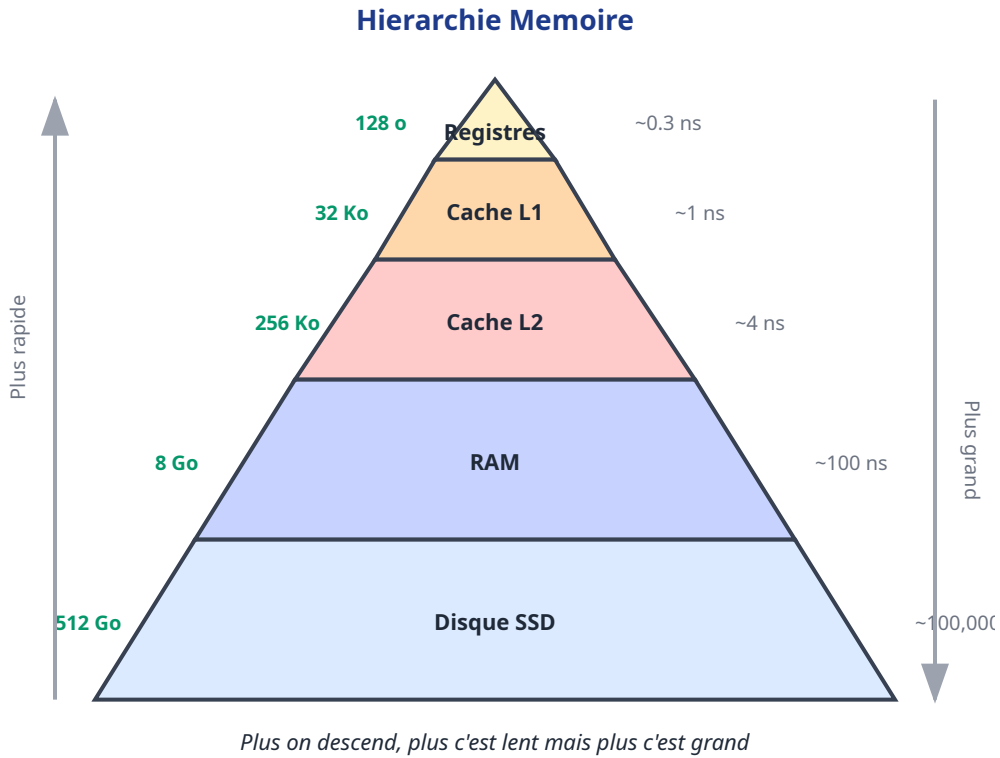
# Chapitre 03 : Logique Séquentielle et Mémoire

> "Le temps est ce qui empêche tout d'arriver en même temps." — John Wheeler

# 🎯 Où en sommes-nous ?




*La mémoire — niveau 3 de notre stack*

Nous apprenons à **mémoriser** !

# Le Problème de l'État

```
x = x + 1;
```

Pour exécuter cette instruction :

1. **Lire**
   la valeur actuelle de `x`
2. **Calculer**
   `x + 1` avec l'ALU
3. **Écrire**
   le résultat dans `x`

**Sans mémoire, pas de "valeur actuelle" !**

# Combinatoire vs Séquentiel

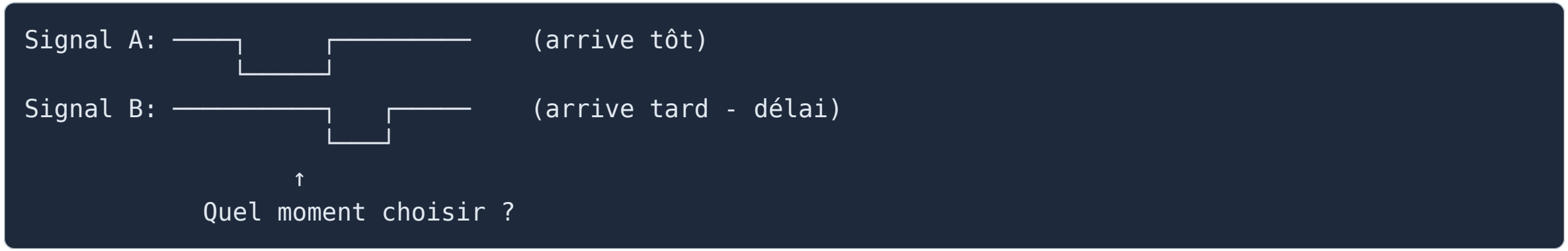
| Circuits Combinatoires | Circuits Séquentiels |
|---|---|
| Sortie = f(entrées) | Sortie = f(entrées, **état**) |
| Pas de mémoire | A de la mémoire |
| Pas d'horloge | Synchronisé par horloge |
| Ex: AND, OR, ALU | Ex: Registres, RAM, CPU |

**Différence fondamentale**

Les circuits séquentiels ont une **notion de temps**

# Le Problème : Quand Capturer ?

**Sans horloge, comment savoir QUAND lire les entrées ?**




**Le problème :**

- Les signaux ont des délais différents
- Certains bits sont "prêts" avant d'autres
- Capturer trop tôt = valeur incorrecte !

**Chaos garanti**

Sans synchronisation, le circuit capture des valeurs incohérentes

# La Solution : Un Chef d'Orchestre

**L'horloge = un signal qui dit "MAINTENANT !" à tout le circuit**

Comme un **chef d'orchestre** qui bat la mesure :

- Tous les musiciens jouent au même moment
- Pas de cacophonie

Comme un **feu de signalisation** :

- Tout le monde attend le feu vert
- Puis tout le monde avance ensemble

**Principe fondamental**

L'horloge donne un **rythme commun** à tous les composants du circuit

# L'Horloge (Clock)

Signal périodique qui oscille entre 0 et 1 :

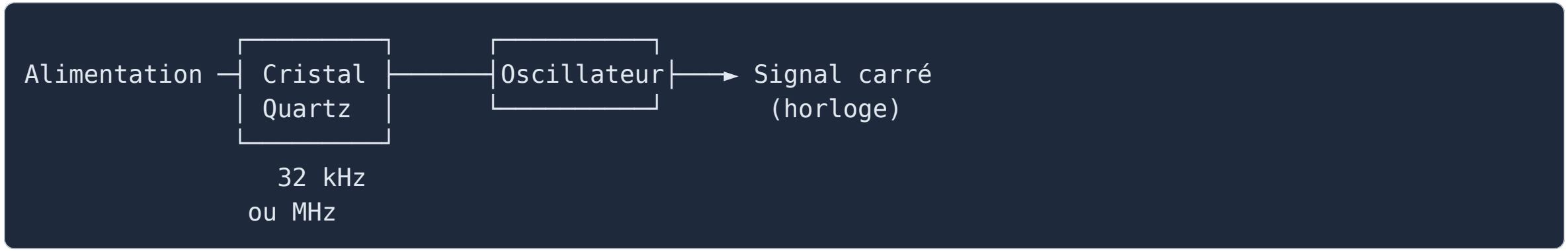

**Vocabulaire :**

- **Front montant** : passage 0→1 (moment de capture)
- **Période** : durée d'un cycle complet
- **Fréquence** : cycles par seconde (Hz)

> 🔗 **ARM**
>
> Un ARM Cortex-M4 à 168 MHz = 168 millions de "MAINTENANT !" par seconde

# D'où Vient l'Horloge ?

**Un cristal de quartz** vibre à fréquence fixe quand on lui applique une tension :




### Pourquoi le quartz ?

- Vibration TRÈS stable (~10 ppm)
- Peu coûteux et robuste
- Même principe que les montres !

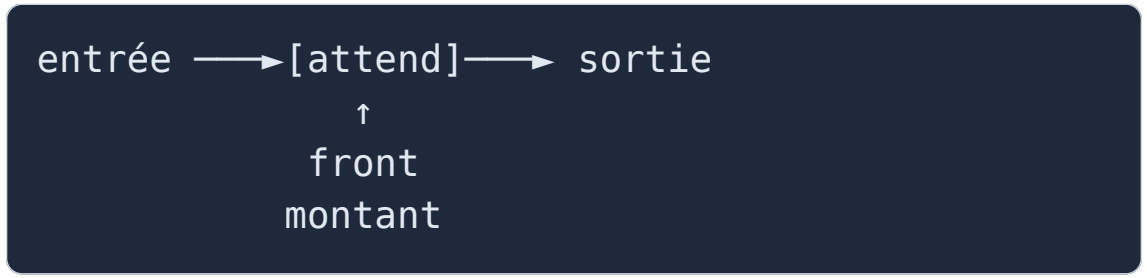
> **Le battement de cœur**
>
> Le cristal est le "cœur" de l'ordinateur — sans lui, rien ne fonctionne

# Horloge et Mémoire : Le Lien Fondamental

**SANS horloge :**


- Sortie change dès que l'entrée change
- Impossible de "figer" une valeur
- = Circuit **combinatoire**

**AVEC horloge :**




- Sortie change SEULEMENT au front montant
- Entre deux fronts = valeur STABLE
- = Circuit **séquentiel** (mémoire !)

**Révélation**

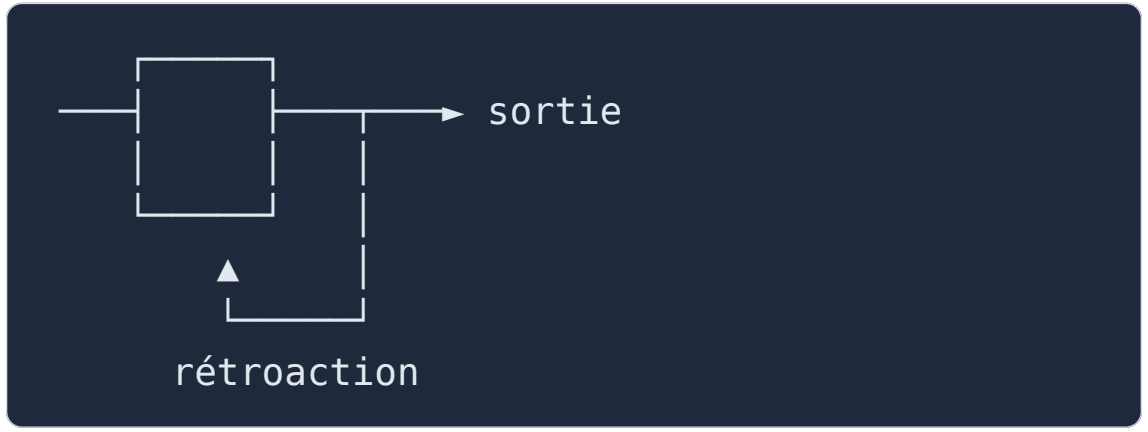
L'horloge transforme un simple fil en **mémoire**

# Comment Construire une Mémoire ?

**Problème :** Comment créer un circuit qui "retient" une valeur ?

**L'idée clé : la rétroaction**

Si la sortie d'une porte revient à son entrée, le circuit peut "se souvenir" de son état.




**Progression pédagogique**

Pour comprendre en détail, voir les animations :

1. **SR Latch** — verrou de base
2. **Gated D Latch** — avec Enable
3. **DFF** — déclenché par front

# La Bascule D (DFF)

**DFF** = Data Flip-Flop = notre **brique de base** pour la mémoire


*Symbole de la DFF*

**Règle fondamentale :**

```
q(t) = d(t-1)
```

La sortie = l'entrée **au front montant précédent**

> **Abstraction**
>
> On utilise le DFF comme "atome" sans détailler son intérieur (voir animations pour les curieux)

# Comportement de la DFF


**Décalage temporel**

La sortie est "en retard" d'un cycle — c'est la mémoire !

# Diagramme d'États de la DFF


La DFF a exactement 2 états : Q=0 ou Q=1

# Le Problème : Garder une Valeur

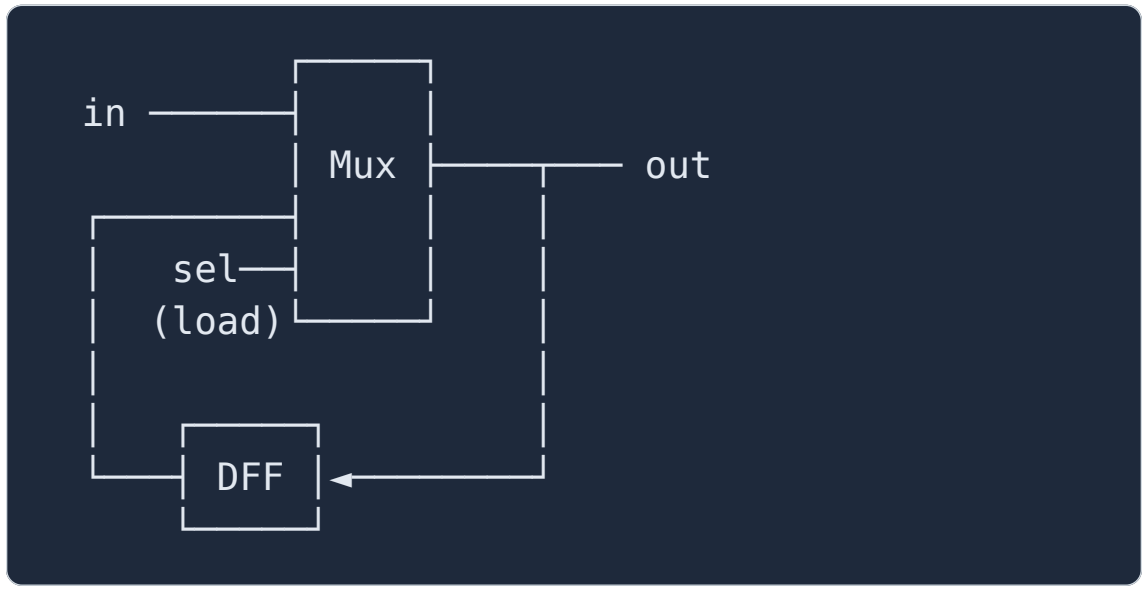
La DFF mémorise UN cycle, puis prend la nouvelle valeur.

**On veut :**

- Si `load = 1` : stocker la nouvelle valeur
- Si `load = 0` : **conserver** l'ancienne

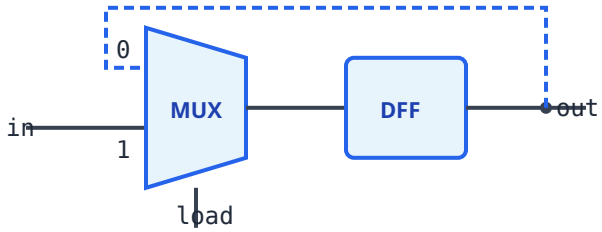
> **Besoin**
>
> Un signal de contrôle pour décider quand écrire

# La Solution : Rétroaction




- Si load=0 : Mux choisit sortie DFF (conservation)
- Si load=1 : Mux choisit `in` (nouvelle valeur)

# Registre 1-bit




*Structure du registre 1-bit*

```
entity BitReg is
  port(
    d    : in bit;
    load : in bit;
    q    : out bit
  );
end entity;
```

Cette boucle transforme un délai en **mémoire permanente** !

# VHDL : Registre avec Load

💾 **VHDL**

```vhdl
process(clk)
begin
  if rising_edge(clk) then
    if load = '1' then
      q <= d;
    -- sinon q garde sa valeur
    end if;
  end if;
end process;
```

# Registre 32-bits

**32 registres 1-bit en parallèle :**


Tous les bits sont capturés **simultanément** sur le front montant.

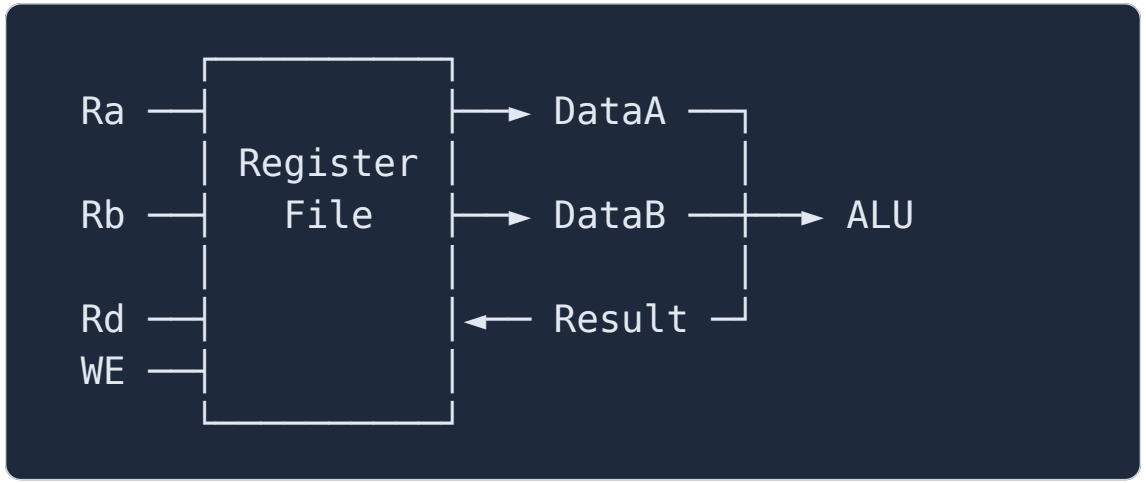
# Banc de Registres : Pourquoi 2 Ports Lecture ?

**Considérons une instruction ALU :**

```
ADD R2, R0, R1    ; R2 = R0 + R1
```

**Besoin en UN cycle :**

1. Lire R0 (premier opérande)
2. Lire R1 (deuxième opérande)
3. Calculer R0 + R1
4. Écrire le résultat dans R2




**2 lectures simultanées**

Pour faire A op B en un cycle, il faut lire A ET B en même temps !

# Registres du CPU nand2c

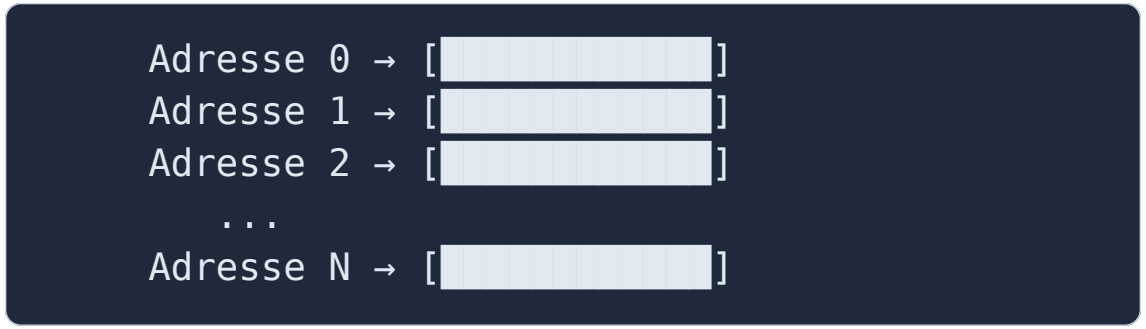
| | Alias | Rôle |
|---|---|---|
| **R0-R12** | - | Registres généraux |
| **R13** | SP | Stack Pointer |
| **R14** | LR | Link Register (retour fonction) |
| **R15** | PC | Program Counter |

> 🔗 **ARM**
>
> Même organisation que ARM ! L'ABI est compatible.

# La RAM : Une Bibliothèque Numérique

## Analogie : La RAM est comme une bibliothèque

| Bibliothèque | | RAM |
|---|---|---|
| Numéro étagère | = | Adresse |
| Livre | = | Donnée (32 bits) |
| Ranger un livre | = | Écriture (load=1) |
| Consulter | = | Lecture (load=0) |




**Random Access = Accès Direct**

On peut accéder à N'IMPORTE quelle adresse directement, sans parcourir les autres

# La RAM (Random Access Memory)

**RAM = Tableau de registres adressables**



*Interface de la RAM*

# Fonctionnement de la RAM

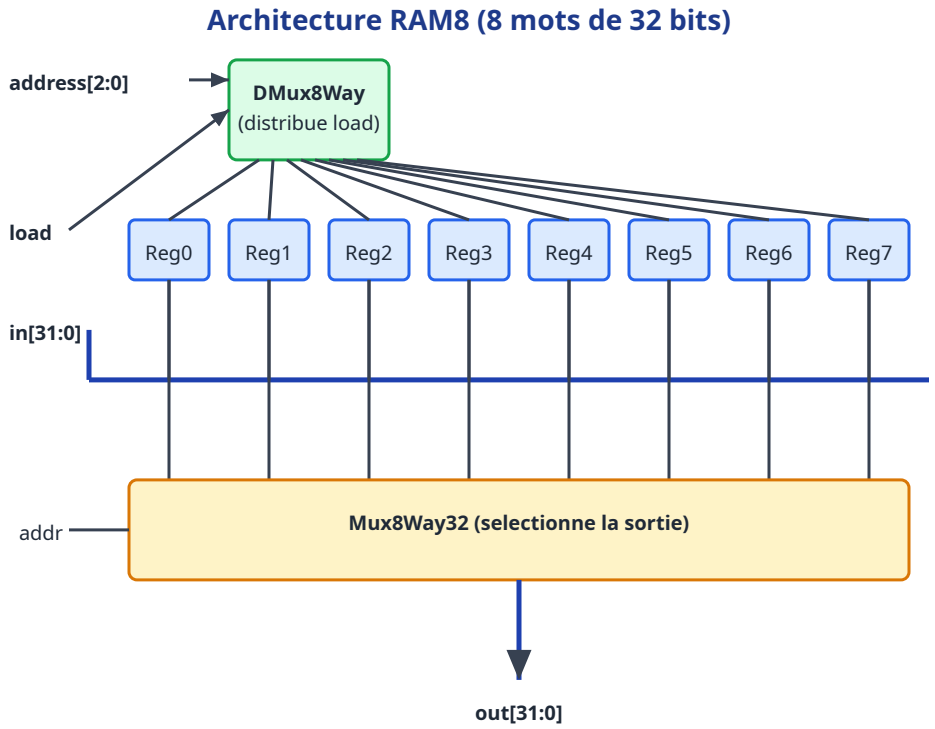
**Lecture (load = 0) :**

- `address` sélectionne une cellule
- `out` = contenu de cette cellule
- Lecture instantanée (combinatoire)

**Écriture (load = 1) :**

- `address` sélectionne une cellule
- `in` est écrit dans cette cellule
- Écriture sur front montant

# Architecture RAM8

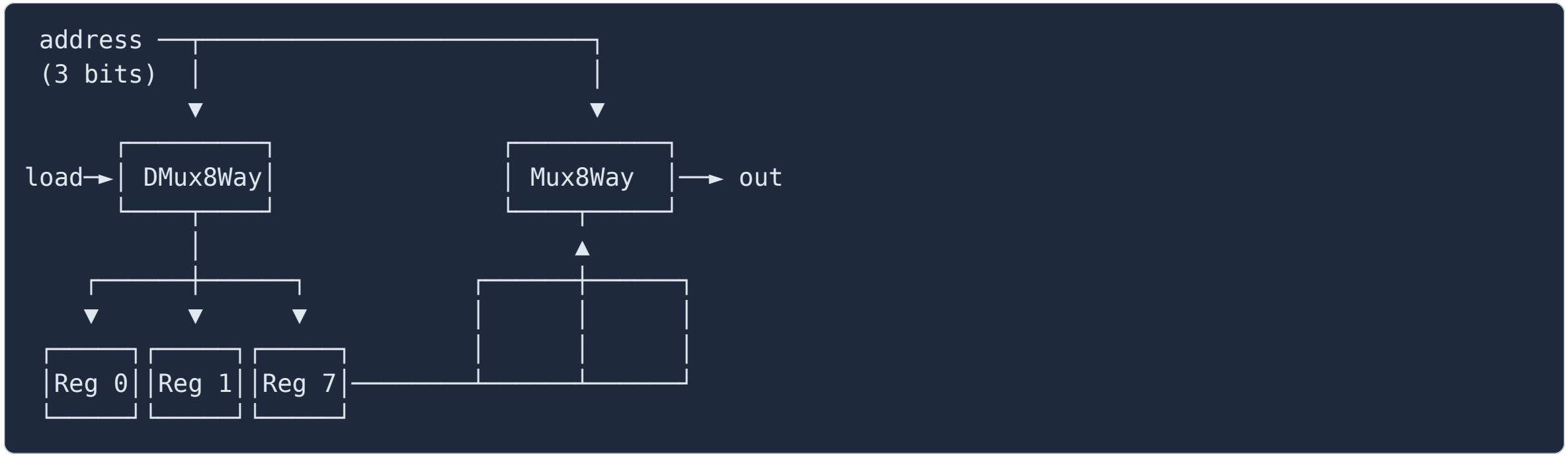



*8 registres avec décodage d'adresse*

## Composants :

- **DMux8Way** : Route le signal load
- **8 Registres** : Stockent les données
- **Mux8Way** : Sélectionne la sortie

# Décodage d'Adresse RAM8



# Construction Hiérarchique

**RAM64 = 8 × RAM8**

```
address[5:0] = [5:3] + [2:0]
                 |       |
         Quelle RAM8    Quel mot dans RAM8
```

**Pattern récursif**

RAM512 = 8 × RAM64, RAM4K = 8 × RAM512, etc.

# Le Compteur de Programme (PC)

**Le PC = le "doigt" qui suit le programme**

```
Adresse  Instruction
───────  ────────────
  0      MOV R0, #5
  1      MOV R1, #3    ◄── PC = 1
  2      ADD R2, R0, R1
  3      ...
```

Le PC pointe vers l'instruction **en cours** (ou la suivante selon l'architecture).

**Question clé :**
Comment le PC sait-il quelle sera la prochaine instruction ?

- Normalement : PC + 1 (séquentiel)
- Parfois : sauter ailleurs (branchement)
- Au démarrage : commencer à 0 (reset)

# Modes du PC (par priorité)

| Priorité | Mode | Action | Usage |
|---|---|---|---|
| 1 | reset | PC ← 0 | Démarrage du CPU |
| 2 | load | PC ← in | Branchement (B, BL) |
| 3 | inc | PC ← PC + 1 | Exécution séquentielle |
| 4 | hold | PC ← PC | Attente (stall) |

**Priorité importante !**

Si reset=1, on ignore tout le reste. Si load=1, on ignore inc. Etc.

# Exemple : Suivons le PC !

```
Addr  Instruction          ; PC après exécution
────  ──────────           ; ──────────────────
 0    MOV R0, #10          ; PC = 1 (inc)
 1    MOV R1, #0           ; PC = 2 (inc)
 2    CMP R0, #0           ; PC = 3 (inc)
 3    B.EQ fin             ; PC = 6 (load!) ou 4 (inc)
 4    ADD R1, R1, R0       ; PC = 5 (inc)
 5    B boucle             ; PC = 2 (load!)
 6    fin: ...
```

**Exécution séquentielle :**

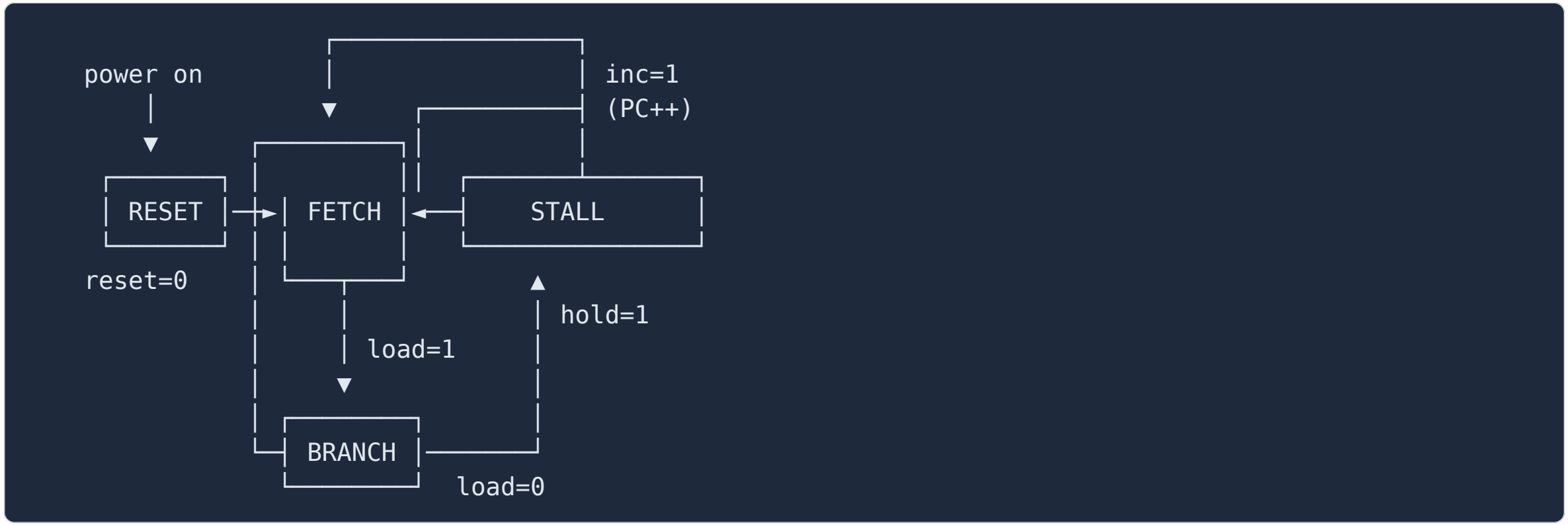
PC = 0 → 1 → 2 → 3 (inc, inc, inc)

**Branchement :**

PC = 3 → 6 si condition vraie (load)

PC = 5 → 2 toujours (load)

# Diagramme d'États du PC



# Implémentation du PC

```
process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      pc <= (others => '0');
    elsif load = '1' then
      pc <= target;
    elsif inc = '1' then
      pc <= pc + 1;
    -- else hold
    end if;
  end if;
end process;
```

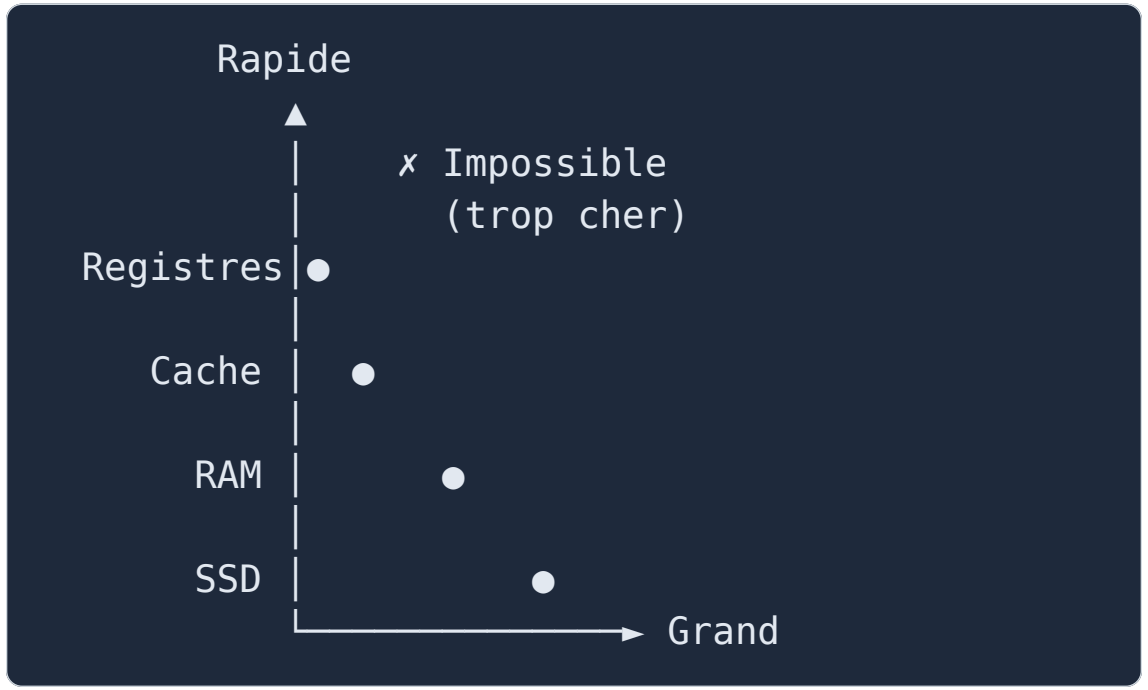
# Cycle d'Exécution du CPU

À chaque cycle d'horloge :

1. **Fetch**
   Lire l'instruction à l'adresse PC
2. **Decode**
   Comprendre l'instruction
3. **Execute**
   Faire le calcul (ALU)
4. **Update PC**
   Incrémenter ou sauter

# Le Compromis Fondamental : Vitesse vs Taille

**Problème : On ne peut pas tout avoir !**




**Pourquoi ?**

- Mémoire rapide = transistors complexes = cher
- Mémoire grande = transistors simples = lent

**Solution :** Utiliser PLUSIEURS niveaux !

# Le Principe de Localité

**Observation clé : Les programmes n'accèdent pas à la mémoire au hasard**

**Localité temporelle :**
Si on accède à une donnée, on y accèdera probablement **bientôt** à nouveau.

```
for (i = 0; i < 1000; i++) {
    sum += i;  // 'sum' accédé 1000 fois !
}
```

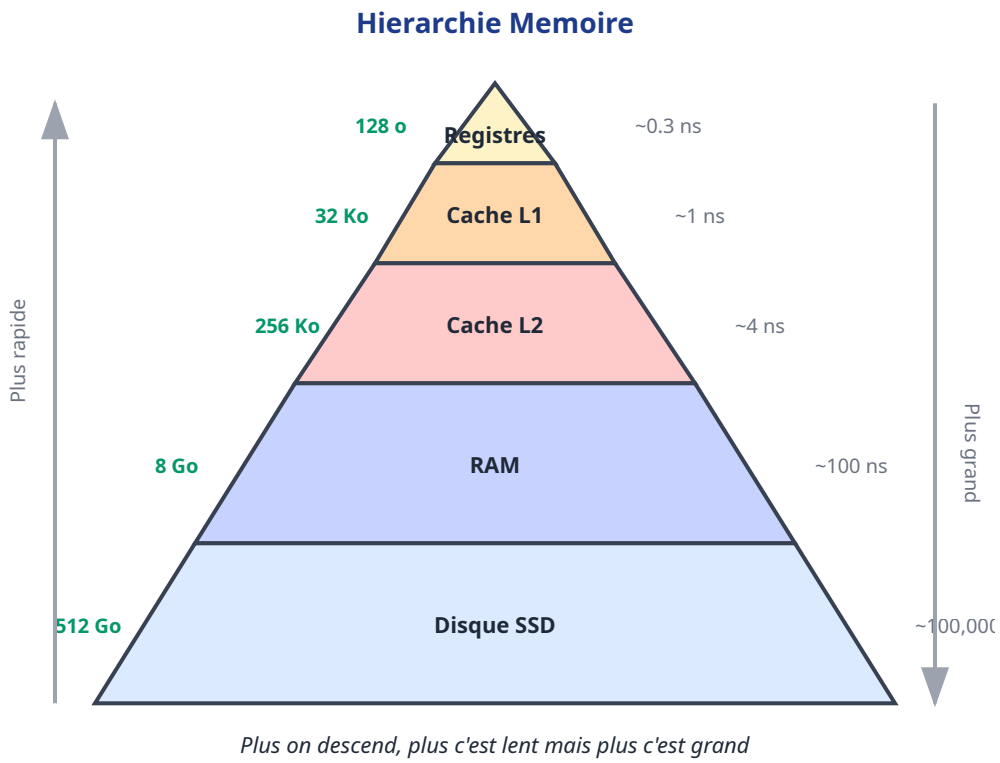
**Localité spatiale :**
Si on accède à une adresse, on accèdera probablement aux adresses **voisines**.

```
for (i = 0; i < 100; i++) {
    sum += tab[i];  // tab[0], tab[1], tab[2]...
}
```

**Idée du cache**

Garder les données récentes/voisines dans une mémoire rapide

# Hiérarchie Mémoire




*Plus rapide en haut, plus grand en bas*

# Comparaison des Niveaux

| Niveau | Capacité | Latence | Technologie |
|---|---|---|---|
| Registres | 16 × 32 bits | 0 cycle | Flip-flops |
| Cache L1 | ~32 KB | 1-3 cycles | SRAM |
| Cache L2 | ~256 KB | 10-20 cycles | SRAM |
| RAM | ~8 GB | 100-300 cycles | DRAM |
| SSD | ~1 TB | 10K+ cycles | Flash |

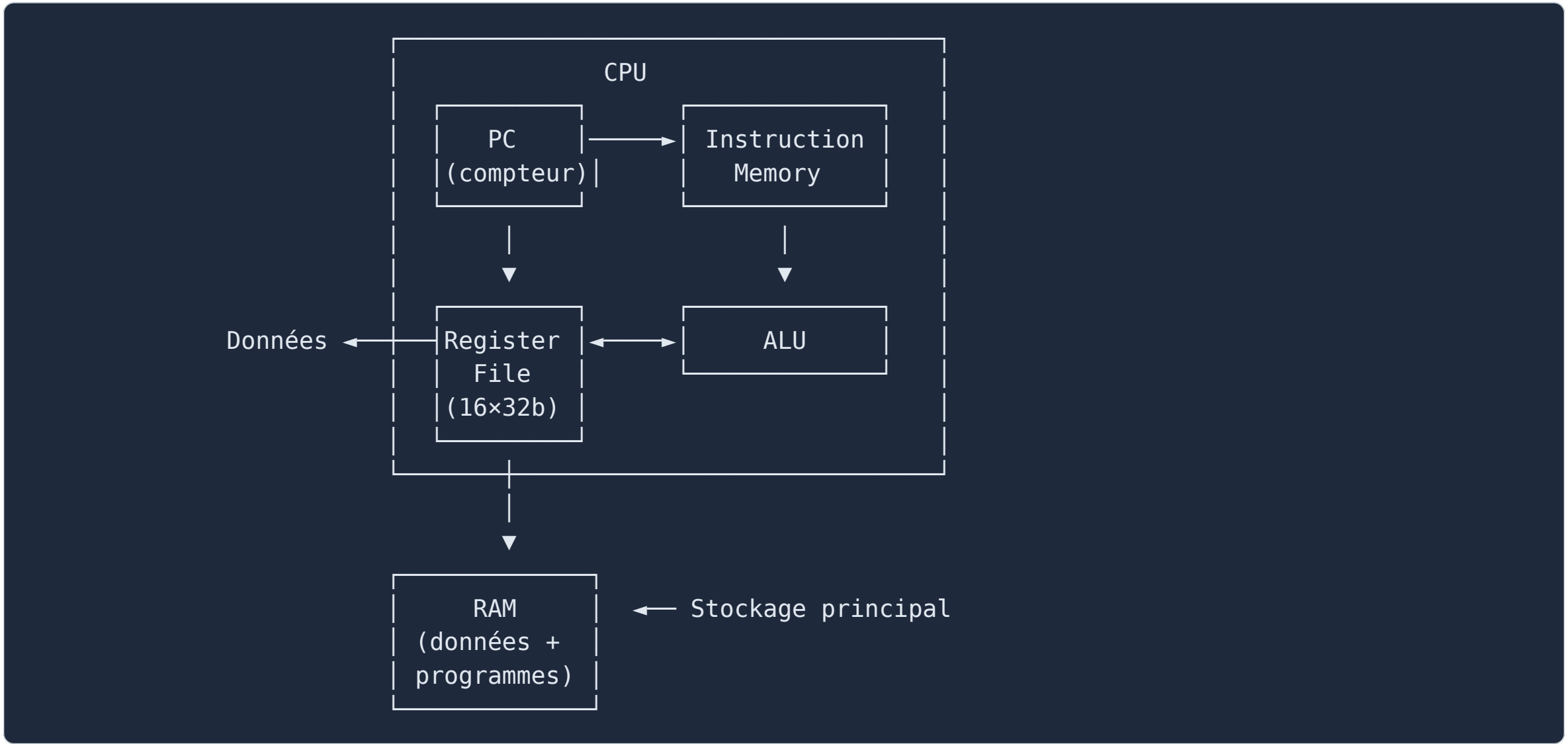
**Illusion de performance**

Grâce à la localité, le CPU "voit" souvent une mémoire rapide (cache hit ~95%)

# Timing Détaillé : Écriture Registre


La nouvelle valeur apparaît après le front montant suivant.

# Vue d'Ensemble : Du Bit au Système




**Tous construits avec des DFF + logique combinatoire !**

# Questions de Réflexion

1. Pourquoi utilise-t-on le front montant plutôt que le niveau haut ?
2. Que se passe-t-il si on lit et écrit la même adresse RAM simultanément ?
3. Combien de DFF faut-il pour une RAM de 1 KB (256 mots de 32 bits) ?
4. Pourquoi le PC a-t-il une priorité sur ses modes ?
5. Comment le CPU sait-il quand la RAM a terminé une lecture ?

# Ce qu'il faut retenir

1. **L'horloge synchronise** : Front montant = capture
2. **DFF = atome** : `q(t) = d(t-1)`
3. **Rétroaction = persistance** : Mux + DFF
4. **RAM = tableau** : DMux + Registres + Mux
5. **PC = guide** : reset > load > inc > hold
6. **Hiérarchie** : Registres > Cache > RAM > Disque

# Questions ?

📚 **Référence :** Livre Seed, Chapitre 03 - Mémoire

👉 **Exercices :** TD et TP disponibles

**Prochain chapitre :** Architecture Machine (ISA)