Chương II: Ngôn ngữ mô tả phần cứng VHDL

1. Giới thiệu về VHDL

VHDL viết tắt của VHSIC HDL (*Very-high-speed-intergrated-circuit Hardware Description Language*) hay ngôn ngữ mô tả phần cứng cho các mạch tích hợp tốc độ cao.Lịch sử phát triển của VHDL trải qua các môc chính như sau:

1981: Phát triển bởi Bộ Quốc phòng Mỹ nhằm tạo ra một công cụ thiết kế phần cứng tiện dụng có khả năng độc lập với công nghệ và giảm thiểu thời gian cũng như chi phí cho thiết kế

1983-1985: Được phát triển thành một ngôn ngữ chính thống bởi 3 công ty Intermetrics, IBM and TI.

1986: Chuyển giao toàn bộ bản quyền cho Viện Kỹ thuật Điện và Điện tử (IEEE).

1987: Công bố thành một chuẩn ngôn ngữ IEEE-1076 1987.

1994: Công bố chuẩn VHDL IEEE-1076 1993.

2000: Công bố chuẩn VHDL IEEE-1076 2000.

2002: Công bố chuẩn VHDL IEEE-1076 2002

2007: công bố chuẩn ngôn ngữ Giao diện ứng dụng theo thủ tục VHDL IEEE-1076c 2007

2009: Công bố chuẩn VHDL IEEE-1076 2002

VHDL ra đời trên yêu cầu của bài toán thiết kế phần cứng lúc bấy giờ, nhờ sử dụng ngôn ngữ này mà thời gian thiết kế của sản phẩm bán dẫn giảm đi đáng kể, đồng thời với giảm thiểu chi phí cho quá trình này do đặc tính độc lập với công nghệ, với các công cụ mô phỏng và khả năng tái sử dụng các khối đơn lẻ. Các ưu điểm chính của VHDL có thể liệt kê ra là:

- *Tính công cộng:* VHDL là ngôn ngũ được chuẩn hóa chính thức của IEEE do đó được sự hỗ trợ của nhiều nhà sản xuất thiết bị cũng như nhiều nhà cung cấp công cụ thiết kế mô phỏng hệ thống, hầu như tất cả các công cụ thiết kế của các hãng phần mềm lớn nhỏ đều hỗ trợ biên dịch VHDL.

- *Được hỗ trợ bởi nhiều công nghệ*: VHDL có thể sử dụng mô tả nhiều loại vi mạch khác nhau trên những công nghệ khác nhau từ các thư viện rời rạc, CPLD, FPGA, tới thư viện cổng chuẩn cho thiết kế ASIC.

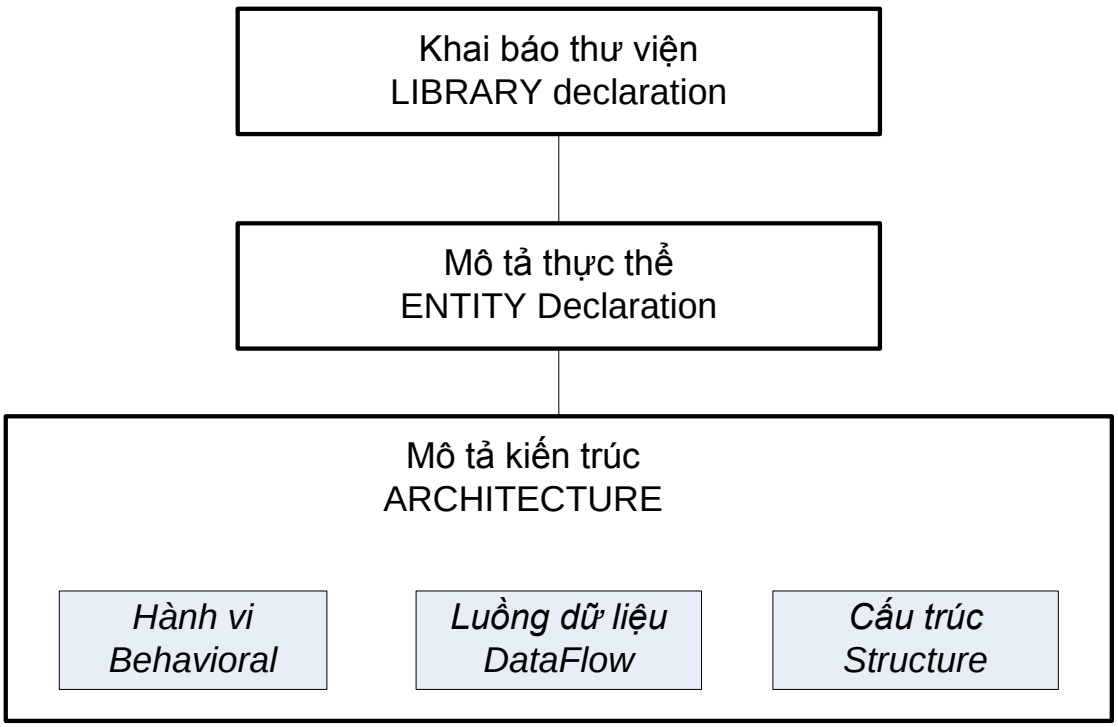
- *Tính độc lập với công nghệ:* VHDL hoàn toàn độc lập với công nghệ chế tạo phần cứng. Một mô tả hệ thống chức năng dùng VHDL thiết kế ở mức thanh ghi truyền tải RTL có thể được tổng hợp thành các mạch trên các công nghệ bán dẫn khác nhau. Nói một cách khác khi một công nghệ phần cứng mới ra đời nó có thể được áp dụng ngay cho các hệ thống đã thiết kế bằng cách tổng hợp các thiết kế đó trên thư viện phần cứng mới.

- *Khả năng mô tả mở rộng:* VHDL cho phép mô tả hoạt động của phần cứng từ mức thanh ghi truyền tải cho đến mức cổng. Hiểu một cách khác VHDL có một cấu trúc mô tả phần cứng chặt chẽ có thể sử dụng ở lớp mô tả chức năng cũng như mô tả cổng (*netlist*) trên một thư viện công nghệ cụ thể nào đó.

- *Khả năng trao đổi kết quả, tái sử dụng:* Việc VHDL được chuẩn hóa giúp cho việc trao đổi các thiết kế giữa các nhà thiết kế độc lập trở nên hết sức dễ dàng. Bản thiết kế VHDL được mô phỏng và kiểm tra có thể được tái sử dụng trong các thiết kế khác mà không phải lặp lại các quá trình trên. Giống như phần mềm thì các mô tả HDL cũng có một cộng đồng mã nguồn mở cung cấp, trao đổi miễn phí các thiết kế chuẩn có thể ứng dụng ở nhiều hệ thống khác nhau.

2. Cấu trúc của chương trình mô tả bằng VHDL

Để thống nhất ta quy ước dùng thuật ngữ “module VHDL” chỉ tới khối mã nguồn của một mô tả thiết kế thiết kế logic độc lập. Cấu trúc tổng thể của một module VHDL gồm ba phần, phần khai báo thư viện, phần mô tả thực thể và phần mô tả kiến trúc.



Hình 2.1: Cấu trúc của một thiết kế VHDL

2.1. Khai báo thư viện

Khai báo thư viện phải được đặt đầu tiên trong mỗi module VHDL, lưu ý rằng nếu ta sử dụng một file để chứa nhiều module khác nhau thì mỗi một module đều phải yêu cầu có khai báo thư viện đầu tiên, nếu không khi biên dịch sẽ phát sinh ra lỗi.

Ví dụ về khai báo thư viện

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

Khai báo thư viện bắt đầu bằng từ khóa **Library** Tên thư viện (chú ý là VHDL không phân biệt chữ hoa chữ thường). Sau đó trên từng dòng kế tiếp sẽ khai báo các gói thư viện con mà thiết kế sẽ sử dụng, mỗi dòng phải kết thúc bằng dấu ";"

Tương tự như đối với các ngôn ngữ lập trình khác, người thiết kế có thể khai báo sử dụng các thư viện chuẩn hoặc thư viện người dùng.Thư viện IEEE gồm nhiều gói thư viện con khác nhau trong đó đáng chú ý có các thư viện sau:.

- Gói `IEEE.std_logic_1164` cung cấp các kiểu dữ liệu std_ulogic, **std_logic**, std_ulogic_vector, **std_logic_vector**, các hàm logic and, or, not, nor, xor… các hàm chuyển đổi giữa các kiểu dữ liệu trên. `Std_logic, std_ulogic` hỗ trợ kiểu logic với 9 mức giá trị (xem 4.2)
- Gói `STD.TEXTIO.all` chứa các hàm vào ra READ/WRITE để đọc ghi dữ liệu từ FILE, STD_INPUT, STD_OUTPUT.
- Gói `IEEE.std_logic_arith.all` định nghĩa các kiểu dữ liệu số nguyên SIGNED, UNSIGNED, INTEGER, cung cấp các hàm số học bao gồm "+", "-", "*", "/", so sánh "<", ">", "<=", ">=", các hàm dịch trái, dịch phải SHL, SHR, các hàm chuyển đổi từ kiểu vector sang các kiểu số nguyên và ngược lại.
- Gói `IEEE.math_real.all; IEEE.math_complex.all;` cung cấp các hàm làm việc với số thực và số phức như SIN, COS, SQRT… hàm làm tròn, CIEL, FLOOR, hàm tạo số ngẫu nhiên SRAND, UNIFORM… cà nhiều các hàm tính toán số thực khác.
- Gói `IEEE.numeric_std.all;` và `IEEE.numeric_bit.all` cung cấp các hàm tính toán và biến đổi với các dữ liệu kiểu số có dấu, không dấu, chuỗi bit và chuỗi dữ liệu kiểu `std_logic`.

  Cụ thể và chi tiết hơn về các thư viện chuẩn của IEEE có thể tham khảo thêm trong tài liệu của IEEE (*VHDL Standard Language reference*), hoặc các nguồn tham khảo khác trên Internet.

2.2. Mô tả thực thể

Khai báo thực thể *(entity)* là khai báo về mặt cấu trúc các cổng vào ra *(port)*, các tham số tĩnh dùng chung *(generic)* của một module VHDL.

```
entity  identifier is
     generic (generic_variable_declarations);
     port (input_and_output_variable_declarations);
end entity identifier ;
```

Trong đó

- `identifier` là tên của module.
- khai báo `generic` là khai báo các tham số tĩnh của thực thể, khai báo này rất hay sử dụng cho những module có những tham số thay đổi kiểu như như độ rộng kênh, kích thước ô nhớ, tham số bộ đếm… ví dụ chúng ta có thể thiết kế bộ cộng cho các hạng tử có độ dài bit thay

đổi, số bit được thể hiện là hằng số trong khai báo **generic** (xem ví dụ dưới đây)

- Khai báo cổng vào ra: liệt kê tất cả các công vào ra của module, Các cổng có thể hiểu là các kênh dữ liệu động của module để phân biệt với các tham số trong khai báo generic. kiểu của các cổng có thể là:
- **in**: cổng vào,
- **out**: cổng ra,
- **inout** vào ra hai chiều.
- **buffer**: cổng đệm có thể sử dụng như tín hiệu bên trong và **output**.
- **linkage:** Có thể là bất kỳ các cổng nào kể trên

Ví dụ cho khai báo thực thể như sau:

```
entity adder is
     generic ( N    : natural := 32);
     port    ( A    : in  bit_vector(N-1 downto 0);
               B    : in  bit_vector(N-1 downto 0);
               cin  : in  bit;
               Sum  : out bit_vector(N-1 downto 0);
               Cout : out bit );
end entity adder ;
```


Đoạn mã trên khai báo một thực thể cho module cộng hai số, trong khai báo trên N là tham số tĩnh **generic** chỉ độ dài bit của các hạng tử, giá trị ngầm định N = 32, việc khai báo giá trị ngầm định là không bắt buộc. Khi module này được sử dụng trong module khác thì có thể thay đổi giá trị của N để thu được thiết kế theo mong muốn. Về các cổng vào ra, module cộng hai số nguyên có 3 cổng vào A, B N-bit là các hạng tử và cổng cin là bít nhớ từ bên ngoài. Hai cổng ra là Sum N-bit là tổng và bít nhớ ra Cout.

Khai báo thực thể có thể chứa chỉ mình khai báo cổng như sau:

```
entity full_Adder is
    port (
          X, Y, Cin  : in  bit;
          Cout, Sum  : out bit
          );
end full_adder ;
```


Khai báo thực thể không chứa cả khai báo `generic` lẫn khai báo `port` vẫn được xem là hợp lệ, ví dụ những khai báo thực thể sử dụng để mô phỏng kiểm tra thiết kế thường được khai báo như sau:

```
entity TestBench is
end TestBench;
```

Ví dụ về cổng dạng **buffer** và **inout**: Cổng buffer được dùng khi tín hiệu được sử dụng như đầu ra đồng thời như một tín hiệu bên trong của module, điển hình như trong các mạch dãy làm việc đồng bộ. Xét ví dụ sau về bộ cộng tích lũy 4-bit đơn giản sau (accumulator):

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
use IEEE.STD_LOGIC_arith.ALL;
----------------------------------------
entity accumulator is
    port(
        data  : in     std_logic_vector(3 downto 0);
        nRST  : in     std_logic;
        CLK   : in     std_logic;
        acc   : buffer std_logic_vector(3 downto 0)
        );
end accumulator;
----------------------------------------
architecture behavioral of accumulator is
begin
    ac  : process (CLK)
    begin
        if CLK = '1' and CLK'event then
            if nRST = '1' then
                acc <= "0000";
            else
                acc <=  acc + data;
            end if;
         end if;
    end process ac;
end behavioral;
----------------------------------------
```

Bộ cộng tích lũy sau mỗi xung nhịp CLK sẽ cộng giá trị hiện có lưu trong acc với giá trị ở đầu vào data, tín hiệu nRST dùng để thiết lập lại giá trị bằng 0 cho acc. Như vậy acc đóng vai trò như thanh ghi kết quả đầu ra cũng như giá trị trung gian được khai báo dưới dạng **buffer**. Trên thực tế thay vì dùng cổng buffer thường sử dụng một tín hiệu trung gian, khi đó cổng acc có thể khai báo như cổng ra bình thường, cách sử dụng như vậy sẽ tránh được một số lỗi có thể phát sinh khi tổng hợp thiết kế do khai báo **buffer** gây ra.

Ví dụ sau đây là mô tả VHDL của một khối đệm ba trạng thái 8-bit, sử dụng khai báo cổng INOUT. Cổng ba trạng thái được điều khiển bởi tín hiệu OE,

khi OE bằng 0 giá trị của cổng là trạng thái trở kháng cao "ZZZZZZZZ", khi OE bằng 1 thì cổng kết nối đầu vào inp với outp.

```
-------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
-------------------------------------------------------
ENTITY bidir IS
    PORT(
        bidir   : inout STD_LOGIC_VECTOR (7 DOWNTO 0);
        oe, clk : in    STD_LOGIC;
        inp     : in    STD_LOGIC_VECTOR (7 DOWNTO 0);
        outp    : out   STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
END bidir;
-------------------------------------------------------
ARCHITECTURE maxpld OF bidir IS
SIGNAL  a : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  b : STD_LOGIC_VECTOR (7 DOWNTO 0);
BEGIN
    PROCESS(clk)
    BEGIN
        IF clk = '1' AND clk'EVENT THEN
            a    <= inp;
            outp <= b;
        END IF;
    END PROCESS;
    PROCESS (oe, bidir)
        BEGIN
        IF( oe = '0') THEN
            bidir <= "ZZZZZZZZ";
            b     <= bidir;
        ELSE
            bidir <= a;
            b     <= bidir;
        END IF;
    END PROCESS;
END maxpld;
-------------------------------------------------------
```

** Trong thành phần của khai báo thực thể ngoài khai báo cổng và khai báo generic còn có thể có hai thành phần khác là khai báo kiểu dữ liệu, thư viện người dùng chung, chương trình con... Và phần phát biểu chung chỉ chứa các phát biểu đồng thời. Các thành phần này nếu có sẽ có tác dụng đối với tất cả các kiến trúc của thực thể. Chi tiết hơn về các thành phần khai báo này có thể xem trong IEEE VHDL Standard Language reference (2002 Edition).*

2.2. Mô tả kiến trúc

Mô tả kiến trúc *(ARCHITECTURE)* là phần mô tả chính của một module VHDL, nếu như mô tả **entity** chỉ mang tính chất khai báo về giao diện của module thì mô tả kiến trúc chứa nội dung về chức năng của module. Cấu trúc của mô tả kiến trúc tổng quát như sau:

```
architecture identifier of entity_name is
     [ declarations]
begin
     [ statements ]
end identifier ;
```

Trong đó

- identifier là tên gọi của kiến trúc, thông thường để phân biệt các kiểu mô tả thường dùng các tên **behavioral** cho mô tả hành vi, **dataflow** cho mô tả luồng dữ liệu, **structure** cho mô tả cấu trúc tuy vậy có thể sử dụng một tên gọi hợp lệ bất kỳ nào khác.

- [declarations] có thể có hoặc không chứa các khai báo cho phép như sau:

Khai báo và mô tả chương trình con (subprogram)

Khai báo kiểu dữ liệu con (subtype)

Khai báo tín hiệu (signal), hằng số (constant), file

Khai báo module con (component)

-[statements] phát biểu trong khối {**begin end process;**} chứa các phát biểu đồng thời (concurrent statements) hoặc các khối process chứa các phát biểu tuần tự (sequential statements).

Có ba dạng mô tả cấu trúc cơ bản là mô tả hành vi (behavioral), mô tả luồng dữ liệu (dataflow) và mô tả cấu trúc (structure). Trên thực tế trong mô tả kiến trúc của những module phức tạp thì sử dụng kết hợp cả ba dạng mô tả này. Để tìm hiểu về ba dạng mô tả kiến trúc ta sẽ lấy ví dụ về module full_adder có khai báo entity như sau

```
entity full_adder is
     port ( A    : in  std_logic;
            B    : in  std_logic;
            cin  : in  std_logic;
            Sum  : out std_logic;
            Cout : out std_logic);
end entity full_adder;
```

2.2.1 Mô tả hành vi

Đối với thực thể `full_adder` như trên kiến trúc hành vi (behavioral) được viết như sau

```
-------------------------------------------------
architecture behavioral of full_adder is
begin
    add: process (A,B,Cin)
    begin
        if    (a ='0' and b='0' and Cin = '0') then
            S    <= '0';
            Cout <='0';
        elsif (a ='1' and b='0' and Cin = '0') or
              (a ='0' and b='1' and Cin = '0') or
              (a ='0' and b='0' and Cin = '1') then
            S    <= '1';
            Cout <='0';
        elsif (a ='1' and b='1' and Cin = '0') or
              (a ='1' and b='0' and Cin = '1') or
              (a ='0' and b='1' and Cin = '1') then
            S    <= '0';
            Cout <= '1';
        elsif (a ='1' and b='1' and Cin = '1') then
            S    <= '1';
            Cout <= '1';
        end if;
    end process add;
end behavioral;
-------------------------------------------------
```

Mô tả hành vi gần giống như mô tả bằng lời cách thức tính toán kết quả đầu ra dựa vào các giá trị đầu vào. Toàn bộ mô tả hành vi phải được đặt trong một khối { `process (signal list) end process;` } ý nghĩa của khối này là nó tạo một quá trình để "theo dõi" sự thay đổi của tất cả các tín hiệu có trong danh sách tín hiệu (`signal list`), khi có bất kỳ một sự thay đổi giá trị nào của tín hiệu trong danh sách thì nó sẽ thực hiện quá trình tính toán ra kết quả tương ứng ở đầu ra. Chính vì vậy trong đó rất hay sử dụng các phát biểu tuần tự như if, case, hay các vòng lặp.

Việc mô tả bằng hành vi không thể hiện rõ được cách thức cấu tạo của vi mạch như các dạng mô tả khác và tùy theo những cách viết khác nhau thì có thể thu được những kết quả tổng hợp khác nhau.

Trong các mạch dãy đồng bộ, khối làm việc đồng bộ thường được mô tả bằng hành vi, ví dụ như trong đoạn mã sau mô tả thanh ghi sau:

```
process(clk)
begin
    if clk'event and clk='1' then
        Data_reg <= Data_in;
    end if;
end process;
```

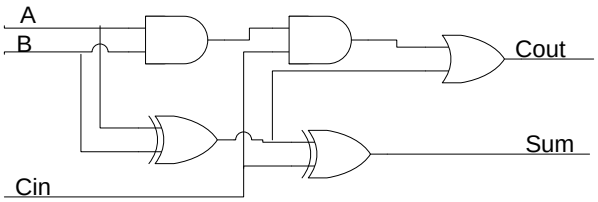
2.2.1 Mô tả luồng dữ liệu

Mô tả luồng dữ liệu (*dataflow*)là dạng mô tả tương đối ngắn gọn và rất hay được sử dụng khi mô tả các module mạch tổ hợp. Các phát biểu trong khối `begin end` là các phát biểu đồng thời `(concurrent statements)` nghĩa là không phụ thuộc thời gian thực hiện của nhau, nói một cách khác không có thứ tự ưu tiên trong việc sắp xếp các phát biểu này đứng trước hay đứng sau trong đoạn mã mô tả. Ví dụ cho module `full_adder` thì mô tả luồng dữ liệu như sau:

```
architecture dataflow of full_adder is
begin
    sum  <= (a xor b) xor Cin;
    Cout <= (a and b) or (Cin and (a xor b));
end dataflow;
```

2.2.1 Mô tả cấu trúc

Mô tả cấu trúc (*structure*) là mô tả sử dụng các mô tả có sẵn dưới dạng module con (*component*). Dạng mô tả này cho kết quả sát với kết quả tổng hợp nhất. Chẳng quan sát như ở mô tả luồng dữ liệu như ở trên có thể thấy có thể dùng hai cổng XOR, một cổng OR và 2 cổng AND để thực hiện thiết kế như sau:



Hình 2.2: Sơ đồ logic của full_adder

Trước khi viết mô tả cho `full_adder` cần phải viết mô tả cho các phần tử cổng AND, OR, XOR như sau

```
------------ 2 input AND gate ---------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
---------------------------------------
entity AND2 is
     port(
          in1, in2 : in  std_logic;
          out1     : out std_logic
```

```
        );
end AND2;
---------------------------------------
architecture model_conc of AND2 is
begin
    out1 <= in1 and in2;
end model_conc;
---------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
------------ 2 input OR gate ----------
entity OR2 is
port (
      in1, in2 : in  std_logic;
      out1     : out std_logic
      );
end OR2;
---------------------------------------
architecture model_conc2 of AND2 is
begin
     out1 <= in1 or in2;
end model_conc2;
------------ 2 input XOR gate ---------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
---------------------------------------
entity XOR2 is
     port (
           in1, in2 : in  std_logic;
           out1     : out std_logic);
end XOR2;
---------------------------------------
architecture model_conc2 of XOR2 is
begin
    out1 <= in1 xor in2;
end model_conc2;
```

Sau khi đã có các cổng trên có thể thực hiện viết mô tả cho `full_adder` như sau

```
---------------------------------------
architecture structure of full_adder is
signal t1, t2, t3: std_logic;

component AND2
    port (
          in1, in2 : in  std_logic;
          out1     : out std_logic
          );
end component;
component OR2
```

```
    port (
          in1, in2 : in  std_logic;
          out1     : out std_logic);
end component;
component XOR2
    port (
          in1, in2 : in  std_logic;
          out1     : out std_logic
         );
end component;

begin
u1 : XOR2 port map (a, b, t1)
u2 : XOR2 port map (t1, Cin, Sum)
u3 : AND2 port map (t1, Cin, t2)
u4 : AND2 port map (a, b, t3)
u5 : OR2  port map (t3, t2, Cout);

end structure;
-------------------------------------
```

Như vậy mô tả cấu trúc tuy khá dài nhưng là mô tả cụ thể về cấu trúc mạch, ưu điểm của phương pháp này là khi tổng hợp trên thư viện cổng sẽ cho ra kết quả đúng với ý tưởng thiết kế nhất. Với mô tả full_adder như trên thì gần như 99% trình tổng hợp đưa ra sơ đồ logic sử dụng 2 cổng XOR, hai cổng AND và 1 cổng OR. Mặt khác mô tả cấu trúc cho phép gộp nhiều mô tả con vào một module lớn mà vẫn giữ được cấu trúc mã rõ ràng và khoa học. Nhược điểm là không thể hiện rõ ràng chức năng của mạch như hai mô tả ở các phần trên.

Ở ví dụ trên có sử dụng khai báo cài đặt module con, chi tiết về khai báo này xem trong mục 7.5.

2.3 Khai báo cấu hình

Một thực thể có thể có rất nhiều kiến trúc khác nhau. Bên cạch đó cấu trúc của ngôn ngữ VHDL cho phép sử dụng các module theo kiểu lồng ghép, vì vậy đối với một thực thể bất kỳ cần có thêm các mô tả để quy định việc sử dụng các kiến trúc khác nhau. Khai báo cấu hình (*Configuration declaration*) được sử dụng để chỉ ra kiến trúc nào sẽ được sử dụng trong thiết kế.

Cách thứ nhất để sử dụng khai báo cấu hình là sử dụng trực tiếp khai báo cấu hình bằng cách tạo một đoạn mã cấu hình độc lập không thuộc một thực thể hay kiến trúc nào theo cấu trúc:

```
configuration identifier of entity_name is
    [declarations]
    [block configuration]
end configuration identifier;
```

Ví dụ sau tạo cấu hình có tên add32_test_config cho thực thể add32_test, cấu hình này quy định cho kiến trúc có tên circuits của thực thể add32_test, khi cài đặt các module con có tên add32 sử dụng kiến trúc tương ứng là WORK.add32(circuits), với mọi module con add4c của thực thể add32 thì sử dụng kiến trúc WORK.add4c(circuit), tiếp đó là quy định mọi module con có tên fadd trong thực thể add4c sử dụng kiến trúc có tên WORK.fadd(circuits).

```
configuration add32_test_config of add32_test is
    for circuits -- of add32_test
      for all: add32
        use entity WORK.add32(circuits);
        for circuits -- of add32
          for all: add4c
            use entity WORK.add4c(circuits);
            for circuits -- of add4c
              for all: fadd
                use entity WORK.fadd(circuits);
              end for;
            end for;
          end for;
        end for;
      end for;
  end configuration add32_test_config;
```

Cặp lệnh cơ bản của khai báo cấu hình là cặp lệnh **for**… **use** … **end for;** có tác dụng quy định cách thức sử dụng các kiến trúc khác nhau ứng với các khối khác nhau trong thiết kế. Bản thân configuration cũng có thể được sử dụng như đối tượng của lệnh **use**, ví dụ:

```
configuration adder_behav of adder4 is
   for structure
     for all: full_adder
        use entity work.full_adder (behavioral);
     end for;
   end for;
 end configuration;
```

Với một thực thể có thể khai báo nhiều cấu hình khác nhau tùy theo mục đích sử dụng. Sau khi được khai báo như trên và biên dịch thì sẽ xuất hiện thêm trong thư viện các cấu hình tương ứng của thực thể. Các cấu hình khác nhau xác định các kiến trúc khác nhau của thực thể và có thể được mô phỏng

độc lập. Nói một cách khác cấu hình là một đối tượng có cấp độ cụ thể cao hơn so với kiến trúc.

Cách thức thứ hai để quy định việc sử dụng kiến trúc là dùng trực tiếp cặp lệnh **for… use … end for;** như minh họa dưới đây, cách thức này cho phép khai báo cấu hình trực tiếp bên trong một kiến trúc cụ thể:

```
-----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library work;
use work.all;
-----------------------------------------
entity adder4 is
    port(
         A   : in  std_logic_vector(3 downto 0);
         B   : in  std_logic_vector(3 downto 0);
         CI  : in  std_logic;
         SUM : out std_logic_vector(3 downto 0);
         CO  : out std_logic
         );
end adder4;
-----------------------------------------
architecture structure of adder4 is
signal C: std_logic_vector(2 downto 0);
-- declaration of component full_adder
component full_adder
    port(
         A    : in  std_logic;
         B    : in  std_logic;
         Cin  : in  std_logic;
         S    : out std_logic;
         Cout : out std_logic
         );
end component;

    for u0: full_adder use entity
work.full_adder(behavioral);
    for u1: full_adder use entity work.full_adder(dataflow);
    for u2: full_adder use entity work.full_adder(structure);
    for u3: full_adder use entity
work.full_adder(behavioral);

    begin
        -- design of 4-bit adder
    u0: full_adder
```

```
        port map (A => A(0), B => B(0), Cin => CI,   S
=>Sum(0), Cout => C(0));
    u1: full_adder
        port map (A => A(1), B => B(1), Cin => C(0), S
=>Sum(1), Cout => C(1));
    u2: full_adder
        port map (A => A(2), B => B(2), Cin => C(1), S
=>Sum(2), Cout => C(2));
    u3: full_adder
        port map (A => A(3), B => B(3), Cin => C(2), S
=>Sum(3), Cout => CO);
    end structure;
    ---------------------------------------
```

Ở ví dụ trên một bộ cộng 4 bit được xây dựng từ 4 khối full_adder nhưng với các kiến trúc khác nhau. Khối đầu tiên dùng kiến trúc hành vi (behavioral), khối thứ hai là kiến trúc kiểu luồng dữ liệu (dataflow), khối thứ 3 là kiến trúc kiểu cấu trúc (structure), và khối cuối cùng là kiến trúc kiểu hành vi.

3. Chương trình con và gói

3.1. Thủ tục

Chương trình con (*subprogram*) là các đoạn mã dùng để mô tả một thuật toán, phép toán dùng để xử lý, biến đổi, hay tính toán dữ liệu. Có hai dạng chương trình con là thủ tục (*procedure*) và hàm (*function*).

Thủ tục thường dùng để thực hiện một tác vụ như biến đổi, xử lý hay kiểm tra dữ liệu, hoặc các tác vụ hệ thống như đọc ghi file, truy xuất kết quả ra màn hình. Khai báo của thủ tục như sau:

```
procedure identifier [(formal parameter list)] is
    [declarations]
begin
    sequential statement(s)
end procedure identifier;
```

ví dụ:


```
procedure print_header ;
procedure build ( A : in    constant integer;
                  B : inout signal   bit_vector;
                  C : out   variable real;
                  D : file);
```

Trong đó `formal parameter list` chứa danh sách các biến, tín hiệu, hằng số, hay dữ liệu kiểu FILE, kiểu ngầm định là biến. Các đối tượng trong

danh sách này trừ dạng file có thể được khai báo là dạng vào (in), ra (out), hay hai chiều (inout), kiểu ngầm định là in. Xét vi dụ đầy đủ dưới đây:

```
-----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use STD.TEXTIO.all;
-----------------------------------------
entity compare is
    port(
         res1, res2 : in bit_vector(3 downto 0)
        );
end compare;
-----------------------------------------
architecture behavioral of compare is
procedure print_to_file(
              val1, val2 : in bit_vector(3 downto 0);
              FILE fout  : text)
              is
    use STD.TEXTIO.all;
    variable str: line;
    begin
        WRITE (str, string'("val1 = "));
        WRITE (str, val1);
        WRITE (str, string'(" val2 = "));
        WRITE (str, val2);
        if val1 = val2 then
            WRITE (str, string'(" OK"));
        elif
            WRITE (str, string'(" TEST FAILED"));
        end if;
        WRITELINE(fout, str);
        WRITELINE(output, str);
end procedure print_to_file;

    FILE file_output : text open WRITE_MODE is
"test_log.txt";
    -- start here
    begin
        proc_compare: print_to_file(res1, res2, file_output);
    end behavioral;
    -----------------------------------------
```

Trong ví dụ trên chương trình con dùng để so sánh và ghi kết quả so sánh của hai giá trị kết quả res1, res2 vào trong file văn bản có tên "test_log.txt". Phần khai báo của hàm được đặt trong phần khai báo của kiến trúc nhưng nếu hàm được gọi trực tiếp trong kiến trúc như ở trên thì khai báo

này có thể bỏ đi. Thân chương trình con được viết trực tiếp trong phần khai báo của kiến trúc và được gọi trực tiếp cặp **begin end behavioral.**

3.2. Hàm

Hàm (*function*) thường dùng để tính toán kết quả cho một tổ hợp đầu vào. Khai báo của hàm có cú pháp như sau:

```
function identifier [(formal parameter list)] return
a_type;
```

ví dụ

```
function random return float ;
function is_even ( A : integer) return boolean ;
```

Danh sách biến của hàm cũng được cách nhau bởi dấu ";" nhưng điểm khác là trong danh sách này không có chỉ rõ dạng vào/ra của biến mà ngầm định tất cả là đầu vào. Kiểu dữ liệu đầu ra của hàm được quy định sau từ khóa **return**. Cách thức sử dụng hàm cũng tương tự như trong các ngôn ngữ lập trình bậc cao khác. Xét một ví dụ đầy đủ dưới đây:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity function_example is
end function_example;
----------------------------------------
architecture behavioral of function_example is
type bv4 is array (3 downto 0) of std_logic;
function mask(mask, val1 : in bv4) return bv4;

signal vector1 : bv4 := "0011";
signal mask1   : bv4 := "0111";
signal vector2 : bv4;

function mask(mask, val1 : in bv4) return bv4 is
    variable temp : bv4;
begin
    temp(0) := mask(0) and val1(0);
    temp(1) := mask(1) and val1(1);
    temp(2) := mask(2) or  val1(2);
    temp(3) := mask(3) or  val1(3);
    return temp;
end function mask;
-- start here
begin
    masking: vector2 <= mask(vector1, mask1);
end behavioral;
```

Ví dụ trên minh họa cho việc sử dụng hàm để thực hiện phép tính mặt nạ (mask) đặc biệt trong đó hai bit thấp của giá trị đầu vào được thực hiện phép logic OR với giá trị mask còn hai bit cao thì thực hiện mask bình thường với phép logic AND. Phần khai báo của hàm được đặt trong phần khai báo của kiến trúc, nếu hàm được gọi trực tiếp trong kiến trúc như ở trên thì khai báo này có thể bỏ đi. Phần thân của hàm được viết trong phần khai báo của kiến trúc trước cặp **`begin end behavioral`**. Khi gọi hàm trong phần thân của kiến trúc thì giá trị trả về của hàm phải được gán cho một tín hiệu, ở ví dụ trên là vector2.

3.3. Gói

Gói (*package*) là tập hợp các kiểu dữ liệu, hằng số, biến, các chương trình con và hàm dùng chung trong thiết kế. Một cách đơn giản gói là một cấp thấp hơn của thư viện, một thư viện cấu thành từ nhiều gói khác nhau. Ngoài các gói chuẩn của các thư viên chuẩn như trình bày ở 2.1, ngôn ngữ VHDL cho phép người dùng tạo ra các gói riêng tùy theo mục đích sử dụng. Một gói bao gồm khai báo gói và phần thân của gói. Khai báo gói có cấu trúc như sau:


```
package identifier is
    [ declarations]
end package identifier ;
```

Phần khai báo chỉ chứa các khai báo về kiểu dữ liệu, biến dùng chung, hằng và khai báo của hàm hay thủ tục nếu có.

Phần thân gói có cú pháp như sau:

```
package body identifier is
    [ declarations]
end package body identifier ;
```

Phần thân gói chứa các mô tả chi tiết của hàm hay thủ tục nếu có. Ví dụ đầy đủ một gói có chứa hai chương trình con như ở các ví dụ ở 3.2 và 3.2 như sau:


```
-----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use STD.TEXTIO.all;
-----------------------------------------
package package_example is
    type bv4 is array (3 downto 0) of std_logic;
    function mask(mask, val1 : in bv4) return bv4;
```

```
        procedure print_to_file(val1, val2 : in bit_vector(3
downto 0); FILE fout :text);
    end package package_example;
    ----------------------------------------
    package body package_example is
    function mask(mask, val1 : in bv4) return bv4;

    signal vector1 : bv4 := "0011";
    signal mask1   : bv4 := "0111";
    signal vector2 : bv4;

    function mask(mask, val1 : in bv4) return bv4 is
        variable temp : bv4;
    begin
        temp(0) := mask(0) and val1(0);
        temp(1) := mask(1) and val1(1);
        temp(2) := mask(2) or  val1(2);
        temp(3) := mask(3) or  val1(3);
        return temp;
    end function mask;
    ----------------------------------------------
    procedure print_to_file(
                  val1, val2 : in bit_vector(3 downto 0);
                  FILE fout  : text)
                  is
        use STD.TEXTIO.all;
        variable str: line;
        begin
            WRITE (str, string'("val1 = "));
            WRITE (str, val1);
            WRITE (str, string'(" val2 = "));
            WRITE (str, val2);
            if val1 = val2 then
                WRITE (str, string'(" OK"));
            elif
                WRITE (str, string'(" TEST FAILED"));
            end if;
            WRITELINE(fout, str);
            WRITELINE(output, str);
    end procedure print_to_file;

    end package body package_example;
    ----------------------------------------------
```

Để sử dụng gói này trong các thiết kế thì phải khai báo thư viện và gói sử dụng tương tự như trong các gói chuẩn ở phần khai báo thư viện. Vì theo ngầm định các gói này được biên dịch vào thư viện có tên là work nên khai báo như sau:

```
-------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use STD.TEXTIO.all;
library work;
use work.package_example.all;
-------------------------------------------
entity pck_example is
port(
    res1, res2 : in bit_vector(3 downto 0)
    );
end pck_example;
-------------------------------------------
architecture behavioral of pck_example is
signal vector2 : bv4;
signal vector1 : bv4 := "0011";
signal mask1   : bv4 := "0111";
FILE file_output : text open WRITE_MODE is
"test_log.txt";
begin

    proc_compare: print_to_file(res1, res2, file_output);
    masking : vector2 <= mask(vector1, mask1);

end behavioral;
-------------------------------------------
```

4. Đối tượng dữ liệu, kiểu dữ liệu

4.1. Đối tượng dữ liệu

Trong VHDL có phân biệt 3 loai đối tượng dữ liệu là biến, hằng và tín hiệu. Các đối tượng được khai báo theo cú pháp

```
Object_type identifier : type [:= initial value];
```

Trong đó object_type có thể là **Variable, Constant** hay **Signal.**

4.1.1. Hằng

Hằng (*Constant*) là những đối tượng dữ liệu dùng khởi tạo để chứa các giá trị xác định trong quá trình thực hiện. Hằng có thể được khai báo trong các gói, thực thể, kiến trúc, chương trình con, các khối và quá trình.

Cú pháp

```
constant identifier :  type [range value] := value;
```

Ví dụ;

```
constant PI :  REAL := 3.141569;
constant vector_1 :  std_logic_vector(8 downto 0) =
"11111111";
```

4.1.2. Biến

Biến (*Variable*) là những đối tượng dữ liệu dùng để chứa các kết quả trung gian, biến chỉ có thể được khai báo bên trong các quá trình hoặc chương trình con. Khai báo biến bao giờ cũng đi kèm với kiểu, có thể có xác định giới hạn giá trị và có giá trị khởi tạo ban đầu, nếu không có giá trị khởi tạo ban đầu thì biến sẽ nhận giá trị khởi tạo là giá trị nhỏ nhất trong miền giá trị.

Cú pháp

```
variable identifier :  type [range value] [:= initial
value ];
```

Ví dụ;

```
variable count      :  integer range 0 to 15 := 0;
variable vector_bit :  std_logic_vector(8 downto 0);
```

4.1.3. Tín hiệu

Tín hiệu (*Signal*) là các đối tượng dữ liệu dùng để kết nối giữa các khối logic hoặc để đồng bộ các quá trình. Tín hiệu có thể được khai báo trong phần khai báo gói, khi đó ta sẽ có tín hiệu là toàn cục, khai báo trong thực thể khi đó tín hiệu là tín hiệu toàn cục của thực thể, trong khai báo kiến trúc, và khai báo trong các khối. Các tín hiệu tuyệt đối không được khai báo trong các quá trình và các chương trình con mà chỉ được sử dụng trong chúng, đặc điểm này thể hiện sự khác biệt rõ nhất giữa biến và tín hiệu.

Cú pháp

```
signal identifier : type [range value] [:= initial
value];
```


Ví dụ:

```
signal a        :  std_logic := '0';
signal vector_b :  std_logic_vector(31 downto 0);
```

4.2. Kiểu dữ liệu

4.2.1 Các kiểu dữ liệu tiền định nghĩa

Trong các kiểu dữ liệu của VHDL có chia ra dữ liệu tiền định nghĩa và dữ liệu người dùng định nghĩa. Dữ liệu tiền định nghĩa là dữ liệu được định nghĩa trong các bộ thư viện chuẩn của VHDL, dữ liệu người dùng định nghĩa là các dữ liệu được định nghĩa lại dựa trên cơ sở dữ liệu tiền định nghĩa, phù hợp cho từng trường hợp sử dụng khác nhau.

Các dữ liệu tiền định nghĩa được mô tả trong các thư viện STD, và IEEE, cụ thể như sau:

- BIT, và BIT_VECTOR, được mô tả trong thư viện STD.STANDARD, BIT chỉ nhận các giá trị '0', và '1'. Ngầm định nếu như các biến dạng BIT không được khởi tạo giá trị ban đầu thì sẽ nhận giá trị 0. Vì BIT chỉ nhận các giá trị tường minh nên không phù hợp khi sử dụng để mô tả thực thể phần cứng thật, thay vì đó thường sử dụng các kiểu dữ liệu STD_LOGIC và STD_ULOGIC. Tuy vậy trong một số trường hợp ví dụ các lệnh của phép dịch, hay lệnh WRITE chỉ hỗ trợ cho BIT và BIT_VECTOR mà không hỗ trợ STB_LOGIC và STD_LOGIC_VECTOR.

- STD_ULOGIC và STD_ULOGIC_VECTOR, STD_LOGIC, STD_ULOGIC được mô tả trong thư viện IEEE.STD_LOGIC_1164, STD_ULOGIC, và STD_LOGIC có thể nhận một trong 9 giá trị liệt kê ở bảng sau:

'U'	Không xác định (Unresolved)	-
'X'	X	Bắt buộc
'0'	0	Bắt buộc
'1'	1	Bắt buộc
'Z'	Trở kháng cao	-
'W'	X	Yếu
'L'	0	Yếu
'H'	1	Yếu
'-'	Không quan tâm	-

Tên đầy đủ của `STD_ULOGIC là standard Unresolved Logic`, hay kiểu logic chuẩn không xác định, đối với kiểu `STD`_ULOGIC thì khi thiết kế

không cho phép để một tín hiệu có nhiều nguồn cấp. Nếu quan sát trong file stdlogic.vhd ta sẽ gặp đoạn mã mô tả hàm resolved, hai tín hiệu giá trị kiểu STD_LOGIC khi kết hợp với nhau theo quy tắc trong bảng rosolved table dưới đây.

```
    SUBTYPE std_logic IS resolved std_ulogic;
    TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF
std_ulogic;
    ---------------------------------------------------------
        -- resolution function
    ---------------------------------------------------------
        CONSTANT resolution_table : stdlogic_table := (
    ---------------------------------------------------------
     --  |  U    X    0    1    Z    W    L    H    -
    ---------------------------------------------------------
    ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
    ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
    ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
    ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
    ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
    ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
    ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
            );
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN
std_ulogic IS
    VARIABLE result : std_ulogic := 'Z';  -- weakest state
default
        BEGIN
            IF    (s'LENGTH = 1) THEN    RETURN s(s'LOW);
            ELSE
                FOR i IN s'RANGE LOOP
                    result := resolution_table(result, s(i));
                END LOOP;
            END IF;
            RETURN result;
        END resolved;
```

Ví dụ đoạn mã sau đây sẽ gây ra lỗi biên dịch:

```
    ----------------------------------------
    library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
    use IEEE.STD_LOGIC_ARITH.ALL;
    ----------------------------------------

    entity logic_expample is
    port(
        A : in  std_ulogic_vector(8 downto 0);
```

```
        U : out std_ulogic_vector(8 downto 0)
        );
    end logic_expample;
    ---------------------------------------
    architecture dataflow of logic_expample is
    begin
        U <= A;
        U <= "X01ZWLH-1";
    end dataflow;
```

Lỗi biên dịch sẽ báo rằng tín hiệu U có hai nguồn đầu vào.

```
    # ** Error: logic_example.vhd(19): Nonresolved signal 'u'
has multiple sources.
```

Đối với tín hiệu STD_LOGIC còn được gọi là resolved logic, đối với tín hiệu này bằng resolved table trong thư viện sẽ quy định cụ thể cách kết hợp các tổ hợp đầu vào để giải quyết xung đột với trường hợp một tín hiệu có nhiều đầu vào. Các xung đột được giải quyết bằng "hàm chuyển" (resolution function) Như đối với đoạn mã trên nếu khai báo U là tín hiệu dạng STD_LOGIC thì vẫn biên dịch được bình thường.

Trên thực tế tùy vào đối tượng cụ thể mà dùng các kiểu tương ứng nhưng để đảm bảo tránh việc ghép nhiều đầu vào chung (việc này cấm kị khi thiết kế mạch thật), thì nên sử dụng STD_ULOGIC.

- BOOLEAN: có các giá trị TRUE, FALSE (đúng/sai).

-INTEGER: số nguyên 32 bits (từ -2.147.483.647 đến +2.147.483.647)

- NATURAL: mso nguyên không âm (từ 0 đến +2.147.483.647)

- REAL: số thực nằm trong khoảng (từ -1.0E38 đến +1.0E38).

- TIME: sử dụng đối với các đại lượng vật lý, như thời gian,điện áp,…Hữu ích trong mô phỏng

- CHARACTER: ký tự ASCII.

- FILE_OPEN_KIND: kiểu mở file gồm các giá trị MODE, WRITE_MODE, APPEND_MODE.

- FILE_OPEN_STATUS: Trạng thái mở file với các giá trị OPEN_OK, STATUS_ERROR, NAME_ERROR, MODE_ERROR.

- SEVERITY_LEVEL: trạng thái lỗi với các giá trị NOTE, WARNING, ERROR, FAILURE.

4.2.2 Các kiểu dữ liệu vô hướng

Dữ liệu vô hướng trong VHDL (*Scalar types*) bao gồm kiểu liệt kê (`enumeration`), kiểu nguyên (`integer`), kiểu số thực dấu phảy động (`real`), kiểu dữ liệu vật lý (`physical type`). Các kiểu dữ liệu dưới đây được xét như các đối tượng dữ liệu người dùng định nghĩa từ các đối tượng dữ liệu tiền định nghĩa ở trên.

4.2.2.1. Kiểu liệt kê

Kiểu liệt kê (*Enumeration*) được định nghĩa bằng cách liệt kê tất cả các giá trị có thể có của kiểu, khai báo như sau

```
type enum_name is (enum_literals list);
```

Ví dụ:

```
type MULTI_LEVEL_LOGIC is (LOW, HIGH, RISING,
FALLING,AMBIGUOUS);
type BIT is ('0','1');
type SWITCH_LEVEL is ('0','1','X');
```

Các kiểu liệt kê định sẵn trong VHDL là:

- kiểu ký tự (CHARACTER).

- kiểu (BIT) gồm hai giá trị 0, 1.

- kiểu logic (BOOLEAN) gồm các giá trị TRUE, FALSE.

- kiểu SEVERITY kiểu cảnh báo lỗi gồm các giá trị NOTE, WARNING, ERROR, FAILURE.

- kiểu dạng và trạng thái File (FILE_OPEN_KIND với các giá trị READ_MODE, WRITE_MODE, APPEND_MODE, kiểu FILE_OPEN_STATUS với các giá trị OPEN_OK, STATUS_ERROR, NAME_ERROR, MODE_ERROR).

4.2.2.2 Kiểu số nguyên

Kiểu số nguyên (*integer*) được định nghĩa sẵn trong VHDL là INTEGER có giới hạn từ –2147483647 đến +2147483647. Các phép toán thực hiện trên kiểu nguyên là +, -, *, /. Các kiểu nguyên thứ sinh được khai báo theo cú pháp sau:

```
type identifier is range interger_range;
```

Trong đó interger_range là một miền con của tập số nguyên, các giá trị giới hạn của miền con phải viết dưới dạng số nguyên và có thể nhận giá trị âm hoặc dương, ví dụ:

```
type word_index is range 30 downto 0;
type TWOS_COMPLEMENT_INTEGER is range -32768 to 32767;
```

4.2.2.3 Kiểu số thực

Kiểu số thực (*Real*) được định nghĩa sẵn trong VHDL là Real có giới hạn từ –1.0E38 tới +1.0E38. Các kiểu thực thứ sinh được khai báo theo cú pháp sau:

```
type identifier is range real_range;
```

Trong đó real_range là miền con của miền số thực các giá trị giới hạn của miền này có thể là các giá trị dương hoặc âm được viết bằng một trong những dạng cho phép của số thực như dạng dấu phẩy động hay dạng thập phân và không nhất thiết phải giống nhau, ví dụ:

```
type my_float1      is range 1.0    to 1.0E6;
type type my_float2 is range -1.0e5 to 1.0E6;
```

4.2.2.4 Kiểu giá trị đại lượng vật lý

Kiểu giá trị đại lượng vật lý (*physical*) được dùng để định nghĩa các đơn vị vật lý như thời gian, khoảng cách, diện tích, … Chỉ có một kiểu giá trị đại lượng vật lý được định nghĩa sẵn trong VHDL là kiểu TIME

```
type Time is range --implementation defined-- ;
  units
     fs;             -- femtosecond
     ps  = 1000 fs;  -- picosecond
     ns  = 1000 ps;  -- nanosecond
     us  = 1000 ns;  -- microsecond
     ms  = 1000 us;  -- millisecond
     sec = 1000 ms;  -- second
     min = 60  sec;  -- minute
     hr  = 60  min;  -- hour
end units;
```

Các kiểu giá trị đại lượng vật lý được khai báo với cú pháp tương tự như trên, sau từ khóa **units** là đơn vị chuẩn và các đơn vị thứ sinh bằng một số nguyên lần các đơn vị chuẩn, danh sách các đơn vị được kết thúc bởi từ khóa **end units**, ví dụ:

```
type distance is range 0 to 1E16
units
```

```
    Ang;                    -- angstrom
    nm    = 10 Ang;         -- nanometer
    um    = 1000 nm;        -- micrometer (micron)
    mm    = 1000 um;        -- millimeter
    cm    = 10 mm;          -- centimeter
    dm    = 100 mm;         -- decameter
    m     = 1000 mm;        -- meter
    km    = 1000 m;         -- kilometer
    mil   = 254000 Ang;     -- mil (1/1000 inch)
    inch  = 1000 mil;       -- inch
    ft    = 12 inch;        -- foot
    yd    = 3 ft;           -- yard
    fthn  = 6 ft;           -- fathom
    frlg  = 660 ft;         -- furlong
    mi    = 5280 ft;        -- mile
    lg    = 3 mi;           -- league
end units;
```

4.2.2 Dữ liệu phức hợp

Dữ liệu phức hợp (*composite*) là dữ liệu tạo thành bằng các tổ hợp các dạng dữ liệu cơ bản trên theo các cách khác nhau. Có hai dạng dữ liệu phức hợp cơ bản là kiểu mảng dữ liệu khi các phần tử dữ liệu trong tổ hợp là đồng nhất và được sắp thứ tự, dạng thứ hai là dạng bản ghi khi các phần tử có thể có các kiểu dữ liệu khác nhau.

4.2.2.1. Kiểu mảng

Kiểu mảng (*Array*) trong VHDL có các tính chất như sau:

- Các phần tử của mảng có thể là mọi kiểu trong ngôn ngữ VHDL.
- Số lượng các chỉ số của mảng (hay số chiều của mảng) có thể nhận mọi giá trị nguyên dương.
- Mảng chỉ có một và một chỉ số dùng để truy cập tới phần tử.
- Miền biến thiên của chỉ số xác định số phần tử của mảng và hướng sắp xếp chỉ số trong của mảng từ cao đến thấp hoặc ngược lại.
- Kiểu của chỉ số là kiểu số nguyên hoặc liệt kê.

Mảng trong VHDL chia làm hai loại là mảng ràng buộc và mảng không ràng buộc. Mảng ràng buộc là mảng được khai báo tường minh có kích thước cố định. Cú pháp khai báo của mảng này như sau:

```
Type array_name is array (index_range) of type;
```

Trong đó array_name là tên của mảng, index_range là miền biến thiên xác định của chỉ số nếu mảng là mảng nhiều chiều thì các chiều biến thiên cách nhau dấu “,”, ví dụ như sau:

```
type mem  is array (0 to 31, 3 to 0) of std_logic;
type word is array (0 to 31) of bit;
type data is array (7 downto 0) of word;
```

đối với mảng khai báo không tương minh thì miền giá trị của chỉ số không được chỉ ra mà chỉ ra kiểu của chỉ số:

```
type array_name is array (type of index range <>) of type;
```

Ví dụ:

```
type mem    is array (natural range <>) of word;
type matrix is array (integer range <>,
                      integer range <>) of real;
```

Cách truy cập tới các phần tử của mảng của một mảng n chiều như sau:

```
array_name (index1, index 2,…, indexn)
```

ví dụ:

```
matrix(1,2), mem (3).
```

4.2.2.1. Kiểu bản ghi

Bản ghi (*Record*) là nhóm của một hoặc nhiều phần tử thuộc những kiểu khác nhau và được truy cập tới như một đối tượng. Bản ghi có những đặc điểm như sau:

- Mỗi phần tử của bản ghi được truy cập tới theo trường.
- Các phần tử của bản ghi có thể nhận mọi kiểu của ngôn ngữ VHDL kể cả mảng và bản ghi.

ví dụ về bản ghi

```
type stuff is
   record
       I : integer;
       X : real;
       day : integer range 1 to 31;
       name : string(1 to 48);
       prob : matrix(1 to 3, 1 to 3);
   end record;
```

Các phần tử của bản ghi được truy cập theo tên của bản ghi và tên trường ngăn cách nhau bằng dấu ".", ví dụ:

```
node.data; stuff.day
```

5. Toán tử và biểu thức

Trong VHDL có tất cả 7 nhóm toán tử được phân chia theo mức độ ưu tiên và trật tự tính toán. Trong bảng sau liệt kê các nhóm toán tử theo trật tự ưu tiên tăng dần:

Toán tử logic	and, or, nand, nor, xor
Các phép toán quan hệ	=, /=, <, <=, >, >=
Các phép toán dịch	sll, srl, sla, sra, rol, ror
Các phép toán cộng, hợp	+, -, &
Toán tử dấu	+, -
Các phép toán nhân	*, /, mod, rem
Các phép toán khác	**, abs, not

Các quy định về trật tự các phép toán trong biểu thức được thực hiện như sau:

- Trong các biểu thức phép toán có mức độ ưu tiên lớn hơn sẽ được thực hiện trước. Các dấu ngoặc đơn "(", ")" phân định miền ưu tiên của từng nhóm biểu thức.
- Các phép toán trong nhóm với cùng một mức độ ưu tiên sẽ được thực hiện lần lượt từ trái qua phải.

5.1 Toán tử logic

Các phép toán logic gồm **and, or, nand, nor, xor,** và **not.** Các phép toán này tác động lên các đối tượng kiểu BIT và Boolean và mảng một chiều kiểu BIT. Đối với các phép toán hai ngôi thì các toán hạng nhất định phải cùng kiểu, nếu hạng tử là mảng BIT một chiều thì phải có cùng độ dài, khi đó các phép toán logic sẽ thực hiện đối với các bit tương ứng của hai toán hạng có cùng chỉ số. Phép toán **not** chỉ có một toán hạng, khi thực hiện với mảng BIT một chiều thì sẽ cho kết quả là mảng BIT lấy đảo ở tất cả các vị trí của mảng cũ.

Ví dụ:

```
-----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------
entity logic_example is
```

```
    port(
         in1   : in   std_logic_vector (5 downto 0);
         in2   : in   std_logic_vector (5 downto 0);
         out1  : out  std_logic_vector (5 downto 0)
         );
end entity;
-------------------------------------------
architecture rtl of logic_example is
begin
    out1(0) <= in1(0) and  in2 (0);
    out1(1) <= in1(1) or   in2 (1);
    out1(2) <= in1(2) xor  in2 (2);
    out1(3) <= in1(3) nor  in2 (3);
    out1(4) <= in1(4) nand in2 (4);
    out1(5) <= in1(5) and  (not in2 (5));
end rtl;
-------------------------------------------
```

5.2 Các phép toán quan hệ

Các phép toán quan hệ gồm =, /=, <, <=, >, >= thực hiện các phép toán so sánh giữa các toán tử có cùng kiểu như Integer, real, character. Và cho kết quả dạng Boolean. Việc so sánh các hạng tử trên cơ sở miền giá trị của các đối tượng dữ liệu. Các phép toán quan hệ rất hay được sử dụng trong các câu lệnh rẽ nhánh

Ví dụ đầy đủ về phép toán so sánh thể hiện ở đoạn mã sau:

```
-------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
-------------------------------------------
entity compare_example is
    port(
         val1   : in   std_logic_vector (7 downto 0);
         val2   : in   std_logic_vector (7 downto 0);
          res   : out  std_logic_vector (2 downto 0)
         );
 end entity;
-------------------------------------------
 architecture rtl of compare_example is
 begin
    compare: process (val1, val2)
    begin
        if val1 > val2 then res(0) <= '1';
        else res (0) <= '0'; end if;
        if val1 = val2 then res(1) <= '1';
        else res (1) <= '0'; end if;
```

```
            if val1 < val2 then res(2) <= '1';
            else res (2) <= '0'; end if;
        end process compare;
end rtl;
```

5.3 Các phép toán dịch

Các phép toán quan hệ gồm **sll, srl, sla, sra, rol, ror** được hỗ trợ trong thư viện ieee.numeric_bit, và ieee.numeric_std. Cú pháp của các lệnh dịch có hai tham số là sho (shift operand) và shv (shift value), ví dụ cú pháp của sll như sau

```
sha  sll shv;
```

Toán tử	Phép toán	Kiểu của sho	Kiểu của shv	Kiểu kết quả
sll	Dịch trái logic	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
srl	Dịch phải logic	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
sla	Dịch trái số học	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
sra	Dịch phải số học	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
rol	Dịch vòng tròn sang trái	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho
ror	Dịch vòng tròn phải	Mảng 1 chiều kiểu BIT hoặc BOOLEAN	Integer	Cùng kiểu sho

Đối với dich logic thì tại các vị trí bị trống sẽ được điền vào các giá trị '0' còn dịch số học thì các các vị trí trống được thay thế bằng bit có trọng số cao nhất MSB nếu dịch phải, và thay bằng bit có trọng số thấp nhất nếu dịch trái. Đối với dịch vòng thì các vị trí khuyết đi sẽ được điền bằng các bit dịch ra ngoài giới hạn của mảng. Quan sát ví dụ dưới đây

Giả sử có giá trị sho = "11000110" , shv = 2, xét đoạn mã sau

```
----------------------------------------
library ieee;
USE ieee.Numeric_STD.all;
USE ieee.Numeric_BIT.all;
library STD;
use STD.TEXTIO.ALL;
----------------------------------------
entity shift_example is
```

```
end entity;
-----------------------------------------
architecture rtl of shift_example is
signal sho: bit_vector(7 downto 0) := "11000110";
begin
shifting: process (sho)
    variable str:line;
    begin
    write(str, string'("sho sll 2 = "));
    write(str, sho sll 2);
    writeline(OUTPUT, str);
    write(str, string'("sho srl 2 = "));
    write(str, sho srl 2);
    writeline(OUTPUT, str);
    write(str, string'("sho sla 2 = "));
    write(str, sho sla 2);
    writeline(OUTPUT, str);
    write(str, string'("sho sra 2 = "));
    write(str, sho sra 2);
    writeline(OUTPUT, str);
    write(str, string'("sho rol 2 = "));
    write(str, sho rol 2);
    writeline(OUTPUT, str);
    write(str, string'("sho ror 2 = "));
    write(str, sho ror 2);
    writeline(OUTPUT, str);
    end process shifting;
end architecture;
-----------------------------------------------
```

Với đoạn mã trên khi mô phỏng sẽ thu được kết quả:

```
# sho sll 2 = 00011000
# sho srl 2 = 00110001
# sho sla 2 = 00011000
# sho sra 2 = 11110001
# sho rol 2 = 00011011
# sho ror 2 = 10110001
```

5.4 Các phép toán cộng trừ và hợp

Các phép toán **+, -** là các phép tính hai ngôi, các phép toán cộng và trừ có cú pháp thông thường, hạng tử của phép toán này là tất cả các kiểu dữ liệu kiểu số gồm INTEGER, REAL.

Toán tử	Phép toán	Kiểu toán tử trái	Kiểu của shv	Kiểu kết quả
+	Phép cộng	Dữ liệu kiểu số	Cùng kiểu	Cùng kiểu
-	Phép trừ	Dữ liệu kiểu số	Cùng kiểu	Cùng kiểu

&	Phép hợp	Kiểu mảng hoặc phần tử mảng	Kiểu mảng hoặc phần tử mảng	Kiểu mảng

Phép toán hợp (concatennation) **&** có đối số có thể là mảng hoặc phần tử mảng và kết quả hợp tạo ra một mảng mới có các phần tử ghép bởi các phần tử của các toán tử hợp, các ví dụ dưới đây sẽ minh họa rõ thêm cho phép hợp

```
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
----------------------------------------
entity dublicate is
    port(
         in1  : in  std_logic_vector(3 downto 0);
         out1 : out std_logic_vector(7 downto 0)
         );
end entity;
----------------------------------------
architecture rtl of dublicate is
begin
    out1 <= in1 & in1;
end architecture;
----------------------------------------
```

Trong ví dụ trên nếu gán giá trị in1 = “1100” thu được tương ứng ở đầu ra out1 = “11001100”.

5.5 Các phép dấu

Các phép toán quan hệ gồm **+, -,** thực hiện với các giá trị dạng số và trả về giá trị dạng số tương ứng.

5.6 Các phép toán nhân chia, lấy dư

Các phép toán ***, /, mod, rem*** là các phép toán hai ngôi tác động lên các toán tử kiểu số theo như bảng sau:

Toán tử	Phép toán	Toán tử trái	Toán tử phải	Kiểu kết quả
*	Phép nhân	Số nguyên Integer	Cùng kiểu	Cùng kiểu
		Số thực dấu phẩy động REAL	Cùng kiểu	Cùng kiểu
/	Phép chia	Số nguyên Integer	Cùng kiểu	Cùng kiểu
		Số thực dấu phẩy động REAL	Cùng kiểu	Cùng kiểu

Mod	Lấy module	Số nguyên Integer	Số nguyên Integer	Số nguyên Integer
Rem	Lấy phần dư (remainder)	Số nguyên Integer	Số nguyên Integer	Số nguyên Integer

Có thể kiểm tra các phép toán số học bằng đoạn mã sau:

```
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
----------------------------------------
entity mul_div_example is
    port(
         m1, m2                  : in  integer;
         res1, res2, res3, res4  : out integer
        );
end entity;
----------------------------------------
architecture rtl of mul_div_example is
begin
    res1 <= m1 *   m2;
    res2 <= m1 /   m2;
    res3 <= m1 mod m2;
    res4 <= m1 rem m2;
end architecture;
----------------------------------------
```

5.6 Các phép toán khác

Các phép toán ******, **abs** là các phép toán lấy mũ e và lấy giá trị tuyệt dối một ngôi tác động lên các toán tử kiểu số theo như bảng sau:

Toán tử	Phép toán	Toán tử	Kiểu kết quả
**	Lấy mũ e ** a = e^a	Số nguyên Integer	Cùng kiểu
		Số thực dấu phẩy động REAL	Cùng kiểu
abs	Lấy trị tuyệt đối	Số nguyên Integer	Cùng kiểu
		Số thực dấu phẩy động REAL	Cùng kiểu

6. Phát biểu tuần tự

Trong ngôn ngữ VHDL phát biểu tuần tự (*sequential statement*) được sử dụng để diễn tả thuật toán thực hiện trong một chương trình con (subprogram) tức là dạng function hoặc procedure hay trong một quá trình

(process). Các lệnh tuần tự sẽ được thực thi một cách lần lượt theo thứ tự xuất hiện của chúng trong chương trình.

Các dạng phát biểu tuần tự bao gồm: phát biểu đợi (`wait statement`), xác nhận (`assert statement`), báo cáo (`report statement`), gán tín hiệu (signal assignment statement), gán biến (`variable assignment statement`), gọi thủ tục (procedure call statement), các lệnh rẽ nhánh và vòng lặp, lệnh điều khiển `if, loop, case, exit, return, next statements`), lệnh trống (`null statement`).

6.1. Phát biểu đợi

Phát biểu đợi (**`wait`**) có cấu trúc như sau

```
wait [sensitivity clause] [condition clause] [time clause];
```

Trong đó :

- `sensitivity clause` = **`on`** `sensitivity list`, dang sách các tín hiệu cần theo dõi, nếu câu lệnh wait dùng cấu trúc này thì nó có ý nghĩa bắt quá trình đợi cho tới khi có bất kỳ sự thay đổi nào của các tín hiệu trong danh sách theo dõi. Cấu trúc này tương đương với cấu trúc **process** dùng cho các phát biểu đồng thời sẽ tìm hiểu ở 7.2.
- `Condition clause = until condition`: trong đó `condition` là điều kiện dạng Boolean. Cấu trúc này bắt quá trình dừng cho tới khi điều kiện trong condition được thỏa mãn.
- `Time clause` = **`for`** `time_expression`; có ý nghĩa bắt quá trình dừng trong một khoảng thời gian xác định chỉ ra trong tham số lệnh.

Ví dụ về các kiểu gọi lệnh wait:

```
wait for   10 ns;  -- timeout clause, specific time delay
wait until clk='1';-- condition clause, Boolean condition
wait until S1 or S2; -- Boolean condition
wait on   sig1, sig2;-- sensitivity clause, any event on any
```

Câu lệnh wait nếu sử dụng trong process thì process không được có danh sách tín hiệu theo dõi (sensitive list), lệnh wait on tương đương với cách sử dụng process có danh sách tín hiệu theo dõi. Xem xét ví dụ đầy đủ dưới đây.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity AND3 is
port (
     A : in  std_logic;
     B : in  std_logic;
     C : in  std_logic;
     S : out std_logic
     );
end AND3;
----------------------------------------
architecture wait_on of AND3 is
begin
    andd: process
    begin
        S <= A and B and C;
        wait on A, B, C;
    end process andd;
end wait_on;
----------------------------------------
architecture use_process of AND3 is
begin
    andd: process  (A, B, C)
    begin
        S <= A and B and C;
    end process andd;
end use_process;
----------------------------------------
```

Module mô tả ở trên thực hiện một hàm AND 3 đầu vào, với kiến trúc `wait_on` sử dụng câu trúc câu lệnh **`wait on`** kèm theo danh sách tín hiệu theo dõi A, B, C. Cách sử dụng đó tương đương với cách viết trong kiến trúc `use_process` khi không dùng wait_on mà sử dụng danh sách tín hiệu theo dõi ngay sau từ khóa **process**.

Với lệnh wait for xem xét ví dụ đầy đủ dưới đây:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity wait_for is
    port(
         A : out std_logic;
         B : out std_logic
         );
end wait_for;
----------------------------------------------
architecture behavioral of wait_for is
```

```
begin
waiting: process
    begin
        A <= 'Z'; B <= 'X';
        wait for 100 ns;
        A <= '1'; B <= 'Z';
        wait for 200 ns;
        A <= '0'; B <= '1';
        wait;
    end process waiting;
end behavioral;
-----------------------------------------
```

Trong ví dụ này các tín hiệu A, B, được gán các giá trị thay đổi theo thời gian, khoảng thời gian được quy định bằng các lệnh `wait for`. Ở ví dụ trên ban đầu A, B nhận các giá trị là Z và X sau 100 ns A bằng 1 còn B bằng Z, sau tiếp 200 ns A nhận giá trị bằng 0 và B bằng 1.

Ở đây ta cũng gặp cấu trúc lệnh wait không có tham số, lệnh này tương đương lệnh đợi trong khoảng thời gian là vô hạn, các tín hiệu A, B khi đó được giữ nguyên giá trị được gán ở câu lệnh trước. Cấu trúc này là các cấu trúc chỉ dùng cho mô phỏng, đặc biệt hay dùng trong các module kiểm tra, phục vụ cho việc tạo các xung đầu vào.

Ví dụ sau đây là ví dụ về lệnh `wait until`

```
-----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----------------------------------------
entity wait_until is
port (D : in  std_logic;
      Q : out std_logic
     );
end wait_until;
-----------------------------------------
architecture behavioral of wait_until is
signal clk : std_logic := '0';
begin
    create_clk: process
    begin
        wait for 100 ns;
        clk <= not clk after 100 ns;
        end process create_clk;
    latch: process
    begin
        wait until clk = '1';
```

```
            Q <= D;
        end process latch;
    end behavioral;
    ----------------------------------------
```

Ví dụ này mô tả phần tử latch, giá trị đầu ra nhận giá trị đầu vào tại mỗi thời điểm mà giá trị tín hiệu đồng bộ bằng 1 trong đó xung nhịp clk được tạo bằng cấu trúc `clk <= not clk after 100 ns;` nghĩa là chu kỳ tương ứng bằng 200 ns.

6.2. Phát biểu xác nhận và báo cáo

Phát biểu xác nhận và báo cáo (*assert and report statement*) có cấu trúc như sau

```
    assert boolean_condition [report string] [ severity name
];
```

Trong đó :

`boolean_condition`: điều kiện dạng Boolean. Cấu trúc này kiểm tra giá trị của `boolean_condition`.

`report string`: báo cáo lỗi nếu được sử dụng thì phải đi cùng một chuỗi thông báo lỗi

`severity name`:thông báo mức độ lỗi nếu được sử dụng thì phải đi kèm với các giá trị định sẵn bao gồm NOTE, WARNING, ERROR, FAILURE

Lệnh report có thể được sử dụng độc lập với assert khi đó nó có cấu trúc

```
    report string  [ severity name ] ;
```

Các ví dụ về lệnh **assert** và **report**:

```
    assert a =(b or c);
    assert j<i report "internal error, tell someone";
    assert clk='1' report "clock not up" severity WARNING;

    report  "finished pass1";  -- default severity name is
NOTE
    report  "inconsistent data." severity FAILURE;
```

6.3. Phát biểu gán biến

Trong ngôn ngữ VHDL cú pháp của phát biểu gán biến (*variable assignment statement*) tương tự như phép gán giá trị của biến như trong các ngôn ngữ lập trình khác, cú pháp như sau:

```
variable := expression;
```

Trong đó `variable` là các đối tượng chỉ được phép khai báo trong các chương trình con và các quá trình, các biến chỉ có tác dụng trong chương trình con hay quá trình khai báo nó, `expression` có thể là một biểu thức hoặc một hằng số có kiểu giống kiểu của variable. Quá trình gán biến diễn ra tức thì với thời gian mô phỏng bằng 0 vì biến chỉ có tác dụng nhận các giá trị trung gian. Ví dụ về gán biến.

```
A   := -B + C * D / E mod F rem G abs H;
Sig := Sa and Sb or Sc nand Sd nor Se xor Sf xnor Sg;
```

Ví dụ sau đây minh họa cho phát biểu gán biến và phát biểu **assert**. Trong ví dụ này một bộ đếm được khai báo dưới dạng biến, tại các thời điểm sườn lên của xung nhịp đồng hồ giá trị counter được tăng thêm 1 đơn vị. Lệnh **assert** sẽ kiểm soát và thông báo khi nào giá trị counter vượt quá 100 và thông báo ra màn hình.

```
-----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----------------------------------------
entity assert_example is
end    assert_example;
-----------------------------------------
architecture behavioral of assert_example is
signal clk: std_logic := '0';
begin
    create_clk: process
    begin
       wait for 100 ns;
       clk <= not clk after 100 ns;
    end process create_clk;
    count: process (clk)
    variable counter : integer := 0;
    begin
        if clk'event and clk = '1' then
            counter := counter + 1;
        end if;
        assert counter <100 report "end of counter"
severity NOTE;
```

```
        end process count;
    end behavioral;
    ------------------------------------------------
```

Khi mô phỏng ta thu được thông báo sau ở màn hình khi counter vượt quá giá trị 100.

```
    # ** Note: end of counter
    #    Time: 20 us  Iteration: 0  Instance: /assert_example
```

6.4. Phát biểu gán tín hiệu

Phát biểu gán tín hiệu (*signal assignment statement*) có cấu trúc như sau

```
    target <= [ delay_mechanism ] waveform ;
```

Trong đó :

- `target`: đối tượng cần gán tín hiệu.
- `Delay mechanism`: cơ chế làm trễ tín hiêu có cấu trúc như sau:

```
transport|[ reject time_expression ]inertial
```

Trong đó nếu có từ khóa **`transport`** thì tín hiệu được làm trễ theo kiểu đường truyền là dạng làm trễ mà không phụ thuộc vào dạng tín hiệu đầu vào, xung đầu vào dù có độ rộng xung như thế nào vẫn được truyền tải đầy đủ. Nếu không có cơ chế làm trễ nào được chỉ ra hoặc sử dụng cấu trúc **inertial** thì cơ chế làm trễ là cơ chế quán tính (inertial delay), theo cơ chế này thì sẽ có một giới hạn độ rộng xung đầu vào được chỉ ra gọi là pulse reject limit được chỉ ra cùng từ khóa **`reject`**, nếu tín hiệu vào có độ rộng xung bé hơn giới hạn này thì không đủ cho phần tử logic kế tiếp thực hiện thao tác chuyển mạch và gây ra lỗi. Nếu giới hạn thời gian chỉ ra sau **reject** là giá trị âm cũng gây ra lỗi.

```
    waveform := wave_form elements := signal + {after
time_expression};
```

- Chỉ ra tín hiệu gán giá trị và thời gian áp dụng cho cơ chế làm trễ đã trình bày ở phần trên.

Ví dụ về gán tín hiệu tuần tự:

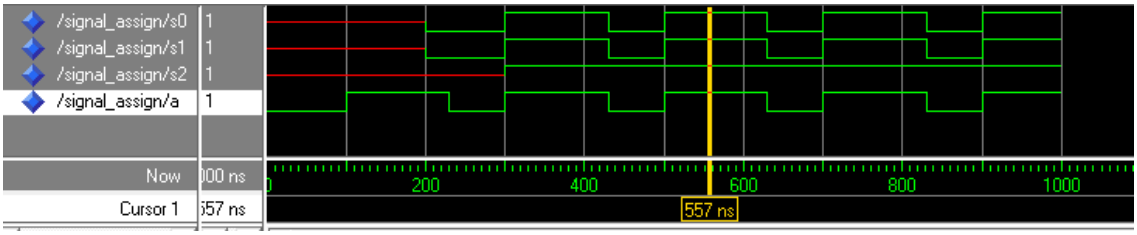
```
    -- gán với trễ quán tính, các lệnh sau tương đương
    Output_pin <= input_pin after 10 ns;
    Output_pin <= inertial input_pin after 10 ns;
    Output_pin <= reject 10 ns inertial Input_pin after 10
ns;
```

```
     -- gán với kiểm soát độ rộng xung đầu vào
     Output_pin <= reject 5 ns inertial Input_pin after 10 ns;
     Output_pin <= reject 5 ns inertial Input_pin after 10 ns,
not Input_pin after 20 ns;
     -- Gán với trễ transport
     Output_pin <= transport Input_pin after 10 ns;
     Output_pin <= transport Input_pin after 10 ns, not
Input_pin after 20 ns;
     -- tương đương các lệnh sau
     Output_pin <= reject 0 ns inertial Input_pin after 10 ns;
     Output_pin <= reject 0 ns inertial Input_pin after 10 ns,
not Input_pin after 10 ns;
```

Xét một ví dụ đầy đủ về lệnh này như sau như sau:

```
     ----------------------------------------
     library IEEE;
     use IEEE.STD_LOGIC_1164.ALL;
     ----------------------------------------
     entity signal_assign is
     port (
           S0 : out std_logic;
           S1 : out std_logic;
           S2 : out std_logic
           );
     end signal_assign;
     ----------------------------------------
     architecture behavioral of signal_assign is
     signal A: std_logic:= '0';
     begin
         create_data: process
         begin
             wait for 100 ns;
             A <= '1';
             wait for 100 ns;
             A <= not A after 30 ns;
         end process create_data;
         andd: process (A)
         begin
             S0 <= transport A after 200 ns;
             S1 <= reject 30 ns inertial A after 200 ns;
             S2 <= reject 100 ns inertial A after 200 ns;
         end process andd;
     end behavioral;
```

Sau khi mô phỏng thu được kết quả trên waveform như sau:



Hình 2.3: Kết quả mô phỏng ví dụ gán tín hiệu trên Modelsim

Tín hiệu A là dạng tín hiệu xung nhịp có chu kỳ 60 ns, A thay đổi giá trị sau mỗi nửa chu kỳ tức là sau mỗi 30 ns. Tín hiệu S0 được gán bằng A theo phương thức **`transport`** vì vậy trùng lặp hoàn toàn với A. Tín hiệu S1 gán theo phương thức **`inertial`** với thời gian reject bằng 30 ns do vậy vẫn thu được giá trị giống A. Tín hiệu S2 cũng được gán bằng phương thức **`inertial`** như với **`reject`** time bằng 100 ns > 30 ns nên có giá trị không thay đổi.

6.5. Lệnh rẽ nhánh và lệnh lặp

Lệnh rẽ nhánh là lệnh lặp là các phát biểu điều khiển quá trình trong các chương trình con hoặc trong các quá trình. Các phát biểu này có cú pháp giống như các phát biểu tương tự trong các ngôn ngữ lập trình khác.

6.5.1 Lệnh rẽ nhánh if

```
if  condition1  then
    sequence-of-statements
elsif  condition2  then
    [sequence-of-statements ]
elsif  condition3  then
    [sequence-of-statements ]
   ..
else
    [sequence-of-statements
end if;
```

Trong đó :

- `condition` : các điều kiện dạng boolean.
- `[sequence-of-statements]` : khối lệnh thực thi nếu điều kiện được thỏa mãn.

Ví dụ sau minh họa cách sử dụng lệnh `if` để mô tả D-flipflop bằng VHDL, điều kiện sườn dương của tín hiệu xung nhịp được kiểm tra bằng cấu trúc `clk = '1' and clk'event` trong đó `clk'event` thể hiện có sự kiện thay đổi giá trị trên `clk` và điều kiện clk = '1' xác định sự thay đổi đó là sườn dương của tín hiệu.

```
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity D_flipflop is
port(
     D   : in  std_logic;
     CLK : in  std_logic;
     Q   : out std_logic
     );
end D_flipflop;
----------------------------------------
architecture behavioral of D_flipflop is
begin
DFF: process (clk, D)
     begin
        if clk = '1' and clk'event then
            Q <= D;
        end if;
     end process DFF;
end behavioral;
----------------------------------------
```

6.5.2.Lệnh chọn case

Lệnh lựa chọn (*case*) có cú pháp như sau:

```
case expression  is
    when choice1 =>
        [sequence-of-statements]
    when choice2 =>
        [sequence-of-statements]
    ...

    when others =>  -- optional if all choices covered
        [sequence-of-statements]
end case;
```

Lệnh case bắt đầu bằng từ khóa **case** theo sau là một biểu thức (expression). Các trường hợp được chọn bắt đầu bằng từ khóa **when** giá trị có thể của expresstion và mã thực thi tương ứng bắt đầu sau dấu "=>". `When others` sẽ quét hết tất cả các giá trị có thể có của `expression` mà chưa được liệt kê ra ở trên (tương đương với từ khóa **default** trong ngôn ngữ lập trình C).

Ví dụ về lệnh **case:**

```
----------------------------------------
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity mux2 is
port(
     A   : in  std_logic;
     B   : in  std_logic;
     Sel : in  std_logic;
     S   : out std_logic
     );
end mux2;
----------------------------------------
architecture behavioral of mux2 is
begin
mux: process (A, B, sel)
     begin
     CASE sel IS
         WHEN '0' =>  S <= A;
         WHEN others => S <= B;
     end case;
     end process mux;
end behavioral;
----------------------------------------
```

6.5.3. Lệnh lặp

Có ba dạng lệnh lặp dùng trong VHDL là lệnh loop, lệnh while, và lệnh for:

```
loop
    sequence-of-statements -- use exit statement to get
out
end loop;
for variable in range loop
    sequence-of-statements
end loop;
while condition  loop
    sequence-of-statements
end loop;
```

Đối với lệnh lặp dạng **loop** thì vòng lặp chỉ kết thúc nếu nó gặp lệnh **exit** ở trong đó. Đối với lệnh lặp dạng **for** thì vòng lặp kết thúc khi đã quét hết tất cả các giá trị của biến chạy. Với vòng lặp dạng **while** thì vòng lặp kết thúc khi điều kiện trong **condition** là FALSE.

Ví dụ đầy đủ về ba dạng lệnh lặp:

```
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity loop_example is
```

```
port(
     vector_in : in  std_logic_vector(7 downto 0);
     out1      : out std_logic
     );
end loop_example;
-----------------------------------------
architecture loop1 of loop_example is
begin
    loop_p: process (vector_in)
    variable I : integer;
    variable p : std_logic :='1';
    begin
    for i in 7 downto 0 loop
        p := p and vector_in(i);
    end loop;
    out1 <= p;
    end process loop_p;
end loop1;
-----------------------------------------
architecture loop2 of loop_example is
begin
    loop_p: process (vector_in)
    variable i: integer   := 7;
    variable p: std_logic :='1';
    begin
    while i > 0 loop
        p := p and vector_in(i);
        i := i-1;
    end loop;
    i   := 7;
    out1 <= p;
    end process loop_p;
end loop2;
-----------------------------------------
architecture loop3 of loop_example is
begin
    loop_p: process (vector_in)
    variable i: integer := 7;
    variable p: std_logic :='1';
    begin
    loop
        p := p and vector_in(i);
        i := i-1;
        exit when i < 0;
    end loop;
    I   := 7;
    out1 <= p;
    end process loop_p;
end loop3;
-----------------------------------------
```

Ví dụ trên thể hiện ba phương pháp khác nhau dùng lệnh để hiện thực hóa hàm AND cho tổ hợp 8 bit đầu vào. Phương pháp thứ nhất sử dụng lệnh lặp xác định, phương pháp thứ hai và thứ ba sử dụng vòng lặp kiểm soát điều kiện đối với biến chạy i, về cơ bản cả ba phương pháp đều cho hiệu quả mô tả giống nhau.

7. Phát biểu đồng thời

Phát biểu đồng thời (*concurrent statements*) được sử dụng để mô tả các kết nối giữa các khối thiết kế, mô tả các khối thiết kế thông qua cách thức làm việc của nó (process statement). Hiểu một cách khác các phát biểu đồng thời dùng để mô tả phần cứng về mặt cấu trúc hoặc cách thức làm việc đúng như nó vốn có. Khi mô phỏng thì các phát biểu đồng thời hoặc được thực hiện song song độc lập với nhau. Mã VHDL không quy định về thứ tự thực hiện của các phát biểu. Bất kể phát biểu đồng thời nào có quy định thứ tự thực hiện theo thời gian đều gây ra lỗi biên dịch.

Vị trí của các phát biểu đồng thời nằm trực tiếp trong khối **begin** end của mô tả kiến trúc:


```
architecture identifier of design is
    {declarative_part}
begin
    {concurrent_statements}
end iddentifier;
```

Có tất cả 7 dạng phát biểu đồng thời: khối (block statement), quá trình (process statement), gọi thủ tục (procedure call statement), xác nhận gán tín hiệu (signal assignment statement), khai báo module con (component declaration statement), và phát biểu phát sinh (generate statement).

7.1. Phát biểu khối

Phát biểu khối (*Block statement*) là một khối các cấu lệnh song song thể hiện một phần của thiết kế, các khối có thể được khai báo lồng ghép trong nhau và có tính chất kế thừa. Phát biểu khai báo block có cấu trúc như sau:

```
block [(guard_expression)] [is]
    block_header
    block_declarative_part
begin
    block_statement_part
end block;
```

Trong đó :

- guard_expression: nếu được sử dụng thì có một tín hiệu tên là GUARD có kiểu Boolean được khai báo một cách không tường minh bằng một biểu thức gán giá trị cho tín hiệu đó, tín hiệu GUARD có thể sử dụng để điều khiển các thao tác bên trong khối.
- block_header và block_declarative_part: là các khai báo và gán giá trị cho các tham số generics, ports.
- block_statement_part: khối chính của block chứa các lệnh đồng thời.

Ví dụ về các sử dụng block:

```
clump : block
          begin
            A <= B or C;
            D <= B and not C;
          end block clump ;

maybe : block ( B'stable(5 ns) ) is
             port (A, B, C : inout std_logic );
             port map ( A => S1, B => S2, C => outp );
             constant delay: time := 2 ns;
             signal temp: std_logic;
          begin
             temp <= A xor B after delay;
             C <= temp nor B;
          end block maybe;
```

Xét một ví dụ đầy đủ về sử dụng **guarded block** để mô phỏng một bus ba trạng thái như sau:

```
-----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
-----------------------------------------------
entity bus_drivers is
end bus_drivers;
-----------------------------------------------
architecture behavioral of bus_drivers is
signal TBUS: STD_LOGIC := 'Z';
signal A, B, OEA, OEB : STD_LOGIC:= '0';
begin
    process
    begin
        OEA <= '1' after 100 ns, '0' after 200 ns;
        OEB <= '1' after 300 ns;
         wait;
    end process;
```

```
-------------------------------------------
    B1 : block (OEA = '1')
    begin
        TBUS <= guarded not A after 3 ns;
    end block;
-------------------------------------------
    B2 : block (OEB = '1')
    begin
        TBUS <= guarded not B after 3 ns;
    end block;
end behavioral;
```

Trong ví dụ trên TBUS được kết nối giá trị đảo của A, B chỉ trong trường hợp các tín hiệu cho phép tương ứng OEA, OEB bằng 1. Điều kiện này được khai báo ẩn trong tín hiệu GUARD của hai block B1 và B2.

7.2. Phát biểu quá trình

Quá trình (process) là một khối chứa các khai báo tuần tự nhưng thể hiện cách thức hoạt động của một phần thiết kế. Trong mô tả kiến trúc của module có thể có nhiều khai báo process và các khai báo này được thực hiện song song, độc lập với nhau không phụ thuộc vào thứ tự xuất hiện trong mô tả. Chính vì thế mặc dù trong khối quá trình chỉ chứa các khai báo tuần tự nhưng một khối được coi là một phát biểu đồng thời. Cấu trúc của quá trình như sau:

```
[postponed] process [(sensitivity_list)] [is]
    process_declarative_part
begin
    [process_statement_part]
end [ postponed ] process [ process_label ];
```

Trong đó :

- `postponed`: nếu được sử dụng thì quá trình là một quá trình được trì hoãn, nếu không đó là một quá trình thông thường. Quá trình trì hoãn là quá trình được thực hiện chỉ sau khi tất cả các quá trình không trì hoãn (**nonpostponded**) đã bắt đầu và kết thúc.
- `sensitivity_list`: tương tự như danh sách tín hiệu được theo dõi đối với lệnh **wait**. Nó có ý nghĩa là mọi thay đổi của các tín hiệu trong danh sách này thì các lệnh trong quá trình này sẽ được thực hiện. Khi mô tả các khối logic tổ hợp thì các danh sách này chính là các tín hiệu đầu vào, còn đối với mạch dãy thì là tín hiệu xung nhịp đồng bộ clk.
- `process_declarative_part`: phần khai báo của quá trình, lưu ý là các khai báo tín hiệu không được đặt trong phần này.

- `process_statement_part`: Phần nội dung của **process**.

Ví dụ về các sử dụng process:

```
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
----------------------------------------
entity counter4 is
    port(
        count   : out std_logic_vector( 3 downto 0);
        enable  : in  std_logic;
        clk     : in  std_logic;
        reset   : in  std_logic
        );
end entity;
----------------------------------------
architecture rtl of counter4 is
signal cnt :std_logic_vector ( 3 downto 0) := "0000";
begin
    process (clk, reset)
    begin
        if (reset = '1') then
            cnt <= "0000";
        elsif (rising_edge(clk)) then
            if (enable = '1') then
                cnt <= cnt + 1;
            end if;
        end if;
    end process;
    count <= cnt;
end architecture;
------------------------------------------------
```

Ở ví dụ trên mô tả cấu trúc một bộ đếm 4 bit bằng cú pháp của process, danh sách `sensitive list` bao gồm tín hiệu đồng bộ CLK và tín hiệu RESET, tín hiệu RESET làm việc không đồng bộ, mỗi khi RESET bằng 1 thì giá trị đếm count được đặt giá trị 0. Nếu RESET bằng o và có sườn dương của xung nhịp CLK thì giá trị đếm được tăng thêm 1.

Khối process còn rất hay được sử dụng để mô tả các tổ hợp, khi đó tất cả các đầu vào của mạch tổ hợp phải được đưa vào danh sách sensitive list, nếu bỏ sót tín hiệu đầu vào trong danh sách này thì mặc dù cú pháp lệnh vẫn đúng nhưng chắc chắn mạch sẽ hoạt động sai về chức năng. Các lệnh bên trong khối process sẽ mô tả sự phụ thuộc logic của đầu ra với các đầu vào.

```
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity mux2 is
port(A   : in  std_logic;
     B   : in  std_logic;
     sel : in  std_logic;
     S   : out std_logic);
end mux2;
----------------------------------------
architecture behavioral of mux2 is
begin
mux: process (A, B, sel)
     begin
        CASE sel IS
            WHEN '0'    => S <= A;
            WHEN others => S <= B;
        end case;
     end process mux;

end behavioral;
----------------------------------------
```

Ở ví dụ trên mô tả một khối chọn kênh (MUX) dùng lệnh case, các đầu vào của gồm A, B và tín hiệu chọn Sel, đầu ra là S, S sẽ chọn bằng A nếu sel = 0 và bằn B nếu sel = 1.

7.3. Phát biểu gán tín hiệu đồng thời

Phát biểu gán tín hiệu đồng thời

```
[label :] [postponed ] conditional_signal_assignment
|[label:] [postponed ] selected_signal_assignment
options ::= [ guarded ] [ delay_mechanism ]
```

Trong đó :

- `postponed`: nếu được sử dụng thì lệnh gán được trì hoãn, nếu không đó là một lệnh gán thông thường. Lệnh gán trì hoãn được thực hiện chỉ sau khi tất cả các lệnh không trì hoãn (`non-postponded`) đã bắt đầu và kết thúc.
- `options`: trong mục này nếu dữ liệu được khai báo tùy chọn bảo vệ (guarded) thì sẽ thực hiện dưới sự điều khiển của tín hiệu ẩn GUARD như đã trình bày ở trên. `delay_mechanism` là phương thức làm việc theo trễ của tín hiệu, có hai dạng là **`transport`** và **`inertial`** như đã trình bày ở 6.4.

Phát biểu gán tín hiệu đồng thời thường sử dụng trong các kiến trúc dạng dataflow. Các lệnh gán này cũng có thể sử dụng các cú pháp có điều kiện kiện sử dụng WHEN hoặc sử dụng tổ hợp WITH/SELECT/WHEN.

7.3.1 Gán tín hiệu dùng WHEN

Cú pháp tổng quát như sau:

```
target <= waveform1 when condition1 else
    waveform2 when condition2 else
    •
    •
    •
    waveformN-1 when conditionN-1 else
    waveformN when conditionN else
    default_waveform;
```

Trong đó default_value có thể chọn giá trị UNAFFECTED. Tương đương với đoạn mã trên có thể sử dụng process và lệnh tuần tự if như sau:

```
process(sel1, waveform)
begin
if condition1 then
    target <= waveform1
elsif condition2 then
    target <= waveform2
    •
    •
   •
elsif conditionN-1 then
    target <= waveformN-1
elsif conditionN then
    target <= waveformN
end if ;
end process;
```

7.3.2. Gán tín hiệu dùng WITH/SELECT/WHEN

Cú pháp tổng quát như sau:

```
WITH expression SELECT
target <= options waveform1 WHEN choice_list1 ,
    waveform2 WHEN choice_list2 ,
    •
    •
    •
    waveformN-1 WHEN choice_listN-1,
    waveformN WHEN choice_listN
    default_value WHEN OTHERS ;
```

Trong đó `default_value` có thể nhận giá trị `UNAFFECTED` Đoạn mã trên cũng có thể thay thế bằng lệnh case trong một process như sau:

```
CASE expression IS
WHEN choice_list1 =>
    target <= waveform1;
WHEN choice_list2 =>
    target <= waveform2;
    •
    •
    •
WHEN choice_listN-1 =>
    target <= waveformN_1;
WHEN choice_listN =>
    target <= waveformN;
END CASE;
```

Để minh họa cho sử dụng cấu trúc WHEN, và WITH/SELECT/WHEN xét mô tả một bộ chọn kênh như sau, module gồm hai bộ giải mã kênh 4 đầu vào và 1 đầu ra độc lập với nhau, điều khiển bởi các tín hiệu sel1, sel2, bộ giải mã thứ nhất sử dụng cấu trúc WHEN, còn tổ hợp thứ hai sử dụng cấu trúc WITH/SELECT/WHEN.


```
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity concurrent_expample is
port(
    A          : in  std_logic_vector(3 downto 0);
    B          : in  std_logic_vector(3 downto 0);
    sel1, sel2 : in  std_logic_vector(1 downto 0);
    O1, O2     : out std_logic
    );
end concurrent_expample;
----------------------------------------
architecture dataflow of concurrent_expample is
begin
    O1 <= A(0) WHEN sel1 = "00" else
          A(1) WHEN sel1 = "01" else
          A(2) WHEN sel1 = "10" else
          A(3) WHEN sel1 = "11" else
          UNAFFECTED;

    WITH sel2  SELECT
    O2 <= B(0) WHEN "00",
          B(1) WHEN "01",
          B(2) WHEN "10",
          B(3) WHEN "11",
```

```
            UNAFFECTED WHEN others;
 end dataflow;
-----------------------------------------
```

Mã nguồn trên có thể thay thế bằng mã sử dụng các phát biểu tuần tự tương đương như sau:

```
-----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----------------------------------------
entity sequential_expample is
port(
    A           : in std_logic_vector(3 downto 0);
    B           : in std_logic_vector(3 downto 0);
    sel1, sel2  : in std_logic_vector(1 downto 0);
    O1, O2      : out std_logic
    );
end sequential_expample;
-----------------------------------------------
architecture dataflow of sequential_expample is
begin
    decodeA: process (A, sel1)
            begin
            if sel1 = "00" then O1 <= A(0); elsif
               sel1 = "01" then O1 <= A(1); elsif
               sel1 = "10" then O1 <= A(2); elsif
               sel1 = "11" then O1 <= A(3);
            end if;
            end process decodeA;
    decodeB: process (B, sel2)
            begin
               CASE sel1 IS
               WHEN "00" => O2 <= B(0);
               WHEN "01" => O2 <= B(1);
               WHEN "10" => O2 <= B(2);
               WHEN "11" => O2 <= B(3);
               WHEN others => O2 <= 'X';
               END CASE;
            end process decodeB;
end dataflow;
-----------------------------------------------
```

7.4. Phát biểu generate

Phát biểu generate (generate statement) là một cú pháp lệnh song song khác. Nó tương đương với khối lệnh tuần tự LOOP trong việc cho phép các đoạn lệnh được thực hiện lặp lại một số lần nào đó. Mẫu dùng của phát biểu này như sau:

```
for generate_parameter_specification
[ if condition
generate
[ { block_declarative_item }
begin ]
{ concurrent_statement }
end generate [label] ;
```

Lưu ý rằng các tham số dùng cho vòng lặp **for** phải là các tham số tĩnh, nếu tham số dạng động sẽ gây ra lỗi biên dịch. Dưới đây là đoạn mã minh họa mô tả bộ cộng 4 bit, thay vì việc mô tả tường minh tất cả các bít của đầu ra ta dùng vòng lặp FOR/GENERATE để gán các giá trị cho chuỗi nhớ C và tổng O theo sơ đồ đệ quy:

```
-------------------------------------   ------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----------------------------------     --------
entity adder4_gen is
port(
    A   : in  std_logic_vector(3 downto 0);
    B   : in  std_logic_vector(3 downto 0);
    CI  : in  std_logic;
    SUM : out std_logic_vector(3 downto 0);
    CO  : out std_logic
    );
end adder4_gen;
-----------------------------------------------
architecture dataflow of adder4_gen is
signal C: std_logic_vector (4 downto 0);
begin
    C(0) <= CI;
    CO   <= C(4);
    Carry: for i in 1 to 4 generate
        C(i) <= (A(i-1) and B(i-1)) or (C(i-1) and (A(i-
1) or B(i-1)));
    end generate Carry;
    Suma: FOR i IN 0 to 3 GENERATE
        SUM(i) <=  A(i) xor B(i) xor C(i);
    END GENERATE Suma;
end dataflow;
-----------------------------------------------
```

Câu lệnh GENERATE còn có thể được sử dụng dưới dạng cấu trúc IF/GENERATE . Ví dụ dưới đây minh họa cho cách sử dụng đó. Module gồm hai đầu vào 4 bit và một đầu ra 2 bít, tại các vị trí bít chẵn của a, b thì đầu ra bằng phép AND của các tín hiệu đầu vào:

```
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity generate_expample2 is
port(
    A : in  std_logic_vector(3 downto 0);
    B : in  std_logic_vector(3 downto 0);
    O : out std_logic_vector(1 downto 0)
    );
end generate_expample2;
----------------------------------------
architecture dataflow of generate_expample2 is
begin

    msk: FOR i IN 0 to 3 GENERATE
         ifgen: IF i rem 2 = 0  GENERATE
            O(i/2) <= A(i) and B(i);
         END GENERATE ifgen;
    END GENERATE msk;
end dataflow;
```

Như thấy ở các ví dụ trên lệnh GENERATE thường được sử dụng để cài đặt các khối xử lý logic giống nhau về mặt cấu trúc, số lượng các khối này có thể rất nhiều nên cú pháp generate cho phép viết mã ngắn gọn. Ngoài ra trong một số trường hợp số lượng các khối cài đặt phụ thuộc vào các tham số tĩnh mà không phải hằng số thì việc cài đặt bắt buộc phải thực hiện bằng lệnh này.

7.5. Phát biểu cài đặt module con

Phát biểu cài đặt module con (*component installation statement*) sử dụng cho mô tả dạng cấu trúc của thiết kế khi module tổng được cấu tạo từ nhiều những module nhỏ (xem ví dụ thêm ở 2.2.1).

Cú pháp tổng quát như sau:

```
component component_name is
     generic (generic_variable_declarations );
     port (input_and_output_variable_declarations);
end component component_name ;
```

Ở đoạn mã trên ta đã khai báo sử dụng các cổng AND2, OR2 và XOR2. Sau khi được khai báo component thì các cổng này có thể được sử dụng nhiều lần với cú pháp đầy đủ như sau:

```
identifier : component_name
     generic map( generic_variables => generic values)
     port map( input_and_output_variables => signals);
```

Trong đó

-identifier là tên của module sẽ sử dụng cho component có tên là component_name

-danh sách các biến **generic** và các cổng được gán tường minh bằng toán tử "=>", khi đó thứ tự gán các biến hoặc cổng không quan trọng. Cách thứ hai là gán các cổng không tường minh, khi đó thứ tự các cổng phảiđúng như thứ tự khai báo trong component. Tuy vậy không khuyến khích viết như vậy vì khi đó chỉ cần nhầm lẫn về thứ tự có thể làm sai chức năng của thiết kế. Để minh họa rõ hơn quay lại ví dụ về bộ cộng nhiều bit ở trên:

```
entity adder is
     generic ( N : natural := 32 ) ;
     port ( A    : in  std_logic_vector(N-1 downto 0);
            B    : in  std_logic_vector(N-1 downto 0);
            cin  : in  std_logic;
            Sum  : out std_logic_vector(N-1 downto 0);
            Cout : out std_logic );
end entity adder ;
```

Khai báo component tương ứng sẽ là

```
component adder is
     generic (N : natural := 32) ;
     port ( A    : in  std_logic_vector(N-1 downto 0);
            B    : in  std_logic_vector(N-1 downto 0);
            cin  : in  std_logic;
            Sum  : out std_logic_vector(N-1 downto 0);
            Cout : out std_logic );
end component adder ;
-------------------------------------------------
```

Khai báo sử dụng component adder 16 bit dạng tường minh sẽ là:

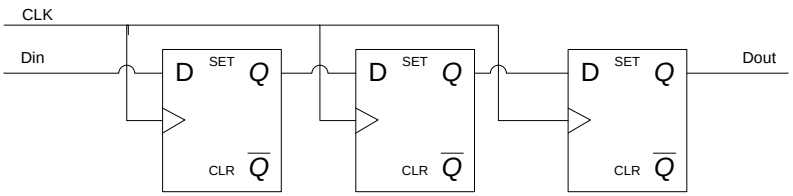
```
U1: adder
    generic map (N => 16)
    port    map (A => in_a, B=> in_b, Cin => in_c
                 Sum => out_s, Cout => out_C
                );
```

Khai báo sử dụng component adder 16 bit dạng không tường minh sẽ là:

```
U1: adder
    generic map (16)
    port    map (
                 in_a, in_b, in_c
                 out_s, out_C
                 );
```

Xét một ví dụ đầy đủ về cài đặt module dưới đây, ở phần 6.4 khi nghiên cứu về phát biểu gán tín hiệu tuần tự ta đã có mô tả VHDL của `D_flipflop`, ví dụ sau dùng phát biểu cài đặt component để mô tả một thanh ghi dịch gồm 3 `D-flipflop`.

Sơ đồ logic của shift_reg:



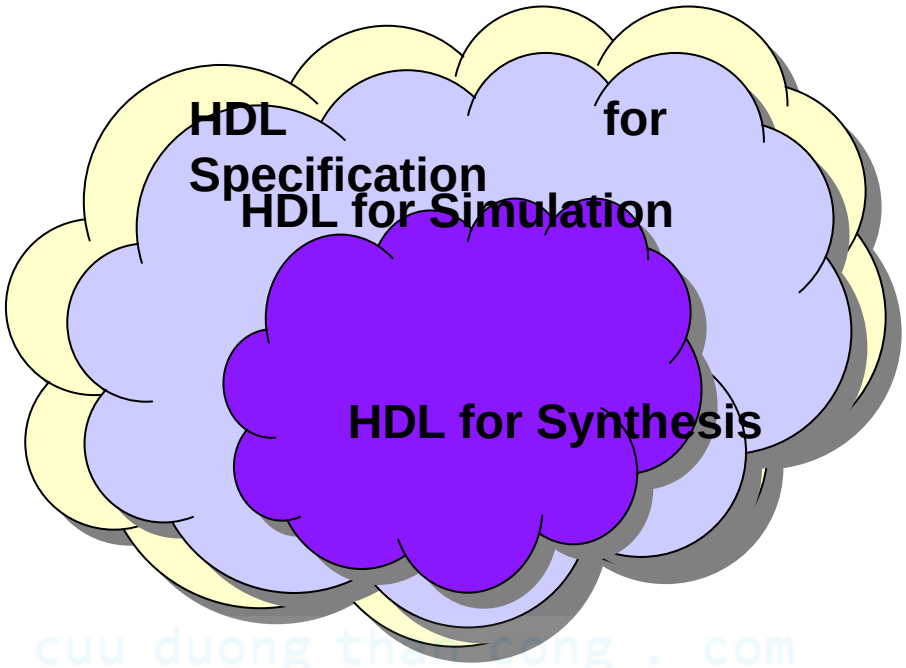
Hình 2.4: sơ đồ khối của thanh ghi dịch

Mô tả VHDL:

```
-----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-----------------------------------------
entity shift_reg is
port(
    Din : in std_logic;
    CLK : in std_logic;
    Dout : out std_logic
    );
end shift_reg;
-----------------------------------------
architecture structure of shift_reg is

signal Q1, Q2 : std_logic;

component D_flipflop  port(
    D : in std_logic;
    CLK : in std_logic;
    Q : out std_logic
    );
end component;
begin

DFF1: D_flipflop port map (D => Din, CLK => CLK, Q =>
Q1);
DFF2: D_flipflop port map (D => Q1, CLK => CLK, Q => Q2);
DFF3: D_flipflop port map (D => Q2, CLK => CLK, Q =>
Dout);
end structure;
-----------------------------------------
```

Ở ví dụ trên các 3 flipflop D được đặt nối tiếp và ta sẽ cài đặt cổng sao cho đầu ra của thanh ghi trước sẽ là đầu vào của thanh ghi sau và sau mỗi xung nhịp chuỗi bit Q1, Q2, Q3 sẽ dịch sang bên phải một bit.

8. Phân loại mã nguồn VHDL



Hình 2.5: Các dạng mã nguồn VHDL

Ngôn ngữ VHDL được xem là một ngôn ngữ chặt chẽ và phức tạp, VHDL hỗ trợ việc mô tả thiết kế từ mức cao cho đến mức thấp và trên thực tế không thể xếp ngôn ngữ này thuộc nhóm bậc cao, bậc thấp hay bậc chung như các ngôn ngữ lập trình khác.

Về phân loại mã nguồn VHDL có thể chia làm ba dạng chính như sau:

- Mã nguồn chỉ dành cho tổng hợp (HDL for Synthesis): Là những mã nguồn nhắm tới mô tả thực của cấu trúc mạch. Ngoài việc tuân thủ chặt chẽ các cấu trúc của ngôn ngữ thì mã nguồn dạng này cần phải tuân thủ những tính chất, đặc điểm vật lý của một mạch tích hợp nhằm đảm bảo mã nguồn có thể được biên dịch trên một công nghệ phần cứng cụ thể nào đó.
- Mã nguồn mô phỏng được (HDL for Simulation): Bao gồm toàn bộ mã tổng hợp được và những mã mà chỉ chương trình mô phỏng có thể biên dịch và thể hiện trên môi trường phần mềm, ví dụ các mã sử

dụng các lệnh tuần tự dùng để gán tín hiệu theo thời gian, các vòng lặp cố định.

- Mã nguồn dành cho mô tả đặc tính (HDL for Specification): Bao gồm toàn bộ mã mô phỏng được được và những cấu trúc dùng để mô tả các đặc tính khác như độ trễ (delay time), điện dung (capacitance)… thường gặp trong các mô tả thư viện cổng. Trong khuôn khổ của chương trình này ta không tìm hiểu sâu về dạng mô tả này.

Một số dạng mã nguồn không tổng hợp được mà chỉ dành cho mô phỏng được liệt kê ở dưới dây:

- Các mã mô tả độ trễ thời gian của tín hiệu, trên thực tế độ trễ các mạch do tính chất vật lý của phần cứng quy định, mã nguồn VHDL độc lập với phần cứng nên các mã quy định độ trễ đều không tổng hợp được mà chỉ dành cho mô phỏng:

```
wait for 5 ns;
A <= B after 3 ns;
```

- Các mã cài đặt giá trị, ban đầu

```
signal a:std_logic_vector (3 downto 0) := "0001";
signal n: BIT := '0';
```

- Mô tả flip-flop làm việc ở cả hai sườn xung nhịp, trên thực tế các Flip-flop chỉ làm việc ở một sườn âm hoặc sườn dương, đoạn mã sau không tổng hợp được thành mạch.

```
PROCESS ( Clk )
    BEGIN
    IF rising_edge(Clk) or falling_edge(CLk) THEN
        Q <= D ;
    END IF ;
END PROCESS ;
```

- Các mã làm việc với kiểu số thực, các trình tổng hợp hiện tại mới chỉ hỗ trợ các mạch tổng hợp với số nguyên, các code mô tả sau không thể tổng hợp được:

```
signal a,b, c: real
begin
    C <= a + b;
end architechture;
```

- Các lệnh báo cáo và theo dõi (`report, assert`), các lệnh này phục vụ quá trình mô phỏng kiểm tra mạch, không thể tổng hợp được:

```
assert a='1' report "it OK" severity NOTE;
report  "finished pass1";
```

9. Kiểm tra thiết kế bằng VHDL.

Một trong những phần công việc khó khăn và chiếm nhiều thời gian trong quá trình thiết kế vi mạch là phần kiểm tra thiết kế. Trên thực tế đã có rất nhiều phương pháp, công cụ khác nhau ra đời nhằm đơn giản hóa và tăng độ tin cậy quá trình kiểm tra. Một trong những phương pháp phổ biến và dễ dùng là phương pháp viết module kiểm tra trên chính HDL, bên cạnh đó một số hướng phát triển hiện nay là thực hiện các module kiểm tra trên các ngôn ngữ bậc cao như System C, System Verilog, các ngôn ngữ này hỗ trợ nhiều hàm bậc cao cũng như các hàm hệ thống khác nhau.

Trong khuôn khổ của chương trình chúng ta sẽ nghiên cứu sử dụng phương pháp thứ nhất là sử dụng mô tả VHDL cho module kiểm tra. Lưu ý rằng mô hình kiểm tra ở dưới đây là không bắt buộc phải theo, người thiết kế sau khi đã nắm được phương pháp có thể viết các mô hình kiểm tra khác nhau tùy theo mục đích và yêu cầu.

Chúng ta bắt đầu với việc kiểm tra module khối cộng 4 bit được viết bằng câu lệnh generate đã có ở phần 7.4:

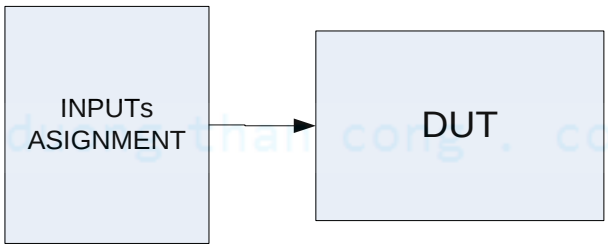
```
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity adder4_gen is
port(
    A   : in std_logic_vector(3 downto 0);
    B   : in std_logic_vector(3 downto 0);
    CI  : in std_logic;
    SUM : out std_logic_vector(3 downto 0);
    CO  : out std_logic
    );
end adder4_gen;
----------------------------------------------
architecture dataflow of adder4_gen is
signal C: std_logic_vector (4 downto 0);
begin
    C(0) <= CI;
    CO <= C(4);
```

```
        Carry: for i in 1 to 4 generate
            C(i) <= (A(i-1) and B(i-1)) or (C(i-1) and (A(i-
1) or B(i-1)));
        end generate Carry;
        --SUM(0) <= A(0) xor B(0) xor CI;
        Suma: FOR i IN 0 to 3 GENERATE
             SUM(i) <=  A(i) xor B(i) xor C(i);
        END GENERATE Suma;
    end dataflow;
    ----------------------------------------------
```

Nhiệm vụ của chúng ta là cần kiểm tra xem với mô tả như trên thì bộ cộng có làm việc đúng chức năng không. Quy trình thiết kế chia làm hai bước, bước thứ nhất sẽ tiến hành kiểm tra nhanh với một vài giá trị đầu vào, bước thứ hai là kiểm tra toàn bộ thiết kế:

9.1.1. Kiểm tra nhanh

Sơ đồ kiểm tra nhanh như sau:



Hình 2.6 Kiểm tra nhanh thiết kế.

Ở sơ đồ trên DUT là viết tắt của *Device Under Test* nghĩa là đối tượng bị kiểm tra, trong trường hợp này là bộ cộng 4 bit, khối Input generator sẽ tạo ra một hoặc một vài tổ hợp đầu vào để gán cho các cổng input của DUT.

Để hiểu kỹ hơn ta phân tích mã nguồn của module kiểm tra như sau:

```
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
----------------------------------------
entity test_adder4_gen is
end test_adder4_gen;
----------------------------------------
architecture testbench of test_adder4_gen is

component adder4_gen is
port(
    A    : in std_logic_vector(3 downto 0);
    B    : in std_logic_vector(3 downto 0);
```

```
    CI   : in std_logic;
    SUM  : out std_logic_vector(3 downto 0);
    CO   : out std_logic
    );
end component;
-- khai bao cac tin hieu vao ra cho DUT
signal  A  : std_logic_vector(3 downto 0) := "0101";
signal  B  : std_logic_vector(3 downto 0) := "1010";
signal  CI : std_logic                    := '1';
-- output---
signal  SUM  : std_logic_vector(3 downto 0);
signal  CO : std_logic;

begin
   DUT: component adder4_gen
        port map (
                  A => A, B=> B, CI => CI,
                  SUM => SUM, CO =>CO
                  );
end testbench;
-------------------------------------------------
```

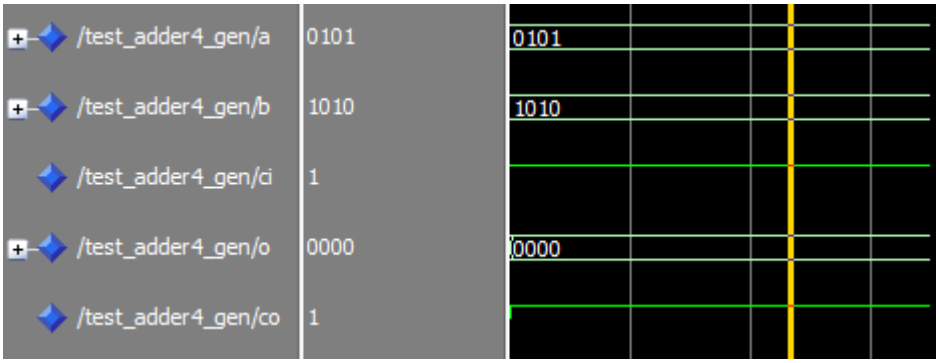
Module kiểm tra thường là một module mà không có cổng vào hoặc ra giống như ở trên. Module kiểm tra sẽ khai báo DUT như một module con do vậy component adder4_gen được khai báo trong phần khai báo của kiến trúc.

Bước tiếp theo căn cứ vào các cổng vào ra của DUT sẽ tiến hành khai báo các tín hiệu tương ứng, để tránh nhầm lẫn cho phép dùng tên các tín hiệu này trùng với tên các tín hiệu vào ra của DUT.

Các tín hiệu tương ứng với các chân input được gán giá trị khởi tạo là các hằng số hợp lệ bất kỳ, việc gán này tương đương với ta sẽ gửi tới DUT các giá trị đầu vào xác định.

Các tín hiệu tương ứng với chân output sẽ để trống và không được phép gán giá trị nào cả.

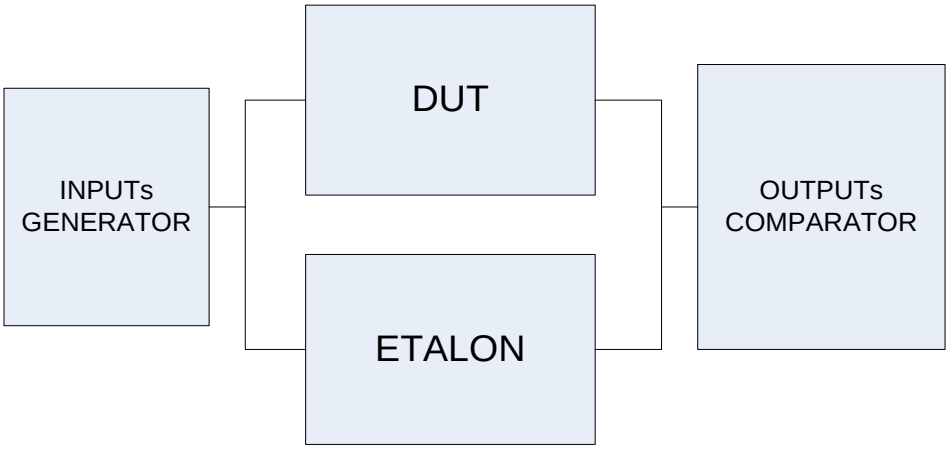
Bước cuối cùng là cài đặt DUT là một module adder4_gen với các chân vào ra được gán tương ứng với các tín hiệu khai báo ở trên. Khi chạy mô phỏng ta thu được kết quả như sau:



9.1.1. Kiểm tra tự động nhiều tổ hợp đầu vào

Việc kiểm tra nhanh cho phép phát hiện những lỗi đơn giản về mặt chức năng và không thể dựa vào kết quả của bước kiểm tra này để kết luận khối của chúng ta làm việc đúng hay chưa. Về mặt lý thuyết, khối thiết kế được coi là làm việc đúng nếu như nó cho ra kết quả đúng với mọi tổ hợp đầu vào. ví dụ trên ta có 3 tính hiệu đầu vào là a(3:0), b(3:0), CI, vì vậy tổ hợp tất cả các tính hiệu đầu vào là $2^4 * 2 *^4 *2^1 = 2^9 = 512$ tổ hợp. Rõ ràng đối với khả năng của máy tính hiện đại đây là một con số rất nhỏ. Như vậy module của adder4 có thể được kiểm tra với độ tin cậy lên tới 100%.

Sơ đồ của kiểm tra tự động như sau:



Hình 16: Mô hình chung của khối kiểm tra thiết kế tự động

Trong đó :

- DUT (device under test) đối tượng kiểm tra, ví dụ như trong trường hợp của chúng ta là adder4

- ETALON: là một thiết kế chuẩn thực hiện chức năng giống như DUT nhưng có thể sử dụng các cấu trúc bậc cao dạng không tổng hợp được (simulation only), sử dụng thuật toán đơn giản nhất có thể, etalon thường được mô tả để làm việc như một function để tránh các cấu trúc phụ thuộc thời gian hay phát sinh ra lỗi. Nhiệm vụ của Etalon là thực hiện chức năng thiết kế một cách đơn giản và chính xác nhất để làm cơ sở so sánh với kết quả của DUT.
- INPUTs GENERATOR: khối tạo đầu vào, khối này sẽ tạo các tổ hợp đầu vào khác nhau và đồng thời gửi đến hai khối là DUT và ETALON.
- OUTPUTs COMPARATORs: Khối này sẽ so sánh kết quả đầu ra tại những thời điểm nhất định và đưa ra tín hiệu cho biết là hai kết quả này có như nhau không.

Để có thể gán nhiều đầu vào tại các điểm khác nhau trong module kiểm tra phải tạo ra một xung nhịp đồng hồ dạng như CLK, các tổ hợp giá trị đầu vào mới sẽ được gán tại các thời điểm nhất định phụ thuộc CLK, thường là vào vị trí sườn âm hoặc sườn dương của xung nhịp.

Quay trở lại với mudule cộng 4 bit adder4_gen ở trên, với module này ta có thể thực hiện kiểm tra cho tất cả các tổ hợp đầu vào (512 tổ hợp), trước hết ta viết một module etalon cho bộ cộng 4 bit như sau:

```
----------- adder 4-bit etalon ----------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-----------------------------------------
entity adder4_etalon is
port(
    A   : in std_logic_vector(3 downto 0);
    B   : in std_logic_vector(3 downto 0);
    CI  : in std_logic;
    SUM : out std_logic_vector(3 downto 0);
    CO  : out std_logic);
end adder4_etalon;
-----------------------------------------
architecture behavioral of adder4_etalon is
signal s_sig: std_logic_vector(4 downto 0);
signal a_sig: std_logic_vector(4 downto 0);
signal b_sig: std_logic_vector(4 downto 0);
begin
    a_sig <= '0' & A;
```

```
    b_sig <= '0' & B;
    plus: process (a_sig, b_sig, CI)
    begin
        s_sig <= a_sig + b_sig + CI;
    end process plus;
    SUM  <= s_sig (3 downto 0);
    CO   <= s_sig (4);
end behavioral;
----------------------------------------
```

Module adder4_etalon thực hiện phép cộng 4 bit bằng lệnh + của VHDL, kết quả phép cộng này là tin cậy 100% nên có thể sử dụng để kiểm tra thiết kế adder4_gen ở trên của chúng ta. Để thực hiện các thao tác kiểm tra tự động, sử dụng một module kiểm tra có nội dung như sau:

```
----------adder 4 testbench_full -------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
library STD;
use STD.TEXTIO.ALL;
----------------------------------------
entity adder4_testbench is
end adder4_testbench;
----------------------------------------
architecture testbenchfull of adder4_testbench is
signal a_t   : std_logic_vector(3 downto 0) := "0000";
signal b_t   : std_logic_vector(3 downto 0) := "0000";
signal sum_t : std_logic_vector(3 downto 0);
signal sum_e : std_logic_vector(3 downto 0);
signal ci_t  : std_logic                    := '0';
signal co_t  : std_logic;
signal co_e  : std_logic;
signal clk   : std_logic                    := '0';

component adder4_gen
    port (A   : in  std_logic_vector (3 downto 0);
          B   : in  std_logic_vector (3 downto 0);
          CI  : in  std_logic;
          SUM : out std_logic_vector (3 downto 0);
          CO  : out std_logic
         );
end component;

component adder4_etalon
    port (A   : in  std_logic_vector (3 downto 0);
          B   : in  std_logic_vector (3 downto 0);
          CI  : in  std_logic;
          SUM : out std_logic_vector (3 downto 0);
```

```
          CO  : out std_logic
         );
    end component;
BEGIN
     --create clock
    create_clock: process
    begin
        wait for 15 ns;
        CLK <= not CLK after 50 ns;
    end process create_clock;

    check:
    process (clk)
    variable info: line;
    variable test_cnt: integer := 0;
    begin
    if clk = '1' and clk'event then
        write(info, string'("Test # "));
        write(info, integer'(test_cnt + 1));
        write(info, string'(" a = "));
        write(info, integer'(conv_integer(a_t)));
        write(info, string'(" b =  "));
        write(info, integer'(conv_integer(b_t)));
        write(info, string'(" CI =  "));
        write(info, integer'(conv_integer(ci_t)));
        write(info, string'(" sum =  "));
        write(info, integer'(conv_integer(sum_t)));
        write(info, string'(" CO =  "));
        write(info, integer'(conv_integer(co_t)));
        write(info, string'(" sum_e =  "));
        write(info, integer'(conv_integer(sum_e)));
        write(info, string'(" CO_e =  "));
        write(info, integer'(conv_integer(co_e)));
        if sum_e /= sum_t or co_e /= co_t then
            write(info, string'("FAILURE"));
        else
            write(info, string'(" OK"));
        end if;
        writeline(output, info);
        -- input data generator.
        test_cnt := test_cnt + 1;
        ci_t <= not ci_t;
        if ci_t = '1' then
        a_t <= a_t +1;
        end if;
        if a_t = "1111" then
            b_t <= b_t + 1;
        end if;
        assert test_cnt < 512
        report "end simulation" severity NOTE;
    end if;
```

```
    end process check;
        -- component installation
    dut: adder4_gen
         port map (
                   A => a_t, B => b_t, CI => ci_t,
                   SUM =>sum_t, CO => co_t);
    etalon: adder4_etalon
         port map (A => a_t, B => b_t, CI => ci_t,
                   SUM =>sum_e, CO => co_e);
END testbenchfull;
----------------------------------------
```

Module kiểm tra cài đặt đồng thời các module con là adder4_gen (DUT) và adder4_etalon (ETALON), các module này có các đầu vào dữ liệu y hệt như nhau lấy từ các tín hiệu tương ứng là a_t, b_t, ci_t. Các đầu ra của hai module này tương ứng là sum_t, co_t cho DUT và sum_e, co_e cho ETALON.

Module này thực chất lặp lại chức năng của module kiểm tra nhanh nhiều lần với nhiều tổ hợp đầu vào, mặt khác sử dụng kết quả tính toán từ module chuẩn ETALON để kiểm tra tự động tính đúng đắn của kết quả đầu ra từ module DUT cần kiểm tra

Để làm được việc đó, một xung nhịp clk được tạo ra, chu kỳ của xung nhịp này có thể nhận giá trị bất kỳ, ví dụ ở đoạn mã trên ta tạo xung nhịp clk có chu kỳ T = 2 x 50 ns = 100 ns. Có xung nhịp này ta sẽ tạo một bộ đếm theo xung đếm là clk để đếm số lượng test, khi đếm đủ 512 test thì sẽ thông báo ra màn hình việc thực hiện test đã xong.

Tại mỗi thời điểm sườn dương của clk giá trị đếm này tăng thêm 1 đồng thời tại thời điểm đó tổ hợp giá trị đầu vào thay đôi tuần tự để quét hết 512 tổ hợp giá trị khác nhau. Đầu tiên các giá trị a_t, b_t, ci_t nhận giá trị khởi tạo là 0, 0, 0. Tại các sườn dương của xung nhịp thay đổi giá trị ci_t trước, nếu ci_t = 1 thì tăng a_t thêm 1 đơn vị, sau đó kiểm tra nếu a_t = “1111” thì sẽ tăng b_t thêm 1 đơn vị.

Các đầu ra sum_t, sum_e, co_t, co_e sẽ được so sánh với nhau sau mỗi xung nhịp, nếu như sum_t = sum_e và co_t = co_t thì kết luận là adder4_gen làm việc đúng (TEST OK) và ngược lại là module làm việc sai (TEST FAILURE)

Ví số lượng tổ hợp đầu vào là lớn nên không thể quan sát bằng waveform nứa, kết quả khi đó được thông báo trực tiếp ra màn hình. Khi chạy mô phỏng ta thu được thông báo như sau:

```
      # Test # 1 a = 0 b = 0 CI = 0 sum =  0 CO = 0  sum_e = 0
CO_e = 0 OK
      # Test # 2 a = 0 b = 0 CI = 1 sum =  1 CO = 0  sum_e = 1
CO_e = 0 OK
      # Test # 3 a = 1 b = 0 CI = 0 sum =  1 CO =  0 sum_e = 1
CO_e = 0 OK
      # Test # 4 a = 1 b = 0 CI = 1 sum =  2 CO =  0 sum_e = 2
CO_e = 0 OK
      # Test # 5 a = 2 b = 0 CI = 0 sum =  2 CO =  0 sum_e = 2
CO_e = 0 OK
      # Test # 6 a = 2 b = 0 CI = 1 sum =  3 CO =  0 sum_e = 3
CO_e = 0 OK
      # Test # 7 a = 3 b = 0 CI = 0 sum =  3 CO =  0 sum_e = 3
CO_e = 0 OK
      # Test # 8 a = 3 b = 0 CI = 1 sum =  4 CO =  0 sum_e = 4
CO_e = 0 OK
      # Test # 9 a = 4 b = 0 CI = 0 sum =  4 CO =  0 sum_e = 4
CO_e = 0 OK
      # Test # 10 a = 4 b = 0 CI = 1 sum =  5 CO = 0 sum_e = 5
CO_e = 0 OK
      # Test # 11 a = 5 b = 0 CI = 0 sum =  5 CO = 0 sum_e = 5
CO_e = 0 OK
      …
      …
      # Test #511a = 15 b = 14 CI = 0 sum = 13 CO = 1 sum_e =13
CO_e = 1OK
      # Test #512a =15 b = 15 CI = 1 sum= 15 CO = 1 sum_e = 15
CO_e = 1 OK
      # ** Note: end simulation
      #    Time: 61385 ns  Iteration: 0  Instance:
/adder4_testbench
```

Nếu như tất cả các trường hợp đều thông báo là OK thì module DUT có thể kết luận module DUT làm việc đúng, quá trình kiểm tra hoàn tất.

Lưu ý rằng khi thực hiện kiểm tra tự động thì module ETALON không phải lúc nào cũng có độ chính xác 100% như ở trên và đôi khi không thể viết được module này. Khi đó ta phải có phương án kiểm tra khác.

Điểm lưu ý thứ hai là trên thực tế việc kiểm tra với mọi tổ hợp đầu vào (full_testbench) thường là không thể vì số lượng các tổ hợp này trong đa số các trường hợp rất lớn ví dụ như nếu không phải bộ cộng 4 bit mà là bộ cộng 32 bit thì số lượng tổ hợp đầu vào là $2^{32x2+1} = 2^{65}$ là một con số quá lớn để kiểm tra hết dù bằng các máy tính nhanh nhất. Khi đó quá trình kiểm tra chia thành hai bước:

Bước 1 sẽ tiến hành kiểm tra bắt buộc với các tổ hợp có tính chất riêng như bằng 0 hay bằng số lớn nhất, hoặc các tổ hợp gây có thể gây ra các ngoại lệ, các tổ hợp đã gây phát sinh lỗi.

Nếu kiểm tra với các tổ hợp này không có lỗi sẽ chuyển sang bước thứ hai là RANDOM_TEST. Bộ phân kiểm tra sẽ cho chạy RANDOM_TEST với số lượng đầu vào không giới hạn. Nếu trong quá trình này phát hiện ra lỗi thì tổ hợp giá trị gây ra lỗi sẽ được bổ xung vào danh sách các tổ hợp bắt buộc phải kiểm tra ở bước 1 và làm lại các bước kiểm tra từ đầu sau khi sửa lỗi. Kiểm tra được coi là thực hiện xong nếu như với một số lượng rất lớn RANDOM_TEST mà không tìm thấy lỗi.

Bài tập chương II

1. Bài tập cơ sở.

1. Thiết kế các cổng logic cơ bản AND, OR, NOT, XOR sử dụng tất cả các dạng kiến trúc khác nhau (dataflow, structure, behavioral).

2. Thiết kế chuỗi nhớ trước 4-bit cho bộ cộng, sử dụng ghép nối chuỗi nhớ 1 bit hoặc dùng lệnh generate, so sánh các cách mô tả khác nhau đó.

3. Thiết kế bộ giải mã 3_to_8 có đầu ra thuận, nghịch.

4. Thiết bộ chọn kênh 8 đầu vào 1 đầu ra MUX8_1.

5. Thiết bộ phân kênh 1 đầu vào 8 đầu ra DEMUX1_8.

6. Thiết kế bộ so sánh 4 bit có dấu và không dấu.

7. Thiết kế các bộ chuyển đổi mã từ

BINARY – BCD, BCD – BINARY,

BCD – GRAY, GRAY – BCD.

BCD – 7SEG, 7SEG – BCD.

7SEG– GRAY, GRAY-7SEG

8. Thiết kế các flip-flop không đồng bộ RS, D, T, JK.

9. Thiết kế các flip-flop đồng bộ RS, D, T, JK.

10. Thiết kế bộ đếm nhị phân dùng JK Flip-flop

11. Thiết kế thanh ghi dịch trái qua phải 16-bit, bit dịch là một số nguyên từ 1-15 sử dụng toán tử dịch.

12. Thiết kế bộ đếm thuận, nghịch, hỗn hợp với Kd = 8 không đồng bộ.

13. Thiết kế bộ đếm thuận, nghịch, hỗn hợp Kd = 8 đồng bộ, RESET không đồng bộ và hỗ trợ tín hiệu Enable.

14. Thiết kế bộ đếm từ 3 đến 10 đồng bộ và không đồng bộ, RESET không đồng bộ và hỗ trợ tín hiệu Enable.

15. Thiết kế bộ đếm từ Kd = 16 có bước đếm nhận các giá trị 1, 2, 4, RESET không đồng bộ và hỗ trợ tín hiệu Enable.

16. Thiết kế bộ đếm thập phân đồng bộ và không đồng bộ, RESET không đồng bộ và hỗ trợ tín hiệu Enable.

17. Sử dụng bộ đếm thiết kế bộ chia tần từ tần số 50Hz thành 1Hz, tần số xung nhịp thu được có dạng đối xứng.

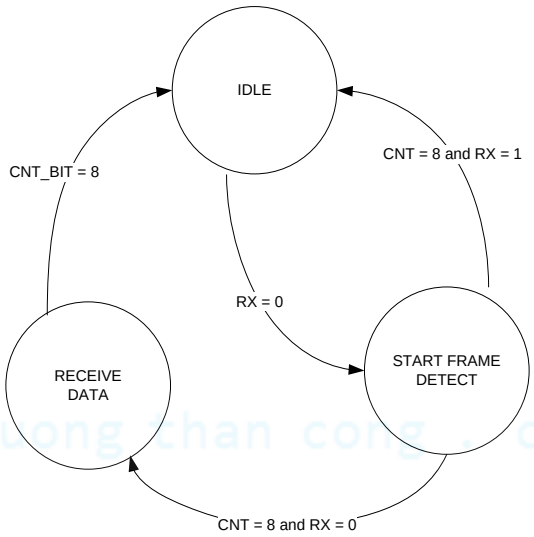
18. Sử dụng VHDL thiết kế và kiểm tra hoạt động các IC họ 74xx sau: 74ls194, 74190, 7447, 7448, 74ls151, 74ls352, 74LS138.

19. Hiện thực sơ đồ mã CRC nối tiếp và song song bằng VHDL.

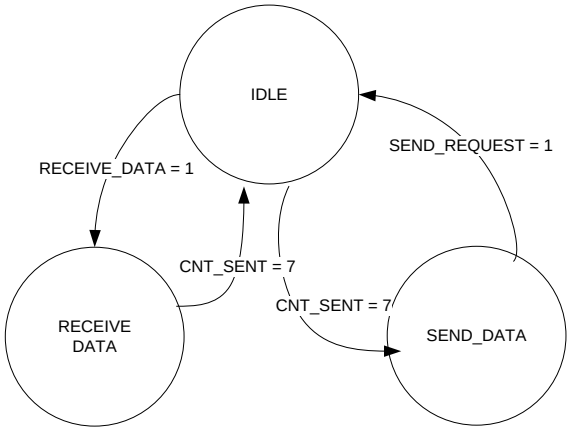
20. Thiết kế khối giải mã ưu tiên, đầu vào là chuỗi 8 bit đầu ra là mã nhị phân 3 bit thể hiện vị trí đầu tiên từ trái qua phải xuất hiện bit '1'.

21. Thiết kế khối giải mã ưu tiên, đầu vào là chuỗi 8 bit đầu ra là mã nhị phân 3 bit thể hiện vị trí đầu tiên từ trái qua phải xuất hiện bit '0'.

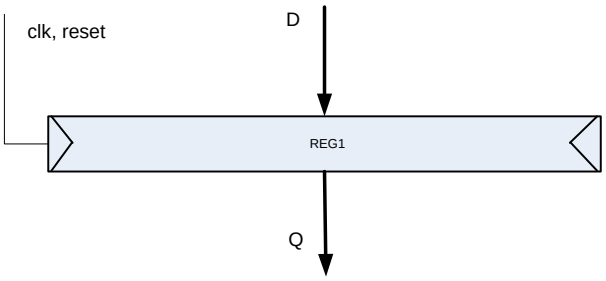
22. Viết mô tả VHDL cho máy trạng thái có sơ đồ sau:



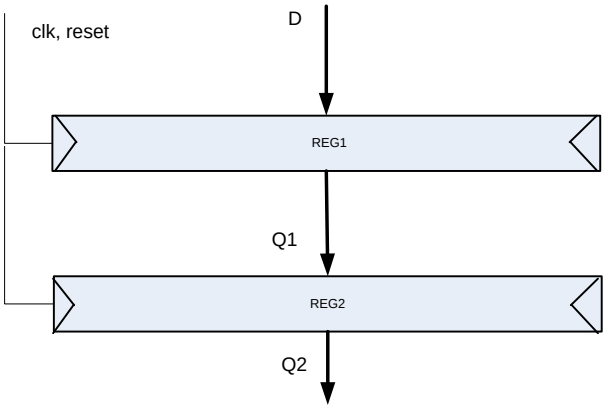
23. Viết mô tả VHDL cho máy trạng thái có sơ đồ sau:



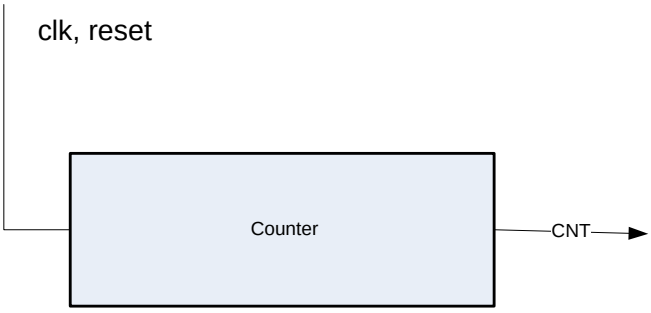
24. Phân tích hoạt động và vẽ giản đồ sóng cho các tín hiệu thanh ghi:



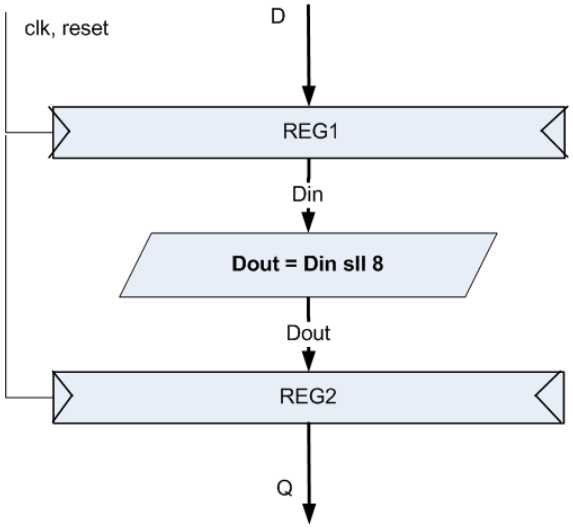
25. Phân tích hoạt động và vẽ giản đồ sóng cho các tín hiệu của các thanh ghi:



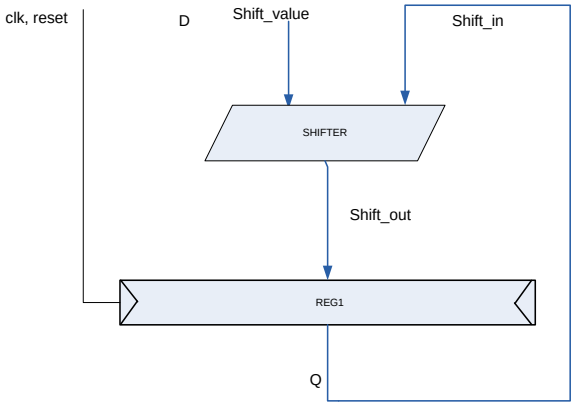
26. Phân tích hoạt động và vẽ giản đồ sóng cho các tín hiệu bộ đếm:



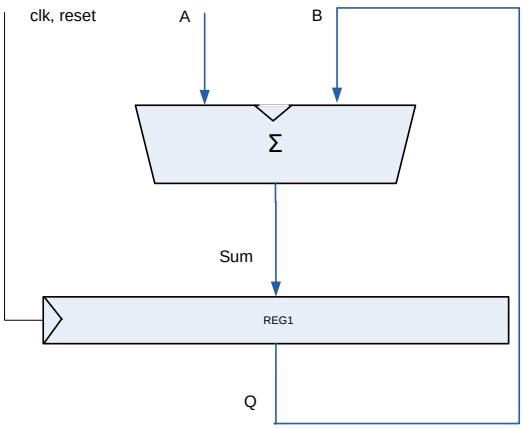
27. Phân tích hoạt động và vẽ giản đồ sóng cho các tín hiệu thanh ghi, giá trị ban đầu trong thanh ghi là Q(31:0) = [A B, C, D] trong đó A, B, C, D lầ các chuỗi 8-bit. Viết mã VHDL



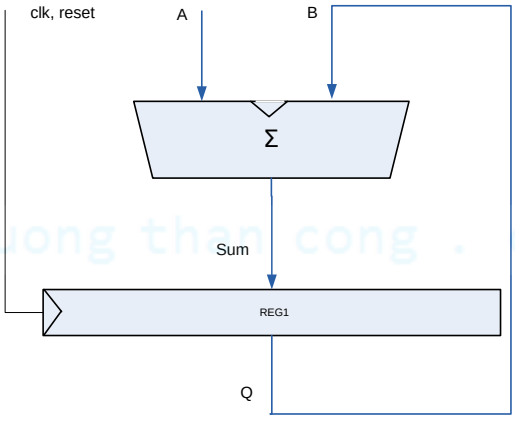
28. Phân tích hoạt động và vẽ giản đồ sóng của thanh ghi dịch như sau, lấy ví dụ bộ dịch phải logic 8 bit, giá trị ban đầu trong thanh ghi là Q(31:0) = [A B, C, D] trong đó A, B, C, D lầ các chuỗi 8-bit. Viết mã VHDL.



29. Phân tích hoạt động và vẽ giản đồ sóng của bộ cộng tích lũy như sau , giá trị ban đầu trong thanh ghi là Q(31:0) = [0], A = 15. Viết mã VHDL



28. Phân tích hoạt động và vẽ giản đồ sóng của bộ cộng tích lũy như sau:



2. Bài tập nâng cao

1. Thiết kế mô phỏng bộ cộng 8-bit thấy nhớ trước, yêu cầu kiểm tra toàn bộ (full_test).

2. Thiết kế mô phỏng bộ cộng/trừ 8-bit thấy nhớ trước, yêu cầu kiểm tra toàn bộ (full_test).

3. Thiết kế thanh ghi dịch trái qua phải 32-bit, bit dịch là một số nguyên từ 1-31 không sử dụng toán tử dịch của VHDL.

4. Thiết kế mô phỏng và kiểm tra bộ dịch trái phải, số học và logic 32-bit. Yêu cầu về mặt giao diện như sau: 8-bit dữ liệu đầu vào (sha), 3-bit quy định giá trị dịch (shv), 8-bit dữ liệu đầu ra (sho), tín hiệu điều khiển hướng dịch (L = '1' nếu dịch trái, L= '0' là dịch phải), tín hiệu điều khiển dạng dịch (ARTH = '1' nếu là dịch số học, ARTH = '0' nếu là dịch logic.

5. Thiết kế mô phỏng và kiểm tra bộ nhân 8-bit không dấu. Yêu cầu về mặt giao diện như sau: 8-bit số nhân (multiplier), 8-bit số bị nhân (multiplicand), 16-bit dữ liệu kết quả đầu ra (mul_out)

6. Thiết kế mô phỏng và kiểm tra bộ nhân 8-bit có dấu. Yêu cầu về mặt giao diện như sau: 8-bit số nhân (multiplier), 8-bit số bị nhân (multiplicand), 16-bit dữ liệu kết quả đầu ra (mul_out)

7. Thiết kế mô phỏng và kiểm tra bộ chia 8-bit không dấu. Yêu cầu về mặt giao diện như sau: 8-bit số chia (divisor), 8-bit số bị chia (dividend), 8-bit dữ liệu kết quả (division), 8-bit số dư (remainder).

8. Thiết kế mô phỏng và kiểm tra bộ chia 8-bit có dấu. Yêu cầu về mặt giao diện như sau: 8-bit số chia (divisor), 8-bit số bị chia (dividend), 8-bit dữ liệu kết quả (division), 8-bit số dư (remainder). (bit đầu tiên của mỗi phần tử trên là bit quy định dấu)

9. Thiết kế module RAM một cổng đồng bộ 128 x 8-bit, hỗ trợ các động tác đọc ghi dữ liệu, mỗi tác vụ đọc ghi được thực hiện trong một xung nhịp đồng hồ. Yêu cầu về mặt giao diện như sau, 7-bit địa chỉ ADDR, 8-bit kênh dữ liệu vào (DIN), 8-bit kênh dữ liệu ra (DOUT), tín hiệu điều khiển đọc ghi (WE = 1 ghi đọc dữ liệu, OE = 1 ouput enable cho đọc dữ liệu), tín hiệu CS để chọn chip.

10. Thiết kế module RAM một cổng ghi đồng bộ, đọc không đồng bộ 128 x 8-bit, hỗ trợ các động tác đọc ghi dữ liệu. Yêu cầu về mặt giao diện như sau, 7-bit địa chỉ ADDR, 8-bit kênh dữ liệu vào (DIN), 8-bit kênh dữ liệu ra (DOUT), tín hiệu điều khiển đọc ghi (WE = 1 ghi đọc dữ liệu, OE = 1 ouput enable cho đọc dữ liệu), tín hiệu CS để chọn chip.

11. Thiết kế module RAM hai cổng đồng bộ 128 x 8-bit, hỗ trợ các động tác đọc ghi dữ liệu trên hai kênh độc lập, mỗi tác vụ đọc ghi được thực hiện trong một xung nhịp đồng hồ. Yêu cầu về mặt giao diện trên mỗi kênh như sau như sau, 7-bit địa chỉ ADDR, 8-bit kênh dữ liệu vào (DIN), 8-bit kênh dữ liệu ra (DOUT), tín hiệu điều khiển đọc ghi (WE = 1 ghi đọc dữ liệu, OE = 1 ouput enable cho đọc dữ liệu), tín hiệu CS để chọn chip. Tín hiệu xung nhip CLK dùng chung cho hai kênh.

12. Thiết kế module FIFO 8-bit, dung lượng tối đa là 128 x 8-bit hỗ trợ các động tác đọc ghi dữ liệu trong một xung nhịp đồng hồ. Yêu cầu về mặt giao diện như

sau, 8-bit kênh dữ liệu vào (DIN), 8-bit kênh dữ liệu ra (DOUT), tín hiệu điều khiển đọc ghi (RW = 1 đọc dữ liệu, RW = 0 ghi dữ liệu), tín hiệu FULL báo hiệu FIFO đã đầy, EMPTY báo hiệu FIFO rỗng.

13. Thiết kế module STACK 8-bit, dung lượng tối đa 128 x 8-bit hỗ trợ đọc ghi dữ liệu, trong một xung nhịp đồng hồ. Yêu cầu về mặt giao diện như sau, 8-bit kênh dữ liệu vào (DIN), 8-bit kênh dữ liệu ra (DOUT), tín hiệu điều khiển đọc ghi (POP = 1 đọc dữ liệu, POP = 0 ghi dữ liệu), tín hiệu FULL báo hiệu STACK đã đầy, EMPTY báo hiệu STACK rỗng

14. Dùng các thiết kế về số họ logic ở trên tổ chức kênh số học logic thực hiện các phép cộng trừ, các phép logic AND, OR, NOT. Giao diện của kênh như sau: tín hiệu 3-bit OPCODE quy định phép toán được thực hiện, tín hiệu OP1, OP2 8-bit là các hạng tử đầu vào. Tín hiệu đầu ra RES 8-bit chứa kết quả.

15. Thiết kế bộ biến đổi nối tiếp – song song 8-bit hoạt động đồng bộ, sử dụng trong giao thức RS232. Cấu trúc sử dụng hai thanh ghi 8-bit, một thanh ghi đóng vai trò là thanh ghi dịch trái qua phải theo từng xung nhịp, khi nhận đủ 8-bit dữ liệu thì sẽ thực hiện ghi song song sang thanh ghi thứ hai.

Câu hỏi ôn tập lý thuyết

1. Trình bày sơ lược về ngôn ngữ mô tả phần cứng, các ngôn ngữ phổ biến, ưu điểm của phương pháp dùng HDL để thiết kế phần cứng

2. Cấu trúc của thiết kế bằng VHDL.

3. Các dạng mô tả kiến trúc khác nhau trong VHDL, ưu điểm, nhược điểm và ứng dụng của từng loại.

4. Trình bày về đối tượng dữ liệu trong VHDL.

5. Trình bày về các kiểu dữ liệu trong VHDL, kiểu dữ liệu tiền định nghĩa và dữ liệu định nghĩa bởi người dùng.

6. Toán tử và biểu thức trong VHDL.

7. Phát biểu tuần tự, bản chất, ứng dụng, lấy ví dụ VHDL đơn giản về phát biểu này.

8. Phát biểu đồng thời, bản chất, ứng dụng, lấy ví dụ VHDL đơn giản về phát biểu này.

9. Phân loại mã nguồn VHDL, thế nào là mã tổng hợp được và mã chỉ dùng mô phỏng.

10. Yêu cầu chung đối với kiểm tra thiết kế trên VHDL, các dạng kiểm tra thiết kế trên VHDL.

11. Vai trò và phương pháp tổ chức kiểm tra nhanh module VHDL.

12. Vai trò và phương pháp tổ chức kiểm tra tự động module VHDL.

13. Mô tả khối tổ hợp trên VHDL, ví dụ.

14. Mô tả mạch tuần tự trên VHDL, ví dụ.