Syntaxe de l'Assembleur A32

Le langage machine lisible par l'humain

> "L'assembleur est la vérité du processeur."

Qu'est-ce que l'Assembleur ?

Assembleur = langage de programmation bas niveau

- Correspondance 1:1 avec le code machine
- Chaque instruction = 32 bits en mémoire
- Contrôle total sur le processeur

> **A32**
>
> Assembleur inspiré d'ARM, adapté pour l'architecture nand2c.

Structure d'un Programme A32

```
.text                   ; Section code
.global _start          ; Point d'entrée exporté

_start:                 ; Label du début
    MOV R0, #0          ; Instruction
    MOV R1, #1          ; Instruction
loop:                   ; Label de boucle
    ADD R0, R0, R1      ; Instruction
    B loop              ; Branchement

.data                   ; Section données
valeur:
    .word 42            ; Donnée 32 bits
```

Anatomie d'une Ligne

```
[label:]  MNEMONIC[.cond][.S]  Op1, Op2, Op3  ; commentaire
```

Élément	Obligatoire	Description
`label:`	Non	Nom symbolique pour l'adresse
`MNEMONIC`	Oui	Nom de l'instruction
`.cond`	Non	Condition d'exécution
`.S`	Non	Mettre à jour les flags
`Op1, Op2...`	Selon instr.	Opérandes
`; ...`	Non	Commentaire

Exemple Annoté Ligne par Ligne

```
.text                   ; 1
.global _start          ; 2

_start:                 ; 3
    MOV R0, #5          ; 4
    MOV R1, #3          ; 5
    ADD.S R2, R0, R1    ; 6
    B.EQ fin            ; 7
    SUB R2, R2, #1      ; 8
fin:                    ; 9
    HALT                ; 10
```

Ligne 1 : `.text`

```
.text
```

Signification :

- `.` indique une **directive** (pas une instruction)
- `text` = section de code exécutable

Pourquoi ?

- Sépare le code des données
- Le CPU exécute `.text`, pas `.data`
- Permet la protection mémoire

Section .text

Contient les instructions exécutables du programme.

Ligne 2 : `.global _start`

```
.global _start
```

Signification :

- `.global` exporte un symbole
- `_start` sera visible par le linker

Pourquoi ?

- Le système doit connaître le point d'entrée
- Sans `.global`, le symbole reste local
- Convention : `_start` ou `main`

Ligne 3 : `_start:`

```
_start:
```

Signification :

- `_start` = identifiant (nom du label)
- `:` termine la déclaration du label

Pourquoi ?

- Donne un nom à cette adresse mémoire
- Permet les branchements (`B _start`)
- Documentation du code

> **Label = Adresse**
>
> `_start` sera remplacé par son adresse (ex: 0x0000) lors de l'assemblage.

Ligne 4 : `MOV R0, #5`

```
    MOV R0, #5
```

Décomposition :

Élément	Signification
`MOV`	Move - copie une valeur
`R0`	Registre destination
`#5`	Valeur immédiate (constante)

Résultat : R0 ← 5

> **Le préfixe #**
>
> Indique une valeur immédiate (constante dans l'instruction), pas un registre.

Ligne 6 : `ADD.S R2, R0, R1`

```
    ADD.S R2, R0, R1
```

Décomposition :

Élément	Signification
`ADD`	Addition
`.S`	Suffixe : mettre à jour les flags NZCV
`R2`	Destination
`R0`	Premier opérande
`R1`	Second opérande

Résultat : R2 ← R0 + R1, flags mis à jour

Ligne 7 : `B.EQ fin`

```
    B.EQ fin
```

Décomposition :

Élément	Signification
`B`	Branch - saut inconditionnel
`.EQ`	Condition : si Z=1 (égalité)
`fin`	Label cible du saut

Comportement :

- Si le flag Z = 1 → saute à `fin`
- Sinon → continue à l'instruction suivante

Les Registres

Registre	Alias	Rôle
R0-R3	—	Arguments/retour
R4-R11	—	Variables locales
R12	IP	Temporaire
R13	SP	Stack Pointer
R14	LR	Link Register
R15	PC	Program Counter

Conventions

R4-R11 doivent être sauvegardés par l'appelé (callee-saved).

Pourquoi 16 Registres ?

Raison technique :

- 4 bits suffisent pour encoder 0-15
- Format d'instruction compact
- Compromis taille/flexibilité

Raison pratique :

- Assez pour les variables locales
- R13-R15 ont des rôles spéciaux
- Compatible avec la convention ARM

Types d'Opérandes

Type	Syntaxe	Exemple	Encodage
Registre	`Rn`	`R0`, `SP`	4 bits
Immédiat	`#val`	`#42`, `#0xFF`	12 bits
Literal	`=val`	`=0xDEADBEEF`	Pool
Mémoire	`[Rn]`	`[R0]`, `[SP, #4]`	Adresse

L'Immédiat : `#valeur`

```
MOV R0, #100        ; Décimal
MOV R1, #0xFF       ; Hexadécimal
MOV R2, #0b1010     ; Binaire
MOV R3, #-7         ; Négatif
```

Formats acceptés :

- Décimal : `42`
- Hexadécimal : `0xFF`
- Binaire : `0b1010`

Limite :

- 12 bits signés
- Range : -2048 à +2047

> **Attention**
>
> `MOV R0, #5000` → Erreur ! Utiliser `=5000` (literal).

Le Literal : `=valeur`

```
LDR R0, =0xDEADBEEF    ; Charge 32 bits
LDR R1, =ma_variable   ; Charge l'adresse
```

Pourquoi ?

- Les immédiats sont limités à 12 bits
- `=` permet des constantes 32 bits
- L'assembleur gère automatiquement

Comment ça marche :

```
LDR R0, =0xDEADBEEF  →  LDR R0, [PC, #offset]
                        ...
                        .word 0xDEADBEEF  ; literal pool
```

Adressage Mémoire

Mode	Syntaxe	Signification
Base	`[Rn]`	Mem[Rn]
Offset	`[Rn, #off]`	Mem[Rn + off]
Write-back	`[Rn]!`	Mem[Rn], puis Rn++
Pré-indexé	`[Rn, #off]!`	Mem[Rn + off], puis Rn += off

```
LDR R0, [R1]            ; R0 = Mem[R1]
LDR R0, [R1, #8]        ; R0 = Mem[R1 + 8]
STR R0, [R1, #4]!       ; Mem[R1+4] = R0, R1 += 4
```

Pourquoi Plusieurs Modes d'Adressage ?

Accès tableau :

```
; arr[i]
LDR R0, [R1, R2, LSL #2]
```

Accès structure :

```
; point.y (offset 4)
LDR R0, [R1, #4]
```

Parcours séquentiel :

```
; *ptr++
LDR R0, [R1]!
```

Pile :

```
; push/pop
STR R0, [SP, #-4]!
LDR R0, [SP], #4
```

Les Conditions (Suffixes)

Code	Signification	Flags testés
`.EQ`	Égal	Z=1
`.NE`	Non égal	Z=0
`.GT`	Supérieur (signé)	Z=0 et N=V
`.LT`	Inférieur (signé)	N≠V
`.GE`	Supérieur ou égal	N=V
`.LE`	Inférieur ou égal	Z=1 ou N≠V

Conditions Non-Signées

Code	Signification	Flags testés
`.HI`	Supérieur (non signé)	C=1 et Z=0
`.LO` / `.CC`	Inférieur (non signé)	C=0
`.HS` / `.CS`	Supérieur ou égal	C=1
`.LS`	Inférieur ou égal	C=0 ou Z=1

Signé vs Non-signé

Utilisez GT/LT/GE/LE pour les nombres signés, HI/LO/HS/LS pour les non-signés.

Pourquoi l'Exécution Conditionnelle ?

```
; Sans condition (classique)
    CMP R0, #0
    BEQ skip
    ADD R1, R1, #1
skip:

; Avec condition (plus efficace)
    CMP R0, #0
    ADD.NE R1, R1, #1   ; Exécuté seulement si R0 ≠ 0
```

Avantages :

- Moins de branchements = moins de stalls pipeline
- Code plus compact
- Meilleure prédiction

Les Flags NZCV

Flag	Nom	Mis à 1 quand...
N	Negative	Résultat négatif (bit 31 = 1)
Z	Zero	Résultat = 0
C	Carry	Dépassement non signé
V	oVerflow	Dépassement signé

```
ADDS R0, R1, R2     ; Met à jour NZCV
CMP R0, R1          ; Toujours met à jour (implicite)
ADD R0, R1, R2      ; Ne change PAS les flags
```

Le Suffixe `.S`

```
ADD R0, R1, R2      ; R0 = R1 + R2, flags inchangés
ADD.S R0, R1, R2    ; R0 = R1 + R2, flags mis à jour
```

Pourquoi deux versions ?

Sans `.S` :

- Préserve les flags
- Pour calculs intermédiaires
- Pas d'effet de bord

Avec `.S` :

- Pour tester le résultat
- Avant un branchement conditionnel
- Détection overflow

Instructions ALU

Instruction	Opération	Exemple
`ADD`	Rd = Rn + Op2	`ADD R0, R1, R2`
`SUB`	Rd = Rn - Op2	`SUB R0, R1, #5`
`RSB`	Rd = Op2 - Rn	`RSB R0, R1, #0`
`AND`	Rd = Rn & Op2	`AND R0, R1, R2`
`ORR`	Rd = Rn | Op2	`ORR R0, R1, #0xFF`
`EOR`	Rd = Rn ^ Op2	`EOR R0, R0, R0`
`BIC`	Rd = Rn & ~Op2	`BIC R0, R1, #0xF`

Instructions de Transfert

Instruction	Opération	Exemple
`MOV`	Rd = Op2	`MOV R0, #42`
`MVN`	Rd = ~Op2	`MVN R0, #0`
`LDR`	Rd = Mem[addr]	`LDR R0, [R1]`
`STR`	Mem[addr] = Rd	`STR R0, [R1]`
`LDRB`	Rd = Mem[addr] (byte)	`LDRB R0, [R1]`
`STRB`	Mem[addr] = Rd (byte)	`STRB R0, [R1]`

Instructions de Comparaison

Instruction	Opération	Met à jour
`CMP`	Rn - Op2	NZCV (toujours)
`CMN`	Rn + Op2	NZCV (toujours)
`TST`	Rn & Op2	NZ (toujours)
`TEQ`	Rn ^ Op2	NZ (toujours)

```
CMP R0, #10          ; Compare R0 avec 10
B.GT greater         ; Saute si R0 > 10
```

Pas de résultat

Ces instructions ne modifient pas les registres, seulement les flags.

Instructions de Branchement

Instruction	Effet	Usage
`B label`	PC = label	Saut simple
`B.cond label`	Si cond: PC = label	Saut conditionnel
`BL label`	LR = PC+4, PC = label	Appel fonction
`BX Rm`	PC = Rm	Retour fonction

```
    BL ma_fonction      ; Appel (sauve adresse retour dans LR)
    ...
ma_fonction:
    ...
    BX LR               ; Retour (PC = LR)
```

Instructions Système

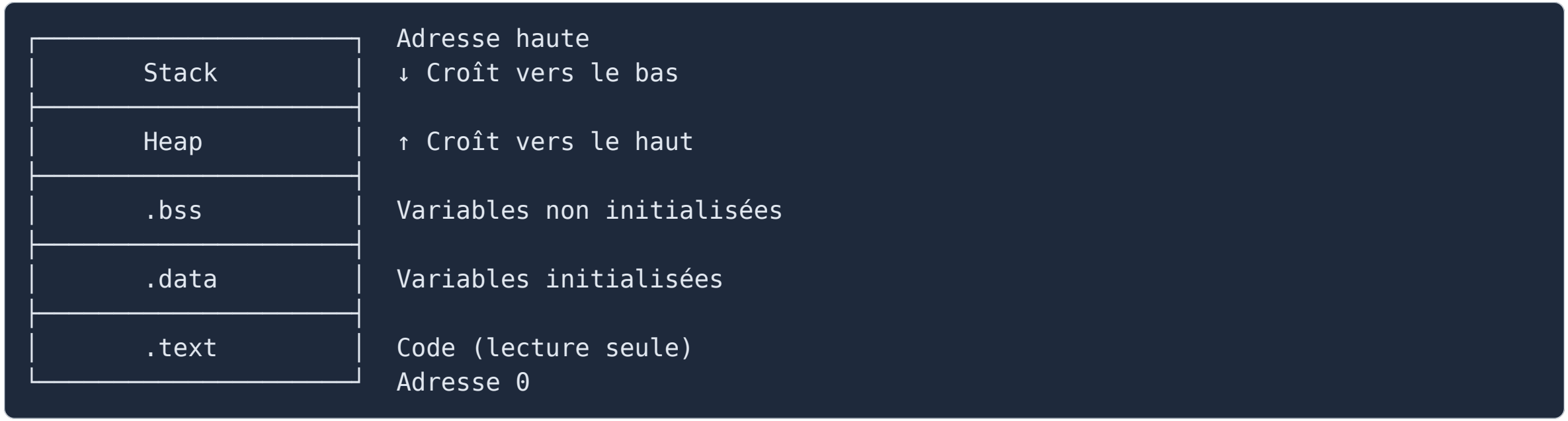
Instruction	Effet
`NOP`	Rien (1 cycle)
`HALT`	Arrête le processeur
`SVC #n`	Appel système (supervisor call)

```
    SVC #0              ; Appel système 0 (exit)
    HALT                ; Stop
```

Les Directives

Directive	Effet	Exemple
`.text`	Section code	`.text`
`.data`	Section données	`.data`
`.bss`	Section non initialisée	`.bss`
`.global`	Exporte un symbole	`.global main`
`.word`	32 bits	`.word 42`
`.byte`	8 bits	`.byte 0xFF`
`.asciz`	Chaîne + \0	`.asciz "Hello"`
`.align`	Alignement	`.align 2`
`.space`	Réserve N octets	`.space 100`

Pourquoi les Sections ?



Registres à Décalage

```
ADD R0, R1, R2, LSL #2    ; R0 = R1 + (R2 << 2)
```

Décalage	Signification	Utilité
`LSL #n`	Logical Shift Left	× 2^n
`LSR #n`	Logical Shift Right	÷ 2^n (non signé)
`ASR #n`	Arithmetic Shift Right	÷ 2^n (signé)
`ROR #n`	Rotate Right	Rotation bits

Exemple : `R2, LSL #2` = R2 × 4 (pour accès tableau d'int)

Pattern : Boucle For

```
; for (i = 0; i < 10; i++)
    MOV R0, #0          ; i = 0
loop:
    CMP R0, #10         ; i < 10 ?
    B.GE done           ; Non → sortie

    ; ... corps de boucle ...

    ADD R0, R0, #1      ; i++
    B loop              ; Recommencer
done:
```

Pattern : If / Else

```
; if (R0 == 0) { R1 = 1; } else { R1 = 2; }
    CMP R0, #0
    B.NE else
    MOV R1, #1          ; then
    B endif
else:
    MOV R1, #2          ; else
endif:
```

Version optimisée :

```
    CMP R0, #0
    MOV.EQ R1, #1       ; Si égal
    MOV.NE R1, #2       ; Si différent
```

Pattern : Appel de Fonction

```
; Appelant
    MOV R0, #5          ; Argument 1
    MOV R1, #3          ; Argument 2
    BL add_func         ; Appel
    ; R0 contient le résultat

; Fonction
add_func:
    PUSH {LR}           ; Sauver adresse retour
    ADD R0, R0, R1      ; R0 = arg1 + arg2
    POP {PC}            ; Retour
```

Pattern : Sauvegarde Registres

```
ma_fonction:
    PUSH {R4-R7, LR}     ; Sauver callee-saved + LR

    ; Utiliser R4-R7 librement
    MOV R4, R0
    MOV R5, R1
    ...

    POP {R4-R7, PC}      ; Restaurer et retourner
```

Convention

R4-R11 DOIVENT être préservés. L'appelé les sauvegarde si nécessaire.

Pattern : Accès Tableau

```
; int arr[10]; sum = 0;
; for (i = 0; i < 10; i++) sum += arr[i];

    LDR R0, =tableau     ; R0 = &arr[0]
    MOV R1, #0           ; sum = 0
    MOV R2, #10          ; count
loop:
    LDR R3, [R0], #4     ; R3 = *R0++
    ADD R1, R1, R3       ; sum += R3
    SUBS R2, R2, #1      ; count--
    B.NE loop
    ; R1 = somme
```

Commentaires

```
; Commentaire style ARM (point-virgule)
@ Commentaire style ARM alternatif
// Commentaire style C

MOV R0, #42    ; Commentaire en fin de ligne
```

Bonne pratique

Commentez le POURQUOI, pas le QUOI. Le code dit ce qu'il fait.

Erreurs Fréquentes

1. Oubli du `#`

```
MOV R0, 42    ; ERREUR !
MOV R0, #42   ; Correct
```

2. Immédiat trop grand

```
MOV R0, #5000  ; ERREUR !
LDR R0, =5000  ; Correct
```

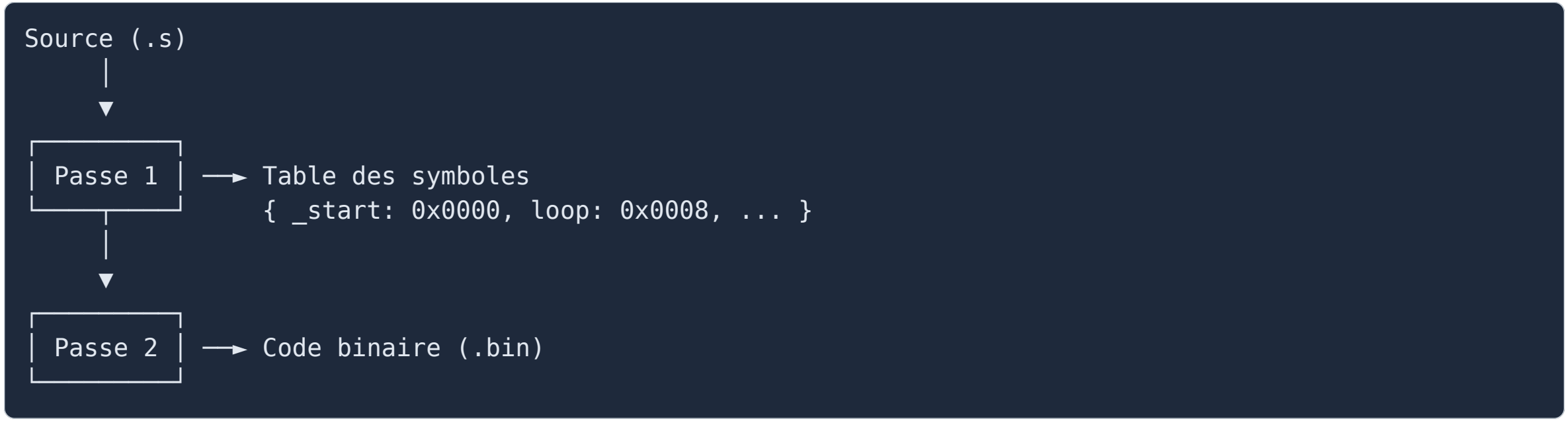
3. Registre invalide

```
MOV R16, #0    ; ERREUR !
```

4. Oubli de sauver LR

```
func:
    BL autre    ; LR écrasé !
    BX LR       ; Retour impossible
```

Le Processus d'Assemblage



Utilisation CLI

```
# Assembler
cargo run -p a32_cli -- assemble prog.s -o prog.bin

# Désassembler
cargo run -p a32_cli -- disasm prog.bin

# Simuler
cargo run -p a32_cli -- run prog.bin

# Debug pas à pas
cargo run -p a32_cli -- debug prog.bin
```

Questions de Réflexion

1. Pourquoi les immédiats sont-ils limités à 12 bits ?
2. Quelle est la différence entre `CMP` et `SUBS` ?
3. Pourquoi sauver LR au début d'une fonction ?
4. Comment accéder à `arr[i]` efficacement ?
5. Pourquoi `.text` est en lecture seule ?

Ce qu'il faut retenir

1. **Une instruction = 32 bits** exactement
2. **Registres R0-R15** avec rôles spéciaux (SP, LR, PC)
3. **Suffixes** : `.cond` pour conditionnel, `.S` pour flags
4. **Immédiats** `#` limités, **literals** `=` pour 32 bits
5. **Sections** : `.text` (code), `.data` (données)
6. **Conventions** : R0-R3 arguments, R4-R11 callee-saved

Questions ?

Référence : `/book/references/carte_isa_a32.md`

Exemples : `/web/demos/*.asm`

Prochain : Le langage C32