TUTORIAL DEL LENGUAJE VHDL

Ing. Daniel Francisco Gómez Prado
dgomezp@unmsm.edu.pe

Profesor de la Facultad de Ingeniería Electrónica, Universidad Nacional Mayor de San Marcos
Lima – Perú

RESUMEN: El presente artículo pretende difundir los conocimientos básicos del lenguaje VHDL (Very High Speed Integrated Circuit Hardware Description Language) diseñado para la descripción y síntesis de sistemas digitales para su implementación en PLD. Se reduce los circuitos lógicos complejos, máquinas de estados e incluso diagrama de flujos a un código sencillo y legible que puede también ser compilado y utilizado como librería para cualquier otro proyecto.

ABSTRACT: This paper tries to diffuse the basic knowledge of the language VHDL (Very High Speed Integrated Circuit Hardware Description Language) designed for the description and synthesis of digital systems. It decreases the complex logical circuits, machines of states and diagram of flows to a simple and readable code that can also be compiled and used as bookstore for any other project.

Palabras Claves: VHDL, síntesis digital, PLD

I. ESTRUCTURA DEL LENGUAJE VHDL

1.1 Unidades de Diseño en VHDL

El lenguaje VHDL está estructurado en las siguientes unidades: Entidad, Arquitectura de una Entidad, Configuración, Declaración de Paquete y Cuerpo del Paquete. Las tres primeras son básicas para la realización del diseño y las dos últimas son utilizadas cuando se desean generar librerías.

El diseño con VHDL, se define en dos partes: la unidad Entidad donde se define la interface exterior del diseño a manera de encapsulado y la unidad de Arquitectura donde se describe el funcionamiento interno de dicho diseño. Además, VHDL permite definir múltiples Arquitecturas asociadas a una única Entidad y el modelo a simular se especifica en la unidad de Configuración indicando que dicha Arquitectura se utiliza para implementar una Entidad.

Las unidades de Paquete se utilizan cuando uno de nuestros diseños es parte de otros, para ello lo empaquetamos como un solo objeto para que otros programas puedan utilizarlo directamente.

1.2 Entidad

En la declaración de Entidad se define el diseño como si fuera un producto encapsulado, indicando el número de pines, los puertos de entrada y salida. La Entidad puede definir bien las entradas y salidas de un circuito integrado por diseñar o puede definir la interface de un módulo que será utilizado en un diseño más grande.

La Entidad es la estructura que declara la interface del sistema y permite ver el diseño como una caja negra, con la cual se puede realizar diseños jerárquicos en VHDL y formar una colección de módulos interconectados entre sí. En VHDL estos módulos se definen mediante la palabra clave ENTITY cuya forma general es:

```
ENTITY nombre IS
        [GENERIC (lista de parámetros)];
        [PORT (lista de puertos)];
        [declaraciones]
[BEGIN
        sentencias]
END [ENTITY] [nombre];
```

La instrucción GENERIC, sirve para definir y declarar propiedades ó constantes generales tales como los tiempos de retardo.

La instrucción PORT, define los puertos del módulo que esta siendo definido en un lista que consiste en un nombre seguido por el modo del puerto (IN, OUT, etc.) y el tipo de datos de la línea (std_logic,bit, etc). Si no se especifica el modo del puerto, el compilador de VHDL supone que se trata del modo IN por defecto

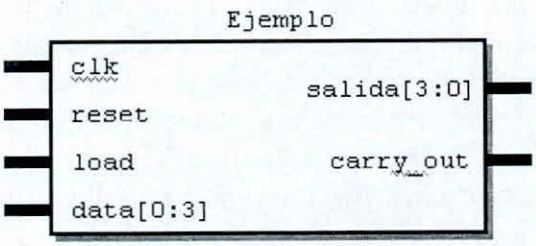

Figura 1 - Contador módulo 4.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Ejemplo IS
    GENERIC (
    retardo:    time:= 15 ns;
    max_freq: frecuency:= 50 MHz  );
 PORT (
    clk, reset: IN    std_logic;
    load:       IN    std_logic_vector(0 DOWNTO 0);
    data:       IN    std_logic_vector(0 TO 3);
    salida:     OUT std_logic_vector(3 DOWNTO 0);
    carry_out: OUT std_logic);
END Ejemplo;
```

Observar que:

- Los comentarios en VHDL se indican con un doble guión `'--'`.
- El lenguaje VHDL no distingue las letras mayúsculas de las minúsculas, por lo que un puerto llamado `data` será equivalente a otro llamado DATA ó Data.
- Por convención todas las palabras reservadas de VHDL se escriben en letras mayúsculas.
- El primer carácter de un puerto sólo puede ser una letra, nunca un número. Así mismo su nombre no puede contener caracteres especiales tales como $, %, ^, @, ... ni dos caracteres de subrayado seguidos.

1.2.1 Tipos de Puertos

Los puertos en VHDL son análogos a los pines de conexión de un símbolo esquemático y toda señal declarada en la Entidad debe tener algún tipo de puerto asignado. De esta manera un puerto es un objeto de información utilizado entre el diseño y otros circuitos digitales ó como referencia a los pines de nuestro encapsulado.

Los diferentes tipos de puertos describen la dirección por donde la información es transmitida y sólo existen los siguientes tipos:

- **IN:** Las señales en este tipo de puerto solamente pueden entrar a la Entidad, se les puede leer pero no se le puede asignar ningún valor, es decir, no se puede cambiar su valor en el programa.

- **OUT:** En este tipo de puerto la señales solamente pueden salir de la Entidad. La señal de salida puede cambiar y se le puede asignar valores, pero no se puede leer. Esto es, no se permite que su valor sea usado internamente en el diseño por que el estado lógico en el que se encuentra no se puede leer.

- **INOUT:** Este tipo de puerto es usado para implementar señales bidireccionales, es decir, para permitir que por un mismo puerto la información fluya tanto hacia dentro como hacia afuera de la entidad.

- **BUFFER:** Es equivalente a un puerto OUT al que se le a añadido un registro, pudiendo ser leído y usado como una realimentación interna. Este modo de puerto sólo puede ser conectado directamente a una señal interna ó a un puerto a modo de buffer de otra entidad.

- **LINKAGE:** Este último tipo es como el INOUT, pero sólo puede ser usado con elementos de tipo LINKAGE. En general se utiliza como interfase para enlazar el modelo diseñado con otros módulos también diseñados con herramientas distintas al VHDL.

Si en la declaración de un puerto no se especifica ningún tipo de puerto, se asume que es del tipo IN.

1.2.2 Tipos de Datos Asociados

Aunque VHDL sólo admite los cinco tipos de puertos mencionados, el tipo de dato asociado a un puerto puede ser tan variado como se desee, por que uno mismo lo puede definir. VHDL incorpora algunos tipos de datos básicos definidos en la norma IEEE 1076/93, como son:

- **Bolean:** Puede tomar los valores de verdadero ó falso.
- **Bit:** Puede tomar los valores de 0 ó 1.
- **Bit_vector:** Es un grupo de bits, donde cada uno puede tomar el valor de 0 ó 1. El orden del bit más significativo que integra el vector, se define según el uso, en este caso se utiliza la palabra reservada **DOWNTO** ó **TO**, así por ejemplo:

```
SIGNAL a: bit_vector(0 TO 3);
SIGNAL b: bit_vector(3 DOWNTO 0);
a <= "0101";
b <= "1010";

Significa que :

a(0)='0';a(1)='1';a(2)='0';a(3)='1'; —MSB=a(0),LSB=a(3)
b(0)='1';b(1)='0';b(2)='1';b(3)='0'; —MSB=b(3),LSB=a(0)
```

- **Integer:** Este tipo resulta útil para manejar índices dentro de lazos, constantes ó parámetros genéricos como tiempos de retardo. Se debe hacer notar que este tipo fue creado para la simulación y no siempre resultará sintetizable en un circuito digital.

- **Enumerated:** Con este tipo el usuario puede definir nuevos tipos de valores, resultando útil para la codificación de los estados en una máquina de Mealy ó Moore.

```
TYPE estado IS (Inicio, Lento, Rápido, Fin);
```

Debido a la necesidad de mejorar la simulación de las señales eléctricas, se amplió el tipo de dato bit mediante la norma IEEE 1164, en donde se define el tipo lógico estándar.

- **Std_ulogic** El cual puede tomar los siguientes valores

```
'U',    — Sin inicializar
'X',    — Forzado a desconocido
'0',    — Forzado a cero
'1',    — Forzado a uno
'Z',    — Alta Impedancia
'W'     — Desconocido débil
'L',    — Cero débil
'H',    — Uno débil
'-',    — No Importa
```

a partir de este tipo std_ulogic se derivaron los siguientes tipos:

- **Std_ulogic_vector:** Es un grupo de std_ulogic.
- **Std_logic:** Es una versión de std_ulogic, el cual posee una función de resolución que define lo que sucede cuando una señal es manejada por múltiples fuentes.
- **Std_logic_vector:** Es un grupo de std_logic_vector

El tipo std_logic y std_logic_vector son en la actualidad el tipo lógico estándar de la industria de semiconductores y aunque todos sus valores son validos en cualquier simulador VHDL, sólo los siguientes valores son reconocidos para síntesis lógica:

```
'0',    — Forzado a cero
'1',    — Forzado a uno
'Z',    — Alta Impedancia
```

'L', — Cero débil
'H', — Uno débil
'-', — No Importa

1.3 Arquitectura

En la Arquitectura se describe el funcionamiento del módulo definido en la Entidad; se establece el diseño real del sistema digital, indicando que hacer con cada entrada para obtener la salida. Si la Entidad es vista como una caja negra, para la cual lo único importante son las entradas y las salidas, entonces, la Arquitectura es el conjunto de detalles interiores de la caja negra. La Declaración de la Arquitectura de una Entidad en VHDL tiene la siguiente estructura:

```
ARCHITECTURE nombre OF nombre_de_entidad IS
        [declaraciones]
BEGIN
        [instrucciones]
END [ARCHITECTURE] [nombre];
```

Entre los bloques ARCHITECTURE...IS y BEGIN se definen todas las señales que se usarán internamente en el diseño. Aquí también se definen los subprogramas, funciones y constantes a utilizar dentro de la arquitectura, en la sección de instrucciones, que es donde se describe propiamente la funcionalidad del dispositivo.

1.3.1 Descripción de Comportamiento

Este tipo de descripción se caracteriza por el alto nivel de abstracción, el cual utiliza instrucciones y órdenes típicas de un lenguaje de programación (IF, THEN, ELSE, WHEN, etc). En este caso, la distribución de las puertas lógicas dentro del PLD no es nuestra principal preocupación, sino el de tener un código legible que describa al sistema tal como funciona en un diagrama de flujo. Las principales ventajas de este tipo de descripción son:

- Se incrementa la portabilidad del diseño, debido a que su descripción no depende de librerías con componentes especializados.
- El código es legible y los errores son fáciles de depurar.

Veamos un ejemplo de este tipo de arquitectura describiendo el comportamiento de la compuerta lógica NOR que se muestra en la figura 2.

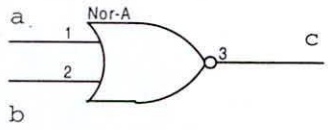

Figura 2. Compuerta NOR

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY NotOr IS PORT (
        a:      IN      std_logic;
        b:      IN      std_logic;
        c:      OUT     std_logic);
END NotOr;

ARCHITECTURE descripcion OF NotOr IS
— aquí se define cualquier señal interna
BEGIN
        c <= NOT(a OR b);
END descripcion;
```

1.3.2 Descripción de Estructura

Este tipo de descripción es más parecido a un circuito esquemático con sus conexiones de bloques a las que se les denomina componentes, los cuales son conectados y evaluados instantáneamente por medio de señales. Las principales ventajas de este tipo de descripción son:

- Permite que los grandes proyectos sean descompuestos en unidades funcionales simples de codificar, simular y corregir. Esta característica permite diseñar un sistema digital a partir de las diferentes partes que lo constituyen y especifica la conexión entre estas, donde cada unidad funcional puede ser descrita especificando su comportamiento y en el caso de unidades funcionales complejas, puedan estar a su vez descrita en forma estructural como subunidades funcionales.

- Al establecerse un sistema jerárquico de unidades funcionales, se orienta el trabajo al uso de librerías, lo que permite volver a utilizar el código elaborado (las unidades funcionales) en otros proyectos según se necesite.

Así por ejemplo se implementa un latch RS utilizando la compuerta NOR descrita como la mostrada en la

Fig. 3.

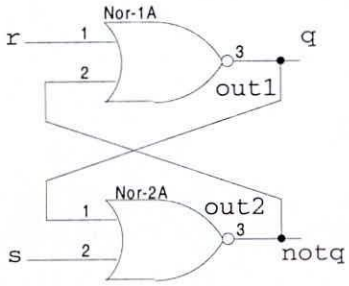

Figura 3. Latch RS

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Latch_RS IS PORT (
    r:   IN   std_logic;
    s:   IN   std_logic;
    q:   OUT std_logic;                                notq: OUT
std_logic);
END ejemplo;

ARCHITECTURE estructura OF Latch_RS IS
   COMPONENT NotOr  PORT (
        a:   IN     std_logic;
        b:   IN     std_logic;
        c:   OUT   std_logic);
    END COMPONENT;
    SIGNAL  out1:  std_logic;
    SIGNAL  out2:  std_logic;
BEGIN
Nor1:   NotOr
        PORT MAP (     a       =>      r,
                       b       =>      out2,
                       c       =>      out1);
Nor2:   NotOr
        PORT MAP (     a       =>      s,
                       b       =>      out1,
                       c       =>      out2);
q       <=      out1;
notq    <=      out2;
END Latch_RS;
```

Observar que:

El motivo por el cual no se asigna directamente los puertos de salida q y notq a los componentes, es que estos son de salida, no pudiéndo ser utilizados como entradas ya que no pueden ser leídos.

1.4 Configuración

En la sección de Configuración es donde se le indica al compilador de VHDL que una Entidad está unida a cierta Arquitectura y que ambos se unen para formar un solo diseño. La sección de Configuración es el único objeto en VHDL que puede ser simulado ó sintetizado y aunque en el caso de simulación se puede controlar el proceso de configuración manualmente cuando se realiza la síntesis el compilador se utilizará para la unión del diseño las reglas establecidas en la sección de configuración. La estructura de la sección de configuración es:

```
CONFIGURATION nombre OF nombre_de_entidad IS
   FOR nombre_de_arquitectura
         [reglas de configuración]
   END FOR;
END [CONFIGURATION] [nombre];
```

Así para el caso de la compuerta lógica NotOr la sección de configuración será:

```
CONFIGURATION config_NotOr OF NotOr IS
   FOR descripcion
   END FOR;
END config_NotOr;
```

Como en este diseño no se declararon componentes internos, no se poseen reglas de Configuración. Esta es la Configuración por defecto que el compilador VHDL asignará al diseño, por lo que puede ser omitida. Para el caso del latch RS la configuración será:

```
CONFIGURATION config_Latch_RS OF Latch_RS IS
   FOR estructura
         FOR Nor1, Nor2: NotOr
               USE ENTITY work.NotOr(descripcion);
         END FOR;
   END FOR;
END config_Latch_RS;
```

1.5 Paquetes y Librerías

Los Paquetes permiten definir funciones, constantes y tipo de datos para ser utilizado por múltiples diseños en VHDL. De esta forma, la definiciones dadas en un paquete serán visibles a cualquier programa que utilice la orden USE con el nombre del paquete correspondiente. El paquete es compuesto por dos partes compilables, las declaraciones PACKAGE y PACKAGE BODY. La primera declaración corresponde a la cabecera del Paquete el cual actúa como la Entidad, declarando las interfases y funciones que realiza el PACKAGE BODY y la ultima declaración actúa como una Arquitectura asociada a

una Entidad. Así un Paquete puede contener:

Declaración PACKAGE:

- Declaración de subprogramas
- Declaración de tipos
- Declaración de subtipos
- Declaración de componentes
- Declaración de constantes

Declaración PACKAGE BODY:

- Descripción de los subprogramas
- Valor de las constantes

Para el caso del Latch_RS ejemplo, se puede utilizar un Paquete para declarar la componente NotOr por medio de:

```
PACKAGE mi_compuerta IS
   COMPONENT NotOr
        PORT   (a,b:     IN  std_logic;
                c:       OUT  std_logic);
   END COMPONENT;
END mi_compuerta;                    entonces
```

La compilación de este Paquete llamado mi_compuerta es añadido a la librería work del entorno VHDL. Si se desea crear un nuevo diseño Latch_RS_2 que utilice la declaración del componente de este Paquete, se debe hacer visible en el diseño el Paquete añadiendo la sentencia USE e identificando la librería y el nombre del Paquete. Es necesario que el Paquete sea compilado antes de poder compilar el nuevo Latch_RS_2 porque la sentencia USE hace referencia a un Paquete ya existente en la librería. El código de la nueva versión del latch RS se muestra a continuación:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Latch_RS_2 IS PORT (
   r:  IN      std_logic;
   s:    IN        std_logic;
   q:    OUT       std_logic;
   notq:OUT        std_logic);
END Latch_RS_2;

USE work.mi_compuerta.ALL;
ARCHITECTURE estructura OF Latch_RS_2 IS
   SIGNAL   out1:  std_logic;
   SIGNAL   out2:  std_logic;
BEGIN
Nor1:    NotOr
         PORT MAP (r, out2, out1);
Nor2:    NotOr
         PORT MAP (s, out1, out2);
q        <=       out1;
notq     <=       out2;
END estructura;
```

Observar que:

- La sentencia PORT MAP referencia sus puertos de acuerdo a la posición en que aparecen, a diferencia del código mostrado en Listado2 en donde sus puertos son asignados mediante el operador <=.
- Ya no se ha declarado el componente NotOr dentro del bloque ARCHITECTURE por que dicha declaración se encuentra en el paquete mi_compuerta que es visible en el diseño.
- No es necesario definir el uso de la librería work porque esta librería se carga automáticamente por defecto como parte de nuestro entorno de trabajo.

La sentencia USE como se ha visto se usa para acceder a un paquete compilado de una librería y hacer su contenido visible en nuestro diseño; por lo tanto esta sentencia debe colocarse antes de la declaración de la Entidad ó de la Arquitectura. Se puede seleccionar todos los ítems dentro de un Paquete ó se puede especificar un ítem particular a usar, utilizando la sintaxis:

```
USE nom_librería.nom_paquete.{ALL,ítem};
```

Así en el ejemplo si el Paquete mi_compuerta hubiera contenido varios componentes y definiciones y sólo se desea seleccionar el componente NotOr en el Latch_RS_2 nuestra declaración, USE se cambiaría por:

```
USE work.mi_compuerta.NotOr;
```

II. EJECUCIÓN DEL VHDL

2.1 Semántica Concurrente

En VHDL las instrucciones dentro de una Arquitectura se ejecutan todas al mismo tiempo, es decir en forma concurrente, lo que permite modelar

el hardware de un diseño como bloques de código paralelo. Por ejemplo, en el anterior programa se escribió el fragmento de código:

```
BEGIN
Nor1:   NotOr
        PORT MAP (r, out2, out1);
Nor2:   NotOr
        PORT MAP (s, out1, out2);
q       <=      out1;
notq    <=      out2;
END estructura;
```

que define simultáneamente los componentes NotOR, cuyas salidas y entradas se interconectan y ejecutan al mismo tiempo.

Aunque todas las instrucciones se ejecutan de manera paralela en VHDL, existen estructuras que permiten una ejecución interna de manera secuencial, manteniendo su ejecución paralela con otras. Los tipos de instrucciones que se ejecutan de manera paralela son:

- Asignación de señales.
- Bloques.
- Ecuaciones booleanas.
- Iniciación de componentes.
- Llamadas a procedimientos.
- Procesos.

2.2 Semántica Secuencial

La estructura que permite una ejecución secuencial dentro de una Arquitectura es el Proceso, el cual se puede ver como un conjunto de instrucciones que se ejecutan en forma secuencial, es decir una después de otra y que representa una sola unidad de simulación que puede estar a su vez ejecutando en paralelo otras instrucciones. De esta manera se puede decir que las instrucciones que se ejecutan dentro de un Proceso, son evaluadas de manera secuencial y las que están fuera son evaluadas de manera concurrente.

2.2.1 Procesos Secuenciales

Los Procesos Secuenciales son iniciados con la instrucción PROCESS. Un Proceso puede estar activo ó desactivo, es decir puede estar ejecutándose ó en espera a que cierta condición se cumpla para iniciar su ejecución. La estructura de la instrucción PROCESS es:

```
[Etiqueta:] PROCESS [ (señal 1,...) ]
                [declaraciones];
            BEGIN
                Instrucciones;
            END PROCESS [Etiqueta];
```

La lista sensitiva está conformada por señales separadas por comas y cuando una de las señales cambia el Proceso es invocado, para ser activada y ejecutada las sentencias correspondientes. Cuando se termina de ejecución se espera que ocurra un nuevo cambio en la lista sensitiva.

En una misma Arquitectura pueden existir varios procesos, que pueden estar ejecutándose a la vez ó no, dependiendo si están activos o desactivos. La única manera que un proceso pueda transmitir información a otro proceso es por medio de señales.

2.2.2 Señales

Las señales en VHDL son objetos declarados al inicio de una arquitectura por medio de la palabra reservada SIGNAL y son implementadas directamente en hardware por lo que se puede considerar como una abstracción de una conexión física; por lo tanto pueden servir para interconectar componentes de un circuito, sincronizar la ejecución de un proceso ó transmitir información entre procesos. Debido a que las señales en VHDL son internas a la estructura no requieren estar asociadas a un puerto (pin de entrada ó salida), pero si deben estar asociadas a un tipo de dato que las defina ante el compilador. Un ejemplo de cómo definir señales se muestra a continuación:

```
ARCHITECTURE primera_señal OF ejemplo IS
    SIGNAL direc:  std_logic_vector(0 TO 11);
    SIGNAL  dato: std_logic_vector(7 DOWNTO 0):= X"F4";
    SIGNAL  logic: bolean := 'true';
    SIGNAL  i: integer;
BEGIN
```

Como se observa la señal puede ser inicializada en su definición en algún valor, como en el caso de las señales dato y logic ó pueden ser dejadas sin inicializar y obtener posteriormente sus valores por medio del

operador de asignación ´<=´. Un punto importante que se debe tener en cuenta cuando se trabaja con señales dentro de un proceso, es que debido a que los procesos, se ejecutan de manera secuencial y sólo la última asignación hecha a la señal, será realizada. Así por ejemplo en el siguiente código:

```
EJM:    PROCESS
        BEGIN
            a <= (c AND b) XOR d;
            a <= c OR b;
        END PROCESS EJM;
```

la primera asignación es remplazada por la segunda, por lo cual el código equivale a

```
EJM:    PROCESS
        BEGIN
            a <= c OR b;
        END PROCESS EJM;
```

2.2.3 Variables

Las variables definidas dentro de un proceso, entre las palabras reservadas PROCESS y BEGIN, son visibles sólo dentro del proceso; por este motivo, no pueden ser utilizados para transmitir información entre procesos distintos. Por otro lado, las variables se diferencian de una señal porque sus valores son asignados inmediatamente dentro de un proceso y no al final como ocurre con el caso de las señales.

Un ejemplo de cómo se definen las variables se muestra a continuación:

```
ARCHITECTURE primeras_variables OF ejemplo IS
BEGIN
EJM:  PROCESS
        VARIABLE contador       : integer := 0;
        VARIABLE nible : std_logic_vector (3 DOWNTO 0);
      BEGIN
        :
      END PROCESS;
END primeras_variables;
```

Se observa que al igual que las señales, las variables pueden ser inicializadas en algún valor ó posteriormente asignarle un valor utilizando el operador ´:=´ como se muestra a continuación:

```
EJM:    PROCESS
            VARIABLE a : std_logic;
        BEGIN
            a := b AND c;
            :
        END PROCESS;
```

2.2.4 Instrucciones Secuenciales

Son las instrucciones secuénciales dentro de un proceso las que dan al VHDL la facilidad de describir el comportamiento del sistema digital de manera sencilla y ordenada siendo la sintaxis de estas instrucciones similar a las de cualquier lenguaje de alto nivel.

IF-THEN-ELSE

Este tipo de construcción es utilizada para realizar una ejecución condicional y su estructura es igual a la de cualquier lenguaje de programación. Como ejemplo se muestra el multiplexor 74LS157, de 8 a 4 bits, de la figura 4.

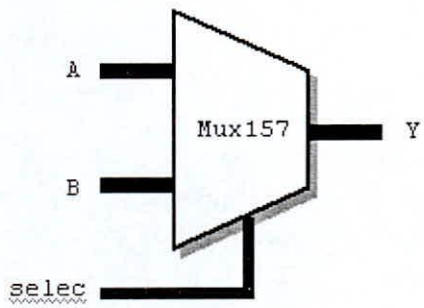

Figura 4. Multiplexor 74LS157

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Mux157 IS PORT(
        A,B     : IN    std_logic_vector(3 DOWNTO 0);

        selec   : IN    std_logic;
        Y       : OUT std_logic_vector(3 DOWNTO 0));
END Mux157;

ARCHITECTURE secuencia OF Mux157 IS
BEGIN
        PROCESS (A,B,selec)             — Lista sensitiva
        BEGIN
                IF selec = '0' THEN
                        Y <= A;
                ELSE
                        Y <= B;
                END IF;
        END PROCESS;
END secuencia;
```

Se debe observar que este tipo de instrucción por su naturaleza misma carece de sentido fuera de un Proceso puesto que su ejecución se desarrolla por medio de evaluaciones secuenciales. La estructura completa de la instrucción `IF` es la siguiente:

```
IF   Expresión lógica  THEN
        instrucciones;
ELSIF  Expresión lógica  THEN
        instrucciones;
        :
ELSIF  Expresión lógica  THEN
        instrucciones;
ELSE
        Instrucciones;
END IF;
```

Con esta estructura se puede colocar las condiciones `ELSIF` que se desee, pudiendo realizar un sistema capaz de evaluar múltiples opciones para la ejecución de una instrucción.

En el ejemplo anterior no fue necesario definir señales ni variables para la descripción del sistema, veamos ahora un ejemplo en donde si es necesario incluirlas como parte del proceso; para ello diseñemos un multiplexor de 8 a 4 bits que posea un buffer interno y una señal de habilitación, tal como se muestra en la figura 5.

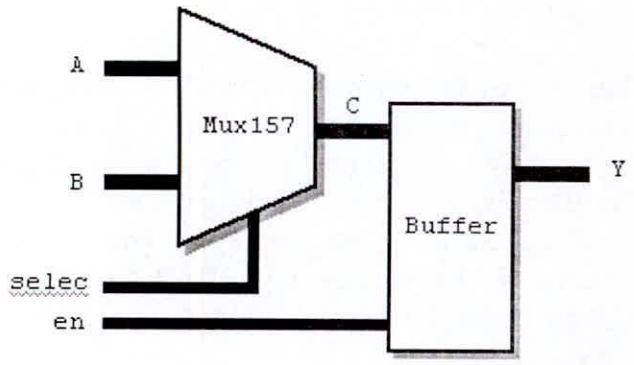

Figura 5. Multiplexor de 8 a 4 bits con Buffer

- Primera Solución: (Utilizando Señales)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Mux_Buffer IS PORT(
        A,B:     IN      std_logic_vector (3 DOWNTO 0);
sel,en: IN      std_logic;
        Y:       OUT     std_logic_vector (3 DOWNTO 0));
END Mux_Buffer;

ARCHITECTURE secuencia OF Mux_Buffer IS
        SIGNAL  C:      std_logic;
BEGIN
PROC1: PROCESS (A,B,sel,en)             — Lista sensitiva
        BEGIN
                IF sel = '0' THEN C <= A;
                ELSE C <= B;
                END IF;
                Y  <=  (Y AND (NOT en)) OR ( C AND en);
        END PROCESS;
END secuencia;
```

El código anterior mostrado posee un error funcional, esto es en la compilación no se genera ningún tipo de error, pero en la simulación el funcionamiento del circuito no será el deseado. El error se debe a que la asignación de la señal
C <= A ó C <= B no tendrá efecto hasta el final del proceso, por lo que en la asignación de Y, el valor de C no será el deseado produciéndose el error. Para solucionar este problema se debe considerar la asignación de Y en:

```
ARCHITECTURE secuencia OF Mux_Buffer IS
        SIGNAL  C:      std_logic;
BEGIN
PROC1: PROCESS (A,B,sel,en)    -- Lista sensitiva
        BEGIN
        IF sel = '0' THEN C <= A;
                ELSE C <= B;
                END IF;
        END PROCESS;            — C se actualiza aquí!!
        Y  <=  (Y AND (NOT en)) OR ( C AND en);
END secuencia;
```

- Segunda Solución: (Utilizando Variables)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Mux_Buffer IS PORT(
  A,B: IN std_logic_vector (3 DOWNTO 0);
  sel,en: IN std_logic;
  Y: OUT std_logic_vector (3 DOWNTO 0));
END Mux_Buffer;

ARCHITECTURE secuencia OF Mux_Buffer IS
BEGIN
        PROCESS (A,B,sel,en)    — Lista sensitiva
           VARIABLE  C: std_logic;
        BEGIN
           IF sel = '0' THEN C := A; — asignación
           ELSE C := B;
            END IF;
           Y  <=  (Y AND (NOT en)) OR ( C AND en);
        END PROCESS;
END secuencia;
```

CASE

La instrucción CASE permite la ejecución condicional de instrucciones basado en el valor de una expresión. La principal diferencia con la instrucción IF se encuentra en que CASE solamente necesita evaluar una expresión para determinar el caso a ejecutar, mientras que IF requiere seguir un proceso de evaluaciones múltiples mediante los bloques ELSIF. Lo anterior no implica que una instrucción es mejor que otra, sólo establece algunas diferencias que se deben tener en cuenta durante la programación. La estructura de una instrucción CASE es la siguiente:

```
CASE Expresión IS
        WHEN  opción 1  =>  instrucciones secuenciales;
        :
        WHEN  opción n  =>  instrucciones secuenciales;
        :
        [WHEN OTHERS  =>  instrucciones secuenciales;]
END CASE;
```

La palabra reservada OTHERS es utilizada para remplazar todas las opciones no listadas en los bloques WHEN, de esta manera en caso que la expresión no satisfaga ninguna de las opciones listadas se ejecutaran las instrucciones especificadas en WHEN OTHERS.

Para aclarar el uso de esta instrucción se ha desarrollado el multiplexor 74LS157, mostrado en la figura 4, por medio de la estructura CASE.

```
ARCHITECTURE secuencia OF Mux157 IS
BEGIN
  PROCESS (A,B,selec)
  BEGIN
        CASE selec IS
                WHEN  '0'        =>       Y <= A;
                WHEN OTHERS =>       Y <= B;
                END CASE;
        END PROCESS;
END secuencia;
```

La Entidad de este diseño está definido en el listado 8, en donde se solucionó el mismo ejemplo usando la instrucción IF. Compare ambas soluciones.

Se observa que la instrucción de asignación Y <= B es ejecutada cada vez que selec toma el valor de 1, es decir OTHERS reemplaza la condición '1' no listada en los casos WHEN. A manera de nota vale la pena mencionar que sólo una condición puede ser cierta en la instrucción CASE, es decir solo un caso WHEN puede estar en ejecución.

FOR

La instrucción FOR permite la ejecución de un juego de instrucciones de manera repetitiva, estableciendo para ello un bucle cerrado. En este tipo de bucle no se necesita declarar una variable de incremento para controlarlo, la misma instrucción FOR se encargará de incrementar su valor en uno en cada ciclo. Esto significa que no se tiene que manipular la variable que controla el bucle FOR. La estructura de FOR es:

```
FOR identificador IN rango LOOP
        Instrucciones;
END LOOP;
```

Por ejemplo, si se está implementando la lectura de un teclado PC-AT, el cual tiene un protocolo de comunicación en serie de 11 bits. La transmisión de datos se inicia con un primer bit 0 indicador de inicio, los siguientes 8 bits de datos (el menos significativo primero), un bit de paridad impar y un último bit 1 indicador de fin. El diagrama se muestra en la Fig. 6.

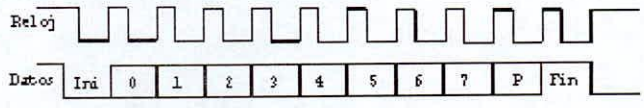

Figura 6. Protocolo de Tx del teclado AT

Este diagrama representa un byte de datos transmitidos desde el teclado sincronizado con el flanco de bajada de la señal del reloj generado por el teclado, el cual se encuentra entre 20khz y 30khz. El bit de paridad impar P se agrega de forma tal que de los 9 bits totales, 8 bits son de datos y el bit P completa un número impar de 1´s.

Entonces si se desea realizar un diseño que permita la lectura del teclado, se necesita validar los datos adquiridos utilizando el bit de paridad impar proporcionado; para ello se necesita implementar un proceso que nos determine la paridad de la señal Byte, en donde se almacenó la tecla pulsada, para luego compararla con el bit de paridad impar P. Entonces el diseño de dicho proceso de detección de paridad está dado por:

```
Paridad: PROCESS
          VARIABLE      parid:std_logic := '0';
        BEGIN
          FOR i IN  0  TO  7 LOOP
                Parid :=  parid XOR Byte(i);
          END LOOP;
          IF (P XOR Parid) = '1' THEN
                — dato valido;
          ELSE
                — no concuerda la paridad;
          END IF;
        END PROCESS;
```

Debe observarse que la etiqueta i utilizada en el lazo FOR no es una variable definida del proceso.

WHILE

Los bucles WHILE permiten ejecutar un conjunto de instrucciones mientras cierta condición se mantiene como verdadera. En este tipo de bucle si es necesario definir una variable que controle el ciclo y por lo tanto tener control sobre ella, pudiendo incrementarla, decrementarla, etc. La estructura de esta instrucción es:

```
WHILE expresión booleana LOOP
        instrucciones;
END LOOP;
```

Para dar un ejemplo de cómo se utiliza la variable de ciclo, se muestra el proceso de detección de paridad por medio de un ciclo WHILE.

```
Paridad: PROCESS
          VARIABLE parid: std_logic := '0';
          VARIABLE      i : integer := 0;
        BEGIN
          WHILE i < 8 LOOP
                Parid :=  parid XOR Byte(i);
                i :=  i + 1;
          END LOOP;
          IF (P XOR Parid) = '1' THEN
                — dato valido;
          ELSE
                — no concuerda la paridad;
          END IF;
        END PROCESS;
```

Se observa en este caso que ha sido necesario declarar una variable i en el proceso para poder controlar el bucle por medio de una instrucción que va incrementando dicha variable en 1 en cada ciclo.

EXIT

La instrucción `EXIT` se utiliza para terminar un bucle de manera abrupta en cualquier momento. Como ejemplo es presentado el contador ascendente que termina cuando ocurre una señal de interrupción `irq:`

```
LOOP
        cuenta      <=      cuenta + 1;
        IF (irq = '1') THEN
           exit;
        END IF;
END LOOP;
```

NEXT

La instrucción NEXT se utiliza para saltar una o más ejecuciones de un bucle LOOP. Así en el siguiente ejemplo las instrucciones del bucle se ejecutarán cuando la etiqueta i tome los valores 1, 2, 3, 5 y 6

```
FOR i IN 1 TO 6 LOOP
        IF (i = 4) THEN NEXT;
        END IF;
        :
END LOOP;
```

Se observa que a diferencia de la instrucción EXIT, NEXT no termina las iteraciones del bucle, simplemente produce que la iteración actual sea descartada y pase a la siguiente.

2.2.5 Registros

El diseño de registros en VHDL se puede hacer de dos maneras:

- Utilizando componentes en una arquitectura de estructura.
- Utilizando un proceso sensible al flanco de reloj.

En el caso de arquitecturas de comportamiento descriptivas se utiliza el segundo caso. El lenguaje VHDL infiere que un registro va a ser creado para una señal q basándose en lo siguiente:

- El elemento es sensible a la señal de reloj.
- Se está sintetizando un elemento síncrono si la asignación de q ocurre en el flanco de subida ó bajada del reloj, esto es cuando:

```
reloj'EVENT AND reloj = '1'
```

- No se incluye la cláusula ELSE en la instrucción IF-THEN, lo que implica que si la condición de flanco de subida no ocurre, q mantiene su valor.

Por ejemplo realicemos el diseño de un contador de 4 bits con carga sincrónica y con una señal de reset síncrona, tal como se muestra en la Fig. 6.

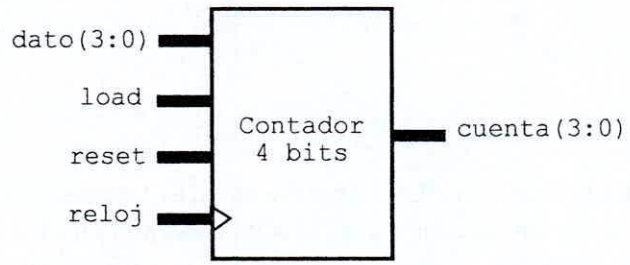

Figura 6. Contador de 4 bits con reset asíncrono

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY Contador IS PORT(
   load,reloj,reset: IN  std_logic;
   dato: IN std_logic_vector (3 DOWNTO 0);
   cuenta: OUT std_logic_vector (3 DOWNTO 0));
END Contador;

ARCHITECTURE registros OF Contador IS
   SIGNAL dcba   : std_logic_vector (3 DOWNTO 0):=X"0";
BEGIN
Conta:     PROCESS (reloj, reset)
           BEGIN
             IF reset = '1' THEN
                   dcba  <=  "0000";
             ELSIF reloj'EVENT AND reloj = '1'  THEN
                   IF load  =  '1'  THEN
                    dcba  <= dato;
                   ELSE
                         dcba  <= dcba + 1;
                   END IF;
             END IF;
           END PROCESS;
           cuenta <= dcba;
END registros;
```

Aunque en apariencia el código anterior funciona satisfactoriamente, en el momento de su compilación va generar un error, debido a la asignación dcba <= dcba + 1, ya que los tipos de datos que se están sumando son del tipo std_logic_vector e integer para el cual, el operador + no está definido.

Dependiendo del entorno en el que se trabaje se podrá añadir una librería que contemple ese tipo de operación ó se tendrá que definir. Por ejemplo, a partir de la versión 4.0 de WARP se incorpora un paquete llamado std_arith que sobrecarga los operadores aritméticos +, - y los operadores de relación =, <, ... para los tipos std_logic, std_logic_vector e integer. Así bajo este entorno sólo se tendrá que añadir al inicio de la Entidad ó Arquitectura, la línea.

```
USE work.std_arith.ALL
```

2.2.6 Wait

Otra manera de activar un proceso es a través de la instrucción WAIT. Ésta es una instrucción secuencial cuya función es suspender la ejecución de un proceso hasta que cierta condición especificada se vuelva valida. Hay tres tipos de modificadores para esta instrucción:

WAIT UNTIL expresión lógica

Detiene la ejecución hasta que la expresión lógica especificada sea verdadera, de esta manera se puede hacer que un proceso espere el flanco de subida de un reloj para que continué su ejecución mediante:

```
Sinc:     PROCESS
          BEGIN
            WAIT UNTIL reloj'EVENT AND reloj ='1';
            :
          END PROCESS;
```

lo cual implica que el proceso implementado es síncrono.

WAIT FOR xx ns

Detiene la ejecución del proceso durante xx nano-segundos, permite no sólo sincronizar distintas señales, sino también simular el comportamiento real de los circuitos integrados añadiéndoles un tiempo de retardo a las asignaciones para emular el tiempo de propagación.

Notas:

- La palabra EVENT es un atributo que se utiliza en las señales para determinar que a ocurrido un cambio en el valor de la señal
- Se puede eliminar la señal dcba utilizando un puerto tipo BUFFER para el puerto cuenta, inclusive se puede eliminar el puerto dato si el puerto cuenta se define como INOUT.

WAIT ON (a,b,c,...)

Detiene la ejecución del proceso hasta que ocurra una transición en cualquiera de las señales a, b, c, etc. Este es el método implementado por defecto cuando se añade una lista sensitiva a un proceso, por consiguiente un proceso con esta lista no puede contener una instrucción WAIT ON internamente, puesto que esta ya ha sido generada de manera implícita.

2.2.7 Memoria Implícita

Los problemas de memoria implícita ocurren en VHDL debido a que las señales dentro de un proceso tienen un valor presente y una valor futuro que será asignado al finalizar el proceso. Por lo tanto si en un proceso el valor futuro de una señal no puede ser determinado, automáticamente se sintetizará un `latch` para almacenar el valor del estado actual. Esta manera de funcionamiento del VHDL tiene la ventaja de permitir la creación de memoria de manera sencilla pero tiene la desventaja de que se pueda generar `latch` indeseados si todos los casos de una instrucción condicional no son considerados.

Por ejemplo, para especificar el funcionamiento de una compuerta AND simple, se desarrolla el siguiente código:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY MiAND IS PORT (
      a:  IN   std_logic;
      b:  IN  std_logic;
      c:  OUT std_logic);
    END MiAND;

ARCHITECTURE memoria OF MiAND IS
BEGIN
        PROCESS (a, b)
                IF a = '1' THEN  c <= b;
                END IF;
        END PROCESS;
END memoria;
```

Debido a que la especificación de la instrucción IF..THEN..ELSE está incompleta, y no se puede determinar el valor que tendrá c cuando a = '0', automáticamente se sintetizará un latch para almacenar el valor actual. De esta manera el circuito sintetizado es como la que se muestra en la Fig. 7.

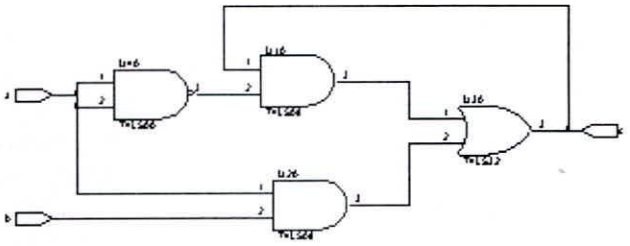

Figura 7. Problema de la memoria implícita.

Dicho circuito tiene su salida c igual a b cuando a es 1, y por medio de la realimentación mantiene c con su valor anterior cuando a es 0. El diseño correcto de la compuerta AND se logra especificando completamente la instrucción condicional como:

```
ARCHITECTURE memoria OF MiAND IS
BEGIN
        PROCESS (a, b)
                IF a = '1' THEN  c <= b;
                ELSE    c <= '0';
                END IF;
        END PROCESS;
END memoria;
```

Esta especificación completa de la instrucción condicional produce que el sintetizador genere una sola AND.

Entonces si se quiere evitar los problemas de memoria implícita ó generación de latch no deseados, se debe terminar siempre toda instrucción IF con una cláusula ELSE, asimismo se deben definir todas las alternativas posibles en una instrucción CASE ó se debe terminar con una cláusula WHEN OTHERS.

2.2.8 Máquinas de Estado Finito

Los circuitos lógicos secuenciales se clasifican dentro de los circuitos conocidos como máquinas de estado en dos tipos:

Figura 8 - Máquina de Moore

Maquina de Moore

En una máquina de Moore las salidas del sistema sólo dependen del estado interno, cambiando únicamente cuando cambia su estado. Un ejemplo de estas máquinas son los contadores up/down, entre otros. El diagrama general de una máquina Moore se muestra en la Fig. 8, en donde se observa que las salidas del sistema son codificadas a partir de los estados mediante una lógica combinacional.
Para implementar una máquina de Moore, se muestra el ejemplo de un tren de lavado de carros de la figura 9.

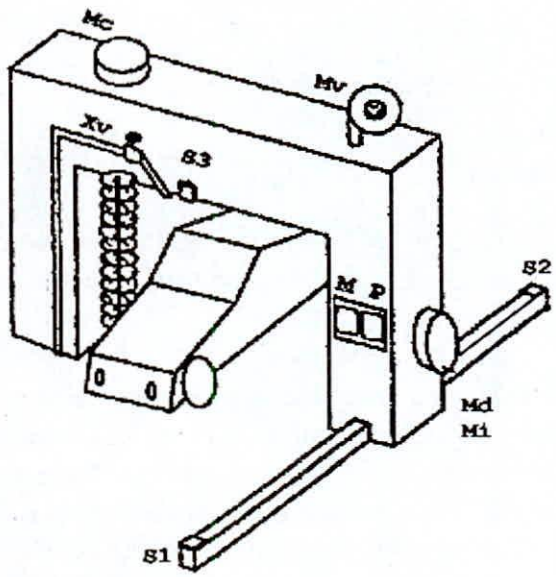

Figura 9 - Lavado de carros.

Para controlar el lavado se necesita:

- Tres motores:
  - Un motor principal que mueve el tren a lo largo del carril con dos señales de actuación MP1 y MP2. Cuando se activa MP1 el tren se mueve de izquierda a derecha. Cuando se activa MP2 el tren se mueve de derecha a izquierda.
  - Un motor para los cepillos, con una única señal de actuación MC.
  - Un motor para el ventilador, con una única señal de actuación MV.
- Una electro-válvula XV que permita la salida del liquido del lavadero hacia el carro.
- Dos sensores fin de carrera S1 y S2 que detectan la llegada del tren a los extremos del carril.
- Un sensor S3 que detecta la presencia del carro.
- Dos pulsadores M y P de marcha y paro respectivamente.

Con estas variables se puede determinar las entradas y salidas de nuestro diseño, así pues:

Las salidas del sistema son:

- Mi = Motor que lleva el tren a la izquierda del carril.
- Md = Motor que lleva el tren a la derecha del carril.
- Mc = Motor de giro de los cepillos.
- Mv = Motor de giro de los ventiladores.
- Xv = Accionamiento de la salida de jabón.

Habiendo establecido las entradas y salidas globales, es decir su Entidad, se puede establecer el comportamiento que se desea. Se describe el comportamiento para establecer su Arquitectura.

Se intuye que en el funcionamiento del sistema, todos los carros idealmente pasan por un mismo proceso, por lo que cada una de las etapas del lavado será un estado del sistema, se tendrá los siguientes estados:

- Estado 1: Inicialmente el sistema se encuentra en el extremo izquierdo, con el sensor fin de carrera izquierdo activado (S1 = 1) ya que ésta es la po-

[1] Notas:
- Es conveniente que todo proceso tenga una lista sensitiva ó una instrucción WAIT ON al comienzo, en caso contrario muchos compiladores producirán error al generar un bucle que se ejecuta de manera infinita.
- Es preferible utilizar la instrucción WAIT ON al inicio del proceso debido a que la gran mayoría de sintetizadores suelen descartar las listas sensitivas.
- Como regla general se debe utilizar solamente una instrucción WAIT por proceso.

sición donde reposa el tren del lavadero. El sistema se pone en marcha al activarse el pulsador (M = 1) siempre y cuando haya un carro dentro del lavado automático, es decir el sensor que indica la presencia de un carro debajo del tren del lavadero está activo (S3 = 1).

- Estado 2: Una vez accionado M, el tren del lavadero comenzará a funcionar desplazándose hacia la derecha (Md = 1), accionando el motor de los cepillos (Mc = 1) y el jabón liquido (Xv = 1) hasta llegar al final del carril derecho, activándose (S2 = 1). En este momento se pasa al siguiente estado.
- Estado 3: En este estado se regresa el tren del lavadero hacia el carril derecho (Mi = 1) y se mantienen los cepillos y la válvula de jabón encendidos (Mc = 1 y Xv = 1). Cuando se llega al final de carril izquierdo se activa (S1 = 1) y se pasa al siguiente estado
- Estado 4: En este estado se regresa el tren del lavadero nuevamente hacia el carril derecho (Md = 1) con el ventilador encendido para secarlo (Mv = 1), hasta llegar nuevamente al fin del carril derecho, momento en que se vuelve a activar (S2 = 1) y se pasa al siguiente estado.
- Estado 5: Se regresa nuevamente el tren del lavadero hacia el carril izquierdo (Mi = 1) manteniendo el ventilador encendido (Mv = 1). Cuando se llega a la posición final del carril izquierdo (S1 = 1) se pasa al estado inicial quedando el trabajo terminado.

El diseño se podría dar por terminado aquí y empezar la implementación en VHDL, pero siempre es bueno considerar imprevistos de emergencia para ello se debe accionar el pulsador de parada P, con el cual se aborta el lavado y el sistema debe regresar al estado inicial, de esta manera se tendrá un estado adicional.

- Estado 6: Cuando se acciona el pulsador de parada (P = 1) en cualquier estado, el tren del lavadero se dirige hacia el carril izquierdo (Mi = 1) y se apagan todos los actuadores (Mc = Mv = Xv = 0). Cuando éste llega al final del carril izquierdo, se pasa al estado inicial.

A continuación se muestra en la Fig.10 un diagrama de estados planteado para el lavadero de carros.

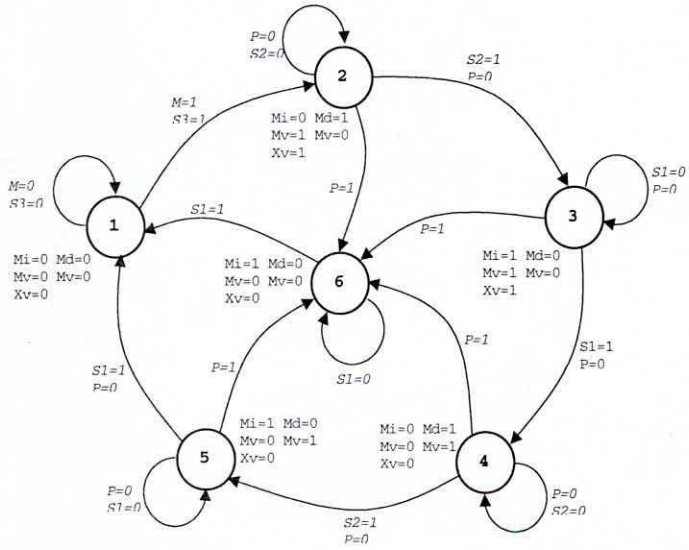

Figura 10. Diagrama de estados del lavadero de carros.

```
LIBRARY ieee;
USE  ieee.std_logic_1164.ALL;
ENTITY lavado_carro IS PORT (
 reset,m,p,s1,s2,s3,clk:  IN  std_logic;
 md,mi,mc,xv:          OUT std_logic);
END    lavado_carro;

ARCHITECTURE moore OF lavado_carro IS
  TYPE maq_estados IS (esta1,esta2,esta3,
                       esta4,esta5,esta6);
  SIGNAL est, prox_est:      maq_estados;
BEGIN
A: PROCESS (m,p,s1,s2,s3)
   BEGIN
     CASE est IS
       WHEN esta1 =>
         IF m = '1' AND s3 = '1' THEN
                  prox_est <= esta2;
         ELSE
                  prox_est <= esta1;
         END IF;
       WHEN esta2 =>
         IF p = '1' THEN
                  prox_est <= esta6;
         ELSIF s2 = '1' THEN
                  prox_est z= esta3;
         ELSE
                  prox_est <= esta2;
         END IF;
       WHEN esta3 =>
         IF p = '1' THEN
                  prox_est <= esta6;
         ELSIF s1 = '1' THEN
                  prox_est z= esta4;
```

```
        ELSE
                prox_est <= esta3;
          END IF;
         WHEN esta4 =>
          IF p = '1' THEN
                prox_est <= esta6;
          ELSIF s2 = '1' THEN
                prox_est z= esta5;
                prox_est <= esta4;
          END IF;
         WHEN esta5 =>
          IF p = '1' THEN
                prox_est <= esta6;
          ELSIF s1 = '1' THEN
                prox_est z= esta1;
          ELSE
                prox_est <= esta5;
          END IF;
         WHEN esta6 =>
          IF s1 = '1' THEN
                prox_est <= esta1;
           ELSE
                prox_est <= esta6;
                  END IF;
         WHEN OTHERS
                prox_est <= esta1;
        END CASE;
        END PROCESS;
B:      PROCESS (reset, clk)
        BEGIN
          IF reset = '1' THEN
                  est <= est1;
          ELSIF (clk'EVENT AND clk = '1') THEN
                  est <= prox_est;
          END IF;
        END PROCESS;
md <= '1' WHEN (est = esta2 OR est = esta4) ELSE '0';
mi <= '1' WHEN (est = esta3 OR est = esta5 OR est = esta6)
        ELSE '0';
mc <= '1' WHEN (est = esta2 OR est = esta3) ELSE '0';
mv <= '1' WHEN (est = esta4 OR est = esta5) ELSE '0';
xv <= '1' WHEN (est = esta2 OR est = esta3) ELSE '0';
END moore;
```

En el programa anterior se ha definido un nuevo tipo de dato llamado maq_estados por medio de la instrucción TYPE, este nuevo tipo de datos toma los valores de esta1, esta2,..., esta6 y las señales creadas con ese tipo nos sirve para almacenar directamente el estado actual de la maquina de Moore.

Se observa que se han utilizado dos señales del tipo maq_estados, que tiene el estado actual de la maquina, y prox_est, que tiene el próximo estado; luego se han implementado dos procesos, uno para determinar la lógica del estado siguiente y otro para determinar el cambio de estado en el flanco de subida del reloj.

Se pudo haber realizado un solo proceso sensible a la señal del reloj incluyendo dentro el bloque CASE para determinar el estado siguiente, pero el código se hubiera hecho menos legible y difícil de seguir.

Finalmente, debido a que las señales de salida sólo dependen del estado actual en una máquina de Moore, las salidas son implementadas fuera de los procesos para que siempre sean evaluadas en forma concurrente.

Máquina de Mealy

En la máquina de Mealy las salidas del sistema cambian debido a un cambio en los estados ó en las entradas. El diagrama general de una máquina de Mealy es como se muestra en la Fig. 11.

Figura 11 - Maquina de Mealy

Se observa que la máquina de Moore es en realidad un caso particular de la maquina de Mealy en la cual no intervienen las entradas para determinar la lógica de salida. De esta manera se ve fácilmente que para la generación de una máquina Mealy lo único que se deberá variar en el programa es la asignación de las salidas, que ahora también dependerán de las entradas del sistema.

Por ejemplo, si en el programa anterior se varían las últimas líneas de asignación de salidas por:

```
md <= '1'  WHEN ((est = est2 AND s1 = '1') OR est = est4)
          ELSE '0';
mi <= '1'  WHEN (est = est3 OR s3 = '1') ELSE '0';
mc <= '1'  WHEN (est = est2 OR est = est3)       ELSE '0';
mv <= '1'  WHEN (s2 = '1' XOR est = est5) ELSE '0';
xv <= '1'  WHEN (est = est4 NOR P = '1') ELSE '0';
```

Se tendrá una máquina de estados de Mealy. Debido a que los cambios realizados han sido al azar, las

variaciones hechas al circuito no conservan ninguna relación con el diseño planteado; estos cambios lo único que muestran es la dependencia hacia las salidas de las entradas s1, s2, s3 y P determinando que la máquina de estados sea considerada como de Mealy.

2.3 Ejemplo de Diseños.

2.3.1 El Ascensor

Enunciado.- Se desea diseñar el controlador de un ascensor para una vivienda de tres pisos: piso 1, piso 2 y piso 3. Las entradas del circuito son tres botones para indicar el piso al cual el usuario quiere ir. Tres sensores indican el piso en el cual se encuentra el ascensor en un momento dado y un sensor en la puerta del mismo para detectar la presencia de algún obstáculo, en cuyo caso la puerta no debe cerrarse. Las salidas del circuito son: el motor que sube, baja y detiene el ascensor; y el motor que abre y cierra la puerta.

Iinicialmente los motores están apagados con la puerta abierta. En el momento que alguien pulsa un botón dentro del ascensor y esta corresponde a un piso diferente del que nos encontramos, se pasa al cerrado de la puerta siempre y cuando el sensor ubicado en la puerta no haya detectado algún obstáculo. Una vez que la puerta está cerrada, se pasa al movimiento del ascensor ascendente si el botón presionado es superior al piso actual y descendente en caso contrario. Una vez que llega al piso deseado se detienen los motores y se abre la puerta del ascensor, permaneciendo así hasta, que ocurra otra llamada. Si mientras el ascensor está en movimiento se pulsan los botones, estos no tendrán efecto alguno.

Solución.- Del enunciado del problema se establece los siguientes estados:

- **Estado 1**: (inicia) Sin importar el piso en el que se encuentra el ascensor, en el estado inicial esta se encuentra con la puerta abierta y en reposo. Esto es el motor de la puerta está abierto y el motor que mueve el ascensor para arriba ó abajo está apagado. Permanece en este estado hasta que se pulse el botón de un piso distinto al piso actual, pasando al siguiente estado.
- **Estado 2**: (cerrar) En este estado se espera que la puerta del ascensor no detecte ningún obstáculo. Cuando el sensor de la puerta indica que está libre de obstáculos, pasa al siguiente estado.
- **Estado 3**: (va) Aquí se procede a cerrar la puerta y luego, si el piso requerido es mayor al piso presente el ascensor asciende y en caso de ser inferior desciende. El ascensor permanece en movimiento hasta que llegue al piso indicado momento en el cual regresa al estado inicial.

El diagrama de estados propuesto para el ascensor es el mostrado en la Fig. 12.

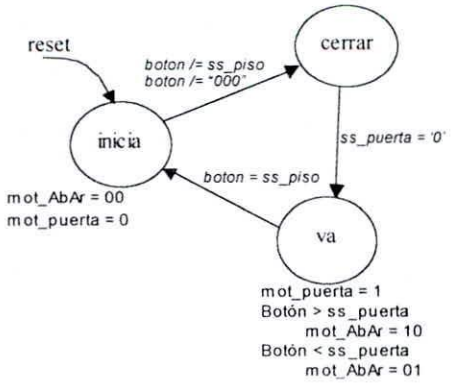

Figura 12. Diagrama de estados del ascensor.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY ascensor IS PORT (
    boton:      IN std_logic_vector(2 downto 0);
    ss_piso:    IN std_logic_vector(2 downto 0);
    clk,reset:  IN std_logic;
    ss_puerta:  IN std_logic;
    mot_ArAb:   OUT std_logic_vector(1 downto 0);
    mot_puerta: OUT std_logic);
END ENTITY ascensor;

ARCHITECTURE control OF ascensor IS
    TYPE estado IS (inicia, cerrar, va);
    SIGNAL presente: estado:= inicia;
    SHARED VARIABLE boton_pulsado: std_logic_vector(2
                                   DOWNTO 0):=boton;
BEGIN
FSM:   PROCESS (reset, clk)
    BEGIN
    IF reset = '1' THEN presente <= inicia;
    ELSIF clk = '1' AND clk'event THEN
        CASE presente IS
                WHEN inicia =>
                IF boton/="000" AND boton/=ss_piso
```

```
                THEN
                        presente<=cerrar;
                        boton_pulsado:=boton;
                END IF;
        WHEN cerrar =>
                IF ss_puerta='0' THEN
                presente<=va;    —Sin obstaculo
                END IF;
        WHEN va =>
                IF boton=ss_piso THEN
                presente<=inicia; —Ya llego al piso
                        END IF;
        END CASE;
    END IF;
    END PROCESS FSM;

SALIDA:  PROCESS (presente)
        —Solo al cambiar de estado
        BEGIN
        CASE presente IS
                WHEN va =>
                        mot_puerta <= '1';   —Cierra
                puerta
                        IF boton _pulsado >ss_piso THEN
                                —Ascensor arriba
                                mot_ArAb <= "10";
                        ELSE    —Ascensor abajo
                                mot_ArAb <= "01";
                                    END IF;
                           WHEN OTHERS =>
                                —Ascensor Parado
                                mot_ArAb  <= "00";
                                  —Abre puerta
                        mot_puerta <= '0';
                              END CASE;
                     END PROCESS SALIDA;
END control;
```

Los resultados de la simulación del programa se muestra en la Fig. 15 del Anexo I.

Observar que:

- En la simulación cuando se entra al estado VA el motor del ascensor empieza a moverse automáticamente, pero el sensor de piso permanece en el piso actual durante un pulso de reloj, esto se debe a que el sistema mecánico del ascensor posee una inercia que lo retarda antes de empezar su movimiento real.
- La etiqueta SHARED utilizada junto con la definición de la variable sirve para poder crear variables globales que puedan ser visibles dentro de distintos procesos.

2.3.2 *El Teclado*

Enunciado.- Se desea convertir la transmisión serie enviada por el teclado en paralelo, detectar la ocurrencia de errores de paridad y errores de marco.

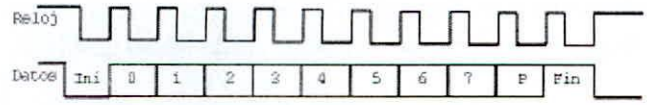

Figura 13. Protocolo de Tx del teclado AT

Solución.- Para resolver este problema primero se diseña un circuito que convierta la entrada serial en paralela, una vez terminada la captura del dato se verifica la paridad de la información y de ser correcta se genera la señal de Cod_ok. El error de marco se determina estableciendo la ocurrencia del bit de inicio en 0 y el bit de fin en 1. Cuando el dato está listo se genera la señal de cod_ack.

Figura 14. Entidad del circuito

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY teclado IS PORT (
        clk_teclado,datos,reset:  IN  std_logic;
        codigo:  OUT  std_logic_vector(7 DOWNTO 0);
        cod_ack,cod_ok: OUT     std_logic;
        error_de_marco: OUT     std_logic);
END ENTITY teclado;

ARCHITECTURE escaneo OF teclado IS
        SIGNAL desplaza:  std_logic_vector(7 DOWNTO 0);
        SIGNAL indicador: integer;
BEGIN
RX:     PROCESS (reset, clk_teclado)
                VARIABLE paridad_rx:    std_logic:='0';
                VARIABLE paridad_calc:  std_logic:='0';
                VARIABLE indice:        integer:=-1;
                VARIABLE marco_mal:     std_logic;
        BEGIN
        IF reset = '1' THEN
                indice := -1;
                codigo <= "ZZZZZZZZ";
                cod_ok  <= 'Z';
                cod_ack <= 'Z';
                desplaza <= "00000000";
                error_de_marco <= 'Z';
        ELSIF clk_teclado'event AND clk_teclado='0' THEN
```

```
        indice := indice + 1;
        CASE indice IS
        WHEN 0 =>
                cod_ack <= '0';
                cod_ok  <= 'Z';
                paridad_calc := '0';
                codigo   <= "ZZZZZZZZ";
                error_de_marco <= 'Z';
                IF datos/='0'THEN
                        marco_mal := '1';
                ELSE marco_mal := '0';
                END IF;
        WHEN 9 =>
                codigo  <=desplaza(7 DOWNTO 0);
                paridad_rx      :=      datos;
        WHEN 10 =>
                indice := -1;
                desplaza <= "--------";
                IF datos/='1'THEN
                        marco_mal := '1';
                        --No se considera ELSE por
                END IF; --si hubo error al inicio
                IF (paridad_calc XOR paridad_rx) = '1'
                THEN
                        cod_ok <= '1';    --Dato valido
                ELSE    cod_ok <= '0';    --Error de paridad
                END IF;
                error_de_marco <= marco_mal;
                cod_ack <= '1';
        WHEN OTHERS =>
                desplaza(7 DOWNTO 1)  <=
                                desplaza(6DOWNTO0);
                desplaza(0) <= datos;
                paridad_calc :=  paridad_calc XOR datos;
        --calcula la paridad inpar
        END CASE;
    END IF;
    indicador <= indice;
    END PROCESS RX;
END escaneo;
```

Los resultados de la simulación del programa se presenta en la Figura 16 del anexo I.

Se Observa que:

- La primera transmisión corresponde a un envío sin error de la tecla "a" cuyo código es 1C; la segunda transmisión es de la tecla "p" que tiene el código 4D y en esta transmisión se tiene error de paridad (está en 0 y debería ser 1) y el error de marco en el bit de parada (esta en 0 y debería ser 1)
- Las señales error_de_marco y cod_ok se fuerzan a alta impedancia para que dichas líneas de control puedan ser utilizadas para otros propósitos mientras el teclado no ha sido decodificado. Para mejorar este proceso la señal de cod_ack se volvería a la petición de IRQ y los datos serían transmitidos cuando un habilitador de la petición sea aceptado. (El error de marco junto con la nota no es parte de la pregunta, se han incluido con el único propósito de mejorar la simulación)
- La señal indicador se ha incluido para poder visualizar en la simulación el estado de la variable índice, ya que las variables no pueden ser vistas.

III. TIPOS DE DATOS Y ATRIBUTOS

El lenguaje VHDL es un lenguaje estricto con relación a su tipo de datos, es decir todo objeto definido en VHDL debe tener un tipo de datos asociado; entendiéndose por objeto a cualquier señal, variable, componente ó constante creada. Debido a esta característica, VHDL no permite que a un objeto definido de un tipo se le pueda asignar un dato de otro tipo. Entre los principales tipos de datos en VHDL se explican seguidamente.

3.1 Tipos Escalares

Este tipo de datos tiene un orden pre establecido que permite utilizar operadores de relaciones entre ellos. Son utilizados para realizar el algoritmo del programa y en caso de ser necesario son pasados al sintetizador para ser implementados en hardware; esto no siempre tiene una correspondencia física con el circuito diseñado. Existen cuatro tipos de datos escalares:

- **Enteros.** Los enteros son el tipo base predefinido `integer` cuyo rango depende de la maquina en la que se encuentra el compilador. Este tipo corresponde a los números enteros usuales y soportan las operaciones matemáticas de suma, resta, multiplicación y división. También se pueden definir otros tipos enteros con rangos específicos mediante:

```
TYPE mi_vida_entera IS RANGE 1978 TO 2003;
```

- **Reales.** Los números reales, son aquellos definidos en los lenguajes de programación como punto flotante y su rango también depende de la maquina donde se encuentre el compilador VHDL.

- **Físicos.** Son datos que trabajan asociados a magnitudes físicas, es decir junto con el valor tienen asociada una unidad. Un ejemplo de este tipo de datos físicos predefinidos es time que nos indica la unidad de tiempo y fue utilizada como argumento de la instrucción WAIT FOR. También se pueden generar nuevos tipos físicos declarando la unidad base, por ejemplo si definimos el tipo, resistor, para representar la resistencia, esta sería de la siguiente forma:

```
TYPE ohm IS RANGE 0 TO 999999999999;
        UNITS
          ohm;                  — unidad base
          K = 1000 ohm;              — Kilo ohm
          M = 1000 K;           — Mega ohm
END UNITS;
```

- **Enumerados.** Los tipos enumerados pueden tomar cualquier valor especificado en una lista y por lo general son usados para especificar los posibles estados de un sistema. Por ejemplo se puede crear un tipo de dato estados que tome los valores s1 a s6 mediante:

```
TYPE estados IS (s1,s2,s3,s4,s5,s6);
```

Este tipo de datos por defecto es inicializado en el valor especificado a la izquierda, siendo importante definir el listado en orden alfabético para mantener esta configuración por defecto. Asimismo, este tipo de datos por ser generado por el usuario no tiene ningún tipo de función de resolución que determine que hacer en casos de contingencias, por ejemplo si se tiene dos procesos que asignan valores a una misma señal como se muestra a continuación:

```
ARCHITECTURE tipos OF señales IS
TYPE muestra IS (dat1,dat2,dat3,dat4);
SIGNAL retener:       muestra;
BEGIN
Proc1: PROCESS
    BEGIN
      retener <= dat1;
    END PROCESS;
Proc2: PROCESS
    BEGIN
      retener <= dat3;
    END PROCESS;
END tipos;
```

El compilador de VHDL no podrá determinar que valor, debe asignar la señal retener por lo cual se generará un error. Para solucionar este tipo de problemas se debe crear funciones de resolución que determine el valor que tomarán las señales en dichos casos.

3.2 Tipos Compuestos

- **ARRAY.** El tipo compuesto ARRAY consiste en un arreglo indexado de elementos del mismo tipo. Estos arreglos pueden ser de una sola dimensión con un solo índice ó n-dimensiónales con múltiples índices. Por ejemplo, se puede definir una memoria de programa de 1Kbyte de la siguiente manera:

```
TYPE memoria IS ARRAY (0 TO 1023)
            OF std_logic_vector (7 DOWNTO 0);
```

utilizando este tipo se puede definir ahora un banco de 4 Kbytes mediante:

```
TYPE banco IS ARRAY (0 TO 3) OF memoria;
```

Si se desea también se pueden definir arreglos multidimensionales, por ejemplo se puede definir una matriz cuadrada de 5x5 elementos enteros por medio de:

```
TYPE matriz IS ARRAY (1 TO 5,1 TO 5) OF integer;
```

- **RECORD.** El tipo compuesto RECORD consiste en un arreglo de elementos de diferentes tipos. Por ejemplo, se puede definir el contador de programa PC para una memoria de 4Kbytes, como un arreglo de 16 bancos de 256 bytes cada uno. De esta manera los 12 bits necesarios para direccionar todas las posiciones de memoria se pueden descomponer en dos, un PCH que seleccione uno los bancos de trabajo y un PCL

que apunte a una de las 256 posiciones de memoria del banco seleccionado, tal como se muestra en la Fig. 17.

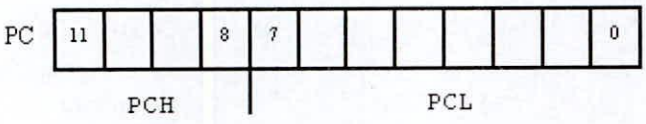

Figura 17. Contador de programa

entonces se puede declarar el tipo RECORD PC y crear una señal de dicho tipo mediante:

```
TYPE PC IS
  RECORD
  PCH  :        std_logic_vector (3 DOWNTO 0);
  PCL  :        std_logic_vector (7 DOWNTO 0);
END RECORD;
SIGNAL memo : PC;
```

Si deseamos seleccionar la dirección de memoria 2AF, debemos actualizar el campo PCH en 2 y el campo PCL en AF por medio de:

```
memo.PCH  <= "0010";
memo.PCL  <= X"AF";
```

3.3 Alias

Los ALIAS son otra manera de llamar una señal ya existente y son utilizadas por lo general para referenciar segmentos de una señal. Por ejemplo otra manera de realizar el direccionamiento de una memoria de 4 Kbytes sería mediante:

```
SIGNAL memo:  std_logic_vector( 11 DOWNTO 0);
ALIAS    PCH:  std_logic_vector( 3 DOWNTO 0)
         IS memo(11 DOWNTO 8);
ALIAS    PCL:  std_logic_vector( 7 DOWNTO 0)
         IS memo(7 DOWNTO 0);
```

los segmentos PCH y PCL son tratados como si fueran señales, por lo que la dirección de memoria 2AF se asignará mediante:

```
PCH    <=     "0010";
PCL    <=  X"AF";
```

3.4 Subtipos

Los subtipos son un subconjunto de valores de un tipo anteriormente definido. Existen dos subtipos enteros predefinidos, definidos como:

```
SUTYPE  natural IS integer  RANGE 0 TO mayor_entero;
SUTYPE  positive IS  integer  RANGE 0 TO mayor_entero;
```

También se pueden crear subtipos de tipos enumerados definiendo el rango de valores que se desea para el subtipo.

3.5 Atributos

Los objetos definidos en VHDL pueden tener información adicional asociada a ellos, que se denomina atributos. Estos atributos son referenciados utilizando la comilla simple y aunque no son parte de las instrucciones de lenguaje VHDL existen un numero estándar de atributos predefinidos.

3.5.1 Asociados al Tipo de Dato

Este tipo de atributos devuelven información referente al tipo de dato asociado a una señal. Para señales de tipo enumerado ó escalar pueden ser usados los siguientes atributos :

- **Left**. Devuelve el valor que se encuentra en el límite izquierdo del tipo.
- **Right**. Devuelve el valor situado en el límite derecho del tipo.
- **Low**. Devuelve el valor del tipo al cual le corresponde la menor codificación.
- High. Devuelve el valor del tipo al cual le corresponde la mayor codificación.
- **Leftof(X).** Da el valor del tipo a la izquierda de X.
- **Rightof(X).** Da el valor del tipo a la derecha de X
- **Pos(X).** Devuelve la posición del valor X dentro de su tipo.
- **Val(N).** Devuelve el valor correspondiente a la posición N.

Veamos como se utilizan estos atributos en el siguiente ejemplo

```
ARCHITECTURE tipo OF ejemplo IS
        TYPE estado IS (S0,S1,S2,S3,S4,S5);
```

```
        TYPE estados_vector IS ARRAY (0 to 7) OF estado;
        SIGNAL est: estados_vector;
        SIGNAL i: integer;
BEGIN
est(0) <= estado'left;              —S0
est(1) <= estado'right;             —S5
est(2) <= estado'low;               —S0
est(3) <= estado'high;              —S5
est(4) <= estado'leftof(S3);        —S2
est(5) <= estado'rightof(S3);       —S4
est(6) <= estado'val(3);            —S3
i       <= estado'pos(S3);          —3
END tipo;
```

dichas asignaciones con atributos son equivalentes a las asignaciones

```
est     <=      S0 & S5 & S0 & S5 & S2 & S4 & S3 & S0;
i       <=      3;
```

Se observa que aunque en el listado no se da una asignación para la señal est(7), esta toma el valor S0. Esto se debe a que toda señal sin inicializar toma su menor valor posible por defecto; es decir, si una señal Y no es inicializada explícitamente, el compilador le asignará el valor Y'low que es igual a Y'left.

3.5.2 Asociados a Arreglos

Este tipo de atributos sólo pueden ser utilizados con arreglos, y sirven para determinar cierta información de ellos. Tenemos los siguientes:

- **Left**(N). Devuelve el valor del limite izquierdo del N arreglo.
- **Right**(N). Devuelve el valor situado en el limite derecho del N arreglo.
- **Low**(N). Devuelve el menor valor del N arreglo.
- **High**(N). Devuelve el mayor valor del N arreglo.
- **Length**(N). Devuelve el valor del tipo al cual le corresponde la mayor codificación.
- **Range**(N). Devuelve la posición del valor X dentro de su tipo.
- **Reverse_range**(N). Devuelve el valor correspondiente a la posición N.

3.5.3 Asociados al Estado de una Señal

Los siguientes atributos devuelven una señal dependiendo de lo ocurrido

- **Event.** Devuelve un valor de verdad si es que ha ocurrido un evento de asignación en la señal, sin importar el cambio real de valor. Por ejemplo, si a una señal binaria que posea el valor 0 se le asigna el valor 0 nuevamente, el atributo event devolverá un estado de verdad ya que ha ocurrido una asignación.
- **Last_event.** Devuelve el tiempo transcurrido desde la ultima asignación en la señal.
- **Last_value.** Devuelve el valor previo en la señal, antes de que ocurra la ultima asignación.

3.5.4 Atributos que Generan Señales

Este tipo de atributos son utilizados para mantener sincronismo ya que permiten generan nuevas señales con respecto a un tiempo determinado. Se tienen los siguientes atributos

- **Delayed**(tiempo). Genera una señal del mismo tipo del dato, pero retrasada el tiempo especificado.
- **Stable**(tiempo). Genera una señal booleana verdadera cuando la señal no tiene ningún evento más allá del tiempo especificado.
- **Quiet**(tiempo). Genera una señal booleana verdadera cuando no ocurre ningún cambio de valor en la señal, es decir la señal no tiene transición.
- **Transaction.** Genera una señal tipo bit que cambia su valor cada vez que la señal tiene una transición.

IV. DIRECTIVAS DE SÍNTESIS EN VHDL

Aunque el lenguaje VHDL es un lenguaje estándar, existen diferentes compiladores proporcionados por distintos fabricantes de PLD, CPLD y FPGA. Por ejemplo Altera produce MaxPlus II, Cypress desarrolla el entorno Warp, Xilinx el Integrated Software Enviroment ISE.

Cada uno de estos sintetizadores de código VHDL proporcionan directivas propias usadas por sus compiladores para controlar diferentes aspectos en

el procedimiento de síntesis de los circuitos digitales. Todas las directivas de síntesis pueden ser controladas insertando en el código fuente VHDL las directivas apropiadas mediante la instrucción ATTRIBUTE

La instrucción ATTRIBUTE puede ser utilizada en casi todo objeto de VHDL, pero la aplicación determinará como estos atributos serán interpretados. Algunas directivas de síntesis son diseñadas para las señales, otras para ser usadas en componentes.

Algunos atributos se pueden definir en la Entidad ó en la Arquitectura de manera que sean globales y cualquier señal definida en una instancia menor heredará dicho atributo por defecto. Otros atributos sin embargo son específicos y tienen que declararse para el objeto deseado.

Los atributos de tipo globales, que pueden ser definidos en cualquiera de los siguientes objetos de VHDL son:

- Entidad.
- Arquitectura.
- Declaración de un componente.
- Referencia a un componente (etiqueta de componente).
- Señales.

De todos estos objetos la Entidad es la que tiene menor precedencia y la Señal es de mayor precedencia, de esta manera una directiva de síntesis colocada en la arquitectura puede ser variada para una señal en particular dentro de una estructura. Es decir las directivas puestas en una Arquitectura sirven como una directiva por defecto para todas las señales dentro de la arquitectura

Estas directivas son descritas en VHDL por medio de la instrucción ATTRIBUTE mediante

```
ATTRIBUTE nombre_atributo OF {nombre_componente|
    nombre_entidad | nombre_ señal | variable | tipo_dato} :
    {componente | entidad | señal | variable | tipo} IS
    valor_atributo;
```

4.1 Directivas de Síntesis en el Entorno Warp

4.1.1 ENUM_ENCODING

Esta directiva de síntesis se utiliza para indicar al compilador la manera en que se debe implementar la codificación de un tipo de dato enumerado. Esta directiva es implementada mediante:

```
ATTRIBUTE enum_encoding OF nom_tipo: TYPE IS
"string";
```

En donde la codificación a utilizar se define entre comillas mediante una cadena de código binario separado por espacios en blanco.

Un ejemplo de como utilizar esta directiva se muestra a continuación:

```
ARCHITECTURE atributo_warp OF ejemplo IS
  TYPE tipostados is (S0, S1, S2, S3);
  ATTRIBUTE enum_encoding OF tipostados : TYPE
                                  IS "11 01 00 10";
  SIGNAL estado1 : tipostados;
BEGIN
```

Esta directiva cuando es utilizada sobre escribe cualquier directiva state_encoding dada al mismo tipo.

4.1.2 FF_TYPE

Esta directiva se utiliza para forzar al compilador a utilizar un tipo de flip-flop específico en la implementación de una señal en los CPLDs. Los posibles valores de atributo de esta directiva son ff_d, que indica al compilador que se sintetizará una señal utilizando flip-flop tipo D, ff_t para utilizar flip-flop tipo T y ff_opt para que el compilador utilice el flip-flop que consuma menos recursos en el dispositivo. Puede ser utilizada de manera global para todos los elementos de una arquitectura ó se puede declarar para una señal especifica. Para usar esta directiva se debe declarar mediante:

```
ATTRIBUTE ff_type OF nombre_señal : {SIGNAL |
            ARCHITECTURE} IS {ff_d | ff_t | ff_opt};
```

Así por ejemplo se puede definir una directiva global que implemente todas las señales con flip flop tipo D y especificar también que una de estas señales sea implementada con un flip flop tipo T.

```
ARCHITECTURE atributo_warp OF ejemplo IS
  SIGNAL c,b,a : std_logic;
  ATTRIBUTE ff_type OF atributo : ARCHITECTURE IS ff_d;
  ATTRIBUTE ff_type OF c: SIGNAL IS ff_t;
BEGIN
Proc1:      PROCESS (clk)
            BEGIN
                IF (clk'event AND clk = '1') THEN
                c <= NOT c;
                b <= dato1;
                a <= dato2;
            END IF;
            END PROCESS;
END atributo_warp;
```

4.1.3 PART_NAME

Esta directiva sirve para especificar dentro del código VHDL el dispositivo ó integrado en el cual se va a grabar el circuito diseñado. Esta directiva sobre escribe cualquier otro dispositivo que se hubiera seleccionado como dispositivo final.

```
ATTRIBUTE part_name OF nombre_entidad: ENTITY
                                IS "nombre_dispositivo";
```

Por ejemplo se puede especificar como dispositivo final un integrado CY7C371 mediante

```
ENTITY ejemplo IS PORT (
  a,b: in std_logic);
  ATTRIBUTE part_name OF counter: ENTITY IS "c371";
END ejemlo;
```

4.1.4 PIN_NUMBERS

Esta directiva se utiliza para asignar los puertos de una entidad a los pines de un dispositivo. Por lo general la primera vez que se sintetiza es aconsejable no utilizar esta directiva para dejar que el propio sintetizador optimice los recursos del dispositivo. La manera de usar esta directiva es la siguiente:

```
ATTRIBUTE pin_numbers OF entity_name: ENTITY
                        IS "nombre_señal:numero_pin";
```

en donde cada par de la forma nombre_ señal:número, están separados por un espacio en blanco y se encuentra entre comillas; así por ejemplo:

```
ENTITY ejemplo IS PORT (
 a,b: in std_logic);
 ATTRIBUTE pin_numbers OF counter: ENTITY IS "a:6 b:7
";
END ejemplo;
```

define el pin 6 para el puerto a y el pin 7 para el puerto b. En caso de que se listen varios pines del dispositivo se puede utilizar el operador de concatenación & de la siguiente manera

```
ATTRIBUTE atributo OF ejemplo: ENTITY IS
        "señal_1:1"&
        "señal_2:2"&
        :
        "señal_n:n";
```

4.1.5 STATE_ENCODING

Esta directiva especifica el tipo de codificación que se va a realizar con los tipos enumerados

```
ATTRIBUTE state_encoding OF nombre_tipo: TYPE
                                IS
                        valor_atributo;
```

Los valores de atributo de esta directiva son:

- **Sequential.** Este tipo de codificación interna representa cada valor del tipo enumerado mediante una secuencia binaria, utilizando tantos bits como sea necesario. Por ejemplo esta codificación aplicada a:

```
ARCHITECTURE atributo OF ejemplo IS
    TYPE tipostados is (S0, S1, S2,S3);
    ATTRIBUTE enum_encoding OF tipostados : TYPE
                                IS sequential
    SIGNAL estado1 : tipostados;
BEGIN
```

  producirá que tipostados sea implementado como la secuencia binaria 00, 01, 10 y 11.

- **One_hot_zero.** Esta directiva especifica una codificación interna con el primer valor del tipo implementado a 0 y cada siguiente valor del tipo posee su propio bit de posición en la codificación puesto en 1. De esta manera:

```
ARCHITECTURE atributo OF ejemplo IS
    TYPE tipostados is (S0, S1, S2, S3);
    ATTRIBUTE enum_encoding OF tipostados : TYPE
                                IS one_hot_zero
    SIGNAL estado1 : tipostados;
BEGIN
```

  producirá una codificación 000, 001, 010 y 100.

- **One_hot_one.** Esta codificación fuerza al compilador de VHDL que utilice un Flip-Flop por estado, en vez del proceso de síntesis normal en el que trata de minimizar el uso de Flip-Flop. Así en el siguiente ejemplo:

```
ARCHITECTURE atributo OF ejemplo IS
    TYPE tipostados is (S0, S1, S2, S3);
    ATTRIBUTE enum_encoding OF   tipostados :
    TYPE IS one_hot_one
    SIGNAL estado1 : tipostados;
BEGIN
```

  La codificación producida será 0001, 0010, 0100 y 1000. Este tipo de codificación tiene la ventaja de dar mayor velocidad a la máquina de estado, reducir el circuito lógico del estado siguiente y disminuir el uso de celdas lógicas. Tiene como principal desventaja el aumento del número de macroceldas utilizadas.

- **Gray.** Cuando el valor de la directiva está puesto en gray, la codificación interna de los valores sucesivos del tipo enumerado son codificados utilizando un formato Gray, es decir cada valor difiere del anterior en un solo bit. Así por ejemplo:

```
ARCHITECTURE atributo OF ejemplo IS
    TYPE tipostados is (S0, S1, S2, S3);
    ATTRIBUTE enum_encoding OF tipostados : TYPE
                                        IS gray
    SIGNAL estado1 : tipostados;
BEGIN
```

  producirá la codificación 00, 01, 11 y 10.

4.2 Directivas de Síntesis para el Entorno ISE

Antes de poder utilizar algunos atributos en el sintetizador del ISE, estos deben ser previamente declarados mediante:

```
ATTRIBUTE nombre_atributo : string;
```

4.2.1 ENUM_ENCODING

Esta directiva de síntesis se utiliza para indicar al compilador la manera en que se debe implementar la codificación de un tipo de dato enumerado. La codificación a implementar se define en el valor del atributo mediante una cadena de código binario separado por caracteres de espacios. Este tipo de directiva solo puede ser utilizada con un tipo enumerado asociado, es decir no se lo puede definir de manera global.

Un ejemplo de como utilizar esta directiva se muestra a continuación:

```
ARCHITECTURE atributo OF ejemplo IS
  TYPE tipostados is (S0,S1,S2,S3);
  ATTRIBUTE enum_encoding OF tiposestados : TYPE
                                  IS "110 101 011 000";
  SIGNAL estado1 : tipostados;
  SIGNAL estado2 : tipostados;
BEGIN
```

4.2.1 FSM_ENCODING

Esta directiva de síntesis se utiliza para seleccionar la técnica de codificación para una máquina de estado. El tipo de codificación utilizado por defecto es auto, modo en el cual se busca la mayor reducción para cada máquina de estado. Los valores de atributo disponibles son:

- Onehot
- Compact
- Sequential
- Gray
- Johnson
- User

Este tipo de directiva puede ser aplicado de manera global asociándolo a una entidad ó arquitectura; ó puede ser definido directamente para una señal. Antes de usar esta directiva se la debe declara mediante

```
ATTRIBUTE fsm_encoding: string;
```

Después de haber sido declara se puede especificar el atributo por medio de:

```
ATTRIBUTE fsm_encoding OF {nombre_entidad |
  nombre_señal} : {ENTITY | SIGNAL} IS "{auto | onehot |
  compact | gray | sequential | Jonson | user}";
```

4.2.3 FSM_FFTYPE

Esta directiva de síntesis se utiliza para definir el tipo de

flip-flops a utilizado para los registros que implementan una máquina de estados. Aunque existen dos valores de atributos D y T, algunas versiones del ISE no permiten utilizar registros tipo T.

Esta directiva puede aplicarse de manera global a una entidad ó se la puede asociar directamente a una señal.

Para usar esta directiva se la debe declara previamente como<

```
ATTRIBUTE fsm_fftype: string;
```

Una vez declarada se la puede utilizar como sigue:

```
ATTRIBUTE fsm_fftype OF { nombre_entidad |
        nombre_señal} : {ENTITY | SIGNAL} IS "{d | t}";
```

4.2.4 PWR_MODE

Esta tipo de directiva define el modo de operación en la cual las macroceldas son implementadas en un dispositivo, existen dos modos, el de bajo consumo y el estándar de alto rendimiento.

Esta directiva se puede aplicar a una señal ó a un módulo, propagándose la directiva a todos los elementos que hereden propiedades del elemento. Se debe declarar el atributo antes de usarlo mediante:

```
ATTRIBUTE pwr_mode: string;
```

Una vez declarado se lo puede utilizar por medio de

```
ATTRIBUTE pwr_mode OF {nombre_señal | nom_compone}
        : {SIGNAL | componente} IS  "{LOW | STD}";
```

4.2.5 OPEN_DRAIN

Esta directiva se aplica a las salidas para generar una salida de colector abierto, de esta manera el estado uno de la señal de salida produce una señal de alta impedancia Z en el pin del dispositivo. Esta directiva no puede ser utilizada de manera global sino que debe estar asociada directamente a una señal de salida ó a uno de los puertos. Antes de utilizar la directiva se la debe declarar mediante:

```
ATTRIBUTE open_drain: string;
```

Y luego aplicarla a una señal de salida mediante:

```
ATTRIBUTE open_drain OF nombre_señal : SIGNAL
                                IS "TRUE";
```

V. BIBLIOGRAFÍA

Cypres. VHDL training for PLDs, CPLDs and FPGAs, 1999 Dispositivas del curso del año 1999.
Iñigo Oleagoridia;Curso en linea de VHDL de la EUITI -Escuela Universitaria de Ingeniería Técnica Industrial; www.ehu.es/~jtpolagi/index.htm; acceso: Enero 2003.
Hamblen. Rapid prototyping of digital systems, 2000.
Skahill, K. VHDL for programmable logic, 1998.
Sudhakar. VHDL starter's guide, 1998.
Teres, T. V. VHDL lenguaje estandar de diseño electronico, 1998.

VI. ANEXO I

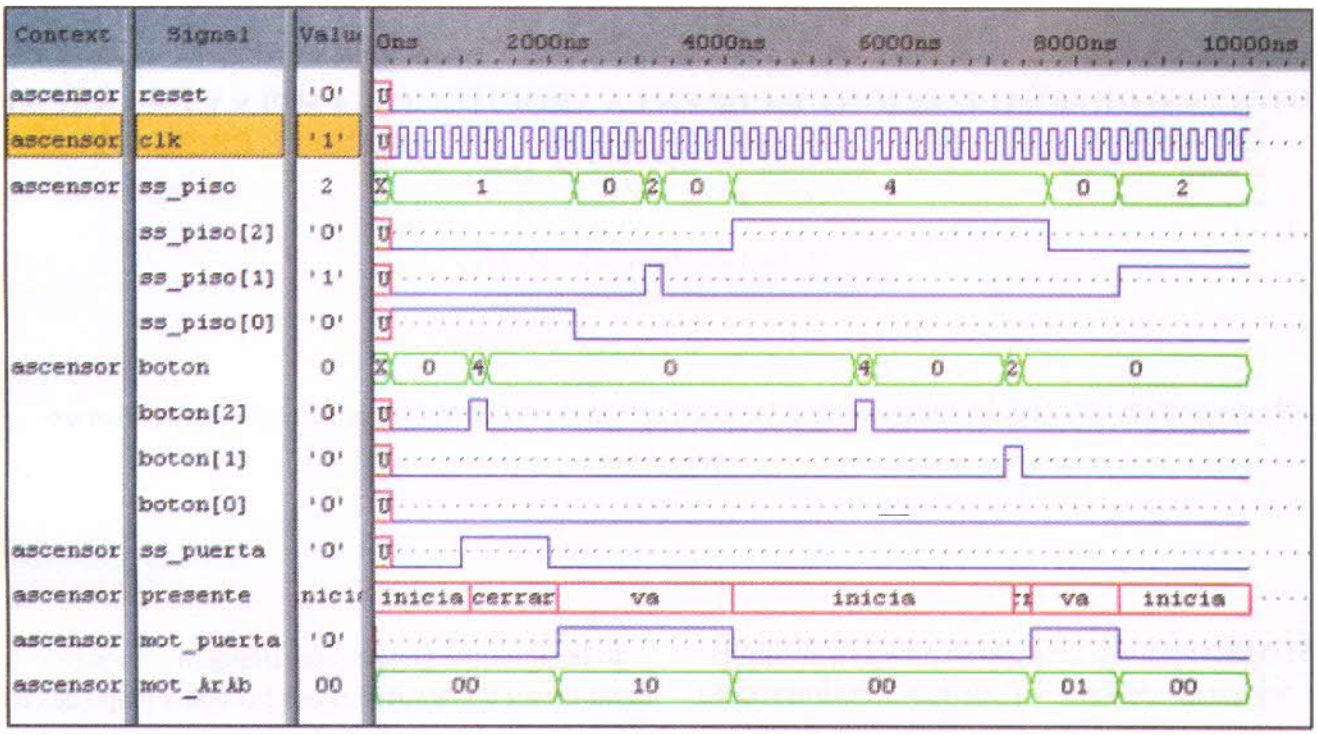

Figura 15 - Simulación del programa del ascensor

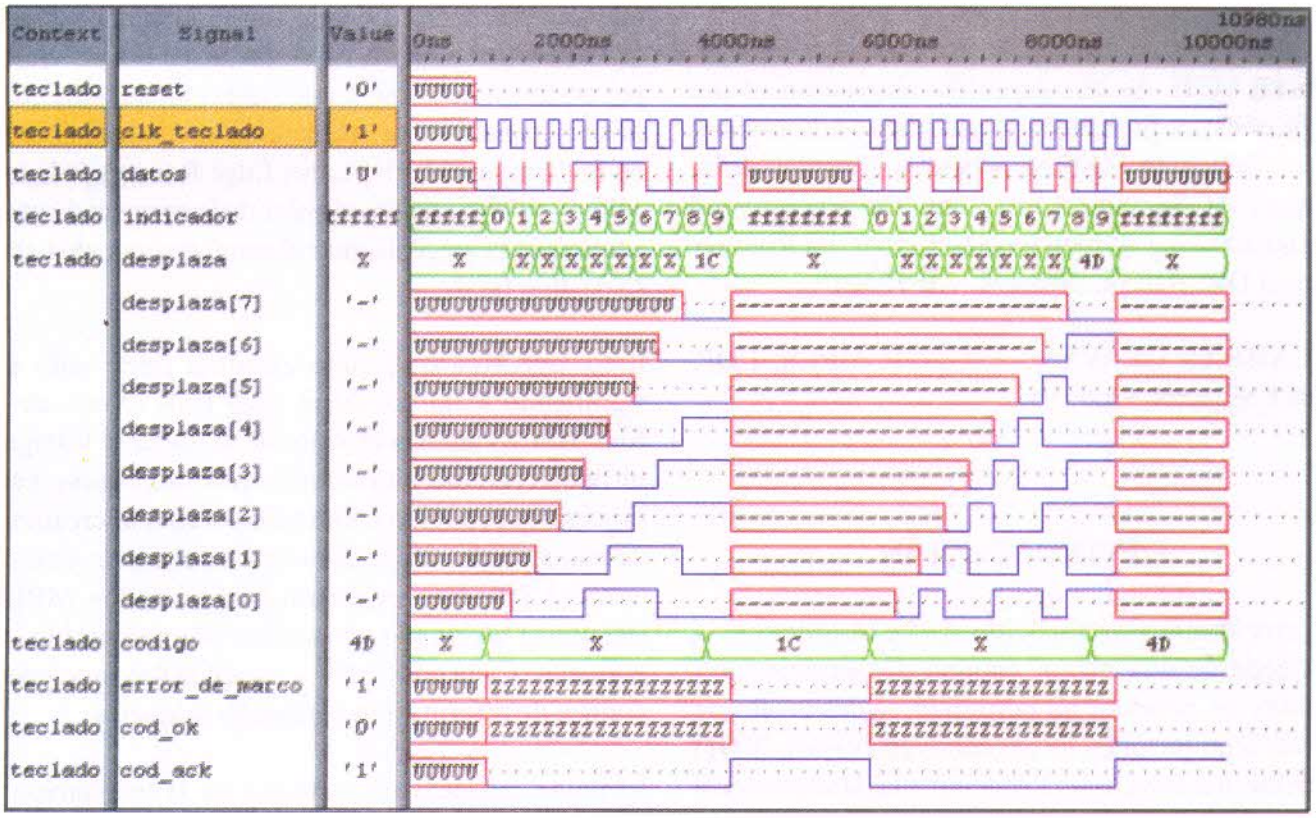

Figura 16 - Simulación del programa de lectura de teclado