Sensors/Actuators for embedded and intelligent systems

Polytech Nice Sophia

B. Miramond

Progression of embedded notions

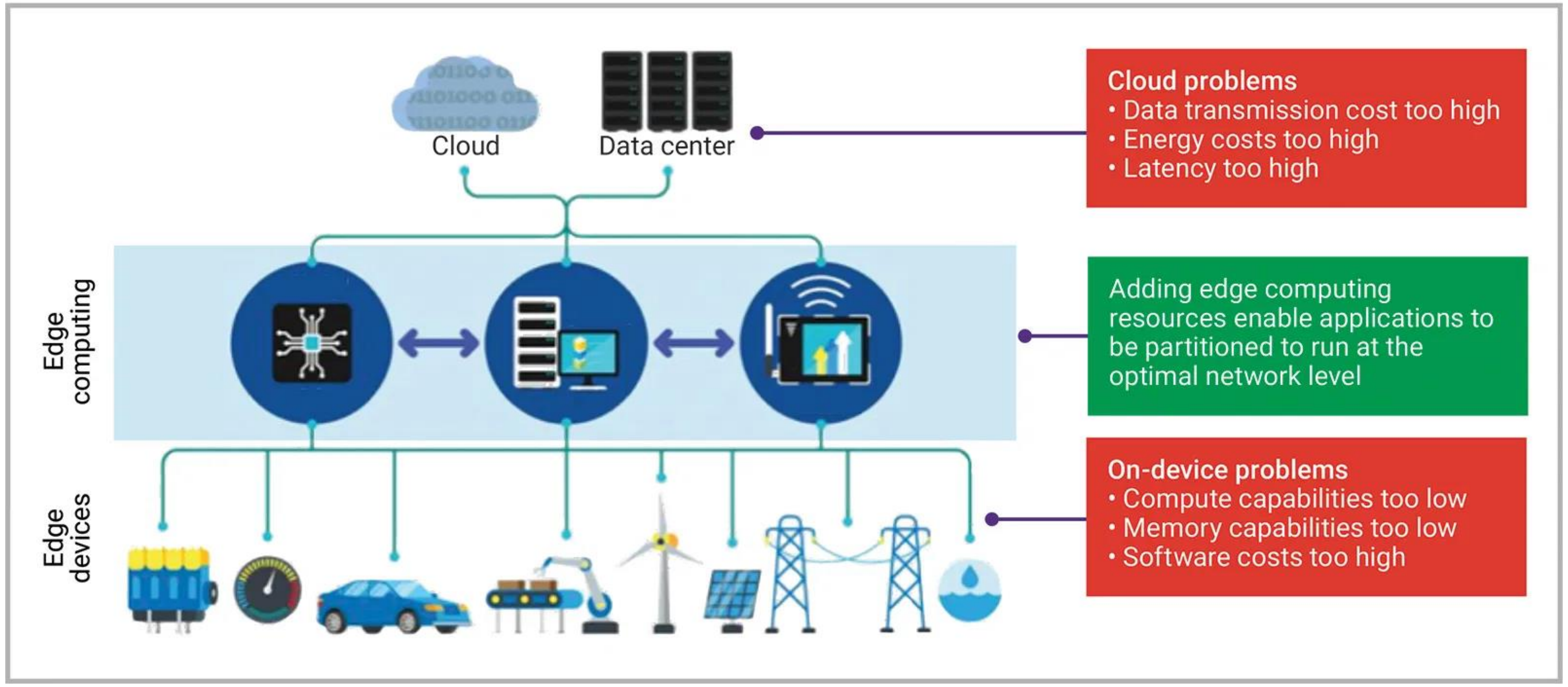
- SI3 – architecture of computers, instruction set, execution principle of Von Neumann machines
- SI4 – Sensors/actuators, real-time processing, embedded AI
- SI5 – hardware-software codesign, SoC, FPGA

Organisation of the course

- 12 class sessions – lecture, 1 hour
- 12 lab sessions, 2 hours => starts 09/02/21
- 3 parts (can be re-organised according to the conditions):
  1. Embedded programming and peripherals
     - face-to-face labs for at least half of the students
  2. Real-time scheduling
     - exercices can be done remotely
  3. Embedded AI on MCU
     - labs can be done remotely
- 3 ratings : test 1, labs, test 2

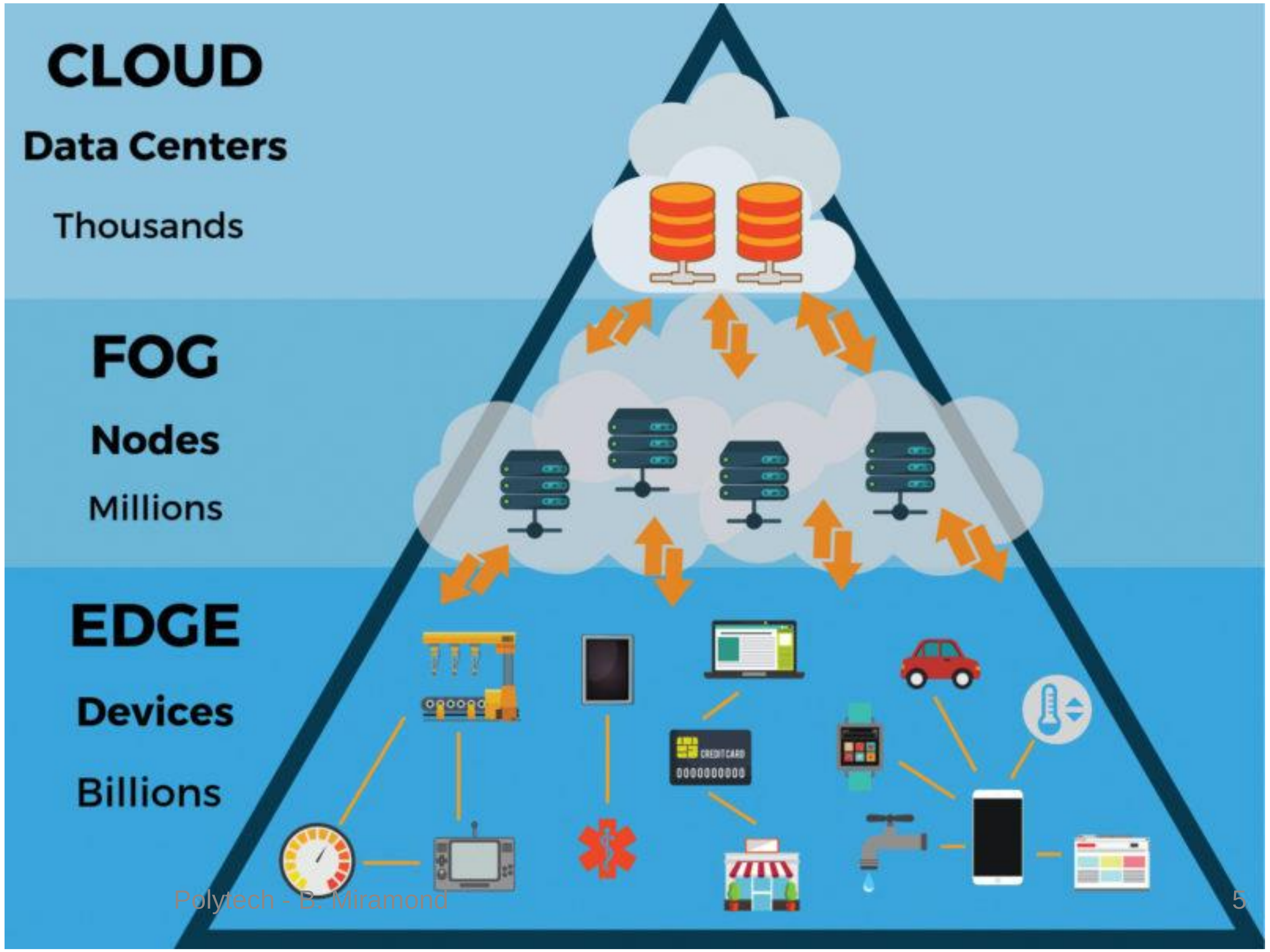
Edge computing

- Lightweight machine learning
- Energy efficient electronic
- Low-power wireless communications
- Bandwith improvements
- Micro-controler programming

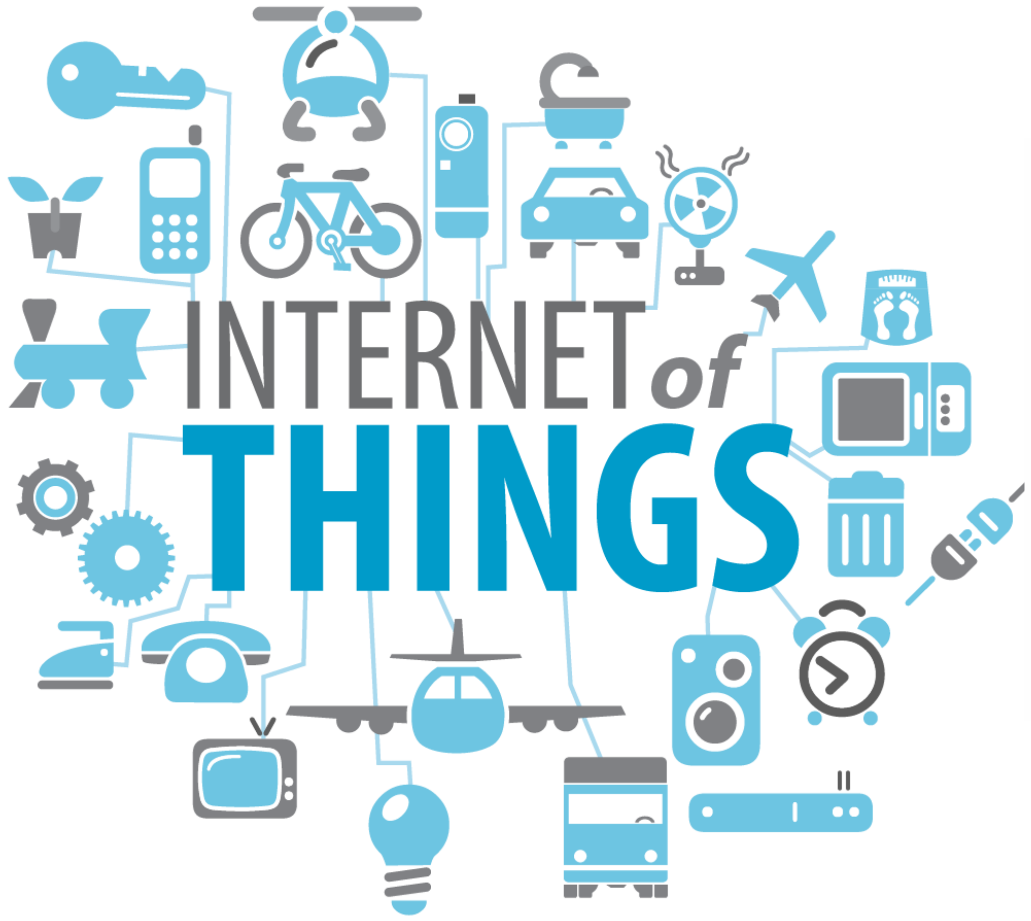

Cloud
Data center
Edge computing
Edge devices
Cloud problems
• Data transmission cost too high
• Energy costs too high
• Latency too high
Adding edge computing resources enable applications to be partitioned to run at the optimal network level
On-device problems
• Compute capabilities too low
• Memory capabilities too low
• Software costs too high

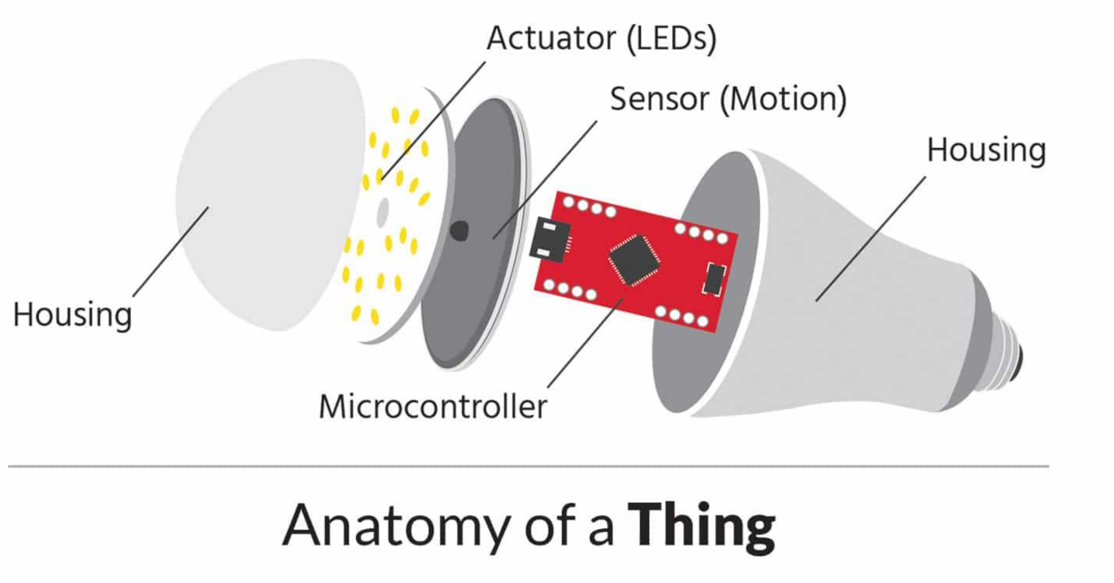
Edge computing

- Reliability
- Privacy
- Latency
- Low-power

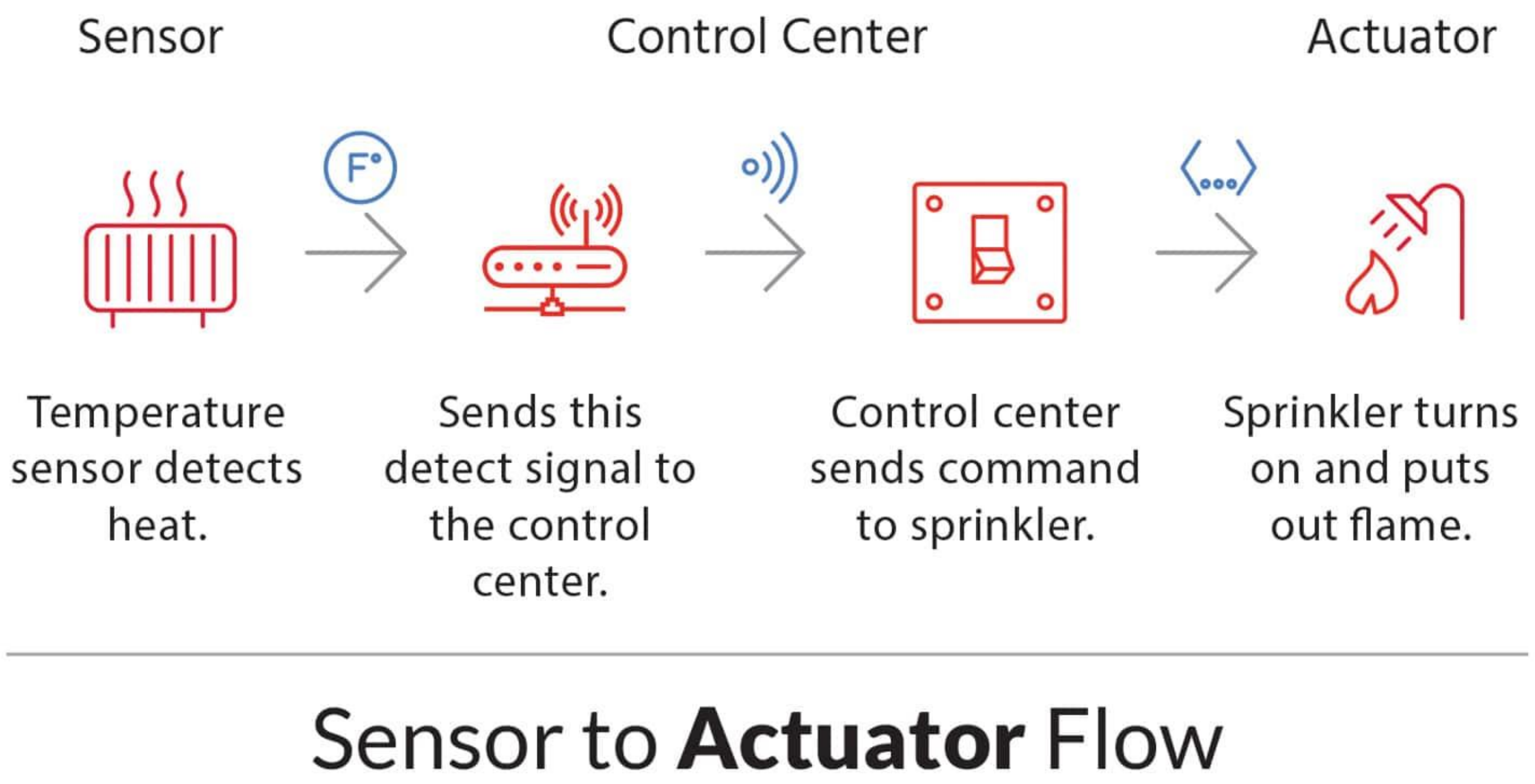


IoT - Internet of Things



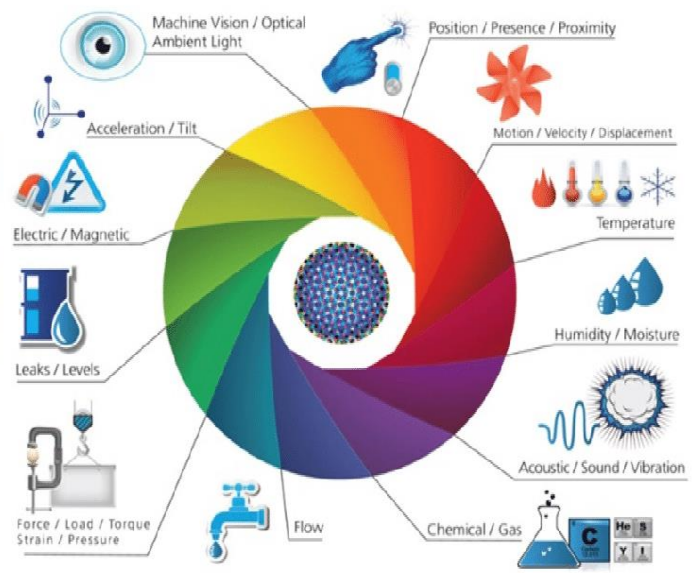


From sensors to actuators


Sensor
Control Center
Actuator
F°
Temperature sensor detects heat.
Sends this detect signal to the control center.
Control center sends command to sprinkler.
Sprinkler turns on and puts out flame.
Sensor to **Actuator** Flow

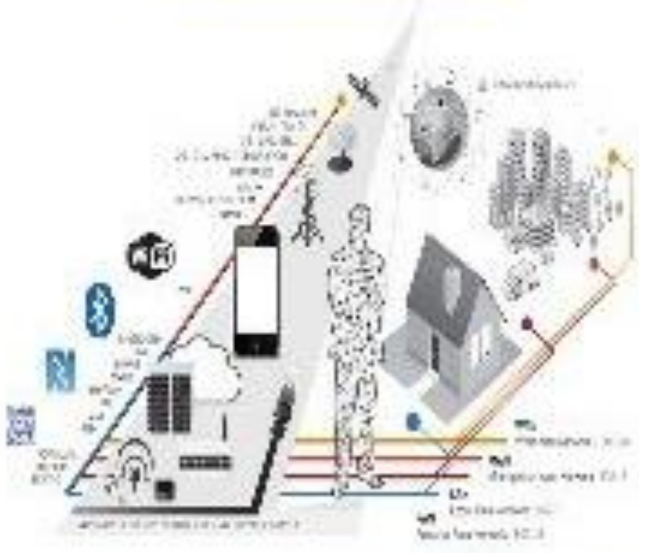
Key elements of IoT

Sensors & Actuators


Machine Vision / Optical Ambient Light
Position / Presence / Proximity
Acceleration / Tilt
Motion / Velocity / Displacement
Electric / Magnetic
Temperature
Humidity / Moisture
Leaks / Levels
Acoustic / Sound / Vibration
Force / Load / Torque Strain / Pressure
Flow
Chemical / Gas

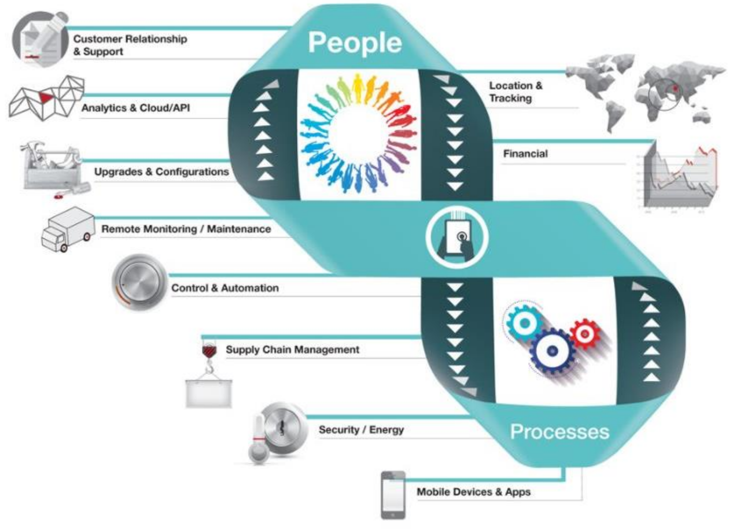
- Providing Digital information

Connectivity



- Wired or Wireless

People & Process



- Computing
  - Local & Cloud
- Controls/Actions/ Interfaces
  - Multimedia & Physical Actions

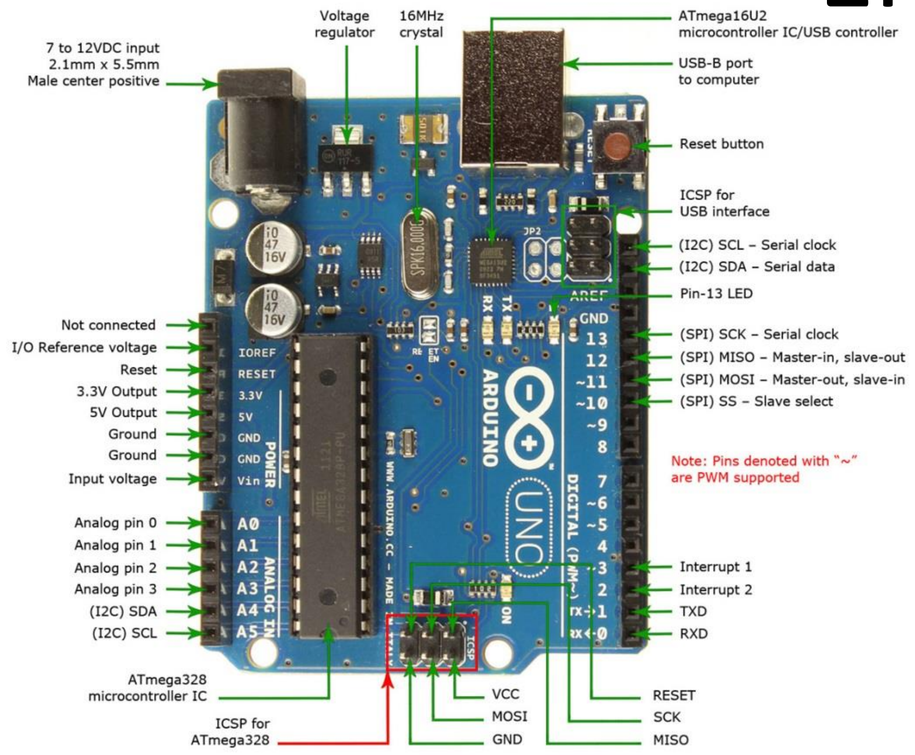
Experiment embedded programming

Raspberry Pi 4 B
ARM-Cortex-A72 4 x 1,50 GHz
4 Go de RAM
35 $

Raspberry Pi Pico

- ARM Cortex M0 x2 - 133 MHz, 264 Ko SRAM + 2 Mo Flash memory
- 26 GPIO
- 2 × SPI, 2 × I2C, 2 × UART, 3 × ADC 12 bits, 16 × PWM
- 4 $

Embedded Dev kits



ATmega 328
32 KB Flash
2KB SRAM

Arduino Zero
Architecture
	ARMv7 Cortex-M0
Processor
	Atmel SAMD21 – 48MHz
RAM
	32 KB

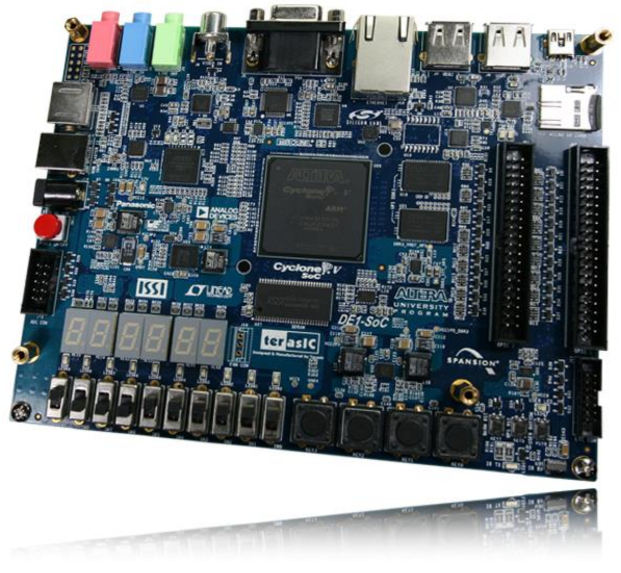
Goals of the course

- Understand the software and hardware mechanisms at play in sensor networks
- Formalize the notion of real time computing
- Study the different types of sensors and actuators
- Understand the principles of embedded processing and associated wireless communications
- Understand the challenges of embedded AI

In practice

- Programming SoC microcontroller and peripherals
- Getting started with programming on a real-time OS
- Program sensors/actuators-based systems
- Deploy AI algorithms onto MCU

The targetted boards



DE1-SoC Board
Cyclone V SoC 5CSEMA5F31C6 Device
Dual-core ARM Cortex-A9 (HPS)
85K Programmable Logic Elements
4,450 Kbits embedded memory

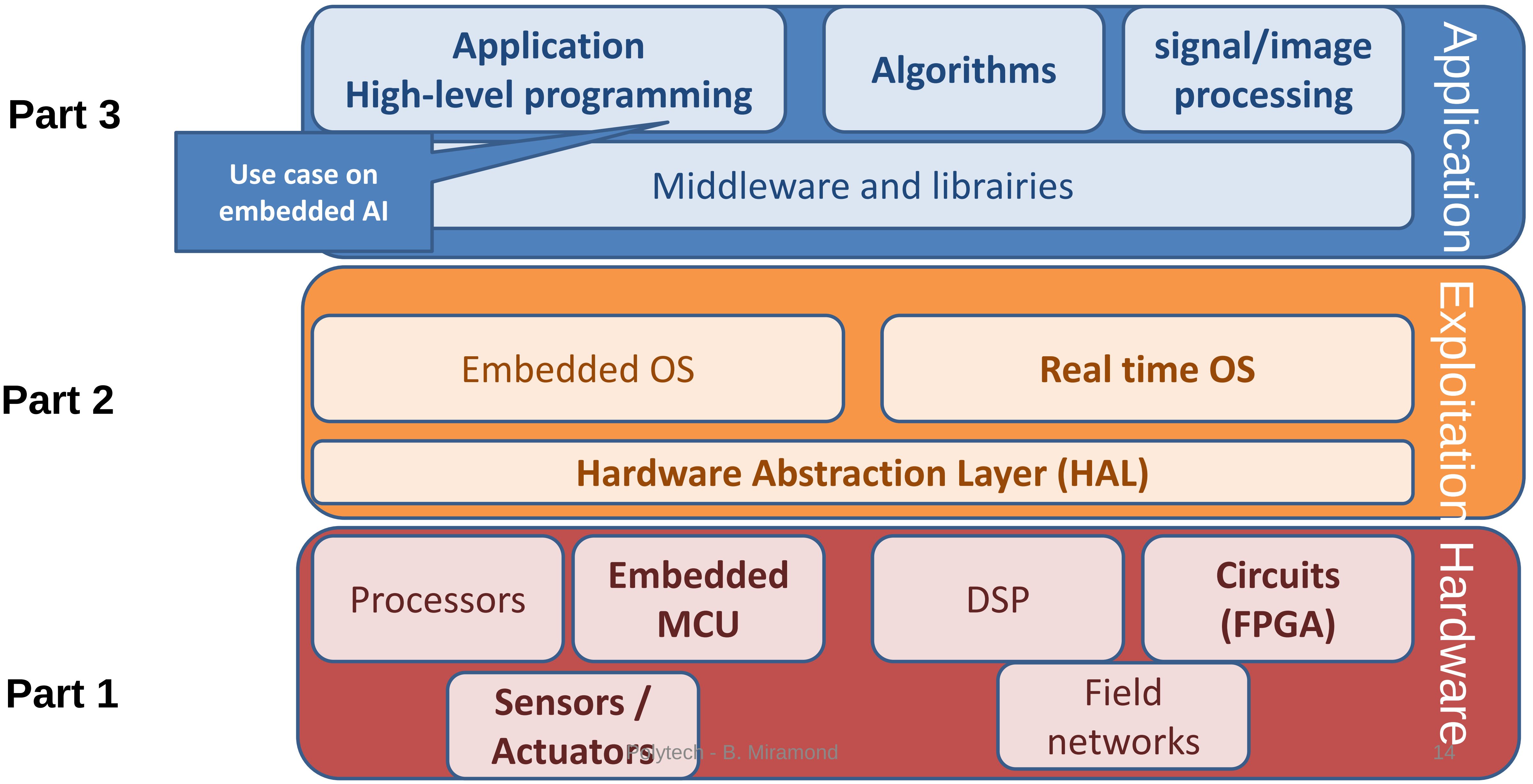
64MB (32Mx16) SDRAM on FPGA side
1GB (2x256Mx16) DDR3 SDRAM on HPS side
128 MB flash

STM32 Nucleo 64 STM32L476

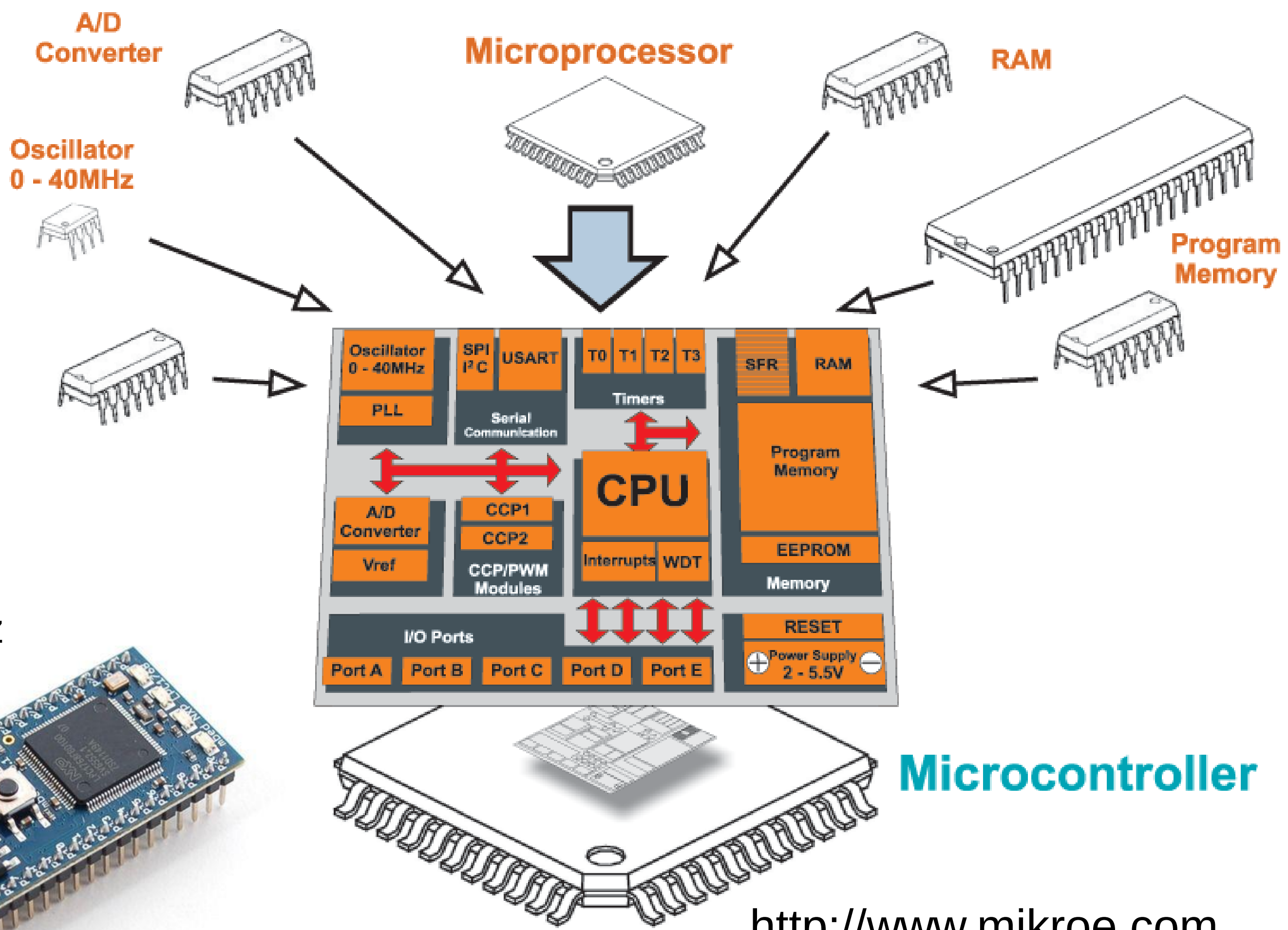
32 bits *ARM Cortex*-M4 @ 80 MHz
Flash memory up to 1 Mbyte,
up to 128 Kbyte of SRAM

PART 1 –
THE EMBEDDED SYSTEM PERIPHERALS

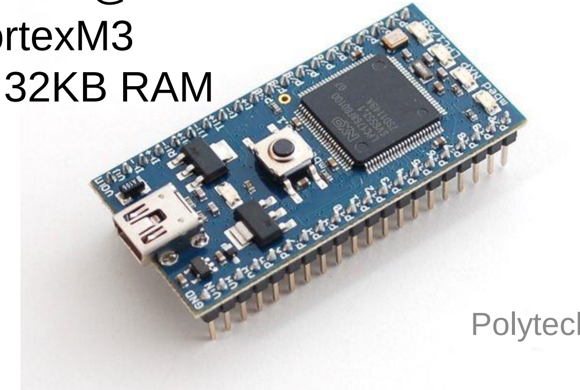
Organisation of the embedded system



The micro-controler unit (MCU) is a SoC

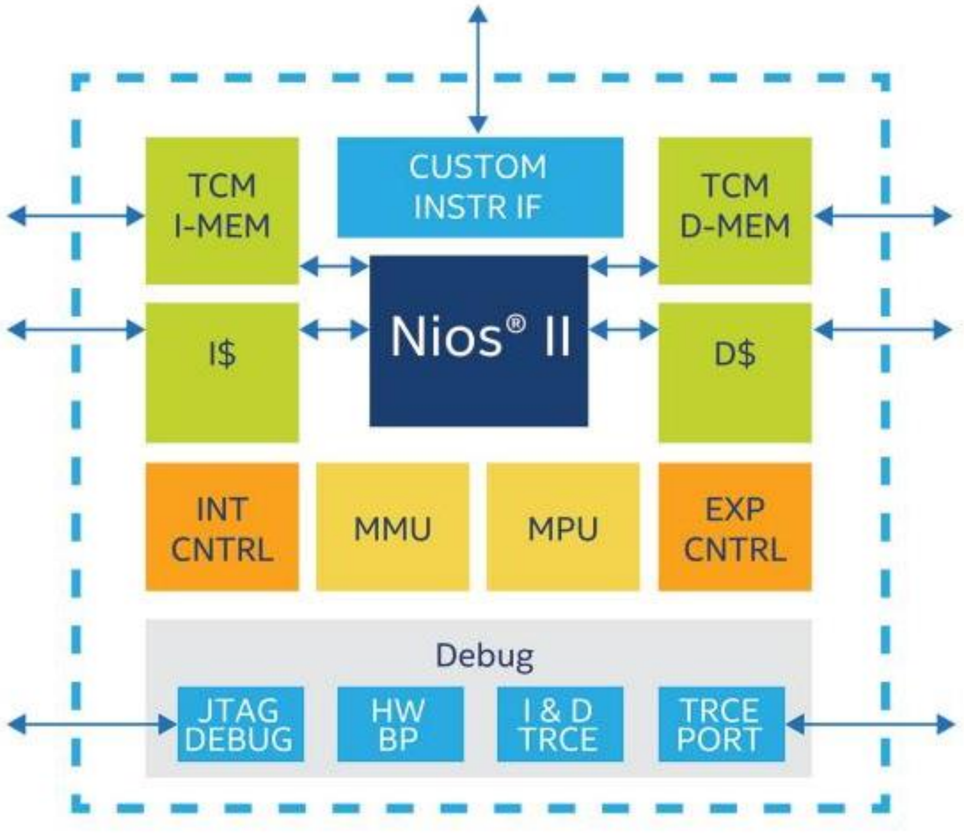


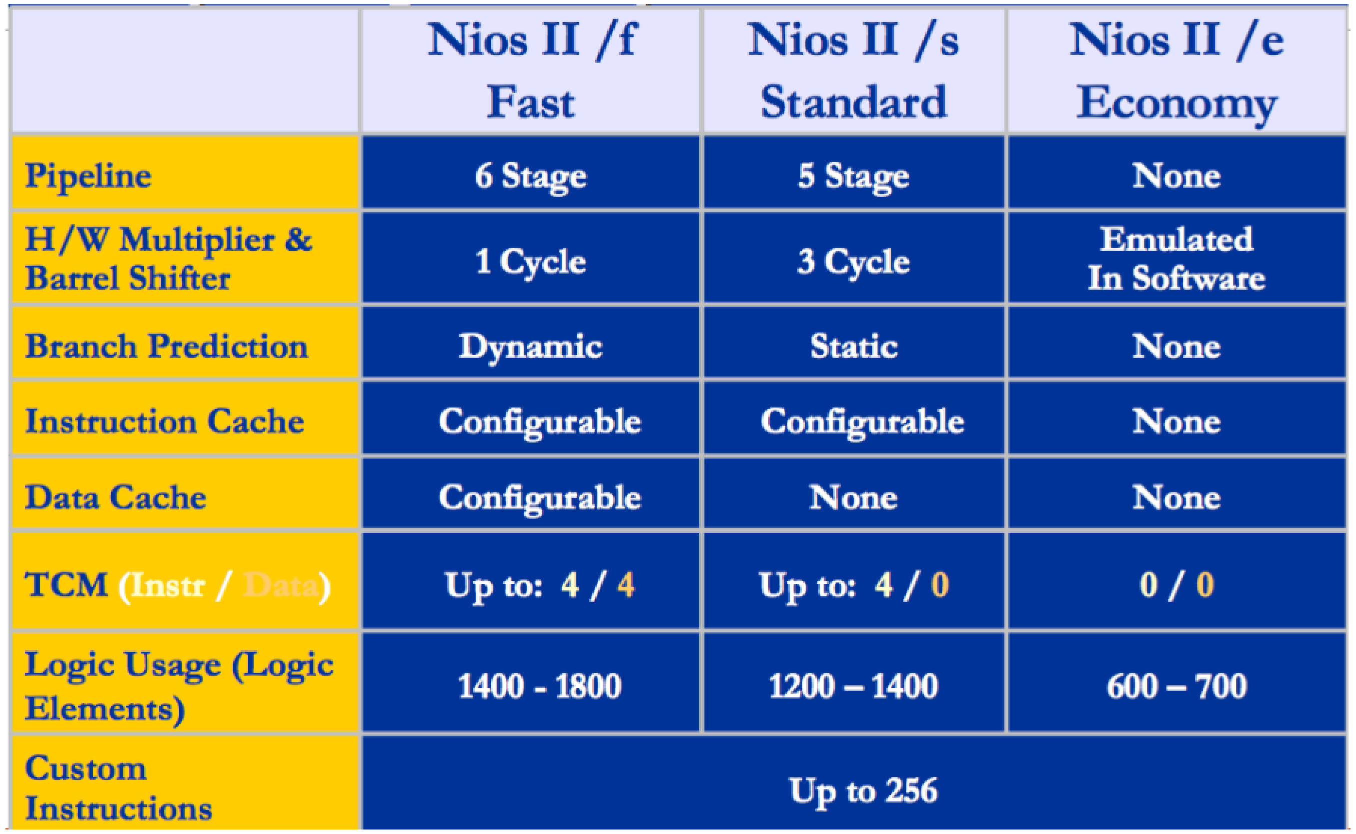
Example with MBED platform:
https://www.mbed.com/
NXP NXP LPC1768 @ 96MHz
ARM 32-bits CortexM3
512KB FLASH, 32KB RAM



http://www.mikroe.com

Nios-II embedded softcore

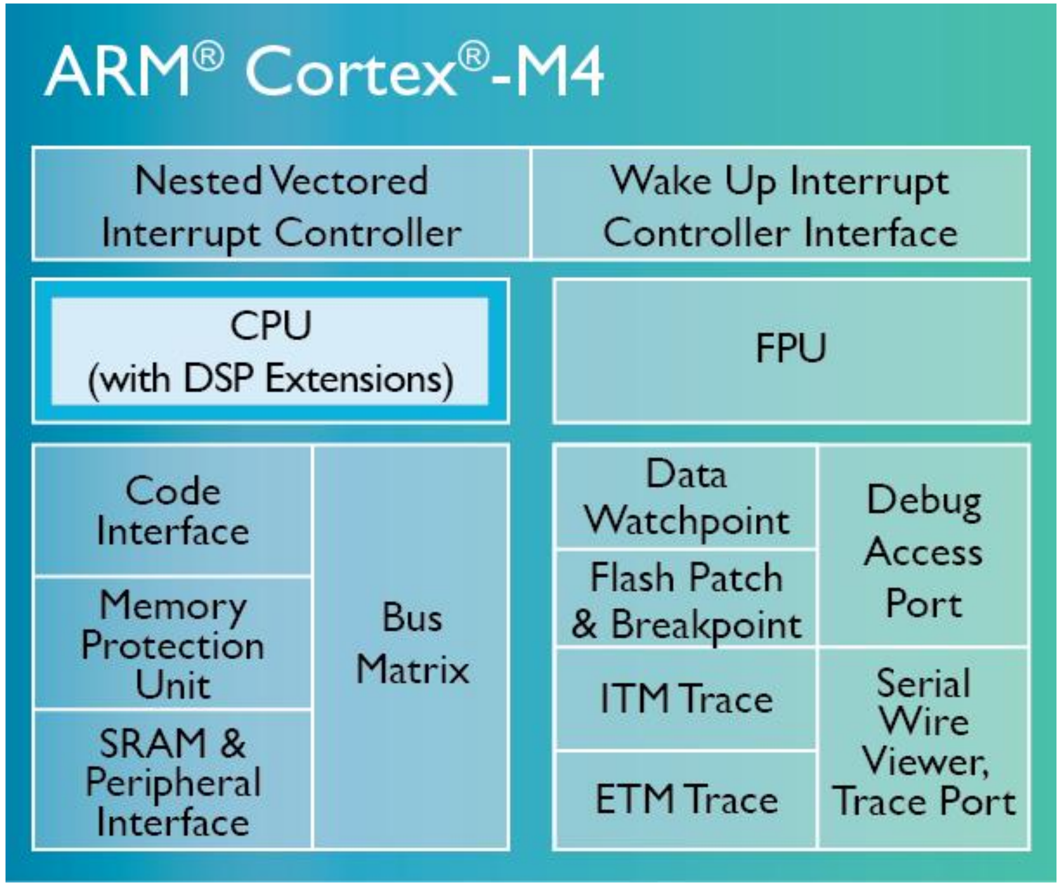




	Nios II /f Fast	Nios II /s Standard	Nios II /e Economy
Pipeline	6 Stage	5 Stage	None
H/W Multiplier & Barrel Shifter	1 Cycle	3 Cycle	Emulated In Software
Branch Prediction	Dynamic	Static	None
Instruction Cache	Configurable	Configurable	None
Data Cache	Configurable	None	None
TCM (Instr / Data)	Up to: 4 / 4	Up to: 4 / 0	0 / 0
Logic Usage (Logic Elements)	1400 - 1800	1200 – 1400	600 – 700
Custom Instructions	Up to 256		

ARM Cortex M4familly

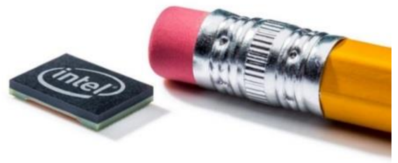
ISA Support	Thumb® / Thumb-2
Pipeline	3-stage
Performance Efficiency	1.25 / 1.50 / 1.89 DMIPS/MHz**
Floating-Point Unit	Optional single precision floating point unit IEEE 754 compliant
Memory Protection	Optional 8 region MPU
Interrupts	1 to 240 physical interrupts
Interrupt Priority Levels	8 to 256 priority levels


ARM® Cortex®-M4
Nested Vectored Interrupt Controller
Wake Up Interrupt Controller Interface
CPU (with DSP Extensions)
FPU
Code Interface
Memory Protection Unit
SRAM & Peripheral Interface
Bus Matrix
Data Watchpoint
Flash Patch & Breakpoint
ITM Trace
ETM Trace
Debug Access Port
Serial Wire Viewer, Trace Port

Programming an embedded system

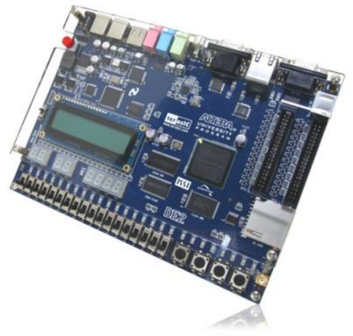
- Cross-compilation
- Limited resources (memory, processor, MMU…)
- Installation on the PC of a compiler dedicated to the ISA
- Download ELF file to the target
  - Direct transfert with JTAG (debugger)
  - Indirect transfert through network
- Standard output redirected to the serial link (JTAG, UART…)

The peripherals of the MCU



- The on-board system is A micro-controller
  - With its memory (flash + RAM)
  - For code execution
  - and therefore programming
- But in order to interact with the outside world, it requires peripherals
  - Sensors, actuators are peripherals
- The MCU communicates with them through different links
  - On board
    - Communication buses, Interruptions
  - Off board
    - Wired field networks (S2I, SPI, I2C, CAN...)
    - Wireless field networks (Bluetooth, Zigbee, Wifi, LoRA, SigFox...)

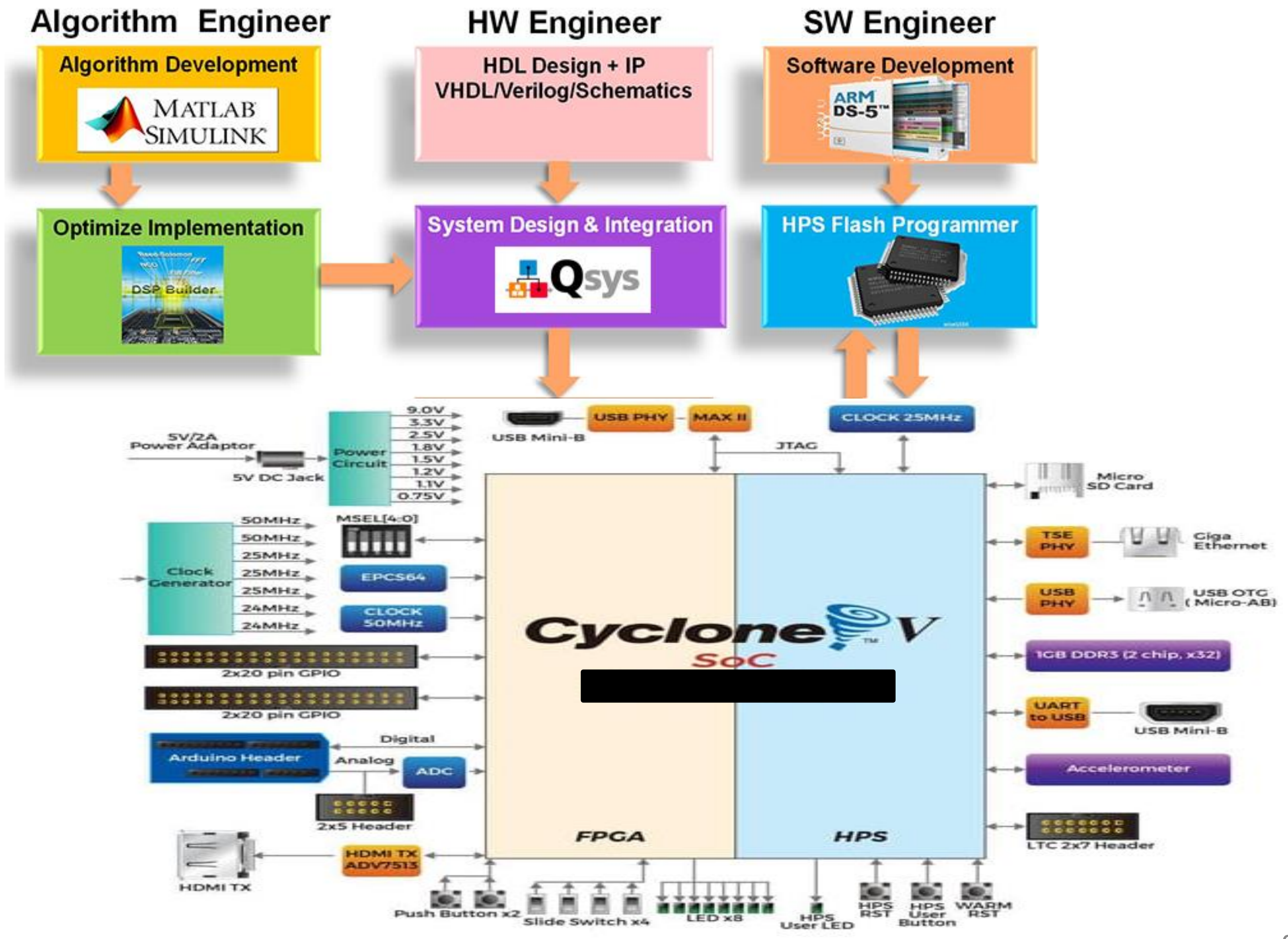
First example: on-board peripherals

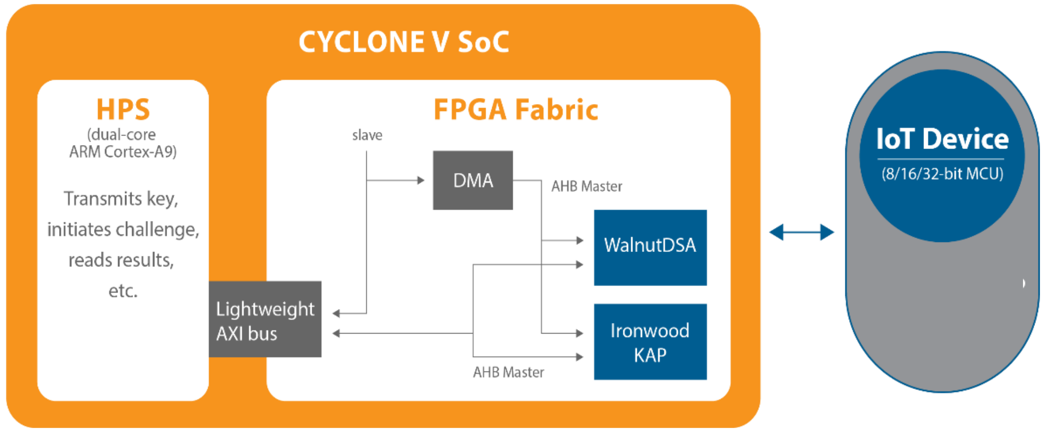


- Board: Terasic DE1 SoC
- Processor: Soft-Core Nios2 (on Altera FPGA)
- Cross-compilation
- Tool: Quartus 16.1 Lite edition (Eclipse IDE)
- Execution Bare-metal or Execution with RTOS uC/OS-II
- Peripherals: LEDs, 7-segments, buttons, switches, LCD ...

What is an FPGA ?

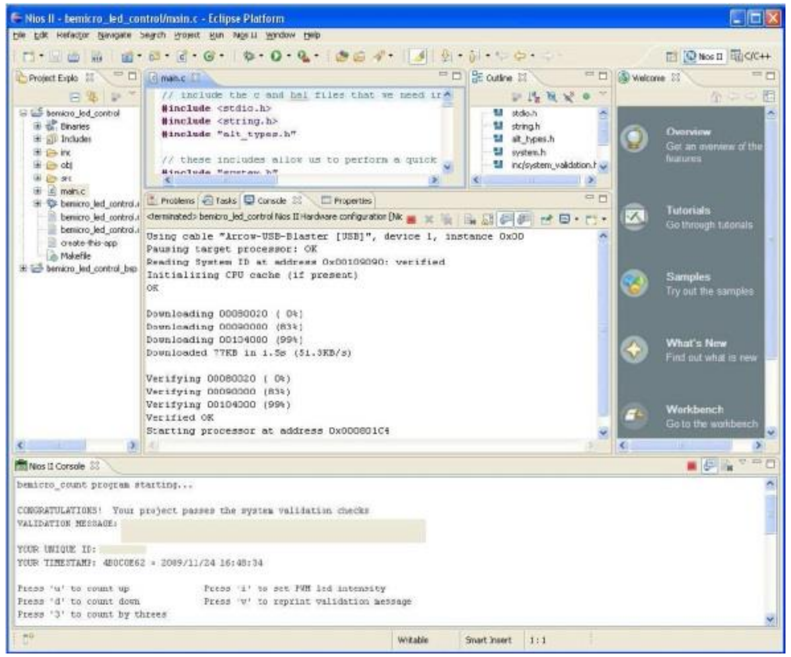
The case of Intel Cyclone® V SE 5CSEMA5F31C6N





Programming steps during labs

1. **FPGA Configuration**
2. Analysis of memory mapped device addresses
3. Preparation of the code in C language
4. Access to peripherals by memory addresses
5. Cross-compilation under IDE
6. Download by JTAG-USB
7. Debug by UART



```
// include the c and hal files that we need ir
#include <stdio.h>
#include <string.h>
#include "alt_types.h"

// these includes allow us to perform a quick
```

```
<terminated> bemicro_led_control Nios II Hardware configuration [Nk
Using cable "Arrow-USB-Blaster [USB]", device 1, instance 0x0D
Pausing target processor: OK
Reading System ID at address 0x00109090: verified
Initializing CPU cache (if present)
OK

Downloading 00080020 ( 0%)
Downloading 00090000 (83%)
Downloading 00104000 (99%)
Downloaded 77KB in 1.5s (51.3KB/s)

Verifying 00080020 ( 0%)
Verifying 00090000 (83%)
Verifying 00104000 (99%)
Verified OK
Starting processor at address 0x000801C4
```

```
bemicro_count program starting...

CONGRATULATIONS!  Your project passes the system validation checks
VALIDATION MESSAGE:

YOUR UNIQUE ID:
YOUR TIMESTAMP: 4B0C0E62 = 2009/11/24 16:48:34

Press 'u' to count up              Press 'i' to set PWM led intensity
Press 'd' to count down            Press 'v' to reprint validation message
Press '3' to count by threes
```

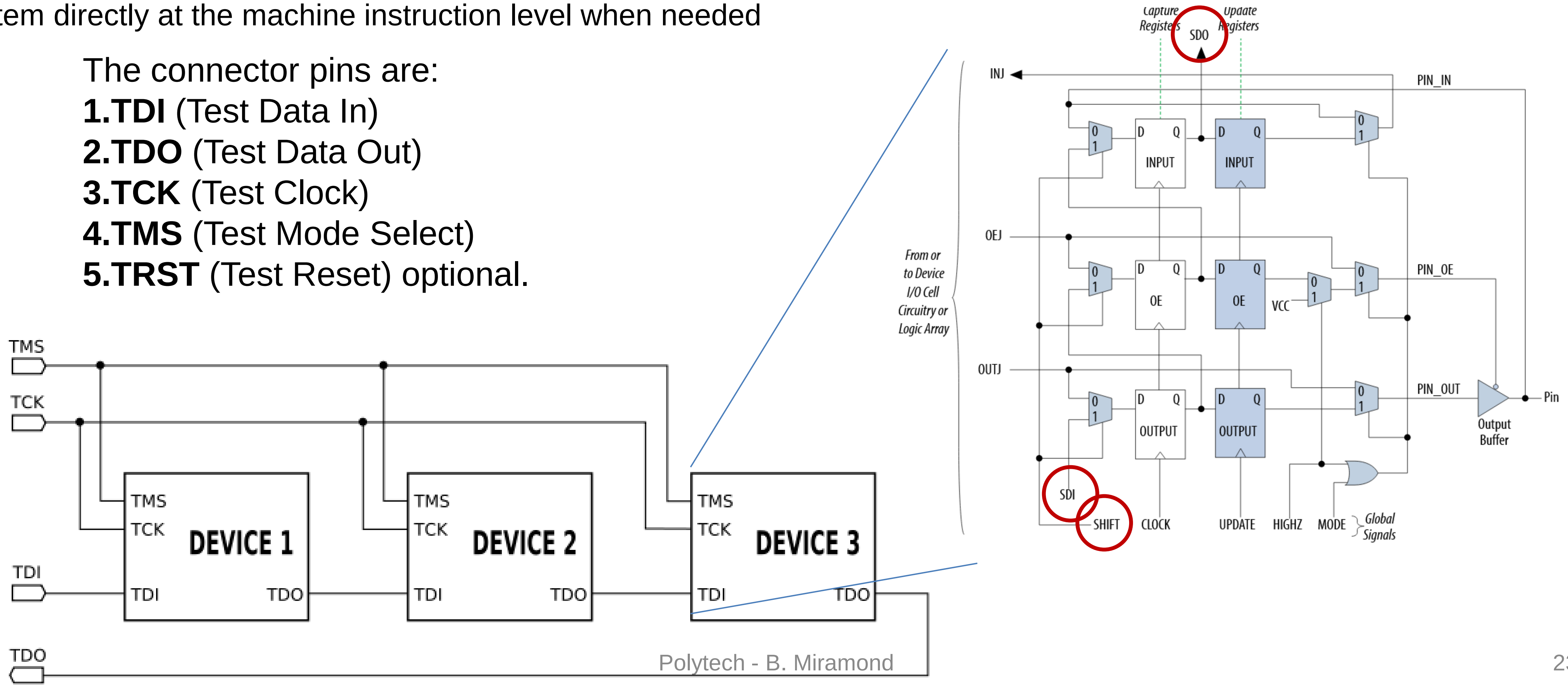
JTAG - Joint Test Action Group

- IEEE 1149.1 : *Standard Test Access Port and Boundary-Scan Architecture*

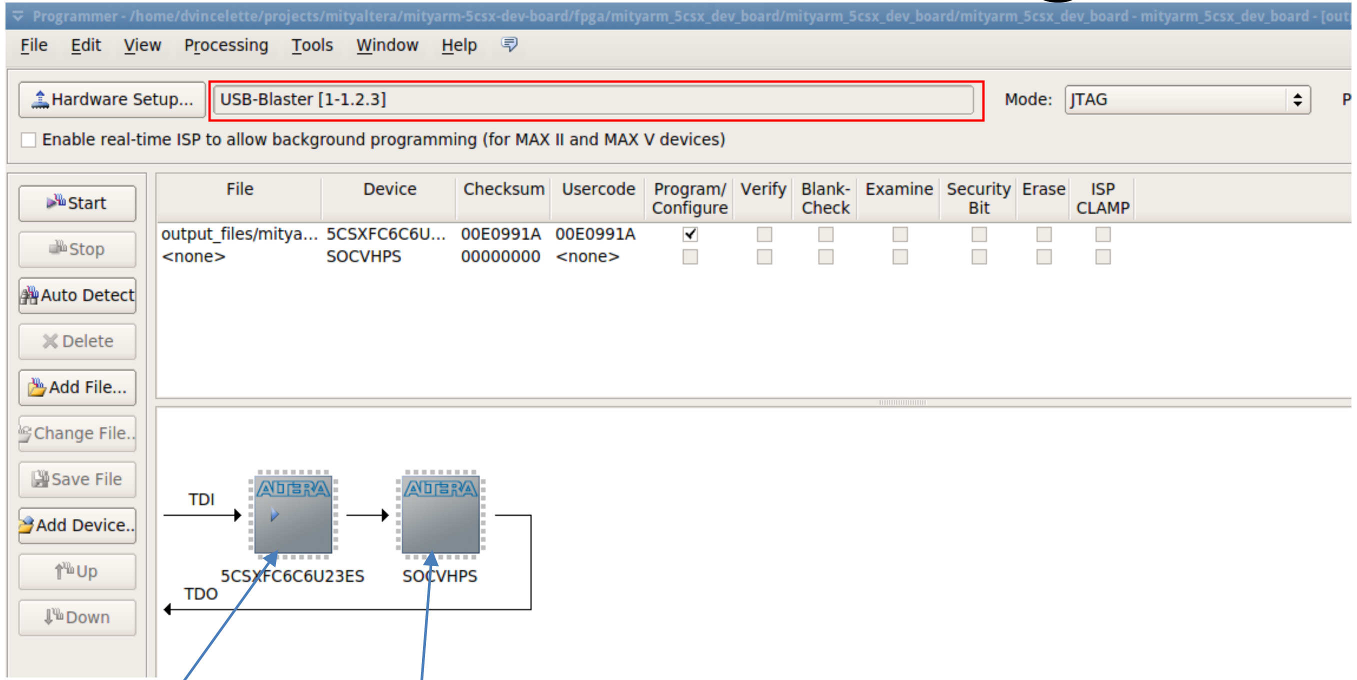
JTAG-based debugging is available from the very first instruction after CPU reset, letting it assist with development of early boot software which runs before anything is set up.. Those modules let software developers debug the software of an embedded system directly at the machine instruction level when needed

The connector pins are:

1. **TDI** (Test Data In)
2. **TDO** (Test Data Out)
3. **TCK** (Test Clock)
4. **TMS** (Test Mode Select)
5. **TRST** (Test Reset) optional.



Quartus JTAG Programmer



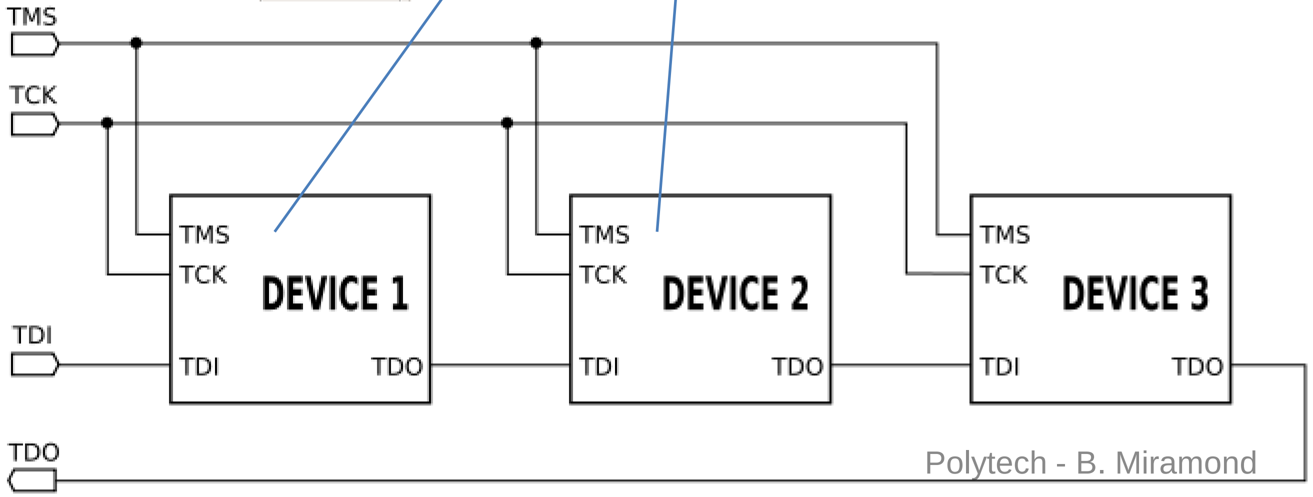
Programmer - /home/dvincelette/projects/mityaltera/mityarm-5csx-dev-board/fpga/mityarm_5csx_dev_board/mityarm_5csx_dev_board/mityarm_5csx_dev_board - mityarm_5csx_dev_board - [out

File Edit View Processing Tools Window Help

Hardware Setup... USB-Blaster [1-1.2.3] Mode: JTAG

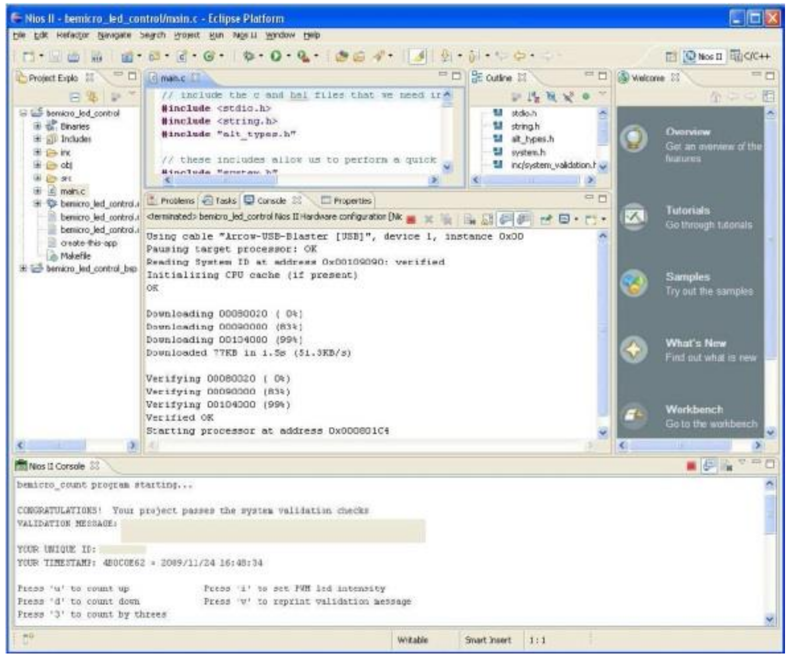
Enable real-time ISP to allow background programming (for MAX II and MAX V devices)

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine	Security Bit	Erase	ISP CLAMP
output_files/mitya...	5CSXFC6C6U...	00E0991A	00E0991A	✓	☐	☐	☐	☐	☐	☐
<none>	SOCVHPS	00000000	<none>	☐	☐	☐	☐	☐	☐	☐



Programming steps during labs

1. **FPGA Configuration**
2. Analysis of memory mapped device addresses
3. Preparation of the code in C language
4. Access to peripherals by memory addresses
5. **Cross-compilation under IDE**
6. Download by JTAG-USB
7. Debug by UART



```
// include the c and hal files that we need ir
#include <stdio.h>
#include <string.h>
#include "alt_types.h"

// these includes allow us to perform a quick
```

```
Using cable "Arrow-USB-Blaster [USB]", device 1, instance 0x0D
Pausing target processor: OK
Reading System ID at address 0x00109090: verified
Initializing CPU cache (if present)
OK

Downloading 00080020 ( 0%)
Downloading 00090000 (83%)
Downloading 00104000 (99%)
Downloaded 77KB in 1.5s (51.3KB/s)

Verifying 00080020 ( 0%)
Verifying 00090000 (83%)
Verifying 00104000 (99%)
Verified OK
Starting processor at address 0x000801C4
```

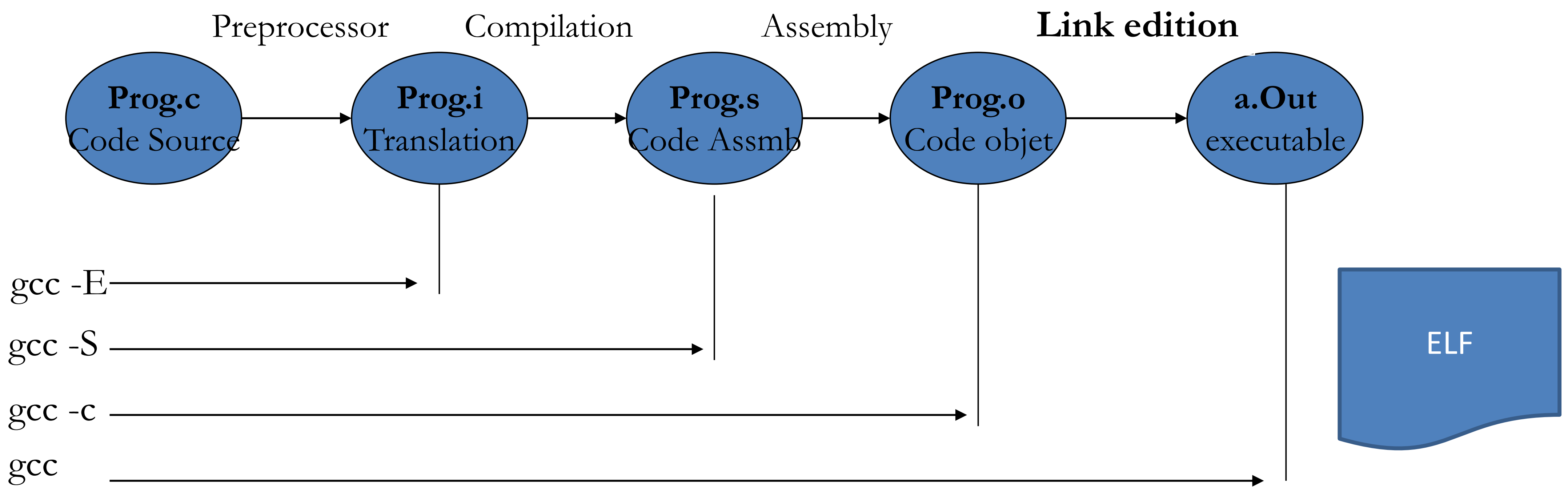
```
bemicro_count program starting...

CONGRATULATIONS!  Your project passes the system validation checks
VALIDATION MESSAGE:

YOUR UNIQUE ID:
YOUR TIMESTAMP: 4B0C0E62 = 2009/11/24 16:48:34

Press 'u' to count up           Press 'i' to set PWM led intensity
Press 'd' to count down         Press 'v' to reprint validation message
Press '3' to count by threes
```

Compilation steps with GNU C Compiler



several object file formats (sections) have been standardised:

- COFF (Common Object File Format) => unix
- **ELF** (Executable and Linkable Format) => linux
- PE (Portable Executable adapted from COFF) => windows 32 /64

Structure of an ELF object module

www.x86.org/ftp/manuals/tools/elf.pdf

- Header
  - File name,
  - Size,
  - Start address
- Object space (divided into sections)
  - Binary code
  - Data area
- Table of symbols
  - Symbols that can be used and to be satisfied
- Additional information
  - Authors, tools used, versions, environment...

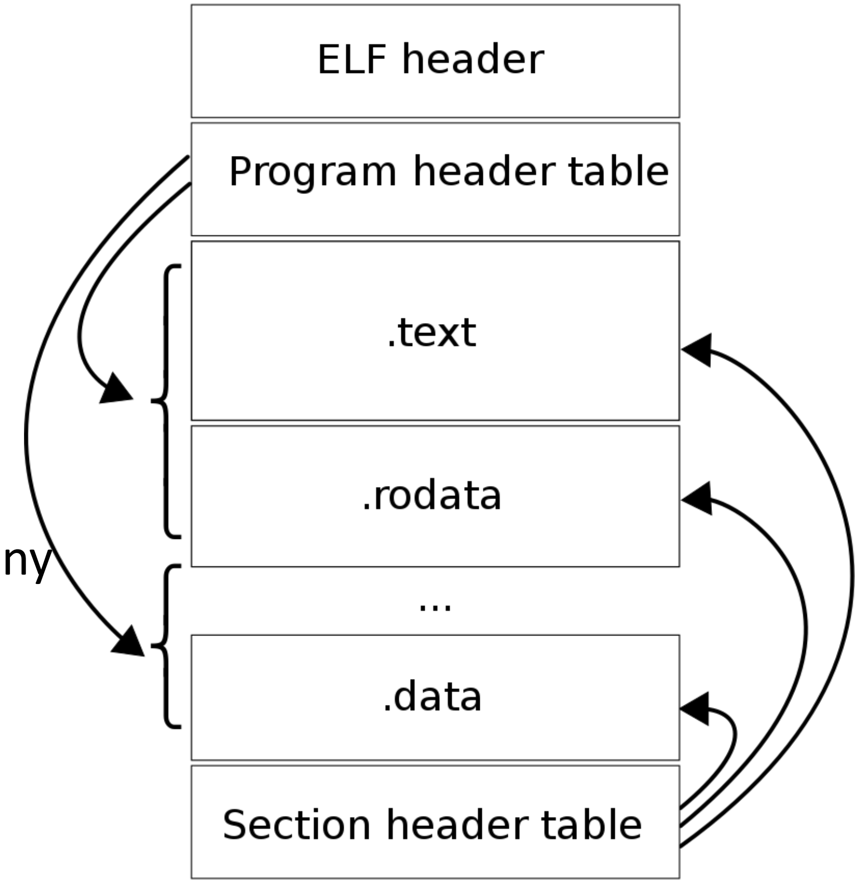
Types de contenu

The compiler organizes the program by content types called sections :

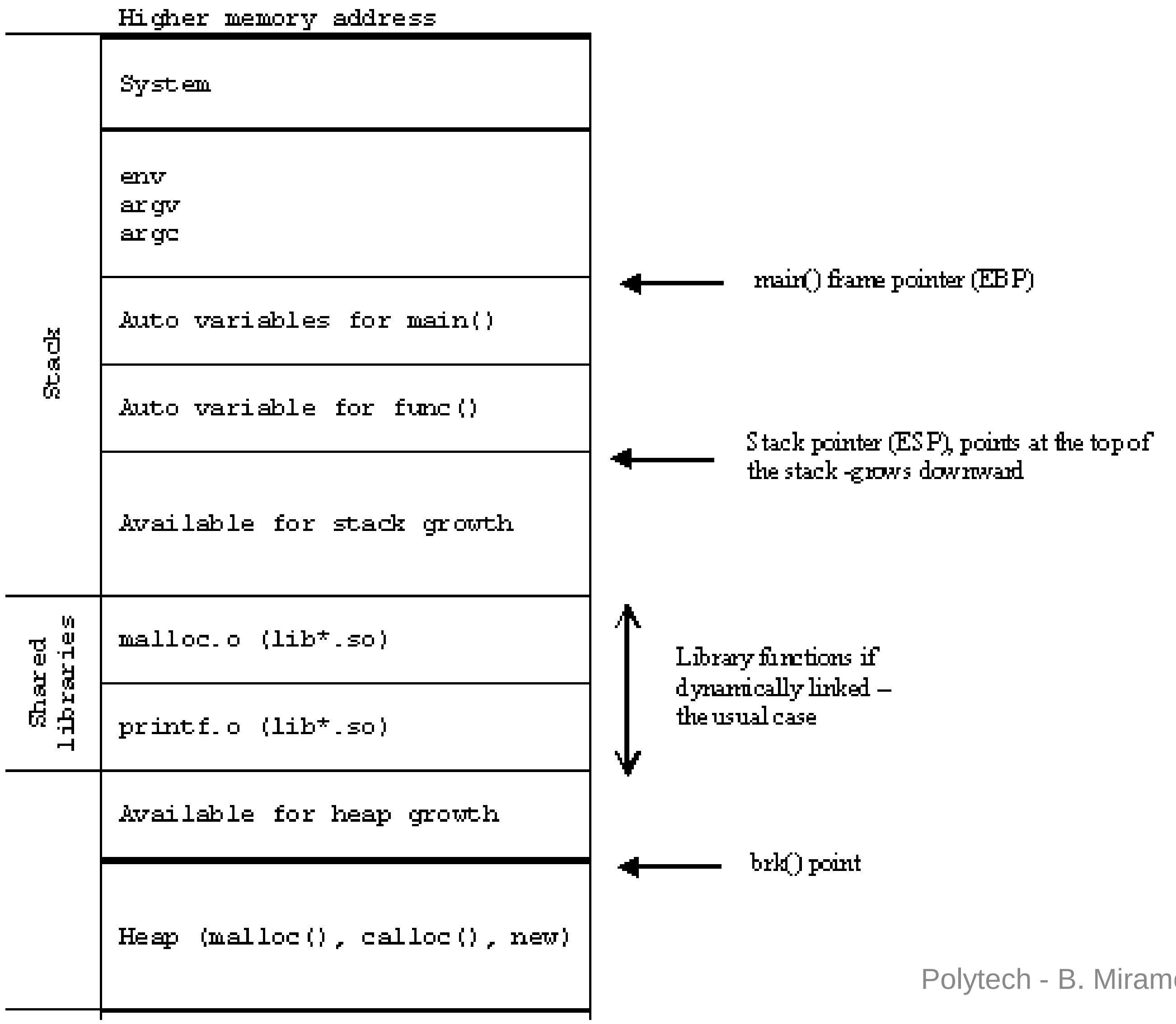
• .text/.code	= binary Instructions
• .data	= initialised data
• .bss	= (Block Started by Symbol) uninitialized data
• .rodata	= Read Only Data (Character Strings …)
• .comment	= comments
• .symtab	= symbols table

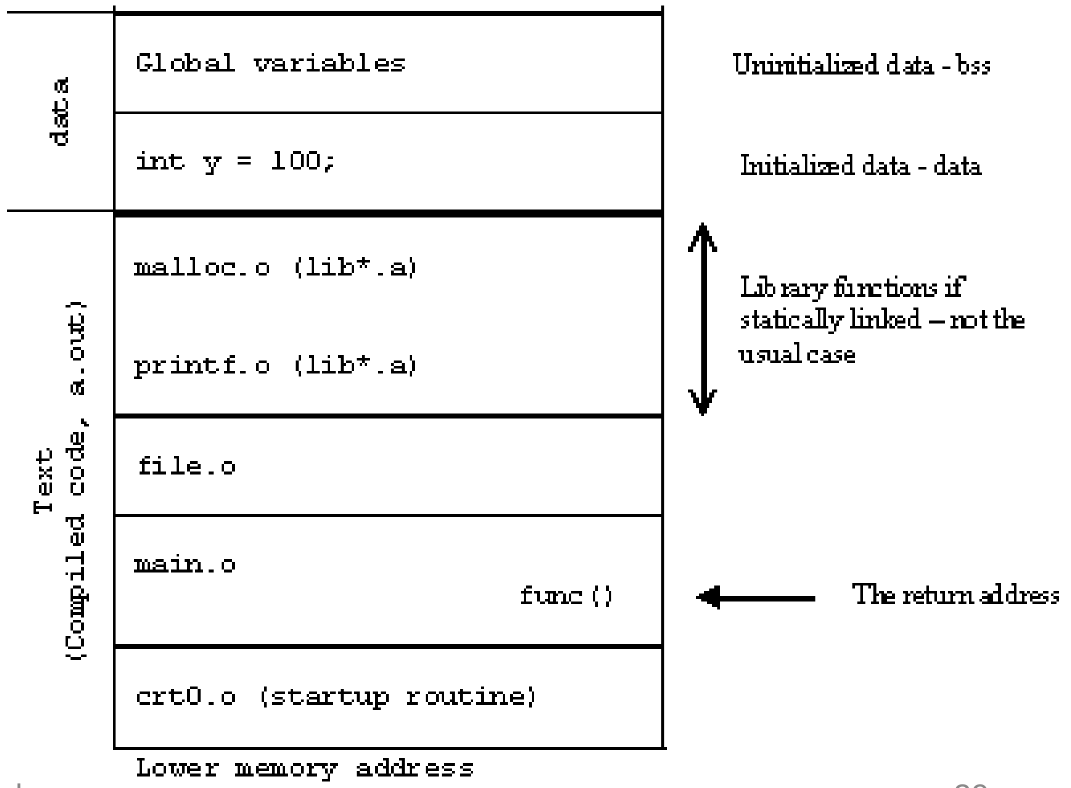
- …
- The ELF Standard allows you to define as many sections as you want with any name.

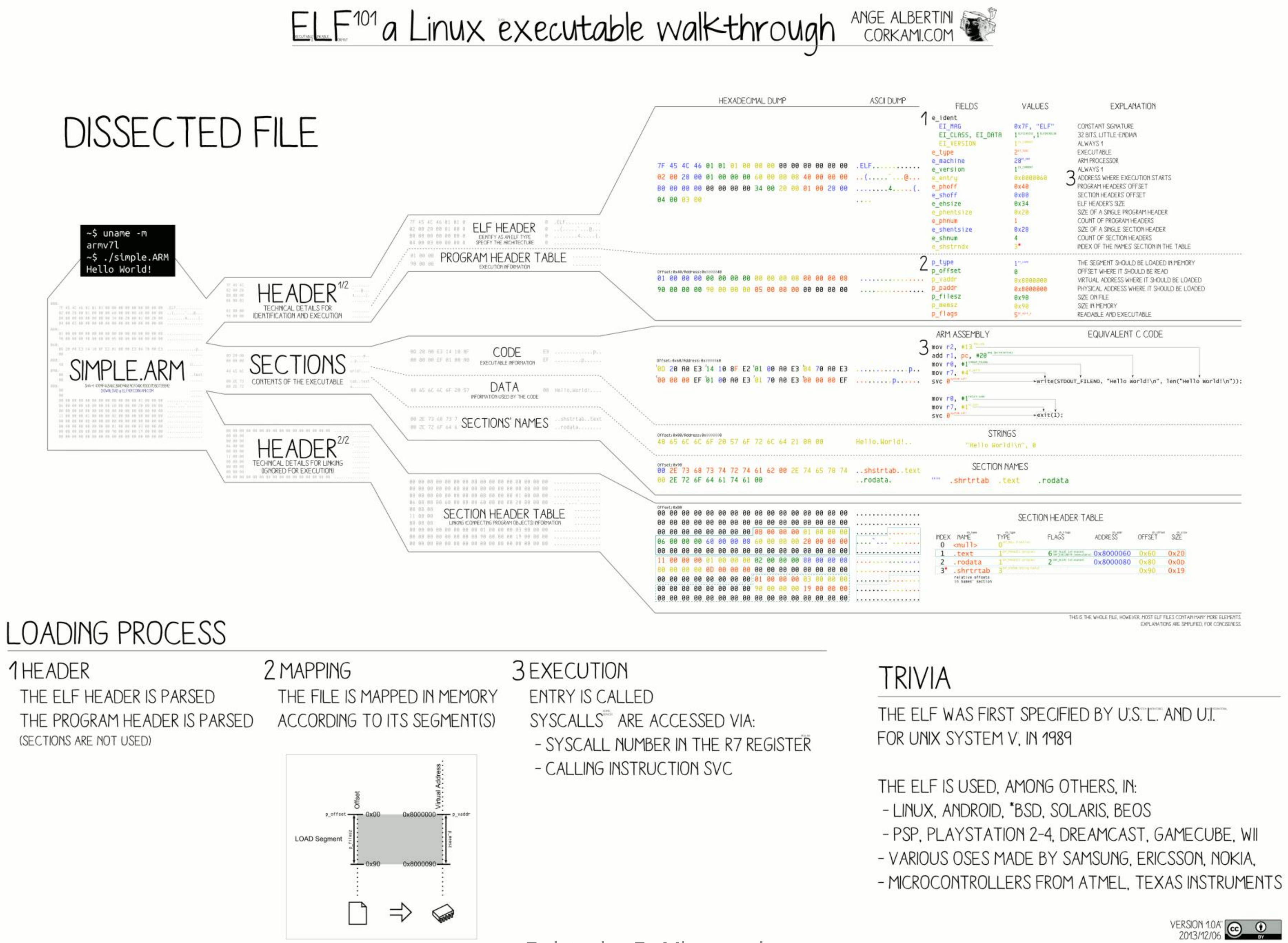
- Sw tools to read ELF files: **objdump** (binary files) or **readelf** (ELF only)



Linking sections in memory


Higher memory address
Stack
System
env
argv
argc
Auto variables for main()
Auto variable for func()
Available for stack growth
Shared libraries
malloc.o (lib*.so)
printf.o (lib*.so)
Available for heap growth
Heap (malloc(), calloc(), new)
main() frame pointer (EBP)
Stack pointer (ESP), points at the top of the stack -grows downward
Library functions if dynamically linked – the usual case
brk() point




ELF101 a Linux executable walkthrough
ANGE ALBERTINI CORKAMI.COM
DISSECTED FILE
~$ uname -m
armv7l
~$ ./simple.ARM
Hello World!
SIMPLE.ARM
HEADER 1/2
TECHNICAL DETAILS FOR IDENTIFICATION AND EXECUTION
ELF HEADER
IDENTIFY AS AN ELF TYPE
SPECIFY THE ARCHITECTURE
PROGRAM HEADER TABLE
EXECUTION INFORMATION
SECTIONS
CONTENTS OF THE EXECUTABLE
CODE
EXECUTABLE INFORMATION
DATA
INFORMATION USED BY THE CODE
SECTIONS' NAMES
HEADER 2/2
TECHNICAL DETAILS FOR LINKING (IGNORED FOR EXECUTION)
SECTION HEADER TABLE
LINKING (CONNECTING PROGRAM OBJECTS) INFORMATION
HEXADECIMAL DUMP
ASCII DUMP
FIELDS
VALUES
EXPLANATION
e_ident
EI_MAG
EI_CLASS, EI_DATA
EI_VERSION
e_type
e_machine
e_version
e_entry
e_phoff
e_shoff
e_ehsize
e_phentsize
e_phnum
e_shentsize
e_shnum
e_shstrndx
0x7F, "ELF"
CONSTANT SIGNATURE
32 BITS, LITTLE-ENDIAN
ALWAYS 1
EXECUTABLE
ARM PROCESSOR
ALWAYS 1
ADDRESS WHERE EXECUTION STARTS
PROGRAM HEADERS' OFFSET
SECTION HEADERS' OFFSET
ELF HEADER'S SIZE
SIZE OF A SINGLE PROGRAM HEADER
COUNT OF PROGRAM HEADERS
SIZE OF A SINGLE SECTION HEADER
COUNT OF SECTION HEADERS
INDEX OF THE NAMES' SECTION IN THE TABLE
p_type
p_offset
p_vaddr
p_paddr
p_filesz
p_memsz
p_flags
THE SEGMENT SHOULD BE LOADED IN MEMORY
OFFSET WHERE IT SHOULD BE READ
VIRTUAL ADDRESS WHERE IT SHOULD BE LOADED
PHYSICAL ADDRESS WHERE IT SHOULD BE LOADED
SIZE ON FILE
SIZE IN MEMORY
READABLE AND EXECUTABLE
ARM ASSEMBLY
EQUIVALENT C CODE
mov r2, #13
add r1, pc, #20
mov r0, #1
mov r7, #4
svc 0
write(STDOUT_FILENO, "Hello World!\n", len("Hello World!\n"));
mov r0, #1
mov r7, #1
svc 0
exit(1);
STRINGS
"Hello World!\n", 0
SECTION NAMES
"" .shrtrtab .text .rodata
SECTION HEADER TABLE
INDEX NAME TYPE FLAGS ADDRESS OFFSET SIZE
0 <null> 0
1 .text 1 6 0x8000060 0x60 0x20
2 .rodata 1 2 0x8000080 0x80 0x0D
3 .shrtrtab 3 0x90 0x19
THIS IS THE WHOLE FILE. HOWEVER, MOST ELF FILES CONTAIN MANY MORE ELEMENTS.
EXPLANATIONS ARE SIMPLIFIED FOR CONCISENESS.
LOADING PROCESS
1 HEADER
THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)
2 MAPPING
THE FILE IS MAPPED IN MEMORY ACCORDING TO ITS SEGMENT(S)
p_offset
Offset
0x00
0x90
0x8000000
0x8000090
Virtual Address
p_vaddr
LOAD Segment
p_filesz
p_memsz
3 EXECUTION
ENTRY IS CALLED
SYSCALLS ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE R7 REGISTER
- CALLING INSTRUCTION SVC
TRIVIA
THE ELF WAS FIRST SPECIFIED BY U.S. L. AND U.I. FOR UNIX SYSTEM V, IN 1989
THE ELF IS USED, AMONG OTHERS, IN:
- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS
VERSION 1.0A 2013/12/06

https://upload.wikimedia.org/wikipedia/commons/e/e4/ELF_Executable_and_Linkable_Format_diagram_by_Ange_Albertini.png

Variable allocation

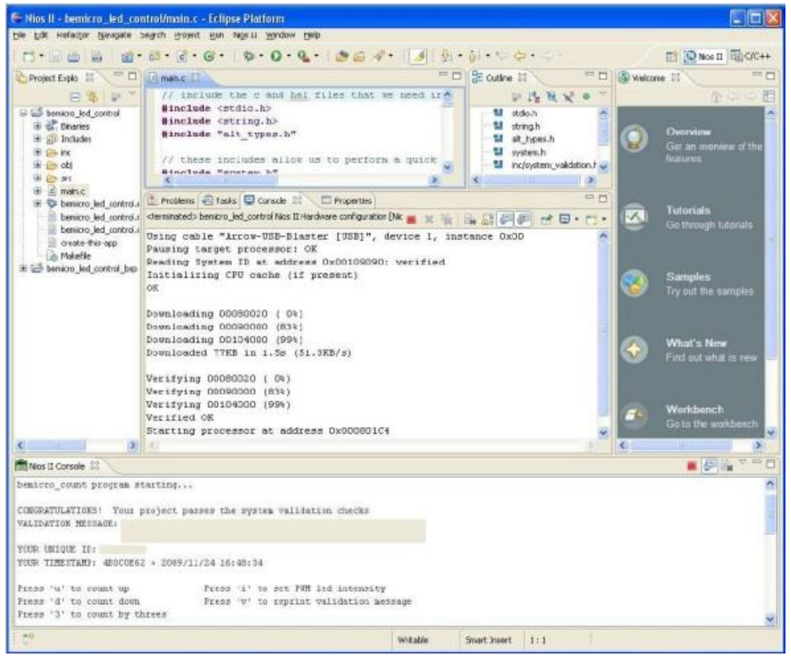
			.data	.bss	.rodata	Pile
Global	static	Initialzed				
		non init.				
	dyna	init.			?	
		non init				
Local	static	Init				
		non init				
	dyna	init				
		non init				
G/L	const					

Variable allocation

			.data	.bss	.rodata	Pile
Global	static	initialized	🟥			
		non init.		🟥		
	dyna	init.	🟥			
		non init		🟥		
Local	static	init	🟥			
		non init		🟥		
	dyna	init				🟥
		non init				🟥
G/L	const				🟥	

Programming steps during labs

1. FPGA Configuration
2. **Analysis of memory mapped device addresses**
3. **Preparation of the code in C language**
4. **Access to peripherals by memory addresses**
5. Cross-compilation under IDE
6. Download by JTAG-USB
7. Debug by UART



```
// include the c and hal files that we need ir
#include <stdio.h>
#include <string.h>
#include "alt_types.h"

// these includes allow us to perform a quick
```

```
Using cable "Arrow-USB-Blaster [USB]", device 1, instance 0x00
Pausing target processor: OK
Reading System ID at address 0x00109090: verified
Initializing CPU cache (if present)
OK

Downloading 00080020 ( 0%)
Downloading 00090000 (83%)
Downloading 00104000 (99%)
Downloaded 77KB in 1.5s (51.3KB/s)

Verifying 00080020 ( 0%)
Verifying 00090000 (83%)
Verifying 00104000 (99%)
Verified OK
Starting processor at address 0x000801C4
```

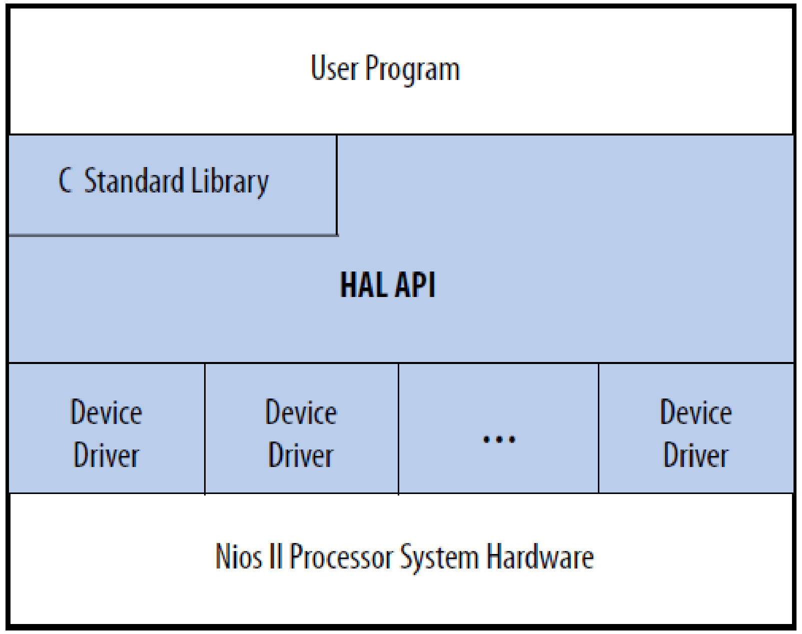
```
bemicro_count program starting...

CONGRATULATIONS!  Your project passes the system validation checks
VALIDATION MESSAGE:

YOUR UNIQUE ID:
YOUR TIMESTAMP: 4B0C0E62 = 2009/11/24 16:48:34

Press 'u' to count up              Press 'i' to set PWM led intensity
Press 'd' to count down            Press 'v' to reprint validation message
Press '3' to count by threes
```

HAL the layer to communicate with peripherals

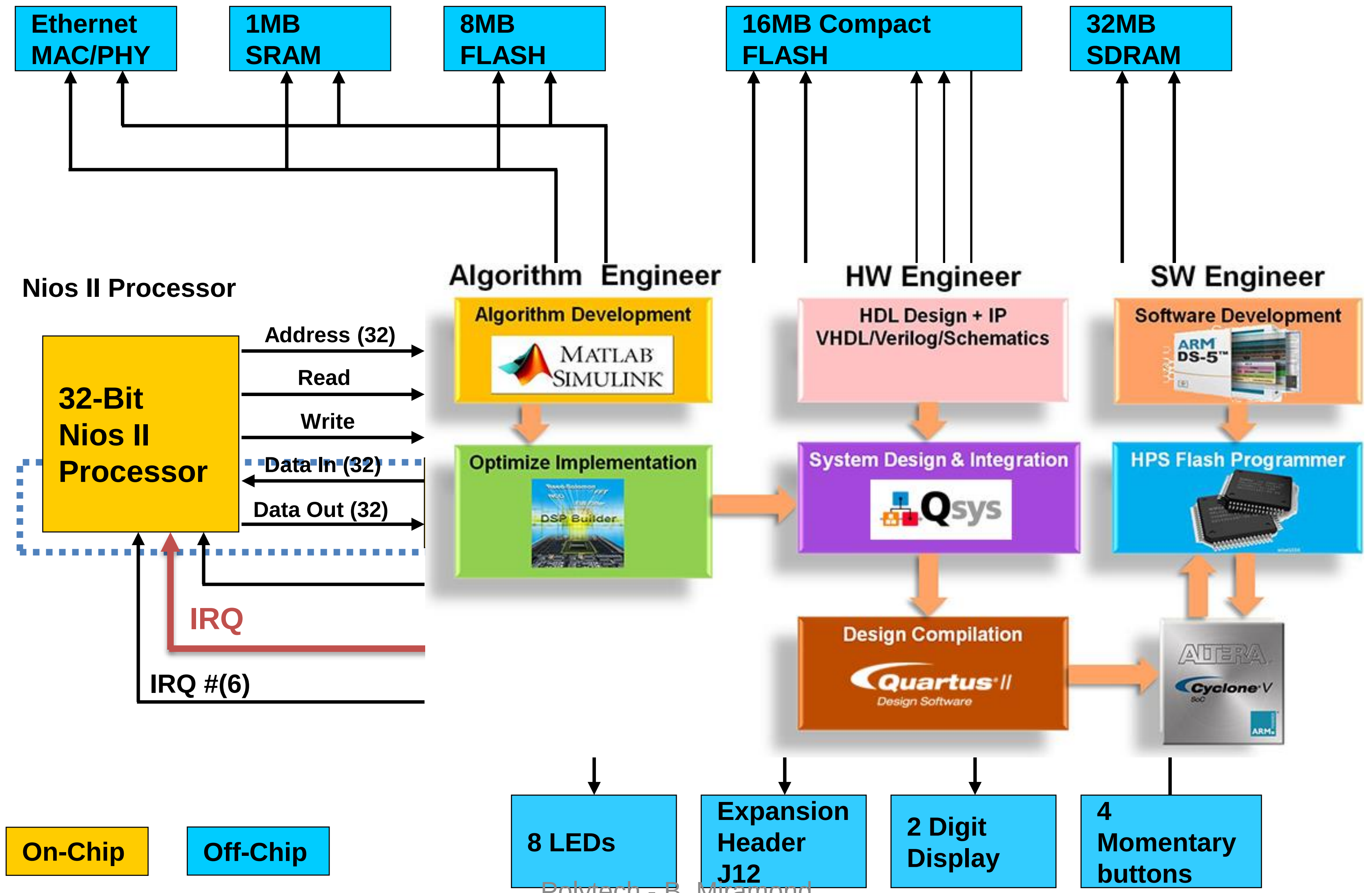


The application program (in C) is based on 2 software layers:

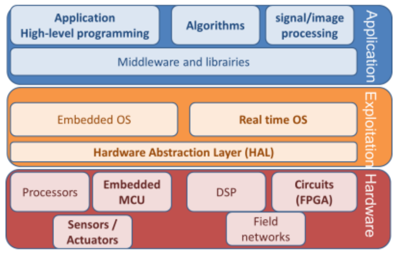
- HAL (Hardware Abstraction Layer) contains the declaration of functions for accessing hardware resources (drivers, processor configuration, Hw initialization, etc.).
  - Example of reading/writing in peripherals:
    - #include "altera_avalon_pio_regs.h“
    - IORD_ALTERA_AVALON_PIO_DATA(B (address, data)
    - data = IOWR_ALTERA_AVALON_PIO_DATA (address)
- The BSP (Board Support Package) contains the specific implementations of the peripherals used on the board

A micro-controller on FPGA

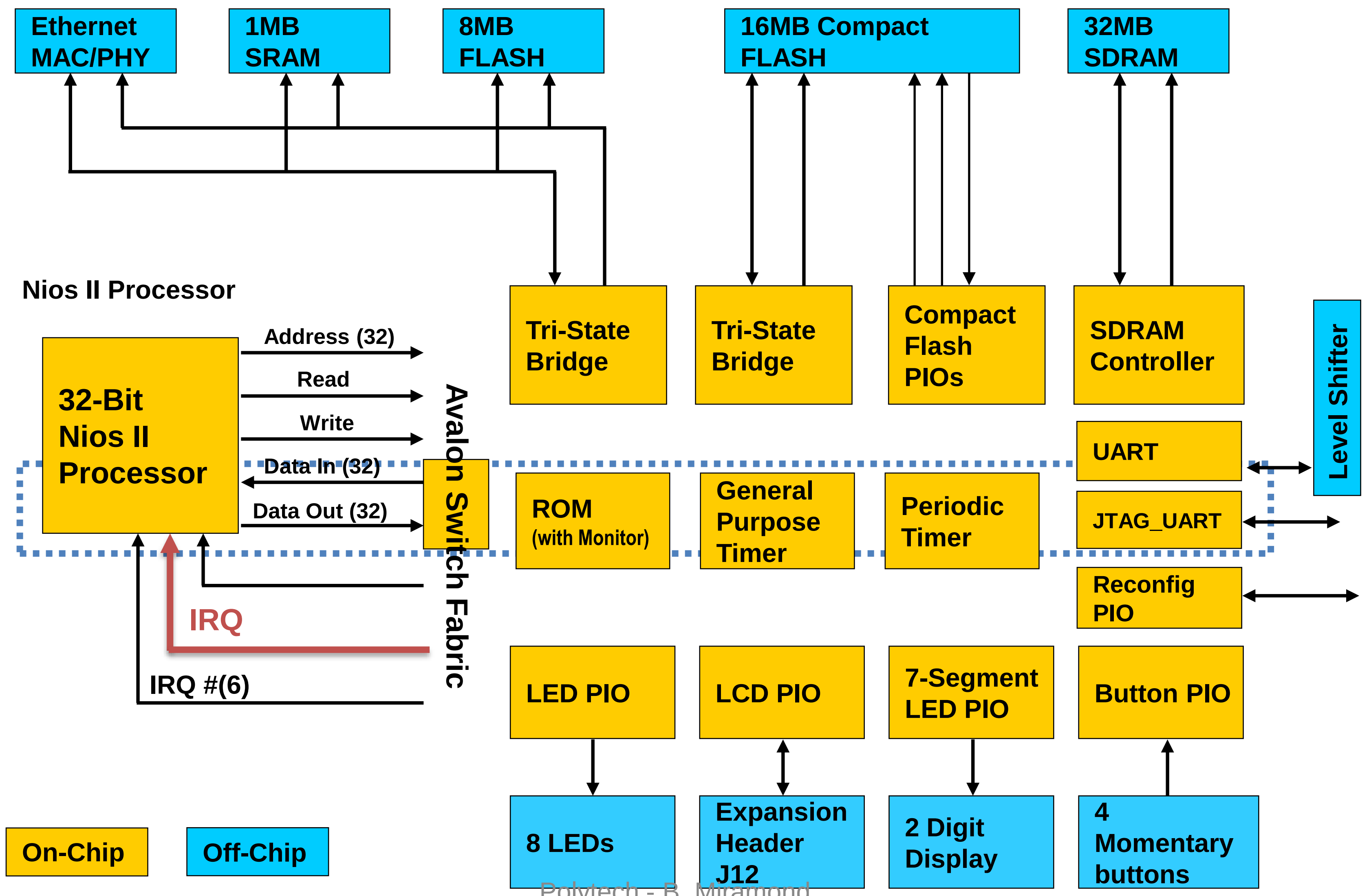
(details in last year course)



Slide 14

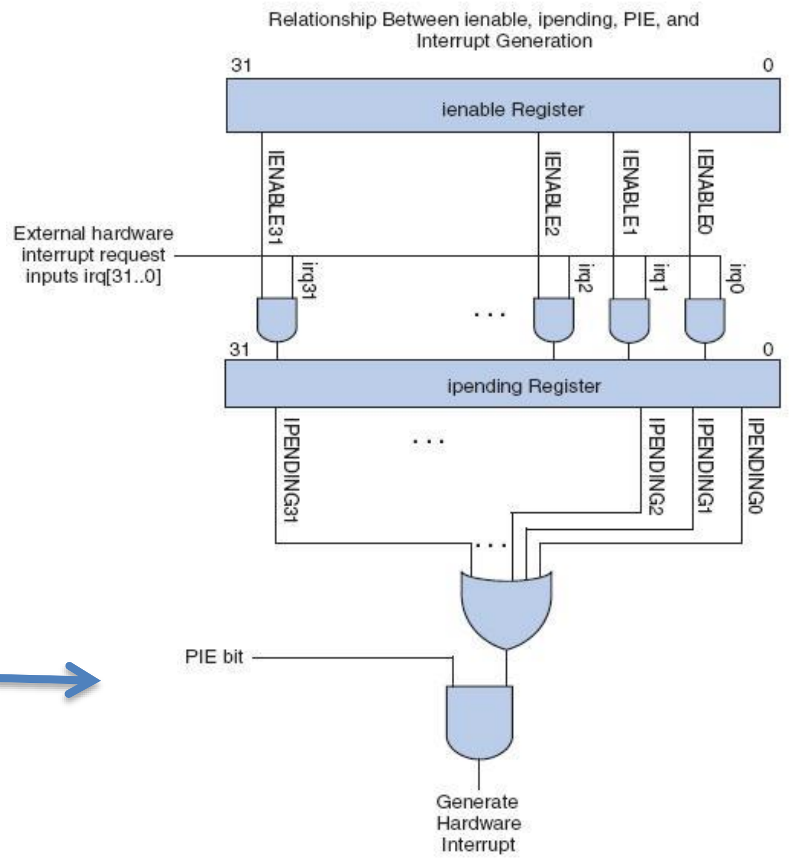


A micro-controller on FPGA

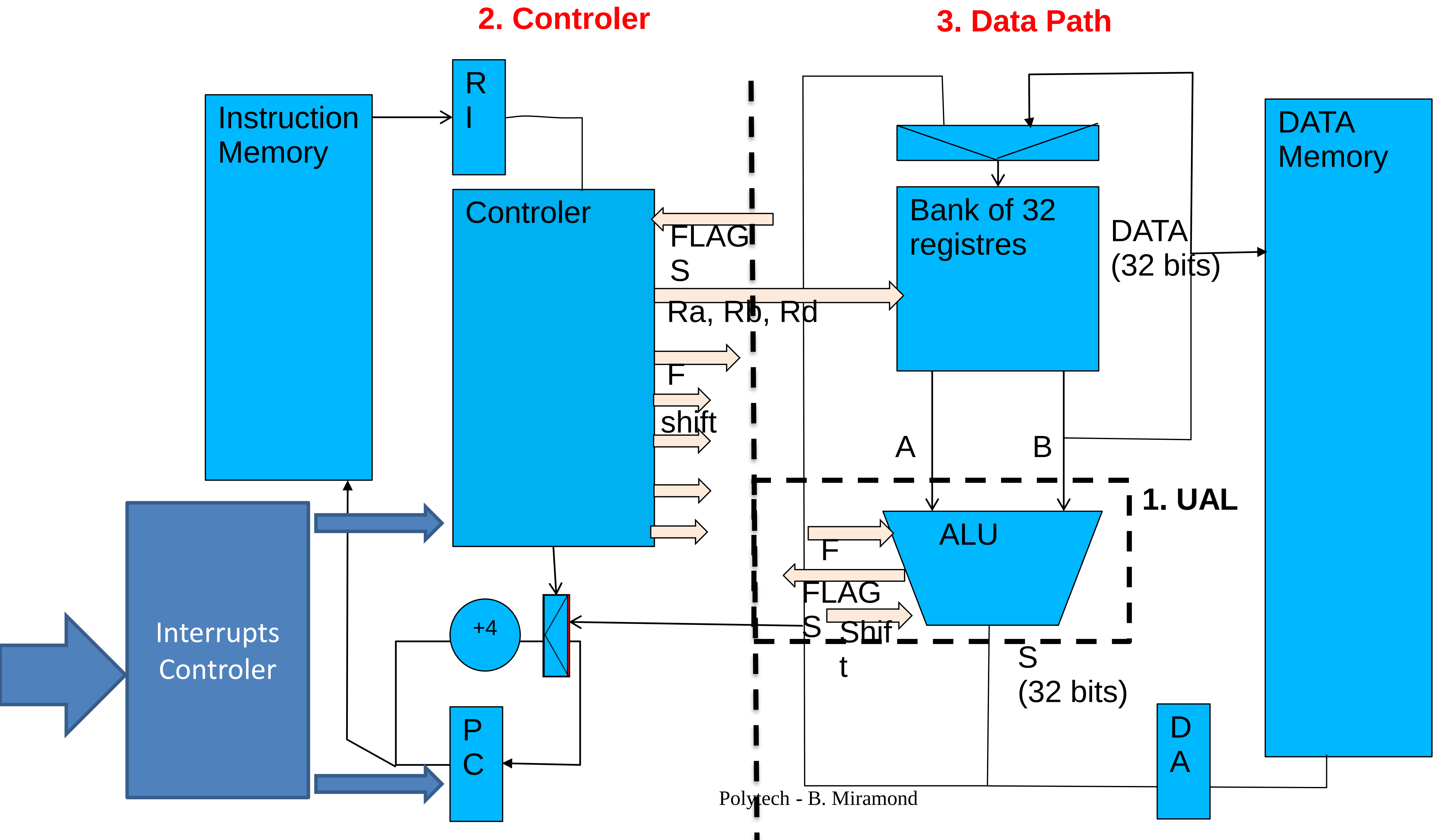


Programming with interrupts

- **In a SoC, only the processor can initiate communications:**
  - With memory, peripherals,
  - He is designated as Master
  - Peripherals like Slave.
- **An interrupt is an asynchronous means of communication to indicate to the processor that an event has occurred and that it can therefore initiate the appropriate communication.**
- **RISC processors have 32 interrupt lines that**
  - Are prioritized (from 1 to 32),
  - Can be masked,
  - Must be programmed
- **The other method of accessing an asynchronous device is Polling,**
  - where the processor reads at regular intervals.
  - Processor time wasted + less reactive

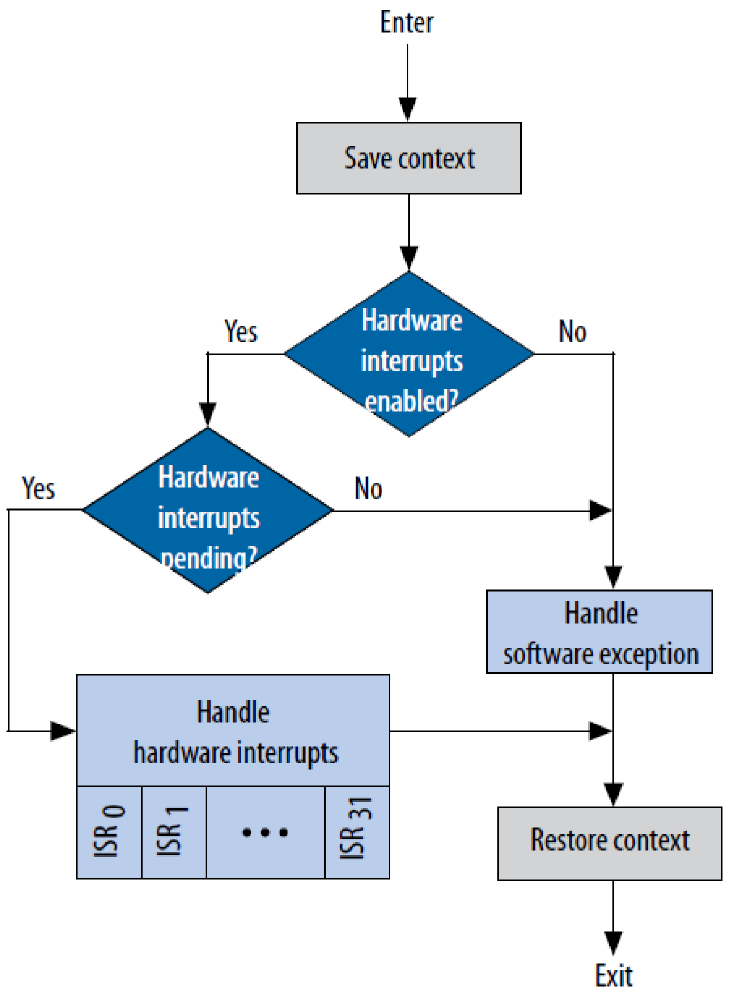

Relationship Between ienable, ipending, PIE, and Interrupt Generation
31
0
ienable Register
IENABLE31
IENABLE2
IENABLE1
IENABLE0
External hardware interrupt request inputs irq[31..0]
irq31
irq2
irq1
irq0
31
0
ipending Register
IPENDING31
IPENDING2
IPENDING1
IPENDING0
PIE bit
Generate Hardware Interrupt

General architecture of the MCU – PARM example



Behaviour of hw interrupts

(hardwired in the MCU)

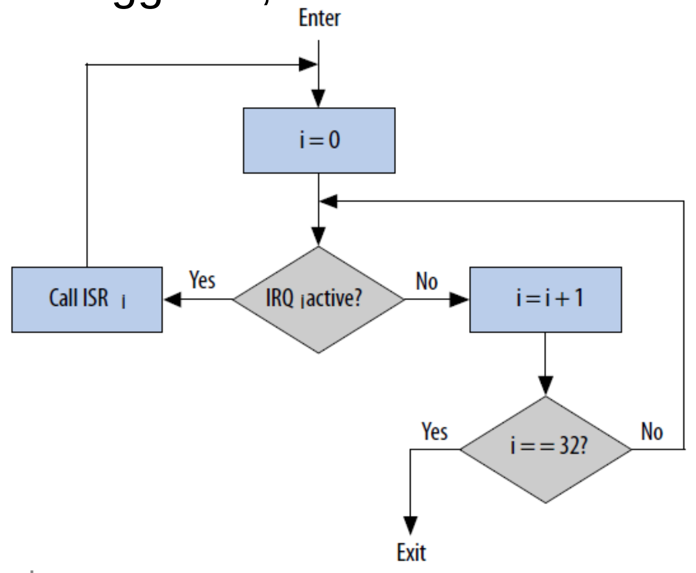


Enter
Save context
Hardware interrupts enabled?
Yes
No
Hardware interrupts pending?
Yes
No
Handle software exception
Handle hardware interrupts
ISR 0
ISR 1
• • •
ISR 31
Restore context
Exit

Masking interrupts

The interruption causes a program sequence to be interrupted:

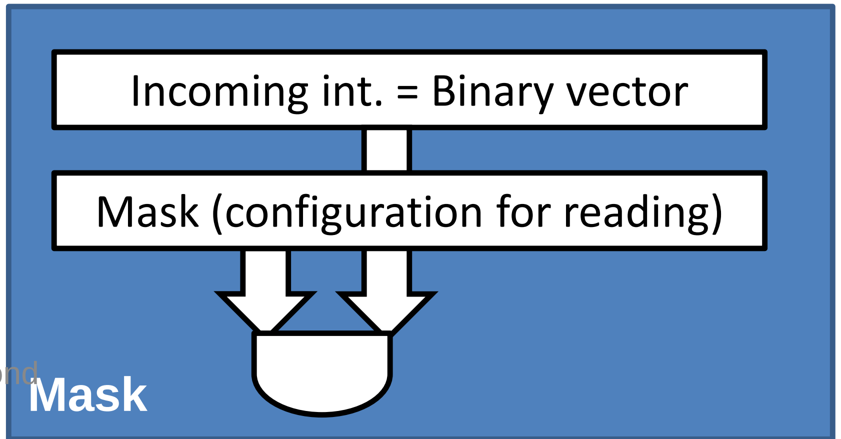
- It causes a change of context to execute the interrupt routine.
- To do this, it empties the pipeline and saves the CPU registers (save context).
- It calls the ISR (jump) and executes its code.
- When ISR returns, the context is restored and program execution resumes.
- If 2 interruptions are triggered, ISRs are executed in order of priority



Priorities of interrupts

Programmation of interrupts

- IRQ programming is done in 4 steps:
  1. Hiding interruptions (allows or not all IRQs),
  2. Initializes the status of the associated device,
  3. Record an interrupt routine that associates an IRQ number with a jump address,
  4. Coding the Interrupt Service Routine (ISR)
- The code of the routine is a critical (uninterruptible) section. It is therefore subject to several rules:
  - Relatively short code, limiting the interrupt latency,
  - No blocking calls,
  - Reset the interrupt source


Incoming int. = Binary vector
Mask (configuration for reading)
Mask

```
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

volatile int edge_capture;

static void init_button_pio()
{
/* Recast the edge_capture pointer to match the alt_irq_register() function prototype. */
void* edge_capture_ptr = (void*) &edge_capture;
/* Enable all 4 button interrupts. */
IOWR_ALT_UP_PARALLEL_PORT_INTERRUPT_MASK(ADDRESS_BASE, 0xf);
/* Reset the edge capture register. */
IOWR_ALT_UP_PARALLEL_PORT_EDGE_CAPTURE(ADDRESS_BASE, 0x0);
/* Register the ISR. */
alt_irq_register( NUMERO_IRQ,edge_capture_ptr,handle_button_interrupts );
}
```

Variable defined as **volatile**
Fonction defined as static

The address of the peripheral is defined as a macro in System.h

1

2

3

4

Volatile and static keywords

- **Volatile**
  - Used in the case of variables whose value can change spontaneously:
    - without processor action, memory mapped devices
    - By another task in case of multithreaded software
  - This prefix tells the compiler to avoid optimizations that generate a systematic memory read instruction.
    - More often used in embedded programming

- **Static**
  - On a variable, keeps the value of the local variable (allocation out of stack)
  - On a function limits the definition of the symbol inside the object (file)

Example 1: peripherals

- Example of a 8-bit register mapper at address 0w1234. The code test if the register is non-zero
  - Uint *ptr= (Uint *) 0X1234;
  - // wait until non-zero
  - While (*ptr==0);
  - // do processing after the loop
- Compiler genereates
  - Mov ptr, #1234
  - Mov a, @ptr
  - Loop:
  - Bz loop // infinite loop
- By declaring ptr as volatile, its value is read each time it is used
  - Mov ptr, #1234
  - Loop :
  - Mov a, @ptr
  - Bz loop

Example 2: interruption handler

- Test end of message on a serial connection (ETX)

```
int etx_rcvd = false;
void mail(){
        …
        while(!etx_rcvd){
        // process message
        }
        // unused code
        // end of communication
}
```

```
void rx_isr(void){
…
If (ETX == rx_char)
        etx_rcvd = TRUE;
…
}
```

Problem: the compiler does not understand that the variable can be changed spontaneously in another function, then the loop is considered as infinite

And the code after the loop is not generated in the binary file !

```
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

static void handle_button_interrupts(void* context, alt_u32 id)
{
/* Cast context to edge_capture's type. It is important that this
be declared volatile to avoid unwanted compiler optimization. */
volatile int* edge_capture_ptr = (volatile int*) context;
/* Read the edge capture register on the button PIO. Store value. */
*edge_capture_ptr =
            IORD_ALT_UP_PARALLEL_PORT_EDGE_CAPTURE(ADDRESS_BASE);
/* Write to the edge capture register to reset it. */
IOWR_ALT_UP_PARALLEL_PORT_EDGE_CAPTURE(ADDRESS_BASE, 0);
/* Read the PIO to delay ISR exit. This is done to prevent a
spurious interrupt in systems with high processor -> pio
latency and fast interrupts. */
IORD_ALT_UP_PARALLEL_PORT_EDGE_CAPTURE(ADDRESS_BASE);
}
```

4

Address defined in system.h

First lab on DE1-SoC boards

- Goal :
  - Cross compilation on embedded SoC target
  - Peripherals programming
  - Programming of interrupts
- To prepare:
  - Install Quartus Prime 16.1 Lite edition from the following links:
    - Linux :

      https://download.altera.com/akdlm/software/acdsinst/16.1/196/ib_installers/QuartusLiteSetup-16.1.0.196-linux.run
    - Windows :

      https://download.altera.com/akdlm/software/acdsinst/16.1/196/ib_installers/QuartusLiteSetup-16.1.0.196-windows.exe
    - Prise en charge Cyclone V circuits :

      https://download.altera.com/akdlm/software/acdsinst/16.1/196/ib_installers/cyclonev-16.1.0.196.qdz