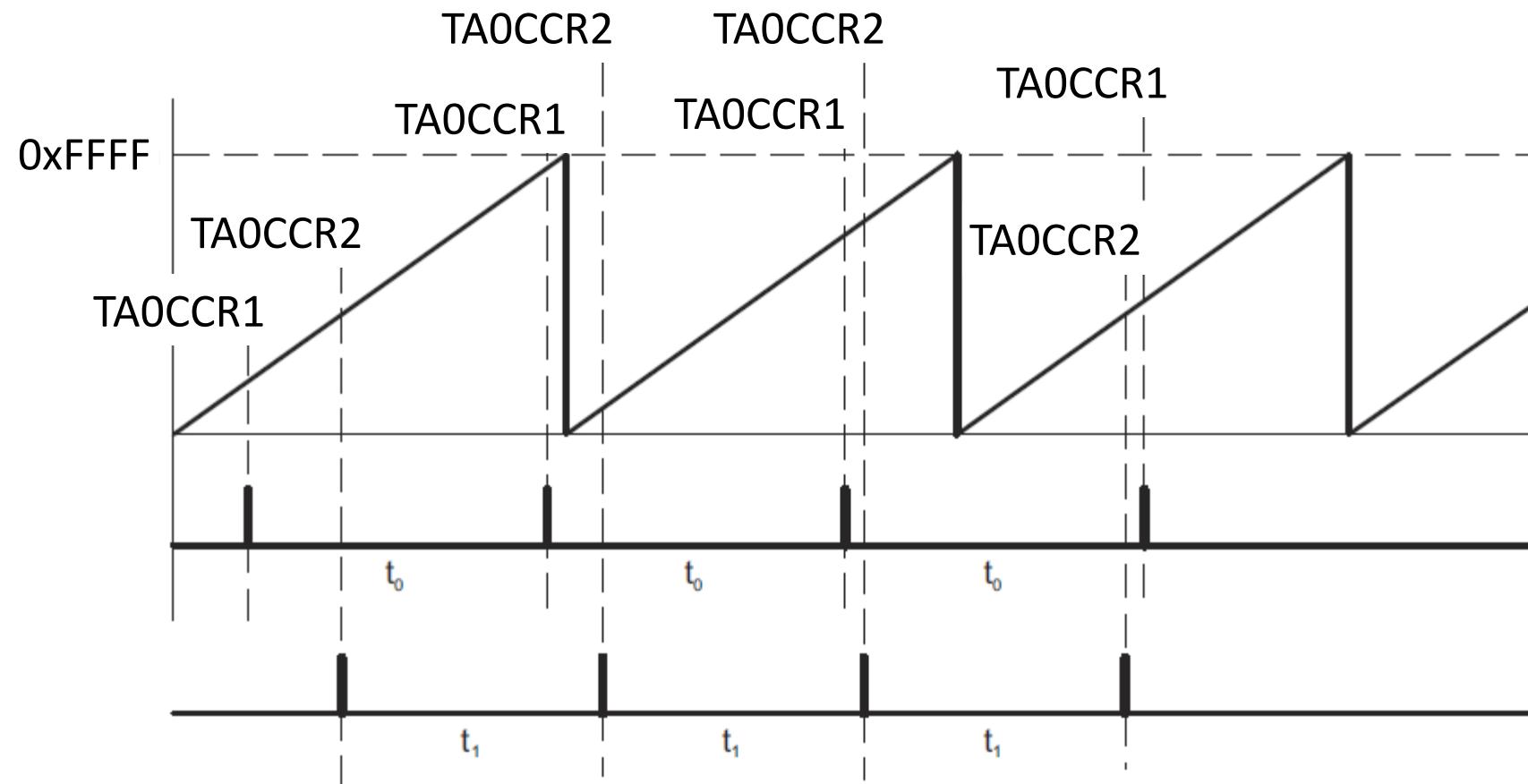
the register TA0IV contains the *interrupt flags* for the registers; after being read, the *highest priority interrupt* (smallest register number) is *cleared automatically*.

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

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



Introduction to Embedded Systems

4. Programming Paradigms

Prof. Dr. Marco Zimmerling



Organization

Join ILIAS course:

- Login: RZ username + password
- Course password: **es-0x8af**

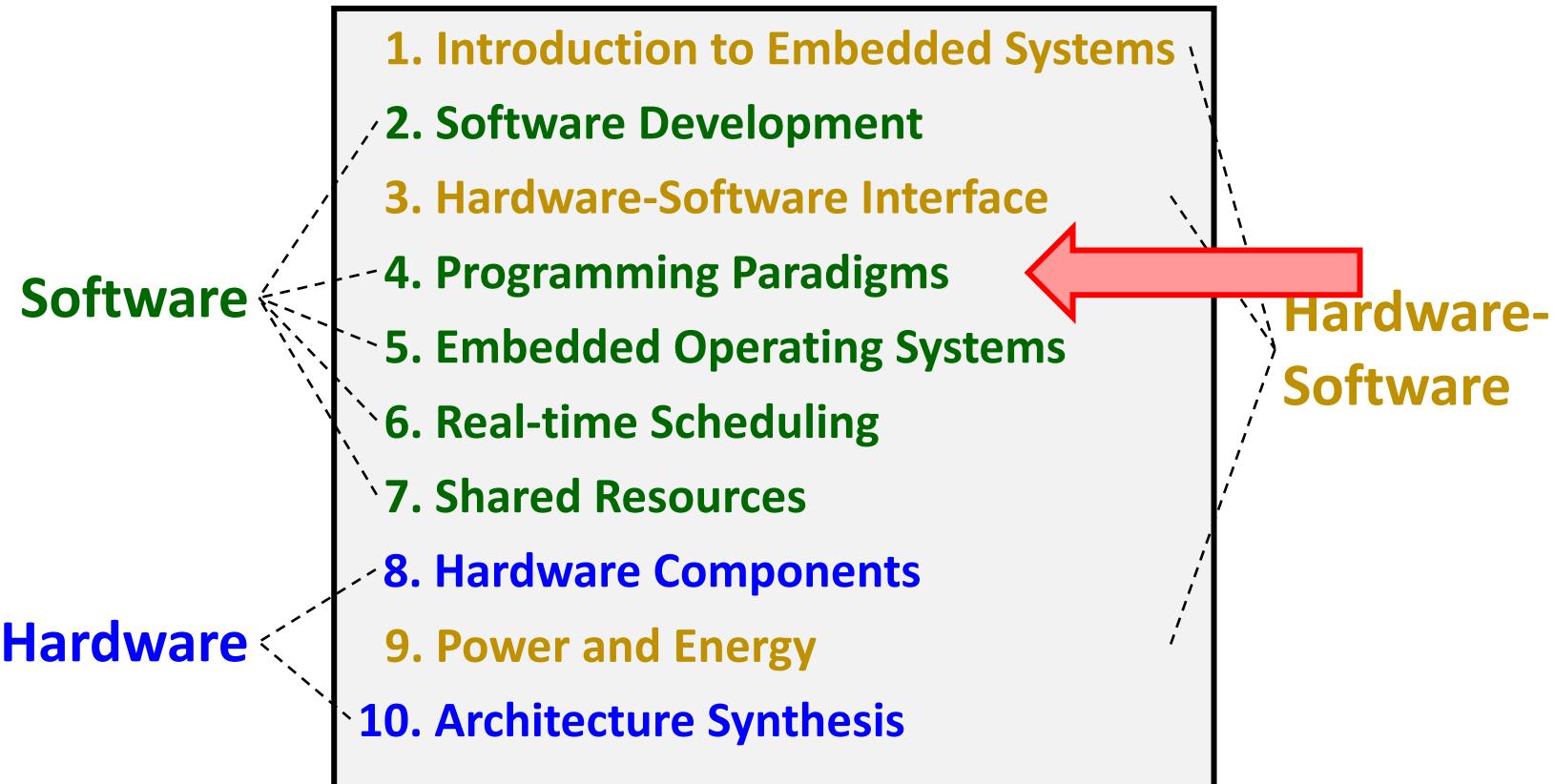
Exercises:

- From 12:00 to 14:00 in HS 00-038 (German) and Kinohörsaal (English)
- Today (November 15):
 - [Solutions](#) of first exercise sheet presented
 - Second exercise sheet [released](#) + [overview](#) of tasks given
- Next week (November 22):
 - [Change in rooms for exercises](#) will be announced during lecture and on ILIAS

Exam: [March 4, 2023](#) from 10:00 to 12:00, room TBD



Where we are ...

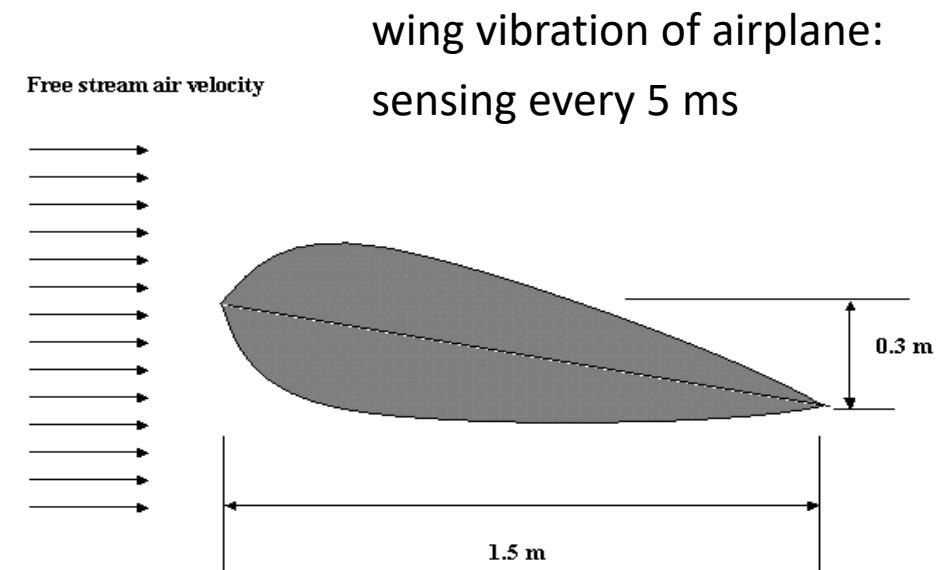


Reactive Systems and Timing

Timing Guarantees

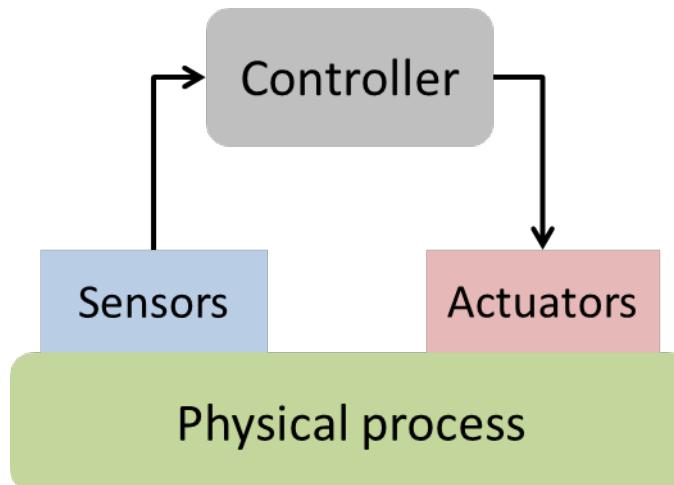
- *Hard real-time systems* can be often found in *safety-critical applications*. They need to provide the result of a computation within a fixed time bound.
- *Typical application domains:*
 - avionics, automotive, train systems, automatic control including robotics, manufacturing, media content production

sideairbag in car:
reaction after event in <10 ms

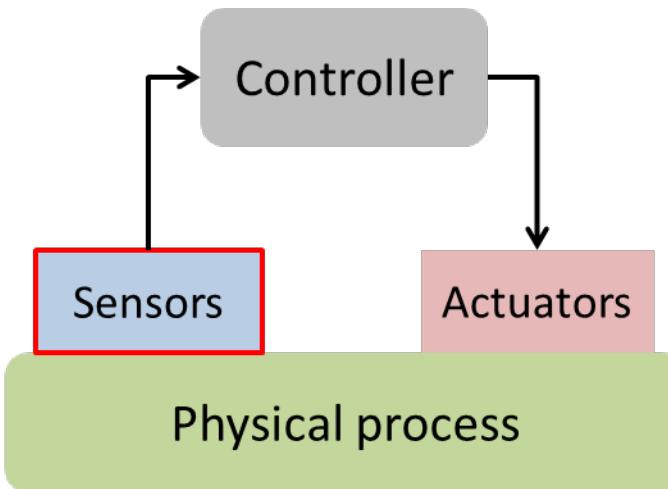


Real-Time Systems

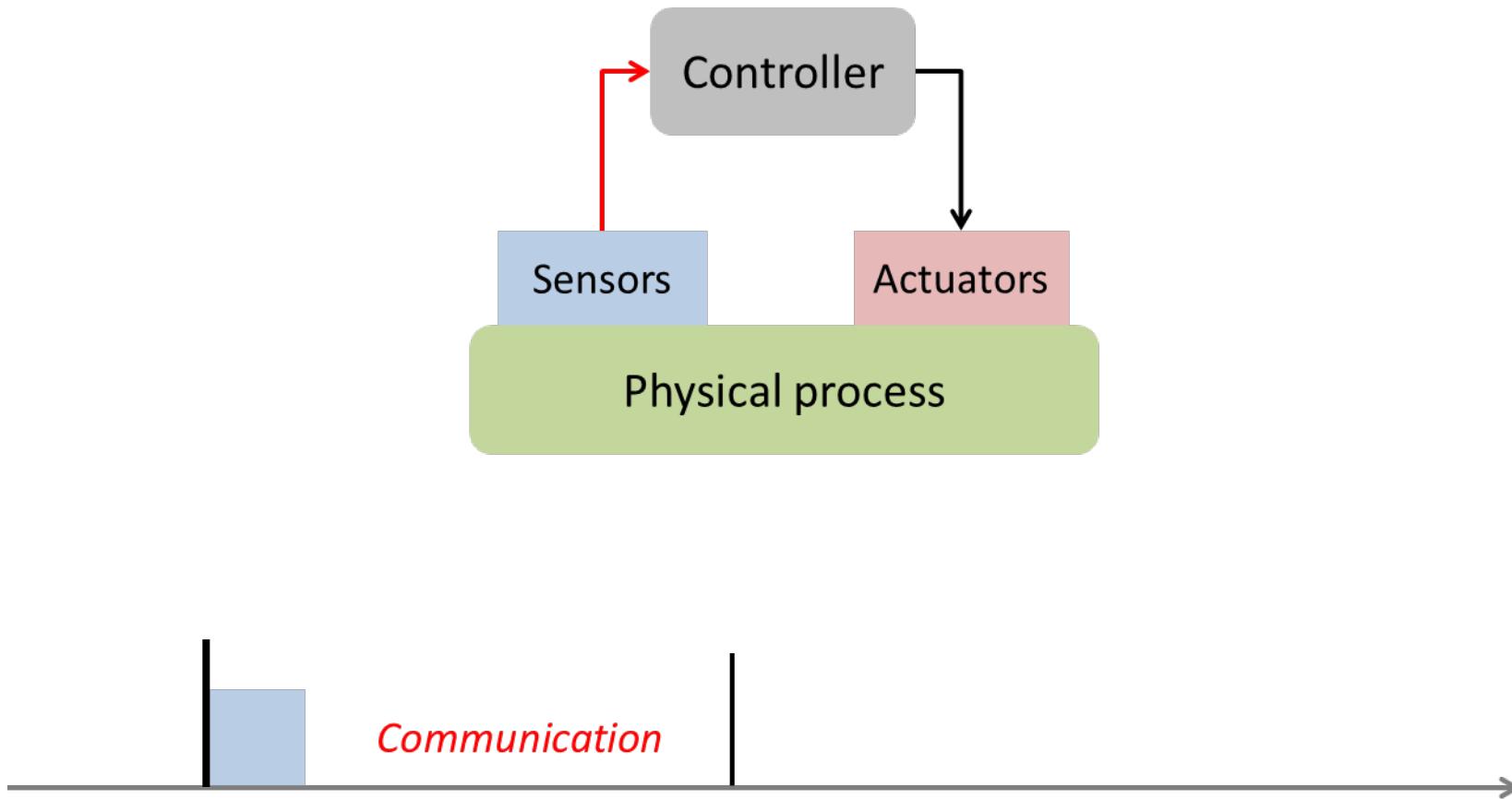
In many *cyber-physical systems (CPSs)*, timing is a matter of *correctness*, not performance: *an answer arriving too late is considered to be an error*.



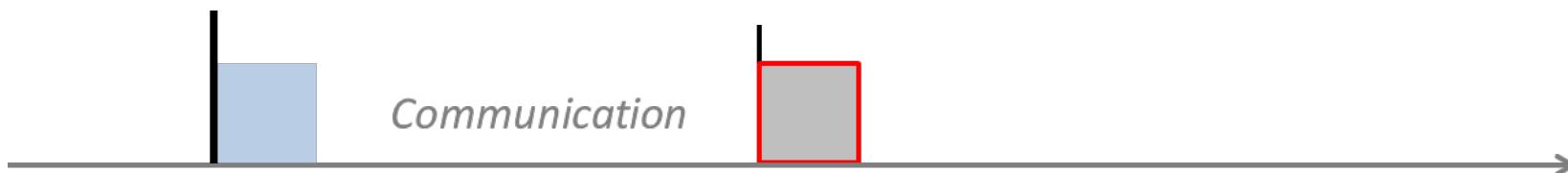
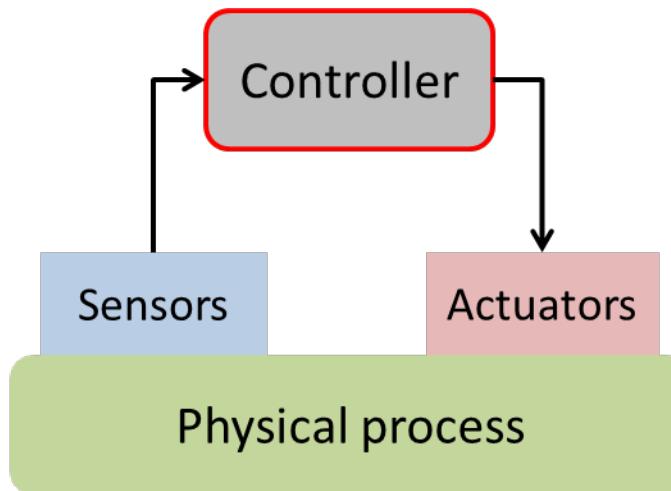
Real-Time Systems



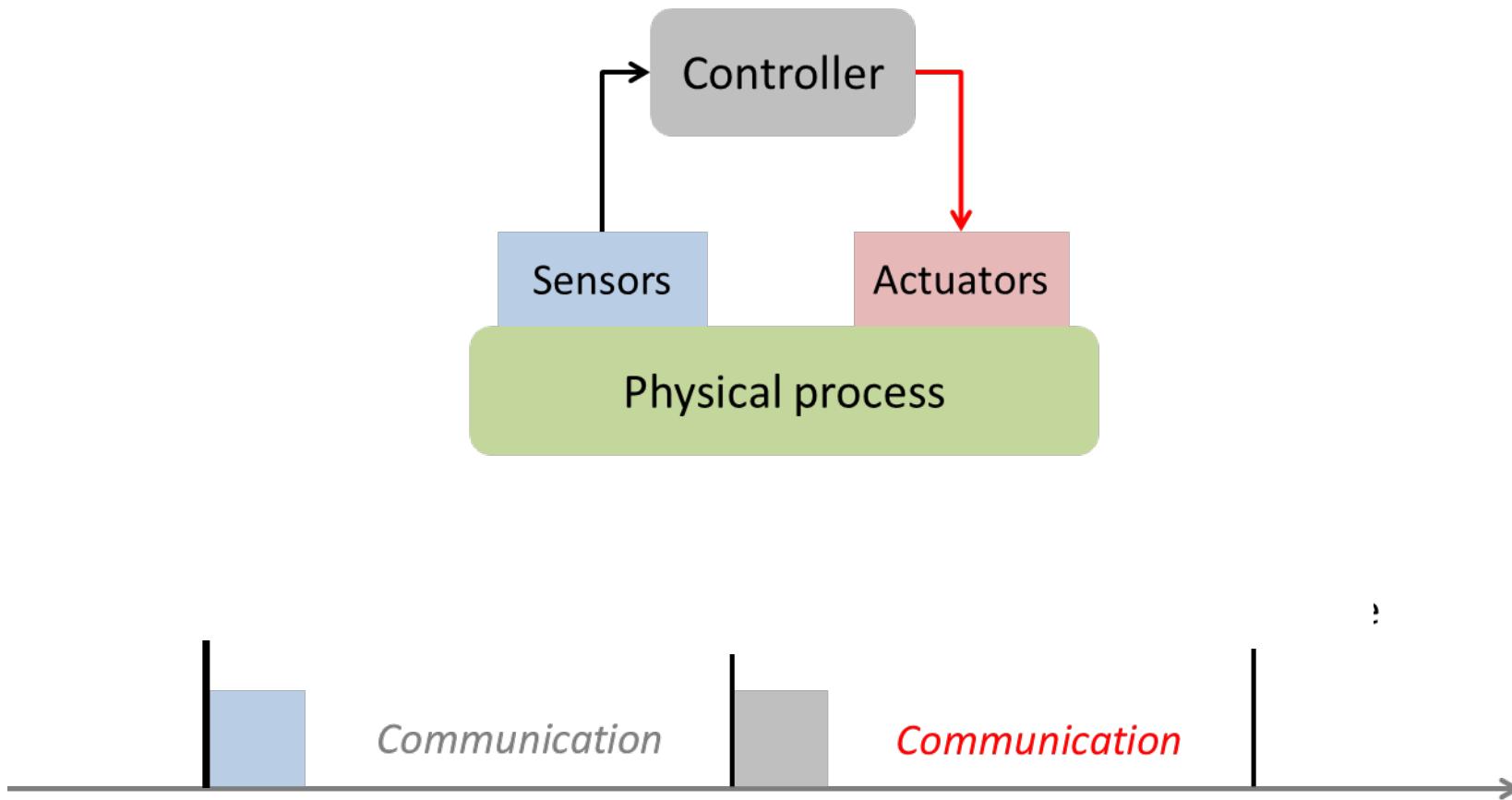
Real-Time Systems



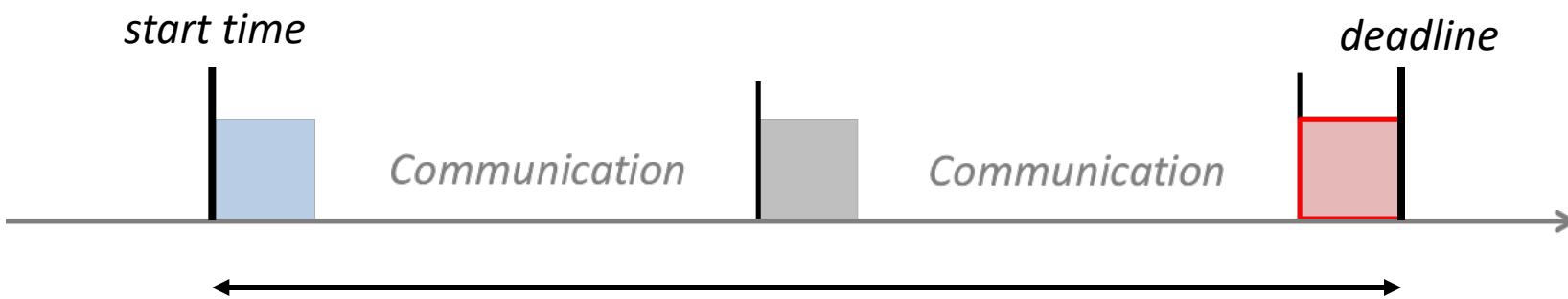
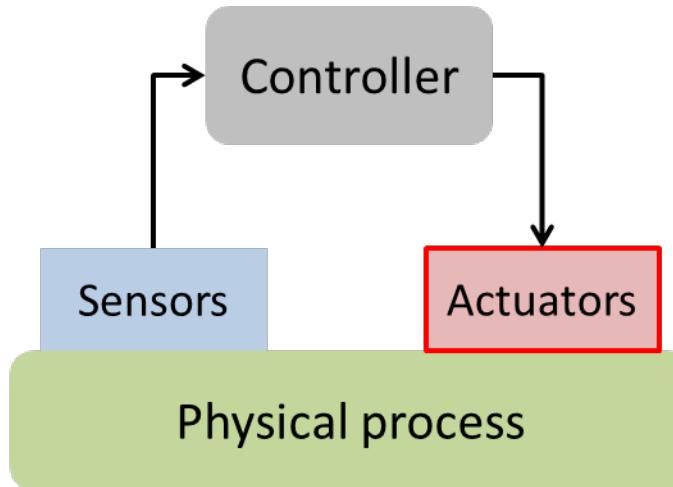
Real-Time Systems



Real-Time Systems



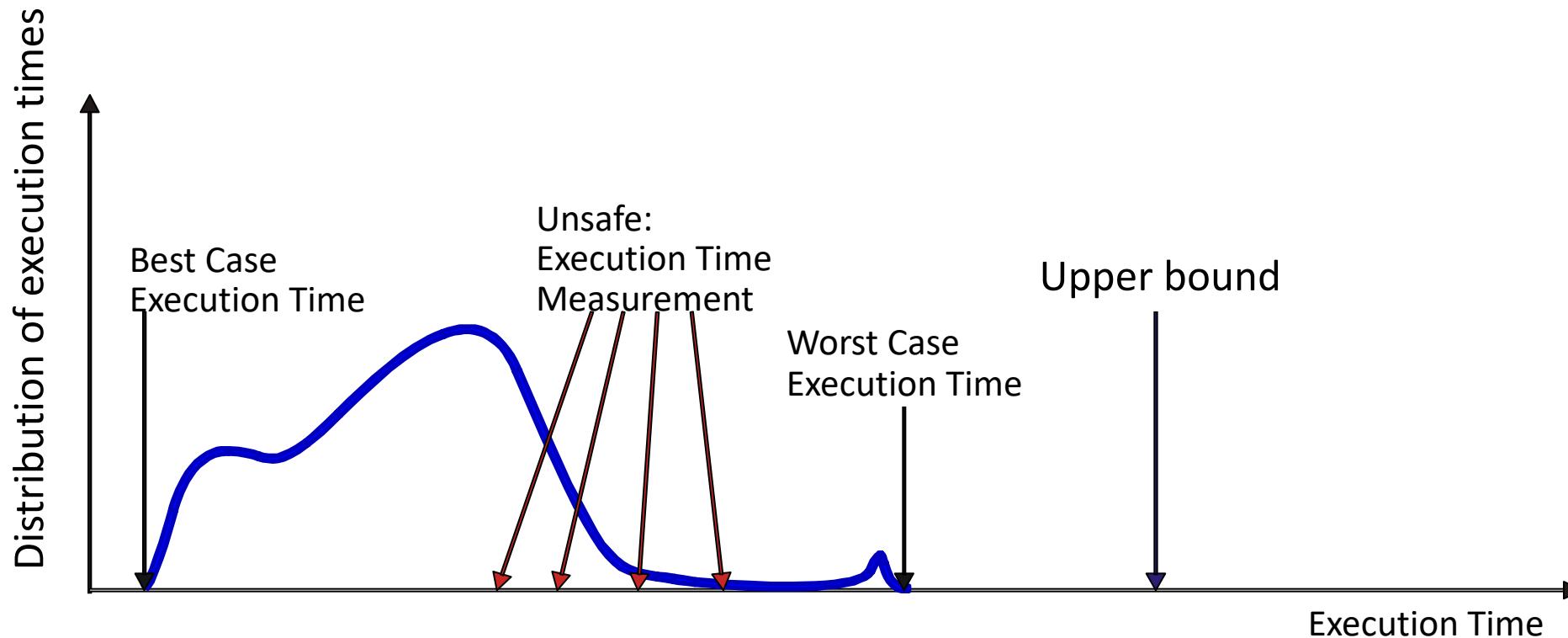
Real-Time Systems



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.
- The upper bound is known as the *Worst-Case Execution Time* (WCET).
- The lower bound is known as the *Best-Case Execution Time* (BCET).

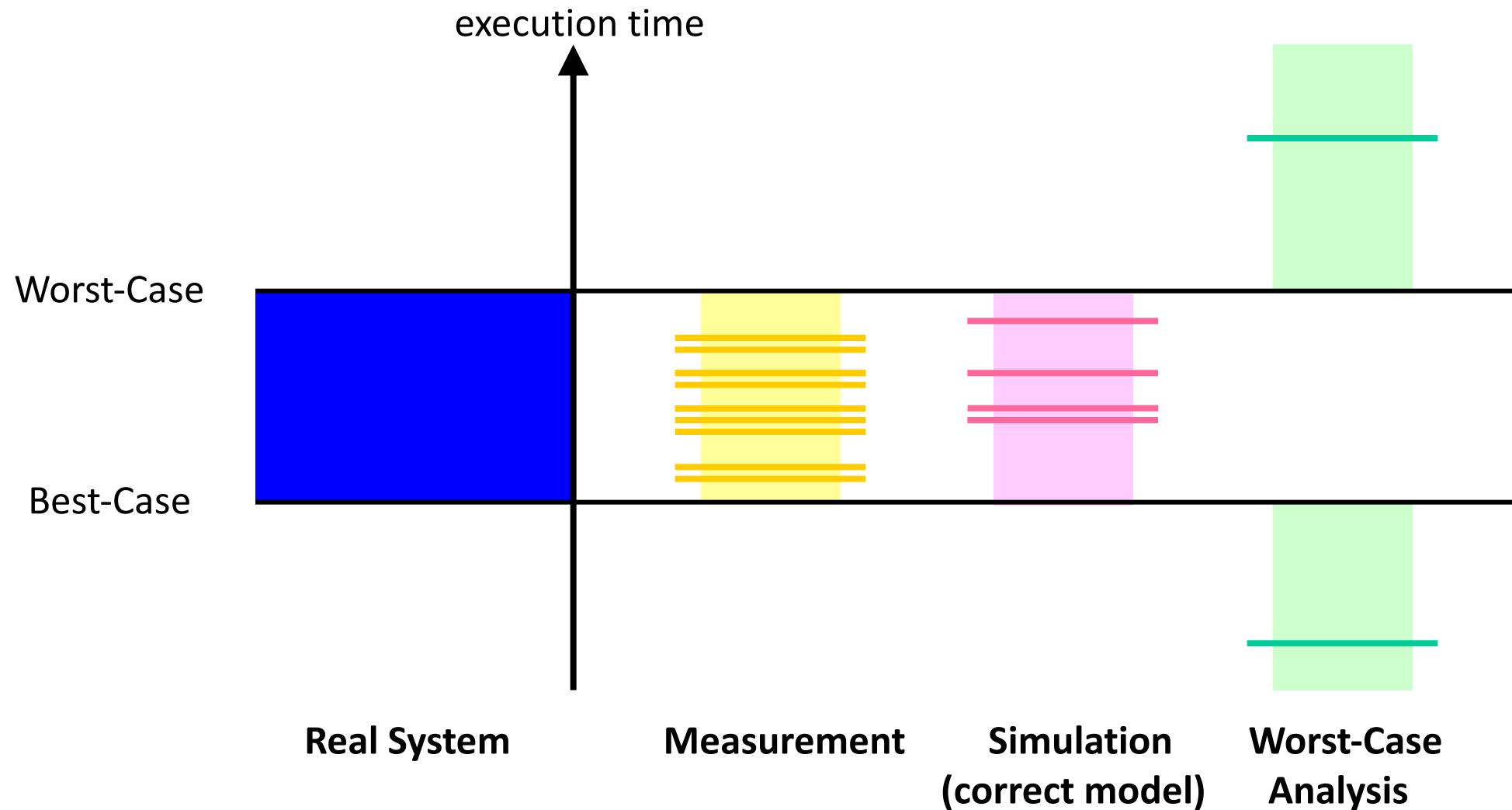
Distribution of Execution Times



Modern Hardware Features

- Modern processors *increase the average performance* (execution of tasks) using, for example, *caches, pipelines, branch prediction*, and *speculation* techniques.
- *These features make the computation of the WCET very difficult*: The execution times of single instructions vary widely (high uncertainty).
- The microarchitecture has a large *time-varying internal state* that is changed by the execution of instructions and that influences the execution times of tasks.
 - *Best case* - everything goes smoothly: no cache misses, operands are ready, required resources are free, branch correctly predicted.
 - *Worst case* - everything goes wrong: all loads miss the cache, operands are not ready, required resources are occupied, branch falsely predicted.
 - *The span between the best case and worst case may be several hundred cycles.*

Methods to Determine the Execution Time of a Task



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, virtual memory, and so on.

Analytical (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.
- *For the rest of the lecture, we assume that reliable bounds on the WCET are available*, for example, by means of formal analysis or exhaustive measurements/simulations.

Different Programming Paradigms

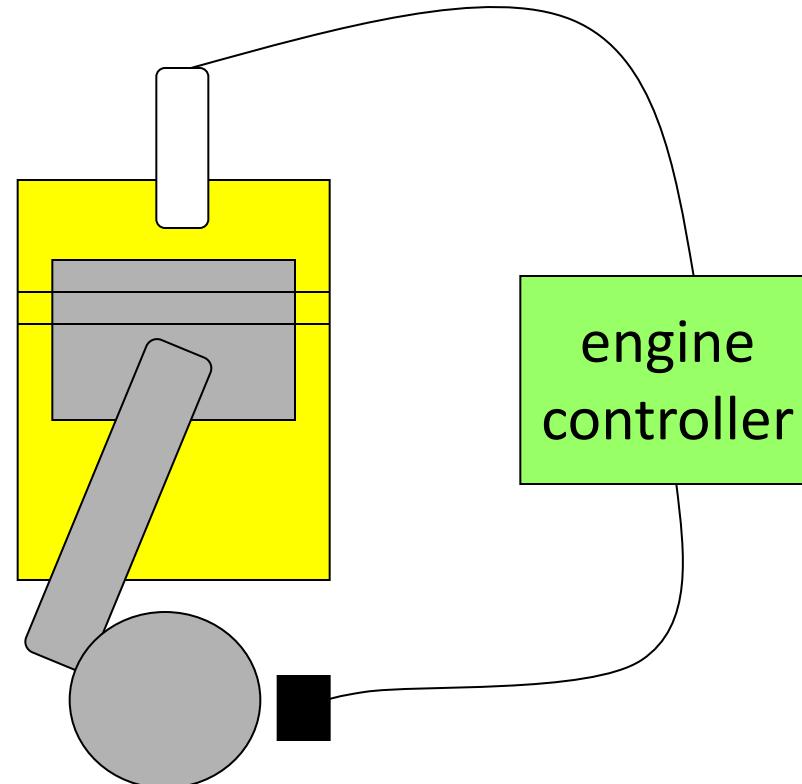
Why Multiple Tasks on one Embedded Device?

- 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



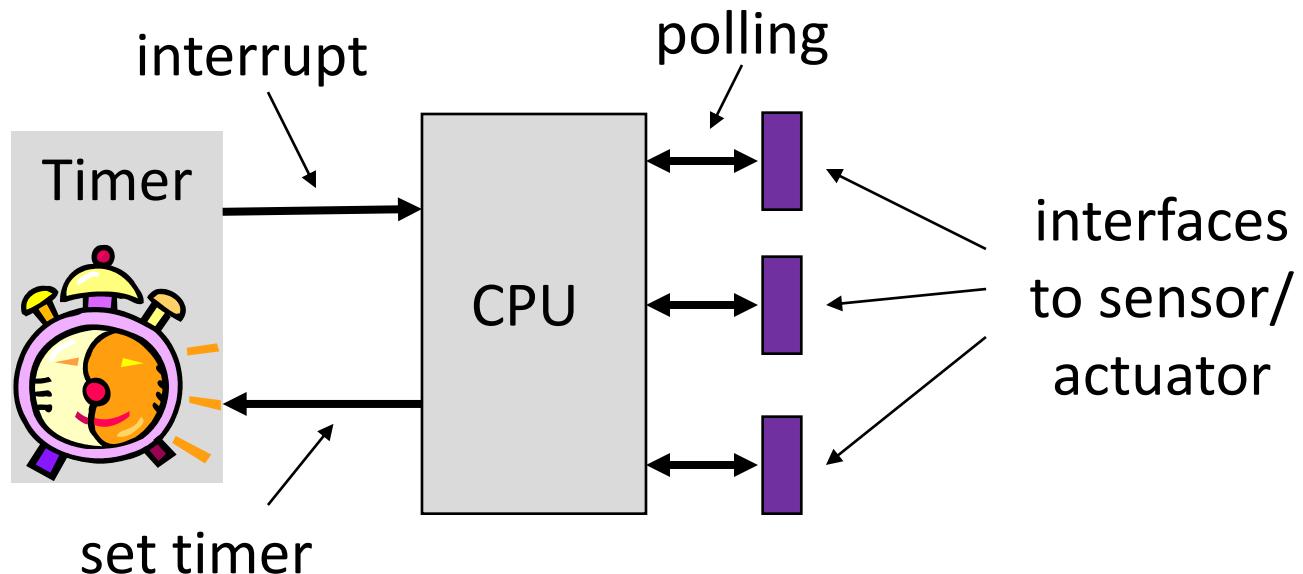
Overview

- 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

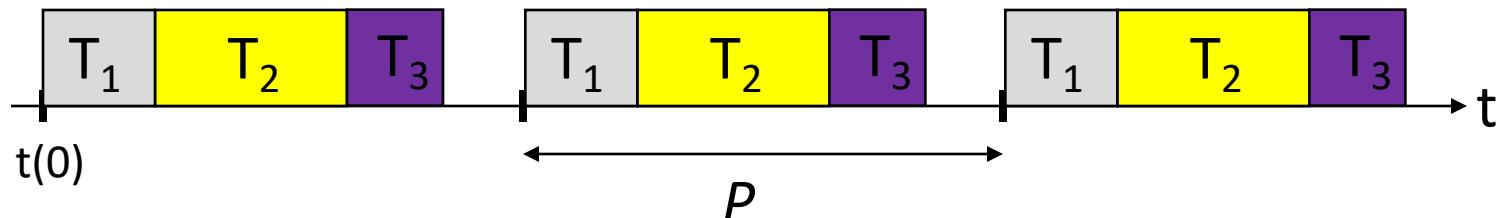
Pure time-triggered model:

- *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();
```

usually done offline

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;
```

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

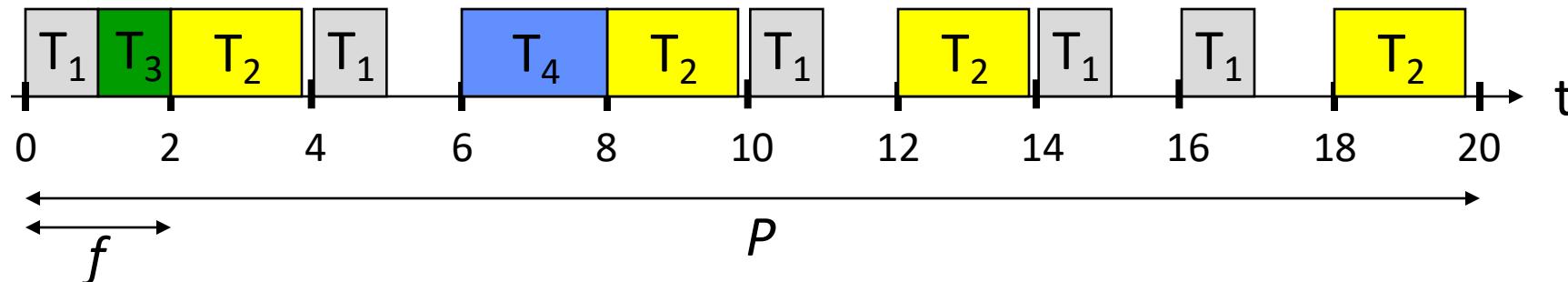
for example using a function pointer in C;
task(= function) returns after finishing.

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 a 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:*

Γ : denotes the set of all periodic tasks

τ_i : denotes a periodic task

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

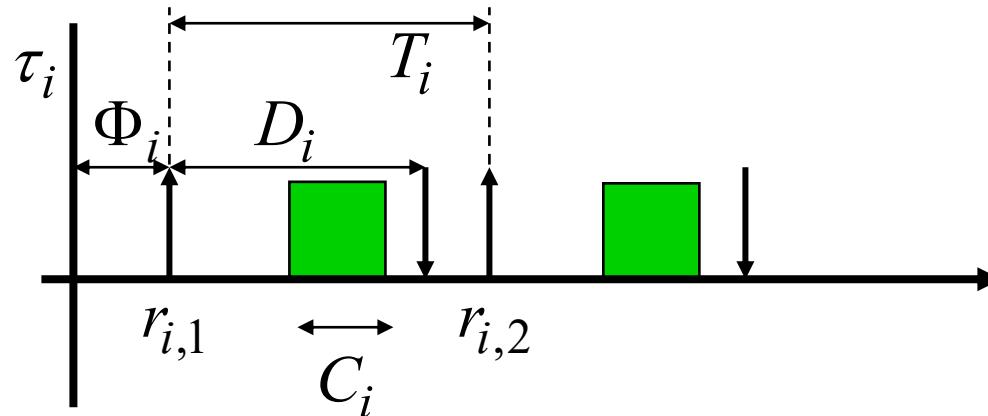
$r_{i,j}, d_{i,j}$: denote the release time and absolute deadline of the j th instance of task i

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

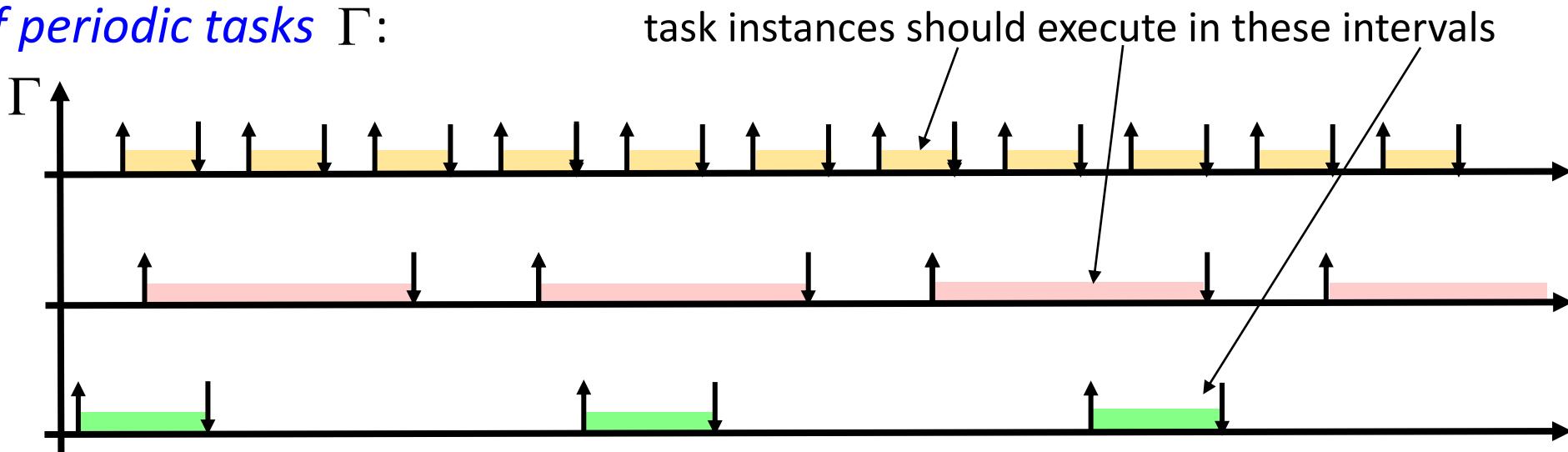
D_i : relative deadline of task i

Time-Triggered Cyclic Executive Scheduling

- *Example* of a single periodic task τ_i :



- *A set of periodic tasks* Γ :



Time-Triggered Cyclic Executive Scheduling

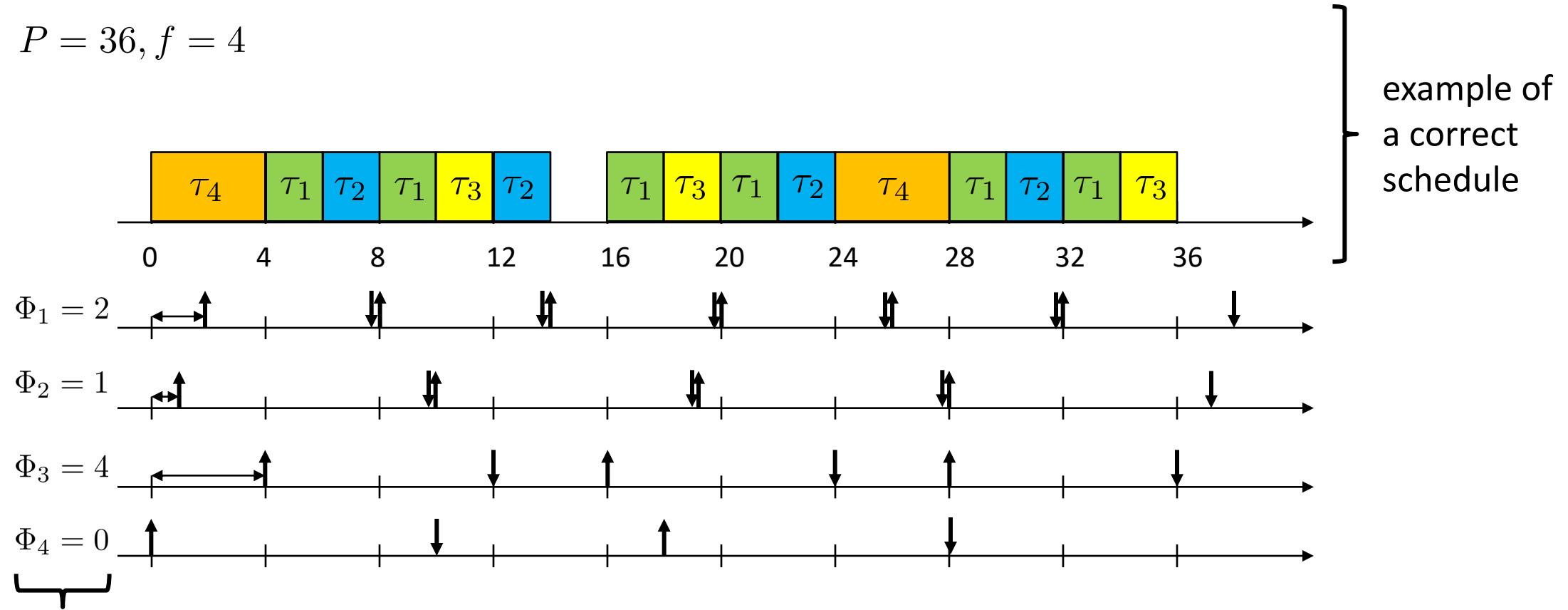
- The following *hypotheses* are assumed on the tasks:
 - *The instances of a periodic task are regularly activated at a constant rate.* The interval T_i between two consecutive activations is called *period*. Therefore, the release times satisfy
$$r_{i,j} = \Phi_i + (j-1)T_i$$
 - *All instances have 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:

- $\tau_1 : T_1 = 6, D_1 = 6, C_1 = 2$ $\tau_2 : T_2 = 9, D_2 = 9, C_2 = 2$
- $\tau_3 : T_3 = 12, D_3 = 8, C_3 = 2$ $\tau_4 : T_4 = 18, D_4 = 10, C_4 = 4$
- $P = 36, f = 4$



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:

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

- 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:

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

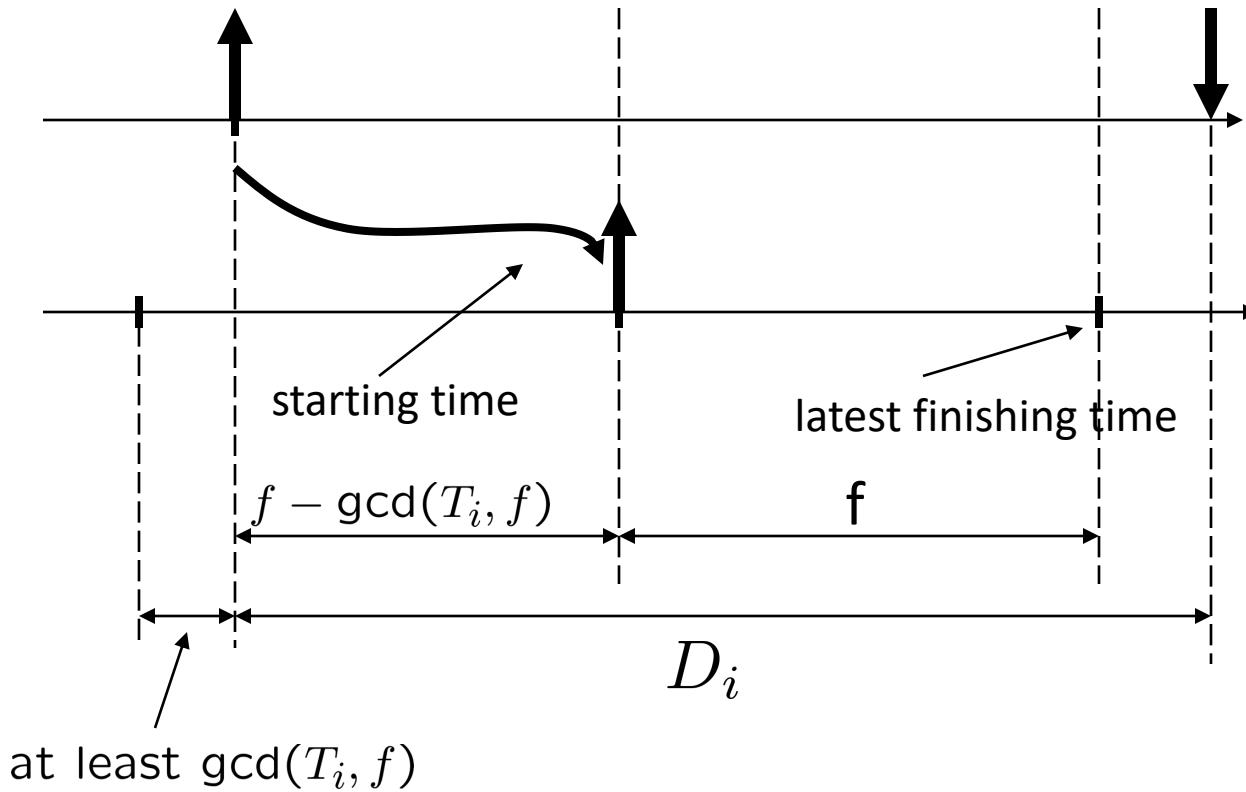
worst case execution time
of task

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

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

relative deadline of task

Sketch of Proof for Last Condition



release times and deadlines of tasks

frames

Example: Cyclic Executive Scheduling

Conditions:

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

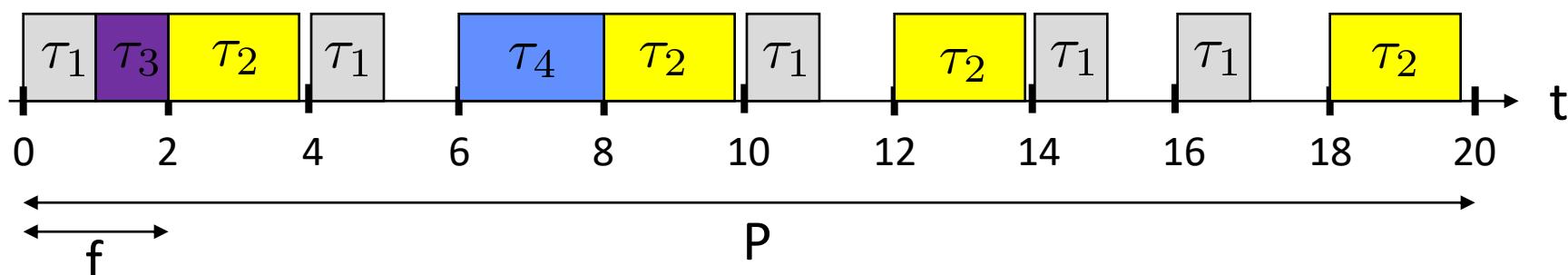
$$f \geq \max\{1.0, 1.0, 1.8, 2.0\} = 2.0$$

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

possible solution: $f = 2$

Γ	T_i	D_i	C_i
τ_1	4	4	1.0
τ_2	5	5	1.8
τ_3	20	20	1.0
τ_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 τ_i executes.
- Is P a common multiple of all periods T_i ?
- Is P a multiple of f ?
- Is the frame sufficiently long?

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

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

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

- Are deadlines respected?

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

Introduction to Embedded Systems

4. Programming Paradigms

Prof. Dr. Marco Zimmerling



Organization

Join ILIAS course:

- Login: RZ username + password
- Course password: **es-0x8af**

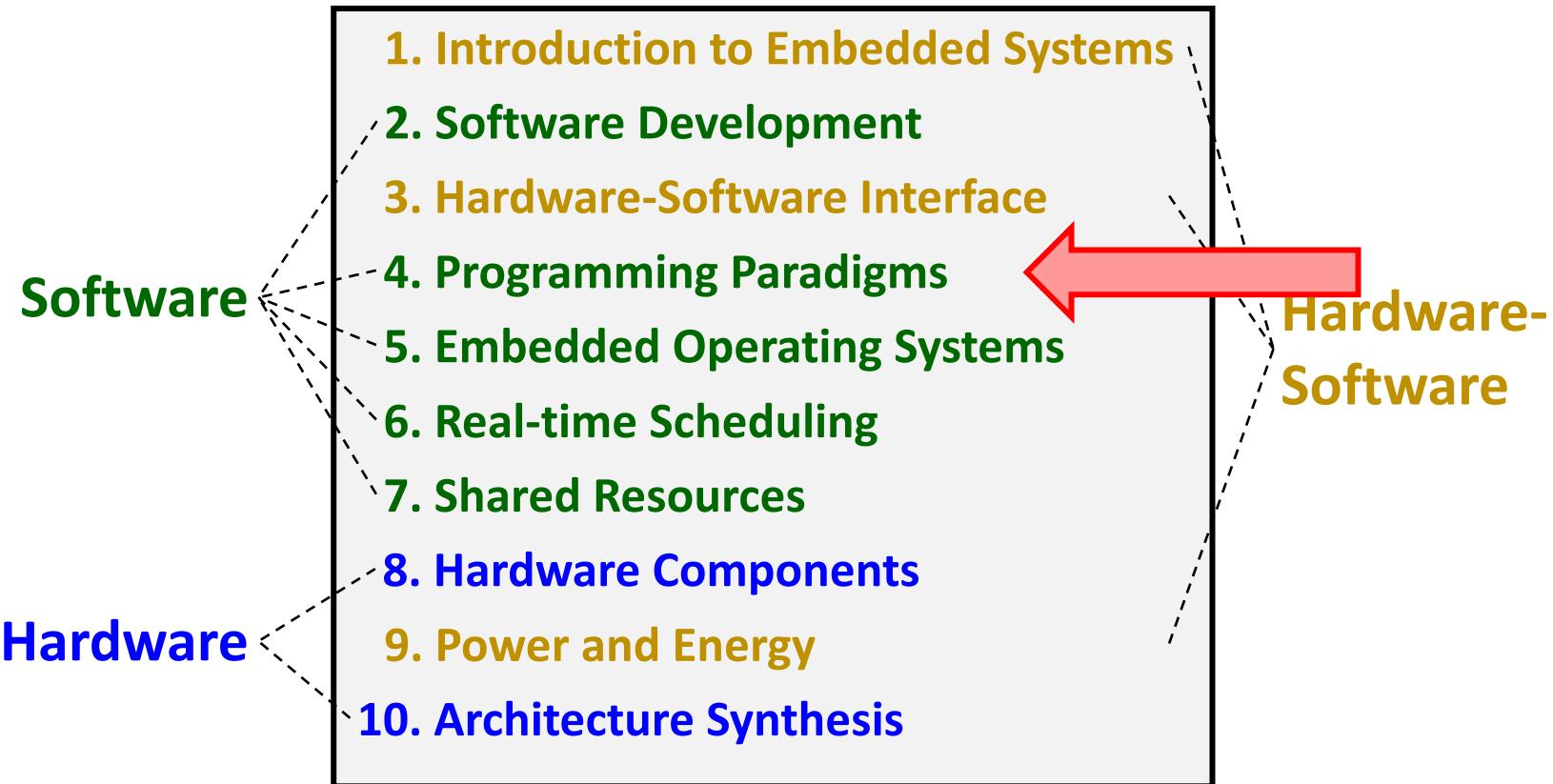
Exercises:

- From 12:00 to 14:00 in [HS 00-036 \(English\)](#) and [SR 02-016/18 \(German\)](#)
- Today (November 22):
 - Solutions of second exercise sheet presented
- Next week (November 29):
 - Third exercise sheet released + overview of tasks given

Exam: [March 4, 2023](#) from 10:00 to 12:00, five rooms in Georges-Köhler-Allee 101



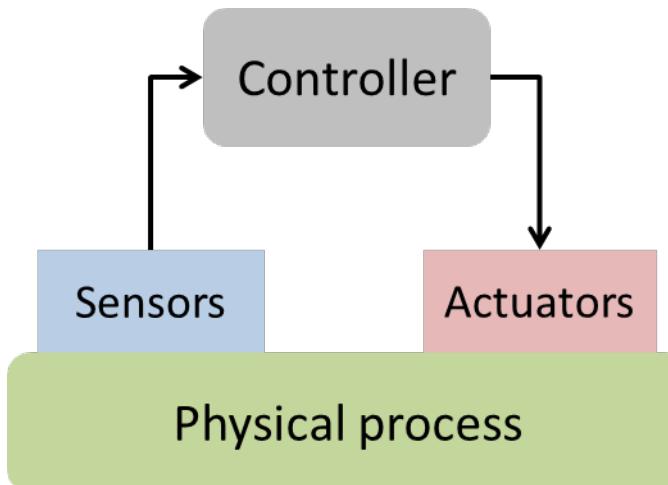
Where we are ...



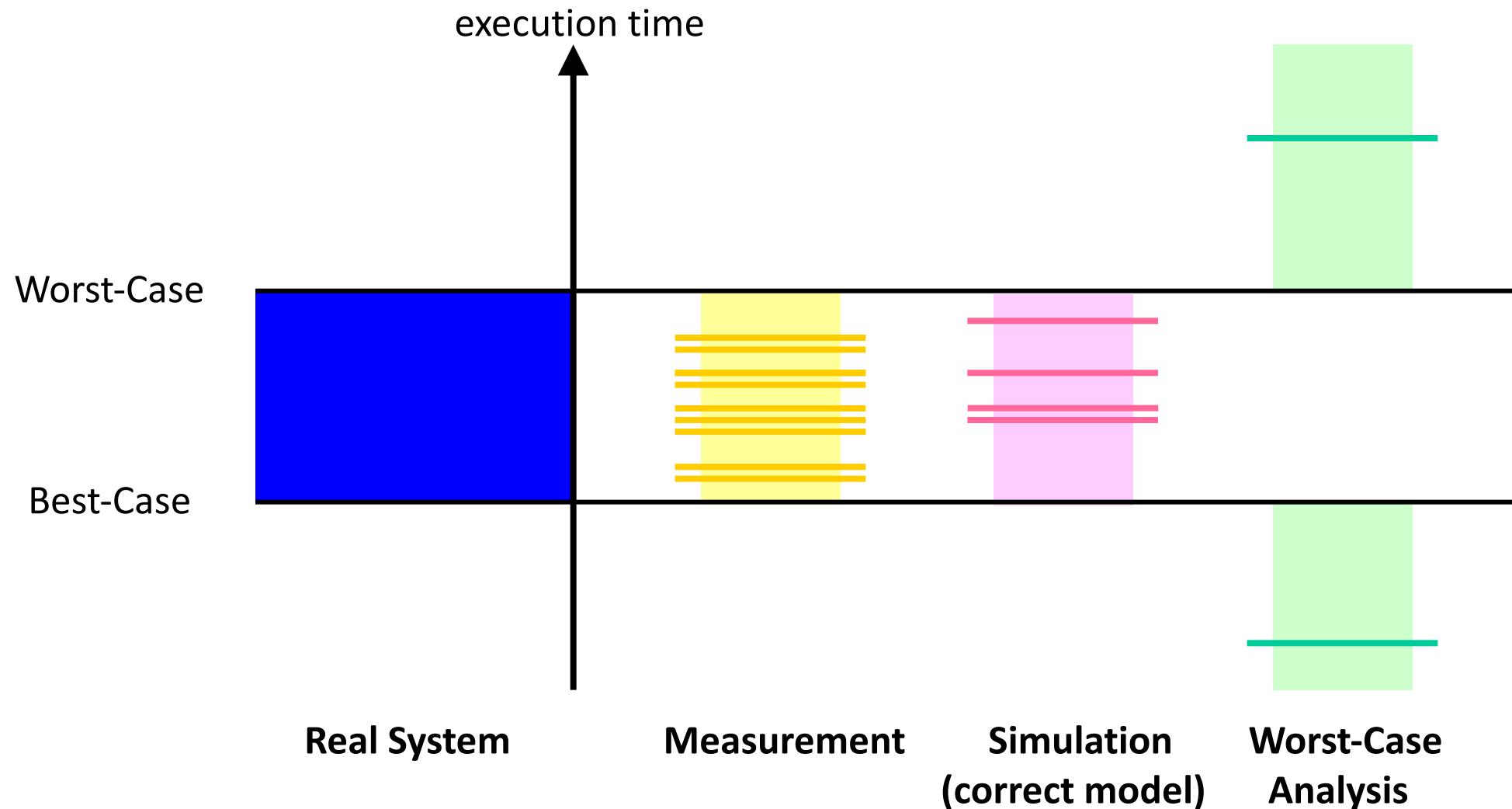
Reactive Systems and Timing

Real-Time Systems

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



Methods to Determine the Execution Time of a Task



Different Programming Paradigms

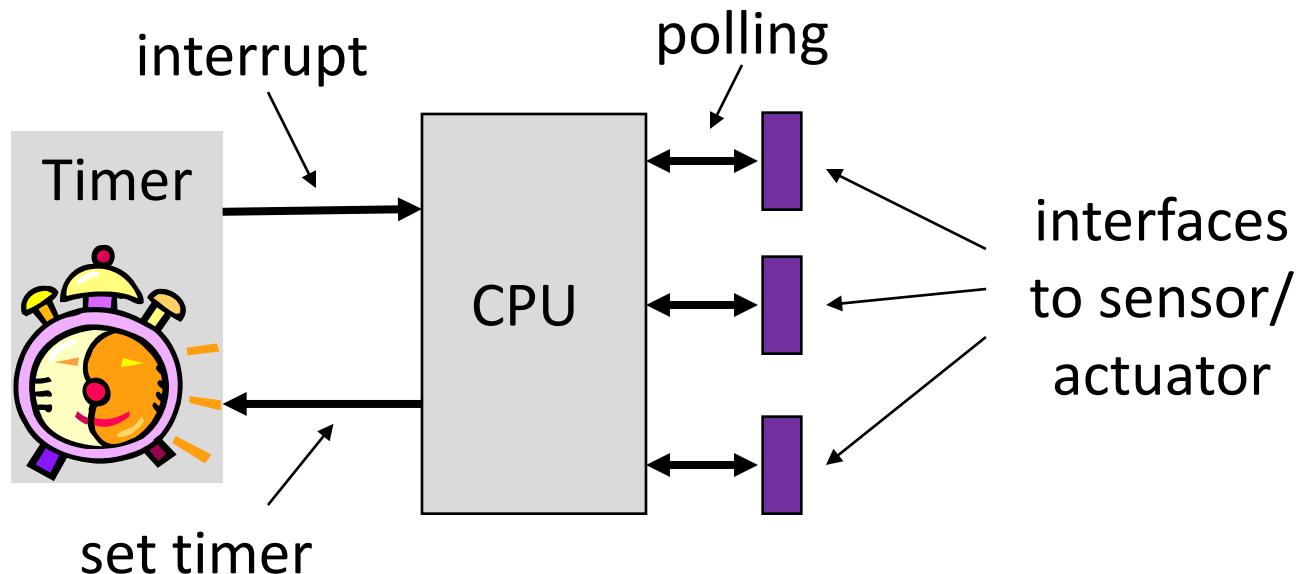
Overview

- 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

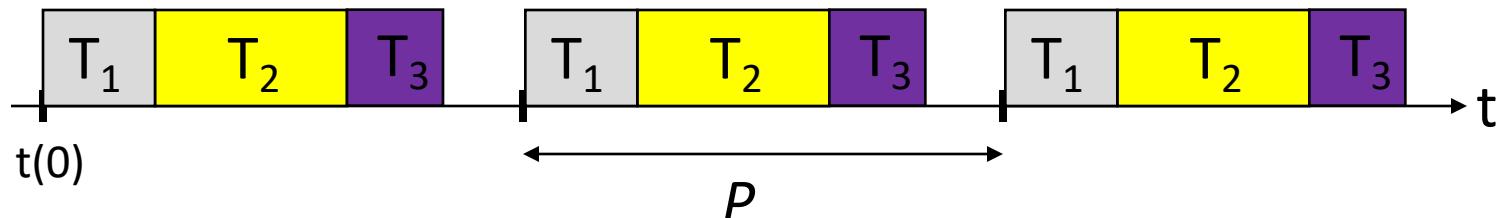
Pure time-triggered model:

- *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* .

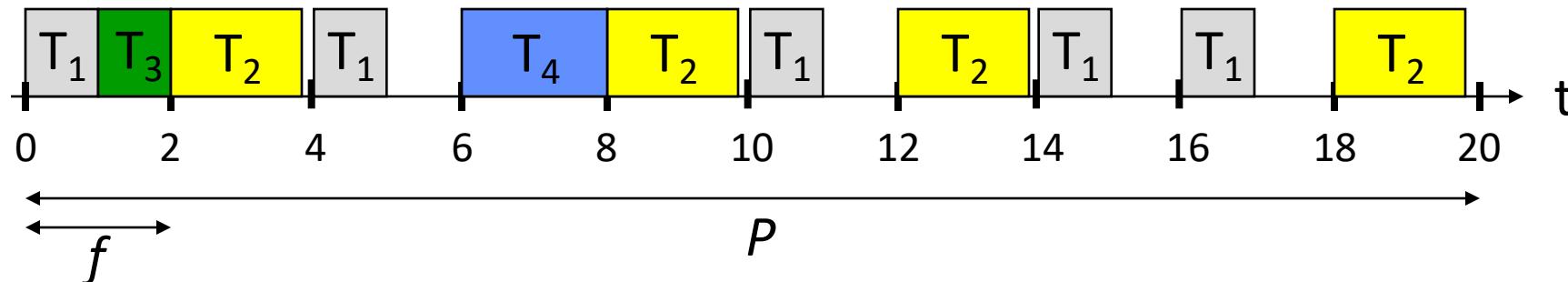


- *Properties:*
 - later tasks, for example T_2 and T_3 , have unpredictable starting times
 - the communication between tasks or the use of common resources is safe, as there is a static ordering of tasks, for example T_2 starts after finishing T_1
 - as a necessary precondition, the sum of WCETs of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

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

Time-Triggered Cyclic Executive Scheduler

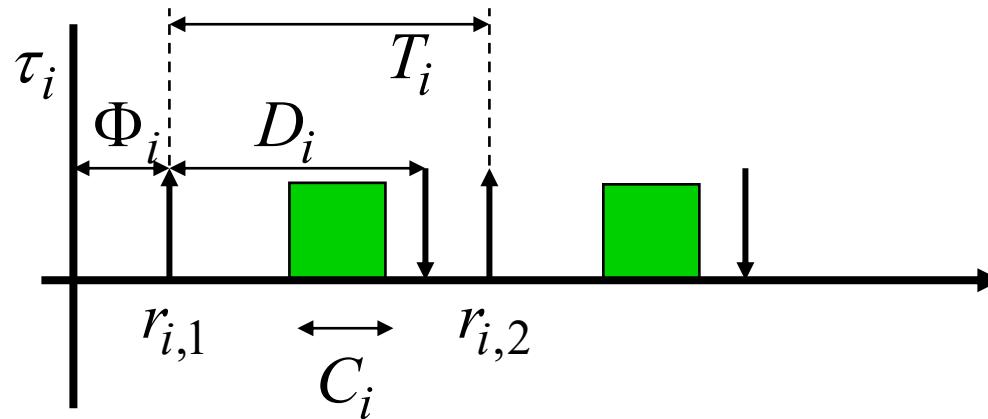
- Suppose now that *tasks may have different periods*.
- To accommodate this situation, the *period P is partitioned into frames of length f*.



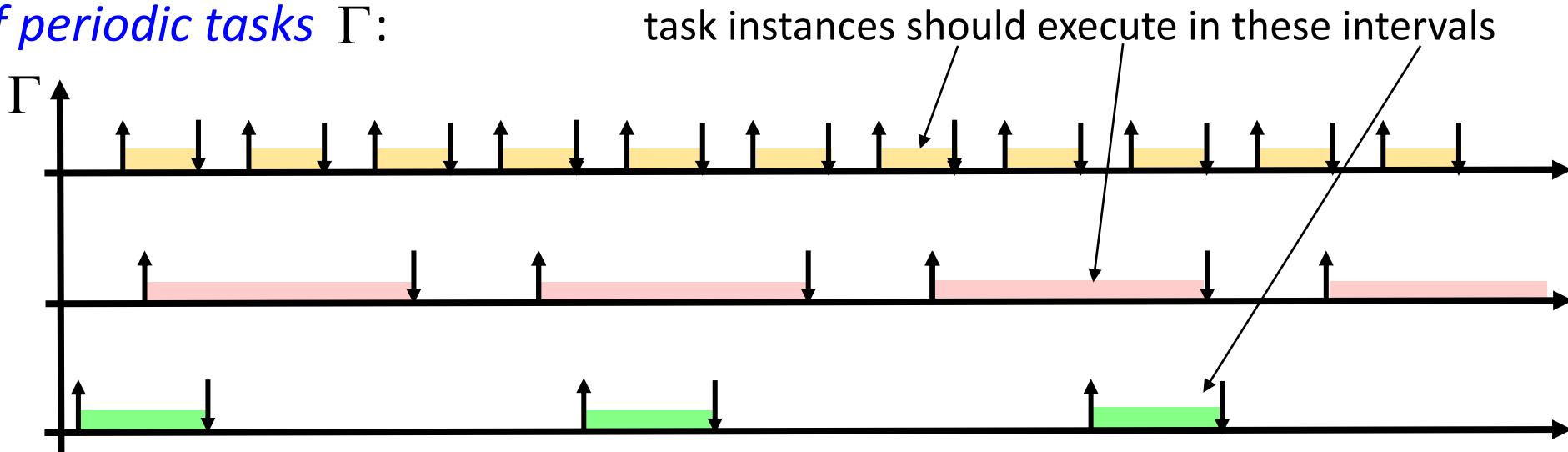
- We have a *problem* to determine a feasible schedule if there are *tasks with a long execution time*.
 - long tasks could be partitioned into a sequence of short sub-tasks
 - but this is a tedious and error-prone process, as the local state of the task must be extracted and stored globally

Time-Triggered Cyclic Executive Scheduling

- *Example* of a single periodic task τ_i :



- *A set of periodic tasks* Γ :



Time-Triggered Cyclic Executive Scheduling

Some conditions for period P and frame length f :

- A task executes at most once within a frame:

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

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:

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

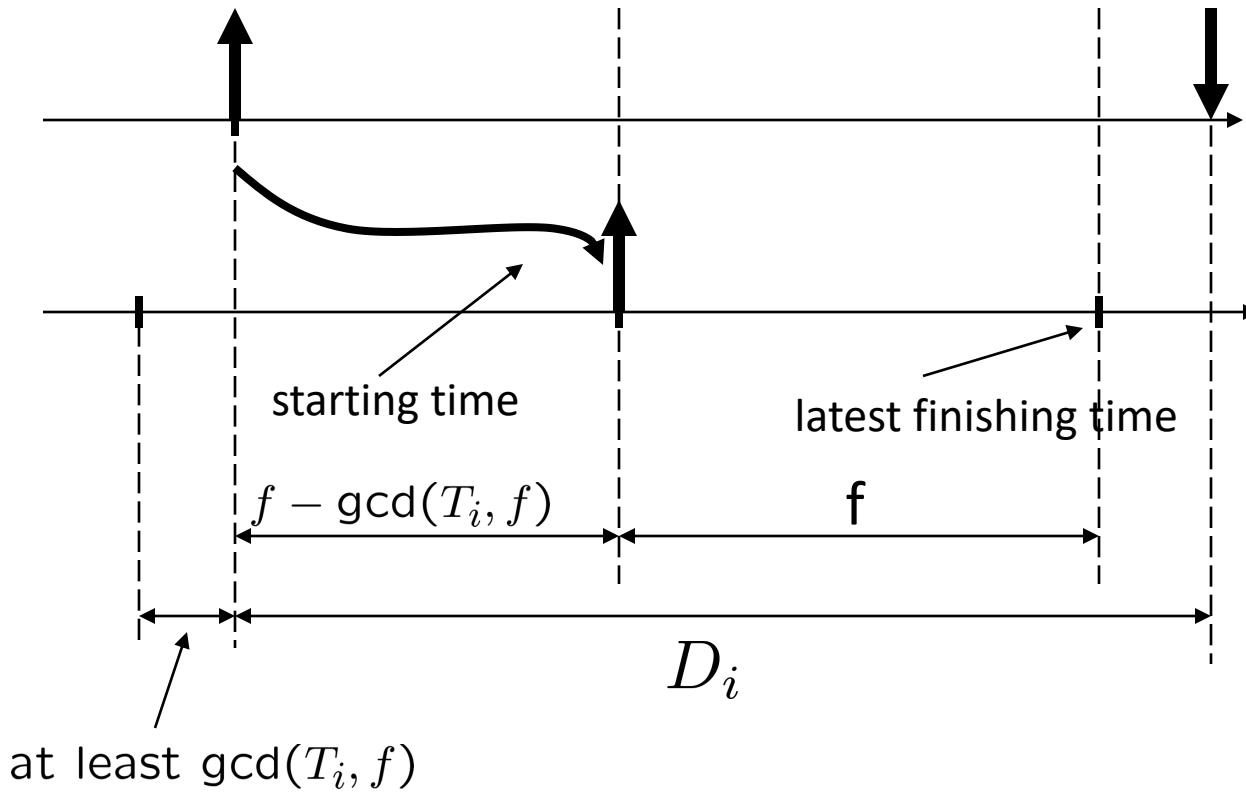
worst case execution time
of task

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

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

relative deadline of task

Sketch of Proof for Last Condition



release times and deadlines of tasks

frames

Example: Cyclic Executive Scheduling

Conditions:

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

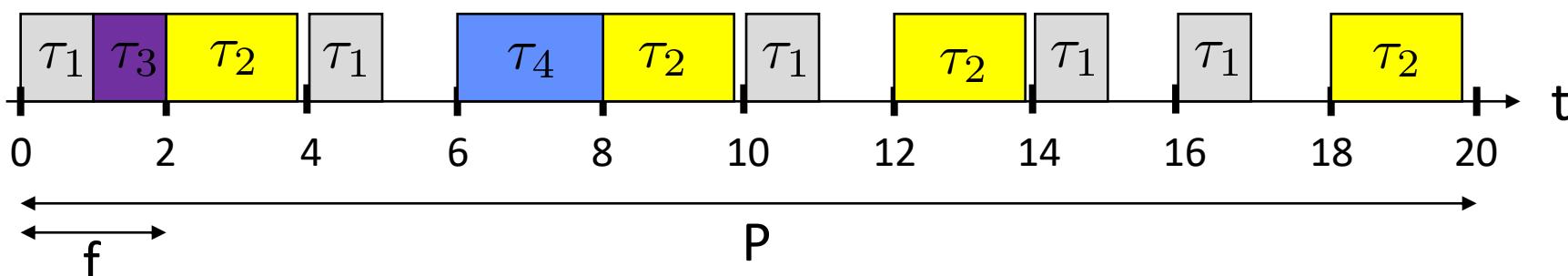
$$f \geq \max\{1.0, 1.0, 1.8, 2.0\} = 2.0$$

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

possible solution: $f = 2$

Γ	T_i	D_i	C_i
τ_1	4	4	1.0
τ_2	5	5	1.8
τ_3	20	20	1.0
τ_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 τ_i executes.
- Is P a common multiple of all periods T_i ?
- Is P a multiple of f ?
- Is the frame sufficiently long?

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

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

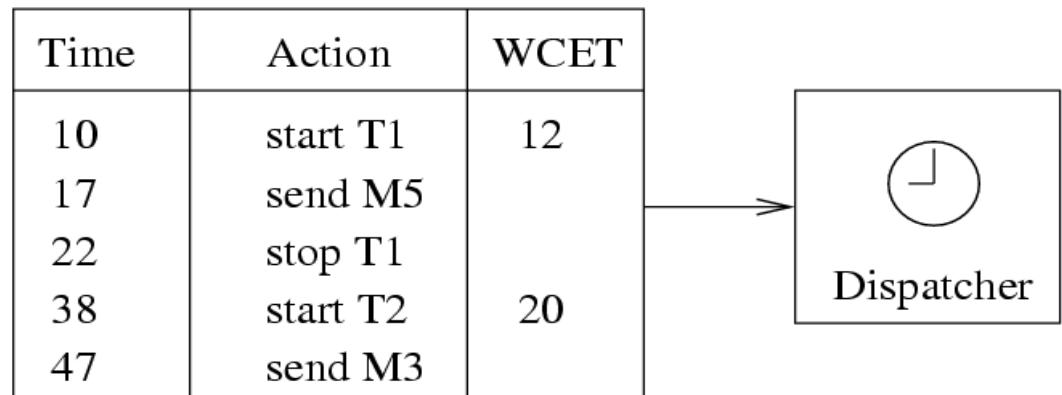
$$\Phi_i = \min_{1 \leq j \leq P/T_i} \{(f_{ij} - 1)f - (j - 1)T_i\} \quad \forall \text{ tasks } \tau_i$$

- Are deadlines respected?

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

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].



start times can be arbitrarily chosen within the period

Simplified Time-Triggered Scheduler

main:

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

Timer Interrupt:

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

Advantages:

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

Disadvantages:

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

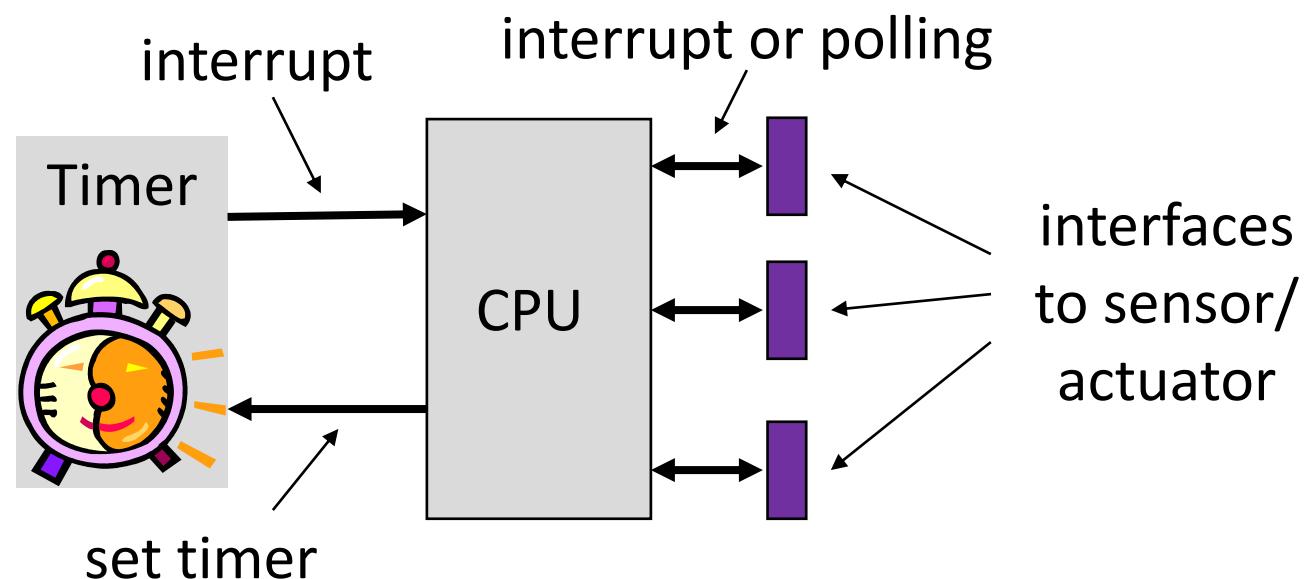
Extensions:

- *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 (e.g., first come first serve), an event is chosen for execution, that is, the corresponding task is executed.
- A running task *cannot be preempted* by another task. It can only be preempted by an interrupt that registers an event and puts it into the queue.

Extensions:

- A *background task*, which has the lowest priority, 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 certain time interval has elapsed. This enables, for example, periodic task instantiations.

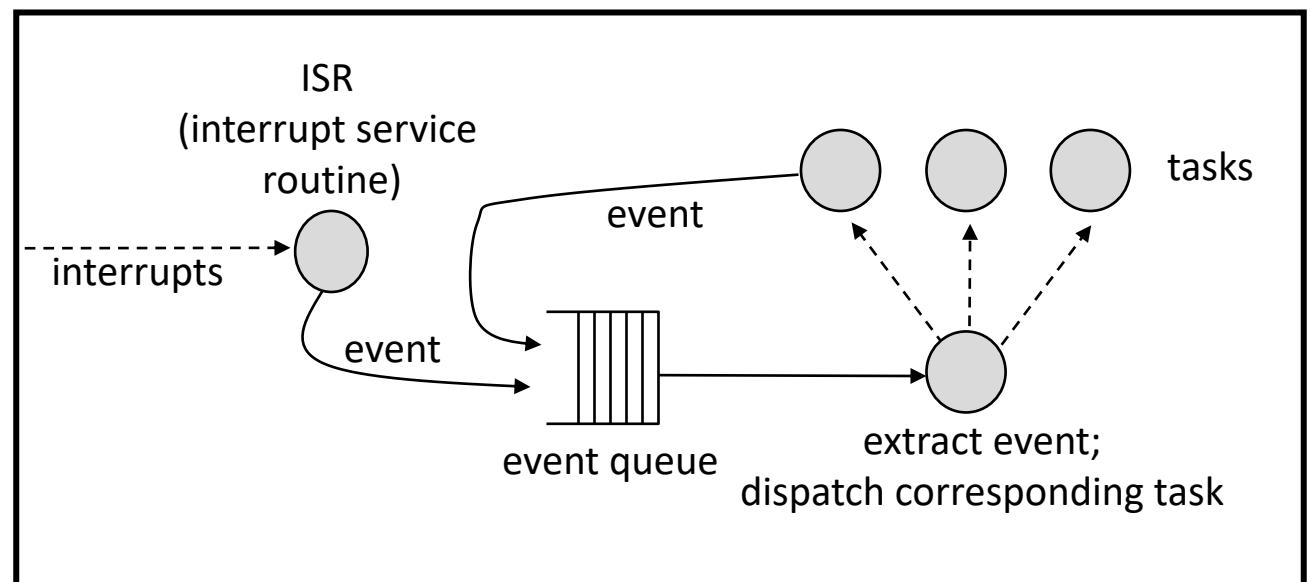
Non-Preemptive Event-Triggered Scheduling

```
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

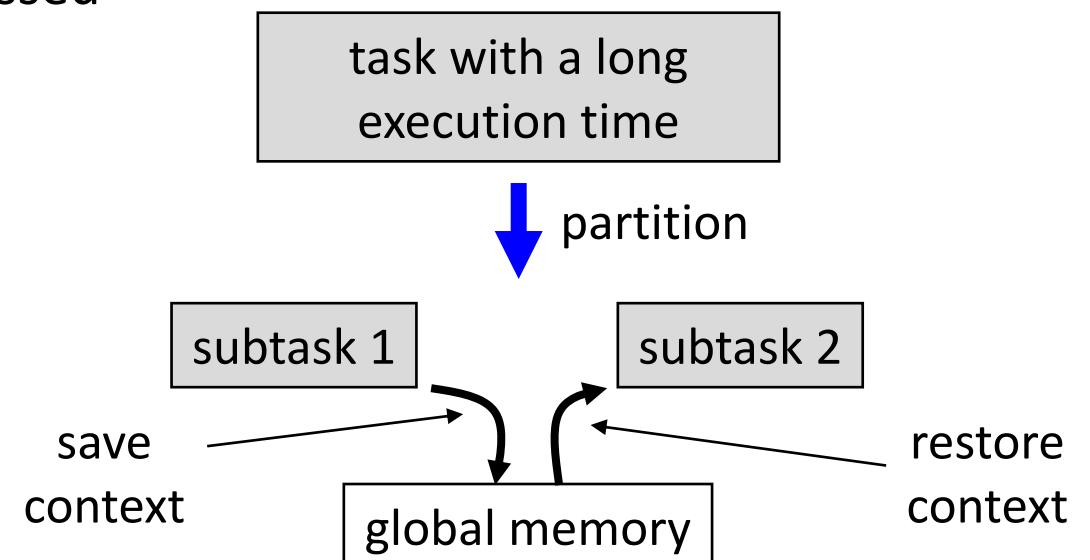
for example using a function pointer in C;
task returns after finishing.



Non-Preemptive Event-Triggered Scheduling

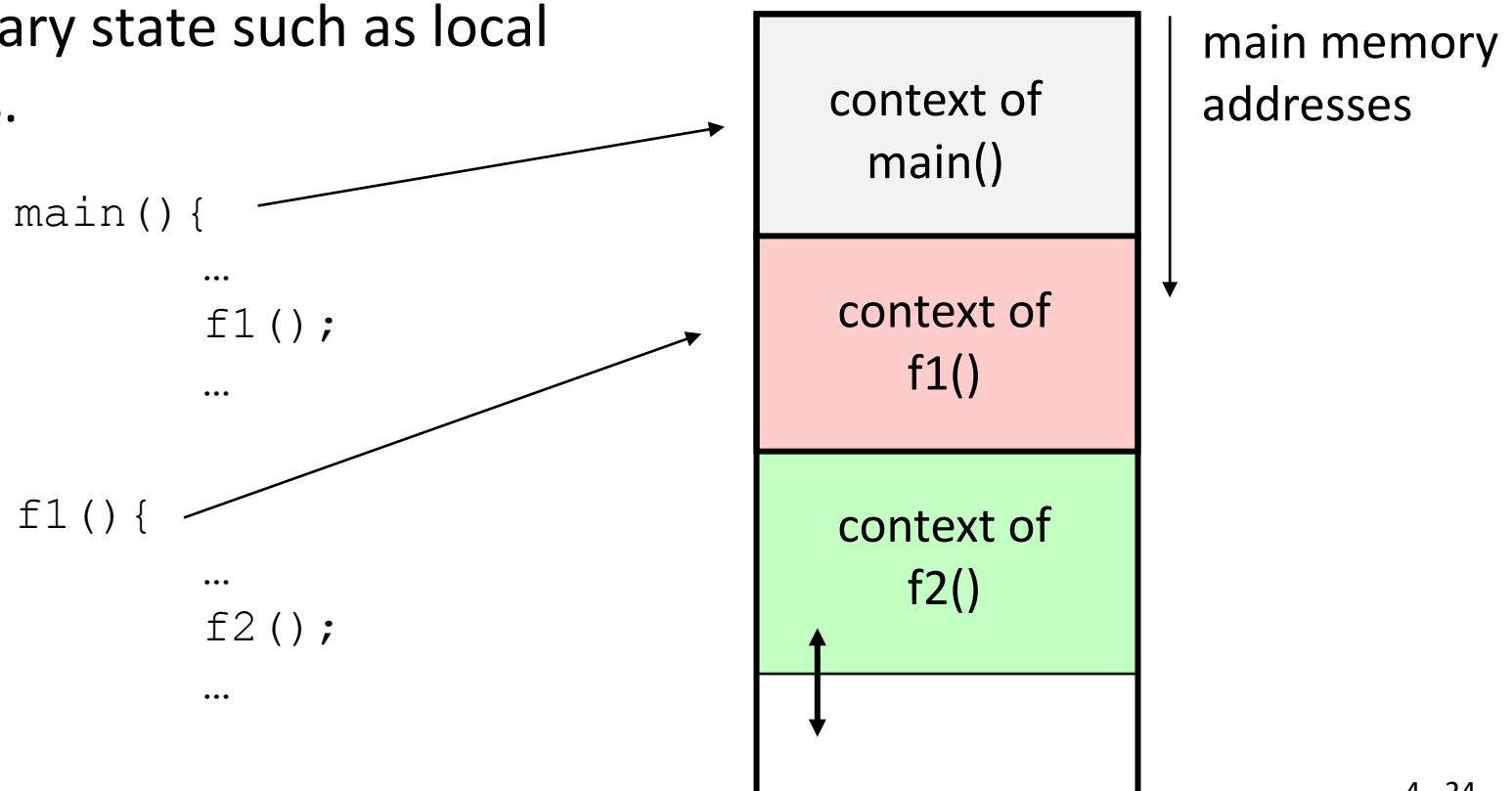
Properties:

- *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* of the event queue may happen if too many events are generated by the environment or by tasks (guarantee requires bounded behavior of environment)
- *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

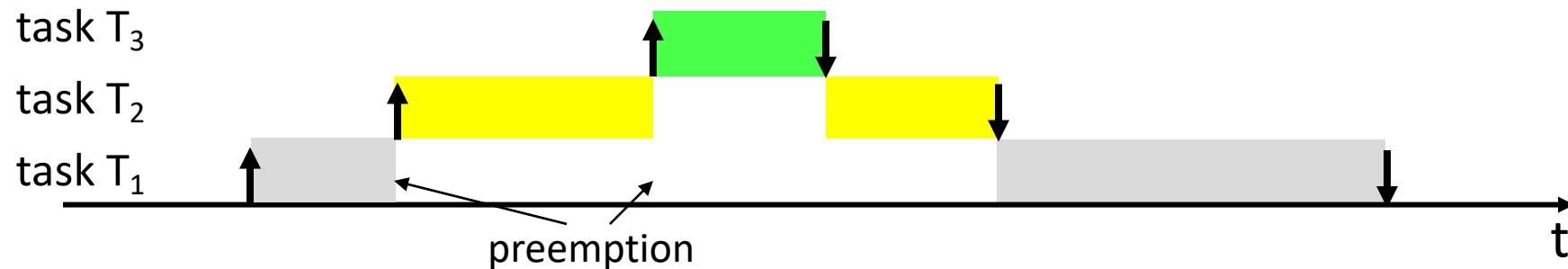


Preemptive Event-Triggered Scheduling – Stack Policy

- Each event/task has a *fixed priority*. Tasks with *higher priority can preempt* tasks with lower priority. This partly solves the problem of long-running tasks.
- 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, that is, the preempting task must finish before the preempted task can continue.
 - this restricts flexibility of the approach
 - 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;
    }
}
```

set CPU to low power mode;
processing continues after interrupt

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

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;
```

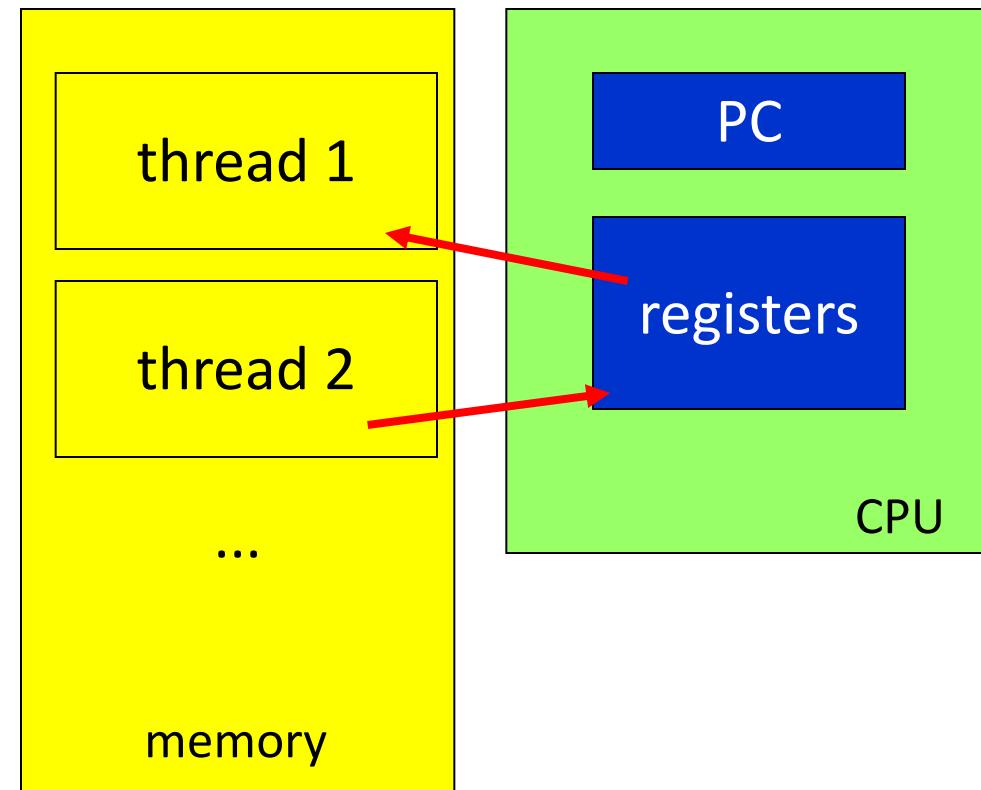
may be called by interrupt service
routines (ISR) or tasks

Thread

- *A thread is a unique execution of a piece of code.*
 - 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*, or *thread context*, contains the thread-local state, including registers and local data structures.
- Each thread has a *fixed memory region* for storing its context.
- *Context switch*:
 - current CPU context (program counter, registers) 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. This could be a different thread, or the same one in case no other thread is ready to run.
- ***Advantages:***
 - predictable, where context switches can occur (programmer has full control)
 - less errors with use of shared resources if the switch locations are chosen carefully
- ***Disadvantages:***
 - programming errors (e.g., if the `cswitch()` function is never called) can keep other threads out as the running thread may never give up the CPU
 - real-time behavior may be at risk if a thread runs for too long before the next context switch is allowed

Example: Co-operative Multitasking

Thread 1

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

Thread 2

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

Scheduler

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

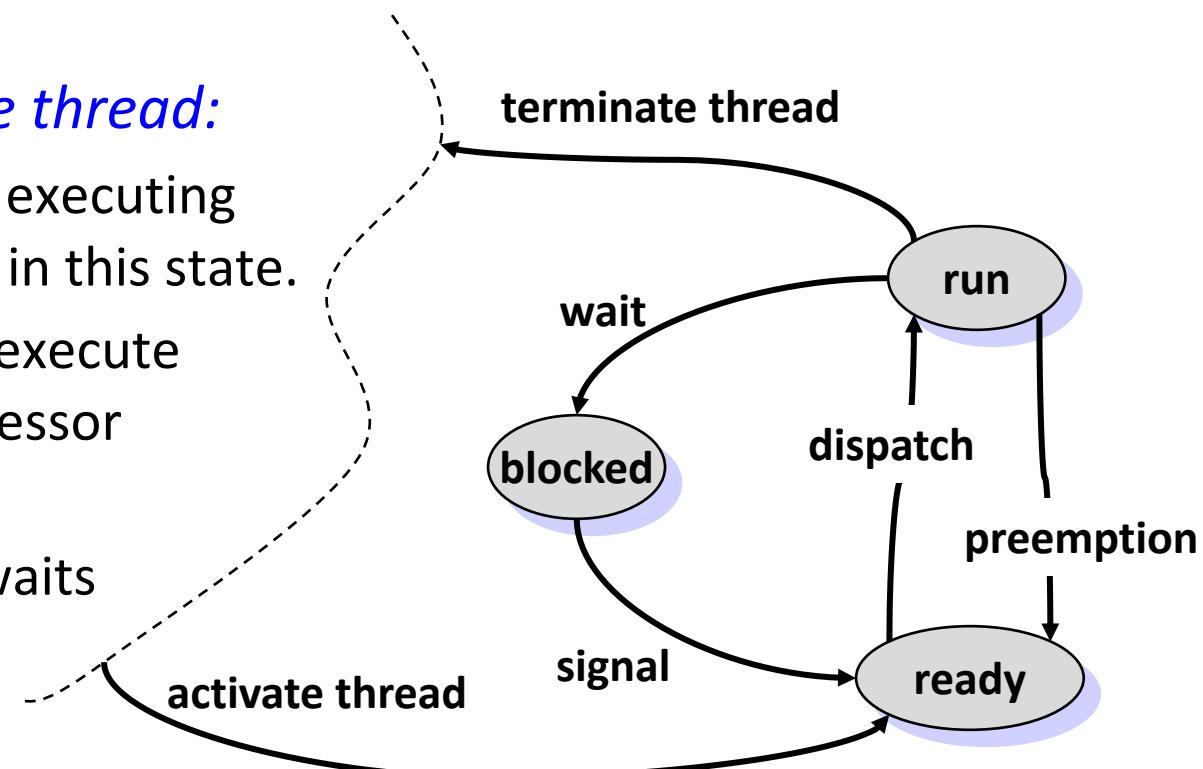
Preemptive Multitasking

- *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. Only one thread can be in this state.
- *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.



Introduction to Embedded Systems

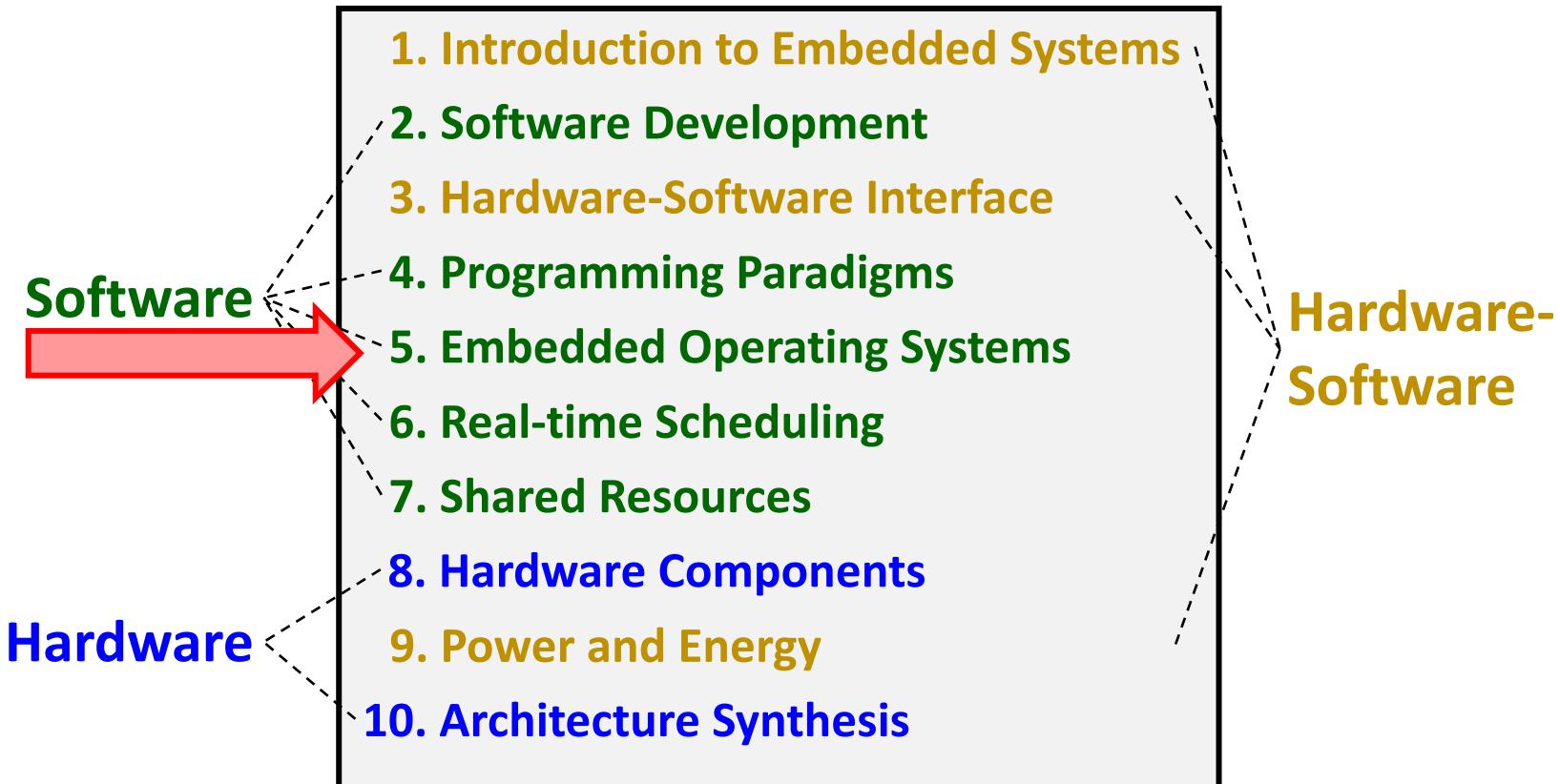
5. Operating Systems

Prof. Dr. Marco Zimmerling



Embedded Operating Systems

Where we are ...



Embedded Operating System: Motivation

- *Why an operating system (OS) at all?*
 - Same reasons why we need one for a traditional computer.
 - Not every device needs all services.
- In embedded systems we find a *large variety of requirements and environments*:
 - Critical applications with broad functionality (medical applications, space shuttle, process automation, ...).
 - Critical applications with little functionality (ABS, pace maker, ...).
 - Not very critical applications with broad range of functionality (smart phone, ...).

Embedded Operating System: Motivation

- *Why is a desktop OS not suited?*

- Monolithic kernel of a desktop OS often not modular, fault-tolerant, and configurable.
- Typically offers many features that may not be needed and consume too many resources (e.g., energy, memory, compute time) for an embedded system.
- Generally not designed for mission- or safety-critical applications. For example, the timing uncertainty may be too high to give any real-time guarantees.

Embedded Operating Systems

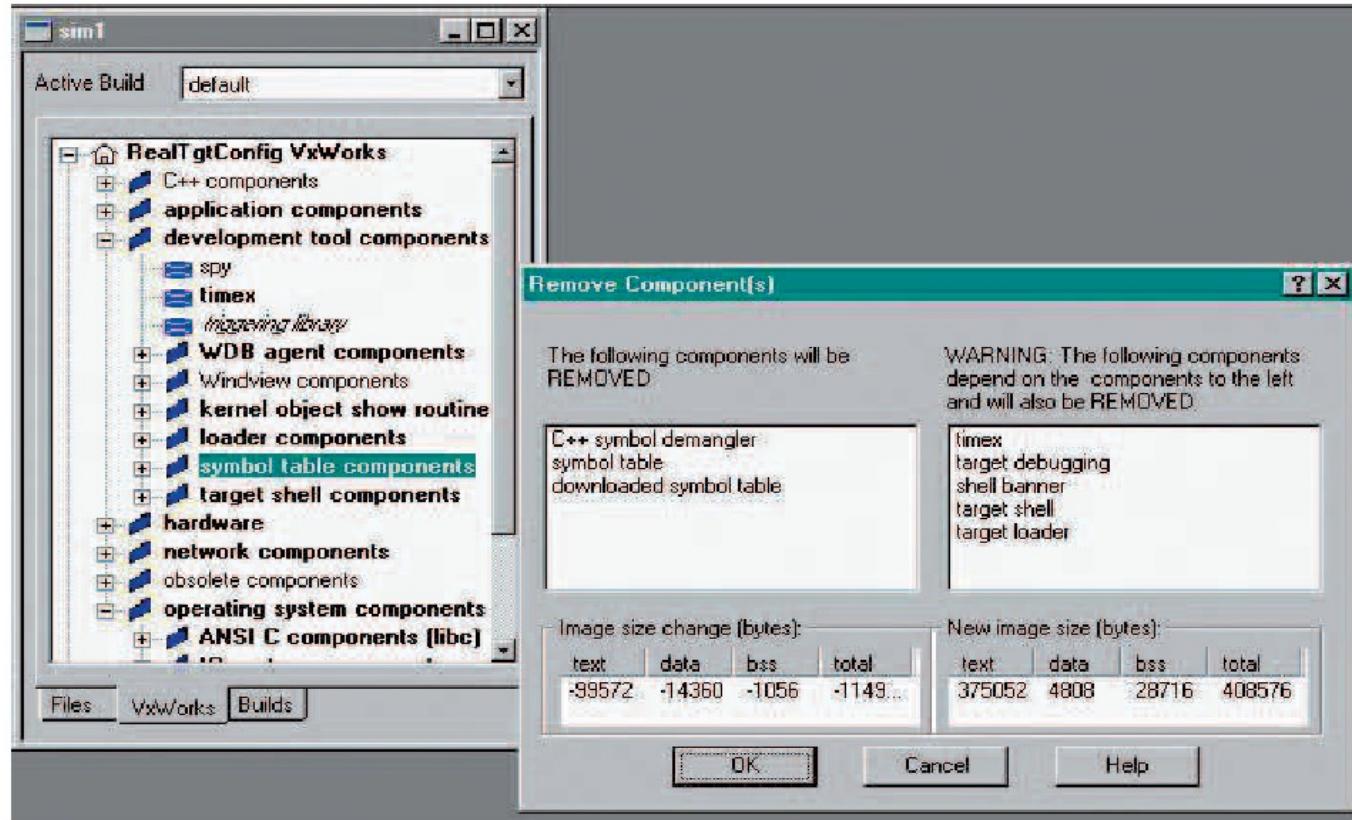
Essential characteristics of an embedded OS: *Configurability*

- *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 (e.g., by the linker).
- *Use conditional compilation* (e.g., using `#if` and `#ifdef` commands in C).
- But deriving a *consistent configuration* becomes challenging if the set of possible combinations of functions is large (e.g., relevant components may be missed).

Example: Configuration of VxWorks

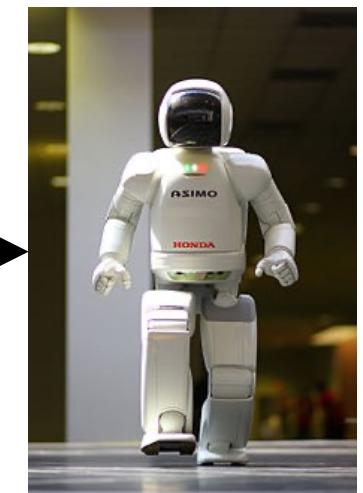


Automatic dependency analysis and size calculations allow users to quickly customize the VxWORKS operating system.

© Windriver



Mars science rover



ASIMO robot

Real-Time Operating Systems (RTOS)

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. *The OS must manage 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

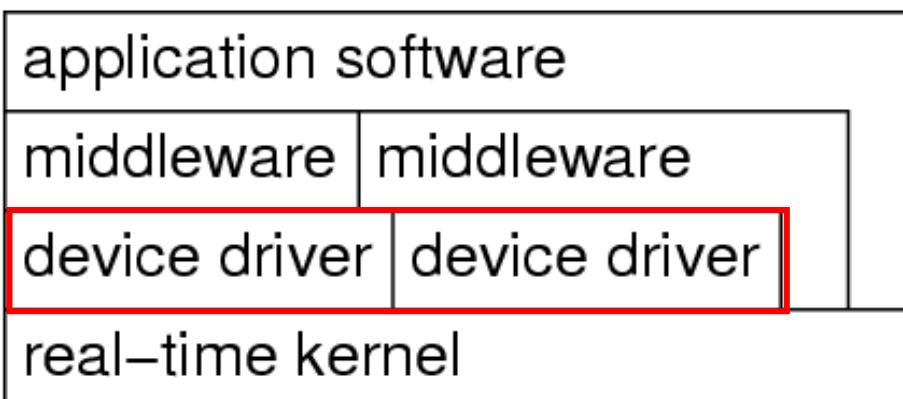
Features and Architecture

Embedded Operating System: Design Choices

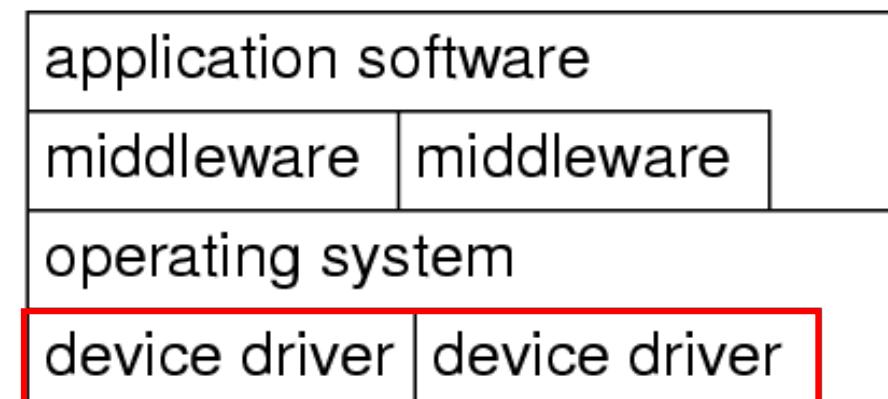
In an embedded OS, device drivers are typically handled directly by tasks rather than managed by the operating system itself:

- This architecture *improves timing predictability* as access to devices is also handled by the scheduler that runs within the real-time kernel.
- 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: Design Choices

Every task can perform an interrupt:

- In a *standard OS*, this would be a *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 start address of a task 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, and software can be considered to be reliable.
- However, protection mechanisms may be needed for *safety and security* reasons.

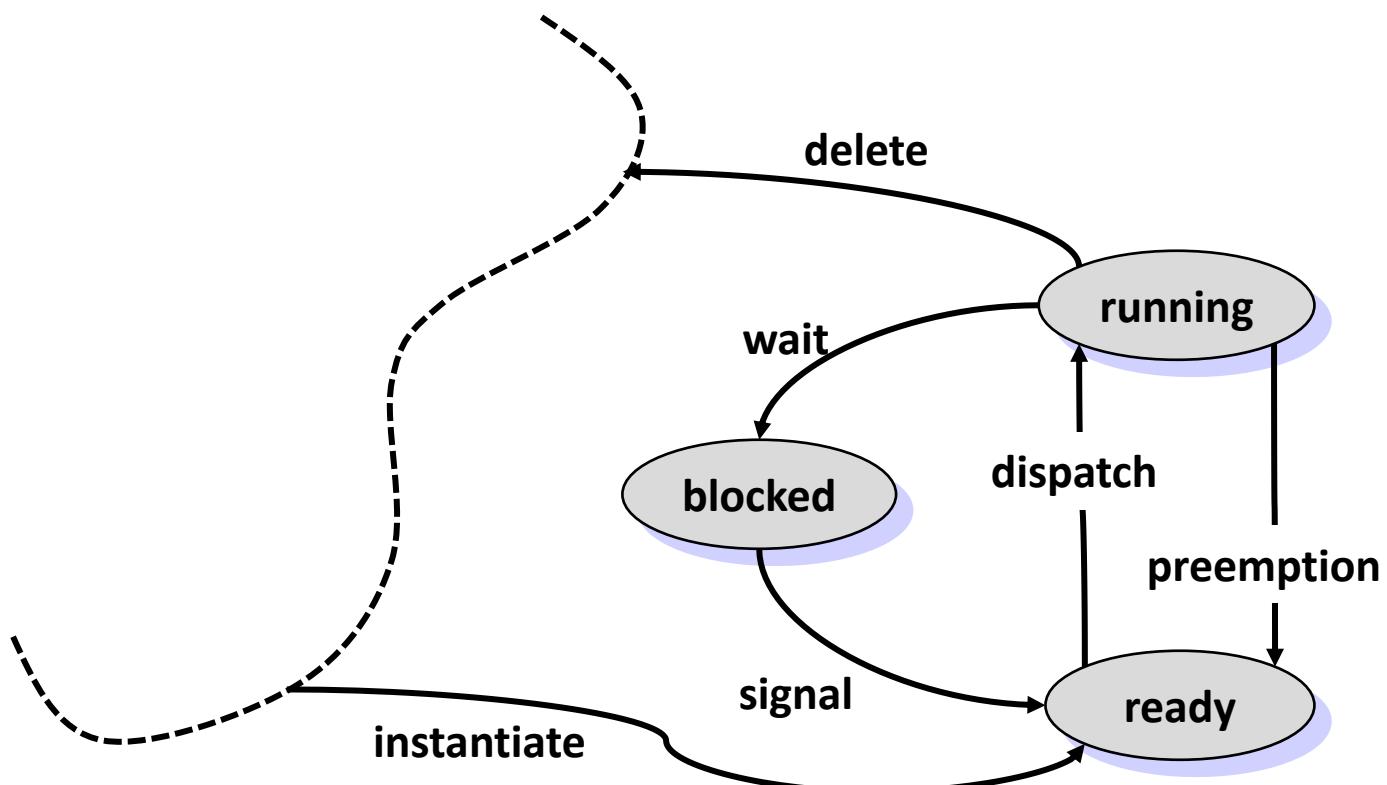
Main Functionality of Real-Time Operating System Kernels

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 are performed via semaphores and monitors.
 - In real-time OS, special semaphores and a deep integration of them into scheduling is necessary (e.g., 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 at 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 is in state “running”).

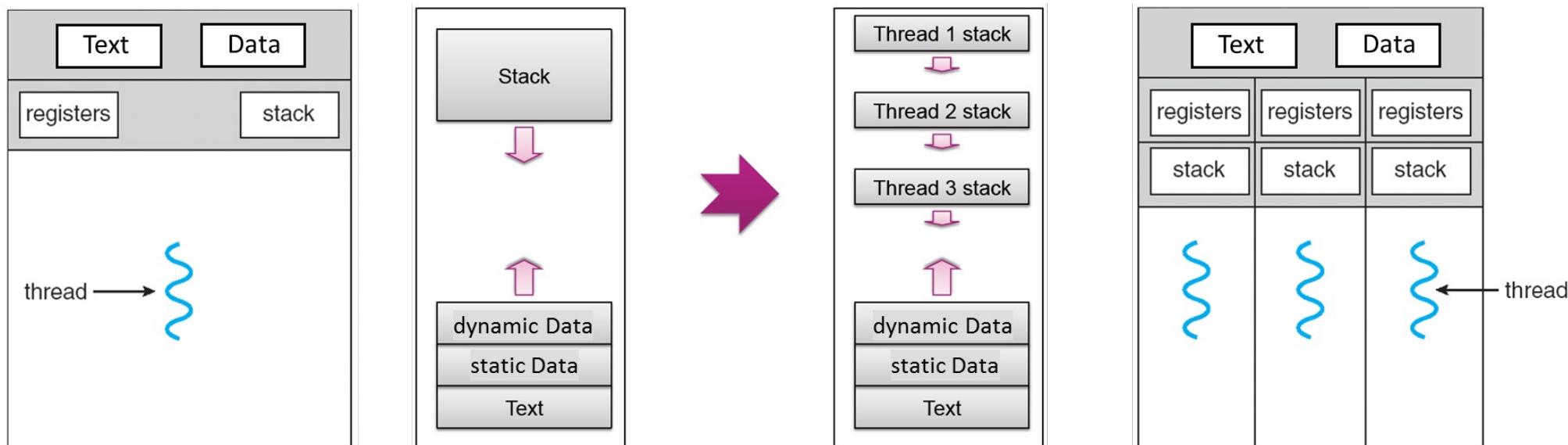
Blocked:

- A task enters this state when it executes a synchronization primitive to wait for an event (e.g., timer expires, mutually exclusive resource becomes available, data in a queue is read, or queue has again enough space to be written). In this case, the task is inserted in a queue associated with the event. The task at the head of the queue is resumed upon the occurrence of the corresponding event.

Threads

A thread is the smallest sequence of program instructions that can be managed independently by a scheduler. Thus, 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 across different threads: memory.
 - Typically owned by each individual thread: registers and stack.



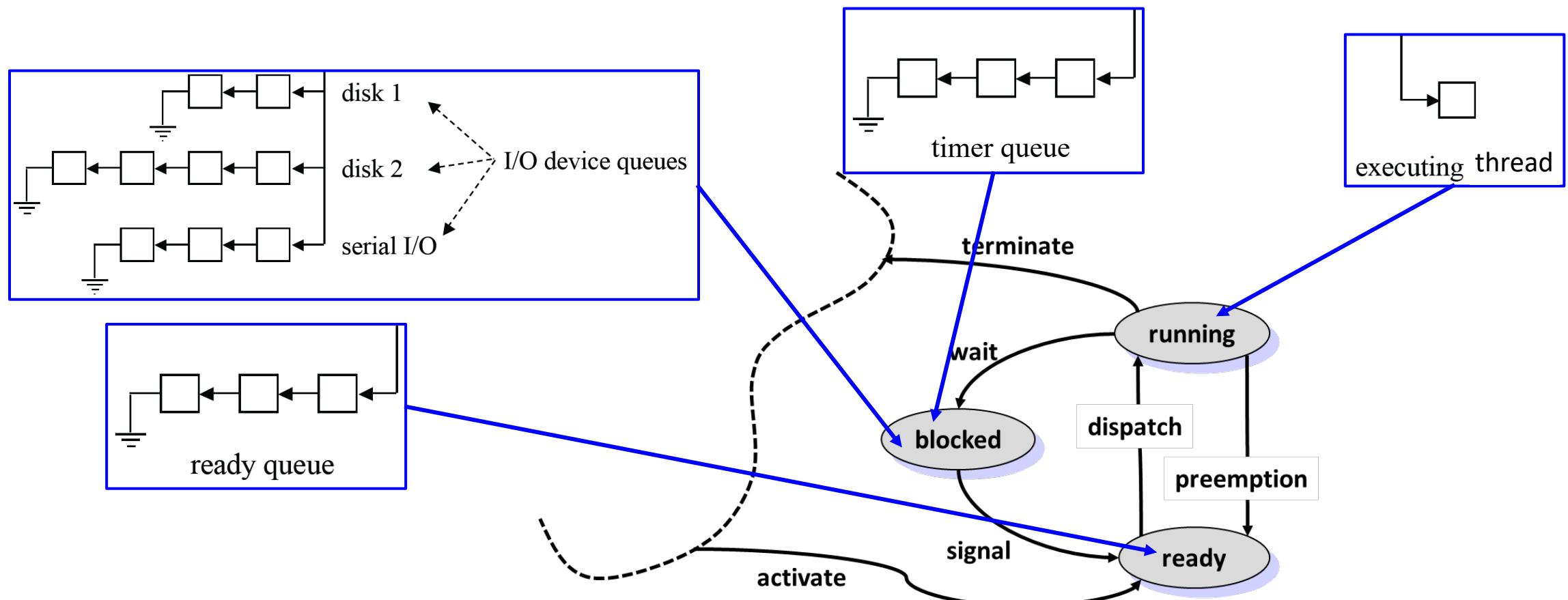
Threads

A thread is the smallest sequence of program instructions that can be managed independently by a scheduler. Thus, a thread is a basic unit of CPU utilization.

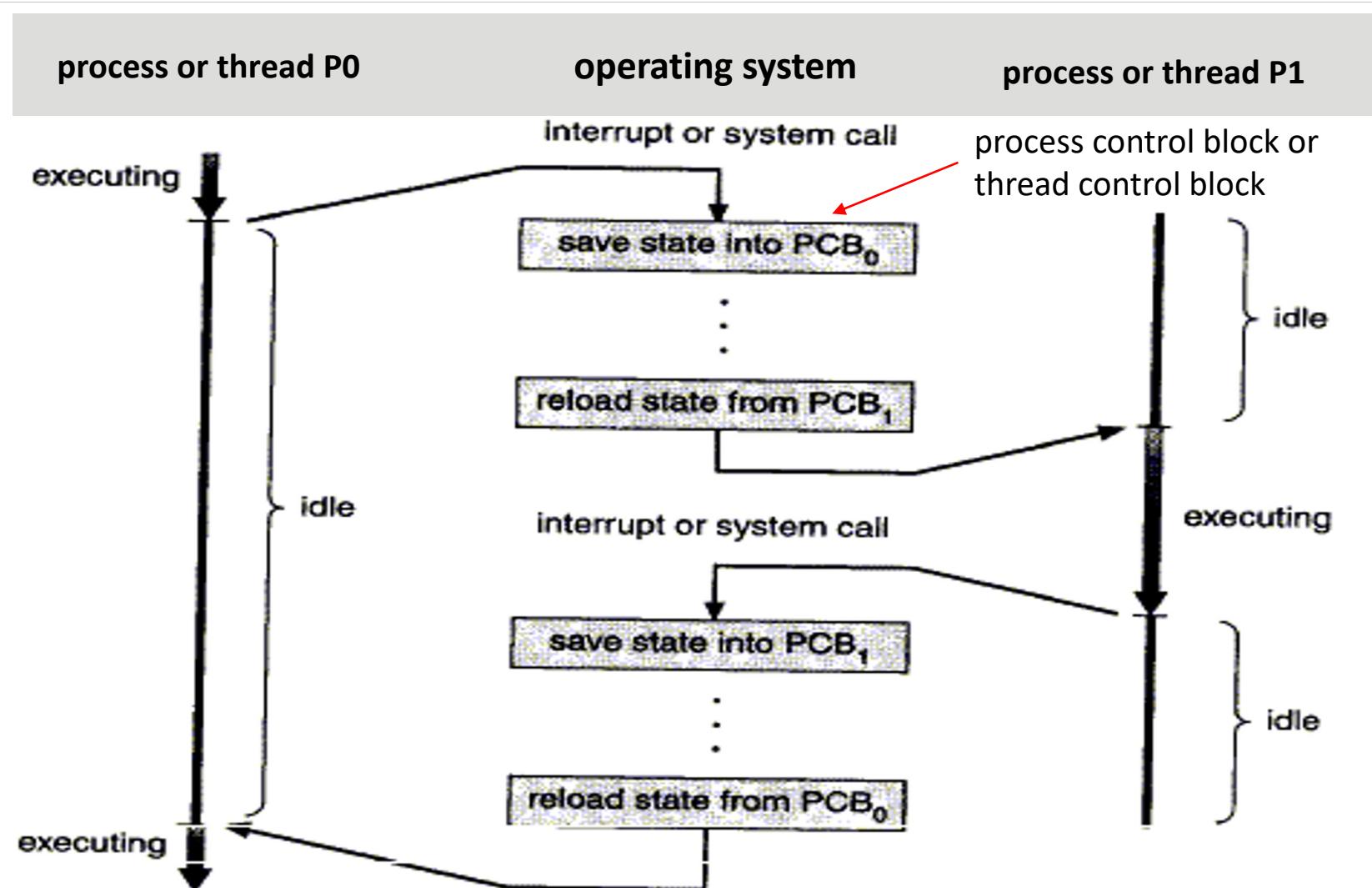
- *Multiple threads can exist within the same process* and share resources such as memory, while different processes do not share these resources:
 - Typically shared across different threads: memory.
 - Typically owned by each individual thread: registers and stack.
- *Thread advantages and characteristics:*
 - Faster to switch between threads than to switch between processes. Switching between user-level threads requires no major intervention by the OS.
 - Typically, an application will have a separate thread for each distinct activity.
 - The OS maintains for each thread a *Thread Control Block (TCB)* that stores all information needed to manage and schedule a thread. This includes the name of the thread, its priority and current state (e.g., program counter, scheduling info).

Thread Control Blocks (TCBs)

- The operating system manages TCBs using linked lists. Below is an example.



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 *predictable* in every respect.

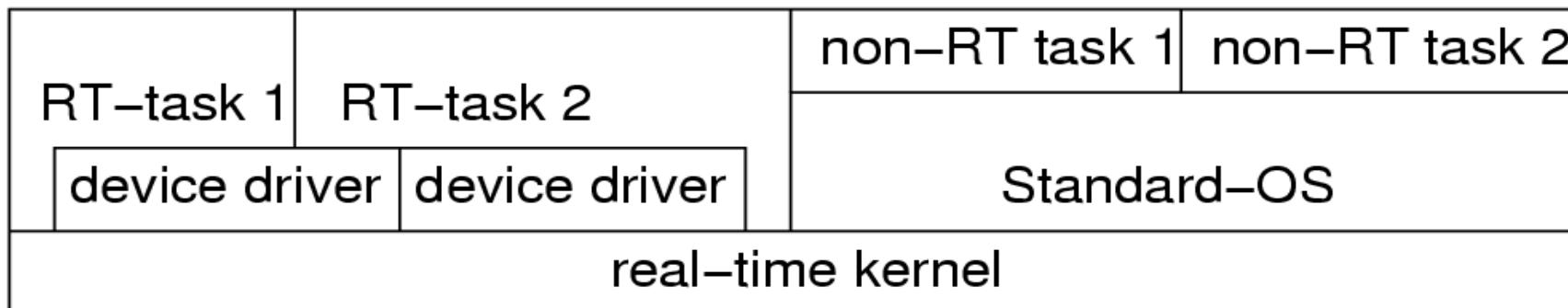
Examples include

FreeRTOS, QNX, eCOS, VxWORKS, LynxOS.

Class 2: Extensions to Standard OSs

Real-time extensions to standard OS:

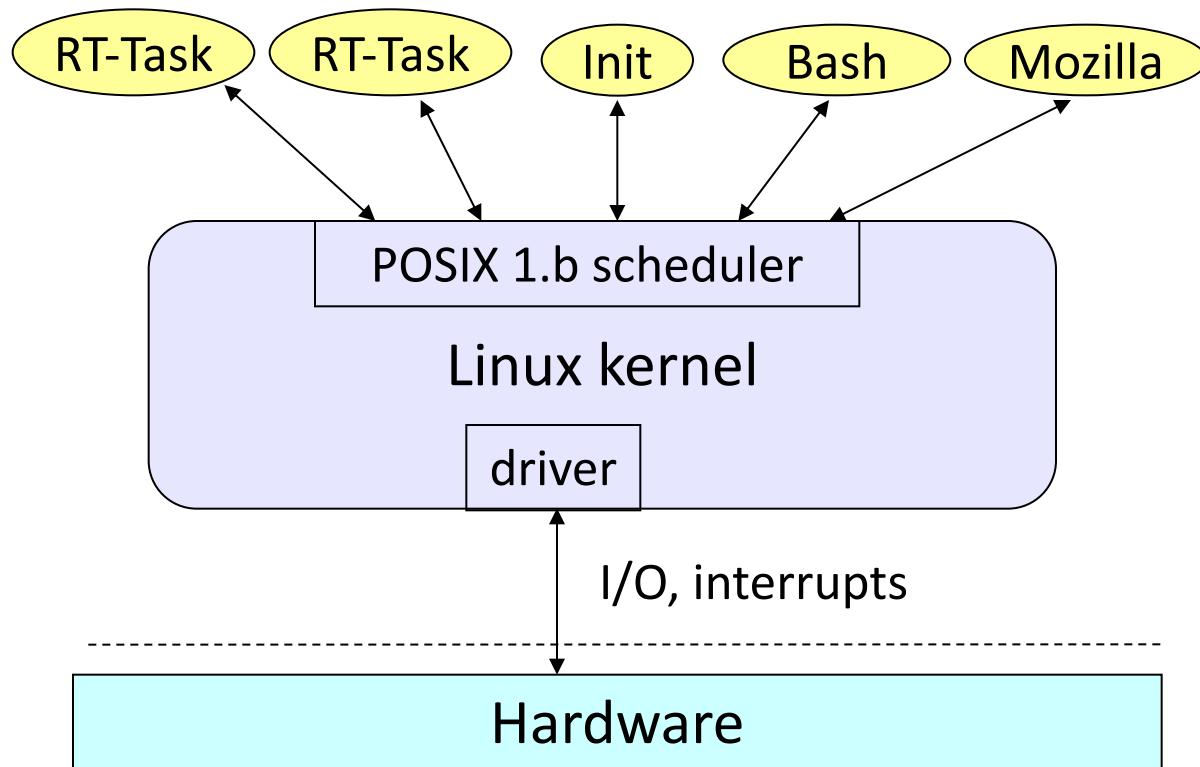
- Attempt to exploit existing and comfortable main stream operating systems.
- A real-time kernel runs all real-time (RT) tasks.
- The standard OS is executed as one task.



+ Crash of standard OS does not affect RT tasks
- RT tasks cannot use standard OS services
(→ less comfortable than expected)

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*.

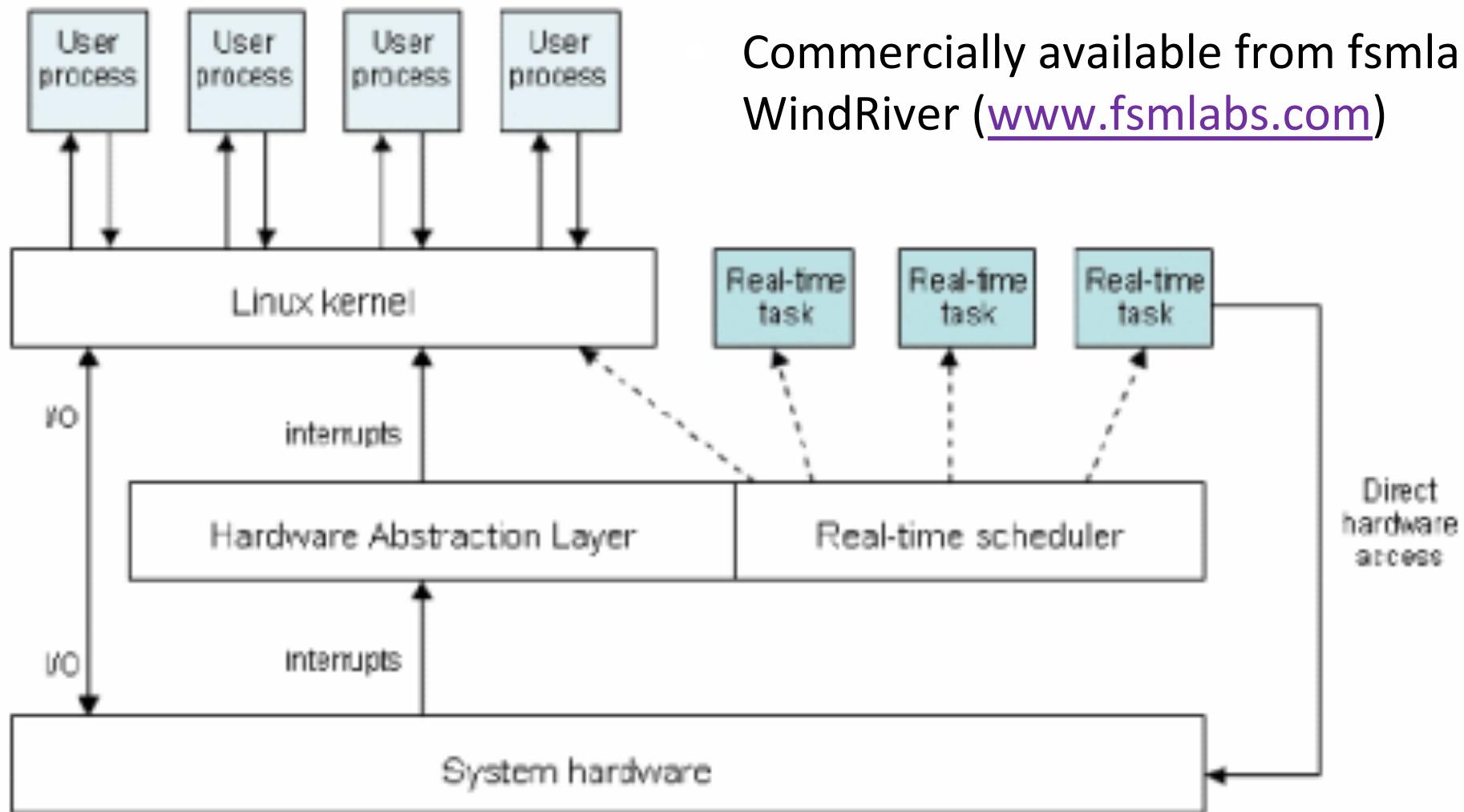


Special calls for real-time as well as standard operating system calls available.

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

Example: RT Linux

RT tasks cannot use standard OS calls.
Commercially available from fsm labs and
WindRiver (www.fsm labs.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

Introduction to Embedded Systems

5. Operating Systems

Prof. Dr. Marco Zimmerling



Organization

Join ILIAS course:

- Login: RZ username + password
- Course password: **es-0x8af**

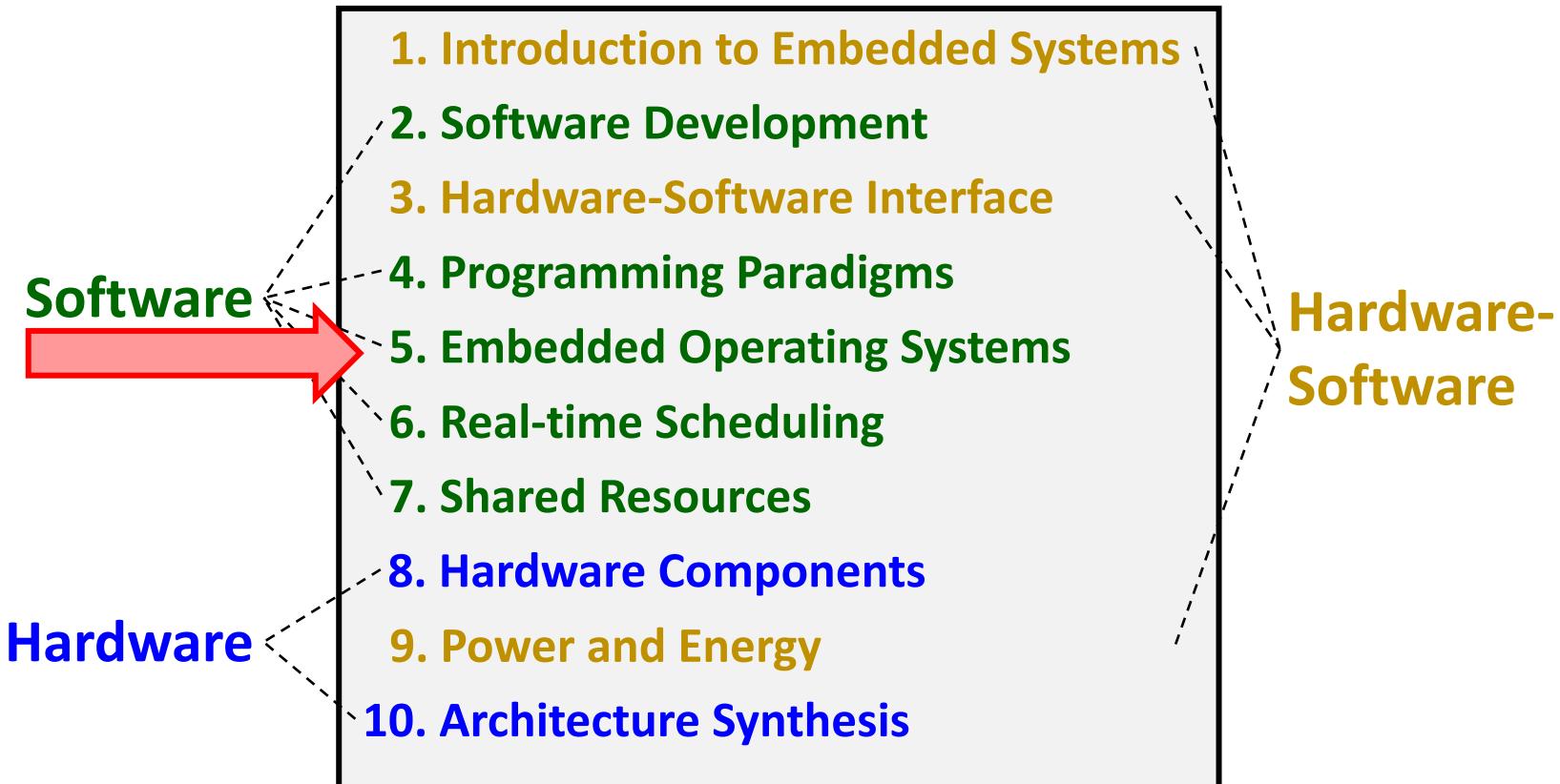
Exercises:

- From 12:00 to 14:00 in [HS 00-036 \(English\)](#) and [SR 02-016/18 \(German\)](#)
- Today (November 29):
 - Third exercise sheet released + overview of tasks given
- Next week (December 6):
 - Solutions of third exercise presented
 - Fourth exercise sheet released + overview of tasks given



Exam: [March 4, 2023](#) from 10:00 to 12:00, five rooms in Georges-Köhler-Allee 101

Where we are ...



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:

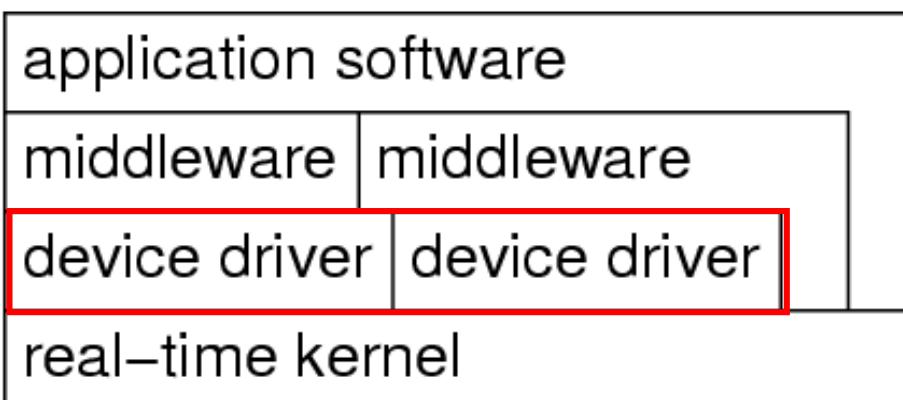
- *Remove unused functions*/libraries (e.g., by the linker).
- *Use conditional compilation* (e.g., using `#if` and `#ifdef` commands in C).
- But deriving a *consistent configuration* becomes challenging if the set of possible combinations of functions is large (e.g., relevant components may be missed).

Embedded Operating System: Design Choices

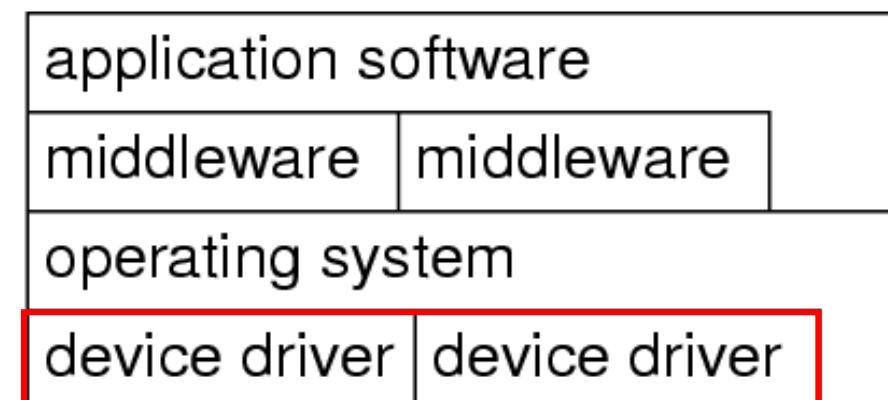
In an embedded OS, device drivers are typically handled directly by tasks rather than managed by the operating system itself:

- This architecture *improves timing predictability* as access to devices is also handled by the scheduler that runs within the real-time kernel.
- 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: Design Choices

Every task can perform an interrupt:

- In a *standard OS*, this would be a *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 start address of a task in the interrupt table). This approach is more efficient and predictable than going through the operating system's interfaces and services.

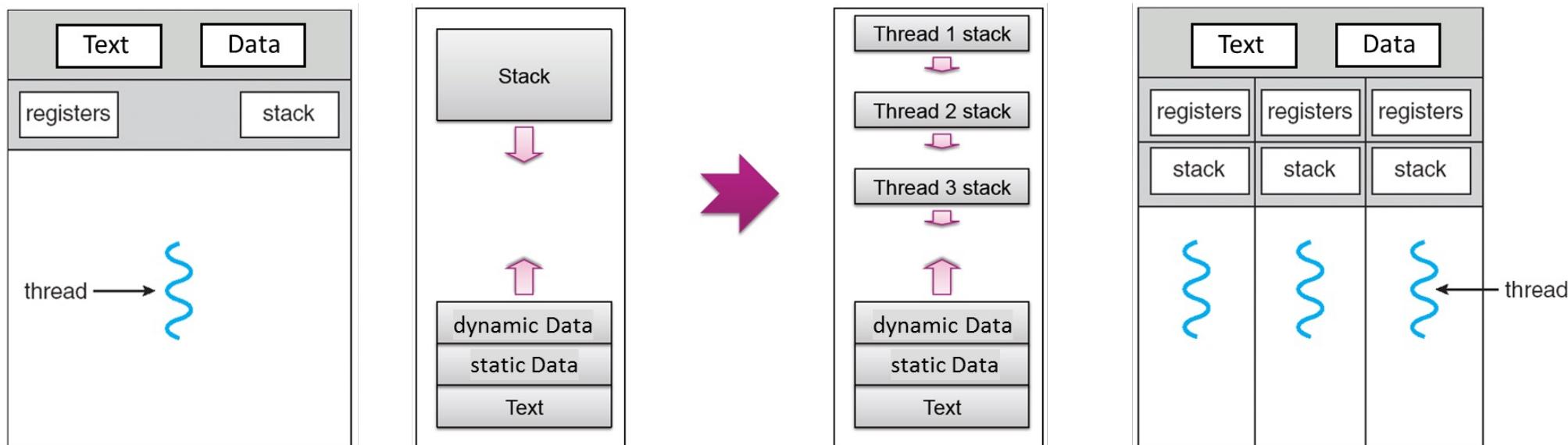
Protection mechanisms are not always necessary in embedded operating systems:

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

Threads

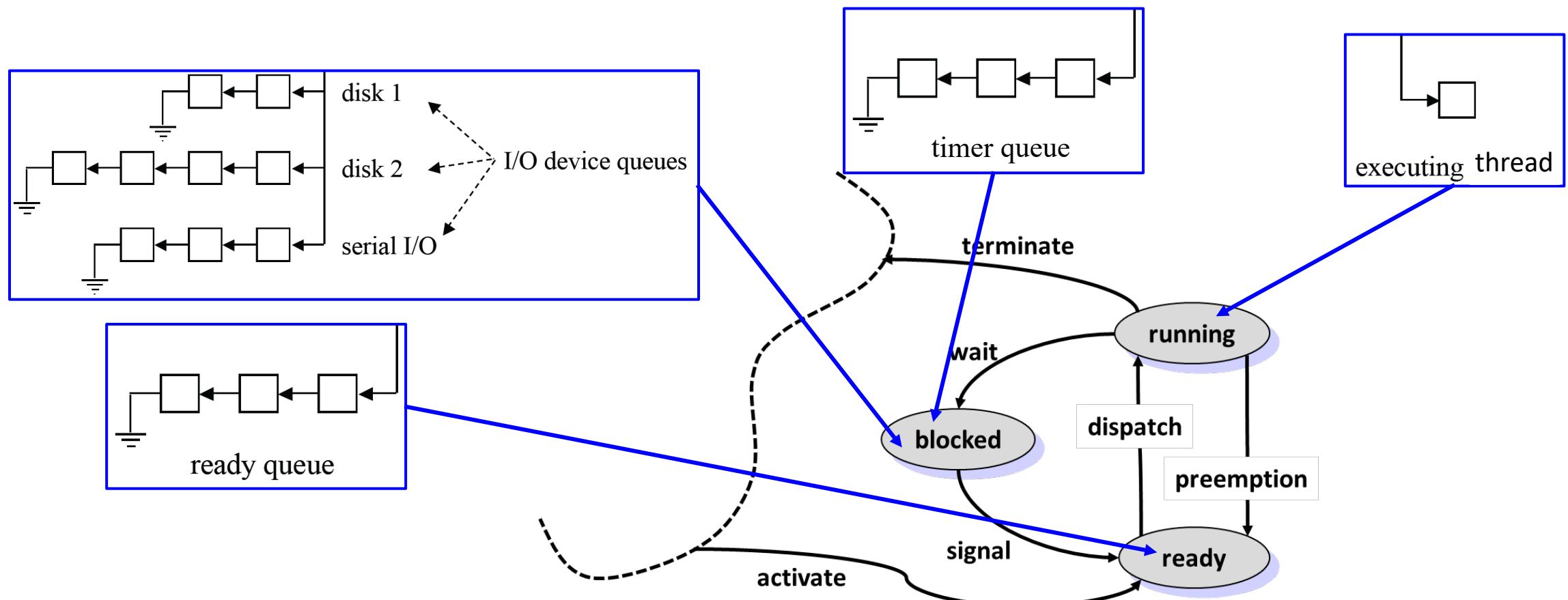
A thread is the smallest sequence of program instructions that can be managed independently by a scheduler. Thus, a thread is a basic unit of CPU utilization.

- *Multiple threads can exist within the same process* and share resources such as memory, while different processes do not share these resources:
 - Typically shared across different threads: memory.
 - Typically owned by each individual thread: registers and stack.

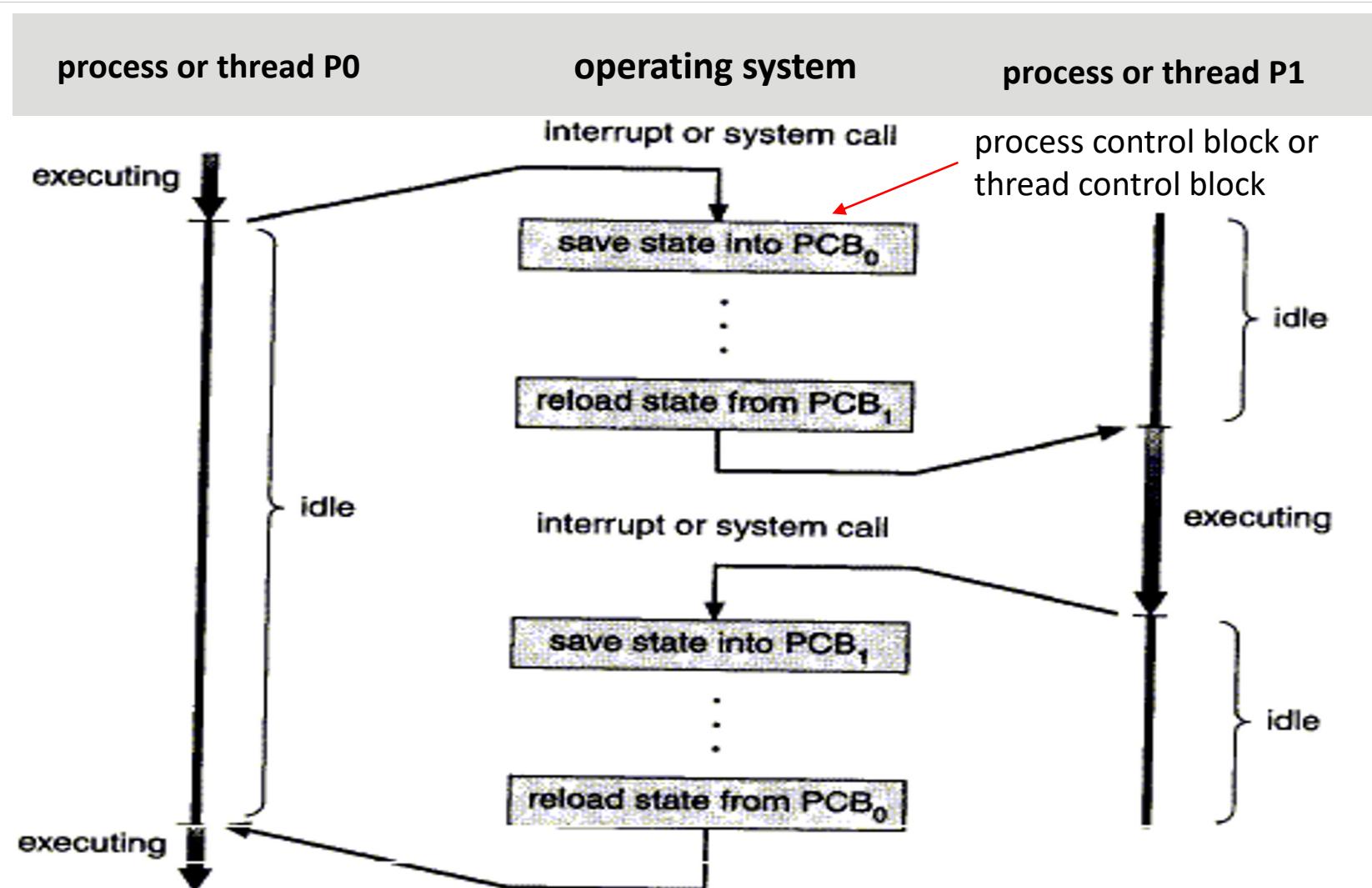


Thread Control Blocks (TCBs)

- The operating system manages TCBs using linked lists. Below is an example.



Context Switch: Processes or Threads



Embedded Operating Systems

Concepts in Action: FreeRTOS

Example: FreeRTOS

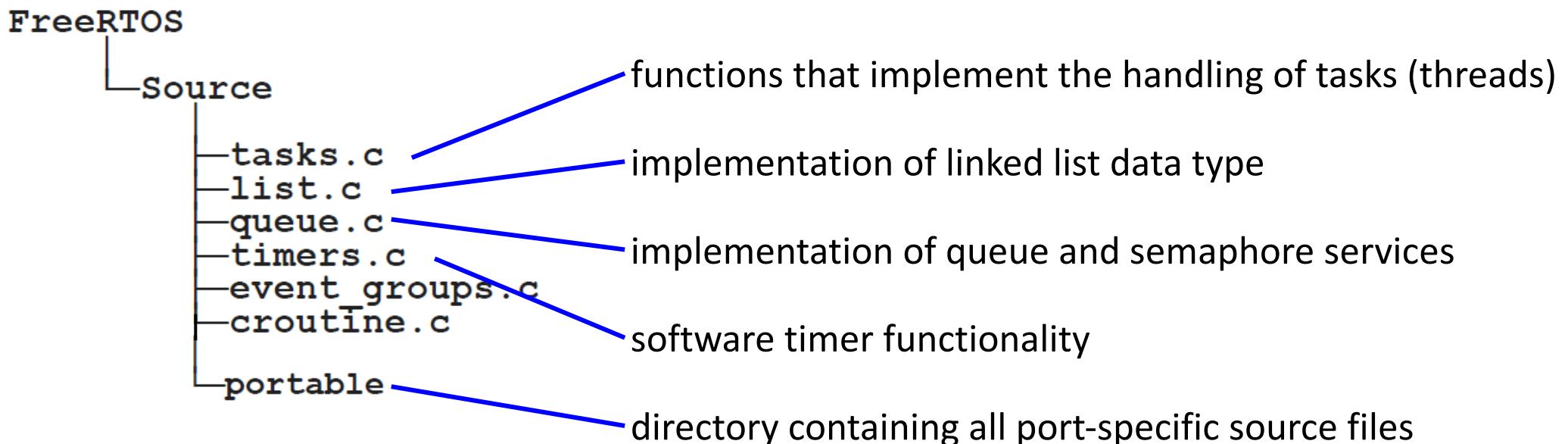
FreeRTOS (<http://www.freertos.org/>) is a typical embedded OS. It is open source, available for many hardware platforms, and widely used in industry.

- FreeRTOS is a *real-time kernel* (or real-time scheduler), but not suited for hard real-time applications (e.g., there are no guarantees on the execution time of OS services).
 - Applications are organized as a *collection of independent threads* of execution, called tasks.
 - *Supports many important concepts*, for instance:
 - cooperative and preemptive multitasking
 - queues, software timers, etc.
 - binary and counting semaphores, mutexes (mutual exclusion)
 - stack overflow checking
 - recording of execution traces



Example: FreeRTOS

Typical directory structure (excerpts):



- *FreeRTOS is configured* by a header file called `FreeRTOSConfig.h` that determines almost all configurations (cooperative versus 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*:

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

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.

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)

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.

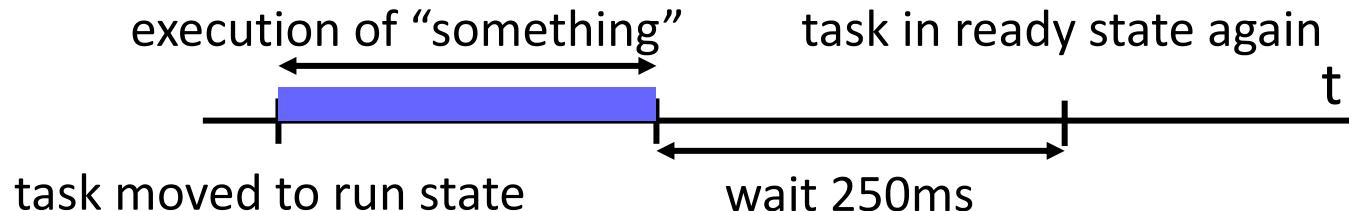
```
void vTaskDelay( TickType_t xTicksToDelay );
```

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().

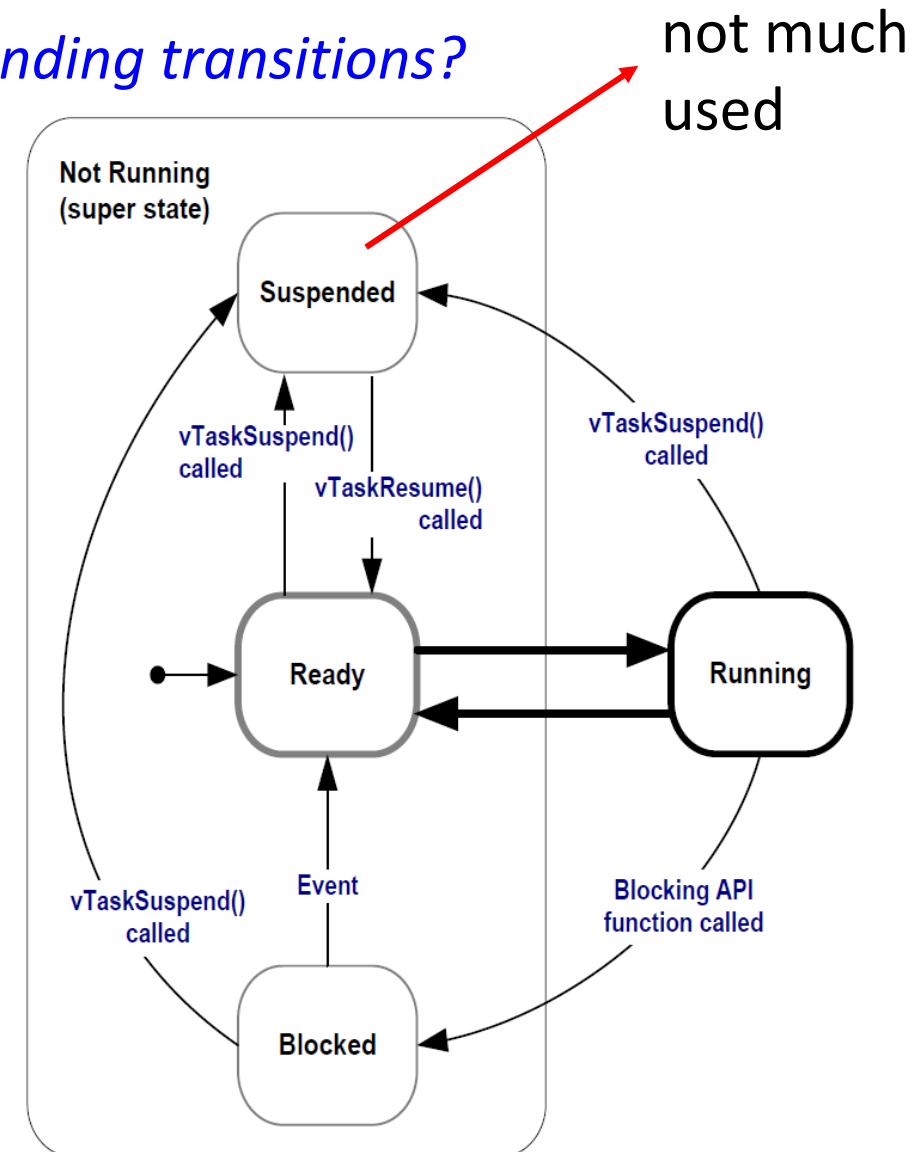
Embedded Operating Systems

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.

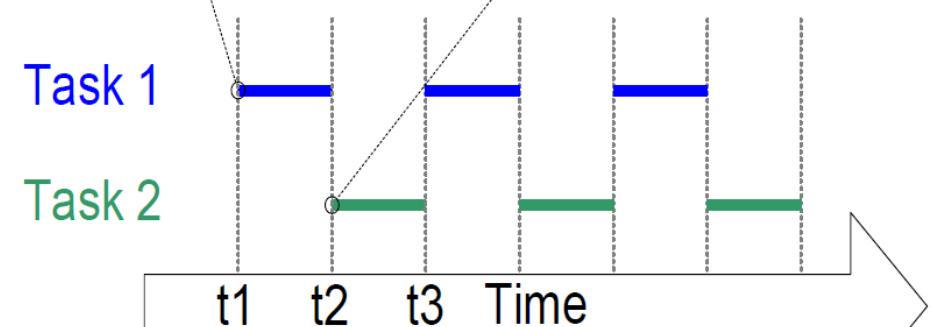
```
void vTask1( void *pvParameters ) {  
    volatile uint32_t ul;  
    for( ;; ) {  
        ... /* do something repeatedly */  
        for( ul = 0; ul < 10000; ul++ ) { }  
    }  
}
```

```
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.

At time t1, Task 1 enters the Running state and executes until time t2
At time t2 Task 2 enters the Running state and executes until time t3 - at which point Task1 re-enters the Running state



Example FreeRTOS – Task States

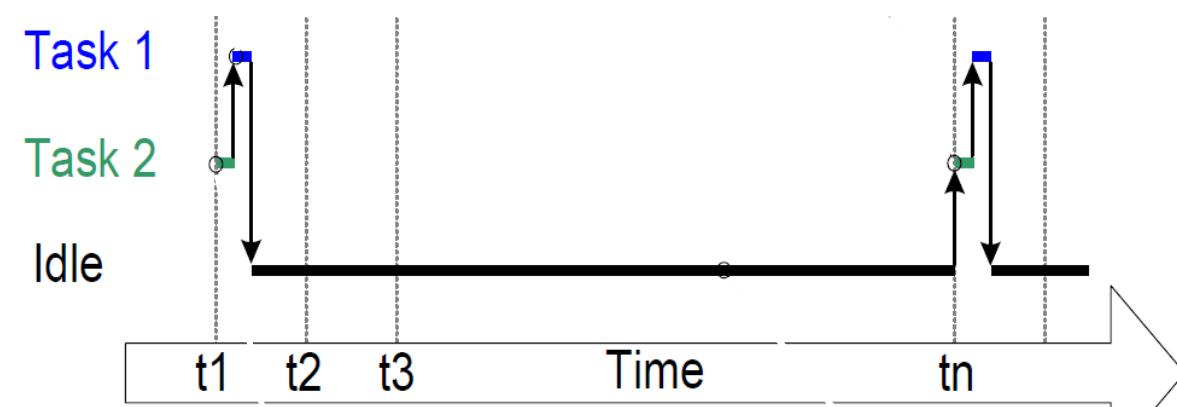
Example 2: Two threads with delay timer.

```
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));  
    }  
}
```

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.

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



Embedded Operating Systems

FreeRTOS Interrupts

Example FreeRTOS – Interrupts

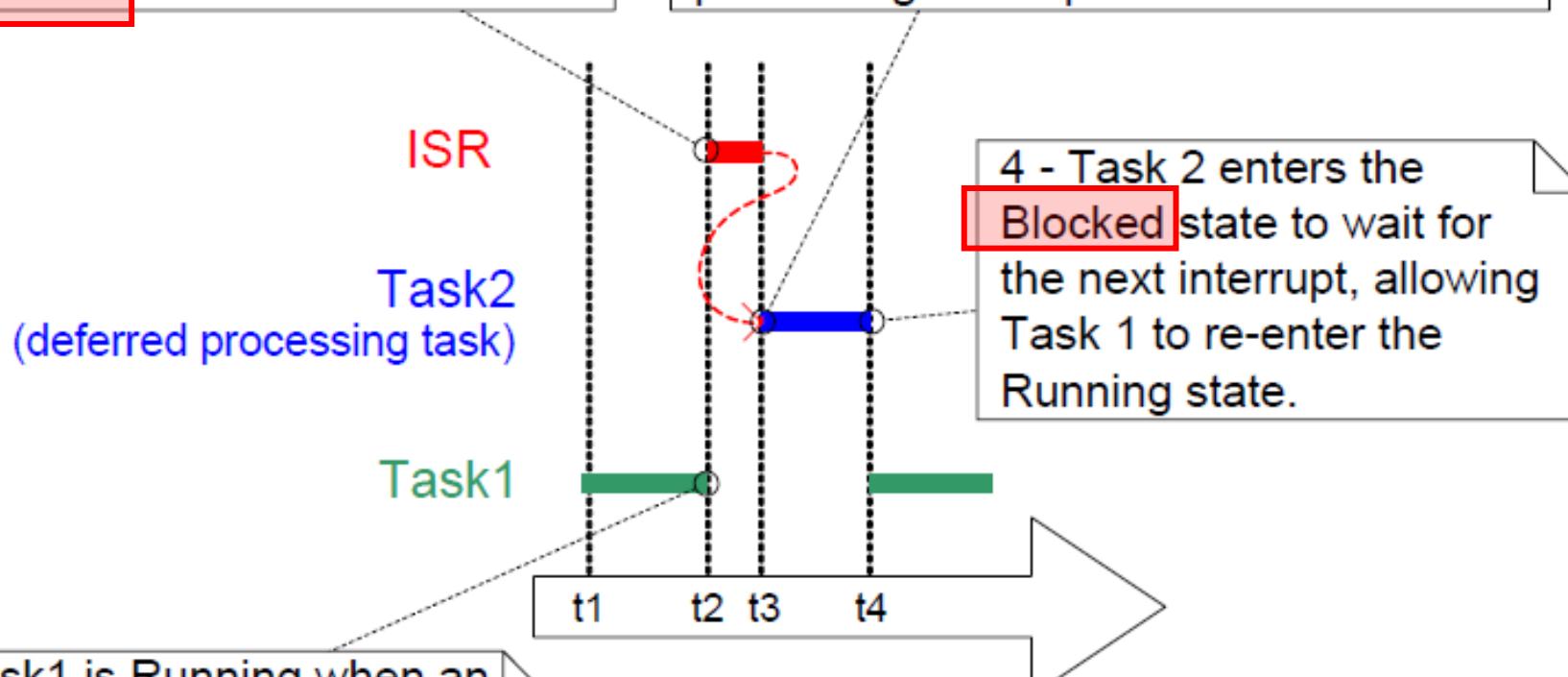
How are tasks (threads) and hardware interrupts scheduled jointly?

- 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

2 - The ISR executes, handles the interrupting peripheral, clears the interrupt, then **unblocks** Task 2.

3 - The priority of Task 2 is higher than the priority of Task 1, so the ISR returns directly to Task 2, in which the interrupt processing is completed.

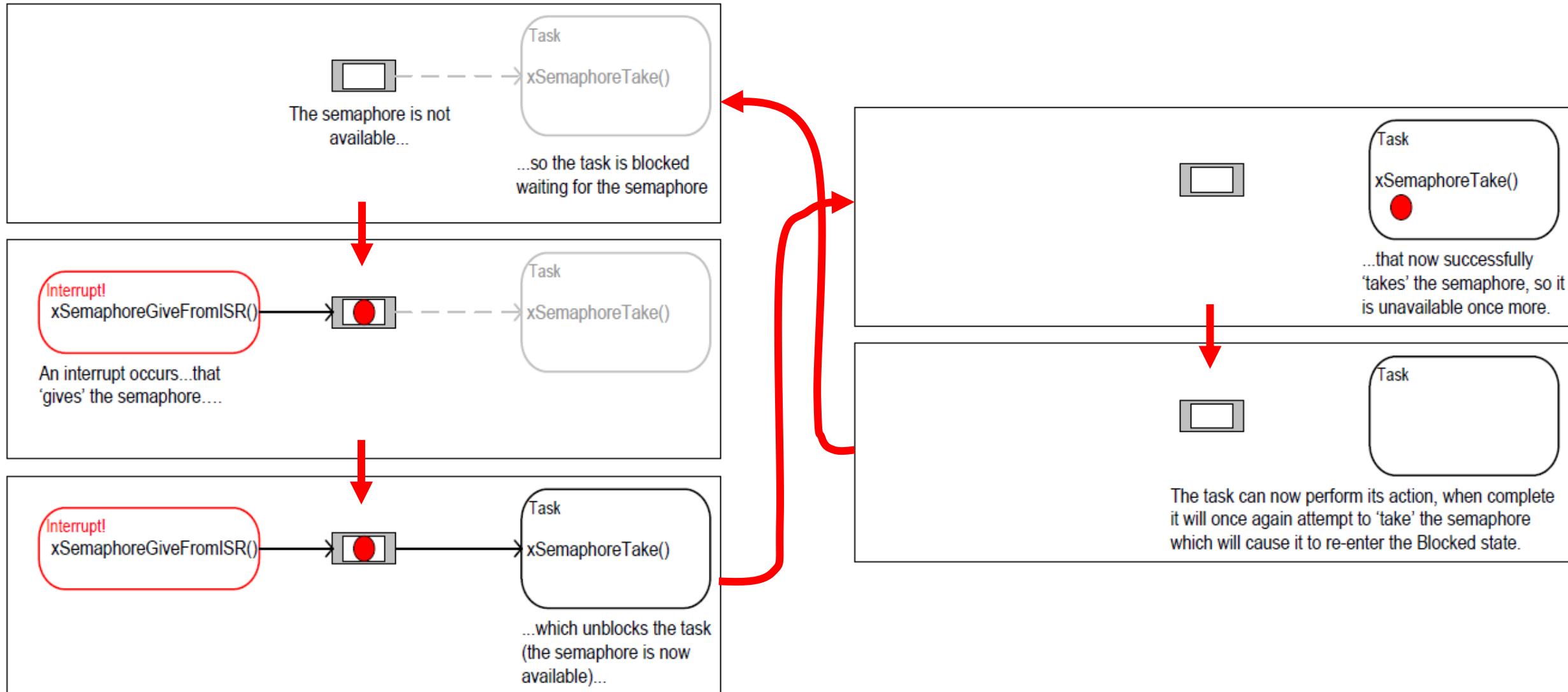


1 - Task1 is Running when an interrupt occurs.

4 - Task 2 enters the **Blocked** state to wait for the next interrupt, allowing Task 1 to re-enter the Running state.

blocking and unblocking is typically implemented via semaphores

Example FreeRTOS – Interrupts



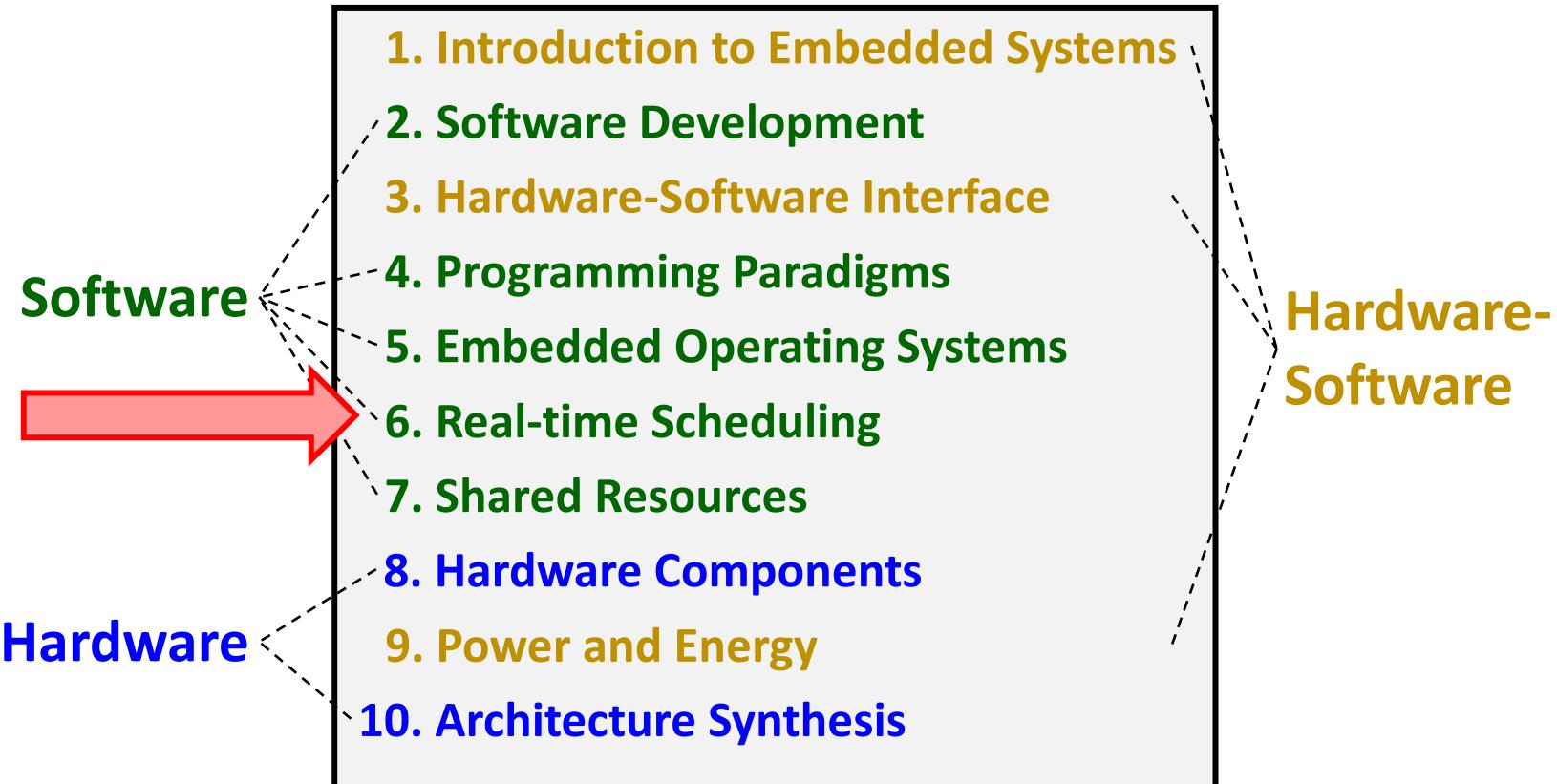
Introduction to Embedded Systems

6. Real-Time Scheduling

Prof. Dr. Marco Zimmerling



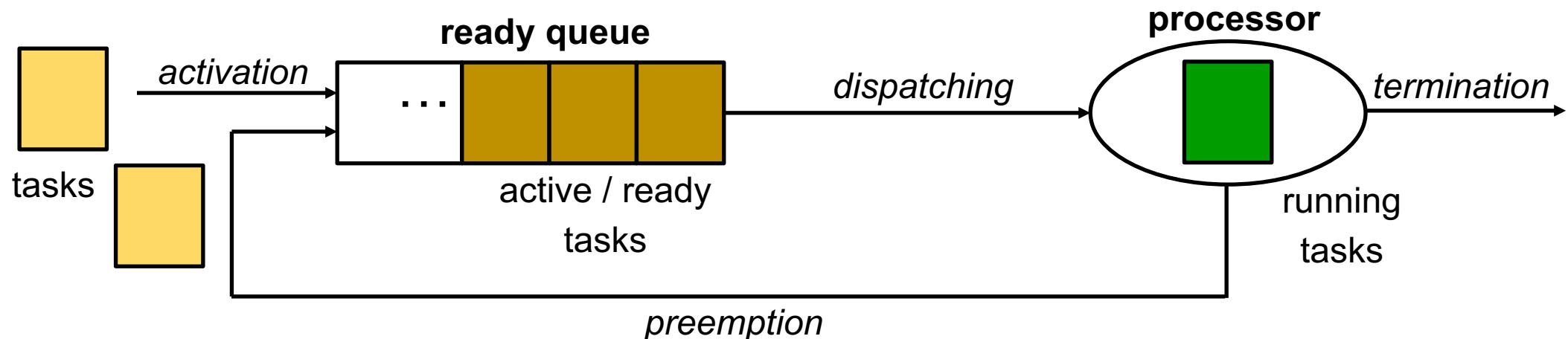
Where we are ...



Terminology and Models

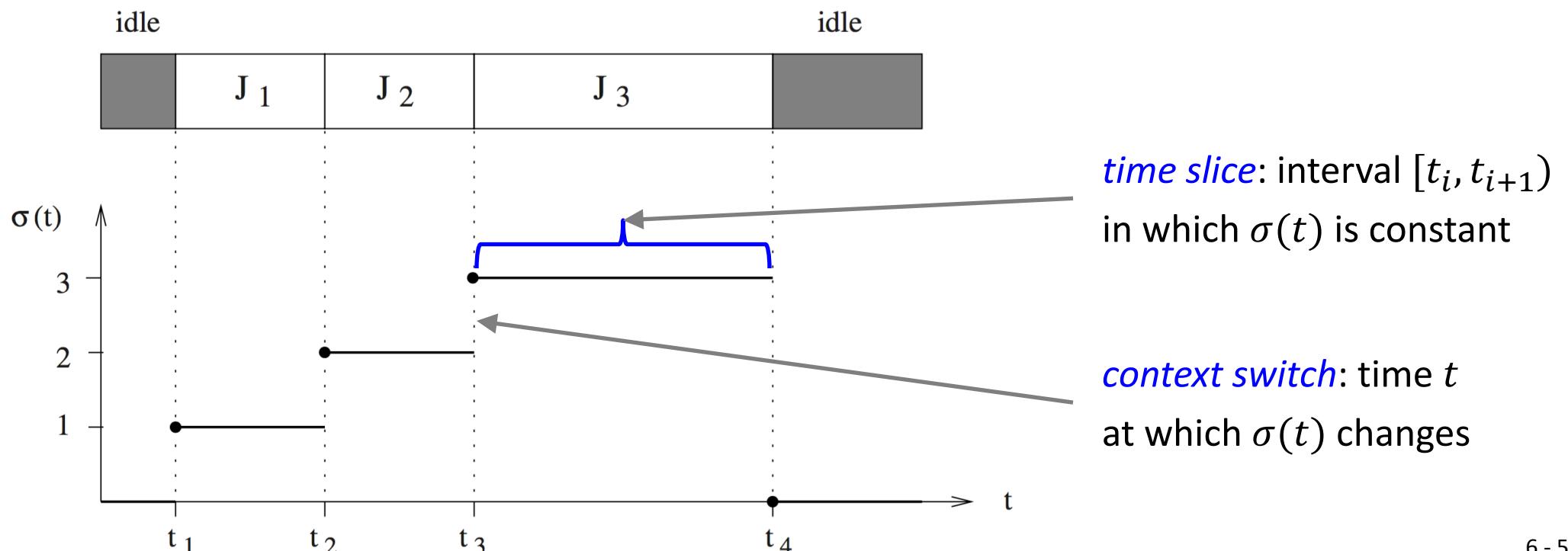
Basic Scheduling Concepts

- A *task* is a computation that is executed by the processor in a sequential fashion.
- A *scheduling algorithm* determines the order in which tasks that can overlap in time are executed on the processor.
- The operation of suspending the running task and inserting it into the ready queue is called *preemption*.



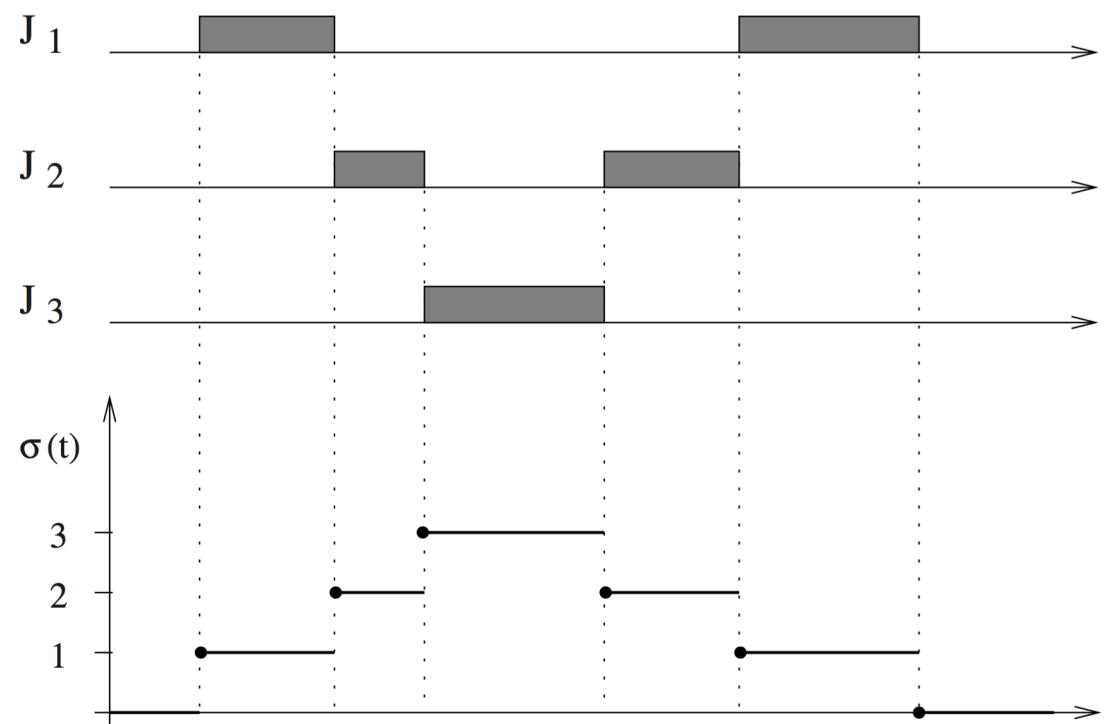
Definition of Schedule

- Given a set of tasks, $J = \{J_1, \dots, J_n\}$, a *schedule* is an assignment of tasks to the processor, so that each task is executed until completion.
- Formally, a schedule is an integer step function $\sigma : \mathbb{R}^+ \rightarrow \mathbb{N}$, where
 - $\sigma(t) = k$ means that the processor *executes* task J_k at time t , and
 - $\sigma(t) = 0$ means that the processor is *idle* at time t .



Important Attributes

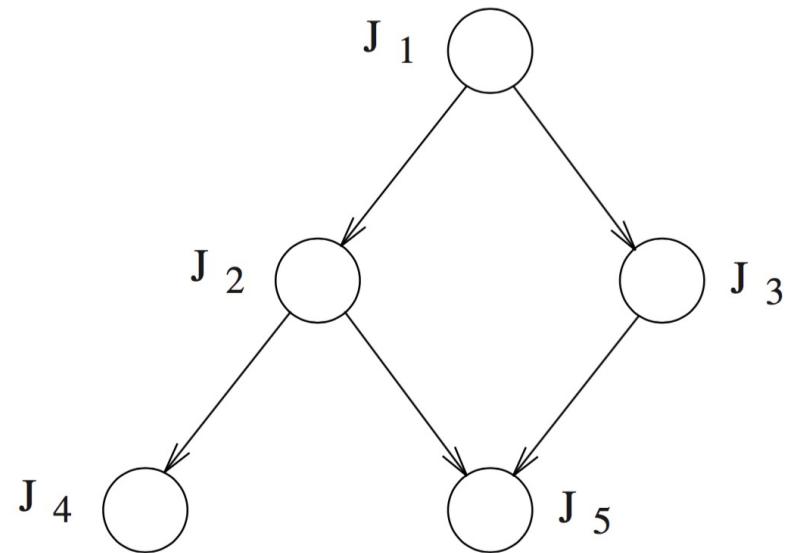
- A *preemptive* schedule is a schedule in which the running task can be arbitrarily suspended at any time to assign the processor to another task.
- A schedule is said to be *feasible* if all tasks can be completed according to a set of specified constraints.
- A set of tasks is said to be *schedulable* if there exists at least one scheduling algorithm that can produce a feasible schedule.



Example of a *preemptive* schedule

Types of Task Constraints

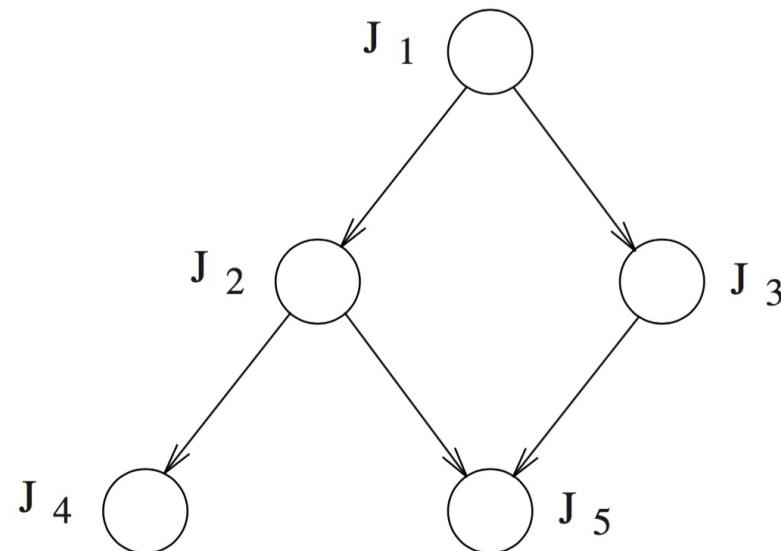
- *Timing constraints*: tasks need to complete before given deadlines
- *Precedence constraints*: tasks need to execute in a given order



- *Resource constraints*: tasks need to execute on given resources (e.g., data structure, piece of a program, memory area, peripheral device)

Types of Task Constraints

- *Timing constraints*: tasks need to complete before given deadlines
- *Precedence constraints*: tasks need to execute in a given order



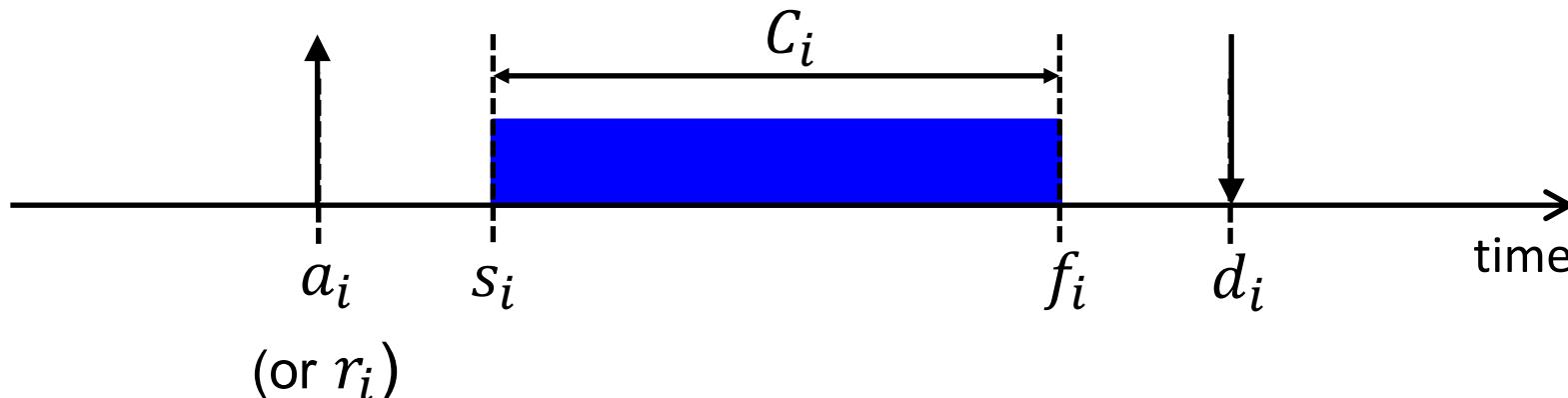
- *Resource constraints*: tasks need to execute on given resources (e.g., data structure, piece of a program, memory area, peripheral device)

Tasks with Timing Constraints

- A typical timing constraint on a task is a *deadline*, representing the time before which the task should complete its execution.
- A task is called a *real-time task* if it is subject to a specified deadline.
- Depending on the consequences of a missed deadline, real-time tasks can be classified into two main categories:
 - A real-time task is said to be *hard* if missing its deadline may cause catastrophic consequences on the system under control. Examples include sensing, actuation, and control tasks in a safety-critical system (e.g., airplane, car, power plant, pace makers).
 - A real-time task is said to be *soft* if missing its deadline does not cause any serious damage and has still some utility for the system; that is, a deadline miss is considered a performance issue, not an issue of correct behavior. Examples are tasks related to user interactions on a smartphone or to convenience functions in a car (e.g., seat warmer).

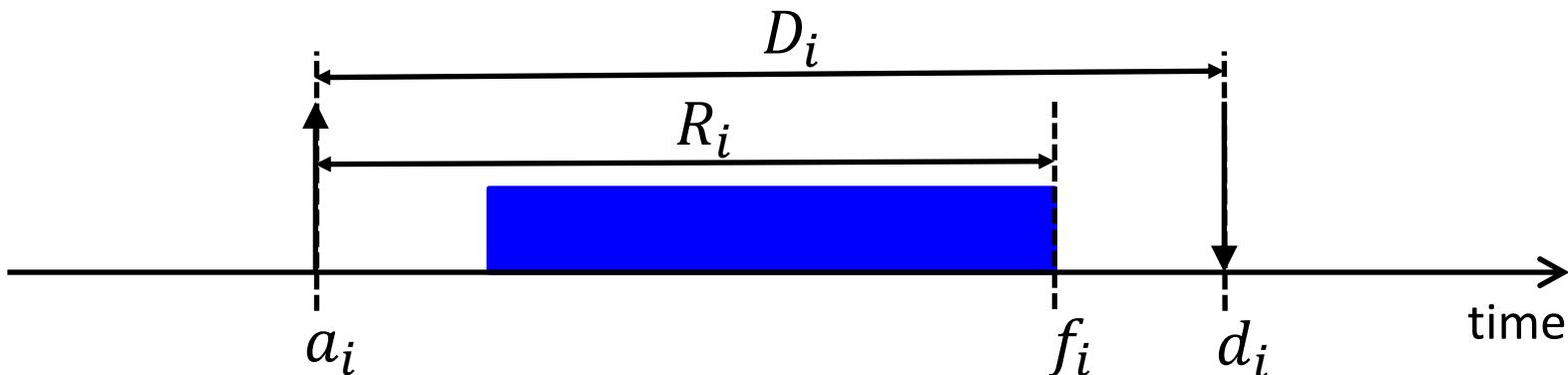
Parameters of Real-Time Tasks (1)

- *Arrival time* a_i or *release time* r_i is the time at which a task becomes ready for execution.
- *Start time* s_i is the time at which a task starts its execution.
- *Execution time* C_i is the time needed by the processor to execute a task without interruption.
- *Finishing time* f_i is the time at which a task finishes its execution.
- *Absolute deadline* d_i is the time by which a task should be completed.

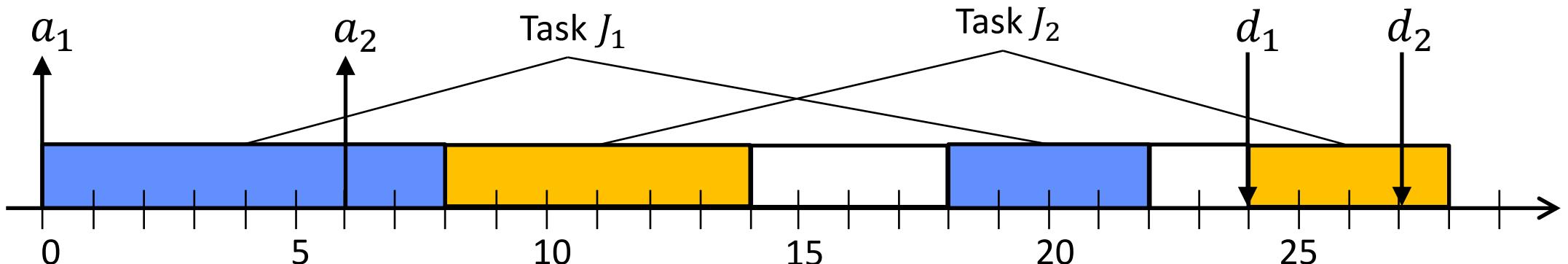


Parameters of Real-Time Tasks (2)

- *Relative deadline* D_i is the difference between the absolute deadline and the arrival time of a task, that is, $D_i = d_i - a_i$.
- *Response time* R_i is the difference between the finishing time and the arrival time of a task, that is, $R_i = f_i - a_i$.
- *Lateness* $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline, that is, $L_i \leq 0$ if a task completes within its deadline.
- *Tardiness* $E_i = \max(0, L_i)$ is the time a task stays active after its deadline.
- *Laxity* $X_i = d_i - a_i - C_i$ is the maximum time a task can be delayed on its execution in order to complete within its deadline.



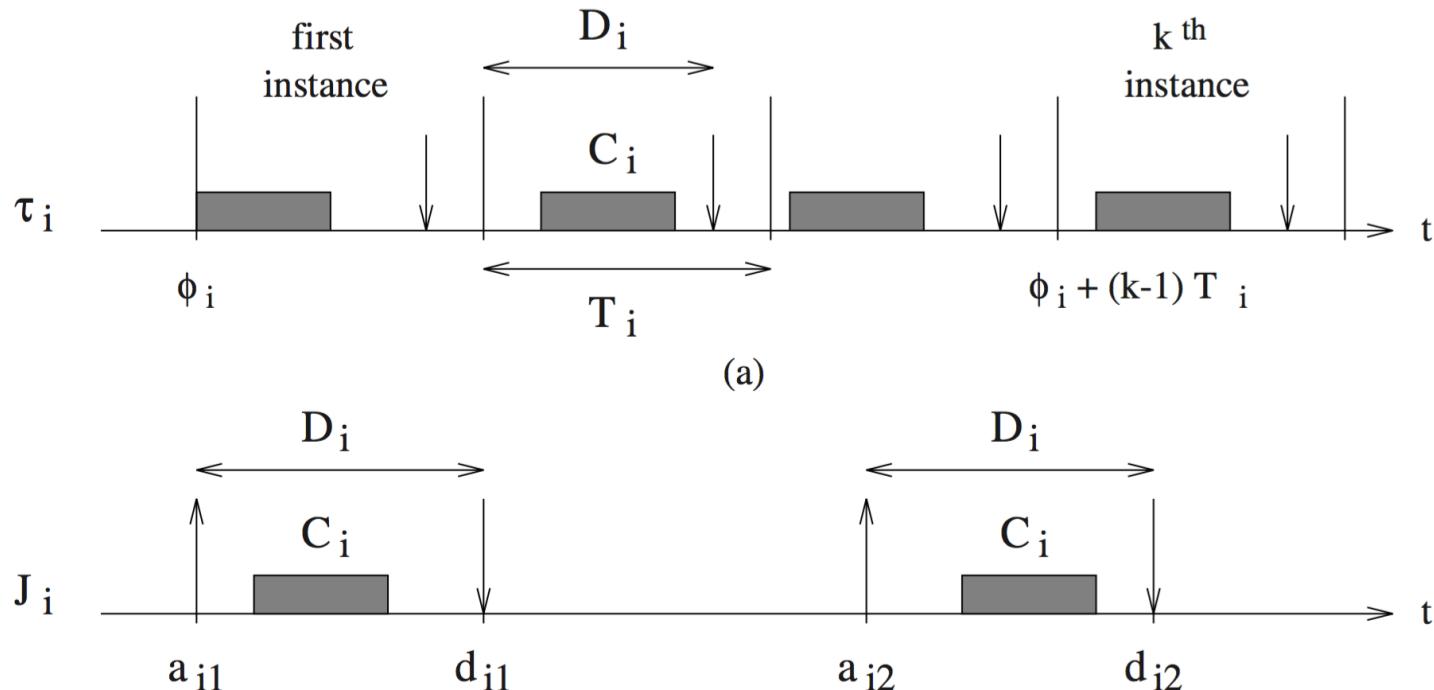
Example: Parameters of Real-time Tasks



- Execution times: $C_1 = 12, C_2 = 10$
- Start times: $s_1 = 0, s_2 = 8$
- Finishing times: $f_1 = 22, f_2 = 28$
- Lateness: $L_1 = -2, L_2 = 1$
- Tardiness: $E_1 = 0, E_2 = 1$
- Laxity: $X_1 = 12, X_2 = 11$

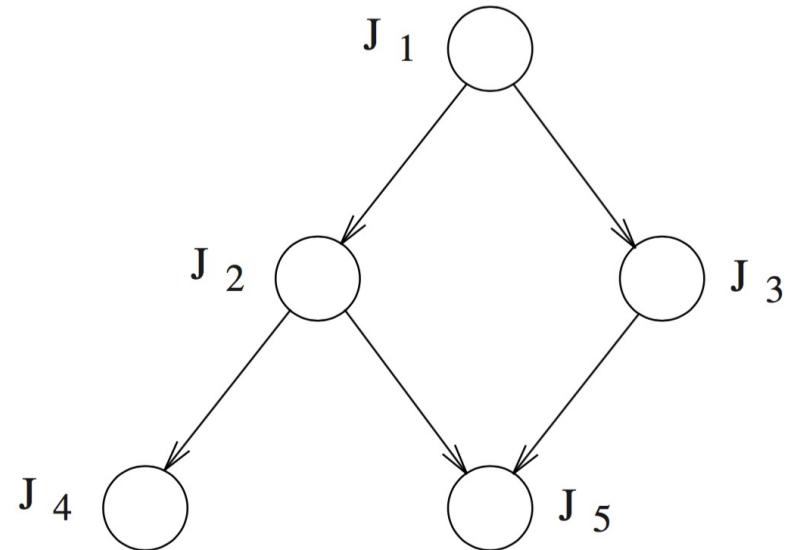
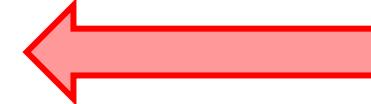
Periodic and Aperiodic Tasks

- A *periodic task* τ_i consists of an infinite sequence of identical activities, called instances or jobs, that are regularly activated with a constant *period* T_i . The arrival time of the first instance is called *phase* Φ_i .
- We use τ_i to denote a periodic task and J_i to denote an aperiodic task. An aperiodic task J_i can arrive (or can be released) at *any point in time*.



Types of Task Constraints

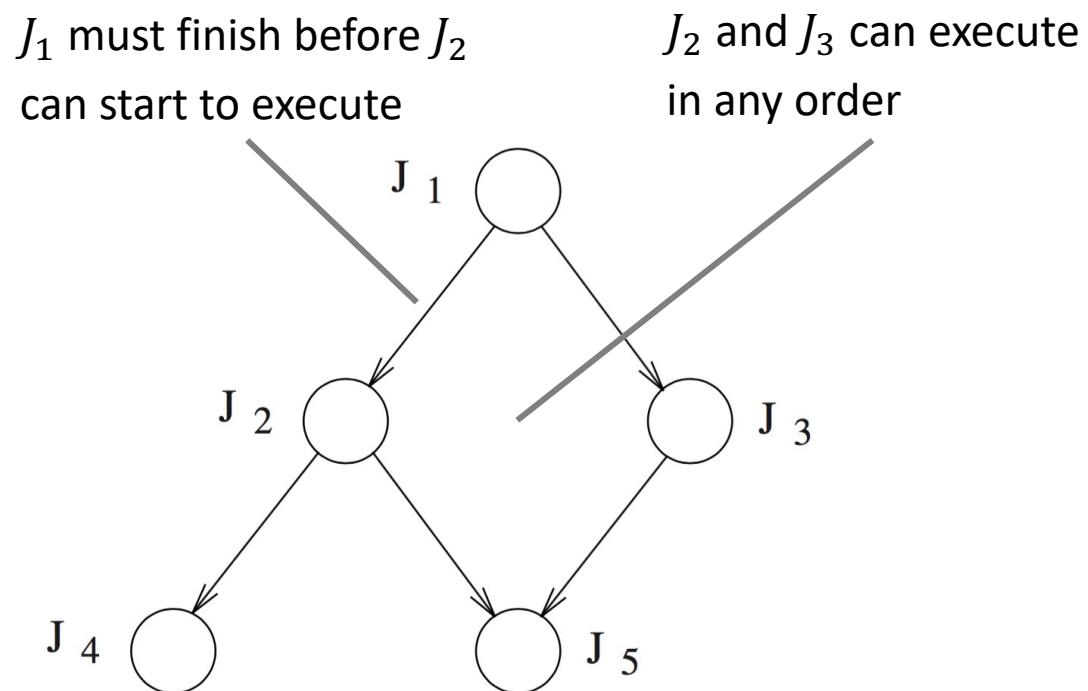
- *Timing constraints*: tasks need to complete before given deadlines
- *Precedence constraints*: tasks need to execute in a given order



- *Resource constraints*: tasks need to execute on given resources (e.g., data structure, piece of a program, memory area, peripheral device)

Tasks with Precedence Constraints

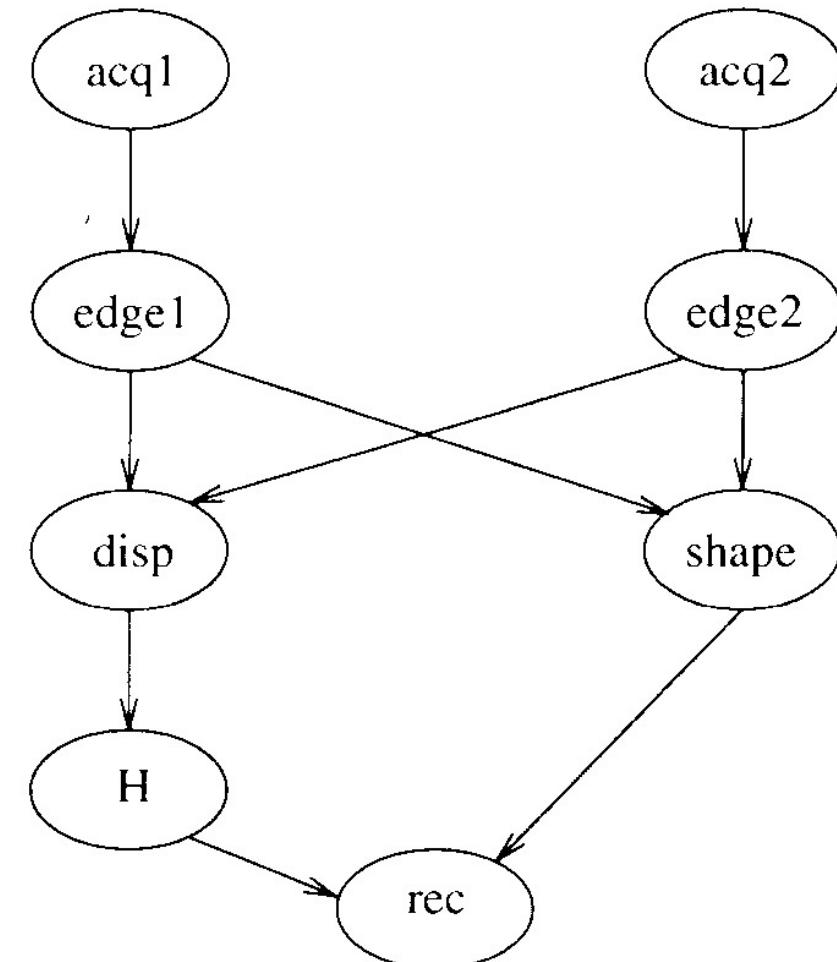
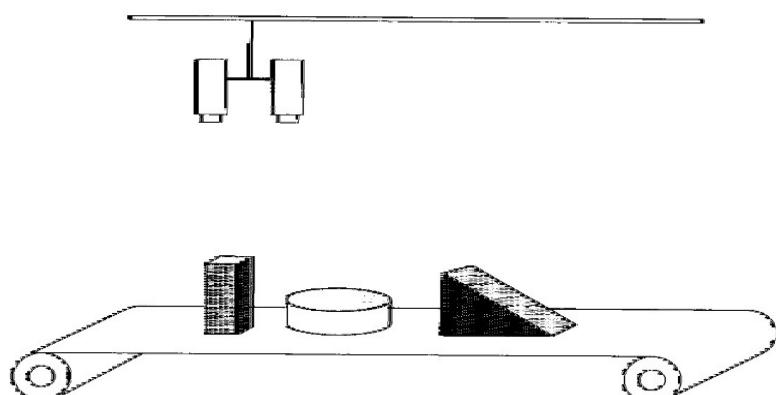
- Precedence constraints between tasks can be described through a *directed acyclic graph (or DAG)* G , where tasks are represented by nodes and precedence constraints by arrows. G induces a partial order on the task set.
- Different interpretations are possible:
 - *Concurrent task execution*: all successors of a task are activated. We will use this interpretation in the lecture.
 - *Non-deterministic choice*: one successor of a task is activated.



Example: Concurrent Activation

Object recognition with two cameras:

- 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 (1)

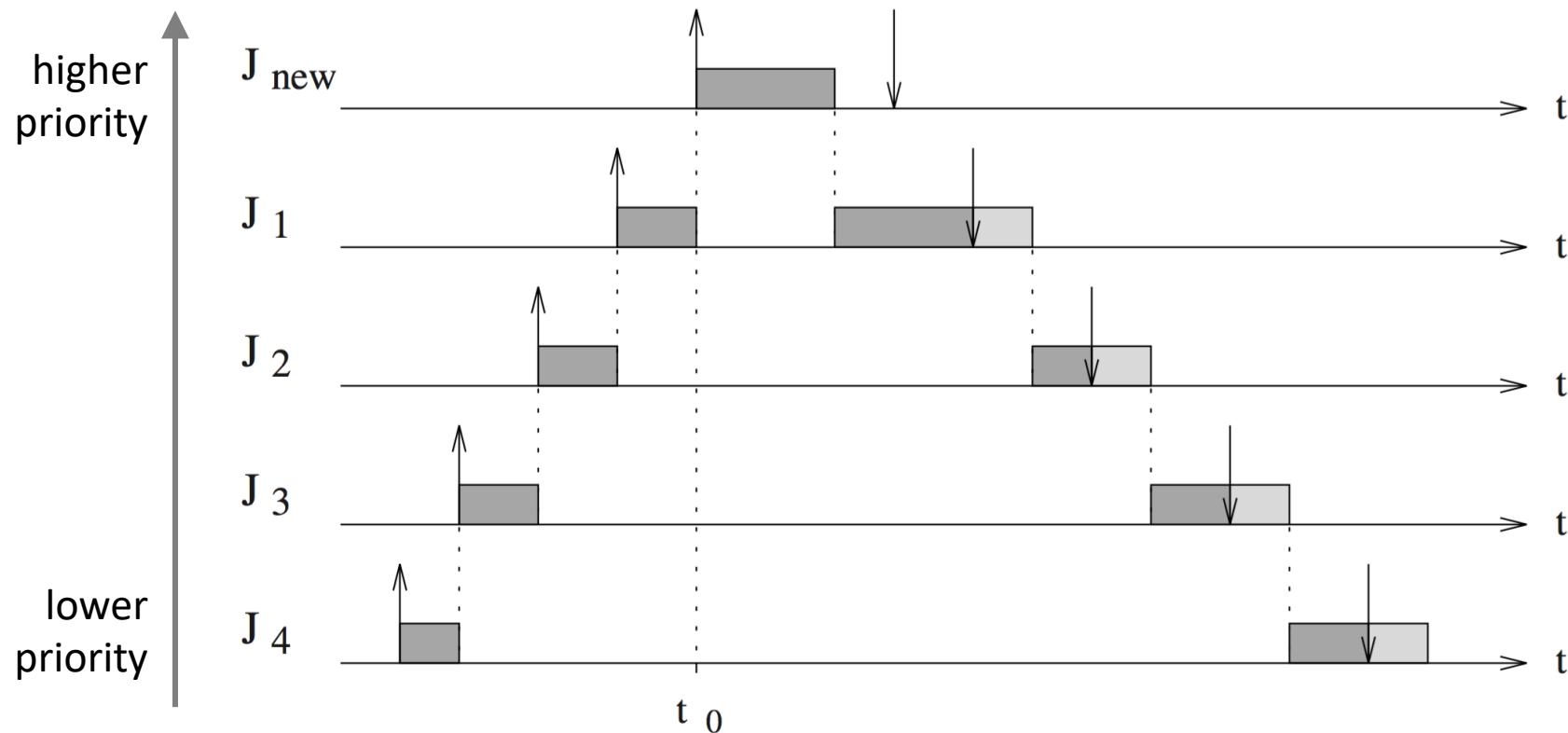
- Using a *preemptive* algorithm, the running task can be interrupted at any time to assign the processor to another active task.
- Using a *non-preemptive* algorithm, a task, once started, is executed by the processor until completion. Interrupts, which are typically very short, are still allowed and can preempt tasks. A task, however, cannot preempt another task.
- *Static* algorithms are those in which scheduling decisions are based on fixed parameters that are assigned to tasks before their activation. For example, this includes all time-triggered algorithms from Chapter 4 on programming paradigms.
- *Dynamic* algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system operation. For example, this includes all event-triggered algorithms from Chapter 4 on programming paradigms.

Classification of Scheduling Algorithms (2)

- An algorithm is used *offline* if it is executed on the entire task set before any task activation. The generated schedule can be stored in a table and then executed at runtime by a dispatcher.
- An algorithm is used *online* if scheduling decisions are taken at runtime every time a new task enters the system or when a running task terminates.
- An algorithm is said to be *optimal* if it minimizes a given cost function defined over the task set.
- An algorithm is said to be *heuristic* if it tends toward the optimal schedule, but does not guarantee finding it.

Schedulability Analysis

- In hard real-time applications, feasibility of the schedule should be guaranteed in advance (*i.e.*, before task execution).
 - Can be checked offline if task set is fixed and known a priori.
 - Must be checked online if tasks can be created at runtime (*acceptance test*).



Domino effect: if task J_{new} were accepted at time t_0 , all other (previously schedulable) tasks would miss their deadline.

Metrics to Evaluate Schedules

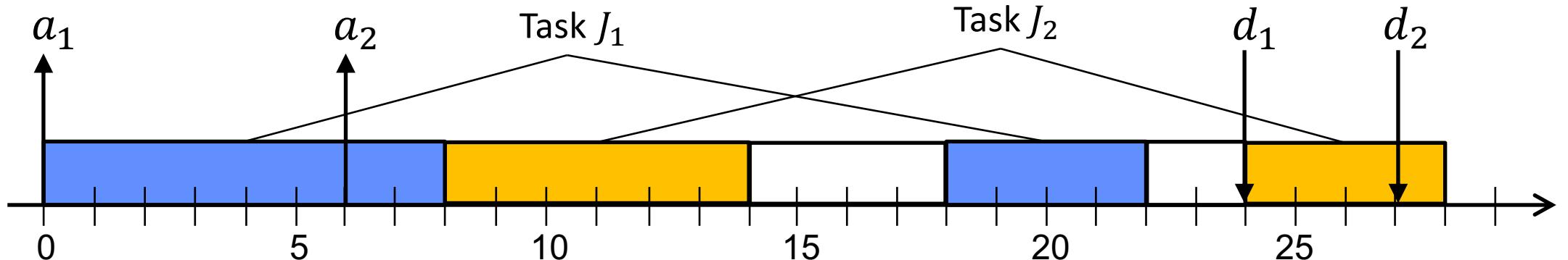
- Average response time: $\bar{t}_r = \frac{1}{n} \sum_{i=1}^n (f_i - a_i)$
- Total completion time: $t_c = \max_i(f_i) - \min_i(a_i)$
- Weighted sum of response times: $t_w = \frac{\sum_{i=1}^n w_i(f_i - a_i)}{\sum_{i=1}^n w_i}$
- Maximum lateness: $L_{max} = \max_i(f_i - d_i)$
- Number of late tasks: $N_{late} = \sum_{i=1}^n miss(f_i)$

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

Metrics to Evaluate Schedules

- Average response time: $\bar{t}_r = \frac{1}{n} \sum_{i=1}^n (f_i - a_i)$
 - Total completion time: $t_c = \max_i(f_i) - \min_i(a_i)$
 - Weighted sum of response times: $t_w = \frac{\sum_{i=1}^n w_i(f_i - a_i)}{\sum_{i=1}^n w_i}$
 - Maximum lateness: $L_{max} = \max_i(f_i - d_i)$
 - Number of late tasks: $N_{late} = \sum_{i=1}^n miss(f_i)$
- where $miss(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$
- }
- Only these metrics are useful to evaluate real-time schedules, as they involve task deadlines.

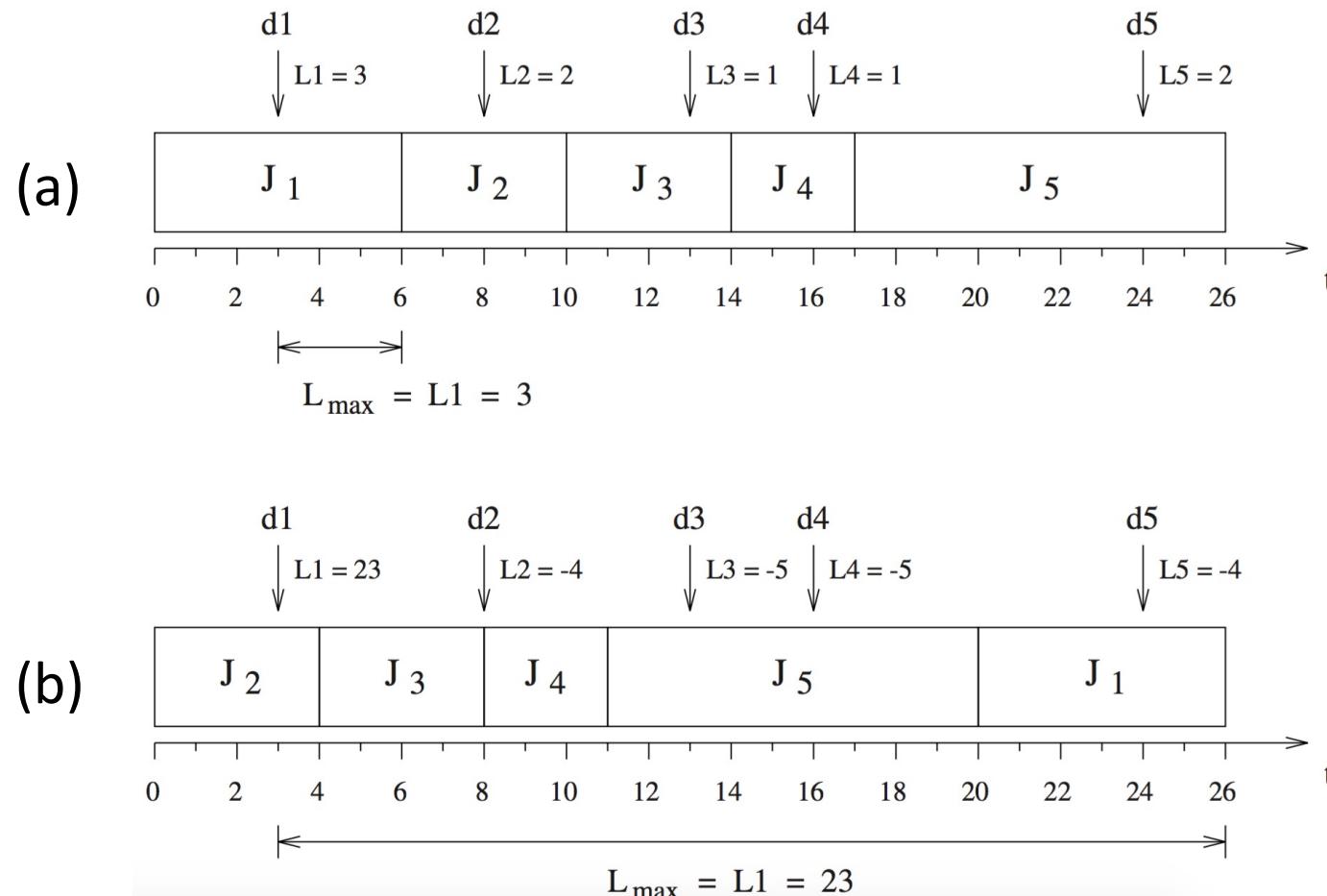
Example: Metrics



- Average response time: $\bar{t}_r = \frac{1}{n} \sum_{i=1}^n (f_i - a_i) = \frac{1}{2} (22 + 22) = 22$
- Total completion time: $t_c = \max_i(f_i) - \min_i(a_i) = 28 - 0 = 28$
- Weighted sum of response times: $t_w = \frac{2 \times 22 + 1 \times 22}{3} = 22$ for $w_1 = 2, w_2 = 1$
- Maximum lateness: $L_{max} = \max_i(f_i - d_i) = \max_2(-2, 1) = 1$
- Number of late tasks: $N_{late} = \sum_{i=1}^n miss(f_i) = 0 + 1 = 1$

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

Example: Maximum Lateness vs. Deadline Misses



- Schedule in (a) *minimizes maximum lateness*, but all tasks miss deadline.
- Schedule in (b) has higher maximum lateness, but *only one deadline miss*.

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 = without precedence constraints)	Independent tasks	EDD (Jackson)	EDF (Horn)
	Dependent tasks	LDF (Lawler)	EDF* (Chetto)

Earliest Deadline Due (EDD)

- Scheduling of aperiodic tasks J_i with *equal arrival times*
 - We assume that all tasks arrive at time $t = 0$ (i.e., $a_i = r_i = 0$ for all tasks J_i).
 - There are no further constraints on the tasks (e.g., no precedence constraints).
 - Thus, each task J_i is fully characterized by execution time C_i and relative deadline D_i .
 - Note: Preemption is not an issue if all tasks arrive at the same time! Hence, EDD is effectively a *non-preemptive* real-time scheduling method.
- *Jackson's rule*: Given a set of n independent tasks, any algorithm that executes the tasks in order of non-decreasing deadline is optimal with respect to minimizing the maximum lateness of the task set.

Example: Feasible Schedule Produced by EDD

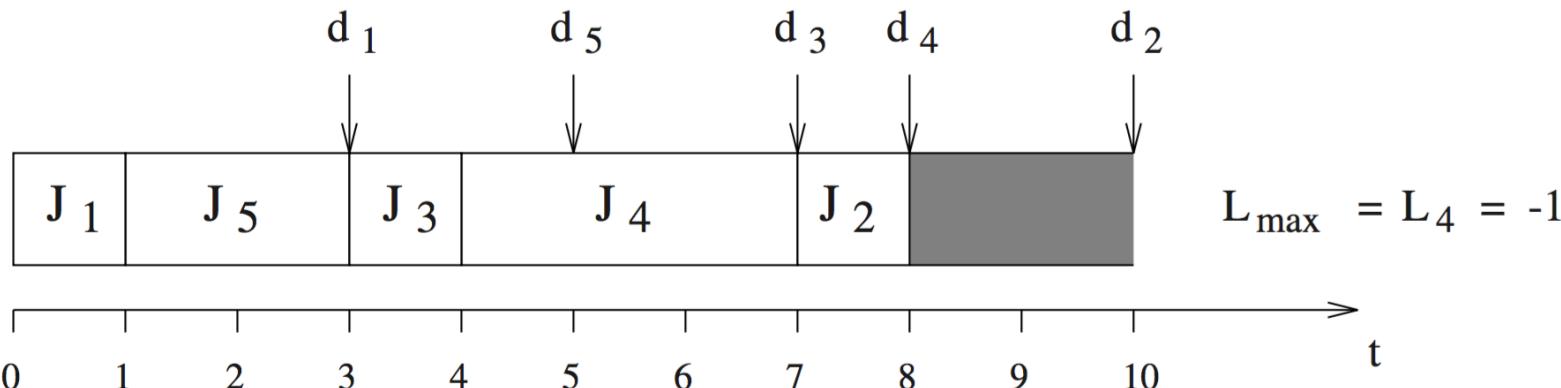
- *Jackson's rule*: Given a set of n independent tasks, any algorithm that executes the tasks in order of non-decreasing deadline is optimal with respect to minimizing the maximum lateness of the task set.

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

Example: Feasible Schedule Produced by EDD

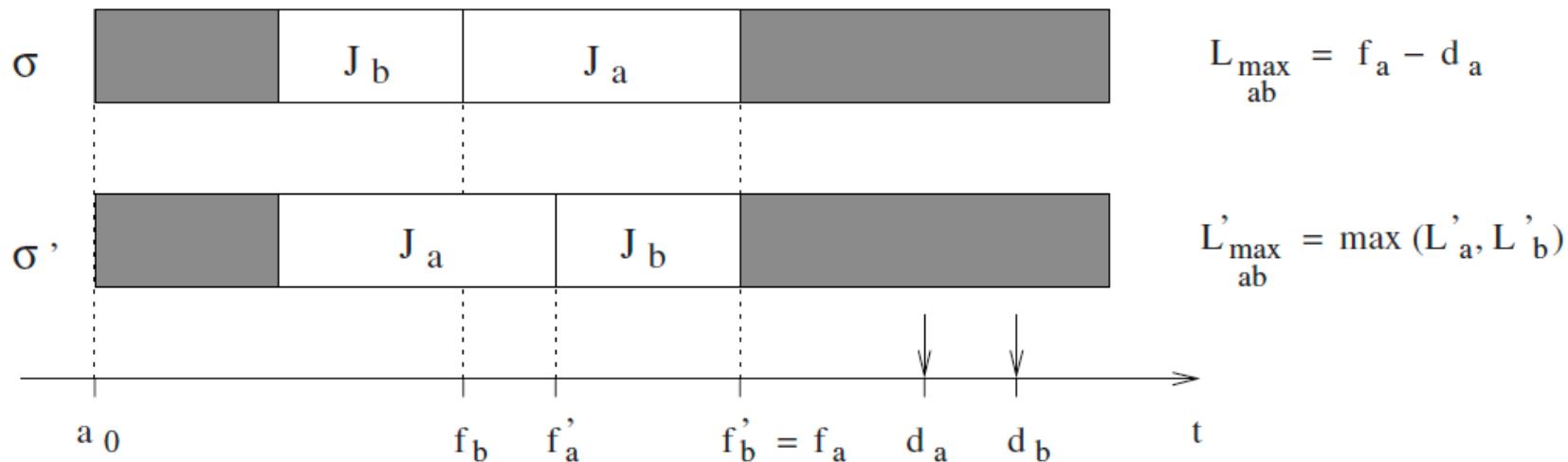
- *Jackson's rule*: Given a set of n independent tasks, any algorithm that executes the tasks in order of non-decreasing deadline is optimal with respect to minimizing the maximum lateness of the task set.

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



Optimality of EDD: Proof Sketch

- Let σ be a schedule produced by an algorithm different from EDD. Then there exist tasks J_a and J_b , with $d_a \leq d_b$, such that J_b immediately precedes J_a in σ .
- Let σ' be a schedule derived from σ where J_a and J_b are exchanged. It can be shown that any such exchange cannot increase the maximum lateness of the task set.



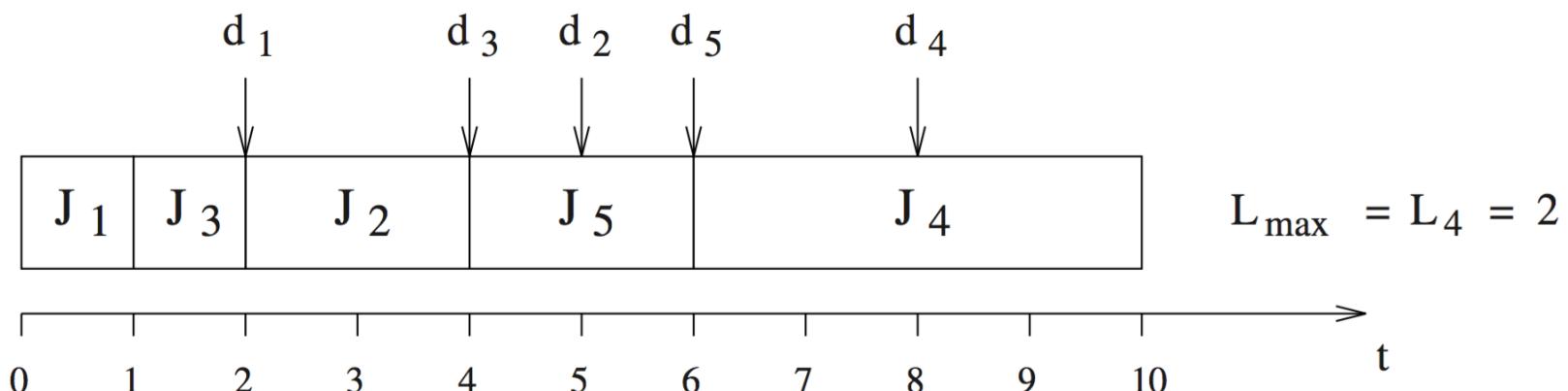
$$\text{if } (L'_a \geq L'_b) \text{ then } L'_{\max ab} = f'_a - d_a < f_a - d_a$$

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

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

Example: Infeasible Schedule Produced by EDD

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



EDD Schedulability Test

- To guarantee that scheduling a set of tasks using EDD produces a feasible schedule, we need to show that in the worst case all tasks can complete before their deadlines, that is, $f_i \leq d_i$ for all tasks J_i .
- If tasks J_1, J_2, \dots, J_n are ordered by increasing deadline, we have

$$f_i = \sum_{k=1}^i C_k$$

- Thus, the EDD schedulability test can be performed (offline) by verifying for each task J_i

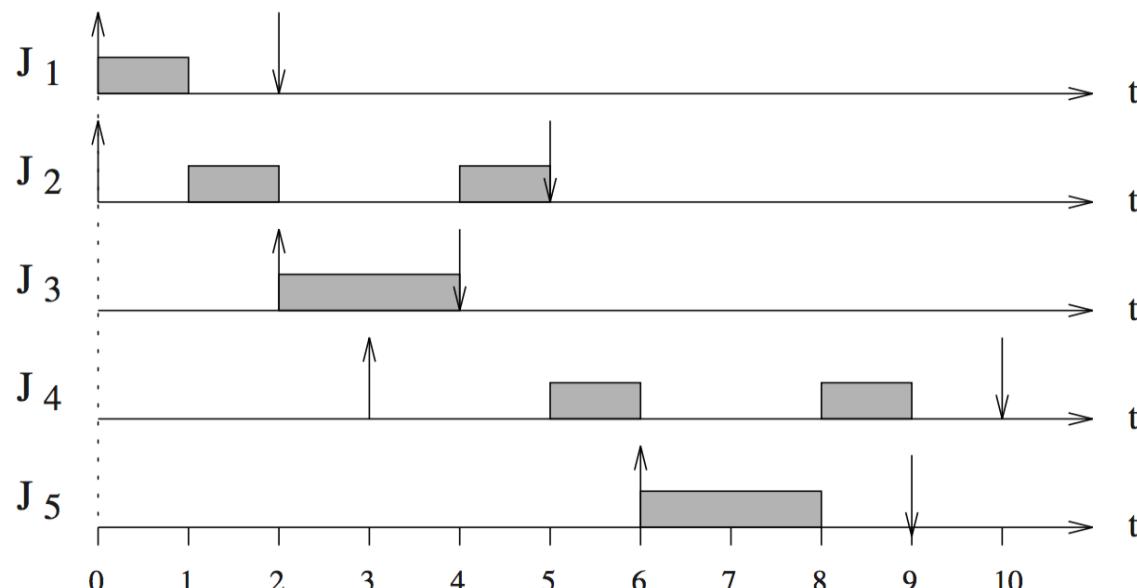
$$\sum_{k=1}^i C_k \leq d_i$$

Earliest Deadline First (EDF)

- Scheduling of aperiodic tasks J_i with *arbitrary arrival times*
 - Tasks J_i arrive dynamically, so preemption is an important factor!
 - There are no further constraints on the tasks (e.g., no precedence constraints).
 - Thus, each task J_i is characterized by its execution time C_i , relative deadline D_i , and two parameters only known at runtime: arrival time a_i and absolute deadline d_i .
- *Horn's rule*: Given a set of n independent tasks with arbitrary arrival times, any algorithm that at any point in time executes the task with the earliest absolute deadline among all the ready tasks is optimal with respect to minimizing the maximum lateness.

Example: Feasible Schedule Produced by EDF

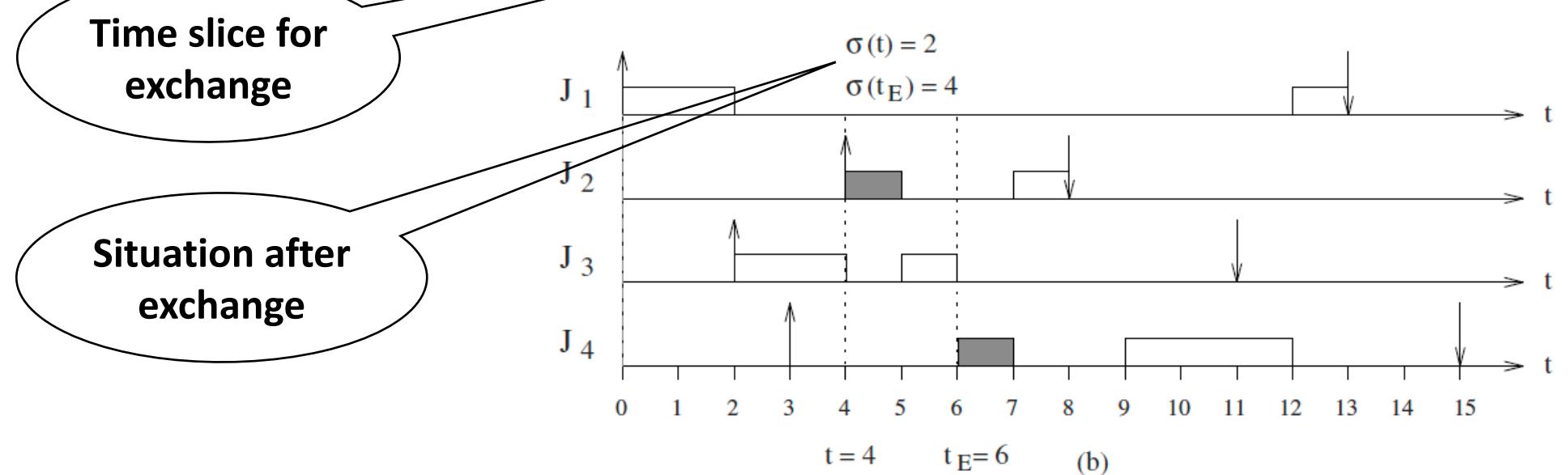
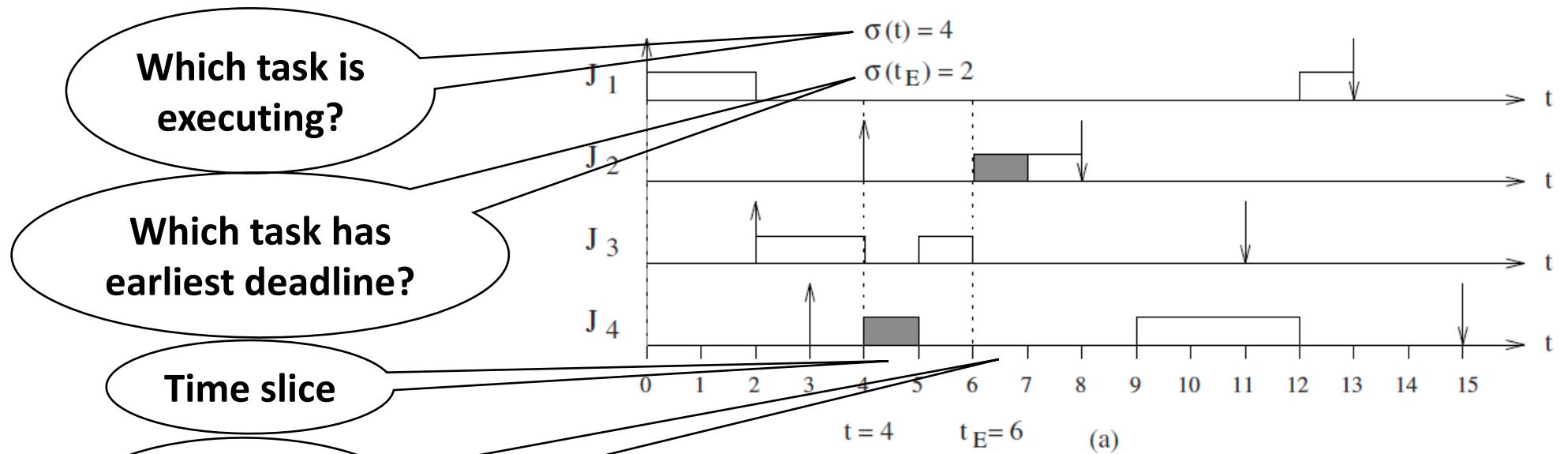
	J ₁	J ₂	J ₃	J ₄	J ₅
a _i	0	0	2	3	6
C _i	1	2	2	2	2
d _i	2	5	4	10	9



Optimality of EDF: Proof Sketch (1)

- The proof is similar to the one for EDD. But rather than exchanging complete tasks, one now needs to exchange pieces of tasks using time slices.
- Let σ be a schedule produced by an algorithm different from EDF. Then there exists a time slice $[t, t + 1)$ in which the executing task, denoted $\sigma(t)$, is not the task with the earliest absolute deadline among all ready tasks, denoted $E(t)$.
- The basic idea is to exchange this time slice with the next time slice in which $E(t)$ is executed in the current schedule. It can be shown that any such exchange cannot increase the maximum lateness of the task set.
- The next slide shows an example of such an exchange of time slices.

Optimality of EDF: Proof Sketch (2)



EDF Schedulability/Acceptance Test: Approach

- Similar to EDD, but the test must be done *online* whenever a new task J_{new} enters the system. Thus, assuming the current set of tasks J is schedulable, we need to check if $J' = J \cup J_{new}$ is also schedulable.
- If tasks J_1, J_2, \dots, J_n are ordered by increasing deadline, the worst-case finishing time of task J_i at time t is given by

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

- Here, $c_k(t)$ is the *remaining worst-case execution* time of task J_k . It is initially equal to C_k , but may have a lower value at time t when J_{new} arrives since task J_k (and others) may have been partially executed.
- Thus, the EDF schedulability/acceptance test performed online at time t amounts to verifying for each task $J_i \in J'$

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

EDF Schedulability/Acceptance Test: Algorithm

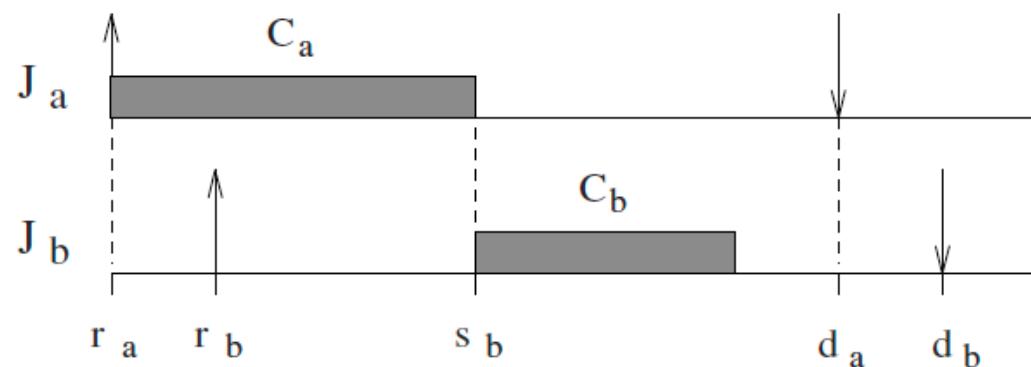
```
edf_schedulability_test(J, Jnew) {  
    J' = J ∪ {Jnew} ; /* ordered by increasing deadline */  
    f0 = get_current_time();  
    for each Ji ∈ J' {  
        fi = fi-1 + ci(t);  
        if (fi > di) {  
            return NOT_SCHEDULABLE;  
        }  
    }  
    return SCHEDULABLE;  
}
```

EDF with Precedence Constraints (EDF*)

- Scheduling of aperiodic tasks J_i with *precedence constraints*
 - Tasks J_i arrive dynamically, so they have arbitrary arrival times as for EDF above.
 - Again, each task J_i is characterized by its execution time C_i , relative deadline D_i , and two parameters only known at runtime: arrival time a_i and absolute deadline d_i .
 - In addition, a precedence graph G is given that specifies precedence constraints, where $J_a \rightarrow J_b$ means that task J_a is an immediate predecessor of task J_b .
 - Recall: all successors of a task in the precedence graph are activated (i.e., released)!
- *EDF**: Given a set J of n dependent tasks with arbitrary arrival times, transform this task set into a set J^* of n independent tasks by adequate modifications of the tasks' release times and deadlines. Then, the task set J^* is scheduled using the standard EDF algorithm. The modifications ensure that J is schedulable and the precedence constraints are satisfied if and only if J^* is schedulable.

Modification of Release Times

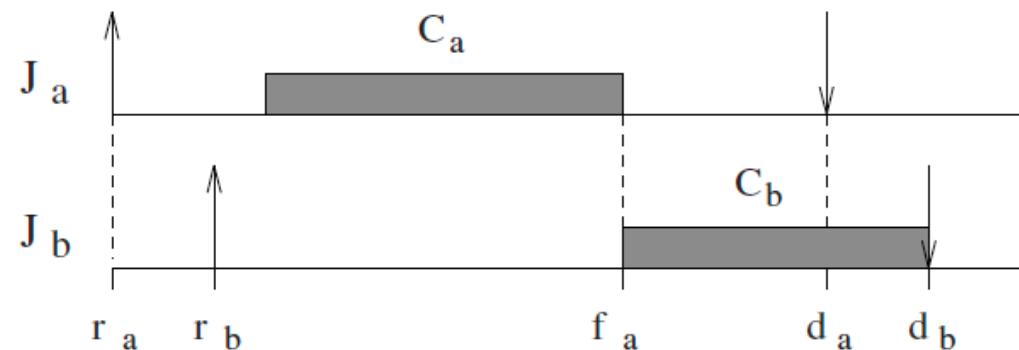
- Given tasks J_a and J_b with $J_a \rightarrow J_b$, in any valid schedule that meets precedence constraints the following two conditions must be satisfied:
 - $s_b \geq r_b$: task J_b must start its execution not earlier than its release time
 - $s_b \geq r_a + C_a$: task J_b must starts its execution not earlier than the minimum finishing time of its immediate predecessor, task J_a



- Thus, the release time r_b of task J_b can be replaced by the new release time
$$r_b^* = \max(r_b, r_a + C_a)$$

Modification of Deadlines

- Given tasks J_a and J_b with $J_a \rightarrow J_b$, in any valid schedule that meets precedence constraints the following two conditions must be satisfied:
 - $f_a \leq d_a$: task J_a must finish the execution before its deadline
 - $f_a \leq d_b - C_b$: task J_a must finish its execution not later than the maximum start time of its immediate successor, task J_b

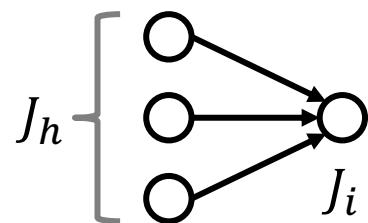


- Thus, the deadline d_a of task J_a can be replaced by the new deadline
$$d_a^* = \min(d_a, d_b - C_b)$$

Algorithm for Modifications

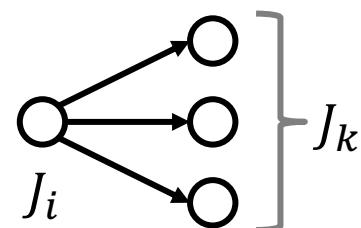
- *Modification of release times:*

1. For any initial node of the precedence graph, set $r_i^* = r_i$.
2. Select a task J_i such that its release time has not been modified the release times of all immediate predecessors J_h have been modified. If no such task exists, exit.
3. Set $r_i^* = \max[r_i, \max(r_h^* + C_h : J_h \rightarrow J_i)]$.
4. Return to step 2.



- *Modification of deadlines:*

1. For any terminal node of the precedence graph, set $d_i^* = d_i$.
2. Select a task J_i such that its deadline has not been modified but the deadlines of all immediate successors J_k have been modified. If no such task exists, exit.
3. Set $d_i^* = \min[d_i, \min(d_k^* - C_k : J_i \rightarrow J_k)]$.
4. Return to step 2.



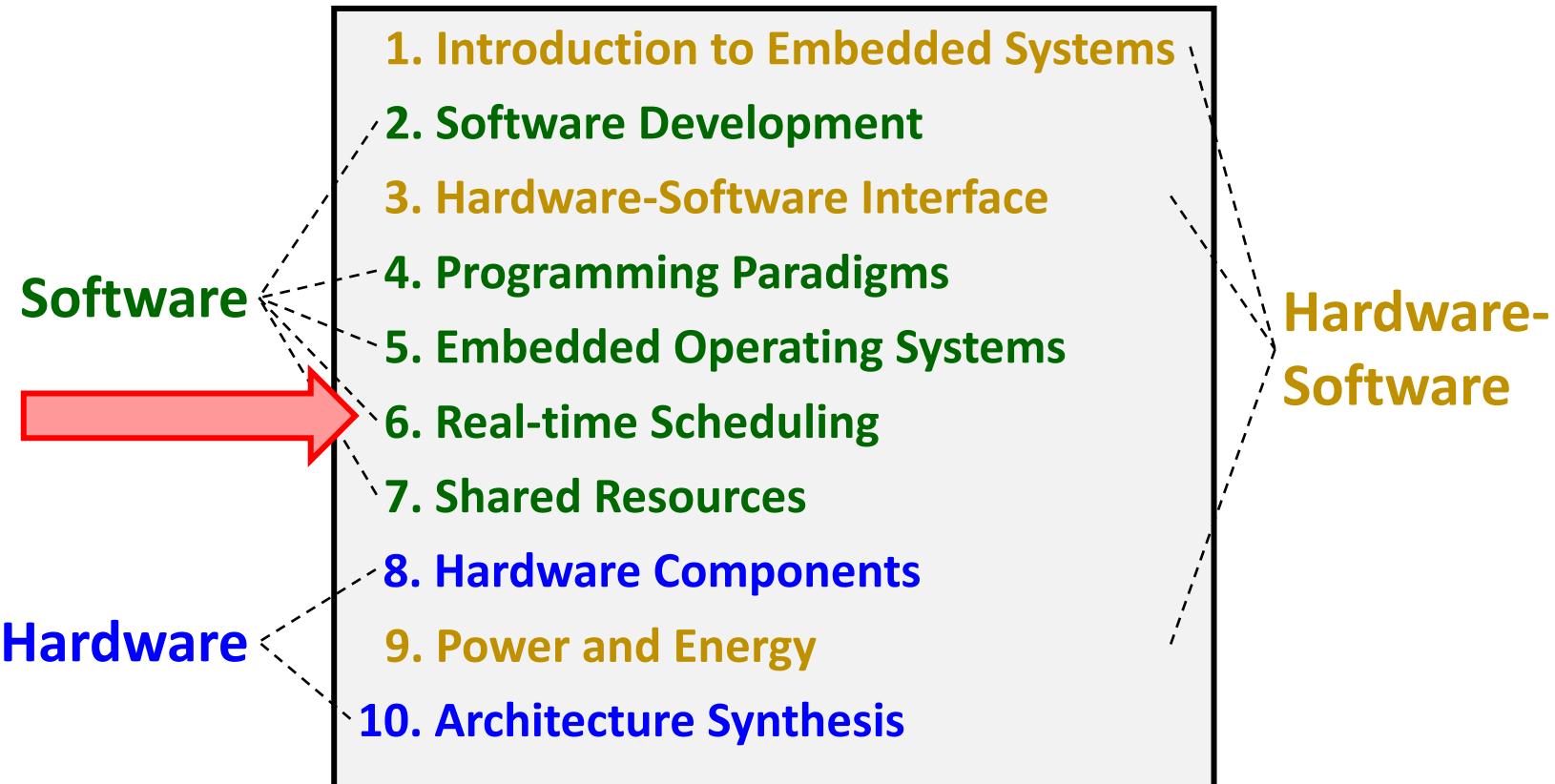
Introduction to Embedded Systems

6. Real-Time Scheduling

Prof. Dr. Marco Zimmerling



Where we are ...



Real-Time Scheduling of Aperiodic Tasks

Overview Aperiodic Task Scheduling

Scheduling of *aperiodic tasks* with real-time constraints:

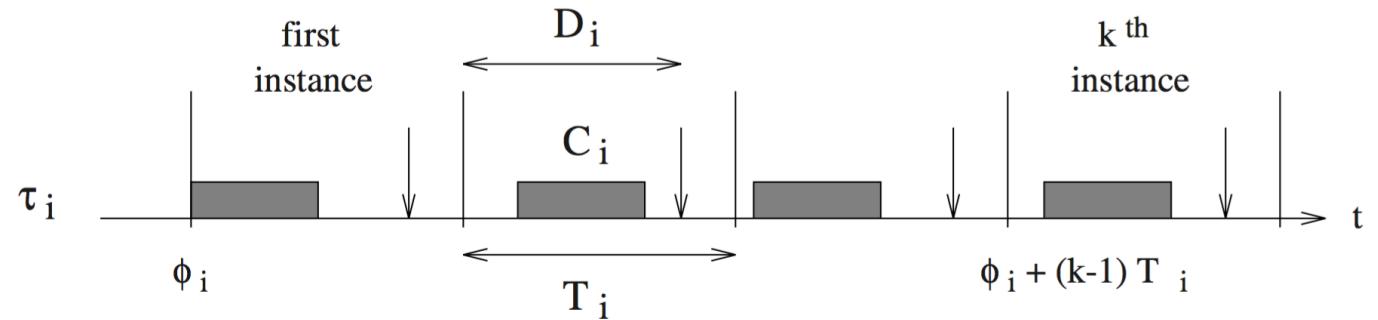
- Table with some known algorithms:

		Equal arrival times non preemptive	Arbitrary arrival times preemptive
(independent = without precedence constraints)	Independent tasks	EDD (Jackson)	EDF (Horn)
	Dependent tasks	LDF (Lawler)	EDF* (Chetto)

Real-Time Scheduling of Periodic Tasks

Periodic Tasks (1)

- Γ : set of periodic tasks
- τ_i : periodic task
- $\tau_{i,j}$: j -th instance of task τ_i
- $r_{i,j}, s_{i,j}, f_{i,j}, d_{i,j}$: release time, start time, finishing time, and absolute deadline of the j -th instance of task τ_i
- Φ_i : phase of task τ_i (release time of first instance)
- C_i : worst-case execution time of task τ_i
- T_i : period of task τ_i
- D_i : relative deadline of task τ_i
- The release times are given by $r_{i,j} = \Phi_i + (j - 1)T_i$ and the absolute deadlines are given by $d_{i,j} = r_{i,j} + D_i = \Phi_i + (j - 1)T_i + D_i$. If we have $D_i = T_i$ (*implicit deadline*), then the absolute deadlines are given by $d_{i,j} = \Phi_i + jT_i$.



Periodic Tasks (2)

- *Examples*: sensory data acquisition, low-level actuation, control loops, action planning, and system monitoring
- When an *application* features several concurrent periodic tasks with individual timing constraints, the OS has to guarantee that each periodic task τ_i is regularly activated at its proper *rate* $1/T_i$ and is completed within its deadline.
- Assumptions:
 - All periodic tasks are *independent*; that is, there are no precedence relations and no resource constraints.
 - *Tasks cannot suspend themselves*, for example, during I/O operations.
 - All *overheads* in the OS kernel are assumed to be *zero*.

Overview

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

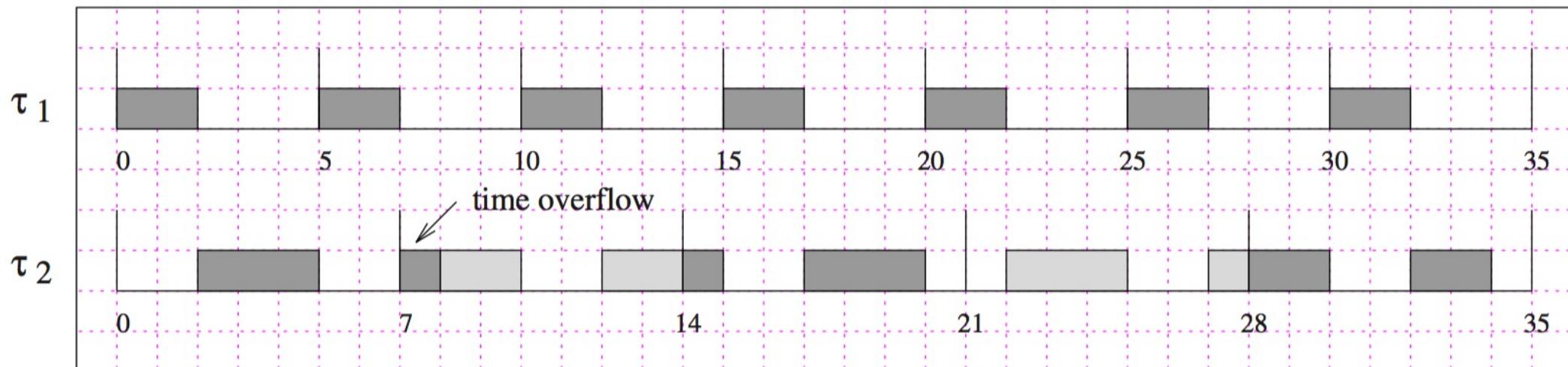
	$D_i = T_i$	$D_i \leq T_i$
	Deadline equals period	Deadline smaller than period
static priority	RM (rate-monotonic)	DM (deadline-monotonic)
dynamic priority	EDF	EDF

Rate-Monotonic (RM) Scheduling

- Scheduling of periodic tasks τ_i with *implicit deadlines*
 - All tasks have relative deadlines equal to their periods (i.e., $C_i \leq D_i = T_i$)
 - Every task has a priority. The priorities are assigned to the tasks before execution and do not change over time. This is called a static or *fixed priority assignment*.
 - The currently executing task instance is *preempted* by an instance of a task with a higher priority. In the following, we will just say “task” instead of “task instance.”
- *RM scheduling rule*: Given a set of n independent periodic tasks with implicit deadlines, assign a fixed priority to each task such that tasks with higher request rates (i.e., with shorter periods) have higher priorities. 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.

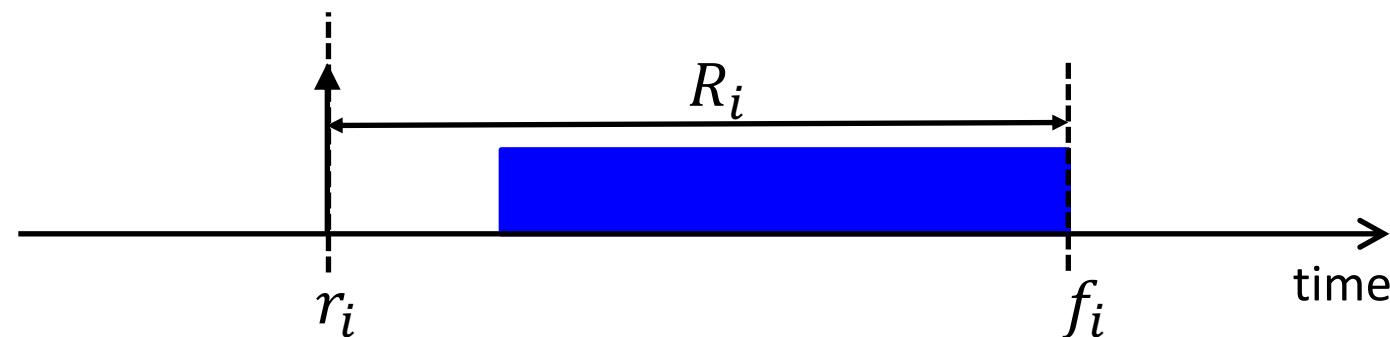
Example: Infeasible Schedule Produced by RM

- Two periodic tasks τ_1 and τ_2 with
 - Phases $\Phi_1 = \Phi_2 = 0$
 - Periods $T_1 = 5$ and $T_2 = 7$
 - Worst-case execution times $C_1 = 2$ and $C_2 = 4$
- Schedule produced by RM, where τ_1 has higher priority than τ_2 since $T_1 < T_2$:



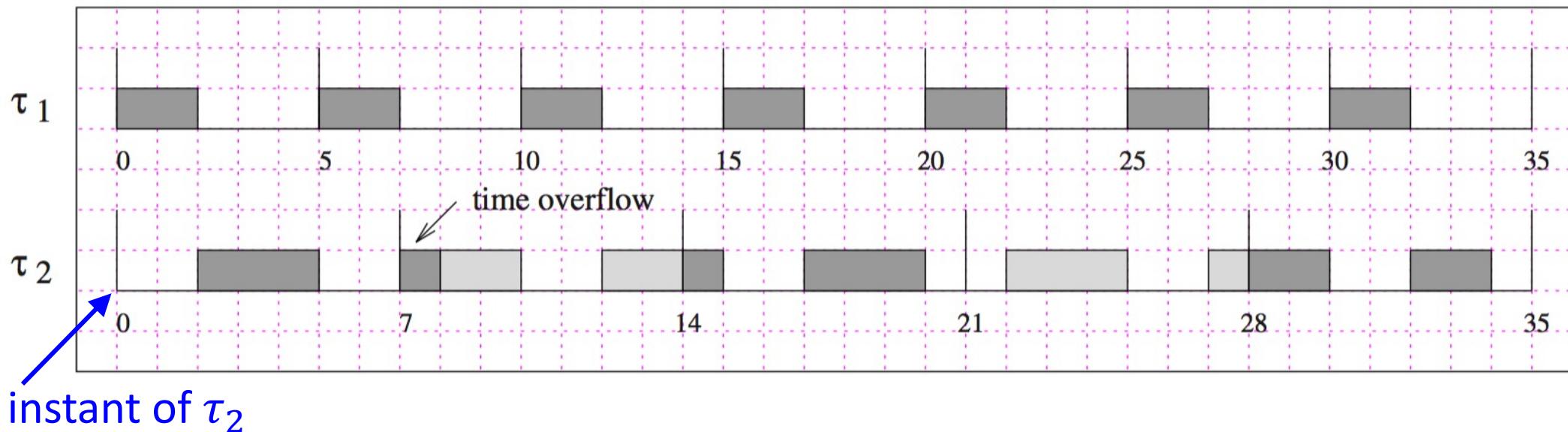
Critical Instant

- 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.
- To proof the optimality of RM, we need the concept of a *critical instant*.
- *Definition*: A critical instant of a task is the release time r_i that produces the largest response time R_i , that is, the largest difference between release time r_i and finishing time f_i .
- *Lemma*: For any task, a critical instant occurs whenever the task is released simultaneously with all higher-priority tasks.



Example: Infeasible Schedule Produced by RM

- Two periodic tasks τ_1 and τ_2 with
 - Phases $\Phi_1 = \Phi_2 = 0$
 - Periods $T_1 = 5$ and $T_2 = 7$
 - Worst-case execution times $C_1 = 2$ and $C_2 = 4$
- Schedule produced by RM, where τ_1 has higher priority than τ_2 since $T_1 < T_2$:

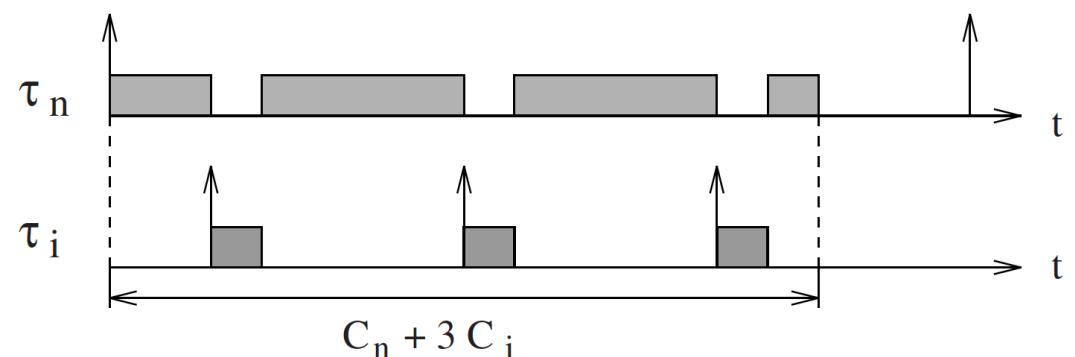
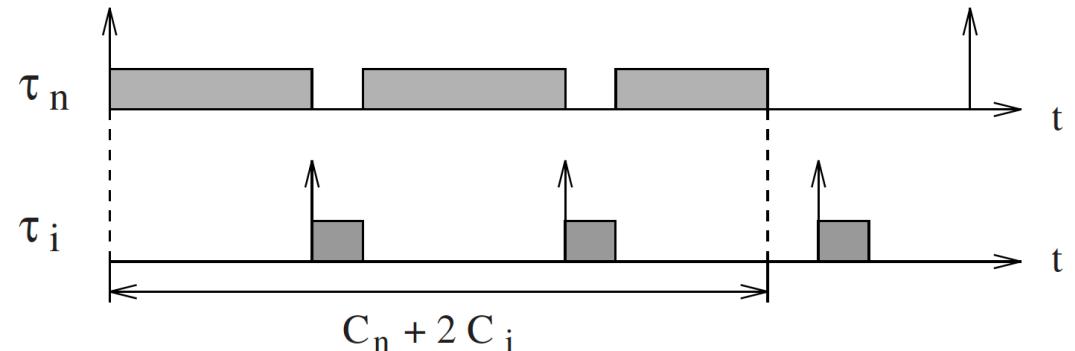


Critical Instant: Proof Sketch

- Let $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$ be a set of periodic tasks ordered by increasing periods. Thus, task τ_n has the longest period and, according to RM, the lowest priority.

- The response time of τ_n is delayed by interference of τ_i with higher priority.

- The response time of τ_n may increase if τ_i is released earlier.



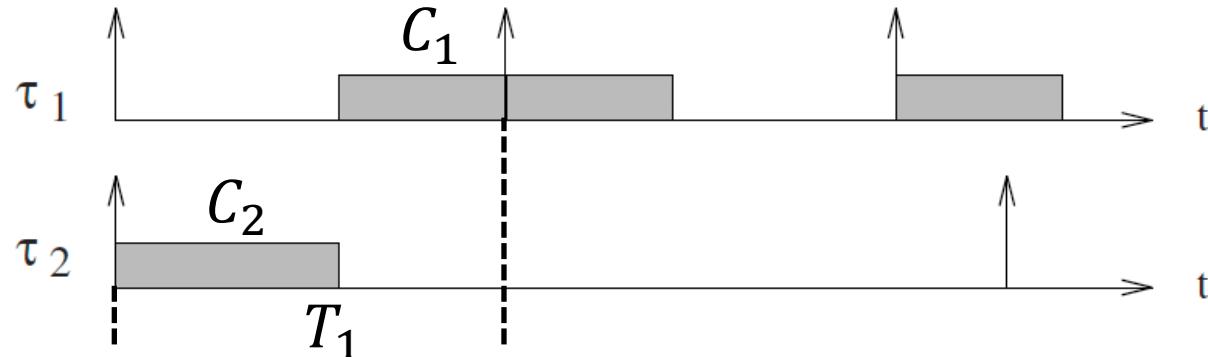
- Thus, the response time is largest if tasks τ_n and τ_i are released simultaneously.
- Repeating the argument for all higher-priority tasks proves the lemma.

Optimality of RM: Proof Overview

- We just proved that, for any task, a critical instant occurs whenever the task is released simultaneously with all higher-priority tasks.
- This means that the schedulability of tasks can easily be checked at their critical instants: If all tasks are feasible at their critical instants (i.e., the largest response time does not exceed the deadline for every task), then the task set is schedulable in any other condition.
- Based on this result, the optimality of RM can be shown in two steps:
 1. Show that, given a set with two periodic tasks, if the task set is schedulable by an arbitrary priority assignment, then it is also schedulable by RM.
 2. Extend the result to a set of n periodic tasks.
- In the following, we will only prove the first step.

Optimality of RM: Proof of First Step (1)

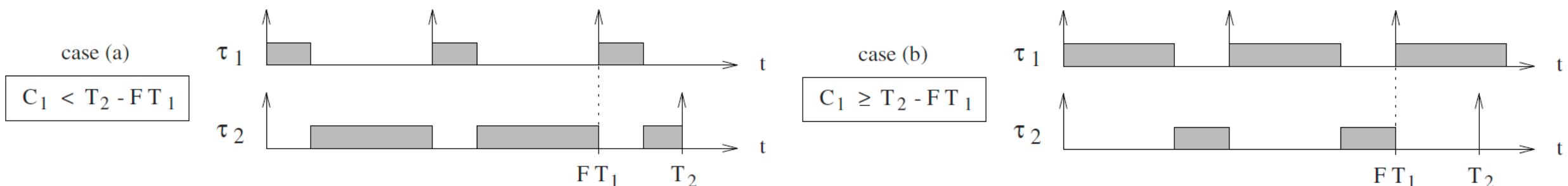
- We have $\Gamma = \{\tau_1, \tau_2\}$ with $T_1 < T_2$.
- If priorities are *not* assigned according to RM, then τ_2 has the highest priority.



- Looking at the critical instant of τ_1 , where it is simultaneously released with all higher-priority tasks (here this is only τ_2), we find that the task set is schedulable if
$$C_1 + C_2 \leq T_1$$

Optimality of RM: Proof of First Step (2)

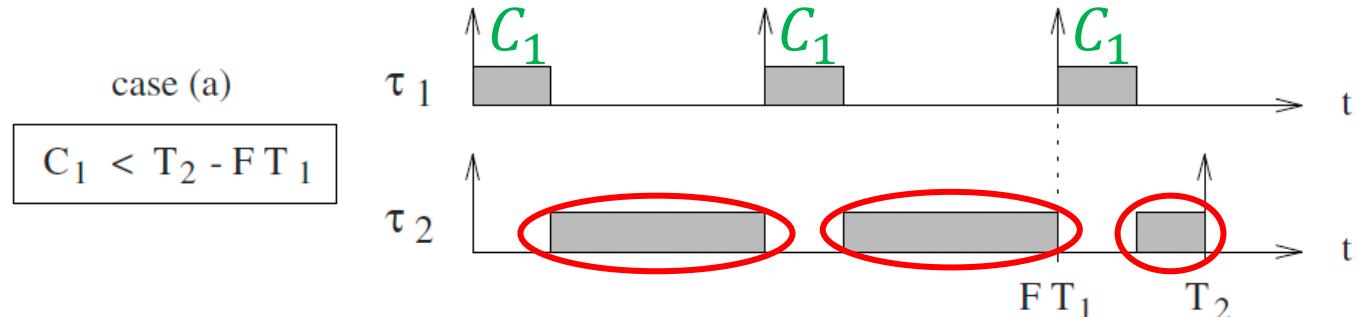
- If priorities *are* assigned according to RM, then τ_1 has the highest priority.
- Two cases must be considered to guarantee a feasible schedule, where $F = \lceil T_2/T_1 \rceil$ is the number of periods of τ_1 that are fully contained in T_2 .
 - *Case (a)*: The computation time of τ_1 , when synchronously activated with τ_2 , is short enough so that all its requests are completed before the second request of τ_2 .
 - *Case (b)*: The computation time of τ_1 , when synchronously activated with τ_2 , is long enough to overlap with the second request of τ_2



- We must show that in either case the condition for schedulability without RM (see previous slide) implies the condition for schedulability with RM (see next slides).

Optimality of RM: Proof of First Step (3)

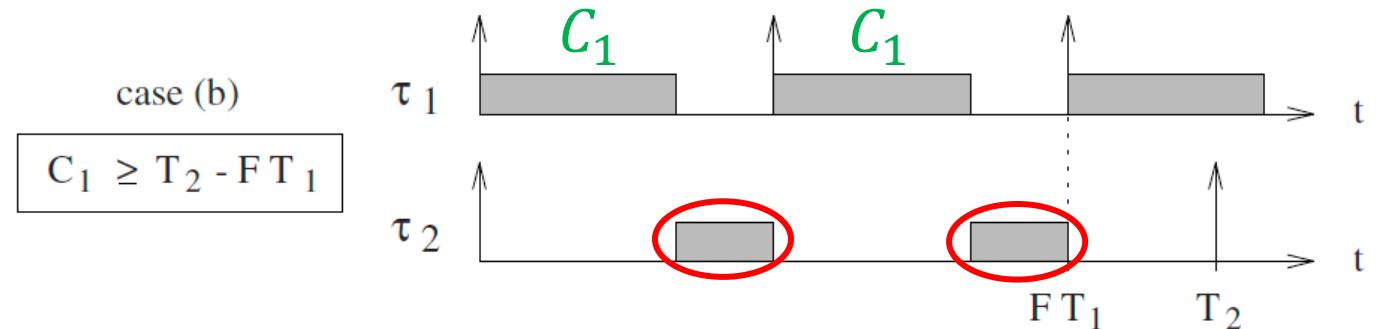
- In *case (a)* the task set is schedulable
if $(F + 1)C_1 + C_2 \leq T_2$



- Thus, we have to show that $C_1 + C_2 \leq T_1 \Rightarrow (F + 1)C_1 + C_2 \leq T_2$
 - We begin with:
Multiply both sides with F :
Since $F \geq 1$ we can write:
Add C_1 to each member:
Since in case (a) $C_1 < T_2 - FT_1$:
- $$\begin{aligned} C_1 + C_2 &\leq T_1 \\ FC_1 + FC_2 &\leq FT_1 \\ FC_1 + C_2 &\leq FC_1 + FC_2 \leq FT_1 \\ (F + 1)C_1 + C_2 &\leq FT_1 + C_1 \\ (F + 1)C_1 + C_2 &\leq FT_1 + C_1 < T_2 \end{aligned}$$

Optimality of RM: Proof of First Step (4)

- In *case (b)* the task set is schedulable if $F C_1 + C_2 \leq FT_1$



- Thus, we have to show that $C_1 + C_2 \leq T_1 \Rightarrow FC_1 + C_2 \leq FT_1$
 - We begin with:
Multiply both sides with F :
Since $F \geq 1$ we can write:
- $$C_1 + C_2 \leq T_1$$
- $$FC_1 + FC_2 \leq FT_1$$
- $$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1$$

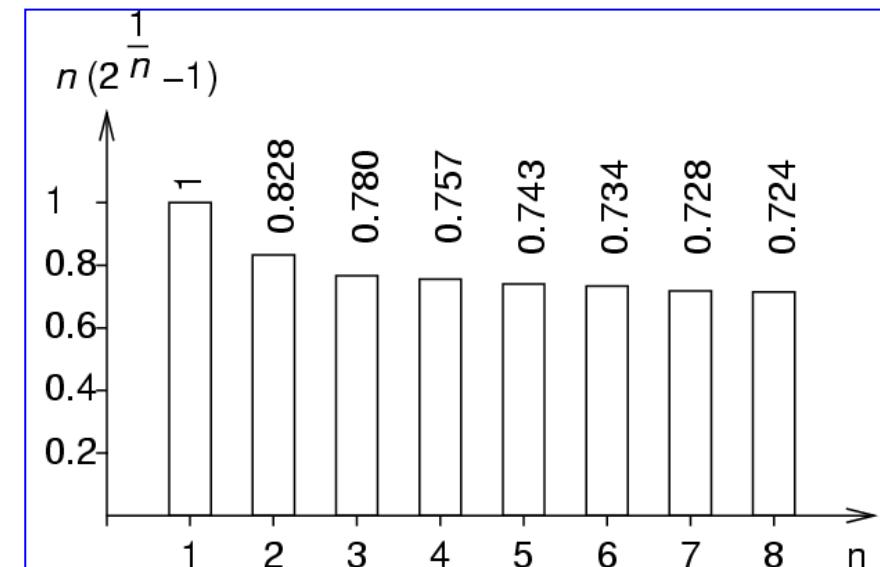
- Summary:* Both cases combined, we have shown that, given two periodic tasks τ_1 and τ_2 with $T_1 < T_2$, if the schedule is feasible with an arbitrary priority assignment, then it is also feasible with RM. In other words, RM is optimal.

RM Schedulability Test

- A set of n periodic real-time tasks is schedulable using RM if

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

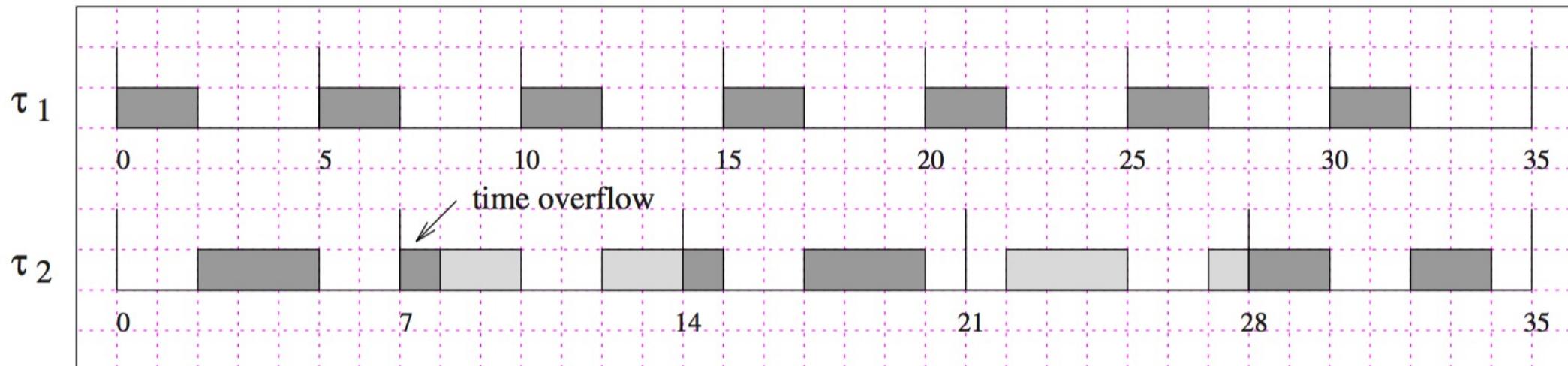
- This condition is *sufficient but not necessary*. That is, if the above condition holds for a given task set, then this task set is definitely schedulable using RM. However, if the above condition does not hold, then the task set may or may not be schedulable using RM.



- The term $\sum_{i=1}^n C_i/T_i$ is called the *processor utilization U* of a set of n periodic real-time tasks. It denotes the fraction of time the processor spends executing the task set (i.e., the “computational load” induced by the task set).

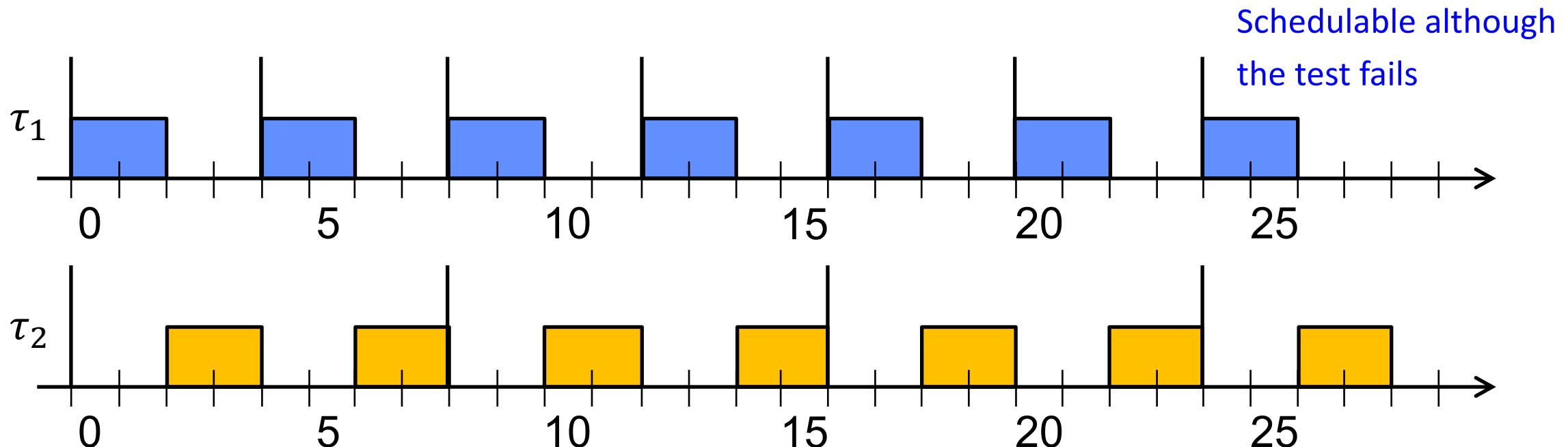
Example Revisited: Processor Utilization

- Two periodic tasks τ_1 and τ_2 with
 - Phases $\Phi_1 = \Phi_2 = 0$
 - Periods $T_1 = 5$ and $T_2 = 7$
 - Worst-case execution times $C_1 = 2$ and $C_2 = 4$
- $\text{Processor utilization } U = \sum_{i=1}^n C_i/T_i = \frac{2}{5} + \frac{4}{7} \approx 0.97 > 2(2^{1/2} - 1) \approx 0.83$



Another Example: Sufficiency of RM Schedulability Test

- Two periodic tasks τ_1 and τ_2 with
 - Phases $\Phi_1 = \Phi_2 = 0$
 - Periods $T_1 = 4$ and $T_2 = 8$, thus task τ_1 has higher priority than task τ_2
 - Worst-case execution times $C_1 = 2$ and $C_2 = 4$
- $\text{Processor utilization } U = \sum_{i=1}^n C_i/T_i = \frac{2}{4} + \frac{4}{8} = 1 > 2(2^{1/2} - 1) \approx 0.83$



Deadline-Monotonic (DM) Scheduling

- Scheduling of periodic tasks τ_i where the relative deadlines may be smaller than the corresponding period (i.e., $C_i \leq D_i \leq T_i$). All other properties are as for RM:
 - Priorities are assigned to tasks before execution and do not change over time.
 - Currently executing task is preempted by higher-priority task.
- *DM scheduling rule:* Given a set of n independent periodic tasks τ_i with $D_i \leq T_i$, assign a fixed priority to each task such that tasks with shorter relative deadlines D_i have higher priorities. Thus, at any instant, the task with the shortest relative deadlines is executed. DM is optimal in the sense that if a task set is schedulable by some fixed-priority assignment, then it is also schedulable by DM.
- *Schedulability test:* A set of n periodic real-time tasks is schedulable using DM if

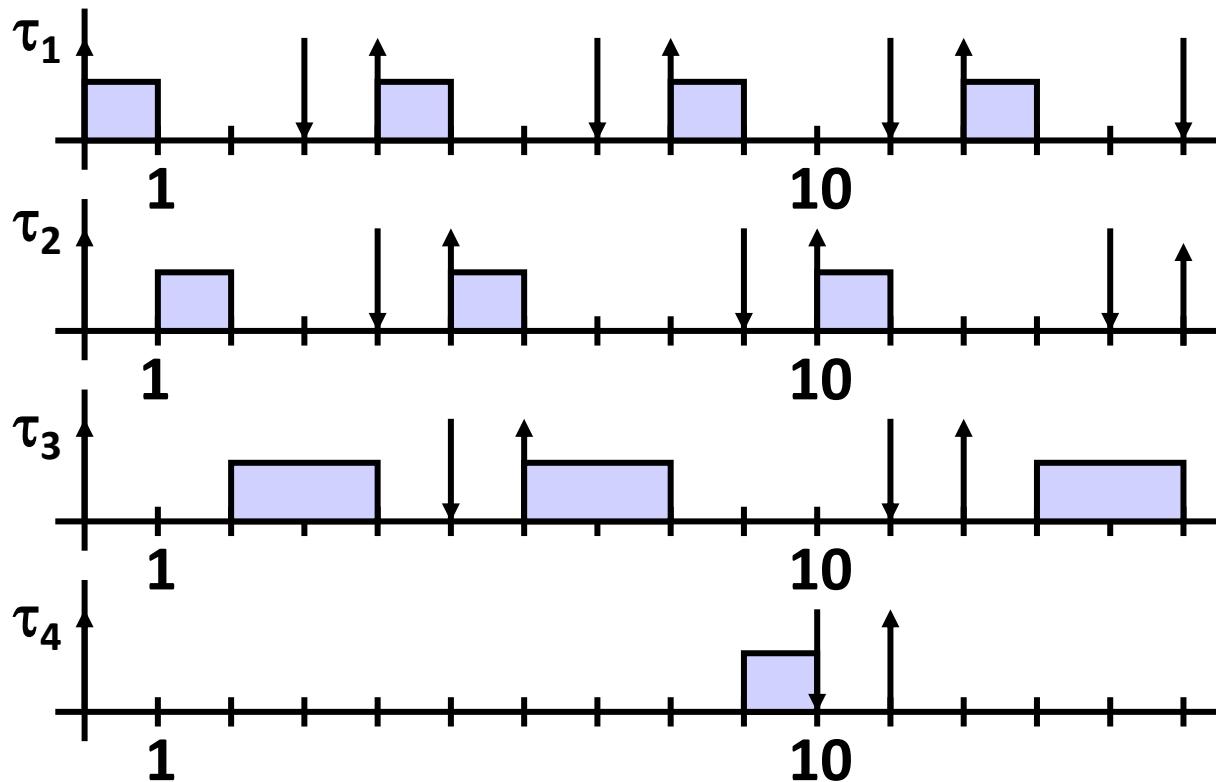
$$\sum_{i=1}^n C_i/D_i \leq n(2^{1/n} - 1)$$

This condition is *sufficient but not necessary*.

Example: Sufficiency of DM Schedulability Test

- Four tasks are given:

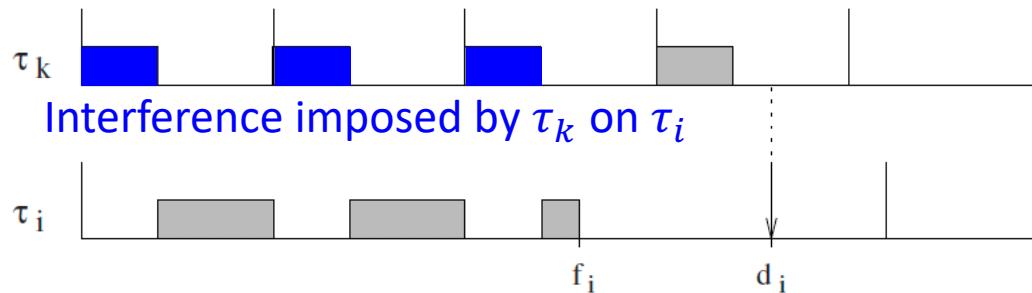
τ_i	Φ_i	T_i	D_i	C_i
τ_1	0	4	3	1
τ_2	0	5	4	1
τ_3	0	6	5	2
τ_4	0	11	10	1



- Processor utilization: $U = \sum_{i=1}^n C_i/T_i = \frac{1}{4} + \frac{1}{5} + \frac{2}{6} + \frac{1}{11} \approx 0.87$ Scheduled although the test fails
- Schedulability test:* $\sum_{i=1}^n C_i/D_i = \frac{1}{3} + \frac{1}{4} + \frac{2}{5} + \frac{1}{10} \approx 1.08 > 4(2^{1/4} - 1) \approx 0.76$

DM Necessary and Sufficient Schedulability Test (1)

- This computationally more involved test is based on the following observations:
 - The *worst-case processor demand* occurs when all tasks are released simultaneously, that is, at their critical instants.
 - To be schedulable, for each task τ_i in the set, the sum of its processing time and the *interference* imposed by all higher-priority tasks must be less than or equal to D_i .



- The *worst-case interference* I_i for task τ_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 $\tau_m < \tau_n \Leftrightarrow D_m < D_n$:

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

DM Necessary and Sufficient Schedulability Test (2)

- The *longest response time* R_i of a periodic task τ_i is computed, at the critical instant, as the sum of its worst-case execution time C_i and the interference I_i due to preemption by higher-priority tasks:

$$R_i = C_i + I_i$$

- Hence, the schedulability test needs to compute, for all tasks τ_i , the smallest R_i that satisfies the following equality:

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

- Then, for a task set to be schedulable, $R_i \leq D_i$ must hold for all tasks τ_i .
- It can be shown that this *condition is necessary and sufficient*.

DM Necessary and Sufficient Schedulability Test: Algorithm

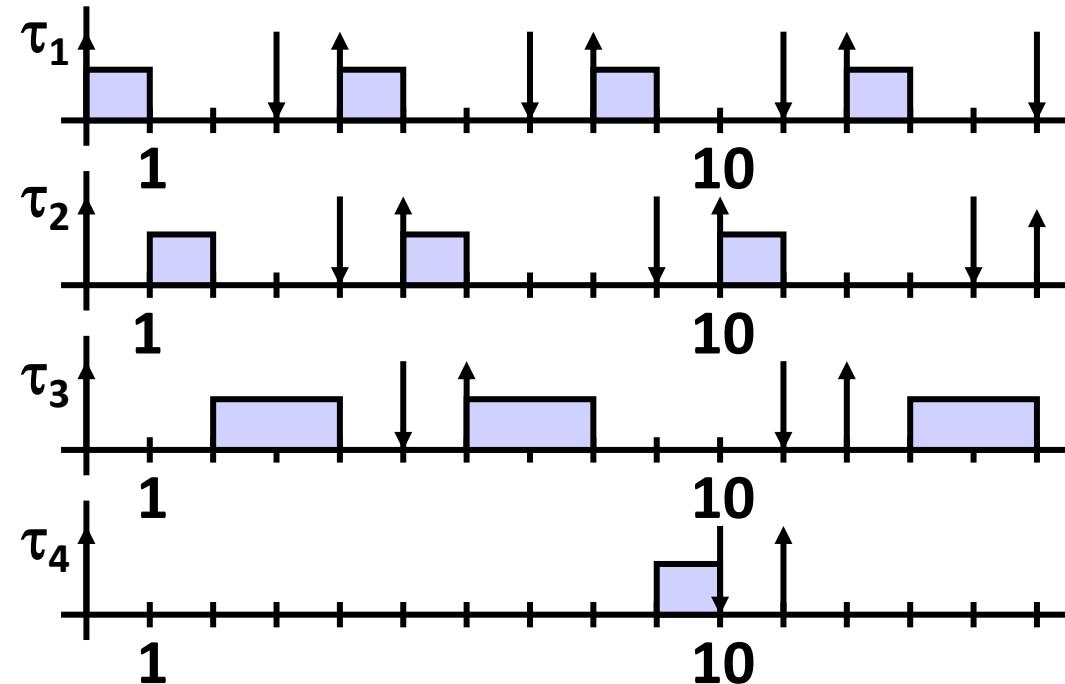
```
DM-guarantee ( $\Gamma$ ) {  
    for (each  $\tau_i \in \Gamma$ ) {  
         $I_i = \sum_{k=1}^{i-1} C_k$ ;  
        do {  
             $R_i = I_i + C_i$ ;  
            if ( $R_i > D_i$ ) return(UNSCHEDULABLE);  
             $I_i = \sum_{k=1}^{i-1} \left\lceil \frac{R_i}{T_k} \right\rceil C_k$ ;  
        } while ( $I_i + C_i > R_i$ );  
    }  
    return(SCHEDULABLE);  
}
```

DM Necessary and Sufficient Schedulability Test: Example

- Four tasks are given:

τ_i	Φ_i	T_i	D_i	C_i
τ_1	0	4	3	1
τ_2	0	5	4	1
τ_3	0	6	5	2
τ_4	0	11	10	1

- What is R_4 ?



- Step 0: $R_4 = \sum_{i=1}^4 C_i = \dots, R_4 > D_4?, I_4 = \sum_{i=1}^3 \left\lceil \frac{R_4}{T_i} \right\rceil = \dots, I_4 + C_4 > R_4?$
- Step 1: $R_4 = I_4 + C_4 = \dots, R_4 > D_4?, I_4 = \sum_{i=1}^3 \left\lceil \frac{R_4}{T_i} \right\rceil = \dots, I_4 + C_4 > R_4?$
- Step 2: ...

DM Necessary and Sufficient Schedulability Test: Example

- Step 0: $R_4 = 5, I_4 = 5, I_4 + C_4 > R_4$
- Step 1: $R_4 = 6, I_4 = 6, I_4 + C_4 > R_4$
- Step 2: $R_4 = 7, I_4 = 8, I_4 + C_4 > R_4$
- Step 3: $R_4 = 9, I_4 = 9, I_4 + C_4 > R_4$
- Step 4: $R_4 = 10, I_4 = 9, I_4 + C_4 = R_4$

- This means that task τ_4 finishes at $R_4 = 10$, which can also be seen by looking at the schedule on the previous slides. Since $R_4 \leq D_4$, τ_4 is schedulable.
- If, like in this example, $R_i \leq D_i$ for all tasks τ_i in the task set, we can conclude that the task set is schedulable by DM.

Earliest Deadline First (EDF)

- As before, we consider preemptive scheduling of periodic tasks τ_i .
- *EDF algorithm*: A dynamic priority is assigned to each task such that tasks with earlier absolute deadlines have higher priorities. The currently executing task is preempted whenever a task with earlier absolute deadline becomes active. EDF is optimal in the sense that no other algorithm can schedule a set of periodic real-time tasks that cannot be scheduled by EDF.
- Using EDF the priorities are assigned *dynamically*, because the absolute deadline $d_{i,j}$ of a periodic task τ_i depends on the j -th instance that is currently active
$$d_{i,j} = \Phi_i + (j - 1)T_i + D_i$$
- Instead, using RM (or DM) the priorities are *fixed*, because these are determined based on the periods T_i (or deadlines D_i) which do not change over time.

EDF Schedulability Test for Implicit Deadlines

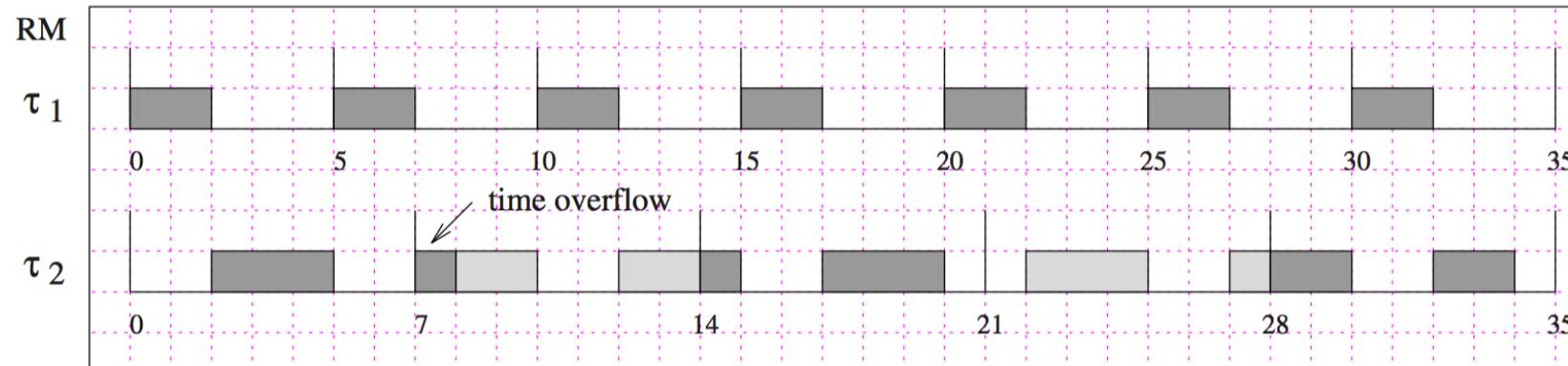
- A set of n periodic real-time tasks, where $D_i = T_i$ for all tasks τ_i , is schedulable using EDF if and only if

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

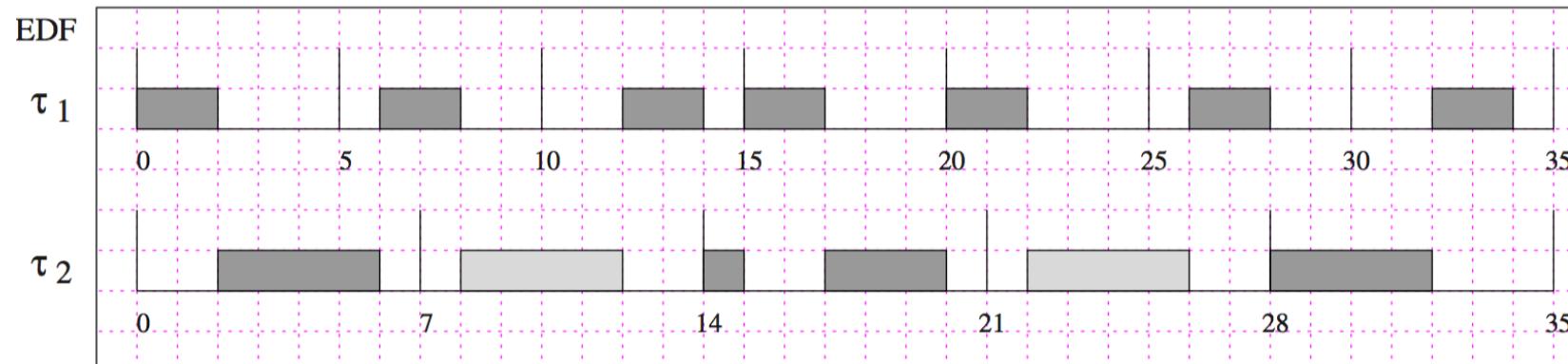
- This condition is both *necessary and sufficient*.
- Proof sketch:
 1. If the processor utilization satisfies $U > 1$, then there exists no valid schedule. This is because the total demand in the time interval $T = T_1 \cdot T_2 \cdot \dots \cdot T_n$ is $\sum_{i=1}^n \frac{C_i}{T_i} T = UT > T$, which exceeds the available processor time in this interval.
 2. If the processor utilization satisfies $U \leq 1$, then there exists a valid schedule. We can prove this by contradiction: Assume that a deadline miss occurs at some time, then we can show that the processor utilization before this time exceeds 1.

Example: RM versus EDF

- Two periodic tasks τ_1 and τ_2 with
 - Phases $\Phi_1 = \Phi_2 = 0$
 - Periods $T_1 = 5$ and $T_2 = 7$
 - Worst-case execution times $C_1 = 2$ and $C_2 = 4$



$$U \approx 0.97 > 0.83$$



$$U \approx 0.97 < 1$$

No deadline miss and fewer preemptions due to dynamic priority assignment

Introduction to Embedded Systems

6. Real-Time Scheduling

Prof. Dr. Marco Zimmerling



Organization

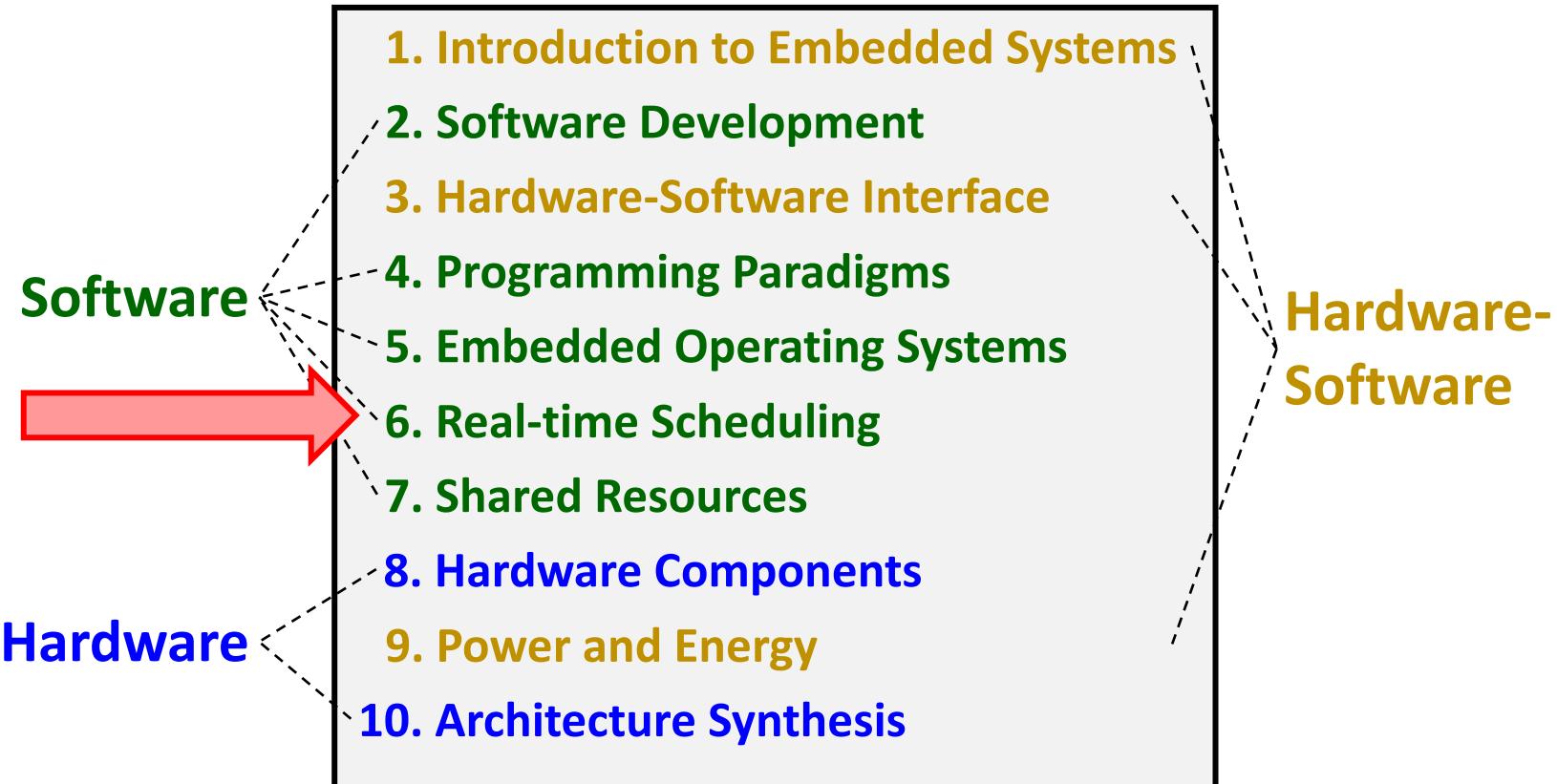
Lectures:

- Next week (December 20): [virtually on Zoom](#) (link will be posted on ILIAS)

Exercises:

- Due to timing constraints, we will no longer introduce the exercise sheets
- Today (December 13): [no exercise](#)
- Next week (December 20):
 - Solutions of fourth exercise sheet presented
 - Fifth exercise sheet released

Where we are ...



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 = without precedence constraints)	Independent tasks	EDD (Jackson)	EDF (Horn)
	Dependent tasks	LDF (Lawler)	EDF* (Chetto)

Overview

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

	$D_i = T_i$	$D_i \leq T_i$
	Deadline equals period	Deadline smaller than period
static priority	RM (rate-monotonic)	DM (deadline-monotonic)
dynamic priority	EDF	EDF

Real-Time Scheduling of Mixed Task Sets

Mixed Task Sets: Motivation and Terminology

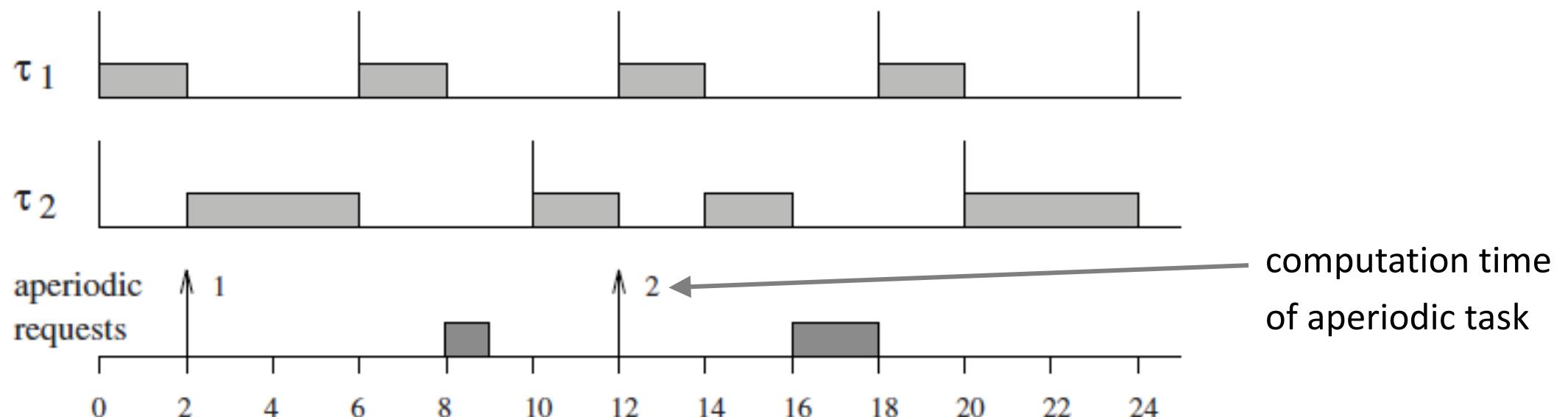
- Many applications feature both periodic and aperiodic tasks:
 - *Periodic tasks are time-driven* and execute critical activities (e.g., control) with hard timing constraints in order to guarantee regular activation rates of the tasks.
 - *Aperiodic tasks are usually event-driven* and may have hard, soft, or non-real-time requirements depending on the specific application.
- There are two types of hard aperiodic tasks:
 - *A sporadic task has a maximum arrival rate* or, equivalently, a minimum inter-arrival time between consecutive instances of the task. This is a worst-case assumption on the environment and allows for an offline guarantee on the task's schedulability.
 - *A firm task* is an aperiodic task for which the maximum arrival rate of the associated event cannot be bounded, but an online guarantee on the schedulability of individual instances of the task is still required. An online acceptance test is used to accept a request for a firm task only if the task instance can be served within its deadline.
- The objective is to meet the timing constraints of all critical periodic or aperiodic tasks and to provide small average response times for soft and non-real-time tasks.

Assumptions

- Each periodic task has a relative deadline equal to its period, that is, $D_i = T_i$.
- All periodic tasks start simultaneously at time $t = 0$.
- The arrival times of aperiodic tasks are unknown.
- All tasks are fully preemptable.

Background Scheduling

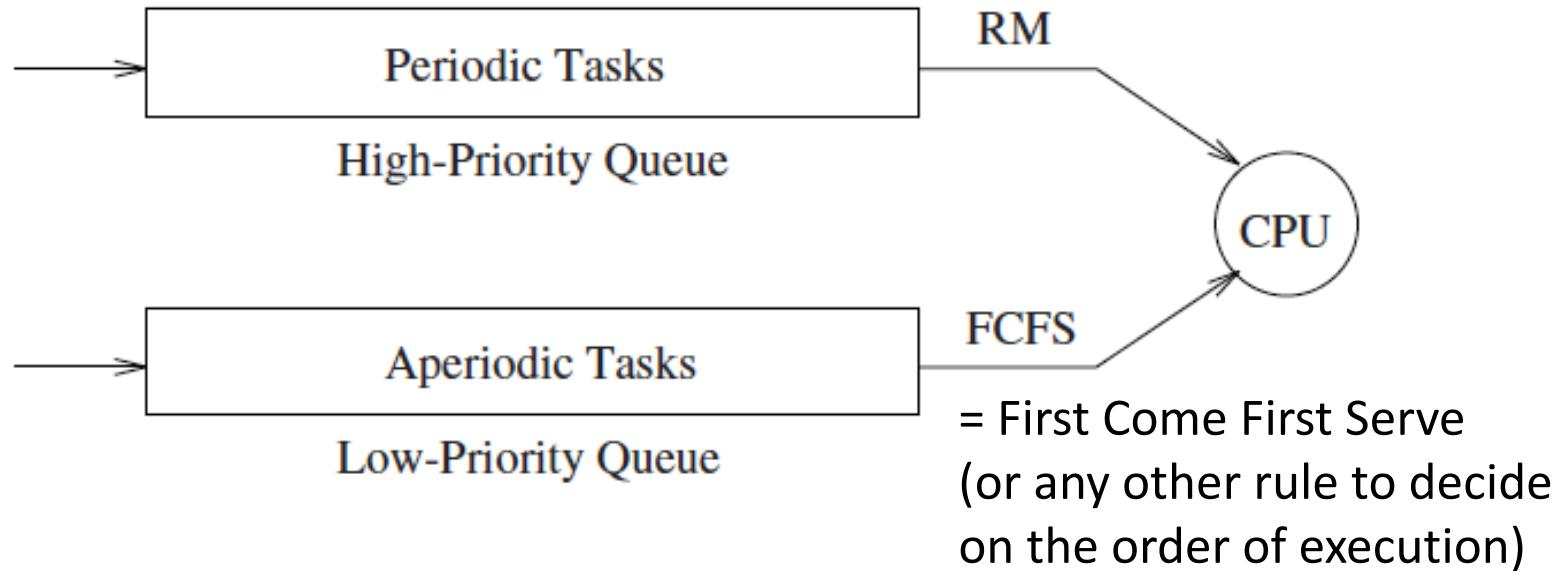
- *Principle:*
 - Periodic tasks are scheduled using RM
 - Aperiodic tasks are served whenever there are no periodic tasks ready to execute, that is, aperiodic tasks are "scheduled in the background"
- *Example:* Two periodic tasks with $C_1 = 2, T_1 = 6$ and $C_2 = 4, T_2 = 10$



Background Scheduling

- *Advantages:*

- Simplicity
- Execution of periodic tasks is not affected



= First Come First Serve
(or any other rule to decide
on the order of execution)

- *Disadvantages:*

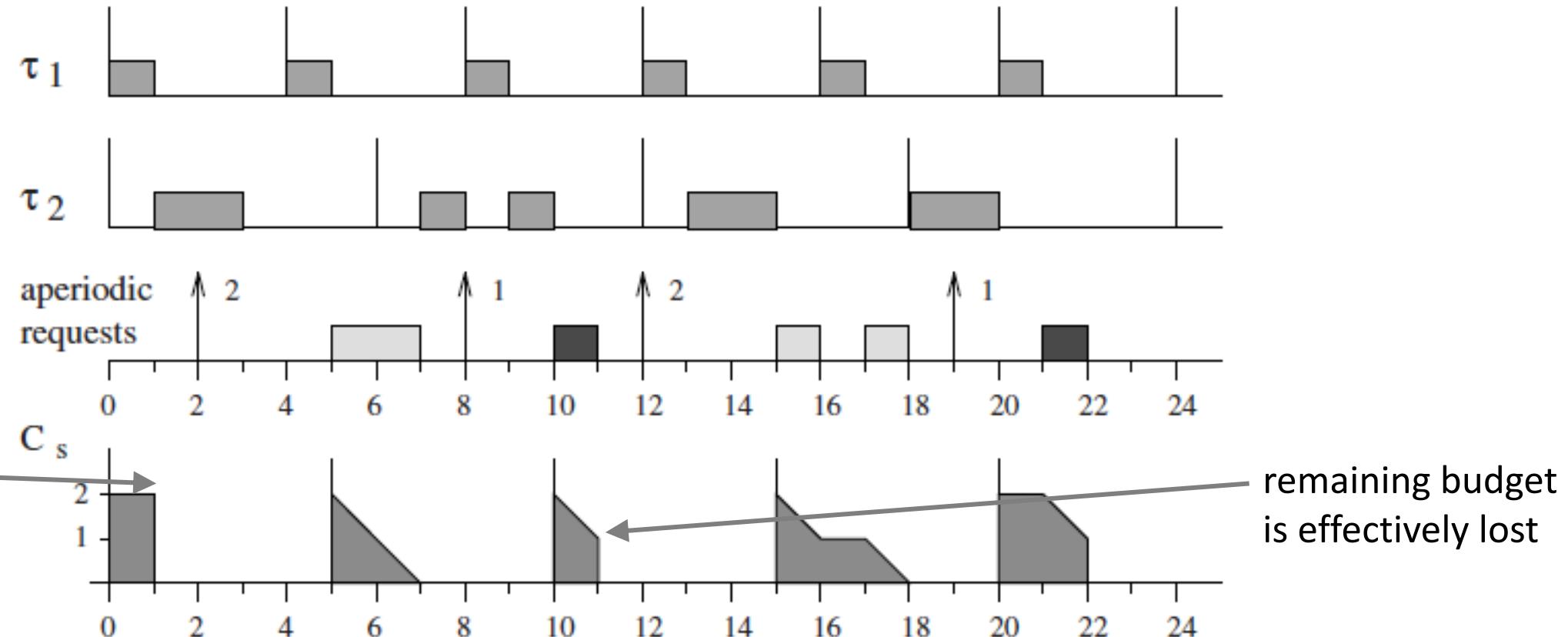
- The response time of aperiodic tasks can be prohibitively long. This problem arises in particular if the load induced by the periodic tasks is high.
- There is no possibility to assign a higher priority to aperiodic tasks.

Polling Server

- *Idea:* Provide short average response times for aperiodic tasks by introducing an artificial periodic task that “serves” aperiodic requests as soon as possible
- *Principle:*
 - Introduce Polling Server (PS) task with period T_s and computation time C_s . The period T_s can be chosen to match the response time requirement of aperiodic tasks. The computation time C_s is called the *capacity* or the *budget* of the server.
 - Schedule PS task like any other periodic task, that is, using RM.
 - When PS has the highest current priority, it serves any pending aperiodic requests until its capacity is exhausted. If no aperiodic requests are pending, PS suspends itself until the beginning of its next period, and the budget originally allocated for aperiodic service is freed up and assigned to periodic tasks.
- *Advantage:* Improves average response time compared to background scheduling
- *Disadvantage:* If an aperiodic request arrives just after the server has suspended, it must wait until the beginning of the next polling period.

Polling Server: Example

- Two periodic tasks with $C_1 = 1, T_1 = 4$ and $C_2 = 4, T_2 = 6$
- PS task with $C_s = 2, T_s = 5$



Polling Server: Schedulability Analysis

- *Observation*: In the worst case, the server introduces the same interference as an equivalent periodic task with period T_s and computation time C_s .
- *Schedulability test*: If a set of n periodic tasks and the server are scheduled by RM, then the schedulability can be guaranteed if

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

utilization of n periodic tasks utilization of the server $n + 1$ because there are effectively that many periodic tasks in the system

- Again, this schedulability test is *sufficient* but not necessary.

Polling Server: Schedulability of Firm Aperiodic Tasks

- At runtime, a request for a firm aperiodic task arrives with computation time C_a and relative deadline D_a . We need an acceptance test to check whether the request, which is handled by the server, can be completed within its deadline.
- If $C_a \leq C_s$, schedulability is guaranteed if

$$2T_s \leq D_a$$

- For arbitrary computation times, schedulability is guaranteed if

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

worst-case waiting time until
the server becomes active

total number of server periods
needed to process the request

Total Bandwidth Server

- *Idea:* Provide short average response times for aperiodic tasks by assigning a possible earlier deadline to aperiodic requests when they arrive
- *Principle:*

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

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

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

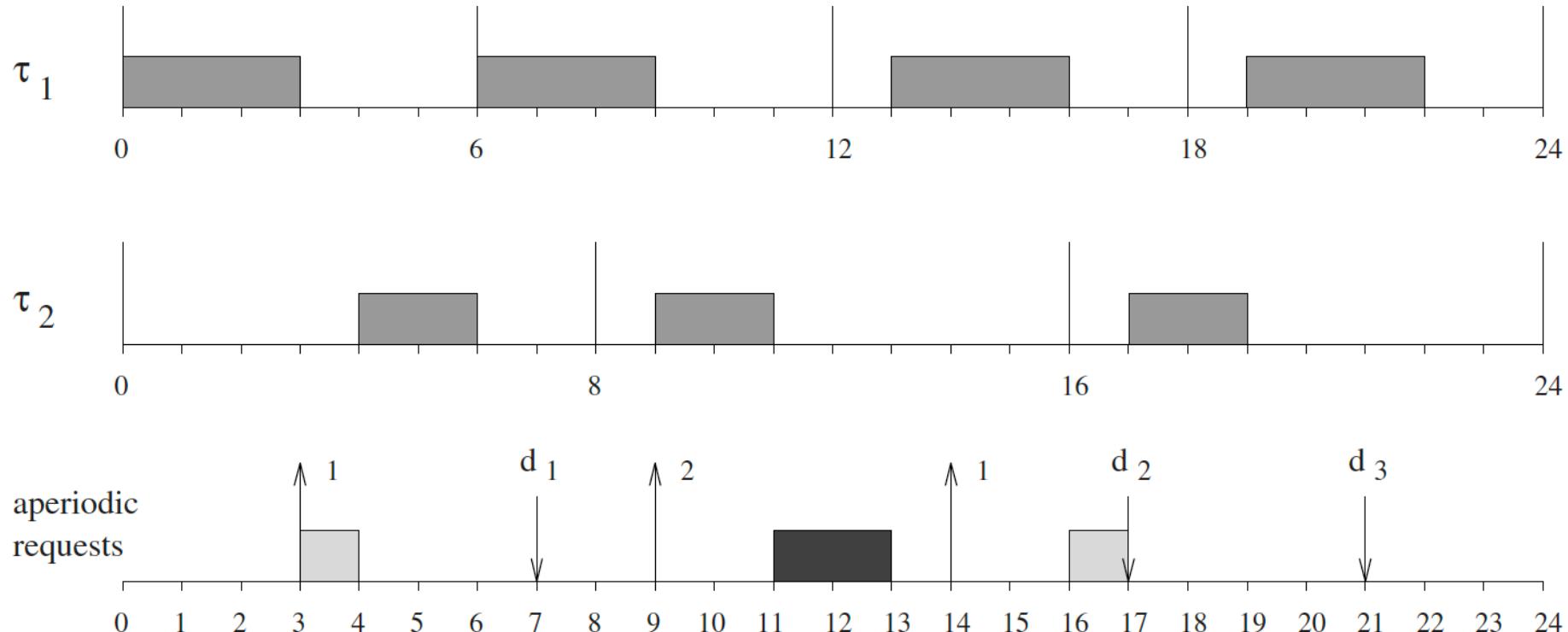
- Once the deadline is assigned, the request is inserted into the ready queue of the system and scheduled by EDF like any other periodic task instance.

- *Advantages:*

- Each time an aperiodic request arrives, total bandwidth of the server is immediately assigned to it, whenever possible. This can reduce the average response time.
- Implementation overhead is practically negligible

Total Bandwidth Server: Example

- Two periodic tasks with $C_1 = 3, T_1 = 6$ and $C_2 = 2, T_2 = 8$
- Total Bandwidth Server (TBS) with $U_s = 1 - U_p = 0.25$



$$d_1 = r_1 + \frac{C_1}{U_s} = 7$$

$$d_2 = r_2 + \frac{C_2}{U_s} = 17$$

$$d_3 = \max(r_3, d_2) + \frac{C_3}{U_s} = 21$$

Total Bandwidth Server: Schedulability Analysis

- Given a set of n periodic tasks with implicit deadlines (i.e., $D_i = T_i$ for all tasks), processor utilization $\sum_{i=1}^n C_i/T_i = U_p$, and a TBS with processor utilization U_s , then the whole set is schedulable by EDF if and only if

$$U_p + U_s \leq 1$$

- This condition is both *necessary and sufficient*.

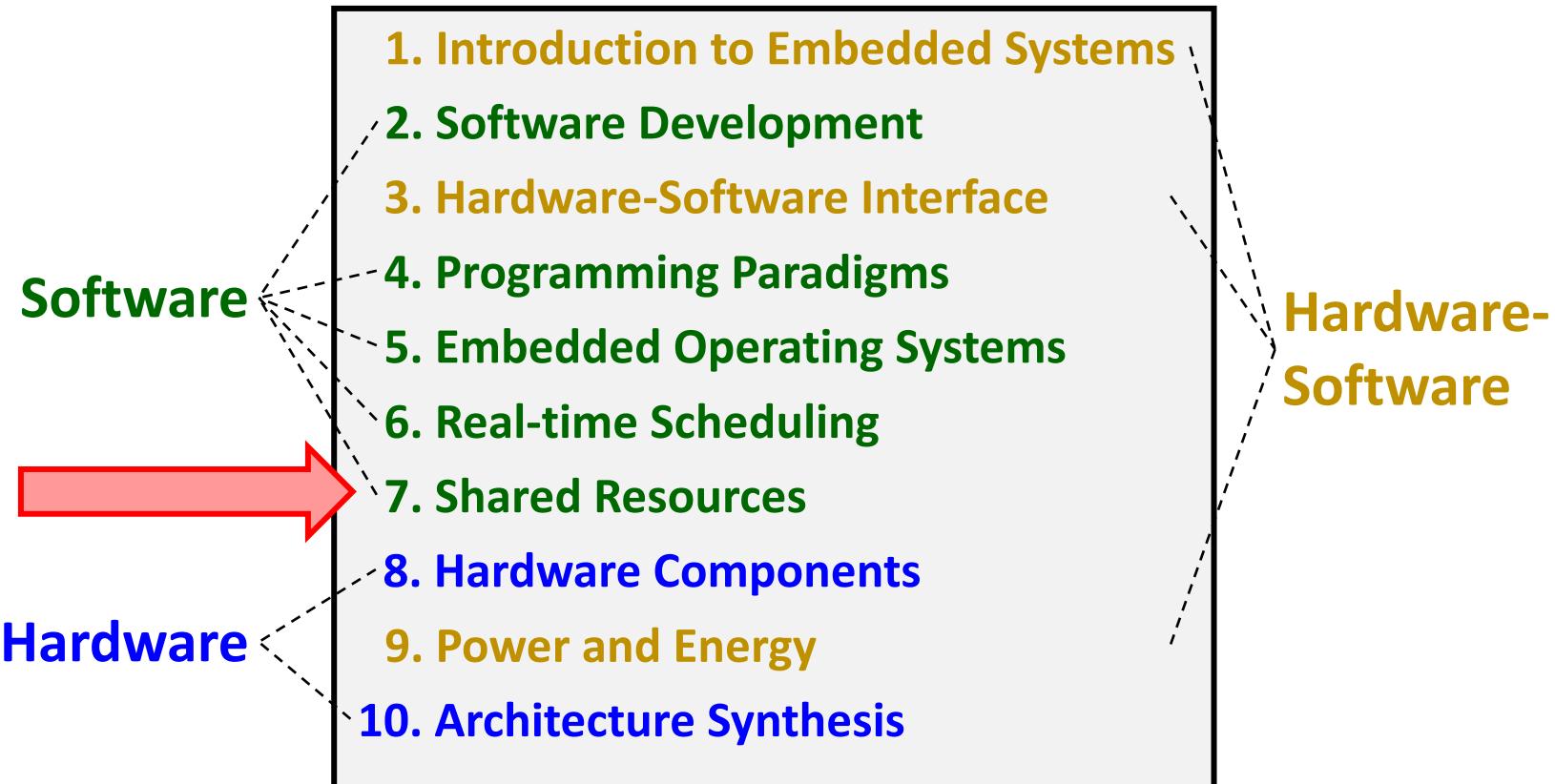
Introduction to Embedded Systems

7. Shared Resources

Prof. Dr. Marco Zimmerling



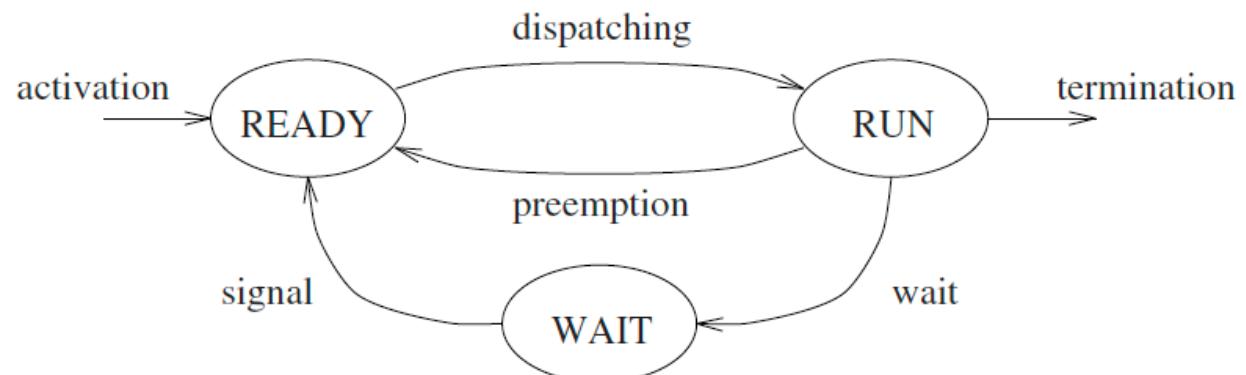
Where we are ...



Ressource Sharing

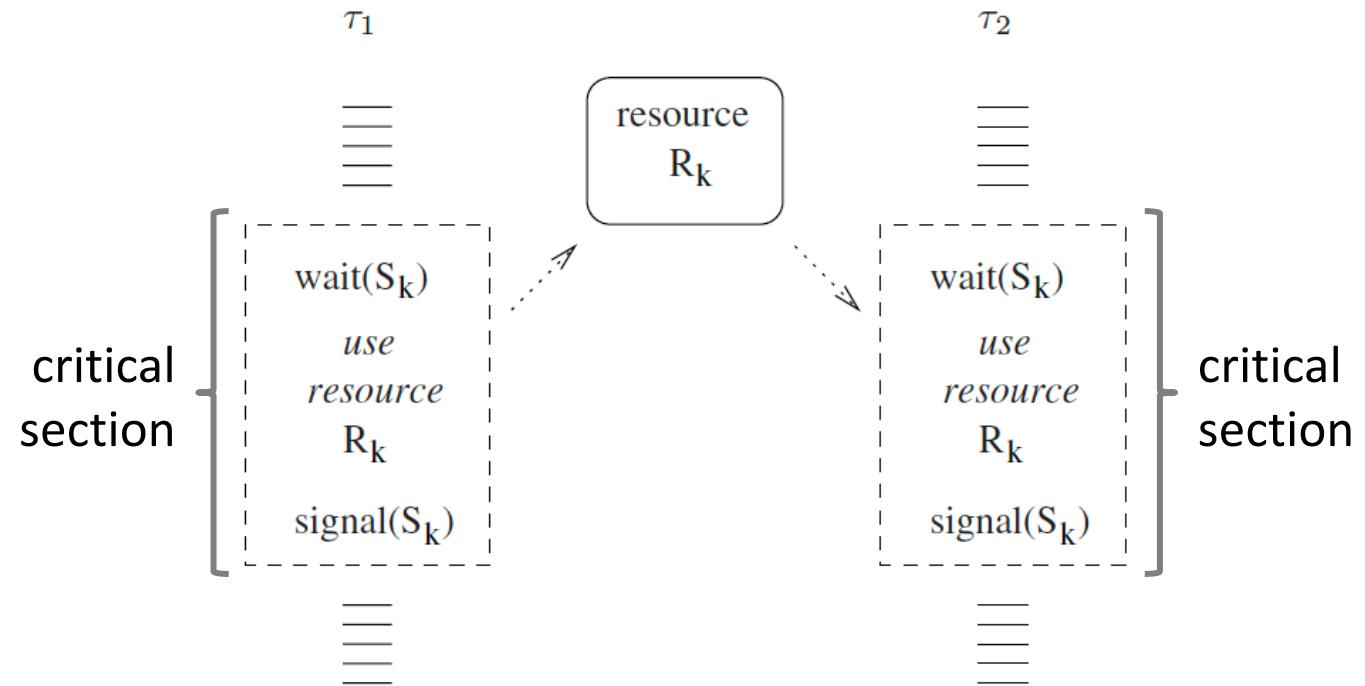
Resource Sharing

- Examples of *shared resources*: data structure, set of variables, main memory area, file, set of registers, peripheral device
- Many shared resources do not allow simultaneous accesses but require *mutual exclusion*. Such resources are called *exclusive resources*. In this case, no two tasks are allowed to operate on the resource at the same time. A piece of code executed under mutual exclusion constraints is called a *critical section*.
- There are several methods available to *protect exclusive resources*, for example,
 - *disabling interrupts* and preemption or
 - using concepts like *semaphores* that put tasks into the wait (or blocked) state if necessary.



Protecting Exclusive Resources using Semaphores

- Each *exclusive resource* R_k must be protected by a different *semaphore* S_k . Each critical section operating on a resource must begin with a `wait (S_k)` primitive and end with a `signal (S_k)` primitive.



- All tasks *blocked* on the same resource are kept in a queue associated with the semaphore. When a running task executes a `wait` primitive on a locked semaphore, it enters a waiting state until another tasks executes a `signal` primitive that unlocks the semaphore. When a task leaves the waiting state, it goes into the ready state (see state transition diagram on the previous slide).

Example: FreeRTOS

- To make ensure that 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* while accessing the shared resource:

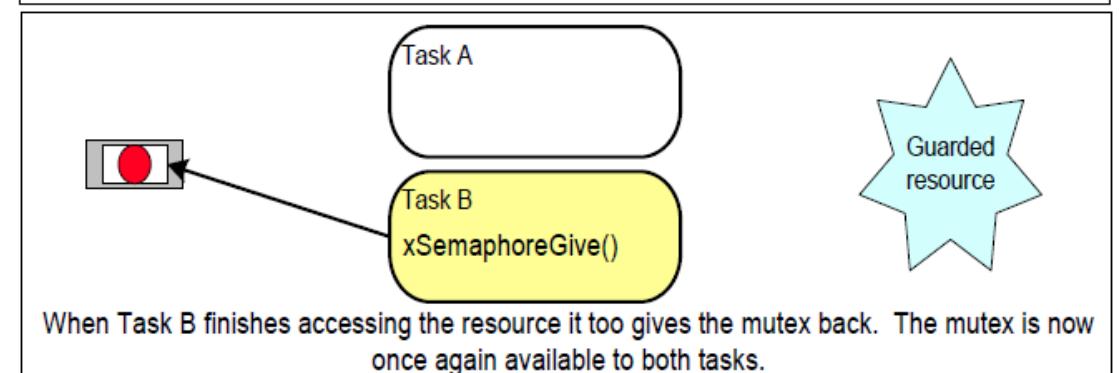
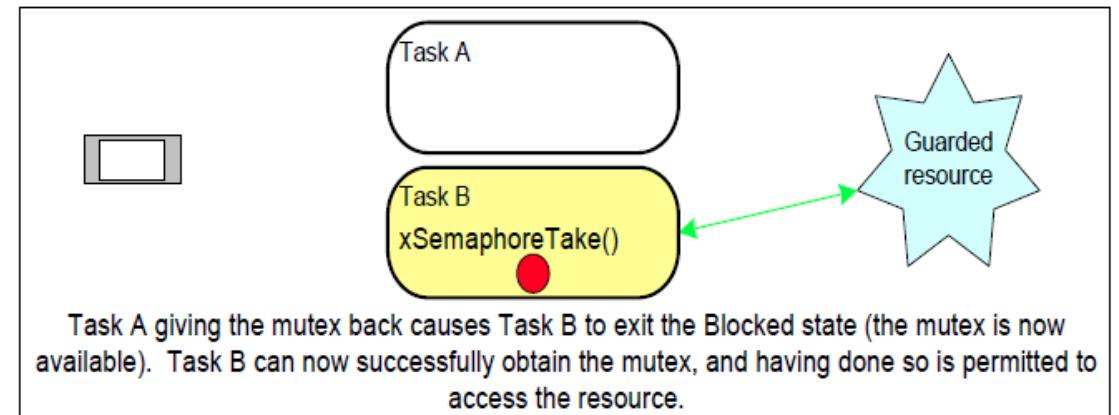
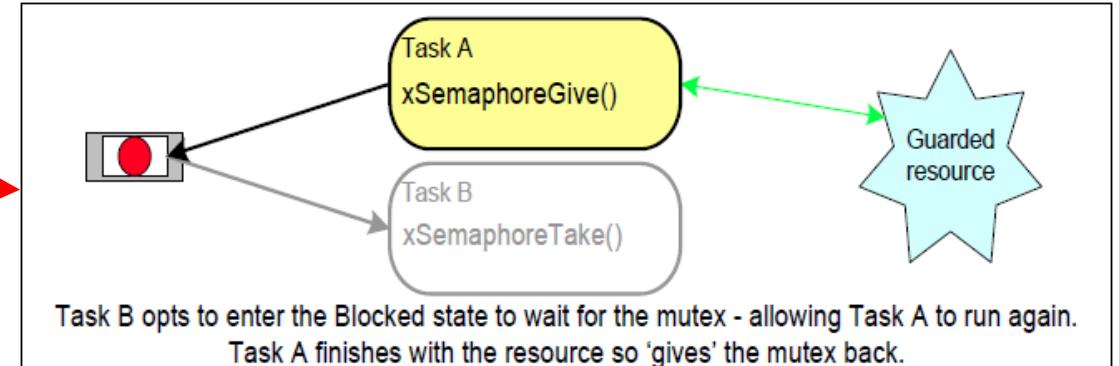
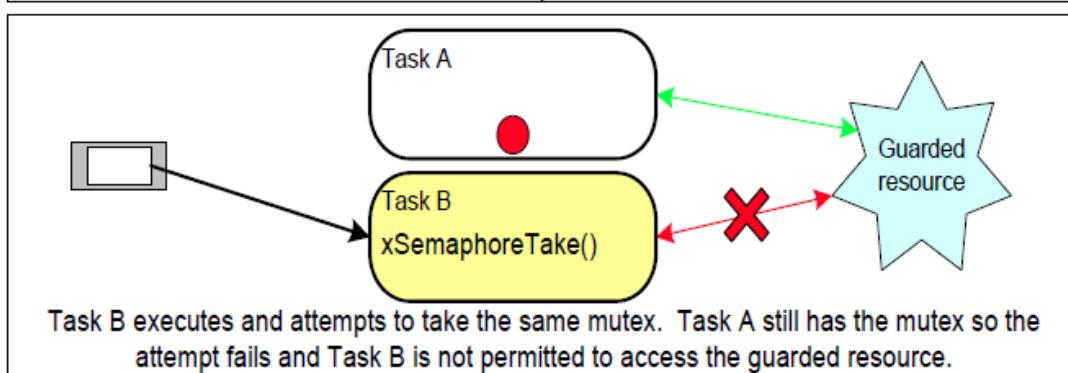
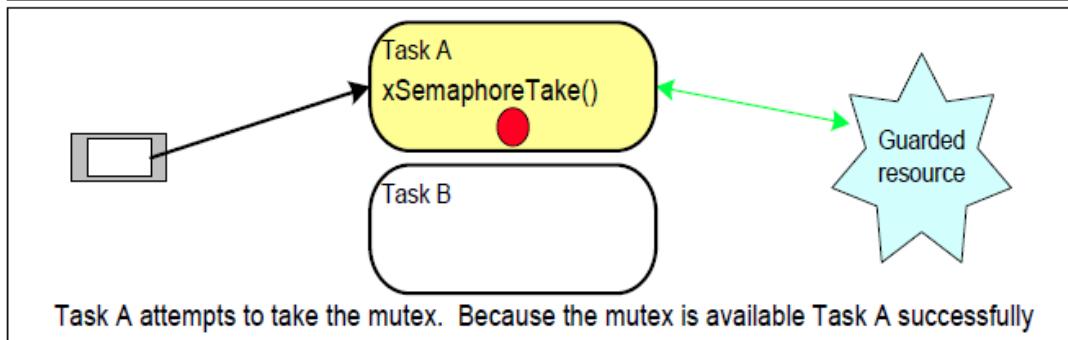
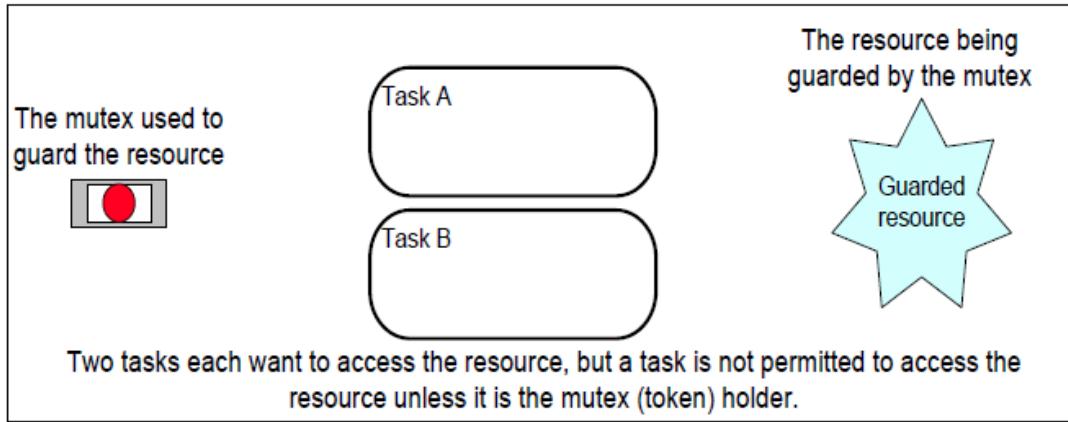
```
...
taskENTER_CRITICAL();
    ... /* access to some exclusive resource */
taskEXIT_CRITICAL();
...
```

- Such critical sections must be kept very short. Otherwise, they will adversely affect interrupt response times.

Example: FreeRTOS

- 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 shared resource.
 - When used in a mutual exclusion scenario, the mutex can be thought of as a *token* that is associated with the shared resource.
 - To access the resource, a task must first successfully *take the token*. In other words, it must be the *token holder*. When the token holder has finished accessing the shared 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



Example: FreeRTOS

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)

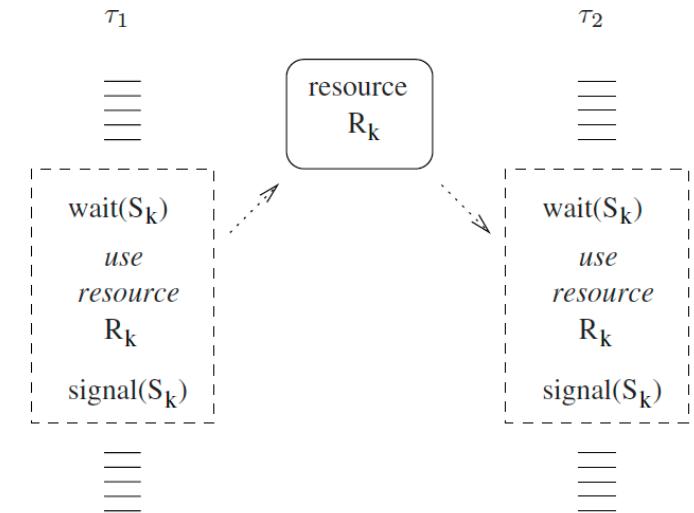
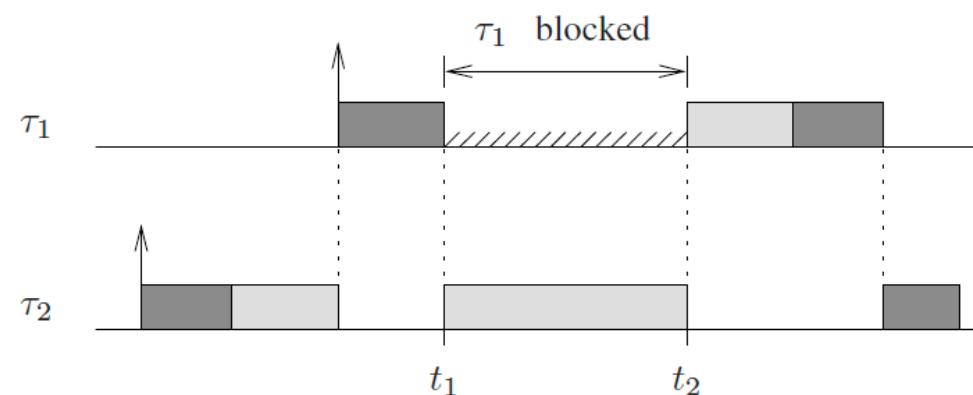
- Consider the following scenario:

- Tasks τ_1 and τ_2 share exclusive resource R_k
- τ_1 has higher priority than τ_2
- Preemption is allowed

- Example execution:

 normal execution

 critical section

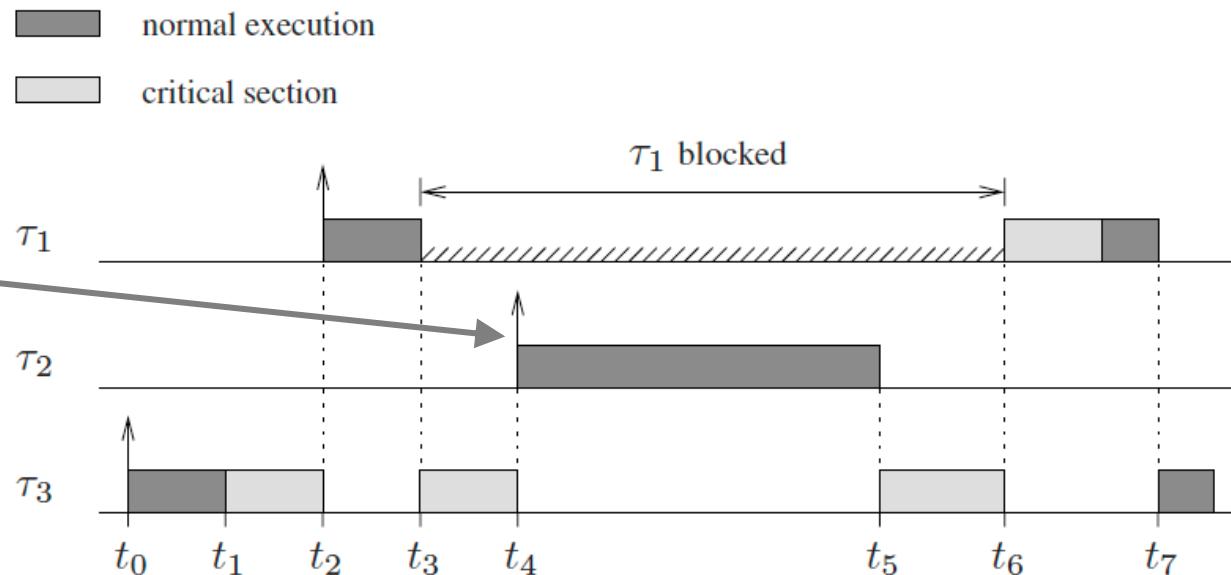


- At time t_1 task τ_1 is blocked on the semaphore, until time t_2 when task τ_2 leaves its critical section. Such blocking is *unavoidable* to guarantee mutual exclusion.

Priority Inversion (2)

- Consider the following scenario: Tasks τ_1 , τ_2 , and τ_3 have decreasing priorities, and τ_1 and τ_3 share an exclusive resource protected by a binary semaphore.
- Example execution:

Maximum time τ_1 can be blocked depends not only on the length of the critical section executed by τ_3 but *also on the WCET of task τ_2* .



- Priority inversion* is said to occur between t_3 and t_6 , since the highest-priority task τ_1 waits until the execution of lower-priority tasks (τ_2 and τ_3) finishes.
- In general, the duration of priority inversion is *unbounded* as any intermediate-priority task, which can preempt τ_3 , will indirectly block τ_1 .

Resource Access Protocols

- Several approaches, so-called *resource access protocols*, have been developed to avoid the unbounded priority inversion problem.
- For fixed-priority scheduling:
 - Non-Preemptive Protocol (NPP)
 - Priority Inheritance Protocol (PIP)
 - Priority Ceiling Protocol (PCP)
 - ...
- For dynamic-priority scheduling:
 - Stack Resource Policy (SRP)
 - ...
- *Basic idea*: Modify the priority of tasks that cause blocking. Specifically, when a task blocks one or more higher-priority tasks, it temporarily gets a higher priority.

Assumptions

- n periodic tasks $\tau_1, \tau_2, \dots, \tau_n$ cooperate via m shared resources R_1, R_2, \dots, R_m .
- Each resource R_k is guarded by a distinct binary semaphore S_k .
- Since a resource access protocol modifies the task priority, each task τ_i is characterized by a fixed *nominal priority* P_i (assigned, for example, by RM) and an *active priority* p_i ($p_i \geq P_i$), which is dynamic and initially set to P_i .
- Tasks $\tau_1, \tau_2, \dots, \tau_n$ are ordered based on their nominal priority, where τ_1 has the highest priority.
- Tasks do not suspend themselves.

Non-Preemptive Protocol (NPP)

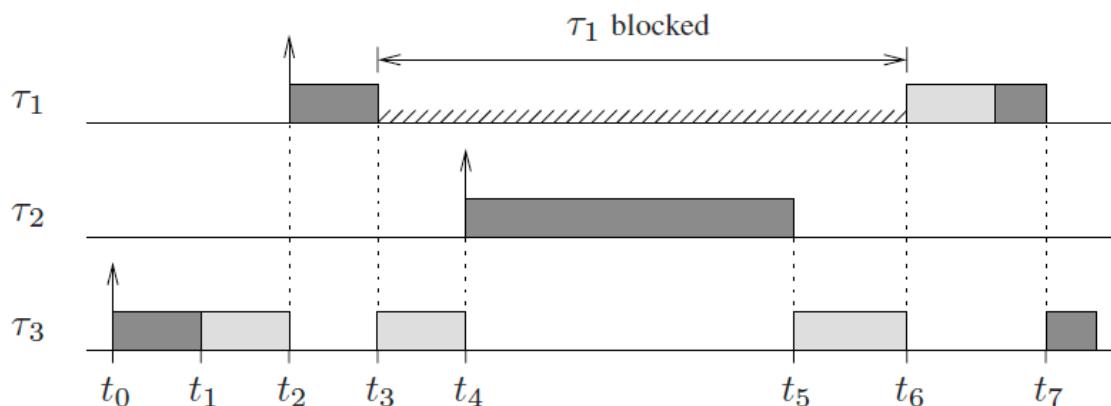
- *Idea*: Disallow preemption during the execution of any critical section
- *Principle*:
 - Raise the priority of a task to the highest priority level whenever it enters a critical section. That is, when τ_1 wants to access resource R_k , its dynamic priority is set to
$$p_i(R_k) = \max_h\{P_h\}$$
 - Tasks are scheduled based on their active priorities.
 - p_i is reset to the nominal priority P_i when the task exits the critical section.
- *Advantage*: Solves the priority inversion problem.
- *Disadvantage*: Creates unnecessary blocking for unrelated tasks.

NPP: Example (1)

- τ_1, τ_2, τ_3 have decreasing priorities. τ_1 and τ_3 share an exclusive resource.
- Example executions without NPP (left) and with NPP (right):

■ normal execution

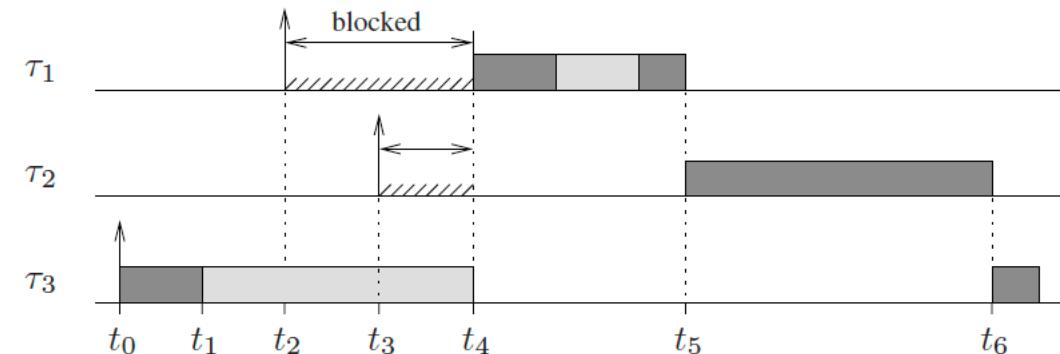
■ critical section



priority inversion occurs between time t_3 and time t_6 (this is the same execution as on slide 6-29)

■ normal execution

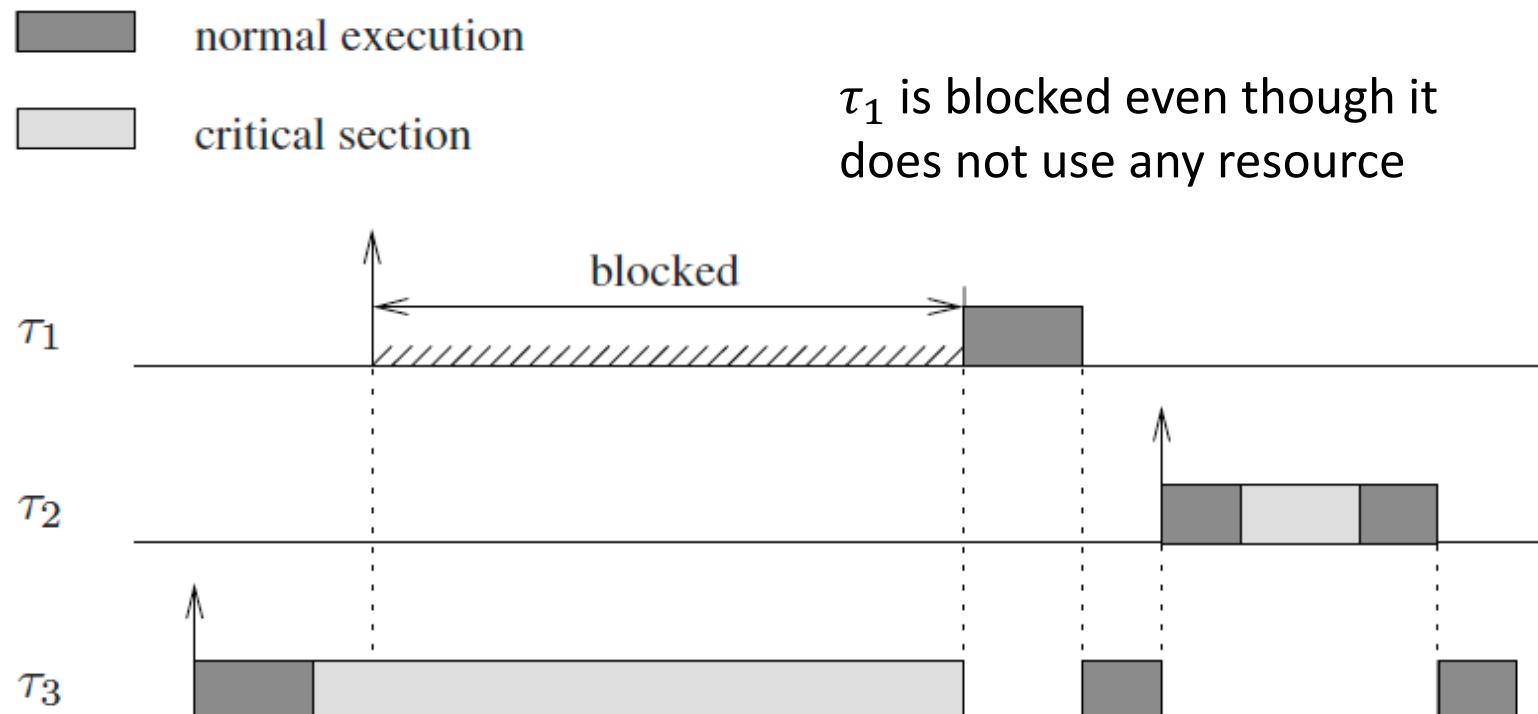
■ critical section



τ_1 and τ_2 cannot preempt τ_3 as τ_3 has the highest active priority while it is inside the critical section

NPP: Example (2)

- τ_1, τ_2, τ_3 have decreasing priorities. τ_2 and τ_3 share an exclusive resource.
- Example execution showing that NPP causes unnecessary blocking on τ_1 :

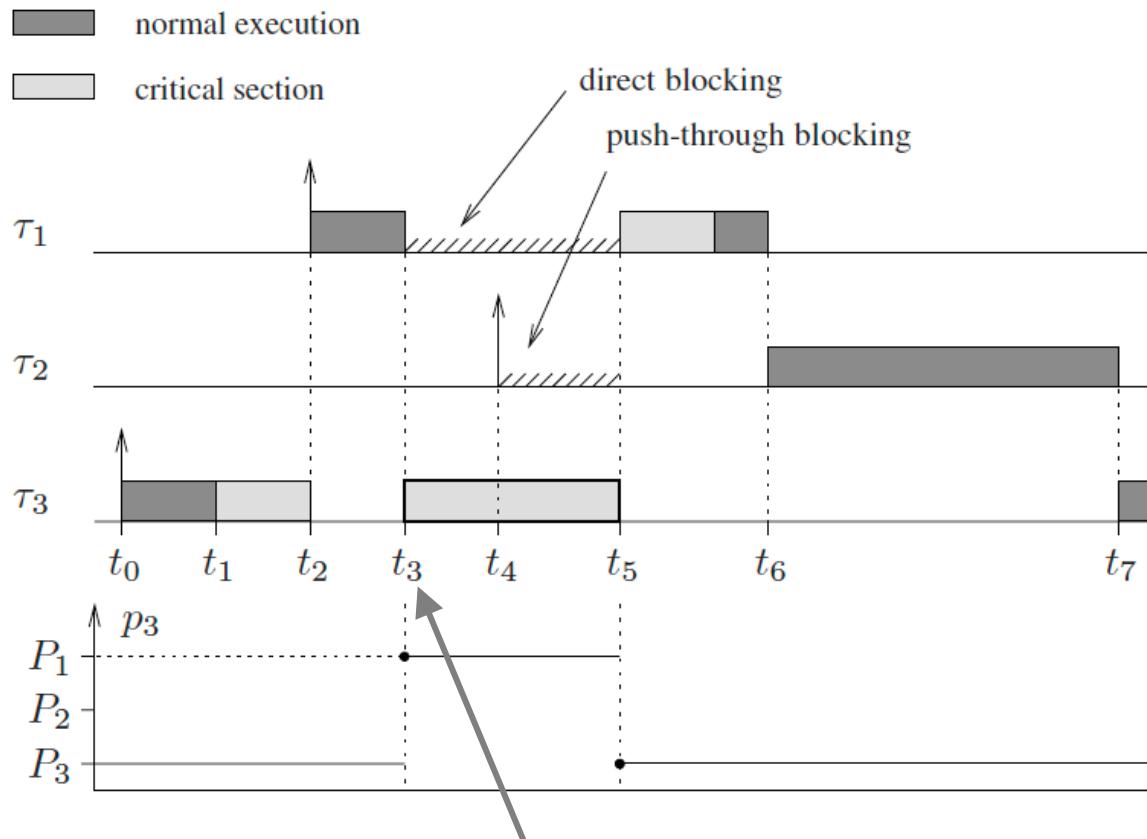


Priority Inheritance Protocol (PIP)

- *Idea:* When a task blocks one or more higher-priority tasks, it temporarily gets assigned (*inherits*) the highest priority of all blocked tasks.
- *Principle:*
 - Tasks are scheduled based on their active priorities. Tasks with the same active priority are executed in First Come First Served (FCFS) order.
 - When task τ_i tries to *enter a critical section* and the resource is already blocked by a lower-priority task τ_j , then τ_i is blocked. Otherwise, τ_i enters the critical section.
 - When task τ_i is *blocked*, it transmits its active priority to task τ_j that holds the semaphore. Thus, τ_j resumes and executes the rest of its critical section with priority $p_j = p_i$. Task τ_j is said to *inherit* the priority of τ_i .
 - When τ_j exits a critical section, it *unlocks* the semaphore and the highest-priority task blocked on that semaphore is awakened. If no other tasks are blocked by τ_j , then p_j is set to P_j . Otherwise, p_j is set to the highest priority of the tasks blocked by τ_j .
 - Priority inheritance is *transitive*. That is, if task τ_3 blocks task τ_2 , and τ_2 blocks task τ_1 , then τ_3 inherits the priority of τ_1 via τ_2 .

PIP: Example

- Again, we consider the scenario from slide 6-29. Tasks τ_1 , τ_2 , and τ_3 have decreasing priorities, and task τ_1 and τ_3 share an exclusive resource.



Direct blocking: Higher-priority task tries to acquire a resource held by a lower-priority task. Necessary to ensure the consistency of shared resources.

Push-through blocking: Medium-priority task is blocked by a lower-priority task that has inherited a higher priority from a task it directly blocks. Necessary to avoid unbounded priority inversion.

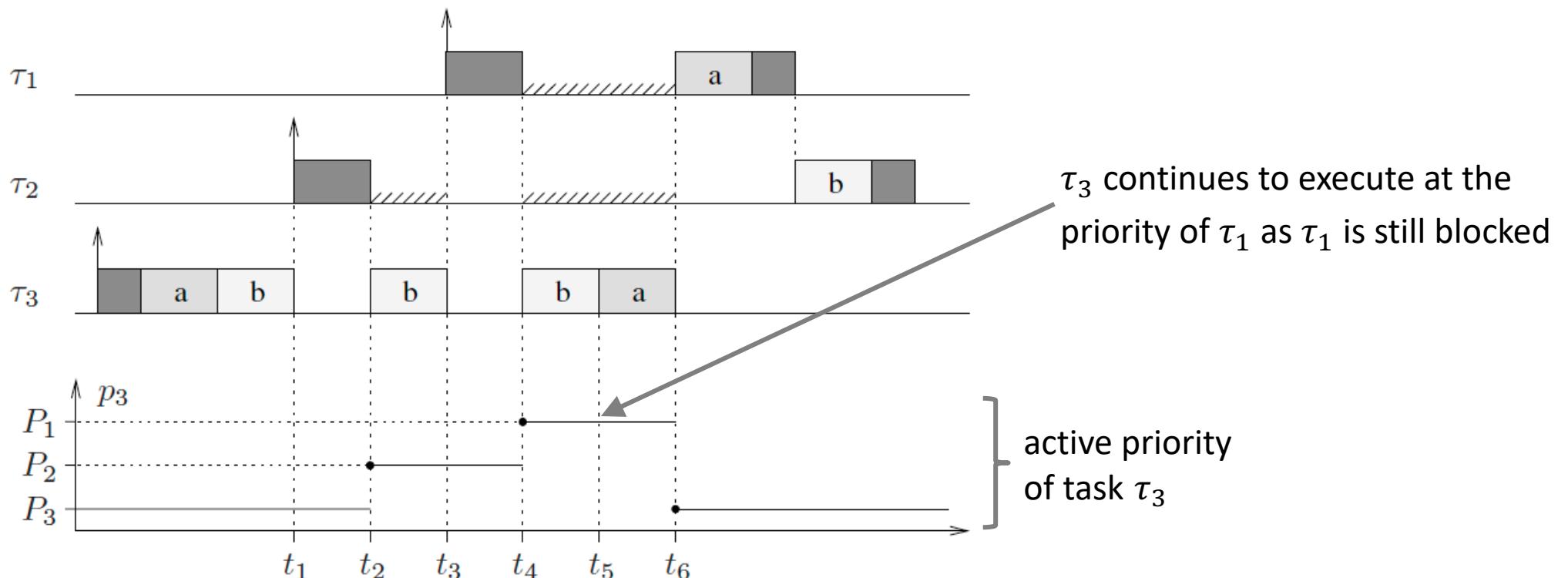
τ_1 is blocked by τ_3 , so τ_3 inherits the priority of τ_1 and executes the remaining part of its critical section *without being preempted by task τ_2 at time t_4*

PIP: Example with Nested Critical Sections

- τ_1, τ_2, τ_3 have decreasing priorities. τ_1 uses resource R_a , τ_2 uses resource R_b , and τ_3 uses both resources in a nested fashion.

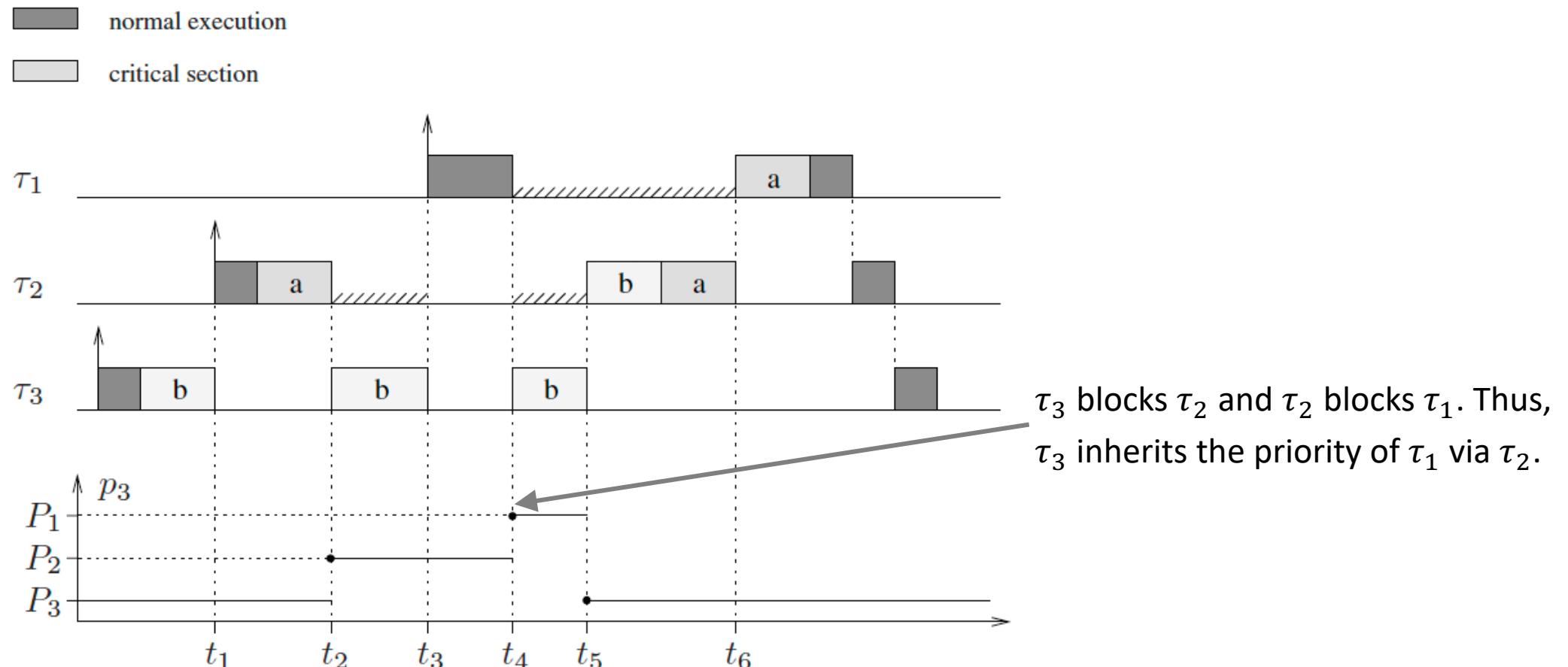
■ normal execution

■ critical section



PIP: Example of Transitive Priority Inheritance

- τ_1, τ_2, τ_3 have decreasing priorities. τ_1 uses resource R_a , τ_3 uses resource R_b , and τ_2 uses both resources in a nested fashion.



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].



Scheduling Anomalies

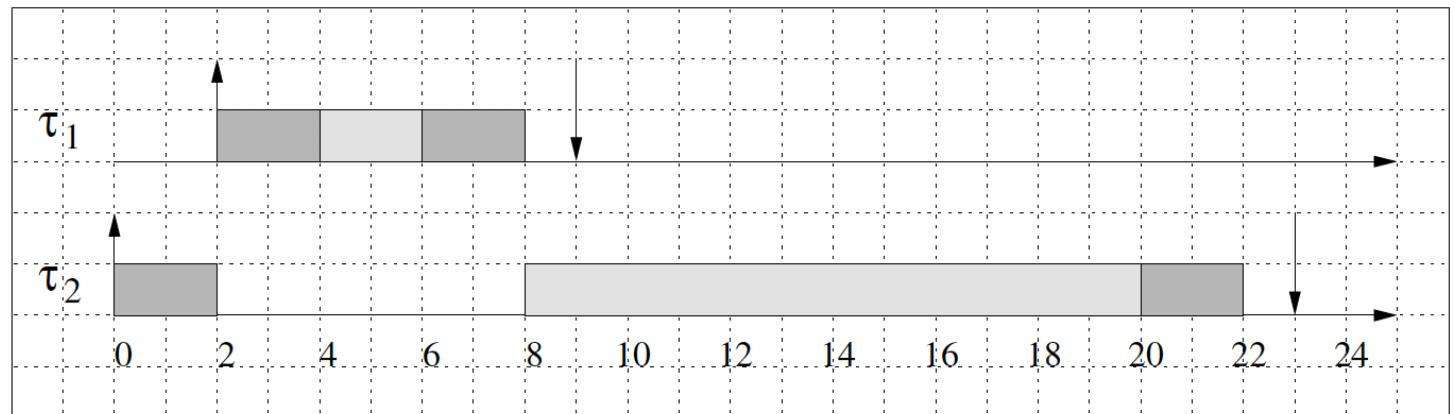
Scheduling Anomalies

- Suppose a real-time system works correctly with a given processor architecture. You replace the processor with a faster one. Are real-time constraints still satisfied?
- Unfortunately, this is not true in general. That is, *making a part of the system operate faster does not generally lead to a faster system execution*. In other words, many software and systems architectures are fragile.
- There can be many timing or scheduling anomalies in a system, from the microarchitecture (caches, pipelines, speculation) through uniprocessor scheduling to multiprocessor scheduling. We will look at a few examples in the following.

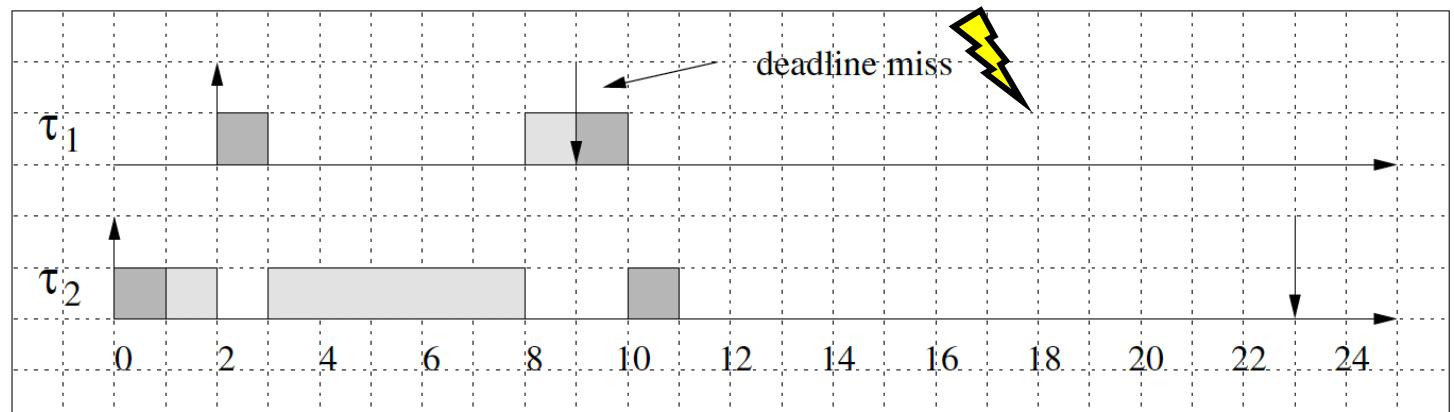
Uniprocessor Example with Critical Sections

Scenario: τ_1 has higher priority than τ_2 . Both need to access the same exclusive resource.

 normal execution
 critical section



replacing the processor with one that is twice as fast leads to a *deadline miss*



Multiprocessor Example (Richard's Anomalies)

Scenario: 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower-numbered tasks have higher priority than higher-numbered tasks. Highest-priority task is assigned to the first available processor.

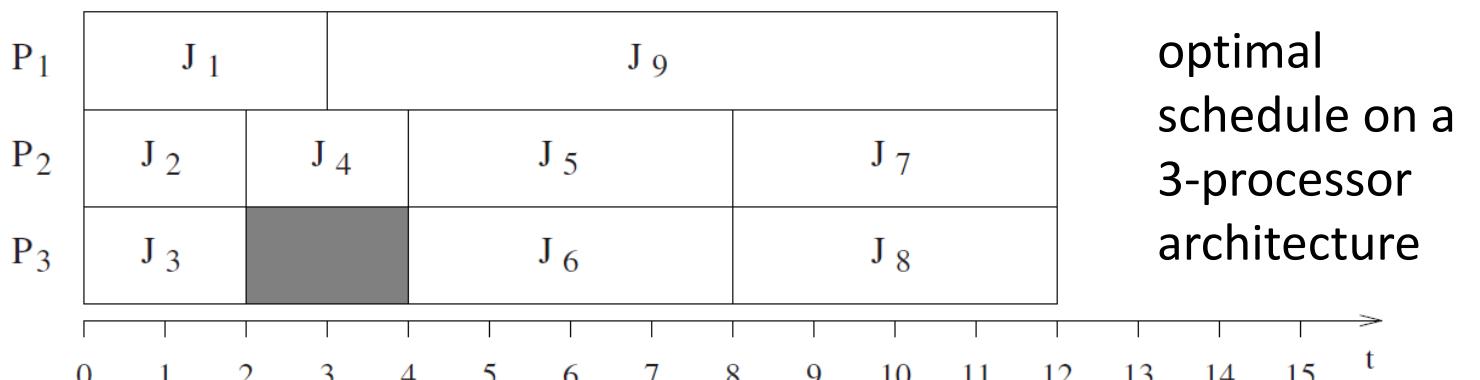
$$C_1 = 3 \quad (1) \rightarrow (9) \quad C_9 = 9$$

$$C_2 = 2 \quad (2) \rightarrow (8) \quad C_8 = 4$$

$$C_3 = 2 \quad (3) \rightarrow (7) \quad C_7 = 4$$

$$C_4 = 2 \quad (4) \rightarrow (6) \quad C_6 = 4$$

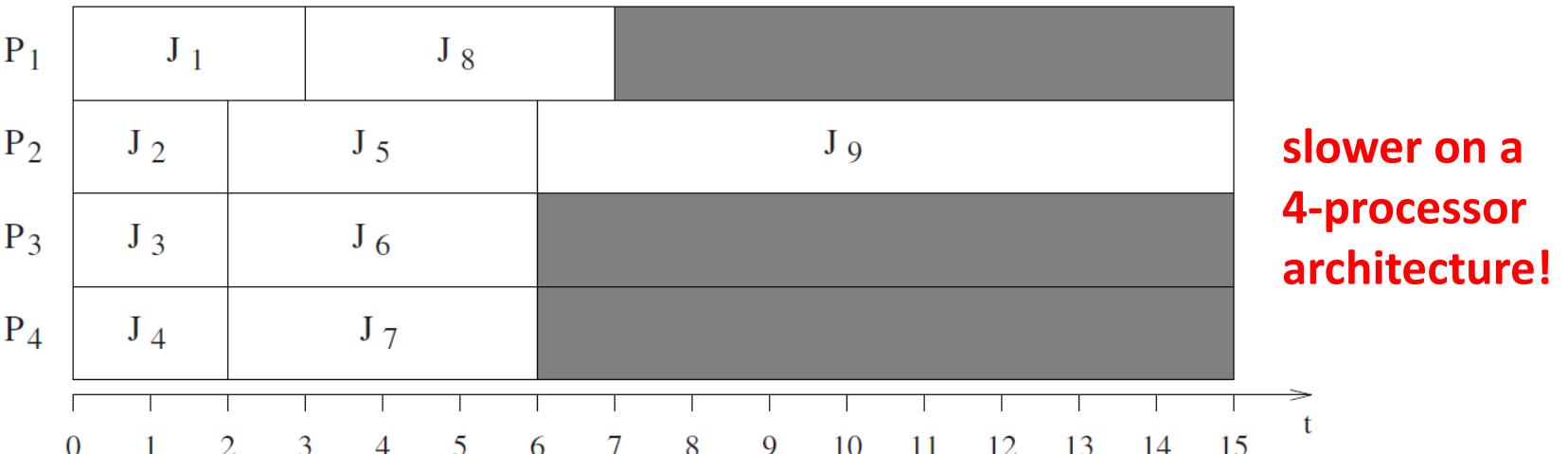
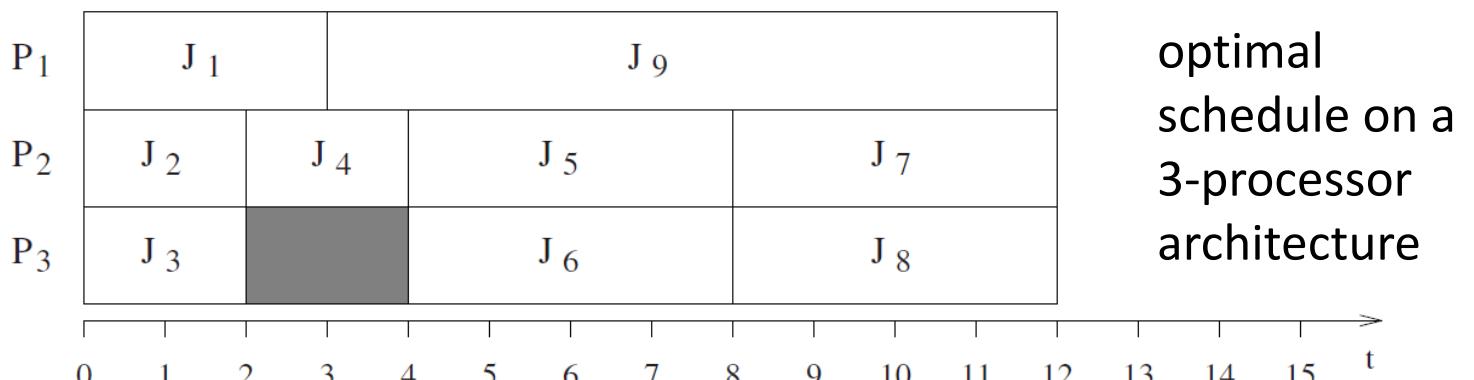
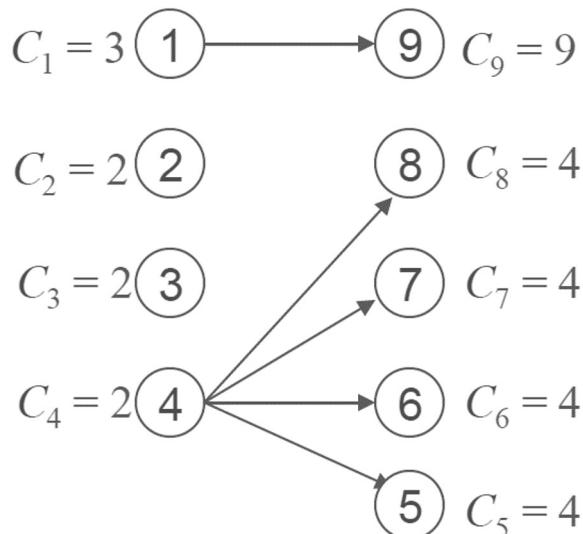
$$C_5 = 4 \quad (4) \rightarrow (5)$$



optimal
schedule on a
3-processor
architecture

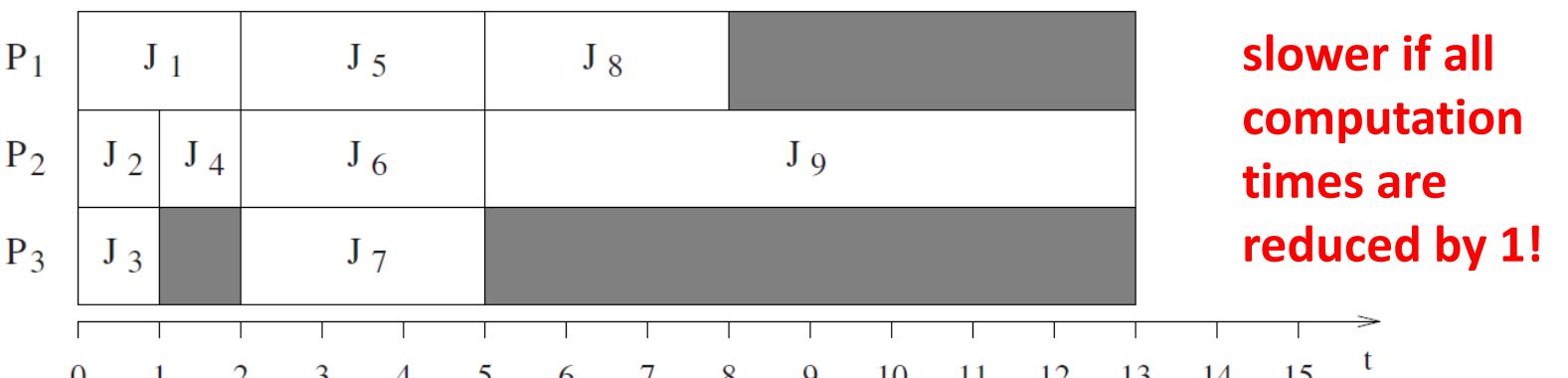
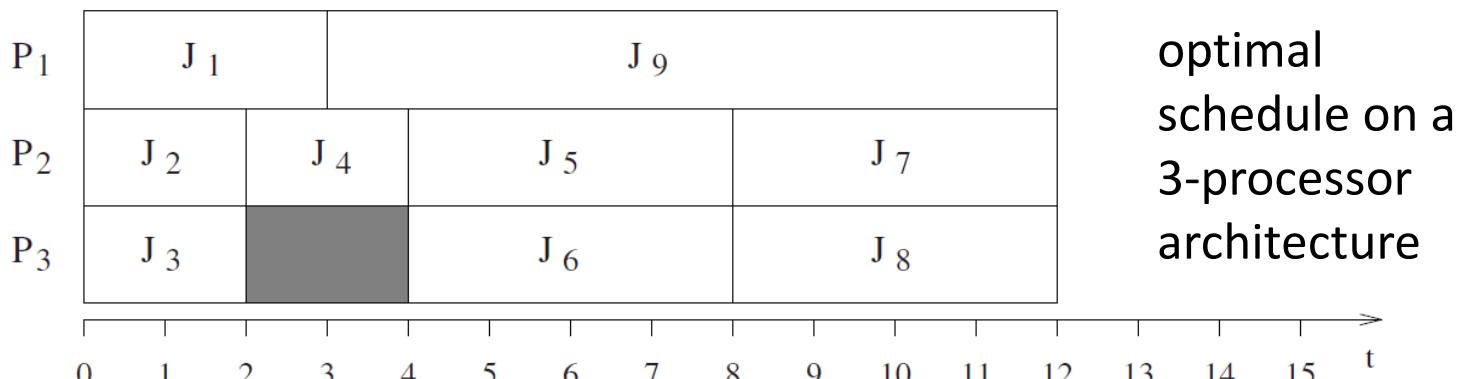
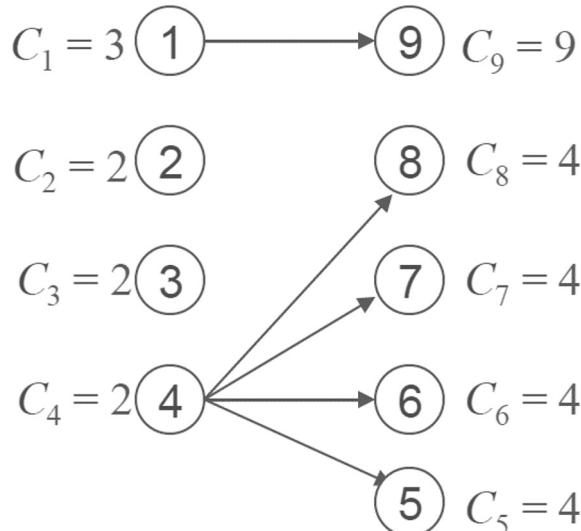
Multiprocessor Example (Richard's Anomalies)

Scenario: 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower-numbered tasks have higher priority than higher-numbered tasks. Highest-priority task is assigned to the first available processor.



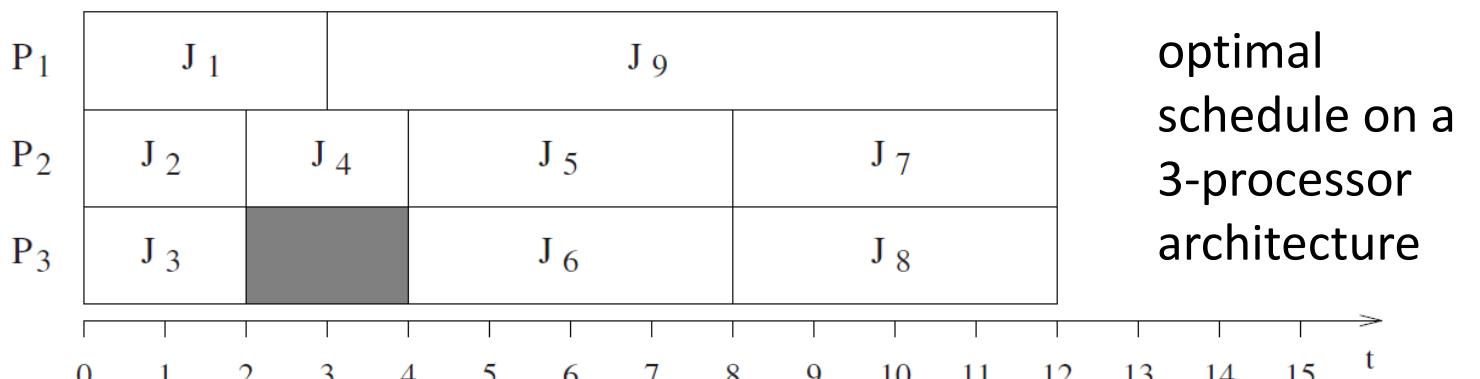
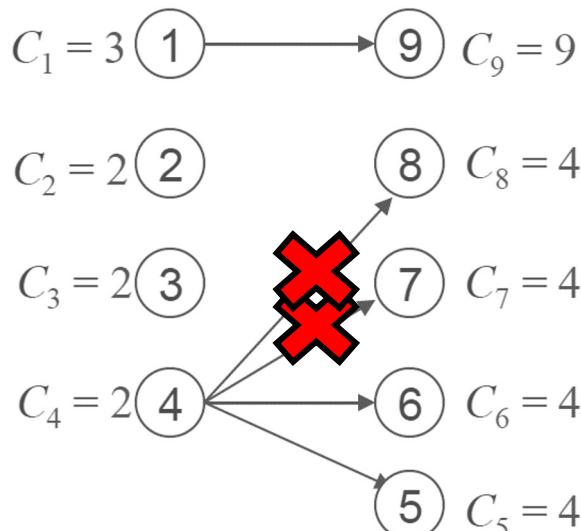
Multiprocessor Example (Richard's Anomalies)

Scenario: 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower-numbered tasks have higher priority than higher-numbered tasks. Highest-priority task is assigned to the first available processor.

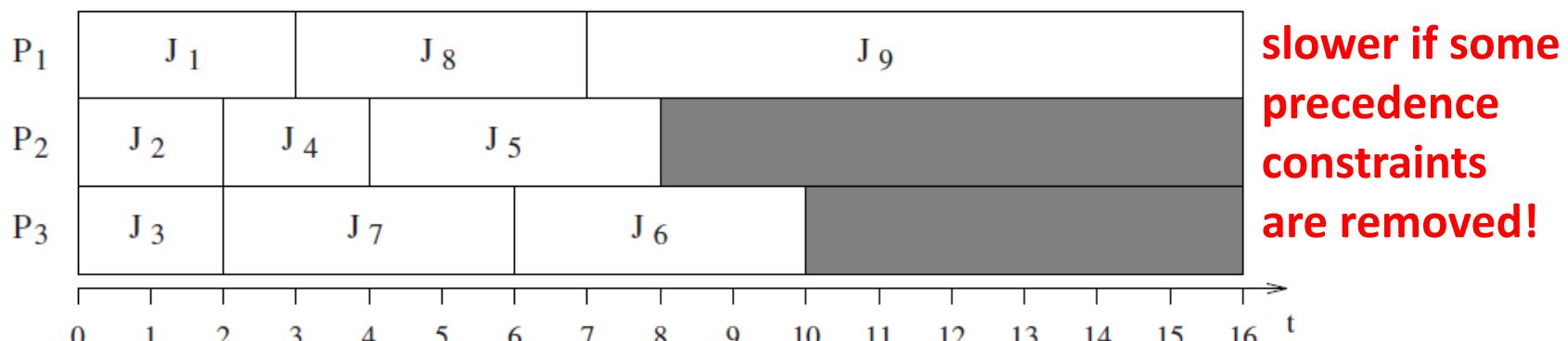


Multiprocessor Example (Richard's Anomalies)

Scenario: 9 tasks with precedence constraints and the shown execution times. Scheduling is preemptive fixed priority, where lower-numbered tasks have higher priority than higher-numbered tasks. Highest-priority task is assigned to the first available processor.



optimal
schedule on a
3-processor
architecture



slower if some
precedence
constraints
are removed!

Introduction to Embedded Systems

7. Shared Resources

Prof. Dr. Marco Zimmerling



Organization

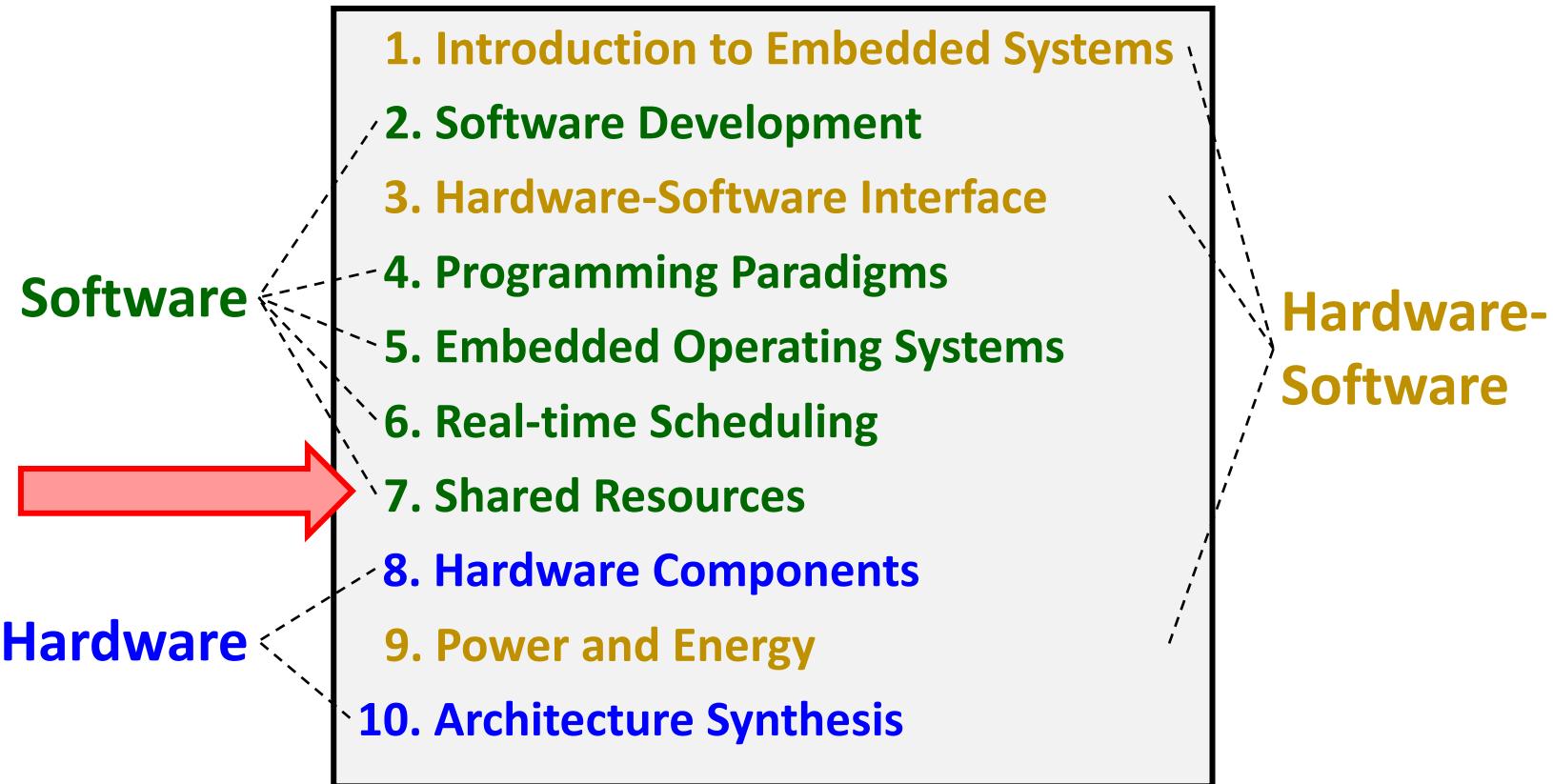
Lectures:

- Next year (January 10): again in presence (+ live streaming and recording)

Exercises:

- Due to timing constraints, we will no longer introduce the exercise sheets
- Today (December 20):
 - Solutions of fourth exercise sheet presented
 - Fifth exercise sheet released
- Next year (January 10):
 - Solutions of fifth exercise sheet presented
 - Sixth exercise sheet released
- We are working on a [mock exam](#) ...

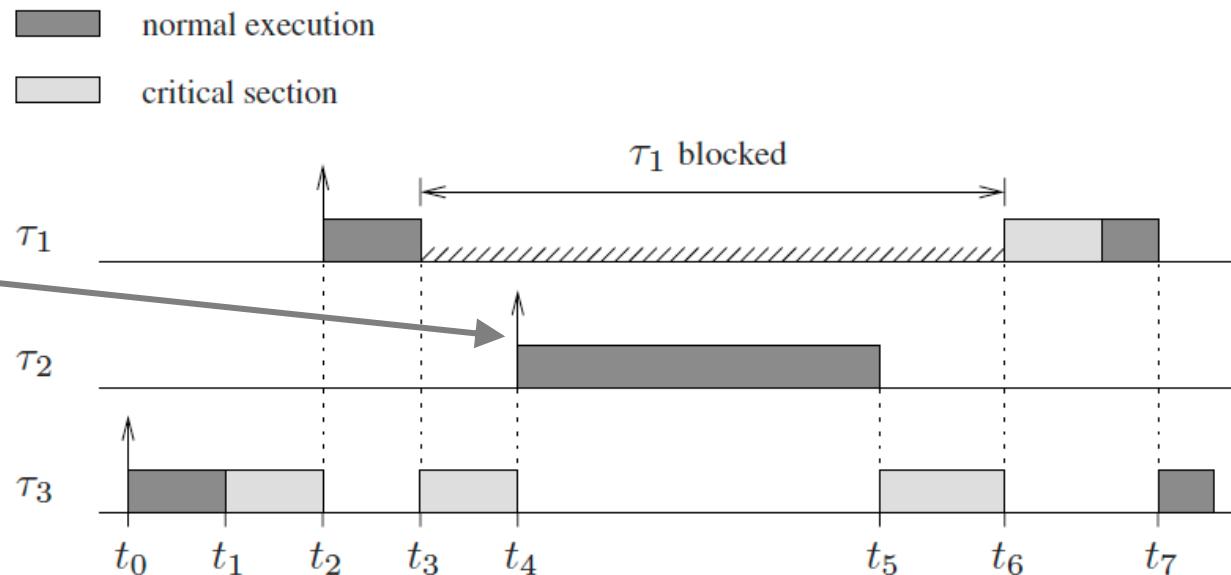
Where we are ...



Priority Inversion (2)

- Consider the following scenario: Tasks τ_1 , τ_2 , and τ_3 have decreasing priorities, and τ_1 and τ_3 share an exclusive resource protected by a binary semaphore.
- Example execution:

Maximum time τ_1 can be blocked depends not only on the length of the critical section executed by τ_3 but *also on the WCET of task τ_2* .



- Priority inversion* is said to occur between t_3 and t_6 , since the highest-priority task τ_1 waits until the execution of lower-priority tasks (τ_2 and τ_3) finishes.
- In general, the duration of priority inversion is *unbounded* as any intermediate-priority task, which can preempt τ_3 , will indirectly block τ_1 .

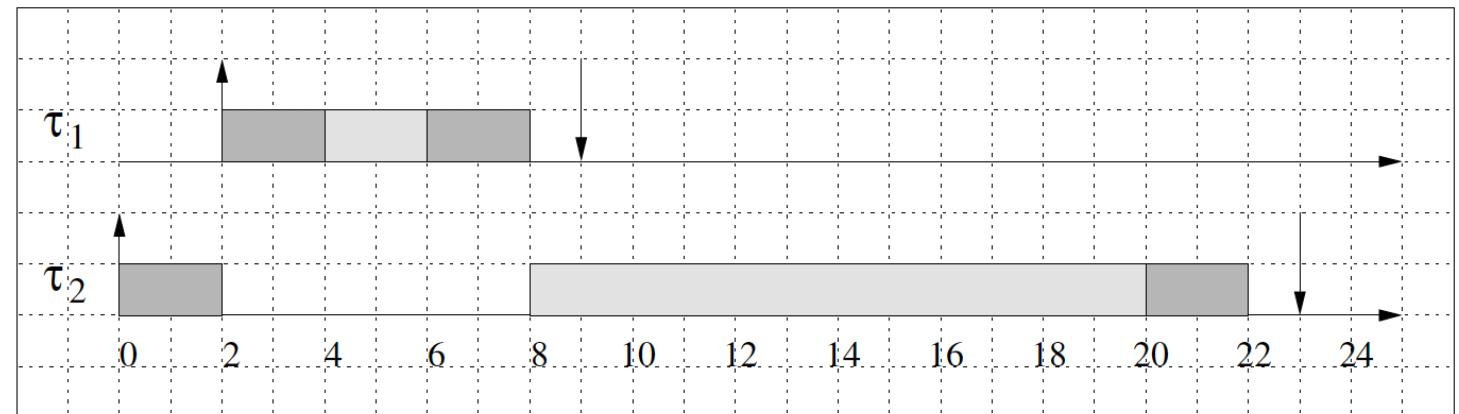
Resource Access Protocols

- Several approaches, so-called *resource access protocols*, have been developed to avoid the unbounded priority inversion problem.
- For fixed-priority scheduling:
 - Non-Preemptive Protocol (NPP)
 - Priority Inheritance Protocol (PIP)
 - Priority Ceiling Protocol (PCP)
 - ...
- For dynamic-priority scheduling:
 - Stack Resource Policy (SRP)
 - ...
- *Basic idea*: Modify the priority of tasks that cause blocking. Specifically, when a task blocks one or more higher-priority tasks, it temporarily gets a higher priority.

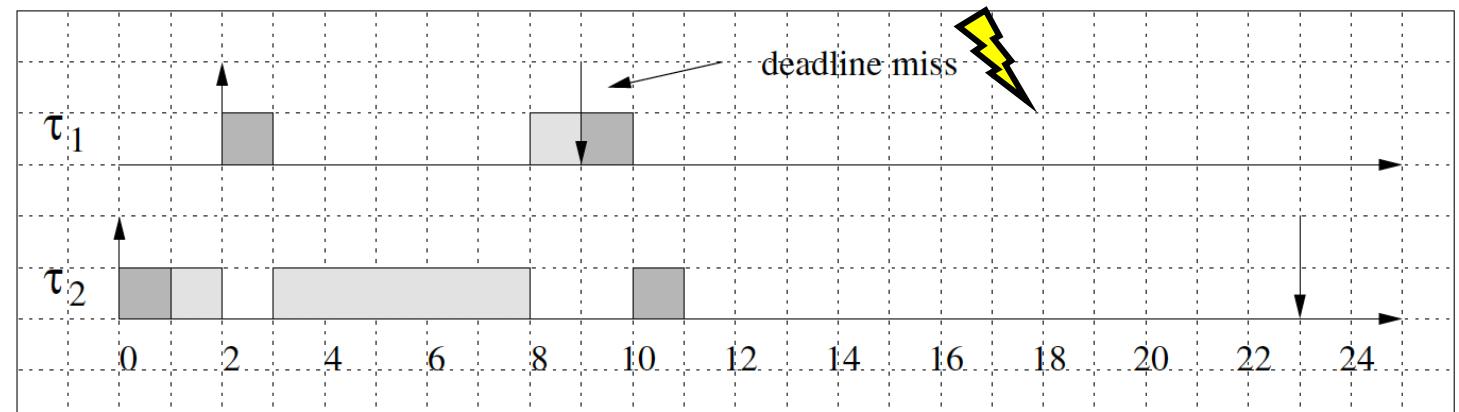
Uniprocessor Example with Critical Sections

Scenario: τ_1 has higher priority than τ_2 . Both need to access the same exclusive resource.

 normal execution
 critical section



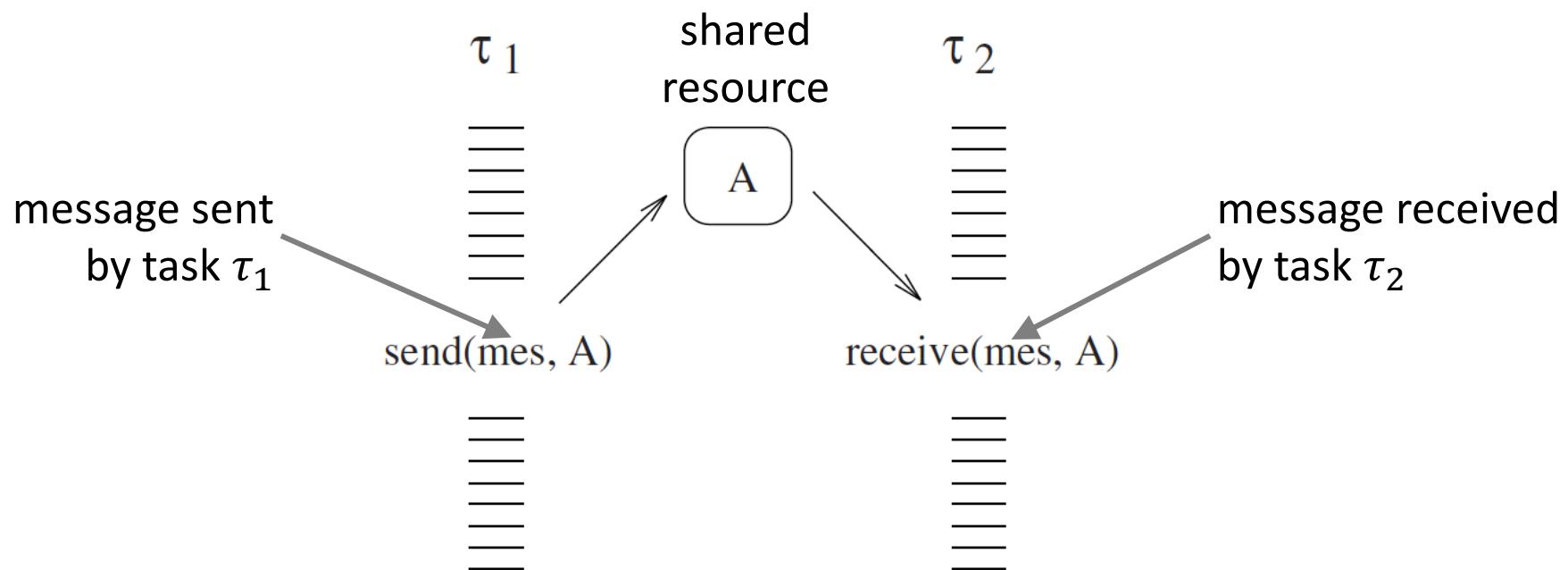
replacing the processor with one that is twice as fast leads to a *deadline miss*



Communication Between Tasks

Communication Between Tasks

- *Problem*: The use of shared memory for implementing communication between tasks may cause priority inversion and unbounded blocking of the tasks.
- Two common schemes for inter-task communication via *message passing*:
 - Synchronous
 - Asynchronous



Synchronous Communication

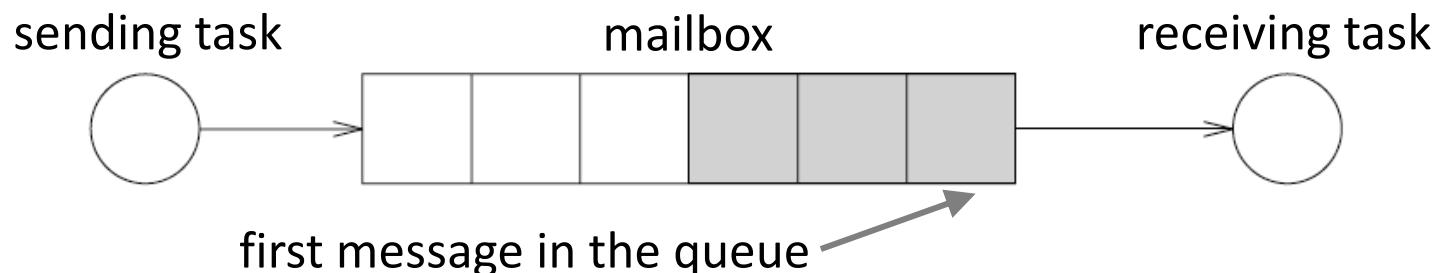
- *Principle:*
 - Two tasks that wish to communicate must be synchronized for a message transfer to take place. This synchronization is called a *rendezvous*.
 - This means that the *tasks may have to wait for each other (blocking)*:
 - If the sending task starts first, it must wait until the receiving task receives the message.
 - If the receiving task starts first, it must wait until the sending task generates the message.
- *Disadvantages:*
 - Communication always requires synchronization between the tasks. Therefore, the timing of the communicating tasks is closely linked (interdependencies).
 - In a dynamic real-time systems, estimating the maximum blocking time of a rendezvous is difficult. This can lead to unpredictable behavior.

Asynchronous Communication

- *Principle:*
 - Two tasks that wish to communicate do not have to be synchronized.
 - This means that, depending on the underlying implementation (see the following slides), the *tasks may not have to wait for each other* at all:
 - The sending task puts its message into the shared resource and continues its execution, irrespective of the current state of the receiving task.
 - The receiving task reads a message from the shared resource, irrespective of the current state of the sending task.
- *Advantages:*
 - The timing of the communicating tasks is highly independent.
 - If no unbounded delays are introduced by inter-task communication, then timing constraints can be guaranteed without increasing the complexity of the system.
 - Hence, asynchronous communication is more suited for dynamic real-time systems.

Mailbox Mechanism

- Common way to implement asynchronous communication:
 - *Mailbox* = shared memory buffer with given capacity (i.e., max. number of messages)
 - Messages are typically kept in a First In First Out (FIFO) queue
 - Two basic operations are `send (mbox, msg)` and `receive (mbox, msg)`
 - `send` inserts message `msg` into the queue of mailbox `mbox`
 - `receive` extracts first message from the queue of mailbox `mbox` and stores it in `msg`



- *Problem:*
 - Sending task is blocked if the queue is full
 - Receiving task is blocked if the queue is empty

Cyclic Asynchronous Buffer (CAB)

- *Principle*: CAB always contains the last message that was written into it
 - A receiving task does not extract a message. Instead, a message is kept in the CAB until it is overwritten by the sending task that inserts a new message into the CAB.
 - This enables *non-blocking communication* once the first message has been inserted:
 - Sending task does not block as a new message overwrites the old one (CAB is never full)
 - Receiving task does not block as it can always read out a message (CAB is never empty)
- *Properties*:
 - A message can be read out more than once. This can be detected by a receiving task.
 - A message can also be lost if it is overwritten before it was read out. This is not a problem in applications that are only interested in the latest data (e.g., control).

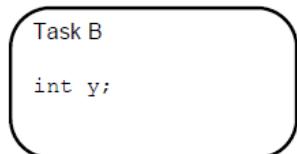
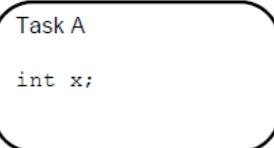
writing

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

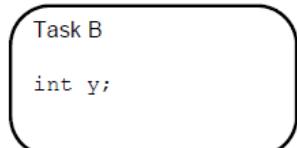
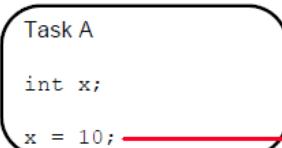
reading

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

Example: FreeRTOS

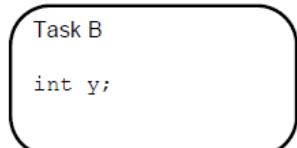
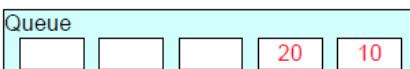
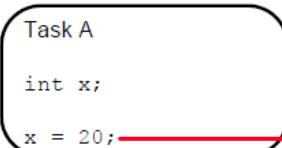


A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



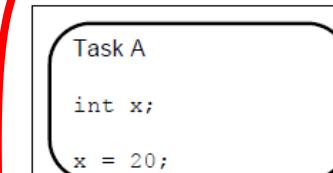
Send

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.

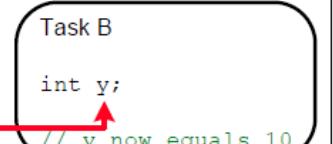


Send

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.

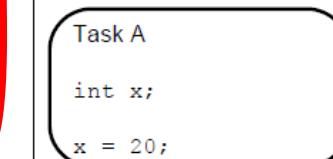


x = 20;

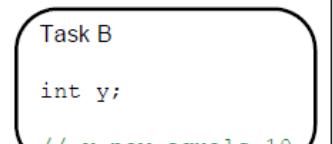


// y now equals 10

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).



x = 20;



// y now equals 10

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

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 );
```

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

a pointer to the
data to be copied
into the queue

Example: FreeRTOS

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

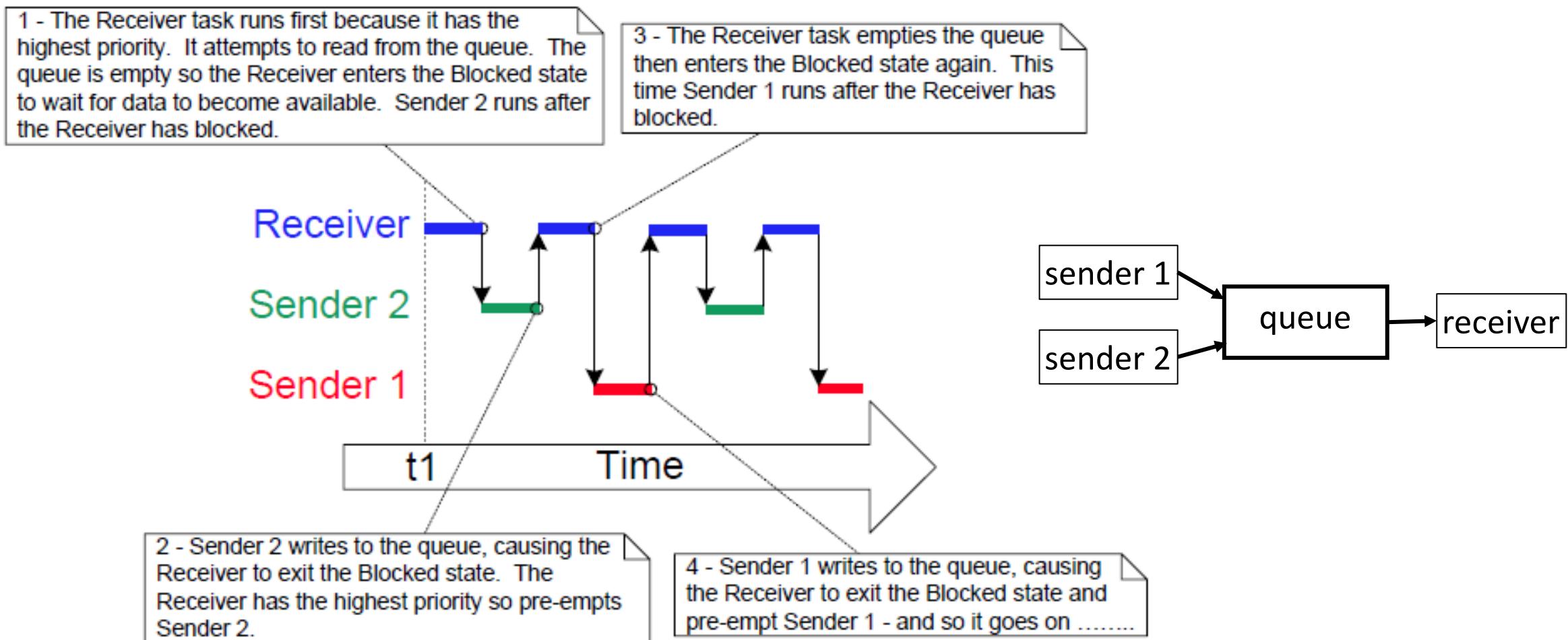
a pointer to the
memory into which
the received data
will be copied

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

Example cont.:



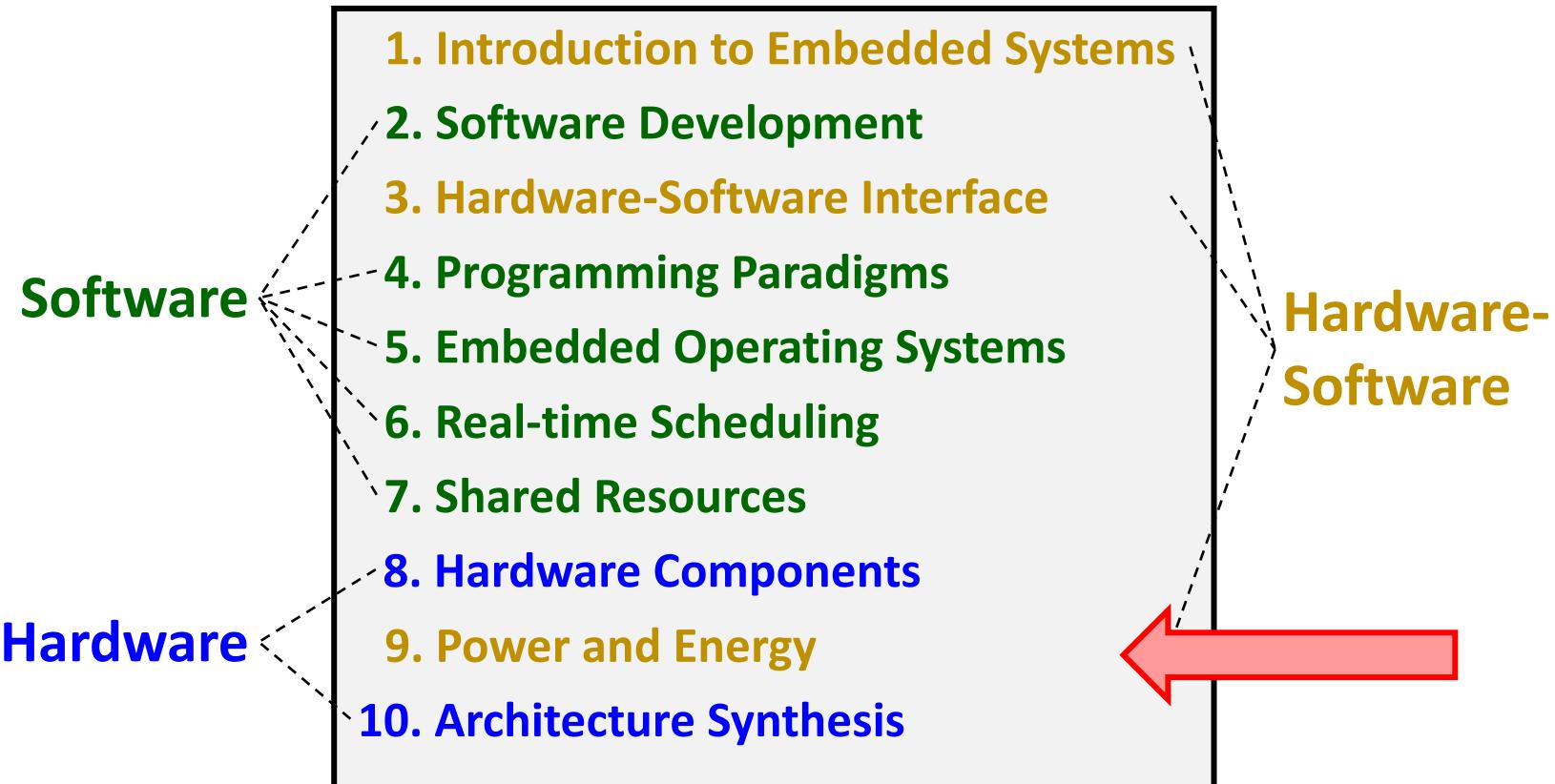
Introduction to Embedded Systems

9. Power and Energy

Prof. Dr. Marco Zimmerling



Where we are ...



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. Eggemont (ed): Embedded Systems Roadmap 2002, STW]

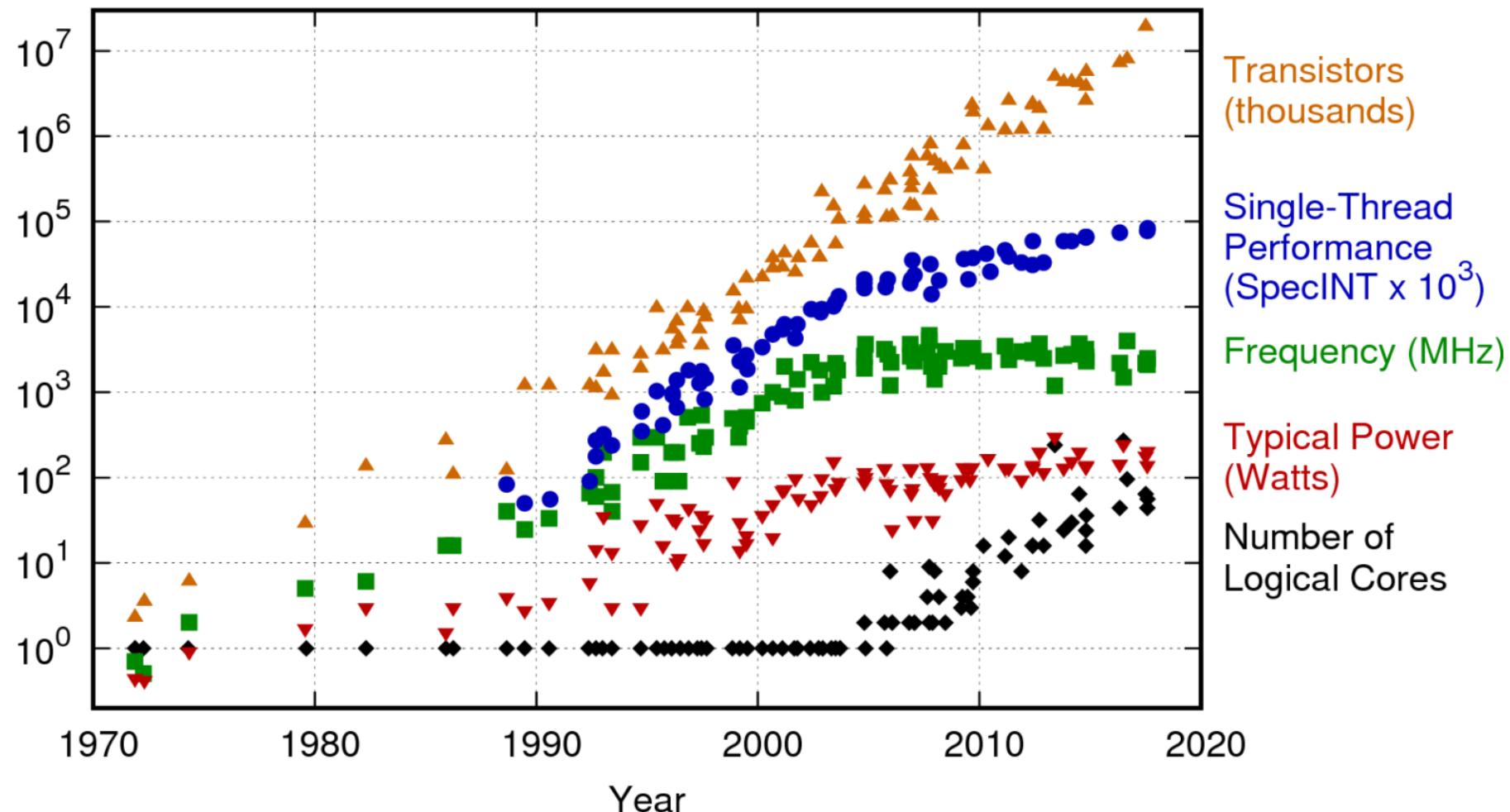
“Power demands are increasing rapidly, yet battery capacity cannot keep up.” [in Ditzel et al.: Power-Aware Architecting for data-dominated applications, 2007, Springer]

- **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 low energy density

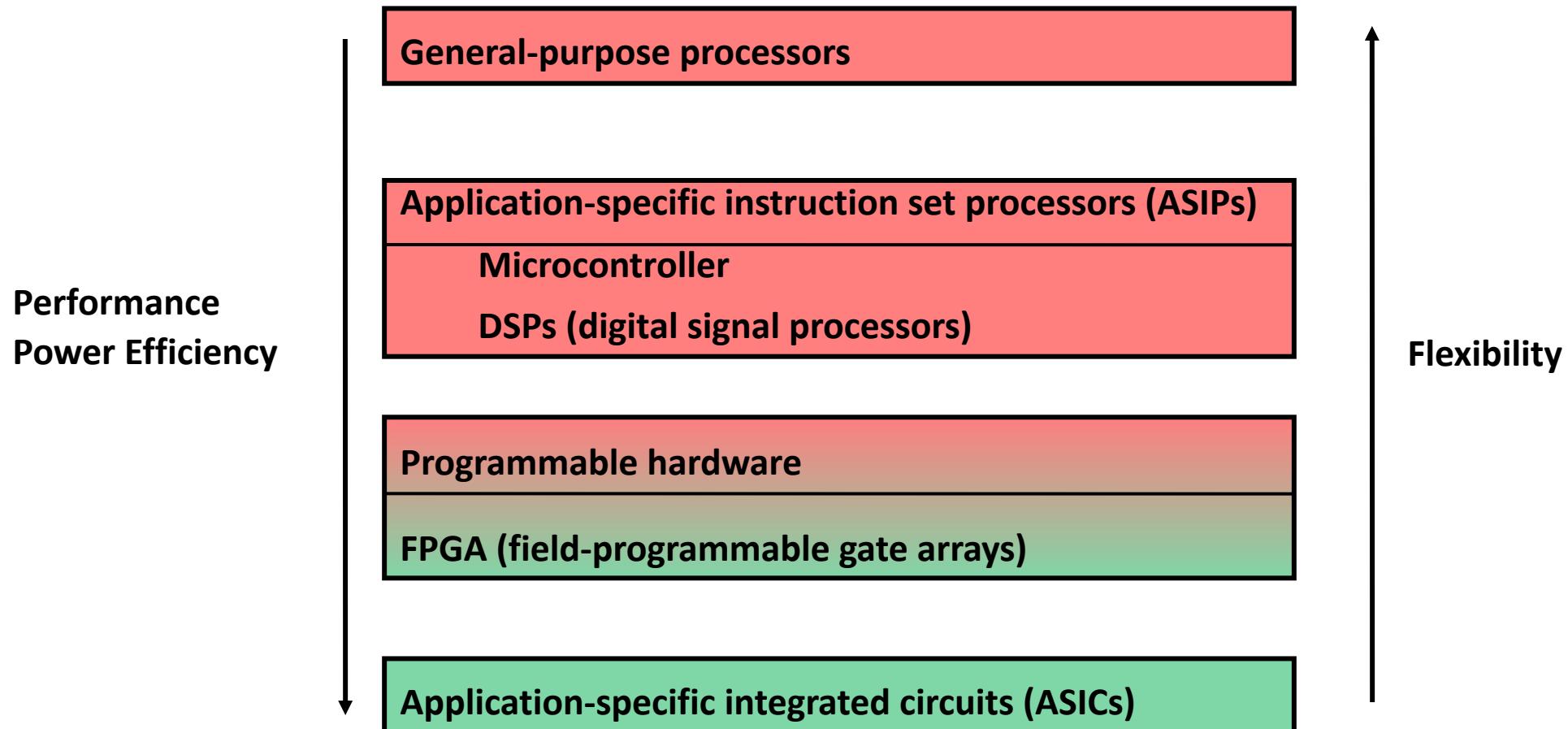


Some Trends



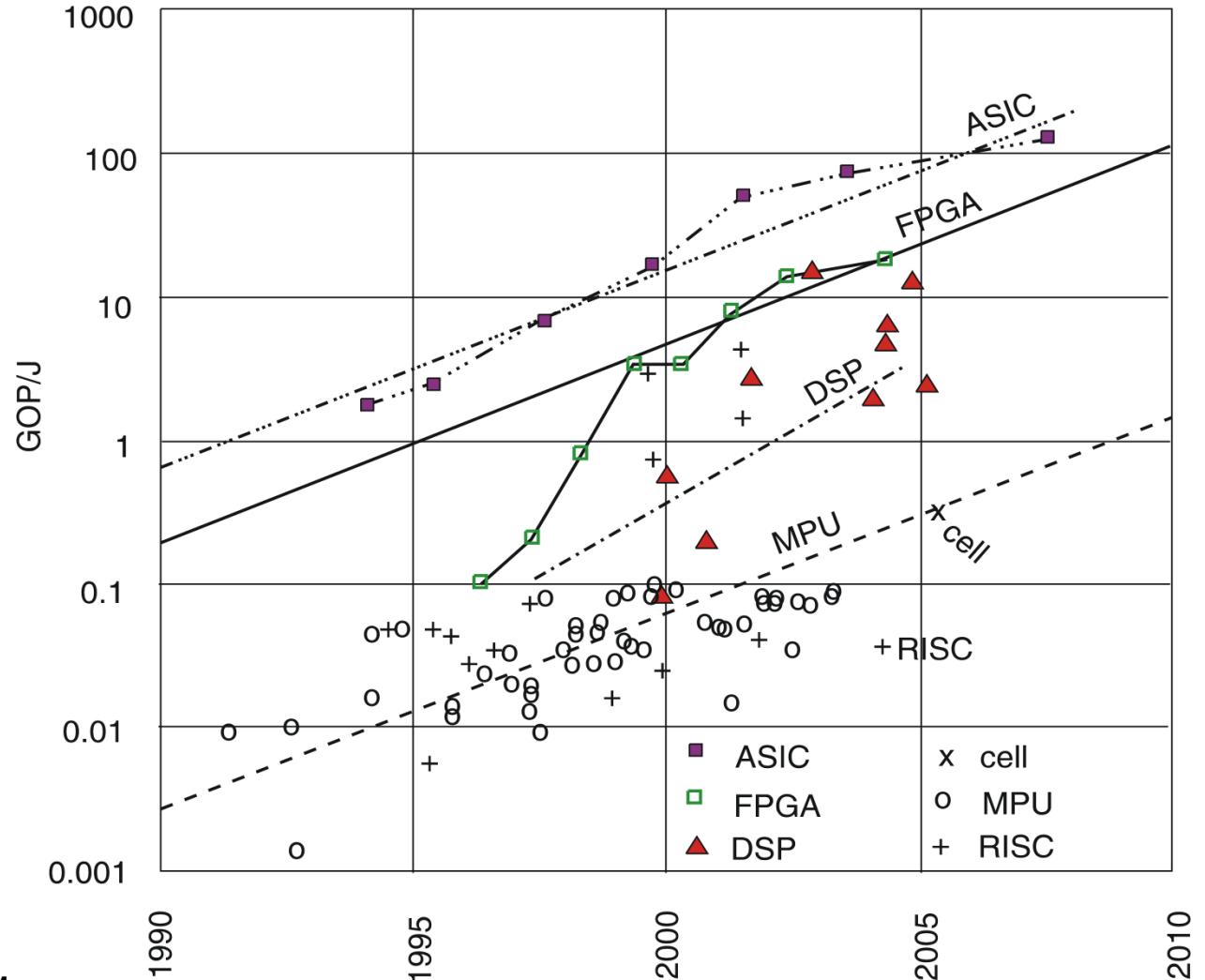
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Implementation Alternatives



Energy Efficiency

- It is necessary to *optimize HW and SW*.
- Use *heterogeneous architectures* in order to adapt to required performance and application.
- Apply *specialization techniques*:
 - Higher parallelism
 - Turn off unused components (e.g., low-power modes)
 - Higher heterogeneity
 - Voltage and frequency scaling



Power and Energy

Power and Energy

$$E = \int P(t)dt = \int V(t) \cdot I(t)dt$$

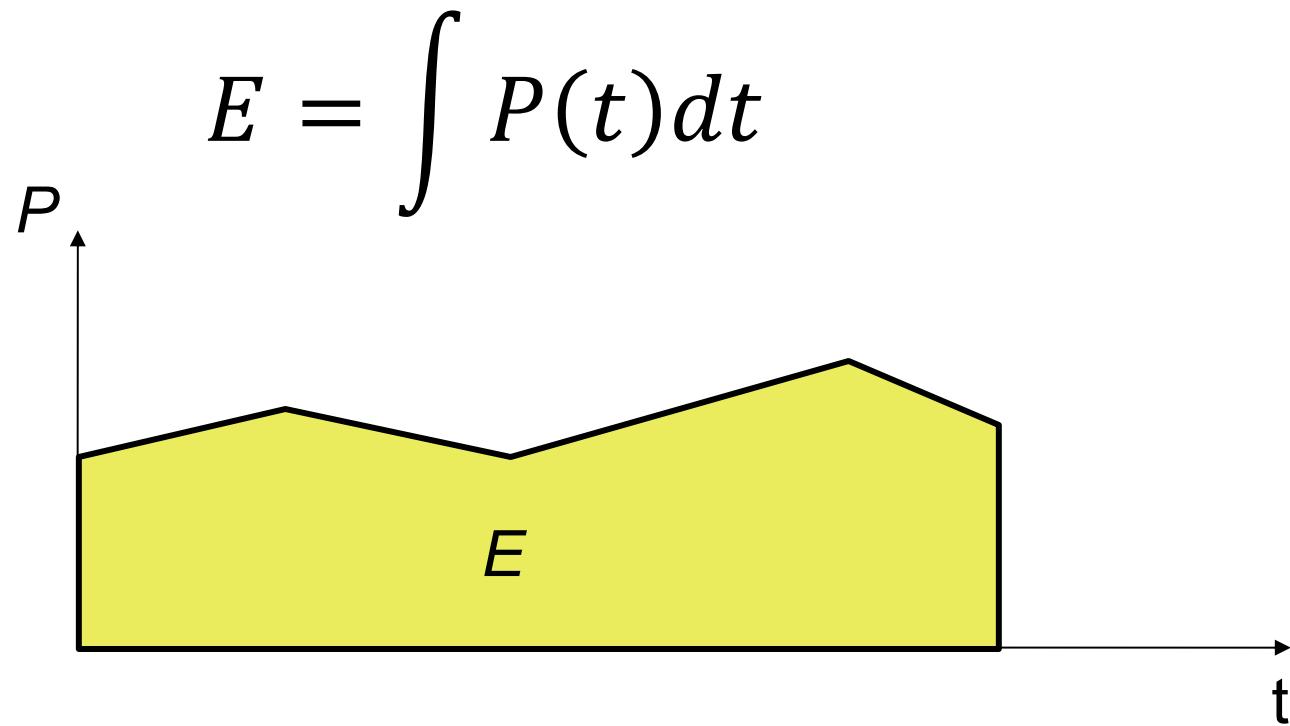


Power is an *instantaneous quantity* measured at time t

Energy is consumed *by a given task* over a certain time interval

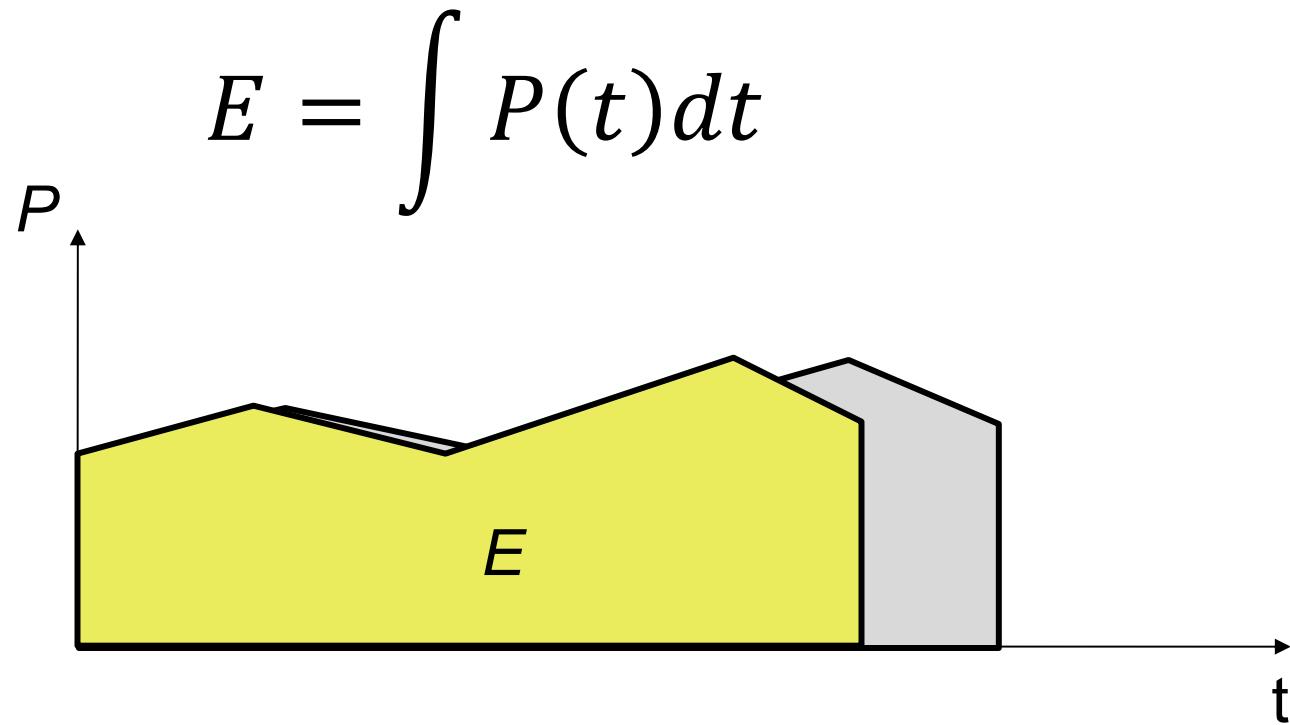
Because energy is not an instantaneous quantity, it is important to specify for *what* a given amount of energy is used (context)!

Power and Energy



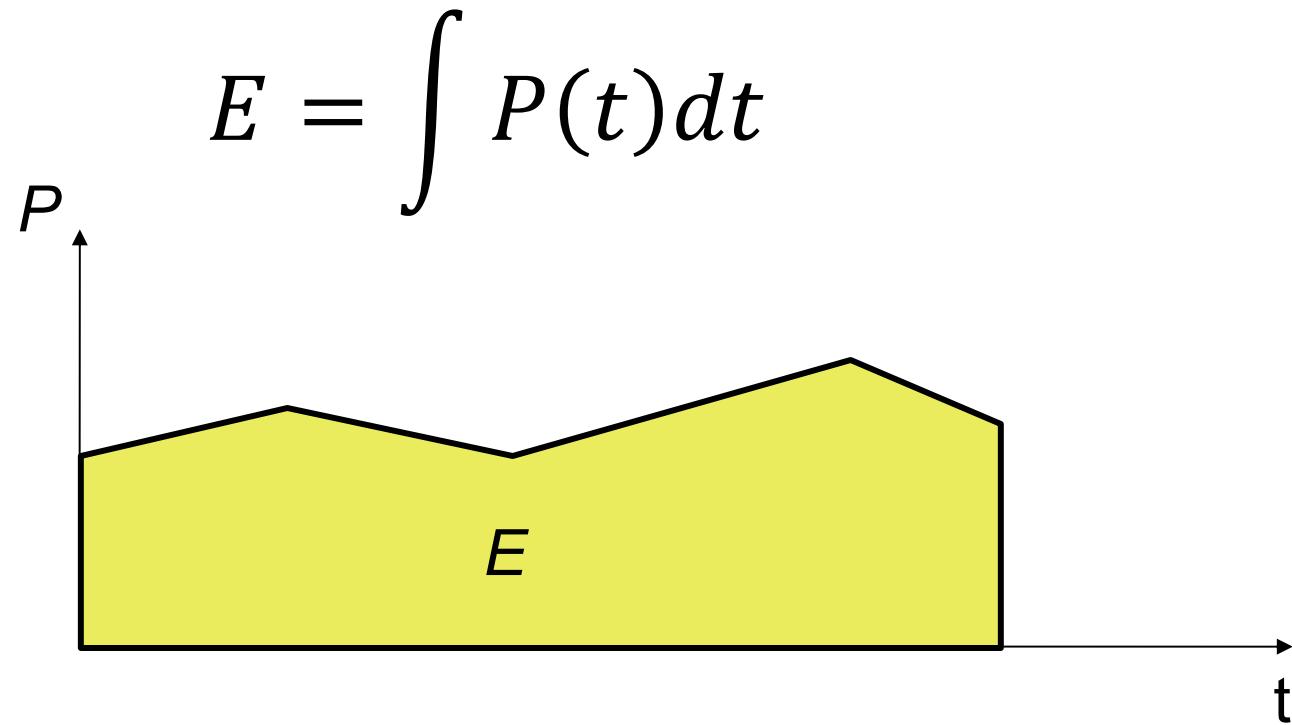
In some cases, faster execution also means less energy, but the opposite may be true if power has to be increased to allow for a faster execution.

Power and Energy



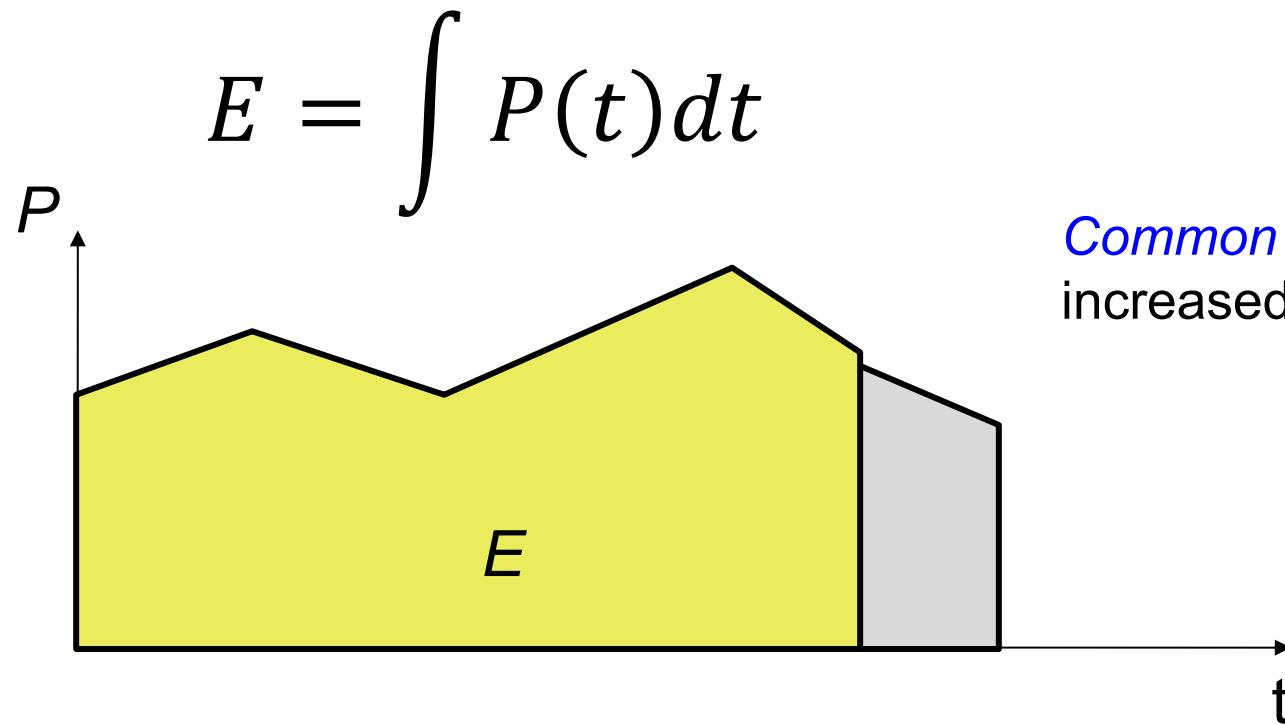
In some cases, faster execution also means less energy, but the opposite may be true if power has to be increased to allow for a faster execution.

Power and Energy



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



Common case: Power must be increased to execute faster.

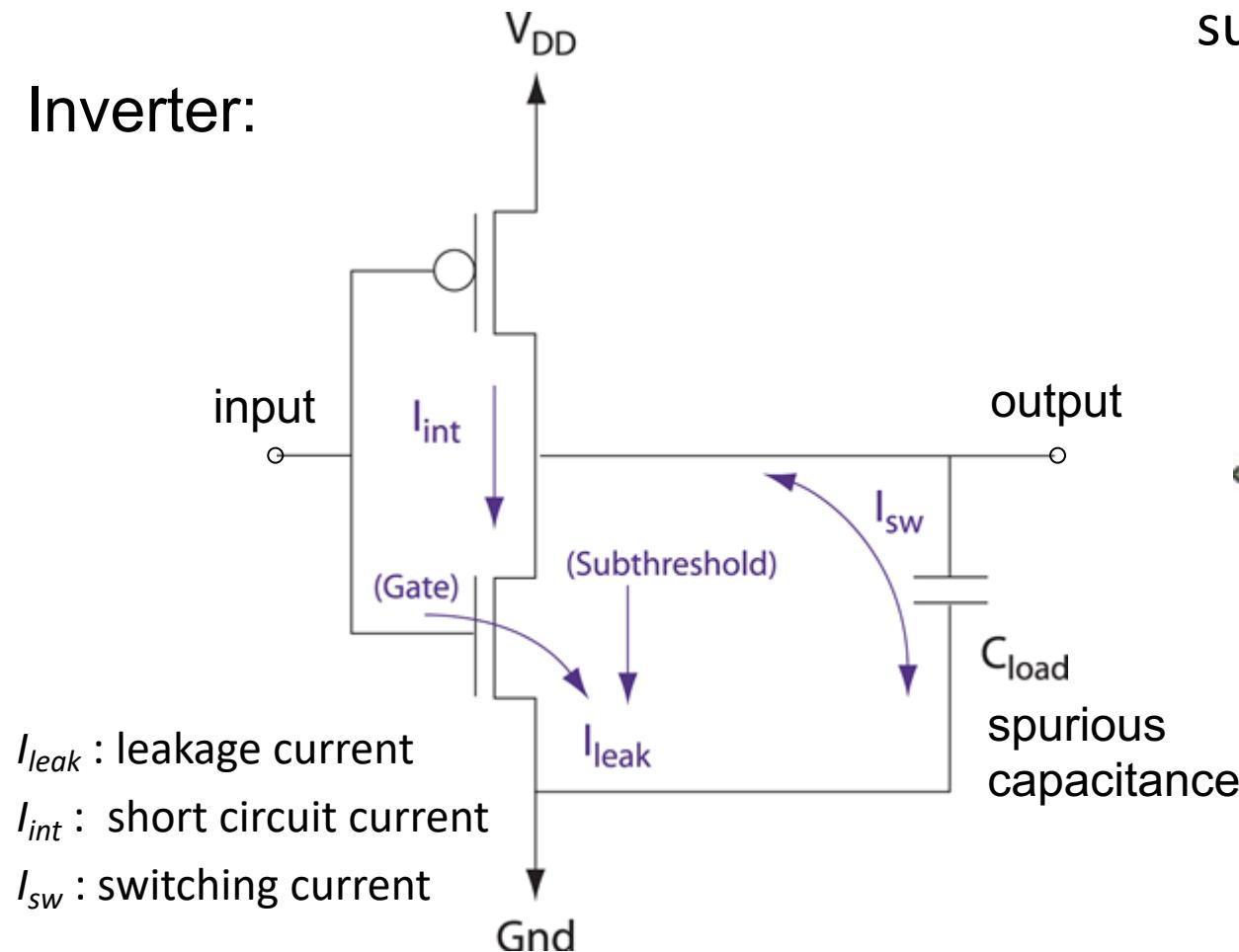
In some cases, faster execution also means less energy, but the opposite may be true if power has to be increased to allow for a faster execution.

Low Power vs. Low Energy

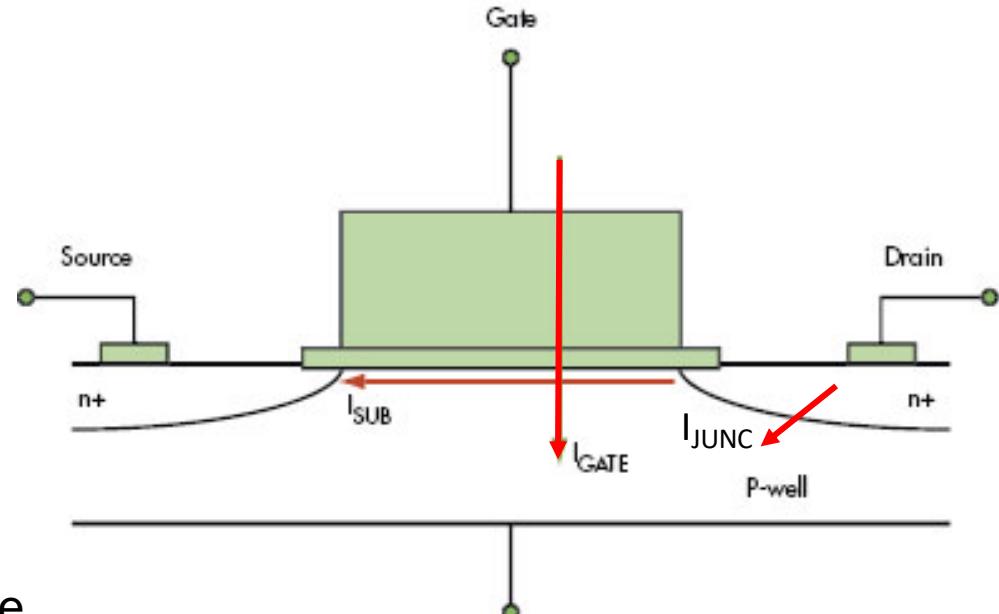
- Power and energy impact system design in different ways
- Minimizing the *power consumption* ($= \text{voltage} \times \text{current}$) is important for
 - the design of the power supply and voltage regulators
 - the dimensioning of interconnect between power supply and components (e.g., reduced power enables the use of thinner wires)
 - cooling (short term cooling)
 - high cost, limited space
- Minimizing the *energy consumption* is important due to
 - restricted availability of energy (e.g., in mobile systems or IoT devices)
 - 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

Inverter:

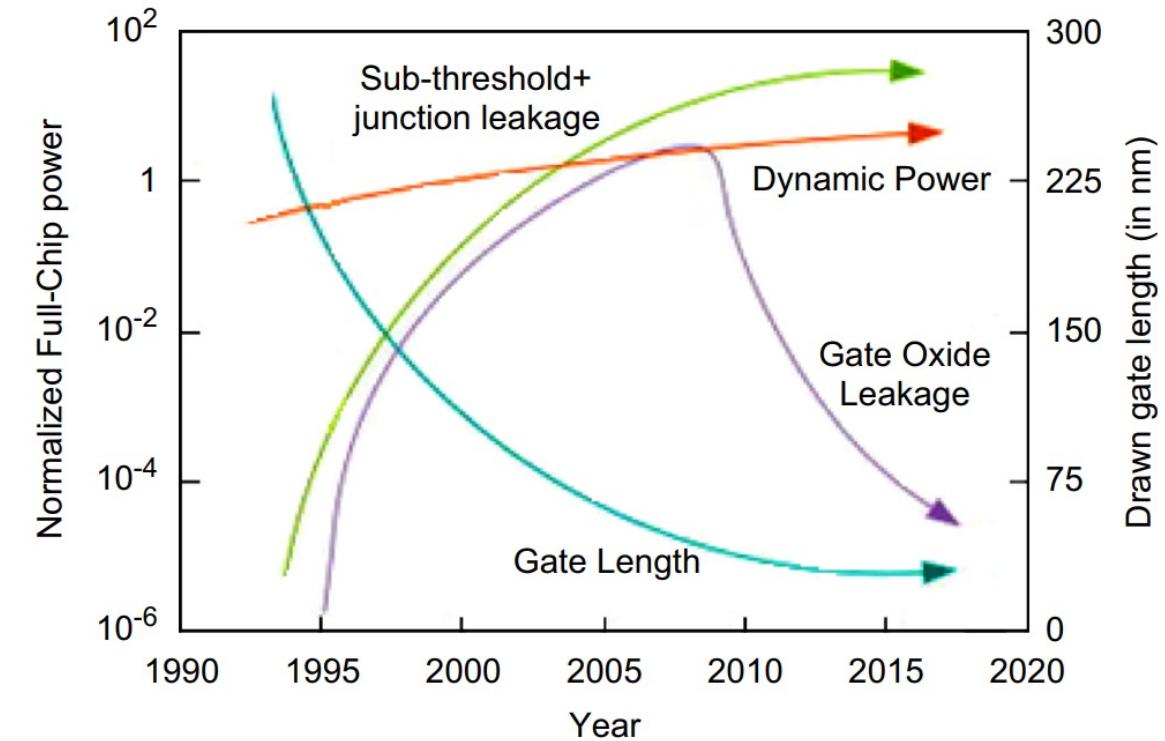


subthreshold (I_{SUB}), junction (I_{JUNC}) and gate-oxide (I_{GATE}) leakage



Main Sources of Power Consumption of a CMOS Processors

- *Dynamic* power consumption:
 - (Dis)charging capacitances while switching
 - Short-circuit power consumption due to a short-circuit path between supply rails while switching
- *Static* power consumption:
 - Gate-oxide, subthreshold, and junction leakage while nothing happens (i.e., no clock, no switching of gate inputs)



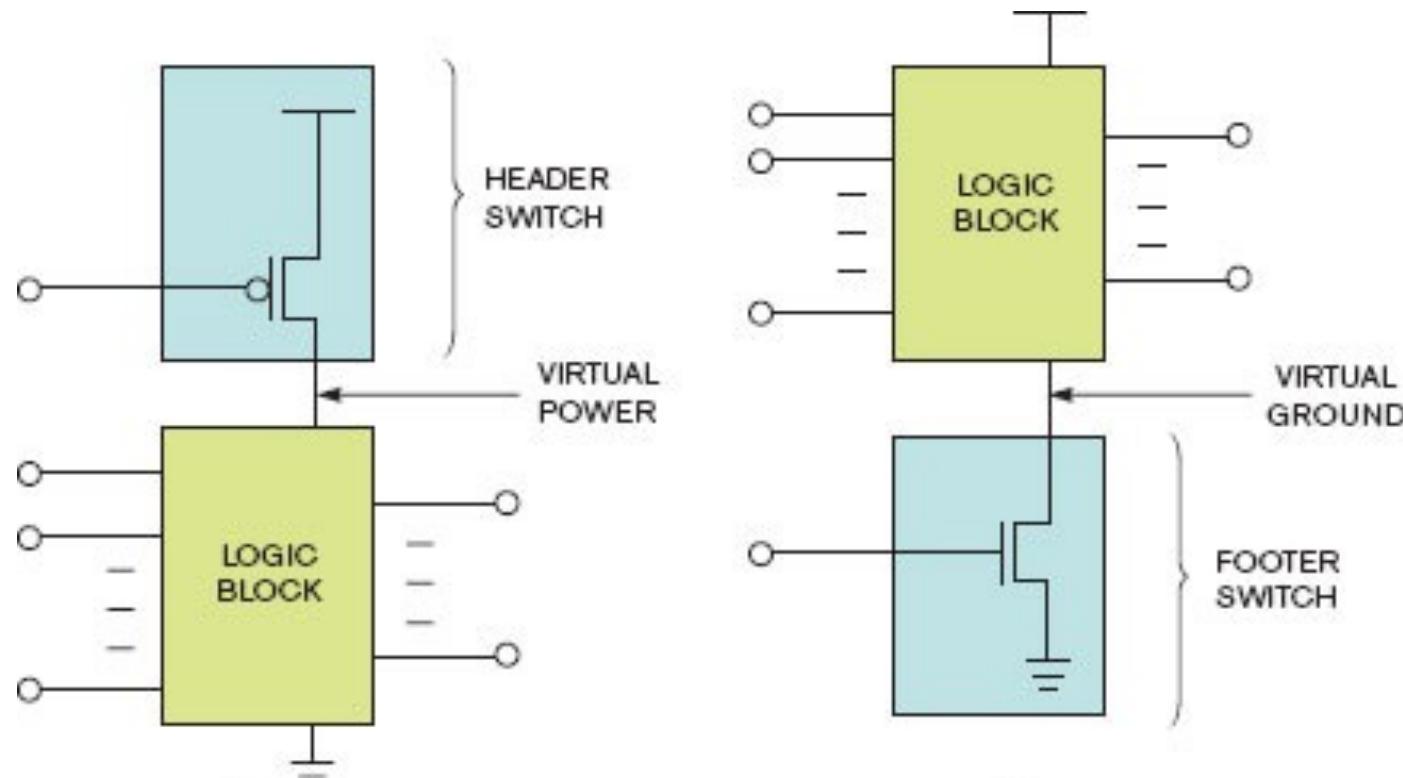
[J. Xue, T. Li, Y. Deng, Z. Yu, Full-chip leakage analysis for 65 nm CMOS technology and beyond, Integration VLSI J. 43 (4) (2010) 353–364]

Techniques to Reduce Static Power

Power Supply Gating

Power gating is one of the most effective ways to reduce static power consumption (leakage)

- Idea: Cut power supply off when components are unused (duty cycling, low-power modes)



Techniques to Reduce Dynamic Power

Voltage Scaling: A Simple Analytical Model

Average power consumption of CMOS circuits (ignoring leakage):

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

V_{dd} : supply voltage

α : switching activity

C_L : load capacity

f : clock frequency

Delay of CMOS circuits:

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

V_{dd} : supply voltage

V_T : threshold voltage

$$V_T \ll V_{dd}$$

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

But decreasing V_{dd} also increases the gate delay τ reciprocally.

Maximal frequency f_{\max} decreases linearly with decreasing V_{dd} (i.e., “reducing the voltage makes the system slower”).

$$f_{\max} \sim \frac{1}{\tau} \sim \frac{V_{dd}}{C_L}$$

Voltage Scaling: A Simple Analytical Model

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

This is the energy per cycle.

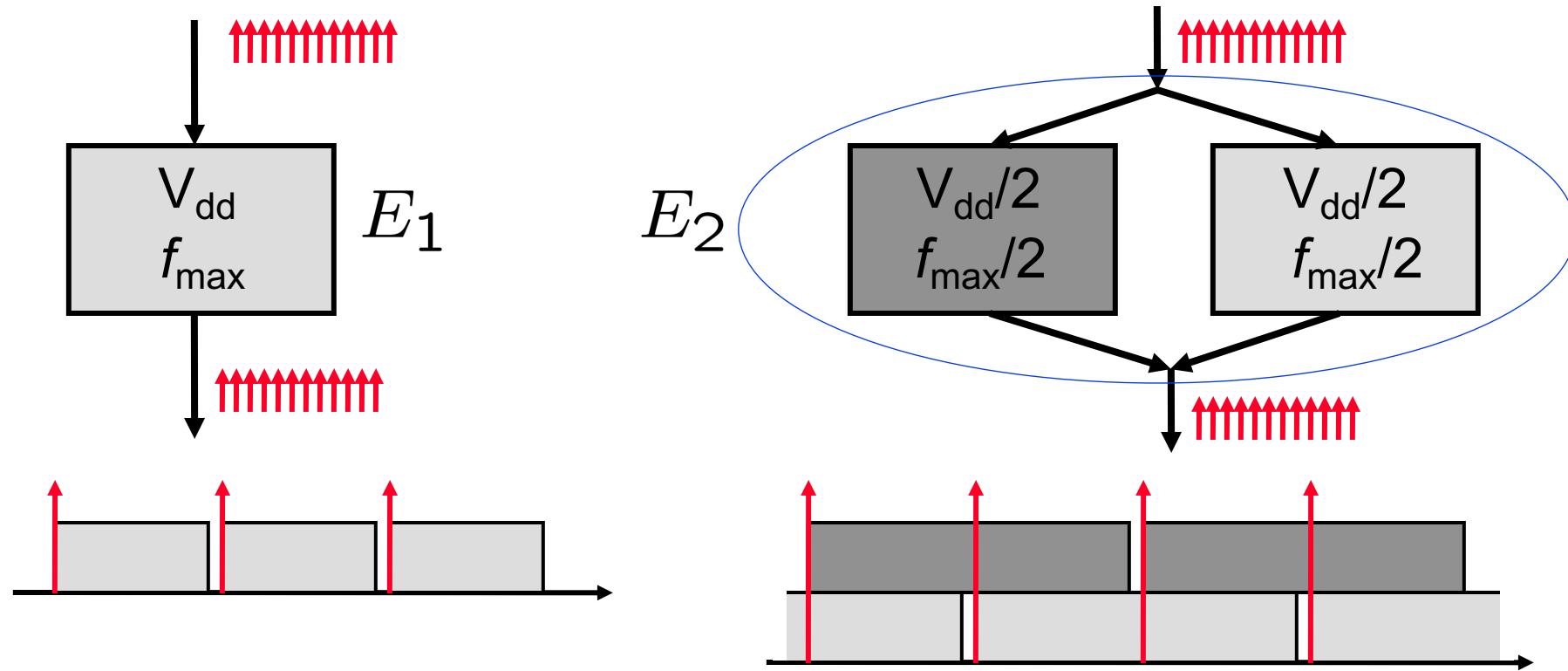
$$E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 (\#cycles)$$

This is the energy needed for executing a task that requires #cycles (i.e., number of cycles).

Saving energy for a given task:

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

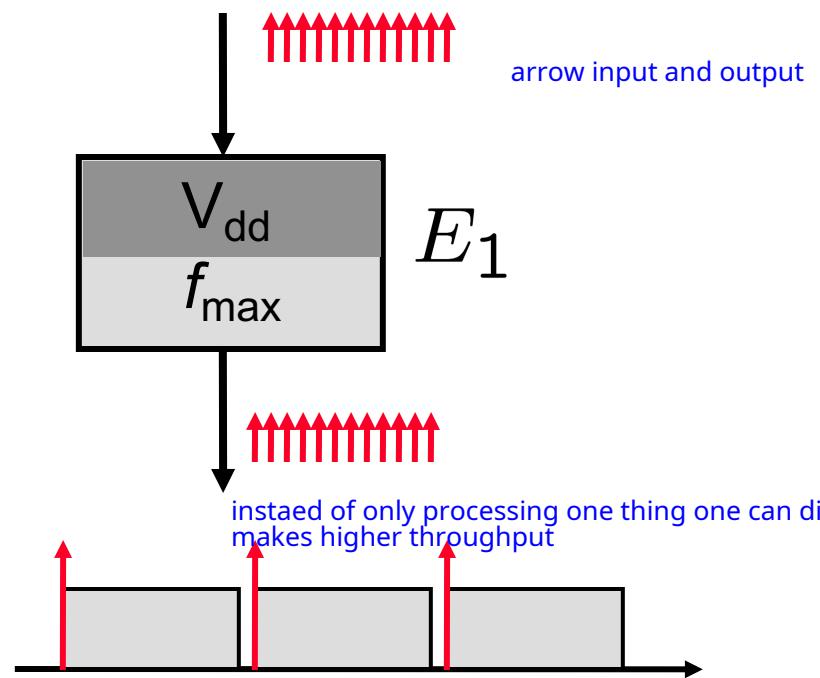
Parallelism



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

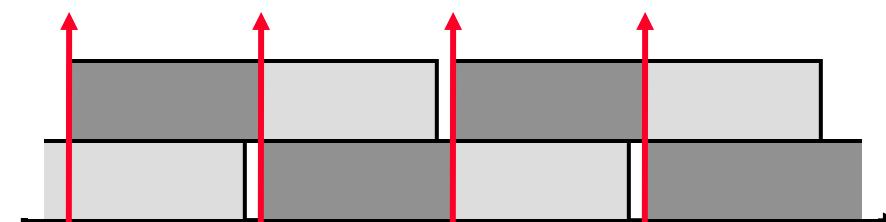
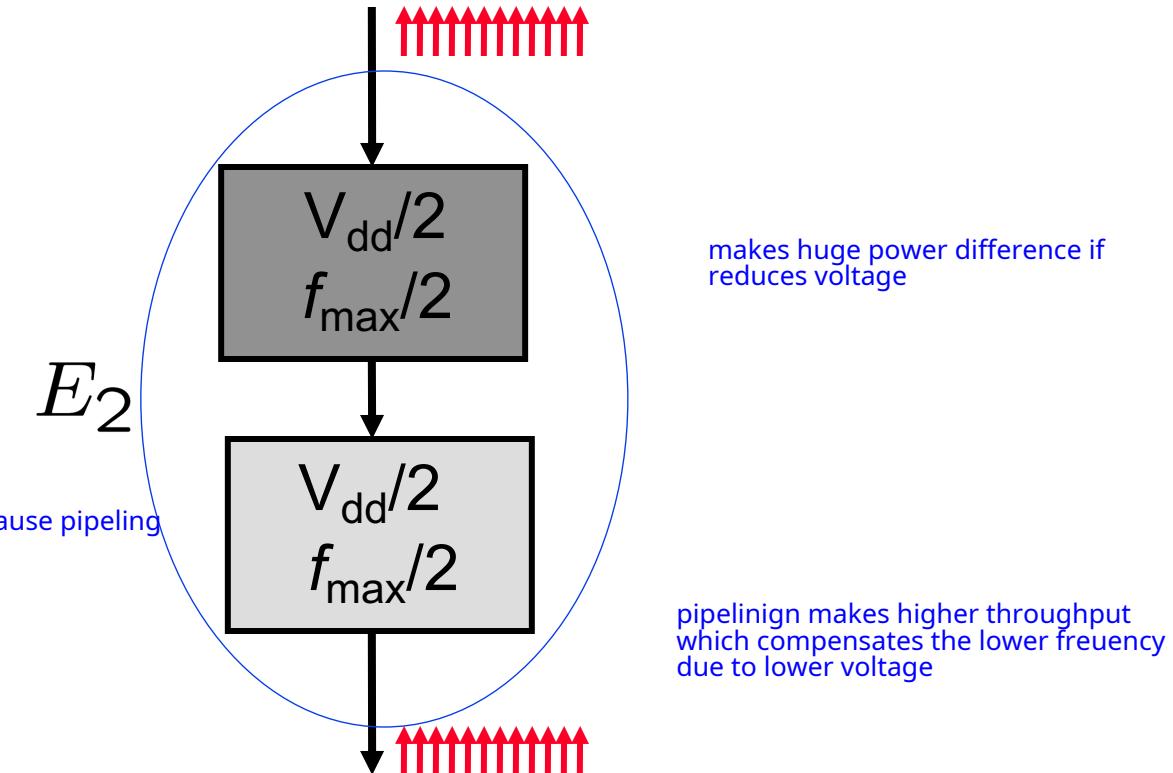
$$E_2 = \frac{1}{4} E_1$$

Pipelining



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

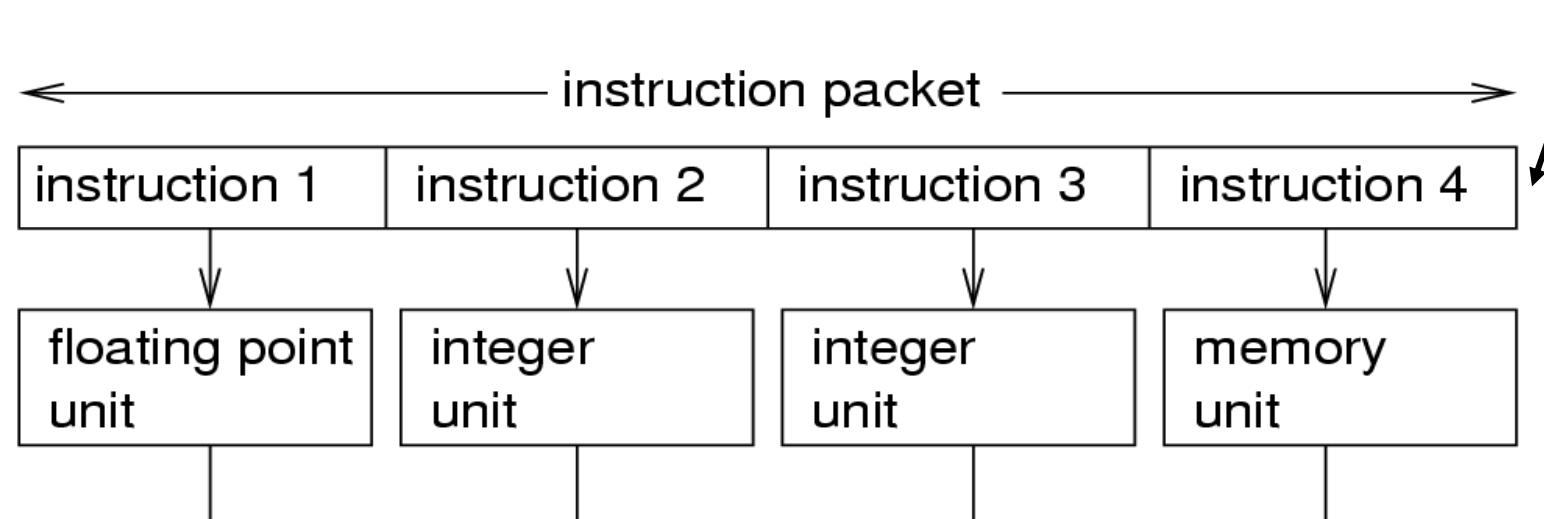
$$E_2 = \frac{1}{4}E_1$$



VLIW (Very Long Instruction Word) Architectures

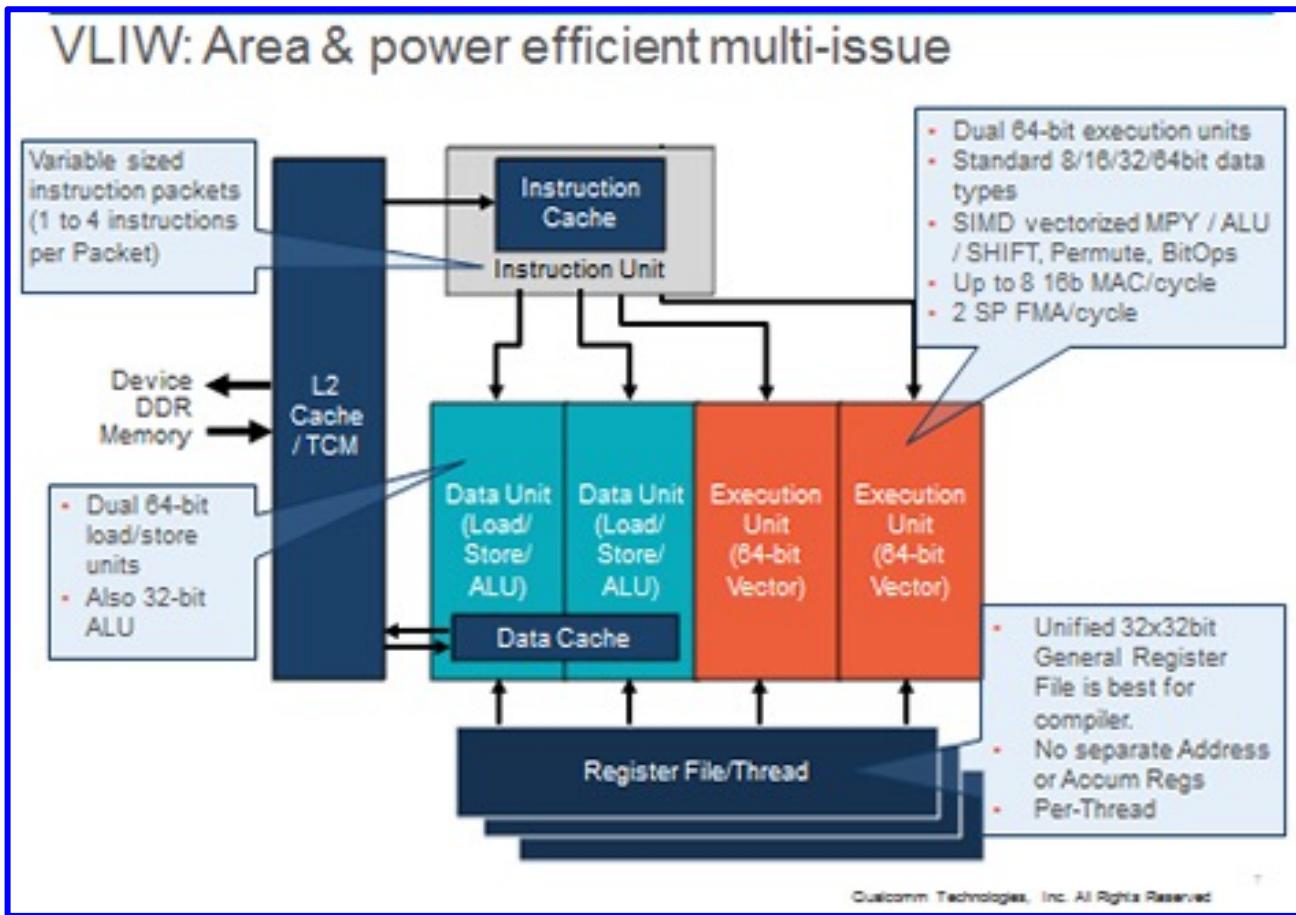
- Large degree of parallelism
 - many parallel computational units, (deeply) pipelined
- Simple hardware architecture
 - explicit parallelism (parallel instruction set)
 - parallelization is done offline (compiler)

all 4 instructions are executed in parallel

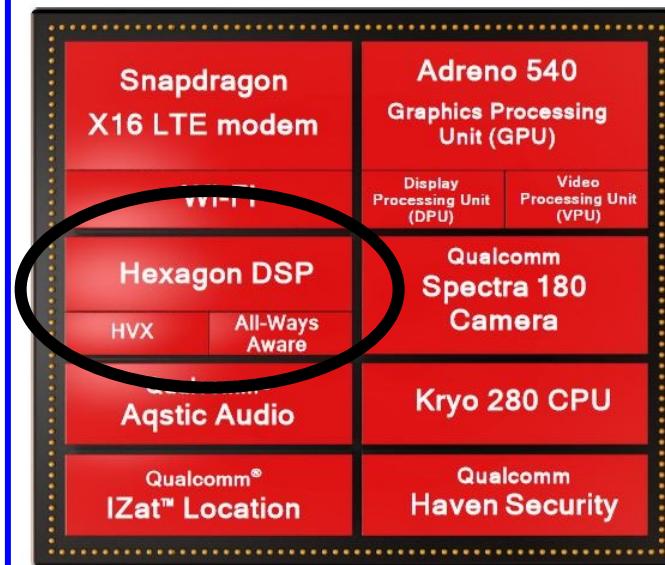


Example: Qualcomm Hexagon

Hexagon DSP



Snapdragon 835 (Galaxy S8)



So far: *Statically* decrease voltage and frequency

Next: *Dynamically* change voltage and frequency

Dynamic Voltage and Frequency Scaling (DVFS)

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

energy per cycle

$$E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 (\#cycles)$$
$$f \sim \frac{1}{\tau} \sim V_{dd}$$

gate delay

maximum frequency of operation

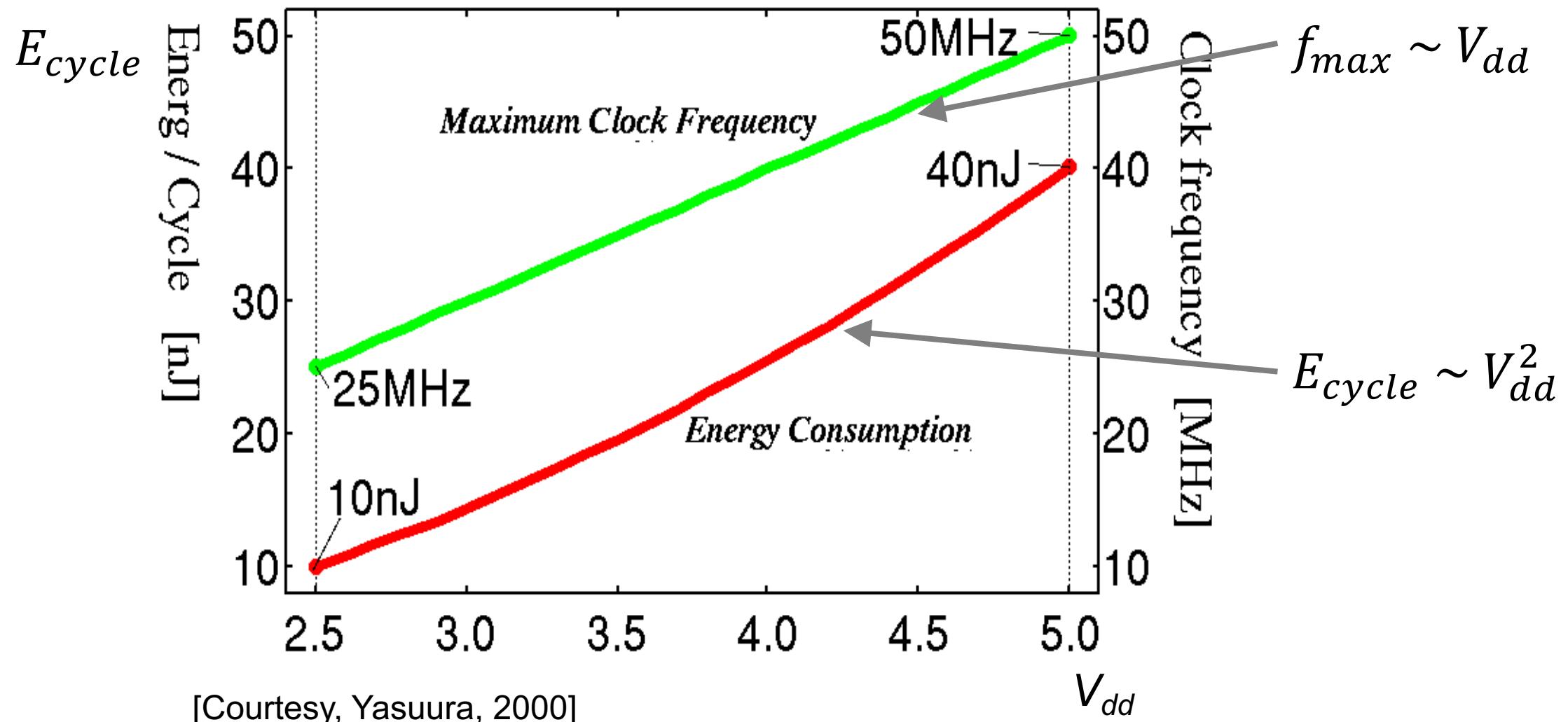
reduce voltage -> reduce energy per task

reduce voltage -> reduce clock frequency

Saving energy for a given task:

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

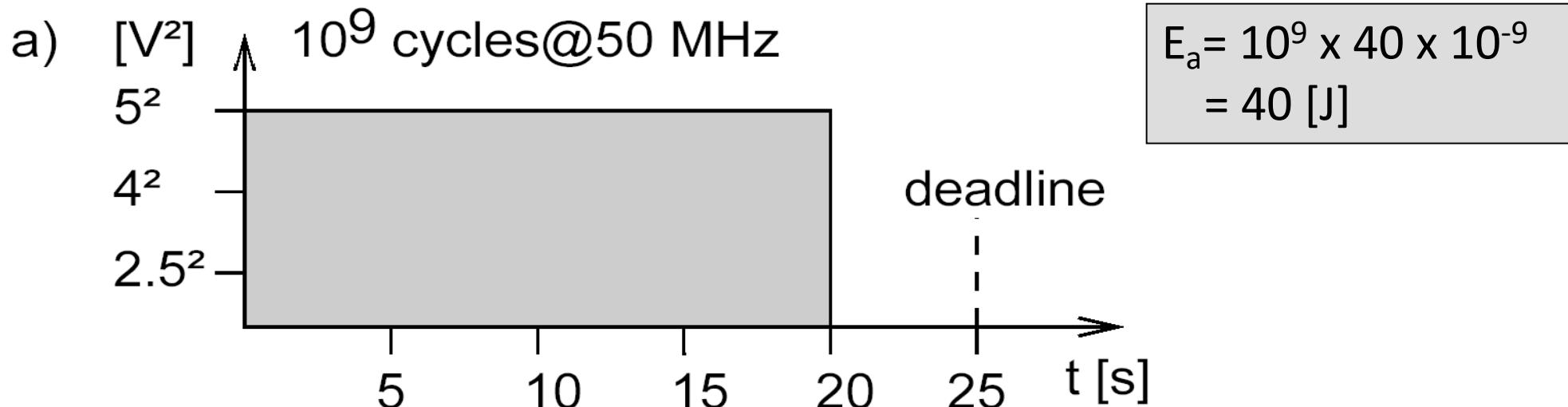
Example: Dynamic Voltage and Frequency Scaling



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

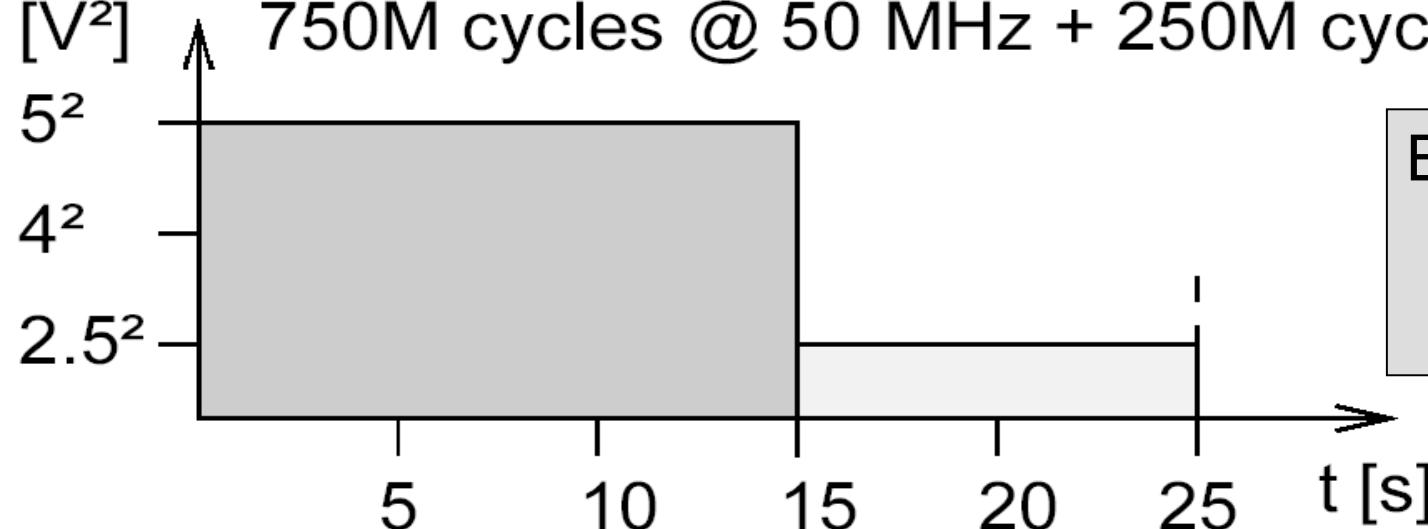
We suppose a task that needs 10^9 cycles to execute within 25 seconds.



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

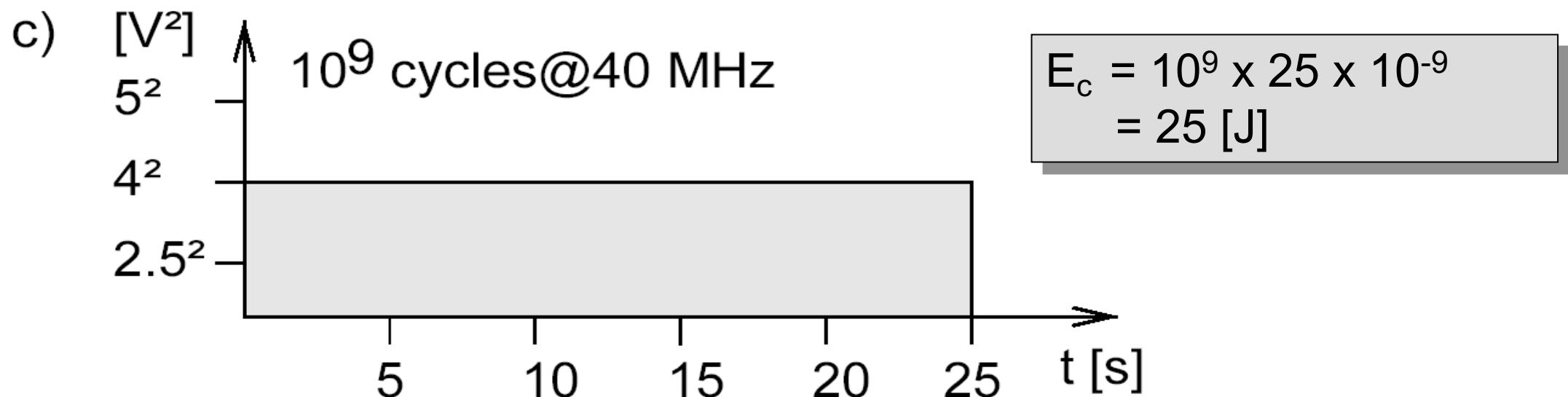
b) $[V^2]$ 750M cycles @ 50 MHz + 250M cycles @ 25 MHz



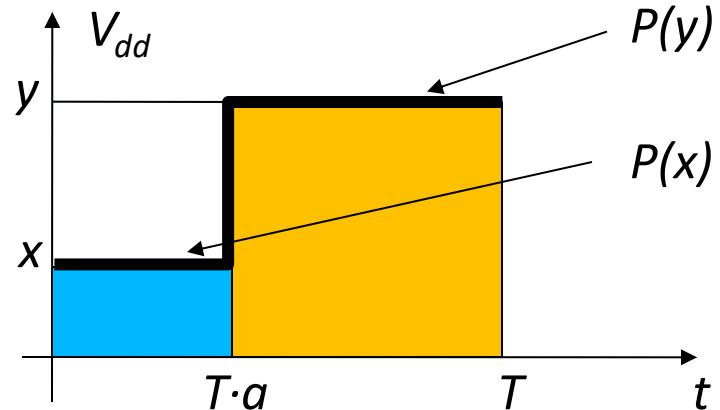
$$\begin{aligned} E_b &= 750 \cdot 10^6 \times 40 \times 10^{-9} \\ &\quad + 250 \cdot 10^6 \times 10 \times 10^{-9} \\ &= 32.5 \text{ [J]} \end{aligned}$$

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



DVFS: Optimal Strategy



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

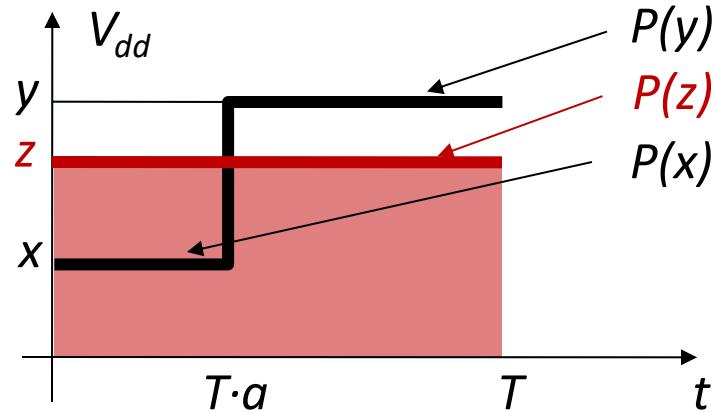
$$\text{gate delay: } \tau \sim \frac{1}{V_{dd}}$$

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

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

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

DVFS: Optimal Strategy



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

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

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

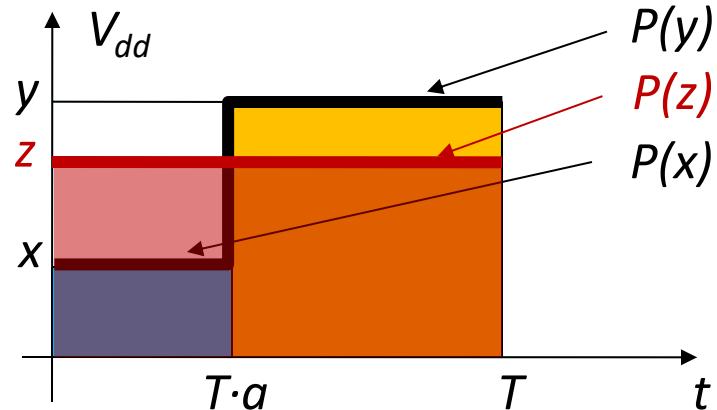
$$\text{invariant: } \int V_{dd}(t) dt = \text{const.}$$

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

Ensures that we compare the energies consumed by executing the same algorithm with the same #cycles

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

DVFS: Optimal Strategy



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

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

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

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

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

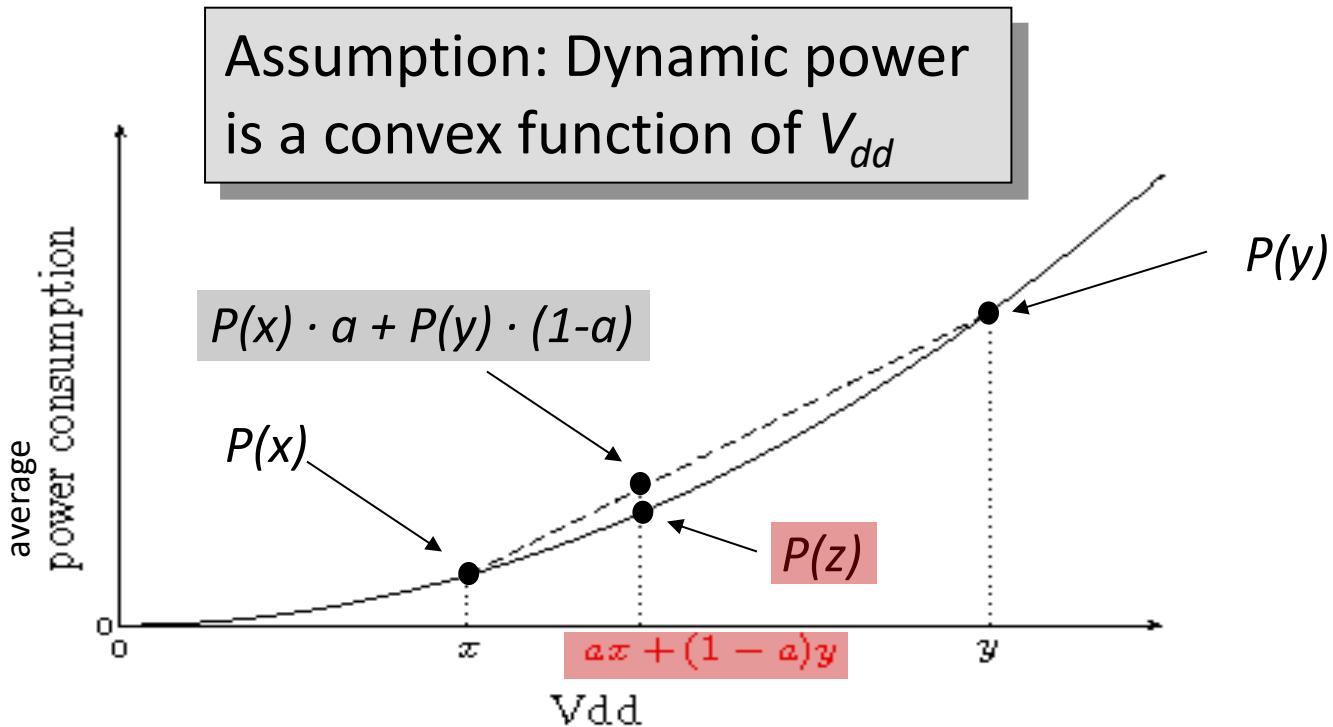
$$\text{invariant: } \int V_{dd}(t) dt = \text{const.}$$

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

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

Ensures that we compare the energies consumed by executing the same algorithm with the same #cycles

DVFS: Optimal Strategy



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

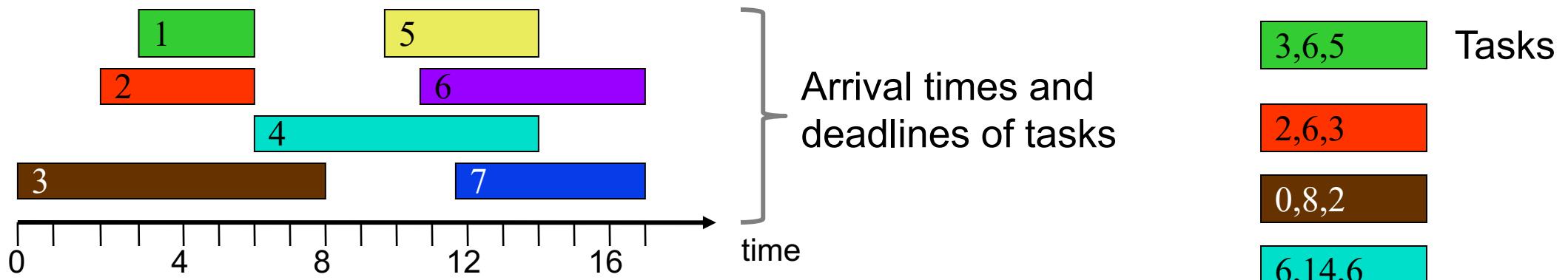
case A is always worse if the power consumption is a convex function of the supply voltage

DVFS: Real-Time Offline Scheduling on One Processor

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



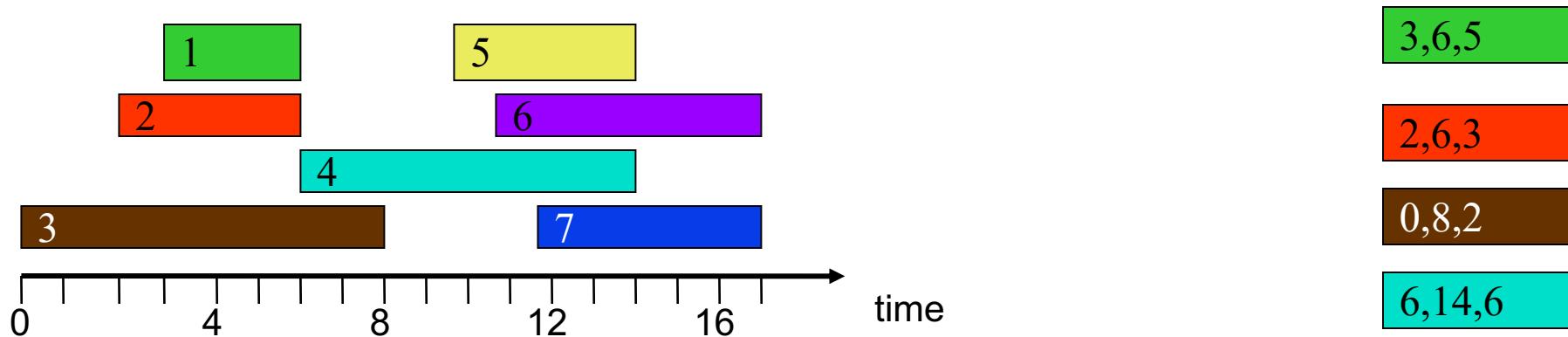
- Define **intensity** $G([z, z'])$ in some time interval $[z, z']$:
 - average accumulated execution time of all tasks that have arrival and deadline in $[z, z']$ relative to the length of the interval $z'-z$

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

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

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.



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

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

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

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

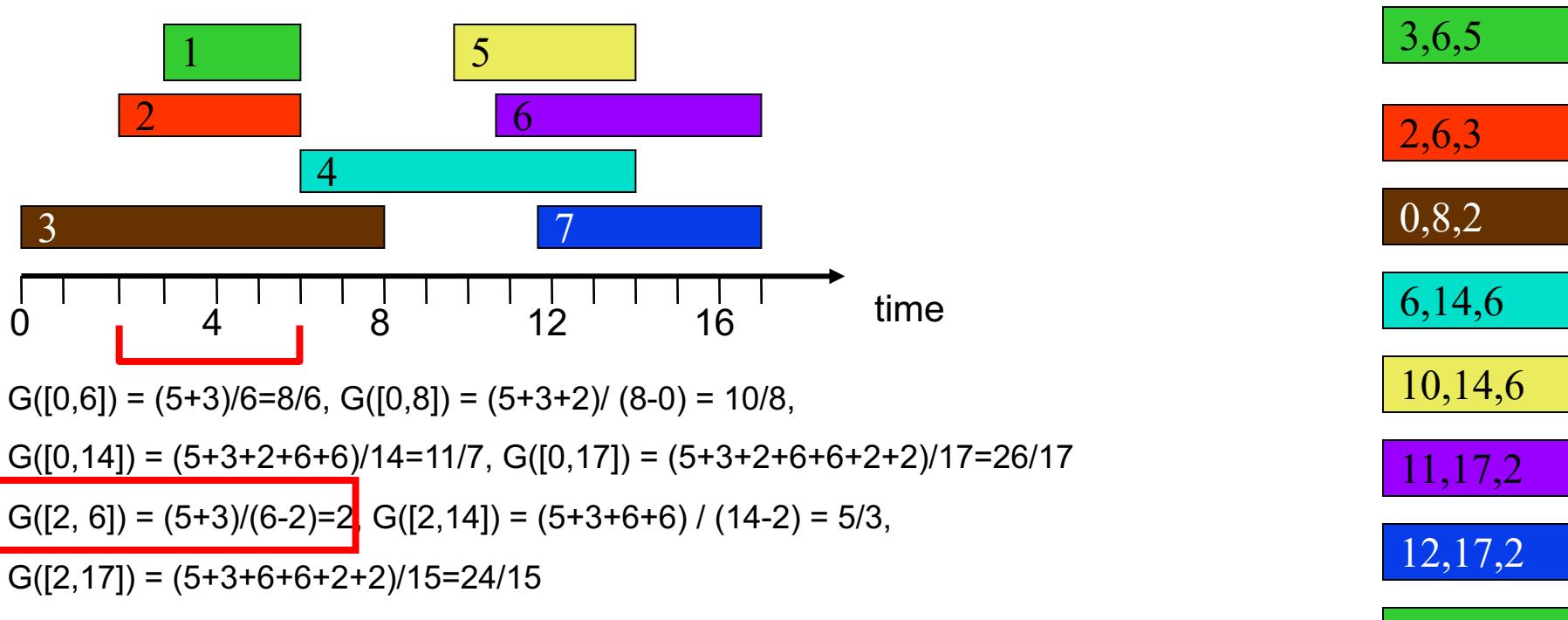
$$G([3, 6]) = 5/3, G([3, 14]) = (5+6+6)/(14-3) = 17/11, G([3, 17]) = (5+6+6+2+2)/14 = 21/14$$

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

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

YDS Optimal DVFS Algorithm for Offline Scheduling

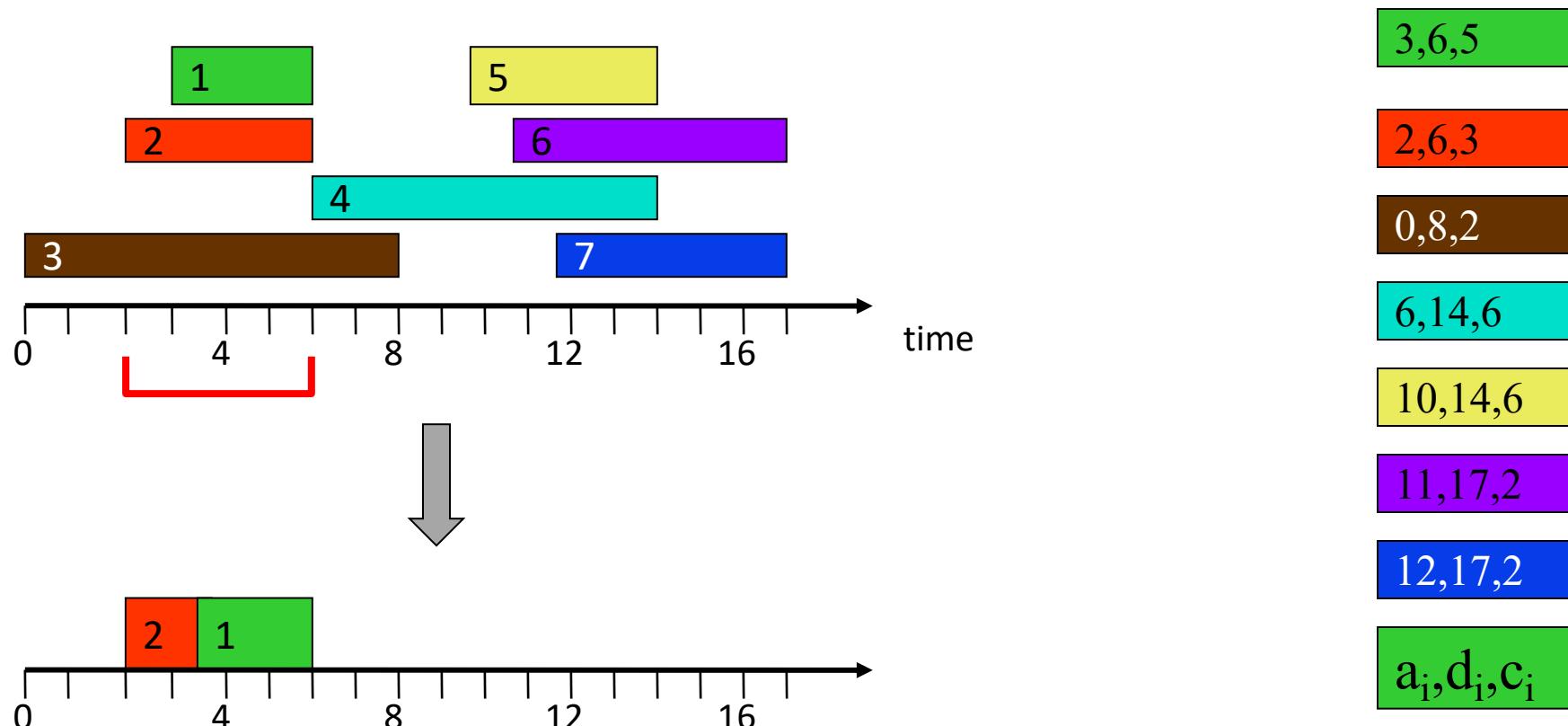
Step 1: Execute jobs in the interval with the highest intensity by using the earliest-deadline first schedule and running at the intensity as the frequency.



*critical
interval*

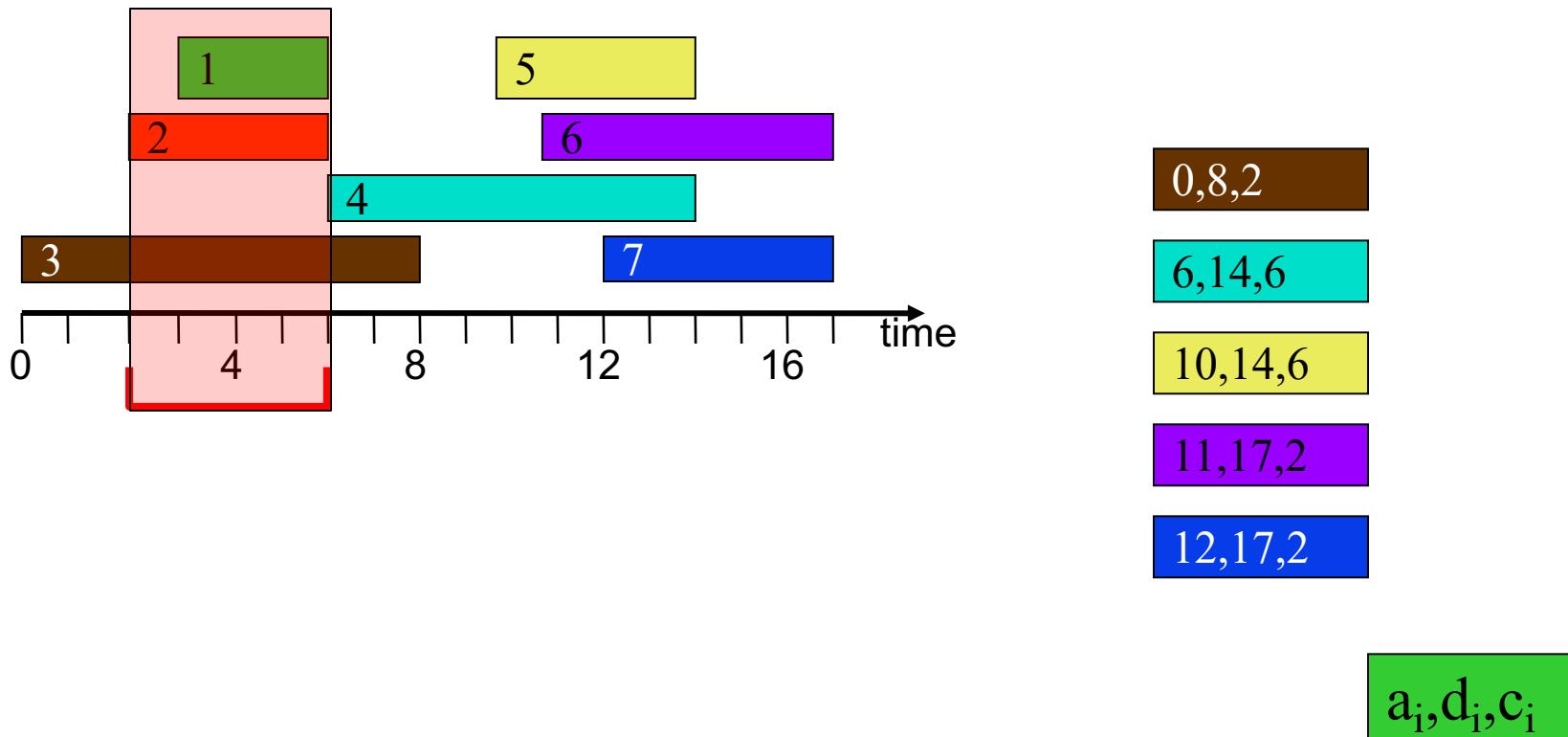
YDS Optimal DVFS Algorithm for Offline Scheduling

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.



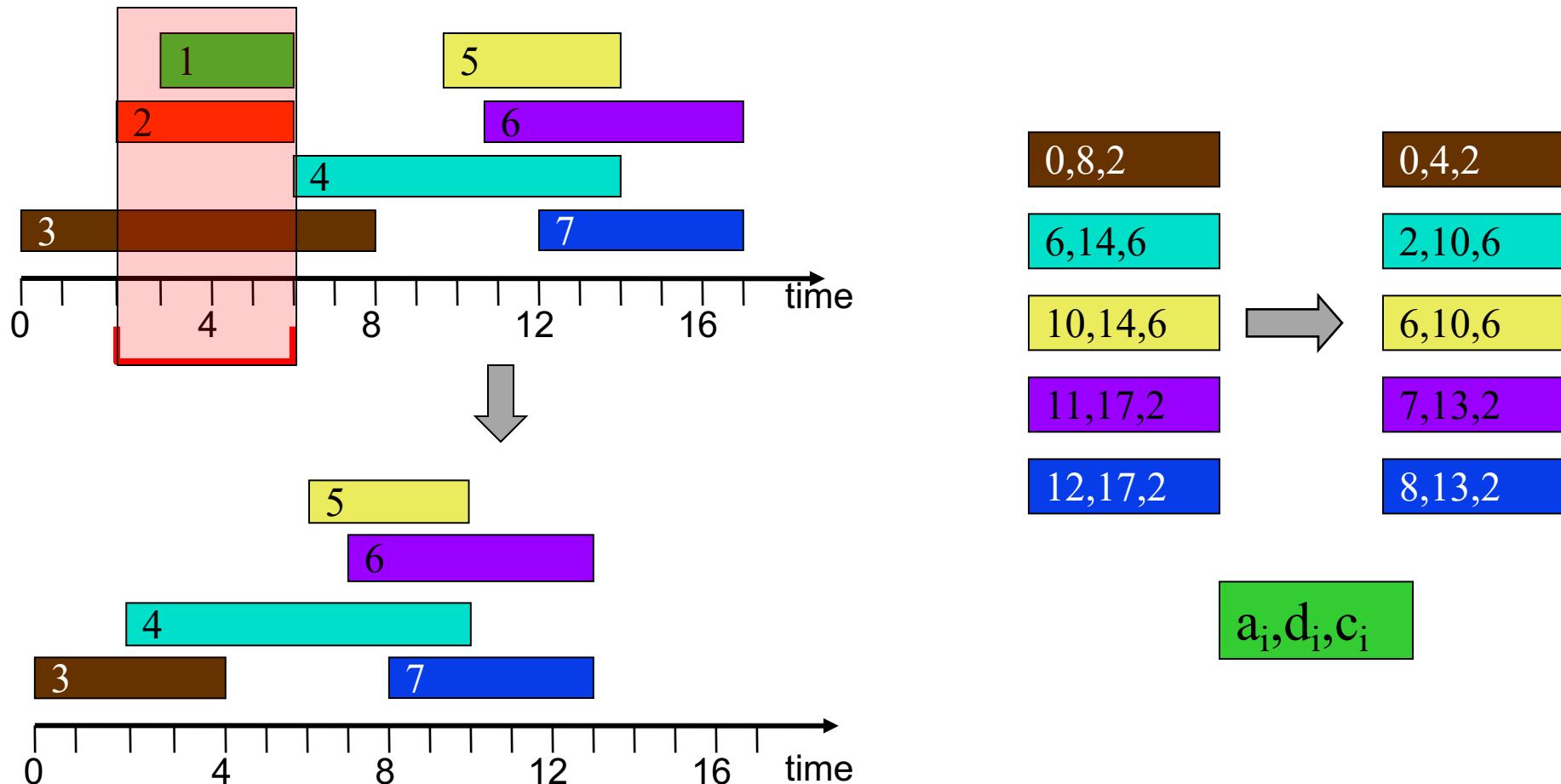
YDS Optimal DVFS Algorithm for Offline Scheduling

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



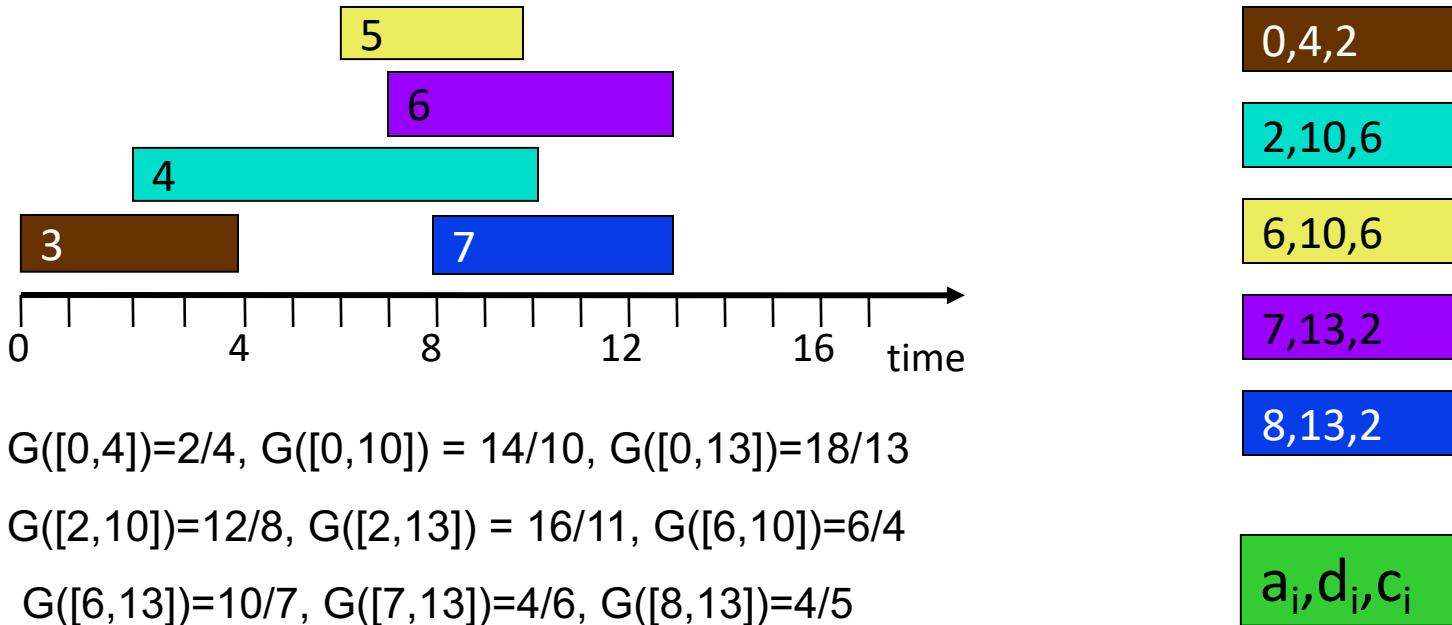
YDS Optimal DVFS Algorithm for Offline Scheduling

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



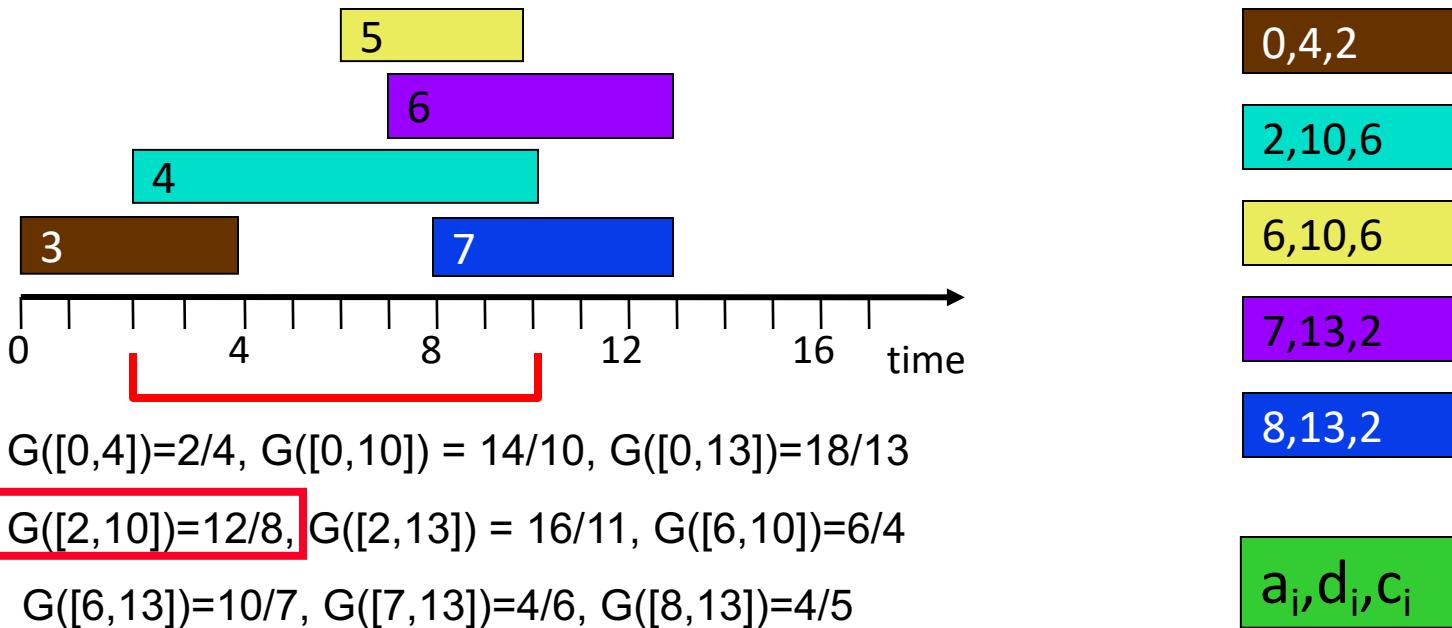
YDS Optimal DVFS Algorithm for Offline Scheduling

Step 3: Run the algorithm for the revised input again



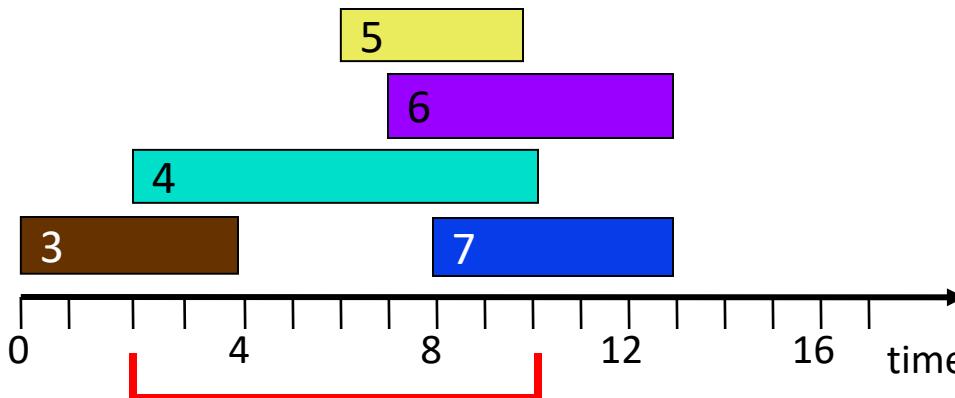
YDS Optimal DVFS Algorithm for Offline Scheduling

Step 3: Run the algorithm for the revised input again



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

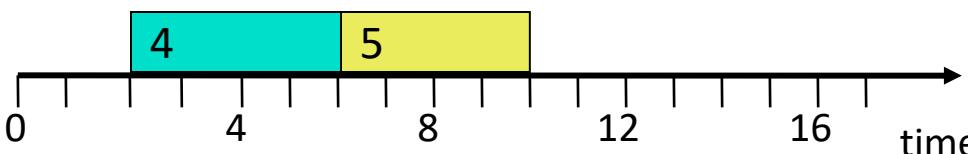
2,10,6

6,10,6

7,13,2

8,13,2

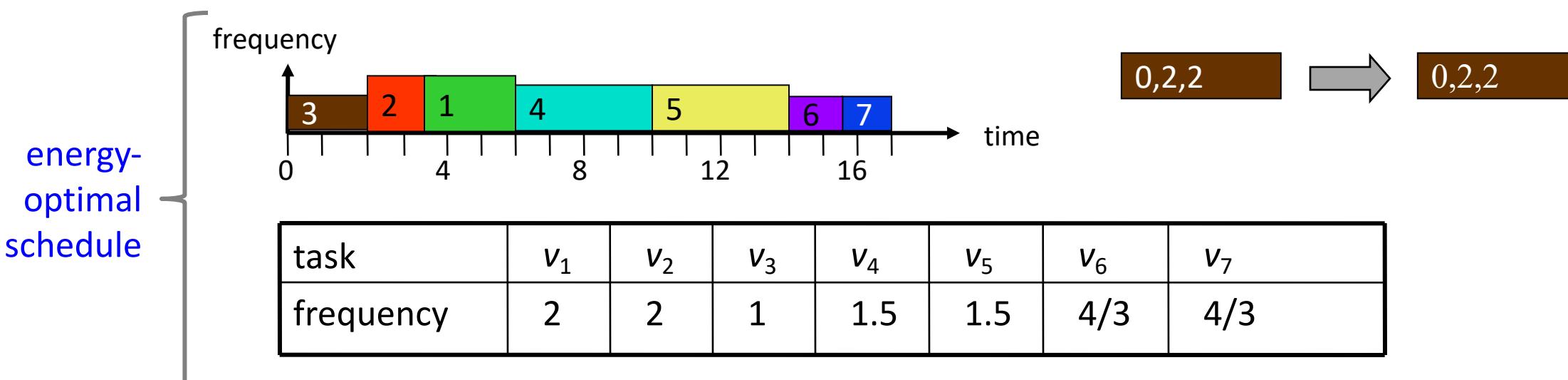
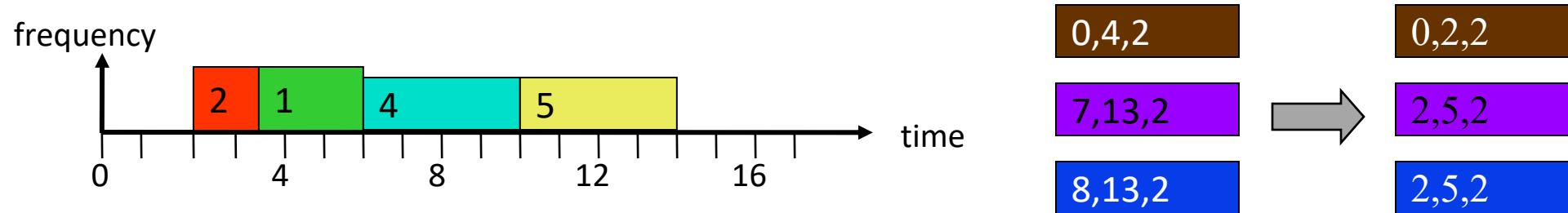
a_i, d_i, c_i



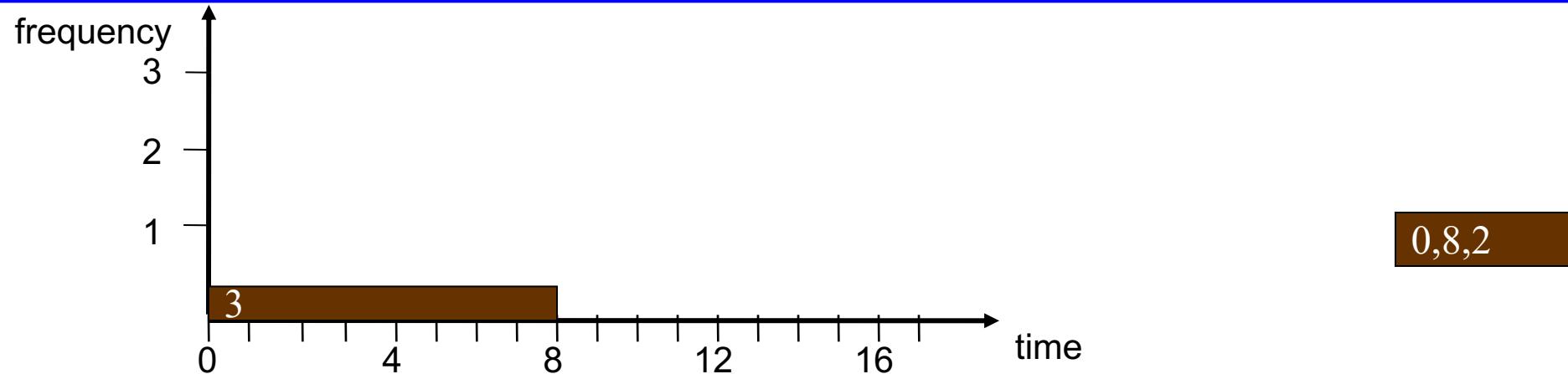
YDS Optimal DVFS Algorithm for Offline Scheduling

Step 3: Run the algorithm for the revised input again

Step 4: Put pieces together



YDS Optimal DVFS Algorithm for Online Scheduling

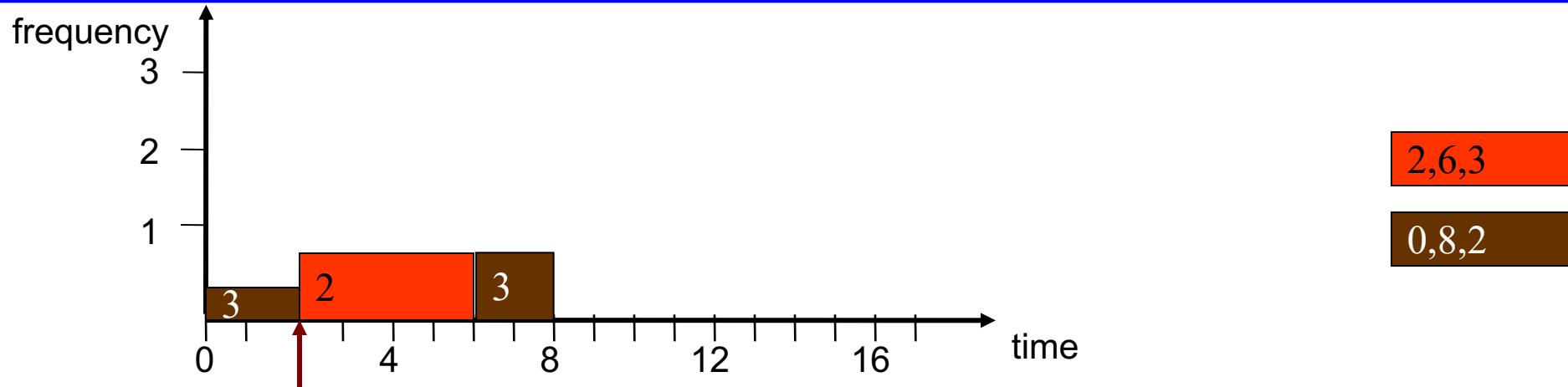


Continuously update to the best schedule for all arrived tasks:

Time 0: task v_3 is executed at 2/8

a_i, d_i, c_i

YDS Optimal DVFS Algorithm for Online Scheduling



Continuously update to the best schedule for all arrived tasks:

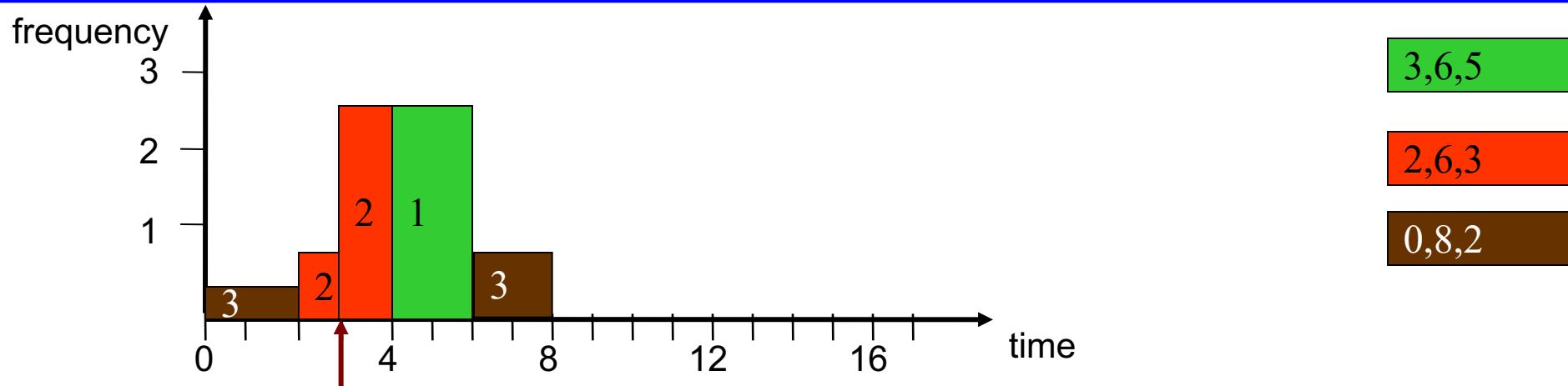
Time 0: task v_3 is executed at $2/8$

Time 2: task v_2 arrives

- $G([2,6]) = \frac{3}{4}$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v_3, v_2 at $\frac{3}{4}$

a_i, d_i, c_i

YDS Optimal DVFS Algorithm for Online Scheduling



Continuously update to the best schedule for all arrived tasks:

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

Time 2: task v_2 arrives

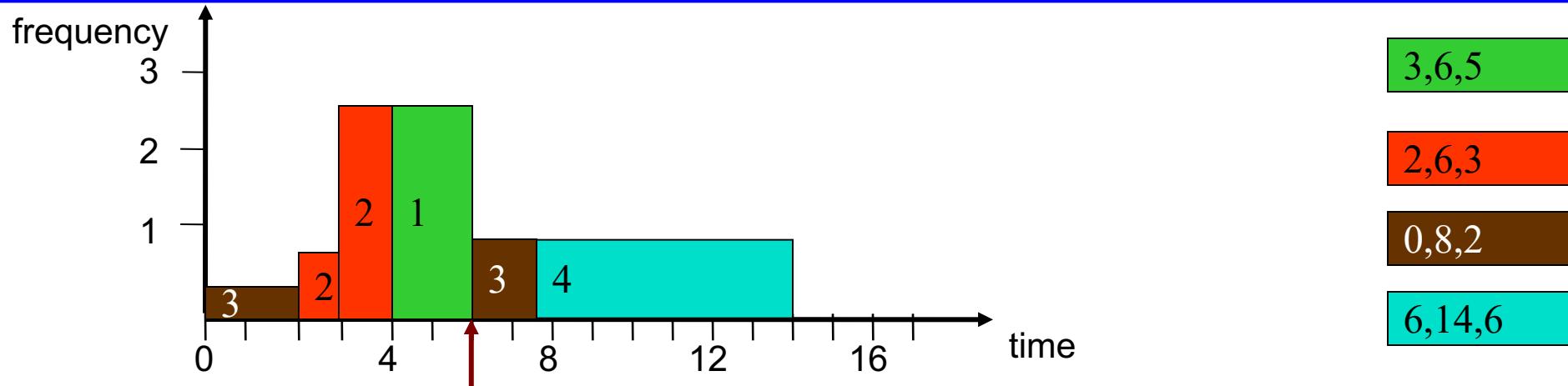
- $G([2,6]) = \frac{3}{4}$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v_3, v_2 at $\frac{3}{4}$

Time 3: task v_1 arrives

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

a_i, d_i, c_i

YDS Optimal DVFS Algorithm for Online Scheduling



Continuously update to the best schedule for all arrived tasks:

Time 0: task v₃ is executed at 2/8

Time 2: task v₂ arrives

- $G([2,6]) = \frac{3}{4}$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v₃, v₂ at $\frac{3}{4}$

Time 3: task v₁ arrives

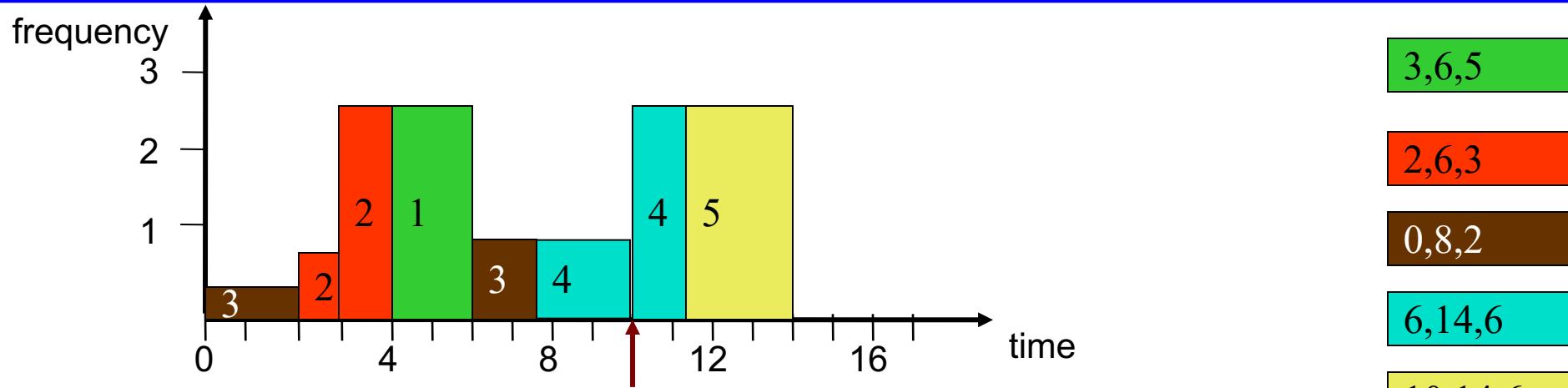
- $G([3,6]) = (5+3-3/4)/3 = 29/12$, $G([3,8]) < G([3,6]) \Rightarrow$ execute v₂ and v₁ at 29/12

Time 6: task v₄ arrives

- $G([6,8]) = 1.5/2$, $G([6,14]) = 7.5/8 \Rightarrow$ execute v₃ and v₄ at 15/16

a_i, d_i, c_i

YDS Optimal DVFS Algorithm for Online Scheduling



Continuously update to the best schedule for all arrived tasks:

Time 0: task v_3 is executed at 2/8

Time 2: task v_2 arrives

- $G([2,6]) = \frac{3}{4}$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v_8 , v_2 at $\frac{3}{4}$

Time 3: task v_1 arrives

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

a_i, d_i, c_i

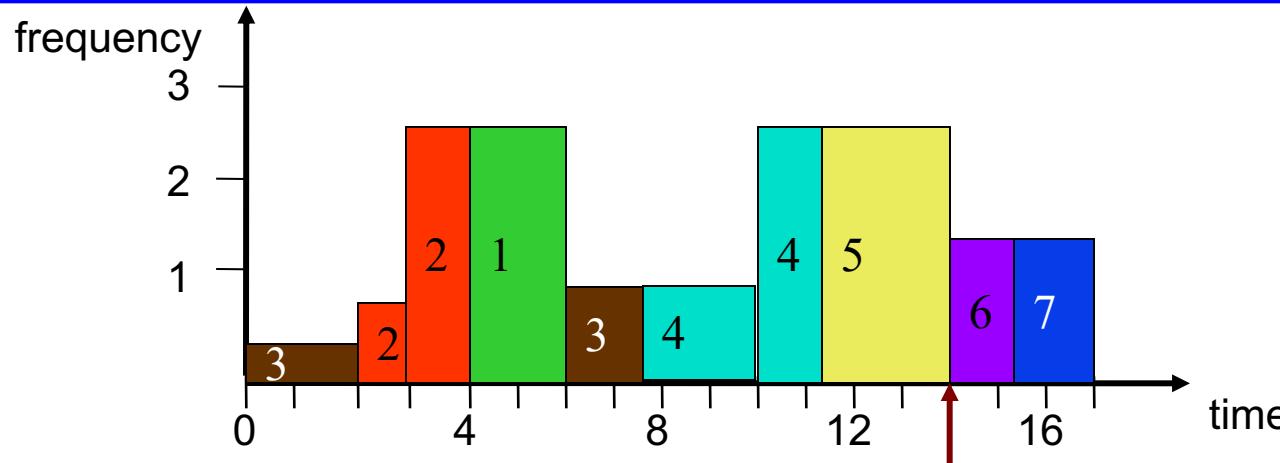
Time 6: task v_4 arrives

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

Time 10: task v_5 arrives

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

YDS Optimal DVFS Algorithm for Online Scheduling



Continuously update to the best schedule for all arrived tasks:

Time 0: task v₃ is executed at 2/8

Time 2: task v₂ arrives

- $G([2,6]) = \frac{3}{4}$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v₈, v₂ at $\frac{3}{4}$

Time 3: task v₁ arrives

- $G([3,6]) = (5+3-3/4)/3 = 29/12$, $G([3,8]) < G([3,6]) \Rightarrow$ execute v₂ and v₁ at 29/12

Time 6: task v₄ arrives

- $G([6,8]) = 1.5/2$, $G([6,14]) = 7.5/8 \Rightarrow$ execute v₃ and v₄ at 15/16

Time 10: task v₅ arrives

- $G([10,14]) = 39/16 \Rightarrow$ execute v₄ and v₅ at 39/16

Time 11 and Time 12

- The arrival of v₆ and v₇ does not change the critical interval

Time 14:

- $G([14,17]) = 4/3 \Rightarrow$ execute v₆ and v₇ at 4/3

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

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.

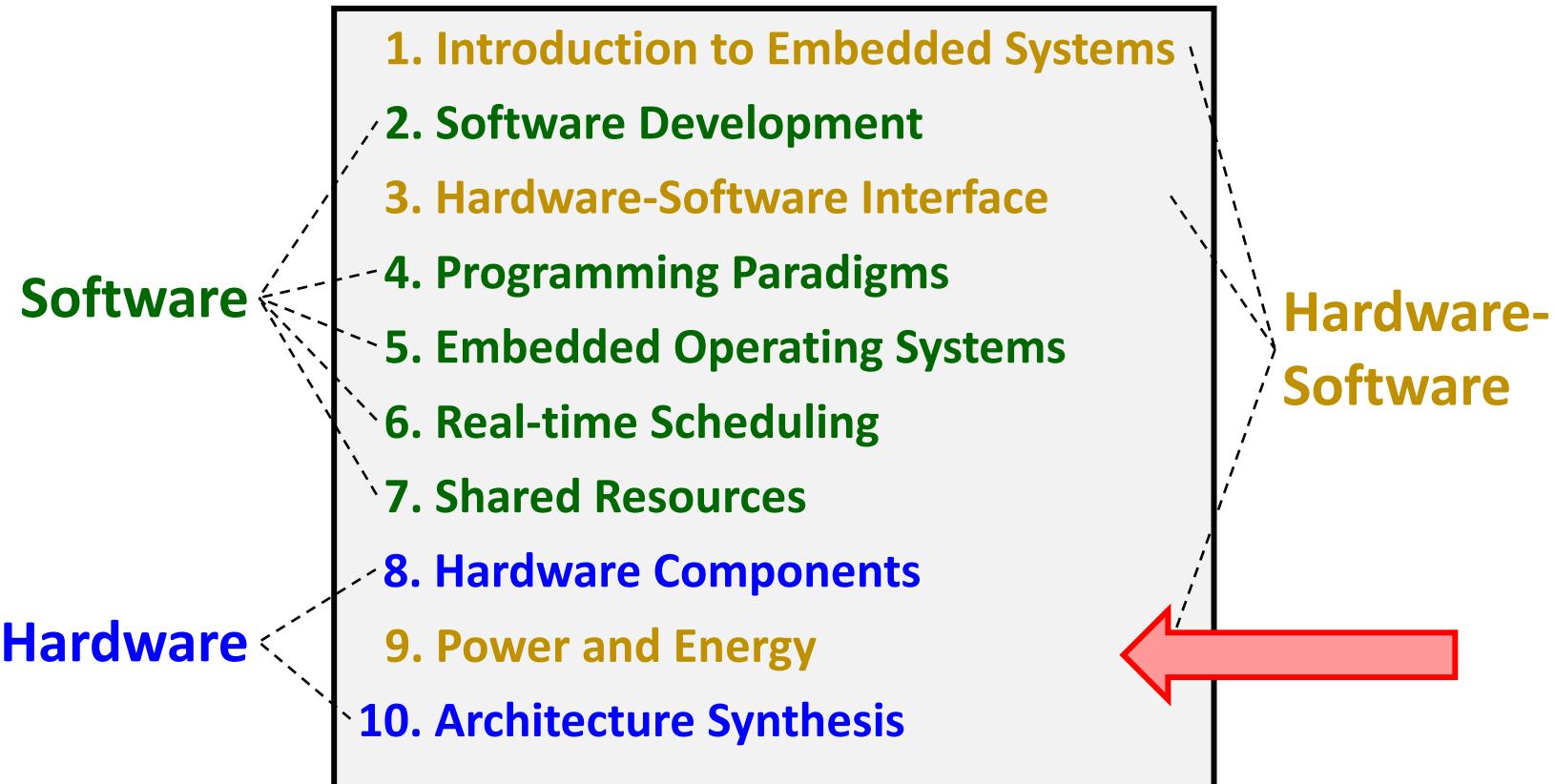
Introduction to Embedded Systems

9. Power and Energy

Prof. Dr. Marco Zimmerling

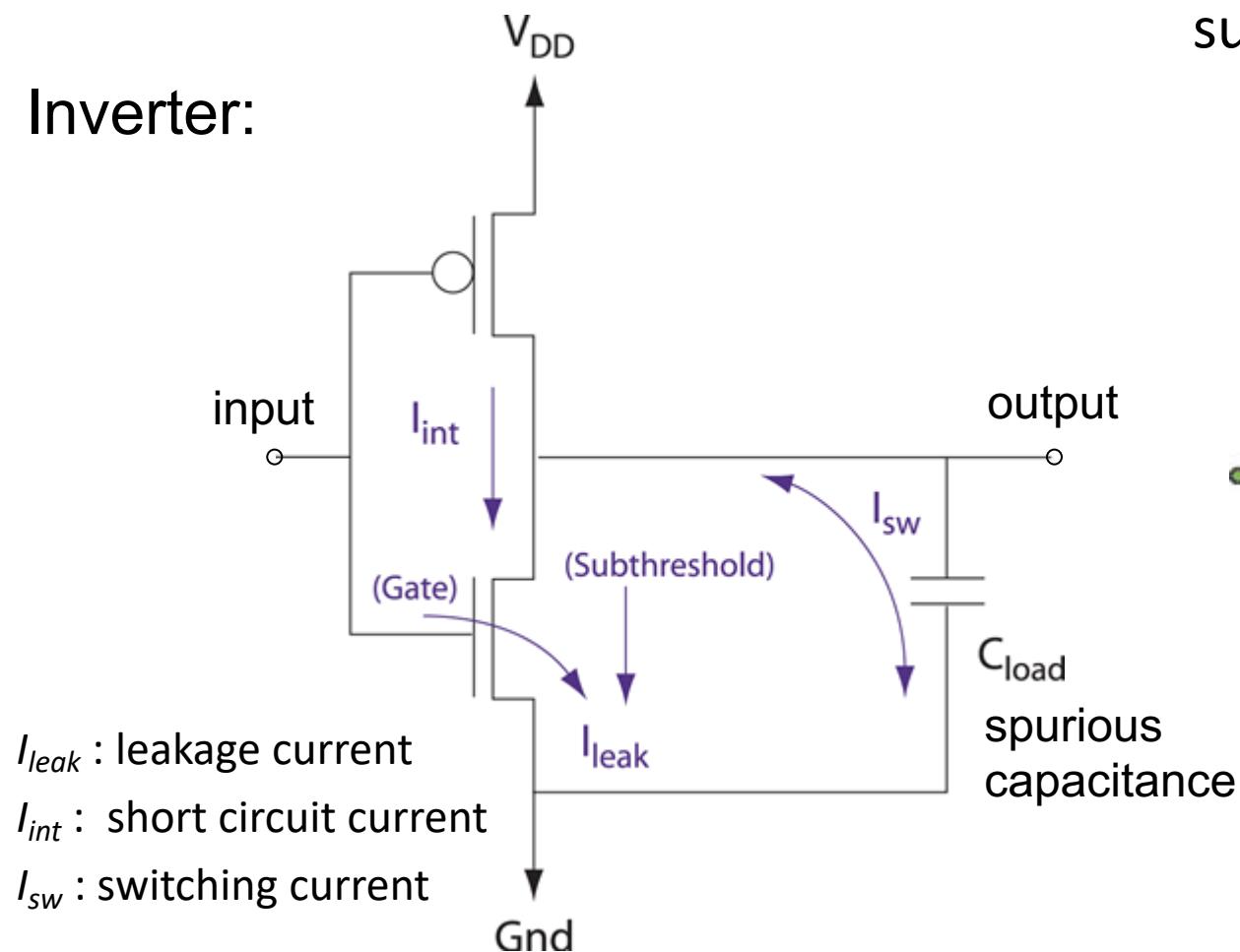


Where we are ...

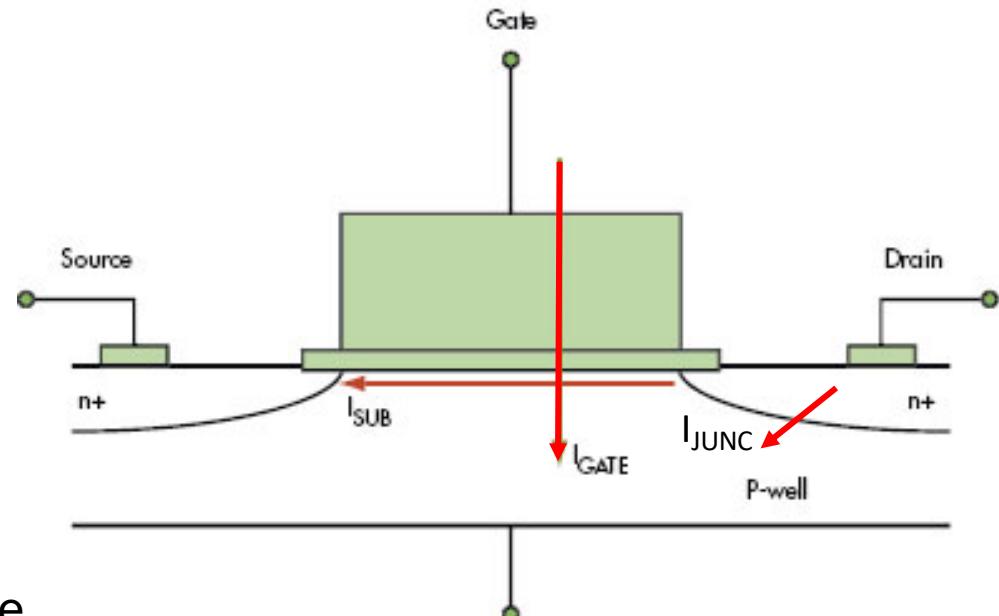


Power Consumption of a CMOS Gate

Inverter:

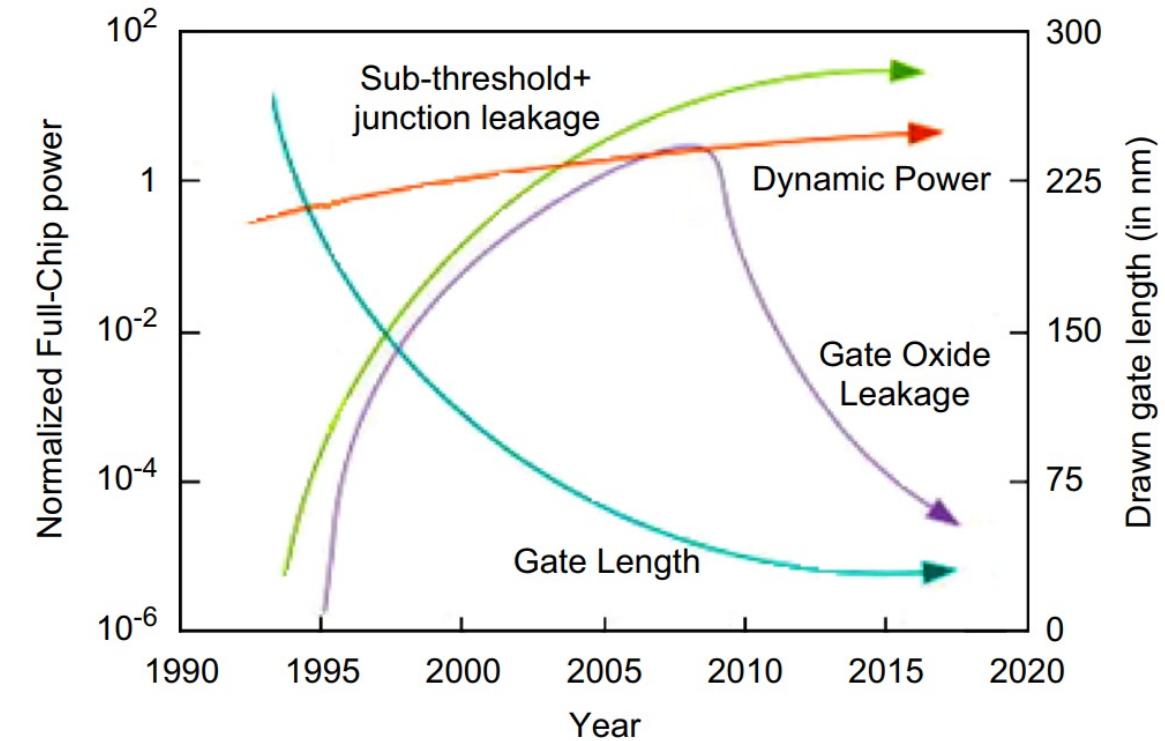


subthreshold (I_{SUB}), junction (I_{JUNC}) and gate-oxide (I_{GATE}) leakage



Main Sources of Power Consumption of a CMOS Processors

- *Dynamic* power consumption:
 - (Dis)charging capacitances while switching
 - Short-circuit power consumption due to a short-circuit path between supply rails while switching
- *Static* power consumption:
 - Gate-oxide, subthreshold, and junction leakage while nothing happens (i.e., no clock, no switching of gate inputs)



[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]

Techniques to Reduce Dynamic Power

Voltage Scaling: A Simple Analytical Model

Average power consumption of CMOS circuits (ignoring leakage):

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

V_{dd} : supply voltage

α : switching activity

C_L : load capacity

f : clock frequency

Delay of CMOS circuits:

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

V_{dd} : supply voltage

V_T : threshold voltage

$$V_T \ll V_{dd}$$

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

But decreasing V_{dd} also increases the gate delay τ reciprocally.

Maximal frequency f_{\max} decreases linearly with decreasing V_{dd} (i.e., “reducing the voltage makes the system slower”).

$$f_{\max} \sim \frac{1}{\tau} \sim \frac{V_{dd}}{C_L}$$

Voltage Scaling: A Simple Analytical Model

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

This is the energy per cycle.

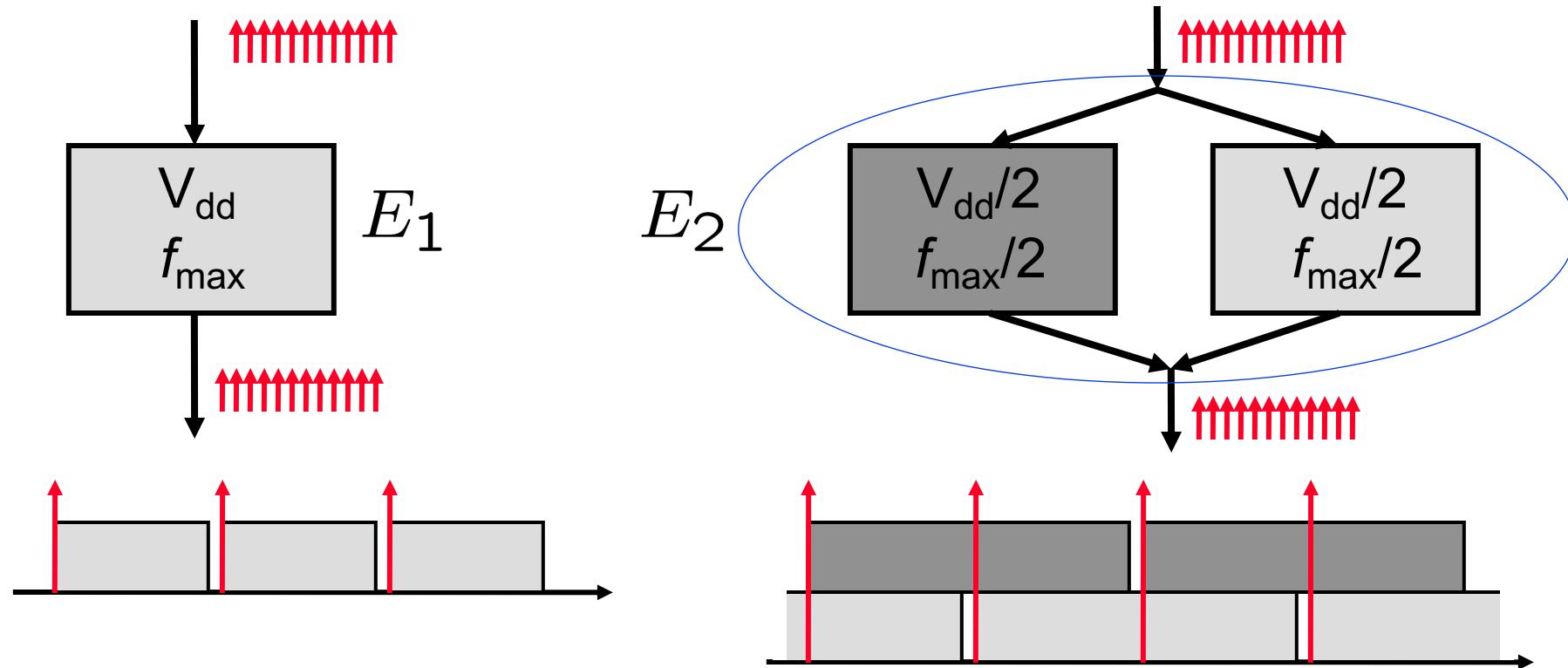
$$E \sim \alpha C_L V_{dd}^2 f t = \alpha C_L V_{dd}^2 (\#cycles)$$

This is the energy needed for executing a task that requires #cycles (i.e., number of cycles).

Saving energy for a given task:

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

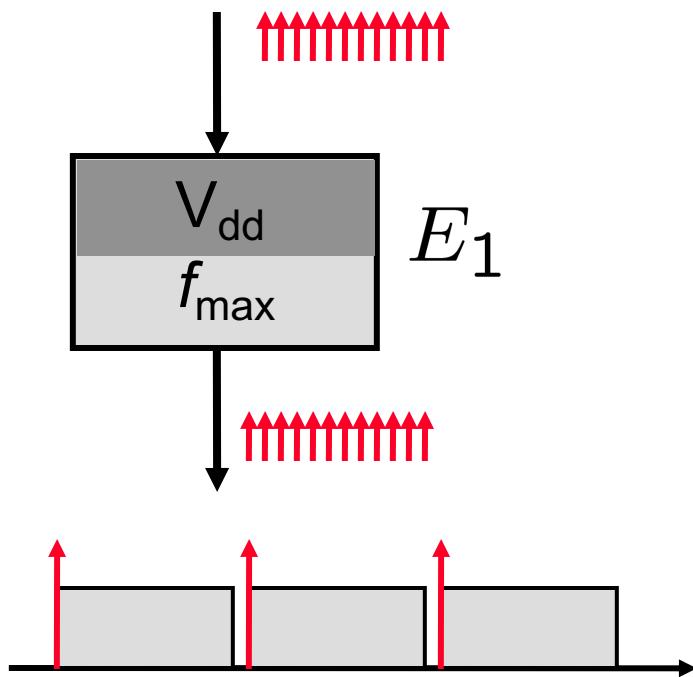
Parallelism



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

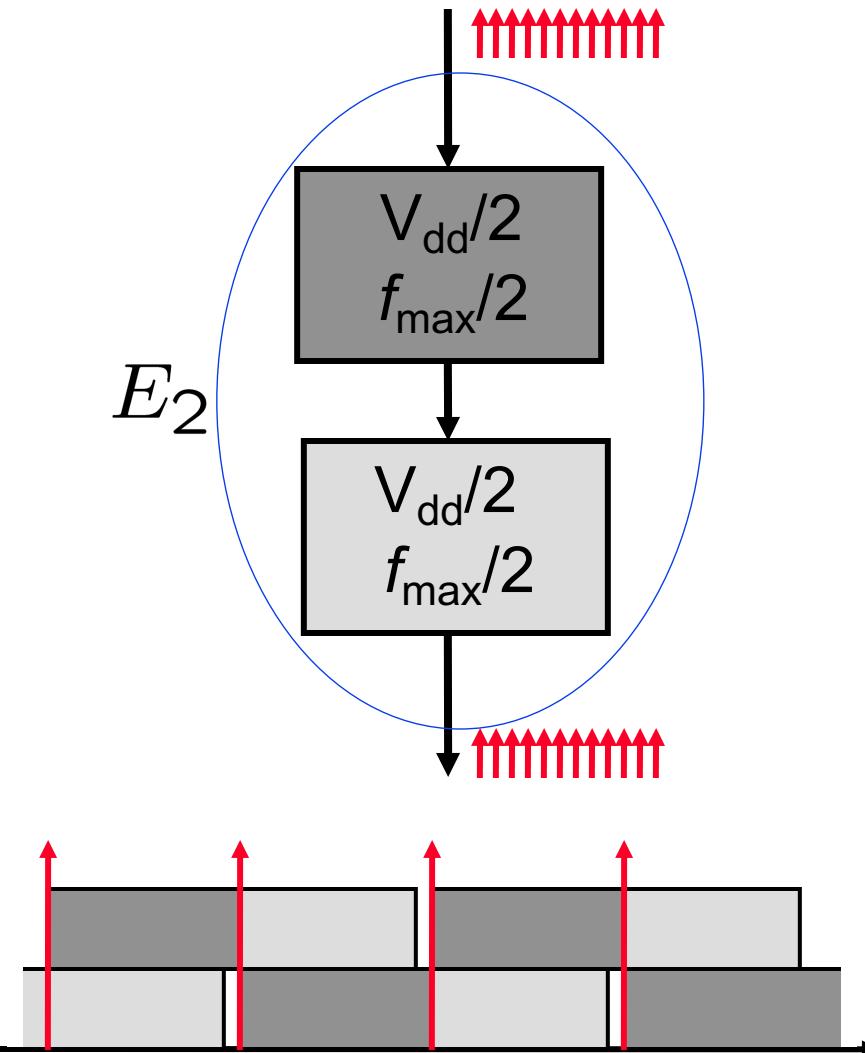
$$E_2 = \frac{1}{4} E_1$$

Pipelining



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

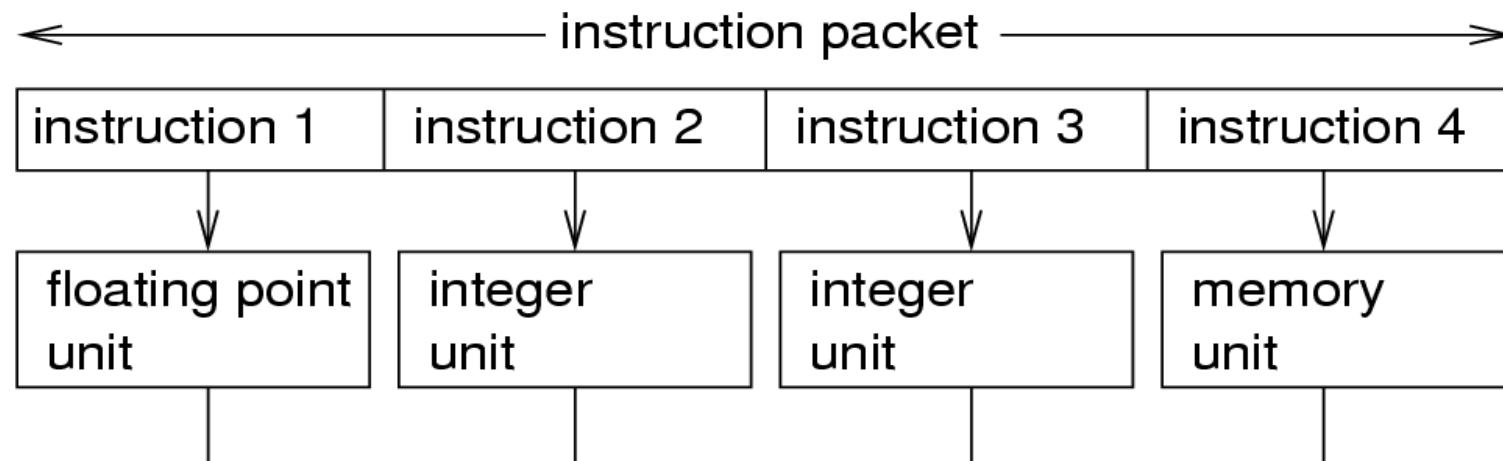
$$E_2 = \frac{1}{4} E_1$$



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)

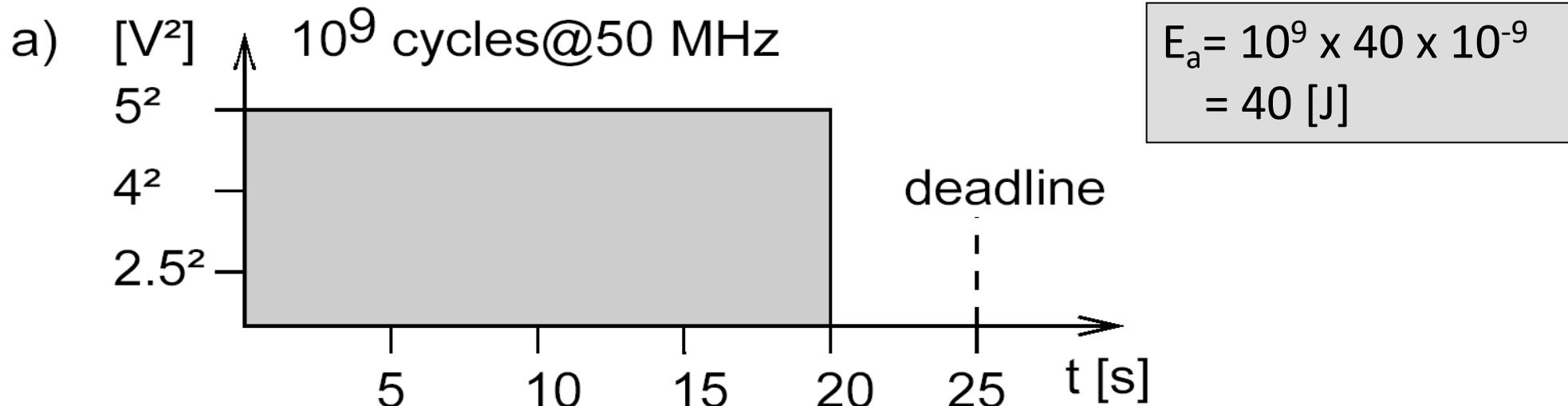
all 4 instructions are
executed in parallel



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

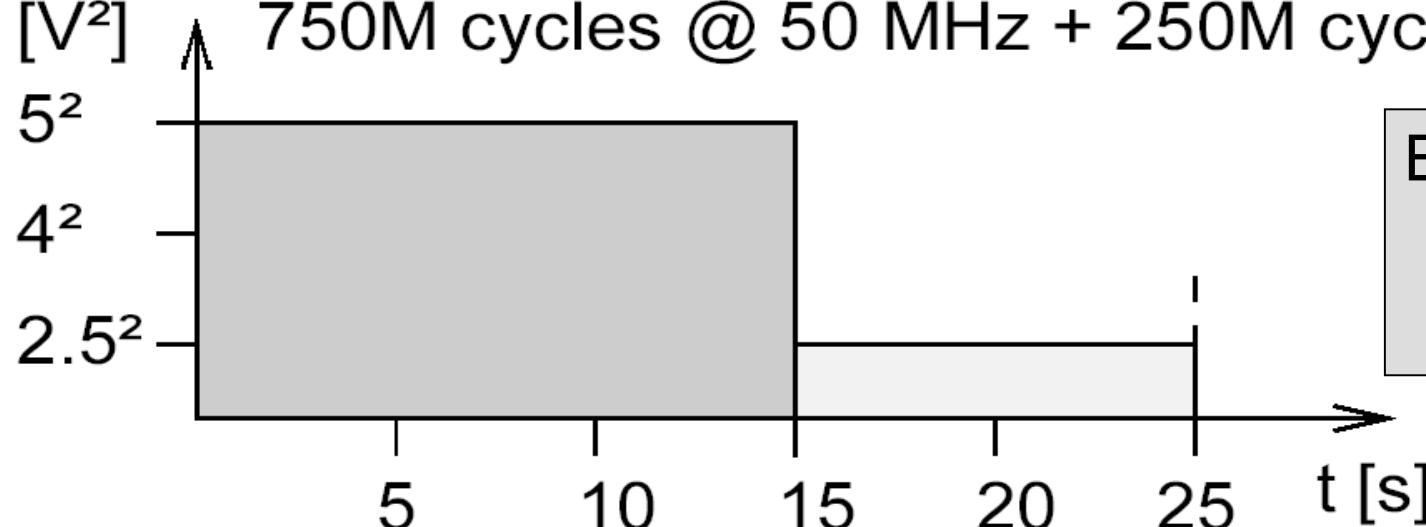
We suppose a task that needs 10^9 cycles to execute within 25 seconds.



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

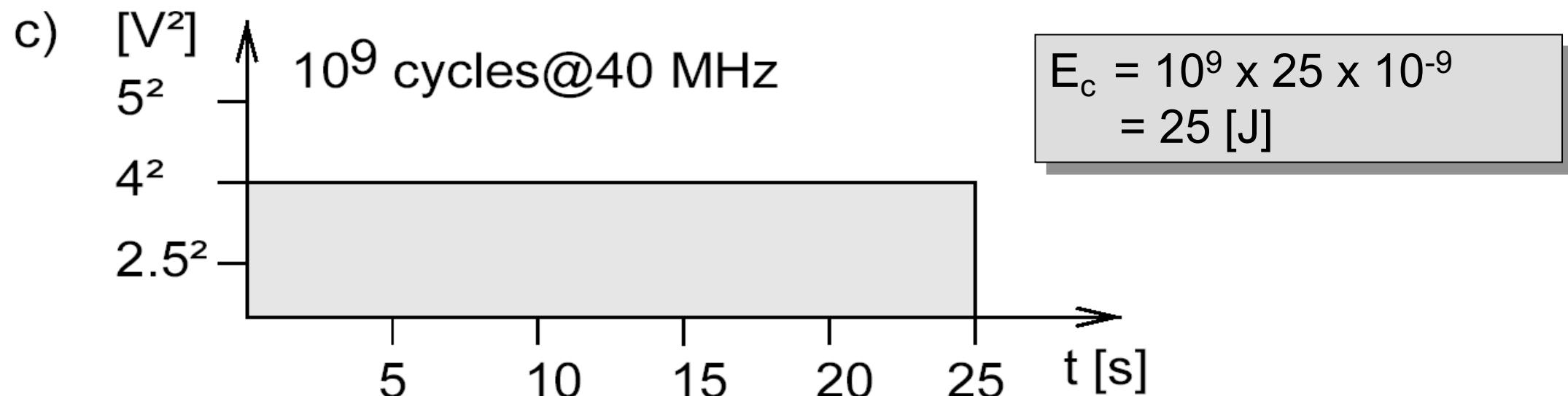
b) $[V^2]$ 750M cycles @ 50 MHz + 250M cycles @ 25 MHz



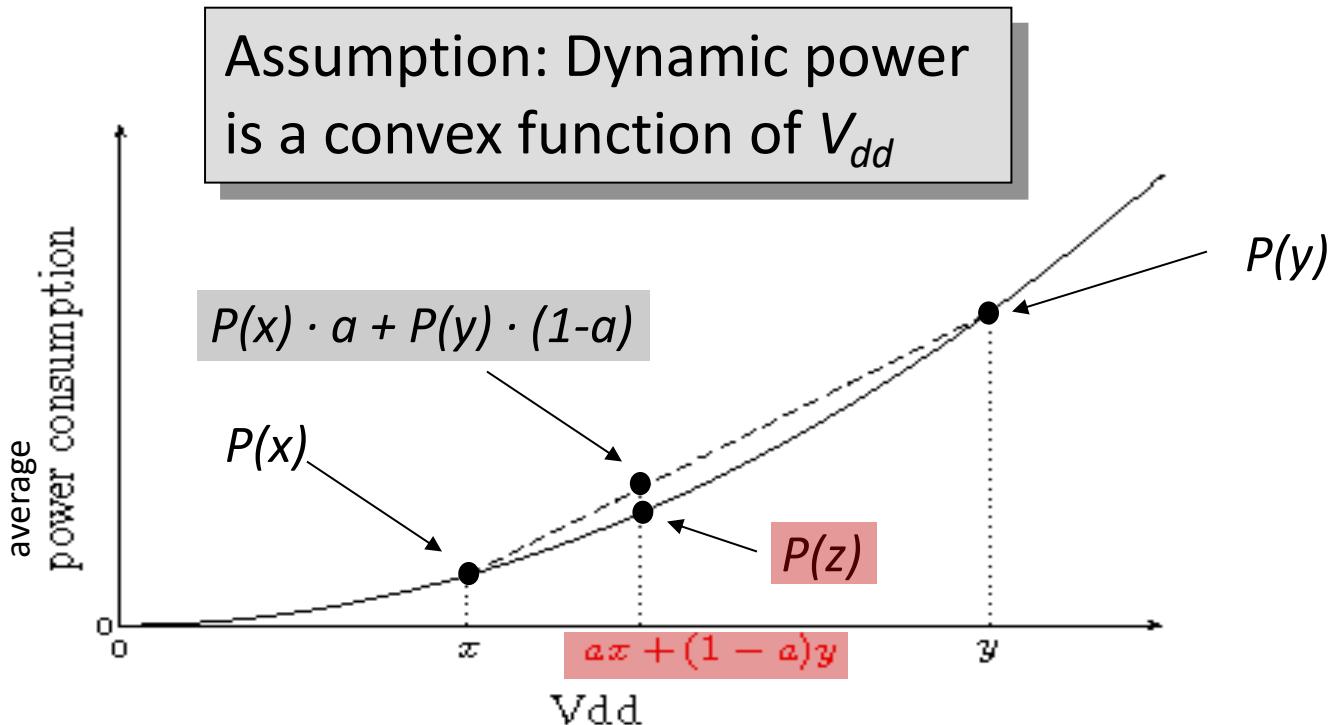
$$\begin{aligned} E_b &= 750 \cdot 10^6 \times 40 \times 10^{-9} \\ &\quad + 250 \cdot 10^6 \times 10 \times 10^{-9} \\ &= 32.5 \text{ [J]} \end{aligned}$$

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



DVFS: Optimal Strategy for Single Tasks



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

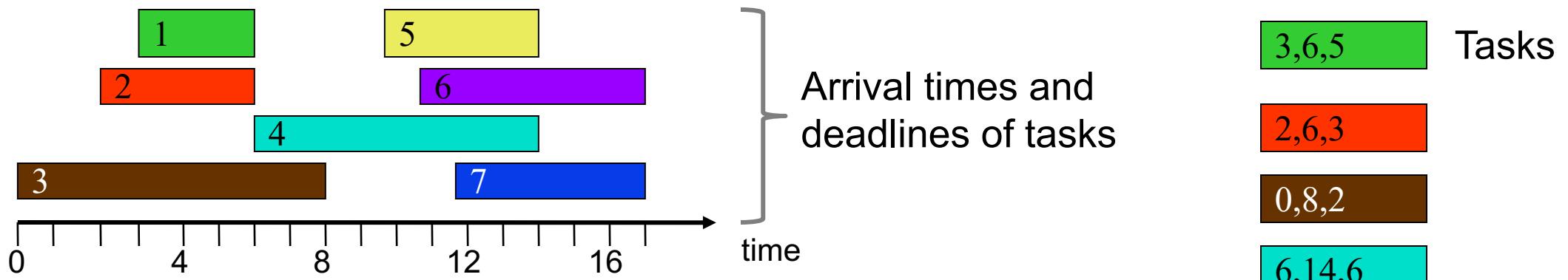
case A is always worse if the power consumption is a convex function of the supply voltage

DVFS: Real-Time Offline Scheduling on One Processor

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



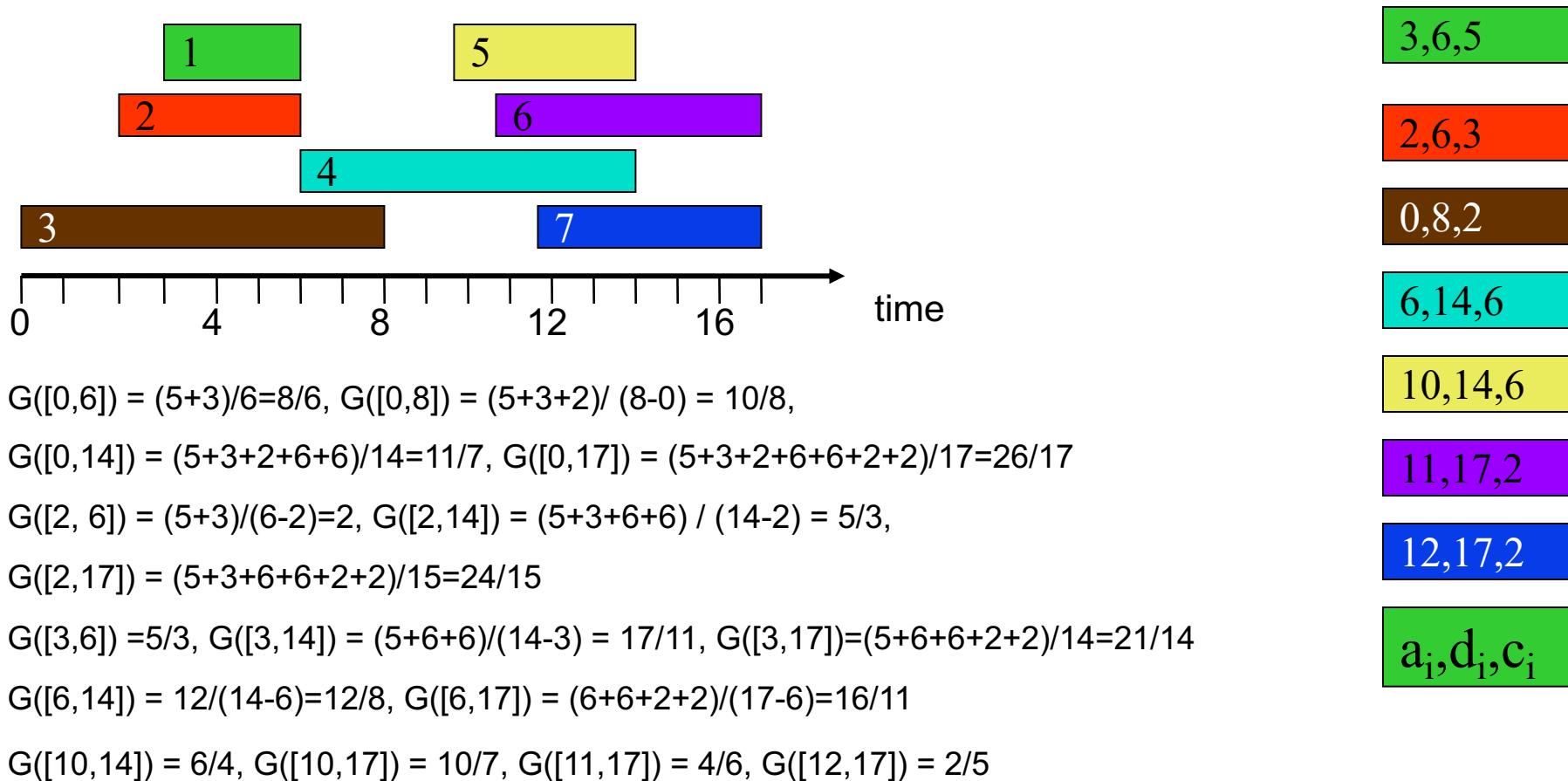
- Define **intensity** $G([z, z'])$ in some time interval $[z, z']$:
 - average accumulated execution time of all tasks that have arrival and deadline in $[z, z']$ relative to the length of the interval $z' - z$

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

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

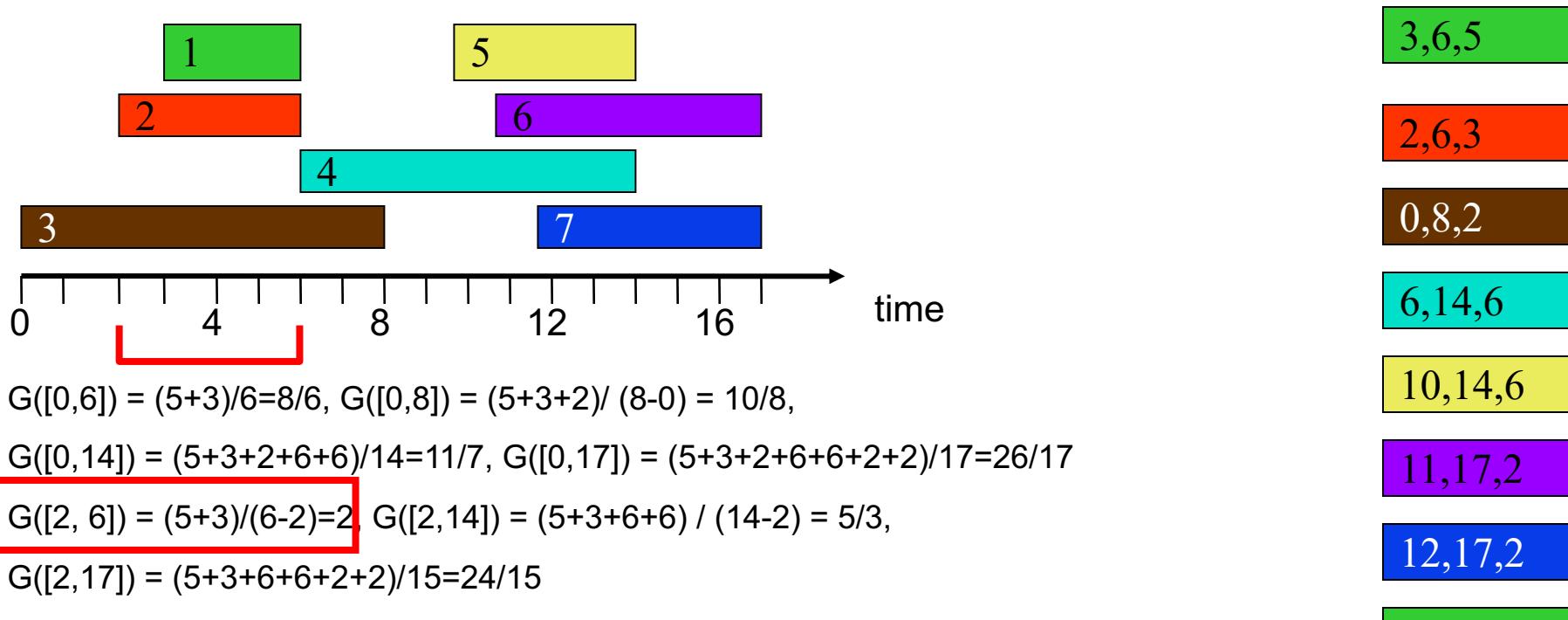
YDS Optimal DVFS Algorithm for Offline Scheduling

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.



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.



*critical
interval*

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

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

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

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

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

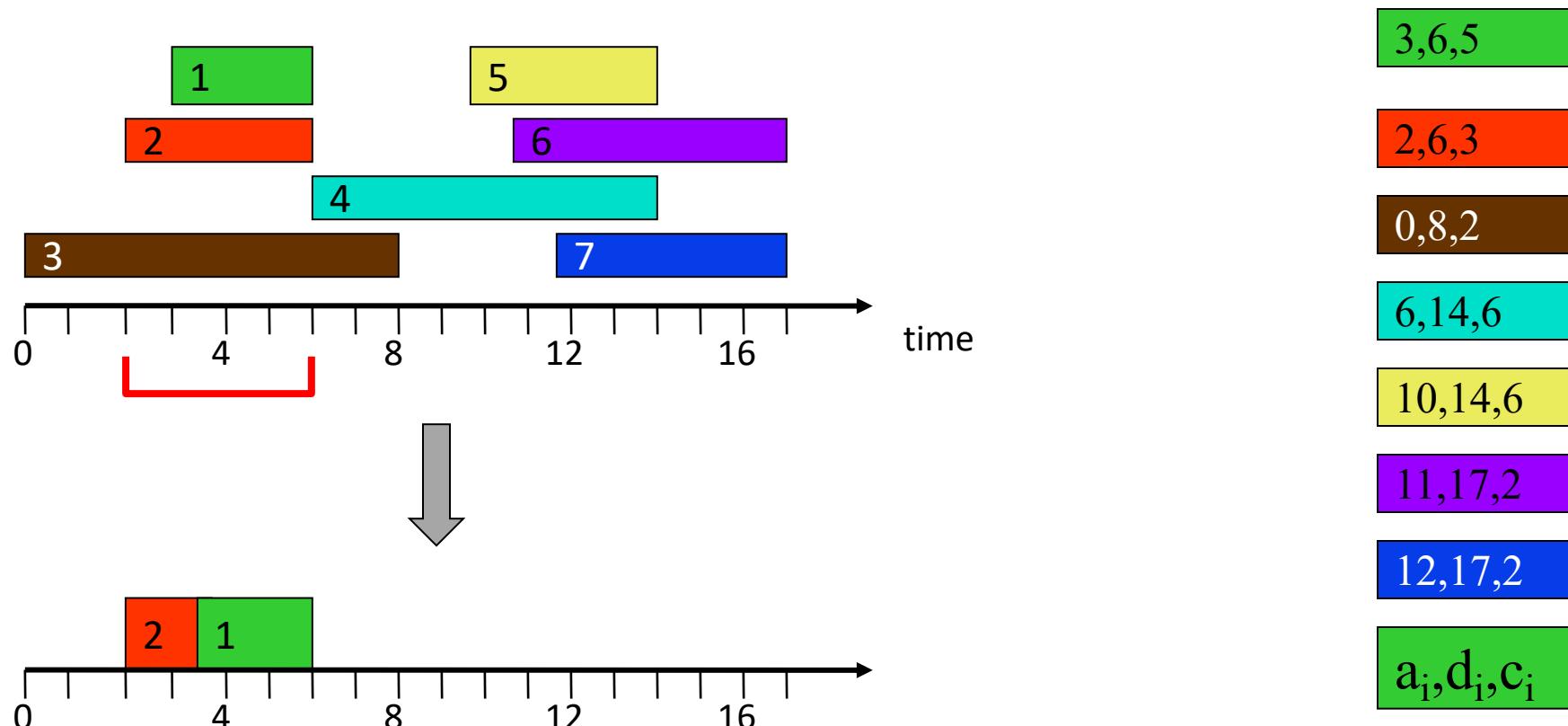
$$G([6,14]) = 12/(14-6)=12/8, G([6,17]) = (6+6+2+2)/(17-6)=16/11$$

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

a_i, d_i, c_i

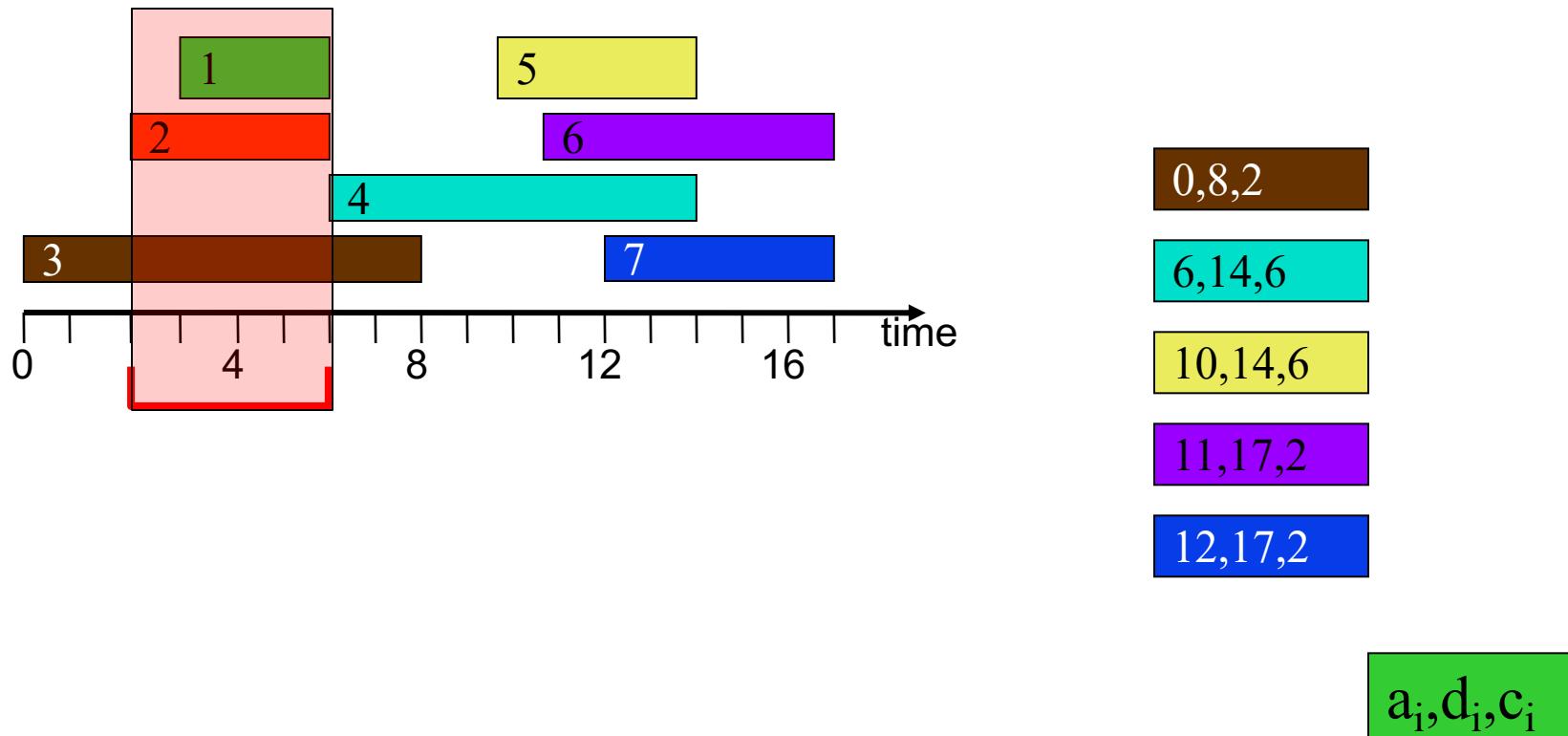
YDS Optimal DVFS Algorithm for Offline Scheduling

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.



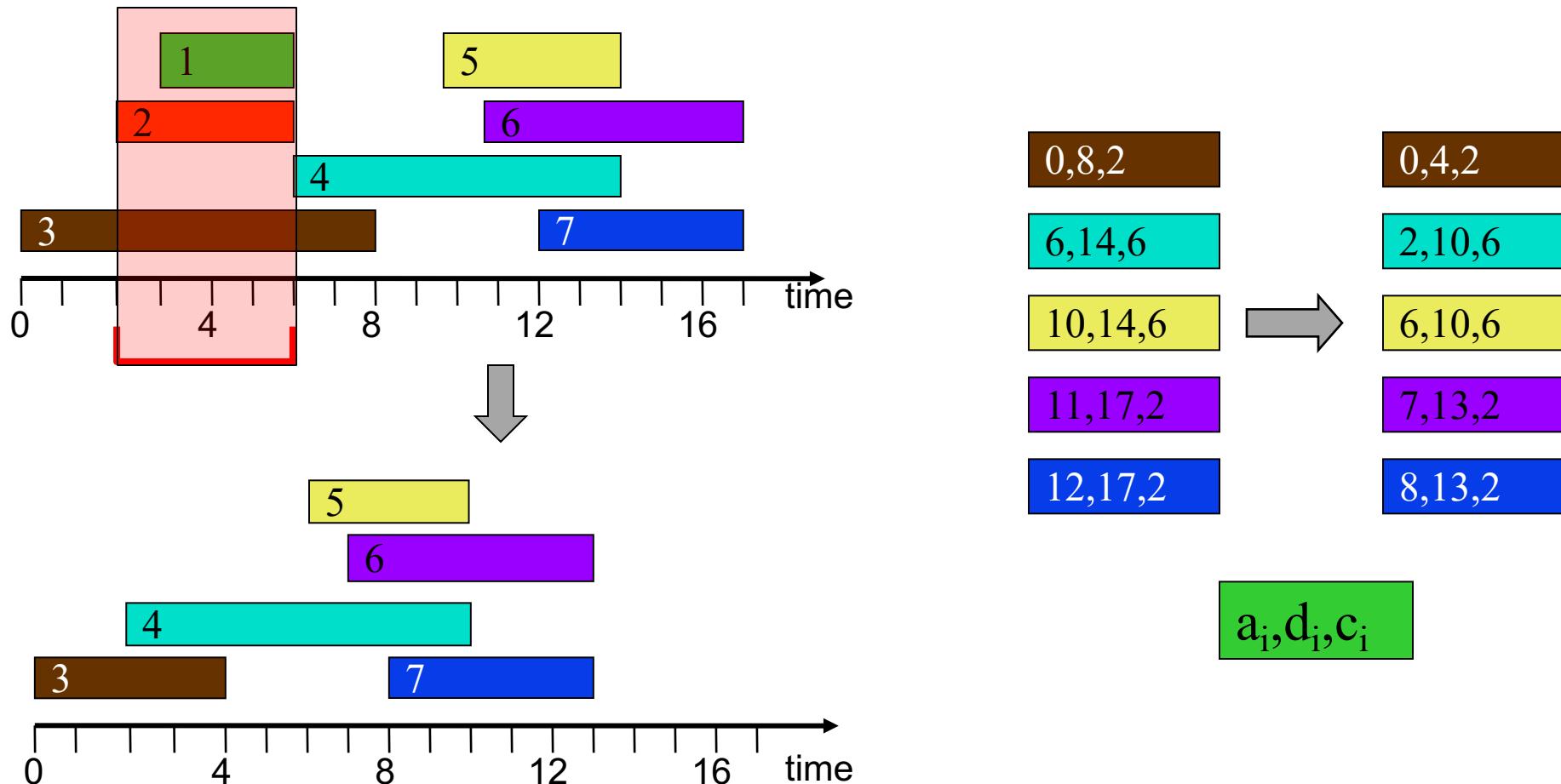
YDS Optimal DVFS Algorithm for Offline Scheduling

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



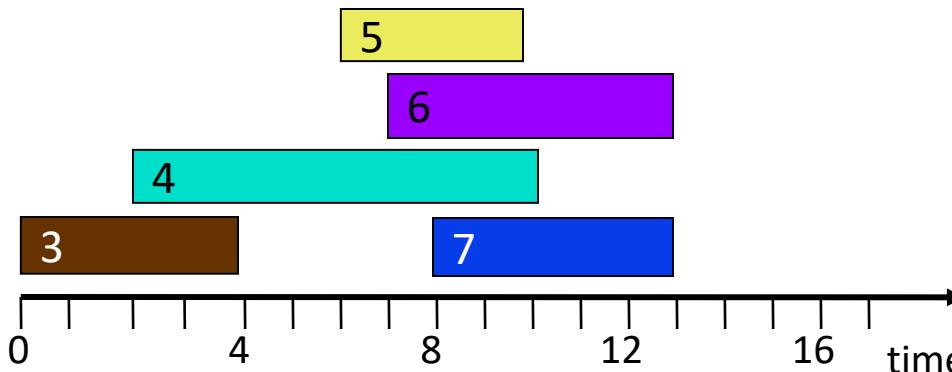
YDS Optimal DVFS Algorithm for Offline Scheduling

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



YDS Optimal DVFS Algorithm for Offline Scheduling

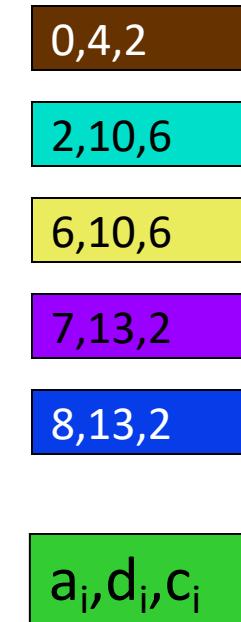
Step 3: Run the algorithm for the revised input again



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

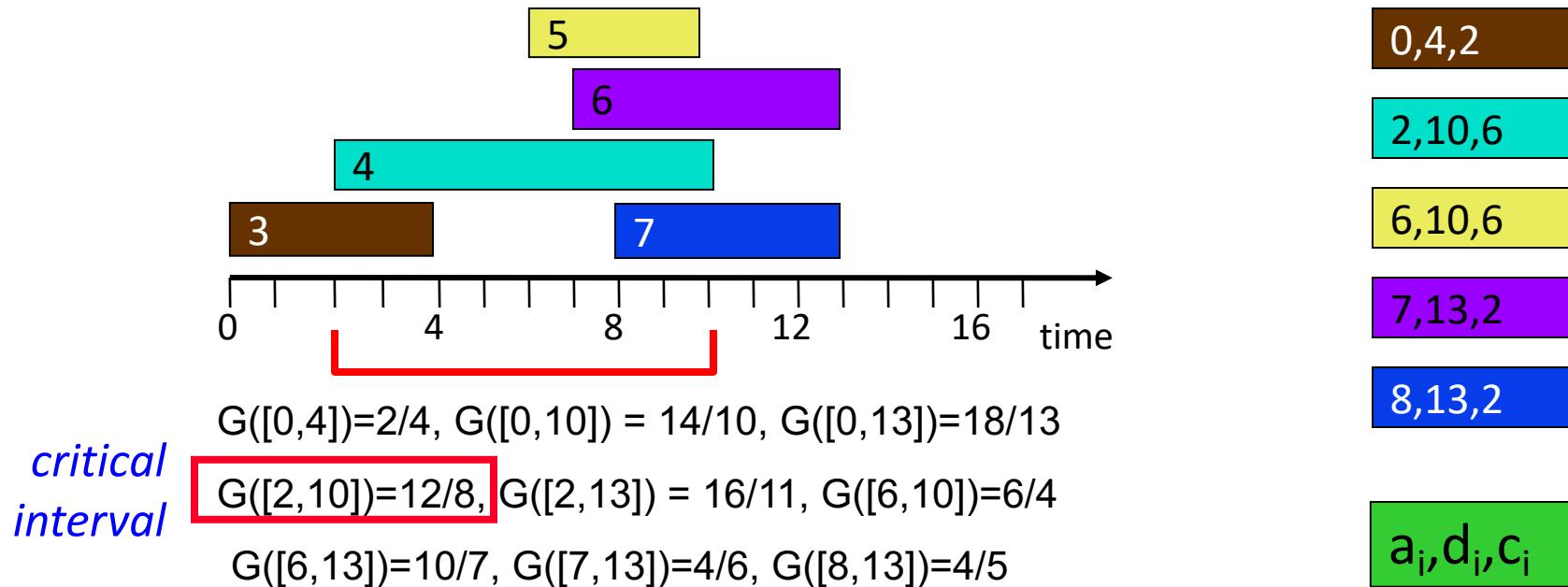
$$G([2,10]) = 12/8, G([2,13]) = 16/11, G([6,10]) = 6/4$$

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



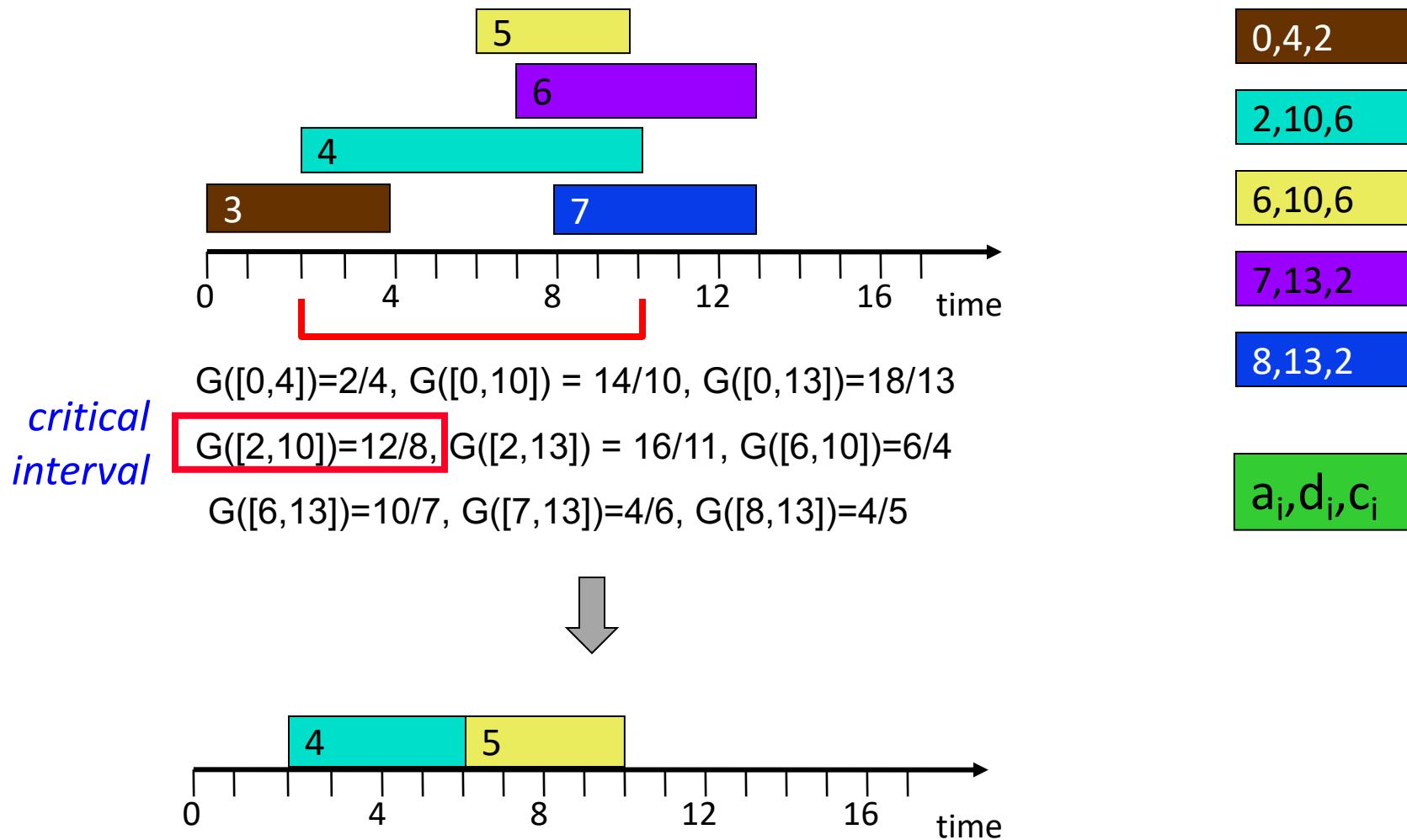
YDS Optimal DVFS Algorithm for Offline Scheduling

Step 3: Run the algorithm for the revised input again



YDS Optimal DVFS Algorithm for Offline Scheduling

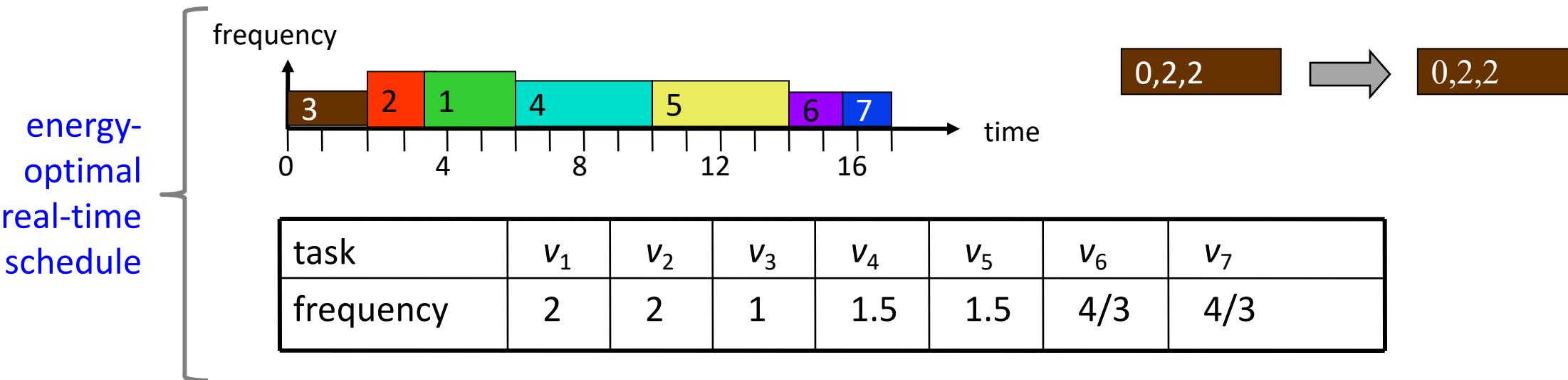
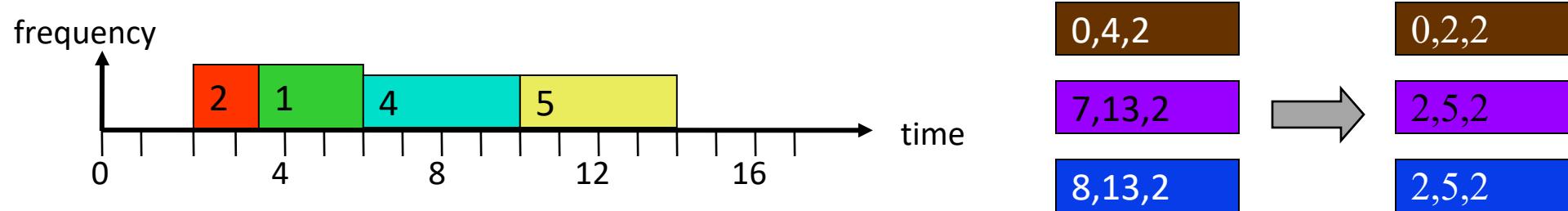
Step 3: Run the algorithm for the revised input again



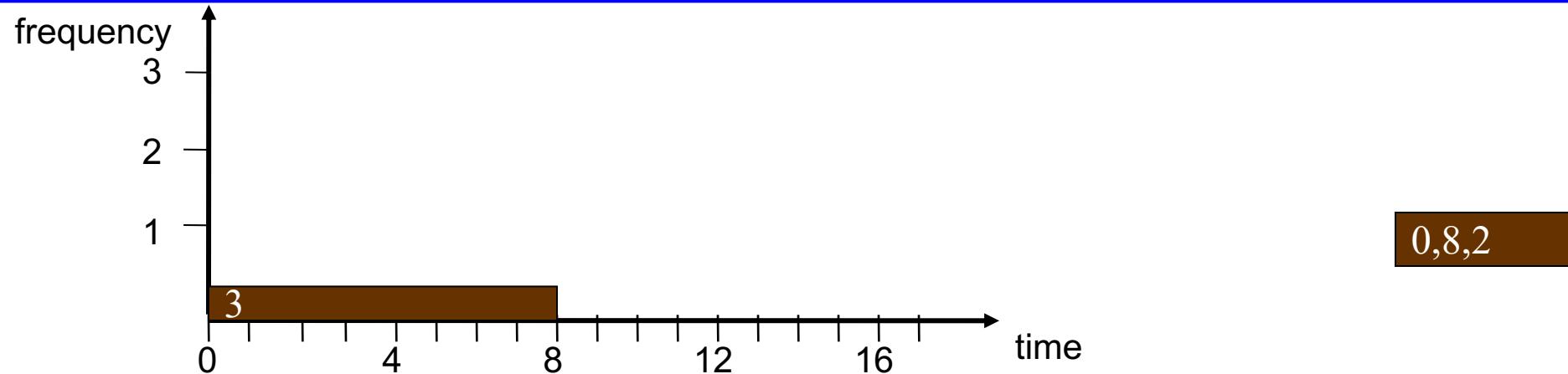
YDS Optimal DVFS Algorithm for Offline Scheduling

Step 3: Run the algorithm for the revised input again

Step 4: Put pieces together



YDS Optimal DVFS Algorithm for Online Scheduling

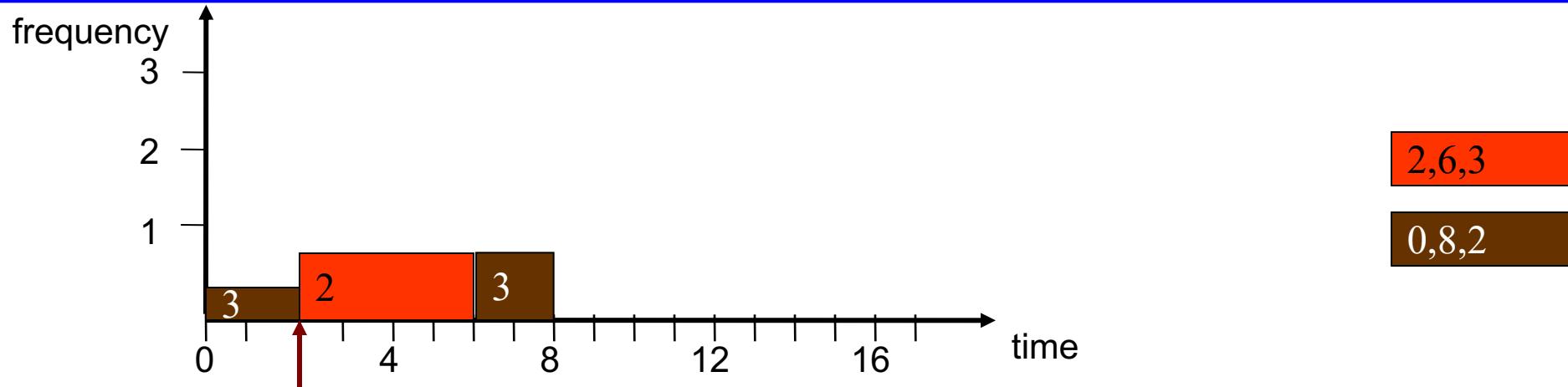


Continuously update to the best schedule for all arrived tasks:

Time 0: task v_3 is executed at 2/8

a_i, d_i, c_i

YDS Optimal DVFS Algorithm for Online Scheduling



Continuously update to the best schedule for all arrived tasks:

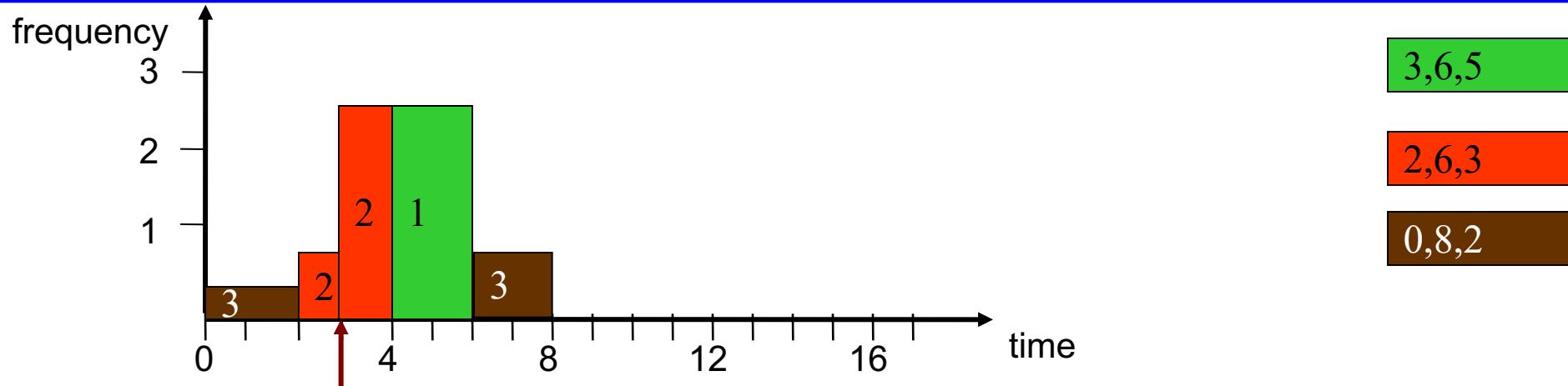
Time 0: task v_3 is executed at 2/8

Time 2: task v_2 arrives

- $G([2,6]) = 3/4$, $G([2,8]) = 4.5/6=3/4 \Rightarrow$ execute v_3, v_2 at 3/4

a_i, d_i, c_i

YDS Optimal DVFS Algorithm for Online Scheduling



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

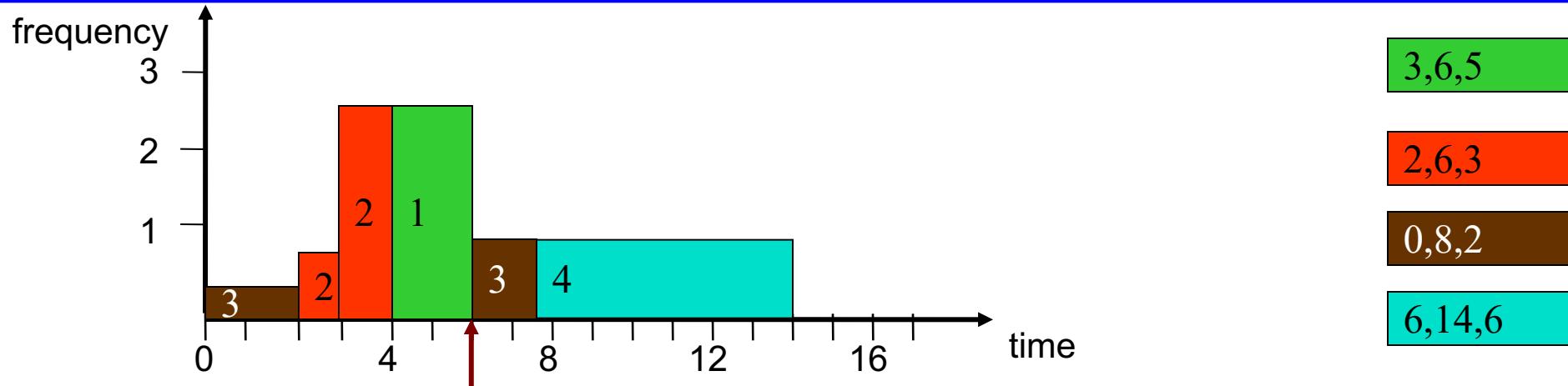
- $G([2,6]) = 3/4$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v_3, v_2 at 3/4

Time 3: task v_1 arrives

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

a_i, d_i, c_i

YDS Optimal DVFS Algorithm for Online Scheduling



Continuously update to the best schedule for all arrived tasks:

Time 0: task v₃ is executed at 2/8

Time 2: task v₂ arrives

- $G([2,6]) = 3/4$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v₃, v₂ at 3/4

Time 3: task v₁ arrives

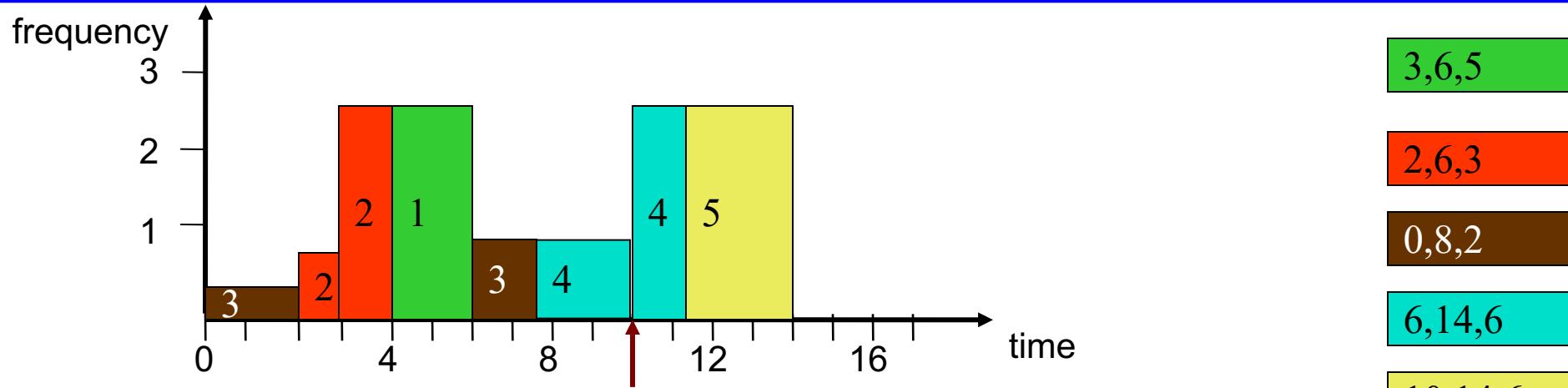
- $G([3,6]) = (5+3-3/4)/3 = 29/12$, $G([3,8]) < G([3,6]) \Rightarrow$ execute v₂ and v₁ at 29/12

Time 6: task v₄ arrives

- $G([6,8]) = 1.5/2$, $G([6,14]) = 7.5/8 \Rightarrow$ execute v₃ and v₄ at 15/16

a_i,d_i,c_i

YDS Optimal DVFS Algorithm for Online Scheduling



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

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

Time 3: task v_1 arrives

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

a_i, d_i, c_i

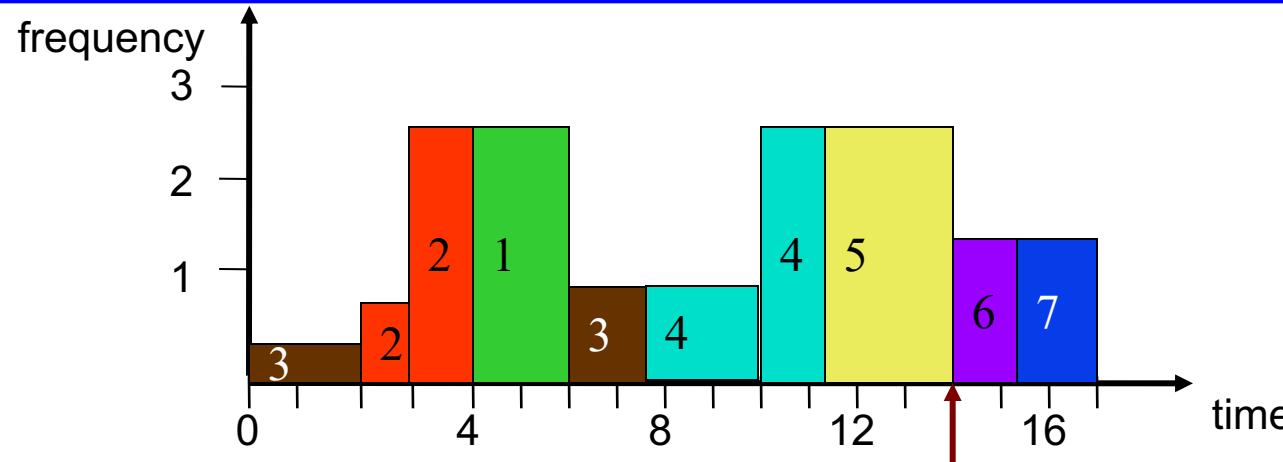
Time 6: task v_4 arrives

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

Time 10: task v_5 arrives

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

YDS Optimal DVFS Algorithm for Online Scheduling



Continuously update to the best schedule for all arrived tasks:

Time 0: task v₃ is executed at 2/8

Time 2: task v₂ arrives

- $G([2,6]) = 3/4$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v₈, v₂ at 3/4

Time 3: task v₁ arrives

- $G([3,6]) = (5+3-3/4)/3 = 29/12$, $G([3,8]) < G([3,6]) \Rightarrow$ execute v₂ and v₁ at 29/12

Time 6: task v₄ arrives

- $G([6,8]) = 1.5/2$, $G([6,14]) = 7.5/8 \Rightarrow$ execute v₃ and v₄ at 15/16

Time 10: task v₅ arrives

- $G([10,14]) = 39/16 \Rightarrow$ execute v₄ and v₅ at 39/16

Time 11 and Time 12

- The arrival of v₆ and v₇ does not change the critical interval

Time 14:

- $G([14,17]) = 4/3 \Rightarrow$ execute v₆ and v₇ at 4/3

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

Remarks on the YDS Algorithm

- *Offline*
 - The algorithm guarantees the minimal energy consumption while satisfying the timing constraints
 - The time complexity is about $O(N^3)$, where N is the number of tasks in V
 - Using a simple approach, 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.

Techniques to Reduce Static Power

Dynamic Power Management (DPM)

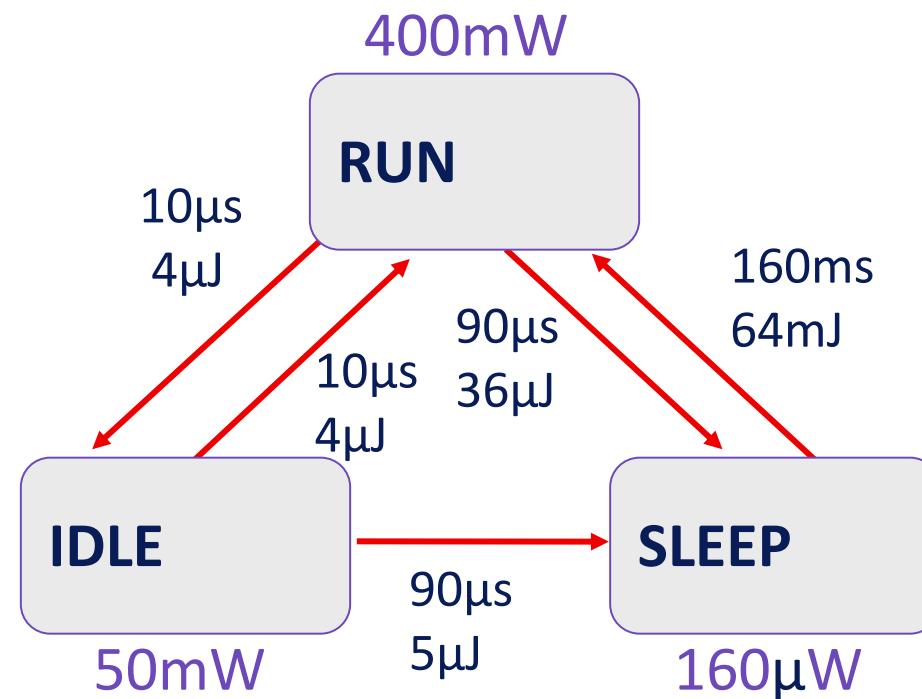
- Dynamic power management (DPM) tries to assign optimal power saving states during program execution
- DPM requires hardware and software support

Example: StrongARM SA1100

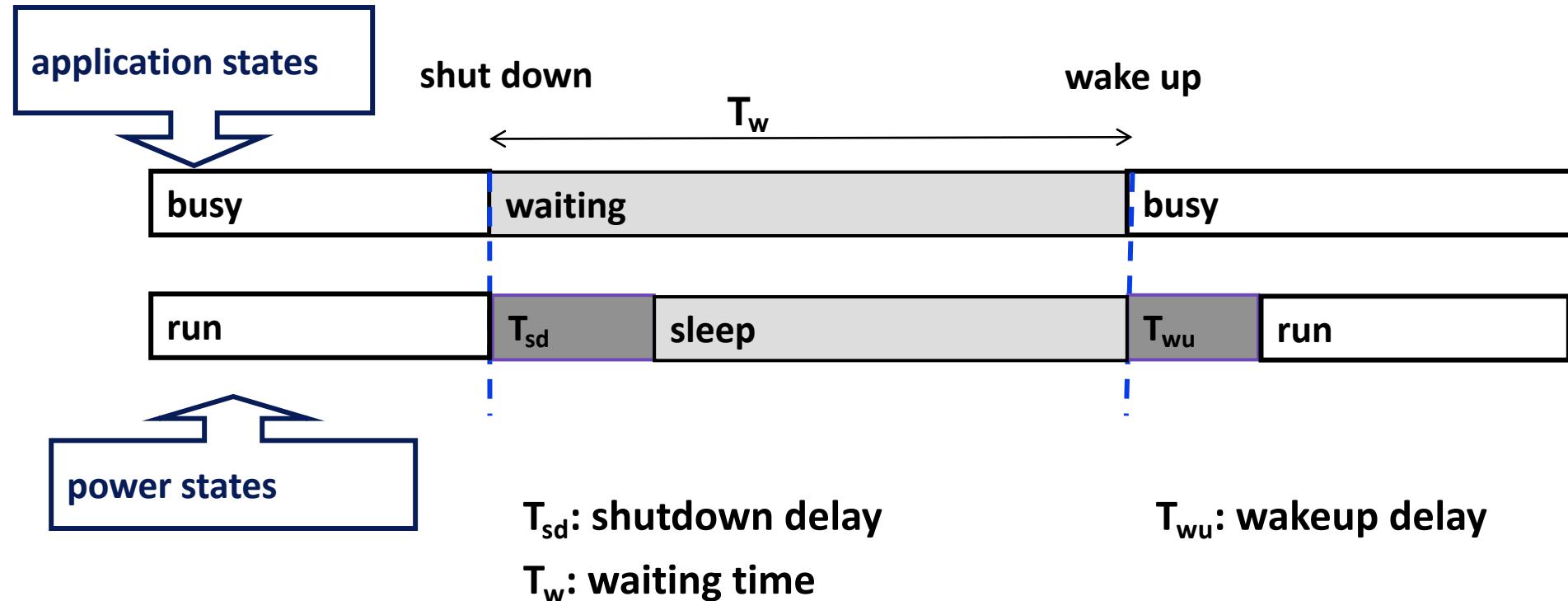
RUN: operational

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)

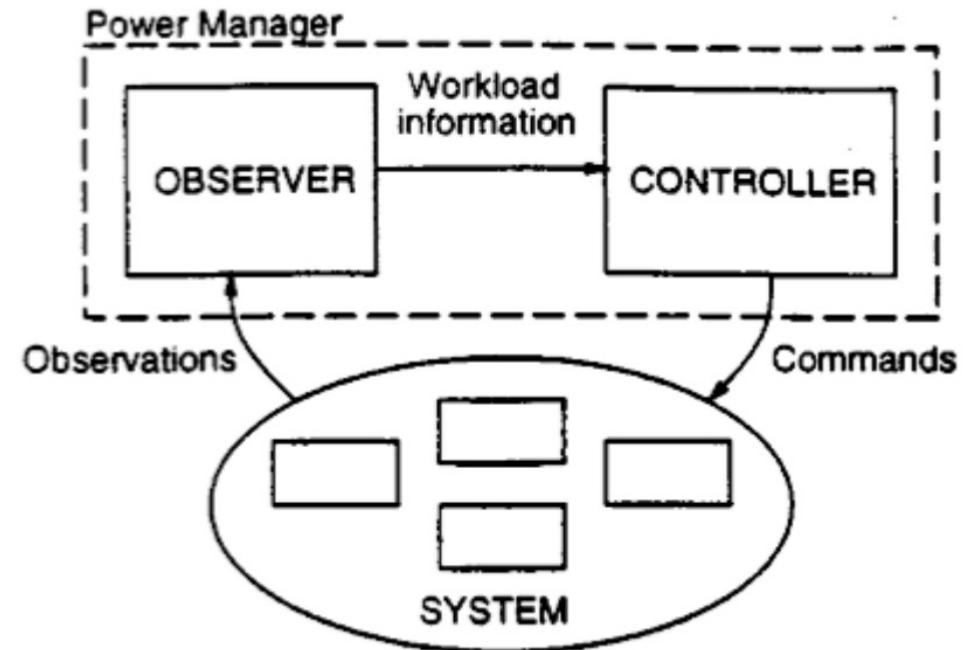


Desired: Shutdown only during long waiting times. This leads to a trade-off between energy saving and overhead.

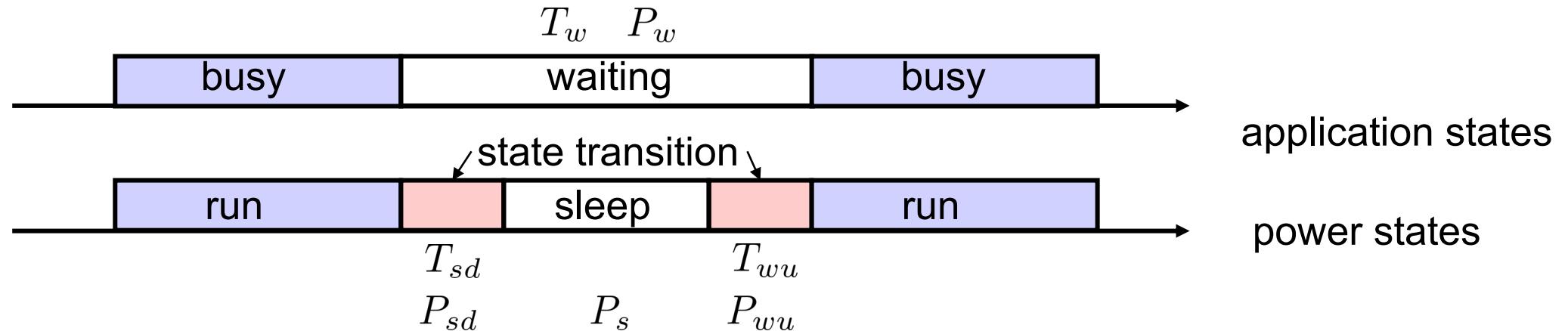
Break-Even Time

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

- Entering 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): $E_2 = T_{sd} \cdot P_{sd} + T_{wu} \cdot P_{wu} + (T_w - T_{sd} - T_{wu}) \cdot P_s$

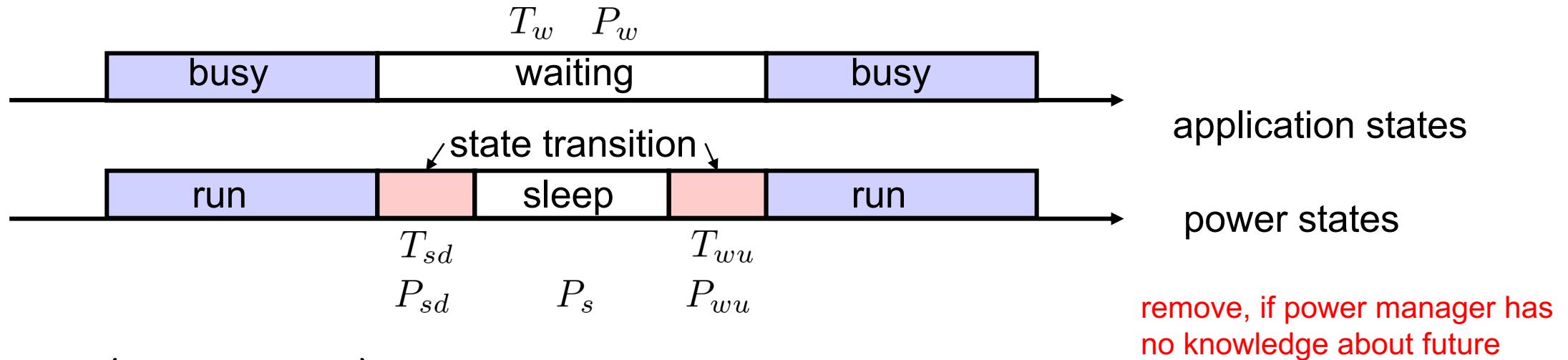
Break-even time: Limit for T_w such that $E_2 \leq E_1$

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

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

break-even
time

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:

$$T_w \geq \frac{T_{sd} \cdot (P_{sd} - P_s) + T_{wu} \cdot (P_{wu} - P_s)}{P_w - P_s}$$

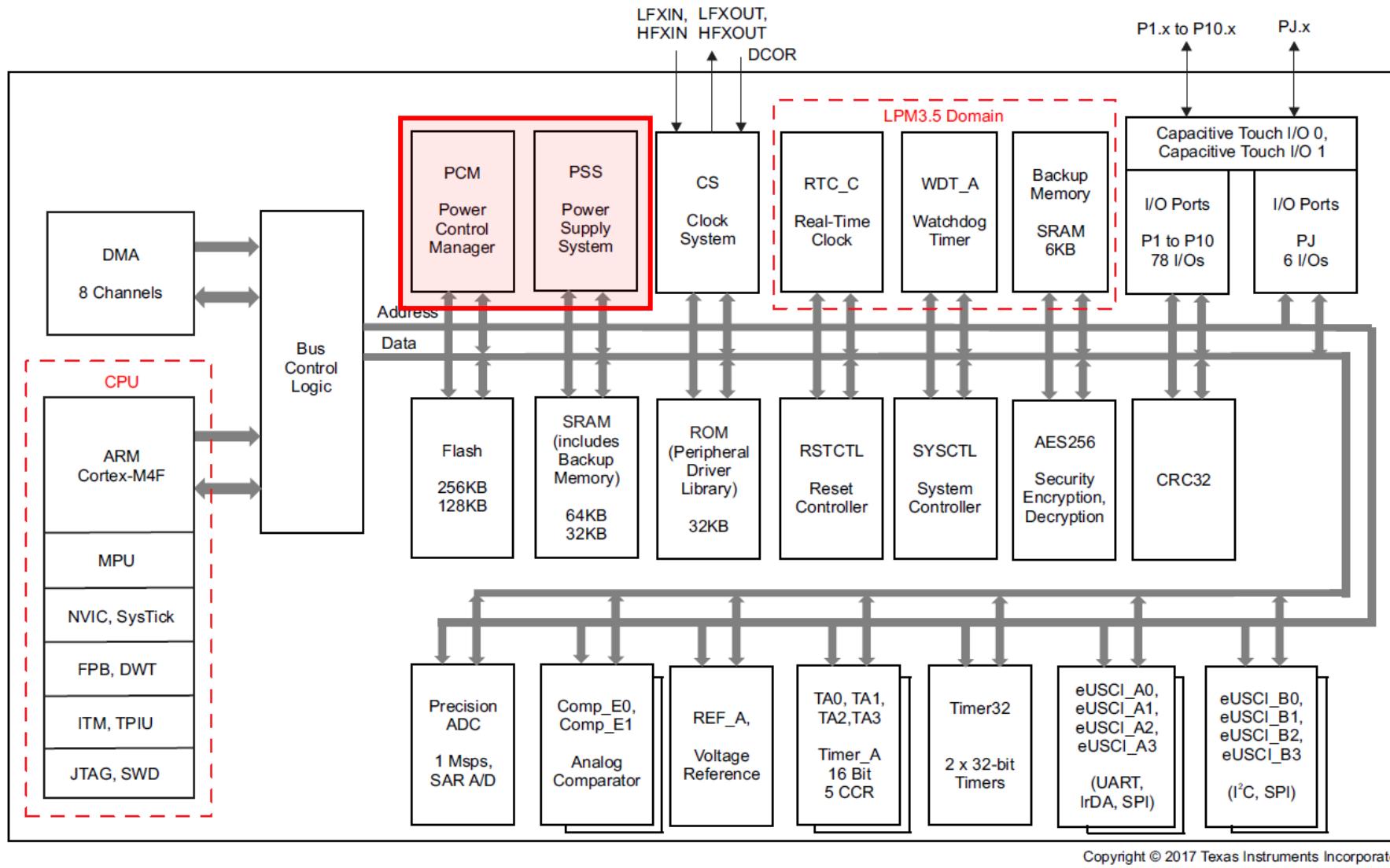
Time constraint:

$$T_w \geq T_{sd} + T_{wu}$$

remove, if power manager has
no knowledge about future

break-even
time

Power Modes of MSP432



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

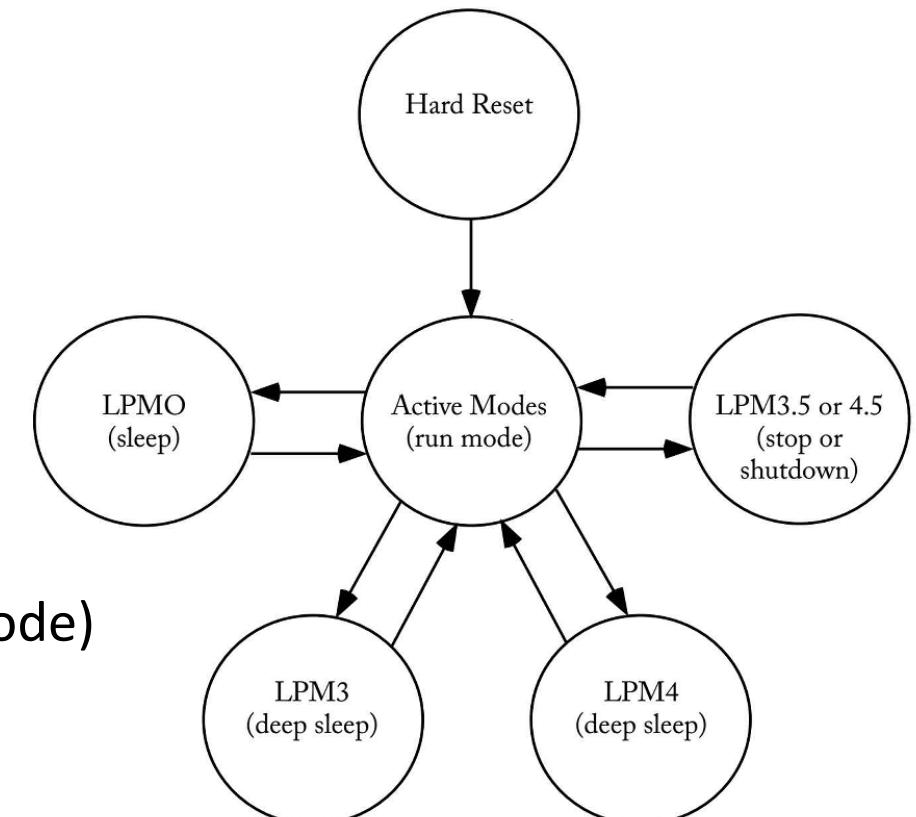
It has 5 major low power modes (LP0, LP3, LP4, LP3.5, LP4.5), some of them can be in one of several configurations.

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 μ W

Power Modes of MSP432

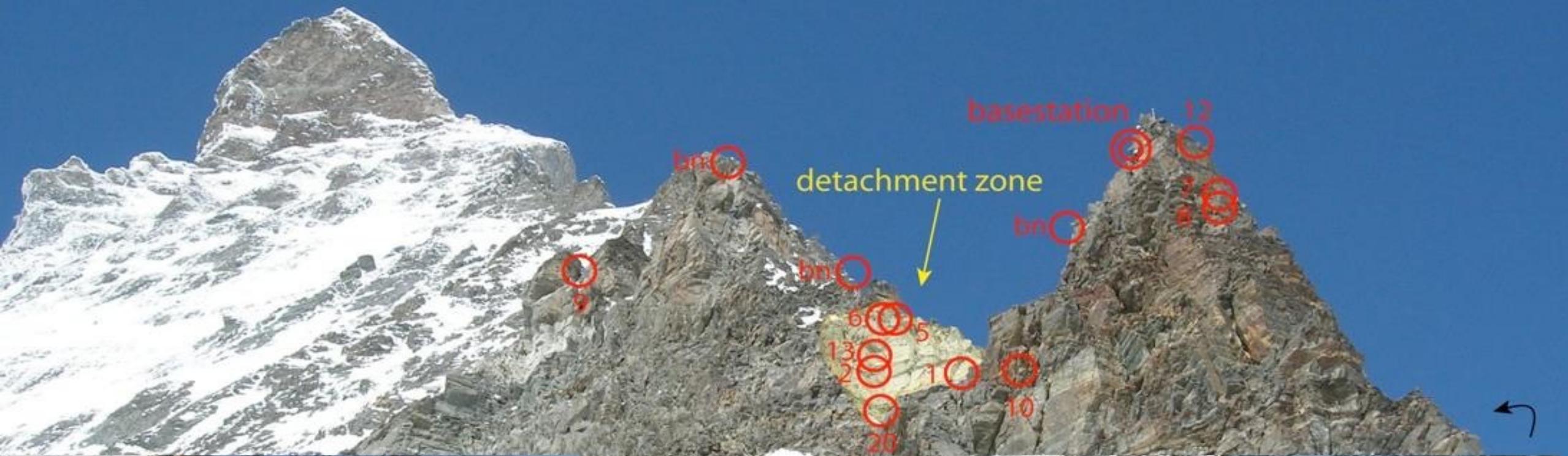
- Transition between modes can be handled using C-level interfaces to the power control manager.
- Examples of interface functions:
 - `uint8_t PCM_getPowerState (void)`
 - `bool PCM_gotoLPM0 (void)`
 - `bool PCM_gotoLPM3 (void)`
 - `bool PCM_gotoLPM4 (void)`
 - `bool PCM_shutdownDevice (uint32_t shutdownMode)`



Battery-Operated Systems and Energy Harvesting

Embedded Systems in the Extreme – PermaSense







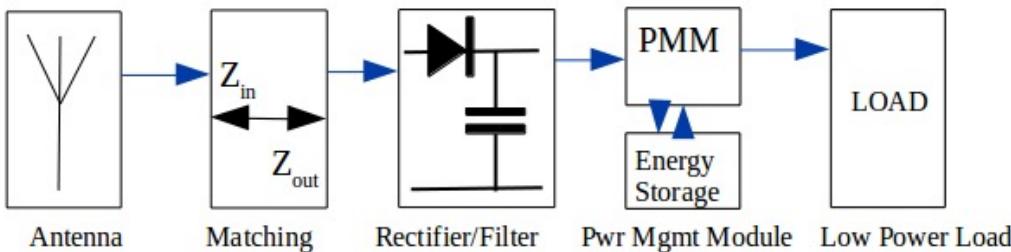
Reasons for Battery-Operated Devices and Harvesting

- Battery operation:

- no continuous power source available
- mobility

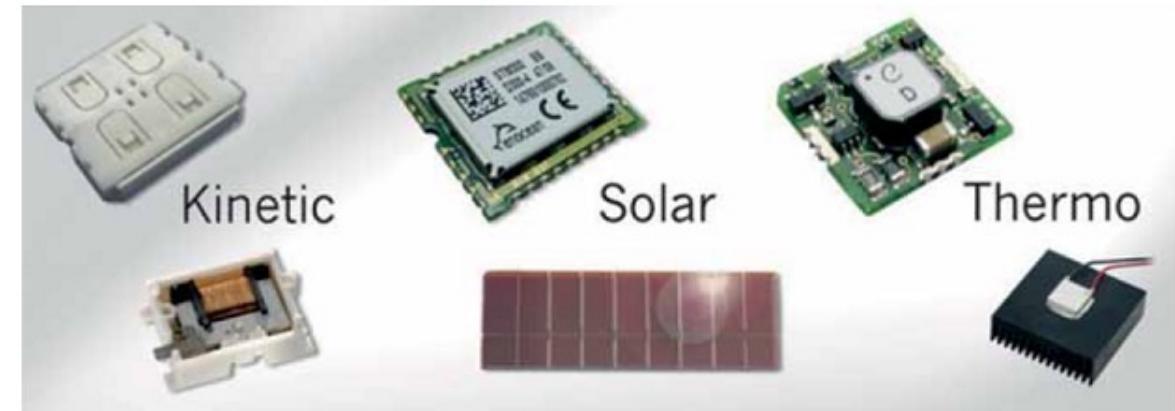
- Energy harvesting:

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

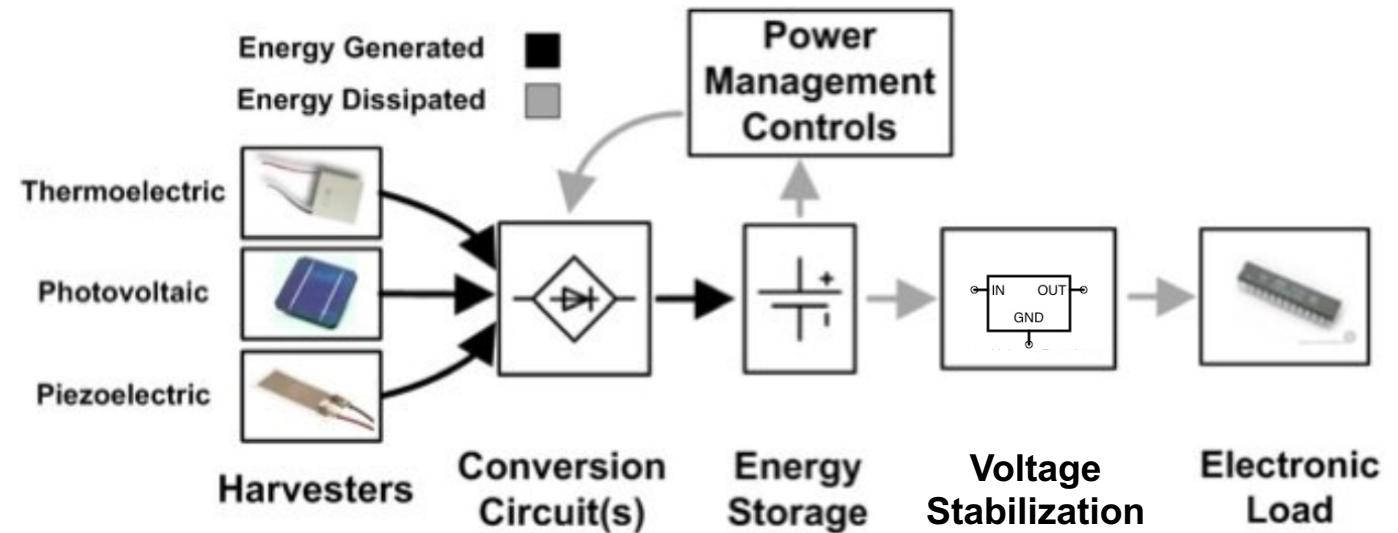


radio frequency (RF) harvesting

Nest
2.0



Typical Power Circuitry – Power Point Tracking

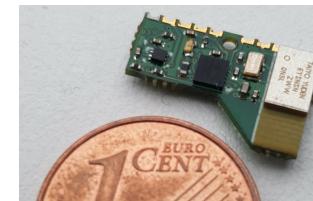


power point tracking / impedance matching; conversion to voltage of energy storage

rechargeable battery, supercapacitor, or multi-layer ceramic capacitor (MLCC)

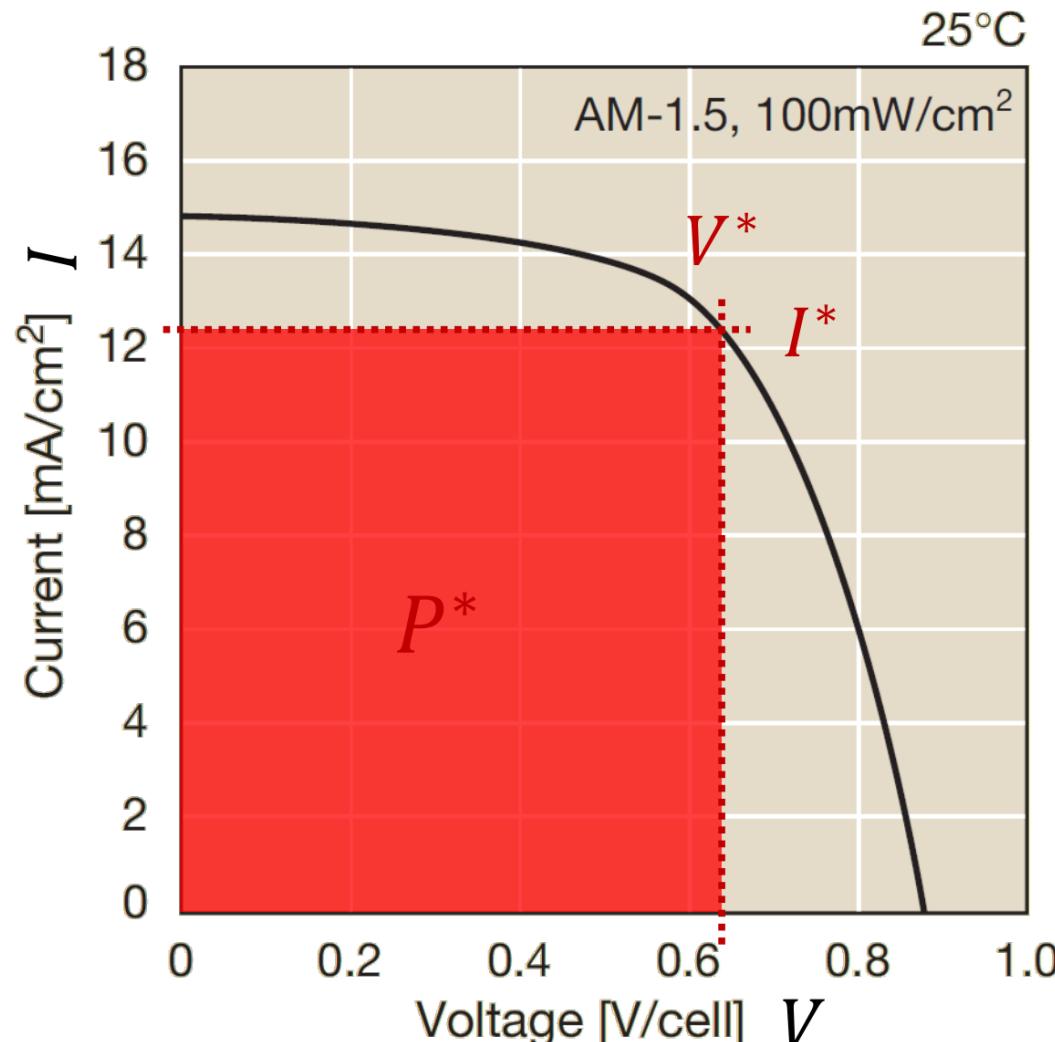


MLCC



battery-free
IoT device

Solar Panel Characteristics

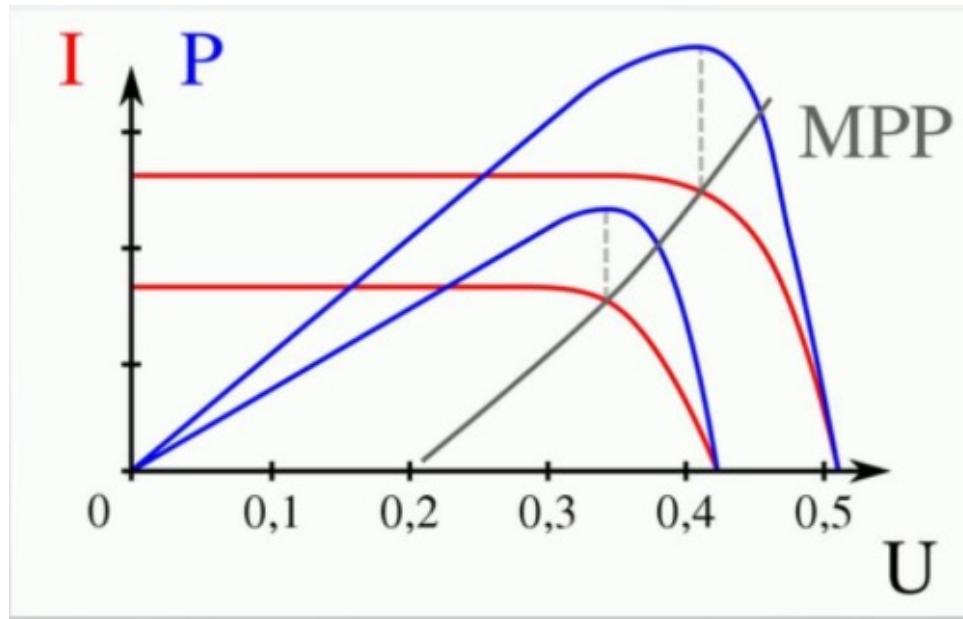


- Variable output power
 - Illuminance level
 - Electrical operation point
 - (Temperature, age, ...)
- I-V-Characteristics
 - Non-linear
 - Dependent on ambient
- Maximum Power Point Tracking
 - Dynamic algorithm to find P^*

Diagram: Amorton Amorphous Silicon Solar Cells Datasheet, © Panasonic

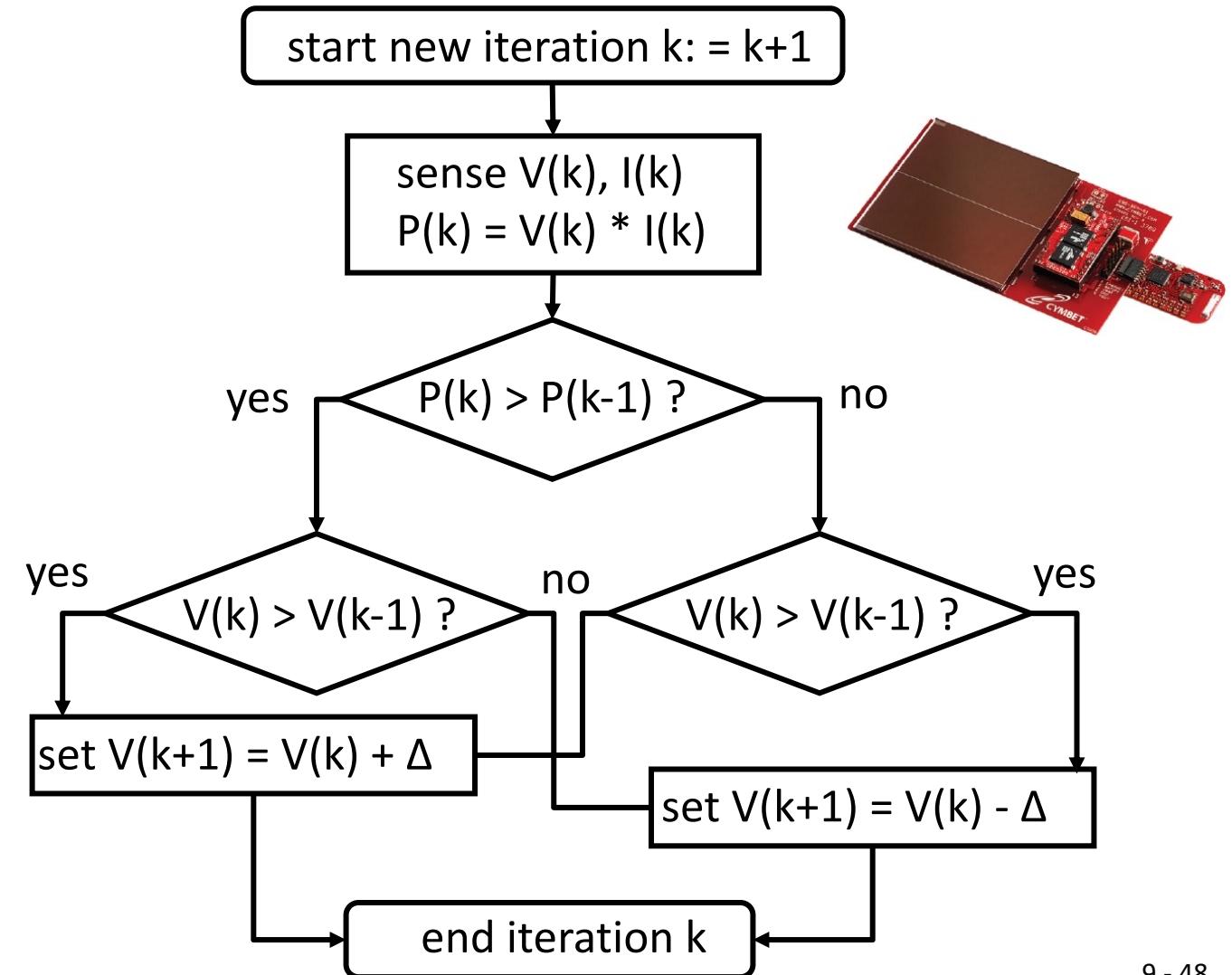
Typical Power Circuitry – Maximum Power Point Tracking

U/I curves of a typical solar cell:

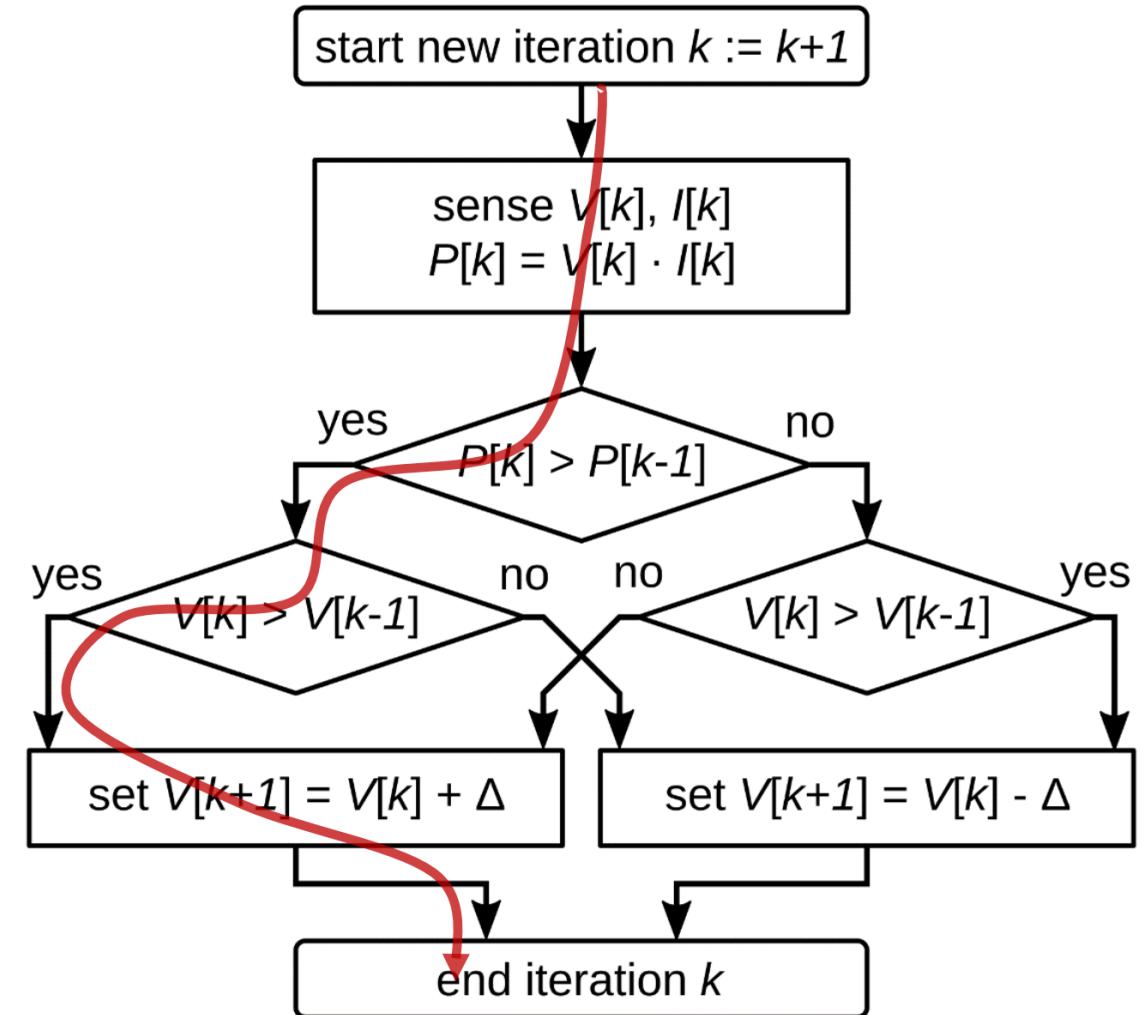
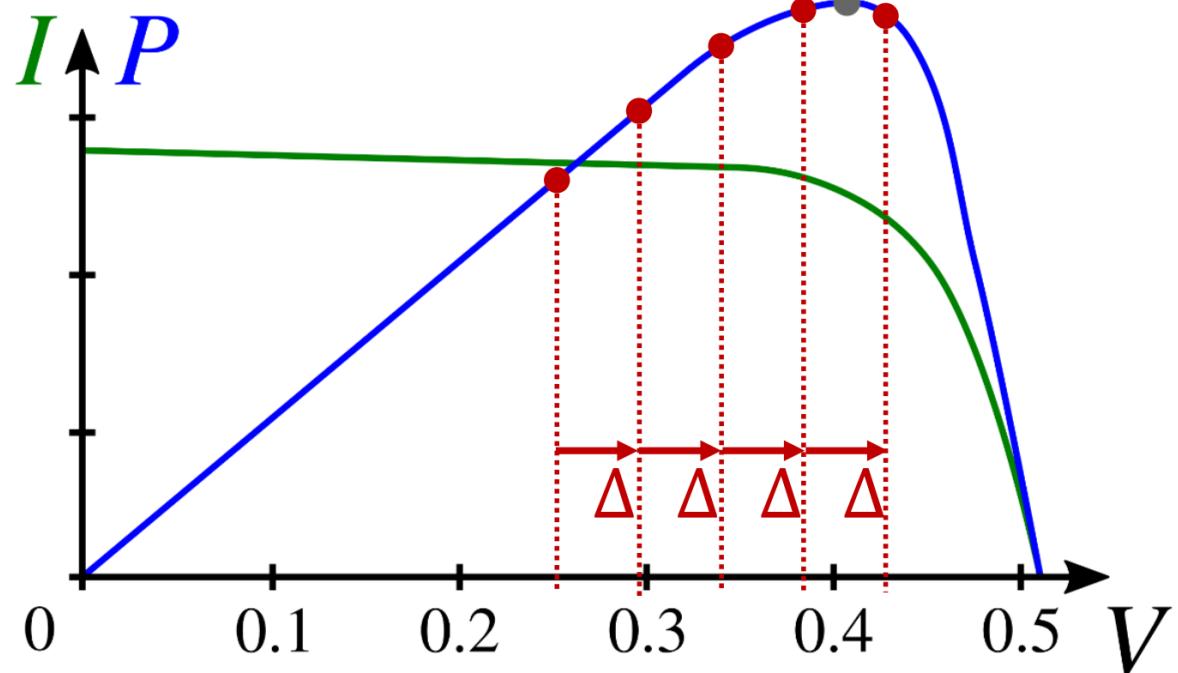


- red:** current for different light intensities
- blue:** power for different light intensities
- grey:** maximal power
- tracking:** determine optimal impedance seen by the solar panel

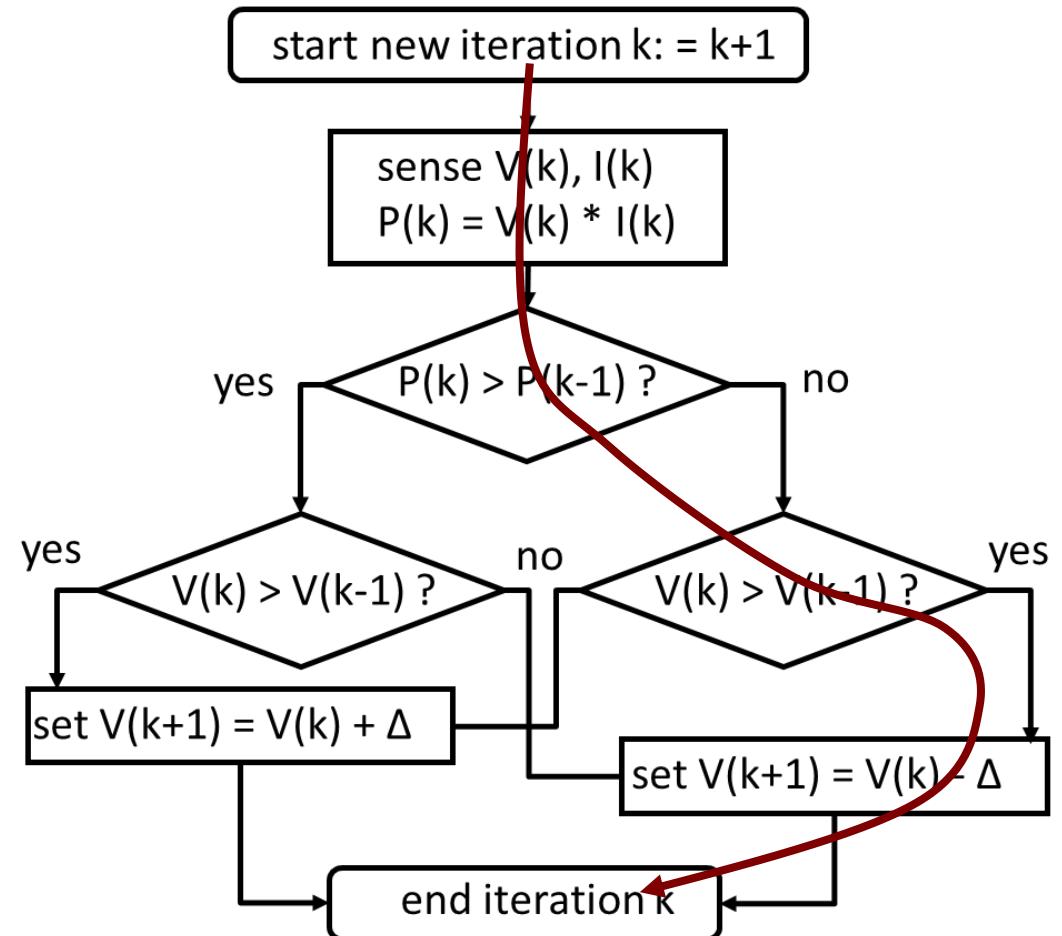
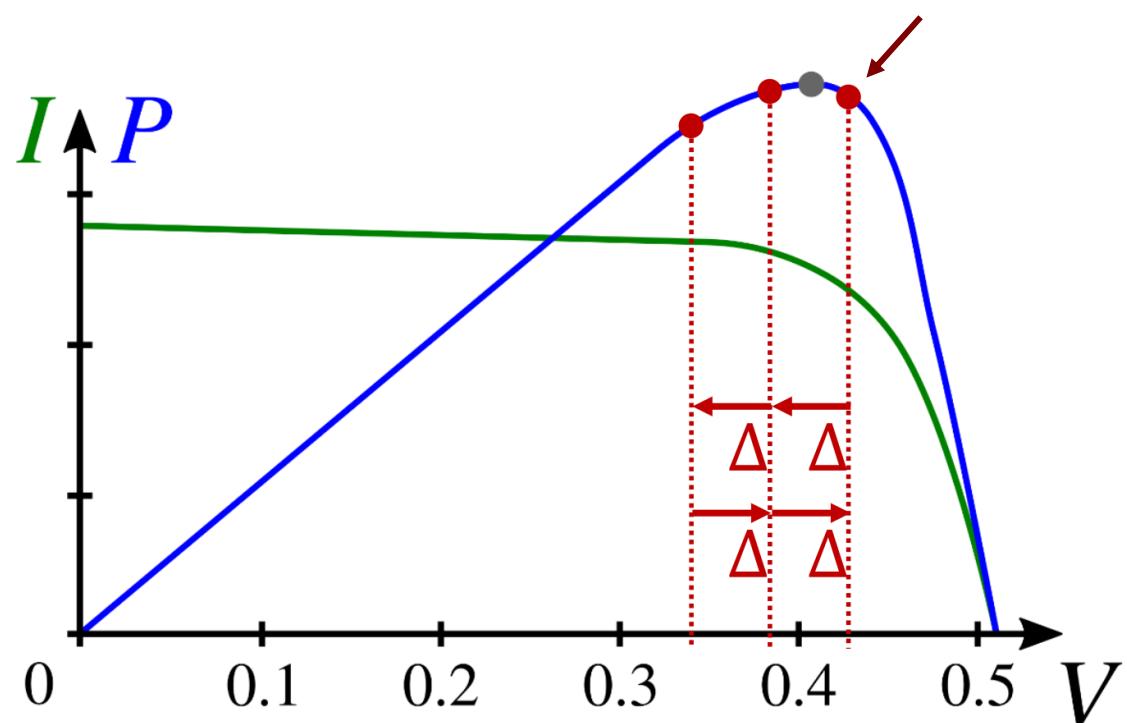
simple tracking algorithm (assume constant illumination) :



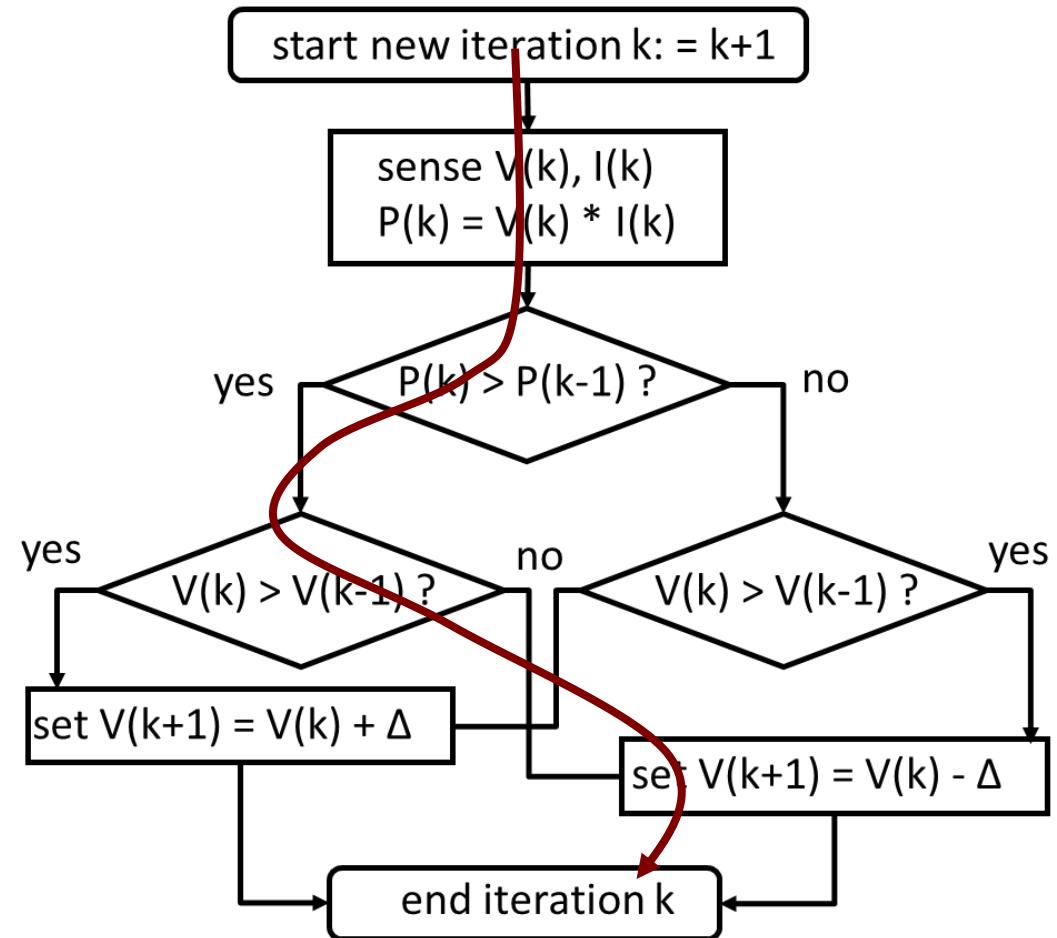
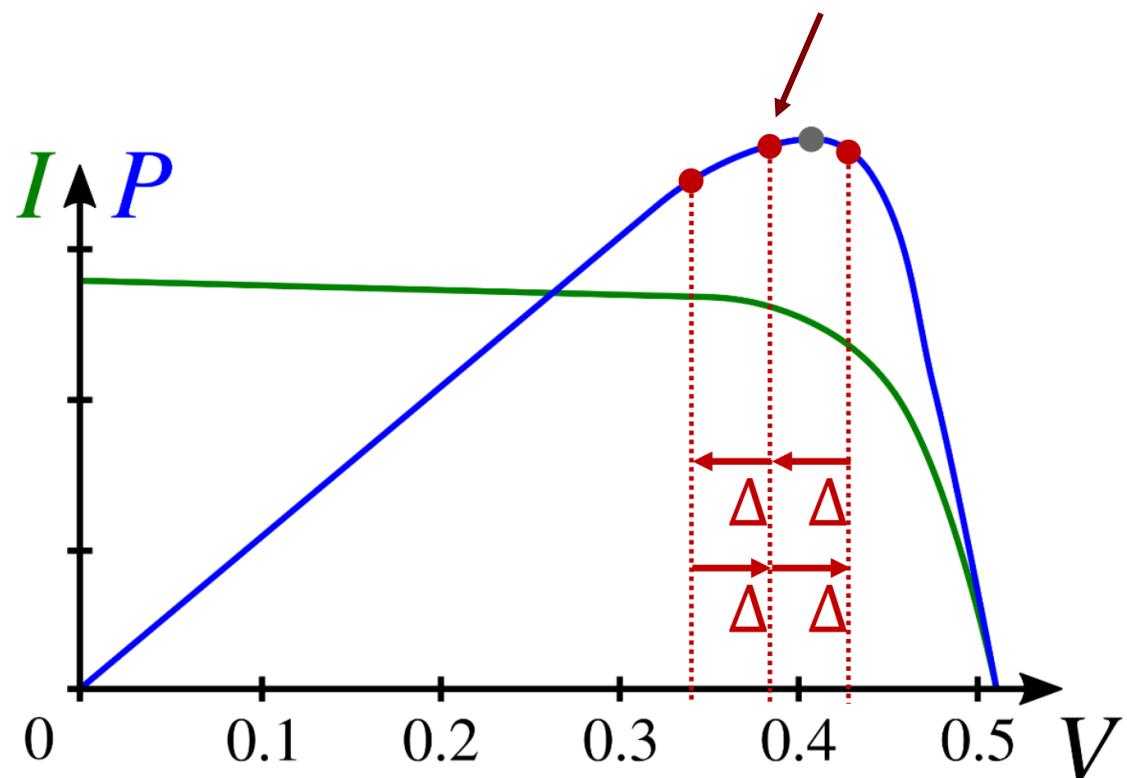
Maximal Power Point Tracking



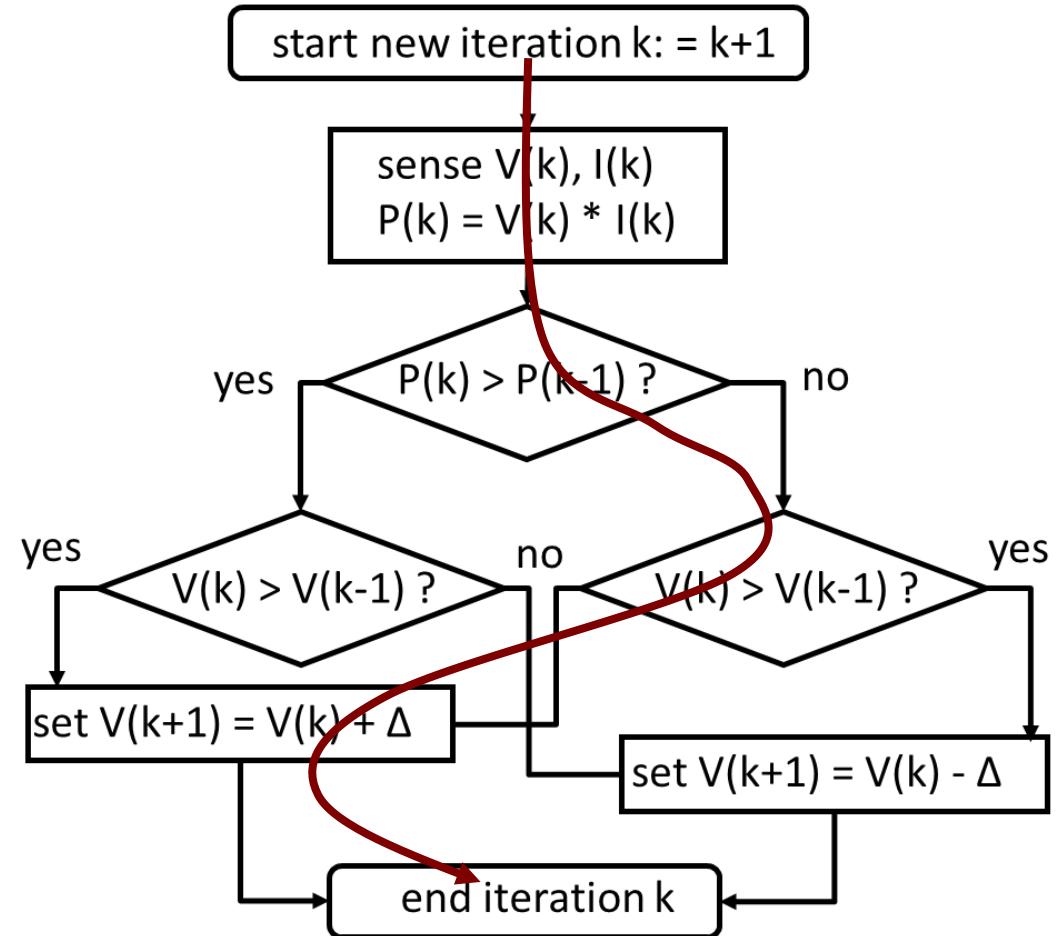
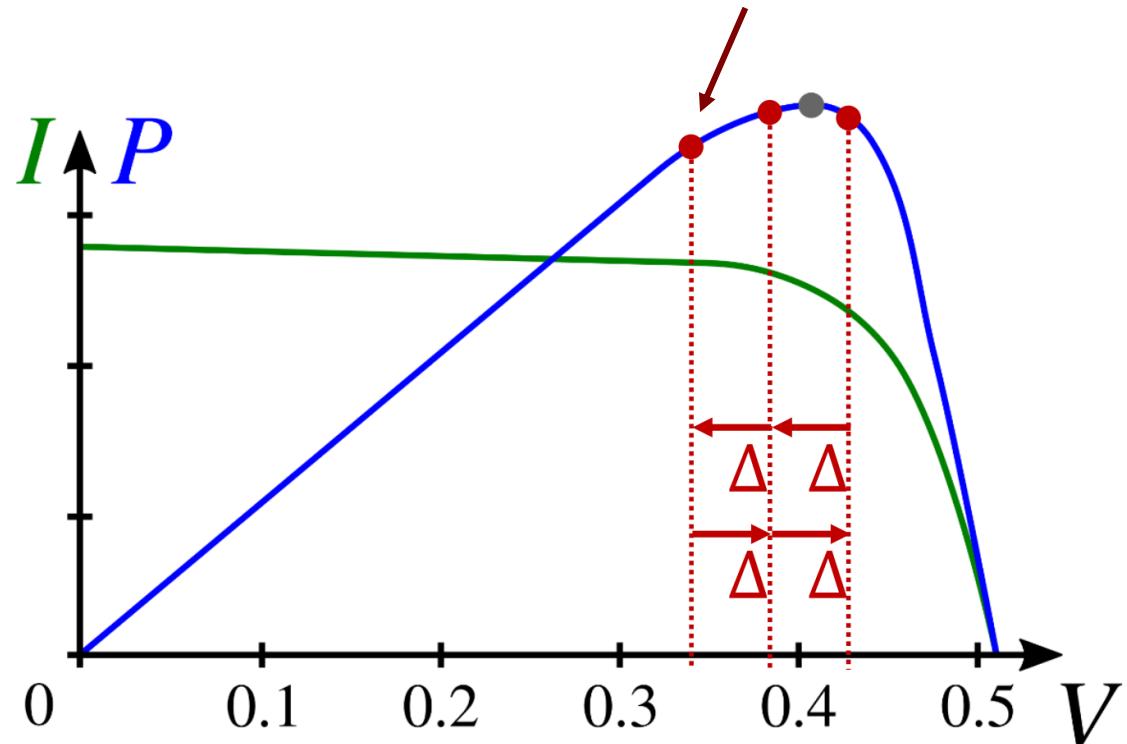
Maximal Power Point Tracking



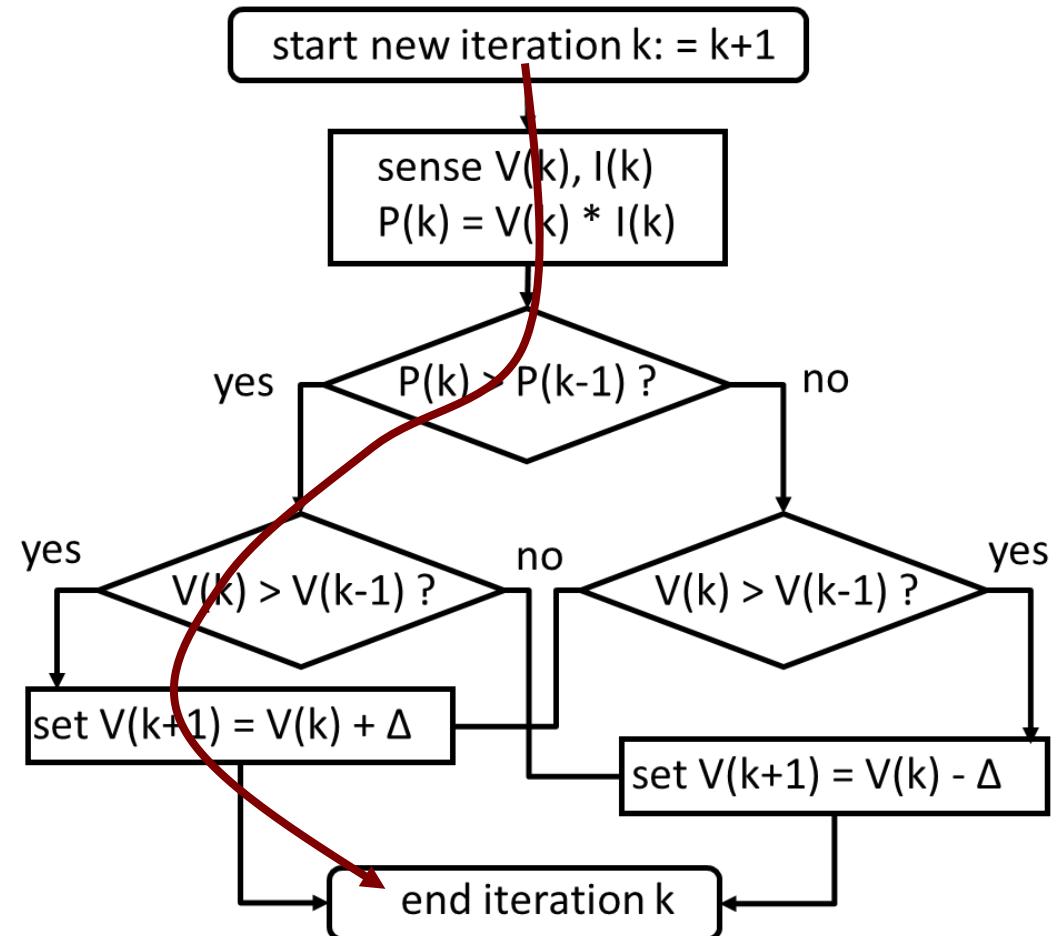
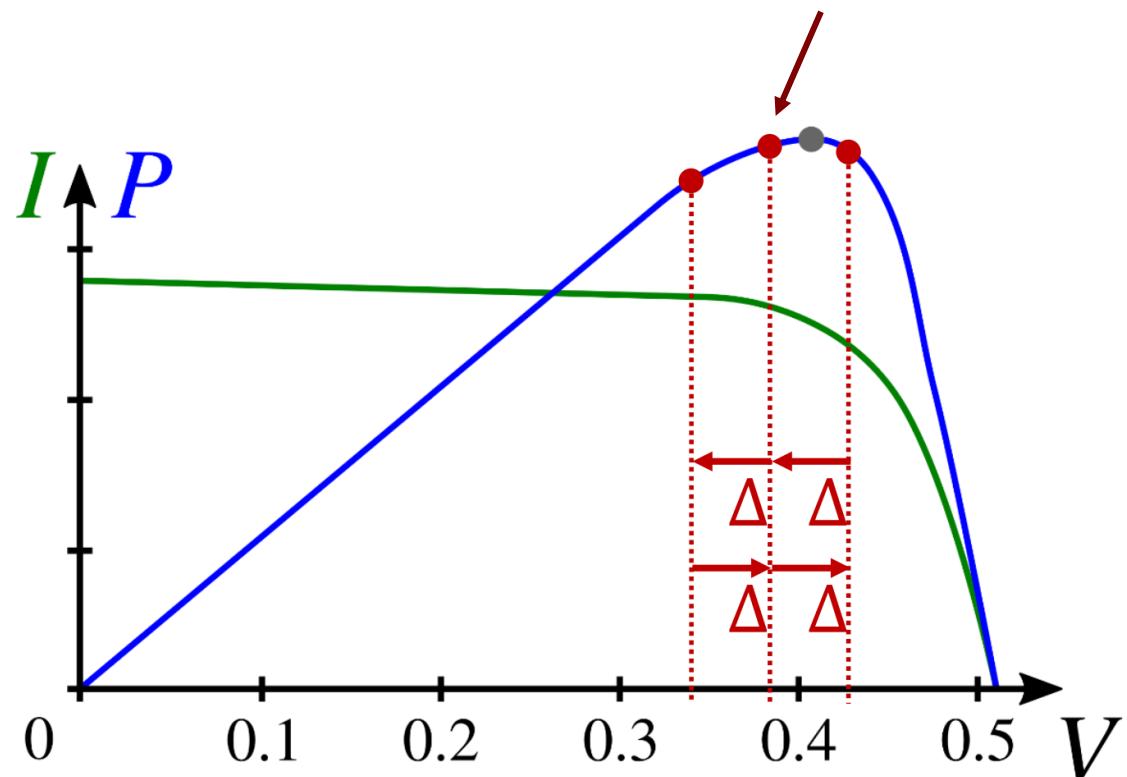
Maximal Power Point Tracking



Maximal Power Point Tracking



Maximal Power Point Tracking



Introduction to Embedded Systems

9. Power and Energy

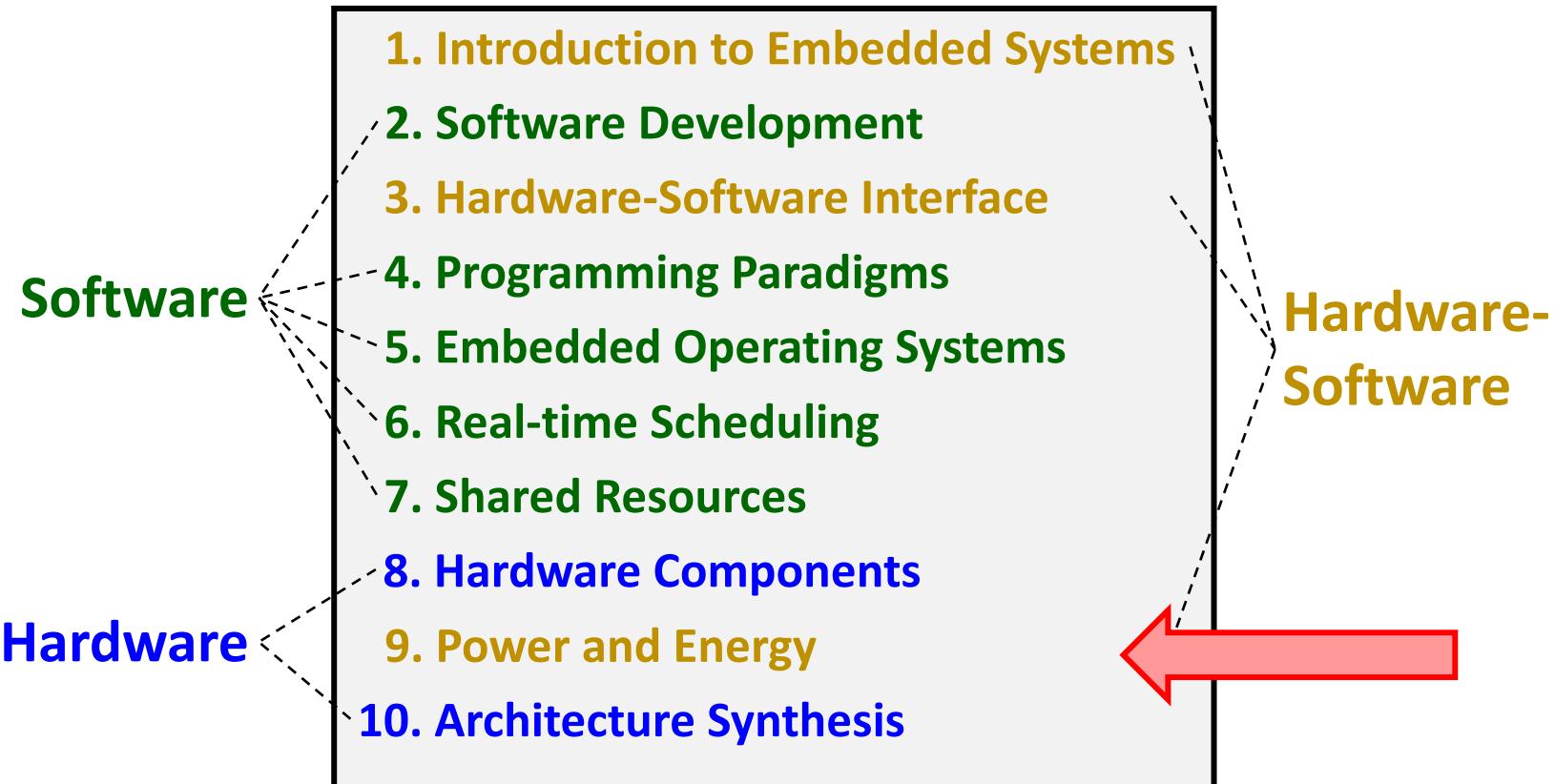
Prof. Dr. Marco Zimmerling



Schedule

- Today (January 17):
 - Exercise 6
- Next week (January 24):
 - *Lecture only on Zoom*
 - Exercise 7
- In two weeks (January 31):
 - Lecture in presence
 - Exercise 8
 - *Mock-up exam questions released on ILIAS*
- In three weeks (February 7):
 - No lecture
 - *Mock-up exam discussed in exercise sessions*
 - Sample solutions to mock-up exam will be available on ILIAS

Where we are ...

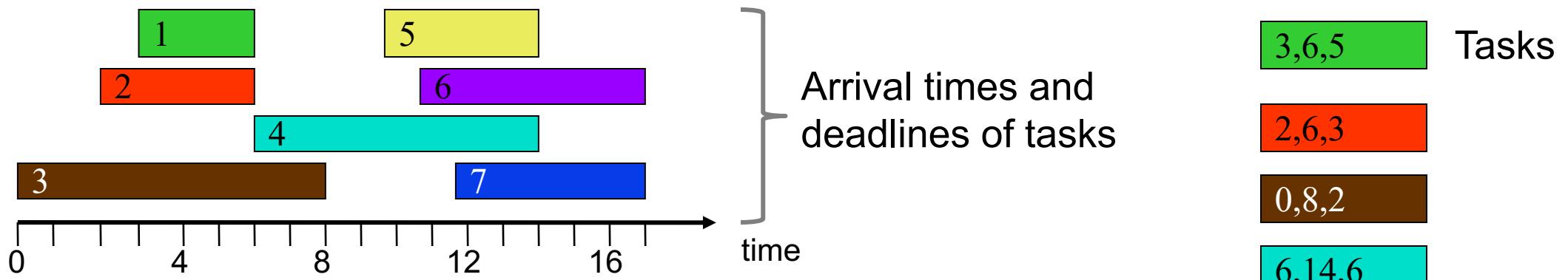


DVFS: Real-Time Offline Scheduling on One Processor

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



- 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 $z' - z$

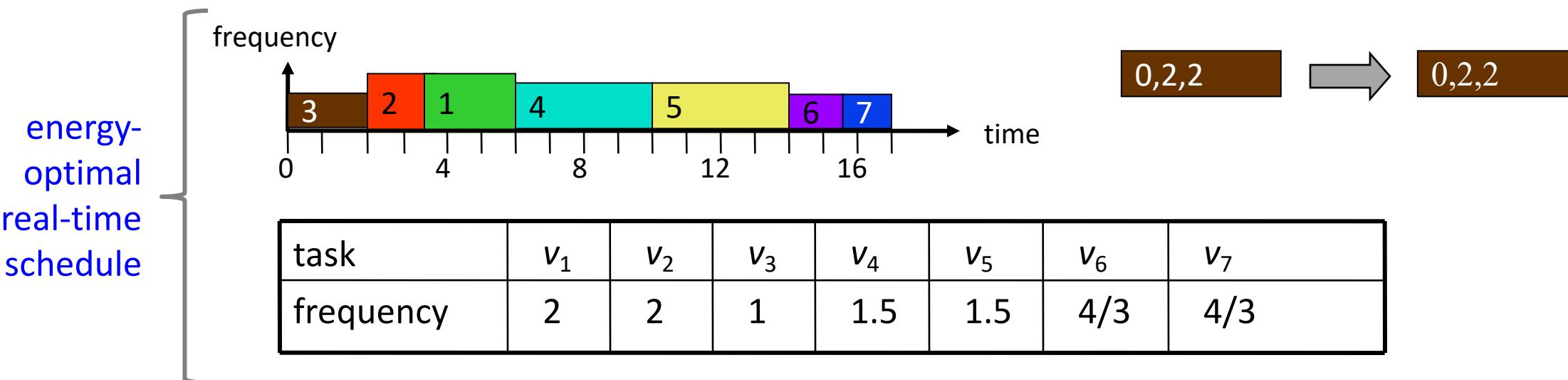
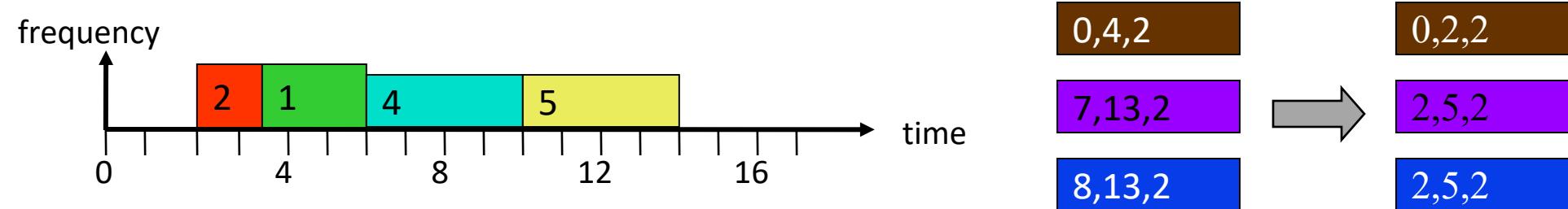
$$V'([z, z']) = \{v_i \in V : z \leq a_i < d_i \leq z'\}$$

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

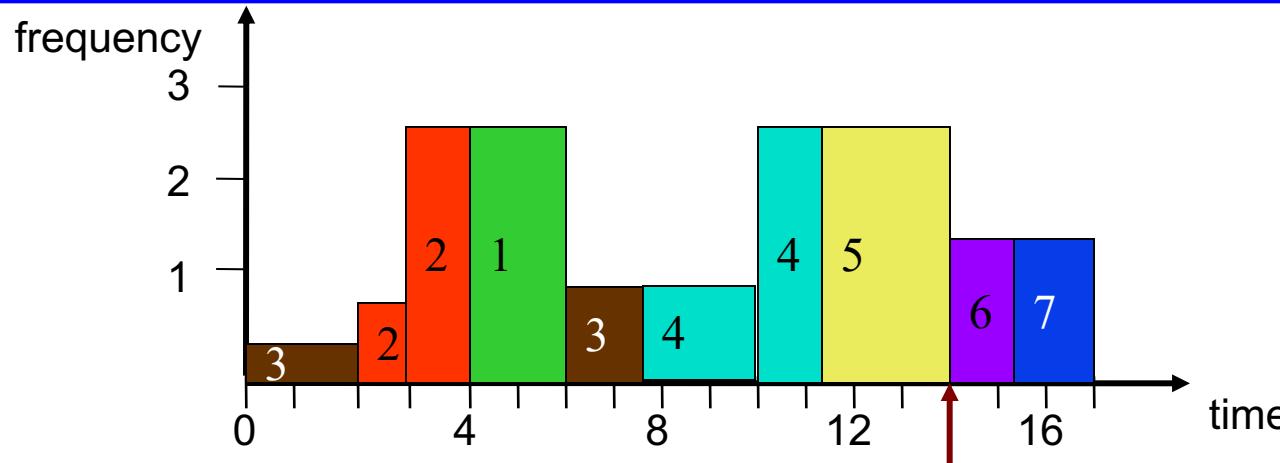
YDS Optimal DVFS Algorithm for Offline Scheduling

Step 3: Run the algorithm for the revised input again

Step 4: Put pieces together



YDS Optimal DVFS Algorithm for Online Scheduling



Continuously update to the best schedule for all arrived tasks:

Time 0: task v₃ is executed at 2/8

Time 2: task v₂ arrives

- $G([2,6]) = 3/4$, $G([2,8]) = 4.5/6 = 3/4 \Rightarrow$ execute v₈, v₂ at 3/4

Time 3: task v₁ arrives

- $G([3,6]) = (5+3-3/4)/3 = 29/12$, $G([3,8]) < G([3,6]) \Rightarrow$ execute v₂ and v₁ at 29/12

Time 6: task v₄ arrives

- $G([6,8]) = 1.5/2$, $G([6,14]) = 7.5/8 \Rightarrow$ execute v₃ and v₄ at 15/16

Time 10: task v₅ arrives

- $G([10,14]) = 39/16 \Rightarrow$ execute v₄ and v₅ at 39/16

Time 11 and Time 12

- The arrival of v₆ and v₇ does not change the critical interval

Time 14:

- $G([14,17]) = 4/3 \Rightarrow$ execute v₆ and v₇ at 4/3

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

Techniques to Reduce Static Power

Dynamic Power Management (DPM)

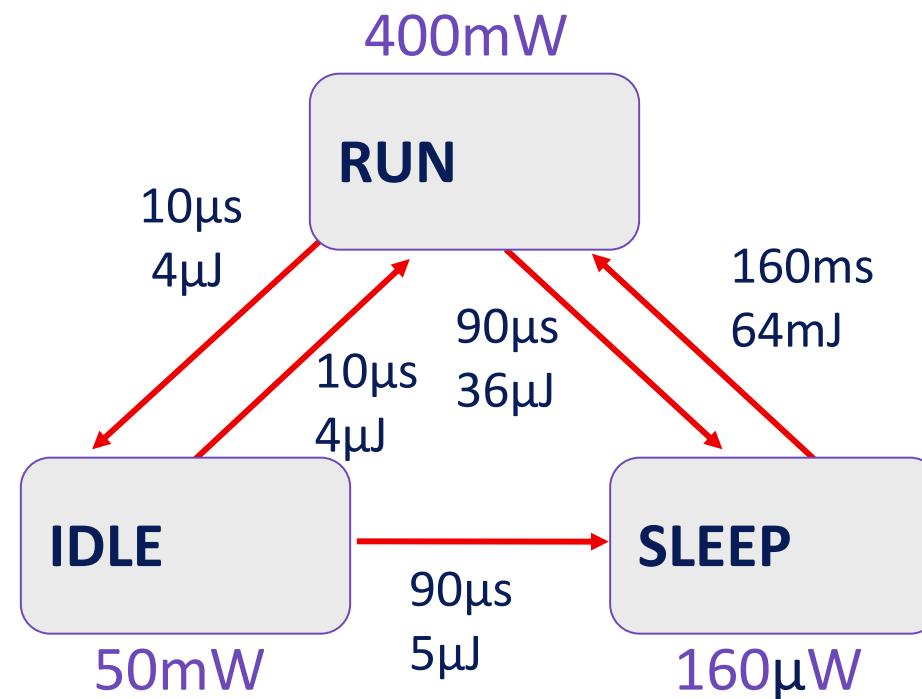
- Dynamic power management (DPM) tries to assign optimal power saving states during program execution
- DPM requires hardware and software support

Example: StrongARM SA1100

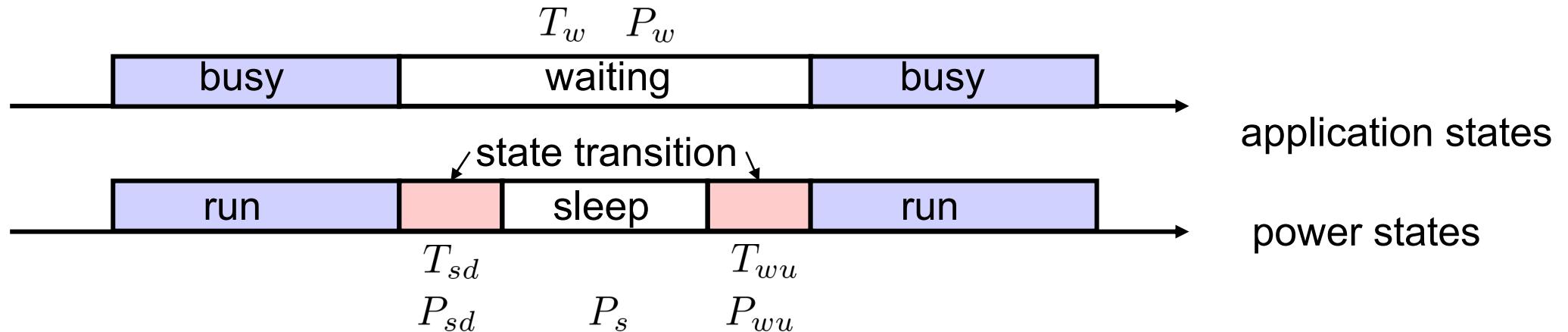
RUN: operational

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

SLEEP: Shutdown of on-chip activity



Break-Even Time



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

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

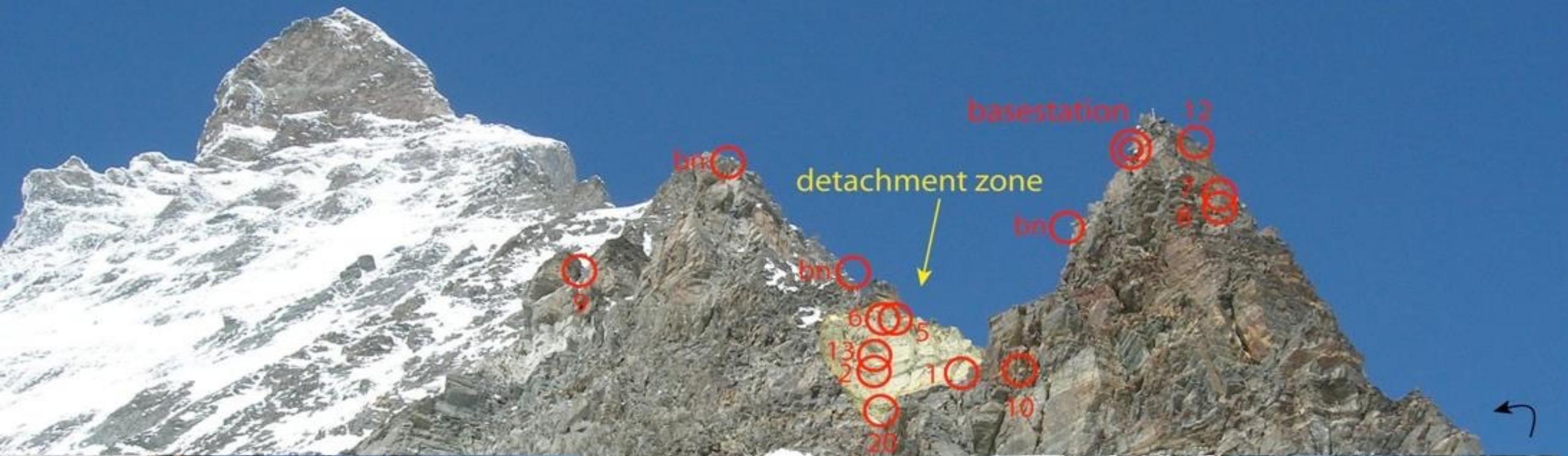
Break-even time: Limit for T_w such that $E_2 \leq E_1$

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

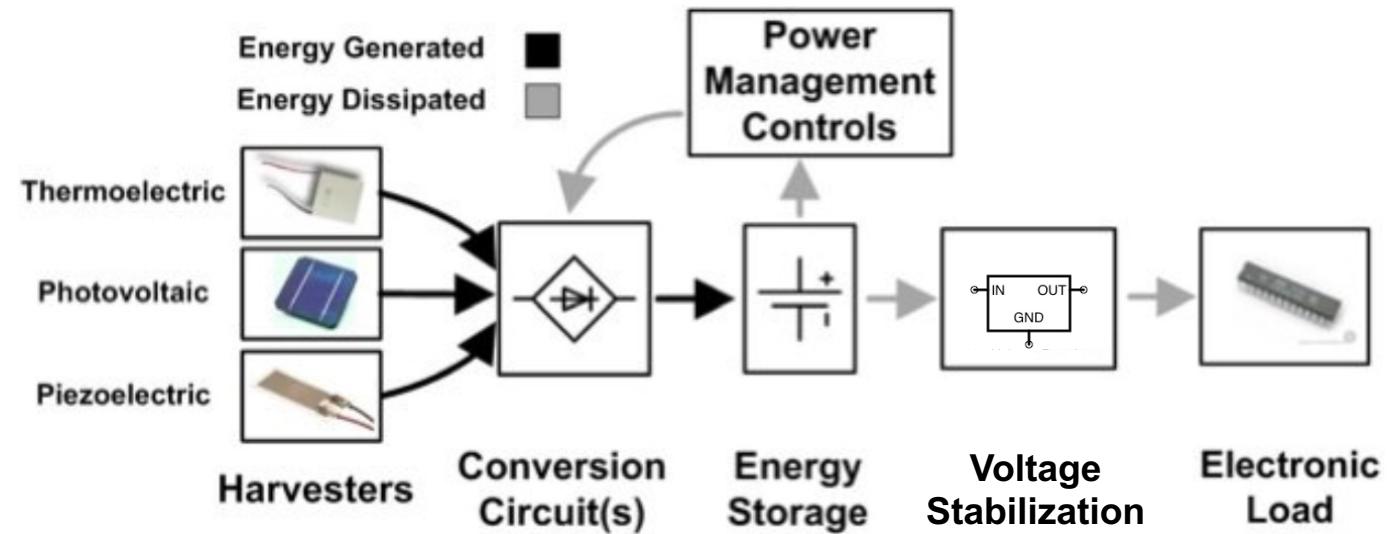
Time constraint: $T_w \geq T_{sd} + T_{wu}$

break-even
time

Battery-Operated Systems and Energy Harvesting



Typical Power Circuitry – Power Point Tracking

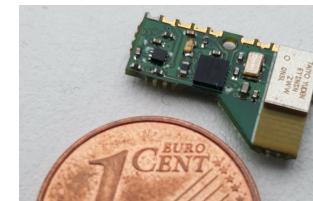


power point tracking / impedance matching; conversion to voltage of energy storage

rechargeable battery, supercapacitor, or multi-layer ceramic capacitor (MLCC)

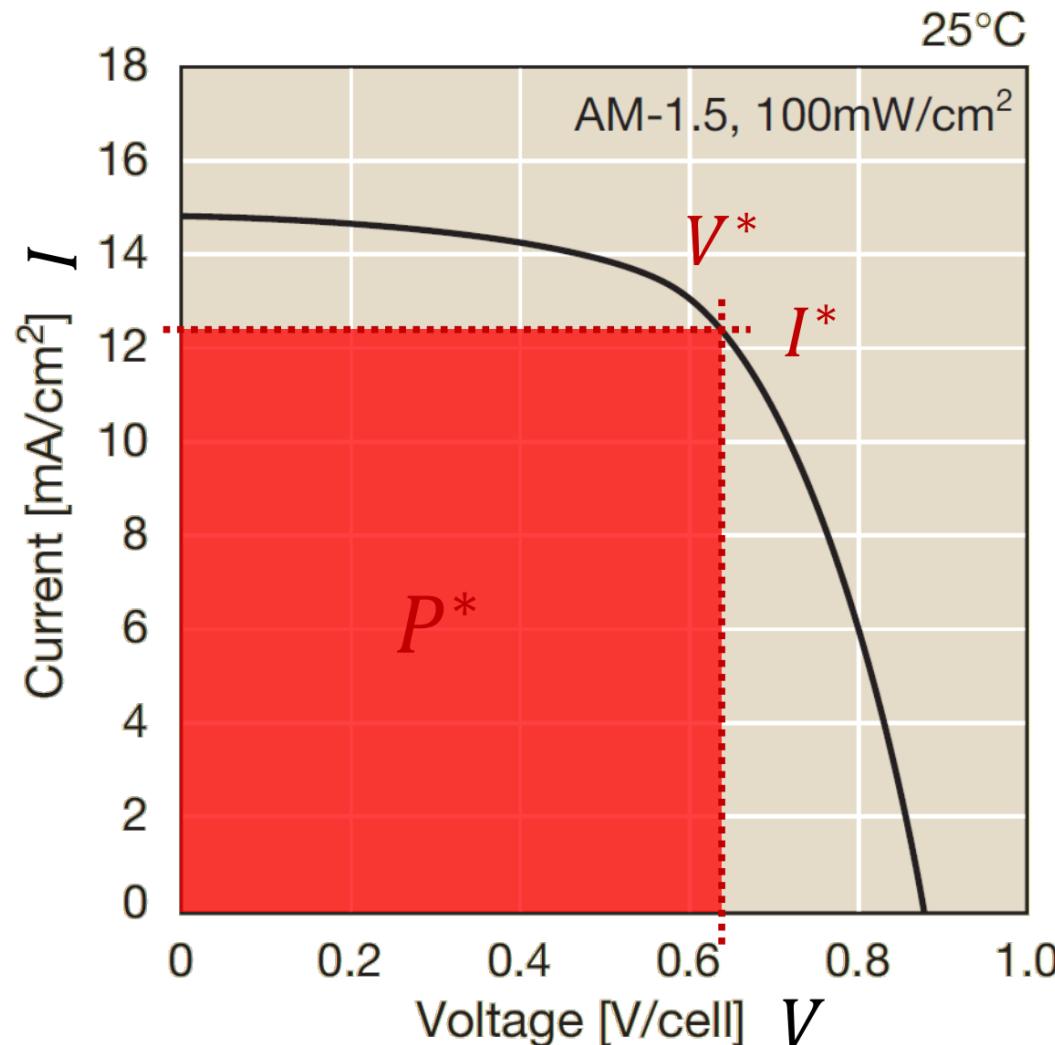


MLCC



battery-free
IoT device

Solar Panel Characteristics

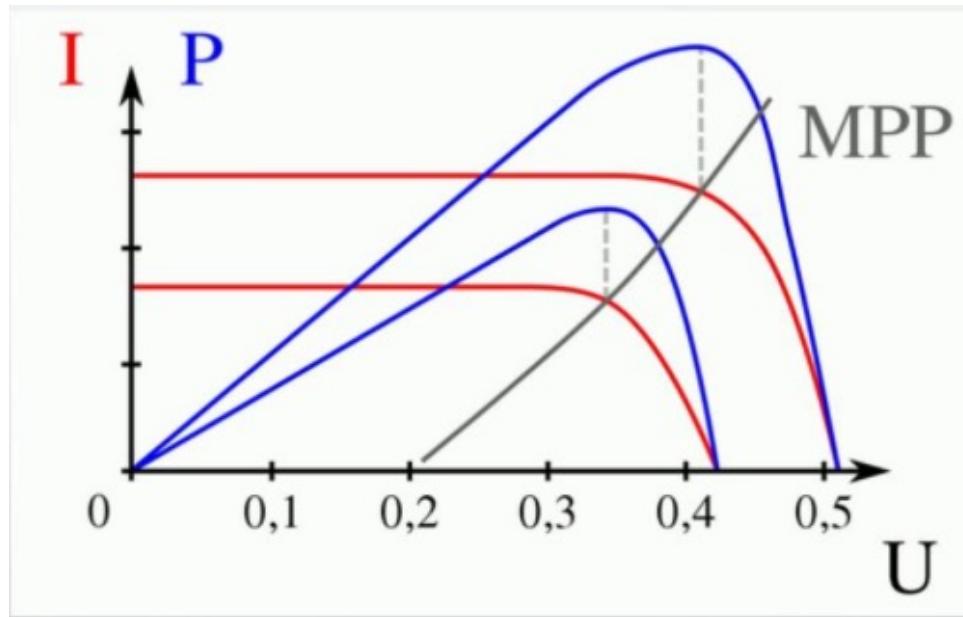


- Variable output power
 - Illuminance level
 - Electrical operation point
 - (Temperature, age, ...)
- I-V-Characteristics
 - Non-linear
 - Dependent on ambient
- Maximum Power Point Tracking
 - Dynamic algorithm to find P^*

Diagram: Amorton Amorphous Silicon Solar Cells Datasheet, © Panasonic

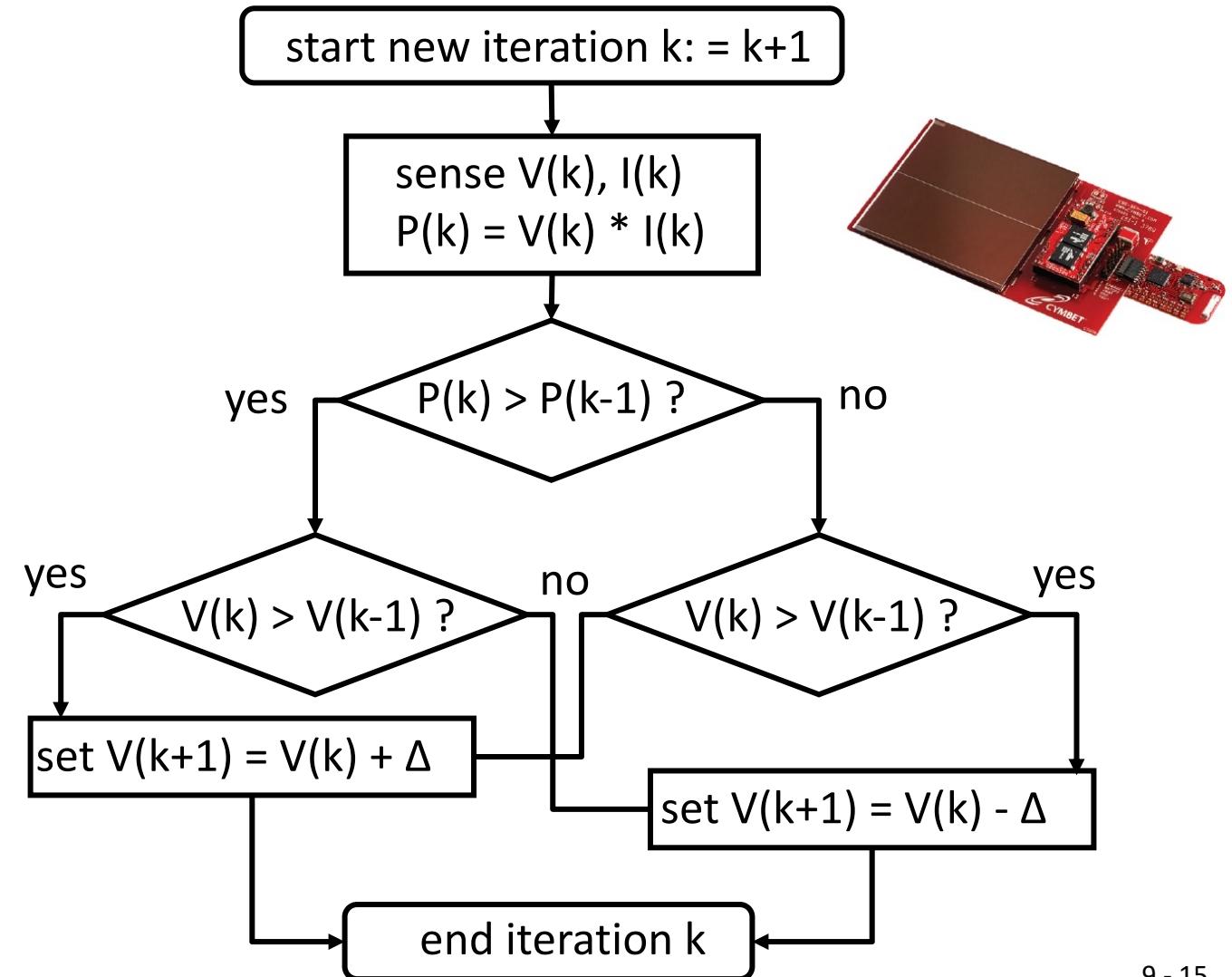
Typical Power Circuitry – Maximum Power Point Tracking

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) :

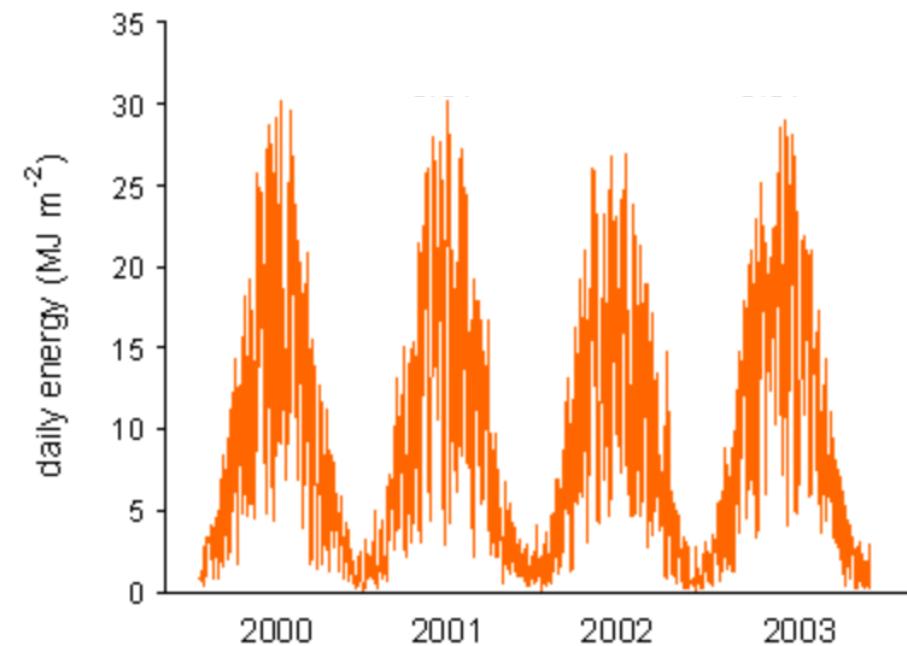
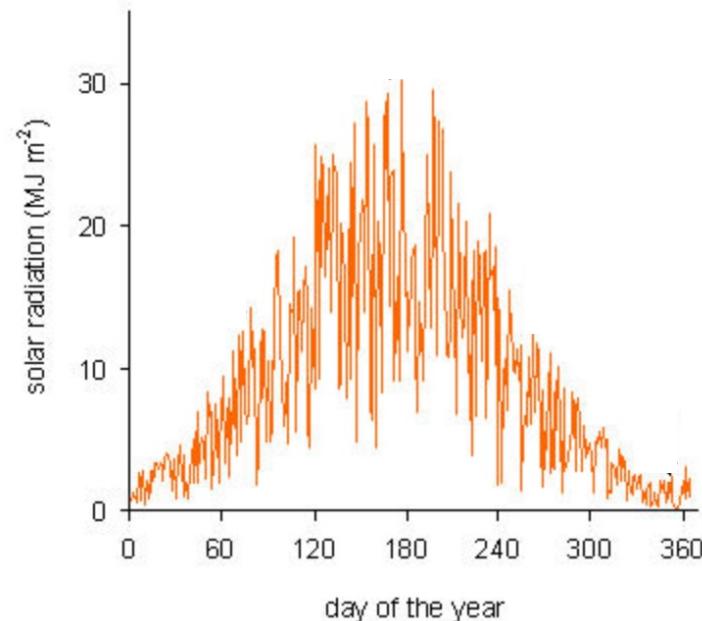


Typical Challenge in (Solar) Harvesting Systems

Challenges:

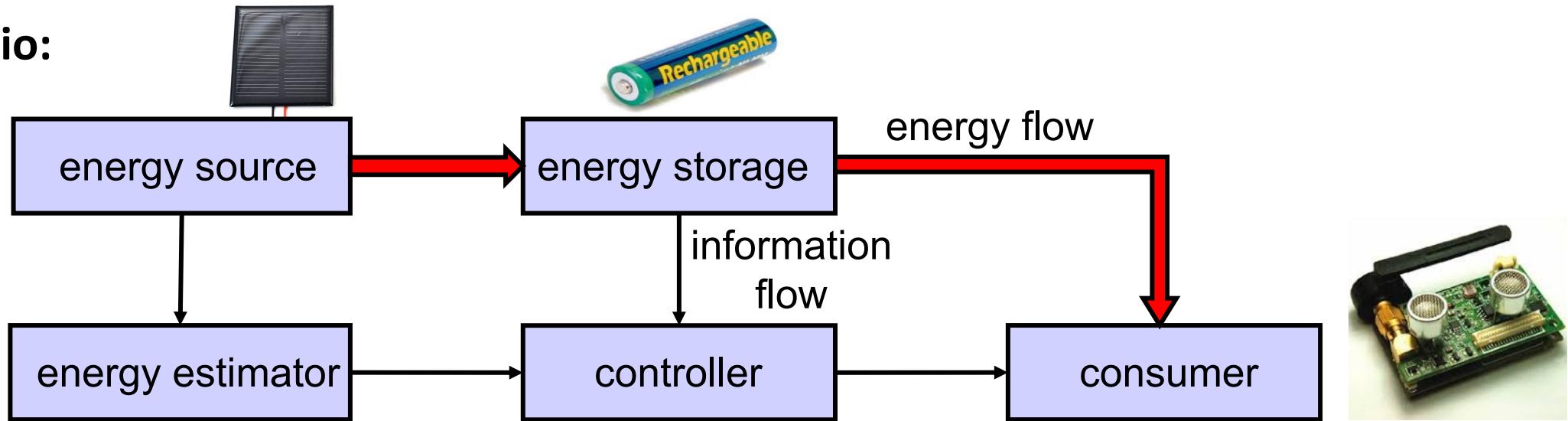
- What is the optimal maximum capacity of the battery?
- What is the optimal area of the solar cell?
- How can we control the application such that a continuous system operation is possible, even under a varying input energy (summer, winter, clouds)?

Example of a solar energy trace:



Example: Application Control

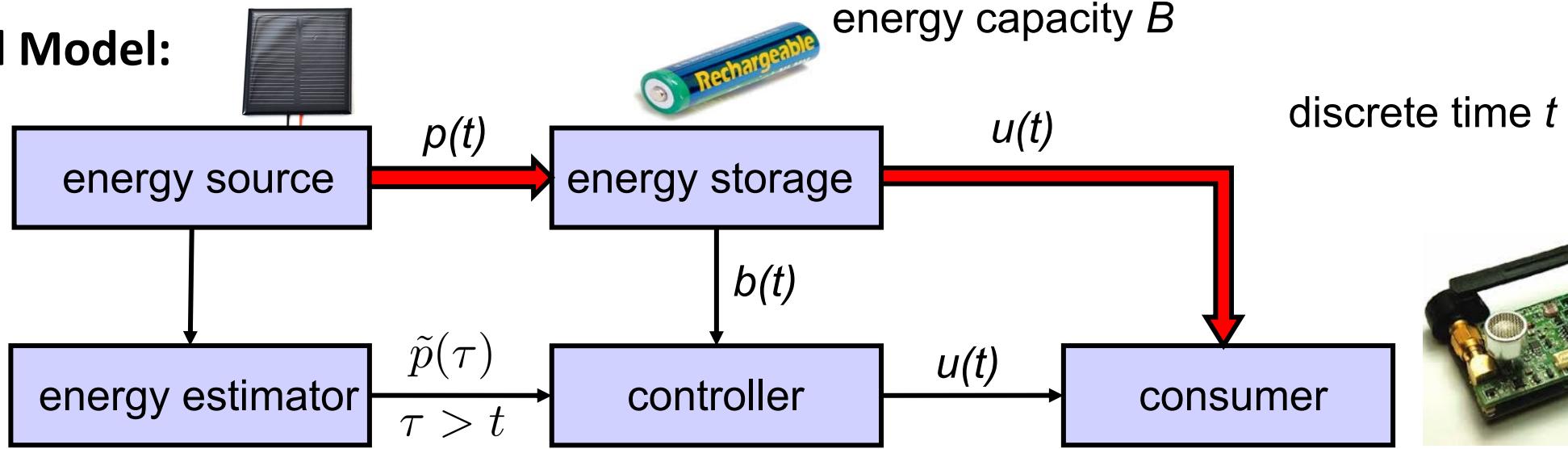
Scenario:



- 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:

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

https://en.wikipedia.org/wiki/Diminishing_returns
not worth measuring sth all the time if it only changes slowly

μ 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:

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

find minimal possible optimal usage function larger than any other minimal use function

- 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., $b^*(T) \geq b^*(0)$.
- 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 $\tau \in (0, T)$:

$$\begin{aligned} u^*(\tau - 1) < u^*(\tau) &\implies b^*(\tau) = 0 && \text{empty battery} \\ u^*(\tau - 1) > u^*(\tau) &\implies b^*(\tau) = B && \text{full battery} \end{aligned}$$

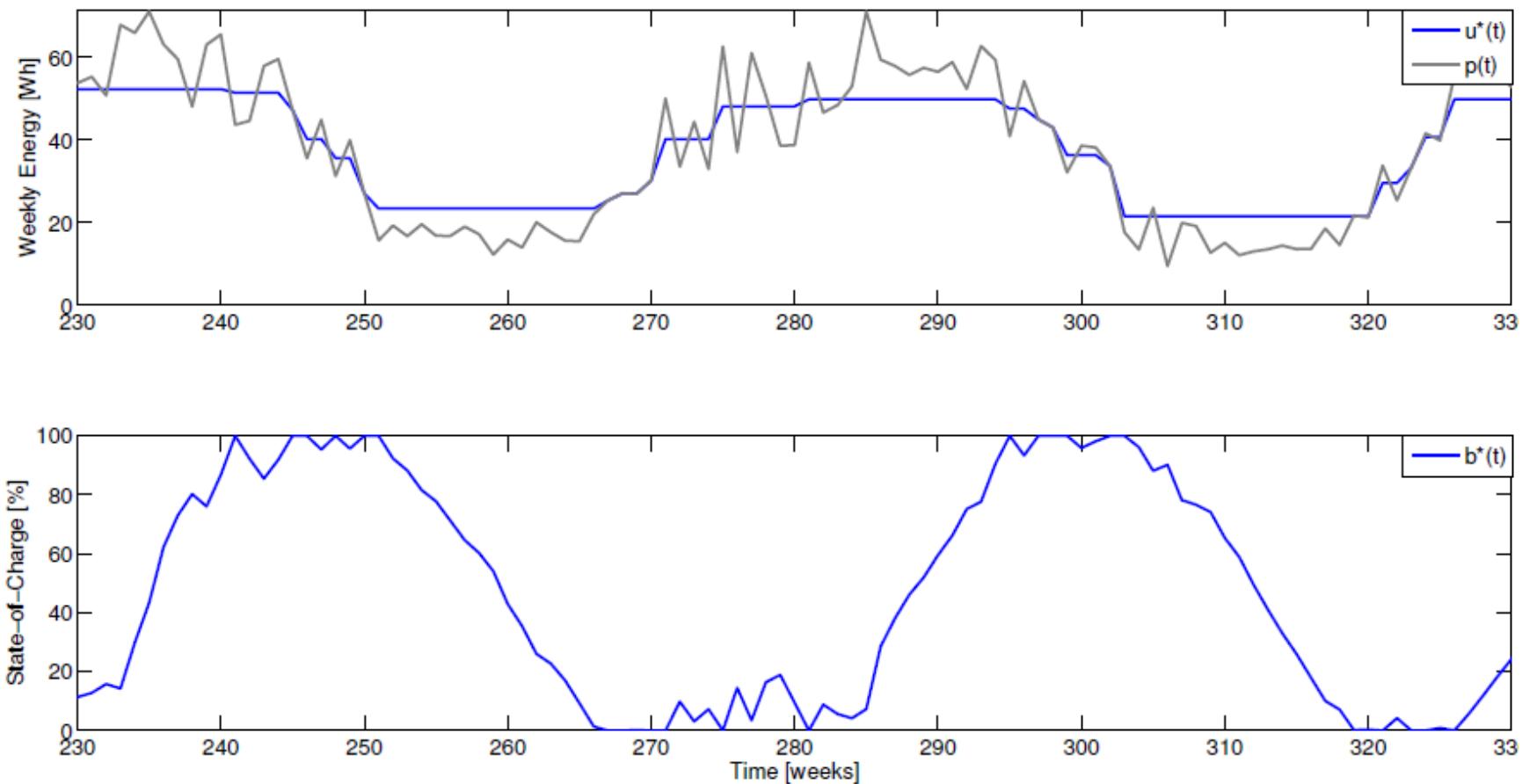
Sketch of a proof: First, let us show that a consequence of the above theorem is true (just reverting the relations):

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

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 $b^*(t)$.

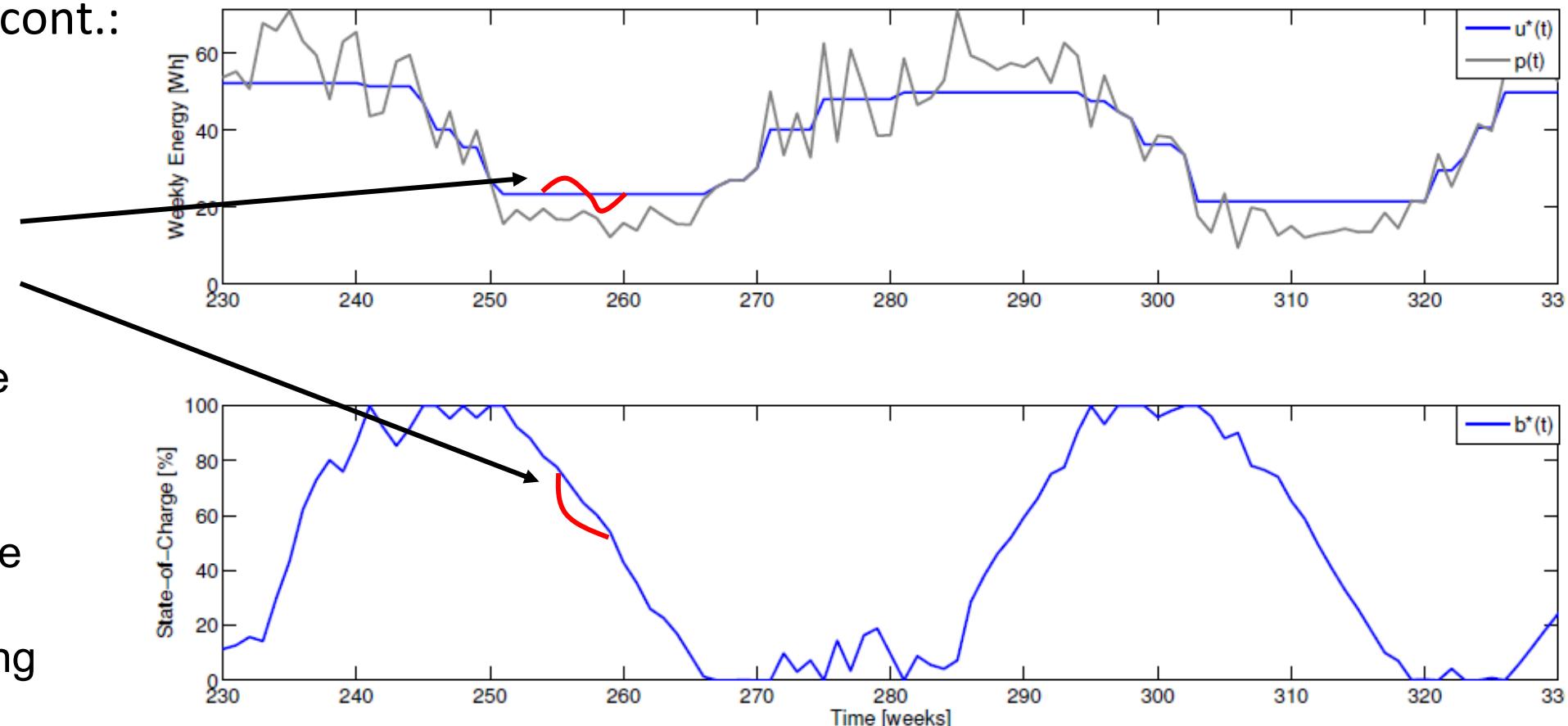
Application Control

- Proof sketch cont.:

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



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



(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

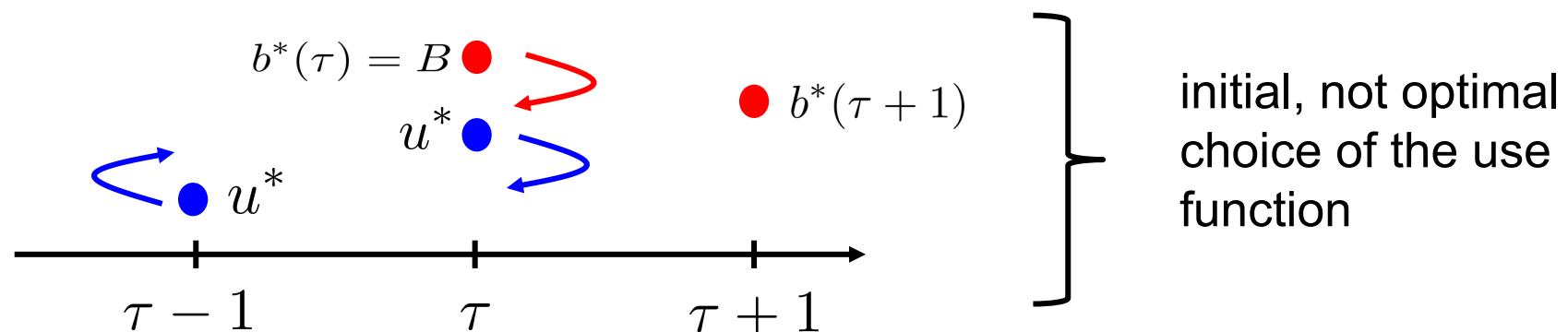
- Proof sketch cont.: Now we show that for all $\tau \in (t, T)$

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

or equivalently

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

We already have shown this for $0 < b^*(\tau) < B$. Therefore, we only need to show that $b^*(\tau) = B \implies u^*(\tau - 1) \geq u^*(\tau)$. Suppose now that we have $u^*(\tau - 1) < u^*(\tau)$ if the battery is full at τ . Then we can increase the use at time $\tau - 1$ and decrease it at time τ 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

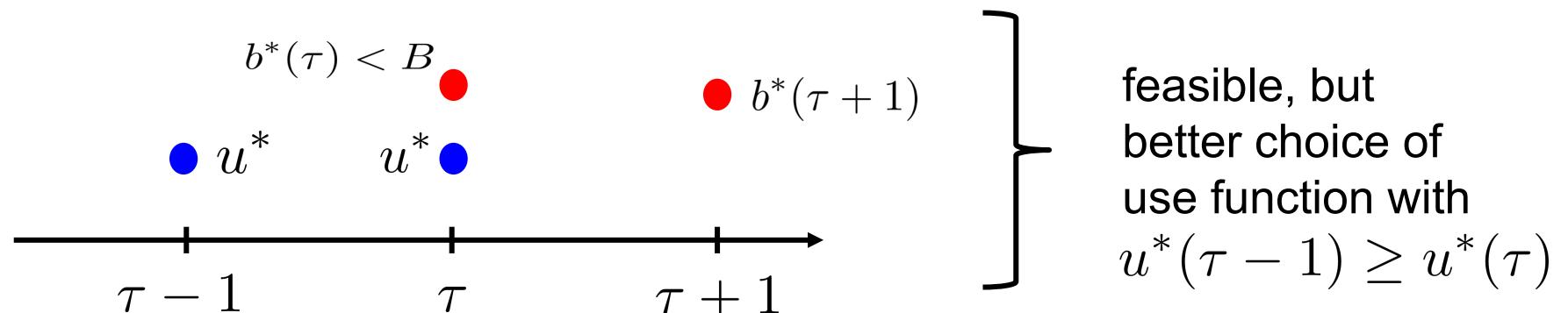
- Proof sketch cont.: Now we show that for all $\tau \in (t, T)$

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

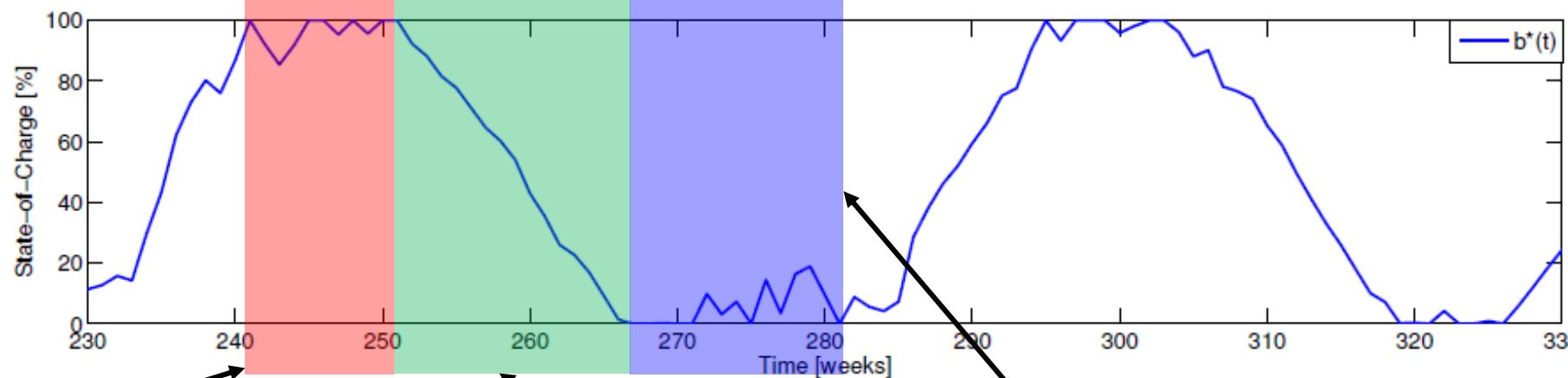
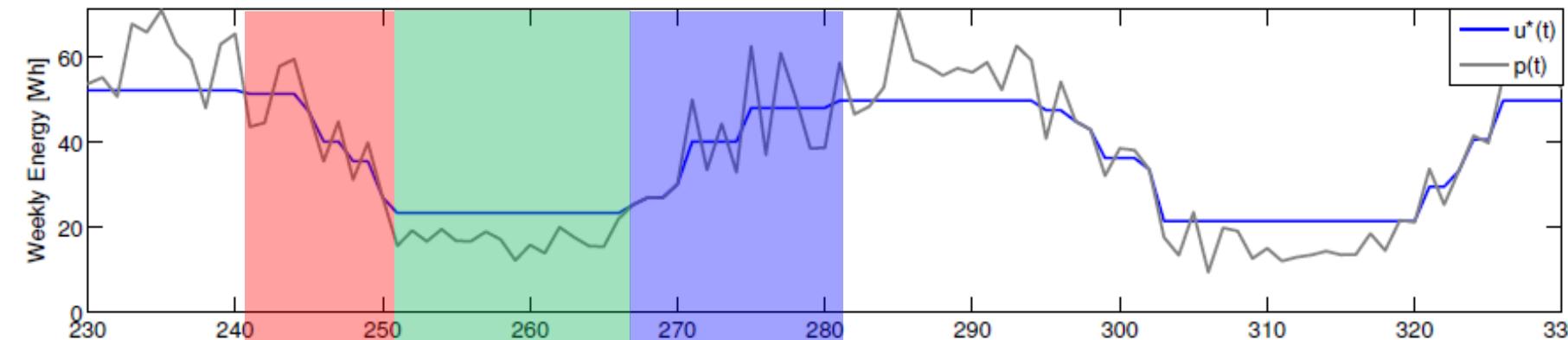
or equivalently

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

We already have shown this for $0 < b^*(\tau) < B$. Therefore, we only need to show that $b^*(\tau) = B \implies u^*(\tau - 1) \geq u^*(\tau)$. Suppose now that we have $u^*(\tau - 1) < u^*(\tau)$ if the battery is full at τ . Then we can increase the use at time $\tau - 1$ and decrease it at time τ 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



if battery is full, the use function decreases or remains constant

if battery is neither full nor empty, the use function is constant

if battery is empty, the use function increases or remains constant

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

$$U(0, T) = \sum_{0 \leq \tau < T} u(\tau)$$

Then the linear program has the form:

[Concave functions μ could be piecewise linearly approximated.
This is not shown here.]

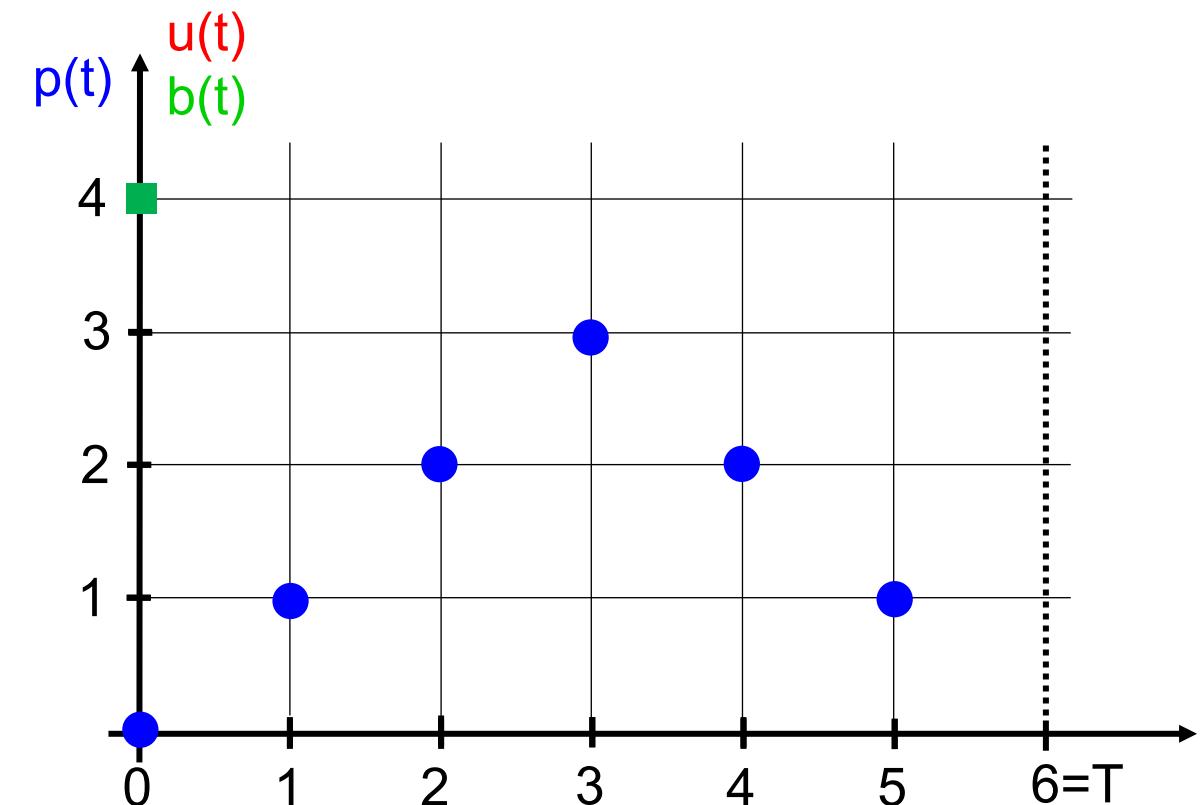
$$\begin{aligned} & \text{maximize} \quad \sum_{0 \leq \tau < T} u(\tau) \\ & \forall \tau \in [0, T] : b(\tau + 1) = b(\tau) - u(\tau) + p(\tau) \\ & \forall \tau \in [0, T] : 0 \leq b(\tau + 1) \leq B \\ & \forall \tau \in [0, T] : u(\tau) \geq 0 \\ & b(T) = b(0) = b_0 \end{aligned}$$

Computing the optimal use function by hand

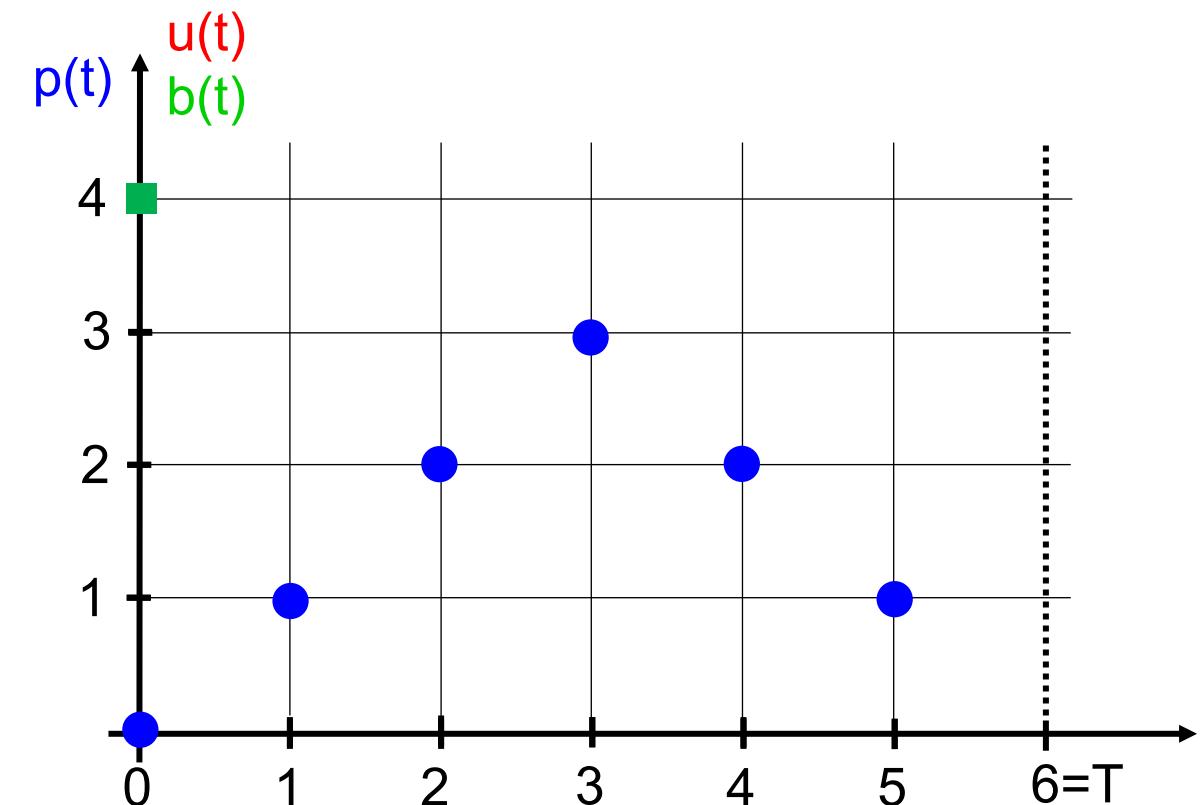
- Case 1:
 - Large battery capacity B
 - Large initial stored energy, for example, $b(0) = 4$
 - What is the optimal use function $u^*(t)$?

- Case 2:
 - Small battery capacity, for example, $B = 2$
 - Small initial stored energy, for example, $b(0) = 1$
 - What is the optimal use function $u^*(t)$?

Case 1: B large, $b(0) = 4$



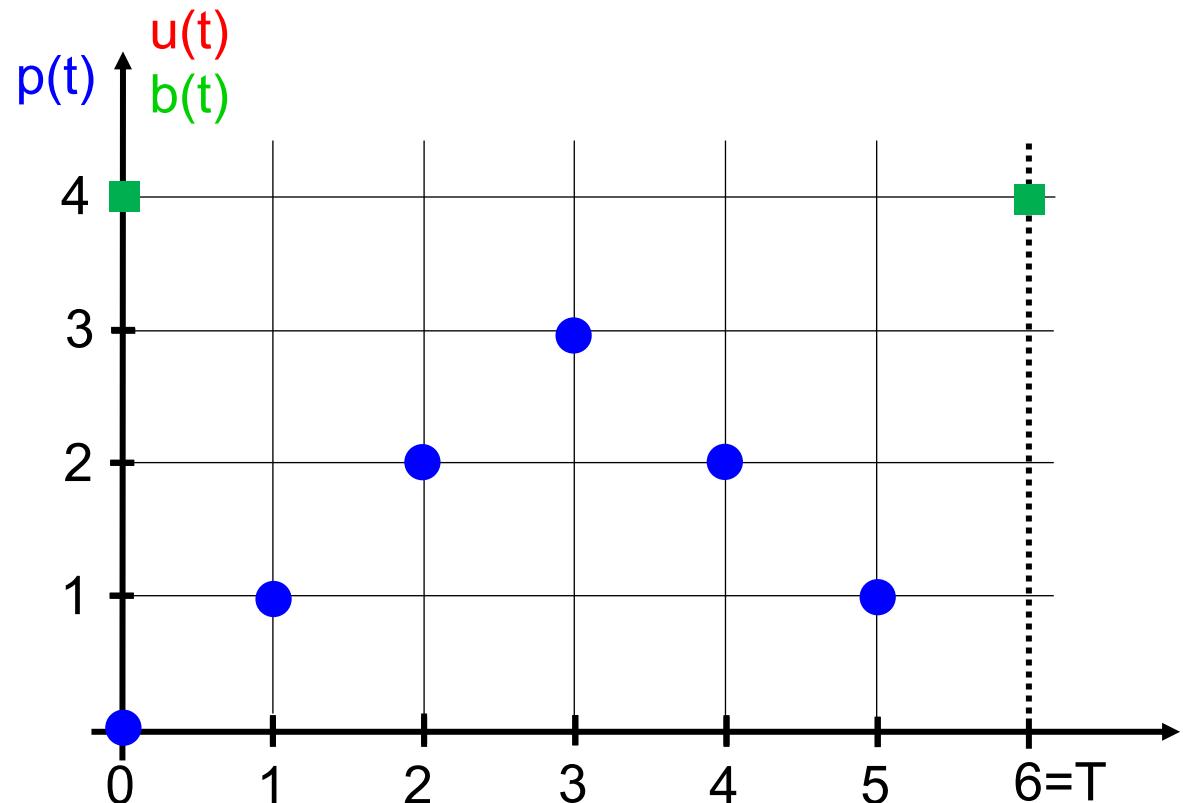
Case 1: B large, $b(0) = 4$



a) Harvested energy in observation window $[0, T]$:

$$\sum_{(t)} p(t) = 1 + 2 + 3 + 2 + 1 = 9$$

Case 1: B large, $b(0) = 4$



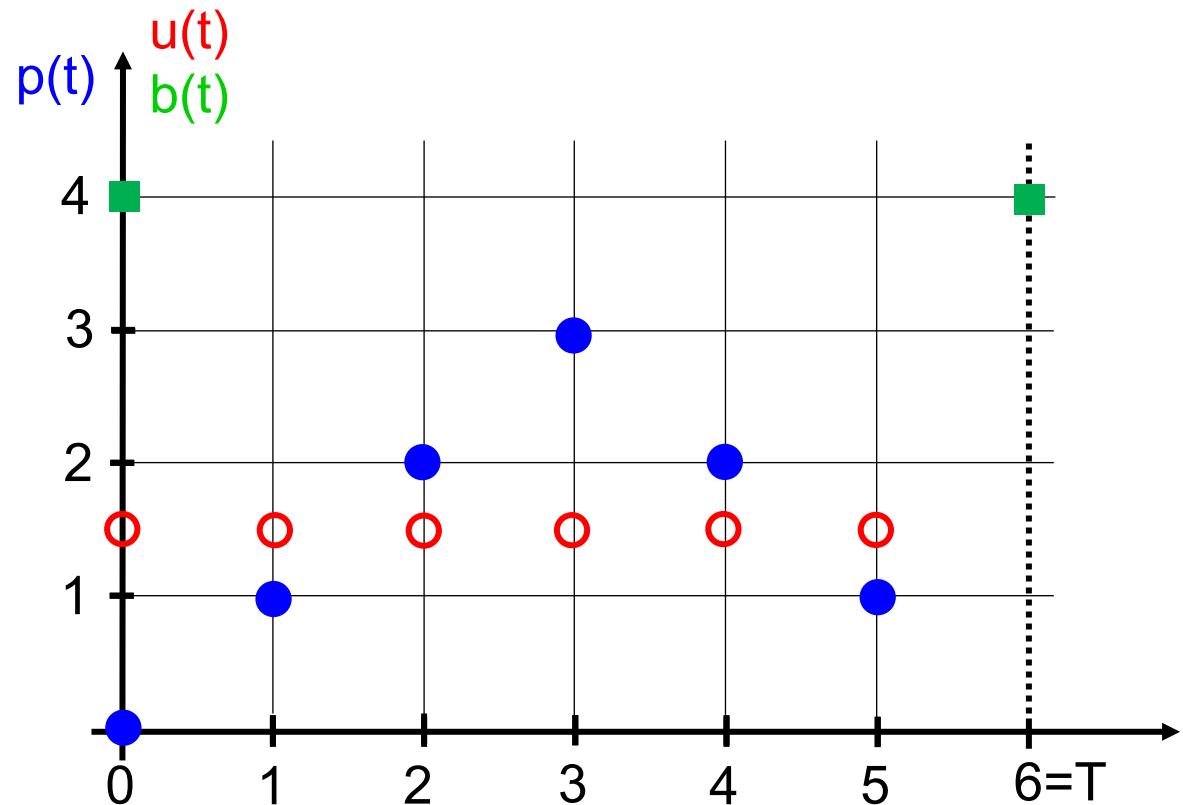
a) Harvested energy in observation window $[0, T]$:

$$\sum_{(t)} p(t) = 1 + 2 + 3 + 2 + 1 = 9$$

b) We require $b(0) \leq b(T)$, that is,

$$\sum_{(t)} u(t) \leq \sum_{(t)} p(t) = 9$$

Case 1: B large, $b(0) = 4$



a) Harvested energy in observation window $[0, T]$:

$$\sum_{(t)} p(t) = 1 + 2 + 3 + 2 + 1 = 9$$

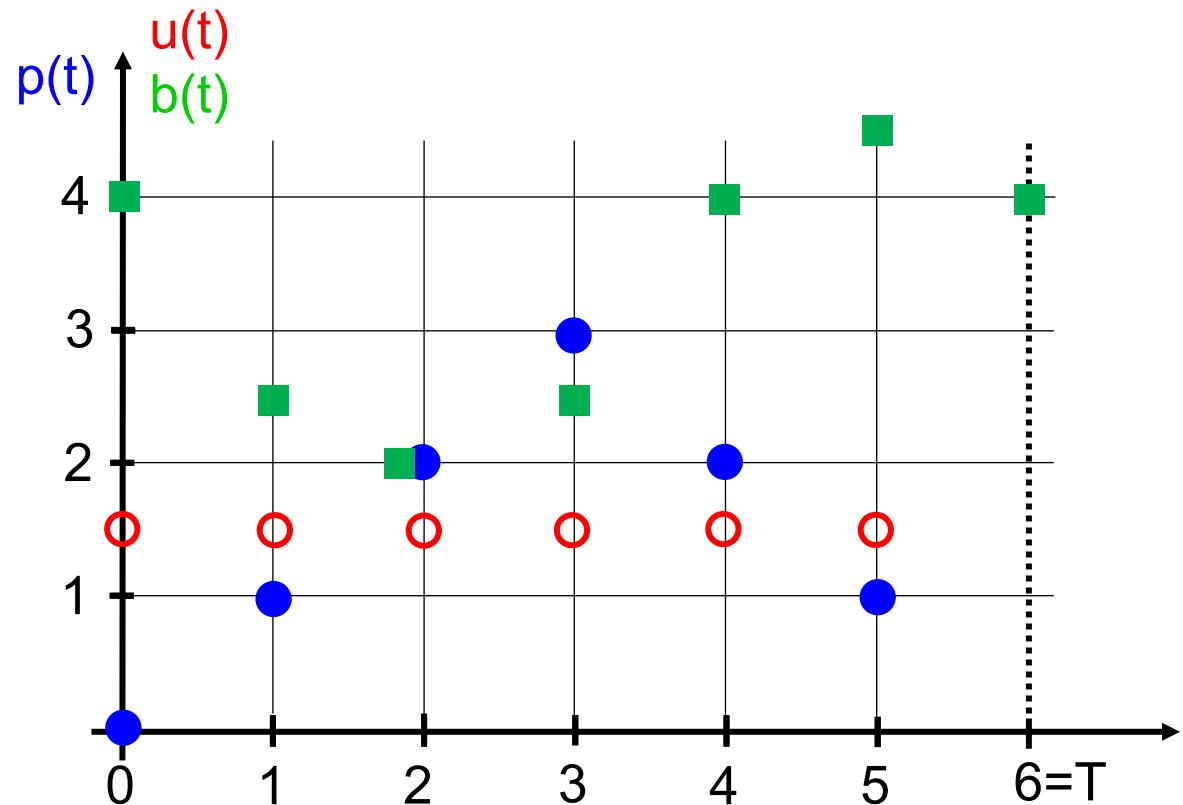
b) We require $b(0) \leq b(T)$, that is,

$$\sum_{(t)} u(t) \leq \sum_{(t)} p(t) = 9$$

From equality in b) and that the optimal use function should be constant if the battery is neither full nor empty follows

$$u^*(t) = \frac{9}{6} = \frac{3}{2} \quad \text{for all } 0 \leq t < T$$

Case 1: B large, $b(0) = 4$



a) Harvested energy in observation window $[0, T]$:

$$\sum_{(t)} p(t) = 1 + 2 + 3 + 2 + 1 = 9$$

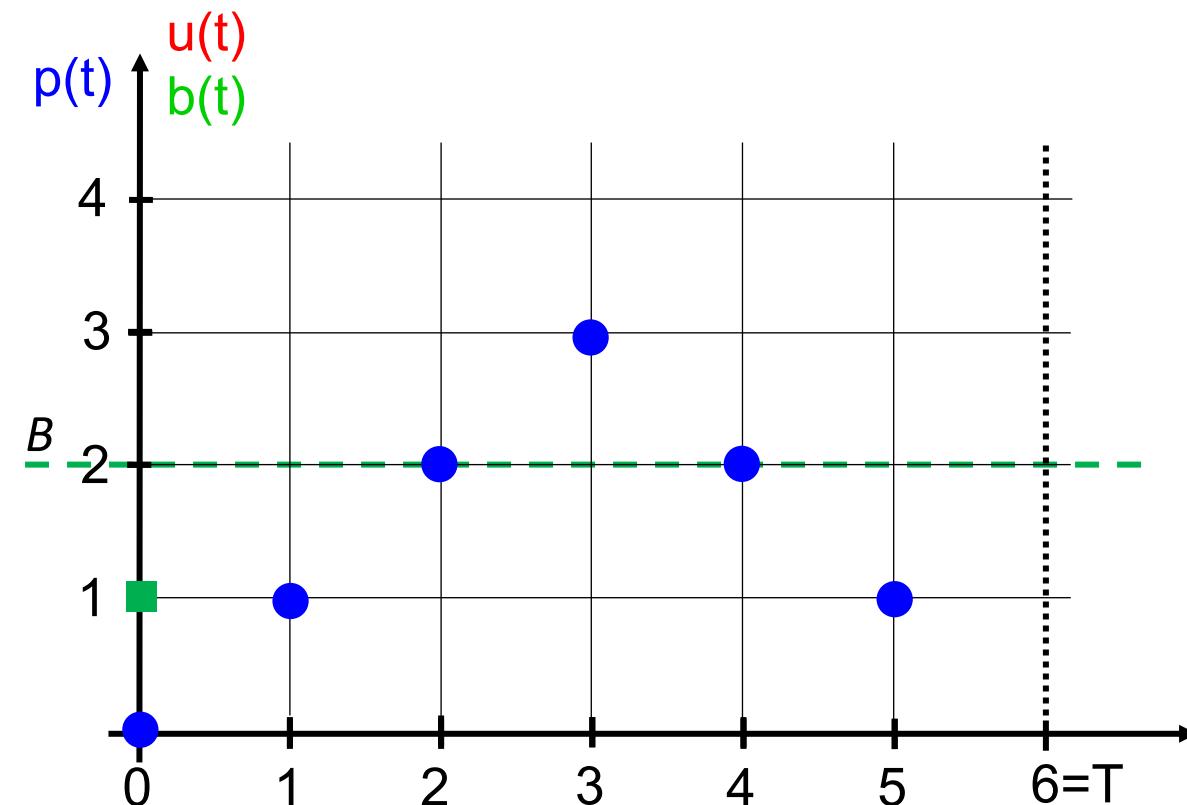
b) We require $b(0) \leq b(T)$, that is,

$$\sum_{(t)} u(t) \leq \sum_{(t)} p(t) = 9$$

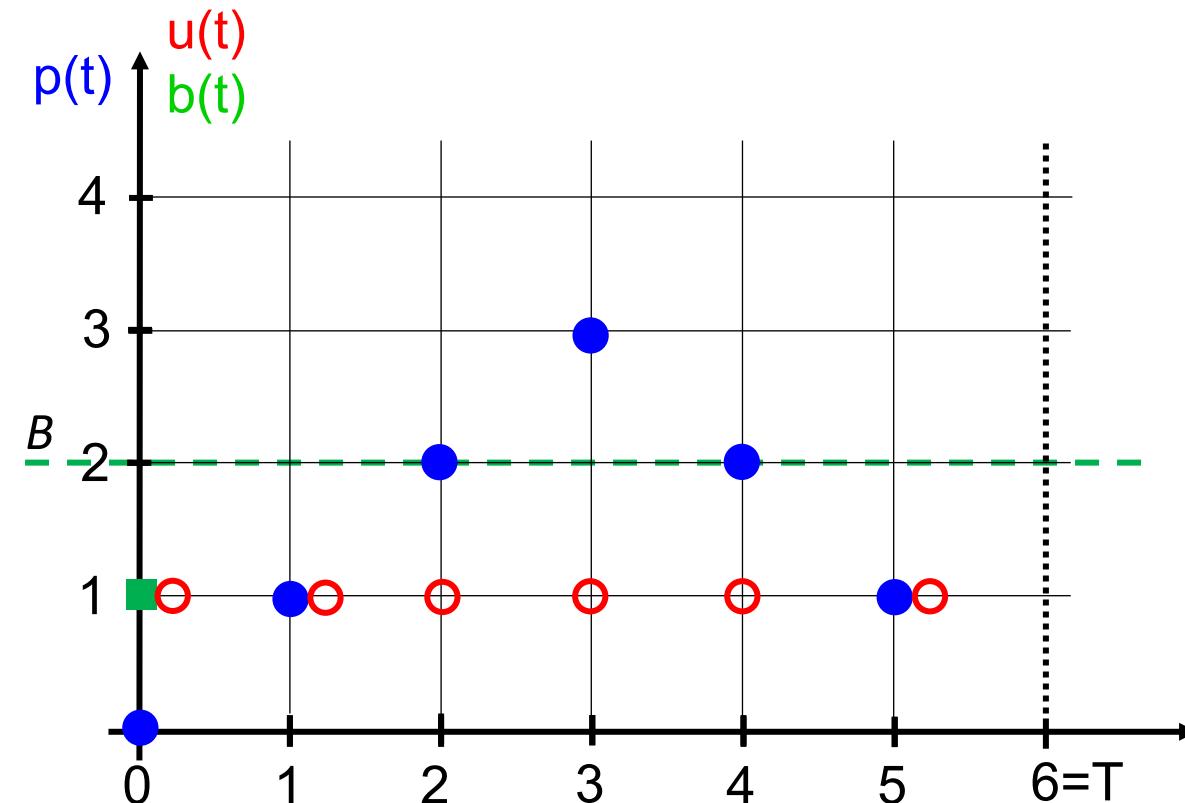
From equality in b) and that the optimal use function should be constant if the battery is neither full nor empty follows

$$u^*(t) = \frac{9}{6} = \frac{3}{2} \quad \text{for all } 0 \leq t < T$$

Case 2: $B = 2$, $b(0) = 1$

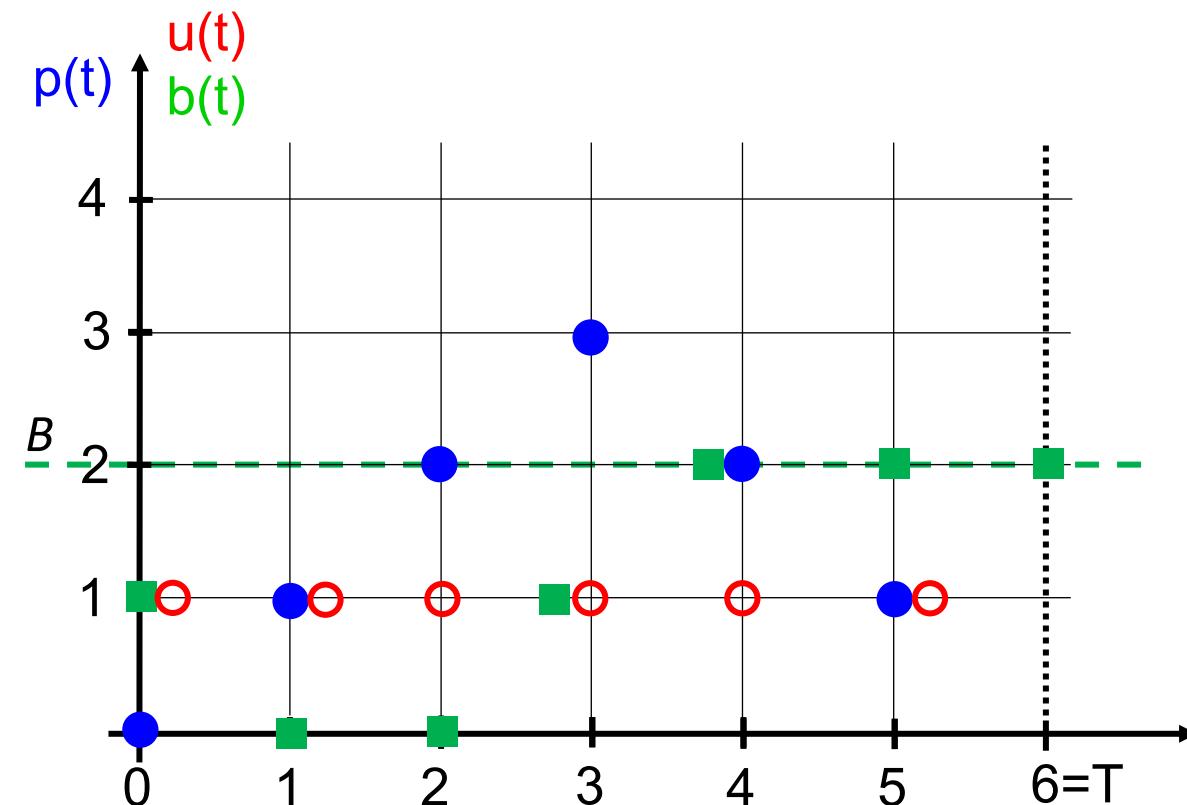


Case 2: $B = 2$, $b(0) = 1$



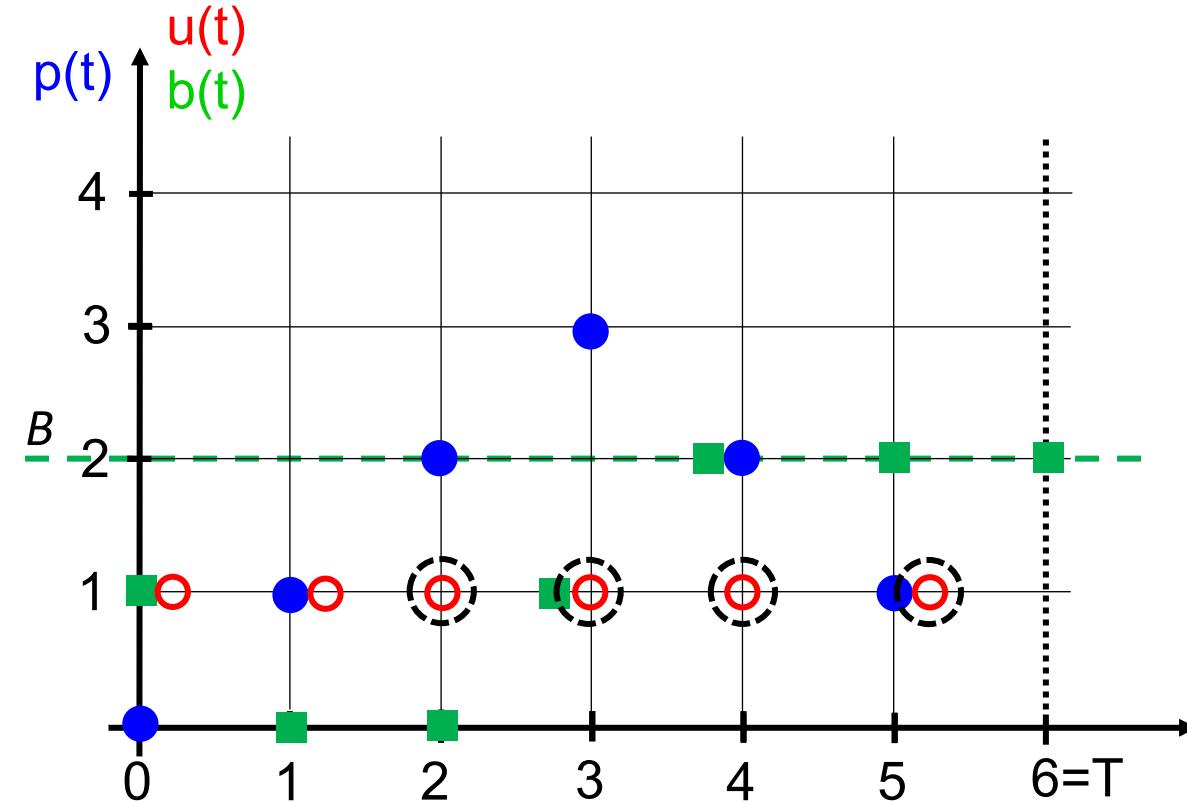
1. Slowly increase constant u until just feasible. (Here: $u(t) = 1$ for all $0 \leq t < 6$)

Case 2: $B = 2$, $b(0) = 1$



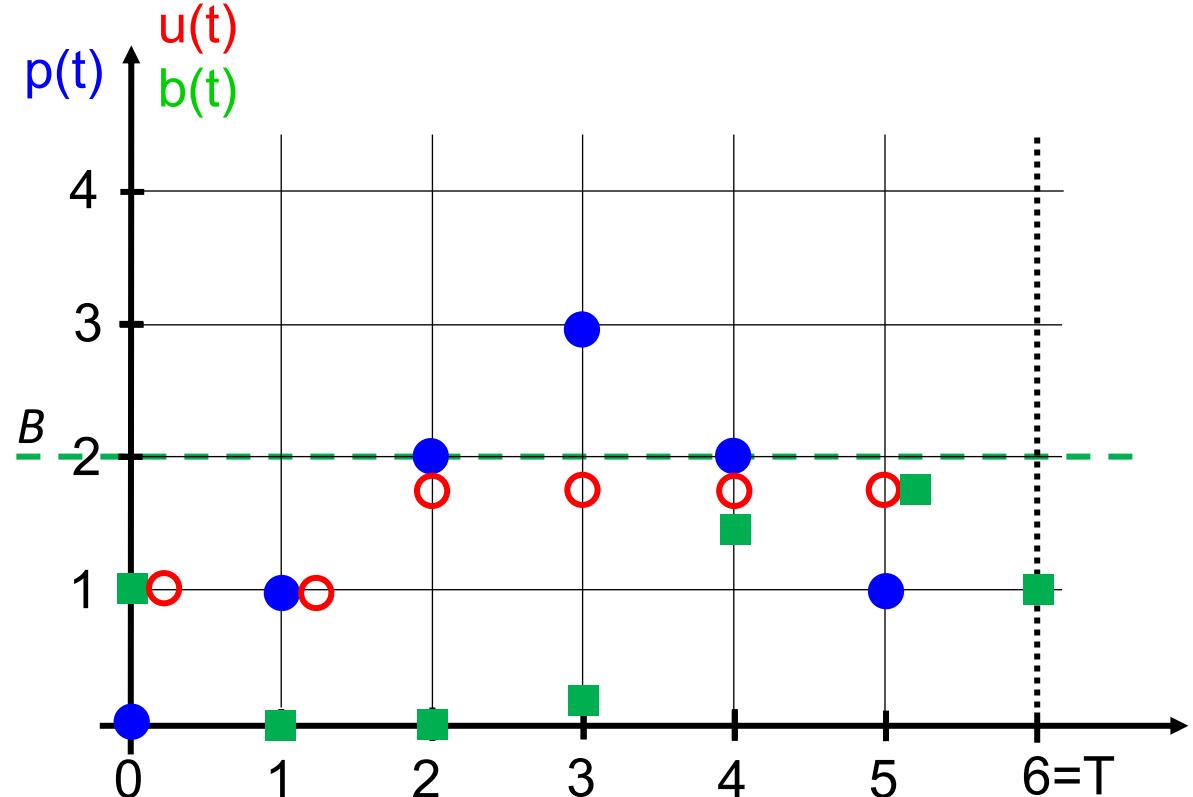
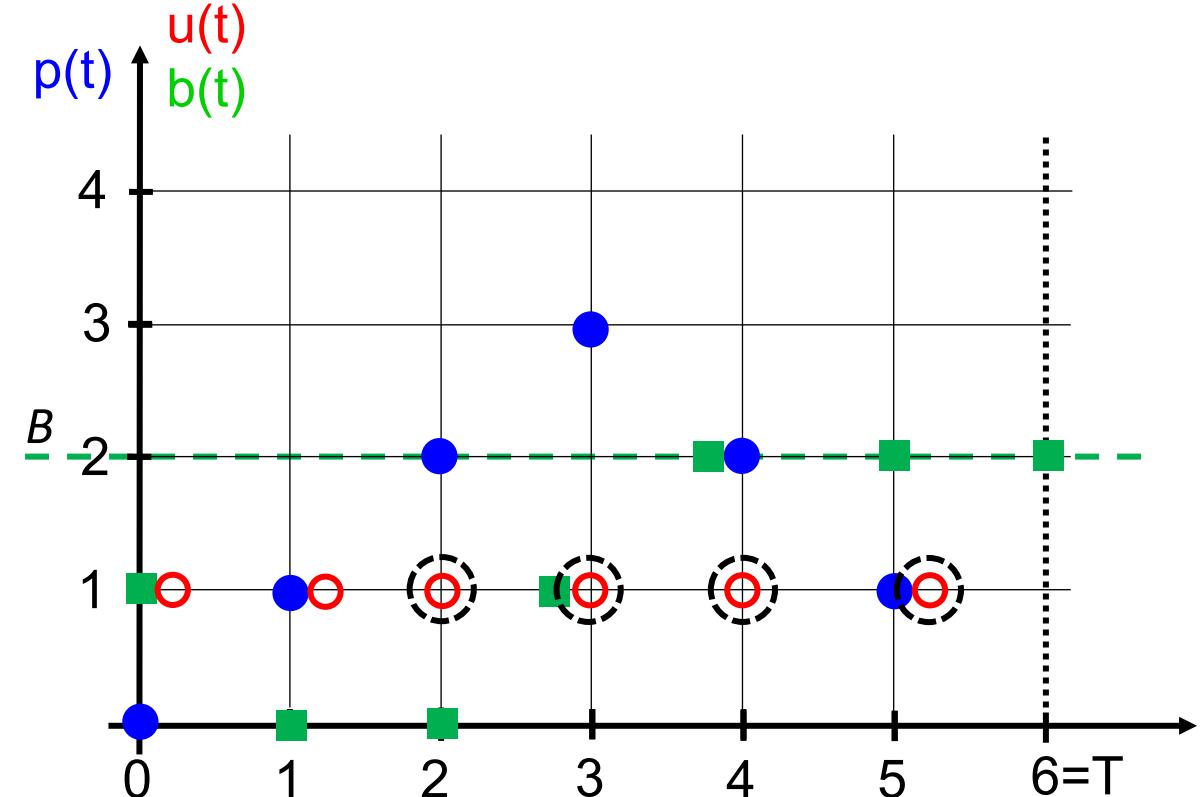
1. Slowly increase constant u until just feasible. (Here: $u(t) = 1$ for all $0 \leq t < 6$)

Case 2: $B = 2$, $b(0) = 1$



1. Slowly increase constant u until just feasible. (Here: $u(t) = 1$ for all $0 \leq t < 6$)
2. Identify $u(t)$ that could still be increased, or stop.

Case 2: $B = 2$, $b(0) = 1$



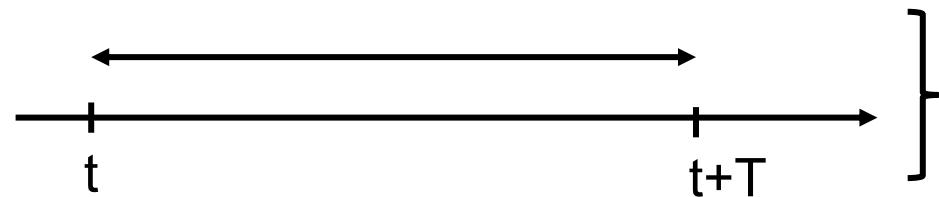
1. Slowly increase constant u until just feasible. (Here: $u(t) = 1$ for all $0 \leq t < 6$)
2. Identify $u(t)$ that could still be increased, or stop.
3. Increase selected $u(t)$ until just feasible, then go to 2.

Application Control

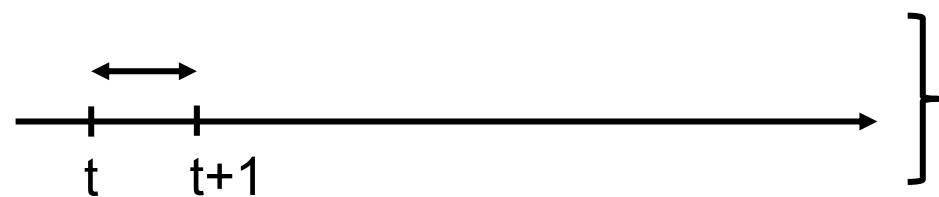
- But what happens if the estimation of the future incoming energy is not correct?
 - If it would be correct, then we would just compute the whole future application control now and would not change anything anymore.
 - This will not work as errors will accumulate and we will end up with many infeasible 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

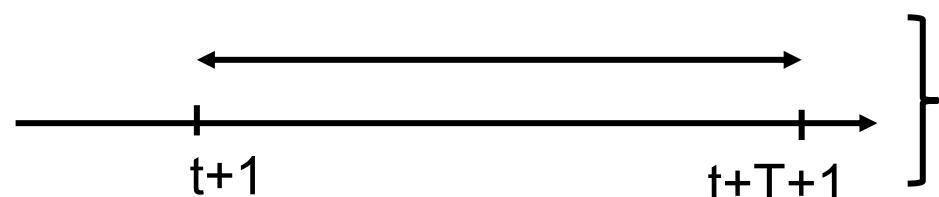
- Finite horizon control:



compute the optimal use function in $[t, t+T)$
using the actual battery state at time t

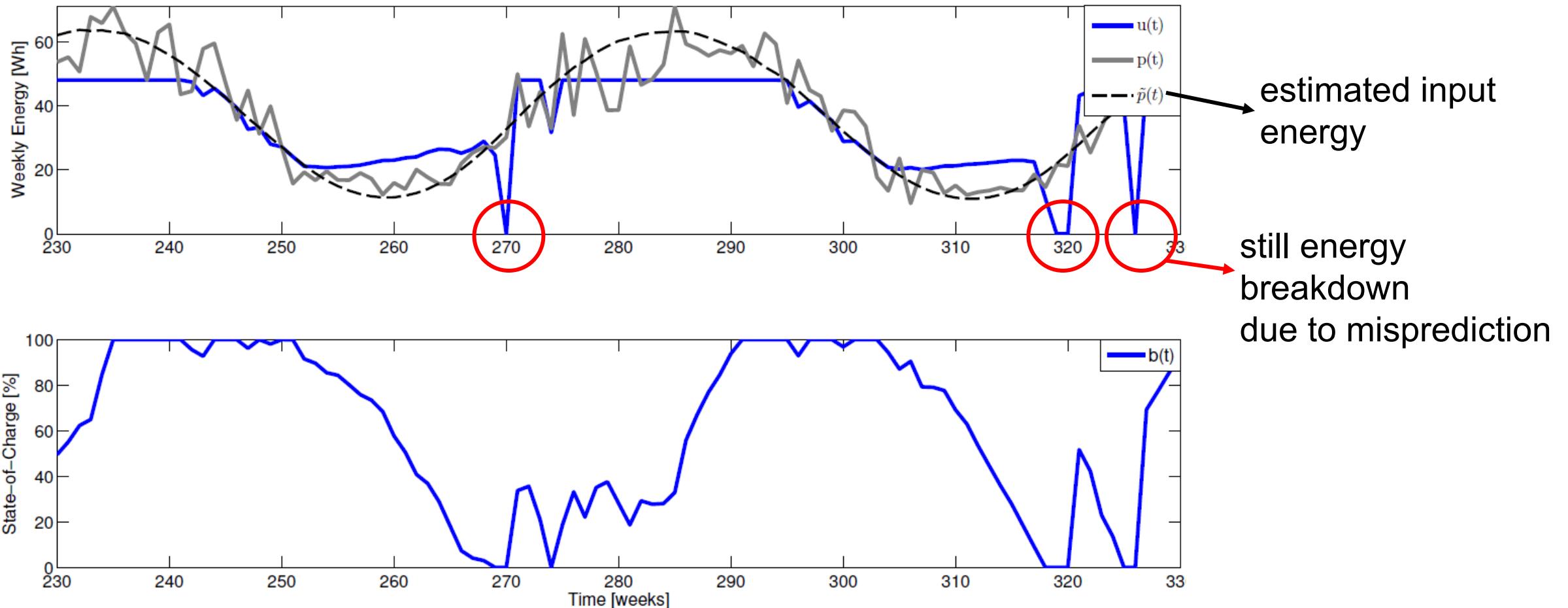


apply this use function in the interval $[t, t+1]$.

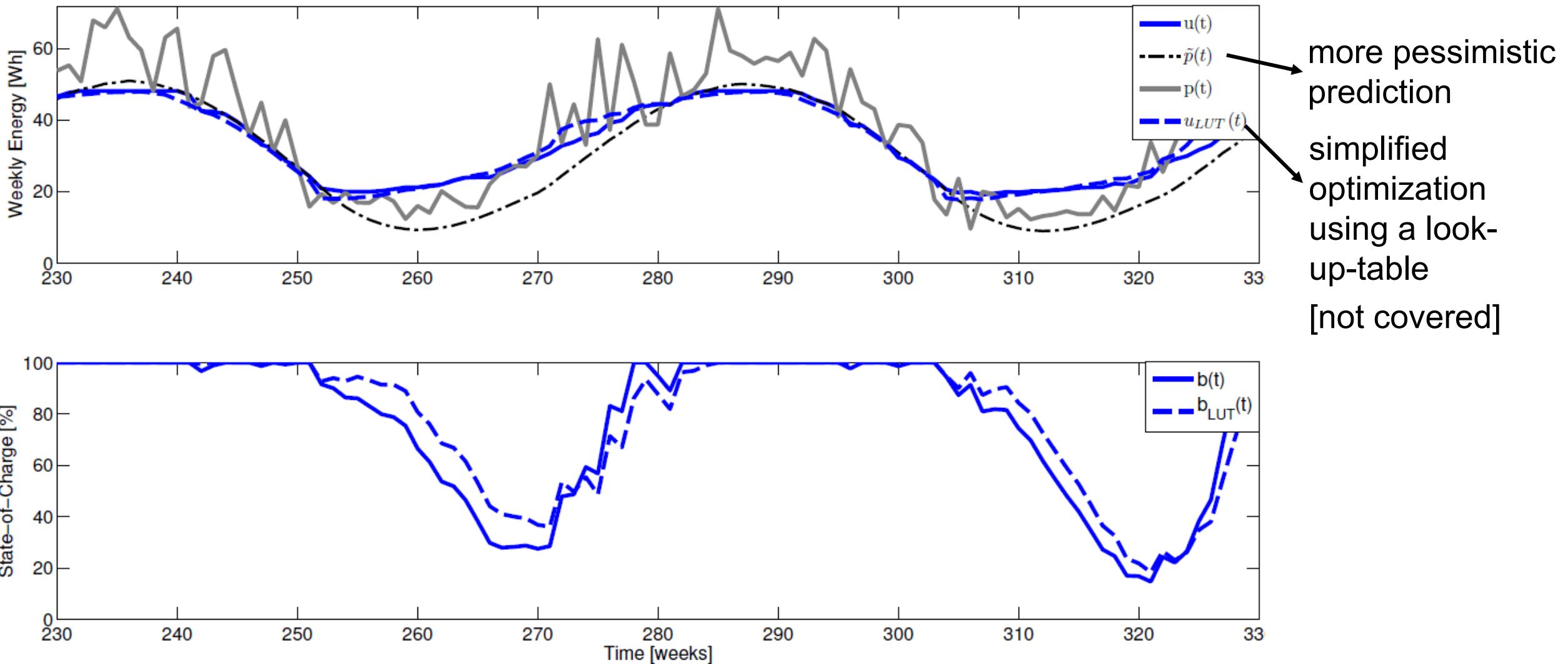


compute the optimal use function in $[t+1, t+T+1)$
using the actual batter state at time $t+1$

Application Control using Finite Horizon



Application Control using Finite Horizon



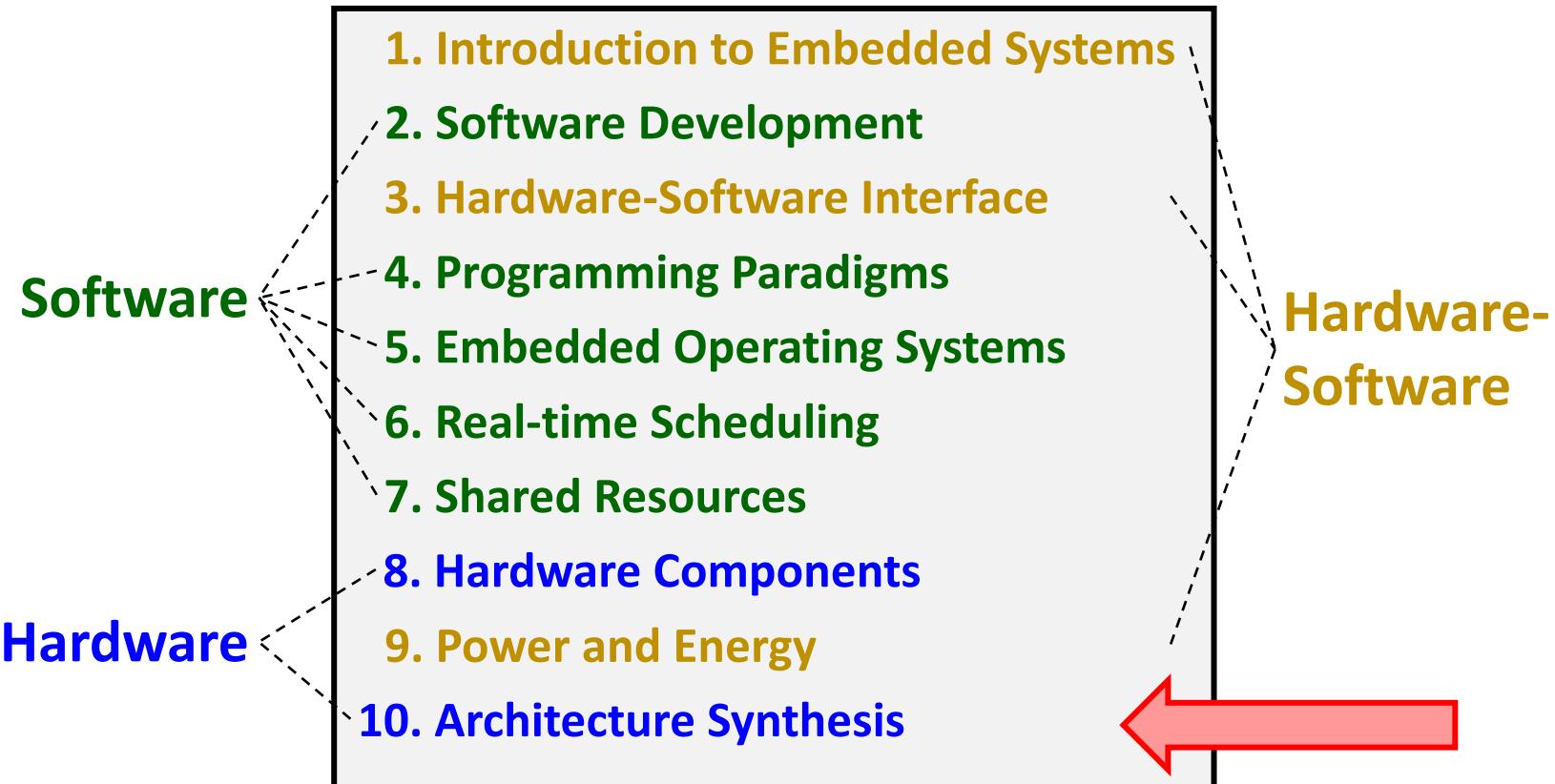
Introduction to Embedded Systems

10. Architecture Synthesis

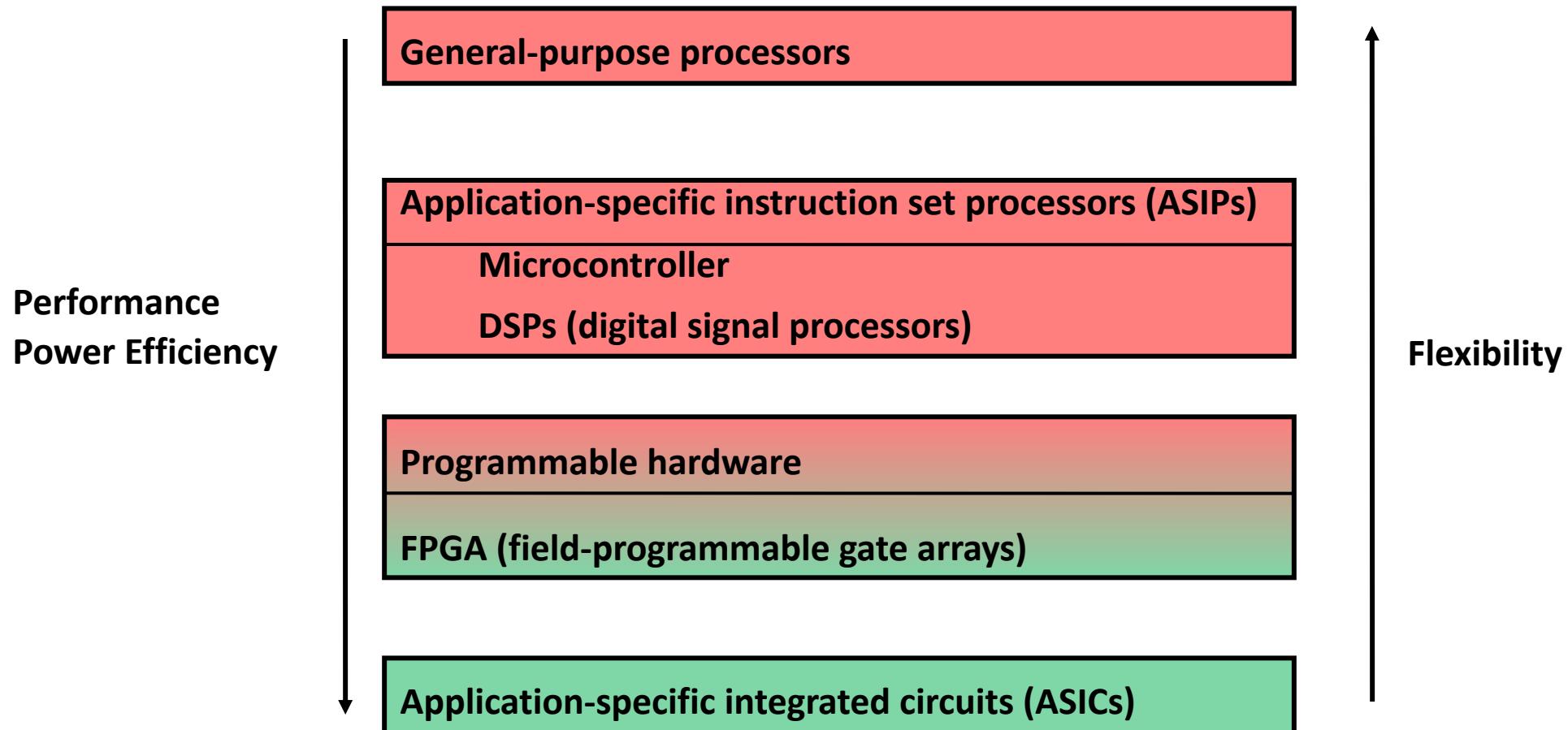
Prof. Dr. Marco Zimmerling



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.

Introduction to Embedded Systems

10. Architecture Synthesis

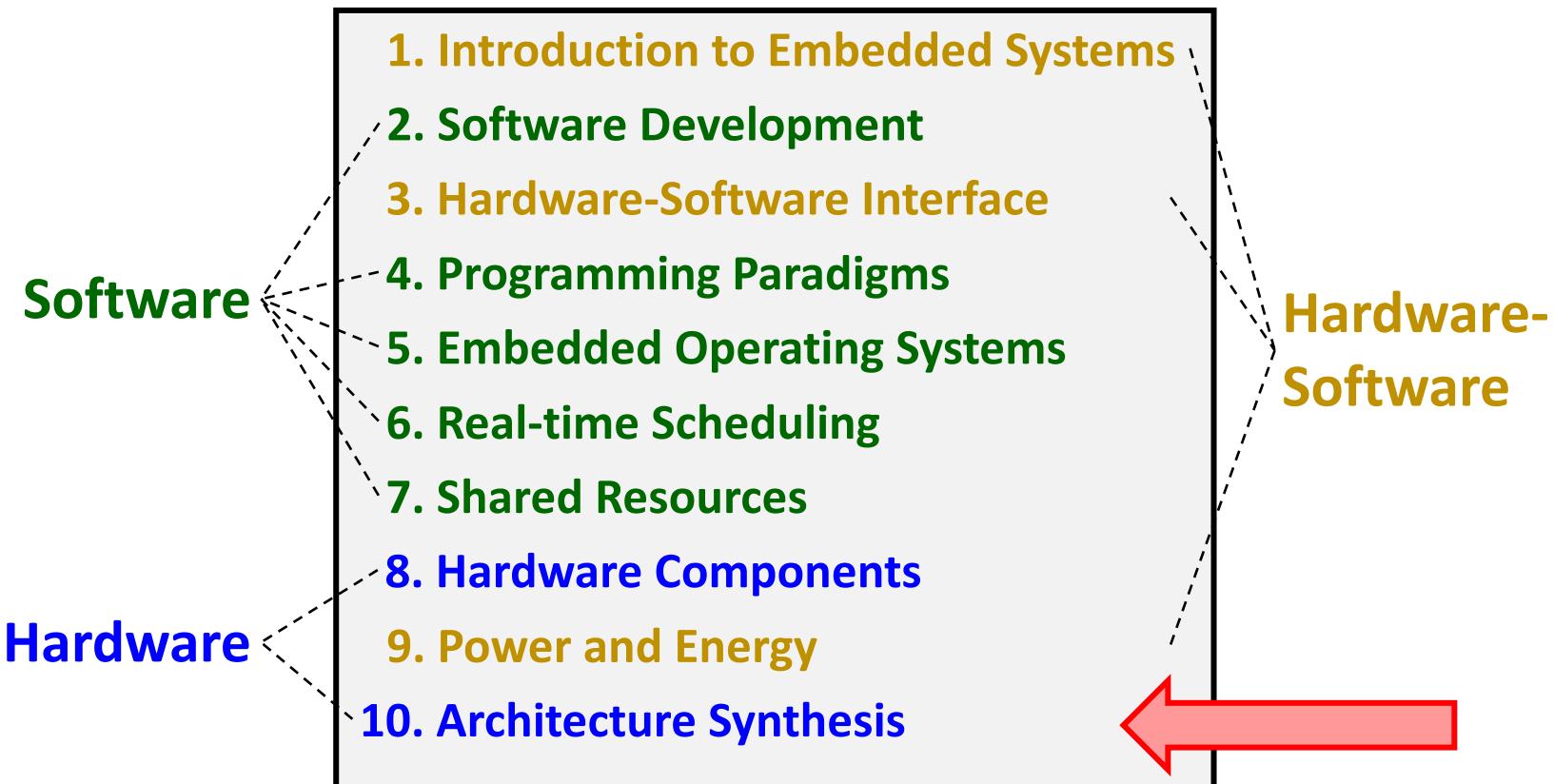
Prof. Dr. Marco Zimmerling



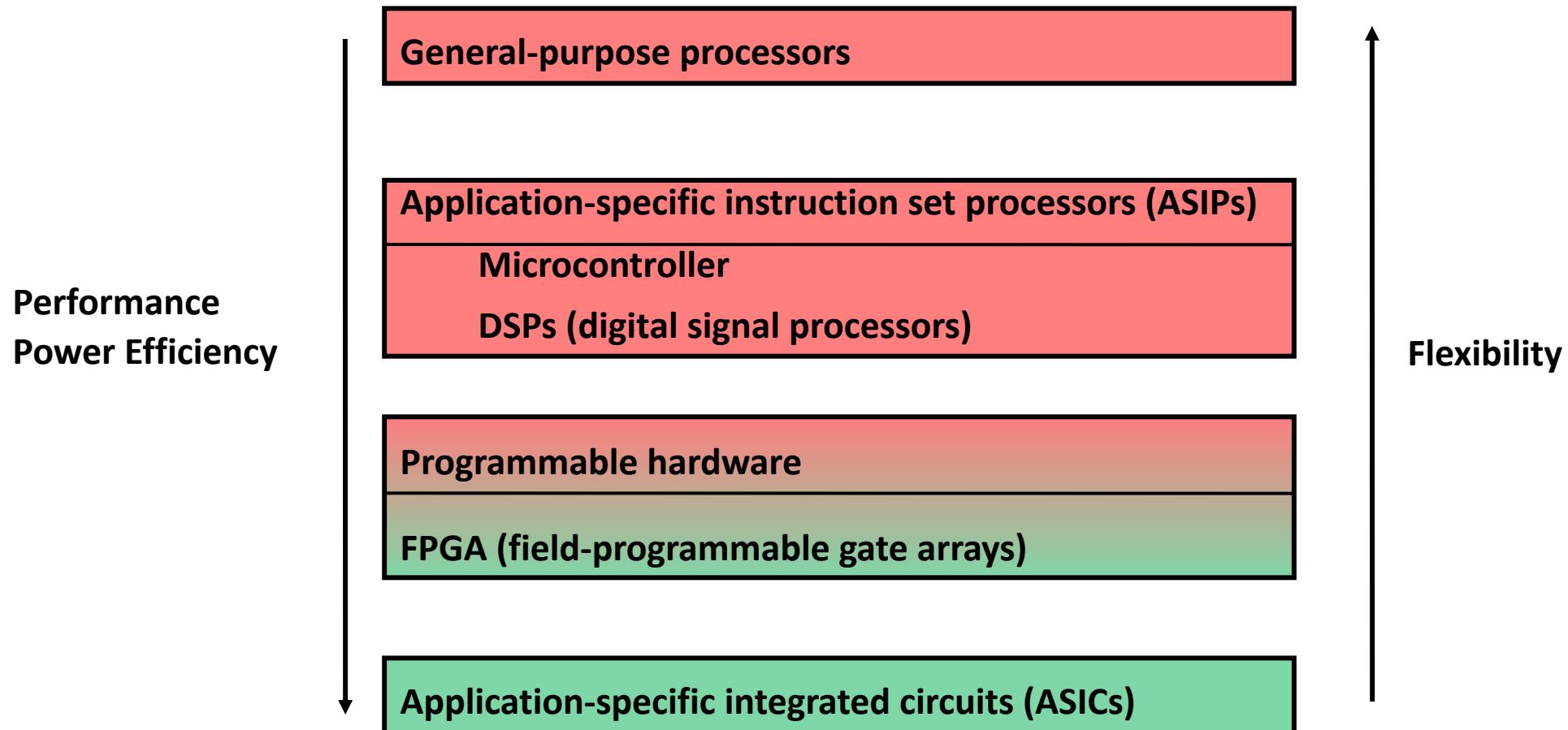
Organization

- Today (January 24):
 - Lecture only on Zoom
 - *Exercise 7 in presence*
- Next week (January 31):
 - Lecture in presence
 - Exercise 8
 - *Mock-up exam questions released on ILIAS*
- In two weeks (February 7):
 - No lecture
 - *Mock-up exam discussed in exercise sessions*
 - Sample solutions to mock-up exam will be available on ILIAS
- Please participate in the ***course evaluation*** until January 29!

Lecture Overview



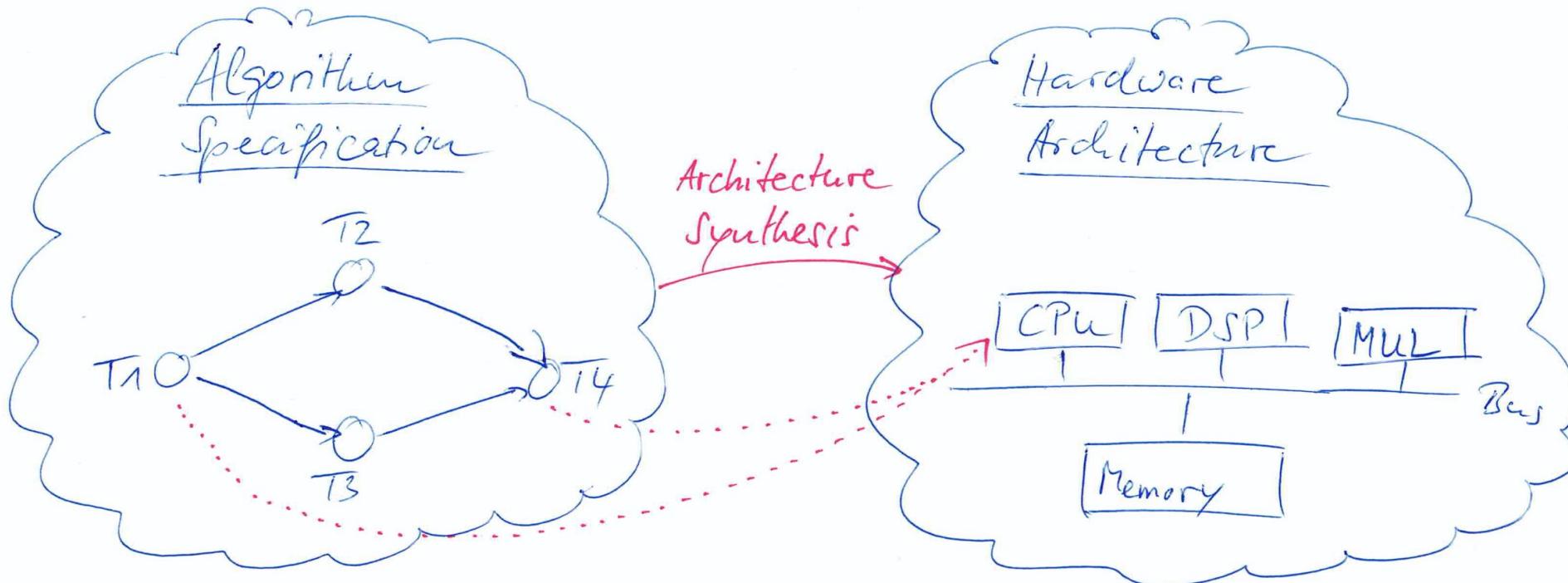
Implementation Alternatives



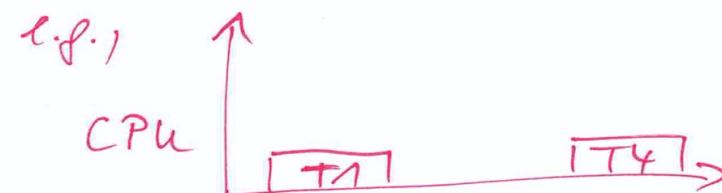
Architecture Synthesis

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.



1. Allocation: Which components? e.g., CPU
2. Binding: Which tasks are executed on which components? e.g., ^{T1 & T4} on CPU
3. Scheduling: In what order / when should tasks be executed?

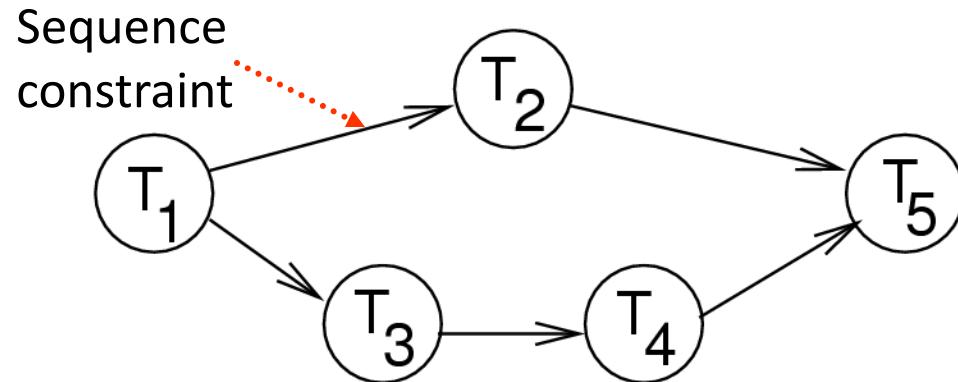


Specification Models

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.

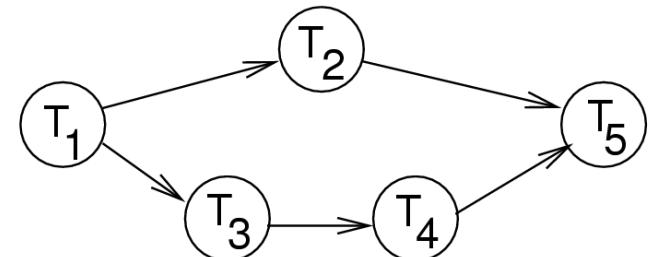
A **dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a partial order.

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

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

Dependence Graph

- 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

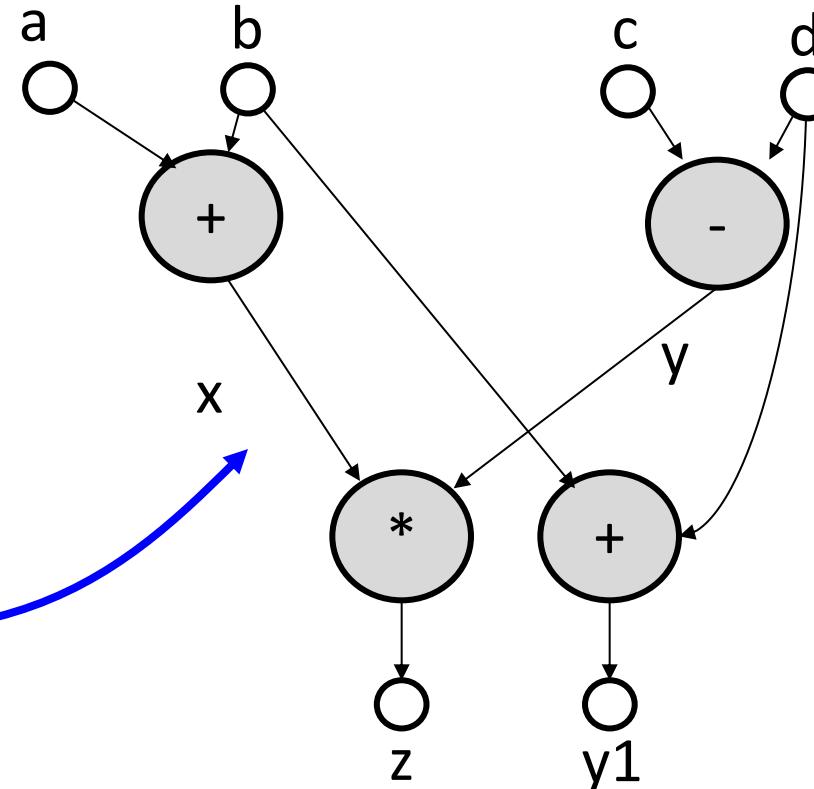
given basic block:

```
x = a + b;  
y = c - d;  
z = x * y;  
y = b + d;
```

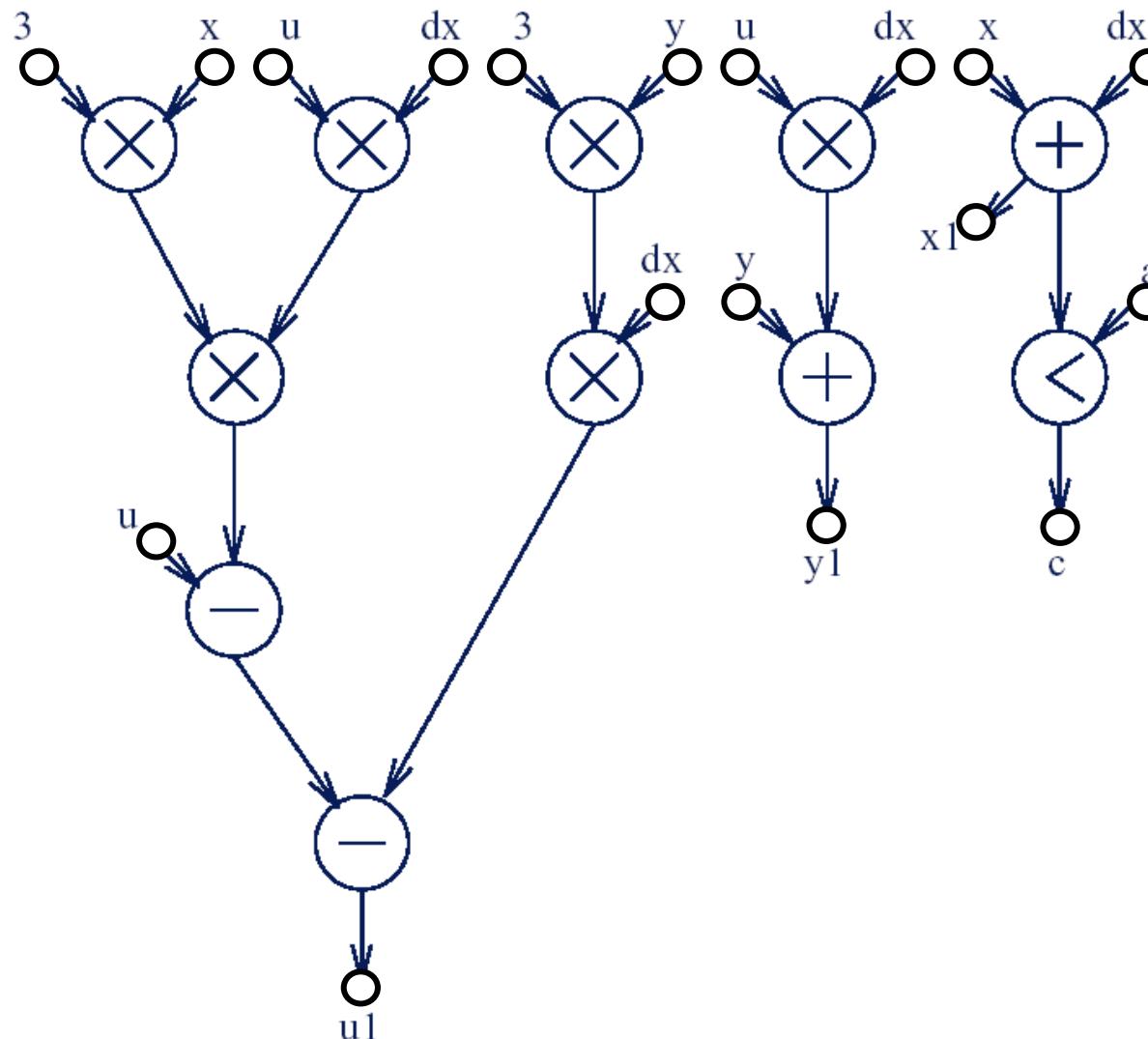
single assignment
form:

```
x = a + b;  
y = c - d;  
z = x * y;  
y1 = b + d;
```

dependence graph

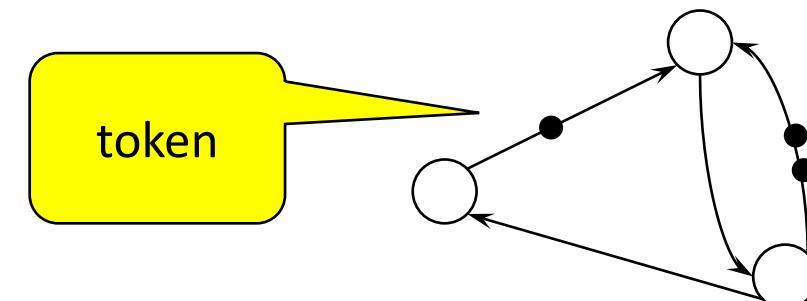
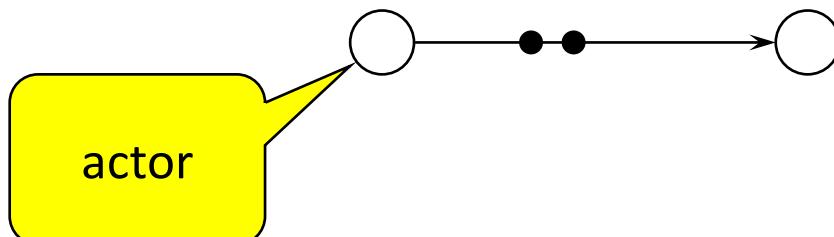


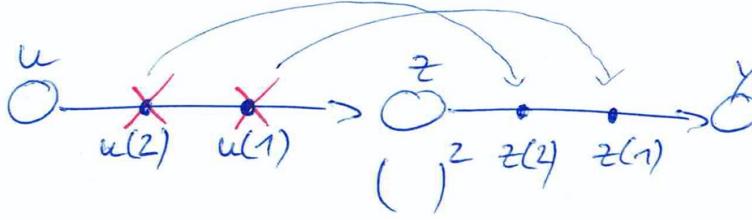
Example of a Dependence Graph



Marked Graph (MG)

- A *marked graph* $G = (V, A, del)$ consists of
 - nodes (actors) $v \in V$
 - edges $a = (v_i, v_j) \in A, A \subseteq V \times V$
 - 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: $del = \begin{pmatrix} del_1 \\ \dots \\ del_i \\ \dots \\ del_{|A|} \end{pmatrix}$

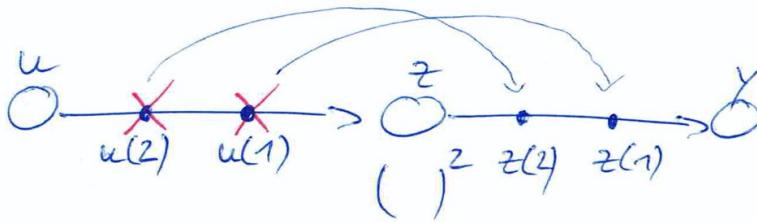




$$z(1) = (u(1))^2$$

$$z(2) = (u(2))^2$$

$$z(t) = (u(t))^2 \quad \forall t \geq 1$$

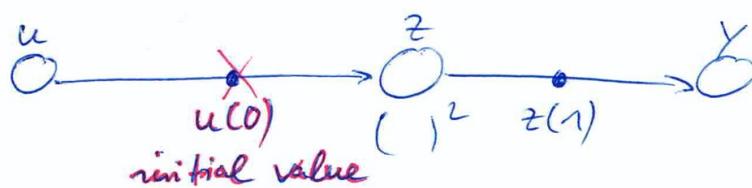


$$z(1) = (u(1))^2$$

$$z(2) = (u(2))^2$$

$$\vdots$$

$$z(t) = (u(t))^2 \quad \forall t \geq 1$$



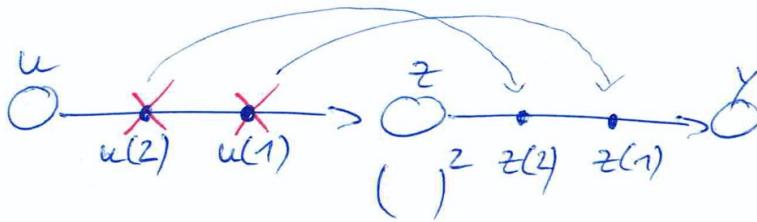
$$z(1) = (u(0))^2$$

$$z(2) = (u(1))^2$$

$$\vdots$$

$$z(t) = (u(\underbrace{t-1})_0)^2 \quad \forall t \geq 1$$

initial value (taken) leads to an index shift of -1

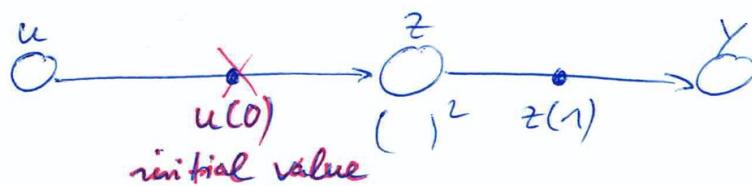


$$z(1) = (u(1))^2$$

$$z(2) = (u(2))^2$$

$$\vdots$$

$$z(t) = (u(t))^2 \quad \forall t \geq 1$$



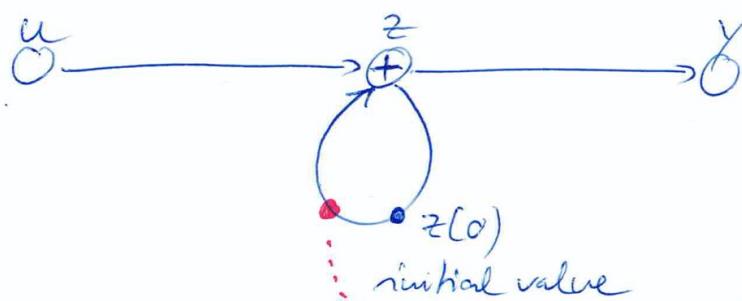
$$z(1) = (u(0))^2$$

$$z(2) = (u(1))^2$$

$$\vdots$$

$$z(t) = (u(t-1))^2 \quad \forall t \geq 1$$

initial value (taken) leads to an index shift of -1



$$z(t) = u(t) + z(t-1) \quad \forall t \geq 1$$

$\rightarrow -2$
a second initial token would lead to an index shift of -2

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 tokens correspond to the processed data.
- Marked graphs are mainly used for modeling regular computations, for example signal flow graphs.

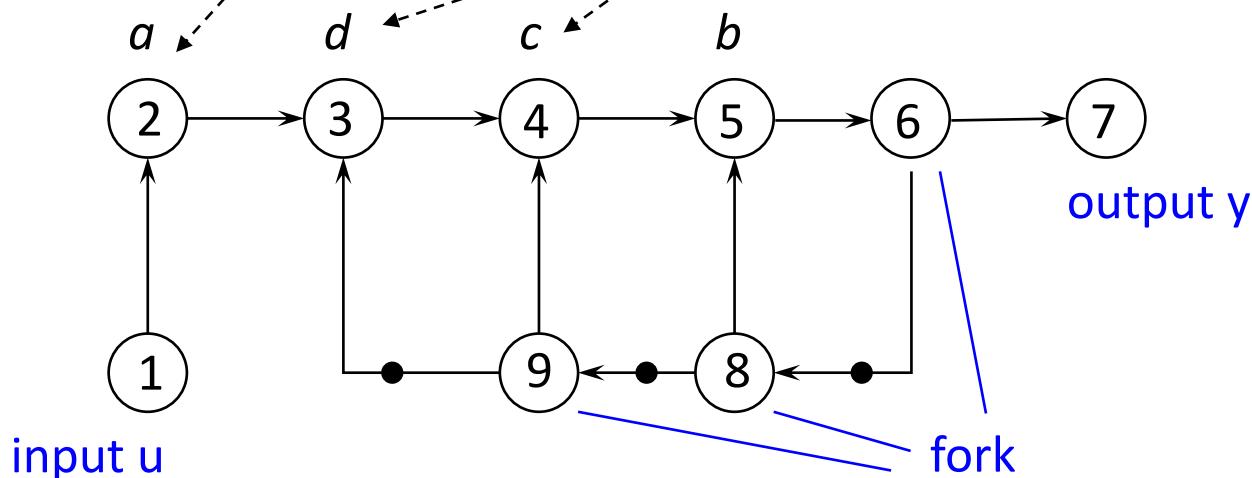
Marked Graph

Example (model of a digital filter with infinite impulse response IIR)

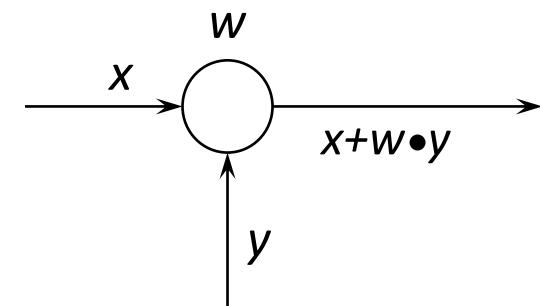
- Filter equation:

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

- Possible model as a *marked graph*:



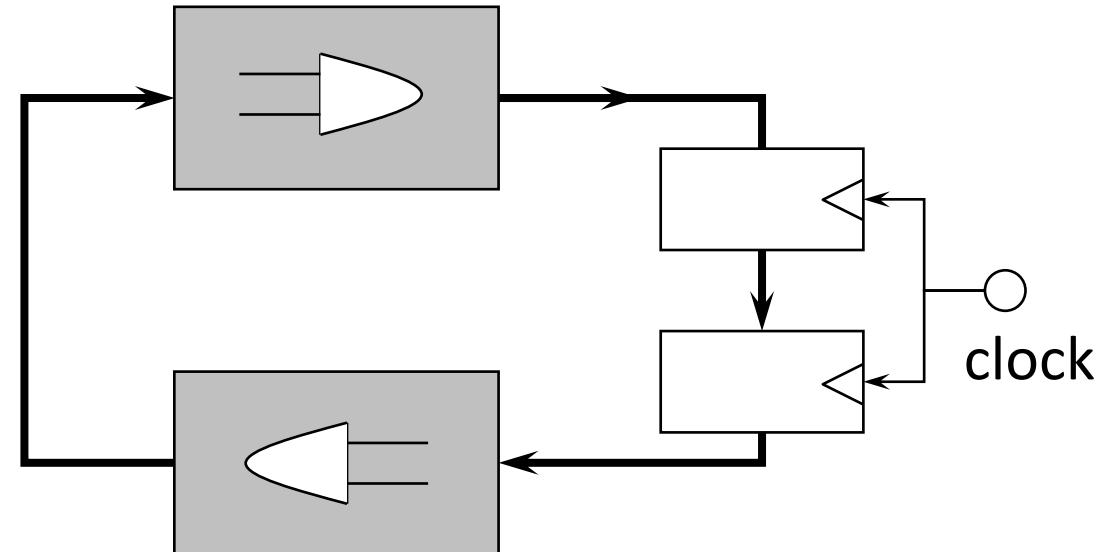
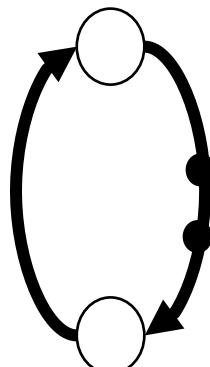
nodes 3-5:



node 2: $x=0$

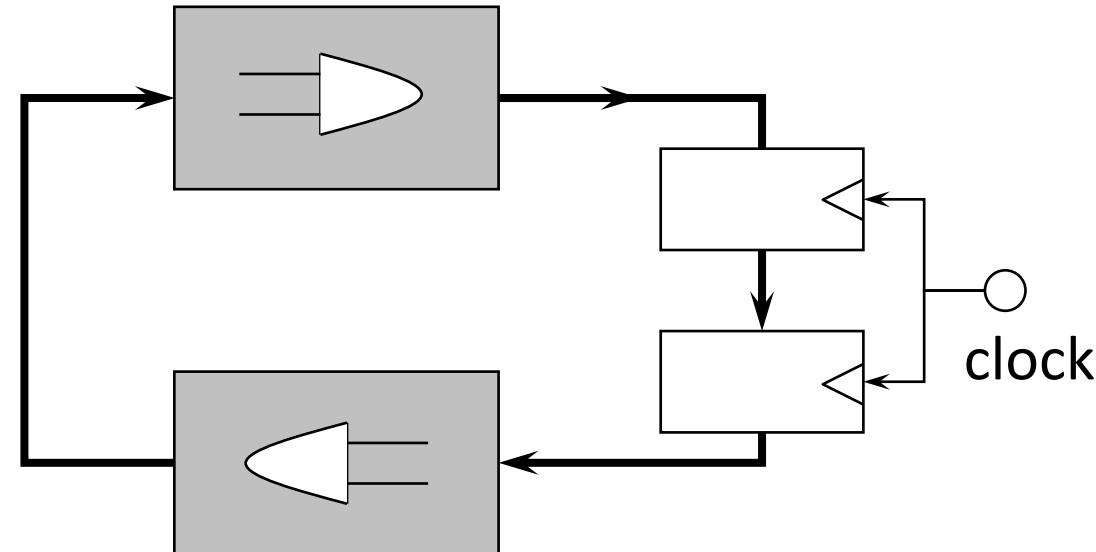
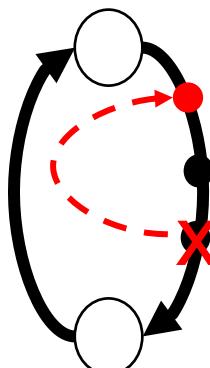
Implementation of Marked Graphs

- 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).



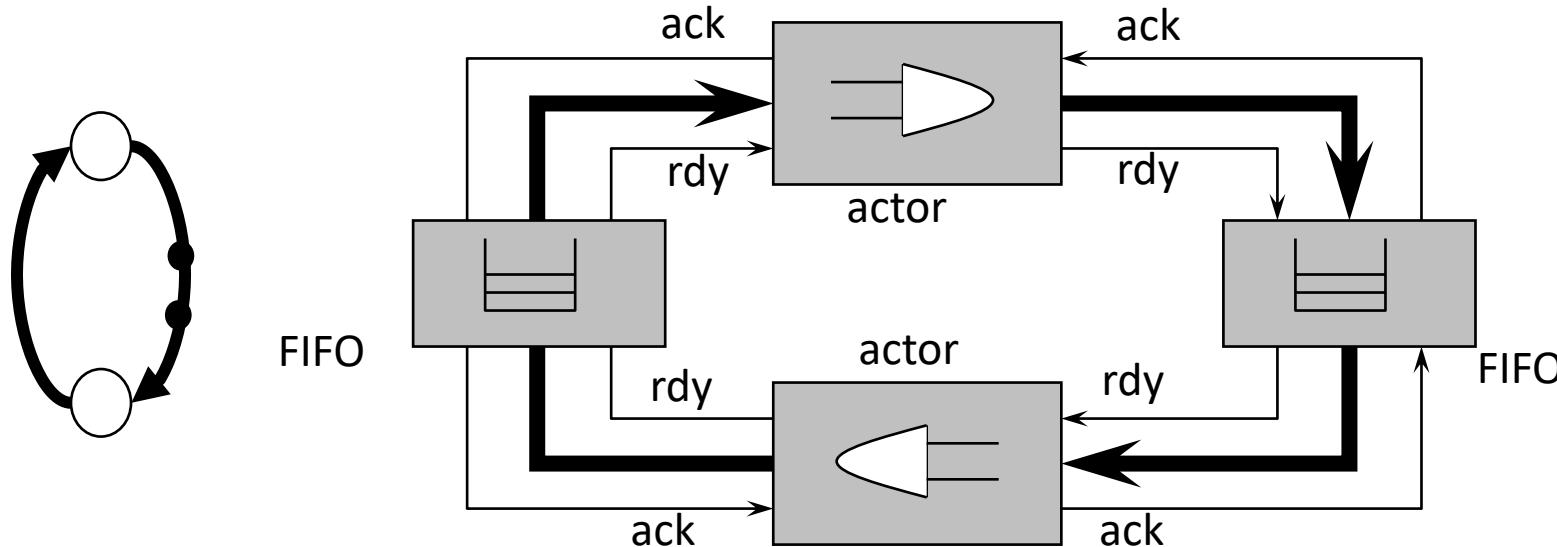
Implementation of Marked Graphs

- 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

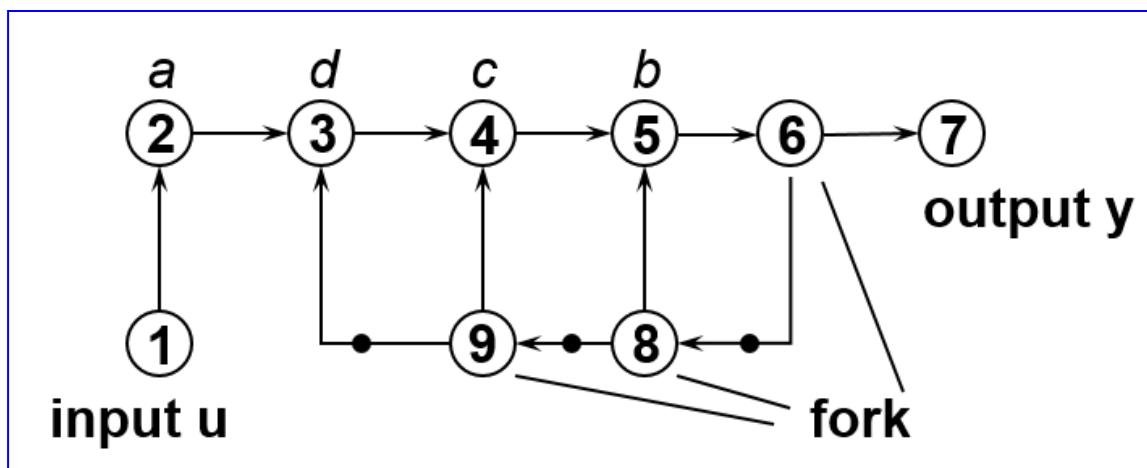
- *Software implementation* with static scheduling:

- 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:
program:

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

```
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

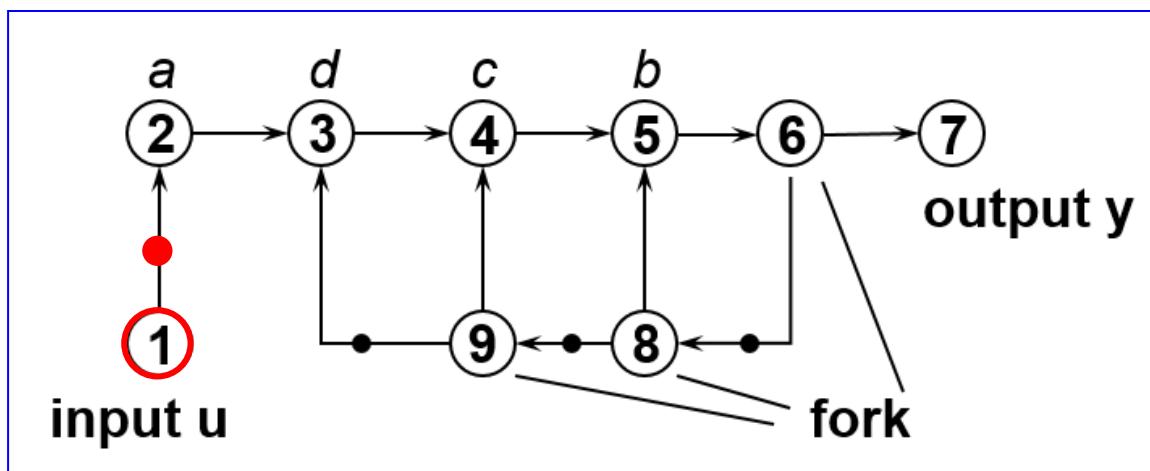
- *Software implementation* with static scheduling:

- 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:
program:

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

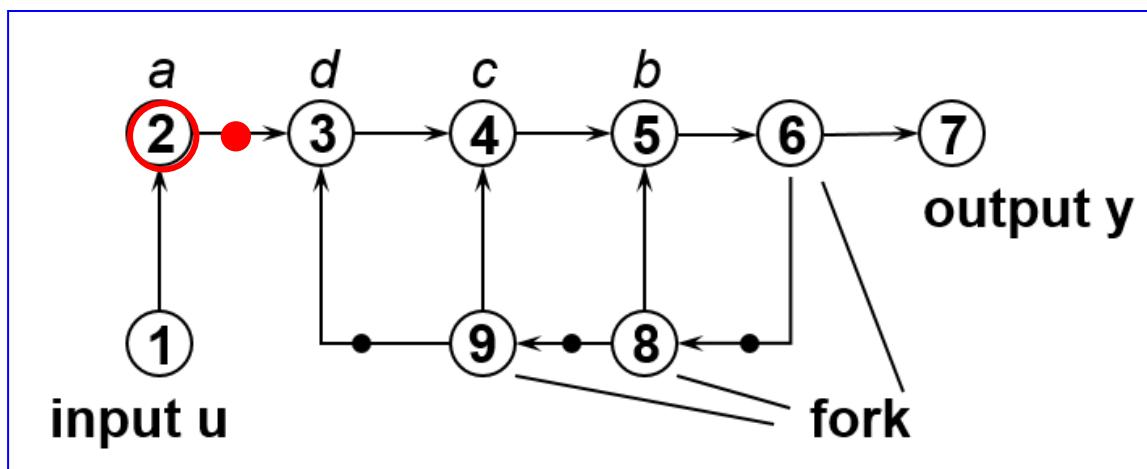
- *Software implementation* with static scheduling:

- 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:
program:

(1, **2**, 3, 9, 4, 8, 5, 6, 7)

```
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

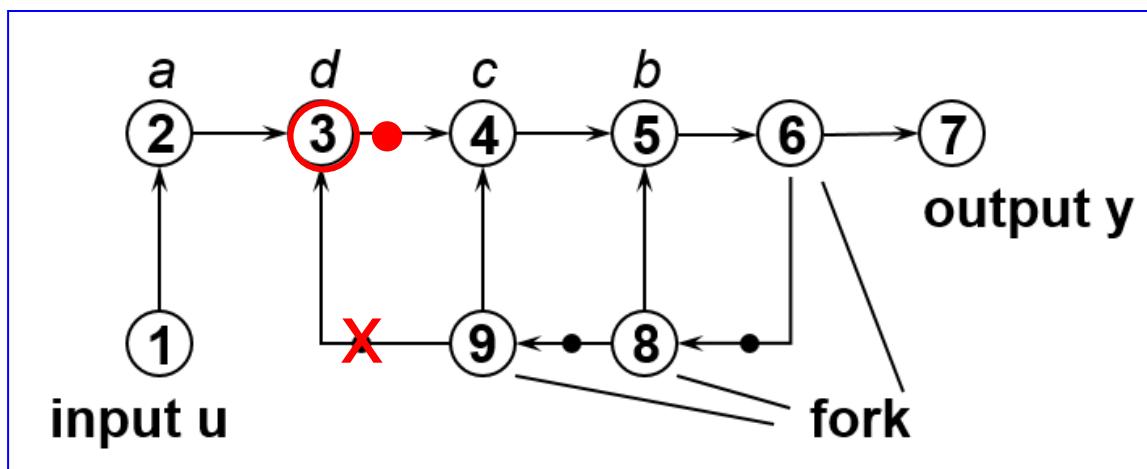
- *Software implementation* with static scheduling:

- 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:
program:

(1, 2, **3**, 9, 4, 8, 5, 6, 7)

```
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

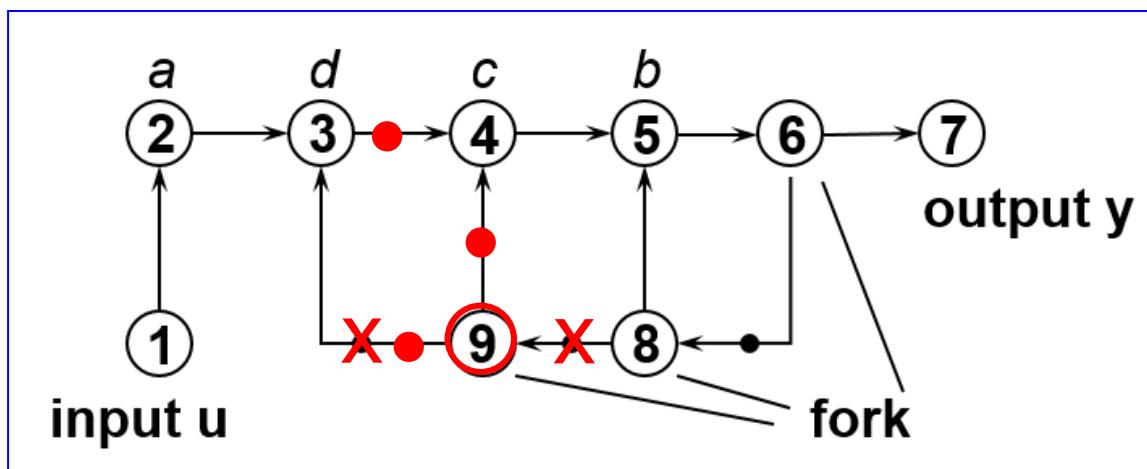
- *Software implementation* with static scheduling:

- 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:
program:

(1, 2, 3, **9**, 4, 8, 5, 6, 7)

```
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

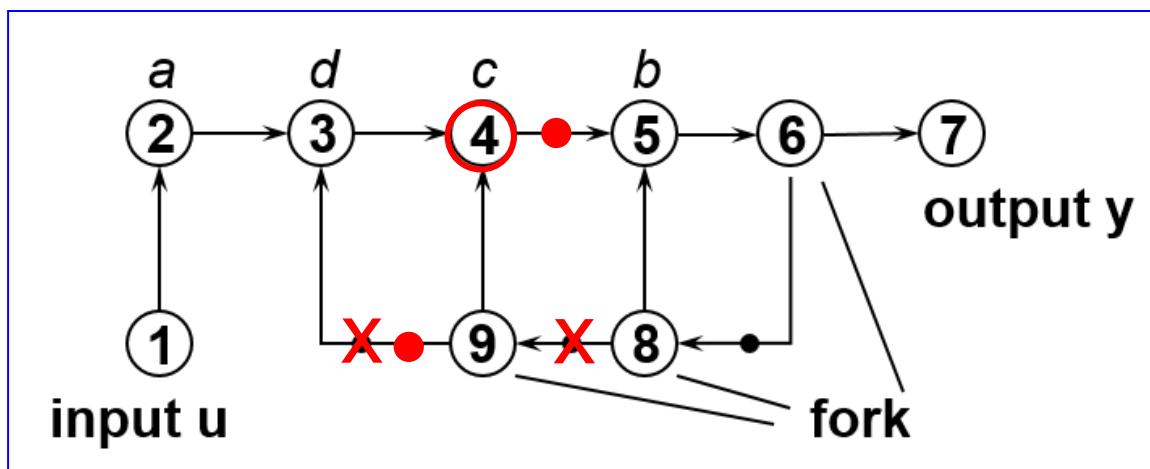
- *Software implementation* with static scheduling:

- 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:
program:

(1, 2, 3, 9, **4**, 8, 5, 6, 7)

```
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

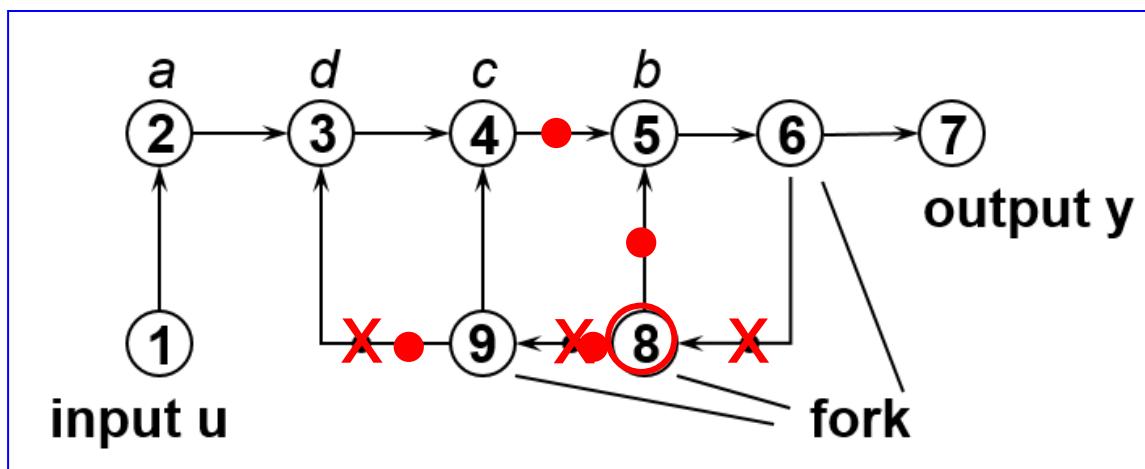
- *Software implementation* with static scheduling:

- 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:
program:

(1, 2, 3, 9, 4, **8**, 5, 6, 7)

```
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); }
```



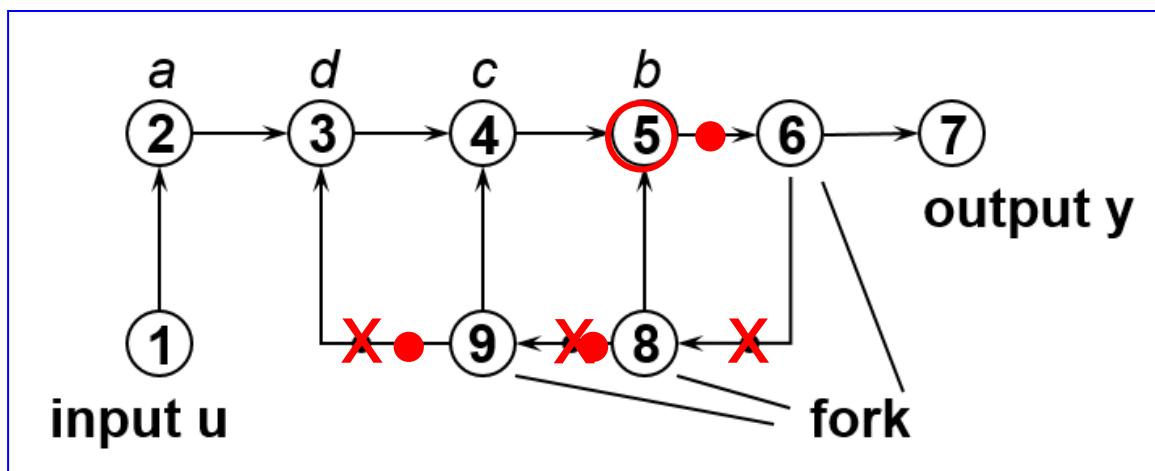
Implementation of Marked Graphs

- *Software implementation* with 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:
program:

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

```
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

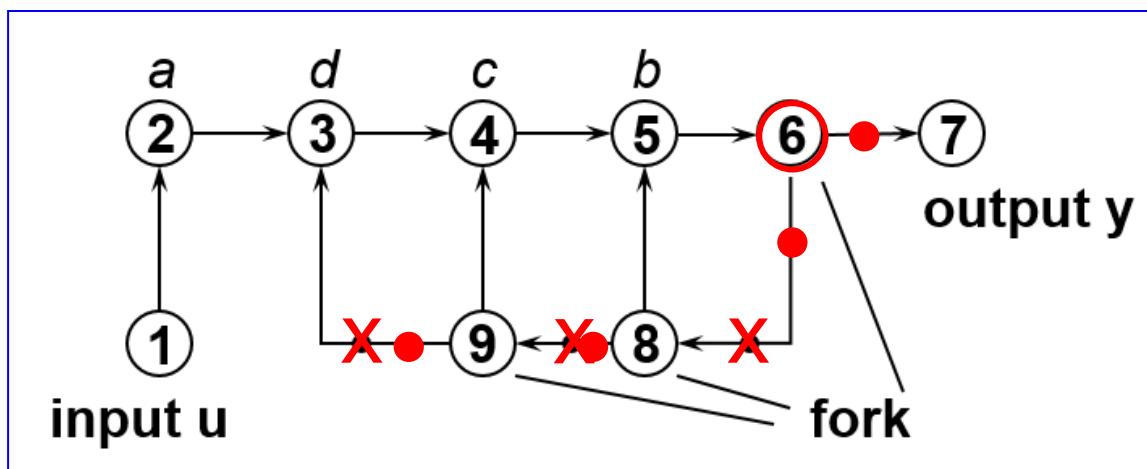
- *Software implementation* with static scheduling:

- 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:
program:

(1, 2, 3, 9, 4, 8, 5, **6**, 7)

```
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

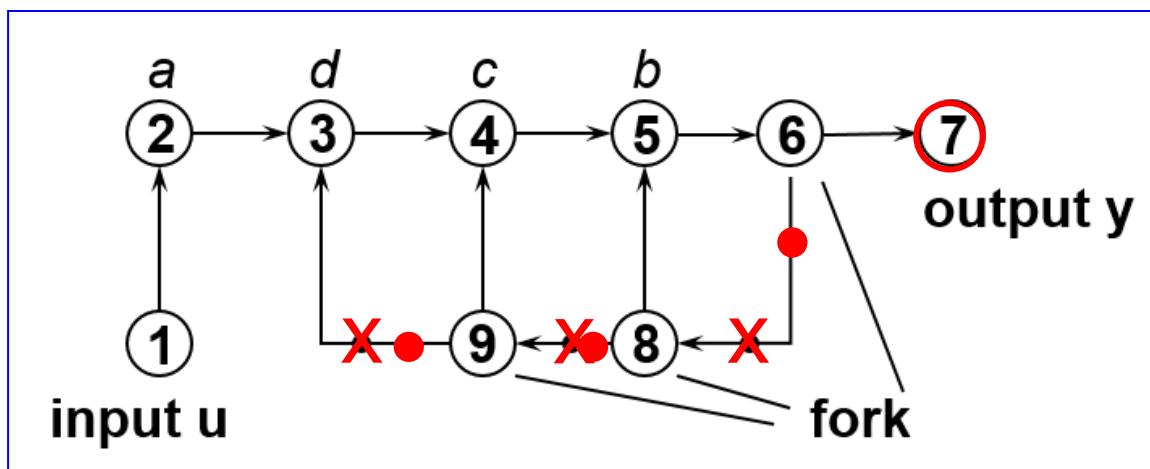
- *Software implementation* with static scheduling:

- 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:
program:

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

```
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); }
```



same token distribution as at the start of the execution

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):

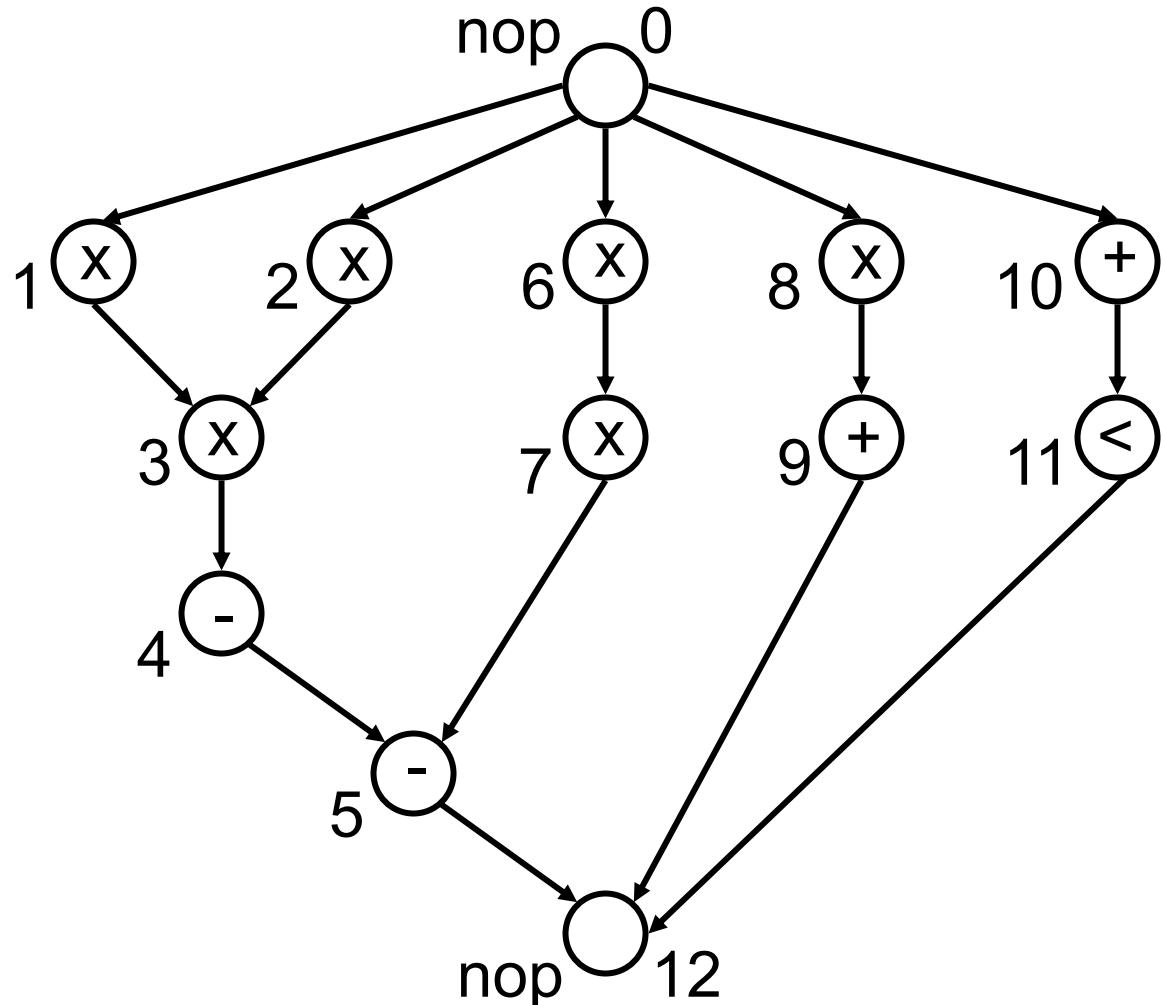
```
int diffeq(int x, int y, int u, int dx, int a) {  
    int x1, u1, y1;  
    while ( x < a ) {  
        x1 = x + dx;  
        u1 = u - (3 * x * u * dx) - (3 * y * dx);  
        y1 = y + u * dx;  
        x = x1;  
        u = u1;  
        y = y1;  
    }  
    return y;  
}
```

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;  
}
```

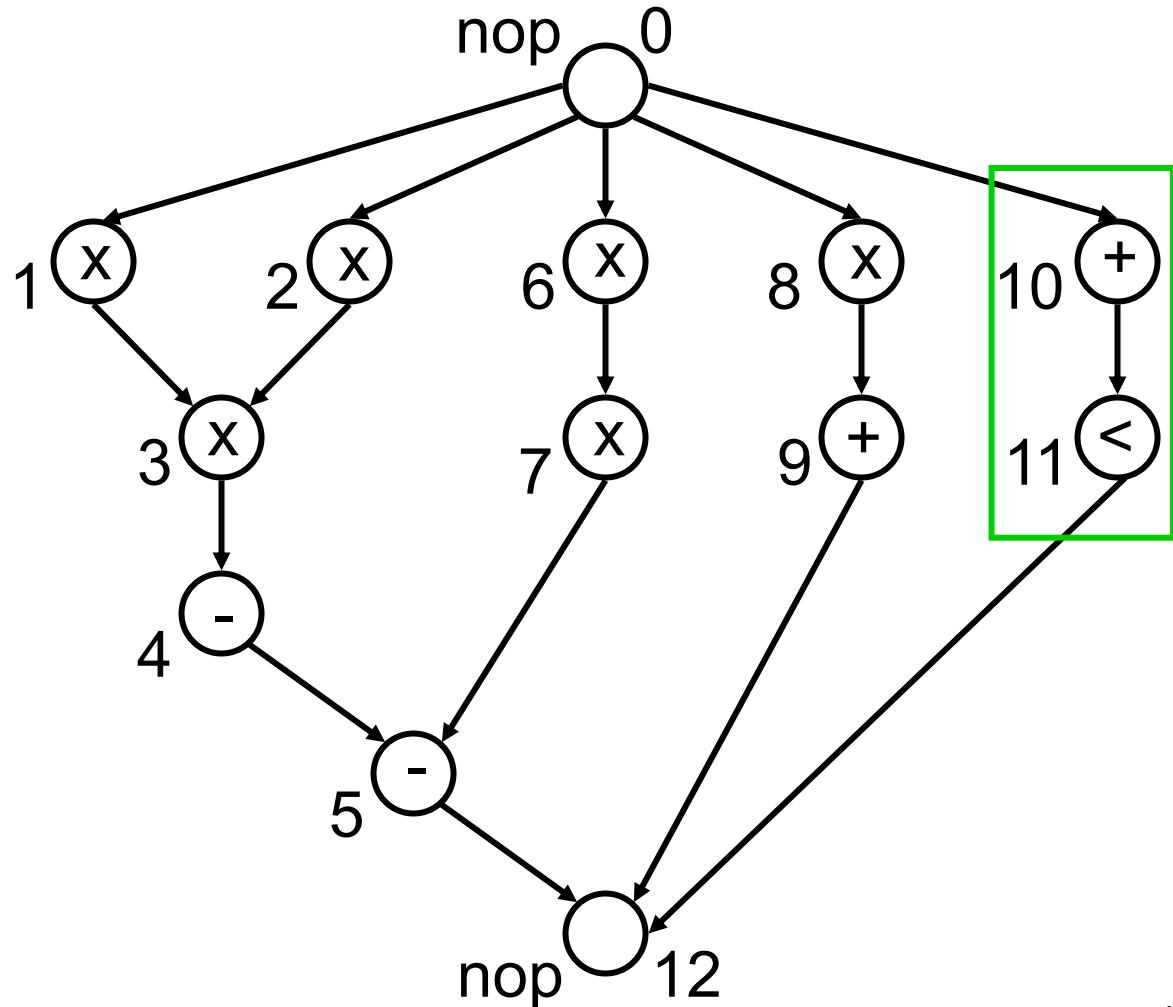


Models for Architecture Synthesis - Example

- Corresponding 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;  
}
```

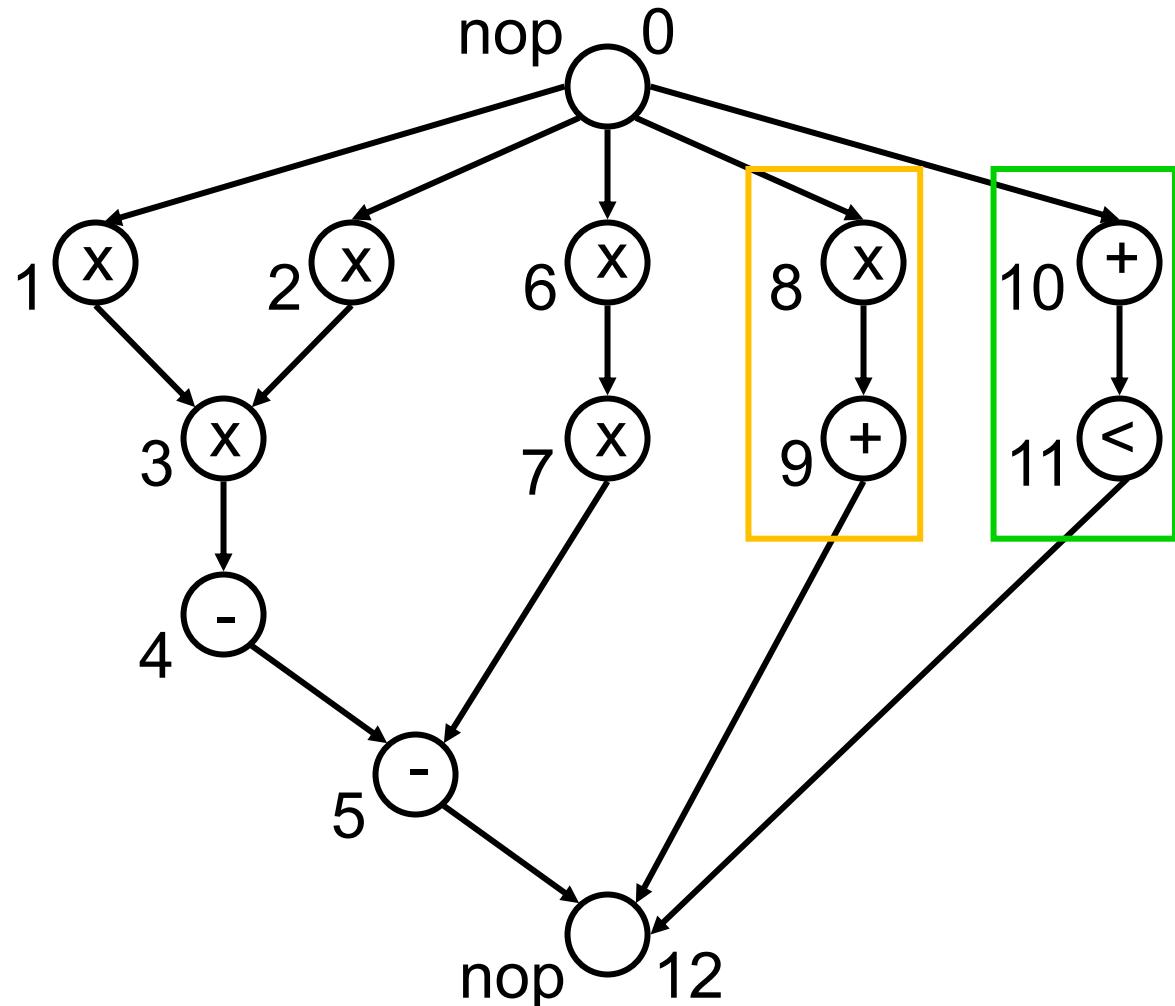


Models for Architecture Synthesis - Example

- Corresponding 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;  
}
```

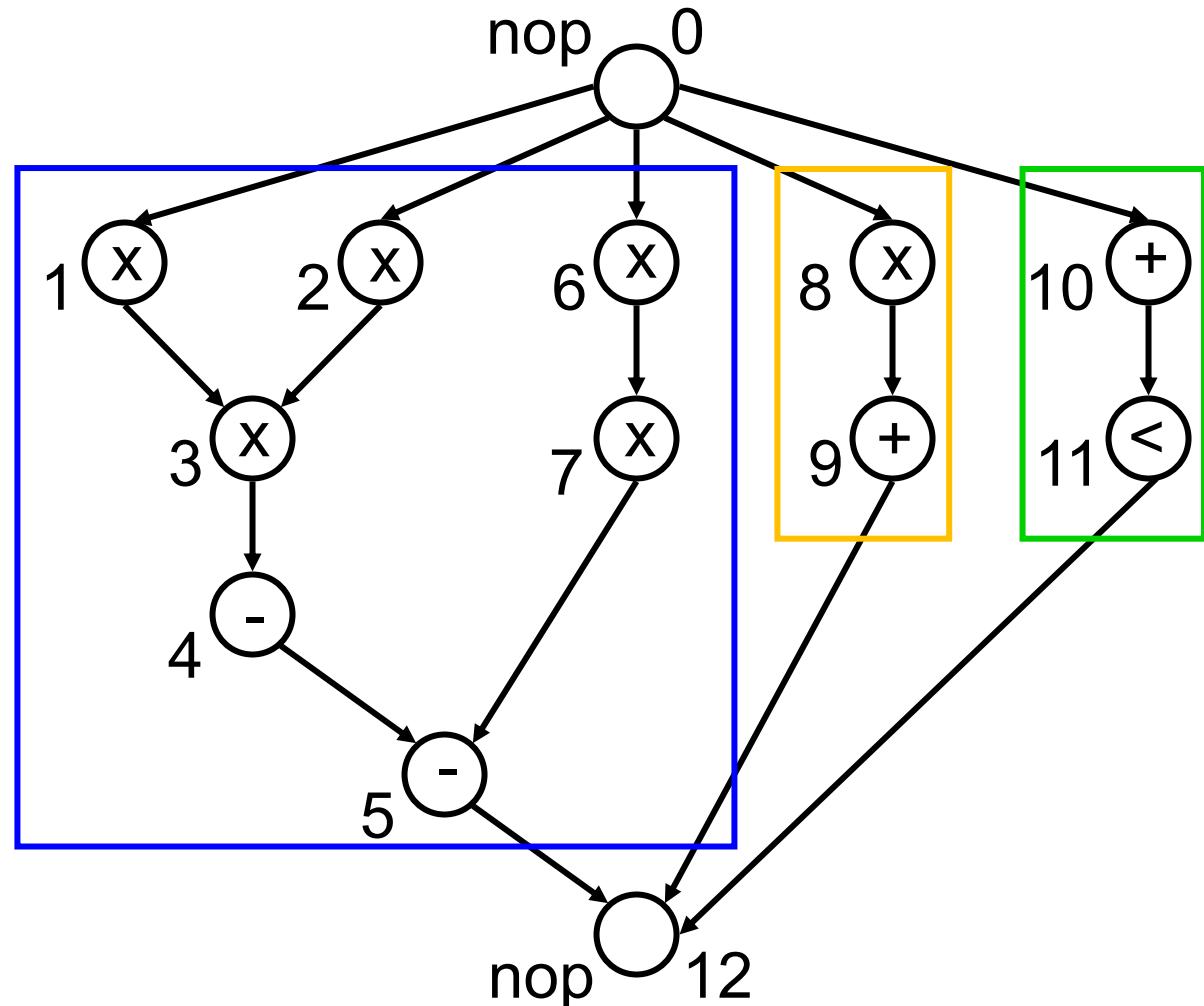


Models for Architecture Synthesis - Example

- Corresponding sequence graph:

```
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 * v * dx);  
        y1 = y + u * dx;  
        x = x1;  
        u = u1;  
        y = y1;  
    }  
    return y;  
}
```

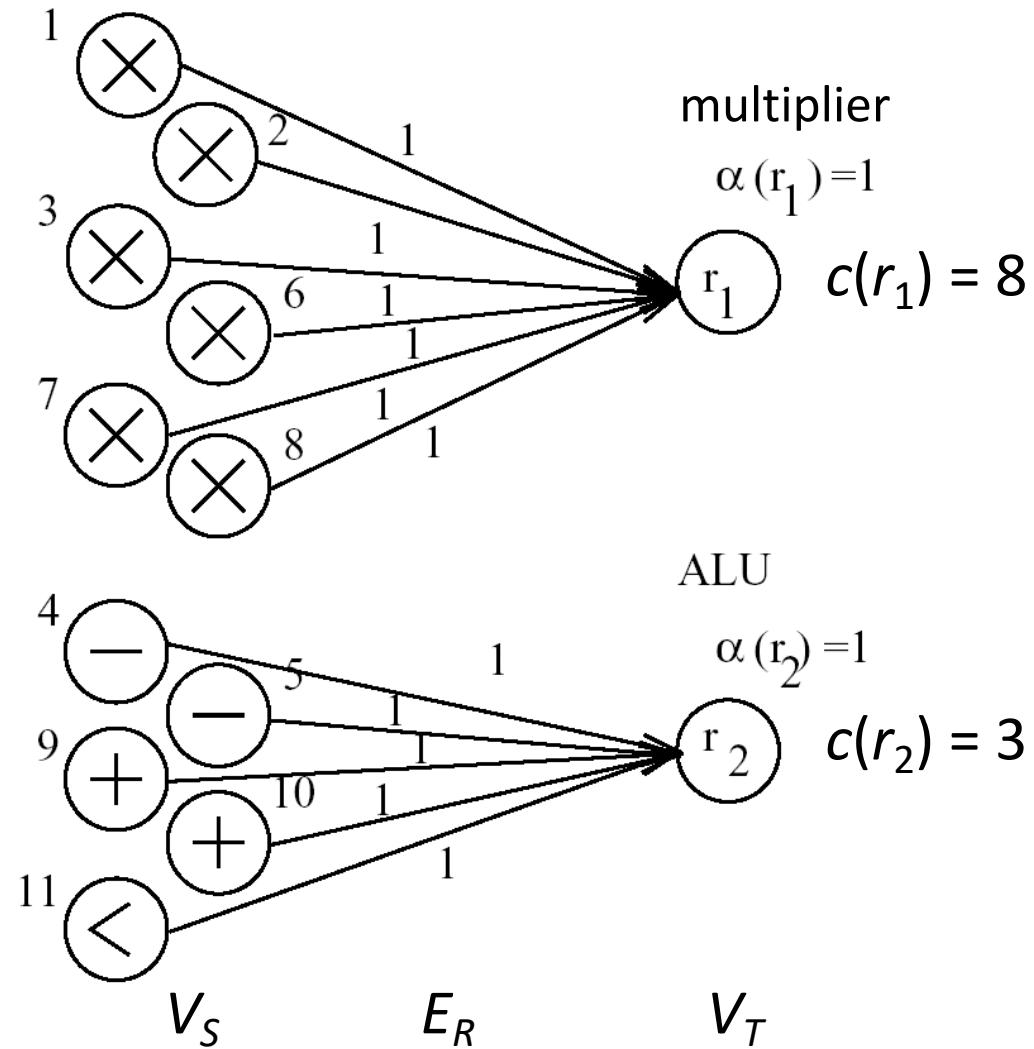
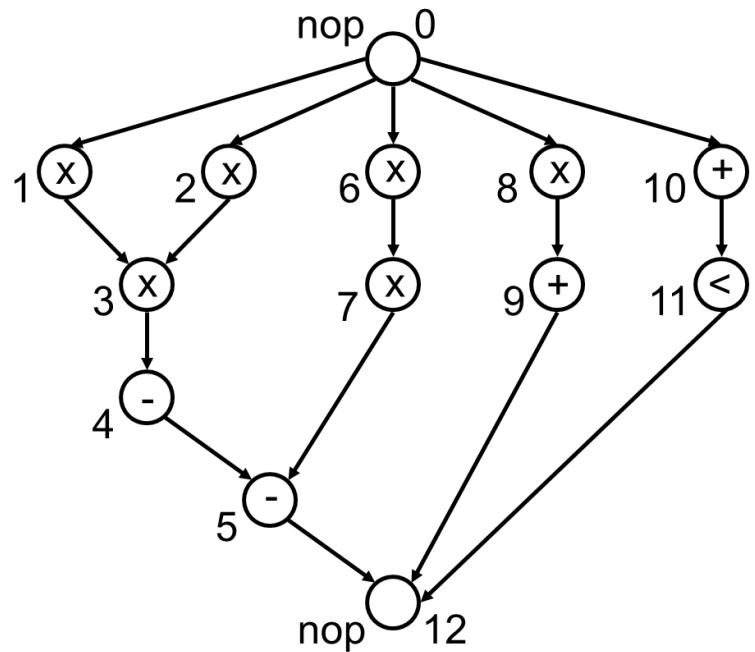
$$G_S = (V_S, E_S)$$



Models for Architecture Synthesis - Example

- *Corresponding resource graph*

with one instance of a multiplier (cost 8) and one instance of an ALU (cost 3):



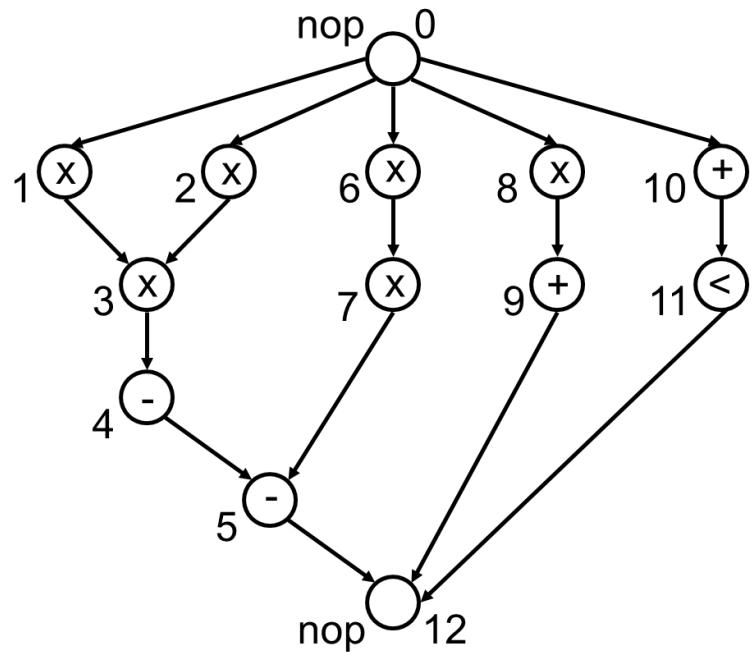
$$G_S = (V_S, E_S)$$

$$G_R = (V_R, E_R), V_R = V_S \cup V_T$$

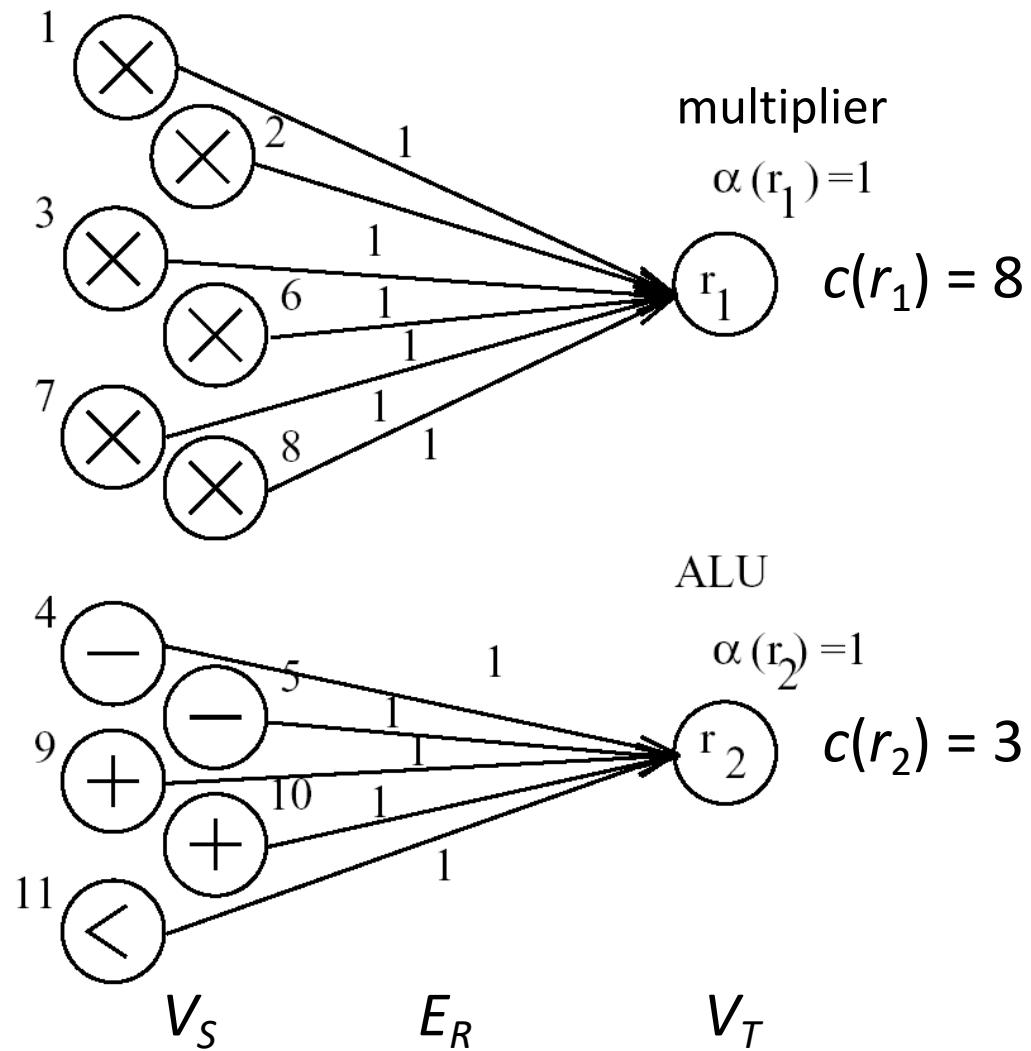
Models for Architecture Synthesis - Example

- *Corresponding resource graph*

with one instance of a multiplier (cost 8) and one instance of an ALU (cost 3):



$$G_S = (V_S, E_S)$$

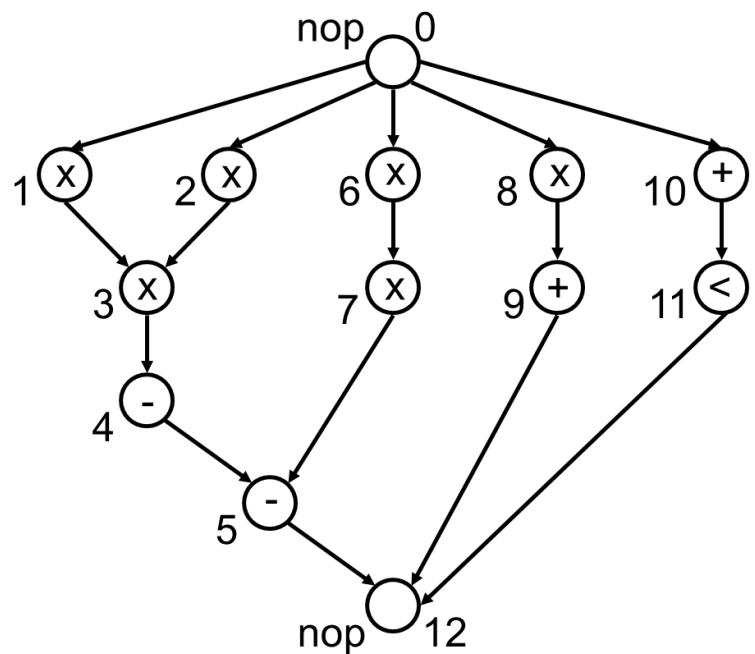


$$G_R = (V_R, E_R), V_R = V_S \cup V_T$$

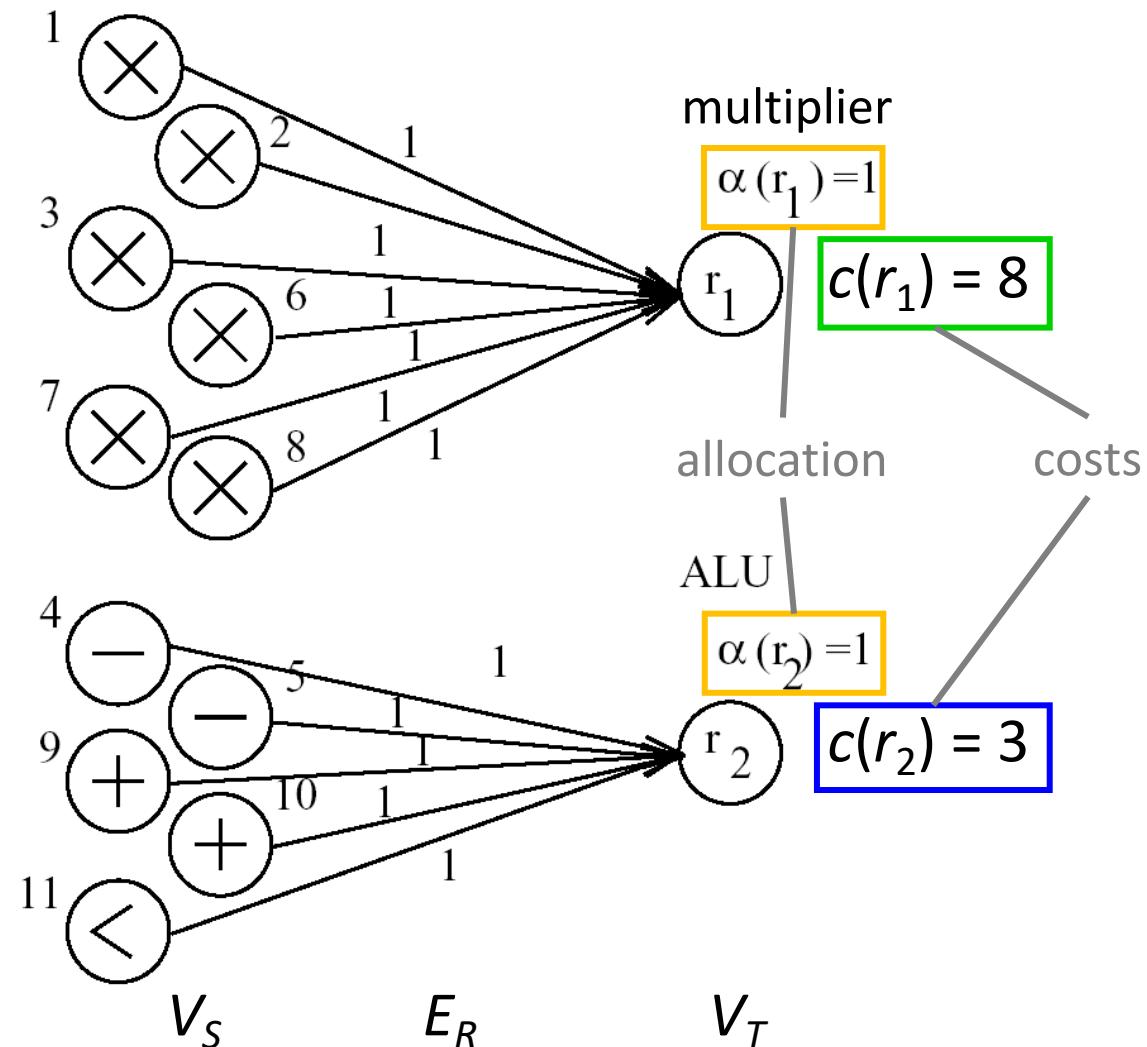
Models for Architecture Synthesis - Example

- Corresponding resource graph

with one instance of a multiplier (cost 8) and one instance of an ALU (cost 3):



$$G_S = (V_S, E_S)$$



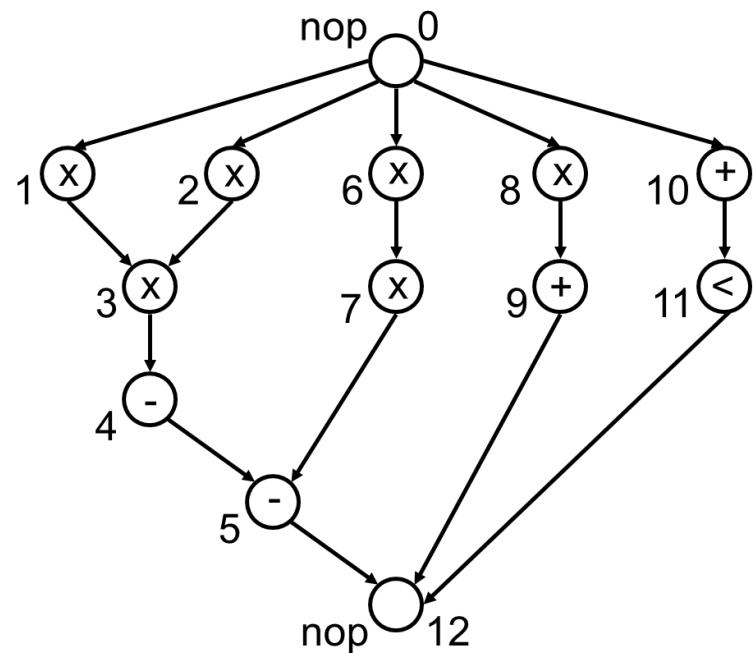
$$G_R = (V_R, E_R), \quad V_R = V_S \cup V_T$$

Models for Architecture Synthesis - Example

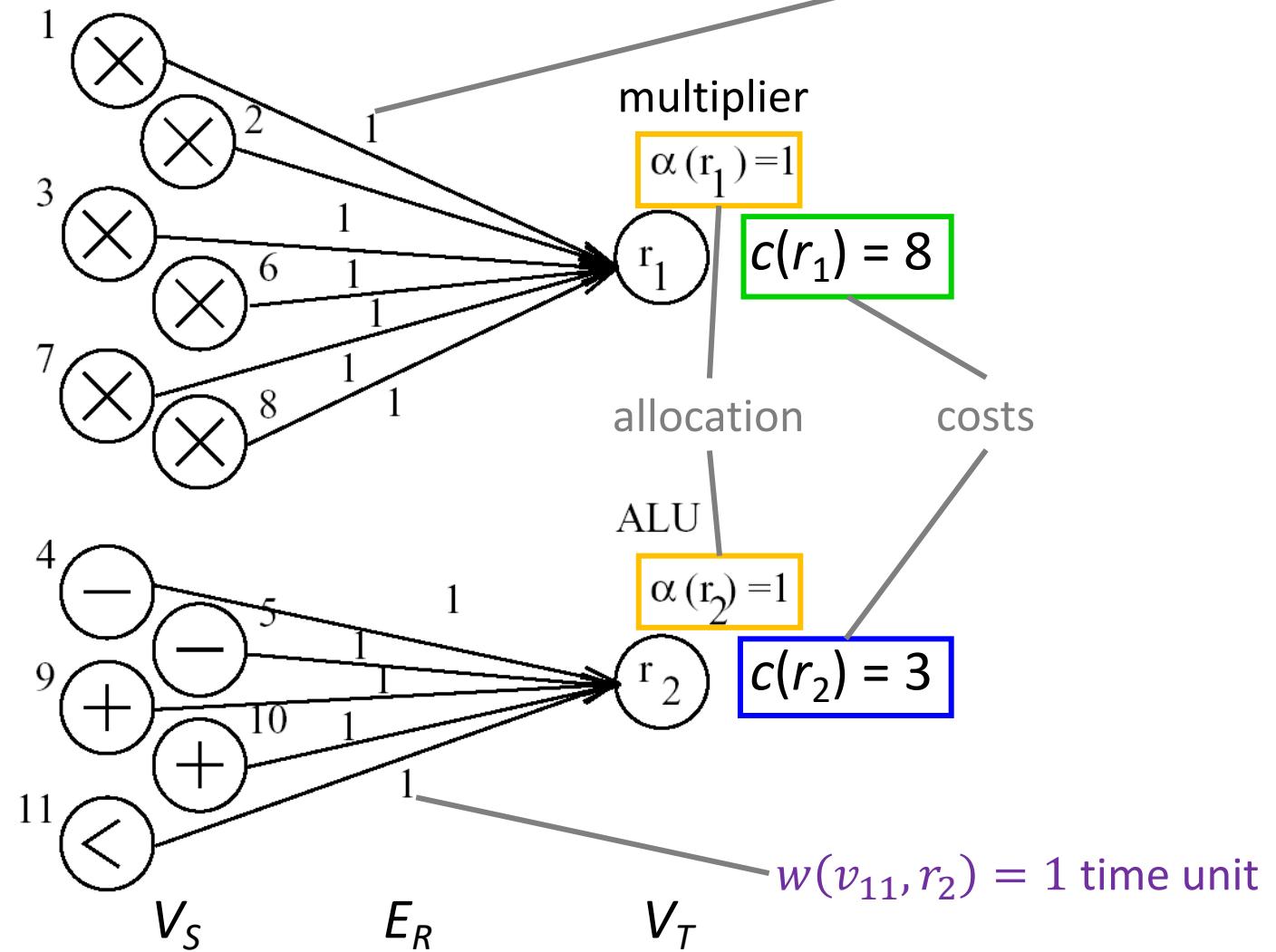
$$w(v_1, r_1) = 1 \text{ time unit}$$

- 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

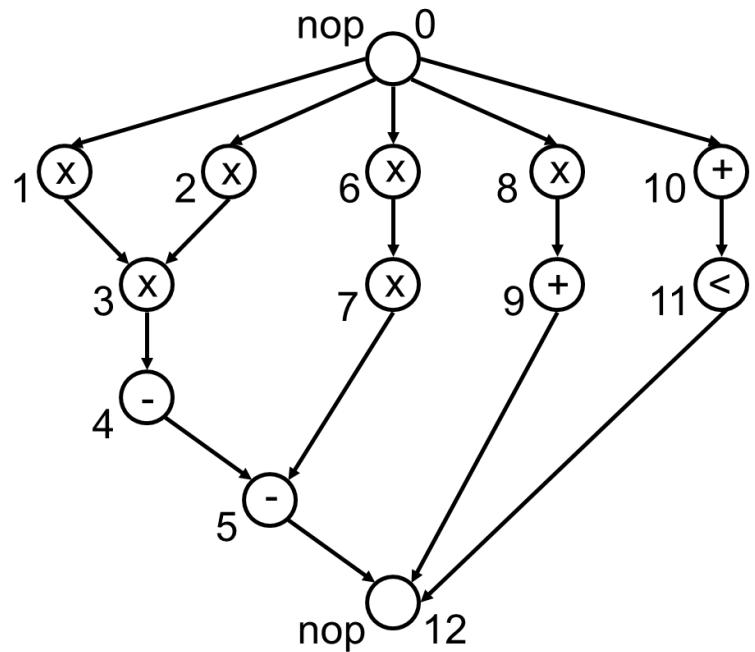
An allocation is a function $\alpha : V_T \rightarrow \mathbf{Z}^{>0}$ that assigns to each resource type $v_t \in V_T$ the number $\alpha(v_t)$ of available instances.

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 $v_t \in V_T$.

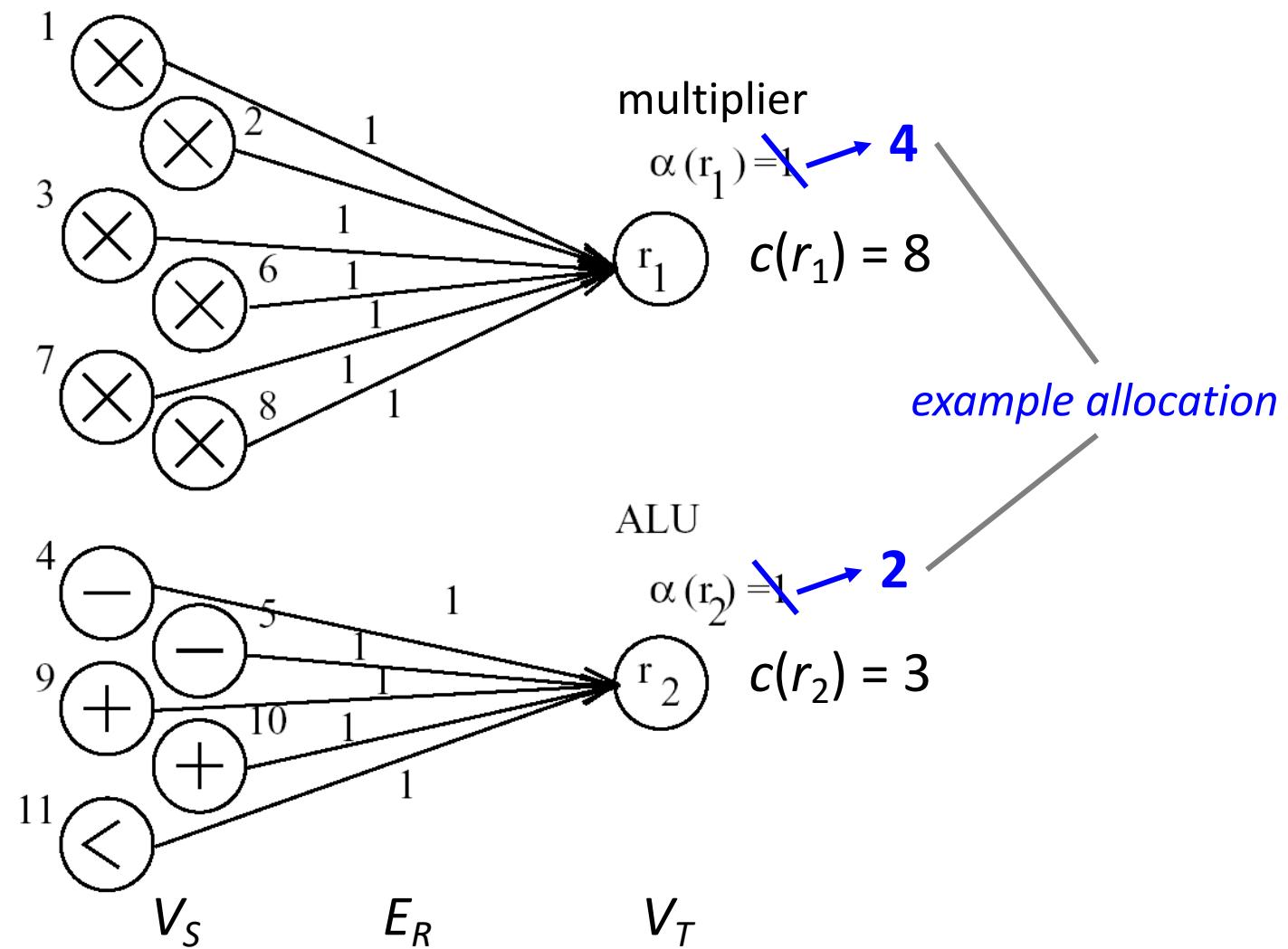
Models for Architecture Synthesis - Example

- *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* for the allocation

$\alpha(r_1) = 4$ multipliers, $\alpha(r_2) = 2$ ALUs:

$$\beta(v_1) = r_1, \gamma(v_1) = 1,$$

$$\beta(v_2) = r_1, \gamma(v_2) = 2,$$

$$\beta(v_3) = r_1, \gamma(v_3) = 2,$$

$$\beta(v_4) = r_2, \gamma(v_4) = 1,$$

$$\beta(v_5) = r_2, \gamma(v_5) = 1,$$

$$\beta(v_6) = r_1, \gamma(v_6) = 3,$$

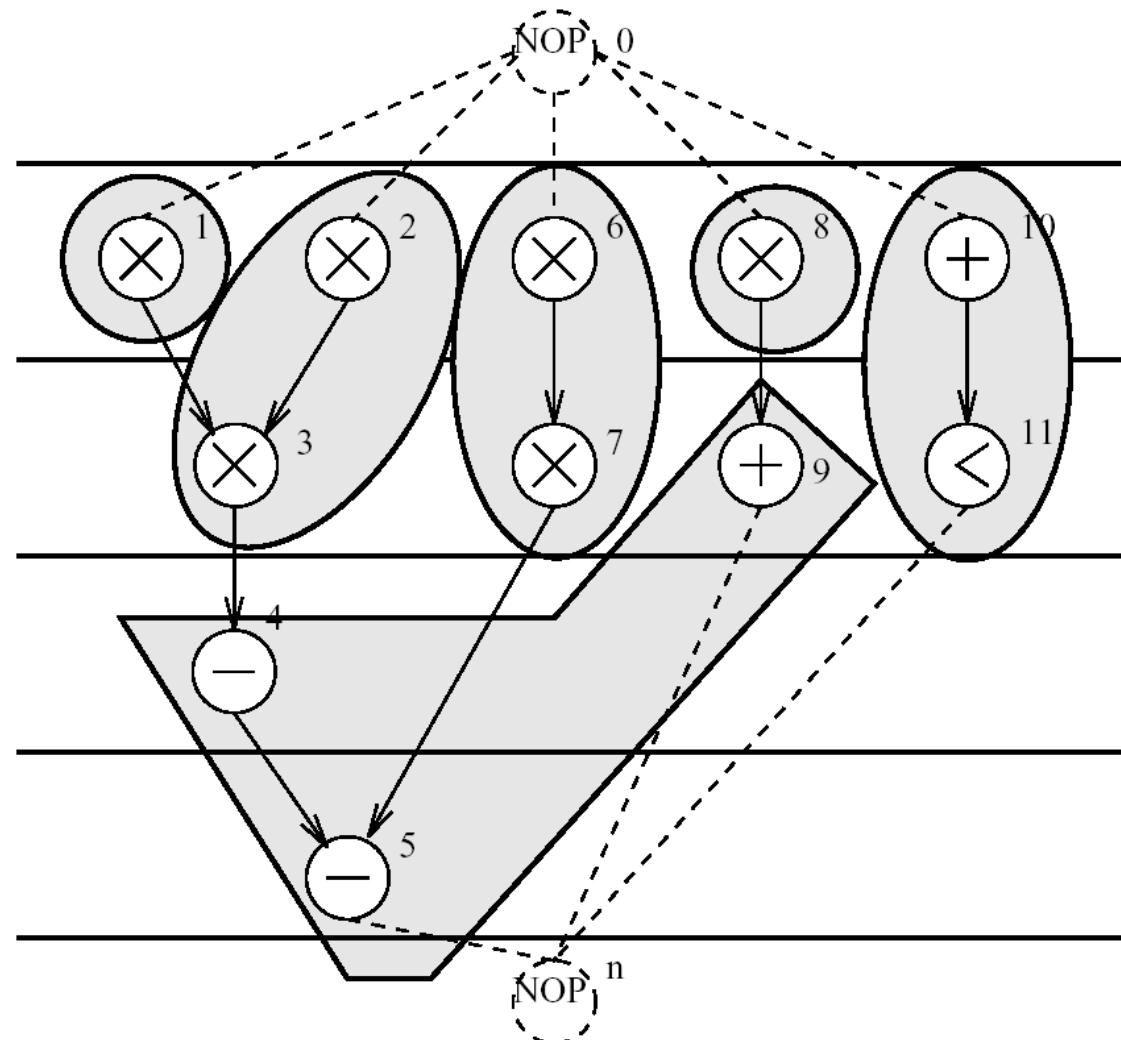
$$\beta(v_7) = r_1, \gamma(v_7) = 3,$$

$$\beta(v_8) = r_1, \gamma(v_8) = 4,$$

$$\beta(v_9) = r_2, \gamma(v_9) = 1,$$

$$\beta(v_{10}) = r_2, \gamma(v_{10}) = 2,$$

$$\beta(v_{11}) = r_2, \gamma(v_{11}) = 2$$



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

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

must hold for *all* edges of the sequence graph

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 :

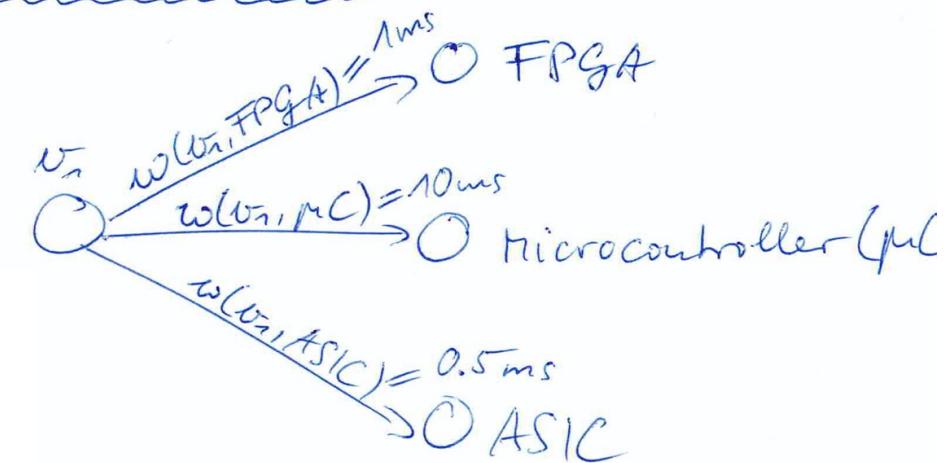
$$L = \tau(v_n) - \tau(v_0)$$

dependence graph:



$$w(v_i) = w(v_i, \beta(v_i))$$

resource graph:

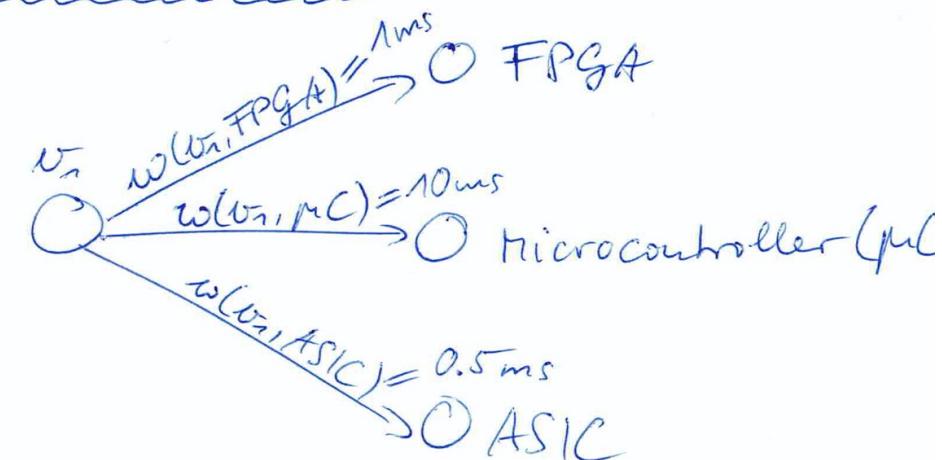


dependence graph:

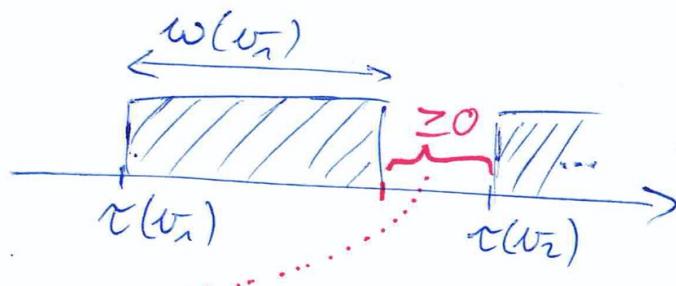


$$w(v_i) = w(v_i, \beta(v_i))$$

resource graph:



scheduling:



$$\Rightarrow \tau(v_2) - \tau(v_1) \geq w(v_1)$$

Models for Architecture Synthesis - Example

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

$$\tau(v_0) = 1$$

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

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

$$\tau(v_3) = 3$$

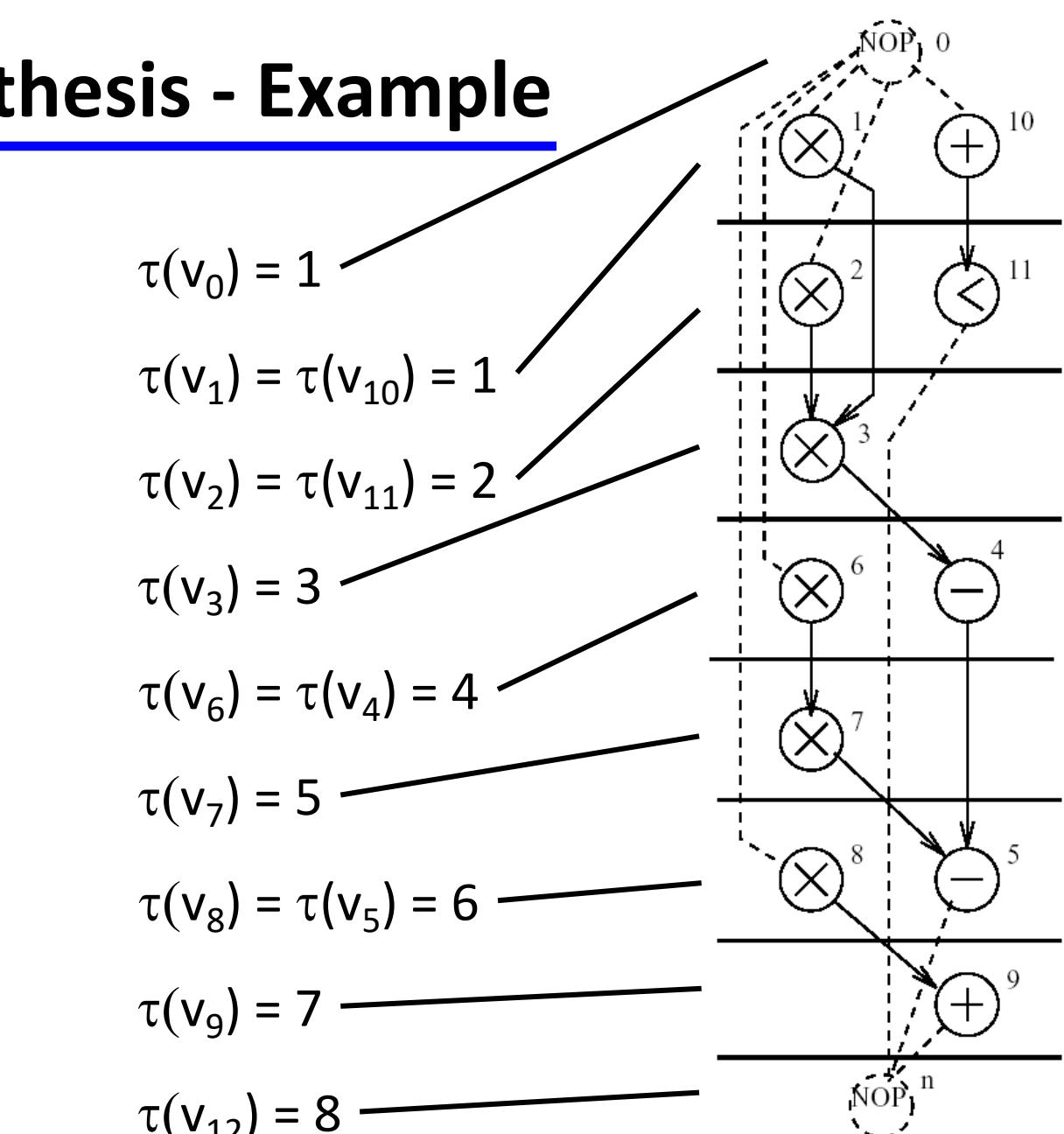
$$\tau(v_6) = \tau(v_4) = 4$$

$$\tau(v_7) = 5$$

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

$$\tau(v_9) = 7$$

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



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

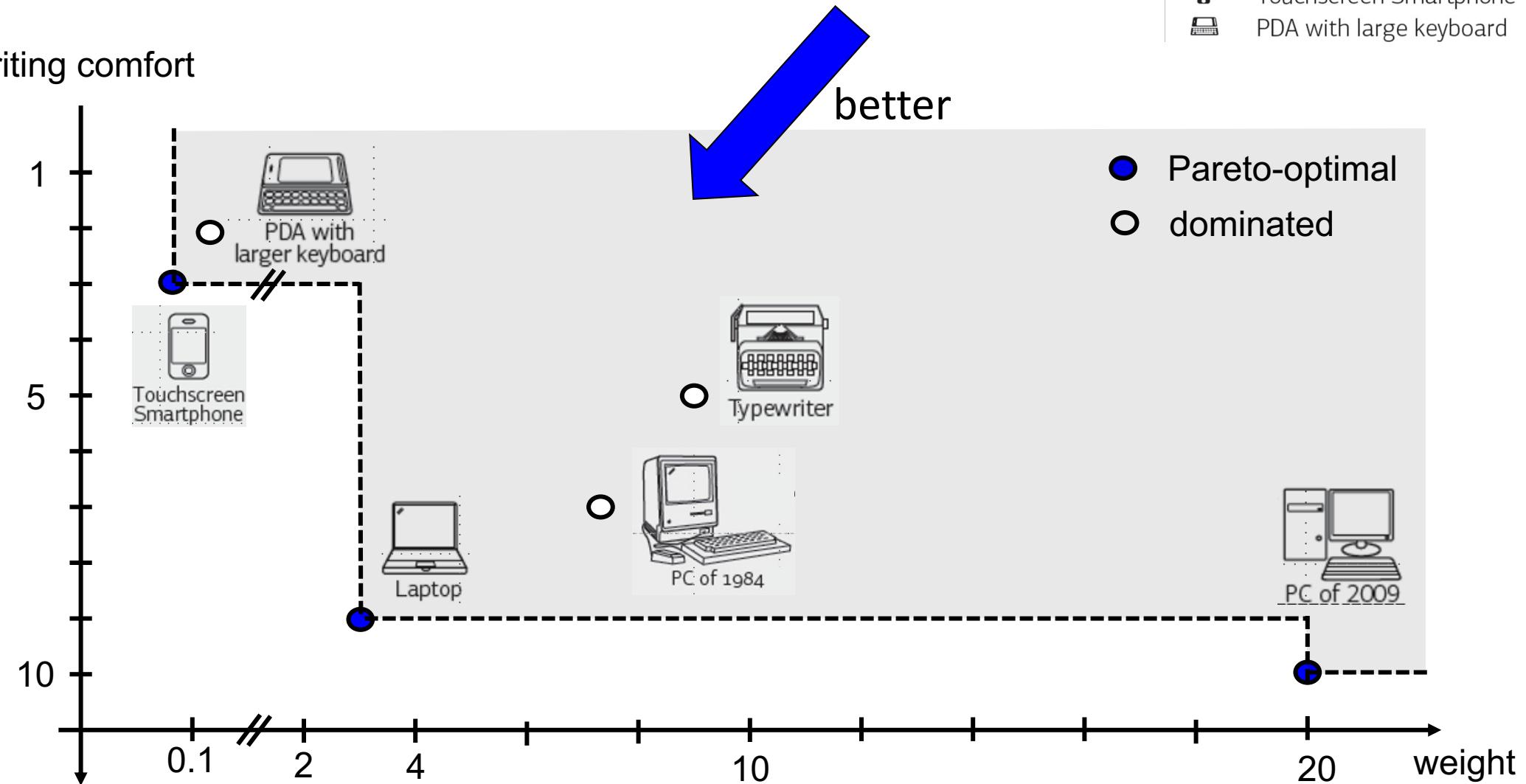
- Let us suppose, we would like to select a typewriting device. Criteria are
 - mobility (related to weight)
 - comfort (related to keyboard size and performance)

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

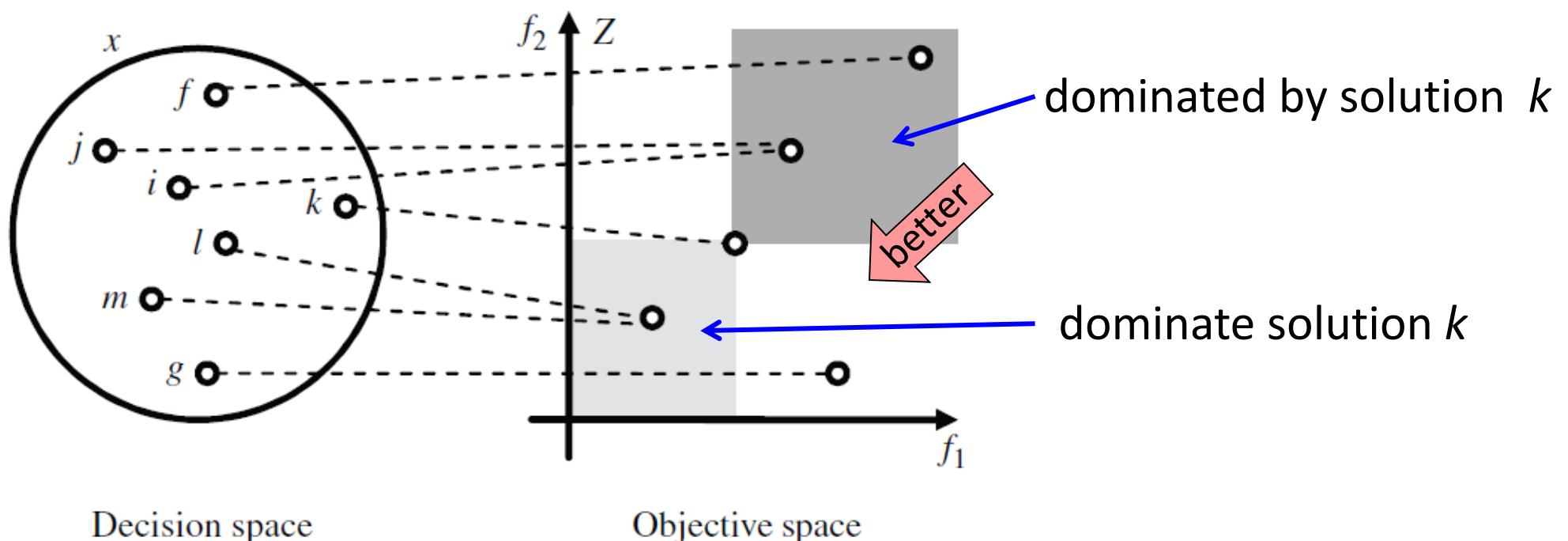
Icon	Device	weight (kg)	comfort rating
PC icon	PC of 2020	20.00	10
Laptop icon	PC of 1984	7.50	7
Laptop icon	Laptop	3.00	9
Typewriter icon	Typewriter	9.00	5
Smartphone icon	Touchscreen Smartphone	0.09	3
PDA icon	PDA with large keyboard	0.11	2

writing comfort



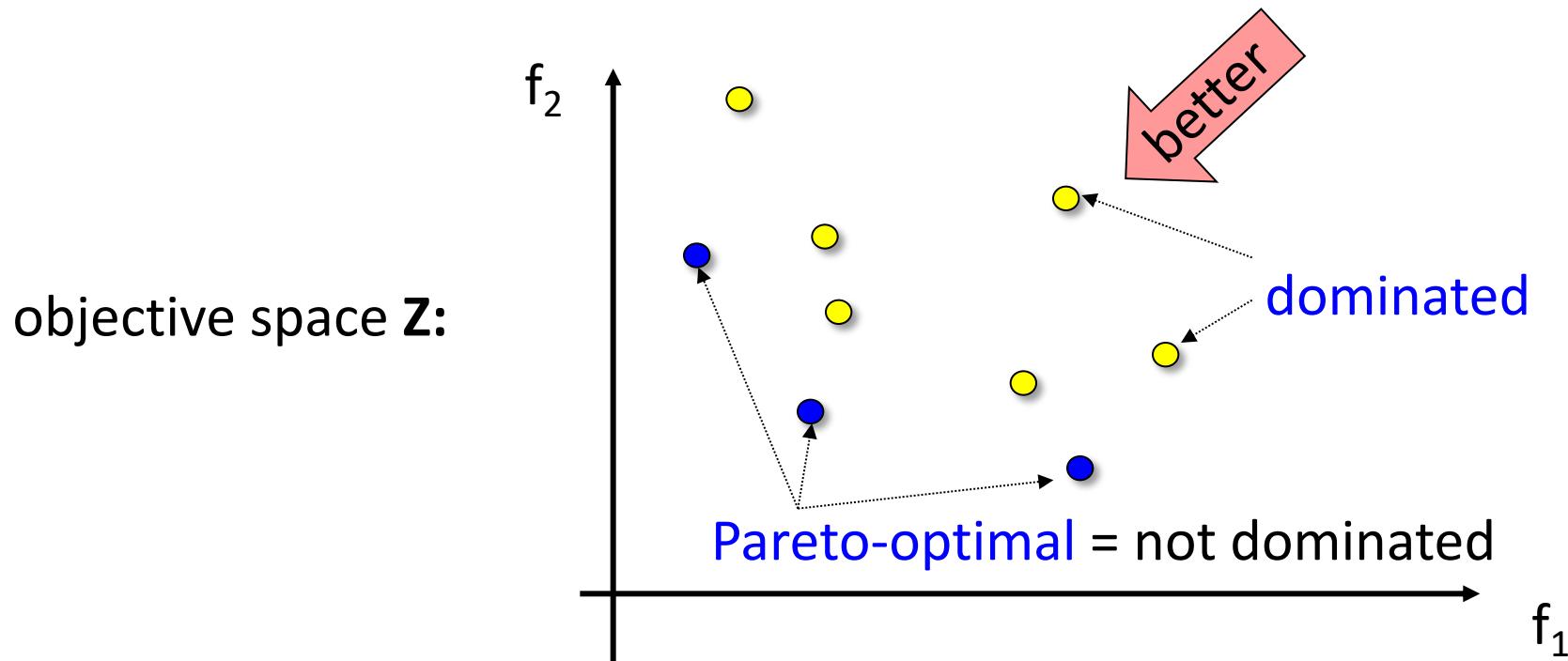
Pareto-Dominance

Definition : A solution $a \in X$ weakly Pareto-dominates a solution $b \in X$, denoted as $a \preceq b$, if it is at 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 than b , denoted as $a \prec b$, iff $(a \preceq b) \wedge (b \not\preceq a)$.



Pareto-optimal Set

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



Architecture Synthesis without Resource Constraints

Synthesis Algorithms

Classification

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

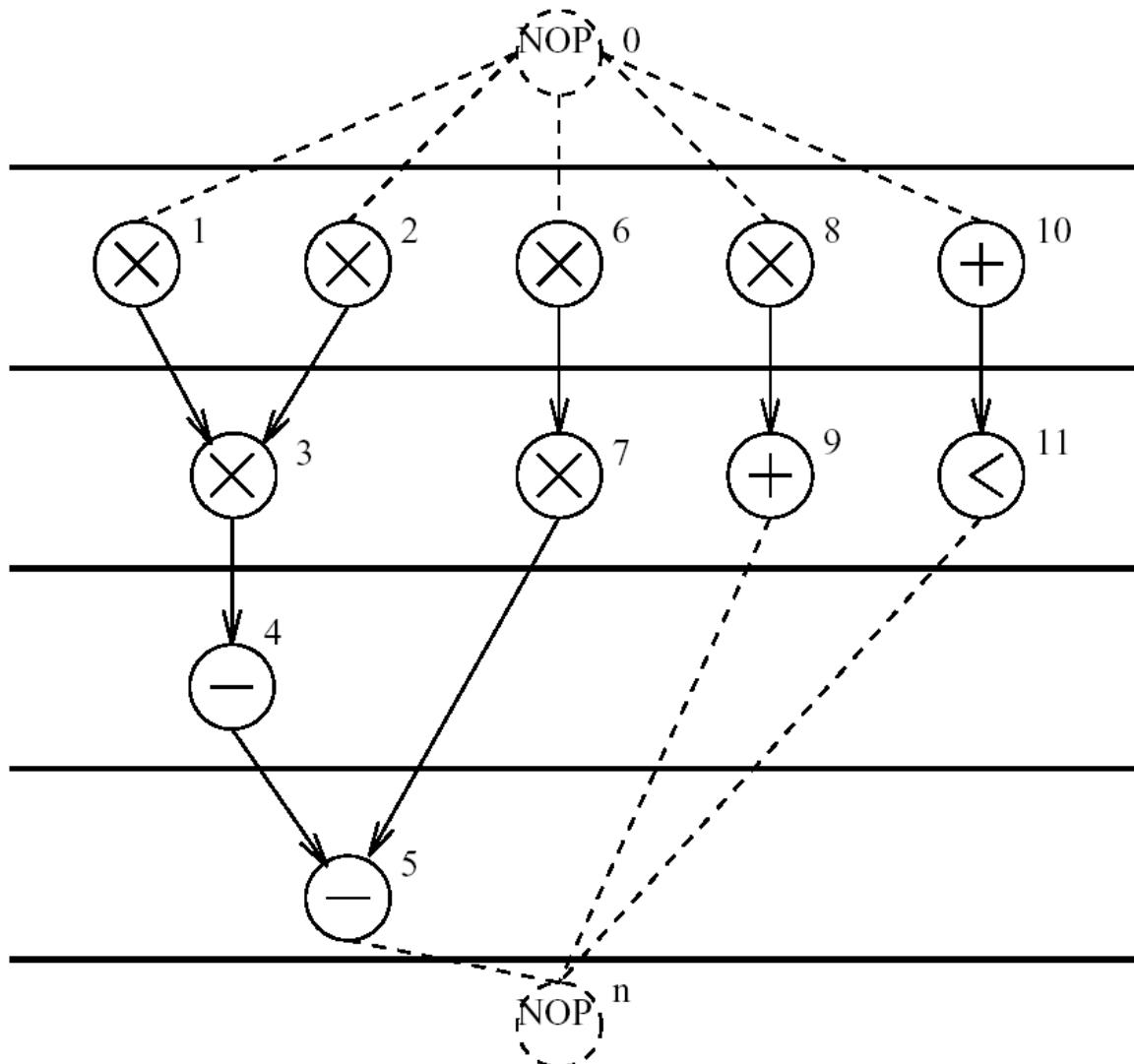
ASAP = As Soon As Possible

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

The ASAP Algorithm - Example

Example:

$$w(v_i) = 1$$



ALAP Algorithm

ALAP = As Late As Possible

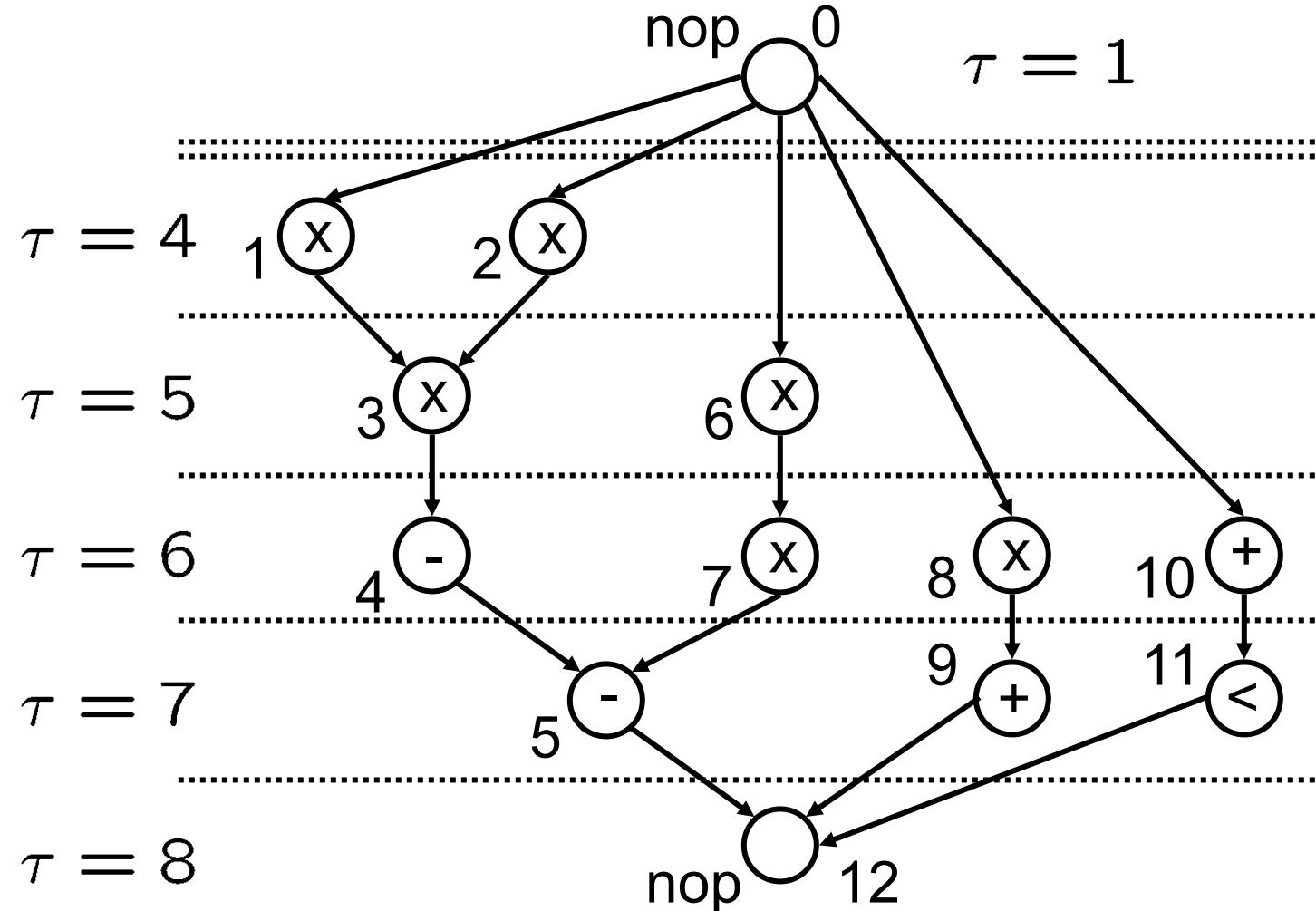
```
ALAP( $G_S(V_S, E_S), w, L_{max}$ ) {  
     $\tau(v_n) = L_{max} + 1;$   
    REPEAT {  
        Determine  $v_i$  whose succ. are planned;  
         $\tau(v_i) = \min\{\tau(v_j) \forall (v_i, v_j) \in E_S\} - w(v_i)$   
    } UNTIL ( $v_0$  is planned);  
    RETURN ( $\tau$ );  
}
```

ALAP Algorithm - Example

Example:

$$L_{\max} = 7$$

$$w(v_i) = 1$$



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 α and binding β .

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 .

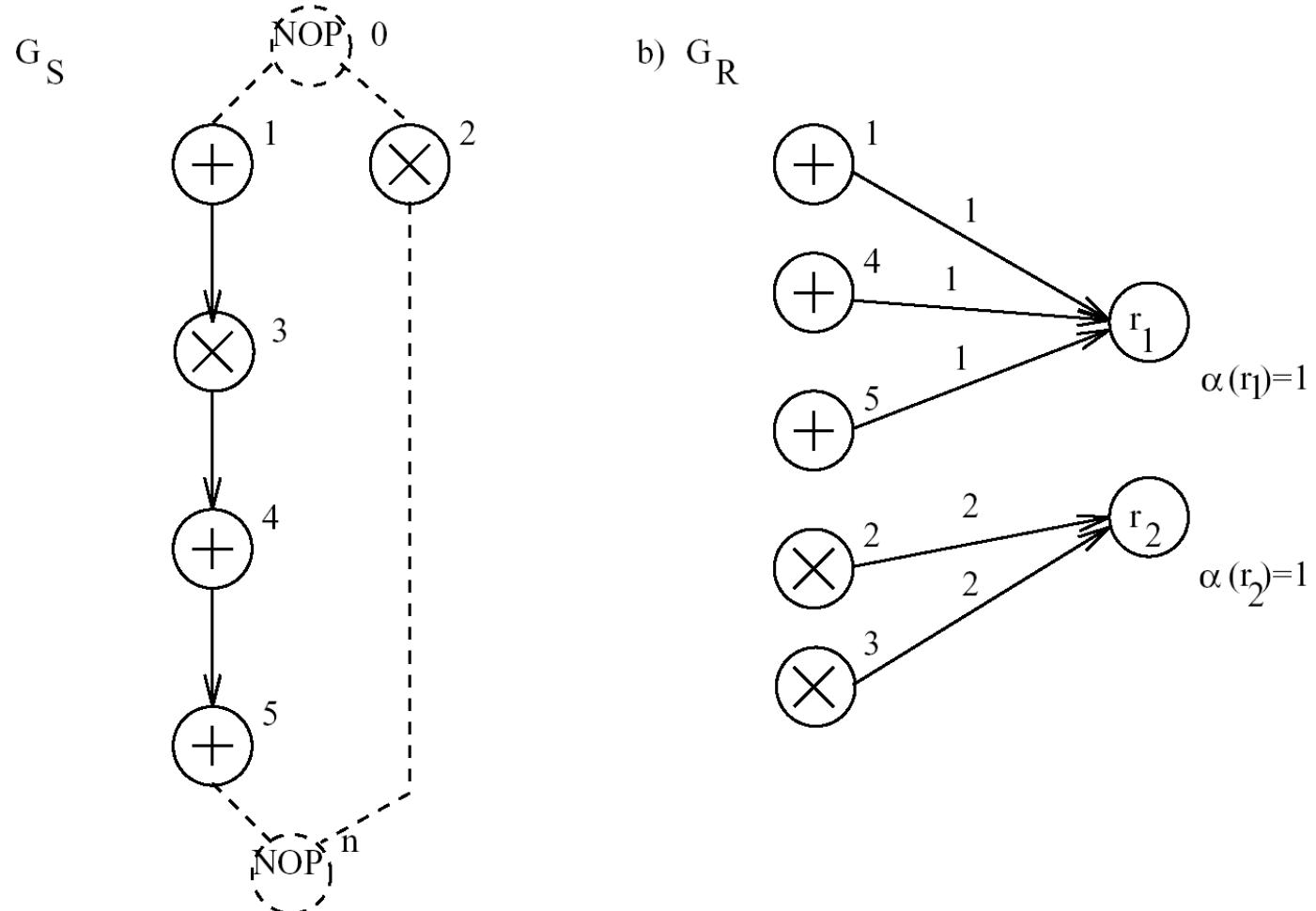
List Scheduling

```
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  $|S_k| + |T_k| \leq \alpha(v_k)$ ;  
             $\tau(v_i) = t \quad \forall v_i \in S_k; \quad }$   
         $t = t + 1;$   
    } UNTIL ( $v_n$  planned)  
    RETURN ( $\tau$ ); }
```

List Scheduling - Example

Example:

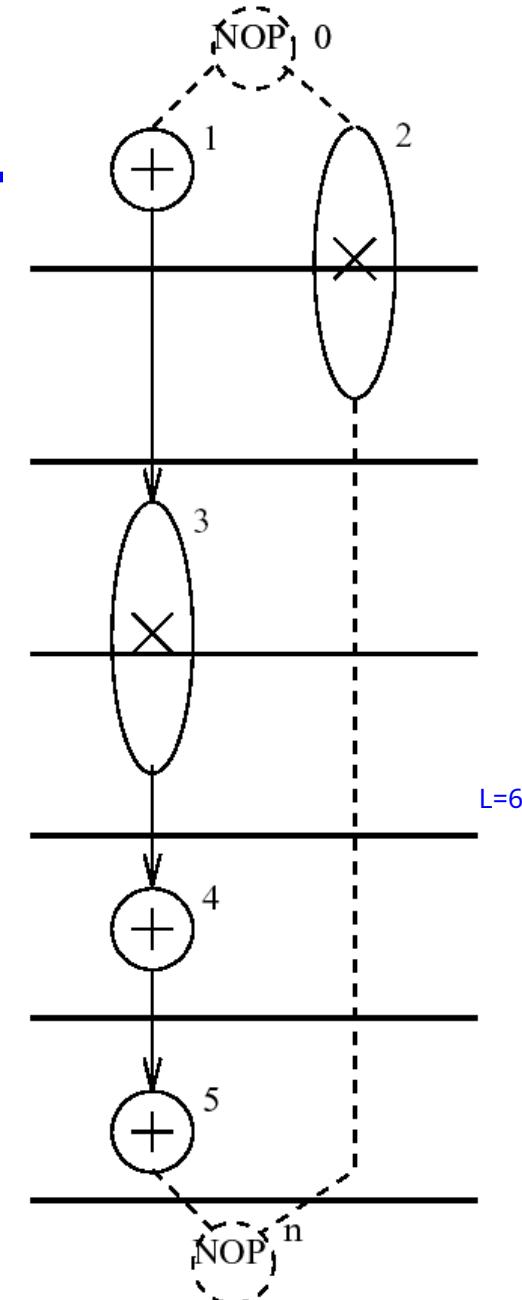
```
LIST( $G_S(V_S, E_S), G_R(V_R, E_R), \alpha, \beta, priorities$ ){\  
     $t = 1;$   
    REPEAT {  
        FORALL  $v_k \in V_T$  {  
            determine candidates to be scheduled  $U_k$ ;  
            determine running operations  $T_k$ ;  
            choose  $S_k \subseteq U_k$  with maximal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



List Scheduling - Example

Solution via list scheduling:

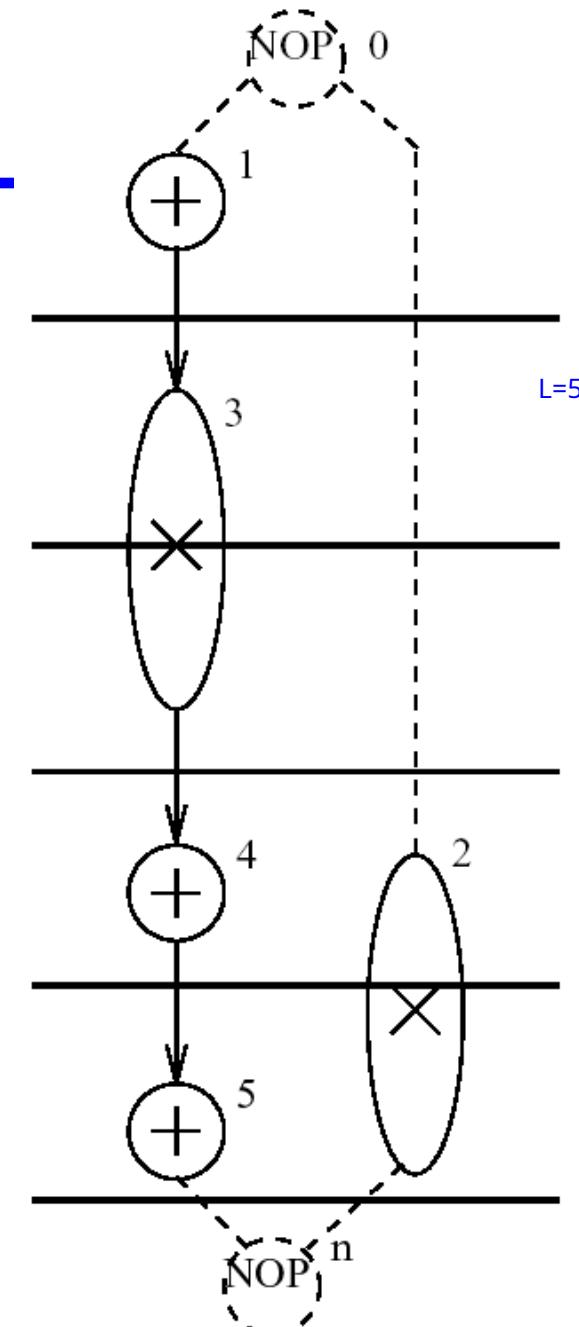
- In the example, the solution is independent of the chosen priority function.
- Because of the greedy selection principle, all resource are occupied in the first time step.
- List scheduling is a heuristic algorithm:
In this example, it does not yield the minimal latency!



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.



Introduction to Embedded Systems

10. Architecture Synthesis

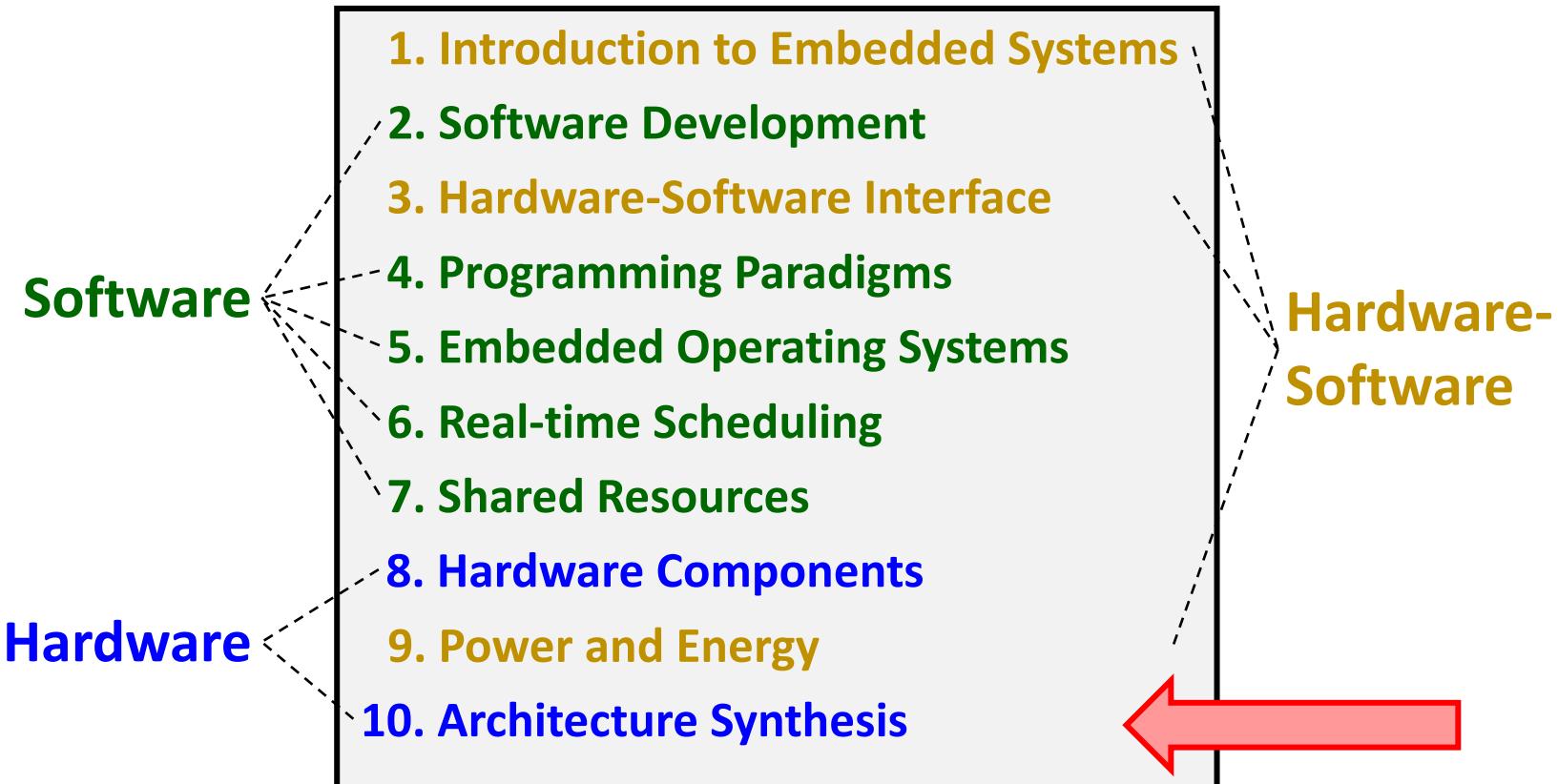
Prof. Dr. Marco Zimmerling

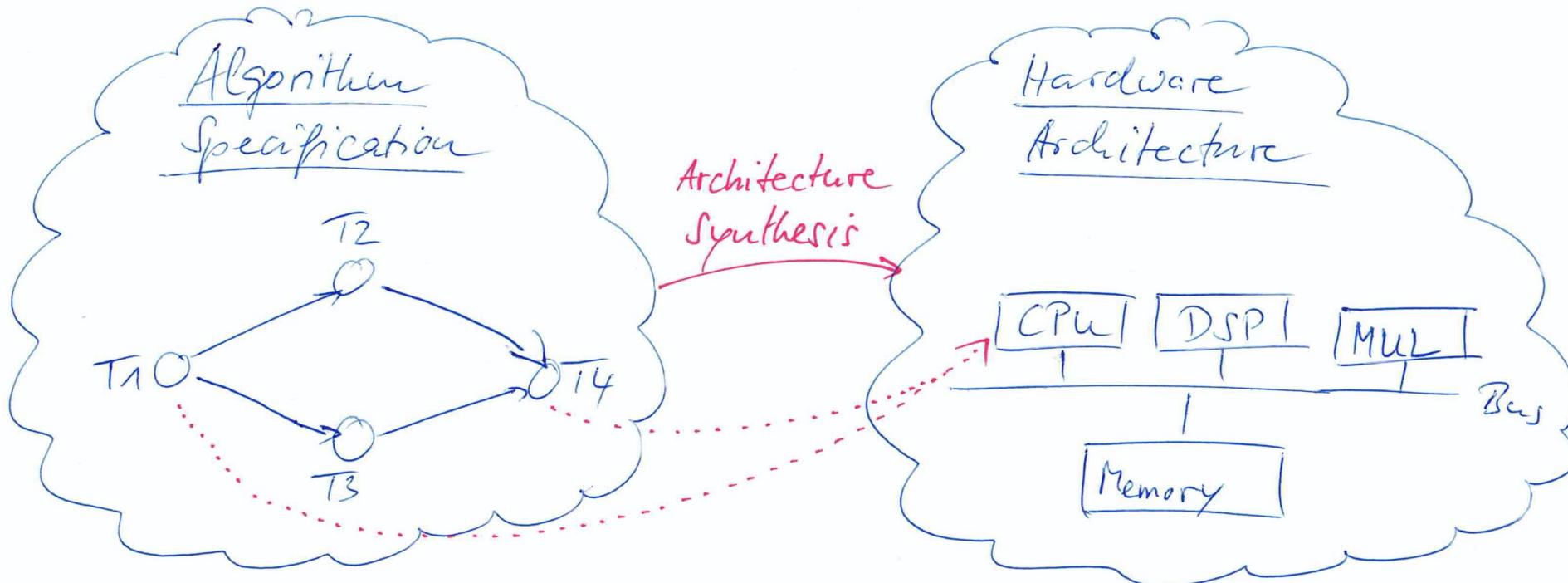


Organization

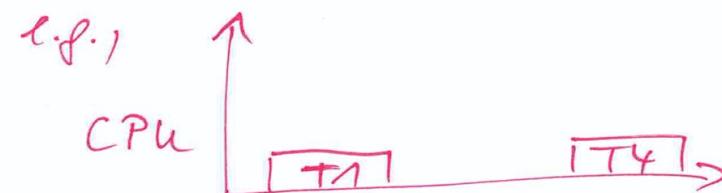
- Today (January 31):
 - Last lecture
 - Exercise 8
 - *Mock-up exam questions released on ILIAS by Thursday*
- Next week (February 7):
 - No lecture
 - *Mock-up exam discussed in exercise sessions*
 - Sample solutions to mock-up exam will be available on ILIAS
- Saturday, March 4, 10-12 AM:
 - *Written exam in building 101*

Lecture Overview





1. Allocation: Which components? e.g., CPU
2. Binding: Which tasks are executed on which components? e.g., $T1 \& T4$ on CPU
3. Scheduling: In what order / when should tasks be executed?



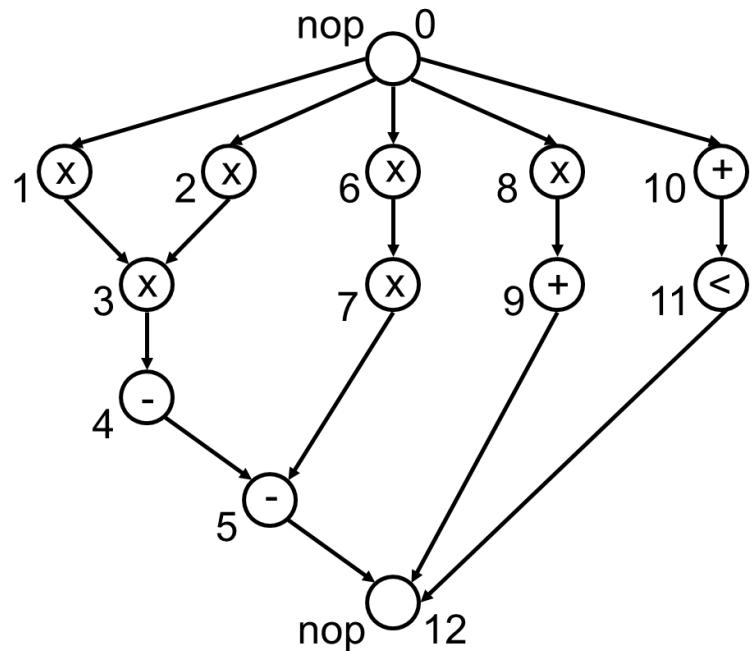
Models for Architecture Synthesis

- *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 $v_t \in V_T$.

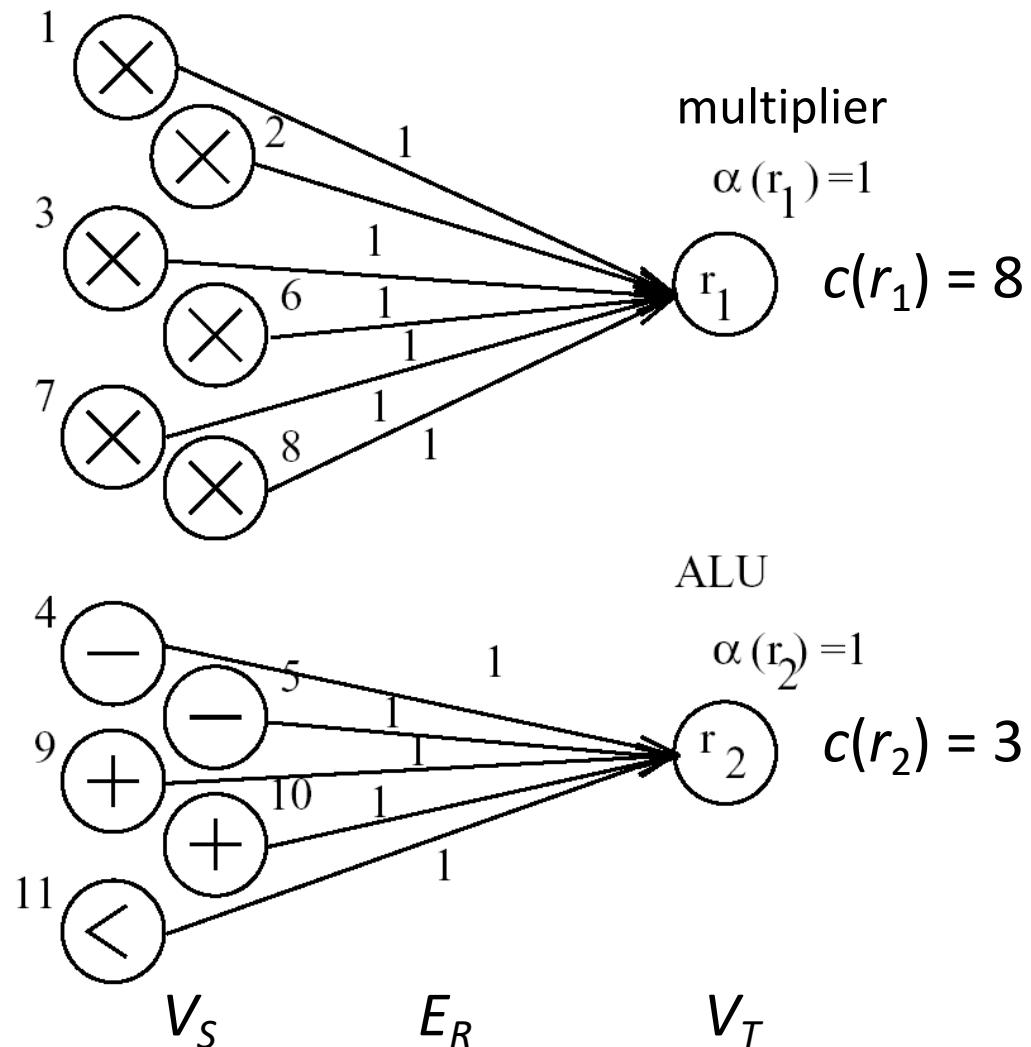
Models for Architecture Synthesis - Example

- *Corresponding resource graph*

with one instance of a multiplier (cost 8) and one instance of an ALU (cost 3):



$$G_S = (V_S, E_S)$$

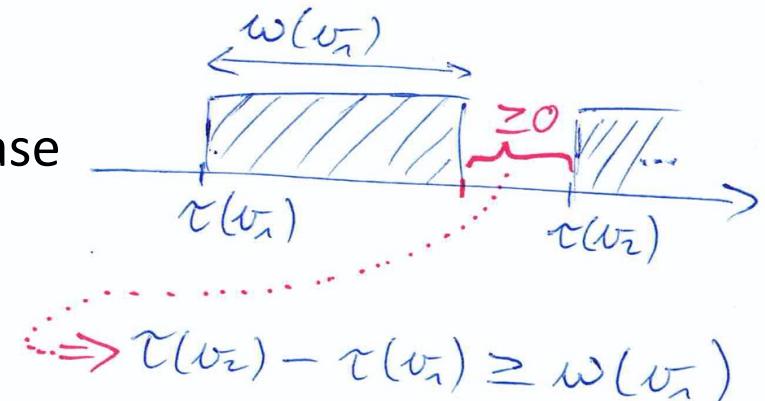


$$G_R = (V_R, E_R), V_R = V_S \cup V_T$$

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

$$L = \min\{\tau(v_n) - \tau(v_0) : \boxed{\tau(v_j) - \tau(v_i) \geq w(v_i, \beta(v_i))} \forall (v_i, v_j) \in E_S\}$$

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 planned;  
         $\tau(v_i) = \max\{\tau(v_j) + w(v_j) \ \forall (v_j, v_i) \in E_S\}$   
    } UNTIL ( $v_n$  is planned);  
    RETURN ( $\tau$ );  
}
```

ALAP Algorithm

ALAP = As Late As Possible

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

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:*

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

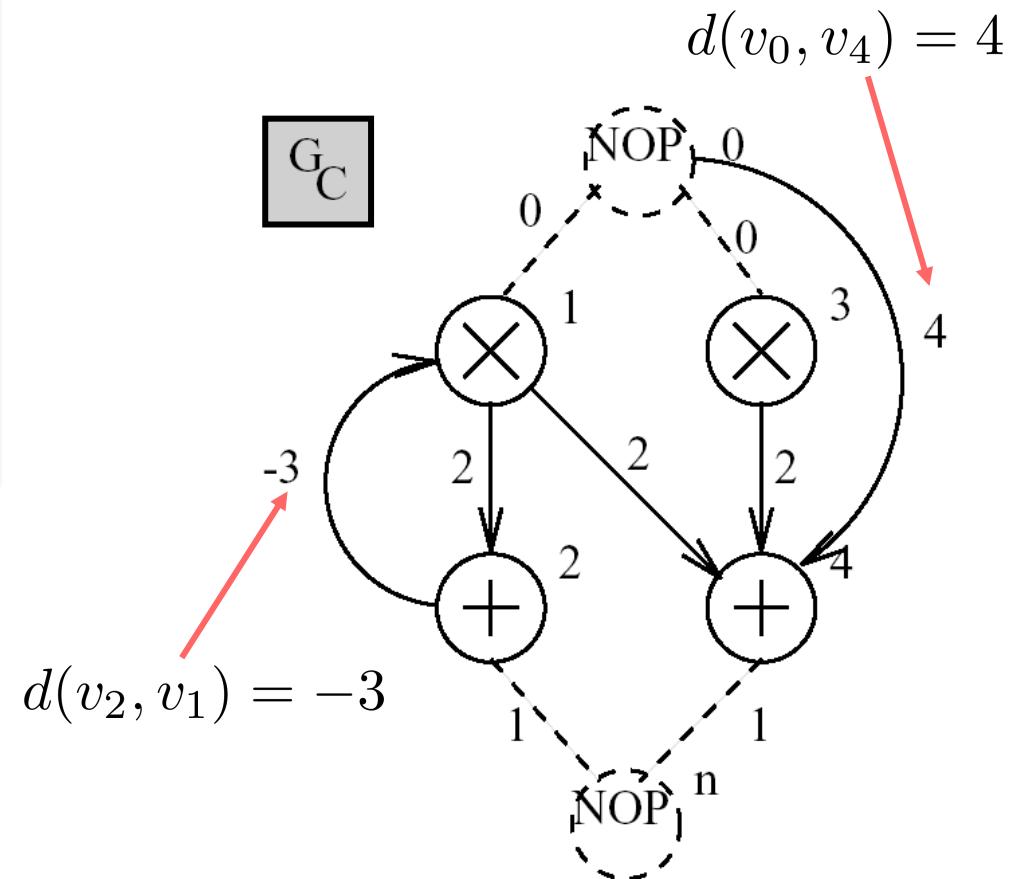
- *Equality constraint:*

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

Weighted Constraint Graph

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

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



Weighted Constraint Graph

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

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

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

- A consistent assignment of starting times $\tau(v_i)$ to all operations can be done by solving a single source longest path problem.
- A possible algorithm (Bellman-Ford) has complexity $O(|V_C| |E_C|)$ ("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 $v_i \in V_C \setminus \{v_0\}$ and $\tau(v_0) = 1$.

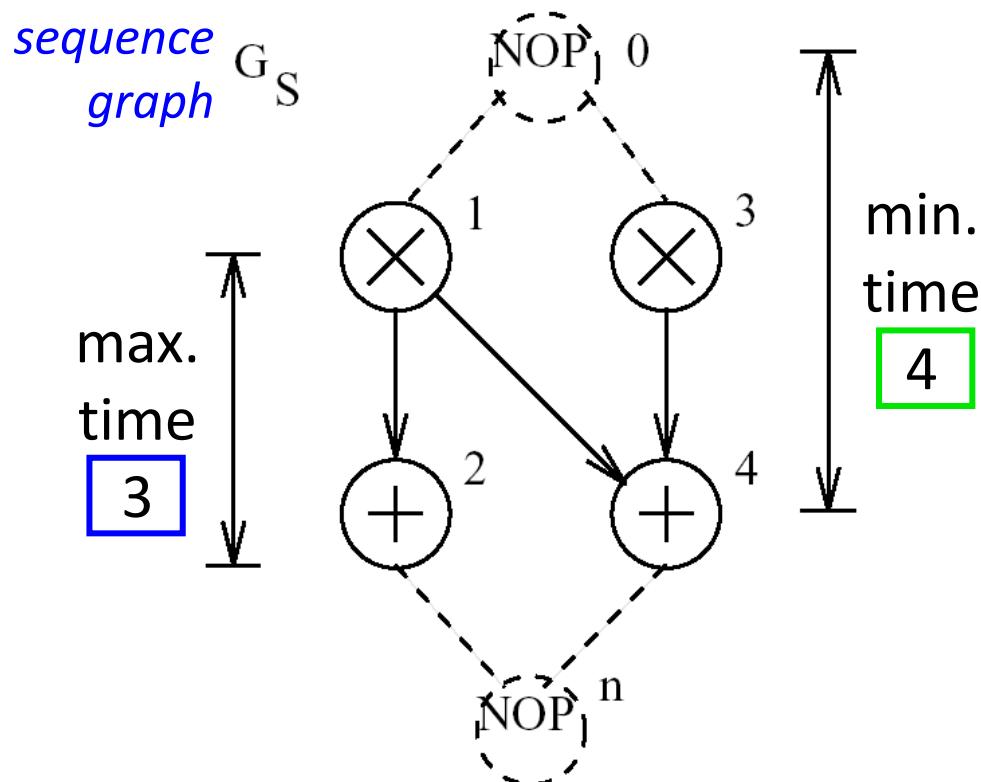
Weighted Constraint Graph - Example

Example:

$$w(v_1) = w(v_3) = 2 \quad w(v_2) = w(v_4) = 1$$

$$\tau(v_0) = \tau(v_1) = \tau(v_3) = 1, \tau(v_2) = 3,$$

$$\tau(v_4) = 5, \tau(v_n) = 6, L = \tau(v_n) - \tau(v_0) = 5$$



Weighted Constraint Graph - Example

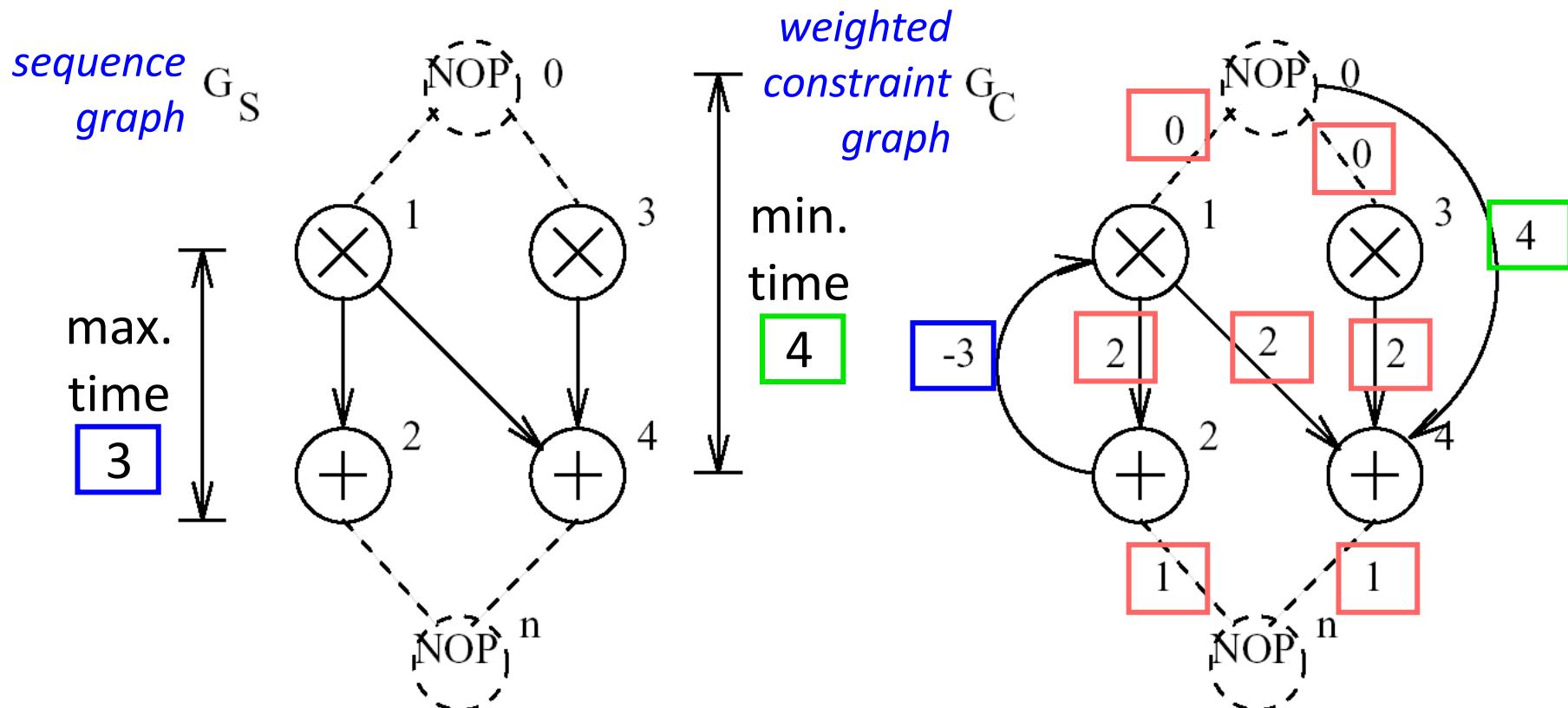
Example:

$$w(v_1) = w(v_3) = 2$$

$$w(v_2) = w(v_4) = 1$$

$$\tau(v_0) = \tau(v_1) = \tau(v_3) = 1, \tau(v_2) = 3,$$

$$\tau(v_4) = 5, \tau(v_n) = 6, L = \tau(v_n) - \tau(v_0) = 5$$



Architecture Synthesis with Resource Constraints

List Scheduling

List scheduling is one of the most widely used algorithms for scheduling under resource constraints.

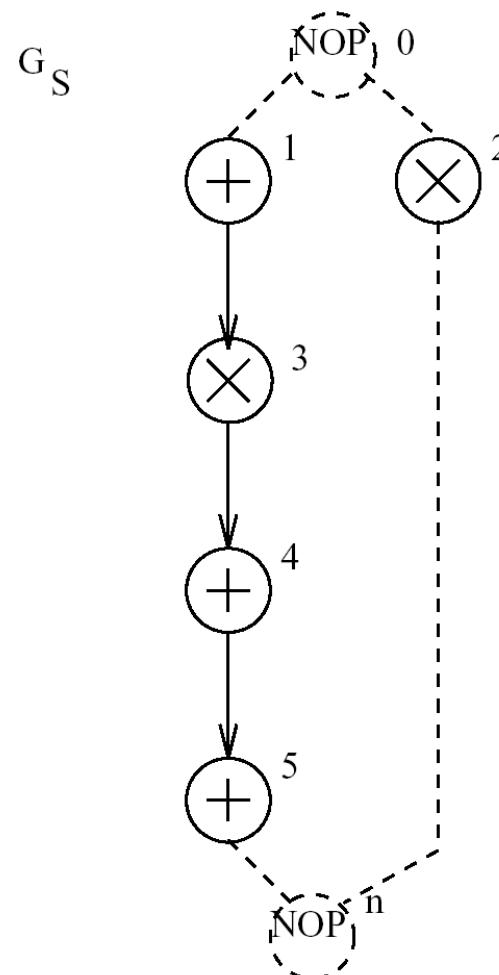
Principles:

- 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 .

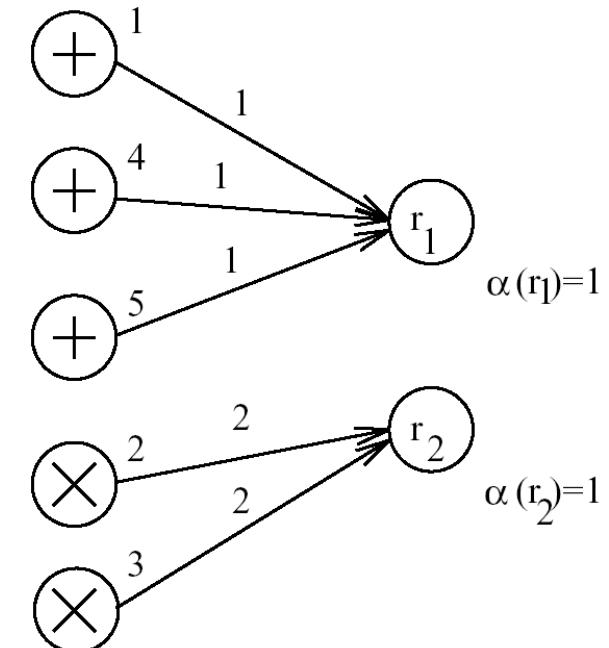
List Scheduling - Example

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  $|S_k| + |T_k| \leq \alpha(v_k)$ ;  
             $\tau(v_i) = t \quad \forall v_i \in S_k; \quad }$   
         $t = t + 1;$   
    } UNTIL ( $v_n$  planned)  
    RETURN ( $\tau$ ); }
```



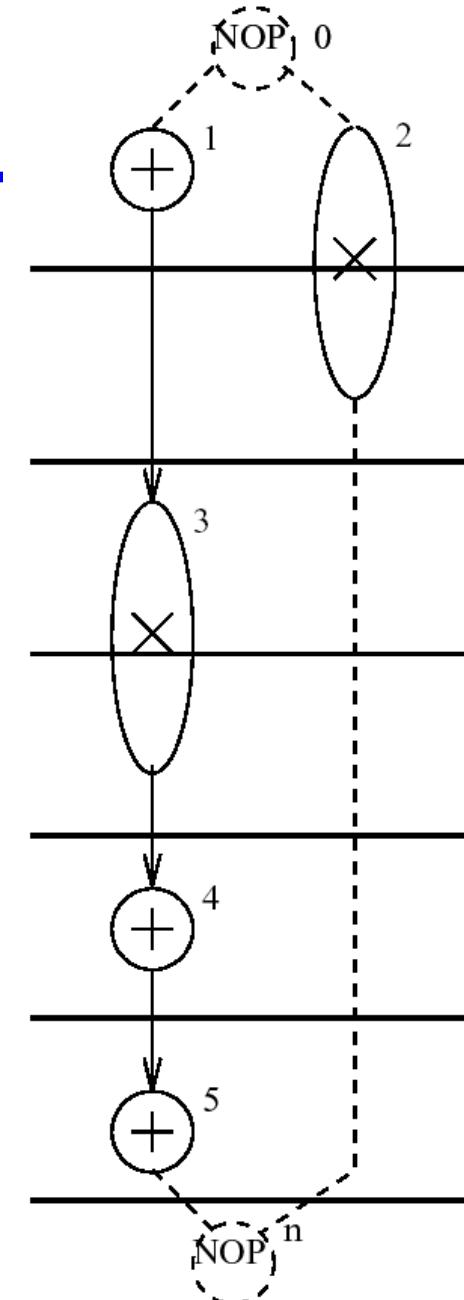
b) G_R



List Scheduling - Example

Solution via list scheduling:

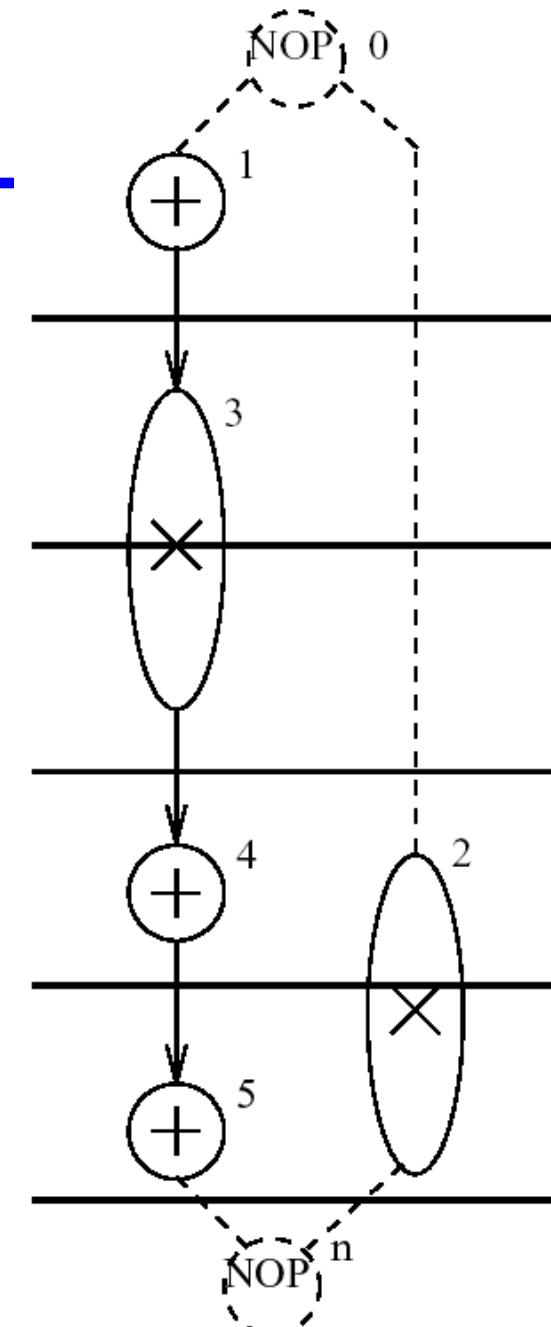
- In the example, the solution is independent of the chosen priority function.
- Because of the greedy selection principle, all resource are occupied in the first time step.
- List scheduling is a heuristic algorithm:
In this example, it does not yield the minimal latency!



List Scheduling

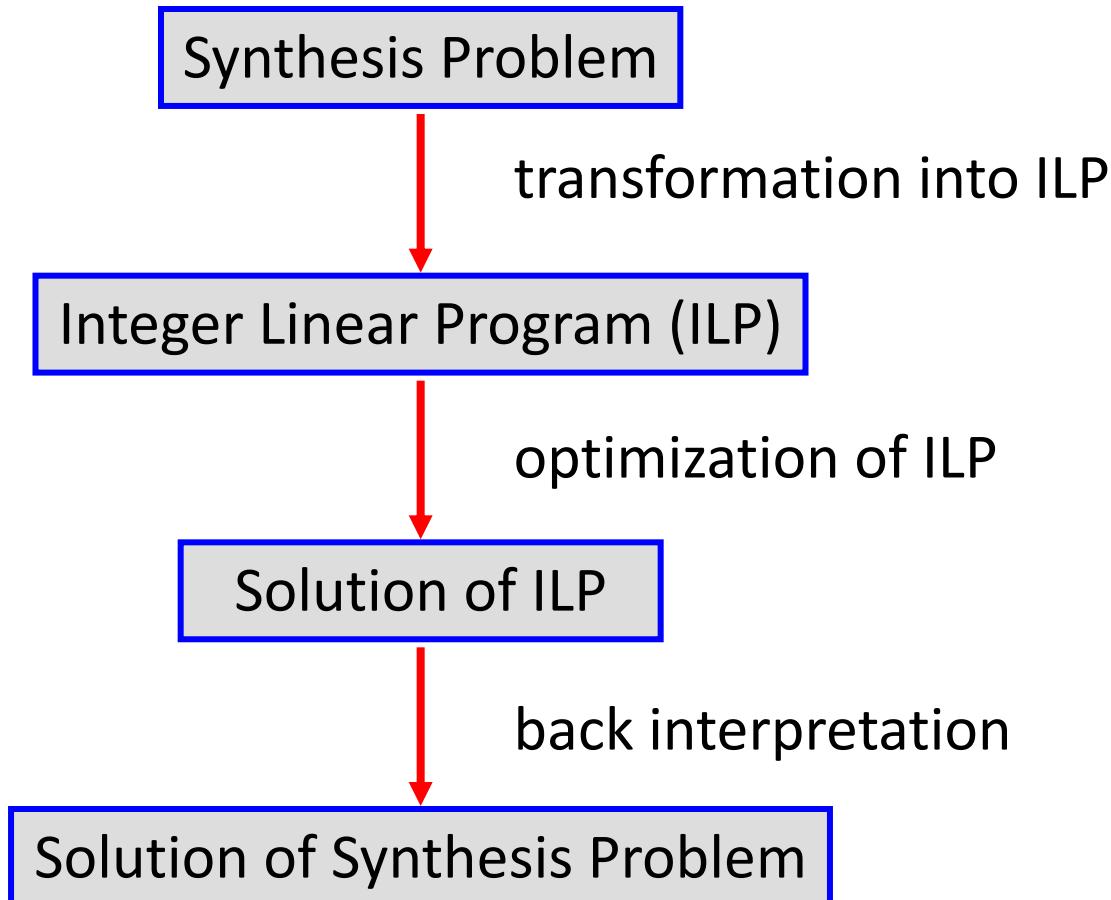
Solution via an optimal method:

- 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

$$\begin{aligned} \text{minimize: } & \tau(v_n) - \tau(v_0) \\ \text{subject to } & x_{i,t} \in \{0, 1\} \quad \forall v_i \in V_S \quad \forall t : l_i \leq t \leq h_i \end{aligned} \quad (1)$$

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

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

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

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

Integer Linear Program

$$\text{minimize: } \tau(v_n) - \tau(v_0)$$

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

Objective
function

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

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

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

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

Integer Linear Program

$$\begin{aligned} \text{minimize: } & \tau(v_n) - \tau(v_0) \\ \text{subject to } & x_{i,t} \in \{0, 1\} \quad \forall v_i \in V_S \quad \forall t : l_i \leq t \leq h_i \end{aligned} \quad (1)$$

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

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

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

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

Scheduling constraints
in terms of starting times
and executing times

Integer Linear Program

minimize: $\tau(v_n) - \tau(v_0)$
subject to $x_{i,t} \in \{0, 1\} \quad \forall v_i \in V_S \quad \forall t : l_i \leq t \leq h_i \quad (1)$

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

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

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

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

Encoding using
integer variables

Integer Linear Program

$$\begin{aligned} \text{minimize: } & \tau(v_n) - \tau(v_0) \\ \text{subject to } & x_{i,t} \in \{0, 1\} \quad \forall v_i \in V_S \quad \forall t : l_i \leq t \leq h_i \end{aligned} \quad (1)$$

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

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

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

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

Linking integer variables
to the starting times

Integer Linear Program

$$\begin{aligned} \text{minimize: } & \tau(v_n) - \tau(v_0) \\ \text{subject to } & x_{i,t} \in \{0, 1\} \quad \forall v_i \in V_S \quad \forall t : l_i \leq t \leq h_i \end{aligned} \quad (1)$$

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

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

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

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

???

Integer Linear Program

$$\begin{aligned} \text{minimize: } & \tau(v_n) - \tau(v_0) \\ \text{subject to } & x_{i,t} \in \{0, 1\} \quad \forall v_i \in V_S \quad \forall t : l_i \leq t \leq h_i \end{aligned} \quad (1)$$

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

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

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

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

Number of operations
using resources of type
 v_k at time t

Integer Linear Program

$$\begin{aligned} \text{minimize: } & \tau(v_n) - \tau(v_0) \\ \text{subject to } & x_{i,t} \in \{0, 1\} \quad \forall v_i \in V_S \quad \forall t : l_i \leq t \leq h_i \end{aligned} \quad (1)$$

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

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

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

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

For any time t and any resource type v_k , make sure we never use more resources than available

Integer Linear Program

Explanations:

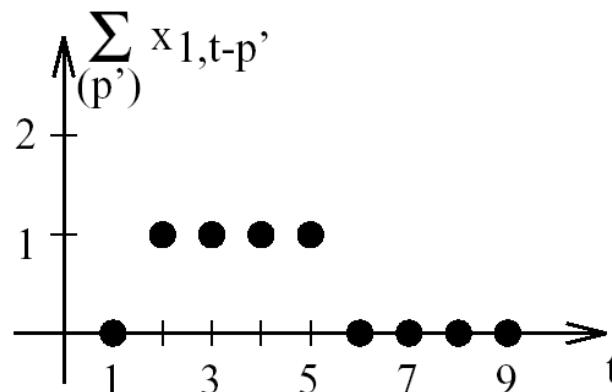
- (1) declares variables x to be binary .
- (2) makes sure that exactly one variable $x_{i,t}$ for all t has the value 1, all others are 0.
- (3) determines the relation between variables x and starting times of operations τ .
In particular, if $x_{i,t} = 1$ then the operation v_i starts at time t ,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

Integer Linear Program

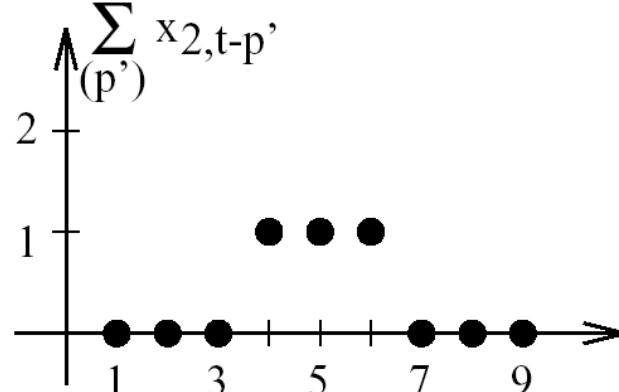
Explanations:

- (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 :

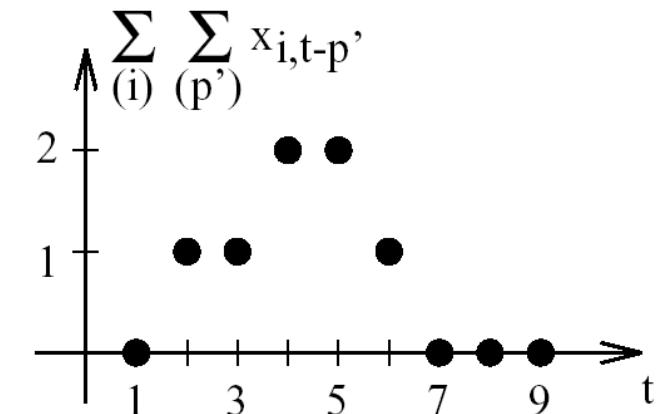
$$\sum_{p'=0}^{w(v_i)-1} x_{i,t-p'} = \begin{cases} 1 & : \quad \forall t : \tau(v_i) \leq t \leq \tau(v_i) + w(v_i) - 1 \\ 0 & : \quad \text{sonst} \end{cases}$$



$$w(v_1) = 4, \tau(v_1) = 2$$

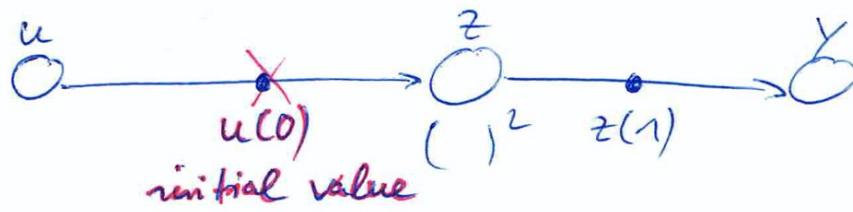


$$w(v_2) = 3, \tau(v_2) = 4$$



Architecture Synthesis for Iterative Algorithms and Marked Graphs

Remember ...

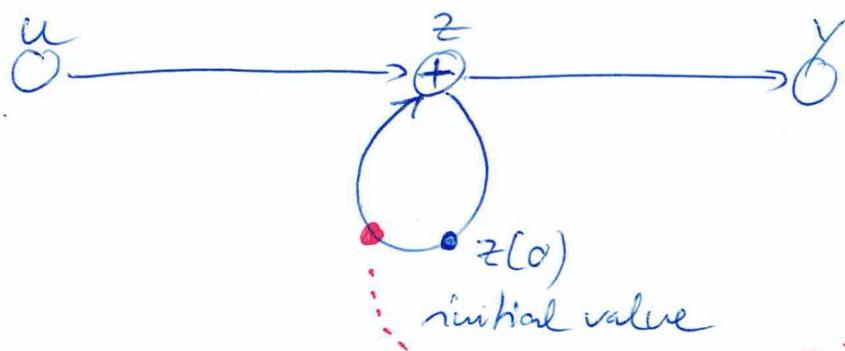


$$z(1) = (u(0))^2$$

$$z(L) = (u(1))^2$$

$$z(t) = (u(t-1))^2 \quad \forall t \geq 1$$

initial value (taken) leads to an index shift of -1



$$z(t) = u(t) + z(t-1) \quad \forall t \geq 1$$

-2

a second initial token would lead to an index shift of -2

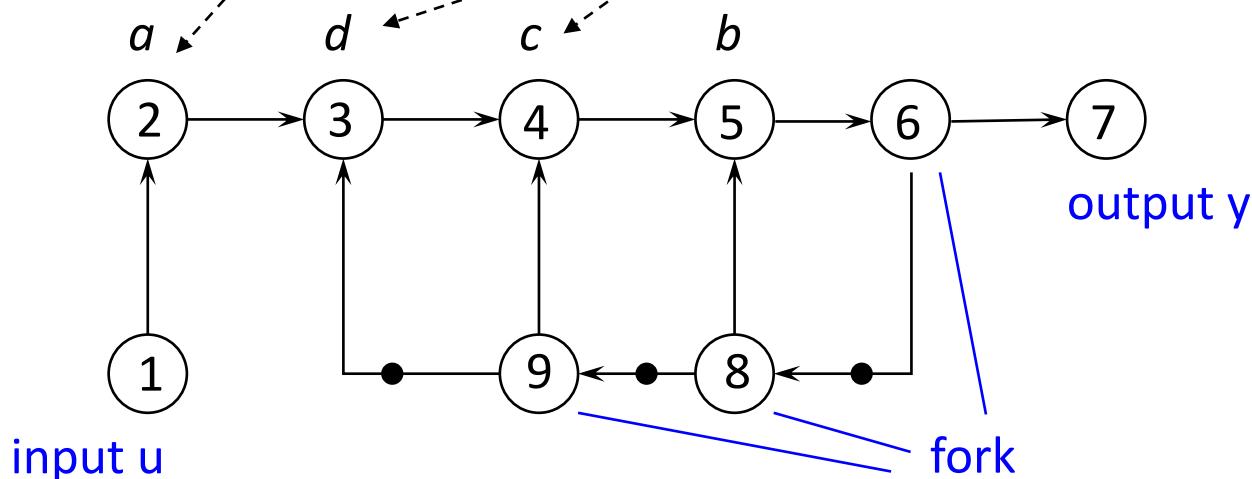
Remember ... : Marked Graph

Example (model of a digital filter with infinite impulse response IIR)

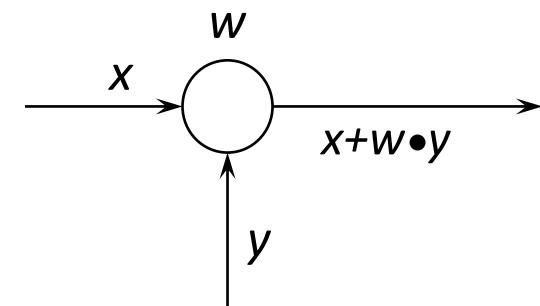
- Filter equation:

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

- 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 :

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

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

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

Iterative Algorithms

Several *representations* of the same iterative algorithm:

- One indexed equation with constant index dependencies:

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

- Equivalent set of indexed equations:

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

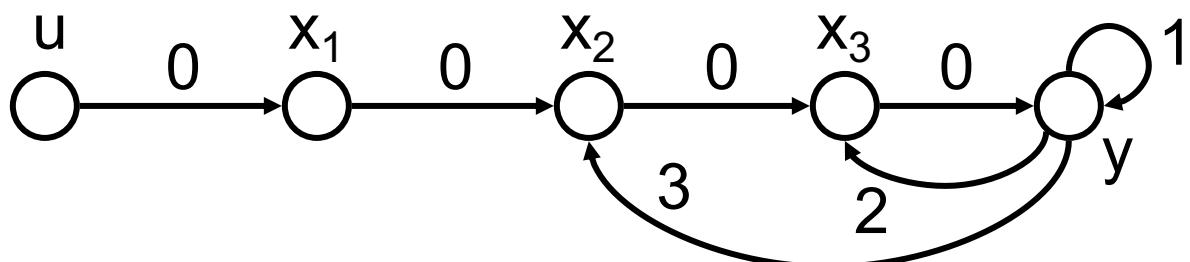
$$x_2[l] = x_1[l] + dy[l - 3] \quad \forall l$$

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

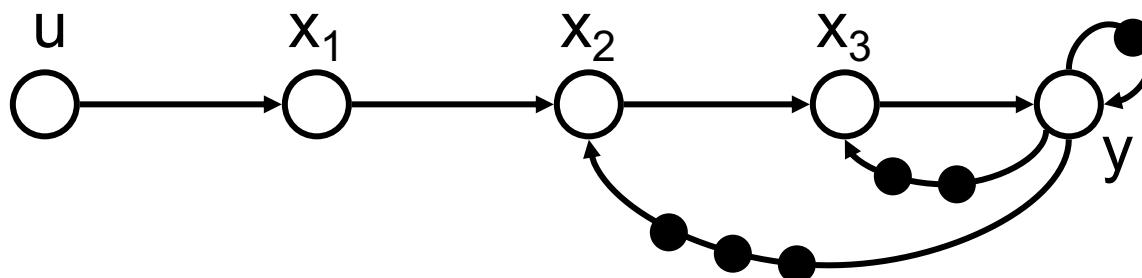
$$y[l] = x_3[l] + by[l - 1] \quad \forall l$$

Iterative Algorithms

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



Equivalent *marked graph*:



Iterative Algorithms

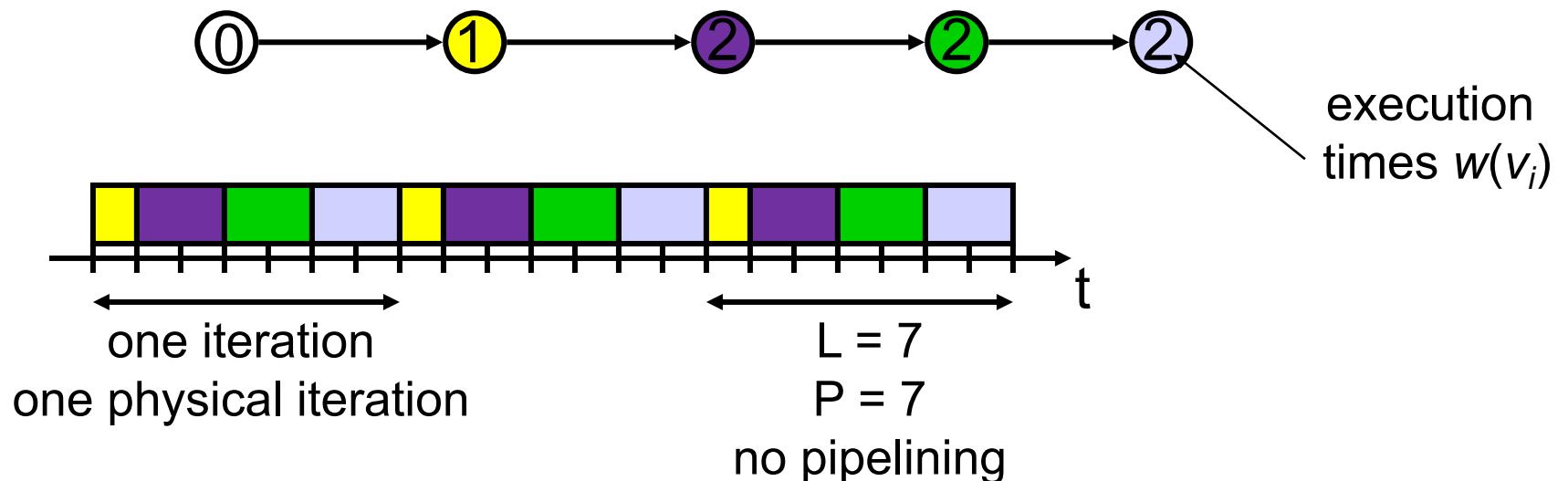
- An *iteration* is the set of all operations necessary to compute all variables $x_i[l]$ for a fixed index l .
- The *iteration interval* P is the time distance between two successive iterations of an iterative algorithm. $1/P$ denotes the *throughput* of the implementation.
- The *latency* L is the maximal time distance between the starting and the finishing times of operations belonging to the same 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:

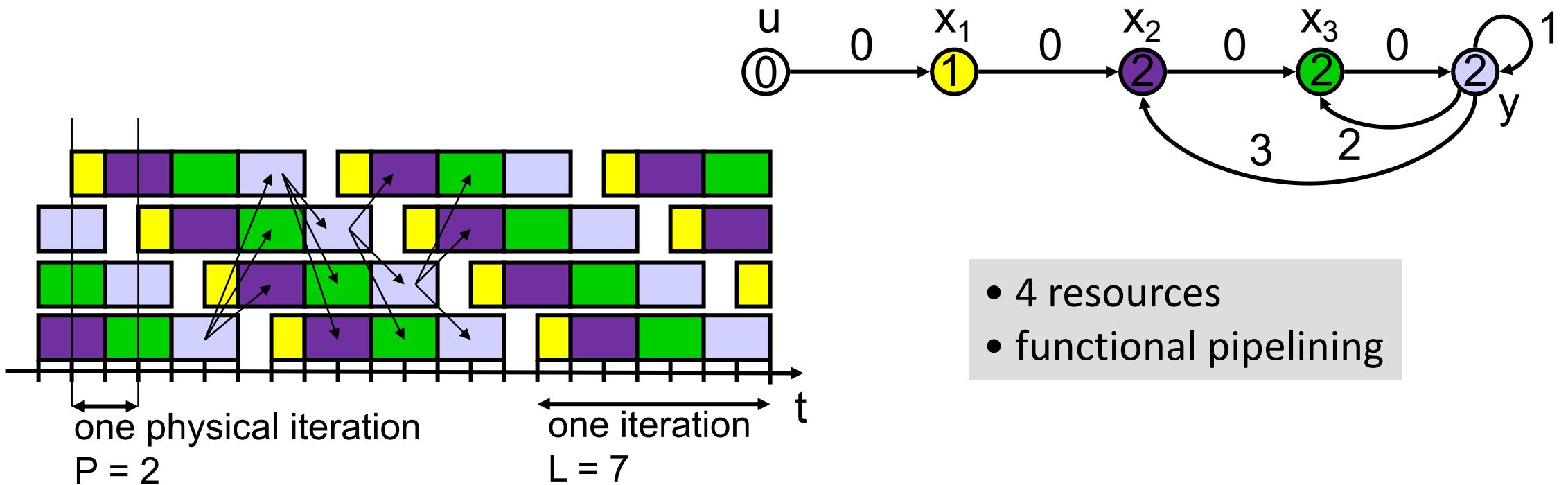


Iterative Algorithms

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!)

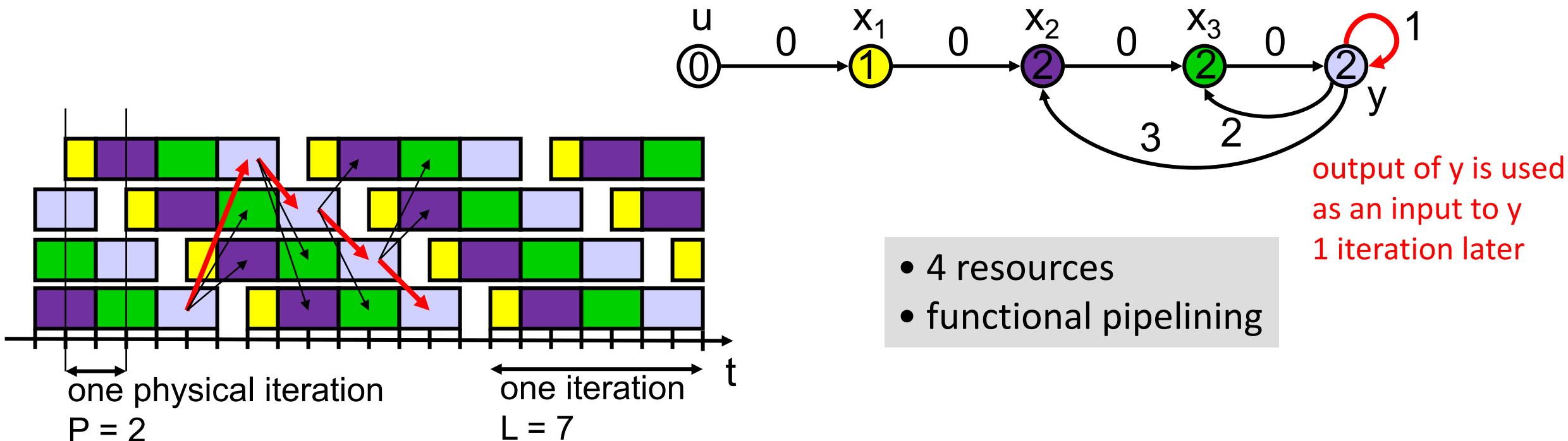


Iterative Algorithms

Implementation principles

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

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

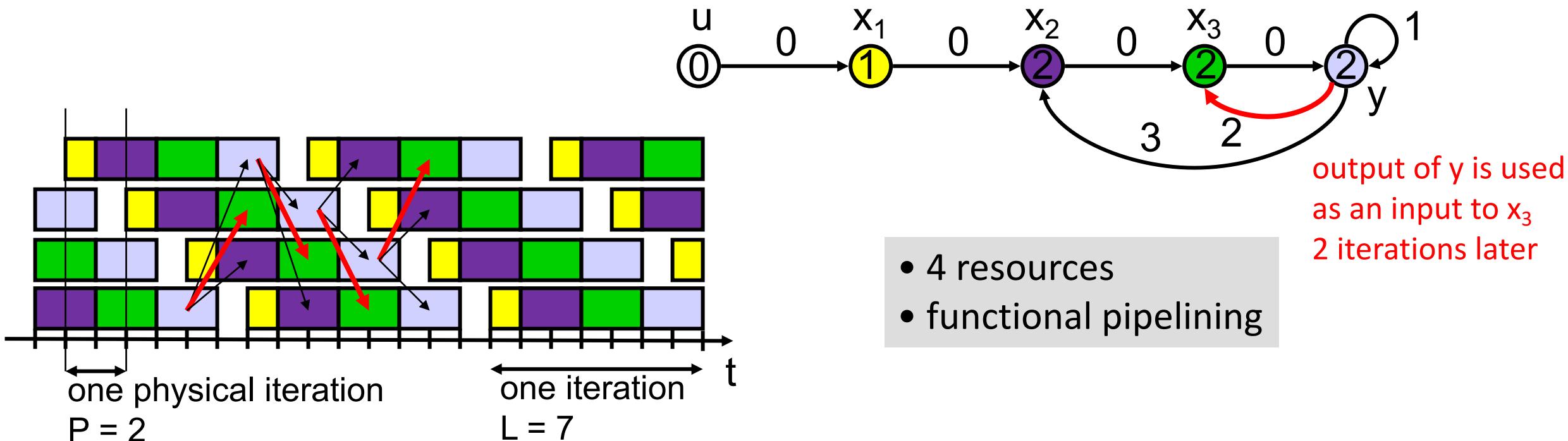


Iterative Algorithms

Implementation principles

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

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

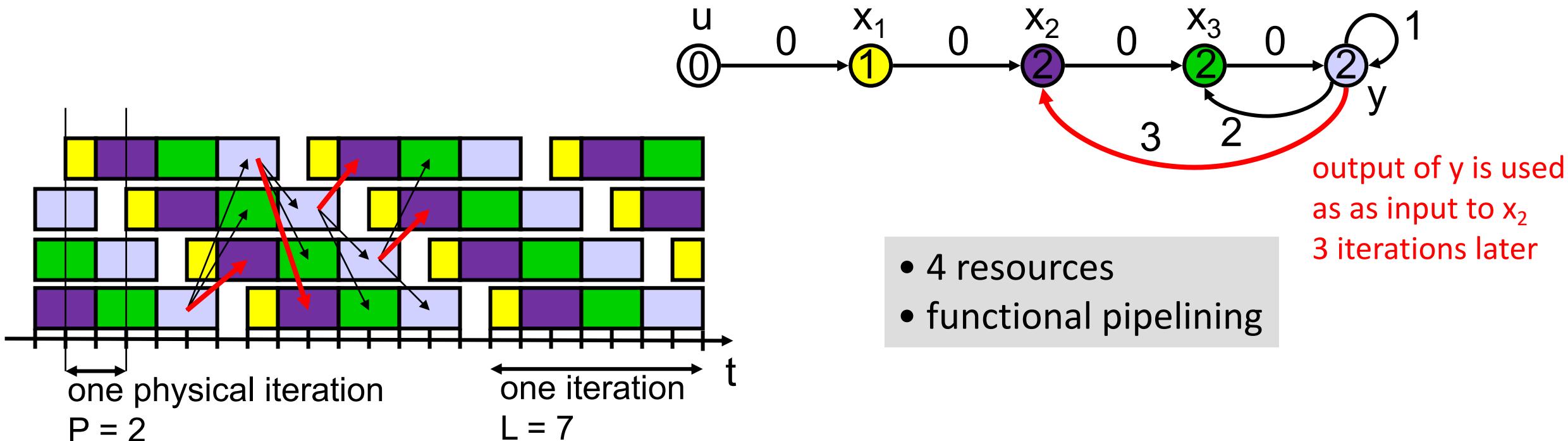


Iterative Algorithms

Implementation principles

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

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



Iterative Algorithms

Solving the synthesis problem using *integer linear programming*:

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

Integer Linear Program

$$\begin{aligned} \text{minimize: } & \tau(v_n) - \tau(v_0) \\ \text{subject to } & x_{i,t} \in \{0, 1\} \quad \forall v_i \in V_S \quad \forall t : l_i \leq t \leq h_i \end{aligned} \quad (1)$$

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

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

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

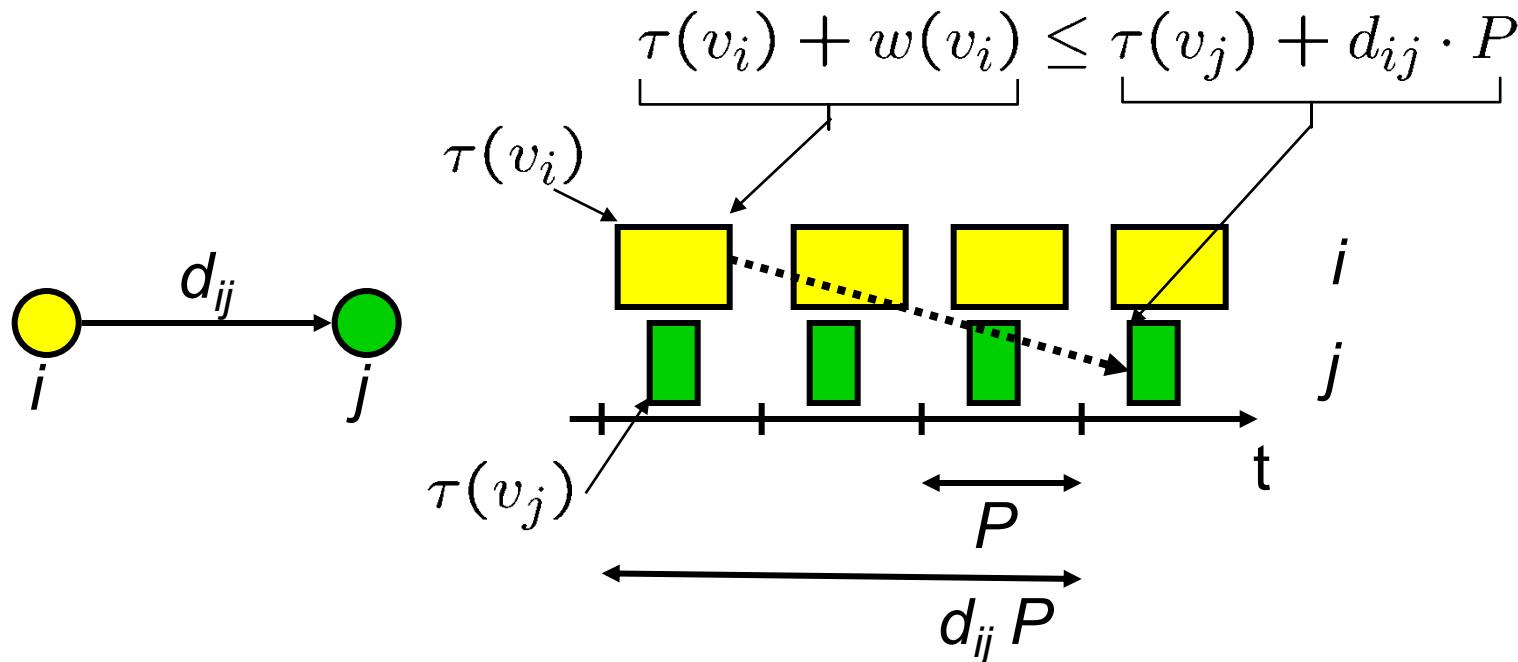
$$\sum_{\forall i: (v_i, v_k) \in E_R} \sum_{p'=\max\{0, t-h_i\}}^{\min\{w(v_i)-1, t-l_i\}} x_{i,t-p'} \leq \alpha(v_k)$$
$$\forall v_k \in V_T \quad \forall t : 1 \leq t \leq \max\{h_i : v_i \in V_S\} \quad (5)$$

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:

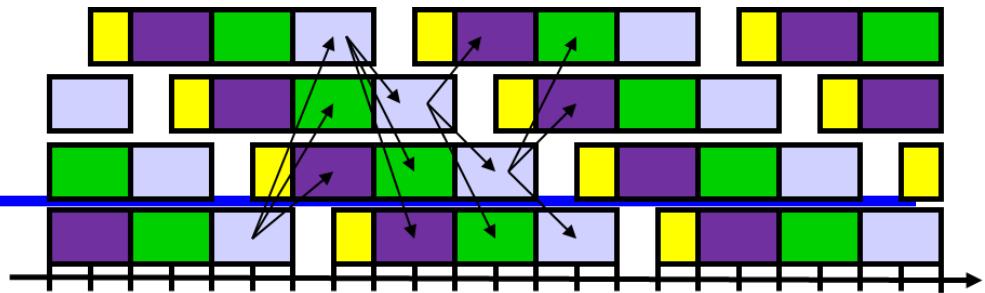


Iterative Algorithms

Eqn. (5) is replaced by

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

$\forall 1 \leq t \leq P, \forall v_k \in V_T$



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

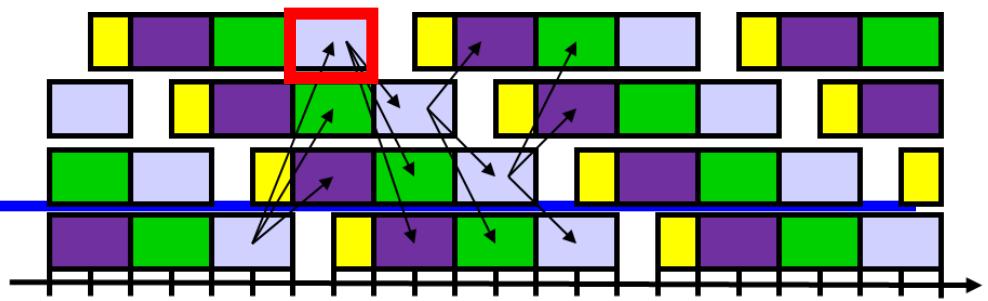
$$t = \tau(v_i) + p' - p \cdot P$$

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

Therefore, we obtain

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

Iterative Algorithms



Eqn. (5) is replaced by

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

$$\forall 1 \leq t \leq P, \forall v_k \in V_T$$

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

$$t = \tau(v_i) + p' - p \cdot P$$

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

while current instance of v_i is running

Therefore, we obtain

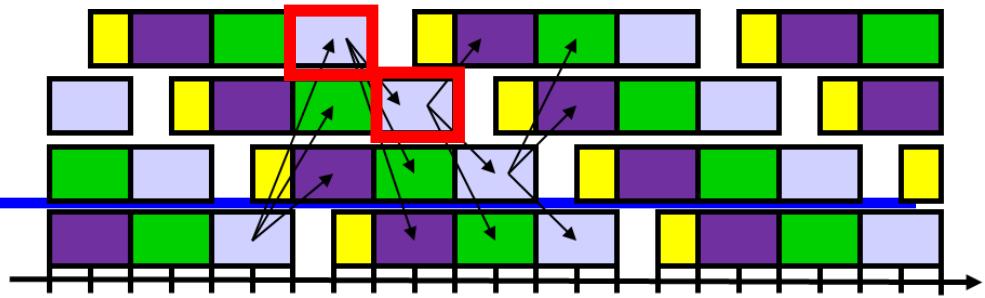
$$\sum_{p'=0}^{w(v_i)-1} \sum_{\forall p: l_i \leq t - p' + p \cdot P \leq h_i} x_{i,t-p'+p \cdot P}$$

Iterative Algorithms

Eqn. (5) is replaced by

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

$$\forall 1 \leq t \leq P, \forall v_k \in V_T$$



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

$$t = \tau(v_i) + p' - p \cdot P$$

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

while current instance of v_i is running

v_i starts to run every P time units

Therefore, we obtain

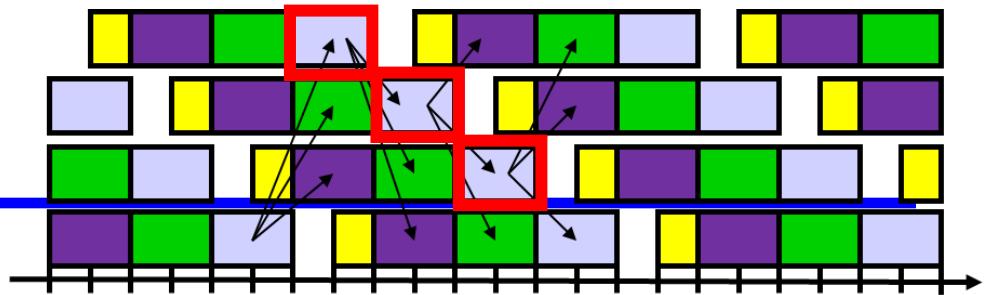
$$\sum_{p'=0}^{w(v_i)-1} \sum_{\forall p: l_i \leq t - p' + p \cdot P \leq h_i} x_{i,t-p'+p \cdot P}$$

Iterative Algorithms

Eqn. (5) is replaced by

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

$$\forall 1 \leq t \leq P, \forall v_k \in V_T$$



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

$$t = \tau(v_i) + p' - p \cdot P$$

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

while current instance of v_i is running

v_i starts to run every P time units

Therefore, we obtain

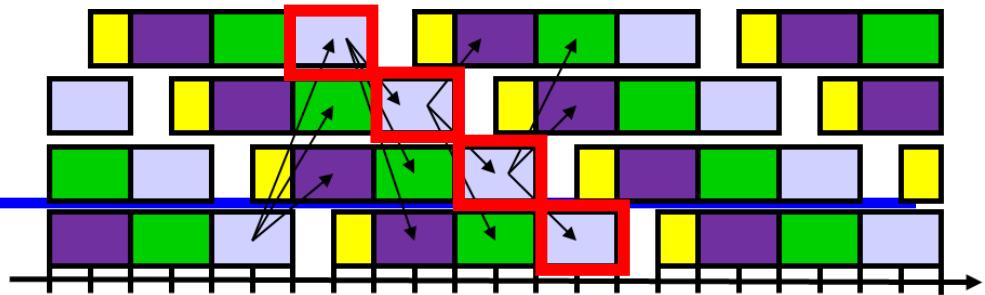
$$\sum_{p'=0}^{w(v_i)-1} \sum_{\forall p: l_i \leq t - p' + p \cdot P \leq h_i} x_{i,t-p'+p \cdot P}$$

Iterative Algorithms

Eqn. (5) is replaced by

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

$$\forall 1 \leq t \leq P, \forall v_k \in V_T$$



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

$$t = \tau(v_i) + p' - p \cdot P$$

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

while current instance of v_i is running

v_i starts to run every P time units

Therefore, we obtain

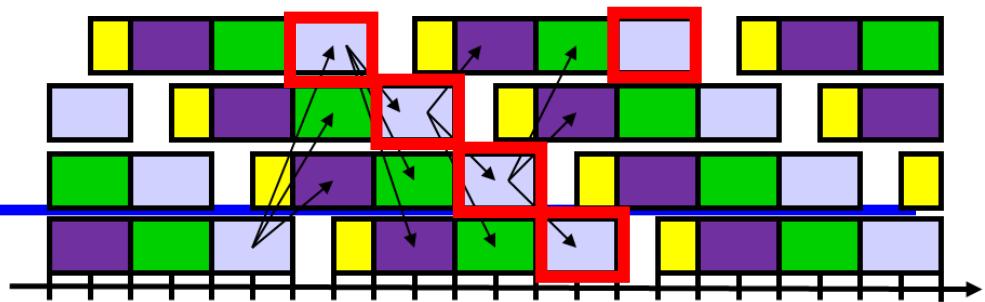
$$\sum_{p'=0}^{w(v_i)-1} \sum_{\forall p: l_i \leq t - p' + p \cdot P \leq h_i} x_{i,t-p'+p \cdot P}$$

Iterative Algorithms

Eqn. (5) is replaced by

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

$$\forall 1 \leq t \leq P, \forall v_k \in V_T$$



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

$$t = \tau(v_i) + p' - p \cdot P$$

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

while current instance of v_i is running

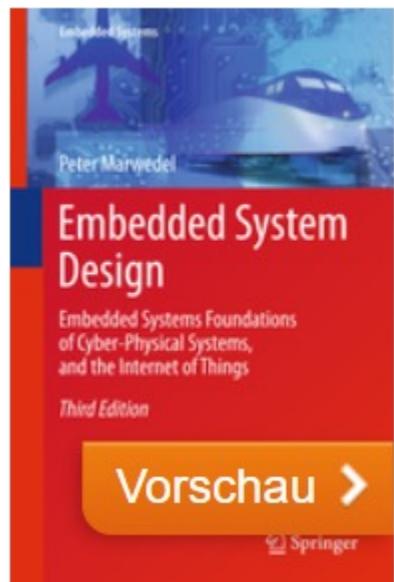
v_i starts to run every P time units

Therefore, we obtain

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

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