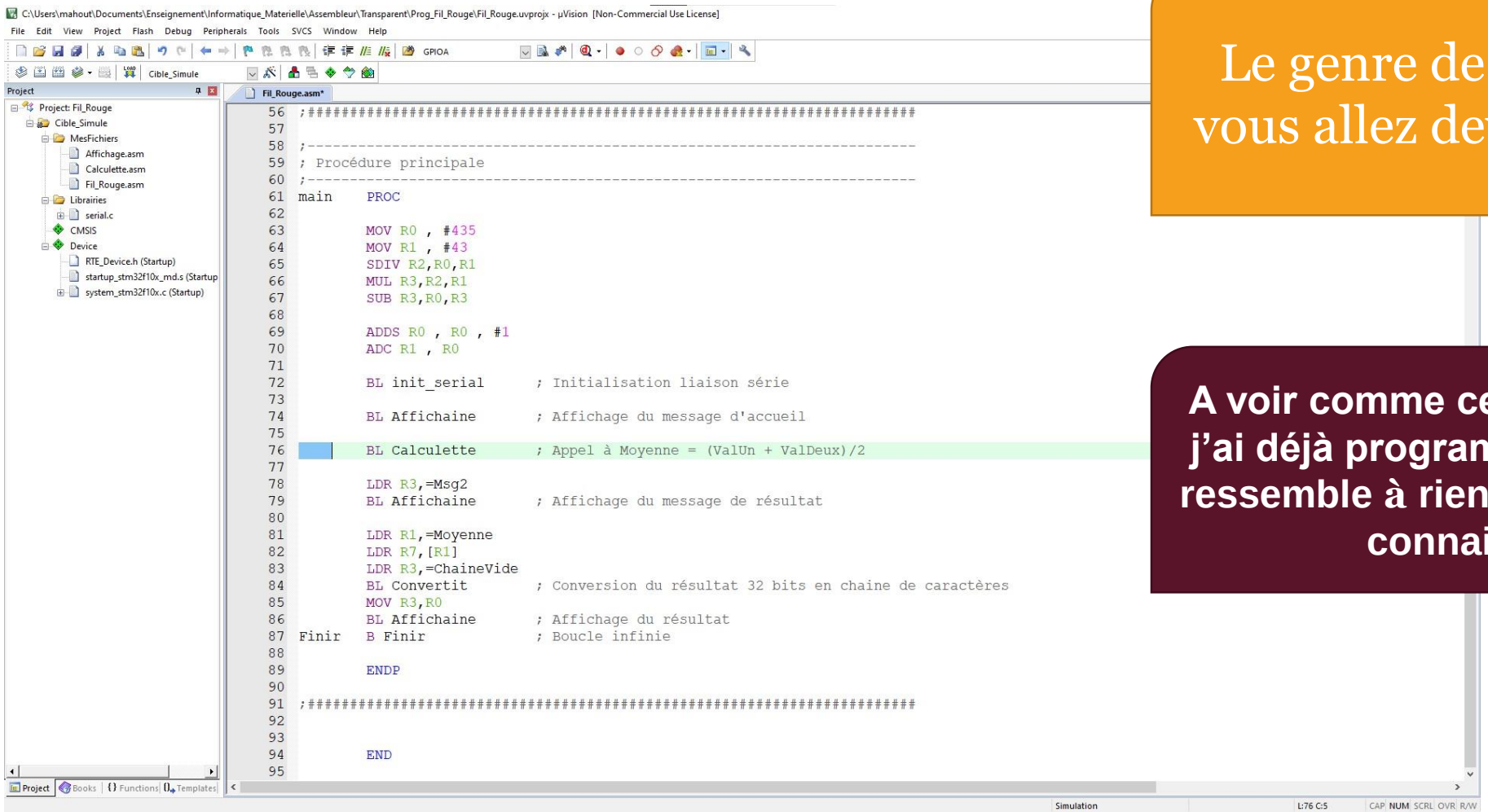
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Premier Pas

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

Extrait de listing :

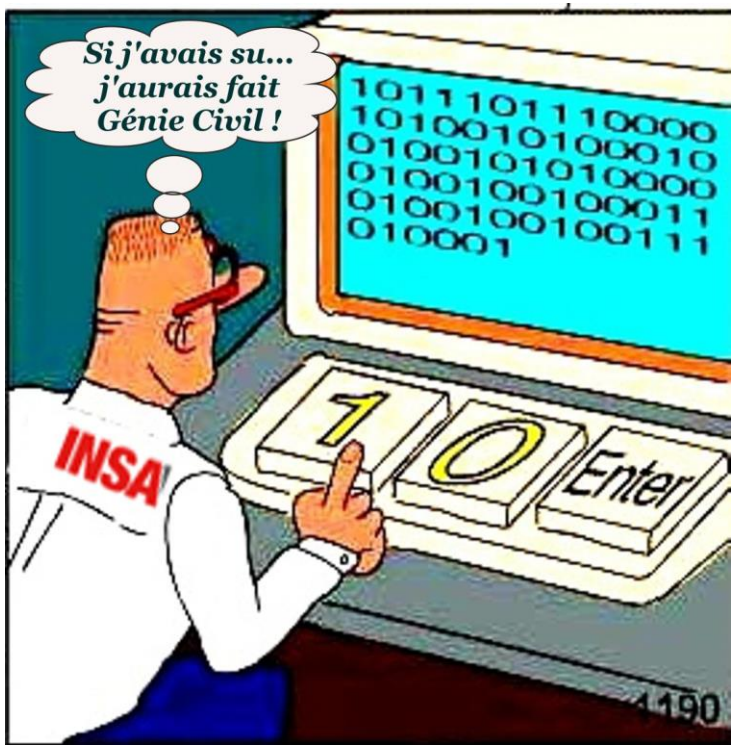


Le genre de chose que vous allez devoir écrire !

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
- ⇒ 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 héxadécimal : `E3A010AA`
  - en binaire : `11100011101000000001000010101010`

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 …) !
- 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

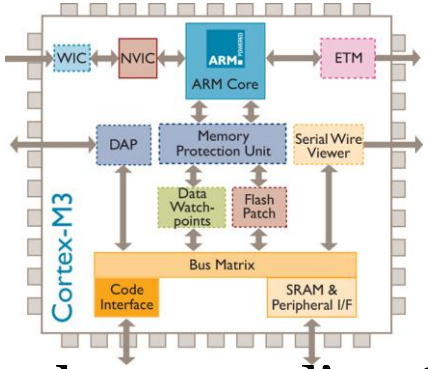
- Efficacité : de façon très temporaire dans un programme C

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

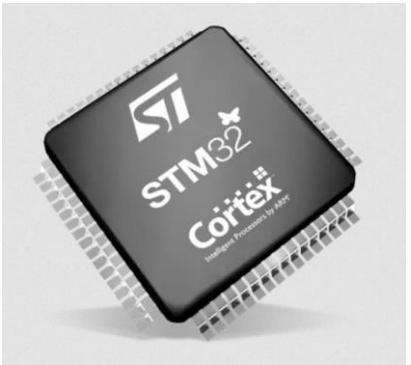
Objectifs

- 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 µvision
- Commencer à comprendre quelques mécanismes de la problématique *Temps Réel* et de la programmation de périphériques

Application au Cortex- M3



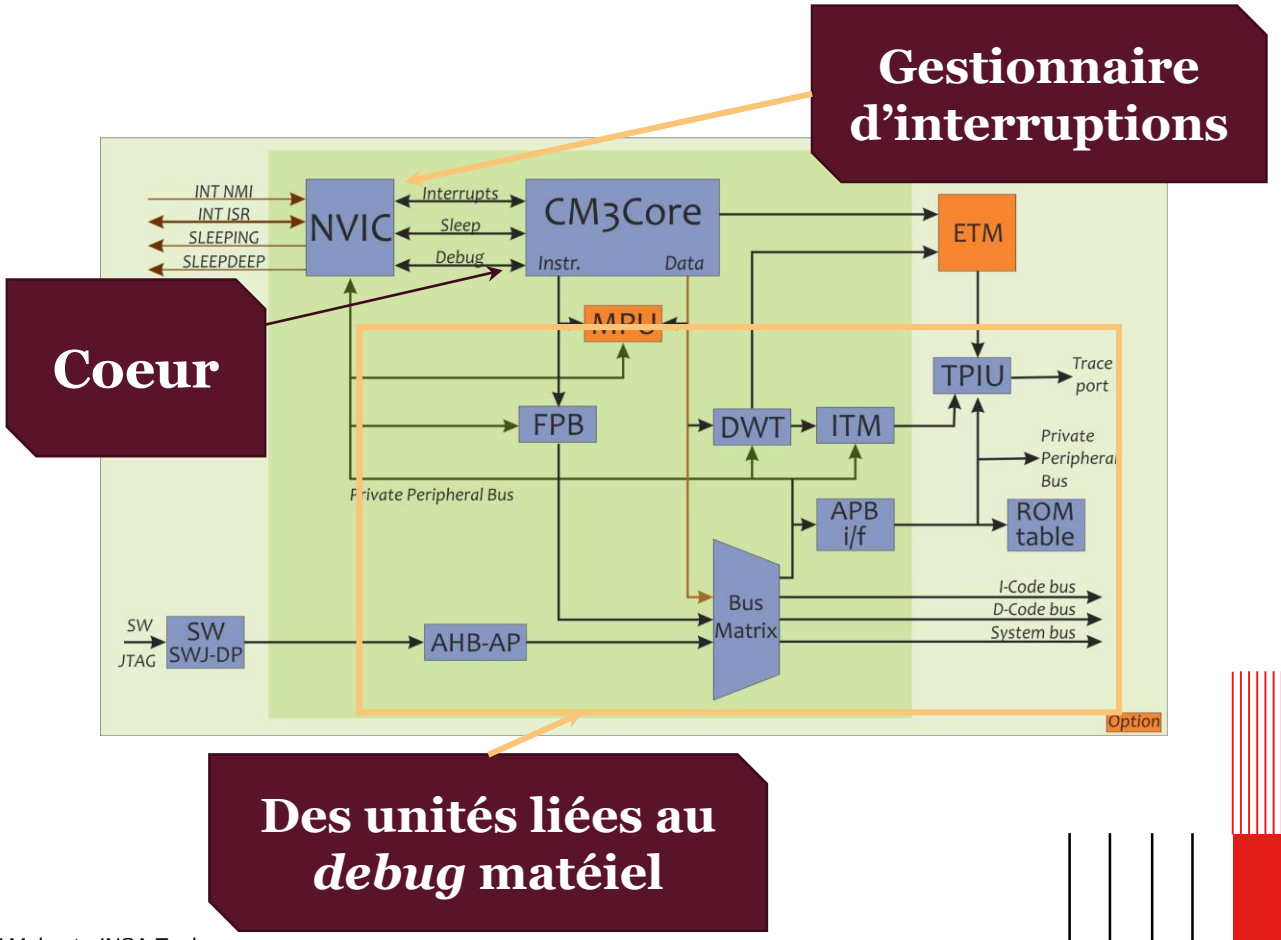
qui est le **cœur** d'un STM32 :



Le Cortex rapidement

- ARM
  - Société anglaise : concepteur uniquement (brevets)
  - **A**dvanced **R**isc **M**achines (*reduced instruction-set computer)*
  - Cortex ⇒ 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

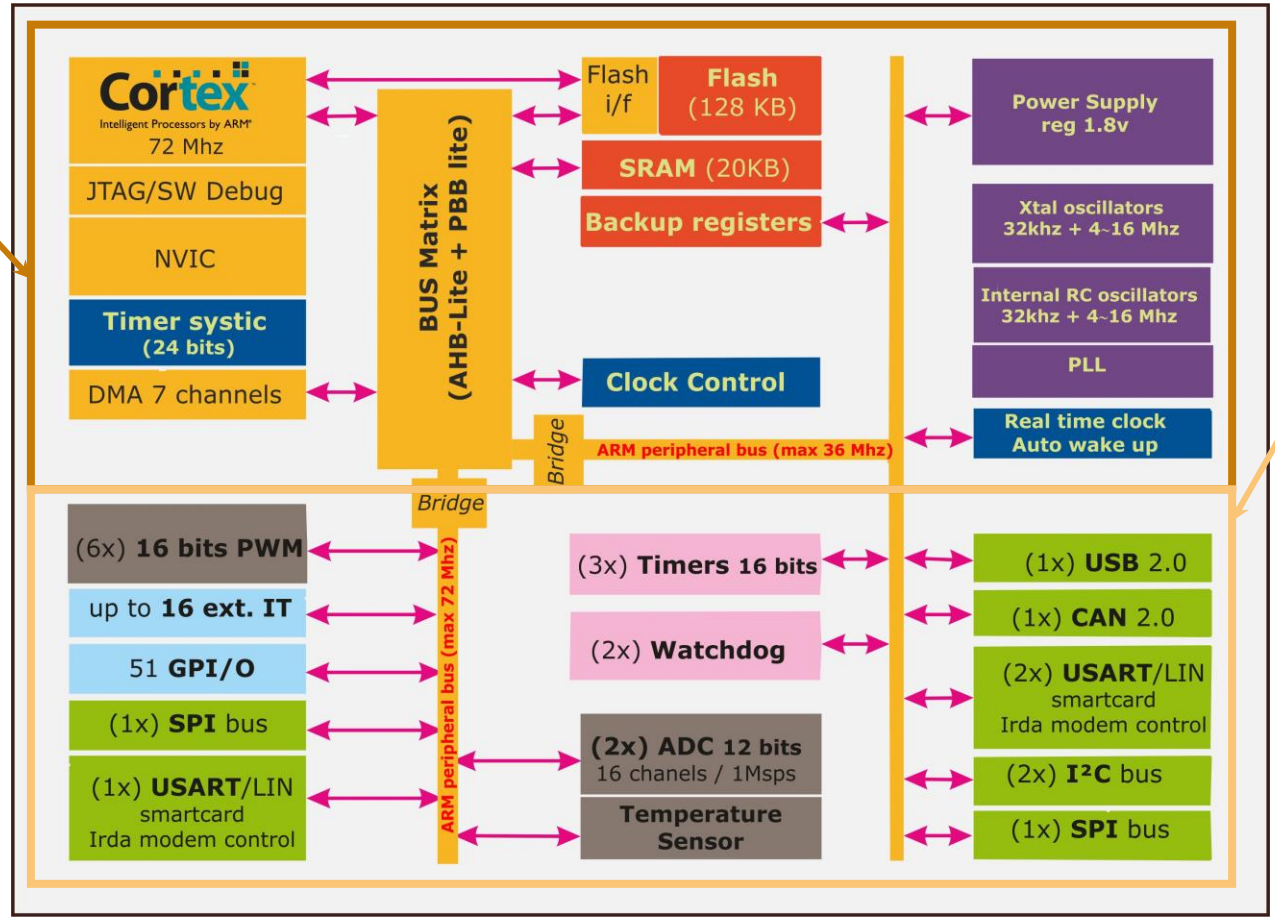
Les unités Debug :

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



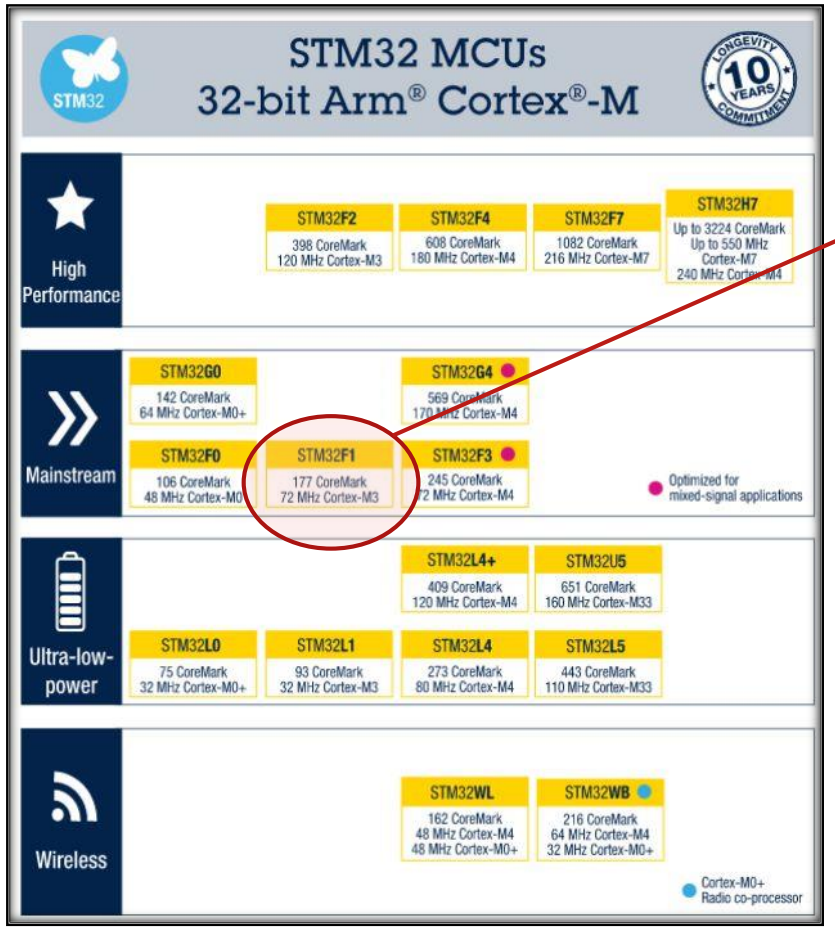
« Micro machines » périphériques

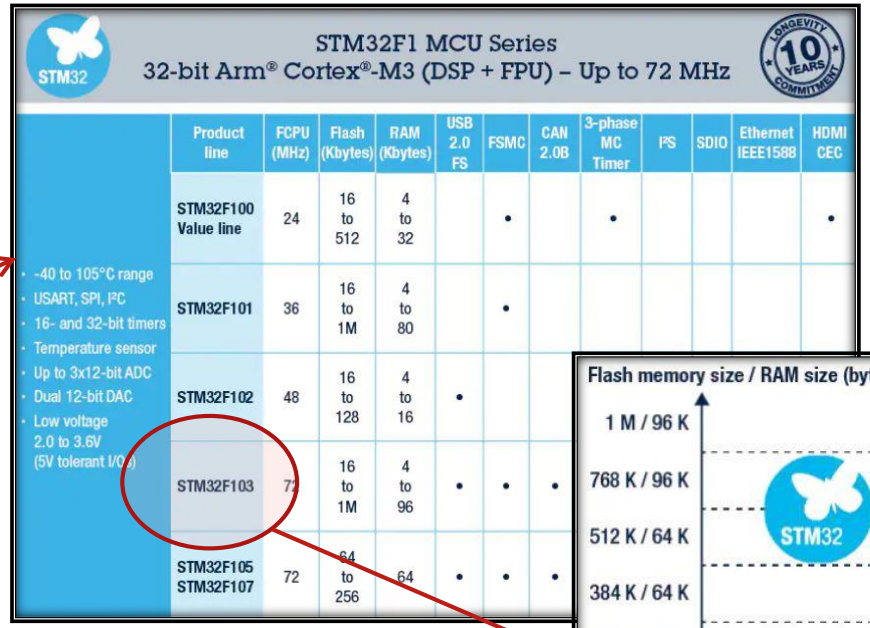
- \+ E/S binaires (bleu)
- \+ E/S analogiques (brun)
- \+ Mesure du temps (rose)
- \+ Bus numériques (vert)

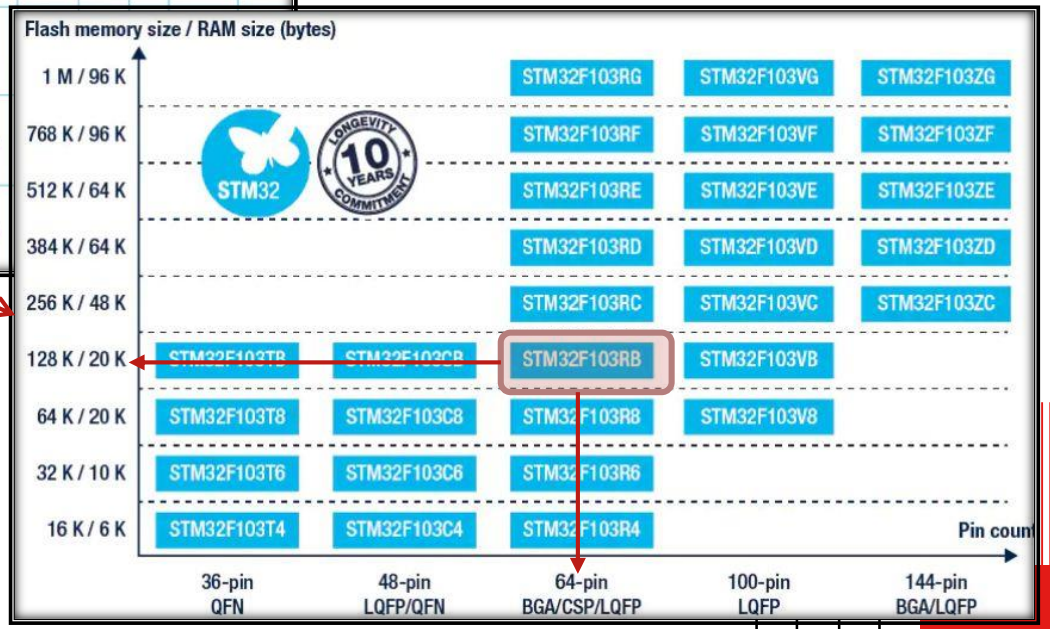
Choix du fondeur

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

STM32 : une bien grande famille



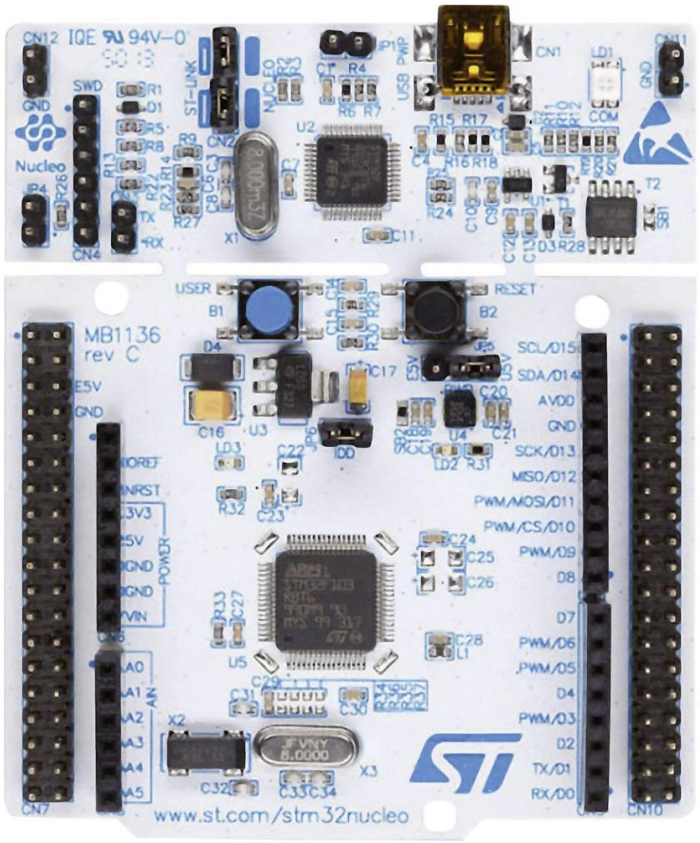




Le STM2 sur lequel on va faire les TP STM32F103RB

Les TP :

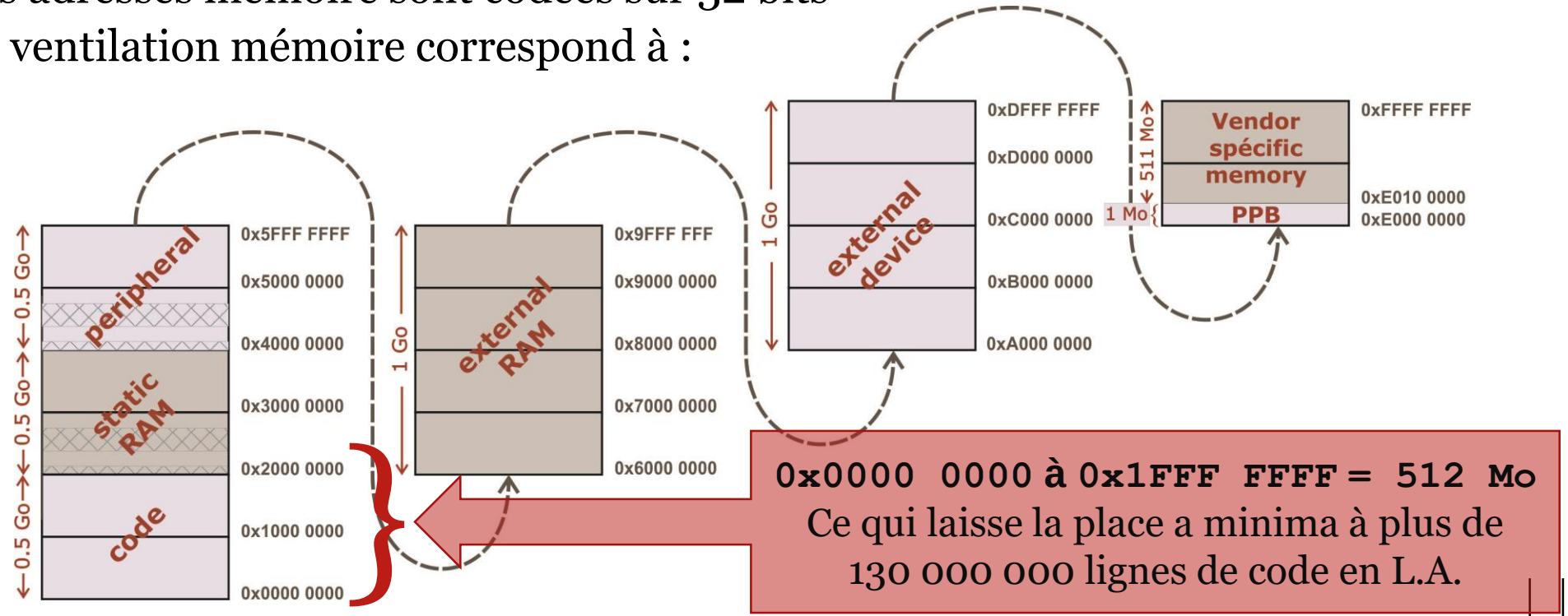
Une carte nucléo



Une roue magique

Continuons à ouvrir le capot :

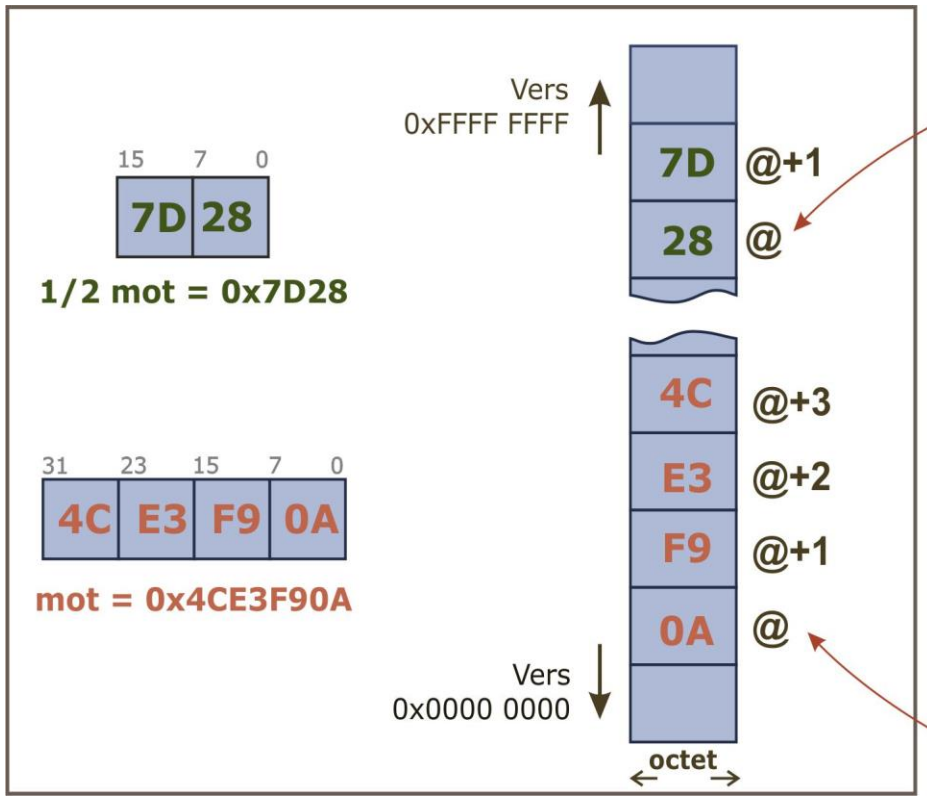
- Le Cortex M3 est une architecture de type Harvard
  - + séparation de la mémoire CODE et de la mémoire DATA
  - + les adresses mémoire sont codées sur 32 bits
  - + la ventilation mémoire correspond à :



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 ½ mot (16 bits)
    - Adresse paire :
      bit de poids 0 est à 0
    - exemple : `0x40002462`

adresse de base paire

  - + accès à un ½ 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...
❶	❷	❸	❹

Les 4 champs :

```
                MOV         R3,R0
                BL          Affichaine          ;Affichage du résultat
Finir           B           Finir               ;Boucle infinie
```

[Etiquette]	mnémonique	[expr1][,expr2][,expr3]	; comments...
❶	❷	❸	❹

- ❶ - **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 1er colonne
- ❷ - **Instructions** : c'est ce que doit faire la CPU à cette étape
  - + l'ensemble des instructions : jeu d'instructions du processeur
- ❸ - **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
- ❹ - **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 ⇒ 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)

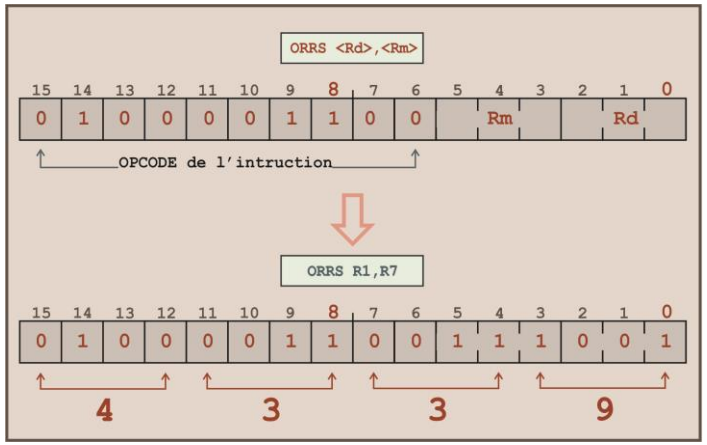
- Si **2** opérandes
  - `instr Rx,Ry`
  - **Rx** : registre destination qui reçoit le résultat
  - **Rx** et **Ry** : registres sources
- Si **3** opérandes
  - `instr Rx,Ry,Rz`
  - **Rx** : registre destination qui reçoit le résultat
  - **Ry** et **Rz** : registres sources

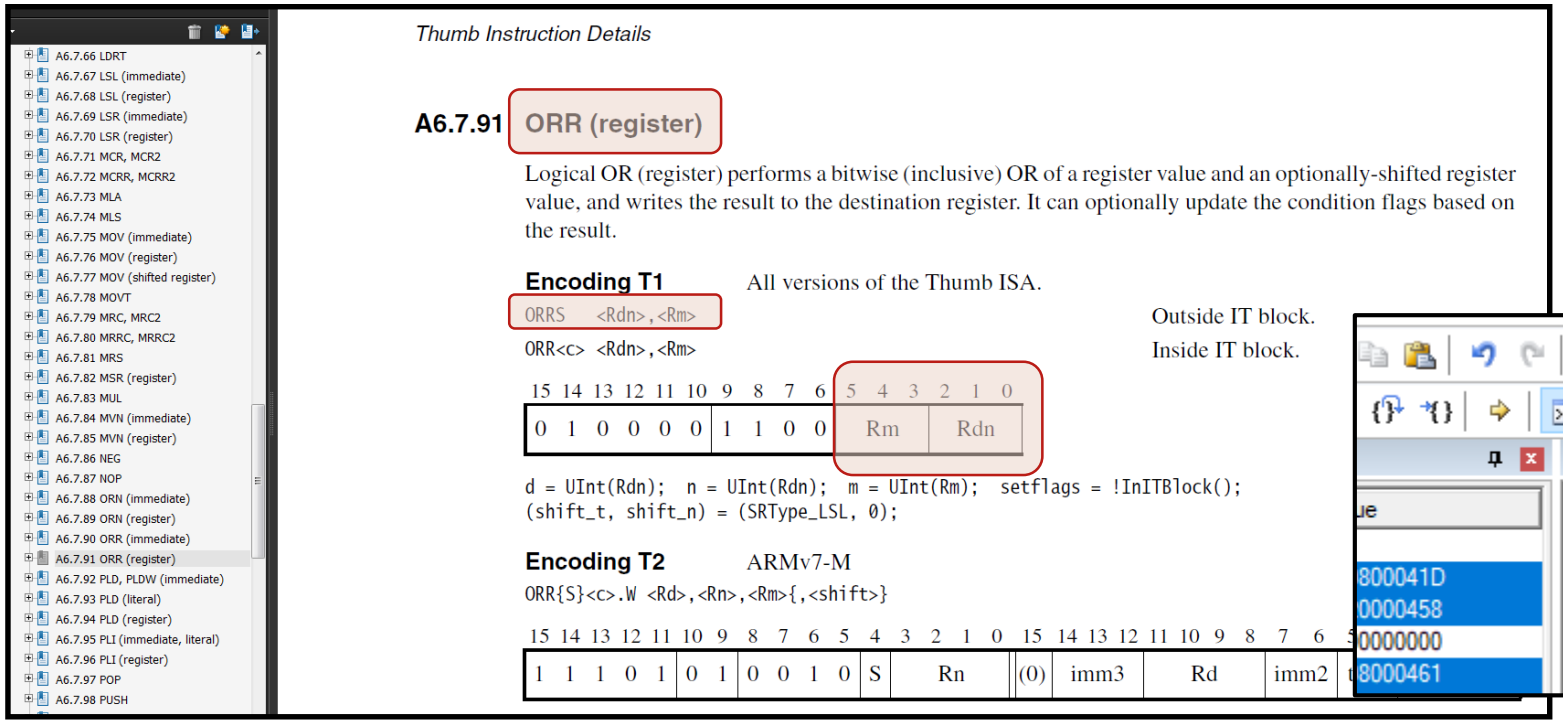
- Exemple :

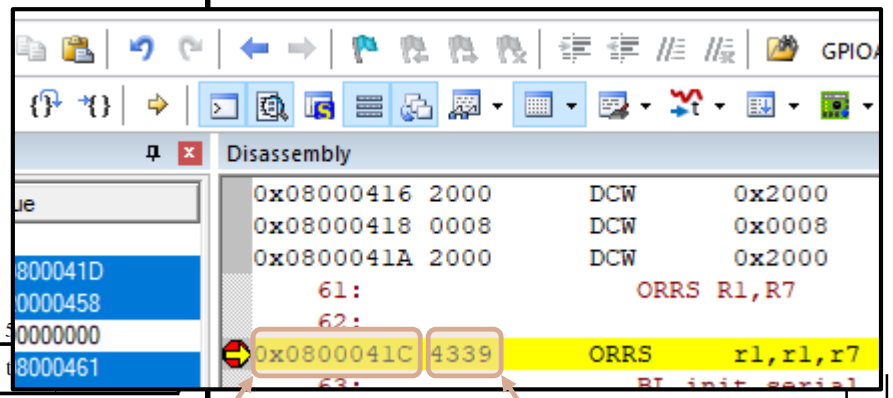
Ligne de code	Signification
`ADC R1,R0`	R1 reçoit R1 + R0 + « carry »
`SUB R3,R0,R3`	R3 reçoit R0 - R3
`LDR R7,[R1]`	R7 est chargé avec le contenu de l'adresse mémoire stockée dans R1
`BX LR`	Le processeur « saute » à l'adresse contenue dans LR

Comment est codé l'opérande dans l'instruction

- Prenons l'instruction : **ORRS R1,R7** qui réalise **R1 ← R1 ou R7**
- Ce que nous dit la documentation ARM





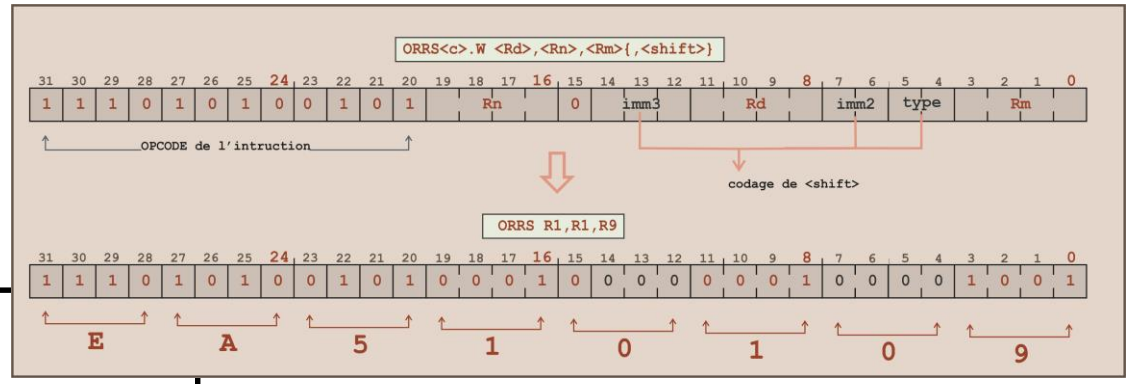


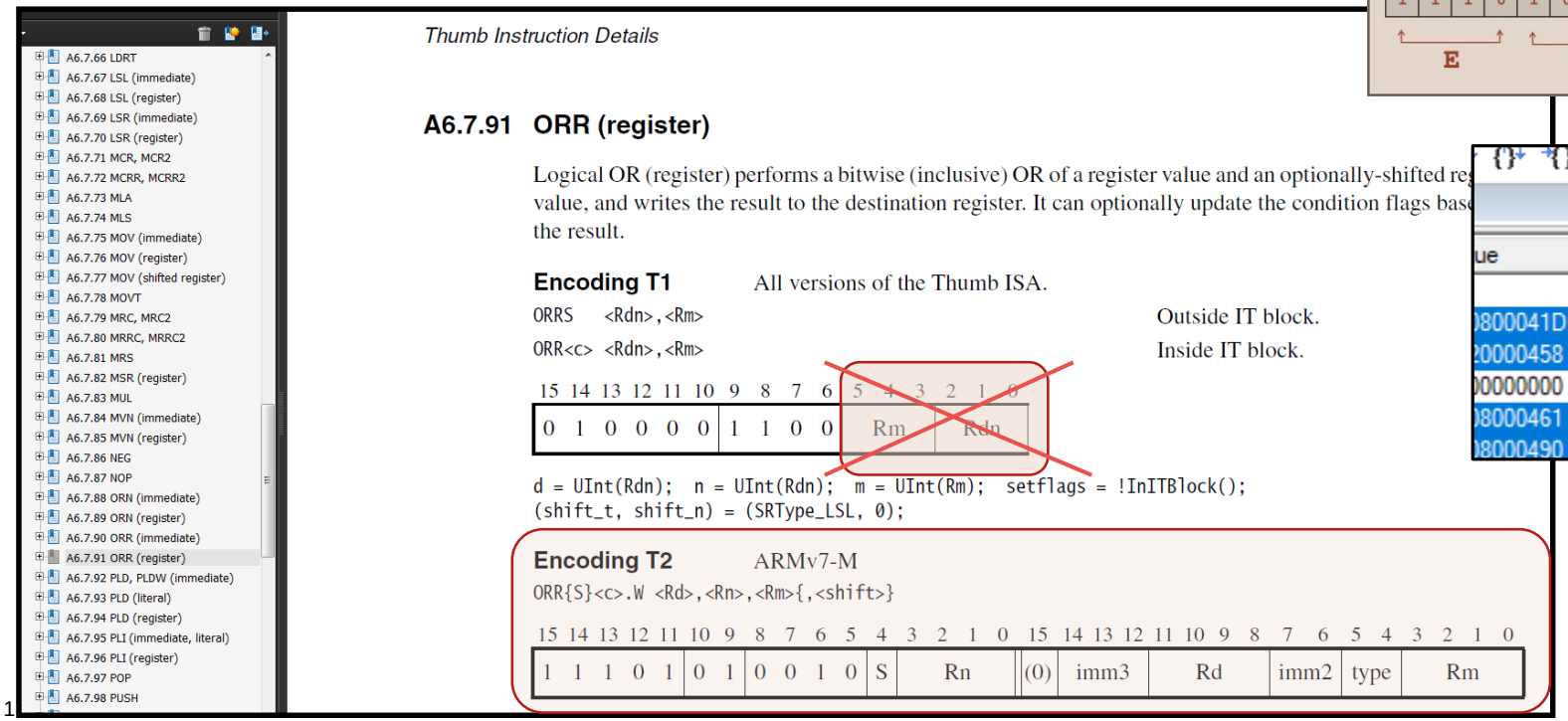
Adresse où est stockée l'instruction

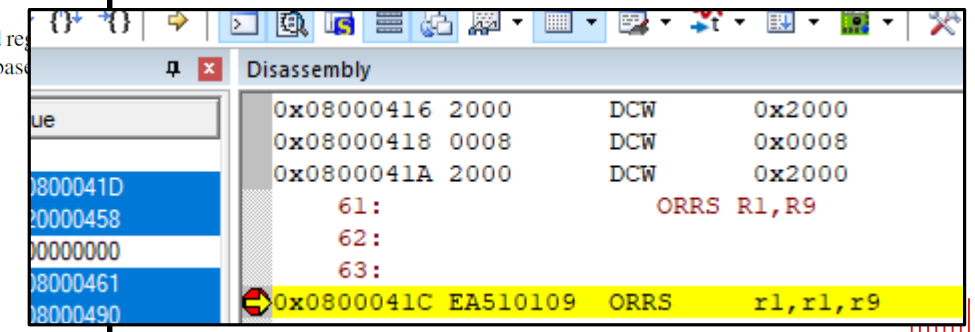
Codage de l'instruction

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







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

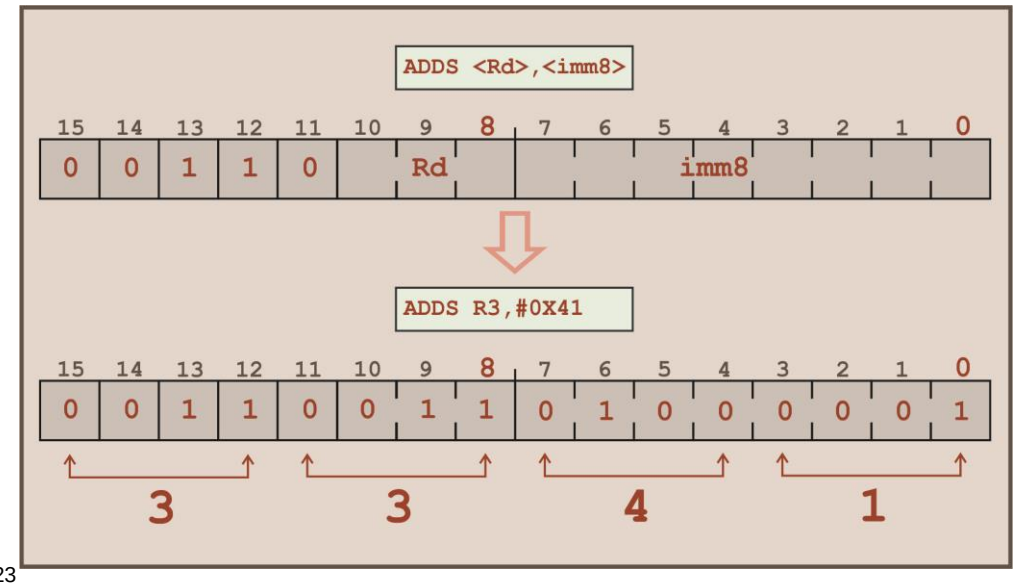
ROR #n : Rotation à droite de **n** bits

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

Que fait :
ADD R1,R1,R1, LSL #3

L'opérande immédiat

- Il ne peux être que source
  + 2ième ou 3ième 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 héxadécimal

  ADD R3,#0x41
- En binaire

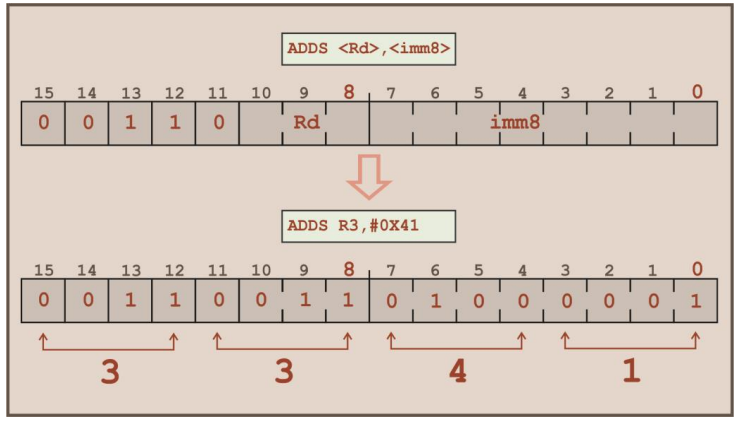
  ADD R3,#2_1000001
- En ASCII :

  ADD R3,#'A'

Dans une base quelconque : #n_?????

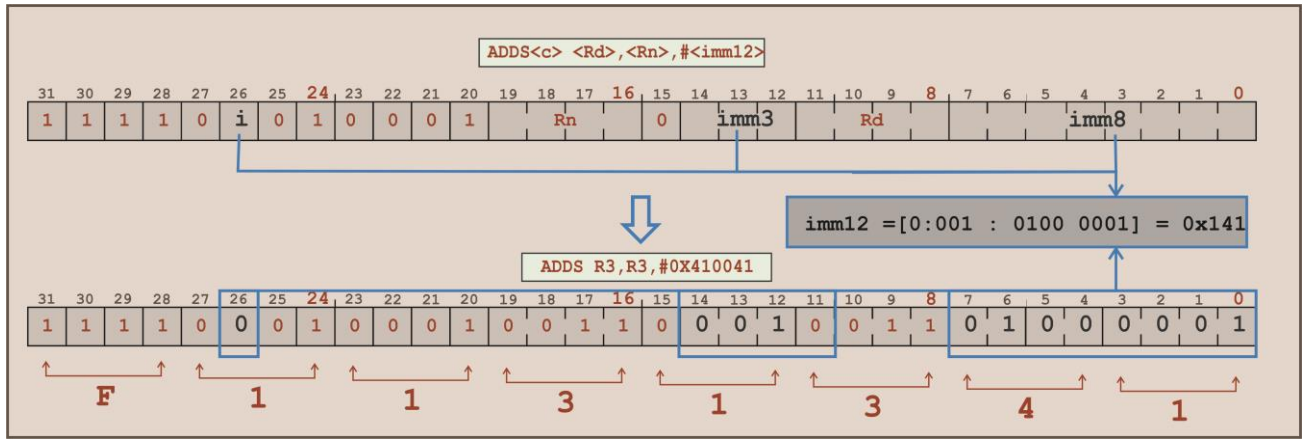
Problème de codage de l'opérande immédiat

- Dans le codage 16 bits de ADD et opérande « immédiat » :



Impossible si imm8 >255

- Passage à un codage sur 32 bits :



- 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 le 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 :

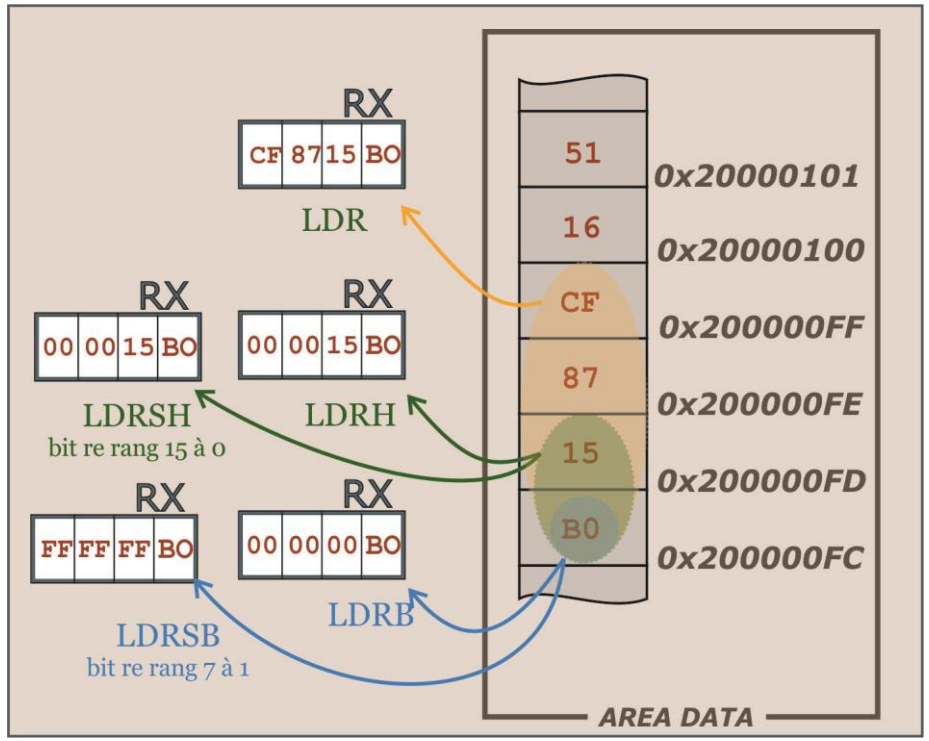
```
MAVALEUR++ ;
```

+ 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**

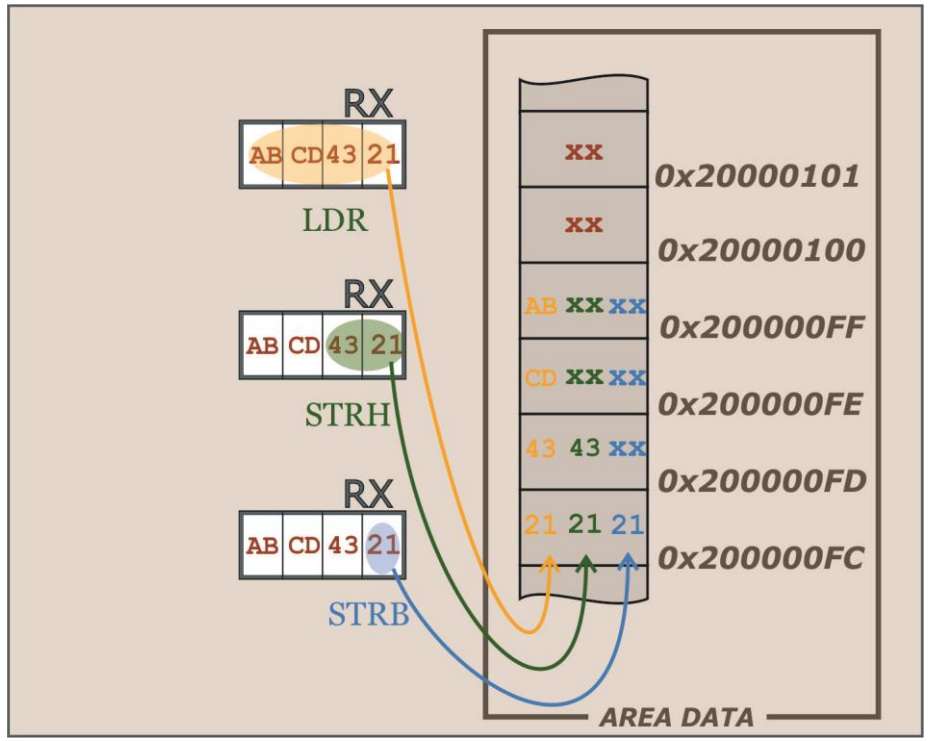
L'opérande *mémoire* : architecture LOAD/STORE



- 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

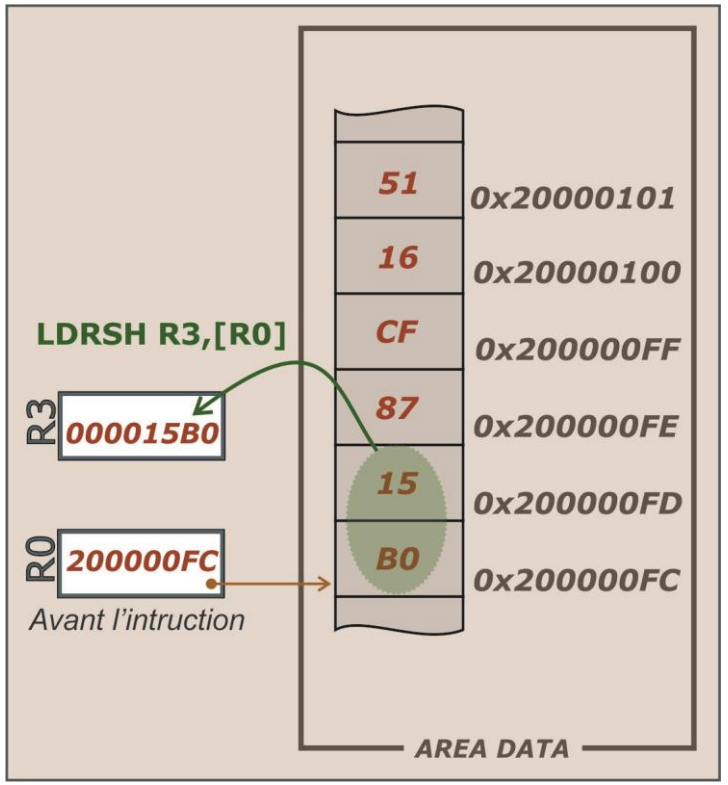
  = Mode d'adressage

- Adressage indirect simple

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

- Exemple précédent :

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



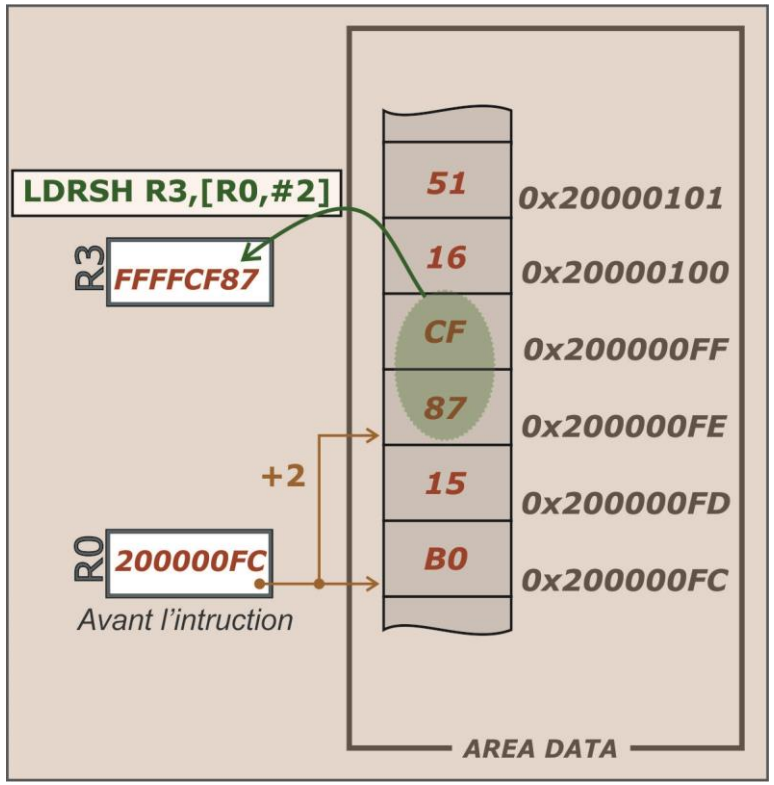
Adressage simple + déplacement

- 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]
```



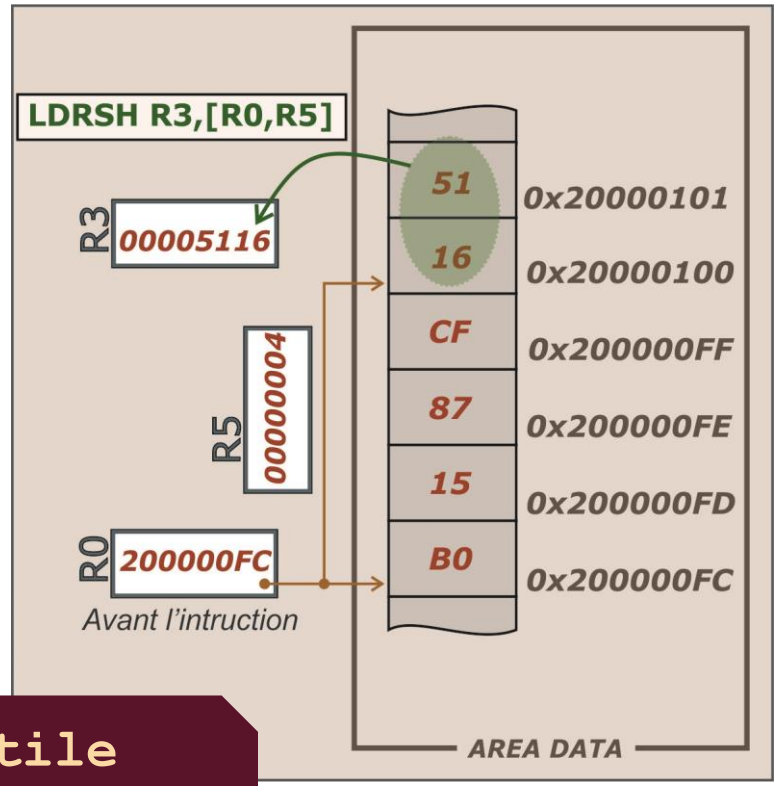
Adressage indexé

- 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]
```

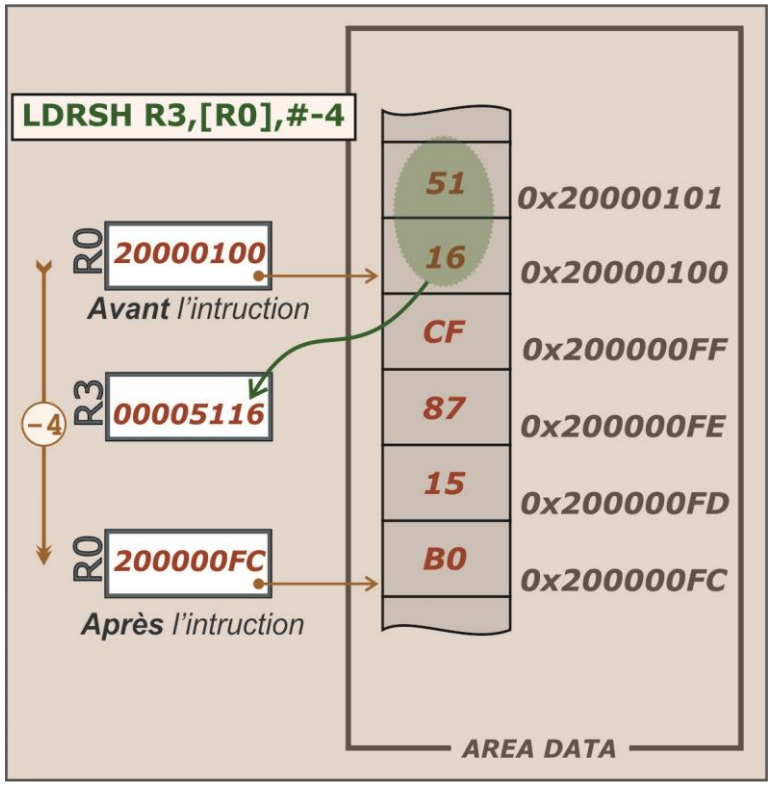
Adressage Post-déplacé

- 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
```



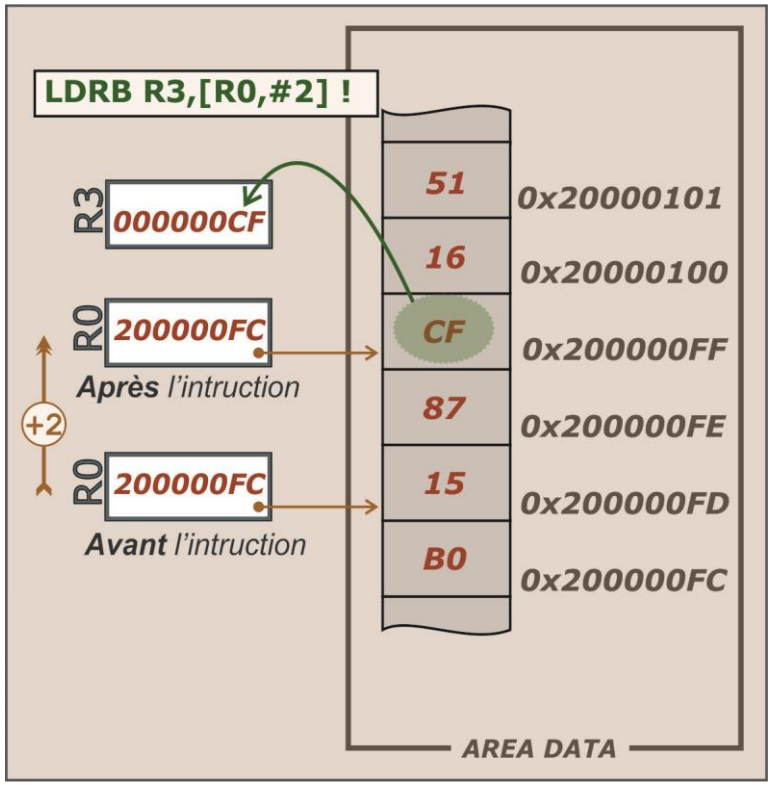
Adressage Pré-déplacé

- 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,=0x20000OFD
LDRB R3,[R0,#2] !
```



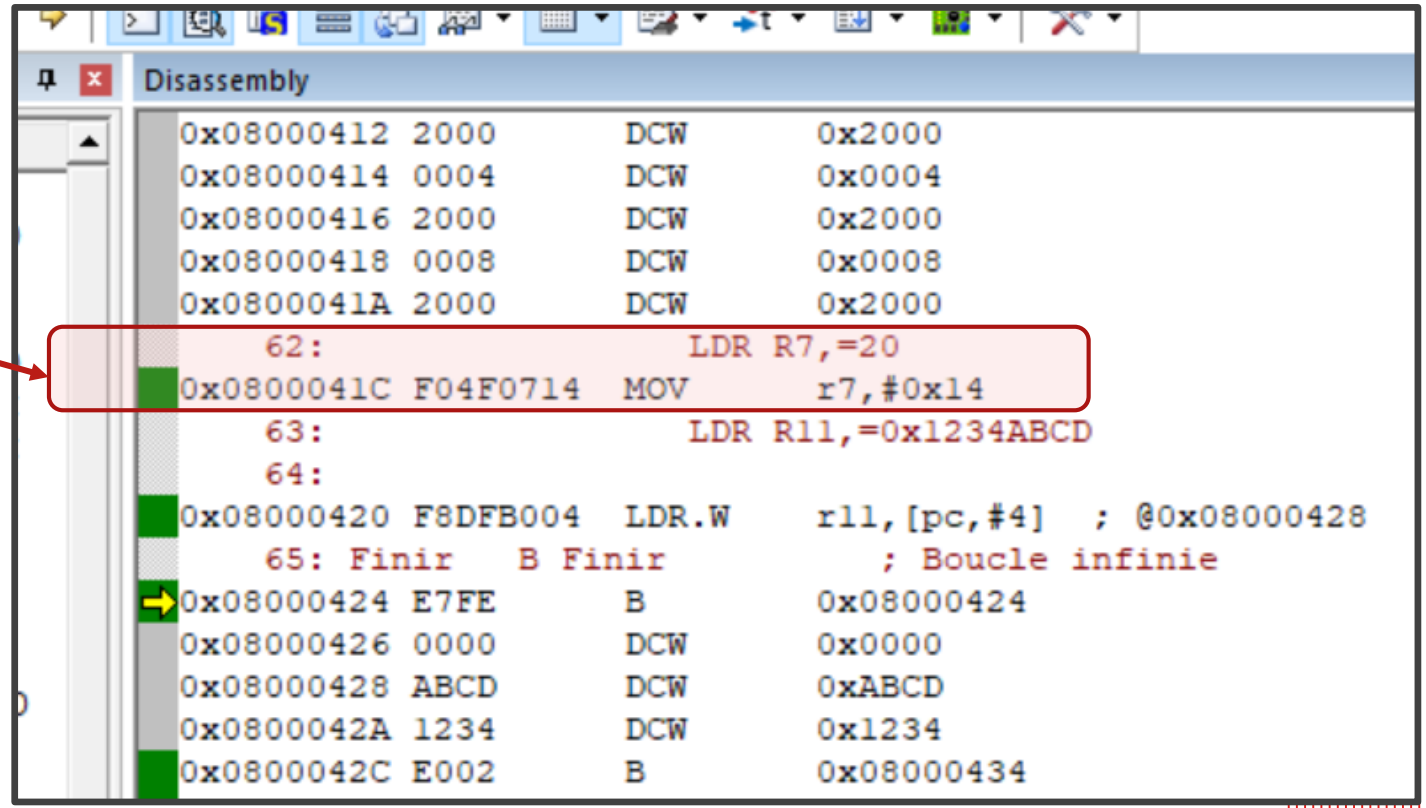
Retour sur la pseudo syntaxe LDR Rx,=….

- Désassemblage de :

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

- 1er 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** !!!

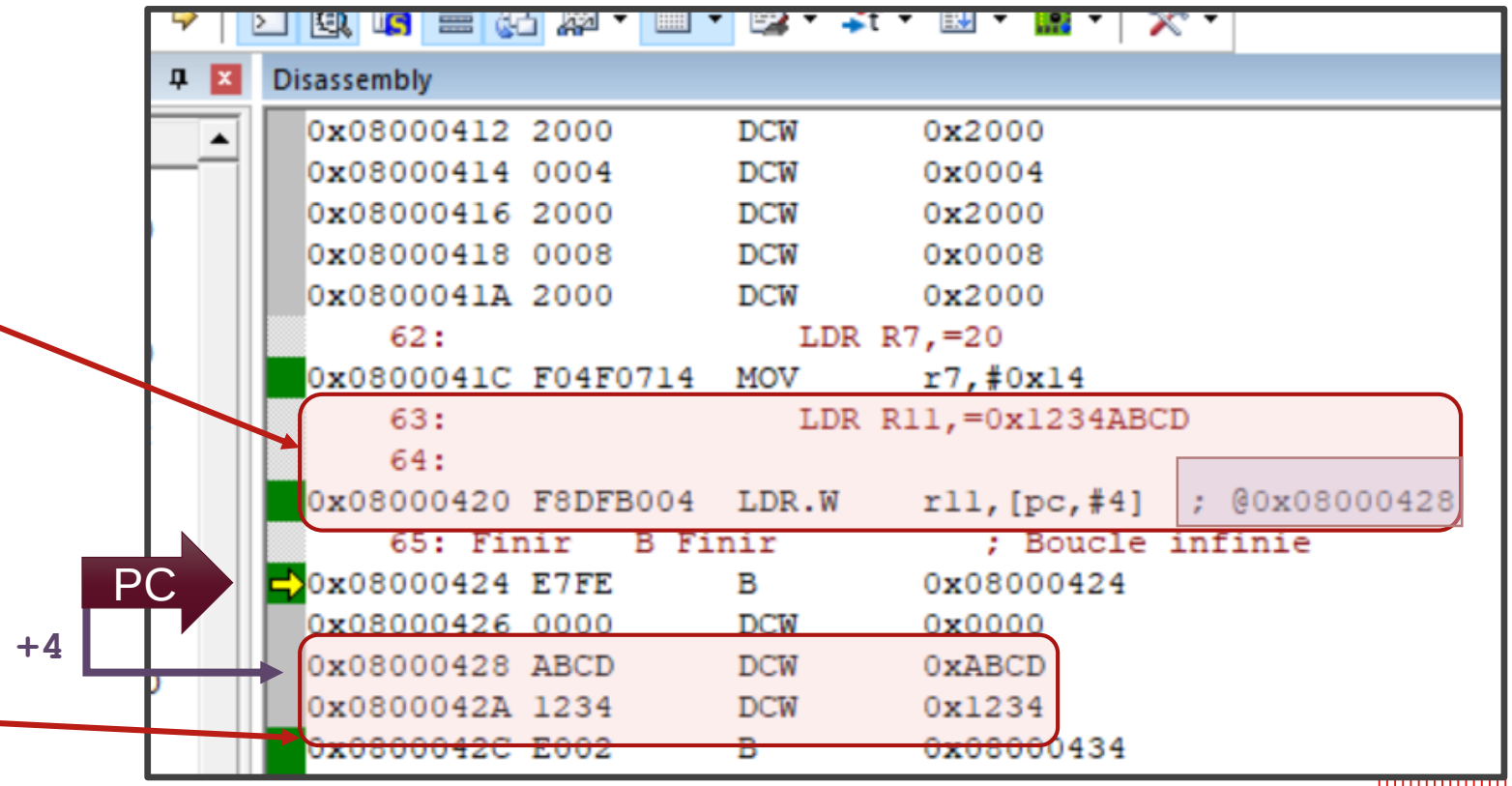


Retour sur la pseudo syntaxe LDR Rx,=….

- Désassemblage de :

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

- 2ième 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*)



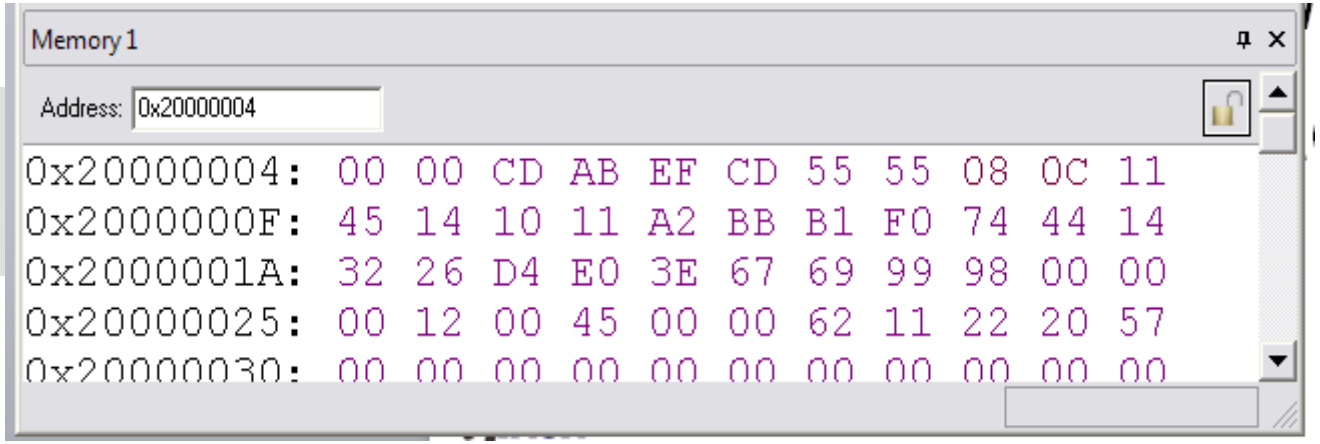
La syntaxe LDR,= n'existe pas pour le processeur !

Résumé des opérandes

- Opérandes registres
  - `R0` à `R13`
  - autres registres (`PC`, `SP`, `LR`) accessibles mais usage réservé (⇒ danger!)
- Opérandes immédiats
  - débutent avec un `#`
  - codés « dans » le code de l'instruction
  - pb si valeur > `255` ⇒ 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,#±imm8]
instr Rt,[Rn,Rm]
instr Rt,[Rn,Rm,LSL #<imm2>]
instr Rt,[Rn],#±imm8
instr Rt,[Rn,#±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]		