

# L'Architecture Codex

De la Porte Logique au Système d'Exploitation 32-bits

Projet Nand2Tetris-Codex

2026

## Table des matières

<b>1</b>	<b>L'Architecture Codex : Guide de l'Étudiant</b>	<b>4</b>
1.1	Table des Matières . . . . .	4
1.2	Comment utiliser ce guide . . . . .	5
<b>2</b>	<b>Introduction</b>	<b>5</b>
2.1	Le Mystère de l'Ordinateur . . . . .	6
2.2	Pourquoi Construire un Ordinateur ? . . . . .	6
2.3	Ce que Vous Allez Construire . . . . .	6
2.4	Codex vs Hack (Nand2Tetris Original) . . . . .	8
2.5	Ce que Vous Allez Apprendre . . . . .	8
2.6	Vos Outils . . . . .	9
2.7	Comment Utiliser ce Livre . . . . .	10
2.8	La Grande Aventure Commence . . . . .	10
<b>3</b>	<b>Logique Booléenne</b>	<b>10</b>
3.1	Où en sommes-nous ? . . . . .	11
3.2	Pourquoi le Binaire ? . . . . .	11
3.3	La Porte NAND : Notre Axiome . . . . .	12
3.4	Construction des Portes Élémentaires . . . . .	13
3.5	Le Lien avec l'Ordinateur Complet . . . . .	16
3.6	Description Matérielle (Codex HDL) . . . . .	16
3.7	Exercices Pratiques . . . . .	19
3.8	Défis Supplémentaires . . . . .	20
3.9	Ce qu'il faut retenir . . . . .	21
<b>4</b>	<b>Arithmétique Binaire</b>	<b>21</b>
4.1	Où en sommes-nous ? . . . . .	21
4.2	Pourquoi l'Arithmétique est-elle si Importante ? . . . . .	22
4.3	Représentation des Nombres . . . . .	23
4.4	L'Addition Binaire . . . . .	24
4.5	Le Demi-Additionneur (Half Adder) . . . . .	25
4.6	L'Additionneur Complet (Full Adder) . . . . .	26
4.7	L'Additionneur 32-bits (Ripple Carry Adder) . . . . .	27

4.8	L'ALU (Arithmetic Logic Unit)	27
4.9	Architecture de l'ALU	29
4.10	Exercices Pratiques	30
4.11	Défis Supplémentaires	31
4.12	Le Lien avec le CPU	31
4.13	Ce qu'il faut retenir	31
<b>5</b>	<b>Logique Séquentielle et Mémoire</b>	<b>32</b>
5.1	Où en sommes-nous ?	32
5.2	Pourquoi la Mémoire est-elle Fondamentale ?	33
5.3	Le Temps et l'Horloge (Clock)	33
5.4	La Bascule D (D Flip-Flop / DFF)	34
5.5	Le Registre 1-bit (Bit)	35
5.6	Le Registre 32-bits	36
5.7	La RAM (Random Access Memory)	36
5.8	Le Compteur de Programme (PC)	38
5.9	Les Différents Types de Mémoire	39
5.10	Exercices Pratiques	39
5.11	Défis Supplémentaires	40
5.12	Le Lien avec la Suite	41
5.13	Ce qu'il faut retenir	41
<b>6</b>	<b>Architecture Machine (ISA A32)</b>	<b>42</b>
6.1	Où en sommes-nous ?	42
6.2	Qu'est-ce qu'une Architecture ?	43
6.3	Pourquoi RISC ? L'architecture Load/Store	43
6.4	Le Cycle de Vie d'une Instruction	44
6.5	Les Registres : Le Plan de Travail	44
6.6	La Carte Mémoire (Memory Map)	45
6.7	Le Format des Instructions	46
6.8	Les Instructions en Détail	47
6.9	La Pile (Stack)	48
6.10	Exemples de Programmes	49
6.11	Gestion des Erreurs (Traps)	50
6.12	Exercices Pratiques	51
6.13	Ce qu'il faut retenir	51
<b>7</b>	<b>Le Processeur (CPU)</b>	<b>52</b>
7.1	Où en sommes-nous ?	52
7.2	Deux Implémentations du CPU	53
7.3	Qu'est-ce qu'un CPU ?	54
7.4	Architecture du CPU (Data Path)	55
7.5	Les Composants du CPU	57
7.6	Le Cycle d'Exécution en Détail	59
7.7	Implémentation du CPU en HDL	60
7.8	Exercices Pratiques	61
7.9	CPU Visualizer : L'Outil Interactif	62
7.10	Conseils de Débogage	64

7.11	Aller Plus Loin : Le CPU Pipeline . . . . .	64
7.12	Le Lien avec la Suite . . . . .	76
7.13	Ce qu'il faut retenir . . . . .	76
<b>8</b>	<b>L'Assembleur</b>	<b>77</b>
8.1	Où en sommes-nous ? . . . . .	77
8.2	Le Rôle de l'Assembleur . . . . .	78
8.3	La Stratégie des Deux Passes . . . . .	78
8.4	Sections et Directives . . . . .	79
8.5	Exemple d'Encodage . . . . .	79
8.6	La Gestion des Grandes Constantes . . . . .	80
8.7	Exercices Pratiques . . . . .	80
8.8	Ce qu'il faut retenir . . . . .	81
<b>9</b>	<b>Construction du Compilateur</b>	<b>82</b>
9.1	Où en sommes-nous ? . . . . .	82
9.2	Le Rôle du Compilateur . . . . .	82
9.3	Les Phases du Compilateur . . . . .	83
9.4	Compilation des Structures de Contrôle . . . . .	84
9.5	Compilation des Fonctions . . . . .	85
9.6	Le Compilateur C32 . . . . .	86
9.7	Construisez Votre Propre Compilateur ! . . . . .	87
9.8	Exercices Pratiques . . . . .	89
9.9	Ce qu'il faut retenir . . . . .	89
<b>10</b>	<b>Langage de Haut Niveau (C32)</b>	<b>90</b>
10.1	Où en sommes-nous ? . . . . .	90
10.2	Pourquoi un Langage de Haut Niveau ? . . . . .	90
10.3	Spécification du Langage C32 . . . . .	91
10.4	Opérateurs . . . . .	92
10.5	Structures de Contrôle . . . . .	93
10.6	Fonctions . . . . .	93
10.7	Pointeurs et Tableaux . . . . .	94
10.8	Accès au Matériel (MMIO) . . . . .	94
10.9	Exemple Complet . . . . .	95
10.10	Exercices Pratiques . . . . .	96
10.11	Les Structures (Structs) . . . . .	96
10.12	Limitations du C32 . . . . .	98
10.13	Ce qu'il faut retenir . . . . .	98
<b>11</b>	<b>Système d'Exploitation</b>	<b>98</b>
11.1	Où en sommes-nous ? . . . . .	99
11.2	Qu'est-ce qu'un Système d'Exploitation ? . . . . .	99
11.3	Gestion de la Mémoire (Le Tas / Heap) . . . . .	100
11.4	Bibliothèque Graphique . . . . .	100
11.5	Entrées / Sorties . . . . .	101
11.6	Interruptions et Timer (Concepts) . . . . .	102
11.7	Applications Démo . . . . .	103

11.8 Exercices Pratiques . . . . .	103
11.9 Le Parcours Complet . . . . .	104
11.10 Ce qu'il faut retenir . . . . .	104
11.11 Félicitations ! . . . . .	104
<b>12 Annexe : Tous les Exercices</b>	<b>105</b>
12.1A. Exercices HDL (Portes Logiques) . . . . .	105
12.2B. Exercices Assembleur A32 . . . . .	106
12.3C. Exercices C32 . . . . .	109
12.4D. Construction du Compilateur . . . . .	111
12.5E. Systeme d'Exploitation . . . . .	113
12.6 Conseils pour les Exercices . . . . .	114
<b>13 Le Cache : Pourquoi Votre Ordinateur Semble Rapide</b>	<b>114</b>
13.1 Le Problème : La RAM est Lente ! . . . . .	115
13.2 La Solution : Le Principe de Localité . . . . .	116
13.3 Comment Fonctionne le Cache ? . . . . .	117
13.4 Hit ou Miss : Que se passe-t-il ? . . . . .	119
13.5 Politiques d'Écriture . . . . .	121
13.6 Impact sur vos Programmes . . . . .	122
13.7 Implémentation HDL du Cache . . . . .	125
13.8 Statistiques et Performance . . . . .	128
13.9 Visualiser le Cache avec le CPU Visualizer . . . . .	129
13.10 Exercices . . . . .	130
13.11 Résumé Visuel . . . . .	131
13.12 Points Clés à Retenir . . . . .	131

## 1 L'Architecture Codex : Guide de l'Étudiant

Ce livre accompagne le projet **Nand2Tetris-Codex**. Il a pour but de vous guider pas à pas dans la construction d'un ordinateur complet, moderne et 32-bits, à partir de rien.

### 1.1 Table des Matières

#### 1. Introduction

- La philosophie du projet.
- Différences avec le Hack (Nand2Tetris original).
- Présentation des outils.

#### 2. Chapitre 1 : Logique Booléenne

- L'abstraction binaire.
- Les portes logiques fondamentales (Nand, And, Or, Xor).
- *Exercices* : Implémentation dans `hdl_lib/01_gates`.

#### 3. Chapitre 2 : Arithmétique Binaire (À venir)

- Représentation des nombres (Complément à 2).
- Additionneurs et ALU (Arithmetic Logic Unit).
- *Exercices* : `hdl_lib/03_arith`.

4. **Chapitre 3 : Logique Séquentielle et Mémoire** (À venir)
  - Le temps et l'horloge.
  - Flip-Flops, Registres et RAM.
  - *Exercices* : `hdl_lib/04_seq` & `02_multibit`.
5. **Chapitre 4 : Architecture Machine (ISA A32)** (À venir)
  - Le jeu d'instructions A32-Lite.
  - Registres, adressage et structure.
  - *Référence* : `SPECS.md`.
6. **Chapitre 5 : Le Processeur (CPU)** (À venir)
  - Implémentation du chemin de données (Data Path).
  - Logique de contrôle.
  - *Exercices* : `hdl_lib/05_cpu`.
7. **Chapitre 6 : L'Assembleur** (À venir)
  - Traduction symbolique vers binaire.
  - Gestion des symboles et des labels.
  - *Outils* : `a32_asm`.
8. **Chapitre 7 : Construction du Compilateur** (À venir)
  - Analyse lexicale et syntaxique (Parsing).
  - Génération de code pour pile.
  - *Outils* : `c32_cli`.
9. **Chapitre 8 : Langage de Haut Niveau (C32)** (À venir)
  - Syntaxe du langage C32.
  - Structures de contrôle et types.
10. **Chapitre 9 : Système d'Exploitation** (À venir)
  - Gestion de la mémoire (Heap).
  - Entrées/Sorties (Clavier, Écran).
  - *Exercices* : `os_lib`.

## 1.2 Comment utiliser ce guide

Chaque chapitre commence par une **Théorie**, expliquant les concepts fondamentaux. Ensuite, la section **Implémentation** détaille ce que vous devez construire, en faisant référence aux dossiers du projet.

**Note** : Ce projet utilise Rust pour l'outillage, mais votre travail consistera principalement à écrire du HDL (Hardware Description Language), de l'assembleur A32 et du C32.

## 2 Introduction

Bienvenue dans le projet **Codex**. Si vous lisez ceci, c'est que vous avez l'ambition de comprendre comment fonctionnent les ordinateurs — non pas en lisant des théories abstraites, mais en en construisant un vous-même, de zéro.

## 2.1 Le Mystère de l'Ordinateur

Prenez un instant pour réfléchir à ce qui se passe quand vous tapez une lettre sur votre clavier :

1. Vos doigts appuient sur une touche physique
2. Un signal électrique est envoyé
3. Ce signal est transformé en code numérique
4. Le processeur le détecte et l'interprète
5. Un programme décide quoi faire de cette information
6. Des pixels s'allument sur votre écran pour afficher la lettre

Entre votre doigt et le pixel, il y a des **dizaines de couches d'abstraction**. Chaque couche fait confiance à celle en dessous et simplifie la vie de celle au-dessus.

Ce livre va vous faire traverser **toutes ces couches**, de la plus basse (les portes logiques) à la plus haute (les applications).

## 2.2 Pourquoi Construire un Ordinateur ?

### 2.2.1 Le problème de la “boîte noire”

Dans notre vie quotidienne, l'ordinateur est une “boîte noire”. Nous tapons sur un clavier, touchons un écran, et la magie opère. En tant qu'ingénieurs logiciels, nous travaillons souvent sur des couches d'abstraction très élevées (Python, Java, React).

Cette abstraction est une bénédiction pour la productivité, mais elle crée un **fossé de compréhension**. Combien de développeurs savent vraiment : - Comment le processeur exécute leur code ? - Pourquoi certaines opérations sont rapides et d'autres lentes ? - Ce qui se passe quand on écrit `x = 5` ? - Comment une image apparaît à l'écran ?

### 2.2.2 L'approche “Du NAND au Tetris”

Ce projet a pour but de **briser l'abstraction**. Nous allons descendre au niveau le plus bas — la porte logique — et remonter couche par couche jusqu'à pouvoir jouer à un jeu vidéo écrit dans un langage de haut niveau sur notre propre machine.

À la fin de ce parcours, quand vous verrez du code s'exécuter, vous saurez **exactement** ce qui se passe dans la machine. Ce n'est plus de la magie — c'est de l'ingénierie que vous maîtrisez.

## 2.3 Ce que Vous Allez Construire

Voici les couches que nous allons traverser, de bas en haut :

COUCHE 7: Applications (Jeux, Shell, Calculatrice, Éditeur)
C'est ce que l'utilisateur voit et utilise. Tout le reste n'existe que pour rendre cette couche possible.

<p>COUCHE 6: Système d'Exploitation (Gestion mémoire, Drivers écran/clavier)</p> <p>L'OS cache la complexité du matériel. Au lieu de manipuler des adresses, on appelle <code>draw_pixel()</code> ou <code>read_key()</code>.</p>
<p>COUCHE 5: Langage de Haut Niveau (C32) (Variables, fonctions, boucles)</p> <p>On écrit du code lisible par un humain. Le compilateur le transforme en instructions machine.</p>
<p>COUCHE 4: Compilateur (Transforme C32 → Assembleur A32)</p> <p>Le compilateur est un traducteur automatique. Il comprend votre intention et génère le code équivalent.</p>
<p>COUCHE 3: Assembleur (A32 ASM) (Mnémoniques → Code machine binaire)</p> <p>L'assembleur traduit les instructions humainement lisibles (ADD, MOV) en suites de 0 et de 1 que le CPU comprend.</p>
<p>COUCHE 2: Architecture Machine (ISA) (Jeu d'instructions, Registres, Mémoire)</p> <p>C'est le "contrat" entre le matériel et le logiciel. Il définit ce que le processeur sait faire.</p>
<p>COUCHE 1: Logique Matérielle (Portes logiques, ALU, RAM, CPU)</p> <p>Des circuits électroniques qui ne connaissent que 0 et 1, mais qui, combinés intelligemment, peuvent tout calculer.</p>
<p>COUCHE 0: La Porte NAND (Notre axiome de départ)</p> <p>Tout commence ici. Une seule porte logique, et à partir d'elle, nous construisons tout le reste.</p>

### 2.3.1 La beauté de l'abstraction

Chaque couche a une propriété remarquable : **elle n'a besoin de connaître que la couche juste en dessous.**

- Le programmeur C32 n’a pas besoin de savoir comment fonctionne l’ALU
- L’ALU n’a pas besoin de savoir qu’elle va être utilisée pour un jeu vidéo
- La porte NAND ne “sait” pas qu’elle fait partie d’un ordinateur

Cette séparation des préoccupations est ce qui rend possible la construction de systèmes complexes.

## 2.4 Codex vs Hack (Nand2Tetris Original)

Ce cours est fortement inspiré du légendaire “Nand2Tetris” de Noam Nisan et Shimon Schocken. Cependant, l’ordinateur *Hack* original a été conçu pour une simplicité maximale, parfois au détriment du réalisme moderne.

L’ordinateur **Codex** propose une approche différente :

Caractéristique	Hack (Original)	Codex (Ce projet)
<b>Architecture</b>	16-bits	<b>32-bits</b>
<b>Registres</b>	2 (A et D)	<b>16 (R0-R15)</b> style ARM
<b>Mémoire</b>	Séparée (Harvard)	<b>Unifiée</b> (Von Neumann)
<b>Instructions</b>	Simple, propriétaire	<b>RISC moderne</b> (Load/Store)
<b>Écran</b>	Monochrome fixe	<b>320×240 couleurs</b>

### 2.4.1 Pourquoi ces changements ?

1. **32 bits** : C’est la taille standard des machines modernes (avant 64 bits). Cela permet d’adresser 4 Go de mémoire et de manipuler des nombres plus grands.
2. **16 registres** : Les processeurs ARM (smartphones, Raspberry Pi) utilisent aussi des registres R0-R15. Comprendre Codex, c’est comprendre ARM.
3. **Architecture RISC** : Les instructions sont simples et régulières. Le CPU fait une chose à la fois, mais le fait vite.
4. **Load/Store** : Le CPU ne calcule jamais directement en mémoire. Il charge d’abord les données dans des registres, calcule, puis stocke le résultat. C’est plus simple à implémenter et à comprendre.

## 2.5 Ce que Vous Allez Apprendre

À la fin de ce livre, vous saurez :

**Au niveau matériel** : - Comment construire des portes logiques à partir de NAND - Comment un additionneur transforme des bits en nombres - Comment la mémoire “se souvient” des données - Comment le CPU orchestre tout cela

**Au niveau logiciel** : - Comment l’assembleur traduit les mnémoniques en binaire - Comment un compilateur transforme du code lisible en instructions - Comment un OS simplifie l’accès au matériel - Comment une application utilise toutes ces couches



**Au niveau conceptuel :** - Pourquoi les ordinateurs utilisent le binaire - Comment l'abstraction permet de gérer la complexité - Pourquoi certaines opérations sont "coûteuses" - Comment le même matériel peut faire des choses très différentes

## 2.6 Vos Outils

Le projet est fourni avec une suite d'outils performants :

### 2.6.1 Les Outils en Ligne de Commande

Outil	Rôle	Exemple
<code>hdl_cli</code>	Simule vos circuits HDL	<code>hdl_cli test Not.hdl</code>
<code>a32_cli</code>	Assemble le code A32 → binaire	<code>a32_cli prog.s -o prog.bin</code>
<code>c32_cli</code>	Compile le code C32 → assembleur	<code>c32_cli prog.c -o prog.s</code>
<code>a32_runner</code>	Exécute le code binaire	<code>a32_runner prog.bin</code>

### 2.6.2 Le Simulateur Web (Recommandé)

Pour une expérience plus visuelle et interactive, utilisez le **Simulateur Web**. Il vous permet de : - Écrire et tester votre HDL directement dans le navigateur - Voir l'état des signaux en temps réel avec des chronogrammes - Compiler et exécuter du code C et Assembleur - Visualiser l'écran, les registres et la mémoire

Pour le lancer :

```
cd web
npm install
npm run dev
```

Ouvrez ensuite votre navigateur à l'adresse indiquée (généralement `http://localhost:5173`).

### 2.6.3 Le CPU Visualizer

Le **CPU Visualizer** est un outil pédagogique spécialement conçu pour comprendre le fonctionnement du processeur. Il affiche en temps réel :

- **Le pipeline** : Les 5 étapes d'exécution (Fetch, Decode, Execute, Memory, Writeback) s'illuminent au fur et à mesure
- **Les registres** : R0-R15 avec mise en évidence des modifications
- **Les flags** : N, Z, C, V avec animations lors des changements
- **Le code source** : Coloration syntaxique et surlignage de la ligne en cours d'exécution
- **Le cache** : Statistiques (hits/misses) et contenu des lignes cache

Accédez-y depuis le menu principal du Simulateur Web ou directement via </visualizer.html>.

**7 démos intégrées** vous permettent d'explorer différents concepts : 1. Addition simple 2. Boucles et branchements 3. Accès mémoire (LDR/STR) 4. Conditions et prédication 5. Tableaux 6. Flags CPU 7. Comportement du cache

## 2.7 Comment Utiliser ce Livre

### 2.7.1 L'approche recommandée

1. **Lisez chaque chapitre en entier** avant de commencer les exercices
2. **Faites les exercices dans l'ordre** — chaque exercice prépare le suivant
3. **Ne regardez pas les solutions** avant d'avoir vraiment essayé
4. **Utilisez le simulateur web** pour visualiser ce qui se passe
5. **Reliez toujours à l'ensemble** — demandez-vous “où cela s'insère-t-il ?”

### 2.7.2 Si vous êtes bloqué

- Relisez la section correspondante du chapitre
- Vérifiez que vous avez bien compris les exercices précédents
- Utilisez le débogueur visuel du simulateur web
- Les erreurs les plus fréquentes sont des problèmes de câblage (mauvaises connexions)

## 2.8 La Grande Aventure Commence

Vous êtes sur le point d'entreprendre un voyage fascinant. Chaque chapitre vous rapprochera un peu plus de la compréhension totale de la machine.

Quand vous aurez terminé, vous regarderez votre ordinateur différemment. Ce ne sera plus une boîte noire mystérieuse, mais une symphonie d'abstractions que vous pouvez comprendre, modifier, et même reconstruire.

Prêt ? Passons à la première brique élémentaire : la logique booléenne.

---

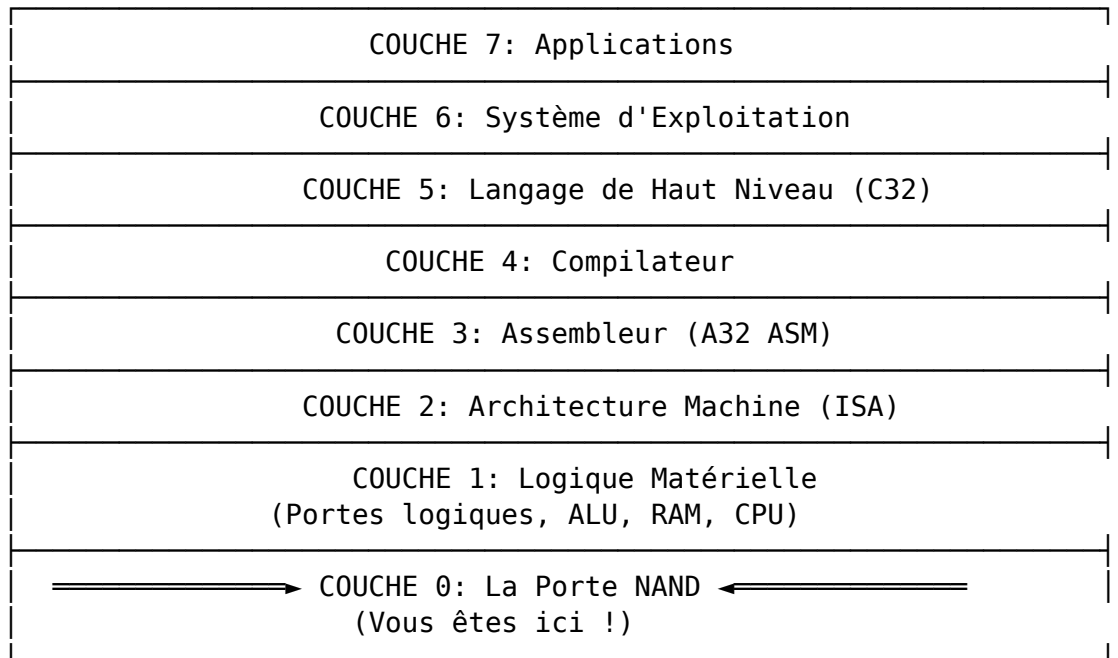
**Rappel important** : Chaque chapitre de ce livre construit sur les précédents. Résistez à la tentation de sauter des étapes — la compréhension profonde vient de la construction progressive.

## 3 Logique Booléenne

“Au commencement était le NAND.”

Tout ordinateur numérique, aussi complexe soit-il, est construit à partir de concepts incroyablement simples : le Vrai (1) et le Faux (0). Ce chapitre traite de la construction de portes logiques élémentaires à partir d'une brique fondamentale : la porte NAND.

### 3.1 Où en sommes-nous ?



Nous commençons tout en bas de la pyramide. C'est ici que nous posons les fondations de tout l'édifice. Chaque porte que vous construirez dans ce chapitre sera utilisée dans les chapitres suivants pour construire des circuits de plus en plus complexes, jusqu'au CPU complet.

### 3.2 Pourquoi le Binaire ?

#### 3.2.1 La question fondamentale

Avant de construire des portes logiques, posons-nous une question essentielle : **pourquoi les ordinateurs utilisent-ils le binaire (0 et 1) plutôt que le système décimal (0-9) que nous utilisons au quotidien ?**

La réponse est d'ordre **physique et pratique** :

1. **Fiabilité** : Distinguer entre deux états (tension haute/basse, courant/pas courant) est beaucoup plus fiable que de distinguer entre dix niveaux différents. Le bruit électrique peut facilement transformer un "7" en "8", mais rarement un "1" en "0".
2. **Simplicité** : Les circuits qui ne gèrent que deux états sont beaucoup plus simples à concevoir et à fabriquer. Un transistor peut facilement agir comme un interrupteur (on/off).

3. **Universalité** : George Boole a prouvé au 19ème siècle que toute la logique peut être exprimée avec seulement deux valeurs : Vrai et Faux.

### 3.2.2 Du voltage au bit

Dans le monde physique, nos ordinateurs utilisent des tensions électriques :

Tension	Signification logique
0V - 0.8V	0 (Faux)
2.4V - 3.3V	1 (Vrai)

La zone entre 0.8V et 2.4V est une “zone interdite” — les circuits sont conçus pour ne jamais s’y trouver de manière stable. C’est cette séparation nette qui rend le binaire si robuste.

### 3.2.3 L’abstraction qui libère

En tant qu’architectes système, nous n’avons pas besoin de penser aux voltages, aux électrons, ou aux transistors. Nous travaillons avec des **abstractions** : - Un **bit** peut valoir 0 ou 1 - Une **fonction booléenne** transforme des bits en d’autres bits

C’est la première de nombreuses couches d’abstraction que nous allons traverser. Chaque couche nous permet de penser à un niveau supérieur sans nous soucier des détails de la couche inférieure.

## 3.3 La Porte NAND : Notre Axiome

### 3.3.1 Pourquoi partir du NAND ?

Il existe de nombreuses portes logiques (AND, OR, NOT, XOR...), mais nous choisissons de tout construire à partir d’une seule : la porte **NAND** (Not-AND).

Pourquoi ce choix ?

1. **Complétude fonctionnelle** : Le NAND est dit “fonctionnellement complet” — on peut construire TOUTES les autres portes logiques à partir de NAND uniquement. C’est mathématiquement prouvé !
2. **Réalité industrielle** : La technologie CMOS (Complementary Metal-Oxide-Semiconductor), utilisée dans tous les processeurs modernes, implémente naturellement les portes NAND de manière très efficace — seulement 4 transistors.
3. **Pédagogie** : Partir d’une seule brique force à comprendre comment les abstractions se construisent les unes sur les autres.

### 3.3.2 Table de Vérité NAND

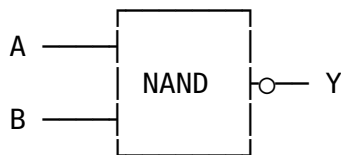
La porte NAND prend deux entrées (A et B) et produit une sortie. Son comportement :

A	B	NAND(A, B)
0	0	<b>1</b>
0	1	<b>1</b>
1	0	<b>1</b>
1	1	<b>0</b>

**Règle simple** : Le résultat est **0** *seulement si* A **et** B sont tous les deux à **1**. Dans tous les autres cas, c'est **1**.

Le nom "NAND" vient de "Not-AND" : c'est l'inverse exact d'une porte AND.

### 3.3.3 Symbole graphique



Le petit cercle (o) indique l'inversion

## 3.4 Construction des Portes Élémentaires

Votre mission dans ce chapitre est de construire les portes suivantes, en utilisant **uniquement** des NAND (et les portes que vous aurez déjà créées).

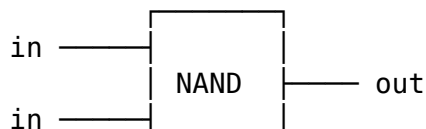
### 3.4.1 A. NOT (Inverseur) — L'inversion de la réalité

**Rôle dans l'ordinateur** : L'inverseur est fondamental. Il permet de créer des conditions négatives ("si PAS égal..."), de complémenter des nombres (pour la soustraction), et d'implémenter des bascules (pour la mémoire).

**Spécification** : | in | out | |---|---| | 0 | 1 | | 1 | 0 |

**Le défi** : Comment utiliser une porte NAND (qui prend 2 entrées) pour n'en traiter qu'une seule ?

**L'astuce** : Connectez le même signal aux deux entrées !



\_\_\_\_\_

Vérifions avec la table de vérité : - Si  $in = 0$  :  $NAND(0, 0) = 1$  - Si  $in = 1$  :  $NAND(1, 1) = 0$

C'est exactement le comportement NOT !

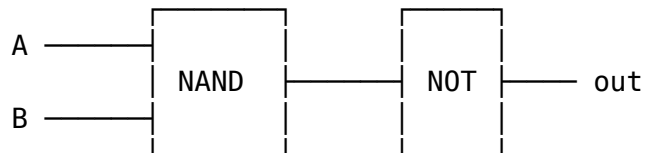
### 3.4.2 B. AND (Et) — La conjonction

**Rôle dans l'ordinateur** : La porte AND est essentielle pour : - Extraire des bits spécifiques d'un nombre (masquage) - Vérifier que TOUTES les conditions sont vraies - Contrôler le passage de données (avec un signal d'activation)

**Spécification** : | A | B | AND(A, B) | |—|—|:—|—| | 0 | 0 | 0 | | 0 | 1 | 0 | | 1 | 0 | 0 | | 1 | 1 | 1 |

**La sortie vaut 1 seulement si A ET B valent 1.**

**L'insight** : Le NAND est un "Not-AND". Si on inverse le résultat d'un NAND... on obtient un AND !



Formule :  $AND(A, B) = NOT(NAND(A, B))$

### 3.4.3 C. OR (Ou) — L'alternative

**Rôle dans l'ordinateur** : La porte OR permet de : - Combiner plusieurs signaux de contrôle - Vérifier qu'AU MOINS UNE condition est vraie - Créer des priorités (interruptions, erreurs)

**Spécification** : | A | B | OR(A, B) | |—|—|:—|—| | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 1 |

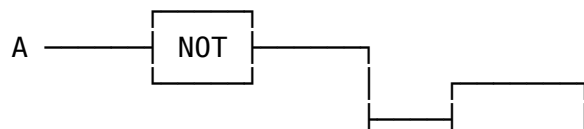
**La sortie vaut 1 si au moins une entrée vaut 1.**

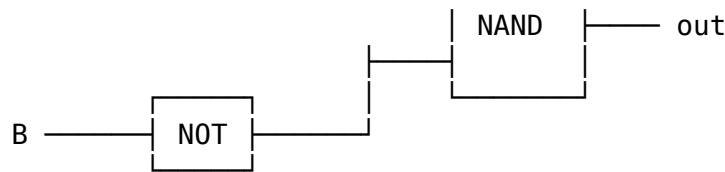
**Le Théorème de De Morgan** : Ce théorème fondamental nous donne la clé :

$$A \text{ OR } B = NOT( (NOT A) \text{ AND } (NOT B) )$$

En utilisant uniquement des NAND :

$$A \text{ OR } B = (NOT A) \text{ NAND } (NOT B)$$





#### 3.4.4 D. XOR (Ou Exclusif) — La différence

**Rôle dans l'ordinateur** : XOR est crucial pour : - **L'addition binaire** : XOR calcule la somme de deux bits (sans la retenue) - **La comparaison** : XOR détecte si deux bits sont différents - **Le cryptage** : XOR est au cœur de nombreux algorithmes de chiffrement - **La détection d'erreurs** : Calcul de parité

**Spécification** : | A | B | XOR(A, B) | | — | — | : — — — | | 0 | 0 | 0 | | 0 | 1 | 1 | | 1 | 0 | 1 | | 1 | 1 | 0 |

**La sortie vaut 1 si les entrées sont différentes.**

**Formule algébrique** :

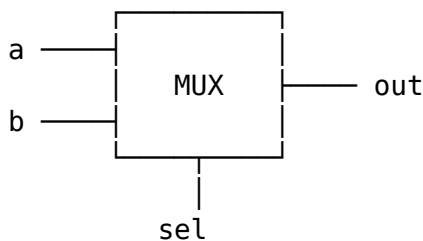
$$\text{XOR}(A, B) = (A \text{ AND NOT } B) \text{ OR } (\text{NOT } A \text{ AND } B)$$

En mots : "A est vrai et B est faux" OU "A est faux et B est vrai".

#### 3.4.5 E. Multiplexeur (Mux) — L'aiguilleur

**Rôle dans l'ordinateur** : Le multiplexeur est l'un des composants les plus importants ! Il permet de : - **Choisir entre plusieurs sources de données** : Comme un aiguillage de train - **Implémenter les conditions** : `if (sel) then b else a` - **Construire la mémoire** : Sélectionner quelle cellule lire

**Spécification** : - Si `sel == 0` alors `out = a` - Si `sel == 1` alors `out = b`



**L'insight** : On peut voir le Mux comme : "soit (a ET non-sel) soit (b ET sel)".

$$\text{out} = (a \text{ AND NOT sel}) \text{ OR } (b \text{ AND sel})$$

**Pourquoi le Mux est-il si important ?**

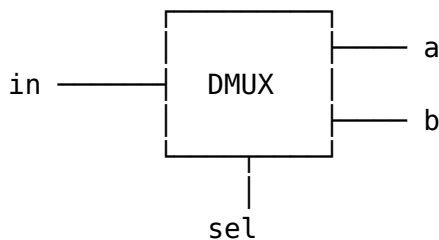
Imaginez que vous construisez un CPU. À chaque cycle, le CPU doit choisir : - D'où vient l'opérande ? De la mémoire ou d'un registre ? - Où va le résultat ? Vers la mémoire ou un registre ? - Quelle instruction exécuter ?

Chacun de ces choix est implémenté par un multiplexeur !

### 3.4.6 F. Démultiplexeur (DMux) — L'inverse de l'aiguilleur

**Rôle dans l'ordinateur** : Le DMux fait l'inverse du Mux — il prend UNE entrée et la dirige vers UNE des sorties possibles. Utile pour : - **L'adressage mémoire** : Activer la bonne cellule - **La distribution de signaux** : Envoyer une commande au bon périphérique

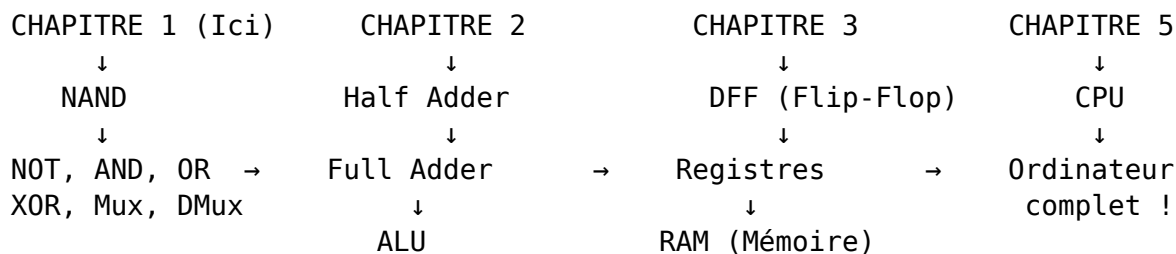
**Spécification** : - Si `sel == 0` alors `a = in, b = 0` - Si `sel == 1` alors `a = 0, b = in`



## 3.5 Le Lien avec l'Ordinateur Complet

Prenons du recul. Pourquoi construisons-nous ces portes ?

### 3.5.1 Du NAND au CPU : La feuille de route



**Chaque porte a un rôle précis** : - **NOT** : Permet la soustraction (via le complément à 2) - **AND** : Masquage de bits, conditions ET - **OR** : Combinaison de signaux, conditions OU - **XOR** : Addition bit à bit, comparaisons - **Mux** : Tous les choix du CPU (quelle donnée ? quelle opération ?) - **DMux** : Adressage mémoire, routage des résultats

Quand vous jouerez à un jeu sur votre ordinateur Codex au Chapitre 9, chaque pixel affiché à l'écran aura été calculé par des millions d'opérations utilisant ces portes élémentaires !

## 3.6 Description Matérielle (Codex HDL)

Pour décrire nos circuits, nous utilisons un langage appelé **HDL** (Hardware Description Language). C'est un langage **déclaratif** : on décrit QUELS composants



existent et COMMENT ils sont connectés, pas DANS QUEL ORDRE les exécuter.

### 3.6.1 Pourquoi un langage de description ?

En électronique réelle, on dessine des schémas. Mais les schémas deviennent illisibles pour des circuits complexes (un CPU contient des millions de portes !). Le HDL permet de : 1. **Décrire** des circuits de manière textuelle 2. **Simuler** leur comportement avant fabrication 3. **Synthétiser** le circuit vers du matériel réel

Notre HDL est inspiré du **VHDL** (Very High-Speed Integrated Circuit HDL), utilisé dans l'industrie.

### 3.6.2 Structure d'un fichier .hdl

Un fichier HDL se compose de deux parties :

```
-- =====
-- 1. L'ENTITÉ : L'interface externe (les "broches" de la puce)
-- =====
entity NomDeLaPuce is
  port(
    a : in bit;      -- Entrée de 1 bit
    b : in bit;      -- Autre entrée
    y : out bit      -- Sortie (pas de point-virgule sur la dernière)
  );
end entity;

-- =====
-- 2. L'ARCHITECTURE : Le contenu interne (le câblage)
-- =====
architecture rtl of NomDeLaPuce is
  -- Déclaration des composants utilisés
  component Nand
    port(a : in bit; b : in bit; y : out bit);
  end component;

  -- Déclaration des signaux internes (fils)
  signal fil_interne : bit;

begin
  -- Instanciation et connexion des composants
  u1: Nand port map (a => a, b => b, y => fil_interne);

  -- Assignment directe (optionnel)
  y <= fil_interne;
end architecture;
```

### 3.6.3 Vocabulaire essentiel

Terme	Signification
entity	L'interface externe — quelles sont les entrées/sorties
architecture	Le contenu — comment c'est câblé à l'intérieur
component	Une puce qu'on veut utiliser (il faut la déclarer)
signal	Un fil interne (pour connecter des composants entre eux)
port map	"Brancher" les fils aux broches d'un composant
=>	"est connecté à" (broche => signal)
<=	Assignation (sortie <= signal)

### 3.6.4 Règles de connexion

1. **À gauche du =>** : Le nom de la broche du composant que vous utilisez
2. **À droite du =>** : Le signal de VOTRE architecture que vous y connectez
3. **Les signaux doivent être déclarés** avant le begin
4. **Chaque instance a un nom unique** ( u1 , u2 , etc.)

### 3.6.5 Bus (Vecteurs de bits)

Pour manipuler plusieurs bits simultanément (ex: un nombre de 32 bits), on utilise `bits(MSB downto LSB)` :

```
-- Un bus de 32 bits
data_bus : in bits(31 downto 0);

-- Accéder à un bit spécifique
data_bus(0)  -- Le bit de poids faible (LSB)
data_bus(31) -- Le bit de poids fort (MSB)

-- Extraire une tranche
data_bus(7 downto 0) -- Les 8 bits de poids faible
```

### 3.6.6 Exemple complet : XOR en HDL

```
-- La porte XOR : sortie = 1 si les entrées sont différentes
-- XOR(a,b) = (a AND NOT b) OR (NOT a AND b)
```

```
entity Xor2 is
  port(
    a : in bit;
    b : in bit;
    y : out bit
  );
end entity;
```

```

architecture rtl of Xor2 is
  -- Composants utilisés
  component Inv
    port(a : in bit; y : out bit);
  end component;
  component And2
    port(a : in bit; b : in bit; y : out bit);
  end component;
  component Or2
    port(a : in bit; b : in bit; y : out bit);
  end component;

  -- Signaux internes
  signal not_a, not_b : bit; -- Les inversions de a et b
  signal w1, w2 : bit;      -- Résultats intermédiaires

begin
  -- NOT a et NOT b
  u_nota: Inv port map (a => a, y => not_a);
  u_notb: Inv port map (a => b, y => not_b);

  -- (a AND NOT b)
  u_and1: And2 port map (a => a, b => not_b, y => w1);

  -- (NOT a AND b)
  u_and2: And2 port map (a => not_a, b => b, y => w2);

  -- Résultat final : OR des deux termes
  u_or: Or2 port map (a => w1, b => w2, y => y);

end architecture;

```

---

## 3.7 Exercices Pratiques

### 3.7.1 Exercices sur le Simulateur Web

Le **Simulateur Web** vous permet de construire et tester vos portes de manière interactive. Lancez-le et allez dans la section **HDL Progression**.

Exercice	Description	Difficulté
Inv	Inverseur (NOT) — Votre première porte	□
And2	Porte AND à 2 entrées	□
Or2	Porte OR à 2 entrées	□
Xor2	Porte XOR (Ou exclusif)	□□
Mux	Multiplexeur 2 vers 1	□□

Exercice	Description	Difficulté
DMux	Démultiplexeur 1 vers 2	□□

**Pour chaque exercice** : 1. Lisez la spécification et la table de vérité 2. Réfléchissez à comment combiner les portes disponibles 3. Écrivez votre code HDL 4. Testez — le simulateur vous montrera un chronogramme des signaux 5. Si un test échoue, analysez quelle entrée produit le mauvais résultat

### 3.7.2 Comment lancer le simulateur web ?

```
cd web
npm install    # (première fois uniquement)
npm run dev
```

Puis ouvrez votre navigateur à l'adresse indiquée (généralement `http://localhost:5173`).

### 3.7.3 Alternative : Tests en ligne de commande

Si vous préférez travailler dans les fichiers du répertoire `hdl_lib/01_gates/` :

```
# Tester une porte spécifique
cargo run -p hdl_cli -- test hdl_lib/01_gates/Not.hdl

# Tester toutes les portes du projet 1
cargo run -p hdl_cli -- test hdl_lib/01_gates/
```

## 3.8 Défis Supplémentaires

### 3.8.1 Défi 1 : Minimiser le nombre de NAND

Construisez la porte XOR en utilisant **seulement 4 portes NAND** (c'est le minimum théorique !). La solution classique en utilise 5.

### 3.8.2 Défi 2 : Implémenter IMPLIES

La fonction "implication" ( $A \rightarrow B$ ) vaut FAUX seulement si A est VRAI et B est FAUX.

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

Construisez cette porte en utilisant les portes élémentaires.

### 3.8.3 Défi 3 : Mux à 4 entrées

Construisez un Mux qui choisit parmi 4 entrées ( `a` , `b` , `c` , `d` ) avec 2 bits de sélection ( `sel[1:0]` ).

## 3.9 Ce qu'il faut retenir

1. **Le binaire simplifie** : Deux états sont plus fiables que dix
2. **NAND est universel** : Toutes les portes peuvent être construites à partir de NAND
3. **L'abstraction est puissante** : On construit des couches les unes sur les autres
4. **Chaque porte a un rôle** :
  - NOT → Inversion, complément
  - AND → Masquage, condition "et"
  - OR → Combinaison, condition "ou"
  - XOR → Addition, comparaison
  - Mux → Choix, sélection
  - DMux → Routage, adressage

**Prochaine étape** : Au Chapitre 2, nous utiliserons ces portes pour construire des circuits qui font de l'**arithmétique** — addition, soustraction, et une ALU (Unité Arithmétique et Logique) complète.

**Conseil** : Ne passez pas au chapitre suivant avant d'avoir réussi tous les exercices de ce chapitre. Chaque porte que vous construisez sera réutilisée dans les chapitres suivants !

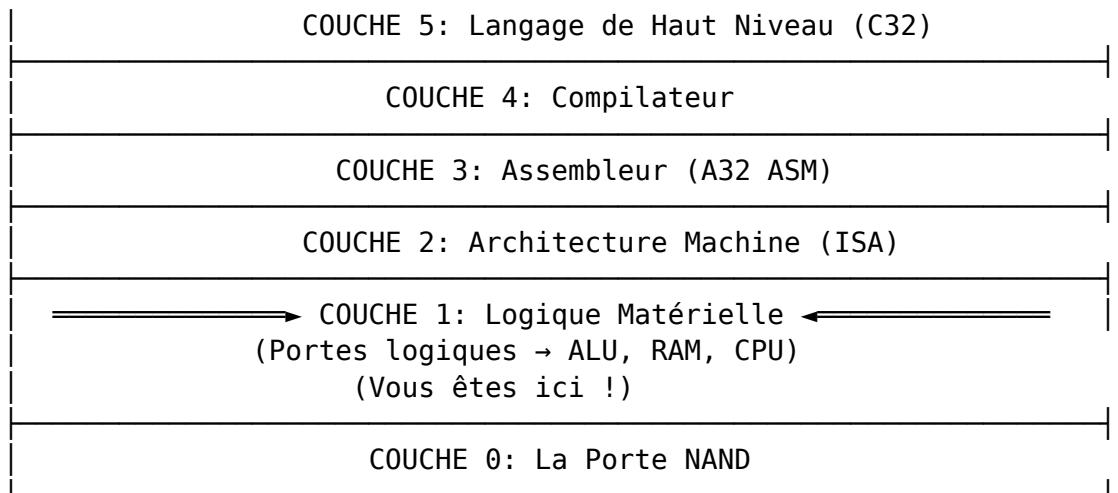
## 4 Arithmétique Binaire

"Les mathématiques sont le langage avec lequel Dieu a écrit l'univers." — Galilée

Dans le chapitre précédent, nous avons appris à manipuler des bits individuels avec des portes logiques. Mais un ordinateur doit savoir compter ! Comment passer de simples portes logiques à une calculatrice capable d'additionner des nombres à 32 bits ?

### 4.1 Où en sommes-nous ?

COUCHE 7: Applications
COUCHE 6: Système d'Exploitation



Nous sommes toujours dans la couche matérielle, mais nous montons d'un niveau. Nous allons combiner les portes logiques du Chapitre 1 pour construire des circuits arithmétiques, culminant avec l'**ALU** (Arithmetic Logic Unit) — le composant qui effectue TOUS les calculs du processeur.

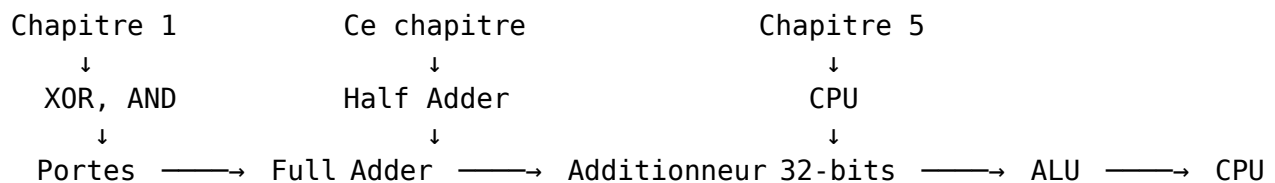
## 4.2 Pourquoi l'Arithmétique est-elle si Importante ?

### 4.2.1 Au cœur de tout calcul

Regardez ce que fait un ordinateur : - **Afficher une image** : Calculer la couleur de chaque pixel (additions, multiplications) - **Jouer un son** : Mélanger des formes d'onde (additions) - **Naviguer sur le web** : Calculer des checksums, décompresser des données - **Exécuter un programme** : Calculer l'adresse de la prochaine instruction (addition)

Même les opérations les plus "abstraites" se réduisent finalement à des opérations arithmétiques sur des nombres binaires. L'ALU que vous allez construire est le moteur qui fait tourner TOUT.

### 4.2.2 Ce que nous allons construire



À la fin de ce chapitre, vous aurez construit une ALU capable d'effectuer : - Addition et soustraction - ET, OU, XOR logiques - Comparaisons (via les drapeaux)

### 4.3 Représentation des Nombres

#### 4.3.1 Le Système Binaire (Base 2)

Avant de construire des additionneurs, comprenons comment les nombres sont représentés.

En décimal (base 10), chaque position représente une puissance de 10 :

$$\begin{array}{ccc} 4 & 2 & 7 \\ \downarrow & \downarrow & \downarrow \\ 10^2 & 10^1 & 10^0 \end{array} \rightarrow 4 \times 100 + 2 \times 10 + 7 \times 1 = 427$$

En binaire (base 2), chaque position représente une puissance de 2 :

Position :	3	2	1	0
Poids :	$2^3$	$2^2$	$2^1$	$2^0$
Valeur :	8	4	2	1

Exemple :  $1011_2 = 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 = 11_{10}$

#### 4.3.2 Taille des nombres dans Codex

Notre ordinateur Codex travaille sur **32 bits**. Cela signifie : - **Plage non-signée** : 0 à  $2^{32} - 1 = 4\,294\,967\,295$  ( $\approx 4$  milliards) - **Plage signée** : -2 147 483 648 à 2 147 483 647 ( $\approx \pm 2$  milliards)

C'est suffisant pour : - Adresser 4 Go de mémoire (chaque octet a une adresse unique)  
- Représenter des coordonnées d'écran, des scores de jeux, des compteurs

#### 4.3.3 Les Nombres Négatifs : Le Complément à 2

Comment représenter des nombres négatifs avec seulement des 0 et des 1 ?

**Le problème** : On pourrait utiliser un bit de signe (0 = positif, 1 = négatif), mais alors on aurait besoin de circuits différents pour l'addition et la soustraction, et on aurait deux représentations du zéro (+0 et -0).

**La solution brillante** : Le **Complément à 2**.

Le bit le plus à gauche (bit 31, le MSB) est le "bit de signe" : - 0 → le nombre est positif ou nul - 1 → le nombre est négatif

Mais attention, ce n'est pas un simple bit de signe ! Le système est conçu pour que **l'addition fonctionne de la même manière** que le nombre soit positif ou négatif.

#### 4.3.4 Comment obtenir le complément à 2 (la valeur négative) ?

Pour obtenir -X à partir de X : 1. **Inverser** tous les bits de X (0→1, 1→0) 2. **Ajouter 1** au résultat

**Exemple sur 4 bits** : Calculons -5

```

5 en binaire : 0101
Inversion   : 1010
Ajouter 1   : + 0001
-----
-5          : 1011

```

**Vérification** : 5 + (-5) devrait donner 0

```

  0101 (5)
+ 1011 (-5)
-----

```

10000 → Les 4 bits de poids faible sont 0000 ✓  
(La retenue "1" est ignorée car on travaille sur 4 bits)

#### 4.3.5 Pourquoi le complément à 2 est-il génial ?

1. **Un seul zéro** : 0000 est le seul zéro (pas de +0 et -0)
2. **L'addition fonctionne universellement** : Le même circuit additionne les positifs et les négatifs
3. **La soustraction devient une addition** :  $A - B = A + (-B) = A + \text{NOT}(B) + 1$

C'est grâce au complément à 2 que notre ALU peut être relativement simple !

## 4.4 L'Addition Binaire

L'addition binaire suit les mêmes règles que l'addition décimale qu'on apprend à l'école : on additionne colonne par colonne, de droite à gauche, en propageant les retenues.

### 4.4.1 Les règles de base (sur 1 bit)

```

0 + 0 = 0 (pas de retenue)
0 + 1 = 1 (pas de retenue)
1 + 0 = 1 (pas de retenue)
1 + 1 = 10 (c'est-à-dire 0 avec une retenue de 1)

```

### 4.4.2 Exemple d'addition sur 4 bits

Calculons  $5 + 3 = 8$  :

```

Retenues : 1 1 1
           -----
  5       : 0 1 0 1
+ 3       : + 0 0 1 1
-----
  8       : 1 0 0 0

```



Détail colonne par colonne (de droite à gauche) : - Colonne 0 :  $1 + 1 = 0$ , retenue 1 - Colonne 1 :  $0 + 1 + 1(\text{retenue}) = 0$ , retenue 1 - Colonne 2 :  $1 + 0 + 1(\text{retenue}) = 0$ , retenue 1 - Colonne 3 :  $0 + 0 + 1(\text{retenue}) = 1$

Résultat :  $1000_2 = 8_{10}$  □

## 4.5 Le Demi-Additionneur (Half Adder)

Le demi-additionneur est le circuit le plus simple pour additionner deux bits. Il produit : - **sum** : La somme (bit de poids faible) - **carry** : La retenue (bit de poids fort)

### 4.5.1 Table de vérité

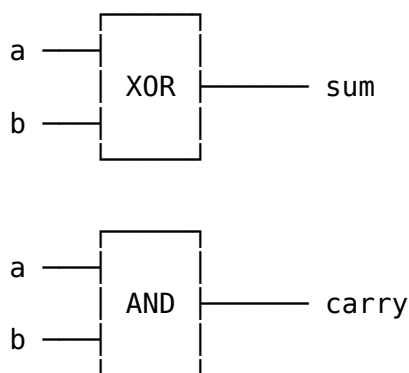
a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

### 4.5.2 L'insight clé

Regardez attentivement les colonnes : - **sum** correspond exactement à **XOR(a, b)** — différent = 1, identique = 0 - **carry** correspond exactement à **AND(a, b)** — les deux à 1 = retenue

C'est pour cela que nous avons construit XOR et AND au Chapitre 1 !

### 4.5.3 Schéma du circuit



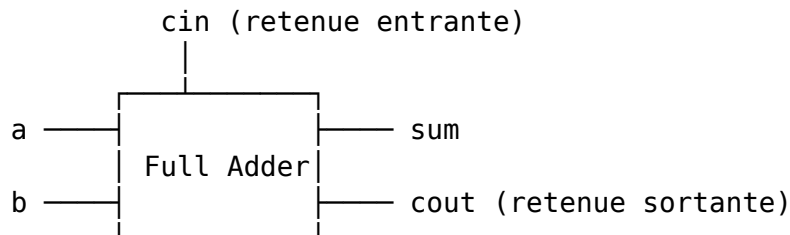
### 4.5.4 Limitation

Le demi-additionneur ne peut pas recevoir de retenue d'une colonne précédente. Il ne fonctionne donc que pour le bit de poids faible (la première colonne).

## 4.6 L'Additionneur Complet (Full Adder)

Pour additionner des nombres de plusieurs bits, chaque colonne (sauf la première) doit pouvoir accepter une retenue venant de la colonne précédente.

### 4.6.1 Interface



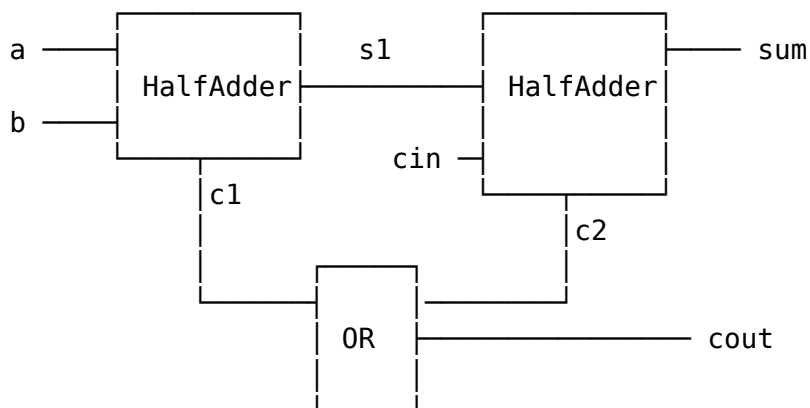
### 4.6.2 Table de vérité

a	b	cin	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

### 4.6.3 Comment le construire ?

Un Full Adder peut être construit avec **deux Half Adders et une porte OR** :

1. Le premier Half Adder additionne **a** et **b**
2. Le second Half Adder additionne le résultat avec **cin**
3. Si l'un des deux Half Adders produit une retenue, on a une retenue finale



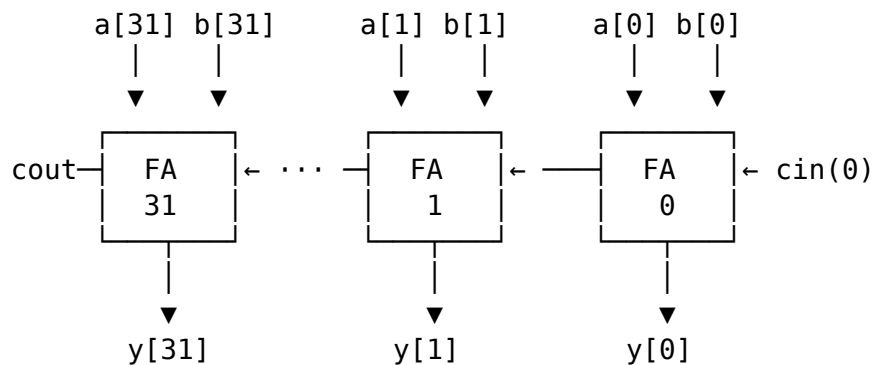
Formules : -  $s1 = \text{XOR}(a, b)$  -  $\text{sum} = \text{XOR}(s1, \text{cin})$  -  $c1 = \text{AND}(a, b)$  -  $c2 = \text{AND}(s1, \text{cin})$   
 -  $\text{cout} = \text{OR}(c1, c2)$

---

## 4.7 L'Additionneur 32-bits (Ripple Carry Adder)

Pour additionner des nombres de 32 bits, nous connectons 32 Full Adders en cascade. La retenue de sortie de chaque additionneur devient la retenue d'entrée du suivant.

### 4.7.1 Schéma simplifié



**Note :** Le premier Full Adder (position 0) a une retenue d'entrée de 0 pour une addition normale. Mais on peut y injecter un 1 pour implémenter la soustraction ( $A + \text{NOT}(B) + 1$ ).

### 4.7.2 Le compromis du Ripple Carry

**Avantage :** Très simple à comprendre et à implémenter.

**Inconvénient :** Les retenues se propagent d'un bout à l'autre. Pour 32 bits, la retenue doit traverser 32 étages. C'est lent !

Dans les vrais processeurs, on utilise des techniques comme le "Carry Lookahead Adder" pour accélérer la propagation. Mais pour notre projet pédagogique, le Ripple Carry est parfait.

---

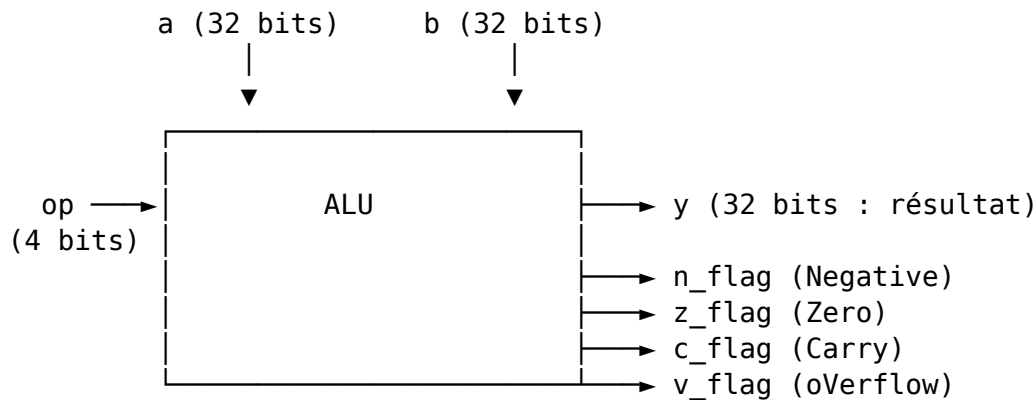
## 4.8 L'ALU (Arithmetic Logic Unit)

L'ALU est le **cœur calculatoire** du processeur. C'est elle qui effectue TOUTES les opérations arithmétiques et logiques.

### 4.8.1 Pourquoi combiner arithmétique et logique ?

Plutôt que d'avoir des circuits séparés pour l'addition, la soustraction, le AND, le OR, etc., l'ALU combine tout en un seul composant. Un signal de contrôle ( $op$ ) lui dit quelle opération effectuer.

### 4.8.2 Interface de l'ALU Codex



### 4.8.3 Les Opérations de l'ALU

Le signal **op** (4 bits) définit l'opération à effectuer :

op (binaire)	Nom	Opération	Description
0000	AND	$a \& b$	ET logique bit à bit
0001	EOR	$a \wedge b$	OU exclusif bit à bit
0010	SUB	$a - b$	Soustraction
0011	ADD	$a + b$	Addition
0100	ORR	$a \vee b$	OU logique bit à bit
0101	MOV	$b$	Copie de b (ignore a)
0110	MVN	$\sim b$	Inversion de b (NOT)

### 4.8.4 Comment implémenter la soustraction ?

Grâce au complément à 2, la soustraction devient une addition :

$$A - B = A + (-B) = A + (\text{NOT } B) + 1$$

En pratique : 1. Inverser tous les bits de B (avec des portes NOT) 2. Additionner A et NOT(B) avec une retenue d'entrée de 1

C'est pour cela que notre additionneur a une entrée **cin** !

### 4.8.5 Les Drapeaux (Flags)

Les drapeaux sont des informations supplémentaires sur le résultat :

Drapeau	Nom	Signification
<b>N</b>	Negative	1 si le résultat est négatif (bit 31 = 1)
<b>Z</b>	Zero	1 si le résultat est exactement 0
<b>C</b>	Carry	1 s'il y a eu une retenue (dépassement non-signé)
<b>V</b>	Overflow	1 s'il y a eu un dépassement signé

### À quoi servent ces drapeaux ?

Ils permettent au CPU de prendre des décisions : - **BEQ** (Branch if Equal) teste si  $Z = 1$  - **BLT** (Branch if Less Than) teste une combinaison de  $N$  et  $V$  - **BCS** (Branch if Carry Set) teste si  $C = 1$

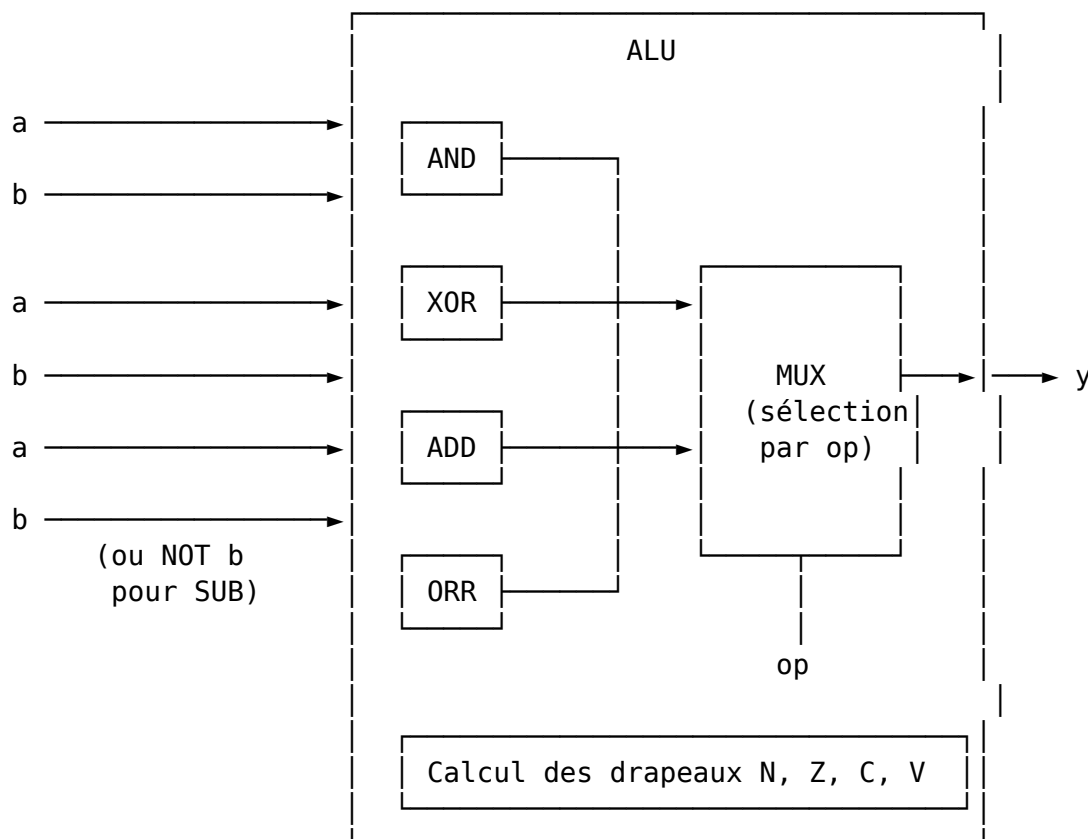
Sans les drapeaux, il serait impossible d'implémenter les conditions **if**, les boucles **while**, etc. !

#### 4.8.6 Comment calculer les drapeaux ?

- **N (Negative)** : C'est simplement le bit 31 du résultat
- **Z (Zero)** : NOR de tous les bits du résultat (tous à 0 ?)
- **C (Carry)** : La retenue de sortie de l'additionneur
- **V (Overflow)** : Se produit quand :
  - Deux positifs donnent un négatif
  - Deux négatifs donnent un positif
  - Formule :  $V = (a[31] == b[31]) \text{ AND } (a[31] != y[31])$  (pour l'addition)

### 4.9 Architecture de l'ALU

Voici comment l'ALU est structurée en interne :



L'idée clé : calculer TOUS les résultats possibles en parallèle, puis utiliser un multiplexeur pour sélectionner le bon selon `op`.

## 4.10 Exercices Pratiques

### 4.10.1 Exercices sur le Simulateur Web

Lancez le **Simulateur Web** et allez dans **HDL Progression** → **Projet 3 : Arithmétique**.

Exercice	Description	Difficulté
HalfAdder	Demi-additionneur (XOR + AND)	□
FullAdder	Additionneur complet (2 Half Adders + OR)	□□
Add16	Additionneur 16 bits en cascade	□□
Inc16	Incrémenteur (+1) — cas spécial utile	□
Sub16	Soustracteur (via complément à 2)	□□
ALU	L'ALU complète avec drapeaux	□□□

### 4.10.2 Conseils pour l'ALU

1. **Commencez par les opérations simples** : AND, OR, XOR sont juste des portes appliquées bit à bit
2. **Pour la soustraction** :  

$$\text{sub} = a + (\text{NOT } b) + 1$$
 Utilisez un inverseur sur b et une retenue d'entrée de 1
3. **Utilisez les Mux** : Un Mux4Way ou Mux8Way sélectionne parmi plusieurs résultats
4. **Les drapeaux** :
  - N : bit 31 du résultat
  - Z : tous les bits sont à 0 ? (utilisez un grand OR puis NOT)
  - C : retenue de sortie de l'additionneur
  - V : comparez les signes des entrées et du résultat

### 4.10.3 Tests en ligne de commande

# Tester le Half Adder

```
cargo run -p hdl_cli -- test hdl_lib/03_arith/HalfAdder.hdl
```

# Tester l'ALU complète

```
cargo run -p hdl_cli -- test hdl_lib/03_arith/ALU.hdl
```

## 4.11 Défis Supplémentaires

### 4.11.1 Défi 1 : Carry Lookahead

Le Ripple Carry est lent car les retenues se propagent séquentiellement. Implémentez un "Carry Lookahead Adder" sur 4 bits qui calcule les retenues en parallèle.

### 4.11.2 Défi 2 : Multiplicateur

Construisez un circuit qui multiplie deux nombres de 4 bits. Indice : la multiplication est une série d'additions décalées.

### 4.11.3 Défi 3 : Comparateur

Construisez un circuit qui compare deux nombres 32 bits et produit trois sorties : -  
lt :  $a < b$  - eq :  $a == b$  - gt :  $a > b$

Indice : Vous pouvez utiliser la soustraction et regarder les drapeaux !

## 4.12 Le Lien avec le CPU

L'ALU que vous venez de construire sera utilisée à CHAQUE cycle d'horloge du CPU :

Instruction	Utilisation de l'ALU
ADD R1, R2, R3	Additionne R2 et R3, stocke dans R1
SUB R1, R2, R3	Soustrait R3 de R2
CMP R1, R2	Soustrait et met à jour les drapeaux (sans stocker)
AND R1, R2, R3	ET logique
LDR R1, [R2]	Calcule l'adresse mémoire ( $R2 + \text{offset}$ )
B label	Calcule la nouvelle adresse ( $PC + \text{offset}$ )

Même les sauts conditionnels (BEQ, BNE, etc.) dépendent des drapeaux produits par l'ALU !

## 4.13 Ce qu'il faut retenir

1. **Le binaire est naturel pour les circuits** : Addition = XOR pour le bit, AND pour la retenue
2. **Le complément à 2 est magique** : Un seul additionneur pour addition ET soustraction
3. **L'ALU est le cœur du calcul** : Toutes les opérations passent par elle
4. **Les drapeaux permettent les décisions** : Sans eux, pas de if, pas de boucles

## 5. La hiérarchie continue :

- Portes → Half Adder → Full Adder → Additionneur 32-bits → ALU

**Prochaine étape** : Au Chapitre 3, nous aborderons la **mémoire**. Comment l'ordinateur peut-il "se souvenir" de données ? Nous construirons des flip-flops, des registres, et de la RAM.

**Conseil** : L'ALU est l'un des composants les plus complexes du projet. Prenez le temps de bien comprendre chaque étape. Si vous avez réussi l'ALU, le reste du projet sera beaucoup plus accessible !

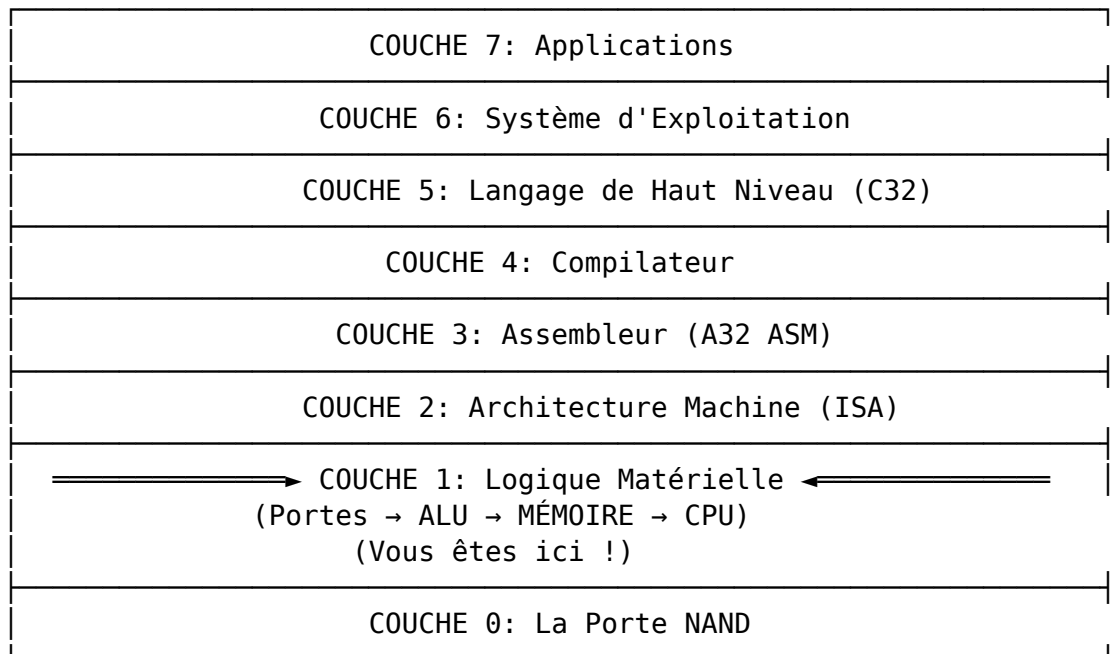
## 5 Logique Séquentielle et Mémoire

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

Jusqu'à présent, nos circuits étaient **combinatoires** : la sortie dépendait uniquement des entrées instantanées, comme une fonction mathématique pure  $y = f(x)$ . Si vous coupez le courant, ils "oublient" tout.

Pour construire un véritable ordinateur, nous devons pouvoir **stocker de l'information** et la récupérer plus tard. C'est le rôle de la **logique séquentielle**.

### 5.1 Où en sommes-nous ?





Nous continuons à construire la couche matérielle. Après les portes logiques (Chapitre 1) et l'ALU (Chapitre 2), nous abordons maintenant la **mémoire** — le composant qui permet à l'ordinateur de “se souvenir”.

## 5.2 Pourquoi la Mémoire est-elle Fondamentale ?

### 5.2.1 Le problème de l'état

Imaginez un programme simple :

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

Pour exécuter cette instruction, l'ordinateur doit : 1. **Lire** la valeur actuelle de `x` 2. **Calculer** `x + 1` (avec l'ALU) 3. **Écrire** le résultat dans `x`

Sans mémoire, il n'y a pas de “valeur actuelle de `x`” à lire. Sans mémoire, le résultat du calcul disparaît immédiatement après avoir été produit.

### 5.2.2 Ce que stocke la mémoire

Un ordinateur en fonctionnement stocke : - **Le programme** : Les instructions à exécuter (le code machine) - **Les données** : Les variables, les tableaux, les objets - **L'état du CPU** : Les registres, le compteur de programme - **La pile d'appels** : Pour les fonctions et les retours

Toutes ces informations vivent dans différentes formes de mémoire.

### 5.2.3 Combinatoire vs Séquentiel

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

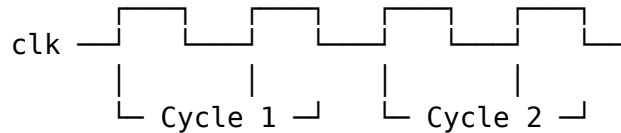
## 5.3 Le Temps et l'Horloge (Clock)

### 5.3.1 Le problème de la synchronisation

Dans un circuit combinatoire, les signaux se propagent à travers les portes avec un léger délai. Si on essaie de lire un résultat avant qu'il soit stable, on obtient des valeurs incorrectes.

**La solution** : L'horloge (`clk`).

L'horloge est un signal qui oscille perpétuellement entre 0 et 1 à une fréquence fixe :



### 5.3.2 Front montant (Rising Edge)

Le moment crucial est le **front montant** : le passage de 0 à 1.

Dans le système Codex, les changements d'état se produisent sur le front montant. Cela signifie : - Pendant que l'horloge est à 0, les circuits combinatoires calculent - Quand l'horloge passe à 1, les résultats sont capturés dans les registres

C'est comme dire : "Tout le monde calcule... et maintenant, on fige les résultats !"

### 5.3.3 Fréquence d'horloge

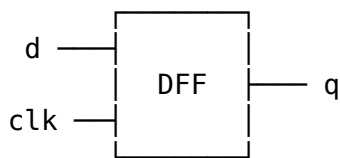
La fréquence de l'horloge détermine la vitesse du processeur : - Un processeur à 1 GHz = 1 milliard de cycles par seconde - À chaque cycle, le CPU peut exécuter (une partie d')une instruction

Plus vite bat l'horloge, plus l'ordinateur est rapide — mais aussi plus il chauffe !

## 5.4 La Bascule D (D Flip-Flop / DFF)

La **DFF** (Data Flip-Flop) est l'atome de la mémoire. C'est le plus petit circuit capable de mémoriser un bit.

### 5.4.1 Interface

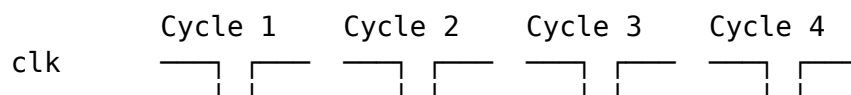


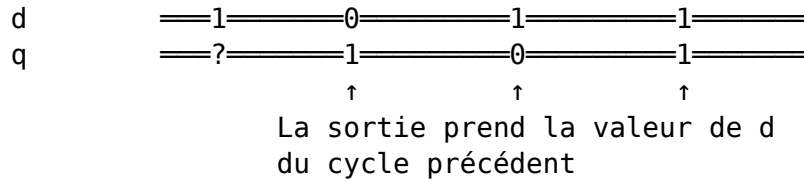
- **d** : La donnée à mémoriser (entrée)
- **q** : La donnée mémorisée (sortie)
- **clk** : L'horloge (parfois implicite)

### 5.4.2 Comportement

**Règle fondamentale** :  $q(t) = d(t-1)$

La sortie à l'instant  $t$  est égale à ce qu'était l'entrée au cycle précédent.





### 5.4.3 Pourquoi est-ce utile ?

La DFF introduit un **délai d'un cycle**. Ce délai permet : 1. De stocker une valeur pour le prochain cycle 2. De casser les boucles (éviter les oscillations infinies) 3. De synchroniser tous les composants sur l'horloge

### 5.4.4 Comment fonctionne une DFF en interne ?

Une DFF peut être construite à partir de deux verrous (latches) en série, eux-mêmes construits à partir de portes NAND. C'est un sujet fascinant mais hors de notre scope — nous considérons la DFF comme une primitive fournie par le simulateur.

## 5.5 Le Registre 1-bit (Bit)

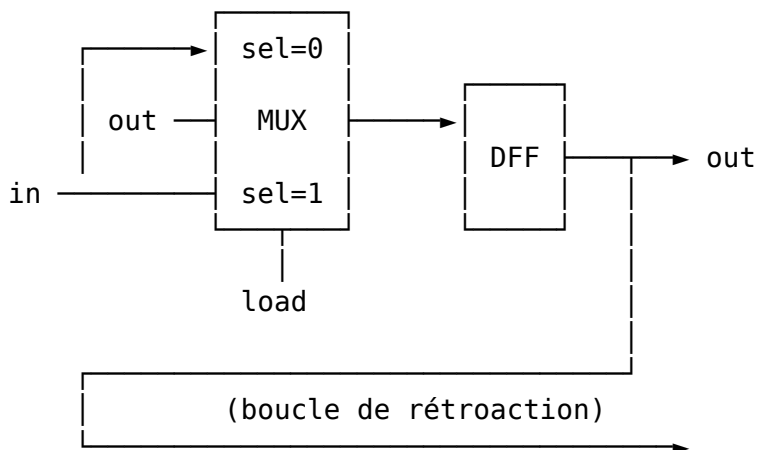
La DFF mémorise pendant UN cycle, puis elle prend la nouvelle valeur d'entrée. Comment garder une valeur **indéfiniment** ?

### 5.5.1 Le problème

On veut un circuit qui : - Si `load = 1` : stocke la nouvelle valeur `in` - Si `load = 0` : conserve l'ancienne valeur

### 5.5.2 La solution : la rétroaction

On utilise un **Mux** pour choisir entre : - L'ancienne valeur (sortie de la DFF) - La nouvelle valeur (`in`)



**Fonctionnement** : - Si `load = 0` : Le Mux sélectionne la sortie de la DFF. La DFF ré-enregistre sa propre valeur. La valeur est **maintenue**. - Si `load = 1` : Le Mux sélectionne `in`. La DFF enregistre la nouvelle valeur.

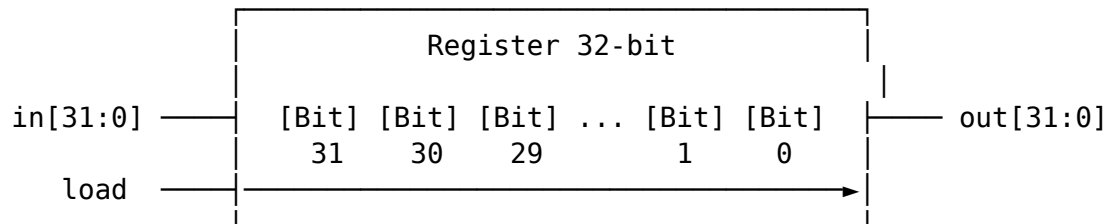
### 5.5.3 C'est magique !

Cette petite boucle de rétroaction transforme un délai d'un cycle en une mémoire permanente. C'est le principe fondamental de toute mémoire électronique.

## 5.6 Le Registre 32-bits

### 5.6.1 Du bit au mot

Un registre 32-bits est simplement **32 registres 1-bit en parallèle**, partageant le même signal `load`.



Quand `load = 1`, les 32 bits sont capturés simultanément. C'est atomique.

### 5.6.2 Le rôle des registres dans le CPU

Le CPU Codex dispose de **16 registres** nommés R0 à R15 :

Registre	Rôle typique
R0-R12	Registres généraux (calculs, variables)
R13 (SP)	Stack Pointer (pointeur de pile)
R14 (LR)	Link Register (adresse de retour)
R15 (PC)	Program Counter (adresse de l'instruction courante)

Les registres sont la mémoire la plus rapide du CPU — ils sont directement connectés à l'ALU et peuvent être lus/écrits en un seul cycle.

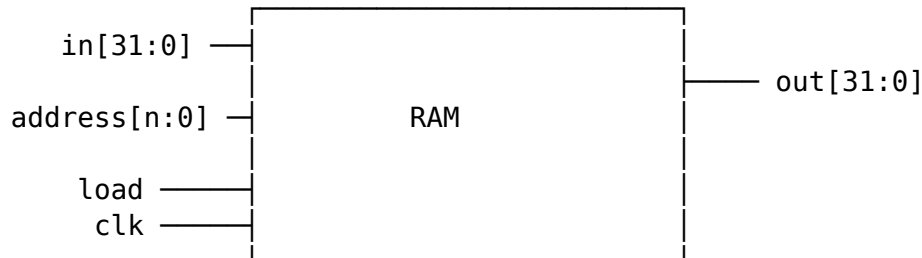
## 5.7 La RAM (Random Access Memory)

### 5.7.1 Du registre à la mémoire

Un registre stocke UN mot de 32 bits. Pour stocker des millions de mots, nous construisons une **RAM** (Random Access Memory).

“Random Access” signifie qu’on peut accéder à n’importe quelle cellule directement, sans parcourir les autres. Contrairement à une bande magnétique où il faut rembobiner !

### 5.7.2 Interface de la RAM



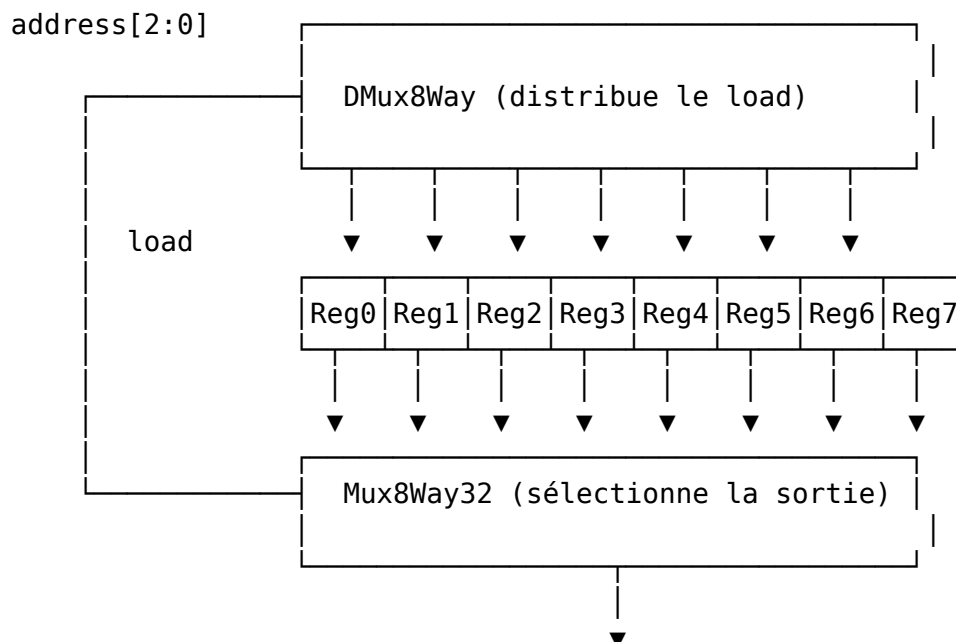
- **in** : La donnée à écrire
- **address** : L’adresse de la cellule à lire/écrire
- **load** : Si 1, écrire **in** à **address**. Si 0, ne rien écrire.
- **out** : La valeur stockée à **address** (toujours disponible en lecture)

### 5.7.3 Comment ça marche ?

La RAM utilise les composants que nous avons construits :

1. **DMux** (Démultiplexeur) : Route le signal **load** vers UN SEUL registre — celui correspondant à l’adresse
2. **Registres** : Stockent les données
3. **Mux** (Multiplexeur) : Sélectionne la sortie du registre correspondant à l’adresse

### 5.7.4 Exemple : RAM8 (8 mots de 32 bits)



```
out[31:0]
```

## Technique de grandes RA

64 (64 mots) à partir d

3 bits supérieurs) +

**L O DING**      6/7

```
e poids fort ( address[
```

ress[2:0] ) choisissent

met de construire de

### Programme (PC)

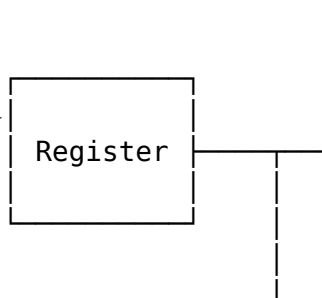
est peut-être le registr

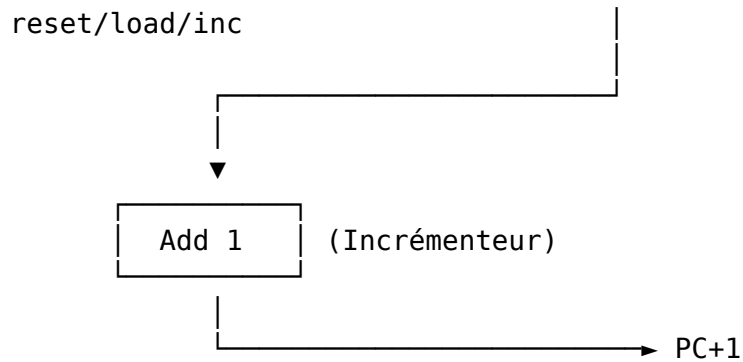
# 1 ?

C doit passer à l'inst

adresse (boucles, con

Condition	Reset
reset = 1	0
load = 1	0
inc = 1	0
sinon	0



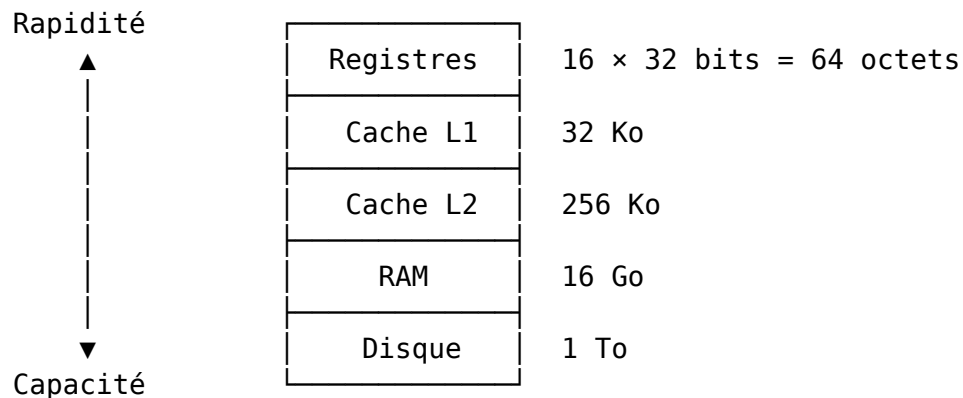


#### 5.8.4 Le lien avec l'exécution du programme

À chaque cycle d'horloge : 1. Le CPU lit l'instruction à l'adresse PC 2. Il décode et exécute l'instruction 3. Il met à jour le PC (incrément ou saut) 4. Le cycle recommence C'est le cœur battant de l'ordinateur !

## 5.9 Les Différents Types de Mémoire

Dans un vrai ordinateur, il y a plusieurs niveaux de mémoire :



Plus on monte dans la pyramide : - Plus c'est rapide - Plus c'est cher par octet - Plus la capacité est faible

Dans le Codex, nous implémentons les registres et la RAM. Les caches et les disques sont des concepts plus avancés.

## 5.10 Exercices Pratiques

### 5.10.1 Exercices sur le Simulateur Web

Lancez le **Simulateur Web** et allez dans **HDL Progression** → **Projet 4 : Séquentiel**.

Exercice	Description	Difficulté
DFF1	Bascule D (fournie comme primitive)	—
BitReg	Registre 1-bit (Mux + DFF)	□
Register16	Registre 16-bits (16 BitReg en parallèle)	□
PC	Compteur de programme avec reset/load/inc	□□
RAM8	RAM de 8 mots (DMux + 8 Registres + Mux)	□□
RAM64	RAM de 64 mots (8 RAM8)	□□
RegFile	Banc de registres (lecture double, écriture simple)	□□□

### 5.10.2 Ordre de progression recommandé

1. **BitReg** : La brique de base. Un Mux et une DFF.
2. **Register16** : 16 BitReg en parallèle. Vérifiez que tous les bits sont synchronisés.
3. **PC** : Attention à la priorité ! reset > load > inc > hold
4. **RAM8** : Utilisez DMux8Way et Mux8Way16
5. **RAM64** : Composition de 8 RAM8

### 5.10.3 Prérequis

Avant de construire les RAMs, assurez-vous d'avoir terminé les composants multi-bits du Projet 2 : - **Mux8Way16** : Sélectionne parmi 8 entrées de 16 bits - **DMux8Way** : Distribue 1 entrée vers 8 sorties

### 5.10.4 Tests en ligne de commande

```
# Tester le registre 1-bit
```

```
cargo run -p hdl_cli -- test hdl_lib/04_seq/BitReg.hdl
```

```
# Tester la RAM8
```

```
cargo run -p hdl_cli -- test hdl_lib/04_seq/RAM8.hdl
```

```
# Tester le PC
```

```
cargo run -p hdl_cli -- test hdl_lib/04_seq/PC.hdl
```

## 5.11 Défis Supplémentaires

### 5.11.1 Défi 1 : RAM avec deux ports de lecture

Construisez une RAM qui permet de lire DEUX adresses différentes simultanément (utile pour le CPU qui doit lire deux opérandes).



### 5.11.2 Défi 2 : Compteur avec valeur maximale

Modifiez le PC pour qu'il s'arrête à une valeur maximale au lieu de continuer à compter (overflow protection).

### 5.11.3 Défi 3 : Registre à décalage (Shift Register)

Construisez un registre où les bits se décalent d'une position à chaque cycle. Utilisé pour : - Les communications série - La multiplication/division par 2 - Les générateurs de nombres aléatoires

## 5.12 Le Lien avec la Suite

La mémoire que vous venez de construire sera utilisée partout dans l'ordinateur :

Composant	Utilisation de la mémoire
<b>Registres R0-R15</b>	16 registres 32-bits pour les calculs
<b>RAM</b>	Stockage du programme et des données
<b>PC</b>	Adresse de l'instruction courante
<b>Pile (Stack)</b>	Zone de RAM pour les appels de fonction
<b>Écran</b>	Zone de RAM mappée aux pixels (MMIO)
<b>Clavier</b>	Registre mappé en mémoire (MMIO)

Au Chapitre 4, nous verrons comment le CPU accède à la mémoire et aux périphériques via le **Memory-Mapped I/O** (MMIO).

## 5.13 Ce qu'il faut retenir

1. **L'horloge synchronise tout** : Les changements se font sur le front montant
2. **La DFF est l'atome de mémoire** :  $q(t) = d(t-1)$
3. **La rétroaction crée la persistance** : Mux + DFF = mémoire permanente
4. **La RAM est un tableau de registres** : Accès par adresse
5. **Le PC guide l'exécution** : Il pointe vers l'instruction courante
6. **Hiérarchie de mémoire** :
  - Registres → Cache → RAM → Disque
  - Rapidité vs Capacité

**Prochaine étape** : Au Chapitre 4, nous définissons l'**Architecture Machine** (ISA). C'est le "contrat" entre le matériel et le logiciel : quelles instructions le CPU comprend-il ? Comment accède-t-il à la mémoire ?

**Conseil** : Prenez le temps de bien comprendre la boucle de rétroaction du registre 1-bit. C'est un concept fondamental qui revient constamment en informatique !

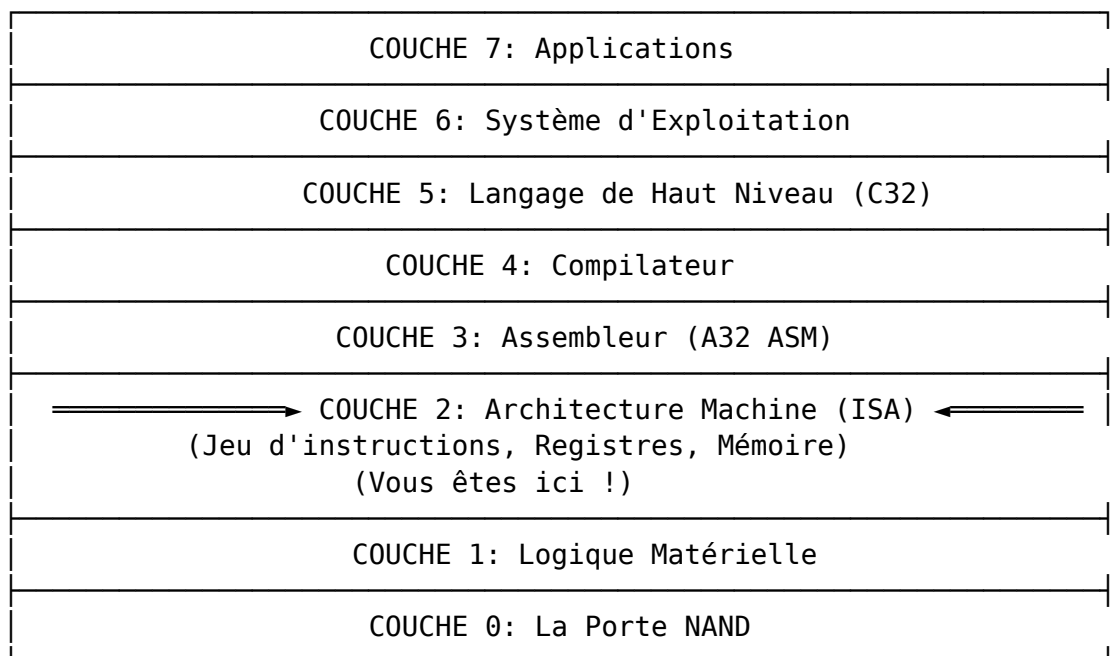
## 6 Architecture Machine (ISA A32)

“Le langage est la limite de mon monde.” — Wittgenstein

Nous avons maintenant des portes logiques, des additionneurs, et de la mémoire. Mais comment **commander** tout cela ? C'est le rôle de l'**Architecture de Jeu d'Instructions** (ISA - Instruction Set Architecture).

L'ISA est le **contrat** entre le matériel et le logiciel. C'est la liste de toutes les instructions que le CPU sait exécuter.

### 6.1 Où en sommes-nous ?



Ce chapitre marque une transition importante. Nous quittons temporairement le monde du matériel pour définir **l'interface** entre matériel et logiciel. L'ISA que nous définissons ici sera : - **Implémentée** par le CPU au Chapitre 5 - **Utilisée** par l'assembleur au Chapitre 6 - **Ciblée** par le compilateur au Chapitre 7

## 6.2 Qu'est-ce qu'une Architecture ?

### 6.2.1 Le contrat fondamental

L'architecture d'un processeur définit : 1. **Les registres** : Combien ? Quelle taille ? Quel rôle ? 2. **Les instructions** : Quelles opérations le CPU peut-il faire ? 3. **L'encodage** : Comment les instructions sont-elles représentées en binaire ? 4. **Le modèle mémoire** : Comment le CPU voit-il la mémoire ?

C'est un **contrat** : - Le matériel **promet** d'exécuter les instructions comme spécifié - Le logiciel **s'engage** à n'utiliser que les instructions définies

### 6.2.2 Codex A32 : Une architecture RISC moderne

L'architecture **Codex A32** est inspirée de ARM, l'architecture qui équipe la plupart des smartphones et le Raspberry Pi. Elle est : - **RISC** (Reduced Instruction Set Computer) : Instructions simples et rapides - **32 bits** : Registres et adresses sur 32 bits - **Load/Store** : Le CPU ne calcule jamais directement en mémoire

## 6.3 Pourquoi RISC ? L'architecture Load/Store

### 6.3.1 CISC vs RISC

CISC (x86, Intel)	RISC (ARM, Codex)
Instructions complexes	Instructions simples
ADD [mem], reg possible	Calcul uniquement entre registres
Vitesse variable par instruction	1 instruction $\approx$ 1 cycle
Plus facile à programmer directement	Plus facile à implémenter en matériel

### 6.3.2 La règle d'or Load/Store

En architecture RISC, le CPU ne peut **jamais** calculer directement sur la mémoire. Toute opération suit le schéma :

1. **LOAD** : Charger les données de la mémoire vers les registres
2. **COMPUTE** : Effectuer le calcul dans les registres
3. **STORE** : Stocker le résultat de retour en mémoire

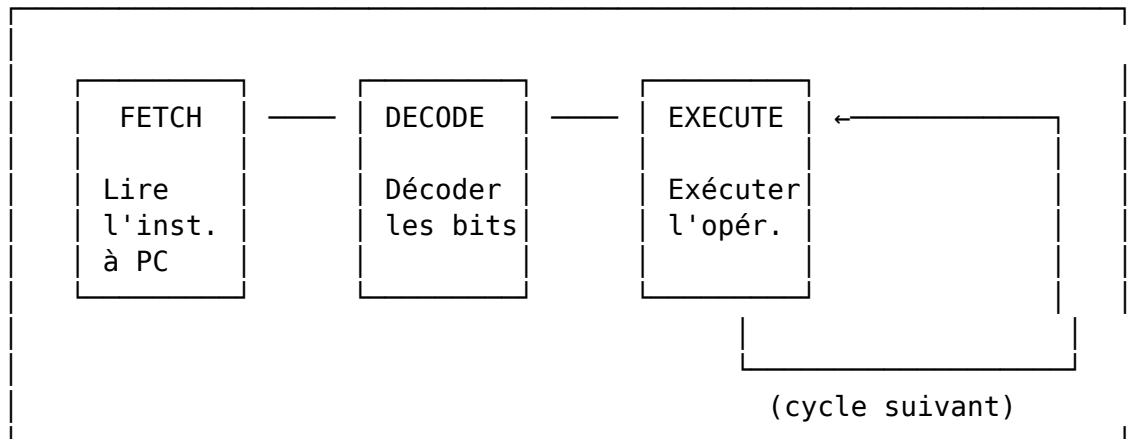
**Exemple** : Incrémenter une variable en mémoire

```
LDR R0, [R1]      ; 1. Charger la valeur depuis l'adresse R1
ADD R0, R0, #1     ; 2. Ajouter 1
STR R0, [R1]      ; 3. Stocker le résultat
```

En x86 (CISC), on pourrait écrire `ADD [mem], 1` en une seule instruction. Mais le CPU RISC est plus simple à construire et peut aller plus vite.

## 6.4 Le Cycle de Vie d'une Instruction

Chaque instruction traverse trois phases :



### 1. **Fetch** (Récupération)

- Le CPU lit l'instruction à l'adresse contenue dans PC
- PC est incrémenté pour pointer vers l'instruction suivante

### 2. **Decode** (Décodage)

- Les 32 bits de l'instruction sont analysés
- Le CPU identifie : quel opération ? quels registres ? quelle valeur immédiate ?

### 3. **Execute** (Exécution)

- L'ALU effectue le calcul
- Le résultat est stocké dans le registre de destination
- Si c'est un branchement, PC peut être modifié

## 6.5 Les Registres : Le Plan de Travail

Le CPU dispose de **16 registres** de 32 bits, nommés R0 à R15.

### 6.5.1 Vue d'ensemble

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

### 6.5.2 Rôles des registres

Registre	Alias	Rôle conventionnel
<b>R0-R3</b>	—	Arguments des fonctions et valeurs de retour
<b>R4-R11</b>	—	Variables locales (préservées par les fonctions)
<b>R12</b>	IP	Registre temporaire (Intra-Procedure call)
<b>R13</b>	<b>SP</b>	Stack Pointer — Pointe vers le sommet de la pile
<b>R14</b>	<b>LR</b>	Link Register — Adresse de retour après BL
<b>R15</b>	<b>PC</b>	Program Counter — Adresse de l'instruction courante

### 6.5.3 Le cas spécial de R15 (PC)

Le PC est accessible comme n'importe quel registre. Si vous écrivez dedans, vous forcez un saut !

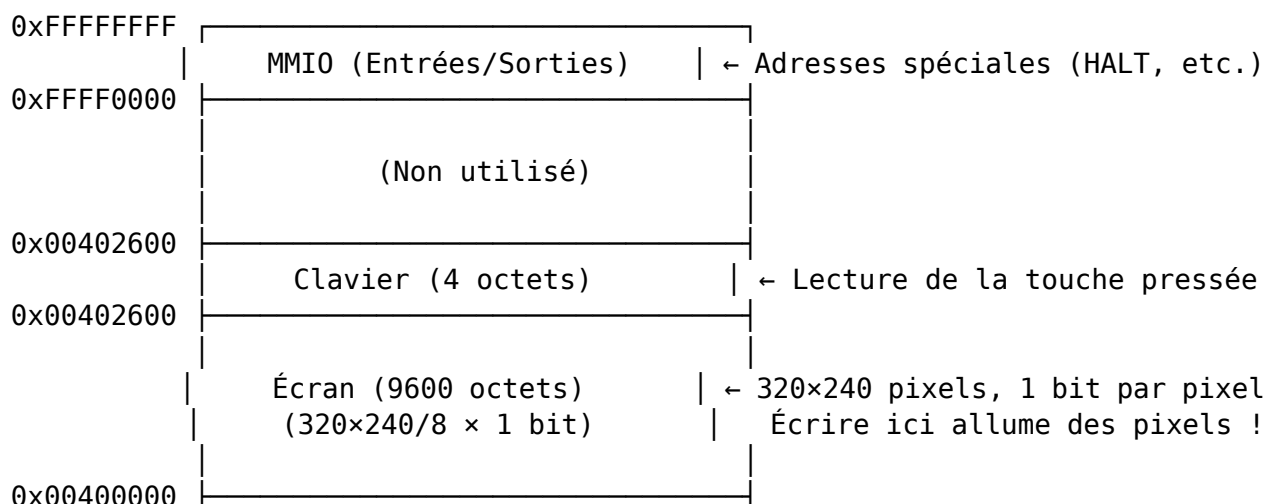
```
MOV PC, R14 ; Équivalent à "return" : saute à l'adresse dans LR
ADD PC, PC, #8 ; Saute de 8 octets plus loin
```

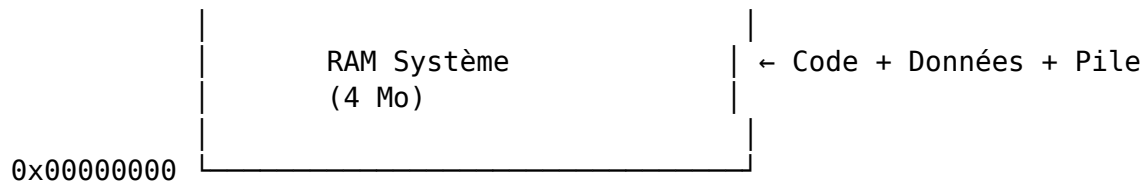
C'est puissant mais dangereux — une erreur de calcul et le CPU saute n'importe où !

## 6.6 La Carte Mémoire (Memory Map)

La mémoire est un espace linéaire de 4 Go ( $2^{32}$  octets), mais toutes les adresses ne sont pas utilisables.

### 6.6.1 Organisation de la mémoire Codex





### 6.6.2 Le Memory-Mapped I/O (MMIO)

En Codex (comme en ARM), les périphériques sont accessibles **comme de la mémoire**. Il n'y a pas d'instructions spéciales **IN / OUT**.

**L'écran** : - Adresse : 0x00400000 à 0x00402580 - Format : 1 bit par pixel, 40 octets par ligne - Écrire un 1 à un bit = pixel blanc

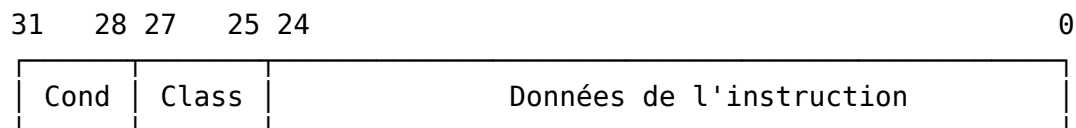
**Le clavier** : - Adresse : 0x00402600 - Lire cette adresse donne le code ASCII de la touche pressée (ou 0)

**Exemple** : Allumer le premier pixel

```
LDR R0, =0x00400000 ; Adresse de l'écran
MOV R1, #0x80       ; Bit 7 = premier pixel de la ligne
STRB R1, [R0]        ; Écrire un octet
```

## 6.7 Le Format des Instructions

Chaque instruction est encodée sur **32 bits**. La structure générale :



### 6.7.1 Les bits de condition (31-28)

**Fonctionnalité unique de ARM/Codex** : Toute instruction peut être conditionnelle !

Au lieu de :

```
CMP R0, #0
BNE skip
MOV R1, #1
skip:
```

On peut écrire :

```
CMP R0, #0
MOV.EQ R1, #1 ; Exécuté SEULEMENT si Z=1 (égal)
```

Code	Suffixe	Condition	Signification
0000	EQ	$Z = 1$	Égal (Equal)
0001	NE	$Z = 0$	Différent (Not Equal)
0010	CS/HS	$C = 1$	Retenue (Carry Set)
0011	CC/LO	$C = 0$	Pas de retenue (Carry Clear)
1010	GE	$N = V$	Plus grand ou égal (signé)
1011	LT	$N \neq V$	Plus petit (signé)
1100	GT	$Z=0$ et $N=V$	Plus grand (signé)
1101	LE	$Z=1$ ou $N \neq V$	Plus petit ou égal (signé)
1110	AL	(toujours)	Toujours exécuter (défaut)

### 6.7.2 Les classes d'instructions (27-25)

Bits 27-25	Classe	Description
000	Data Processing (reg)	Opérations ALU avec registre
001	Data Processing (imm)	Opérations ALU avec immédiat
010	Load/Store	Accès mémoire (LDR, STR)
011	Branch	Branchements (B, BL)
100	Block Transfer	Push/Pop multiple (LDM, STM)
111	System	Instructions système (SVC, HALT)

## 6.8 Les Instructions en Détail

### 6.8.1 A. Opérations Arithmétiques et Logiques

**Format général :** `OP{cond}{S} Rd, Rn, Operand2`

Instruction	Opération	Exemple
ADD	Addition	ADD R1, R2, R3 → $R1 = R2 + R3$
SUB	Soustraction	SUB R1, R2, #5 → $R1 = R2 - 5$
AND	ET logique	AND R1, R2, R3 → $R1 = R2 \& R3$
ORR	OU logique	ORR R1, R2, R3 → $R1 = R2   R3$
EOR	XOR	EOR R1, R2, R3 → $R1 = R2 \wedge R3$
MOV	Copie	MOV R1, R2 → $R1 = R2$
MVN	Copie inversée	MVN R1, R2 → $R1 = \sim R2$
CMP	Comparaison	CMP R1, R2 → met à jour les flags
TST	Test bits	TST R1, R2 → AND sans stocker

**Le suffixe S :** Ajouter `S` met à jour les drapeaux NZCV.

ADDS R1, R2, R3 ; Met à jour les flags

ADD R1, R2, R3 ; Ne touche pas aux flags

### 6.8.2 B. Accès Mémoire (Load/Store)

**Format général :** LDR/STR{B} Rd, [Rn, #offset]

Instruction	Action	Exemple
LDR	Charger 32 bits	LDR R0, [R1] → R0 = MEM[R1]
STR	Stocker 32 bits	STR R0, [R1] → MEM[R1] = R0
LDRB	Charger 8 bits	LDRB R0, [R1] → R0 = MEM[R1] (1 octet)
STRB	Stocker 8 bits	STRB R0, [R1] → MEM[R1] = R0 (1 octet)

**Modes d'adressage :**

```
LDR R0, [R1]           ; Simple : adresse = R1
LDR R0, [R1, #4]       ; Offset : adresse = R1 + 4
LDR R0, [R1, R2]       ; Register : adresse = R1 + R2
```

### 6.8.3 C. Branchements

**Format :** B{cond} label ou BL{cond} label

Instruction	Action
B label	Saut inconditionnel
BL label	Branch with Link (appel de fonction)
BEQ label	Saut si égal (Z=1)
BNE label	Saut si différent (Z=0)
BLT label	Saut si plus petit (signé)
BGT label	Saut si plus grand (signé)

**Le mystère de BL :**

main:

```
BL ma_fonction          ; 1. Sauvegarde PC+4 dans LR
                        ; 2. Saute à ma_fonction
                        ; ... (on revient ici après le retour)
```

ma\_fonction:

```
; ... faire quelque chose ...
MOV PC, LR              ; Retour : saute à l'adresse dans LR
```

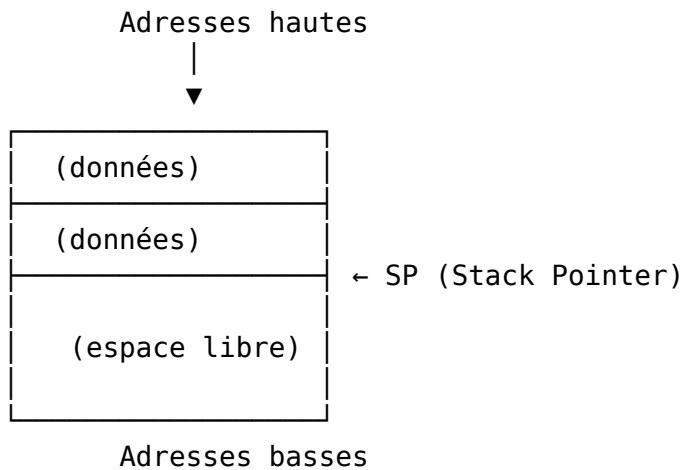
## 6.9 La Pile (Stack)

La pile est une zone de mémoire utilisée pour : - Sauvegarder les registres - Stocker les variables locales - Passer des arguments aux fonctions



### 6.9.1 Fonctionnement

La pile **grandit vers le bas** (des adresses hautes vers les basses) :



### 6.9.2 Push et Pop (manuel)

Codex n'a pas d'instructions `PUSH / POP` natives. On les simule :

```
; PUSH R0 (empiler R0)
SUB SP, SP, #4      ; Faire de la place (pile descend)
STR R0, [SP]        ; Stocker R0 au sommet

; POP R0 (dépiler dans R0)
LDR R0, [SP]        ; Lire depuis le sommet
ADD SP, SP, #4      ; Libérer l'espace
```

---

## 6.10 Exemples de Programmes

### 6.10.1 Exemple 1 : Somme de 1 à N

```
; Calcule 1 + 2 + ... + 10
.text
.global _start

_start:
    MOV R0, #0      ; sum = 0
    MOV R1, #1      ; i = 1

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

```
done:
    ; R0 contient 55
    HALT
```

### 6.10.2 Exemple 2 : Maximum de deux nombres (avec prédication)

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

### 6.10.3 Exemple 3 : Dessiner un pixel

```
; Allumer le pixel (10, 20)
; Adresse = 0x00400000 + (y * 40) + (x / 8)
; Bit = 7 - (x % 8)

LDR R0, =0x00400000 ; Base de l'écran
MOV R1, #20         ; y = 20
MOV R2, #40         ; octets par ligne
MUL R1, R1, R2       ; offset_y = y * 40
ADD R0, R0, R1       ; adresse_ligne

MOV R1, #10         ; x = 10
MOV R2, R1, LSR #3   ; x / 8
ADD R0, R0, R2       ; adresse_finale

AND R1, R1, #7       ; x % 8
RSB R1, R1, #7       ; 7 - (x % 8)
MOV R2, #1
LSL R2, R2, R1       ; masque = 1 << bit_pos

LDRB R3, [R0]        ; Lire l'octet actuel
ORR R3, R3, R2        ; Allumer le bit
STRB R3, [R0]        ; Écrire l'octet
```

## 6.11 Gestion des Erreurs (Traps)

Si le CPU rencontre une situation invalide, il déclenche une **trap** :

Trap	Cause
ILLEGAL	Instruction invalide (opcode inconnu)
MEM_FAULT	Accès à une adresse non mappée
MISALIGNED	Accès 32-bits à une adresse non alignée (ex: 0x3)

Trap	Cause
DIV_ZERO	Division par zéro

## 6.12 Exercices Pratiques

### 6.12.1 Exercices sur le Simulateur Web

Lancez le **Simulateur Web** et allez dans **A32 Assembly**.

Exercice	Description	Difficulté
Hello World	Afficher du texte	□
Addition	Additionner deux registres	□
Soustraction	Soustraire avec le drapeau	□
Logique	Opérations AND, OR, XOR	□
Conditions	Utiliser les branches conditionnelles	□□
Boucles	Implémenter une boucle while	□□
Multiplication	Multiplier par additions successives	□□
Fibonacci	Calculer la suite de Fibonacci	□□
Tableaux	Parcourir un tableau en mémoire	□□
Maximum Tableau	Trouver le max dans un tableau	□□□
Fonctions	Appeler des fonctions avec BL	□□□
Pixel	Allumer un pixel à l'écran	□□
Ligne	Dessiner une ligne	□□□
Rectangle	Dessiner un rectangle	□□□
Lire un Caractère	Lire le clavier	□□

### 6.12.2 Tests en ligne de commande

*# Assembler un fichier*

```
cargo run -p a32_cli -- assemble mon_prog.s -o mon_prog.bin
```

*# Exécuter un binaire*

```
cargo run -p a32_runner -- mon_prog.bin
```

## 6.13 Ce qu'il faut retenir

1. **L'ISA est un contrat** : Entre le matériel et le logiciel
2. **RISC = Simple** : Load, Compute, Store — jamais de calcul direct en mémoire
3. **16 registres** : R0-R12 généraux, R13=SP, R14=LR, R15=PC

4. **Tout est conditionnel** : `ADD.EQ` , `MOV.GT` évitent les branchements
5. **Memory-Mapped I/O** : L'écran et le clavier sont des adresses mémoire
6. **Le cycle Fetch-Decode-Execute** : Le cœur battant du CPU

**Prochaine étape** : Au Chapitre 5, nous construirons le CPU qui **implémente** cette architecture. Vous verrez comment les circuits du Chapitre 1-3 sont combinés pour exécuter ces instructions.

---

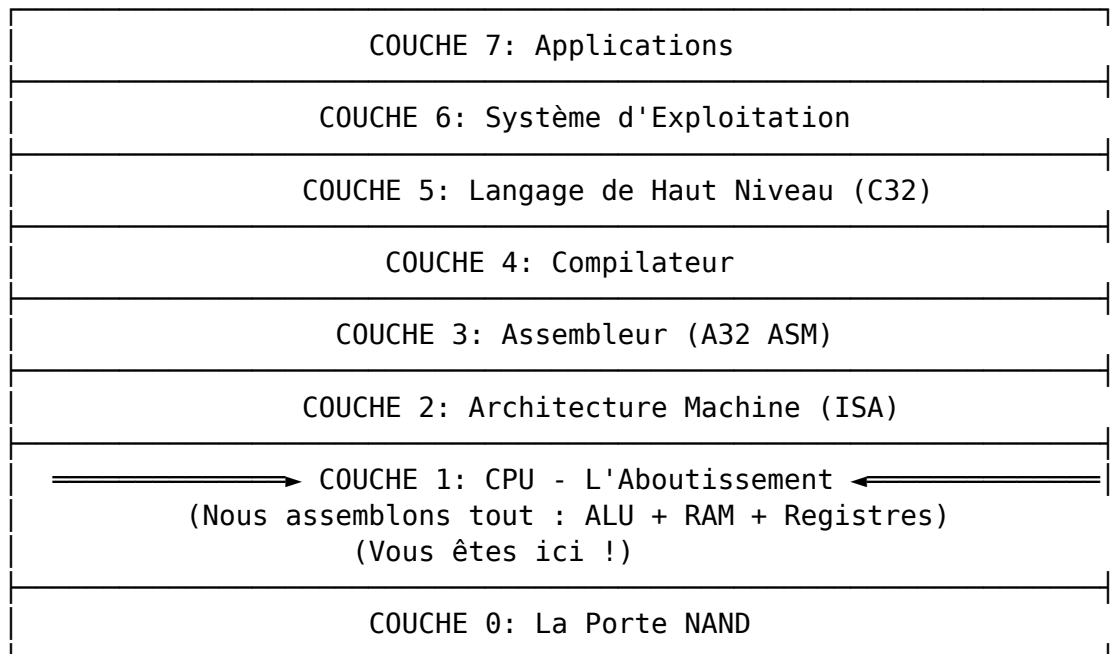
**Conseil** : Passez du temps sur le simulateur web à écrire des programmes en assembleur. Comprendre l'assembleur vous aidera énormément à comprendre le compilateur plus tard !

## 7 Le Processeur (CPU)

“Si vous ne pouvez pas le construire, vous ne le comprenez pas.” — Richard Feynman

C'est le grand moment. Nous allons assembler toutes les pièces du puzzle — portes logiques, ALU, registres, mémoire — pour construire le **cœur de l'ordinateur** : le CPU A32.

### 7.1 Où en sommes-nous ?



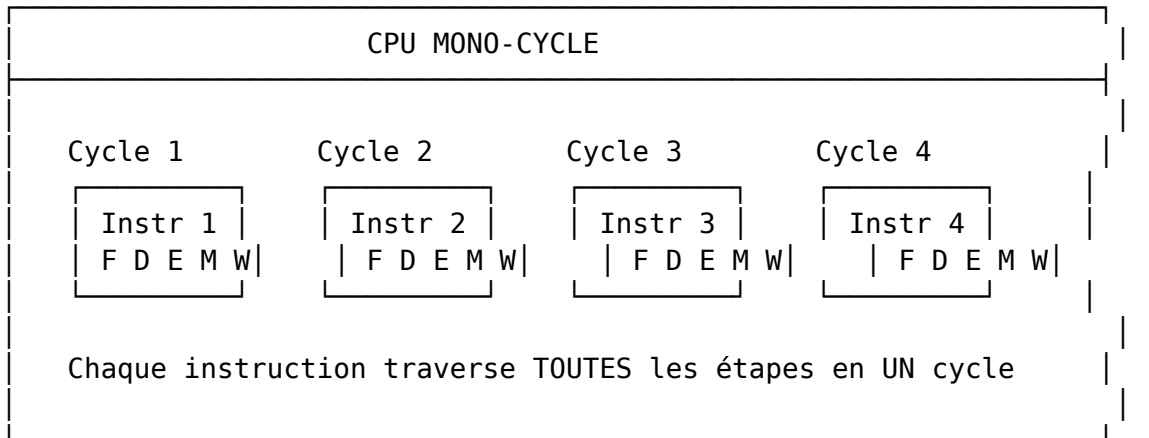
Ce chapitre est le **point culminant** de tout le travail matériel. Après ce chapitre, vous aurez construit un ordinateur complet capable d'exécuter du vrai code !

## 7.2 Deux Implémentations du CPU

Le projet Codex propose **deux implémentations** du CPU A32, chacune avec un objectif pédagogique différent :

### 7.2.1 CPU Mono-cycle (Simulateur Rust)

Le **simulateur Rust** ( `a32_core` ) implémente un CPU **mono-cycle** :

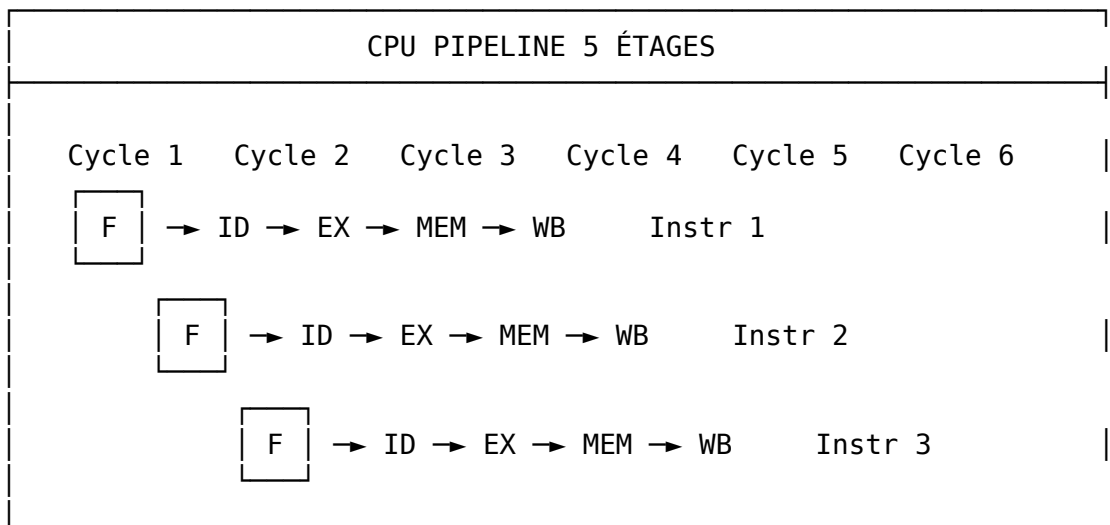


**Utilisé par :** - Le **CPU Visualizer** (interface web) - Le **runner** ( `a32_runner` ) - L'**IDE web** (exécution des programmes) - Les **tests** C32 et A32

**Avantages :** Simple à comprendre, facile à déboguer, comportement prévisible.

### 7.2.2 CPU Pipeline (HDL)

Le **CPU en HDL** ( `hdl_lib/05_cpu/CPU_Pipeline.hdl` ) implémente un vrai **pipeline 5 étages** :



Jusqu'à 5 instructions en vol simultanément !

### Composants HDL :

Fichier	Rôle
IF_ID_Reg.hdl	Registre pipeline IF→ID
ID_EX_Reg.hdl	Registre pipeline ID→EX
EX_MEM_Reg.hdl	Registre pipeline EX→MEM
MEM_WB_Reg.hdl	Registre pipeline MEM→WB
HazardDetect.hdl	Détection des hazards (stall)
ForwardUnit.hdl	Bypass/forwarding des données
CPU_Pipeline.hdl	CPU complet assemblé

**Utilisé par :** - Les **exercices HDL** (apprentissage hardware) - Le **simulateur HDL** (hdl\_cli)

**Avantages :** Réaliste, montre les vrais défis du design CPU (hazards, forwarding, stalls).

#### 7.2.3 Pourquoi deux implémentations ?

Aspect	Mono-cycle (Rust)	Pipeline (HDL)
<b>Objectif</b>	Exécuter des programmes	Apprendre le hardware
<b>Complexité</b>	Simple	Réaliste
<b>Performance</b>	1 instr/cycle	Jusqu'à 1 instr/cycle (avec hazards)
<b>Hazards</b>	Aucun	Gérés (stall + forward)
<b>Utilisation</b>	IDE, tests, visualizer	Exercices HDL

**Note :** Le CPU Visualizer affiche les étapes (Fetch, Decode, Execute, Memory, Writeback) de manière **pédagogique**, mais l'exécution sous-jacente est mono-cycle. Pour voir un vrai pipeline avec hazards et forwarding, utilisez le simulateur HDL avec CPU\_Pipeline.hdl.

## 7.3 Qu'est-ce qu'un CPU ?

### 7.3.1 Le chef d'orchestre

Le CPU (Central Processing Unit) est le composant qui : 1. **Lit** les instructions depuis la mémoire 2. **Décode** ces instructions pour comprendre quoi faire 3. **Exécute** les opérations (calculs, accès mémoire, branchements) 4. **Répète** à l'infini (jusqu'à HALT) C'est une machine à états qui exécute une instruction après l'autre, inlassablement.

### 7.3.2 Ce que nous avons construit jusqu'ici

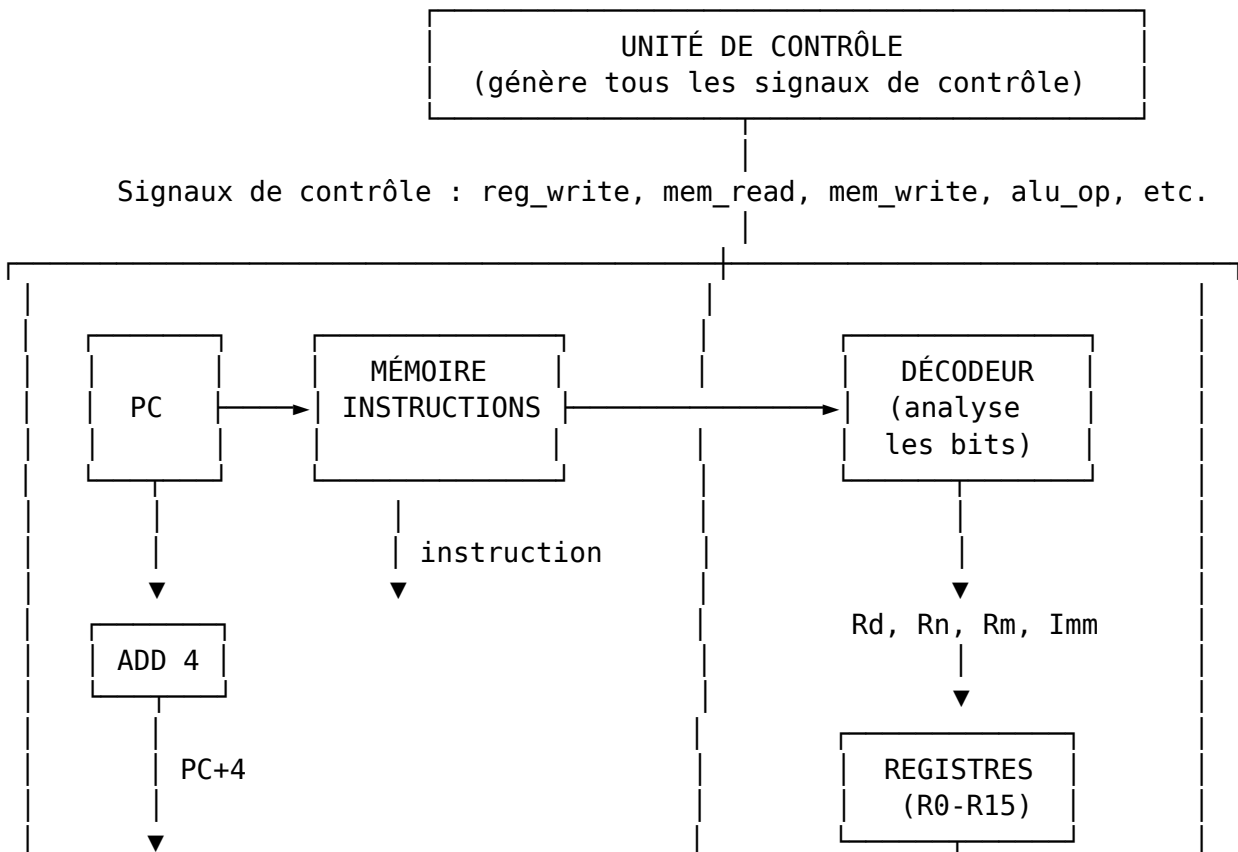
Chapitre	Composant	Rôle dans le CPU
1	Portes logiques	Briques de base de tout circuit
2	ALU	Effectue les calculs (ADD, SUB, AND...)
3	Registres	Stockent les données du CPU (R0-R15)
3	PC	Pointe vers l'instruction courante
3	RAM	Stocke le programme et les données
4	ISA	Définit les instructions à supporter

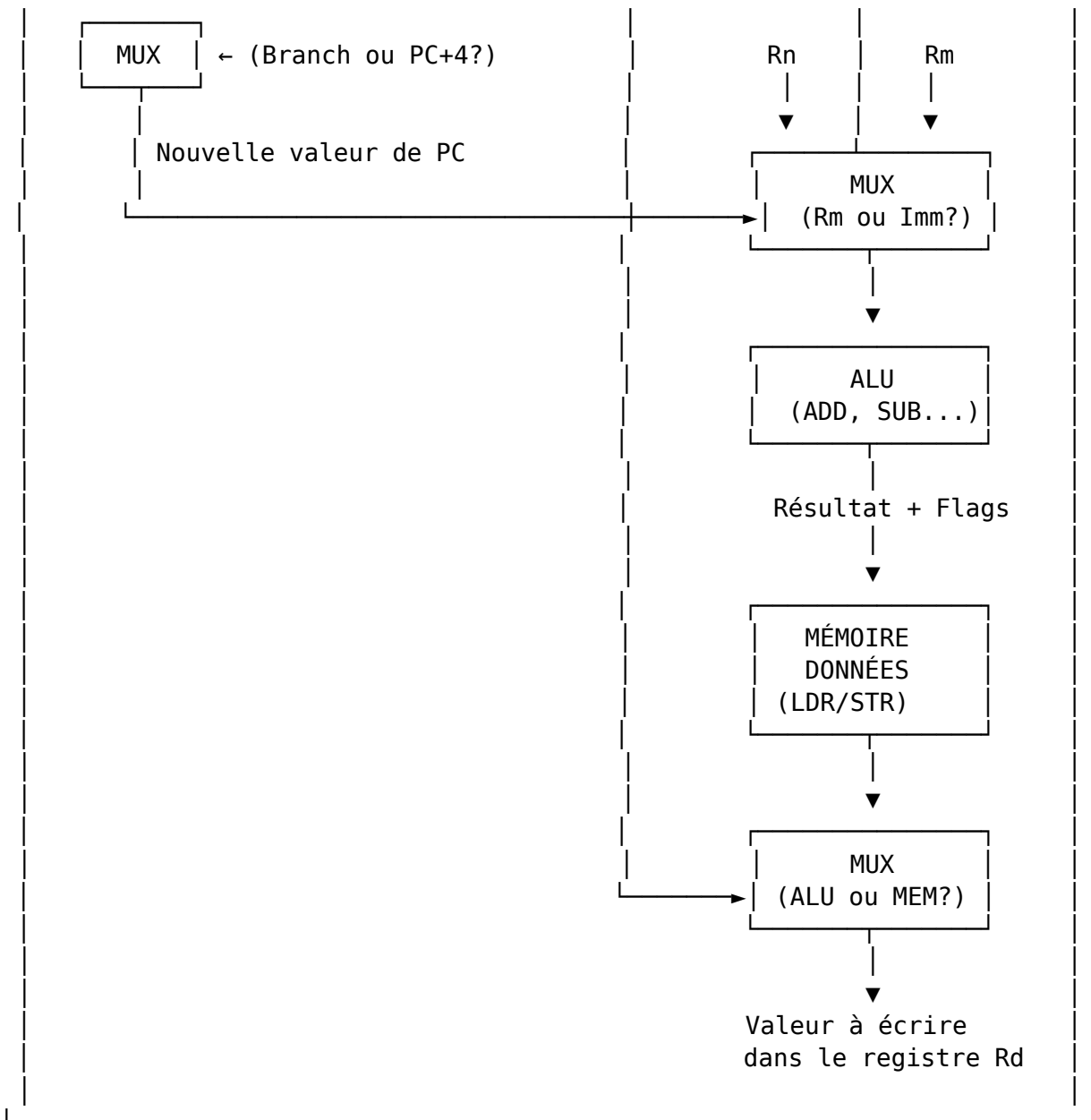
### 7.3.3 Ce qu'il reste à construire

- **Décodeur** : Analyse les bits de l'instruction
- **Unité de contrôle** : Décide quoi activer
- **Multiplexeurs de données** : Routent les données entre composants
- **Le CPU lui-même** : L'assemblage final

## 7.4 Architecture du CPU (Data Path)

Voici le schéma complet du CPU. Chaque flèche est un fil (ou un bus de 32 fils). Chaque boîte est un composant que vous avez construit ou que vous allez construire.





#### 7.4.1 Les flux de données

1. **Fetch** : PC → Mémoire Instructions → Instruction (32 bits)
2. **Decode** : Instruction → Décodeur → (Rd, Rn, Rm, Imm, opcode)
3. **Register Read** : Rn, Rm → Banc de Registres → Valeurs
4. **Execute** : Valeurs → ALU → Résultat
5. **Memory** : Résultat → Mémoire Données (si LDR/STR)
6. **Writeback** : Résultat → Registre Rd



## 7.5 Les Composants du CPU

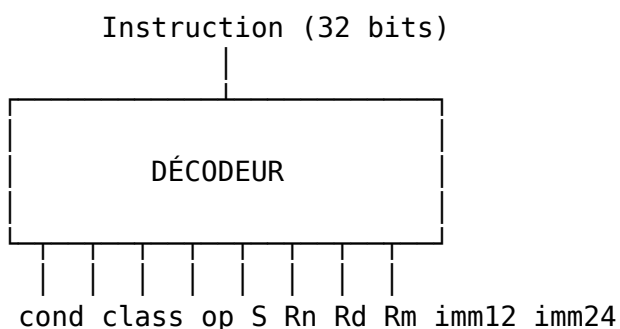
### 7.5.1 1. Le Compteur de Programme (PC)

Vous l'avez déjà construit au Chapitre 3 ! Le PC contient l'adresse de l'instruction courante.

**Modes de fonctionnement** : - `inc = 1` :  $PC \leftarrow PC + 4$  (instruction suivante) - `load = 1` :  $PC \leftarrow$  adresse de branchement - `reset = 1` :  $PC \leftarrow 0$  (redémarrage)

### 7.5.2 2. Le Décodeur (Decoder)

Le décodeur est un circuit **purement combinatoire** qui "découpe" les 32 bits de l'instruction.



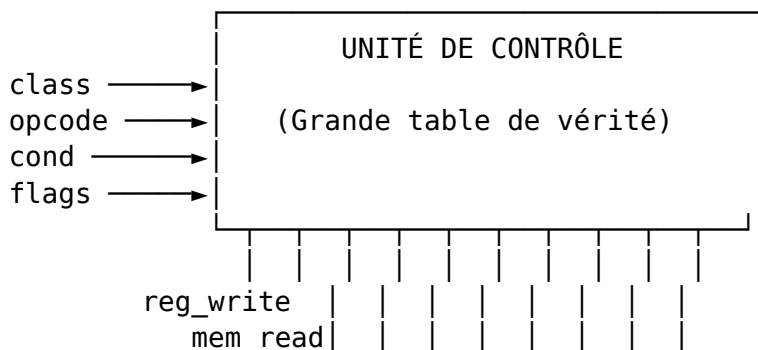
**Sorties du décodeur** :

Signal	Bits	Description
cond	31-28	Code de condition (EQ, NE, LT...)
class	27-25	Classe d'instruction (ALU, MEM, BRANCH)
op	24-21	Opération ALU (ADD, SUB, AND...)
S	20	Mettre à jour les drapeaux ?
Rn	19-16	Registre source 1
Rd	15-12	Registre destination
Rm	3-0	Registre source 2
imm12	11-0	Valeur immédiate (12 bits)
imm24	23-0	Offset de branchement (24 bits)

Le décodeur ne fait que du **câblage** — il ne calcule rien, il ne fait que router les bits vers les bonnes sorties.

### 7.5.3 3. L'Unité de Contrôle (Control)

L'unité de contrôle est le **chef d'orchestre**. Elle regarde la classe et l'opcode, et décide quels signaux activer.



**Exemples de signaux de contrôle :**

Instruction	reg_write	mem_read	mem_write	alu_src	branch
ADD	1	0	0	0 (reg)	0
ADD #imm	1	0	0	1 (imm)	0
LDR	1	1	0	1 (imm)	0
STR	0	0	1	1 (imm)	0
B	0	0	0	X	1
CMP	0	0	0	0	0

**7.5.4 4. Le Vérificateur de Condition (CondCheck)**

Ce petit circuit vérifie si la condition est satisfaite.

**Entrées** : - **cond** : Le code de condition (4 bits, ex: 0000 = EQ) - **flags** : Les drapeaux NZCV

**Sortie** : - **ok** : 1 si la condition est vraie, 0 sinon

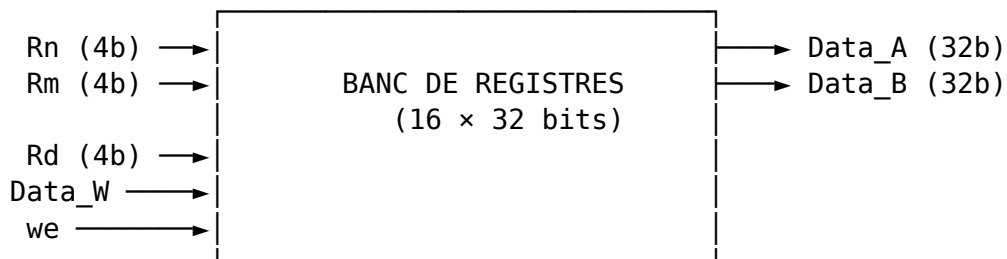
cond = 0000 (EQ) et Z = 1 → ok = 1  
 cond = 0000 (EQ) et Z = 0 → ok = 0  
 cond = 1110 (AL) → ok = 1 (toujours)

**Pourquoi est-ce important ?**

Si **ok = 0**, l'instruction est "annulée" — on n'écrit pas dans le registre, on ne fait pas le branchement. C'est la **prédication** en action !

**7.5.5 5. Le Banc de Registres (RegFile)**

Vous l'avez construit au Chapitre 3 (RAM8, RAM16...). Le banc de registres est une RAM spéciale avec : - **2 ports de lecture** : Lire Rn ET Rm simultanément - **1 port d'écriture** : Écrire dans Rd

**7.5.6 6. Les Multiplexeurs**

Les multiplexeurs routent les données entre les composants :

Mux	Choix	Signification
<b>ALU_src</b>	0: Rm, 1: Imm	Deuxième opérande de l'ALU
<b>Writeback</b>	0: ALU, 1: MEM	Source de la valeur à écrire
<b>PC_src</b>	0: PC+4, 1: Branch	Prochaine valeur du PC

## 7.6 Le Cycle d'Exécution en Détail

Notre CPU est **single-cycle** : chaque instruction s'exécute en un seul cycle d'horloge.

### 7.6.1 Phase 1 : Fetch (Récupération)

PC → Mémoire Instructions → instruction (32 bits)

Le PC envoie son adresse à la mémoire d'instructions. La mémoire renvoie les 32 bits de l'instruction.

### 7.6.2 Phase 2 : Decode (Décodage)

instruction → Décodeur → cond, class, op, Rn, Rd, Rm, imm  
 |  
 → Unité de Contrôle → signaux

Le décodeur découpe l'instruction. L'unité de contrôle décide quoi activer.

### 7.6.3 Phase 3 : Register Read (Lecture des registres)

Rn, Rm → Banc de Registres → Data\_A, Data\_B

Les valeurs des registres sources sont lues.

### 7.6.4 Phase 4 : Execute (Exécution)

Data\_A ————  
 Data\_B ou Imm → MUX ————  
 } → ALU → Résultat, Flags

L'ALU effectue l'opération. Les drapeaux sont mis à jour (si S=1).

### 7.6.5 Phase 5 : Memory (Accès mémoire)

Résultat (adresse) ————  
 { ———— Si LDR : MEM[addr] → valeur  
 ———— Si STR : valeur → MEM[addr]

Pour les instructions LDR/STR, on accède à la mémoire de données.

### 7.6.6 Phase 6 : Writeback (Écriture)

Résultat (ALU ou MEM) → MUX → Banc de Registres → Rd

Si `reg_write = 1` ET `cond_ok = 1`, on écrit dans le registre destination.

### 7.6.7 Phase 7 : PC Update

PC+4 — { Si branch : PC + (offset × 4)  
Sinon : PC + 4

Le PC est mis à jour pour le prochain cycle.

## 7.7 Implémentation du CPU en HDL

Voici un squelette de l'architecture du CPU :

```
entity CPU is
  port(
    clk      : in bit;
    reset    : in bit;
    -- Interface mémoire
    instr    : in bits(31 downto 0);
    mem_in   : in bits(31 downto 0);
    pc_out   : out bits(31 downto 0);
    mem_addr : out bits(31 downto 0);
    mem_out  : out bits(31 downto 0);
    mem_we   : out bit
  );
end entity;

architecture rtl of CPU is
  -- Signaux internes
  signal pc, pc_next : bits(31 downto 0);
  signal cond, op : bits(3 downto 0);
  signal rd, rn, rm : bits(3 downto 0);
  signal imm12 : bits(11 downto 0);
  signal data_a, data_b, alu_result : bits(31 downto 0);
  signal reg_write, mem_read, mem_write, alu_src, branch : bit;
  signal n_flag, z_flag, c_flag, v_flag, cond_ok : bit;

begin
  -- Instanciation des composants
  u_decoder: Decoder port map (...);
  u_control: Control port map (...);
  u_condcheck: CondCheck port map (...);
  u_regfile: RegFile port map (...);
```

```

u_alu: ALU port map (...);
u_pc: PC port map (...);

-- Multiplexeurs
alu_b <= imm12 when alu_src = '1' else data_b;
writeback <= mem_in when mem_read = '1' else alu_result;
pc_next <= branch_addr when (branch and cond_ok) = '1' else pc_plus_4;

end architecture;

```

## 7.8 Exercices Pratiques

### 7.8.1 Exercices sur le Simulateur Web

Lancez le **Simulateur Web** et allez dans **HDL Progression** → **Projet 5 : CPU**.

Exercice	Description	Difficulté
Decoder	Découper l'instruction en champs	□□
CondCheck	Vérifier les conditions (EQ, NE, LT...)	□□
Control	Générer les signaux de contrôle	□□□
CPU	L'assemblage final !	□□□□

### 7.8.2 Ordre de progression

1. **Decoder** : Commencez par là. C'est du pur câblage.
  - Utilisez la syntaxe `instr(31 downto 28)` pour extraire les bits
2. **CondCheck** : Table de vérité des conditions
  - EQ :  $Z = 1$
  - NE :  $Z = 0$
  - LT :  $N \neq V$
  - etc.
3. **Control** : La logique de commande
  - Pour chaque classe d'instruction, décidez les signaux
  - Attention aux cas spéciaux (CMP ne fait pas `reg_write`)
4. **CPU** : L'assemblage final
  - Suivez le schéma du data path
  - N'oubliez pas les multiplexeurs !

### 7.8.3 Tests en ligne de commande

# Tester le décodeur

```
cargo run -p hdl_cli -- test hdl_lib/05_cpu/Decoder.hdl
```

# Tester le CPU complet

```
cargo run -p hdl_cli -- test hdl_lib/05_cpu/CPU.hdl
```

## 7.9 CPU Visualizer : L'Outil Interactif

Pour mieux comprendre comment le CPU exécute les instructions, le projet inclut un **CPU Visualizer** interactif. C'est un outil web qui vous permet de voir en temps réel le fonctionnement du processeur.

### 7.9.1 Accéder au Visualizer

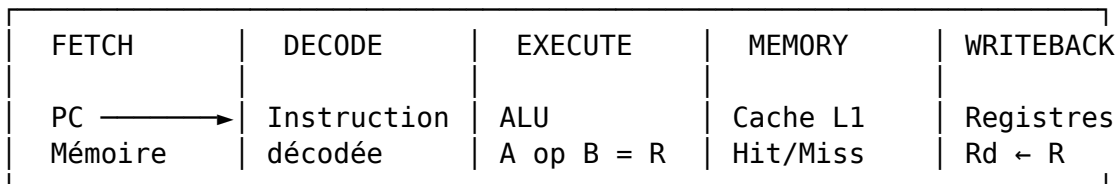
1. Lancez le serveur web :

```
cd web
npm install
npm run dev
```

2. Ouvrez votre navigateur à l'adresse indiquée (généralement <http://localhost:5173>)
3. Cliquez sur **CPU Visualizer** dans la barre de navigation

### 7.9.2 Fonctionnalités du Visualizer

**7.9.2.1 Vue Pipeline** Le Visualizer affiche les **5 étapes du cycle d'exécution** :



Chaque étape s'illumine en jaune quand elle est active, vous permettant de suivre la progression de l'instruction.

**7.9.2.2 Panneau Registres** Affiche les **16 registres** (R0-R15) avec les alias : - **SP** (R13) : Stack Pointer - **LR** (R14) : Link Register - **PC** (R15) : Program Counter

Les registres modifiés s'illuminent en vert pendant un instant.

**7.9.2.3 Panneau Flags (CPSR)** Les 4 drapeaux du processeur sont affichés : - **N** (Negative) : Le résultat est négatif - **Z** (Zero) : Le résultat est zéro - **C** (Carry) : Retenue/emprunt - **V** (Overflow) : Débordement signé

Les flags changent de couleur quand ils sont actifs.

**7.9.2.4 Panneau Code Source** Affiche le code assembleur avec : - **Coloration syntaxique** : Instructions, registres, nombres, commentaires - **Surlignage de la ligne courante** : La ligne en cours d'exécution est mise en évidence en jaune - **Défilement automatique** : Le code défile pour suivre l'exécution

**7.9.2.5 Panneau Mémoire et Cache** Affiche : - **Vue mémoire** : Les octets en mémoire autour du PC - **Statistiques cache** : Hits, Misses, Taux de réussite - **Contenu du cache L1** : Lignes valides avec tag et données - **Indicateur HIT/MISS** : Flash vert pour hit, rouge pour miss

### 7.9.3 Les Démonstrations Intégrées

Le Visualizer inclut 7 démonstrations prêtes à l'emploi :

Demo	Description	Concept illustré
<b>1. Addition</b>	$5 + 3 = 8$	Instructions ALU basiques
<b>2. Boucle</b>	Somme 1-5	Branchements conditionnels
<b>3. Mémoire</b>	LDR/STR	Accès mémoire
<b>4. Condition</b>	Valeur absolue	Prédication
<b>5. Tableau</b>	Somme tableau	Boucle + mémoire
<b>6. Flags</b>	N, Z, C, V	Drapeaux CPU
<b>7. Cache</b>	Parcours mémoire	Cache hits/misses

### 7.9.4 Contrôles

Bouton	Raccourci	Action
<b>Reset</b>	Ctrl+R	Remet le CPU à zéro
<b>Step</b>	N, F10	Exécute une instruction
<b>Play/Pause</b>	Espace	Lance/arrête l'exécution continue
<b>Vitesse</b>	Slider	Ajuste la vitesse d'exécution

### 7.9.5 Charger Votre Propre Code

1. Cliquez sur **Charger fichier**
2. Sélectionnez un fichier `.asm`, `.a32` ou `.a32b`
3. Le code est assemblé et chargé automatiquement

### 7.9.6 Exercice Pratique

Utilisez le Visualizer pour observer ces comportements :

1. **Suivez une addition** : Chargez la démonstration "Addition" et observez comment ADD lit deux registres et écrit le résultat
2. **Observez un branchement** : Chargez la démonstration "Boucle" et regardez comment B.LE revient au début de la boucle
3. **Analysez le cache** : Chargez la démonstration "Cache" et observez les miss au premier parcours, puis les hits au second

## 7.10 Conseils de Débogage

### 7.10.1 Le PC reste à 0 ?

- Vérifiez que `inc = 1` par défaut
- Vérifiez que le reset n'est pas bloqué

### 7.10.2 Les branchements ne marchent pas ?

- L'offset dans l'instruction est en mots ( $\times 4$  pour avoir des octets)
- Vérifiez que `cond_ok` est correct
- Vérifiez le calcul de l'adresse de branchement

### 7.10.3 Rien ne s'écrit dans les registres ?

- `reg_write` doit être à 1
- `cond_ok` doit être à 1
- Le registre destination ne doit pas être R15 (géré à part)

### 7.10.4 LDR/STR ne fonctionne pas ?

- Vérifiez le calcul de l'adresse (base + offset)
- Vérifiez les signaux `mem_read` et `mem_write`
- Attention à l'alignement (adresses multiples de 4)

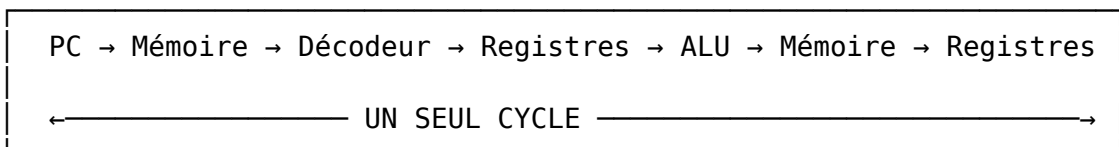
## 7.11 Aller Plus Loin : Le CPU Pipeline

Le CPU single-cycle que nous avons construit est simple et pédagogique. Mais dans le monde réel, il serait **très lent**. Les vrais processeurs utilisent une technique appelée **pipeline** pour être beaucoup plus rapides.

Cette section explique en détail ce qu'est un pipeline, pourquoi il est nécessaire, et comment le construire.

### 7.11.1 Pourquoi le CPU Single-Cycle est Lent

**7.11.1.1 Le problème de la chaîne critique** Dans notre CPU single-cycle, une instruction doit traverser **tous** les composants en un seul cycle :





Le cycle d'horloge doit être assez **long** pour que le signal traverse tout ce chemin. Si chaque étape prend 1 nanoseconde, le cycle doit faire au minimum 6 ns.

**Résultat** : Même si certaines instructions n'ont pas besoin de la mémoire de données (comme ADD), elles prennent quand même 6 ns.

**7.11.1.2 Une analogie : La laverie** Imaginez que vous avez 4 lessives à faire. Chaque lessive a 4 étapes : 1. **Laver** (30 min) 2. **Sécher** (30 min) 3. **Plier** (30 min) 4. **Ranger** (30 min)

**Approche "single-cycle"** : Attendre qu'une lessive soit complètement terminée avant de commencer la suivante.

Lessive 1 : |—Laver—|—Sécher—|—Plier—|—Ranger—|  
Lessive 2 : |—Laver—|—Sécher—|...

Temps total : 4 lessives  $\times$  2h = **8 heures**

**Approche "pipeline"** : Dès que la machine à laver est libre, commencer la lessive suivante !

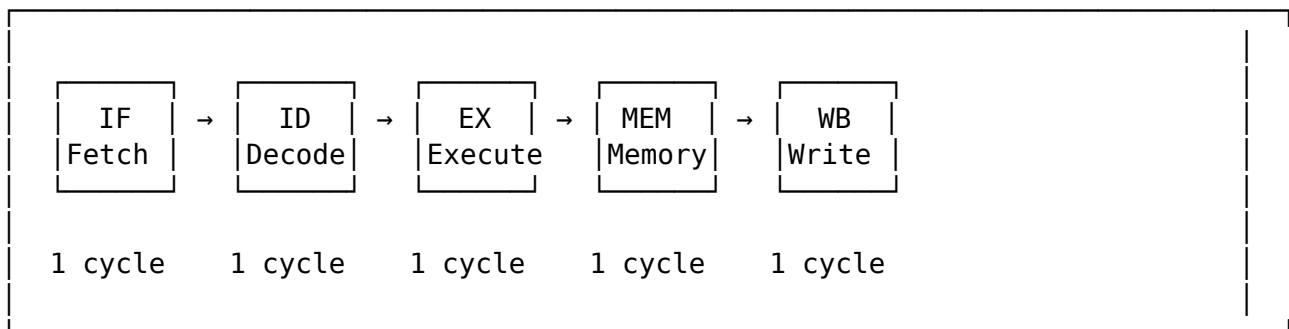
Lessive 1 : |—Laver—|—Sécher—|—Plier—|—Ranger—|  
Lessive 2 : |—Laver—|—Sécher—|—Plier—|—Ranger—|  
Lessive 3 : |—Laver—|—Sécher—|—Plier—|—Ranger—|  
Lessive 4 : |—Laver—|—Sécher—|—Plier—|—Ranger—|

Temps total : 2h + 3  $\times$  30min = **3h30**

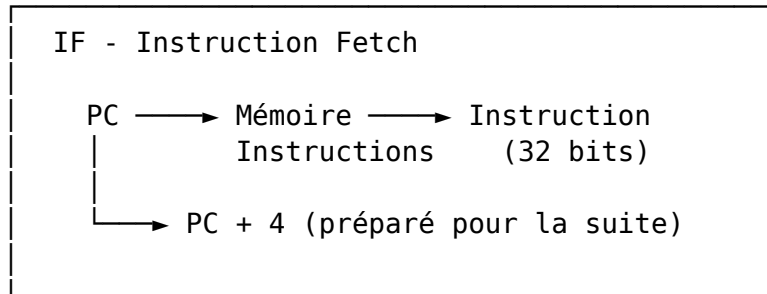
Le pipeline ne rend pas une lessive individuelle plus rapide, mais il permet de traiter **plus de lessives par heure** !

### 7.11.2 Le Pipeline à 5 Étages

Notre CPU pipeliné divise l'exécution en **5 étapes**, chacune prenant exactement 1 cycle d'horloge :



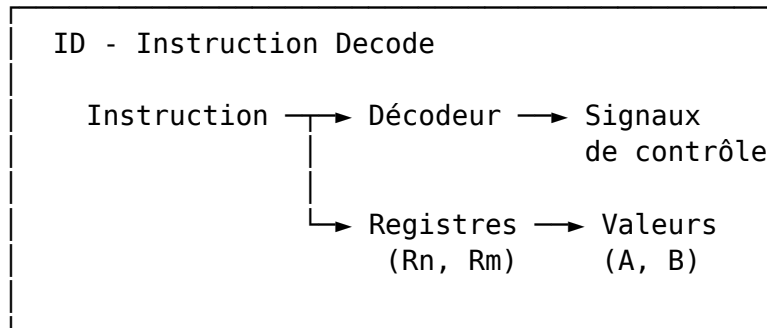
**7.11.2.1 Étape 1 : IF (Instruction Fetch)** **But** : Aller chercher l'instruction en mémoire.



**Ce qui se passe** : 1. Le PC (Program Counter) envoie son adresse à la mémoire 2. La mémoire renvoie l'instruction (32 bits) 3. On calcule PC + 4 pour l'instruction suivante

**Sortie** : L'instruction et PC+4 sont stockés dans le registre IF/ID.

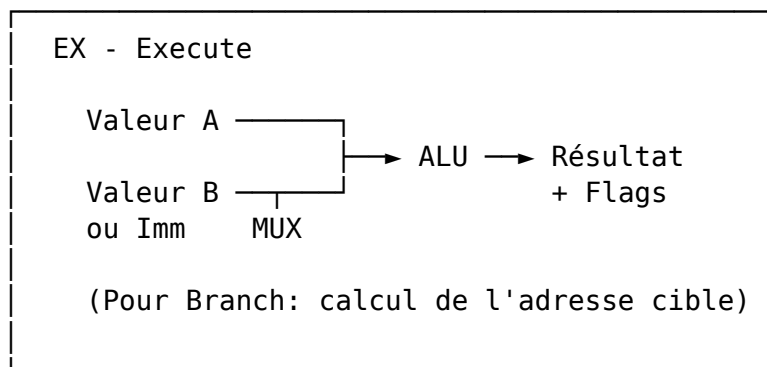
**7.11.2.2 Étape 2 : ID (Instruction Decode) But** : Comprendre l'instruction et lire les registres sources.



**Ce qui se passe** : 1. Le décodeur extrait les champs (Rd, Rn, Rm, opcode, etc.) 2. L'unité de contrôle génère les signaux (reg\_write, mem\_read, etc.) 3. On lit les valeurs des registres Rn et Rm 4. On détecte les éventuels aléas (hazards)

**Sortie** : Tout est stocké dans le registre ID/EX.

**7.11.2.3 Étape 3 : EX (Execute) But** : Effectuer le calcul.

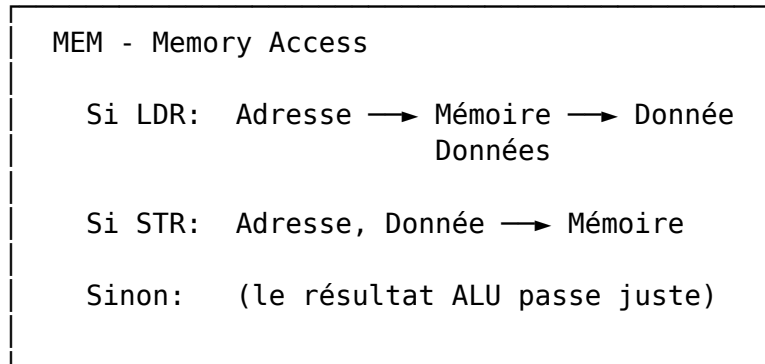


**Ce qui se passe** : 1. L'ALU effectue l'opération (ADD, SUB, AND, etc.) 2. Les flags (N, Z, C, V) sont calculés 3. Pour les branchements, on calcule l'adresse cible 4. Le

forwarding peut injecter des valeurs ici (on verra plus tard)

**Sortie** : Le résultat ALU est stocké dans le registre EX/MEM.

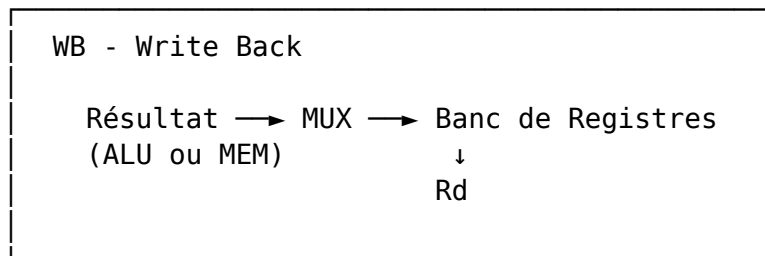
**7.11.2.4 Étape 4 : MEM (Memory Access) But** : Lire ou écrire en mémoire (pour LDR/STR seulement).



**Ce qui se passe** : 1. Pour LDR : on lit la mémoire à l'adresse calculée 2. Pour STR : on écrit la valeur en mémoire 3. Pour les autres instructions : rien (le résultat ALU est juste transmis)

**Sortie** : Le résultat (ALU ou mémoire) est stocké dans le registre MEM/WB.

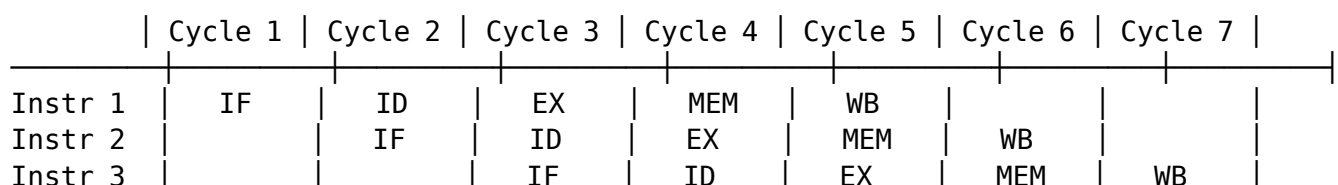
**7.11.2.5 Étape 5 : WB (Write Back) But** : Écrire le résultat dans le registre destination.

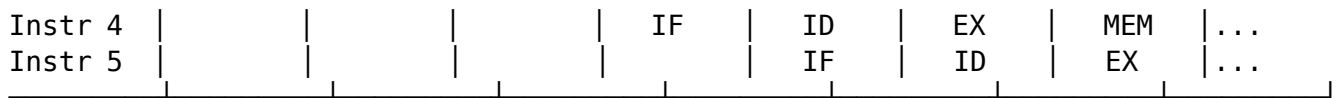


**Ce qui se passe** : 1. On choisit le résultat à écrire (ALU ou mémoire) 2. Si `reg_write = 1`, on écrit dans le registre Rd

### 7.11.3 Visualisation du Pipeline en Action

Voici comment 5 instructions traversent le pipeline :





**Observation clé :** À partir du cycle 5, le pipeline est “rempli” et on termine **une instruction par cycle** !

**Comparaison des performances :**

CPU	100 instructions	Temps (si cycle = 1 ns)
Single-cycle	100 cycles	100 ns
Pipeline 5 étages	104 cycles*	104 ns

\*Attendez... le pipeline n’est pas plus rapide ?

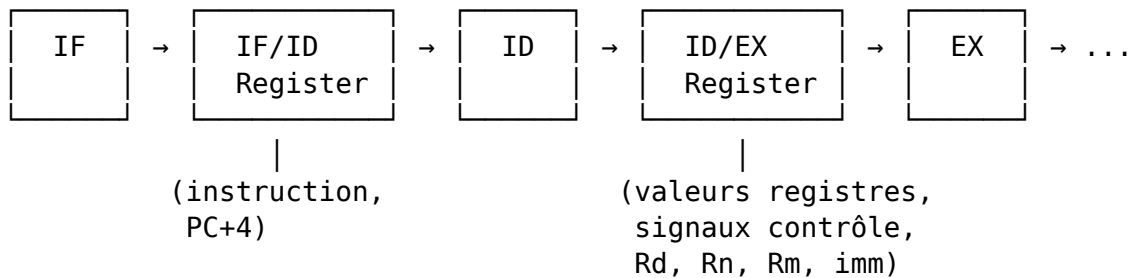
C’est parce que le cycle du pipeline est **5× plus court** ! Chaque étage ne fait qu’une partie du travail.

CPU	Durée cycle	100 instructions	Temps réel
Single-cycle	5 ns	100 cycles	<b>500 ns</b>
Pipeline	1 ns	104 cycles	<b>104 ns</b>

Le pipeline est **~5× plus rapide** !

#### 7.11.4 Les Registres de Pipeline

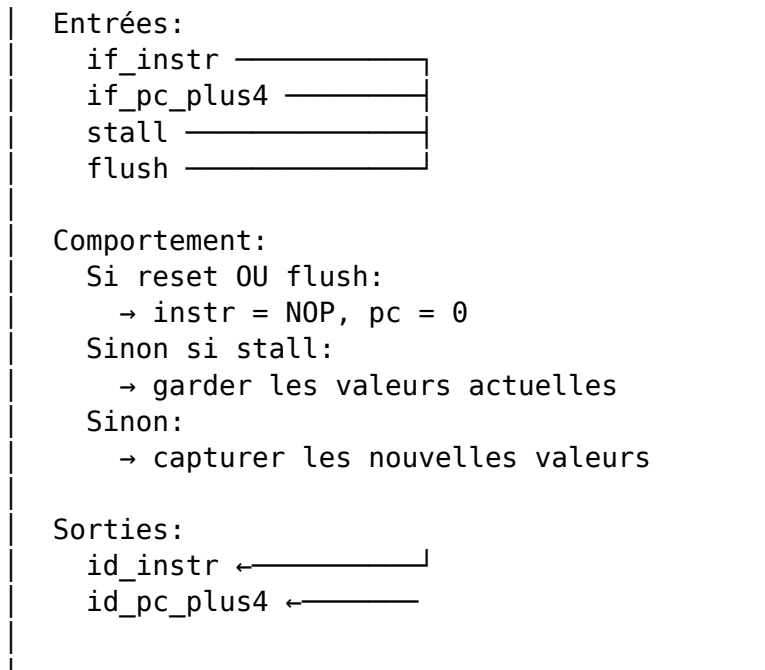
Pour que le pipeline fonctionne, il faut **stocker** les résultats intermédiaires entre chaque étage. C’est le rôle des **registres de pipeline**.



##### 7.11.4.1 Le registre IF/ID Stocke : - L’instruction (32 bits) - PC+4 (32 bits)

**Signaux spéciaux :** - **stall** : Si 1, garder les mêmes valeurs (ne pas avancer) - **flush** : Si 1, mettre l’instruction à NOP (annuler)





### 7.11.5 Les Aléas (Hazards)

Le pipeline crée de nouveaux problèmes. Quand une instruction dépend du résultat d'une instruction précédente qui n'est pas encore terminée, on a un **aléa**.

#### 7.11.5.1 Aléa de Données (Data Hazard) Exemple problématique :

ADD R1, R2, R3 ; Instruction 1: R1 = R2 + R3  
 SUB R4, R1, R5 ; Instruction 2: R4 = R1 - R5 (utilise R1!)

Visualisons dans le pipeline :

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	
ADD R1	IF	ID	EX	MEM	WB	← R1 écrit ici!
SUB R4, R1		IF	ID			← R1 lu ici!

**Le problème** : SUB lit R1 au cycle 3 (étage ID), mais ADD n'écrit R1 qu'au cycle 5 (étage WB). SUB va lire l'**ancienne** valeur de R1 !

**7.11.5.2 Solution 1 : Le Forwarding (Bypass)** Au lieu d'attendre que R1 soit écrit dans le banc de registres, on peut **transférer** le résultat directement depuis l'étage EX ou MEM vers l'étage où on en a besoin.

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	
ADD R1	IF	ID	EX			Résultat disponible!
SUB R4, R1		IF	ID	EX		← Forward!

|  
(on prend le résultat  
directement de ADD)

**Le ForwardUnit** détecte ces situations et redirige les données :

#### ForwardUnit

##### Entrées:

ex\_rn, ex\_rm : registres sources de l'instruction en EX  
 mem\_rd : registre destination en MEM  
 mem\_reg\_write : MEM va écrire un registre?  
 wb\_rd : registre destination en WB  
 wb\_reg\_write : WB va écrire un registre?

##### Logique:

Si MEM.rd = EX.rn et MEM écrit → forward depuis MEM  
 Sinon si WB.rd = EX.rn et WB écrit → forward depuis WB  
 Sinon → pas de forwarding

##### Sorties (2 bits chacune):

forward\_a : 00=rien, 01=depuis MEM, 10=depuis WB  
 forward\_b : 00=rien, 01=depuis MEM, 10=depuis WB

**7.11.5.3 Solution 2 : Le Stall (pour Load-Use)** Le forwarding ne résout pas tous les cas. Considérons :

LDR R1, [R2] ; Charge R1 depuis la mémoire  
 ADD R4, R1, R5 ; Utilise R1 immédiatement!

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
LDR R1	IF	ID	EX	MEM ← R1 disponible ici	
ADD R4,R1		IF	ID	EX ← Besoin de R1 ici!	

**Problème** : ADD a besoin de R1 dans son étage EX (cycle 4), mais LDR ne lit la mémoire qu'à l'étage MEM (aussi cycle 4). On ne peut pas faire de forwarding vers le passé !

**Solution** : Insérer une **bulle** (stall) pour retarder ADD d'un cycle.

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
LDR R1	IF	ID	EX	MEM	→ Forward possible!	
ADD R4,R1		IF	ID	STALL	EX ← Forward OK!	
Instr 3			IF	STALL	ID	EX

**Le HazardDetect** détecte ces situations :

**HazardDetect****Entrées:**

id\_rn, id\_rm : registres sources en ID  
 id\_rn\_used : Rn est utilisé par l'instruction?  
 id\_rm\_used : Rm est utilisé par l'instruction?  
 ex\_rd : registre destination en EX  
 ex\_mem\_read : instruction en EX est un LDR?

**Logique:**

Si EX est un load (ex\_mem\_read = 1)  
 ET ID utilise ce registre (id\_rn = ex\_rd ou id\_rm = ex\_rd)  
 → Déclencher un STALL

**Sortie:**

stall : 1 = bloquer IF et ID, insérer NOP en EX

**7.11.5.4 Aléa de Contrôle (Control Hazard)** Les branchements posent un autre problème :

```

BEQ label      ; Si égal, sauter à label
ADD R1, R2, R3 ; Cette instruction est-elle exécutée?
SUB R4, R5, R6 ; Et celle-ci?
label:
MOV R7, #42
  
```

	Cycle 1	Cycle 2	Cycle 3	Cycle 4
BEQ	IF	ID	EX ← On sait si on branche	
ADD		IF	ID	???
SUB			IF	???

**Problème** : Quand on exécute BEQ, on a déjà commencé à chercher les instructions suivantes ! Si le branchement est pris, ADD et SUB n'auraient jamais dû être exécutées.

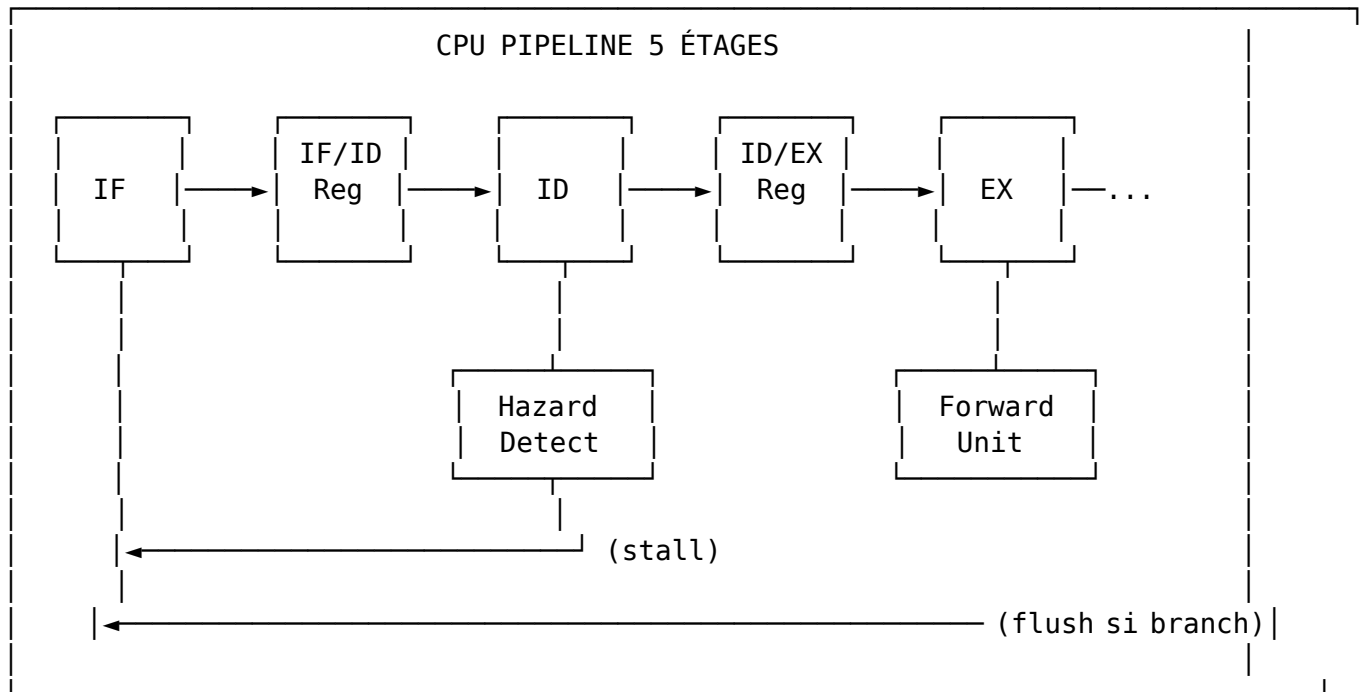
**Solution : Le Flush**

Si le branchement est pris, on **annule** les instructions qui n'auraient pas dû être chargées :

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
BEQ	IF	ID	EX	MEM	WB
ADD		IF	ID	FLUSH	
SUB			IF	FLUSH	
MOV R7				IF	ID ...

Le signal `flush` met les registres de pipeline à NOP (instruction qui ne fait rien).

### 7.11.6 Architecture Complète du CPU Pipeline



### 7.11.7 Exercices Pratiques : Projet 6

Le **Projet 6 : CPU Pipeline** vous permet de construire ces composants.

**7.11.7.1 Exercice 1 : IF\_ID\_Reg** **Objectif** : Implémenter le registre de pipeline IF/ID.

**Comportement** : 1. Sur `reset='1'` OU `flush='1'` : mettre l'instruction à NOP (0xE0000000) 2. Sur `stall='1'` : garder les valeurs actuelles 3. Sinon : capturer les nouvelles valeurs

**Squelette** :

```

architecture rtl of IF_ID_Reg is
    signal instr_reg : bits(31 downto 0);
    signal pc_plus4_reg : bits(31 downto 0);
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if (reset = '1') or (flush = '1') then

```



```

    instr_reg <= x"E0000000"; -- NOP
    pc_plus4_reg <= x"00000000";
  elsif stall = '0' then
    instr_reg <= if_instr;
    pc_plus4_reg <= if_pc_plus4;
  end if;
  -- Si stall='1', on ne fait rien (garde les valeurs)
end if;
end process;

id_instr <= instr_reg;
id_pc_plus4 <= pc_plus4_reg;
end architecture;

```

#### 7.11.7.2 Exercice 2 : HazardDetect Objectif : Détecter les aléas load-use.

##### Logique :

```

rn_hazard = ex_mem_read AND id_rn_used AND (id_rn = ex_rd)
rm_hazard = ex_mem_read AND id_rm_used AND (id_rm = ex_rd)
stall = rn_hazard OR rm_hazard

```

**En HDL** (attention : pas de `when...else`, utiliser la logique booléenne) :

```

architecture rtl of HazardDetect is
  signal rn_hazard : bit;
  signal rm_hazard : bit;
begin
  rn_hazard <= ex_mem_read and id_rn_used and (id_rn = ex_rd);
  rm_hazard <= ex_mem_read and id_rm_used and (id_rm = ex_rd);
  stall <= rn_hazard or rm_hazard;
end architecture;

```

#### 7.11.7.3 Exercice 3 : ForwardUnit Objectif : Générer les signaux de forwarding.

**Encodage** : - 00 : Pas de forwarding - 01 : Forward depuis MEM - 10 : Forward depuis WB

##### Logique :

```

mem_fwd_a = mem_reg_write AND (mem_rd = ex_rn)
wb_fwd_a = wb_reg_write AND (wb_rd = ex_rn) AND (NOT mem_fwd_a)
forward_a = wb_fwd_a & mem_fwd_a (concaténation de bits)

```

**En HDL :**

```

architecture rtl of ForwardUnit is
  signal mem_fwd_a, wb_fwd_a : bit;
  signal mem_fwd_b, wb_fwd_b : bit;
begin
  mem_fwd_a <= mem_reg_write and (mem_rd = ex_rn);
  wb_fwd_a <= wb_reg_write and (wb_rd = ex_rn) and (not mem_fwd_a);

```

```

mem_fwd_b <= mem_reg_write and (mem_rd = ex_rm);
wb_fwd_b <= wb_reg_write and (wb_rd = ex_rm) and (not mem_fwd_b);

forward_a <= wb_fwd_a & mem_fwd_a;
forward_b <= wb_fwd_b & mem_fwd_b;
end architecture;

```

**7.11.7.4 Exercice 4 : CPU\_Pipeline (Projet Final)** C'est le grand défi !  
Assembler tous les composants en un CPU pipeliné complet.

**Conseil :** L'implémentation de référence est dans `hdl_lib/05_cpu/CPU_Pipeline.hdl` (~450 lignes).

### 7.11.8 Comment Tester le CPU Pipeline HDL

Pour tester le CPU Pipeline en HDL, créez un script de test `.tst` :

**Fichier `CPU_Pipeline_test.tst` :**

```

-- Test du CPU Pipeline
load CPU_Pipeline.hdl;

-- Charger les composants requis
load IF_ID_Reg.hdl;
load ID_EX_Reg.hdl;
load EX_MEM_Reg.hdl;
load MEM_WB_Reg.hdl;
load HazardDetect.hdl;
load ForwardUnit.hdl;
load Decoder.hdl;
load Control.hdl;
load CondCheck.hdl;
load RegFile16.hdl;
load ALU32.hdl;
load Shifter32.hdl;
load Add32.hdl;
load PC.hdl;
load Mux32.hdl;

-- Initialiser
set reset 1;
tick; tock;
set reset 0;

-- Charger une instruction ADD R1, R0, #5 en mémoire
set instr_data 0x22100005; -- ADD R1, R0, #5

```

```
-- Exécuter plusieurs cycles (5 cycles pour traverser le pipeline)
tick; tock; -- IF
tick; tock; -- ID
tick; tock; -- EX
tick; tock; -- MEM
tick; tock; -- WB

-- Vérifier l'état
expect halted 0;
```

### Exécuter le test :

```
cargo run -p hdl_cli -- CPU_Pipeline_test.tst
```

### Structure des fichiers HDL pour le pipeline :

```
hdl_lib/05_cpu/
├── CPU_Pipeline.hdl      # CPU complet assemblé
├── IF_ID_Reg.hdl         # Registre pipeline IF→ID
├── ID_EX_Reg.hdl         # Registre pipeline ID→EX
├── EX_MEM_Reg.hdl        # Registre pipeline EX→MEM
├── MEM_WB_Reg.hdl        # Registre pipeline MEM→WB
├── HazardDetect.hdl      # Détection load-use hazards
├── ForwardUnit.hdl       # Bypass/forwarding
├── Decoder.hdl           # Décodage instruction
├── Control.hdl           # Signaux de contrôle
└── CondCheck.hdl         # Vérification conditions
```

### Observer le pipeline en action :

Pour voir les hazards et le forwarding, testez avec des instructions dépendantes :

```
-- Test de forwarding (EX→EX)
-- ADD R1, R0, #5      ; R1 = 5
-- ADD R2, R1, #3      ; R2 = R1 + 3 = 8 (forward depuis EX)
set instr_data 0x22100005; tick; tock;
set instr_data 0x22210003; tick; tock;
-- Le ForwardUnit détecte que R1 est produit par l'instruction précédente
-- et bypass la valeur directement sans attendre le writeback

-- Test de stall (load-use hazard)
-- LDR R1, [R0]        ; R1 = mem[R0]
-- ADD R2, R1, #3      ; R2 = R1 + 3 (doit attendre le load)
set instr_data 0x51100000; tick; tock;
set instr_data 0x22210003; tick; tock;
-- Le HazardDetect insère un stall car le LDR n'a pas encore
-- la valeur disponible au moment où ADD en a besoin
```

#### 7.11.9 Résumé : Pipeline vs Single-Cycle

Aspect	Single-Cycle	Pipeline
<b>Instructions en parallèle</b>	1	Jusqu'à 5
<b>Throughput</b>	1 instr / 5 unités de temps	1 instr / 1 unité de temps
<b>Complexité</b>	Simple	Plus complexe
<b>Aléas</b>	Aucun	Data hazards, Control hazards
<b>Composants supplémentaires</b>	Aucun	Registres pipeline, Hazard Detect, Forward Unit

#### 7.11.10 Pour Aller Encore Plus Loin

Les vrais processeurs modernes vont bien au-delà :

- **Superscalaire** : Plusieurs pipelines en parallèle
- **Exécution dans le désordre** : Réorganiser les instructions
- **Prédiction de branchement** : Deviner si un branchement sera pris
- **Cache** : Mémoire ultra-rapide proche du CPU

Mais ces concepts dépassent le cadre de ce livre. Le pipeline à 5 étages reste la base sur laquelle tout le reste est construit !

## 7.12 Le Lien avec la Suite

**Félicitations !** Vous venez de construire un ordinateur complet.

Ce CPU que vous avez construit peut maintenant : - Exécuter des programmes écrits en assembleur (Chapitre 6) - Exécuter des programmes compilés depuis C32 (Chapitre 7-8) - Faire tourner un système d'exploitation minimal (Chapitre 9)

### 7.12.1 Le parcours complet

Chapitre 1-5 : MATÉRIEL

NAND → Portes → ALU → Mémoire → CPU

↓

Chapitre 6-9 : LOGICIEL

Assembleur → Compilateur → Langage C32 → OS

À partir de maintenant, nous passons du côté **logiciel**. Le matériel est terminé !

## 7.13 Ce qu'il faut retenir

1. **Le CPU orchestre tout** : Fetch → Decode → Execute → Memory → Writeback

2. **Le décodeur analyse** : 32 bits → signaux individuels
3. **L'unité de contrôle décide** : Quels composants activer
4. **Les multiplexeurs routent** : Les données entre composants
5. **Les drapeaux permettent les conditions** : NZCV → CondCheck → ok/pas ok
6. **Single-cycle = simple** : Tout en un cycle (mais lent en vrai)

**Prochaine étape** : Au Chapitre 6, nous construirons l'**Assembleur** — le programme qui traduit le code assembleur en binaire exécutable par votre CPU.

**Conseil** : Si vous avez réussi le CPU, vous avez accompli quelque chose de remarquable. Prenez le temps de savourer : vous avez construit un ordinateur complet, de la porte NAND au processeur fonctionnel !

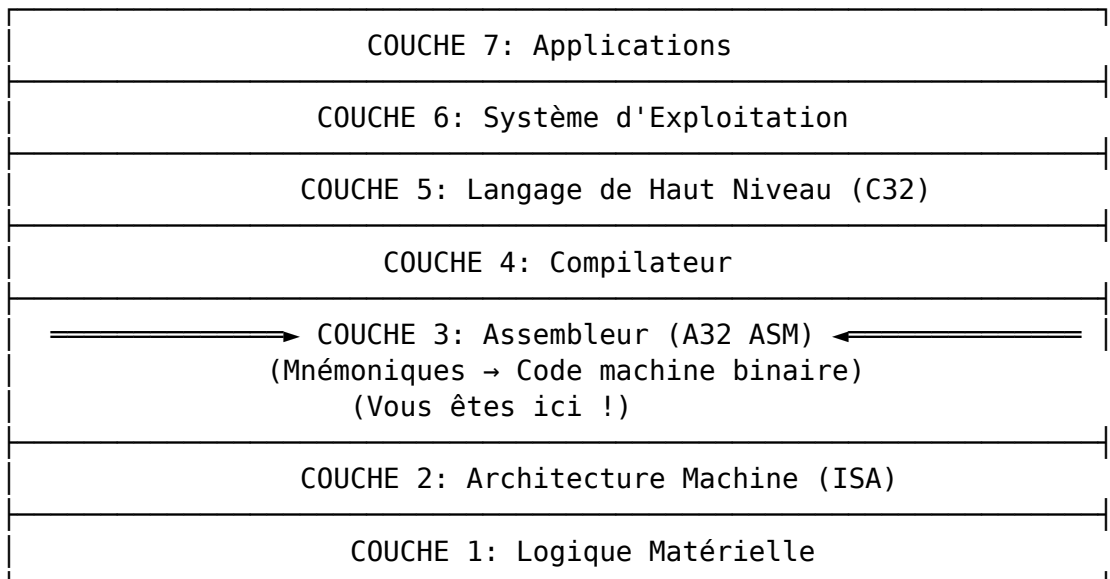
## 8 L'Assembleur

“Traduire, c’est trahir ?” — Pas ici.

Dans les chapitres précédents, nous avons conçu le matériel capable d’exécuter des instructions 32 bits. Mais écrire un programme en hexadécimal (comme `0xE2801001`) est extrêmement pénible et source d’erreurs.

L'**Assembleur** est l’outil logiciel qui fait le pont entre le programmeur et la machine. Il traduit un fichier texte contenant des mnémoniques lisibles (ex: `ADD R1, R1, #1`) en un fichier binaire exécutable par le CPU.

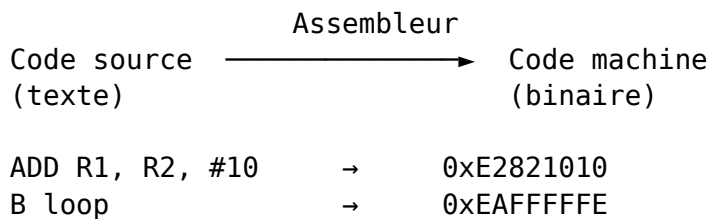
### 8.1 Où en sommes-nous ?



Nous entrons maintenant dans le monde du **logiciel** ! L'assembleur est le premier programme que nous construisons pour notre machine.

## 8.2 Le Rôle de l'Assembleur

### 8.2.1 Du texte au binaire



### 8.2.2 Les trois tâches de l'assembleur

1. **Analyse (Parsing)** : Lire le code source et comprendre les instructions, les opérandes, les labels.
2. **Résolution des Symboles** : Transformer les étiquettes (labels) comme `loop:` en adresses numériques.
3. **Encodage** : Transformer chaque instruction en son équivalent binaire de 32 bits selon la spécification de l'ISA.

## 8.3 La Stratégie des Deux Passes

### 8.3.1 Pourquoi deux passes ?

Regardez ce code :

```

B suite    ; Où est 'suite' ? On ne le sait pas encore !
MOV R0, #1
suite:
ADD R0, R0, #1

```

À la ligne 1, l'assembleur ne sait pas encore où est `suite`. C'est le problème des **références vers l'avant**.

### 8.3.2 Passe 1 : Construction de la Table des Symboles

L'assembleur parcourt le fichier et note l'adresse de chaque label :

```

Adresse 0x0000 : B suite           (4 octets)
Adresse 0x0004 : MOV R0, #1        (4 octets)
Adresse 0x0008 : suite:           ← On note : suite = 0x0008
Adresse 0x0008 : ADD R0, R0, #1

```

**Table des symboles** : { "suite": 0x00000008 }

### 8.3.3 Passe 2 : Génération du Code

L'assembleur reparcourt le fichier. Quand il voit `B suite`, il regarde dans sa table et génère l'offset correct.

`B suite` → offset =  $(0x0008 - 0x0000 - 8) / 4 = -2 \rightarrow 0xEFFFFFFE$

---

## 8.4 Sections et Directives

### 8.4.1 Les Sections

Un programme n'est pas fait que d'instructions. Il contient aussi des données.

Section	Contenu
<code>.text</code>	Le code (les instructions) — généralement en lecture seule
<code>.data</code>	Les variables globales initialisées
<code>.bss</code>	Les variables globales non initialisées (mises à zéro)

### 8.4.2 Les Directives

Les directives (commençant par `.`) guident l'assembleur :

Directive	Signification
<code>.text</code>	Début de la section code
<code>.data</code>	Début de la section données
<code>.global _start</code>	Exporte le symbole <code>_start</code>
<code>.word 123</code>	Réserve 4 octets avec la valeur 123
<code>.asciz "Hello"</code>	Chaîne terminée par un zéro
<code>.align 2</code>	Aligne sur un multiple de 4 octets
<code>.ltorg</code>	Force l'émission du literal pool

---

## 8.5 Exemple d'Encodage

Comment l'assembleur encode-t-il `ADD R1, R2, #10` ?

### 8.5.1 Étape 1 : Identifier l'instruction

- **Mnémonique** : `ADD` → opcode = 0011
- **Registres** : Rd = R1, Rn = R2
- **Immédiat** : #10

### 8.5.2 Étape 2 : Déterminer la classe

- Classe `001` car on utilise un immédiat

### 8.5.3 Étape 3 : Assembler les bits

31-28	27-25	24-21	20	19-16	15-12	11-0
Cond	Class	Op	S	Rn	Rd	Imm12
1110	001	0011	0	0010	0001	000000001010

= `0xE2821010`

## 8.6 La Gestion des Grandes Constantes

### 8.6.1 Le problème

Une instruction fait 32 bits. Un immédiat fait 12 bits maximum. Comment charger `0xDEADBEEF` (32 bits) dans un registre ?

### 8.6.2 La solution : Le Literal Pool

L'assembleur offre une syntaxe magique : `LDR R0, =0xDEADBEEF`

Ce n'est **pas** une vraie instruction LDR — c'est une **pseudo-instruction** que l'assembleur transforme :

1. La valeur `0xDEADBEEF` est stockée dans le **literal pool** (une zone de données après le code)
2. L'instruction est remplacée par `LDR R0, [PC, #offset]` qui va chercher la valeur

*; Code source*

```
LDR R0, =0xDEADBEEF
```

*; Ce que l'assembleur génère*

```
LDR R0, [PC, #8] ; Va chercher la valeur 8 octets plus loin
```

```
...
```

`literal_pool:`

```
.word 0xDEADBEEF ; La valeur est stockée ici
```

## 8.7 Exercices Pratiques

### 8.7.1 Exercices sur le Simulateur Web

Tous les exercices de la section **A32 Assembly** du simulateur web vous font pratiquer l'écriture d'assembleur. L'assembleur intégré traduit votre code en binaire automatiquement.



Catégorie	Exercices
Basique	Hello World, Addition, Soustraction, Logique
Contrôle	Conditions, Boucles, Multiplication, Fibonacci
Mémoire	Tableaux, Maximum Tableau, Fonctions
Graphique	Pixel, Ligne, Rectangle, Damier
Avancé	Recherche Dichotomique, Dégradé

### 8.7.2 Exercice manuel : Encodage

Traduisez ces instructions en binaire (32 bits) :

1. `MOV R0, #5`
2. `SUB R1, R1, #1`
3. `B -2` (saut de 2 instructions en arrière)

### 8.7.3 Exercice : Table des symboles

Calculez l'adresse de chaque label :

```
.text
start:
    MOV R0, #0
loop:
    CMP R0, #10
    BEQ end
    ADD R0, R0, #1
    B loop
end:
    HALT
```

### 8.7.4 Utilisation de l'outil CLI

```
# Assembler un fichier
cargo run -p a32_cli -- assemble mon_prog.s -o mon_prog.bin

# Examiner le binaire généré
hexdump -C mon_prog.bin
```

## 8.8 Ce qu'il faut retenir

1. **L'assembleur traduit** : Texte lisible → Binaire exécutable
2. **Deux passes** : D'abord collecter les symboles, puis générer le code
3. **Les directives organisent** : `.text`, `.data`, `.word`, `.asciz`
4. **Le literal pool résout** : Les constantes 32 bits via `LDR R0, =value`

5. **Un symbole = une adresse** : Les labels deviennent des nombres

**Prochaine étape** : Au Chapitre 7, nous construirons un **Compilateur** qui traduit du code C32 en assembleur. C'est l'étape suivante vers l'abstraction !

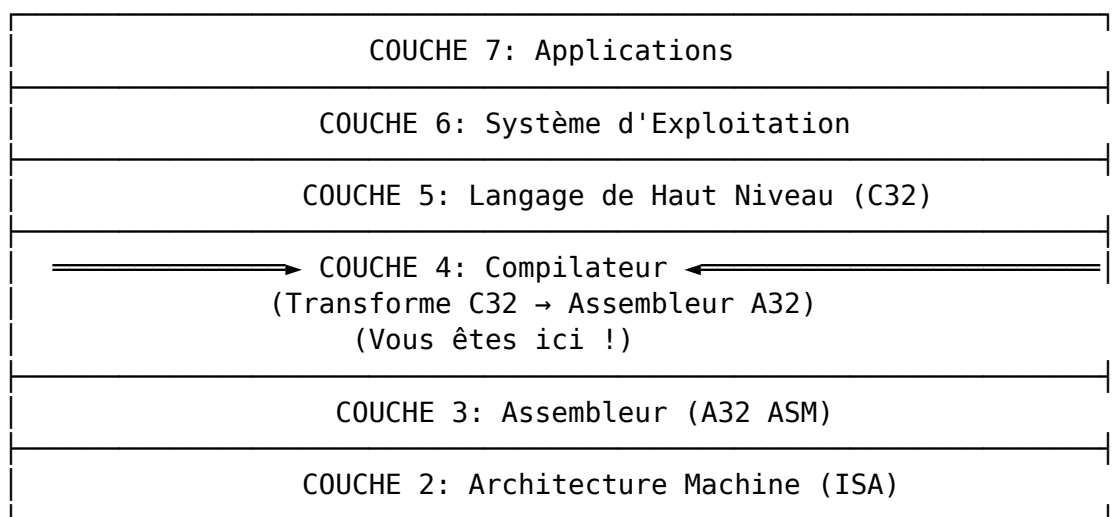
**Conseil** : Faites beaucoup d'exercices en assembleur. Plus vous serez à l'aise avec l'assembleur, mieux vous comprendrez ce que fait le compilateur.

## 9 Construction du Compilateur

"Pour comprendre la récursivité, il faut d'abord comprendre la récursivité."

Dans ce chapitre, nous allons construire le **pont** entre le langage de haut niveau (C32) et l'assembleur (A32). Le compilateur est l'outil qui permet aux humains d'écrire du code lisible tout en profitant de la vitesse du code machine.

### 9.1 Où en sommes-nous ?



Le compilateur est le **traducteur automatique** qui transforme du code lisible par les humains en code exécutable par la machine.

### 9.2 Le Rôle du Compilateur

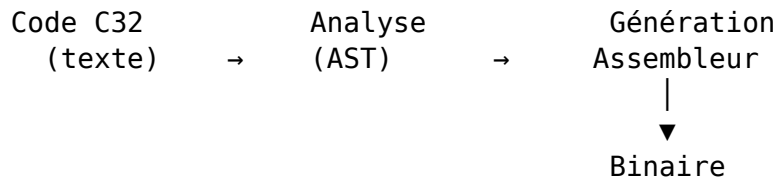
#### 9.2.1 Pourquoi un compilateur ?

Assembleur	C32 (haut niveau)
ADD R0, R0, #1	x = x + 1;
Gestion manuelle des registres	Variables nommées

Assembleur	C32 (haut niveau)
Sauts et labels	if, while, for
Appels manuels avec conventions	return fonction();

Le compilateur traduit le second en premier, automatiquement.

### 9.2.2 Les étapes de compilation



1. **Analyse lexicale** : Découpe le texte en tokens ( `int` , `x` , `=` , `5` , `;` )
2. **Analyse syntaxique** : Construit un arbre (AST) représentant la structure
3. **Analyse sémantique** : Vérifie les types, les portées, etc.
4. **Génération de code** : Produit l'assembleur équivalent

## 9.3 Les Phases du Compilateur

### 9.3.1 Phase 1 : Lexer (Analyse Lexicale)

Le lexer transforme le flux de caractères en tokens :

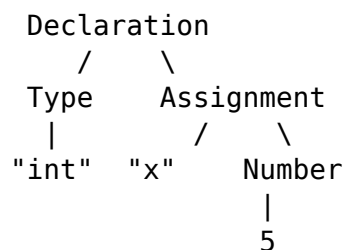
```
int x = 5;
```

Devient :

```
[INT] [ID:"x"] [EQUAL] [NUMBER:5] [SEMICOLON]
```

### 9.3.2 Phase 2 : Parser (Analyse Syntaxique)

Le parser construit un **AST** (Abstract Syntax Tree) :



### 9.3.3 Phase 3 : Génération de Code

Le générateur parcourt l'AST et produit l'assembleur :

```

; int x = 5;
MOV R0, #5
STR R0, [SP, #-4]!    ; Push x sur la pile

```

---

## 9.4 Compilation des Structures de Contrôle

### 9.4.1 Variables locales

Les variables locales vivent sur la **pile** :

```

int a = 10;
int b = 20;

; Prologue
SUB SP, SP, #8    ; Réserve 8 octets pour a et b

; a = 10
MOV R0, #10
STR R0, [SP, #4]   ; a est à SP+4

; b = 20
MOV R0, #20
STR R0, [SP, #0]   ; b est à SP+0

```

### 9.4.2 Expressions

Pour `a + b * 2` :

```

; Évaluation de b * 2
LDR R0, [SP, #0]    ; R0 = b
MOV R1, #2
MUL R0, R0, R1      ; R0 = b * 2

; Évaluation de a + (b * 2)
LDR R1, [SP, #4]    ; R1 = a
ADD R0, R1, R0      ; R0 = a + b*2

```

### 9.4.3 If / Else

```

if (x > 0) {
    y = 1;
} else {
    y = 0;
}

LDR R0, [SP, #x_offset]
CMP R0, #0
BLE else_label

```

```

; Then branch
MOV R0, #1
STR R0, [SP, #y_offset]
B endif_label

else_label:
; Else branch
MOV R0, #0
STR R0, [SP, #y_offset]

endif_label:

```

#### 9.4.4 Boucle While

```

while (i < 10) {
    i = i + 1;
}

while_start:
    LDR R0, [SP, #i_offset]
    CMP R0, #10
    BGE while_end

; Corps de la boucle
    ADD R0, R0, #1
    STR R0, [SP, #i_offset]

    B while_start

while_end:

```

#### 9.4.5 Boucle For

`for (init; cond; incr) { body }` est équivalent à :

```

init;
while (cond) {
    body;
    incr;
}

```

## 9.5 Compilation des Fonctions

### 9.5.1 Convention d'appel

Comment passe-t-on les arguments ? Comment retourne-t-on une valeur ?

Registre	Rôle
R0-R3	Arguments 1-4, valeur de retour en R0
R4-R11	Sauvegardés par l'appelé (callee-saved)
R13 (SP)	Pointeur de pile
R14 (LR)	Adresse de retour

### 9.5.2 Prologue et Épilogue

```

int add(int a, int b) {
    return a + b;
}

add:
    ; Prologue
    SUB SP, SP, #4          ; Place pour LR
    STR LR, [SP]            ; Sauvegarde LR

    ; Corps : a est dans R0, b dans R1
    ADD R0, R0, R1          ; Résultat dans R0

    ; Épilogue
    LDR LR, [SP]            ; Restaure LR
    ADD SP, SP, #4
    MOV PC, LR              ; Retour

```

### 9.5.3 Appel de fonction

```

result = add(5, 3);

    MOV R0, #5              ; Premier argument
    MOV R1, #3              ; Deuxième argument
    BL add                  ; Appel (sauve PC+4 dans LR)
    STR R0, [SP, #result_offset] ; Sauve le résultat

```

## 9.6 Le Compilateur C32

Le compilateur C32 du projet Codex ( `c32_cli` ) implémente toutes ces transformations.

### 9.6.1 Utilisation

```

# Compiler un fichier C32 en assembleur
cargo run -p c32_cli -- mon_fichier.c -o mon_fichier.s

# Compiler directement en binaire
cargo run -p c32_cli -- mon_fichier.c -o mon_fichier.bin

```

### 9.6.2 Exemple complet

```
// fibonacci.c
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}

int main() {
    return fib(10);
}
```

Produit de l'assembleur avec : - Gestion automatique de la pile - Appels récursifs - Sauvegarde/restauration des registres

## 9.7 Construisez Votre Propre Compilateur !

Le simulateur web contient une section **Compilateur: Construction** avec **18 exercices progressifs** organisés en 7 phases. Vous construirez un mini-compilateur qui génère du vrai code assembleur A32.

### 9.7.1 Phase 1 : Lexer (Analyse Lexicale)

Le lexer transforme le texte en tokens.

Exercice	Description
1.1 Reconnaître un Chiffre	Implémenter <code>is_digit(c)</code> pour détecter '0'-'9'
1.2 Lire un Nombre	Parser un entier depuis une chaîne
1.3 Compter les Tokens	Compter les tokens dans <code>"12 + 34 * 5"</code>

### 9.7.2 Phase 2 : Parser (Analyse Syntaxique)

Le parser construit une représentation structurée et évalue les expressions.

Exercice	Description
2.1 Précédence des Opérateurs	Déterminer la priorité ( <code>*</code> > <code>+</code> )
2.2 Évaluer une Opération	Évaluer <code>3 + 4</code> ou <code>6 * 7</code>
2.3 Parser avec Précédence	Descente récursive pour <code>2 + 3 * 4</code>
2.4 Parenthèses	Supporter <code>(2 + 3) * 4</code>

### 9.7.3 Phase 3 : Émission ASM (Génération de Code)

Générer des instructions A32 sous forme de chaînes.

Exercice	Description
3.1 Générer MOV	Produire "MOV R0, #42"
3.2 Opération Binaire	Mapper + → ADD, * → MUL
3.3 Comparaison	Générer CMP et codes de condition

#### 9.7.4 Phase 4 : CodeGen Expressions

Générer du code A32 complet pour des expressions.

Exercice	Description
4.1 Constante → A32	Générer code pour charger une constante
4.2 Addition → A32	a + b → MOV R0, #a / MOV R1, #b / ADD R0, R0, R1
4.3 Expression → A32	Expression complète avec précedence

#### 9.7.5 Phase 5 : Structures de Contrôle

Générer du code pour if/else et while.

Exercice	Description
5.1 If/Else → A32	Générer les sauts conditionnels et labels
5.2 While → A32	Générer les boucles avec labels

#### 9.7.6 Phase 6 : Fonctions

Gérer les appels de fonction et la pile.

Exercice	Description
6.1 Prologue/Épilogue	Sauvegarder LR, réserver la pile
6.2 Appel de Fonction	Passer les arguments, appeler avec BL

#### 9.7.7 Phase 7 : Projet Final

Exercice	Description
7.1 Mini-Compilateur Complet	Compiler une expression en A32 exécutable

Le projet final combine toutes les phases : lexer → parser → codegen pour produire du code assembleur A32 fonctionnel.



### 9.7.8 Techniques Clés

Ces exercices utilisent la technique de **descente récursive** : - `parse_expr()` gère + et - (basse priorité) - `parse_term()` gère \* et / (haute priorité) - `parse_factor()` gère les nombres et les parenthèses

## 9.8 Exercices Pratiques

### 9.8.1 Exercices sur le Simulateur Web

La section **C32** du simulateur web vous permet de compiler et exécuter du C32.

Catégorie	Exercices
Bases	Variables, Expressions, Modulo, Incrémentation
Contrôle	Conditions, Else-If, Maximum de 3
Boucles	For, While, Imbriquées, Multiplication
Fonctions	Appels, Paramètres, Valeur Absolue, Min/Max
Tableaux	Accès, Maximum, Comptage
Pointeurs	Adresses, Swap, Tableaux via pointeurs
Récursion	Factorielle, Fibonacci, PGCD
Algorithmes	Tri à Bulles, Recherche Binaire

### 9.8.2 Exercice : Traduire manuellement

Traduisez ce code C32 en assembleur à la main :

```
int sum = 0;
for (int i = 1; i <= 10; i = i + 1) {
    sum = sum + i;
}
```

Comparez avec la sortie du compilateur !

## 9.9 Ce qu'il faut retenir

1. **Le compilateur traduit** : C32 (lisible) → Assembleur (exécutable)
2. **Trois phases** : Lexer → Parser → Générateur de code
3. **Les variables locales vivent sur la pile** : Accès via `[SP, #offset]`
4. **Les structures de contrôle deviennent des sauts** : `if` → `CMP + B`
5. **Les fonctions suivent une convention** : Arguments en R0-R3, retour en R0

**Prochaine étape** : Au Chapitre 8, nous explorerons le langage C32 en détail — sa syntaxe, ses types, et ses possibilités.

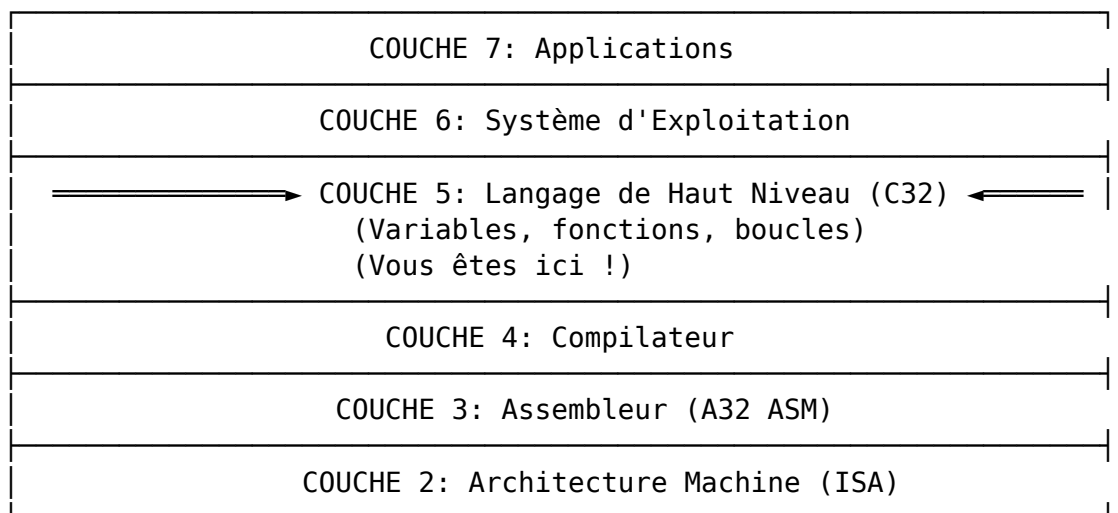
**Conseil** : Pour vraiment comprendre le compilateur, écrivez du C32 et regardez l'assembleur généré. Cherchez à prédire ce que le compilateur va produire !

## 10 Langage de Haut Niveau (C32)

“Le logiciel est l'esprit qui anime la machine.”

Jusqu'à présent, nous avons construit le matériel et appris à lui parler en assembleur. Mais écrire des applications complexes en assembleur est laborieux. C'est ici qu'intervient le **C32** — un langage de haut niveau qui vous permet de vous concentrer sur la **logique** de votre programme.

### 10.1 Où en sommes-nous ?



Le C32 est un sous-ensemble du langage C. Si vous connaissez le C, le Java ou le C++, vous vous sentirez chez vous.

### 10.2 Pourquoi un Langage de Haut Niveau ?

#### 10.2.1 Le problème de l'assembleur

Assembleur

```
MOV R0, #0
MOV R1, #1
loop:
    CMP R1, #10
    BGT done
```

C32

```
int sum = 0;
for (int i = 1; i <= 10; i = i + 1) {
    sum = sum + i;
}
```

```

ADD R0, R0, R1
ADD R1, R1, #1
B loop
done:

```

Le C32 est : - **Plus lisible** : Variables nommées, structures de contrôle - **Plus maintenable** : Moins de code, moins de bugs - **Portable** : Le même code peut cibler différentes architectures

### 10.2.2 L'abstraction

```

[ Pensée Humaine ] → "Calculer la moyenne"
      ↓
[ Langage C32 ]    → sum = sum + tab[i];
      ↓
[ Assembleur A32 ] → LDR R0, [R1, R2]; ADD R3, R3, R0...
      ↓
[ Code Machine ]   → 0xE0833000...

```

## 10.3 Spécification du Langage C32

### 10.3.1 Les Types de Données

Type	Taille	Description
int	32 bits	Entier signé (complément à 2)
uint	32 bits	Entier non-signé
char	8 bits	Caractère ASCII
bool	1 bit	true ou false
void	—	Pour les fonctions sans retour
type*	32 bits	Pointeur (adresse mémoire)

### 10.3.2 Variables

```

int x = 42;           // Variable globale ou locale
int tab[10];          // Tableau de 10 entiers
int* p = &x;          // Pointeur vers x

```

### 10.3.3 Portée des variables

- **Globales** : Déclarées hors des fonctions, accessibles partout
- **Locales** : Déclarées dans une fonction, vivent sur la pile

## 10.4 Opérateurs

### 10.4.1 Arithmétiques

Opérateur	Signification
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste)

### 10.4.2 Comparaison

Opérateur	Signification
==	Égal
!=	Différent
<	Inférieur
>	Supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal

### 10.4.3 Logiques

Opérateur	Signification
&&	ET logique
	OU logique
!	NON logique

### 10.4.4 Binaires

Opérateur	Signification
&	ET bit à bit
	OU bit à bit
^	XOR bit à bit
~	Inversion
<<	Décalage gauche
>>	Décalage droite

## 10.5 Structures de Contrôle

### 10.5.1 If / Else

```
if (score > 100) {  
    win();  
} else if (score > 50) {  
    try_again();  
} else {  
    game_over();  
}
```

### 10.5.2 While

```
while (x < 10) {  
    x = x + 1;  
}
```

### 10.5.3 For

```
for (int i = 0; i < 10; i = i + 1) {  
    sum = sum + i;  
}
```

### 10.5.4 Do-While

```
do {  
    x = x - 1;  
} while (x > 0);
```

---

## 10.6 Fonctions

### 10.6.1 Définition

```
int add(int a, int b) {  
    return a + b;  
}
```

```
void greet() {  
    putchar('H');  
    putchar('i');  
}
```

### 10.6.2 Appel

```
int result = add(5, 3);  
greet();
```

### 10.6.3 Récursion

```
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

---

## 10.7 Pointeurs et Tableaux

### 10.7.1 Pointeurs

Un pointeur contient une **adresse mémoire** :

```
int x = 42;
int* p = &x;    // p contient l'adresse de x
*p = 100;       // x vaut maintenant 100
```

### 10.7.2 Tableaux

Un tableau est une suite de valeurs consécutives en mémoire :

```
int scores[5];
scores[0] = 10;
scores[4] = 50;
```

### 10.7.3 Lien entre pointeurs et tableaux

```
int tab[10];
int* p = tab;    // Équivalent à &tab[0]
p[3] = 42;       // Équivalent à tab[3] = 42
*(p + 3) = 42;   // Même chose !
```

---

## 10.8 Accès au Matériel (MMIO)

### 10.8.1 L'écran

```
// L'écran commence à 0x00400000
// 320x240 pixels, 1 bit par pixel

void set_pixel(int x, int y) {
    uint* screen = (uint*)0x00400000;
    int offset = y * 10 + (x / 32);
    uint mask = 1 << (31 - (x % 32));
    screen[offset] = screen[offset] | mask;
}

void clear_screen() {
```

```

uint* screen = (uint*)0x00400000;
for (int i = 0; i < 2400; i = i + 1) {
    screen[i] = 0;
}
}

```

### 10.8.2 Le clavier

*// Le clavier est à 0x00402600*

```

int get_key() {
    int* keyboard = (int*)0x00402600;
    return *keyboard;
}

```

```

void wait_key() {
    while (get_key() == 0) {
        // Attendre
    }
}

```

## 10.9 Exemple Complet

*// Programme qui dessine un rectangle et attend une touche*

```

extern void putchar(char c);

int main() {
    // Dessiner un rectangle 10x5 à la position (20, 30)
    uint* screen = (uint*)0x00400000;

    for (int y = 30; y < 35; y = y + 1) {
        for (int x = 20; x < 30; x = x + 1) {
            int offset = y * 10 + (x / 32);
            uint mask = 1 << (31 - (x % 32));
            screen[offset] = screen[offset] | mask;
        }
    }

    // Afficher un message
    putchar('D');
    putchar('o');
    putchar('n');
    putchar('e');
    putchar('!');
}

```

```

// Attendre une touche
int* kbd = (int*)0x00402600;
while (*kbd == 0) {}

return 0;
}

```

## 10.10 Exercices Pratiques

### 10.10.1 Exercices sur le Simulateur Web

La section **C32** contient de nombreux exercices progressifs :

Catégorie	Exercices clés
<b>Bases</b>	Variables, Expressions, Modulo
<b>Contrôle</b>	Conditions, Else-If, Opérateurs Logiques
<b>Boucles</b>	For, While, Imbriquées
<b>Fonctions</b>	Paramètres, Valeur Absolue, Min/Max
<b>Tableaux</b>	Accès, Maximum, Comptage
<b>Pointeurs</b>	Adresses, Swap
<b>Récursion</b>	Factorielle, Fibonacci, PGCD
<b>Algorithmes</b>	Tri à Bulles, Recherche Binaire
<b>Graphique</b>	Pixel, Ligne, Rectangle, Damier

### 10.10.2 Défis suggérés

1. **Hello World** : Affichez votre nom à l'écran
2. **Jeu de devinette** : Le programme choisit un nombre, l'utilisateur devine
3. **Calculatrice** : Lisez deux nombres et affichez leur somme

## 10.11 Les Structures (Structs)

Le C32 supporte les structures (structs) pour regrouper plusieurs variables :

### 10.11.1 Définition d'une structure

```

struct Point {
    int x;
    int y;
};

```



### 10.11.2 Utilisation

```

struct Point p;      // Déclarer une variable de type struct
p.x = 10;             // Accéder aux champs avec .
p.y = 20;

```

### 10.11.3 Pointeur vers structure

```

struct Point *ptr = &p;
ptr->x = 30;           // Accéder via pointeur avec ->
ptr->y = 40;

```

### 10.11.4 Exemple complet

```

struct Point { int x; int y; };

int distance_sq(struct Point *p) {
    return p->x * p->x + p->y * p->y;
}

int main() {
    struct Point p;
    p.x = 3;
    p.y = 4;
    return distance_sq(&p); // Retourne 25
}

```

### 10.11.5 Structures imbriquées

```

struct Point { int x; int y; };
struct Rectangle {
    struct Point corner;
    int width;
    int height;
};

int main() {
    struct Rectangle r;
    r.corner.x = 0;
    r.corner.y = 0;
    r.width = 100;
    r.height = 50;
    return r.width * r.height; // Retourne 5000
}

```

---

## 10.12 Limitations du C32

Pour rester simple et pédagogique, le C32 a quelques limites :

Fonctionnalité	État
<code>struct</code>	Supporté
<code>float</code> , <code>double</code>	Non supporté
<code>malloc</code> / <code>free</code>	Via OS uniquement
Chaînes de caractères	Basique
Préprocesseur	Minimal

## 10.13 Ce qu'il faut retenir

1. **C32 simplifie la programmation** : Variables nommées, structures de contrôle
2. **Les types de base** : `int`, `char`, `bool`, `void`, pointeurs
3. **Pointeurs = adresses** : Accès direct à la mémoire
4. **MMIO** : Écran à 0x00400000, clavier à 0x00402600
5. **Fonctions** : Modularité et réutilisation du code

**Prochaine étape** : Au Chapitre 9, nous construirons un **Système d'Exploitation** minimal — gestion mémoire, graphiques, entrées/sorties.

**Conseil** : Le C32 est proche du C. Si vous voulez aller plus loin, apprenez le C — c'est le langage de base de Linux, Windows, et de presque tous les systèmes embarqués !

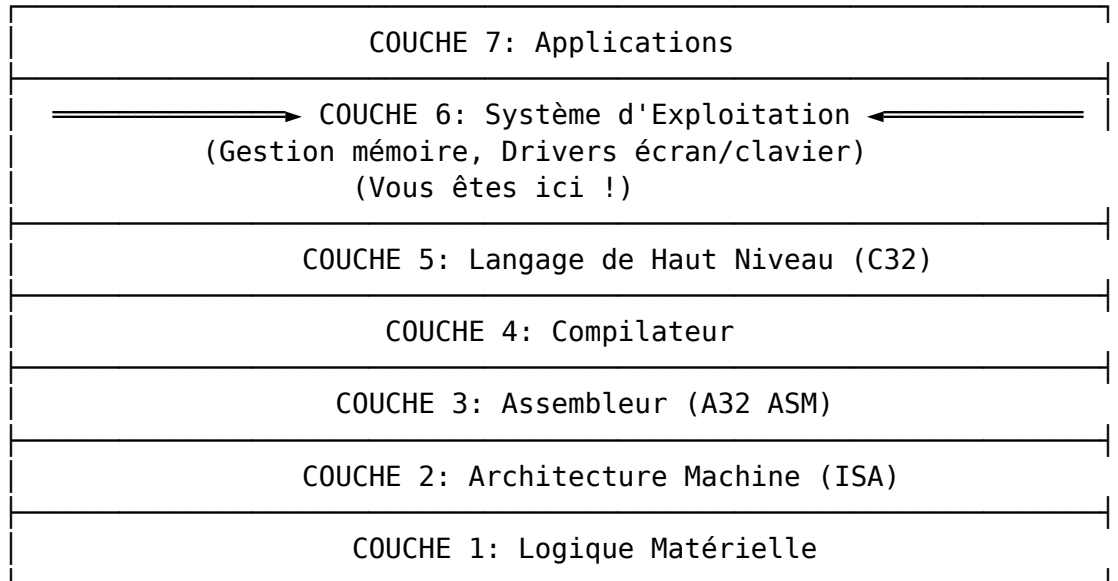
## 11 Système d'Exploitation

“Un OS est ce qui reste quand on a enlevé tout ce qui est utile.” — Ken Thompson

Félicitations ! Vous avez construit le matériel, l'assembleur et le compilateur. Votre machine Codex est fonctionnelle. Mais pour l'instant, chaque programmeur doit réinventer la roue : comment dessiner un cercle ? comment lire une chaîne de caractères ?

Dans ce dernier chapitre, nous allons construire une **Bibliothèque Système** qui simplifie l'accès au matériel.

## 11.1 Où en sommes-nous ?



C'est le **sommet de la pyramide** logicielle ! L'OS cache la complexité du matériel et offre des services de haut niveau aux applications.

## 11.2 Qu'est-ce qu'un Système d'Exploitation ?

### 11.2.1 La hiérarchie logicielle

Application Utilise l'API de l'OS	(Ex: Snake, Calculatrice)
Bibliothèque Système Cache les détails	(malloc, draw_line, printf)
Matériel (CPU)	(Registres, MMIO, Mémoire)

### 11.2.2 Ce que fait un OS

Fonction	Description
<b>Gestion mémoire</b>	Allouer/libérer de la mémoire dynamiquement
<b>Graphiques</b>	Dessiner des formes, du texte
<b>Entrées/Sorties</b>	Lire le clavier, afficher à l'écran
<b>Fichiers</b>	Sauvegarder/charger des données (avancé)
<b>Multitâche</b>	Exécuter plusieurs programmes (avancé)

Notre OS Codex implémente les trois premiers points.

## 11.3 Gestion de la Mémoire (Le Tas / Heap)

### 11.3.1 Le problème

Jusqu'à présent, nous utilisons : - **Variables globales** : Taille fixée à la compilation - **Variables locales** : Sur la pile, libérées automatiquement

Mais que faire si on veut allouer une taille **inconnue à l'avance** ?

### 11.3.2 L'allocateur "Bump" (Simple)

L'allocateur le plus simple : un pointeur qui avance à chaque allocation.

```
char* heap_ptr = (char*)HEAP_START;

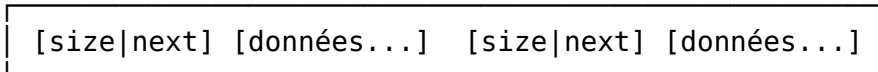
char* malloc(int size) {
    char* result = heap_ptr;
    heap_ptr = heap_ptr + size;
    return result;
}

void free(char* ptr) {
    // Ne fait rien ! La mémoire n'est jamais récupérée.
}
```

**Avantage** : Très simple, très rapide. **Inconvénient** : On ne peut pas réutiliser la mémoire libérée.

### 11.3.3 L'allocateur par Liste Chaînée (Avancé)

Pour réutiliser la mémoire, chaque bloc contient : - Sa taille - Un pointeur vers le bloc libre suivant



Quand on libère un bloc, on l'ajoute à la liste des blocs libres. Quand on alloue, on cherche un bloc de taille suffisante.

## 11.4 Bibliothèque Graphique

### 11.4.1 Le problème

Pour allumer un pixel, il faut : 1. Calculer l'adresse de l'octet 2. Calculer la position du bit 3. Faire un OR pour allumer (ou AND + NOT pour éteindre)

C'est fastidieux et source d'erreurs.

### 11.4.2 Les fonctions graphiques

```
void screen_init();
void screen_clear();
void screen_set_color(int color);

void screen_draw_pixel(int x, int y);
void screen_draw_line(int x1, int y1, int x2, int y2);
void screen_draw_rect(int x, int y, int w, int h);
void screen_draw_circle(int cx, int cy, int r);
void screen_print(char* text, int x, int y);
```

### 11.4.3 L'algorithme de Bresenham

Pour dessiner des lignes droites avec uniquement des additions et des comparaisons :

```
void screen_draw_line(int x1, int y1, int x2, int y2) {
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int sx = x1 < x2 ? 1 : -1;
    int sy = y1 < y2 ? 1 : -1;
    int err = dx - dy;

    while (1) {
        screen_draw_pixel(x1, y1);
        if (x1 == x2 && y1 == y2) break;
        int e2 = 2 * err;
        if (e2 > -dy) { err = err - dy; x1 = x1 + sx; }
        if (e2 < dx) { err = err + dx; y1 = y1 + sy; }
    }
}
```

Pas de multiplication, pas de division — uniquement des opérations que notre CPU fait rapidement !

---

## 11.5 Entrées / Sorties

### 11.5.1 Printf simplifié

```
void putchar(char c) {
    // Appel système SVC pour afficher un caractère
}

void puts(char* s) {
    while (*s != 0) {
```

```

        putchar(*s);
        s = s + 1;
    }
}

void print_int(int n) {
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n >= 10) {
        print_int(n / 10);
    }
    putchar('0' + (n % 10));
}

```

### 11.5.2 Lecture du clavier

```

int keyboard_read() {
    int* kbd = (int*)0x00402600;
    return *kbd;
}

int keyboard_wait() {
    int key;
    while ((key = keyboard_read()) == 0) {
        // Attendre
    }
    return key;
}

```

---

## 11.6 Interruptions et Timer (Concepts)

### 11.6.1 Le problème du polling

Dans notre approche actuelle, le CPU vérifie constamment le clavier :

```
while (keyboard_read() == 0) {} // CPU occupe à ne rien faire !
```

### 11.6.2 Les interruptions

Avec les interruptions, le matériel **signale** au CPU qu'un événement s'est produit :

1. Le CPU exécute le programme
2. Une touche est pressée → Le matériel déclenche une **interruption**
3. Le CPU s'arrête et saute vers le **handler d'interruption**
4. Le handler traite l'événement
5. Le CPU reprend le programme là où il s'était arrêté

C'est la base du multitâche et des systèmes réactifs !

## 11.7 Applications Démo

Le répertoire `demos/` contient des exemples complets :

Démo	Description
01_hello	Hello World classique
02_counter	Compteur avec affichage
03_graphics	Dessins géométriques
04_snake	Jeu du serpent complet !
05_shell	Interface en ligne de commande

### 11.7.1 Compiler et exécuter une démo

# Compiler

```
cargo run -p c32_cli -- demos/04_snake/main.c -o snake.bin
```

# Exécuter

```
cargo run -p a32_runner -- snake.bin
```

Ou utilisez le **Simulateur Web** pour une expérience visuelle !

## 11.8 Exercices Pratiques

### 11.8.1 Exercices sur le Simulateur Web

La section **OS** contient des exercices interactifs :

Exercice	Description
Calculatrice	Calculatrice interactive avec clavier
Variables	Shell pour définir des variables
Timer	Affichage d'un compteur animé
Scheduler	Simulation d'un ordonnanceur
Projet Mini-OS	Shell multi-applications
Task Manager	Gestionnaire de tâches visuel

### 11.8.2 Défis suggérés

1. **Étendre l'OS** : Ajoutez `screen_draw_triangle()` en utilisant trois appels à `screen_draw_line()`.

2. **Gestion mémoire** : Testez l'allocateur avec plusieurs `malloc()` et observez le comportement.
  3. **Projet final** : Créez votre propre application dans  `demos/` en utilisant tout ce que vous avez appris !
- 

## 11.9 Le Parcours Complet

Vous avez parcouru tout le chemin :

Chapitre 0 : Introduction

↓

Chapitre 1-5 : MATÉRIEL

NAND → Portes → ALU → Mémoire → CPU

↓

Chapitre 6-9 : LOGICIEL

Assembleur → Compilateur → C32 → OS

↓

Applications : Jeux, Calculatrices, Shell...

Vous comprenez maintenant que l'ordinateur n'est pas une boîte magique, mais une **pyramide d'abstractions** magnifiquement ordonnées.

---

## 11.10 Ce qu'il faut retenir

1. **L'OS cache le matériel** : `draw_circle()` au lieu de manipuler des bits
  2. **Gestion mémoire dynamique** : `malloc()` et `free()`
  3. **Bibliothèque graphique** : Lignes, rectangles, cercles, texte
  4. **Entrées/Sorties** : `putchar()` , `keyboard_read()`
  5. **Interruptions** : Le matériel peut signaler des événements au CPU
- 

## 11.11 Félicitations !

Vous avez parcouru tout le chemin, de la **porte NAND** au **système d'exploitation**. Vous comprenez maintenant :

- Comment les bits deviennent des calculs (ALU)
- Comment les calculs deviennent de la mémoire (RAM)
- Comment la mémoire devient un programme (CPU)
- Comment un programme devient une application (Compilateur + OS)

Quand vous verrez du code s'exécuter, vous saurez **exactement** ce qui se passe dans la machine. Ce n'est plus de la magie — c'est de l'ingénierie que vous maîtrisez.



---

**Et maintenant ?**

- Apprenez le C pour approfondir la programmation système
- Étudiez le noyau Linux pour voir un vrai OS
- Explorez les architectures ARM/RISC-V modernes
- Construisez vos propres projets sur la base de Codex !

## 12 Annexe : Tous les Exercices

Cette annexe contient les enonces de tous les exercices disponibles sur le simulateur web.

---

### 12.1 A. Exercices HDL (Portes Logiques)

Ces exercices construisent progressivement un ordinateur a partir de la porte NAND.

#### 12.1.1 Projet 1 : Portes de Base

Exercice	Description
Inv	Inverseur (NOT) : $\text{Inv}(a) = \text{Nand}(a, a)$
And2	Porte AND : $\text{And2}(a, b) = \text{Inv}(\text{Nand}(a, b))$
Or2	Porte OR : $\text{Or2}(a, b) = \text{Nand}(\text{Inv}(a), \text{Inv}(b))$
Xor2	Porte XOR : $\text{Xor2}(a, b) = \text{Or2}(\text{And2}(a, \text{Inv}(b)), \text{And2}(\text{Inv}(a), b))$
Mux	Multiplexeur : si $\text{sel}=0$ alors $y=a$ sinon $y=b$
DMux	Demultiplexeur : distribue $x$ vers $a$ ou $b$ selon $\text{sel}$

#### 12.1.2 Projet 2 : Arithmetique

Exercice	Description
HalfAdder	Demi-additionneur : $\text{sum} = a \text{ XOR } b$ , $\text{carry} = a \text{ AND } b$
FullAdder	Additionneur complet avec retenue entrante
Add32	Additionneur 32 bits (ripple carry)
Inc32	Incrementeur : ajoute 1 a l'entree
Alu32	ALU 32 bits avec 6 operations

#### 12.1.3 Projet 3 : Memoire

Exercice	Description
DFF	D Flip-Flop (fourni)
Bit	Registre 1 bit avec load
Register	Registre 32 bits
RAM8	8 registres avec adressage 3 bits
RAM64	64 registres avec adressage 6 bits
PC	Compteur de programme avec inc/load/reset

#### 12.1.4 Projet 5 : CPU

Exercice	Description
CPU	Processeur complet A32

#### 12.1.5 Projet 6 : CPU Pipeline (Avance)

Ces exercices construisent un CPU pipeline 5 etages avec gestion des aleas.

Exercice	Description
IF_ID_Reg	Registre pipeline IF/ID avec stall et flush
HazardDetect	Detection des aleas load-use
ForwardUnit	Bypass des donnees (forwarding)
CPU_Pipeline	CPU pipeline 5 etages complet

#### 12.1.6 Projet 7 : Cache L1

Ces exercices implementent un cache memoire direct-mapped.

Exercice	Description
CacheLine	Ligne de cache (valid, dirty, tag, data)
TagCompare	Comparateur de tags pour detecter hit/miss
WordSelect	Selecteur de mot dans une ligne de 128 bits
CacheController	Machine a etats (IDLE, FETCH, WRITEBACK)

## 12.2 B. Exercices Assembleur A32

Ces exercices enseignent la programmation en assembleur A32.

### 12.2.1 Bases

Exercice	Objectif	Resultat
Hello World	Charger 42 dans R0	R0 = 42
Addition	Calculer 15 + 27	R0 = 42
Soustraction	Calculer 100 - 58	R0 = 42
Logique	0xFF AND 0x0F	R0 = 15
Doubler	21 * 2 sans MUL	R0 = 42

### 12.2.2 Controle de Flux

Exercice	Objectif	Resultat
Conditions	Maximum de 25 et 17	R0 = 25
Valeur Absolue	Calculer	-42
Boucles	Somme 1 + 2 + ... + 10	R0 = 55
Multiplication	6 * 7 par additions	R0 = 42
Fibonacci	Calculer F(10)	R0 = 55

### 12.2.3 Memoire

Exercice	Objectif	Resultat
Tableaux	Somme de {10, 20, 30, 40, 50}	R0 = 150
Maximum Tableau	Max de {12, 45, 7, 89, 23}	R0 = 89
Memoire	Store puis Load	R0 = 30

### 12.2.4 Structures

Ces exercices preparent aux structs en C en montrant comment les donnees structurees sont organisees en memoire.

Exercice	Objectif	Resultat
Structure Simple	Lire champs d'un Point (x+y)	R0 = 42
Initialiser Structure	Ecrire dans les champs d'un Point	R0 = 42
Structure Rectangle	Structure avec 4 champs, calcul aire	R0 = 42
Tableau de Structures	Parcourir tableau de Points, somme des x	R0 = 33
Somme x+y Structures	Acceder aux deux champs de chaque Point	R0 = 42

### 12.2.5 Fonctions

Exercice	Objectif	Resultat
Fonctions	Fonction double(21)	R0 = 42
Fonction Add3	add3(10, 15, 17)	R0 = 42

### 12.2.6 Entrees/Sorties

Exercice	Objectif	Resultat
Ecrire Caractere	Ecrire 'A' a PUTC	R0 = 65
Hello String	Afficher "Hi"	R0 = 2
Print Loop	Afficher "ABCD" avec boucle	R0 = 4

### 12.2.7 Ecran (320x240, 1 bit/pixel)

Exercice	Objectif	Resultat
Pixel	Allumer pixel (0,0)	R0 = 0x80
Ligne Horizontale	8 pixels horizontaux	R0 = 0xFF
Ligne Verticale	8 pixels verticaux	R0 = 8
Rectangle	Carre 8x8	R0 = 8
Damier	Motif en damier	R0 = 8

### 12.2.8 Jeux Interactifs

Exercice	Objectif
Lire Caractere	Lire clavier et convertir ASCII
Lire 2 Chiffres	Former un nombre a 2 chiffres
Deviner Nombre	Jeu de devinette (secret = 7)
Degrade	Effet dithering sur ecran
Recherche Dichotomique	Trouver 42 en 7 essais

### 12.2.9 Cache (Patterns d'Acces Memoire)

Ces exercices illustrent l'impact des patterns d'accès memoire sur les performances cache.

Exercice	Objectif	Resultat
Acces Sequentiel	Parcours cache-friendly (adresses consecutives)	R0 = 100
Acces avec Stride	Parcours avec sauts de 16 bytes (moins efficace)	R0 = 28
Reutilisation Registre	Charger une fois, reutiliser plusieurs fois	R0 = 91

## 12.3 C. Exercices C32

Ces exercices enseignent la programmation en C32.

### 12.3.1 Bases

Exercice	Objectif	Resultat
Variables	x=10, y=32, retourner x+y	42
Expressions	(5+3)*(10-4)/2	24
Modulo	(100%7) + (45%8)	7
Incrementation	5 -> +3 -> *2 -> -1	15

### 12.3.2 Conditions

Exercice	Objectif	Resultat
Conditions	Maximum de 25 et 17	25
Else-If	Classifier note 75	3
Operateurs Logiques	15 dans [10,20] ?	1
Maximum de 3	Max de 15, 42, 27	42

### 12.3.3 Boucles

Exercice	Objectif	Resultat
Boucle For	Somme 1 a 10	55
Boucle While	Compter chiffres de 12345	5
Boucles Imbriquees	Double boucle i*j	60
Multiplication	78 sans operateur	56

### 12.3.4 Fonctions

Exercice	Objectif	Resultat
Fonctions	square(7)	49
Parametres Multiples	add3(10, 20, 12)	42
Valeur Absolue	abs(-15) + abs(10)	25
Min et Max	max(10,25) - min(10,25)	15

### 12.3.5 Tableaux

Exercice	Objectif	Resultat
Tableaux	Somme de {3,7,2,9,5}	26
Maximum Tableau	Max de 6 elements	56

Exercice	Objectif	Resultat
Compter Elements	Nombres pairs	4

### 12.3.6 Pointeurs

Exercice	Objectif	Resultat
Pointeurs	Modifier via *p	42
Swap	Echanger x et y	20
Pointeurs et Tableaux	Somme avec *(p+i)	50

### 12.3.7 Operations Binaires

Exercice	Objectif	Resultat
Operations Binaires	(10&12)	(10^12)
Puissance de 2	is_pow2(16)+is_pow2(15)+is_pow2(32)	2

### 12.3.8 Recursion

Exercice	Objectif	Resultat
Factorielle	fact(5)	120
Fibonacci	fib(10)	55
Somme Recursive	sum(10)	55

### 12.3.9 Algorithmes Avances

Exercice	Objectif	Resultat
PGCD (Euclide)	gcd(48, 18)	6
Puissance	power(2, 10)	1024
Test Primalite	Premiers <= 20	8
Tri a Bulles	Trier et retourner min	12
Recherche Binaire	Trouver 23 dans tableau	5
Inverser Tableau	Inverser {1,2,3,4,5}	35
Somme Chiffres	digit_sum(12345)	15
Palindrome	12321 + 1221 + 123	2

### 12.3.10 Structures

Les structures permettent de regrouper plusieurs variables liees.

Exercice	Objectif	Resultat
Definition Struct	struct Point, p.x + p.y	42
Pointeur Struct	Utiliser l'operateur ->	42
Struct et Fonctions	distance_sq(Point*)	25
Structs Imbriquee	Rectangle avec Point	42
Tableau de Structs	Point[3], somme des x	33
Sizeof Struct	Taille des structures	16

### 12.3.11 Cache (Patterns d'Acces Memoire)

Ces exercices illustrent l'impact de la localite sur les performances cache.

Exercice	Objectif	Resultat
Parcours en Ligne	Parcours row-major (cache-friendly)	120
Parcours en Colonne	Parcours column-major (moins efficace)	120
Traitement par Blocs	Technique de blocking	120
Localite Temporelle	Reutiliser les donnees en cache	30

### 12.3.12 Entrees/Sorties

Exercice	Objectif	Resultat
Ecrire Caractere	putchar(65)	65
Afficher Chaine	print("HI")	2
Afficher Nombre	print_int(42)	42
Dessiner Pixel	Pixel (0,0)	128
Ligne Horizontale	16 pixels	16
Dessiner Rectangle	Carre 8x8	64

### 12.3.13 Projets Avances

Exercice	Objectif	Resultat
Crible Eratosthene	Premiers <= 50	15
Suite Collatz	Longueur pour n=27	112
Projet Final	Diviseurs de 28	28

## 12.4 D. Construction du Compilateur

Ces exercices construisent progressivement un compilateur C -> A32.

**12.4.1 Phase 1 : Lexer**

Exercice	Description
1.1 Reconnaître Chiffre	<code>is_digit(c)</code> : retourne 1 si '0'-'9'
1.2 Lire Nombre	<code>parse_number(s, pos)</code> : extrait un entier
1.3 Identifier Tokens	<code>next_token()</code> : retourne type du token

**12.4.2 Phase 2 : Parser**

Exercice	Description
2.1 Evaluer $a + b$	Parser et calculer une operation
2.2 Evaluer $a + b + c$	Chaine d'operations gauche a droite
2.3 Precedence	Descente recursive : <code>*</code> avant <code>+</code>
2.4 Parentheses	Supporter $(2 + 3) * 4$

**12.4.3 Phase 3 : Emission ASM**

Exercice	Description
3.1 Generer MOV	Produire <code>"MOV R0, #42"</code>
3.2 Operation Binaire	Mapper <code>+</code> -> <code>ADD</code> , <code>*</code> -> <code>MUL</code>
3.3 Comparaison	Generer <code>CMP</code> et conditions

**12.4.4 Phase 4 : CodeGen Expressions**

Exercice	Description
4.1 Constante -> A32	Code pour charger une constante
4.2 Addition -> A32	<code>a + b</code> -> <code>MOV/MOV/ADD</code>
4.3 Expression -> A32	Expression complete avec precedence

**12.4.5 Phase 5 : Structures de Controle**

Exercice	Description
5.1 If/Else -> A32	Sauts conditionnels et labels
5.2 While -> A32	Boucles avec labels

**12.4.6 Phase 6 : Fonctions**



Exercice	Description
6.1 Prologue/Epilogue	Sauvegarder LR, reserver pile
6.2 Appel de Fonction	Arguments et BL

#### 12.4.7 Phase 7 : Projet Final

Exercice	Description
7.1 Mini-Compilateur	Compiler expression en A32 executable

### 12.5 E. Systeme d'Exploitation

Ces exercices introduisent les concepts OS.

#### 12.5.1 Initialisation

Exercice	Description	Resultat
Bootstrap	Initialiser SP, effacer BSS, appeler main	42

#### 12.5.2 Gestion Memoire

Exercice	Description	Resultat
Bump Allocator	Allocation simple par incrementation	100
Free List	Allocateur avec liberation	1

#### 12.5.3 Drivers

Exercice	Description	Resultat
Driver Ecran	Fonctions set_pixel, clear_screen	4
Police Bitmap	Dessiner caracteres 8x8	51

#### 12.5.4 Console et Clavier

Exercice	Description	Resultat
Console	Console texte avec curseur (40x30)	1
Driver Clavier	Lire les touches, buffer clavier	3

### 12.5.5 Shell et Applications

Exercice	Description	Resultat
Shell	Interpreteur de commandes basique	1
Calculatrice	Evaluer expressions arithmetiques	42
Variables Shell	Variables dans le shell (\$x, \$y)	15
Compte a Rebours	Timer avec affichage	0

### 12.5.6 Multitache

Exercice	Description	Resultat
Interruptions	Gestion des interruptions timer	10
Coroutines	Changement de contexte manuel	2
Scheduler	Ordonnanceur round-robin	6

### 12.5.7 Projets OS

Exercice	Description
Projet 1: Mini-OS Shell	Shell complet avec commandes
Projet 2: Task Manager	Gestionnaire de taches multiples

## 12.6 Conseils pour les Exercices

1. **Commencez simple** : Les premiers exercices de chaque section sont accessibles
2. **Lisez l'enonce** : Chaque exercice contient des indices
3. **Testez souvent** : Le simulateur execute votre code instantanement
4. **Consultez les solutions** : Apres avoir essaye, comparez avec la solution
5. **Progresssez** : Chaque exercice prepare le suivant

**Bon courage !**

## 13 Le Cache : Pourquoi Votre Ordinateur Semble Rapide

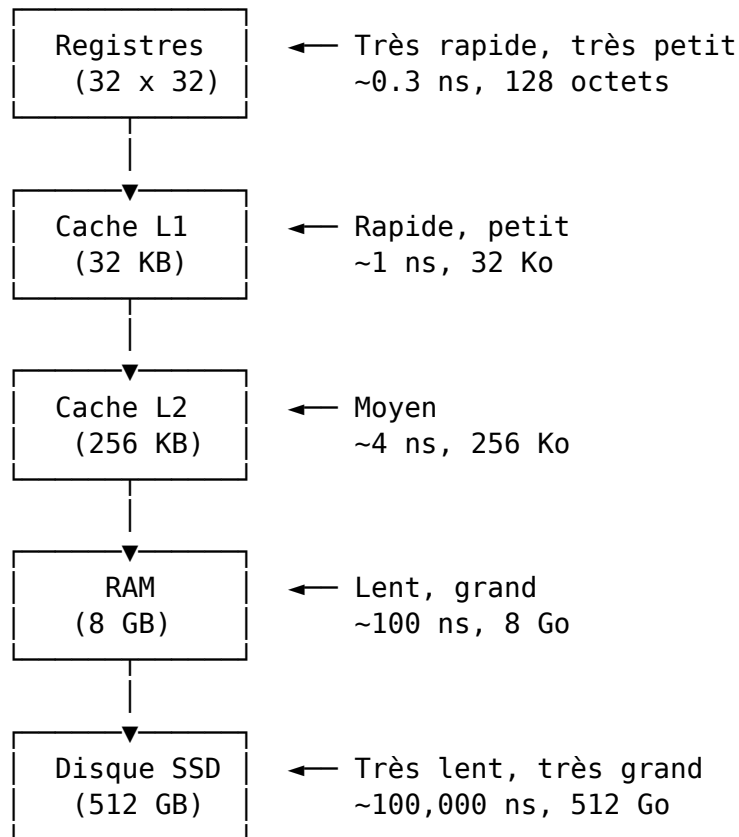
Imaginez que vous travaillez dans un bureau et que vous avez besoin de consulter des documents. Vous avez deux options : - **Votre bureau** : les documents sont juste devant vous, vous pouvez les lire instantanément - **Les archives au sous-sol** : il faut descendre 5 étages, chercher le bon dossier, remonter... cela prend plusieurs minutes

Le **cache** est exactement comme votre bureau : une petite zone de stockage très rapide où l'on garde les documents (données) les plus utilisés, pour éviter d'aller constamment aux archives (la mémoire RAM).

## 13.1 Le Problème : La RAM est Lente !

### 13.1.1 La Hiérarchie Mémoire

Voici comment est organisée la mémoire dans un ordinateur, du plus rapide au plus lent :



### 13.1.2 L'écart de vitesse visualisé

Si on convertissait ces temps en échelle humaine :

ÉCHELLE HUMAINE DES TEMPS D'ACCÈS		
Registres (0.3 ns)	→	Cliner des yeux (0.3 seconde)
Cache L1 (1 ns)	→	Une seconde
Cache L2 (4 ns)	→	4 secondes
RAM (100 ns)	→	1 minute 40 sec
SSD (100,000 ns)	→	27 heures !

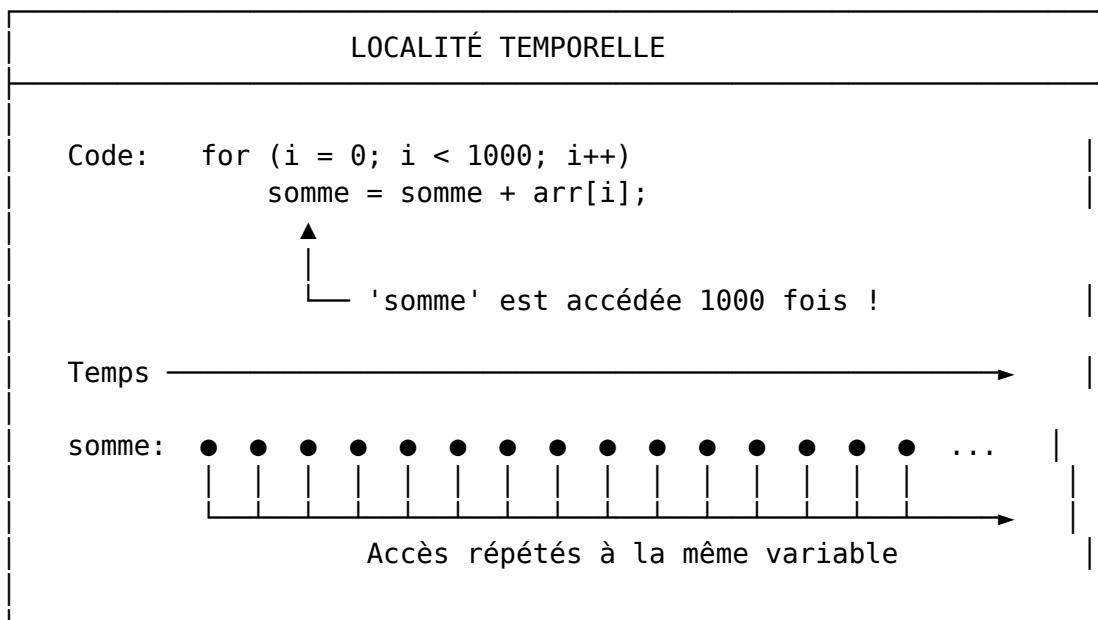
Sans cache, le processeur passerait **99% de son temps à attendre !**

## 13.2 La Solution : Le Principe de Localité

Les programmes ne lisent pas la mémoire au hasard. Ils suivent des **patterns** prévisibles.

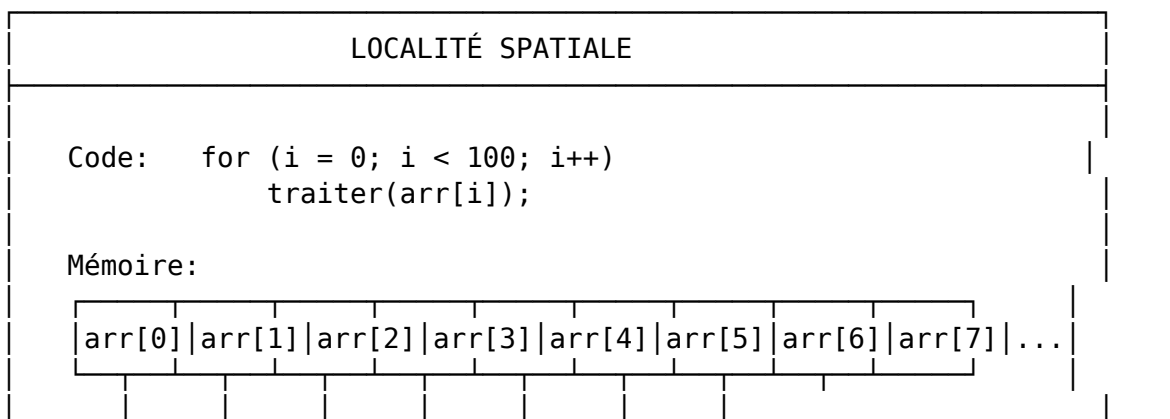
### 13.2.1 Localité Temporelle

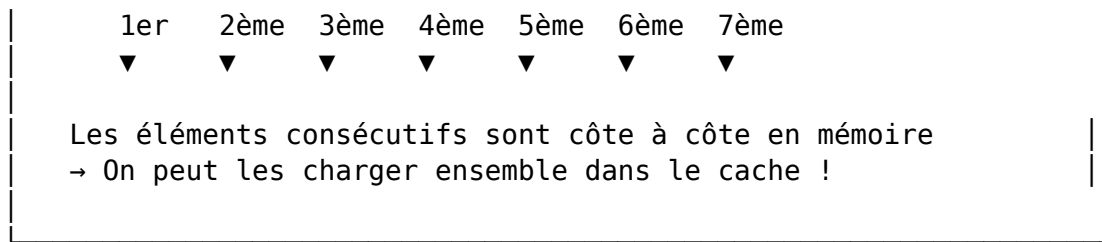
**“Si j’utilise une donnée maintenant, je vais probablement la réutiliser bientôt”**



### 13.2.2 Localité Spatiale

**“Si j’utilise une donnée, je vais probablement utiliser ses voisines”**

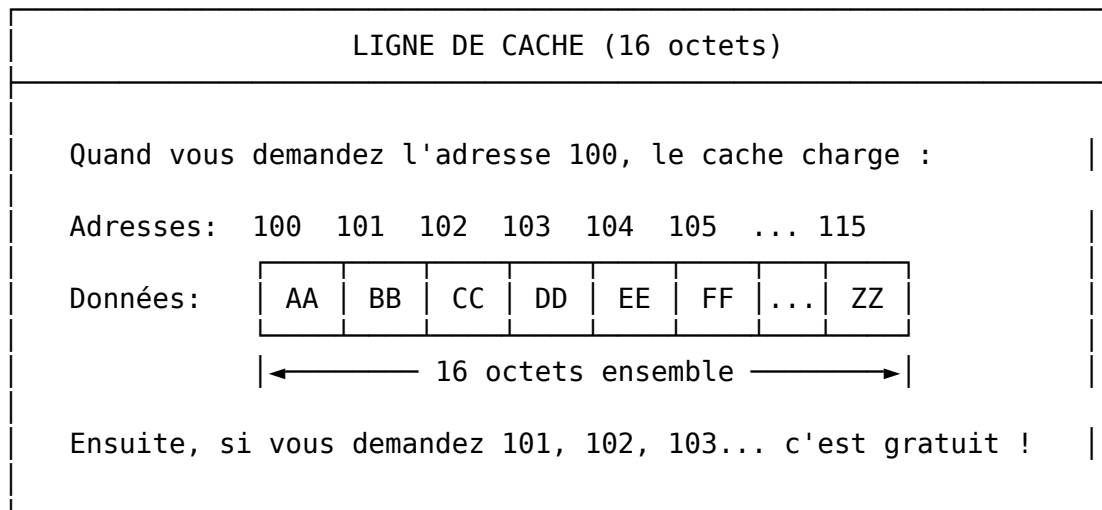




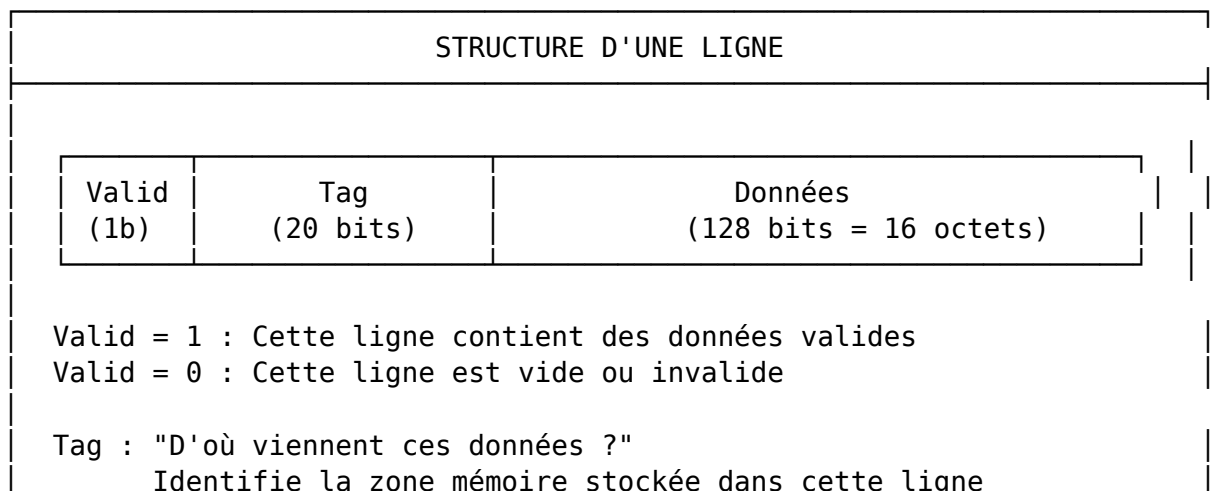
## 13.3 Comment Fonctionne le Cache ?

### 13.3.1 Les Lignes de Cache

Le cache stocke des **blocs** de données appelés **lignes de cache**, pas des octets individuels.



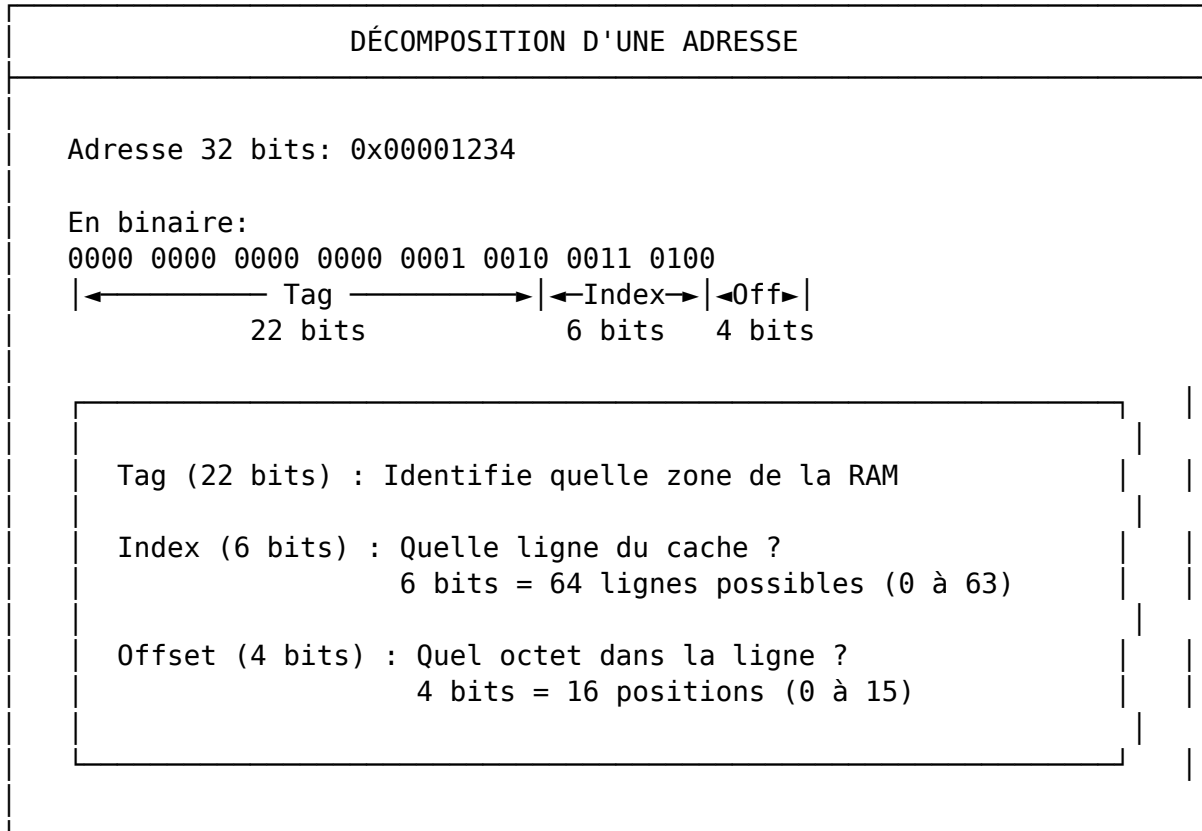
### 13.3.2 Structure d'une Ligne de Cache



Données : Les 16 octets de données eux-mêmes  
Contient 4 mots de 32 bits ( $4 \times 4 = 16$  octets)

### 13.3.3 Découpage d'une Adresse

Quand le CPU demande l'adresse `0x00001234`, comment le cache la trouve-t-il ?



### 13.3.4 Structure Complète du Cache

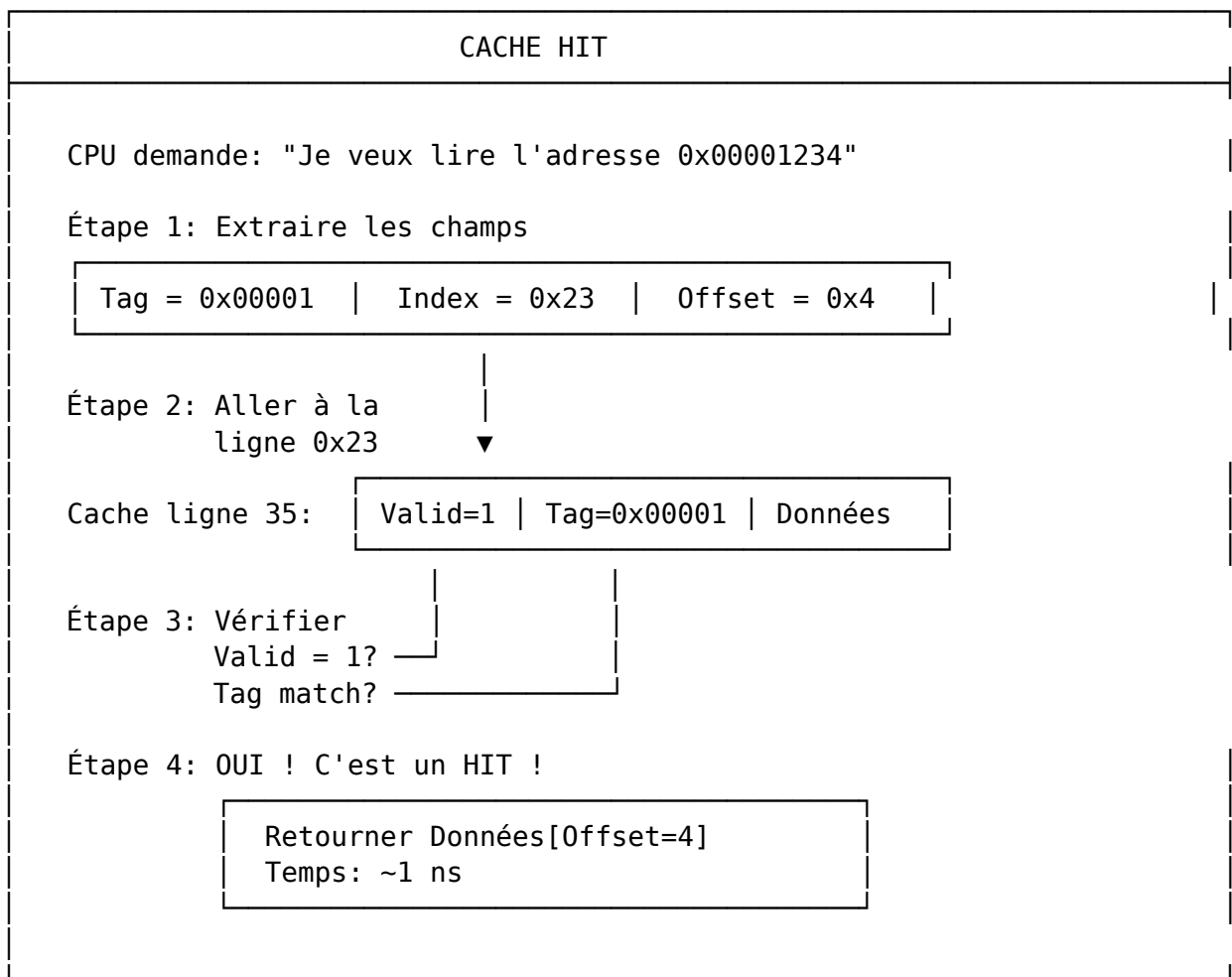
CACHE DIRECT-MAPPED (64 lignes)			
Index	Valid	Tag	Données (128 bits)
0	1	0x00012	[Mot0][Mot1][Mot2][Mot3]
1	0	-----	-----
2	1	0x00045	[Mot0][Mot1][Mot2][Mot3]
3	1	0x00123	[Mot0][Mot1][Mot2][Mot3] ← Ligne active
4	0	-----	-----

5	1	0x00067	[Mot0][Mot1][Mot2][Mot3]
...	...	...	...
63	1	0x00ABC	[Mot0][Mot1][Mot2][Mot3]

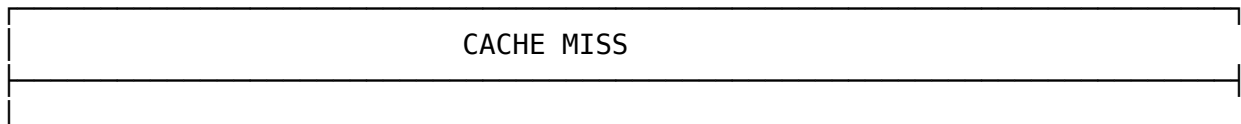
Chaque ligne = 1 bit Valid + 20 bits Tag + 128 bits Data

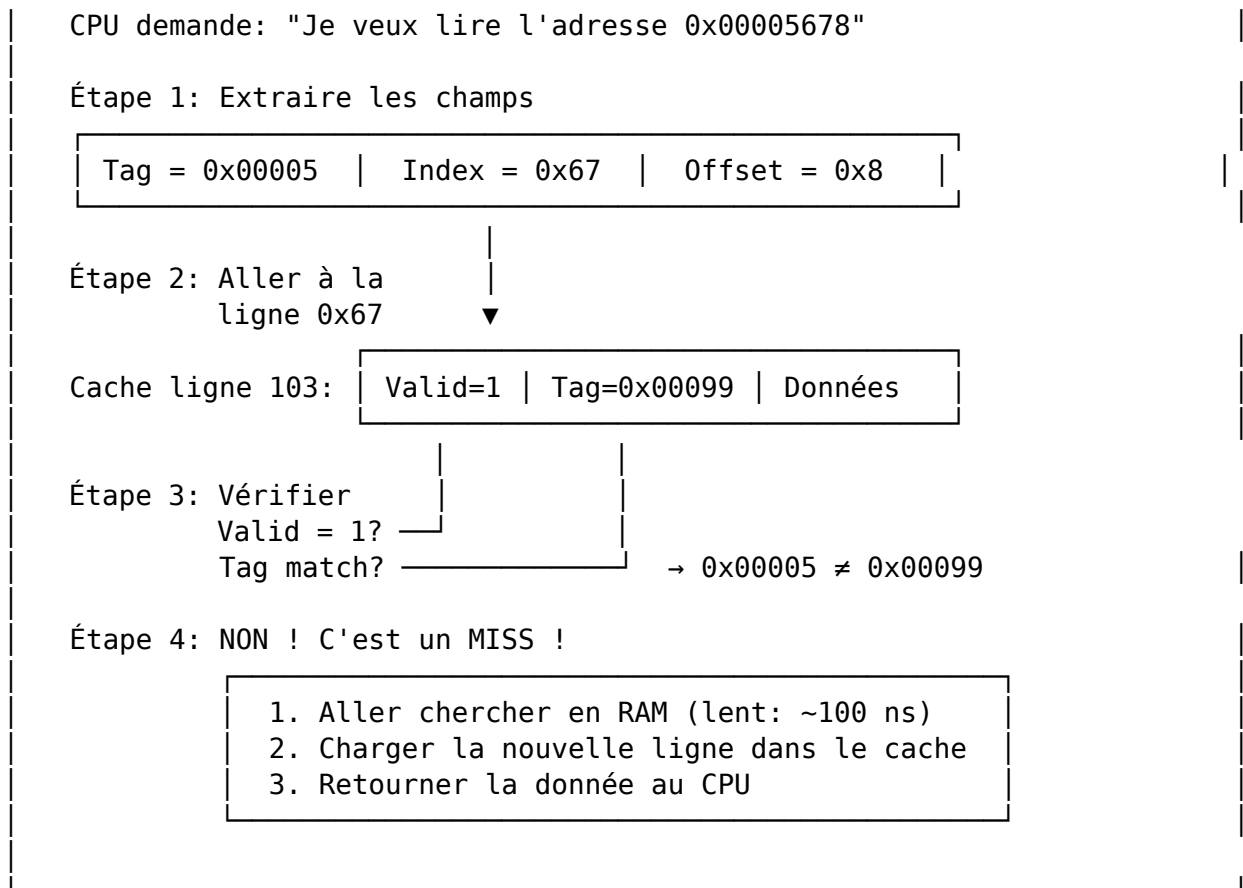
## 13.4 Hit ou Miss : Que se passe-t-il ?

### 13.4.1 Scénario 1 : Cache HIT (Succès)

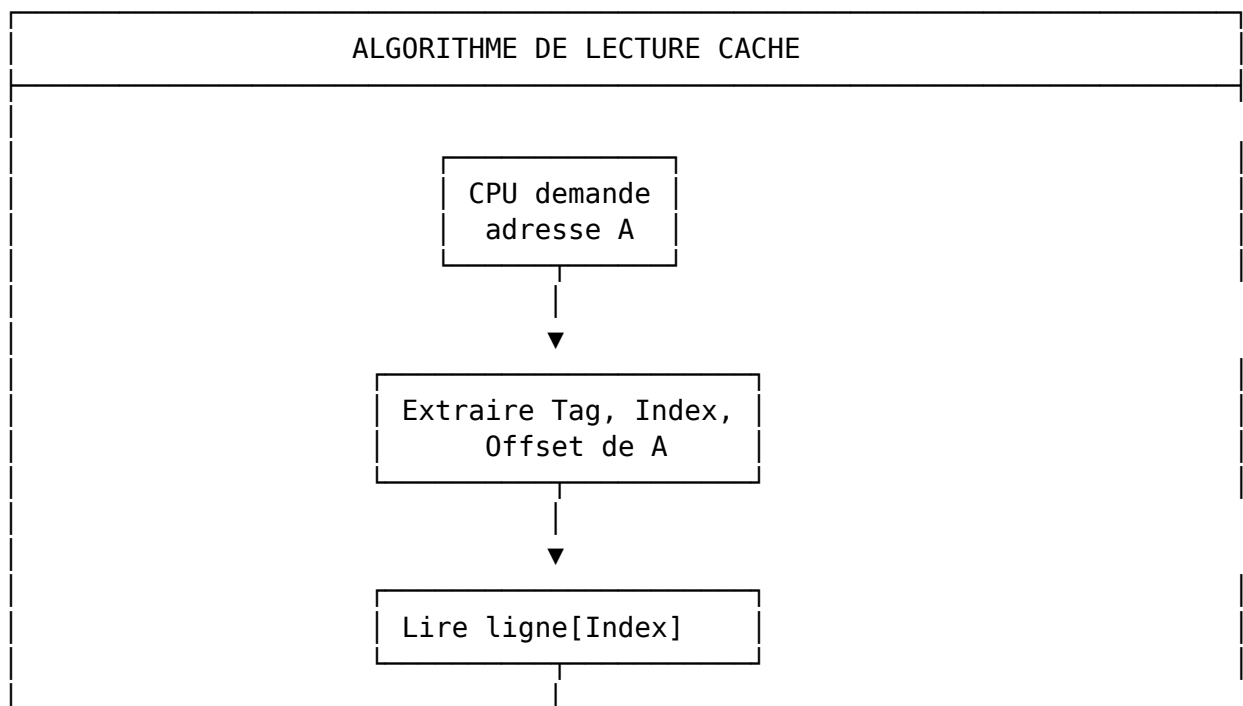


### 13.4.2 Scénario 2 : Cache MISS (Échec)

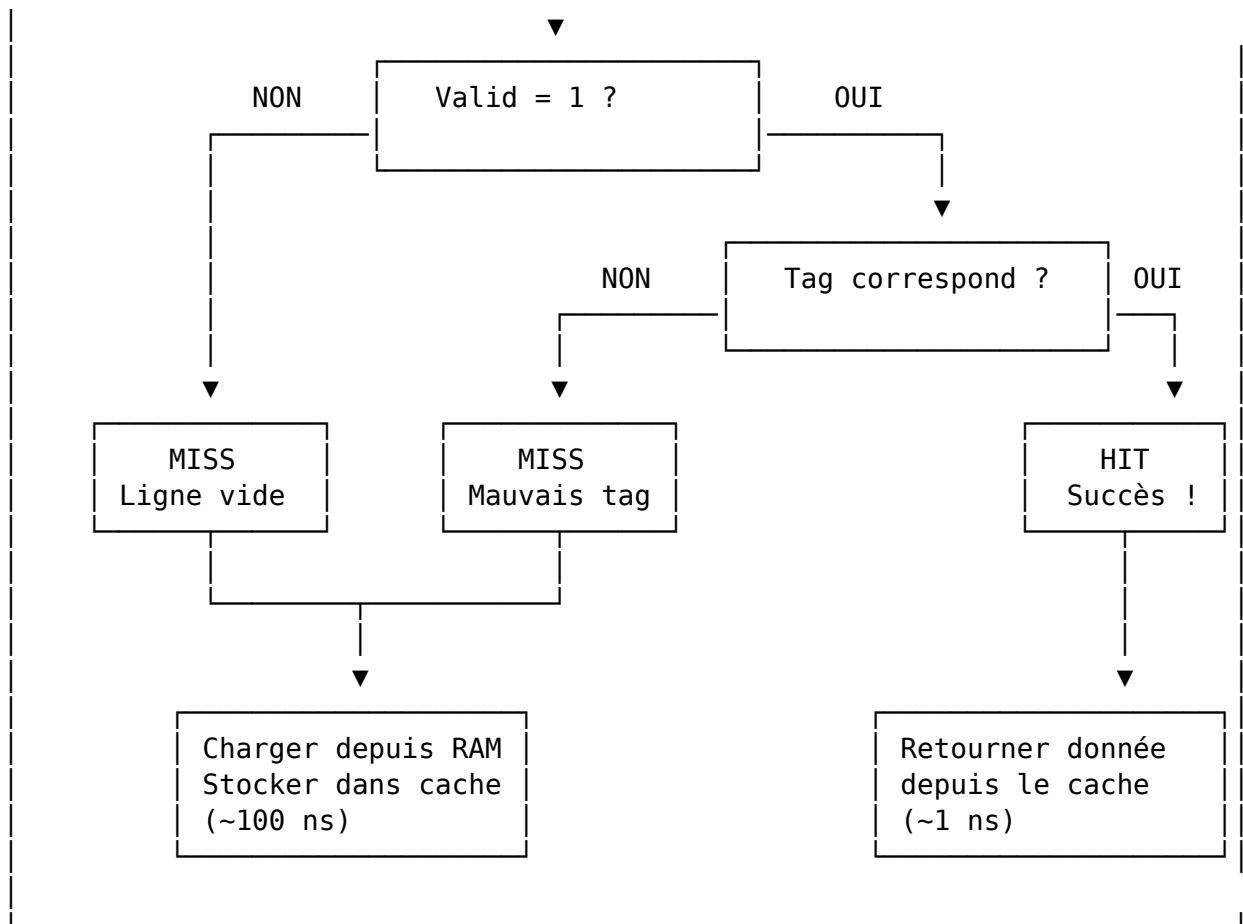




### 13.4.3 Diagramme de Flux Complet

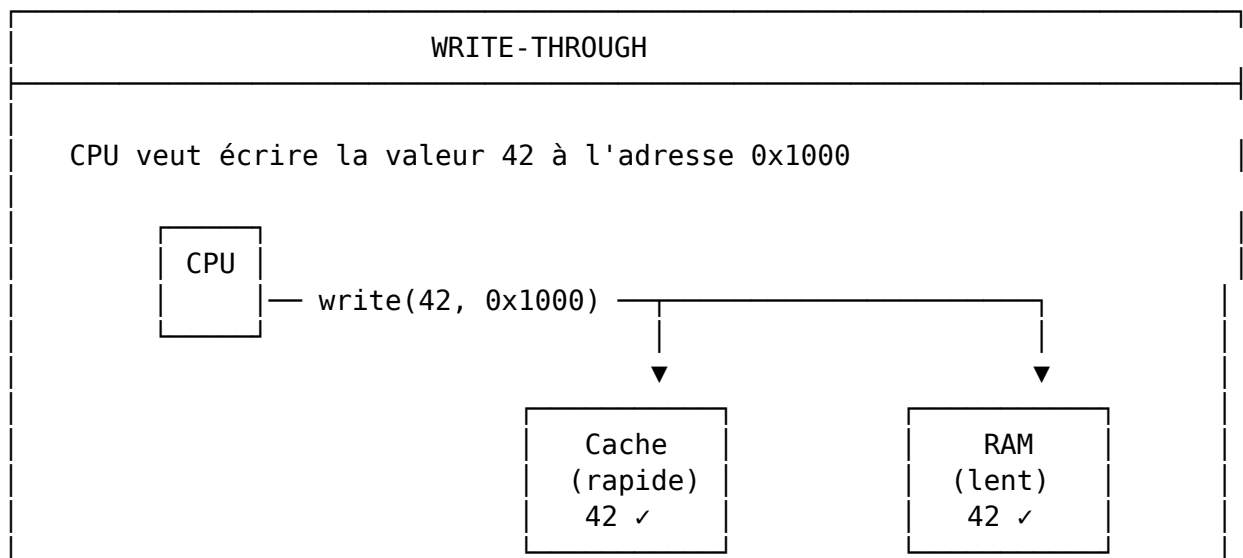






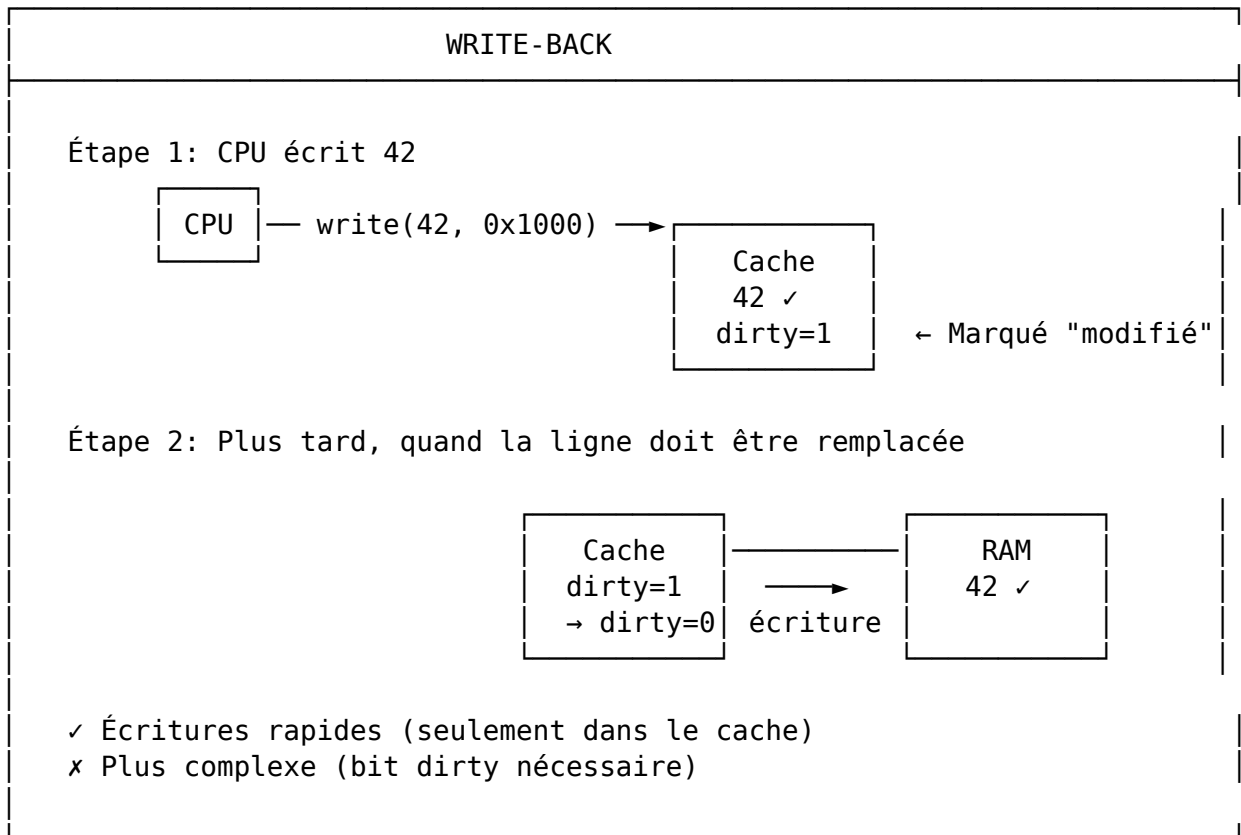
## 13.5 Politiques d'Écriture

### 13.5.1 Write-Through (Écriture Directe)



- ✓ Cache et RAM sont TOUJOURS synchronisés
- ✗ Chaque écriture attend la RAM (lent)

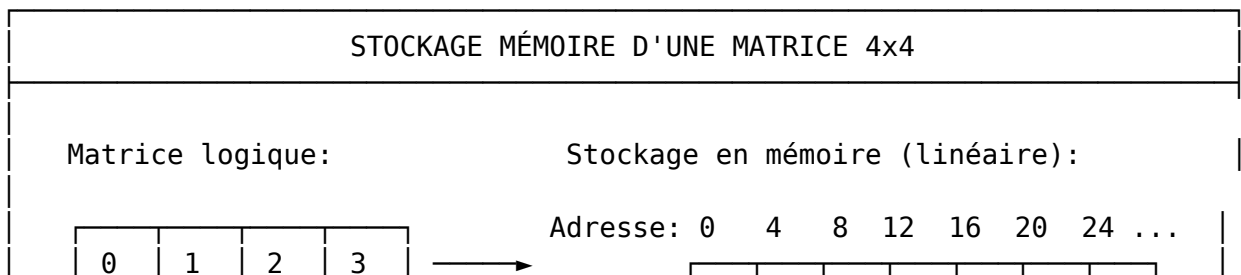
### 13.5.2 Write-Back (Écriture Différée)

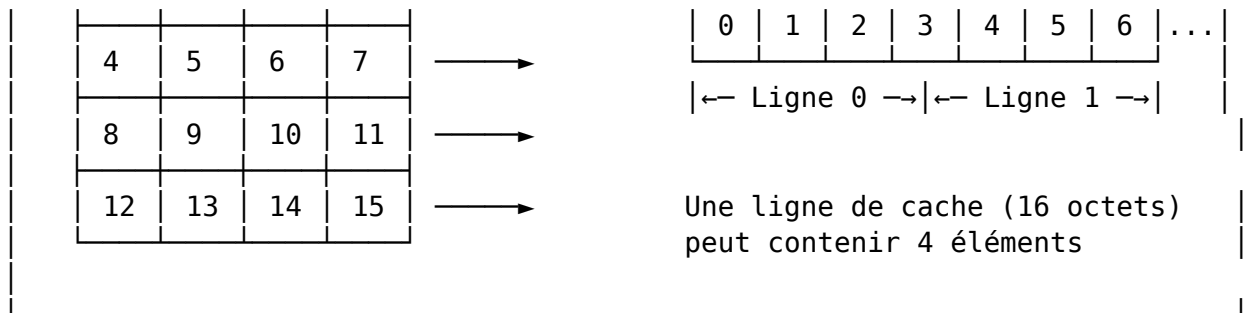


## 13.6 Impact sur vos Programmes

### 13.6.1 Parcours de Tableaux 2D : L'Ordre Compte !

Une matrice 4x4 est stockée **ligne par ligne** en mémoire (row-major) :





### 13.6.2 Parcours Row-Major (En Ligne) - EFFICACE

PARCOURS ROW-MAJOR (EFFICACE)

Code: 

```
for (i = 0..3)
  for (j = 0..3)
    accès arr[i][j]
```

Ordre d'accès:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

← D'abord toute la ligne 0  
← Puis toute la ligne 1  
← Puis toute la ligne 2  
← Puis toute la ligne 3

Mémoire: [0][1][2][3][4][5][6][7][8][9][10][11][12][13][14][15]  
Accès: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

→  
Accès SÉQUENTIEL = Excellent pour le cache !

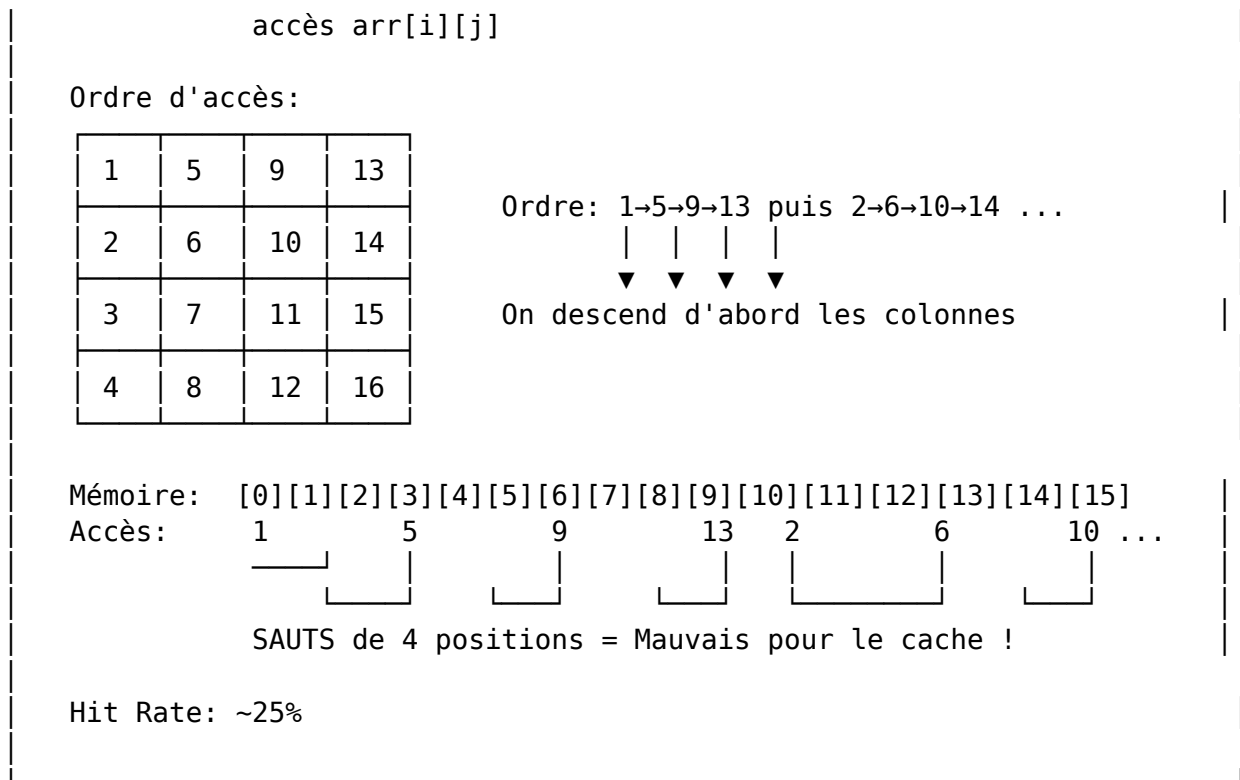
Hit Rate: ~95%

### 13.6.3 Parcours Column-Major (En Colonne) - INEFFICACE

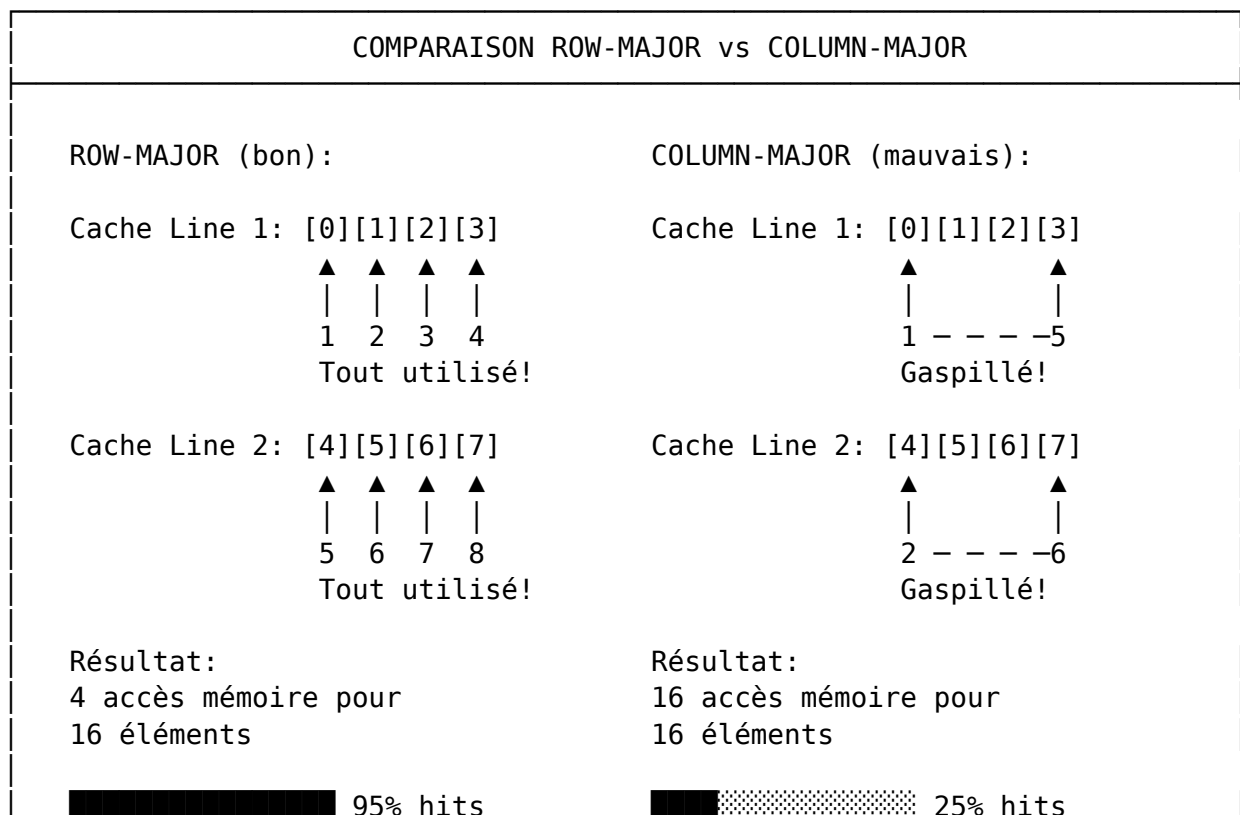
PARCOURS COLUMN-MAJOR (INEFFICACE)

Code: 

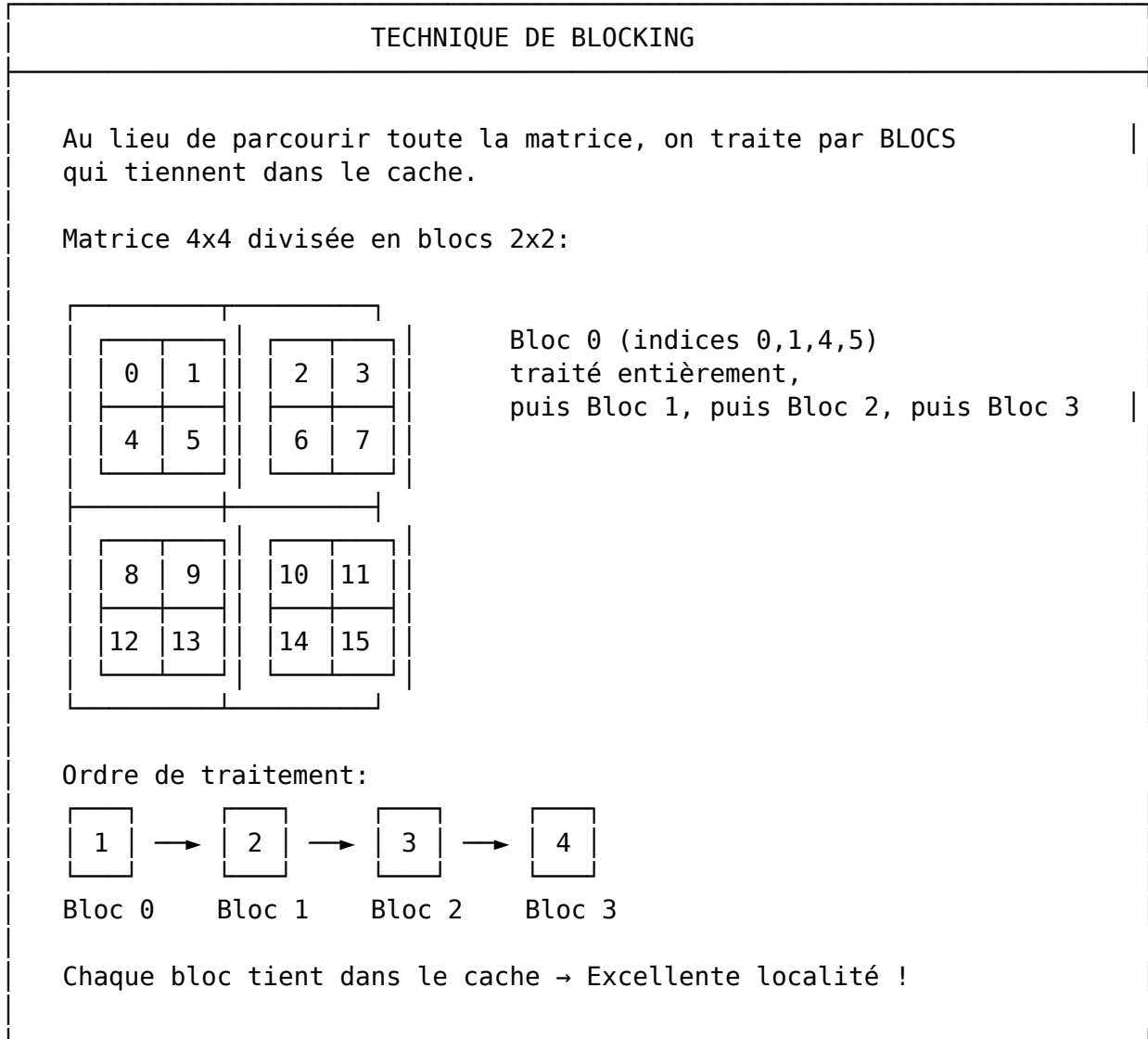
```
for (j = 0..3)
  for (i = 0..3)
```



### 13.6.4 Comparaison Visuelle

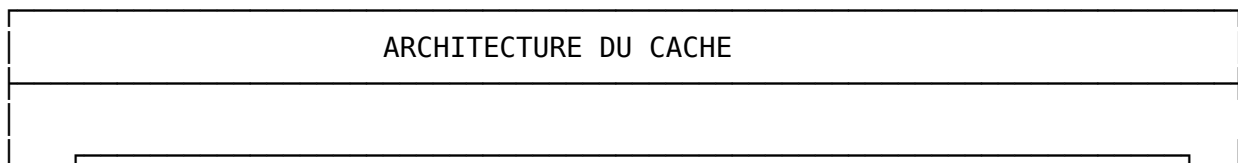


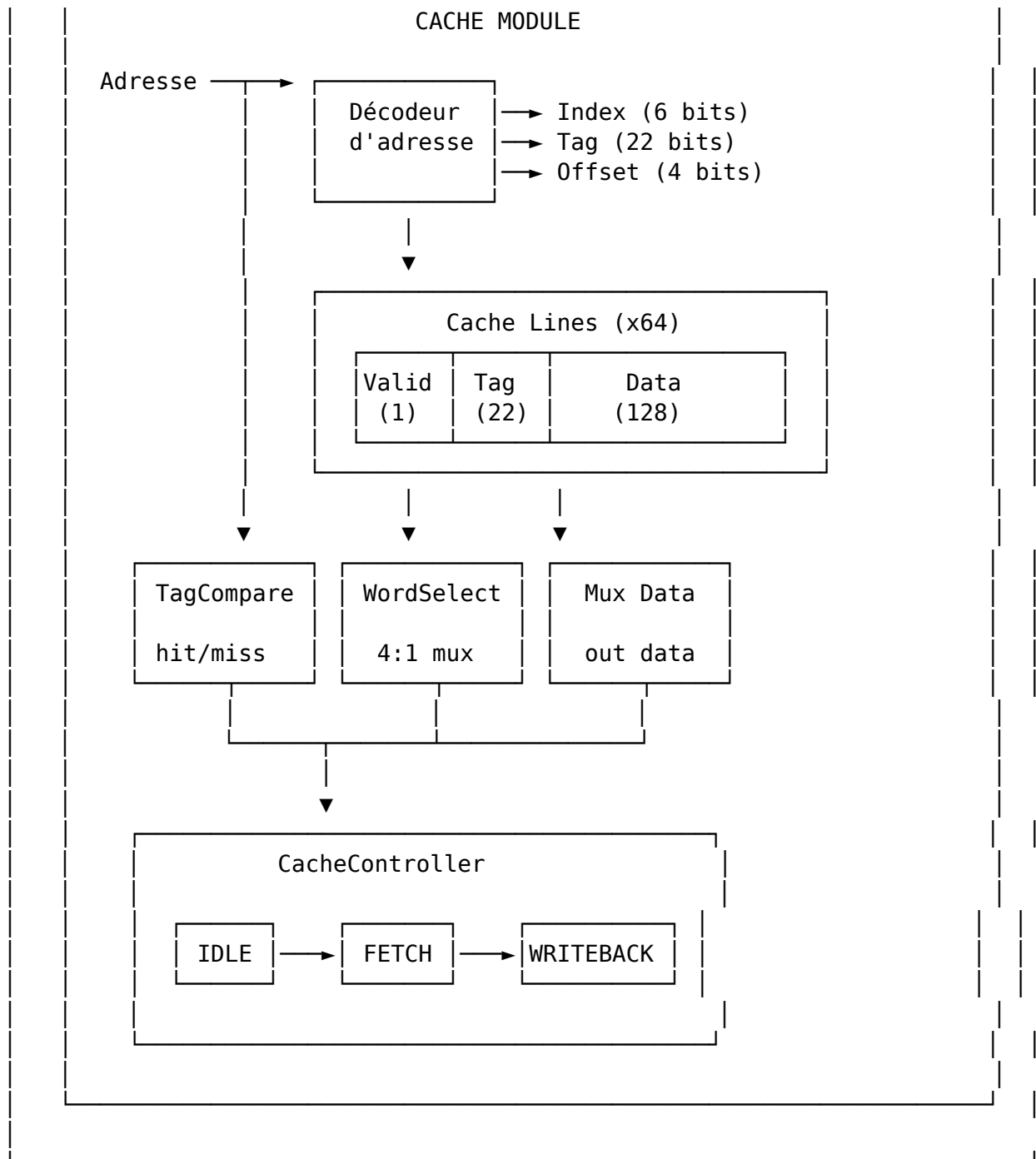
### 13.6.5 Technique du Blocking



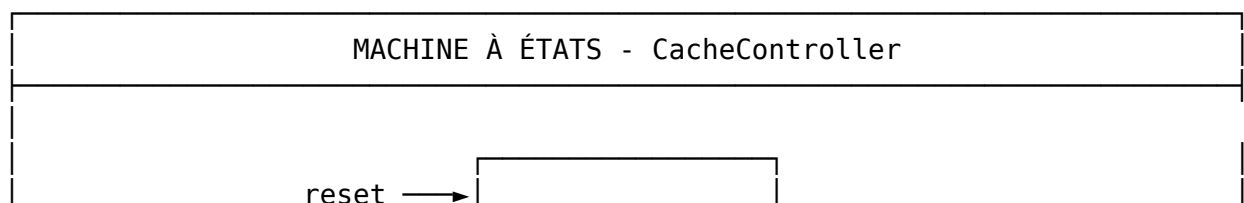
## 13.7 Implémentation HDL du Cache

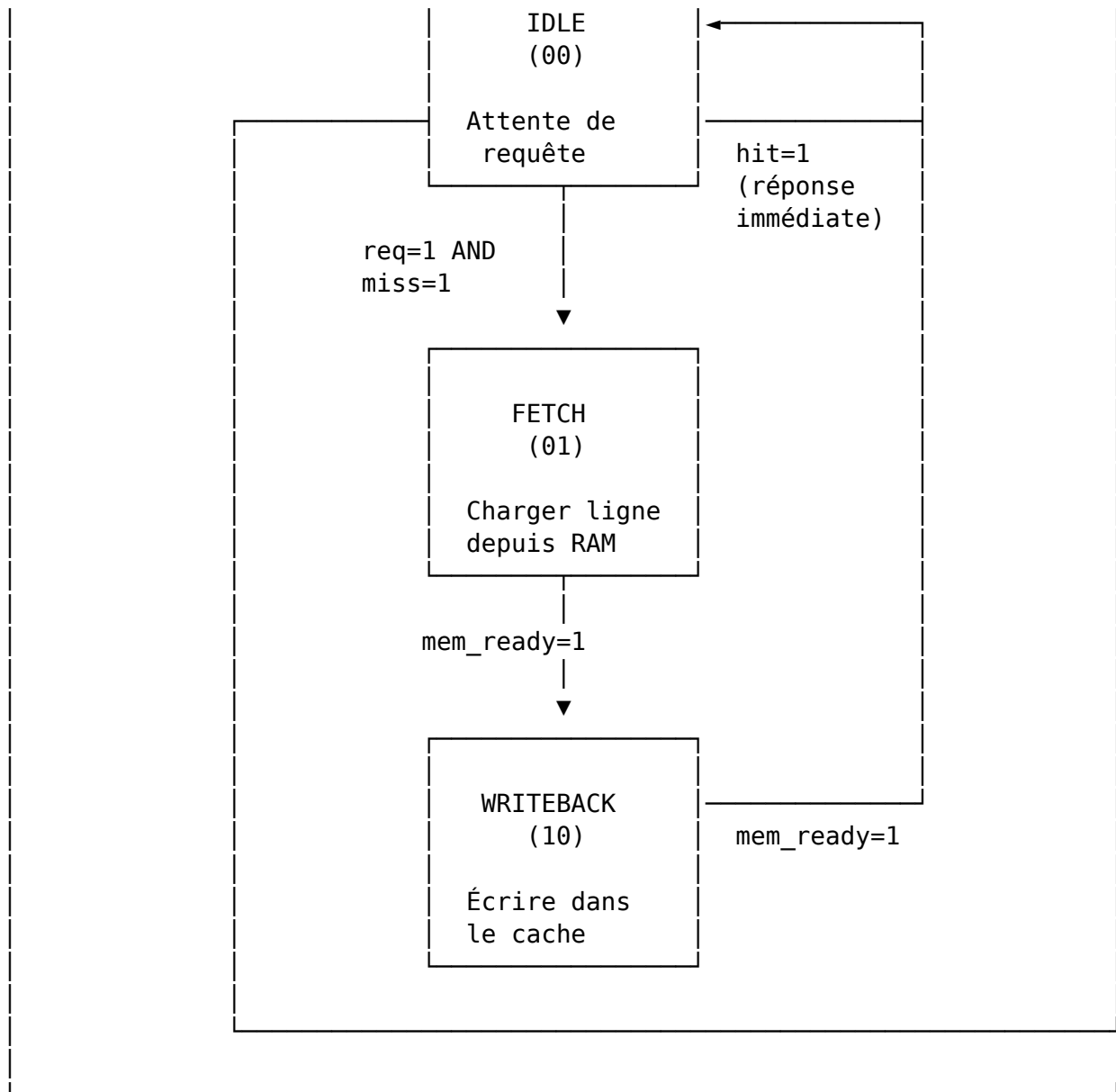
### 13.7.1 Architecture Globale



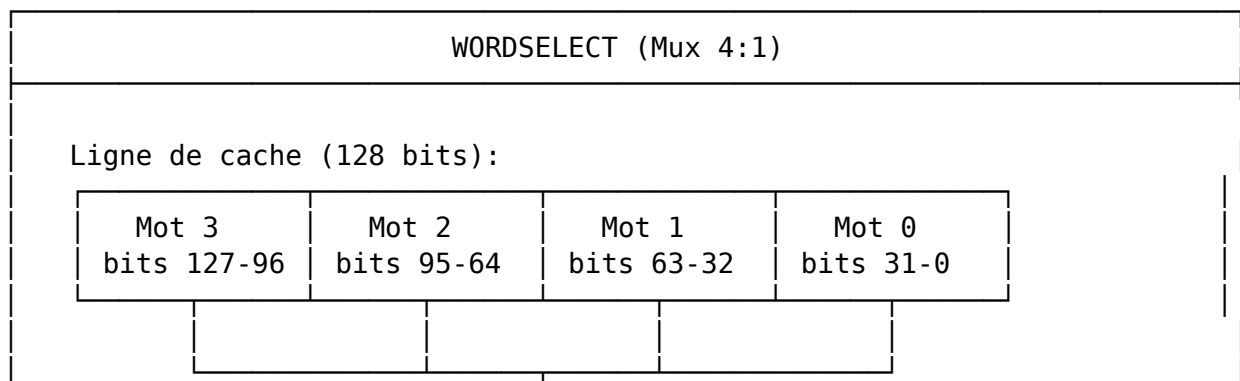


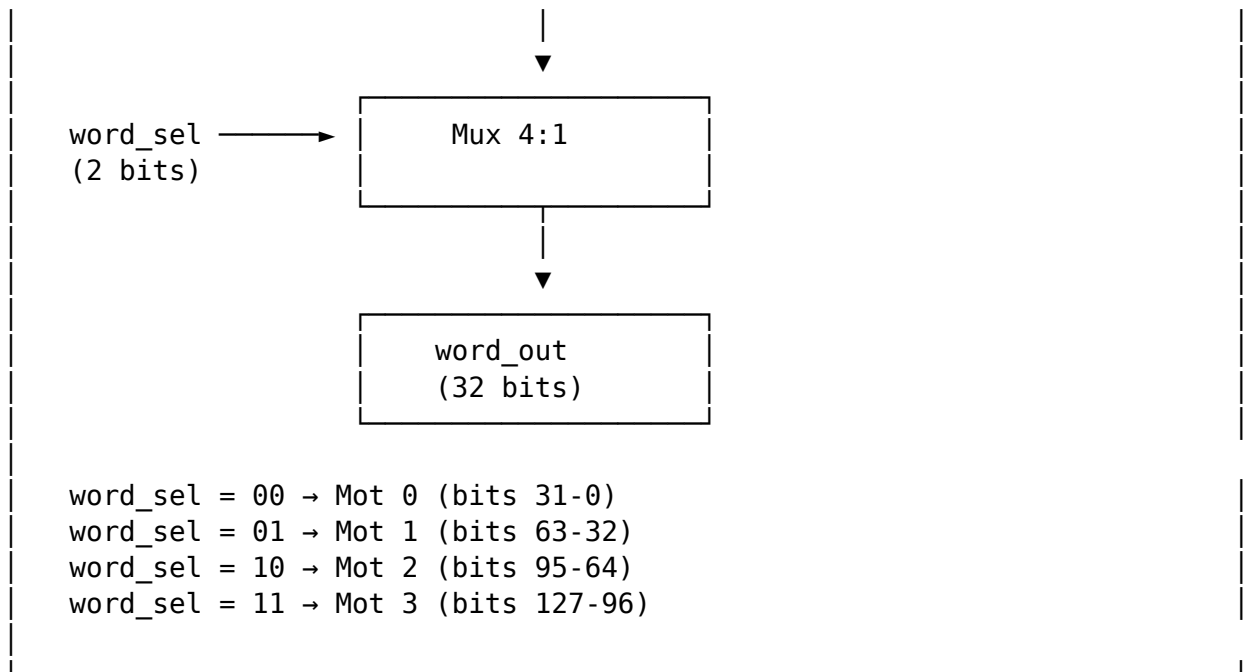
### 13.7.2 Machine à États du Contrôleur





### 13.7.3 WordSelect : Sélection du Mot



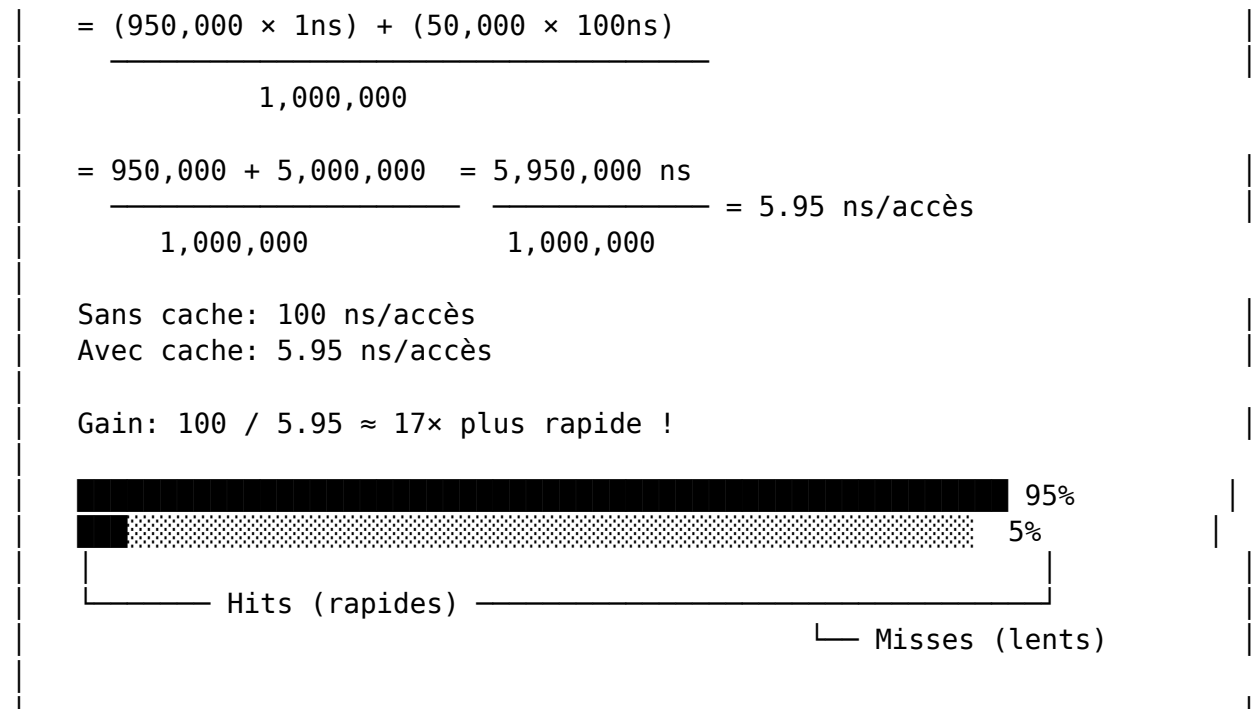


## 13.8 Statistiques et Performance

### 13.8.1 Calcul du Hit Rate

CALCUL DU HIT RATE
$\text{Hit Rate} = \frac{\text{Nombre de Hits}}{\text{Nombre de Hits} + \text{Nombre de Misses}} \times 100\%$
Exemple: 1 million d'accès mémoire
<ul style="list-style-type: none"> <li>• 950,000 hits (trouvés dans le cache)</li> <li>• 50,000 misses (chargés depuis la RAM)</li> </ul>
$\text{Hit Rate} = \frac{950,000}{1,000,000} \times 100\% = 95\%$
Temps moyen par accès:





## 13.9 Visualiser le Cache avec le CPU Visualizer

Le **CPU Visualizer** vous permet d'observer le comportement du cache en temps réel pendant l'exécution d'un programme.

### 13.9.1 Accéder au Visualizer

```
cd web
npm run dev
# Ouvrir http://localhost:5173 -> CPU Visualizer
```

### 13.9.2 La Démo "Cache"

Chargez la démo "**7. Cache**" dans le menu déroulant. Ce programme : 1. Parcourt un tableau de 16 éléments une première fois (cache misses) 2. Parcourt le même tableau une seconde fois (cache hits)

### 13.9.3 Ce que vous verrez

**Panneau "Cache L1"** : - **Hits** : Nombre d'accès trouvés dans le cache - **Misses** : Nombre d'accès qui ont dû aller en RAM - **Taux** : Pourcentage de hits (ex: "94.2%") - **Indicateur HIT/MISS** : Flash vert pour hit, rouge pour miss

**Contenu du cache** : - **Ligne** : Numéro de la ligne (0-63) - **Valid** : 1 si la ligne contient des données valides - **Tag** : Identifie quelle zone mémoire est stockée - **Données** : Le mot stocké dans la ligne

### 13.9.4 Exercice Pratique

1. Lancez la démo “Cache” et observez :
  - Au premier parcours : beaucoup de **MISS** (flash rouge)
  - Au second parcours : beaucoup de **HIT** (flash vert)
2. Regardez le taux de hits évoluer :
  - Début : ~0% (cache vide)
  - Après premier parcours : ~50%
  - Fin : ~85-95%
3. Observez les lignes de cache se remplir :
  - Les bits Valid passent de 0 à 1
  - Les Tags s’affichent
  - Les données apparaissent

## 13.10 Exercices

### 13.10.1 Exercices HDL

Exercice	Description
<b>CacheLine</b>	Implémenter une ligne de cache avec valid, tag, data
<b>TagCompare</b>	Comparateur de tags pour détecter hit/miss
<b>WordSelect</b>	Sélecteur de mot (4:1) dans une ligne 128 bits
<b>CacheController</b>	Machine à états (IDLE, FETCH, WRITEBACK)

### 13.10.2 Exercices Assembleur A32

Exercice	Description	Résultat
<b>Accès Séquentiel</b>	Parcours cache-friendly d’un tableau	R0 = 100
<b>Accès avec Stride</b>	Parcours avec sauts (moins efficace)	R0 = 28
<b>Réutilisation Registre</b>	Garder les données en registre	R0 = 91

### 13.10.3 Exercices C32

Exercice	Description	Résultat
<b>Parcours en Ligne</b>	Accès row-major (cache-friendly)	120
<b>Parcours en Colonne</b>	Accès column-major (moins efficace)	120
<b>Traitement par Blocs</b>	Technique de blocking	120
<b>Localité Temporelle</b>	Réutiliser les données	30

## 13.11 Résumé Visuel

RÉSUMÉ DU CACHE
<p><u>POURQUOI ?</u></p> <p>RAM (~100ns) est 100× plus lente que le CPU (~1ns)  → Le cache comble cet écart</p> <p><u>COMMENT ?</u></p> <p>Exploite la LOCALITÉ:</p> <ul style="list-style-type: none"> <li>• Temporelle: réutilisation des données récentes</li> <li>• Spatiale: données voisines accédées ensemble</li> </ul> <p><u>STRUCTURE:</u></p> <p>Adresse → [Tag   Index   Offset]  Ligne de cache → [Valid   Tag   Données (16 octets)]</p> <p><u>OPTIMISATION:</u></p> <ul style="list-style-type: none"> <li>✓ Parcours row-major (accès séquentiels)</li> <li>✓ Blocking (traitement par blocs)</li> <li>✓ Réutilisation des registres</li> <li>✗ Parcours column-major (sauts en mémoire)</li> </ul>

## 13.12 Points Clés à Retenir

1. **La RAM est lente** : ~100× plus lente que le cache
2. **Le cache exploite la localité** : temporelle et spatiale
3. **L'ordre d'accès compte** : row-major » column-major
4. **Réutilisez les données** : gardez-les en registre ou en cache
5. **Pensez en blocs** : traitez des données qui tiennent dans le cache

Ces principes s'appliquent à tous les niveaux de programmation, du code assembleur aux applications modernes !