

Capteurs/actionneurs

Polytech Nice Sophia - SI4

B. Miramond

Progression des notions d'embarqué

- SI3 – Principes d'exécution, micro-contrôleur, architecture
- SI4 – Capteurs/actionneurs, traitement temps réel
- SI5 – Conception conjointe logicielle/matérielle, SoC, FPGA, Parallélisme

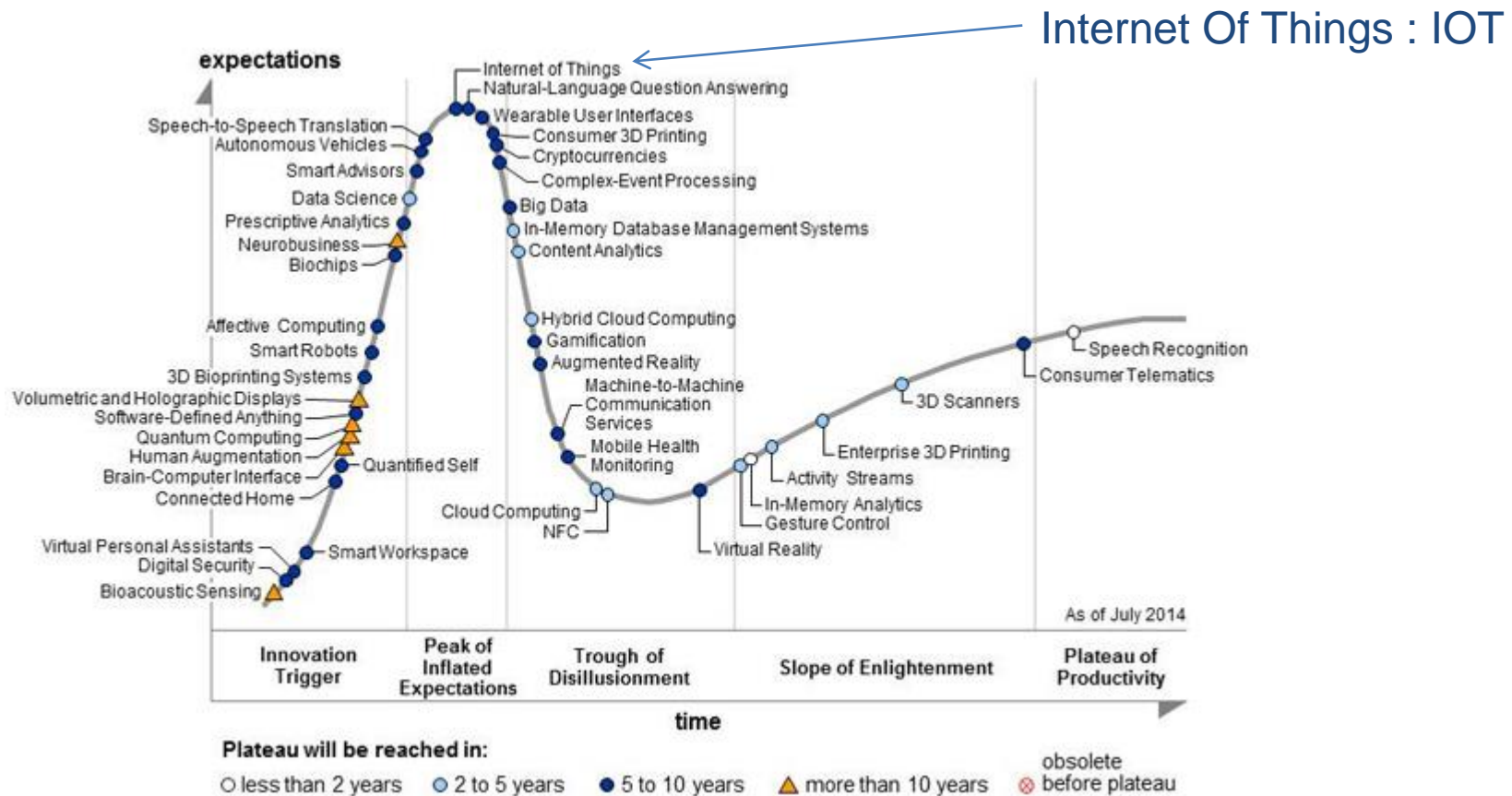
Les mots clés

- Systèmes embarqués
- Systèmes temps réel
- Micro-contrôleurs
 - ARM / MIPS / ATmega / PIC / AVR / x86
- Capteurs / actionneurs

Organisation du module

- 12 séances de cours de 1 heure
- 12 séances de TD de 3 heures
- 3 notes : contrôle 1, TP, contrôle 2

Des systèmes embarqués aux objets connectés



<http://www.gartner.com/technology/research/hype-cycles/>

[illegible]

Kit de développement embarqués



BeagleBoard

Architecture

ARMv7I Cortex-A8

Processor

TI OMAP 3530 720MHz

RAM

256MB

Raspberry Pi

Architecture

ARMv6I

Processor

Broadcom BCM2835 700MHz

RAM

512MB



Raspberry Pi 2

Architecture

ARMv7I Cortex-A7

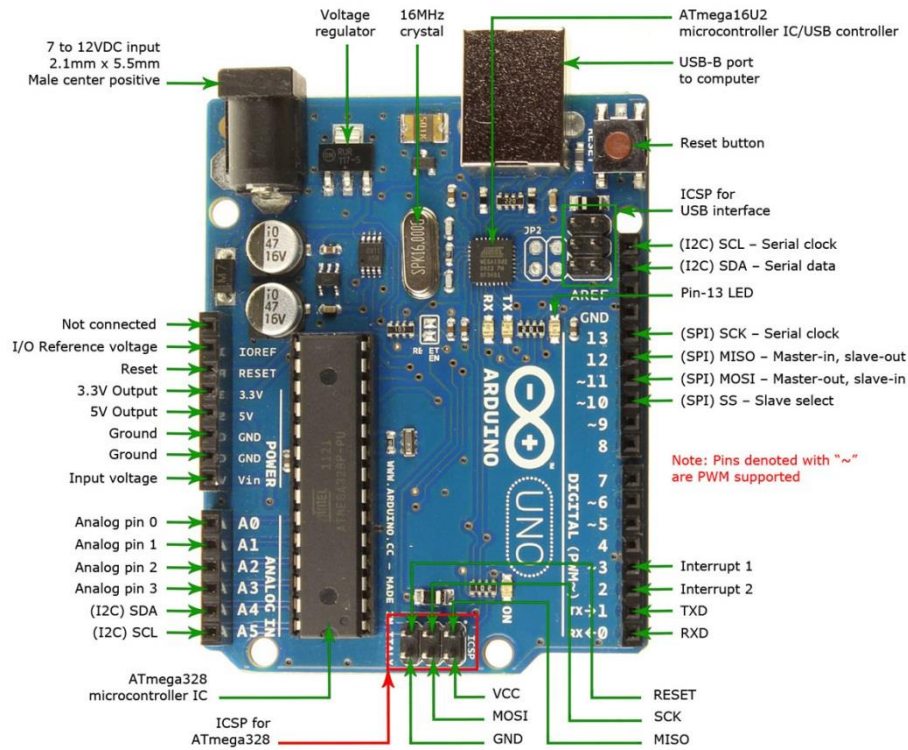
Processor

Broadcom BCM2836 900MHz

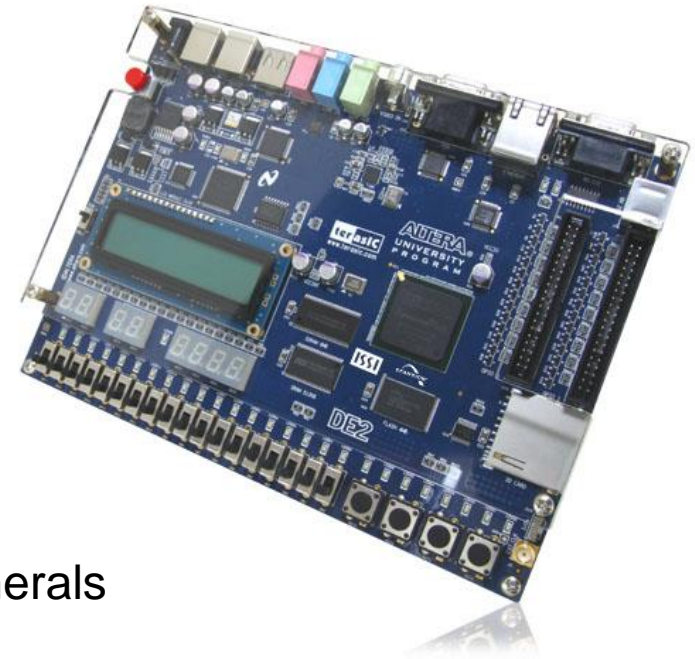
RAM

1024MB

Kits de développement embarqués

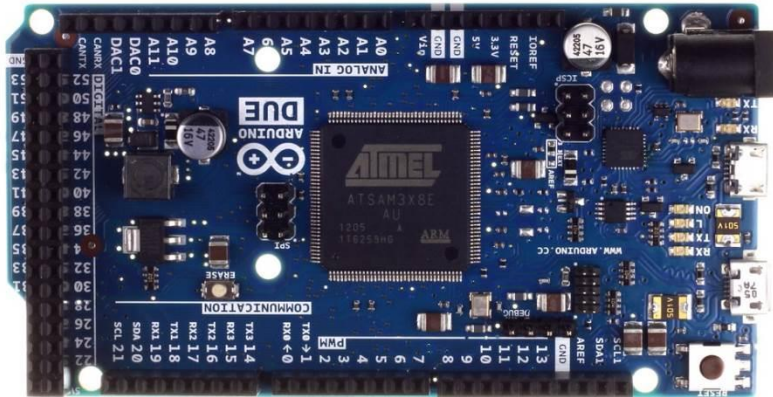


ATmega 328
32 KB Flash
2KB SRAM



FPGA Cyclone II
4MB Flash
8MB SRAM
Integrated peripherals

les cartes Arduino à base de ARM



Arduino Due

Architecture

ARMv7 Cortex-M3

Processor

Atmel SAM3X8E – 84MHz

RAM

96KB

Arduino Zero

Architecture

ARMv7 Cortex-M0

Processor

Atmel SAMD21 – 48MHz

RAM

32 KB



CARTE Intel Genuino 101

Main page : <https://www.arduino.cc/en/Main/ArduinoBoard101>

Microcontroller	Intel Curie
Operating Voltage	3.3V (5V tolerant I/O)
Input Voltage (recommended)	7-12V
Input Voltage (limit)	7-20V
Digital I/O Pins	14 (of which 4 provide PWM output)
PWM Digital I/O Pins	4
Analog Input Pins	6
DC Current per I/O Pin	20 mA
Flash Memory	196 kB
SRAM	24 kB
Clock Speed	32MHz
Features	Bluetooth LE, 6-axis accelerometer/gyro
Length	68.6 mm
Width	53.4 mm



Objectifs du module

- Comprendre les mécanismes logiciels et matériels en jeu dans les réseaux de capteurs
- Formaliser la notion de temps réel
- Etudier les différents types de capteurs et d'actionneurs
- Comprendre les principes de traitement embarqués et de communications sans-fils associés
- Prototyper les premiers projets

En pratique

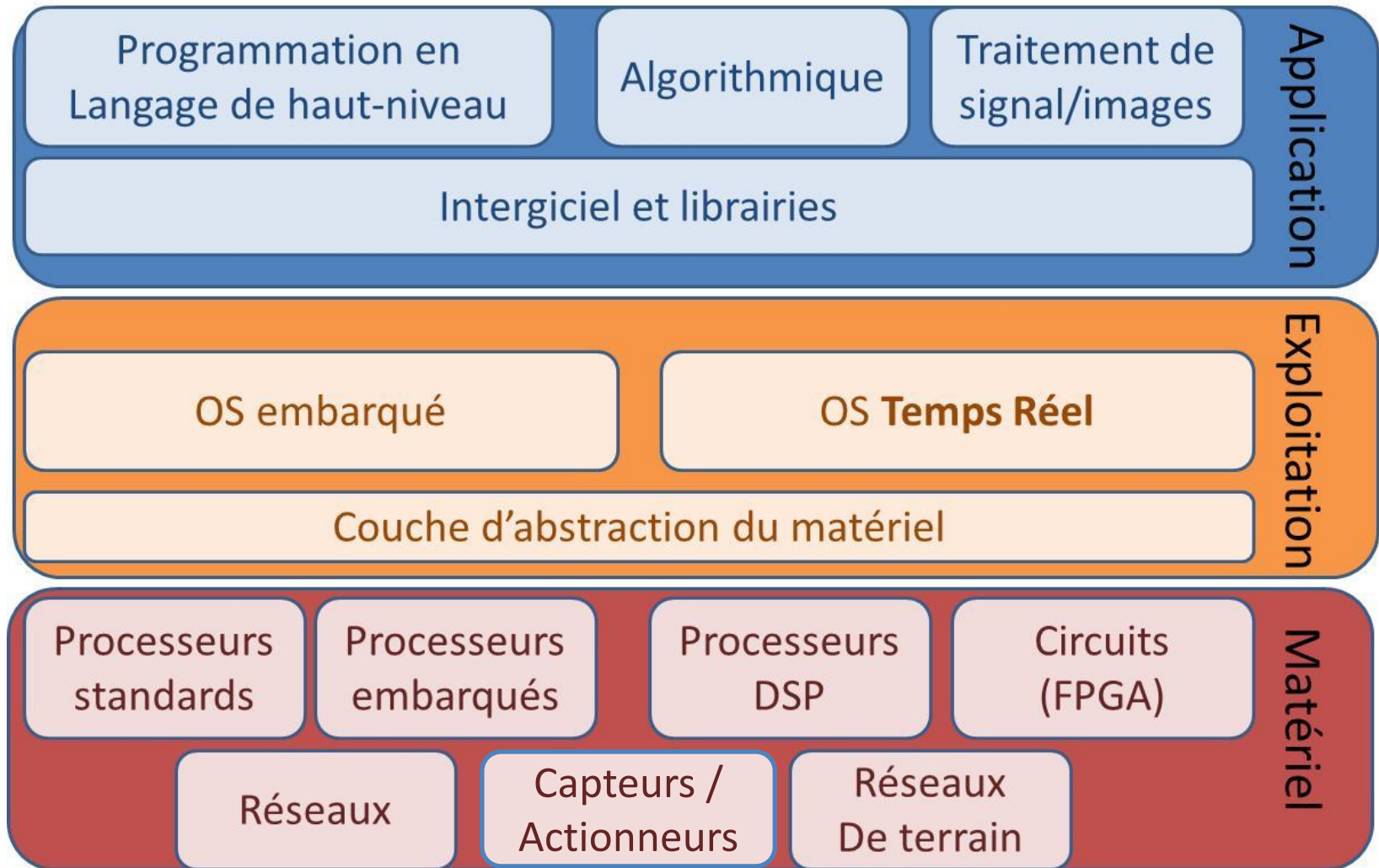
- Programmer les périphériques de micro-contrôleur SoC
- S'initier à la programmation sur OS temps réel
- Réaliser des systèmes capteurs / actionneurs à base d'Arduino
- Programmer des capteurs sans fils

Quelques liens sur l'embarqué

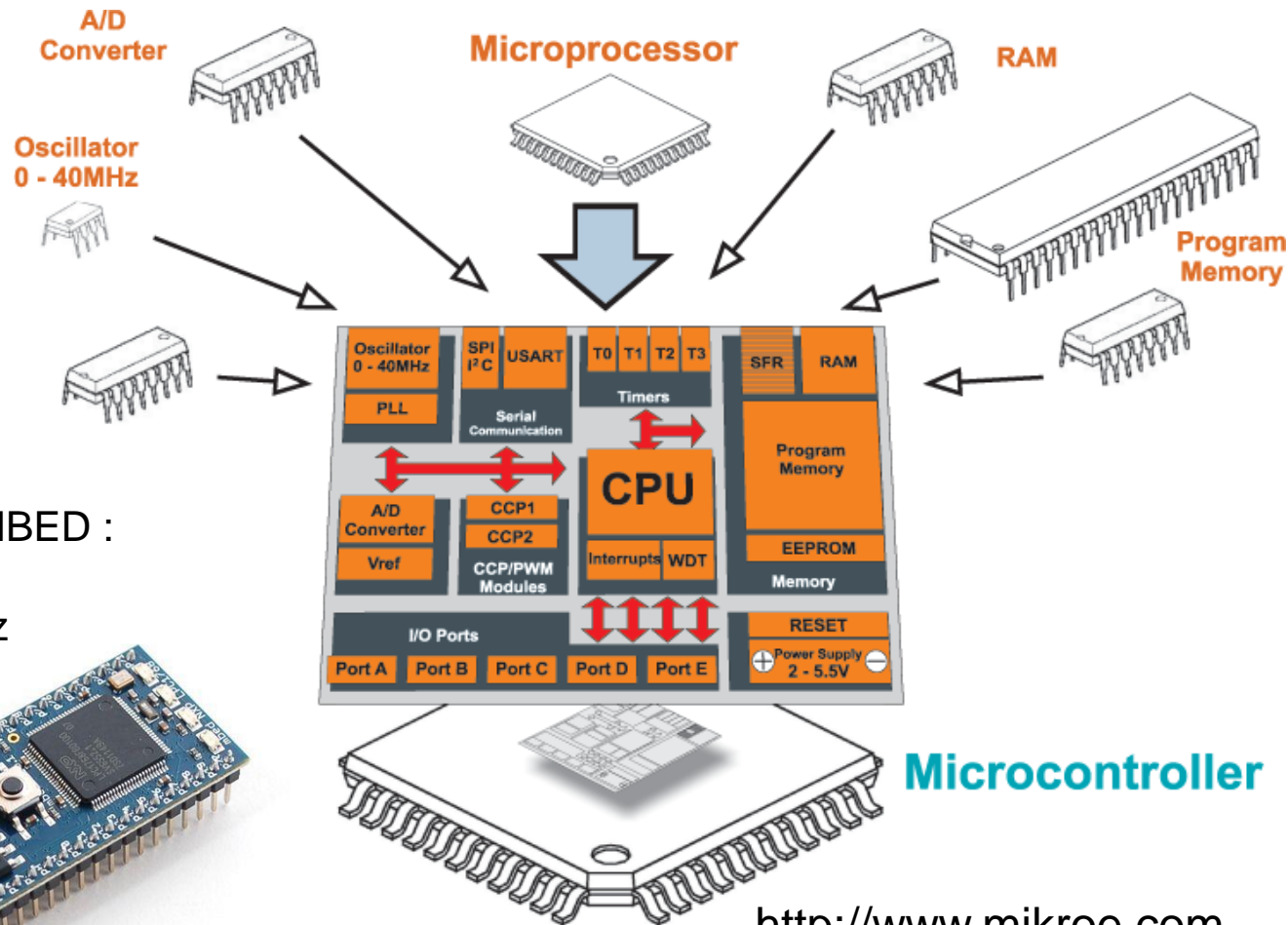
- Embedded linux
 - <http://elinux.org/>
- ARM micro-contrôleurs
 - <https://www.arm.com/markets/internet-of-things-iot.php>
- Magazine en ligne l'embarqué
 - <http://www.lembarque.com/>
- Boutique en ligne et projets DIY libres
 - Adafruit : <https://www.adafruit.com/>

PREMIÈRE PARTIE - LES PÉRIPHÉRIQUES DU SYSTÈME EMBARQUÉ

Organisation du système embarqué



a) Le micro-contrôleur



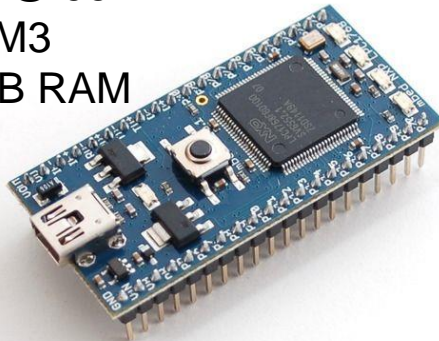
Exemple avec la plateforme MBED :

<https://www.mbed.com/>

NXP NXP LPC1768 @ 96MHz

ARM 32-bits CortexM3

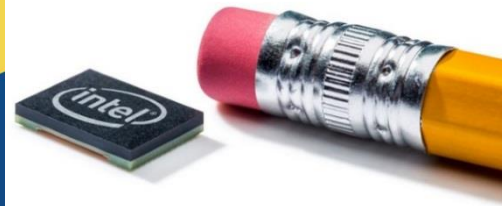
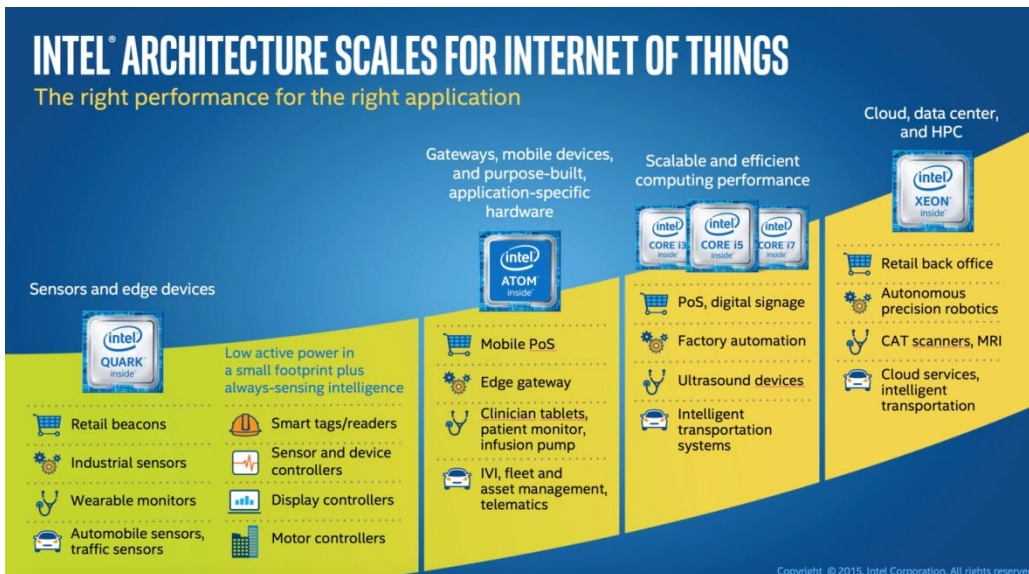
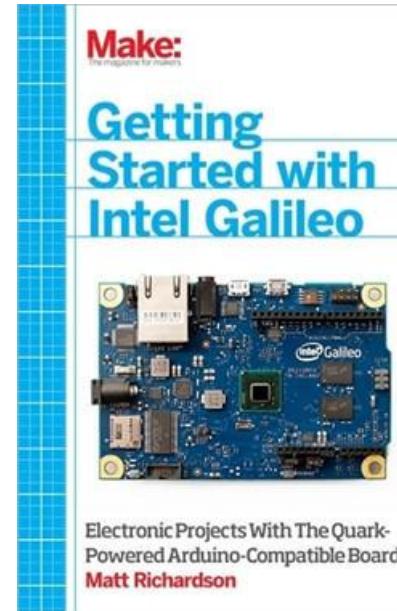
512KB FLASH, 32KB RAM



<http://www.mikroe.com>

Intel Edison / Curie SoC

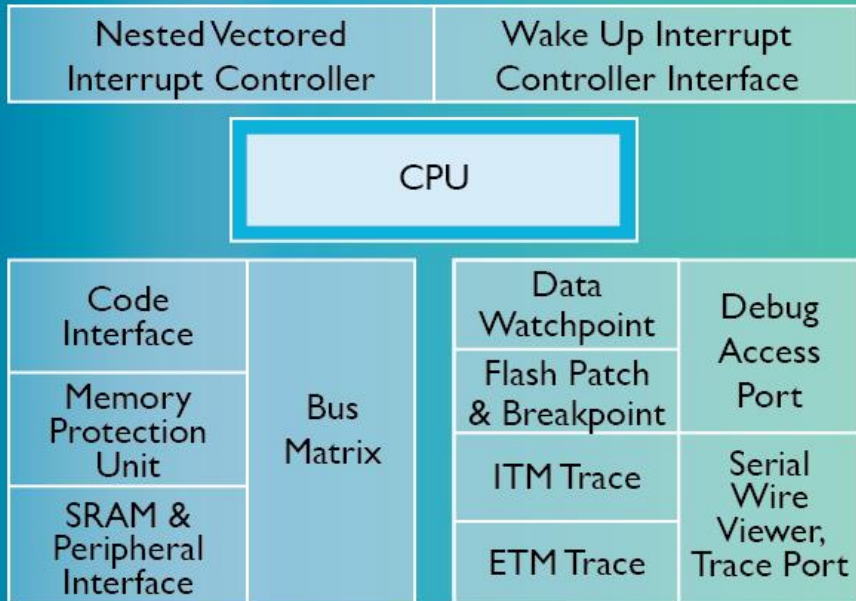
- 22 nm SoC design
- Dual Core Atom @500MHz
- Quark MCU @ 100MHz
- BLE
- Curie : Arduino 101



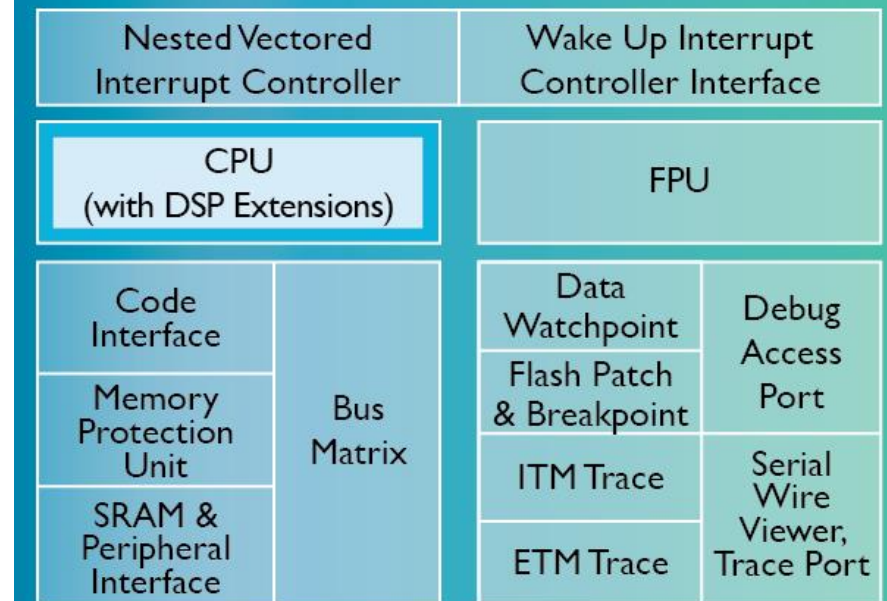
Famille Cortex M3 (cf. Cours SI3, PEP)

ISA Support	Thumb® / Thumb-2
Pipeline	3-stage
Performance Efficiency	1.25 / 1.50 / 1.89 DMIPS/MHz**
Memory Protection	Optional 8 region MPU
Interrupts	1 to 240 physical interrupts
Interrupt Priority Levels	8 to 256 priority levels

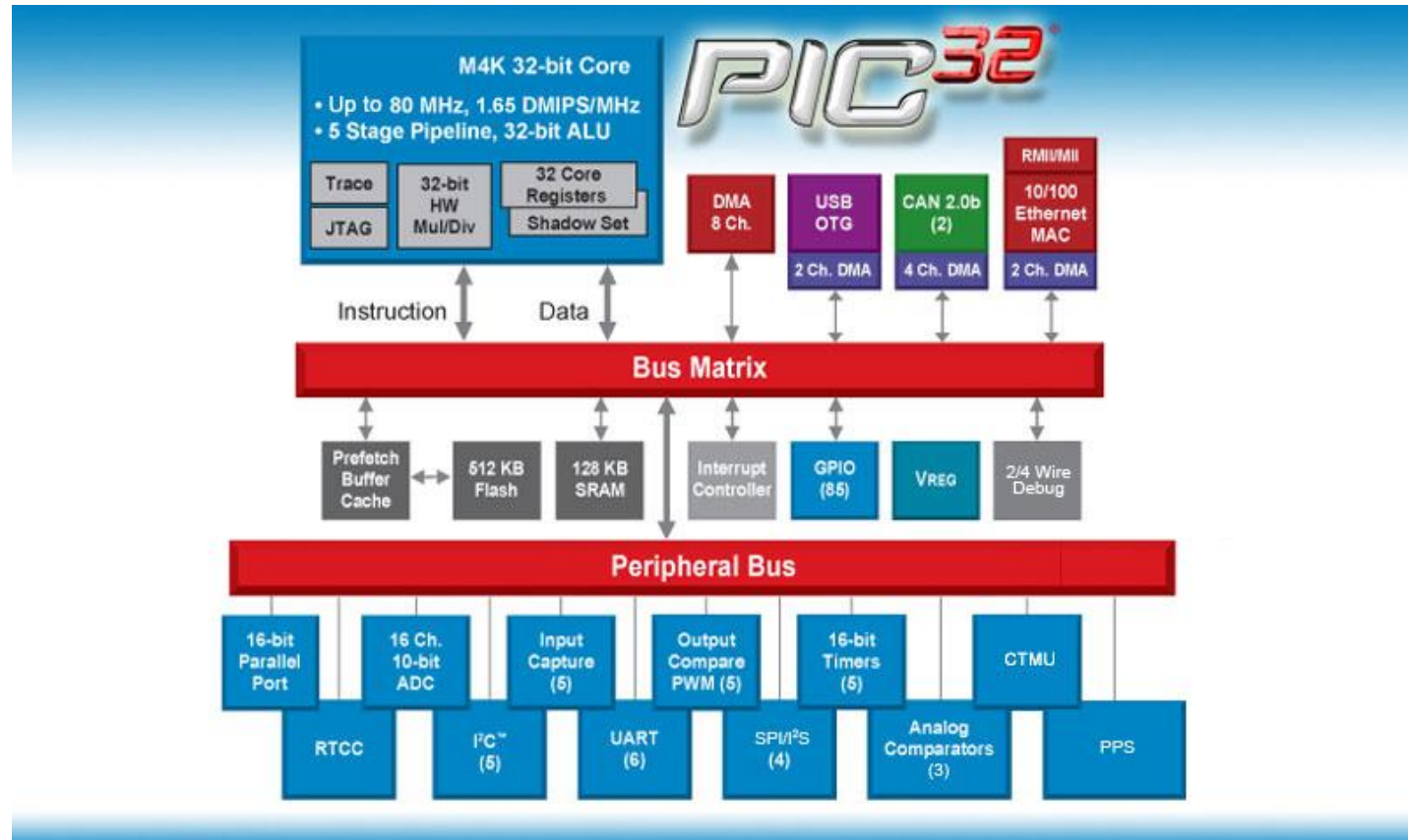
ARM® Cortex®-M3



ARM® Cortex®-M4



Micro-contrôleur PIC



Programmation d'un système embarqué

- Compilation croisée
- Cible à ressources limitées (mémoire, processeur, MMU...)
- Installation sur PC d'un compilateur dédié à l'architecture cible
- Code ELF non interprétable par le PC => téléchargement sur la cible
 - Transfert direct par liaison JTAG (debugger)
 - Transfert indirect : connexion réseau, SD card...
- Sortie standard redirigée sur liaison série (JTAG, UART...)

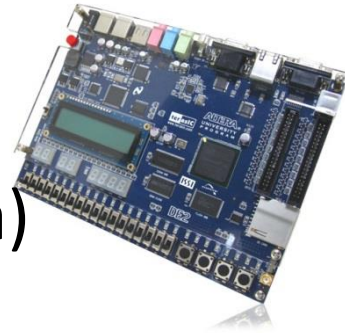
Les périphériques du micro-contrôleur

- Le système embarqué est avant tout
 - Un micro-contrôleur
 - Avec sa mémoire (flash + RAM)
 - Pour l'exécution du code et donc la programmation
- Mais pour interagir avec le monde extérieur, il nécessite des périphériques
 - Capteurs, actionneurs sont des périphériques
- Le MCU communique avec eux par différentes liaisons
 - On board
 - Des bus de communications
 - Des interruptions
 - Off board
 - Réseaux de terrains filaires (S2I, SPI, I2C, CAN...)
 - Réseaux de terrains sans-fils (Bluetooth, Zigbee, Wifi, LoRA, SigFox...)

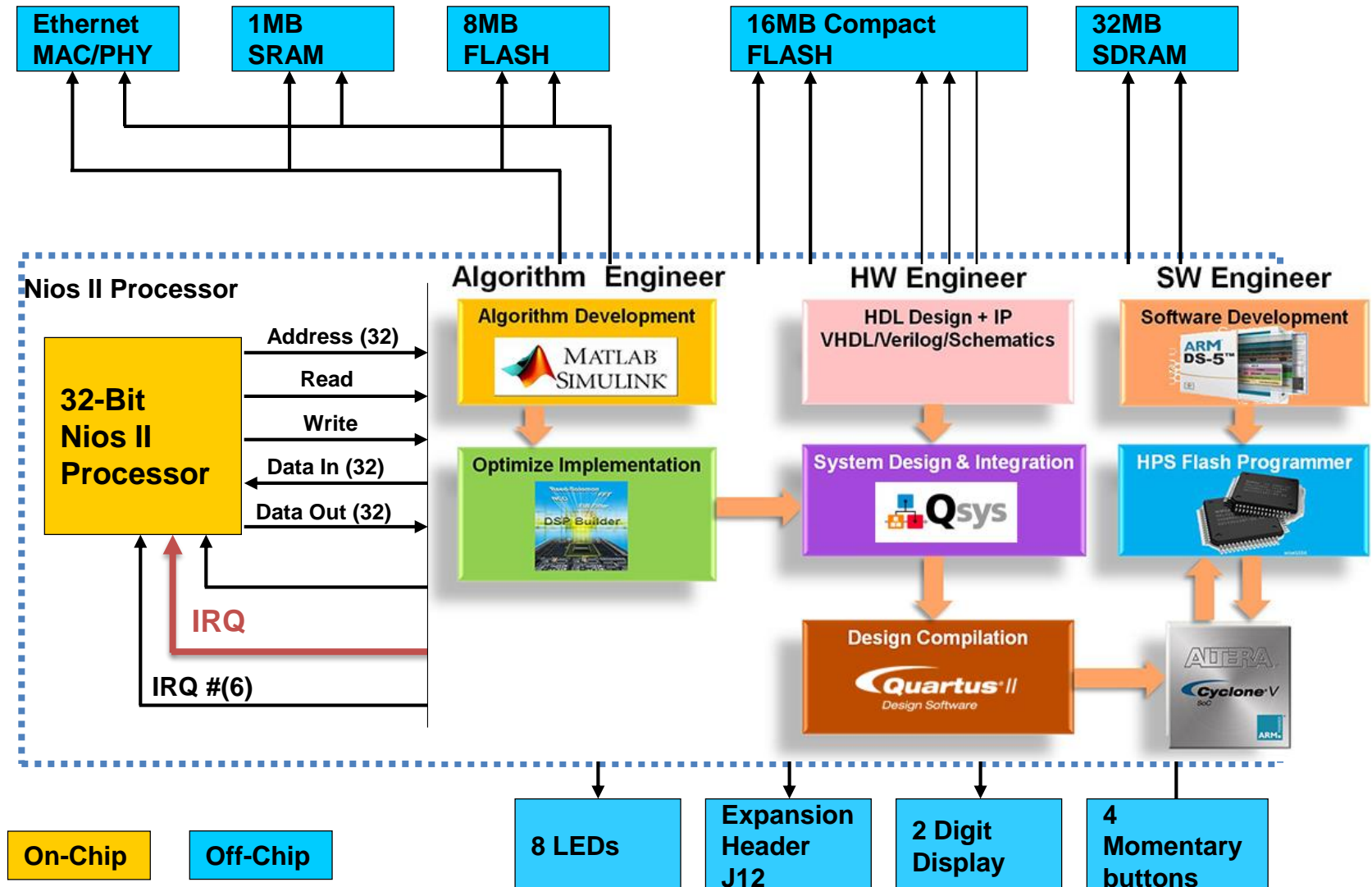


Premier cas : périphériques on-board

- Carte : Terasic DE1 SoC
- Processeur : Soft-Core Nios2 (sur FPGA Altera)
- Cross-compilation
- Outil : Quartus 16.1 Lite edition (IDE Eclipse)
- Exécution Bare-metal ou
- Exécution avec RTOS μ C/OS-II
- Périphériques : LEDs, 7-segments, boutons, switches, LCD ...

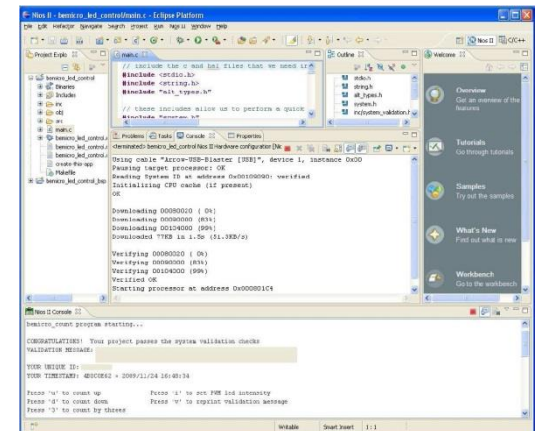


SoC micro-contrôleur sur FPGA



Etapes de conception

1. Configuration du FPGA
2. Analyse des adresses des périphériques mappés mémoires
3. Préparation du code en langage C
 - Accès aux périphériques par adresses mémoire
4. Cross-compilation sous IDE
5. Téléchargement par JTAG-USB
6. Debug par UART



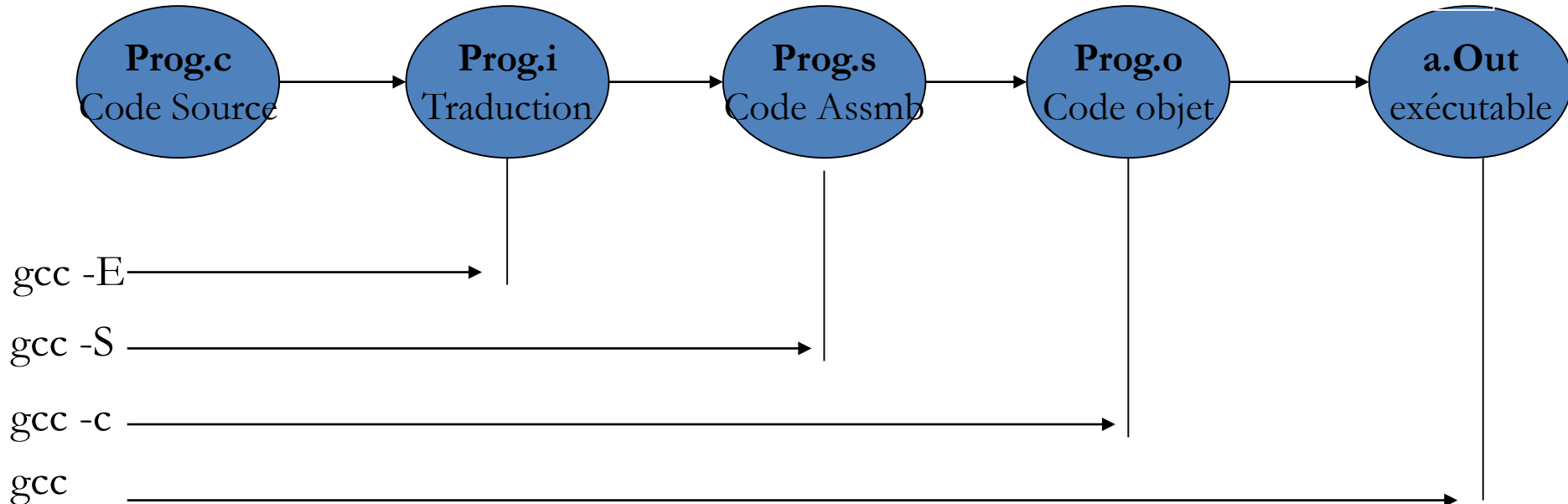
Etapes du compilateur GNU

Preprocesseur

Compilation

Assembleur

Edition de lien



Format des modules objets (.o)

- De manière à pouvoir utiliser des compilateurs, assembleurs et éditeurs de liens provenant de vendeurs différents (interopérabilité), 2 formats de fichiers objets (sections) ont été standardisés
 - COFF (Common Object File Format)
 - ELF (Executable and Linker Format)

Structure d'un module objet ELF

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

- En-tête
 - Nom de fichier, Taille, adresse de début
- Espace objet (divisé en sections)
 - Code binaire
 - Zone de données
- Table des symboles
 - Symboles utilisables et à satisfaire
- Informations complémentaires
 - Auteurs, outils utilisé, versions, environnement...

Types de contenu

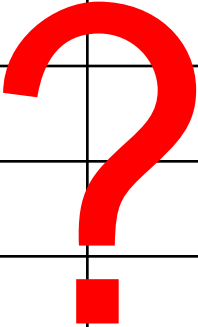
- Le compilateur organise le programme par types de contenus appelés *sections* :
- `.text/.code` = Instructions binaires
- `.data` = Données binaires initialisées
- `.bss` = (Block Started by Symbol) Données globales non initialisées
- `.rodata` = Read Only Data (Chaîne de caractères)
- `.comment` = commentaires
- `.symtab` = table des symboles
- `.rel` = table de résolution
- ...
- Le Standard ELF permet de définir autant de sections qu'on le souhaite avec n'importe quel nom
- Outils pour lire les sections : **objdump** (tous fichiers binaires) et **readelf** (ELF seulement)

Types de contenu affichés par la commande **'nm'**

- B – dans la zone .bss
- D – dans la zone .data
- C – non initialisé
- T – dans la zone .text
- U – Undefined symbol

Rangement des variables

			.data	.bss	.rodata	Pile
Globale	static	initialisée				
		non init.				
	dyna	init.				
		non init				
Locale	static	init				
		non init				
	dyna	init				
		non init				
G/L	const					

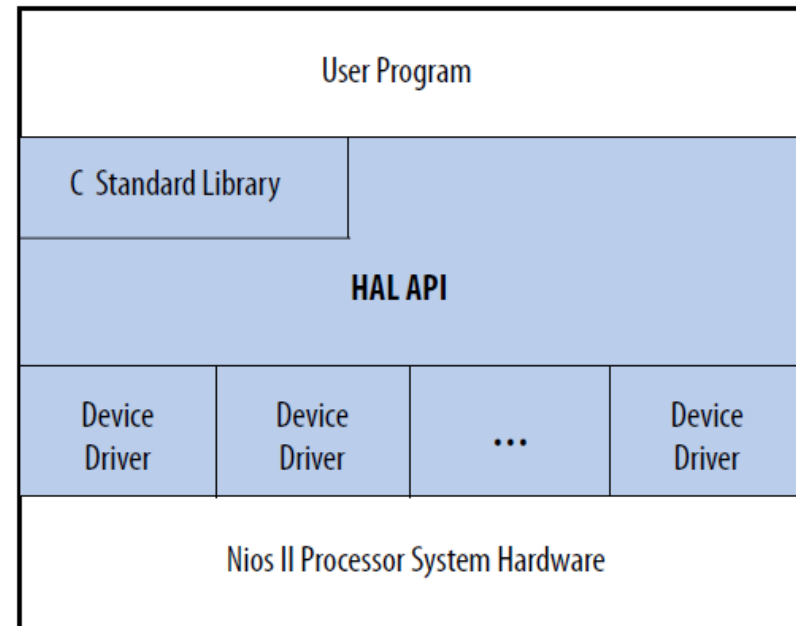


Rangement des variables

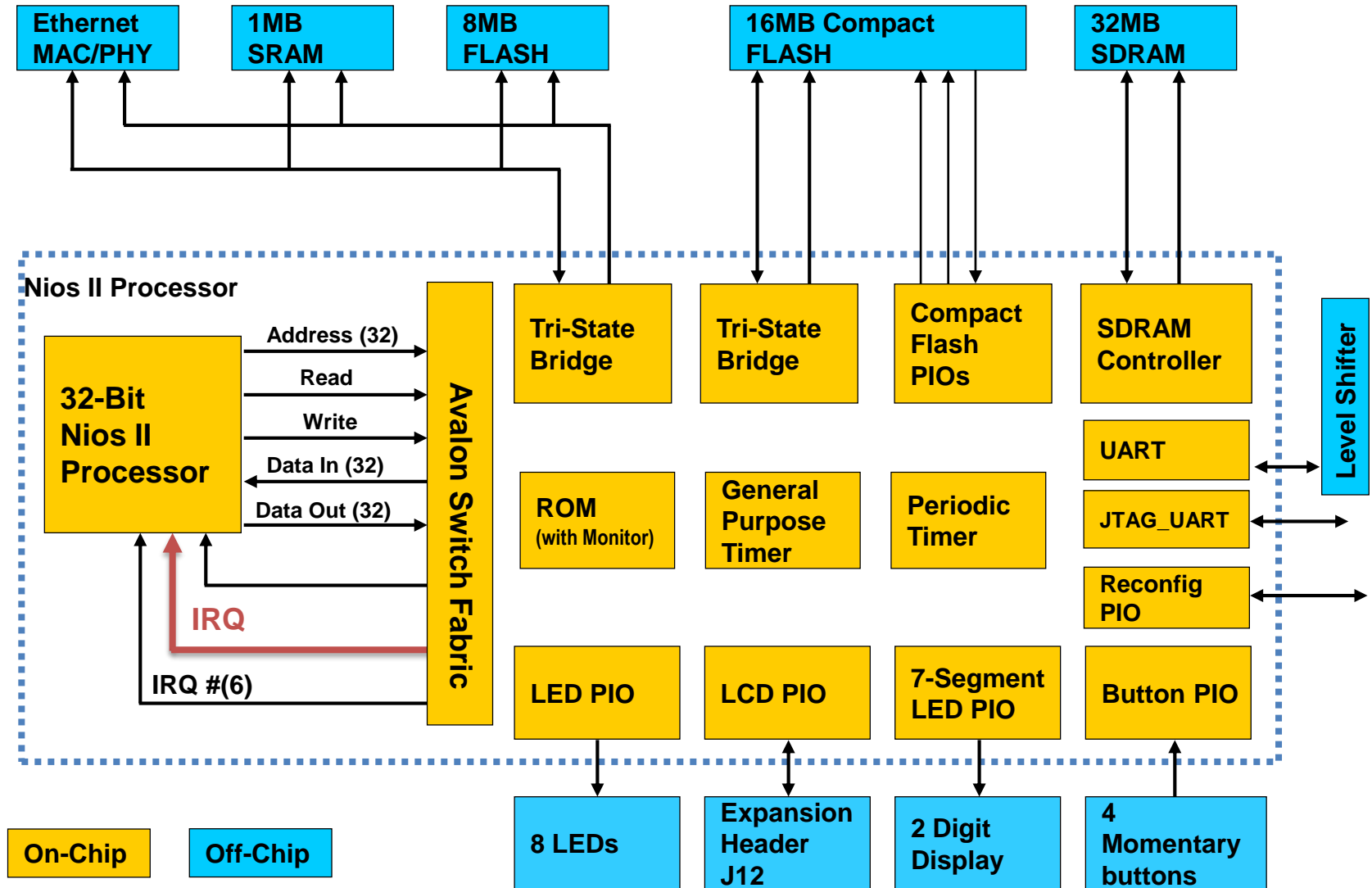
			.data	.bss	.rodata	Pile
Globale	static	initialisée				
		non init.				
	dyna	init.				
		non init				
Locale	static	init				
		non init				
	dyna	init				
		non init				
G/L	const					

HAL pour l'accès aux périphériques

- Le programme applicatif (en C) s'appuie sur 2 couches logicielles :
- HAL (Hardware Abstraction Layer) contient les déclarations de fonctions d'accès aux ressources matérielles (drivers, configuration du processeur, initialisation Hw...)
- Exemple de lecture / écriture en périphériques :
IOWR_ALT_UP_PARALLEL_PORT_DATA (address, data)
data = IORD_ALT_UP_PARALLEL_PORT_DATA (address)
- Le BSP (Board Support Package) contient les implémentations spécifiques aux périphériques utilisés sur la carte

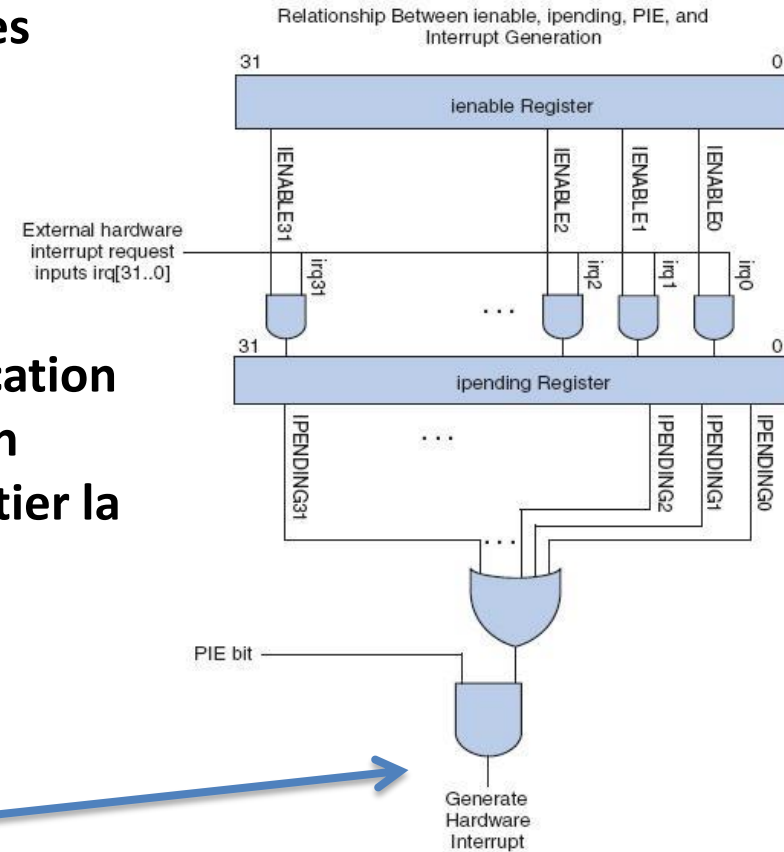


Architecture d'un SoC microcontrôleur



Les interruptions

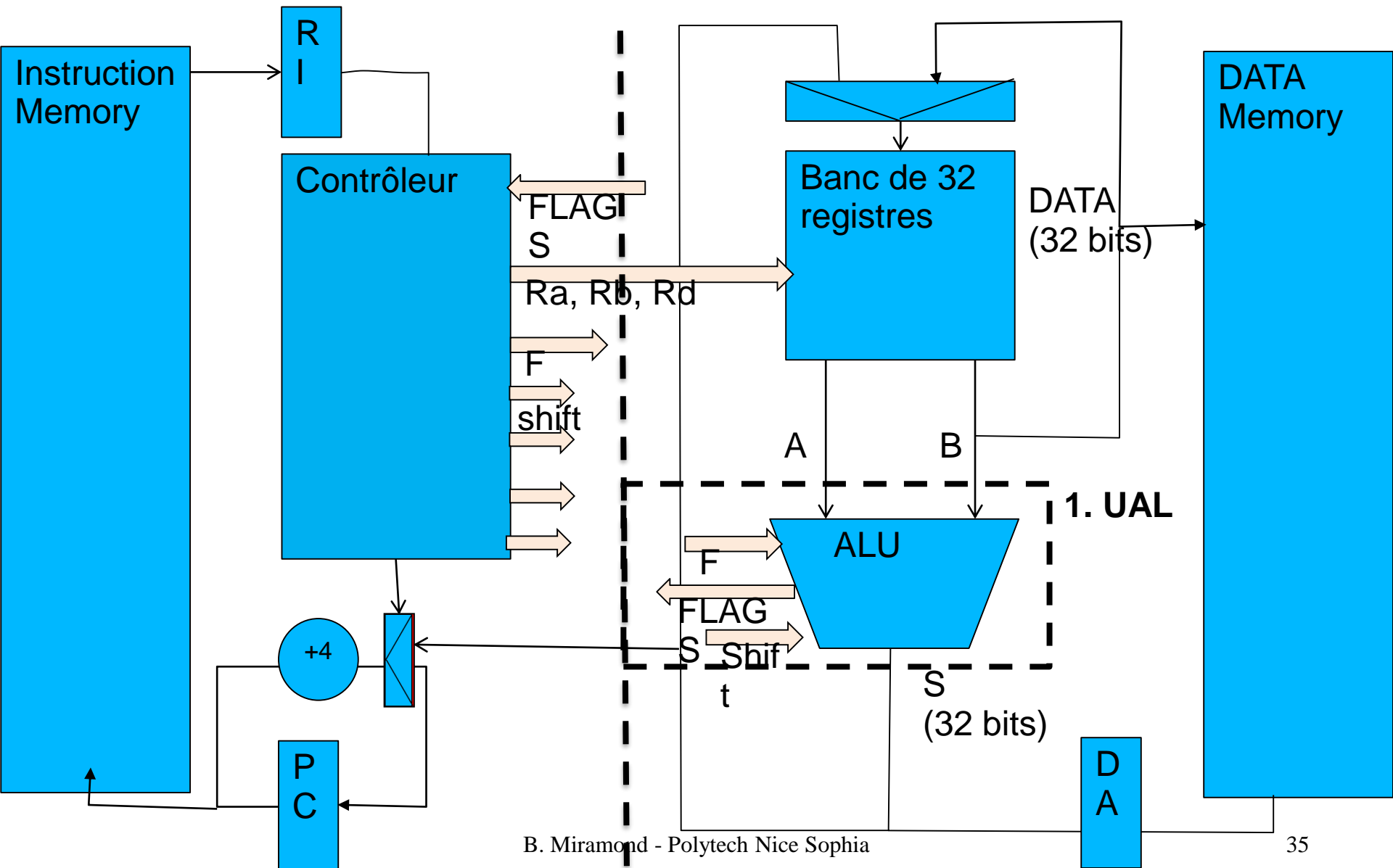
- Dans un SoC, seul le processeur peut initier des communications :
 - Avec la mémoire, les périphériques,
 - Il est désigné comme Master
 - Les périphériques comme Slave.
- Une interruptions est un moyen de communication asynchrone pour indiquer au processeur qu'un évènement s'est produit et qu'il peut donc initier la communication appropriée
- Les processeurs RISC disposent de 32 lignes d'interruptions qui
 - Sont priorisées (de 1 à 32),
 - Peuvent être masquées,
 - Doivent être programmées
- L'autre méthode d'accès à un périphérique asynchrone est le Polling ou méthode par scrutation où le processeur lit à période régulière
 - Gaspillage de temps processeur + moins réactif



Architecture générale du MCU

2. Contrôleur

3. Chemin de données

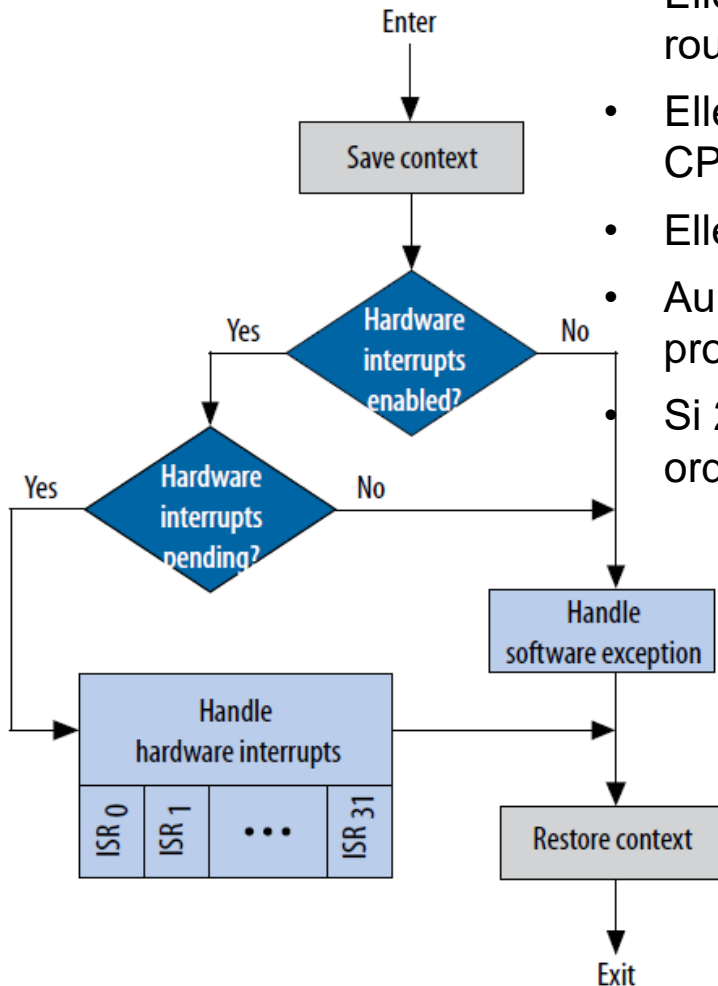


Fonctionnement des interruptions

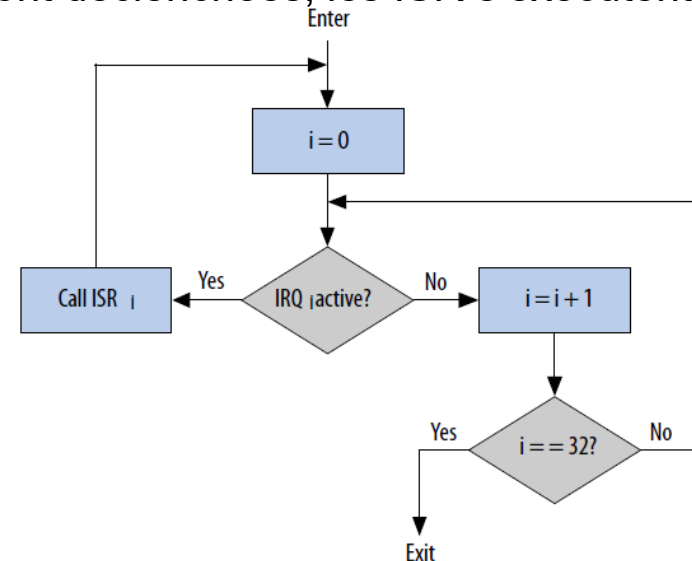
(cablées dans le CPU)

L'interruption provoque une rupture de séquence du programme:

- Elle provoque un changement de contexte pour exécuter la routine d'interruption
 - Elle vide pour cela le pipeline et sauvegarde les registres du CPU (save context)
 - Elle appelle l'ISR (saut) et exécute son code
 - Au retour d'ISR, le contexte est restauré, l'exécution du programme reprend
- Si 2 interruptions sont déclenchées, les ISR s'exécutent par ordre de priorité



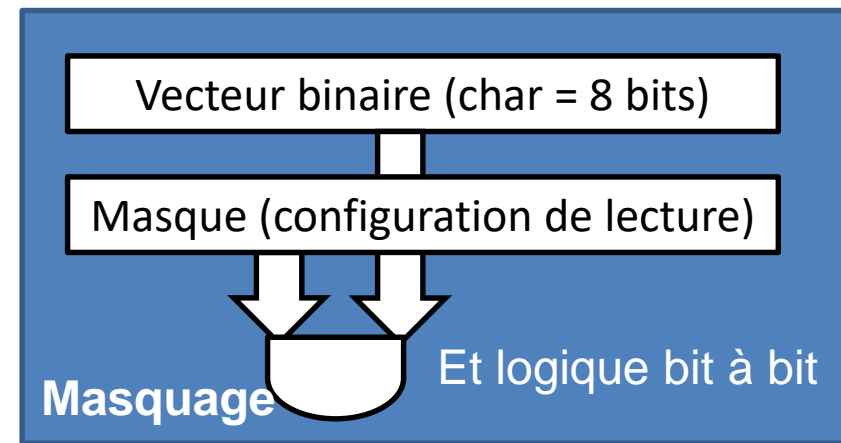
Masquage des interruptions



Priorités des interruptions

Programmation des interruptions

- La programmation des IRQ se fait en 4 étapes :
 1. Masquage des interruptions (autorise ou non toutes les IRQ),
 2. Initialise l'état du périphérique associé,
 3. Enregistrement d'une routine d'interruption qui associe un numéro de l'IRQ à une adresse de saut,
 4. Coder la routine d'interruption ou ISR (Interrupt Service Routine)
- Le code de la routine est une section critique (non interruptible). Il est donc soumis à plusieurs règles :
 - Code relativement court, limitant la latence d'interruption,
 - Pas d'appel bloquant,
 - Ré-initialiser la source d'interruption




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

Variable définie comme
volatile
Fonction définie comme
static

L'adresse du
périphérique est définie
comme une macro dans
le fichier system.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. */
```

1

```
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);
```

2

```
/* Register the ISR. */
```

```
alt_irq_register( NUMERO_IRQ,edge_capture_ptr,handle_button_interrupts );
```

```
}
```

3

Mots clés volatile et static

- Volatile
 - Utilisé dans le cas de variables dont la valeur peut changer spontanément :
 - sans action du processeur, périphériques mappés mémoire
 - Par une autre tâche, logiciel multithread
 - Ce préfixe indique au compilateur d'éviter les optimisations qui génère une instruction de lecture mémoire systématique
- Static
 - Sur une variable, conserve la valeur de la variable locale (allocation hors pile)
 - Sur une fonction limite la définition du symbole à l'intérieur de l'objet (du fichier)

```
#include "system.h"
```

```
#include "altera_avalon_pio_regs.h"
```

```
#include "alt_types.h"
```

4

```
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);
```

```
}
```

L'adresse du
périphérique est définie
comme une macro dans
le fichier system.h

Cadre du premier TP

- Objectif :
 - Cross compilation sur cible embarquée SoC
 - Programmation des périphériques
 - Programmation des interruptions
- A préparer :
 - Installer Quartus Prime 16.1 Lite edition depuis la clé USB ou sur le site de Altera :
<https://www.altera.com/downloads/download-center.html>
 - Installer altera University program 16.1 :
<https://www.altera.com/support/training/university/materials-software.html#University-Program-Installer>

