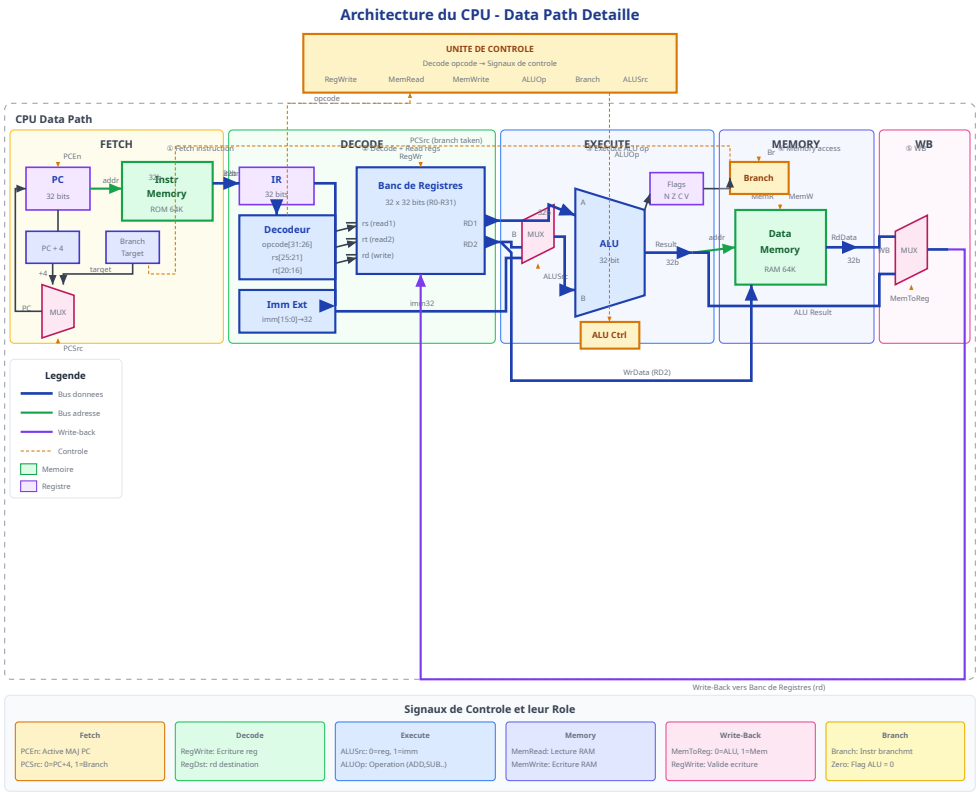


# Chapitre 05 : Le Processeur (CPU)

"Si vous ne pouvez pas le construire, vous ne le comprenez pas." — Feynman

# Où en sommes-nous ?



*Le CPU — point culminant du matériel*

Nous assemblons toutes les pièces !

# Qu'est-ce qu'un CPU ?

---

Le CPU (Central Processing Unit) :

- 1 Lit**  
les instructions depuis la mémoire
- 2 Décode**  
pour comprendre quoi faire
- 3 Exécute**  
les opérations
- 4 Répète**  
à l'infini (jusqu'à HALT)

## Ce qu'on a construit

---

Chapitre	Composant	Rôle
1	Portes	Briques de base
2	ALU	Calculs
3	Registres	R0-R15
3	PC	Adresse courante
3	RAM	Programme + données
4	ISA	Instructions

## Ce qu'il reste à construire

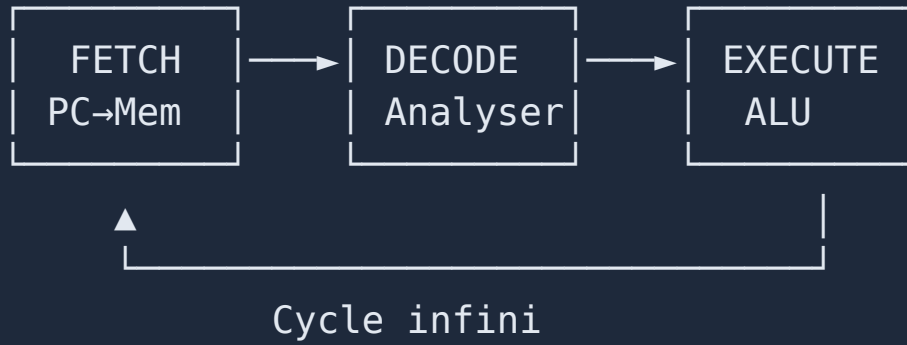
---

- **Décodeur** : Analyse les bits
- **Unité de contrôle** : Décide quoi activer
- **Multiplexeurs** : Routent les données
- **Le CPU** : L'assemblage final !

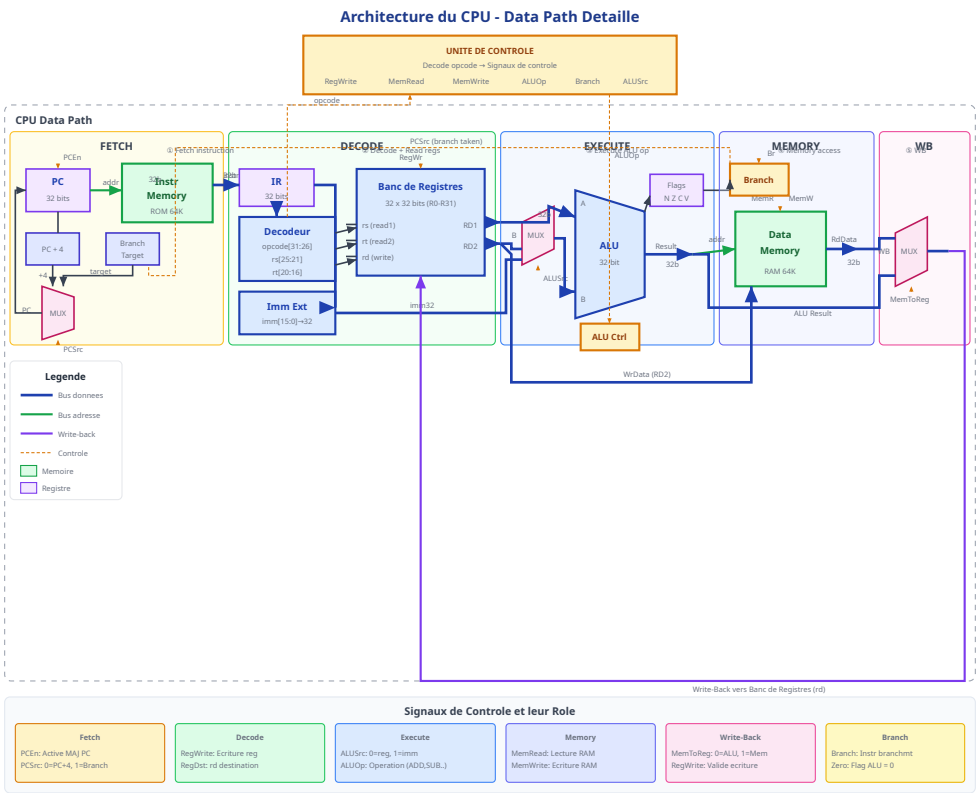
### Assemblage

Connecter les composants existants avec la logique de contrôle

## Le Cycle Fetch-Decode-Execute

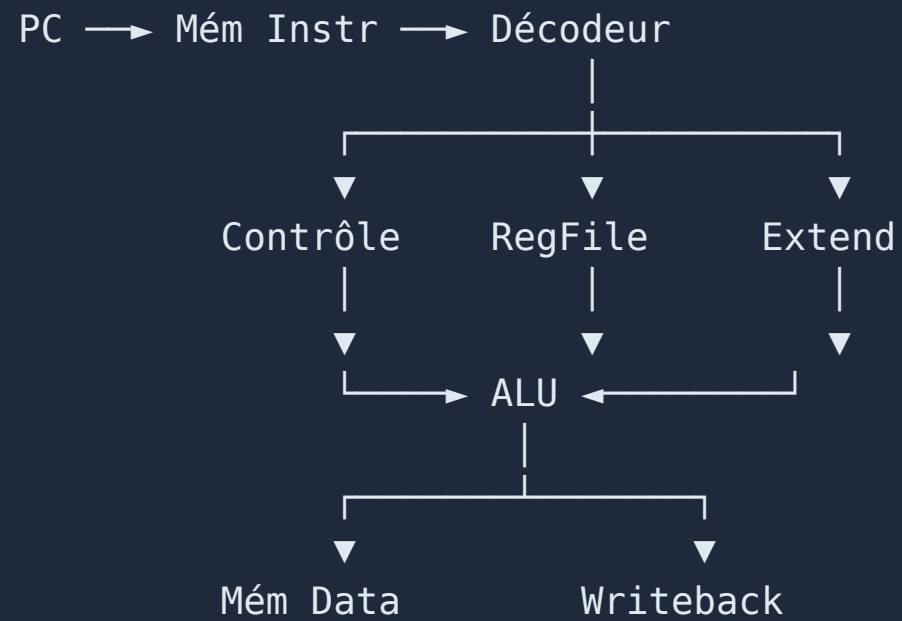


# Architecture du CPU (Datapath)



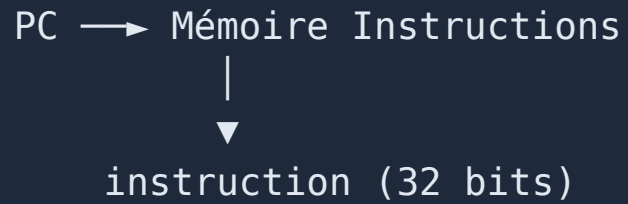
Le datapath — chemin des données à travers le CPU

## Vue Schématique du Datapath





## Étage 1 : Fetch (IF)



```
graph TD; PC --> MI[Mémoire Instructions]; MI --> I[instruction 32 bits];
```

PC → Mémoire Instructions  
↓  
instruction (32 bits)

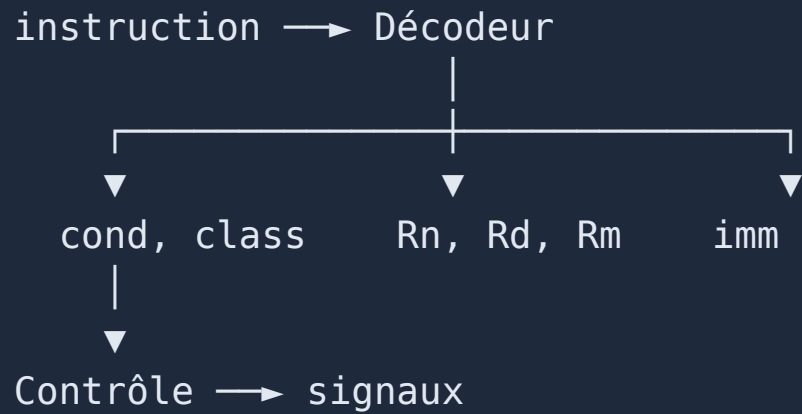
### Instruction Fetch

Lecture de 32 bits à l'adresse pointée par PC

### Actions :

- PC envoie l'adresse
- Mémoire renvoie l'instruction
- PC préparé pour PC+4

## Étage 2 : Decode (ID)



Découpe les champs :

- Condition (4 bits)
- Classe d'instruction
- Registres sources/dest
- Immédiat étendu

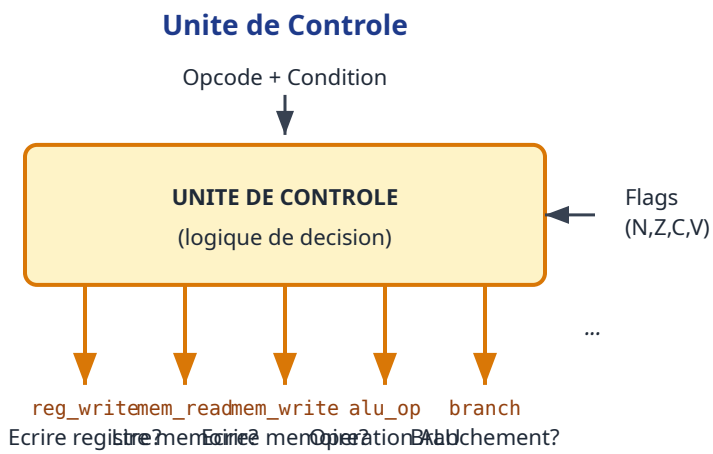
## Le Décodeur — Détail des Champs

---

Signal	Bits	Description
cond	31-28	Condition (EQ, NE...)
class	27-25	Type (ALU, MEM, BRANCH)
op	24-21	Opération ALU
S	20	Mettre à jour flags ?
Rn	19-16	Source 1
Rd	15-12	Destination
Rm/Imm	11-0	Source 2 / Immédiat

# L'Unité de Contrôle

Génère les **signaux de contrôle** basés sur l'opcode.



*L'unité de contrôle génère les signaux*

## Signaux de Contrôle

---

Instruction	reg_write	mem_read	mem_write	alu_src
ADD	1	0	0	reg
ADD #imm	1	0	0	imm
LDR	1	1	0	imm
STR	0	0	1	imm
B	0	0	0	—
CMP	0	0	0	reg

## Étage 3 : Register Read (ID suite)

---

$R_n, R_m \rightarrow \text{Banc de Registres} \rightarrow \text{Data\_A}, \text{Data\_B}$

### Lecture simultanée

Le banc de registres a 2 ports de lecture, on lit  $R_n$  et  $R_m$  en parallèle.

## Étage 4 : Execute (EX)

---



### Actions :

- L'ALU effectue l'opération (ADD, SUB, AND...)
- Les flags (N, Z, C, V) sont calculés
- Les flags sont mis à jour si S=1

## Étage 5 : Memory (MEM)

---

Pour LDR :

MEM[adresse] → valeur

Pour STR :

valeur → MEM[adresse]

Sinon : (rien)

### Accès mémoire

Uniquement pour les instructions Load/Store



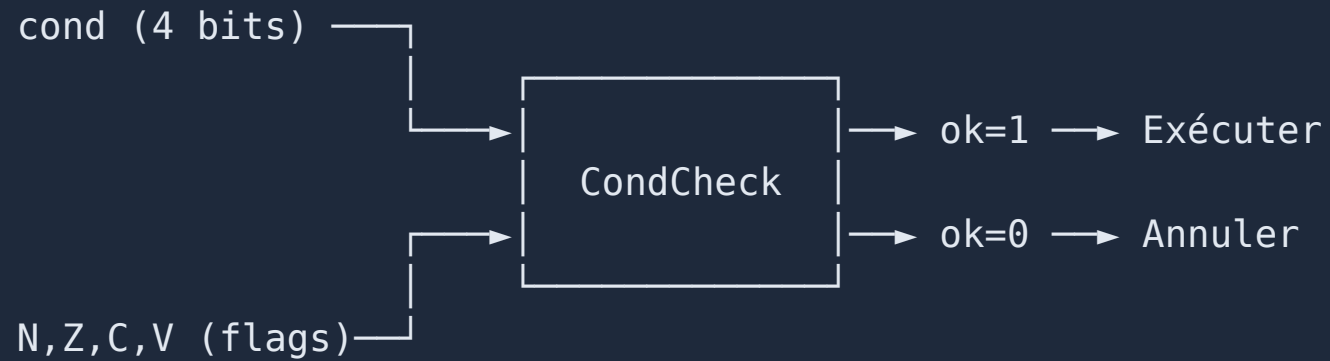
## Étage 6 : Writeback (WB)

---

```
Résultat → MUX → Banc de Registres → Rd
           |
           | ALU_out ou MEM_out ?
```

Si `reg_write = 1` ET `cond_ok = 1`, on écrit dans Rd.

## Le CondCheck — Exécution Conditionnelle



Si la condition n'est pas satisfaite, l'instruction est **annulée**.

## Logique CondCheck

---

```
case cond is
  when "0000" => ok := Z;           -- EQ
  when "0001" => ok := not Z;       -- NE
  when "1010" => ok := (N = V);     -- GE
  when "1011" => ok := (N /= V);    -- LT
  when "1100" => ok := (Z='0') and (N=V); -- GT
  when "1110" => ok := '1';         -- AL
  when others => ok := '0';
end case;
```

## Les Multiplexeurs du CPU

---

Mux	Choix 0	Choix 1	Contrôle
ALU_src	Rm	Imm	imm_src
Writeback	ALU_out	MEM_out	mem_to_reg
PC_src	PC+4	Branch_target	branch_taken



## Parcours de ADD R1, R2, R3

---

- IF** Lire l'instruction à PC
- ID** Décoder : class=ALU, reg\_write=1, lire R2 et R3
- EX** ALU calcule R2 + R3
- MEM** (rien)
- WB** Écrire résultat dans R1

## Exemple : LDR R0, [R1, #8]

---

- ID** class=MEM, mem\_read=1
- EX** ALU calcule  $R1 + 8$
- MEM** Lire MEM[R1+8]
- WB** Écrire dans R0

## Exemple : B.EQ label

---

- ID** class=BRANCH, calcul adresse cible
- EX** CondCheck vérifie  $Z = 1$  ?
- PC** Si ok :  $PC \leftarrow \text{cible}$ , sinon  $PC \leftarrow PC+4$



## CPU Mono-cycle vs Pipeline

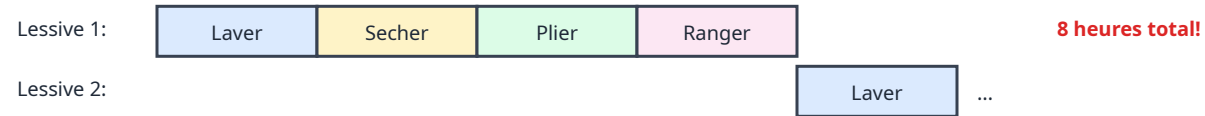
---

Mono-cycle	Pipeline
1 instruction à la fois	5 en parallèle
Cycle long (toutes les phases)	Cycles courts (1 phase)
Simple à concevoir	Plus complexe
Notre implémentation	Processeurs réels

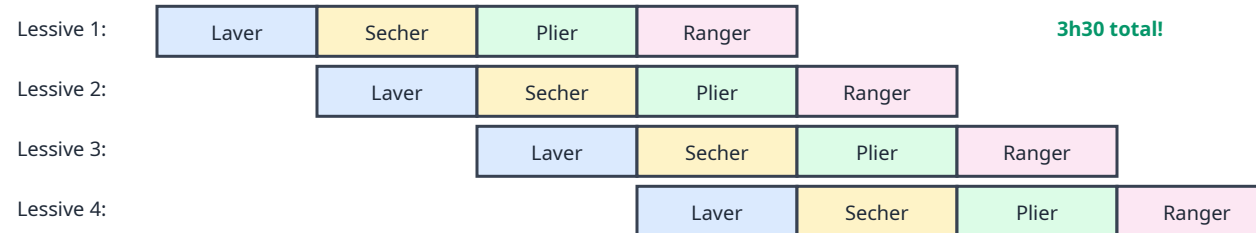
# Analogie du Pipeline : La Laverie

## Analogie : La Laverie

### Approche "Single-Cycle" : une lessive à la fois



### Approche "Pipeline" : parallelisation

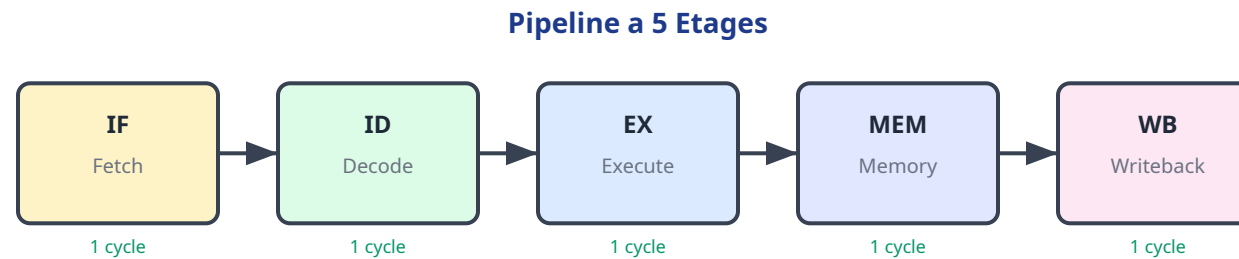


Le pipeline traite plus de lessives par heure (debit), meme si chaque lessive prend toujours 2h

*Pipeline = plusieurs charges en parallèle*

## Pipeline 5 Étages

---



5 instructions peuvent etre en cours d'execution simultanement

*IF → ID → EX → MEM → WB*

## Vue Temporelle du Pipeline

Cycle:	1	2	3	4	5	6	7
Instr 1	IF	ID	EX	MEM	WB		
Instr 2		IF	ID	EX	MEM	WB	
Instr 3			IF	ID	EX	MEM	WB

Débit = 1 instruction/cycle (après remplissage)

# Hazards (Problèmes Pipeline)

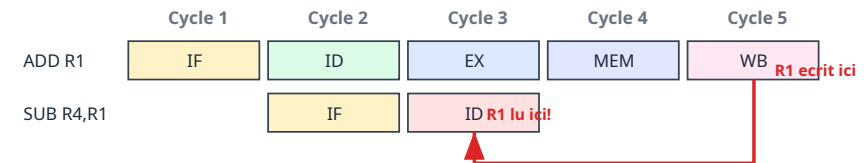
## Data Hazard :

```
ADD R1, R2, R3 ; Écrit R1  
SUB R4, R1, R5 ; Lit R1 → Problème !
```

R1 n'est pas encore écrit quand SUB le lit !

### Alea de Donnees (Data Hazard)

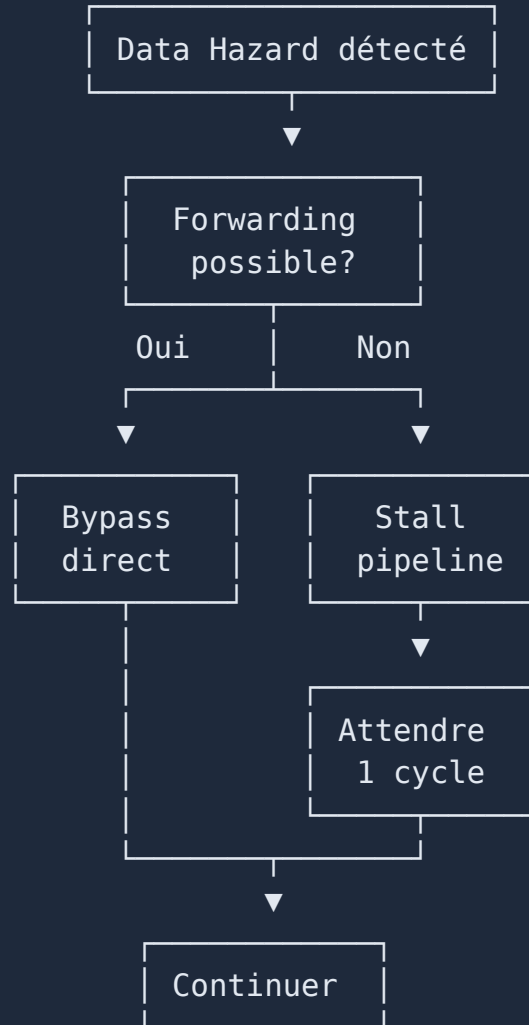
```
ADD R1, R2, R3 ; R1 = R2 + R3  
SUB R4, R1, R5 ; R4 = R1 - R5 (utilise R1!)
```



PROBLEME: SUB lit R1 au cycle 3, mais ADD n'ecrit R1 qu'au cycle 5!

*Dépendance de données*

# Solutions aux Hazards



## Forwarding (Bypass)

---

```
ADD R1, R2, R3  
SUB R4, R1, R5
```

Le résultat de ADD est disponible à la sortie de l'ALU **avant** d'être écrit dans R1.

**Forwarding** : Envoyer le résultat directement à l'entrée de l'ALU pour l'instruction suivante.

# CPU Visualizer

---

 [Ouvrir le CPU Visualizer](#)

## Fonctionnalités :

- Vue pipeline (5 étapes)
- Registres R0-R15
- Flags NZCV
- Code source avec surlignage
- Mode pas-à-pas
- 7 démos interactives



## Questions de Réflexion

---

1. Combien de MUX minimum faut-il dans un CPU simple ?
2. Pourquoi le PC est-il incrémenté de 4 et pas de 1 ?
3. Que se passe-t-il si on charge une instruction invalide ?
4. Pourquoi le forwarding ne résout-il pas tous les hazards ?
5. Comment le pipeline gère-t-il un branchement ?

## Ce qu'il faut retenir

---

1. **Fetch → Decode → Execute → Mem → WB**
2. **Décodeur** analyse les bits de l'instruction
3. **Contrôle** génère les signaux d'activation
4. **MUX** routent les données selon le contexte
5. **CondCheck** permet l'exécution conditionnelle
6. **Pipeline** = performances (5× potentiel)

# Questions ?



**Référence** : Livre Seed, Chapitre 05 - CPU



**Exercices** : TD et TP + CPU Visualizer

**Prochain chapitre** : Assembleur