

Premier Pas

Comprendre ce qu'est le **Langage
d'Assemblage**



Extrait de listing :

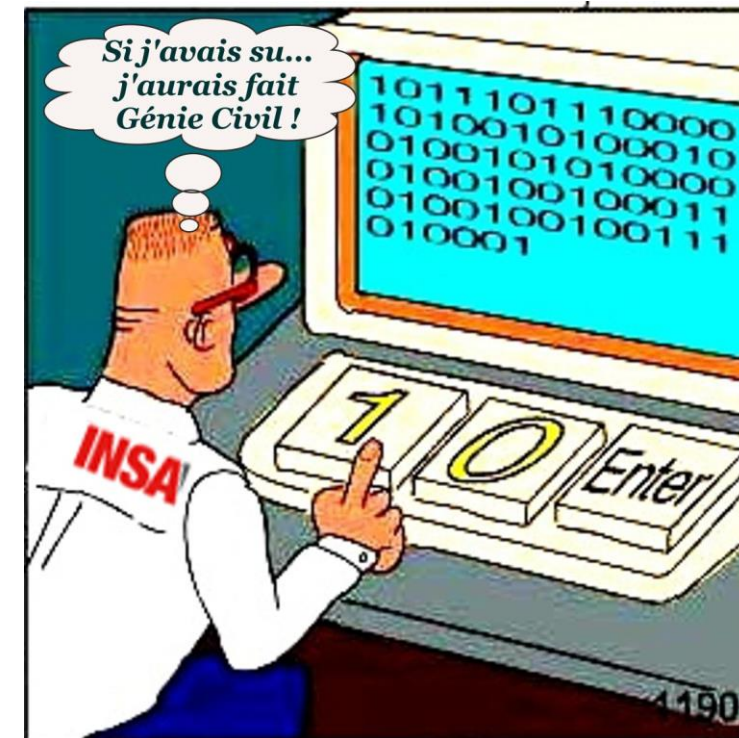
```
C:\Users\mahout\Documents\Enseignement\Informatique_Materielle\Assembleur\Transparent\Prog_Fil_Rouge\Fil_Rouge.uvprojx - µVision [Non-Commercial Use License]
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Cible_Simule
Project: Fil_Rouge
  Cible_Simule
  MesFichiers
    Affichage.asm
    Calcullette.asm
    Fil_Rouge.asm
  Libraries
    serial.c
    CMSIS
    Device
    RTE_Device.h (Startup)
    startup_stm32f10x_md.s (Startup)
    system_stm32f10x.c (Startup)
  Fil_Rouge.asm*
56 ; #####
57
58 ; -----
59 ; Procédure principale
60 ; -----
61 main    PROC
62
63     MOV R0 , #435
64     MOV R1 , #43
65     SDIV R2,R0,R1
66     MUL R3,R2,R1
67     SUB R3,R0,R3
68
69     ADDS R0 , R0 , #1
70     ADC R1 , R0
71
72     BL init_serial      ; Initialisation liaison série
73
74     BL Affichaine       ; Affichage du message d'accueil
75
76     BL Calcullette      ; Appel à Moyenne = (ValUn + ValDeux)/2
77
78     LDR R3,=Msg2
79     BL Affichaine       ; Affichage du message de résultat
80
81     LDR R1,=Moyenne
82     LDR R7,[R1]
83     LDR R3,=ChaineVide
84     BL Convertit        ; Conversion du résultat 32 bits en chaine de caractères
85     MOV R3,R0
86     BL Affichaine       ; Affichage du résultat
87 Finir  B Finir          ; Boucle infinie
88
89     ENDP
90
91 ; #####
92
93
94     END
95
```

Le genre de chose que vous allez devoir écrire !

A voir comme cela....mais si j'ai déjà programmé, cela ne ressemble à rien de ce que je connais !

Qu'est qu'un programme informatique ?

- = Séquence d'ordres numériques, appelées instructions
 - + lues en mémoire
 - + codées sous forme binaire que l'on exprime (affichage) souvent en hexadécimal
- \Rightarrow Ecrire un programme = écrire des séquences binaires dans un espace mémoire accessible par un processeur
- On parle de *Code Machine*



Niveau supérieur au code machine...

- Le **langage d'assemblage**
 - + listing = ce que doit faire (instruction) la CPU à chaque « cycle d'horloge »
 - + il est propre et associé au processeur utilisé
 - + il nécessite un **assembleur** : programme qui traduit le listing en code machine

- Exemple :

MOV R1 , #0xAA

- Le registre général **R1** du processeur reçoit la valeur **170** .
- Le code machine, traduit par l'assembleur est :
 - + en hexadécimal : **E3A010AA**
 - + en binaire : **1110001110100000001000010101010**

Formellement :

**Assembleur
(Logiciel de traduction)
≠
Langage d'assemblage**

Pourquoi utiliser du LA ?

Gros défaut

- Langage d'assemblage propre à chaque processeur....
- Jeu d'instructions réduit et rudimentaire
- Pénible et fastidieux
- Très peu lisible
- Impossible de développer de gros projets

Intérêt

- Maîtrise « totale » du code exécuté
- Comprendre un compilateur
- Mieux appréhender : structures algorithmiques de base, pointeurs, structures de données, variables locales, passage d'arguments...
- Optimisation (temps ou taille de code)

Quand utiliser ce type de langage ?

- Quand on n'a pas le choix (on fait l'INSA par exemple ...) !
- Efficacité : de façon très temporaire dans un programme C
- Besoin d'efficacité
- Besoin de certification
- Besoin d'accéder à des ressources très bas niveau (d'autres langages le peuvent aussi)
- Besoin de mettre au point (bogue vicieuse)
- Besoin de hacker
- Besoin de développer un compilateur

```
int f(int x)
{
    int r0;
    asm
    {
        ADD r0, x, 1
        EOR x, r0, x
    }
    return x;
}
```

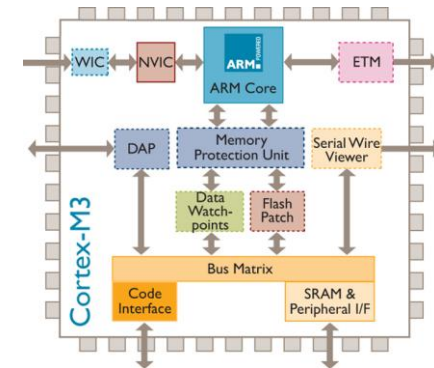


Objectifs

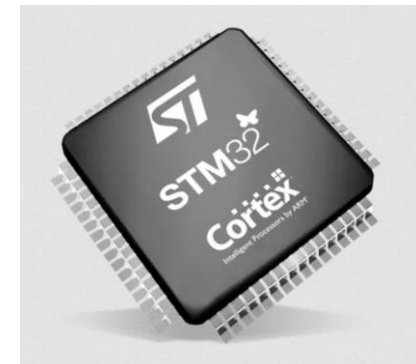
- Appréhender le langage d'assemblage avec une vision générique
 - + les mécanismes de base sont toujours similaires
 - + programmation modulaire systématique
- Se former en pratique (TP) avec des méthodes et des outils de développement avancés : compilateur ARM Keil μ vision
- Commencer à comprendre quelques mécanismes de la problématique *Temps Réel* et de la programmation de périphériques



Application au Cortex- M3



qui est le **cœur** d'un STM32 :

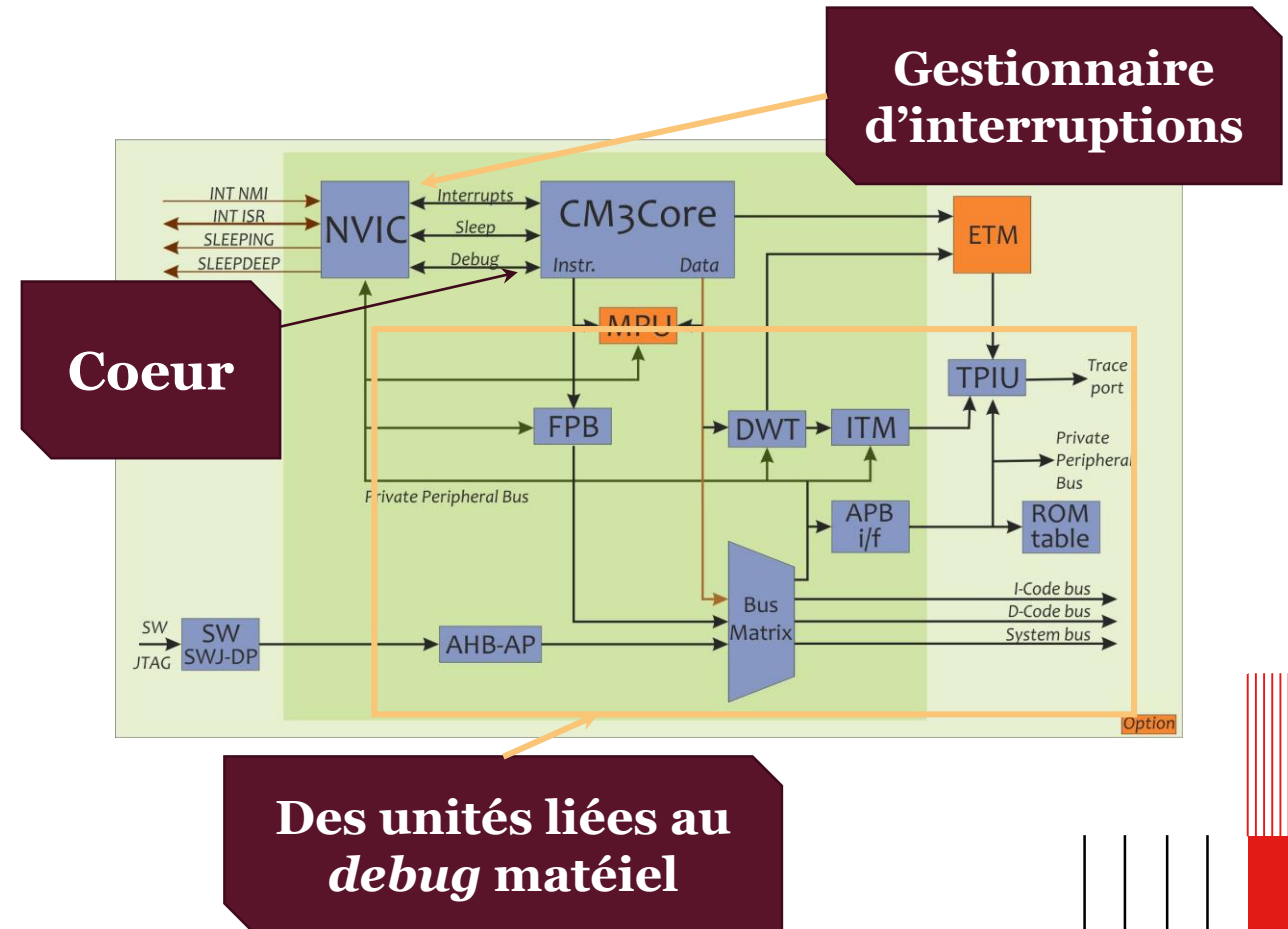


Le Cortex rapidement

■ ARM®

- + Société anglaise : concepteur uniquement (brevets)
- + **A**dvanced **R**isc **M**achines (*reduced instruction-set computer*)
- + Cortex \Rightarrow architecture RISC
- + grosse implantation dans l'embarqué : 95% du marché de la téléphonie mobile

Sans entrer dans le détail :



Le Cortex rapidement

Le Cœur

- Processeur 32 bits
- Fréquence habituelle 72Mhz
- 13 registres généraux
- 5 registres « spécialisés »

Le NVIC

- Nested Vector Interrupt Control
- Cortex = cœur + des micro-machines périphériques
- NVIC = Chef d'orchestre pour ordonnancer

Les unités Debug :

Idée générique : offrir la possibilité de tester, fiabiliser (voire certifier) le code sans jamais introduire la moindre modification

Le Cortex est « vendu » à des fondeurs (NXP, TI, AMTEL,..) qui en font un processeur.
Chez **STmicro** c'est toute la **gamme STM32**

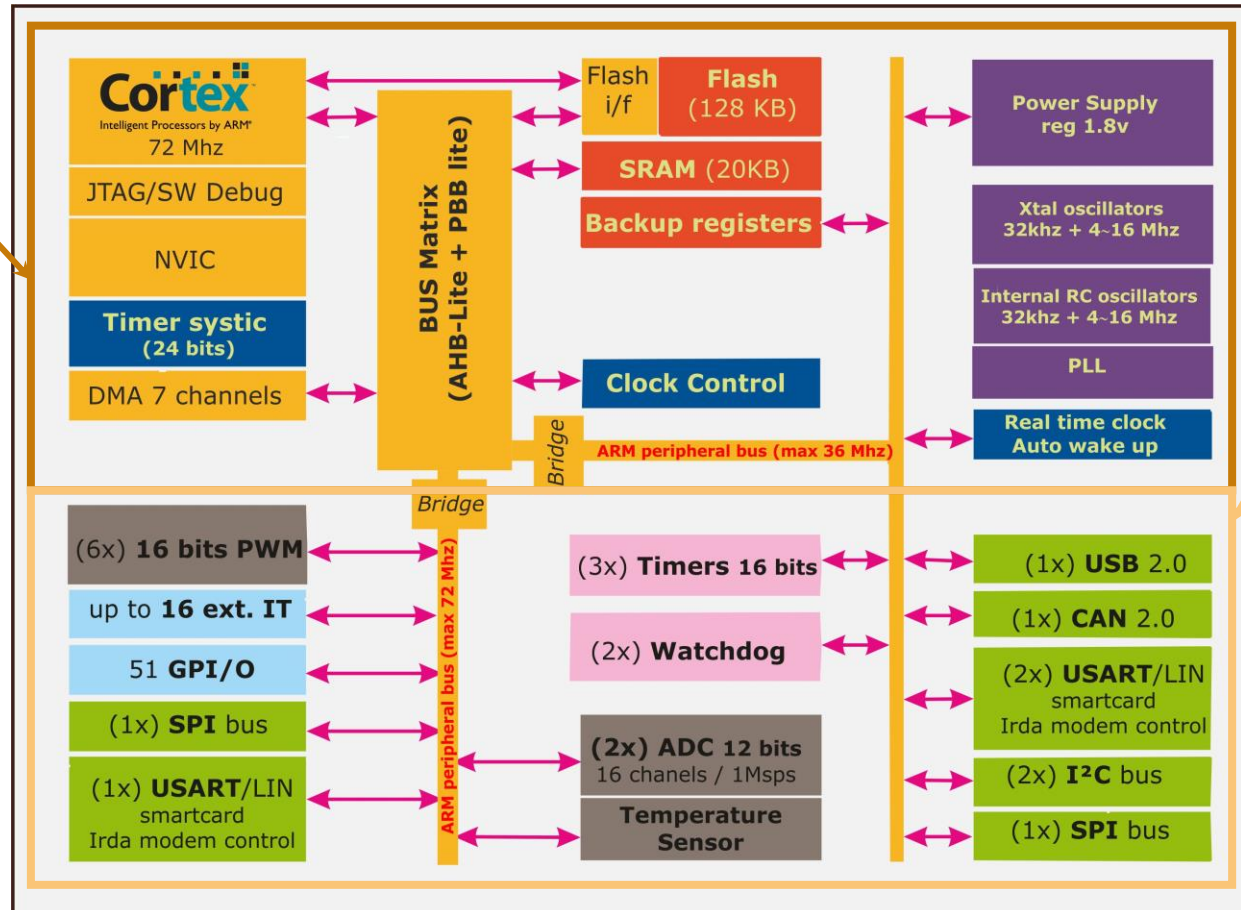


Un STM32 rapidement

Le Cortex + Mémoire + Horloges

Choix du fondeur

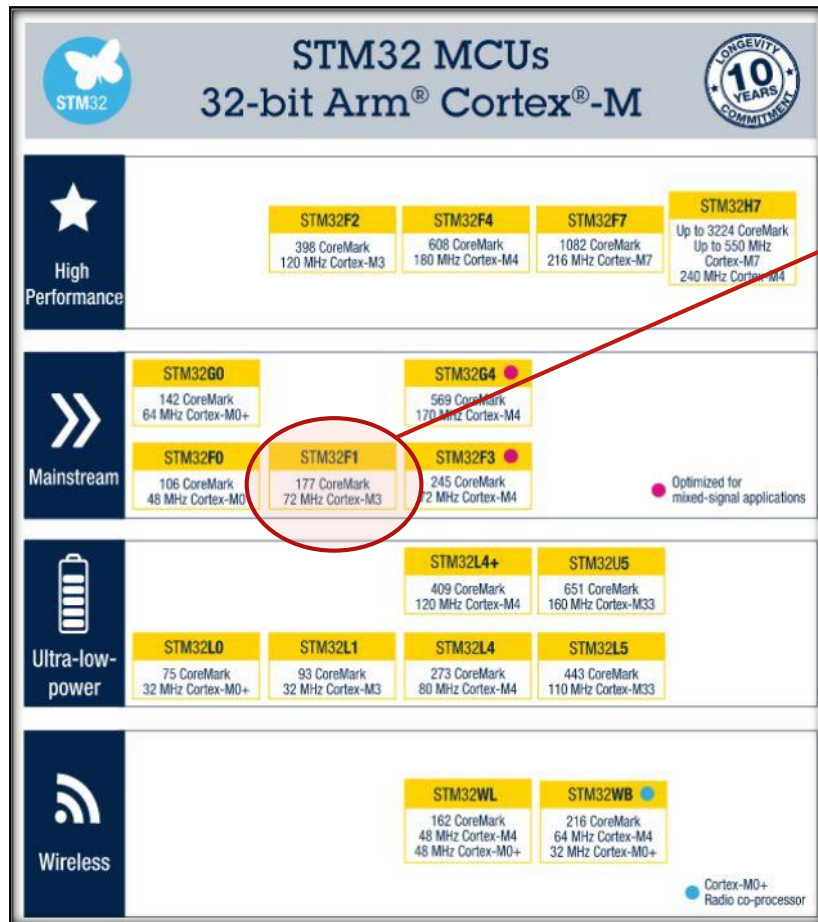
- Quantité de mémoire
- Types et quantité de périphériques
- Type de boîtier



« Micro machines » périphériques

- + E/S binaires (bleu)
- + E/S analogiques (brun)
- + Mesure du temps (rose)
- + Bus numériques (vert)

STM32 : une bien grande famille

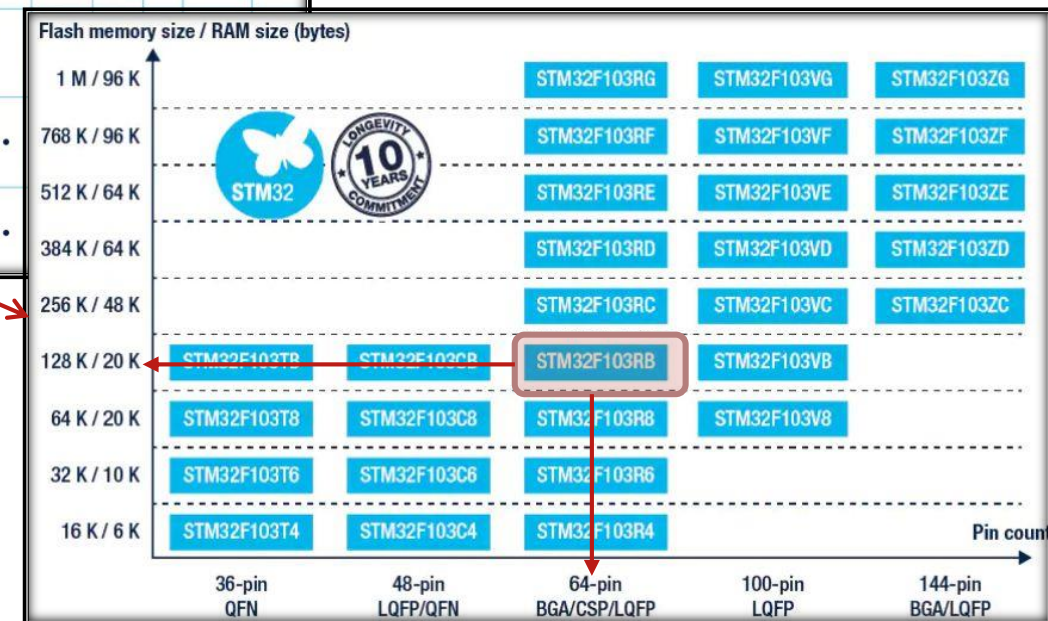


STM32F1 MCU Series
32-bit Arm Cortex-M3 (DSP + FPU) – Up to 72 MHz

Product line	FCPU (MHz)	Flash (Kbytes)	RAM (Kbytes)	USB 2.0 FS	FSMC	CAN 2.0B	3-phase MC Timer	IPS	SDIO	Ethernet IEEE1588	HDMI CEC
STM32F100 Value line	24	16 to 512	4 to 32		•		•				•
STM32F101	36	16 to 1M	4 to 80		•						
STM32F102	48	16 to 128	4 to 16	•							
STM32F103	72	16 to 1M	4 to 96	•	•	•					
STM32F105 / STM32F107	72	64 to 256	64	•	•	•					

Features:
 • -40 to 105°C range
 • USART, SPI, I2C
 • 16- and 32-bit timers
 • Temperature sensor
 • Up to 3x12-bit ADC
 • Dual 12-bit DAC
 • Low voltage 2.0 to 3.6V (5V tolerant I/O)

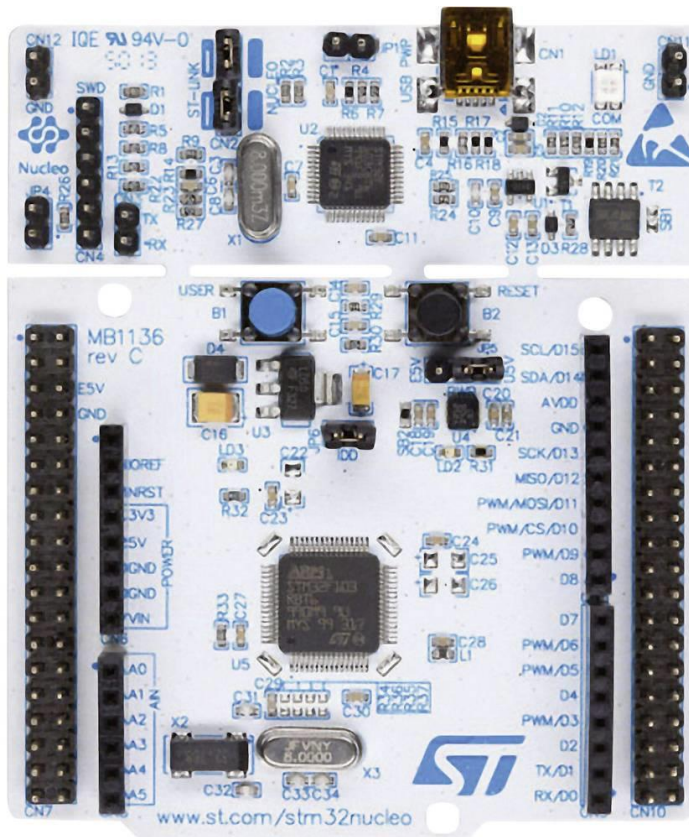
Le STM2 sur lequel on va faire les TP
STM32F103RB





Les TP :

Une carte nucléo

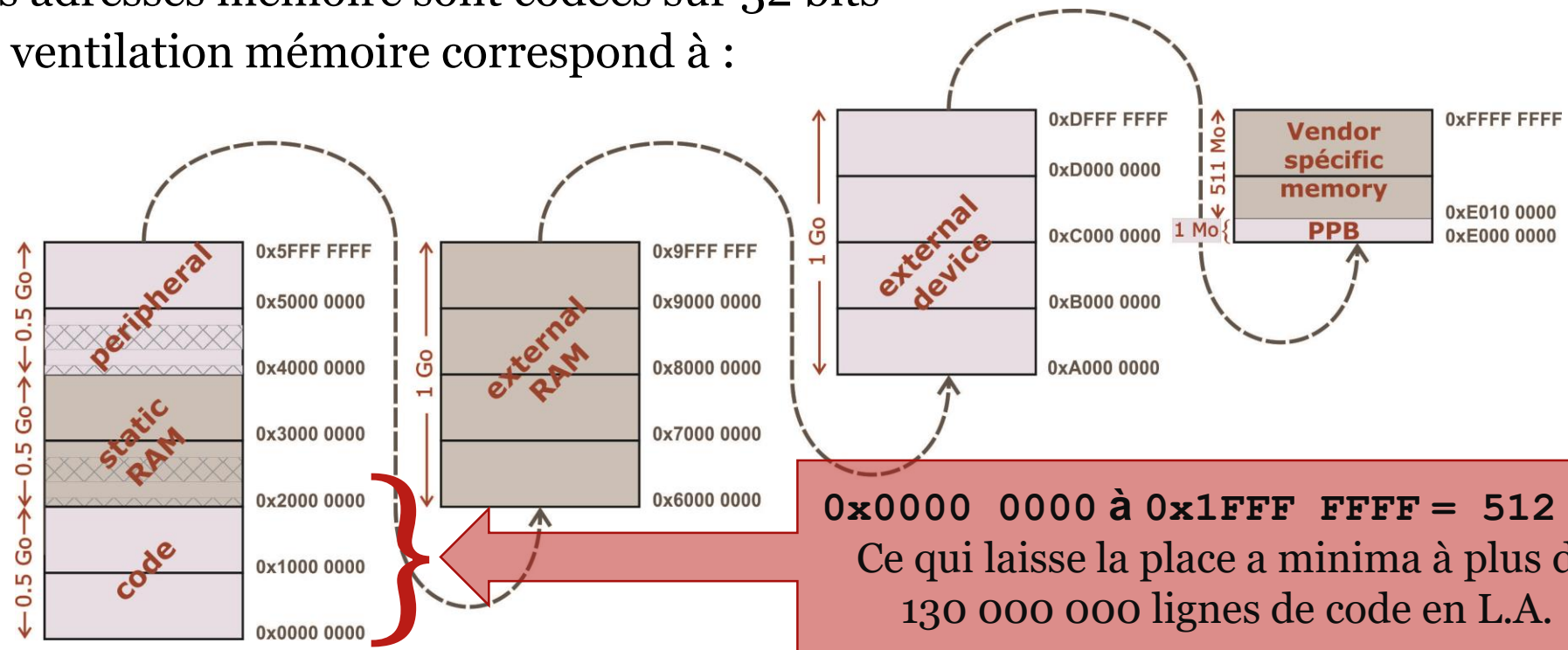


Une roue magique



Continuons à ouvrir le capot :

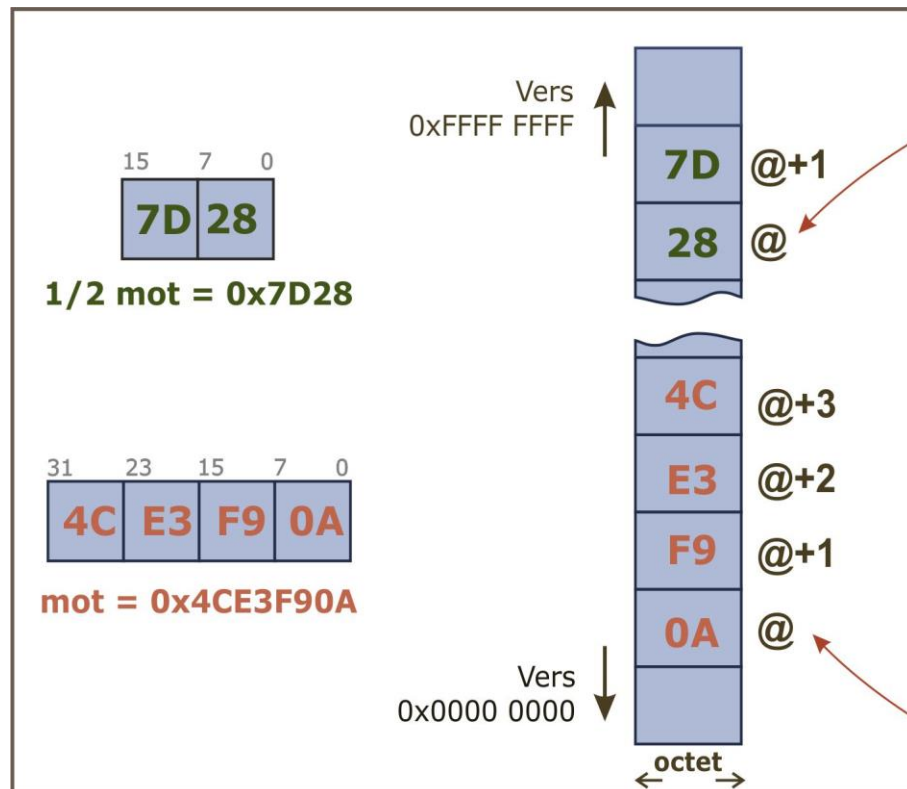
- Le Cortex M3 est une architecture de type Harvard
 - + séparation de la mémoire CODE et de la mémoire DATA
 - + les adresses mémoire sont codées sur 32 bits
 - + la ventilation mémoire correspond à :



Toujours à propos de la mémoire

- Organisation en little-endian.

une @ mémoire (32 bits) = un octet



- L'accès alignée à la mémoire

+ accès à un 1/2 mot (16 bits)

- Adresse paire :
bit de poids 0 est à 0
- exemple : 0x40002462

adresse de base
paire

+ accès à un 1/2 mot (16 bits)

- Adresse doublement paire
bit de poids 0 est à 0
bit de poids 1 est à 0
- exemple 0x40002464

adresse de base
doublement paire



Commençons l'analyse du code

- Prenons les 3 dernières lignes de fichier *Principal.asm*
- Une ligne type se compose de 4 champs :

	MOV	R3,R0	
	BL	Affichaine	;Affichage du résultat.
Finir	B	Finir	;Boucle infinie
[Etiquette]	mnémonique	[expr1] [,expr2] [,expr3]	; comments...
1	2	3	4

Les 4 champs :

	MOV	R3,R0	
	BL	Affichaine	;Affichage du resultat
Finir	B	Finir	;Boucle infinie
[Etiquette]	mnémonique	[expr1] [,expr2] [,expr3]	; commentaires...
①	②	③	④

- **① - Etiquettes** : c'est un symbole qui permet de repérer une ligne de code particulière
 - + le symbole correspondra à une adresse de la mémoire « code »
 - + doit être OBLIGATOIREMENT en 1^{er} colonne
- **② - Instructions** : c'est ce que doit faire la CPU à cette étape
 - + l'ensemble des instructions : jeu d'instructions du processeur
- **③ - Opérandes** : c'est ce qu'utilise l' instruction pour réaliser son instruction
 - + il peut en avoir aucun, **1**, **2** ou **3** (exceptionnellement 4)
 - + cela peut être : Valeur, Symbole, Registre, Adresse mémoire
- **④ - Commentaires** : ce sont les commentaires
 - + commencent avec un « ; » et termine avec le fin de la ligne



Les opérandes : les **registres généraux**

- Le Cortex possède 13 registres dits à usage général : **R0** à **R12**
 - Les 3 registres suivants correspondent à des fonctions particulières
 - + **R13** : pointeur de pile (**SP**)
 - + **R14** : registre de lien (**LR**)
 - + **R15** : pointeur ordinal (**PC**)
 - + **xPSR** : registre « statut »
 - Tous les registres sont 32 bits
 - Ils sont **non sécables** (en 2x16, en 4x8)
- Exemple :

Ligne de code	Signification
ADC R1 , R0	R1 reçoit $R1 + R0 +$ « carry »
SUB R3 , R0 , R3	R3 reçoit $R0 - R3$
LDR R7 , [R1]	R7 est chargé avec le contenu de l'adresse mémoire stockée dans R1
BX LR	Le processeur « saute » à l'adresse contenue dans LR

Quand l'ALU travaille avec un registre, il travaille toujours avec les 32 bits du registre

Les registres SP,LR et PC

- Utilisation dédiée par le processeur
- A ne pas utiliser directement sauf en « sachant » ce que l'on fait
- **SP** : pointeur de la pile système
 - + pointe obligatoirement sur une zone mémoire accessible
 - + usage : sauvegarde temporaire, passage arguments , interruptions
- **LR** : registre de lien
 - + rôle dans l'appel à des procédures
 - + rôle dans les interruptions
- **PC** : pointeur d'instructions
 - + pointe sur l'adresse mémoire de la prochaine instruction
 - + s'incrémente automatiquement avec le déroulement du programme
 - + instructions toujours codées sur 2 ou 4 octets \Rightarrow incrémentation de +2 ou +4
 - + appel à une procédure = affectation de **PC** avec l'adresse où est stockée la procédure
- **xPSR** : Processor Status Register
 - + 3 rôles différents
 - + principal : compte-rendu sur l'exécution d'une instruction

Les opérandes : les registres généraux (2)

- Si **2** opérandes

- + **instr** **Rx**, **Ry**

- + **Rx** : registre destination qui reçoit le résultat

- + **Rx** et **Ry** : registres sources

- Si **3** opérandes

- + **instr** **Rx**, **Ry**, **Rz**

- + **Rx** : registre destination qui reçoit le résultat

- + **Ry** et **Rz** : registres sources

- Exemple :

Ligne de code	Signification
ADC R1 , R0	R1 reçoit R1 + R0 + « carry »
SUB R3 , R0 , R3	R3 reçoit R0 - R3
LDR R7 , [R1]	R7 est chargé avec le contenu de l'adresse mémoire stockée dans R1
BX LR	Le processeur « saute » à l'adresse contenue dans LR



Comment est codé l'opérande dans l'instruction

- Prenons l'instruction : **ORRS R1,R7** qui réalise $R1 \leftarrow R1 \text{ ou } R7$
- Ce que nous dit la documentation ARM

Thumb Instruction Details

A6.7.91 ORR (register)

Logical OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb ISA.

ORRS <Rdn>, <Rm>

ORR<c> <Rdn>, <Rm>

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	0	0	0	0	1	1	0	0	Rm	Rdn
---	---	---	---	---	---	---	---	---	---	----	-----

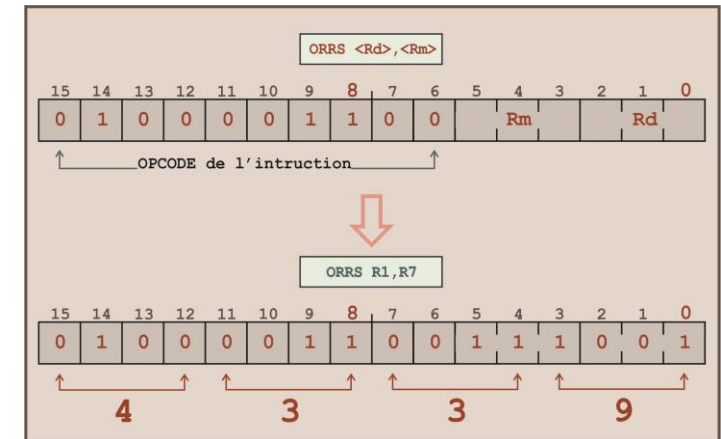
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock(); (shift_t, shift_n) = (SRTYPE_LSL, 0);

Encoding T2 ARMv7-M

ORR{S}<c>.W <Rd>, <Rn>, <Rm>{, <shift>}

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	0	1	0	1	0	0	0	1	0	S	Rn	(0)	imm3	Rd	imm2
---	---	---	---	---	---	---	---	---	---	---	---	---	----	-----	------	----	------



Disassembly window showing the instruction ORRS R1,R7 at address 0x0800041C.

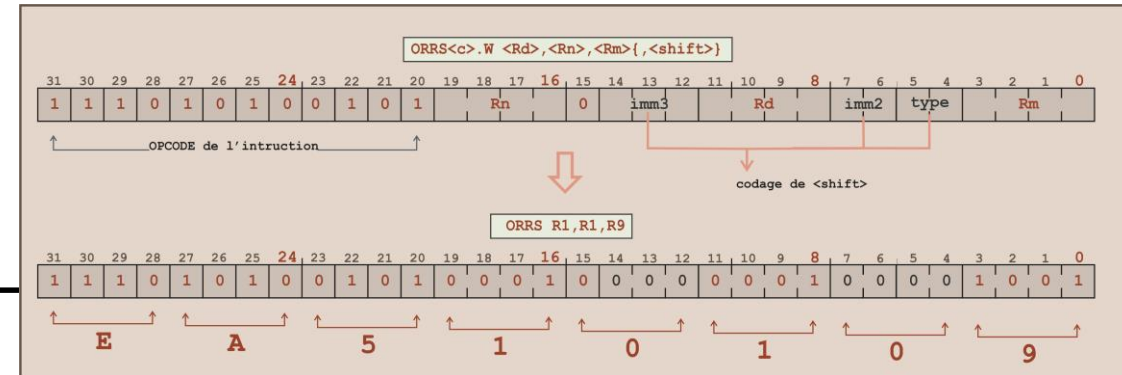
Address	Disassembly	Comment
0x08000416	2000	DCW 0x2000
0x08000418	0008	DCW 0x0008
0x0800041A	2000	DCW 0x2000
61:		ORRS R1,R7
62:	0x0800041C 4339	ORRS r1,r1,r7

Adresse où est stockée l'instruction

Codage de l'instruction

Comment est codé l'opérande dans l'instruction (2)

- Que se passe t'il pour coder ORRS R1, R9 ?
- Passage à un codage « T2 » sur 32 bits
 - + Codage des 3 registres sur 4 bits (R0 à R15)
 - + Codage d'un « shift » ... à voir plus tard



- 6.6.7.66 LDRT
- 6.6.7.67 LSL (immediate)
- 6.6.7.68 LSL (register)
- 6.6.7.69 LSR (immediate)
- 6.6.7.70 LSR (register)
- 6.6.7.71 MCR, MCR2
- 6.6.7.72 MCRR, MCCR2
- 6.6.7.73 MLA
- 6.6.7.74 MLS
- 6.6.7.75 MOV (immediate)
- 6.6.7.76 MOV (register)
- 6.6.7.77 MOV (shifted register)
- 6.6.7.78 MOVT
- 6.6.7.79 MRC, MRC2
- 6.6.8.0 MRRC, MRRC2
- 6.6.7.81 MRS
- 6.6.7.82 MSR (register)
- 6.6.7.83 MUL
- 6.6.7.84 MVIN (immediate)
- 6.6.7.85 MVIN (register)
- 6.6.7.86 NEG
- 6.6.7.87 NOP
- 6.6.7.88 ORN (immediate)
- 6.6.7.89 ORN (register)
- 6.6.7.90 ORR (immediate)
- 6.6.7.91 ORR (register)
- 6.6.7.92 PLD, PLDW (immediate)
- 6.6.7.93 PLD (literal)
- 6.6.7.94 PLD (register)
- 6.6.7.95 PLI (immediate, literal)
- 6.6.7.96 PLI (register)
- 6.6.7.97 POP
- 6.6.7.98 PUSH

Thumb Instruction Details

A6.7.91 ORR (register)

Logical OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

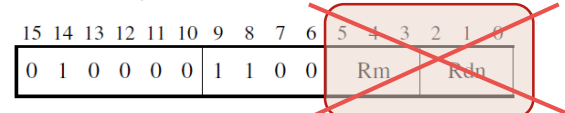
Encoding T1	All versions of the Thumb ISA.
--------------------	--------------------------------

ORRS $\langle R_{dn} \rangle, \langle R_m \rangle$

Outside IT block.

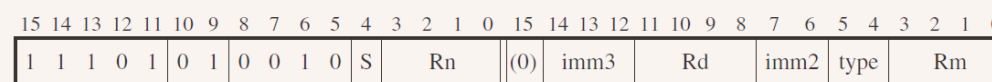
ORR<c> <Rdn>,<Rm>

Inside IT block.



```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();  
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

Encoding T2 ARMv7-M

$$\text{ORR}\{S\}\langle c \rangle.W \langle Rd \rangle, \langle Rn \rangle, \langle Rm \rangle \{, \langle \text{shift} \rangle\}$$


Disassembly

Address	Disassembly
0x08000416	DCW 0x2000
0x08000418	DCW 0x0008
0x0800041A	DCW 0x2000
61:	ORRS R1,R9
62:	
63:	
0x0800041C	EA510109 ORRS r1,r1,r9

Le choix des registres R0 à R7 peut minimiser la taille du code

Opérande registre avec **shift**

- Codage « T2 » de **ORRS** est :

ORRS<c>.W <Rd>, <Rn>, <Rm> { , <shift> }

- + <c> indique un conditionnement possible
- + .W indique que l'on force le codage sur 32 bits.
- + 3 opérandes "registre" **Rd Rn Rm**
- + **Rd Rn Rm** non forcément distincts

- Le **<shift>** est un opérateur logique qui s'applique sur le dernier registre (**Rm**) avant que soit réalisé l'instruction

Les **<shift>** possibles

LSL #n : décalage logique à gauche de **n** bits

LSR #n : décalage logique à droite de **n** bits

ASR #n : décalage arithmétique droite de **n** bits

ROR #n : Rotation à droite de **n** bits

RRX : Rotation d'un bit à droite via le fanion **C**

Que fait :

ADD R1,R1,R1, LSL #3



L'opérande **immédiat**

- Il ne peut être que source
+ 2^{ème} ou 3^{ème} opérande
- Il débute par un #
- Peut s'exprimer dans différentes bases
- Il est codé avec l'instruction

Les « bases » possibles

(Ajouter 65 à R3)

- En décimal (défaut)

ADD R3,#65

- En hexadécimal

ADD R3,#0x41

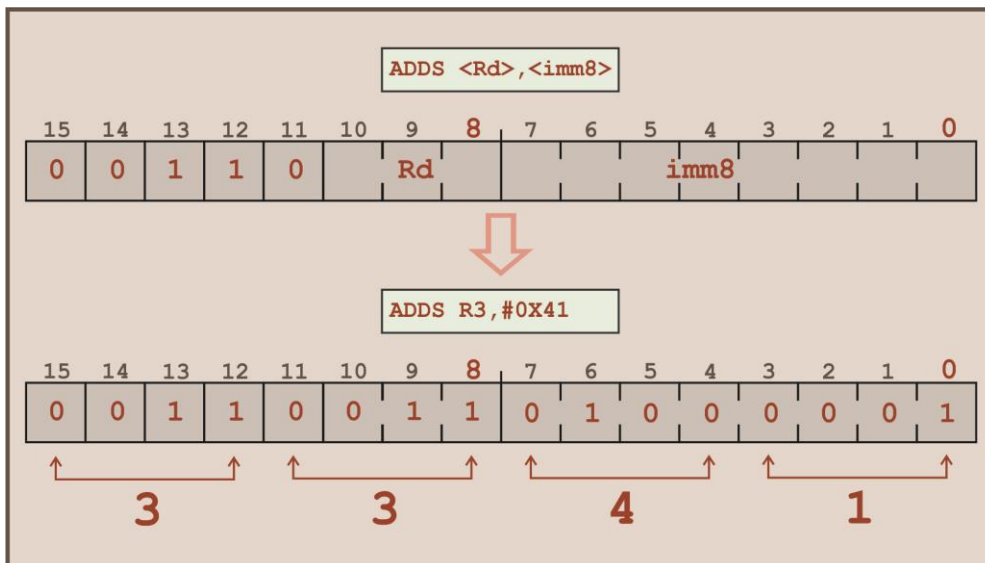
- En binaire

ADD R3,#2_1000001

- En ASCII :

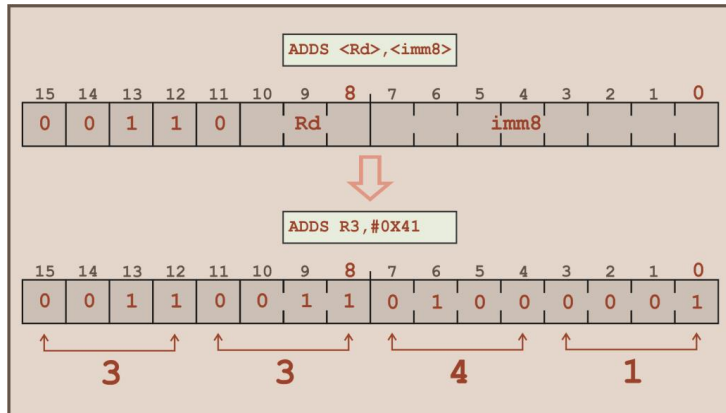
ADD R3,#'A'

Dans une base quelconque : #n_????

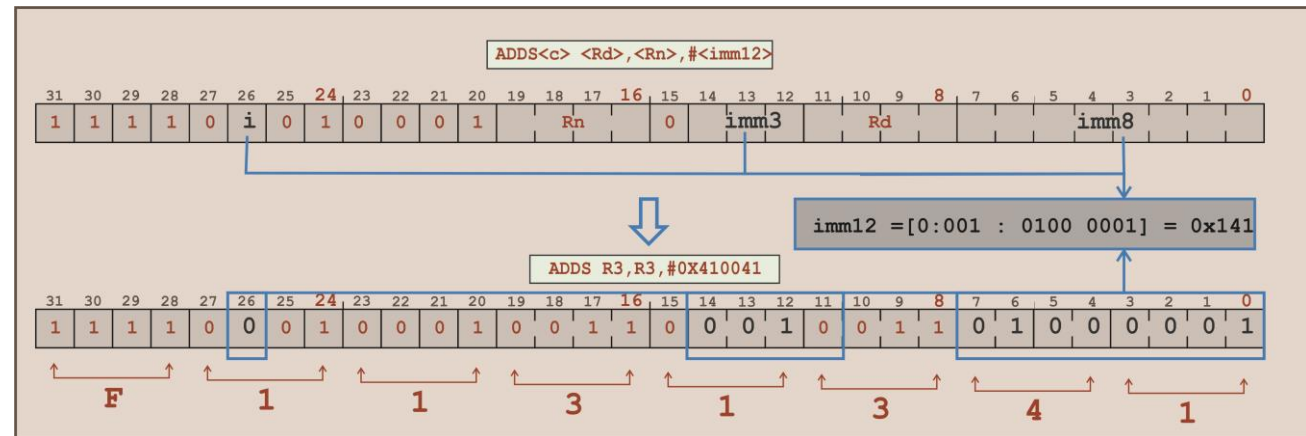


Problème de codage de l'opérande immédiat

- Dans le codage 16 bits de ADD et opérande « immédiat » :
- Passage à un codage sur 32 bits :



Impossible si $\text{imm8} > 255$



- Le codage (bizarre !) de la valeur est sur 12 bits
- Problème **non complètement résolu**

Comment faire pour coder une valeur immédiate codée sur 32 bits ?

Solution : passage à la *Literal Pool*

- Il faut passer à un codage 64 bits : 32 bits pour l'instruction + 32 bits pour la valeur....
- Pas exactement ce qui est fait.
- La valeur 32 bits est codé en mémoire CODE plus loin dans la mémoire
 - + Création d'une littéral pool
- L'accès se fait pas une lecture de la mémoire ..
 - + Charge un registre **Rx** avec cette lecture
 - + On réalise l'opération (ADD par exemple avec ce registre)

En pratique

Exemple charger un registre avec une valeur

- Si la valeur est inférieur à 255
 - + pas de souci

MOV R5, #-34

- Si la valeur est supérieur à 255
 - + utilisation d'une **pseudo syntaxe**:

LDR R11, =0x1234ABCD





L'opérande *mémoire*

- Comment se traduit en L.A. le code

```
INT MAvALEUR ;  
...  
MAVALEUR++ ;
```

- Si variable locale : utilisation possible d'un registre **Rx** (ou la pile système)
- Si variable globale ou statique
 - + => en L.A il faudra prévoir la déclaration/réservation mémoire
 - À voir plus tard
 - + l'utilisation par accès à la mémoire - Pas simple car :
 - Architecture **LOAD/STORE** de ARM
 - Accès obligatoirement par **INDIRECTION**

L'opérande *mémoire* : architecture **LOAD/STORE**

- Seules 2 familles peuvent accéder à la mémoire
- Elle ne peuvent faire qu'un transfert mémoire à registre.
- Si on veut par exemple incrémenter une valeur stockée en mémoire :

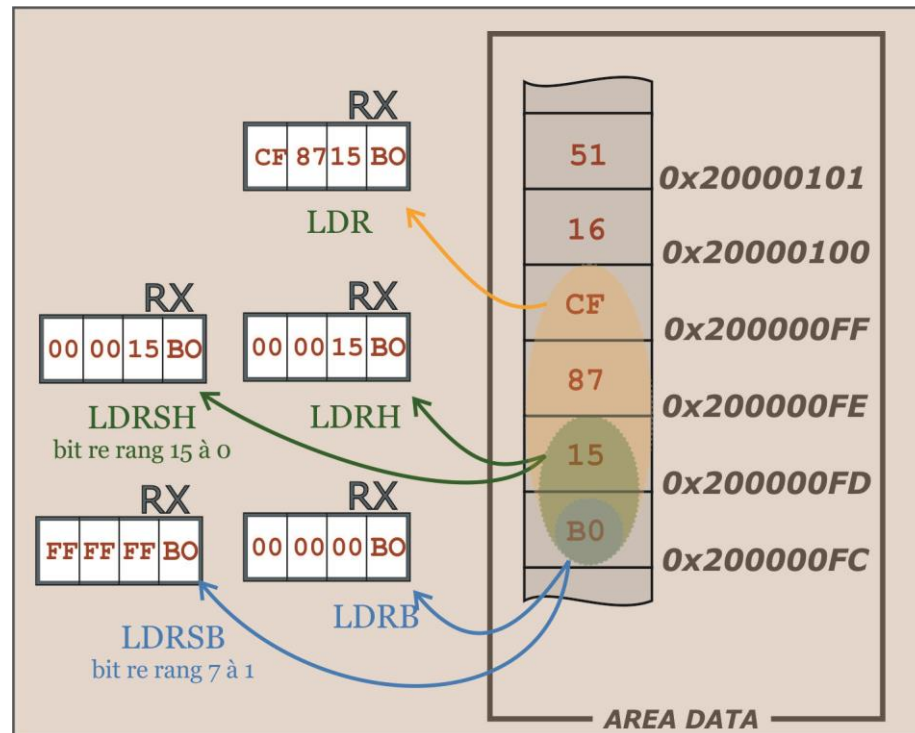
```
MAVALEUR++ ;
```

- + Transférer la valeur dans un registre
- + Additionner le registre avec 1
- + Transférer la valeur du registre en mémoire

Les différentes formes

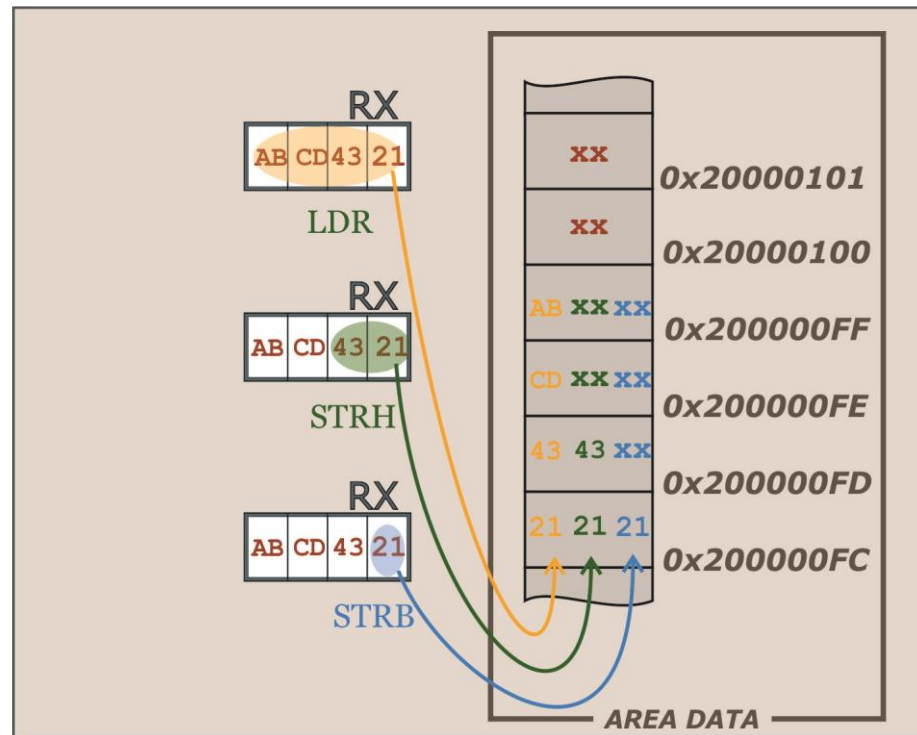
- En lecture (transfert mémoire vers registre)
 - 32 bits → 32bits **LDR**
 - 16 bits → 32bits **LDRH / LDRSH**
 - 8 bits → 32bits **LDRB / LDRSB**
- En écriture (transfert registre vers mémoire)
 - 32 bits → 32bits **STR**
 - 16 bits → 32bits **STRH**
 - 8 bits → 32bits **STRB**

L'opérande *mémoire* : architecture **LOAD/STORE**



- En lecture (transfert mémoire vers registre)
 - 32 bits → 32bits **LDR**
 - 16 bits → 32bits **LDRH / LDRSH**
 - 8 bits → 32bits **LDRB / LDRSB**
- **LDRSH** (resp. **LDRSB**) : extension du bit de signe du 1/2 mot (resp. octet) :
 - + la valeur codée sur 32 bits doit représenter la même valeur que celle codée en 16 (resp. 8) bits

L'opérande *mémoire* : architecture **LOAD/STORE**



- En écriture (transfert registre vers mémoire)
 - 32 bits → 32bits **STR**
 - 16 bits → 32bits **STRH**
 - 8 bits → 32bits **STRB**

L'opérande *mémoire* : accès **INDIRECT**

- Syntaxe : les **[]** indique l'indirection
+ on récupère le contenu qui se trouve à l'adresse donnée entre parenthèse
- Idéalement on devrait pouvoir faire :

```
LDRSH R3, [0x200000FC]
```

- Ou en supposant que **MaValeur** soit un variable

```
LDR R4, MAVALEUR
```

- En pratique ce n'est pas possible. Il faut passer par un registre

Concrètement :

- Chargement d'une adresse absolue

```
LDR R0, =0x200000FC  
LDRSH R3, [R0]
```

- Chargement d'une adresse de variable

```
LDR R11, =MAVALEUR  
LDR R4, [R11]
```





Opérande mémoire : @dressage simple

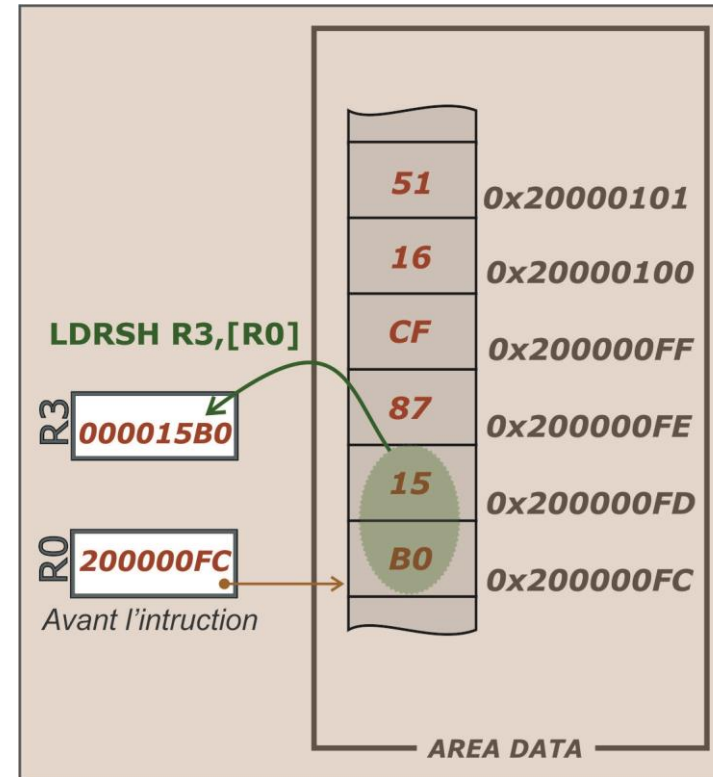
- Différentes façon de faire l'indirection
= **Mode d'adressage**

- Adressage indirect simple

```
LDR Rx, [Ry]  
STR Rx, [Ry]
```

- Exemple précédent :

```
LDR R0, =0x200000FC  
LDRSH R3, [R0]
```



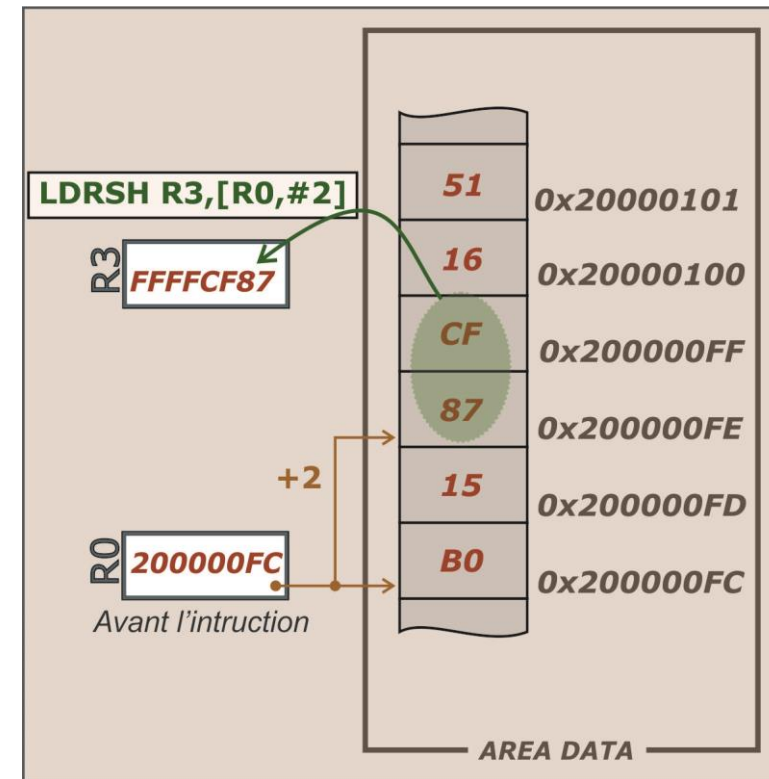
Adressage simple + déplacement

- Ajout d'un déplacement de **n** octets pour accéder à l'adresse mémoire de base

```
LDR Rx, [Ry, #n]  
STR Rx, [Ry, #n]
```

- Le registre **Ry** n'est pas modifié
- Usage :
 - + structure de données
- Exemple

```
LDR R0, =0x200000FC  
LDRSH R3, [R0, #2]
```



Adressage indexé

- Ajout d'un déplacement correspondant au contenu d'un troisième registre

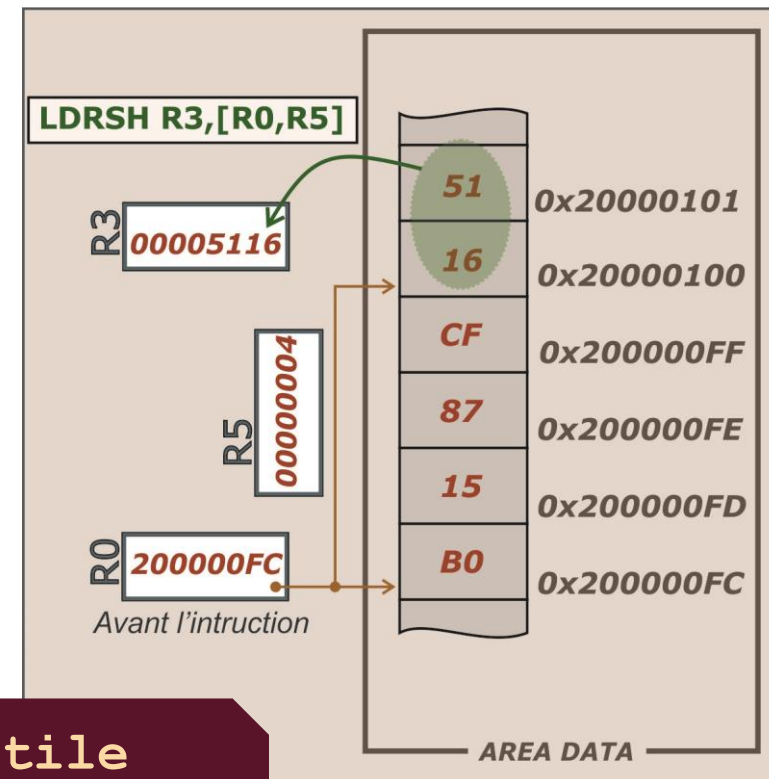
LDR Rx, [Ry, Rz]
STR Rx, [Ry, Rz]

- Les registre **Ry** et **Rz** ne sont pas modifiés
- Usage :
 - + tableau
- Exemple :

```
LDR R0, =0x200000FC
MOV R5, #4
LDRSH R3, [R0, R5]
```

Variation utile

LDR Rx, [Ry, Rz, LSL #2]



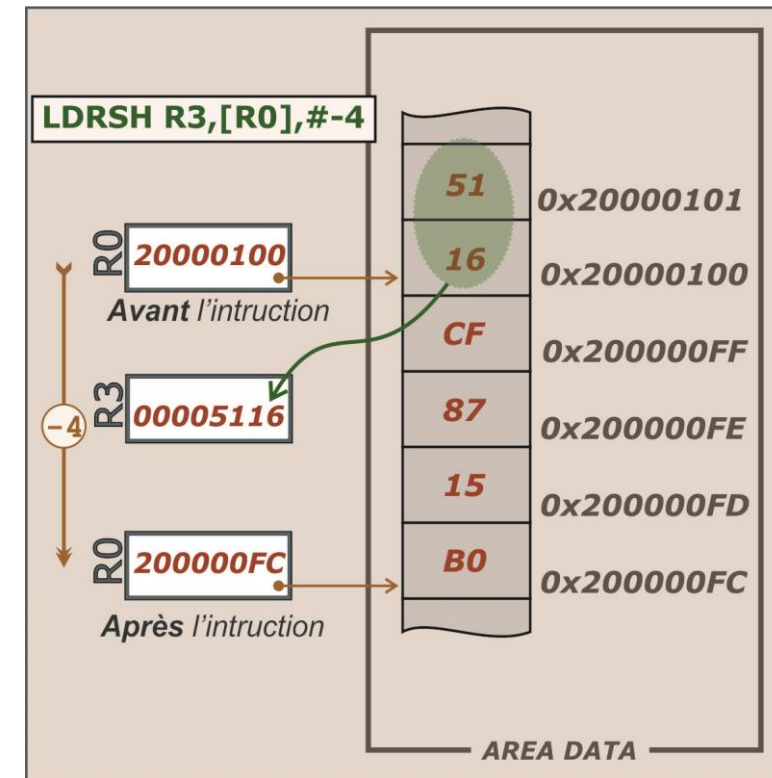
Adressage Post-déplacé

- Le pointeur (Ry) est modifié APRES le transfert de la quantité **n**

LDR Rx, [Ry], #n
STR Rx, [Ry], #n

- Le registre **Ry** est modifié
- Usage :
 - + pointeur glissant sur zone de données
- Exemple (lecture post-décrémenté):

```
LDR R0, =0x20000100
LDRSH R3, [R0], #-4
```



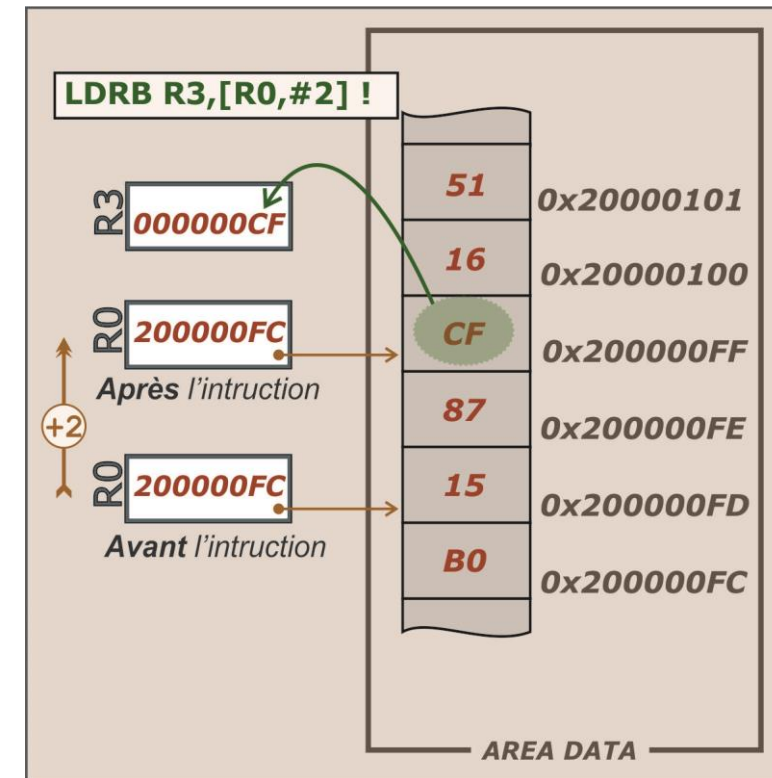
Adressage Pré-déplacé

- Le pointeur (Ry) est modifié AVANT le transfert de la quantité **n**

```
LDR Rx, [Ry, #n] !  
STR Rx, [Ry, #n] !
```

- Le registre **Ry** est modifié
- Usage :
 - + pile à pré-déplacement (pile système)
- Exemple (lecture pré-incrémenté):

```
LDR R0, =0x200000FD  
LDRB R3, [R0, #2] !
```





Retour sur la pseudo syntaxe LDR Rx,=....

■ Désassemblage de :

```
LDR R7,=0x20  
LDR R11,=0x1234ABCD
```

■ 1^{er} ligne traduit par

- + MOV R7,#0x14
- + codage 32 bits : F0F40714
- + adresse stockage : 0x0800041C

■ Si la valeur est < 255

- + le **LDR Rx,=** est traduit en **MOV**
- + autant utiliser un **MOV** !!!

Disassembly			
0x08000412	2000	DCW	0x2000
0x08000414	0004	DCW	0x0004
0x08000416	2000	DCW	0x2000
0x08000418	0008	DCW	0x0008
0x0800041A	2000	DCW	0x2000
62:		LDR R7,=20	
0x0800041C	F0F40714	MOV	r7,#0x14
63:		LDR R11,=0x1234ABCD	
64:			
0x08000420	F8DFB004	LDR.W	r11,[pc,#4] ; @0x08000428
65: Finir	B Finir		; Boucle infinie
0x08000424	E7FE	B	0x08000424
0x08000426	0000	DCW	0x0000
0x08000428	ABCD	DCW	0xABCD
0x0800042A	1234	DCW	0x1234
0x0800042C	E002	B	0x08000434

Retour sur la pseudo syntaxe LDR Rx,=....

■ Désassemblage de :

```
LDR R7,=0x20  
LDR R11,=0x1234ABCD
```

■ 2^{ème} ligne traduit par

- + LDR R11, [PC, #4]
- + accès mémoire code (PC)
- + indirection avec déplacement
- + la quantité à mettre dans R11 est stockée plus loin (en *zone literal pool*)

Disassembly

Address	Offset	OpCode	Comment
0x08000412	2000	DCW	0x2000
0x08000414	0004	DCW	0x0004
0x08000416	2000	DCW	0x2000
0x08000418	0008	DCW	0x0008
0x0800041A	2000	DCW	0x2000
62:		LDR R7,=20	
0x0800041C	F04F0714	MOV	r7,#0x14
63:		LDR R11,=0x1234ABCD	
64:			
0x08000420	F8DFB004	LDR.W	r11,[pc,#4] ; @0x08000428
65: Finir	B Finir		; Boucle infinie
0x08000424	E7FE	B	0x08000424
0x08000426	0000	DCW	0x0000
0x08000428	ABCD	DCW	0xABCD
0x0800042A	1234	DCW	0x1234
0x0800042C	E002	B	0x08000434

La syntaxe LDR,= n'existe pas pour le processeur !

Résumé des opérandes

■ Opérandes registres

- + R0 à R13
- + autres registres (PC, SP, LR) accessibles mais usage réservé (\Rightarrow danger!)

■ Opérandes immédiats

- + débutent avec un #
- + codés « dans » le code de l'instruction
- + pb si valeur $> 255 \Rightarrow$ utilisation d'un pseudo syntaxe `LDR, =`

■ Opérandes mémoires

- + uniquement deux familles d'instruction **LDR** et **STR**
- + indirection : utilisation d'un registre pointeur
- + mode d'adressage :

instr Rt, [Rn]

instr Rt, [Rn, # $\pm imm8$]

instr Rt, [Rn, Rm]

instr Rt, [Rn, Rm, LSL # $\langle imm2 \rangle$]

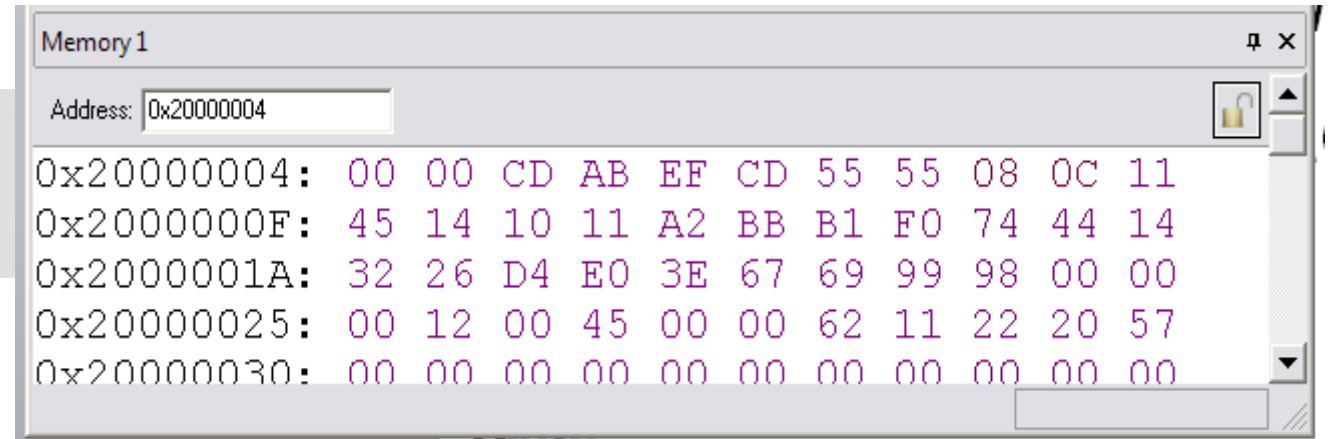
instr Rt, [Rn], # $\pm imm8$

instr Rt, [Rn, # $\pm imm8$] !





Exercice applicatif



Instruction	R0 (après exécution)	L'autre registre (après exécution)
LDR R0, = 0x2000000F		/
LDRB R1,[R0]		
LDRH R2,[R0]		
LDRB R3,[R0,#4]		
LDRB R4,[R0],#-2		
LDRB R5,[R0, #-6] !		
STRB R6,[R0]		
LDRSB R7,[R0,#1]		