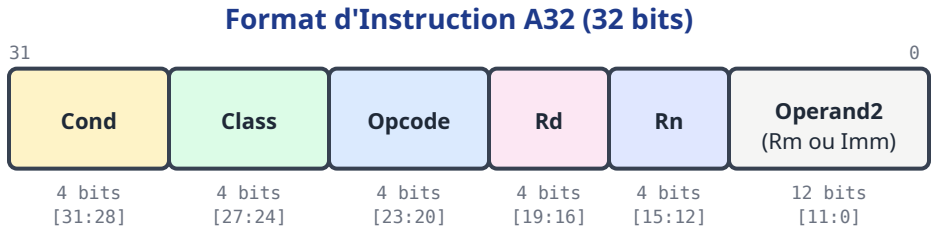
Chapitre 04 : Architecture Machine

> "Le langage est la limite de mon monde." — Wittgenstein

🎯 Où en sommes-nous ?



L'ISA — le contrat entre matériel et logiciel

L'**ISA** = le langage que parle le processeur !

Qu'est-ce qu'une Architecture ?

L'architecture définit :

1. **Les registres** : Combien ? Quelle taille ?
2. **Les instructions** : Quelles opérations possibles ?
3. **L'encodage** : Représentation binaire
4. **Le modèle mémoire** : Comment accéder aux données ?

ISA = Instruction Set Architecture

C'est un **contrat** entre matériel et logiciel.

nand2c A32 : Architecture RISC

Inspirée de ARM (smartphones, Raspberry Pi) :

- **RISC** : Reduced Instruction Set Computer
- **32 bits** : Registres et adresses
- **Load/Store** : Calcul uniquement entre registres

> 🔗 **ARM**
>
> Les mêmes concepts s'appliquent à ARM — syntaxe très proche.

CISC vs RISC

CISC (x86)	RISC (ARM, A32)
Instructions complexes	Instructions simples
`ADD [mem], reg` OK	Calcul entre registres seulement
Vitesse variable	~1 instruction/cycle
Plus facile à programmer	Plus facile à construire

Avantage RISC

Pipeline plus efficace, consommation réduite

La Règle Load/Store

En RISC, **jamais de calcul direct en mémoire** :

1. **LOAD**
   Mémoire → Registre
2. **COMPUTE**
   Calcul dans les registres
3. **STORE**
   Registre → Mémoire

Exemple Load/Store

Incrémenter une variable en mémoire :

```
LDR R0, [R1]      ; Charger depuis mémoire
ADD R0, R0, #1    ; Ajouter 1
STR R0, [R1]      ; Stocker en mémoire
```

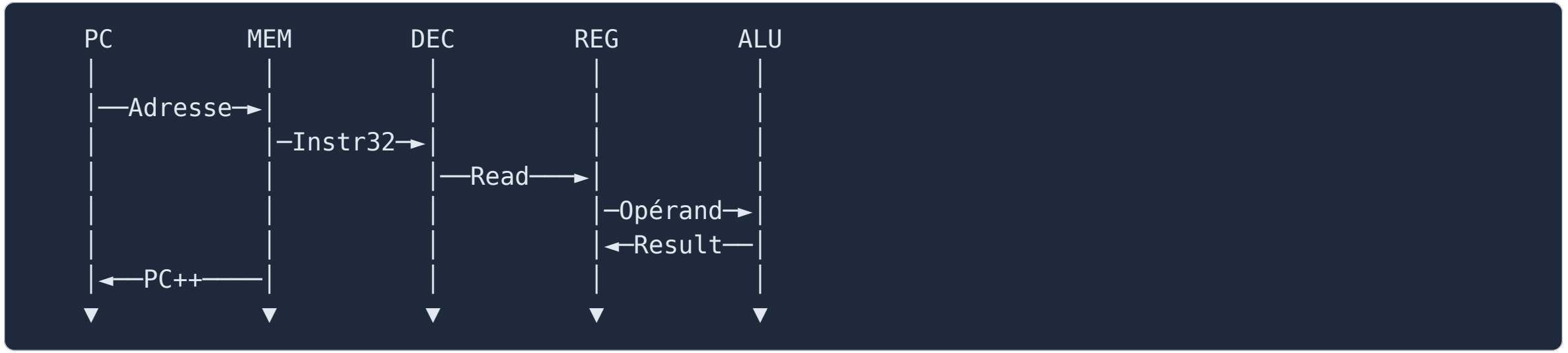
3 instructions pour x++

CISC le fait en 1 instruction, mais le matériel est plus complexe

Le Cycle Fetch-Decode-Execute

Le cœur du fonctionnement CPU

Détail du Cycle



Les 16 Registres

	Alias	Rôle
`R0-R3`	—	Arguments, retours
`R4-R11`	—	Variables locales
`R12`	IP	Temporaire
`R13`	**SP**	Stack Pointer
`R14`	**LR**	Link Register
`R15`	**PC**	Program Counter

Registres Spéciaux

R13 (SP) : Pointe vers le sommet de la pile

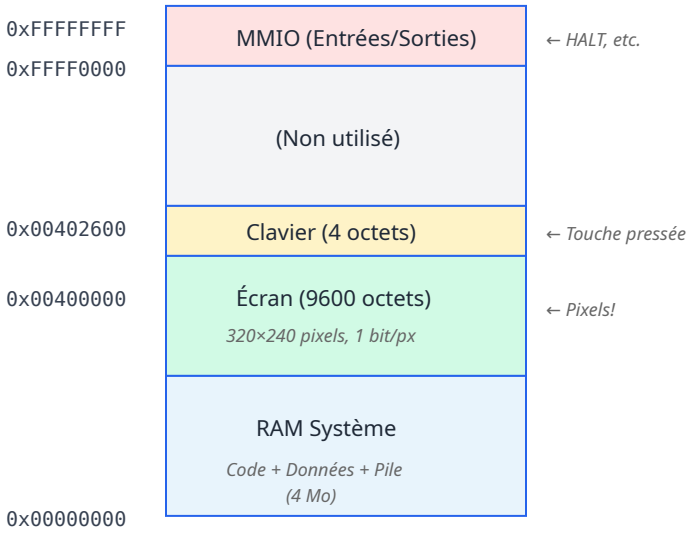
R14 (LR) : Adresse de retour après `BL`

R15 (PC) : Adresse de l'instruction courante

```
MOV PC, LR    ; Équivalent à "return"
```

> 🔗 **ARM**
>
> Organisation identique à ARM (ABI standard).

La Carte Mémoire



Organisation de l'espace d'adressage

Zone	Adresse	Usage
Code	0x00000000	Instructions
Data	0x00200000	Variables
Screen	0x00400000	MMIO
Keyboard	0x00402600	MMIO

Memory-Mapped I/O (MMIO)

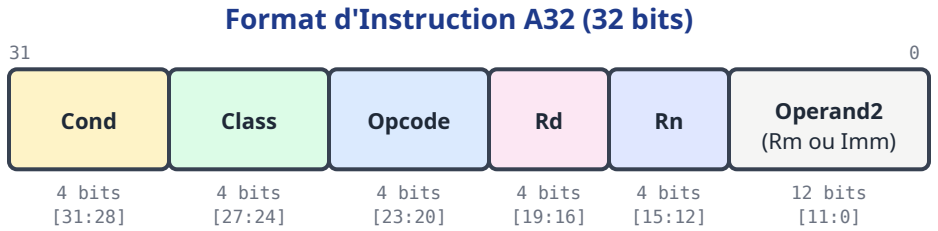
Les périphériques sont des **adresses mémoire** :

Écran : `0x00400000` - 1 bit par pixel

Clavier : `0x00402600` - Code ASCII

```
; Allumer premier pixel
LDR R0, =0x00400000
MOV R1, #0x80
STRB R1, [R0]
```

Format des Instructions (32 bits)



Structure commune à toutes les instructions

Détail du Format

Bits	Champ	Taille	Rôle
31-28	Cond	4 bits	Condition d'exécution
27-25	Class	3 bits	Type d'instruction
24-21	Opcode	4 bits	Opération spécifique
20	S	1 bit	Update flags
19-16	Rn	4 bits	Registre source 1
15-12	Rd	4 bits	Registre destination
11-0	Op2	12 bits	Opérande 2

Exemple d'Encodage : ADD R1, R2, R3

```
Cond  Class Opcode S Rn   Rd   Shift  Rm
1110  000   0100   0 0010 0001 0000   0011
|     |     |      | |    |    |      |
AL    Data  ADD    N R2   R1   0      R3
```

Encodage fixe 32 bits

Simplifie le décodage et le pipeline

Exécution Conditionnelle

Toute instruction peut être conditionnelle !

Avec branchement :

```
CMP R0, #0
B.NE skip
MOV R1, #1
skip:
```

Avec prédication :

```
CMP R0, #0
MOV.EQ R1, #1   ; Exécuté SI Z=1
```

Codes de Condition Complets

Code	Suffixe	Condition	Test
0000	EQ	Égal	Z = 1
0001	NE	Différent	Z = 0
0010	CS/HS	Carry Set / ≥ non-signé	C = 1
0011	CC/LO	Carry Clear / < non-signé	C = 0
1010	GE	≥ signé	N = V
1011	LT	< signé	N ≠ V
1100	GT	> signé	Z=0, N=V
1101	LE	≤ signé	Z=1 ou N≠V
1110	AL	Toujours	–

Classes d'Instructions

Bits	Classe	Description	Exemples
000	Data (reg)	Opérations registre-registre	ADD, SUB, AND
001	Data (imm)	Opérations avec immédiat	ADD R0, R1, #42
010	Load/Store	Accès mémoire	LDR, STR
011	Branch	Branchements	B, BL
111	System	Instructions système	HALT

Instructions Arithmétiques

```
ADD Rd, Rn, Rm      ; Rd = Rn + Rm
ADD Rd, Rn, #imm    ; Rd = Rn + imm
SUB Rd, Rn, Rm      ; Rd = Rn - Rm
MUL Rd, Rn, Rm      ; Rd = Rn * Rm
```

Suffixe S

ADDS met à jour les flags (N, Z, C, V), ADD ne les modifie pas.

Instructions Logiques

```
AND Rd, Rn, Rm      ; Rd = Rn & Rm
ORR Rd, Rn, Rm      ; Rd = Rn | Rm
EOR Rd, Rn, Rm      ; Rd = Rn ^ Rm
MVN Rd, Rm          ; Rd = ~Rm
MOV Rd, Rm          ; Rd = Rm
```

> 💾 **VHDL**
>
> Ce sont les mêmes opérations que l'ALU que vous avez construite !

Instructions de Comparaison

```
CMP Rn, Rm          ; Calcule Rn - Rm, modifie flags
CMP Rn, #imm        ; Compare avec immédiat
TST Rn, Rm          ; Calcule Rn & Rm, modifie flags
```

CMP = SUB sans destination

Le résultat est jeté, seuls les flags comptent

Accès Mémoire

```
LDR Rd, [Rn]        ; Rd = MEM[Rn]
LDR Rd, [Rn, #off] ; Rd = MEM[Rn + off]
STR Rd, [Rn]        ; MEM[Rn] = Rd
LDRB Rd, [Rn]       ; Charger 1 octet
STRB Rd, [Rn]       ; Stocker 1 octet
```

> 🔗 **ARM**
>
> Syntaxe identique à ARM — les modes d'adressage sont compatibles.

Modes d'Adressage

Mode	Syntaxe	Calcul adresse
Direct	`[Rn]`	Rn
Offset immédiat	`[Rn, #off]`	Rn + off
Offset registre	`[Rn, Rm]`	Rn + Rm
Pre-indexé	`[Rn, #off]!`	Rn = Rn + off, puis accès
Post-indexé	`[Rn], #off`	Accès, puis Rn = Rn + off

Branchements

```
B label             ; Saut inconditionnel
BL label            ; Branch with Link (appel)
B.EQ label          ; Saut si égal
B.NE label          ; Saut si différent
B.GT label          ; Saut si > (signé)
B.LT label          ; Saut si < (signé)
```

Appel de Fonction (BL)

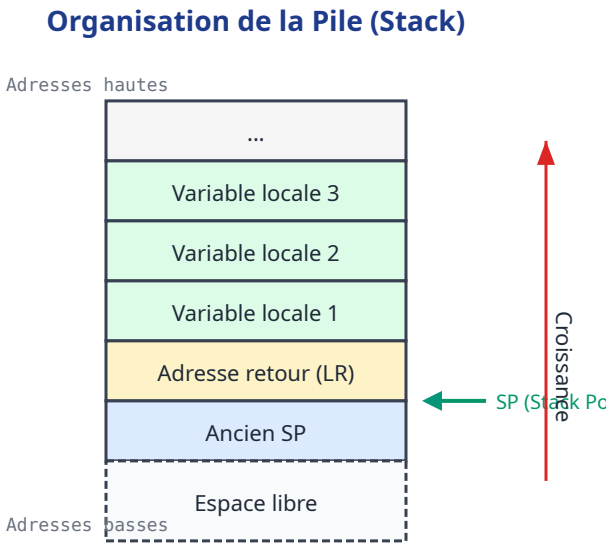
```
main:
    BL ma_fonction  ; LR = PC+4, puis saute
    ; ... on revient ici

ma_fonction:
    ; ... code
    MOV PC, LR      ; Retour (saute à LR)
```

BL = Branch and Link

Sauvegarde l'adresse de retour dans LR avant de sauter

La Pile (Stack)



La pile grandit vers le bas

- SP pointe vers le sommet
- PUSH = décrémenter SP, puis écrire
- POP = lire, puis incrémenter SP

Push et Pop

PUSH R0 :

```
SUB SP, SP, #4    ; Réserver place
STR R0, [SP]      ; Stocker
```

POP R0 :

```
LDR R0, [SP]      ; Lire
ADD SP, SP, #4    ; Libérer place
```

> 🔗 **ARM**
>
> ARM a les instructions PUSH et POP natives.

Exemple : Somme de 1 à 10

```
    MOV R0, #0        ; sum = 0
    MOV R1, #1        ; i = 1

loop:
    CMP R1, #10
    B.GT done         ; si i > 10, sortir
    ADD R0, R0, R1    ; sum += i
    ADD R1, R1, #1    ; i++
    B loop

done:
    HALT              ; R0 = 55
```

Tracé de l'Exemple

Cycle	R0	R1	Instruction
1	0	?	MOV R0, #0
2	0	1	MOV R1, #1
3	0	1	CMP R1, #10
4	0	1	B.GT done (non pris)
5	1	1	ADD R0, R0, R1
6	1	2	ADD R1, R1, #1
...	...	...	...
fin	55	11	HALT

Exemple : Max sans branchement

```
; R2 = max(R0, R1)
CMP R0, R1
MOV.GE R2, R0     ; Si R0 >= R1
MOV.LT R2, R1     ; Si R0 < R1
```

Avantage de la prédication

Évite les branchements coûteux (pipeline flush)

Questions de Réflexion

1. Pourquoi RISC est-il plus adapté aux smartphones qu'aux PC ?
2. Combien de bits faut-il pour encoder un numéro de registre parmi 16 ?
3. Pourquoi le PC est-il un registre visible (R15) ?
4. Comment fonctionne une boucle `while` en assembleur ?
5. Que se passe-t-il si on oublie le `B loop` ?

Ce qu'il faut retenir

1. **ISA = contrat** matériel/logiciel
2. **RISC** : Load, Compute, Store
3. **16 registres** : R13=SP, R14=LR, R15=PC
4. **Tout est conditionnel** : ADD.EQ, MOV.GT
5. **MMIO** : Périphériques = adresses
6. **Fetch-Decode-Execute** : Le cycle CPU

Questions ?

📚 **Référence :** Livre Seed, Chapitre 04 - Architecture

👉 **Exercices :** TD et TP sur le simulateur

Prochain chapitre : CPU (implémentation de l'ISA)