Diseño Automático de Sistemas Fiables

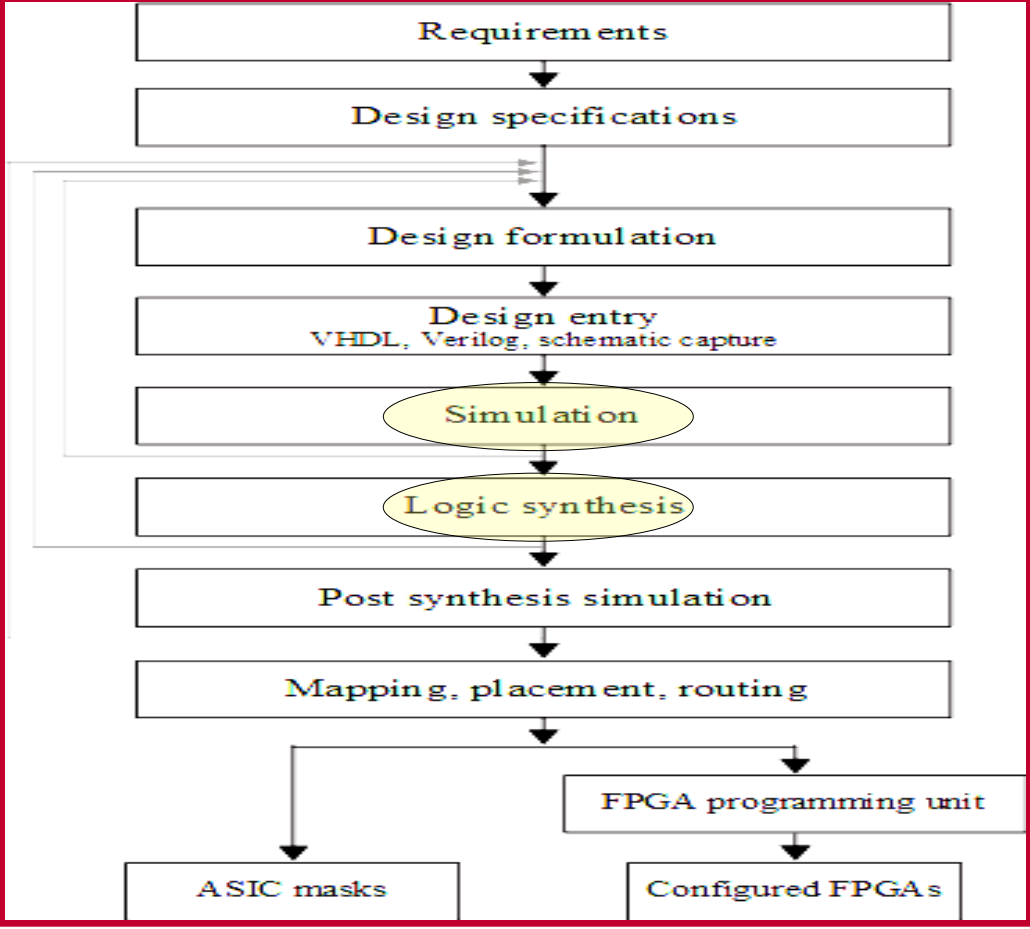
VHDL: Conceptos (Roth cap 2)

UNIVERSIDAD NEBRIJA

Objetivos Clase

- VHDL Description of Combinational Circuits
- VHDL Modules
- Sequential Statements and VHDL Processes
- Modeling Flip-Flops Using VHDL Processes
- Processes Using Wait Statements
- Two Types of VHDL Delays:Transport and Inertial Delays
- Compilation, Simulation, and Synthesis of VHDL Code
- VHDL Data Types and Operators
- VHDL Models for Multiplexers
- VHDL Libraries
- Modeling Registers and Counters Using VHDL Processes
- Behavioral and Structural VHDL
- Variables, Signals, and Constants - Arrays
- Loops in VHDL
- Assert and Report Statements

Flujo de Diseño



Sentencias Concurrentes

La mayor dificultad para modelar hardware utilizando un lenguaje informático de propósito general es representar hardware que funciona simultáneamente (concurrente).

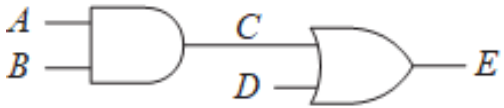
- Los Programas informáticos “normales” son secuencias de instrucciones con un orden bien definido
- En cualquier momento durante la ejecución, el programa se encuentra en un punto específico de su flujo y ejecuta las diferentes partes del programa secuencialmente.
- Para modelar circuitos combinacionales, que tienen varias compuertas (todas ellas trabajando simultáneamente), es necesario poder “simular” la ejecución de varias partes del circuito al mismo tiempo.

Sentencias Concurrentes

- VHDL modela circuitos combinacionales mediante lo que se denomina sentencias concurrentes.
- Son declaraciones que siempre están listas para ejecutarse.
- Se evalúan en cualquier momento y cada vez que cambia una señal involucrada.

Sentencias Concurrentes

- Comenzaremos describiendo un circuito simple de compuertas en VHDL.




```
C<=A  and B after 5 ns;
E<=C  or D after 5 ns;
```

- Si cada puerta en el circuito tiene un retraso de propagación de 5 ns, el circuito puede describirse mediante dos declaraciones VHDL como se muestra, donde A, B, C,D y E son señales. Una señal en VHDL normalmente corresponde a una señal en un sistema físico.

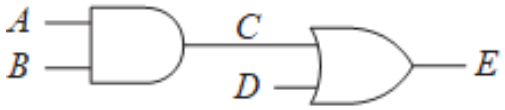
Sentencias Concurrentes

- Cuando se simulan las sentencias anteriores,
  la primera será evaluada cada vez que A o B cambien,
  la segunda se evaluará cada vez que cambie C o D.

- Supongamos que inicialmente A=1, B=C=D=E = 0.

- Si B cambia a 1 en el momento 0, C cambiará a 1 en el momento 5 ns.

- Entonces E cambiará a 1 en el instante 10 ns.




```
C<=A  and B after 5 ns;
E<=C  or D after 5 ns;
```

Sentencias Concurrentes

Las sentencias de asignación de señales en VHDL, son ejemplo de sentencias concurrentes.

- El simulador VHDL monitorea cada sentencia concurrente, y cada vez que cambia una señal, la expresión de la derecha se reevalúa inmediatamente. El nuevo valor se asigna a la señal de la izquierda, después de un retraso adecuado.
- Esta es exactamente la forma en que funciona el hardware.
- La ubicación en el programa de la sentencia concurrente no es importante.

Sentencias Concurrentes

- Cuando describimos inicialmente un circuito, es posible que no nos preocupemos por los retrasos de propagación.
- Si escribimos
  C <= A y B;
  E <= C o D;
  implica que los retrasos de propagación son 0 ns. En este caso, el simulador supone un retraso **infinitesimal** denominado Δ (delta).

- Supongamos que inicialmente A = 1, B=C=D=E = 0.
- Si B cambia a 1 en el instante 1 ns, entonces C cambiará en el instante 1+Δ y E cambiarán en el instante 1+ 2Δ .

Sentencias Concurrentes

- La expresión se evalúa cuando se ejecuta la sentencia y la señal en el lado izquierdo está programada para cambiar después de un retraso.

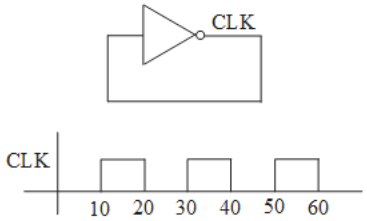
  signal_name <= expression [after delay];

- Los corchetes indican que el retraso es opcional; Si se omite after, la señal se programa para actualizarse después de un retraso delta.

- Tenga en cuenta que el momento en que se ejecuta la sentencia y el momento en el que se actualiza la señal no son los mismos.

Sentencias Concurrentes

- Aunque un programa VHDL no tenga bucles explícitos, las sentencias concurrentes pueden ejecutarse repetidamente como si estuvieran en bucle.




```
CLK <= not CLK after 10 ns;
```

- La Figura muestra un inversor con la salida conectada nuevamente a la entrada.
- Si la salida es "0", se retroalimenta a la entrada y la salida del inversor cambia a "1" después del retraso del inversor, que se supone es de 10 ns.
- Luego, el '1' retroalimenta la entrada y la salida cambia a '0' después del retraso del inversor...

Sentencias Concurrentes

- La señal CLK seguirá oscilando entre "0" y "1" como se muestra en la forma de onda. La correspondiente declaración VHDL concurrente producirá el mismo resultado.
- Si CLK se inicializa en "0", la declaración se ejecuta y CLK cambia a "1" después de 10 ns. Dado que CLK ha cambiado, la declaración se ejecuta nuevamente y CLK volverá a cambiar a "0" después de otros 10 ns. Este proceso continuará indefinidamente..
- Esta sentencia genera una forma de onda de reloj con un período medio de 10 ns.

Sentencias Concurrentes

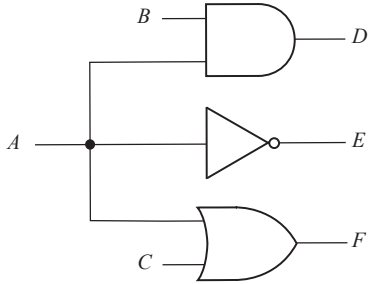
- Por otra parte, la sentencia concurrente

  CLK <= no CLK;

  provocará un error de tiempo de ejecución durante la simulación.

- Como hay 0 retraso, el valor de CLK cambiará en los instantes 0+ Δ , 0+ 2Δ , 0+ 3Δ , etc.
  Dado que es un tiempo infinitesimal, nunca avanzará a 1 ns.

Sentencias Concurrentes




```
-- when A changes, these concurrent
-- statements all execute at the
-- same time
  D <= A  and  B after  2 ns;
  E <=  not  A after  1 ns;
  F <= A  or  C after  3 ns;
```

- Las tres declaraciones concurrentes se ejecutan simultáneamente, cada vez que A cambia.
- Sin embargo, si las puertas tienen retrasos diferentes, las salidas cambiarán en diferentes instantes. Si las puertas tienen retrasos de 2 ns, 1 ns y 3 ns, respectivamente, y A cambia en el tiempo 5 ns, luego las salidas D, E y F pueden cambiar en los tiempos 7 ns, 6 ns y 8 ns.
- Si no se especificaran retrasos, entonces D, E y F serían todas actualizadas en el instante 5+ Δ.

Vectores

- Hasta ahora cada señal es de tipo bit, lo que significa que puede tener un valor de '0' o '1'. (Los valores de bits en VHDL están entre comillas simples para distinguirlos de números enteros.)
- En diseño digital, a menudo necesitamos realizar la misma operación en un grupo de señales.

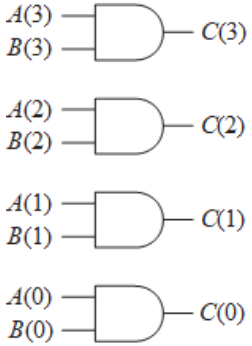
Una matriz unidimensional de señales de bits se denomina **bit-vector**.

- Se declara un vector de bits usando una sentencia como:

B: in bit_vector(3 downto 0);

- El vector de 4 bits llamado B tiene un rango de índice de 0 a 3, entonces los cuatro elementos del vector se denominan B(0), B(1), B(2) y B(3).
- La sentencia *B <= "1100"* asigna '1' a B(3), '1' a B(2), '0' a B(1) y '0' a B(0).

Vectores




```
-- the hard way
C(3) <= A(3) and B(3);
C(2) <= A(2) and B(2);
C(1) <= A(1) and B(1);
C(0) <= A(0) and B(0);
```

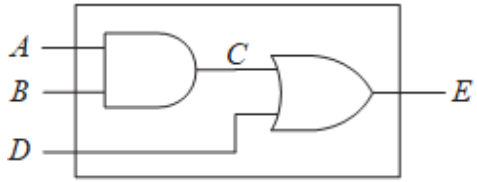
```
-- the easy way assuming C, A and
-- B are 4-bit bit-vectors

C <= A and B;
```

- Aunque podemos escribir cuatro sentencias VHDL para representar las cuatro puertas, es mucho más eficiente escribir una única sentencia que realice la operación en los vectores A y B.

- Cuando se aplica a vectores de bits, el operador *AND* realiza la operación en los pares de elementos correspondientes.

Modulos VHDL

- La estructura general de un módulo VHDL es una descripción de **Entidad** y una descripción de **Arquitectura**.
- La descripción de la entidad declara las señales de entrada y salida, y la descripción de la arquitectura especifica el funcionamiento interno del módulo.




```
entity two_gates is
  port(A, B, D: in bit; E: out bit);
end two_gates;

architecture gates of two_gates is
signal C: bit;
begin
  C <= A and B; -- concurrent
  E <= C or D;  -- statements
end gates;
```

Modulos VHDL

```
entity two_gates is
  port(A, B, D: in bit; E: out bit);
end two_gates;

architecture gates of two_gates is
signal C: bit;
begin
  C <= A and B; -- concurrent
  E <= C or D;  -- statements
end gates;
```

- La declaración Entity da el nombre *two_gates* al módulo.
- La declaración Port especifica las entradas y salidas del módulo.
- A, B y D son señales de entrada de tipo bit y E es una señal de salida de tipo bit.
- La arquitectura se denomina *gates*.
- La señal C se declara dentro de la arquitectura ya que es una señal interna.
- Las dos sentencias concurrentes que describen *gates* se colocan entre las palabras clave **begin** y **end**.

Modulos VHDL

- La descripción Entity puede considerarse como la imagen de **caja negra** del módulo que se está diseñando y su interfaz externa (representa las interconexiones del módulo con el mundo exterior).

```
entity entity-name is
[port(interface-signal-declaration);]
end [entity] [entity-name];
```

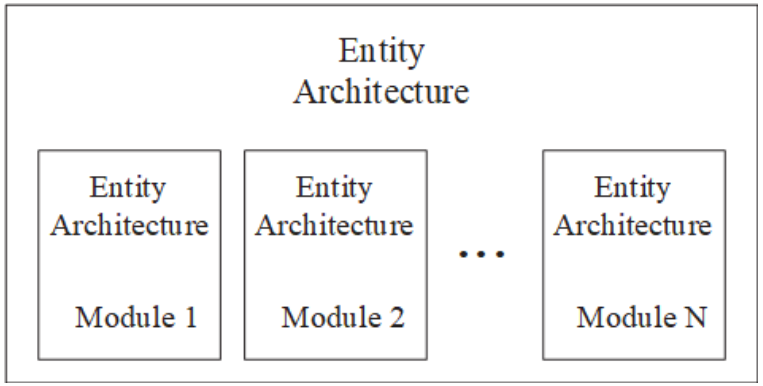
La declaración de señal de interfaz (en port) tiene la forma:

```
list-of-interface-signals: mode type [: initial-value]
{; list-of-interface-signals: mode type [: initial-value]};
```

Ejemplo:

```
port(A, B: in integer : 2; C, D: out bit);
```

Modulos VHDL



- Asociada con cada entidad hay una o más declaraciones de arquitectura de la forma

```
architecture architecture-name of entity-name is
[declarations]
begin
architecture body
end [architecture] [architecture-name];
```

- En la sección declarations, podemos declarar **señales** y **componentes** que se utilizan dentro de la arquitectura.
- El cuerpo de la arquitectura contiene declaraciones que describen el funcionamiento del módulo.

Modulos VHDL

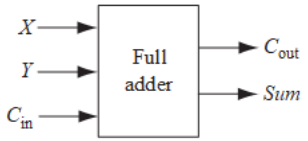
Componente

- Un componente es una unidad de diseño ideal o "virtual". Cuando realiza un diseño top down (es decir, armando el nivel superior antes de diseñar los bloques del nivel inferior), puede usar un componente para describir el tipo de interfaz que esperar para cada unidad de diseño.
- Puede pensar en esto como un marcador de posición o caja negra para una futura implementación real.
- Un componente le dice al compilador "habrá algo con este tipo de conexiones que será llamado en algún momento, pero no te preocupes por ahora".
- Tenga en cuenta que no necesita un componente; puede realizar una instanciación directa, que significa que el compilador ya conoce una entidad, por lo que no es necesario definir el "socket" por separado.

Modulos VHDL

Ejemplo: Sumador (Componente 1 bit)

- Un sumador completo suma 2 bits y una entrada de acarreo para generar un bit de suma y un bit de salida de acarreo.



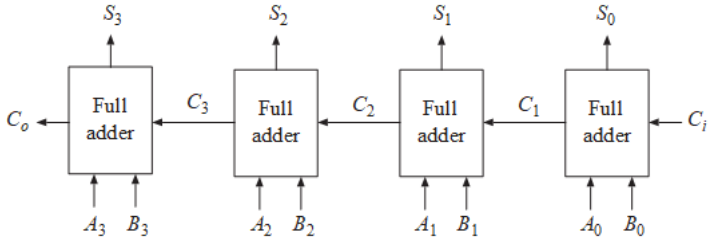

```
entity FullAdder is
  port(X, Y, Cin: in bit;     --Inputs
       Cout, Sum: out bit);   --Outputs
end FullAdder;
```

```
architecture Equations of FullAdder is
begin          -- concurrent assignment statements
  Sum  <= X xor Y xor Cin after 10 ns;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

Modulos VHDL

Ejemplo: Sumador (4 bits)

- Instanciar 4 copias del componente definido.




```
entity Adder4 is
  port(A, B: in bit_vector(3 downto 0); Ci: in bit; -- Inputs
       S: out bit_vector(3 downto 0); Co: out bit); -- Outputs
end Adder4;
architecture Structure of Adder4 is
component FullAdder
  port (X, Y, Cin: in bit;        -- Inputs
        Cout, Sum: out bit);      -- Outputs
end component;
signal C: bit_vector(3 downto 1); -- C is an internal signal
begin     --instantiate four copies of the FullAdder
  FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
  FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
  FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
  FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```

Modulos VHDL

Ejemplo: Sumador (4 bits)

- Primero declaramos el sumador de 4 bits como una entidad. Dado que las entradas y la salida de la suma tienen 4 bits de ancho, las declaramos como bit-vector dimensionados de 3 *downto* 0 (o de 1 *to* 4).
- Especificamos FullAdder como un componente dentro de la arquitectura de Adder4. La especificación del componente es muy similar a la declaración de entidad para el FullAdder, y las señales de los puertos de entrada y salida corresponden a las declaradas.
- Cuando un módulo creado en una parte del código deba usarse en otra parte, se usa una declaración de componente. No es necesario que la declaración de componente esté en el mismo archivo donde se utiliza. Es típico crear bibliotecas de componentes para su reutilización en el código y, normalmente, las declaraciones de los componentes se colocan en la biblioteca.

Modulos VHDL

Ejemplo: Sumador (4 bits)

- Siguiendo la declaración del componente, declaramos una señal de transporte interna C de 3 bits. En el cuerpo de la arquitectura, creamos varias instancias del componente FullAdder. (En la jerga CAD, “creamos instancias” de cuatro copias de FullAdder). Cada copia de FullAdder tiene un nombre (como FA0) y un mapa de puertos. Los nombres de las señales que siguen al mapa de puertos se corresponden uno a uno con las señales en el puerto del componente.
- Así, A(0), B(0) y Ci corresponden a las entradas X, Y y Cin, respectivamente.
- C(1) y S(0) corresponden a las salidas Cout y Sum. Tenga en cuenta que el orden de las señales en el mapa de puertos debe ser el mismo que en el puerto de declaración del componente.

Sentencias secuenciales y Procesos

- Las declaraciones concurrentes de la sección anterior son útiles para modelar la lógica combinacional. La lógica combinacional reacciona constantemente a los cambios de entrada.
- Por el contrario, la lógica secuencial síncrona responde a cambios que dependen del reloj. Es posible que se ignoren muchos cambios en la entrada, ya que los cambios en salida y estado ocurren solo en condiciones válidas del reloj.
- Modelar la lógica secuencial requiere primitivas para modelar la actividad selectiva condicionada al reloj, dispositivos activados por borde, secuencia de operaciones, etc.
- Un **proceso** VHDL tiene la siguiente forma:

```
process(sensitivity-list)
begin
   sequential-statements
end process;
```

Sentencias secuenciales y Procesos

- Cuando se utiliza un proceso, las declaraciones entre Begin y End se ejecutan secuencialmente.
- La expresión entre paréntesis se denomina **lista de sensibilidad** y el proceso se ejecuta cada vez que cambia cualquier señal en la lista de sensibilidad.
- Por ejemplo, si el proceso comienza con *process (A, B, C)*, esas sentencias se ejecutan **cada vez** que cambia cualquiera de A, B o C.
- Cuando *process* termina de ejecutarse, vuelve al principio y espera a que una señal en la lista de sensibilidad cambie nuevamente.

```
C <= A and B;   -- concurrent
E <= C or D;    -- statements
```

no es igual que

```
process(A, B, C, D)
begin
  C <= A and B;   -- sequential
  E <= C or D;    -- statements
end process;
```

Sentencias secuenciales y Procesos

- Se debe tener mucho cuidado al utilizar procesos para representar **lógica combinacional**.

```
entity nogates is
  port(A, B, C: in bit;
       D: buffer bit;
       E: out bit);
end nogates;

architecture behave of nogates is
begin
  process(A, B, C)
  begin
    D <= A or B after 5 ns;   -- statement 1
    E <= C or D after 5 ns;   -- statement 2
  end process;
end behave;
```

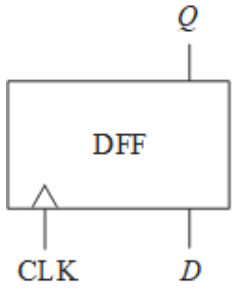
Se puede escribir este código pensando en dos puertas en cascada; sin embargo, en realidad no representa tal circuito.

Sentencias secuenciales y Procesos

- La lista de sensibilidad del proceso solo incluye A, B y C, las únicas entradas externas al circuito.
- Supongamos que todas las variables son '0' en 0 ns. Luego A cambia a '1' en 10 ns. Eso hace que se ejecute el proceso. Ambas declaraciones dentro del proceso se ejecutan una vez de forma secuencial, pero el cambio en D no ocurre justo en el momento de la ejecución. Por tanto, la sentencia 2 se ejecuta con el valor de D al inicio del proceso.
- D se convierte en "1" a los 15 ns, pero E permanece en "0". Dado que el cambio en D no se propaga a la señal E, este modelo VHDL no es equivalente a dos puertas.
- Si D estuviera incluido en la lista de sensibilidad del proceso, el proceso se ejecutaría nuevamente haciendo que E cambiara a los 20 ns. Esto daría como resultado que las salidas de la simulación coincidan con un circuito con puertas en cascada, pero es preferible realizar puertas usando declaraciones concurrentes.

Modelado de flip-flops utilizando procesos VHDL

- Un flip-flop puede cambiar de estado en el flanco ascendente o descendente de la entrada del reloj. Este tipo de comportamiento se modela en VHDL mediante un proceso. Para un flipflop D simple con una salida Q que cambia en el flanco ascendente de CLK, el proceso correspondiente es:




```
process(CLK)
begin
  if CLK'event and CLK = '1'  -- rising edge of CLK
    then Q <= D;
  end if;
end process;
```

Modelado de flip-flops utilizando procesos VHDL

- Siempre que *CLK* cambia, el proceso se ejecuta una vez y luego espera hasta que *CLK* cambie nuevamente.
- La instrucción *if* prueba si hay un flanco ascendente del reloj, y Q se establece igual a D cuando ocurre.
- La expresión *CLK 'event* se utiliza para lograr la funcionalidad de un dispositivo activado por flanco. *'event* es un atributo predefinido para cualquier señal.
- Hay dos tipos de atributos de señal en VHDL, los que devuelven valores y los que devuelven señales. El atributo *'event* devuelve un valor.
- La expresión *CLK'event* (leída como “evento de tictac de reloj”) es TRUE cada vez que la señal *CLK* cambia.
- Si *CLK* = *'1'* también es TRUE, esto significa que el cambio fue de '0' a '1', que es un flanco ascendente.

Modelado de flip-flops utilizando procesos VHDL

ATENCIÓN

- Si VHDL se utiliza sólo con fines de **simulación**, se podría utilizar sólo *if CLK = '1'*... y obtener la acción correspondiente al flanco ascendente.
- Sin embargo, cuando se utiliza el código para **sintetizar** hardware, esta declaración dará como resultado latches (pestillos), mientras que la expresión *CLK'event* da como resultado dispositivos activados por flanco, lo que queremos obtener.

Modelado de flip-flops utilizando procesos VHDL

```
process(CLK)
begin
  if CLK'event and CLK = '1'  -- rising edge of CLK
    then Q <= D;
  end if;
end process;
```

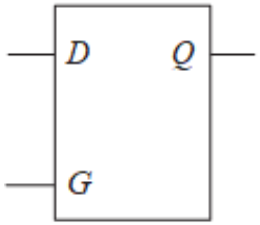
- Las declaraciones entre el inicio y el final de un proceso operan como declaraciones **secuenciales**.
- En el proceso anterior, Q <= D; es una declaración **secuencial** que solo se ejecuta siguiendo el flanco ascendente de CLK.
- Si sintetizamos el proceso anterior, el sintetizador **infiere** que Q debe ser un flip-flop ya que solo cambia en el flanco ascendente de CLK.

- Por el contrario, la declaración **concurrente** Q <= D; se ejecuta siempre que D cambia. Si sintetizamos la declaración concurrente Q <= D;, el sintetizador simplemente conectará D a Q con un cable o un buffer.

Modelado de flip-flops utilizando procesos VHDL

```
process(CLK)
begin
  if CLK'event and CLK = '1'  -- rising edge of CLK
    then Q <= D;
  end if;
end process;
```

- D no está en la lista de sensibilidad porque cambiar D no hará que el flip-flop cambie de estado.

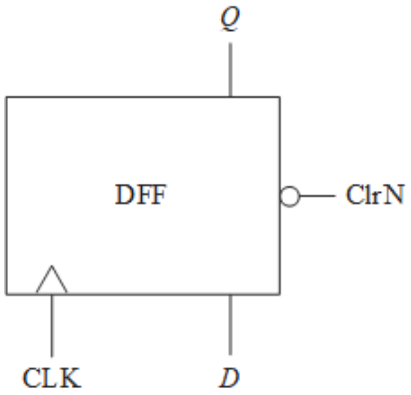



```
process(G, D)
begin
  if G = '1' then Q <= D; end if;
end process;
```

- **Latch** transparente y su representación VHDL.
- Tanto G como D están en la lista de sensibilidad ya que si G = '1', un cambio en D hace que Q cambie. Si G cambia a '0', el proceso se ejecuta, pero Q no cambia.

Modelado de flip-flops utilizando procesos VHDL

- Si un flip-flop tiene una entrada de borrado asíncrona activa-baja (ClrN) para restablecer el flip-flop independientemente del reloj, debemos modificar el proceso para que se ejecute cuando cambie CLK o ClrN.
- Agregamos ClrN a la lista de sensibilidad.
- Dado que la señal asíncrona ClrN anula CLK, ClrN se testea primero y el flip-flop se borra si ClrN es "0". De lo contrario, se verifica CLK y se actualiza Q si se ha producido un flanco ascendente.




```
process(CLK, ClrN)
begin
  if CLRn    '0' then Q <= '0';
  else if CLK'event and CLK = '1'
    then Q <= D;
  end if;
  end if;
end process;
```

Modelado de flip-flops utilizando procesos VHDL

- Las declaraciones ***if*** de VHDL son sentencias **secuenciales** que se pueden usar dentro de un proceso, pero **no** se pueden usar como concurrentes (fuera de un process).

- La forma más general de la sentencia **if** es:

```
if condition then
      sequential statements
{elsif condition then
      sequential statements}
      -- 0 or more elsif clauses may be included
[else sequential statements]
end if;
```

```
if (C1) then S1; S2;
   else if (C2) then S3; S4;
        else if (C3) then S5; S6;
             else S7; S8;
             end if;
        end if;
end if;
```

```
if (C1) then S1; S2;
   elsif (C2) then S3; S4;
   elsif (C3) then S5; S6;
   else S7; S8;
end if;
```

Proceso con Sentencias wait

- Una forma alternativa para un proceso: utilizar sentencias **wait**, en lugar de una lista de sensibilidad.
- Un proceso no puede tener a la vez sentencias **wait** y lista de sensibilidad.

```
process
begin
    sequential-statements
    wait-statement
    sequential-statements
    wait-statement
    . . .
end process;
```

- Este proceso ejecutará las sentencias secuenciales hasta que encuentre una sentencia **wait** y esperará hasta que se cumpla la condición especificada.
- Luego ejecutará el siguiente conjunto de sentencias secuenciales hasta que se encuentre otra **wait**.
- Continuará de esta manera hasta llegar al final del proceso.
- Y luego comenzará de nuevo desde el inicio del proceso.

Proceso con Sentencias wait

Las sentencias wait pueden ser de tres formas diferentes:

wait on sensitivity-list;
wait for time-expression;
wait until Boolean-expression;

- La primera forma espera hasta que cambie una de las señales en la lista .

  wait on A, B, C; espera hasta que A, B o C cambien y continúa la ejecución.

- La segunda espera hasta que haya transcurrido el tiempo especificado.

  wait for 5 ns, el proceso espera 5 ns antes de continuar. (Wait for 0 ns, espera un tiempo delta).

- La tercera evalúa la expresión booleana cada vez que una de las señales en la expresión cambie y el proceso continúa ejecutándose si la expresión es TRUE.

  wait until A = B; esperará hasta que A o B cambien. Luego evalúa si A = B y si el resultado es VERDADERO, el proceso continuará; de lo contrario, el proceso continuará esperando hasta que A o B cambien nuevamente y A = B.

Proceso con Sentencias wait

- Las sentencias wait pueden ser de tres formas diferentes:

  wait on sensitivity-list;
  wait for time-expression;
  wait until Boolean-expression;

.

Las sentencias *wait for xxx ns* son útiles para escribir código VHDL para simulación; sin embargo, no deben usarse al escribir VHDL para síntesis ya que no son sintetizables.

Un process no puede tener a la vez sentencias wait y lista de sensibilidad. No es aceptable que algunas de las señales estén en una lista de sensibilidad y otras en sentencias wait.

Debe tener o bien lista de sensibilidad o sentencias wait. De otro modo no se podrá simular el código.

Tipos de retraso

- El retraso inercial representa el tiempo que tarda una señal en propagarse a través de una puerta o un cable, pero con una duración mínima que debe cumplirse antes de que se propague la señal. También se conoce como retraso realista o retraso de evento.
  - Es el tiempo requerido por un sistema para reconocer los cambios en su entrada, por lo que los cambios se reflejarán en la salida si y solo si la entrada permanece estable más que el retraso inercial.

    En VHDL, se modela usando "*after*".
- El retraso de transporte representa el tiempo que tarda una señal en propagarse a través de una puerta o un cable. También se conoce como retraso unitario.
  - Es el tiempo que requiere el sistema para pasar sus cambios desde la entrada a la salida.

    En VHDL, se modela usando la keyword "*transport*".

Tipos de retraso

- El retraso inercial está destinado a modelar compuertas y otros dispositivos que no propagan pulsos cortos desde la entrada a la salida. Si una puerta tiene un retardo inercial ideal T, además de retrasar las señales de entrada en el tiempo T, cualquier pulso con un ancho menor que T se rechaza.
- Por ejemplo, si una puerta tiene un retraso inercial de 5 ns, pasaría un pulso de 5 ns de ancho, pero se rechazaría un pulso de 4,999 ns.
- Los dispositivos reales no se comportan de esta manera. Quizás rechacen impulsos espurios muy estrechos, pero podría no ser razonable suponer que se rechazarán todos los impulsos más estrechos que la duración del retraso.

Tipos de retraso

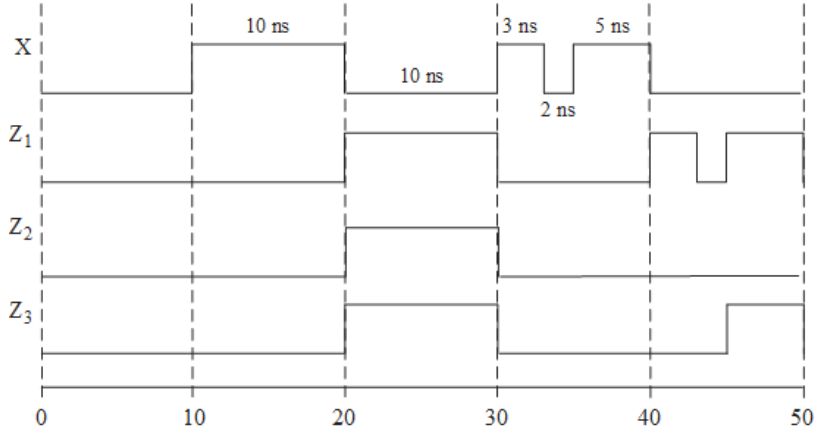
- VHDL permite modelar dispositivos que rechazan sólo pulsos muy estrechos. El rechazo de pulsos de cualquier duración arbitraria hasta el retraso de inercia especificado se puede modelar agregando una cláusula de rechazo a la declaración de asignación.

  signal_name <= ***reject*** *pulse-width* ***after*** *delay-time*

  evalúa la expresión, rechaza cualquier pulso cuyo ancho sea menor que *pulse-width* y luego establece la señal igual al resultado después de un retraso *delay-time*. En declaraciones de este tipo, el ancho del pulso de rechazo debe ser menor que el tiempo de retraso.

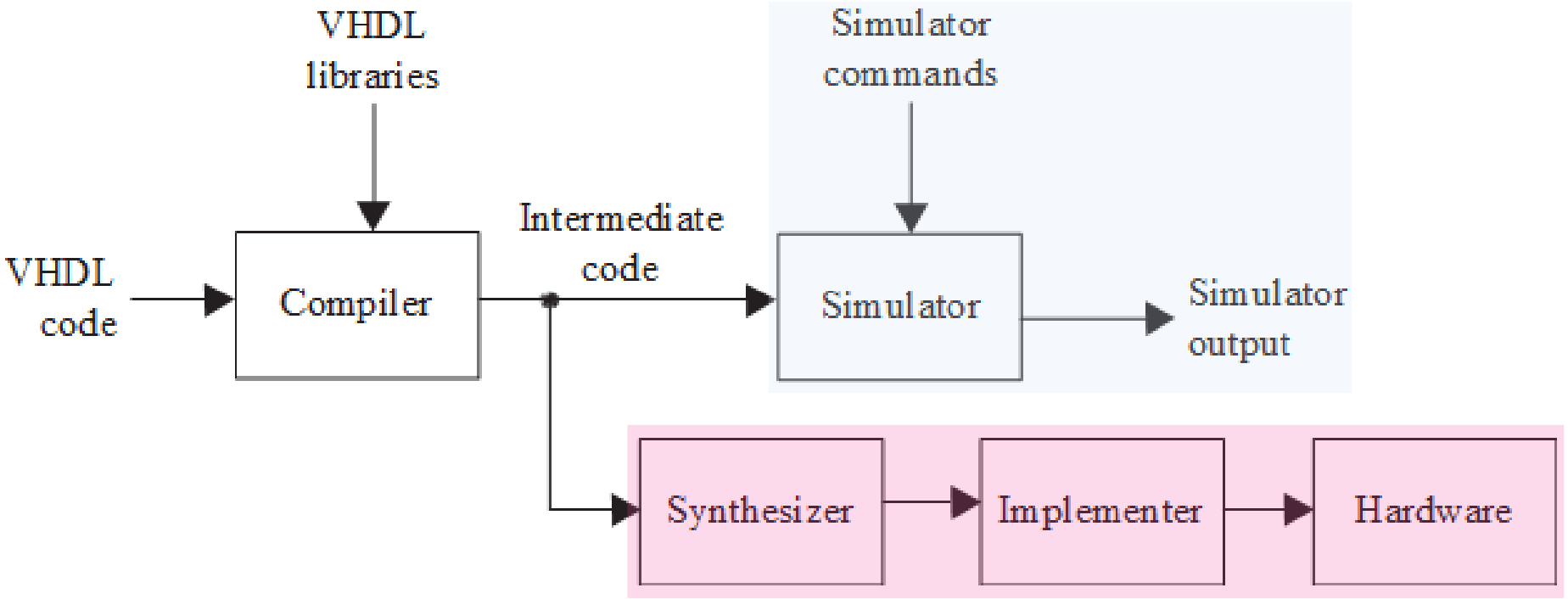
Tipos de retraso

```
Z1 <= transport X after 10 ns;  -- transport delay
Z2 <= X after 10 ns;            -- inertial delay
Z3 <= reject 4 ns X after 10 ns;  -- delay with specified
                                  -- rejection pulse width
```



- Z1 es igual que X, excepto que se desplaza 10 ns en el tiempo.
- Z2 es similar a Z1, excepto que los pulsos en X inferiores a 10 ns se filtran y no aparecen en Z2.
- Z3 es igual que Z2, excepto que solo se han rechazado los pulsos de ancho menor a 4 ns.

Compilación, Simulación y Síntesis

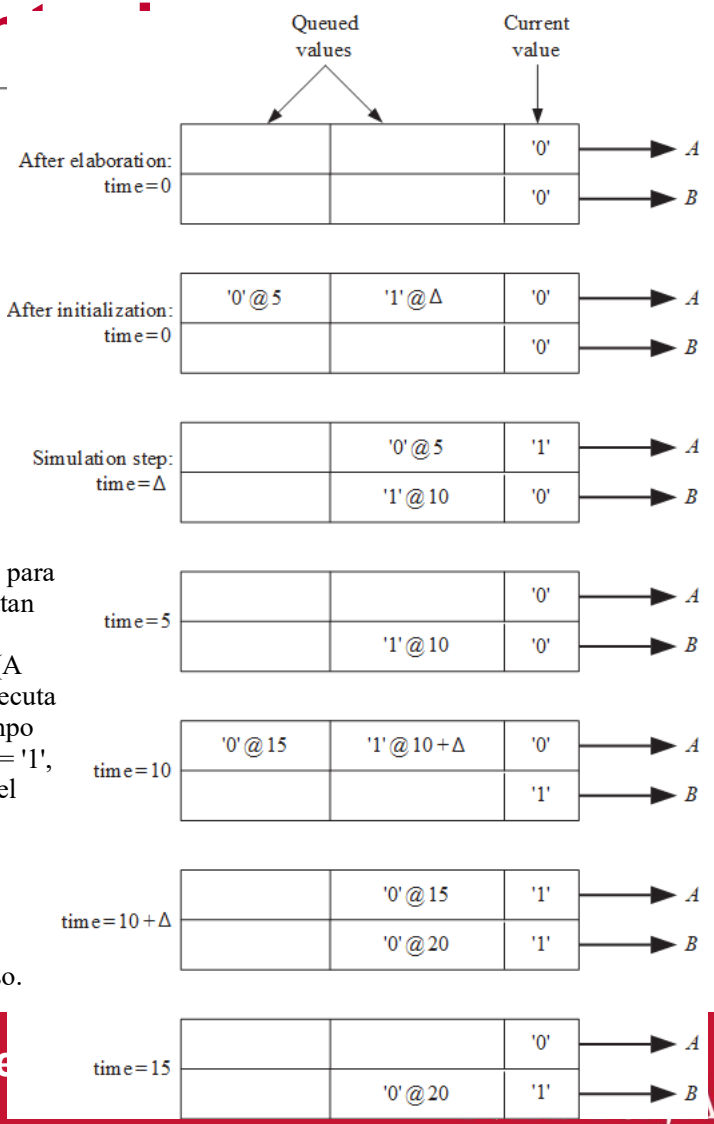


Compilación, Simulación y Síntesis

- Simulación: ver pag. 78 a 81

Compilación, Simulación y Sín

```
entity simulation_example is
end simulation_example;

architecture test1 of simulation_example is
signal A,B: bit;
begin
  P1: process(B)
  begin
    A <= '1';
    A <= transport '0' after 5 ns;
  end process P1;

  P2: process(A)
  begin
    if A = '1' then B <= not B after 10 ns; end if;
  end process P2;
end test1;
```



Una vez finalizada la elaboración, cada driver contiene "0", que es el valor inicial predeterminado para un bit. Cuando comienza la simulación, se lleva a cabo la inicialización. Ambos procesos se ejecutan simultáneamente una vez, y luego los procesos esperan hasta que cambie una señal en la lista de sensibilidad. Cuando el proceso P1 se ejecuta en el tiempo cero, se programan dos cambios en A (A cambia a '1' en el instante Δ y vuelve a '0' en el tiempo = 5 ns). Mientras tanto, el proceso P2 se ejecuta en el tiempo cero, pero no se produce ningún cambio en B, ya que A todavía es '0' en 0 ns. El tiempo avanza hasta Δ y A cambia a '1'. El cambio en A hace que se ejecute el proceso P2 y, dado que A = '1', B se programa para cambiar a '1' en el instante 10 ns. El siguiente cambio programado ocurre en el instante = 5 ns, cuando A cambia a ' 0'. Este cambio hace que se ejecute P2, pero B no cambia. B cambia a '1' en t = 10 ns. El cambio en B hace que se ejecute P1 y se programan 2 cambios en A. Cuando A cambia a '1' en t = 10+ Δ, se ejecuta el proceso P2 y se programa que B cambie en el instante 20 ns. Luego A cambia en el instante 15 ns .
La simulación continúa así hasta que se alcanza el límite de tiempo de ejecución.
Debe entenderse que A cambia en 15 ns y no en 15+ Δ. Δ se usa cuando no se especifica un retraso.

Compilación, Simulación y Síntesis

- Inicialmente la simulación era el objetivo principal de VHDL; sin embargo, hoy en día, uno de sus usos más importantes es crear automáticamente hardware a partir del código.
- El software de síntesis para VHDL traduce el código a una descripción de circuito que especifica los componentes necesarios y las conexiones entre los componentes. Aunque la síntesis se puede realizar en paralelo a la simulación, los diseñadores normalmente quieren detectar errores antes de sintetizar.
- Después de **simular** el código VHDL, para verificar que funciona correctamente, el código se puede **sintetizar** para producir una lista de los componentes necesarios y sus interconexiones. Luego, la salida del sintetizador se puede utilizar para **implementar** el sistema digital utilizando hardware específico, como un CPLD, FPGA, o ASIC.

Compilación, Simulación y Síntesis

- Las herramientas de síntesis intentan inferir los componentes de hardware necesarios "mirando" el código VHDL. Para que el código se sintetice correctamente, se deben seguir ciertas convenciones.
- Al escribir código VHDL, siempre hay que tener en cuenta que se está diseñando hardware, no escribiendo un programa de ordenador.
- Cada declaración VHDL implica ciertos requisitos de hardware. Por lo tanto, un código mal escrito puede dar lugar a hardware mal diseñado.
- Incluso si el código VHDL da el resultado correcto cuando se simula, es posible que resulte en hardware que no funcione correctamente.
- Los problemas de sincronización pueden impedir que el hardware funcione correctamente, aunque los resultados de la simulación sean correctos.

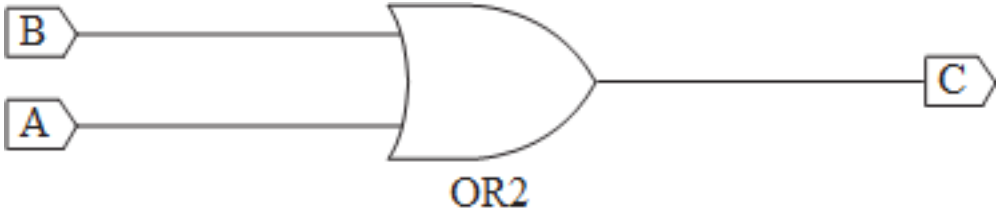
Compilación, Simulación y Síntesis. Ejemplo 1

```
entity Q1 is
  port(A, B: in bit;
       C: out bit);
end Q1;

architecture circuit of Q1 is
begin
  process(A)
  begin
    C <= A or B after 5 ns;
  end process;
end circuit;
```

Note que B falta en la lista de sensibilidad del proceso.

- Siempre que A cambie, hará que el proceso se ejecute una vez. El valor de C reflejará los valores de A y B cuando comenzó el proceso.
- Si B cambia, eso no hará que el proceso se ejecute.
- Si se sintetiza este código, se generará una compuerta OR, con el warning sobre que B no está en la lista de sensibilidad, pero seguirá adelante y sintetizará el código “correctamente”.
- El sintetizador también ignorará el retraso de 5 ns en la declaración. Si desea modelar un retraso de 5 ns, deberá utilizar contadores.

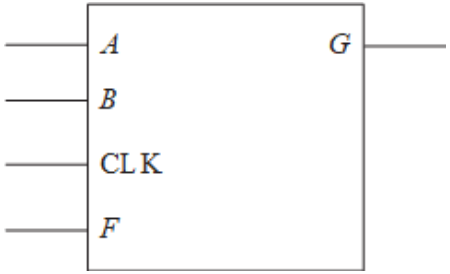
Compilación, Simulación y Síntesis. Ejemplo 1



- La salida del simulador no coincidirá con la salida del sintetizador ya que el proceso no se ejecutará cuando B cambie.
- Este es un ejemplo en el que el sintetizador adivinó un poco más de lo escrito; supuso que probablemente se requería una puerta OR y crea ese circuito (acompañado de una advertencia).
- Pero este circuito funciona de manera diferente a lo simulado antes de la síntesis. Es importante verificar siempre los warning del sintetizador sobre señales faltantes en la lista de sensibilidad.
- Quizás el sintetizador ayudó; tal vez creó hardware diferente de lo que se pretendía.

Compilación, Simulación y Síntesis. Ejemplo 2

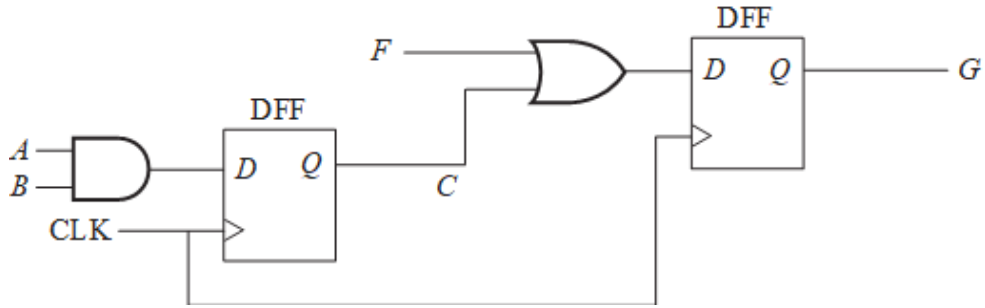
```
entity Q3 is
  port(A,B,F, CLK: in bit;
       G: out bit);
end Q3;

architecture circuit of Q3 is
signal C: bit;
begin
  process(Clk)
  begin
    if (Clk = '1' and Clk'event) then
      C <= A and B; -- statement 1
      G <= C or F; -- statement 2
    end if;
  end process;
end circuit;
```



- Pensemos en el **diagrama de bloques** del circuito representado por este código sin preocuparnos por los detalles internos.
- La capacidad de ocultar detalles y utilizar abstracciones es una parte importante de un buen diseño de sistema.
- Note que C es una señal interna, por tanto, no aparece en el diagrama.

Compilación, Simulación y Síntesis. Ejemplo 2

- Pensemos los detalles del circuito dentro del bloque. El circuito no consta de dos puertas en cascada; la asignación de señales están en un proceso.
- Un reloj activado por flanco está implícito en el uso de clk'event en la sentencia que precede a la asignación de la señal. Dado que los valores de C y G deben conservarse después del flanco del reloj, se requieren flip-flops tanto para C como para G.
- Tenga en cuenta que un cambio en el valor de C de la instrucción 1 no se considerará durante la ejecución de la instrucción 2 en ese paso del proceso. Se considerará sólo en el próximo paso, y el flip-flop para C hace que esto suceda en el hardware.



Compilación, Simulación y Síntesis. Ejemplo 2

- Entendamos por qué este código no representa una puerta AND con entradas G y D.
- Si G = '1', una puerta AND dará como resultado la salida correcta que coincida con la declaración if. Sin embargo, ¿qué sucede si actualmente Q = '1' y luego G cambia a '0'?
- Cuando G cambia a '0', una puerta AND lo propagaría a la salida; sin embargo, el dispositivo que hemos modelado aquí no debería hacerlo. Se espera que no se realicen cambios en la salida si G no es igual a "1". Por lo tanto, está claro que este dispositivo tiene que ser un latch D y no una puerta AND.
- Para inferir flip-flops o registros que cambian de estado en el flanco ascendente de una señal de reloj, el sintetizador requiere una cláusula if de la forma:

```
if clock'event and clock = '1' then . . . end if;
```

Compilación, Simulación y Síntesis. Ejemplo 2

- Para cada sentencia de asignación entre *then* y *end if*, una señal en el lado izquierdo de la sentencia provocará la creación de un registro o flip-flop.
- La moraleja de esta historia es: si no desea crear flip-flops innecesarios, no coloque las asignaciones de señales en un proceso con reloj.
- Si se omite el evento del reloj, el sintetizador puede producir latch (pestillos) en lugar de flip-flops.

Compilación, Simulación y Síntesis. Ejemplo 3

```
entity no_syn is
  port(A,B, CLK: in bit;
       D: out bit);
end no_syn;

architecture no_synthesis of no_syn is
  signal C: bit;
begin
  process(Clk)
  begin
    if (Clk='1' and Clk'event) then
      C <= A and B;
    end if;
  end process;
end no_synthesis;
```

- Si intenta sintetizar este código, el sintetizador generará un diagrama de bloques vacío. Esto se debe a que D, la salida del bloque, nunca se asigna. Generará advertencias:

  Input <CLK> is never used.
  Input <A> is never used.
  Input <B> is never used.
  Output <D> is never assigned.