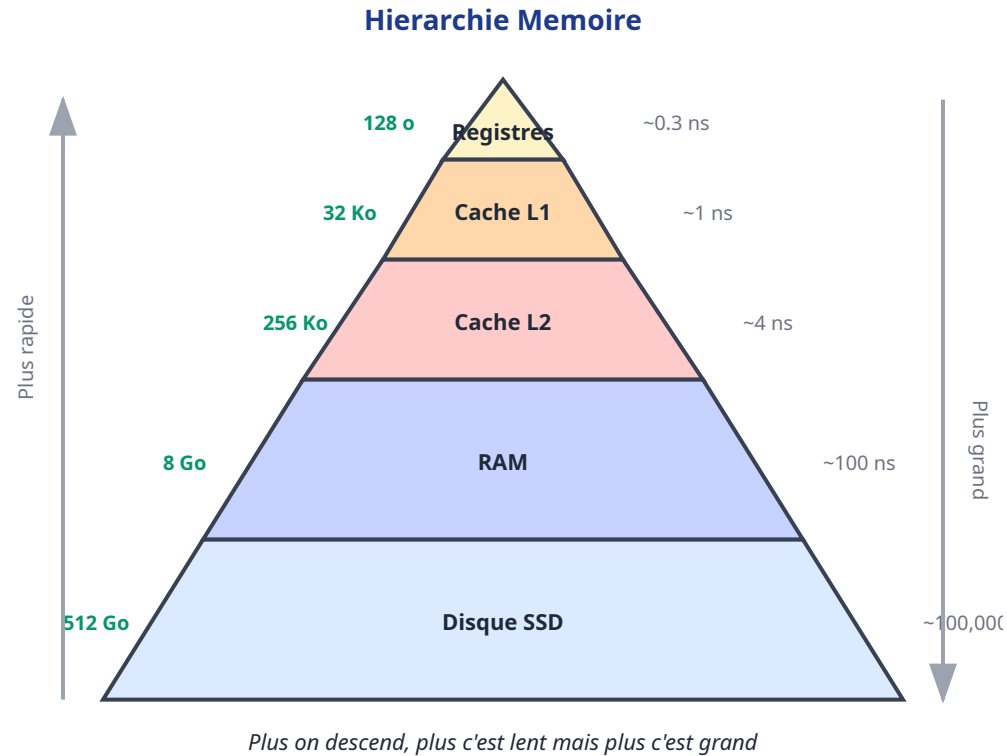


# Chapitre 03 : Logique Séquentielle et Mémoire

"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

---

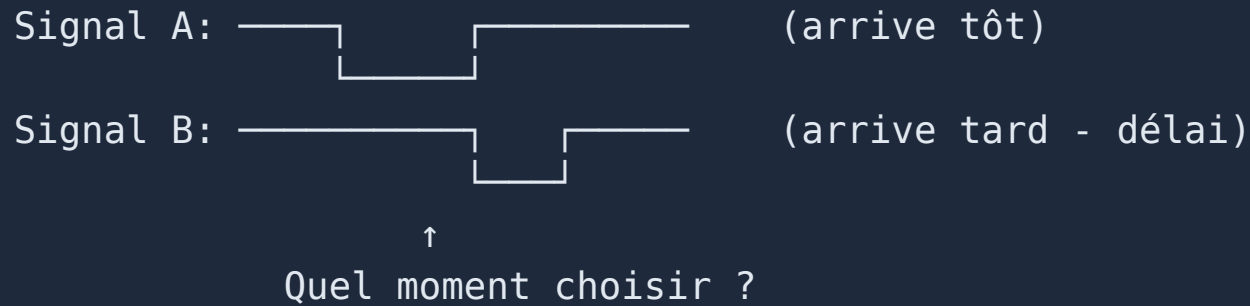
Circuits Combinatoires	Circuits Séquentiels
Sortie = $f(\text{entrées})$	Sortie = $f(\text{entrées}, \text{état})$
Pas de mémoire	A de la mémoire
Pas d'horloge	Synchronisé par horloge
Ex: AND, OR, ALU	Ex: Registres, RAM, CPU

### 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)

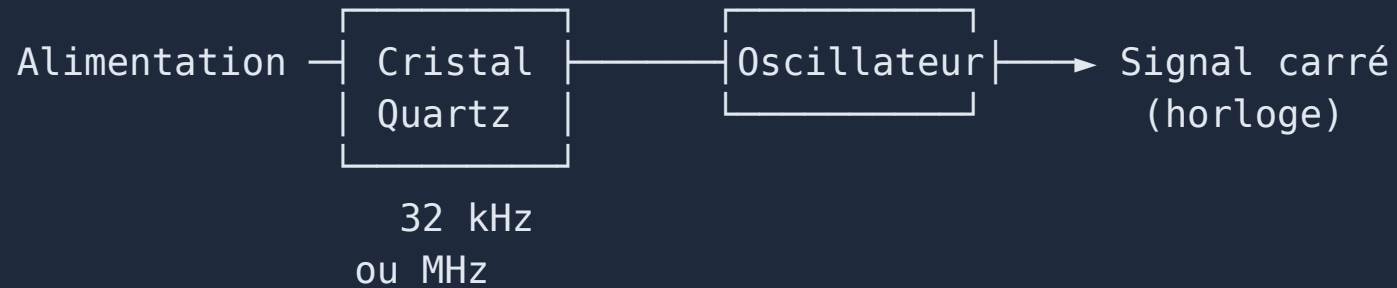


**ARM**

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 :

entrée → sortie

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

AVEC horloge :

entrée → [attend] → sortie  
                  ↑  
                front  
                montant

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

## Révélation

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

## La Bascule D (DFF)

---

DFF = Data Flip-Flop = atome de mémoire




*Symbole de la DFF*

Règle fondamentale :

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

La sortie = l'entrée du cycle précédent

## Comportement de la DFF

clk: 

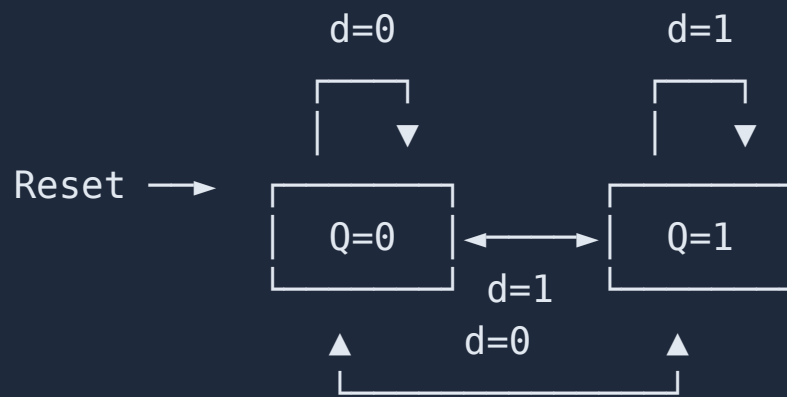
d: —[A]—[B]—[C]—[D]—

q: —[?]—[A]—[B]—[C]—

### 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.

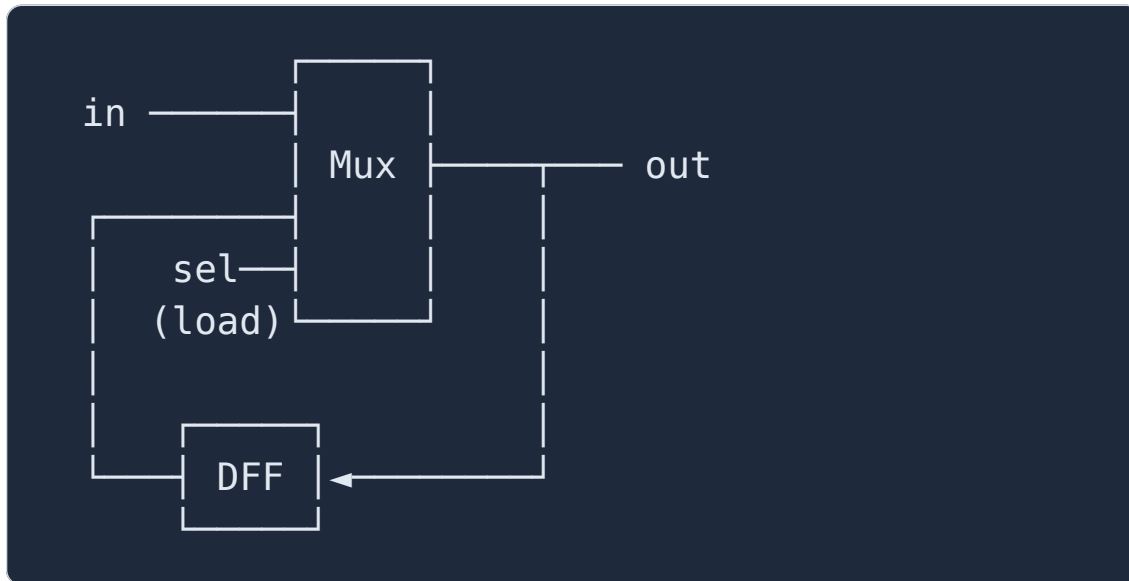
On veut :

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

### Besoin

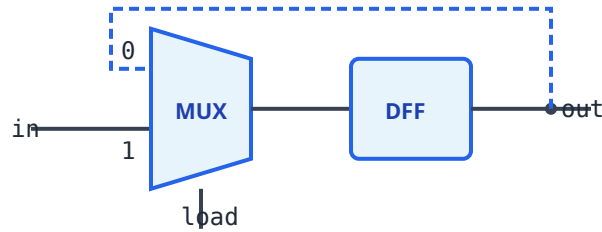
Un signal de contrôle pour décider quand écrire

## 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** !

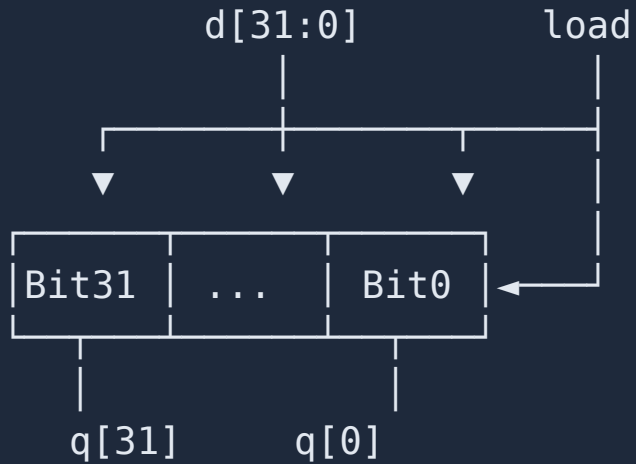
## VHDL : Registre avec Load

### VHDL

```
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

---

Banc de Registres (32 bits × 16)						
<b>R0</b> Arg0	<b>R1</b> Arg1	<b>R2</b> Arg2	<b>R3</b> Arg3	<b>R4</b> Var	<b>R5</b> Var	<b>R6</b> Var
<b>R7</b> Var	<b>R8</b> Var	<b>R9</b> Var	<b>R10</b> Var	<b>R11</b> Var	<b>R12</b> Temp	<b>R13</b> SP
<b>R14 (LR)</b>		<b>R15 (PC)</b>				

*16 registres avec 2 ports lecture, 1 port écriture*

## Interface :

- 2 ports de lecture (Ra, Rb)
- 1 port d'écriture (Rd)
- Signal write enable

## Registres du CPU nand2c

---

	Alias	Rôle
<b>R0-R12</b>	-	Registres généraux
<b>R13</b>	SP	Stack Pointer
<b>R14</b>	LR	Link Register (retour fonction)
<b>R15</b>	PC	Program Counter

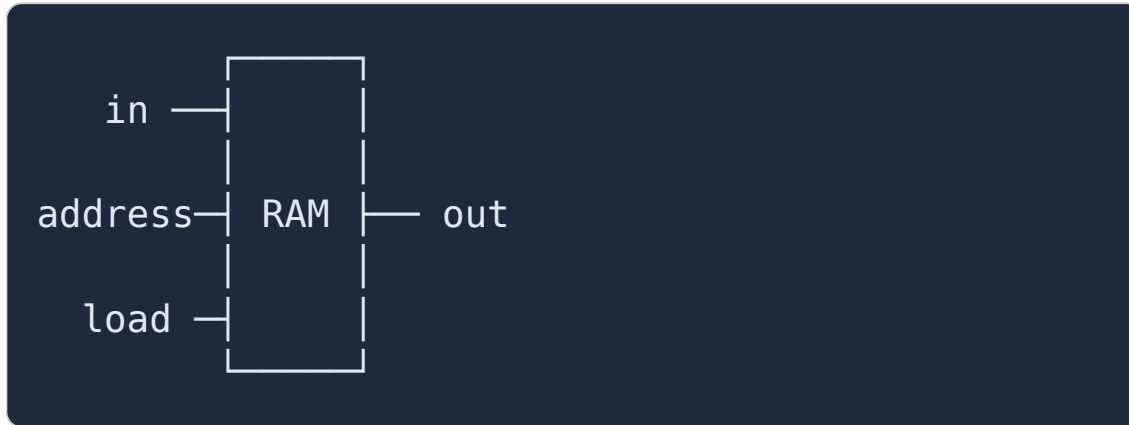
### ARM

Même organisation que ARM ! L'ABI est compatible.

## La RAM (Random Access Memory)

---

RAM = Tableau de registres adressables



*Interface de la RAM*

## Fonctionnement de la RAM

---

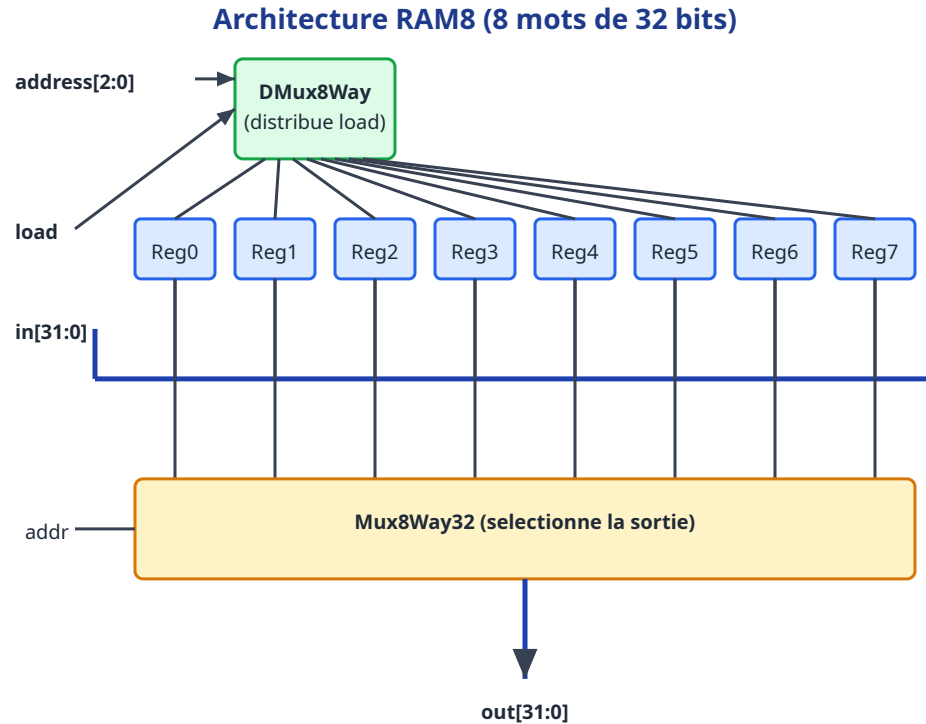
### Lecture (load = 0) :

- `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

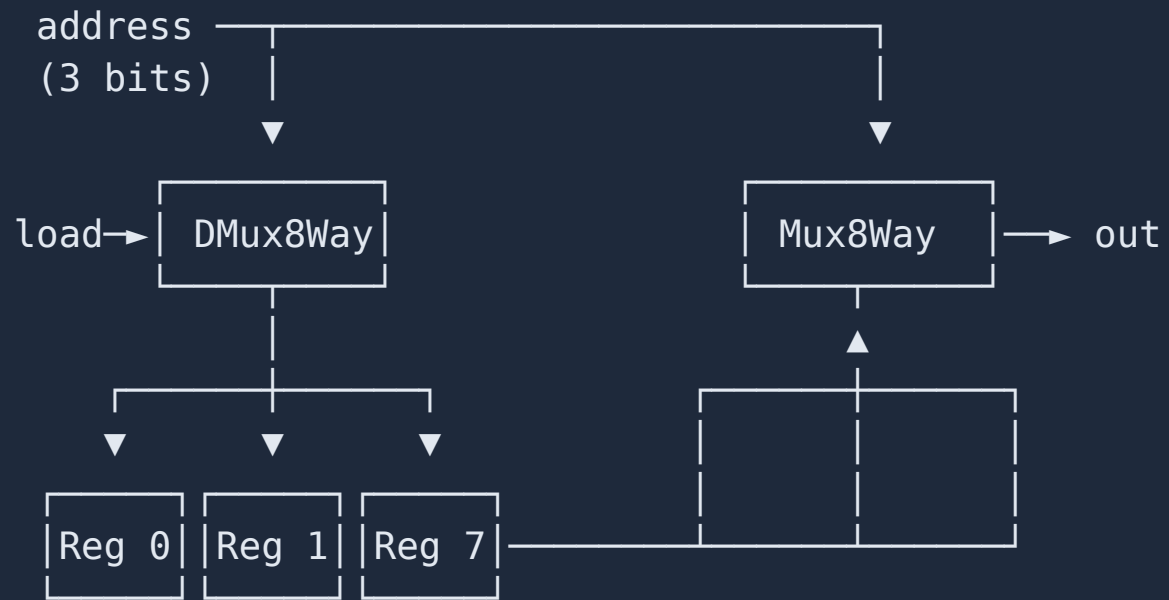


*8 registres avec décodage d'adresse*

## Composants :

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

## Décodage d'Adresse RAM8



## Construction Hiérarchique

---

$\text{RAM64} = 8 \times \text{RAM8}$

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

### Pattern récursif

$\text{RAM512} = 8 \times \text{RAM64}$ ,  $\text{RAM4K} = 8 \times \text{RAM512}$ , etc.

## Le Compteur de Programme (PC)

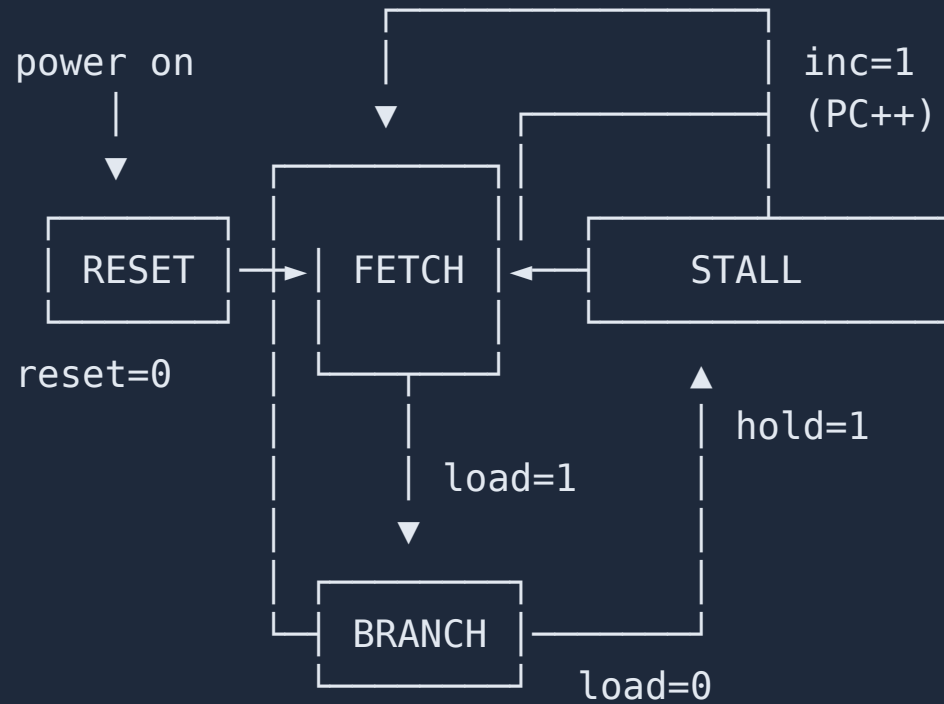
---

Le PC contient l'adresse de la **prochaine instruction**.

**Modes (par priorité) :**

Priorité	Mode	Action	Usage
1	reset	$PC \leftarrow 0$	Démarrage
2	load	$PC \leftarrow in$	Branchement
3	inc	$PC \leftarrow PC + 1$	Séquentiel
4	hold	$PC \leftarrow PC$	Stall

## 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;
```

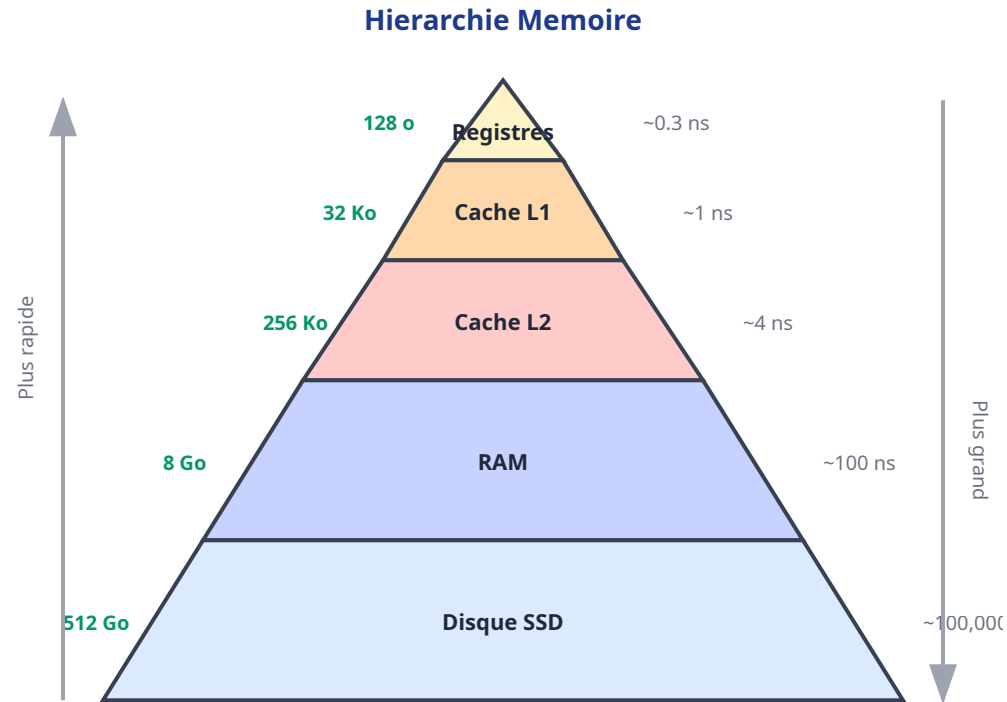
# Cycle d'Exécution du CPU

---

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

# Hiérarchie Mémoire



*Plus on descend, plus c'est lent mais plus c'est grand*

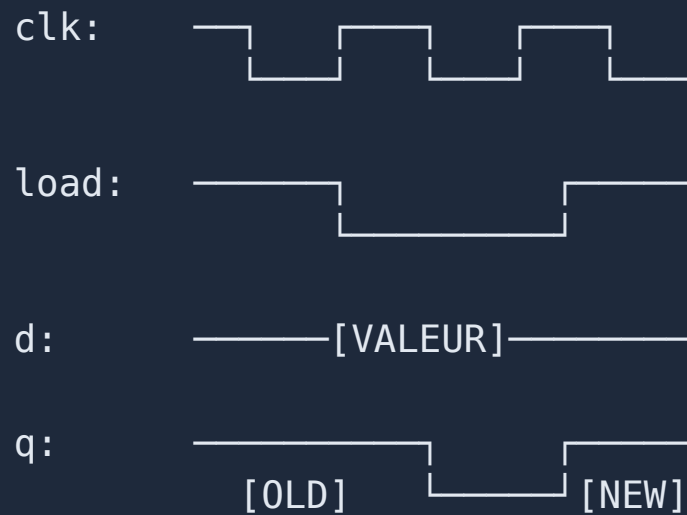
*Plus rapide en haut, plus grand en bas*

## 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

## Timing Détaillé : Écriture Register



La nouvelle valeur apparaît après le front montant suivant.

## 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)