

# L'Architecture Codex

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

Projet Nand2Tetris-Codex

2026

## Table des matières

<b>1</b>	<b>L'Architecture Codex : Guide de l'Étudiant</b>	<b>3</b>
1.1	Table des Matières . . . . .	3
1.2	Comment utiliser ce guide . . . . .	4
<b>2</b>	<b>Introduction</b>	<b>4</b>
2.1	Pourquoi construire un ordinateur ? . . . . .	4
2.2	Codex vs Hack (Nand2Tetris Original) . . . . .	4
2.3	Vue d'ensemble du système . . . . .	5
2.4	Vos Outils . . . . .	5
<b>3</b>	<b>Chapitre 1 : Logique Booléenne</b>	<b>6</b>
3.1	1.1 L'Abstraction Binaire . . . . .	6
3.2	1.2 La Porte NAND (Not-And) . . . . .	6
3.3	1.3 Portes Élémentaires à construire . . . . .	6
3.4	1.4 Description Matérielle (Codex HDL) . . . . .	8
3.5	1.5 Vos Exercices . . . . .	9
<b>4</b>	<b>Chapitre 2 : Arithmétique Binaire</b>	<b>10</b>
4.1	2.1 Représentation des Nombres . . . . .	10
4.2	2.2 L'Additionneur . . . . .	10
4.3	2.3 L'ALU32 (Arithmetic Logic Unit) . . . . .	11
4.4	2.4 Vos Exercices . . . . .	12
<b>5</b>	<b>Chapitre 3 : Logique Séquentielle et Mémoire</b>	<b>12</b>
5.1	3.1 Le Temps et l'Horloge (Clock) . . . . .	12
5.2	3.2 La Brique de Base : La Bascule D (DFF) . . . . .	13
5.3	3.3 Construire une Mémoire Persistante . . . . .	13
5.4	3.4 Des Bits aux Registres . . . . .	13
5.5	3.5 Le Compteur de Programme (PC) . . . . .	14
5.6	3.6 Vos Exercices . . . . .	14
<b>6</b>	<b>Chapitre 4 : Architecture Machine (ISA A32)</b>	<b>15</b>
6.1	4.1 Concepts Fondamentaux . . . . .	15
6.2	4.2 La Mémoire (Memory Map) . . . . .	15

6.3	4.3 Les Registres (Le plan de travail) . . . . .	16
6.4	4.4 Le Format des Instructions (32 bits) . . . . .	16
6.5	4.5 Les Classes d'Instructions . . . . .	17
6.6	4.6 Le Langage Assembleur (ASM) par l'exemple . . . . .	18
6.7	4.7 Gestion des Erreurs (Traps) . . . . .	18
6.8	4.8 Vos Exercices . . . . .	19
<b>7</b>	<b>Chapitre 5 : Le Processeur (CPU)</b>	<b>19</b>
7.1	5.1 Architecture Globale (Data Path) . . . . .	19
7.2	5.2 Les Composants Clés . . . . .	20
7.3	5.3 Le Cycle d'Exécution en Détail . . . . .	20
7.4	5.4 Vos Exercices . . . . .	21
7.5	5.5 Conseils de débogage . . . . .	21
<b>8</b>	<b>Chapitre 6 : L'Assembleur</b>	<b>22</b>
8.1	6.1 Le Rôle de l'Assembleur . . . . .	22
8.2	6.2 La Stratégie des Deux Passes . . . . .	22
8.3	6.3 Sections et Directives . . . . .	22
8.4	6.4 Exemple d'Encodage . . . . .	23
8.5	6.5 La Gestion des Grandes Constantes ( <code>=value</code> ) . . . . .	23
8.6	6.6 Vos Exercices . . . . .	23
<b>9</b>	<b>Chapitre 7 : Construction du Compilateur</b>	<b>24</b>
9.1	7.1 Le Défi "MicroC" . . . . .	24
9.2	7.2 Le Code Source de MicroC . . . . .	24
9.3	7.3 L'Expérience de "Self-Compilation" . . . . .	26
9.4	7.4 Comprendre la Génération de Code A32 . . . . .	26
9.5	7.5 Vos Exercices . . . . .	27
9.6	8.2 Spécification du Langage C32 . . . . .	28
9.7	8.3 Pointeurs et Tableaux . . . . .	29
9.8	8.4 Accès au Matériel (MMIO) . . . . .	29
9.9	8.5 Organisation d'un programme C32 . . . . .	29
9.10	8.6 Limitations (Ce que le C32 ne fait pas encore) . . . . .	30
9.11	8.7 Vos Exercices . . . . .	30
<b>10</b>	<b>Chapitre 9 : Système d'Exploitation</b>	<b>30</b>
10.1	9.1 La Hiérarchie Logicielle . . . . .	30
10.2	9.2 Gestion de la Mémoire (Le Tas / Heap) . . . . .	31
10.3	9.3 Bibliothèque Graphique ( <code>screen.c</code> ) . . . . .	31
10.4	9.4 Entrées / Sorties et Système ( <code>io.c</code> , <code>sys.c</code> ) . . . . .	31
10.5	9.5 Interruptions et Multitâche (Concept) . . . . .	32
10.6	9.6 Le Grand Final : Votre Application . . . . .	32
10.7	Vos Exercices . . . . .	32

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

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

## 1.1 Table des Matières

### 1. Introduction

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

### 2. Chapitre 1 : Logique Booléenne

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

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

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

### 4. Chapitre 3 : Logique Séquentielle et Mémoire (À venir)

- Le temps et l'horloge.
- Flip-Flops, Registres et RAM.
- *Exercices* : `hdl_lib/04_seq` & `02_multibit`.

### 5. Chapitre 4 : Architecture Machine (ISA A32) (À venir)

- Le jeu d'instructions A32-Lite.
- Registres, adressage et structure.
- *Référence* : `SPECS.md`.

### 6. Chapitre 5 : Le Processeur (CPU) (À venir)

- Implémentation du chemin de données (Data Path).
- Logique de contrôle.
- *Exercices* : `hdl_lib/05_cpu`.

### 7. Chapitre 6 : L'Assembleur (À venir)

- Traduction symbolique vers binaire.
- Gestion des symboles et des labels.
- *Outils* : `a32_asm`.

### 8. Chapitre 7 : Construction du Compilateur (À venir)

- Analyse lexicale et syntaxique (Parsing).
- Génération de code pour pile.
- *Outils* : `c32_cli`.

### 9. Chapitre 8 : Langage de Haut Niveau (C32) (À venir)

- Syntaxe du langage C32.
- Structures de contrôle et types.

### 10. Chapitre 9 : Système d'Exploitation (À venir)

- Gestion de la mémoire (Heap).
- Entrées/Sorties (Clavier, Écran).
- *Exercices* : `os_lib`.

## 1.2 Comment utiliser ce guide

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

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

## 2 Introduction

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

### 2.1 Pourquoi construire un ordinateur ?

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).

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.

### 2.2 Codex vs Hack (Nand2Tetris Original)

Ce cours est fortement inspiré du légendaire “Nand2Tetris”. Cependant, l'ordinateur *Hack* original a été conçu pour une simplicité maximale, parfois au détriment du réalisme moderne.

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

Caractéristique	Hack (Original)	Codex (Ce projet)
<b>Architecture</b>	16-bits	<b>32-bits</b>
<b>Registres</b>	2 (A et D)	<b>16 (r0-r15)</b> (Inspiré par ARM)
<b>Mémoire</b>	Séparée (Harvard)	<b>Unifiée</b> (Von Neumann pour instructions/données)
<b>Instructions</b>	Simple, propriétaire	<b>RISC moderne</b> (Load/Store, Branch)
<b>Écran</b>	Monochrome	<b>Couleur</b> (Framebuffer)

**Pourquoi ce changement ?** Parce que comprendre une architecture à registres généraux (comme ARM ou RISC-V) est beaucoup plus pertinent pour un ingénieur moderne. Le défi est légèrement plus grand, mais la récompense est une compréhension directe des machines qui propulsent nos smartphones et serveurs actuels.

## 2.3 Vue d'ensemble du système

Voici les couches que nous allons traverser :

Applications (Jeux, Shell, Calculatrice)	Chapitre 10
Langage de Haut Niveau (C32)	Chapitre 8
Compilateur & Bibliothèques	Chapitre 7 & 9
Système d'Exploitation (Gestion Mémoire, Drivers E/S)	Chapitre 9
Assembleur (A32 ASM)	Chapitre 6
Architecture Machine (ISA) (Jeu d'instructions, Registres)	Chapitre 4 & 5
Logique Matérielle (Gates, ALU, RAM, CPU implementation)	Chapitre 1, 2, 3

## 2.4 Vos Outils

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

1. **hdl\_cli** : Le simulateur de matériel en ligne de commande. Il lit vos fichiers **.hdl** et teste vos circuits.
2. **a32\_cli** : L'assembleur. Il transforme votre code assembleur **.a32** en binaire.
3. **c32\_cli** : Le compilateur. Il transforme votre code C **.c** en assembleur.
4. **a32\_runner** : L'émulateur. Il exécute le code binaire final.

### 2.4.1 Le Simulateur Web (Web Simulator)

Pour une expérience plus visuelle et interactive, vous pouvez utiliser le **Simulateur Web**. Il vous permet de : \* Écrire et tester votre HDL directement dans le navigateur. \* Voir l'état des signaux en temps réel. \* Compiler et exécuter du code C et Assembleur avec une visualisation de l'écran et des registres.

Pour le lancer localement :

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

Ensuite, ouvrez votre navigateur à l'adresse indiquée (généralement **http://localhost:5173**).

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

### 3 Chapitre 1 : 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.

#### 3.1 1.1 L’Abstraction Binaire

Dans le monde physique, nos ordinateurs utilisent des tensions électriques. Par exemple, 0 Volt pour “0” et 3.3 Volts pour “1”. En tant qu’architectes système, nous ne nous soucions pas de la physique (les transistors). Nous nous soucions de la **logique**.

Une **Fonction Booléenne** prend une ou plusieurs entrées binaires et produit une sortie binaire.

#### 3.2 1.2 La Porte NAND (Not-And)

La technologie actuelle (CMOS) rend la construction de portes NAND très efficace. C’est notre axiome. Nous supposons qu’elle existe déjà.

**Table de Vérité NAND :**

A	B	NAND(A, B)
0	0	1
0	1	1
1	0	1
1	1	0

Remarquez : Le résultat est 0 *seulement si* A et B sont tous les deux à 1.

#### 3.3 1.3 Portes Élémentaires à construire

Votre mission est de construire les portes suivantes en utilisant uniquement des NAND (puis les portes que vous aurez créées).

##### 3.3.1 A. NOT (Inverseur)

L’inverseur change un 0 en 1 et un 1 en 0.

*Spécification :* If in=0 then out=1 else out=0

**Astuce d’implémentation :** Comment utiliser une porte NAND (qui prend 2 entrées) pour n’en traiter qu’une seule ? Si on connecte le même signal **in** aux deux entrées d’un NAND...

```

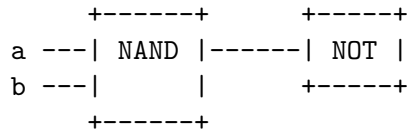
+-----+
in ---| a   |
      | NAND|
in ---| b   |
+-----+
```

### 3.3.2 B. AND (Et)

La sortie vaut 1 si et seulement si A et B valent 1.

*Spécification :* `If a=1 and b=1 then out=1 else out=0`

**Schéma Conceptuel :** Le NAND est un “Not-And”. Si on inverse le résultat d’un NAND... on obtient un AND !



### 3.3.3 C. OR (Ou)

La sortie vaut 1 si au moins une des entrées vaut 1.

*Table de Vérité :*

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

**Théorème de De Morgan :** `A OR B` est équivalent à `NOT( (NOT A) AND (NOT B) )`. Ou avec des NANDs : `(NOT A) NAND (NOT B)`.

### 3.3.4 D. XOR (Ou Exclusif)

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

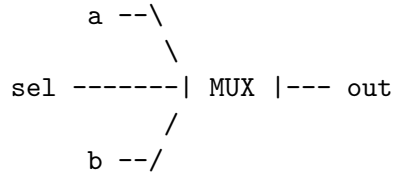
A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

C’est une porte cruciale pour l’addition binaire (comme nous le verrons au chapitre 2).

### 3.3.5 E. Multiplexeur (Mux)

Le Mux est l’aiguilleur du monde numérique. Il choisit entre deux entrées (`a` ou `b`) basé sur un signal de sélection (`sel`).

- Si `sel == 0` alors `out = a`
- Si `sel == 1` alors `out = b`



C'est fondamental pour le routage des données dans le CPU.

### 3.4 1.4 Description Matérielle (Codex HDL)

Pour décrire nos circuits, nous utilisons un langage inspiré du **VHDL**. C'est un langage déclaratif : on définit des composants (entités) et on décrit comment ils sont câblés (architectures).

#### 3.4.1 Structure d'un fichier .hdl

Un fichier HDL se compose de deux parties : 1. **L'entité ( entity )** : Elle définit l'interface (les broches d'entrée et de sortie). 2. **L'architecture ( architecture )** : Elle définit le contenu de la puce.

```
-- C'est un commentaire
entity NomDeLaPuce is
  port(
    a : in bit;      -- Entrée de 1 bit
    b : in bit;
    y : out bit      -- Sortie
  );
end entity;

architecture rtl of NomDeLaPuce is
  -- Déclaration des composants que l'on va utiliser
  component Nand
    port(a : in bit; b : in bit; y : out bit);
  end component;

  -- Signaux internes (fils)
  signal fil_interne : bit;
begin
  -- Instanciation et connexion (Port Map)
  u1: Nand port map (a => a, b => b, y => fil_interne);
  -- Assignment directe
  y <= not fil_interne;
end architecture;
```

#### 3.4.2 Règles de Connexion

1. **Instanciation** : On donne un nom d'instance (ex: `u1`) suivi du nom du composant.
2. **Port Map** :
  - À gauche du `=>` : Le nom de la broche du composant utilisé.
  - À droite du `=>` : Le signal de *votre* architecture que vous y connectez.
3. **Signaux** : Vous devez déclarer vos fils internes avec le mot-clé `signal` avant le `begin`.

4. **Assignations** : `y <= x`; signifie que le fil `y` est branché sur `x`.

### 3.4.3 Bus (Vecteurs de bits)

Pour manipuler plusieurs bits (ex: 32 bits), on utilise `bits(MSB downto LSB)`.

- `bus : in bits(31 downto 0)`; : Un bus de 32 bits.
- `bus(0)` : Le bit de poids faible.
- `bus(31)` : Le bit de poids fort.

```
-- Exemple de connexion d'un bit spécifique
u_gate: MyGate port map (a => mon_bus(5), ...);
```

### 3.4.4 Exemple Complet : Xor

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

architecture rtl of Xor is
  component Not port(a : in bit; y : out bit); end component;
  component And port(a : in bit; b : in bit; y : out bit); end component;
  component Or port(a : in bit; b : in bit; y : out bit); end component;

  signal nota, notb, w1, w2 : bit;
begin
  u_nota: Not port map (a => a, y => nota);
  u_notb: Not port map (a => b, y => notb);
  u_and1: And port map (a => a, b => notb, y => w1);
  u_and2: And port map (a => nota, b => b, y => w2);
  u_or: Or port map (a => w1, b => w2, y => y);
end architecture;
```

## 3.5 1.5 Vos Exercices

Rendez-vous dans le répertoire `hdl_lib/01_gates`. Vous y trouverez les fichiers squelettes (`.hdl`).

**Ordre conseillé** : 1. `Not.hdl`, `And.hdl`, `Or.hdl`, `Xor.hdl`, `Mux.hdl`, `DMux.hdl`.

### 3.5.1 Comment tester ?

**Option A : En ligne de commande (CLI)** Utilisez la commande suivante pour tester une porte spécifique :

```
cargo run -p hdl_cli -- test hdl_lib/01_gates/Not.hdl
```

**Option B : Simulateur Web (Recommandé pour débiter)** 1. Lancez le serveur web (`cd web && npm run dev`). 2. Ouvrez l'interface. 3. Allez dans la section **HDL Progression**. 4. Vous pouvez copier-coller votre code ou travailler directement dans l'éditeur intégré pour voir les chronogrammes et les tests passer en temps réel.

**Validation :** L'objectif est que tous les tests passent. Si une porte échoue, vérifiez votre table de vérité et assurez-vous de ne pas avoir créé de cycles (boucles) dans votre logique combinatoire).

## 4 Chapitre 2 : Arithmétique Binaire

Dans le chapitre précédent, nous avons appris à manipuler des bits individuels. Mais un ordinateur doit savoir compter et effectuer des opérations mathématiques sur des nombres. Comment passer d'une simple porte logique à une calculatrice capable de traiter des nombres sur 32 bits ?

### 4.1 2.1 Représentation des Nombres

#### 4.1.1 Le Binaire (Base 2)

Un nombre binaire est une suite de bits. Chaque position correspond à une puissance de 2. Pour un nombre de 4 bits :

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

Exemple : 1011 (binaire) =  $1*8 + 0*4 + 1*2 + 1*1 = 11$  (décimal)

Dans l'ordinateur Codex, nous travaillons sur **32 bits**, ce qui permet de représenter des nombres allant de 0 à  $2^{32} - 1$  (environ 4 milliards).

#### 4.1.2 Nombres Signés : Le Complément à 2

Comment représenter des nombres négatifs ? On utilise le système du **Complément à 2**. Le bit le plus à gauche (MSB, bit 31) est le "bit de signe". \* S'il vaut 0, le nombre est positif. \* S'il vaut 1, le nombre est négatif.

**La règle d'or du Complément à 2 :** Pour obtenir  $-X$  à partir de  $X$  : 1. Inverser tous les bits de  $X$ . 2. Ajouter 1 au résultat.

*Pourquoi ?* Parce que cela permet à l'additionneur de fonctionner de la même manière pour les additions et les soustractions sans circuit spécial pour le signe.

### 4.2 2.2 L'Additionneur

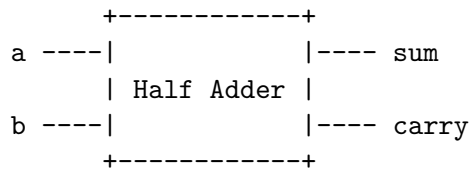
L'addition binaire suit les mêmes règles que l'addition décimale : on additionne colonne par colonne en gérant la retenue (*carry*).

#### 4.2.1 A. Le Demi-Additionneur (Half Adder)

Il additionne deux bits et produit une somme et une retenue.

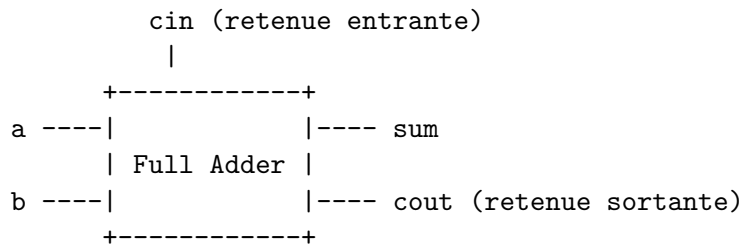
a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

**Observation :** \* **sum** correspond à la fonction **XOR(a, b)**. \* **carry** correspond à la fonction **AND(a, b)**.



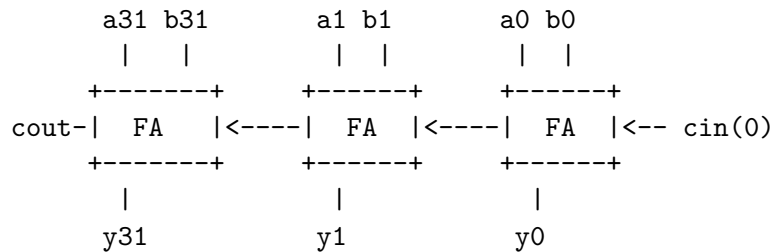
#### 4.2.2 B. L'Additionneur Complet (Full Adder)

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



#### 4.2.3 C. L'Additionneur 32-bits (Add32)

En cascade, nous connectons 32 Full Adders. La retenue de l'un va vers l'entrée de l'autre. C'est le "Ripple Carry Adder".



### 4.3 2.3 L'ALU32 (Arithmetic Logic Unit)

L'ALU est le cœur battant du processeur. C'est elle qui effectue les calculs. L'ALU du Codex est une version 32-bits évoluée. Elle prend deux entrées (**a**, **b**) et un code d'opération (**op**).

#### 4.3.1 Les Opérations de l'ALU32

L'ALU reçoit un signal **op** de 4 bits qui définit l'action à réaliser :

op	Nom	Opération
0000	AND	Et logique ( <b>a &amp; b</b> )
0001	EOR	Ou exclusif ( <b>a ^ b</b> )
0010	SUB	Soustraction ( <b>a - b</b> )
0011	ADD	Addition ( <b>a + b</b> )
0100	ORR	Ou logique ( <b>a   b</b> )



Un “cycle” d’horloge se compose d’un front montant (le passage de 0 à 1) et d’un front descendant. Dans le système Codex, les changements d’état se produisent sur le **front montant** (*rising edge*).

## 5.2 3.2 La Brique de Base : La Bascule D (DFF)

La **DFF** (Data Flip-Flop) est notre atome de mémoire.

- Entrée **d** (Data)
- Sortie **q**
- Horloge **clk** (implicite ou explicite selon les outils)

**Comportement :**  $out(t) = in(t-1)$

La sortie à l’instant  $t$  est égale à ce qu’était l’entrée à l’instant  $t-1$  (le cycle précédent). Elle “mémoire” le bit pendant un cycle.

Dans le simulateur Codex, vous devrez souvent câbler explicitement le signal **clk**.

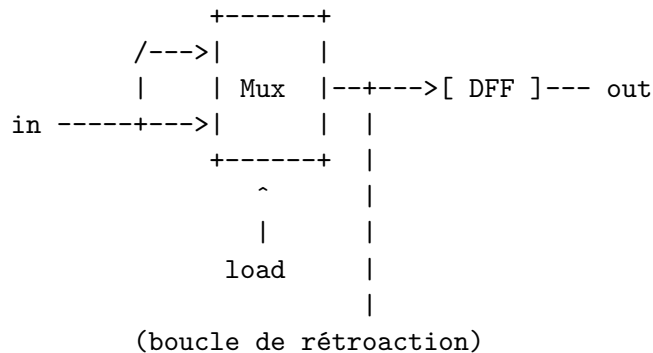
## 5.3 3.3 Construire une Mémoire Persistante

Comment se souvenir d’une valeur indéfiniment, et pas seulement pendant un cycle ? En bouclant la sortie sur l’entrée !

### 5.3.1 Le Registre 1-bit (Bit)

Nous voulons une puce qui : 1. Si **load=1**, stocke la nouvelle valeur **in**. 2. Si **load=0**, garde l’ancienne valeur.

Pour cela, on combine une DFF avec un Multiplexeur.



Si **load=0**, le Mux sélectionne l’ancienne sortie de la DFF. La DFF ré-enregistre donc sa propre valeur. Si **load=1**, le Mux laisse passer la nouvelle donnée **in**.

## 5.4 3.4 Des Bits aux Registres

### 5.4.1 Le Registre 32-bits (Register)

C’est simplement 32 puces **Bit** mises en parallèle, partageant le même signal **load**. C’est la brique fondamentale du CPU pour stocker des entiers.

### 5.4.2 La Mémoire RAM (Random Access Memory)

La RAM est un énorme tableau de registres. Pour lire ou écrire, on doit spécifier une **adresse**.

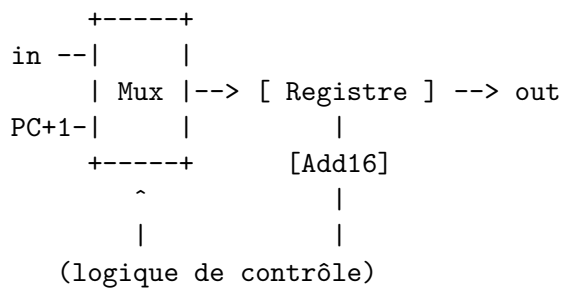
- **RAM8** : 8 registres. Adresse sur 3 bits ( $2^3 = 8$ ).
- **RAM64** : 64 registres. Adresse sur 6 bits.
- ... et ainsi de suite jusqu'à la **RAM4K**.

**Structure d'une RAM8** : On utilise : \* Un **DMux8Way** pour envoyer le signal **load** vers *un seul* des 8 registres (celui qu'on veut écrire). \* 8 **Register** pour stocker les données. \* Un **Mux8Way32** pour sélectionner *une seule* sortie parmi les 8 (celle qu'on veut lire).

## 5.5 3.5 Le Compteur de Programme (PC)

Le PC (Program Counter) est un registre spécial qui indique au CPU l'adresse de la *prochaine instruction* à exécuter.

- Il a trois modes de fonctionnement : 1. **Reset** (**reset=1**) : Retour à 0 (démarrage du programme). 2. **Saut** (**load=1**) : On va à une adresse spécifique (pour les boucles, les **if**, les appels de fonction). 3. **Incrément** (**inc=1**) : On passe à l'instruction suivante (**PC = PC + 1**).



## 5.6 3.6 Vos Exercices

Rendez-vous dans **hdl\_lib/04\_seq** (et **02\_multibit** pour les Mux larges).

**Ordre conseillé :**

1. **Bit.hdl** : Le registre 1 bit (Mux + DFF).
2. **Register.hdl** : Le registre 32 bits.
3. **RAM8.hdl** : Une mémoire de 8 mots de 32 bits.
4. **RAM64.hdl** : Combinez 8 **RAM8** pour faire une **RAM64**.
  - *Astuce* : Les bits de poids fort de l'adresse choisissent *quelle* RAM8 activer. Les bits de poids faible choisissent le registre *dans* cette RAM8.
5. **PC.hdl** : Le compteur de programme. Il faut gérer la priorité : Reset > Load > Inc > Maintien.

**Note sur les bus larges** : Pour construire **RAM8**, vous aurez besoin de **Mux8Way32** et **DMux8Way**. Ces composants sont souvent à implémenter dans le dossier **hdl\_lib/02\_multibit**. Assurez-vous de les faire si ce n'est pas déjà fait !

**Validation** : Testez vos RAMs une par une. Si **RAM8** ne marche pas, **RAM64** ne marchera pas non plus.

cargo run -p hdl\_cli -- test hdl\_lib/04\_seq/RAM8.hdl

Ou via le **Simulateur Web**.

## 6 Chapitre 4 : 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). C’est le contrat entre le matériel (le processeur) et le logiciel (votre code).

L’architecture **Codex A32-Lite** est une architecture **RISC 32-bits**, inspirée de la célèbre architecture ARM. Elle est conçue pour être simple à comprendre mais suffisamment puissante pour supporter un système d’exploitation moderne.

### 6.1 4.1 Concepts Fondamentaux

#### 6.1.1 Pourquoi 32 bits ?

Dire qu’une machine est “32 bits”, c’est dire que ses “mots” standards font 32 bits de large. \* Elle peut compter jusqu’à  $2^{32} \approx 4$  milliards. \* Elle peut adresser 4 Go de mémoire. \* Ses registres font 32 bits.

#### 6.1.2 RISC vs CISC

- **CISC (Complex Instruction Set Computer)** : Comme les processeurs Intel (x86). Ils ont des instructions très complexes (ex: “copie une chaîne de caractères de A à B”). C’est puissant mais difficile à construire.
- **RISC (Reduced Instruction Set Computer)** : Comme ARM, MIPS, ou Codex. Les instructions sont simples et rapides.
  - *Règle d’or* : Le CPU ne calcule **jamais** directement en mémoire. Il doit d’abord charger les données dans des registres ( **Load** ), calculer ( **Add** ), puis sauvegarder ( **Store** ). C’est l’architecture **Load/Store**.

#### 6.1.3 Le Cycle de Vie d’une Instruction

Le cœur du processeur bat au rythme de trois étapes, répétées à l’infini : 1. **Fetch (Chercher)** : Le CPU demande à la mémoire : “Donne-moi l’instruction à l’adresse pointée par mon PC”. 2. **Decode (Décoder)** : Le CPU regarde les bits (0 et 1) et comprend : “Ah, c’est une addition entre R1 et R2”. 3. **Execute (Exécuter)** : L’ALU effectue l’addition et le résultat est rangé.

### 6.2 4.2 La Mémoire (Memory Map)

La mémoire n’est pas juste un sac de données. Elle est organisée géographiquement. L’adresse est sur 32 bits ( `0x00000000` à `0xFFFFFFFF` ).

Adresse Début	Adresse Fin	Nom	Description
<code>0x00000000</code>	<code>0x003FFFFFFF</code>	<b>RAM Système</b>	C’est ici que vit votre programme (Code + Données + Pile).
...	...	<i>Vide</i>	Si vous tapez ici -> <b>MEM_FAULT</b> .

Adresse Début	Adresse Fin	Nom	Description
0x00400000	0x00402580	Screen (Écran)	Écrire ici allume des pixels sur l'écran (Noir & Blanc).
0x00402600	...	Clavier	Lire ici donne la touche pressée.
0xFFFF0000	...	MMIO (E/S)	Adresses magiques pour afficher du texte ou éteindre la machine.

### 6.3 4.3 Les Registres (Le plan de travail)

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

Registre	Nom	Rôle (Convention)
<b>R0 - R3</b>	Args / Return	Pour passer les arguments aux fonctions et récupérer le résultat.
<b>R4 - R11</b>	Variable	Pour stocker vos variables locales.
<b>R12</b>	Temp	Registre temporaire (IP).
<b>R13</b>	<b>SP</b> (Stack Pointer)	Pointe vers le sommet de la Pile (Stack). La pile grandit vers le bas (des adresses hautes vers les basses).
<b>R14</b>	<b>LR</b> (Link Register)	Retient l'adresse de retour lors d'un appel de fonction ( <b>BL</b> ).
<b>R15</b>	<b>PC</b> (Program Counter)	Contient l'adresse de l'instruction <i>en cours d'exécution</i> .

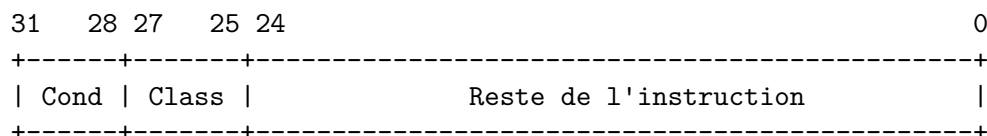
**Note importante sur le PC (R15) :** Si vous écrivez dans R15, vous forcez le processeur à sauter ailleurs. C'est comme ça que fonctionnent les boucles et les **goto**.

### 6.4 4.4 Le Format des Instructions (32 bits)

Chaque instruction est un mot de 32 bits. Le processeur doit "décoder" ces bits pour savoir quoi faire.

#### 6.4.1 Structure Générale

Les bits **31..28** sont *toujours* la **Condition**. Les bits **27..25** définissent la **Classe** de l'instruction.



### 6.4.2 La Condition (Bits 31-28) et la Prédication

C'est une fonctionnalité unique et puissante d'ARM/Codex. **Toutes** les instructions peuvent être conditionnelles. Au lieu de faire des petits sauts ( `if x then jump` ), on peut dire "Exécute cette instruction *seulement si* Zéro est à 1".

Code	Suffixe ASM	Signification	Exemple
1110	(vide)	Always (Toujours)	ADD ... (défaut)
0000	EQ	Equal (Z=1)	ADD.EQ ... (Si égal)
0001	NE	Not Equal (Z=0)	ADD.NE ... (Si différent)
1011	LT	Less Than (Signé)	MOV.LT ... (Si plus petit)
1100	GT	Greater Than (Signé)	MOV.GT ... (Si plus grand)

## 6.5 4.5 Les Classes d'Instructions

### 6.5.1 A. Opérations Arithmétiques (ALU) - Classe 000 et 001

C'est ici qu'on fait les maths. Format : `OP Rd, Rn, Operand2 -> Rd = Rn OP Operand2`

**Encodage :** \* `Op` (24-21) : L'opération (ADD=0011, SUB=0010, MOV=0101...). \* `S` (20) : Si 1, met à jour les Flags (N, Z, C, V). \* `Rn` (19-16) : Premier opérande (registre). \* `Rd` (15-12) : Registre de destination. \* **Operand2** : \* Si Classe 000 : Un registre `Rm` (bits 3-0). \* Si Classe 001 : Une valeur immédiate 12 bits (bits 11-0). Attention, on ne peut mettre que des petits nombres (0-4095).

**Exemple :** `ADD R1, R2, #10` ( $R1 = R2 + 10$ )

### 6.5.2 B. Accès Mémoire (Load/Store) - Classe 010

Pour lire ou écrire dans la RAM. Format : `LDR Rd, [Rn, #offset]`

**Encodage :** \* `L` (24) : 1=Load (LDR), 0=Store (STR). \* `B` (23) : 1=Byte (8 bits), 0=Word (32 bits). \* `U` (21) : 1=Add offset, 0=Subtract offset. \* `Rd` (20-17) : Registre à charger/sauvegarder. \* `Rn` (16-13) : Registre de base (l'adresse). \* `Offset` (12-0) : Décalage immédiat de 13 bits.

**Exemple :** `LDR R0, [R1, #4]` (Charge dans R0 la valeur à l'adresse  $R1 + 4$ ).

### 6.5.3 C. Branchements (Branch) - Classe 011

Pour sauter dans le code. Format : `B Label` ou `BL Label`

**Encodage :** \* `L` (24) : 1=Branch with Link (BL), 0=Branch simple (B). \* `Offset` (23-0) : Décalage signé relatif au PC actuel.

**Le mystère du BL (Branch with Link) :** Quand on appelle une fonction ( `BL fonction` ), le processeur fait deux choses : 1. Il sauvegarde l'adresse de l'instruction suivante dans `LR` (R14). 2. Il saute à l'adresse de la fonction. Pour revenir, la fonction fera simplement `MOV PC, LR`.

## 6.6 4.6 Le Langage Assembleur (ASM) par l'exemple

### 6.6.1 Les Variables et Constantes (Literal Pools)

Comme une instruction ne fait que 32 bits, on ne peut pas y faire entrer une constante de 32 bits (comme une grande adresse). L'assembleur utilise une astuce : le **Literal Pool**.

```
LDR R0, =0x12345678 ; L'assembleur stocke cette valeur plus loin en mémoire
                    ; et génère un LDR relatif au PC pour aller la chercher.
```

### 6.6.2 La Pile (Stack) : Push et Pop

Contrairement à x86, il n'y a pas d'instruction `PUSH` ou `POP`. On utilise `LDR/STR` avec `SP`.

- **PUSH R0** (Empiler R0) : `asm SUB SP, SP, #4 ; On fait de la place (la pile descend)`
- **POP R0** (Dépiler dans R0) : `asm LDR R0, [SP] ; On lit ADD SP, SP, #4 ; On rend`

### 6.6.3 Structures de Contrôle

1. Conditionnelle (If/Else) C : `if (r0 == r1) r2 = 1; else r2 = 0;`

ASM (Classique) :

```
CMP R0, R1
BEQ egaux
MOV R2, #0
B fin
egaux:
MOV R2, #1
fin:
```

ASM (Prédication - Plus rapide !) :

```
CMP R0, R1
MOV.EQ R2, #1 ; Exécuté seulement si égal
MOV.NE R2, #0 ; Exécuté seulement si différent
```

2. Boucle (While) C : `while (r0 < 10) r0++;`

ASM :

```
loop:
CMP R0, #10
BGE done ; Si R0 >= 10, on sort
ADD R0, R0, #1
B loop
done:
```

## 6.7 4.7 Gestion des Erreurs (Traps)

Si le processeur rencontre une situation impossible, il déclenche une **Trap** (piège) et arrête tout.

- **ILLEGAL** : L'instruction ne correspond à aucun format connu.
- **MEM\_FAULT** : Tentative d'accès à une mémoire qui n'existe pas.

- **MISALIGNED** : Tentative de lire un mot de 32 bits (4 octets) à une adresse qui n'est pas multiple de 4 (ex: adresse 3).
- **DIV\_ZERO** : Division par zéro.

## 6.8 4.8 Vos Exercices

Vous ne construisez pas le hardware tout de suite (ce sera au Chapitre 5). Pour l'instant, vous allez **programmer** la machine.

Rendez-vous dans le **Simulateur Web** (Section A32 Assembly) ou utilisez `a32_cli`.

**Défis suggérés :** 1. **Somme :** Écrire un programme qui calcule la somme des nombres de 1 à 10. 2. **Max :** Écrire un programme qui trouve le maximum entre deux registres sans utiliser de branchement ( `MOV.GT` ). 3. **Graphique :** Écrivez dans la mémoire vidéo ( `0x00400000` ) pour dessiner une ligne blanche.

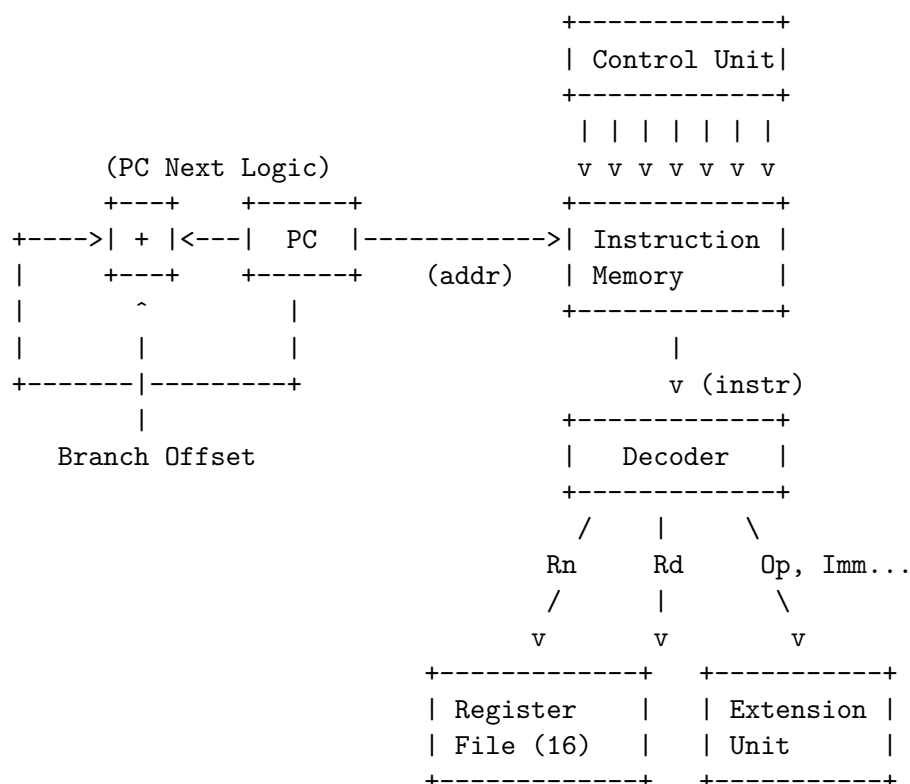
## 7 Chapitre 5 : Le Processeur (CPU)

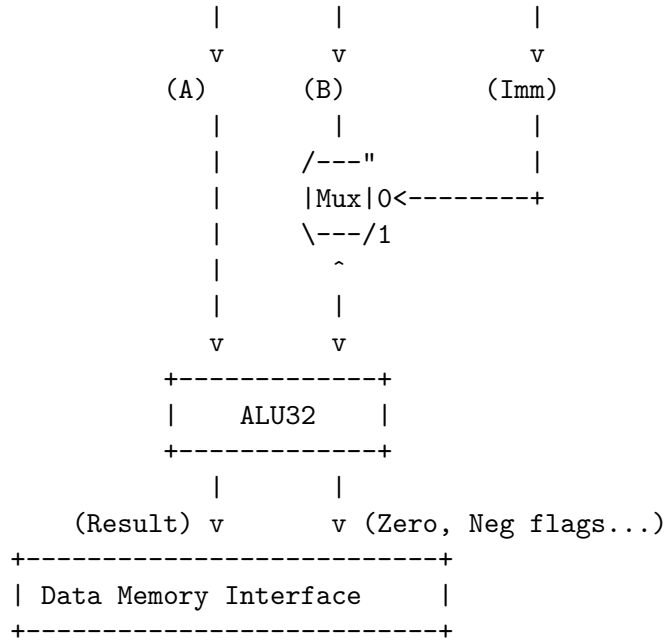
“Si vous ne pouvez pas le construire, vous ne le comprenez pas.”

C'est le grand moment. Nous allons assembler toutes les pièces du puzzle (ALU, Registres, Décodeur, Mémoire) pour construire le cœur de l'ordinateur : le **CPU A32-Lite**.

## 7.1 5.1 Architecture Globale (Data Path)

Voici le schéma complet de l'intérieur du processeur que vous allez construire. Prenez le temps de l'étudier.





## 7.2 5.2 Les Composants Clés

### 7.2.1 1. Le Décodeur ( `Decoder.hdl` )

Le CPU est aveugle. Il reçoit un mot de 32 bits et doit le découper pour activer les bons fils. Le décodeur prend l'instruction brute et extrait : \* `class` : Le type d'instruction (ALU, MEM, BRANCH). \* `op` : L'opération précise (ADD, SUB...). \* `rd`, `rn`, `rm` : Les numéros des registres. \* `imm12`, `imm24` : Les valeurs immédiates.

C'est du pur câblage (des `Or`, des `And` et des connexions de bus).

### 7.2.2 2. L'Unité de Contrôle ( `Control.hdl` )

C'est le chef d'orchestre. En fonction de ce que dit le décodeur, elle active ou désactive les composants. Exemple : Si l'instruction est `LDR` (Load) : \* `mem_read` = 1 (Lire la mémoire) \* `reg_write` = 1 (Écrire dans le registre) \* `alu_src` = 1 (Utiliser l'offset immédiat, pas Rm)

### 7.2.3 3. Le Banc de Registres ( `RegFile16.hdl` )

Il contient les 16 registres R0-R15. Il a deux ports de lecture (`ra`, `rb`) et un port d'écriture (`wa`, `wd`, `we`). \* **Important** : Le registre R15 (PC) est spécial. Dans notre implémentation simplifiée, le PC est souvent géré à part, mais l'architecture logique le place ici.

## 7.3 5.3 Le Cycle d'Exécution en Détail

Notre CPU est "Single-Cycle" : il effectue tout le travail en un seul coup d'horloge.

### 7.3.1 Phase 1 : Fetch (Récupération)

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

### 7.3.2 Phase 2 : Decode (Décodage)

L'instruction est "éclatée" par le `Decoder`. L'unité de contrôle (`Control`) décide quoi faire. Le `RegFile` lit les valeurs des registres sources `Rn` et `Rm`.

### 7.3.3 Phase 3 : Execute (Exécution)

L'ALU entre en scène. \* Entrée A : Toujours `Rn`. \* Entrée B : Soit `Rm`, soit une valeur immédiate (sélectionnée par un Mux). L'ALU calcule le résultat. Elle met aussi à jour les drapeaux (N, Z, C, V) si demandé (`s_flag`).

### 7.3.4 Phase 4 : Memory / Writeback (Mémoire et Écriture)

- Si c'est une opération ALU (ADD...) : Le résultat de l'ALU est renvoyé vers le `RegFile` pour être stocké (`reg_write=1`).
- Si c'est un Store (STR) : La valeur de `Rm` est envoyée à la mémoire de données.
- Si c'est un Load (LDR) : On lit la mémoire de données et le résultat va vers le `RegFile`.

### 7.3.5 Phase 5 : PC Update (Mise à jour du PC)

Où va-t-on ensuite ? \* Par défaut : `PC = PC + 4`. \* Si branchement (`B`, `BL`) : `PC = PC + Offset` (calculé par un additionneur dédié ou l'ALU).

## 7.4 5.4 Vos Exercices

Rendez-vous dans `hdl_lib/05_cpu`. Vous devez implémenter les fichiers dans cet ordre :

1. `CondCheck.hdl` : Un petit circuit qui regarde les drapeaux (N, Z, C, V) et le code de condition (EQ, NE, LT...) et dit "Oui" ou "Non". C'est crucial pour la prédication.
2. `Decoder.hdl` : Découpez l'instruction de 32 bits. Utilisez la syntaxe de sous-bus (`instr[20..24]`).
3. `Control.hdl` : La logique de commande. C'est une grosse table de vérité.
  - *Astuce* : Regardez la classe et l'opcode pour décider si `reg_write` doit être à 1.
4. `CPU.hdl` : Le grand assemblage.
  - Instanciez le `PC`, le `Decoder`, le `Control`, le `RegFile16`, l'`ALU32`.
  - Connectez-les selon le schéma ci-dessus.
  - N'oubliez pas les multiplexeurs pour choisir les entrées de l'ALU et les données à écrire !

## 7.5 5.5 Conseils de débogage

- **Le PC reste bloqué à 0 ?** Vérifiez le signal `inc` de votre PC.
- **Les branchements ne marchent pas ?** Vérifiez le calcul de l'offset (il doit être multiplié par 4 ou décalé, selon la spec).
- **Rien ne s'écrit ?** Vérifiez `reg_write`. Il ne doit être à 1 que si la condition (`CondCheck`) est vraie !

Une fois ce chapitre terminé, vous aurez construit un ordinateur complet capable d'exécuter du vrai code. Félicitations !

## 8 Chapitre 6 : 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 (ex: `ADD`) en un fichier binaire exécutable par le CPU.

### 8.1 6.1 Le Rôle de l'Assembleur

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

### 8.2 6.2 La Stratégie des Deux Passes

Pourquoi ne peut-on pas traduire le fichier d’un seul coup, ligne par ligne ? À cause des **références vers l’avant**.

Regardez ce code :

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

Pour résoudre ce problème, l’assembleur travaille en deux étapes (passes) :

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

L’assembleur parcourt le fichier et compte la taille de chaque instruction. Il note l’adresse de chaque étiquette qu’il rencontre. \* `suite` est à l’adresse 8 (car il y a deux instructions de 4 octets avant). \*  
Table des symboles : `{ "suite": 0x00000008 }`.

#### 8.2.2 Passe 2 : Génération du Code

L’assembleur reparcourt le fichier. Lorsqu’il voit `B suite`, il regarde dans sa table, voit `0x00000008`, calcule la distance relative, et encode l’instruction binaire complète.

### 8.3 6.3 Sections et Directives

Un programme n’est pas fait que d’instructions. Il contient aussi des données (du texte, des constantes). On utilise des **directives** (commençant par un point) pour guider l’assembleur.

#### 8.3.1 Les Sections

- `.text` : C’est ici que vous placez votre code (les instructions). Cette zone est généralement en lecture seule.

- `.data` : Pour les variables globales initialisées (ex: un message de bienvenue).
- `.bss` : Pour les variables globales non initialisées (mises à zéro au démarrage).

### 8.3.2 L'Allocation de Données

- `.word 123` : Réserve 4 octets et y écrit la valeur 123.
- `.asciz "Hello"` : Réserve de la place pour une chaîne de caractères terminée par un zéro (null-terminated).
- `.align 2` : Aligne l'adresse suivante sur un multiple de 4 ( $2^2$ ) octets. C'est crucial pour le CPU Codex qui exige que les instructions soient alignées.

## 8.4 6.4 Exemple d'Encodage

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

1. **Mnémonique** : `ADD` -> Code d'opération `0011`.
2. **Condition** : Pas de suffixe -> `1110` (Always).
3. **Classe** : Immédiat utilisé -> Classe `001`.
4. **Registres** : `R1` (Dest) -> `0001`, `R2` (Src) -> `0010`.
5. **Immédiat** : `10` -> `000000001010` (12 bits).

Résultat final :

Cond	Cls	Op	S	Rn	Rd	Imm12
1110	001	0011	0	0010	0001	000000001010

Soit en hexadécimal : `0xE282100A`.

## 8.5 6.5 La Gestion des Grandes Constantes (=value)

Comme vu au chapitre 4, on ne peut pas mettre une valeur de 32 bits dans une instruction de 32 bits. L'assembleur Codex offre une syntaxe magique : `LDR R0, =0xDEADBEEF`.

Lorsque l'assembleur voit cela : 1. Il ajoute la valeur `0xDEADBEEF` dans une zone spéciale appelée **Literal Pool** (générée via la directive `.pool` ou en fin de fichier). 2. Il remplace l'instruction par un `LDR` relatif au PC : `LDR R0, [PC, #offset]`.

## 8.6 6.6 Vos Exercices

L'outillage de ce projet contient déjà un assembleur (`a32_asm`). Votre but est de comprendre comment il fonctionne ou d'en simuler une partie.

**Exercice 1 : Encodage Manuel** Traduisez les instructions suivantes en binaire (32 bits) : `* MOV R0, #5` \* `SUB R1, R1, #1` \* `B 0` (Un saut sur soi-même, offset = -8 octets car PC est à +8 par rapport à l'instruction en cours dans le pipeline ARM réel, mais dans notre simulateur simplifié, adaptez selon la spec).

**Exercice 2 : La Table des Symboles** Étant donné le code suivant, calculez l'adresse de chaque label (en supposant que le code commence à 0) :

```
.text
start:
    MOV R0, #0
```

```
loop:
    CMP R0, #10
    BEQ end
    ADD R0, R0, #1
    B loop
end:
    HALT
```

**Exercice 3 : Utilisation de l’outil** Créez un fichier `test.a32`, assemblez-le avec `cargo run -p a32_cli -- assemble test.a32 -o test.bin`, puis examinez le contenu binaire (avec `hexdump` ou un éditeur hexadécimal).

*Note* : L’assembleur du projet génère des fichiers au format **A32B** (A32 Binary), qui incluent un en-tête pour le simulateur web.

## 9 Chapitre 7 : Construction du Compilateur

“Pour comprendre la récursivité, il faut d’abord comprendre la récursivité.”

Dans ce chapitre, nous allons réaliser un tour de magie : écrire un compilateur... en C, pour notre propre machine. L’objectif ultime est le **Self-Hosting** : utiliser notre compilateur pour compiler notre compilateur.

Comme notre compilateur principal ( `c32_cli` ) est écrit en Rust (trop complexe pour être porté tout de suite), nous allons étudier et utiliser un **Micro-Compilateur** écrit en C32.

### 9.1 7.1 Le Défi “MicroC”

Nous allons porter le célèbre compilateur pédagogique **TinyC** de Marc Feeley pour qu’il tourne sur notre architecture Codex. Ce compilateur : 1. Est écrit en un seul fichier C. 2. Lit du code C sur l’entrée standard ( `stdin` ). 3. Génère de l’assembleur A32 sur la sortie standard ( `stdout` ).

#### 9.1.1 Limitations de C32

Notre langage C32 a quelques limites par rapport au C standard qu’il faut contourner : \* Pas de `struct` (on utilisera des tableaux et des offsets manuels). \* Pas de `malloc` complexe (on utilisera un gros tableau global comme “tas”). \* Pas de `FILE*` (on utilise `getchar` et `putchar`).

### 9.2 7.2 Le Code Source de MicroC

Voici une version adaptée de TinyC. Copiez ce code dans un fichier `micro_c.c`.

```
// micro_c.c - Un compilateur C minimaliste pour Codex A32
// Basé sur TinyC de Marc Feeley

// --- Lexer ---
int ch;
int sym;
int int_val;
// Symboles
int DO_SYM = 0;    int ELSE_SYM = 1; int IF_SYM = 2; int WHILE_SYM = 3;
```

```

int LBRA = 4;      int RBRA = 5;      int LPAR = 6;      int RPAR = 7;
int PLUS = 8;      int MINUS = 9;      int LESS = 10;      int SEMI = 11;
int EQUAL = 12;     int INT = 13;      int ID = 14;        int EOI = 15;

void next_ch() {
    ch = getchar(); // Appel système A32
}

void next_sym() {
    while (ch == 32 || ch == 10) next_ch(); // Skip espaces et sauts de ligne
    if (ch == -1) { sym = EOI; return; }

    if (ch == 123) { next_ch(); sym = LBRA; return; } // {
    if (ch == 125) { next_ch(); sym = RBRA; return; } // }
    if (ch == 40) { next_ch(); sym = LPAR; return; } // (
    if (ch == 41) { next_ch(); sym = RPAR; return; } // )
    if (ch == 43) { next_ch(); sym = PLUS; return; } // +
    if (ch == 45) { next_ch(); sym = MINUS; return; } // -
    if (ch == 60) { next_ch(); sym = LESS; return; } // <
    if (ch == 59) { next_ch(); sym = SEMI; return; } // ;
    if (ch == 61) { next_ch(); sym = EQUAL; return; } // =

    if (ch >= 48 && ch <= 57) { // 0-9
        int_val = 0;
        while (ch >= 48 && ch <= 57) {
            int_val = int_val * 10 + (ch - 48);
            next_ch();
        }
        sym = INT;
    } else if (ch >= 97 && ch <= 122) { // a-z
        // Simplification : une seule lettre pour les variables
        int_val = ch - 97; // 0 for 'a', 1 for 'b'...
        next_ch();
        // Check keywords (très simplifié pour l'exemple)
        // Dans une vraie implémentation, il faudrait un buffer de string
        sym = ID;
        // Note: Ici on triche, on assume que if/while/do sont gérés
        // par un lexer plus complet ou que l'utilisateur n'utilise pas de variables qui clashent
        // Pour cet exemple pédagogique, supposons des variables d'une lettre et pas de mots clés
    }
}

// --- Générateur de Code (A32 Stack Machine) ---

void g_push(int val) {
    // MOV R0, #val; SUB SP, SP, #4; STR R0, [SP]
    printf(".word 0xE3A00000 + %d\n", val); // Hack: on émet du binaire ou pseudo-asm
    // Pour simplifier l'exemple, imaginons qu'on sort du texte ASM :

```

```

    // printf("MOV R0, #%d\n", val); ...
}

// --- Parser & Compilateur ---

// ... (Le parser récursif descendrait ici, similaire à TinyC)
// ... (Il appellerait g_push, g_add, g_sub pour générer l'ASM A32)

int main() {
    printf(".text\n.global _start\n_start:\n");

    next_ch();
    next_sym();
    // parser(); // Lancer le parsing

    printf("HALT\n");
    return 0;
}

```

*Note : Le code ci-dessus est un squelette. Pour le défi complet, vous devrez porter l'intégralité de la logique de TinyC.*

### 9.3 7.3 L'Expérience de "Self-Compilation"

Voici la procédure pour réaliser l'Inception :

1. **Écriture** : Écrivez votre compilateur `micro_c.c` (en C32).
2. **Compilation** : Utilisez notre compilateur Rust ( `c32_cli` ) pour compiler `micro_c.c` en binaire A32. `bash cargo run -p c32_cli -- compile micro_c.c -o compiler.bin`
3. **Exécution** : Lancez le simulateur avec ce binaire.
  - L'entrée standard ( `stdin` ) sera votre code source C (ex: `i=10;` ).
  - La sortie standard ( `stdout` ) sera le code Assembleur généré !

```
echo "i=10;" | cargo run -p a32_runner -- compiler.bin
```

Si tout fonctionne, vous verrez s'afficher à l'écran :

```

MOV R0, #10
STR R0, [SP, #-4]!
...

```

C'est la preuve que votre ordinateur (simulé) est capable de comprendre et de traduire des langages de haut niveau.

### 9.4 7.4 Comprendre la Génération de Code A32

Votre `micro_c` doit générer de l'assembleur valide. Voici les modèles (templates) à utiliser :

#### 9.4.1 Push Constante

```

printf("MOV R0, #%d\n", val);
printf("SUB SP, SP, #4\n");

```

```
printf("STR R0, [SP]\n");
```

#### 9.4.2 Addition (Pop, Pop, Add, Push)

```
printf("LDR R1, [SP]\n");
printf("ADD SP, SP, #4\n"); // Pop R1
printf("LDR R0, [SP]\n");
printf("ADD SP, SP, #4\n"); // Pop R0
printf("ADD R0, R0, R1\n");
printf("SUB SP, SP, #4\n");
printf("STR R0, [SP]\n"); // Push Res
```

#### 9.4.3 Variables (a-z)

On peut mapper les variables `a..z` à des adresses mémoire fixes (ex: `0x1000 + 4*index`).

```
// Lecture 'a'
printf("LDR R0, =0x1000\n");
printf("LDR R0, [R0]\n");
printf("SUB SP, SP, #4\n");
printf("STR R0, [SP]\n");
```

### 9.5 7.5 Vos Exercices

**Défi Ultime :** Portez le code complet de `tinyc.c` (disponible dans les ressources du livre original ou sur le web) pour qu'il fonctionne en C32. 1. Remplacez les `enum` par des `int`. 2. Remplacez les `struct` par des variables globales ou des tableaux. 3. Remplacez la génération de bytecode par des `printf` d'instructions ASM A32.

Si vous y arrivez, vous aurez créé un compilateur C qui tourne sur votre propre CPU !

# Chapitre 8 : 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 murmurer à l'oreille en assemblage.

C'est ici qu'intervient le **C32**. C'est un langage de haut niveau qui vous permet de vous concentrer sur le logiciel.

## 8.1 Pourquoi un langage de haut niveau ?

Imaginez que vous vouliez calculer la moyenne de 10 nombres.

\* **En Assembleur** : Vous devez décider quel registre contient le compteur, lequel contient la somme, lequel la moyenne, etc.

**L'Abstraction :**

```
''ascii
[ Pensée Humaine ] -> "Calculer la moyenne"
|
```

```

      v
[ Langage C32 ]    -> sum = sum + tab[i];
      |
      v
[ Assembleur A32 ] -> LDR R0, [R1, R2]; ADD R3, R3, R0...
      |
      v
[ Code Machine ]  -> 0xE0833000...

```

## 9.6 8.2 Spécification du Langage C32

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

### 9.6.1 1. Les Types de Données

Le Codex est une machine 32-bits, donc le type de base est le mot de 32 bits.

Type	Taille	Description
<code>int</code>	32 bits	Entier signé (Complément à 2).
<code>uint</code>	32 bits	Entier non-signé (positif uniquement).
<code>char</code>	8 bits	Caractère (ASCII).
<code>bool</code>	1 bit	<code>true</code> ou <code>false</code> .
<code>void</code>	-	Type "vide" pour les fonctions qui ne retournent rien.
<code>type*</code>	32 bits	Pointeur (adresse mémoire d'un autre objet).

### 9.6.2 2. Variables et Portée

- **Variables Globales** : Déclarées en dehors des fonctions. Elles vivent dans la section `.data` ou `.bss` et sont accessibles partout.
- **Variables Locales** : Déclarées à l'intérieur d'une fonction. Elles vivent sur la **Pile (Stack)** et disparaissent quand la fonction se termine.

### 9.6.3 3. Structures de Contrôle

**Si / Sinon (If / Else) :**

```

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

```

**Boucle Tant Que (While) :**

```

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

```

**Boucle Pour (For) :**

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

## 9.7 8.3 Pointeurs et Tableaux

C'est ici que le C32 montre sa puissance et sa proximité avec le matériel.

### 9.7.1 Les Tableaux

Un tableau est une suite de valeurs consécutives en mémoire.

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

### 9.7.2 Les 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
```

**Pourquoi est-ce important ?** C'est grâce aux pointeurs que nous allons pouvoir manipuler le matériel (Screen, Keyboard) en écrivant directement aux adresses magiques définies au Chapitre 4.

## 9.8 8.4 Accès au Matériel (MMIO)

En C32, on accède aux périphériques en utilisant des pointeurs vers des adresses fixes.

```
// L'écran commence à 0x00400000
void clear_screen() {
    uint* screen = (uint*)0x00400000;
    for (int i = 0; i < 9600/4; i = i + 1) {
        screen[i] = 0;
    }
}

// Lire une touche
int get_key() {
    int* keyboard = (int*)0x00402600;
    return *keyboard;
}
```

## 9.9 8.5 Organisation d'un programme C32

Tout programme doit avoir une fonction `main`, qui est le point d'entrée.

```
// Inclusion de bibliothèques (si disponibles)
extern void putchar(char c);

int main() {
```

```

char* msg = "Hello Codex!";
int i = 0;
while (msg[i] != 0) {
    putchar(msg[i]);
    i = i + 1;
}
return 0;
}

```

## 9.10 8.6 Limitations (Ce que le C32 ne fait pas encore)

Pour rester simple et pédagogique, le compilateur C32 actuel a quelques limites : 1. **Pas de `struct`** : Vous ne pouvez pas créer de types complexes. 2. **Pas de flottants** : Uniquement des entiers (pas de `float` ou `double`). 3. **Gestion mémoire manuelle** : Il n'y a pas de ramasse-miettes (Garbage Collector). Si vous voulez de la mémoire dynamique, vous devez utiliser `malloc` (que nous verrons au Chapitre 9).

## 9.11 8.7 Vos Exercices

Rendez-vous dans le dossier `tests_c/`. Vous y trouverez une multitude de programmes de test.

**Défis suggérés** : 1. **T01\_sum\_to.c** : Calculez la somme des entiers de 1 à N. 2. **T06\_array\_ptr.c** : Manipulez un tableau via des pointeurs. 3. **Application Graphique** : Créez un fichier `mon_dessin.c` qui dessine un rectangle sur l'écran en utilisant une boucle `for` et l'adresse `0x00400000`.

**Compilation et Test :**

```

# Compiler le fichier C vers un binaire A32
cargo run -p c32_cli -- compile mon_dessin.c -o mon_dessin.bin

# Exécuter dans le simulateur
cargo run -p a32_runner -- mon_dessin.bin

```

Ou, plus simplement, utilisez le **Simulateur Web** pour une visualisation immédiate !

# 10 Chapitre 9 : Système d'Exploitation

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

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** (`os_lib`). Ce n'est pas encore un Windows ou un Linux, mais c'est la fondation de ce qu'on appelle un **Système d'Exploitation** : une couche logicielle qui simplifie l'accès au matériel.

## 10.1 9.1 La Hiérarchie Logicielle

L'OS s'insère entre votre application et le matériel brut.

Application	(Ex: Snake, Calculatrice)
Bibliothèque OS	(os_lib: malloc, draw_line, printf)
Matériel (CPU)	(Registres, MMIO, Interruptions)

## 10.2 9.2 Gestion de la Mémoire (Le Tas / Heap)

Jusqu'à présent, nous utilisons des variables globales ou locales (sur la pile). Mais que faire si nous avons besoin d'une quantité de mémoire inconnue à l'avance (ex: charger une image) ?

C'est le rôle de l'allocateur de mémoire (`alloc.c`). Il gère une zone appelée le **Tas (Heap)**.

### 10.2.1 L'Allocateur "Bump" (Simple)

C'est la méthode la plus simple : on a un pointeur qui avance à chaque allocation.

```
char *malloc(uint size) {
    uint result = heap_ptr;
    heap_ptr = heap_ptr + size; // On avance le curseur
    return (char *)result;
}
```

*Inconvénient* : On ne peut pas libérer (`free`) de la mémoire individuellement.

### 10.2.2 L'Allocateur par Liste Chaînée (Avancé)

L'OS Codex propose aussi une version plus maligne où chaque bloc de mémoire libre contient un pointeur vers le suivant. Cela permet de réutiliser les trous laissés par les blocs libérés.

## 10.3 9.3 Bibliothèque Graphique (`screen.c`)

Plutôt que de calculer manuellement les bits pour chaque pixel (comme nous l'avons fait au Chapitre 8), l'OS nous offre des fonctions de haut niveau.

**L'algorithme de Bresenham** : C'est l'algorithme classique utilisé dans `os_lib` pour dessiner des lignes droites avec uniquement des additions et des soustractions (pas besoin de nombres à virgule !).

```
void screen_draw_line(int x1, int y1, int x2, int y2) {
    // Calcul complexe des pixels...
}
```

Grâce à l'OS, dessiner devient un jeu d'enfant :

```
screen_init();
screen_set_color(COLOR_BLACK);
screen_draw_circle(160, 120, 50); // Un cercle au centre !
```

## 10.4 9.4 Entrées / Sorties et Système (`io.c`, `sys.c`)

L'OS cache aussi la complexité des appels système (`SVC`).

- `printf` : Formate et affiche du texte en utilisant `putchar`.
- `keyboard_read` : Lit une touche et gère les codes spéciaux (flèches, entrée).
- `exit(code)` : Arrête proprement le programme.

## 10.5 9.5 Interruptions et Multitâche (Concept)

Bien que nous ne construisions pas un système multitâche complet, le Codex supporte les **Interruptions**. Imaginez que vous jouez à un jeu (Snake) : 1. Le CPU exécute le code du jeu. 2. Un **Timer matériel** déclenche une interruption toutes les 10ms. 3. Le CPU s'arrête instantanément et saute vers un "Handler" de l'OS. 4. L'OS met à jour l'horloge ou vérifie le clavier. 5. Le CPU reprend le jeu là où il s'était arrêté.

C'est la base de tout système moderne : le matériel peut "forcer" le logiciel à réagir à un événement.

## 10.6 9.6 Le Grand Final : Votre Application

Vous avez tous les outils en main. Le répertoire  `demos/`  contient des exemples de ce que vous pouvez réaliser avec l'OS Codex : \*  `01_hello`  : Le classique. \*  `04_snake`  : Un jeu complet utilisant les graphismes et le clavier. \*  `05_shell`  : Une interface en ligne de commande qui tourne sur votre CPU !

## 10.7 Vos Exercices

**Exercice 1 : Extension de l'OS** Ajoutez une fonction  `screen_draw_triangle`  à  `os_lib/screen.c`  en utilisant trois appels à  `screen_draw_line` .

**Exercice 2 : Gestion Mémoire** Testez l'allocateur. Allouez 10 petits blocs avec  `malloc` , puis essayez d'en libérer certains avec la version  `ll_free`  (liste chaînée). Observez-vous une fragmentation ?

**Exercice 3 : Le Projet Final** Créez votre propre application dans un nouveau dossier  `demos/08_mon_projet/` . Utilisez tout ce que vous avez appris : le HDL pour comprendre le CPU, l'assembleur pour optimiser les parties critiques, le C32 pour la logique, et l'OS pour l'interface.

---

**Félicitations !** Vous avez parcouru tout le chemin, de la porte NAND au système d'exploitation. Vous comprenez maintenant que l'ordinateur n'est pas une boîte magique, mais une immense pile d'abstractions magnifiquement ordonnées.