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.

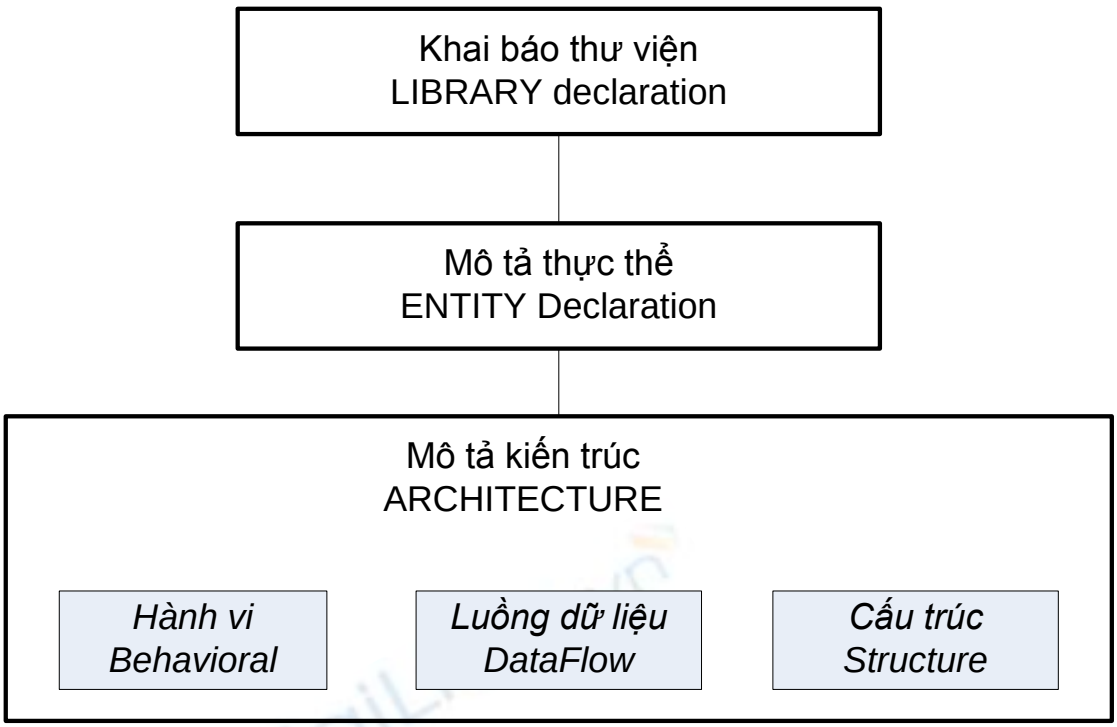
- *Tính độc lập với công nghệ:* VHDL hoàn toàn độc lập với công nghệ chế tạo phần cứng. Một mô tả hệ thống chức năng dùng VHDL thiết kế ở mức thanh ghi truyền tải RTL có thể được tổng hợp thành các mạch trên các công nghệ bán dẫn khác nhau. Nói một cách khác khi một công nghệ phần cứng mới ra đời nó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

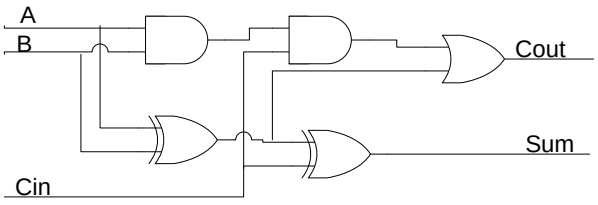
2.2.1 Mô tả luồng dữ liệu

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 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;