MI11 — Architecture ARM

Stéphane Bonnet

Université de Technologie de Compiègne

bonnetst@utc.fr

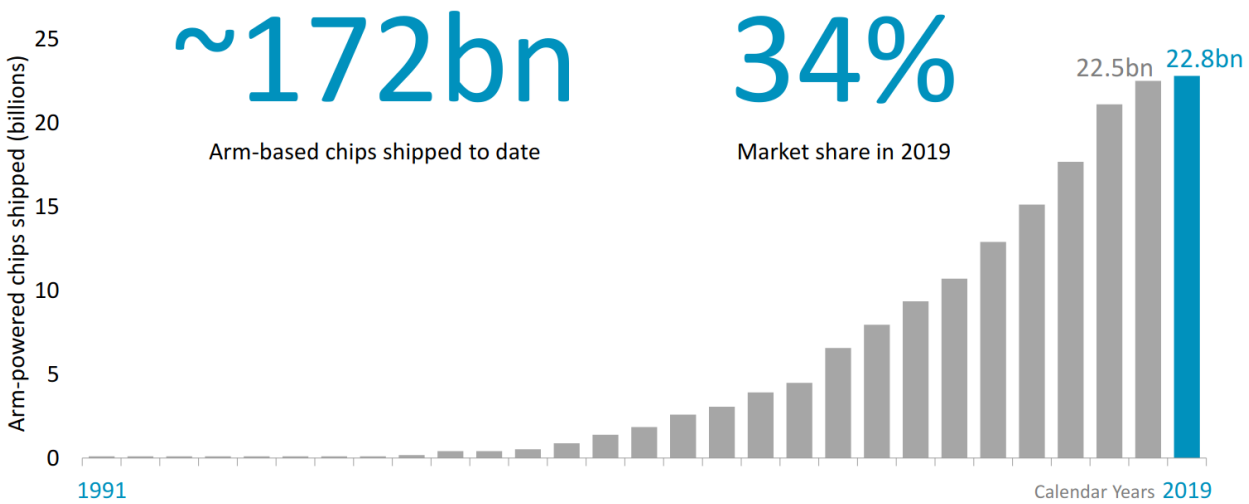
Printemps 2022

- ARM Holdings Plc. : entreprise britannique spécialisée dans la conception de microprocesseurs 32 et 64 bits de type RISC (*Reduced Instruction Set Computer*).
  - 1978 : Acorn Computers fabrique des ordinateurs 8-bits pour le marché britannique
  - Années 80 : Acorn crée avec VLSI Technology un processeur RISC 32-bits pour sa future gamme professionnelle, l'Acorn Risc Machine (Architecture ARM2).
  - 1990 : Acorn se sépare de sa branche R&D qui devient Advanced Risc Machines, Ltd. Architecture ARM6, utilisée par Apple dans son PDA « Newton ».
  - 1998 : devient simplement ARM Holdings Plc.
  - 2016 : rachat par Softbank Group Corp.
  - 2020 : proposition de rachat par Nvidia Corp (abandonnée en février 2022).

arm

- Par extension, ARM est le nom générique de l'architecture des processeurs de ARM Holdings Plc.
- ARM ne vend que des licences : entreprise de semi-conducteurs « fabless »
  - Licences « core » : utilisation directe du design ARM dans les produits du client
  - Licences « architecture » : le client conçoit lui-même un cœur conforme à l'architecture ARM
- 830+ clients, dont Allwinner, AMD, Apple, Broadcom, Freescale (NXP), Huawei, IBM, Intel, Marvell, Nvidia, Qualcomm, Samsung, STMicroelectronics, Texas Instruments, etc.

- ARM se spécialise dans les cœurs faible consommation (rapport Watts / MIPS) plutôt que la puissance brute (cf. Intel).
- Adapté aux systèmes mobiles sur batterie, mais aussi aux systèmes embarqués en général
- Applications du Smart Phone (virtuellement tous) au thermostat connecté en passant par les automates industriels, les systèmes d'aide à la conduite ou les équipements réseau : en 2016, près de 80% des processeurs 32-bits
- De plus en plus d'ARM dans les serveurs (Amazon AWS Graviton) et dans les ordinateurs personnels (Apple M1).



Source : Softbank Group Corp.

La plupart des cœurs ARM se trouvent dans des « System-on-Chip » (*SoC*).

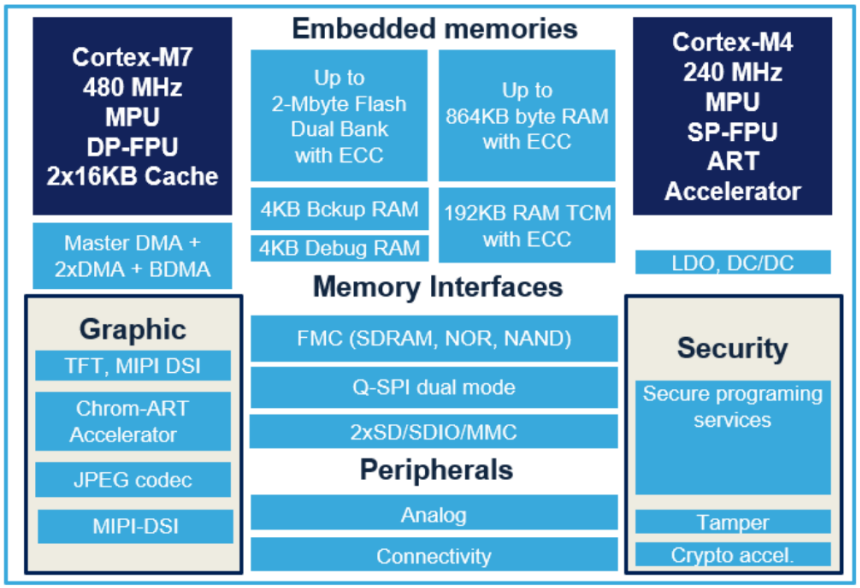
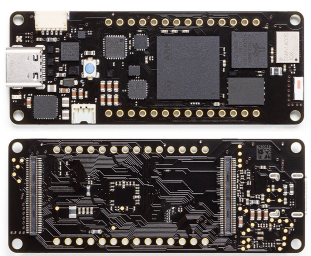


Schéma-bloc d'un *SoC* STM32H7 basé sur deux cœur ARM Cortex-M7 et Cortex-M4.

System-on-Chip

Système complet sur une seule puce comprenant un ou plusieurs CPU, GPU, DSP, mémoire (mais pas toujours), périphériques (par exemple Ethernet, HDMI, SATA)



SoC est un terme vague. En général on parle soit de microcontrôleur, soit de processeur d'application.

Microcontrôleur

SoC incluant également les mémoires de travail (SRAM) et de programme (ROM, Flash). Ne nécessite aucun composant externe hormis les alimentations et parfois les horloges. Utilisé dans les systèmes embarqués profonds.

Processeur d'application

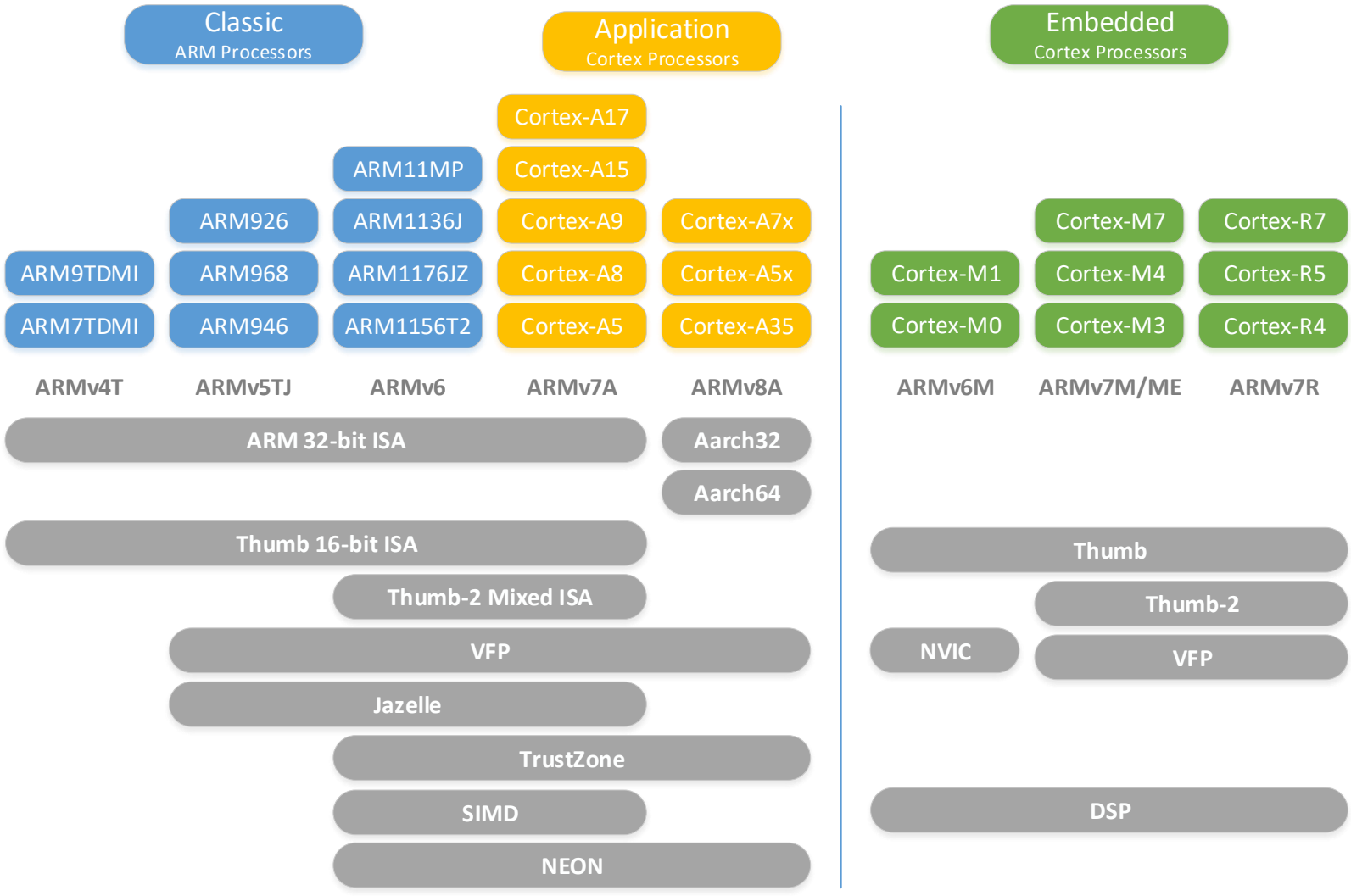
Nécessite au moins une mémoire de programme externe (ROM, Flash). Mémoire de travail souvent externe (type DDR SDRAM). Propose toujours un bus d'extension générique (bus données / adresses / contrôle classique, PCIe,...)

Les frontières entre les deux sont parfois très floues...

Architecture

Modèle de programmation + jeu d'instructions

- Exclut les spécificités d'un *SoC* particulier. Les éléments autres que le CPU dans les *SoC* (GPU, périphériques,...) sont spécifiques à chaque fabricant.
- Peut avoir plusieurs implémentations physiques (*architecture matérielle*) très différentes.
- ARM : architecture RISC (*Reduced Instruction Set Computer*)
  - Instructions de taille fixe
  - Instructions générales manipulent des constantes et des registres
  - Accès mémoire par instructions type LOAD et STORE spécifiques



Évolution de l'architecture et de la gamme ARM 32 bits

Depuis la version 6 de l'architecture pour les microcontrôleurs et la version 7 pour les processeurs d'application, gamme séparée en plusieurs architectures (« Profils ») :

Profil A Il s'agit du profil *Application*. Architecture ARMv7-A (*Cortex-A*) et suivantes.

Profil M Il s'agit du profil *Microcontroller*. Architecture ARMv6-M (*Cortex-M*) et suivantes.

Pas de différentiation pour les architectures $\leq$ ARMv5, qui correspondent au profil Application actuel.

Profil *Cortex-A*

- Architecture ARM historique
- Jeux d'instruction ARM et Thumb
- Gestion de mémoire virtuelle complète (unité MMU, *Memory Management Unit*) pour les OS complexes type Linux
- Mémoires externes
- Cœurs 32 ou 64 bits (à partir de ARMv8-A)

Profil *Cortex-M*

- Jeu d'instruction Thumb uniquement
- Pas de gestion de mémoire virtuelle, éventuellement unité de protection de la mémoire (MPU, *Memory Protection Unit*)
- Mémoires intégrées
- Cœurs 32 bits

C'est ce profil qui est considéré dans la suite du cours.

Modèle de programmation

Vue abstraite du fonctionnement d'un calculateur du point de vue d'un programme (par opposition à l'architecture physique)

Les modèles de programmation de l'architecture ARM dépendent :

- du profil (A ou M)
- du modèle à l'intérieur du profil

Dans la suite, le modèle illustré est conforme à l'architecture ARMv7E-M comme implémentée sur les microcontrôleurs Cortex-M7.

Tailles de données ARM

byte	Un octet (8 bits)
halfword	Un mot de 16 bits
word	Un mot de 32 bits
doubleword	Un mot de 64 bits

Jeux d'instructions

La plupart des cœurs ARM implémentent deux jeux d'instruction :

- jeu d'instructions 32 bits ARM
- jeu d'instructions 16 bits Thumb

En plus pour certains cœurs :

- jeu d'instructions mixte 16/32 bits Thumb-2
- exécution bytecode Java Jazelle-DBX

Alignement des données en mémoire

- Un accès est dit aligné si l'adresse de la donnée est un multiple de sa taille
- Pour les architectures ARMv7E-M, les accès réalisés par certaines instructions peuvent être non alignés. Tous les autres accès mémoire doivent être alignés. En pratique, il vaut mieux aligner **tous les accès**.
- Un accès non aligné non autorisé provoque une exception de type *UsageFault*.

Ordre de stockage des mots

- L'architecture ARMv7E-M n'impose pas d'ordre spécifique de stockage des octets dans les mots.
- Configuré par les fabricants de *SoC* au moyen de bits de configuration physiques
- En général, les fabricants de *SoC* choisissent l'ordre *little endian*

- Deux modes de fonctionnement :

  Mode *Thread* Mode d'exécution normal de l'application. Ce mode est actif au démarrage du cœur.

  Mode *Handler* Mode d'exécution actif pendant les exceptions. Le processeur revient au mode *Thread* à la fin du traitement de l'exception.

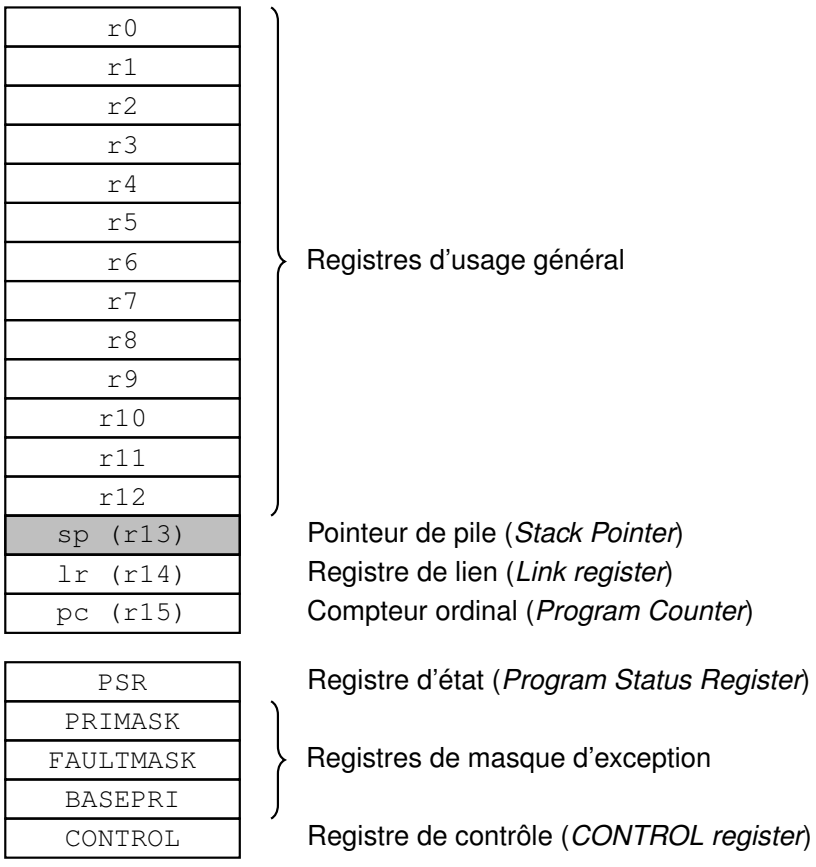
- Deux niveaux de privilège :

  Non privilégié Le logiciel n'a accès qu'à certaines ressources et périphériques. Certaines instructions sont interdites.

  Privilégié Le logiciel a accès sans restrictions à toutes les ressources.

- En mode *Thread*, le registre `CONTROL` permet de choisir le niveau de privilège. L'exécution en mode *Handler* est toujours privilégiée.

Registres ARM



- 12 registres généraux 32 bits `r0-r12`
- `r7` = `fp` (pointeur de cadre de pile, *frame pointer*) si nécessaire pour *gcc*
- `r13` = `sp` (pointeur de pile)
- `r14` = `lr` (adresse de retour dans les appels)
- `r15` = `pc` (adresse de la prochaine instruction)

Pointeurs de pile 1/2

La pile est gérée en mode *full descending* :

- Le pointeur de pile contient l'adresse du dernier élément empilé
- Pour empiler une donnée, le processeur **décrémente** le pointeur de pile puis la stocke à cette adresse.

Pile principale et pile processus

Deux piles existent en parallèle :

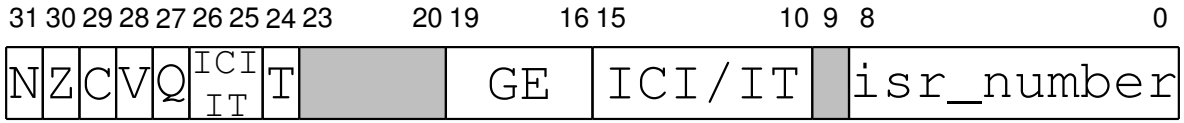
- *pile principale* ou *main stack* (registre `msp`)
- *pile processus* ou *process stack* (registre `psp`)
- Le registre `CONTROL` détermine lequel des deux est accessible par `sp`.

Pointeurs de pile 2/2

Piles et modes de fonctionnement

- En mode **Thread**, la pile utilisée dépend de la valeur de `CONTROL`.
- En mode **Handler**, la pile utilisée est toujours la pile principale.

Registre d'état PSR (*Program Status Register*)



Codes conditions

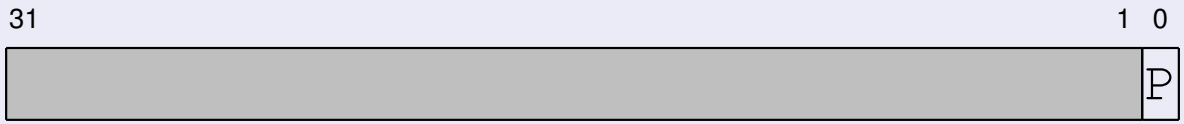
- N : résultat Négatif
- Z : résultat nul (Zero)
- C : bit de retenue (Carry)
- V : débordement (oVerflow)
- Q : débordement ou saturation SIMD

isr_number

Indique le numéro d'exception en cours. 0 en mode *Thread*.

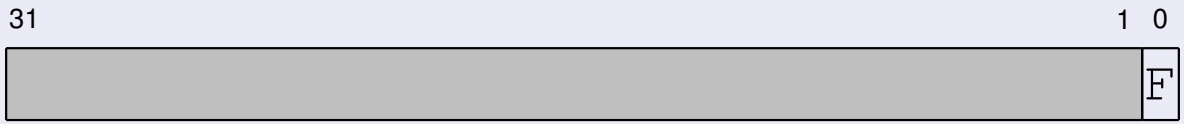
Les registres PRIMASK, FAULTMASK et BASEPRI permettent d'activer ou de désactiver les exceptions.

Registre *PRIMASK*



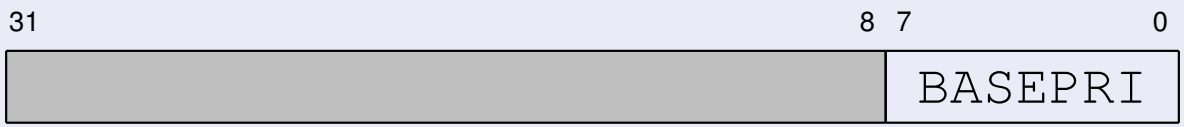
P=1 ⇒ interdit toute exception à priorité configurable (IRQ)

Registre *FAULTMASK*



F=1 ⇒ interdit toute exception sauf l'interruption non masquable *NMI*

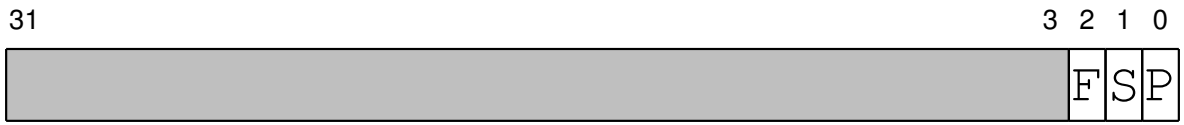
Registre *BASEPRI*



BASEPRI : valeur de priorité maximale de prise en compte des exceptions.

Le registre `CONTROL` permet de définir :

- L'utilisation de l'unité de calcul virgule flottante (*FPU, Floating-Point Unit*) en fonction du bit `FPCA` (*Inactive* si 0, *Active* si 1)
- La pile utilisée en fonction du bit `SPSEL` (*Main stack* si 0, *Process Stack* si 1)
- Le niveau de privilège en fonction du bit `PRIV` (*Privilégié* si 0, *Non privilégié* si 1)



FPU

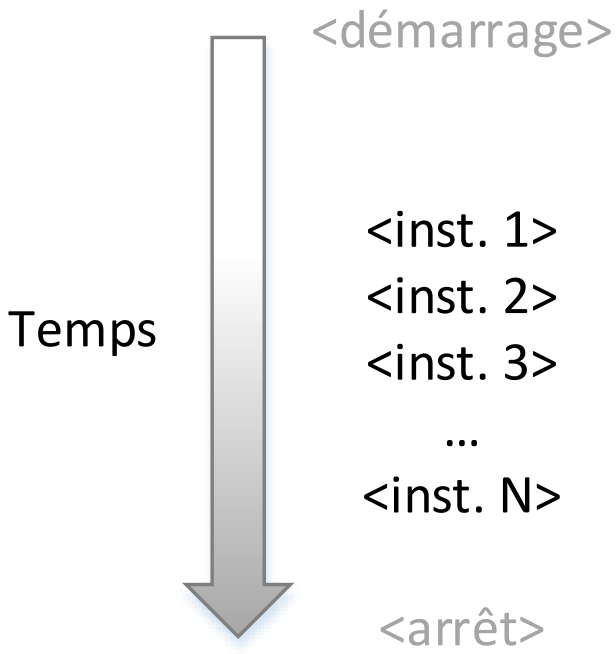
L'utilisation d'instructions en virgule flottante positionne automatiquement le bit `FPCA`. Ce bit est utilisé pour décider de la sauvegarde automatique du contexte FPU lors des exceptions.

Synchronisation

Les changements d'identité du pointeur de pile dans `CONTROL` doivent être suivis d'une instruction de synchronisation `ISB` pour garantir que le nouveau pointeur est bien utilisé.

Flot de contrôle

- Un processeur exécute une séquence d'instructions de son démarrage à son extinction, une par une
- Cette séquence est le flot de contrôle du processeur



Flot de contrôle *physique*
(point de vue du processeur)

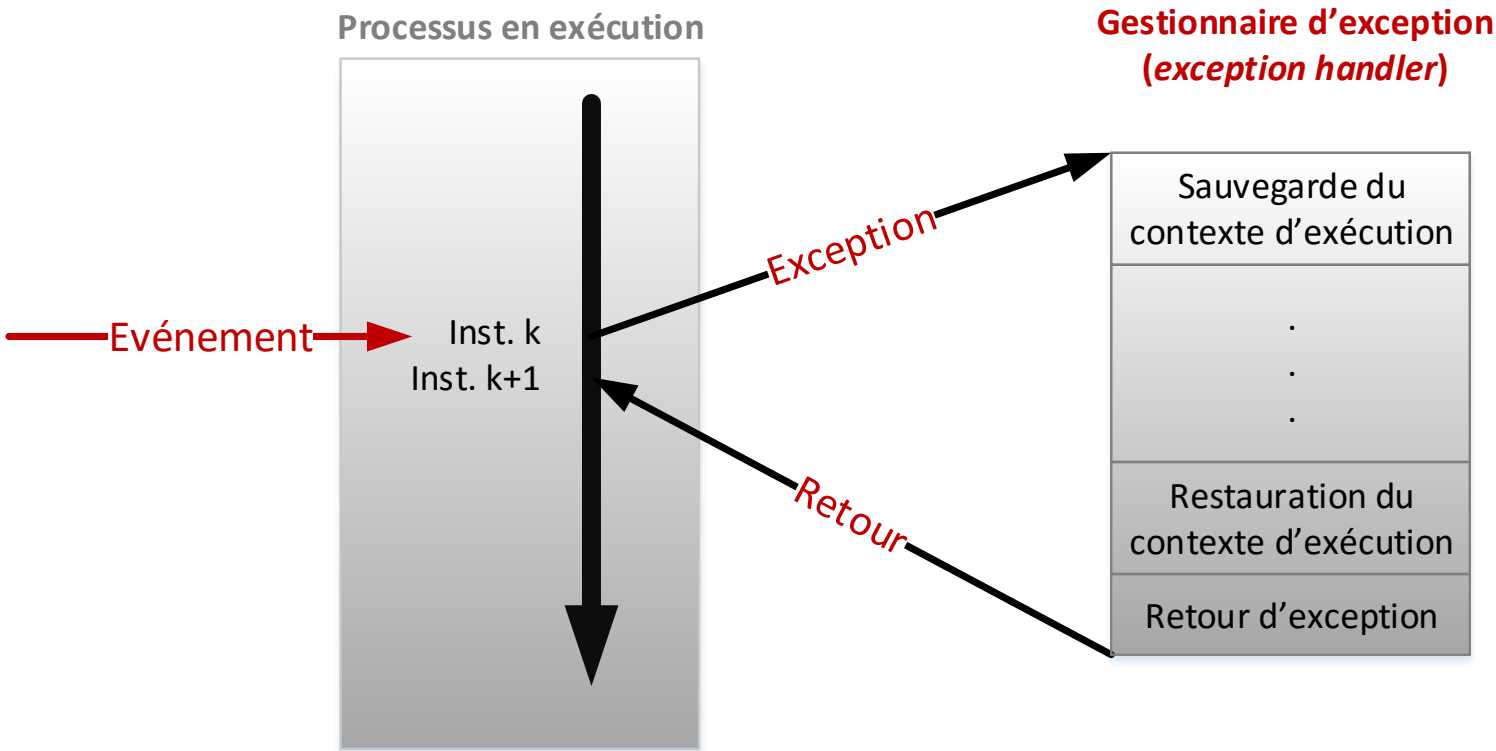
Modification du flot de contrôle

- Règle générale : modifier la valeur du compteur de programme (r15, pc) — Le compteur de programme contient l'adresse mémoire de la prochaine instruction à exécuter.
  - Par programmation :
    - Sauts (conditionnels et inconditionnels)
    - Appels / retours de sous-programmes

  Réaction à des changements de l'état du programme
- Insuffisant pour un système complet
  Doit réagir à des changements de l'état du système
  - Un périphérique signale un événement externe
  - Une instruction conduit à une division par 0
  - Une temporisation système expire. . .
- Nécessité d'un mécanisme pour gérer les flots de contrôle exceptionnels

Exceptions

Une exception est le transfert de contrôle à une adresse particulière en réponse à un *événement* (p. ex. un changement de l'état du processeur)



Causes d'exception

Événement interne

- Exception due au processeur lui-même
- Réaction à des erreurs (division par 0, accès à une zone mémoire interdite . . .)
- Provoquée explicitement dans le programme
- Exceptions synchrones

Événement externe

- Exception due à un périphérique
- Un périphérique active un signal spécifique, le processeur reçoit ce signal et traite un gestionnaire associé à ce signal
- Interruptions asynchrones

Types d'exceptions

Exception	**Signification**	**Type**
Reset	Déclenchée à la mise sous tension	Asynchrone
NMI	Interruption externe non masquable	Asynchrone
HardFault	Erreur lors du traitement d'une exception	
MemManage	Faute de protection mémoire	Synchrone
BusFault	Faute d'accès à la mémoire	Synchrone
UsageFault	Instruction illégale / incorrecte	Synchrone
SVCall	Appel superviseur	Synchrone
PendSV	Requête d'appel superviseur	Asynchrone
SysTick	Expiration timer système	Asynchrone
IRQ	Interruption externe (jusqu'à 239)	Asynchrone

Vecteurs d'exception

0x00000000	msp initial
0x00000004	Reset
0x00000008	NMI
0x0000000c	HardFault
0x00000010	MemManage
0x00000014	BusFault
0x00000018	UsageFault
0x0000001c	
0x00000020	
0x00000024	
0x00000028	
0x0000002c	SVCall
0x00000030	
0x00000034	
0x00000038	PendSV
0x0000003c	Systick
0x00000040	IRQ0
0x00000044	IRQ1
0x00000048	IRQ2
	...
0x000003fc	IRQ239

- À chaque exception est associé un gestionnaire d'exception qui sera appelé par le processeur lorsqu'elle survient.
- Une table en mémoire permet de donner les adresses de tous les gestionnaires d'exception : la *table des vecteurs d'exception*.
- Le bit de poids faible de chaque adresse de gestionnaire doit valoir 1. Il sera ignoré par le processeur.

Où est la table des vecteurs ?

- L'adresse de cette table est par défaut 0. Elle peut être déplacée par le fabricant du *SoC* (par exemple `0x08000000` sur STM32 pour la positionner dans la mémoire Flash) ou par logiciel.
- Elle doit être fournie soit par le programme d'application, soit par l'OS s'il y en a un.

Déroulement d'une exception

Quand une exception survient, le cœur réalise les opérations suivantes :

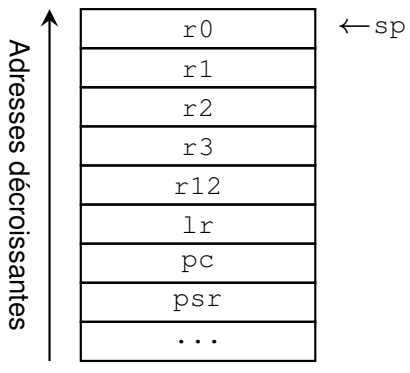
- Sauvegarde partielle du contexte d'exécution courant sur la *pile courante*
- Lecture de l'adresse du gestionnaire d'exception dans la table des vecteurs
- Chargement dans `lr` d'un code de retour d'exception `EXC_RETURN` approprié
- Chargement de `pc` avec l'adresse du gestionnaire d'exception

À la fin du traitement, le gestionnaire d'exception charge dans `pc` la valeur de `EXC_RETURN`, permettant au processeur de détecter la fin du gestionnaire. Le cœur réalise les opérations suivantes :

- Restauration du contexte à partir de la pile (*main stack* ou *process stack* en fonction de `EXC_RETURN`)
- Poursuite de l'exécution au point où l'exception s'était produite.

Structure de la pile d'exception

La pile d'exception a la structure suivante après entrée dans le gestionnaire (on suppose que l'état du FPU n'a pas été sauvegardé) :



- Les registres sauvegardés sont les registres volatils spécifiés par la convention d'appel AAPCS (*Arm Architecture Procedure Call Standard*).
- Un gestionnaire d'exception peut donc être simplement une fonction classique : elle sauvegardera tous les autres registres si nécessaire.
- La valeur du pointeur de pile est toujours alignée sur 8 octets.

Remarques

Exception PendSV

L'exception PendSV est déclenchée par logiciel mais ne sera exécutée que quand **toutes** les autres exceptions seront terminées. Elle peut être utilisée dans les OS pour planifier une préemption de tâche.

Interruption logicielle SVC

- Provoquée par l'exécution de l'instruction svc #n, $n \in [0, 255]$
- Utilisée pour provoquer des commutations de niveau de privilège (appels système par exemple) : elle permet à l'application de passer du mode *Thread* non privilégié au mode *Handler* privilégié par programme.

Écriture de gestionnaires d'exception

- En assembleur si besoins spécifiques
- En C (on peut toujours utiliser l'assembleur inline si besoin)

Gestionnaires avec gcc

```
1   void __attribute__((interrupt)) irq_handler(void)
2   {
3         /* Code du gestionnaire d'exception */
4   }
```

Assembleur inline

```
1   void do_nothing_4_times(void)
2   {
3         asm("mov  r0, r0\t\n"
4             "mov  r0, r0\t\n"
5             "mov  r0, r0\t\n"
6             "mov  r0, r0\t\n");
7   }
```

- L'adresse associée au vecteur est simplement l'adresse de la fonction **dont le bit de poids faible a été mis à 1**.
- L'attribut `interrupt` n'est pas strictement nécessaire mais il garantit un alignement correct de la pile.

Jeu d'instructions Thumb-2

- Instructions encodées sur 16 ou 32 bits
- La plupart s'exécutent en un seul cycle
- Toutes les instructions sont conditionnelles
- Architecture LOAD / STORE (*RISC*) : instructions spécifiques d'accès à la mémoire
- Exemple d'instructions arithmétiques

```
sub     r0,r1,#5          @ r0 = r1 + r5
add     r2,r3,r3,lsl #2   @ r2 = r3 + (r3 * 4)
```

- Exemples de branchements

```
b       <etiquette>       @ branchement inconditionnel
beq     <etiquette>       @ branchement si condition "égal" vraie
@ Les branchements sont limités a +/- 32 Mo par rapport à pc
```

- Exemples d'accès à la mémoire

```
ldr     r0,[r1]           @ charger le mot d'adresse r1 dans r0
strb    r2,[r3,r4]        @ charger l'octet de poids faible de r2
                          @ dans l'octet d'adresse r3 + r4
push {r4-r8,lr}           @ stocker les registres r4 à r8 dans la
                          @ pile
```

Jeu d'instructions ARM

- Jeu d'instructions encodées sur 32 bits
- Jeu d'instruction « traditionnel Arm », utilisé par défaut par les profils A
- En pratique, les deux jeux d'instructions convergent avec l'approche *Unified Assembly Language*
- Pas utilisé en MI11.

Exemple de source Assembleur GNU

```
1   .title "Mon programme"
2   .data       @ Section de données initialisées
3   myword:     .word   0x42424242  @ Une donnée 32 bits
4   mybyte:     .byte   0x55        @ Une donnée 8 bits
5   mybyte2:    .byte   0b10010111  @ Une donnée 8 bits en binaire
6   mybyte3:    .byte   -12         @ En décimal
7   mystr:      .asciz  "Hello!"    @ Une chaîne (terminée par 0)
8
9   .text       @ Section de code
10  .thumb                          @ Code ARM
11  .global myfunc                  @ myfunc est un symbole global
12  .func myfunc                    @ myfunc est une fonction (debug)
13  myfunc:                         @ étiquette myfunc
14      push    {fp,lr}
15      mov     fp, sp
16      ldr     r0,=myword          @ valeur de l'étiquette
17      ldr     r0,[r0]
18      cmp     r0, #0              @ Constante 0
19      mov     r0, #0xffffffff     @ Constante 2^32-1
20      pop     {fp,lr}
21      bx      lr
22  .endfunc
23
24  /*
25      Une section en lecture / écriture, non exécutable (aw)
26  */
27  .section mysection, "aw", %progbits
28  .align  8                       @ Alignement, 4 par défaut
29              .ascii "Ma super section"
30  .end
```

Lors de la survenue de l'exception *reset*, le processeur :

- Initialise les registres de contrôle du processeur avec leurs valeurs par défaut : mode Thread, *main stack* active, interruptions désactivées, fautes activées.
- Charge le premier mot de la table des vecteurs (32 bits) dans `sp`
- Charge le second mot de la table des vecteurs (vecteur de reset) dans `pc`. Le premier bit est mis à 0 pour obtenir une adresse paire.
- Commence l'exécution du programme à cette adresse.

Le programme est alors démarré.

Adresse de la table des vecteurs d'exception

- Par défaut, 0.
- Dépend de l'implémentation et de la configuration réalisée par le fabricant du *SoC*. Sur STM32, l'adresse par défaut est `0x8000000` mais est modifiable par l'utilisateur.

Ce cours n'est qu'une introduction aux points essentiels. Pour aller plus loin :

- Guide de l'utilisateur Cortex-M7 : `https://developer.arm.com/documentation/dui0646/a?lang=en`
- Manuel de référence de l'architecture ARMv7-M : `https://developer.arm.com/documentation/ddi0403/latest/`
- Documentations de l'implémentation STMicro (STM32H747) utilisée en TP : `https://www.st.com/en/microcontrollers-microprocessors/stm32h747-757.html`

Il faut tout faire à la main ?

- En pratique, non. ARM fournit un standard logiciel et des bibliothèques qui font une abstraction du matériel de des microcontrôleurs : le CMSIS (*Cortex Microcontroller Software Interface Standard*). Ce standard simplifie le développement et la réutilisation des logiciels.
  (`https://developer.arm.com/tools-and-software/embedded/cmsis`)
- En MI11, oui. Le but est d'apprendre comment les choses fonctionnent et ne pas se cacher derrière une abstraction !