

Contenidos

1 Introducción

- Objetivos y características
- Ejemplo de diseño

2 Descripción por comportamiento

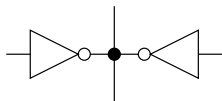
- Introducción
- Instrucción process
- Instrucción if-then-elsif-else
- Instrucción case
- Tipo de dato enumerado
- Variables
- Circuito de Mealy
- Circuito de Moore
- Circuito de asincrónico

Objetivos del Lenguaje Descriptor de Hardware

- Especificar circuitos electrónicos
- Simular el circuito previo a su construcción
- Utilizar las ventajas que brinda un compilador en el control de errores en la construcción del circuito.

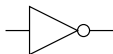
Ejemplos :

- Conectar dos salidas a un mismo nodo.

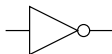


- Interfaces eléctricas incompatibles.

'1' $\Rightarrow 2.8V < V_{out} < 3.5V$
'0' $\Rightarrow 0V < V_{out} < 0.8V$



$V_{in} < 1V \Rightarrow '0'$
 $V_{in} > 4V \Rightarrow '1'$



El Lenguaje Descriptor de Hardware VHDL

■ Objetivos:

- Normalizar la especificación de los circuitos integrados de aplicación específica o A.S.I.C.
- Especificar el comportamiento de circuitos digitales pero no su construcción.

■ Origen

- VHDL es una sigla que significa VHSIC Hardware Description Language.
- VHSIC se refiere a un programa gubernamental del Departamento de Defensa de EE.UU. que dió su origen y significa Very High Speed Integrated Circuit.
- Su especificación se encuentra en la NORMA IEEE-1076.
- Permite la descripción de los circuitos en base a tres modelos
 - Comportamiento
 - Estructural
 - Data Flow

Ejemplo de diseño

Descripción en VHDL de un contador de dos bits:

- Una entrada CLK que con el flanco ascendente el contador incrementa su valor.
- Una entrada RST que cuando tenga el valor 1 las salidas se pongan en 0 independientemente de las transiciones y valores de CLK.
- Dos salidas Q0 y Q1, este último de mayor peso, cuyos valores en binario representan la cuenta hasta el momento.

La entidad de diseño

- La entidad de diseño es equivalente al encapsulado de los circuitos integrados.
- Define cuáles son sus puertos y los modos de dichos puertos.
- Ejemplos de encapsulados:



DIP

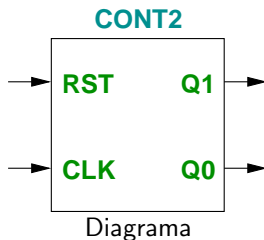


SMD



PLCC

Encapsulado y entidad de diseño



```
entity cont2 is
  port (
    rst : in  bit;
    clk : in  bit;
    q0  : buffer bit;
    q1  : buffer bit);
end;
```

Código VHDL

Descripción de la Entidad de diseño

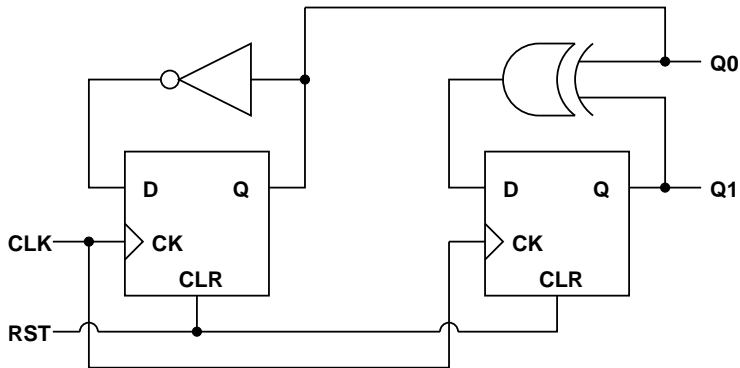
Pasos a seguir

- Indentificar la entidad con un nombre.
- Definir parámetros de construcción **generic**.
- Indentificar y declarar los puertos de acceso **port**.
- Definir los modos de los puertos **in**, **out**, **inout**, etc.

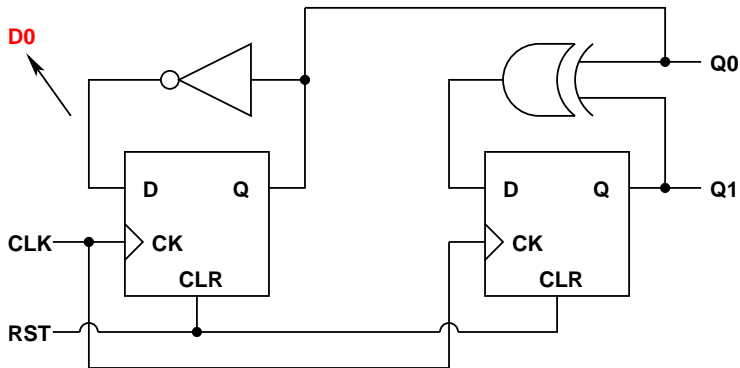
Modelos estructural y data flow

- Las instrucciones que se utilizan en estos modelos se llaman concurrentes.
- La secuencia en que aparecen las instrucciones no afecta el comportamiento del circuito especificado.
- Modelo estructural : se interconectan componentes por medio de las señales.
- Modelo data-flow : se asigna un valor a una señal mediante una expresión. También se la conoce como Lenguaje de Transferencia de Registro o RTL.

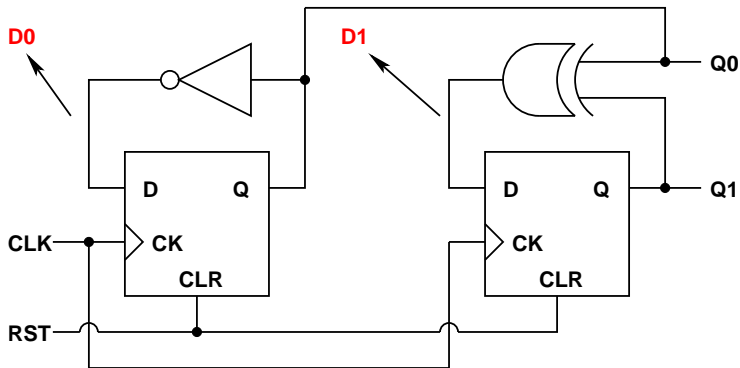
Diagrama en bloque



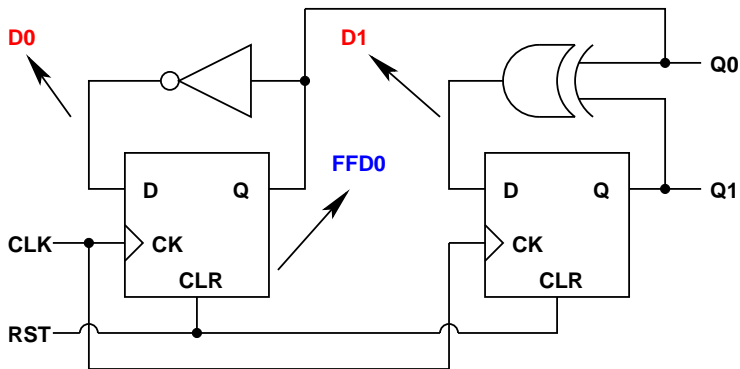
Identificación de los nodos de interconexión



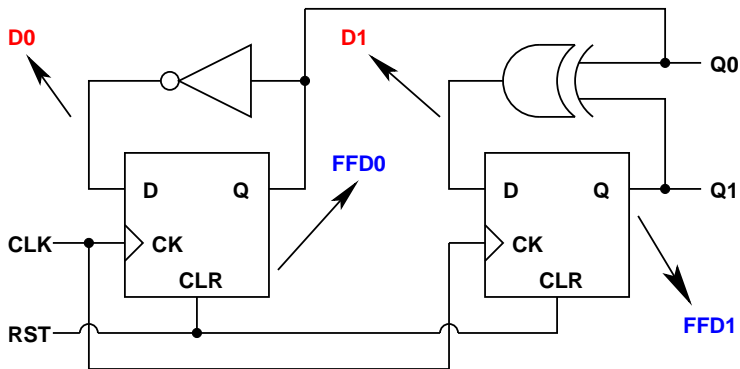
Identificación de los nodos de interconexión



Identificación de unidades de los componentes



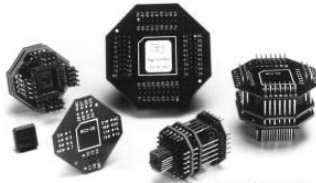
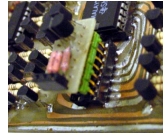
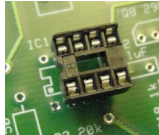
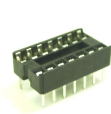
Identificación de unidades de los componentes



Compuertas y componentes

- Las compuertas lógicas se pueden describir directamente con los operadores:
 - *not*
 - *and*
 - *or*
 - *xor*
 - *nor*
 - *nand*
- Los componentes son los equivalentes de los zócalos en donde luego se conectarán los circuitos integrados representados por la entidad de diseño.

Zócalos



BC4-028-PCC6-0000

BC4-052-PCC4-0000

BC4-020-PCC2-0000

Declaración de componentes

- En el diagrama hay dos unidades de un flip-flop D.
- Se debe declarar un componente cuyo nombre arbitrario será *ffd* que luego se referirá a una entidad que cumpla con la función de dicho flip-flop (*default binding*).

```
component ffd
  port (
    clr : in    bit;
    ck  : in    bit;
    d   : in    bit;
    q   : out  bit);
end component;
```


Sintaxis del Cuerpo de arquitectura

architecture nombre **of** entidad **is**

Conjunto de declaraciones

begin

Conjunto de instrucciones

end ;

Cuerpo de arquitectura del contador de 2 bits

```
architecture mix of cont2 is
    signal d0, d1 : bit;
    component ffd
        port (
            clr : in  bit;
            ck  : in  bit;
            d   : in  bit;
            q   : out bit);
    end component;
begin
    ffd0 : ffd port map (rst, clk, d0, q0);
    ffd1 : ffd port map (rst, clk, d1, q1);
    d0 <= not q0;
    d1 <= q0 xor q1;
end;
```

Pasos a seguir

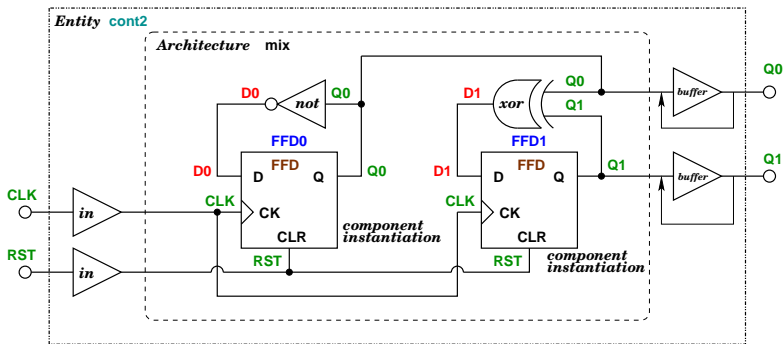
- Realizar un diagrama en bloque.
- Identificar los nodos de interconexión.
- Identificar las unidades de los componentes a utilizar.
- Declarar las señales que representan los nodos de interconexión.
- Declarar los componentes a utilizar.
- Interconectar los componentes a través de dichas señales.
- Realizar las asignaciones de señales para generar las compuertas necesarias que acompañen el circuito.

Especificación completa del contador de dos bits

```
entity cont2 is
  port (
    rst : in  bit;
    clk : in  bit;
    q0  : buffer bit;
    q1  : buffer bit);
end;
```

```
architecture mix of cont2 is
  signal d0, d1 : bit;
  component ffd
    port (
      clr : in  bit;
      ck  : in  bit;
      d   : in  bit;
      q   : out bit);
  end component;
begin
  ffd0 : ffd port map (rst, clk, d0, q0);
  ffd1 : ffd port map (rst, clk, d1, q1);
  d0 <= not q0;
  d1 <= q0 xor q1;
end;
```

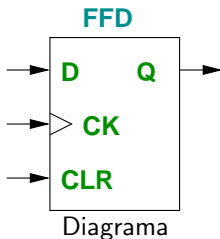
Diagrama en bloque completo del contador de dos bits



Modelo de comportamiento

- El circuito se especifica en base a una secuencia de instrucciones que indica como sus salidas se modifican en base al valor y/o cambio de sus entradas.
- La secuencia en que se escriben puede afectar el comportamiento

Entidad de diseño del flip-flop D



```
entity ffd is
  port (
    clr : in  bit;
    ck  : in  bit;
    d   : in  bit;
    q   : out bit);
end;
```

Código VHDL

Cuerpo de arquitectura del flip-flop D

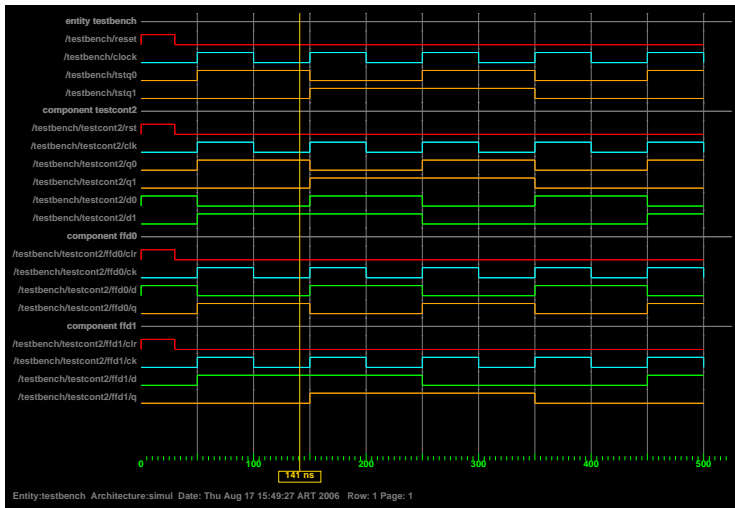
Descripción por comportamiento

```
architecture beh of ffd is
begin
    process (ck, clr)
    begin
        if clr='1' then
            q <= '0';
        elsif ck'event and ck='1' then
            q <= d;
        end if;
    end process;
end;
```


Entidad de simulación

```
entity testbench is  
end;  
  
architecture simul of testbench is  
    signal clock : bit;  
    signal reset : bit;  
    signal tstq0 : bit;  
    signal tstq1 : bit;  
  
    component cont2  
        port (  
            rst : in bit;  
            clk : in bit;  
            q0  : buffer bit;  
            q1  : buffer bit);  
    end component;  
  
begin  
    reset <= '1', '0' after 30 ns;  
    clock <= not clock after 50 ns;  
    testcont2 : cont2 port map (reset , clock , tstq0 , tstq1);  
  
end;
```

Diagrama de tiempos



Descripción por comportamiento

Introducción

- El circuito digital se describe en base a una secuencia de instrucciones que se llaman secuenciales
- Las instrucciones secuenciales solo pueden ir dentro de las instrucciones:
 - *process*
 - *function*
 - *procedure*
- Las principales instrucciones secuenciales son:
 - *if-then-else*
 - *case*
 - *asignación de señales*
 - *asignación de variables*
 - *loop*
 - *next*
 - *exit*

Nociones de expresiones sintácticas

- Una expresión sintáctica consiste de un *lado izquierdo*, el símbolo $::=$ que se puede leer como “se reemplaza por”, y un *lado derecho*.

iteration_scheme $::=$

while condition

 | **for** identifier *in* discrete_range

- Las palabras escritas en **negritas** simbolizan palabras claves. Por ejemplo: **array**, **port**, **entity**, etc.
- Las llaves { } encierran items que se repiten en el lado derecho de la expresión. Los items pueden aparecer cero o más veces.
 - term $::=$ factor { multiplying_operator factor }
 - term $::=$ factor | term { multiplying_operator factor }
- Los corchetes [] encierran items opcionales. Las siguientes expresiones son equivalentes.
 - return_statement $::=$ **return** [expression] ;
 - return_statement $::=$ **return** ; | **return** expression ;
- La barra vertical “|” separa alternativas a menos que ocurra inmediatamente después de la apertura de una llave “{”, en ese caso significa el caracter |.
 - letter_or_digit $::=$ letter | digit
 - choices $::=$ choice { | choice }

Instrucción *process*

- Es una *instrucción concurrente*
- Su bloque de instrucciones permite la descripción de circuitos en base al modelo de comportamiento.

Sintaxis :

```
[ etiqueta : ] process [ ( lista_de_sensibilidad ) ]  
    declaraciones  
    begin  
        instrucciones  
    end process ;
```

Instrucción *if-then-elsif-else*

Instrucciones secuenciales

■ Sintaxis :

```
if condition then
    secuencia_de_instrucciones_secuenciales
{ elsif condition then
    secuencia_de_instrucciones_secuenciales }
[ else
    secuencia_de_instrucciones_secuenciales ]
end if ;
```

Ejemplo de uso

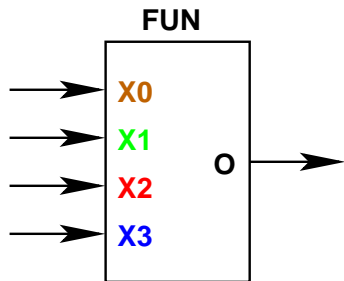
Instrucción *if-then-elsif-else*

Descripción del circuito lógico correspondiente al siguiente mapa de Karnaugh por medio del modelo de comportamiento:

	X0				
X2	X1	0 0	0 1	1 1	1 0
X3					
0 0		1	1		
0 1			1	1	
1 1					1
1 0		1		1	

Entidad de diseño

Ejemplo de uso



```
entity FUN is
  port (
    x0 : in  bit;
    x1 : in  bit;
    x2 : in  bit;
    x3 : in  bit;
    o  : out bit);
end;
```

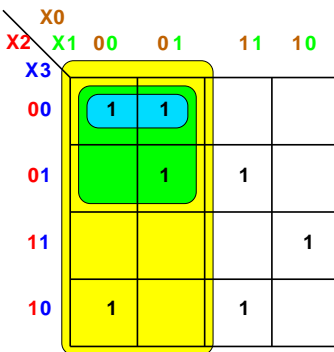

Cuerpo de arquitectura

Ejemplo de uso

```

architecture compt of fun is
begin
  process (x0, x1, x2, x3)
  begin
    o <= '0';
    if x0='0' then
      if x2='0'
        if x3='0' then
          o <= '1';
        elsif x1='1' then;
          o <= '1';
        end if;
      elsif x3='0' and x1='0' then
        o <= '1';
      end if;
    else
      if x1='1' then
        if x2='0' and x3='1' then
          o <= '1';
        elsif x2='1' and x3='0' then
          o <= '1';
        end if;
      elsif x2='1' and x3='1' then
        o <= '1';
      end if;
    end if;
  end process;
end;

```



Instrucción case

Instrucciones secuenciales

■ Sintaxis

```
case expression is  
  when option { | option } =>  
    secuencia_de_instrucciones_secuenciales  
{ when option { | option } =>  
    secuencia_de_instrucciones_secuenciales }  
end case;
```

- *option* puede ser **others** en referencia al resto de los casos de *expression* no enumerados en los valores de *option*

Ejemplo instrucción case

Instrucciones secuenciales

```
case x is  
when "00" | "11" =>  
    z <= '1';  
when others =>  
    z <= '0';  
end case;
```

Tipo de dato enumerado

- Se caracteriza por un conjunto de valores.
- Ejemplo
 - El tipo *bit* permite tener solo dos valores el '1' y el '0'.
 - Un tipo más concreto permitiría tener otros valores como por ejemplo :
 - 0 débil
 - 1 débil
 - alta impedancia
- Dos tipos distintos podrían caracterizar dos interfaces electricas diferentes.
- Los tipos se declaran en la parte declarativa de cualquier instrucción.

Tipos enumerados

Introducción a los tipos de datos

■ Sintaxis

type identifier **is** (enumeration_literal {, enumeration_literal })

enumeration_literal ::= identifier | character_literal

- *identifier* es una de las alternativas posibles que puede tener dicho tipo descripto como un identificador.
- *character_literal* es otra forma de representar una alternativa como un caracter.
- Indica los valores que puede tomar un objeto: *signal*, *variable*, *constant*, declarado con dicho tipo.
- Ejemplos:

type bit **is** ('0', '1') ;

type boolean **is** (FALSE, TRUE) ;

Variables

Instrucciones secuenciales

- Son objetos auxiliares que se utilizan en la descripción por comportamiento
- Se declaran en la parte declarativa de las instrucciones. secuenciales:
process, function, procedure.
- Sintaxis

variable *identifier_list* : *subtype_indication* [:= *expression*];

- *identifier_list* es la lista de variables a declarar.
 - *subtype_indication* indica el tipo de dato que tendrán las variables de datos declaradas.
 - := *expression* inicializa la lista de variables con el valor de la expresión.
- Ejemplo:

variable aux : bit := '1';

Asignación de *variables*

Instrucciones secuenciales

- Modifica el valor de la variable.

- Sintaxis:

[label :] target := expression ;

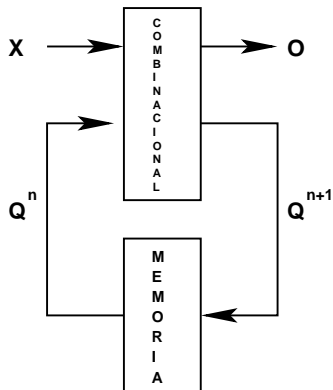
- *target* es el nombre de la variable que se quiere asignar.
- *expression* es una expresión cuyo valor se asignará en la variable.

- Ejemplo:

```
aux := cols * rows;
```

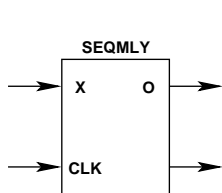
Diagrama en bloque de un circuito de Mealy

Instrucciones secuenciales

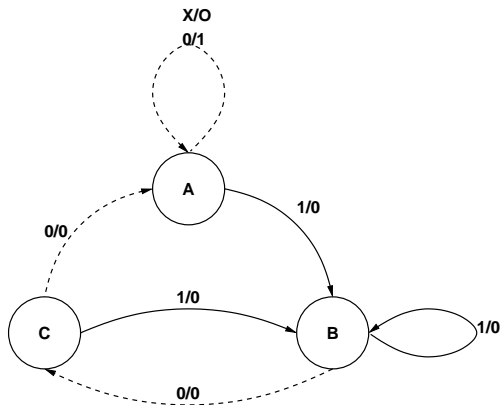


Entidad de diseño y Diagrama de estados

Ejemplo de un circuito de Mealy



```
entity seqmly is
  port (
    clk : in  bit;
    x    : in  bit;
    o    : out bit);
end;
```



Cuerpo de arquitectura y tabla de transiciones

Ejemplo de un circuito de Mealy

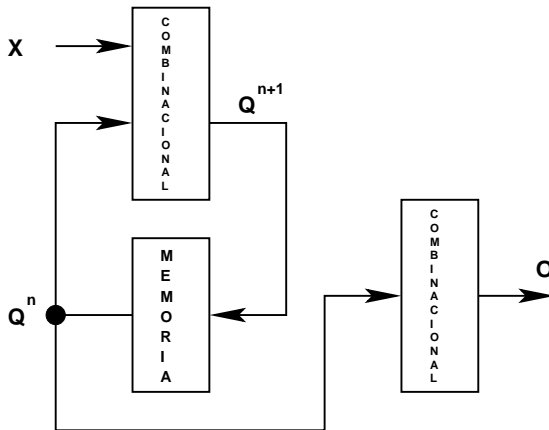
```

architecture beh of seqmly is
begin
  process (clk,x)
    type estados is (a,b,c);
    variable estado : estados;
  begin
    if clk='1' and clk'event then
      if x='0' then
        case estado is
          when a =>
            estado := a;
          when b =>
            estado := c;
          when c =>
            estado := a;
          end case;
        else
          estado := b;
        end if;
      end if;
      if x='0' then
        case estado is
          when a =>
            o <= '1';
          when others =>
            o <= '0';
          end case;
        else
          o <= '0';
        end if;
      end if;
    end process;
  
```

	X	
	0	1
q		
A	A,1	B,0
B	C,0	B,0
C	A,0	B,0

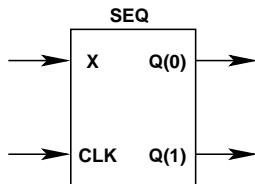
Diagrama en bloque de un circuito de Moore

Instrucciones secuenciales

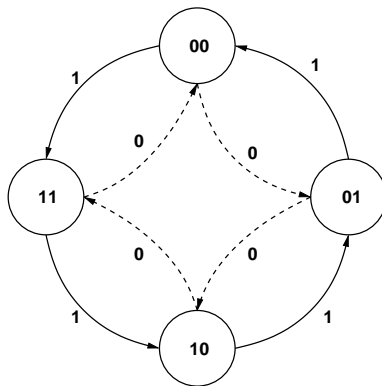


Ejemplo de un circuito de Moore

Entidad de diseño y diagrama de estados



```
entity seq is
  port (
    clk : in  bit;
    x    : in  bit;
    q    : out bit_vector(0 to 1));
end;
```



Cuerpo de arquitectura y tabla de transiciones

Circuito de Moore

```

architecture beh of seq is
begin
  process (clk)
    variable estado : bit_vector(1 downto 0);
    begin
      if clk='1' and clk'event then
        if x='0' then
          case estado is
            when "00" =>
              estado := "01";
            when "01" =>
              estado := "10";
            when "10" =>
              estado := "11";
            when "11" =>
              estado := "00";
          end case;
        else
          case estado is
            when "00" =>
              estado := "11";
            when "01" =>
              estado := "00";
            when "10" =>
              estado := "01";
            when "11" =>
              estado := "10";
          end case;
        end if;
        q <= estado;
      end if;
    end process;
end;

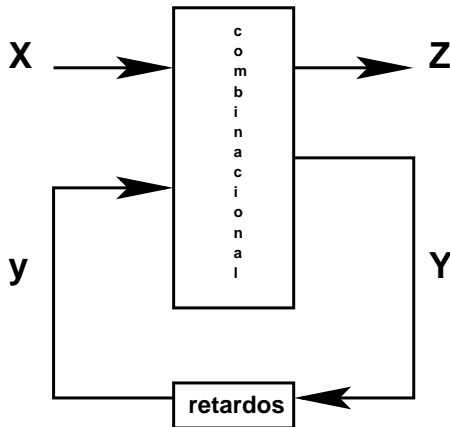
```

X

q	0	1
00	01	11
01	10	00
10	11	01
11	00	10

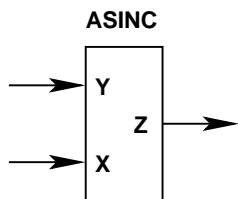
Diagrama en bloque de un circuito asincrónico

Instrucciones secuenciales



Ejemplo de un circuito asincrónico

Entidad de diseño y diagrama de estados



```
entity asinc is
  port (
    x : in  bit;
    y : in  bit;
    z : out bit);
end;
```

XY

	00	01	11	10	Z
0	a	b	a	a	0
0	a	b	c	--	0
1	d	c	c	c	1
1	d	c	--	a	1

Ejemplo de un circuito asíncrono

Cuerpo de arquitectura

```

entity asinc is
  port (
    x : in  bit;
    y : in  bit;
    z : out bit);
end;

architecture beh of asinc is
  type estados is (a,b,c,d);
  signal estado : estados;

begin
  process (x,y,estado)
  begin
    case estado is
      when a|b =>
        z <= '0';
      when c|d =>
        z <= '1';
      end case;

      case bit_vector(0 to 1)'(x&y) is
        when "00" =>
          case estado is
            when b =>
              estado <= a;
            when c =>
              estado <= d;

```

```

        when others =>
          null;
        end case;

      when "01" =>
        case estado is
          when a =>
            estado <= b;
          when d =>
            estado <= c;
          when others =>
            null;
          end case;
        when "11" =>
          case estado is
            when b =>
              estado <= c;
            when others =>
              null;
            end case;
          when "10" =>
            if estado = d then
              estado <= a;
            end if;
          end case;
        end process;
end;

```


Instrucción *loop*

■ Sintaxis

```
[ loop_label : ] [ iteration_scheme ] loop  
    sequence_of_statement  
end loop;
```

iteration_scheme ::=
 while *condition*
 | **for** *identifier in discrete_range*

- *discrete_range* indica un rango de valores. Ej : **0 to 9**.
- *identifier* es el nombre de un índice que puede referenciarse luego dentro del *loop*. Alcanza los valores especificados por *discrete_range*.

Instrucción *next*

■ Sintaxis

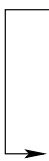
```
[ label : ] next [ loop_label ] [ when condition ] ;
```

- La instrucción *next* salta al final de la instrucción *loop* que se encuentra anidada identificada con *loop_label* y continua con la siguiente iteración si la condición *condition* vale *TRUE*.
- Si *loop_label* es omitido se continua con la siguiente iteración del loop en el que se encuentra anidado
- Si la condición es omitida entonces se salta siempre que se alcance la instrucción *next*.

Instrucción *next*

Ejemplo I

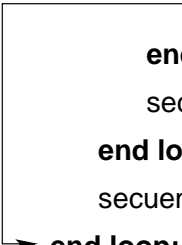
```
etiqueta1: loop  
    etiqueta2: loop  
        secuencia_de_instrucciones  
        if ..... then  
            next;  
        end if;  
        secuencia_de_instrucciones  
    end loop;  
    secuencia_de_instrucciones  
end loop;
```



Instrucción *next*

Ejemplo II

```
etiqueta1: loop  
    etiqueta2: loop  
        secuencia_de_instrucciones  
        if ..... then  
            next etiqueta1;  
        end if;  
        secuencia_de_instrucciones  
    end loop;  
    secuencia_de_instrucciones  
end loop;
```



Instrucción *exit*

■ Sintaxis


[label :] **exit** [*loop_label*] [**when** condition] ;

- La instrucción *exit* sale de la instrucción *loop* en la que se encuentra anidada e identificada con *loop_label* si la condición *condition* vale *TRUE*.
- Si *loop_label* es omitido se continua con la siguiente iteración del loop en el que se encuentra anidado
- Si la condición es omitida entonces se sale siempre que se alcance la instrucción *exit*.

Instrucción *exit*

Ejemplo I

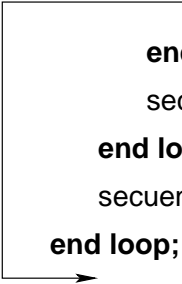
```
etiqueta1: loop  
    etiqueta2: loop  
        secuencia_de_instrucciones  
        if ..... then  
            exit;  
        end if;  
        secuencia_de_instrucciones  
    end loop;  
    secuencia_de_instrucciones  
end loop;
```



Instrucción *exit*

Ejemplo I

```
etiqueta1: loop  
    etiqueta2: loop  
        secuencia_de_instrucciones  
        if ..... then  
            exit etiqueta1;  
        end if;  
        secuencia_de_instrucciones  
    end loop;  
    secuencia_de_instrucciones  
end loop;
```



Instrucción *wait*

■ Sintaxis

[label] : **wait** [**on** sensitivity_clause] [**until** condition] [**for** *time_expression*];

- Se utiliza dentro un *process* y permite expresar comportamientos de circuitos asincronicos.

■ Función

- La descripción del circuito permanece en el mismo estado hasta que haya un cambio en cualquiera de las señales especificadas en *sensitivity_clause* y que *condition* sea *TRUE*.
- Si pasó un lapso mayor que *time_expression* esperando por un cambio y verificación de *condition* entonces se considera cumplida las condiciones anteriores y se continua con la propagación de los valores del circuito.

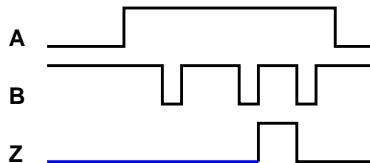
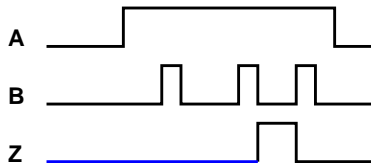
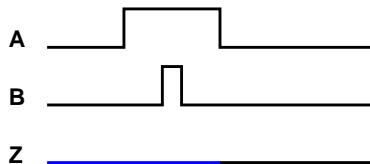
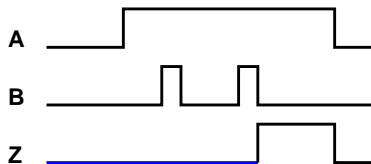
■ Ejemplos:

- **wait on** clk,rst
- **wait until** clk='1';
- **wait for** 10 ns;

Ejemplo de uso de la Instrucción *wait*

- Diseñar un circuito asincrónico de 2 entradas A y B y una salida Z.
- El circuito producirá un pulso cada vez que dentro de un pulso de A se produzcan 2 pulsos completos de B.

Diagrama de tiempos

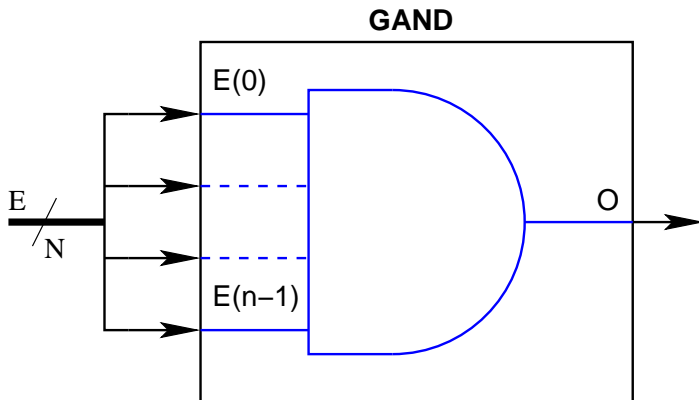


Descripción en VHDL

```
entity ejemp is
  port (
    a : in  bit;
    b : in  bit;
    z : out bit);
end;

architecture beh of ejemp is
begin
  process
  begin
    z <= '0';
    wait on a until a='1';
    for i in 0 to 4 loop
      wait on a,b;
      exit when a='0';
      z <= '0';
      if i = 3 then
        z <= '1';
      end if;
    end loop;
    if a='1' then
      wait on a;
    end if;
  end process;
end;
```

Diseño de una compuerta *and* parametrizable



Descripción de la entidad *gand*

Ejemplo I

```
entity gand is
  generic (
    n : integer);
  port (
    e : in  bit_vector (0 to n-1);
    o : out bit);
end;

architecture beh of gand is
begin
  process (e)
  begin
    o <= '1';
    for i in 0 to n-1 loop
      if e(i)='0' then
        o <= '0';
        exit;
      end if;
    end loop;
  end process;
end;
```

Descripción de la entidad *gand*

Ejemplo II

```
entity gand is
  generic (
    n : integer);
  port (
    e : in  bit_vector (0 to n-1);
    o : out bit);
end;

architecture beh of gand is
begin
  process (e)
    variable aux : bit;
  begin
    aux := e(0);
    for i in 1 to n-1 loop
      aux := aux and e(i);
    end loop;
    o <= aux;
  end process;
end;
```

Instrucciones concurrentes

- Se utilizan en la descripción de circuitos por medio de los modelos *estructural* y *data flow*.
- Su secuencia no afecta la descripción del circuito. Se utilizan para definir la interconexión de bloques. Las instrucciones concurrentes se encuentran principalmente en el bloque de instrucciones de *architecture*, *block*, *generate*.
- Las instrucciones concurrentes son las siguientes:
 - *block*
 - *process*
 - *llamada a procedimiento concurrente*
 - *asignación de señal concurrente*
 - *component*
 - *generate*

Instrucciones concurrentes

Instrucción *block*

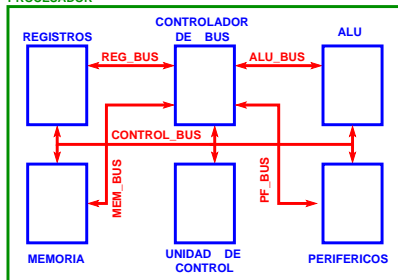
La instrucción define un bloque que representa una porción del diseño. La instrucción *block* puede anidarse para permitir la descomposición jerárquica del diseño.

Sintaxis:

```
block_statement ::=  
    block_label : block [ guard_expression ]  
        block_declarative_part  
begin  
    block_statement_part  
end ;
```


Instrucciones concurrentes

PROCESADOR



```
entity procesador is
```

```
    ....
end;
```

```
architecture estruct of procesador is
```

```
    signal pf_bus ..
    signal control_bus, mem_bus ...
    signal reg_bus, alu_bus ...
```

```
begin
```

```
    registros: block
        type ...
```

```
        signal .....
    begin

    end block;
    memoria : mem port map (.....

    unidad_de_control: block
    begin
        ....
    end block;

    perifericos: block
    begin
        ....
    end block;

    alu: block
    begin
        ....
    end block;

    controlador_de_bus : block
    begin
        ....
    end block;
end;
```

Instrucciones concurrentes

Asignación concurrente de señal

```
concurrent_signal_assignment ::=  
    [ label : ] conditional_signal_assignment  
    [ label : ] selected_signal_assignment
```

Instrucciones concurrentes

Asignación condicional de señal

Sintaxis

```
conditional_signal_assignment ::=  
    target <= options condicional_waveform ;
```

```
conditional_waveforms ::=  
    { waveform when condition else }  
    waveform
```

Instrucciones concurrentes

Asignación condicional de señal

Ejemplo

```
entity priomux is
  port (
    ena : in  bit_vector(0 to 3);
    inp  : in  bit_vector(0 to 3);
    o    : out bit);
end;

architecture df of priomux is
begin
  o <=
    inp(0) when ena(0)='1' else
    inp(1) when ena(1)='1' else
    inp(2) when ena(2)='1' else
    inp(3) when ena(3)='1' else
    '0';
end;
```

```
architecture beh of priomux is
begin
  process (ena, inp)
  begin
    if ena(0) = '1' then
      o <= inp(0);
    elsif ena(1) = '1' then
      o <= inp(1);
    elsif ena(2) = '1' then
      o <= inp(2);
    elsif ena(3) = '1' then
      o <= inp(3);
    else
      o <= '0';
    end if;
  end process;
end;
```

Instrucciones concurrentes

Asignación seleccionada de señal

Sintaxis :

```
selected_signal_assignment ::=  
    with expression select  
    target <= options selected_waveform ;
```

```
selected_waveforms ::=  
    { waveform when choices, }  
    waveform when choices
```

Instrucciones concurrentes

Asignación seleccionada de señal

Ejemplo

```
entity mux4 is
  port (
    s : in  bit_vector(1 downto 0);
    e : in  bit_vector(0 to 3);
    o : out bit);
end;

architecture df of mux4 is
begin
  with s select
    o <=
      e(0) when "00",
      e(1) when "01",
      e(2) when "10",
      e(3) when "11";
end;
```

```
architecture beh of mux4 is
begin
  process (e,s)
  begin
    case s is
      when "00" =>
        o <= e(0);
      when "01" =>
        o <= e(1);
      when "10" =>
        o <= e(2);
      when "11" =>
        o <= e(3);
    end case;
  end process;
end;
```

Instrucciones concurrentes

Equivalencia con un *process*

```
entity logic is
  port (
    a : in  bit;
    b : in  bit;
    c : in  bit;
    d : in  bit;

    s : in  bit;
    o : out bit);
end;

architecture df of logic is
begin
  o <= a and b when s='0' else
    c or d;
```

```
end;

architecture df of logic is
begin
  process (a,b,c,d,s)
  begin
    if s = '0' then
      o <= a and b;
    else
      o <= c or d;
    end if;
  end process;
end;
```

Instrucciones concurrentes

component instantiation

Crea un componente y lo conecta al circuito que se está describiendo.

```
component_instantiation_statement ::=  
    instantiation_label : component_label  
        [ generic map ( generic_association_list ) ]  
        [ port map ( port_association_list ) ] ;
```

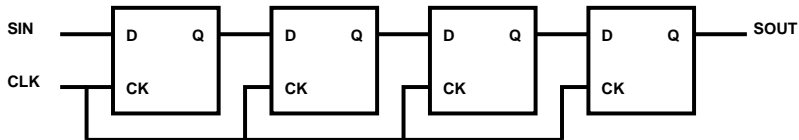
Ejemplos:

```
ffd0 : ffd  
    port map ( reset , clock , d0 , q0 );  
  
ffd1 : ffd  
    port map ( clr => reset , ck => clock , d => d0 , q => q0 );  
  
and8 : gand  
    generic map ( 8 );  
    port map ( e => inp , o => s );
```


Instrucciones concurrentes

Introducción a la instrucción *generate*

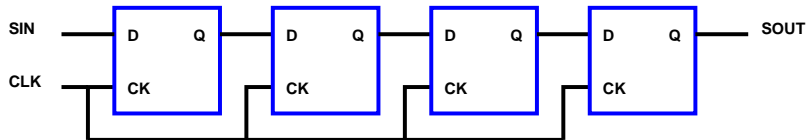
Registro de desplazamiento



Instrucciones concurrentes

Introducción a la instrucción *generate*

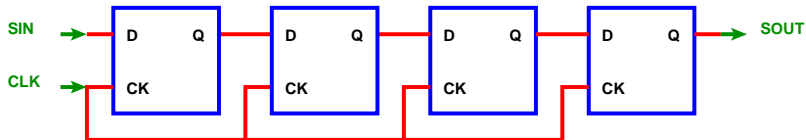
Registro de desplazamiento



Instrucciones concurrentes

Introducción a la instrucción *generate*

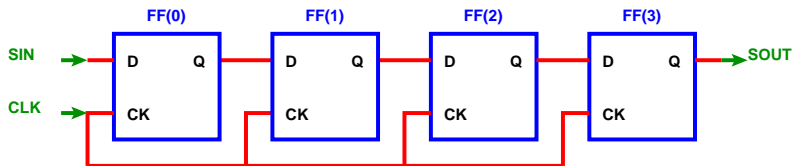
Registro de desplazamiento



Instrucciones concurrentes

Introducción a la instrucción *generate*

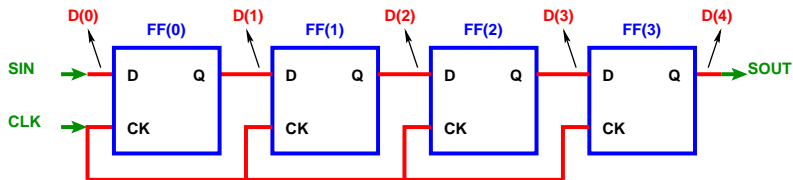
Registro de desplazamiento



Instrucciones concurrentes

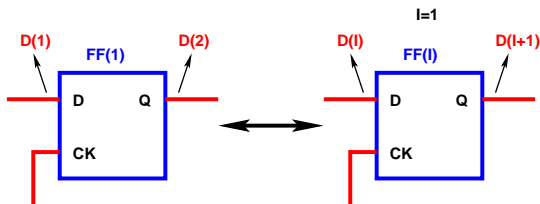
Introducción a la instrucción *generate*

Registro de desplazamiento



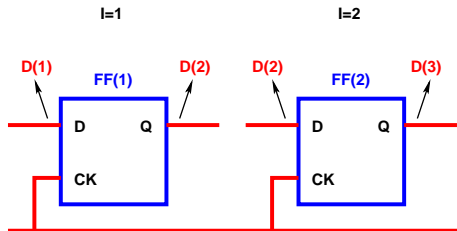
Instrucciones concurrentes

Introducción a la instrucción *generate*



Instrucciones concurrentes

Introducción a la instrucción *generate*



Instrucciones concurrentes

Ejemplo instrucción *generate*

```
entity shtreg4 is
  port (
    clk : in  bit;
    sin  : in  bit;
    sout : out bit);
end;

architecture estruc of shtreg4 is
  component ffd
    port (
      clr : in  bit := '0';
      ck  : in  bit;
      d   : in  bit;
      q   : out bit);
  end component;
  signal d : bit_vector (0 to 4);
begin
  gensht: for i in 0 to 3 generate
    ff : ffd port map (ck => clk, d => d(i), q => d(i+1));
  end generate;
  d(0) <= sin;
  sout <= d(4);
end;
```


Instrucciones concurrentes

generate

Es un mecanismo que permite la elaboración iterativa o condicional de una porción de la descripción del circuito.

Sintaxis :

```
generate_statement ::=  
    generate_label : generate_scheme generate  
        { concurrent_statement }  
    end generate ;
```

```
generation_scheme ::=  
    for generate_parameter_specification  
    | if condition
```

Instrucciones concurrentes

Ejemplo instrucción *generate*

```

entity shtreg4 is
  port (
    clk  : in  bit;
    sin  : in  bit;
    sout : out bit);
end;

architecture estruc of shtreg4 is
  component ffd
    port (
      clr : in  bit := '0';
      ck  : in  bit;
      d   : in  bit;
      q   : out bit);
  end component;
  signal d : bit_vector (1 to 3);
begin
  gensht: for i in 0 to 3 generate
    first: if i=0 generate

```

```

      ff : ffd port map (
        ck => clk, d => sin,
        q => d(i+1));
    end generate;

    middle: if i > 0 and i < 3 generate
      ff : ffd port map (
        ck => clk, d => d(i),
        q => d(i+1));
    end generate;

    last: if i=3 generate
      ff : ffd port map (
        ck => clk, d => d(i),
        q => sout);
    end generate;
  end generate;
end;

```

Instrucciones concurrentes

Ejemplo instrucción *generate*

```

entity gshtreg is
  generic (n : integer)
  port (
    clk : in bit;
    sin : in bit;
    sout : out bit);
end;

architecture estruc of gshtreg is
  component ffd
    port (
      clr : in bit := '0';
      ck : in bit;
      d : in bit;
      q : out bit);
  end component;
  signal d : bit_vector (1 to n-1);
begin
  gensht: for i in 0 to n-1 generate

    first: if i=0 and n > 1 generate
      ff : ffd port map (
        ck => clk, d => sin,
        q => d(i+1));

```

```

    end generate;

    middle: if i > 0 and i < n-1 generate
      ff : ffd port map (
        ck => clk, d => d(i),
        q => d(i+1));
    end generate;

    last: if i=n-1 and n > 1 generate
      ff : ffd port map (
        ck => clk, d => d(i),
        q => dout));
    end generate;

    onlyone: if n=1 generate
      ff : ffd port map (
        ck => clk, d => din,
        q => sout));
    end generate;

  end generate;
end;

```

Tipo de datos escalares

Enteros

Sintaxis :

type *type_identifier* **is range_constraint** ;

range_constraint ::= **range** *range*

range ::= *simple_expression* [**to** | **downto**] *simple_expression*

Ejemplo :

type *twos_complement_integer* **is range** **-32768 to 32767**;

type *byte_length_integer* **is range** **0 to 255**;

type *word_index* **is range** **31 downto 0**;

Tipo de datos escalares

Punto flotante

Sintaxis :

type *type_identifier* **is** range_constraint ;

range_constraint ::= **range** range

range ::= simple_expression [**to** | **downto**] simple_expression

Ejemplo :

type real **is** -1.0e38 **to** 1.0e38

Tipo de datos compuestos

Arreglos con contorno

Sintaxis :

```
type type_identifier is constrained_array_definition ;
```

```
constrained_array_definition ::=  
    array index_constraint of element_subtype_indication ;
```

```
index_constraint ::= ( discrete_range [ , discrete_range ] )
```

Ejemplo :

```
type byte is array (7 downto 0) of bit;
```

```
type mem32kb is array (0 to 32767) of bit_vector(7 to 0);
```

```
type mat is array (0 to 7, 5 to 10) of bit;
```

Tipo de datos compuestos

Arreglos sin contornos

Sintaxis :

type *type_identifier* **is** *unconstrained_array_definition* ;

unconstrained_array_definition ::=

array (*index_subtype_definition* { , *index_subtype_definition* })
of *element_subtype_indication*

index_subtype_definition ::= *type_mark* **range** <>

Ejemplo :

type *bit_vector* **is** **array** (*natural* **range** <>) **of** *bit*;

type *mem8* **is** **array** (*natural* **range** <>) **of** *byte*;

type *mem32* **is** **array** (*natural* **range** <>) **of** *bit_vector*(0 **to** 31);

Tipo de datos compuestos

Acceso a arreglos

```
architecture ...
```

```
signal a : bit;  
signal b : byte;  
signal c : mat;  
signal d : mem32kb;
```

```
begin
```

```
— Se copia el bit 0 de "b" a "a"
```

```
a    <= b(0);
```

```
— Se copia el bit mat(1,9) al bit 5 de "b"
```

```
b(0) <= mat(1,9);
```

```
— Se copia el byte "b" al byte 0 de "d"
```

```
d(0) <= b;
```

```
— Se copia el bit 1 del byte d(0) a "a"
```

```
a    <= d(0)(1);
```

```
end ;
```


Tipo de datos compuestos

Acceso a *slice*

Un *slice* es un arreglo unidimensional compuesto de una secuencia de elementos de otro arreglo.

Ejemplo

```
architecture ...
```

```
signal b : byte;  
signal e : mem32(0 to 15);
```

```
begin
```

```
— Se copia los 4 bits superiores a los inferiores  
b(3 downto 0) <= b(7 downto 3);
```

```
— Se copia los 4 bits del medio de "b" a  
— los bits de 8 a 11 de la palabra 0 de "d"  
e(0)(8 to 11) <= b(6 downto 3);
```

```
end ;
```

Tipo de datos compuestos

record

Sintaxis :

type *type_identifier* **is** *record_type_definition* ;

record_type_definition ::=

record

element_declaration

{ *element_declaration* }

end record

element_declaration ::=

identifier_list : *element_subtype_definition*

identifier_list ::= *identifier* { , *identifier* }

element_subtype_definition ::= *subtype_indication*

Ejemplo :

type *date* **is**

record

day : integer **range** 1 **to** 31;

month : month_name;

year : integer **range** 0 **to** 4000;

end record;

Tipo de dato discretos

Físico

Sintaxis :

type *type_identifier* **is** *physical_type_definition* ;

physical_type_definition ::=
 range_constraint
 units
 base_unit_declaration
 { *secondary_unit_declaration* }
 end units

base_unit_declaration ::= *identifier* ;

secondary_unit_declaration ::= *identifier* = *physical_literal* ;

physical_literal ::= [*abstract_literal*] *unit_name*

Tipo de dato discretos

Ejemplo de declaración de tipos físicos

```

type time is range -1e18 to 1e18
units
    fs;           — femtosegundo
    ps = 1000 fs; — picosegundo
    ns = 1000 ps; — nanosegundo
    us = 1000 ns; — microsegundo
    ms = 1000 us; — milisegundo
    sec = 1000 ms; — segundo
    min = 60 sec;  — minuto
end units;

type distance is range 0 to 1e16
units
    A;           — angstron
    — unidades metricas
    nm = 10 A;    — nanometer
    um = 1000 nm; — micrometer
    mm = 1000 um; — millimeter
    cm = 10 mm;   — icentimeter
    m = 10 A;     — meter
    km = 10 A;    — kilometer
    — unidades inglesas
    mil = 254000 A; — mil
    inch = 1000 mil; — pulgada
    ft = 12 inch;   — pie
    yd = 3 ft;      — yarda
    fm = 6 ft;      — fathom
    mi = 5280 ft;   — milla
    lg = 3 mi;      — legua
end units;

```

Tipo de dato discretos

Ejemplo de uso de tipos físicos

```
....  
    variable x : distance;  
    variable x1 : distance;  
    variable y : time;  
    variable z : integer;  
begin  
    — Esto se puede hacer  
    x := 5 A + 13 ft -25 inch;  
    y := 3 ns + 5 min;  
    z := ns / ps;  
    x := z * mi;  
    y := y / 10;  
    z := x / x1;  
  
    — Esto no se puede hacer sin  
    — sobre cargar operadores  
    — x*x  
    — x/y  
end;
```

Subtipos

subtype_declaration ::=
 subtype identifier **is** subtype_indication ;

subtype_indication ::=
 [*resolution_function_name* type_mark [constraint]

type_mark ::=
 type_name
 subtype_name

constraint ::=
 range_constraint
 index_constraint

Expresiones

Operadores ordenados por precedencia descendente

misc	::=	** abs not
multiplying_operator	::=	* / mod rem
sign	::=	+ -
adding_operator	::=	+ - &
relational_operator	::=	= / = < <= > >=
logic_operator	::=	and or nand nor xor xnor

Expresiones

Sintaxis

```
expression ::=
    relation { and relation }
    | relation { or relation }
    | relation { xor relation }
    | relation [ nand relation ]
    | relation [ nor relation ]
    | relation { xnor relation }
```

```
relation ::=
    simple_expression
    [ relational_operator simple_expression ]
```

```
relational_operator ::= = | /= | < | <= | > | >=
```

```
simple_expression ::=
    [ sign ] term { adding_operator term }
```

```
adding_operator ::= + | - | &
```

```
sign ::= + | -
```

```
term ::=
    factor { multiplying_operator factor }
```

```
multiplying_operator ::= * | / | mod | rem
```

```
factor ::=
    primary [ ** primary ]
    | abs primary
    | not primary
```

```
primary ::=
    name
    | literal
    | aggregate
    | function_call
    | qualified_expression
    | type_conversion
    | ( expression )
```


Subprograma

Sintaxis

```
subprogram_declaration ::=  
    subprogram_specification ;
```

```
subprogram_specification ::=  
    procedure designator [ ( formal_paramter_list ) ]  
    function designator [ ( formal_paramter_list ) ] return type_mark;
```

```
subprogram_specification ::=  
    subprogram_specification is  
        subprogram_declarative_part  
    begin  
        subprogram_statement_part  
    end ;
```

```
formal_parameter_list ::= parameter_interface_list
```

Subprograma

Ejemplo de una función

```
entity muxn is
  generic (n : integer);
  port (
    s : in  bit_vector (n-1 downto 0);
    e : in  bit_vector (2**n-1 downto 0);
    o : out bit);
end;

architecture df of muxn is
  function conv_integer (arg : bit_vector) return integer is
    variable aux : integer;
  begin
    aux := 0;
    for i in arg'range loop
      aux := aux*2;
      if arg(i)='1' then
        aux := aux+1;
      end if;
    end loop;
  end;
begin
  o <= e(conv_integer(s));
end;
```

Subprograma

Ejemplo de una función

```
package ffd_pack is
  function f_ffd(signal clr : bit; signal ck : bit; d : bit) return bit;

  procedure p_ffd(signal clr : bit; signal ck : bit; d : bit; signal q : out bit);
end;

package body ffd_pack is
  function f_ffd(signal clr : bit; signal ck : bit; d : bit) return bit is
  begin
    if clr='1' then
      return '0';
    elsif ck='1' and ck'event then
      return d;
    end if;
  end;

  procedure p_ffd(signal clr : bit; signal ck : bit; d : bit; signal q : out bit) is
  begin
    q <= f_ffd (clr,ck,d);
  end;
end;
```

Definición de parámetros

Sintaxis

```
interface_list ::=  
    interface_element { ; interface_element }
```

```
interface_element ::= interface_declaration
```

```
interface_declaration ::=  
    interface_constant_declaration  
    | interface_signal_declaration  
    | interface_variable_declaration
```

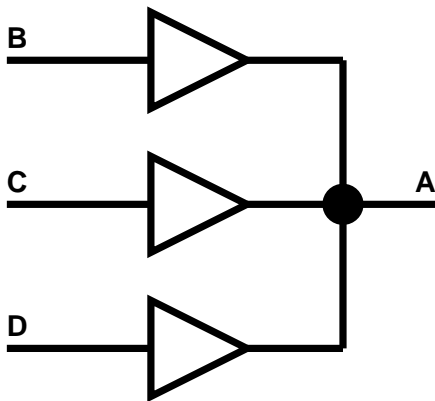
```
interface_constant_declaration ::=  
    [ constant ] identifier_list : [ in ] subtype_indication [ := static_expression ]
```

```
interface_signal_declaration ::=  
    [ signal ] identifier_list : [ mode ] subtype_indication [ := static_expression ]
```

```
interface_variable_declaration ::=  
    [ variable ] identifier_list : [ mode ] subtype_indication [ := static_expression ]
```

```
mode ::= in | out | inout | buffer | linkage
```

Función de resolución



Función de resolución

```
entity wand is
  port (
    b : in  bit;
    c : in  bit;
    d : in  bit;
    a : out bit);
end;

architecture mix of wand is

  function reswand (arg : bit_vector) return bit is
    variable aux : bit;
  begin
    for i in arg'range loop
      if arg(i) = '0' then
        return '0';
      end if;
    end loop;
    return '1';
  end;

  signal a : reswand bit;

begin
  a <= b;
  a <= c;
  a <= d;
end;
```

Packages

Ejemplo

```
package mio is
  type mibit ('u', 'x', '0', '1', 'z');
  type mibit_vector is array (integer range <>) of bit;
  function conv_integer (arg : mibit_vector) return integer;
  component ffd (
    port (clr : in bit; ck : in bit; d : in bit; q : out bit);
  end component;
end;

package body mio is
  function conv_integer (arg : mibit_vector) return integer is
    variable aux : integer;
  begin
    aux := 0;
    for i in arg'range loop
      aux := aux*2;
      if arg(i)='1' then
        aux := aux+1;
      end if;
    end loop;
  end;
end;
```

Packages

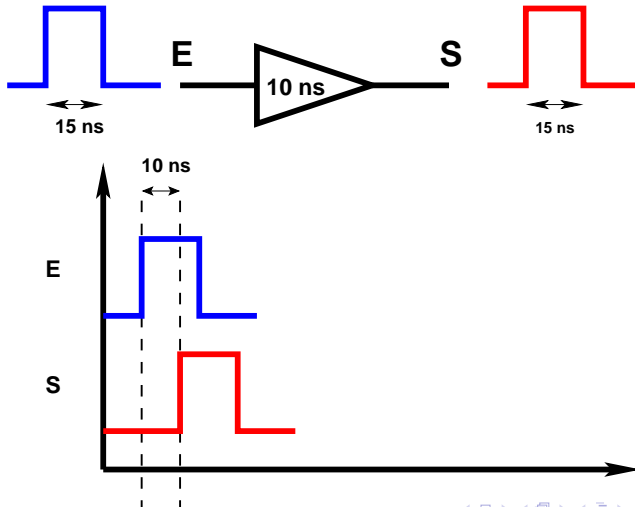
Sintaxis

```
package_declaration ::=  
    package identifier is  
        package_declarative_part  
    end ;
```

```
package_body ::=  
    package body identifier is  
        package_body_declarative_part  
    end ;
```

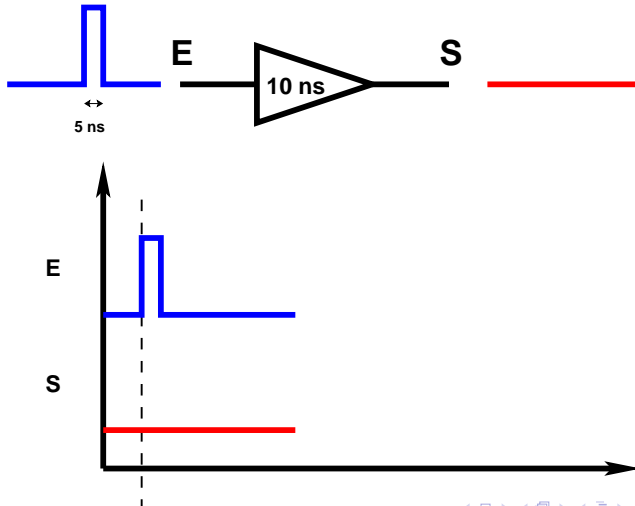

Retardo inercial

Ancho de pulso mayor que el retrado de compuerta



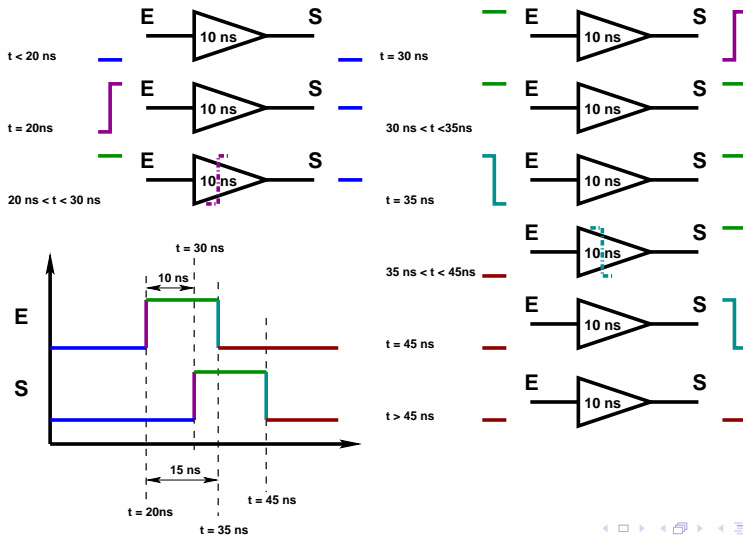
Retardo inercial

Ancho de pulso menor que el retrado de compuerta



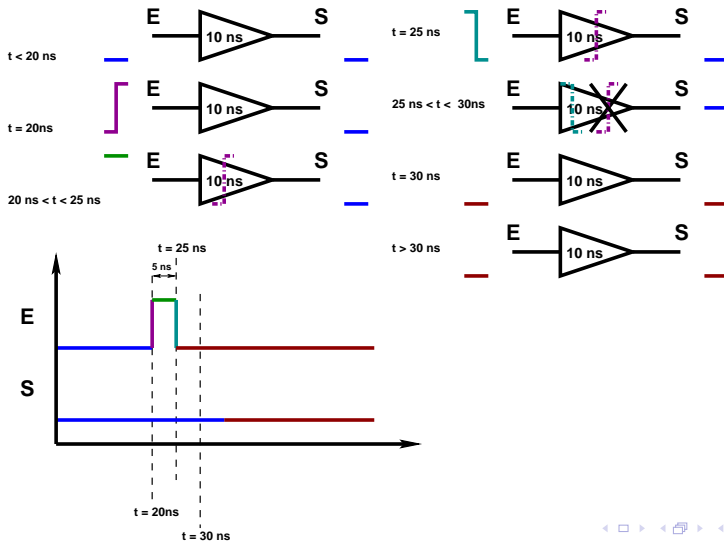
Retardo inercial

Evento por evento, pulso mayor que el retardo de compuerta



Retardo inercial

Evento por evento, pulso menor que el retardo de compuerta

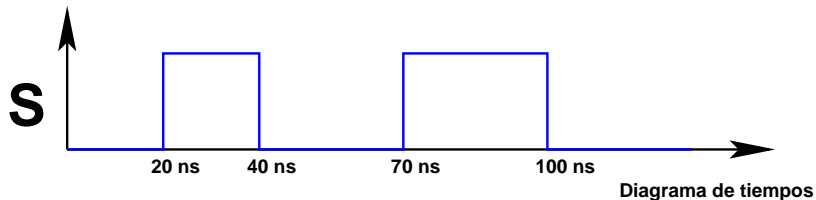


Retardo inercial

En el retardo inercial :

- La compuerta no puede almacenar más que un cambio.
- Todo cambio nuevo en la entrada borra cualquier cambio anterior.

Forma de onda



Elemento de forma de onda
o waveform element

`S <= '0', '1' after 20 ns , '0' after 40 ns , '1' after 70 ns , '0' after 100 ns ;`

Forma de onda o "waveform"

- Los tiempos son relativos al momento en que produce dicha asignación.
- Estos tiempos no son relativos entre si y deben tener valores crecientes.

Asignación inercial

Prueba con pulso de 15 ns

```
entity buf is
  port (
    e : in  bit;
    s : out bit);
end;

architecture beh of buf is
begin
  process (e)
  begin
    s <= e after 10 ns;
  end process;
end;

entity pulse15ns is
end;

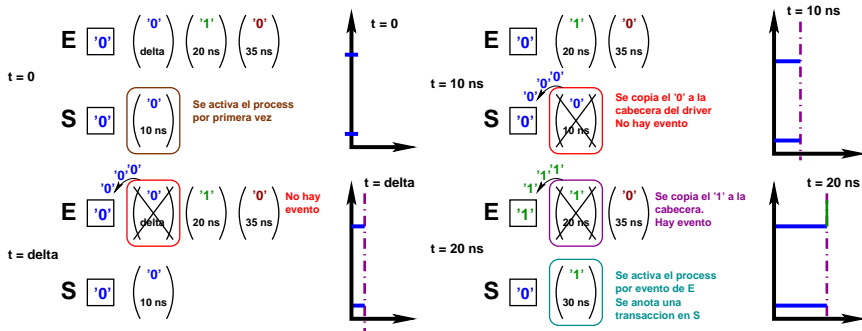
architecture mix of pulse15 is
  signal e : bit;
  signal s : bit;

  component buf
    port (
      e : in  bit;
      s : out bit);
  end component;

begin
  e <= '0', '1' after 20 ns, '0' after 35 ns;
  test : buf port map (e, s);
end;
```

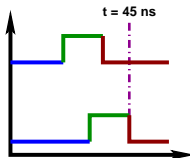
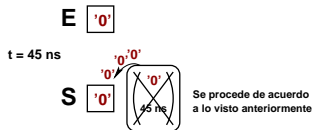
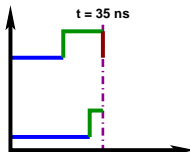
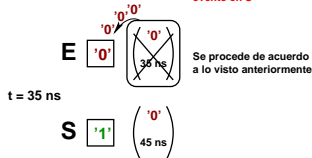
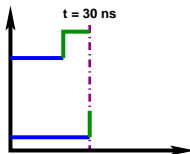
Retardo inercial

Evento por evento, pulso mayor que el retardo de compuerta



Retardo inercial

Evento por evento, pulso mayor que el retardo de compuerta



Asignación inercial

Prueba con pulso de 5 ns

```
entity buf is
  port (
    e : in  bit;
    s : out bit);
end;

architecture beh of buf is
begin
  process (e)
  begin
    s <= e after 10 ns;
  end process;
end;

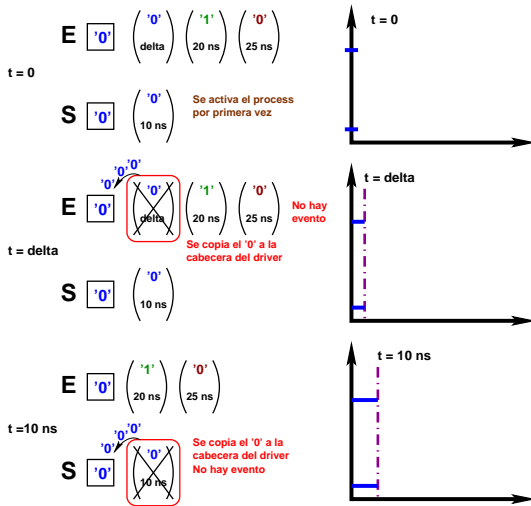
entity pulse5ns is
end;

architecture mix of pulse5 is
  signal e : bit;
  signal s : bit;

  component buf
    port (
      e : in  bit;
      s : out bit);
  end component;
begin
  e <= '0', '1' after 20 ns, '0' after 25 ns;
  test : buf port map (e, s);
end;
```

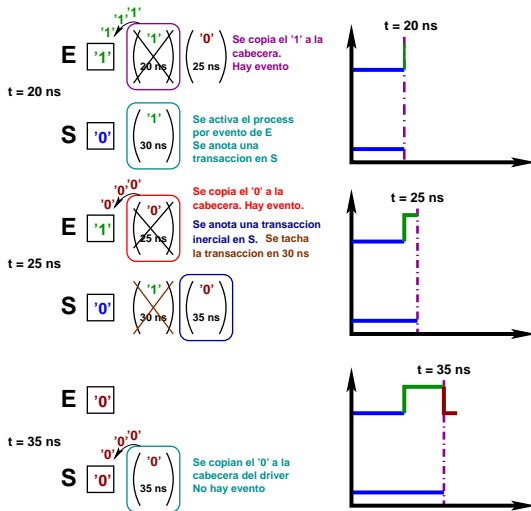
Retardo inercial

Evento por evento, pulso menor que el retardo de compuerta



Retardo inercial

Evento por evento, pulso menor que el retardo de compuerta



Asignación de señales

Composición de las señales

- Las señales en VHDL esta formadas por uno o más drivers.
- Cada driver está compuesto por:
 - Una cabecera con el valor actual.
 - Una cola de transacciones con los valores a propagar.
 - Las transacciones se corresponden con los *elementos de la forma de onda* que se encuentran descriptas en la asignación de señal.
 - Una transacción genera *evento* cuando la transacción cambia el valor de la cabecera.

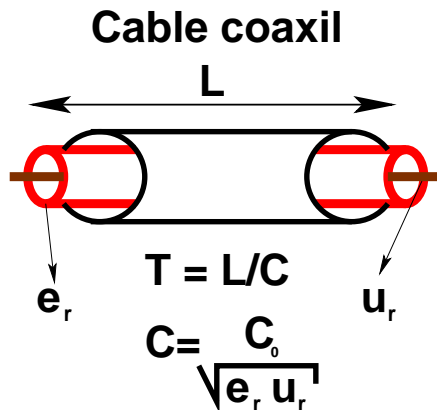
Asignación inercial

Regla a aplicar

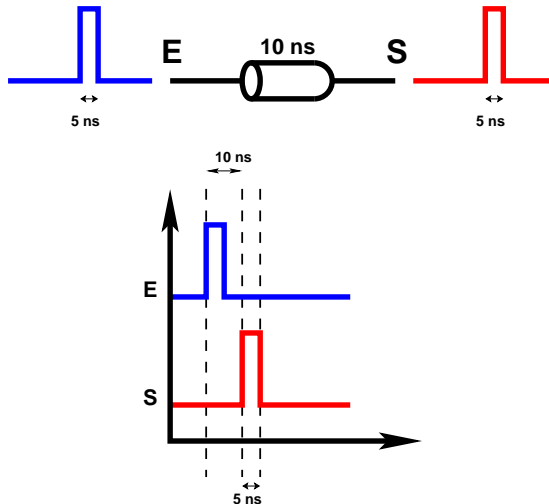
- Se buscan todas las transacciones que se encuentran en la cola del *driver* a partir del tiempo correspondiente de la nueva transacción hacia los tiempos decrecientes.
- Todas aquellas transacciones que a partir de ahí tengan el mismo valor se conservan.
- Las otras transacciones, a partir de la primera cuyo valor difiera respecto del valor de la primer transacción nueva que se va anotar, se anulan.
- Se anulan todas las transacciones que se encuentran en la cola del *driver* que tengan tiempos mayores o iguales que las transacciones que se anotan en el momento de la asignación.

Línea de transmisión

En una línea de transmisión, la señal que ingresa sale aproximadamente igual independientemente del retardo que posea.



Pulso en una linea de transmisión



Asignación de linea de transmisión

Descripción en VHDL

```
entity coaxil is
  port (
    e : in  bit;
    s : out bit);
end;

architecture beh of coaxil is
begin
  process (e)
  begin
    s <= transport e after 10 ns;
  end process;
end;

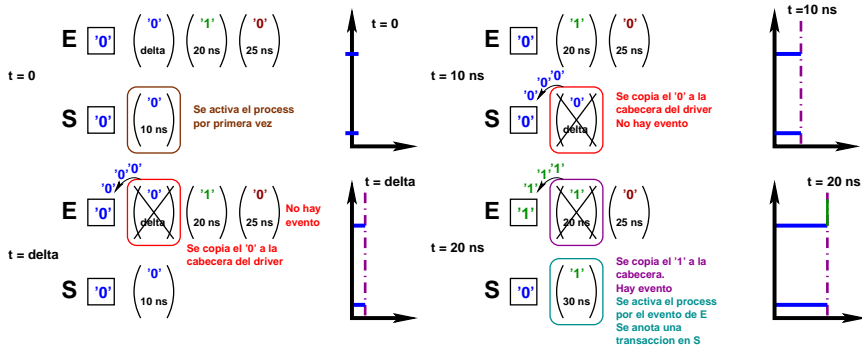
entity pulse5ns is
end;

architecture mix of pulse5 is
  signal e : bit;
  signal s : bit;

  component buf
    port (
      e : in  bit;
      s : out bit);
  end component;
begin
  e <= '0', '1' after 20 ns, '0' after 25 ns;
  test : buf port map (e, s);
end;
```

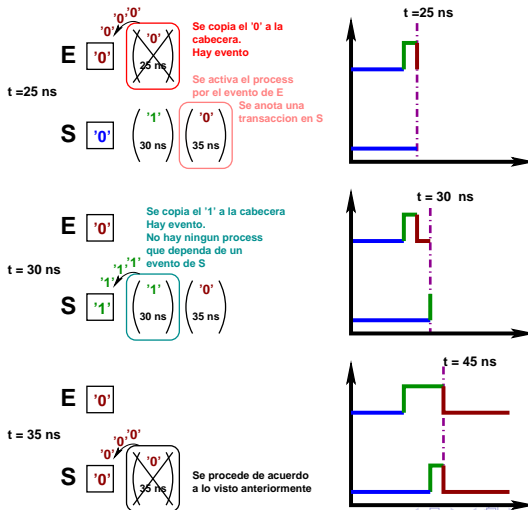
Pulso en una linea de transmisión

Evento por evento



Pulso en una linea de transmisión

Evento por evento



Asignación de línea de transmisión

Regla a aplicar

- Se anulan todas las transacciones que se encuentran en la cola del *driver* que tengan tiempos mayores o iguales que las transacciones que se anotan en el momento de la asignación.
- No se modifica ninguna otra transacción.

Asignación secuencial de señales

Sintaxis:

```
signal_assignment_statement ::=
    [ label : ] target <= [ delay_mechanism ] waveform ;
```

```
delay_mechanism ::=
    transport
    | inertial
```

```
target ::=
    name
    | aggregate
```

```
waveform ::=
    waveform_element { , waveform_element }
```

```
waveform_element ::=
    expression [ after time_expression ]
```