1 Représentation de l'information

1.1 OPÉRATIONS DE BASE

1.1.1 Le modulo

Définition 1.1.1: Modulo

Le *modulo* est l'opération qui donne le reste de la division entière de deux nombres. On le note %.

$a\%b = a - b\left\lfloor \frac{a}{b} \right\rfloor$ où $\lfloor q \rfloor$ est l'arrondi à l'inférieur de $q \in \mathbb{R}$

Pour comprendre le principe On prends la ligne des entiers et on l'enroule en spirale. On prends une roue coupée en b parts comme une pizza ou une horloge. On la gradue avec des nombre de 0 à $b-1$ (On veux b tranches) et en enroule la ligne des entiers autour. On commence à zéro et on avance sur la roue de a parts. Le modulo est le nombre sur lequel on tombe.

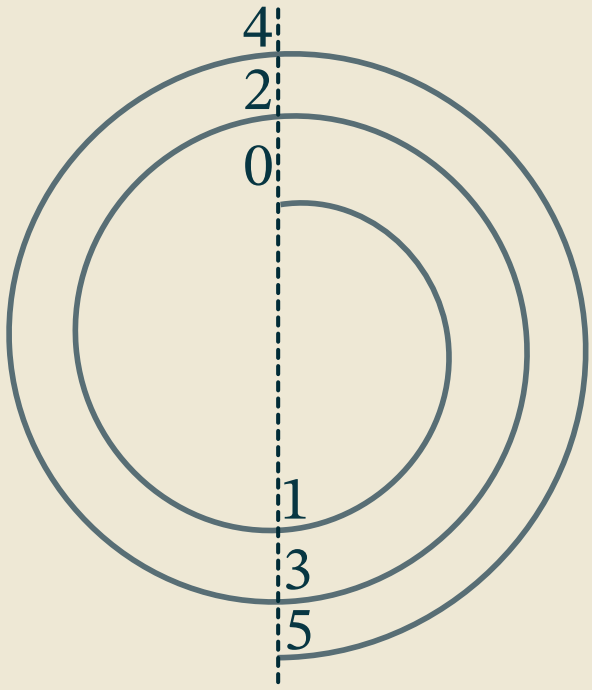
Dans le cas du *modulo* 2, le résultat est de 0 si le nombre est paire et 1 s'il est impaire.

Exemple

Prenons le cas du nombre 12. Cela devient comme sur une horloge. Par exemple $14\%12$ c'est comme 14h donc 2h.

Exemple

Si on prends $5\%2$, on a $5-2\left\lfloor\frac{5}{2}\right\rfloor = 5-2\times 2 = 5-4 = 1$.



Exercice

Faire le schéma de $12\%8$ et de $3\%2$.

1.1.2 Puissance et logarithme

Un petit rappel sur les puissances et logarithmes.

Définition 1.1.2: Puissance

La *puissance* est une petite notation placée en haut à droite d'un nombre, appelée *base*, pour indiquer combien de fois cette base doit être multipliée par elle-même. On écrit cette opération sous la forme $b^n = a$.

Cas particulier $b^0 = 1$

Exemple

Par exemple, 3^4 signifie que 3 est multiplié par lui-même 4 fois : $3^4 = 3\times 3\times 3\times 3 = 81$

Définition 1.1.3: Logarithme

Le logarithme en base b d'un nombre a est l'inverse de la puissance. On le note $log_b(a) = n$.

Le logarithme répond à la question suivante : "À quelle puissance doit-on élever une base donnée pour obtenir un certain nombre ?"

Avertissement

Le logarithme naturel ln **n'est pas** le logarithme en base n. En effet $ln = log_e$ où e est le nombre d'Euler. Faites attention sur vos calculatrices !

$e \approx 2,7182818$

1.2 BASES NUMÉRIQUES

1.2.1 Notions de base

Un peu de vocabulaire : un chiffre est un caractère numérique unique. Il en existe 10 dans le système décimal. Un nombre est une suite de chiffres.

Exemple

Par exemple, 42 est un nombre constitué des chiffres 4 et 2.

1.2.2 Base décimale

La base que l'on utilisez tout les jours est la base *décimale* aussi appelée base 10. Elle utilise 10 chiffre de 0 à 9. Chaque position dans un nombre décimal représente une puissance de 10.

Exemple

Par exemple, 1729 en base 10 signifie : $1729 = 1 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 9 \times 10^0 = 1000 + 700 + 20 + 9$

On note n_b la représentation du nombre n en base b. Dans l'exemple précèdant on peut noter 1729_{10}.

1.2.3 Base binaire

La base 2 utilise seulement 2 chiffres, 0 et 1, et est largement utilisée en informatique. Chaque position dans un nombre binaire représente une puissance de 2.

Exemple

Par exemple, le nombre 1011 en base 2 signifie : $1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11_{10}$

1.2.4 Autre bases notables

- **Base 8 (Octale) :** Moins courante, la base 8 utilise les chiffres 0 à 7. Elle est utilisée en informatique pour représenter des données binaires en regroupant les bits par groupes de 3.

  Exemple

  $57_8 = 5 \times 8^1 + 7 \times 8^0 = 40 + 7 = 47$

  Conversion : Pour convertir un nombre binaire en octal, regroupez les bits en triplets à partir de la droite, puis convertissez chaque groupe en chiffre octal.

  Exemple

  $110101_2 \rightarrow 110.101 = 6.5 = 65_8$

- **Base 16 (Hexadécimale) :** Utilise 16 symboles (0-9, A-F). Elle est couramment employée pour représenter des nombres binaires de manière compacte.

  Exemple

  $1A3_{16} = 1 \times 16^2 + A \times 16^1 + 3 \times 16^0 = 1 \times 256 + 10 \times 16 + 3 = 419$

  Conversion : Pour convertir un nombre binaire en hexadécimal, regroupez les bits en groupes de 4, puis convertissez chaque groupe en chiffre hexadécimal.

  Exemple

  $11010111_2 \rightarrow 1101\,0111 = D\,7 = D7_{16}$

- **Base 64 :** Utilisée principalement pour l'encodage des données (comme dans les URL ou les e-mails), la base 64 emploie 64 caractères (lettres majuscules A-Z, minuscules a-z, chiffres 0-9, et deux symboles, généralement + et /).

  Chaque caractère représente un groupe de 6 bits. Par exemple, un octet (8 bits) est divisé en deux groupes de 6 bits chacun, et les deux bits restants sont utilisés comme remplissage.

Exemple

L'octet 01101011 (107 en décimal) est représenté en base 64 par deux caractères. Le premier groupe de 6 bits "011010" correspond à 26 (base 10), qui est "a" en base 64, et les deux bits restants sont complétés pour former un second groupe.

La base 64 est particulièrement utile pour transporter des données binaires sur des canaux qui ne supportent que du texte.

1.2.5 Conversion de base

Pour convertir un nombre n vers une base b il faut utiliser la méthode suivante :

- À la première position vous mettez le chiffre correspondant à $n\%b$
- Vous effectuez $\lfloor n \div b \rfloor = n'$
- À la position suivante vous mettez le chiffre correspondant à $n'\%b$
- Et ainsi de suite jusqu'a ce que $n' = 0$

Exemple: Décimal en Binaire

Convertir 45_{10} en binaire :

- $45 \div 2 = 22$ reste 1
- $22 \div 2 = 11$ reste 0
- $11 \div 2 = 5$ reste 1
- $5 \div 2 = 2$ reste 1
- $2 \div 2 = 1$ reste 0
- $1 \div 2 = 0$ reste 1

Lire les restes de bas en haut : $45_{10} = 101101_2$.

Exemple: Décimal en Hexadécimal

Convertir 254_{10} en hexadécimal :

- $254 \div 16 = 15$ reste 14 (E en hexadécimal)
- $15 \div 16 = 0$ reste 15 (F en hexadécimal)

Lire les restes de bas en haut : $254_{10} = FE_{16}$.

Pour connaître le nombre de chiffres d'un nombre n dans la base b on peut faire $\lfloor log_b(n) \rfloor$.

On est très habitué à la base décimale. C'est pourquoi il est souvent souhaitable de convertir vers la base décimale. Pour ce faire on utilise la méthode suivante :

- De droite à gauche, mettez les puissances de b en dessous des chiffres en commençant par 0.
- Multipliez chaque nombre avec sa puissance correspondante.
- Additionner les nombres résultats.

Exemple: Binaire en Décimal

Prenons le nombre binaire 1011 (base 2). Chaque chiffre est une puissance de 2, en commençant par la droite (la position 0).

- $1011_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
- $= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$
- $= 8 + 0 + 2 + 1$
- $= 11_{10}$

Donc, $1011_2 = 11_{10}$.

Exemple: Hexadécimal en Décimal

Prenons le nombre hexadécimal 3A (base 16). Chaque position représente une puissance de 16.

- $3A_{16} = 3 \times 16^1 + A \times 16^0$
- $A = 10$ en base 10, donc :
- $= 3 \times 16 + 10 \times 1$
- $= 48 + 10$
- $= 58_{10}$

Donc, $3A_{16} = 58_{10}$.

Exercice

Écrire 101010_2 en hexadécimal en passant par le décimal.

1.3 REPRÉSENTATION PHYSIQUE

1.3.1 La Base Binaire et les Circuits Électroniques

En informatique, les nombres sont représentés en **binaire** (base 2) car les ordinateurs fonctionnent à partir de circuits électroniques qui ont deux états stables :

- **État haut** (1), correspondant à une tension électrique positive.

- **État bas** (0), correspondant à une absence de tension ou à une tension négative.

Ces deux états correspondent aux **bits** (binary digits) qui sont les unités de base de l'information en informatique. Un ensemble de bits peut représenter des nombres, des caractères, des instructions machine, etc.

Par exemple, un groupe de 8 bits forme un **octet** (ou **byte** en anglais), qui est capable de représenter 256 valeurs différentes (de 0 à 255 en décimal).

1.3.2 Les Octets et les Registres

Un **octet** est composé de 8 bits. Les octets sont les unités fondamentales de la mémoire informatique. Ils permettent de stocker des valeurs entières, des caractères, et d'autres types de données.

Chaque octet peut représenter :

- **Un entier non signé :** une valeur entre 0 et 255.
- **Un caractère ASCII :** par exemple, le code ASCII pour la lettre "A" est 65.

Les **registres** sont des petites unités de stockage ultra-rapides situées directement dans le processeur (**CPU**). Ils sont utilisés pour les opérations arithmétiques, logiques, et autres traitements.

- **Registre 32-bit :** Peut contenir 32 bits d'information, ce qui permet de représenter des entiers non signés jusqu'à $2^{32} - 1$ (environ 4,29 milliards).
- **Registre 64-bit :** Peut contenir 64 bits d'information, ce qui permet de représenter des entiers non signés jusqu'à $2^{64} - 1$ (environ 18,4 quintillions).

Les processeurs modernes utilisent généralement des registres 64 bits, ce qui permet de traiter des quantités de données plus importantes et d'adresser une mémoire beaucoup plus grande par rapport aux processeurs 32 bits.

1.3.3 Nombres Signés

Pour représenter des nombres entiers signés (positifs et négatifs), on utilise souvent la méthode du **complément à 2.**

- Le bit le plus à gauche (bit de poids fort) indique le signe : 0 pour positif, 1 pour négatif.
- Le reste des bits représente la valeur absolue du nombre.

Exemple

- En 8 bits :
  - $00000001_2 = +1_{10}$
  - $11111111_2 = -1_{10}$
  - $10000000_2 = -128_{10}$

La plage de valeurs pour un entier signé sur 8 bits est de -128 à 127.

1.3.4 Nombres Flottants

Les **nombres flottants** sont utilisés pour représenter des valeurs décimales (réelles) comme 3.14159 ou 0.000001, qui ne peuvent pas être représentées avec des entiers. Ces nombres sont représentés en informatique selon le standard **IEEE 754**.

1.3.4.1 Structure d'un Nombre Flottant

Un nombre flottant en virgule flottante se compose de trois parties :

1. **Le signe (1 bit) :** Indique si le nombre est positif (0) ou négatif (1).
2. **L'exposant (8 ou 11 bits) :** Indique la puissance de 2 par laquelle la mantisse doit être multipliée.
3. **La mantisse (23 ou 52 bits) :** Contient les chiffres significatifs du nombre.

Exemple

Le nombre 13.25_{10} en binaire est 1101.01_2.
Pour le représenter en flottant :

- Normaliser : $1101.01_2 = 1.10101 \times 2^3$
- Le signe est 0 (positif).
- L'exposant est 3, codé en ajoutant un biais de 127 : $3 + 127 = 130$, donc $130_{10} = 10000010_2$.
- La mantisse est 10101 (on retire le "1" initial).

Le nombre en IEEE 754 est donc :
$0\,|\,10000010\,|\,10101000000000000000000$

1.3.4.2 Précision Simple vs Double

- **Simple précision :** 32 bits, avec une précision d'environ 7 chiffres décimaux.

- **Double précision :** 64 bits (1 bit pour le signe, 11 bits pour l'exposant, 52 bits pour la mantisse), avec une précision d'environ 15-16 chiffres décimaux.

1.3.4.3 Limites et Bugs

- **Précision limitée :** Les nombres flottants peuvent introduire des erreurs d'arrondi. Par exemple, certains nombres décimaux simples comme 0.1 ne peuvent pas être représentés exactement en binaire.
- **Dépassement d'entier :** En arithmétique entière, l'ajout de deux nombres très grands peut provoquer un dépassement (overflow), donnant un résultat incorrect.
- **Sous-flux et sur-flux en flottant :** Les nombres flottants très petits ou très grands peuvent entraîner des erreurs de sous-flux (underflow) ou sur-flux (overflow), où les nombres sont arrondis à zéro ou à l'infini.

2 Logique Booléenne

2.1 PROPOSITIONS ET VALEUR DE VÉRITÉ

2.1.1 Propositions

Définition 2.1.1: Proposition

En logique, une **proposition** est une déclaration qui est soit **vraie** (True) soit **fausse** (False).

Exemple

- "Le soleil se lève à l'est." est une proposition vraie.
- "2 + 2 = 5." est une proposition fausse.

Les propositions sont les éléments de base de la logique booléenne, où elles sont notées par des lettres comme P, Q, R, etc.

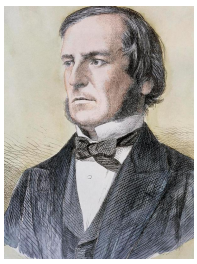
2.1.2 Valeur de Vérité

Définition 2.1.2: Valeur de vérité

La **valeur de vérité** d'une proposition est la valeur (True ou False) associée à cette proposition.

- Si une proposition est vraie, on note sa valeur de vérité V ou 1.
- Si une proposition est fausse, on note sa valeur de vérité F ou 0.

2.2 ALGÈBRE DE BOOLE



George Boole (1815 - 1864)

L'algèbre de Boole est un système mathématique qui traite des opérations sur des valeurs de vérité (True/False ou 1/0). Elle est essentielle en informatique pour concevoir des circuits logiques et des programmes.

2.2.1 Opérateurs Booléens

Définition 2.2.1: Complément

- **NON (NOT) :** L'opération $\neg P$ inverse la valeur de vérité de P.

P	$\neg P$
0	1
1	0

Définition 2.2.2: OU logique

- **OU (OR) :** L'opération $P \vee Q$ est vraie si au moins l'une des propositions P ou Q est vraie. C'est une *addition* logique.

P	Q	$P \vee Q$
0	0	0
0	1	1
1	0	1
1	1	1

Définition 2.2.3: ET logique

- **ET (AND) :** L'opération $P \wedge Q$ est vraie si et seulement si P et Q sont toutes deux vraies. C'est un *produit* logique.

P	Q	$P \wedge Q$
0	0	0
0	1	0
1	0	0
1	1	1

Définition 2.2.4: NON-OU logique

- **NON-OU (NOR) :** L'opération $P \overline{\vee} Q$ est vraie si aucune des propositions P ou Q est vraie.

P	Q	$P \overline{\vee} Q$
0	0	1
0	1	0
1	0	0
1	1	0

Définition 2.2.5: NON-ET logique

- **NON-ET (NAND) :** L'opération $P \overline{\wedge} Q$ est fausse si P et Q sont toutes deux vraies. Sinon l'opération est vraie.

P	Q	$P \overline{\wedge} Q$
0	0	1
0	1	1
1	0	1
1	1	0

Définition 2.2.6: OU exclusif

- **X-OU (XOR) :** L'opération $P \veebar Q$ est vraie si une des propositions P ou Q est vraie mais pas les deux.

P	Q	$P \veebar Q$
0	0	0
0	1	1
1	0	1
1	1	0

2.2.2 Théorèmes de Base

Theorème 2.2.1: Identité

- $P \wedge 1 = P$
- $P \vee 0 = P$

Theorème 2.2.2: Domination

- $P \vee 1 = 1$
- $P \wedge 0 = 0$

Theorème 2.2.3: Double Négation

- $\neg(\neg P) = P$

Theorème 2.2.4: Idempotence

- $P \wedge P = P$
- $P \vee P = P$

Theorème 2.2.5: Complémentarité

- $P \wedge \neg P = 0$
- $P \vee \neg P = 1$

Theorème 2.2.6: Distributivité

- $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
- $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$

2.2.3 Théorèmes de De Morgan

Les **lois de De Morgan** sont des règles fondamentales en algèbre de Boole, très utiles pour simplifier les expressions logiques.

Auguste De Morgan (1806 - 1871) il a fondé avec Boole la logique moderne

Theorème 2.2.7: De Morgan

- **Première loi de De Morgan :** $\neg(P \wedge Q) = \neg P \vee \neg Q$
  Cela signifie que la négation d'un "ET" est équivalente à l'"OU" des négations.
- **Deuxième loi de De Morgan :** $\neg(P \vee Q) = \neg P \wedge \neg Q$
  Cela signifie que la négation d'un "OU" est équivalente à l'"ET" des négations.

Exemple

Simplifier l'expression $\neg(A \wedge B)$:
En appliquant la première loi de De Morgan :
$\neg(A \wedge B) = \neg A \vee \neg B$

2.3 SIMPLIFICATION DES FORMULES

2.3.1 Tableau de Karnaugh

Maurice Karnaugh (1924 - 2022) Ingénieur en télécomunication au laboratoire Bell.

Le **tableau de Karnaugh** (ou carte de Karnaugh) est un outil graphique utilisé pour simplifier les expressions logiques. Il permet de visualiser les combinaisons de valeurs de vérité et de minimiser les termes en regroupant les 1 de manière optimale.

Pour une fonction de deux variables A et B, le tableau de Karnaugh est un carré 2x2, chaque cellule représentant une combinaison de A et B.

A/B	0	1
0	-	-
1	-	-

- Les lignes et colonnes sont étiquetées avec les valeurs possibles des variables.
- Les cases sont remplies avec les valeurs de la fonction pour chaque combinaison.

Pour simplifier une fonction booléenne :

1. **Identifier les 1 dans le tableau :** Repérez les cases contenant des 1.
2. **Regrouper les 1 adjacents :** Formez des groupes (rectangles) de 1 adjacents de tailles 1, 2 et 4.
3. **Écrire la nouvelle expression :** Pour chaque groupe, identifiez les variables qui restent constantes (même valeur dans tout le groupe) et écrivez le produit (ET) de ces variables.

Exemple

Simplifions la fonction suivante $f(A,B)$ donnée par sa table de vérité :

A/B	0	1
0	1	1
1	0	1

En regroupant les 1, vous pourriez obtenir une expression simplifiée :
$f(A,B) = B \vee \neg A$

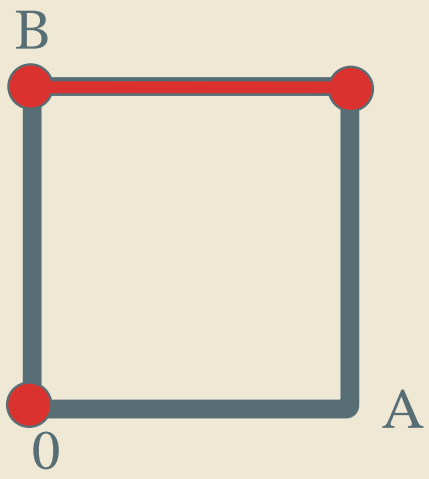
2.3.2 Autre méthode

2.3.2.1 Avec 2 variables

Une autre méthode similaire consiste à faire un carré avec un axe pour chaque variable. On place un point à chaque sommet correspondant à un 1. Ensuite ou regroupe les points par carré, arrete ou sommet.

Exemple

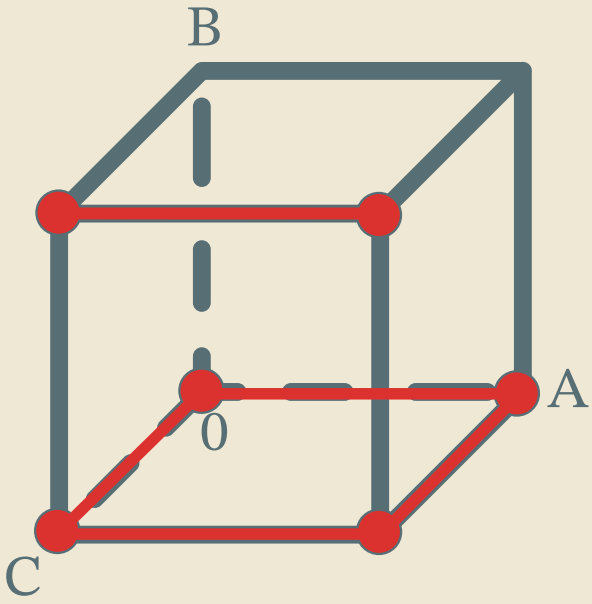
Pour L'exemple précédent, on peut obtenir le carré suivant:



2.3.2.2 Avec 3 variables

Pour ajouter une variable on ajoute une dimension. Passez le carré en cube en ajoutant l'axe pour la troisième variable. La méthode reste inchangée.

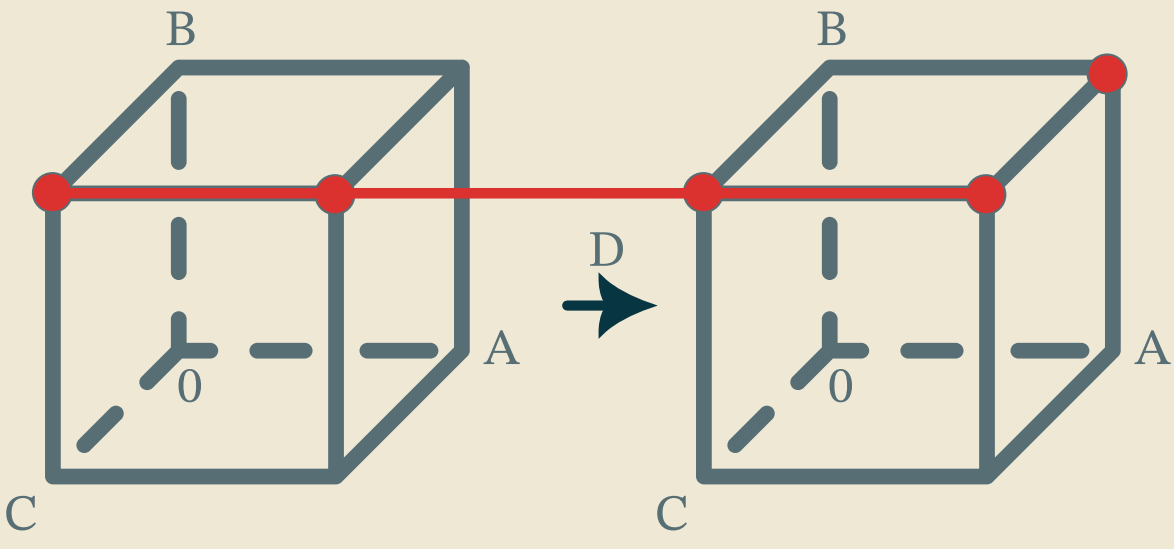
Exemple



2.3.2.3 Avec plus de variables ?

On ajoute une dimmension par variable. C'est plus simple que ça en à l'air car l'espace est booléen. Ajoutez un axe qui part de votre cube et arrive à un autre cube, cet axe sera votre variable suplémentaire. Pour une variable de plus on utilise un axe de plus pour 4 cubes. Puis un cube constitué de 8 cube et ainsi de suite. Bien sûr, cette méthode devient exponentiellement plus difficile avec les variable mais reste une solution réalisable.

Exemple



3 Portes Logiques

Exemple

Les **portes logiques** sont des circuits électroniques de base qui réalisent des opérations logiques sur un ou plusieurs bits d'entrée pour produire un bit de sortie. Elles sont les éléments fondamentaux utilisés dans les circuits numériques pour réaliser des calculs et des prises de décision.

3.1 NOTATION

Les portes logiques sont souvent représentées par des symboles dans les schémas de circuits électroniques. Chaque porte a son symbole unique et une table de vérité associée qui montre toutes les combinaisons possibles d'entrées et la sortie correspondante.

3.1.1 Les Principales Portes Logiques

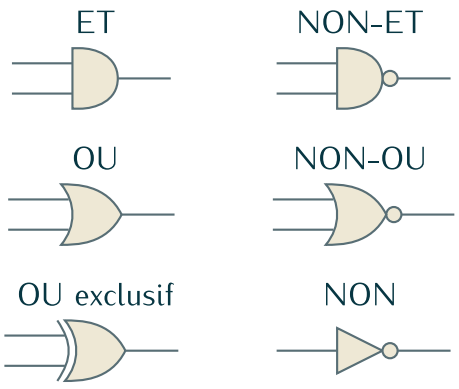


Figure 3.1: Symboles des portes logique

3.2 LES PORTES MÉMOIRE : BISTABLES D ET RS

3.2.1 Bistable RS (Set-Reset)

Le bistable **RS** est un circuit mémoire de base, capable de stocker un bit d'information (0 ou 1). Il est constitué de deux portes **NOR** (ou **NAND**) croisées.

- **Entrées :**
  - **S** (Set) : Met la sortie à 1.
  - **R** (Reset) : Met la sortie à 0.
- **Table de vérité :**

S	R	Q	Description
0	0	Q	Aucune modification
0	1	0	Reset
1	0	1	Set
1	1	?	Indéterminé

3.2.2 Bistable D (Data ou Delay)

Le bistable **D**, ou bascule **D**, est un type de circuit mémoire qui capture la valeur d'entrée à un moment précis, déterminé par un signal d'horloge. Il évite les états indéterminés du bistable **RS**.

- **Entrées :**
  - **D** (Data) : Valeur à stocker.
  - **CLK** (Clock) : Signal d'horloge qui contrôle quand la valeur **D** est capturée. On appelle ↑ un front montant et ↓ un front descendant qui correspondent respectivement au changement d'état de 0 à 1 et 1 à 0.
- **Table de vérité :**

CLK	D	Q	Description
↑	0	0	Capturer la valeur de D = 0
↑	1	1	Capturer la valeur de D = 1
↓	X	Q	Aucune modification

3.3 COMPOSANTS DE BASE

3.3.1 La Mémoire en Portes Logiques

Les circuits de mémoire sont conçus à partir de bistables (**D** ou **RS**) combinés en réseaux complexes pour stocker des bits.

Exemple

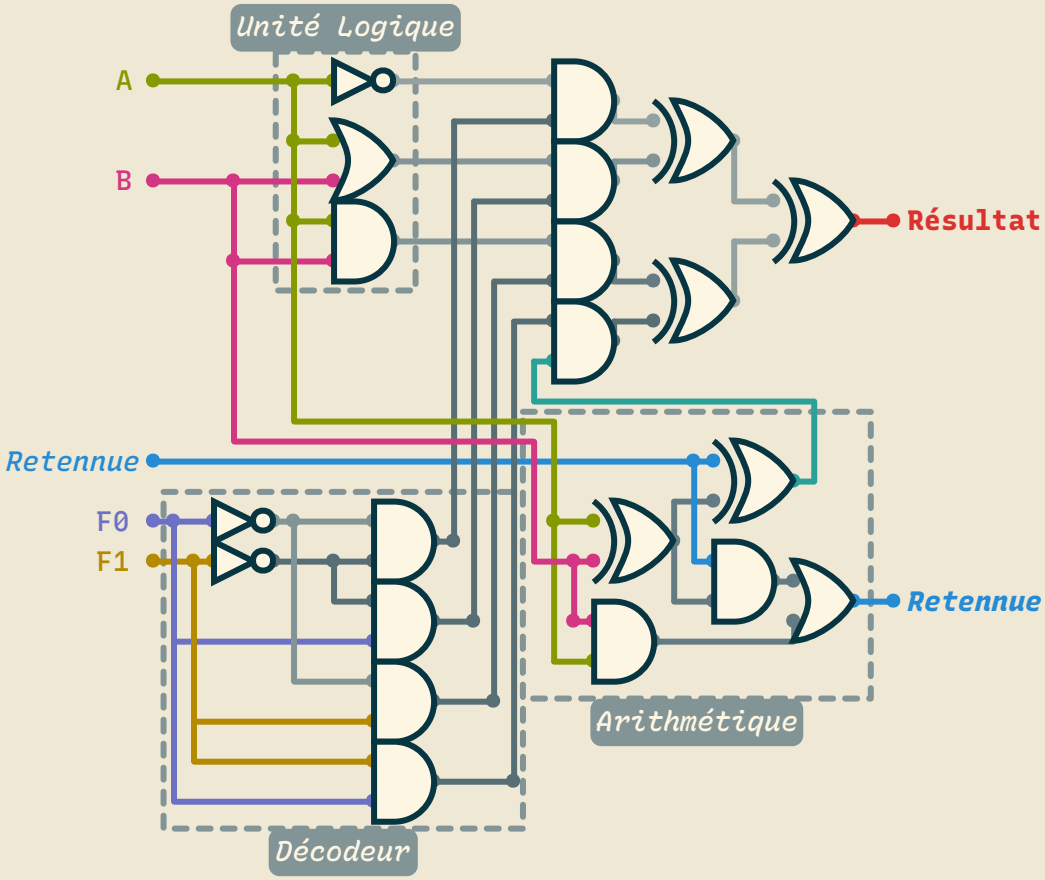
3.3.2 L'Unité Arithmétique et Logique (UAL) en Portes Logiques

L'UAL (Unité Arithmétique et Logique) est construite à partir de nombreuses portes logiques combinées pour effectuer des opérations mathématiques et logiques.

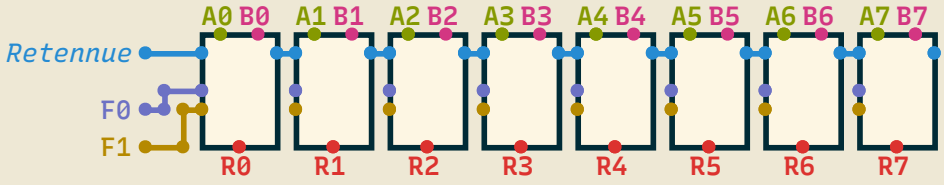
- **Additionneur :** Réalise l'addition binaire en utilisant des portes **XOR** pour chaque bit et des portes **AND/OR** pour gérer les retenues.
- **Comparateur :** Utilise des portes **XOR** pour comparer chaque bit et des portes **AND/OR** pour générer une sortie indiquant l'égalité ou l'inégalité.
- **Multiplexeur/Démultiplexeur :** Utilise des portes **AND**, **OR**, et **NOT** pour sélectionner ou router les signaux.
- **ALU Complexe :** Combinée pour réaliser des opérations telles que l'addition, la soustraction, la multiplication, les opérations logiques, etc. Les portes logiques sont arrangées de manière à produire la sortie correcte selon l'opération sélectionnée.

Exemple

- **Addition de deux bits :** Utilisation de portes XOR pour obtenir la somme et de portes AND/OR pour la retenue.
- **Opérations logiques (ET, OU, NON) :** Directement réalisées par des portes logiques de base.



On peut les cabler entre-eux pour faire des opération sur des nombres d'un octet.



4 Introduction aux Structures et Composants d'un Ordinateur

4.1 QU'EST-CE QU'UN ORDINATEUR ?

Définition 4.1.1: Ordinateur

Un ordinateur est une machine électronique capable de traiter des informations de manière automatique. Il exécute des instructions prédéfinies sous forme de programmes pour effectuer une grande variété de tâches, allant du calcul à la gestion de données en passant par le traitement d'images ou le contrôle de dispositifs.

4.1.1 Composants Principaux

Les composants principaux d'un ordinateur sont :

- **Le matériel (hardware) :** Ensemble des éléments physiques de l'ordinateur (processeur, mémoire, disque dur, etc.).
- **Le logiciel (software) :** Ensemble des programmes et des systèmes d'exploitation qui permettent de faire fonctionner le matériel et d'exécuter des tâches spécifiques.

4.1.2 Fonctionnement Général

L'ordinateur fonctionne en suivant un cycle basique appelé **cycle de traitement des informations :** 1. **Entrée** (Input) : Les données sont introduites via des dispositifs d'entrée (clavier, souris, etc.). 2. **Traitement** (Processing) : Le processeur traite les données selon les instructions du programme. 3. **Stockage** (Storage) : Les données peuvent être sauvegardées pour un usage ultérieur sur des dispositifs de mémoire (disques durs, **SSD**, etc.). 4. **Sortie** (Output) : Les résultats sont renvoyés via des dispositifs de sortie (écran, imprimante, etc.).

4.2 LE PROCESSEUR (CPU)

Définition 4.2.1: Processeur

Le **processeur**, ou **unité centrale de traitement (CPU)**, est le cerveau de l'ordinateur. Il interprète et exécute les instructions des programmes.

4.2.1 Composants du Processeur

- **Unité de Contrôle (Control Unit) :** Coordonne les opérations de l'ordinateur en envoyant des signaux aux autres composants pour exécuter les instructions.
- **Unité Arithmétique et Logique (UAL) :** Effectue toutes les opérations arithmétiques (addition, soustraction, etc.) et logiques (comparaison, AND, OR, etc.).
- **Registres :** Petits espaces de stockage à l'intérieur du processeur utilisés pour stocker temporairement les données pendant le traitement.

4.2.2 Le Cycle d'Instruction

Le CPU suit un cycle d'instruction en trois étapes pour traiter les instructions d'un programme :

1. **Fetch :** Récupérer l'instruction depuis la mémoire.
2. **Decode :** Décoder l'instruction pour déterminer quelles actions doivent être effectuées.
3. **Execute :** Exécuter l'instruction en utilisant les composants appropriés du CPU.

4.3 LA MÉMOIRE

Définition 4.3.1: Mémoire

La **mémoire** de l'ordinateur est un composant essentiel où sont stockées les instructions des programmes ainsi que les données en cours de traitement. Elle permet au processeur d'accéder rapidement aux informations nécessaires à l'exécution des tâches.

4.3.1 Types de Mémoire

- **Mémoire Vive (RAM – Random Access Memory) :** Mémoire volatile qui stocke temporairement les données et les instructions pendant que le programme est en cours d'exécution. Elle se vide lorsque l'ordinateur est éteint.
- **Mémoire Morte (ROM – Read-Only Memory) :** Mémoire non-volatile qui contient les instructions permanentes de l'ordinateur, comme le **BIOS**. Ces données ne sont pas effacées lorsque l'ordinateur est éteint.
- **Stockage Persistant (HDD, SSD) :** Stocke les données de manière permanente. Contrairement à la **RAM**, les données restent présentes même lorsque l'ordinateur est éteint.

La principale différence entre les types de mémoire est leur vitesse et leur capacité. La technologie de base est très similaire.

4.3.2 La Hiérarchie de la Mémoire

La mémoire de l'ordinateur est hiérarchisée en fonction de la rapidité d'accès et de la capacité : - **Registres :** Très rapides mais de petite taille. - **Cache :** Plus lente que les registres, mais plus rapide que la **RAM**. - **RAM :** Plus lente que le cache, mais plus rapide que les disques de stockage. - **Stockage :** **HDD** ou **SSD**, beaucoup plus lents que la **RAM**, mais avec une capacité bien supérieure.

4.4 LES BUS

Définition 4.4.1: Bus

Un bus est un ensemble de fils électriques qui permet de transporter des données, des adresses et des signaux de contrôle entre les différents composants d'un ordinateur (**CPU**, mémoire, périphériques).

4.4.1 Types de Bus

- **Bus de Données :** Transporte les données entre le processeur, la mémoire et les périphériques.
- **Bus d'Adresse :** Transporte les adresses des emplacements mémoire ou des périphériques que le **CPU** doit lire ou écrire.

- **Bus de Contrôle :** Transporte les signaux de commande (comme lecture/écriture) émis par le **CPU** pour contrôler les autres composants.

4.4.2 Importance des Bus

Les bus sont cruciaux pour la communication interne au sein de l'ordinateur. Leur largeur (nombre de bits qu'ils peuvent transporter simultanément) affecte directement les performances du système.

4.5 LES PÉRIPHÉRIQUES D'ENTRÉE/SORTIE (I/O)

4.5.1 Périphériques d'Entrée

Les **périphériques d'entrée** permettent aux utilisateurs de fournir des données et des commandes à l'ordinateur. Exemples : - **Clavier :** Permet de saisir du texte et des commandes. - **Souris :** Dispositif de pointage pour interagir avec l'interface graphique. - **Scanner :** Convertit des documents physiques en données numériques.

4.5.2 Périphériques de Sortie

Les **périphériques de sortie** permettent à l'ordinateur de restituer des informations traitées à l'utilisateur. Exemples : - **Écran :** Affiche les informations sous forme visuelle. - **Imprimante :** Produit une version papier des documents numériques. - **Haut-parleurs :** Restitue les informations sonores.

4.5.3 Périphériques d'Entrée/Sortie

Certains périphériques peuvent fonctionner à la fois en entrée et en sortie : - **Disques durs (HDD/SSD) :** Lisent et écrivent des données. - **Écrans tactiles :** Combinaison d'un écran (sortie) et d'un dispositif tactile (entrée)

5 Architectures des Ordinateurs

5.1 LES MACHINES DE TURING

5.1.1 Contexte et Importance

La **machine de Turing**, introduite par Alan Turing en 1936, est un modèle théorique fondamental qui formalise les concepts de calcul et d'algorithme. Elle est la base de la théorie de la calculabilité en informatique.

5.1.2 Structure de la Machine de Turing

- **Bande infinie :** Sert de mémoire, divisée en cellules contenant des symboles.
- **Tête de lecture/écriture :** Se déplace sur la bande pour lire et écrire des symboles.
- **État de la machine :** La machine change d'état en fonction du symbole lu et des règles définies dans une table de transition.
- **Cycle d'instruction :** La machine lit, écrit, se déplace, et change d'état jusqu'à atteindre un état d'arrêt.

5.1.3 Signification

La machine de Turing est un modèle universel de calcul : elle peut simuler tout autre algorithme. La **thèse de Church-Turing** affirme que toute fonction calculable par un algorithme peut l'être par une machine de Turing. Ce modèle a jeté les bases de l'informatique moderne.

5.2 LES MACHINES À ÉTATS : UNE VUE D'ENSEMBLE

5.2.1 Qu'est-ce qu'une Machine à États ?

Une **machine à états** est un modèle mathématique qui passe d'un état à un autre en fonction d'entrées externes. C'est un

concept clé dans la théorie des automates et les systèmes numériques.

5.2.2 Composants Principaux

- **États :** Représentent les différentes conditions ou configurations dans lesquelles la machine peut se trouver.
- **Transitions :** Définissent les règles pour passer d'un état à un autre en réponse à des entrées spécifiques.
- **État initial et état final :** La machine commence dans un état initial et peut atteindre un état final, où elle s'arrête.

5.2.3 Types de Machines à États

- **Automates finis :** Machines à états simples avec un nombre fini d'états, largement utilisés dans les systèmes numériques pour le contrôle séquentiel.
- **Automates déterministes :** Chaque entrée entraîne une transition vers un état unique.
- **Automates non déterministes :** Une entrée peut entraîner plusieurs transitions possibles.

5.3 APPLICATIONS ET IMPACT

5.3.1 Machines de Turing

Les machines de Turing sont essentielles pour comprendre les limites de ce qui est calculable et pour concevoir des algorithmes et des systèmes informatiques.

5.3.2 Machines à États

Les machines à états sont utilisées dans la conception de circuits numériques, les systèmes embarqués, et les protocoles de communication, où des transitions prévisibles entre états sont essentielles pour le fonctionnement correct du système.

5.4 INTRODUCTION AUX ARCHITECTURES INFORMATIQUES

Les architectures informatiques définissent la structure, le fonctionnement, et l'organisation des composants d'un ordinateur, en particulier la manière dont le processeur (**CPU**), la mémoire, et les périphériques interagissent. Comprendre les différentes architectures est essentiel pour saisir comment les ordinateurs modernes ont évolué et comment ils fonctionnent aujourd'hui.

5.5 L'ARCHITECTURE DE VON NEUMANN

5.5.1 Histoire et Contexte

L'architecture de Von Neumann, proposée par le mathématicien John von Neumann dans les années 1940, est le modèle fondamental qui a défini les ordinateurs modernes. Cette architecture a été décrite dans un rapport en 1945 intitulé "First Draft of a Report on the EDVAC". Ce document jetait les bases d'une nouvelle génération d'ordinateurs.

5.5.2 Principes Clés

L'architecture de Von Neumann repose sur plusieurs concepts clés :

- **Stockage Commun des Données et des Instructions :** Dans cette architecture, les données et les instructions sont stockées dans la même mémoire. Cela signifie que l'ordinateur ne fait pas de distinction entre les instructions d'un programme et les données qu'il traite.
- **Cycle d'Instruction :** L'ordinateur suit un cycle d'instruction où il récupère une instruction depuis la mémoire, la décode, l'exécute, puis passe à l'instruction suivante.
- **Bus Unique :** Un seul bus est utilisé pour accéder à la mémoire, ce qui peut entraîner des goulets d'étranglement lorsque l'accès aux instructions et aux données est simultané. Ce problème est connu sous le nom de **bottleneck de Von Neumann**.

5.5.3 Avantages et Limites

- **Avantages :** Simplicité de conception et flexibilité dans l'exécution des instructions.
- **Limites :** Le goulet d'étranglement causé par le bus unique limite la performance, surtout lorsque des calculs intensifs en données sont nécessaires.

5.6 L'ARCHITECTURE HARVARD

5.6.1 Histoire et Contexte

L'architecture Harvard a été développée en parallèle avec les premières architectures de Von Neumann, mais elle trouve son origine dans le Harvard Mark I, un des premiers ordinateurs électromécaniques. Contrairement à l'architecture de Von Neumann, l'architecture Harvard propose une séparation claire entre les instructions et les données.

5.6.2 Principes Clés

L'architecture Harvard se distingue par :

- **Mémoire Séparée :** Les instructions et les données sont stockées dans des mémoires distinctes, chacune ayant son propre bus. Cela permet un accès simultané aux instructions et aux données, évitant ainsi le goulet d'étranglement de Von Neumann.
- **Efficacité et Performance :** Cette séparation permet une plus grande efficacité, notamment dans les systèmes embarqués et les microcontrôleurs, où la rapidité et l'efficacité sont cruciales.

5.6.3 Avantages et Limites

- **Avantages :** Accès plus rapide aux instructions et aux données, idéal pour des applications spécifiques nécessitant une haute performance.
- **Limites :** Moins flexible que l'architecture de Von Neumann pour les systèmes informatiques généraux, car elle nécessite des mémoires séparées et plus de complexité dans la conception.

5.7 ÉVOLUTION VERS LES ARCHITECTURES MODERNES

5.7.1 Le Développement de l'Architecture x86 (Intel)

L'architecture **x86** a ses racines dans les microprocesseurs développés par Intel dans les années 1970. Le premier processeur x86, le **Intel 8086**, est lancé en 1978 et est rapidement devenu la norme pour les ordinateurs personnels.

- **Compatibilité Ascendante :** L'une des forces de l'architecture x86 est sa compatibilité ascendante, permettant aux logiciels conçus pour d'anciens processeurs de fonctionner sur des processeurs plus récents.
- **Complexité CISC (Complex Instruction Set Computing) :** L'architecture x86 fait partie de la famille des architectures **CISC**, caractérisées par des jeux d'instructions complexes et variés. Cela permet une plus grande flexibilité dans la programmation mais ajoute de la complexité au niveau matériel.
- **Évolution :** Les processeurs x86 ont évolué pour inclure des fonctionnalités avancées comme l'hyper-threading, les extensions 64 bits (x86-64), et la virtualisation. Des fabricants comme **AMD** ont également contribué à l'évolution de cette architecture.

5.7.2 L'Architecture ARM

L'architecture **ARM** (Advanced **RISC** Machine) est née dans les années 1980, développée par Acorn **C**omputers au Royaume-**U**ni. **ARM** a pris une direction différente de x86 en adoptant le paradigme **RISC** (Reduced Instruction **S**et **C**omputing).

- **Simplicité et Efficacité :** **ARM** utilise un jeu d'instructions simplifié, ce qui permet de concevoir des processeurs plus petits, plus économes en énergie, et très performants dans des tâches spécifiques.
- **Adoption Massive :** **ARM** est devenu l'architecture dominante dans les appareils mobiles (smartphones, tablettes) en raison de son efficacité énergétique et de sa puissance adaptée aux besoins des dispositifs embarqués.
- **Évolution :** L'architecture **ARM** a également évolué pour inclure des extensions 64 bits (**ARM**v8-A), la prise en charge du multiprocesseur, et des améliorations en termes de sécurité et de performances.

5.7.3 L'Architecture Xtensa (Tensilica)

L'architecture **Xtensa** a été développée par Tensilica (désormais propriété de **C**adence **D**esign **S**ystems). Elle se distingue par sa flexibilité et sa personnalisation.

- **Architecture Configurable :** Xtensa permet aux concepteurs de processeurs de personnaliser le jeu d'instructions et les fonctionnalités en fonction des besoins spécifiques de l'application. C'est une architecture idéale pour les systèmes sur puce (**S**o**C**) et les applications spécialisées comme l'audio, le traitement du signal, et les réseaux.
- **Approche RISC :** Comme ARM, Xtensa adopte une approche **RISC**, mais avec une flexibilité qui permet de l'adapter à des applications spécifiques.
- **Utilisation :** Xtensa est largement utilisé dans les systèmes embarqués et les produits nécessitant des performances élevées dans des domaines spécifiques comme l'IoT, les communications sans fil, et le traitement multimédia.