L’Architecture Codex

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

Tahiry RAZAFINDRALAMBO

2026

Table des matières

1 L'Architecture Codex : Guide de l'Étudiant

Ce livre vous guide pas à pas dans la construction d'un ordinateur complet, moderne et 32-bits, à partir de rien : des portes logiques jusqu'au système d'exploitation.

1.1 Table des Matières

- 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 :

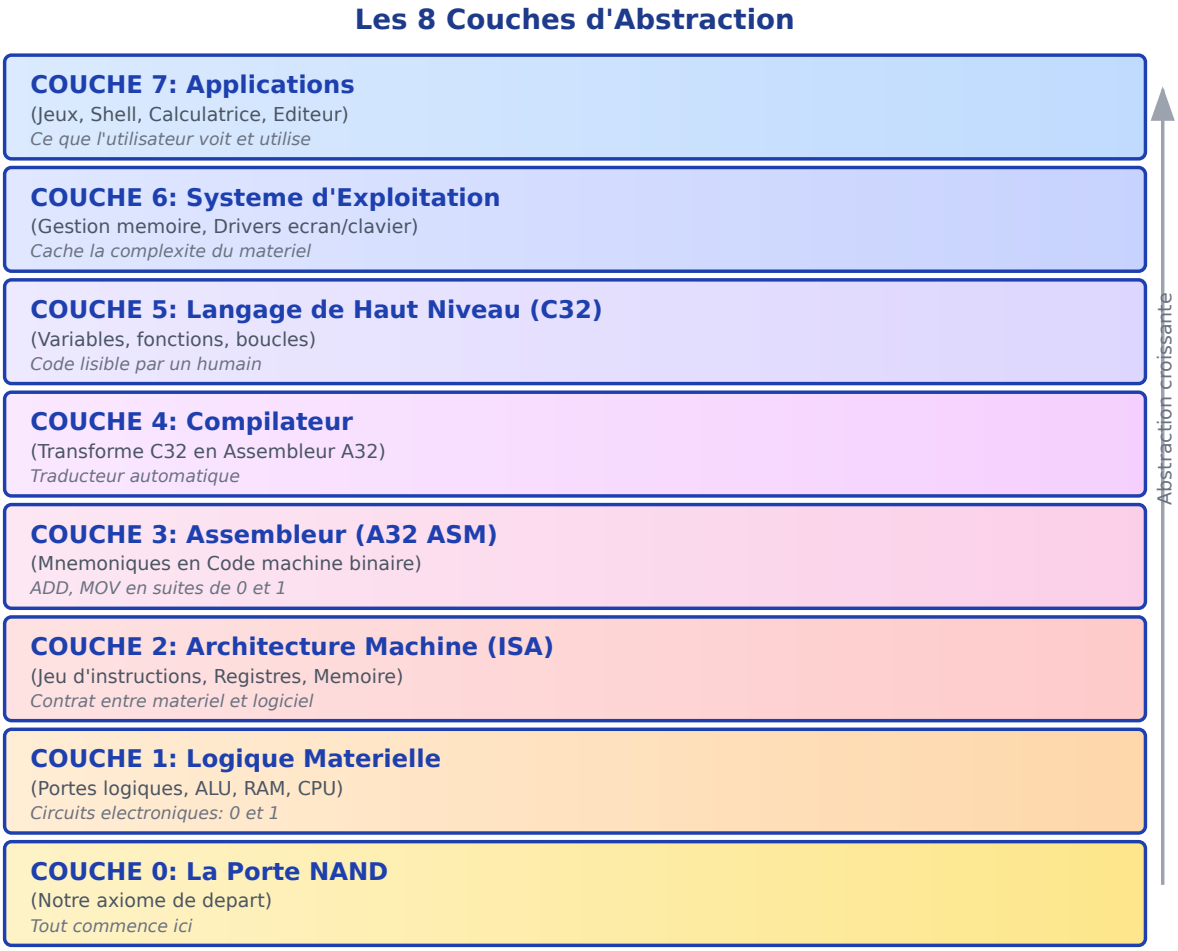


Figure 2.1: Les 8 couches d'abstraction

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 L'Architecture Codex A32

L'ordinateur **Codex** est conçu pour être à la fois pédagogique et réaliste. Il s'inspire des architectures ARM modernes tout en restant accessible.

Comparé aux architectures pédagogiques classiques plus simples (comme Hack), Codex offre :

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

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

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 ?

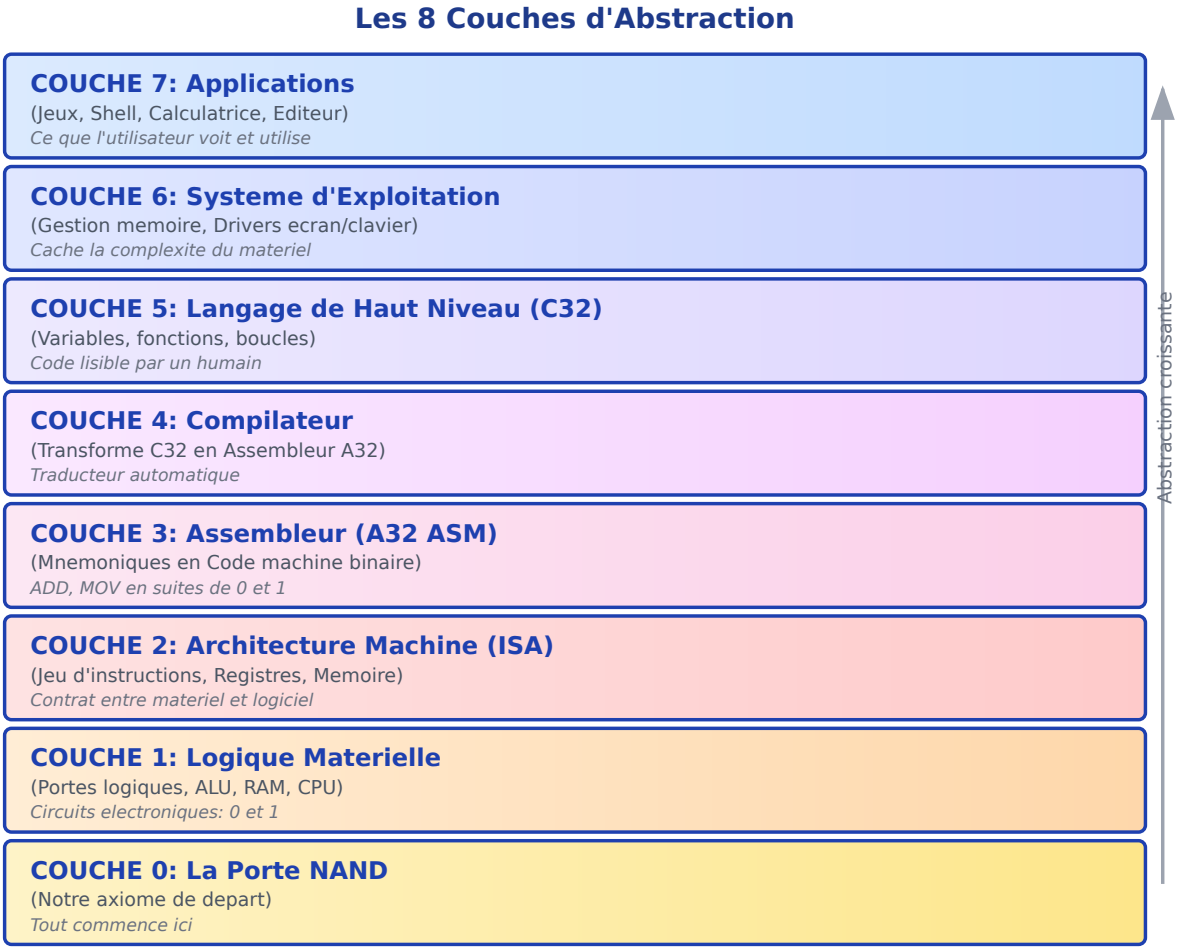


Figure 3.1: Position dans l'architecture

Nous sommes à la Couche 0 : La Porte NAND - Notre axiome de départ

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	**1**
0	1	**1**
1	0	**1**
1	1	**0**

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

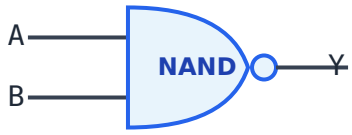


Figure 3.2: Porte NAND

Le petit cercle (○) 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 !

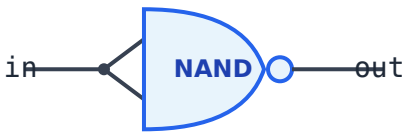


Figure 3.3: NOT construit à partir de NAND

Vérifions avec la table de vérité :

- Si `in = 0` : NAND(0, 0) = 1 OK
- Si `in = 1` : NAND(1, 1) = 0 OK

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 !

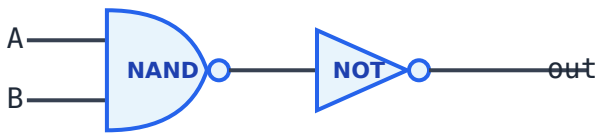


Figure 3.4: AND construit à partir de NAND et NOT

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 OR B = NOT( (NOT A) AND (NOT B) )
```

En utilisant uniquement des NAND :

```
A OR B = (NOT A) NAND (NOT B)
```

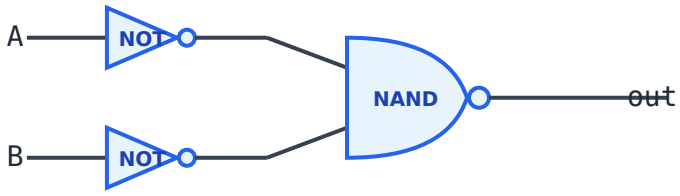


Figure 3.5: OR construit à partir de NOT et NAND

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 :

```
XOR(A, B) = (A AND NOT B) OR (NOT A 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`

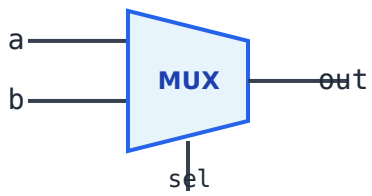


Figure 3.6: Multiplexeur

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

```
out = (a AND NOT sel) OR (b 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`

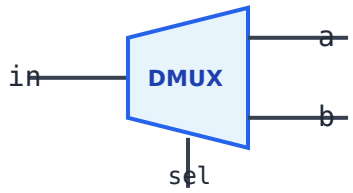


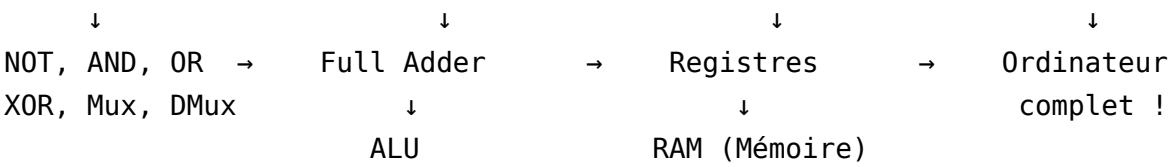
Figure 3.7: Démultiplexeur

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

```
CHAPITRE 1 (Ici)     CHAPITRE 2          CHAPITRE 3          CHAPITRE 5
      ↓                    ↓                   ↓                   ↓
    NAND              Half Adder           DFF (Flip-Flop)       CPU
```



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 :

```vhdl
-- ============================================================
-- 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);

  -- Assignation 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 Portes Multi-Entrées et Multi-Bits

Jusqu'ici, nous avons construit des portes à **2 entrées** (And2, Or2, Xor2). Mais en pratique, nous avons souvent besoin de : - **Portes à N entrées** : Or8Way, And4Way - **Portes sur N bits** : And16, Or32

3.7.1 Généralisation à N Entrées

3.7.1.1 Or8Way : OU sur 8 bits

Imaginons qu'on veuille savoir si **au moins un bit** parmi 8 est à 1. C'est un OR à 8 entrées :

```
Or8Way(a[0..7]) = a[0] OR a[1] OR a[2] OR ... OR a[7]
```

Construction hiérarchique (en arbre) :

```
Niveau 1 :  Or2(a[0], a[1]) → t0    Or2(a[2], a[3]) → t1
            Or2(a[4], a[5]) → t2    Or2(a[6], a[7]) → t3
```

```
Niveau 2 :  Or2(t0, t1) → t4        Or2(t2, t3) → t5

Niveau 3 :  Or2(t4, t5) → sortie
```

Avantage : Seulement 3 niveaux de profondeur ($\log_2(8) = 3$), au lieu de 7 en série.

```vhdl
entity Or8Way is
  port(
    a : in bits(7 downto 0);
    y : out bit
  );
end entity;

architecture rtl of Or8Way is
  component Or2
    port(a : in bit; b : in bit; y : out bit);
  end component;
  signal t0, t1, t2, t3, t4, t5 : bit;
begin
  -- Niveau 1 : paires
  u0: Or2 port map (a => a(0), b => a(1), y => t0);
  u1: Or2 port map (a => a(2), b => a(3), y => t1);
  u2: Or2 port map (a => a(4), b => a(5), y => t2);
  u3: Or2 port map (a => a(6), b => a(7), y => t3);
  -- Niveau 2
  u4: Or2 port map (a => t0, b => t1, y => t4);
  u5: Or2 port map (a => t2, b => t3, y => t5);
  -- Niveau 3
  u6: Or2 port map (a => t4, b => t5, y => y);
end architecture;
```

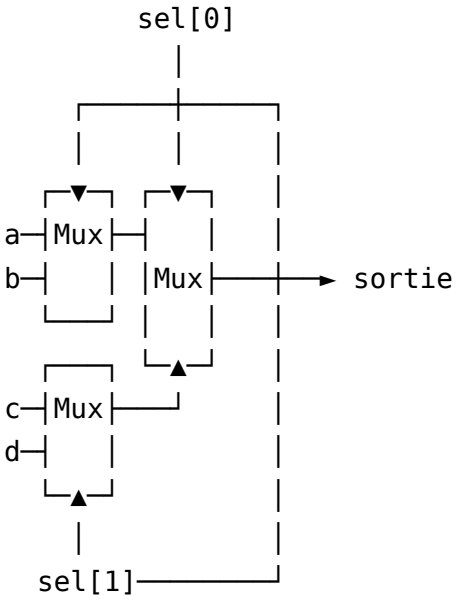
3.7.2 Multiplexeurs Multi-Entrées

3.7.2.1 Mux4Way : Choisir parmi 4 entrées

Avec 2 bits de sélection, on peut choisir parmi 4 entrées :

sel[1]	sel[0]	Sortie
0	0	a
0	1	b
1	0	c
1	1	d

Construction avec des Mux 2:1 :




```vhdl
entity Mux4Way is
  port(
    a, b, c, d : in bit;
    sel : in bits(1 downto 0);
    y : out bit
  );
end entity;

architecture rtl of Mux4Way is
  component Mux
    port(a : in bit; b : in bit; sel : in bit; y : out bit);
  end component;
  signal ab, cd : bit;
begin
  -- Niveau 1 : sélection par sel[0]
  u_ab: Mux port map (a => a, b => b, sel => sel(0), y => ab);
  u_cd: Mux port map (a => c, b => d, sel => sel(0), y => cd);
  -- Niveau 2 : sélection par sel[1]
  u_out: Mux port map (a => ab, b => cd, sel => sel(1), y => y);
end architecture;
```

3.7.2.2 Mux8Way : 8 entrées, 3 bits de sélection

Le même principe s'étend : 2 Mux4Way + 1 Mux donne un Mux8Way.

3.7.3 Démultiplexeurs Multi-Sorties

3.7.3.1 DMux4Way : Router vers 4 destinations

L'inverse du Mux : une entrée, 4 sorties possibles.

sel[1]	sel[0]	a	b	c	d
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	in

Construction :

```
entity DMux4Way is
  port(
    input : in bit;
    sel : in bits(1 downto 0);
    a, b, c, d : out bit
  );
end entity;

architecture rtl of DMux4Way is
  component DMux
    port(input : in bit; sel : in bit; a : out bit; b : out bit);
  end component;
  signal top, bottom : bit;
begin
  -- Niveau 1 : sel[1] divise en haut/bas
  u_split: DMux port map (input => input, sel => sel(1), a => top, b => bottom);
  -- Niveau 2 : sel[0] divise chaque moitié
  u_top: DMux port map (input => top, sel => sel(0), a => a, b => b);
  u_bot: DMux port map (input => bottom, sel => sel(0), a => c, b => d);
end architecture;
```

3.7.4 Portes sur N Bits (Bus)

Pour traiter des mots de 16 ou 32 bits, on réplique la porte bit par bit :

```
-- And32 : ET bit-à-bit sur 32 bits
entity And32 is
  port(
    a : in bits(31 downto 0);
```

```
    b : in bits(31 downto 0);
    y : out bits(31 downto 0)
  );
end entity;

architecture rtl of And32 is
  component And2
    port(a : in bit; b : in bit; y : out bit);
  end component;
begin
  -- Instancier 32 portes AND en parallèle
  gen: for i in 0 to 31 generate
    u: And2 port map (a => a(i), b => b(i), y => y(i));
  end generate;
end architecture;
```

Note : La construction `for ... generate` crée automatiquement 32 instances. C'est équivalent à écrire 32 lignes `port map`.

3.7.5 Pourquoi c'est Important ?

Ces portes multi-entrées sont essentielles pour :

Composant	Utilisation
Or8Way	Tester si un registre est non-nul (flag Z de l'ALU)
Mux4Way/8Way	Sélectionner un registre parmi 4, 8, ou 16
DMux4Way/8Way	Router un signal `load` vers le bon registre
And32, Or32	Opérations bit-à-bit de l'ALU
Mux32	Choisir entre deux bus 32 bits

Au chapitre 3 (Mémoire), vous utiliserez DMux8Way pour adresser 8 registres. Au chapitre 5 (CPU), Mux16Way sélectionnera parmi 16 registres.

3.8 Exercices Pratiques

3.8.1 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	[**]
`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

3.8.2 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`).

3.8.3 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/
```

3.9 Défis Supplémentaires

3.9.1 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.

3.9.2 Défi 2 : Implémenter IMPLIES

La fonction "implication" (`A → B`) vaut FAUX seulement si A est VRAI et B est FAUX.

A	B	A → B
0	0	1
0	1	1
1	0	0
1	1	1

Construisez cette porte en utilisant les portes élémentaires.

3.9.3 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]`).

3.10 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 !

3.11 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

3.11.1 Questions de compréhension

Q1. Pourquoi les ordinateurs utilisent-ils le binaire plutôt que le décimal ?

Q2. Pourquoi dit-on que NAND est une porte "universelle" ?

Q3. Quelle est la sortie de `XOR(1, 1)` ? Et de `XOR(0, 1)` ?

Q4. Un multiplexeur (MUX) a 2 entrées de données `a` et `b`, et un signal de sélection `sel`. Si `sel = 1`, quelle entrée est transmise en sortie ?

Q5. Combien de portes NAND faut-il au minimum pour construire un inverseur (NOT) ?

3.11.2 Mini-défi pratique

Complétez cette table de vérité pour la fonction `F(a, b) = AND(OR(a, b), NOT(a))` :

a	b	OR(a,b)	NOT(a)	F
0	0	?	?	?
0	1	?	?	?

a	b	OR(a,b)	NOT(a)	F
1	0	?	?	?
1	1	?	?	?

*Les solutions se trouvent dans le document **Codex_Solutions**.*

3.11.3 Checklist de validation

Avant de passer au chapitre 2, assurez-vous de pouvoir :

- ☐ Expliquer pourquoi NAND est universel
- ☐ Construire NOT, AND, OR à partir de NAND
- ☐ Lire et écrire une table de vérité
- ☐ Comprendre le rôle du MUX (sélection) et du DMUX (routage)
- ☐ Écrire du code HDL simple avec `port map`

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 ?

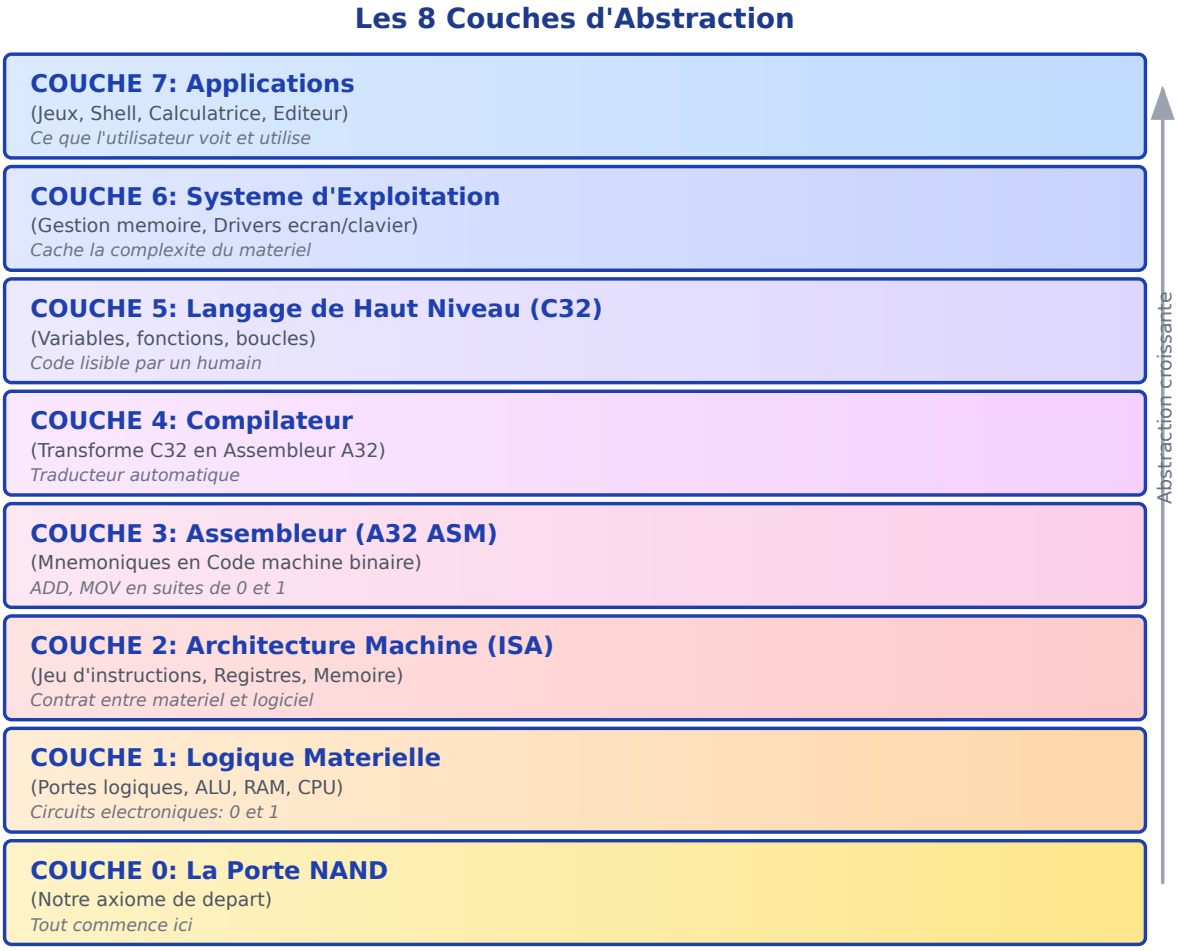


Figure 4.1: Position dans l'architecture

Nous sommes à la Couche 1 : Logique Matérielle (Portes logiques - ALU, RAM, CPU)

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

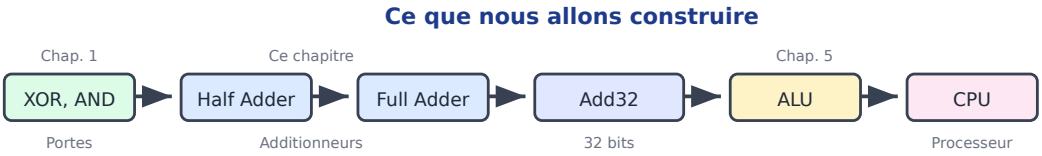


Figure 4.2: Feuille de route : des portes à l'ALU

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 :

```
  4   2   7
  ↓   ↓   ↓
10² 10¹ 10⁰  →  4×100 + 2×10 + 7×1 = 427
```

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

```
Position :   3    2    1    0
Poids    :  2³   2²   2¹   2⁰
```

```
Valeur   :   8    4    2    1

Exemple : 1011₂ = 1×8 + 0×4 + 1×2 + 1×1 = 11₁₀
```

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 (≈ 4 milliards) - **Plage signée** : -2 147 483 648 à 2 147 483 647 (≈ ±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 OK
          (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 + 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(retenue) = 0, retenue 1
- Colonne 2 : 1 + 0 + 1(retenue) = 0, retenue 1
- Colonne 3 : 0 + 0 + 1(retenue) = 1

Résultat : $1000_2 = 8_{10}$ OK

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

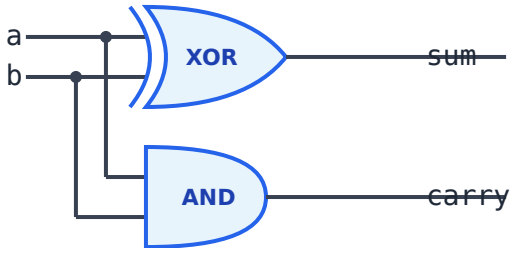


Figure 4.3: Demi-additionneur

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

Figure 4.4: Additionneur complet

4.6.2 Table de vérité

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

a	b	cin	sum	cout
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

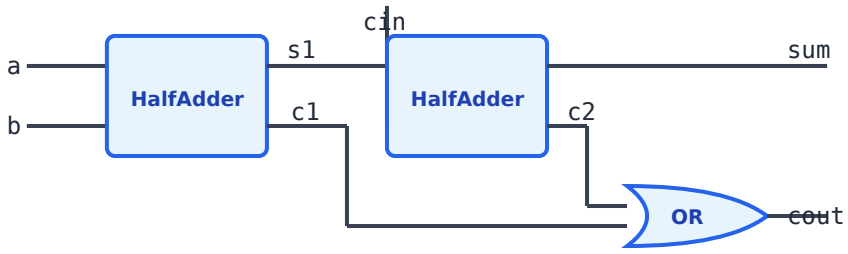


Figure 4.5: Construction du Full Adder

Formules :

- `s1 = XOR(a, b)`
- `sum = XOR(s1, cin)`
- `c1 = AND(a, b)`
- `c2 = AND(s1, cin)`
- `cout = 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

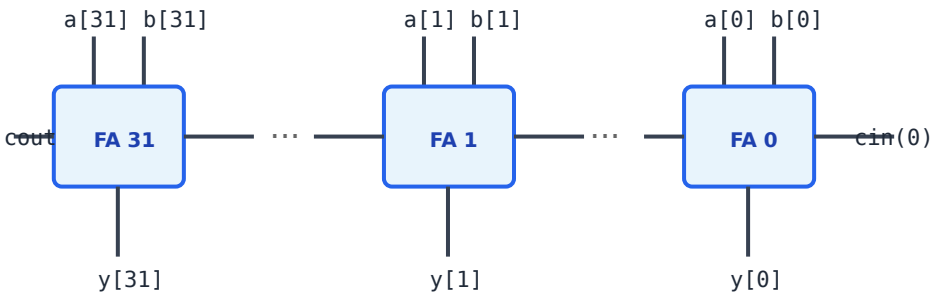


Figure 4.6: Additionneur 32 bits à propagation de retenue

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 + 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

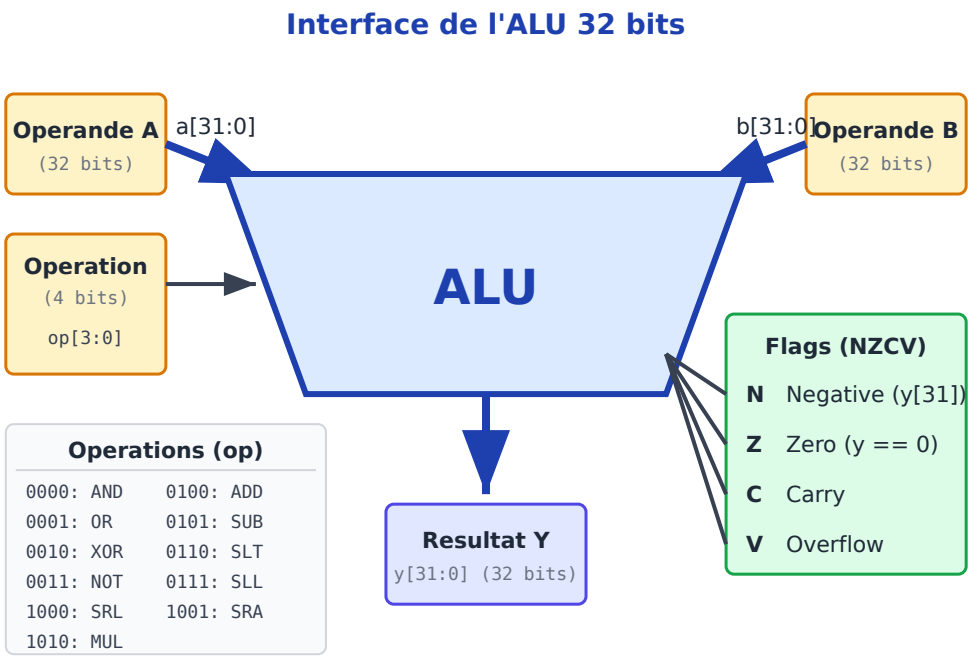


Figure 4.7: Interface de l'ALU

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 ^ b`	OU exclusif bit à bit
0010	SUB	`a - b`	Soustraction
0011	ADD	`a + b`	Addition
0100	ORR	`a | b`	OU logique bit à bit
0101	MOV	`b`	Copie de b (ignore a)
0110	MVN	`~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 + (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
N	Negative	1 si le résultat est négatif (bit 31 = 1)
Z	Zero	1 si le résultat est exactement 0
C	Carry	1 s'il y a eu une retenue (dépassement non-signé)
V	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]) AND (a[31] != y[31])` (pour l'addition)

4.8.7 Quand Utiliser Chaque Drapeau ?

C'est la question cruciale ! Les drapeaux ont des significations différentes selon qu'on travaille en **signé** ou **non-signé**.

4.8.7.1 Arithmétique Non-Signée (unsigned)

Pour des nombres non-signés (0 à 2^{32}-1), utilisez **C** et **Z** :

Condition	Test	Explication
a == b	Z = 1	Résultat de a - b est zéro
a != b	Z = 0	Résultat non nul
a < b	C = 0	Pas de retenue → emprunt
a >= b	C = 1	Retenue présente → pas d'emprunt
a > b	C = 1 ET Z = 0	Pas d'emprunt et non égal
a <= b	C = 0 OU Z = 1	Emprunt ou égal

Pourquoi C ? En soustraction non-signée, si a < b, le résultat "wrappe" (passe par 0) et il n'y a pas de retenue. Si a >= b, la soustraction ne wrappe pas et produit une retenue.

4.8.7.2 Arithmétique Signée (signed)

Pour des nombres signés (-2^{31} à 2^{31}-1), utilisez **N**, **V**, et **Z** :

Condition	Test	Explication
a == b	Z = 1	Résultat de a - b est zéro
a != b	Z = 0	Résultat non nul
a < b	N ≠ V	Voir explication ci-dessous
a >= b	N = V	L'inverse
a > b	Z = 0 ET N = V	Non égal et pas moins que
a <= b	Z = 1 OU N ≠ V	Égal ou moins que

Pourquoi N ≠ V pour "moins que" ?

C'est subtil mais élégant. Après `a - b` :

1. **Sans overflow (V = 0)** : Le signe du résultat est fiable
   - Si N = 1 (négatif) → a < b
   - Si N = 0 (positif) → a >= b
2. **Avec overflow (V = 1)** : Le signe est **inversé** !
   - Si N = 1 mais V = 1 → En réalité a >= b (l'overflow a inversé le signe)

- Si N = 0 mais V = 1 → En réalité a < b

Donc : `a < b` ⟺ (N = 1 et V = 0) OU (N = 0 et V = 1) ⟺ **N ≠ V**

4.8.7.3 Table de Référence des Branchements

Instruction	Signification	Flags testés	Signé/Non-signé
BEQ	Equal	Z = 1	Les deux
BNE	Not Equal	Z = 0	Les deux
BCS/BHS	Carry Set / Higher or Same	C = 1	Non-signé
BCC/BLO	Carry Clear / Lower	C = 0	Non-signé
BHI	Higher	C = 1 et Z = 0	Non-signé
BLS	Lower or Same	C = 0 ou Z = 1	Non-signé
BGE	Greater or Equal	N = V	Signé
BLT	Less Than	N ≠ V	Signé
BGT	Greater Than	Z = 0 et N = V	Signé
BLE	Less or Equal	Z = 1 ou N ≠ V	Signé
BMI	Minus (negative)	N = 1	—
BPL	Plus (positive/zero)	N = 0	—
BVS	Overflow Set	V = 1	—
BVC	Overflow Clear	V = 0	—

4.8.7.4 Exemple Pratique : Comparaison

```
; Comparer deux nombres signés
    CMP R0, R1      ; Calcule R0 - R1, met à jour les flags
    BLT moins       ; Si R0 < R1 (signé), sauter

; Comparer deux nombres non-signés
    CMP R2, R3      ; Calcule R2 - R3, met à jour les flags
    BLO moins_u     ; Si R2 < R3 (non-signé), sauter
```

Attention : Utiliser BLT pour du non-signé ou BLO pour du signé donne des résultats incorrects !

`Exemple : R0 = 0xFFFFFFFF, R1 = 0x00000001`

```
- En non-signé : 4294967295 > 1 → BHI pris, BLO non pris ✓
- En signé : -1 < 1 → BLT pris, BGE non pris ✓
```

4.9 Architecture de l'ALU

Voici comment l'ALU est structurée en interne :

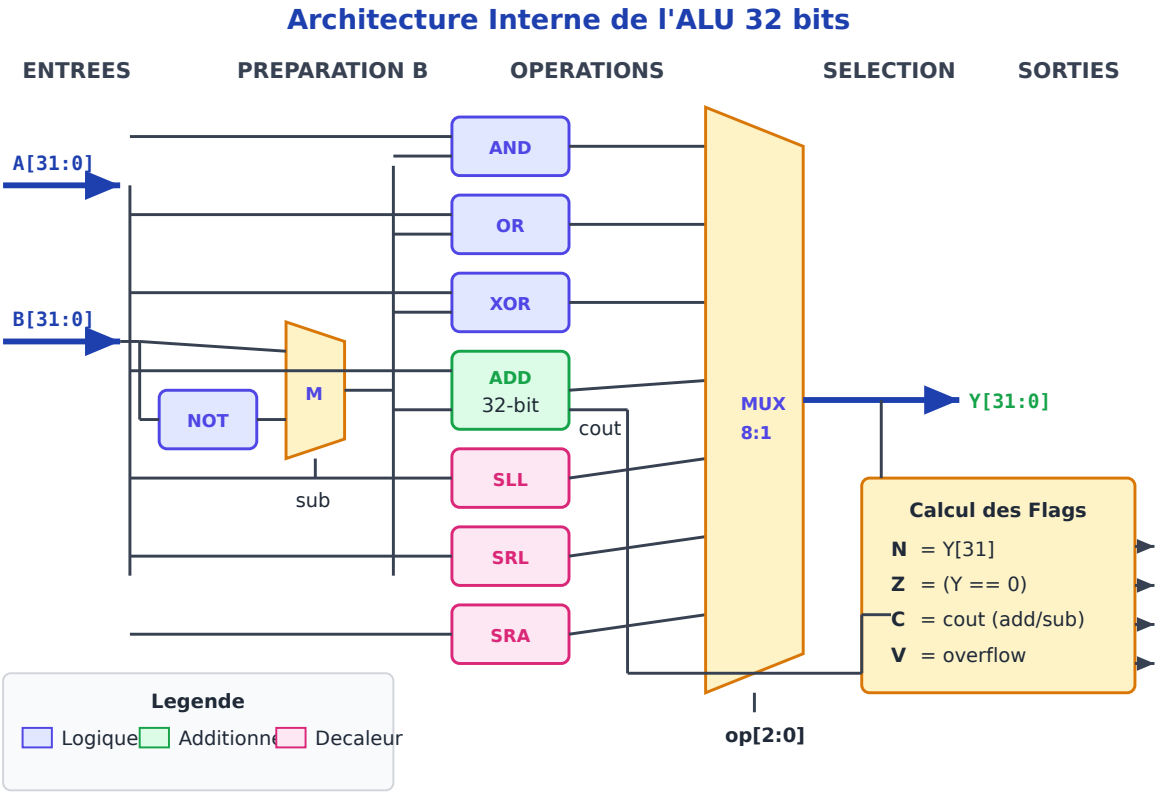


Figure 4.8: Architecture de l'ALU

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** :

   `sub = a + (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 :

/bin/bash: ligne 1: q : commande introuvable - `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 + offset)
`B label`	Calcule la nouvelle adresse (PC + 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 !

4.14 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

4.14.1 Questions de compréhension

Q1. Dans un Half Adder, quelle porte logique calcule le bit de somme ? Et la retenue ?

Q2. Quelle est la représentation en complément à 2 de -1 sur 8 bits ?

Q3. Comment l'ALU effectue-t-elle une soustraction A - B ?

Q4. Que signifient les drapeaux N, Z, C, V ?

Q5. Pourquoi `CMP R1, R2` n'a pas besoin de registre destination ?

4.14.2 Mini-défi pratique

Calculez mentalement le résultat de ces opérations sur 8 bits :

Opération	A (hex)	B (hex)	Résultat	Drapeaux
ADD	0x7F	0x01	?	N=? Z=? V=?
ADD	0xFF	0x01	?	N=? Z=? C=?
SUB	0x05	0x05	?	Z=?

*Les solutions se trouvent dans le document **Codex_Solutions**.*

4.14.3 Checklist de validation

Avant de passer au chapitre 3, assurez-vous de pouvoir :

- ☐ Expliquer la différence entre Half Adder et Full Adder
- ☐ Convertir un nombre négatif en complément à 2
- ☐ Expliquer comment la soustraction utilise l’addition
- ☐ Décrire le rôle de chaque drapeau (N, Z, C, V)
- ☐ Implémenter un additionneur 32 bits en chaînant des Full Adders

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 ?

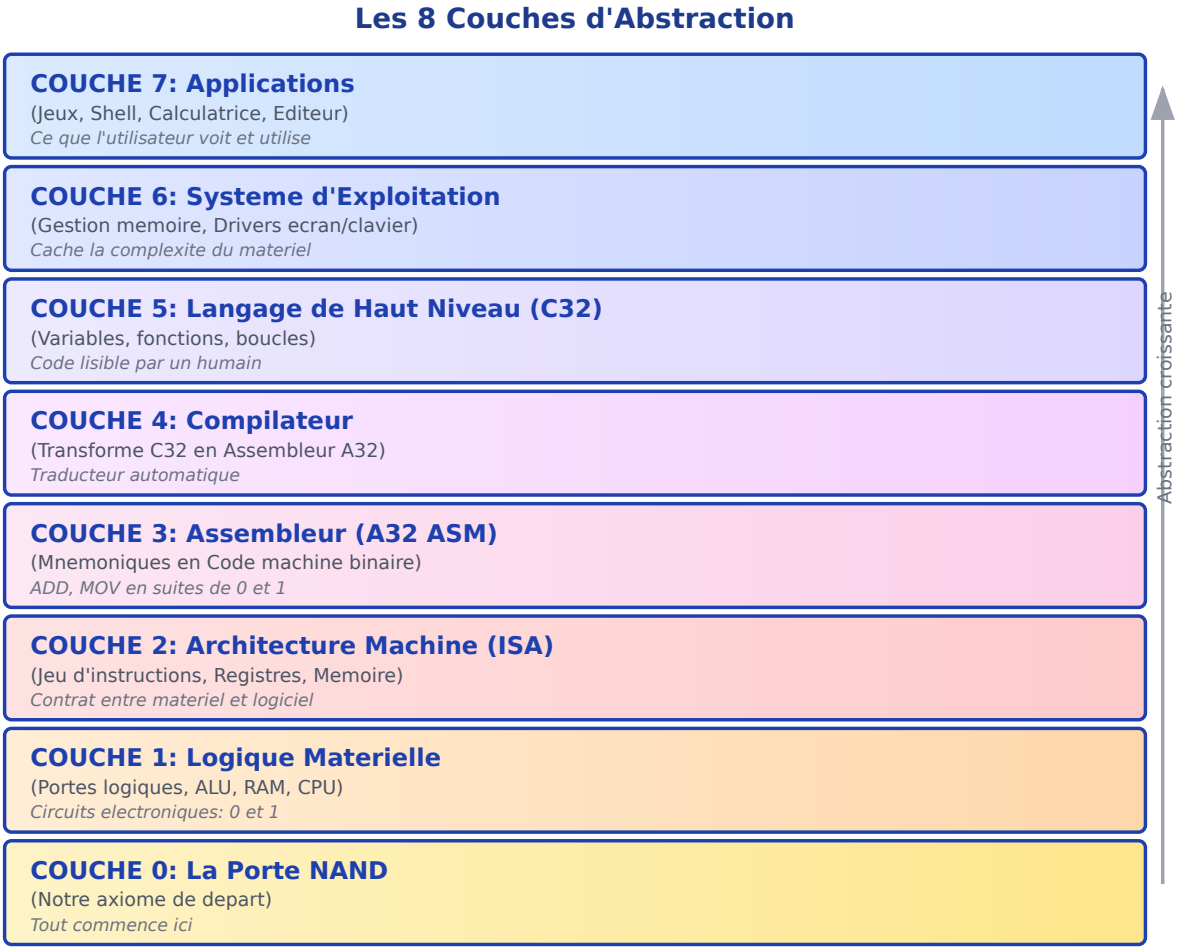


Figure 5.1: Position dans l'architecture

Nous sommes à la Couche 1 : Logique Matérielle (Portes - ALU - MEMOIRE - CPU)

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(entrées)	Sortie = f(entrées, **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 :

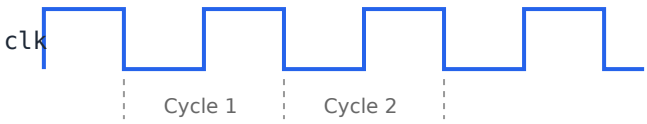


Figure 5.2: Signal d’horloge

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

Figure 5.3: Bascule D (DFF)

- `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.

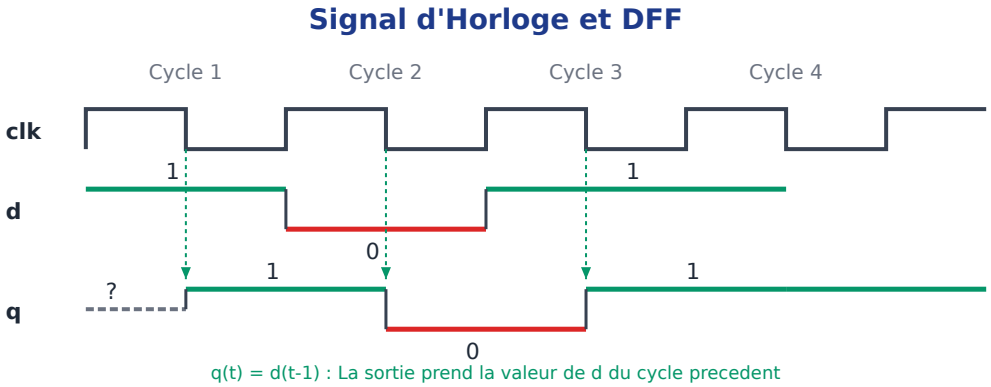


Figure 5.4: Signal d'horloge et DFF

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`)

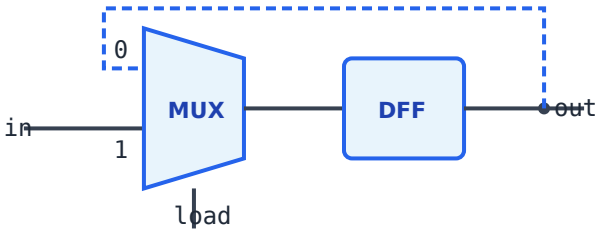


Figure 5.5: Registre 1-bit avec rétroaction

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`.

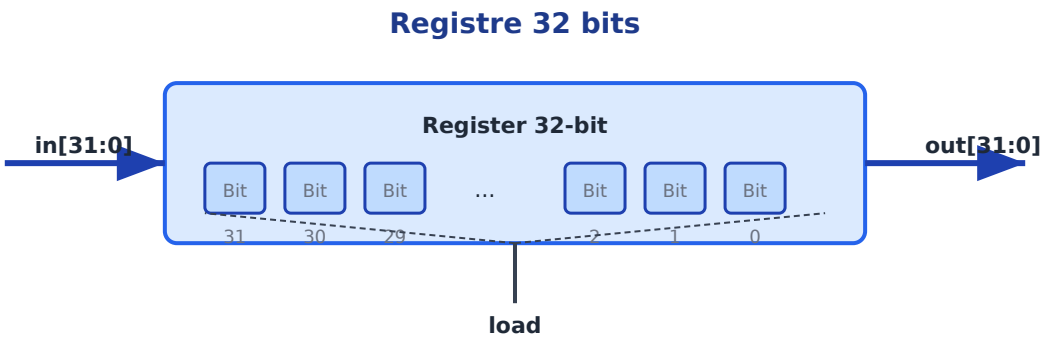


Figure 5.6: Registre 32 bits

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

Figure 5.7: 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)

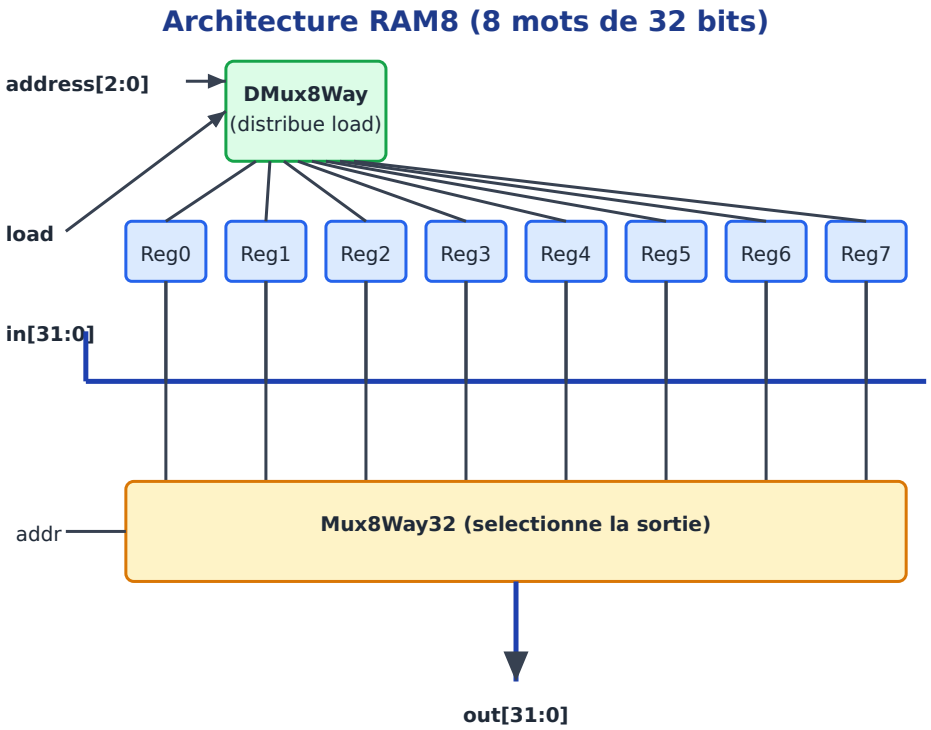


Figure 5.8: Architecture RAM8

5.7.5 Construction hiérarchique de grandes RAMs

Comment construire une RAM64 (64 mots) à partir de RAM8 ?

```
address[5:0] = address[5:3] (3 bits supérieurs) + address[2:0] (3 bits inférieurs)
                    ↓                                       ↓
          Sélectionne laquelle des 8 RAM8    Sélectionne le mot dans cette RAM8
```

On utilise 8 RAM8 :

- Les 3 bits de poids fort (`address[5:3]`) choisissent **quelle RAM8**
- Les 3 bits de poids faible (`address[2:0]`) choisissent **quel mot dans la RAM8**

C'est un pattern récursif qui permet de construire des mémoires de n'importe quelle taille !

5.8 Le Compteur de Programme (PC)

Le **Program Counter** (PC) est peut-être le registre le plus important du CPU. Il contient l'adresse de la prochaine instruction à exécuter.

5.8.1 Pourquoi est-il spécial ?

Après chaque instruction, le PC doit passer à l'instruction suivante. Mais parfois :

- On veut **sauter** à une autre adresse (boucles, conditions)
- On veut **revenir à 0** (redémarrage)

5.8.2 Les trois modes du PC

Priorité	Mode	Condition	Action
1	**Reset**	`reset = 1`	`PC ← 0`
2	**Jump**	`load = 1`	`PC ← in`
3	**Increment**	`inc = 1`	`PC ← PC + 1`
4	**Hold**	sinon	`PC ← PC`

5.8.3 Schéma simplifié

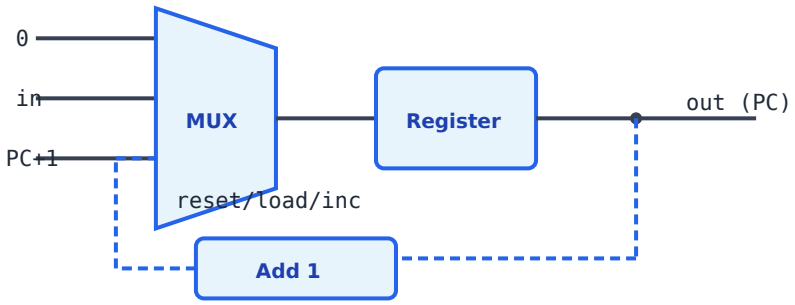


Figure 5.9: Compteur de programme (PC)

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 :

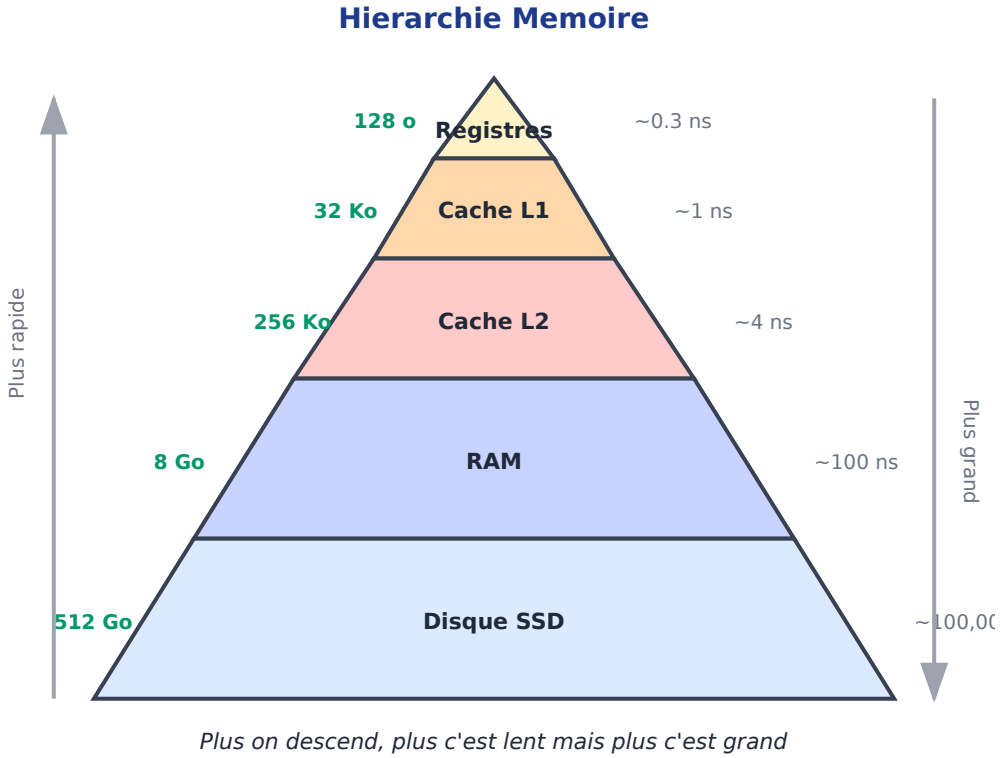


Figure 5.10: Pyramide de la hiérarchie mémoire

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
Registres R0-R15	16 registres 32-bits pour les calculs
RAM	Stockage du programme et des données
PC	Adresse de l'instruction courante
Pile (Stack)	Zone de RAM pour les appels de fonction
Écran	Zone de RAM mappée aux pixels (MMIO)
Clavier	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 !

5.14 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

5.14.1 Questions de compréhension

Q1. Quelle est la différence fondamentale entre un circuit combinatoire et un circuit séquentiel ?

Q2. Pourquoi le signal `load` est-il nécessaire dans un registre ?

Q3. Comment fonctionne l'adressage dans une RAM de 8 mots ?

Q4. Le PC (Program Counter) a plusieurs modes : reset, load, inc, hold. Dans quel ordre de priorité fonctionnent-ils ?

Q5. Pourquoi la RAM est-elle plus lente que les registres ?

5.14.2 Mini-défi pratique

Dessinez le schéma bloc d’un registre 1-bit avec les composants suivants : - 1 MUX 2-vers-1 - 1 DFF

Indice : La sortie du DFF retourne vers une entrée du MUX.

*Les solutions se trouvent dans le document **Codex_Solutions**.*

5.14.3 Checklist de validation

Avant de passer au chapitre 4, assurez-vous de pouvoir :

- ☐ Expliquer pourquoi l’horloge est nécessaire (synchronisation)
- ☐ Décrire le comportement d’une DFF : `q(t) = d(t-1)`
- ☐ Construire un registre 1-bit avec MUX + DFF
- ☐ Expliquer comment RAM8 utilise DMux et Mux pour l’adressage
- ☐ Décrire les priorités du PC (reset > load > inc > hold)

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 ?

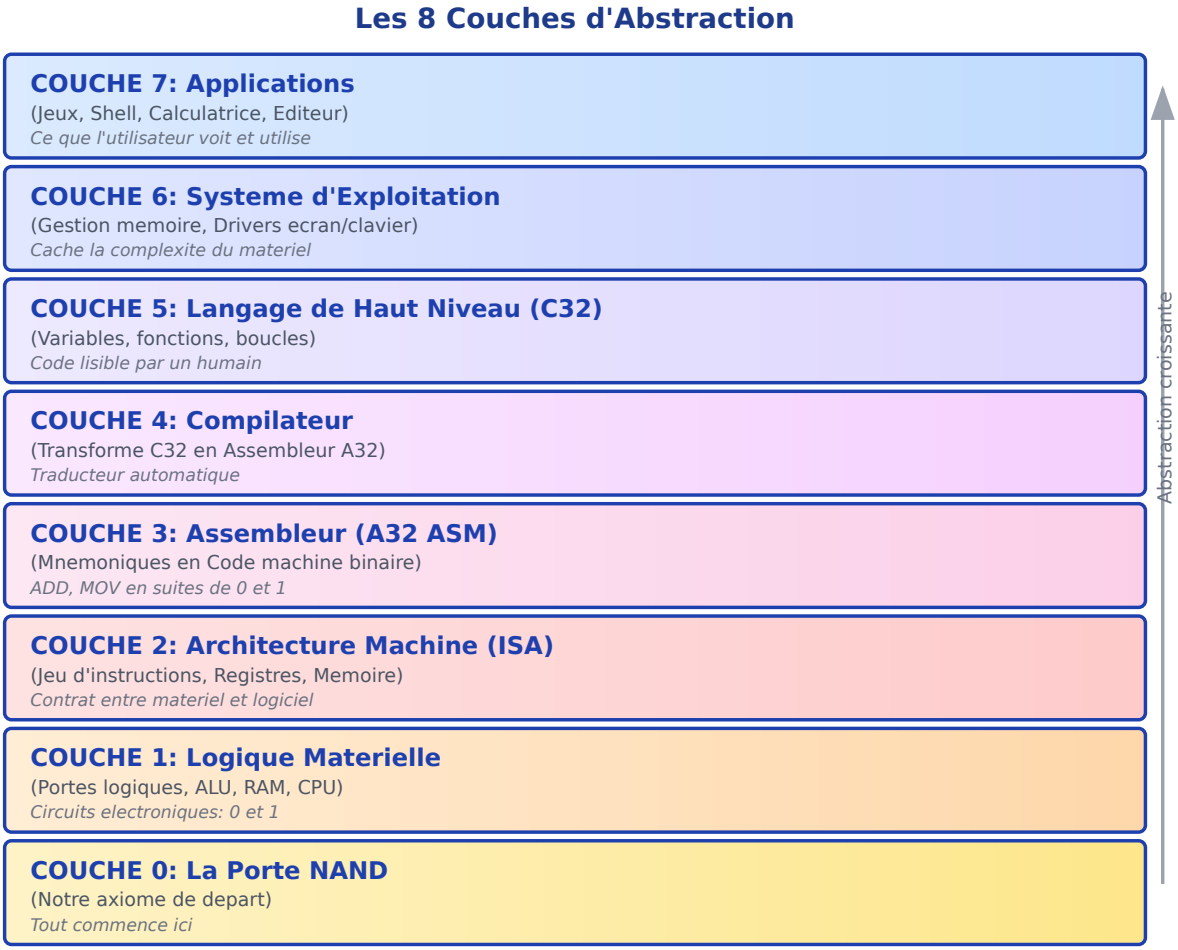


Figure 6.1: Position dans l'architecture

Nous sommes à la Couche 2 : Architecture Machine (ISA) - Le contrat entre matériel et logiciel

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 ≈ 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 :

Figure 6.2: Cycle Fetch-Decode-Execute

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

Figure 6.3: Banc de registres

6.5.2 Rôles des registres

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

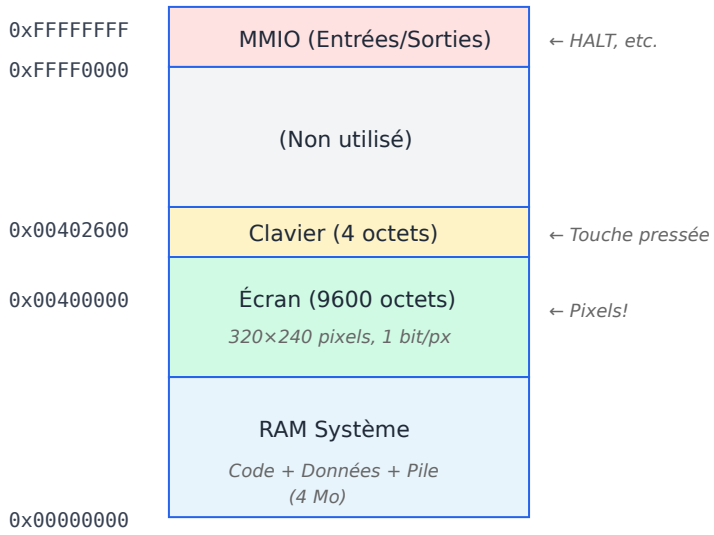


Figure 6.4: Carte mémoire Codex

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 :

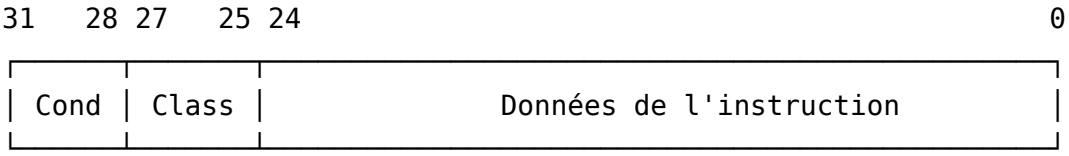
- Adresse : `0x00402600`
- Lire cette adresse donne le code ASCII de la touche pressée (ou 0)

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)

Code	Suffixe	Condition	Signification
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 ≠ V	Plus petit (signé)
1100	GT	Z=0 et N=V	Plus grand (signé)
1101	LE	Z=1 ou N≠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

Instruction	Opération	Exemple
`EOR`	XOR	`EOR R1, R2, R3` → R1 = R2 ^ R3
`MOV`	Copie	`MOV R1, R2` → R1 = R2
`MVN`	Copie inversée	`MVN R1, R2` → R1 = ~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]    ; Registre : 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)

Instruction	Action
`B.EQ label`	Saut si égal (Z=1)
`B.NE label`	Saut si différent (Z=0)
`B.LT label`	Saut si plus petit (signé)
`B.GT label`	Saut si plus grand (signé)

Le mystère de `BL` :

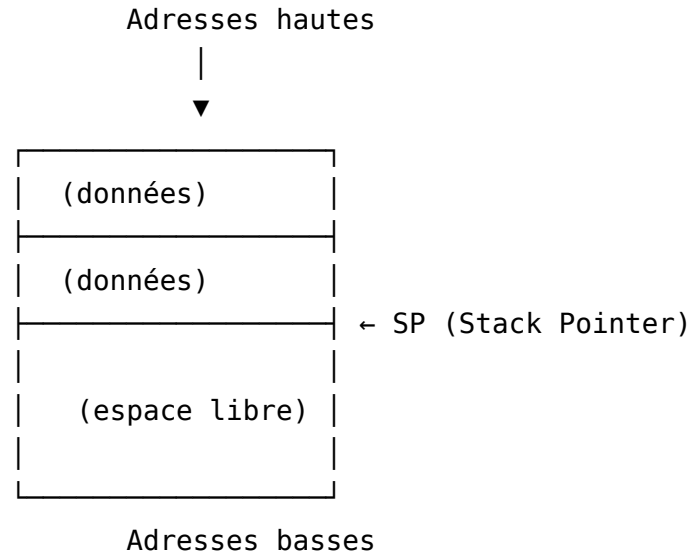
```
main:
    BL ma_fonction    ; 1. Sauvegarde PC+4 dans LR
                      ; 2. Saute à ma_fonction
    ; ... (on revient ici après le retour)

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)
`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 !

6.14 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

6.14.1 Questions de compréhension

Q1. Qu'est-ce qu'une architecture RISC et pourquoi A32 en est une ?

Q2. Pourquoi y a-t-il 16 registres (R0-R15) et pas 8 ou 32 ?

Q3. Que font les registres spéciaux R13, R14, R15 ?

Q4. Comment fonctionne l'exécution conditionnelle (ex: `ADD.EQ`) ?

Q5. Qu'est-ce que le Memory-Mapped I/O (MMIO) ?

6.14.2 Mini-défi pratique

Écrivez un programme assembleur qui : 1. Met 10 dans R0 2. Met 3 dans R1 3. Calcule R0 - R1 et stocke dans R2 4. Si le résultat est positif, met 1 dans R3, sinon 0

*Les solutions se trouvent dans le document **Codex_Solutions**.*

6.14.3 Checklist de validation

Avant de passer au chapitre 5, assurez-vous de pouvoir :

- ☐ Expliquer la différence entre RISC et CISC
- ☐ Décrire le rôle de chaque registre spécial (SP, LR, PC)
- ☐ Écrire un programme simple en assembleur A32
- ☐ Utiliser les conditions (EQ, NE, GT, LT, etc.)
- ☐ Expliquer le cycle Fetch-Decode-Execute

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 ?

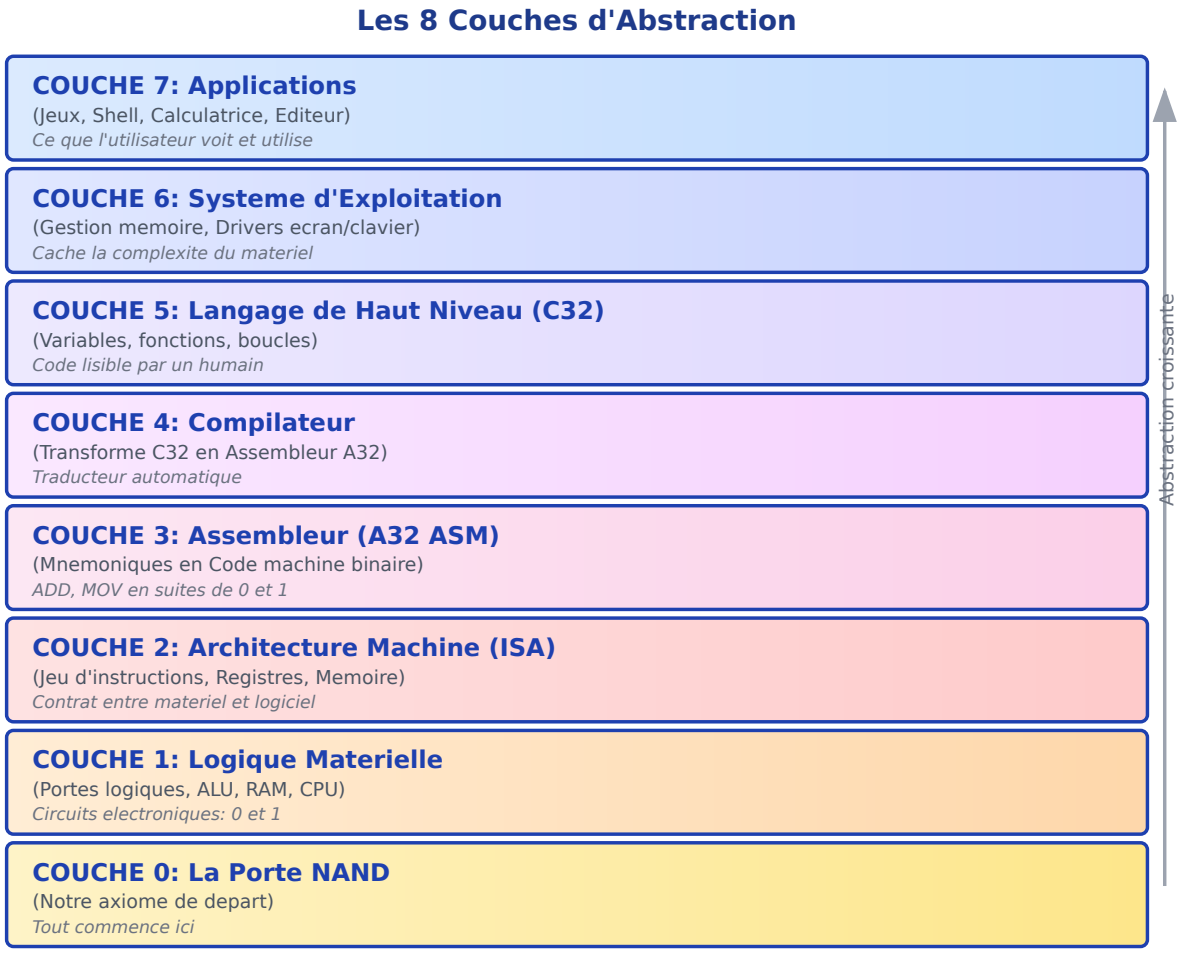


Figure 7.1: Position dans l'architecture

Nous sommes à la Couche 1 : CPU - L'Aboutissement (ALU + RAM + Registres)

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** :

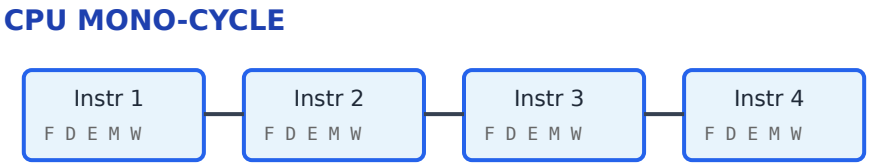


Figure 7.2: CPU Mono-cycle

Chaque instruction traverse TOUTES les étapes en UN 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** :

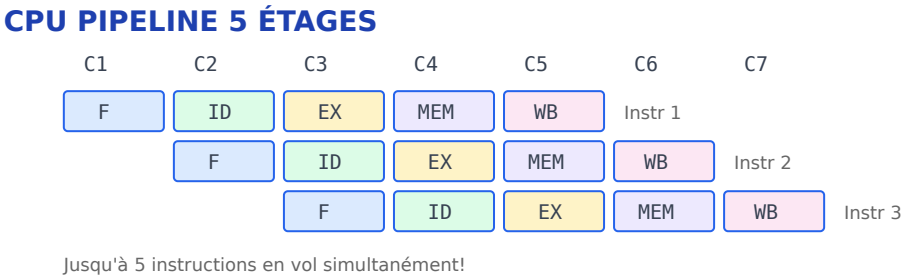


Figure 7.3: CPU Pipeline 5 étages

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)

Fichier	Rôle
`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)
Objectif	Exécuter des programmes	Apprendre le hardware
Complexité	Simple	Réaliste
Performance	1 instr/cycle	Jusqu'à 1 instr/cycle (avec hazards)
Hazards	Aucun	Gérés (stall + forward)
Utilisation	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.

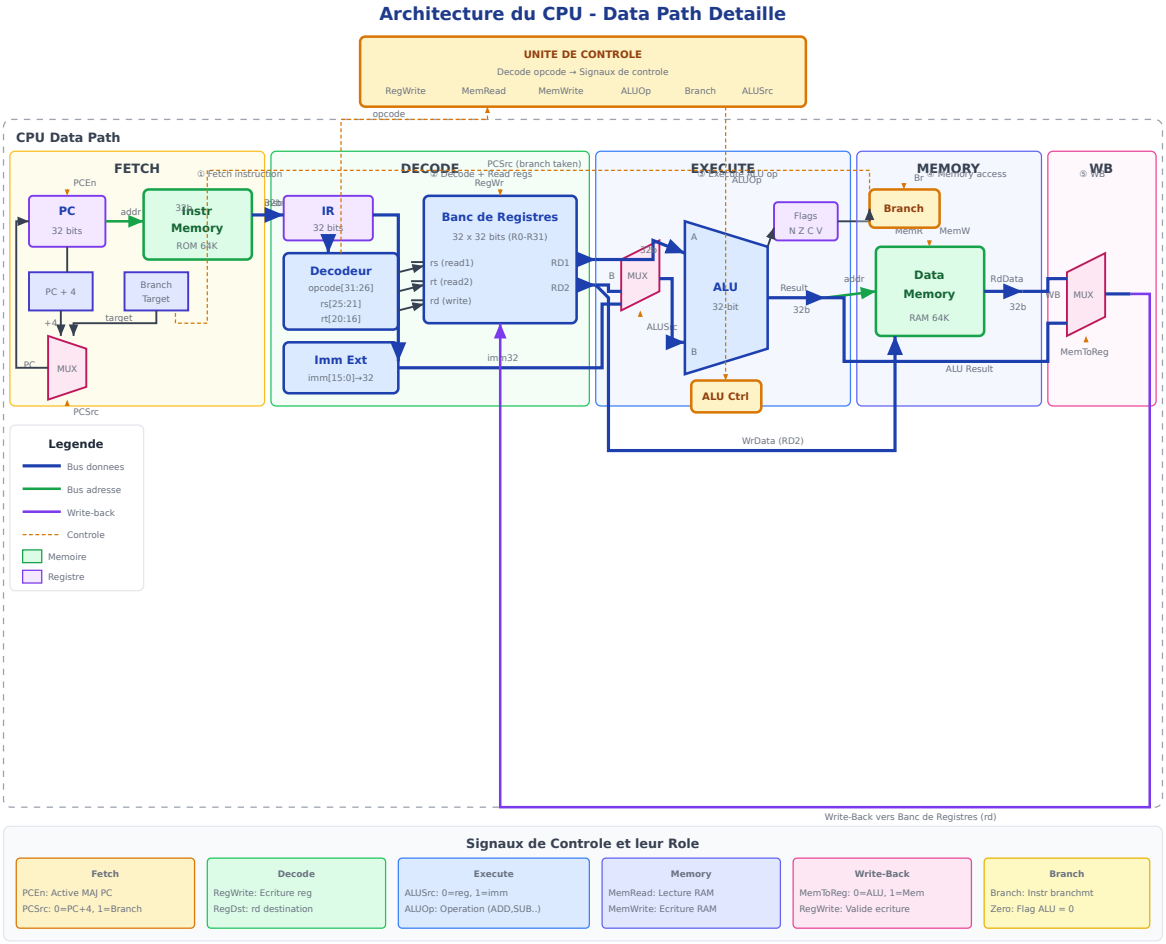


Figure 7.4: Architecture du CPU (Data Path)

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 ← PC + 4 (instruction suivante)
- `load = 1` : PC ← adresse de branchement
- `reset = 1` : PC ← 0 (redémarrage)

7.5.2 2. Le Décodeur (Decoder)

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

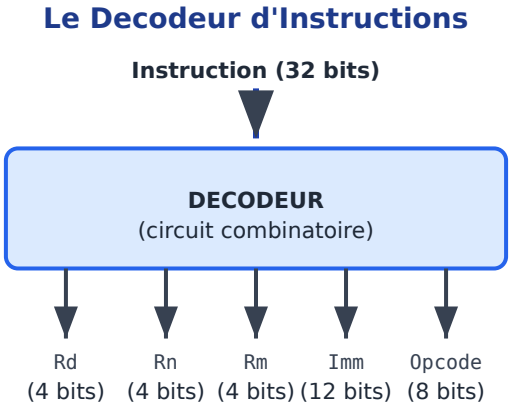


Figure 7.5: Le décodeur d'instructions

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.

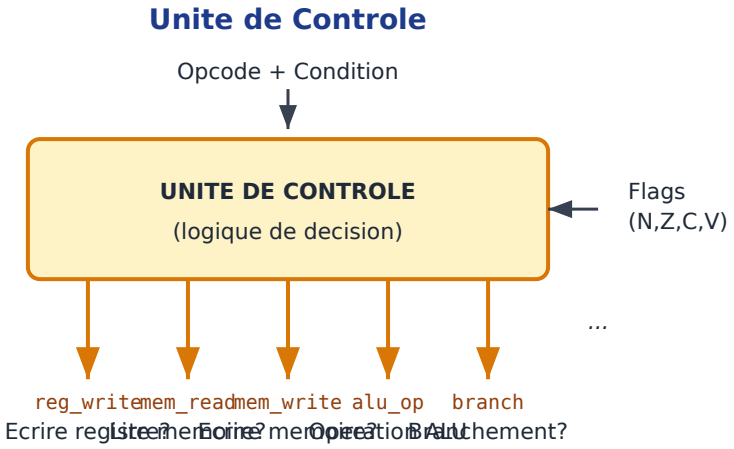


Figure 7.6: Unité de contrôle

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

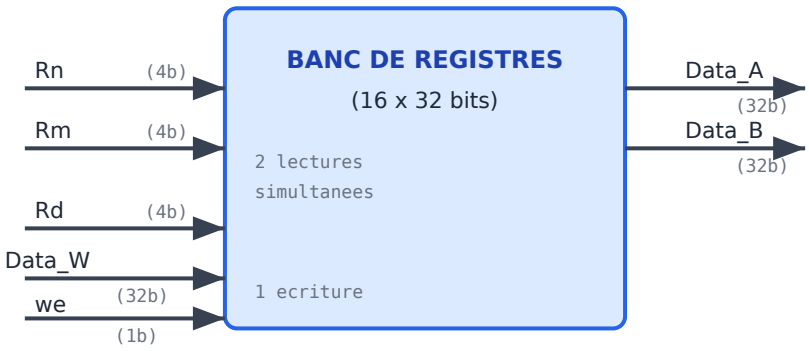


Figure 7.7: Interface du banc de registres

7.5.6 6. Les Multiplexeurs

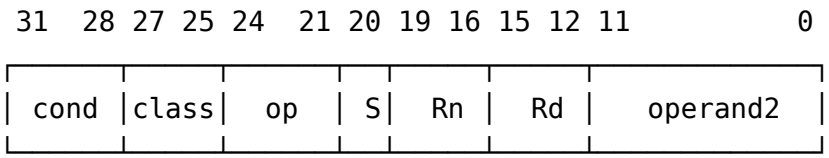
Les multiplexeurs routent les données entre les composants :

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

7.6 Du Format d'Instruction au Hardware

C'est une question fondamentale : **comment chaque bit de l'instruction devient-il une action dans le hardware ?**

7.6.1 Rappel : Format d'une Instruction A32



7.6.2 Mapping ISA → Hardware

Voici comment chaque champ de l'instruction contrôle le hardware :

Bits	Champ	Hardware	Fonction
[31:28]	`cond`	**Cond Check**	Compare avec NZCV, génère `cond_ok`
[27:25]	`class`	**Decoder**	Distingue ALU/Mémoire/Branch/etc.
[24:21]	`op`	**Control Unit**	Génère `ALU_op`, `reg_write`, `mem_write`
[20]	`S`	**Flag Register**	Active la mise à jour des flags
[19:16]	`Rn`	**RegFile port A**	Adresse du registre source 1
[15:12]	`Rd`	**RegFile port W**	Adresse du registre destination
[11:0]	`operand2`	**Imm Extender** ou **RegFile port B**	Deuxième opérande

7.6.3 Exemple Détaillé : ADD R1, R2, R3

L'instruction `ADD R1, R2, R3` en binaire :

```
1110 000 0100 0 0010 0001 00000000 0011
|    |   |    | |    |    |         |
```

```
│    │   │    │ │    │    │         └── Rm = R3 (0011)
│    │   │    │ │    │    └── padding (ignoré)
│    │   │    │ │    └── Rd = R1 (0001)
│    │   │    │ └── Rn = R2 (0010)
│    │   │    └── S = 0 (ne met pas à jour les flags)
│    │   └── opcode = 0100 (ADD)
│    └── class = 000 (ALU avec registre)
└── cond = 1110 (AL = Always)
```

Parcours dans le hardware :

1. **Cond Check** : `cond = 1110` (AL) → `cond_ok = 1` (toujours vrai)
2. **Decoder** : `class = 000` → Instruction ALU avec registre
   - `reg_write = 1` (on écrit dans un registre)
   - `mem_read = 0`, `mem_write = 0` (pas d'accès mémoire)
   - `ALU_src = 0` (opérande 2 = registre, pas immédiat)
3. **Control Unit** : `op = 0100` → `ALU_op = ADD`
4. **RegFile** :
   - Port A lit `Rn = R2` → `Data_A = valeur de R2`
   - Port B lit `Rm = R3` → `Data_B = valeur de R3`
5. **ALU** : Calcule `Data_A + Data_B`
6. **Writeback** : Écrit le résultat dans `Rd = R1`

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

L'instruction `LDR R0, [R1, #8]` :

```
1110 010 1 1001 0001 0000 000000001000
│    │   │ │    │    │    │
│    │   │ │    │    │    └── offset = #8
│    │   │ │    │    └── Rd = R0
│    │   │ │    └── Rn = R1
│    │   │ └── opcode (bits) = 1001 (LDR avec offset positif)
│    │   └── L = 1 (Load, pas Store)
│    └── class = 010 (Mémoire)
└── cond = 1110 (AL)
```

Parcours :

1. **Decoder** : `class = 010` → Instruction mémoire

- `mem_read = 1` (on lit la mémoire)
- `ALU_src = 1` (opérande 2 = immédiat)
- `ALU_op = ADD` (pour calculer l'adresse)

2. **RegFile** : Lit `Rn = R1` → adresse de base
3. **Imm Extender** : Extrait offset = 8
4. **ALU** : Calcule `R1 + 8` → adresse effective
5. **Mémoire** : Lit MEM[adresse] → valeur
6. **Writeback** : `Rd = R0` reçoit la valeur lue

7.6.5 Exemple Détaillé : BEQ label

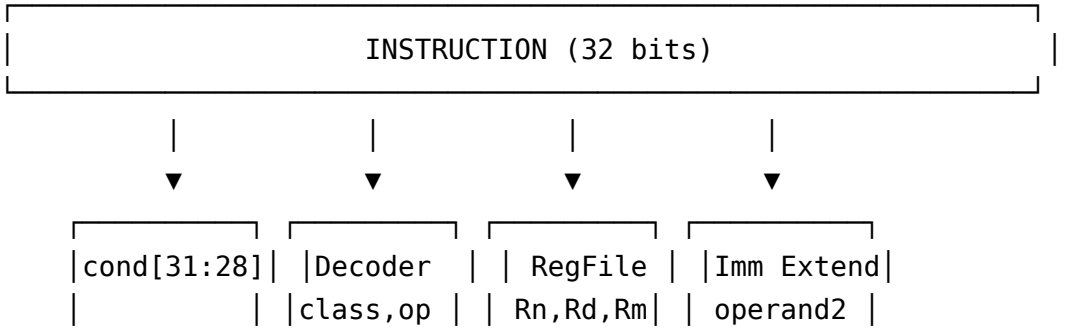
L'instruction `BEQ label` (avec offset de 10 instructions) :

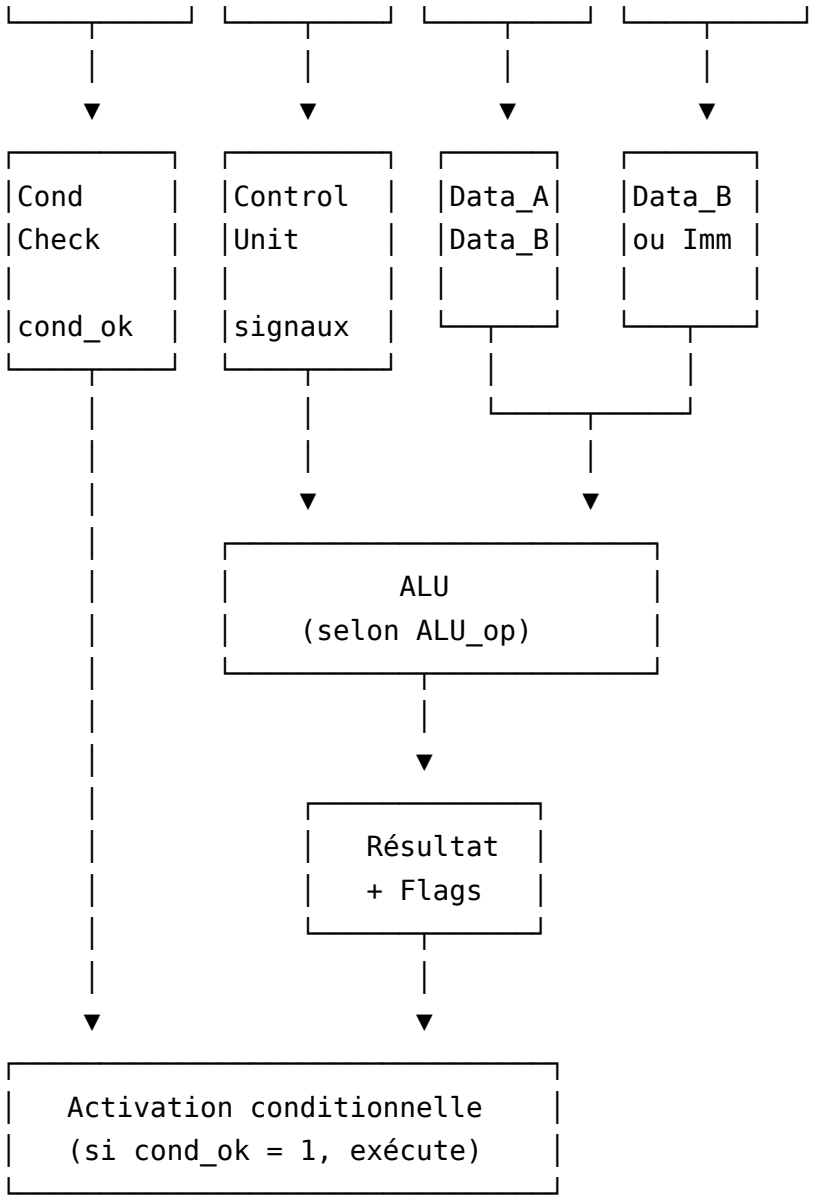
```
0000 101 0 00000000000000000001010
│    │   │ │
│    │   │ └── offset = 10 (en nombre d'instructions)
│    │   └──── L = 0 (Branch, pas Branch-Link)
│    └──────── class = 101 (Branch)
└───────────── cond = 0000 (EQ = Equal, Z=1)
```

Parcours :

1. **Cond Check** : `cond = 0000` (EQ) → `cond_ok = Z` (prend le flag Z actuel)
2. **Decoder** : `class = 101` → Instruction de branchement
   - `PC_src = branch_taken` (si cond_ok = 1)
   - `reg_write = 0` (pas d'écriture registre)
3. **Calcul de l'adresse cible** : `PC + 4 + (offset × 4)` = `PC + 4 + 40`
4. **MUX PC** : Si `cond_ok = 1` (Z était à 1), PC ← adresse cible

7.6.6 Schéma Récapitulatif





7.7 Le Cycle d'Exécution en Détail

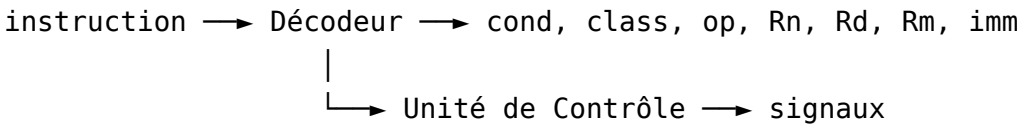
Notre CPU est **single-cycle** : chaque instruction s'exécute en un seul cycle d'horloge.

7.7.1 Phase 1 : Fetch (Récupération)

```
PC ⟶ Mémoire Instructions ⟶ instruction (32 bits)
```

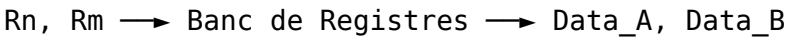
Le PC envoie son adresse à la mémoire d'instructions. La mémoire renvoie les 32 bits de l'instruction.

7.7.2 Phase 2 : Decode (Décodage)



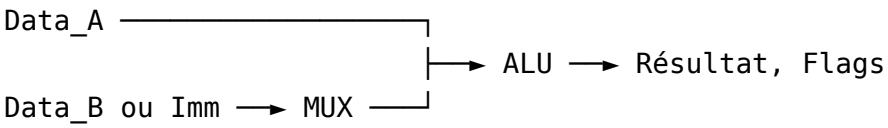
Le décodeur découpe l'instruction. L'unité de contrôle décide quoi activer.

7.7.3 Phase 3 : Register Read (Lecture des registres)



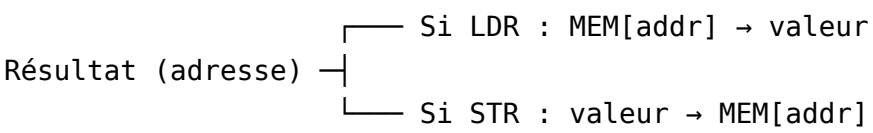
Les valeurs des registres sources sont lues.

7.7.4 Phase 4 : Execute (Exécution)



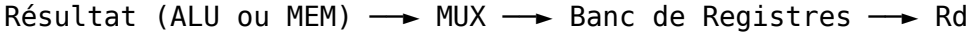
L'ALU effectue l'opération. Les drapeaux sont mis à jour (si S=1).

7.7.5 Phase 5 : Memory (Accès mémoire)



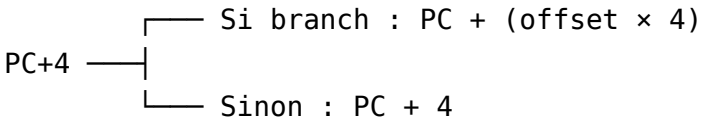
Pour les instructions LDR/STR, on accède à la mémoire de données.

7.7.6 Phase 6 : Writeback (Écriture)



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

7.7.7 Phase 7 : PC Update



Le PC est mis à jour pour le prochain cycle.

7.8 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;
```

7.9 Exercices Pratiques

7.9.1 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 !	[****]

7.9.2 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 ≠ 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 !

7.9.3 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
```

7.10 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.

7.10.1 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

7.10.2 Fonctionnalités du Visualizer

7.10.2.1 Vue Pipeline

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

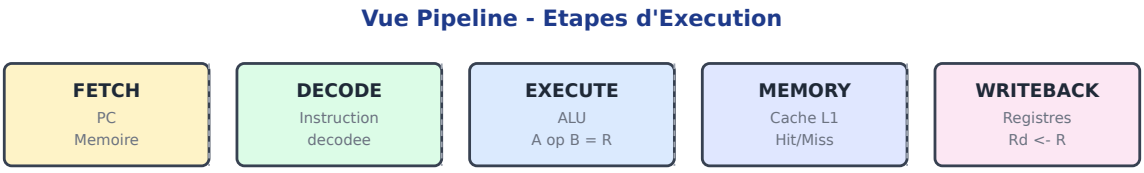


Figure 7.8: Vue Pipeline - Étapes d'exécution

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

7.10.2.2 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.

7.10.2.3 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.

7.10.2.4 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

7.10.2.5 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

7.10.3 Les Démos Intégrées

Le Visualizer inclut 7 démos prêtes à l'emploi :

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

7.10.4 Contrôles

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

7.10.5 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

7.10.6 Exercice Pratique

Utilisez le Visualizer pour observer ces comportements :

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

---

7.11 Conseils de Débogage

7.11.1 Le PC reste à 0 ?

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

7.11.2 Les branchements ne marchent pas ?

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

7.11.3 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)

7.11.4 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)

7.12 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.

7.12.1 Pourquoi le CPU Single-Cycle est Lent

7.12.1.1 Le problème de la chaîne critique

Dans notre CPU single-cycle, une instruction doit traverser **tous** les composants en un seul cycle :

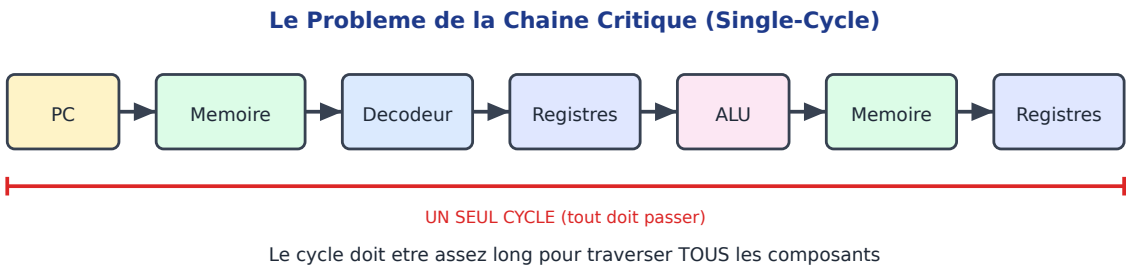


Figure 7.9: Le problème de la chaîne critique

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.

7.12.1.2 Une analogie : La laverie

Imaginez que vous avez 4 lessives à faire. Chaque lessive a 4 étapes : laver (30 min), sécher (30 min), plier (30 min), ranger (30 min).

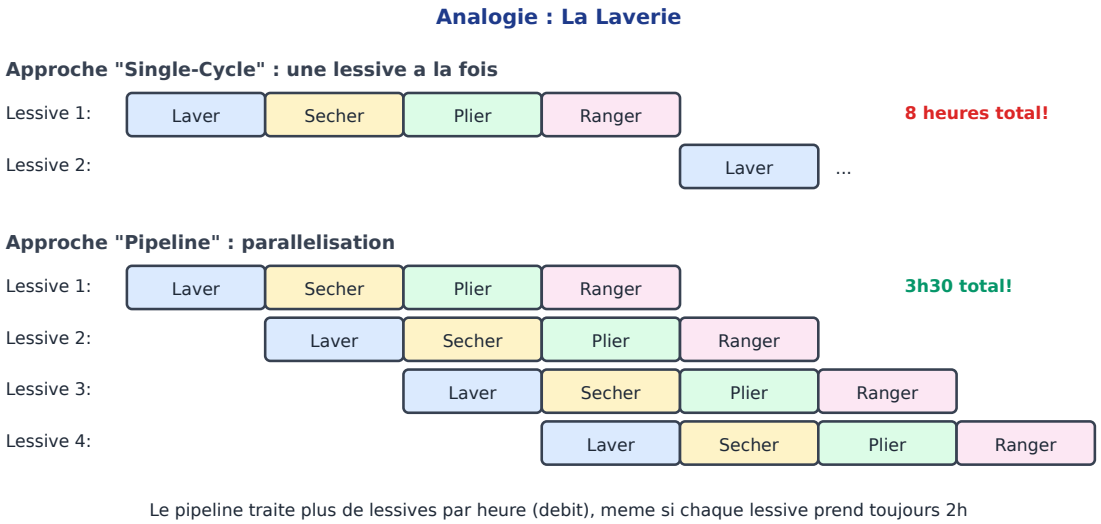


Figure 7.10: Analogie de la laverie

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

7.12.2 Le Pipeline à 5 Étages

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

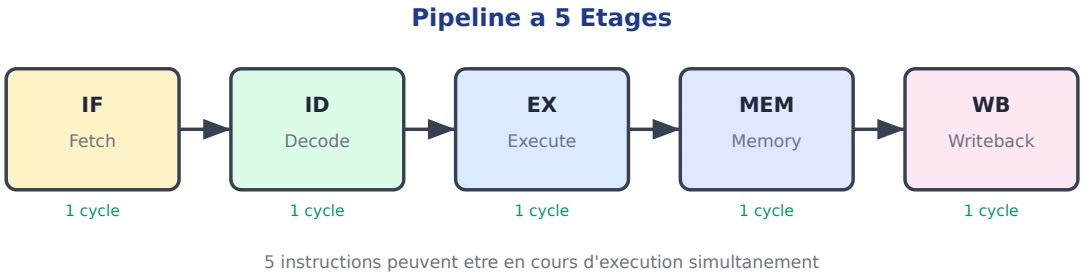


Figure 7.11: Pipeline à 5 étages

7.12.2.1 Étape 1 : IF (Instruction Fetch)

But : Aller chercher l'instruction en mémoire.

```
┌─────────────────────────────────────────────┐
│  IF - Instruction Fetch                     │
│                                             │
│    PC ──────► Mémoire ──────► Instruction   │
│    │          Instructions    (32 bits)     │
│    │                                        │
│    └──────► PC + 4 (préparé pour la suite)  │
│                                             │
└─────────────────────────────────────────────┘
```

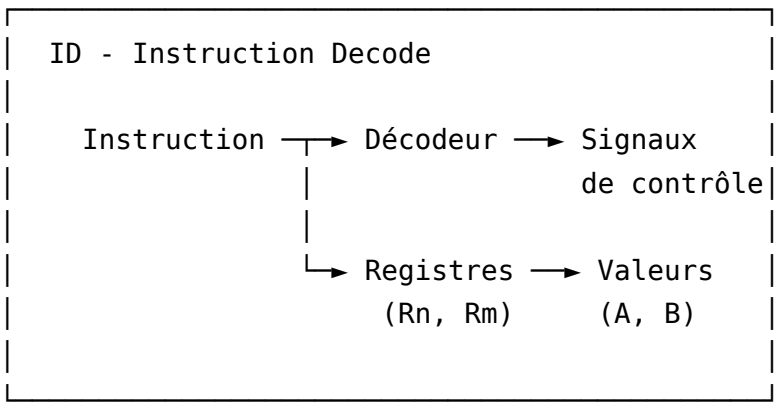
Ce qui se passe :

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.

7.12.2.2 Étape 2 : ID (Instruction Decode)

But : Comprendre l'instruction et lire les registres sources.



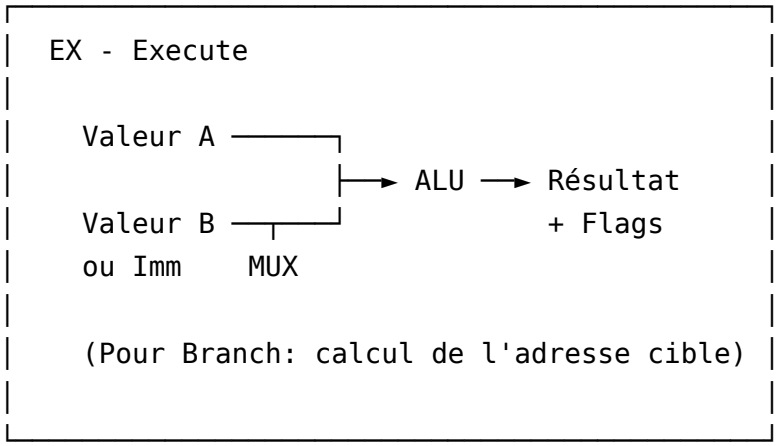
Ce qui se passe :

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.

7.12.2.3 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.

7.12.2.4 Étape 4 : MEM (Memory Access)

But : Lire ou écrire en mémoire (pour LDR/STR seulement).

```
┌─────────────────────────────────────────────┐
│  MEM - Memory Access                        │
│                                             │
│    Si LDR:  Adresse ──► Mémoire ──► Donnée  │
│                         Données             │
│                                             │
│    Si STR:  Adresse, Donnée ──► Mémoire     │
│                                             │
│    Sinon:   (le résultat ALU passe juste)   │
│                                             │
└─────────────────────────────────────────────┘
```

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.

7.12.2.5 Étape 5 : WB (Write Back)

But : Écrire le résultat dans le registre destination.

```
┌─────────────────────────────────────────────┐
│  WB - Write Back                            │
│                                             │
│    Résultat ──► MUX ──► Banc de Registres   │
│    (ALU ou MEM)          ↓                  │
│                         Rd                  │
│                                             │
└─────────────────────────────────────────────┘
```

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

7.12.3 Visualisation du Pipeline en Action

Voici comment 5 instructions traversent le pipeline :

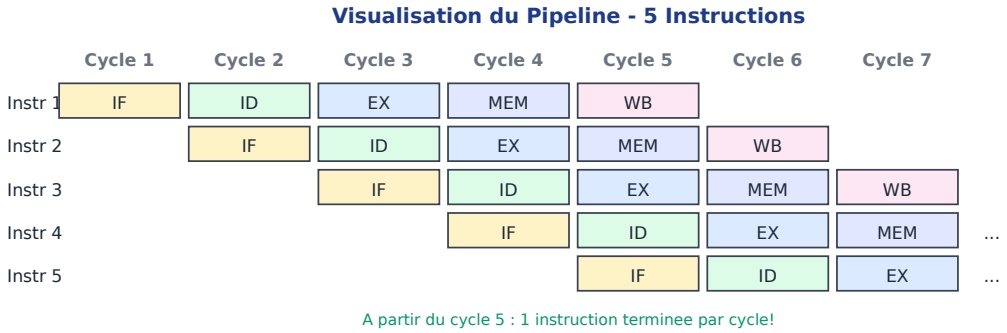


Figure 7.12: Visualisation du timing du pipeline

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 = 1ns)
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	**500 ns**
Pipeline	1 ns	104 cycles	**104 ns**

Le pipeline est **~5× plus rapide** !

7.12.4 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**.

7.12.4.1 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)

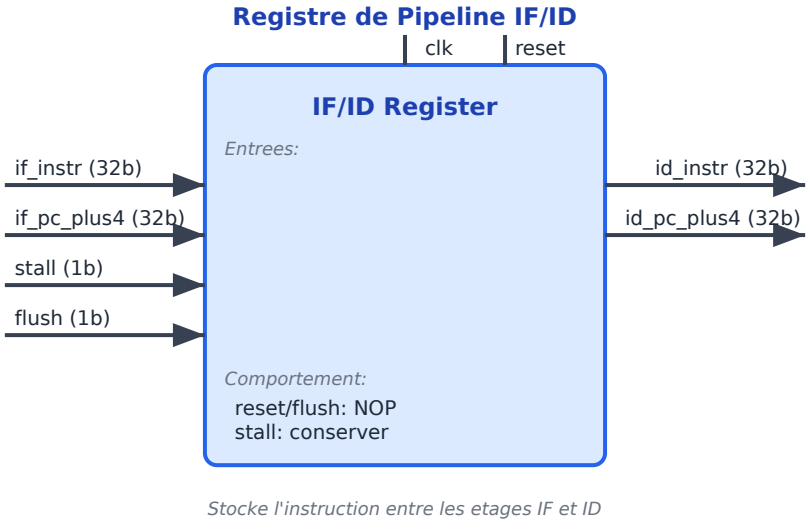


Figure 7.13: Registre de pipeline IF/ID

7.12.5 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**.

7.12.5.1 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!)
```

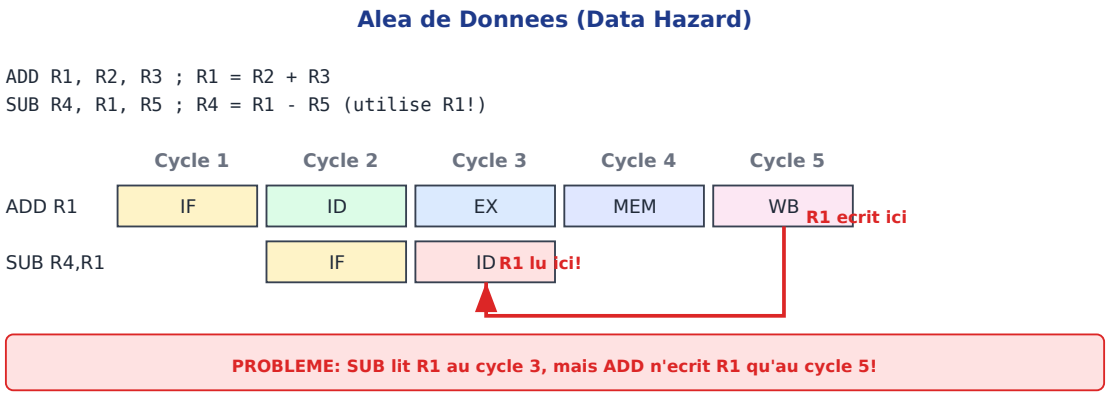


Figure 7.14: Aléa de données (Data Hazard)

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 !

7.12.5.2 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         │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

7.12.5.3 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            │
│                                                               │
└───────────────────────────────────────────────────────────────┘
```

7.12.5.4 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).

7.12.6 Architecture Complète du CPU Pipeline

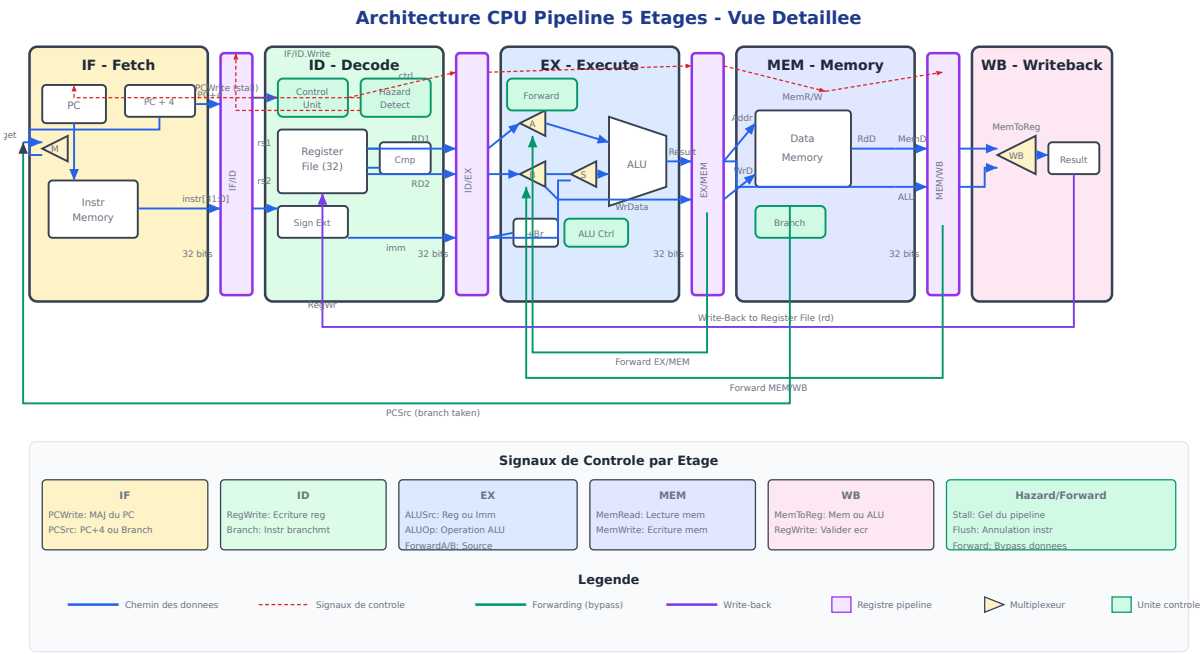


Figure 7.15: Architecture du CPU Pipeline

7.12.7 Exercices Pratiques : Projet 6

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

7.12.7.1 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;
```

7.12.7.2 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;
```

7.12.7.3 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;
```

7.12.7.4 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).

7.12.8 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
```

7.12.9 Résumé : Pipeline vs Single-Cycle

Aspect	Single-Cycle	Pipeline
Instructions en parallèle	1	Jusqu'à 5
Throughput	1 instr / 5 unités de temps	1 instr / 1 unité de temps
Complexité	Simple	Plus complexe

Aspect	Single-Cycle	Pipeline
Aléas	Aucun	Data hazards, Control hazards
Composants supplémentaires	Aucun	Registres pipeline, Hazard Detect, Forward Unit

---

7.12.10 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 !

7.13 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)

7.13.1 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 !

7.14 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 !

7.15 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

7.15.1 Questions de compréhension

Q1. Quelles sont les 5 étapes du cycle d'exécution d'une instruction ?

Q2. Quel est le rôle du décodeur d'instructions ?

Q3. Pourquoi les branchements conditionnels dépendent-ils des drapeaux NZCV ?

Q4. Dans un CPU pipeline 5 étages, qu'est-ce qu'un hazard de données ?

Q5. Quelle est la différence entre un CPU single-cycle et un CPU pipeliné ?

7.15.2 Mini-défi pratique

Tracez l’exécution de cette instruction dans le CPU :

```
ADD R1, R2, R3
```

Décrivez ce qui se passe à chaque étape (Fetch, Decode, Execute, Memory, Writeback).

*Les solutions se trouvent dans le document **Codex_Solutions**.*

7.15.3 Checklist de validation

Avant de passer au chapitre 6, assurez-vous de pouvoir :

- ☐ Décrire les 5 étapes d’exécution d’une instruction
- ☐ Expliquer le rôle du décodeur et de l’unité de contrôle
- ☐ Tracer le chemin des données pour ADD, LDR, et B
- ☐ Comprendre pourquoi les MUX sont essentiels (choix des sources)
- ☐ Expliquer le concept de pipeline et ses avantages

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 ?

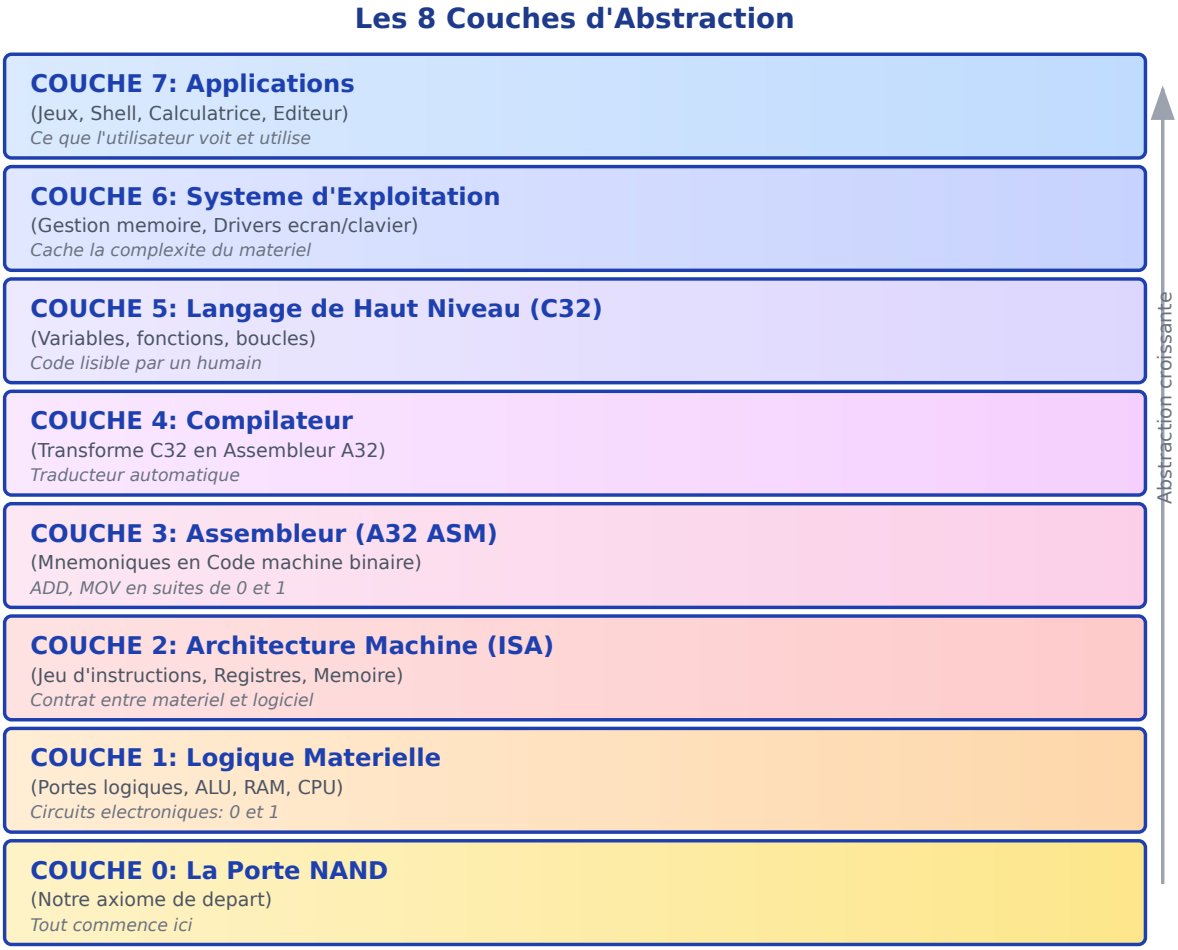


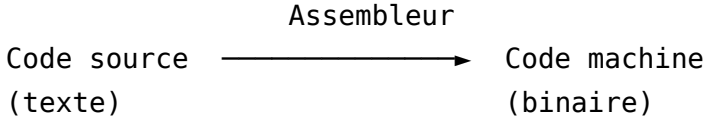
Figure 8.1: Position dans l'architecture

Nous sommes à la Couche 3 : Assembleur - Du texte au binaire

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



```
ADD R1, R2, #10     →     0xE2821010
B loop              →     0xEAFFFFFE
```

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 → 0xEAFFFFFE
```

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
`.text`	Le code (les instructions) — généralement en lecture seule
`.data`	Les variables globales initialisées
`.bss`	Les variables globales non initialisées (mises à zéro)

8.4.2 Les Directives

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

Directive	Signification
`.text`	Début de la section code
`.data`	Début de la section données
`.global _start`	Exporte le symbole `_start`
`.word 123`	Réserve 4 octets avec la valeur 123
`.asciz "Hello"`	Chaîne terminée par un zéro
`.align 2`	Aligne sur un multiple de 4 octets
`.ltorg`	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.

8.9 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

8.9.1 Questions de compréhension

Q1. Pourquoi l'assembleur fait-il deux passes sur le code source ?

Q2. Quelle est la différence entre `.text` et `.data` ?

Q3. Comment l'assembleur gère-t-il `LDR R0, =0xDEADBEEF` ?

Q4. Que se passe-t-il si un branchement est trop loin (offset > 24 bits) ?

Q5. Qu'est-ce qu'un fichier binaire A32B contient exactement ?

8.9.2 Mini-défi pratique

Encodez manuellement cette instruction en binaire 32 bits :

```
ADD R1, R2, R3
```

Indice : Format ALU registre = `[cond:4][000][opcode:4][S][Rn:4][Rd:4][00000000][Rm:4]`

*Les solutions se trouvent dans le document **Codex_Solutions**.*

8.9.3 Checklist de validation

Avant de passer au chapitre 7, assurez-vous de pouvoir :

- ☐ Expliquer le rôle des deux passes de l'assembleur
- ☐ Utiliser les directives `.text`, `.data`, `.word`, `.asciz`
- ☐ Comprendre le fonctionnement du literal pool
- ☐ Encoder une instruction simple à la main (format binaire)
- ☐ Lire et interpréter un hexdump de fichier binaire

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 ?

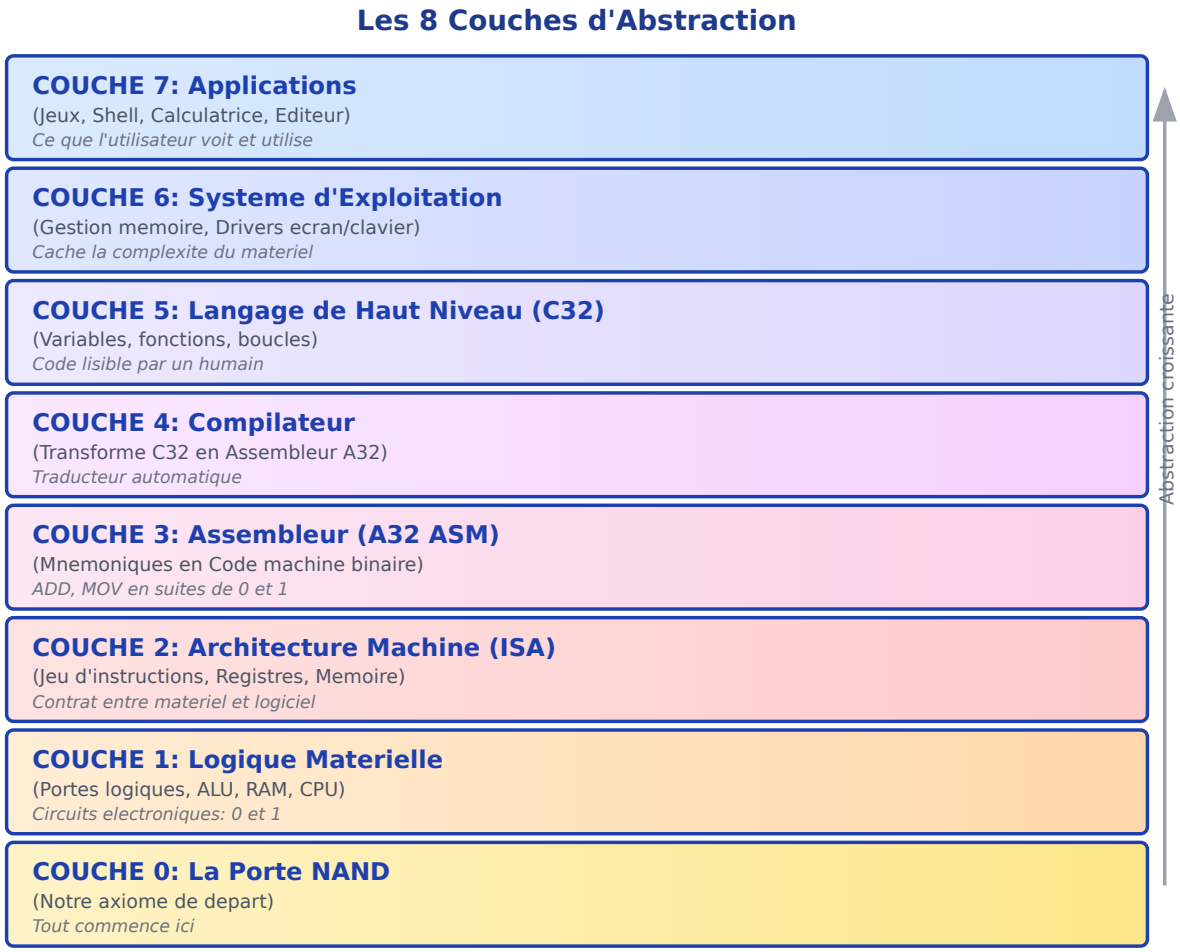


Figure 9.1: Position dans l’architecture

Nous sommes à la Couche 4 : Compilateur - Transforme C32 en Assembleur A32

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

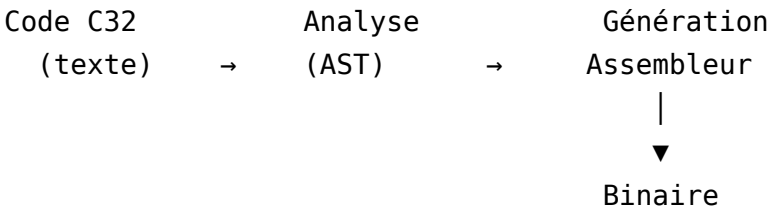
9.2 Le Rôle du Compilateur

9.2.1 Pourquoi un compilateur ?

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

```
          Declaration
             /    \
          Type    Assignment
           |        /    \
        "int"   "x"    Number
                         |
                         5
```

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

```c
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

```c
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 Du Code C32 à l'Exécution : Trace Complète

Suivons pas à pas le voyage d'un programme simple à travers toutes les couches.

9.7.1 Le Programme C32

```
// simple.c
int square(int x) {
    return x * x;
}

int main() {
    int n = 5;
    return square(n);
}
```

9.7.2 Étape 1 : Analyse Lexicale (Lexer)

Le lexer découpe le texte en tokens :

```
int      → [KEYWORD: int]
square   → [IDENTIFIER: "square"]
(        → [LPAREN]
int      → [KEYWORD: int]
x        → [IDENTIFIER: "x"]
)        → [RPAREN]
{        → [LBRACE]
return   → [KEYWORD: return]
x        → [IDENTIFIER: "x"]
*        → [STAR]
x        → [IDENTIFIER: "x"]
;        → [SEMICOLON]
}        → [RBRACE]
...
```

9.7.3 Étape 2 : Analyse Syntaxique (Parser)

Le parser construit un AST (Abstract Syntax Tree) :

```
Program
├── Function: square(x: int) -> int
│   └── Return
│       └── BinaryOp: *
│           ├── Var: x
│           └── Var: x
│
└── Function: main() -> int
    └── Block
        ├── VarDecl: n = 5
        └── Return
            └── Call: square
                └── Var: n
```

9.7.4 Étape 3 : Génération de Code (Assembleur A32)

Le générateur parcourt l'AST et produit :

```
; ========================================
; Fonction: square(x)
; Paramètre x dans R0
; ========================================
square:
    ; Prologue (sauvegarde du contexte)
    PUSH {LR}               ; Sauver adresse de retour

    ; Corps: return x * x
    ; x est déjà dans R0
    MUL R0, R0, R0          ; R0 = x * x

    ; Épilogue (restauration et retour)
    POP {LR}                ; Restaurer LR
    BX LR                   ; Retour (résultat dans R0)

; ========================================
; Fonction: main()
; ========================================
main:
    ; Prologue
    PUSH {R4, LR}           ; Sauver R4 et LR
    SUB SP, SP, #4          ; Réserver 4 octets pour n

    ; int n = 5
    MOV R4, #5              ; R4 = 5 (utilise R4 car callee-saved)
    STR R4, [SP, #0]        ; n = 5 (sur la pile)

    ; return square(n)
    MOV R0, R4              ; Argument: R0 = n
    BL square               ; Appel de square

    ; Épilogue
    ADD SP, SP, #4          ; Libérer la pile
    POP {R4, LR}            ; Restaurer R4 et LR
    BX LR                   ; Retour (R0 contient 25)
```

9.7.5 Étape 4 : Assemblage (Binaire)

L'assembleur traduit chaque instruction en code machine 32 bits :

```
Adresse   Binaire (hex)  Instruction           Encodage
--------  ------------  --------------------  --------------------------
0x0000    E92D4000      PUSH {LR}             Cond=E, opcode=92D, Rn=D(SP)
0x0004    E0000090      MUL R0, R0, R0        Cond=E, Rm=R0, Rs=R0, Rd=R0
0x0008    E8BD4000      POP {LR}              Cond=E, opcode=8BD, Rn=D
0x000C    E12FFF1E      BX LR                 Cond=E, opcode=12FFF1, Rm=E
```

```
0x0010    E92D4010       PUSH {R4, LR}         Reglist = R4 + LR
0x0014    E24DD004       SUB SP, SP, #4        Imm = 4
0x0018    E3A04005       MOV R4, #5            Imm = 5
0x001C    E58D4000       STR R4, [SP, #0]      Offset = 0
0x0020    E1A00004       MOV R0, R4            Rm = R4
0x0024    EBFFFFF5       BL square             Offset = -11 (vers 0x0000)
0x0028    E28DD004       ADD SP, SP, #4        Imm = 4
0x002C    E8BD4010       POP {R4, LR}          Reglist = R4 + LR
0x0030    E12FFF1E       BX LR                 Retour
```

9.7.6 Étape 5 : Exécution (Cycle par Cycle)

Trace d'exécution sur le CPU (simplifié) :

```
┌──────────────────────────────────────────────────────────────────┐
│                    EXÉCUTION PAS À PAS                           │
├─────────┬──────────────────────┬─────────────────────────────────┤
│ Cycle   │ PC / Instruction     │ État après exécution            │
├─────────┼──────────────────────┼─────────────────────────────────┤
│    1    │ 0x0010 PUSH {R4,LR}  │ SP=0xFFF8, Mem[SP]=LR, R4       │
│    2    │ 0x0014 SUB SP,SP,#4  │ SP=0xFFF4                       │
│    3    │ 0x0018 MOV R4,#5     │ R4=5                            │
│    4    │ 0x001C STR R4,[SP]   │ Mem[0xFFF4]=5 (n=5)             │
│    5    │ 0x0020 MOV R0,R4     │ R0=5 (argument pour square)     │
│    6    │ 0x0024 BL square     │ LR=0x0028, PC=0x0000            │
│         │                      │ ─── Entrée dans square ───      │
│    7    │ 0x0000 PUSH {LR}     │ SP=0xFFF0, Mem[SP]=0x0028       │
│    8    │ 0x0004 MUL R0,R0,R0  │ R0=5*5=25                       │
│    9    │ 0x0008 POP {LR}      │ LR=0x0028, SP=0xFFF4            │
│   10    │ 0x000C BX LR         │ PC=0x0028                       │
│         │                      │ ─── Retour dans main ───        │
│   11    │ 0x0028 ADD SP,SP,#4  │ SP=0xFFF8                       │
│   12    │ 0x002C POP {R4,LR}   │ R4 et LR restaurés, SP=0x10000  │
│   13    │ 0x0030 BX LR         │ Retour au système avec R0=25    │
└─────────┴──────────────────────┴─────────────────────────────────┘
```

```
Résultat final : R0 = 25 (square(5) = 5 * 5 = 25)
```

9.7.7 Visualisation de la Pile

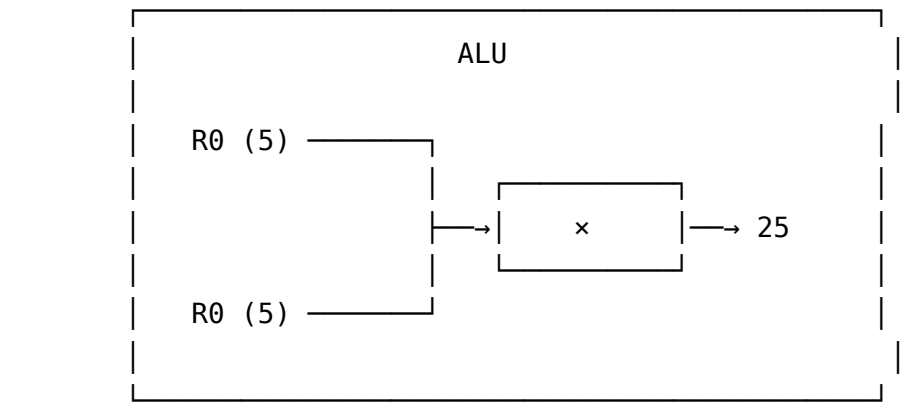
```
Adresse    Avant main    Après PUSH    Après SUB     Pendant square
```

```
           (SP=0x10000)  (SP=0xFFF8)   (SP=0xFFF4)   (SP=0xFFF0)
──────────  ────────────  ────────────  ────────────  ────────────
0x10000    [libre]       [libre]       [libre]       [libre]
0x0FFFC                  LR (retour)   LR (retour)   LR (retour)
0x0FFF8                  R4 (sauvé)    R4 (sauvé)    R4 (sauvé)
0x0FFF4                                n = 5         n = 5
0x0FFF0                                              LR = 0x0028
           ↑ SP          ↑ SP          ↑ SP          ↑ SP
```

9.7.8 Ce qui se passe dans le Hardware

À chaque cycle, le CPU effectue le cycle Fetch-Decode-Execute :

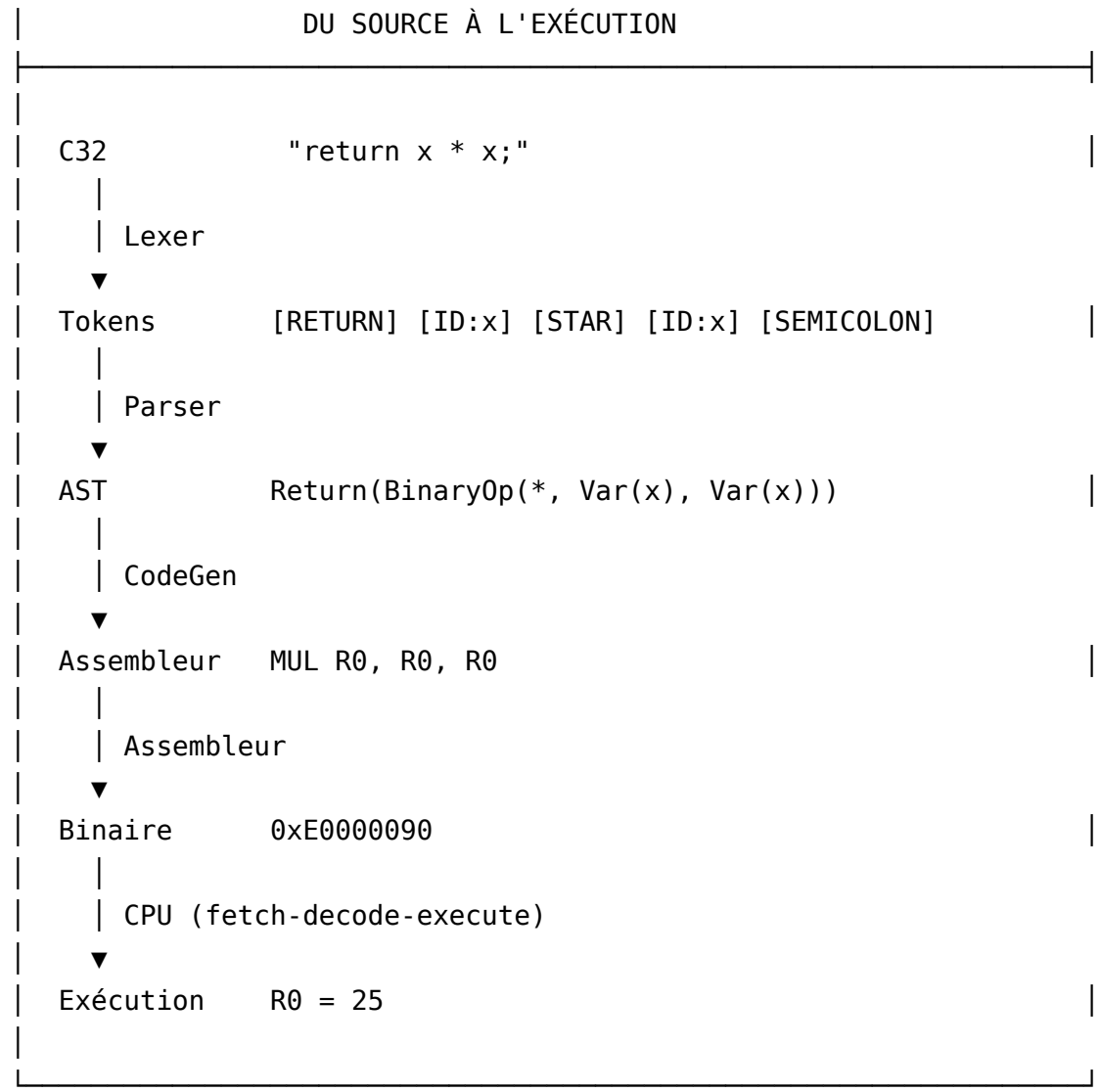
```
Cycle 8 : MUL R0, R0, R0
──────────────────────────────────────────────────────────────────

1. FETCH : PC=0x0004 → Instruction Register = 0xE0000090
           PC = PC + 4 (préparation instruction suivante)

2. DECODE : Opcode = MUL
            Rd = R0 (destination)
            Rm = R0 (premier opérande)
            Rs = R0 (second opérande)
            → Activer le multiplicateur de l'ALU

3. EXECUTE :
```




```
4. WRITE BACK : R0 = 25
```

9.7.9 Résumé du Voyage

```
┌──────────────────────────────────────────────────────────────────────────┐
```



Cette trace montre comment chaque couche d'abstraction transforme le code : - Le **lexer** voit des caractères → produit des tokens - Le **parser** voit des tokens → produit une structure (AST) - Le **codegen** voit un AST → produit de l'assembleur - L'**assembleur** voit des mnémoniques → produit du binaire - Le **CPU** voit du binaire → exécute des opérations

---

9.8 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.

9.8.1 Phase 1 : Lexer (Analyse Lexicale)

Le lexer transforme le texte en tokens.

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

9.8.2 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 (* > +)
2.2 Évaluer une Opération	Évaluer `3 + 4` ou `6 * 7`
2.3 Parser avec Précédence	Descente récursive pour `2 + 3 * 4`
2.4 Parenthèses	Supporter `(2 + 3) * 4`

9.8.3 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

9.8.4 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 précédence

9.8.5 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

9.8.6 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

9.8.7 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.

9.8.8 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

9.9 Exercices Pratiques

9.9.1 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

9.9.2 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 !

9.10 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 !

9.11 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

9.11.1 Questions de compréhension

Q1. Quelles sont les trois phases principales d'un compilateur ?

Q2. Comment le compilateur gère-t-il les variables locales ?

Q3. Comment un `if-else` est-il traduit en assembleur ?

Q4. Quelle est la convention d'appel pour les arguments de fonction ?

Q5. Pourquoi le compilateur génère-t-il un prologue et un épilogue pour chaque fonction ?

9.11.2 Mini-défi pratique

Prédisez l'assembleur généré pour ce code C32 :

```
int double_it(int x) {
    return x + x;
}
```

*Les solutions se trouvent dans le document **Codex_Solutions**.*

9.11.3 Checklist de validation

Avant de passer au chapitre 8, assurez-vous de pouvoir :

- ☐ Décrire le rôle du lexer, parser, et codegen
- ☐ Expliquer comment les variables locales sont stockées
- ☐ Traduire un `if-else` simple en assembleur
- ☐ Connaître la convention d'appel (R0-R3, pile, LR)
- ☐ Comprendre le prologue/épilogue d'une fonction

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 ?

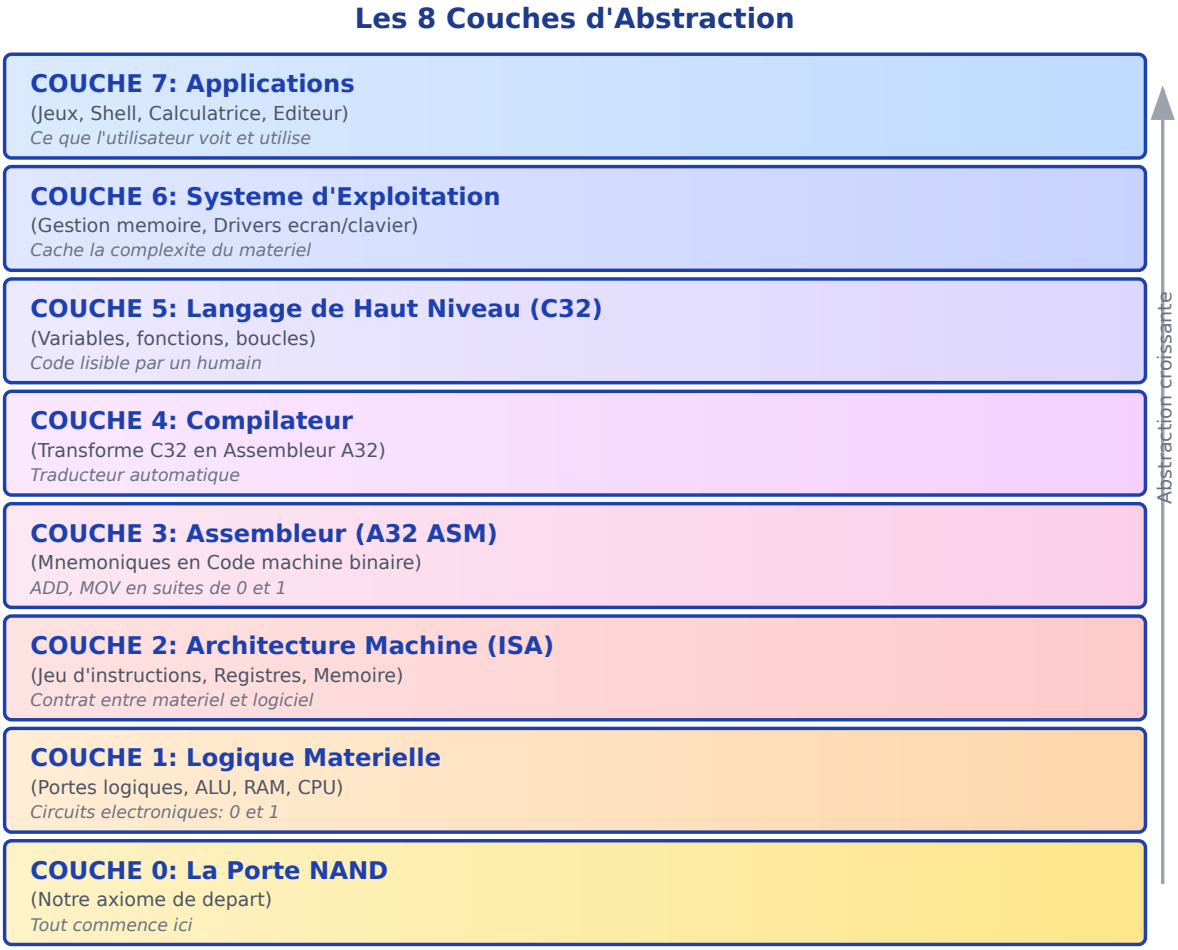


Figure 10.1: Position dans l'architecture

Nous sommes à la Couche 5 : Langage de Haut Niveau (C32) - Variables, fonctions, boucles

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                    C32
──────────                    ──
```

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

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

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 !
```

10.7.4 Arithmétique des Pointeurs

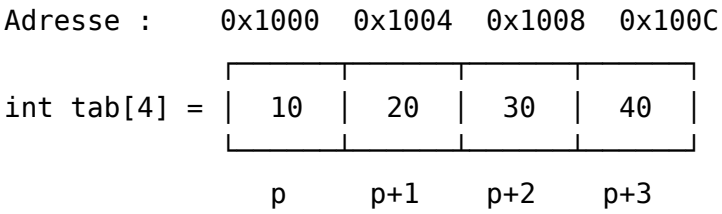
C'est un concept **crucial** mais souvent mal compris. Quand vous ajoutez 1 à un pointeur, il n'avance pas d'un octet, mais d'un **élément** !

10.7.4.1 Le Principe : Scaling par sizeof

```
int* p;      // Pointeur vers int (4 octets)
char* c;     // Pointeur vers char (1 octet)

p = p + 1;   // Avance de 4 octets (sizeof(int))
c = c + 1;   // Avance de 1 octet (sizeof(char))
```

Pourquoi ? Parce que `p + 1` signifie "l'élément suivant", pas "l'octet suivant".

10.7.4.2 Exemple en Mémoire



Si `p = &tab[0]` (adresse 0x1000) : - `p + 1` = 0x1004 (pas 0x1001 !) - `p + 2` = 0x1008 - `*p` = 10 - `*(p + 2)` = 30 = `p[2]`

10.7.4.3 Équivalence Pointeur/Tableau

Ces expressions sont **identiques** :

Forme tableau	Forme pointeur	Signification
`tab[i]`	`*(tab + i)`	Valeur à l'indice i
`&tab[i]`	`tab + i`	Adresse de l'indice i

10.7.4.4 Différence `*p + 1` vs `*(p + 1)`

Attention aux parenthèses !

```
int tab[3] = {10, 20, 30};
int* p = tab;

*p + 1     // = 10 + 1 = 11  (valeur + 1)
*(p + 1)   // = 20           (élément suivant)
```

10.7.4.5 Soustraction de Pointeurs

La différence entre deux pointeurs donne le **nombre d'éléments** :

```
int tab[10];
int* debut = &tab[0];
int* fin = &tab[7];

int distance = fin - debut;  // = 7 (éléments), pas 28 (octets)
```

10.7.4.6 Parcours de Tableau avec Pointeurs

Deux façons équivalentes de parcourir :

```
// Style indexation
for (int i = 0; i < 10; i = i + 1) {
    tab[i] = tab[i] * 2;
}

// Style pointeur
int* p = tab;
int* end = tab + 10;
while (p < end) {
    *p = *p * 2;
    p = p + 1;
}
```

10.7.4.7 Pointeurs vers Différents Types

```
// Même adresse, interprétation différente
int val = 0x41424344;    // 'ABCD' en ASCII
int* pi = &val;
char* pc = (char*)&val;  // Cast nécessaire

*pi       // = 0x41424344 (un int de 4 octets)
pc[0]     // = 0x44 = 'D' (premier octet, little-endian)
pc[1]     // = 0x43 = 'C'
pc[2]     // = 0x42 = 'B'
pc[3]     // = 0x41 = 'A'
```

10.8 Accès au Matériel (MMIO)

10.8.1 L'écran

```
// L'écran commence à 0x00400000
// 320×240 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

```c
// Programme qui dessine un rectangle et attend une touche

extern void putchar(char c);

int main() {
    // Dessiner un rectangle 10×5 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
Bases	Variables, Expressions, Modulo
Contrôle	Conditions, Else-If, Opérateurs Logiques
Boucles	For, While, Imbriquées

Catégorie	Exercices clés
Fonctions	Paramètres, Valeur Absolue, Min/Max
Tableaux	Accès, Maximum, Comptage
Pointeurs	Adresses, Swap
Récursion	Factorielle, Fibonacci, PGCD
Algorithmes	Tri à Bulles, Recherche Binaire
Graphique	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

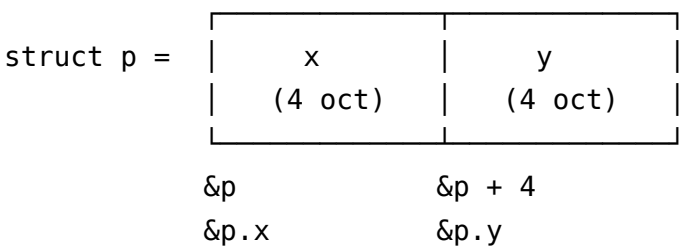
```
struct Point p;     // Déclarer une variable de type struct
p.x = 10;           // Accéder aux champs avec .
p.y = 20;
```

10.11.3 Organisation en Mémoire

Concept clé : Les champs d'une structure sont placés **consécutivement** en mémoire.

```
struct Point {
    int x;    // Offset 0, taille 4
    int y;    // Offset 4, taille 4
};            // Taille totale : 8 octets
```

En mémoire (si p est à l'adresse 0x1000) :



10.11.4 Calcul des Offsets

Le compilateur calcule l'**offset** de chaque champ :

```
struct Player {
    int health;     // Offset 0
    int x;          // Offset 4
    int y;          // Offset 8
    char name[16];  // Offset 12
};                  // Taille totale : 28 octets
```

Quand vous écrivez `player.y`, le compilateur génère :

```
; player.y = 100
; Si R0 contient l'adresse de player :
MOV R1, #100
STR R1, [R0, #8]   ; Offset de y = 8
```

10.11.5 Alignement (Padding)

En C standard, les structures peuvent avoir du "padding" pour l'alignement. En C32, on suppose un alignement simple sur 4 octets pour les int.

```
struct Mixed {
    char a;    // Offset 0, taille 1
    // (3 octets de padding)
    int b;     // Offset 4, taille 4
    char c;    // Offset 8, taille 1
    // (3 octets de padding)
};             // Taille : 12 octets (pas 6 !)
```

10.11.6 Tableaux de Structures

```
struct Point points[3];

// En mémoire (chaque Point = 8 octets) :
// points[0] à offset 0
// points[1] à offset 8
// points[2] à offset 16
```

Pour accéder à `points[i].y` :

```
adresse = base + i * sizeof(struct Point) + offset_y
        = base + i * 8 + 4
```

10.11.7 Pointeur vers structure

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

10.11.8 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
}
```

10.11.9 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
`struct`	Supporté
`float`, `double`	Non supporté
`malloc`/`free`	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 !

10.14 Auto-évaluation

Testez votre compréhension avant de passer au chapitre suivant.

10.14.1 Questions de compréhension

Q1. Quelle est la différence entre `int` et `uint` en C32 ?

Q2. Qu'est-ce qu'un pointeur et comment l'utilise-t-on ?

Q3. Comment accède-t-on à l'écran en C32 ?

Q4. Quelle est la syntaxe d'une boucle `for` en C32 ?

Q5. Comment définit-on et utilise-t-on un tableau en C32 ?

10.14.2 Mini-défi pratique

Écrivez une fonction C32 qui calcule la somme des éléments d'un tableau :

```
int sum(int *arr, int len) {
    // À compléter
}
```

*Les solutions se trouvent dans le document **Codex_Solutions**.*

10.14.3 Checklist de validation

Avant de passer au chapitre 9, assurez-vous de pouvoir :

- ☐ Utiliser les types de base : `int`, `uint`, `char`, `bool`
- ☐ Déclarer et utiliser des pointeurs
- ☐ Accéder à la mémoire mappée (écran, clavier)
- ☐ Écrire des boucles `for` et `while`
- ☐ Manipuler des tableaux et des chaînes de caractères

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 ?

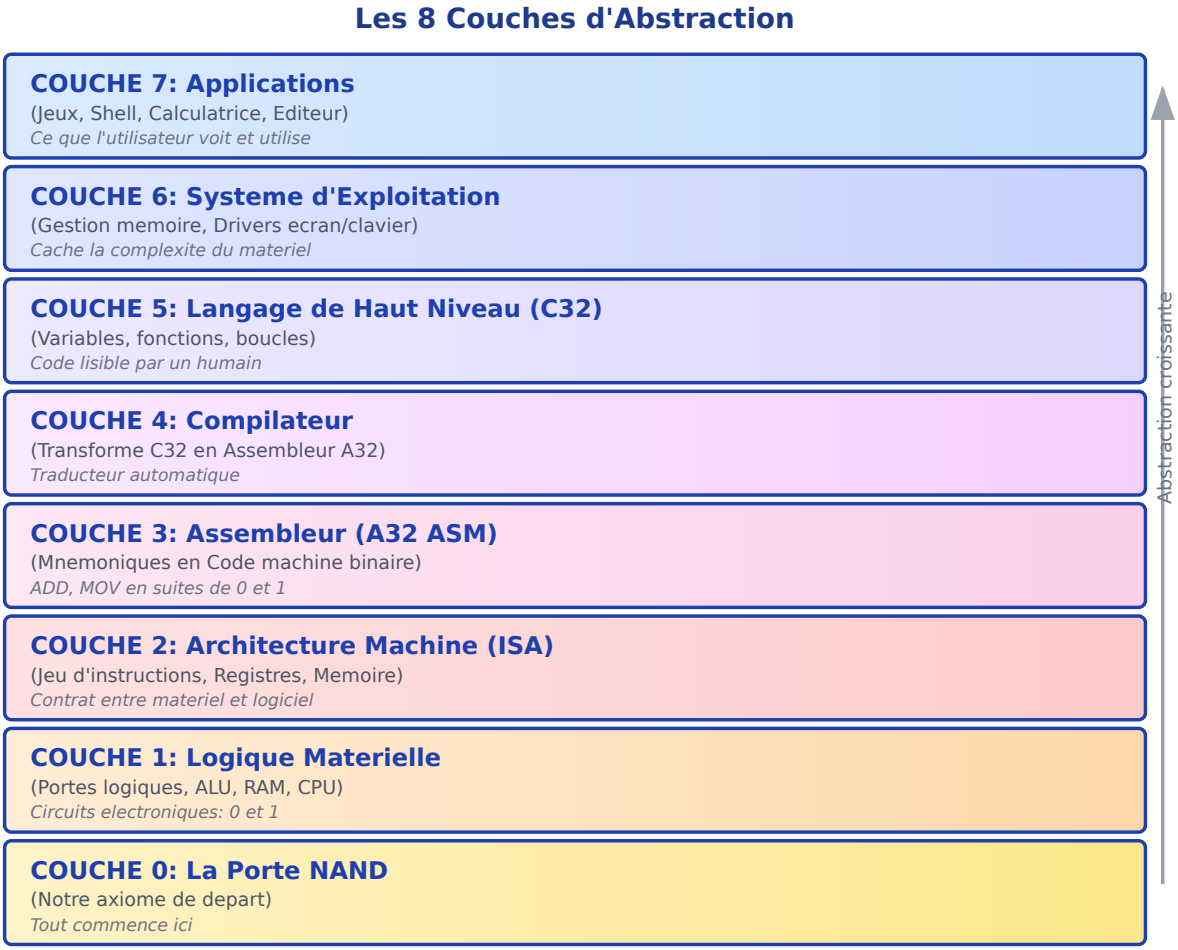


Figure 11.1: Position dans l'architecture

Nous sommes à la Couche 6 : Système d'Exploitation - Le sommet de la pyramide logicielle !

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

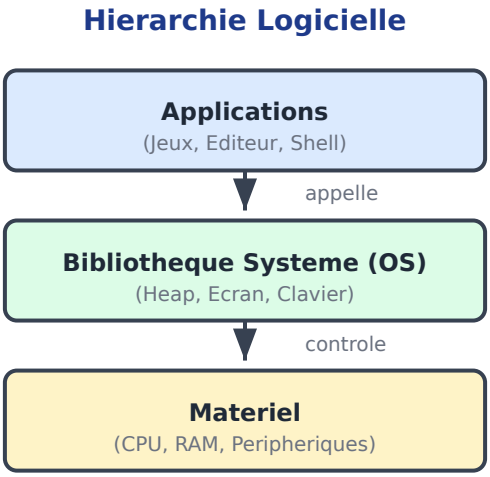


Figure 11.2: Hiérarchie logicielle

11.2.2 Ce que fait un OS

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

- **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

```
┌──────────────────────────────────────────────────────────┐
│ [size|next] [données...]  [size|next] [données...] │
└──────────────────────────────────────────────────────────┘
```

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 !

11.12 Auto-évaluation

Testez votre compréhension de l'ensemble du parcours.

11.12.1 Questions de compréhension

Q1. Qu'est-ce qu'un allocateur mémoire et pourquoi en a-t-on besoin ?

Q2. Comment fonctionne l'affichage graphique via MMIO ?

Q3. Quelle est la différence entre polling et interruptions ?

Q4. Pourquoi sépare-t-on le code en bibliothèques (Math, String, Screen, etc.) ?

Q5. Résumez le chemin complet du code source à l'exécution.

*Les solutions se trouvent dans le document **Codex_Solutions**.*

11.12.2 Réflexion finale

Vous avez maintenant une vision complète de comment fonctionne un ordinateur :

Couche	Ce que vous avez appris
Portes logiques	NAND → NOT, AND, OR, XOR, MUX
Arithmétique	Half Adder → Full Adder → ALU
Mémoire	DFF → Registre → RAM → PC
CPU	Fetch-Decode-Execute, pipeline

Couche	Ce que vous avez appris
ISA	Instructions A32, encodage binaire
Assembleur	Texte → Binaire (2 passes)
Compilateur	C32 → Assembleur (lexer, parser, codegen)
OS	Allocation mémoire, graphiques, I/O

Vous n’êtes plus un utilisateur passif de la technologie — vous comprenez comment elle fonctionne !

11.12.3 Checklist de validation finale

☐ Je peux expliquer comment NAND permet de construire toutes les portes
☐ Je comprends le complément à 2 et les drapeaux NZCV
☐ Je sais tracer l’exécution d’une instruction dans le CPU
☐ Je peux écrire et déboguer un programme en assembleur A32
☐ Je comprends comment le compilateur traduit C32 en assembleur
☐ Je sais utiliser le MMIO pour l’écran et le clavier
☐ Je peux expliquer la hiérarchie mémoire (registres → cache → RAM)

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) : `Inv(a) = Nand(a, a)`
And2	Porte AND : `And2(a,b) = Inv(Nand(a,b))`
Or2	Porte OR : `Or2(a,b) = Nand(Inv(a), Inv(b))`
Xor2	Porte XOR : `Xor2(a,b) = Or2(And2(a, Inv(b)), And2(Inv(a), b))`
Mux	Multiplexeur : si sel=0 alors y=a sinon y=b
DMux	Demultiplexeur : distribue x vers a ou b selon sel

12.1.2 Projet 2 : Arithmetique

Exercice	Description
HalfAdder	Demi-additionneur : sum = a XOR b, carry = a AND b
FullAdder	Additionneur complet avec retenue entrante

Exercice	Description
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'acces memoire sur les performances cache.

Exercice	Objectif	Resultat
Acces Sequentiel	Parcours cache-friendly (adresses consecutives)	R0 = 100

Exercice	Objectif	Resultat
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	7*8 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
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

Exercice	Objectif	Resultat
Structs Imbriquees	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 Reconnaitre Chiffre	`is_digit(c)` : retourne 1 si '0'-'9'
1.2 Lire Nombre	`parse_number(s, pos)` : extrait un entier
1.3 Identifier Tokens	`next_token()` : 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 : * avant +
2.4 Parentheses	Supporter `(2 + 3) * 4`

12.4.3 Phase 3 : Emission ASM

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

12.4.4 Phase 4 : CodeGen Expressions

Exercice	Description
4.1 Constante -> A32	Code pour charger une constante
4.2 Addition -> A32	`a + b` -> `MOV/MOV/ADD`
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. **Progressez** : Chaque exercice prepare le suivant

Bon courage !

13 L’Art du Débogage

> “Le débogage, c’est comme être détective dans un film policier où vous êtes aussi le meurtrier.” — Filipe Fortes

Le débogage est une compétence fondamentale en programmation. Ce chapitre vous apprend à trouver et corriger les erreurs de manière méthodique.

13.1 Pourquoi ce Chapitre ?

Vous avez appris à construire des circuits, écrire de l’assembleur, et utiliser un compilateur. Mais que faire quand **ça ne marche pas** ?

Ce chapitre couvre : 1. Comment lire les messages d’erreur 2. Comment utiliser le simulateur pas-à-pas 3. Les erreurs les plus courantes et leurs solutions 4. Une méthodologie systématique de débogage

---

13.2 1. Comprendre les Messages d’Erreur

13.2.1 Codes d’Erreur du Projet Codex

Les outils du projet utilisent des codes d’erreur standardisés :

Préfixe	Source	Description
E1xxx	Assembleur	Erreurs d’assemblage
E2xxx	Compilateur C32	Erreurs de compilation
E3xxx	Linker	Erreurs de liaison

13.2.2 Erreurs d'Assemblage (E1xxx)

Code	Signification	Solution
E1001	Mnémonique inconnu	Vérifiez l'orthographe de l'instruction
E1002	Opérande invalide	Vérifiez le format des registres/immédiats
E1003	Label non défini	Ajoutez le label ou corrigez son nom
E1004	Immédiat hors plage	Utilisez `LDR R0, =value` pour les grandes valeurs
E1005	Registre invalide	Utilisez R0-R15 uniquement
E1008	Literal pool overflow	Placez `.ltorg` plus tôt dans le code

Exemple : Erreur E1004

```
; ERREUR : L'immédiat 0x12345678 ne tient pas sur 12 bits
MOV R0, #0x12345678

; SOLUTION : Utiliser le literal pool
LDR R0, =0x12345678
```

13.2.3 Erreurs de Compilation (E2xxx)

Code	Signification	Solution
E2001	Erreur de syntaxe	Vérifiez les parenthèses, points-virgules
E2002	Type incompatible	Vérifiez les types des opérandes
E2003	Variable non déclarée	Déclarez la variable avant utilisation
E2004	Fonction non définie	Définissez la fonction ou incluez le header

Code	Signification	Solution
E2008	Return manquant	Ajoutez `return` à la fin de la fonction

Exemple : Erreur E2003

```
// ERREUR : 'x' non déclaré
int main() {
    x = 5;  // E2003: Variable 'x' non déclarée
    return 0;
}

// SOLUTION : Déclarer la variable
int main() {
    int x;
    x = 5;
    return 0;
}
```

13.2.4 Erreurs de Liaison (E3xxx)

Code	Signification	Solution
E3001	Symbole non résolu	Définissez le symbole ou liez la bibliothèque
E3002	Point d'entrée manquant	Ajoutez `_start:` ou `main()`
E3004	Débordement mémoire	Réduisez la taille du programme

13.3 2. Utiliser le Simulateur Pas-à-Pas

13.3.1 Le Visualiseur CPU

Le visualiseur web (`web/visualizer.html`) est votre meilleur ami pour déboguer.

Fonctionnalités clés : - **Step (N ou F10)** : Exécute une instruction - **Registres** : Voir R0-R15, SP, LR, PC et les flags - **Mémoire** : Inspecter la RAM - **Code** : Ligne courante surlignée

13.3.2 Méthodologie de Débogage Pas-à-Pas

1. **Chargez votre programme** dans le visualiseur
2. **Identifiez le symptôme** : Qu'est-ce qui ne va pas ?
3. **Formulez une hypothèse** : Où le bug pourrait-il être ?
4. **Placez-vous avant** la zone suspecte
5. **Exécutez pas-à-pas** en vérifiant les registres
6. **Trouvez la divergence** : Où le comportement diffère-t-il de l'attendu ?

13.3.3 Exemple Pratique

Programme bugué :

```
; Censé calculer 5 + 3 = 8
    MOV R0, #5
    MOV R1, #3
    SUB R2, R0, R1   ; BUG : devrait être ADD !
    SVC #0
```

Débogage : 1. Après `MOV R0, #5` : R0 = 5 □ 2. Après `MOV R1, #3` : R1 = 3 □ 3. Après `SUB R2, R0, R1` : R2 = 2 □ (attendu : 8)

→ **Bug trouvé** : `SUB` au lieu de `ADD`

13.4 3. Erreurs Courantes et Solutions

13.4.1 3.1 Assembleur

13.4.1.1 Oubli de sauvegarder LR

Symptôme : Le programme ne retourne pas correctement d'une fonction.

```
; BUGUÉ
my_func:
    BL other_func    ; LR est écrasé !
    BX LR            ; Retourne... où ?

; CORRECT
my_func:
    PUSH {LR}        ; Sauvegarder LR
    BL other_func
    POP {LR}         ; Restaurer LR
    BX LR
```

13.4.1.2 Off-by-one dans les boucles

Symptôme : La boucle s'exécute une fois de trop ou de moins.

```
; BUGUÉ : S'arrête à 9 au lieu de 10
    MOV R0, #0
loop:
    CMP R0, #10
    BEQ done          ; BEQ : si R0 == 10, sauter
    ; ... corps de la boucle ...
    ADD R0, R0, #1
    B loop
done:

; Vérifiez toujours : combien d'itérations attendues ?
; R0 va de 0 à 9 = 10 itérations ✓
```

13.4.1.3 Mauvais alignement mémoire

Symptôme : Erreur MISALIGNED ou données corrompues.

```
; BUGUÉ : Adresse non alignée sur 4 octets
    LDR R0, [R1, #3]   ; 3 n'est pas multiple de 4 !

; CORRECT
    LDR R0, [R1, #4]   ; OK : 4 est multiple de 4
```

13.4.2 3.2 Compilateur C32

13.4.2.1 Oubli du return

Symptôme : Valeur de retour incorrecte ou aléatoire.

```
// BUGUÉ
int add(int a, int b) {
    int c = a + b;
    // Oops, pas de return !
}

// CORRECT
int add(int a, int b) {
    int c = a + b;
    return c;
}
```

13.4.2.2 Confusion pointeur/valeur

Symptôme : Crash ou données incorrectes.

```
// BUGUÉ
void increment(int x) {
    x = x + 1;  // Modifie la COPIE, pas l'original !
}

// CORRECT
void increment(int *x) {
    *x = *x + 1;  // Modifie la valeur POINTÉE
}

// Appel
int n = 5;
increment(&n);  // Passer l'ADRESSE
```

13.4.2.3 Division par zéro

Symptôme : Trap DIV_ZERO ou résultat infini.

```
// BUGUÉ
int divide(int a, int b) {
    return a / b;  // Et si b == 0 ?
}

// CORRECT
int divide(int a, int b) {
    if (b == 0) {
        return 0;  // Ou gérer l'erreur autrement
    }
    return a / b;
}
```

13.4.3 3.3 HDL

13.4.3.1 Signal non connecté

Symptôme : Sortie toujours à 0 ou X (indéfini).

```
-- BUGUÉ : 'result' n'est jamais assigné
entity Adder is
  port(a, b : in bit; result : out bit);
end entity;
```

```
architecture rtl of Adder is
begin
  -- Oops, rien ici !
end architecture;

-- CORRECT
architecture rtl of Adder is
begin
  result <= a xor b;  -- Connexion explicite
end architecture;
```

13.4.3.2 Boucle combinatoire

Symptôme : Le simulateur ne converge pas ou donne des résultats instables.

```
-- BUGUÉ : 'x' dépend de lui-même instantanément
signal x : bit;
x <= not x;  -- Boucle infinie !

-- CORRECT : Utiliser une DFF pour la rétroaction
-- x(t+1) <= not x(t)  -- Via une DFF
```

13.5 4. Méthodologie Systématique

13.5.1 La Méthode Scientifique du Débogage

1. **Observer** : Quel est le symptôme exact ?
    - Message d'erreur ?
    - Mauvais résultat ?
    - Crash ?
2. **Hypothétiser** : Quelle pourrait être la cause ?
    - Erreur de logique ?
    - Mauvaise valeur ?
    - Condition incorrecte ?
3. **Tester** : Vérifier l'hypothèse
    - Ajouter des prints/traces
    - Exécuter pas-à-pas

- Simplifier le code

4. **Conclure** : L'hypothèse était-elle correcte ?
   - Si oui → corriger
   - Si non → nouvelle hypothèse

13.5.2 La Technique de la Bissection

Quand le bug est difficile à trouver :

1. **Divisez** le code en deux moitiés
2. **Testez** chaque moitié indépendamment
3. **Identifiez** quelle moitié contient le bug
4. **Répétez** jusqu'à isoler le problème

```
Code complet (bug quelque part)
    ├── Première moitié
    │   └── ✓ OK
    └── Deuxième moitié
        └── ✗ Bug ici !
            ├── Premier quart → ✓ OK
            └── Deuxième quart → ✗ Bug trouvé !
```

13.5.3 Ajouter des Traces

En C32 :

```c
void debug_print(int value) {
    // Afficher la valeur pour debug
    putc('[');
    // ... afficher value ...
    putc(']');
    putc('\n');
}

int main() {
    int x = compute_something();
    debug_print(x);  // Voir la valeur
    // ...
}
```

En assembleur :

```
; Afficher R0 pour debug (caractère ASCII)
    ADD R0, R0, #'0'    ; Convertir en ASCII
    LDR R1, =0xFFFF0000
    STR R0, [R1]        ; Afficher
```

13.6 5. Erreurs de Traps et leur Diagnostic

13.6.1 Types de Traps

Trap	Cause	Solution
DIV_ZERO	Division par 0	Vérifier le diviseur avant
MEM_FAULT	Accès mémoire invalide	Vérifier les pointeurs
MISALIGNED	Adresse non alignée	Aligner sur 4 octets
ILLEGAL	Instruction invalide	Vérifier l'encodage

13.6.2 Déboguer un MEM_FAULT

1. **Notez l'adresse** du PC quand le trap se produit
2. **Trouvez l'instruction** à cette adresse
3. **Examinez les registres** utilisés pour l'adressage
4. **Questions à se poser** :
    - Le pointeur est-il initialisé ?
    - Est-il dans la plage valide (0 - RAM_SIZE) ?
    - A-t-il été corrompu par une autre partie du code ?

13.7 6. Checklist de Débogage

Avant de chercher un bug complexe, vérifiez ces points simples :

13.7.1 Assembleur

☐ Toutes les instructions sont correctement orthographiées ?

- ☐ Les registres sont dans la plage R0-R15 ?
- ☐ Les immédiats sont dans la plage autorisée ?
- ☐ LR est sauvegardé avant les appels de fonction ?
- ☐ La pile est équilibrée (PUSH = POP) ?
- ☐ Les accès mémoire sont alignés sur 4 octets ?

13.7.2 C32

- ☐ Toutes les variables sont déclarées ?
- ☐ Toutes les fonctions ont un return ?
- ☐ Les pointeurs sont initialisés avant déréférencement ?
- ☐ Pas de division par zéro possible ?
- ☐ Les indices de tableau sont dans les bornes ?

13.7.3 HDL

- ☐ Toutes les sorties sont assignées ?
- ☐ Pas de boucles combinatoires ?
- ☐ Les signaux sont de la bonne largeur (bit vs bits) ?
- ☐ Les port maps connectent les bons signaux ?

13.8 7. Exercices de Débogage

13.8.1 Exercice 1 : Trouvez le bug

```
; Programme censé afficher "Hello"
    LDR R0, =message
    LDR R1, =0xFFFF0000
loop:
    LDRB R2, [R0]
    CMP R2, #0
    BEQ done
    STR R2, [R1]
    ADD R0, R0, #1
    B done              ; <-- Bug ici !
done:
    SVC #0

message: .asciz "Hello"
```

Voir la solution

Le bug est `B done` qui devrait être `B loop`. Le programme n'affiche que le premier caractère car il saute directement à `done` au lieu de boucler.

```
    B loop    ; CORRECT : retourner au début de la boucle
```

13.8.2 Exercice 2 : Trouvez le bug

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return n * factorial(n);  // <-- Bug ici !
}
```

Voir la solution

Le bug est `factorial(n)` qui devrait être `factorial(n - 1)`. Sans décrémenter `n`, la récursion est infinie → stack overflow.

```
return n * factorial(n - 1);  // CORRECT
```

13.8.3 Exercice 3 : Trouvez le bug

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 5;
    int y = 10;
    swap(x, y);
    // x est toujours 5, y toujours 10 !
    return 0;
}
```

Voir la solution

Le bug est le passage par valeur au lieu de passage par pointeur. Les modifications de `a` et `b` n'affectent pas `x` et `y`.

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5;
    int y = 10;
    swap(&x, &y);  // Passer les adresses
    return 0;
}
```

13.9 Ce qu'il faut retenir

1. **Lisez les messages d'erreur** : Ils contiennent souvent la solution
2. **Utilisez le pas-à-pas** : Le visualiseur est votre meilleur outil
3. **Formulez des hypothèses** : Approche scientifique
4. **Connaissez les erreurs courantes** : Elles représentent 90% des bugs
5. **Simplifiez** : Réduisez le problème au minimum reproductible

Le débogage est une compétence qui s'améliore avec la pratique. Chaque bug que vous trouvez vous rend meilleur !

14 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).

14.1 Le Problème : La RAM est Lente !

14.1.1 La Hiérarchie Mémoire

Voici comment est organisée la mémoire dans un ordinateur, du plus rapide au plus lent :

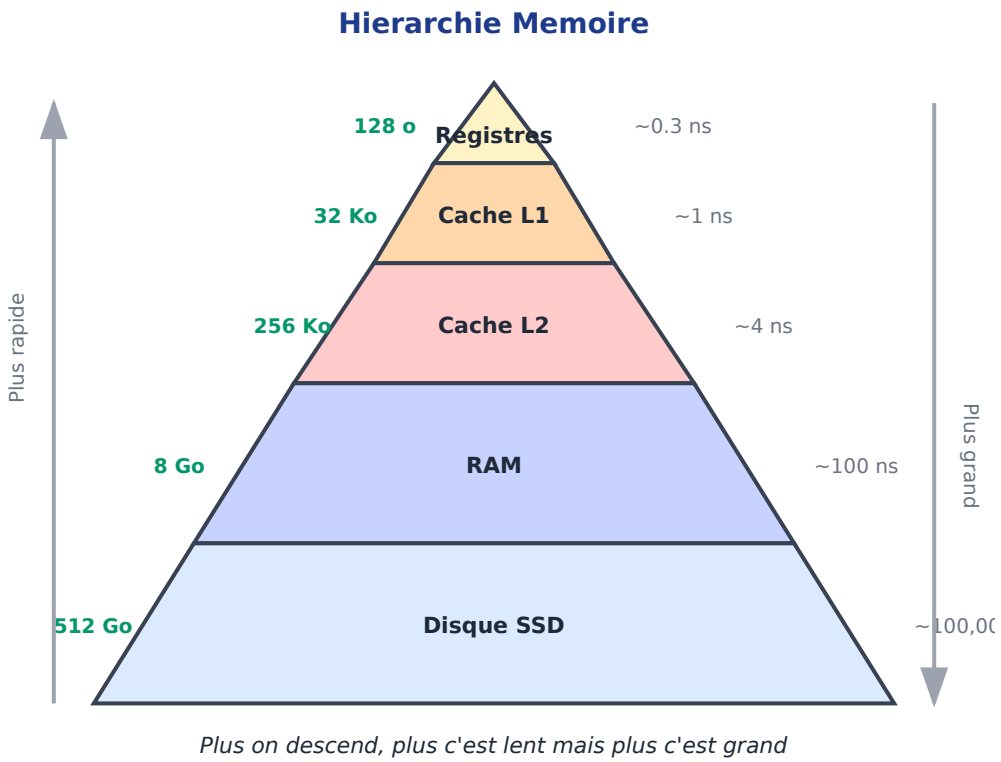


Figure 14.1: Hiérarchie mémoire

14.1.2 L'écart de vitesse visualisé

Si on convertissait ces temps en échelle humaine, les registres seraient comme cligner des yeux (0.3 seconde), le cache L1 comme une seconde, le cache L2 comme 4 secondes, la RAM comme 1 minute 40 secondes, et accéder au SSD prendrait… **27 heures** !

Sans cache, le processeur passerait **99% de son temps à attendre** !

14.2 La Solution : Le Principe de Localité

Les programmes ne lisent pas la mémoire au hasard. Ils suivent des **patterns** prévisibles.

14.2.1 Localité Temporelle

"Si j'utilise une donnée maintenant, je vais probablement la réutiliser bientôt"

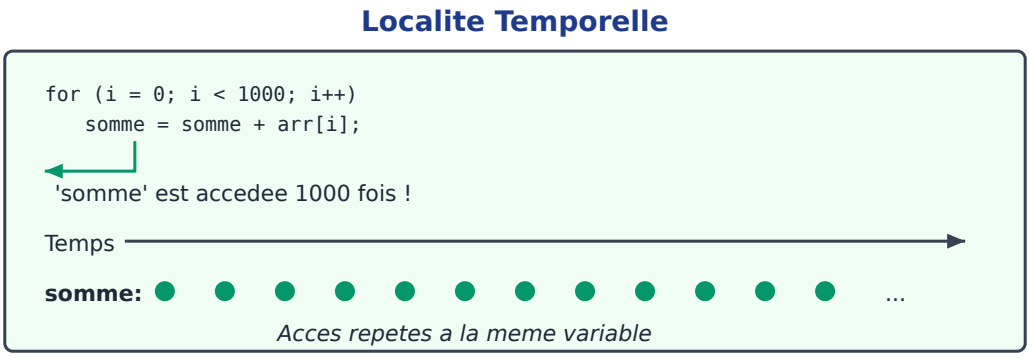


Figure 14.2: Localité temporelle

14.2.2 Localité Spatiale

"Si j'utilise une donnée, je vais probablement utiliser ses voisines"

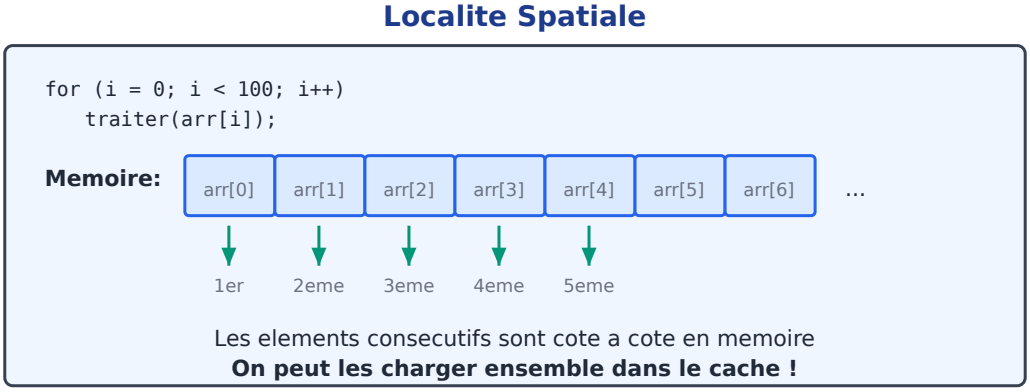


Figure 14.3: Localité spatiale

14.3 Comment Fonctionne le Cache ?

14.3.1 Les Lignes de Cache

Le cache stocke des **blocs** de données appelés **lignes de cache**, pas des octets individuels.

Quand vous demandez l'adresse 100, le cache charge **16 octets ensemble** (adresses 100 à 115). Ensuite, si vous demandez 101, 102, 103... c'est gratuit car déjà en cache !

14.3.2 Structure d'une Ligne de Cache

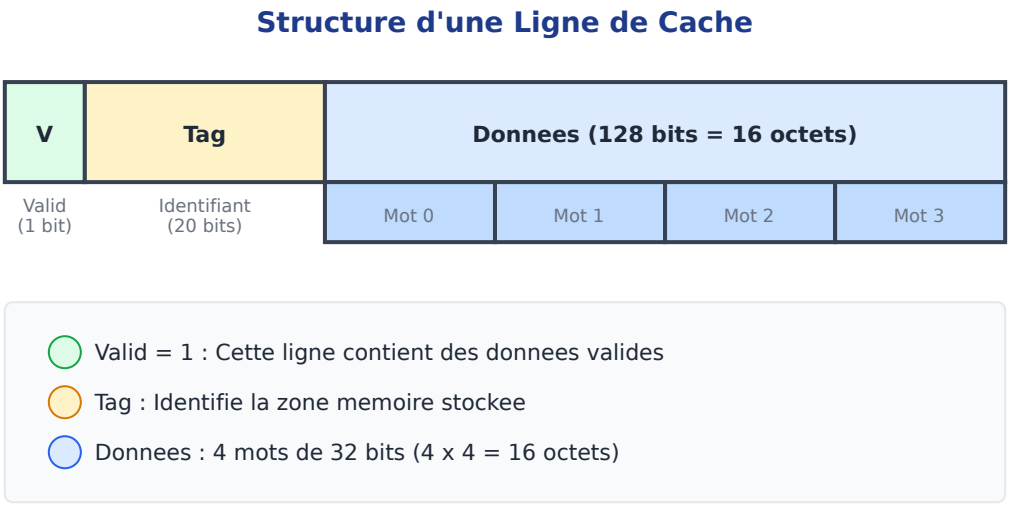


Figure 14.4: Structure d'une ligne de cache

14.3.3 Découpage d'une Adresse

Quand le CPU demande l'adresse `0x00001234`, comment le cache la trouve-t-il ?

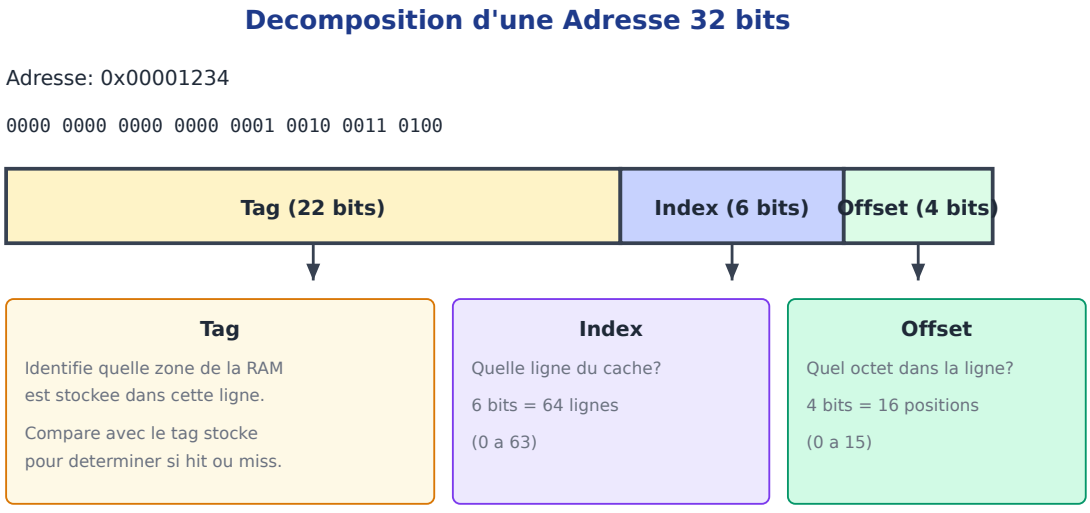


Figure 14.5: Décomposition d'une adresse

14.3.4 Structure Complète du Cache

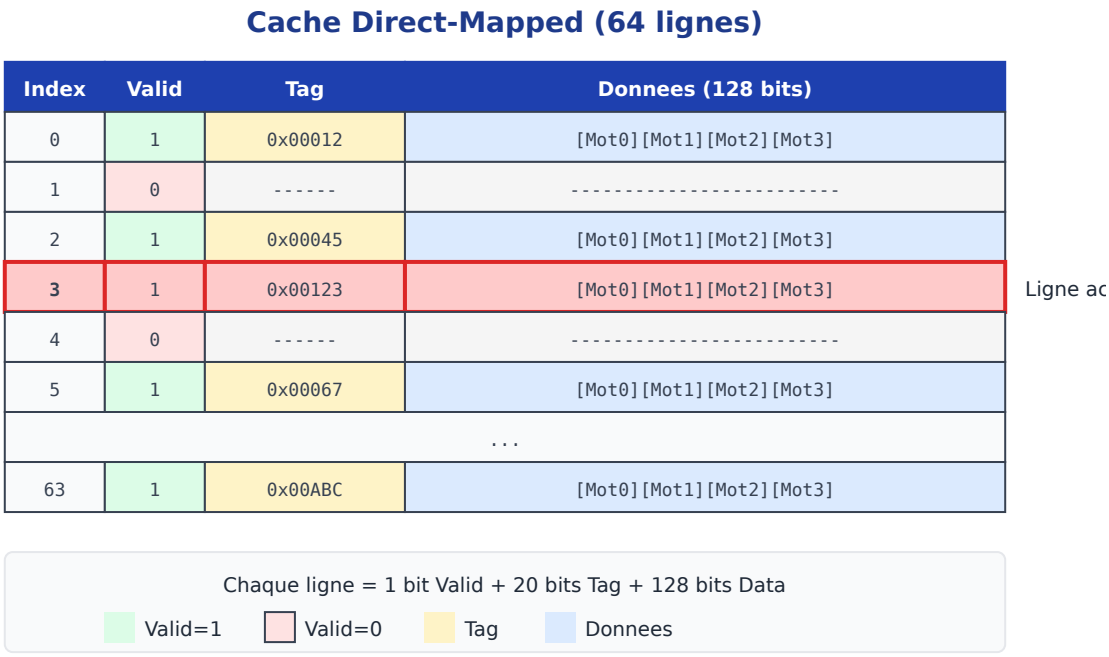


Figure 14.6: Cache direct-mapped

14.4 Hit ou Miss : Que se passe-t-il ?

14.4.1 Scénario 1 : Cache HIT (Succès)

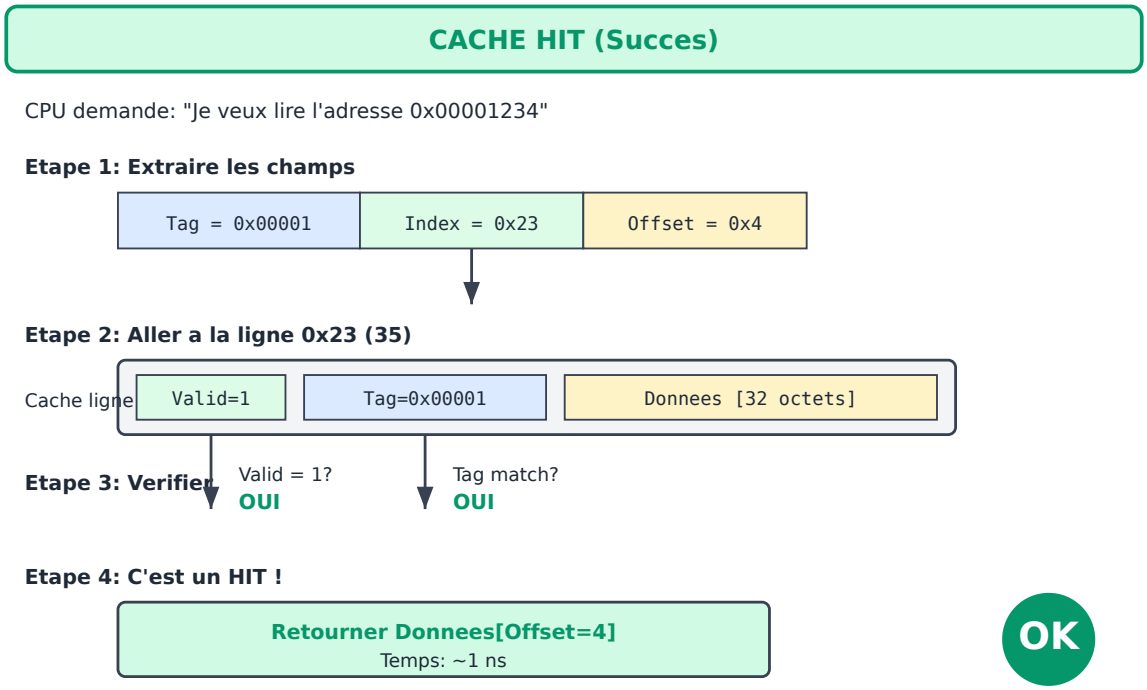


Figure 14.7: Scénario Cache HIT

14.4.2 Scénario 2 : Cache MISS (Échec)

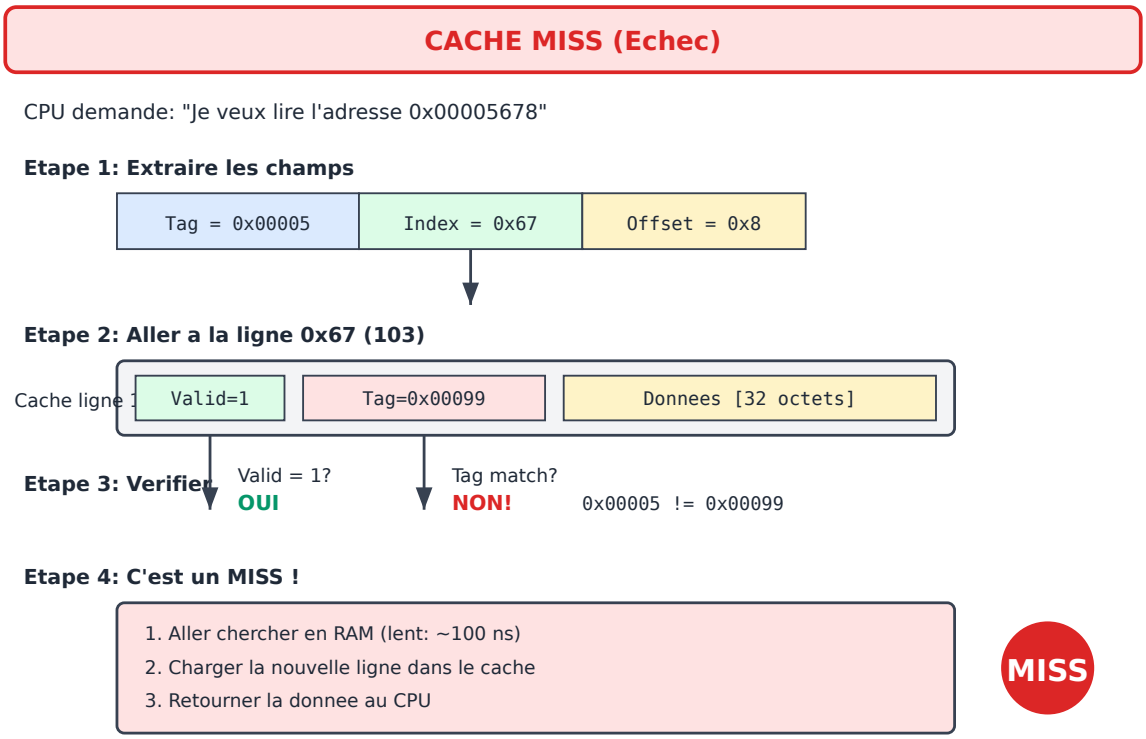


Figure 14.8: Scénario Cache MISS

14.4.3 Diagramme de Flux Complet

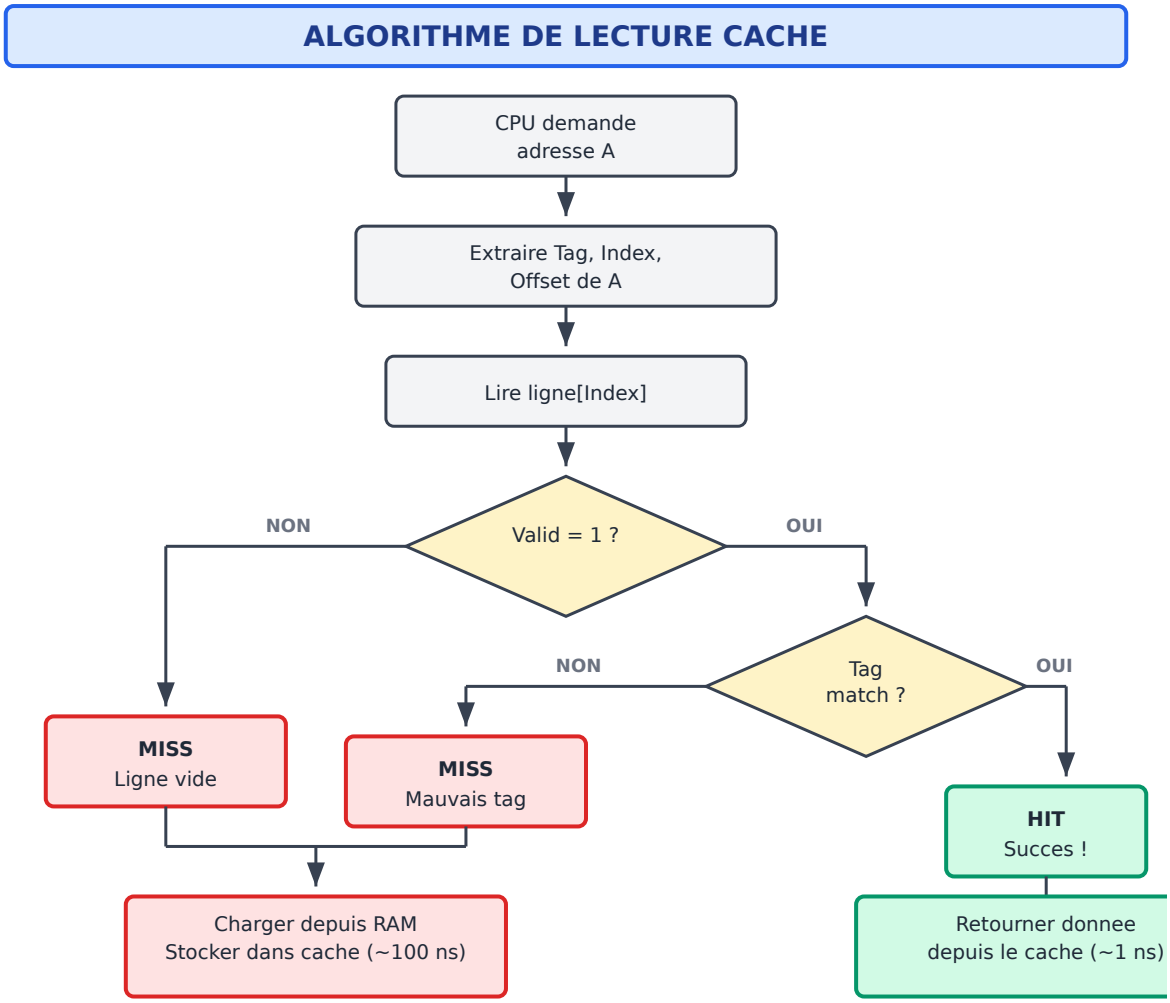


Figure 14.9: Algorithme de lecture cache

14.5 Politiques d'Écriture

14.5.1 Write-Through (Écriture Directe)

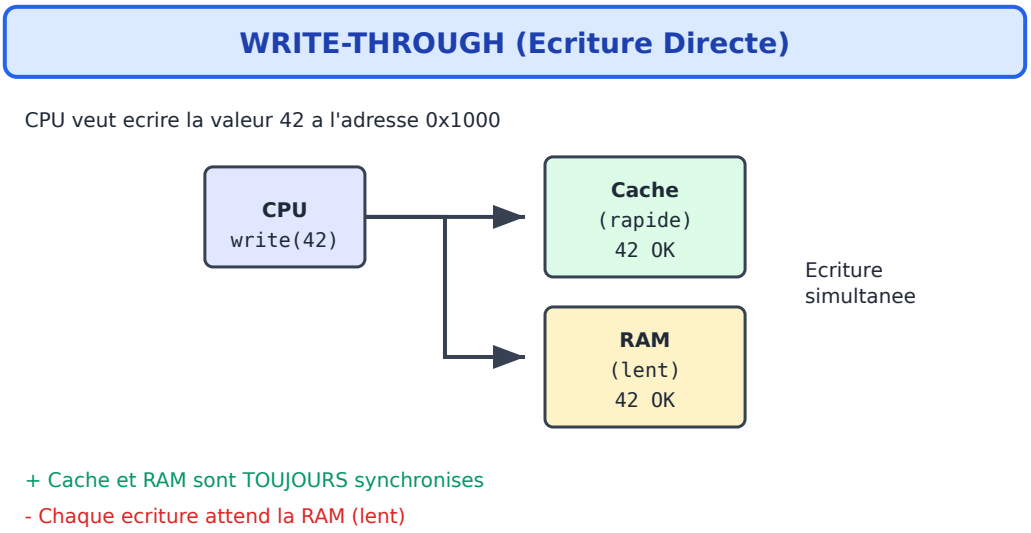


Figure 14.10: Write-Through

14.5.2 Write-Back (Écriture Différée)

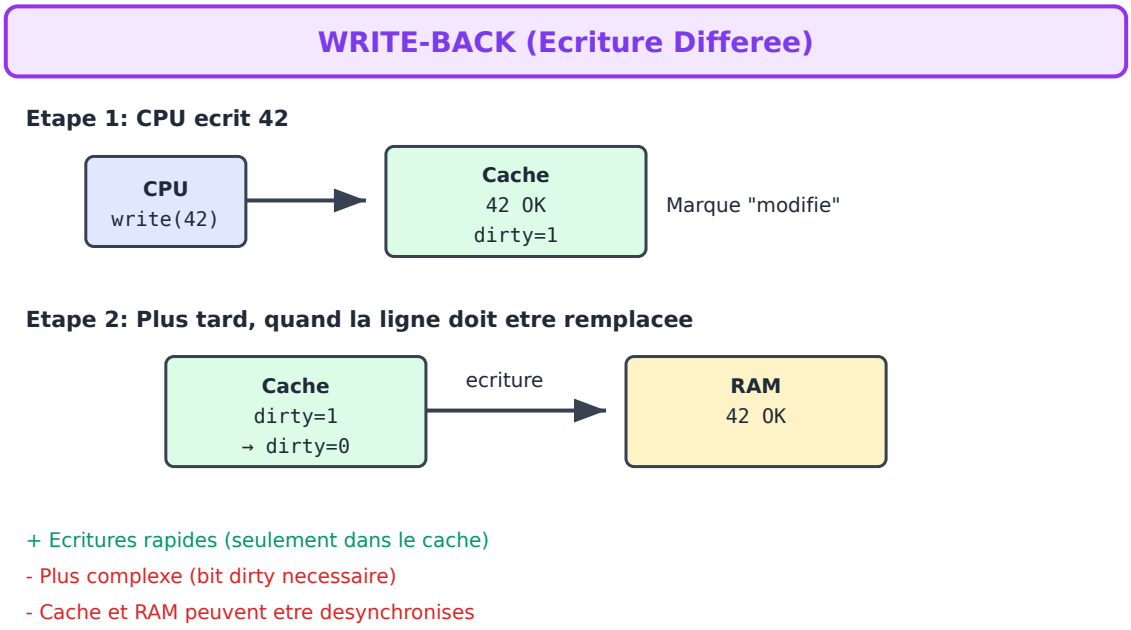


Figure 14.11: Write-Back

14.6 Types de Cache : Direct-Mapped vs Associatif

14.6.1 Cache Direct-Mapped (Correspondance Directe)

Dans un cache **direct-mapped**, chaque adresse mémoire correspond à **une seule ligne** possible dans le cache. C'est simple et rapide, mais peut causer des **conflits**.

```
Adresse → Index = (Adresse / Taille_Ligne) mod Nombre_Lignes

Exemple avec 4 lignes de cache :
- Adresse 0x000 → Ligne 0
- Adresse 0x100 → Ligne 0  ← Conflit !
- Adresse 0x200 → Ligne 0  ← Conflit !
```

Problème : Si votre programme alterne entre les adresses 0x000 et 0x100, chaque accès provoque un **miss** car elles partagent la même ligne !

14.6.2 Cache Fully-Associatif

Un cache **fully-associatif** permet à chaque bloc de se placer **n'importe où** dans le cache.

```
┌──────────────────────────────────────────────────┐
│ Cache Fully-Associatif (4 lignes)                │
├──────────┬──────────┬──────────┬──────────┬──────┤
│ Entrée   │ Valid    │ Tag      │ Data     │ LRU  │
├──────────┼──────────┼──────────┼──────────┼──────┤
│ 0        │ 1        │ 0x000    │ ...      │ 3    │
│ 1        │ 1        │ 0x100    │ ...      │ 1    │  ← Coexistent !
│ 2        │ 1        │ 0x200    │ ...      │ 0    │
│ 3        │ 0        │ ---      │ ---      │ 2    │
└──────────┴──────────┴──────────┴──────────┴──────┘
```

Avantage : Pas de conflits, les adresses 0x000, 0x100 et 0x200 coexistent. **Inconvénient** : Comparaison parallèle de tous les tags = hardware complexe et lent.

14.6.3 Cache Set-Associatif (N-Way)

Compromis entre les deux : le cache est divisé en **ensembles (sets)**, chaque adresse peut aller dans **N lignes** différentes au sein de son set.

```
Cache 2-Way Set-Associatif (4 sets × 2 ways = 8 lignes)
┌──────────────────────────────────────────────────────────┐
│ Set 0: │ Way 0: [0x000, data] │ Way 1: [0x100, data] │
│ Set 1: │ Way 0: [0x040, data] │ Way 1: [libre]       │
│ Set 2: │ Way 0: [0x080, data] │ Way 1: [0x280, data] │
│ Set 3: │ Way 0: [0x0C0, data] │ Way 1: [libre]       │
└──────────────────────────────────────────────────────────┘

Index du Set = (Adresse / Taille_Ligne) mod Nombre_Sets
```

Caches typiques des processeurs modernes : - L1 : 8-way associatif - L2 : 8-16 way associatif - L3 : 16-20 way associatif

---

14.7 Politiques de Remplacement

Quand le cache est plein et qu'un nouveau bloc doit être chargé, **quel bloc évincer** ?

14.7.1 LRU (Least Recently Used)

Principe : Évincer le bloc qui n'a pas été utilisé depuis le plus longtemps.

```
État initial (cache 4 lignes, ordre LRU : 3→2→1→0) :
┌─────────┬──────────┬──────────┐
│ Ligne   │ Donnée   │ LRU      │
├─────────┼──────────┼──────────┤
│ 0       │ Bloc A   │ 0 (MRU)  │  ← Most Recently Used
│ 1       │ Bloc B   │ 1        │
│ 2       │ Bloc C   │ 2        │
│ 3       │ Bloc D   │ 3 (LRU)  │  ← Least Recently Used
└─────────┴──────────┴──────────┘

Accès au Bloc E (miss) → Évincer Bloc D (LRU = 3)
┌─────────┬──────────┬──────────┐
│ 0       │ Bloc A   │ 1        │
│ 1       │ Bloc B   │ 2        │
│ 2       │ Bloc C   │ 3 (LRU)  │
│ 3       │ Bloc E   │ 0 (MRU)  │  ← Nouveau !
└─────────┴──────────┴──────────┘
```

Avantage : Exploite bien la localité temporelle. **Inconvénient** : Coûteux à implémenter pour N élevé (N! états possibles).

14.7.2 Pseudo-LRU (Tree-PLRU)

Approximation de LRU utilisant un arbre binaire de bits.

```
Cache 4-way avec 3 bits de direction :

        [bit0]
       /      \
   [bit1]    [bit2]
   /   \     /   \
Way0  Way1 Way2  Way3

Chaque bit pointe vers le sous-arbre "moins récent".
Pour évincer : suivre les bits jusqu'à une feuille.
Pour mettre à jour : inverser les bits sur le chemin vers le way accédé.
```

14.7.3 FIFO (First-In First-Out)

Principe : Évincer le bloc qui est dans le cache depuis le plus longtemps.

```
Ordre d'arrivée : A → B → C → D
Prochain miss : Évincer A (premier arrivé)
```

Simple mais ne tient pas compte de la fréquence d'utilisation.

14.7.4 Random (Aléatoire)

Principe : Évincer un bloc au hasard.

```
Utilisé par : ARM Cortex-A9, certains L2 caches
```

Avantage : Hardware très simple, pas d'état à maintenir. **Inconvénient** : Peut évincer des données utiles.

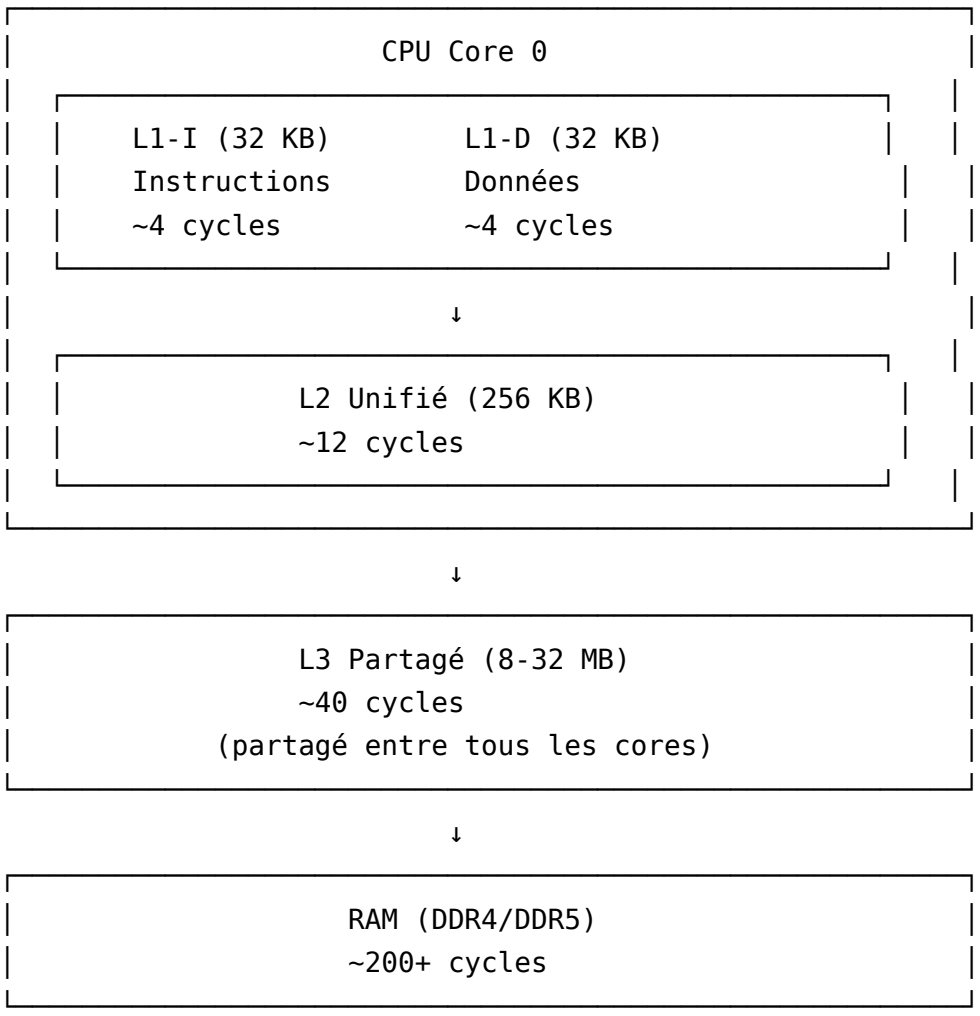
Fait surprenant : Random est souvent proche de LRU en pratique !

14.7.5 Comparaison des Politiques

Politique	Complexité HW	Performance	Cas d'utilisation
LRU	Élevée	Excellente	L1 cache (petit)
PLRU	Moyenne	Très bonne	L2/L3 cache
FIFO	Faible	Bonne	TLB, caches simples
Random	Très faible	Correcte	Grands caches

14.8 Caches Multi-Niveaux (L1/L2/L3)

Les processeurs modernes utilisent une **hiérarchie de caches** :



14.8.1 Caractéristiques Typiques (2024)

Niveau	Taille	Latence	Associativité	Partagé
L1-I	32-64 KB	4 cycles	8-way	Non (par core)
L1-D	32-64 KB	4-5 cycles	8-12 way	Non (par core)
L2	256 KB-1 MB	12-14 cycles	8-16 way	Non (par core)
L3	8-96 MB	30-50 cycles	16-20 way	Oui (tous cores)

14.8.2 Politiques d'Inclusion

Inclusive : Tout ce qui est dans L1 est aussi dans L2 et L3. - Avantage : Cohérence simple - Inconvénient : Gaspillage d'espace

Exclusive : Les données ne sont que dans UN niveau à la fois. - Avantage : Capacité totale = L1 + L2 + L3 - Inconvénient : Gestion plus complexe

NINE (Non-Inclusive Non-Exclusive) : Pas de garantie. - Utilisé par Intel depuis Skylake pour les grands L3

14.9 Patterns de Code et Localité

14.9.1 Pattern 1 : Parcours Séquentiel (Localité Spatiale)

```
// EXCELLENT : Accès séquentiels
int sum = 0;
for (int i = 0; i < 1000; i++) {
    sum += array[i];  // Adresses consécutives
}
// Hit rate ~94% (1 miss pour 16 hits si ligne = 64 octets, int = 4 octets)
```

14.9.2 Pattern 2 : Accès avec Stride (Mauvaise Localité)

```
// MAUVAIS : Stride de 64 octets = 1 ligne de cache
for (int i = 0; i < 1000; i += 16) {
    sum += array[i];  // Saute une ligne entière !
}
// Hit rate ~0% : chaque accès est un miss
```

14.9.3 Pattern 3 : Réutilisation de Variables (Localité Temporelle)

```
// BON : Même variable réutilisée
int accumulator = 0;
for (int i = 0; i < 1000; i++) {
    accumulator += array[i];
    accumulator *= 2;
    accumulator -= 1;  // 'accumulator' reste en registre/L1
}

// MAUVAIS : Variable différente à chaque fois
for (int i = 0; i < 1000; i++) {
    temps[i] = array[i] * 2;  // Écrit dans tout le tableau
}
for (int i = 0; i < 1000; i++) {
    result += temps[i];       // Relit tout le tableau
}
```

14.9.4 Pattern 4 : Structure de Données Cache-Friendly

```
// MAUVAIS : Array of Structures (AoS)
struct Particule {
    float x, y, z;      // Position
    float vx, vy, vz;   // Vitesse (souvent inutilisé)
    float mass;
    int type;
};
Particule particles[10000];

for (int i = 0; i < 10000; i++) {
    particles[i].x += particles[i].vx;  // Charge 32 octets, utilise 8
}

// BON : Structure of Arrays (SoA)
struct Particules {
    float x[10000];
    float y[10000];
    float z[10000];
    float vx[10000];
    // ...
};
Particules p;

for (int i = 0; i < 10000; i++) {
    p.x[i] += p.vx[i];  // Accès séquentiel, 100% utile
}
```

14.9.5 Pattern 5 : Loop Tiling / Blocking

```
// NAÏF : Multiplication de matrices
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            C[i][j] += A[i][k] * B[k][j];  // B[k][j] : accès par colonne !
        }
    }
}

// OPTIMISÉ : Blocking (blocs de 32×32)
#define BLOCK 32
for (int ii = 0; ii < N; ii += BLOCK) {
    for (int jj = 0; jj < N; jj += BLOCK) {
        for (int kk = 0; kk < N; kk += BLOCK) {
            // Traiter le bloc [ii:ii+BLOCK, jj:jj+BLOCK]
            for (int i = ii; i < ii + BLOCK; i++) {
                for (int j = jj; j < jj + BLOCK; j++) {
                    for (int k = kk; k < kk + BLOCK; k++) {
                        C[i][j] += A[i][k] * B[k][j];
                    }
                }
            }
        }
    }
}
// Le bloc de B tient en cache → réutilisé N/BLOCK fois
```

14.10 Modélisation des Performances

14.10.1 Temps d'Accès Moyen (AMAT)

```
AMAT = Hit_Time + Miss_Rate × Miss_Penalty

Exemple :
- Hit Time L1 = 4 cycles
- Miss Rate L1 = 5%
- Miss Penalty (accès L2) = 12 cycles

AMAT = 4 + 0.05 × 12 = 4.6 cycles
```

14.10.2 AMAT Multi-Niveaux

```
AMAT = Hit_Time_L1 + Miss_Rate_L1 × (
         Hit_Time_L2 + Miss_Rate_L2 × (
           Hit_Time_L3 + Miss_Rate_L3 × RAM_Latency
         )
       )

Exemple réaliste :
- L1 : 4 cycles, 5% miss
- L2 : 12 cycles, 20% miss (de ce qui arrive en L2)
- L3 : 40 cycles, 30% miss
- RAM : 200 cycles

AMAT = 4 + 0.05 × (12 + 0.20 × (40 + 0.30 × 200))
     = 4 + 0.05 × (12 + 0.20 × (40 + 60))
     = 4 + 0.05 × (12 + 20)
     = 4 + 0.05 × 32
     = 4 + 1.6
     = 5.6 cycles

Sans cache : 200 cycles → Le cache accélère par 35× !
```

14.10.3 Exercice de Calcul

Calculez l'AMAT pour ce système : - L1 : 3 cycles, miss rate 8% - L2 : 10 cycles, miss rate 25% - RAM : 150 cycles

Solution :

```
AMAT = 3 + 0.08 × (10 + 0.25 × 150)
     = 3 + 0.08 × (10 + 37.5)
     = 3 + 0.08 × 47.5
     = 3 + 3.8
     = 6.8 cycles
```

14.11 Impact sur vos Programmes

14.11.1 Parcours de Tableaux 2D : L'Ordre Compte !

Une matrice 4x4 est stockée **ligne par ligne** en mémoire (row-major) :

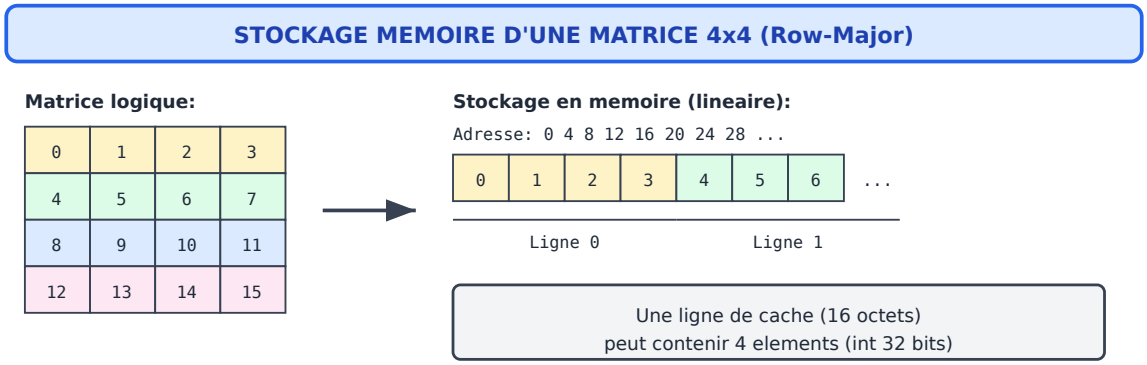


Figure 14.12: Stockage mémoire d'une matrice

14.11.2 Parcours Row-Major (En Ligne) - EFFICACE

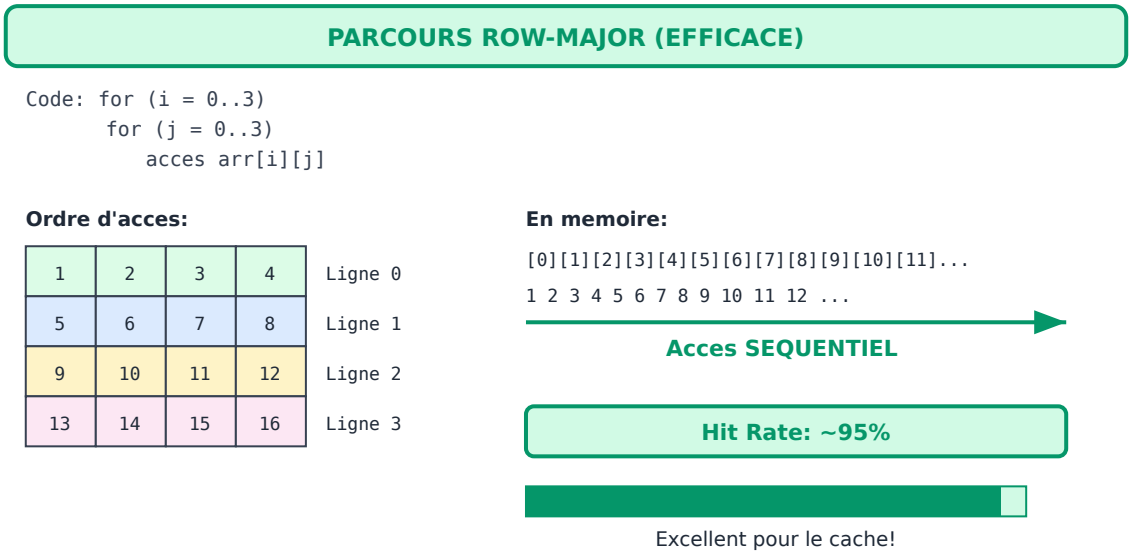


Figure 14.13: Parcours Row-Major

14.11.3 Parcours Column-Major (En Colonne) - INEFFICACE

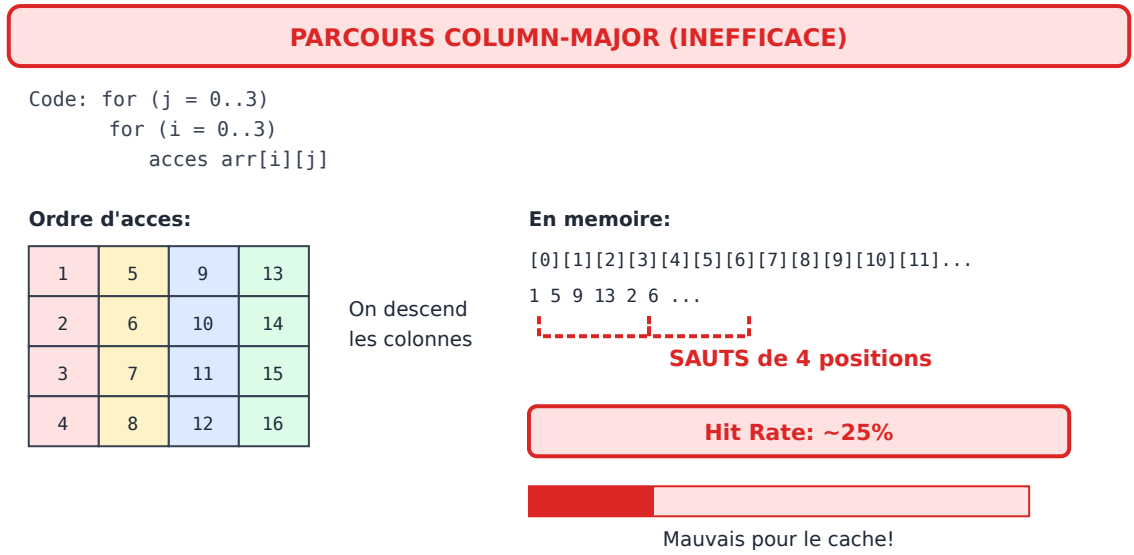


Figure 14.14: Parcours Column-Major

14.11.4 Comparaison Visuelle

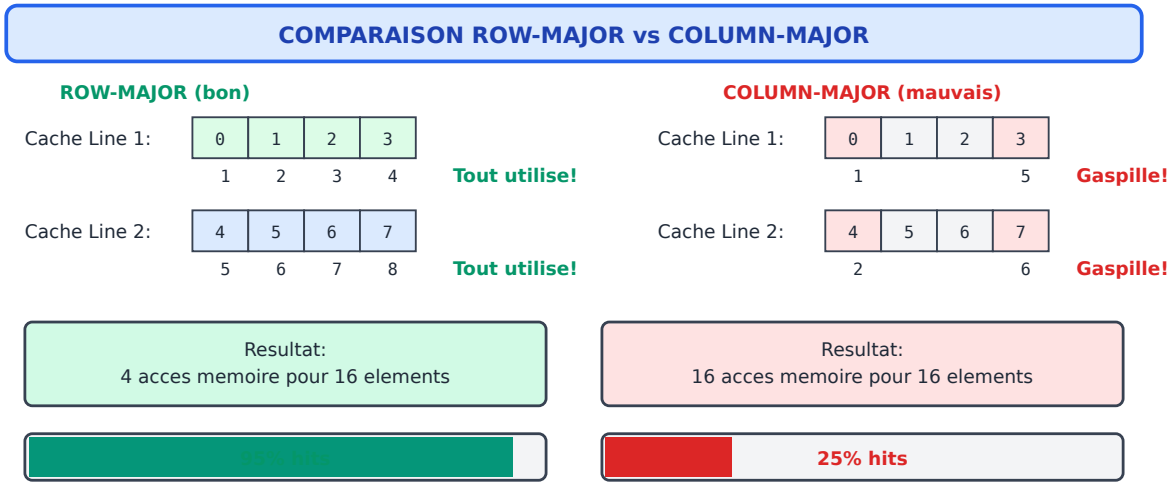


Figure 14.15: Comparaison Row-Major vs Column-Major

14.11.5 Technique du Blocking

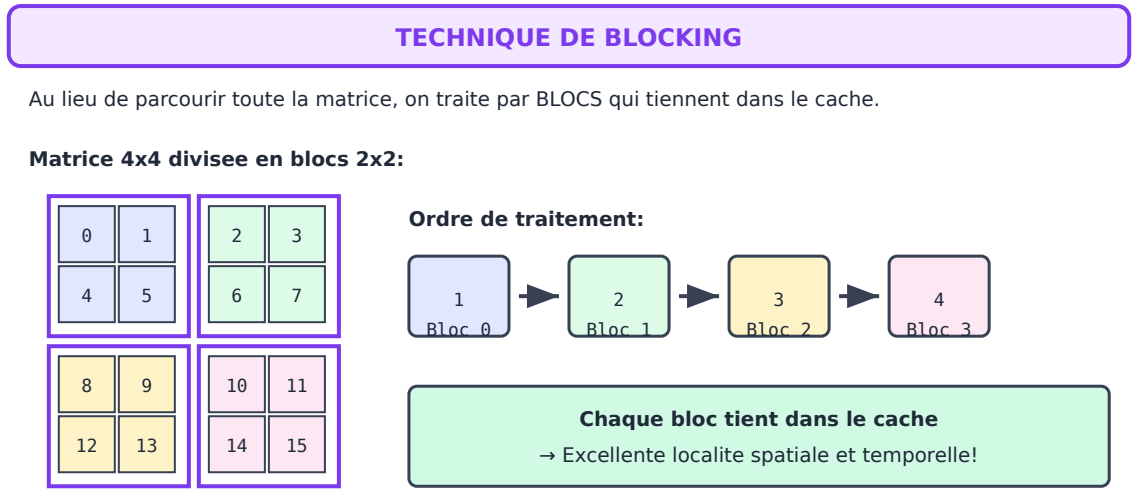


Figure 14.16: Technique de Blocking

14.12 Implémentation HDL du Cache

14.12.1 Architecture Globale

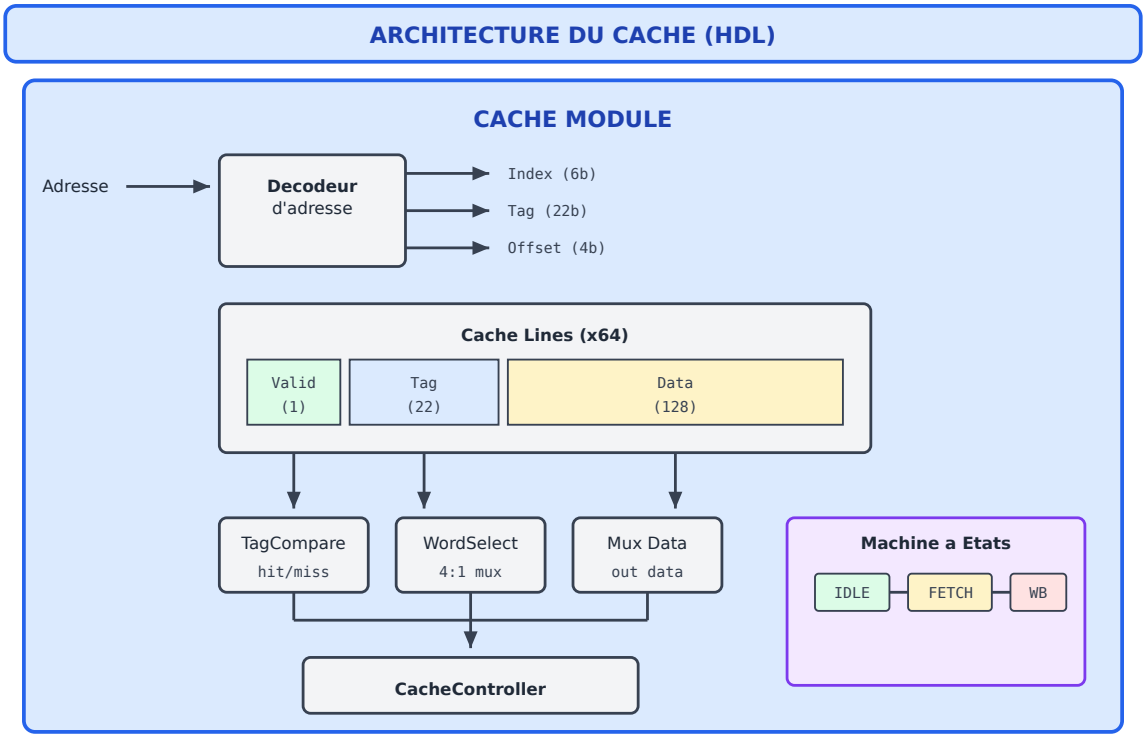


Figure 14.17: Architecture du cache HDL

14.12.2 Machine à États du Contrôleur

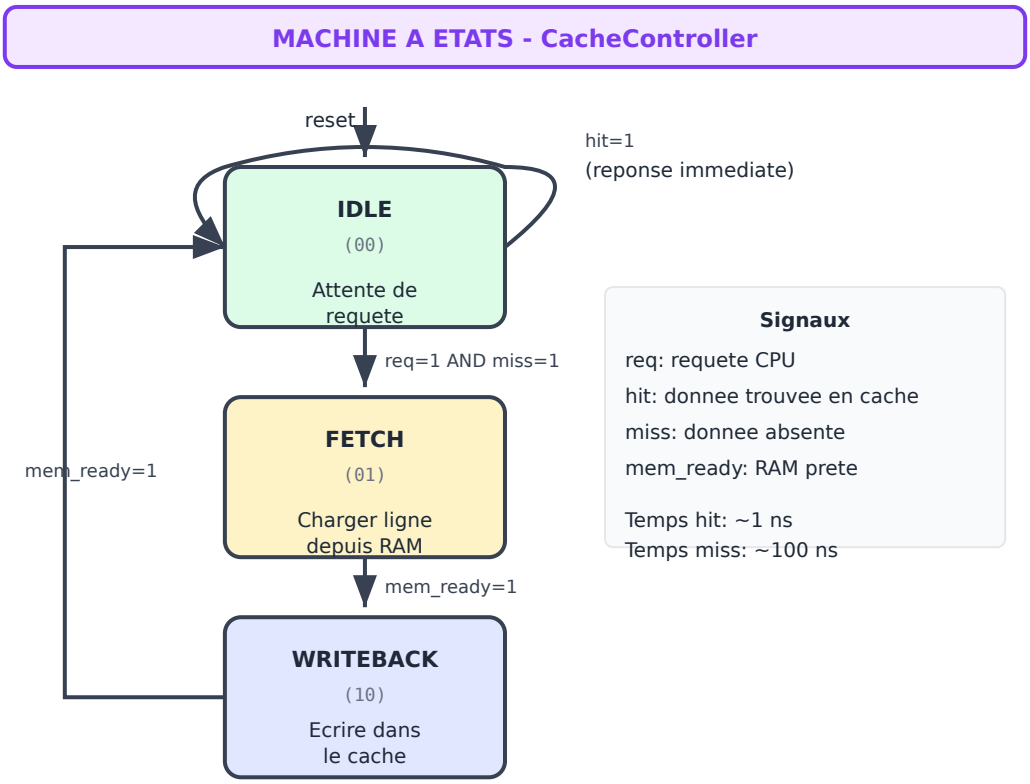


Figure 14.18: Machine à états du CacheController

14.12.3 WordSelect : Sélection du Mot

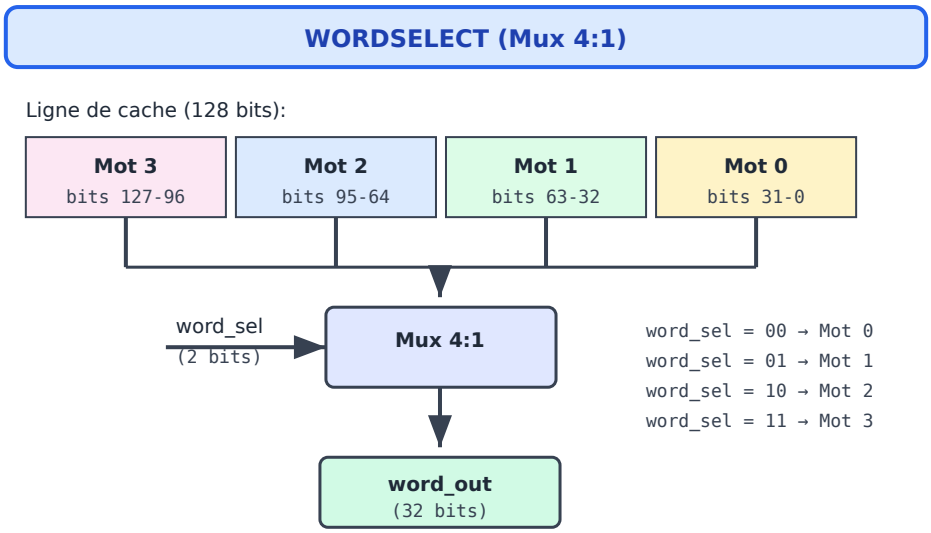


Figure 14.19: WordSelect

14.13 Statistiques et Performance

14.13.1 Calcul du Hit Rate

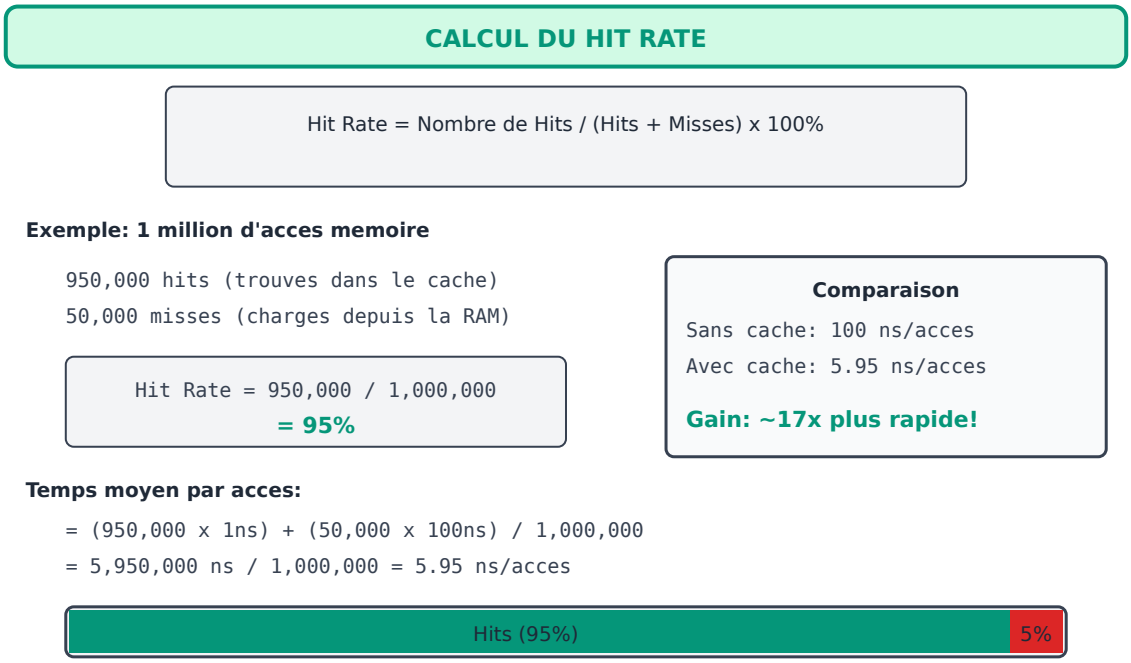


Figure 14.20: Calcul du Hit Rate

14.14 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.

14.14.1 Accéder au Visualizer

```
cd web
npm run dev
# Ouvrir http://localhost:5173 -> CPU Visualizer
```

14.14.2 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)

14.14.3 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

14.14.4 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

14.15 Exercices

14.15.1 Exercices HDL

Exercice	Description
CacheLine	Implémenter une ligne de cache avec valid, tag, data
TagCompare	Comparateur de tags pour détecter hit/miss
WordSelect	Sélecteur de mot (4:1) dans une ligne 128 bits
CacheController	Machine à états (IDLE, FETCH, WRITEBACK)

14.15.2 Exercices Assembleur A32

Exercice	Description	Résultat
Accès Séquentiel	Parcours cache-friendly d’un tableau	R0 = 100
Accès avec Stride	Parcours avec sauts (moins efficace)	R0 = 28
Réutilisation Registre	Garder les données en registre	R0 = 91

14.15.3 Exercices C32

Exercice	Description	Résultat
Parcours en Ligne	Accès row-major (cache-friendly)	120
Parcours en Colonne	Accès column-major (moins efficace)	120
Traitement par Blocs	Technique de blocking	120
Localité Temporelle	Réutiliser les données	30

14.16 Résumé Visuel

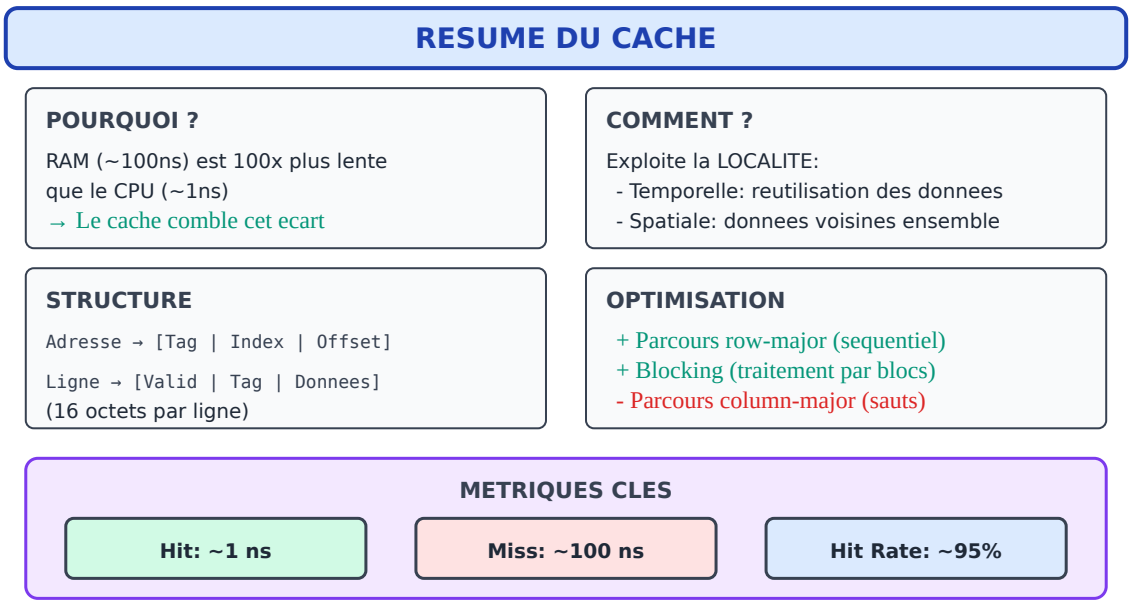


Figure 14.21: Résumé du cache

14.17 Points Clés à Retenir

1. **La RAM est lente** : ~100x 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 !

14.18 Auto-évaluation

Testez votre compréhension du cache.

14.18.1 Questions de compréhension

Q1. Qu'est-ce que la localité temporelle et la localité spatiale ?

Q2. Pourquoi le parcours en ligne (row-major) est-il plus efficace que le parcours en colonne ?

Q3. Qu'est-ce qu'un cache hit et un cache miss ?

Q4. Comment fonctionne un cache direct-mapped ?

Q5. Qu'est-ce que la technique de "blocking" ?

14.18.2 Mini-défi pratique

Calculez le hit rate pour ce pattern d'accès avec un cache de 4 lignes de 16 octets :

Accès séquentiels aux adresses : 0, 4, 8, 12, 16, 20, 24, 28, 0, 4, 8, 12

*Les solutions se trouvent dans le document **Codex_Solutions**.*

14.18.3 Checklist de validation

- ☐ Expliquer la différence entre localité temporelle et spatiale
- ☐ Comprendre pourquoi le cache existe (hiérarchie mémoire)
- ☐ Calculer un hit rate simple
- ☐ Optimiser un parcours de tableau pour le cache
- ☐ Appliquer la technique de blocking sur une matrice

15 Les Interruptions : Quand le Monde Extérieur Frappe à la Porte

Imaginez que vous êtes concentré sur un travail important quand soudain : - Votre téléphone sonne - Quelqu'un frappe à la porte - L'alarme incendie se déclenche

Dans chaque cas, vous devez **interrompre** votre travail actuel, **gérer** l'événement, puis **reprendre** où vous en étiez.

Les **interruptions** en informatique fonctionnent exactement de la même façon : elles permettent au CPU de réagir à des événements externes sans avoir à constamment vérifier leur état.

15.1 Pourquoi les Interruptions ?

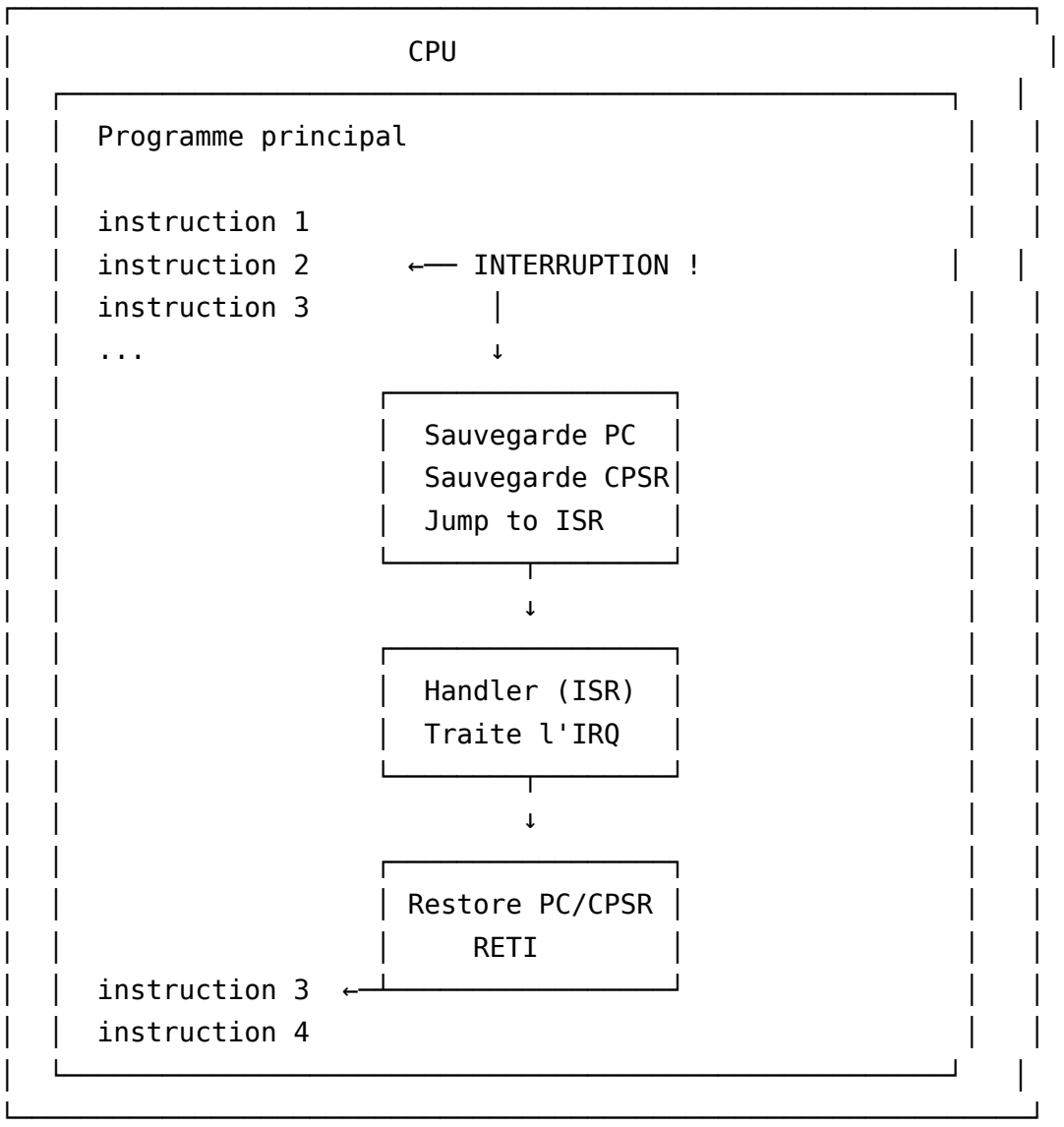
15.1.1 Le Problème : Le Polling

Sans interruptions, le CPU doit constamment vérifier si quelque chose s'est passé :

```
// MAUVAIS : Polling (attente active)
while (1) {
    if (keyboard_has_key()) {
        handle_key();
    }
    if (timer_expired()) {
        handle_timer();
    }
    if (network_has_data()) {
        handle_network();
    }
    // Le CPU ne peut rien faire d'autre !
}
```

Problèmes du polling : - Gaspille des cycles CPU - Latence variable (peut rater des événements) - Plus il y a de périphériques, pire c'est

15.1.2 La Solution : Les Interruptions



Avantages : - Le CPU travaille normalement jusqu'à l'événement - Réponse immédiate (latence minimale) - Scale bien avec le nombre de périphériques

---

15.2 Types d'Interruptions

15.2.1 Interruptions Matérielles (IRQ)

Déclenchées par des périphériques externes :

Source	Exemple	Priorité Typique
Timer	Tick système (100 Hz - 1 kHz)	Haute
Clavier	Touche pressée	Moyenne
Souris	Mouvement, clic	Moyenne
Disque	Transfert DMA terminé	Haute
Réseau	Paquet reçu	Haute
USB	Périphérique connecté	Basse

15.2.2 Interruptions Logicielles (Traps/SWI)

Déclenchées intentionnellement par le programme :

```
; Appel système (syscall)
MOV R0, #1      ; Code syscall = write
MOV R1, #buffer ; Adresse buffer
MOV R2, #10     ; Longueur
SWI #0          ; Software Interrupt → appelle l'OS
```

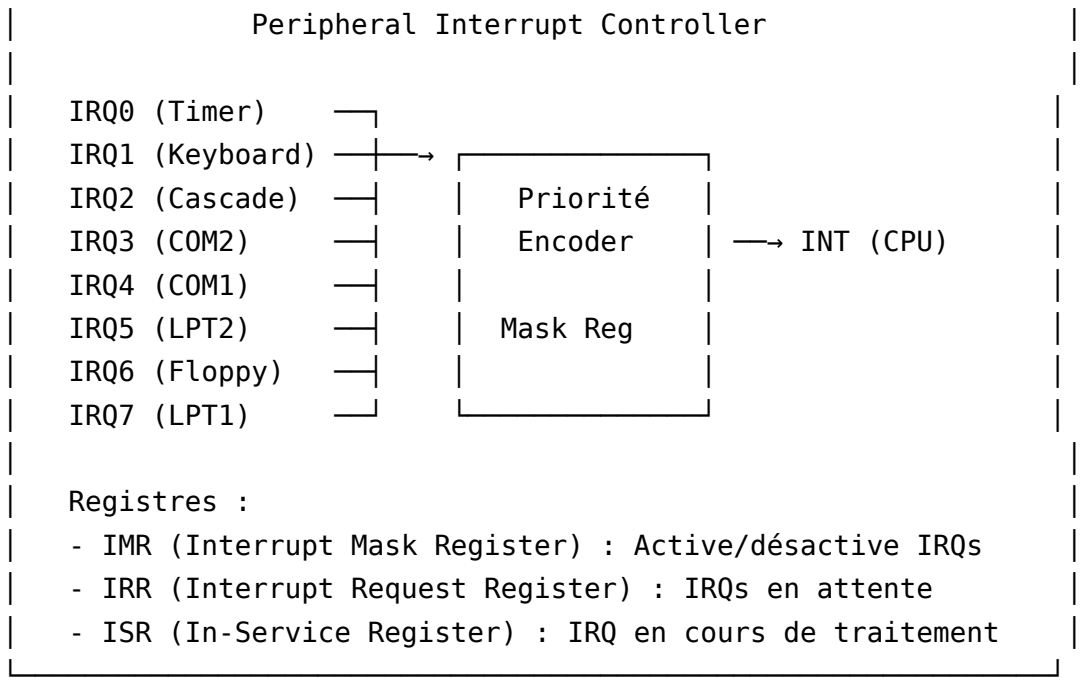
15.2.3 Exceptions

Erreurs pendant l'exécution :

Exception	Cause	Récupérable ?
Division par zéro	`DIV R0, R1, #0`	Oui
Adresse invalide	Accès hors mémoire	Parfois
Instruction invalide	Opcode inconnu	Non
Page fault	Page non en RAM	Oui (MMU)
Breakpoint	Débogage	Oui

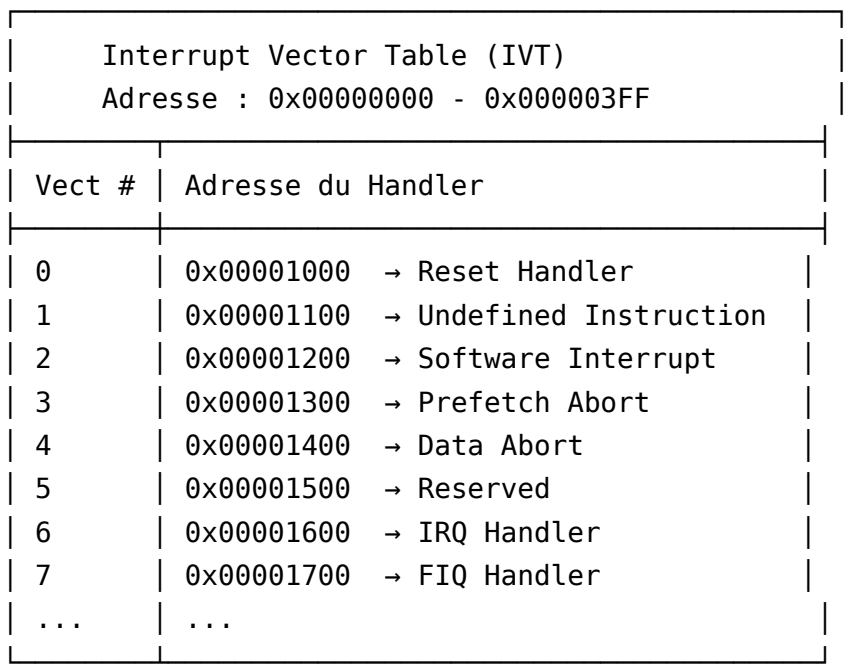
15.3 Architecture d'un Système d'Interruptions

15.3.1 Le Contrôleur d'Interruptions (PIC/APIC)



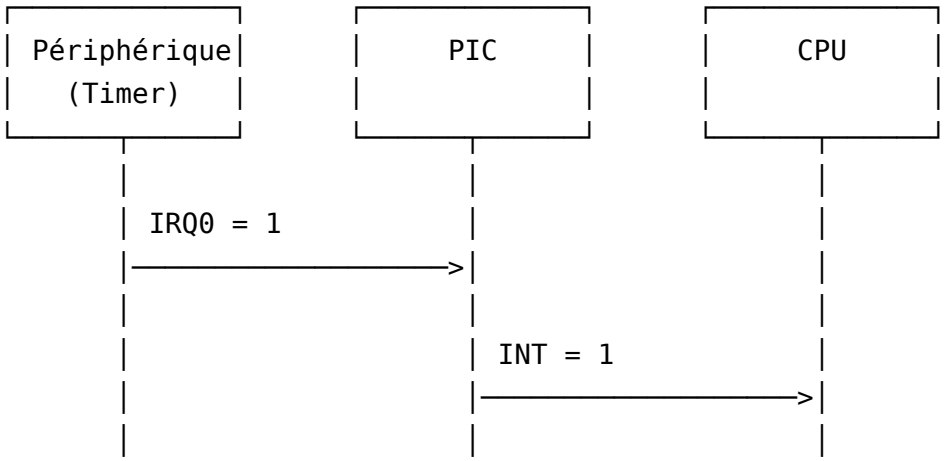
15.3.2 La Table des Vecteurs d'Interruption (IVT)

Chaque interruption a un **numéro** qui indexe une table de pointeurs vers les handlers :

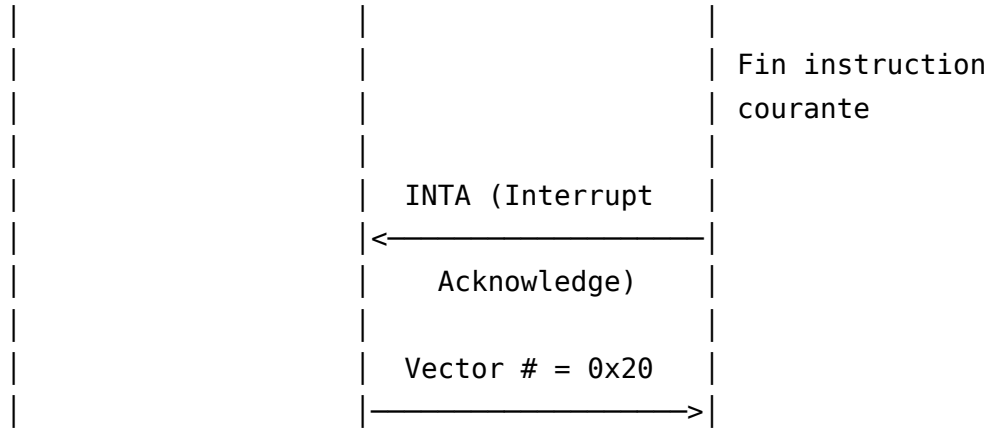


15.4 Le Cycle de Vie d'une Interruption

15.4.1 1. Détection



15.4.2 2. Reconnaissance



15.4.3 3. Sauvegarde du Contexte

Le CPU sauvegarde automatiquement :

```
; Ce que le CPU fait automatiquement :
; 1. Désactive les interruptions (I flag = 1)
; 2. Sauvegarde PC dans LR_irq
; 3. Sauvegarde CPSR dans SPSR_irq
; 4. Passe en mode IRQ
; 5. PC = adresse du handler (depuis IVT)
```

15.4.4 4. Exécution du Handler (ISR)

```
irq_handler:
    ; Sauvegarder les registres utilisés
    PUSH {R0-R3, R12, LR}

    ; Identifier la source de l'IRQ
    LDR R0, =PIC_BASE
    LDR R1, [R0, #IRR_OFFSET]    ; Lire les IRQ en attente

    ; Dispatcher vers le bon handler
    TST R1, #0x01                ; IRQ0 (Timer) ?
    BNE timer_handler
    TST R1, #0x02                ; IRQ1 (Keyboard) ?
    BNE keyboard_handler
    ; ...

    B irq_done

timer_handler:
    ; Traiter le timer
    BL handle_timer_tick
    ; Acquitter l'IRQ au PIC
    LDR R0, =PIC_BASE
    MOV R1, #0x20                ; EOI (End Of Interrupt)
    STR R1, [R0, #EOI_OFFSET]
    B irq_done

keyboard_handler:
    ; Lire la touche
    LDR R0, =KEYBOARD_BASE
    LDR R1, [R0]                 ; Scancode
    BL handle_keypress
    ; Acquitter
    LDR R0, =PIC_BASE
    MOV R1, #0x20
    STR R1, [R0, #EOI_OFFSET]
    B irq_done

irq_done:
    ; Restaurer les registres
    POP {R0-R3, R12, LR}
    ; Retour d'interruption
    SUBS PC, LR, #4              ; RETI : restore PC et CPSR
```

15.4.5 5. Retour (RETI)

```
SUBS PC, LR, #4
; Cette instruction spéciale :
; 1. Restaure PC depuis LR (moins 4 pour pipeline)
; 2. Restaure CPSR depuis SPSR
; 3. Réactive les interruptions
; 4. Reprend le programme interrompu
```

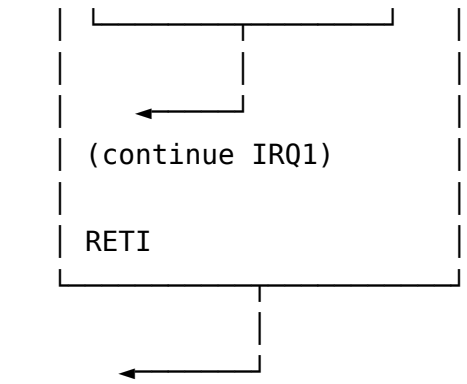
15.5 Gestion des Priorités

15.5.1 Priorité Fixe vs Programmable

```
┌─────────────────────────────────────────────────────────────┐
│               Système de Priorités                          │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  Priorité 0 (Haute) : Reset, NMI                            │
│  Priorité 1         : Data Abort                            │
│  Priorité 2         : FIQ (Fast Interrupt)                  │
│  Priorité 3         : IRQ                                   │
│  Priorité 4 (Basse) : Software Interrupt                    │
│                                                             │
│  Règle : Une IRQ de priorité N peut interrompre             │
│          un handler de priorité < N                         │
└─────────────────────────────────────────────────────────────┘
```

15.5.2 Interruptions Imbriquées (Nested)

```
Programme principal
       │
       ▼ IRQ1 (basse priorité)
   ┌──────────────────────┐
   │ Handler IRQ1         │
   │    │                 │
   │    ▼ IRQ2 (haute)    │
   │ ┌──────────────┐     │
   │ │ Handler IRQ2 │     │
   │ │              │     │
   │ │ RETI         │     │
```

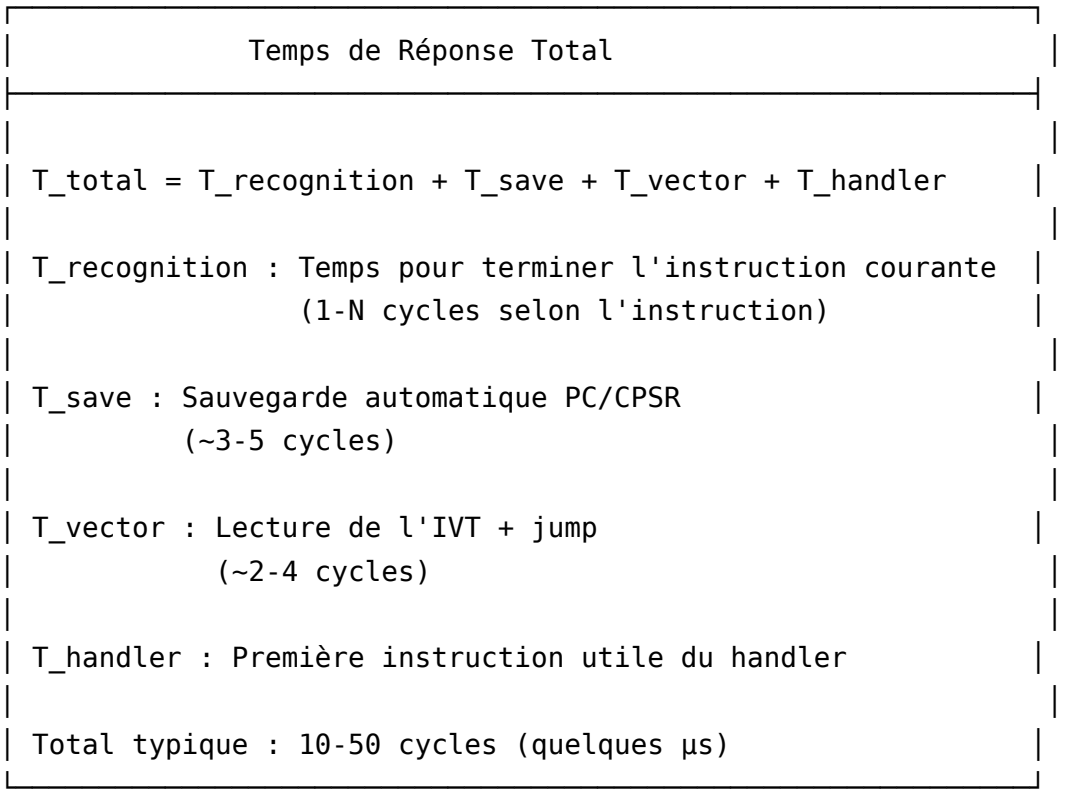


Pour autoriser les interruptions imbriquées :

```
nested_irq_handler:
    ; Sauver le contexte
    SUB LR, LR, #4
    PUSH {LR}
    MRS LR, SPSR
    PUSH {LR}
    PUSH {R0-R12}

    ; IMPORTANT : Réactiver les interruptions
    ; pour permettre le nesting
    MRS R0, CPSR
    BIC R0, R0, #0x80    ; Clear I bit
    MSR CPSR_c, R0

    ; Traiter l'IRQ...
    BL do_irq_work

    ; Désactiver avant de restaurer
    MRS R0, CPSR
    ORR R0, R0, #0x80
    MSR CPSR_c, R0

    ; Restaurer
    POP {R0-R12}
    POP {LR}
    MSR SPSR_cxsf, LR
    POP {PC}^            ; Restore avec SPSR
```

15.6 Latence d'Interruption

15.6.1 Composants de la Latence



15.6.2 Optimiser la Latence

1. **FIQ (Fast Interrupt Request)** : Registres banqués dédiés

   ```
   Mode FIQ : R8_fiq à R14_fiq sont séparés
   → Pas besoin de PUSH/POP = gain de 20+ cycles
   ```

2. **Handlers courts** : Faire le minimum, déférer le reste

   ```c
   // MAUVAIS : tout dans l'ISR
   void timer_isr() {
       update_all_timers();     // Long !
       recalculate_scheduling(); // Très long !
       ack_interrupt();
   }

   // BON : flag + traitement différé
   volatile int timer_pending = 0;
   void timer_isr() {
       timer_pending = 1;
       ack_interrupt();
   ```

```
}
// Dans la boucle principale ou thread dédié :
if (timer_pending) {
    timer_pending = 0;
    do_heavy_work();
}
```

3. **Interrupt coalescing** : Regrouper plusieurs événements

```
Au lieu de : 1 IRQ par paquet réseau
Faire      : 1 IRQ pour N paquets ou après T ms
```

15.7 Interruptions et Systèmes d'Exploitation

15.7.1 Le Timer : Le Coeur Battant de l'OS

```
// Tick système (ex: 1000 Hz = toutes les 1 ms)
void timer_handler() {
    // 1. Incrémenter le compteur système
    system_ticks++;

    // 2. Vérifier les timeouts
    check_sleeping_threads();

    // 3. Time-slice scheduling
    current_thread->time_remaining--;
    if (current_thread->time_remaining == 0) {
        schedule();  // Préemption !
    }

    // 4. Mise à jour des statistiques
    if (in_user_mode) user_time++;
    else kernel_time++;
}
```

15.7.2 Changement de Contexte (Context Switch)

Quand le scheduler décide de changer de thread :

```
context_switch:
    ; Sauver le contexte du thread courant
```

```
    LDR R0, =current_thread
    LDR R0, [R0]

    ; Sauver tous les registres dans la structure thread
    STMIA R0, {R0-R14}^     ; Registres user mode
    MRS R1, CPSR
    STR R1, [R0, #60]       ; Sauver CPSR

    ; Charger le nouveau thread
    LDR R0, =next_thread
    LDR R0, [R0]

    ; Restaurer son contexte
    LDR R1, [R0, #60]
    MSR CPSR_cxsf, R1
    LDMIA R0, {R0-R14}^

    ; Mettre à jour current_thread
    ; ...

    ; Retour (au nouveau thread)
    MOVS PC, LR
```

15.8 Problèmes de Concurrence

15.8.1 Race Conditions

```
// PROBLÈME : variable partagée sans protection
volatile int counter = 0;

void main_program() {
    // Peut être interrompu entre ces opérations !
    int temp = counter;     // ← IRQ ICI ?
    temp = temp + 1;        // ← OU ICI ?
    counter = temp;         // ← OU ICI ?
}

void irq_handler() {
    counter++;  // Modification pendant que main lit ?
}

// Résultat possible :
// main lit counter = 5
```

```
// IRQ: counter = 6
// main écrit counter = 6  ← Incrémentation perdue !
```

15.8.2 Sections Critiques

```
// SOLUTION : désactiver les interruptions
void safe_increment() {
    disable_interrupts();   // CLI
    counter++;
    enable_interrupts();    // STI
}

// En assembleur A32 :
safe_increment:
    MRS R1, CPSR
    ORR R2, R1, #0x80       ; Set I bit
    MSR CPSR_c, R2          ; Disable IRQ

    LDR R0, =counter
    LDR R3, [R0]
    ADD R3, R3, #1
    STR R3, [R0]

    MSR CPSR_c, R1          ; Restore (enable if was)
    BX LR
```

15.8.3 Attention : Ne Pas Rester Trop Longtemps !

```
// MAUVAIS : interruptions désactivées trop longtemps
void bad_function() {
    disable_interrupts();
    do_long_operation();    // 10 ms !
    enable_interrupts();
}
// Risque : perte d'IRQ, latence horrible

// BON : minimiser la section critique
void good_function() {
    // Préparation hors section critique
    prepare_data();

    disable_interrupts();
    quick_update();         // < 10 µs
```

```
    enable_interrupts();

    // Traitement hors section critique
    process_result();
}
```

15.9 Exercices Pratiques

15.9.1 Exercice 1 : Gestionnaire de Timer Simple

Implémentez un handler de timer qui compte les secondes :

```
; Variables globales
.data
ticks:      .word 0
seconds:    .word 0

.text
; TODO: Implémenter timer_handler
; - Incrémenter ticks
; - Si ticks == 1000, incrémenter seconds et reset ticks
; - Acquitter l'interruption
timer_handler:
    ; Votre code ici
```

15.9.2 Exercice 2 : Buffer Circulaire pour Clavier

```
// Implémenter un buffer circulaire pour stocker les touches
#define BUFFER_SIZE 16

typedef struct {
    char data[BUFFER_SIZE];
    int head;  // Prochain emplacement d'écriture (ISR)
    int tail;  // Prochain emplacement de lecture (main)
} RingBuffer;

// TODO: Implémenter
void keyboard_isr();        // Ajoute une touche au buffer
int get_key();              // Lit une touche (bloquant ou non)
int buffer_empty();         // Test si vide
```

15.9.3 Exercice 3 : Ordonnanceur Simple

```
// Implémenter un scheduler round-robin basé sur timer
#define MAX_THREADS 4
#define TIME_SLICE 10  // ms

typedef struct {
    int id;
    int state;          // READY, RUNNING, BLOCKED
    void* stack_ptr;
    int time_remaining;
} Thread;

// TODO: Implémenter
void scheduler_init();
void timer_isr();           // Décrémente time_remaining, schedule si 0
void schedule();            // Choisit le prochain thread
void context_switch();      // Change de thread
```

15.9.4 Exercice 4 : Analyse de Latence

Calculez la latence d'interruption pour : - Instruction en cours : LDM (chargement de 8 registres) - Sauvegarde automatique : 3 cycles - Fetch du vecteur : 2 cycles - Handler : PUSH de 6 registres

---

15.10 Auto-évaluation

15.10.1 Quiz

Q1. Quelle est la différence entre une interruption matérielle et une exception ?

Q2. Pourquoi le polling est-il inefficace comparé aux interruptions ?

Q3. Que contient la table des vecteurs d'interruption (IVT) ?

Q4. Qu'est-ce qu'une race condition et comment l'éviter ?

Q5. Pourquoi est-il important que les handlers d'interruption soient courts ?

Q6. Expliquez le rôle de l'instruction RETI (ou équivalent).

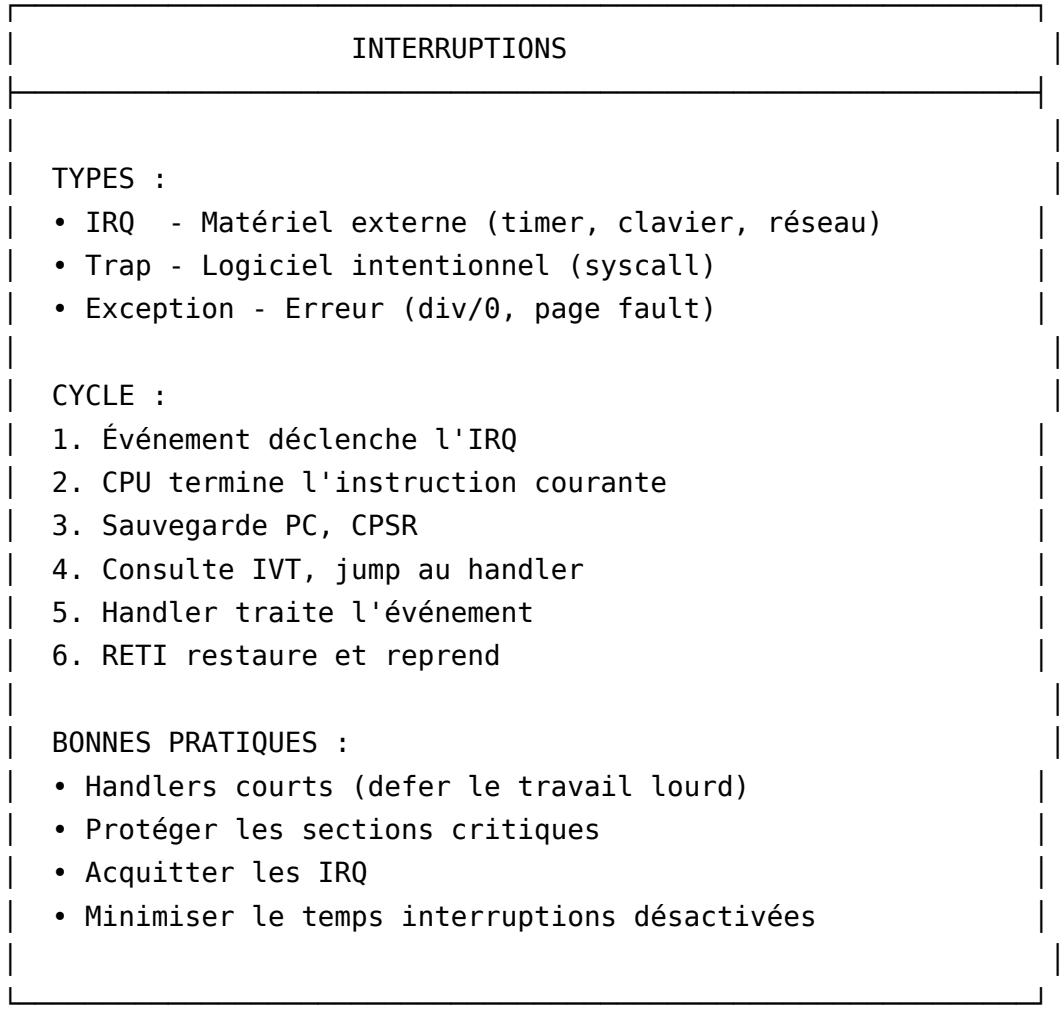
Q7. Qu'est-ce que le FIQ et pourquoi est-il plus rapide que l'IRQ ?

15.10.2 Exercice de Réflexion

Un système a les caractéristiques suivantes : - Timer IRQ toutes les 1 ms - Chaque IRQ timer prend 50 µs à traiter - Un thread doit répondre à des événements réseau en moins de 100 µs

Ce système peut-il garantir la latence réseau demandée ? Justifiez.

15.11 Résumé



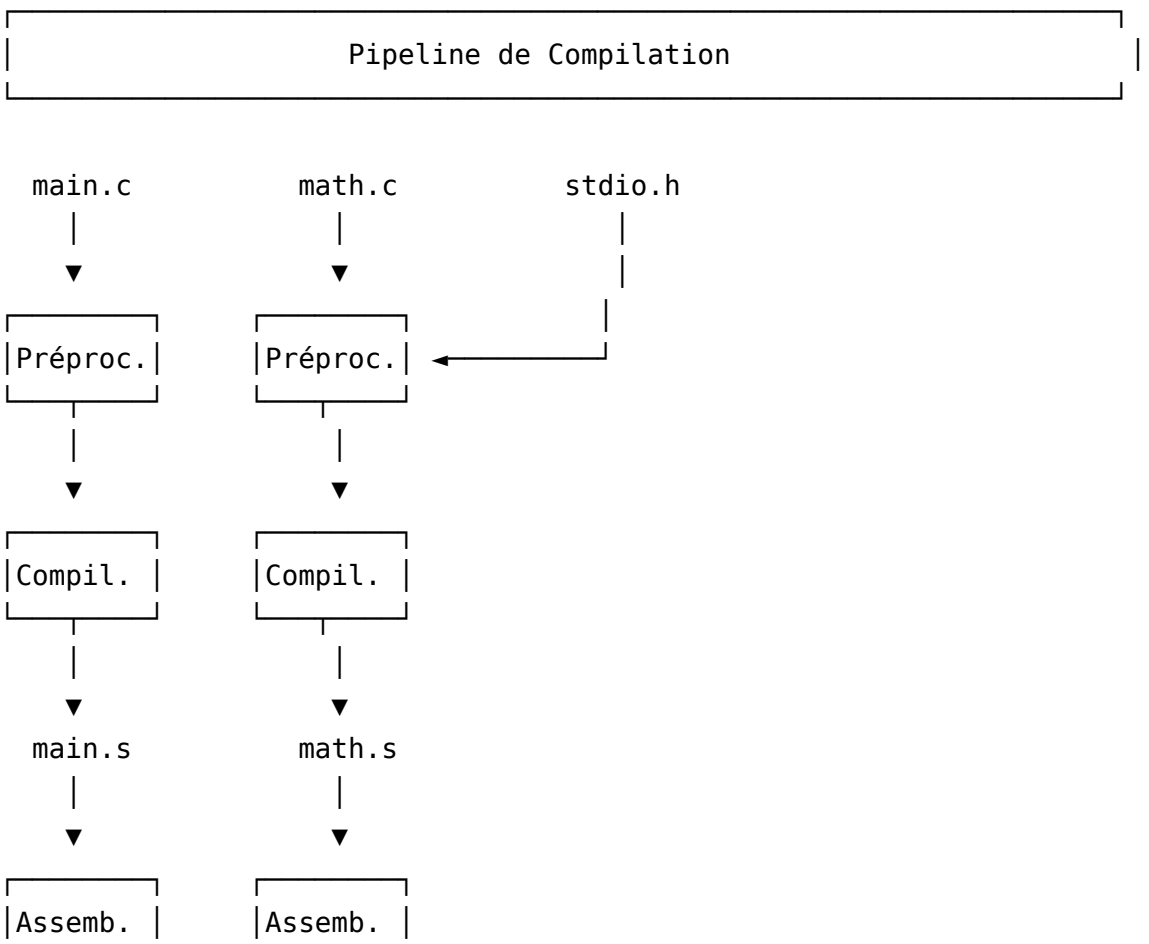
Les interruptions sont fondamentales pour tout système réactif. Elles permettent au matériel de communiquer efficacement avec le logiciel, et sont la base de concepts avancés comme le multitâche préemptif et les systèmes temps réel.

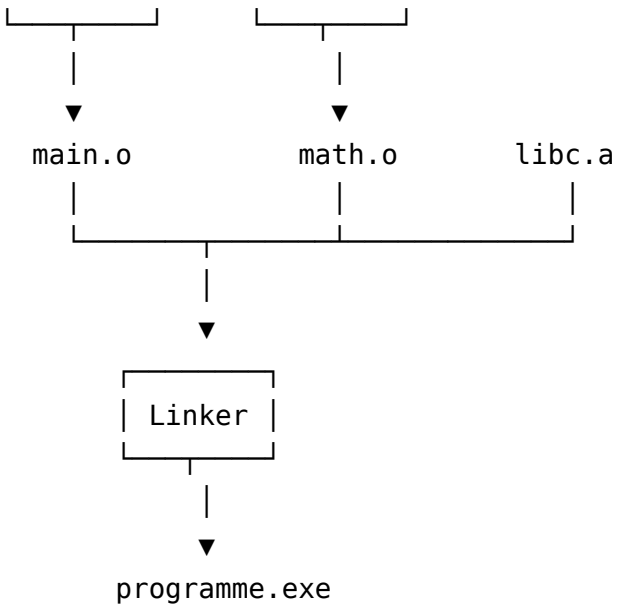
16 Concepts Avancés : Du Code Source à l'Exécution

Ce chapitre explore des concepts essentiels qui relient le code que vous écrivez à son exécution réelle : comment les programmes sont assemblés et liés, comment gérer les erreurs proprement, et comment faire tourner plusieurs tâches "en parallèle".

16.1 Partie 1 : Compilation et Linking

16.1.1 Le Voyage du Code Source





16.1.2 Le Préprocesseur

Le préprocesseur traite le code **avant** la compilation :

```c
// Inclusion de fichiers
#include <stdio.h>      // Fichiers système : /usr/include/
#include "myheader.h"   // Fichiers locaux : ./

// Macros de substitution
#define MAX_SIZE 100
#define SQUARE(x) ((x) * (x))

int arr[MAX_SIZE];          // → int arr[100];
int y = SQUARE(5);          // → int y = ((5) * (5));
int z = SQUARE(a + b);      // → int z = ((a + b) * (a + b));

// ATTENTION aux pièges des macros !
#define BAD_SQUARE(x) x * x
int bad = BAD_SQUARE(a + b);  // → int bad = a + b * a + b; FAUX !

// Compilation conditionnelle
#ifdef DEBUG
    printf("Variable x = %d\n", x);
#endif

#if PLATFORM == WINDOWS
    #include "win32.h"
#elif PLATFORM == LINUX
    #include "linux.h"
#else
    #error "Platform non supportée"
```

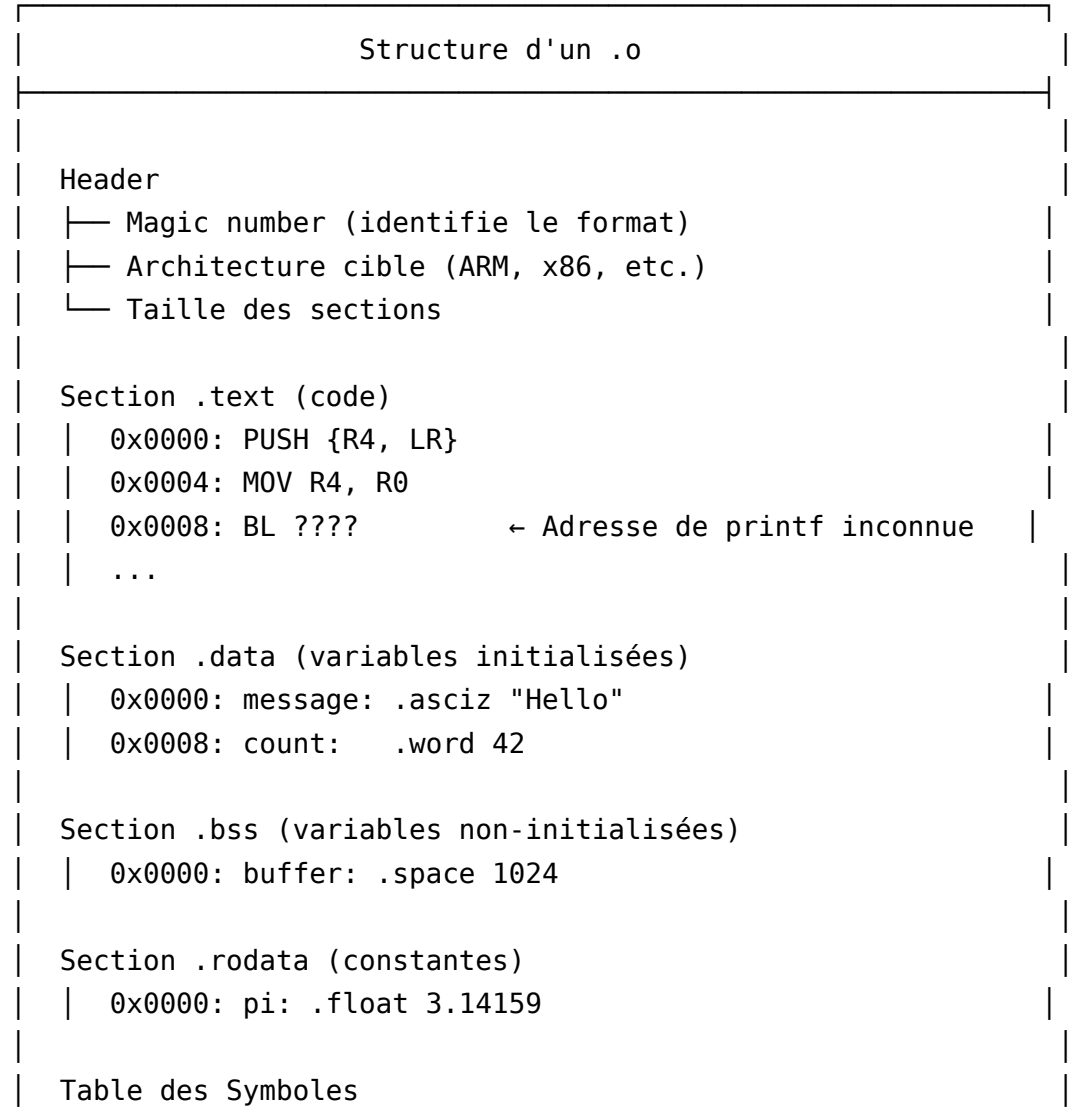
```
#endif

// Protection contre les inclusions multiples
#ifndef MYHEADER_H
#define MYHEADER_H
// ... contenu du header ...
#endif
```

Voir le résultat du préprocesseur :

```
gcc -E main.c -o main.i    # Sortie préprocessée
```

16.1.3 Les Fichiers Objets (.o)

Un fichier objet contient :



```
│  │  main        : .text + 0x0000  (GLOBAL, DEFINED)         │
│  │  helper      : .text + 0x0040  (LOCAL, DEFINED)          │
│  │  printf      : ???             (GLOBAL, UNDEFINED)  ←    │
│  │  malloc      : ???             (GLOBAL, UNDEFINED)  ←    │
│                                                             │
│  Table de Relocation                                        │
│  │  .text+0x0008 : référence à 'printf'                     │
│  │  .text+0x0020 : référence à 'message' (.data)            │
│  │  ...                                                     │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

16.1.4 Le Linker (Éditeur de Liens)

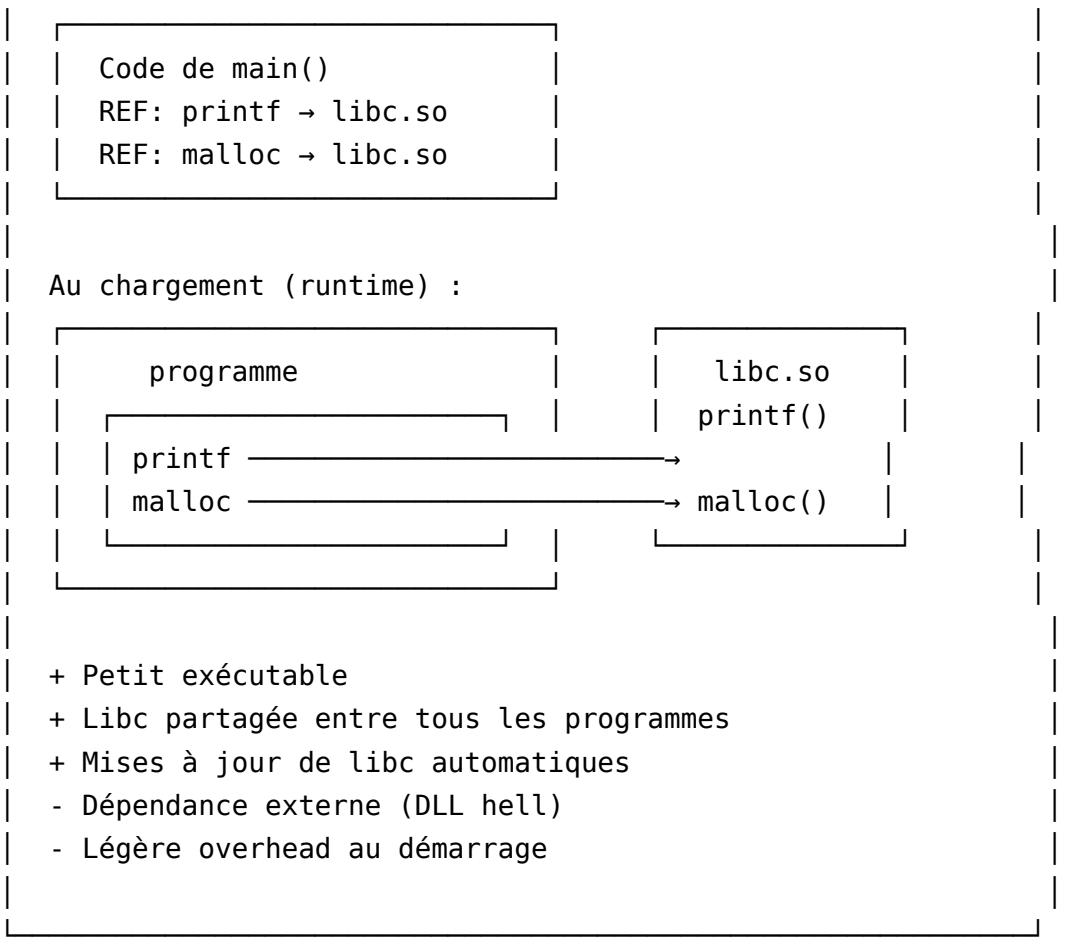
Le linker combine les fichiers objets :

```
┌─────────────────────────────────────────────────────────────┐
│                  Travail du Linker                          │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  1. COLLECTE : Rassembler toutes les sections               │
│                                                             │
│     main.o            math.o            libc.a            │
│     ┌─────────┐       ┌─────────┐       ┌─────────┐         │
│     │ .text   │       │ .text   │       │ printf  │         │
│     │ .data   │   +   │ .data   │   +   │ malloc  │         │
│     │ .bss    │       │ .bss    │       │ ...     │         │
│     └─────────┘       └─────────┘       └─────────┘         │
│                                                             │
│  2. FUSION : Combiner les sections de même type             │
│                                                             │
│     Exécutable final :                                      │
│     ┌──────────────────────────────────┐                    │
│     │ .text : main.o.text              │ 0x00001000         │
│     │         math.o.text              │ 0x00001200         │
│     │         libc.text (printf, etc.) │ 0x00001400         │
│     ├──────────────────────────────────┤                    │
│     │ .data : main.o.data              │ 0x00002000         │
│     │         math.o.data              │ 0x00002100         │
│     ├──────────────────────────────────┤                    │
│     │ .bss  : (alloué mais pas stocké) │ 0x00003000         │
│     └──────────────────────────────────┘                    │
│                                                             │
```

```
│  3. RELOCATION : Résoudre les adresses                          │
│                                                                 │
│     Avant : BL ????        (printf non résolu)                  │
│     Après : BL 0x00001480  (adresse finale de printf)           │
│                                                                 │
│  4. RÉSOLUTION : Vérifier que tout est défini                   │
│                                                                 │
│     Si un symbole reste UNDEFINED → Erreur de link !            │
│     "undefined reference to 'foo'"                              │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

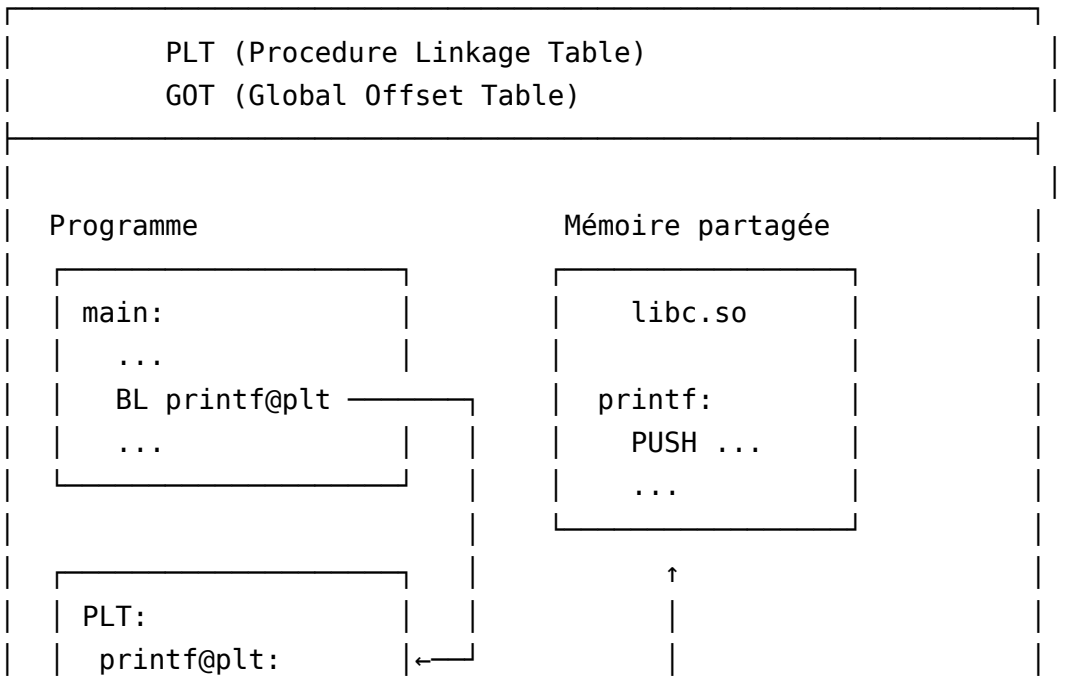
16.1.5 Linking Statique vs Dynamique

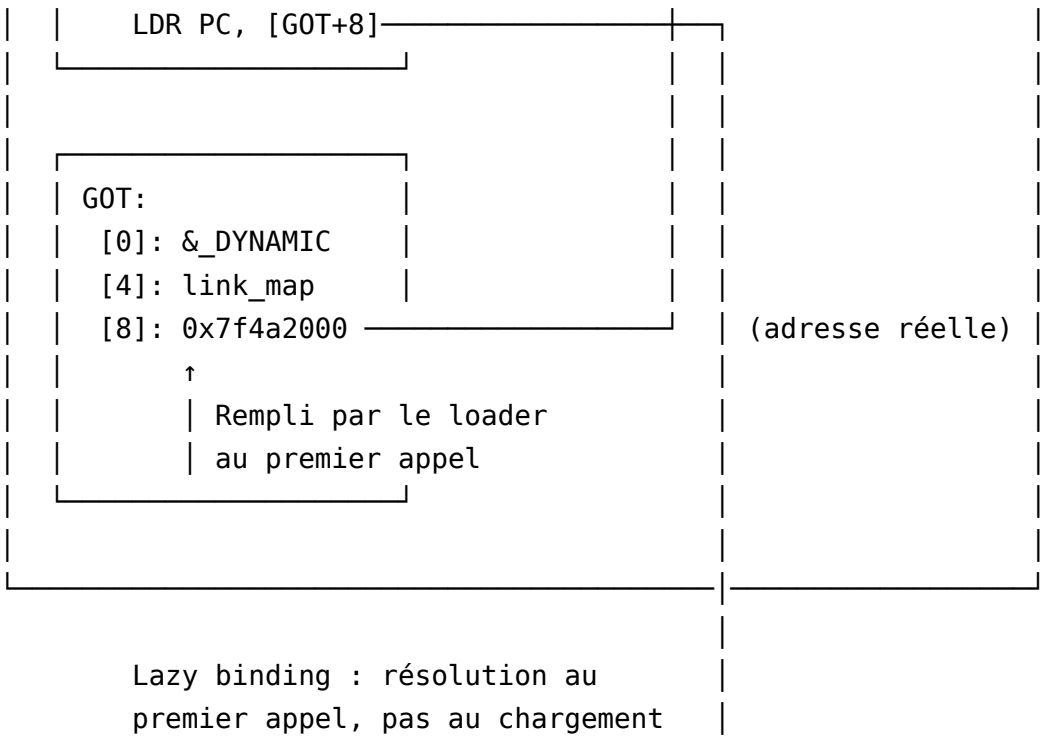
```
┌─────────────────────────────────────────────────────────────────┐
│              LINKING STATIQUE                                   │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  gcc main.o -static -o programme                                │
│                                                                 │
│  Exécutable contient TOUT :                                     │
│  ┌──────────────────────────────┐                               │
│  │  Code de main()              │                               │
│  │  Code de printf()    ← copié de libc.a                       │
│  │  Code de malloc()    ← copié de libc.a                       │
│  │  ...toute la libc... ← GROS fichier !                        │
│  └──────────────────────────────┘                               │
│                                                                 │
│  + Autonome, pas de dépendances                                 │
│  + Légèrement plus rapide (pas de résolution runtime)           │
│  - Gros exécutable (plusieurs MB)                               │
│  - Pas de mise à jour de libc sans recompiler                   │
│  - Duplication si plusieurs programmes utilisent libc           │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────────────────┐
│              LINKING DYNAMIQUE                                  │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  gcc main.o -o programme                                        │
│                                                                 │
│  Exécutable minimal + références :                              │
```



16.1.6 PLT et GOT : Comment ça Marche

Pour le linking dynamique, le loader utilise deux tables :





16.2 Partie 2 : Gestion des Exceptions

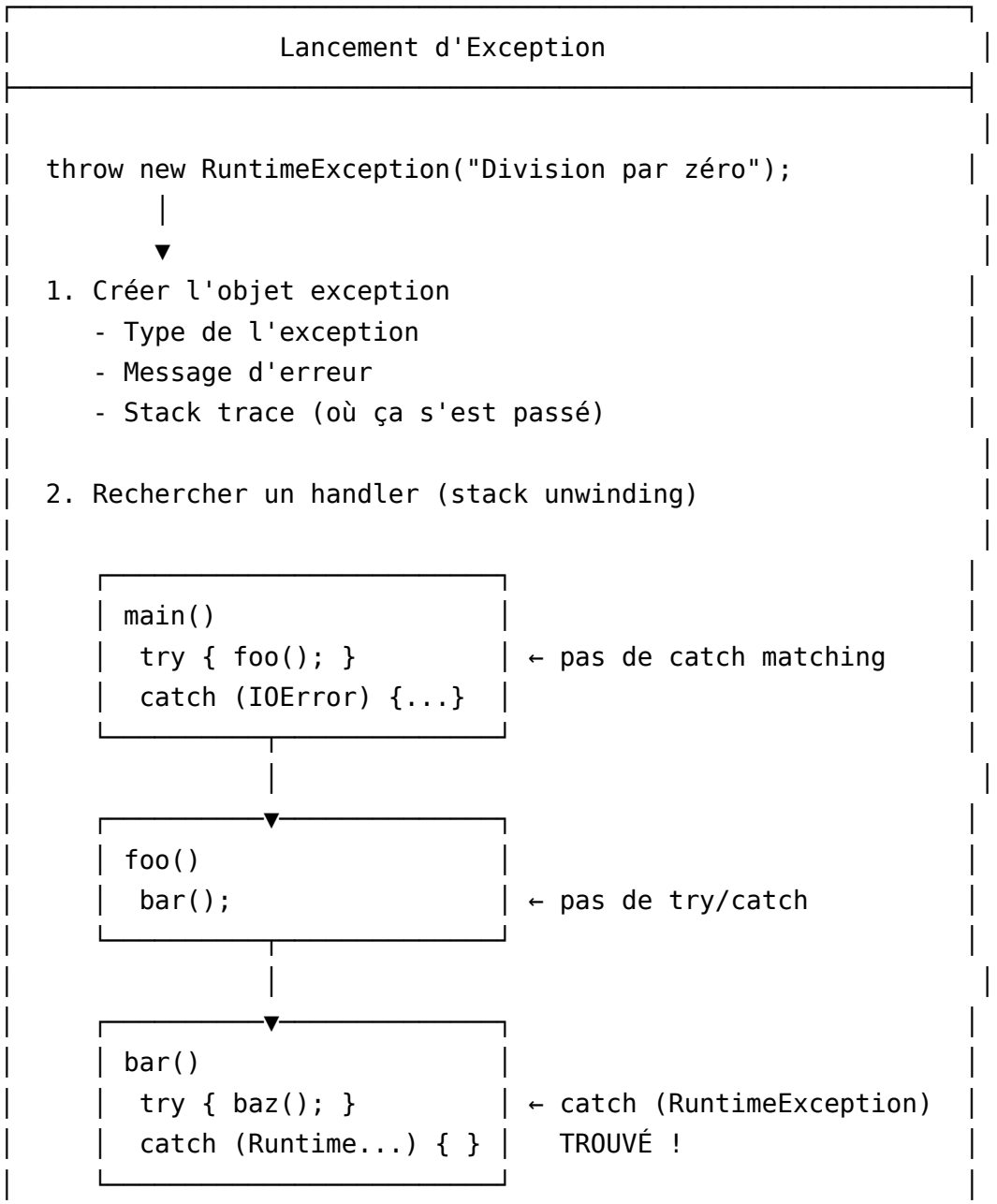
16.2.1 Qu'est-ce qu'une Exception ?

Une **exception** signale qu'une situation anormale s'est produite. Contrairement au retour d'erreur classique, les exceptions **déroulent la pile** jusqu'à trouver un gestionnaire approprié.

```
// SANS exceptions (style C)
int result;
FILE* f = fopen("data.txt", "r");
if (f == NULL) {
    return ERROR_FILE_NOT_FOUND;  // Propagation manuelle
}
result = parse_data(f);
if (result < 0) {
    fclose(f);
    return ERROR_PARSE_FAILED;   // Oubli facile du cleanup !
}
// ...

// AVEC exceptions (style C++/Java/Python)
try {
    File f = open("data.txt");     // Peut lever FileNotFound
```

```
    Data d = parse(f);              // Peut lever ParseError
    process(d);                     // Peut lever ProcessError
} catch (FileNotFound e) {
    log("Fichier non trouvé: " + e.path);
} catch (ParseError e) {
    log("Erreur de parsing ligne " + e.line);
} finally {
    // Toujours exécuté, même si exception
    cleanup();
}
```

16.2.2 Anatomie d'une Exception



```
│                 │                                                  │
│                 ▼                                                  │
│  3. Dérouler la pile jusqu'au handler                              │
│     - Appeler les destructeurs                                     │
│     - Libérer les ressources                                       │
│     - Exécuter les blocs finally                                   │
│                                                                    │
│  4. Exécuter le handler                                            │
│     - catch block reçoit l'objet exception                         │
│     - Peut re-throw si nécessaire                                  │
│                                                                    │
└────────────────────────────────────────────────────────────────────┘
```

16.2.3 Implémentation Bas Niveau (SJLJ)

La méthode **setjmp/longjmp** est une façon simple d'implémenter les exceptions en C :

```c
#include <setjmp.h>

jmp_buf exception_env;
int exception_code;

// Équivalent de "try"
if (setjmp(exception_env) == 0) {
    // Code normal
    risky_operation();
} else {
    // Handler (équivalent de "catch")
    printf("Exception %d attrapée!\n", exception_code);
}

void risky_operation() {
    if (error_condition) {
        exception_code = 42;
        longjmp(exception_env, 1);  // "throw"
    }
}
```

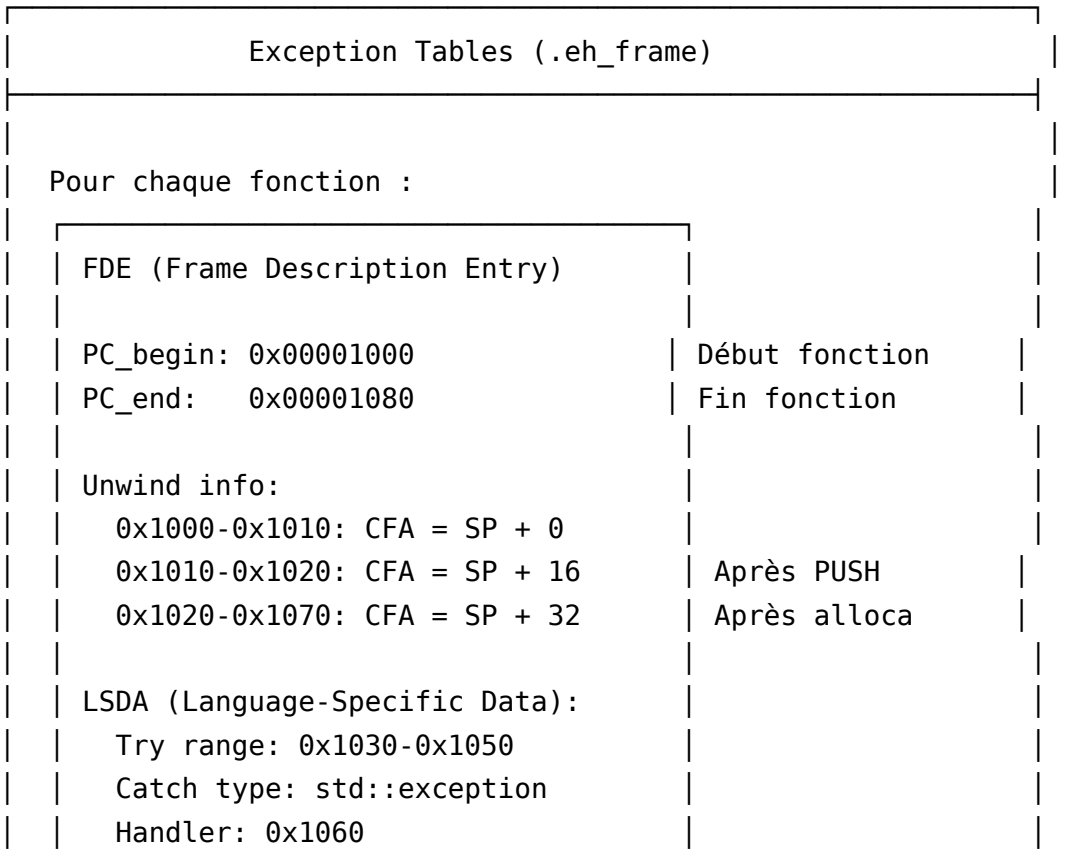
Comment ça marche au niveau assembleur :

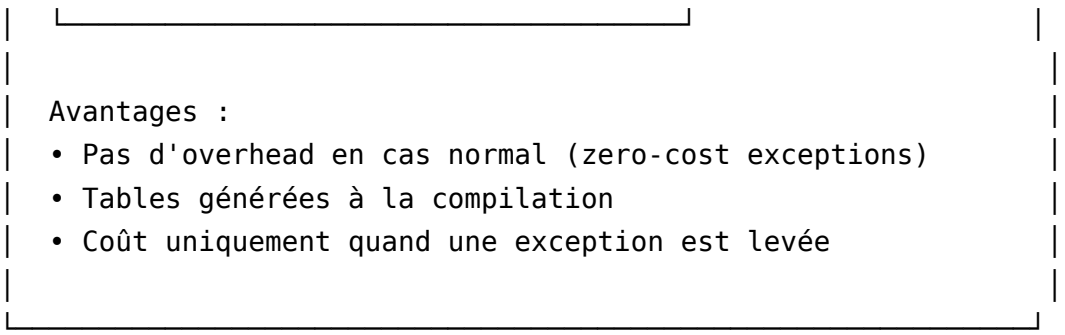
```
; setjmp : sauvegarde le contexte
setjmp:
    ; Sauver tous les registres callee-saved
    STR SP, [R0, #0]      ; Stack pointer
```

```
    STR LR, [R0, #4]      ; Return address
    STR R4, [R0, #8]
    STR R5, [R0, #12]
    ; ... R6-R11 ...
    MOV R0, #0            ; Retourne 0 (première fois)
    BX LR

; longjmp : restaure le contexte
longjmp:
    ; R0 = jmp_buf, R1 = valeur de retour
    LDR SP, [R0, #0]      ; Restaurer stack pointer
    LDR LR, [R0, #4]      ; Restaurer return address
    LDR R4, [R0, #8]
    LDR R5, [R0, #12]
    ; ... R6-R11 ...
    MOV R0, R1            ; Retourne la valeur passée
    CMP R0, #0
    MOVEQ R0, #1          ; Ne jamais retourner 0
    BX LR                 ; "Retourne" à setjmp !
```

16.2.4 Table-Driven Exception Handling (Moderne)

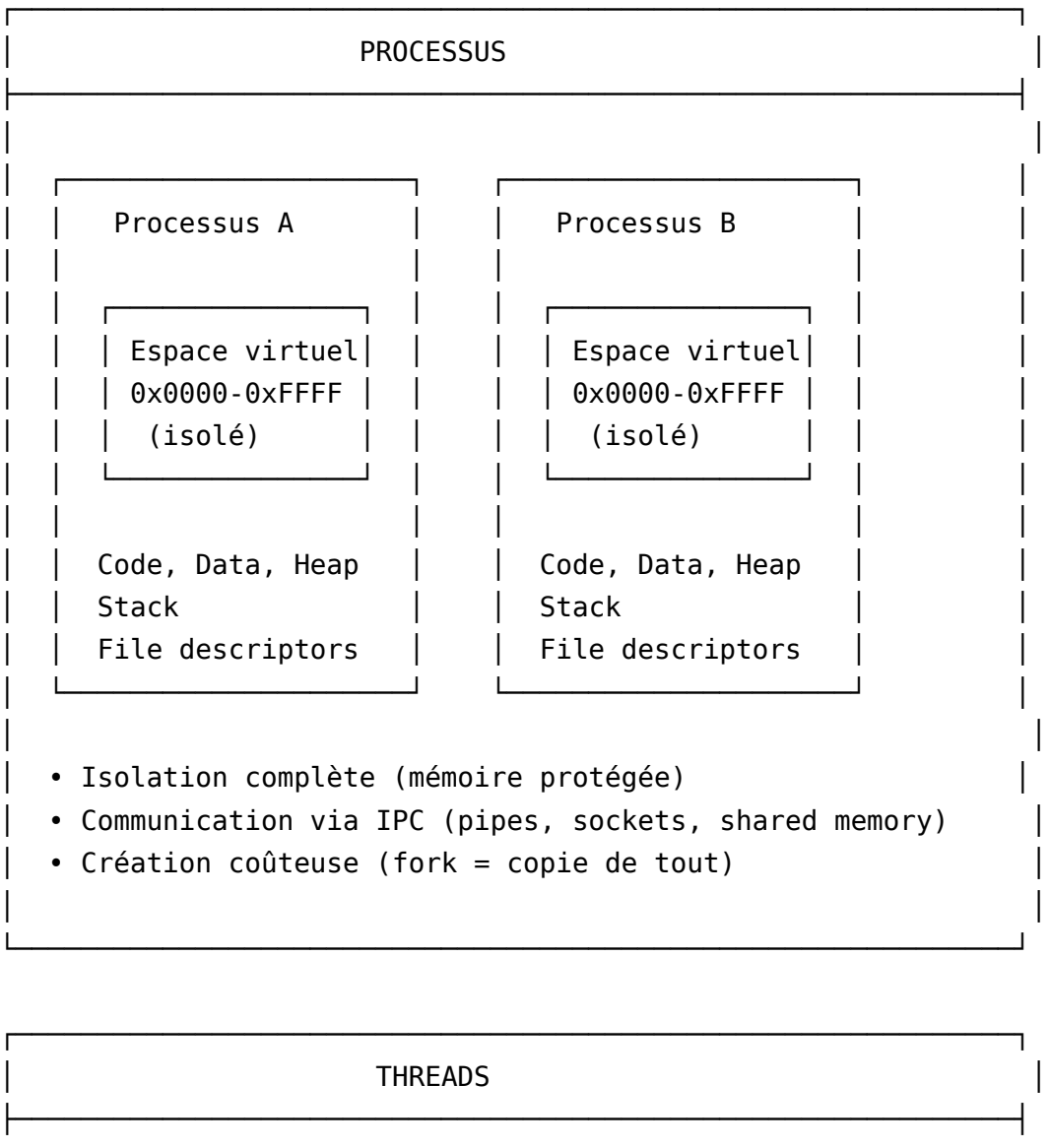
Les compilateurs modernes utilisent des tables plutôt que du code :

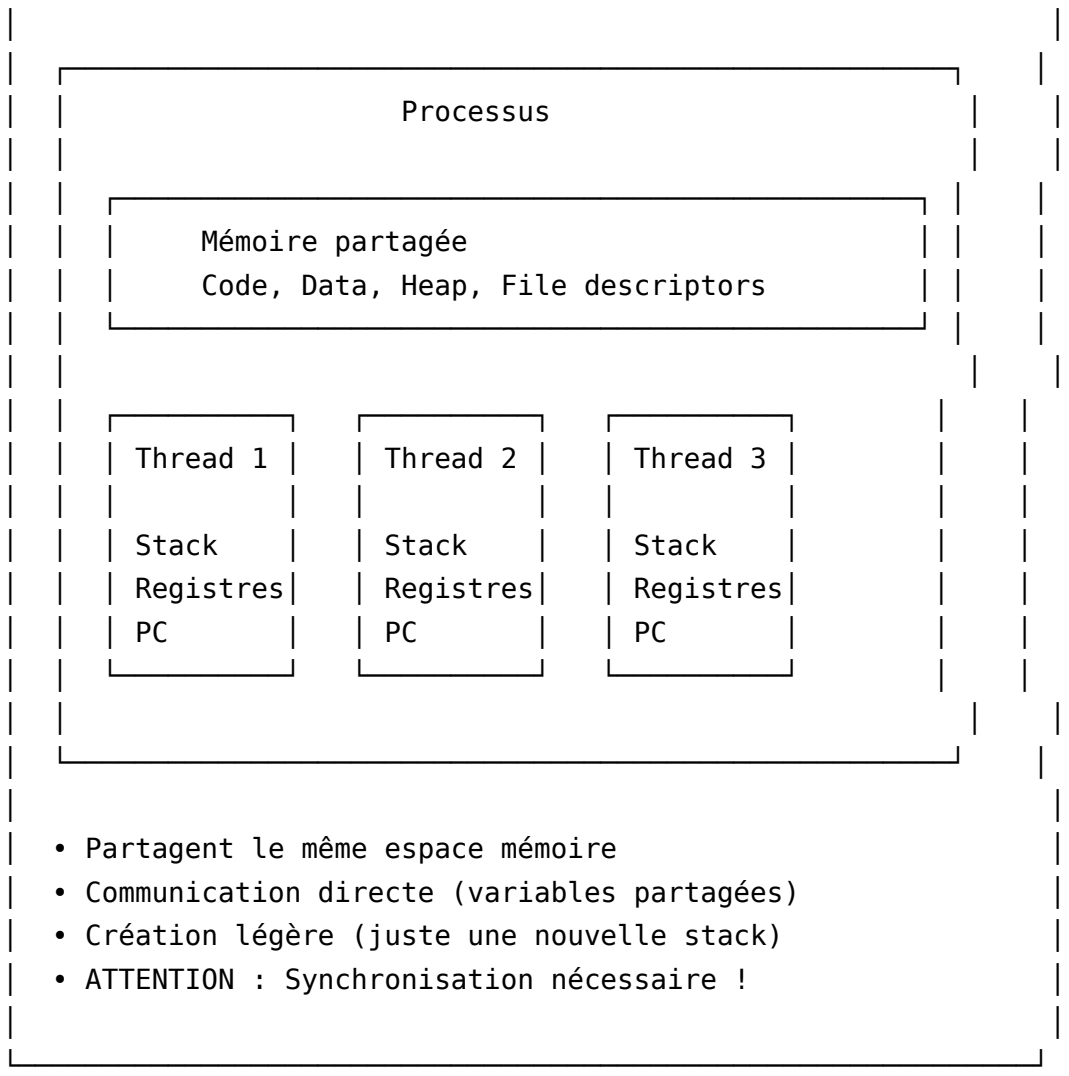




16.3 Partie 3 : Threads et Concurrence

16.3.1 Processus vs Threads





16.3.2 Création de Threads

```c
// POSIX Threads (pthreads)
#include <pthread.h>

void* thread_function(void* arg) {
    int id = *(int*)arg;
    printf("Thread %d démarre\n", id);
    // Travail...
    return NULL;
}

int main() {
    pthread_t threads[4];
    int ids[4] = {0, 1, 2, 3};

    // Créer 4 threads
    for (int i = 0; i < 4; i++) {
```

```
        pthread_create(&threads[i], NULL, thread_function, &ids[i]);
    }

    // Attendre qu'ils terminent
    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

16.3.3 Le Problème : Race Conditions

```
// DANGER : Accès concurrent non protégé
int counter = 0;

void* increment(void* arg) {
    for (int i = 0; i < 1000000; i++) {
        counter++;  // NON ATOMIQUE !
    }
    return NULL;
}

// Avec 2 threads : résultat attendu = 2,000,000
// Résultat réel : souvent ~1,500,000 !

// Pourquoi ? counter++ se décompose en :
// 1. LOAD  R0, [counter]   ← Thread A lit 42
// 2. ADD   R0, R0, #1      ← Thread A calcule 43
//    --- CONTEXT SWITCH ---
// 1. LOAD  R0, [counter]   ← Thread B lit AUSSI 42
// 2. ADD   R0, R0, #1      ← Thread B calcule 43
// 3. STORE R0, [counter]   ← Thread B écrit 43
//    --- CONTEXT SWITCH ---
// 3. STORE R0, [counter]   ← Thread A écrit 43
// Résultat : counter = 43 au lieu de 44 !
```

16.3.4 Synchronisation : Mutex

```
#include <pthread.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;
```

```
void* safe_increment(void* arg) {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);    // Acquérir le verrou
        counter++;                     // Section critique
        pthread_mutex_unlock(&lock);   // Libérer le verrou
    }
    return NULL;
}
```

Implémentation bas niveau d'un mutex (spinlock) :

```
; Test-and-Set atomique
; R0 = adresse du lock, R1 = valeur à écrire
acquire_lock:
    MOV R1, #1
spin:
    LDREX R2, [R0]        ; Load Exclusive
    CMP R2, #0            ; Lock libre ?
    BNE spin              ; Non, réessayer
    STREX R3, R1, [R0]    ; Store Exclusive
    CMP R3, #0            ; Succès ?
    BNE spin              ; Non, quelqu'un d'autre a gagné
    DMB                   ; Memory barrier
    BX LR

release_lock:
    DMB                   ; Memory barrier
    MOV R1, #0
    STR R1, [R0]          ; Libérer le lock
    BX LR
```

16.3.5 Autres Primitives de Synchronisation

```
// SÉMAPHORE : Compteur de ressources
sem_t slots;
sem_init(&slots, 0, 5);      // 5 slots disponibles

sem_wait(&slots);            // Décrémenter (bloque si 0)
use_resource();
sem_post(&slots);            // Incrémenter

// CONDITION VARIABLE : Attendre un événement
pthread_mutex_t mutex;
pthread_cond_t cond;
```

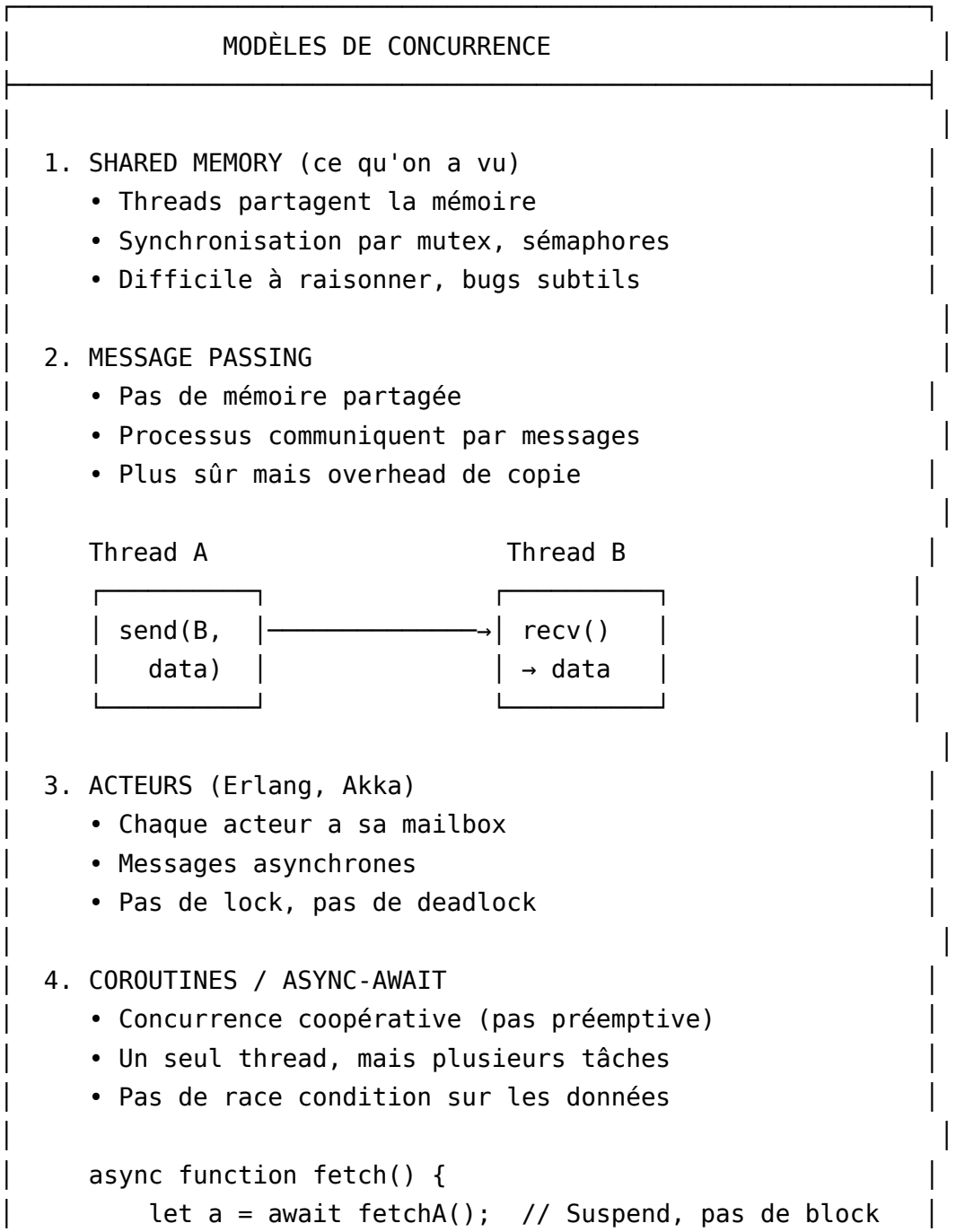
```
int data_ready = 0;

// Thread producteur
pthread_mutex_lock(&mutex);
produce_data();
data_ready = 1;
pthread_cond_signal(&cond);  // Réveiller un consommateur
pthread_mutex_unlock(&mutex);

// Thread consommateur
pthread_mutex_lock(&mutex);
while (!data_ready) {        // Boucle pour éviter spurious wakeups
    pthread_cond_wait(&cond, &mutex);  // Attend + libère/réacquiert mutex
}
consume_data();
pthread_mutex_unlock(&mutex);

// READ-WRITE LOCK : Plusieurs lecteurs OU un écrivain
pthread_rwlock_t rwlock;

// Lecteurs (peuvent être multiples)
pthread_rwlock_rdlock(&rwlock);
read_data();
pthread_rwlock_unlock(&rwlock);

// Écrivain (exclusif)
pthread_rwlock_wrlock(&rwlock);
write_data();
pthread_rwlock_unlock(&rwlock);
```

16.3.6 Deadlocks

```
// DEADLOCK : Deux threads qui s'attendent mutuellement

pthread_mutex_t lockA, lockB;

// Thread 1                    // Thread 2
lock(&lockA);                  lock(&lockB);
lock(&lockB);  // BLOQUÉ !     lock(&lockA);  // BLOQUÉ !
//             ↑                              ↑
//             └──── Attend lockB ────────────┘
//                   qui attend lockA

// SOLUTION 1 : Ordre fixe d'acquisition
// Toujours acquérir lockA avant lockB
```

```
// SOLUTION 2 : trylock avec timeout
if (pthread_mutex_trylock(&lockB) != 0) {
    pthread_mutex_unlock(&lockA);  // Libérer et réessayer
    goto retry;
}

// SOLUTION 3 : Lock hierarchies
// Chaque lock a un niveau, on ne peut acquérir que des niveaux croissants
```

16.3.7 Modèles de Concurrence



```
│         let b = await fetchB();                                  │
│         return a + b;                                            │
│     }                                                            │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

16.4 Exercices Pratiques

16.4.1 Exercice 1 : Analyse d'un Fichier Objet

Utilisez `objdump` ou `readelf` pour analyser un fichier `.o` :

```
# Compiler sans linker
gcc -c hello.c -o hello.o

# Examiner les sections
objdump -h hello.o

# Voir les symboles
nm hello.o

# Désassembler
objdump -d hello.o
```

Questions : 1. Quels symboles sont définis ? Lesquels sont indéfinis ? 2. Quelle taille fait la section `.text` ? 3. Où est stockée la chaîne "Hello World" ?

16.4.2 Exercice 2 : Linking Manuel

Créez deux fichiers :

```
// main.c
extern int add(int a, int b);
int main() {
    return add(3, 4);
}

// math.c
int add(int a, int b) {
    return a + b;
}
```

1. Compilez séparément : `gcc -c main.c` et `gcc -c math.c`
2. Examinez les symboles de chaque `.o`
3. Linkez manuellement : `ld main.o math.o -o programme`
4. Que se passe-t-il ? Pourquoi ? (Indice : `_start`, libc)

16.4.3 Exercice 3 : Exception Handling en C

Implémentez un système d'exceptions simple avec setjmp/longjmp :

```
// TODO: Implémenter TRY, CATCH, THROW
// Utilisation souhaitée :
TRY {
    if (error) THROW(42);
    do_work();
} CATCH(code) {
    printf("Exception %d\n", code);
} END_TRY;
```

16.4.4 Exercice 4 : Producteur-Consommateur

Implémentez le problème classique producteur-consommateur :

```
#define BUFFER_SIZE 10
int buffer[BUFFER_SIZE];
int count = 0;

// TODO: Implémenter avec mutex et condition variables
void* producer(void* arg);   // Produit des items
void* consumer(void* arg);   // Consomme des items
```

16.4.5 Exercice 5 : Détection de Deadlock

Analysez ce code et identifiez le deadlock potentiel :

```
pthread_mutex_t m1, m2, m3;

void* thread_a(void* arg) {
    lock(&m1); lock(&m2);
    work();
    unlock(&m2); unlock(&m1);
}

void* thread_b(void* arg) {
    lock(&m2); lock(&m3);
```

```
    work();
    unlock(&m3); unlock(&m2);
}

void* thread_c(void* arg) {
    lock(&m3); lock(&m1);
    work();
    unlock(&m1); unlock(&m3);
}
```

1. Y a-t-il un deadlock possible ?
2. Dessinez le graphe de dépendance
3. Proposez une solution

16.5 Auto-évaluation

16.5.1 Quiz

Q1. Quelle est la différence entre linking statique et dynamique ?

Q2. Qu'est-ce que la table des symboles dans un fichier objet ?

Q3. Que fait le préprocesseur avec `#include` et `#define` ?

Q4. Expliquez le concept de "stack unwinding" lors d'une exception.

Q5. Quelle est la différence entre un processus et un thread ?

Q6. Qu'est-ce qu'une race condition et comment l'éviter ?

Q7. Qu'est-ce qu'un deadlock ? Donnez les quatre conditions nécessaires.

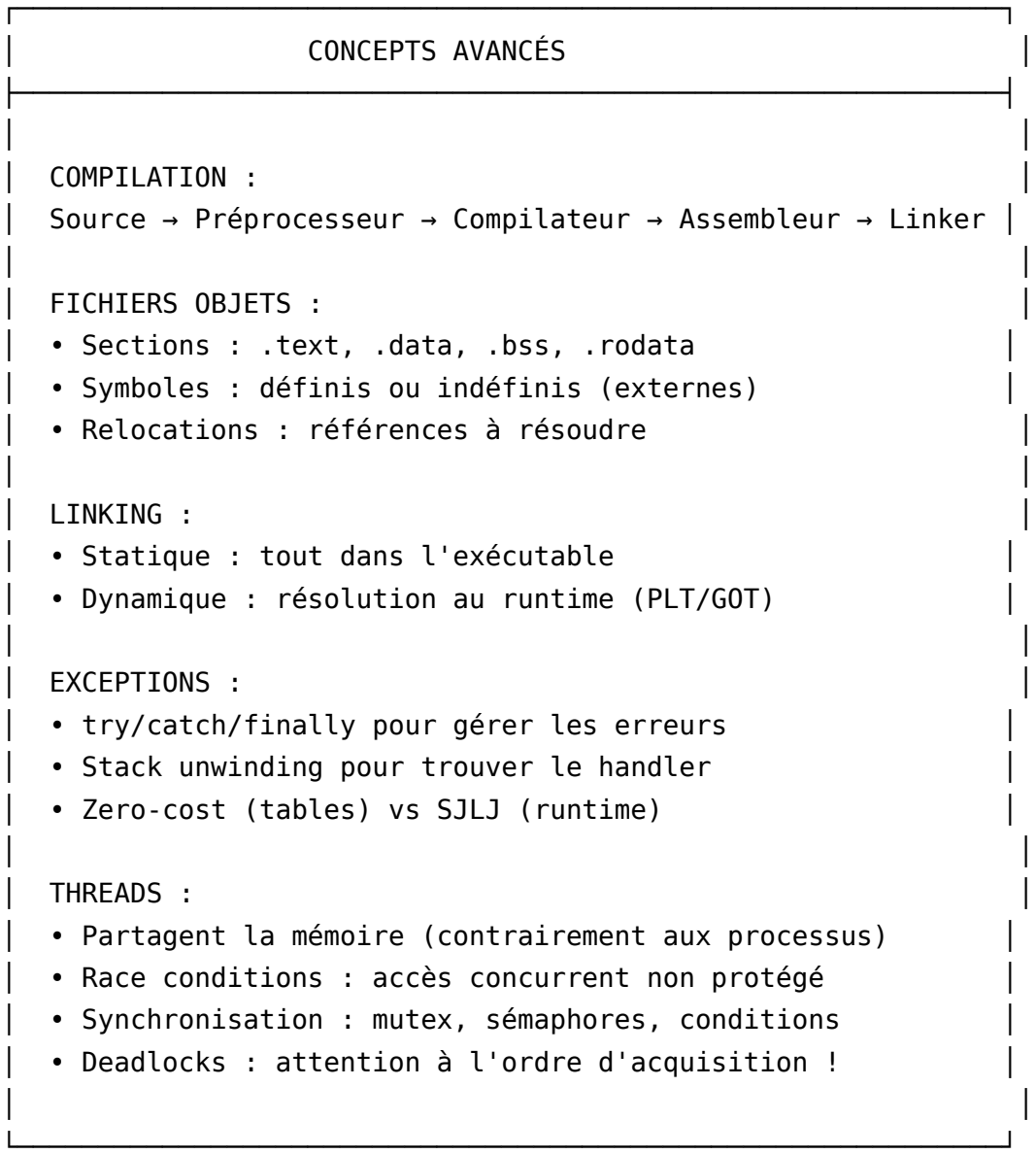
Q8. Pourquoi les opérations atomiques sont-elles nécessaires pour les mutex ?

16.5.2 Défi

Implémentez un pool de threads simple :

```
typedef struct {
    // TODO: votre structure
} ThreadPool;

ThreadPool* pool_create(int num_threads);
void pool_submit(ThreadPool* pool, void (*task)(void*), void* arg);
void pool_destroy(ThreadPool* pool);
```

16.6 Résumé



Ces concepts sont fondamentaux pour comprendre comment les programmes réels fonctionnent, du code source jusqu'à l'exécution multi-threadée. La maîtrise de la compilation/linking aide au débogage, tandis que la compréhension de la concurrence est essentielle pour écrire des programmes corrects et performants.