Chương 2

NGÔN NGỮ LẬP TRÌNH VHDL

Hình và bảng

I. SỰ RA ĐỜI NGÔN NGỮ VHDL

VHDL (Very high speed integrated circuit Hardware Description Language) là một trong các ngôn ngữ mô tả phần cứng được sử dụng rộng rãi hiện nay. VHDL là ngôn ngữ mô tả phần cứng cho các vi mạch tích hợp có tốc độ cao, được phát triển dùng cho chương trình VHSIC (Very High Speed Integrated Circuit) của bộ quốc phòng Mỹ.

Mục đích của việc nghiên cứu và phát triển là tạo ra một ngôn ngữ mô phỏng phần cứng chuẩn và thống nhất, cho phép thử nghiệm các hệ thống số nhanh hơn, hiệu quả hơn, và nhanh chóng đưa các hệ thống đó vào ứng dụng.

Tháng 7 năm 1983, ba công ty Internetic, IBM, Texas Instruments bắt đầu nghiên cứu. Sau một thời gian, phiên bản đầu tiên của ngôn ngữ VHDL được công bố vào tháng 8 năm 1985.

Vào năm 1986, VHDL được công nhận như một chuẩn IEEE. VHDL đã qua nhiều lần kiểm nghiệm và chỉnh sửa cho đến khi được công nhận như một chuẩn IEEE 1076 vào tháng 12 năm 1987.

VHDL được nghiên cứu phát triển nhằm giải quyết tốc độ phát triển, các thay đổi và xây dựng các hệ thống điện tử số. Với một ngôn ngữ phần cứng tốt thì việc xây dựng các hệ thống điện tử số có tính linh hoạt, phức tạp trở nên dễ dàng hơn. Việc mô tả hệ thống số bằng ngôn ngữ cho phép xem xét, kiểm tra toàn bộ hoạt động của hệ thống trong một mô hình thống nhất.

II. CÁC THUẬT NGỮ CỦA VHDL

Cấu trúc của một chương trình VHDL như sau:

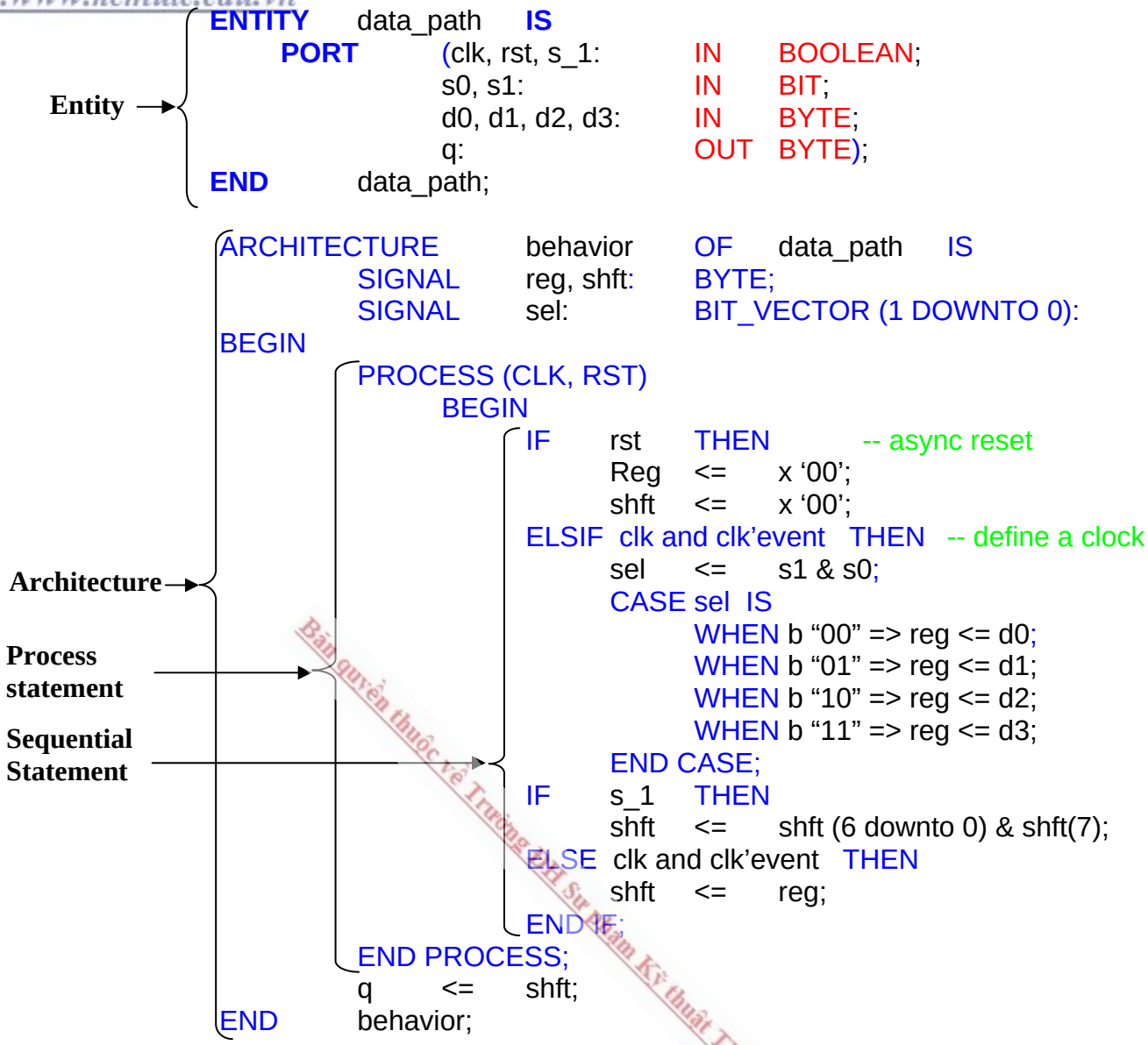
Comment →

```
----------------------------------------------------------------------------------
-- Company:
-- Engineer:
--
-- Create Date:    07:52:37 09/26/2007
-- Design Name:
-- Module Name:    mux - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
```

package →

```
package    typedef        IS
      SUBTYPE    byte    IS      bit_vector (7 downto 0);
END ;
```

Use clause → `USE  work.typedef.all`



Để tìm hiểu chương trình thì chúng ta cần định nghĩa một số thuật ngữ được sử dụng trong ngôn ngữ VHDL.

Entity (thực thể) tất cả các thiết kế đều được biểu diễn ở dạng các thuật ngữ thực thể (entity). Một thực thể là một khối xây dựng cơ bản nhất trong thiết kế. Mức cao nhất của thực thể là mức đỉnh. Nếu thiết kế có thứ bậc thì mô tả ở mức cao nhất sẽ chứa các mô tả ở mức thấp hơn nằm bên trong. Những mô tả ở mức thấp hơn này sẽ chứa các thực thể ở mức thấp hơn nữa. Trong VHDL thì thực thể dùng để khai báo các cổng input_output của các thành phần và tên của nó.

Architecture (kiến trúc) tất cả các thực thể có thể được mô phỏng đều có một mô tả kiến trúc. Kiến trúc mô tả hành vi của thực thể. Một thực thể đơn có thể có nhiều kiến trúc. Một kiến trúc có thể mô tả hành vi (***behavioral description***) trong khi đó một kiến trúc khác có thể mô tả cấu trúc (***structural description***).

Configuration (cấu hình) phát biểu cấu hình được sử dụng để ràng buộc một thể hiện (*instance)* thành phần với một cặp "thực thể - kiến trúc". Một cấu hình có thể được khảo sát giống như một danh sách các thành phần của một thiết kế. Danh sách các thành phần mô tả hành vi để sử dụng cho mỗi thực thể, giống như danh sách liệt kê các phần mô tả sử dụng cho mỗi thành phần trong thiết kế.

Package (gói) một gói là một tập hợp các loại dữ liệu được dùng phổ biến và các chương trình con (subprogram) được sử dụng trong thiết kế. Xem package như là một hộp công cụ chứa nhiều công cụ được dùng để xây dựng các thiết kế.

Driver (nguồn kích) là nguồn kích của một tín hiệu. Nếu một tín hiệu được kích bởi hai nguồn, thì cả hai nguồn đều ở mức tích cực, khi đó ta xem tín hiệu có 2 driver.

Bus (nhóm tín hiệu) thuật ngữ "bus" xem một nhóm các tín hiệu hoặc một phương pháp truyền thông đặc biệt được sử dụng trong thiết kế phần cứng. Trong VHDL, bus là loại tín hiệu đặc biệt có nhiều nguồn kích ở trạng thái tắt.

Attribute (thuộc tính) là dữ liệu được gắn cho các đối tượng VHDL hoặc dữ liệu đã định nghĩa trước liên quan đến các đối tượng VHDL. Ví dụ là khả năng kích dòng của một mạch đệm hoặc nhiệt độ hoạt động cực đại của linh kiện.

Generic là thuật ngữ của VHDL dùng cho một thông số, thông số này chuyển thông tin đến một thực thể. Thí dụ, nếu một thực thể là một mô hình cổng có trì hoãn cạnh lên và trì hoãn cạnh xuống, các giá trị của các trì hoãn lên và xuống có thể được chuyển vào trong thực thể bằng các dùng generic.

Process (quá trình) quá trình là một đơn vị thực thi cơ bản trong VHDL. Tất cả các hoạt động – được thực hiện trong mô phỏng của một mô tả VHDL – thì được chia ra thành một hoặc nhiều quá trình xử lý.

III. MÔ TẢ PHẦN CỨNG TRONG VHDL

Các mô tả VHDL chứa nhiều **đơn vị thiết kế sơ cấp** và nhiều đơn **vị thiết kế thứ cấp**.

- Đơn vị thiết kế sơ cấp là thực thể (**Entity**) và gói (**Package**).
- Đơn vị thiết kế thứ cấp là cấu hình (**Configuration**) và thân gói (**Package Body**).

Các đơn vị thiết kế thứ cấp thì luôn có mối liên hệ với đơn vị thiết kế sơ cấp. Các thư viện chứa nhiều các đơn vị thiết kế sơ cấp và thứ cấp.

1. ENTITY (THỰC THỂ)

Entity dùng để khai báo tên của thực thể, các port của thực thể và các thông tin liên quan đến thực thể. Tất cả các thiết kế được xây dựng dùng một hoặc nhiều thực thể.

Ví dụ 2_1: Khai báo đơn giản về thực thể:

```
ENTITY    mux   IS
  PORT    (a, b, c, d:   IN   BIT;
           s0, s1:       IN   BIT;
           x:            OUT  BIT);
END   mux;
```

Từ khoá **ENTITY** báo cho biết bắt đầu một phát biểu thực thể.

Trong các mô tả được trình bày trong toàn bộ tài liệu, các từ khoá của ngôn ngữ và các loại dữ liệu được cung cấp cho gói chuẩn (**STANDARD**) thì được trình bày ở dạng chữ hoa. Ví dụ: trong ví dụ đã trình bày thì các từ khoá là **ENTITY, IS, PORT, IN, INOUT**, ... Loại dữ liệu chuẩn là **BIT**. Tên của các đối tượng do người dùng định nghĩa ví dụ như **mux** trong ví dụ trên là ở dạng chữ thường.

Tên của thực thể là **mux**. Thực thể có 7 port trong câu lệnh khai báo **PORT** – 6 port cho kiểu **IN** và 1 port cho kiểu **OUT**. 4 port dữ liệu ngõ vào (**a, b, c, d**) là dạng **BIT**. Hai ngõ vào lựa chọn mạch đa hợp (**s0, s1**) cũng thuộc kiểu dữ liệu **BIT**. Ngõ ra cũng là **BIT**.

Thực thể mô tả giao tiếp với thế giới bên ngoài. Thực thể chỉ định rõ bao nhiêu port, hướng tín hiệu của port và loại dữ liệu của port.

2. ARCHITECTURE (KIẾN TRÚC)

Thực thể mô tả giao tiếp với mô hình VHDL.

Kiến trúc mô tả chức năng cơ bản của thực thể và chứa nhiều phát biểu mô phỏng hành vi của thực thể. Kiến trúc luôn luôn có liên quan đến thực thể và các mô tả hành vi của thực thể.

Một kiến trúc của bộ đa hợp ở trên có dạng như sau:

```
ARCHITECTURE    dataflow    OF    mux    IS
SIGNAL          select:     INTEGER;
BEGIN
  Select <=  0  WHEN s0 = '0' AND  s1= '0' ELSE
             1  WHEN s0 = '1' AND  s1= '0' ELSE
             2  WHEN s0 = '0' AND  s1= '1' ELSE
             3;
  x      <=  a  AFTER 0.5 NS  WHEN  select = 0 ELSE
             b  AFTER 0.5 NS  WHEN  select = 1 ELSE
             c  AFTER 0.5 NS  WHEN  select = 2 ELSE
             d  AFTER 0.5 NS  ;
END     dataflow;
```

Từ khoá **ARCHITECTURE** cho biết phát biểu này mô tả kiến trúc cho một thực thể. Tên của kiến trúc là **dataflow**. Kiến trúc của thực thể đang được mô tả được gọi là **mux**.

Lý do cho kết nối giữa thực thể và kiến trúc là một thực thể có thể có nhiều kiến trúc mô tả hành vi của thực thể. Ví dụ một kiến trúc có thể là một mô tả hành vi và một kiến trúc khác có thể là mô tả cấu trúc.

Vùng ký tự nằm giữa từ khoá **ARCHITECURE** và từ khoá **BEGIN** là nơi khai báo các phần tử và các tín hiệu logic cục bộ để sau này dùng. Trong ví dụ trên biến tín hiệu **select** được khai báo là tín hiệu cục bộ.

Vùng chứa các phát biểu của kiến trúc bắt đầu với từ khoá **BEGIN**. Tất cả các phát biểu nằm giữa các câu lệnh **BEGIN** và **END** được gọi là các phát biểu đồng thời bởi vì tất cả các phát biểu được thực hiện cùng một lúc.

a. Gán Các Tín Hiệu Đồng Thời

Trong ngôn ngữ lập trình thông thường như C hoặc C++ thì mỗi phát biểu gán thực hiện một lần sau một phát biểu gán khác và theo một thứ tự được chỉ định. Thứ tự thực hiện được xác định bởi thứ tự của các phát biểu trong file chương trình nguồn.

Trong kiến trúc VHDL thì không có thứ tự chỉ định nào cho các phát biểu gán. Thứ tự thực hiện được chỉ định rõ bởi sự kiện xảy ra trên tín hiệu mà phát biểu gán hướng đến.

Khảo sát phát biểu gán đầu tiên được trình bày như sau:

```
Select <=  0  WHEN s0 = '0' AND  s1= '0' ELSE
           1  WHEN s0 = '1' AND  s1= '0' ELSE
           2  WHEN s0 = '0' AND  s1= '1' ELSE
           3;
```

Gán tín hiệu được thực hiện bằng kí hiệu **<=**. Tín hiệu **select** sẽ được gán giá trị dựa vào giá trị của **s0** và **s1**. Phát biểu gán này được thực hiện bất kỳ lúc nào khi một hoặc hai tín hiệu **s0** và **s1** có thay đổi.

Một phát biểu gán tín hiệu được xem là **nhạy** với các thay đổi trên bất kỳ tín hiệu nào nằm bên phải của kí hiệu gán **<=**. Phát biểu gán tín hiệu của ví dụ trên thì nhạy với **s0** và **s1**. Phát biểu gán tín hiệu khác trong kiến trúc **dataflow** nhạy với tín hiệu lựa chọn.

Chúng ta sẽ khảo sát cách hai phát biểu ở trên hoạt động thực sự ra sao. Giả sử rằng chúng ta có điều kiện ổn định khi **s0** và **s1** đều có giá trị là 0 và các tín hiệu hiện hành **a, b, c** và **d** đều có giá trị là 0. Tín hiệu **x** sẽ có giá trị là 0 vì nó được gán cho giá trị của tín hiệu **a**.

Bây giờ giả sử: chúng ta tạo ra một sự kiện thay đổi trên tín hiệu **a** từ giá trị 0 lên 1.

Khi sự kiện tín hiệu a xảy ra thì phát biểu gán đầu tiên không được thực hiện bởi vì phát biểu này không nhạy với sự thay đổi của tín hiệu **a** vì tín hiệu **a** không nằm bên phải của toán tử.

Phát biểu gán thứ 2 sẽ được thực hiện bởi vì nó nhạy với sự kiện xảy ra trên tín hiệu **a**. Khi phát biểu gán thứ 2 được thực hiện thì giá trị mới của **a** sẽ được gán cho tín hiệu **x**. Ngõ ra **x** bây giờ sẽ thay đổi sang 1.

Tiếp theo chúng ta sẽ khảo sát trường hợp khi tín hiệu **s0** thay đổi. Giả sử cho **s0** và **s1** đều ở mức 0 và các port **a, b, c** và **d** có giá trị theo thứ tự là 0, 1, 0 và 1. Cho tín hiệu **S0** thay đổi giá trị từ 0 lên 1.

Phát biểu gán tín hiệu đầu tiên nhạy với **s0** nên nó sẽ được thực hiện.

Khi các phát biểu đồng thời thực hiện, việc tính toán giá trị biểu thức sẽ dùng giá trị hiện hành cho tất cả các tín hiệu chứa trong phát biểu.

Khi phát biểu đầu tiên thực hiện sẽ tính giá trị mới để được gán cho **select** từ giá trị hiện hành của biểu thức tín hiệu nằm bên phải của kí hiệu gán <=. Việc tính toán giá trị biểu thức sẽ dùng giá trị hiện hành cho tất cả các tín hiệu chứa trong phát biểu.

Với giá trị của **s0** bằng 1 và **s1** bằng 0 thì tín hiệu **select** sẽ nhận giá trị mới là 1. Giá trị mới của tín hiệu **select** được xem như sự kiện xảy ra trên tín hiệu **select**, làm phát biểu gán thứ 2 cũng được thực hiện theo. Phát biểu gán thứ 2 sẽ dùng giá trị mới của tín hiệu **select** để gán giá trị của port **b** cho ngõ ra **x** và **x** sẽ thay đổi giá trị từ 0 lên 1.

b. Thời gian trễ

Việc gán tín hiệu cho tín hiệu **x** không xảy ra ngay lập tức. Mỗi một giá trị được gán cho tín hiệu **x** đều chứa phát biểu **AFTER**. Giá trị của **x** trong các phát biểu gán ở trên chỉ được nhận giá trị sau khoảng thời gian 0,5 ns.

c. Đồng bộ lệnh

Phát biểu gán đầu tiên chỉ được thực hiện khi các sự kiện xảy ra ở các port **s0** và **s1**. Phát biểu gán tín hiệu thứ 2 sẽ không thực hiện trừ khi sự kiện xảy ra trên tín hiệu **select** hoặc sự kiện xảy ra trên các tín hiệu **a, b, c, d**.

Hai phát biểu gán tín hiệu trong kiến trúc **behave** hình thành mô hình hành vi (**behavioral model**), hoặc kiến trúc cho thực thể **mux**.

Kiến trúc **dataflow** thì không có cấu trúc.

3. CÁC THIẾT KẾ CÓ CẤU TRÚC (STRUCTURAL DESIGNS)

Một cách khác để viết thiết kế **mux** là xây dựng các thành phần phụ mà chúng thực hiện các hoạt động nhỏ hơn của mô hình đầy đủ. Với mô hình đơn giản nhất của mạch đa hợp 4 ngõ vào như chúng ta đã dùng là mô tả ở cấp độ cổng đơn giản.

Kiến trúc được trình bày sau đây là mô tả cấu trúc của thực thể **mux**.

```
ARCHITECTURE    netlist OF    mux    IS
COMPONENT       andgate
        PORT(a, b, c: IN BIT;       x: OUT BIT);
END COMPONENT;
COMPONENT       inverter
        PORT(in1: IN BIT;           x: OUT BIT);
END COMPONENT;
COMPONENT       orgate
        PORT(a, b, c, d: IN BIT; x: OUT BIT);
END COMPONENT;
SIGNAL          s0_inv, s1_inv, x1, x2, x3, x4: BIT;
BEGIN
U1:     inverter (s0, s0_inv);
U2:     inverter (s1, s1_inv);
U3:     andgate (a, s0_inv, s1_inv, x1);
U4:     andgate (b, s0, s1_inv, x2);
U5:     andgate (c, s0_inv, s1, x3);
U6:     andgate (d, s0, s1, x4);
U7:     orgate (x2 => b, x1 => a, x4 => d, x3 => c, x => x);
END     netlist;
```


Mô tả này sử dụng một số các thành phần mức thấp hơn để mô hình hoá hành vi của thiết bị **mux**. Có một thành phần cổng đảo **inverter**, một thành phần cổng **andgate**, và một thành phần **orgate**. Một trong các thành phần này được khai báo trong phần khai báo kiến trúc – nằm giữa câu lệnh kiến trúc và **BEGIN**.

Một số các tín hiệu được dùng để kết nối một trong các thành phần để thành lập mô tả kiến trúc. Các loại tín hiệu này được khai báo dùng khai báo **SIGNAL**.

Vùng chứa phát biểu kiến trúc được thiết lập tại vị trí ngay sau từ khoá **BEGIN**. Trong ví dụ này có một số phát biểu của các thành phần. Các thành phần này được đặt tên là **U1÷U7**.

Phát biểu **U1** là phát biểu cho cổng đảo. Phát biểu này nối port **s0** với port ngõ vào của thành phần cổng đảo và tín hiệu **s0_inv** với port ngõ ra của thành phần cổng đảo.

Kết quả là port **in1** của cổng đảo thì được nối tới port **s0** của thực thể **mux** và port **x** của cổng đảo được nối tới tín hiệu cục bộ **s0_inv**. Trong phát biểu này thì các port được nối tới theo thứ tự mà chúng xuất hiện trong phát biểu.

Chú ý phát biểu thành phần U7 – phát biểu này dùng các kí hiệu như sau:

```
U7:     orgate (x2 => b, x1 => a, x4 => d, x3 => c, x => x);
```

Phát biểu này kết hợp các tên để tương thích với các port. Ví dụ port **x2** của cổng **orgate** thì được nối tới port **b** của thực thể của phát biểu kết hợp đầu tiên. Sự kết hợp và thứ tự có thể không theo thứ tự nhưng không nên thực hiện.

4. HOẠT ĐỘNG TUẦN TỰ (SEQUENTIAL BEHAVIOR)

Có một cách khác để mô tả chức năng của thiết bị **mux** trong ngôn ngữ VHDL. Thực ra ngôn ngữ VHDL có nhiều cách trình bày cho chức năng với kết quả tương tự. Cách thứ 3 để mô tả chức năng của **mux** là sử dụng phát biểu quá trình (process) để mô tả chức năng trình bày theo thuật toán. Cách này được dùng cho kiến trúc **sequential** như sau:

```
ARCHITECTURE    sequential   OF    mux   IS
  PROCESS (a, b, c, d, s0, s1)
  VARIABLE   sel: INTEGER;
  BEGIN
        IF        s0 = '0' and s1 = '0' THEN  sel:= 0 ;
        ELSIF     s0 = '1' and s1 = '0' THEN  sel:= 1 ;
        ELSIF     s0 = '0' and s1 = '1' THEN  sel:= 2 ;
        ELSE      sel:= 3 ;
        END  IF;

        CASE   sel  IS
         WHEN  0 =>          x <= a ;
         WHEN  1 =>          x <= b ;
         WHEN  2 =>          x <= c ;
         WHEN  OTHERS  => x <= d ;
        END CASE;
  END PROCESS;
END sequential;
```


Kiến trúc này chỉ chứa 1 phát biểu duy nhất được gọi là phát biểu quá trình (process). Được bắt đầu với hàng có từ khoá **PROCESS** và kết thúc với hàng có từ khoá **END PROCESS**. Tất cả các phát biểu nằm giữa hai hàng trên được xem thành phần của phát biểu quá trình.

a. Các phát biểu quá trình

Phát biểu quá trình chứa nhiều thành phần.

- Thành phần thứ nhất được gọi là danh sách các phần tử nhạy.
- Thành phần thứ hai được gọi là thành phần khai báo quá trình.
- Thành phần thứ 3 là các phát biểu.

Trong ví dụ trên thì danh sách liệt kê các tín hiệu nằm trong dấu ngoặc sau từ khoá **PROCESS** được gọi là danh sách nhạy. Danh sách này liệt kê chính xác những tín hiệu làm cho phát biểu quá trình được thực hiện. Trong ví dụ này thì danh sách chứa các tín hiệu là **a, b, c, d, s0** và **s1**. Chỉ có những sự kiện xảy ra trên các tín hiệu này làm cho phát biểu quá trình được thực hiện.

b. Vùng khai báo quá trình

Phần khai báo quá trình là vùng nằm giữa: sau danh sách nhạy và từ khoá **BEGIN**. Trong ví dụ trên phần khai báo chứa khai báo biến cục bộ **sel**. Biến này là biến cục bộ chỉ được dùng để tính toán giá trị dựa vào port **s0** và **s1**.

c. Thành phần phát biểu quá trình

Thành phần phát biểu quá trình bắt đầu với từ khoá **BEGIN** và kết thúc với hàng có từ khoá **END PROCESS**. Tất cả các phát biểu nằm trong vùng quá trình là những phát biểu tuần tự. Điều này có nghĩa là phát biểu này được thực hiện xong thì câu lệnh tiếp theo sẽ được thực hiện giống như một ngôn ngữ lập trình bình thường. ***Chú ý:*** thứ tự các phát biểu trong kiến trúc (***architecture***) thì không cần vì chúng thực hiện đồng thời, tuy nhiên trong quá trình (***process***) thì cần phải theo thứ tự – thứ tự thực hiện trong quá trình là thứ tự các phát biểu.

d. Thực hiện quá trình

Chúng ta xem cách hoạt động quá trình bằng cách phân tích hoạt động của ví dụ **sequential** trong kiến trúc theo từng hàng phát biểu. Để phù hợp chúng ta giả sử **s0** thay đổi về 0. Bởi vì **s0** nằm trong danh sách nhạy của phát biểu quá trình.

Mỗi phát biểu trong quá trình được thực hiện theo trình tự. Trong ví dụ trên, phát biểu **if** được thực hiện đầu tiên và tiếp theo là phát biểu **case**.

Kiểm tra thứ nhất xem **s0** có bằng 0 hay không. Phát biểu này sẽ không thực hiện nếu **s0** bằng 1 và **s1** bằng 0. Phát biểu gán tín hiệu theo sau phát biểu kiểm tra thứ nhất sẽ không được thực hiện. Thay vào đó phát biểu kế được thực hiện. Phát biểu này kiểm tra đúng trạng thái và phát biểu gán theo sau lệnh kiểm tra **s0** =1 và s1 = 0 được thực hiện. Giá trị **sel**:=1.

e. Các phát biểu tuần tự

Phát biểu sẽ thực hiện tuần tự. Khi một phát biểu kiểm tra thoả điều kiện thì phát biểu được thực hiện thành công và các bước kiểm tra khác sẽ không được thực hiện. Phát biểu **IF** đã thực hiện xong và bây giờ thì đến phát biểu **case** sẽ được thực hiện.

Phát biểu **case** sẽ căn cứ vào giá trị của biến **sel** đã được tính toán trước đó ở câu lệnh **IF** và thực hiện đúng phát biểu gán tương ứng với giá trị của biến **sel**. Trong trường hợp này thì giá trị của **sel = 1** nên câu lệnh gán **x<= b** sẽ được thực hiện.

Giá trị của port **b** sẽ được gán cho port **x** và quá trình thực hiện sẽ chấm dứt bởi vì không còn phát biểu nào trong kiến trúc.

5. LỰA CHỌN KIẾN TRÚC

Cho đến bây giờ 3 kiến trúc đã được dùng để mô tả cho một thực thể. Kiến trúc nào sẽ được dùng để xây dựng mô hình cho thực thể thì tuỳ thuộc vào độ chính xác mong muốn và nếu thông tin về cấu trúc được yêu cầu.

Nếu mô hình sẽ được dùng để điều khiển công cụ layout thì kiến trúc **netlist** là lựa chọn hợp lý.

Nếu mô hình cấu trúc không được yêu cầu vì nhiều lý do thì mô hình hiệu suất cao được sử dụng. Một trong 2 phương pháp còn lại (kiến trúc **dataflow** và **sequential**) thì có thể đạt hiệu suất cao hơn về yêu cầu không gian bộ nhớ và tốc độ thực hiện.

Cách để lựa chọn giữa 2 phương pháp này có thể làm nảy sinh một câu khỏi về kiểu lập trình. Người xây dựng mô hình thích viết chương trình VHDL theo kiểu đồng thời hay theo kiểu trình tự?

Nếu người xây dựng mô hình muốn viết mã VHDL kiểu đồng thời thì phải chọn kiểu kiến trúc **dataflow**, ngược lại thì chọn kiểu kiến trúc **sequential**. Thường thì người xây dựng mô hình quen với kiểu lập trình tuần tự nhưng kiểu đồng thời là những công cụ mạnh để viết cho các mô hình nhỏ hiệu suất cao.

6. CÁC PHÁT BIỂU CẤU HÌNH

Một thực thể có thể có nhiều hơn một kiến trúc nhưng làm thế nào để người xây dựng mô hình chọn kiến trúc nào để sử dụng trong mô phỏng đã cho. Phát biểu cấu hình sắp xếp các thuyết minh thành phần cho thực thể. Với khả năng mạnh của cấu hình người xây dựng mô hình có thể được lựa chọn dùng xây dựng mô hình cho thực thể tại mỗi cấp độ trong thiết kế.

Chúng ta sẽ xem xét phát biểu cấu hình dùng kiến trúc liệt kê của thực thể **mux**.

Ví dụ 2-2: Phát biểu hình như sau:

```
CONFIGURATION   muxcon1     OF     mux   IS
FOR     netlist
        FOR   U1, U2: Inverter USE ENTITY WORK.myinv (version1);
        END FOR;

        FOR   U3, U4, U5, U6: andgate  USE ENTITY WORK.myand (version1);
        END FOR;

        FOR   U7: orgate  USE ENTITY WORK.myor (version1);
        END FOR;

END muxcon1;
```


Chức năng của phát biểu cấu hình là diễn tả chính xác kiến trúc nào dùng cho mỗi thành phần trong mô hình. Điều này xảy ra ở kiểu hệ thống có cấp bậc. Thực thể có cấp bậc cao nhất trong thiết kế cần có kiến trúc để sử dụng cho các chỉ định cũng như bất kỳ thành phần nào cần được thuyết minh trong thiết kế.

Bắt đầu phát biểu cấu hình là tên của cấu hình **muxcon1** cho thực thể **mux**. Sử dụng kiến trúc **netlist** như là kiến trúc cho thực thể cấp cao nhất đó là **mux**.

Đối với 2 thành phần **U1** và **U2** của cổng đảo **inverter** được thuyết minh cho kiến trúc **netlist**, sử dụng thực thể **myinv**, kiến trúc **version1** từ thư viện được gọi là **WORK**.

Đối với các thành phần **U3 ÷ U6** của and **andgate**, dùng thực thể **myand**, kiến trúc **version1** từ thư viện **WORK**.

Đối với thành phần **U7** của cổng **orgate** sử dụng thực thể **myor**, kiến trúc **version1** từ thư viện **WORK**.

Tất cả các thực thể bây giờ đều có các kiến trúc được chỉ định cho chúng. Thực thể mux có kiến trúc **netlist** và các thành phần khác có kiến trúc được đặt tên chỉ định **version1**.

SỨC MẠNH CỦA CẤU HÌNH

Khi biên dịch các thực thể, các kiến trúc và cấu hình đã chỉ định trước thì có thể xây dựng mô hình có thể mô phỏng. Nhưng điều gì sẽ xảy ra nếu không muốn mô phỏng ở cấp độ cổng? Và nếu muốn dùng kiến trúc BEHAVE thay thế. Sức mạnh của cấu hình cho phép bạn không cần biên dịch lại toàn bộ thiết kế mà chỉ cần biên dịch lại cấu hình mới.

Ví dụ 2-3: Cho cấu hình như sau:

```
CONFIGURATION muxcon2 OF mux IS
FOR dataflow
END FOR;
END muxcon2;
```

Cấu hình này có tên là **muxcon2** cho thực thể **mux**. Sử dụng kiến trúc **dataflow** cho thực thể cấp cao nhất là **mux**. Khi biên dịch cấu hình này thì kiến trúc **dataflow** được lựa chọn cho thực thể **mux** trong mô phỏng.

Cấu hình này không cần thiết trong VHDL chuẩn nhưng cung cấp cho người thiết kế sự tự do để chỉ định chính xác kiến trúc nào sẽ được dùng cho thực thể. Kiến trúc mặc nhiên được dùng cho thực thể là kiến trúc sau cùng được biên dịch cho vào thư việc làm việc.

7. TÓM TẮT

Trong phần này đã giới thiệu cơ bản về VHDL và cách sử dụng ngôn ngữ để xây dựng mô hình hành vi của thiết bị và thiết kế. Ví dụ thứ nhất đã trình bày cách xây dựng mô hình **dataflow** đơn giản trong VHDL được chỉ rõ. Ví dụ thứ 2 trình bày cách một thiết kế lớn có thể được thực hiện từ những thiết kế nhỏ hơn – trong trường hợp này bộ đa hợp 4 ngõ vào đã được xây dựng dùng các cổng **AND, OR,** và **INVERTER**. Ví dụ này cung cấp tổng quan cấu trúc của VHDL.

IV. GIỚI THIỆU VỀ MÔ HÌNH HÀNH VI

Phát biểu gán tín hiệu là dạng cơ bản nhất của mô hình hành vi trong VHDL.

Ví dụ 2-4: Phát biểu gán tín hiệu như sau:

```
a <= b;
```

Phát biểu này được đọc như sau: "**a** có giá trị của **b**". Kết quả của phát biểu gán này là giá trị hiện tại của **b** được gán cho tín hiệu **a**. Phát biểu gán này được thực hiện bất kỳ lúc nào tín hiệu b thay đổi giá trị. Tín hiệu **b** nằm trong danh sách nhạy của câu lệnh này. Bất kỳ tín hiệu nào trong danh sách nhạy của phát biểu gán tín hiệu thay đổi giá trị thì phát biểu gán tín hiệu được thực hiện.

Nếu kết quả thực hiện có giá trị mới khác với giá trị trước đó thì sau một thời gian trễ thì tín hiệu sẽ xuất hiện tại tín hiệu đích.

Nếu kết quả thực hiện có cùng giá trị thì sẽ không có thời gian trễ tín hiệu nhưng sự chuyển trạng thái vẫn được tạo ra. Sự chuyển trạng thái luôn được tạo ra khi mô hình được đánh giá nhưng chỉ những tín hiệu có giá trị thay đổi mới có sự kiện trễ.

Phát biểu sau sẽ giới thiệu về phát biểu gán tín hiệu sau một thời gian trễ:

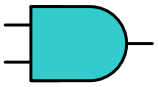
```
a <= b AFTER 10 ns;
```

Phát biểu này được đọc là "a có giá trị của b sau thời gian trễ 10 ns"

Cả hai phát biểu đều là các phát biểu gán tín hiệu đồng thời, và nhạy với sự thay đổi giá trị của tín hiệu b. Khi b thay đổi giá trị thì các phát biểu gán thực hiện và giá trị mới được gán cho giá trị của tín hiệu a.

Ví dụ 2-5: Phát biểu gán tín hiệu đồng thời cho mô hình cổng AND như sau:

```
ENTITY      and2  IS
  PORT      (a, b:      IN    BIT;
             c:         OUT   BIT);
END   and2;
ARCHITECTURE    and2_behav OF   and2  IS
BEGIN
  c <= a AND b AFTER 5 ns;
```



Hình 2-1. Cổng A có 2 ngõ vào.

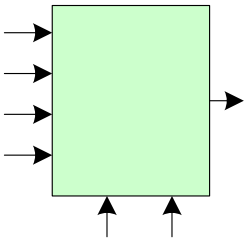
Cổng AND có 2 ngõ vào **a, b** và một ngõ ra **c** như hình 2-1. Giá trị của tín hiệu **c** có thể được gán cho giá trị mới khi một hoặc cả hai tín hiệu **a** và **b** thay đổi giá trị.

Đơn vị thiết kế thực thể mô tả các port của cổng **and2**: có 2 ngõ vào **a** và **b**, một ngõ ra **c**. Kiến trúc **and2_behav** cho thực thể **and2** chứa một phát biểu gán tín hiệu đồng thời. Phát biểu gán này nhạy với cả 2 tín hiệu **a** và tín hiệu **b**.

Giá trị của biểu thức **a** and với **b** được tính toán trước, kết quả tính toán được đưa đến ngõ ra sau khoảng thời gian trễ 5 ns từ lúc tính toán xong.

Ví dụ 2-6: trình bày phát biểu gán tín hiệu phức tạp hơn nhiều và minh họa cho khái niệm đồng thời một cách chi tiết hơn.

Hình 2-2 trình bày sơ đồ khối của bộ đa hợp 4 ngõ vào và mô hình hành vi cho mux như sau:



Hình 2-2. Kí hiệu của mux có 4 ngõ vào.

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      mux4  IS
  PORT      (I0, I1, I2, I3, a, b:   IN    STD_LOGIC;
             q:                      OUT   STD_LOGIC);
END         mux4;
```

```
ARCHITECTURE    mux4 OF    mux4 IS
SIGNAL          sel:    INTEGER;
BEGIN
WITH    sel     SELECT
  q     <=  I0        AFTER  10 ns  WHEN  0
            I1        AFTER  10 ns  WHEN  1
            I2        AFTER  10 ns  WHEN  2
            I3        AFTER  10 ns  WHEN  3
            'X'       AFTER  10 ns  WHEN  OTHERS;

  sel   <=    0       WHEN   a  = '0' AND  b = '0'  ELSE
              1       WHEN   a  = '1' AND  b = '0'  ELSE
              2       WHEN   a  = '0' AND  b = '1'  ELSE
              3       WHEN   a  = '1' AND  b = '1'  ELSE
              4       ;
END     mux4;
```

Thực thể entity cho mô hình này có 6 port ngõ vào và 1 port ngõ ra. 4 port ngõ vào (**I0, I1, I2, I3**) tượng trưng cho các tín hiệu sẽ được gán cho tín hiệu ngõ ra **q**. Chỉ 1 trong các tín hiệu được gán cho tín hiệu ngõ ra **q** dựa vào kết quả của 2 tín hiệu ngõ vào khác là **a** và **b**. Bảng sự thật cho bộ đa hợp được trình bày như hình 2-3.

Để ứng dụng chức năng đã mô tả ở trên, chúng ta dùng phát biểu gán tín hiệu điều kiện và phát biểu gán tín hiệu có lựa chọn.

Phát biểu thứ 2 trong ví dụ được gọi là phát biểu gán tín hiệu có điều kiện. Phát biểu này gán giá trị cho tín hiệu đích dựa vào các điều kiện được đánh giá cho mỗi lệnh. Các điều kiện **WHEN** được thực hiện một lần tại một thời điểm theo thứ tự tuần tự cho đến khi gặp điều kiện tương thích. Phát biểu thứ 2 gán giá trị cho tín hiệu đích khi tương thích điều kiện, tín hiệu đích là **sel**. Tuỳ thuộc vào giá trị của a và b thì các giá trị từ 0 đến 4 sẽ được gán cho **sel**.

Nếu có nhiều điều kiện của một phát biểu tương thích thì phát biểu đầu tiên mà nó tương thích sẽ được thực hiện và các giá trị của phát biểu tương thích còn lại sẽ bị bỏ qua.

Phát biểu thứ 1 được gọi là gán tín hiệu có lựa chọn và lựa chọn giữa số lượng các tuỳ chọn để gán giá trị đúng cho tín hiệu đích – trong ví dụ này tín hiệu đích là **q**.

B	A	Q
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Hình 2-3. Bảng trạng thái của mux có 4 ngõ vào.

Biểu thức (giá trị **sel** trong ví dụ này) được đánh giá và phát biểu mà nó tương thích với giá trị của biểu thức được gán giá trị cho tín hiệu đích. Tất cả các giá trị có thể có của biểu thức phải có sự lựa chọn tương thích trong cách gán tín hiệu đã lựa chọn.

Mỗi một tín hiệu ngõ vào có thể được gán cho ngõ ra **q** – tuỳ thuộc vào các giá trị của 2 ngõ vào **a** và **b**. Nếu các giá trị của **a** và **b** không xác định thì giá trị sau cùng '**X**' (không xác định) được gán cho ngõ ra **q**. Trong ví dụ này, khi các ngõ vào lựa chọn ở giá trị không xác định thì ngõ ra được gán cho giá trị không xác định.

Phát biểu thứ hai nhạy với các tín hiệu **a** và **b**. Bất kỳ lúc nào khi **a** hoặc **b** thay đổi giá trị, phát biểu gán thứ hai được thực hiện và tín hiệu **sel** được cập nhật. Phát biểu thứ 1 nhạy với tín hiệu **sel**. Khi tín hiệu **sel** thay đổi giá trị thì gán tín hiệu tín hiệu thứ nhất được thực hiện.

Nếu ví dụ này được xử lý bởi công cụ tổng hợp, thì kết quả cấu trúc cổng được xây dựng giống như một bộ đa hợp 4 đường sang 1 đường. Nếu thư viện tổng hợp chứa bộ đa hợp 4 đường sang 1 đường thì bộ đa hợp này có thể được cấp phát dựa vào sự phức tạp của công cụ tổng hợp và đặt vào trong thiết kế.

1. DELAY QUÁN TÍNH VÀ DELAY TRUYỀN

Trong VHDL có 2 loại delay có thể dùng cho mô hình hành vi. Delay quán tính thì được sử dụng phổ biến, trong khi delay truyền được sử dụng ở những nơi mà mô hình delay dây dẫn được yêu cầu.

a. Delay quán tính:

Delay quán tính là mặc nhiên trong VHDL. Nếu không có kiểu delay được chỉ định thì delay quán tính được sử dụng. Delay quán tính là delay mặc nhiên bởi vì trong hầu hết các trường hợp thì nó thực hiện giống như thiết bị thực.

Giá trị của delay quán tính bằng với delay trong thiết bị.

Nếu bất kỳ xung tín hiệu có chu kỳ với thời gian của tín hiệu ngắn hơn thời gian delay của thiết bị thì giá trị tín hiệu ngõ ra không thay đổi.

Nếu thời gian của tín hiệu được duy trì ở một giá trị đặc biệt dài hơn thời gian delay của thiết bị thì delay quán tính sẽ được khắc phục và thiết bị sẽ thay đổi sang trạng thái mới.

Hình 2-4 là một ví dụ về kí hiệu bộ đệm rất đơn giản – có 1 ngõ vào **A** và một ngõ ra **B**, dạng sóng được trình bày cho tín hiệu ngõ vào **A** và ngõ ra **B**.

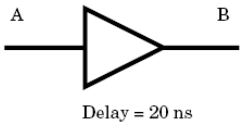
Tín hiệu A thay đổi từ '0' sang '1' tại mốc thời gian **10ns** và từ '1' sang '0' tại mốc thời gian **20ns**. Với các khoảng thời gian này cho phép xây dựng một xung hoặc xung nhọn có thời gian nhỏ hơn **10ns**. Cho bộ đệm có thời gian trễ là **20ns**.

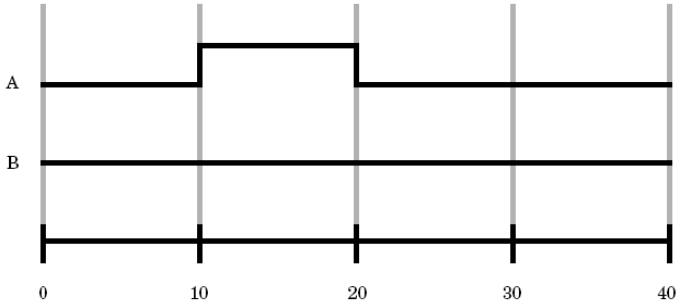
Chuyển trạng thái từ '0' sang '1' trên tín hiệu **A** làm cho mô hình bộ đệm được thực hiện và theo dự kiến thì giá trị '1' xuất hiện ở ngõ ra **B** tại mốc thời gian **30ns**.

Ở mốc thời gian **20ns**, sự kiện tiếp theo trên tín hiệu **A** xảy ra (tín hiệu **a** xuống mức '0') thì mô hình bộ đệm dự kiến một sự kiện mới sẽ xảy ra trên ngõ ra **B** có giá trị 0 tại mốc thời gian **40ns**. Trong khi đó sự kiện đã dự kiến ở ngõ ra **B** cho mốc thời gian **30ns** vẫn chưa xảy ra. Sự kiện mới được dự đoán bởi mô hình bộ đệm xung đột với sự kiện trước và trình mô phỏng ưu tiên cho sự kiện có mốc thời gian **30ns**.

Kết quả của việc ưu tiên là xung bị nuốt (mất). Lý do xung bị nuốt là tuỳ thuộc vào mô hình delay quán tính, sự kiện thứ nhất tại mốc thời gian **30ns** chưa có đủ thời gian để hoàn thành delay quán tính của tín hiệu ngõ ra.

Mô hình thời gian delay quán tính thường được sử dụng trong tất cả các trình mô phỏng. Trong hầu hết các trường hợp mô hình delay quán tính đủ chính xác cho các yêu cầu của người thiết kế. Một trong những lý do cho việc mở rộng sử dụng thời gian delay quán tính là nó ngăn chặn thời gian trì hoãn của xung xuyên qua thiết bị.





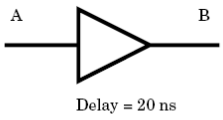
Hình 2-4. Dạng sóng có delay quán tính của bộ đệm.

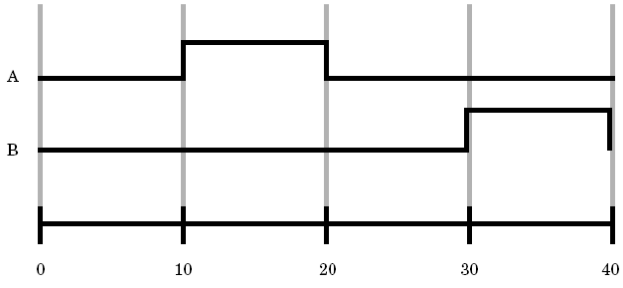
b. Delay truyền tín hiệu

Delay truyền tín hiệu không phải là delay mặc nhiên của VHDL mà phải được chỉ định. Delay truyền tượng trưng cho delay dây dẫn, bất kỳ xung nào được truyền đến ngõ ra đều được delay với một giá trị delay theo chỉ định. Delay truyền rất có ích để xây dựng mô hình thiết bị có trể trên đường dây, trên dây dẫn của bo mạch.

Nếu chúng ta xem mạch đệm đã được trình bày ở hình 2-4 nhưng thay thế các dạng sóng delay quán tính bằng các dạng sóng delay truyền thì chúng ta có kết quả như hình 2-5. Cùng dạng sóng của ngõ vào nhưng dạng sóng ngõ ra **B** thì hoàn toàn khác. Với delay truyền thì các xung nhọn sẽ xuất hiện nhưng các sự kiện xếp theo thứ tự trước khi truyền đi.

Tại mốc thời gian **10ns**, mô hình bộ đệm được thực hiện và dự kiến một sự kiện ngõ ra sẽ lên mức '1' tại mốc thời gian **30ns**. Tại mốc thời gian **20ns** mô hình bộ đệm bị kích và dự đoán một giá trị mới sẽ xuất hiện ở ngõ ra tại mốc thời gian **40ns**. Với thuật toán delay truyền thì các sự kiện được đặt theo thứ tự. Sự kiện cho mốc thời gian **40ns** được đặt trong danh sách các sự kiện nằm sau sự kiện của mốc thời gian **30ns**. Xung không bị nuốt nhưng được truyền nguyên vẹn sau thời gian delay của thiết bị.





Hình 2-5. Dạng sóng có delay truyền của bộ đệm.

c. Mô hình Delay quán tính

Mô hình tiếp theo trình bày cách viết một mô hình delay quán tính. Giống như những mô hình khác mà chúng ta đã khảo sát, delay mặc nhiên là delay quán tính do đó không cần thiết phải chỉ định kiểu delay là delay quán tính:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      buf    IS
  PORT      (a:    IN     STD_LOGIC;
             b:    OUT    STD_LOGIC);
END         buf;

ARCHITECTURE    buf    OF    buf    IS
BEGIN
        b       <= a  AFTER  20 ns ;
END         buf;
```

d. Mô hình Delay truyền

Ví dụ cho mô hình delay truyền được trình bày như sau:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      delay_line    IS
  PORT      (a:    IN     STD_LOGIC;
             b:    OUT    STD_LOGIC);
END         delay_line;

ARCHITECTURE    delay_line    OF    delay_line    IS
BEGIN
        b       <= TRANSPORT   a   AFTER  20 ns ;
END         delay_line;
```

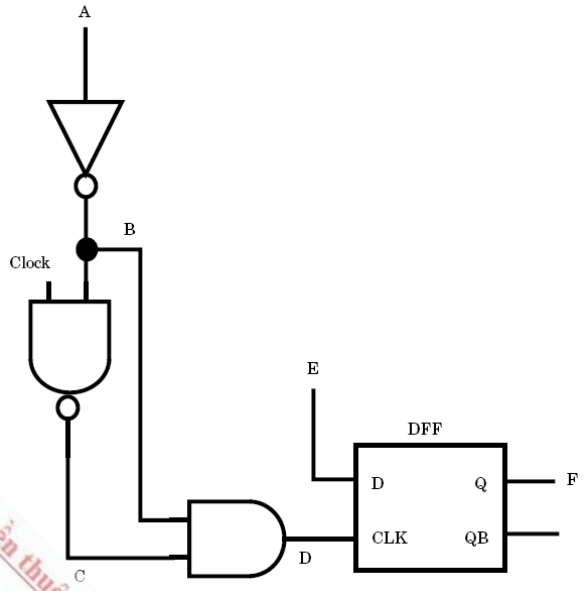

Giống như mô hình delay quán tính chỉ khác là có thêm từ khoá **TRANSPORT** trong lệnh gán tín hiệu cho tín hiệu **b**. Khi từ khoá này tồn tại, kiểu delay được dùng trong phát biểu là delay truyền.

2. MÔ PHỎNG DELTA

Mô phỏng delta được dùng để xếp thứ tự cho nhiều loại sự kiện trong mô phỏng thời gian. Đặc biệt các sự kiện delay bằng 0 phải được xếp theo thứ tự để tạo ra các kết quả thích hợp. Nếu các sự kiện delay zero không theo thứ tự hợp lý thì kết quả có thể khác nhau giữa các lần mô phỏng khác nhau. Một ví dụ cho kiểu này được trình bày ở hình 2-6. Mạch điện này là một phần của sơ đồ mạch đồng hồ.

Mạch điện gồm có một cổng đảo, một cổng NAND và một cổng AND thúc ngõ vào đồng hồ của thành phần flip flop. Cổng NAND và cổng AND được dùng để gác ngõ vào xung clock đến flip flop.

Chúng ta sẽ khảo sát hoạt động của mạch dùng cơ cấu delay delta và cơ cấu khác. Bằng cách kiểm tra 2 cơ cấu delay chúng ta sẽ hiểu rõ hơn cách delay delta sắp xếp các sự kiện.



Hình 2-6. So sánh 2 cơ cấu đánh giá.

Để dùng delay delta thì tất cả các thành phần của mạch điện phải có delay zero theo chỉ định. Delay cho cả 3 cổng được chỉ định là zero. (các mạch điện thực tế thì không có đặc tính như thế).

Khi có cạnh xuống xảy ra trên tín hiệu **A**, ngõ ra của cổng đảo thay đổi đúng thời điểm 0. Chúng ta giả sử rằng sự kiện xảy ra tại mốc thời gian 10ns. Tín hiệu ra **B** của cổng đảo thay đổi ngược với giá trị mới của ngõ vào.

Khi giá trị **B** thay đổi, cả hai cổng **AND** và **NAND** được đánh giá lại. Trong ví dụ này giả sử ngõ vào xung **clock** là 1. Nếu cổng **NAND** được đánh giá trước thì giá trị mới ở ngõ ra cổng **NAND** là '0'.

Khi cổng **AND** được đánh giá thì tín hiệu **B** có giá trị là '1' và tín hiệu **C** có giá trị là '0', do đó cổng **AND** dự đoán kết quả mới là '0'.

Nhưng điều gì sẽ xảy ra nếu cổng AND đánh giá trước. Ở cổng **AND** sẽ có giá trị '1' ở tín hiệu **B** và giá trị '1' ở tín hiệu **C** (cổng **NAND** chưa đánh giá). Ngõ ra cổng **AND** có giá trị mới là '1'.

Bây giờ cổng **NAND** mới được đánh giá và giá trị mới ở ngõ ra là '0'. Sự thay đổi ở ngõ ra **NAND** làm cho cổng **AND** đánh giá lại lần nữa. Cổng **AND** có giá trị của B là '1' và giá trị mới của tín hiệu C là '0'. Ngõ ra cổng **AND** bây giờ sẽ có giá trị là '0'. Quá trình này được tóm tắt như hình 2-7.

Cả hai tín hiệu đến ngõ vào **D**. Tuy nhiên khi cổng **AND** được đánh giá trước thì có xung cạnh lên với độ rộng delta xuất hiện ở ngõ ra **D**. Cạnh lên này có thể kích flip flop, tuỳ thuộc vào cách mô hình flip flop được xây dựng.

AND FIRST	NAND FIRST
Evaluate inverter	Evaluate inverter
B <= 1	B <= 1
Evaluate AND (C = 1)	Evaluate NAND
D <= 1	C <= 0
Evaluate NAND	Evaluate AND
C <= 0	D <= 0
Evaluate AND	
D <= 0	

Hình 2-7. So sánh 2 cơ cấu đánh giá.

Trọng tâm của vấn đề là nếu không có cơ cấu đồng bộ delta thì kết quả của mô phỏng có thể tuỳ thuộc vào cách các cấu trúc dữ liệu mô phỏng được xây dựng. Ví dụ, khi biên dịch mạch điện lần thứ nhất thì có thể cổng AND được đánh giá trước, nếu biên dịch lại lần nữa thì có thể cổng NAND được đánh giá trước – cho ra kết quả không mong muốn, mô phỏng delta sẽ ngăn chặn hành vi này xảy ra.

Cùng một mạch điện được đánh giá dùng cơ cấu delay delta VHDL sẽ đánh giá như hình 2-8.

Time	Delta	
10 ns	(1)	A <= 0 Evaluate inverter
	(2)	B <= 1 Evaluate AND Evaluate NAND
	(3)	D <= 1 C <= 0 Evaluate AND
	(4)	D <= 0
11 ns		

Hình 2-8. Cơ cấu đánh giá delay delta.

Ở điểm thời gian delta đầu tiên của **10ns**, tín hiệu **A** nhận giá trị '0'. Giá trị này làm cho cổng đảo được đánh giá lại với giá trị mới. Tín hiệu ngõ ra cổng đảo **B** có giá trị là '1'. Giá trị này không truyền ngay lập tức mà chờ cho đến điểm thời gian delta thứ 2.

Sau đó trình mô phỏng bắt đầu thực hiện điểm thời gian delta thứ 2. Tín hiệu B được cập nhật giá trị là '1' và cổng **AND** và cổng **NAND** được đánh giá lại. Cả hai cổng **AND** và **NAND** phải chờ các giá trị mới ở điểm thời gian delta thứ 3.

Khi điểm thời gian delta thứ 3 xảy ra, tín hiệu **D** nhận giá trị là '1' và tín hiệu **C** nhận giá trị là '0'. Do tín hiệu **C** cũng thúc cổng **AND**, cổng **AND** được đánh giá lại và chờ kết quả ngõ ra ở điểm thời gian delta thứ 4. Cuối cùng ngõ ra **D** bằng '0'.

Tóm lại mô phỏng delta là lượng thời gian vô cùng nhỏ được dùng như một cơ cấu đồng bộ khi các sự kiện delay zero xuất hiện. Delay delta được dùng khi delay zero được chỉ định và trình bày như sau:

```
a <= b after 0 ns;
```

Một trường hợp khác dùng delay delta là khi delay zero được chỉ định.

Ví dụ 2-7: Phát biểu hình như sau:

```
a <= b ;
```

Trong cả 2 trường hợp, tín hiệu **b** thay đổi giá trị từ 1 sự kiện, tín hiệu **a** được gán tín hiệu sau khoảng thời gian delay delta.

Một mô hình VHDL tương đương của mạch điện được trình bày như hình 2-6:

```
ENTITY      reg    IS
  PORT      (a, clock:    IN     BIT;
             d:           OUT    BIT);
END         reg;

ARCHITECTURE    test   OF    reg    IS
SIGNAL      b,c:   BIT;
BEGIN
        b  <=  NOT (a);              -- no delay
        c  <=  NOT (clock AND b);
        d  <=  c AND b;
END         test;
```

3. DRIVER

VHDL có một phương pháp duy nhất để xử lý các tín hiệu có nhiều nguồn kích. Các tín hiệu có nhiều nguồn kích rất hữu ích cho mô hình bus dữ liệu, bus hai chiều, ... Mô hình chính xác các loại mạch điện này trong VHDL yêu cầu phải biết các khái niệm về mạch kích (thúc) tín hiệu. Mỗi một driver của VHDL xem như một tín hiệu góp phần cho giá trị tổng thể của một tín hiệu.

Một tín hiệu có nhiều nguồn kích sẽ có nhiều driver. Các giá trị của tất cả các driver được phân tích cùng nhau để tạo ra giá trị duy nhất cho tín hiệu này. Phương pháp phân tích tất cả các tín hiệu góp phần thành một giá trị duy nhất là thông qua hàm phân tích. Một hàm phân tích là hàm do người thiết kế viết, sẽ được gọi mỗi khi một driver của tín hiệu thay đổi giá trị.

a. Tạo driver:

Các driver được tạo ra bằng các phát biểu tín hiệu. Một phép gán tín hiệu đồng thời bên trong một kiến trúc tạo ra một driver cho một phép gán tín hiệu. Do đó nhiều phép gán tín hiệu sẽ tạo ra nhiều driver cho một tín hiệu. Hãy khảo sát kiến trúc sau đây

```
ARCHITECTURE    test   OF    test   IS
BEGIN
        a  <=  b      AFTER  10ns;
        a  <=  c      AFTER  10ns;
END         test;
```

Tín hiệu ***a*** sẽ được kích từ hai nguồn ***b*** và ***c.*** Mỗi phát biểu gán tín hiệu đồng thời sẽ tạo ra một driver cho tín hiệu ***a***.

Phát biểu thứ nhất tạo ra một driver chứa giá trị của tín hiệu *b* được trì hoãn 10 ns.

Phát biểu thứ hai tạo ra một driver chứa giá trị của tín hiệu *c* được trì hoãn 10 ns.

Những người thiết kế sử dụng VHDL không muốn tùy ý thêm vào các ràng buộc ngôn ngữ đối với hành vi của tín hiệu. Khi tổng hợp ví dụ ở trên sẽ nối tắt ***b*** và ***c*** với nhau.

b. Mô hình nhiều driver xấu:

Ta hãy khảo sát một mô hình thoạt nhìn có vẻ đúng nhưng lại không thực hiện chức năng như người sử dụng dự định. Mô hình này sử dụng một mạch đa hợp 4 đường sang 1 đường đã thảo luận:

```
USE         IEEE.std_logic_1164.ALL;
ENTITY      mux4 IS
  PORT      (i0, i1, i2, i3, a, b:   IN    STD_LOGIC;
             q:                      OUT   STD_LOGIC);
END         mux4;

ARCHITECTURE    bad   OF    mux4 IS
BEGIN
  q     <= i0 WHEN a = '0' AND b = '0'  ELSE '0'
  q     <= i1 WHEN a = '1' AND b = '0'  ELSE '0'
  q     <= i2 WHEN a = '0' AND b = '1'  ELSE '0'
  q     <= i3 WHEN a = '1' AND b = '1'  ELSE '0'
END     bad;
```

Mô hình này gán: *i0* cho *q* khi *a* bằng '0' và *b* bằng '0';

i1 khi *a* bằng '1' và *b* bằng '0'; ...

Thoạt nhìn, mô hình này có vẻ hoạt động đúng. Tuy nhiên mỗi phép gán cho tín hiệu *q* tạo ra một tín driver mới cho tín hiệu *q*. Bốn driver cho tín hiệu *q* được tạo ra trong mô hình này.

Mỗi driver sẽ kích giá trị của một trong các ngõ vào *i0*, *i1*, *i2*, *i3* hoặc '0'. Giá trị được kích phụ thuộc vào các tín hiệu *a* và *b*.

Nếu *a* bằng '0' và *b* bằng '0', phát biểu gán đầu tiên đặt giá trị của *i0* và một trong các driver của *q*. Ba phát biểu gán khác không thỏa điều kiện và do vậy sẽ kích giá trị '0'. Như vậy, ba driver sẽ kích giá trị '0' và một driver sẽ kích giá trị của *i0*.

Các hàm phân tích điển hình khó mà dự đoán kết quả ngõ ra *q* mong muốn, nhưng giá trị thực của nó chính là *i0*. Cách tốt hơn để viết cho mô hình này là chỉ xây dựng một mạch driver (kích) cho tín hiệu *q* như được trình bày sau đây

```
ARCHITECTURE    better OF    mux4 IS
BEGIN
  q     <=    i0     WHEN a = '0' AND b = '0'  ELSE
              i1     WHEN a = '1' AND b = '0'  ELSE
              i2     WHEN a = '0' AND b = '1'  ELSE
              i3     WHEN a = '1' AND b = '1'  ELSE
              'X';
END     better;
```

4. GENERIC

Generic là một cơ cấu tổng quát được dùng để chuyển thông tin đến thực thể. Thông tin được chuyển đến thực thể là một trong các kiểu được VHDL cho phép.

Generic cũng có thể được dùng để chuyển các kiểu dữ liệu bất kỳ do người thiết kế định nghĩa bao gồm các thông tin như điện dung tải, điện trở, ... Các thông số tổng hợp như độ rộng đường dữ liệu, độ rộng tín hiệu, ... có thể chuyển được dưới dạng các generic.

Tất cả dữ liệu được chuyển đến một thực thể là các thông tin rõ ràng. Các giá trị dữ liệu có liên quan đến instance đang được truyền dữ liệu. Ở phương pháp này, người thiết kế có thể truyền các giá trị khác nhau đến các instance khác nhau trong thiết kế.

Dữ liệu được truyền đến instance là dữ liệu tĩnh. Sau khi mô hình được cho thêm chi tiết (liên kết với trình mô phỏng), dữ liệu sẽ không thay đổi trong thời gian mô phỏng. Các generic không thể được gán thông tin cho các thành phần chạy chương trình mô phỏng. Thông tin chứa trong các generic được chuyển đến instance thành phần hoặc một khối có thể được sử dụng để thay đổi các kết quả trong mô phỏng, nhưng các kết quả không thể sửa đổi các generic.

Ví dụ 2-8: thực thể cổng AND có 3 generic kết hợp:

```
ARCHITECTURE    load_dependent    OF    and2  IS
SIGNAL          internal          BIT;

BEGIN
  internal      <= a    AND b;
  c             <= Internal  AFTER (rise + (load*2 ns))   Internal  = '1' ELSE Internal
                   AFTER (rise + (load*3 ns));
END     load_dependent;
```


Kiến trúc này khai báo một tín hiệu cục bộ gọi là ***internal*** để lưu giá trị của biểu thức ***a*** và ***b***. Các giá trị tính toán trước dùng cho nhiều instance là một phương pháp rất hiệu quả cho việc xây dựng mô hình.

Các generic ***rise***, ***fall*** và ***load*** chứa các giá trị đã được chuyển vào bởi phát biểu của instance thành phần. Ta hãy khảo sát một phần của mô hình mà nó thể hiện các thành phần loại **AND2** trong một cấu trúc khác:

```
ENTITY          test    IS
  GENERIC       (rise, fall:          TIME; load:   INTEGER);
  PORT          (ina, inb, inc, ind:  IN    STD_LOGIC;
                 Out1, out2:          OUT   STD_LOGIC);
END             test;

ARCHITECTURE    test_arch     OF    test    IS
COMPONENT       and2
        GENERIC (rise, fall:  TIME; load:   INTEGER);
        PORT(a, b: IN BIT;  c: OUT BIT);
END COMPONENT;
BEGIN
U1:     and2 GENERIC MAP (10 ns, 12 ns, 3) PORT MAP (ina, inb, out1);
U2:     and2 GENERIC MAP (9 ns, 11 ns, 5) PORT MAP (inc, ind, out2);
END     test_arch;
```

Phát biểu kiến trúc đầu tiên khai báo các thành phần sẽ được sử dụng trong mô hình. Trong ví dụ này thành phần **AND2** được khai báo.

Tiếp theo, thân của phát biểu kiến trúc chứa hai phát biểu thể hiện thành phần của các thành phần **U1** và **U2**. Port ***a*** của thành phần **U1** được ánh xạ đến tín hiệu ***ina***, port ***b*** được ánh xạ đến tín hiệu ***inb*** và port ***c*** được ánh xạ đến tín hiệu ***out1*.** Cùng phương pháp như vậy thành phần **U2** được ánh xạ đến các tín hiệu ***inc*, *ind*** và ***out2*.**

Generic ***rise*** của thể hiện **U1** được ánh xạ đến **10ns**, generic ***fall*** được ánh xạ đến **12ns** và generic **load** được ánh xạ đến **3**. Các generic của thành phần U2 được ánh xạ đến các giá trị **9 ns**, **11ns** và giá trị **5**.

Các generic cũng có thể có giá trị mặc định, các giá trị này được ghi đè nếu các giá trị thực tế được ánh xạ đến các generic. Ví dụ tiếp theo trình bày hai thể hiện thành phần loại **AND2**.

Trong thành phần **U1**, giá trị thực tế được ánh xạ đến generic và các giá trị này được dùng để điều khiển hành vi mô phỏng nếu được chỉ định rõ, ngược lại sẽ phát sinh lỗi.

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      test    IS
  GENERIC   (rise, fall:          TIME; load:   INTEGER);
  PORT      (ina, inb, inc, ind:  IN    STD_LOGIC;
             Out1, out2:          OUT   STD_LOGIC);
END         test;

ARCHITECTURE    test_arch   OF    test    IS
COMPONENT       and2
        GENERIC (rise, fall:    TIME:= 10 ns;       load:   INTEGER := 0);
        PORT(a, b: IN BIT;  c: OUT BIT);
END COMPONENT;
BEGIN
U1:     and2 GENERIC MAP (10 ns, 12 ns, 3) PORT MAP (ina, inb, out1);
U2:     and2 PORT MAP (inc, ind, out2);
END     test_arch;
```


Như đã nhìn thấy, các generic có nhiều cách dùng. Việc sử dụng generic chỉ bị giới hạn bởi sự sáng tạo của người viết mô hình.

5. CÁC PHÁT BIỂU KHỐI

Các khối là cơ cấu từng phần trong VHDL – cho phép người thiết kế các khối trong mô hình. Ví dụ nếu bạn thiết kế CPU thì có thể chia ra thành nhiều khối như khối ALU, khối bank thanh ghi và các khối khác.

Mỗi khối có thể khai báo các tín hiệu cục bộ, kiểu dữ liệu, hằng số, ... Bất kỳ đối tượng nào – mà nó có thể được khai báo trong phần khai báo kiến trúc – thì có thể được khai báo trong phần khai báo khối.

Ví dụ 2-9: dùng các phát biểu khối:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;
```

```
package     bit32  IS
  TYPE      tw32   IS    ARRAY  ( 31 downto 0 ) OF std_logic;
END         bit32;

LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;
USE         WORK.bit32.ALL;

ENTITY      cpu    IS
 PORT       (clk, interrupt:      IN          STD_LOGIC;
             addr:                OUT         tw32;
             data:                INOUT       tw32;);
END         cpu;

ARCHITECTURE    cpu_blk     OF    cpu    IS
SIGNAL          ibus, dbus:   tw32;
BEGIN
 ALU:  BLOCK
       SIGNAL       qbus:  tw32;
            BEGIN
                 -- alu behavior statements
            END   BLOCK  ALU;

 REG8: BLOCK
       SIGNAL       zbus:  tw32;
            BEGIN
                 REG1:        BLOCK
                 SIGNAL       qbus:  tw32;
                      BEGIN
                           -- REG1  behavior statements
                      END   BLOCK  REG1;
                      -- more REG8  statements

            END   BLOCK  REG8;
END     cpu_blk;
```


Thực thể **cpu** có khai báo thực thể ngoài cùng trong mô hình (mô hình cho ví dụ này chưa hoàn chỉnh). Thực thể **cpu** khai báo 4 port – dùng cho giao tiếp mô hình. Các port **clk** và **interrupt** là các port ngõ vào, **addr** là port ngõ ra và **data** là port hai chiều **inout**. Tất cả các port này được nhìn thấy ở bất kỳ khối nào, được khai báo trong kiến trúc cho thực thể này. Các port ngõ vào có thể đọc và các port ngõ ra có thể được gán giá trị.

Các tín hiệu **ibus** và **dbus** là các tín hiệu cục bộ được khai báo trong kiến trúc **cpu_blk**. Các tín hiệu này là cục bộ đối với kiến trúc **cpu_blk** và do đó các khối bên ngoài không được truy xuất các tín hiệu này.

Tuy nhiên, bất kỳ khối nằm nào bên trong kiến trúc đó đều có thể truy xuất các tín hiệu này. Các khối có cấp mức độ thấp thì có thể truy xuất đến các tín hiệu có cấp mức độ cao hơn, nhưng

các khối có cấp mức độ cao hơn thì không thể truy xuất đến các tín hiệu cục bộ của khối cấp thấp hơn.

Tín hiệu **qbus** được khai báo trong phần khai báo khối của khối **ALU**. Tín hiệu này là tín hiệu cục bộ cho khối **ALU** và các khối bên ngoài không thể truy xuất. Tất cả các phát biểu nằm bên trong khối **ALU** có thể truy xuất **qbus**, nhưng các phát biểu bên ngoài khối **ALU** thì không thể dùng **qbus**.

Tương tự, tín hiệu **zbus** cho khối **REG8**. Khối REG1 nằm bên trong khối **REG8** có thể truy xuất tín hiệu **zbus** và tất cả các phát biểu khác trong khối **REG8** cũng có thể truy xuất tín hiệu **zbus**.

Trong phần khai báo của khối **REG1** còn khai báo một tín hiệu khác gọi là **qbus**. Tín hiệu này có cùng tên với tín hiệu **qbus** được khai báo trong khối **ALU** – điều này có gây ra xung đột gì không ? Đối với chương trình chương trình biên dịch thì hai tín hiệu này là độc lập. Hai tín hiệu này được khai báo trong hai vùng khác nhau và chỉ có hiệu lực cho vùng đó.

Một trường hợp khác lồng vào nhau như sau:

```
BLK1: BLOCK
        SIGNAL      qbus:  tw32;
        BEGIN
                BLK2: BLOCK
                        SIGNAL      qbus:  tw32;
                        BEGIN
                                -- blk2 statements
                        END   BLOCK  BLK2;
                -- blk1 statements
        END   BLOCK  BLK1;
```


Trong ví dụ này, tín hiệu **qbus** được khai báo trong 2 khối. Cấu trúc lồng vào nhau trong mô hình này là một khối có chứa một khối khác.

Khối **BLK2** truy xuất 2 tín hiệu được gọi là **qbus**: tín hiệu qbus thứ nhất khai báo trong BLK2 và tín hiệu qbus thứ 2 khai báo trong BLK1. Khối **BLK1** là cha của khối **BLK2**. Tuy nhiên, khối **BLK2** xem tín hiệu **qbus** nằm trong chính nó, nhưng tín hiệu **qbus** của khối BLK1 sẽ bị ghi đè bởi khai báo cùng tên của tín hiệu khối **BLK2**.

Tín hiệu **qbus** từ **BLK1** có thể được nhìn thấy bên trong khối **BLK2**, nếu tên của tín hiệu **qbus** được bổ sung thêm bằng tên của khối. Cụ thể cho ví dụ ở trên, để truy xuất tín hiệu **qbus** từ khối **BLK1** thì dùng **BLK1.qbus**.

Như đã đề cập ở trên, các khối chứa các vùng của mô hình bên trong nó. Nhưng các khối là duy nhất bởi vì một khối có thể chứa các port và các generic. Điều này cho phép người thiết kế ánh xạ lại các tín hiệu và các generic bên ngoài đến các tín hiệu và generic nằm bên trong khối. Nhưng tại sao người thiết kế muốn làm điều này.

Dung lượng của các port và các generic trong một khối cho phép người thiết kế dùng lại các khối đã viết cho mục đích khác trong thiết kế mới.

Giả sử ta muốn cải tiến thiết kế CPU và cần mở rộng thêm chức năng cho khối ALU, và ta giả sử rằng một người thiết kế khác có khối ALU mới có thể thực hiện được các chức năng mà ta

cần thì vấn đề chỉ còn là hiệu chỉnh lại tên của các port và các generic cho tương thích với khối mới là xong. Phải ánh xạ các tên của tín hiệu và các thông số generic trong thiết kế đang cải tiến với các port và các generic đã xây dựng của khối ALU mới.

Ví dụ 2-10: minh hoạ cho sự cải tiến:

```
package     math        IS
  TYPE      tw32        IS    ARRAY  ( 31 downto 0 ) OF std_logic;
  FUNCTIOB  tw_add(a,b; tw32)  RETURN tw32;
  FUNCTIOB  tw_sub(a,b; tw32)  RETURN tw32;
END         math;
USE         WORK.math.ALL;

LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      cpu   IS
 PORT       (clk, interrupt:     IN       STD_LOGIC;
             addr:               OUT      tw32;
            cont :               IN       INTEGER;
             data:               INOUT    tw32);
END         cpu;

ARCHITECTURE    cpu_blk     OF    cpu    IS
SIGNAL      ibus, dbus:   tw32;
BEGIN
 ALU:  BLOCK
       PORT (abus, bbus:  IN     tw32;
             D_out:       OUT    tw32;
             ctbus:       IN     INTEGER);
       PORT MAP (abus => ibus, bbus => dbus, d_out => data, ctbus => cont);
       SIGNAL       qbus:  tw32;
             BEGIN
                  D_out <=    tw_add (abus, bbus) WHEN ctbus = 0 ELSE
                              tw_add (abus, bbus) WHEN ctbus = 1 ELSE
                              abus;
             END  BLOCK  ALU;
END    cpu_blk;
```


Về cơ bản mô hình này giống như mô hình đã trình bày ngoại trừ port và các phát biểu ánh xạ port trong phần khai báo khối ALU. Phát biểu port khai báo số lượng được dùng cho khối, hướng của port và loại dữ liệu của port. Phát biểu ánh xạ port sẽ ánh xạ port mới với các tín hiệu hoặc các port tồn tại bên ngoài khối. Port **abus** được ánh xạ cho tín hiệu cục bộ **ibus** của kiến trúc **CPU_BLK**, port **bbus** được ánh xạ cho **dbus**. Các port d_out và ctbus được ánh xạ cho các port bên ngoài của thực thể.

Ánh xạ có nghĩa là kết nối giữa port và tín hiệu bên ngoài chẳng hạn như khi có một sự thay đổi giá trị trên một tín hiệu nối đến 1 port thì port sẽ thay đổi sang giá trị mới. Nếu sự thay đổi xảy ra trong tín hiệu **ibus** thì giá trị mới của **ibus** được truyền vào khối ALU và port abus sẽ có giá trị mới. Tương tự cho tất cả các port.

Các khối có bảo vệ

Các phát biểu khối có khối hành vi lồng vào bên trong được xem như những khối có bảo vệ. Một khối có bảo vệ chứa một biểu thức bảo vệ – có thể cho phép và không cho phép các driver bên trong khối.

Biểu thức bảo vệ là biểu thức đại số boolean: nếu bằng true thì các driver bên trong khối được phép và nếu bằng false thì các driver bị cấm.

Chúng ta sẽ khảo sát ví dụ 2-11:

Ví dụ 2-11: có khối bảo vệ

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      latch IS
  PORT      (d, clk:    IN      STD_LOGIC;
             q, qb:     OUT     STD_LOGIC);
END         latch;

ARCHITECTURE    latch_guard OF  latch IS
BEGIN
  G1:   BLOCK (clk = '1')
        BEGIN
              q     <=  GUARDED       d       AFTER 5 ns;
              qb    <=  GUARDED   NOT (d)     AFTER 7 ns;
        END   BLOCK  G1;
END     latch_guard;
```


Mô hình này minh họa cách thức mô hình mạch chốt có thể được viết dùng khối có bảo vệ. Thực thể khai báo 4 port cần thiết cho mạch chốt và chỉ có một phát biểu trong kiến trúc. Phát biểu chính là phát biểu khối có bảo vệ.

Phát biểu khối có bảo vệ giống như phát biểu khối bình thường, ngoại trừ biểu thức bảo vệ nằm sau từ khoá **BLOCK**. Biểu thức bảo vệ trong vệ trong ví dụ này là (**clk='1'**). Đây là biểu thức luận lý và trả về kết quả là **true** khi giá trị của **clk** bằng '**1**' và sẽ có giá trị là **false** khi **clk** có giá trị khác.

Khi biểu thức bảo vệ là true thì tất cả driver của các phát biểu gán tín hiệu bảo vệ được phép hoặc được mở. Khi biểu thức bảo vệ là false thì tất cả các driver của các phát biểu gán tín hiệu bảo vệ bị cấm hoặc bị tắt.

Có 2 phát biểu gán tín hiệu bảo vệ trong mô hình: phát biểu thứ nhất là gán giá trị **q** và câu lệnh gán còn lại là gán giá trị **qb**. Phát biểu gán tín hiệu có bảo vệ được nhận ra bằng từ khoá **GUARDED** nằm giữa "<=" và thành phần của biểu thức.

Khi port **clk** của thực thể có giá trị là '1' thì biểu thức bảo vệ có giá trị là true và khi giá trị của ngõ vào **d** sẽ xuất hiện ở ngõ ra **q** sau khoảng thời gian trễ 7ns.

Khi port **clk** có giá trị là '0' hoặc bất kỳ giá trị nào khác hợp lệ của kiểu dữ liệu thì ngõ ra **q** và **qb** chuyển sang tắt và giá trị ngõ ra của tín hiệu được xác định bởi giá trị được gán cho một giá trị mặc định bởi hàm phân giải. Khi **clk** không bằng '1' thì các driver được xây dựng cho các lệnh gán tín hiệu **q** và **qb** trong kiến trúc sẽ chuyển sang tắt. Các driver không tham gia vào giá trị tổng thể của tín hiệu.

Gán tín hiệu có thể được bảo vệ bằng cách dùng từ khoá **GUARDED**. Tín hiệu mới được khai báo hoàn toàn trong khối khi khối có biểu thức bảo vệ. Tín hiệu này được gọi là **GUARD.** Giá trị của nó là giá trị của biểu thức bảo vệ. Tín hiệu này có thể được dùng để các xử lý khác xảy ra.

Các khối rất tiện lợi để chia nhỏ thiết kế thành các khối nhỏ hơn, các đơn vị dễ quản lý hơn. Chúng cho phép người thiết kế sự mềm dẻo để xây dựng những thiết kế lớn từ những khối nhỏ hơn và cung cấp phương pháp thuận tiện cho điều khiển các driver đối với tín hiệu.

6. TÓM TẮT

- Cách gán tín hiệu là dạng cơ bản nhất của mô hình hành vi.
- Phát biểu gán tín hiệu có thể được lựa chọn tuỳ vào điều kiện.
- Phát biểu gán tín hiệu có thể chứa thời gian trễ.
- VHDL chứa trì hoãn trễ quán tính và trì hoãn truyền.
- Các điểm thời gian mô phỏng delta dùng để các sự kiện hoạt động đúng thời gian.
- Các driver cho một tín hiệu được xây dựng bởi các phát biểu gán tín hiệu.
- Generic được dùng để truyền dữ liệu cho thực thể.
- Các phát biểu khối cho phép xây dựng nhóm trong cùng một thực thể.
- Các phát biểu khối bảo vệ cho phép khả năng tắt các driver trong một khối.

V. XỬ LÝ TUẦN TỰ

Ở phần trước chúng ta đã khảo sát mô hình hành vi dùng các phát biểu đồng thời. Chúng ta đã thảo luận các phát biểu gán đồng thời cũng như các phát biểu khối và thể hiện thành phần.

Trong phần này chúng ta tập trung cho phát biểu tuần tự. Các phát biểu tuần tự là các phát biểu thực hiện nối tiếp nhau.

1. PHÁT BIỂU

Trong một kiến trúc của một thực thể, tất cả các phát biểu là đồng thời. Vậy thì các phát biểu tuần tự tồn tại ở đâu trong VHDL ?

Có một phát biểu được gọi là phát biểu quá trình – chỉ chứa các phát biểu tuần tự. Phát biểu quá trình cũng chính là phát biểu đồng thời. Phát biểu quá trình có thể tồn tại trong kiến trúc và những vùng xác định trong kiến trúc – nơi chứa các lệnh tuần tự.

Phát biểu quá trình có phần khai báo và phần phát biểu. Trong phần khai báo thì các kiểu, các biến, các hằng số, các chương trình con, ... , có thể được khai báo. Phần phát biểu chỉ chứa các phát biểu tuần tự. Các phát biểu tuần tự chứa các phát biểu **CASE**, phát biểu **IF THEN ELSE**, phát biểu **LOOP**, ...

a. Danh sách nhạy

Phát biểu quá trình có thể có một danh sách nhạy rõ ràng. Danh sách này định nghĩa các tín hiệu mà chúng làm cho các phát biểu bên trong phát biểu quá trình thực hiện khi có một hoặc nhiều phần tử trong danh sách thay đổi giá trị. Danh sách nhạy là danh sách của các tín hiệu mà chúng sẽ làm cho quá trình thực hiện.

b. Ví dụ về quá trình

Chúng ta hãy quan sát ví dụ 2-12 của phát biểu quá trình trong kiến trúc

Ví dụ 2-12:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      nand2       IS
  PORT      (a, b:      IN    STD_LOGIC;
             c:         OUT   STD_LOGIC);
END         nand2;

ARCHITECTURE  nand2   OF    nand2   IS
BEGIN
        PROCESS (a,b)
        VARIABLE temp : STD_LOGIC;
        BEGIN
                temp  :=    NOT   (a   AND   b);
                IF    (temp = '1' )  THEN     c <= temp   AFTER     6 ns;
                ELSIF  (temp = '0' )  THEN    c <= temp   AFTER     5 ns;
                ELSE  c <= temp   AFTER     6 ns;
                END IF;
        END PROCESS;
END     and2;
```


Trong ví dụ này trình bày cách viết mô hình cho một cổng NAND đơn giản có 2 ngõ vào dùng phát biểu tuần tự.

Phát biểu **USE** khai báo gói VHDL để cung cấp những thông tin cần thiết cho phép xây dựng mô hình cho cổng **NAND**. Phát biểu **USE** được dùng để cho mô hình có thể được mô phỏng với trình mô phỏng VHDL mà không cần thêm bước hiệu chỉnh nào.

Thực thể khai báo 3 port cho cổng **nand2**. Port **a** và port **b** là ngõ vào và port **c** là ngõ ra. Tên của kiến trúc cùng tên với thực thể.

Kiến trúc chỉ chứa một phát biểu, một phát biểu quá trình đồng thời.

Phần khai báo quá trình bắt đầu tại từ khoá **PROCESS** và kết thúc tại từ khoá **BEGIN**.

Phần phát biểu quá trình bắt đầu tại từ khoá **BEGIN** và kết thúc tại từ khoá **END PROCESS**. Phần khai báo quá trình có hai phát biểu tuần tự: một phát biểu gán biến:

```
        temp  :=    NOT   (a   AND   b);
```

Và một phát biểu **IF THEN ELSE**

```
IF      (temp = '1' ) THEN      c <= temp    AFTER      6 ns;
ELSIF   (temp = '0' ) THEN      c <= temp    AFTER      5 ns;
ELSE  c <= temp     AFTER       6 ns;
END IF;
```

Quá trình chứa danh sách nhạy rõ ràng với 2 tín hiệu chứa bên trong:

```
PROCESS (a,b)
```

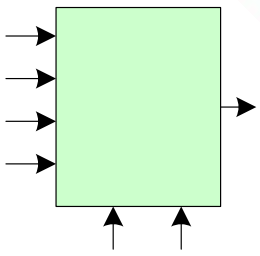
Quá trình đang thực hiện (nhạy) với 2 tín hiệu **a** và **b**. Trong ví dụ này, **a** và **b** là 2 port ngõ vào của mô hình. Các port ngõ vào xây dựng các tín hiệu có thể được dùng như các ngõ vào, các port ngõ ra xây dựng các tín hiệu có thể được dùng như các ngõ ra, các port **inout** xây dựng các tín hiệu có thể được dùng cho cả 2 vào – ra.

Khi port **a** hoặc **b** thay đổi giá trị thì phát biểu bên trong quá trình được thực hiện. Mỗi phát biểu được thực hiện theo thứ tự nối tiếp bắt đầu với phát biểu trên cùng của phát biểu quá trình và thực hiện từ trên xuống dưới. Sau khi tất cả các phát biểu đã được thực hiện một lần, quá trình đợi cho đến khi có sự thay đổi tín hiệu hoặc port nằm trong danh sách nhạy.

2. GÁN BIẾN KHÁC VỚI GÁN TÍN HIỆU

Phát biểu thứ nhất trong phát biểu quá trình là gán biến – gán giá trị cho biến **temp**. Ở phần trước chúng ta đã thảo luận về cách các tín hiệu nhận giá trị sau thời gian trì hoãn hoặc sau thời gian trì hoãn delta. Gán biến xảy ra ngay lập tức khi phát biểu được thực hiện. Ví dụ: trong mô hình này, phát biểu thứ nhất phải gán giá trị cho biến **temp** để phát biểu thứ hai sử dụng. Gán biến không có thời gian trì hoãn, xảy ra ngay lập tức.

Chúng ta sẽ khảo sát hai ví dụ minh họa để phân biệt các ***lệnh gán tín hiệu*** và ***gán biến***. Cả hai ví dụ đều là mô hình của mạch đa hợp 4 đường sang 1 đường. Kí hiệu và bảng trạng thái như hình 2-9. Một trong 4 ngõ vào được truyền đến ngõ ra tùy thuộc vào giá trị của A và B.



B	A	Q
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Hình 2-9. Kí hiệu mạch đa hợp và bảng trạng thái.

Mô hình thứ nhất là mô hình không đúng và mô hình thứ hai là mô hình đúng.

a. Ví dụ mô hình mạch đa hợp không đúng

Mô hình không đúng của bộ đa hợp có thiếu sót làm cho mô hình hoạt động không đúng. Mô hình này được trình bày như sau:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      mux   IS
  PORT      (i0, i1, i2, i3, a, b:   IN    STD_LOGIC;
             q:                      OUT   STD_LOGIC);
END         mux;

ARCHITECTURE    wrong  OF   mux   IS
SIGNAL      muxval:      INTEGER;
BEGIN
        PROCESS (i0, i1, i2, i3, a, b)
        BEGIN
             muxval  <=  0;
             IF     (a = '1' )      THEN  muxval<= muxval + 1;
             END IF;
             IF     (b = '1' )      THEN  muxval<= muxval + 2;
             END IF;

             CASE  muxval      IS
                  WHEN  0  => q <= I0 AFTER  10 ns;
                  WHEN  1  => q <= I1 AFTER  10 ns;
                  WHEN  2  => q <= I2 AFTER  10 ns;
                  WHEN  3  => q <= I3 AFTER  10 ns;
                  WHEN  OTHERS => NULL;
             END CASE;
        END PROCESS;
END     wrong;
```


Khi có 1 trong các tín hiệu ngõ vào nằm trong danh sách nhạy thay đổi giá trị thì các phát biểu tuần tự được thực hiện.

Phát biểu quá trình trong ví dụ này chứa 4 phát biểu tuần tự.

Phát biểu thứ nhất khởi tạo tín hiệu cục bộ **muxval** với giá trị '0'. Các phát biểu tuần tự con cộng giá trị cho tín hiệu tuỳ thuộc vào của các tín hiệu vào **a** và **b**.

Phát biểu **case** cuối cùng lựa chọn một ngõ vào để truyền đến ngõ ra tuỳ thuộc vào giá trị của tín hiệu **muxval**. Mô hình này có một thiếu sót nghiêm trọng với phát biểu: **muxval <=0;** làm cho giá trị 0 được sắp xếp như một sự kiện đối với tín hiệu **muxval**. Thực tế thì giá trị 0 được sắp xếp trong một sự kiện cho thời gian trễ delta để mô phỏng bởi vì không có thời gian trì hoãn.

Khi phát biểu thứ 2:

```
             IF     (a = '1' )      THEN  muxval <= muxval + 1;
             END IF;
```

được thực hiện, giá trị của tín hiệu **muxval** là giá trị được truyền ở lần cuối cùng. ***Giá trị mới đã sắp xếp từ phát biểu thứ nhất chưa được truyền đến***. Trong thực tế thì khi nhiều phép gán cho tín hiệu xảy ra trong cùng phát biểu quá trình thì giá trị gán sau cùng là giá trị được truyền.

Tín hiệu **muxval** có giá trị vô nghĩa (không xác định) khi bắt đầu quá trình. ***Giá trị của* muxval *không thay đổi cho đến khi các phát biểu nằm trong quá trình được thực hiện xong***. Nếu tín hiệu **b** có giá trị là '1' thì sau đó giá trị vô nghĩa được cộng thêm với 2.

Ví dụ tiếp theo sẽ chặt chẽ hơn. Sự khác nhau giữa 2 mô hình của 2 ví dụ là khai báo **muxval** và phép gán cho **mulval**. Trong mô hình ví dụ trước, **muxval** là ***tín hiệu*** và ***phát biểu gán tín hiệu*** được dùng để gán giá trị cho **muxval**. Trong mô hình ví dụ này thì **muxval** là ***biến*** và ***phép gán biến*** được dùng để gán giá trị cho **muxval**.

b. Ví dụ mô hình mạch đa hợp đúng

Trong ví dụ này thì mô hình không đúng ở trên được viết lại để cho thấy cách giải quyết vấn đề của mô hình không đúng:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      mux   IS
  PORT      (i0, i1, i2, i3, a, b:   IN    STD_LOGIC;
             q:                      OUT   STD_LOGIC);
END         mux;

ARCHITECTURE     better  OF   mux   IS
BEGIN
        PROCESS (i0, i1, i2, i3, a, b)
        VARIABLE   muxval : INTEGER;
        BEGIN
                muxval  :=  0;
                IF      (a = '1' )      THEN  muxval := muxval + 1;
                END IF;
                IF      (b = '1' )      THEN  muxval := muxval + 2;
                END IF;

                CASE  muxval        IS
                        WHEN  0  => q <= I0 AFTER  10 ns;
                        WHEN  1  => q <= I1 AFTER  10 ns;
                        WHEN  2  => q <= I2 AFTER  10 ns;
                        WHEN  3  => q <= I3 AFTER  10 ns;
                        WHEN  OTHERS => NULL;
                END CASE;
        END PROCESS;
END     better;
```


Khi phát biểu thứ nhất **muxval:=0;** được thực hiện thì giá trị 0 được đặt vào cho biến **muxval** ngay lập tức. Giá trị không được sắp xếp vì **muxval** trong ví dụ này là ***biến***, không phải là tín hiệu. ***Các biến tượng trưng cho ô nhớ lưu trữ cục bộ khác với tín hiệu tượng trưng cho kết nối mạch***

điện bên trong. Ô nhớ lưu trữ cục bộ được cập nhật ngay lập tức và giá trị mới có thể được dùng sau đó trong mô hình cho các tính toán sau đó.

Do biến **muxval** được khởi tạo giá trị 0 ngay lập tức nên hai phát biểu gán trong quá trình dùng giá trị 0 như giá trị khởi tạo và cộng với các con số thích hợp tuỳ thuộc vào tuỳ thuộc vào giá trị của tín hiệu **a** và **b**. Các phát biểu gán này cũng được thực hiện ngay lập tức và do đó khi phát biểu **case** được thực hiện thì biến **muxval** đã chứa giá trị đúng. Từ giá trị này tín hiệu ngõ vào đúng có thể được truyền đến ngõ ra.

3. CÁC PHÁT BIỂU TUẦN TỰ

Các phát biểu tuần tự nằm bên trong phát biểu quá trình và nằm trong các chương trình con. Trong phần này chúng ta sẽ khảo sát các phát biểu tuần tự nằm bên trong phát biểu quá trình.

Các phát biểu tuần tự sẽ được trình bày là: **IF, CASE, LOOP, EXIT, ASSERT, WAIT.**

4. PHÁT BIỂU IF

Phát biểu **IF** cho phép chọn một trong các câu lệnh để thực hiện. Kết quả trả về của mệnh đề điều kiện là giá trị kiểu **BOOLEAN**. Dựa vào kết quả trả về của mệnh đề điều kiện để cho phép một lệnh có được thực thi hay không.

Cú pháp của phát biểu IF như sau

```
if        condition then  sequential statements;
   [elsif condition then  sequential statements;]
   [else                  sequential statements;]
end if;
```

Ví dụ 2-13: cho phát biểu IF

```
IF      (x < 10 )      THEN  a:= b;
END IF;
```

Phát biểu được bắt đầu bằng từ khoá **IF**. Theo sau từ khoá **IF** là mệnh đề điều kiện **(x < 10).** Điều kiện trả về true khi **x** nhỏ hơn 10, ngược lại thì có giá trị false. Khi điều kiện là true thì phát biểu giữa **THEN** và **END IF** được thực hiện. Trong ví dụ này thì lệnh phát biểu gán (**a:=b**) được thực hiện bất kỳ lúc nào **x** nhỏ hơn 10.

Ví dụ 2-14: cho phát biểu IF THEN ELSE:

```
IF      (day = sunday )     THEN   weekend := true;
ELSIF   (day = saturday )   THEN   weekend := true;
ELSE    weekday := true;
END IF;
```

Trong ví dụ này có hai biến – **weekend** và **weekday** – được thiết lập giá trị tuỳ thuộc vào giá trị của tín hiệu **day**. Biến **weekend** được thiết lập là **true** khi **day** bằng **Saturday** hoặc **Sunday**. Ngược lại biến **weekday** được thiết lập là **true**.

Thực hiện phát biểu **IF** bắt đầu kiểm tra xem biến **day** có bằng với **Sunday** hay không.

Nếu kết quả là true thì phát biểu kế được thực hiện và điều khiển được chuyển tới phát biểu nằm sau từ khoá **END IF**.

Ngược lại điều khiển được chuyển tới phần phát biểu **ELSIF** và kiểm tra **day** có phải là **Saturday** hay không. Nếu biến **day** là **Saturday** thì phát biểu kế được thực hiện và điều khiển được chuyển tới phát biểu nằm sau từ khoá **END IF**.

Cuối cùng nếu **day** không bằng **Sunday** và **Saturday** thì phần phát biểu **ELSE** được thực hiện. Phát biểu IF có thể có nhiều phần phát biểu **ELSIF** nhưng chỉ có duy nhất một lần phát biểu **ELSE**.

5. PHÁT BIỂU CASE

Phát biểu CASE được sử dụng khi giá trị của biểu thức duy nhất có thể được dùng để lựa chọn một trong số hoạt động. Cú pháp cho phát biểu CASE như sau:

```
case expression is
        when choices  =>  sequential statements;
        when choices  =>  sequential statements;
              -- branches are allowed
        [ when others => sequential statements ];
    end case;
```

Kết quả biểu thức là số nguyên, hoặc kiểu liệt kê của mảng một chiều chẳng hạn như bit_vector. Phát biểu case đánh giá biểu thức và so sánh giá trị của biểu thức với mỗi giá trị của các lựa chọn. Mệnh đề **when** tương ứng với lựa chọn trùng hợp sẽ được thực hiện. Các nguyên tắc sau phải nhớ:

- Không có 2 lựa chọn trùng lắp (lựa chọn này bao phủ lựa chọn kia).
- Nếu phát biểu lựa chọn "**when others**" không hiện diện thì tất cả giá trị có thể có của biểu thức phải bao phủ hết bởi các lựa chọn.

Phát biểu **CASE** chứa từ khoá **CASE** theo sau là ***biểu thức*** và từ khoá **IS**. Biểu thức có giá trị tương thích với **CHOICES** nằm trong phát biểu **WHEN** hoặc tương thích với phát biểu **WHEN OTHERS**.

Nếu biểu thức tương thích với phần **CHOICES** của các phát biểu **WHEN choices =>** thì **sequence_of_statement** theo sau sẽ được thực hiện. Sau khi các phát biểu này được thực hiện xong thì điều khiển chuyển tới phát biểu nằm sau từ khoá **END CASE**.

Ví dụ 2-15: phát biểu case :

```
CASE instruction IS
        WHEN load_accum       => accum    <= data;
        WHEN store_accum      => data_out <= accum;
        WHEN load|store       => process_IO (addr) ;
        WHEN OTHERS => process_error (instruction);
END CASE;
```

Phát biểu **CASE** thực hiện phát biểu tương ứng tuỳ thuộc vào giá trị của biểu thức ngõ vào. Nếu giá trị của biểu thức là một giá trị nằm trong các giá trị được liệt kê trong các mệnh đề **WHEN** thì sau đó phát biểu theo sau mệnh đề **WHEN** được thực hiện. Ngược lại thì phát biểu theo sau mệnh đề **OTHERS** được thực hiện.

Trong ví dụ này khi giá trị của biểu thức là **load_accum** thì phát biểu gán đầu tiên được thực hiện. Nếu giá trị của biểu thức là **load** hoặc **store** thì thủ tục **process_IO** được gọi.

Nếu giá trị của biểu thức nằm ngoài dãy lựa chọn đã cho thì sau đó mệnh đề **OTHERS** tương thích với biểu thức và phát biểu theo sau mệnh đề **OTHERS** được thực hiện. Sẽ phát sinh lỗi nếu không có mệnh đề **OTHERS** và các lựa chọn đã cho không bao trùm giá trị có thể có của biểu thức.

Ví dụ 2-16 với biểu thức có kết quả trả về phức tạp hơn. Phát biểu **CASE** dùng kiểu dữ liệu này để lựa chọn một trong các phát biểu.

Ví dụ 2-16:

```
TYPE   vectype      IS   ARRAY (0 TO 1) OF BIT;
VARIABLE     bit_vector: vectype;
...
               CASE  bit_vector     IS
                     WHEN  "00"          =>     RETURN 0;
                     WHEN  "01"          =>     RETURN 1;
                     WHEN  "10"          =>     RETURN 2;
                     WHEN  "11"          =>     RETURN 3;
               END CASE;
```

Ví dụ này trình bày một phương pháp chuyển đổi một mảng bit thành một số nguyên.

Khi hai bit của biến **bit_vec** có giá trị '0' thì lựa chọn "00" tương thích và giá trị trả về là 0.

Khi hai bit của biến **bit_vec** có giá trị '1' thì lựa chọn "11" tương thích và giá trị trả về là 3.

Phát biểu **CASE** không cần mệnh đề **OTHERS** vì tất cả các giá trị của biến **bit_vec** được liệt kê bởi các lựa chọn.

6. PHÁT BIỂU LOOP

Phát biểu LOOP được sử dụng để lập lại chuỗi các lệnh tuần tự. Cú pháp cho phát biểu LOOP như sau:

```
[ loop_label :]iteration_scheme loop
        sequential statements
        [next  [label] [when condition];
        [exit  [label] [when condition];
    end loop [loop_label];
```

Phát biểu **next** và **exit** là các phát biểu tuần tự chỉ có thể được sử dụng bên trong vòng lặp.

- Phát biểu **next** chấm dứt phần còn lại của vòng lặp hiện tại và sau đó sẽ lặp lại vòng lặp kế.
- Phát biểu **exit** bỏ qua phần còn lại của phát biểu, chấm dứt hoàn toàn vòng lặp và tiếp tục với phát biểu kế sau vòng lặp vị thoát.

Có 3 loại vòng lặp:

- Vòng lặp **loop** cơ bản
- Vòng lặp **while ... loop**
- Vòng lặp **for ... loop**

a. Phát biểu vòng lặp LOOP cơ bản

Vòng lặp thực hiện liên tục cho đến khi bắt gặp phát biểu exit hoặc next. Cú pháp như sau:

```
[ loop_label :] loop
        sequential statements
        [next  [label] [when condition];
        [exit  [label] [when condition];
    end loop [ loop_label];
```

Vòng lặp cơ bản phải có ít nhất một phát biểu **wait**. Ví dụ một bộ đếm 5 bit đếm từ 0 đến 31. Khi bộ đếm đạt giá trị 31 thì bộ đếm bắt đầu tràn về 0. Phát biểu **wait** có chứa trong chương trình để cho vòng lặp sẽ thực hiện mỗi khi xung clock thay đổi từ '0' sang '1'.

Ví dụ 2-17: sử dụng vòng lặp cơ bản cho mạch đếm từ 0 đến 31.

```
ENTITY      count31     IS
  PORT      (CLK:       IN    STD_LOGIC;
              Count:    OUT   INTEGER);
END         count31;

ARCHITECTURE    behav_count  OF  count31 IS
BEGIN
P_COUNT:    PROCESS
        VARIABLE    intern_value : INTEGER :=0;
        BEGIN
            Count <=  intern_value;
            LOOP
                WAIT UNTIL CLK = '1';
                intern_value :=  (intern_value + 1) mod 32;
                Count <=  intern_value;
            END LOOP;
        END PROCESS P_COUNT;
END     behav_count;
```


b. Phát biểu vòng lặp While – LOOP

Vòng lặp **while** … **loop** đánh giá điều kiện lặp dạng Boolean. Khi điều kiện là TRUE, vòng lặp thực hiện, ngược lại vòng lặp thực hiện liên tục cho đến khi bắt gặp phát biểu **exit** hoặc **next**. Cú pháp như sau:

```
[ loop_label :] while condition loop
        sequential statements
        [next  [label] [when condition];
        [exit  [label] [when condition];
    end loop[ loop_label ];
```

Điều kiện lặp được kiểm tra trước mỗi lần lặp kể cả lần lặp đầu tiên. Nếu điều kiện là false thì vòng lặp chấm dứt.

c. Phát biểu vòng lặp FOR – LOOP:

Vòng lặp for loop dùng giản đồ lặp số nguyên để xác định số lần lặp. Cú pháp như sau:

```
[ loop_label :] for identifier in range loop
        sequential statements
        [next  [label] [when condition];
        [exit  [label] [when condition];
```

```
end loop[ loop_label ];
```

Chỉ số lặp được khai báo tự động bởi chính vòng lặp do đó không cần khai báo riêng lẻ. Giá trị của chỉ số chỉ có thể được đọc bên trong vòng lặp và không có hiệu lực ở ngoài vòng lặp. Không thể gán hoặc thay đổi giá trị của chỉ số lặp. Chỉ số lặp này đối ngược với vòng lặp **while – loop** khi điều kiện của vòng lặp **while-loop** có chứa biến và được hiệu chỉnh bên trong vòng lặp.

Dãy số của vòng lặp phải là một dãy số nguyên có thể tính toán được ở một trong các dạng sau, trong mỗi dạng thì integer_expression phải là một số nguyên:

- *integer_expression* **to** *integer_expression*
- *integer_expression* **downto** *integer_expression*

d. Phát biểu Next và Exit:

Phát biểu **next** bỏ qua việc thực hiện để đến thực hiện vòng lặp kế của phát biểu vòng lặp. Cú pháp như sau:

```
next [label] [when  condition];
```

Từ khoá **when** là tuỳ chọn và sẽ được thực hiện phát biểu kế khi điều kiện đánh giá là true.

Phát biểu **exit** bỏ qua phần còn lại của phát biểu, chấm dứt hoàn toàn vòng lặp và tiếp tục với phát biểu kế sau khi vòng lặp bị thoát. Cú pháp như sau:

```
exit [label] [when  condition];
```

Từ khoá **when** là tuỳ chọn và sẽ được thực hiện phát biểu kế khi điều kiện đánh giá là true.

Chú ý: sự khác nhau giữa phát biểu **next** và **exit** là phát biểu **exit** chấm dứt vòng lặp.

Ví dụ 2-18: Minh hoạ cho vòng lặp next

```
PROCESS (A, B)
CONSTANT    max_limit: INTEGER :=255;
BEGIN
        FOR i IN 0 TO max_limit
        LOOP
                IF      (done(i) = true )      THEN  next;
                ELSE   done(i) = true;
                END IF;
                q(i) <=  a(i) and b(i);
        END LOOP;
END     PROCESS;
```

Phát biểu quá trình chứa một phát biểu vòng lặp **LOOP**. Phát biểu **LOOP** là and các bit của mảng **a** và mảng **b** và đặt kết quả vào mảng **q**. Mô tả hành vi tiếp tục cho đến khi nào cờ trong mảng **done** là false.

Nếu giá trị của cờ done là true với chỉ số i thì phát biểu **next** được thực hiện. Việc thực hiện tiếp tục với phát biểu đầu tiên của vòng lặp và chỉ số **i** bây giờ có giá trị là **i + 1**.

Nếu giá trị của cờ done là false thì phát biểu **next** không được thực hiện và việc thực hiện tiếp tục với phát biểu chứa trong mệnh đề **ELSE** cho phát biểu **IF**.

Phát biểu **next** cho phép người thiết kế khả năng ngừng việc thực hiện các lệnh của vòng lặp và tiếp tục với vòng lặp tiếp theo. Có một số trường hợp khác thì cần thoát khỏi vòng lặp thì khả năng này được thực hiện bởi phát biểu **EXIT**.

Trong thời gian thực thi một phát biểu LOOP, có thể ta cần nhảy ra khỏi vòng lặp. Điều này có thể xảy ra do một lỗi quan trọng xuất hiện trong thời gian thực thi mô hình hoặc toàn bộ việc xử lý kết thúc sớm.

Phát biểu EXIT của VHDL cho phép người thiết kế thoát hoặc nhảy ra khỏi một phát biểu LOOP hiện đang thực thi. Phát biểu EXIT làm cho việc thực thi dừng ở vị trí của phát biểu này. Việc thực thi sẽ tiếp tục ở phát biểu theo sau phát biểu LOOP.

Ví dụ 2-19: Minh hoạ cho vòng lặp exit

```
PROCESS (A)
CONSTANT    int_a: INTEGER;
BEGIN
        Int_a := a;
        FOR i IN 0 TO max_limit
        LOOP
                IF      (int_a <0 )     THEN  exit;
                ELSE   (int_a := int_a - 1);
                        q(i) <=  3.14 / REAL (int_a* i);      -- signal assign
                END IF;
        END LOOP;
        y <=  q;
END     PROCESS;
```


Bên trong phát biểu của quá trình này, giá trị của ***int_a*** luôn luôn được giả định là giá trị dương lớn hơn 0. Nếu giá trị của ***int_a*** âm hoặc bằng 0 thì sinh ra lỗi và việc tính toán sẽ không được hoàn tất. Nếu giá trị của ***int_a*** nhỏ hơn hoặc bằng 0 thì phát biểu **IF** là đúng và phát biểu **EXIT** sẽ được thực thi. Vòng lặp kết thúc ngay lập tức và phát biểu kế tiếp được thực thi chính là phát biểu gán cho ***y*** sau phát biểu LOOP.

Phát biểu EXIT có 3 loại cơ bản. Loại thứ nhất yêu cầu phát biểu **EXIT** không có nhãn vòng lặp hoặc **WHEN** condition. Nếu các điều kiện này đúng, phát biểu **EXIT** sẽ hoạt động như sau: phát biểu EXIT chỉ thoát khỏi phát biểu **LOOP** hiện tại. Nếu phát biểu EXIT ở bên trong một phát biểu **LOOP** và phát biểu **LOOP** này được lồng trong một phát biểu **LOOP** khác, phát biểu **EXIT** chỉ thoát khỏi phát biểu **LOOP** bên trong. Việc thực thi vẫn duy trì trong phát biểu **LOOP** bên ngoài. Phát biểu **EXIT** chỉ thoát khỏi phát biểu **LOOP** gần nhất.

Ví dụ 2-20: Minh hoạ

```
PROCESS (a)
BEGIN
  First_loop:    FOR i IN 0 TO 100 LOOP
        second_loop:   FOR j IN 0 TO 10 LOOP
                ….
                Exit   second_loop;          -- exit the second loop only
                ….
                Exit   first_loop;           -- exit the first loop and second loop
        END LOOP;
```

```
    END LOOP;
END     PROCESS;
```

Nếu phát biểu có thêm điều kiện **WHEN** thì phát biểu EXIT chỉ thoát khỏi vòng lặp khi điều kiện là **TRUE**. Phát biểu kế được thực hiện tuỳ thuộc vào điều kiện của phát biểu **EXIT** có nhãn chỉ định hay không.

Nếu nhãn của vòng lặp được chỉ định thì phát biểu kế được thực hiện thì chứa trong phát biểu LOOP chỉ định bởi nhãn vòng lặp.

Nếu không có nhãn thì phát biểu kế được thực hiện thì nằm ở vòng lặp kế bên ngoài.

Ví dụ 2-21: cho phát biểu EXIT với điều kiện WHEN

```
Exit   first_loop  WHEN (i < 10);
```

Phát biểu này kết thúc việc thực hiện của vòng lặp có nhãn là **first_loop** khi biểu thức i < 10.

Phát biểu **EXIT** cung cấp một phương pháp dễ dàng và nhanh chóng để thoát khỏi phát biểu **LOOP** khi toàn bộ công việc xử lý kết thúc hoặc một lỗi hay cảnh báo xảy ra.

7. PHÁT BIỂU ASSERT

Phát biểu ASSERT là phát biểu rất hữu ích để báo cáo chuỗi văn bản đến người thiết kế. Phát biểu **ASSERT** kiểm tra giá trị của một biểu thức logic xem đúng hay sai. Nếu giá trị là đúng, phát biểu này không làm gì cả. Nếu giá trị là sai, phát biểu **ASSERT** xuất một chuỗi dạng văn bản được chỉ định bởi người sử dụng đến ngõ ra chuẩn của thiết bị đầu cuối.

Người thiết kế cũng có thể chỉ ra mức độ nghiêm trọng để xuất chuỗi dạng văn bản. Theo trình tự tăng dần của mức độ nghiêm trọng ta có 4 mức: chú ý, cảnh báo, lỗi và thất bại. Mức độ nghiêm trọng cung cấp cho người thiết kế khả năng phân loại thông điệp thành các loại thích hợp.

Phát biểu ASSERT được sử dụng chủ yếu để quản lý khi viết mô hình, không có phần cứng nào được xây dựng.

Cú pháp:

```
assert_statement ::=
  ASSERT condition
[REPORT expression];
```

Từ khóa ASSERT được theo bởi một biểu thức có giá trị logic được gọi là một điều kiện (condition). Điều kiện này xác định biểu thức dạng văn bản được phép xuất ra hay không bởi phát biểu REPORT. Nếu sai, biểu thức dạng văn bản được xuất, còn nếu đúng, biểu thức dạng văn bản không được xuất.

Chúng ta khảo sát ví dụ thực tế cho phát biểu ASSERT, ví dụ này thực hiện việc kiểm tra thiết lập dữ liệu giữa hai tín hiệu điều khiển flip flop D. Hầu hết các flip flop yêu cầu dữ liệu ngõ vào ***din*** phải ở giá trị ổn định với một khoảng thời gian xác định trước khi có cạnh xung clock xuất hiện. Thời gian này được gọi là thời gian thiết lập và đảm bảo rằng dữ liệu vào ***din*** sẽ được chốt vào bên trong flip flop. Ví dụ 2-22 về phát biểu ASSERT tạo ra thông báo lỗi cho người thiết kế biết nếu thời gian thiết lập không đủ hay bị vi phạm.

Ví dụ 2-22:

```
PROCESS (clk, din)
```

```
        VARIABLE   last_d_change:      TIME :=0 ns;
        VARIABLE   last_d_value :      STD_LOGIC :='X';
        VARIABLE   last_clk_value:     STD_LOGIC :='X';
BEGIN
        IF      (last_d_value /= din)        THEN  -- /=  is not equal
                last_d_change : = NOW;
                last_d_value : = din;
        END IF;
        IF      (last_clk_value /= clk)      THEN
                last_clk_value: = clk;
                IF      (clk= '1')      THEN
                        ASSERT (NOW – last_d_change >= 20ns)
                        REPORT "setup violation"
                        SEVERITY WARNING;
                END IF;
        END IF;
END     PROCESS;
```

Quá trình dùng 3 biến cục bộ để ghi lại thời gian và giá trị sau cùng của tín hiệu **din** cũng như giá trị của tín hiệu **clk**. Do lưu trữ giá trị sau cùng của **clk** và **din** nên chúng ta có thể xác định xem tín hiệu có thay đổi giá trị hay không.

Bằng cách ghi lại thời gian sau cùng mà tín hiệu **din** thay đổi nên chúng ta có thể đo được ***thời gian hiện tại*** so với lần chuyển trạng thái sau cùng của **din**, từ đó chúng ta sẽ biết được thời gian thiết lập có bị vi phạm hay không.

Bước thứ nhất trong quá trình là kiểm tra xem tín hiệu **din** có thay đổi hay không. Nếu có thay đổi thì thời gian của chuyển đổi được lưu lại dùng hàm **NOW**. Hàm này trả về thời gian mô phỏng hiện tại. Tương tự, giá trị sau cùng của **din** cũng được lưu trữ cho việc kiểm tra sau này.

Bước tiếp theo là kiểm tra xem tín hiệu clk có chuyển trạng thái hay không. Nếu biến **last_clk_value** không bằng với giá trị hiện tại của clk thì sẽ có sự chuyển trạng thái xảy ra. Nếu tín hiệu **clk** là '1' thì xem như đã có cạnh lên của xung clk.

Khi có cạnh lên của xung clk thì chúng ta cần kiểm tra xem thời gian thiết lập có bị vi phạm hay không. Nếu lần chuyển sau cùng của tín hiệu **d** nhỏ hơn 20 ns thì biểu thức:

```
NOW – last_d_change
```

Trả về giá trị nhỏ hơn 20ns. Phát biểu **ASSERT** bị kích và gởi ra thông tin thời gian thiết lập bị vi phạm cảnh báo đến người thiết kế.

Nếu lần chuyển trạng thái sau cùng trên tín hiệu d xảy ra dài hơn 20 ns thì biểu thức trên sẽ trả về kết quả với giá trị lớn hơn 20 ns và phát biểu **ASSERT** không bị kích.

8. PHÁT BIỂU WAIT

Phát biểu WAIT cung cấp cho người thiết kế khả năng dừng tạm thời việc thực thi tuần tự của một quá trình hoặc một chương trình con. Các điều kiện để tiếp tục việc thực thi quá trình hoặc chương trình con tạm dừng có thể được thực hiện bằng một trong 3 cách như sau:

- **WAIT ON** (chờ) các thay đổi tín hiệu.
- **WAIT UNTIL** (chờ cho đến khi) một biểu thức là đúng.

- **WAIT FOR** (chờ trong) một khoảng thời gian cụ thể.

Các phát biểu **WAIT** có thể được sử dụng cho một số mục đích khác nhau. Hiện nay lệnh **WAIT** được dụng phổ biến nhất là dùng để chỉ ra các ngõ vào xung clock cho các công cụ tổng hợp. Phát biểu **WAIT** chỉ định xung clock cho phát biểu quá trình được đọc bởi các công cụ tổng hợp nhằm tạo ra mạch logic tuần tự chẳng hạn như các thanh ghi và các flip flop.

Các công dụng khác là trì hoãn việc thực thi quá trình trong một khoảng thời gian hoặc hiệu chỉnh danh sách nhạy của một quá trình.

Chúng ta khảo sát phát biểu quá trình sử dụng phát biểu WAIT được dùng để tạo mức logic tuần tự:

```
PROCESS (clock, d)
BEGIN
        WAIT UNTIL clock = '1'  AND clock'EVENT;
                q  <= d;
END     PROCESS;
```

Quá trình này dùng để phát hiện xung clock cạnh lên của flip flop. Thuộc tính **'EVENT** đi cùng với xung clock là true khi ngõ vào xung clock có thay đổi. Tổ hợp của 2 điều kiện là xung clock có giá trị '1' và xung clock có thay đổi nên có thể xem như xung clock vừa xuất hiện xung cạnh lên. Kết quả của quá trình này là đợi hoặc chờ phát biểu **WAIT** cho đến khi xuất hiện xung clock cạnh lên và giá trị của **d** được gán cho ngõ ra **q**.

Thêm chức năng reset đồng bộ cho ví dụ trên như sau:

```
PROCESS (clock, d)
BEGIN
        WAIT UNTIL clock = '1'   AND clock'EVENT;
        IF      (reset = '1')   THEN  q  <= '0';
        ELSE            q  <= d;
        END IF;
END     PROCESS;
```

Trong ví dụ trên thì khi có xung clock thì tín hiệu **reset** được kiểm tra trước: nếu tín hiệu reset tích cực thì gán giá trị '0' cho ngõ ra **q**, ngược lại thì gán tín hiệu **d**.

Chức năng reset không đồng bộ cũng được thực hiện như sau:

```
PROCESS (clock, d)
BEGIN
        IF      (reset = '1')   THEN  q  <= '0';
        ELSE  clock'EVENT  AND    clock = '1' THEN  q  <= d;
        END IF;
        WAIT  ON reset, clock;
END     PROCESS;
```

Quá trình này chứa phát biểu **WAIT ON** sẽ làm cho quá trình ngừng thực hiện cho đến khi có một trong hai sự kiện **reset** hoặc xung **clock** thay đổi. Sau đó phát biểu **IF** được thực hiện và nếu tín hiệu **reset** là tích cực thì flip flop bị **reset** bất đồng bộ, ngược lại xung clock được kiểm tra để phát hiện cạnh lên và nếu đúng thì sẽ chuyển trạng thái ngõ vào **d** cho ngõ ra **q**.

Phát biểu WAIT cũng có thể được sử dụng để điều khiển các tín hiệu của quá trình hay chương trình con nhạy với bất kỳ điểm nào trong khi thực hiện chương trình.

Ví dụ 2-23:

```
PROCESS
BEGIN
        WAIT  ON     a;
        WAIT  ON     b;
END     PROCESS;
```

a. Phát biểu WAIT ON signal

Ví dụ về phát biểu này đã được trình bày ở trên, phát biểu WAIT ON chỉ định một danh sách một hoặc nhiều tín hiệu mà phát biểu WAIT sẽ đợi chờ sự thay đổi. Nếu có tín hiệu nào thay đổi thì việc thực hiện sẽ được tiếp tục với phát biểu nằm sau phát biểu WAIT.

Ví dụ 2-24:

```
    WAIT      ON     a, b;
```

Khi một trong hai sự kiện xảy ra cho a hoặc b thì quá trình sẽ tiếp tục với phát biểu tiếp theo phát biểu WAIT.

b. Phát biểu WAIT UNTIL expression

Mệnh đề **WAIT UNTIL boolean_expression** sẽ ngừng quá trình thực hiện cho đến khi kết quả trả về của biểu thức là true. Phát biểu này thiết lập một danh sách nhạy các tín hiệu được dùng trong biểu thức. Khi bất kỳ tín hiệu nào trong biểu thức thay đổi thì biểu thức được đánh giá. Biểu thức trả về kết quả kiểu Boolean: khi kết quả là true thì quá trình thực hiện sẽ tiếp tục với phát biểu theo sau lệnh **WAIT**. Ngược lại thì quá trình sẽ tiếp tục ngừng.

Ví dụ 2-25:

```
    WAIT      UNTIL  ((x*10) < 100);
```

Trong ví dụ này thì khi x lớn hơn hay bằng 10 thì quá trình tiếp tục ngừng và chỉ thực hiện lệnh tiếp theo khi x nhỏ hơn 10.

c. Phát biểu WAIT FOR time_expression

Mệnh đề **WAIT FOR time_expression** sẽ ngừng quá trình thực hiện trong một khoảng thời gian được chỉ định bởi biểu thức thời gian. Sau khi thời gian được chỉ định trong biểu thức đã hết thì quá trình tiếp tục với phát biểu nằm sau lệnh **WAIT**.

Ví dụ 2-26:

```
        WAIT  FOR  10 ns;
        WAIT  FOR  (a * (b * c));
```

Phát biểu thứ nhất thì biểu thức chỉ đơn giản là hằng số.

Phát biểu thứ hai thì biểu thức thời gian là một biểu thức phải được tính toán trước và trả về là giá trị thời gian. Sau khi giá trị này đã được tính toán thì phát biểu WAIT sẽ dùng giá trị thời gian này làm thời gian đợi.

d. Phát biểu WAIT với nhiều sự kiện

Phát biểu WAIT với nhiều điều kiện với ví dụ 2-27 như sau:

Ví dụ 2-27:

```
WAIT    ON  nmi, interrupt UNTIL ((nmi = true) or (interrupt = true) ) FOR 5 usec;
```

Phát biểu này đợi một sự kiện trên các tín hiệu nmi và interrupt và chỉ tiếp tục nếu interrupt hoặc nmi là true hoặc cho đến hết thời gian 5µs. Chỉ khi một hoặc nhiều điều kiện trên là true thì quá trình thực hiện mới tiếp tục. Hãy xem ví dụ 2-28:

Ví dụ 2-28:

```
WAIT    UNTIL  (interrupt = true ) OR (old_clk = '1');
```

Phải chắc chắn có ít nhất một giá trị trong biểu thức chứa tín hiệu – điều này là cần thiết để đảm bảo phát biểu WAIT không phải chờ đợi mãi.

Nếu cả hai interrupt và old_clk đều là biến thì phải biểu WAIT không phải đánh giá lại khi hai biến này thay đổi giá trị. Chỉ cần 1 tín hiệu thay đổi sẽ làm cho phát biểu WAIT hoặc các phát biểu gán tín hiệu đồng thời đánh giá lại.

VI. CÁC KIỂU ĐỐI TƯỢNG TRONG VHDL

Các đối tượng của VHDL chứa một trong các kiểu sau:

- Signal: tượng trưng cho các dây dẫn kết nối bên trong dùng để kết nối các port của các thành phần với nhau.
- Variable: được dùng như một ô nhớ cục bộ để lưu dữ liệu tạm thời chỉ nhìn thấy được bên trong quá trình.
- Constant: là khai báo hằng số.

1. KHAI BÁO TÍN HIỆU (SIGNAL):

Kiểu tín hiệu được sử dụng để kết nối các thực thể lại với nhau để tạo ra một module. Signal là phương thức truyền các tín hiệu động giữa các thực thể với nhau.

Kiểu signal được khai báo như sau

```
SIGNAL signal_name: signal_type[:=initial_value];
```

Theo sau từ khoá **SIGNAL** là một hoặc nhiều tên tín hiệu. Với mỗi tên tín hiệu sẽ tạo ra một signal mới. Phân biệt tên với loại tín hiệu bằng dấu ':'. Loại tín hiệu chỉ định loại dữ liệu của thông tin chứa trong tín hiệu. Cuối cùng thì tín hiệu có thể chứa một giá trị bắt đầu (giá trị khởi gán) để cho giá trị tín hiệu có thể được khởi động.

Các tín hiệu có thể được khai báo trong phần khai báo của thực thể, trong kiến trúc và trong khai báo gói. Các tín hiệu trong khai báo gói được xem như các tín hiệu toàn cục.

Ví dụ 2-29: về khai báo tín hiệu như sau:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      mux   IS
  PORT      (i0, i1, i2, i3, a, b:   IN    STD_LOGIC;
             q:                      OUT   STD_LOGIC);
END         mux;

PACKAGE     sigdecl      IS
        TYPE      bus_type     IS     ARRAY (0 to 7) OF std_logic;
```

```
        SIGNAL      vcc:        std_logic := '1';
        SIGNAL      ground:     std_logic := '0';
        FUNCTION    magic_function (a:  IN bus_type)  RETURN bus_type;
END     sigdecl;
```

2. KHAI BÁO BIẾN (VARIABLE):

Các biến số dùng trong VHDL không tạo ra bất kỳ phần cứng nào, các biến lưu trữ giá trị tạm thời của các tín hiệu. Kiểu biến được khai báo như sau:

```
VARIABLE variable_name : variable_type[:=value];
```

Theo sau từ khoá variable là một hoặc nhiều tên biến. Các tên biến cách nhau bằng dấu ';'. Mỗi tên biến sẽ tạo ra một biến mới. Variable_type sẽ xác định kiểu dữ liệu cho biến.

Ví dụ 2-30:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;

ENTITY      and5  IS
  PORT      (a, b, c, d, e: IN    std_logic;
             q:             OUT   std_logic);
END         and5;

ARCHITECTURE    and5  OF    and5  IS
BEGIN
        PROCESS  (a, b, c, d, e)
        VARIABLE   state:       STD_LOGIC;
        VARIABLE   delay :      time;
        BEGIN

        state := a AND b AND c AND d  AND e;
                IF      state = '1'     THEN  delay:= 4 ns;
                ELSIF state = '0'       THEN  delay := 3 ns
                ELSE  delay := 4 ns
                END IF;
                delay <= state AFTER delay;
        END PROCESS ;
END     and5;
```

Ví dụ này là kiến trúc cho cổng AND có 5 ngõ vào. Có 2 biến được khai báo là biến **state** và biến **delay**, biến **state** được sử dụng để cất giữ giá trị tạm thời các ngõ vào của cổng **AND**, **delay** được sử dụng để lưu giá trị thời gian trễ. Cả hai dữ liệu này không thể là dữ liệu tĩnh bởi vì giá trị của chúng phụ thuộc vào giá trị của các ngõ vào a, b, c, d, e. Các tín hiệu có thể được dùng để lưu trữ dữ liệu nhưng không được dùng bởi vì các lý do sau:

- Các biến thường hiệu quả hơn cho phép gán tín hiệu bởi vì nó xảy ra ngay lập tức, trong khi tín hiệu thì phải chờ sắp xếp.
- Các biến chiếm ít bộ nhớ hơn trong khi các tín hiệu cần nhiều thông tin để cho phép sắp xếp và các thuộc tính tín hiệu.
- Dùng tín hiệu phải yêu cầu phát biểu WAIT để đồng bộ phép gán tín hiệu cho mỗi lần thực hiện khi sử dụng.

Khi bất kỳ tín hiệu **a, b, c, d,** hoặc **e** thay đổi thì quá trình thực hiện. Biến **state** được gán cho hàm AND của tất cả các ngõ vào. Bước tiếp theo thì dựa vào giá trị của biến **state** mà biến **delay** được gán giá trị thời gian trễ. Dựa vào giá trị thời gian đã được gán cho biến **delay**, tín hiệu ngõ ra **q** sẽ có giá trị của biến **state**.

3. KHAI BÁO HẰNG SỐ :

Hằng số giữ một giá trị không đổi trong quá trình thiết kế. Hằng số được khai báo như sau

```
CONSTANT constant_name :type_name[:=value];
```

Các tên hằng cách nhau bằng dấu ';'. Các giá trị của hằng là tuỳ ý, kiểu hằng có quy định giống kiểu tín hiệu.

Hằng có thể sử dụng trong toàn thực thể nếu hằng được khai báo trong khối khai báo của thực thể, hoặc có thể được sử dụng trong toàn package nếu nó được khai báo trong đoạn khai báo của package.

Ví dụ 2-31:

```
CONSTANT    PI : REAL := 3.1414;
```

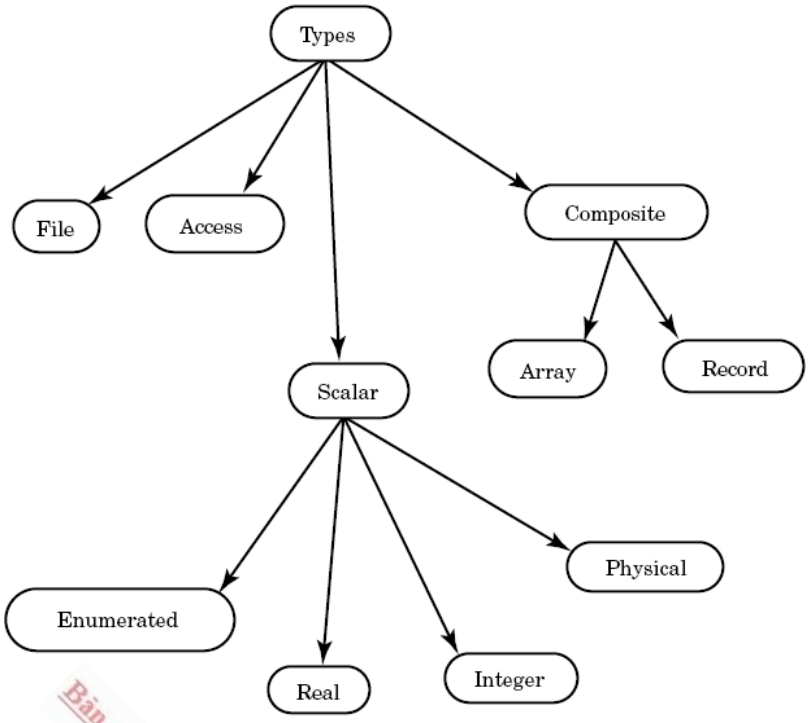
VII. CÁC KIỂU DỮ LIỆU TRONG VHDL

Tất cả các đối tượng đã trình bày là tín hiệu, biến và hằng số có thể khai báo dùng các kiểu dữ liệu. Ngôn ngữ VHDL chứa rất nhiều kiểu dữ liệu dùng để xây dựng cho các đối tượng từ đơn giản đến phức tạp.

Để định nghĩa một loại dữ liệu mới thì phải khai báo loại dữ liệu. Khai báo loại dữ liệu định nghĩa tên của loại dữ liệu và tầm vực hay giới hạn của dữ liệu. Các khai báo dữ liệu được phép khai báo trong phần khai báo gói, phần khai báo thực thể, phần khai báo kiến trúc, phần khai báo chương trình con và phần khai báo quá trình.

Hình 2-10 trình bày các kiểu dữ liệu có sử dụng trong ngôn ngữ VHDL. Bốn loại dữ liệu lớn là loại scalar, loại đa hợp (composite), loại access và loại file.

- Loại scalar chứa các loại dữ liệu đơn giản như số thực và số nguyên.
- Loại đa hợp bao gồm mảng và bản ghi.
- Loại access tương đương với con trỏ trong các ngôn ngữ lập trình thông thường.
- Loại file cho người thiết kế khả năng khai báo đối tượng file với các loại file được định nghĩa bởi người thiết kế.



Hình 2-10. Giản đồ các loại dữ liệu trong VHDL.

1. LOẠI SCALAR

Loại dữ liệu scalar bao gồm các dữ liệu như sau:

- Kiểu số nguyên.
- Kiểu số thực.
- Kiểu đếm, liệt kê được.
- Kiểu dữ liệu vật lý.

a. Kiểu số nguyên INTEGER

Kiểu số nguyên giống như kiểu số nguyên của toán học. Tất cả các hàm toán học đã xác định như hàm cộng, trừ, nhân và chia đều có thể áp dụng cho kiểu dữ liệu này.

Trong ngôn ngữ VHDL không có chỉ định giới hạn số nguyên cực đại nhưng có chỉ định giới hạn cực tiểu của số nguyên từ "-2,147, 483,647 đến 12,147,483,647". Giới hạn cực tiểu được chỉ định bởi gói chuẩn chứa trong thư viện chuẩn.

Các ví dụ về các giá trị số nguyên như sau:

Ví dụ 2-32:

```
ARCHITECTURE    test    OF    test    IS
BEGIN
        PROCESS  (X)
        VARIABLE   a:      INTEGER;
        VARIABLE   b:      int type;
        BEGIN
        a := 1;         -- ok 1
        a := -1;        -- ok 2
        a := 1.0;       -- error 3
        END PROCESS
```

```
END test;
```

Hai phát biểu thứ 1 và 2 là các phép gán số nguyên. Phát biểu thứ 3 là phép cho biến số nguyên là một số không phải số nguyên, khi biên dịch thì phát biểu này có thể sinh ra lỗi. Bất kỳ con số nào có dấu chấm được xem là số thực.

b. Kiểu dữ liệu đã định nghĩa

VHDL chứa nhiều loại dữ liệu đã được định nghĩa, đặc biệt là các chuẩn IEEE 1076 và IEEE 1164. Đặc biệt hơn nữa chẳng hạn như định nghĩa các loại dữ liệu có thể được tìm thấy trong thư viện và trong gói.

- Gói chuẩn ***standard*** (package standard) của thư viện std: xác định các kiểu dữ liệu **BIT**, **BOOLEAN, INTEGER** và **REAL**.
- Gói ***std_logic_1164*** của thư viện IEEE: định nghĩa các kiểu dữ liệu **STD_LOGIC** và **STD_ULOGIC**.
- Gói ***std_logic_arith*** của thư viện IEEE: định nghĩa các kiểu dữ liệu **SIGNED** và **STD_ULOGIC**, cùng với nhiều hàm chuyển đổi dữ liệu như *conv_integer (p)*, *conv_unsigned(p,b)*, *conv_signed(p,b)* và *conv_std_vector(p,b)*.
- Gói ***std_logic_signed*** và ***std_logic_unsigned*** của thư viện IEEE: chứa các hàm cho phép hoạt động với các dữ liệu **STD_LOGIC_VECTOR** được thực hiện khi dữ liệu loại **SIGNED** hoặc **UNSIGNED**.

Tất cả các dữ liệu đã định nghĩa (nằm trong các gói hoặc thư viện ở trên) được mô tả như sau:

BIT (and **BIT_VECTOR**): có 2 mức logic là '0' và '1'.

Ví dụ 2-33: khai báo các tín hiệu dạng BIT và BIT_VECTOR

```
SIGNAL x: BIT;  -- x được khai báo là một tín hiệu bit
SIGNAL y: BIT_VECTOR (3 DOWNTO 0);  -- y là một vector 4 bit với bit bên trái là MSB.
SIGNAL w: BIT_VECTOR (0 TO 7);  -- w là một vector 8 bit với bit bên phải là MSB.
```

Dựa vào các tín hiệu ở trên thì các phát biểu gán sau là hợp lệ:

```
x <= '1';          -- x được gán với giá trị là 1.
```

Chú ý kí hiệu dấu nháy đơn chỉ được dùng cho 1 bit đơn.

```
y <= "0111";    -- y được gán tín hiệu 4 bit có giá trị là "0111" với bit MSB là bit '0'.
```

Chú ý kí hiệu dấu nháy kép được dùng cho vector.

```
w <= "01110001";  -- w được gán tín hiệu 8 bit có giá trị là "01110001" với bit MSB là bit '1'.
```

STD_LOGIC và **STD_LOGIC_VECTOR**: có 8 giá trị logic được giới thiệu trong chuẩn IEEE 1164 chuẩn:

'X'	có nghĩa là chưa xác định
'0'	có nghĩa là mức thấp
'1'	có nghĩa là mức cao
'Z'	trạng thái tổng trở cao
'W'	yếu chưa xác định
'L'	yếu thấp

'H' yếu cao

'-' bất chấp

Ví dụ 2-34: khai báo các tín hiệu dạng STD_LOGIC và STD_LOGIC_VECTOR

```
SIGNAL x: STD_LOGIC        -- x được khai báo là tín hiệu bit kiểu STD_LOGIC.
SIGNAL y: STD_LOGIC_VECTOR (3 DOWNTO 0):= "0001";
 -- y được khai báo là một vector 4 bit với bit bên trái là MSB và giá trị khởi tạo là "0001".
```

Chú ý kí hiệu toán tử ":=" được dùng để thiết lập giá trị khởi gán.

STD_ULOGIC và STD_ULOGIC_VECTOR: có 9 giá trị logic được giới thiệu trong chuẩn IEEE 1164 chuẩn là ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'). Kí hiệu 'U' tượng trưng cho "unresolve" – không quyết đoán.

- Kiểu BOOLEAN: có 2 giá trị là TRUE và FALSE.
- Kiểu số nguyên INTEGER: là số nguyên 32 bit từ -2,147,483,657 đến +2,147, 483, 647.
- Kiểu số nguyên dương NATURAL: từ 0 đến +2,147, 483, 647.
- Kiểu số thực REAL: từ -1.0E38 đến +1.0E38. Không được tổng hợp.
- Kiểu các con số vật lý: được dùng cho các đại lượng vật lý như thời gian, điện áp, ... dùng cho mô phỏng. Không được tổng hợp.
- Kiểu SIGNED và UNSIGNED: kiểu dữ liệu đã định nghĩa trong gói ***std_logic_arith*** của thư viện IEEE.

Ví dụ 2-35: về các lệnh gán bit, vector, các kiểu hệ thống số:

```
x0 <= '0';              -- có thể xem x là bit, std_logic hoặc std_ulogic có giá trị là '0'.
x1 <= "00011111";       -- có thể xem là bit_vector, std_logic_vector, std_ulogic, signed hoặc unsigned.
x2 <= "0001_1111";      -- dấu gạch cho phép để dễ nhìn.
x3 <= "101111";         -- tượng trưng cho số nhị phân có giá trị thập phân là 47.
x4 <= B"101111";        -- tượng trưng cho số nhị phân có giá trị thập phân là 47.
x5 <= O"57";            -- tượng trưng cho bát phân có giá trị thập phân là 47.
x6 <= X"2F";            -- tượng trưng cho số thập lục phân có giá trị thập phân là 47.
n<= 1200";              -- số nguyên.
m<= 1_200";             -- số nguyên cho phép tách ra để dễ nhìn.
 IF ready THEN ...      -- ready kiểu boolean được thực hiện nếu ready bằng true.
y <= 1.2E-5;            -- số thực, nhưng không tổng hợp.
q <= d AFTER 10 ns;     -- vật lý nhưng không tổng hợp.
```

Ví dụ 2-36: về các khai báo hợp lệ và không hợp lệ với các loại dữ liệu khác nhau:

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR (7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
```

```
a       <= b(5);    -- phép gán bit thứ 5 của b cho a là hợp lệ vì cùng dữ liệu BIT.
b(0)    <= a;       -- phép gán a cho bit thứ 0 của b là hợp lệ vì cùng dữ liệu BIT.
c       <= d(5);    -- phép gán bit thứ 5 của d cho c là hợp lệ vì cùng dữ liệu STD_LOGIC.
d(0)    <= c;       -- phép gán này là hợp lệ vì cùng kiểu dữ liệu STD_LOGIC.
a       <= c;       -- phép gán này không hợp lệ vì khác kiểu dữ liệu BIT và STD_LOGIC.
b       <= d;       -- không hợp lệ vì khác kiểu BIT_VECTOR và STD_LOGIC_VECTOR.
e       <= b;       -- không hợp lệ vì khác kiểu INTEGER và BIT_VECTOR.
e       <= d;       -- không hợp lệ vì khác kiểu INTEGER và STD_LOGIC_VECTOR.
```

c. Kiểu dữ liệu do người dùng định nghĩa

VHDL cho phép người dùng định nghĩa các loại dữ liệu riêng. Hai loại dữ liệu do người dùng định nghĩa được trình bày là kiểu số nguyên INTEGER và kiểu liệt kê.

Kiểu dữ liệu số nguyên do người dùng định nghĩa:

```
TYPE   integer IS RANGE - 2147483647 TO    2147483647;
TYPE   natural IS RANGE 0       TO    2147483647;
TYPE   my_integer   IS RANGE - 32    TO    32;
TYPE   student_grade IS RANGE  0     TO    100;
TYPE   color   IS  (red, green, blue, white);
```

Kiểu dữ liệu liệt kê do người dùng định nghĩa:

```
TYPE   bit IS  ('0', '1');
TYPE   my_logic IS  ('0', '1', 'Z');
```

d. Kiểu dữ liệu SUBTYPE

SUBTYPE là kiểu dữ liệu ép kiểu. Lý do chính để dùng kiểu subtype tốt hơn là do các phép toán giữa các dữ liệu khác nhau không được phép thực hiện mà chúng chỉ cho phép thực hiện kiểu dữ subtype với các kiểu dữ liệu cơ bản khác.

Ví dụ 2-37: cho các kiểu subtype cho các kiểu dữ liệu đã định nghĩa ở trên:

```
SUBTYPE      natural IS INTEGER RANGE 0 TO INTEGER'HIGH;
SUBTYPE      my_logic IS STD_LOGIC RANGE '0' TO 'Z';
-- Gọi lại STD_LOGIC = ('X', '0', '1', 'Z', 'W', 'L', 'H', '-') nhưng my_logic = ('0', '1', 'Z').
SUBTYPE      my_color IS color RANGE red TO blue;
-- Gọi lại color =  (red, green, blue, white) nhưng my_color = (red, green,blue).
SUBTYPE      small_integer IS INTEGER RANGE -32 TO 32;
```

Ví dụ 2-38: về các phép toán hợp lệ và không hợp lệ với các kiểu dữ liệu và subtype:

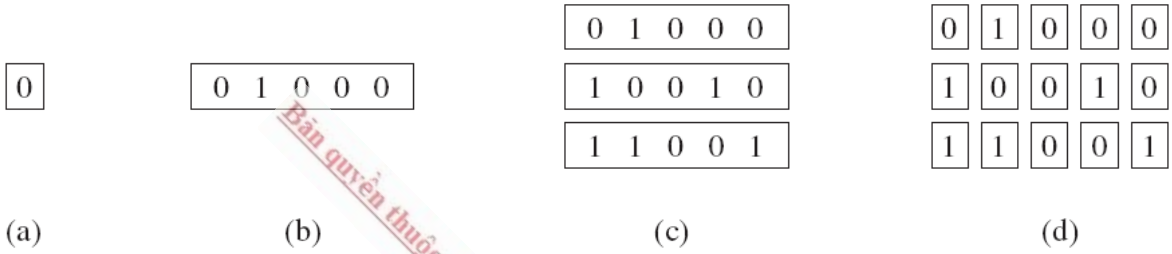
```
SUBTYPE  my_logic  IS STD_LOGIC  RANGE  '0' TO '1';
SIGNAL   a: BIT;
```

```
SIGNAL   b: STD_ LOGIC;
SIGNAL   c: my_logic ;

b   <= a;        -- Không hợp lệ vì khác kiểu dữ liệu BIT và STD_LOGIC.
b   <= c;        -- Hợp lệ vì cùng dữ liệu cơ bản STD_LOGIC.
```

e. Kiểu dữ liệu mảng ARRAY

Array là tập hợp các đối tượng cùng một kiểu dữ liệu, có thể là mảng 1 chiều (1D), mảng 2 chiều (2D) hoặc mảng 1 chiều × 1 chiều (1D × 1D). Các mảng có thể có nhiều chiều hơn nữa nhưng chúng không được tổng hợp. Hình 2-11 minh hoạ cấu trúc của mảng dữ liệu. Hình (a) là giá trị đơn (scalar), hình (b) là mảng 1 chiều (1D), hình (c) là mảng các vector (1D ×1D) và hình (d) là mảng scalar (2D).



Hình 2-11. Các kiểu mảng dữ liệu.

Các loại dữ liệu có thể tổng hợp đã định nghĩa cho các kiểu mảng ở trên là

- Dữ liệu Scalar: BIT, STD_LOGIC, STD_ULOGIC và BOOLAEN.
- Các Vector: BIT_VECTOR, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, INTEGER, SIGNED và UNSIGNED.

Cú pháp chỉ định cho một kiểu dữ liệu mảng mới như sau:

```
TYPE     type_name IS  ARRAY  (specification) OF  data_type;
```

Để sử dụng kiểu dữ liệu mảng mới này thì khai báo tín hiệu như sau:

```
SIGNAL  signal_name:  type_name   [:=initial_value];
```

Trong cú pháp ở trên tín hiệu SIGNAL được khai báo. Tuy nhiên cũng có thể là CONSTANT hoặc VARIABLE. Chú ý tuỳ chọn giá trị khởi gán chỉ được dùng cho mô phỏng.

Ví dụ 2-39: về mảng 1D×1D:

Chúng ta muốn xây dựng một mảng chứa 4 vector, mỗi vector chứa 8 bit – đây là mảng 1D×1D. Chúng ta gọi mỗi vector là một hàng và một mảng đầy đủ là một ma trận. Cách khai báo như sau:

```
TYPE     row  IS  ARRAY  (7 DONWTO 0) OF  STD_LOGIC;     -- 1D array
TYPE     matrix  IS  ARRAY  (0 TO 3) OF  row;             -- 1D×1D array
SIGNAL  x:     matrix;                                    -- 1D×1D signal
```

Ví dụ về khai báo mảng kiểu khác:

```
TYPE     matrix  IS  ARRAY  (0 TO 3) OF  STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Ví dụ về khởi gán cho mảng:

```
... :=  "0001";                                  -- gán cho mảng 1D
```

... := ('0', '0', '0', '1'); -- gán cho mảng 1D

... := (('0', '1', '1', '1'), ('1', '1', '1', '0')); -- gán cho mảng 1D×1D hoặc 2D

Ví dụ 2-40: về các phép gán hợp lệ và không hợp lệ của mảng:

Cho các mảng được khai báo như sau:

```
TYPE    row     IS  ARRAY  (7 DONWTO 0) OF  STD_LOGIC; -- 1D array
TYPE    array1  IS  ARRAY  (0 TO 3) OF  row;                    -- 1D×1D array
TYPE    array2  IS  ARRAY  (0  TO  3)  OF   STD_LOGIC_VECTOR  (7  DOWNTO  0); --
1D×1D array
TYPE    array3  IS  ARRAY  (0 TO 3, 7 DOWNTO  0) OF  STD_LOGIC;-- 2D array
SIGNAL   x: row;
SIGNAL   y: array1;
SIGNAL   v: array2;
SIGNAL   w: array3;
```

Các phép gán bit như sau:

```
x(0)      <= y(1) (2);   -- gán bit thứ 2 của vector y1 cho x(0).
x(1)      <= v(2) (1);   -- gán bit thứ 1 của vector v2 cho x(1).
x(2)      <= w(2,1);     -- gán bit thứ 1 của vector w2 cho x(2).
y(1) (1)  <= x(6);
y(2) (0)  <= v(0) (0);
```

Các phép gán vector như sau:

```
x  <= y(0) ;              --  hợp lệ vì cùng kiểu dữ liệu row.
x  <= v(1) ;              -- không hợp lệ vì không tương thích kiểu dữ liệu row×
STD_LOGIC_VECTOR.
x  <= w(2) ;              -- không hợp lệ vì không tương thích kiểu dữ liệu row×
STD_LOGIC_VECTOR.
x  <= w(2, 2 downto 0);        -- không hợp lệ vì không tương thích kiểu dữ liệu.
v(0) <= w(2, 2 downto 0);      -- không hợp lệ vì không tương thích kiểu dữ liệu.
v(0) <= w(2) ;                 -- không hợp lệ vì không tương thích kiểu dữ liệu.
y(1) <= v(3) ;                 -- không hợp lệ vì không tương thích kiểu dữ liệu.

y(1) (7 downto 3) <= x(4  downto 0);      -- hợp lệ vì cùng kiểu và kích thước dữ liệu.
v(1) (7 downto 3) <= v(2) ( 4 downto 0);  -- hợp lệ vì cùng kiểu và kích thước dữ liệu.
w(1, 5 downto 1) <= v(2) ( 4 downto 0);   -- không hợp lệ vì không cùng kiểu.
```


f. Kiểu dữ liệu mảng port

Không có các kiểu dữ liệu đã định nghĩa nhiều hơn 1 chiều. Tuy nhiên trong chỉ định khai báo PORT ngõ vào hoặc ra của mạch điện nằm trong entity, chúng ta có thể chỉ định các port

giống như các mảng vector. Do kiểu khai báo TYPE không được phép khai báo trong thực thể entity nên giải pháp để khai báo các kiểu dữ liệu do người dùng định nghĩa trong PACKAGE, sau đó có thể dùng cho toàn thiết kế.

Ví dụ 2-41:

```
-- khai báo gói package--
    LIBRARY     IEEE;
    USE         IEEE.std_logic_1164.ALL;
    PACKAGE     my_data_type      IS
      TYPE  vector_array  IS ARRAY  ( NATURAL RANGE <>) OF std_logic_vector (7
downto 0);
    END         my_data_type;
-- khai báo dữ liệu dùng cho chương trình --
    LIBRARY     IEEE;
    USE         IEEE.std_logic_1164.ALL;
    USE         WORK.my_data_type.ALL;        -- gói do người dùng định nghĩa

    ENTITY      mux  IS
     PORT       (inp:  IN     vector_array (0 to 3);
                …:             );
    END         mux;
```

g. Kiểu dữ liệu bảng ghi record

Record giống như mảng nhưng chỉ khác là chúng chứa nhiều đối tượng dữ liệu khác nhau.

Ví dụ 2-42: trình bày như sau:

```
TYPE   birthday IS  RECORD
          Day: INTEGER  RANGE  1 to 31;
          month: month_name;
END    RECORD;
```

h. Kiểu dữ liệu SIGNED và UNSIGNED

Là các dữ liệu đã định nghĩa trong gói ***std_logic_arith*** của thư viện IEEE. Cú pháp khai báo như sau:

Ví dụ 2-43:

```
SIGNAL      x:     SIGNED     (7 downto 0);
SIGNAL      y:     UNSIGNED   (0 to 3);
```

Giá trị của số không dấu thì không được nhỏ hơn 0. Ví dụ “0101” tượng trưng cho số thập phân 5, “1101” tượng trưng cho số thập phân 13. Nếu số có dấu được sử dụng thì có giá trị cả âm và dương ở dạng bù 2. Ví dụ “0101” tượng trưng cho số thập phân 5, còn “1101” tượng trưng cho số thập phân -3.

Để sử dụng các kiểu dữ liệu SIGNED và UNSIGNED thì gói ***std_logic_arith*** của thư viện IEEE phải được khai báo.

Ví dụ 2-44: các phép toán hợp lệ và không hợp lệ với các dữ liệu có dấu và không dấu:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;
```

```
USE         IEEE.std_logic_arith.ALL;  -- gói dữ liệu khai báo thêm
...
SIGNAL      a: IN       SIGNED      (7 downto 0);
SIGNAL      b: IN       SIGNED      (7 downto 0);
SIGNAL      x: OUT      SIGNED      (7 downto 0);
...
v  <= a + b ;           -- hợp lệ vì cùng kiểu dữ liệu toán học.
w <= a AND b ;          -- không hợp lệ vì không tương thích phép với phép toán logic.
```

Ví dụ 2-45: các phép toán hợp lệ và không hợp lệ với dữ liệu STD_LOGIC_VECTOR:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;
...
SIGNAL      a: IN       STD_LOGIC_VECTOR    (7 downto 0);
SIGNAL      b: IN       STD_LOGIC_VECTOR    (7 downto 0);
SIGNAL      x: OUT      STD_LOGIC_VECTOR    (7 downto 0);
...
v  <= a + b ;           -- không hợp lệ vì kiểu dữ liệu logic.
w <= a AND b ;          --  hợp lệ vì phép toán logic sử dụng kiểu logic.
```


Mặc dù bị cấm ở trên nhưng có một cách rất đơn giản cho phép dữ liệu kiểu STD_LOGIC_VECTOR tham gia trực tiếp vào các phép toán số học. Thư viện IEEE cung cấp 2 gói dữ liệu STD_LOGIC_SIGNED và STD_LOGIC_UNSIGNED cho phép các phép toán trên các dữ liệu STD_LOGIC_VECTOR có thể được thực hiện giống như các dữ liệu loại SIGNED và UNSIGNED theo thứ tự.

Ví dụ 2-46: các phép toán với dữ liệu STD_LOGIC_VECTOR:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;
USE         IEEE.std_logic_unsigned.ALL;    -- khai báo thêm .
...
SIGNAL      a: IN       STD_LOGIC_VECTOR    (7 downto 0);
SIGNAL      b: IN       STD_LOGIC_VECTOR    (7 downto 0);
SIGNAL      x: OUT      STD_LOGIC_VECTOR    (7 downto 0);
...
v  <= a + b ;           -- hợp lệ
w <= a AND b ;          --  hợp lệ
```

i. Kiểu số thực REAL

Kiểu số thực được dùng để khai báo các đối tượng sử dụng số thực. Giới hạn cực tiểu của số thực cũng được chỉ định bởi gói chuẩn trong thư viện chuẩn và nằm trong khoảng từ "-1.0E +38 đến +1.0E + 38".

Ví dụ 2-47: về các giá trị số thực như sau:

```
ARCHITECTURE    test    OF    test    IS
```

```
SIGNAL      a: REAL;
BEGIN
        a <= 1.0;           -- ok 1
        a <= 1;             -- error 2
        a <= -1.0E10;       -- ok 3
        a <= 1.5E-20;       -- ok 4
        a <= 5.3 ns;        -- error 5
END     test;
```

Hàng 1 trình bày cách gán số thực cho tín hiệu loại REAL. Tất cả các con số thực đều có dấu chấm để phân biệt với số nguyên.

Hàng thứ 2 sẽ phát sinh lỗi vì gán số nguyên cho biến kiểu số thực.

Hàng thứ 3 gán một số thực rất lớn và hàng thứ 4 gán một số thực rất nhỏ.

Hàng thứ 5 sẽ phát sinh lỗi vì không thể gán thời gian cho tín hiệu số thực.

j. Kiểu liệt kê

Kiểu liệt kê là một công cụ hỗ trợ đắc lực cho quá trình thiết kế bằng ngôn ngữ VHDL.

Người thiết kế có thể dùng các loại dữ liệu liệt kê đại diện chính xác cho các giá trị chính xác được yêu cầu cho phép toán chỉ định. Tất cả các giá trị của kiểu dữ liệu liệt kê do người dùng định nghĩa. Các giá trị này có thể là tên hoặc các hằng số đặc tính đơn, ví dụ cho tên là x, abc, ... hằng số đặc tính đơn là 'X', '1' và '0'.

Loại dữ liệu liệt kê cho hệ thống có 4 giá trị mô phỏng như sau:

```
TYPE    fourval     IS      ('X', '0', '1', 'Z');
```

Một ứng dụng có dùng kiểu liệt kê là minh hoạ cho tất cả các lệnh của vi xử lý. Ví dụ 2-48 kiểu liệt kê cho một vi xử lý đơn giản như sau:

Ví dụ 2-48:

```
        TYPE    instruction     IS      (add, sub, lda, ldb, sta, stb, outa, xfr);
```

Và mô hình cho hệ thống vi xử lý:

```
PACKAGE     instr   IS
        TYPE        instruction     IS      (add, sub, lda, ldb, sta, stb, outa, xfr);
END         instr;

USE  work.instr.ALL;
ENTITY      mp      IS
  PORT      (instr:     IN          instruction;
             addr:      IN          INTEGER;
             data:      INOUT       INTEGER);
END         mp;

ARCHITECTURE    mp      OF      mp      IS
BEGIN
    PROCESS  (instr)
        TYPE regtype        IS      ARRAY (0 to 255 ) OF INTEGER;
        VARIABLE    a, b:   INTEGER;
        VARIABLE    reg:    regtype;
    BEGIN
```

```
                CASE  instr   IS
                        WHEN  lda   =>    a:= data;          -- load a accumulator
                        WHEN  ldb   =>    b:= data;          -- load b accumulator
                        WHEN  add   =>    a:= a + b;         -- add accumulator
                        WHEN  sub   =>    a:= a - b;         -- subtract accumulator
                        WHEN  sta   =>    reg(addr) :=  b;   -- put b accu in reg array
                        WHEN  out   =>    data<= a ;         -- output a accumulator
                        WHEN  xfr   =>    a:= b ;            -- transfer b to a
                END CASE;
        END PROCESS ;
END     mp;
```

Mô hình nhận một chuỗi lệnh (**instr**), một địa chỉ (**addr**) và một chuỗi dữ liệu (**data**). Dựa vào giá trị của **instr** được liệt kê mà lệnh tương ứng được thực hiện. Phát biểu **CASE** được dùng để lựa chọn lệnh để thực hiện. Phát biểu được thực hiện và sau đó quá trình sẽ đợi cho đến lệnh kế.

2. KIỂU VẬT LÝ:

Kiểu vật lý được sử dụng để mô tả các đại lượng vật lý như: khoảng cách, dòng điện, thời gian ... Ví dụ 2-49 về kiểu dữ liệu vật lý về dòng điện như sau:

Ví dụ 2-49:

```
TYPE    current     IS  RANGE 0 TO 1000000000;
UNITS
        na;             -- nano amps
        ua = 1000 na;   -- micro amps
        ma = 1000 ua;   -- mili amps
         a  = 1000 ma;  -- amps
END     UNITS;
```


Việc xác định kiểu dữ liệu được bắt đầu với câu lệnh khai báo tên kiểu và vùng của kiểu (0 to 1000000000), các khai báo được thực hiện trong đoạn UNITS. Trong ví dụ trên đơn vị chính của UNITS là na. Sau khi đơn vị chính của UNITS được khai báo các đơn thể khác sẽ được xác định.

Kiểu vật lý đã được định nghĩa: trong VHDL có kiểu vật lý đã được định nghĩa là thời gian như sau:

```
TYPE    TIME  IS    <implementation defined>;
UNITS
        fs;             -- femtosecond
        ps = 1000 fs;   -- picosecond
        ns = 1000 ps;   -- nanosecond
        us = 1000 ns;   -- microsecond
        ms = 1000 ns;   -- milisecond
        sec = 1000 ms;  -- second
        min = 60 sec;   -- minute
        hr =  60 min;   -- hour
END     UNITS;
```

3. CÁC THUỘC TÍNH:

VHDL hỗ trợ 5 loại thuộc tính. Các thuộc tính đã định nghĩa luôn được áp dụng tiếp đầu ngữ như tên của tín hiệu, tên của biến hoặc kiểu. Các thuộc tính được dùng để trả về nhiều loại thông tin khác nhau như tín hiệu, biến hoặc kiểu. Các thông tin chứa dấu phẩy (') theo sau là tên của thuộc tính.

a. Thuộc tính tín hiệu

Bảng sau đây liệt kê các thuộc tính của tín hiệu:

Thuộc tính	Chức năng
signal_name**'event**	Trả về giá trị Boolean là True nếu có sự kiện trên tín hiệu xảy ra, ngược lại thì trả về giá trị false.
signal_name**'active**	Trả về giá trị Boolean là True nếu có tích cực (gán) trên tín hiệu xảy ra, ngược lại thì trả về giá trị false.
signal_name**'transaction**	Trả về tín hiệu kiểu "bit" lật trạng thái (0 sang 1 hoặc 1 sang 0) mỗi lần có chuyển trạng thái trên tín hiệu.
signal_name**'last_event**	Trả về giá trị khoảng thời gian từ khi xảy ra sự kiện sau cùng trên tín hiệu.
signal_name**'last_active**	Trả về giá trị khoảng thời gian từ khi xảy ra mức tích cực trên tín hiệu.
signal_name**'last_value**	Cung cấp giá trị của tín hiệu trước khi sự kiện sau cùng xảy ra trên tín hiệu.
signal_name**'delayed(T)**	Cung cấp tín hiệu trễ đi T lần so với tín hiệu gốc. T là tuỳ chọn, mặc nhiên T = 0.
signal_name**'stable(T)**	Trả về giá trị Boolean, là true nếu không có sự kiện xảy ra trên tín hiệu trong khoảng thời gian T, ngược lại thì trả về giá trị false. T là tuỳ chọn, mặc nhiên T = 0.
signal_name**'quiet(T)**	Trả về giá trị Boolean, là true nếu không có sự thay đổi xảy ra trên tín hiệu trong khoảng thời gian T, ngược lại là false. T là tuỳ chọn và mặc nhiên T = 0.

Bảng 2-1. Thuộc tính tín hiệu.

Ví dụ 2-50: về các thuộc tính:

if (CLOCK**'event and** CLOCK= '1') **then** …

Biểu thức này kiểm tra sự xuất hiện của xung clock cạnh lên. Để tìm khoảng thời gian từ khi có xung clock cạnh lên sau cùng thì dùng thuộc tính sau:

CLOCK**'last_event**

b. Thuộc tính dữ liệu scalar

Một vài thuộc tính kiểu dữ liệu scalar được hỗ trợ như bảng sau:

Thuộc tính	Giá trị
scalar_type**'left**	Trả về giá trị đầu tiên hoặc giá trị tận cùng bên trái của kiểu dữ liệu scalar trong kiểu đã định nghĩa.
scalar_type**'right**	Trả về giá trị sau cùng hoặc giá trị tận cùng bên phải của kiểu dữ liệu scalar trong kiểu đã định nghĩa.

scalar_type**'low**	Trả về giá trị thấp nhất của kiểu dữ liệu scalar trong kiểu đã định nghĩa.
scalar_type**'high**	Trả về giá trị cao nhất của kiểu dữ liệu scalar trong kiểu đã định nghĩa
scalar_type**'ascending**	Là true nếu T là dãy tăng ngược lại thì false.
scalar_type'**value(s)**	Trả về giá trị của T được tượng trưng bởi s (string value).

Bảng 2-2. Thuộc tính dữ liệu scalar.

Ví dụ 2-51: về các thuộc tính:

```
Type conductance is range  1E-6 to  1E3
Units mho;
End units conductance;
Type my_index is range  3 to  15;
Type my_levels is (low, high, dontcare, highZ);

conductance'right          trả về      1E3
conductance'high                       1E3
conductance'low                        1E-6
my_index'left                          3
my_index'value(5)                      "5"
my_levels'left                         low
my_levels'low                          low
my_levels'high                         highZ
my_levels'value(dontcare)              "dontcare"
```


c. Thuộc tính mảng

Bằng cách dùng các thuộc tính mảng sẽ trả về giá trị chỉ số tương ứng với dãy của mảng. Các thuộc tính được xây dựng như sau:

Thuộc tính	Trả về
MATRIX'**left**(N)	Chỉ số phần tử tận cùng bên trái
MATRIX'**right**(N)	Chỉ số phần tử tận cùng bên phải
MATRIX'**high**(N)	Giới hạn trên
MATRIX'**low**(N)	Giới hạn dưới
MATRIX'**length**(N)	Số lượng các phần tử
MATRIX'**range**(N)	Dãy
MATRIX'**reverse_range**(N)	Dãy bảo vệ
MATRIX'**ascending**(N)	Trả về giá trị true nếu chỉ số theo thứ tự tăng, ngược lại thì bằng false.

Bảng 2-3. Thuộc tính mảng.

Con số N nằm trong dấu ngoặc được xem như chiều của mảng. Đối với mảng 1 chiều thì có thể bỏ nhưng đối với mảng 2 chiều thì phải dùng con số N để chỉ rõ. Các ví dụ về thuộc tính mảng như sau:

Ví dụ 2-52: về các thuộc tính:

```
Type myarr8x4        is range  (8 downto 1, 0 to 3) of  boolean;
```

Type myarr1 **is range** (-2 to 4) **of** integer;

MYARR1'left	trả về:	-2
MYARR1'right		4
MYARR1'high		4
MYARR1'reverse_range		4 downto -2
MYARR8x4'left(1)		8
MYARR8x4'left(2)		0
MYARR8x4'right(2)		3
MYARR8x4'high(1)		8
MYARR8x4'low(1)		1
MYARR8x4'ascending(1)		False

VIII. CÁC TOÁN TỬ CƠ BẢN TRONG VHDL

VHDL hổ trợ 7 loại toán tử khác nhau để xử lý các tín hiệu, biến và hằng số. Các loại toán tử được liệt kê như sau:

Thứ tự	Loại						
1	Toán logic	**and**	**or**	**nand**	**nor**	**xor**	**xnor**
2	Toán tử quan hệ	=	/=	<	<=	>	>=
3	Toán tử dịch	**sll**	**srl**	**sla**	**sra**	**rol**	**ror**
4	Toán tử số học	+	-	&			
5	Toán tử không xác định	+	-				
6	Toán tử nhân chia	*	/	**mod**	**rem**		
7	Toán tử hỗn hợp	**	**abs**	**not**			

Bảng 2-4. Tất cả các toán tử.

Thứ tự ưu tiên cao nhất cho toán tử thứ 7, tiếp theo là thứ 6 và thấp nhất là toán tử thứ 1. Trừ trường hợp dấu ngoặc được sử dụng thì toán tử có thứ tự ưu tiên cao nhất sẽ được thực hiện trước. Nếu các toán tử cùng thứ tự ưu tiên thì các toán tử sẽ được thực hiện từ trái sang phải của biểu thức.

Ví dụ 2-53: Cho các dữ liệu như sau: X (='010'), Y(='10'), and Z (='10101') đều thuộc kiểu std_ulogic_vectors.

not X & Y **xor** Z **rol** 1

thì sẽ tương đương với ((**not** X) & Y) **xor** (Z **rol** 1) = ((101) & 10) **xor** (01011) =(10110) **xor** (01011) = 11101.

1. CÁC TOÁN TỬ LOGIC:

Toán tử logic (**And, Or, Nand, Nor, Xor** và **Xnor**) được dùng cho các loại dữ liệu "bit", "boolean", "std_logic", "std_ulogic" và các vector. Các toán tử này được dùng để xác định biểu thức logic Boolean hoặc thực hiện các phép toán bit với bit trên một mảng bit. Kết quả cùng kiểu

dữ liệu như các tác tố (Bit hoặc Boolean). Các toán tử này có thể được áp dụng cho các tín hiệu, các biến và các hằng số.

Chú ý: các toán tử nand và nor không thể kết hợp. Phải sử dụng dấu ngoặc để chia các toán tử nand và nor để không phát sinh lỗi khi biên dịch:

X **nand** Y **nand** Z sẽ phát sinh lỗi và phải viết như sau (X **nand** Y) **nand** Z.

2. CÁC TOÁN TỬ QUAN HỆ:

Toán tử quan hệ kiểm tra các giá trị quan hệ của 2 loại dữ liệu scalar và cho kết quả là kiểu Boolean true hoặc false.

Toán tử	Mô tả	Kiểu toán tử	Kiểu kết quả
=	Bằng	Bất kỳ	Boolean
/=	Không bằng	Bất kỳ	Boolean
<	Nhỏ hơn	Kiểu scalar hoặc mảng rời rạc	Boolean
<=	Nhỏ hơn hoặc bằng	Kiểu scalar hoặc mảng rời rạc	Boolean
>	Lớn hơn	Kiểu scalar hoặc mảng rời rạc	Boolean
>=	Lớn hơn hoặc bằng	Kiểu scalar hoặc mảng rời rạc	Boolean

Bảng 2-5. Các toán tử quan hệ.

Toán tử quan hệ kiểm tra các giá trị quan hệ của 2 loại dữ liệu scalar và cho kết quả là kiểu Boolean true hoặc false.

Chú ý: kí hiệu "<=" (nhỏ hơn hay bằng) giống như kí hiệu của phép gán giá trị cho tín hiệu.

Ví dụ 2-54: về các thuộc tính:

```
Variable  STS :       boolean;
Constant  A   :       integer:= 24;
Constant  B_count:    integer:= 32;
Constant  C   :       integer:= 14;
STS <= (A < B_count);                                  -- gán giá trị "true" cho STS
STS <= ((A >= B_count) or (A>C));                      -- gán giá trị "true" cho STS
STS <= (std_logic('1','0','1')<std_logic('0','1','1')); -- gán giá trị "false" cho STS

Type new_std_logic    is ('0', '1', 'Z', '-');
Variable  A1  :       new_std_logic := '1';
Variable  A2  :       new_std_logic := 'Z';
STS <= (A1 < A2);                  -- gán giá trị "true" cho STS vì '1' nằm bên trái của 'Z'
```

3. CÁC TOÁN SỐ HỌC:

Toán tử số học được dùng để thực hiện các phép toán cộng và trừ trên các tác tố của bất kì kiểu dữ liệu nào. Toán tử **&** được dùng để nối hai vector tạo thành 1 vector dài hơn. Để dùng các toán tử số học thì phải khai báo các thư viện std_logic_unsigned.all hoặc std_logic_arith package vào thư viện std_logic_1164 package.

Toán tử	Mô tả	Tác tố bên trái	Tác tố bên phải	Kết quả

+	Phép cộng	Kiểu số	Giống tác tố bên trái	Cùng kiểu
-	Phép trừ	Kiểu số	Giống tác tố bên trái	Cùng kiểu
&	Nối dài	Kiểu mảng hoặc phần tử	Giống tác tố bên trái	Cùng kiểu mảng

Bảng 2-6. Các toán tử số học.

Ví dụ 2-55: cho các tín hiệu
signal MYBUS :std_logic_vector (15 **downto** 0);
signal STATUS :std_logic_vector (2 **downto** 0);
signal RW, CS1, CS2 :std_logic;
signal MDATA :std_logic_vector (0 **to** 9);
Thực hiện: MYBUS <= STATUS **&** RW **&** CS1 **&** SC2 **&** MDATA;

MYARRAY (15 **downto** 0) <= "1111_1111" **&** MDATA (2 **to** 9);
NEWWORD <= "VHDL" **&** "93";

4. CÁC TOÁN TỬ CÓ DẤU:

Toán tử "+" và "-" được dùng để chỉ định dấu của dữ liệu số

Toán tử	Mô tả	Kiểu dữ liệu tác tố	Kiểu dữ liệu kết quả
+	Số dương	Bất kỳ dữ liệu số nào	Cùng kiểu
-	Số âm	Bất kỳ dữ liệu số nào	Cùng kiểu

Bảng 2-7. Các toán tử có dấu.

5. CÁC TOÁN TỬ NHÂN CHIA:

Toán tử nhân được dùng để thực hiện các hàm toán học trên các kiểu dữ liệu số nguyên hoặc kiểu dấu chấm động.

Tác tố	Mô tả	Kiểu dữ liệu tt trái	Kiểu dữ liệu tt phải	Kiểu KQ
*	Nhân	kiểu integer và DCĐ	Cùng kiểu	Cùng kiểu
		kiểu vật lý	Kiểu số nguyên hoặc thực	Cùng kiểu tt trái
		kiểu số nguyên hoặc thực	Kiểu vật lý	Cùng kiểu tt phải
/	Chia	Số nguyên hoặc dấu chấm động	Số nguyên hoặc dấu chấm động	Cùng kiểu
		Kiểu vật lý	Số nguyên hoặc số thực	Cùng kiểu tt trái
		Kiểu vật lý	Cùng kiểu	Số nguyên
mod	Chia nguyên	Kiểu số nguyên		Cùng kiểu
rem	Remainder	Kiểu số nguyên		Cùng kiểu

Bảng 2-8. Các toán tử nhân chia.

Ví dụ 2-56: cho các tín hiệu
11 **rem** 4 kết quả bằng 3
(-11) **rem** 4 kết quả bằng -3
9 **mod** 4 kết quả bằng 1
7 **mod** (-4) kết quả bằng –1 (7 – 4*2 = -1).

6. CÁC TOÁN TỬ DỊCH:

Toán tử thực hiện dịch chuyển từng bit hoặc xoay từng bit trên dữ liệu mảng 1 chiều của các phần tử kiểu dữ liệu bit hoặc std_logic hoặc Boolean.

Toán tử	Mô tả	Kiểu dữ liệu tác tố
sll	Dịch sang trái – lắp đầy bằng các bit 0. (Shift left logical)	Mảng 1 chiều với các phần tử mảng kiểu bit hoặc Boolean; Right: integer
srl	Dịch sang phải – lắp đầy bằng các bit 0. (Shift right logical)	Giống như trên
sla	Dịch sang trái lắp đầy bằng bit tận cùng bên phải. (Shift left arithmetic)	Giống như trên
sra	Dịch sang phải lắp đầy bằng bit tận cùng bên trái. (Shift right arithmetic)	Giống như trên
rol	Xoay vòng tròn sang trái (Rotate left circular)	Giống như trên
ror	Xoay vòng tròn sang phải (Rotate right circular)	Giống như trên

Bảng 2-9. Các toán tử dịch.

Toán tử nằm bên trái toán tử và số lần dịch nằm bên phải toán tử – xem ví dụ 2-57:

Ví dụ 2-57: Cho **variable** NUM1 :bit_vector := "10010110";

Thực hiện NUM1 **srl** 2;
Kết quả NUM1 = "00100101".

Khi số lần dịch là số âm thì hoạt động dịch xảy ra theo chiều ngược lại, dịch trái sẽ trở thành dịch phải.

Ví dụ 2-58: Cho **variable** NUM1 :bit_vector := "10010110";
NUM1 **srl** –2 sẽ tương đương với NUM1 **sll** 2 và cho kết quả là "01011000".

Ví dụ 2-59: Cho **variable** A: bit_vector := "101001";
A **sll** 2 results in "100100"
A **srl** 2 results in "001010"
A **sla** 2 results in "100111"
A **sra** 2 results in "111010"
A **rol** 2 results in "100110"
A **ror** 2 results in "011010"

7. CÁC TOÁN HỖN HỢP:

Toán tử hỗn hợp gồm toán tử trị tuyệt đối và toán tử số mũ có thể áp dụng cho các kiểu dữ liệu số. Toán tử not sẽ cho cùng giá trị nhưng ngược dấu.

Toán tử	Mô tả	Dữ liệu bên trái	Dữ liệu bên phải	Dữ liệu kết quả
**	Hàm mũ	Số nguyên	Số nguyên	Giống dữ liệu bên trái
		Dấu chấm	Số nguyên	Giống dữ liệu bên trái
abs	Hàm trị tuyệt đối	Kiểu số		Cùng kiểu
not	Hàm phủ định	Kiểu bit hoặc Boolean		Cùng kiểu

Bảng 2-10. Các toán tử hỗn hợp.

IX. CHƯƠNG TRÌNH CON VÀ GÓI

1. CHƯƠNG TRÌNH CON:

Chương trình con bao gồm các thủ tục và các hàm. Thủ tục có thể trả về nhiều hơn 1 đối số, còn hàm thì chỉ trả về 1 đối số. Trong một hàm thì tất cả các thông số đều là thông số ngõ vào, trong thủ tục có thể là thông số ngõ vào, thông số ngõ ra và có thể là so vào ra.

Có hai kiểu cho hàm và thủ tục: thủ và hàm đồng thời, thủ tục và hàm tuần tự. Hàm và thủ tục đồng thời chỉ tồn tại nằm bên ngoài phát biểu quá trình hoặc một chương trình con; hàm và thủ tục tuần tự chỉ tồn tại trong phát biểu quá trình hoặc trong 1 chương trình con khác.

Tất cả các phát biểu nằm bên trong một chương trình con là tuần tự. Các phát biểu giống nhau tồn tại trong một phát biểu quá trình thì có thể được dùng trong một chương trình con bao gồm cả phát biểu WAIT.

a. Hàm - FUNCTION:

Ví dụ 2-60 là một hàm, hàm này nhận vào một mảng có kiểu STD_LOGIC và trả về một giá trị số nguyên. Giá trị số nguyên này biểu diễn giá trị số học của tất cả các bit được xử lý dưới dạng số nhị phân:

Ví dụ 2-60: cách viết hàm

```
LIBRARY      IEEE;
USE          IEEE.std_logic_1164.ALL;

PACKAGE      num_type     IS
  TYPE log8  IS      ARRAY (0 TO 7) OF std_logic;     -- line 1
END          num_type;


USE  work.num_type.ALL;
ENTITY       convert      IS
   PORT      (I1:         IN          log8;          -- line 2
              O1:         OUT         INTEGER);      -- line 3
END          convert;

ARCHITECTURE      behave  OF   convert   IS

FUNCTION   vector_to_int(S:log8)                     -- line 4
  RETURN   INTEGER is                                -- line 5
  VARIABLE  result: INTEGER := 0                     -- line 6
  BEGIN
             For i    IN     0 TO 7 LOOP             -- line 7
                    result  := result * 2;           -- line 8
                    IF  S(i)  = '1'   THEN           -- line 9
                          result  := result + 1;     -- line 10
                    END IF  ;
             END   LOOP;
             RETURN      result ;                    -- line 11
  END   vector_to_int;

  BEGIN
        O1 <= vector_to_int (I1);

  END behave;
```

Dòng 1 khai báo kiểu mảng được sử dụng cho toàn bộ chương trình.

Các dòng 2 và 3 khai báo các port ngõ vào, ngõ ra của thực thể convert và các kiểu dữ liệu.

Các dòng từ 4 đến 11 mô tả một hàm được khai báo trong miền khai báo của kiến trúc behave. Bằng cách khai báo hàm trong miền khai báo của kiến trúc, hàm này sẽ được sử dụng bất kỳ miền nào của kiến trúc.

Các dòng 4 và 5 khai báo tên của hàm, các đối số của hàm và kiểu mà hàm trả về.

Ở dòng 6 một biến cục bộ của hàm được khai báo. Các hàm có các miền khai báo rất giống với các phát biểu quá trình. Các biến, các hằng và các kiểu có thể được khai báo nhưng tín hiệu thì không.

Các dòng từ 7 đến 10 khai báo một phát biểu vòng lặp cho mỗi giá trị trong mảng. Giải thuật cơ bản của hàm này là dịch và cộng với mỗi vị trí bit trong mảng. Đầu tiên kết quả được dịch (tức là nhân 2) và tiếp theo là vị trí bit là 1 thì giá trị 1 được cộng vào kết quả.

Ở cuối phát biểu vòng lặp, biến result sẽ chứa giá trị nguyên của mảng được chuyển vào. Giá trị của hàm được chuyển ngược về thông qua phát biểu RETURN. Trong ví dụ trên thì phát biểu RETURN được trình bày ở dòng 11.

Cuối cùng, dòng 12 trình bày cách thức một hàm được gọi. Tên của hàm được theo sau bởi các đối số của hàm ở trong hai ngoặc đơn. Hàm sẽ luôn luôn trả về một giá trị, do vậy quá trình gọi, phát biểu đồng thời, ... phải có một vị trí để hàm có thể trả về giá trị này. Trong ví dụ này, ngõ ra của hàm được gán cho một port ngõ ra.

Các thông số của hàm là dữ liệu nhập. Không có phép gán nào được thực hiện cho bất kỳ thông số nào của hàm. Trong ví dụ trên các thông số thuộc loại hằng số do không có loại rõ ràng được chỉ định và mặc định là hằng. Các đối số được xử lý như thể chúng là các hằng được khai báo trong miền khai báo của hàm.

Loại thông số của hàm có thể là thông số tín hiệu. Với một thông số tín hiệu, các thuộc tính của tín hiệu được chuyển vào trong hàm và sẵn sàng được sử dụng. Ngoại lệ đối với phát biểu thuộc tính 'STABLE, 'QUIET, 'TRANSACTION, và 'DELAYED sẽ tạo ra các tín hiệu đặc biệt.

Một ví dụ 2-61 cho thấy một hàm chứa các thông số tín hiệu như sau:

Ví dụ 2-61:

```
LIBRARY     IEEE;
USE         IEEE.std_logic_1164.ALL;
ENTITY      dff     IS
  PORT      (d, clk:      IN          std_logic;
               q:         OUT         std_logic);
FUNCTION  resing_edge (SINGAL S : std_logic)      -- line 1
  RETURN  BOOLEAN  IS                             -- line 2
  BEGIN
        IF  (S'EVENT) AND (S='1') AND             -- line 3
            (S'LAST_VALUE = '0')  THEN            -- line 4
              RETURN   TRUE;                      -- line 5
        ELSE  RETURN   FALSE;                     -- line 6
        END IF;
  END resing_edge;
END         dff;

ARCHITECTURE     behave  OF   dff   IS

  BEGIN
        PROCESS (CLK)
        BEGIN
              IF  rising_edge(clk)  THEN          -- line 7
                    q  <=  d;                     -- line 8
```

```
            END IF;
        END PROCESS;
    END behave;
```

Ví dụ này cung cấp phương pháp phát hiện cạnh lên cho mô hình flip flop D. Hàm khai báo trong phần khai báo thực thể và do vậy có thể sử dụng cho bất kỳ kiến trúc nào của thực thể này.

Các dòng 1 và 2 cho thấy khai báo hàm. Chỉ có một thông số (S) cho hàm và thông số này thuộc loại tín hiệu.

Các dòng 3 và 4 là một phát biểu IF, phát biểu này xác định có phải tín hiệu vừa thay đổi hay không, có phải giá trị hiện tại là '1' hay không và có phải giá trị trước đó là '0' hay không. Nếu tất cả các điều kiện này là đúng, phát biểu IF sẽ trả về giá trị đúng (true), có nghĩa là một cạnh tăng đã được phát hiện trên tín hiệu. Nếu một điều kiện nào đó trong các điều kiện này không đúng, giá trị được trả về sẽ là sai (false), như được trình bày ở dòng 6.

Dòng 7 gọi hàm sử dụng tín hiệu được tạo ra bởi port clk của thực thể *dff*. Nếu có một cạnh tăng trên tín hiệu clk, giá trị của *d* được chuyển đến ngõ ra *q*.

Công dụng phổ biến nhất của hàm là trả về một giá trị trong một biểu thức, tuy nhiên còn có hai công dụng nữa có sẵn trong VHDL. Công dụng đầu tiên là hàm chuyển đổi (conversion function) và công dụng thứ hai là hàm phân tích (resolution function). Các hàm chuyển đổi được sử dụng để chuyển đổi từ kiểu này sang kiểu khác. Các hàm phân tích được sử dụng để phân tích việc tranh chấp bus trên một tín hiệu có nhiều nguồn kích (multiply-driven signal).

b. Hàm chuyển đổi:

Các hàm chuyển đổi được sử dụng để chuyển đổi một đối tượng có kiểu này thành đối tượng có kiểu khác. Các hàm chuyển đổi được sử dụng trong các phát biểu thể hiện thành phần để cho phép việc ánh xạ các tín hiệu và port có các kiểu khác nhau. Loại tình huống này thường phát sinh khi một người thiết kế muốn sử dụng một thực thể từ một thiết kế khác.

Giả định rằng người thiết kế A đang sử dụng kiểu dữ liệu có 4 giá trị như sau:

```
TYPE fourval IS (X, L, H, Z);
```

Người thiết kế B đang sử dụng kiểu dữ liệu cũng chứa 4 giá trị nhưng các định danh giá trị lại khác, như được trình bày sau đây

```
TYPE fourvalue IS ('X', '0', '1', 'Z');
```

Cả hai kiểu này đều có thể được sử dụng để biểu diễn các trạng thái của một hệ thống giá trị 4-trạng thái cho một mô hình của VHDL. Nếu người thiết kế A muốn sử dụng một mô hình từ người thiết kế B, nhưng người thiết kế B đã sử dụng các giá trị từ kiểu *fuorvalue* làm các port giao diện của mô hình, người thiết kế A không thể sử dụng mô hình này mà không chuyển đổi kiểu của các port thành các giá trị được sử dụng bởi người thiết kế B. Vấn đề này có thể giải quyết được thông qua việc sử dụng các hàm chuyển đổi.

Trước tiên ta hãy viết hàm chuyển đổi giá trị giữa hai hệ thống.

Các giá trị từ hệ thống thứ nhất biểu diễn các trạng thái phân biệt:

X – giá trị chưa biết.

L – giá trị logic 0.

H – giá trị logic 1.

Z – giá trị tổng trở cao.

Các giá trị từ hệ thống thứ hai biểu diễn các trạng thái:

'X' – giá trị chưa biết.

'0' – giá trị logic 0.

'1' – giá trị logic 1.

'Z' – giá trị tổng trở cao.

Từ mô tả trên của hai hệ thống giá trị ta có một ví dụ về hàm chuyển đổi như sau:

Ví dụ 2-62:

```
FUNCTION   convert4val (S : fourval)  RETURN   fourvalue  IS
        BEGIN
        CASE  S IS
                WHEN X =>   RETURN   'X';
                WHEN L =>   RETURN   '0';
                WHEN H =>   RETURN   '1';
                WHEN Z =>   RETURN   'Z';
        END CASE ;
  END convert4val;
```

Hàm này sẽ nhận một giá trị có kiểu *fourval* và trả về một giá trị có kiểu *fourvalue*. Ví dụ sau đây cho thấy nơi mà hàm có thể được sử dụng:

Ví dụ 2-63:

```
PACKAGE     my_std      IS
  TYPE fourval      IS    (X, L, H, Z) ;
  TYPE fourvalue    IS    ('X', 'L', 'H', 'Z') ;
  TYPE fvector4     IS    ARRAY (0 TO 3) OF fourval;

END           my_std;

USE           WORK.my_std.ALL;
ENTITY        reg    IS
   PORT       ( a    IN          fvector4;
                clr:  IN          fourval;
                clk:  IN          fourval;
                q:    OUT         fvector4);

     FUNCTION   convert4val (S : fourval)  RETURN   fourvalue  IS
        BEGIN
                CASE  S IS
                        WHEN X =>   RETURN   'X';
                        WHEN L =>   RETURN   '0';
```

```
                WHEN H =>   RETURN   '1';
                WHEN Z =>   RETURN   'Z';
            END CASE ;
        END convert4val;

        FUNCTION   convert4value (S : fourvalue)  RETURN   fourval IS
          BEGIN
            CASE  S IS
                WHEN 'X' =>   RETURN   X;
                WHEN '0' =>   RETURN   0;
                WHEN '1' =>   RETURN   1;
                WHEN 'Z' =>   RETURN   Z;
            END CASE ;
          END convert4value;
    END reg;

ARCHITECTURE    structure  OF   reg IS
    COMPONENT       dff
     PORT       ( d, clk, clr:    IN      fourvalue;
                  q:     OUT      fourvalue;);
    END COMPONENT;

    BEGIN
        U1:   dff    PORT MAP ( convert4val(a(0)),
                                convert4val(clk),
                                convert4val(clr),
                                convert4value(q) => q(0));

        U2:   dff    PORT MAP ( convert4val(a(1)),
                                convert4val(clk),
                                convert4val(clr),
                                convert4value(q) => q(1));

        U3:   dff    PORT MAP ( convert4val(a(2)),
                                convert4val(clk),
                                convert4val(clr),
                                convert4value(q) => q(2));

        U4:   dff    PORT MAP ( convert4val(a(3)),
                                convert4val(clk),
                                convert4val(clr),
                                convert4value(q) => q(3));

    END structure;
```

Ví dụ này là một thanh ghi 4 bit được xây dựng bằng các flip flop. Kiểu được sử dụng trong khai báo thực thể cho thanh ghi là một vector kiểu *fourval*. Tuy nhiên các flip flop được thể hiện có các port có kiểu *fourvalue*. Một lỗi không tương thích kiểu sẽ được tạo ra nếu các port của thực thể thanh ghi được ánh xạ trực tiếp đến các port thành phần. Do vậy một hàm chuyển đổi được cần đến để chuyển đổi hai hệ thống giá trị.

Nếu các port đều ở chế độ IN thì chỉ một chuyển đổi được cần đến để ánh xạ từ kiểu của thực thể chứa đến kiểu của thực thể được chứa. Trong ví dụ này, nếu các port đều ở chế độ ngõ vào thì chỉ có hàm *convert4value* được yêu cầu.

Nếu thành phần cũng có các port ngõ ra, các giá trị ngõ ra của thực thể được chứa cần được chuyển đổi trả về kiểu của thực thể chứa. Trong ví dụ này port q của thành phần *dff* là một port ngõ ra. Kiểu của các giá trị ngõ ra. Kiểu của các giá trị ngõ ra sẽ là *fourvalue*. Các giá trị này không thể được ánh xạ đến các port kiểu *fourval*. Hàm *convert4value* sẽ chuyển đổi từ kiểu *fourvalue* thành kiểu *fourval*. Áp dụng hàm này trên các port ngõ ra sẽ cho phép ánh xạ port xảy ra.

Có 4 thể hiện thành phần sử dụng các hàm chuyển đổi này: các thành phần từ U1 đến U4. Lưu ý rằng các port ngõ vào sử dụng hàm chuyển đổi *convert4val* trong khi các port ngõ ra sử dụng hàm chuyển đổi *convert4value*.

Dùng dạng kết hợp đặt này của ánh xạ cho thể hiện thành phần U1 như sau:

```
U1:  dff   PORT MAP ( d   => convert4val(a(0)),
                      clk => convert4val(clk),
                      clr  => convert4val(clk),
                      convert4value(p) => p(0));
```

Các hàm chuyển đổi giải phóng người thiết kế khỏi việc tạo ra nhiều tín hiệu hoặc biến tạm thời để thể hiện việc chuyển đổi. Ví dụ 2-64 trình bày một phương pháp khác để thực hiện các hàm chuyển đổi:

Ví dụ 2-64:

```
Temp1       <=     convert4val(a(0));
Temp2       <=     convert4val(clk));
Temp3       <=     convert4val(clr);

   U1:  dff   PORT MAP ( d   => temp1,
                         clk => temp2,
                         clr  => temp3,
                         q => temp4);,
q(0)   <=  convert4value(temp4);
```

Phương pháp này dài dòng, yêu cầu một biến tạm trung gian cho mỗi port của thành phần được ánh xạ. Phương pháp ít được lựa chọn.

Nếu một port ở chế độ INOUT, các hàm chuyển đổi không thể thực hiện được với ký hiệu vị trí. Các port phải sử dụng kết hợp đặt tên do hai hàm chuyển đổi phải được kết hợp với port

INOUT. Một hàm chuyển đổi sẽ được sử dụng cho phần ***ngõ vào*** của port INOUT và một hàm khác sẽ được sử dụng cho phần ***ngõ ra*** của port nhập/xuất.

Trong ví dụ sau đây, linh kiện truyền hai chiều được chứa trong thực thể có tên là *trans2*:

Ví dụ 2-64:

```
PACKAGE     my_pack     IS
  TYPE nineval       IS     (Z0, Z1, ZX, R0, R1, RX, F0, F1, FX) ;
  TYPE nvector2      IS     ARRAY (0 TO 1) OF  (nineval) ;
  TYPE fourstate     IS     (X, L, H, Z);

    FUNCTION  convert4state (a : fourstate)  RETURN   nineval;
    FUNCTION  convert9val(a : nineval)  RETURN   fourstate;
END my_pack;

PACKAGE     body   my_pack      IS
    FUNCTION  convert4state (a : fourstate)  RETURN   nineval IS;
      BEGIN
          CASE  a       IS
                WHEN X =>   RETURN   FX;
                WHEN L =>   RETURN    F0;
                WHEN H =>   RETURN   F1;
                WHEN Z =>   RETURN   ZX;
          END CASE ;
      END convert4state;

    FUNCTION  convert9val (a : nineval)  RETURN   fourstate   IS;
      BEGIN
          CASE  a       IS
                WHEN Z0 =>   RETURN   Z;
                WHEN Z1 =>   RETURN   Z;
                WHEN ZX =>   RETURN   Z;
                WHEN R0 =>   RETURN   L;
                WHEN R1 =>   RETURN   H;
                WHEN RX =>   RETURN    X;
                WHEN F0 =>   RETURN   L;
                WHEN F1 =>   RETURN   H;
                WHEN FX =>   RETURN   X;
          END CASE ;
      END convert9val;
END my_pack;

USE          WORK.my_pack.ALL;
ENTITY       trans2       IS
  PORT       ( a, b:      INOUT      nvector2;
               enable:    IN         nineval);
END    trans2;

ARCHITECTURE     struct  OF   trans2  IS
  COMPONENT      trans
   PORT       ( x1, x2:    INOUT      fourstate;
                En:        OUT        fourstate);
  END COMPONENT;
```

```
BEGIN
   U1:  trans   PORT MAP ( convert4state(x1) => convert9val(a(0)) ,
                           ( convert4state(x2) => convert9val(b(0)) ,
                             en => convert9val(enable));

   U2:  trans   PORT MAP ( convert4state(x1) => convert4state(a(1)) ,
                           ( convert4state(x2) => convert4state(b(1)) ,
                             en => convert9val(enable));
END  struct;
```

Mỗi thành phần là một linh kiện truyền hai chiều có tên là ***trans***. Linh kiện ***trans*** chứa 3 port: các port ***x1*** và ***x2*** là các port vào-ra còn port ***en*** là port ngõ vào. Khi port ***en*** có giá trị H thì ***x1*** được chuyển đến ***x2*** và khi port ***en*** có giá trị L, ***x2*** được chuyển đến ***x1***.

Các thành phần ***trans*** sử dụng kiểu ***fourstate*** làm kiểu của các port trong khi thực thể chứa sử dụng kiểu ***nineval***. Các hàm chuyển đổi được yêu cầu để cho phép thể hiện của các thành phần ***trans*** trong kiến trúc ***struct*** của thực thể ***trans2***.

Phát biểu thể hiện thành phần đầu tiên cho thành phần ***trans*** có nhãn là **U1** trình bày cách thức mà các hàm chuyển đổi được sử dụng cho các port vào-ra. Ánh xạ port đầu tiên sẽ ánh xạ port ***x1*** đến ***a(0)***. Port ***a(0)*** có kiểu ***nineval*** do đó tín hiệu được tạo bởi port này có kiểu ***nineval***.

Khi tín hiệu này được ánh xạ đến port ***x1*** của thành phần ***trans***, tín hiệu này phải được chuyển đổi thành ***fourstate***. Hàm chuyển đổi ***convert9val*** phải được gọi để hoàn tất việc chuyển đổi. Khi dữ liệu được chuyển ra đến port ***x1*** đối với phần xuất của port vào-ra, hàm chuyển đổi ***convert4state*** phải được gọi.

Khi ***x1*** thay đổi, hàm ***convert4state*** được gọi để chuyển đổi giá trị ***fourstate*** thành giá trị ***nineval*** trước khi được chuyển đến thực thể chứa ***trans2***. Ngược lại khi port ***a(0)*** thay đổi, hàm ***convert9val*** được gọi để chuyển đổi giá trị ***nineval*** thành giá trị ***fourstate***, giá trị này có thể được sử dụng bên trong mô hình ***strans***.

Các hàm chuyển đổi được sử dụng để chuyển đổi một giá trị của một kiểu này thành một giá trị của kiểu khác. Các hàm này có thể được gọi một cách rõ ràng như là một phần của việc thực thi hoặc không rõ ràng từ một ánh xạ trong một thể hiện thành phần.

c. Hàm phân tích:

Hàm phân tích được sử dụng để trả về giá trị của một tín hiệu khi tín hiệu được kích bởi nhiều driver. Sẽ không hợp lệ trong VHDL khi có một tín hiệu với nhiều driver mà không có hàm phân tích gán cho tín hiệu đó. Một hàm phân tích được gọi mỗi khi một driver của tín hiệu có một sự kiện xảy ra. Hàm phân tích sẽ được thực thi và sẽ trả về một giá trị duy nhất trong tất cả các giá trị của các driver; giá trị này sẽ là giá trị mới của tín hiệu.

Trong các trình mô phỏng điển hình, các hàm phân tích được cài đặt sẵn hoặc cố định. Với VHDL người thiết kế có khả năng định nghĩa bất kỳ loại hàm phân tích nào mong muốn, wired-or, wired-and, giá trị trung bình tín hiệu, ...

Một hàm phân tích có một ngõ vào đối số duy nhất và trả về một giá trị duy nhất. Đối số ngõ vào duy nhất này bao gồm một dãi ràng buộc các giá trị của driver của tín hiệu mà hàm phân tích được gán. Nếu tín hiệu có hai driver, dải không ràng buộc sẽ có hai phần tử; nếu tín hiệu có ba driver, dải không ràng buộc sẽ có ba phần tử. Hàm phân tích sẽ xem xét các giá trị của tất cả các driver và trả về một giá trị duy nhất gọi là giá trị phân tích (resolved value) của tín hiệu.

Ta hãy khảo sát một hàm phân tích đối với kiểu ***fourval*** đã được sử dụng trong các ví dụ hàm chuyển đổi. Khai bao kiểu cho ***fourval*** như sau:

```
TYPE fourval IS (X, L, H, Z);
```

4 giá trị phân biệt được khai báo biểu diễn tất cả các giá trị có thể có mà tín hiệu có thể chứa. Giá trị L biểu diễn logic 0, giá trị H logic 1, giá trị Z biểu diễn điều kiện tổng trở cao, giá trị X biểu diễn điều kiện chưa biết, trong đó giá trị có thể biểu diễn logic 0 hoặc logic 1 (nghĩa là tùy định) nhưng ta không chắc là giá trị nào. Các điều kiện này có thể xảy ra khi hai driver đang kích một tín hiệu – một driver kích với logic H và driver kia kích với logic L.

Liệt kê vào theo thứ tự độ mạnh, với yếu nhất ở trên cùng, các giá trị này như sau.

Z – yếu nhất – H, L và X có thể ghi đè.

H, L – trung bình – chỉ có X có thể ghi đè.

X – mạnh nhất – không bị ghi đè.

Bằng cách sử dụng thông tin này, một bảng giá trị có hai ngõ vào có thể được phát triển như được trình bày ở bảng dưới.

Bảng 2-11 cho giá trị ngõ ra có 2 ngõ vào

	Z	L	H	X
Z	Z	L	H	X
L	L	L	X	X
H	H	X	H	X
X	X	X	X	X

Bảng 2-11.

Bảng giá trị này dùng cho các giá trị hai ngõ vào, ta có thể mở rộng nhiều ngõ vào hơn bằng cách áp dụng liên tiếp bảng này cho hai giá trị ở một thời điểm. Điều này có thể thực hiện được do bảng này có tính giao hoán và kết hợp.

Một L và một Z hoặc một Z và một L sẽ cùng cho kết quả.

Một (L, Z) với H sẽ cho kết quả giống như một (H, Z) với một L.

Các nguyên tắc này rất quan trọng do thứ tự các giá trị của driver bên trong đối số ngõ vào của hàm phân tích là không định trước theo quan điểm của người thiết kế. Bất kỳ phụ thuộc nào trên thứ tự đều có thể gây ra kết quả không định trước từ hàm phân tích.

Bằng cách sử dụng tất cả các thông tin này, một người thiết kế có thể viết một hàm phân tích cho kiểu này. Hàm phân tích sẽ duy trì độ mạnh cao nhất trong chừng mực thấy được và so sánh giá trị này với giá trị mới, một phần tử duy nhất ở một thời điểm cho đến khi tất cả các giá trị đều đã được sử dụng hết. Giải thuật này sẽ trả về giá trị có độ mạnh cao nhất. Dưới đây là một ví dụ cho một hàm phân tích như vậy.

Ví dụ 2-65:

```
PACKAGE     fourpack     IS
  TYPE  fourval              IS     (X, L, H, Z) ;
  TYPE  fourval_vector       IS     ARRAY (nineval RANGE <>) OF  fourval;
     FUNCTION   resolve (s : fourval_vector)  RETURN   fourval;
```

```
END             fourpack;

PACKAGE     BODY   fourpack     IS
     FUNCTION   resolve (s : fourval_vector)  RETURN   fourval IS
        VARIABLE result : fourval := Z;

        BEGIN
              FOR  i IN   s'RANGE   LOOP
                  CASE  result      IS
                      WHEN Z =>
                         CASE  s(i)         IS
                               WHEN H =>   result := H;
                               WHEN L =>   result := L;
                               WHEN X =>   result := X;
                               WHEN OTHERS  =>   NULL;
                         END CASE ;

                      WHEN L =>
                         CASE  s(i)         IS
                               WHEN H =>   result := X;
                               WHEN X =>   result := X;
                               WHEN OTHERS  =>   NULL;
                         END CASE ;

                      WHEN H =>
                         CASE  s(i)         IS
                               WHEN L =>   result := X;
                               WHEN X =>   result := X;
                               WHEN OTHERS  =>   NULL;
                         END CASE ;

                      WHEN X =>   result := X;
                  END CASE ;
             END LOOP ;
             RETURN result ;

        END resolve ;
END fourpack ;
```


Đối số ngõ vào là một mảng không ràng buộc có kiểu nền của driver là ***fourval***. Hàm phân tích sẽ khảo sát tất cả các giá trị của các driver được chuyển vào đối số ***s***, một giá trị ở một thời điểm, rồi trả về giá trị duy nhất có kiểu ***fourval*** để được định thời như là giá trị của tín hiệu.

Biến ***result*** được khởi động bằng giá trị Z cho trường hợp không có driver nào đối với tín hiệu. Trong trường hợp này vòng lặp sẽ không bao giờ được thực thi và giá trị của result được trả về sẽ là giá trị khởi động. Đây cũng là một ý hay nếu ta khởi động giá trị của ***result*** bằng giá trị yếu nhất của hệ thống giá trị để cho phép ghi đè bởi các giá trị mạnh hơn.

Nếu có một driver được tiến hành, vòng lặp sẽ được thực thi một lần cho mỗi giá trị của driver được chuyển vào đối số *s*. Mỗi giá trị của driver được so sánh với giá trị hiện tại được lưu trong biến ***result***. Nếu giá trị mới mạnh hơn theo qui luật đã được nêu ở trên, giá trị của ***result*** sẽ được cập nhật bằng giá trị mới.

d. Thủ tục :

Thủ tục có thể có nhiều thông số ngõ vào, ra và vào-ra. Gọi thủ tục được xem như một phát biểu riêng, hàm thường tồn tại như một phần của biểu thức. Trong hầu hết các trường hợp sử dụng thủ tục chỉ khi có nhiều hơn 1 giá trị được trả về.

Thủ tục có những quy định về cú pháp giống như hàm. Phần khai báo thủ tục bắt đầu với từ khoá PROCEDURE, tiếp theo là tên của thủ tục và sau đó là danh sách các đối số. Sự khác nhau giữa hàm và thủ tục là danh sách các đối số của thủ tục giống như có hướng kết hợp với mỗi thông số, còn danh sách của hàm thì không có. Trong thủ tục, có nhiều đối số có thể ở kiểu IN, OUT hoặc INOUT, trong hàm thì tất cả các đối số ở kiểu IN.

Ví dụ 2-66 về cách sử dụng thủ tục:

Ví dụ 2-66:

```
USE             LIBRARY   IEEE;
USE             IEEE.std_logic_1164.ALL;
PROCEDURE   vector_to_int (z :       IN std_logic_vector;
                  x_flag:  OUT BOOLEAN, q : INOUT INTEGER) IS
        BEGIN
                Q := 0;
                X_flag := false;
                FOR  I   IN   z'RANGE   LOOP
                        Q := q * 2;
                        IF  z(i) = '1'    THEN  q := q + 1;
                        ELSIF  z(i) /= F0    THEN  x_flag  := true;
                        END  IF;
              END LOOP ;
        END vector_to_int  ;
```

Hành vi của thủ tục là chuyển đổi đối số ngõ vào *z* từ mảng kiểu số nguyên. Tuy nhiên nếu mảng ngõ vào có giá trị chưa xác định thì giá trị số nguyên không thể được tạo ra từ mảng. Khi điều kiện này xảy ra thì đối số x_flag được thiết lập ở giá trị ***true*** để xác định giá trị số nguyên ngõ ra là không xác định. Thủ tục được yêu cầu để điều khiển hành vi này bởi vì có nhiều kết quả trả về. Chúng ta hãy kiểm tra kết quả từ thủ tục với mảng giá trị ngõ vào như sau:

'0' '0' '1' '1'

Bước thứ nhất thủ tục sẽ khởi động các giá trị ngõ ra với các điều kiện đã biết, trong trường hợp đối số ngõ vào dài bằng 0 được truyền vào. Đối số ngõ ra x_flag được khởi tạo ở trạng thái false và tiếp tục ở trạng thái false cho đến khi chứng minh trạng thái ngược lại.

Phát biểu vòng lặp xuyên qua vector ngõ vào z và tiếp tục cộng mỗi giá trị của vector cho đến khi tất cả các giá trị đã được cộng.

Nếu giá trị là '1' thì sau đó nó được cộng vào kết quả. Nếu giá trị là '0' thì không cộng.

Nếu bất kỳ giá trị nào được tìm thấy trong vector thì kết quả x_flag được thiết lập là true xác định rằng điều kiện chưa biết đã được tìm thấy ở một trong các ngõ vào. (Chú ý thông số q đã được định nghĩa như thông số vao-ra, điều này là cần thiết bởi vì giá trị được đọc trong thủ tục).

Thủ tục không có thông số

Ví dụ 2-67 trình bày một thủ tục có 1 đối số vao-ra thuộc dạng bản ghi. Bản ghi chứa một mảng 8 số nguyên cùng với trường số được dùng để lưu giá trị trung bình của tất cả các số nguyên. Thủ tục tính toán giá trị trung bình của các giá trị số nguyên, ghi giá trị trung bình trong vùng trường trung bình của bản ghi và trở về với bản ghi đã cập nhật:

Ví dụ 2-67:

```
PACKAGE     intpack       IS
  TYPE  bus_stat_vec      IS     ARRAY (0 TO 7)    OF INTEGER;
  TYPE  bus_stat_t        IS
     RECORD
          bus_val: bus_stat_vec;
          average_val: INTEGER;
     END RECORD;

  PROCEDURE   bus_average (x :  INOUT  bus_stat_t);
END     intpack;

PACKAGE     BODY   intpack      IS
  PROCEDURE   bus_average (x :  INOUT  bus_stat_t) IS
    VARIABLE     total : INTEGER := 0;
        BEGIN
             FOR  i IN   0 TO 7    LOOP total  := total + x.bus_val(i);
             END LOOP ;
             x.average_val  := total / 8 ;
        END bus_average ;
END intpack ;

PROCESS     (mem_update)
    VARIABLE     bus_statistics  : bus_stat_t;
    BEGIN
        bus_statistics.bus_val := (50, 40, 30, 35, 45, 55, 65, 85);
        bus_average(bus_statistics);
        average  <= bus_statistics.average_val;
END PROCESS ;
```


Phát biểu đầu tiên là gán biến. Phát biểu thứ hai là gọi thủ tục ***bus_average*** để thực hiện tính toán giá trị trung bình. Để bắt đầu, đối số cho thủ tục bus_average là một giá trị ngõ vào nhưng sau khi thủ tục thực hiện xong thì đối số trở thành giá trị ngõ ra – có thể được sử dụng bên trong cho việc gọi xử lý. Giá trị ngõ ra từ thủ tục được gán cho tín hiệu ngõ ra nằm ở hàng cuối cùng của quá trình.

2. GÓI:

Mục đích quan cơ bản của gói là gói gọn các phần tử có thể dùng chung, bao gồm hai hay nhiều đơn vị thiết kế. Gói là miền lưu trữ chung được sử dụng để lưu trữ dữ liệu dùng chung giữa một số thực thể. Việc khai báo dữ liệu bên trong một gói cho phép dữ liệu được tham chiếu bởi các thực thể khác.

Một gói gồm có hai phần: phần khai báo gói và phần thân của gói. Khai báo gói định nghĩa giao diện cho gói với cùng phương pháp mà một thực thể định nghĩa giao diện cho một mô hình. Thân của gói chỉ ra hành vi thực sự của gói theo cùng phương pháp mà phát biểu kiến trúc thực hiện đối với một mô hình.

a. Khai báo gói:

Phần khai báo gói có thể chứa các khai báo sau:

– Khai báo chương trình con.

– Khai báo kiểu, kiểu con.

– Khai báo hằng, hằng trì hoãn (deferred constant).

– Khai báo tín hiệu, tạo ra một tín hiệu toàn cục.

– Khai báo tập tin.

– Khai báo bí danh (tên khác).

– Khai báo thành phần.

– Khai báo thuộc tính, thuộc tính do người sử dụng định nghĩa.

– Đặc tả thuộc tính.

– Đặc tả không kết nối.

– Mệnh đề USE.

Tất cả các mục được khai báo trong phần khai báo gói được nhìn thấy ở bất kỳ đơn vị thiết kế nào sử dụng gói này bằng mệnh đề **USE**. Giao diện của một gói chứa bất kỳ chương trình con nào hoặc các hằng trì hoãn được khai báo trong phần khai báo gói. Các khai báo chương trình con và hằng trì hoãn phải có một thân chương trình con tương ứng và giá trị hằng trì hoãn tương ứng trong thân của gói.

Hằng trì hoãn: Các hằng trì hoãn là các hằng có tên và kiểu được khai báo trong phần khai báo gói nhưng có giá trị thực sự được chỉ định trong phần thân của gói. Ví dụ 2-68 về hằng trì hoãn trong khai báo gói như sau:

Ví dụ 2-68:

```
PACKAGE tpack IS
        CONSTANT timing_mode: t_mode;
END tpack;
```

Ví dụ trên trình bày một hằng trì hoãn có tên là ***timing_mode*** được định nghĩa có kiểu ***t_mode***. Giá trị thực sự của hằng này sẽ được chỉ ra khi thân của gói được dịch.

b. Khai báo chương trình con:

Một mục khác tạo thành giao diện cho gói là khai báo chương trình con. Khai báo chương trình con cho phép người thiết kế chỉ định giao diện của một chương trình con tách biệt khỏi thân chương trình con. Chức năng này cho phép bất kỳ người thiết kế nào sử dụng chương trình con để bắt đầu hoặc tiếp tục việc thiết kế, trong khi đó đặc tả giao diện của các chương trình con được trình bày chi tiết. Chức năng này cũng cung cấp cho nhà thiết kế thân các chương trình con sự tự do thay đổi các hoạt động bên trong của các chương trình con mà không ảnh hưởng đến bất kỳ thiết kế nào sử dụng chương trình con đó. Ví dụ 2-69 khai báo chương trình con như sau:

Ví dụ 2-69:

```
PACKAGE     cluspack     IS
  TYPE  nineval      IS     (Z0, Z1, ZX, R0, R1, RX, F0, F1, FX);
  TYPE  t_cluster    IS     ARRAY (0 TO 15)   OF nineval;
  TYPE  t_clus_vec   IS     ARRAY (natural range <>)  OF  t_cluster;

  FUNCTION   resolve_cluster (s :   t_clus_vec)  RETURN   t_cluster;
  SUSTYPE   t_wclus IS resolve_cluster  t_cluster;
  CONSTANT   undriven:  t_wclus;
END     cluspack;
```

Khai báo chương trình con cho ***resolve_cluster*** chỉ định tên của chương trình con, các đối số bất kỳ của chương trình con, các kiểu và các loại của các đối số và trả về kiểu nếu chương trình con là một hàm. Khai báo này có thể được sử dụng để biên dịch bất kỳ mô hình nào dự định sử dụng chương trình con mà chưa có thân chương trình con thực sự được chỉ định. Thân chương trình con phải hiện hữu trước khi trình mô phỏng được xây dựng.

Thân của gói: Mục đích chính của thân của gói là định nghĩa các giá trị cho các hằng trì hoãn và chỉ định các thân chương trình con cho bất kỳ khai báo chương trình con nào từ khai báo gói. Tuy nhiên thân gói cũng có thể chứa các khai báo sau:

– Khai báo chương trình con.

– Thân chương trình con.

– Khai báo kiểu, kiểu con.

– Khai báo hằng, khai báo này điền vào giá trị của hằng trì hoãn.

– Khai báo tập tin.

– Khai báo bí danh.

– Mệnh đề USE.

Tất cả các khai báo trong thân của gói – ngoại trừ khai báo hằng mà chúng chỉ định giá trị của hằng trì hoãn và khai báo thân chương trình con – là cục bộ đối với thân của gói. Chúng ta hãy khảo sát thân của gói cho khai báo gói đã được đề cập ở phần trước.

Ví dụ 2-70:

```
PACKAGE     BODY  cluspack     IS
 CONSTANT   undriven:  t_wclus:=
                      (ZX, ZX, ZX, ZX,
                       ZX, ZX, ZX, ZX,
                       ZX, ZX, ZX, ZX,
                       ZX, ZX, ZX, ZX,);

 FUNCTION   resolve_cluster (s :   t_clus_vec)   RETURN   t_cluster IS
  VARIABLE      result : t_cluster;
  VARIABLE      driver_count : integer;
       BEGIN
          IF  S'LENGTH = 0    THEN     RETURN      undriven;
          END IF;
          FOR  i      IN   S'RANGE    LOOP
             IF       S(i) /= undriven    THEN     driver_count := driver_count  + 1;
                IF  driver_count = 1    THEN    result := s(i);
                   ELSE                    result := undriven;
                   ASSERT  FALSE
                   REPORT "multiple drivers detected"
                   SEVERITY ERROR;
                END IF;
             END IF;
          END LOOP ;
          RETURN   result;
       END resolve_cluster ;
END cluspack;
```

Phát biểu thân của gói tương tự như khai báo gói ngoại trừ từ khóa **BODY** theo sau **PACKAGE**. Tuy nhiên các nội dung của hai đơn vị thiết kế này rất khác nhau. Thân gói cho ví dụ này chỉ chứa hai mục: giá trị hằng trì hoãn của hằng trì hoãn ***undriven*** và thân chương trình con của chương trình con ***resolve_cluster***. Ta hãy lưu ý đến cách thức mà đặc tả giá trị hằng trì hoãn tương thích với khai báo hằng trì hoãn trong khai báo gói và thân chương trình con tương thích với khai báo chương trình con trong khai báo gói. Thân chương trình con phải tương thích chính xác với khai báo chương trình con về số thông số, kiểu của các thông số và kiểu trả về.

Thân của gói cũng có thể chứa các khai báo cục bộ chỉ được sử dụng bên trong thân của gói để xây dựng các thân chương trình con khác hoặc các giá trị hằng trì hoãn. Các khai báo này không được nhìn thấy từ bên ngoài thân của gói nhưng có thể rất có ích bên trong thân gói. Ví dụ 2-71 về một gói hoàn chỉnh sử dụng tính chất này như sau:

Ví dụ 2-71:

```
LIBRARY      IEEE;
USE          IEEE.std_logic_1164.ALL;
PACKAGE      math  IS
  TYPE  st16  IS     ARRAY (0 TO 15)   OF std_logic;
  FUNCTION  add(a, b :     IN  st16)  RETURN   st16;
  FUNCTION  sub(a, b :     IN  st16)  RETURN   st16;
END     math;

PACKAGE     BODY  math IS
  FUNCTION  vect_to_int (s :  st16)  RETURN   INTEGER IS
    VARIABLE     result : INTEGER:=0;
        BEGIN
           FOR  i     IN   0 TO 7    LOOP
              result := result * 2;
              IF      S(i) = '1' THEN       result := result  +1;
              END IF;
           END LOOP ;

           RETURN   result;
        END vect_to_int ;

  FUNCTION   in_to_st16 (s :   INTEGER)   RETURN   st16 IS
    VARIABLE     result : st16;
    VARIABLE     digit: INTEGER := 2**15;
    VARIABLE     local : INTEGER;
        BEGIN
           Local:= s;
           FOR  i     IN   15 DOWNTO 0    LOOP
              IF  local/digit >=  1 THEN   result(i) := '1';
                  Local:= Local - digit;
              ELSE  result(i) := '0';
              END IF;
              digit := digit /2;
           END LOOP ;
           RETURN   result;
        END in_to_st16 ;

  FUNCTION   add (a,b :   IN st16)   RETURN   st16 IS
    VARIABLE     result : INTEGER;
```

```
        BEGIN
            Result := vect_to_int(a) + vect_to_int(b) ;
            RETURN   int_to_st16 (result);

        END add;
    FUNCTION   sub (a,b :   IN st16)   RETURN   st16 IS
      VARIABLE      result : INTEGER;
        BEGIN
            Result := vect_to_int(a) - vect_to_int(b) ;
            RETURN   int_to_st16 (result);

        END sub;
END     math;
```

Khai báo gói ở trên là một khai báo kiểu ***st16*** và hai hàm ***add*** và ***sub*** hoạt động theo kiểu nêu trên. Thân gói có chứa thân các hàm cho các khai báo hàm *add* và *sub* và cũng chứa hai hàm chỉ được sử dụng trong thân của gói đó là các hàm này là ***int_to_st16*** và ***vec_to_int***. Các hàm này không được thấy từ bên ngoài thân của gói. Để làm cho các hàm này có thể nhìn thấy được, một khai báo hàm cần phải thêm vào phần khai báo gói cho mỗi hàm.

Các hàm ***vec_to_int*** và ***int_to_st16*** phải được khai báo trước hàm ***add*** để dịch chương trình cho đúng. Tất cả các hàm phải được khai báo trước khi chúng được sử dụng.

X. CÂU HỎI ÔN TẬP VÀ BÀI TẬP

Câu 2-1. Hãy phân biệt sự khác nhau giữa biến và tín hiệu?

Câu 2-2. Hãy phân biệt sự khác nhau giữ khai báo BIT và STD_LOGIC ?

Câu 2-3. Hãy phân biệt sự khác nhau giữ khai báo BIT và BIT_VECTOR?

Câu 2-4. Hãy phân biệt sự khác nhau giữ khai báo BIT và STD_LOGIC ?

end