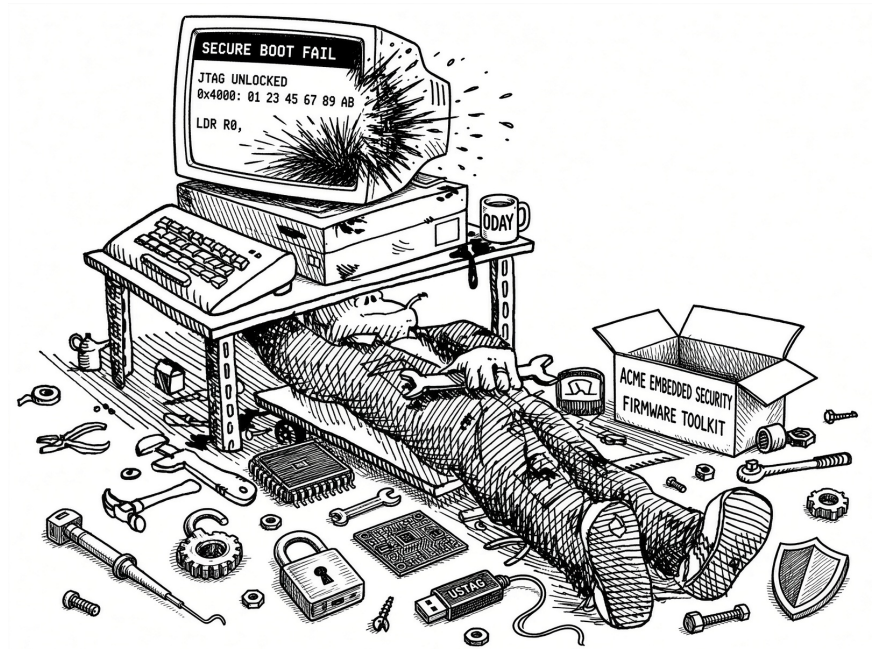




But du module d'enseignement

Retro-conception de firmware :

- ▷ utilisation de sonde hardware : JTAG/SWD et openOCD ;
- ▷ désassemblage de code : assembleur ARM et gdb ;
- ▷ exécution pas à pas et modification du contexte d'exécution.

Interception de transmission :

- ▷ entre composant d'un circuit : SPI, I2C, série ;
- ▷ par transmission radio : nrf24L01, LoRa ;
- ▷ analyse de firmware : extraction de clé et secrets ;

Éléments de sécurité pour l'embarqué :

- ▷ protection des accès mémoires : MPU ;
- ▷ chiffrement et authentification : AES, certificats ;
- ▷ protocole sécurisé : DTLS ;
- ▷ co-processeur pour la sécurité : ATECC508 ;

Conception de circuit pour la sécurité :

- ▷ LFSR, «Linear Feedback Shift Register» ;
- ▷ TRNG, «True Random Number Generator» ;
- ▷ intégration dans un SoC basé RISC-V ;
- ▷ Utilisation au travers de la programmation en C.

Moyens pratiques

Outils pour la rétro-conception :

- ▷ programmation assembleur ARM : instructions thumb, accès périphériques, interruptions, appels systèmes, piles et changement de contexte ;
- ▷ utilisation d'openOCD/gdb : SWD ;
- ▷ simulateur de SoC : renode ;
- ▷ analyseur logique : communication SPI/I2C, série ;

Étude et extension d'un micro-os :

- ▷ utilisation de Piccolo OS pour Raspberry Pico ;
- ▷ intégration d'un service dans l'OS utilisable par appel système ;
- ▷ protection de la mémoire à l'aide de la MPU ;
- ▷ sécurité des données sensibles
- ▷ bootloader et code sécurisé exécuté en RAM.

Utilisation du RTOS Zephyr :

- ▷ openOCD/gdb et «OS awareness» ;
- ▷ utilisation du protocole de transmission radio nrf52833 ;
- ▷ utilisation et interception du protocole DTLS ;

Développement de circuit logique pour la sécurité :

- ▷ découvrir la **programmation de circuit logique** avec Verilog sur FPGA ;
- ▷ Utilisation du PicoSoc pour intégrer le circuit de sécurité et le programmer en C :

Moyens techniques

Logiciels :

- ☐ cross-compileur ARM et RISC-V : gcc en mode bare-metal ;
- ☐ débogueur software/hardware : openOCD et gdb ;
- ☐ synthèse de circuit logique pour FPGA basé verilog : yosys ;
- ☐ simulateur de SoC : renode ;
- ☐ analyse de trace de communication : sigrok et pulseview ;
- ☐ développement embarqué rapide : circuitpython ;

Matériels :

- ☐ Raspberry Pico ;
- ☐ Micro:bitv2 ;
- ☐ analyseur logique ;
- ☐ FPGA Black ICE II avec Lattice iCE 40 hx8k ;
- ☐ composant sécurisé ATECC508 en I2C.

Différences Embarqué et ordinateur classique

Du point de vue hardware

6

Critère	Ordinateur classique	Système embarqué
Architecture processeur	CPU génériques multicœurs (x86, ARM), fréquence élevée, pipelines profonds, prédiction de branchement, exécution out-of-order	Cœurs plus simples : MCU (Cortex-M), microcontrôleurs 8/16 bits, DSP, cœurs applicables (ex. cœur dédié à une tâche) ; souvent un seul cœur
Mémoire	Grande capacité (DRAM + SSD), hiérarchie de caches volumineuse, mémoire virtuelle (MMU), pagination	Mémoire réduite et figée : SRAM/ROM/Flash embarquée, mémoire intégrée à la puce (SoC), souvent pas de MMU (MPU ou pas du tout) ; adressage physique direct
Peripheral / I/O	Bus standardisés (PCIe, USB, SATA), périphériques amovibles et génériques	Interfaces applicatives : GPIO, ADC/DAC, PWM, CAN, LIN, I ² C/SPI vers des capteurs et actionneurs. Le nombre de broches est un argument économique fort
Ressources	dimensionnées « en excès » pour la généralité (power-hungry)	Dimensionnées au plus juste : coût par unité critique (un million d'unités vendues → 1 € de moins par unité = gros impact), contraintes de consommation, d'encombrement, de température
Fiabilité / durée de vie	Renouvellement rapide (3–5 ans), pannes réparables, redémarrages fréquents	Longévité longue (automobile : 15–20 ans), environnements hostiles (température, vibrations, CEM), pas de disque mécanique souvent, contraintes temps réel
Réactivité	Haut débit privilégié (throughput)	Faible latence et déterminisme temporel privilégiés (temps réel dur ou mou)
Énergie	Secteur ou batterie de grande capacité	Souvent contraint par l'énergie disponible (batterie limitée, energy harvesting), modes basse consommation

Critère	Ordinateur classique	Système embarqué
Système d'exploitation	OS généraliste riche (Linux/Windows/macOS) avec processus, mémoire virtuelle, pilotes dynamiques, shell interactif	Selon les cas : pas d'OS (boucle principale + interruptions), super-boucle, RTOS (FreeRTOS, RTEMS, Zephyr, OSEK/VDX en automobile), Linux embarqué pour les systèmes complexes
Temps réel	Quasi-absent : latences « moyennes » acceptées, ordonnancement équitable	Central : tâches périodiques ou aperiodiques avec échéances (deadline). RTOS à ordonnancement préemptif basé sur la priorité. Distinction hard / soft / firm real-time
Cycle de développement	Logiciel indépendant du matériel, mis à jour fréquemment (app store, hotfix)	Développement en Y-in-V, co-design hardware/software : le logiciel est souvent développé avant que le silicium soit disponible (simulateurs, FPGA, boards d'évaluation). Le logiciel doit parfois être figé (certification)
Déploiement / mise à jour	Installation dynamique, téléchargement de paquets	Firmware compilé et flashé en usine ; mises à jour difficiles (pas d'interface utilisateur, pas de clavier), souvent par bootloader spécifique ou OTA quand c'est possible
Interaction humain-machine	Riche : écran, clavier, souris, terminal	Souvent minimale ou absente : boutons, LED, diagnostics via port série/CAN. Quand IHM il y a, elle est dédiée (écran d'un GPS automobile)

Critère	Ordinateur classique	Système embarqué
Qualité et certification	Tests fonctionnels, pas d'exigence réglementaire générale pour le grand public	Exigences normatives sectorielles fortes : ISO 26262 (automobile), DO-178C / EN 50129 (ferroviaire/aéronautique), IEC 61508 (industriel), medical devices. Traçabilité des exigences, revue de code, couverture de tests
Robustesse	Panne redémarrage acceptable	Panne potentiellement catastrophe (freins, pacemaker). Watchdog hardware obligatoire, redondances parfois
Part du logiciel	Le logiciel est le produit	Dans beaucoup de systèmes modernes, le logiciel embarqué est la part la plus coûteuse et la plus complexe (ex. voiture : des millions de lignes de code)

Rappels sur GDB

Cette extension de `gdb` est écrite en Python et ajoute les fonctionnalités suivantes :

- la commande `dashboard` affiche un écran composé de:

Assembly																	
0x000006e0	?	isb	sy														
0x000006e4	?	cpsid	i														
0x000006e6	?	bx	lr														
0x000006e8	?	ldr	r3,	[pc,	#52]	; (0x720)											
0x000006ea	?	ldr	r2,	[pc,	#56]	; (0x724)											
0x000006ec	?	ldr	r1,	[pc,	#56]	; (0x728)											
0x000006ee	?	mov	r0, sp														
0x000006f0	?	cmp	r0, r2														
0x000006f2	?	bhi.n	0x6fc														
0x000006f4	?	cmp	r0, r3														
Breakpoints																	
Expressions																	
History																	
Memory																	
\$sp																	
0x200001b0	00	00	00	00	50	08	00	20	06	00	00	00	8d	06	00	00	P.....
0x200001c0	00	00	00	00	fd	ff	ff	01	00	00	00	08	08	00	20		
0x200001d0	00	00	00	10	00	ed	00	e0	08	00	00	00	d3	01	00	00	
0x200001e0	7c	06	00	00	00	00	00	41	95	01	00	00	00	00	00	00	A.....
Registers																	
r0	0x20000828	r4	0x20000808	r8	0x00000008	r12	0x20000770	xPSR	0x410e000e	primask		0x00					
r1	0x20000850	r5	0x00000000	r9	0x00000009	sp	0x200001b0	fpscr	0x00000000	basepri		0x00					
r2	0x10000000	r6	0x20000850	r10	0x0000000a	lr	0x000002b1	misp	0x200001b0	faultmask		0x00					
r3	0x00000000	r7	0x00000007	r11	0x0000000b	pc	0x000006e0	psp	0x200007a8	control		0x00					
Source																	
Stack																	
[0] from 0x000006e0																	
Threads																	
[1] id 0 from 0x000006e0																	
Variables																	

<code>ctrl-c</code>	<i>interrompt l'exécution du program</i>
<code>c/continue</code>	<i>reprend l'exécution</i>
<code>s/step</code>	<i>avance d'une insrustion dans une fonction</i>
<code>s 10</code>	<i>avance de 10 instructions</i>
<code>n/next</code>	<i>avance à la prochaine instruction dans la fonction</i>
<code>u/until 20</code>	<i>avance jusqu'à la ligne 20 du fichier courant</i>
<code>f/finish</code>	<i>avance jusqu'à la fin de la fonction</i>
<code>run</code>	<i>démarre le programme</i>
<code>b/break fonc</code>	<i>mets un «breakpoint» lors de l'exécution de la fonc</i>
<code>b main.c:fonc</code>	<i>mets un «breakpoint» sur fonc du fichier «main.c»</i>
<code>b main.c:18 if var > 20</code>	<i>breakpoint seulement si var > 20</i>
<code>tbreak main</code>	<i>se déclenche une fois et s'efface ensuite</i>
<code>info breakpoints</code>	<i>donne la liste des breakpoints</i>
<code>ignore 3 20</code>	<i>ignorer 20 fois le breakpoint 3</i>
<code>disable 3</code>	<i>désactive le breakpoint 3</i>
<code>delete 3</code>	<i>supprime le breakpoint 3</i>
<code>monitor reset halt</code>	<i>réinitialise le firmware dans OpenOCD et s'arrête après le reset</i>

<code>info locals</code>	les variables locales
<code>info variables</code>	les variables globales
<code>info args</code>	les arguments de la fonction
<code>info registers</code>	les valeurs des registres

<code>watch var</code>	surveille les modifications de var
<code>watch tableau[10].val</code>	surveille le champ val de la 11 ^{ème} structure
<code>watch *0xdeadcafe</code>	surveille le contenu de la mémoire par adresse
<code>watch var if var > 20</code>	surveillance conditionnelle
<code>watch var if var - 10 > 20</code>	avec une expression
<code>info watchpoints</code>	donne la liste des watchpoints
<code>delete 5</code>	supprime la 5 ^{ème} surveillance

<code>bt</code>	«backtrace»: l'historique des appels de fonction
<code>frame</code>	la «frame» courante dans la pile
<code>up</code>	remonter dans la pile d'appel
<code>down</code>	descendre dans la pile d'appel

p/print /FMT expression	a (address)	o (octal)
	c (char)	t (binary int)
	d (decimal int)	u (unsigned decimal int)
	f (float)	x (hex int)
p var	affiche la valeur de var	
p x+y	affiche le résultat de l'expression	
p/x &main	affiche l'adresse de la fonction main	
p/x \$r4	affiche le contenu du registre «r4»	
p/a *(uint32_t[8] *) 0xdeadbabe	affiche un tableau de 8 entier l'adresse donnée	

x /FMT adresse	x (hex)	b (byte)
	d (decimal)	h (halfword 2B)
	u (unsigned decimal)	w (word 4B)
	f (float)	a (address)
	i (instruction)	c (char)
	s (string)	z (padded hex)
x var	affiche l'adresse de var	
x/4c 0xdeadbabe	affiche 4 char à l'adresse indiquée	
x/4xw &main	affiche 4 mots en hexa à l'adresse de main	

Afficher le source et les instructions machine

<code>list</code>	affiche le source à l'endroit courant
<code>list *0x12341234</code>	affiche le source à l'adresse indiquée
<code>list main.c:func</code>	affiche le source de la fonction définie dans le fichier main.c
<code>disas func</code>	desassemble la fonction

Rechercher dans la mémoire

<code>find /b 0x0,0x10000,'H','e','l','l','o'</code>	chercher une séquence entre 0x0 et 0x10000
<code>⇒0x581f 1 pattern found</code>	
<code>x/s 0x581f</code>	examiner la chaine à l'adresse 0x581f
<code>⇒"Hello world !"</code>	

Charger la table des Symboles

<code>symbol-file mon_exec.elf</code>	charge un nouveau fichier
---------------------------------------	---------------------------

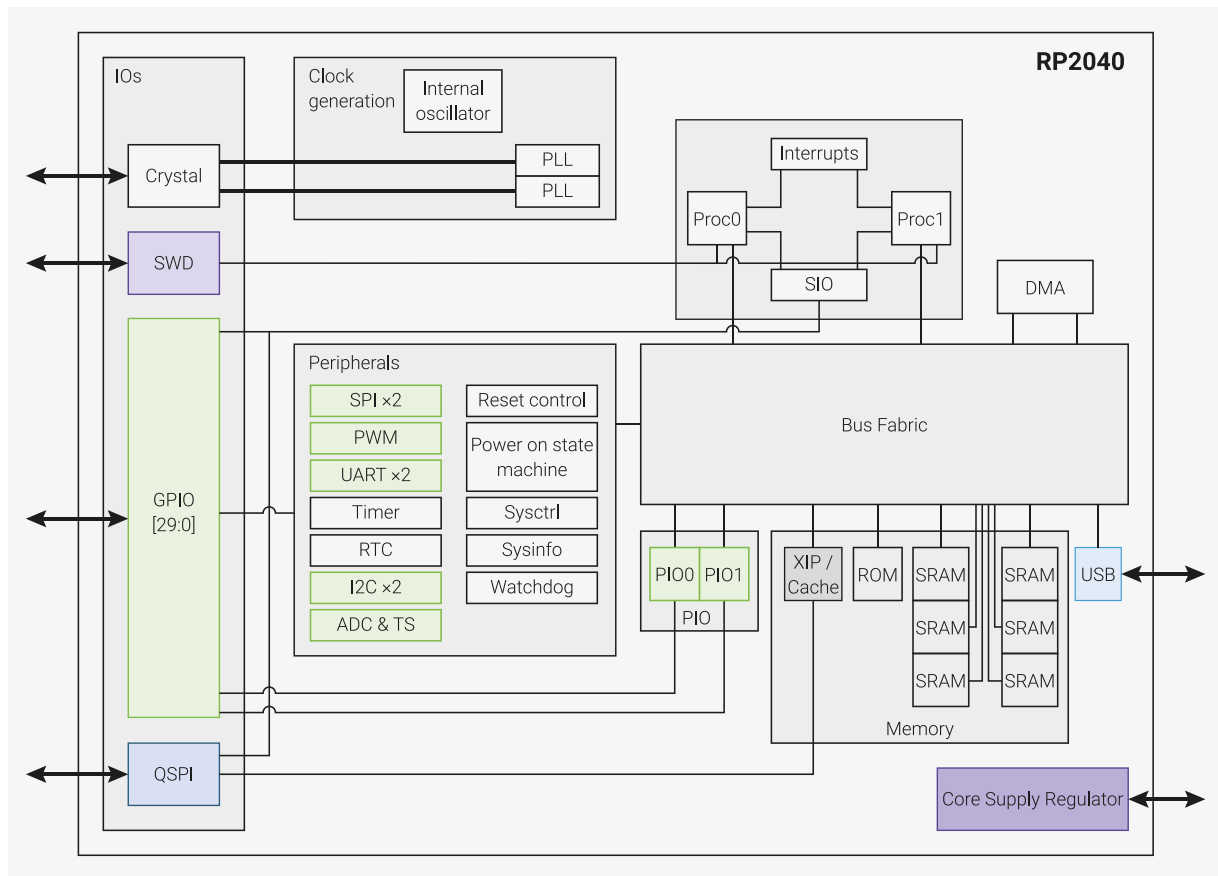
La plateforme Raspberry Pico



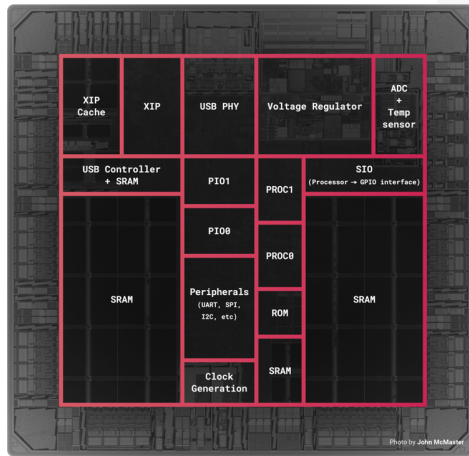
RP2040 is a low-cost, **high-performance microcontroller** device with **flexible** digital interfaces.

- ☐ **Dual Cortex M0+** processor cores, up to 133MHz ;
- ☐ **264kB** of embedded SRAM in 6 banks ;
- ☐ 30 multifunction **GPIO** ;
- ☐ 6 dedicated IO for **SPI Flash** (supporting XIP, «*eXecute-In-Place*» : considérer la mémoire Flash externe comme de la mémoire interne) ;
- ☐ Dedicated hardware for **commonly used peripherals** ;
- ☐ Programmable IO for **extended peripheral support** ;
- ☐ **4 channel ADC** with internal temperature sensor, 500ksps, 12-bit conversion ;
- ☐ USB 1.1 **Host/Device**.

- **Code** may be executed **directly from external memory** through a dedicated SPI, DSPI or QSPI interface.
A small cache improves performance for typical applications.
- **Debug** is available via the **SWD interface**.
- **Internal SRAM** can contain code or data. It is addressed as a single 264 kB region, but physically partitioned into 6 banks to allow **simultaneous parallel access** from different masters.
- **DMA** bus masters are available to offload repetitive data transfer tasks from the processors.
GPIO pins can be driven directly, or from a variety of dedicated logic functions.
- **Dedicated hardware** for fixed functions such as SPI, I2C, UART.
- **Flexible configurable PIO** controllers can be used to provide a wide variety of IO functions.
- A **USB controller** with embedded PHY can be used to provide FS, «Full Speed, 12Mbps»/LS, «Low Speed, 1,5Mbps» Host or Device connectivity under software control.
Four ADC inputs which are shared with GPIO pins.
- **Two PLLs** to provide a fixed 48MHz clock for USB or ADC, and a flexible system clock up to 133MHz.
- An **internal Voltage Regulator** to supply the core voltage so the end product only needs supply the IO voltage.

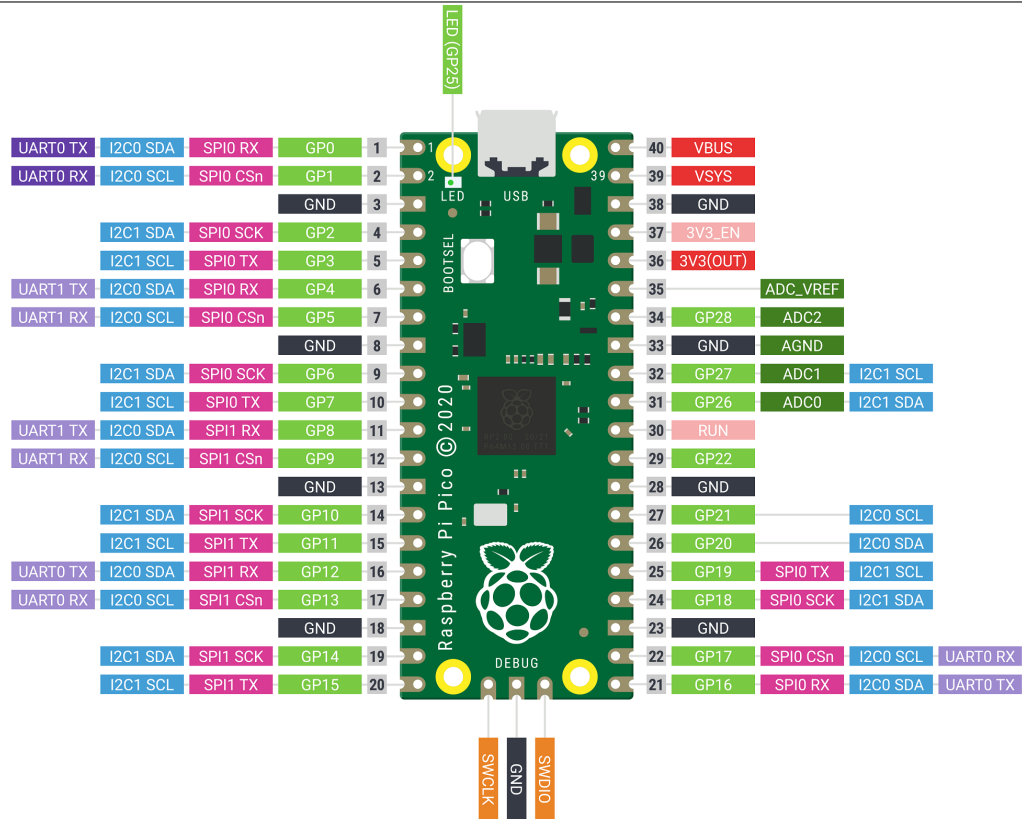


19



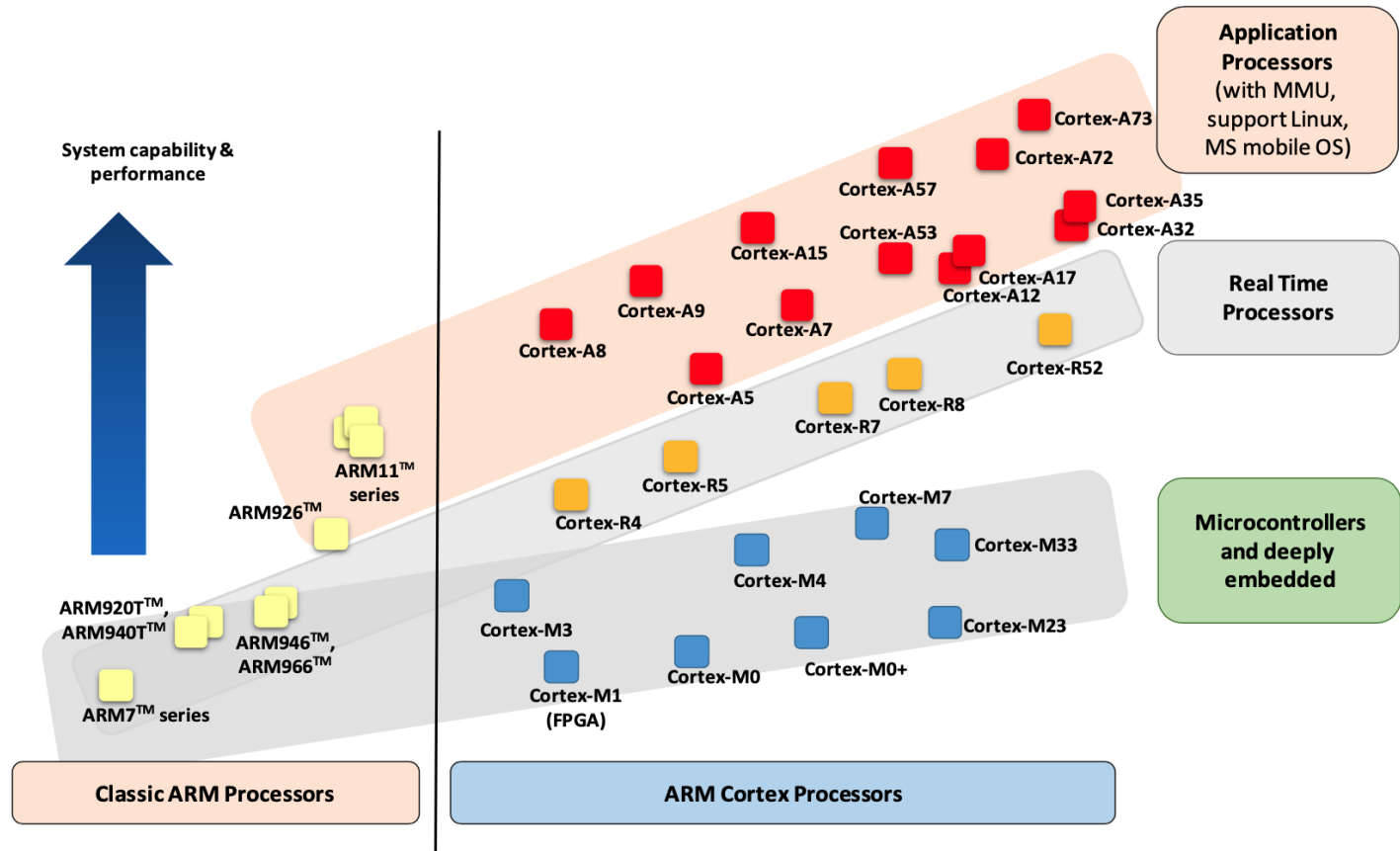
Présentation du Raspberry Pico : le brochage ou «*pinout*»

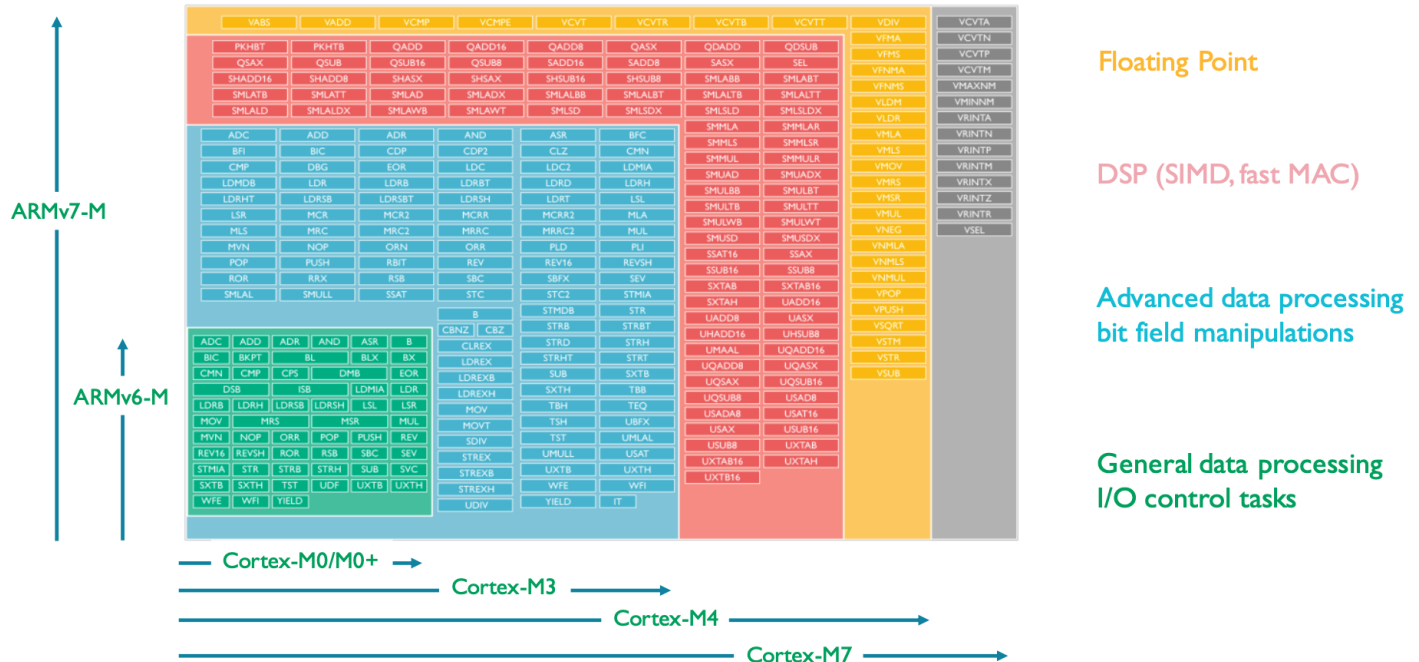
20

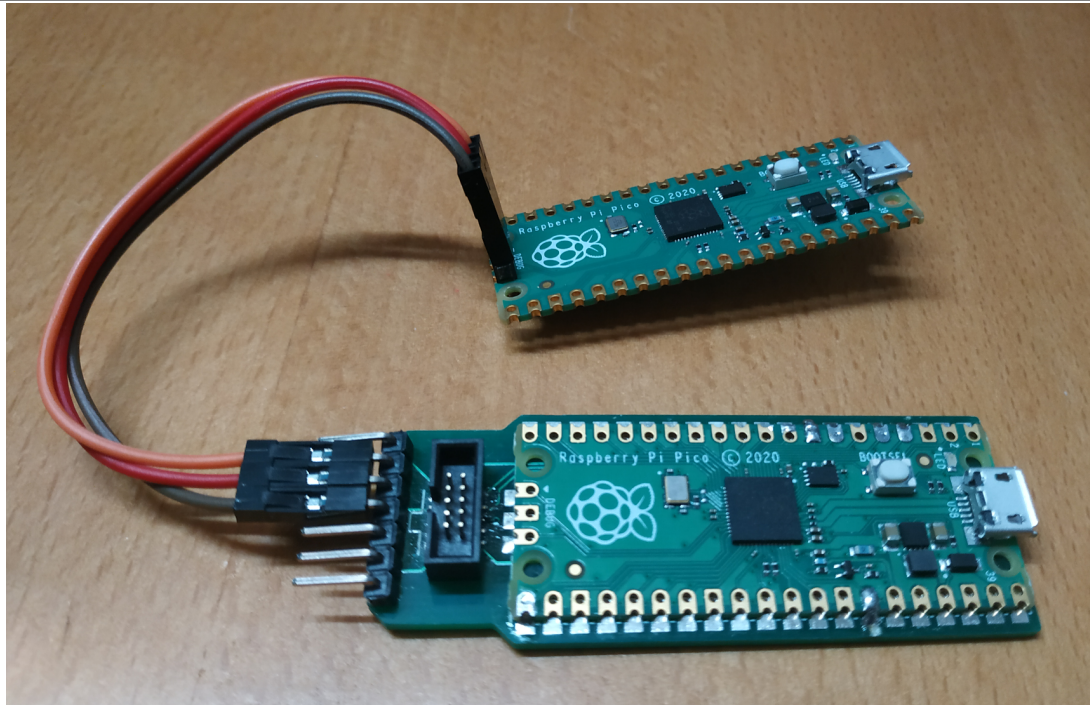


The ARM Cortex-M0+ processor features and benefits are:

- ❑ **Tight integration** of system peripherals reduces area and development costs.
- ❑ **Thumb instruction** set combines high code density with 32-bit performance.
- ❑ Support for **single-cycle I/O** access.
- ❑ **Power control** optimization of system components.
- ❑ **Integrated sleep modes** for low-power consumption.
- ❑ **Fast code execution** enables running the processor with a slower clock or increasing sleep mode time.
- ❑ **Optimized code fetching** for reduced flash and ROM power consumption.
- ❑ Hardware **multiplier**.
- ❑ **Deterministic**, high-performance «interrupt handling» for time-critical applications.
- ❑ **Deterministic instruction cycle timing**.
- ❑ Support for **system level debug authentication**.
- ❑ **Serial Wire Debug** reduces the number of pins required for debugging.







- ▷ le premier Raspberry Pi Pico sert de **sonde**, «*probe*», de débogage ;
- ▷ le second est sous contrôle matériel du premier :
 - ◇ on peut interrompre/reprendre l'exécution de son firmware ;
 - ◇ on peut connaître le contenu des registres et de la mémoire lors de l'arrêt de l'exécution.

On liste les périphériques USB connectés :

```
xterm
$ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 007: ID 2e8a:0005 MicroPython Board in FS mode
```

La connexion du Raspberry Pico crée un «*device*» série :

```
xterm
$ ls /dev
... ttyACM0 ...
```

On peut voir les événements «*hardware*» :

```
xterm
$ sudo dmesg
...
[ 2049.165618] usb 1-9.4: new full-speed USB device number 7 using xhci_hcd
[ 2049.271582] usb 1-9.4: New USB device found, idVendor=2e8a, idProduct=0005, bcdDevice= 1.00
[ 2049.271596] usb 1-9.4: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 2049.271602] usb 1-9.4: Product: Board in FS mode
[ 2049.271607] usb 1-9.4: Manufacturer: MicroPython
[ 2049.271611] usb 1-9.4: SerialNumber: e6614103e76d1f22
[ 2049.309954] cdc_acm 1-9.4:1.0: ttyACM0: USB ACM device
[ 2049.310040] usbcore: registered new interface driver cdc_acm
[ 2049.310045] cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters
[ 3319.456183] usb 1-9.4: USB disconnect, device number 7
[ 3492.738513] usb 1-9.4: new full-speed USB device number 8 using xhci_hcd
[ 3492.840166] usb 1-9.4: New USB device found, idVendor=2e8a, idProduct=0004, bcdDevice= 1.00
[ 3492.840181] usb 1-9.4: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 3492.840187] usb 1-9.4: Product: Picoprobe
[ 3492.840192] usb 1-9.4: Manufacturer: Raspberry Pi
[ 3492.840195] usb 1-9.4: SerialNumber: E6614C30930D512E
[ 3492.855791] cdc_acm 1-9.4:1.0: ttyACM0: USB ACM device
```

nouveau périphérique USB connecté

création du device série lié au Raspberry Pico branché en USB

```
xterm
$ minicom -D /dev/ttyACM0

Welcome to minicom 2.8

OPTIONS: I18n
Port /dev/ttyACM0, 20:23:10

Press CTRL-A Z for help on special keys

>>>
MPY: soft reboot
MicroPython v1.19.1 on 2022-06-18; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> help()
Welcome to MicroPython!

For online help please visit https://micropython.org/help/.

For access to the hardware use the 'machine' module.  RP2 specific commands
are in the 'rp2' module.

Quick overview of some objects:
  machine.Pin(pin) -- get a pin, eg machine.Pin(0)
  machine.Pin(pin, m, [p]) -- get a pin and configure it for IO mode m, pull mode p
  methods: init(..), value([v]), high(), low(), irq(handler)
...
  machine.Timer(freq, callback) -- create a software timer object
  eg: machine.Timer(freq=1, callback=lambda t:print(t))

Pins are numbered 0-29, and 26-29 have ADC capabilities
Pin IO modes are: Pin.IN, Pin.OUT, Pin.ALT
Pin pull modes are: Pin.PULL_UP, Pin.PULL_DOWN

Useful control commands:
  CTRL-C -- interrupt a running program
  CTRL-D -- on a blank line, do a soft reset of the board
  CTRL-E -- on a blank line, enter paste mode

For further help on a specific object, type help(obj)
For a list of available modules, type help('modules')
```

Faire clignoter une LED

```
xterm
>>> import machine
>>> led = machine.Pin(25, machine.Pin.OUT)
>>> led.toggle()
>>> led.toggle()
>>> def blink(timer):
...     led.toggle()
...
...
...
>>> timer.init(freq=2.5, mode=machine.Timer.PERIODIC, callback=blink)
```

On allume la LED

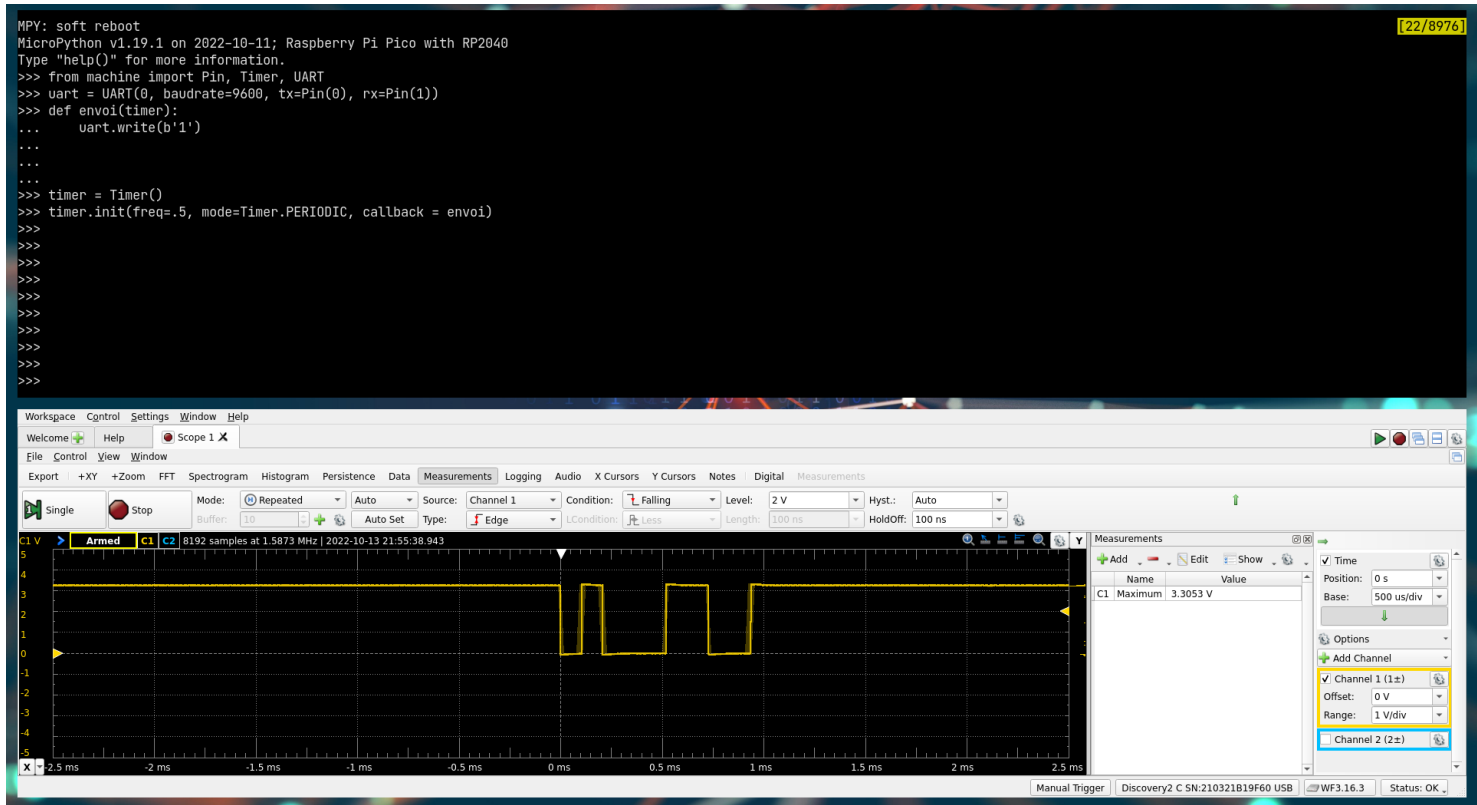
On éteint la LED

Utiliser le port série

```
xterm
MPY: soft reboot
MicroPython v1.19.1 on 2022-10-11; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> from machine import Pin, Timer, UART
>>> uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))
>>> def envoi(timer):
...     uart.write(b'l')
...
...
...
>>> timer = Timer()
>>> timer.init(freq=.5, mode=Timer.PERIODIC, callback = envoi)
```

On utilise les broches 0 et 1 du Raspberry Pico

À l'aide d'un oscilloscope on peut «*intercepter*» la transmission série sur la broche 0 :



Lorsque le port série ne transmet rien, la broche est au niveau haut, c-à-d à 3,3v.

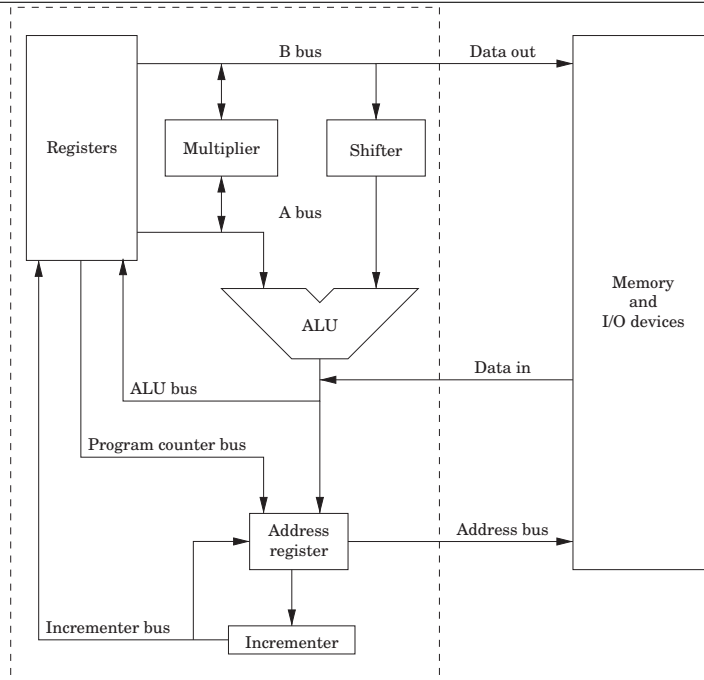
Grâce à l'**oscilloscope**, on peut déterminer la **vitesse de transmission** :



Ici, on voit que $1/\Delta X = 9,6\text{KHz}$ ce qui est le cas : on a configuré le port série pour une transmission à 9600bps :

```
xterm  
...  
>>> uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))  
...
```

Assembleur ARM



- **deux registres** peuvent servir de **source** pour une instruction en passant par les bus A et B ;
- les données sur le bus B passent par un «*shifter*» : on peut décaler la seconde opérande avant qu'elle atteigne l'ALU ;
- les bus A et B peuvent fournir des **opérandes** pour le «*multiplier*» et le «*multiplier*» peut fournir des données pour les bs A et B ;
- les données en **lecture** depuis la mémoire ou des I/Os peuvent aller directement dans l'ALU puis dans un registre.
- les données en **écriture** vers la mémoire ou dans les I/Os sont prises directement dans le bus B, qui peuvent provenir de registres, mais ces données ne peuvent être modifiées sur le chemin.

- le **registre d'adresse** est un registre temporaire utilisé à chaque opération de lecture/écriture mémoire/I/Os.
Peut être chargé :
 - ◇ depuis le «*program counter*» pour **chercher**, «*fetch*», la **prochaine instruction** ;
 - ◇ depuis l'ALU pour permettre des **modes d'adressages** où un registre est utilisé comme **adresse de base** et un **décalage** est calculé à la volée.
 Après l'accès, l'**adresse de base** peut être **incrémentée** et cette valeur **stockée** dans un registre ;
 ⇒ utilisé pour **incrémenter** le «*program counter*» à chaque instruction ;
 ⇒ utilisé pour certains mode d'adressage où un pointeur est **incrémenté** à chaque accès mémoire.

Le processeur dispose de 16 registres

- **R0 à R12** : 13 registres à usage générique utilisable comme on le veut ;
 - **R13** : le registre de pile, «*stack*» ;
 - **R14** : le registre de lien, «*link*» ;
 - **R15** : le «*program counter*» ou registre ordinal.
 - le **CPSR**, «*Current Program Status Register*» : registre d'état contient des bits d'information sur la dernière instruction utilisée. Utilisé notamment pour les conditions et branchements.
- ❶ Ces registres sont utilisés dans le cas des appels de fonction :
- Le «*link register*» contient l'adresse de retour après exécution de la fonction.
- Le «*stack register*» contient l'adresse du sommet de la pile, où on empilera les valeurs courantes des registres, et pour créer les variables locales de la fonction appelée.

Exécution des instructions

Effet Pipeline

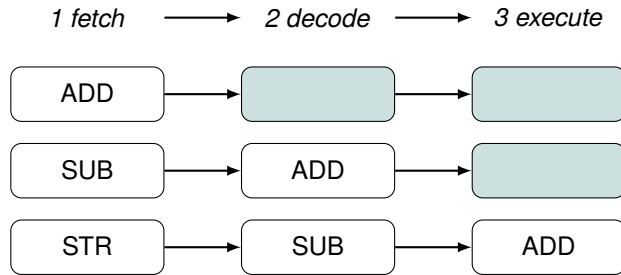
Chaque instruction est exécutée en **trois cycles d'horloge** :

- ▷ un cycle pour le **chargement de l'instruction** depuis la mémoire : *premier étage du pipeline* ;
- ▷ un cycle pour **décoder l'instruction** : *second étage du pipeline* ;
- ▷ un cycle pour **l'exécution** : *troisième étage du pipeline* ;

Lorsqu'une instruction quitte le premier étage du pipeline, une nouvelle instruction peut y entrer :

⇒ une instruction **sort à chaque cycle** du pipeline

⇒ une séquence d'instruction est exécutée **en un cycle chacune** !



En ARMv7 le pipeline a 3 étapes :

- ❑ **fetch** : charge une instruction depuis la mémoire ;
- ❑ **decode** : identifie l'instruction à exécuter et établit les chemins passant par les bus d'échanges ;
- ❑ **execute** : exécute l'instruction et écrit le résultat dans un registre.

Pipeline et «program counter»

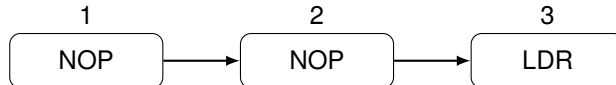
0x8000	LDR pc, [pc, #0]
0x8004	NOP
0x8008	NOP
0x800C	DCD JumpAddress

La notation assembleur `LDR pc, [pc, #0]` se traduit par :

- ▷ charge dans PC, «program counter» ❶ ;
 - ▷ le contenu de PC, «program counter», + 0 ❷.
- ⇒ ce qui permet un adressage par décalage.

Mais que vaut PC ?

⇒ cela dépend du pipeline d'instruction :



Le PC évolue de la manière suivante :

1. étape 1 du pipeline : PC = 0x8000 pour l'instruction LDR ;
2. étape 2 du pipeline : PC = 0x8004 pour l'instruction NOP ;
3. étape 3 du pipeline : PC = 0x8008 pour l'instruction NOP ;

⇒ lorsque l'instruction LDR est exécutée dans l'étape 3 du pipeline, PC vaut 0x8008 !

31-28	27-25	24-21	20	19-16	15-12	11-0
Condition	Operand type	OpCode	Set Condition Codes	Operand register	Destination register	Immediate Operand

- ❑ **Condition** : autorise l'exécution de l'instruction suivant les bits dans le registre CPSR ;
- ❑ **Operand type** : spécifie le format de l'opérande des bits 19-0 :
 - ◇ par exemple deux registres suivis d'une opérande immédiate ;
- ❑ **Opcode** : quelle instruction on veut réaliser comme Add ou MUL ;
- ❑ **Set condition code** : un seul bit indiquant si l'instruction doit mettre à jour le registre CPSR, valeur 0, ou non, valeur 1 ;
- ❑ **Operand register** : un registre à utiliser comme entrée ;
- ❑ **Destination register** : un registre à utiliser comme sortie ;
- ❑ **Immediate operand** : une donnée de petite taille que l'on peut donner directement dans l'instruction.
 - ◇ exemple : si on veut ajouter 1 à un registre, on peut mettre la donnée à 1 ce qui évite de mettre 1 dans un autre registre et de faire la somme de ces deux registres.

Architecture 32 bits et 64 bits

- En **32bits** :
 - ◇ les **adresses mémoires** sont sur 32 bits ;
 - ◇ les **registres du processeur** sont sur 32 bits ;
- En **64bits** :
 - ◇ les **adresses mémoires** sont sur 64 bits ;
 - ◇ les **registres du processeur** sont sur 64 bits ;
- En **32bits** comme en **64bits**, les **instructions sont sur 32bits** :
 - ◇ *Comment peut-on charger une variable depuis la mémoire dans un registre donné avec une instruction sur 32bits ?*
 - ★ l'instruction fait 32bits ;
 - ★ 4bits sont utilisés pour un opcode ;
 - ★ 4bits pour une instruction conditionnelle ;
 - ★ 3bits sont utilisés pour indiquer le type de l'opérande ;
 - ★ 1bit pour indiquer si l'opération affecte le CPSR ;
 - ★ 4bits sont nécessaires pour indiquer le registre ;

⇒ **il reste 16bits pour indiquer l'adresse mémoire !**

⇒ **il reste 12bits pour indiquer l'adresse mémoire si on a besoin d'indiquer 2 registres !**
- Comment faire ?
 - ◇ on peut utiliser un registre pour indiquer l'adresse mémoire : **accès mémoire indirect**
 - ◇ Mais comment charger l'adresse mémoire initialement dans le registre en une seule instruction ?
 - ◇ charger dans deux registres séparés l'adresse désirée, puis décaler et combiner les deux registres en un seul registre ⇒ 4 instructions pour arriver au résultat ce qui est excessif !
 - ◇ Et alors ? On peut utiliser le PC, «*Program Counter*» pour charger une **mémoire pas trop éloignée**, à 12bits de décalage par rapport au PC, soient **4096 octets accessibles** et beaucoup plus **par décalage de ces bits**.

Définition de contenu

```
label: .byte 74, 0112, 0b00101010, 0x4A, 0X4a, 'J', 'H' + 2
      .word 0x1234ABCD, -1434
      .ascii "Hello World\n"
```

Charger un registre

L'instruction `LDR` peut servir à charger une adresse dans le registre ou la donnée pointée par cette adresse.

Il est également possible d'indexer la mémoire en utilisant la valeur d'un registre comme valeur d'index.

- ▷ adressage relatif au PC, «*Program Counter*» ;
- ▷ charger depuis la mémoire ;
- ▷ indexer à travers la mémoire.

Adressage relatif au PC

Si les données ne sont pas loin de l'instruction utilisant leur adresse, on peut utiliser cet adressage :

```
LDR R1,=helloworld
```

ce qui donne après assemblage :

```
LDR R1, [pc, #20]
```

Ici, la valeur du décalage est de $0x20$ en hexa, soit 32.

Ce décalage est :

- ▷ déterminé lors de l'assemblage ;
- ▷ sur 12 bits dans l'instruction elle-même :
 - ◊ ce qui donne de 0 à 4095 ;
 - ◊ un bit est utilisé dans l'instruction pour indiquer dans quel sens :
⇒ le décalage peut être de ± 4095 words au maximum.
 - ◊ le décalage peut être par multiple de 1 à 2 octets, comme indiqué dans la table ⇒

`LDR{type} Rt, =label`

Type	Meaning
B	Unsigned byte
SB	signed byte
H	Unsigned halfword (16 bits)
SH	signed halfword (16 bits)
–	omitted for word

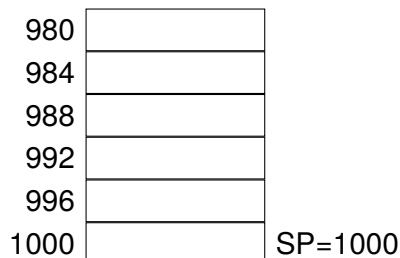
Utilisation de la pile

Deux opérations possibles :

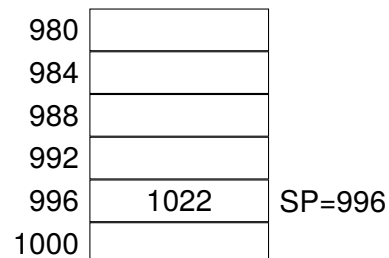
- ▷ **push** : ajouter un élément ;
- ▷ **pop** : enlever et retourner l'élément le plus récemment ajouté.

Gestion dans le processeur ARM :

- **registre R13** aussi appelé **SP**, «*Stack Pointer*» : il pointe sur l'emplacement de la pile en mémoire ;
- deux instructions du jeu d'instruction ARM32 : **LDM**, «*Load Multiple*» et **STM**, «*Store Multiple*» ;
- ces instructions sont adaptables :
 - ◇ choix du sens d'augmentation de la pile par incrémentation ou décrémentation des adresses ;
 - ◇ le registre pointe soit sur la fin de la pile, soit sur le prochain emplacement libre ;*⇒ le système est adaptable aux besoins de différents systèmes d'exploitation.*
- l'assembleur GNU offre des pseudo-instructions qui s'appuient sur les instructions LDM et STM :
 - ◇ **PUSH** liste de registre : `PUSH {R0, R5-R12}` ;
 - ◇ **POP** liste de registres : `POP {R0-R4, R6, R9-R12}` ;



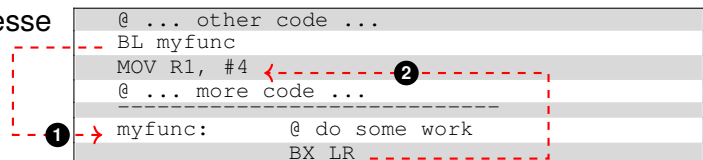
Push R5
R5 = 1022



Branchement avec lien de retour

Après l'appel de la fonction, il faut retourner à l'exécution des instructions qui suivent :

- le registre R14, «*link register*» : sert à stocker l'adresse de retour ;
- l'instruction BL, «*Branch with Link*» : réalise un branchement après avoir stocké l'adresse de la prochaine instruction dans LR ;
- l'instruction BX : réalise le retour de fonction en sautant à l'adresse présente dans le registre LR ;
- ▷ BL saute à l'adresse de ❶ et sauvegarde l'adresse de retour dans LR ;
- ▷ BX saute à l'adresse stockée dans LR ❷



Et comment cela se passe si la fonction appelle une autre fonction ?

On utilise de nouveau dans la fonction `myfunc`, l'instruction BL :

⇒ BL **copie** l'adresse de la prochaine instruction dans le registre LR

⇒ *ce qui écrase l'ancienne valeur contenue dans LR*

⇒ la fonction `myfunc` **ne pourra plus retourner** !

⇒ il faut :

- ▷ **sauvegarder** la valeur du registre LR dans la pile❸ avant le BL
- ▷ **restaurer** la valeur du registre LR❹ avant de retourner avec BX

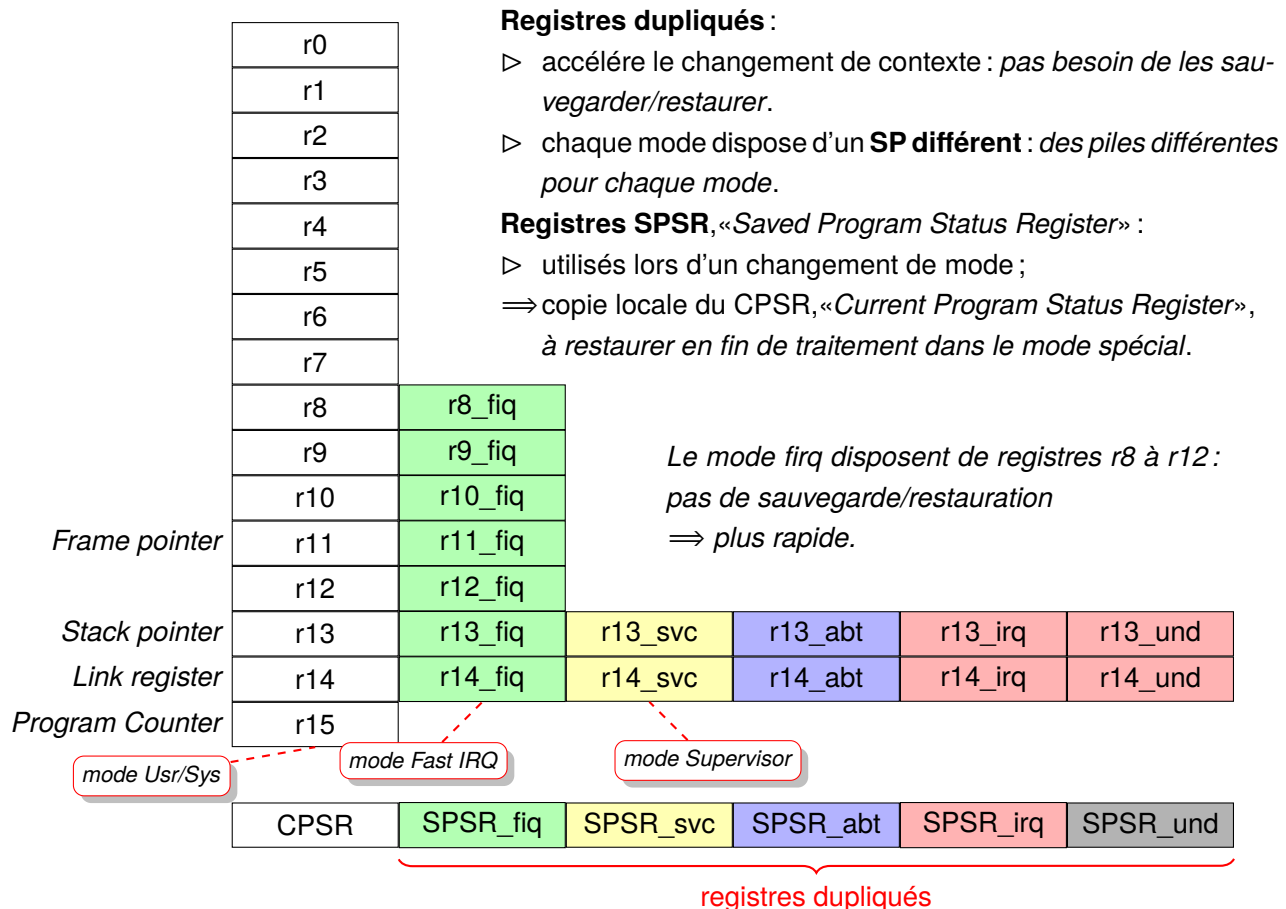
```
@ ... other code ...
BL myfunc
MOV R1, #4
@ ... more code ...
-----
myfunc: PUSH {LR}❸
        @ do some work ...
        BL myfunc2
        @ do some more work...
        POP {LR}❹
        BX LR
myfunc2: @ do some work
        BX LR
```

- 1 mode utilisateur, «*usr*», non privilégié ;
- 5 modes pour des exceptions + 1 pour le système, privilégiés :

Mode	Sigle	Description
User	usr	mode normal d'exécution non privilégié
Interrupt	_irq	activé lors d'une exception causée par une interruption matérielle
Fast Interrupt	_fiq	activé lors d'une exception causée par une interruption matérielle à gérer rapidement
Abort	_abt	activé lors d'une exception d'accès à la mémoire
Undefined	_und	activé lorsqu'une instruction indéfinie est exécuté (une valeur sur 32bits qui ne correspond à aucune instruction existante)
Supervisor	_svc	Software Interrupt: activé lors du reset ou lors d'un appel système
System	sys	mode privilégié pour le travail de l'OS.

- le passage d'un mode à l'autre peut être fait **manuellement** en modifiant des bits du registre CPSR ;
- les **modes privilégiés** sont activés pour traiter des **interruptions** ou des **exceptions** ;
- le **mode système**, «*sys*», est un **mode spécial** pour accéder à des **ressources protégées** (comme la MMU par exemple si disponible) ;
- les modes traitant les **interruptions** disposent de **registres dupliqués**, remplaçant les registres qu'ils dupliquent durant l'exécution du traitement, ce qui évitent les collisions et corruptions.
- les modes «*usr*» et «*sys*» **partagent le même ensemble** de registres.

Registres et mode d'exécution : 37 registres au total pour 7 modes d'opérations 40



Mais ça marche comment les interruptions ?

Assigner les interruptions

C'est le **designer du SoC** qui décide quel matériel peut produire quelle interruption : il s'agit d'interconnecter des circuits avec le processeur.

- ▷ SWI, «*SoftWare Interrupt*» ou svc, utilisée pour accéder à des fonctions privilégiées de l'OS ;
- ▷ IRQ, assignées à des interruptions génériques comme des timers périodiques ;
- ▷ FIQ, réservée pour une seule source qui nécessite un faible temps de réponse.

Les latences dans le traitement d'une interruption

La **latence** est l'intervalle de temps entre :

- ▷ l'instant où le signal externe change d'état ;
- ▷ la première instruction du traitement est chargé dans le processeur (fetch) ;

Le système essaie d'atteindre deux buts :

- ☐ gérer **plusieurs interruptions** simultanément ;
- ☐ **minimiser** la latence.

Deux méthodes pour l'atteindre :

- ▷ autoriser la **gestion imbriquée**, «*nested*», des interruptions ;
- ▷ donner des **priorités** aux différentes sources d'interruption.

Activer/désactiver les interruptions

Les instructions dédiées :

MRS	lire le CPSR
MSR	stocker dans le CPSR
BIC	effacer un bit
ORR	opération OR

Activer une IRQ / FIRQ :

```
MRS r1, cpsr
BIC r1, r1, #0x80/0x40
MSR cpsr_c, r1
```

Désactiver une IRQ / FIRQ :

```
MRS r1, cpsr
ORR r1, r1, #0x80 / 0x40
MSR cpsr_c, r1
```

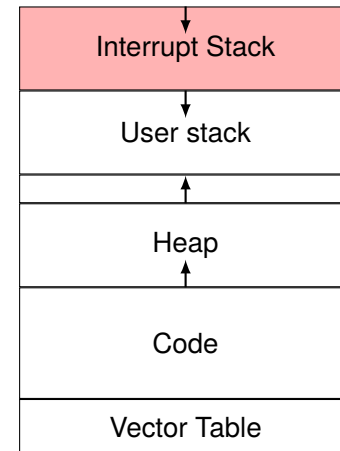
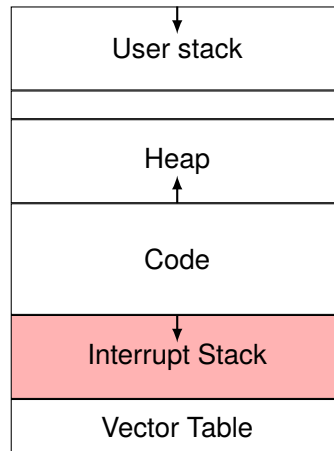
Pourquoi une pile ?

La pile est nécessaire pour le changement de contexte :

- ▷ empiler les valeurs des registres pour les sauvegarder ;
- ▷ dépiler les valeurs pour restaurer les registres ;
- ▷ variables locales nécessaires pour la gestion des interruptions.

Il faut choisir :

- ☐ la taille de la pile ;
- ☐ sa position dans la mémoire.



Dans ce cas là, la table des vecteurs d'interruption ne peut être écrasée lors d'un dépassement de pile.

Table des vecteurs d'interruption

44

La **table des vecteurs d'interruption** est une table d'adresses :

- ▷ le processeur ARM **saute** à l'adresse associée à l'exception qui s'est déclenchée ;
- ▷ à cette adresse se trouve une **instruction de saut** vers le code de traitement de cette exception.

Dans le tableau, on trouve une instruction de branchement :

```
ldr pc, [pc, #_IRQ_HANDLER_OFFSET].....
```

Où #_IRQ_HANDLER_OFFSET est le décalage pour atteindre le code de gestion de l'interruption.

Adresse	Exception	Mode en entrée
0x00000000	Reset	Superviseur
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIRQ	FIRQ

Cette table doit être remplie par l'OS lors du démarrage de la machine.

Priorités

indique si le handler de l'interruption peut être ou non interrompu

définie quelle exception est la plus importante parmi celles déclenchées

Exception	Priorité	bit I	bit F
Reset	1	1	1
Data Abort	2	1	-
FIQ	3	1	1
IRQ	4	1	-
Prefetch Abort	5	1	
SWI	6	1	-
Undefined Instruction	6	1	-

déclenchée lors de l'étape
«execute» du pipeline

Décalage du registre PC sauvegardé dans le registre LR

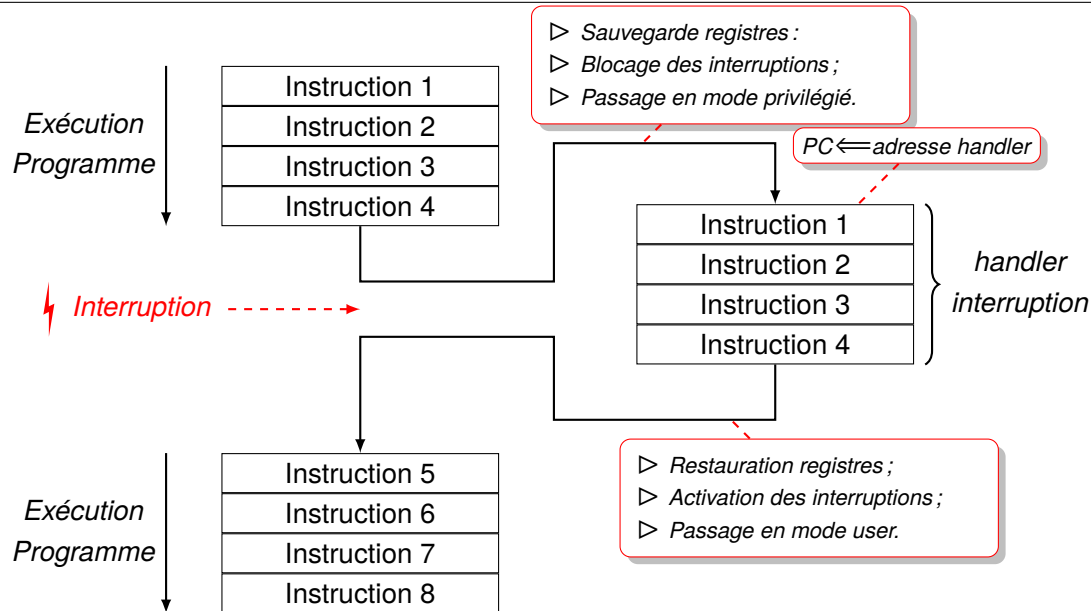
Exception	Adresse de retour
Reset	aucune
Data Abort②	LR - 8
FIQ, IRQ, prefetch Abort②	LR - 4
SWI, Undefined Instruction	LR

Dans certain cas, le registre PC a avancé au delà de l'instruction qui a causé l'exception :

▷ en ①, il faut revenir en arrière de deux instructions par rapport à la valeur du PC sauvegardée dans le registre LR ;

▷ en ②, il faut revenir en arrière d'une seule instruction.

Ceci est du au pipeline d'exécution du processeur ARM.



Entrée dans le handler d'exception :

- ⇒ sauver l'adresse de l'instruction suivante dans le registre LR approprié ;
- ⇒ copier le CPSR dans le SPSR du nouveau mode ;
- ⇒ changer le mode en modifiant les bits du CPSR ;
- ⇒ aller chercher l'instruction suivante dans la table des vecteurs d'interruption.

Sortie du handler d'exception :

- ⇒ charger le registre LR dans le PC (avec le bon décalage) ;
- ⇒ restaurer le SPSR dans le CPSR, ce qui remettra les bits de mode comme avant ;
- ⇒ effacer le bit de blocage des interruptions s'il avait été positionné.

Et sur un exemple d'interruption logicielle
ça donne quoi ?

On va utiliser l'assembleur `as` pour assembler l'exécutable :

```
@
@ Assembler program to print "Hello World!"
@ to stdout.
@
@ R0-R2 - parameters to linux function services
@ R7 - linux function number
@

.global _start @ Provide program starting
@ address to linker
@ Set up the parameters to print hello world
@ and then call Linux to do it.
_start: mov R0, #1 @ 1 = StdOut
        ldr R1, =helloworld @ string to print
        mov R2, #13
        mov R7, #4
        svc 0 @ Call linux to print.

@ Set up the parameters to exit the program
@ and then call Linux to do it.
@ length of our string
@ linux write system call
        mov R0, #0 @ Use 0 return code
        mov R7, #1 @ Service command code 1
        @ terminates this program
        svc 0 @ Call linux to terminate.

.data
helloworld: .ascii "Hello World!\n"
```

l'étiquette `_start` indique au «linker» le point d'entrée du programme

On utilise un appel système pour faire l'affichage

On utilise un appel système pour terminer le programme

.ascii indique à l'assembleur de mettre la chaîne dans le segment data, et d'utiliser l'étiquette pour avoir son adresse

Assemblage et exécution :

```
❏ — xterm —
$ as -o HelloWorld.o HelloWorld.s
$ ld -o HelloWorld HelloWorld.o
$ ./HelloWorld
Hello World!
$
```

- ▷ l'instruction `MOV` déplace des données dans un registre.
- ▷ le `#4` indique une opérande directe, soit la valeur 4.
- ▷ `LDR R1, =helloworld` charge le registre avec l'adresse de la chaîne.
- ▷ `svc 0` réalise une interruption logicielle pour donner le contrôle au noyau Linux.

Comment utiliser une fonctionnalité de l'OS ?

```
_start: mov R0, #1 @ 1 = StdOut
        ldr R1, =helloworld @ string to print
        mov R2, #13
        mov R7, #4
        svc 0 @ Call linux to print
```

Les **registres R0 à R4** vont être utilisés pour passer les paramètres à l'appel système.

Lors du retour de l'appel système, la **valeur de retour** sera donnée dans le registre R0.

Ici, on fait appel à l'appel système `print` :

- ▷ on indique dans le registre R0 la destination `stdout` ;
- ▷ l'adresse de la chaîne à afficher est mise dans le registre R1 ;
- ▷ le numéro de l'appel système est donné dans le registre R7, ici c'est la valeur 4 pour indiquer `print`.

On réalise ensuite une **interruption logicielle** avec l'instruction `svc 0` :

- ▷ le contrôle, le CPU, est transmis au code du traitement de cette interruption ;
- ▷ le **numéro** de l'appel système permet de savoir quel code doit être appelé pour réaliser l'appel système ;
- ▷ le code de traitement de l'interruption et de l'appel système est dans le noyau Linux ;
- ▷ grâce au **mécanisme d'interruption** :
 - ◇ le programme **ne sait pas où se trouve** ce code de traitement : *il n'y a même pas accès pour cause de protection d'accès mémoire* ;
 - ◇ le code de traitement est **exécuté dans un mode protégé** du processeur : *il accède à toutes les ressources de la machine, comme l'écran pour y afficher du texte*.
 - ◇ si le code de traitement de l'appel système est **mis à jour**, il n'y a **pas de problème** : *le programme utilise simplement un numéro pour l'identifier*.

```
$ objdump -s -d HelloWorld.o
HelloWorld.o:          file format elf32-littlearm

Contents of section .text:
 0000 0100a0e3 14109fe5 0d20a0e3 0470a0e3 .....p..
 0010 000000ef 0000a0e3 0170a0e3 000000ef .....p.....
 0020 00000000                ....

Contents of section .data:
 0000 48656c6c 6f20576f 726c6421 0a          Hello World!.
Contents of section .ARM.attributes:
 0000 41110000 00616561 62690001 07000000  A....aeabi.....
 0010 0801                ..

Disassembly of section .text:

00000000 <_start>:
 0: e3a00001      mov     r0, #1
 4: e59f1014      ldr     r1, [pc, #20] ; 20 <_start+0x20>
 8: e3a0200d      mov     r2, #13
c: e3a07004      mov     r7, #4
10: ef000000      svc     0x00000000
14: e3a00000      mov     r0, #0
18: e3a07001      mov     r7, #1
1c: ef000000      svc     0x00000000
20: 00000000      .word   0x00000000
```

instructions en mode LittleEndian

les différents segments du programme

Le segment de code

mnémonique

*L'intérêt du mode LittleEndian ?
pour convertir un entier sur 4 octets en 1 octet,
il suffit de lire que le premier octet.*

valeur réelle de l'instruction

- ▷ il n'y a plus d'étiquettes comme dans le source assembleur : elles sont remplacées par des adresses ;
- ▷ les **instructions** sont indiquées avec **leur valeur réelle**, indiquée en **hexadécimal** en plus de leur notation sous forme de **mnémonique**, comme `e3a00001` pour `mov r0, #1`.

Décomposition d'une instruction

00000000	<_start>:		
0:	e3a00001	mov	r0, #1
4:	e59f1014	ldr	r1, [pc, #20] ; 20 <_start+0x20>
8:	e3a0200d	mov	r2, #13
c:	e3a07004	mov	r7, #4
10:	ef000000	svc	0x00000000

Soit l'instruction e3a00001 mov r0, #1:

Hex Digit	e	3	a	0	0	0	0	1
Binary	1110	0011	1100	0000	0000	0000	0000	0001

- Chaque instruction du code commence par un digit hexa à e :
 - ◇ champs sur 4bits indiquant une exécution conditionnelle suivant les valeurs du registre CSPR ;
 - ◇ *ici, la valeur est e qui indique que l'instruction doit être réalisée de manière inconditionnelle*
- les 3bits suivants 001 indique le **type des opérandes** :
 - ◇ *ici, un registre et une valeur immédiate ;*
- les 4bits suivants 1110 est l'**opcode** pour l'instruction MOV ;
- le bit suivant 0 indique le **type du paramètre** pour le mode immédiat, qui ici ne sert pas ;
- les 4bits suivants 0000 indiquent le **registre R0** ;
- les 4bits suivants 0000 indiquent un **autre registre** dans le cas où le MOV travaillerait sur deux registres *ce qui n'est pas le cas ici* ;
- les 12bits restants : la **valeur immédiate**, *ici 1*.

Le source :

```
_start: mov R0, #1 @ 1 = StdOut
        ldr R1, =helloworld @ string to print
        mov R2, #13
        mov R7, #4
        svc 0 @ Call linux to print
```

est devenu :

```
00000000 <_start>:
0:  e3a00001      mov     r0, #1
4:  e59f1014      ldr     r1, [pc, #20] ; 20 <_start+0x20>
8:  e3a0200d      mov     r2, #13
c:  e3a07004      mov     r7, #4
10: ef000000      svc     0x00000000
```

ldr R1, =helloworld est devenu ldr r1, [pc, #20]; 20 <_start+0x20>

L'assembleur :

- ▷ l'étiquette =helloworld a été traduite en un décalage, «*offset*», depuis le «*program counter*» ;
- ▷ la chaîne de caractères a été placée 20*4 octets plus loin que l'origine du programme indiqué par l'étiquette _start, c-à-d :

```
1c:  ef000000      svc     0x00000000
20:  00000000      .word  0x00000000
```

juste après le code.

- ▷ l'accès à cette mémoire est fait depuis un registre contenant déjà une adresse, car une seule instruction ARM ne permet pas de charger une **adresse entière sur 32bits** directement dans un registre.

\$ gdb helloworld	
GNU gdb (Raspbian 10.1-1.7) 10.1.90.20210103-git	
Copyright (C) 2021 Free Software Foundation, Inc.	
Reading symbols from helloworld...	
(gdb) list	GDB
1	
2 @	Command (short form)
3 @ Assembler program to print "Hello World!"	Description
4 @ to stdout.	list (l)
5 @	break (b) line
6 @ R0-R2 - parameters to linux function services	run (r)
7 @ R7 - linux function number	step (s)
8 @	continue (c)
9	quit (q or control-d)
10 .global _start @ Provide program starting	control-c
(gdb) l	info registers (i r)
11 @ address to linker	info break
12 @ Set up the parameters to print hello world	delete n
13 @ and then call Linux to do it.	x /nuf expression
14 _start: mov R0, #1 @ 1 = StdOut	
15 ldr R1, =helloworld @ string to print	
16 mov R2, #13	
17 mov R7, #4	
18 svc 0 @ Call linux to print	
19	
20 @ Set up the parameters to exit the program	
(gdb) l	
21 @ and then call Linux to do it.	
22 @ length of our string	
23 @ linux write system call	
24 mov R0, #0 @ Use 0 return code	
25 mov R7, #1 @ Service command code 1	
26 @ terminates this program	
27 svc 0 @ Call linux to terminate	
28 .data	
29 helloworld: .ascii "Hello World!\n"	
(gdb)	

```

xterm
(gdb) disassemble _start
Dump of assembler code for function _start:
0x00010074 <+0>:  mov    r0, #1
0x00010078 <+4>:  ldr     r1, [pc, #20] ;
                   0x10094 <_start+32>
0x0001007c <+8>:  mov     r2, #13
0x00010080 <+12>: mov     r7, #4
0x00010084 <+16>: svc     0x00000000
0x00010088 <+20>: mov     r0, #0
0x0001008c <+24>: mov     r7, #1
0x00010090 <+28>: svc     0x00000000
0x00010094 <+32>: muleq   r2, r8, r0
End of assembler dump.
(gdb)

```

décalage

```

xterm
(gdb) b _start
Breakpoint 1 at 0x10074: file HelloWorld.s, line 14.
(gdb) r
Starting program:
/home/pi/ASMRASPI/hello_world_asm/helloworld
Breakpoint 1, _start () at HelloWorld.s:14
14  _start: mov R0, #1 @ 1 = StdOut
(gdb) s
15      ldr R1, =helloworld @ string to print
(gdb) info registers
r0             0x1             1
r1             0x0             0
r2             0x0             0
r3             0x0             0
r4             0x0             0
r5             0x0             0
r6             0x0             0
r7             0x0             0
r8             0x0             0
r9             0x0             0
r10            0x0             0
r11            0x0             0
r12            0x0             0
sp             0x7efff620      0x7efff620
lr             0x0             0
pc             0x10078         0x10078 <_start+4>
cpsr           0x10             16
fpscr          0x0             0
(gdb) info breakpoints
Num   Type             Disp Enb Address      What
1     breakpoint       keep y  0x00010074
HelloWorld.s:14
      breakpoint already hit 1 time

```

décalage

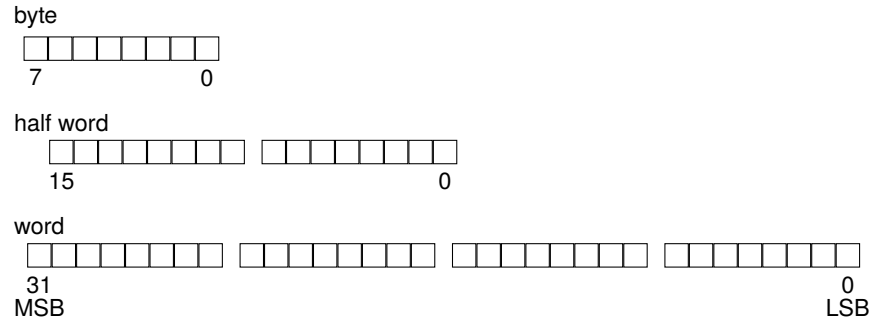
```

xterm
(gdb) r
Starting program: /home/pi/ASMRASPI/hello_world_asm/helloworld

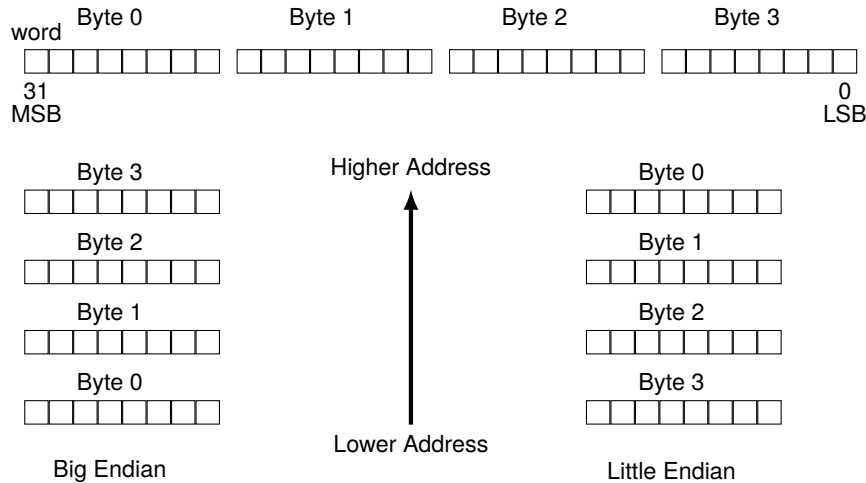
Breakpoint 1, _start () at HelloWorld.s:14
14  _start: mov R0, #1 @ 1 = StdOut
(gdb) s
15      ldr R1, =helloworld @ string to print
(gdb) info registers
r0             0x1             1
r1             0x0             0
r2             0x0             0
r3             0x0             0
r4             0x0             0
r5             0x0             0
r6             0x0             0
r7             0x0             0
r8             0x0             0
r9             0x0             0
r10            0x0             0
r11            0x0             0
r12            0x0             0
sp             0x7efff620      0x7efff620
lr             0x0             0
pc             0x10078         0x10078 <_start+4>
cpsr           0x10           16
fpscr          0x0             0
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint       keep y   0x00010074 HelloWorld.s:14
breakpoint already hit 1 time
(gdb) x /4ubft _start
0x10074 <_start>:  00000001  00000000  10100000  11100011
(gdb) x /4ubfi _start
0x10074 <_start>:  mov    r0, #1
=> 0x10078 <_start+4>:  ldr    r1, [pc, #20] ; 0x10094 <_start+32>
0x1007c <_start+8>:  mov    r2, #13
0x10080 <_start+12>: mov    r7, #4
(gdb) x /4ubfx _start
0x10074 <_start>:  0x01  0x00  0xa0  0xe3
(gdb) x /4ubfd _start
0x10074 <_start>:  1  0  -96  -29

```

ARM family	ARM architecture
ARM7	ARM v4
ARM9	ARM v5
ARM11	ARM v6
Cortex-A	ARM v7-A
Cortex-R	ARM v7-R
Cortex-M	ARM v7-M



Big Endian vs Little Endian



Instructions et accès mémoire :

ldr	Load Word
ldrh	Load unsigned Half Word
ldrsh	Load signed Half Word
ldrb	Load unsigned Byte
ldrsh	Load signed Bytes
str	Store Word
strh	Store unsigned Half Word
strsh	Store signed Half Word
strb	Store unsigned Byte
strsb	Store signed Byte

Number	Alias	Purpose
R0	–	General purpose
R1	–	General purpose
R2	–	General purpose
R3	–	General purpose
R4	–	General purpose
R5	–	General purpose
R6	–	General purpose
R7	–	Holds Syscall Number
R8	–	General purpose
R9	–	General purpose
R10	–	General purpose
R11	FP	Frame Pointer
Special Purpose Registers		
R12	IP	Intra Procedural Call
R13	SP	Stack Pointer
R14	LR	Link Register
R15	PC	Program Counter
CPSR	–	Current Program Status Register

31	30	29	28	27	-	24	-	19	16	-	9	8	7	6	5	4	-	0
N	Z	C	V	Q		J		GE			E	A	I	F	T	M		

- **Negative** : N vaut 1 si la valeur est négative, et 0 si la valeur est positive ;
- **Zero** : Z vaut 1 si le résultat est zéro (par exemple lors d'une comparaison), et 1 si le résultat n'est pas zéro ;
- **Carry** : indique pour
 - ◇ une opération d'addition s'il y a un bit de retenu comme résultat du calcul, c-à-d un dépassement, «*overflow*», de capacité ;
 - ◇ une opération de soustraction s'il y a un bit de retenu, c-à-d un dépassement, «*underflow*», de capacité ;
 - ◇ une opération de décalage, le bit qui a été décalé vers l'extérieur ;
- **Overflow** : pour l'addition et la soustraction, indique un «*overflow*».
Pour certaines autres instructions, ce bit peut être utilisé pour indiquer une erreur.
- les bits liés aux **interruptions** :
 - ◇ **I** : lorsqu'il est à 1 : désactive les IRQ, «*interrupt request*» ;
 - ◇ **F** : lorsqu'il est à 1 : désactive les FIQ, «*Fast interrupt request*» (par exemple : traiter les paquets réseau, les mouvements de la souris) ;
 - ◇ **A** : lorsqu'il est à 1 : désactive les abandons ;
- les bits liés aux **instructions** :
 - ◇ **Thumb** : lorsqu'il est à 1 : indique des instructions compactes sur 16 bits ;
 - ◇ **Jazelle** : lorsqu'il est à 1 : mode obsolète pour l'exécution directe de bytecode Java ;
- les **autres** bits :
 - ◇ **Q** : lorsqu'il est à 1 : indique un «*underflow*» ;
 - ◇ **GE** : contrôle le «*Greater than Equal*» dans le traitement des données SIMD ;
 - ◇ **E** : contrôle l'«*endianness*» pour le traitement des données.
- **M** : indique si le processeur est en mode «*user*» ou «*supervisor*».

On consulte le registre CPSR pour déterminer la condition :

```
B{condition} label
```

Condition Code	Meaning (for cmp or subs)	Status of Flags
EQ	Equal	Z==1
NE	Not Equal	Z==0
GT	Signed Greater Than	(Z==0) && (N==V)
LT	Signed Less Than	N!=V
GE	Signed Greater Than or Equal	N==V
LE	Signed Less Than or Equal	(Z==1) (N!=V)
CS or HS	Unsigned Higher or Same (or Carry Set)	C==1
CC or LO	Unsigned Lower (or Carry Clear)	C==0
MI	Negative (or Minus)	N==1
PL	Positive (or Plus)	N==0
AL	Always executed	—
NV	Never executed	—
VS	Signed Overflow	V==1
VC	No signed Overflow	V==0
HI	Unsigned Higher	(C==1) && (Z==0)
LS	Unsigned Lower or same	(C==0) (Z==0)

Format d'une instruction

	MNEMONIC{S}{condition} {Rd}, Operand1, Operand2
MNEMONIC	Short name (mnemonic) of the instruction
{S}	An optional suffix. If S is specified the condition flags are updated on the result of the operation
{condition}	Condition that is needed to be met in order for the instruction to be executed according to bit(s) of the CPSR
{Rd}	Register (destination) for storing the result of the instruction
Operand1	First operand: either a register or an immediate value
Operand2	Second (flexible) operand: can be an immediate value (number) or a register with an optional shift

La seconde opérande est flexible, elle peut être sous différentes formes : une valeur immédiate, un registre ou un registre avec un décalage :

#123	Immediate value (with limited set of values)
Rx	Register x (like R1, R2, R3 ...)
Rx, ASR n	Register x with arithmetic shift right by n bits (1 <= n <= 32)
Rx, LSL n	Register x with logical shift left by n bits (0 <= n <= 31)
Rx, LSR n	Register x with logical shift right by n bits (1 <= n <= 32)
Rx, ROR n	Register x with rotate right by n bits (1 <= n <= 31)
Rx, RRX	Register x with rotate right by one bit, with extend

Exemples d'instructions :

ADD R0, R1, R2	Adds contents of R1 (Operand1) and R2 (Operand2 in a form of register) and stores the result into R0 (Rd)
ADD R0, R1, #2	Adds contents of R1 (Operand1) and the value 2 (Operand2 in a form of an immediate value) and stores the result into R0 (Rd)
MOVLE R0, #5	Moves number 5 (Operand2, because the compiler treats it as MOVLE R0, R0, #5) to R0 (Rd) ONLY if the condition LE (Less Than or Equal) is satisfied $\implies$ CPSR[N], <i>negative</i> , bit at 1 (result of a previous instruction CMP)
MOV R0, R1, LSL #1	Moves the contents of R1 (Operand2 in a form of register with logical shift left) shifted left by one bit to R0 (Rd). So if R1 had value 2, it gets shifted left by one bit and becomes 4. 4 is then moved to R0.

Instruction	Description	Instruction	Description
MOV	Move data	EOR	Bitwise XOR
MVN	Move and negate	LDR	Load
ADD	Addition	STR	Store
SUB	Subtraction	LDM	Load Multiple
MUL	Multiplication	STM	Store Multiple
LSL	Logical Shift Left	PUSH	Push on Stack
LSR	Logical Shift Right	POP	Pop off Stack
ASR	Arithmetic Shift Right	B	Branch
ROR	Rotate Right	BL	Branch with Link
CMP	Compare	BX	Branch and eXchange
AND	Bitwise AND	BLX	Branch with Link and eXchange
ORR	Bitwise OR	SWI/SVC	System Call

la valeur à l'adresse indiquée dans Rb
est chargée dans Ra

LDR Ra , [Rb]

LDR R2, [R0] @ [R0] - origin address is the value found in R0.
STR R2, [R1] @ [R1] - destination address is the value found in R1.

STR Ra , [Rb]

la valeur de Ra est stockée
à l'adresse indiquée dans Rb

Exemple de code

```
.data      /* the .data section is dynamically created and its addresses is not easily known */
var1: .word 3 /* variable 1 in memory */
var2: .word 4 /* variable 2 in memory */

.text      /* start of the text (code) section */
.global _start

_start:
    ldr r0, adr_var1 @ load the memory address of var1 via label adr_var1 into R0
    ldr r1, adr_var2 @ load the memory address of var2 via label adr_var2 into R1
    ldr r2, [r0]      @ load the value (0x03) at memory address found in R0 to register R2
    str r2, [r1]      @ store the value found in R2 (0x03) to the memory address found in R1
    bkpt

adr_var1: .word var1 /* address to var1 stored here */
adr_var2: .word var2 /* address to var2 stored here */
```

Une instruction Load/Store peut utiliser un «offset» :

- ▷ avec une valeur immédiate ;
- ▷ la valeur d'un registre (comme sur l'exemple plus haut) ;
- ▷ la valeur d'un registre avec un décalage ;

```

xterm
$ arm-none-eabi-as exemple.S
$ ll
total 16K
drwxrwxr-x  2 pef pef 4.0K Nov  6 20:18 ./
drwxr-x--- 85 pef pef 4.0K Nov  6 20:18 ../
-rw-rw-r--  1 pef pef  772 Nov  6 20:18 a.out
-rw-rw-r--  1 pef pef  741 Nov  6 20:17 exemple.S
$ arm-none-eabi-objdump -d a.out

a.out:      file format elf32-littlearm

Disassembly of section .text:

00000000 <_start>:
  0:  e59f000c      ldr     r0, [pc, #12]    @ 14 <adr_var1>
  4:  e59f100c      ldr     r1, [pc, #12]    @ 18 <adr_var2>
  8:  e5902000      ldr     r2, [r0]
 c:  e5812000      str     r2, [r1]
10:  e1200070      bkpt    0x0000

00000014 <adr_var1>:
14:  00000000      .word   0x00000000

00000018 <adr_var2>:
18:  00000004      .word   0x00000004

```

adressage relatif par rapport au registre pc

Pour calculer la valeur 12 de l'accès par offset de l'instruction à l'adresse 0, on sait que :

- ▷ lors de l'exécution de l'instruction à l'adresse 0, le pc est déjà, deux instructions en avant ;
- ▷ chaque instruction est sur 32bits ou 4 octets ;
- ▷ la donnée «*adr_var1*» est à l'adresse 20, «14 en hexa», soit un décalage de $20 - (2 * 4) = 12$ par rapport à la valeur courante du pc

Pour l'instruction à l'adresse 4, c'est la même chose avec une donnée «*adr_var2*» en adresse 24, «18 en hexa».

Utilisation d'un registre et d'un offset avec mise à jour du registre

```

.data
var1: .word 3
var2: .word 4

.text
.global _start

_start:
    ldr r0, adr_var1 @ load the memory address of var1 via label adr_var1 into R0
    ldr r1, adr_var2 @ load the memory address of var2 via label adr_var2 into R1
    ldr r2, [r0]      @ load the value (0x03) at memory address found in R0 to register R2
    str r2, [r1, #2] @ address mode: offset. Store the value found in R2 (0x03) to the memory address found in R1 plus 2. Base register (R1) unmodified.
    str r2, [r1, #4]! @ address mode: pre-indexed. Store the value found in R2 (0x03) to the memory address found in R1 plus 4. Base register (R1) modified: R1 = R1+4
    ldr r3, [r1], #4 @ address mode: post-indexed. Load the value at memory address found in R1 to register R3. Base register (R1) modified: R1 = R1+4
    bkpt

adr_var1: .word var1
adr_var2: .word var2

```

Différents accès

On peut utiliser la valeur d'un registre pour servir d'offset :

```

str r2, [r1, r2] @ address mode: offset. Store the value found in R2 (0x03) to the memory address found in R1 with the offset R2 (0x03). Base register unmodified.

```

On peut modifier l'adresse de base **avant** de l'utiliser :

```

str r2, [r1, r2]! @ address mode: pre-indexed. Store value found in R2 (0x03) to the memory address found in R1 with the offset R2 (0x03). Base register modified: R1 = R1+R2.

```

On peut modifier l'adresse de base **après** de l'utiliser :

```

ldr r3, [r1], r2 @ address mode: post-indexed. Load value at memory address found in R1 to register R3. Then modify base register: R1 = R1+R2.

```

Les registres spéciaux

xPSR	Program Status Register	ApplicationPSR⇒ N egative Z ero C arry o V erflow
		InterruptPSR⇒bits[5:0] numéro de l'exception context courant 0:Thread 2:NMI 3:HardFault
		ExecutionPSR⇒bit [24] T-bit ou mode thumb
MSP	Main Stack Pointer	pointeur de pile par défaut utilisé dans le mode Thread et dans les handler d'exceptions
PSP	Process Stack Pointer	pointeur de pile utilisé en mode normal ou en mode Thread à l'aide du registre CONTROL
CONTROL	bit[0]: niveau de privilège 0⇒privileged 1⇒unprivileged) bit[1]: choix de la pile active en mode Thread 0⇒MSP 1⇒PSP	
PRIMASK	masque toutes les exceptions avec priorité configurable s'il vaut 1	

Les opérations de manipulations

mrs	Move from Special Register to Register
msr	Move from Register to Special Register
stmia	Store Multiple Increment After

Exemple :

```
mrs r0, psp ⇒ mets la valeur de PSP dans le registre r0
stmia r0!, {r4, r5, r6, r7}⇒ store les valeurs de registres à l'adresse
donnée par r0 et décrémente r0
```

Sauvegarde des registres

stmia	Store Multiple Increment After	permet de sauvegarder les différents registres indiqués
	exemple	stmia

Exemple:

```
mrs r0, psp
subs r0, #16
stmia r0!, {r4, r5, r6, r7}
```

- ▷ on mets l'adresse de la pile dans r0 ;
- ▷ on fait de la place dans la pile pour 4 registres de 32bits ;
- ▷ on sauvegarde les 4 registres dans la pile ;
- ▷ le «!» indique que r0 va être automatiquement incrémenté.

⇒ il faut le décrémenter de nouveau pour le placer en sommet de pile.

0x2000FFD0

0x2000FFCC

0x2000FFC8

0x2000FFC4

SP →

R7

R6

R5

R4

stack grows down
address decreases