

Hw #10 { Subprogramas }

Lenguajes de descripción de hardware

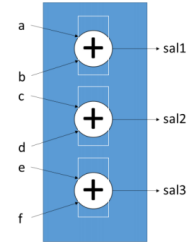
Hernández Castellanos César Uriel

1 Sección uno

Los subprogramas son una facilidad que dan los lenguajes de descripción de hardware para estructurar el código haciéndolo más legible. Pueden crearse subprogramas que pueden llamarse una o más veces de acuerdo a lo que se necesite en la descripción. En VHDL existen dos tipos de subprogramas: funciones y procedimientos. A continuación, se mencionan sus principales características:

1. Una función siempre devuelve un solo valor, mientras que un procedimiento puede hacerlo a través de sus argumentos (uno o varios).
2. Los argumentos en una función son siempre de entrada, en los procedimientos pueden ser de entrada, salida o entrada/salida.
3. Las funciones no pueden asignar valores a señales externas a ellas, los procedimientos si pueden.
4. Las funciones no pueden asignar valores a señales externas a ellas, los procedimientos si pueden.
5. Las funciones se usan en expresiones de asignación, los procedimientos no.
6. Lo declarado dentro de una función o procedimiento es local (sólo puede verse en el cuerpo del subprograma donde se declara).
7. Los subprogramas suelen tener descripciones cortas: una, dos o tres líneas. Se compilan y simulan junto con el código donde se llaman o invocan a diferencia de los componentes que se pueden compilar y simular en forma independiente.

A continuación, se mostrará un ejemplo de uso de funciones y procedimientos. Primero se mostrará con funciones y después con procedimientos. El ejemplo consiste en crear un subprograma para hacer una suma, la cual se utilizará tres veces para crear el siguiente circuito:



El cual tiene entradas y salidas de 4 bits. A continuación, se muestra el código que describe al circuito anterior utilizando funciones:

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

entity funciones is
port (
  a,b,c:in std_logic_vector(3 downto 0);
  d,e,f: in std_logic_vector(3 downto 0);
  sal1: out std_logic_vector(3 downto 0);
  sal2: out std_logic_vector(3 downto 0);
  sal3: out std_logic_vector(3 downto 0)
);
end funciones;

architecture rtl of funciones is
function suma(op1, op2: std_logic_vector)
return std_logic_vector;
function suma(op1, op2: std_logic_vector)
return std_logic_vector is
variable resultado: std_logic_vector(3downto0);
begin
  resultado := op1 + op2;
  return resultado;
end suma;

begin
  sal1 <= suma(a,b);
```

```

    sal2 <= suma(c,d);
    sal3 <= suma(e,f);
end rtl;

```

Notar que cuando se utiliza una función es necesario poner el prototipo de la misma en la zona declarativa de la arquitectura y a continuación la cabecera y cuerpo de la función. Tanto en el prototipo como en la cabecera se define el tipo (stdlogicvector) sin necesidad de indicar las dimensiones. Como los argumentos sólo son de entrada no es necesario definir la dirección. Es importante ver que entre la cabecera de la función y el inicio de la misma (begin) se abre otra zona declarativa, es ahí donde definimos una variable usada para contener el resultado de la suma. En el cuerpo de la función se observa que para asignar a una variable se utiliza el operador “:=”, y que se usa la palabra return para devolver el valor calculado por la función. El uso de una variable en este caso, obedece a la razón de que sirve para generar tres señales (una señal sólo se puede asignar a una sola señal). En la arquitectura se llama a la función en una sentencia de asignación:

```
sal1 <= suma(a,b);
```

Los argumentos en la función son siempre de entrada y el valor que retorna es asignado en la sentencia, en este caso a la señal de salida sal1.

Para realizar la descripción del mismo circuito con procedimientos, el código queda de la siguiente forma:

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity proc is
    port (
        a,b,c: in std_logic_vector(3 downto 0);
        d,e,f: in std_logic_vector(3 downto 0);
        sal1: out std_logic_vector(3 downto 0);
        sal2: out std_logic_vector(3 downto 0);
        sal3: out std_logic_vector(3 downto 0)
    );
end proc;

architecture comp of proc is
    procedure suma (

```

```

        op1, op2: in std_logic_vector;
        resultado: out std_logic_vector);
    procedure suma(op1, op2: in std_logic_vector;
        resultado: out std_logic_vector) is
    begin
        resultado := op1 + op2;
    end suma;
begin
    process(a,b,c,d,e,f)
    variable sum1: std_logic_vector(3 downto 0);
    variable sum2: std_logic_vector(3 downto 0);
    variable sum3: std_logic_vector(3 downto 0);
    begin
        suma(a,b,sum1);
        suma(c,d,sum2);
        suma(e,f,sum3);
        sal1 <= sum1;
        sal2 <= sum2;
        sal3 <= sum3;
    end process;
end comp;

```

Al igual que en el uso de funciones, los procedimientos requieren que en la zona declarativa de la arquitectura se coloque el prototipo y, la cabecera y cuerpo del procedimiento. Como los argumentos pueden ser de entrada, salida o entrada/salida; es necesario definir la dirección tanto en el prototipo como en la cabecera. Al igual que en las funciones, el tipo stdlogicvector no requiere la definición de dimensiones. Por default las salidas de un procedimiento son variables y no señales, es por esta razón que no es necesario definir una variable en el procedimiento, como si fue necesario en la función. En la arquitectura se hace el llamado al procedimiento, y como la salida de dicho procedimiento es una variable, no podemos usar directamente la señal de salida. Por lo anterior es necesario definir tres variables para asignar la salida del procedimiento. Una característica interesante es que las variables no pueden definirse en la zona declarativa de la arquitectura, pero en la zona declarativa de un bloque process si es posible. Por la razón anterior en la arquitectura se usa un bloque process para definir las tres variables que contendrán la salida de los llamados al procedimiento. Estas variables posteriormente ya se pueden asignar a las señales de salida.

Pareciera que el uso de los procedimientos requiere de más asignaciones que el uso de las funciones, pero haciendo un pequeño cambio, el uso de los procedimientos se simplifica. En el prototipo y en la cabecera del procedimiento se usará el modificador signal a la salida, con esto la salida ya no es una variable, ahora es una señal. Al ser señal la salida, ya no es necesario declarar las tres variables y por lo mismo ya no se requiere del bloque process. Ahora la salida del procedimiento puede manejar directamente a una salida de la arquitectura. El anterior código puede simplificarse de la siguiente manera:

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

```
entity proc is
  port (
    a,b,c: in std_logic_vector(3 downto 0);
    d,e,f: in std_logic_vector(3 downto 0);
    sal1: out std_logic_vector(3 downto 0);
    sal2: out std_logic_vector(3 downto 0);
    sal3: out std_logic_vector(3 downto 0)
  );
end proc;
```

```
architecture comp of proc is
  procedure suma (
    op1,op2: in std_logic_vector;
    signal resultado: out std_logic_vector
  );
```

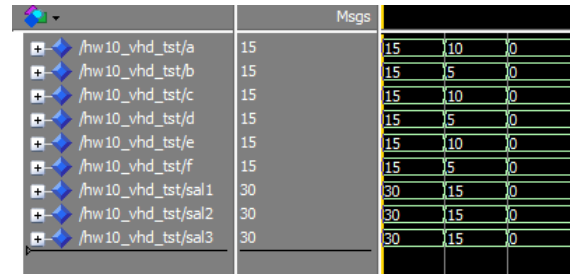
```
  procedure suma (
    op1,op2: in std_logic_vector;
    signal resultado:out std_logic_vector)
  is
  begin
    resultado <= op1 + op2;
  end suma;
begin
  suma(a,b,sal1);
  suma(c,d,sal2);
  suma(e,f,sal3);
end comp;
```

La forma anterior es más sencilla y es más parecida al

uso de las funciones.

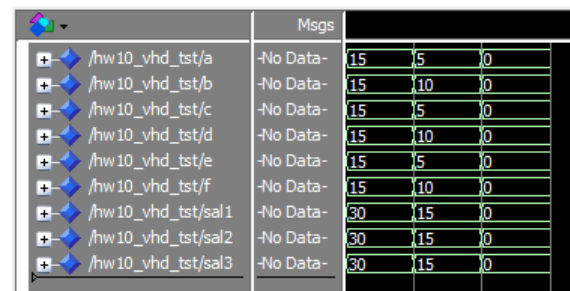
2 Sección dos

Probar los códigos anteriores escritos en VHDL y simularlos.



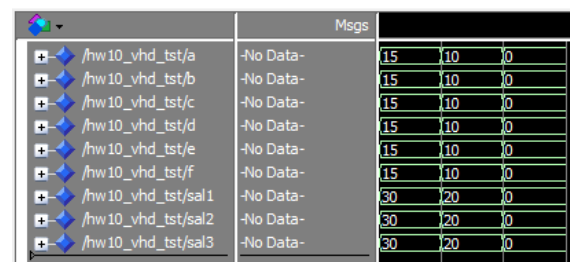
Time	Signal	Value
15	/hww10_vhd_tst/a	15
15	/hww10_vhd_tst/b	15
15	/hww10_vhd_tst/c	15
15	/hww10_vhd_tst/d	15
15	/hww10_vhd_tst/e	15
15	/hww10_vhd_tst/f	15
30	/hww10_vhd_tst/sal1	30
30	/hww10_vhd_tst/sal2	30
30	/hww10_vhd_tst/sal3	30

Figure 1: Simulación uno



Time	Signal	Value
-No Data-	/hww10_vhd_tst/a	-No Data-
-No Data-	/hww10_vhd_tst/b	-No Data-
-No Data-	/hww10_vhd_tst/c	-No Data-
-No Data-	/hww10_vhd_tst/d	-No Data-
-No Data-	/hww10_vhd_tst/e	-No Data-
-No Data-	/hww10_vhd_tst/f	-No Data-
-No Data-	/hww10_vhd_tst/sal1	-No Data-
-No Data-	/hww10_vhd_tst/sal2	-No Data-
-No Data-	/hww10_vhd_tst/sal3	-No Data-

Figure 2: Simulación dos



Time	Signal	Value
15	/hww10_vhd_tst/a	15
15	/hww10_vhd_tst/b	15
15	/hww10_vhd_tst/c	15
15	/hww10_vhd_tst/d	15
15	/hww10_vhd_tst/e	15
15	/hww10_vhd_tst/f	15
30	/hww10_vhd_tst/sal1	30
30	/hww10_vhd_tst/sal2	30
30	/hww10_vhd_tst/sal3	30

Figure 3: Simulación tres

3 Sección tres

En verilog existen las funciones (functions) y las tareas (tasks) como subprogramas. Hacer una breve descripción de sus características (como la que aparece al inicio de este documento para funciones y procedimientos)

3.1 Funciones en verilog

Una función es similar a una rutina en cualquier otro lenguaje de programación, donde se tienen argumentos de entrada y puede retornar un valor, pero en verilog existen varias restricciones:

- No se puede invocar a otra tarea, pero si función.
- No puede tener control de tiempo.
- Solo puede contener una salida.
- Solo se puede usar para modelar lógica combinatorial.

La definición de una función debe estar incluida en el módulo de un diseño. La sintaxis de las funciones es la siguiente:

```
function <rango> <nombre>;
    <argumentos>
    <declaraciones>
    <funcionalidad>
endfunction
```

Un ejemplo de función podría ser el siguiente.

```
module func;
    function [7:0] add;
        input [7:0] a, b;
        reg [7:0] res;
        begin
            res = a + b;
            add = res;
        end
    endfunction

    reg [7:0] A;

    initial
    begin
        A = add(8'b1010, 8'b1);
    end
endmodule

// resultado A: 8'b1011
```

3.2 Tareas en verilog

Una tarea en verilog puede tener argumentos, pero no un valor de retorno. Su finalidad es reducir el código de diseño y sus llamadas se realizan en tiempo de compilación. Las características de las tareas son las siguientes:

- Se define en el módulo en el que se utilizan o

pueden estar definidas en un fichero aparte y ser incluidas mediante la palabra reservada include.

- Puede invocar a otras funciones o tareas.
- Puede contener control de tiempo(delay,posedge,negedge)
- Puede tener cualquier número de entradas y/o salidas, estas marcan el orden que éstas deben pasarse en las tareas.
- Las variables declaradas son locales a dicha tarea.
- Las tareas pueden usar y/o asignar valores a cualquier señal declarada como global.
- Pueden utilizarse para modelar lógica combinatorial o secuencial.
- La llamada a una tarea no se puede realizar dentro de una expresión.

La sintaxis de las tareas es la siguiente:

```
task <nombre>;
    <argumentos>
    <declaraciones>
    <funcionalidad>
endtask
```

Un ejemplo de tarea es la siguiente

```
module tsk;
    reg ack;
    reg [7:0] data;
    task send;
        input [7:0] a;
        begin
            data = a;
            #5 S = 1;
            wait(ack != 0);
            #5 S = 0;
        end
    endtask

    initial
    begin
        #5 send(100);
    end
endmodule
```

4 Sección cuatro

Codificar la versión del circuito de los tres sumadores usando primero una función (function) y después usando una tarea (task).

4.1 Sumadores con funciones

```
module Hw10verilog(a,b,c,d,e,f,
                  sal1,
                  sal2,
                  sal3);

input  [7:0] a,b,c,d,e,f;
output [7:0] sal1,sal2,sal3;

function [7:0] sumarNumeros;
input  [7:0] opA,opB;
reg    [7:0] tmp;
begin
    tmp = opA + opB;
    sumarNumeros=tmp[7:0];
end
endfunction

assign sal1[7:0] = sumarNumeros(a,b);
assign sal2[7:0] = sumarNumeros(c,d);
assign sal3[7:0] = sumarNumeros(e,f);

endmodule
```

4.2 Sumadores con tareas

```
module Hw10verilog(a,b,c,d,e,f,
                  sal1,
                  sal2,
                  sal3);

input  [7:0] a,b,c,d,e,f;
output [7:0] sal1,sal2,sal3;

task sum (input [7:0] opA,opB,
         output [7:0] sum);
begin
    sum = opA + opB;
end
endtask

initial
begin
    sum(a,b,sal1);
    sum(c,d,sal2);
    sum(e,f,sal3);

end
endmodule
```