บทที่ 4
โปรแกรมย่อย (Subprogram)

โปรแกรมย่อยคือกลุ่มของคำสั่งลำดับที่สามารถเรียกใช้งานได้ โปรแกรมย่อยนี้สามารถเขียน ไว้ในส่วนของ Package หรือ Architecture หรือ Process ก็ได้ แต่การเรียกใช้งาน จะขึ้นอยู่กับตำแหน่งที่ประกาศโปรแกรมย่อยไว้ดังนี้

- ประกาศไว้ใน Package สามารถเรียกใช้ได้จากทุกๆที่ของโมเดล
- ประกาศไว้ใน Architecture ของโมเดลใดก็สามารถเรียกใช้ได้จากภายในโมเดลนั้นเท่านั้น
- ประกาศไว้ใน Process ใดก็สามารถเรียกใช้ได้จากภายใน Process นั้นเท่านั้น

ดังนั้นจะเห็นได้ว่าถ้าต้องการให้สามารถเรียกใช้งานได้ง่ายและทุกๆที่ ก็ควรประกาศไว้ภายใน Package จะดีกว่าการประกาศไว้ที่อื่นๆ

ภาษา VHDL แบ่งโปรแกรมย่อยออกเป็น 2 ประเภทคือ ฟังก์ชัน (Function) และโปรซีเยอร์ (Procedure)

4.1 ฟังก์ชัน (Function)

ฟังก์ชันเป็นโปรแกรมย่อยชนิดหนึ่ง เมื่อเรียกใช้จะส่งค่ากลับคืนเพียงหนึ่ งค่า ทั้งนี้การเรียกใช้อาจมีการส่งค่าพารามิเตอร์ (Passing parameter) ให้กับฟังก์ชันหรือไม่ก็ได้ แต่ถ้ามีการส่งค่า เข้าสู่ฟังก์ชัน พารามิเตอร์นั้น ต้องไม่มีการเปลี่ยนแปลงค่า แต่ค่าของพารามิเตอร์จะถูกนำไปใช้เพื่อคำนวณหาผลลัพธ์ โดยผลลัพธ์นี้จะถูกส่งกลับไปยังโมเดลที่เรียกใช้ ตรงตำแหน่งที่ฟังก์ชันนั้นถูกเรียกใช้ ลักษณะของฟังก์ชั นมีคุณสมบัติดังนี้

- การทำงานเป็น นิพจน์(expression) ไม่ใช่ statement
- ฟังก์ชันจะคำนวณให้ผลลัพธ์เพียงค่าเดียว และส่งค่ากลับคืนให้โมเดล
- ภายในฟังก์ชันต้องมีคำสั่ง RETURN เสมอ
- การทำงานของฟังก์ชันต้องไม่เปลี่ยนแปลงค่าออบเจ็กต์ (Object) ที่ถูกส่งผ่าน
- ฟังก์ชันต้องประกอบด้วย **function body** และอาจจะมี **function declaration** ด้วยก็ได้

Function declaration

เป็นส่วนที่อาจจะมีหรือไม่ก็ได้ มีรูปแบบดังนี้

```
FUNCTION name [(formal_parameter_list)] RETURN TYPE;
```

- **name** เป็นชื่อของฟังก์ชัน
- **formal parameter list** เป็นรายชื่อของพารามิเตอร์ที่ฟังก์ชันต้องการให้ส่งผ่าน สามารถกำหนดคุณสมบัติเพิ่มเติมให้พารามิเตอร์ได้ เช่น CLASS, MODE และ TYPE
- **RETURN** ใช้กำหนด TYPE ของค่าที่จะส่งกลับ

เช่น FUNCTION bl2bit (a : boolean) RETURN BIT;

Function body

เป็นส่วนที่บรรยายพฤติกรรมของฟังก์ชัน ถ้าจะให้ฟังก์ชันทำอะไรส่วนนี้จะต้องมี ภายใน body เป็น คำสั่งลำดับ ห้ามมีคำสั่ง concurrent และต้องมีคำสั่ง RETURN ซึ่งเป็นคำสั่งให้ส่งค่ากลับและเป็นคำสั่งสิ้นสุดการทำงานด้วย รูปแบบของ function body เป็นดังนี้

```
FUNCTION name [(formal_parameter_list)] RETURN TYPE IS
        declaration statement
BEGIN
        [sequential statements]
        [include RETURN expressito]
END name;
```

ตัวอย่างเช่น

```
FUNCTION bl2bit (a : BOOLEAN) RETURN BIT IS
BEGIN
        IF a then
                RETURN '1';
        ELSE
                RETURN '0';
        END IF;
END bl2bit;
```

จากตัวอย่าง ฟังก์ชัน bl2bit ข้างต้นเมื่อนำมาเขียนให้สมบูรณ์และเก็บไว้ใน Package จะเขียนได้ดังนี้

```
-- i2bv : Integer to Bit_vector.
-- In :  Integer, Value and width.
-- Return : Bit_vector, with left bit is the most significant bit.
package my_package is
        function i2bv (val, width : integer) return bit_vector;
end my_package;
package body my_package is
        function i2bv (val, width : integer) return bit_vector is
                variable result : bit_vector( width-1 downto 0) := (others => '0');
        variable bits : integer := width;
        begin
                for i  in 0 to bits-1 loop
                        if ( (val/(2**i) ) mod 2 = 1) then
                                result(i) := '1';
                        end if;
        end loop;
        return (result);
        end i2bv;
end my_package;
```

เมื่อต้องการเรียกใช้ฟังก์ชัน bl2bit ใน package สามารถทำได้ดังนี้

```
-- cnt4 : 4-bit binary counter .
-- model : behavioral
use work.my_package.all;

entity cnt4 is
        port(dout : out bit_vector(3 downto 0);
        clk, reset  : in bit);
end cnt4;
architecture beh of cnt4 is
begin
        counter: process(clk, reset)
        variable cnum : integer;
        begin
                if reset = '1' then
                cnum := 0;
                elsif (clk = '1') then
                cnum := cnum + 1;
                end if;
                dout <= i2bv(cnum,4);
        end process;
end beh;
```

ตัวอย่างฟังก์ชันสำหรับวงจร Full adder เมื่อเขียนอยู่ใน Package

```
library ieee;
use ieee.std_logic_1164.all;
package my_package is
        function FULLADD (A,B, CIN : std_logic )  return std_logic_vector;
end my_package ;
package body my_package is
        function FULLADD (A,B, CIN : std_logic )  return std_logic_vector is
                variable SUM, COUT : std_logic;
                variable RESULT : std_logic_vector  (1 downto 0);
        begin
                SUM := A xor B xor CIN;
                COUT := (A and B) or (A and CIN) or (B and CIN);
                RESULT := COUT & SUM;
                return RESULT;
        end FULLADD;
end my_package ;
```

เมื่อต้องการเรียกใช้ฟังก์ชัน FULLADD ใน package สามารถทำได้ดังนี้

```
library ieee;
use ieee.std_logic_1164.all;
use work. my_package.all;
entity EXAMPLE is
        port ( A,B : in std_logic_vector (3 downto 0);
                CIN : in std_logic;
                S : out std_logic_vector (3 downto 0);
                COUT : out std_logic );
end EXAMPLE;
```

```
architecture ARCHI of EXAMPLE is
        signal S0, S1, S2, S3 : std_logic_vector (1 downto 0);
        begin
                S0 <= FULLADD (A(0), B(0), CIN);
                S1 <= FULLADD (A(1), B(1), S0(1));
                S2 <= FULLADD (A(2), B(2), S1(1));
                S3 <= FULLADD (A(3), B(3), S2(1));
                S <= S3(0) & S2(0) & S1(0) & S0(0);
                COUT <= S3(1);
end ARCHI;
```

4.1.1 Recursive Function

หมายถึงฟังก์ชันที่สามารถเรียกใช้ตัวเองได้ เช่นตัวอย่างฟังก์ชันการหาค่า factorial ของจำนวน n ใดๆ สามารถเขียนเป็นฟังก์ชันได้ดังนี้

```
package my_package is
        function my_factorial(x : integer) return integer;
end my_package ;
package body my_package is
        function my_factorial(x : integer) return integer is
        begin
                if x = 1 then
                        return x;
                else
                        return (x*my_factorial(x-1));
                end if;
        end function my_factorial;
end my_package ;
```

เมื่อต้องการเรียกใช้ฟังก์ชัน my_factorial ใน package สามารถทำได้ดังนี้

```
use work. my_package.all;
entity EXAMPLE_3 is
        port (A : in integer range 0 to 255;
                Y : out integer range 0 to 255);
 end EXAMPLE_3;
architecture ARC of EXAMPLE_3 is
begin
        S <= my_factorial(A);
end ARC;
```

4.2 โปรซีเยอร์ (Procedure)

ทั้ง ฟังก์ชัน และโปรซีเยอร์ ต่างก็เป็นโปรแกรมย่อย (Subprogram) ที่มีวัตถุประสงค์ของการเขียนเหมือนกัน คือต้องการรวบรวมคำสั่งแบบลำดับที่มีการเรียกใช้บ่อยๆ เอา ไว้ให้เรียกใช้ได้สะดวกโดยไม่ต้องเขียนคำสั่งนั้นทุกๆครั้งที่ต้องการใช้งาน โครงสร้างของโปรซีเยอร์ก็คล้ายกับของฟังก์ชัน แต่มีข้อแตกต่างกันคือ

- ฟังก์ชัน ไม่สามารถเปลี่ยนแปลงค่าที่ส่งเข้าไปได้ แต่โปรซีเยอร์ทำได้
- ฟังก์ชันมี RETURN เพื่อส่งค่าคืน แต่ โปรซีเยอร์มีหรือไม่ก็ได้ ถ้ามี อาจเพียงเป็นการหยุดการทำงานของโปรซีเยอร์
- ฟังก์ชันคืนค่ากลับเพียงค่าเดียว แต่โปรซีเยอร์สามารถส่งคืนได้หลายค่า
- ทิศทางหรือ Mode ของพารามิเตอร์ที่ส่งผ่าน (formal parameter list) เข้าสู่ฟังก์ชันเป็นประเภท IN เท่านั้น แต่ของโปรซีเยอร์เป็นได้ทั้ง IN, OUT และ INOUT
- การเรียกฟังก์ชันเป็นการเรียกแบบ Expression แต่ในโปรซีเยอร์เป็นการเรียกแบบ Statement
- เช่นเดียวกับฟังก์ชัน โปรซีเยอร์ประกอบด้วย **Procedure declaration** และ **Procedure body**

Procedure Declaration

```
PROCEDURE name (formal_parameter_list);
```

- **name** ชื่อของ โปรซีเยอร์
- **formal_parameter_list** รายชื่อพารามิเตอร์ที่ ใช้ในโปรซีเยอร์ มีหน้าที่เป็นตัวบอก CLASS, NAME, MODE และ TYPE ของ object รูปแบบการเขียนเป็นดังนี้

```
(CLASS object_name : MODE TYPE)
```

- **CLASS** หมายถึงชั้นของ object ซึ่งอาจจะเป็น SIGNAL, VARIABLE หรือ CONSTANT ถ้าไม่มีการกำหนด CLASS และ MODE ภาษา VHDL จะถือว่าเป็น CONSTANT MODE IN แต่ถ้าไม่กำหนด CLASS แต่กำหนด MODE เป็น IN จะถือว่าเป็น VARIABLE
- **Object_name** ชื่อของ object
- **MODE** ทิศทางการไหลของข้อมูล มีสามชนิดคือ
  - **IN** ค่าของพารามิเตอร์ที่ส่งเข้าโปรซีเยอร์ แก้ไขหรือเปลี่ยนแปลงไม่ได้
  - **OUT** ค่าของพารามิเตอร์ที่ส่งออกจากโปรซีเยอร์
  - **INOUT** ค่าของพารามิเตอร์ที่ส่งเข้าได้รับคืนจากโปรซีเยอร์ แก้ไขหรือเปลี่ยนแปลงได้
- **TYPE** เป็นตัวกำหนดกลุ่มของค่าต่างๆที่ object สามารถมีได้

ตัวอย่างของโปรซีเยอร์ declaration ที่อยู่ใน Package declaration

```
PACKAGE util IS
      PROCEDURE add_element(element : IN REAL;
                  VARIABLE filter_data : INOUT filter_data_type);
      PROCEDURE zero_out( x : x_data_type);
      PROCEDURE still_busy;
END util
```

Procedure Body

ในส่วนของ body นี้ ประกอบด้วยลำดับของ Sequential statement ที่บรรยายความสัมพันธ์ระหว่างค่าของ input parameter กับค่าที่จะส่งกลับ ในส่วนนี้ห้ามมี Concurrent statement ในโปรซีเยอร์ไม่จำเป็นต้องมีคำสั่ง RETURN คำสั่ง RETURN มีหน้าที่หยุดการทำงานของโปรซีเยอร์ สำหรับโครงสร้างที่ไม่มีคำสั่ง RETURN การทำงานของโปรซีเยอร์จะหยุดได้ด้วยคำสั่ง END ในบรรทัดสุดท้ายของโปรซีเยอร์ รูปแบบโครงสร้าง procedure body เป็นดังนี้

```
PROCEDURE name (formal_parameter_list) IS
        -- declarative_statments
BEGIN
        --sequential_statements
END name;
```

ตัวอย่าง Procedure body

```
PACKAGE BODY util IS
        PROCEDURE add_element(element : IN REAL;
                        VARIABLE filter_data : INOUT filter_data_type ) IS
        BEGIN
                FOR IN filter_data'HIGH DOWNTO filter_data' LOW+1 LOOP
                        filter_data(i) := filter_data (i-1);
                END LOOP;
                filter_data(filter_data'LOW) := element;
        END add_element;

        PROCEDURE zero_out(input : INOUT filter_data_type) IS
        BEGIN
                FOR i IN input'RANGE LOOP
                        input(i) := 0.0;
                END LOOP;
        END zero_out;

        PROCEDURE still_busy IS
        BEGIN
                ASSERT FALSE REPORT "Still Busy!" SEVERITY NOTE;
        END stil_busy;
END util;
```

Procedure Calls

การเรียกโปรซีเยอร์มาใช้ ทำได้โดยการเขียนชื่อ (name) ของ โปรซีเยอร์ นั้น ตามด้วยรายชื่อของพารามิเตอร์ต้องการส่งค่าผ่านให้กับโปรซีเยอร์ ตามรูปแบบการเขียนดังนี้

```
procedure_name (passing_parameter_list);
```

Passing Parameter list

- สัมพันธ์โดยตำแหน่ง (Position Association) แบบนี้ให้เขียนชื่อพารามิเตอร์ที่ต้องการส่งผ่านให้ตรงกับตำแหน่งพารามิเตอร์ที่เป็นตัวรับ เช่นถ้าโปรซีเยอร์ชื่อ FULLADD
  PROCEDURE FULLADD (A,B, CIN : in BIT; C : out BIT_VECTOR (1 downto 0));
  เมื่อเรียกใช้ ด้วย FULLADD(X1,X2,X3,Y);

จึงมีความหมายว่า ค่าของ X1 ส่งให้ A X2 ส่งให้ B X3 ส่งให้ CIN และ Y ได้จาก C

- สัมพันธ์โดยชื่อ (Named Association) เป็นการบอกความสัมพันธ์ระหว่างชื่อกับชื่อ ด้วยการกำหนดว่าพารามิเตอร์ตัวส่งชื่ออะไร ต้องการส่งให้พารามิเตอร์ตัวรับชื่อว่าอะไร โดยใช้เครื่องหมาย " => " เช่น

  FULLADD(A => X1, B => X2, CIN => X3, C => Y);

ตัวอย่างโปรซีเยอร์สำหรับวงจร Full adder เมื่อเขียนอยู่ใน Package

```
library ieee;
use ieee.std_logic_1164.all;
package my_package is
  procedure fulladd(A,B, CIN : in std_logic;
                    C : out std_logic_vector(1 downto 0) );
end my_package;
```

```
package body my_package is
  procedure fulladd (A,B, CIN : in std_logic;
                     C : out std_logic_VECTOR (1 downto 0)) is
  variable S, COUT : std_logic;
  begin
    S := A xor B xor CIN;
    COUT := (A and B) or (A and CIN) or (B and CIN);
    C := COUT & S;
  end fulladd;
end my_package;
```

เมื่อต้องการเรียกโปรซีเยอร์ FULLADD ใน package สามารถทำได้ดังนี้

```
library ieee;
use ieee.std_logic_1164.all;
use work.my_package.all;
entity EXAMPLE is
        port (  A,B : in std_logic_vector (3 downto 0);
                CIN : in std_logic;
                SUM : out std_logic_vector (3 downto 0);
                COUT : out std_logic );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
        process (A,B,CIN)
        variable S0, S1, S2, S3 : std_logic_vector (1 downto 0);
        begin
                fulladd (A(0), B(0), CIN, S0);
                fulladd (A(1), B(1), S0(1), S1);
                fulladd (A(2), B(2), S1(1), S2);
                fulladd (A(3), B(3), S2(1), S3);
                SUM <= S3(0) & S2(0) & S1(0) & S0(0);
                COUT <= S3(1);
        end process;
end ARCHI;
```

4.3 การแปลงประเภทของข้อมูล (Type conversion)

เนื่องจากภาษา VHDL ค่อนข้างเข้มงวดเกี่ยวกับประเภทของข้อมูล กล่าวคือการส่งข้อมูลระหว่างออบเจ็กต์นั้น ออบเจ็กต์ต้องเป็นประเภทเดียวกัน เช่น BIT กับ BIT หรือ BIT_VECTOR กับ BIT_VECTOR จะเป็นคนละชนิดไม่ได้ เช่น BIT กับ STD_LOGIC หรือ STD_LOGIC_VECTOR กับ BIT_VECTOR ก็ไม่ได้ ดังนั้นใน LIBRARY IEEE จึงมี Package ที่ได้บรรจุฟังก์ชันเพื่อการแปลงข้อมูลบางประเภทไว้ดังตัวอย่างต่อไปนี้

ตัวอย่างฟังก์ชันที่อยู่ใน **std_logic_1164**

ฟังก์ชัน	แปลงจาก	เป็น
TO_BIT	STD_ULOGIC	BIT
TO_BITVECTOR	STD_LOGIC_VECTOR	BIT_VECTOR
TO_BITVECTOR	STD_ULOGIC_VECTOR	BIT_VECTOR
TO_STDULOGIC	BIT	STD_ULOGIC
TO_STDLOGICVECTOR	BIT_VECTOR	STD_LOGIC_VECTOR
TO_STDLOGICVECTOR	STD_ULOGIC_VECTOR	STD_LOGIC_VECTOR
TO_STDULOGICVECTOR	BIT_VECTOR	STD_ULOGIC_VECTOR
TO_STDULOGICVECTOR	STD_LOGIC_VECTOR	STD_ULOGIC_VECTOR

ตัวอย่างฟังก์ชันที่อยู่ใน **Std_Logic_Arith**

ฟังก์ชัน	จาก(ประเภท, ขนาด)	เป็น
CONV_INTEGER	INTEGER	INTEGER
CONV_INTEGER	UNSIGNED	INTEGER
CONV_INTEGER	SIGNED	INTEGER
CONV_INTEGER	STD_ULOGIC	SMALL_INT
CONV_UNSIGNED	INTEGER, INTEGER UNSIGNED,	UNSIGNED
CONV_UNSIGNED	INTEGER SIGNED, INTEGER	UNSIGNED
CONV_UNSIGNED	STD_ULOGIC, INTEGER	UNSIGNED
CONV_UNSIGNED	INTEGER, INTEGER UNSIGNED,	UNSIGNED
CONV_SIGNED CONV_SIGNED	INTEGER SIGNED, INTEGER	SIGNED
CONV_SIGNED CONV_SIGNED	STD_ULOGIC, INTEGER	SIGNED
CONV_STD_LOGIC_VECTOR	INTEGER, INTEGER UNSIGNED,	SIGNED
CONV_STD_LOGIC_VECTOR	INTEGER SIGNED, INTEGER	SIGNED STD_LOGIC_VECTOR
CONV_STD_LOGIC_VECTOR	STD_ULOGIC, INTEGER	STD_LOGIC_VECTOR
CONV_STD_LOGIC_VECTOR		STD_LOGIC_VECTOR
		STD_LOGIC_VECTOR

ฟังก์ชันที่อยู่ **Std_Logic_Unsigned** แปลงจาก STD_LOGIC_VECTOR ไปเป็น UNSIGNED INTEGER

```
CONV_INTEGER(arg: STD_LOGIC_VECTOR) return INTEGER;
```

ฟังก์ชันที่อยู่ **Std_Logic_Signed** แปลงจาก STD_LOGIC_VECTOR ไปเป็น SIGNED INTEGER

```
CONV_INTEGER(arg: STD_LOGIC_VECTOR) return INTEGER;
```

ตัวอย่างการใช้งานมีดังนี้

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY ex_conversion IS
        PORT ( a, b : IN std_logic_vector(3 downto 0);
        c : OUT std_logic_vector(7 downto 0));
END ex_conversion;
ARCHITECTURE behav OF ex_conversion IS
SIGNAL a_i, b_i, c_i : integer range 0 to 255;
BEGIN
        -- convert from std_logic_vector to integer
        a_i <= (conv_integer(a));
        b_i <= (conv_integer(b));
        c_i <= a_i * b_i;
        -- convert from integer to a 8 bit std_logic_vector
        c <= (conv_std_logic_vector(c_i,8));
END behav;
```

นอกจากนี้ยังมีฟังก์ชันที่ใช้ในการแปลงอยู่อีกมาก สามารถดูได้จาก LIBRARY IEEE

แบบฝึกหัด

4.1 จาก VHDL Code ถ้าให้ A = “0011” และ “1111” Y จงหาค่า Y

```
library ieee;
use ieee.std_logic_1164.all;

entity P7_1 is
        port (A : in  std_logic_vector(3 downto 0);
            Y : out std_logic_vector(3 downto 0));
end P7_1;
architecture str of P7_1 is
   function SL1(x: std_logic_vector(3 downto 0))
         return std_logic_vector is
     variable q: std_logic_vector(3 downto 0);
```

```
  begin
        for i in 0 to 2 loop
              q(i + 1) := x(i);
        end loop;
        q(0) := '0';
       return q;
  end SL1;
begin
  Y <= SL1(A);
end str;
```

4.2 จากฟังก์ชันการหาค่าสูงสุดของจำนวน 2 จำนวน จงเขียน VHDL Code เพื่อหาค่าสูงสุดของข้อมูลอินพุท A B C และ D เพื่อนำออกที่เอาท์พุท Y กำหนดให้ A B C D และ Y เป็น std_logic_vector ขนาด 4 บิต

```
library ieee;
use ieee.std_logic_1164.all;
package my_package is
  function MAX(A1, A2: std_logic_vector(3 downto 0))
        return std_logic_vector;
end my_package;
package body my_package is
  function MAX(A1, A2: std_logic_vector(3 downto 0))
        return std_logic_vector is
    variable Q: std_logic_vector(3 downto 0);
  begin
    if  (A1>A2)  then
      Q := A1;
    elsif (A2 > A1) then
      Q := A2;
    else
      Q := A1;
    end if;
    return (Q);
 end MAX;
end my_package;
```

4.3 จงเขียนโปรซีเยอร์การหาค่าต่ำสุดของจำนวน 2 จำนวน แล้วเก็บไว้ใน Package กำหนดให้ จำนวนทั้งสองเป็น std_logic_vector ขนาด 4 บิต