

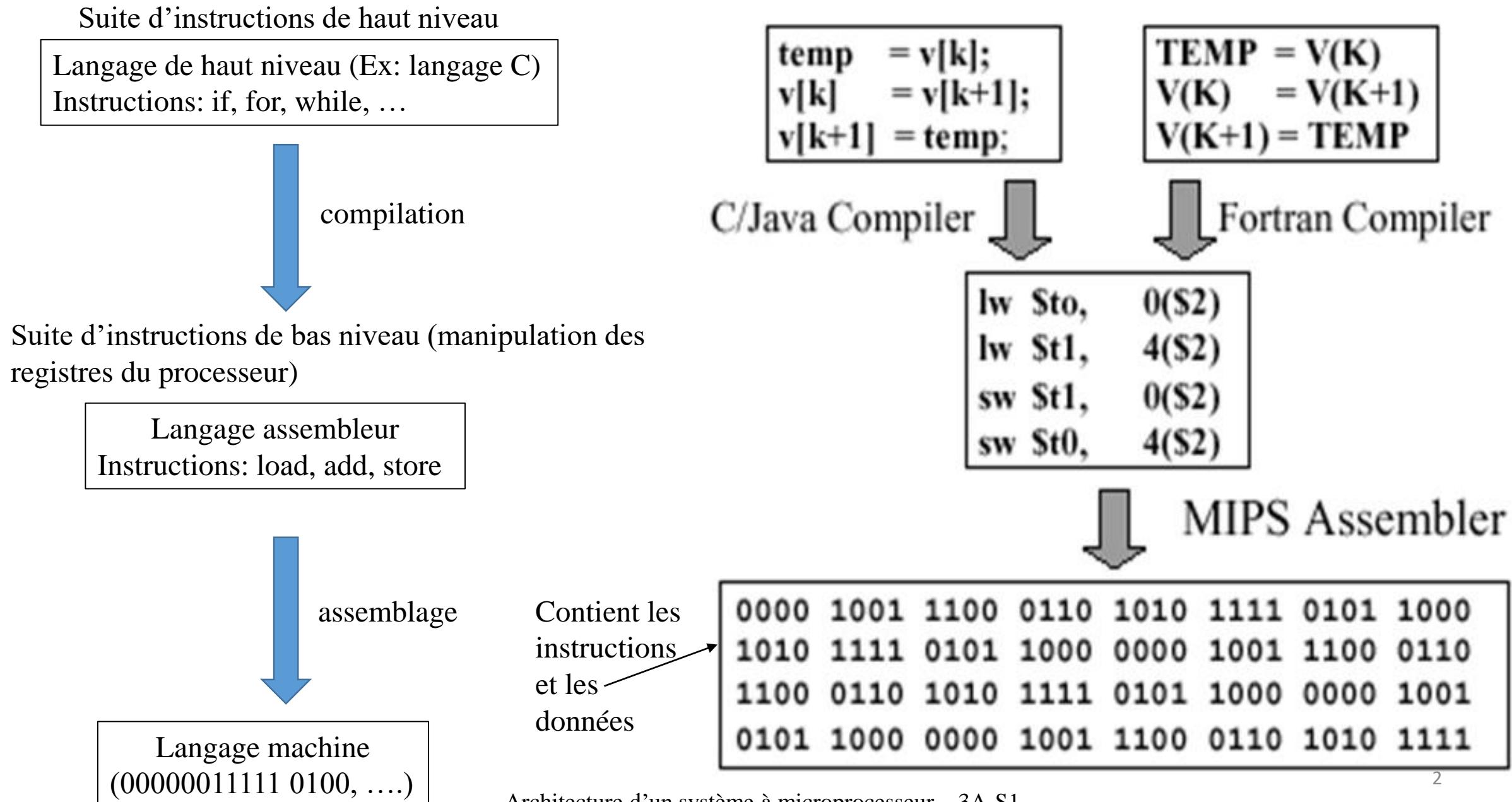
Introduction à la programmation assembleur

Exemple d'ARM Cortex-M0+

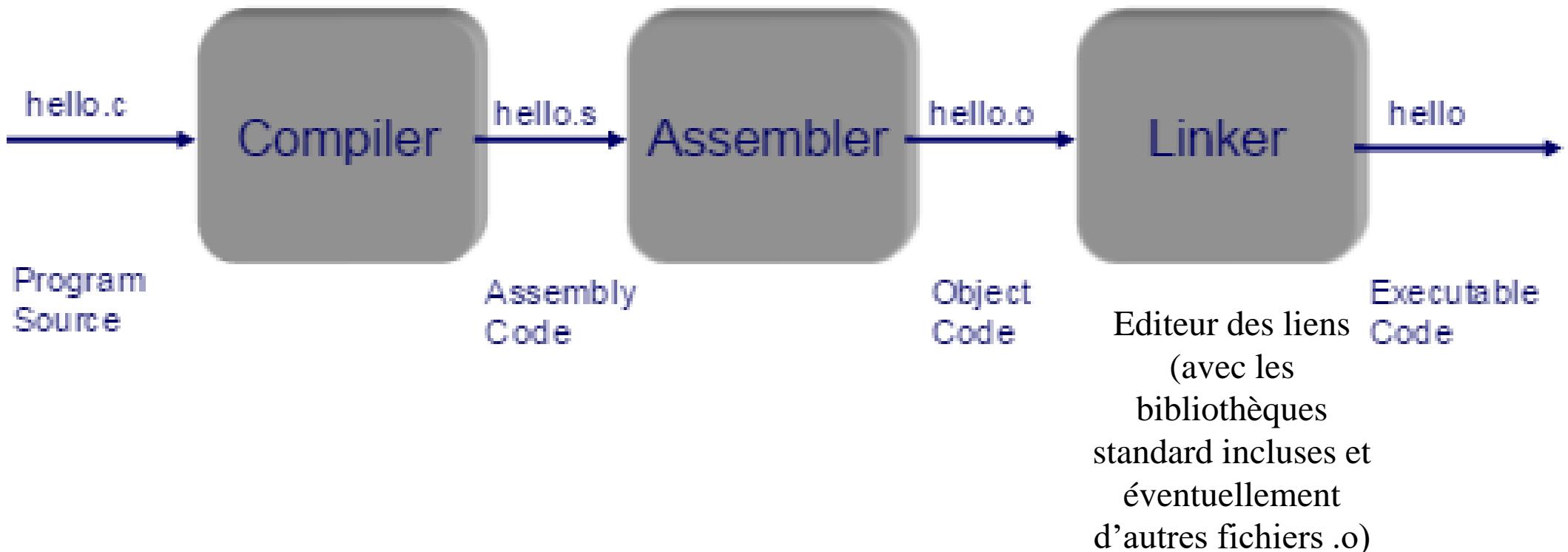
CHIRAZ TRABELSI

trabelsi@esiea.fr

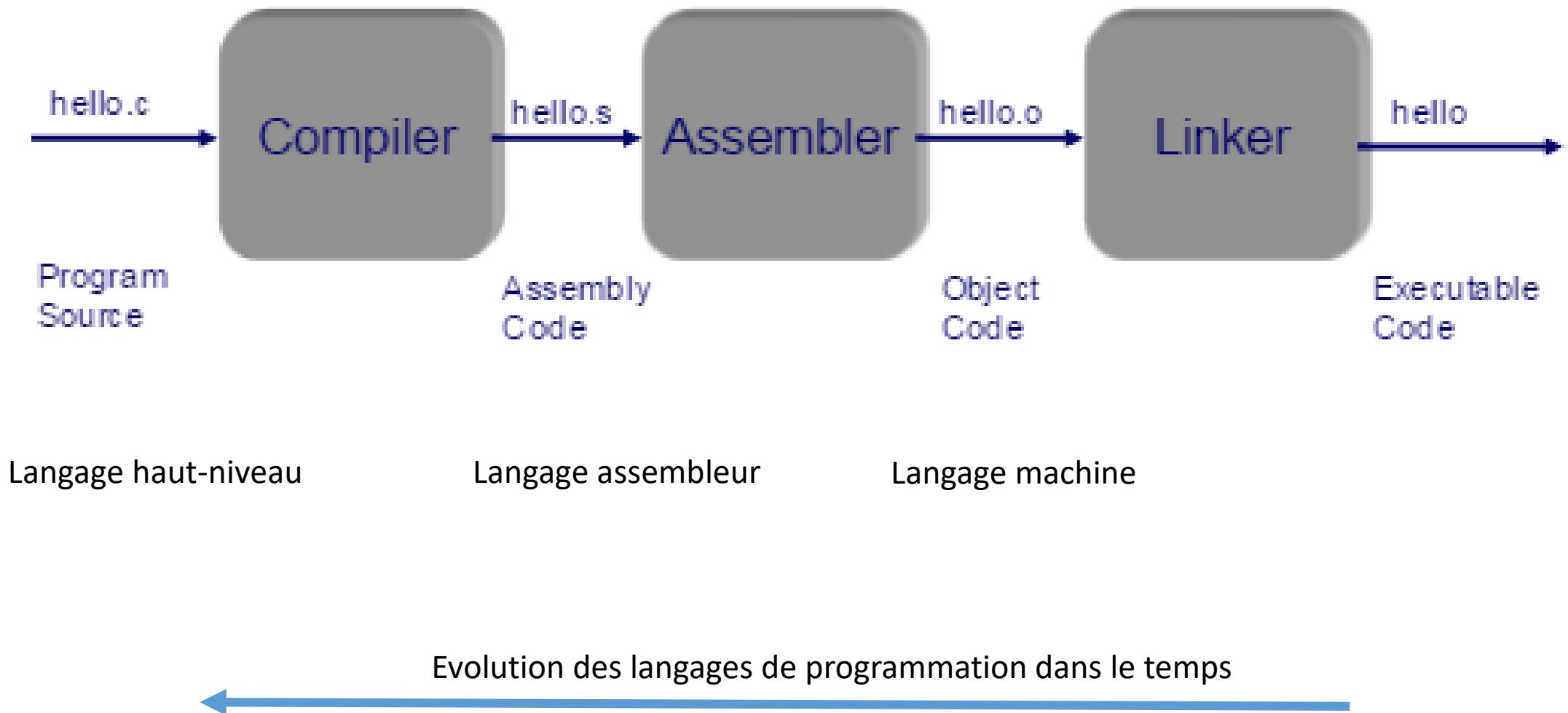
Du langage haut niveau au langage machine



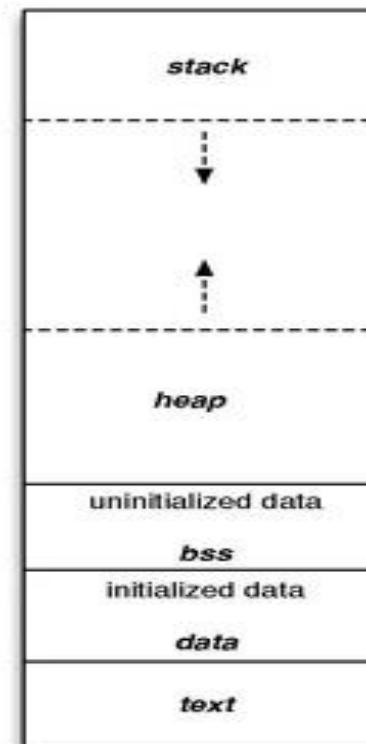
Du langage haut niveau au langage machine



```
gcc -S hello.c → hello.s //génération du code assembleur  
gcc -c hello.c → hello.o //génération du fichier objet  
gcc -o hello hello.o → hello (exécutable) //édition des liens  
gcc -o hello hello.c → hello (exécutable) //tout en une commande
```

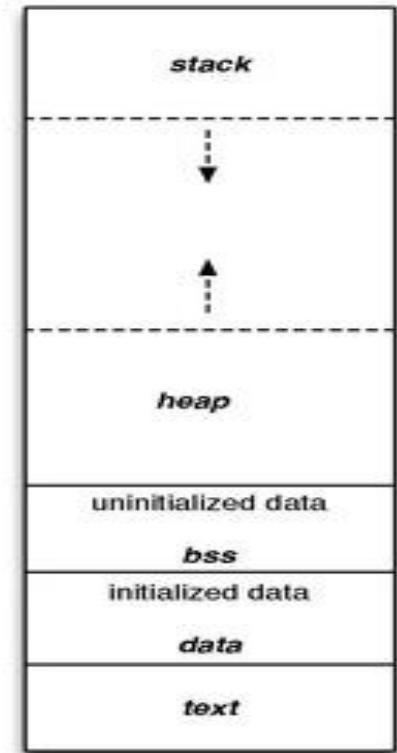


- Suite de 0 et de 1 représentant les instructions et les données d'un programme
- Un fichier objet .o contient du code machine
- Un fichier objet est organisé en différentes sections
- Lors de l'exécution du programme, chaque section sera chargée dans la mémoire RAM à un emplacement dédié



- Lors de l'exécution du programme, chaque section sera chargée dans la mémoire RAM à un emplacement dédié
 - La section text: contient les instructions
 - La section data: variables globales et variables statiques initialisées (exemple: val et string)
 - La section bss: variables globales et variables statiques non initialisées (exemple: i)
 - La section heap (tas): le tas binaire est géré par les instructions d'allocation dynamique (malloc, free, etc.)
 - La section stack (pile): la pile est un espace mémoire dans la mémoire principale utilisé essentiellement pour stocker des données temporaires, des variables locales (exemple: a) et des adresses de retour des fonctions

```
int val = 3;  
  
char string[] = "Hello World";  
  
static int i;  
  
int main(){  
int a;  
....  
}
```



- Suite d'instructions de bas niveau (manipulation des registres du processeurs, accès mémoire, etc.)
- Les ordinateurs ont pour la plupart le même jeu d'instructions
- Un jeu d'instructions: ensemble d'instructions élémentaires réalisées par le processeur
- Exemple: x86 (32 bits), x86-64 (64 bits)
- Si ce n'était pas le cas, les logiciels vendus comme fichiers exécutables seraient fournis en plusieurs versions selon le jeu d'instruction du processeur cible

- Pour les microcontrôleurs/processseurs spécifiques (consoles de jeux, machines à laver, matériel médical, etc.), les instructions sont différentes d'un processeur à un autre
 - Exemple : voici 4 implémentations assembleur différentes du chargement d'une valeur numérique dans un registre

Zilog Z80

→ LD A,72h

Motorola 68hc11

→ LDAA #\$72

Philips 8051

→ MOV A,#31h

Microchip PIC 16F84

→ MOVLW 0x31

- Les instructions assembleur qui ont la même fonctionnalité se ressemblent beaucoup entre processeurs
- C'est facile d'apprendre/comprendre l'assembleur d'un processeur si on a déjà travaillé avec de l'assembleur sur un autre processeur

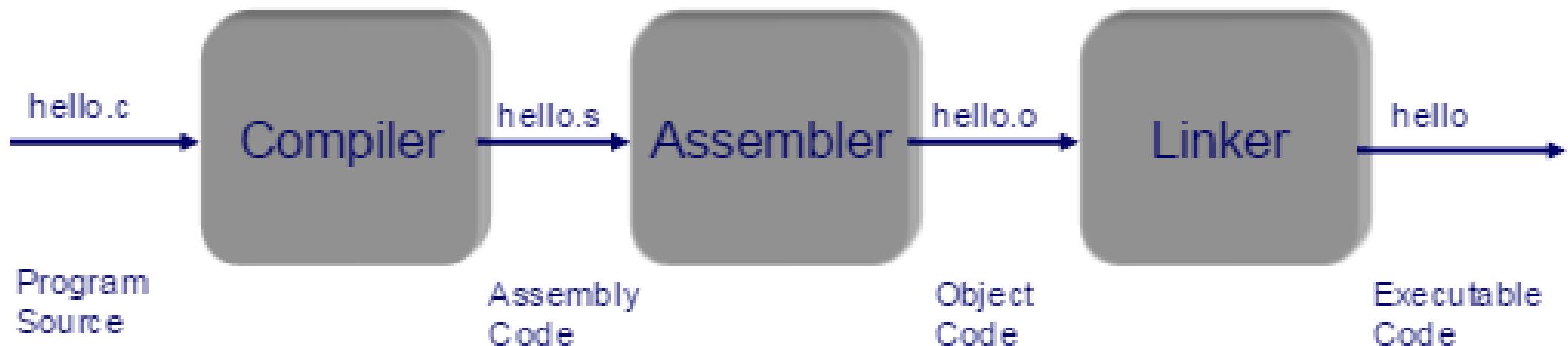
- Apparition
 - Les programmeurs utilisaient le langage machine (des 0 et des 1)
 - Les codes des opérations sont représentés en binaire
 - Les opérandes (registres, adresses mémoires, valeurs numériques) sont également représentés en binaire
 - ➔ Difficulté de programmation: il est difficile de retenir tous les codes binaires des opérations et des opérandes
- L'assembleur est apparu pour pallier ce problème en utilisant des symboles dits « mnémoniques » c'est-à-dire faciles à retenir
 - Exemple pour un processeur AVR: **add r16, r17** ($r16 \leftarrow r16 + r17$)

- Réécriture en langage haut-niveau d'un algorithme d'une manière plus efficace (plus de performance et moins de mémoire) en jetant un coup d'œil sur le code assembleur généré par le compilateur
- Faciliter le débogage de certains codes en accédant aux registres du processeur, mémoires et E/S
- Optimisation des performances pour un système temps-réel (équipement médical, système de navigation dans l'avionique, etc.) en écrivant certaines tâches en assembleur
- Exploitation des instructions bas-niveau offertes pour un processeur pour des programmes nécessitant une grande performance comme les jeux vidéos

- Reverse-engineering: reconstruire le code source à partir d'un exécutable (exemple: adaptation des anciens jeux vidéos à des plateformes modernes, étude d'un virus pour l'éradiquer, ...)
- Développement des systèmes d'exploitation
- Développement des drivers (pilotes) de périphériques
- Développement/mise à jour des compilateurs pour des logiciels sur ordinateur/microcontrôleurs/consoles de jeux
- Conception d'un nouveau processeur spécialisé

- Maîtriser les aspects bas niveau de l'informatique, en ayant conscience du processeur sur lequel on programme → écrire des programmes plus optimaux que ce soit en langage haut niveau (C, Java, Python, etc.) ou bien de bas niveau (assembleur)
- Les mêmes principes de base sont valables pour la programmation bas niveau sur un processeur d'un ordinateur ou celui d'un smartphone/tablette

- Exemples: C, C++, java, etc.
 - Avantages: faciliter et accélérer la programmation,
 - Ces langages permettent de cacher la complexité du processeur et d'utiliser des opérations d'une façon plus intuitive
 - C'est le compilateur et l'assembleur qui gèrent la traduction du langage haut niveau en langage machine



Exemple du microcontrôleur ARM Cortex-M0+

- Le processeur qu'on va prendre comme exemple pour étudier l'assembleur est le Cortex-M0+

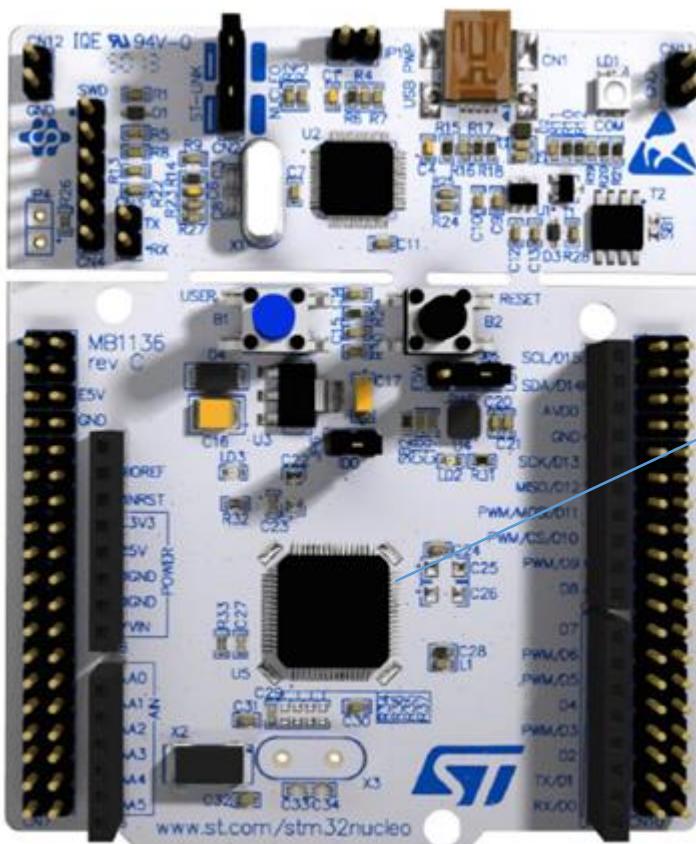
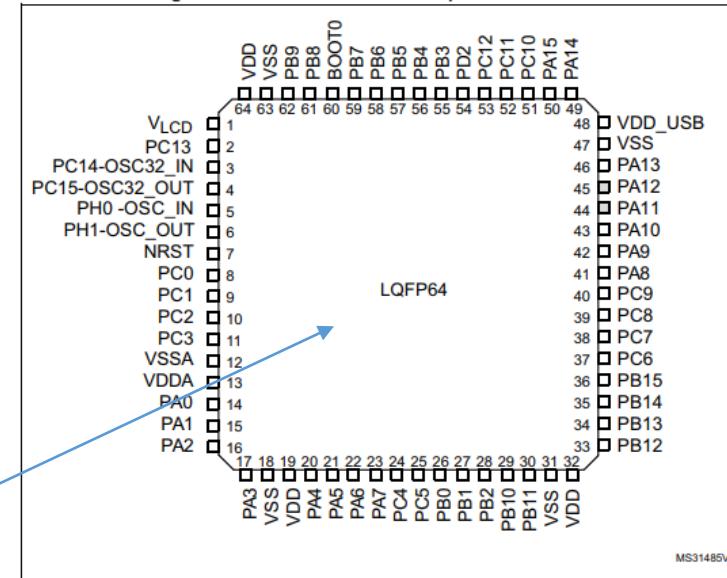


Figure 3. STM32L053x6/8 LQFP64 pinout - 10 x 10 mm



1. The above figure shows the package top view.
2. I/O pin supplied by VDD_USB.

La puce contient entre autres une CPU

- ARM: Advanced RISC Machines
 - ➔ des machines RISC de 32bits et de 64 bits
- ARM ne vend pas de circuits intégrés mais des architectures
- 3 familles d'ARM
 - Cortex-A: dédié aux applications, haute performance, supporte un OS (exemple: un Raspberry PI 3 est équipé d'un ARM Cortex-A53)
 - Cortex-R: Real-Time (automobile, avionique, médical, etc.)
 - Cortex-M: Microcontrôleur, basse consommation (exemple: Cortex-M0+)

- Pourquoi cette carte microcontrôleur en particulier?
 - Les cartes sur lesquels on va travailler présentent plusieurs intérêts par rapport aux cartes Arduino sur lesquelles vous avez pu travailler avant
 - Se familiariser avec l'architecture d'un processeur de la famille ARM qui équipe plusieurs dispositifs de nos jours (Raspberry Pi, smartphones tels que les téléphones Samsung et iPhone, etc.)
 - Plus de fréquence (32 MHz contre 16 MHz pour un ArduinoMega) → plus de performance de calcul
 - Processeur 32 bits (l'Arduino Mega contient un processeur de 8 bits)
 - Un espace mémoire assez intéressant (20 Kbytes de SRAM, jusqu'à 192 Kbytes de Flash contre 8K de SRAM et 256K de flash pour l'Arduino Mega)
 - Prix moins cher (\approx 12 €, contre \approx 35 € pour un Mega)

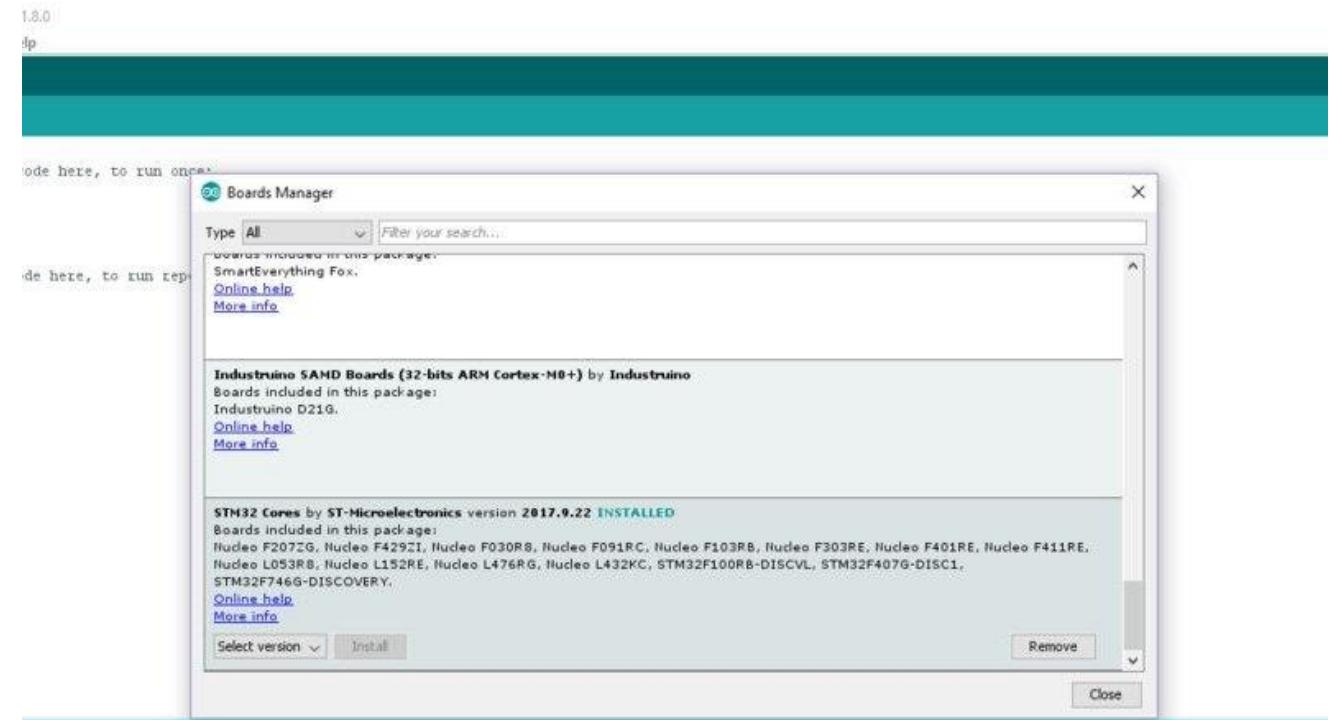
- Pourquoi cette carte microcontrôleur en particulier?
 - Une basse consommation (quelques dizaines de mW, contre quelques centaines pour un Mega)
 - Un nombre assez important de pins (52 pins numériques dont 6 qui peuvent être configurées en analogique), contre, 56 pins numériques et 16 analogiques pour un Mega)
 - Débogage: un module intégré dans la carte

- Possibilités de programmation

1) En Arduino

- Niveau débutant

- Avantages: Facilité de la prise en main de l'environnement de développement en utilisant des librairies déjà existantes
 - Inconvénients:
 - L'architecture interne du microcontrôleur est invisible au développeur
- Difficultés dans la détection des bugs



<https://www.instructables.com/id/Quick-Start-to-STM-Nucleo-on-Arduino-IDE/>

- Possibilités de programmation

2) En langage C et en Assembleur

- Niveau avancé

- Avantages: Une maîtrise de l'architecture du microcontrôleur

→ Plus de possibilités d'optimisation du code
 → Plus de facilités dans la détection des bugs

- Inconvénients:

- Un certain temps d'apprentissage pour maîtriser l'architecture

The screenshot shows the Eclipse IDE interface during a debugging session. The top menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The left sidebar shows the project structure under 'InterruptBP.elf [Ac6 STM32 Debugging]'. The main workspace displays several views:

- Debug**: Shows Thread #1 (Suspended : Signal : SIGINT:Interrupt) with stack trace: LL_SYSTICK_EnableIT() at stm32l0xx_ll_cortex.h:281 0x80004a8.
- Variables**: Shows General Registers: r0=0x1 (Hex), r1=0x5 (Hex), r2=0xe000e010 (Hex), r3=0x7 (Hex).
- Memory**: Shows a memory dump from 0x00000000 to 0x000000A0, with the first few bytes being 00 20 00 00.
- Disassembly**: Shows assembly code for the SysTick_Handler, starting with b.n 0x80004a8 <main>.

 The bottom status bar indicates 'Warning: the current language does not match this frame.' and shows temporary breakpoints at main() and line 67.

- C'est le fait de compiler un programme sur une architecture qui n'est pas celle de l'architecture cible.
- Le microcontrôleur a un jeu d'instruction différent de celui de l'ordinateur.
- Lorsqu'on compile le code qu'on écrit dans un IDE (Integrated Development Environment) Arduino ou STM32, on utilise un compilateur qui fonctionne sur le processeur d'ordinateur mais qui génère un fichier exécutable pour une architecture cible différente

- Ces informations sont disponibles dans **STM32L0 Series Cortex®-M0+ programming manual**

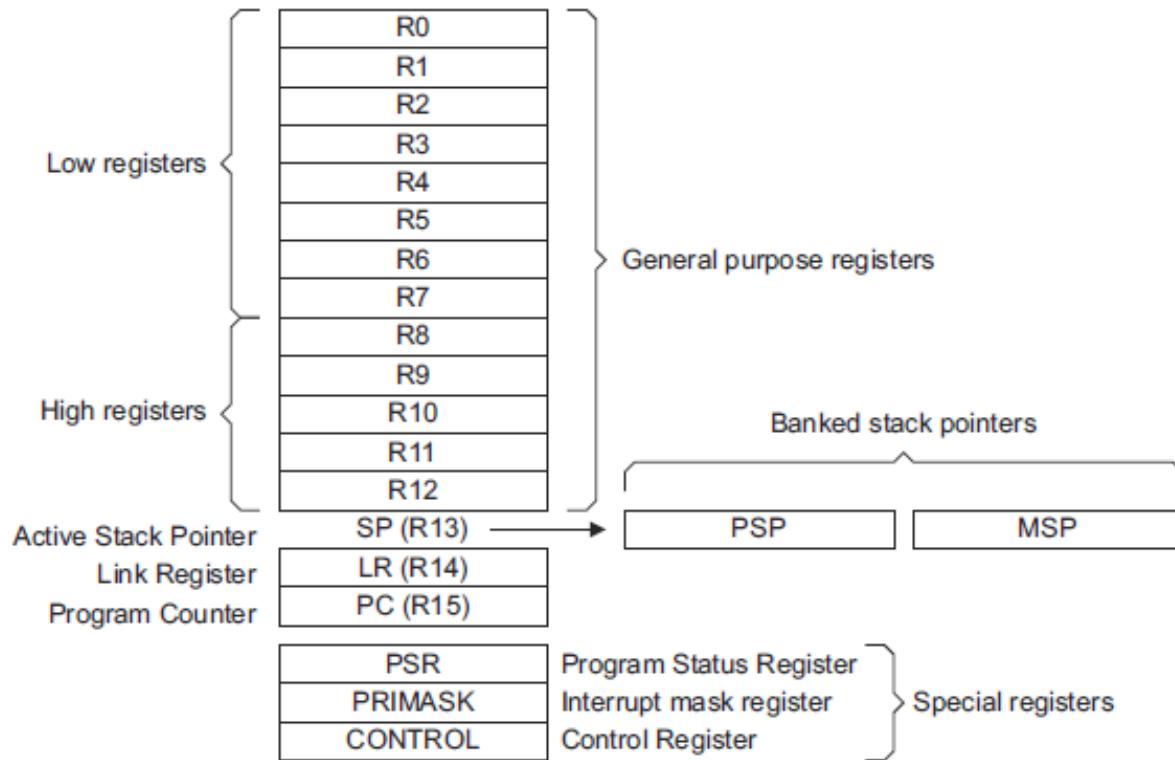
https://www.st.com/resource/en/programming_manual/dm00104451.pdf

L'architecture et les accès mémoire

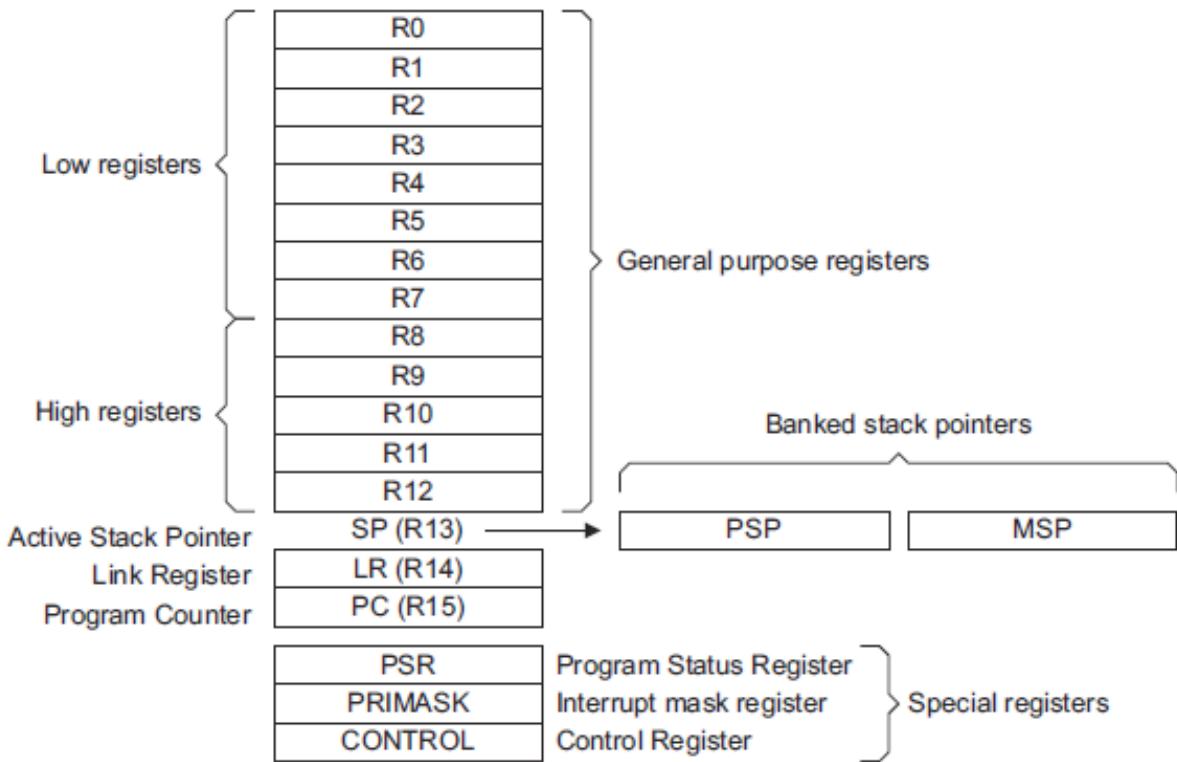
2	The STM32L0 Cortex-M0+ Processor	12
2.1	Programmers model	12
2.1.1	Processor modes and privilege levels for software execution	12
2.1.2	Stacks	12
2.1.3	Core registers	13
2.1.4	Exceptions and interrupts	18
2.1.5	Data types	19
2.1.6	The Cortex Microcontroller Software Interface Standard	19
2.2	Memory model	20
2.2.1	Memory regions, types and attributes	21
2.2.2	Memory system ordering of memory accesses	21
2.2.3	Behavior of memory accesses	22
2.2.4	Additional memory access constraints for caches and shared memory	23
2.2.5	Software ordering of memory accesses	23
2.2.6	Memory endianness	24

Le jeu d'instructions

3	The STM32L0 Cortex-M0+ instruction set	36
3.1	Instruction set summary	36
3.2	Intrinsic functions	39
3.3	About the instruction descriptions	40
3.3.1	Operands	40
3.3.2	Restrictions when using PC or SP	40
3.3.3	Shift operations	40
3.3.4	Address alignment	43
3.3.5	PC-relative expressions	43
3.3.6	Conditional execution	43
3.4	Memory access instructions	45
3.4.1	ADR	46
3.4.2	LDR and STR, immediate offset	47
3.4.3	LDR and STR, register offset	48
3.4.4	LDR, PC-relative	49
3.4.5	LDM and STM	50
3.4.6	PUSH and POP	52
3.5	General data processing instructions	53
3.5.1	ADC, ADD, RSB, SBC, and SUB	54
3.5.2	AND, ORR, EOR, and BIC	56
3.5.3	ASR, LSL, LSR, and ROR	57



- R0-R12: Registres génériques
- Stack Pointer: Pointeur de pile
- LR (=R14)
 - Link Register
 - Utilisé pour stocker les adresses de retour des fonctions

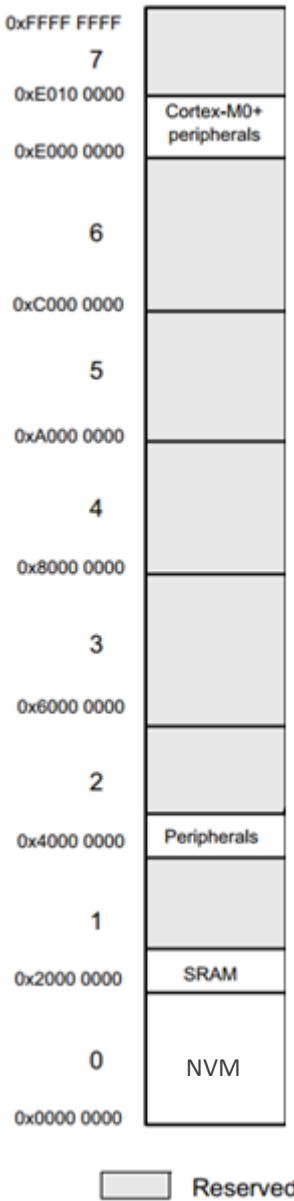


- **PC**
 - Program Counter
 - Il contient l'adresse 0x00000004 après un reset
→ la première instruction à exécuter se trouve à l'adresse 0x00000004

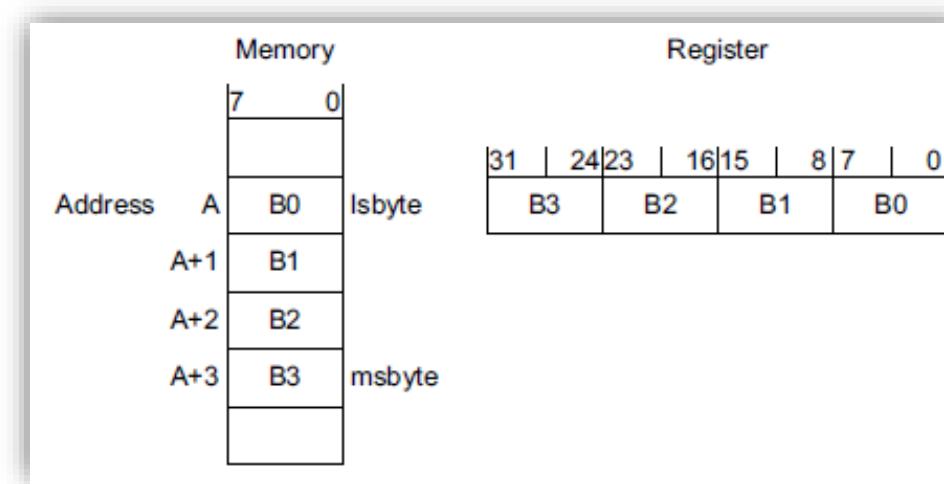
- **PSR**
 - Program Status Register: combinaison de trois registres: APSR, IPSR et EPSR
 - APSR: Application Program Status Register

Bits	No m	Function
[31]	N	Negatif
[30]	Z	Zero
[29]	C	Carry
[28]	V	Overflow
[27:0]	-	Réservé

- IPSR: Interrupt Program Status Register
- EPSR: Execution Program Status Register

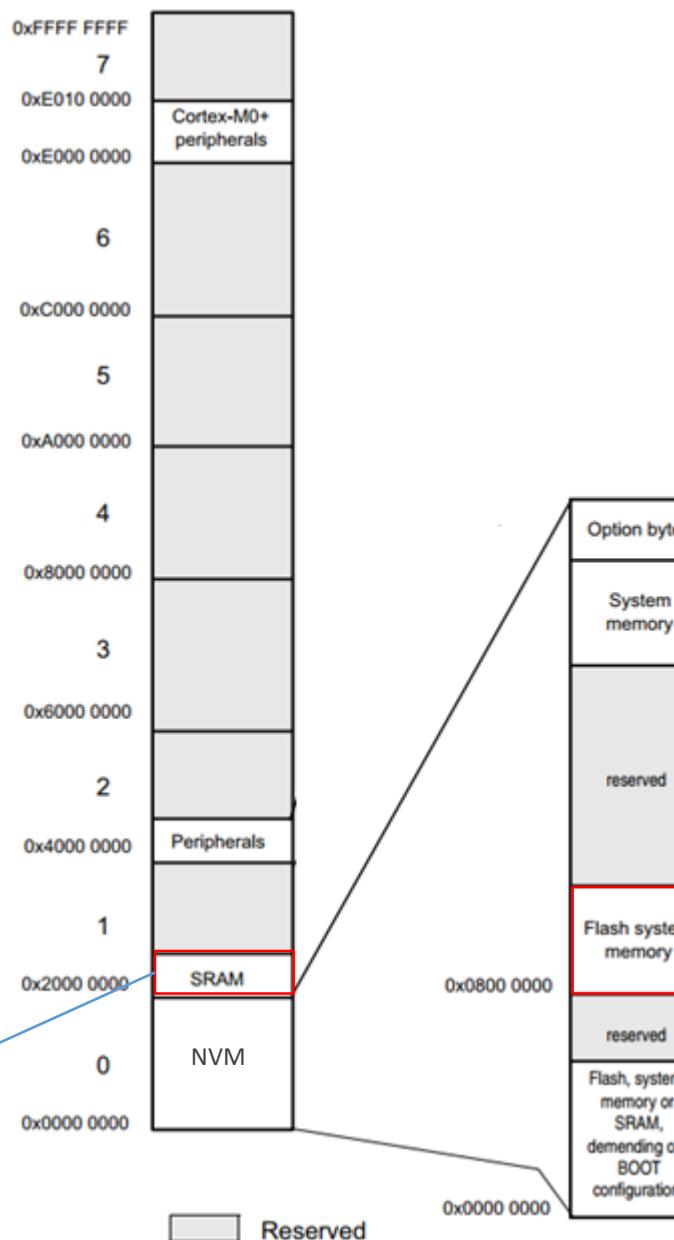


- Un espace d'adressage de $2^{32} = 4\text{Go}$
- L'espace d'adressage est divisé sur 8 blocs de 512Mo
- Les octets sont codés en Little-endian → l'octet de poids faible dans un mot a l'adresse la plus petite



Mémoire de données

Mémoire de programme



- **data**
 - Définit le début de la section données
- **text**
 - Définit le début de la section code (programme)

```
.data      // début de la section de données
var1: .space 4    //réserver 4 octets dans la SRAM pour var1
var2: .word 6     /* réservé 4 octets dans la SRAM pour var2
                    et l'initialiser à 6*/
.text      // début de la section de code
movs r0,#5
movs r1,#6
.....
```

- **Les commentaires**
 - Trois formes possibles de commentaires

```
movs r0,#5 @charger 5 dans r0
mov r0,r1 //copier r1 dans r0
mov r1,r0 /*copier r0 dans r1*/
```

- **EQU et SET**
 - Affecte une valeur à une étiquette (c'est l'équivalent de #define dans le langage C)
 - La valeur ne peut pas changer par la suite dans le programme

```
.EQU io_offset, 0x23
.SET io_offset, 0x23
```

- **Byte, word, hword, space**
 - Réservent des octets dans la mémoire SRAM

```
.data // début de la section de données
var1: .byte 3 //var1 s'étend sur un octet, sa valeur initiale est 3
var2: .word 5 //var2 s'étend sur 4 octets, sa valeur initiale est 5
var3: .space 3 // réservé 3 octets dans la mémoire
Var4: .hword 6 //var4 s'étend sur 2 octets (half word), sa valeur initiale est 6
```

- Arithmétiques/logiques
 - Addition, soustraction, multiplication, AND, OR, décalage, etc.
- Chargement / rangement
 - LDR (load): permet de charger une donnée de la mémoire dans un registre
 - STR (Store): permet de stocker le contenu d'un registre dans la mémoire principale
- Contrôle
 - Branchement conditionnel/inconditionnel, appel de fonction

- Les instructions sur 16 bits

16-bit Thumb instructions supported on Cortex-M0/M0+ processors									
ADC	ADD	ADR	AND	ASR	B	BIC	BLX	BKPT	BX
CMN	CMP	CPS	EOR	LDM	LDR	LDRH	LDRSH	LDRB	LDRSB
LSL	LSR	MOV	MVN	MUL	NOP	ORR	POP	PUSH	REV
REV16	REVSH	ROR	RSB	SBC	SEV	STM	STR	STRH	STRB
SUB	SVC	SXTB	SXTH	TST	UXTB	UXTH	WFE	WFI	YIELD

- Les instructions sur 32 bits

32-bit Thumb instructions supported on Cortex-M0/M0+ processors					
BL	DSB	DMB	ISB	MRS	MSR

Les instructions ARM sont toutes sur 32 bits à la base, mais avec la version Thumb (c'est la version que le Cortex-M0+ supporte), on a des instructions sur 16 bits qui font la même chose que leurs équivalentes sur 32 bits. Cela permet de réduire la taille de la mémoire et le coût du système.

Format Summary

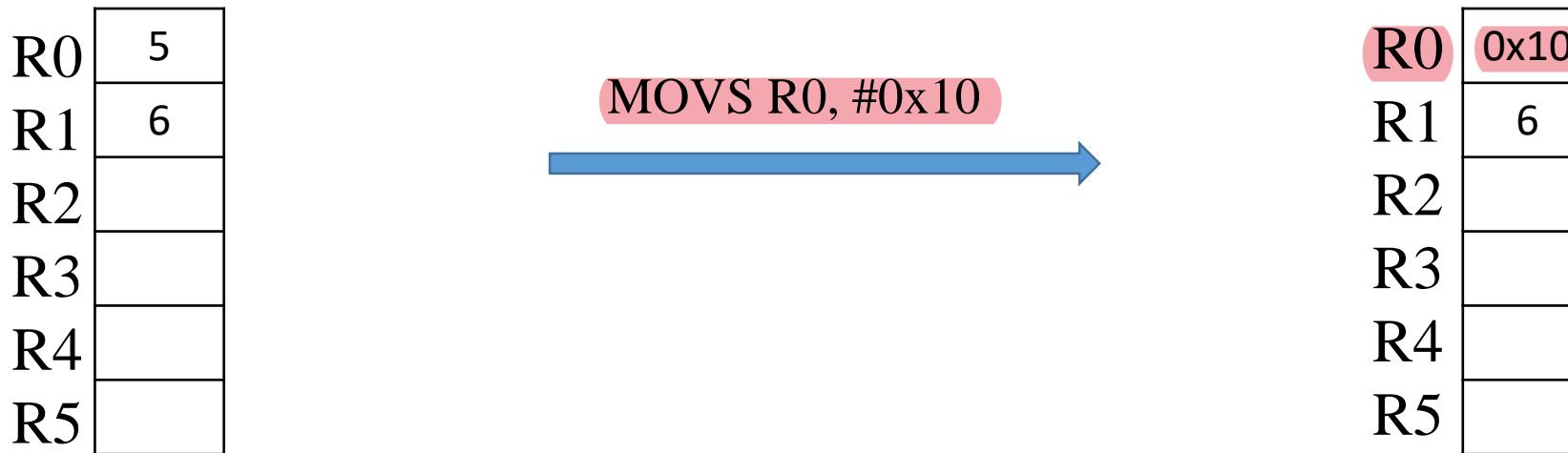
The THUMB instruction set formats are shown in the following figure.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Op												
2	0	0	0	1	1	I	Op	Rn/offset3								
3	0	0	1	Op			Rd									
4	0	1	0	0	0	0		Op								
5	0	1	0	0	0	1	Op	H1	H2							
6	0	1	0	0	1		Rd									
7	0	1	0	1	L	B	0		Ro							
8	0	1	0	1	H	S	1		Ro							
9	0	1	1	B	L			Offset5								
10	1	0	0	0	L			Offset5								
11	1	0	0	1	L		Rd									
12	1	0	1	0	SP		Rd									
13	1	0	1	1	0	0	0	0	S							
14	1	0	1	1	L	1	0	R								
15	1	1	0	0	L		Rb									
16	1	1	0	1			Cond									
17	1	1	0	1	1	1	1	1								
18	1	1	1	0	0				Offset11							
19	1	1	1	1	H				Offset							

- Les instructions ont une taille de 16 bits
- En regardant les bits de gauche à droite, on peut déterminer le type de l'opération (c'est la phase de décodage de l'instruction)
- Exemple: si 3 bits de poids forts ont une valeur 000, l'opération peut être soit du premier soit du deuxième type (move shifted register ou add/subtract)
- Op désigne le code de l'opération. Il peut être écrit soit sur 1 bit (deuxième type), 2 bits (1^{er} type) ou 4 bits (4^{ème} type)
- Offset: valeur immédiate
- Rs: Registre source
- Rd: registre destination
- I:Immediate flag
 - 0 - Register operand
 - 1 - Immediate operand

- Selon le type des opérandes (valeur immédiate, adresse mémoire, registre, le compteur ordinal), il y a quatre principaux types d'adressage
 - Adressage immédiat: utilisation d'opérande immédiat (une valeur numérique)
 - Adressage direct: l'instruction contient directement l'adresse de la case mémoire à laquelle on veut accéder
 - Adressage indirect: l'adresse mémoire à laquelle on veut accéder est contenue dans un registre, on y accède indirectement en référençant ce registre
 - Adresse relatif au compteur ordinal: l'adresse à laquelle on veut accéder se trouve dans la mémoire de programme à un certain décalage par rapport à l'instruction en cours d'exécution

- La valeur de la donnée à utiliser est contenue dans l'instruction.
- Ce type d'adressage est appelé immédiat car la valeur de la donnée à traiter est disponible au moment du décodage de l'instruction et on n'a pas besoin d'un accès mémoire/registre pour la récupérer
- MOVS R0, #0x10 //charger la valeur immédiate 0x10 dans le registre R0
- Remarque: les opérandes numériques sont précédés par #



- L'instruction contient directement l'adresse de la case mémoire à laquelle on veut accéder
 - Exemple: dans l'AVR 8-bits (Arduino 8 bits): L'instruction **LDS R16, 0x0300** charge dans le registre R16 le contenu de la case mémoire d'adresse 0x0300

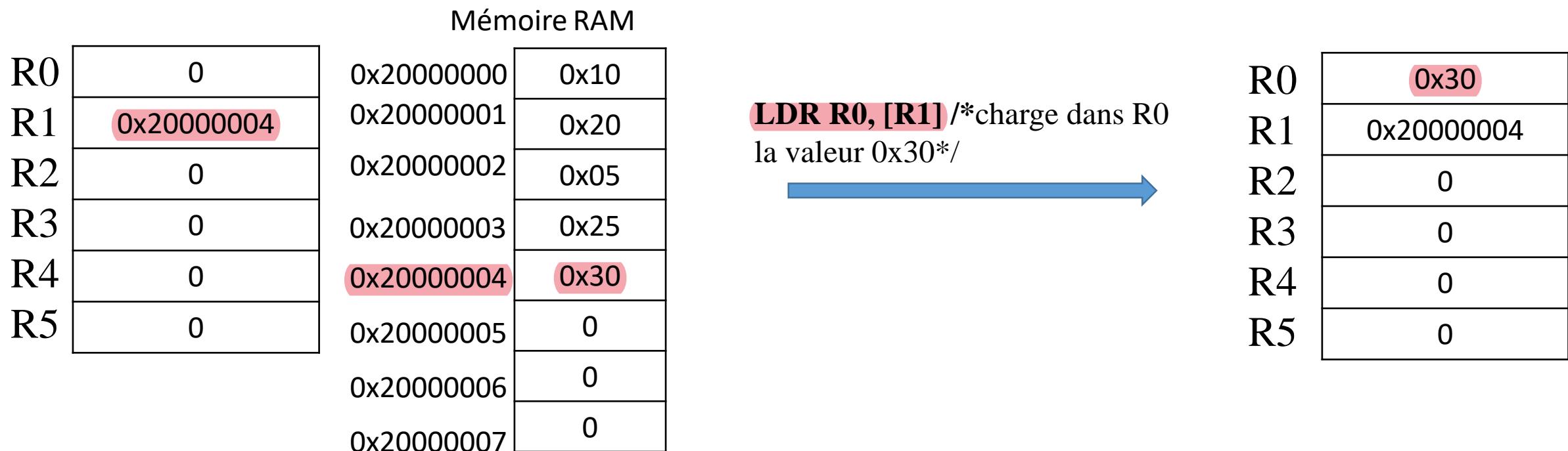
LDS R16, 0x0300 /*charger
dans R16 la valeur 98*/

Mémoire RAM

0x02FE	4
0x02FF	5
0x0300	98
0x0301	175
0x0302	36

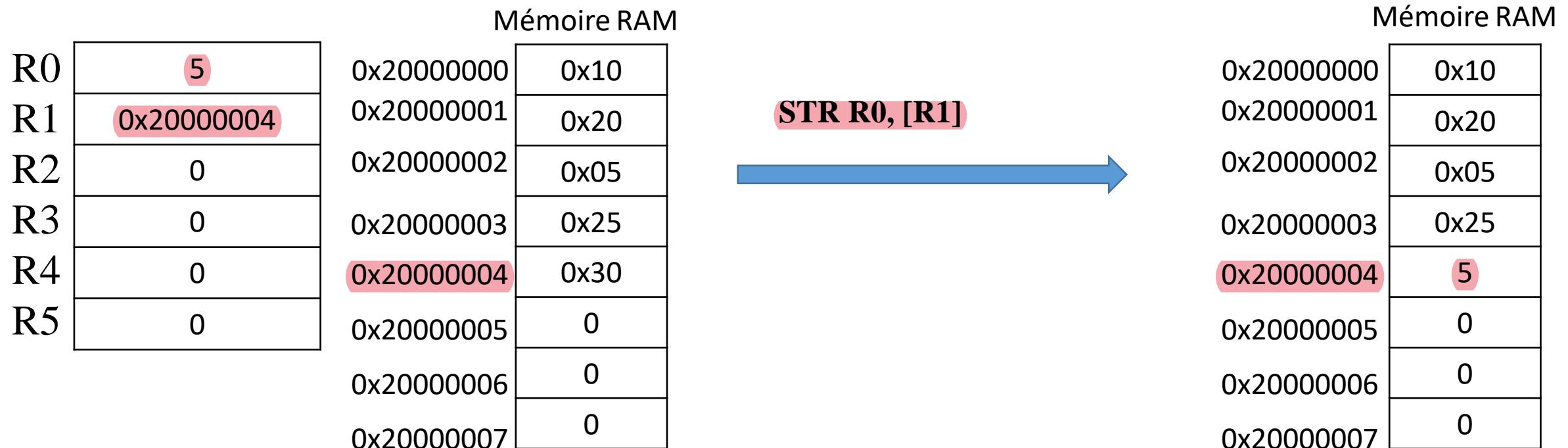
- Exemple Intel x86
 - **MOV AX, adresse1;** charger dans le registre AX le contenu de la case mémoire à l'adresse 1523
 -
 - **Adresse1 DW 1523;** définie dans la section des données
- **L'adressage direct n'est pas disponible pour le processeur ARM Cortex-M0+ car les adresses s'écrivent sur 32-bits et ce n'est pas possible donc de les écrire dans des instructions de 32-bits ou de 16-bits**

- L'adresse mémoire à laquelle on veut accéder est contenu dans un registre, on y accède indirectement en référençant ce registre
 - Exemple: L'instruction **LDR R0, [R1]** charge dans le registre R0 le contenu de la case mémoire dont l'adresse est contenu dans le registre R1



L'adressage indirect

- Exemple: L'instruction **STR R0, [R1]** charge le contenu du registre R0 dans la case mémoire dont l'adresse est contenu dans le registre R1



- Variantes
 - **LDR Rd, [Rn/SP]** → charger dans le registre Rd, le **mot** (4 octets) situé à l'adresse contenue dans Rn ou SP
 - **LDRH Rd, [Rn]** → charger dans le registre Rd, le **demi-mot** (2 octets) situé à l'adresse contenue dans Rn
 - **LDRB Rd, [Rn]** charger dans le registre Rd, l'**octet** situé à l'adresse contenue dans Rn
 - **STR, STRH et STRB** suivent le même principe
- Restrictions
 - Les registres Rd et Rn doivent être **R0 à R7**
 - Les adresses contenues dans le registre Rn doivent être alignées
 - LDR/STR: l'adresse doit être divisible par 4
 - LDRH/STRH: l'adresse doit être divisible par 2
 - Le registre SP n'est pas utilisable pour les instructions LDRH, LDRB, STRH et STRB

- Exercice
 - Sachant que l'état actuel des registres et de la mémoire est comme suit:

R0	0x20000000	0x20000000	0x10
R1	0	0x20000001	0x20
R2	0	0x20000002	0x05
R3	0	0x20000003	0x25
		0x20000004	0x30

Mémoire RAM

- Donner le résultat obtenu après l'exécution du code suivant:

LDR R1, [R0]

LDRH R2, [R0]

LDRB R3, [R0]

Attention: Chaque adresse dans le schéma de la mémoire RAM correspond à une case mémoire d'1 octet

- Solution
 - Sachant que l'état actuel des registres et de la mémoire est comme suit:

R0	0x20000000	0x20000000	0x10
R1	0	0x20000001	0x20
R2	0	0x20000002	0x05
R3	0	0x20000003	0x25
		0x20000004	0x30

Mémoire RAM

- Donner le résultat obtenu après l'exécution du code suivant:

**LDR R1, [R0]
LDRH R2, [R0]
LDRB R3, [R0]**



R0	0x20000000
R1	0x25052010
R2	0x2010
R3	0x10

- Exercice
 - Sachant que l'état actuel des registres et de la mémoire est comme suit:

R0	0x20000000
R1	0x11
R2	0x20000003
R3	0

0x20000000	0x10
0x20000001	0x20
0x20000002	0x05
0x20000003	0x25
0x20000004	0x30

Mémoire RAM

- Donner le résultat obtenu après l'exécution du code suivant:

STRH R1, [R0]

STRH R1, [R2]

- Solution
 - Sachant que l'état actuel des registres et de la mémoire est comme suit:

R0	0x20000000
R1	0x11
R2	0x20000003
R3	0

0x20000000	0x10
0x20000001	0x20
0x20000002	0x05
0x20000003	0x25
0x20000004	0x30

Mémoire RAM

- Donner le résultat obtenu après l'exécution du code suivant:

STRH R1, [R0] → OK

STRH R1, [R2] → Erreur d'exécution



0x20000000	0x11
0x20000001	0x00
0x20000002	0x05
0x20000003	0x25
0x20000004	0x30

Mémoire RAM

- Dans ce mode d'adressage, l'adresse de la donnée est donné par le contenu d'un registre + un décalage
- Le décalage peut être soit une valeur immédiate, soit contenue dans l'un des registres R0 à R7
 - Exemple:
 - L'instruction **LDR R0, [R1,#4]** charge dans le registre R0 le contenu de la case mémoire dont l'adresse est donnée par le contenu du registre R1+ un décalage de 4
 - L'instruction **LDR R3, [R1, R2]** charge dans le registre R3 le contenu de la case mémoire dont l'adresse est donnée par le contenu du registre R1+ un décalage contenu dans R2

R0	0
R1	0x20000000
R2	4
R3	0
R4	0
R5	0

Mémoire RAM

0x20000004	1
0x20000005	2
0x20000006	3
0x20000007	4
0x20000008	5

**LDR R0, [R1,#4]
LDR R3, [R1,R2]**



R0	0x04030201
R1	0x20000000
R2	4
R3	0x04030201
R4	0
R5	0

- Variantes

LDR *Rt*, [*Rn* | SP] {, #*imm*}]

LDR<B|H> *Rt*, [*Rn* {, #*imm*}]

STR *Rt*, [*Rn* | SP], {, #*imm*}]

STR<B|H> *Rt*, [*Rn* {, #*imm*}]

LDR *Rt*, [*Rn*, *Rm*]]

LDR<B|H> *Rt*, [*Rn*, *Rm*]]

LDR<SB|SH> *Rt*, [*Rn*, *Rm*]]

STR *Rt*, [*Rn*, *Rm*]]

STR<B|H> *Rt*, [*Rn*, *Rm*]]

Restrictions

In these instructions:

- *Rt* and *Rn* must only specify R0-R7.
- *imm* must be between:
 - 0 and 1020 and an integer multiple of four for LDR and STR using SP as the base register.
 - 0 and 124 and an integer multiple of four for LDR and STR using R0-R7 as the base register.
 - 0 and 62 and an integer multiple of two for LDRH and STRH.
 - 0 and 31 for LDRB and STRB.
- The computed address must be divisible by the number of bytes in the transaction, see [3.3.4: Address alignment on page 43](#).

Source: STM32L0 Series Cortex®-M0+ programming manual

Remarques:

Le registre SP n'est pas utilisable quand le décalage est contenu dans un registre

Les instructions LDRSB et LDRSH permettent de charger des données signées (extension de signes).

Ces instructions ne sont utilisables que dans le mode d'adressage indirect avec un décalage dans un registre

- Exemples de l'instruction LDRSH:

R0	0
R1	0x20000000
R2	0
R3	0
R4	0
R5	0

0x20000000	0x10
0x20000001	0xa0
0x20000002	0x05
0x20000003	0x25
0x20000004	0x30

Mémoire RAM

- LDRSH R0, [R1,R0]** → OK
LDRSH R0, [R1] → Erreur de syntaxe
LDRSH R0, [R1,#0] → Erreur de syntaxe



R0	0xffffa010
R1	0x20000000
R2	0
R3	0
R4	0
R5	0

- L'adresse à laquelle on veut accéder se trouve dans la mémoire de programme à un certain décalage par rapport à l'instruction en cours d'exécution
- Deux formes:
 - LDR Rt, [PC, #offset]
 - LDR Rt, label
- Dans le deuxième cas, l'assembleur va transformer l'instruction sous la première forme (décalage par rapport au compteur ordinal)

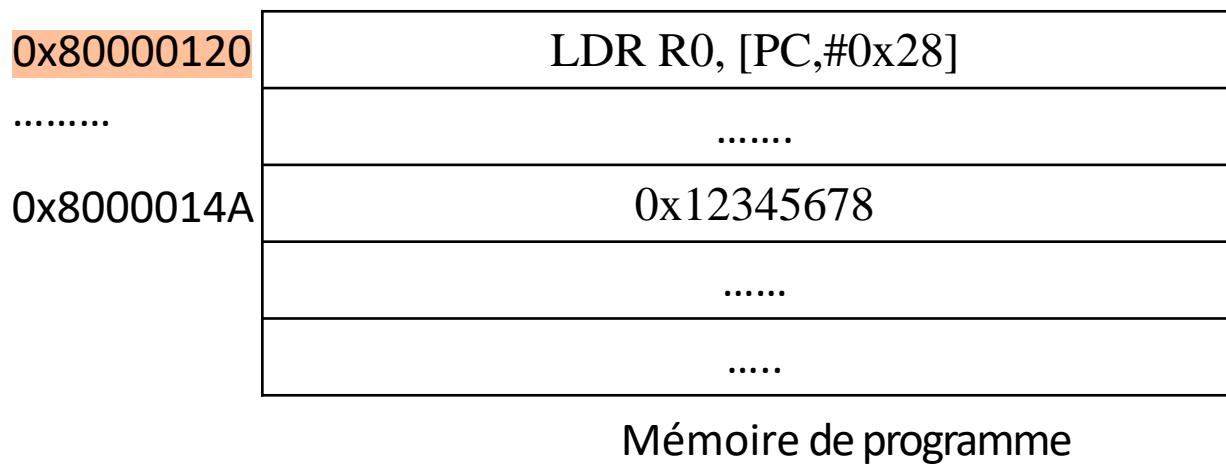
Exemples:

```
LDR R0, LookUpTable ; Load R0 with a word of data from an address
                     ; labelled as LookUpTable.
LDR R3, [PC, #100]  ; Load R3 with memory word at (PC + 100).
```

Restrictions

In these instructions, `label` must be within 1020 bytes of the current PC and word aligned.

- Utilisations
 - 1) Charger une valeur numérique de 4 octets dans un registre
 - MOVS R0, #0x12345678 → erreur de syntaxe (la valeur numérique pour MOVS doit être entre 0 et 255)
 - LDR R0, = 0x12345678 → OK (l'assembleur va stocker la valeur dans une case mémoire proche de celle de l'instruction courante et va transformer l'instruction sous la forme LDR R0, [pc, #offset])



3.5.5 MOV and MVN

Move and Move NOT.

Syntax

`MOV[S] Rd, Rm`

`MOVS Rd, #imm`

`MVNS Rd, Rm`

where:

S Is an optional suffix. If *S* is specified, the condition code flags are updated on the result of the operation, see [3.3.6: Conditional execution on page 43](#).

Rd Is the destination register.

Rm Is a register.

Imm Is any value in the range 0-255.

$$0x8000014A = \underbrace{0x80000120 + 2}_{\text{PC}} + \underbrace{0x28}_{\text{décalage}}$$

Attention: les instructions MOV et LDR sont des instructions sur 16 bits

- Utilisations

- Charger une valeur négative dans un registre

- MOVS R0, #-1 → erreur de syntaxe
- LDR R0, = -1 → OK (l'assembleur va stocker la valeur dans une case mémoire proche de celle de l'instruction courante et va transformer l'instruction sous la forme LDR R0, [pc, #offset])

Cette solution permet de contourner les limites de l'instruction MOV

3.5.5 MOV and MVN

Move and Move NOT.

Syntax

MOV{S} Rd, Rm

MOVS Rd, #imm

MVNS Rd, Rm

where:

S Is an optional suffix. If *S* is specified, the condition code flags are updated on the result of the operation, see [3.3.6: Conditional execution on page 43](#).

Rd Is the destination register.

Rm Is a register.

Imm Is any value in the range 0-255.

- Utilisations
 - 3) Charger la valeur d'une variable déclarée dans la section de code

.text

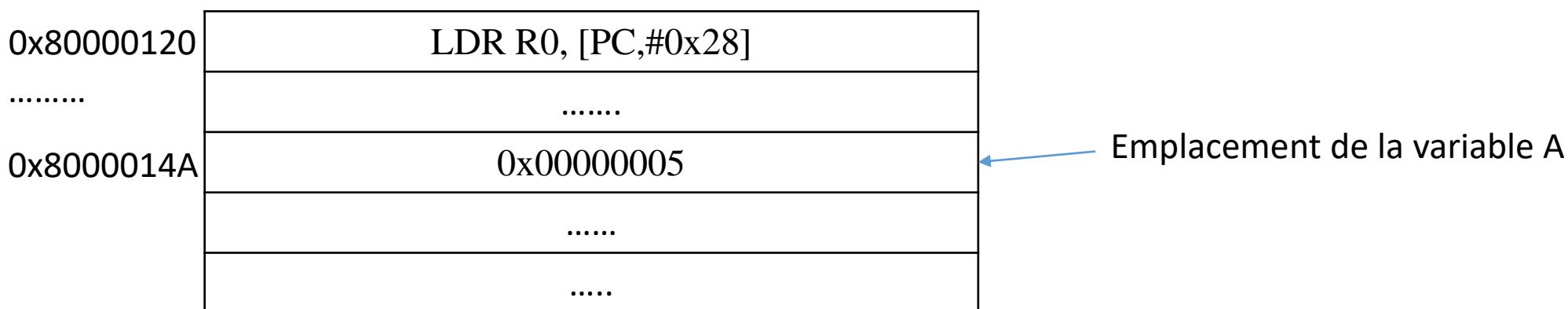
.....

LDR R0, A → charger dans R0 la valeur 5

.....

A: .word 5

- Remarque: la déclaration de A doit être après l'instruction LDR, car le processeur ne peut pas faire des décalages négatifs



- Utilisations
 - 4) Charger la valeur d'une variable déclarée dans la section de données
- .data
- A: .word 5 //stockée à l'adresse 0x20000000

.text

LDR R0, A → erreur de syntaxe

- Solution:
 - LDR R1, =A → l'assembleur transforme cette instruction comme suit: stockage de l'adresse de A dans une case mémoire et ajout d'une instruction sous la forme LDR R1, [PC, #offset]. Ici l'offset correspond au décalage entre l'adresse de l'instruction courante et celle où l'adresse de A est stockée
 - LDR R0, [R1]

0x80000120	LDR R1, [PC,#20]
0x80000122	LDR R0, [R1]
.....
0x80000136	0x20000000

- Exemple: Accès à un tableau
 - Soit Tab un tableau de 3 caractères. Ecrire un programme qui remplit le tableau avec les valeurs 1, 2 et 3, en utilisant l'adressage indirect avec un décalage relatif à un registre (pour le moment on n'utilise pas une boucle)

```
.data
Tab: .space 3

.text
LDR R0, =Tab //charger dans R0 l'adresse du tableau
MOVS R1, #0 //R1 contient l'indice de la case du tableau
MOVS R2,#1 //R2 contient la valeur à affecter
STRB R2, [R0, R1] //stocker 1 dans la première case

ADDS R1, R1, #1 //incrémenter l'indice
ADDS R2, R2, #1 //incrémenter la valeur à stocker
STRB R2, [R0, R1] //stocker 2 dans la deuxième case

ADDS R1, R1, #1 //incrémenter l'indice
ADDS R2, R2, #1 //incrémenter la valeur à stocker
STRB R2, [R0, R1] //stocker 3 dans la troisième case
```

- 1) Adressage immédiat
 - La valeur de la donnée à traiter se trouve dans l'instruction
 - MOVS R0, #0x10 //charger la valeur immédiate 0x10 dans le registre R0
- 2) Adressage direct
 - L'adresse de la donnée à traiter se trouve dans l'instruction
 - Exemple Arduino AVR 8-bits: **LDS R16, 0x0300** /*charger dans R16 la valeur contenue dans l'adresse 0x0300*/
 - Cela est possible puisque les adresses des données s'écrivent sur 16 bits (espace d'adressage limité à 64Ko) et la taille maximale d'une instruction est 32bits
 - Ce type d'adressage n'est pas possible pour le Cortex-M0+ puisque les adresses sont sur 32bits (espace d'adressage de 4Go) et la taille maximale d'une instruction est 32bits

3) Adressage indirect

- L'adresse de la donnée à traiter se trouve dans un registre
 - LDR R0, [R1]** /*charge dans le registre R0 le contenu de la case mémoire dont l'adresse est contenu dans le registre R1*/
 - LDR R0, [R1,#4]** /*charge dans le registre R0 le contenu de la case mémoire dont l'adresse est donnée par le contenu du registre R1+ un décalage de 4*/

4) Adressage relatif au compteur ordinal (PC)

LDR R0, = 0x12345678 /*charger dans R0 la valeur hexadécimale sur 32 bits*/

LDR R0, A /*charger dans R0 la valeur de la variable A déclarée dans la section de code*/

LDR R1, =B

LDR R0, [R1] /*charger dans R0 la valeur de la variable B déclarée dans la section de données*/

- C'est une architecture où le processeur ne peut pas faire directement des opérations arithmétiques et logiques sur des cases mémoires (la plupart des processeurs RISC ont cette architecture)
 - Les opérations arithmétiques et logiques ne se font qu'entre deux registres
 - Il n'y a que les instructions Load et Store qui peuvent utiliser une adresse mémoire comme opérande
- Exemple: Dans un processeur Cortex-M0+, une opération du genre $a=a+10;$ ne peut pas se faire en une seule instruction
 - $a=a+10; \rightarrow LDR\ R0,\ =a$
 $LDR\ R1,\ [R0]$
 $ADDS\ R1,\ R1,\ #10$
 $STR\ R1,\ [R0]$

- Les sauts/branchements: implémentations bas niveau des instructions si/sinon et les boucles
- Les opérations arithmétiques et logiques
- Les appels de fonction

- Deux types
 - Inconditionnels: utilisés par exemple pour l'implémentation des goto
 - Conditionnels: utilisés par exemple pour l'implémentation des si/sinon

B{cond} label /* se brancher sur l'étiquette label avec ou sans condition*/

BL label /* stocker l'adresse de retour dans le registre LR et se brancher sur l'étiquette label (appel de fonction) */

BX Rm /*se brancher sur l'adresse indiquée dans le registre Rm */

BLX Rm /* stocker l'adresse de retour dans le registre LR et se brancher sur l'adresse indiquée dans le registre Rm (appel de fonction) */

Table 22. Branch ranges

Instruction	Branch range
B <i>label</i>	-2 KB to +2 KB.
Bcond <i>label</i>	-256 bytes to +254 bytes.
BL <i>label</i>	-16 MB to +16 MB.
BX <i>Rm</i>	Any value in register.
BLX <i>Rm</i>	Any value in register.

Restrictions

In these instructions:

- Do not use SP or PC in the BX or BLX instruction.
- For BX and BLX, bit[0] of *Rm* must be 1 for correct execution. Bit[0] is used to update the EPSR T-bit and is discarded from the target address.

- Les sauts conditionnels

`B{cond} label`

- Exemples: BEQ, BNE, etc.
- Les conditions dans un saut conditionnel se basent sur le contenu du registre d'état (ceci est valable pour tous les processeurs)

Table 17. Condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal, last flag setting result was zero.
NE	Z = 0	Not equal, last flag setting result was non-zero.
CS or HS	C = 1	Higher or same, unsigned.
CC or LO	C = 0	Lower, unsigned.
MI	N = 1	Negative.
PL	N = 0	Positive or zero.
VS	V = 1	Overflow.
VC	V = 0	No overflow.
HI	C = 1 and Z = 0	Higher, unsigned.
LS	C = 0 or Z = 1	Lower or same, unsigned.
GE	N = V	Greater than or equal, signed.
LT	N != V	Less than, signed.
GT	Z = 0 and N = V	Greater than, signed.
LE	Z = 1 or N != V	Less than or equal, signed.
AL	Can have any value	Always. This is the default when no suffix is specified.

- Exemple: L'instruction BNE (Branch if Not Equal)
 - Le branchement est exécuté si Z=0

```
int a=2;
int b=3;

if (a==b) a=a+2;
else a=a+5;
a=a+10;
```

```
.data
a: .word 2
b: .word 3

.text
LDR R0, =a
LDR R1,[R0] /*stocker dans R1 la valeur de a*/
LDR R0, =b
LDR R2, [R0] /*stocker dans R2 la valeur de b*/
CMP R1, R2 /* la comparaison a le même impact sur le registre d'état que la soustraction, si a et b sont égaux le résultat est nul*/
BNE Sinon
ADDS R1, #2
LDR R0, =a
STR R1, [R0]
B Suite
Sinon:
ADDS R1, #5
LDR R0, =a
STR R1, [R0]
Suite:
ADDS R1, #10
LDR R0, =a
STR R1, [R0]
```

- Exercice:
 - Refaire le même exemple en utilisant l'instruction BEQ (Branch if Equal)

```
int a=2;  
int b=3;  
  
if (a==b) a=a+2;  
else a=a+5;  
a=a+10;
```

- Solution:
 - Refaire le même exemple en utilisant l'instruction BEQ (Branch if Equal)

```
int a=2;  
int b=3;  
  
if (a==b) a=a+2;  
else a=a+5;  
a=a+10;
```

```
.data  
A: .word 2  
B: .word 3  
  
.text  
LDR R0, =a  
LDR R1,[R0]  
LDR R0, =b  
LDR R2, [R0]  
CMP R1, R2  
BEQ Egaux  
ADDS R1, #5  
LDR R0, =a  
STR R1, [R0]  
B Suite  
Egaux:  
ADDS R1, #2  
LDR R0, =a  
STR R1, [R0]  
Suite:  
ADDS R1, #10  
LDR R0, =a  
STR R1, [R0]
```

- Il n'y a pas d'instructions équivalentes à for, while ou do ... while
 - Les boucles sont réalisées à l'aide de branchements

```
char Tab[4]={7, 10, 5, 12};  
int i, somme=0;  
for(i=0;i<4;i++)  
    somme+=Tab[i];
```

```
.data  
    Tab: .byte 7, 10, 5, 12  
    somme: .word 0  
.text  
    MOVS R0, #0 //somme=0  
    MOVS R1, #0 //i=0  
    LDR R2, =Tab //R2 contient l'adresse de la première case du tableau  
Boucle:  
    LDRB R3, [R2, R1] //R3 contient la valeur de Tab[i]  
    ADD R0, R3 // Ajouter Tab[i] à Somme  
    ADDS R1, #1 //incrémenter i de 1  
CMP R1, #4  
BNE Boucle  
    LDR R4, =somme  
    STR R0, [R4]
```

- Exemple avec une boucle while

```
int i,somme=0;  
i=1;  
while (i<=5)  
{somme+=i;  
}
```

```
.data  
    somme: .word 0  
.text  
  
    MOVS R0, #0 //somme=0  
  
    MOVS R1, #1 //i=1  
  
    CMP R1, #5  
  
    BLS Boucle //brancher si inférieur ou égal  
    B Suite  
  
    Boucle:  
        ADD R0, R1 // ajouter i à Somme  
        ADDS R1, #1 //incrémenter i de 1  
        CMP R1, #5  
        BLS Boucle  
  
    LDR R4, =somme  
  
    STR R0, [R4]  
  
    Suite:
```

- 1^{ère} méthode: sans passer par la pile

```
char Tab[4]={7, 10, 5, 12};  
int somme;
```

```
void som(){  
int i;  
for(i=0;i<4;i++)  
somme+=Tab[i];  
}
```

```
int main(){  
som();  
  
somme++;
```

Suite:

.....

.data

```
Tab: .byte 7, 10, 5, 12  
somme: .word 0
```

.text

.....

```
BL main
```

som:

```
MOVS R1, #0 //i=0  
LDR R2, =Tab /*R2 contient l'adresse de la  
première case du tableau*/  
Boucle:  
LDRB R3, [R2, R1] /*R3 contient la valeur de  
Tab[i] */  
ADD R0, R3 // Ajouter Tab[i] à Somme  
ADDS R1, #1 //incrémenter i de 1  
CMP R1, #4
```

BNE Boucle

LDR R4, =somme

STR R1, [R4]

BX LR /*se brancher sur le contenu de LR (l'adresse de retour de la fonction som())*/

...

main:

MOVS R0, #0 //somme=0

BL som /*charge l'adresse de retour dans LR avant de sauter à la fonction som()*/

LDR R4, =somme

LDR R1, [R4]

ADDS R1, #1

STR R0, [R4]

Suite:

.....

- 1^{ère} méthode: sans passer par la pile

Pc: 0x8000120
La fonction som()

```
.data
    Tab: .byte 7, 10, 5, 12
    somme: .word 0

.text
.....
.som:
    MOVS R1, #0 //i=0
.....
BX LR /*se brancher sur le contenu de LR (l'adresse de retour de la fonction som())*/
```

La fonction main()

Pc: 0x800013a

Pc: 0x800013e

```
...
main:
.....
    MOVS R0, #0 //somme=0
BL som /*charge l'adresse de retour (0x800013e) dans LR et charge dans PC l'adresse de la première instruction de la fonction som() (0x8000120)*/
    LDR R4, =somme
.....
```

- 2^{ème} méthode: en passant par la pile

```
char Tab[4]={7, 10, 5, 12};  
int somme;
```

```
void som(){  
int i;  
for(i=0;i<4;i++)  
somme+=Tab[i];  
}
```

```
int main(){  
som();  
  
somme++;
```

Suite:

.....

```
.data  
Tab: .byte 7, 10, 5, 12  
somme: .word 0  
.text  
.....  
BL main  
  
som:  
Push {LR} /*charger l'adresse de retour  
(stockée dans LR) dans la pile*/  
MOVS R1, #0 //i=0  
LDR R2, =Tab /*R2 contient l'adresse de la  
première case du tableau*/  
Boucle:  
LDRB R3, [R2, R1] //R3 contient la valeur  
de Tab[i]  
ADD R0, R3 // Ajouter Tab[i] à Somme  
ADDS R1, #1 //incrémenter i de 1  
CMP R1, #4
```

BNE Boucle

```
LDR R4, =somme  
STR R1, [R4]
```

Pop {pc} /*charger la valeur en
tête pile dans PC*/

...

main:

```
Push {LR} /*charger l'adresse  
de retour (stockée dans LR)  
dans la pile*/  
MOVS R0, #0 //somme=0
```

BL som /*charge l'adresse de
retour dans LR avant de sauter à
la fonction som()*/

```
LDR R4, =somme  
LDR R1, [R4]  
ADDS R1, #1  
STR R0, [R4]
```

Suite:

.....

- 2^{ème} méthode:
en passant par la pile

Pc: 0x8000120

La fonction som()

```
.data
    Tab: .byte 7, 10, 5, 12
    somme: .word 0

.text
.....
.som:
    Push {LR} /*charger le contenu de LR (l'adresse de retour de la
fonction) dans la pile */

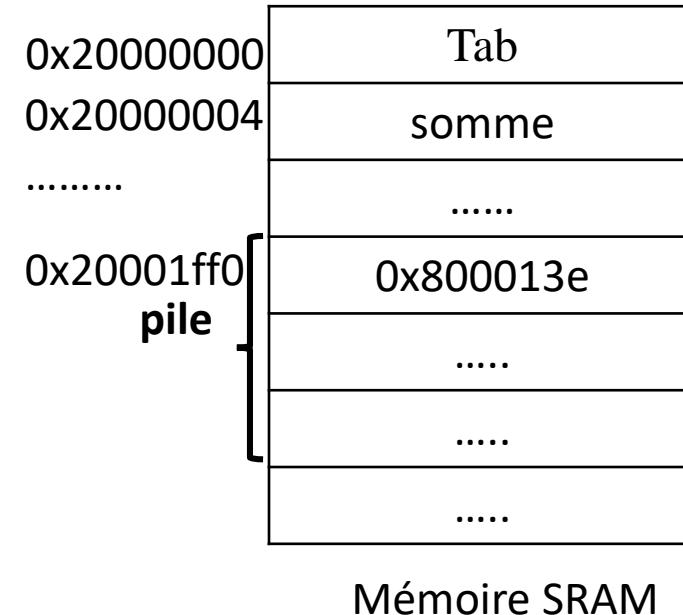
    MOVS R1, #0 //i=0
.....
    Pop {pc}/*charge le contenu de la pile (l'adresse de retour) dans le
PC*/
.....
```

La fonction main()

Pc: 0x800013a

Pc: 0x800013e

```
main:
.....
    MOVS R0, #0 //somme=0
    BL som /*charge l'adresse de retour (0x800013e) dans LR et charge
dans PC l'adresse de la première instruction de la fonction som()
(0x8000120)*/
.....
    LDR R4, =somme
.....
```



- Attention: Si la fonction modifie le contenu du registre LR (fait appel à une autre fonction), la première méthode ne marche plus

```
void fonction_B(){  
.....  
}  
  
void fonction_A(){  
fonction_B();  
.....  
}  
  
int main(){  
fonction_A();  
.....
```

```
.text  
.....  
  
fonction_B:  
.....  
BX lr  
  
fonction_A:  
BL fonction_B /*écrase le contenu de LR (perte de  
l'adresse de retour dans le main*)/  
....  
BX lr  
  
.....  
main:  
....  
BL fonction_A  
.....
```

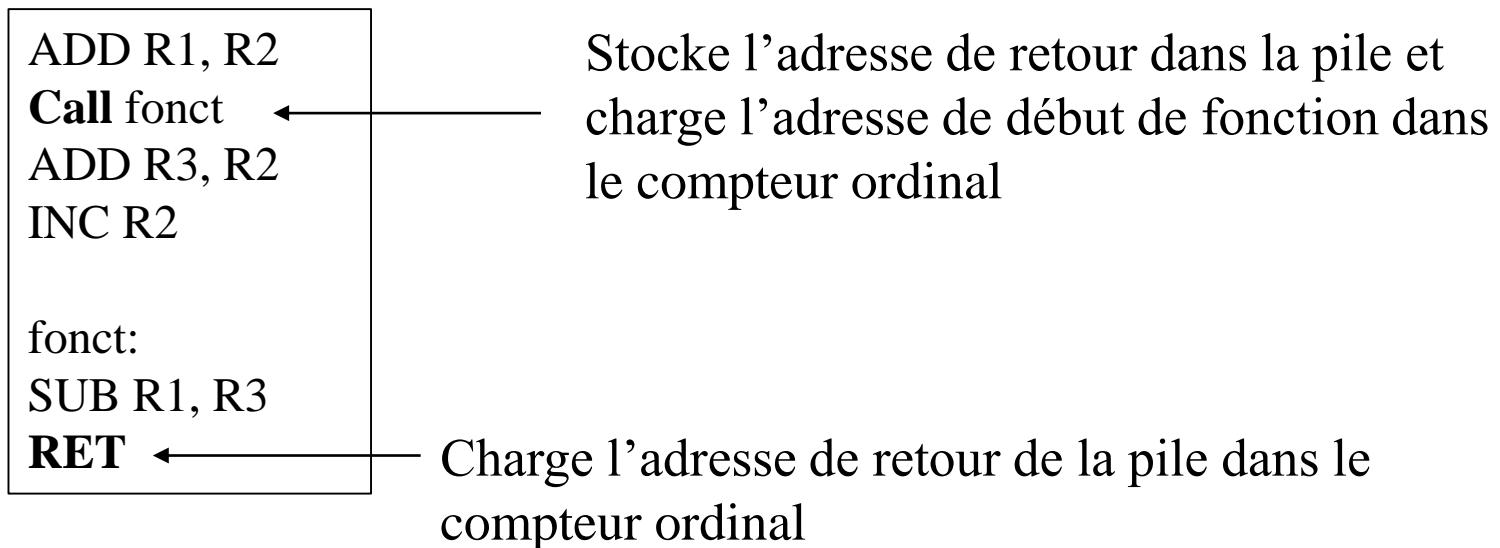
- Attention: Si la fonction modifie le contenu du registre LR (fait appel à une autre fonction), la première méthode ne marche plus
- Solution: utilisation de la pile

```
void fonction_B(){  
    ....  
}  
  
void fonction_A(){  
  
fonction_B();  
    ....  
}  
  
int main(){  
fonction_A();  
    ....
```

```
.text  
  
.....  
  
fonction_B:  
push {lr}  
.....  
pop {pc}  
  
fonction_A:  
push {lr}  
BL fonction_B  
.....  
pop {pc}  
  
....  
main:  
BL fonction_A  
.....
```

Les appels de fonction

- Il y a des processeurs où l'appel d'une fonction (call) stocke implicitement l'adresse de retour dans la pile et le retour charge implicitement cette adresse dans le compteur ordinal.
- Exemple: processeur de l'Arduino Uno/Mega (AVR)



- Variantes de l'addition:
 - **ADD R1,R1,R2 ou ADD R1,R2**
⇒ $r1=r1+r2$ fait l'addition des deux registres sans mettre à jour le registre d'état APSR
 - **ADD R1, R1,#5 ou ADD R1,#5**
⇒ $r1=r1+5$ fait l'addition de r1 et 5 sans mettre à jour le registre d'état APSR
 - **ADDS R1,R1,R2**
⇒ $r1=r1+r2$ fait l'addition des deux registres et met à jour le registre d'état APSR
 - **ADDS R1,R1,#10**
⇒ $r1=r1+10$ fait l'addition de r1 et 10 et met à jour le registre d'état APSR
 - **ADCS R1,R1,R2**
⇒ $r1=r1+r2+retenu$ fait l'addition des deux registres + la retenue et met à jour le registre d'état APSR
- Attention: Si on utilise 3 opérandes, le registre destination (premier opérande) doit être le même que le deuxième opérande Exemple: ADD R1,R1,R2 fonctionne normalement, mais ADD R2, R1, R2 génère une erreur

- L'addition sur 8 bits

```
uint8_t a=3,b=4;  
...  
a=a+b;
```

```
.data  
a: .byte 3  
b: .byte 4  
  
.text  
....  
LDR R0, =a /*R0 pointe sur a*/  
LDRB R1, [R0] /*charger dans R1 le contenu pointé par R0*/  
  
LDR R3, =b /*R1 pointe sur b*/  
LDRB R2, [R3] /*charger dans R2 le contenu pointé par R3*/  
  
ADD R1, R2 /*fait l'addition de R1 et R2 et met le résultat dans R1*/  
  
STRB R1, [R0] /*stocke le contenu de R1 à l'adresse a pointée par R0 */
```

- L'addition sur 16 bits

```
uint16_t a=3,b=4;  
...  
a=a+b;
```

```
.data  
a: .hword 3  
b: .hword 4  
  
.text  
....  
LDR R0, =a  
LDRH R1, [R0]  
LDR R3, =b  
LDRH R2, [R3]  
  
ADD R1, R2  
  
STRH R1, [R0]
```

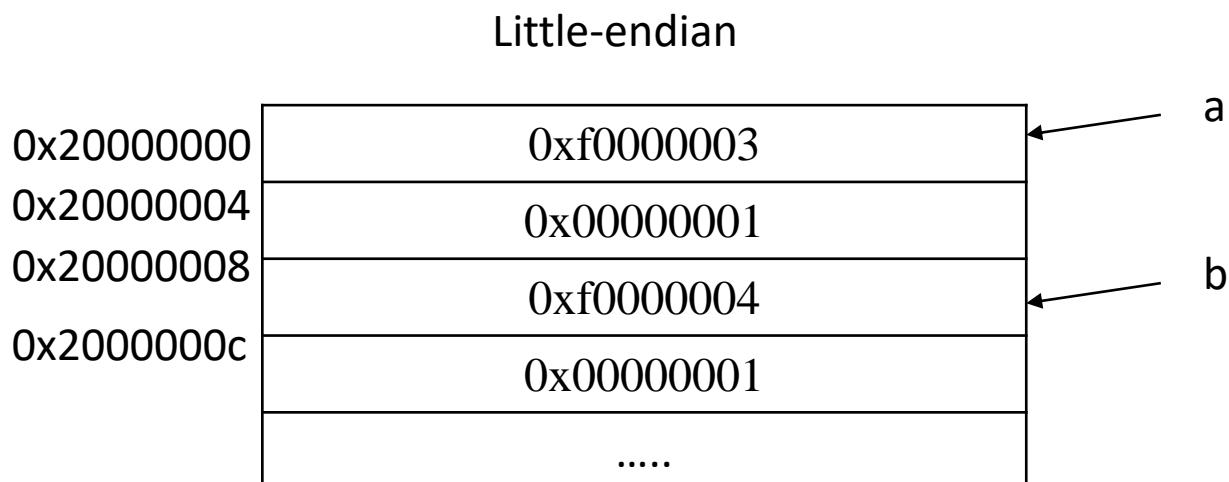
- L'addition sur 32 bits

```
uint32_t a=3,b=4;  
...  
a=a+b;
```

```
.data  
a: .word 3  
b: .word 4  
  
.text  
....  
LDR R0, =a  
LDR R1, [R0]  
LDR R3, =b  
LDR R2, [R3]  
  
ADD R1, R2  
  
STR R1, [R0]
```

- L'addition sur 64 bits

```
Uint64_t  
a=0x00000001f0000003,b=0x00000001f0000004;  
...  
a=a+b;
```



```
.data  
a: .word 0xf0000003,1  
b: .word 0xf0000004,1  
.text  
....  
LDR R0, =a  
LDR R1, [R0] /*stocker les 32 bits de poids  
faible de a dans R1*/  
ADDS R0,#4  
LDR R2, [R0] /*stocker les 32 bits de poids  
fort de a dans R2*/  
  
LDR R0, =b  
LDR R3, [R0] /*stocker les 32 bits de poids  
faible de b dans R3*/  
ADDS R0,#4  
LDR R4, [R0] /*stocker les 32 bits de poids  
fort de b dans R4*/  
  
/*Addition avec retenue*/  
ADDS R1, R3  
ADCS R2, R4  
LDR R0,=a  
STR R1,[R0]  
ADDS R0,#4  
STR R2, [R0]
```

- La multiplication

MULS

Multiply using 32-bit operands, and producing a 32-bit result.

Syntax

MULS Rd, Rn, Rm

where:

Rd Is the destination register.

Rn, Rm Are registers holding the values to be multiplied.

Operation

The MUL instruction multiplies the values in the registers specified by Rn and Rm, and places the least significant 32 bits of the result in Rd. The condition code flags are updated on the result of the operation, see [3.3.6: Conditional execution on page 43](#).

The results of this instruction does not depend on whether the operands are signed or unsigned.

Restrictions

In this instruction:

- Rd, Rn, and Rm must only specify R0-R7.
- Rd must be the same as Rm.

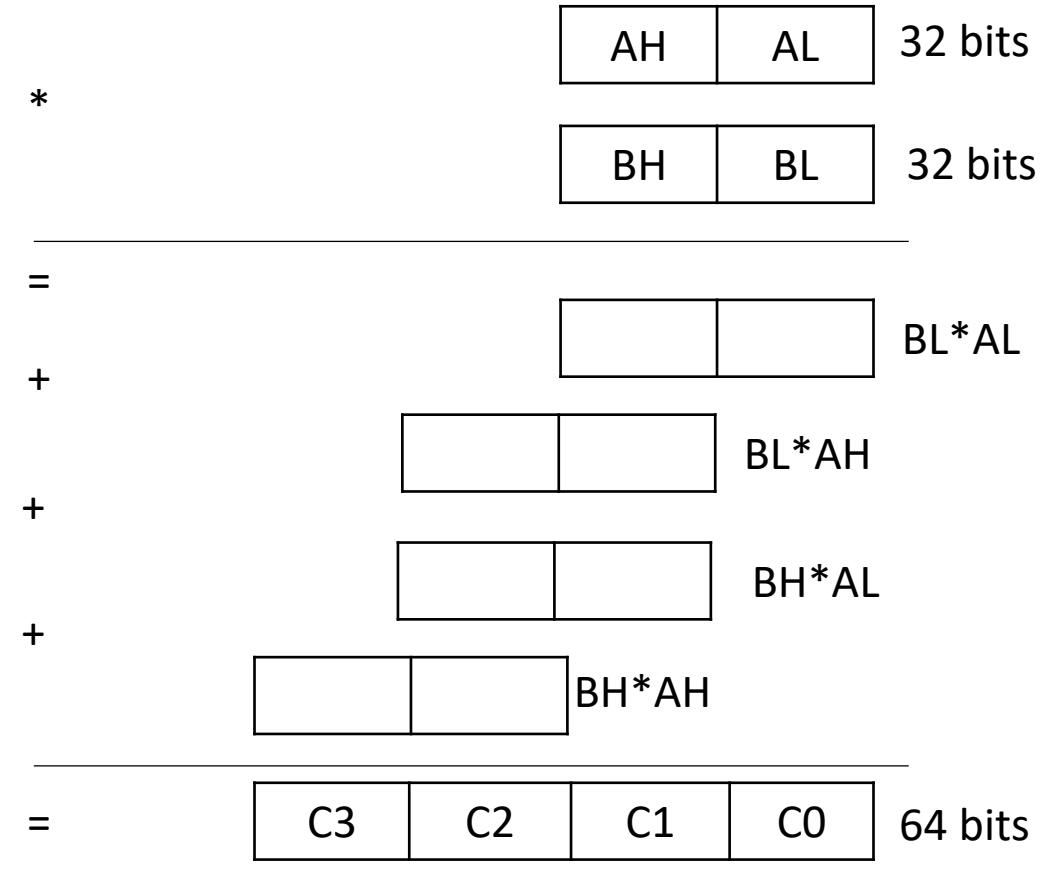
- Multiplication sur 32 bits
 - Exemple d'une multiplication sans débordement

```
uint32_t a=2,b=3;  
uint32_t c;  
...  
c=a*b;
```

```
.data  
a: .word 2  
b: .word 3  
c: .space 4  
.text  
....  
LDR R0, =a  
LDR R1, [R0]  
LDR R3, =b  
LDR R2, [R3]  
  
MULS R1, R2,R1  
LDR R0, =c  
STR R1, [R0]
```

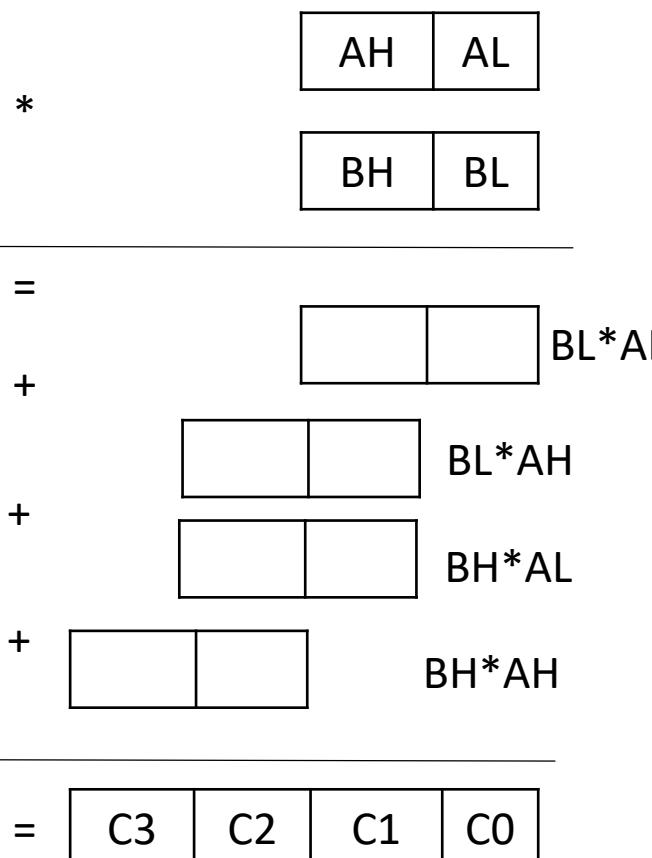
- Multiplication sur 32 bits
 - Exemple d'une multiplication avec débordement (résultat sur 64 bits)

```
uint32_t  
a=0x70000000,b=0x70000000;  
uint64_t c;  
...  
c=a*((uint64_t) b);
```



- Multiplication sur 32 bits

- Exemple d'une multiplication avec débordement (résultat sur 64 bits)



.data	MULS R5, R3, R5 /*result=R5= ah*bh*/ MOV R7, R5 LSLS R5, #16 // result <= 16 LSRS R7, #16 ADDS R7, R1	MOV R1, R5 LSLS R5, #16 // result <= 16 LSRS R7, #16 LSRS R7, #16 ADDS R7, R1
a: .word 0x70000000		
b: .word 0x70000000		
c: .space 8		
.text		
....		
LDR R0,=a	MOV R6, R1 //R6=ah	MOV R6, R2 //R6=al
LDR R1,[R0] //R1=a	MULS R6, R4, R6 //R6=ah*bl	MULS R6, R4, R6 //R6=al*bl
LDR R2, =0xffff		
MULS R2, R1, R2 //R2=al	MOVS R0,#0	ADDS R5, R6 // result+=ah*bl
LSRS R1, #16 //R1=ah	ADDS R5, R6 // result+=ah*bl	ADCS R7, R0
	ADCS R7, R0	ADCS R7, R0
LDR R0,=b	MOV R6, R2 //R6=al	LDR R0,=c
LDR R3,[R0] //R3=a	MULS R6, R3, R6 //R6=al*bh	STR R5,[R0]
LDR R4, =0xffff		ADDS R0,#4
MULS R4, R3, R4 //R4=bl	ADDS R5, R6 // result+=al*bh	STR R7,[R0]
LSRS R3, #16 //R3=bh	ADCS R7, R0	
MOV R5, R1		

- Division
 - Il n'y a pas un module matériel qui fait la division sur le Cortex-M0+ (le matériel nécessaire était trop coûteux et prend beaucoup de place dans un CPU en comparaison aux autres opérateurs)
 - Comment le Cortex-M0+ traduit une division écrite en langage C?
 - Deux cas:
 - Division par une puissance de 2 → décalage à droite
 - Division par un nombre qui n'est pas une puissance de 2 → traduction en un ensemble d'opérations d'addition, multiplication, etc.

- Division
 - Division par une puissance de 2

```
uint8_t a,b;  
a=255;  
...  
b=a/4;
```

- LSR: Logical Shift Right

```
.data  
a: .byte 255  
b: .space 1  
.text  
LDR R0,=a  
LDRB R1,[R0]  
LSRS R1,#2  
.....  
LDRB R0,=b  
STRB R1, [R0]
```

11111111

255

LSRS R1,#2



00111111

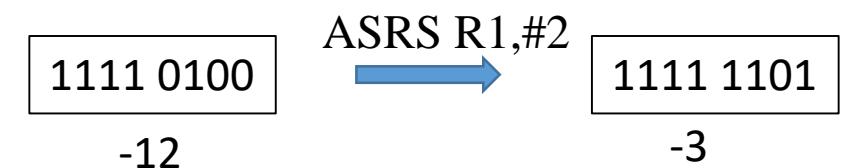
63

- Division
 - Division par une puissance de 2

```
int8_t a,b;  
a=-12;  
...  
b=a/4;
```

- ASRS: Arithmetic Shift Right
 - Fait le décalage avec une extension du signe

```
.data  
a: .byte -12  
b: .space 1  
.text  
LDR R0,a  
MOVS R2,#0  
LDRSB R1,[R0,R2]  
ASRS R1,#2  
.....  
LDR R0,b  
STRB R1, [R0]
```



- Division
 - Division par un nombre qui n'est pas une puissance de 2
 - 1^{ère} méthode: traduire la division en un ensemble d'opérations équivalentes

```
uint8_t a,b;  
a=255;  
...  
b=a/5;
```

```
.data  
a: .byte 255  
b: .space 1  
.text  
LDR R0,=a  
LDRB R1,[R0]  
LDR R2,=205  
MULS R1,R2,R1  
LSRS R1, #10  
.....  
LDRB R0,=b  
STRB R1, [R0]
```

La division par 5:

- Une multiplication par 205
- Division de ce résultat par 1024 (décalage à droite de 10 bits)

$$\rightarrow ((255*205)>>10= 51$$

- Division
 - Division par un nombre qui n'est pas une puissance de 2
 - 2^{ème} méthode: écrire en langage C et laisser la main au compilateur

```
uint8_t a,b;  
a=255;  
...  
b=a/5;
```

```
08000140 <__udivsi3>:  
 8000140:2200    movsr2, #0  
 8000142:0843    lsrsr3, r0, #1  
 8000144:428b    cmpr3, r1  
 8000146:d374    bcc.n8000232  
  
<__udivsi3+0xf2>  
 8000148:0903    lsrsr3, r0, #4  
 800014a:428b    cmpr3, r1  
 800014c:d35f    bcc.n800020e  
  
<__udivsi3+0xce>  
 800014e:0a03    lsrsr3, r0, #8  
 8000150:428b    cmpr3, r1  
 8000152:d344    bcc.n80001de
```

.....

```
#include "stm32l0xx.h"
#include "stm32l0xx_nucleo.h"

uint8_t a=5;
uint8_t b=7;
uint8_t c;

void addition(void);
int main(void)
{addition();
}
```

Un fichier « .c » pour le main et les fonction en C

```
.global addition
.syntax unified
.text

.addition:

push{lr}

ldr r0,=a
ldrb r1,[r0]

ldr r2,=b
ldrb r3,[r2]

ldr r4,=c

add r1, r3
strb r1, [r4]

pop{pc}
```

Un fichier « .s » pour les fonctions assembleur

- Le passage des paramètres et le retour d'une fonction: les paramètres d'une fonction déclarée en langage C sont automatiquement stockés dans les registre r0-r3

Registre	Utilisation
r0	Premier argument et valeur de retour Si le retour est sur 64 bits, il est écrit dans r0 et r1 Si le premier argument est sur 64 bits, il est écrit dans r0 et r1
r1	Deuxième argument
r2	Troisième argument
r3	Quatrième argument Si le retour est sur 128 bits, il est écrit dans r0, r1, r2 et r3 Si la taille totale des argument dépasse 128 bits, la pile est utilisée

```

uint8_t a=5;
uint8_t b=7;
uint8_t c;

uint8_t addition(uint8_t a, uint8_t b);
int main(void)
{.....
c=addition(a,b); /*c prend automatiquement la
valeur de r0*/
.....
}

```



.....
.text

addition:

push{lr}

.....

add r0,r1 /*il faut s'assurer que le résultat de la fonction soit stockée dans r0*/

.....

pop{pc}

- Le passage des paramètres et le retour d'une fonction

```
#include "stm32l0xx.h"
#include "stm32l0xx_nucleo.h"

uint64_t a=0xffffffff;
uint64_t b=0x20000001;
uint64_t c;

uint64_t addition(uint64_t a, uint64_t b);
int main(void)
{.....
c=addition(a,b);
...
}
```

r1-r0 r3-r2

```
.global addition
.syntax unified

.text
addition:
push{lr}
... .
adds r0,r2
adcs r1,r3 /*la valeur de retour est écrite dans r0 et r1*/
.....
pop{pc}
```

- Jeu d'instructions du Cortex-M0+: https://www.st.com/resource/en/programming_manual/dm00104451.pdf
- Les appels de fonctions:
<http://www.eng.auburn.edu/~nelson/courses/elec2220/slides/ARM%20prog%20model%206%20subroutines.pdf>