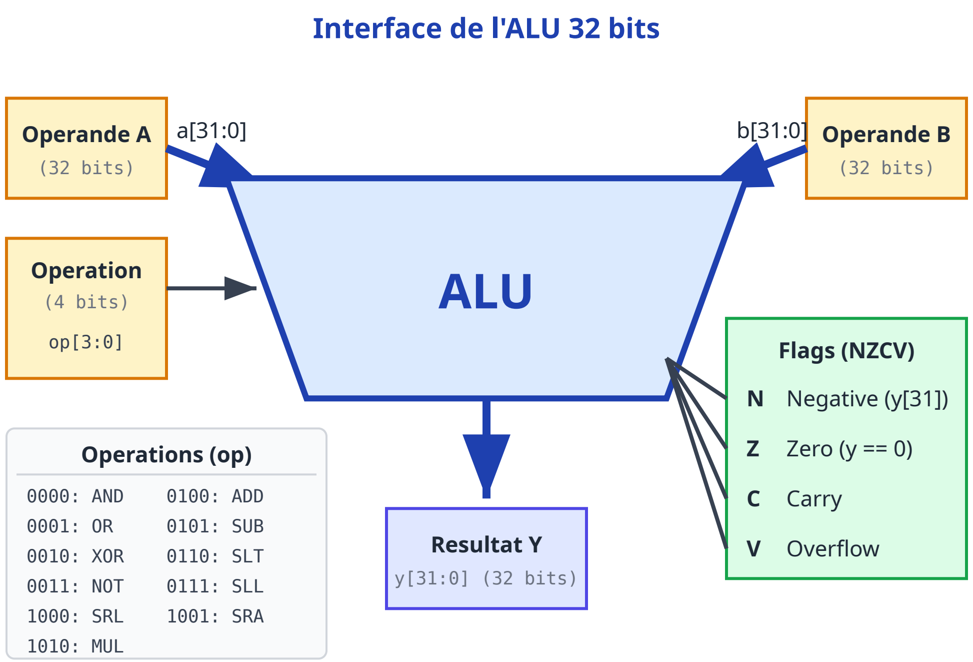
# Chapitre 02 : Arithmétique Binaire

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

# 🎯 Où en sommes-nous ?




*L'ALU — le cœur calculatoire du processeur*

Nous combinons les portes pour construire l'**ALU** !

# Pourquoi l'Arithmétique ?

**Tout est calcul :**

- **Afficher une image** : Calculer la couleur de chaque pixel
- **Jouer un son** : Mélanger des formes d'onde
- **Exécuter un programme** : Calculer l'adresse de la prochaine instruction
- **Traiter du texte** : Comparer des codes ASCII

**L'ALU (Arithmetic Logic Unit)**

Le cœur calculatoire du CPU — effectue toutes les opérations

# Le Système Binaire

### Base 10 (décimal) :

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

### Base 2 (binaire) :

```
Position :   3    2    1    0
Poids    :  2³   2²   2¹   2⁰
Valeur   :   8    4    2    1

1011₂ = 8+0+2+1 = 11₁₀
```

# Conversion Décimal → Binaire

**Méthode des divisions successives par 2 :**

| Division | Quotient | Reste |
|---|---|---|
| 13 ÷ 2 | 6 | **1** |
| 6 ÷ 2 | 3 | **0** |
| 3 ÷ 2 | 1 | **1** |
| 1 ÷ 2 | 0 | **1** |

Lecture de bas en haut : $13_{10}$ = $\mathbf{1101_2}$

# Nombres dans nand2c (32 bits)

| Type | Plage | Exemples |
|---|---|---|
| **Non-signé** | 0 à 4 294 967 295 | Adresses mémoire, compteurs |
| **Signé** | -2 147 483 648 à +2 147 483 647 | Coordonnées, températures |

🔗 **ARM**

Les registres ARM R0-R15 sont aussi sur 32 bits, avec les mêmes plages de valeurs.

# Le Problème des Nombres Négatifs

**Question :** Comment représenter -5 avec seulement des 0 et 1 ?

**Solution naïve : Bit de signe** (0 = positif, 1 = négatif)

**Problèmes :**

- Deux zéros (+0 et -0)
- Circuits différents pour + et -

**Solution brillante**

Le **Complément à 2**

# Complément à 2

**Pour obtenir -X à partir de X :**

1. Inverser tous les bits
2. Ajouter 1

**Exemple (4 bits) : Calculer -5**

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

# Visualisation du Complément à 2 (4 bits)

| Binaire | Non-signé | Signé |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | +1 |
| 0010 | 2 | +2 |
| 0011 | 3 | +3 |
| 0100 | 4 | +4 |
| 0101 | 5 | +5 |
| 0110 | 6 | +6 |

| Binaire | Non-signé | Signé |
|---|---|---|
| 1000 | 8 | **-8** |
| 1001 | 9 | **-7** |
| 1010 | 10 | **-6** |
| 1011 | 11 | **-5** |
| 1100 | 12 | **-4** |
| 1101 | 13 | **-3** |
| 1110 | 14 | **-2** |

# Roue du Complément à 2

```
flowchart LR
    subgraph Positifs
        P0[0000 = 0]
        P1[0001 = +1]
        P2[0010 = +2]
        P7[0111 = +7]
    end
    subgraph Négatifs
        N1[1111 = -1]
        N2[1110 = -2]
        N8[1000 = -8]
    end
    P7 --> N8
    N1 --> P0
```





Le passage de +7 à -8 est le **point de débordement** (overflow).

# Vérification : 5 + (-5) = 0

```
  0101   (5)
+ 1011   (-5)
──────
 10000   → Les 4 bits = 0000 ✓
```

La retenue est ignorée (on travaille sur 4 bits).

**Magie du complément à 2**

L'addition fonctionne identiquement pour les positifs et négatifs !

# Avantages du Complément à 2

1. **Un seul zéro** : `0000` uniquement
2. **Addition universelle** : Même circuit pour +/-
3. **Soustraction = Addition** : A - B = A + NOT(B) + 1

> 💾 **VHDL**
>
> Le type `signed` en VHDL utilise automatiquement le complément à 2.

# L'Addition Binaire

**Règles de base (1 bit) :**

| A | B | Somme | Retenue |
|---|---|---|---|
| 0 | 0 | 0 | **0** |
| 0 | 1 | 1 | **0** |
| 1 | 0 | 1 | **0** |
| 1 | 1 | 0 | **1** |

Comme l'addition décimale, mais en base 2 !

# Exemple : 5 + 3 = 8

```
Retenues :   1 1 1
            ─────
   5      :   0 1 0 1
+  3      : + 0 0 1 1
          ───────────
   8      :   1 0 0 0
```

Colonne par colonne, de droite à gauche.

# Le Demi-Additionneur (Half Adder)

**Entrées :** a, b (1 bit chacun)

**Sorties :** sum (somme), carry (retenue)

| a | b | sum | carry |
|---|---|---|---|
| 0 | 0 | 0 | **0** |
| 0 | 1 | 1 | **0** |
| 1 | 0 | 1 | **0** |
| 1 | 1 | 0 | **1** |

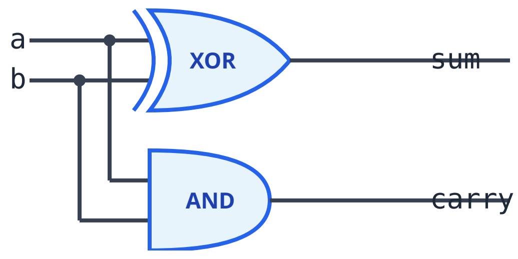



*Schéma du Half Adder*

# Half Adder = XOR + AND

**Observation clé :**

- **sum** = XOR(a, b) — différent = 1
- **carry** = AND(a, b) — les deux à 1

💾 **VHDL**

```
sum   <= a xor b;
carry <= a and b;
```

# L'Additionneur Complet (Full Adder)

**Problème :** Half Adder ne peut pas recevoir de retenue !

**Full Adder :**

- 3 entrées : a, b, cin
- 2 sorties : sum, cout


*Schéma du Full Adder*

# Table de vérité du Full Adder

| a | b | cin | sum | cout |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** |
| 0 | 0 | 1 | 1 | **0** |
| 0 | 1 | 0 | 1 | **0** |
| 0 | 1 | 1 | 0 | **1** |
| 1 | 0 | 0 | 1 | **0** |
| 1 | 0 | 1 | 0 | **1** |
| 1 | 1 | 0 | 0 | **1** |
| 1 | 1 | 1 | 1 | **1** |

# Construction du Full Adder

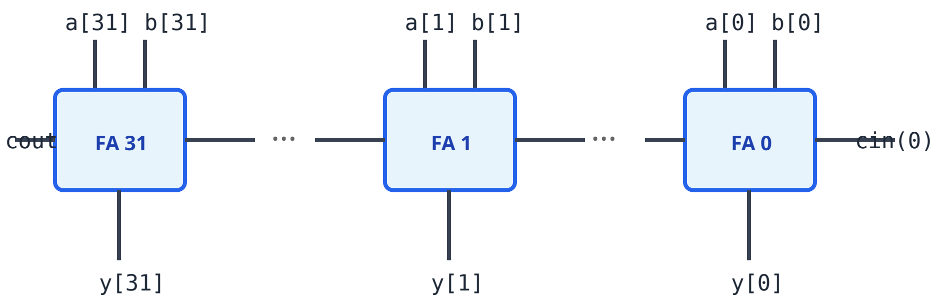
## 2 Half Adders + 1 OR

```
sum  = a XOR b XOR cin
cout = (a AND b) OR ((a XOR b) AND cin)
```

**Astuce de construction**

Premier HA additionne a et b, second HA ajoute cin au résultat.

# Additionneur 32 bits (Ripple Carry)




*32 Full Adders en cascade*

La retenue "ondule" (ripple) à travers tous les additionneurs.

# Délai de propagation

**Limitation du Ripple Carry**

Le délai total = 32 × délai d'un Full Adder

**Solutions avancées :**

- Carry Lookahead Adder (CLA)
- Carry Select Adder

🔗 **ARM**

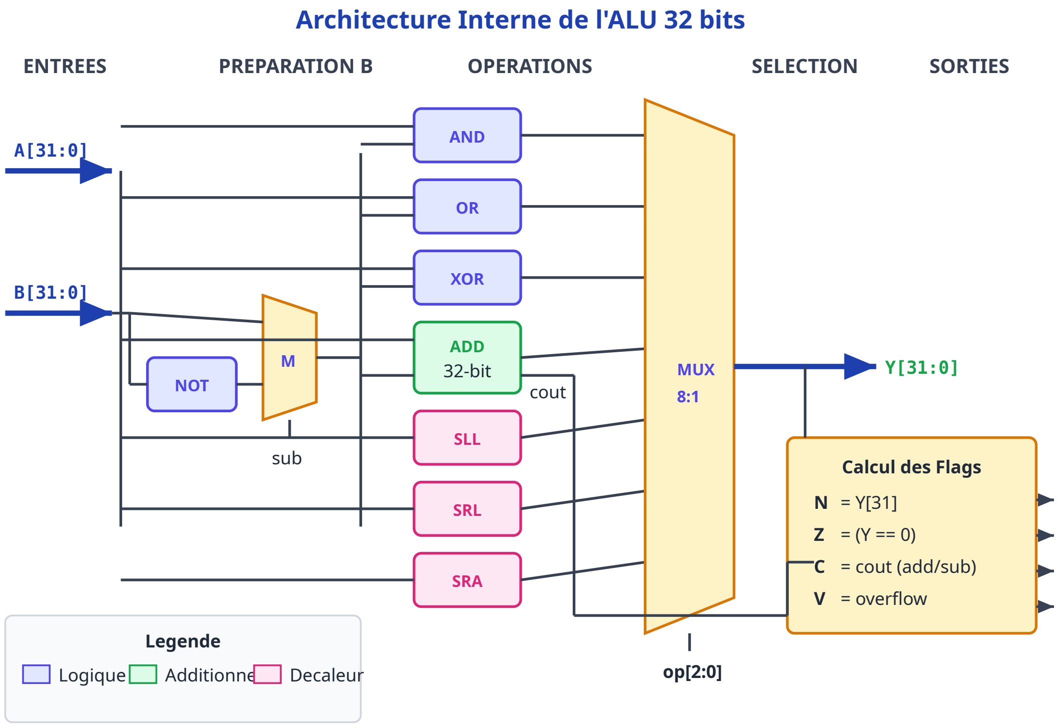
Les processeurs ARM modernes utilisent des additionneurs optimisés avec carry lookahead.

# L'ALU : Le Cœur du CPU

L'**ALU** effectue TOUTES les opérations arithmétiques et logiques.

**Interface :**

- Entrées : a[31:0], b[31:0], op[3:0]
- Sorties : y[31:0], N, Z, C, V




*Vue interne de l'ALU*

# Principe de l'ALU

```
flowchart LR
    A[a] --> AND[AND]
    B[b] --> AND
    A --> OR[OR]
    B --> OR
    A --> ADD[ADD]
    B --> ADD
    A --> XOR[XOR]
    B --> XOR
    AND --> MUX[MUX]
    OR --> MUX
    ADD --> MUX
    XOR --> MUX
    OP[op] --> MUX
    MUX --> Y[y]
```

# Opérations de l'ALU

| op | Nom | Opération | Usage |
| --- | --- | --- | --- |
| 0000 | AND | a & b | Masquage bits |
| 0001 | EOR | a ^ b | Comparaison |
| 0010 | SUB | a - b | Soustraction |
| 0011 | ADD | a + b | Addition |
| 0100 | ORR | a \| b | Combinaison |
| 0101 | MOV | b | Copie |
| 0110 | MVN | ~b | Inversion |

# La Soustraction via Complément à 2

```
A - B = A + (-B) = A + NOT(B) + 1
```

**Implémentation :**

1. Inverser les bits de B (NOT)
2. Additionner avec cin = 1

**Réutilisation**

Même additionneur pour ADD et SUB !

# Les Drapeaux (Flags)

| Flag | Nom | Signification | Calcul |
|---|---|---|---|
| **N** | Negative | Résultat négatif | bit 31 |
| **Z** | Zero | Résultat = 0 | NOR de tous les bits |
| **C** | Carry | Dépassement non-signé | Retenue de l'additionneur |
| **V** | Overflow | Dépassement signé | Logique spéciale |

🔗 **ARM**

Ces flags sont stockés dans le registre CPSR en ARM.

# Calcul du Flag V (Overflow)

**Overflow se produit si :**

- Deux positifs → résultat négatif
- Deux négatifs → résultat positif

**Formule :**

```
V = (a[31] == b[31]) AND (a[31] != y[31])
```

Pour la soustraction (où b est inversé) :

```
V = (a[31] != b[31]) AND (a[31] != y[31])
```

# Exemple : Détection d'overflow

**100 + 50 = 150 (sur 8 bits signés)**

```
  01100100  (+100)
+ 00110010  (+50)
──────────
  10010110  = -106 en signé !
```

**Overflow détecté**

V = 1 car deux positifs donnent un négatif

# Drapeaux et Branchements

| Instruction | Condition | Test | Usage |
|---|---|---|---|
| B.EQ | Equal | $Z = 1$ | Égalité |
| B.NE | Not Equal | $Z = 0$ | Différence |
| B.LT | Less Than | $N \neq V$ | Moins que (signé) |
| B.GE | Greater/Equal | $N = V$ | Plus ou égal (signé) |
| B.LO | Lower | $C = 0$ | Moins que (non-signé) |
| B.HS | Higher/Same | $C = 1$ | Plus ou égal (non-signé) |

# Exemple : CMP et Branchement

```
CMP R0, R1      ; Calcule R0 - R1, met à jour flags
B.EQ egaux      ; Si Z=1, sauter à 'egaux'
B.LT plus_petit ; Si N≠V, sauter à 'plus_petit'
```

**Si R0 = 5, R1 = 5 :**

- R0 - R1 = 0
- Z = 1 → B.EQ pris

**Si R0 = 3, R1 = 5 :**

- R0 - R1 = -2
- N = 1, V = 0, N≠V → B.LT pris

# Exemple Tracé : ADD avec Flags

Calculons `ADD R2, R0, R1` avec R0 = 5, R1 = 3 :

1. **Entrées**

   a = 0000...0101, b = 0000...0011

2. **Addition**

   y = 0000...1000 (8)

3. **Flags**

   N=0, Z=0, C=0, V=0

# Exemple Tracé : SUB avec Overflow

Calculons `SUB` avec a = -100, b = 50 (8 bits) :

```
  10011100  (-100)
- 00110010  (50)    → + 11001101 + 1 = 11001110
 ──────────
  01101010  = +106 ?!
```

**Overflow !** V = 1 car négatif - positif = positif impossible.

# Architecture de l'ALU — Vue Détaillée

```
            a          b
            │          │
       ┌────┴────┐ ┌───┴───┐
       │         │ │ INV?  │  ← si SUB
       │         │ └───┬───┘
       │         │     │
       ▼         ▼     │
  ┌────────┐ ┌────────┐│
  │  AND   │ │  ADD   │◄┘  cin=1 si SUB
  └────┬───┘ └────┬───┘
       │          │
       └────┬─────┘
            │
            ▼
       ┌─────────┐
       │   MUX   │ ◄──── op
```

# Questions de Réflexion

1. Pourquoi le complément à 2 est-il préféré au signe+magnitude ?
2. Que se passe-t-il si on additionne -1 et +1 en complément à 2 ?
3. Comment l'ALU sait-elle si une opération est signée ou non-signée ?
4. Pourquoi le flag C est-il utile pour les comparaisons non-signées ?
5. Comment faire une multiplication avec l'ALU ?

# Du Half Adder à l'ALU

```
CHAPITRE 1              CHAPITRE 2
    ↓                        ↓
  NAND                  Half Adder
    ↓                        ↓
XOR, AND, OR  →    Full Adder  →  Add32  →  ALU
Mux, DMux                   ↓
                     Flags (N,Z,C,V)
```

# Ce qu'il faut retenir

1. **XOR + AND = Half Adder**
2. **2 Half Adders + OR = Full Adder**
3. **32 Full Adders = Additionneur 32-bits**
4. **Complément à 2 = Soustraction avec le même additionneur**
5. **Les Flags (N, Z, C, V) permettent les décisions**
6. **L'ALU calcule tout, le Mux sélectionne**

# Questions ?

📚 **Référence :** Livre Seed, Chapitre 02 - Arithmétique

👉 **Exercices :** TD et TP disponibles

**Prochain chapitre :** Mémoire (DFF, Registres, RAM)