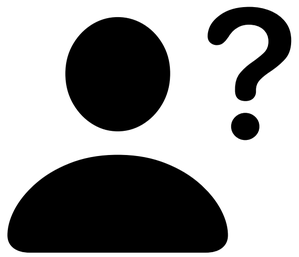
POLYTECH©
DIJON

Architecture interne des ordinateurs

Dernière mise à jour le 25 janvier 2024 par dG

arm

Architecture



C’est quoi un processeur arm ?

Les processeurs ARM (Advanced RISC Machine) sont une famille de processeurs qui reposent sur une architecture RISC (Reduced Instruction Set Computer).

Les architectures ARM représentent une approche différente de la conception d'un système par rapport aux architectures plus connues comme x86 et visent à offrir le meilleur équilibre possible entre coût, consommation d’énergie et performance.

Aujourd'hui, les architectures ARM sont majoritairement utilisées dans le monde des smartphones grâce à leurs systèmes sur puce (SoC) intégrant sur une seule puce : CPU, GPU, NPU, DSP et autres contrôleurs de périphériques.

Un peu d'histoire

1978 — 1985 — 1990 — 2016 — 2020 →

Acorn Computer est une société anglaise créée en 1978 qui crée des micro-ordinateurs. En 1985, elle développe un processeur RISC nommé ARM (Acorn RISC Machine puis Advanced RISC Machine).

La société ARM Limited est créée en 1990 par Acorn, Apple et VLSI Technology. ARM appartient depuis 2016 à SoftBank, un fond d'investissement japonais.

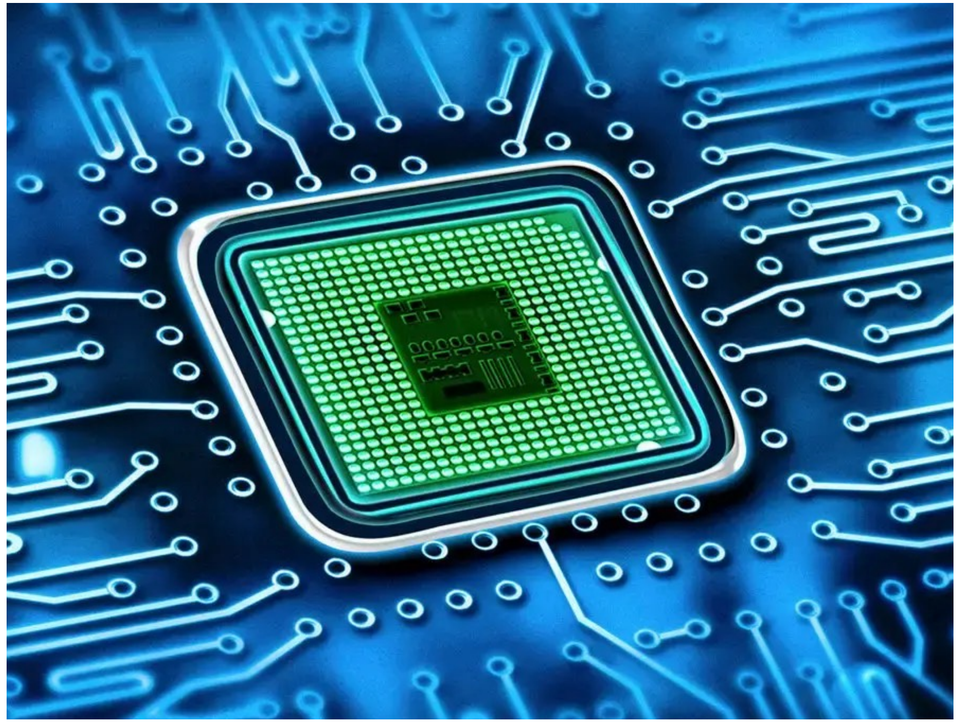
En 2020, NVIDIA a proposé d'acquérir ARM mais les autorités britanniques se sont opposées.

arm AUJOURD'HUI

ARM ne fabrique pas et ne vend pas de micro-processeurs. ARM développe seulement des architectures de micro-processeurs 32/64 bits et les jeux d'instructions associés.

ARM licencie des technologies de processeurs à d'autres sociétés qui vont les customiser et les intégrer dans leurs produits (Ex: Apple, Samsung, ...).

Les architectures ARM sont les plus utilisées au monde avec plus de 200 milliards de processeurs ARM produits depuis 1991 (aujourd'hui' environ 1000 processeurs par seconde, soit plus de 2 milliards par mois).



ARCHITECTURES & PRODUITS

ARM1 (ARMv1) : R&D interne

....

ARM7TDMI (ARMv4) : 1er succés, Nintendo DS, Lego NXT, GPS Garmin, Nokia 611

ARM926EJ-S (ARMv5) : Nintendo Wii

ARM946E-S (ARMv5) : Canon 5D Mark II, Nintendo 3DS

ARM1176JZ(F)-S (ARMv6) : iPhone, Raspberry Pi

Cortex-A7,8,9 (ARMv7a) : Raspberry Pi 2 (A7), IPhone 3GS (A8), Samsung Galaxy S (A8), Samsung Galaxy S II et III (A9)

Cortex-A72 (ARMv8a) : Raspberry Pi 4

Cortex-A76 (ARMv8.2) : Raspberry Pi 5

....

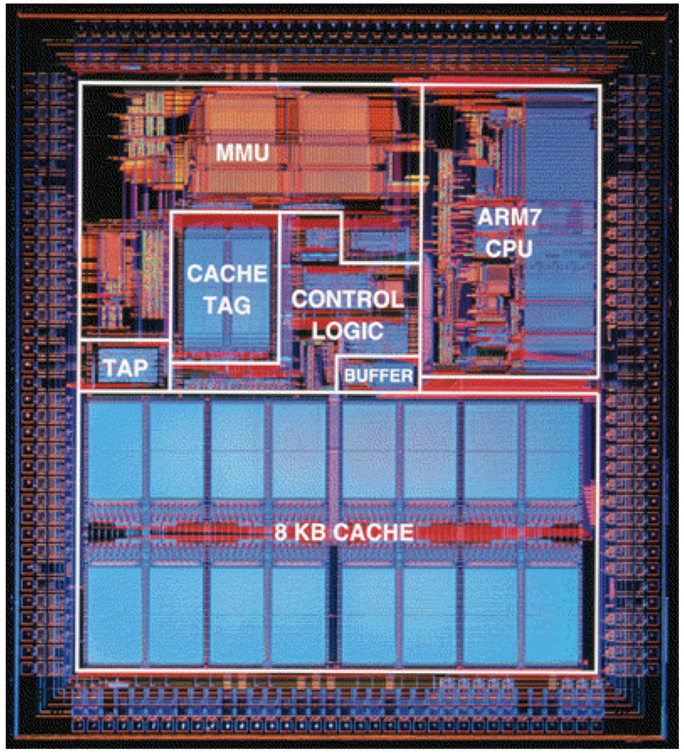
Série Kryo (ARMv8-A) : Snapdragon SOC

Série Mx (ARMv8.5-A) : Apple M1, M2, M3

https://en.wikipedia.org/wiki/List_of_products_using_ARM_processors

Quelle architecture ARM pour le cours ?

ARM7TDMI



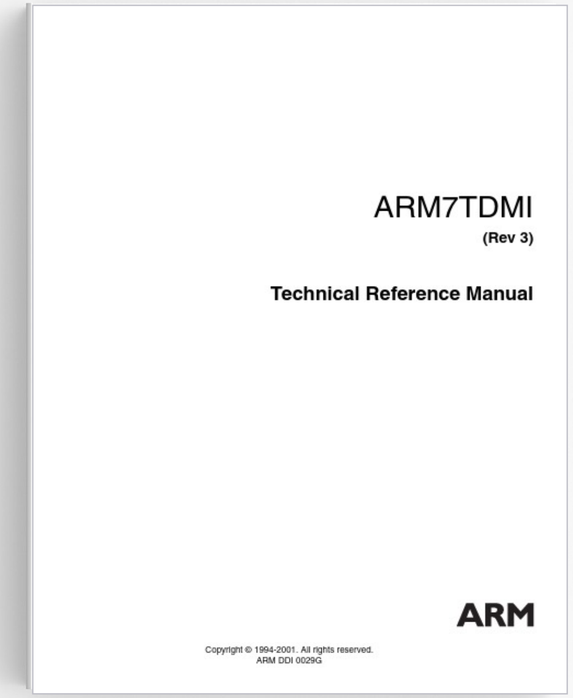
Architecture très populaire

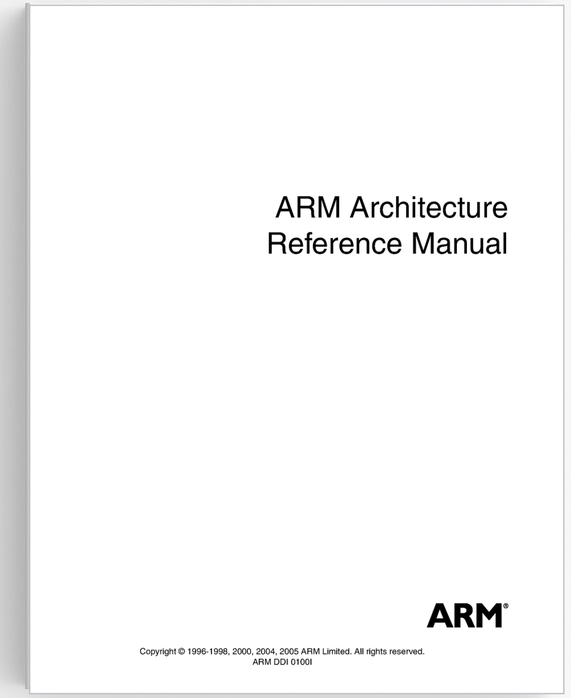
- ➡ Architecture 32/16 bits
- ➡ Jeu d'instruction 32 bits très simple
- ➡ Faible consommation

Que veut dire ARM7TDMI ?

- ➡ 7: 7ème famille de processeurs (ISA ARMv4T)
- ➡ T : Thumb (support d'un second jeu d'instructions 16 bits compact)
- ➡ D : Debug (extension pour la mise au point des programmes)
- ➡ M : Multiplier (Multiplier 32 bits avec résultat sur 64 bits)
- ➡ I : In-Circuit Emulator (extension matérielle pour le debug)

ARM7TDMI : Documentation





Disponibles au téléchargement sur https://github.com/dginhac/esirem-archi

ARM7TDMI : Organisation de la **mémoire**

Taille d'une case mémoire : 1 octet (8 bits)

Largeur du bus de données : 4 octets (32 bits)

4 addresses consécutives pour stocker un mot

Stockage en **Little Endian** : Octet le moins significatif dans la plus petite adresse de la mémoire

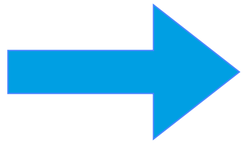
0x1000	0x1001	0X1002	0X1003	0X1004	0X1005	0X1006	0X1007
78	56	34	12	90	37	13	42

Stockage Little endian de 0x12345678 à l'adresse 0x1000
Stockage Little endian de 0x42133790 à l'adresse 0x1004

ARM7TDMI : Instructions et PC

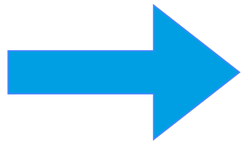
Taille d'une case mémoire : 1 octet (8 bits)

Taille d'une instruction : 4 octets (32 bits)



4 addresses consécutives pour stocker une instruction

	Instructions
0x80	MOV R0, #0x1000
+4	
0x84	LDR R1, [R0]
+4	
0x88	ADD R0, R0, #4
+4	
0x8C	LDR R2, [R0]
+4	
0x90	ADD R3, R1, R2
+4	
0x94	ADD R0, R0, #4
+4	
0x98	STR R3, [R0]



Incrémentation de PC

PC ← PC + 4

PC est le registre "Program Counter" qui stocke l'adresse de la prochaine instruction à exécuter.

ARM7TDMI : Pipeline d’instructions

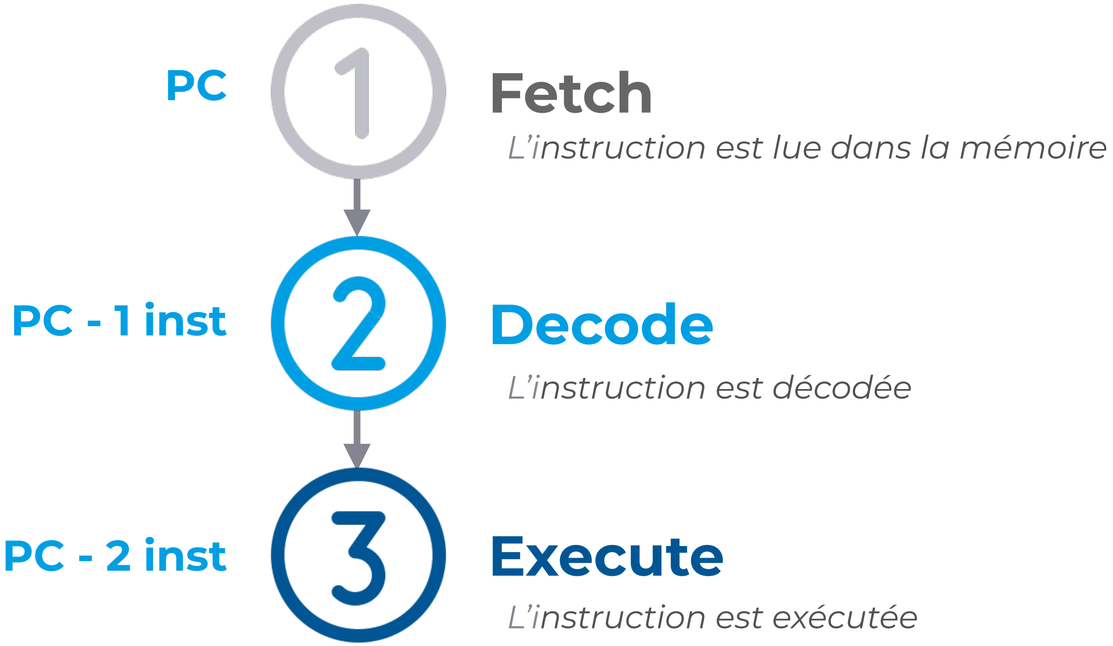
Apparu au début des années 90, les pipelines d’instructions ont pour objectif de séparer l’architecture en plusieurs unités distinctes pouvant être exécutées simultanément et ainsi augmenter la vitesse de traitement.

Modèle directement inspiré des lignes d’assemblage des usines de production.

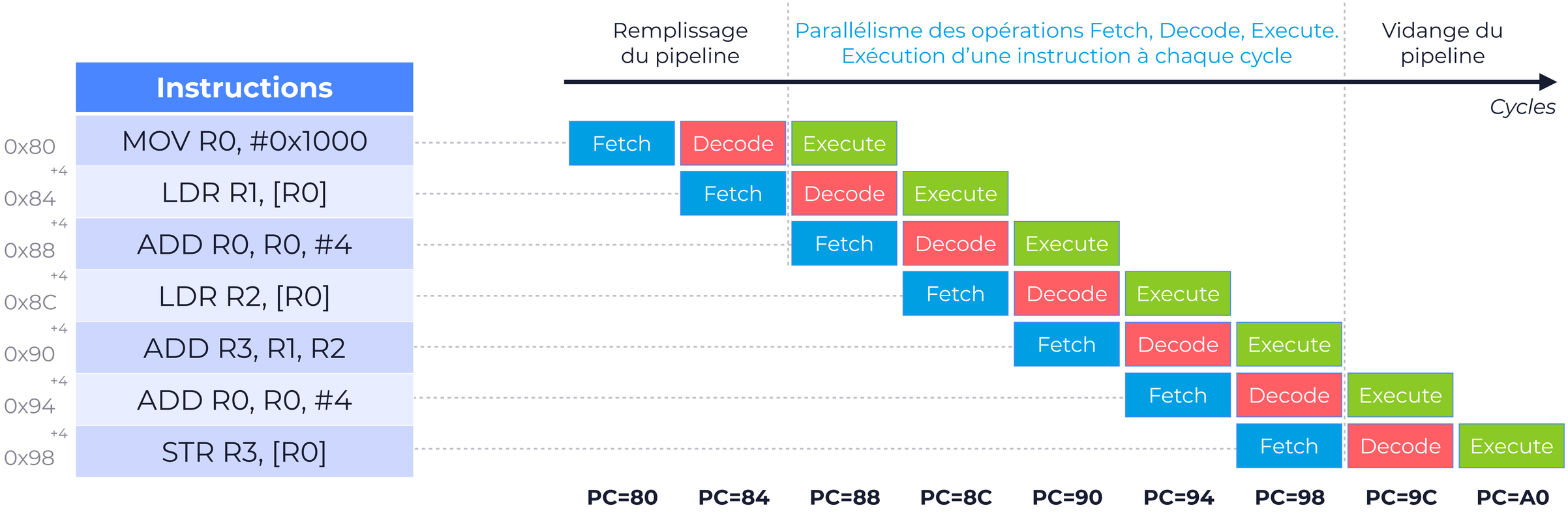
Le pipeline ARM est divisé en 3 étapes : Fetch, Decode, Execute.

PC contient toujours l’adresse de la prochaine instruction à lire et non l’adresse de l’instruction en cours d’exécution.

PC indique donc l’adresse de deux instructions plus loin.



ARM7TDMI : Pipeline d’instructions



PC = @instruction_courante + 8

ARM7TDMI : 37 Registres

16 registres généraux de 32 bits
accessibles tout le temps

- R0 à R12 : registres généraux
- R13 à R15 : registres spécifiques ayant des fonctionnalités prédéfinies
  - R13 : Stack Pointer ou SP
  - R14 : Link Register ou LR (pour les appels de fonctions via l'instruction Branch with Link ou BL
  - R15 : Program Counter ou PC

1 registre d'état CPSR
Current Program Status Register
accessible tout le temps

- Mémorise les résultats des opérations arithmétiques et logiques dans les 4 MSB
  - Bit 31 : Negative ou N
  - Bit 30 : Zero ou Z
  - Bit 29 : Carry ou C.
  - Bit 28 : Overflow ou V
- Mémorise le mode d'exécution dans les 8 LSB (plus de détails ultérieurement)

15 registres généraux de 32 bits
5 registres Saved Program Status Register

accessibles selon le mode d'exécution
(plus de détails ultérieurement)

Différences entre notre processeur et l'ARM7TDMI

Notre processeur

- Mots de 1 octet (8 bits)
- Données sur 1 octet (8 bits)
- Adresses sur 1 octet (8 bits)
- Instructions sur 2 octets (16 bits)
- 4 registres de 1 octet (8 bits)

ARM7TDMI

- Mots de 1 octet (8 bits)
- Données sur 4 octets (32 bits)
- Adresses sur 4 octets (32 bits)
- Instructions sur 4 octets (32 bits)
- 16 registres de 4 octets (32 bits)

Code	*Explications*	*Code ARM7TDMI*
MOV R0, #0x0A LDR R1, [R0] MOV R0, #0x0B *; ou ADD R0, #0x01* LDR R2, [R0] ADD R2, R1 MOV R0, #0x0C *; ou ADD R0, #0x01* STR R2, [R0]	Adresse de la première valeur dans R0 Lecture de la première valeur dans R1 Adresse de la deuxième valeur dans R0 Lecture de la deuxième valeur dans R2 R2 ← R1 + R2 Adresse du résultat dans R0 Ecriture du résultat	MOV R0, **#0x1000** LDR R1, [R0] **ADD R0, R0, #4** LDR R2, [R0] **ADD R2, R1, R2** **ADD R0, R0, #4** STR R2, [R0]

Démo Simulateur

Présentation des sections INTVEC, CODE, DATA

Organisation de la mémoire

Contenu de la mémoire

Exemple de programme sur le **simulateur**

```
SECTION INTVEC
B main ; Etiquette du programme principal
```

Section de la table des vecteurs d'interruption

Début 0x0 - Fin 0x7F

```
addr       00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x00000000 1E 00 00 EA -- -- -- -- -- -- -- -- -- -- -- --
```

```
SECTION CODE
main
MOV R0, #0x1000 ; Adresse de la première valeur à lire
LDR R1, [R0]    ; Chargement de la valeur 1 dans R0
ADD R0, R0, #4  ; Incrémentation de R0 pour lire la 2eme valeur
LDR R2, [R0]    ; Chargement de la valeur 2 dans R1
ADD R2, R1, R2  ; Addition R2 <- R0 + R1
ADD R0, R0, #4  ; Incrémentation de R0 pour enregistrer le résultat
STR R2, [R0]    ; Stockage du résultat
fin
B fin
```

Section du code principal

Début 0x80 - Fin 0x0FFF

```
0x00000080 01 0A A0 E3 00 10 90 E5 04 00 80 E2 00 20 90 E5
0x00000090 02 20 81 E0 04 00 80 E2 00 20 80 E5 FE FF FF EA
0x000000a0 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

Correspondance entre code et mémoire

L'instruction ADD R2, R1, R2 est située à l'adresse 0x90 (5ème mot de 32 bits) et contient 0xE0812002 (stockage Little Endian)

```
SECTION DATA
; Variables contenant des valeurs préinitialisées
v1 ASSIGN32 0x1337
v2 ASSIGN32 0xFD
; Variable allouée sans valeur initiale
resultat ALLOC32 1
```

Section des données

Début 0x1000 - Fin 0x...

```
0x00000ff0 -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
0x00001000 37 13 00 00 FD 00 00 00 34 14 00 00
```

Déclaration / Définition de variables

Un constat : Pas très pratique de devoir connaitre l'adresse mémoire d'une donnée et d'utiliser 2 instructions MOV et LDR pour charger une donnée mémoire dans un registre.

Une solution : Donner des noms aux adresses mémoire et les transformer en variables

Définition de variables
nom ASSIGNxx valeurs

- ➡ nom : nom de la variable
- ➡ xx : nombre de bits (8, 16, 32)
- ➡ valeurs : suite de valeurs à stocker

v1 ASSIGN32 0x1337
tableau1 ASSIGN32 0x48, 0x65, 0x6C

eq. C : int v1=0x1337;
Int tableau1[3] = {0x48, 0x65, 0x6C}

Déclaration de variables
nom ALLOCxx nombre

- ➡ nom : nom de la variable
- ➡ xx : nombre de bits (8, 16, 32)
- ➡ nombre : nombre d'éléments à allouer

resultat ALLOC32 3

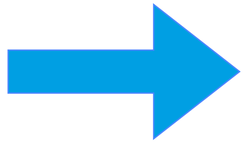
eq. C : int resultat[3];

Pseudo instructions

Une pseudo-instruction est un élément du langage assembleur qui apparait comme une instruction ARM du point de vue programmer mais qui est en réalité traduite en succession d'instructions de base.

Coté programmeur	Coté processeur
Adressage direct d'une variable LDR R0, v1	*Adressage indirect en calculant la position de v1 par rapport à PC* LDR R0, [R15, #0xF78] *R15 + 0xF78 = 0x88 + 0xF78 = 0x1000* *0x1000 est l'adresse de la première case mémoire à lire*
Adressage indirect d'une variable LDR R0, =v1	*Adressage indirect en calculant la position de v1 par rapport à PC* LDR R0, [R15, #0x1C] *R15 + 0x1C = 0x88 + 0x1C = 0xA4* *Le compilateur ARM a rajouté une case mémoire dans la section CODE dans laquelle on retrouve la valeur 0x1000 qui sera l'adresse de la mémoire à lire*

```
MOV R0, #0x1000
LDR R1, [R0]
ADD R0, R0, #4
LDR R2, [R0]
ADD R2, R1, R2
ADD R0, R0, #4
STR R2, [R0]
```



```
LDR R0, v1
LDR R1, v2
ADD R2, R0, R1
LDR R3, =resultat
STR R2, [R3]
```

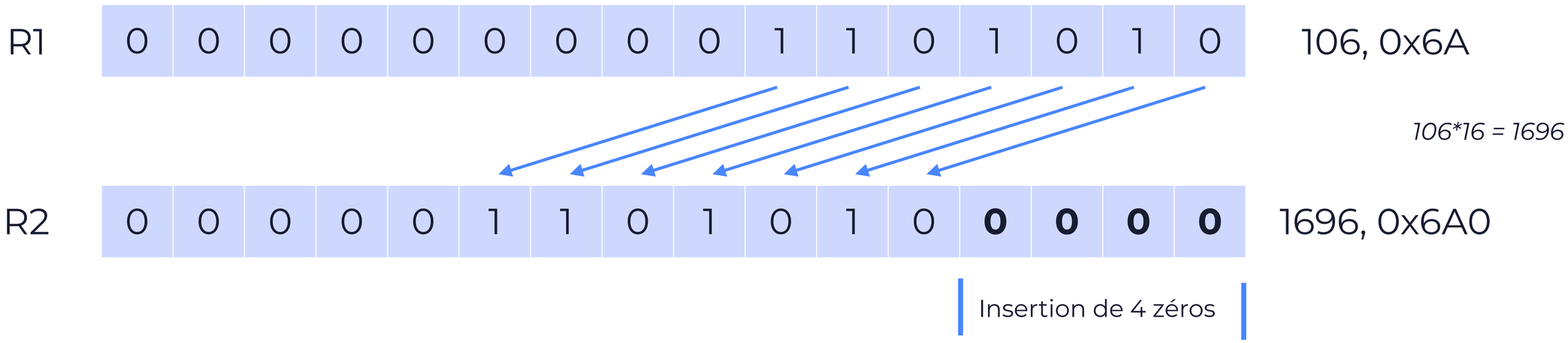
Instructions de **Décalage de bits**

LSL : Logical Shift Left - Décalage vers la gauche

Insertion de 0 à droite - Equivalent à une multiplication par 2^N

Ex 1	LSL R2, R1, #0x4	R2 ← R1 * 2^4
Ex 2	LSL R2, R1, R0	R2 ← R1 * 2^R0

LSL R2, R1, #0x4 Décalage de 4 bits vers la gauche → Multiplication par 2^4 = 16



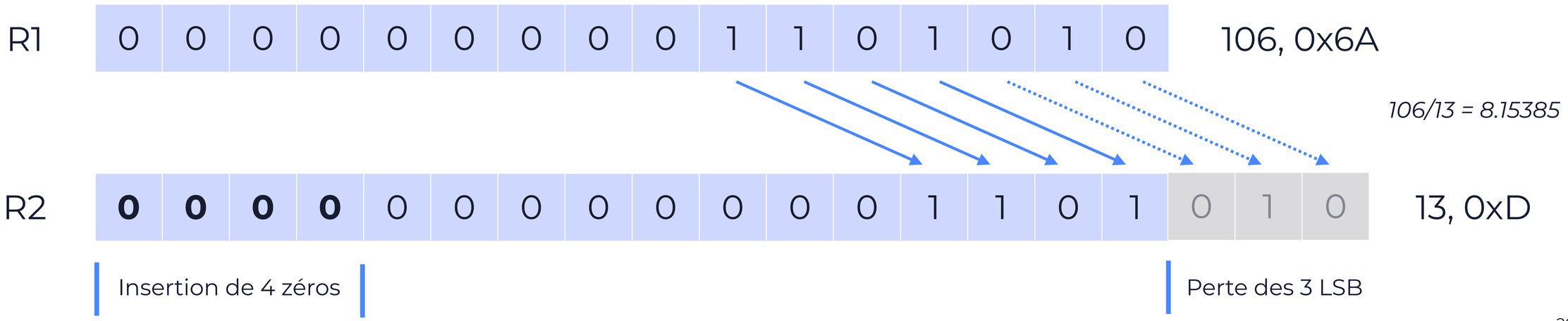
Instructions de Décalage de bits

LSR : Logical Shift Right - Décalage vers la droite

Insertion de 0 à gauche - Equivalent à une division entière non signée par 2^N

Ex 3	LSR R2, R1, #0x3	R2 ← R1 / 2^3
Ex 4	LSR R2, R1, R0	R2 ← R1 / 2^R0

LSR R2, R1, #0x3 Décalage de 3 bits vers la droite → Division entière par 2^3 = 8



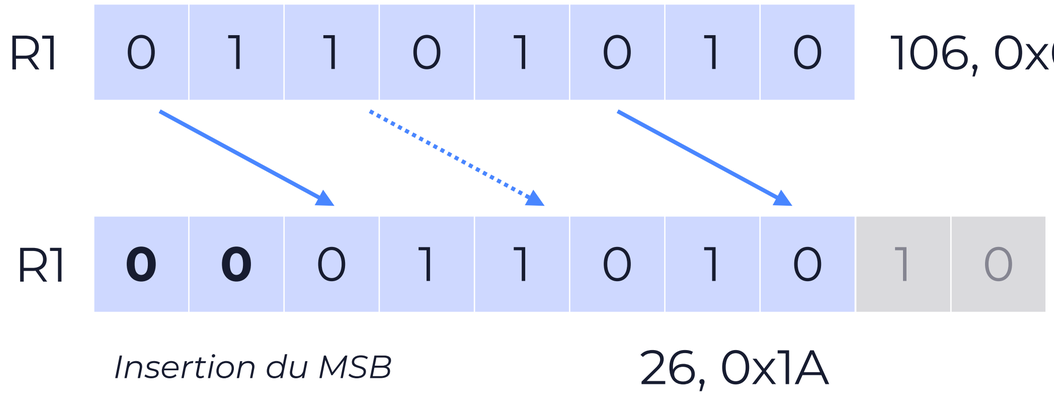
Instructions de Décalage de bits

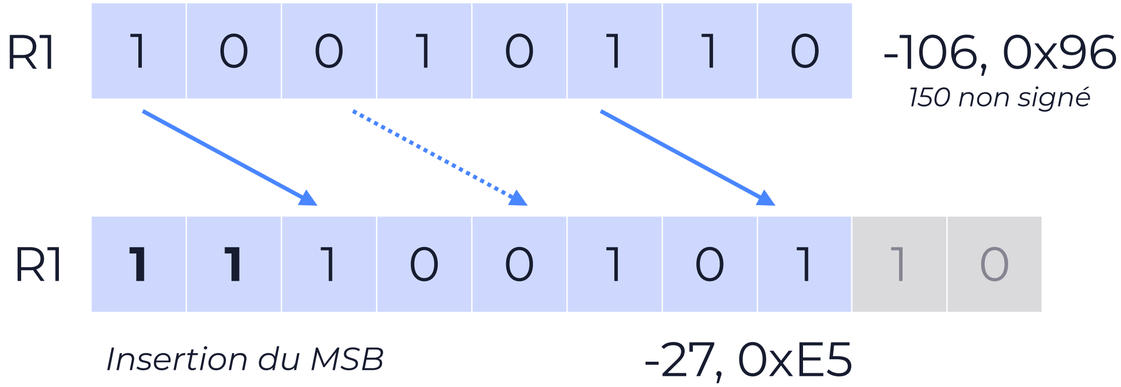
ASR : Arithmetic Shift Right - - Décalage signé vers la droite

Propagation du bit de signe à gauche - Equivalent à une division entière par 2^N

Ex 4	ASR R2, R1, #0x2	R2 ← R1 / 2^2
Ex 5	ASR R2, R1, R0	R2 ← R1 / 2^R0

ASR R2, R1, #0x2 Décalage de 2 bits vers la droite → Division entière par 2^2 = 4





Mouvement avec **Décalage de bits**

Ex 6

MOV R0, R1, LSL #2	R0 ←R1 * 4

```
MOV R1, #0x1000          ; R1 ←0x1000
MOV R0, R1, LSL #0x2     ; R1 * 4 = 0x4000 et R0 ←0x4000
```

R1 pas modifié

Ex 7

LDR R0, [R1, #4]	R0 ←[R1 + 4]

```
MOV R1, #0x1000          ; R1 ←0x1000
LDR R0, [R1, #0x4]       ; R1 + 4 = 0x1004 et R0 ←[0x1004]
```

R1 pas modifié

Ex 8

LDR R0, [R1, R2, LSL #4]	R0 ←[R1 + R2 * 16]

```
MOV R1, #0x1000          ; R1 ←0x1000
MOV R2, #0x10            ; R2 ←0x10
LDR R0, [R1, R2, LSL #0x4] ; R1 + R2*16 = 0x1100  et R0 ←[0x1100]
```

R1, R2 pas modifiés

Variantes de LDR/STR avec **pre-indexing**

On peut modifier le registre source avant le calcul d'accès mémoire (utilisation symbole "!")

Ex 9	LDR R0, [R1, #4]	R0 ←[R1 + 4]
Ex 10	LDR R0, [R1, #4]!	R1 ← R1+4 puis R0 ←[R1]
Ex 11	STR R0, [R1, #4]	R0→[R1+4]
Ex 12	STR R0, [R1, #4]!	R1 ← R1+4 puis [R0] →R1

R1 modifié
Equivalent C
++ptr;

R1 modifié
Equivalent C
++ptr;

```
LDR R0, =source     ; R0 ←source (0x1000)          R0 ← 0x1000
LDR R1, =dest       ; R1 ←dest (0x1010)            R1 ←0x1010
LDR R2, [R0, #4]!   ; R0 ← R0+4 puis R2 ←[R0]      R0 ← R0+4 = 0x1004 puis R2 ←0x65
STR R2, [R1, #4]!   ; R1 ← R1+4 puis R2 → [R1]     R1 ← R1+4 = 0x1014 puis R2=0x65 →0x1014
...
source ASSIGN32 0x48, 0x65, 0x6C, 0xFD
dest ALLOC32 4
```

```
0x00001000  48 00 00 00 65 00 00 00 6C 00 00 00 FD 00 00 00
0x00001010  FF FF FF FF 65 00 00 00 FF FF FF FF FF FF FF FF
0x00001020  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Variantes de LDR/STR avec **post-indexing**

On peut modifier le registre source après le calcul d'accès mémoire

Ex 13	LDR R1, [R0], #0x04	R1←[R0] puis R0 ← R0+4
Ex 14	STR R1, [R0], #0x04	R1→[R0] puis R0 ← R0+4

Equivalent C
ptr++;

```
LDR R0, =source     ; R0 ←source
LDR R1, =dest       ; R1 ←dest
LDR R2, [R0], #4    ; R2 ←[R0] puis R0 ← R0+4
STR R2, [R1], #4    ; R2 →[R1] puis R1 ← R1+4
LDR R2, [R0], #4    ; R2 ←[R0] puis R0 ← R0+4
STR R2, [R1], #4    ; R2 →[R1] puis R1 ← R1+4
LDR R2, [R0], #4    ; R2 ←[R0] puis R0 ← R0+4
STR R2, [R1], #4    ; R2 →[R1] puis R1 ← R1+4
...
source ASSIGN32 0x48, 0x65, 0x6C, 0xFD
dest ALLOC32 4
```

```
0x00001000  48 00 00 00 65 00 00 00 6C 00 00 00 FD 00 00 00
0x00001010  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
0x00001020  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

```
0x00001000  48 00 00 00 65 00 00 00 6C 00 00 00 FD 00 00 00
0x00001010  48 00 00 00 65 00 00 00 6C 00 00 00 FD 00 00 00
0x00001020  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

Récap MOV/LDR/STR

MOV : Déplacements entre des registres seulement

Ex 15

MOV R0, #0x1000 MOV R0, R1 MOV R0, R1, LSL #0x04	*R0 ← 0x100* *R0 ← R1* *R0 ← R1 * 2^4*

LDR/STR: Déplacements de la mémoire vers les registres ou des registres vers la mémoire

Ex 16

LDR R0, [R1] LDR R0, valeur LDR R0, = valeur LDR R0, [R1, #0x04] LDR R0, [R1, R2] LDR R0, [R1, R2, LSL #0x04] LDR R0, [R1, #0x04]! LDR R0, [R1, R2]! LDR R0, [R1], #0x04 LDR R0, [R1], R2	*R0 ← [R1]* *R0 ← valeur* *R0 ← adresse de valeur* *R0 ← [R1 + 0x04]* *R0 ← [R1 +R2]* *R0 ← [R1 +R2* 2^4]* *R1 ← R1 + 0x04 puis R0 ← [R1]* *R1 ← R1 + R2 puis R0 ← [R1]* *R0 ← [R1] puis R1 ← R1 + 0x04* *R0 ← [R1] puis R1 ← R1 + R2*	STR R0, [R1] STR R0, [R1, #0x04] STR R0, [R1, R2] STR R0, [R1, R2, LSL #0x04] STR R0, [R1, #0x04]! STR R0, [R1, R2]! STR R0, [R1], #0x04 STR R0, [R1], R2	*R0→ [R1]* *R0→ [R1 + 0x04]* *R0→ [R1 + R2]* *R0→ [R1 + R2 * 2^4]* *R1 ← R1 + 0x04 puis R0→ [R1]* *R1 ← R1 +R2 puis R0→ [R1]* *R0→ [R1] puis R1 ← R1 + 0x04* *R0→ [R1] puis R1 ← R1 +R2*

Instructions conditionnelles

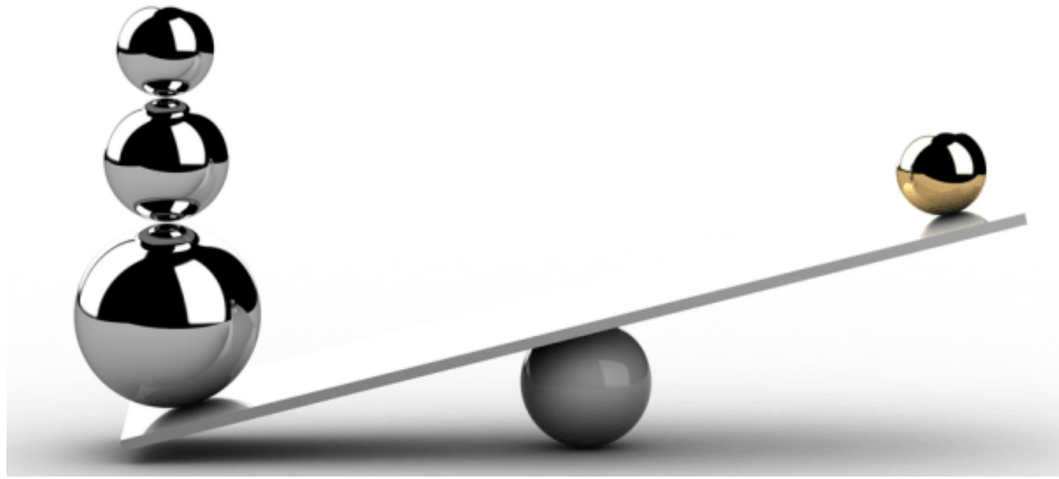
Comment comparer des valeurs situées dans des registres ?

Exemple : Comparaison du registre R1 avec le registre R2

Si R2 > R1 alors R0 ←R2 sinon R0 ← R1 ; *plus grande valeur dans R0*

Besoin de 3 mécanismes :

1. Des instructions pour comparer R1 et R2
2. Un “endroit” pour stocker le résultat des comparaisons
3. Des instructions conditionnelles qui ne sont activées que si les comparaisons répondent à des critères précis



1. Instructions de **comparaison**

CMP Rx Ry
CMP Rx, #CST

Effectue la soustraction Rx - Ry ou Rx - #CST

CMN Rx Ry
CMN Rx, #CST

Effectue l'addition Rx + Ry ou Rx + #CST

TST Rx Ry
TST Rx, #CST

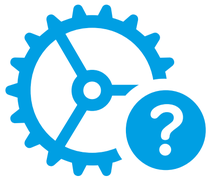
Effectue un AND entre Rx et Ry ou Rx et #CST

TEQ Rx Ry
TEQ Rx, #CST

Effectue un XOR entre Rx et Ry ou Rx et #CST

Instructions arithmétiques en ajoutant un S aux instructions arithmétiques

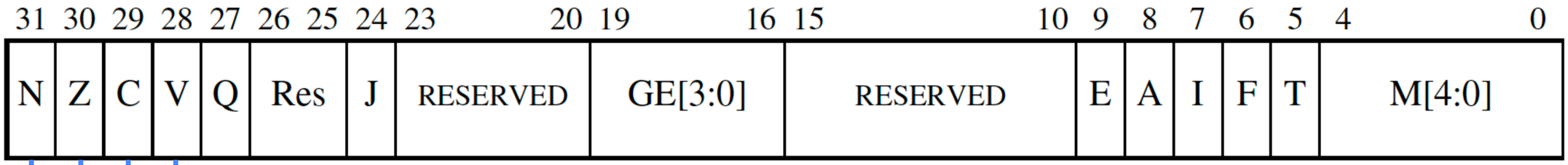
```
SUB   R0, R1, R2      ; R0 ← R1 - R2
SUBS  R0, R1, R2      ; R0 ← R1 - R2 et compare le résultat avec zéro
```



Comment et où sont stockées les résultats des opérations ?

2. Où stocker ?

Le registre CPSR - Current Program Status Register



V : Débordement (overflow)

Produit un 1 si une addition de 2 nombres positifs (négatifs) donne un nombre négatif (positif)

C : Retenue (carry/borrow)

Produit un 1 si une addition de 2 nombres produit une retenue sur le bit de poids fort

Z : Valeur nulle (zero)

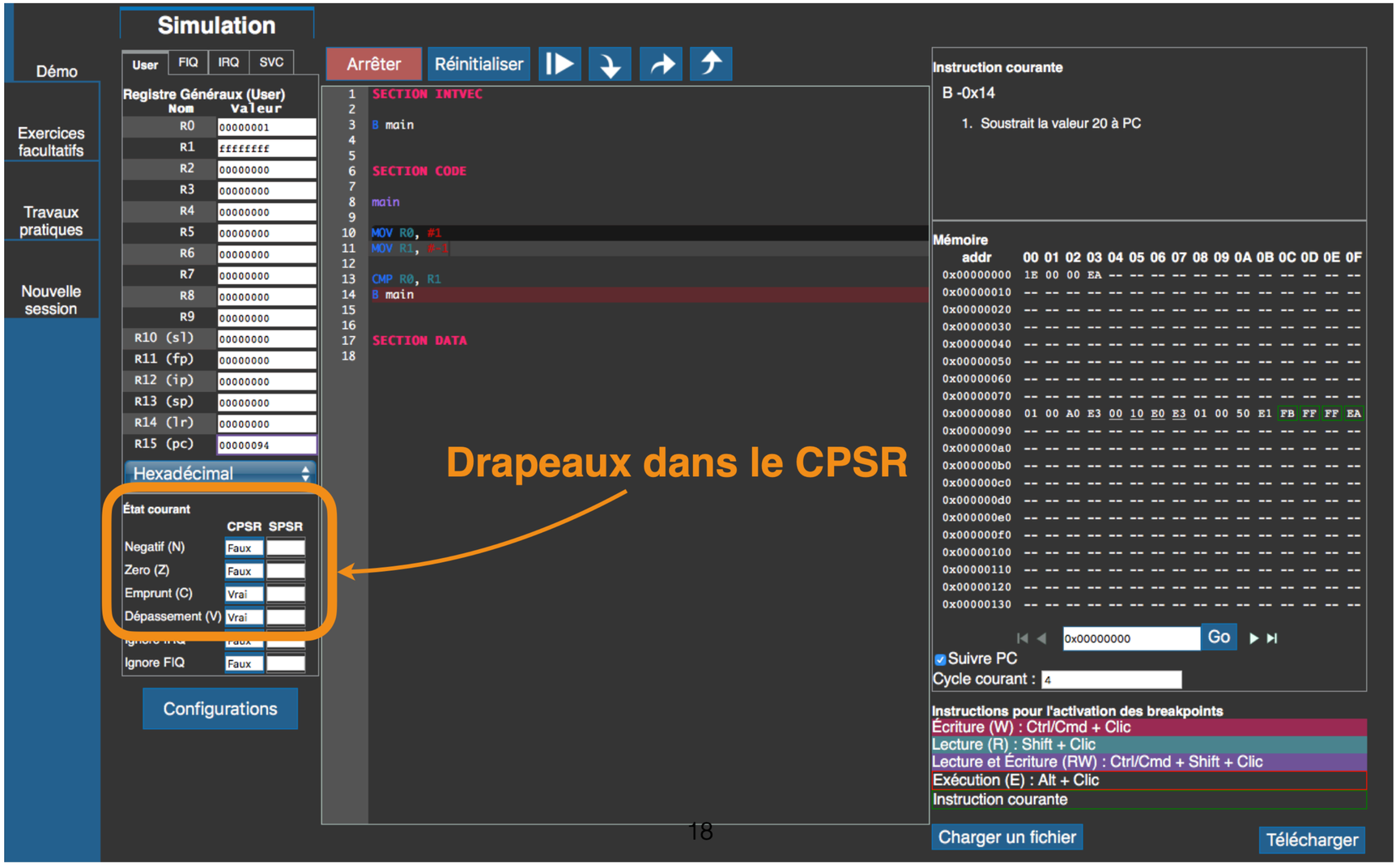
Produit un 1 si le résultat de l'opération est 0, souvent utilisé pour comparer 2 nombres

N : Nombre négatif (négative)

Produit un 1 si le résultat de l'opération est négatif

bits N Z C V = drapeaux (flags)

Dans le **simulateur**



3. Instructions conditionnelles

En ajoutant une condition particulière, l'instruction est effectuée seulement si la condition est vérifiée.

Exemple : MOVEQ R0, R1

R0 ←R1 si et seulement si le drapeau Z = 1

Exemple : ADDNE R0, R1, R2

R0 ←R1+R2 si et seulement si le drapeau Z = 0

0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-

3. Instructions conditionnelles

Comparaison de 2 nombres placés dans les registres R1 et R2

Si R2 > R1 alors R0 ←R2 sinon R0 ← R1 ; *plus grande valeur dans R0*

Ex 17

CMP R1, R2	Calcule R1-R2, change les drapeaux
MOVLE R0, R2	R1 <= R2 alors R0 ←R2
MOVGT R0, R1	R1 > R2 alors R0 ←R1

Etat initial

Flags	N	Z	C	V
	F	F	F	F

CMP R1, R2

R1 < R2

Ex: R1=0x05, R2=0x0F

	N	Z	C	V
	N	Z	C	V
	T	F	F	F

MOVLE RO, R2
exécuté

R2 = R1

Ex: R1=0x05, R2=0x05

	N	Z	C	V
	N	**Z**	**C**	V
	F	**T**	**T**	F

MOVLE RO, R2
exécuté

R1 > R2

Ex: R1=0x0F, R2=0x05

	N	Z	C	V
	N	Z	**C**	V
	F	F	**T**	F

MOVGT RO, R1
exécuté

Pour une soustraction, C fonctionne a l'opposé de l'addition: Si R1-R2<0, alors C=True, sinon C= False

3. Instructions conditionnelles

Calcul de la valeur absolue d'une différence

R0 ← abs (R1 - R2)

Ex 17

CMP R1, R2	Calcule R1-R2, change les drapeaux
SUBLE R0, R2, R1	R1 <= R2 alors R0 ←R2-R1
SUBGT R0, R2, R1	R1 > R2 alors R0 ←R1-R2

Etat initial

Flags

N	Z	C	V
F	F	F	F

CMP R1, R2

R1 < R2

Ex: R1=0x05, R2=0x0F

N	Z	C	V
T	F	F	F

SUBLE RO, R2, R1
exécuté

R2 = R1

Ex: R1=0x05, R2=0x05

N	**Z**	**C**	V
F	**T**	**T**	F

SUBLE RO, R2, R1
exécuté

R1 > R2

Ex: R1=0x0F, R2=0x05

N	Z	**C**	V
F	F	**T**	F

SUBGT RO, R2, R1
exécuté

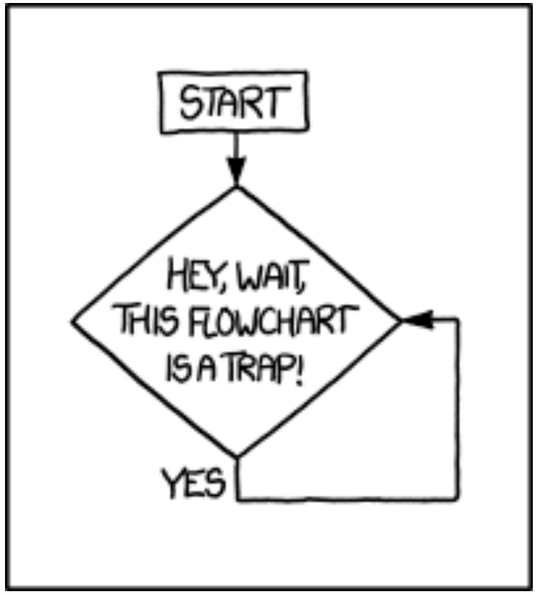
Séquence d'exécution des instructions

Branchements et autres modifications

Par défaut, les instructions s'exécutent séquentiellement et PC est incrémenté automatiquement de 4 octets à chaque nouvelle instruction.

Certains instructions et certaines données nécessitent de modifier l'ordre d'exécution des programmes

1. Instructions de saut direct à une autre instruction
2. Instructions conditionnelles de type "If then else"
3. Répétitions de code avec des boucles de type "for" or "while"
4. Appel de sous programme de type "fonctions"



Branchement **inconditionnel**

ou Saut d'une adresse mémoire à une autre

L'instruction B etiquette permet de sauter directement aux instructions situées juste après l'étiquette.

Pour cela, le programme assembler convertit automatiquement l'étiquette par son adresse relative à PC.

```
B calcul ; Saute à l'adresse indiquée par calcul
...
...
calcul ; étiquette calcul
MOV R0, #0x42
MOV R1, #0x1000
...
```

Sur le simulateur ARM, la première instruction est B main que l'assembleur traduit en B 0x78. Sachant que PC=0x08, la nouvelle valeur de PC sera 0x78+0x08 =0x80 qui est la première adresse de la section CODE.

Branchements conditionnels

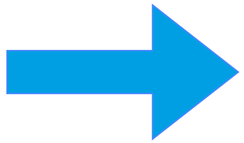
ou Saut d'une adresse mémoire à une autre sous condition

L'instruction B etiquette peut être utilisée avec une condition (NE, EQ, LT, GT, …)

Exemple du if

```
…

If (maVar == 4) {
    // Instructions à exécuter
}
// On continue
…
```



```
…
LDR     R0, maVar   ; Met la valeur de maVar dans R0
CMP     R0, #4      ; Calcul R0 - #4 et met à jour les drapeaux
BNE     PasEgal     ; NE = « Not Equal »

; exécute la tâche 1 (nous sommes dans le « if »)
MOV R1, #1

PasEgal
; le programme continue pour tout le monde
MOV R1, #2
…
```

Branchements conditionnels

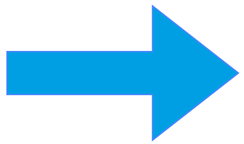
ou Saut d'une adresse mémoire à une autre sous condition

L'instruction B etiquette peut être utilisée avec une condition (NE, EQ, LT, GT, ...)

Exemple du if/else

```
...

If (maVar == 4) {
    // Instructions à exécuter
} else {
    // Autres instructions
}
// On continue
...
```



```
...
LDR     R0, maVar   ; Met la valeur de maVar dans R0
CMP     R0, #4      ; Calcul R0 - #4 et met à jour les drapeaux
BNE     PasEgal     ; NE = « Not Equal »

; exécute la tâche 1 (nous sommes dans le « if »)
MOV R1, #1
B Continue          ; On saute le else et on va à Continue

PasEgal             ; On arrive dans le else
; exécute la tâche 2 (nous sommes dans le « else »)
MOV R1, #2

Continue            ; Pour tout le monde
; le programme continue
MOV R1, #2
...
```

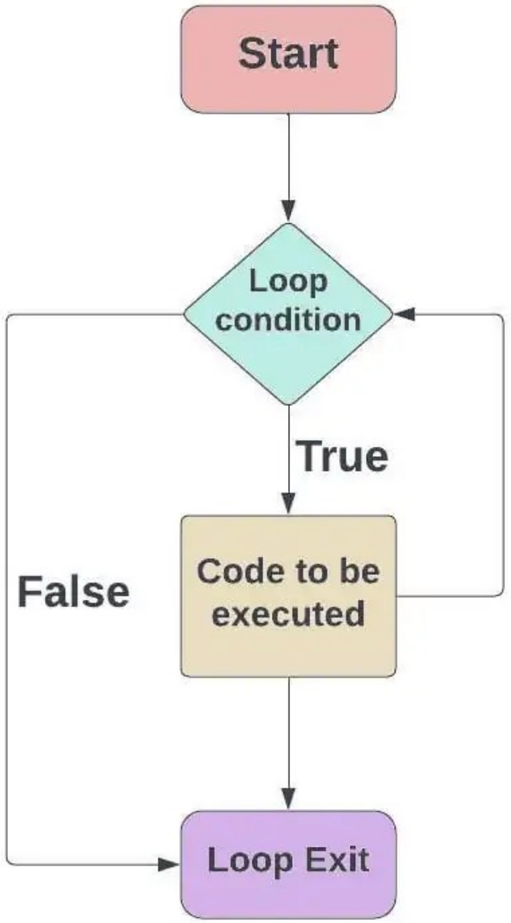
Boucle For

ou Répétition de code N fois (N étant connu)

Principes d'une boucle :

1. Définir les **conditions initiales** de la boucle
2. Tester la **condition d'arrêt**
3. Faire une **opération mathématique** qui entrainera la validation de la condition d'arrêt

```
for (int cpt=0; cpt<100; cpt++) {
    // Code à exécuter dans la boucle
    }
// Suite du code
```

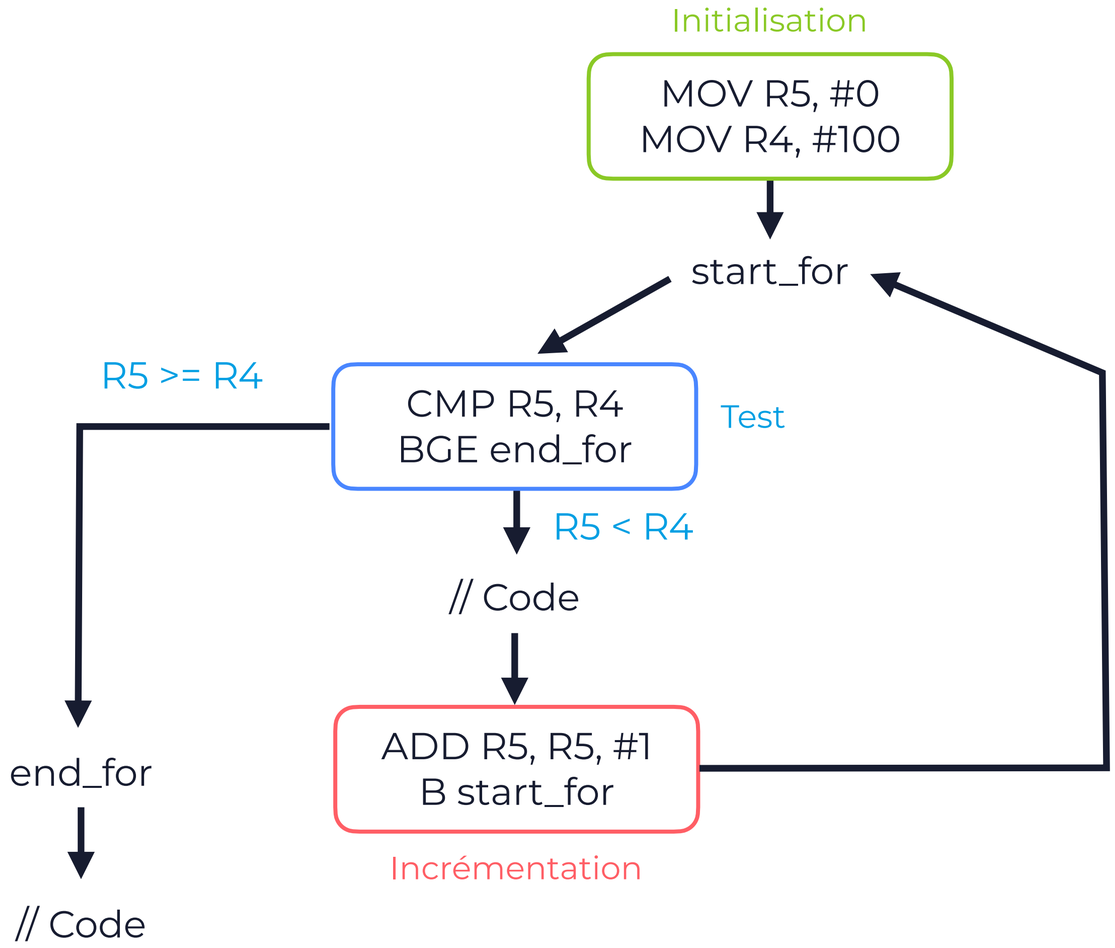


Boucle For

ou Répétition de code N fois (N étant connu)

```
for (int cpt=0; cpt<100; cpt++) {
    // Code à exécuter dans la boucle for
    }
// Suite du code
```

```
MOV R5, #0          ; cpt = 0
MOV R4, #100        ; 100 iterations
start_for           ; début de la boucle
CMP R5, R4          ; comparaison avec 100
BGE end_for         ; si cpt >= 100, on sort de la boucle
; Code a exécuter dans la boucle
ADD R5, R5, #1      ; cpt++
B start_for
end_for             ; fin de la boucle
; le programme continue...
```



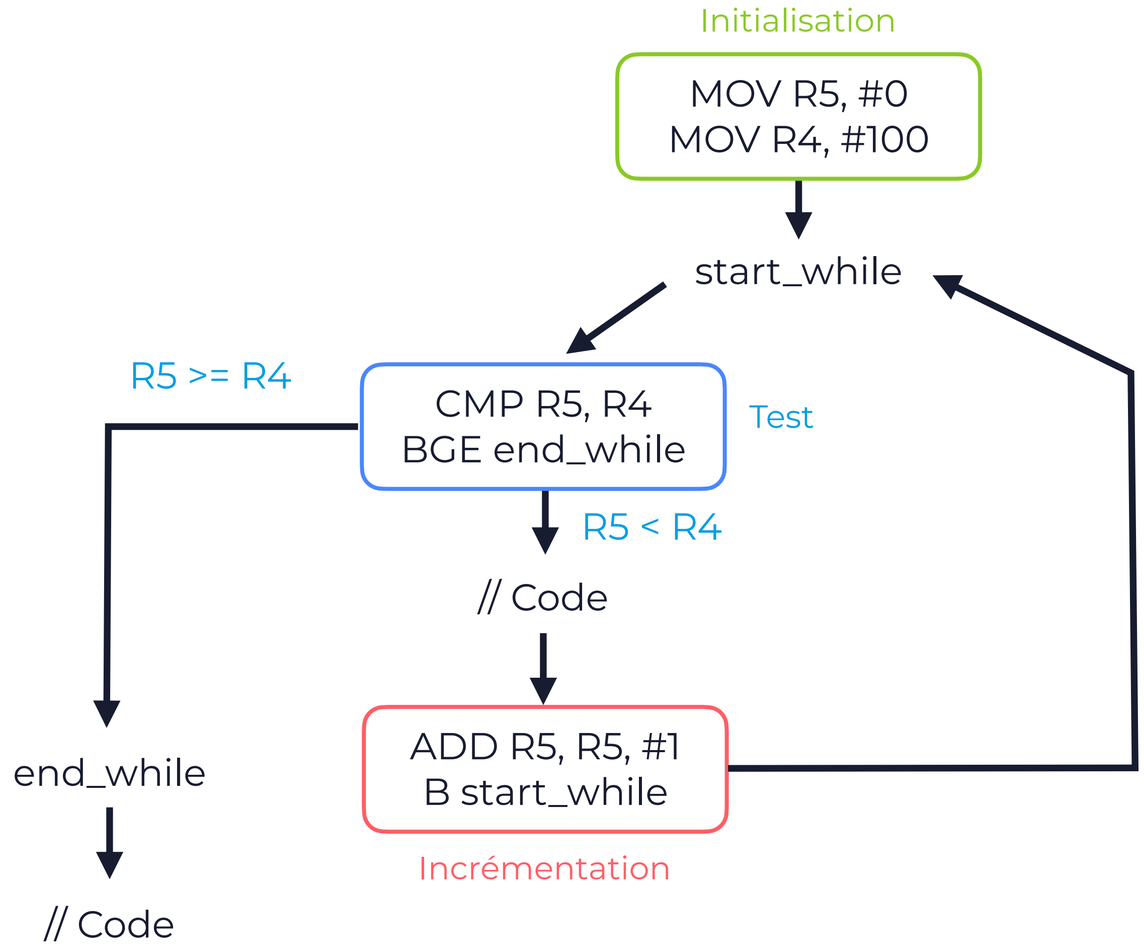
Utilisation d'un test GE (>=) pour sortir de la boucle, correspondant à l'inverse du test cpt < 100

Boucle While

ou Répétition de code

```
int cpt=0;
while (cpt<100) {
    // Code à exécuter dans la boucle while
    cpt++
    }
// Suite du code
```

```
MOV R5, #0          ; cpt = 0
MOV R4, #100        ; 100 iterations
start_while         ; début de la boucle
CMP R5, R4          ; comparaison avec 100
BGE end_while       ; si cpt >= 100, on sort de la boucle
; Code a exécuter dans la boucle
ADD R5, R5, #1      ; cpt++
B start_while
end_while           ; fin de la boucle
; le programme continue...
```



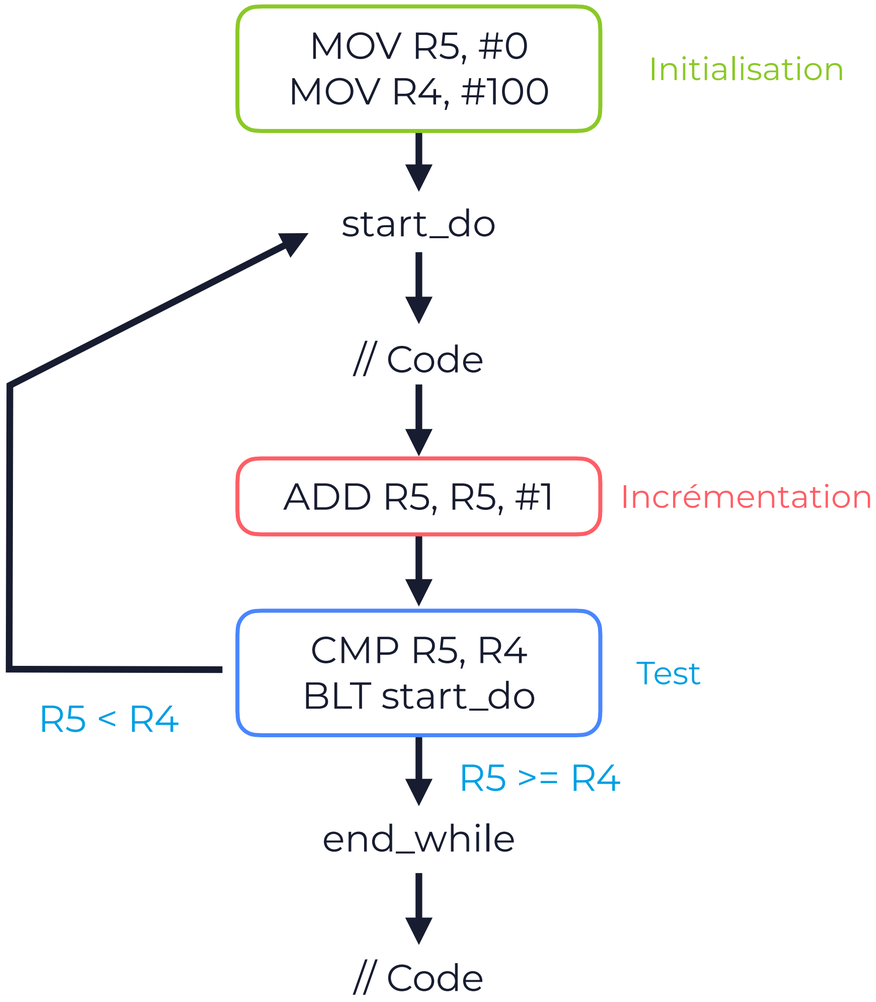
Code strictement identique à celui de la boucle for

Boucle do ... While

ou Répétition de code

```
int cpt=0;
do {
    // Code à exécuter dans la boucle do ... while
    cpt++
    } while (cpt<100);
// Suite du code
```

```
MOV R5, #0          ; cpt = 0
MOV R4, #100        ; 100 iterations
start_do            ; début de la boucle
; Code a exécuter dans la boucle
ADD R5, R5, #1      ; cpt++
CMP R5, R4          ; comparaison avec 100
BLT start_do        ; si cpt >= 100, on sort de la boucle
; fin de la boucle le programme continue...
```



Test fait à la fin de la boucle
Test LT pour itérer la boucle identique au test cpt < 100

Exemple concret de **boucles**

Recopie des éléments d'un tableau vers un autre tableau

```
int source[4] = {0x48, 0x65, 0x6C, 0xFD};
int dest[4];

for (int i = 0; i < 4; I++) {
  dest[i] = source[I];
}
```

Les données sources sont disponibles dans un tableau pré-rempli :
source ASSIGN32 0x48, 0x65, 0x6C, 0xFD

Le tableau destination est alloué :
dest ALLOC32 4

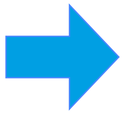
```
LDR R1, =source
LDR R2, =dest
MOV R0, #0          ; cpt = 0
start_for           ; début de la boucle
CMP R0, #4          ; comparaison avec 4
BGE end_for         ; si cpt >= 100, on sort de la boucle
; Début copie des données en utilisant R3 comme registre tampon
LDR R3, [R1], #4    ; R3<-[source]; source ++
STR R3, [R2], #4    ; R3->[dest]; source ++
; Fin copie des données
ADD R0, R0, #1      ; cpt++
B start_for
end_for             ; fin de la boucle
; le programme continue...
```

Exemple concret de **boucles**

Recopie des éléments d'un tableau vers un autre tableau

```
int source[4] = {0x48, 0x65, 0x6C, 0xFD};
int dest[4];

for (int i = 0; i < 4; I++) {
  dest[i] = source[I];
}
```



```
LDR R1, =source
LDR R2, =dest
MOV R0, #4          ; cpt = 0
start_for           ; début de la boucle
SUBS R0, R0, #1     ; décrémentation et modification des flags
BLT end_for         ; si cpt >= 100, on sort de la boucle
; Début copie des données
LDR R3, [R1], #4    ; R3<-[source]; source ++
STR R3, [R2], #4    ; R3->[dest]; source ++
; Fin copie des données
B start_for
end_for             ; fin de la boucle
; le programme continue...
```

En partant de l'indice 4 et en décrémentant, on peut utiliser l'instruction SUBS et les flags.
Lorsque R0 < 0, le flag N passe à true et permet le branchement BLT qui termine la boucle.

Il est toujours préférable de faire une comparaison à zéro.
Le code assembleur est souvent plus simple. Il repose ici sur SUBS plutôt que sur CMP / ADD.

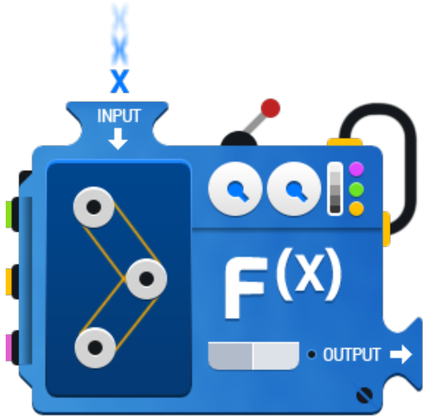
Fonctions en assembleur ARM

Peut-on regrouper des instructions dans une fonction ?

Une fonction (ou routine) encapsule un ensemble d'instructions qu'on utilise souvent, afin d'éviter des "copier/coller" qui sont source d'erreurs.

En assembleur ARM, une fonction est un ensemble d'instructions :

1. Situées à une adresse spécifique accessible par une étiquette.
2. Accessible par un branchement à cette étiquette
3. Utilisant des registres comme paramètres d'entrée/sortie
4. Possédant une adresse de retour lui permettant de revenir à l'instruction devant être exécutée après la fonction.



Fonctions en assembleur ARM

Un premier exemple : bon ou mauvais ?

```
...
B maFonction        ; Branchement vers maFonction
retour              ; Etiquette pour retour après fonction
MOV R1, R0          ; Suite des instructions
...
...
...

; fonction maFonction
maFonction
ADD R2, R1, R0      ; Liste des instructions de la fonction
...
B retour            ; Fin de la fonction et retour vers l'étiquette
...
```

L'instruction B maFonction permet de sauter directement aux instructions situées juste après l'étiquette.

On sort de la fonction après les calculs et on revient à l'étiquette retour.

Mauvais

Fonctions en assembleur ARM

Pourquoi mauvais ?

```
...
B maFonction        ; Branchement vers maFonction
retour              ; Etiquette pour retour après fonction
MOV R1, R0          ; Suite des instructions
...
B maFonction        ; 2ème appel de maFonction
MOV R2, R0          ; Suite des instructions
...

; fonction maFonction
maFonction
ADD R2, R1, R0      ; Liste des instructions de la fonction
...
B retour            ; Fin de la fonction et retour vers l'étiquette
...
```

Impossible de faire un autre appel de maFonction car l'instruction B retour nous ramènera toujours à la sortie du premier appel de la fonction.

Fonctions et appel de fonctions

Appel de fonction avec BL (Branch and Link)

```
...
BL maFonction      ; Branchement vers maFonction
MOV R1, R0         ; Suite des instructions
...
...
...

; fonction maFonction
maFonction
ADD R2, R1, R0     ; Liste des instructions de la fonction
...
...
```

Chaque appel de fonction doit mémoriser sa propre adresse de retour avant de s'exécuter.

Utilisation de l'instruction BL (Branch and Link) :

1. Stocke l'adresse de retour dans le registre R14 ou LR (Link Register)
2. Saute à l'étiquette maFonction

Plus besoin d'étiquette de retour.

LR = PC - 4 car LR est égal à l'adresse située immédiatement après l'instruction BL.

Fonctions et appel de fonctions

Retour de fonction avec BX (Branch and Exchange)

```
...
BL maFonction      ; Branchement vers maFonction
MOV R1, R0         ; Suite des instructions
...
BL maFonction      ; 2ème appel de maFonction
MOV R2, R0         ; Suite des instructions
...

; fonction maFonction
maFonction
ADD R2, R1, R0     ; Liste des instructions de la fonction
...
BX LR              ; Fin de la fonction et retour vers LR
```

A la fin de la fonction, on veut revenir à l'adresse de retour qui a été stockée dans LR par l'instruction BL.

Utilisation de l'instruction BX (Branch and eXchange) :

1. Récupération de l'adresse LR et modification de PC (PC ← LR)
2. Branchement à PC

Fonctionne parfaitement avec plusieurs appels de la même fonction.

Fonctions et appel de fonctions

Des fonctions imbriquées ?

```
...
BL maFonction1    ; Branchement vers maFonction
MOV R1, R0        ; Suite des instructions
...
; fonction maFonction1
maFonction1
BL maFonction2 ; Appel de maFonction2 dans maFonction1
SUB R3, R1, R0    ; Première instruction après maFonction2
...
BX LR             ; Fin de maFonction1 et retour vers LR
...
; fonction maFonction2
maFonction2
MOV R2, R1        ; Liste des instructions de maFonction2
...
BX LR             ; Fin de maFonction2 et retour vers LR
```

L'appel de maFonction1 mémorise l'adresse de retour dans LR.

maFonction1 appelle maFonction2 qui, à son tour, mémorise son adresse de retour dans LR et écrase la précédente valeur.

Lorsque l'instruction BX LR de maFonction1 est exécutée, la valeur de LR n'est pas la bonne et on se branche à la première instruction après maFonction2 et on boucle indéfiniment dans maFonction1.

1 seul registre LR

Mise en place d'une **pile**

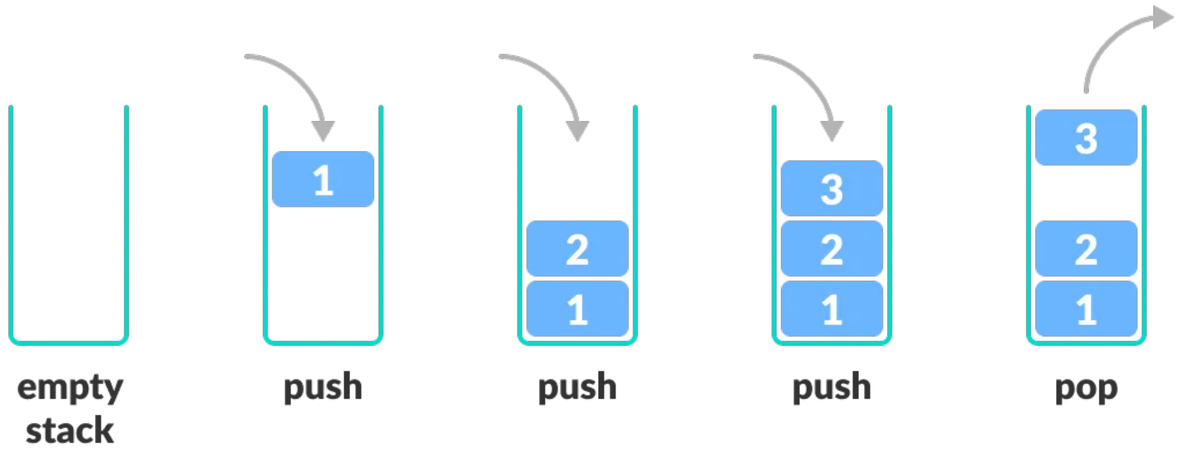
ou Comment sauvegarder le "contexte" du processeur lors de l'appel d'une fonction

Une pile (ou stack) est une structure de données très utilisée en informatique :

- ➡ Basée sur le principe LIFO (Last in, First Out)
- ➡ Utilisant 2 opérations simples : on empile (push) sur le dessus de la pile et on dépile (pop) l'élément le plus haut.

Dans l'ARM, le registre R13 ou SP est le pointeur de pile (Stack Pointer) :

- ➡ SP contient l'adresse du dessus de la pile
- ➡ SP est décrémenté ou incrémenté automatiquement en fonction des PUSH/POP



Ne modifiez jamais directement SP. Utilisez toujours PUSH et POP

Mise en place d'une **pile**

ou Comment sauvegarder le "contexte" du processeur lors de l'appel d'une fonction

Stockage d'une donnée dans la pile : **PUSH {Rx}**

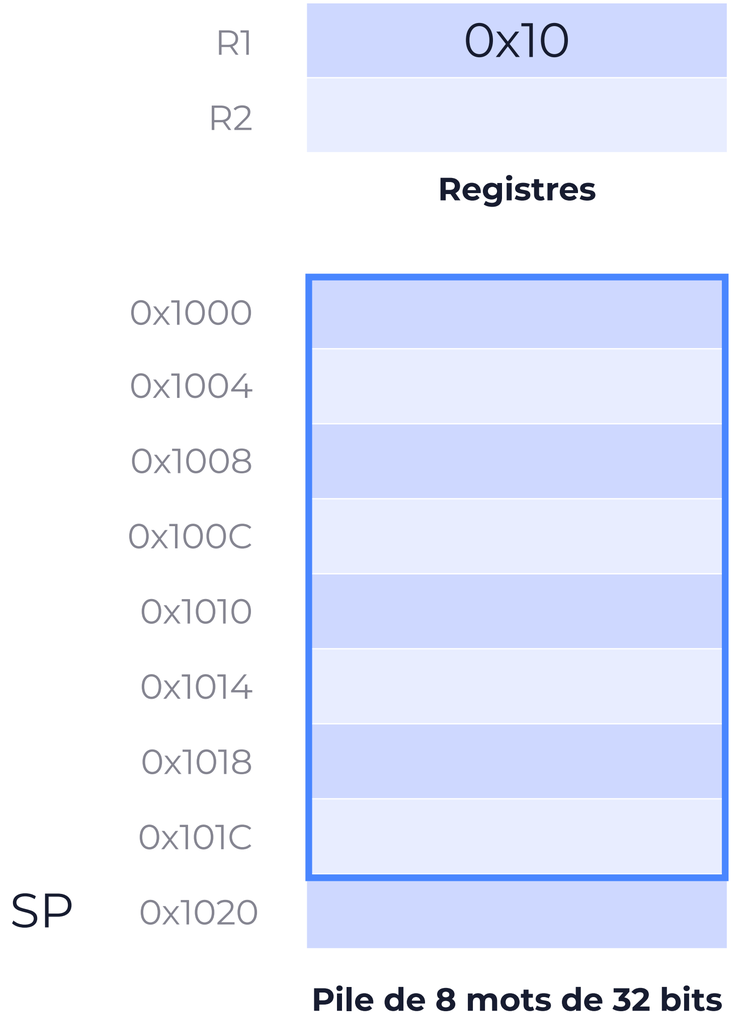
➡ Décrémentation du pointeur de pile : SP ← SP - 4

➡ Copie le contenu du registre Rx à l'adresse pointée par SP

```
LDR SP, =pile          ; Initialisation du pointeur de pile au début du tableau
ADD SP, SP, #32        ; Déplacement de SP à l'adresse 1020 (fin du tableau)

MOV R1, #0x10
PUSH {R1}
PUSH {R1}
POP {R2}

SECTION DATA
pile ALLOC32 8         ; Allocation d'un espace mémoire de 32 octets
```



Mise en place d'une **pile**

ou Comment sauvegarder le "contexte" du processeur lors de l'appel d'une fonction

Stockage d'une donnée dans la pile : **PUSH {Rx}**

➡ Décrémentation du pointeur de pile : SP ← SP - 4

➡ Copie le contenu du registre Rx à l'adresse pointée par SP

```
LDR SP, =pile          ; Initialisation du pointeur de pile au début du tableau
ADD SP, SP, #32        ; Déplacement de SP à l'adresse 1020 (fin du tableau)

MOV R1, #0x10
PUSH {R1}              ; SP ← SP - 4  = 0x1020 - 4 = 0x101C et R1 → [SP]
PUSH {R1}
POP {R2}

SECTION DATA
pile ALLOC32 8         ; Allocation d'un espace mémoire de 32 octets
```

Mise en place d'une **pile**

ou Comment sauvegarder le "contexte" du processeur lors de l'appel d'une fonction

Stockage d'une donnée dans la pile : **PUSH {Rx}**

➡ Décrémentation du pointeur de pile : SP ← SP - 4

➡ Copie le contenu du registre Rx à l'adresse pointée par SP

```
LDR SP, =pile          ; Initialisation du pointeur de pile au début du tableau
ADD SP, SP, #32        ; Déplacement de SP à l'adresse 1020 (fin du tableau)

MOV R1, #0x10
PUSH {R1}              ; SP ← SP - 4  = 0x1020 - 4 = 0x101C et R1 → [SP]
PUSH {R1}              ; SP ← SP - 4  = 0x101C - 4 = 0x1018 et R1 → [SP]
POP {R2}

SECTION DATA
pile ALLOC32 8         ; Allocation d'un espace mémoire de 32 octets
```

Mise en place d'une **pile**

ou Comment sauvegarder le "contexte" du processeur lors de l'appel d'une fonction

Récupération d'une donnée de la pile : **POP {Rx}**

- ➡ Copie le contenu de l'adresse pointée par SP dans le registre Rx
- ➡ Incrémentation du pointeur de pile : SP ← SP + 4

```
LDR SP, =pile          ; Initialisation du pointeur de pile au début du tableau
ADD SP, SP, #32        ; Déplacement de SP à l'adresse 1020 (fin du tableau)

MOV R1, #0x10
PUSH {R1}              ; SP ← SP - 4  = 0x1020 - 4 = 0x101C et R1 → [SP]
PUSH {R1}              ; SP ← SP - 4  = 0x101C - 4 = 0x1018 et R1 → [SP]
POP {R2}               ; R2 ← [SP] et SP ← SP + 4  = 0x1018 + 4 = 0x101C

SECTION DATA
pile ALLOC32 8         ; Allocation d'un espace mémoire de 32 octets
```

Mise en place d'une **pile**

ou Comment sauvegarder le "contexte" du processeur lors de l'appel d'une fonction

Stockage de plusieurs données dans la pile : **PUSH {Rx, Ry, Rz, ...}**

- ➡ Les registres sont empilés du plus grand au plus petit indice de registre, quel que soit l'ordre des registres passés à PUSH
- ➡ PUSH {R1, R2, R3} est équivalent à PUSH {R3, R2, R1}

```
MOV R1, #0x10
MOV R2, #0x20
MOV R3, #0x30

PUSH {R2, R3, R1}    ; SP ← SP - 0xC  = 0x1020 - 0xC = 0x1011 et R3, R2, R1 → [SP]
POP {R6, R5}

SECTION DATA
pile ALLOC32 8       ; Allocation d'un espace mémoire de 32 octets
```

Mise en place d'une **pile**

ou Comment sauvegarder le "contexte" du processeur lors de l'appel d'une fonction

Lecture de plusieurs données dans la pile : **POP {Rx, Ry, Rz, ...}**

- ➡ Les registres sont dépilées du plus petit au plus grand indice de registre, quel que soit l'ordre des registres passés à POP
- ➡ PUSH {R5, R6} est équivalent à PUSH {R6, R5}

```
MOV R1, #0x10
MOV R2, #0x20
MOV R3, #0x30

PUSH {R2, R3, R1}   ; SP ← SP - 0xC  = 0x1020 - 0xC = 0x1011 et R3, R2, R1 → [SP]
POP {R6, R5}        ; R5 ← [SP] et SP ← SP + 4  et R6 ← [SP] et SP ← SP + 4

SECTION DATA
pile ALLOC32 8      ; Allocation d'un espace mémoire de 32 octets
```

Fonctions et appel de fonctions

Des fonctions imbriquées en utilisant la pile

```
BL maFonction1    ; Branchement vers maFonction
MOV R1, R0        ; Suite des instructions
...
; fonction maFonction1
maFonction1
PUSH {LR}         ; Sauvegarde de LR dans la pile
BL maFonction2    ; Appel de maFonction2 dans maFonction1
SUB R3, R1, R0    ; Première instruction après maFonction2
...
POP {LR}          ; Restauration de la valeur originale de LR
BX LR             ; Fin de maFonction1 et retour vers LR
...
; fonction maFonction2
maFonction2
MOV R2, R1        ; Liste des instructions de maFonction2
...
BX LR             ; Fin de maFonction2 et retour vers LR
```

Pour éviter que l'appel de maFonction2 écrase la valeur de LR mémorisé par maFonction1, il suffit de sauvegarder LR dans la pile au début de maFonction1 puis de restaurer LR avant de sortir de maFonction1.

Sauvegardez et Restaurez systématiquement LR dans chaque fonction qui appelle une autre fonction.

Fonctions et appel de fonctions

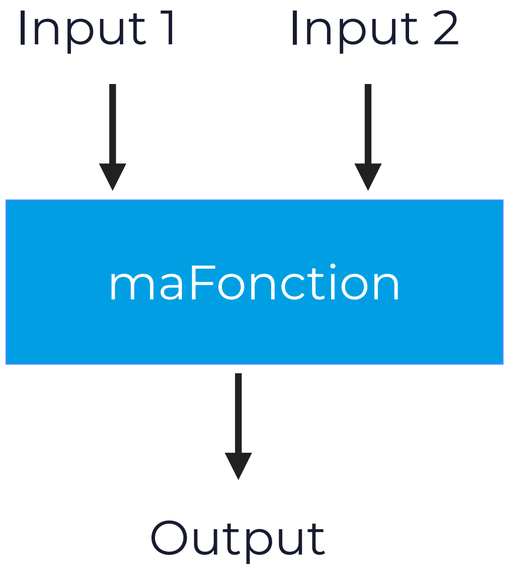
Passage de paramètres et Valeurs de retour

Paramètres = entrées de la fonction

- ➡ Moins de 4 : on utilise directement les registres R0 à R3
- ➡ Plus de 4 : on utilise la pile

Valeur de retour = sortie de la fonction = résultat

- ➡ 1 seul résultat : on utilise directement le registre R0
- ➡ Plus de 1 résultat : on utilise la pile



Exemple de fonction avec 2 entrées et une sortie

Fonctions et appel de fonctions

Passage de paramètres et Valeurs de retour

Ex : y = a x + b avec en entrées R0=x, R1=a, R2=b et en sortie y dans R0

```
MOV R0, #4          ; x = première entrée dans R0
MOV R1,  #5         ; a = deuxième entrée dans R1
MOV R2, #-3         ; b = troisième entrée dans R1
MOV R3, #17         ; autre valeur utilisée par le programme

BL axplusb          ; on appelle la fonction
MOV R4, R0          ; on stocke le résultat dans R4
CMP R3, R4          ; on compare le résultat avec R3
....                ; on continue les opérations

axplusb             ; 3 paramètres : R0, R1, R2 - 1 résultat : R0
  MUL R3, R0, R1    ; y = ax stocké dans un registre temporaire
  ADD R0, R3, R2    ; y = ax+b stocké dans R0
  BX LR             ; Fin de axplusb et retour vers LR
```

Quel est le problème de ce code ?

La fonction utilise le registre R3 pour faire des calculs intermédiaires et écrase la valeur de R3 utilisé par le programme principal.

Pourquoi ce problème ?

1. Le code principal ne connait pas les registres utilisés par les fonctions
2. Les fonctions ne connaissent pas les registres utilisés par le code principal

Une solution ?

Protéger les registres en utilisant la pile

Fonctions et appel de fonctions

Passage de paramètres et Valeurs de retour

Ex : y = a x + b avec en entrées R0=x, R1=a, R2=b et en sortie y dans R0

```
PUSH {R0, R1, R2}.   ; on sauvegarde les entrees sorties
BL axplusb           ; on appelle la fonction
MOV R4, R0           ; on stocke le résultat dans R4
POP {R0, R1, R2}     ; on restaure les entrees sorties
CMP R3, R4           ; on compare le résultat avec R3
....                 ; on continue les opérations

axplusb              ; 3 paramètres : R0, R1, R2 - 1 résultat : R0
  PUSH {R3, LR}      ; on sauvegarde les registres utilisés
  MUL R3, R0, R1     ; y = ax stocké dans registre temporaire
  ADD R0, R3, R2     ; y = ax+b stocké dans R0
  POP {R3, LR}       ; on restaure les registres utilisés
  BX LR              ; Fin de axplusb et retour vers LR
```

Dans le code principal :

1. On sauvegarde dans la pile tous les registres utilisés en entrée/sortie de la fonction
2. On restaure les registres après l'appel de la fonction

Dans la fonction :

1. On sauvegarde dans la pile tous les registres utilisés par la fonction, autres que les entrées/sortie.
2. On inclut LR par défaut.
3. On restaure les registres avant de sortir.

Fonctions et **appel de fonctions**

	PUSH {R0, R1, R2}	PUSH {R3, LR}	MUL R3, R0, R1	ADD R0, R3, R2	POP {R3, LR}	MOV R4, R0	POP {R0, R1, R2}
R0	0x4	0x4	0x4	**0x11**	0x11	0x11	**0x4**
R1	0x5	0x5	0x5	0x5	0x5	0x5	**0x5**
R2	0x-3	0x-3	0x-3	0x-3	0x-3	0x-3	**0x-3**
R3	0x11	0x11	**0x14**	0x14	**0x11**	0x11	0x11
R4						**11**	
LR		**0xA0**	0xA0	0xA0	**0xA0**	0xA0	0xA0
SP	**0x1014**	**0x100C**	0x100C	0X100C	**0x1014**	0x1014	**0X1020**
0x1000							
0x1004							
0x1008							
0x100C		**0x11**	0x11	0x11	~~0x11~~	~~0x11~~	~~0x11~~
0x1010		**0xA0**	0xA0	0xA0	~~0xA0~~	~~0xA0~~	~~0xA0~~
0x1014	**0x4**	0x4	0x4	0x4	0x4	0x4	~~0x4~~
0x1018	**0x5**	0x5	0x5	0x5	0x5	0x5	~~0x5~~
0x101C	**0x-3**	0x-3	0x-3	0x-3	0x-3	0x-3	~~0x-3~~

```
PUSH {R0, R1, R2}
BL axplusb
MOV R4, R0
POP {R0, R1, R2}
CMP R3, R4
....

axplusb
  PUSH {R3, LR}
  MUL R3, R0, R1
  ADD R0, R3, R2
  POP {R3, LR}
  BX LR
```

Fonctions et appel de fonctions

Passage de N paramètres (N > 4)

Paramètres indépendants en entrée

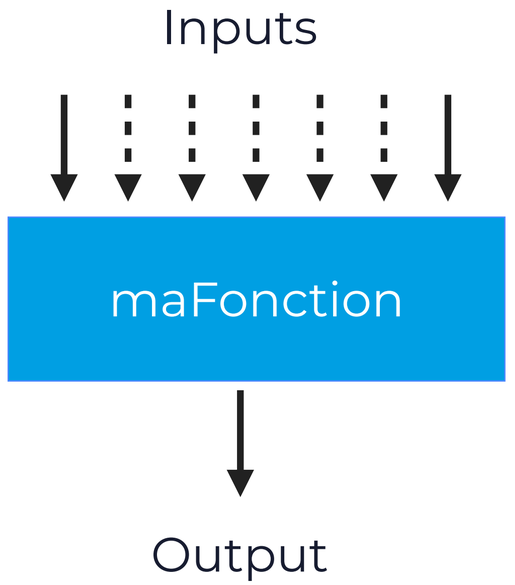
➡ On utilise la pile pour stocker les paramètres puis la fonction les récupère un par un dans la pile.

Paramètres de type Tableau ou Vecteurs

➡ On place l'adresse de départ (= pointeur) et le nombre de valeurs dans la pile puis la fonction les récupère dans la pile via une boucle.

Paramètre de type Structure regroupant plusieurs paramètres de taille différentes (ex: 32bits + 8bits)

➡ On place l'adresse de départ dans la pile et la fonction lit la structure dans la pile en tenant compte de l'organisation des données de la structure.



Exemple de fonction avec N entrées et une sortie

Fonctions et appel de fonctions

Retour de N résultats (N > 1)

Cas d'une seule sortie

➡ Le résultat est stocké dans R0 (= return value)

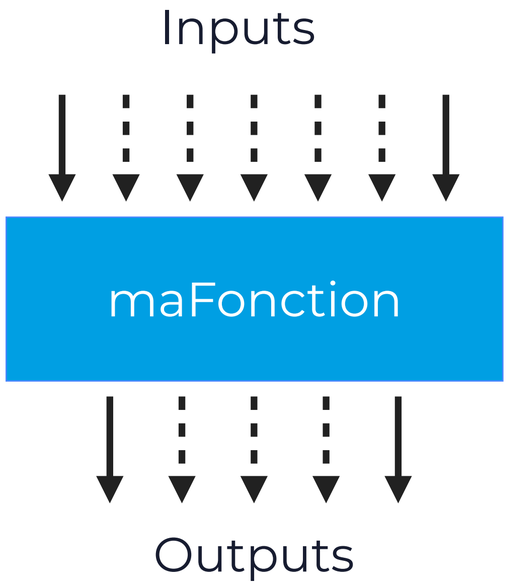
Cas de plusieurs sorties

➡ Le programme principal passe des adresses et plus des valeurs à la fonction (via les registres R0 à R3 ou la pile)

➡ La fonction utilise ces adresses pour stocker les résultats.

➡ Le programme principal retrouve les résultats en lisant les cases mémoires concernées

➡ Ce mécanisme fonctionne pour tout type de données



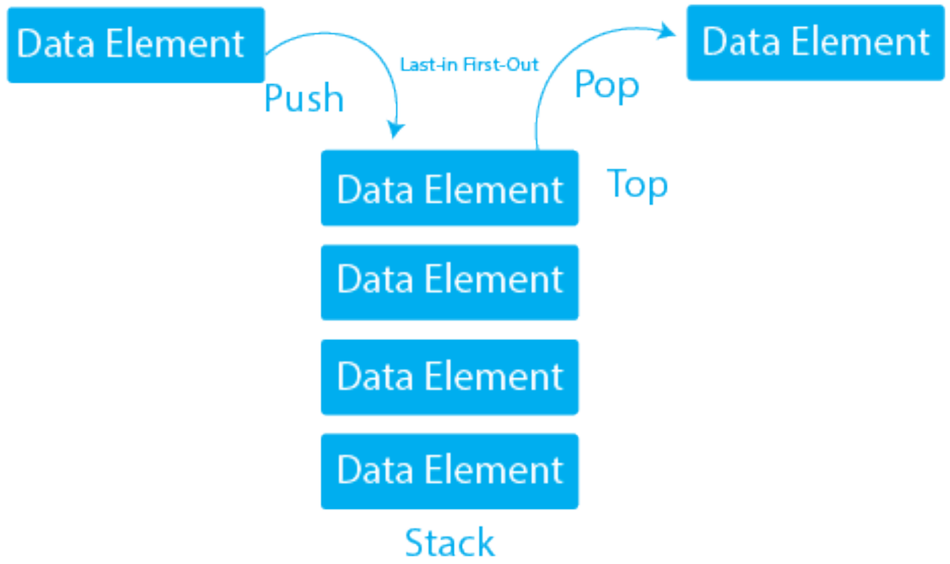
Exemple de fonction avec N entrées et M sorties

Pour terminer sur la pile

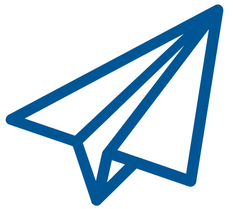
Toujours se servir de la pile !

La pile (ou stack) est très utilisée :

- ➡ Pour passer des valeurs ou des adresses comme paramètres d'une fonction.
- ➡ Pour renvoyer des résultats en retour de fonction.
- ➡ Pour sauvegarder/restaurer des registres utilisés dans une fonction ou avant/après l'appel d'une fonction.
- ➡ Pour créer des variables locales au sein d'une fonction.
- ➡ Pour sauvegarder le contexte du processeur (état courant des registres)) lors d'une interruption.
- ➡ ...



TAKE HOME MESSAGE



#3

Le processeur ARM7TDMI est un processeur **RISC 32 bits** reposant sur un jeu très simple d'instructions et un **pipeline à 3 étages** (Fetch, Decode, Execute)

Les instructions possibles permettent de réaliser des opérations **arithmétiques** et **logiques**, des branchements conditionnels ainsi que l'appel de fonctions via un mécanisme de **pile**.

Questions

Photo by Alexander Andrews on Unsplash

Contacts

Pr. Dominique Ginhac

dginhac@u-bourgogne.fr

Retrouvez toutes les infos sur :

https://github.com/dginhac/esirem-archi

POLYTECH DIJON