TP Chapitre 02 : Construction de l'ALU

Objectifs pratiques

- Implémenter le Half Adder et Full Adder
- Construire un additionneur multi-bits
- Assembler l'ALU complète
- Générer les drapeaux

Durée estimée : 2h30

Prérequis : TP Chapitre 01 terminé (portes logiques)

Préparation

Accès au Simulateur

👉 **Ouvrir le Simulateur HDL**

Allez dans **HDL Progression → Projet 3 : Arithmétique**

Alternative locale :

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

Exercice 1 : Half Adder

Objectif : Additionner 2 bits sans retenue d'entrée

Spécification

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

Formules

```
sum   = XOR(a, b)
carry = AND(a, b)
```

👉 **Ouvrir l'exercice HalfAdder**

Code à compléter

```
entity HalfAdder is
  port(
    a     : in bit;
    b     : in bit;
    sum   : out bit;
    carry : out bit
  );
end entity;

architecture rtl of HalfAdder is
  component Xor2 port(a, b : in bit; y : out bit); end component;
  component And2 port(a, b : in bit; y : out bit); end component;
begin
  -- TODO: Compléter
end architecture;
```

▸ Solution

Exercice 2 : Full Adder

Objectif : Additionner 2 bits + retenue d'entrée

Spécification

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

Construction

👉 **Ouvrir l'exercice FullAdder**

Code à compléter

```
entity FullAdder is
  port(
    a    : in bit;
    b    : in bit;
    cin  : in bit;
    sum  : out bit;
    cout : out bit
  );
end entity;

architecture rtl of FullAdder is
  component HalfAdder
    port(a, b : in bit; sum, carry : out bit);
  end component;
  component Or2 port(a, b : in bit; y : out bit); end component;
  signal s1, c1, c2 : bit;
begin
  -- TODO: Compléter
end architecture;
```

▸ Solution

Exercice 3 : Additionneur 16 bits

Objectif : Chaîner des Full Adders

Spécification

- Entrées : a[15:0], b[15:0]
- Sorties : y[15:0], cout

Construction en cascade

```
FA0: cin = '0'          → y[0], c0
FA1: cin = c0           → y[1], c1
FA2: cin = c1           → y[2], c2
...
FA15: cin = c14         → y[15], cout
```

👉 **Ouvrir l'exercice Add16**

Code avec generate

```
entity Add16 is
  port(
    a    : in bits(15 downto 0);
    b    : in bits(15 downto 0);
    y    : out bits(15 downto 0);
    cout : out bit
  );
end entity;

architecture rtl of Add16 is
  component FullAdder
    port(a, b, cin : in bit; sum, cout : out bit);
  end component;
  signal c : bits(16 downto 0);
begin
  c(0) <= '0';  -- Pas de retenue initiale

  gen: for i in 0 to 15 generate
    fa: FullAdder port map (
      a => a(i), b => b(i), cin => c(i),
      sum => y(i), cout => c(i+1)
    );
  end generate;

  cout <= c(16);
end architecture:
```

Exercice 4 : Incrémenteur (Inc16)

Objectif : Ajouter 1 à un nombre

Astuce

`Inc(a) = Add(a, 1)` mais on peut simplifier !

```
a + 1 = a + 0...01
```

Le second opérande est toujours 0...01.

👉 **Ouvrir l'exercice Inc16**

▸ Solution simplifiée

Exercice 5 : Soustracteur (Sub16)

Objectif : Utiliser le complément à 2

Rappel

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

Construction

1. Inverser tous les bits de B
2. Additionner avec cin = 1

👉 **Ouvrir l'exercice Sub16**

Code à compléter

```
entity Sub16 is
  port(
    a : in bits(15 downto 0);
    b : in bits(15 downto 0);
    y : out bits(15 downto 0)
  );
end entity;

architecture rtl of Sub16 is
  component Inv port(a : in bit; y : out bit); end component;
  component FullAdder port(a,b,cin : in bit; sum,cout : out bit); end component;
  signal not_b : bits(15 downto 0);
  signal c : bits(16 downto 0);
begin
  -- Étape 1 : Inverser B
  gen_inv: for i in 0 to 15 generate
    inv: Inv port map (a => b(i), y => not_b(i));
  end generate;

  -- Étape 2 : Additionner avec cin = 1
  c(0) <= '1';
  gen_add: for i in 0 to 15 generate
    fa: FullAdder port map (
      a => a(i), b => not_b(i), cin => c(i),
      sum => y(i), cout => c(i+1)
    );
  end generate;
end architecture:
```

Exercice 6 : ALU Complète

Objectif : Assembler toutes les opérations

Interface

```
entity ALU is
  port(
    a  : in bits(31 downto 0);
    b  : in bits(31 downto 0);
    op : in bits(3 downto 0);
    y  : out bits(31 downto 0);
    n  : out bit;  -- Negative
    z  : out bit;  -- Zero
    c  : out bit;  -- Carry
    v  : out bit   -- Overflow
  );
end entity;
```

👉 **Ouvrir l'exercice ALU**

Architecture de l'ALU

Principe : Calculer tous les résultats, puis MUX pour sélectionner

Étapes d'implémentation

1. **Opérations logiques** : AND32, OR32, XOR32, NOT32
2. **Addition/Soustraction** :
    - Inverser B si SUB
    - cin = 1 si SUB
3. **Sélection** : Mux8Way selon `op`
4. **Drapeaux** :
    - N = y(31)
    - Z = NOR de tous les bits de y
    - C = cout de l'additionneur
    - V = calcul d'overflow

Calcul du flag V (Overflow signé)

L'overflow se produit quand :

- Deux positifs donnent un négatif
- Deux négatifs donnent un positif

```
-- Pour l'addition :
-- V = (a[31] == b[31]) AND (a[31] != y[31])

signal same_sign, sign_changed : bit;

u_same: Xnor2 port map (a => a(31), b => b(31), y => same_sign);
u_changed: Xor2 port map (a => a(31), b => y(31), y => sign_changed);
u_v: And2 port map (a => same_sign, b => sign_changed, y => v);
```

Calcul du flag Z (Zero)

Tous les bits du résultat doivent être à 0 :

```
-- Option 1 : Or32Way puis NOT
signal any_one : bit;
u_or: Or32Way port map (a => y, y => any_one);
u_z: Inv port map (a => any_one, y => z);

-- Option 2 : Chaîne de NOR
```

Exercice 7 : Test de l'ALU

Objectif : Vérifier le fonctionnement

Tests à effectuer

a	b	op	Résultat attendu	Flags
5	3	ADD	8	N=0, Z=0
5	5	SUB	0	Z=1
-1	1	ADD	0	Z=1, C=1
0x7FFFFFFF	1	ADD	0x80000000	N=1, V=1

👉 **Tester dans le simulateur**

Exercice Bonus : Multiplicateur 8 bits

Objectif : Comprendre la multiplication matérielle

Principe des produits partiels

```
      a[7:0]
    × b[7:0]
    ────────

pp0 = a AND b[0]     (décalage 0)
pp1 = a AND b[1]     (décalage 1)
...
pp7 = a AND b[7]     (décalage 7)
──────────────────────

y[15:0] = somme des pp
```

Structure

1. **8 And8** pour générer les produits partiels
2. **Addition en arbre** (7 additions)
3. **Résultat sur 16 bits**

👉 **Ouvrir l'exercice Mul8**

Récapitulatif

Composants implémentés

Composant	Complexité	Prérequis
HalfAdder	★	XOR, AND
FullAdder	★★	HalfAdder, OR
Add16	★★	FullAdder
Inc16	★	Add16
Sub16	★★	Add16, NOT
ALU	★★★	Tout !
Mul8	★★★	And8, Add16

Validation Finale

Checklist

- [ ] HalfAdder : tous les tests passent
- [ ] FullAdder : tous les tests passent
- [ ] Add16 : additionne correctement
- [ ] Sub16 : soustrait via complément à 2
- [ ] ALU : toutes les opérations fonctionnent
- [ ] Drapeaux N, Z, C, V calculés correctement

Prochaine étape

➡️ **Chapitre 03 : Mémoire** — Construire des registres et de la RAM !

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