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.

Où en sommes-nous ?

```
                    COUCHE 7: Applications

                 COUCHE 6: Système d'Exploitation

                COUCHE 5: Langage de Haut Niveau (C32)

                     COUCHE 4: Compilateur

                   COUCHE 3: Assembleur (A32 ASM)

                 COUCHE 2: Architecture Machine (ISA)

                    COUCHE 1: Logique Matérielle
               (Portes logiques, ALU, RAM, CPU)

          COUCHE 0: La Porte NAND
                   (Vous êtes ici !)
```

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.

Pourquoi le Binaire ?

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.

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.

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.

La Porte NAND : Notre Axiome

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.

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.

Symbole graphique

Le petit cercle (○) indique l'inversion

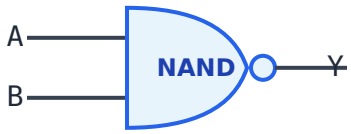


Figure 1: Porte NAND

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).

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 !

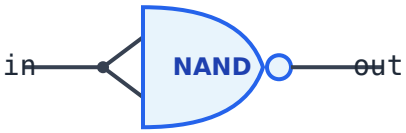


Figure 2: NOT construit à partir de NAND

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 !

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 !

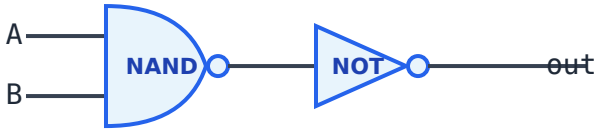


Figure 3: AND construit à partir de NAND et NOT

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

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)
```

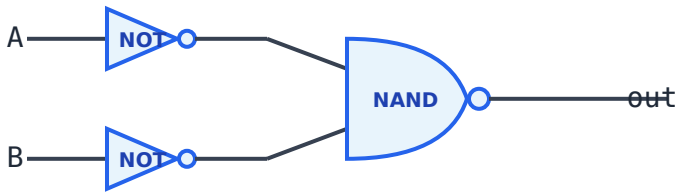


Figure 4: OR construit à partir de NOT et NAND

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".

E. Multiplexeur (Mux) — L'aiguilleur

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

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

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

```
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

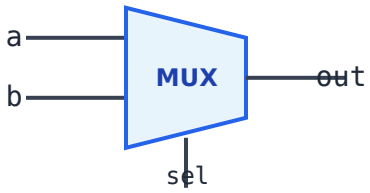


Figure 5: Multiplexeur

? - Où va le résultat ? Vers la mémoire ou un registre ? - Quelle instruction exécuter ?

Chacun de ces choix est implémenté par un multiplexeur !

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`

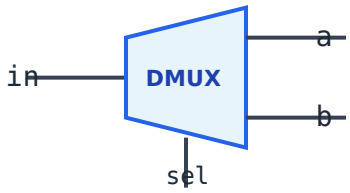


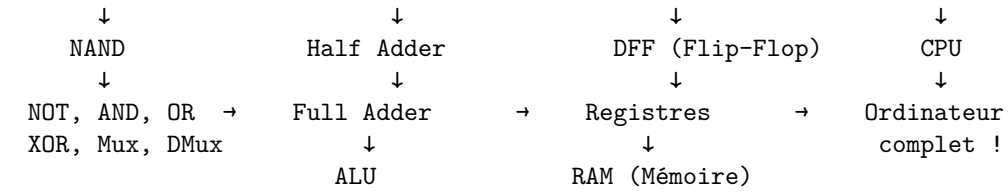
Figure 6: Démultiplexeur

Le Lien avec l'Ordinateur Complet

Prenons du recul. Pourquoi construisons-nous ces portes ?

Du NAND au CPU : La feuille de route

`CHAPITRE 1 (Ici)` `CHAPITRE 2` `CHAPITRE 3` `CHAPITRE 5`



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 !

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.

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.

Structure d'un fichier `.hdl`

Un fichier HDL se compose de deux parties :

```
-- ============================================================
-- 1. L'ENTITÉ : L'interface externe (les "broches" de la puce)
-- ============================================================
entity NomDeLaPuce is
  port(
    a : in bit;      -- Entrée de 1 bit
    b : in bit;      -- Autre entrée
```

```
    y : out bit      -- Sortie (pas de point-virgule sur la dernière)
  );
end entity;

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

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

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

  -- Assignation directe (optionnel)
  y <= fil_interne;

end architecture;
```

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)

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.)

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
```

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;
```

Exercices Pratiques

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

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`).

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/
```

Défis Supplémentaires

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.

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.

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]`).

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 !