Le Langage HDL

Hardware Description Language

> "Décrire un circuit comme on écrit du code."

Qu'est-ce que le HDL ?

HDL = Hardware Description Language

Un langage pour décrire des circuits :

- Structure (composants, connexions)
- Comportement (logique combinatoire)
- Timing (logique séquentielle)

nand2c HDL

Un sous-ensemble simplifié de VHDL, suffisant pour construire un CPU complet.

Structure d'un Fichier HDL

```
┌──────────────────────────────────────────┐
│  entity ComponentName is                 │  ← Déclaration
│    port(...);                            │    (interface)
│  end entity;                             │
├──────────────────────────────────────────┤
│  architecture rtl of ComponentName is    │  ← Implémentation
│    -- déclarations                       │    (corps)
│  begin                                   │
│    -- assignations                       │
│  end architecture;                       │
└──────────────────────────────────────────┘
```

Exemple Complet Annoté

```
entity And is                                   -- 1
  port(                                         -- 2
    a : in bit;                                 -- 3
    b : in bit;                                 -- 4
    y : out bit                                 -- 5
  );                                            -- 6
end entity;                                     -- 7

architecture rtl of And is                      -- 8
  component Nand                                -- 9
    port(a, b : in bit; y : out bit);           -- 10
  end component;                                -- 11
  signal n : bit;                               -- 12
begin                                           -- 13
  u1: Nand port map (a=>a, b=>b, y=>n);         -- 14
  u2: Nand port map (a=>n, b=>n, y=>y);         -- 15
end architecture;                               -- 16
```

Ligne 1 : `entity And is`

```
entity And is
```

Signification :

- `entity` : mot-clé qui déclare un composant
- `And` : nom du composant (identifiant)
- `is` : début de la définition

Pourquoi ?

- Définit l'**interface publique** du composant
- Le nom doit correspondre au nom du fichier
- Permet à d'autres composants de l'utiliser

> **Convention**
>
> Nom de fichier = `And.hdl` pour l'entité `And`

Lignes 2-6 : `port(...)`

```
port(
  a : in bit;
  b : in bit;
  y : out bit
);
```

Chaque ligne déclare un port :

- **Nom** : `a`, `b`, `y` (identifiant unique)
- **Direction** : `in` (entrée) ou `out` (sortie)
- **Type** : `bit` (un fil) ou `bits(N downto 0)` (bus)

> **Point-virgule**
>
> Sépare les ports, mais **pas** après le dernier !

Ligne 7 : `end entity;`

```
end entity;
```

Signification :

- Ferme le bloc `entity`
- Termine la déclaration d'interface

Pourquoi obligatoire ?

- Délimite clairement la fin de l'interface
- Syntaxe VHDL héritée
- Permet au parseur de vérifier la structure

À ce stade

On a défini QUOI fait le composant (3 ports), mais pas COMMENT.

Ligne 8 : `architecture rtl of And is`

```
architecture rtl of And is
```

Signification :

- `architecture` : bloc d'implémentation
- `rtl` : nom de l'architecture
- `of And` : associée à l'entité `And`
- `is` : début des déclarations

Pourquoi `rtl` ?

- RTL = Register Transfer Level
- Convention pour la logique synthétisable
- Une entité peut avoir plusieurs architectures

Lignes 9-11 : Déclaration de Composant

```
component Nand
  port(a, b : in bit; y : out bit);
end component;
```

Pourquoi déclarer les composants ?

1. **Vérification** : le compilateur connaît l'interface
2. **Documentation** : on voit les dépendances
3. **Typage** : erreurs détectées à la compilation

Syntaxe compacte

`a, b : in bit` déclare deux ports du même type.

Ligne 12 : `signal n : bit;`

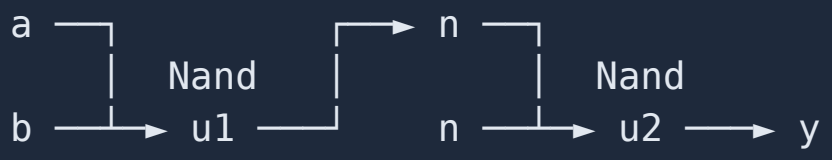
```
signal n : bit;
```

Signification :

- `signal` : fil interne
- `n` : nom du signal
- `bit` : type (1 fil)

Pourquoi un signal ?

- Connecte la sortie de `u1` à l'entrée de `u2`
- N'apparaît pas dans l'interface (interne)
- Permet des calculs intermédiaires



Ligne 13 : `begin`

```
begin
```

Sépare deux zones distinctes :

Avant `begin`	Après `begin`
Déclarations	Assignations
`component`, `signal`	Instanciations, `<=`
Statique	Concurrent

> **Erreur fréquente**
>
> Mettre une instanciation avant `begin` = erreur de syntaxe !

Ligne 14 : Instanciation `u1`

```
u1: Nand port map (a=>a, b=>b, y=>n);
```

Décomposition :

Élément	Signification
`u1`	Label unique (obligatoire)
`Nand`	Composant à instancier
`port map`	Mot-clé de connexion
`a=>a`	Port `a` de Nand ← signal `a`
`y=>n`	Port `y` de Nand → signal `n`

Le Port Map en Détail

```
port map (a => a, b => b, y => n)
         |    |
         |    └── Signal local (côté droit)
         └─── Port du composant (côté gauche)
```

Règle d'or

`port_composant => signal_local`

"La broche X reçoit/envoie le signal Y"

Analogie : Brancher une prise (port) sur une multiprise (signal).

Ligne 15 : Instanciation `u2`

```
u2: Nand port map (a=>n, b=>n, y=>y);
```

Ce qui se passe :

1. Le signal `n` (sortie de `u1`) entre dans `a` ET `b` de `u2`
2. La sortie `y` de `u2` va vers le port `y` de l'entité
3. C'est un `NOT(n)` car `NAND(n,n) = NOT(n)`

> **Résultat**
>
> `And = NAND(NAND(a,b), NAND(a,b)) = NOT(NAND(a,b))`

Ligne 16 : `end architecture;`

```
end architecture;
```

- Ferme le bloc `architecture`
- Termine l'implémentation
- Le composant est complet et utilisable

Récapitulatif

`entity` = QUOI (interface)
`architecture` = COMMENT (implémentation)

Vision Globale du Fichier

```
┌─── INTERFACE ──────────────────────────────────┐
│ entity And is                                  │
│   port(a, b : in bit; y : out bit);            │
│ end entity;                                    │
└────────────────────────────────────────────────┘

┌─── IMPLÉMENTATION ─────────────────────────────┐
│ architecture rtl of And is                     │
│ ┌─── DÉCLARATIONS ──────────────────────┐      │
│ │  component Nand ... end component;    │      │
│ │  signal n : bit;                      │      │
│ └───────────────────────────────────────┘      │
│ begin                                          │
│ ┌─── CONNEXIONS ────────────────────────┐      │
│ │  u1: Nand port map (...);             │      │
│ │  u2: Nand port map (...);             │      │
│ └───────────────────────────────────────┘      │
│ end architecture;                              │
└────────────────────────────────────────────────┘
```

L'Entité : L'Interface

```
entity And is
  port(
    a : in bit;          -- Entrée 1
    b : in bit;          -- Entrée 2
    y : out bit          -- Sortie
  );
end entity;
```

Entité = Boîte noire

Définit les entrées/sorties sans révéler l'implémentation.

Les Types de Ports

Type	Description	Exemple
`bit`	Un seul fil (0 ou 1)	`a : in bit`
`bits(N downto 0)`	Bus de N+1 fils	`data : in bits(31 downto 0)`
`bits(0 to N)`	Bus inversé	`addr : out bits(0 to 7)`

Direction :

- `in` = entrée
- `out` = sortie

Convention

Toujours `downto` pour MSB à gauche.

L'Architecture : Le Corps

```
architecture rtl of And is
  component Nand
    port(a, b : in bit; y : out bit);
  end component;

  signal n : bit;  -- Signal interne
begin
  u1: Nand port map (a => a, b => b, y => n);
  u2: Nand port map (a => n, b => n, y => y);
end architecture;
```

Anatomie de l'Architecture

```
architecture rtl of ComponentName is
   ┌──────────────────────────────────┐
   │  component ...                   │  Zone déclarative
   │  signal ...                      │  (avant begin)
   └──────────────────────────────────┘
begin
   ┌──────────────────────────────────┐
   │  u1: Comp port map (...);        │  Zone des
   │  signal <= expression;           │  assignations
   └──────────────────────────────────┘  (après begin)
end architecture;
```

Déclaration de Composants

```
component Nand
  port(
    a : in bit;
    b : in bit;
    y : out bit
  );
end component;
```

Obligatoire

Chaque composant utilisé doit être déclaré avant `begin`.

Déclaration de Signaux

```
signal temp : bit;                    -- 1 bit
signal data : bits(7 downto 0);       -- 8 bits
signal addr : bits(31 downto 0);      -- 32 bits
```

Signal

Un fil interne qui connecte des composants ou stocke un résultat intermédiaire.

Instanciation de Composants

```
label: ComponentName port map (
  port1 => signal1,
  port2 => signal2,
  port3 => signal3
);
```

Règles :

- Label obligatoire et unique
- Tous les ports mappés
- `=>` pour l'association

Exemple :

```
u_add: FullAdder port map (
  a => x,
  b => y,
  cin => c_in,
  sum => s,
  cout => c_out
);
```

Port Map : L'Ordre Compte !

```
-- CORRECT : broche => signal
u1: Nand port map (a => my_a, b => my_b, y => result);

-- FAUX : signal => broche (inversé)
u1: Nand port map (my_a => a, my_b => b, result => y);
```

Mnémotechnique

`broche => signal` = "où ça va" => "d'où ça vient"

Les Littéraux

Type	Syntaxe	Exemples
Bit unique	`'0'`, `'1'`	`reset <= '0';`
Binaire	`b"..."`	`b"1010"`, `b"0000_1111"`
Hexadécimal	`x"..."`	`x"FF"`, `x"DEAD_BEEF"`
Entier	nombre	`42`, `0x2A`, `0b101`

Lisibilité

Utilisez `_` comme séparateur : `b"1111_0000"`

Assignation Concurrente

```
-- Assignation simple
y <= a and b;

-- Opérations logiques
result <= (a and b) or (c and d);

-- Sélection de bits
low_byte <= data(7 downto 0);
high_bit <= data(31);
```

Règle fondamentale

Un signal ne peut avoir qu'UN SEUL driver (une seule assignation).

Les Opérateurs Logiques

Opérateur	Fonction	Exemple
`and`	ET logique	`y <= a and b;`
`or`	OU logique	`y <= a or b;`
`xor`	OU exclusif	`y <= a xor b;`
`not`	Négation	`y <= not a;`
`nand`	NON-ET	`y <= a nand b;`
`nor`	NON-OU	`y <= a nor b;`

Les Opérateurs Arithmétiques

Opérateur	Fonction	Exemple
`+`	Addition	`sum <= a + b;`
`-`	Soustraction	`diff <= a - b;`
`-a`	Négation	`neg <= -value;`

Complément à 2

Les opérations signées utilisent automatiquement le complément à 2.

Opérateurs de Décalage

Opérateur	Fonction	Exemple
`<<`	Décalage gauche	`doubled <= value << 1;`
`>>`	Décalage droite	`halved <= value >> 1;`

```
signal data : bits(7 downto 0);
signal shifted : bits(7 downto 0);

shifted <= data << 2;  -- Multiplie par 4
shifted <= data >> 3;  -- Divise par 8
```

Opérateurs de Comparaison

Opérateur	Fonction	Exemple
`=`	Égalité	`eq <= a = b;`
`/=` ou `<>`	Différence	`ne <= a /= b;`
`<`	Inférieur	`lt <= a < b;`
`<=`	Inférieur ou égal	`le <= a <= b;`
`>`	Supérieur	`gt <= a > b;`
`>=`	Supérieur ou égal	`ge <= a >= b;`

Le résultat est toujours de type `bit`.

La Concaténation

```
-- Opérateur &
full <= high & low;  -- MSB & LSB

-- Exemples
signal byte : bits(7 downto 0);
signal word : bits(15 downto 0);

word <= x"AB" & byte;           -- 16 bits
addr <= b"0000" & offset;       -- Extension
result <= a(3 downto 0) & b(3 downto 0);  -- Fusion
```

Ordre

`a & b` : `a` est le MSB, `b` est le LSB.

Sélection de Bits

```
signal data : bits(31 downto 0);

-- Signal complet
all_bits <= data;           -- 32 bits

-- Bit unique
bit_5 <= data(5);           -- 1 bit

-- Tranche (slice)
low_byte <= data(7 downto 0);    -- 8 bits
high_word <= data(31 downto 16); -- 16 bits
```

Assignation Conditionnelle

```
-- Syntaxe when-else
y <= a when sel = '1' else b;

-- Conditions multiples
result <= val1 when sel = b"00" else
          val2 when sel = b"01" else
          val3 when sel = b"10" else
          val4;  -- default (others)
```

Usage

Idéal pour les multiplexeurs et la logique conditionnelle simple.

Réplication avec Others

```
signal sel : bit;
signal sel32 : bits(31 downto 0);

-- Répliquer un bit sur 32 bits
sel32 <= (others => sel);

-- Tous à zéro
zeros <= (others => '0');

-- Tous à un
ones <= (others => '1');
```

Logique Séquentielle : Process

```
process(clk)
begin
  if rising_edge(clk) then
    q <= d;  -- Capture sur front montant
  end if;
end process;
```

> **rising_edge(clk)**
>
> Détecte le passage de 0 à 1 du signal d'horloge.

Process avec Reset

```
process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      q <= (others => '0');  -- RAZ synchrone
    else
      q <= d;
    end if;
  end if;
end process;
```

Process avec Load

```
process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      q <= (others => '0');
    elsif load = '1' then
      q <= d;           -- Charge nouvelle valeur
    -- else : q conserve sa valeur
    end if;
  end if;
end process;
```

Case dans un Process

```
process(clk)
begin
  if rising_edge(clk) then
    case state is
      when b"00" => next_state <= b"01";
      when b"01" => next_state <= b"10";
      when b"10" => next_state <= b"00";
      when others => next_state <= b"00";
    end case;
  end if;
end process;
```

Obligatoire

Toujours inclure `when others` pour couvrir tous les cas.

Fonctions Utilitaires

Fonction	Usage	Exemple
`rising_edge(clk)`	Front montant	`if rising_edge(clk)`
`resize(x, N)`	Extension zéro	`resize(byte, 32)`
`sresize(x, N)`	Extension signe	`sresize(signed_val, 32)`

```
signal byte : bits(7 downto 0);
signal word : bits(31 downto 0);

word <= resize(byte, 32);   -- 0x000000XX
word <= sresize(byte, 32);  -- Signe étendu
```

Exemple Complet : Mux 2-to-1

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

architecture rtl of Mux is
begin
  y <= b when sel = '1' else a;
end architecture;
```

Exemple Complet : Registre 8-bit

```
entity Register8 is
  port(
    clk  : in bit;
    load : in bit;
    d    : in bits(7 downto 0);
    q    : out bits(7 downto 0)
  );
end entity;

architecture rtl of Register8 is
begin
  process(clk)
  begin
    if rising_edge(clk) then
      if load = '1' then
        q <= d;
      end if;
    end if;
  end process;
end architecture;
```

Exemple : Additionneur 8-bit

```
entity Add8 is
  port(
    a, b : in bits(7 downto 0);
    sum  : out bits(7 downto 0);
    cout : out bit
  );
end entity;

architecture rtl of Add8 is
  signal result : bits(8 downto 0);  -- 9 bits
begin
  result <= resize(a, 9) + resize(b, 9);
  sum <= result(7 downto 0);
  cout <= result(8);
end architecture;
```

Bonnes Pratiques

Nommage :

- Signaux : `snake_case`
- Entités : `PascalCase`
- Labels : `u_descriptif`

Structure :

- Un composant par fichier
- Nom fichier = nom entité
- Commentaires sur les ports

> **Convention Seed**
>
> `ComponentName.hdl` contient l'entité `ComponentName`.

Erreurs Fréquentes

1. Double driver

```
-- ERREUR !
y <= a;
y <= b;  -- 2e driver
```

2. Port map inversé

```
-- ERREUR !
port map (sig => port);
```

3. Oubli de when others

```
case x is
  when b"00" => ...
  -- Manque others !
end case;
```

4. Confusion bit/bits

```
signal x : bit;
x <= b"01";  -- ERREUR !
```

Ce que HDL ne supporte PAS

- Logique asynchrone
- Instruction `wait`
- `generate` statements
- Generics (`generic`)
- Variables (seulement `signal`)
- Types `record`
- Fonctions utilisateur
- Attributs (`'length`)

Pourquoi ces restrictions ?

Le HDL Seed est conçu pour l'apprentissage, pas pour la production FPGA.

Hiérarchie des Composants Seed

```
Nand (primitive)
  └── Not, And, Or, Xor
        └── Mux, DMux
              └── Add32, ALU32
                    └── Register, RAM
                          └── CPU
```

Du simple au complexe

Chaque niveau n'utilise que les composants du niveau inférieur.

Workflow de Développement

1. **Définir l'interface**
   Écrire l'entité avec les ports
2. **Déclarer les dépendances**
   Lister les composants et signaux
3. **Implémenter la logique**
   Connecter les composants
4. **Tester**
   Simuler avec des vecteurs de test

Commandes CLI

```
# Vérifier la syntaxe
cargo run -p hdl_cli -- check MonComposant.hdl

# Simuler un composant
cargo run -p hdl_cli -- sim MonComposant.hdl

# Générer un schéma
cargo run -p hdl_cli -- diagram MonComposant.hdl
```

Questions de Réflexion

1. Pourquoi un signal ne peut-il avoir qu'un seul driver ?
2. Quelle est la différence entre `<=` et `=>` ?
3. Pourquoi `rising_edge` et pas le niveau haut ?
4. Comment implémenter un compteur 4-bit ?
5. Pourquoi déclarer les composants avant `begin` ?

Ce qu'il faut retenir

1. **Entité** = interface (ports in/out)
2. **Architecture** = implémentation (composants + signaux)
3. **Port map** = connexion avec `broche => signal`
4. **Un driver** par signal (règle fondamentale)
5. **Process** = logique séquentielle (sur front d'horloge)
6. **When-else** = assignation conditionnelle

Questions ?

Référence : `/book/references/hdl_syntax.md`

Bibliothèque : `/hdl_lib/` (73 composants)

Prochain : Construction des portes logiques