Sentencias

Fernando Rincón (frincon@inf-cr.uclm.es)

Arquitectura y Redes de Computadores (arco.inf-cr.uclm.es)
Escuela Superior de Informática (www.inf-cr.uclm.es)
Universidad de Castilla-La Mancha (www.uclm.es)

Contenidos

- Secuencia versus Concurrente
- Sentencias Secuenciales
- Sentencias Concurrentes
- Subprogramas

Sencuencia versus Concurrente

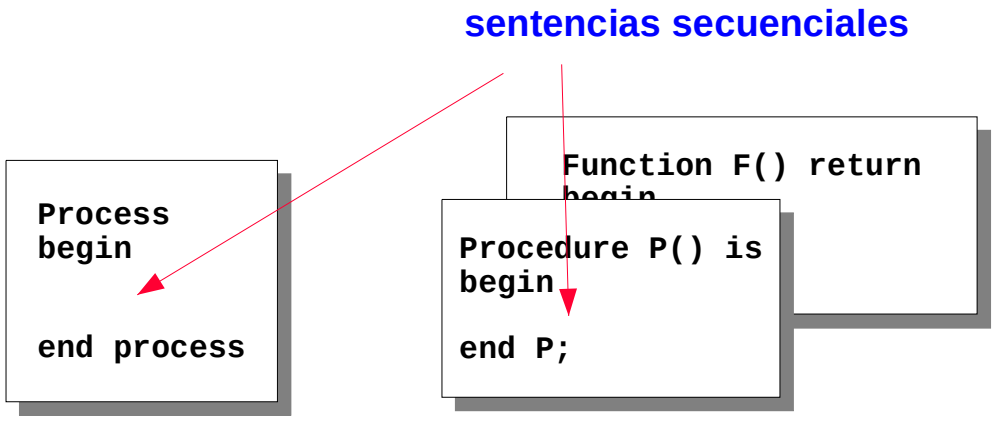
- Sentencias Concurrentes→ Describen paralelismo
  - Resultados independientes del orden de ejecución
  - Las sentencias concurrentes equivalen a procesos
- Sentencias Secuenciales → Describen algoritmos
  - El orden de las sentencias es relevante
  - Las sentencias que se encuentran dentro de los procesos **son secuenciales**.

Sencuencial versus Concurrente

- Sentencias secuenciales
  - Similares a las de los lenguajes de software (**if**, **case**, **loop**, **exit**, **return**, …)
  - Propias de VHDL (asignación a señal, **assert**, …)
  - De enlace entre secuencial y concurrente (**wait**)
- Sentencias concurrentes
  - Algunas equivalentes a las secuenciales
  - De modelado de concurrencia (**process**)
  - De descripcion de estructura (**component**)

Sentencias Secuenciales

- Se ejecutan según el orden establecido
- Se encuentran en los **procesos** o en los **cuerpos de los subprogramas**



Sentencias Secuenciales

La sentencia **wait**

- Indica el punto en el que debe suspenderse la ejecución del proceso
- Indica las condiciones de reactivación
- Puede haber más de una sentencia **wait** en un mismo proceso o subprograma
- Sintaxis:

```
[etiqueta:] wait [on <señal> {, ...}]
                 [until <expresión booleana>]
                 [for <expresión tiempo>];
```

Sentencias Secuenciales

La sentencia **wait**

```
process
begin
  <sentencias secuenciales>
  wait;
end process;
```

Sin condición de reactivación

```
process
begin
  q <= d;
  wait until Reloj = '1';
end process;
```

Sensible a una condición

```
Process
begin
  c <= a and b;
  wait on  a, b;
end process;
```

Sensible a un evento en la señal

```
Process
begin
  Reloj <= not Reloj;
  wait for 10 ns;
end process;
```

Establece un tiempo para la reactivación

Sentencias Secuenciales

La sentencia **wait**

- Formas complejas de la sentencia wait:

```
wait on a,b until c='1';
```

La condición solo se testea si hay un evento en a o b la señal c no forma parte de la lista de sensibilidad

```
wait on a, b for 10 ns;
```

Se activa por un evento o después del intérvalo de tiempo, aunque no se haya producido el evento

```
wait until a='1' for 1 ns;
```

Se activa por la condición o después del intérvalo de tiempo, aunque no se haya dado la condición

```
wait on a, b
until c='1' for 10 ns;
```

Se activa por evento o por condición o por el paso del intérvalo de tiempo especificado

Sentencias Secuenciales

Asignación a variable

- Reemplaza el valor actual con el especificado
- La asignación se realiza en el momento en que se ejecuta la sentencia
- Sintaxis:

```
[etiqueta:] <nombre variable> := <expresión>;
```

  - Estas asignaciones deben aparecer en el proceso o subprograma en el que se ha declarado la variable

Sentencias Secuenciales

Asignación a variable

- Las siguientes asignaciones no son equivalentes

```
A := B;
B := A;
```

```
B := A;
A := B;
```

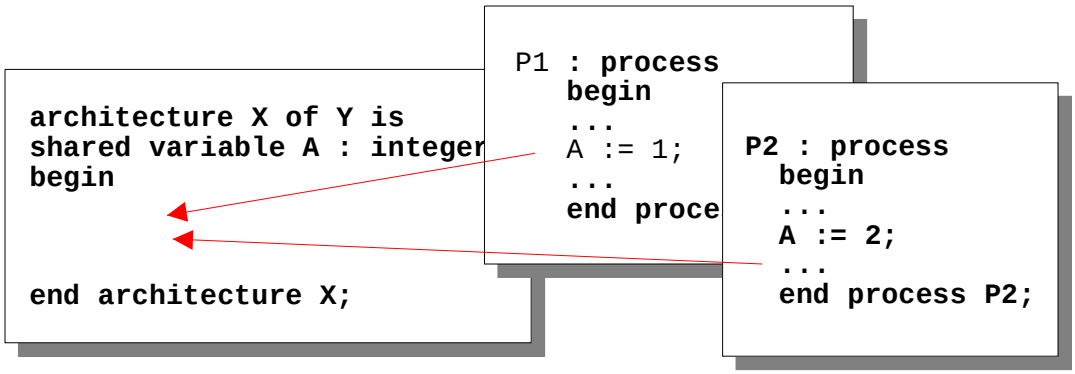
- Algunos ejemplos de asignación a variable

```
Var := '0';
Vector := "00011100";
Cadena := "Hoy es jueves";
A := B;
B := mi_funcion(3,datos)
expresion := mi_funcion(4, dir) + 2 * data;
```

Sentencias Secuenciales

Asignación a variable

- VHDL−93 incorpora las variables compartidas
  - Se declaran fuera de los procesos o subprogramas, ya que si no serían locales a estos
  - Pueden ser asignadas desde distintos procesos



Sentencias Secuenciales

Asignación a señal

- Coloca en el *driver* de la señal una nueva transacción, con el valor y tiempo indicados
- Sintaxis:

```
[etiqueta:] <nombre señal> <= [transport] <expresión>
                              {after <expresión tiempo>};
```

- Para intercambiar el valor de dos señales:

```
A <= B;
B <= A;
```

=

```
B <= A;
A <= B;
```

- Dos tipos de retardo: transporte e inercial

- Modelo de retardo tipo transporte

```
process
begin
  s <= transport '0' after 10 ns;
  s <= transport '1' after 20 ns;
  wait;
end process;
```

Cola de eventos de s

t	10 ns
v	'0'

t	10 ns	20 ns
v	'0'	'1'

```
process
begin
  s <= transport '1' after 20 ns;
  s <= transport '0' after 10 ns;
  wait;
end process;
```

t	20 ns
v	'1'

t	~~20 ns~~	10 ns
v	~~'1'~~	'0'

- Modelo de retardo tipo inercial

```
process
begin
  s <= '0' after 10 ns;
  s <= '1' after 20 ns;
  wait;
end process;
```

Cola de eventos de s

t	10 ns
v	'0'

t	~~10 ns~~	20 ns
v	~~'0'~~	'1'

```
process
begin
  s <= '1' after 20 ns;
  s <= '0' after 10 ns;
  wait;
end process;
```

t	20 ns
v	'1'

t	~~20 ns~~	10 ns
v	~~'1'~~	'0'

Sentencias Secuenciales

Asignación a señal

- Asignación de múltiples valores en una única sentencia
- Sintaxis

```
[etiqueta:] <nombre señal> <= [transport]
                              { Valor [expresión tiempo],};
```

- Todos los valores de la asignación se proyectan en el driver independientemente del tipo de retardo

```
A <= '0', '1' after 10 ns, '0' after 30 ns, '1' after 50 ns;
```


Sentencias Secuenciales

Asignación a señal

- Para retardo transporte

```
process
begin
 s <= transport 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;
 s <= transport 4 after 4 ns;
 wait;
end process;
```

t	1 ns	3 ns	5 ns
v	1	3	5

t	1 ns	3 ns	4 ns
v	1	3	4

Sentencias Secuenciales

Asignación a señal

- Para retardo inercial

```
process
begin
  s <= 1 after 1 ns, 3 after 3 ns, 5 after 5 ns;
  s <= 3 after 4 ns, 4 after 5 ns;
  wait;
end process;
```

t	1 ns	3 ns	5 ns
v	1	3	5

t	1 ns	3 ns	4 ns	5 ns
v	1	3	3	4

Sentencias Secuenciales

Asignación a señal

- VHDL'93 permite especificar la anchura de pulso mínima para la asignación a señal
- Sintaxis

```
[etiqueta:] <nombre señal> <= [transport] |
                              [reject <expresion_tiempo>]
                              <expresión> {after <expresión tiempo>};
```

  - Permite rechazar pulsos menores que el retardo del dispositivo
  - Sólo aplica para el retardo inercial

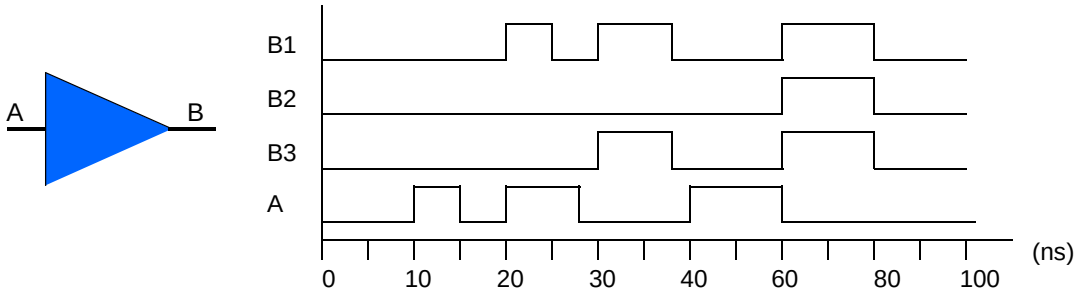
Sentencias Secuenciales

Asignación a señal

- Uso práctico de los distintos tipos de retardo

```
B1 <= transport  A after 10 ns;
B2 <= A after 10 ns;
B3 <= reject 5 ns A after 10 ns;

A <= '0', '1' after 10 ns, '0' after 15 ns, '1' after 20 ns,
     '0' after 28 ns, '1' after 40 ns, '0' after 60 ns;
```



Sentencias Secuenciales

La Sentencia **if**

- La sentencia **if** selecciona en función de una condición booleana el grupo de sentencias a ejecutar

- Sintaxis:

```
[etiqueta:] if <condicion> then
              <sentencias secuenciales>
            {elsif <condicion> then
              <sentencias secuenciales}
            [else
              <sentencias secuenciales>]
            end if [etiqueta];
```

Sentencias Secuenciales

La Sentencia **if**

- Algunos ejemplos de la sentencia **if**

```
process
begin
  if Carga='1' then
    Biestable <= Dato;
  end if;
  wait on Carga, Dato;
end process;
```

Latch

```
process
begin
  if Habilita ='1' then
    Salida <= Entrada;
  else
    Salida <= 'Z';
  end if;
  wait on Habilita, Entrada;
end process;
```

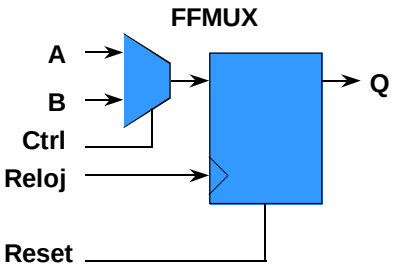
Buffer Triestado

```
process
begin
  if Iniciar ='0' then
    Q <= '0';
  elsif Reloj'event and Reloj='1' then
    Q  <= Dato;
  end if;
  wait on Iniciar, Reloj;
end process;
```

FF con reset

Sentencias Secuenciales

La Sentencia **if**




```
entity FFMUX is
port(
  Reset, Reloj, Ctrl, A, B : in bit;
  Q : out bit);
end FFMUX;

architecture funcional of FFMUX is
begin
  process
  begin
    if Reset = '1' then
      Q <= '0';
    elsif Reloj'event and Reloj ='1' then
      if Ctrl = '0' then
        Q <= A;
      else
        Q <= B;
      end if;
    end if;
    wait on Reset, Reloj;
  end process;
end funcional;
```

Sentencias Secuenciales

La Sentencia **case**

- La sentencia **case** escoge el grupo de sentencias a ejecutar de entre varias posibilidades, en función del valor de una expresión

- Sintaxis

```
[etiqueta:] case <condicion> is
              {when <valor> =>
              <sentencias secuenciales>}
              [when others =>
              <sentencias secuenciales>]
            end case [etiqueta];
```


Sentencias Secuenciales

La Sentencia **case**

- Características de los valores de selección
  - Deben cubrir todo el rango de valores de la expresión de selección
  - Tipo discreto o array unidimensional de caracteres (bit_vector, string, etc…)
  - Ayudas a la especificación de rangos:
    - Unión de valores : “00” | “01”
    - Rangos de valores (sólo aplicable a tipos enteros o enumerados): a **to** c → a, b y c.
    - **others** : sólo puede haber una, y debe ser la última opción. Se refiere a los valores restantes

Sentencias Secuenciales

La Sentencia **case**

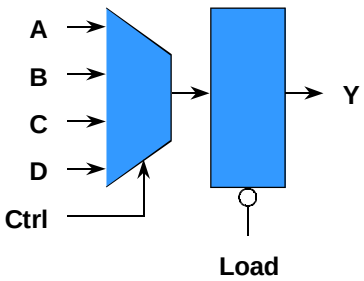
- Algunos ejemplos de la sentencia **case**

```
process
begin
  case Dia is
    when Lunes to Viernes =>
      ClaseDia <= Laboral;
    when Sabado | Domingo =>
      ClaseDia <= Festiv
  end case;
  wait on Dia;
end process;
```

```
Process
begin
  case ValEnt is
    when 0 => Res := 5;
    when 1 | 2 | 8     => Res := ValEnt;
    when 3 to 7    => Res := ValEnt + 5;
    when others    => Res := 0;
  end case;
  wait on ValEnt;
end process;
```


Sentencias Secuenciales

La Sentencia **case**




```
entity LatMux is
port(
  Load       : in bit;
  A, B, C, D : in bit;
  Ctrl       : in bit_vector(0 to 1);
  Y    : out bit);
end LatMux;
architecture funcional of LatMux is
begin
  process
  begin
    if Load='0' then
      case Ctrl is
        when "00" => Y <= A;
        when "01" => Y <= B;
        when "10" => Y <= C;
        when "11" => Y <= D;
      end case;
    end if;
  wait on Load, A, B, C, D;
  end process;
end funcional;
```

Sentencias Secuenciales

La Sentencia **loop**

- Ejecuta un grupo de sencuencias secuenciales de manera repetida un cierto número de veces
- Sintaxis:

```
[etiqueta:] [while <condicion_booleana> | for <control_repetición>]
            loop
              <sentencias secuenciales>}
            end loop [etiqueta];
```

  - Dos formas de controlar las repeticiones : **while** o **for**

Sentencias Secuenciales

La Sentencia **loop**

- Ejemplos de la sentencia **loop**

```
process
begin
  Cont <= 0;
  loop
    wait until Reloj='1';
    Cont <= (Cont + 1) mod 16;
  end loop;
end process;
```

Sin control de iteraciones

```
process
begin
  Cont <= 0;
  wait until Reloj='1';
    while Cont < 15 loop
      Cont <= Cont + 1;
      wait until Reloj='1'
  end loop;
end process;
```

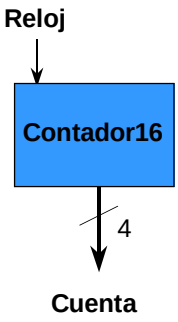
Controlado por while

```
Process
begin
  Cont <= 0;
  for I in 0 to 15 loop
    wait until Reloj='1';
      Cont <= Cont + 1;
  end loop;
  wait until Reloj='1';
end process;
```

Controlado por for

Sentencias Secuenciales

La Sentencia **loop**



Contador módulo 16

```
entity Contador16 is
port(
  Reloj  : in bit;
  Cuenta : out natural);
end Contador16;

architecture funcional of Contador16 is
signal cont : natural;
begin
  process
  begin
    Cont <= 0;
    loop
    wait until Reloj = '1';
      Cont <= (Cont + 1) mod 16;
    end loop;
  end process;
  Cuenta <= Cont;
end funcional;
```


Sentencias Secuenciales

La Sentencia **exit**

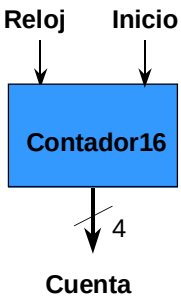
- Termina la ejecución de un bucle (**loop**)
- Sintaxis:

```
[etiqueta:] exit [etiqueta_loop] [when <condicion booleana>];
```

  - Para terminar la ejecución del loop debe cumplirse la condición (*condicion_booleana*)
  - Si se indica *etiqueta_loop*, se finaliza el loop que se identifique con dicha etiqueta, sino se finaliza el loop donde se encuentra la sentencia **exit**

Sentencias Secuenciales

La Sentencia **exit**



Contador módulo 16 con señal de inicialización asíncrona

```
entity Contador16 is
port(
  Reloj, Inicio : in bit;
  Cuenta : out natural);
end Contador16;

architecture funcional of Contador16 is
signal Cont : natural;
begin
  process
  begin
    Cont <= 0;
    loop
      wait until (Reloj'event and Reloj = '1')
                  or Inicio = '0';
      exit when Inicio = '1';
      Cont <= (Cont + 1) mod 16;
    end loop;
  end process;
  Cuenta <= Cont;
end funcional;
```


Sentencias Secuenciales

La Sentencia **next**

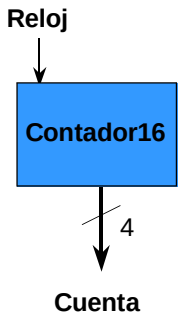
- Detiene la ejecución de un bucle y pasa a la siguiente iteración
- Sintaxis:

```
[etiqueta:] next [etiqueta_loop] [when <condicion booleana>];
```

  - Para detener la ejecución del loop debe cumplirse la condición (*condicion_booleana*)
  - Si se indica *etiqueta_loop*, se detiene el loop que se identifique con dicha etiqueta, sino se finaliza el loop donde se encuentra la sentencia **next**

Sentencias Secuenciales

La Sentencia **next**



Contador módulo 16 que nunca toma el valor 8

```
entity Contador16 is
port(
  Reloj  : in bit;
  Cuenta : out natural);
end Contador16;

architecture funcional of Contador16 is
signal cont : natural;
begin
  process
  begin
    cont <= 0;
    for I in 0 to 15 loop
      next when I=8;
      Cont <= I;
      wait until Reloj='1';
    end loop;
  end process;
  Cuenta <= Cont;
end funcional;
```


Sentencias Secuenciales

La Sentencia **assert**

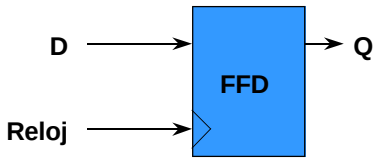
- Reporta mensajes en función de si cierta condición es o no FALSA, y permite interrumpir la simulación en función de dicha condición
- Sintaxis:

```
[etiqueta :] assert <expresion booleana>
             [report <cadena de caracteres>]
             [severity (note | warning | error |failure);
```

  - En ausencia de mensaje reporta : “Assertion violation”
  - Nivel de severidad por defecto: “error”

Sentencias Secuenciales

La Sentencia **assert**



FF tipo D con anchura mínima de pulso de 5ns

```
architecture funcional of FFD is
begin
  process (Reloj)
    variable UltimoPulso: time;
  begin
    if Reloj = '1' then
      Q <= D;
      UltimoPulso := now;
    elsif Reloj = '0' then
      assert (now - UltimoPulso) >= 5 ns
      report "Pulso de reloj menor de 5 ns"
      severity warning;
    end if;
  end process;

end funcional;
```


Sentencias Secuenciales

La Sentencia **report**

- Es una variación del VHDL−93 de la sentencia **assert**
- Sintaxis

```
[etiqueta :] [report <cadena de caracteres>]
             [severity (note | warning | error |failure);
```

  - Se envia el mensaje sin necesidad de condición
  - Si no se asume nivel de severidad se asume "note"
  - Las siguientes sentencias son equivalentes:

```
report "paso por aqui";
assert FALSE "paso por aqui" severity note;
```

Sentencias Secuenciales

Llamada a Subprograma

- Los subprogramas pueden llamarse en cualquier parte del código secuencial. Después de ejecutarse retornan el control a la siguiente instrucción

```
[etiqueta :] <nombre procedimiento> [(<parámetros>)];
```

```
<nombre función> ([<parámetros>]);
```

  - La llamada a función forma parte de la expresión de una asignación
  - La llamada a procedimiento es una sentencia secuencial en si misma

Sentencias Secuenciales

La Sentencia **return**

- Finaliza la ejecución de un subprograma
- Sintaxis:

```
[etiqueta :] return [expresion];
```

  - Retorna el control del programa a la instrucción siguiente a la que realizó la llamada
  - En un procedimiento debe usarse sin ninguna expresión
  - En una función puede usarse con una expresión, y ésta dará el valor que deba retornar la función

Sentencias Secuenciales

La Sentencia **null**

- Indica que no se debe realizar ninguna acción
- Sintaxis:

```
[etiqueta :] null;
```

- Ejemplo:

```
case valor is
  when 0 | 1  => null;
  when others => valor := valor mod 2;
end case;
```

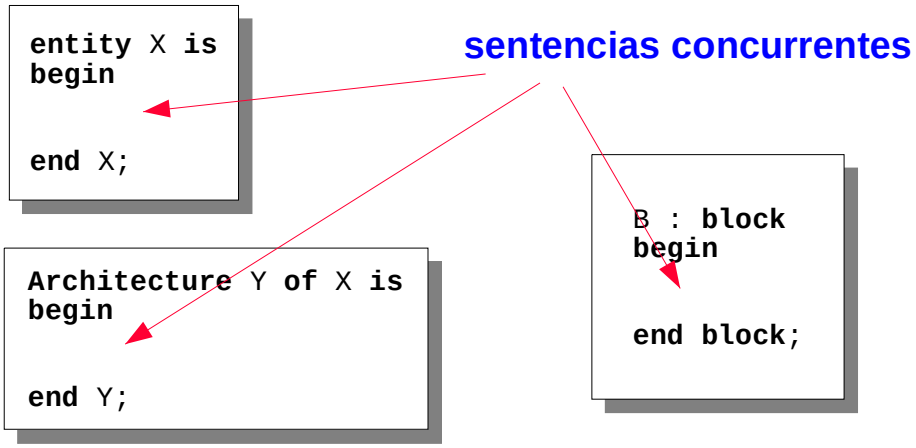

Sentencias Secuenciales

Diferencias VHDL'93 respecto al VHDL'87

- Ampliación de los modelos de retardo en las asignaciones a señal (**reject**)
- Posibilidad de uso de variables compartidas en la asignación a variable
- Sentencia **report** derivada de la sentencia assert
- Etiquetas en las sentencias (**loop** ya la tenía)
  - Mejora de la legibilidad del código
  - Posibilidad de definir atributos sobre la sentencias, estos serían usados para la síntesis y serían dependientes de la herramienta

Sentencias Concurrentes

- Se ejecutan todas en paralelo
- Se encuentran en las arquitecturas, en los bloques y, algunas de ellas, también en las entidades



Sentencias Concurrentes

- Para modelar comportamiento
  - Procesos
  - Asignaciones concurrentes
  - Sentencias **assert**
  - Llamadas a subprogramas
- Para modelar estructura
  - Referencia a componentes
  - Sentencias **generate**
  - Sentencias **block**

Sentencias Concurrentes

La Sentencia **process**

- Contiene sentencias secuenciales que definen su comportamiento

```
[<etiqueta>:] process [(<señal> ,{<señal>, ...})] [is]
  <declaraciones>
begin
  <sentencias secuenciales>
end process [<etiqueta>];
```

  - Es un bucle infinito, cuando se finaliza la última instrucción se comienza de nuevo por la primera
  - La ejecución se detiene mediante un sentencia **wait**
  - Pueden declararse datos y subprogramas locales al proceso

Sentencias Concurrentes

La Sentencia **process**

- Pueden contener más de una sentencia **wait**
- Hay una sintaxis simplificada para algunos procesos:

```
process
begin
  <sentencias secuenciales>
  wait on <lista de
    sensibilidad>;
end process;
```

=

```
process (<lista de
    sensibilidad>)
begin
  <sentencias secuenciales>
end process;
```

- Un proceso que no contenga ninguna sentencia **wait** ¡¡¡ impide el avance del tiempo de simulación **!!!**

Sentencias Concurrentes

Asignación a Señal

- Equivale a un proceso con una asignación secuencial
- Sintaxis:

```
[<etiqueta>:] <señal> <= [tipo_retardo] <forma de onda>;
```

- Proceso equivalente :

```
S <= A and B;
```

=

```
process (A, B)
begin
  S <= A and B;
end process;
```


Sentencias Concurrentes

Asignación a Señal Condicional

- Equivale a una asignación secuencial dentro de una sentencia **if**

```
[<etiqueta>:] <señal> <= [tipo_retardo]
  {<forma de onda> when <expresión booleana> else}
  <forma de onda> [when <expresión booleana>];
```

- Proceso equivalente

```
S <= E1 when Sel = '0' else
     E2;
```

=

```
process (Sel, E1, E2)
begin
  if Sel = '0' then
    S <= E1;
  else
    S <= E2;
  end if;
end process;
```


Sentencias Concurrentes

Asignación a Señal Condicional

- Diferencias en VHDL−93:
  - Posibilidad de definir la anchura del pulso (**reject**)
  - Permite que la última asignación sea condicinal
  - Permite no realizar asignación en alguna de las opciones (**unnafected**)

```
S <= E1 when Sel = '0' else
     E2 when Sel = '1' else
     unnafected when others;
```

=

```
process (Sel, E1, E2)
begin
  if Sel = '0' then
    S <= E1;
  elsif Sel = '1' then
    S <= E2;
  else
    null;
  end if;
end process;
```


Sentencias Concurrentes

Asignación a Señal con Selección

- Equivale a una asignación secuencial junto con una sentencia **case**
- Sintaxis:

```
[<etiqueta>] with <expresión> select
  <señal> <= [tipo_retardo]
    {<forma de onda> when <valor>,}
    <forma de onda> when <valor>;
```

- VHDL−93 incorpora las mismas variaciones que para la asignación condicional

Sentencias Concurrentes

Asignación a Señal con Selección

- Proceso equivalente:

```
with Operacion select
       R <= Op1  +  Op2 when suma,
            Op1  -  Op2 when resta,
            Op1 and Op2 when YL,
            Op1  or Op2 when OL;
```

=

```
process (Op1, Op2, Operacion)
begin
 case Operacion is
  when suma  => R <= Op1+Op2;
  when resta => R <= Op1-Op2;
  when YL => R <= Op1 and Op2;
  when OL => R <= Op1 or Op2;
 end case;
end process;
```


Sentencias Concurrentes

La Sentencia **assert**

- Equivale a una sentencia assert secuencial dentro de un proceso
- Sintaxis

```
[<etiqueta>:] assert <expresion booleana>
[report <cadena de caracteres>]
[severity (note | warning | error |failure);
```

- Proceso equivalente:

```
assert not(S='1' and R='1')
report "uso incorrecto del latch RS";
```

=

```
process(R, S)
begin
  assert not(S='1' and R='1')
  report "uso incorrecto del latch RS";
end process;
```

  - Puede aparecer en la entidad del diseño
  - VHDL−93 permite la sentencia **report**

Sentencias Concurrentes

Llamada a Subprogramas

- Equivalentes a la llamada secuencial en un proceso cuya lista de sensibilidad sean los parámetros del subprograma

```
[etiqueta:] <nombre procedimiento> [(<parámetros>)];
```

```
<nombre función> ([<parámetros>]);
```

  - La llamada a procedimiento es una sentencia concurrente en si misma
  - La llamada a función formará parte de una asignación

Sentencias Concurrentes

Componentes

- Permiten realizar descripciones estructurales
- Sintaxis de la declaración (arquitectura o paquete):

```
component <nombre> [is]
  [generic (<lista de genéricos>);]
  [port (<lista de puertos>);]
end [component] [<nombre>];
```

- Sintaxis de la referencia:

```
<etiqueta>: <nombre componente>
  [generic map (<lista de asociación>);]
  [port map (<lista de asociación>);]
```

Sentencias Concurrentes

Componentes

- Características de la lista de asociación **port map**
  - Asociación nominal o posicional
  - Se puede inicializar el valor del puerto en la definición del mismo
  - Indicar que un puerto de salida se deja desconectado **open**
  - VHDL−87 solo acepta señales como objetos para los puertos actuales de los componentes. También se aceptan llamadas a funciones de conversión de tipo
  - VHDL−93 también acepta constantes

Sentencias Concurrentes

Componentes

```
component ejemplo is
port(
  a, b : in bit;
  c : in bit := '1';
  d : out bit);
end component prova
```

Inicialización del puerto

Asociación por posición

```
U2 : ejemplo port map (X, Y, W, Z)
```

```
U4 : ejemplo port map
          (d=>Z, a=>X, b=>Y, c=>W);
```

Asociación por nombre

```
signal VCC : bit := '1';
signal X : std_logic;
U6 : ejemplo port map(
  a : std_logic_to_bit(X);
  b : VCC;
  c : Y;
  d : open);
```

VHDL'87

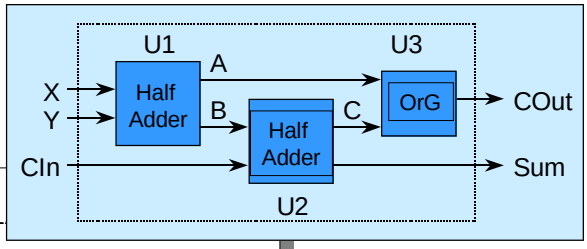
```
signal X : std_logic;
U6 : ejemplo port map(
  a : bit_vector_to_bit(X);
  b : '1';
  c : Y;
  d : open);
```

VHDL'93

Sentencias Concurrentes

Componentes

```
entity FullAdder is
begin
  port(X, Y, CIn   : in bit;
       Cout, Sum : out bit);
end FullAdder;
```




```
architecture estructural of Full
  component HalfAdder
    port(I1, I2    : in std_logic;
         COut, Sum : out std_logic);
  end component;
  component OrG
    port(I1, I2 : in std_logic;
         O      : out std_logic);
  end component;
  signal A, B, C : std_logic;
begin
  U1: HalfAdder port map (X, Y, A, B);
  U2: HalfAdder port map (B, CIn, C, Sum);
  U3: OrG port map (O => COut, I1 => A, I2 => C);
end estructural;
```

Asociación por posición

Asociación por nombre

Sentencias Concurrentes

Componentes

- VHDL−93 permite referenciar un componente sin haberlo declarado antes

```
<etiqueta>:
  entity nombre_entidad [nombre arquitectura]
  [generic map (<lista de asociación>);]
  [port map (<lista de asociación>);]
```

- En el ejemplo anterior se escribiría:

```
architecture estructural of FullAdder is
  signal A, B, C : std_logic;
begin
  U1: entity work.HalfAdder port map (X, Y, A, B);
  U2: entity work.HalfAdder port map (B, CIn, C, Sum);
  U3: entity work.OrG port map (O => COut, I1 => A, I2 => C);
end estructural;
```

Sentencias Concurrentes

La Sentencia **generate**

- Realiza varias copias de un mismo elemento
- Sintaxis:

```
<etiqueta>: {[for <especificación for> | if <condición> ]}
generate
  {<sentencias concurrentes>}
end generate;
```

  - *sentencias* puede contener cualquier sentencia concurrente VHDL
  - Su uso más habitual es la copia de N componentes (sean iguales o distintos)

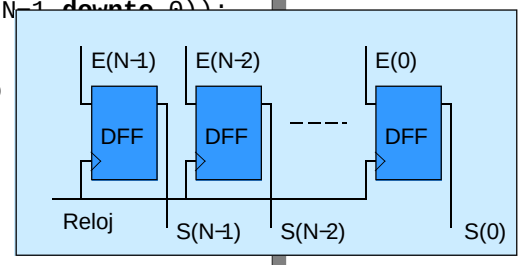
Sentencias Concurrentes

La Sentencia **generate**

```
entity Registro is
  generic (N: positive);
  port( Reloj : in  std_logic;
        E     : in  std_logic_vector(N-1 downto 0);
        S     : out std_logic_vector(N-1 downto 0));
end Registro;

architecture estructural of  Registro
  component DFF
    port (Reloj : in  std_logic;
          E     : in std_logic;
          S     : out std_logic);
  end component;

begin
  GeneraRegistro: for I in N-1 downto 0 generate
    Reg: DFF port map(Reloj, E(I), S(I));
  end generate;
end estructural;
```

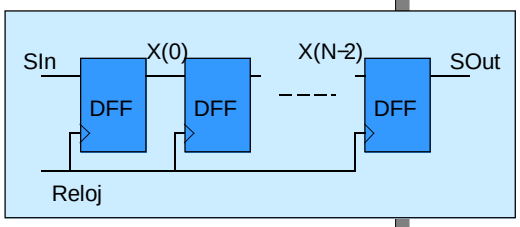


Registro de N bits

Sentencias Concurrentes

La Sentencia **generate**

```
entity ShiftReg is
  generic (N: positive);
  port( Reloj, SIn : in bit ;
        SOut       : out bit);
end ShiftReg;
architecture estructural of  ShiftReg is
  component DFF
    port (Reloj, E : in  bit;
          S        : out bit);
  end component;
  signal X : bit_vector(0 to N-2);
begin
  GeneraRegistro: for I in 0 to N-1 generate
    G1 : if (I=0) generate
           CIzq: DFF port map(Reloj, SIn, X(I)); end generate;
    G2 : if (I>0) and (I<N-1) generate
           CCen: DFF port map(Reloj, X(I-1), X(I)); end generate;
    G3 : if (I=N-1) generate
           CDer: DFF port map(Reloj, X(I-1), SOut); end generate;
  end generate;
end estructural;
```



Registro de desplazamiento de N bits

Sentencias Concurrentes

La Sentencia **block**

- La sentencia **block** puede cumplir tres funciones:
  - Estructuración de código
  - Descripción de jerarquía
  - Realizar asignaciones "guardadas" (*guarded*)
- Sintaxis:

```
<etiqueta>:block [<expresion guarda>] [is]
  [generic (<lista de genéricos>);
  [generic map (<lista de asociación>);]]
  [port (<lista de puertos>);
  [port map (<lista de asociación>);]]
  {<parte declarativa}
begin
  {<sentencias concurrentes>}
end block [<etiqueta>];
```

Sentencias Concurrentes

La Sentencia **block**

```
FFD : block (Reloj='1' and not Reloj'stable)
begin
  Q <= guarded D;
end block FFD;
```

Asignación guardada implementa un FFD

```
B1 : block
signal S : bit;
begin
  S <= A and B;
  B2 : block
  signal S : bit;
  begin
    S <= C and D;
  end block;
end block;
```

Estructuración del código :

La parte declarativa del bloque es visible en éste y en sus hijos

Señal S del bloque B1

Señal S del bloque B2

Sentencias Concurrentes

La Sentencia **block**

```
architecture ConBlock of multiplexor is
  constant NumBits : positive := 16;
  signal S1, S2, S3 : bit_vector(0 to NumBits-1);
  signal Sel : bit;
begin
  Mux : block is
    generic(Bits : positive);
    generic map(Bits => NumBits);
    port(D0, D1: in bit_vector(0 to Bits-1);
         Sel : in bit;
         Y   : out bit_vector(0 to Bits-1));
    port map (D0 => S1, D1 => S2, Y => S3, Sel => Sel);
    constant Zero : bit_vector(0 to Bits-1) := (others =>'0');
    signal A, B : bit_vector(0 to Bits-1);
    begin
      A <= D0 when Sel='0' else Zero;
      B <= D1 when Sel='1' else Zero;
      Y <= A or B;
      end block Mux;
end architecture ConBlock;
```

Descripción de jerarquía con la sentencia block

Subprogramas

Funciones

- Orientadas a realizar cálculos
- Retornan un valor resultante de su ejecución
- Sintaxis:

```
function <nombre> [(<lista de parámetros>)] return <tipo>;
```

```
function <nombre> [(<lista de parámetros>)] return <tipo> is
  {<parte declarativa>}
begin
  {<sentencias secuenciales>}
end [function] [<nombre>];
```


Subprogramas

Funciones

- Sólo pueden tener parámetros de entrada
- Debe existir al menos una sentencia **return**, que retorna el resultado
- Las declaraciones sólo son visibles dentro de la función (o en los subprogramas que se declaren en ella)
- Los datos locales sólo existen cuando la función está activa, y se crean e inicializan cada vez que se llama
- No puede declarar señales ni contener sentencias **wait**, ni modificar señales o variables externas (aunque sí puede usarlas)

Subprogramas

Funciones

```
function bv_to_natural (S: bit_vector(0 to 7))
                        return integer;
-------------------------------------------------
function bv_to_natural (S: bit_vector(0 to 7))
                        return integer is
  variable Resultado: integer := 0;
begin
  for I in S'range loop
    Resultado := Resultado * 2 + bit'pos(S(I));
  end loop;
  return Resultado;
end bv_to_natural;
```

Definición

Declaración del cuerpo de la función

```
Process
  variable Entrada : bit_vector(0 to 7);
  variable Salida1, Salida2 : natural;
begin
  ...
  Salida1 := bv_to_natural(Entrada);
  ...
  Salida2 := bv_to_natural(S=>Entrada);
  ...
end process;
```

Llamadas a función;

Paso de parámetros por posición

Paso de parámetros por nombre

Subprogramas

Procedimientos

- Realizan una serie de cambios en los datos que tratan
- Sintaxis:

```
procedure <nombre> [(<lista de parámetros>)];
```

```
procedure <nombre> [(<lista de parámetros>)] is
  {<parte declarativa>}
begin
  {<sentencias secuenciales>}
end [procedure] [<nombre>];
```

Subprogramas

Procedimientos

- Puede tener parámetros de tipo **in** (por defecto), **out** o **inout**. Los tipo **in** se consideran constantes por defecto, mientras que **out** e **inout** variables
- Retorna el control al llegar al **end**, pero puede avanzarlo con una sentencia **return**
- Las declaraciones sólo son visibles dentro del procedimiento (o en los subprogramas que se declaren en ella), y los datos locales sólo existen cuando el procedimiento está activo, y se crean e inicializan cada vez que se llama
- No puede declarar señales, pero sí contener sentencias **wait**. No puede modificar señales o variables externas (aunque sí puede usarlas)

Subprogramas

Procedimientos

```
procedure bv_to_natural (S: variable bit_vector(0 to 7);
                         X: out integer);
-------------------------------------------------------
procedure bv_to_natural (S: variable bit_vector(0 to 7);
                         X: out integer) is
  variable Resultado: integer := 0;
begin
  for I in S'range loop
    Resultado := Resultado * 2 + bit'pos(S(I));
  end loop;
  X := Resultado;
end bv_to_natural;
```

Definición

Declaración del cuerpo del procedimiento

```
process
  variable Entrada : bit_vector(0 to 7);
  variable Salida : natural;
begin
  ...
  bv_to_natural(Entrada, Salida);
  ...
  bv_to_natural(S=>Entrada, X=>Salida);
  ...
end process;
```

Llamadas a procedimiento;

Paso de parámetros por posición

Paso de parámetros por nombre