1. Ejemplos de Aplicaciones del VHDL.

1.1. INTRODUCCION

En los capítulos anteriores se describió el proceso de diseño para sistemas digitales y las herramientas disponibles (tanto software como hardware) para su implementación. En este capítulo se trataran ejemplos prácticos de diseño de sistemas digitales, iniciando con su descripción hasta llegar a su implementación final utilizando en algunos casos componentes discretos y VHDL.

1.2. ¿QUE ES UNA FSM?

Una máquina de estados es un sistema secuencial sincrónico que posee un número fijo de posibles estados. El valor de sus salidas y la transición entre los estados depende del valor de sus entradas y del estado en que se encuentra actualmente. **Todos los cambios de estado ocurren ante un determinado flanco de la señal de reloj** (ya sea de subida o bajada).

Para entender mejor este concepto imaginemos que nuestra máquina de estados es un televisor, que sólo puede sintonizar cuatro canales y que se puede controlar por un control remoto que tiene solo dos teclas para aumentar o disminuir el canal. Los estados de nuestro sistema están representados por el número de canales que se pueden sintonizar, así pues, solo tendremos cuatro posibles estados (Número fijo de estados). En la Figura 1 se muestra un diagrama de nuestro sistema:

Figura 1. Ejemplo de FSM.

Ahora hagámonos una pregunta: ¿Qué nos produce un cambio en el estado de nuestro sistema?, Lo único que nos puede producir un cambio de estado (Canal sintonizado) es un cambio en las teclas de nuestro control remoto (Entradas del sistema). Observemos como actúa el sistema ante cambios de las entradas:

Si oprimimos la tecla de aumentar el canal, el televisor aumentará en uno la cadena sintonizada, por ejemplo, si estaba en el canal 1 pasará al 2, si estaba en el 2 al 3, del 3 al 4 y del 4 al 1 nuevamente.

Si oprimimos la tecla menos es canal diminuirá pasando del 4 al 3, del 3 al 2, del 2 al 1 y del 1 al 4 nuevamente. Si no oprimimos una tecla, el televisor debe permanecer en la cadena sintonizada actualmente y quedarse ahí hasta que se le ordene un cambio por medio del control remoto.

Note que el estado (canal sintonizado) al que pasará el sistema depende del estado actual (canal actual) y de las entradas (tecla del control remoto oprimida). Si las entradas no cambian el sistema no cambia su posición (esto no es necesariamente cierto para todas las máquinas de estado).

1.3. TABLAS Y DIAGRAMAS DE ESTADO

Existen varias formas de representar el comportamiento de las máquinas de estado, siendo los más utilizados las tablas y los diagramas de estado. Tomemos nuevamente el ejemplo del televisor y representemos en una tabla los estados actual y siguiente (Estado al que pasa el sistema ante un cambio de las entradas).

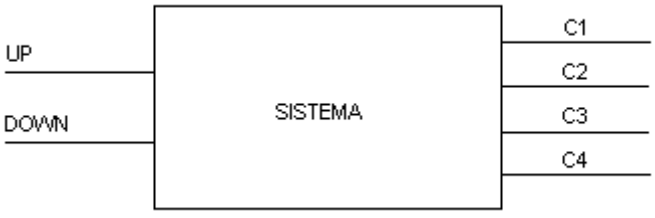


Figura 2. Entradas y salidas del Sistema.

La Figura 2 muestra el sistema como una caja negra en la que sólo se indican las entradas y las salidas. Supongamos que nuestro sistema tiene dos entradas que son las correspondientes a Adelantar (UP) y disminuir (DN) canal, y que tiene cuatro salidas C1, C2, C3, C4 que corresponden a los cuatro canales y que me indican cual canal se está sintonizando actualmente. En este punto debemos tomar varias decisiones:

¿Cuando se considera que una entrada esta activa?, Es decir, con que valor lógico de la entrada se produce un cambio y con cual no. En nuestro caso un valor lógico alto en las entradas producirá un cambio de estado, es decir, si UP = '1' el canal sintonizado aumentará o si DN = '1' el canal disminuirá. Otra decisión que debemos tomar y que se deriva de esta es: que sucede si por error las dos entradas son '1' simultáneamente, lo más conveniente es que el sistema conserve su estado ante esta posible situación.

El valor de las salidas para cada estado, en este ejemplo, un uno lógico en una salida indica que el canal ha sido seleccionado (por ejemplo, un uno en C1 indicara que actualmente el canal seleccionado es el 1), por lo tanto sólo una salida puede tener un valor lógico alto.

Una vez definido completamente el funcionamiento de nuestro sistema se procede a representarlo mediante una tabla de estados:

1.3.1. Tabla de Estados

Todas las condiciones posibles de las entradas

	Estado Siguiente (S*)			
Estado Actual (S)	UP =0, DN=0	UP=0,DN=1	UP=1,DN=0	UP=1,DN=1
Canal 1	Canal 1	Canal 4	Canal 2	Canal 1
Canal 2	Canal 2	Canal 1	Canal 3	Canal 2
Canal 3	Canal 3	Canal 2	Canal 4	Canal 3
Canal 4	Canal 4	Canal 3	Canal 1	Canal 4

Figura 3. Tabla de estados del sistema Televisor.

La Figura 3. Muestra una tabla de estados típica en la cual se resume el comportamiento del sistema. La tabla tiene tres secciones: El estado actual: Lista de todos lo posibles estados.

Posibles combinaciones de las entradas: El número de entradas del sistema determina el número de columnas de la tabla de estados. Así, si la máquina tiene n entradas, la tabla tendrá 2^n +1 Columnas.

El estado siguiente: Indica a que estado pasará la FSM cuando se presente una determinada entrada, Por ejemplo, Si UP=0 y DOWN = 1 y el estado actual es el canal 4 la máquina de estados irá al estado Canal 3.

Otra forma de representar el estado de las entradas es utilizando una convención en la que si la variable aparece negada entonces toma un valor de cero lógico, pero si no lo esta tiene un valor de uno lógico. Esto se hace para simplificar las tablas. Por ejemplo:

A: A = '1'

!A: A = 0;

Con lo que la tabla de estados se convierte en :

	Estado Siguiente				Salidas
Estado Actual	!UP.!DN	!UP.DN	UP.!DN	UP.DN	C1,C2,C3,C4
Canal 1	Canal 1	Canal 4	Canal 2	Canal 1	1,0,0,0
Canal 2	Canal 2	Canal 1	Canal 3	Canal 2	0,1,0,0
Canal 3	Canal 3	Canal 2	Canal 4	Canal 3	0,0,1,0
Canal 4	Canal 4	Canal 3	Canal 1	Canal 4	0,0,0,1

Tabla 1. Tabla de Estados del sistema Televisor.

1.3.2. Diagrama de Estados

Otra forma de representar el comportamiento de una máquina de estados es el diagrama de estados, este diagrama es una representación gráfica del funcionamiento del sistema.

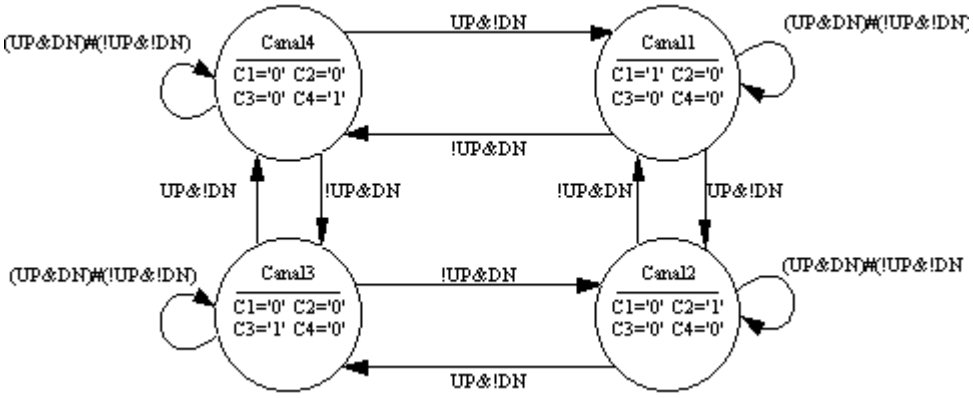


Figura 4. Diagrama de estados del sistema Televisor.

La Figura 4 muestra el diagrama de bloques del sistema televisor. Observamos que cada estado está representado por un círculo en el que se indica el nombre del mismo y el valor de las salidas. Las líneas que unen los estados representan el cambio que sufre la máquina cuando se cumple una determinada regla de transición. (La regla de transición normalmente se indican en las líneas que unen los estados). En esta figura se introduce una nueva nomenclatura para representar las funciones lógicas, el operador ***not*** se representa con el signo ***!***, la operación ***AND*** con el signo ***&*** y la ***OR*** con #.

Podemos observar que del estado Canal1, salen dos líneas: Una hacia el estado Canal2, lo que indica que la máquina pasará de Canal1 a Canal2 si UP = '1' y DN = '0' y se presenta un flanco adecuado en la señal de reloj; Otra hacia el estado Canal4, lo que indica que la máquina de estados pasará de Canal1 a Canal4 si UP = '0', DN = '1' y se presenta un flanco adecuado en la señal de reloj.

Así mismo tenemos dos líneas que llegan al Estado Canal1: Una proviene del estado Canal2, con lo que el sistema pasará de Canal2 a Canal 1 si UP = '0', DN = '1' y se presenta un flanco adecuado en la señal de reloj; y otra desde el estado Canal4. lo que hace que el sistema pase de Canal4 a Canal 1 si UP = '1' y DN = '0' y se presenta un flanco adecuado en la señal de reloj.

Por último existe una curva que sale del Canal1 y vuelve a entrar a Canal1, esto indica que la máquina mantendrá su estado si: (UP = '0' Y DN = '0') O (UP = '1' Y DN = '1').

1.4. SINTESIS DE FSM

En esta sección analizaremos la arquitectura de la FSM y el proceso de síntesis. Como vimos en el capítulo anterior la síntesis parte de una descripción funcional y llega a una descripción a nivel de compuertas.

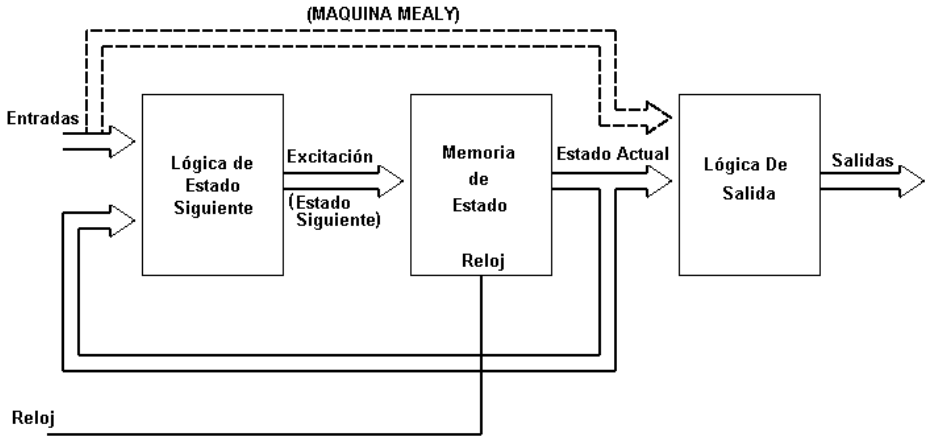


Figura 5. Estructura de una FSM.

1.4.1. Estructura de una FSM

La estructura general de una máquina de estados se muestra en la Figura 5. Como puede observarse existen tres bloques bien definidos:

Lógica de estado siguiente: Está encargada de generar las señales necesarias para producir un cambio de estado en la FSM (Estado Siguiente). Puede observarse que el estado siguiente depende del estado actual y de las entradas

Memoria de Estado: Normalmente esta formada por Flip-Flops tipo D o JK, los cuales tienen la misma señal de reloj. Las salidas de los Flip-Flops determinan el estado actual de la FSM, cada salida del Flip-Flop puede tomar dos valores distintos '1' o '0', por lo tanto se pueden tener dos estados con un Flip-Flop. Si tenemos N Flip-Flops tendremos 2^N estados.

Lógica de Salida: La lógica de salida esta encargada de producir los valores necesarios a la salida de la FSM, su la arquitectura esta basada en decodificadores.

Existen dos tipos de máquinas de estados: Moore: El estado siguiente depende del valor de las entradas y del estado actual; Y la salida depende únicamente del estado actual; y Mealy: Al igual que la máquina de Moore el estado siguiente depende del valor de las entradas y del estado actual, pero se diferencian en que la salida depende del estado actual y del valor de las entradas.

1.4.2. Proceso de Síntesis

El primer paso en el proceso de síntesis de una FSM y en general de cualquier sistema digital es la descripción del sistema ya sea mediante un algoritmo o de un diagrama de tiempos. El siguiente paso consiste en pasar la descripción funcional a un diagrama de estados para su posterior representación en tablas de estado y de salida. A continuación debemos reducir el número de estados (si es posible) utilizando algoritmos de minimización. Después debemos realizar la codificación de estados, es decir, asignarle a los estados un grupo único de valores que corresponden a los valores que tomarán los Flip-Flops. A continuación se debemos obtener las ecuaciones de estado siguiente y de salidas. El siguiente paso consiste en la elección del tipo de Flip-Flop que se va a utilizar para la implementación, recuerde que todos los Flip_Flops tienen diferentes ecuaciones de estado siguiente:

Tipo de F-F	Estado Siguiente (Q^*)
D	Q* = D
JK	Q* = J,!Q + !K.Q
T	Q* = !Q
SR	Q* = S + !R.Q

Tabla 2. Ecuaciones de Estado siguiente de los Flip-Flop.

Una vez elegido el Flip-Flop se procede a obtener las ecuaciones de las señales de excitación de los FFs. Después se de be realizar un proceso de minimización con las ecuaciones obtenidas anteriormente para realizar un diagrama de la implementación lógica de las ecuaciones. Finalmente debemos verificar el correcto funcionamiento del circuito, esto se logra simulando el circuito obtenido y si cumple con lo requerimientos se llevará al plano físico para realizar pruebas de tiempos. La figura 6 resume los pasos a seguir en el proceso de síntesis.

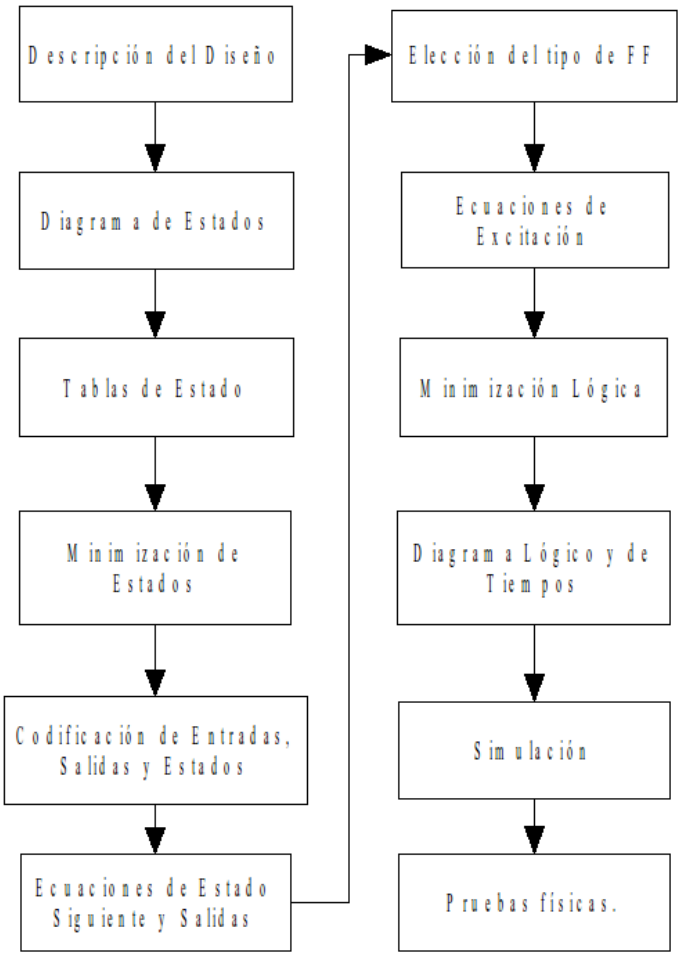


Figura 6. Diagrama de flujo del proceso de Síntesis para FSM

Para entender mejor el proceso de síntesis realizaremos paso a paso un sencillo ejemplo. Desarrollando todas las posibilidades de implementación para buscar la más óptima.

1.4.3. Control de Motor de Paso.

Un motor de paso a diferencia de los motores de Corriente Continua necesita una secuencia determinada en sus cuatro terminales para originar el giro de su rotor. La secuencia necesaria para controlar el motor es la siguiente:

	A	B	C	D
S1	V+	GND	V+	GND
S2	V+	GND	GND	V+
S3	GND	V+	GND	V+
S4	GND	V+	V+	GND

Para que el motor gire un paso (normalmente 1.8 grados) es necesario que se aplique S1 y luego S2, o S2 y S3 o S3 y S4 o S4 y S1. Si se desea que el motor gire cinco pasos se debe seguir la secuencia S1, S2, S3, S4, S5. La inversión del sentido de giro se logra invirtiendo la secuencia anterior, es decir, S1, S4, S3, S2, S1.

Diagrama de Caja Negra El primer paso consiste en realizar un diagrama de caja negra en el que se indiquen las entradas y salidas (Figura 7).

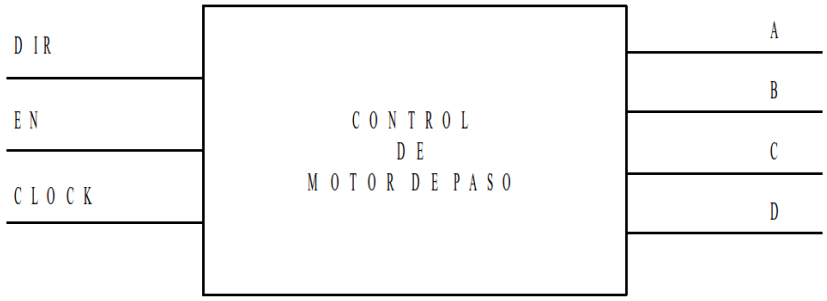


Figura 7. Diagrama de Caja Negra del Controlador de Motor de Paso.

A continuación realizaremos una breve descripción del funcionamiento del circuito. Como se observa en la Figura 7. Se tienen tres entradas:

DIR: Encargada de indicar la dirección de giro del motor. DIR = 0 Secuencia Directa (S1, S2, S3, S4), DIR = 1 Secuencia Invertida (S4, S3, S2, S1) EN: ENABLE, encargada de habilitar nuestro control Si EN = 1 el circuito realizará su función si EN = 0 el control conservará el último estado de las salidas.

CLOCK: Es el reloj del sistema y gobierna todas las transiciones entre estados. Y las cuatro salidas A, B, C, D son las encargadas de manejar los terminales del motor de paso.

Diagrama de Estado Una vez se conoce el funcionamiento del circuito, o las funciones que debe cumplir se procede a realizar el diagrama de estados del mismo. La Figura 8 muestra este diagrama para el controlador de motor de paso.

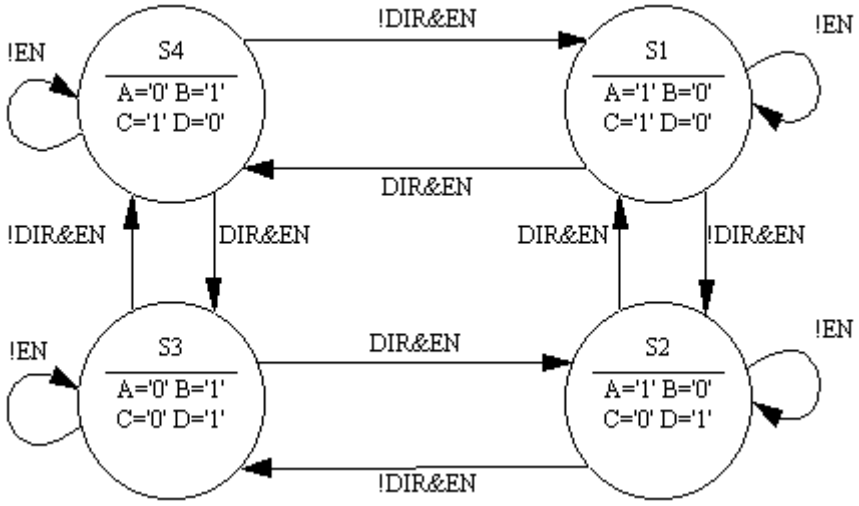


Figura 8 Diagrama de Estados del Controlador de Motor de Paso.

Tabla de estados Una vez realizado el diagrama de estados se procede a la realización de la tabla de estados y salidas. Dicha tabla para el controlador de motor de paso se presente a continuación:

	Estado Siguiente				Salidas
Estado Actual	!EN.!DIR	!EN.DIR	EN.!DIR	EN.DIR	A, B, C, D
S1	S1	S1	S2	S4	1, 0, 1, 0
S2	S2	S2	S3	S1	1, 0, 0, 1
S3	S3	S3	S4	S2	0, 1, 0, 1
S4	S4	S4	S1	S3	0, 1, 1, 0

Minimización de Estados El objetivo de la minimización de estados es la reducción del número de Flip-Flops necesarios para la implementación, reduciendo de este modo el costo de la misma. Sin embargo para reducir el número de Flip-Flops de un circuito es necesario reducir el número de

estados en múltiplos de 2. Por ejemplo, supongamos que tenemos 7 estados, para lo cual necesitamos 3 FFs, para utilizar sólo 2 FFs necesitamos reducir el número de estados de 7 a 4 o menos.

La minimización de estados se basa en la equivalencia funcional, por ejemplo, se dice que dos circuitos son equivalentes cuando proporcionan la misma salida ante los mismos cambios de entrada. Dos o más estados son equivalentes sí:

Ambos estados producen las mismas salidas ante igual cambio en las señales de entrada. Ambos estados tienen los mismos estados siguientes ante los mismos cambios de las señales de entrada.

En nuestro caso no tenemos equivalentes ya que todos tienen diferentes valores de salida y diferentes estados siguientes ante variaciones de las entradas.

Codificación de estados La codificación de estados consiste en la asignación de valores a las salidas de los FFs para cada estado, estos valores deben ser únicos para cada estado, es decir, no se deben repetir. Debido a que nuestra máquina tiene cuatro estados, tenemos 2 FFs y por lo tanto debemos asignar a cada estado un código formado por dos bits. Existen muchas posibles codificaciones para los cuatro estados, unas más eficientes que otras. En este libro no se tratarán las técnicas existentes para hallar la codificación óptima ya que esta función las realizan las herramientas CAD y además, se sale del objetivo de este libro. Para nuestro ejemplo utilizaremos la siguiente codificación:

	Q1	Q0
S1	0	0
S2	0	1
S3	1	0
S4	1	1

Donde Q1 y Q0 son las salidas del primer y segundo FF correspondientemente. Con esta asignación de estados la tabla de estados queda de la siguiente forma:

Estado Actual (S)		Estado Siguiente (S^*)						SALIDAS
		!EN		EN				
				!DIR		DIR		A, B, C, D
Q1	Q0	$Q1^*$	$Q0^*$	$Q1^*$	$Q0^*$	$Q1^*$	$Q0^*$	
0	0	0	0	0	1	1	1	1, 0, 1, 0
0	1	0	1	1	0	0	0	1, 0, 0, 1
1	0	1	0	1	1	0	1	0, 1, 0, 1
1	1	1	1	0	0	1	0	0, 1, 1, 0

Donde $Q1^*$ y $Q0^*$ representan los valores siguientes de las señales Q1 y Q0 respectivamente. Note que no se representaron los casos !EN.!DIR y !EN.DIR, esto se debe a que cuando la señal En tiene un valor lógico bajo la FSM conserva su estado sin importar el valor de DIR.

Ecuaciones de estado siguiente Una vez realizada la codificación de estados se procede a la obtención de las ecuaciones de estado siguiente $Q1^*$ y $Q0^*$. Para obtener estas ecuaciones debemos sumar todos los unos de la tabla de estados (suma de minitérminos). Para $Q1^*$ debemos observar todas las columnas correspondientes a $Q1^*$ y sumar los minitérminos asociados. Por ejemplo, la primera columna de Q1* correspondiente a la asociada con !EN, el primer minitérmino asociado es: !EN.Q1.!Q0 ya que está en la fila correspondiente a Q1 = 1 y Q0 = 0. Y el segundo !EN.Q1.Q0.

$Q1^*$ = !EN.Q1.!Q0 + !EN.Q1.Q0 + EN.!DIR.!Q1.Q0 + EN.!DIR.Q1.!Q0 + EN.DIR.!Q1.!Q0 + EN.DIR.Q1.Q0.

$Q0^*$ = !EN.!Q1.Q0 + !EN.Q1.Q0 + EN.!DIR.!Q1.!Q0 + EN.!DIR.Q1.!Q0 + EN.DIR.!Q1.!Q0 + EN.DIR.Q1.!Q0.

Estas ecuaciones deben pasar a través de un proceso de minimización para encontrar una implementación óptima.

Q1* = !EN.Q1(!Q0+Q1) + EN.(!DIR.(!Q1.Q0 + Q1.!Q0) + DIR (!Q1.!Q0 + Q1.Q0))
Q1* = !EN.Q1 + EN.(!DIR.Q1 Q0 + DIR.!(Q1 Q0))
Q1* = !EN.Q1 + EN.DIR (Q1 XOR Q0)
Q0* = !EN.Q0.(!Q1 + Q1) + EN.(!Q1.!Q0.(!DIR + DIR) + Q1.!Q0.(!DIR + DIR))
Q0* = !EN.Q0 + EN.(!Q1.!Q0 + Q1.!Q0)
Q0* = !EN.Q0 + EN.(!Q0.(!Q1+ Q1))
Q0* = !EN.Q0 + EN.!Q0
Q0* = EN XOR Q0
Las ecuaciones para las salidas son:
A = !Q1.!Q0 + !Q1.Q0 = !Q1 B = Q1.!Q0 + Q1.Q0 = Q1 C = !Q1.!Q0 + Q1.Q0 = !(Q1 Q0) D = !Q1.Q0 + Q1.!Q0 = Q1 Q0 A = !B = !Q1 C = !D = Q1 XOR Q0

Elección del tipo de Flip-Flop, ecuaciones de excitación y minimización Utilizando las ecuaciones obtenidas anteriormente para Q1* y Q0*, debemos escoger el tipo de Flip-Flop a utilizar. La siguiente tabla resume los valores necesarios en las entradas de los FFs para obtener los cambios indicados en sus salidas. Por ejemplo en un FF JK si Q tiene un valor de 0 lógico y se quiere que pase a tener un valor de 1 lógico es necesario que J = 1 y el valor de K no importa.

Q Q*	S R	J K	T	D
0 0	0 X	0 X	0	0
0 1	1 0	1 X	1	1
1 0	0 1	X 1	1	0
1 1	X 0	X 0	0	1

El diagrama de Karnough para la máquina de estados es el siguiente

	00	01	11	10
00	**0** 0	**0** 0	**1** 1	**0** 1
01	**0** 1	**0** 1	**0** 0	**1** 0
11	**1** 1	**1** 1	**1** 0	**0** 0
10	**1** 0	**1** 0	**0** 1	**1** 1

La región sombreada corresponde a los valores de las señales Q1* (en negrilla) y Q0*.
Para el FF D tenemos: Q* = D. Por lo tanto:
D1 = !EN.Q1 + EN.DIR (Q1 XOR Q0) D0 = EN XOR Q0
Para el FF JK debemos ordenar las ecuaciones obtenidas de la forma: Q* =J.!Q + !K.Q, por lo que para Q1:

	00	01	11	10
00	**0** X	**0** X	**1** X	**0** X
01	**0** X	**0** X	**0** X	**1** X
11	**X** 0	**X** 0	**X** 0	**X** 1
10	**X** 0	**X** 0	**X** 1	**X** 0

La región sombreada indican los valores que deben tener las señales J1(en negrilla) y K1. J1 = !Q1.!Q0.EN.DIR + !Q1.Q0.EN.!DIR = EN.!Q1(Q0 XOR DIR) K1 = Q1.!Q0.EN.DIR + Q1.Q0.EN.!DIR = EN.Q1.(Q0 XOR DIR)
Para Q0

	00	01	11	10
00	**0** X	**0** X	**1** X	**1** X
01	**X** 0	**X** 0	**X** 1	**X** 1

11	**X** 0	**X** 0	**X** 1	**X** 1
10	**0** X	**0** X	**1** X	**1** X

J0 = EN K0 = EN
Flip-Flop tipo T:
Para Q1:

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

T1 = !Q0.EN.DIR + Q0.EN.!DIR = EN.(Q0 XOR DIR)
Para Q0:

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

T0 = EN
Flip-Flop tipo SR.
Para Q1:

	00	01	11	10
00	**0** X	**0** X	**1** 0	**0** X
01	**0** X	**0** X	**0** X	**1** 0
11	**X** 0	**X** 0	**X** 0	**0** 1
10	**X** 0	**X** 0	**0** 1	**X** 0

S1 = !Q1.!Q0.EN.DIR + !Q1.Q0.EN.!DIR = EN.!Q1(Q0 XOR DIR)
R1 = Q1.!Q0.EN.DIR + Q1.Q0.EN.!DIR = EN.Q1.(Q0 XOR DIR)
Para Q0:

	00	01	11	10
00	**0** X	**0** X	**1** 0	**1** 0
01	**X** 0	**X** 0	**0** 1	**0** 1
11	**X** 0	**X** 0	**0** 1	**0** 1
10	**0** X	**0** X	**1** 0	**1** 0

S0 = EN.!Q0 R0 = EN.Q0

Del análisis anterior observamos que la implementación que ocupa el menor número de compuertas es la correspondiente a los Flip-Flops tipo T.

SIMULACION En la Figura 9 se muestra la implementación final de la máquina de Estados (Sin la Unidad de Salida) y en la Figura 10 la simulación correspondiente.

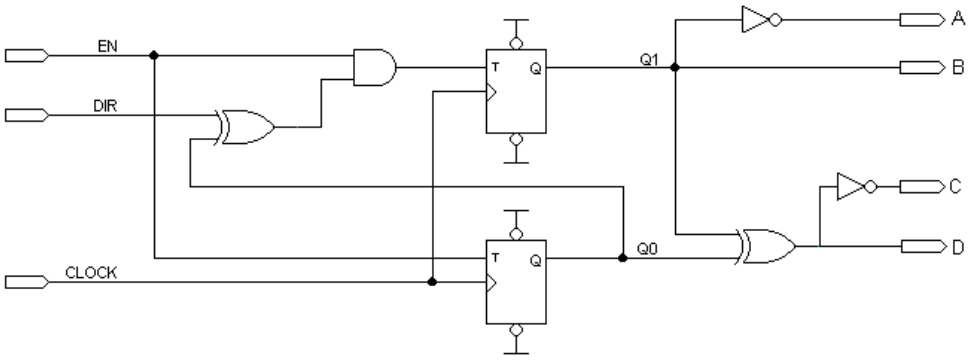


Figura 9. Diagrama a nivel de compuertas del controlador de Motor de Paso.

Como puede observarse en la Figura 9, la máquina de estados está formada por la lógica de estado siguiente (Compuertas XOR y AND), la memoria de estado (FFs tipo T con la misma señal de Reloj) y la lógica de salida, (Compuertas NOT y XOR) que en este caso corresponde al de una máquina de estados tipo MOORE.

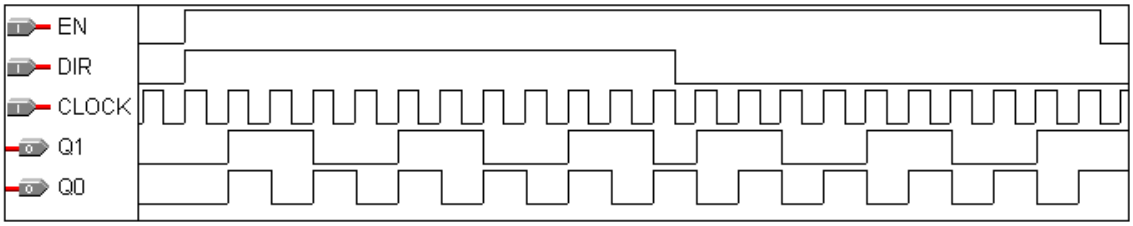


Figura 10. Simulación del Controlador de Motor de Paso.

1.5. DESCRIPCION VHDL DE UNA FSM

En esta sección se realizará la descripción en VHDL del controlador de motor de paso. Inicialmente se hará una descripción estructural, basándose en los resultados obtenidos en la sección 5.4.3.7. Y a continuación se utilizará la descripción funcional para la implementación del controlador.

1.5.1. DESCRIPCION ESTRUCTURAL

Recordemos que la descripción estructural en VHDL realiza la interconexión entre los componentes que forman el sistema. En nuestro caso (Ver Figura 9), primero debemos realizar la descripción de los componentes:

```
ENTITY and2 IS -- Debido a que AND es una palabra reservada del lenguaje
-- no puede utilizarse para nombrar una entidad
PORT(
     A : IN BIT;
     B : IN BIT;
     Y : OUT BIT);
     END and2;
ARCHITECTURE Y OF and2 IS
BEGIN
     Y <= A AND B;
END;
```

```
ENTITY xor2 IS -- Debido a que XOR es una palabra reservada del lenguaje
-- no puede utilizarse para nombrar una entidad
  PORT(
```

```
    A : IN BIT;
    B : IN BIT;
    Y : OUT BIT);
END xor2;

ARCHITECTURE XR OF xor2 IS
  BEGIN
    Y <= A XOR B;
  END;
```

```
INVERSOR ENTITY inv IS
  PORT(
    A : IN BIT;
    Y : OUT BIT);
END inv;
ARCHITECTURE NO OF inv IS
  BEGIN
    Y <= not (A);
  END;
```

```
FLIP FLOP TIPO T ENTITY FFT IS
  PORT(
    T,CLK : IN BIT;
    Q : BUFFER BIT);
END FFT;
ARCHITECTURE T OF FFT IIS
  BEGIN
   process (CLK) begin
    if (CLK'event and CLK = '1') then
     if (t = '1') then
      Q <= not (Q);
     else
      Q <= Q;
     end if;
    end if;
   end process;
  END;
```

Una vez realizada la implementación de los componentes, se procede a su unión. Para lograr esto se deben sintetizar las anteriores declaraciones, y sus archivos deben estar en el mismo directorio de trabajo.

```
CONTROL DE MOTOR DE PASO ENTITY StepMotor IS
  PORT(
    CLK, EN, DIR : IN BIT;
    AO, BO, CO, DO : BUFFER BIT);
END StepMotor;

ARCHITECTURE CONTROLLER OF StepMotor IS
  SIGNAL I1, T1, Q0N: BIT;
  COMPONENT AND2E
    PORT (A, B : IN BIT;
```

```
            Y : OUT BIT);
  END COMPONENT;
  COMPONENT XOR2
    PORT (A, B : IN BIT;
            Y : OUT BIT);
  END COMPONENT;
  COMPONENT INV
    PORT (A : IN BIT;
          Y : OUT BIT);
  END COMPONENT;
  COMPONENT FFT
    PORT(
    CLK, T: IN BIT;
        Q : OUT BIT);
  END COMPONENT;

  BEGIN
    X1: XOR2 port map(DIR, Q0N, I1);
    X2: AND2E port map(EN, I1, T1);
    X3: FFT port map(CLK, T1, BO);
    X4: FFT port map(CLK, EN, Q0N);
    X5: XOR2 port map(BO, Q0N, DO);
    X6: INV port map(BO, AO);
    X7: INV port map(DO, CO);
  END CONTROLLER;
```

1.5.2. DESCRIPCION FUNCIONAL

La descripción Estructural no es el mejor ejemplo de la potencialidad del VHDL, ya que como vimos, es necesario realizar todo el proceso de síntesis para obtener las ecuaciones de excitación de los Flip-Flops. Realizando la descripción funcional no es necesario preocuparse por escoger el Flip – Flop que reduzca el número de compuertas ni de minimizar las ecuaciones booleanas obtenidas, todos estos procesos los realizan automáticamente las herramientas CAD disponibles comercialmente. El código VHDL del controlador del motor de paso se muestra a continuación, y en la Figura 11 se muestra el resultado de la simulación.

```
library ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY FSMF IS
  PORT(
    clk          : IN std_logic;
    EN, DIR      : IN std_logic;
    A, B, C, D : OUT std_logic);
END FSMF;

ARCHITECTURE funcional OF FSMF IS
TYPE estados IS (S1, S2, S3, S4);
SIGNAL state : estados;

BEGIN
  PROCESS (clk)
   BEGIN
    IF (clk'event and clk = '1') then
     case state is
      when S1 => A <= '1'; B <= '0'; C <= '1'; D <= '0';
       IF (EN = '0') then
```

```
          state <= S1;
         ELSIF (DIR = '0') then
          state <= S2;
         ELSE
          state <= S4;
         END IF;

        when S2 => A <= '1'; B <= '0'; C <= '0'; D <= '1';
         IF (EN = '0') then
          state <= S2;
         ELSIF (DIR = '0') then
          state <= S3;
         ELSE
          state <= S1;
         END IF;

        when S3 => A <= '0'; B <= '1'; C <= '0'; D <= '1';
         IF (EN = '0') then
          state <= S3;
         ELSIF (DIR = '0') then
          state <= S4;
         ELSE
          state <= S2;
         END IF;
        when S4 => A <= '0'; B <= '1'; C <= '1'; D <= '0';
         IF (EN = '0') then
          state <= S4;
         ELSIF (DIR = '0') then
          state <= S1;
         ELSE
          state <= S3;
        END IF;
       END CASE;
      END IF;
   END PROCESS;
END funcional;
```

El código para la realización del Test Bench es:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY testbench IS
END testbench;
ARCHITECTURE behavior OF testbench IS
  COMPONENT fsmf
  PORT(
    clk : IN std_logic;
    EN : IN std_logic;
    DIR : IN std_logic;
    A : OUT std_logic;
    B : OUT std_logic;
    C : OUT std_logic;
    D : OUT std_logic
    );
  END COMPONENT;
  SIGNAL clk : std_logic;
  SIGNAL EN : std_logic;
  SIGNAL DIR : std_logic;
  SIGNAL A : std_logic;
```

```
  SIGNAL B : std_logic;
  SIGNAL C : std_logic;
  SIGNAL D : std_logic;
constant ncycles : integer := 40;
constant halfperiod : time  := 5 ns;
BEGIN
  uut: fsmf PORT MAP(
    clk => clk,
    EN => EN,
    DIR => DIR,
    A => A,
    B => B,
    C => C,
    D => D);
-- Generacion del Reloj
Clock_Source: process
begin
for i in 0 to ncycles loop -- Genera ncyclos de periodo 10 ns
clk <= '0';
wait for halfperiod;
clk <= '1';
wait for halfperiod;
end loop;
wait;
end process Clock_Source;
tb : PROCESS
  BEGIN
for i in 1 to ncycles/4 loop -- Durante ncyclos/4 genera las diferentes
wait until Clk='1' and Clk'event; -- Combinaciones de EN y DIR
EN <= '0';
DIR <= '0';
end loop;
for i in 1 to ncycles/4 loop
wait until Clk='1' and Clk'event;
EN <= '0';
  DIR <= '1';
end loop;
for i in 1 to ncycles/4 loop
wait until Clk='1' and Clk'event;
EN <= '1';
  DIR <= '0';
end loop;
for i in 1 to ncycles/4 loop
wait until Clk='1' and Clk'event;
EN <= '1';
  DIR <= '1';
end loop;
END PROCESS;
END;
```

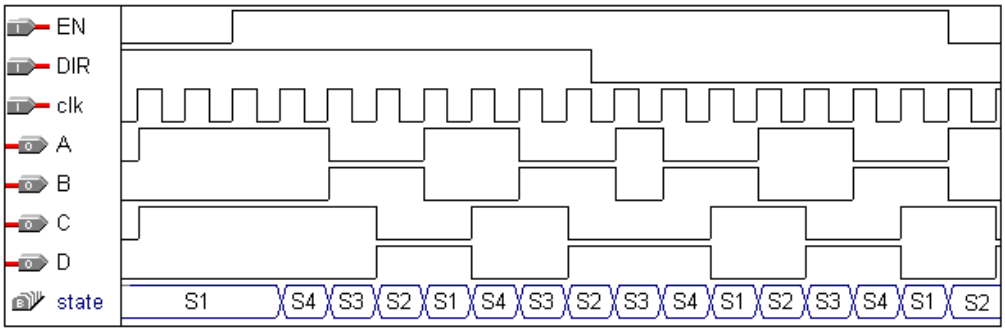


Figura 11. Simulación de la descripción funcional del controlador de motor de paso.

Con la descripción funcional no es necesario preocuparse por el tipo de Flip-Flop ni por los procesos de minimización.

Normalmente las herramientas CAD proporcionan información sobre resultado de ls síntesis del diseño, las ecuaciones implementadas en un CPLD por una de estas herramientas es:

clk : INPUT;
DIR : INPUT;
EN : INPUT;
A = DFFE(state˜2 $ VCC, GLOBAL(clk), VCC, VCC, VCC);
B = DFFE(!state˜2 $ VCC, GLOBAL(clk), VCC, VCC, VCC);
C = DFFE(!state˜1 $ state˜2, GLOBAL(clk), VCC, VCC, VCC);
D = DFFE(state˜1 $ state˜2, GLOBAL(clk), VCC, VCC, VCC);
state˜1 = TFFE(EN, GLOBAL(clk), VCC, VCC, VCC);
state˜2 = TFFE(_EQ001, GLOBAL(clk), VCC, VCC, VCC);
_EQ001 = !DIR & EN & state˜1 # DIR & EN & !state˜1;

En donde se puede observar que se llegó a las mismas ecuaciones obtenidas anteriormente. Observe que se utilizaron 2 Flip-Flops tipo T uno de los cuales tiene a la señal EN como entrada (primera seña que aparece dentro de los paréntesis después de TFFE.). Mientras que el otro tiene a la señal _EQ001 conectada a su entrada T. La ecuación para _EQ001 es:

!DIR & EN & state˜1 # DIR & EN & !state˜1 = EN.((!DIR.sate˜1) + (DIR.!state˜1)

Como vimos en capítulos anteriores la arquitectura de un CPLD no posee compuertas XOR, por lo tanto las ecuaciones son de la forma de suma de productos. Lo interesante es observar el ahorro de tiempo asociado a la utilización del VHDL como herramienta de diseño.

2. Máquinas de Estados Algorítmicas (ASM)

Una máquina de estados algorítmica (ASM) permite la implementación de cualquier tipo de algoritmo; está compuesta por el *camino de datos* y el *control*. El camino de datos, como su nombre lo indica, modifica los datos o variables del algorítmo; mientras que el control determina cuando son modificados. Los pasos que se realizan para el diseño e implementación de una máquina de estados algorítmica son los siguientes:

1. Elaboración del diagrama de flujo del algoritmo.
2. Identificación de los componentes del camino de datos.
3. Identificación de las señales necesarias para controlar el camino de datos e interconexión.
4. Especificación de la unidad de control utilizando diagramas de estado.
5. Implementación de los componentes del camino de datos y de la unidad de control utilizando lenguajes de descripción de hardware.
6. Simulación y pruebas

2.1. Multiplicación de números binarios usando una ASM

El algoritmo de multiplicación que se implementará se basa en productos parciales; el primer producto parcial siempre es cero (ver Figura 1, a continuación se realiza la multiplicación iniciando con el bit menos significativo del multiplicador, el resultado de la multiplicación se suma al primer producto parcial y se obtiene el segundo producto parcial; si el bit del multiplicador es '0' no se afecta el contenido de PP, por lo que no se realiza la suma. A continuación se realiza la multiplicación del siguiente bit (a la izquierda del LSB) y el resultado se suma al producto parcial dos pero corrido un bit a la izquierda, esto para indicar que la potencia del siguiente bit tiene un grado más; este corrimiento se debe realizar ya que si un número binario se multiplica por 2 el resultado es el mismo número corrido a la izquierda, por ejemplo:

15 (1111) X 2 = 11110 = (30); 15 (1111) X 4 = 111100 = (60)

Este proceso continúa hasta completar los bits del multiplicador y el último producto parcial es el resultado.

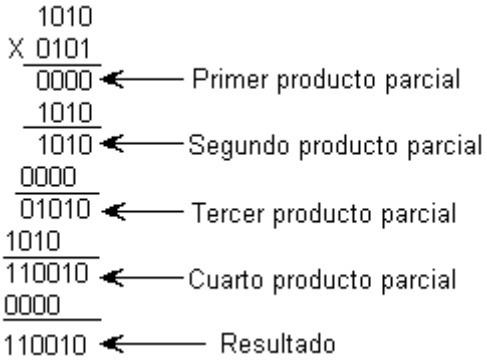


Figura 1: Multiplicación de numeros binarios usando productos parciales.

2.1.1. Diagrama de Flujo

En la Figura 2 se muestra un diagrama de flujo para la implementación de este algoritmo. El primer paso para realizar la multiplicación es hacer el producto parcial (PP) sea igual a cero, a continuación se realiza una verificación del bit menos significativo del multiplicador, esto se hace para sumar únicamente los resultados que no son cero. En este caso se utiliza un corrimiento a la izquierda para obtener el siguiente bit del multiplicador, si por ejemplo al número *101**0*** se le realiza un corrimiento a la derecha se obtiene el número *010**1***, con lo que el bit menos significativo corresponde al segundo bit de *1010*, si se realiza otro corrimiento a la derecha se obtiene *001**0*** y de nuevo el bit menos significativo corresponde al tercer bit de *1010*, al realizar de nuevo un corrimiento se obtiene *000**1***, con lo que tendríamos todas las cifras del multiplicador de forma consecutiva en el mit menos significativo. Cuando se realiza un nuevo corrimiento el resultado es *0000* lo que indica que el producto parcial no puede cambiar y podemos terminar el algortimo. Este método para finalizar el algoritmo produce que el número de iteraciones depende del valor del multiplicador; otra forma de terminar el algoritmo sin que dependa del valor del multiplicador se obtiene al contar el número de bits del multiplicador y realizar el corrimiento n veces, donde n es el número de bits del multiplicador.

Para indicar que cada vez que se toma un bit del multiplicador, este tiene una potencia mayor que el bit anterior, debemos multiplicar el resultado por la base, la cual es 2 en este caso; como se mencionó anteriormente, multiplicar por 2 equivale a realizar un corrimiento a la izquierda, por lo que siempre que se tome un nuevo bit del multiplicador debemos correr a la izquierda el multiplicando.

Una vez conocido el funcionamiento del sistema se procede a realizar el diagrama de caja negra de entradas y salidas. En la Figura 4 se muestra el multiplicando y el multiplicador (A y B), señales de m bits cada una, el resultado de la multiplicación PP (Bus de 2m Bits), la señal de reloj (CLOCK). Las señales INIT y DONE se utilizan para iniciar el proceso de multiplicación e indicar

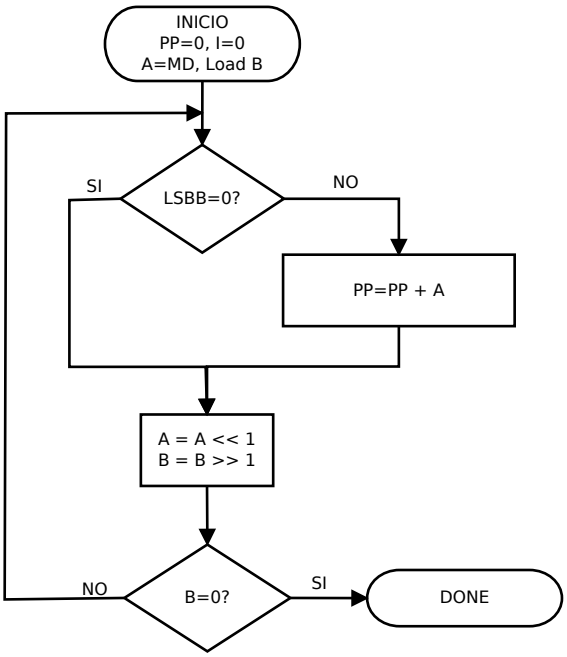


Figura 2: Diagrama de flujo para la multiplicación de numeros binarios.

que el resultado está disponible respectivamente; es importante que todo sistema digital posea la forma de interactuar con el exterior, ya que sin ello el sistema carecería de utilidad.

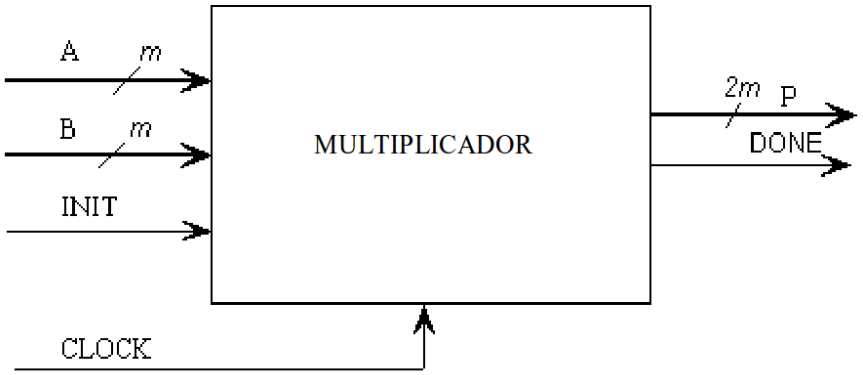


Figura 3: Diagrama de caja negra para el multiplicador de numeros binarios.

2.1.2. Identificación de los componentes del camino de datos

A continuación se identifican los componentes del camino de datos, esto se realiza recorriendo el diagrama de flujo para encontrar las operaciones que se realizan.

En la figura **??** se resaltan las operaciones que se deben realizar para la correcta operación del algorítmo; la primera es una operación de acumulación correspondiente a $PP = PP + A$; la segunda operación que encontramos son los dos corrimientos a la izquierda y derecha del multiplicando (A) y el multiplicador (B) respectivamente, estas operaciones se realizan al mismo tiempo pero en módulos diferentes; el último módulo es un comparador que indica que el multiplicador es igual a cero, indicando que el algoritmo puede finalizar.

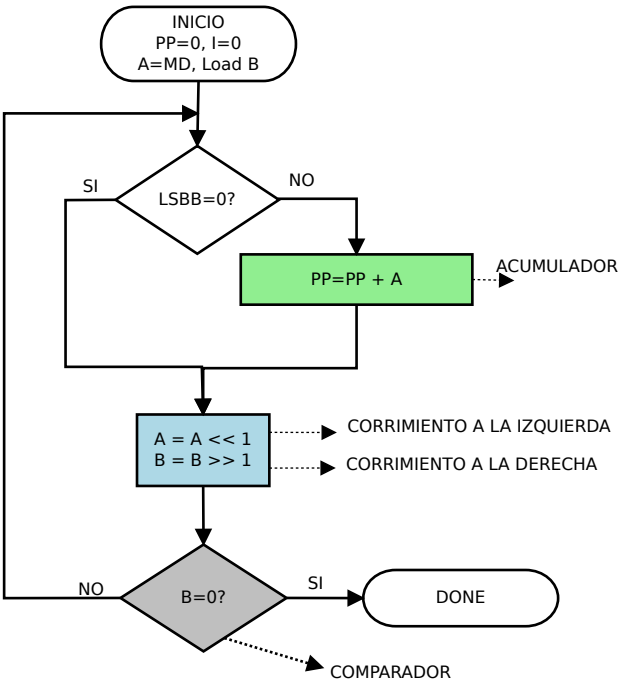


Figura 4: Identificación de los componentes del camino de datos para el multiplicador de numeros binarios.

2.1.3. Identificación de las señales de control e interconexión del camino de datos

En la figura 5 se muestra la interconexión de los componentes del camino de datos y las señales que lo controlan.

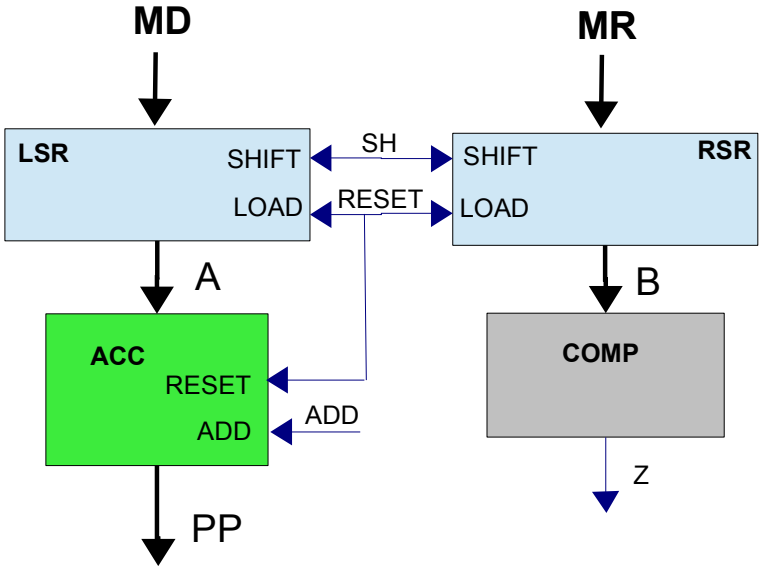


Figura 5: Identificación de las señales de control e interconexión del camino de datos.

La primera operación que aparece en el diagrama de flujo es la del acumulador, el cual acumula el valor de la salida del registro de corrimiento que almacena el multiplicando, de aquí obtenemos la conexión entre el registro de corrimiento (LSR) a la izquierda y el acumulador (ACC). La segunda operación que aparece es la de los registros de corrimiento, por lo que los valores del multiplicando y multiplicador deben cargarse para su posterior corrimiento a las unidades de corrimiento a la izquierda y derecha respectivamente. La salida del corrimiento a la derecha del multiplicador es comparada en cada ciclo para determinar si se llegó al final del algoritmo, por lo que la entrada del comparador es la salida del registro de corrimiento del multiplicador.

Para determinar las señales de control de cada componente del camino de datos, se debe identificar su función y las operaciones que debe realizar; los registros de corrimiento deben permitir la carga de un valor inicial y el corrimiento de las mismas, esto se realiza con las señales *LOAD* y *SHIFT* respectivamente; el acumulador debe tener la posibilidad de inicializarse en cero y una señal para que sume el valor de la entrada al que tiene almacenado, esto se hace con las señales *RESET* y *ADD*; por último el comparador debe proporcionar una señal que indique que el valor de su entrada es igual a cero, *Z* en este caso.

Aunque es posible que la máquina de control maneje todas las señales de control del camino de datos, es mejor aguparlas de acuerdo a su activación; esto es, si una señal se activa al mismo tiempo que otra, se puede utilizar una señal que las controle a ambas. Para esto se utiliza el diagrama de flujo y se observa en que momento se realizan las operaciones: Se observa que se cargan los valores de los registros de corrimiento y se inicializa en cero el acumulador únicamenet al comenzar el algoritmo y durante la ejecución del mismo no se vuelve a relizar esta operación, por este motivo utilizaremos la misma señal (*RESET*) para cargar los registros de desplazamiento e inicializar en cero el acumulador; la señal que controla el momento en que el acumulador se incrementa es única, ya que no se realiza ninguna operación en ese punto del algoritmo y en este caso recibe el nombre de *ADD*; las operaciones de corrimiento se realizan en el mismo lugar, por lo que se puede utilizar una señal común, que en este caso llamaremos *SH*; por último la salida del comparador *Z* y el bit menos significativo de B *LSB* son señales de salida del camino de datos que le darán a la unidad de control la información necesaria para tomar la acción adecuada en los bloques de decisión.

2.1.4. Especificación de la unidad de control utilizando diagramas de estado

Una vez que se conoce el camino de datos, las señales que lo controlan y las señales que ayudarán a la unidad de control a tomar decisiones, se procede con la especificación de la unidad de control, la cual, es una máquina de estados finitos, por lo que la mejor forma de especificarla es utilizando un diagrama de estados; en la figura 6 se muestra la relación entre el diagrama de flujo y el diagrama de estados.

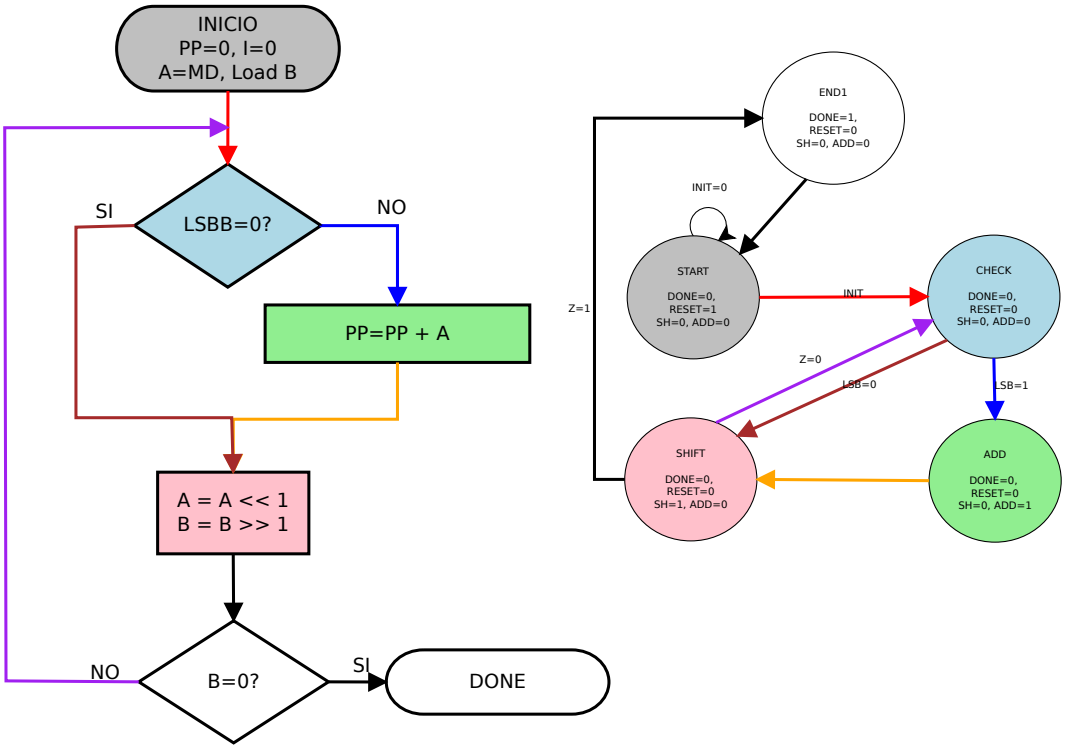


Figura 6: Diagrama de estados de la unidad de control del multiplicador binario.

Como se puede observar, existe una relación muy estrecha entre el diagrama de estados y el diagrama de flujo, cada operación del diagrama de flujo corresponde a un estado de la máquina de control y las transiciones entre ellos son idénticas, observe las líneas del mismo color en la figura 6

La máquina de estados debe iniciar en START y se queda en este estado siempre que la señal INIT tenga un valor de '0', En el estado INIT la señal RESET = '1', con lo que el valor del acumulador se hace cero y se cargan los registros A y B. Cuando INIT = '1', entramos al estado CHECK el cual evalúa la señal LSB, sí LSB = '0', no se debe realizar la suma de MD, pero si se debe realizar un corrimiento para obtener el siguiente bit del multiplicador y realizar el corrimiento necesario en MD. Si LSB = '1' se debe sumar el valor de las salidas de LSR al valor del acumulador, y después se debe realizar un corrimiento. En el estado ADD se hace la salida ADD = '1' para que el valor a la entrada del acumulador se sume al valor almacenado en él. En el estado SHIFT se realiza el corrimiento de RSR y LSR haciendo el valor de la señal SH = 1.

Para verificar el buen funcionamiento del diagrama de estado debemos realizar una prueba de escritorio: Supongamos que tenemos A = 7 y B = 5 y que INIT = 1:

ESTADO	SH	LSR	RSR	Z	LSB	ADD	DONE	ACC
CHECK	0	00000111	0101	0	1	0	0	00000000
ADD	0	00000111	0101	0	1	1	0	00000111
SHIFT	1	00001110	0010	0	0	0	0	00000111
CHECK	0	00001110	0010	0	0	0	0	00000111
SHIFT	1	00011100	0001	0	1	0	0	00000111
CHECK	0	00011100	0001	0	1	0	0	00000111
ADD	0	00011100	0001	0	1	1	0	00100011
SHIFT	1	00111000	0000	1	0	0	0	00100011
CHECK	0	00111000	0000	1	0	0	0	00100011
END1	0	00111000	0000	1	0	0	1	00100011
START	0	00000111	0101	0	1	0	0	00000000

Como puede observarse el resultado es correcto (35), en la tabla las casillas sombreadas corresponden a las señales que cambian de un estado a otro.

2.1.5. Implementación de los componentes del camino de datos y de la unidad de control

Existe abundante literatura sobre el uso de lenguajes de descripción de hardware para la implementación de sistemas digitales; por este motivo, en este libro no se presentará el código que implementa los diferentes módulos que hacen parte de las máquinas de estado algorítmicas estudiadas.

Es muy importante anotar la importancia de la portabilidad del código, como es bien sabido existen varias empresas que suministran entornos de desarrollo que permiten la entrada de diseño utilizando diferentes medios; las herramientas gráficas utilizados por ellos no son compatibles entre sí, lo que hace imposible el paso de un diseño implementado en una herramienta gráfica a otra de otro fabricante; sin embargo, todas las herramientas aceptan texto con el estándard del lenguaje utilizad; por esto, se recomienda utilizar únicamente entrada de texto en las descripciones.

2.1.6. Simulación

Como se mencionó anteriormente, es posible realizar las simulaciones utilizando las herramientas gráficas de cada uno de los entornos de desarrollo que proporcionan los fabricantes de dispositivos lógicos programables, sin embargo, su uso dificulta la portabilidad del diseño. Por este motivo, se recomienda el uso de *testbench* escritos con el lenguaje estándard. Como parte del proceso de diseño, cada módulo debe ser simulado antes de ser integrado en la descripción de más alto nivel.

Es bueno tener en cuenta los diferentes niveles de simulación que se pueden realizar a un sistema bajo prueba; la simulación más rápida es la que tiene en cuenta únicamente el lengiaje de descripción de hardware utilizado, sin embargo, no es posible garantizar que los resultados del circuito sintetizado sean los mismos que la simulación del lenguaje; por esto, existe la simulación

post-síntesis, en la que se simula el RTL (lógica de transferencia de registros) o las compuertas lógicas básicas obtenidas del proceso de síntesis, esta simulación garantiza que el circuito obtenido del proceso de síntesis se comporta como lo deseamos; el tercer nivel de simulación se obtiene cuando se adiciona un modelo de tiempo al diagrama de compuertas del nivel anterior, en este nivel, se tienen en cuenta las capacitancias de carga y la capacitancia de los caminos de interconexión para obtener el retardo de cada elemento del circuito, esta simulación es la más precisa y permite conocer la velocidad máxima a la que puede operar el sistema, esta simulación en algunos entornos de desarrollo recibe el nombre de *simulación post place & route*.

2.1.7. Pruebas

Aunque la simulación es una buena herramienta para la detección de errores, es necesario realizar una prueba sobre el circuito configurado en el dispositivo lógico programable, para esto existen dos opciones: realizar el montaje de la aplicación y probar la funcionalidad del dispositivo configurado, dependiendo de la complejidad del sistema esta puede ser una tarea tediosa; la segunda opción es utilizar el puerto JTAG para aplicar los vectores de prueba y capturar los resultados, este proceso se describirá en la siguiente sección.

2.2. Implementación de un divisor de n bits sin signo

El proceso de división de números binarios es similar al de números decimales: Inicialmente se separan dígitos del Dividendo de izquierda a derecha hasta que la cifra así formada sea mayor o igual que el divisor. En la figura 7 separamos el primer dígito de la derecha (0) y le restamos el divisor (la operación de resta se realizó en complemento a dos), el resultado de esta operación es un número negativo (los números negativos en representación complemento a dos comienzan por 1). Esto indica que el número es menor que el divisor, por lo tanto, colocamos un cero en el resultado parcial de la división (este dígito será el más significativo) y separamos los dos primeros dígitos (00) y repetimos el proceso.

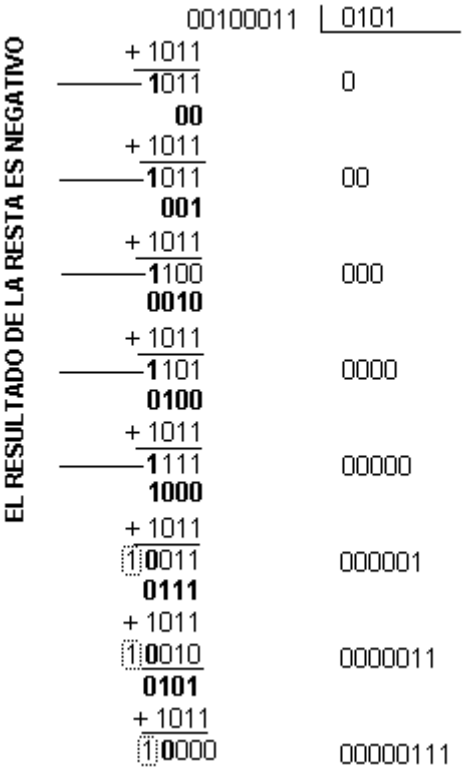


Figura 7: División de numeros binarios.

Sólo hasta el sexto resultado parcial obtenemos un cero en la primera cifra de la resta (recuerde que en complemento a dos los números tienen una longitud fija en nuestro caso 4 bits, sí una operación provoca un bit adicional este se descarta, los bits descartados se encerraron en líneas punteadas en la Figura 7), lo que indica que el número es mayor o igual que el divisor, en este caso, se coloca un '1' en el resultado parcial y se conserva el valor de la operación de resta, el cual se convierte en el nuevo residuo parcial, este proceso se repite hasta haber "bajado " todos los dígitos del dividendo. En la Figura 24 se muestra el algoritmo de división de dos números sin signo.

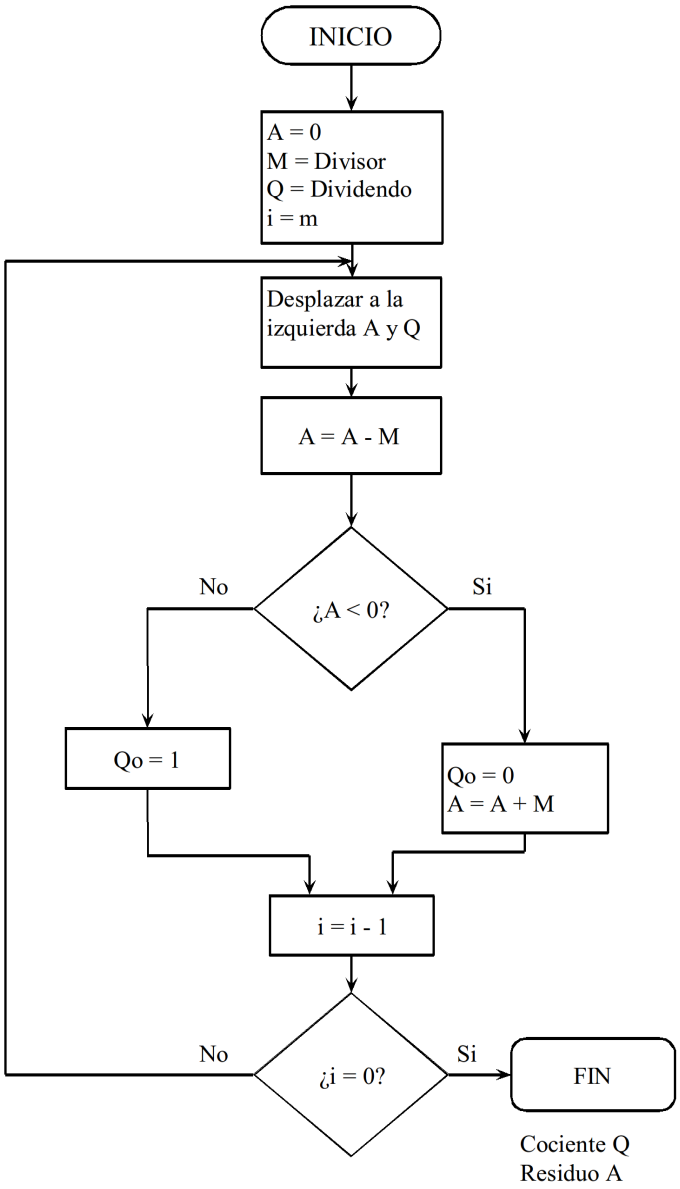


Figura 8: Algoritmo para la división de numeros binarios.

Figura 24. Diagrama de flujo para la división de números binarios sin signo.

Primera Implementación: El divisor de N Bits se implementará de dos formas: La primera siguiendo un procedimiento similar al del multiplicador y después se realizará una implementación directa del algoritmo de división utilizando un package propio.

Del diagrama de flujo podemos deducir que los bloques necesarios para la implementación del divisor son:

Un registro de corrimiento a la izquierda con precarga para A (LSR): El registro de corrimiento debe tener precarga ya que cuando es necesario modificar el valor de este registro.

- Un registro de corrimiento a la izquierda para Q (LSRQ).

- Un Sumador Restador (ADDSUB).
- Un contador descendente (DEC): El cual está encargado de controlar el número de corrimientos.
- Una unidad de control (CONTROLL)

En la siguiente figura se muestra el diagrama de entradas y salidas del divisor y el diagrama de bloques del mismo.

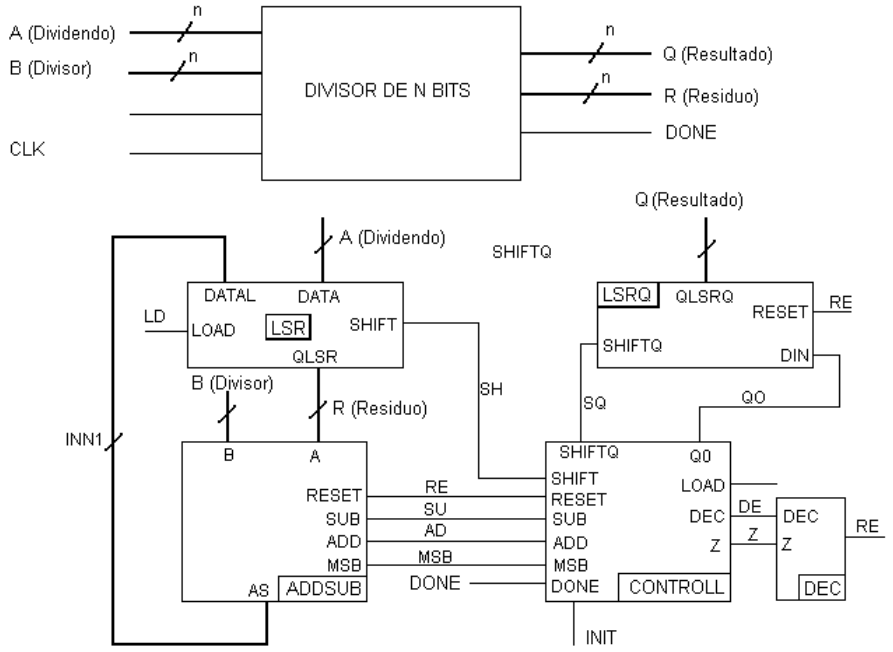


Figura 25. Diagrama de bloques del divisor de N bits.

Con el diagrama de bloques anterior podemos generar la descripción estructural del divisor:

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
Entity divisor is
Generic(width:natural:=7);
Port(
A,B      : IN STD_LOGIC_VECTOR(width downto 0);
Init,clk  : IN STD_LOGIC;
D,R      : BUFFER STD_LOGIC_VECTOR(width downto 0);
Done     : OUT STD_LOGIC);
End Divisor;
Architecture estructural of divisor is
component controll
Port(
Clk,Init, msb, Z    :IN  STD_LOGIC;
shift,reset,q0,done  :OUT STD_LOGIC;
sub,add,dec,load,sq  :OUT STD_LOGIC);
End Component;
Component lsrq
GENERIC(width:natural:=7);
PORT(
clk,shift,DIN,reset: IN  STD_LOGIC;
Qlsrq:     BUFFER STD_LOGIC_VECTOR(width downto 0));
END Component;
Component lsr
GENERIC(width:natural:=7);
```

PORT(
clk,shift,load,reset: IN STD_LOGIC;
Data,DataL: IN STD_LOGIC_VECTOR(width downto 0);
Qlsr: BUFFER STD_LOGIC_VECTOR(width downto 0));
END Component;
Component Decrement
Port(
Clk,Init,dec : IN STD_LOGIC;
Z : OUT STD_LOGIC);
End Component;
Component addsub
GENERIC(width:natural:=7);
PORT(
clk,reset,add,sub: IN STD_LOGIC;
msb: OUT STD_LOGIC;
A,B : IN STD_LOGIC_VECTOR(width downto 0);
AS: BUFFER STD_LOGIC_VECTOR(width downto 0));
END Component;
Signal ld,sh,re,ad,su,Z,msb,de,q0,sq : STD_LOGIC;
Signal INN1 :STD_LOGIC_VECTOR(width downto 0);
Begin
A1: lsr
Generic map(width)
Port Map(clk,sh,ld,re,A,INN1,R);
A2: lsrq
Generic map(width)
Port Map(clk,sq,q0,re,D);
A3: Decrement
Port Map(Clk,re,de,Z);
A4: AddSub
Generic map(width)
Port Map(clk,re,ad,su,msb,R,B,INN1);
A5: Controll
Port Map(Clk,Init,msb,Z,sh,re,q0,done,su,ad,de,ld,sq);
End Estructural;

CONTROLL

A continuación debemos realizar la descripción funcional de cada uno de los componentes. Comenzando por la unidad de control. En la siguiente figura se muestra el diagrama de estados de la misma:

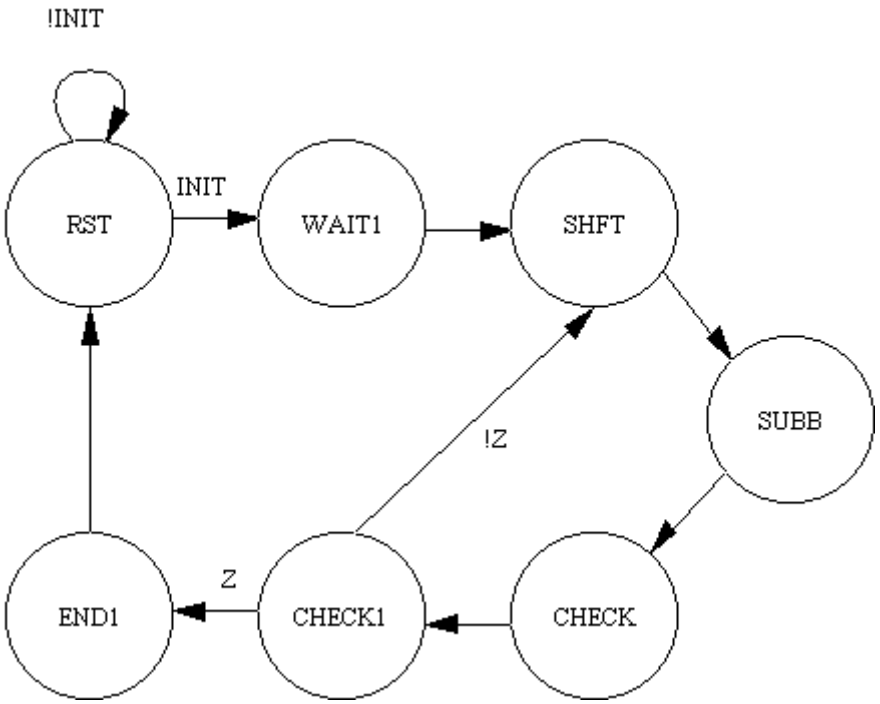


Figura 26. Diagrama de estados de la unidad de control.

Como puede observarse en el diagrama de estados primero se realiza un corrimiento del dividendo seguido por una operación de resta, en este diseño no se realiza la suma para restaurar el registro A, ya que A no cambia hasta que se cargue el registro. Por lo tanto solo se cargará el registro con el resultado de la resta cuando el resultado de la misma sea positiva.

La descripción funcional del módulo controll es la siguiente:

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
Entity Controll is
Port(
Clk,Init, msb, Z    : IN STD_LOGIC;
shift,reset,q0,done  : OUT STD_LOGIC;
sub,add,dec,load,sq  :OUT STD_LOGIC);
End Controll;
Architecture funcional of Controll is
Type state is (rst, shft, subb, check, end1, check1, Wait1);
signal next_state,current_state : state;
Begin
Process (Init,msb,Z, current_state)
Begin
Case current_state is
When rst => shift <= '0';reset <= '0';q0 <= '0';sub <= '0';
add <= '0';dec <='0'; load <= '0';done <= '0';sq <= '0';
if init = '1' then
next_state <= wait1;
else
next_state <= rst;
End If;
When wait1 => shift <= '0';reset <= '1';q0 <= '0';sub <= '0';
add <= '0';dec <='0'; load <= '0';done <= '0';sq <= '0';
next_state <= shft;
When shft => shift <= '1';reset <= '0';sub <= '0';add <= '0';
dec <='1'; load <= '0';done <= '0';sq <= '1';
next_state <= subb;
```

```
When subb => shift <= '0';sub <= '1';reset <= '0';add <= '0';
dec <='0'; load <= '0';done <= '0';q0<='0';sq <= '0';
next_state <= check;
when check => shift <= '0';sub <= '1';reset <= '0';add <= '0';
dec <='0';done <= '0';sq <= '0';
if msb = '1' then
q0 <= '0';
load <= '0';
else
q0 <= '1';
load <= '1';
end if;
next_state <= check1;
when check1 => shift <= '0';sub <= '0';add <= '0';reset <= '0';
dec <='0';done <= '0';sq <= '0';
if Z = '1' then
next_state <= end1;
else
next_state <= shft;
end if;
when end1 => shift <= '0';sub <= '0';add <= '0';dec <='0';
reset <= '0'; load <= '0';done <= '1';sq <= '1';
next_state <= rst;
when others =>
shift <= '0';
sub <= '0';
add <= '0';
dec <='0';
reset <= '0';
load <= '0';
done <= '0';
sq <= '0';
q0 <= '0';
next_state <= rst;
End Case;
End Process;
Process (clk)
Begin
if (clk'event and clk = '0') then
current_state <= next_state;
End if;
End Process;
End funcional;
```

La simulación de este módulo se muestra a continuación:

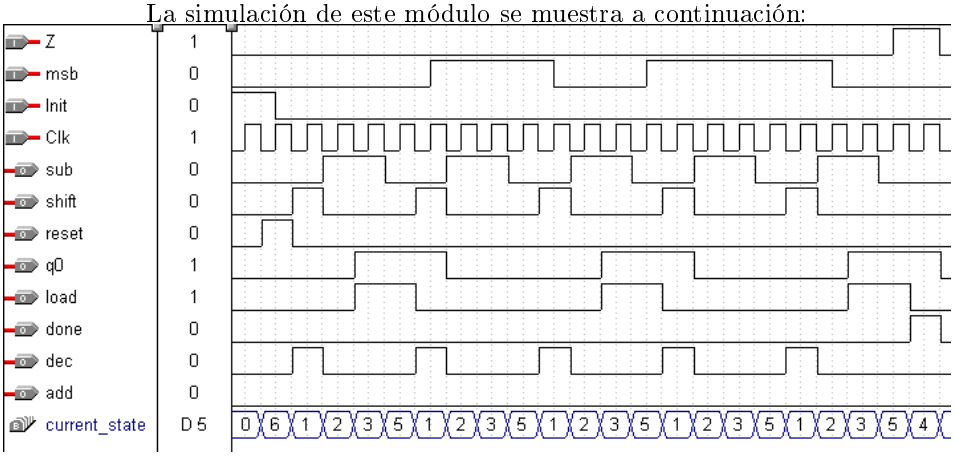


Figura 27 Simulación de la unidad de control.

LSRQ

Este bloque es un registro de corrimiento a la izquierda con una entrada que determina el valor del bit menos significativo. Su descripción funcional es la siguiente:

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
ENTITY lsrq IS
GENERIC(width:natural:=7);
PORT(
clk,shift,DIN,reset : IN  STD_LOGIC;
Qlsrq : BUFFER STD_LOGIC_VECTOR(width downto 0));
END lsrq;
Architecture funcional of lsrq is
Begin
Process(Clk)
Begin
if (Clk'event and Clk = '1') then
if reset = '1' then
Qlsrq <= "00000000";
elsif (shift = '1') then
Qlsrq(width downto 1) <= Qlsrq((width - 1) downto 0);
Qlsrq(0) <= DIN;
else
Qlsrq <= Qlsrq;
end if;
end if;
End Process;
End funcional;
```

La simulación del módulo LSRQ se muestra en la siguiente figura.

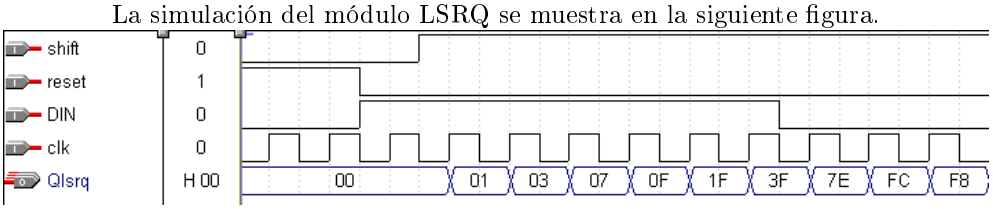


Figura 28. Simulación del módulo LSRQ.

LSR

Este módulo está encargado de realizar el corrimiento a la izquierda del dividendo y almacenar los resultados de la resta. La descripción funcional de este módulo es el siguiente:

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
ENTITY lsr IS
GENERIC(width:natural:=7);
PORT(
clk,shift,load,reset: IN STD_LOGIC;
Data,DataL:   IN STD_LOGIC_VECTOR(width downto 0);
Qlsr:      BUFFER STD_LOGIC_VECTOR(width downto 0));
END lsr;
ARCHITECTURE funcional OF lsr IS
SIGNAL INN1: STD_LOGIC_VECTOR(width downto 0);
BEGIN
Process(CLK)
Begin
if (clk'event and clk = '1') then
if Reset = '1' then
Qlsr <="00000000";
```

```
INN1 <= Data;
elsif load = '1' then
Qlsr <= DataL;
elsif shift = '1' then
Qlsr(width downto 1) <= Qlsr((Width -1 ) downto 0);
Qlsr(0) <= INN1(width);
INN1(width downto 1) <= INN1((Width -1 ) downto 0);
INN1(0) <= '0';
end if;
else
Qlsr <= Qlsr;
end if;
End Process;
END funcional;
```

La simulación de este módulo se muestra en la Figura 29.

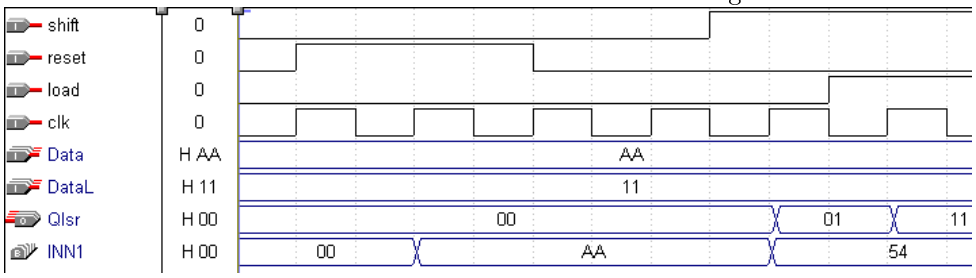


Figura 29. Simulación del módulo LSR.

DECREMENT

Este módulo está encargado de controlar el número de veces que se realizan las iteraciones. La descripción funcional se muestra a continuación:

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
Entity Decrement is
Port(
Clk,Init,dec  : IN  STD_LOGIC;
Z     : OUT STD_LOGIC);
End Decrement;
Architecture funcional of Decrement is
Signal INN1 : Std_logic_vector (2 downto 0);
signal end1 : Std_Logic;
Begin
Process (Clk,dec)
Begin
if (Clk'event and Clk = '1') then
if Init = '1' then
Z <= '0';
INN1 <= "111";
end1 <= '0';
elsif (dec = '1' and end1 = '0') then
if (INN1 > 0) then
z <= '0';
INN1 <= INN1 - 1;
else
z<= '1';
end1 <= '1';
End if;
else
INN1 <= INN1;
end if;
```

End if;
End Process;
End funcional;

La simulación de este módulo se muestra en la Figura 30.

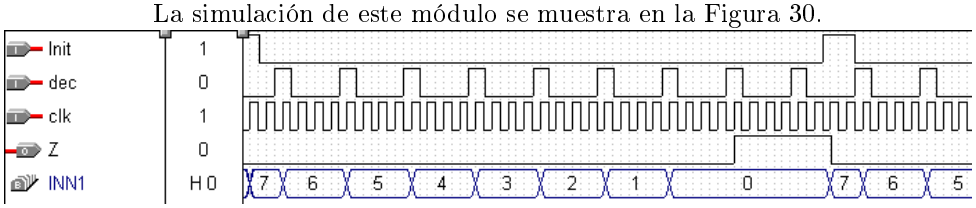


Figura 30. Simulación del módulo DECREMENT.

ADDSUB

Este módulo está encargado de realizar la resta para determinar si al número contenido en el módulo LSR se le puede restar el divisor. La descripción funcional de este módulo se muestra a continuación.

Library *IEEE;*
USE *IEEE.STD_LOGIC_1164.all;*
USE *IEEE.STD_LOGIC_ARITH.all;*
USE *IEEE.STD_LOGIC_UNSIGNED.all;*
ENTITY *addsub* ***IS***
GENERIC(width:natural:=7);
PORT*(*
clk,reset,add,sub: ***IN*** *STD_LOGIC;*
msb: ***OUT*** *STD_LOGIC;*
A,B : ***IN*** *STD_LOGIC_VECTOR(width downto 0);*
AS: ***BUFFER*** *STD_LOGIC_VECTOR(width downto 0));*
END *addsub;*
ARCHITECTURE *funcional* ***OF*** *addsub* ***IS***
BEGIN
PROCESS *(clk, Reset)*
BEGIN
if *Reset = '1'* ***then***
AS <= "00000000";
else
if *(clk'****event*** ***and*** *clk='1')* ***then***
if *add = '1'* ***and*** *sub = '0'* ***then***
AS <= A + B;
elsif *add = '0'* ***and*** *sub = '1'* ***then***
AS <= A - B;
else
AS <= "00000000";
end if;
end if;
End if;
END PROCESS;
Process*(AS)*
Begin
MSB <= AS(width);
End Process;
END funcional;

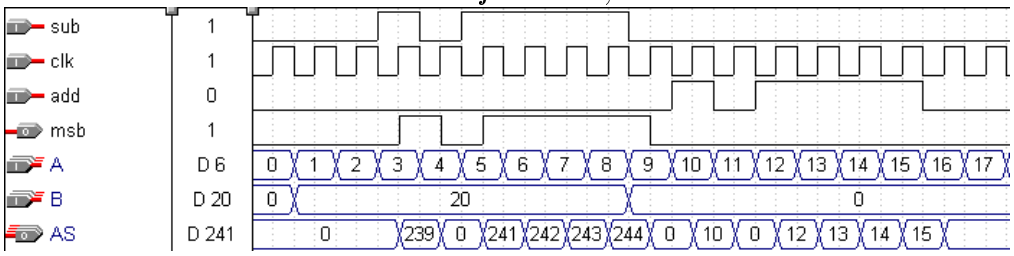


Figura 31. Simulación del módulo ADDSUB.

DIVISOR

Finalmente se realizó la simulación del archivo DIVISOR.VHD. El cual contiene la unión de los módulos anteriores. La simulación del divisor se muestra en la Figura 32. Y la implementación en VHDL del *testbench* se muestra a continuación:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY testbench IS
END testbench;
ARCHITECTURE behavior OF testbench IS
COMPONENT divisor
PORT(
A : IN std_logic_vector(7 downto 0);
B : IN std_logic_vector(7 downto 0);
Init : IN std_logic;
clk : IN std_logic;
D : BUFFER std_logic_vector(7 downto 0);
R : BUFFER std_logic_vector(7 downto 0);
Done : OUT std_logic
);
END COMPONENT;
SIGNAL A : std_logic_vector(7 downto 0);
SIGNAL B : std_logic_vector(7 downto 0);
SIGNAL Init : std_logic;
SIGNAL clk : std_logic;
SIGNAL D : std_logic_vector(7 downto 0);
SIGNAL R : std_logic_vector(7 downto 0);
SIGNAL Done : std_logic;
constant ncycles : integer := 80;
constant halfperiod : time := 5 ns;
BEGIN
uut: divisor PORT MAP(
A => A,
B => B,
Init => Init,
clk => clk,
D => D,
R => R,
Done => Done
);
-- Generacion del Reloj
Clock_Source: process
begin
for i in 0 to ncycles loop -- Genera ncyclos de periodo 10 ns
CLK <= '0';
wait for halfperiod;
CLK <= '1';
wait for halfperiod;
end loop;
wait;
end process Clock_Source;
tb : PROCESS
BEGIN
A <= "01100100";
B <= "00000011";
INIT <= '0'; wait for halfperiod;
INIT <= '1';
wait until CLK'event and CLK='1';
```

```
wait until CLK'event and CLK='1';
wait until CLK'event and CLK='0';
INIT <= '0';
wait until Done'event and Done = '0';
for i in 1 to 6 loop
wait until CLK'event and CLK='1';
end loop;
INIT <= '1';
B <= "00001111";
wait until CLK'event and CLK='1';
wait until CLK'event and CLK='1';
wait until CLK'event and CLK='0';
INIT <= '0';
wait until Done'event and Done = '0';
wait;
END PROCESS;
END;
```

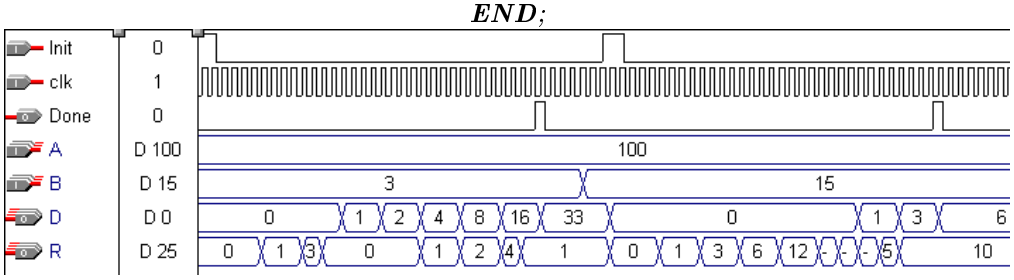


Figura 32. Simulación del divisor.

Segunda Implementación: En esta sección se realizará la implementación del divisor utilizando un procedimiento que realice la operación de división. Inicialmente declararemos un package al cual llamaremos *aritmetico* para después utilizarlo en una entidad:

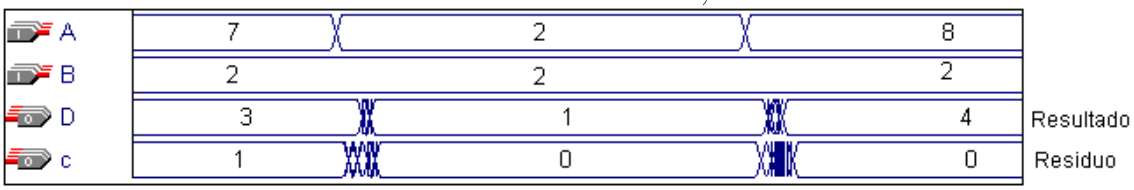
```
library ieee;
use ieee.std_logic_1164.all;
package aritmetico is
-- Todo package se divide en dos partes en la primera (encabezado) se realiza la
-- declaración de los elementos que los constituyen, indicando únicamente sus entradas
-- y salidas:
procedure add ( A, B : in bit_vector;
suma : out bit_vector; --Declaración de un sumador
over : out boolean );
procedure sub ( A, B : in bit_vector;
resta : out bit_vector; --Declaración de un restador
over : out boolean );
procedure divisor
( Q, M : in bit_vector;
cociente : out bit_vector; --Declaración de un divisor.
residuo : out bit_vector;
div_cero : out boolean);
end package aritmetico;
-- La segunda parte del Package recibe el nombre de cuerpo (body) y debe
-- contener la descripción funcional de cada uno de los componentes declarados
-- en la primera parte.
package body aritmetico is
--Una vez realizada la descripción de los elementos que forman el package se
--debe ndicar su funcionemiento
--****************************
--*****SUMADOR DE N BITS*******
--****************************
procedure add ( A, B : in bit_vector;
suma : out bit_vector; --Esta declaración debe ser idéntica a la anterior
```

```
over : out boolean ) is
alias op1 : bit_vector(A'length - 1 downto 0) is A;
alias op2 : bit_vector(B'length - 1 downto 0) is B;
variable result : bit_vector(suma'length - 1 downto 0);
variable carry_in : bit;
variable carry_out : bit := '0';
begin
for index in result'reverse_range loop
carry_in := carry_out; -- Carry IN = Carry OUT del bit anterior
result(index) := op1(index) xor op2(index) xor carry_in; -- Sumador de un bit
carry_out := (op1(index) and op2(index)) -- con Carry.
or (carry_in and (op1(index) xor op2(index)));
end loop;
suma := result;
over := carry_out /= carry_in;
end add;
--*****************************
--*****RESTADOR DE N BITS*******
--*****************************
procedure sub ( A, B : in bit_vector;
resta : out bit_vector;
over : out boolean ) is
alias op1 : bit_vector(A'length - 1 downto 0) is A;
alias op2 : bit_vector(B'length - 1 downto 0) is B;
variable result : bit_vector(resta'length - 1 downto 0);
variable carry_in : bit;
variable carry_out : bit := '1' -- Para realizar la suma en complemento a dos
-- Es necesario realizar (A + Complemento (B) + 1).
begin
--Implementación de un restador, utilizando suma en complemento a dos
for index in result'reverse_range loop
carry_in := carry_out;
result(index) := op1(index) xor (not op2(index)) xor carry_in;
carry_out := (op1(index) and (not op2(index)))
or (carry_in and (op1(index) xor (not op2(index))));
end loop;
resta := result;
over := carry_out /= carry_in;
end sub;
end package body aritmetico;
```

A continuación se presenta un ejemplo de llamado del procedimiento divisor por una entidad:

```
library ieee;
use ieee.std_logic_1164.all;
--Al hacer el llamado del Package se debe indicar que se encuentra en el
--directorio de trabajo (libreria work);
library work;
use work.aritmetico.all;
--Declaración de la Entidad
entity cain is
port(
A,B : in bit_vector (4 downto 1);
D,C : out bit_vector(4 downto 1));
end entity cain;
-- Declaración de la arquitectura
architecture cain of cain is
begin
test: process (A)
variable hh : boolean;
```

```
variable ss, tt : bit_vector(4 downto 1);
begin
divisor(A, B, ss, tt,hh);
D <= ss;
C <= tt;
end process test;
end architecture cain;
```



En la Figura 33. Se muestra la simulación del divisor.

EJEMPLO 4: Cronómetro digital

Se desea diseñar un sistema digital que sea capaz de medir segundos y décimas de segundo utilizando una fuente de reloj externa de FMHZ MHz y dos displays de 7 segmentos para la visualización. El diagrama de bloques del sistema aparece en la Figura 34.

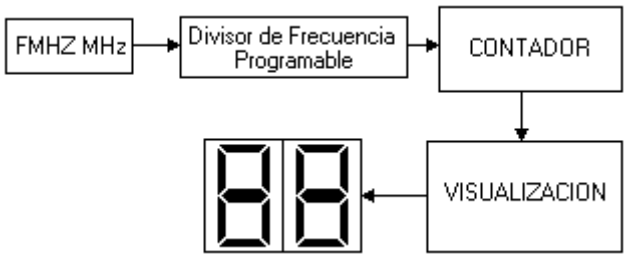


Figura 34. Diagrama de bloques del Cronómetro Digital.

Divisor de frecuencia Programable (DIVISOR): Como se habrán dado cuenta el divisor de frecuencia programable no tiene una entrada fija, esta entrada depende de la disponibilidad del sistema, por lo que en este caso es un *genérico*. El código en VHDL del contador es el siguiente:

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
Entity clk_div Is
GENERIC( FMHZ: Natural := 8 ); -- FMHZ genérico 8MHz por defecto
PORT(
clock_FMHZ : IN STD_LOGIC; -- Señal de reloj externa.
Reset : IN Std_Logic; -- Reset del Módulo
clock_1Hz : OUT STD_LOGIC); -- Salida a 1Hz.
end clk_div;
Architecture RT of clk_div is
Signal count_1Mhz : Std_Logic_Vector(4 DOWNTO 0);
Signal Clock_1MHz : Std_Logic;
Signal Count_1Hz : Integer Range 0 to 1000000;
Begin
Process
Begin
-- Divisor por FMHZ
wait until clock_FMHZ'event and clock_FMHZ = '1';
if Reset = '1' then
if count_1Mhz < ( FMHZ - 1 ) then
count_1Mhz <= count_1Mhz + 1;
else
count_1Mhz <= "00000";
end if;
if count_1Mhz < FMHZ/2 then
```

```
clock_1MHz <= '0';
else
clock_1MHz <= '1';
end if;
else
Count_1Mhz <= "00000";
end if;
end Process;
-- Divisor por 1000000
Process ( clock_1Mhz, Reset )
Begin
if Reset = '0' then
Count_1Hz <= 0;
else
if clock_1Mhz'event and clock_1Mhz = '1' then
if count_1Hz < 1000000 then
count_1Hz <= count_1Hz + 1;
else
count_1Hz <= 0;
end if;
if count_1Hz < 5000000 then
clock_1Hz <= '0';
else
clock_1Hz <= '1';
end if;
end if;
end if;
end Process;
end RT;
```

Como puede observarse existen dos procesos dentro de la arquitectura el primero es un divisor de frecuencia por FMHZ el cual proporciona una señal de reloj de 1 MHz, esto se hizo con el fin de proporcionarle adaptabilidad al sistema a cualquier señal de reloj mayor a 1 MHz. El segundo proceso es un divisor de frecuencia por 1 millón, la salida de este proceso es una señal con frecuencia de 1 Hz.

CONTADOR: Una vez obtenida la frecuencia de 1 Hz se procede a realizar el contador. El código VHDL del contador se muestra a continuación:

```
Library Ieee;
Use Ieee.Std_Logic_1164.all;
Use Ieee.Std_Logic_Arith.all;
Use Ieee.Std_Logic_unsigned.all;
Entity Contador is
Port(
Clock : In Std_Logic;
Reset : In Std_Logic;
Unidades : Buffer Std_Logic_Vector( 3 Downto 0 );
Decenas : Buffer Std_logic_Vector( 3 Downto 0 )
);
end Entity Contador;
Architecture RT of Contador is
Begin
Process (Clock, Reset)
Begin
If Reset = '0' then
Unidades <= "0000";
Decenas <= "0000";
else
if Clock'event and Clock = '1' then
If Unidades < "1001" then
```

```
Unidades <= Unidades + "0001";
else
Unidades <= "0000";
if Decenas < "1001" then
Decenas <= Decenas + "0001";
else
Unidades <= "0000";
Decenas <= "0000";
end if;
end if;
end if;
end if;
End Process;
End Architecture RT;
```

VISUALIZACION: Para este ejemplo utilizaremos visualización dinámica; el módulo debe contar con una salida que sea capaz de manejar un display de 7 segmentos y señales para la selección del display. A continuación se muestra el diagrama de bloques y las entradas y salidas del visualizador:

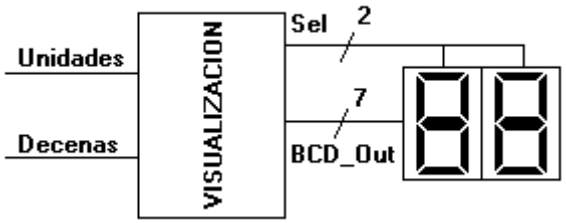


Figura. 35 Diagrama de Bloques de la visualización.

La visualización dinámica funciona de esta forma: Los segmentos comunes de los Displays están unidos entre sí de tal forma que el valor a desplegar se pueda mostrar en cualquiera de los dos. Las entradas de selección indican en cual de ellos se mostrará el valor enviado. Por ejemplo para mostrar el número 69 se deben seguir los siguientes pasos:

Seleccionar el Display de la Derecha haciendo Sel[0] = 1 y Sel[1] = 0.

Colocar la información correspondiente al Número 9 en BCD_Out:

1. Esperar TD milisegundos.
2. Hacer BCD_Out = 0, (Esto para evitar que por un breve instante de tiempo se muestre la información del otro display)
3. Seleccionar el Display de la izquierda haciendo Sel[0] = 0 y Sel[1] = 1.
4. Colocar la información correspondiente al Número 6 en BCD_Out:
5. Esperar TD milisegundos.
6. Repetir el paso 1.

Los pasos que debe realizar el módulo de visualización son lo mismos del ejemplo, pero la información que muestran en cada momento son los valores de las entradas “Unidades” y “Decenas”.

Debido a que necesitamos una señal de reloj con un período de TD ms, podemos incluir un divisor de frecuencia dentro de la visualización, pero esto implica gastar más compuertas lógicas; la solución más óptima es incluir el divisor de frecuencia en el módulo divisor. Con lo que el código del divisor queda de la siguiente forma:

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
Entity clk_div2 Is
```

```
GENERIC(
FMHZ: Natural := 8;
TD : Natural := 20 ); -- FMHZ genérico 8MHz por defecto
PORT(
clock_FMHZ : In Std_Logic; -- Señal de reloj externa.
Reset : In Std_Logic; -- Reset del Módulo
clock_1Hz : Out Std_Logic;
Clock_TD : Out Std_Logic ); -- Salida a 1Hz.
END clk_div2;
Architecture RT of clk_div2 is
Signal count_1Mhz : STD_LOGIC_VECTOR(4 DOWNTO 0);
Signal Clock_1MHz : Std_Logic;
Signal Clock_1ms : Std_Logic;
Signal Count_1Hz : Integer Range 0 to 100;
Signal Count_1ms : Integer Range 0 to 100;
Signal Count_TDms : Integer Range 0 to 100;
Begin
Process
Begin
-- Divisor por FMHZ
wait until clock_FMHZ'event and clock_FMHZ = '1';
if Reset = '1' then
if count_1Mhz < ( FMHZ - 1 ) then
count_1Mhz <= count_1Mhz + 1;
else
count_1Mhz <= "00000";
end if;
if count_1Mhz < FMHZ/2 then
clock_1MHz <= '0';
else
clock_1MHz <= '1';
end if;
else
Count_1Mhz <= "00000";
end if;
end Process;
-- Divisor por 1000
Process ( clock_1Mhz, Reset )
Begin
if Reset = '0' then
Count_1ms <= 0;
else
if clock_1Mhz'event and clock_1Mhz = '1' then
if count_1ms < 1000 then
count_1ms <= count_1ms + 1;
else
count_1ms <= 0;
end if;
if count_1ms < 500 then
clock_1ms <= '0';
else
clock_1ms <= '1';
end if;
end if;
end if;
End Process;
-- Divisor por 1000
Process ( clock_1ms, Reset )
```

```
Begin
if Reset = '0' then
Count_1Hz <= 0;
else
if clock_1ms'event and clock_1ms = '1' then
if count_1Hz < 1000 then
count_1Hz <= count_1Hz + 1;
else
count_1Hz <= 0;
end if;
if count_1Hz < 500 then
clock_1Hz <= '0';
else
clock_1Hz <= '1';
end if;
end if;
end if;
End Process;
-- Base De tiempo para la visualización dinámica
Process ( clock_1ms, Reset )
Begin
if Reset = '0' then
Count_TDms <= 0;
else
if clock_1ms'event and clock_1ms = '1' then
if count_TDms < TD then
count_TDms <= count_TDms + 1;
else
count_TDms <= 0;
end if;
if count_TDms < TD/2 then
clock_TD <= '0';
else
clock_TD <= '1';
end if;
end if;
end if;
end Process;
END RT;
```

En esta versión del divisor de frecuencia se obtiene una señal con una frecuencia de 1KHz al realizar la división de la señal de 1MHz entre 1000. Esta señal se toma como base para obtener las señales de Base de tiempo para la visualización dinámica (Dividiendo por TD) y base de tiempo de 1 Hz.

Finalmente el código del módulo de visualización se muestra a continuación:

```
Library Ieee;
Use Ieee.Std_Logic_1164.all;
Use Ieee.Std_Logic_Arith.all;
Use Ieee.Std_Logic_unsigned.all;
Entity Show is
Port(
Unidades : In Std_Logic_vector( 3 downto 0 );
Decenas : In Std_Logic_vector( 3 downto 0 );
Clk_TDms : In Std_Logic;
Sel : Out Std_Logic_vector( 1 downto 0 );
BCD_Out : Out Std_Logic_Vector( 6 downto 0 )
-- BCD_Out[7..0] = a, b, c, d, e, f, g
);
End Entity Show;
```

```
Architecture RT of Show is
Signal Binary : Std_logic_Vector ( 3 downto 0 );
Type Estados is ( Show_Decenas, Show_Unidades );
Signal State : Estados;
Begin
Process( Binary )
Begin
Case Binary is
When "0000" =>
BCD_Out <= "0111111";
When "0001" =>
BCD_Out <= "0100001";
When "0010" =>
BCD_Out <= "1110110";
When "0011" =>
BCD_Out <= "1110011";
When "0100" =>
BCD_Out <= "1101001";
When "0101" =>
BCD_Out <= "1011011";
When "0110" =>
BCD_Out <= "1011111";
When "0111" =>
BCD_Out <= "0110001";
When "1000" =>
BCD_Out <= "1111111";
When "1001" =>
BCD_Out <= "1111011";
When others =>
BCD_Out <= "0000000";
end Case;
End Process;
Process(Clk_TDms)
Begin
If (Clk_TDms'event and Clk_TDms = '1') Then
Case State is
When Show_Decenas =>
Binary <= Decenas;
State <= Show_Unidades;
Sel <= "01";
When Show_Unidades =>
Binary <= Unidades;
State <= Show_Decenas;
Sel <= "10";
End Case;
End If;
End Process;
End Architecture RT;
```

Una vez realizados los diferentes módulos del sistema se debe realizar una descripción estructural para realizar la interconexión entre ellos:

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
Entity Cronometro is
Port(
Reset,clk  : IN STD_LOGIC;
Sel  : Out STD_LOGIC_VECTOR( 1 downto 0 );
```

```
Seven_Seg : Out STD_LOGIC_VECTOR( 6 downto 0 ) );
End Entity Cronometro;
Architecture Estructural Of Cronometro is
Component Show
Port(
Unidades : In Std_Logic_vector( 3 downto 0 );
Decenas : In Std_Logic_vector( 3 downto 0 );
Clk_TDms : In Std_Logic;
Sel : Out Std_Logic_vector( 1 downto 0 );
BCD_Out : Out Std_Logic_Vector( 6 downto 0 ) );
End Component Show;
Component Contador
Port(
Clock : In Std_Logic;
Reset : In Std_Logic;
Unidades : Buffer Std_Logic_Vector( 3 Downto 0 );
Decenas : Buffer Std_logic_Vector( 3 Downto 0 )
);
end Component Contador;
Component clk_div2
GENERIC(
FMHZ: Natural := 8;
TD : Natural := 20 );
PORT(
clock_FMHZ : In Std_Logic;
Reset : In Std_Logic;
clock_1Hz : Out  Std_Logic;
Clock_TD : Out  Std_Logic );
END Component clk_div2;
Signal CLK1Hz, CLKTD : Std_logic;
Signal Unidades, Decenas : Std_logic_vector( 3 downto 0 );
Begin
A1: clk_div2
Generic map(8, 20)
Port Map( clk, Reset, CLK1Hz, CLKTD );
A2: Contador
port map( CLK1Hz, Reset, Unidades, Decenas );
A3: Show
port map( Unidades, Decenas, CLKTD, Sel, Seven_Seg );
End Architecture Estructural;
```

El código correspondiente al *testbench* del cronometro es:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
ENTITY testbench IS
END testbench;
ARCHITECTURE behavior OF testbench IS
-- Component Declaration
Component cronometro
PORT(
Reset, clk : IN std_logic;
Sel : OUT std_logic_vector(1 downto 0);
Seven_Seg : OUT std_logic_vector(6 downto 0)
);
END COMPONENT;
SIGNAL Reset : std_logic;
SIGNAL clk : std_logic;
SIGNAL Sel : std_logic_vector(1 downto 0);
```

```
SIGNAL Seven_Seg : std_logic_vector(6 downto 0);
constant ncycles : integer := 40000000;
constant halfperiod : time := 5 ns;
BEGIN
-- Component Instantiation
uut: cronometro PORT MAP(
Reset => Reset,
clk => clk,
Sel => Sel,
Seven_Seg => Seven_Seg
);
-- Generacion del Reloj
Clock_Source: process
begin
for i in 0 to ncycles loop -- Genera ncyclos de periodo 10 ns
clk <= '0';
wait for halfperiod;
clk <= '1';
wait for halfperiod;
end loop;
wait;
end process Clock_Source;
tb : PROCESS
BEGIN
Reset <= '0';
wait until clk'event and clk = '1';
wait until clk'event and clk = '1';
Reset <= '1';
wait;
END PROCESS;
END;
```

EJEMPLO 5: Contador UP/DOWN

En este ejemplo se diseñará un contador ascendente descendente de 0 a 99. El sistema tendrá como entradas de control tres pulsadores: Reset, Aumento y dismunición. El diagrama de bloques del sistema se muestra en la siguiente figura:

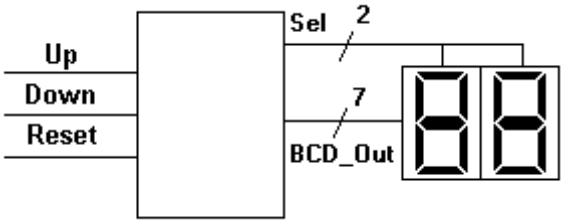


Figura 36. Interfaz del Contador Up/Down

El diagrama de bloques de este sistema es el siguiente:

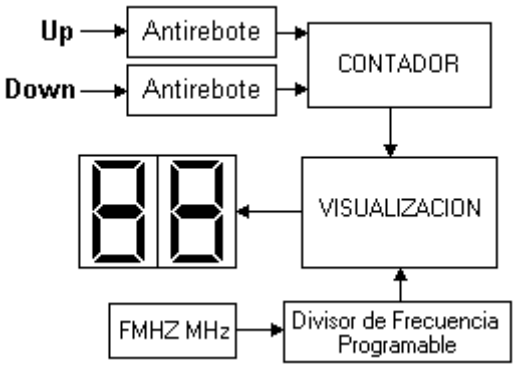


Figura 37. Diagrama de Bloques del contador Up/Down

Como se puede ver el circuito de visualización es idéntico al del ejemplo anterior. Debido a que utilizaremos elementos mecánicos (pulsadores) debemos diseñar un circuito que elimine el ruido eléctrico introducido por estos.

ANTIREBOTE: Este módulo se encarga de eliminar el ruido eléctrico introducido por los pulsadores, previniendo de esta forma conteos erróneos. Además incorpora un circuito que genera un pulso de duración de un período de la señal de reloj para evitar que cuando se deja oprimido el Pulsador se genere más de un conteo.

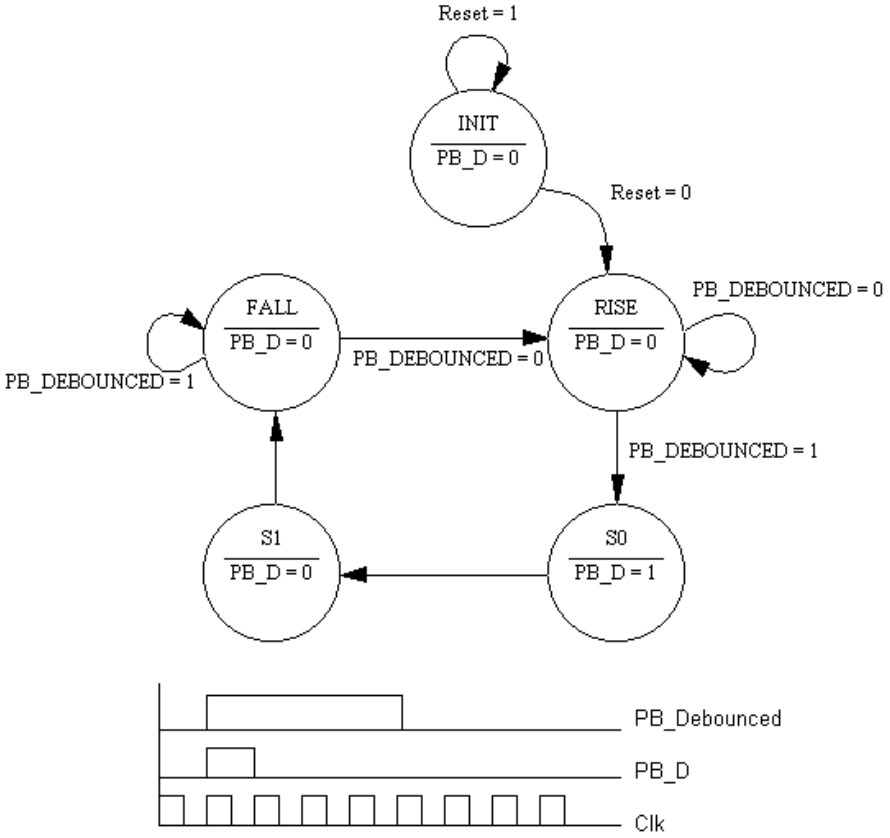


Figura 38. Diagrama de estados del módulo antirrebote

El código en VHDL de este módulo es el siguiente:

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
Entity debounce Is
Port(
Push_Button, Clk : In Std_Logic;
Reset : In Std_Logic;
PB_D  : Out Std_Logic);
End debounce;
Architecture RT OF debounce Is
Type estados Is ( Rise, fall, S0, S1 );
Signal state  : estados;
Signal SHIFT_PB : Std_Logic_Vector( 3 downto 0 );
Signal PB_DEBOUNCED : Std_Logic;
Begin
Process -- Este proceso elimina el ruido eléctrico generado por los pulsadores
begin
wait until ( Clk'EVENT ) AND ( Clk = '1' );
SHIFT_PB( 2 Downto 0 ) <= SHIFT_PB( 3 Downto 1 );
SHIFT_PB(3) <= NOT Push_Button;
```

```
If SHIFT_PB( 3 Downto 0 )="0000" THEN
PB_DEBOUNCED <= '0';
Else
PB_DEBOUNCED <= '1';
End if;
end process;
Process ( Clk, PB_DEBOUNCED, Reset )
Begin
If Reset = '1' then
state <= Rise;
PB_D <= '0';
else
If Clk'event and Clk = '0' then
Case state is
When Rise =>
PB_D <= '0';
If PB_DEBOUNCED = '0' then
State <= Rise;
else
State <= S0;
end if;
When S0 =>
PB_D <= '1';
State <= S1;
When S1 =>
PB_D <= '0';
State <= Fall;
When Fall =>
PB_D <= '0';
If PB_DEBOUNCED = '1' then
State <= Fall;
else
State <= Rise;
end if;
When Others =>
State <= Rise;
End Case;
End If;
End If;
End Process;
end architecture RT;
```

Figura 39. Interfaz del módulo contador.

CONTADOR: Este módulo está encargado de aumentar o disminuir el valor de sus salidas de acuerdo a la entrada que este actuando. La interfaz de este módulo se muestra en la figura anterior y el código en VHDL se muestra a continuación.

```
Library Ieee;
Use Ieee.Std_Logic_1164.all;
Use Ieee.Std_Logic_Arith.all;
Use Ieee.Std_Logic_unsigned.all;
Entity Contador is
```

```
Port(
Up : In Std_Logic;
Down : In Std_Logic;
Reset : In Std_logic;
Clk : In Std_logic;
Unidades : Buffer Std_Logic_Vector( 3 Downto 0 );
Decenas : Buffer Std_logic_Vector( 3 Downto 0 )
);
end Entity Contador;
Architecture RT of Contador is
Begin
Process ( Up, Down, Reset, Clk )
Begin
If Reset = '1' then
Unidades <= "0000";
Decenas <= "0000";
else
If Clk'event and clk='1' then
if Up = '1' and Down = '0' then
If Unidades < "1001" then
Unidades <= Unidades + "0001";
else
Unidades <= "0000";
if Decenas < "1001" then
Decenas <= Decenas + "0001";
else
Unidades <= "0000";
Decenas <= "0000";
end if;
end if;
elsif Up = '0' and Down = '1' then
If Unidades > "0000" then
Unidades <= Unidades - "0001";
else
Unidades <= "1001";
if Decenas > "0000" then
Decenas <= Decenas - "0001";
else
Unidades <= "1001";
Decenas <= "1001";
end if;
end if;
else
Unidades <= Unidades;
Decenas <= Decenas;
end if;
end if;
end if;
End Process;
End Architecture RT;
```

La descripción estructural del contador Up/Down es la siguiente:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity UDCounter is
port (
Up : in STD_LOGIC;
Down : in STD_LOGIC;
Clk : in STD_LOGIC;
```

```
Reset : in STD_LOGIC;
Sel : Buffer STD_LOGIC_VECTOR (1 downto 0);
SEG : out STD_LOGIC_VECTOR (6 downto 0)
);
end UDCounter;
architecture UDCounter_arch of UDCounter is
Component Show
Port(
Unidades : In Std_Logic_vector( 3 downto 0 );
Decenas : In Std_Logic_vector( 3 downto 0 );
Clk_TDms : In Std_Logic;
Sel : Buffer Std_Logic_vector( 1 downto 0 );
BCD_Out : Out Std_Logic_Vector( 6 downto 0 ) );
End Component Show;
Component Contador
Port(
Up : In Std_Logic;
Down : In Std_Logic;
Reset : In Std_logic;
Clk : In Std_logic;
Unidades : Buffer Std_Logic_Vector( 3 Downto 0 );
Decenas : Out Std_logic_Vector( 3 Downto 0 )
);
end Component Contador;
Component debounce
Port(
Push_Button, Clk : In Std_Logic;
Reset : In Std_Logic;
PB_D : Out Std_Logic);
End Component debounce;
Component clk_div2
GENERIC(
FMHZ: Natural := 8;
TD : Natural := 20 );
PORT(
clock_FMHZ : In Std_Logic;
Reset : In Std_Logic;
clock_1Hz : Out Std_Logic;
Clock_TD : Out Std_Logic );
END Component clk_div2;
Signal CLKTD, CLK1Hz, UPI, DNI : Std_Logic;
Signal Unidades, Decenas : Std_logic_vector( 3 downto 0 );
begin
A1: clk_div2
Generic map(8, 20)
Port Map( Clk, Reset, CLK1Hz, CLKTD );
A2: debounce
Port Map( Up, CLKTD, Reset, UPI );
A3: debounce
Port Map( Down, CLKTD, Reset, DNI );
A4: Contador
Port Map( UPI, DNI, Reset, CLKTD, Unidades, Decenas );
A5: Show
Port Map( Unidades, Decenas, CLKTD, Sel, SEG );
end UDCounter_arch;
```

Ejemplo 6: UART (Universal Asynchronous Receiver & Transmitter)

La comunicación asíncrona es ampliamente utilizada en los sistemas digitales, por esta razón en este ejemplo se diseñará un sistema capáz de realizar una comunicación serial asíncrona.

En la Figura 40 se muestra una trama típica de este tipo de comunicación, el primer bit en ser enviado es el Bit de Start (Cero Lógico), después se envían los bits de datos los cuales pueden variar de 6 a 8 Bits, a continuación el bit de paridad el cual es útil para la detección de errores en la transferencia; Este bit no es obligatorio y en este ejemplo no lo utilizaremos y por último está el bit de stop.

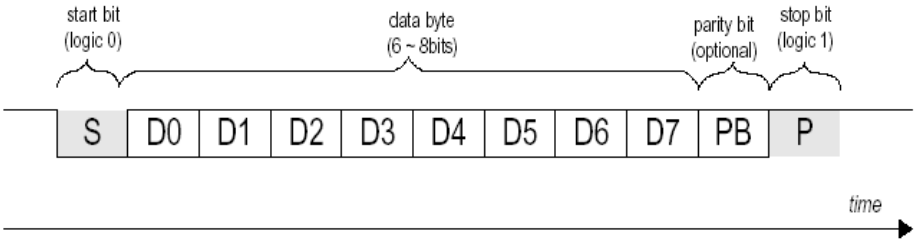


Figura 40. Trama básica de la transmisión asíncrona: 1 bit de Start, 8 bits de Datos, un bit de paridad (opcional) y un Bit de Stop.

En un protocolo asíncrono el emisor y el receptor no comparten el mismo reloj, debido a esto se debe conocer la velocidad de transmisión (Baud Rate) antes de iniciarse las comunicaciones. Por lo cual los relojes internos del transmisor y el receptor se deben fijar a la misma frecuencia. El receptor sincroniza su reloj interno al iniciar cada trama, esto se puede realizar detectando la transición de alto a bajo en señal de recepción.

Un concepto clave en el diseño de UARTs es que el reloj interno de la UART debe ser más rápido que la velocidad de transmisión. En la UART 16450 su reloj es 16 veces más rápido que la Tasa de Baudios. Una vez que el inicio de la transmisión se detecta se debe esperar un tiempo igual a 24 ciclos de muestreo (16 del Bit de Start + 8 del Bit) esto se hace para que el receptor pueda relizar la lectura del dato entrante en la mitad de cada BIT. Después de esto se muestrea la señal de entrada cada 16 ciclos del reloj de muestreo. En la Figura 41 se muestra este concepto.

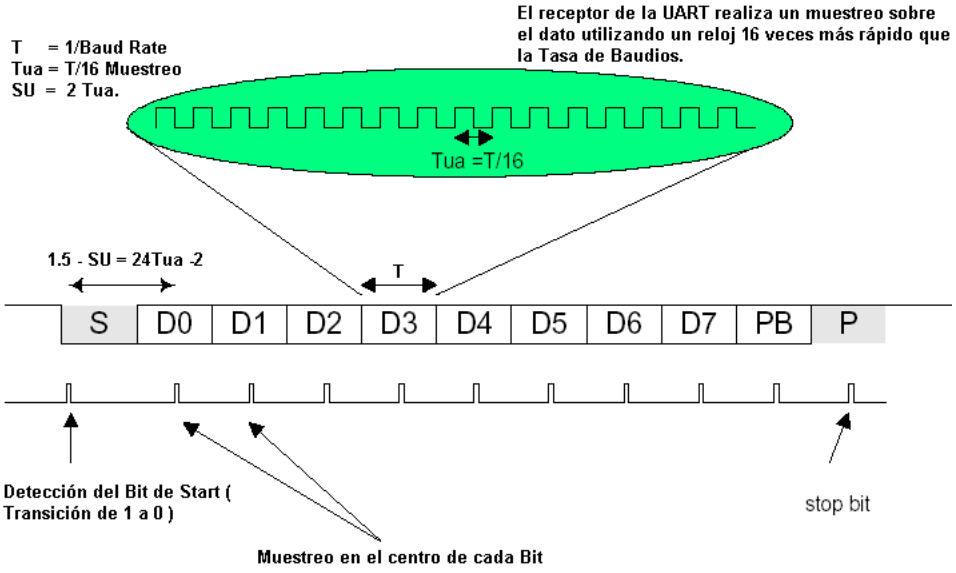


Figura 41. Puntos de muestreo del receptor.

El diseño del transmisor es más sencillo ya que el puede iniciar a enviar datos en cualquier momento y simplemente debe realizar el corrimiento de los datos cada 1/Baud Rate.

En la siguiente figura se muestra la interfaz y el diagrama de bloques de la UART.

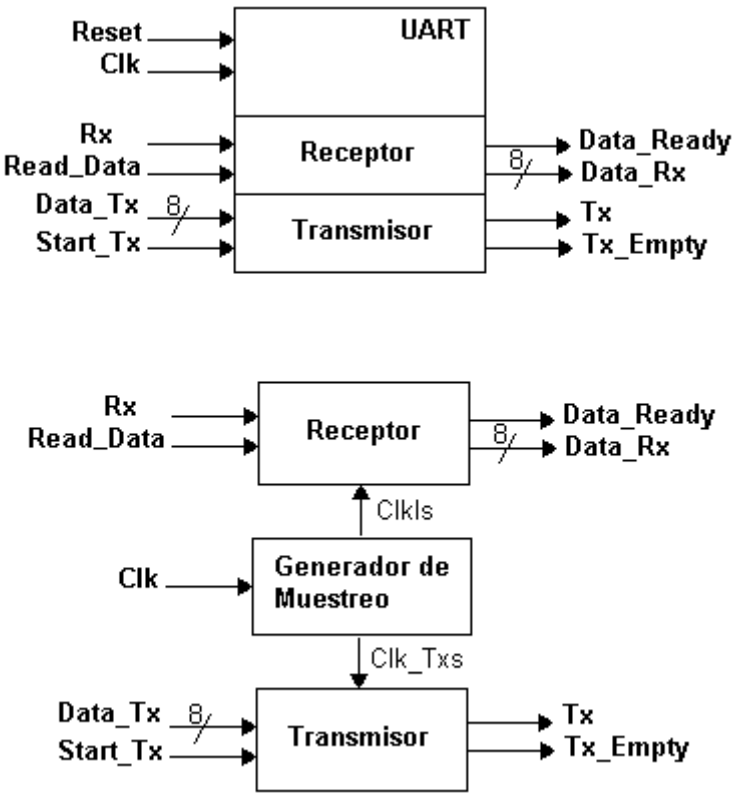


Figura 42. Diagrama de Bloques de la UART.

Donde:

Reset : Reset general del sistema activa alta.

Clk : Señal de Reloj.

Rx : Señal de entrada serial asíncrona.

Read_Data : Activa alta le informa al receptor que el dato a sido leído.

Data_Ready : Indica la recepción de un dato por la línea Rx. El dato está disponible en Data_Rx.

Data_Rx(7..0) : Dato recibido por el receptor.

Data_Tx(7..0) : Dato a transmitir por la UART.

Start_Tx : Inicio de la transmisión.

Tx : Señal de salida serial

Tx_Empty : En estado lógico alto ndica que el transmisor está disponible para una nueva transmisión.

Generador de Muestreo: Este módulo está encargado de generar la frecuencia de muestreo del receptor y el reloj del transmisor. En la siguiente figura se muestra la interfaz y el diagrama de bloques de este módulo.

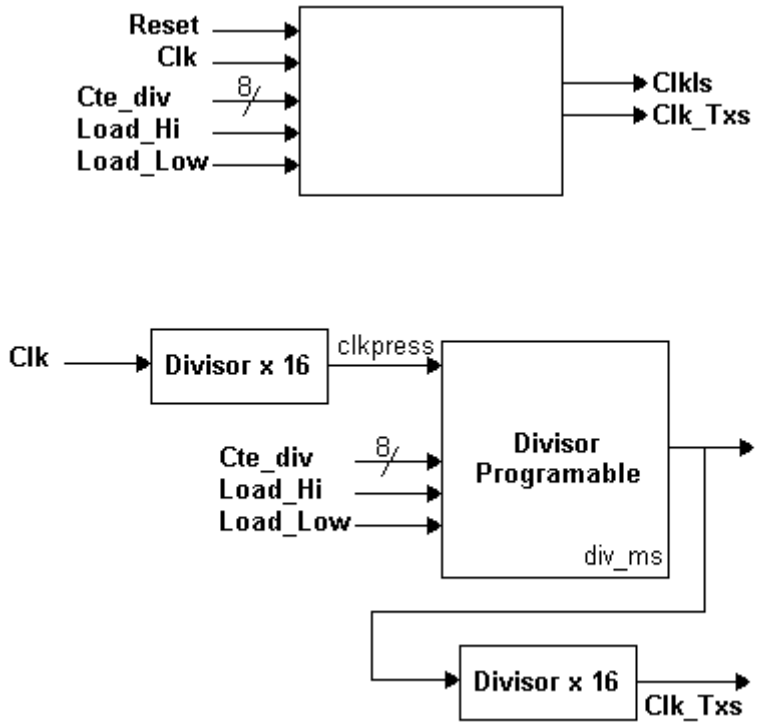


Figura 43. Diagrama de bloques del generador de muestreo.

El bloque " Divisor x 16 " como su nombre lo indica divide la señal de entrada en 16. El divisor de frecuencia programable es capaz de dividir la señal de entrada de 1 a FFFF H (65535). El factor de división debe ser ingresado de la siguiente forma: Colocar el byte menos significativo en *cte_div* y hacer *Load_Hi = 0* y *Load_Low* = 1. Colocar el byte más significativo en *cte_div* y hacer *Load_Hi* = 1 y *Load_Low* = 0. Asignar *Load_Hi = 0* y *Load_Low = 0*. Como se observa en la Figura anterior el reloj del transmisor Clk_Txs se genere al dividir la señal de reloj entrante entre 16*16*cte_div, si por ejemplo tenemos un cristal de 4.9125 Mhz y deseamos realizar una transmisión a 2400 BPS *cte_div* será igual a:

$$\mathrm{cte_{div}} = \frac{\mathrm{fin}}{\mathrm{BPS} * 256}$$

Que para nuestro ejemplo sería:

A continuación se muestrá el código de los módulos div16 y div_ms:

```
-- Inicio del módulo div16
Library Ieee;
Use Ieee.Std_Logic_1164.all;
Use Ieee.Std_Logic_Arith.all;
Use Ieee.Std_Logic_unsigned.all;
entity div16 is
port (
Reset: in STD_LOGIC;
Clk: in STD_LOGIC;
Clk_In: in STD_LOGIC;
Clk_Out: out STD_LOGIC
);
end div16;
architecture div16_arch of div16 is
Signal divider : Std_Logic_Vector( 3 downto 0 );
begin
Process( CLK, Reset )
Begin
if clk'event and clk = '1' then
```

```
if Reset = '1' then
divider <= "0000";
elsif Clk_In = '1' then
if divider = "1111" then
divider <= "0000";
Clk_Out <= '1';
else
divider <= divider + "0001";
Clk_Out <= '0';
end if;
end if;
end if;
end process;
end div16_arch;
--Inicio del módulo div_ms
Library Ieee;
Use Ieee.Std_Logic_1164.all;
Use Ieee.Std_Logic_Arith.all;
Use Ieee.Std_Logic_unsigned.all;
entity Div_ms is
port (
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
Clk_In : in STD_LOGIC;
Divider : in STD_LOGIC_Vector( 7 downto 0 );
Ld_Div_Low : in STD_LOGIC;
Ld_Div_Hi : in STD_LOGIC;
Clk_Out : out STD_LOGIC
);
end Div_ms;
architecture Div_ms_arch of Div_ms is
Signal count, Div_Factor : Std_Logic_Vector( 15 downto 0 );
begin
Process( CLK, Reset )
Begin
if clk'event and clk = '1' then
If Reset = '1' then
count <= ( others => '0' );
else
if Ld_Div_Low = '1' then
Div_Factor( 7 downto 0 ) <= Divider;
count <= ( others => '0' );
end if;
if Ld_Div_Hi = '1' then
Div_Factor( 15 downto 8 ) <= Divider;
count <= ( others => '0' );
end if;
if count = Div_Factor then
count <= ( others => '0' );
Clk_Out <= '1';
else
if Clk_In = '1' then
count <= count + 1;
Clk_Out <= '0';
end if;
end if ;
end if;
end if;
```

```
end process;
end Div_ms_arch;
```

Transmisor: Este módulo está encargado de generar la trama a la velocidad determinada en la señal Tx. El transmisor es bastante sencillo ya que lo único que debe hacer es realizar un corrimiento a la izquierda de un registro de 10 bits conformado de la siguiente forma:

Dígito 0 : 0 (Bit de Start).

Dígitos $1 - 9$: Dato a transmitir.

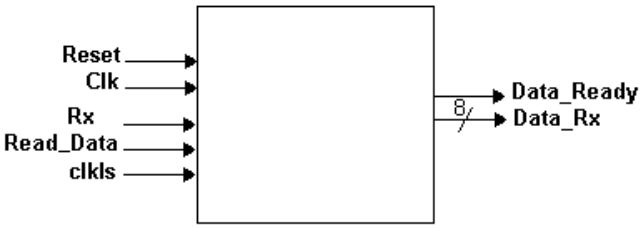
Dígito 10 : Bit de Stop.

El código en VHDL del transmisor se muestra a continuación:

```
Library Ieee;
Use Ieee.Std_Logic_1164.all;
Use Ieee.Std_Logic_Arith.all;
Use Ieee.Std_Logic_unsigned.all;
entity Buffer_TX is
port (
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
Load : in STD_LOGIC;
Shift : in STD_LOGIC;
Data_IN : in STD_LOGIC_VECTOR (7 downto 0);
Tx : out STD_LOGIC;
Tx_Empty : Buffer STD_LOGIC
);
end Buffer_TX;
architecture Buffer_TX_arch of Buffer_TX is
Signal Count_Tx : Std_Logic_Vector ( 3 downto 0 );
Signal Reg_Tx : Std_Logic_Vector( 9 downto 0 );
Signal Tx_On : Std_Logic; -- '1' indica transmisión en progreso
Bejín
-- Este proceso realiza un corrimiento cada vez que la señal Shift (Clk_Txs ) es igual a '1'
Process( CLK, Reset, Load, Shift )
Begin
if clk'event and clk = '1' then
If Reset = '1' then
Reg_Tx <= ( others => '1' );
Tx <= '1';
else
if Load = '1' and Tx_On = '0' then --Formación de la trama a transmitir.
Reg_Tx( 8 downto 1 ) <= Data_In;
Reg_Tx( 0 ) <= '0';
Reg_Tx( 9 ) <= '1';
else
if Shift = '1' then – Corrimiento de la trama.
Reg_Tx( 8 downto 0 ) <= Reg_Tx( 9 downto 1 );
Tx <= Reg_Tx( 0 );
end if;
end if;
end if;
end if;
end process;
Process( CLK, Reset, Load, Shift )
-- Este proceso controla el número de corrimientos realizados por el transmisor
Begin
if clk'event and clk = '1' then
If Reset = '1' then
Tx_On <= '0';
Tx_Empty <= '1';
Count_Tx <= "0000";
```

```
else
if Load = '1' and Tx_On = '0' then
Tx_Empty <= '0';
Tx_On <= '1';
elsif Shift = '1' then
if Tx_On = '1' then
Count_Tx <= Count_Tx + 1;
if Count_Tx = "1001" then
Count_Tx <= "0000";
Tx_On <= '0';
Tx_Empty <= '1';
else
Tx_On <= '1';
end if;
end if;
else
Tx_Empty <= Tx_Empty;
Tx_On <= Tx_On;
end if;
end if;
end if;
end process;
end Buffer_TX_arch;
```

Receptor: El receptor está encargado de generar los pulsos de muestreo en la mitad de cada Bit y detectar el inicio y fin de la trama (Bit de Start y Bit de Stop). Como puede verse en la Figura 43. El generador de pulsos de muestreo produce la señal *clkls* la cual es 16 veces más rápida que la señal Clk_Txs la cual se produce cada 1/Baud Rate. Por lo tanto el receptor debe esperar 8 ciclos de la señal clkls para leer los bits de datos y Stop. En la siguiente figura se muestra la interfaz y el diagrama de bloques del receptor.



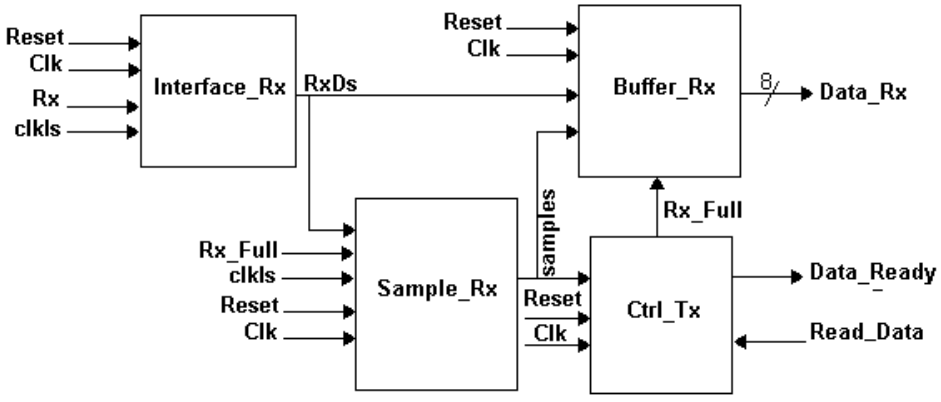


Figura 46. Diagrama de bloques del receptor.

Interface_Rx : Está encargada de sincronizar la señal de entrada *Rx* con la señal *clkls*. Como salida tiene la señal *RxDs*.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity Interface_Rx is
port (
```

```
Reset: in STD_LOGIC;
Clk: in STD_LOGIC;
clk_ms: in STD_LOGIC;
RxD: in STD_LOGIC;
RxDs: out STD_LOGIC
);
end Interface_Rx;
architecture Interface_Rx_arch of Interface_Rx is
Signal ifrxd : Std_Logic_Vector( 2 downto 0 );
begin
Process( CLK, Reset )
Begin
if clk'event and clk = '1' then
if Reset = '1' then
ifrxd <= "111";
else
if ( ifrxd(0) = ifrxd(2) ) and ( ifrxd(0) /= ifrxd(1) ) then
ifrxd(2) <= ifrxd(0);
else
ifrxd(2) <= ifrxd(1);
ifrxd(1) <= ifrxd(0);
ifrxd(0) <= RxD;
end if;
RxDs <= ifrxd(2);
end if;
end if;
end process;
end Interface_Rx_arch;
```

Sample_Rx: Este módulo está encargado de generar la señal de muestreo *samples*. La generación de pulsos inicia al detectar una transición de alto a bajo en la línea *RxDs* y finaliza cuando la señal *Rx_Full* sea igual a '1'.

```
library IEEE;
use IEEE.std_logic_1164.all;
Use Ieee.Std_Logic_Arith.all;
Use Ieee.Std_Logic_unsigned.all;
entity SAMPLE_RX is
port (
Reset: in STD_LOGIC;
CLK: in STD_LOGIC;
Rst_S: in STD_LOGIC;
Clkms: in STD_LOGIC;
RxDs: in STD_LOGIC;
Sample: out STD_LOGIC
);
end SAMPLE_RX;
architecture SAMPLE_RX_arch of SAMPLE_RX is
Signal cont_m : Std_Logic_Vector( 3 downto 0 ); -- Contador del modulo de muestreo
Signal Flag_Rx: Std_Logic; -- Flag de Recepcion en curso
begin
Process( CLK, Reset )
Begin
if clk'event and clk = '1' then
If Reset = '1' then
cont_m <= "0000";
sample <= '0';
Flag_Rx <= '0';
elsif Rst_S = '1' then
cont_m <= "0000";
```

```
sample <= '0';
Flag_Rx <= '0';
elsif Clkms = '1' then
if Flag_Rx = '0' and RxDs = '0' then -- Bit de Start
Flag_Rx <= '1'; -- Inicio de la recepcion
elsif Flag_Rx = '1' then
cont_m <= cont_m + "0001";
if cont_m = "0110" then
sample <= '1';
else
sample <= '0';
end if;
end if;
end if;
end if;
End Process;
end SAMPLE_RX_arch;
```

ctrl_tx: Esté módulo es básicamente un contador de pulsos de muestreo samples, cuando el conteo llega a 10 (Bit de Start, 8 bits de datos y Bit de Stor), la señal Rx_Full es igual a '1'.

```
library IEEE;
use IEEE.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
entity ctrl_tx is
port (
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
Load : in STD_LOGIC;
Shift : in STD_LOGIC;
Tx_Empty : Buffer STD_LOGIC
);
end ctrl_tx;
architecture ctrl_tx_arch of ctrl_tx is
Signal Count_Tx : Std_Logic_Vector ( 3 downto 0 );
Signal Tx_On : Std_Logic;
begin
Process( CLK, Reset, Load, Shift )
Begin
if clk'event and clk = '1' then
If Reset = '1' then
Tx_On <= '0';
Tx_Empty <= '1';
Count_Tx <= "0000";
else
if Load = '1' and Tx_On = '0' then
Tx_Empty <= '0';
Tx_On <= '1';
elsif Shift = '1' then
if Tx_On = '1' then
Count_Tx <= Count_Tx + 1;
if Count_Tx = "1001" then
Count_Tx <= "0000";
Tx_On <= '0';
Tx_Empty <= '1';
else
Tx_On <= '1';
end if;
end if;
```

```
else
Tx_Empty <= Tx_Empty;
Tx_On <= Tx_On;
end if;
end if;
end if;
end process;
end ctrl_tx_arch;
```

Buffer_Rx: Está encargado de realizar la conversión de seria a paralelo del dato de entrada.

```
library IEEE;
use IEEE.std_logic_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
entity Buffer_TX is
port (
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
Load : in STD_LOGIC;
Shift : in STD_LOGIC;
Data_IN : in STD_LOGIC_VECTOR (7 downto 0);
Tx : out STD_LOGIC;
Tx_Empty : Buffer STD_LOGIC
);
end Buffer_TX;
architecture Buffer_TX_arch of Buffer_TX is
Signal Count_Tx : Std_Logic_Vector ( 3 downto 0 );
Signal Reg_Tx : Std_Logic_Vector( 9 downto 0 );
Signal Tx_On : Std_Logic;
begin
Process( CLK, Reset, Load, Shift )
Begin
if clk'event and clk = '1' then
If Reset = '1' then
Reg_Tx <= ( others => '1' );
Tx <= '1';
else
if Load = '1' and Tx_On = '0' then
Reg_Tx( 8 downto 1 ) <= Data_In;
Reg_Tx( 0 ) <= '0';
Reg_Tx( 9 ) <= '1';
else
if Shift = '1' then
Reg_Tx( 8 downto 0 ) <= Reg_Tx( 9 downto 1 );
Tx <= Reg_Tx( 0 );
end if;
end if;
end if;
end if;
end process;
Process( CLK, Reset, Load, Shift )
Begin
if clk'event and clk = '1' then
If Reset = '1' then
Tx_On <= '0';
Tx_Empty <= '1';
Count_Tx <= "0000";
else
if Load = '1' and Tx_On = '0' then
```

```
Tx_Empty <= '0';
Tx_On <= '1';
elsif Shift = '1' then
if Tx_On = '1' then
Count_Tx <= Count_Tx + 1;
if Count_Tx = "1001" then
Count_Tx <= "0000";
Tx_On <= '0';
Tx_Empty <= '1';
else
Tx_On <= '1';
end if;
end if;
else
Tx_Empty <= Tx_Empty;
Tx_On <= Tx_On;
end if;
end if;
end if;
end process;
end Buffer_TX_arch;
```

A continuación se presenta la descripción estructural de la Uart:

```
library IEEE;
Use Ieee.Std_Logic_1164.all;
Use Ieee.Std_Logic_Arith.all;
Use Ieee.Std_Logic_unsigned.all;
entity uart is
port (
-- Senales comunes
Reset : in STD_LOGIC; -- Reset Activo Alto
Clk : in STD_LOGIC; -- Clock del sistema
-- Receptor
Rx : in STD_LOGIC; -- Linea de recepcion
Read_Data : in STD_LOGIC; -- '1' : Lectura del dato recibido
Data_Ready : out STD_LOGIC; -- '1' : Dato disponible
Data_Rx : out STD_LOGIC_VECTOR ( 7 downto 0 );
-- Transmisor
Data_Tx : in STD_LOGIC_Vector( 7 downto 0 ); -- Dato a transmitir
Start_Tx : in STD_LOGIC; -- '1' : Inicio de la transmision
Tx : Out STD_LOGIC; -- Linea de transmision
Tx_Empty : Buffer STD_LOGIC -- '1' Transmisor disponible.
);
end UART;
architecture UART_arch of UART is
Signal Hi: Std_Logic;
Component SAMPLE_RX
port (
Reset : in STD_LOGIC;
CLK : in STD_LOGIC;
Rst_S : in STD_LOGIC;
Clkms : in STD_LOGIC;
RxDs : in STD_LOGIC;
Sample : out STD_LOGIC
);
end Component SAMPLE_RX;
Component Ctrl_Rx
port (
Reset : in STD_LOGIC;
```

```
Clk : in STD_LOGIC;
Samples : in STD_LOGIC;
Read_Data : in STD_LOGIC;
Data_Ready: out STD_LOGIC;
Rx_Full : Buffer STD_LOGIC
);
end Component Ctrl_Rx;
Component Buffer_Rx
port (
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
RxDs : in STD_LOGIC;
Samples : in STD_LOGIC;
End_Rx : in STD_LOGIC;
Data_Rx : out STD_LOGIC_VECTOR (7 downto 0)
);
end Component Buffer_Rx;
Component Interface_Rx
port (
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
clk_ms : in STD_LOGIC;
RxD : in STD_LOGIC;
RxDs : out STD_LOGIC
);
end Component Interface_Rx;
Component div16
port (
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
Clk_In : in STD_LOGIC;
Clk_Out : out STD_LOGIC
);
end Component div16;
Component pulso
port (
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
Data_asyn : in STD_LOGIC;
Data_syn : out STD_LOGIC
);
end Component pulso;
Component Div_ms
port (
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
Clk_In : in STD_LOGIC;
Divider : in STD_LOGIC_Vector( 7 downto 0 );
Ld_Div_Low : in STD_LOGIC;
Ld_Div_Hi : in STD_LOGIC;
Clk_Out : out STD_LOGIC
);
end Component Div_ms;
Component Control_RX
GENERIC(
Div_Hi : Integer := 0;
Div_Low : Integer := 2 );
port (
```

```
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
Ld_Div_Low : out STD_LOGIC;
Ld_Div_Hi : out STD_LOGIC;
Data_Out : out STD_LOGIC_VECTOR (7 downto 0)
);
end Component Control_RX;
Component Buffer_TX
port (
Reset : in STD_LOGIC;
Clk : in STD_LOGIC;
Load : in STD_LOGIC;
Shift : in STD_LOGIC;
Data_IN : in STD_LOGIC_VECTOR (7 downto 0);
Tx : out STD_LOGIC;
TX_Empty : Buffer STD_LOGIC
);
end Component Buffer_TX;
Signal Rx_Full, RxDs, samples, sample, clk_pres, clkls, clkl : Std_Logic;
Signal Clk_Tx, Clk_Txs : STD_LOGIC;
Signal carga_div_low, carga_div_Hi : STD_LOGIC;
Signal Data_In : STD_LOGIC_VECTOR( 7 downto 0 );
begin
Hi <= '1';
--
************************************************************************************************
-- * Generador de Frecuencia: La frecuencia de muestreo debe ser 16 veces mayor que *
-- * la velocidad de transmision. *
--
************************************************************************************************
A1 : div16
port map( Reset, CLK, Hi, clk_pres );
A2 : div_ms
port map( Reset, CLK, clk_pres,data_in, carga_div_low, carga_div_Hi, clkl );
A3 : pulso
port map( Reset, CLK, clkl, clkls );
A4 : div16
port map( Reset, CLK, clkls, Clk_Tx );
A5 : pulso
port map( Reset, CLK, Clk_Tx, Clk_Txs );
-- **************
-- * Receptor *
-- **************
-- Sincroniza la señal de entrada Rx con clkls, RxDs señal de salida sincronizada
A6 : Interface_Rx
port map( Reset, CLK, clkls, Rx, RxDs );
-- Detecta Bit de Start y genera una señal (sample) que indica cuando se debe
-- leer un bit. Sample tiene una duracion de un ciclo de reloj CLK.
A7 : Sample_Rx
port map( Reset, CLK, Rx_Full, clkls, RxDs, sample );
-- Sincroniza la senal de entrada sample, samples senal sincronizada.
A8 : pulso
port map( Reset, CLK, sample, samples );
-- Control del numero de bits, cuando se reciben 10 pulsos de la señal sample
-- Rx_Full se hace igual a '1'.
A9 : ctrl_rx
port map( Reset, CLK, samples, Read_Data, Data_Ready, Rx_Full );
-- Realiza la conversión de serie a paralelo del dato recibido.
```

```
A10: Buffer_Rx
port map( Reset, CLK, RxDs, samples, Rx_Full, Data_Rx );
-- Carga los registros del divisor para funcionar a 9600 BPS
A11: Control_RX
generic map( 0, 2 )
port map( Reset, CLK, carga_div_low, carga_div_Hi, Data_In );
-- **************
-- * Transmisor *
-- **************
A12: Buffer_Tx
port map( Reset, Clk, Start_Tx, Clk_Txs, Data_Tx, Tx, Tx_Empty );
end UART_arch;
```

El test bench de la UART se muestra a continuación, junto con su correspondiente simulación.

```
Use Ieee.Std_Logic_1164.all;
Use Ieee.Std_Logic_Arith.all;
Use Ieee.Std_Logic_unsigned.all;
ENTITY testbench IS
END testbench;
ARCHITECTURE behavior OF testbench IS
-- Component Declaration
Component UART
port (
-- Senales comunes
Reset : in STD_LOGIC; -- Reset Activo Alto
Clk : in STD_LOGIC; -- Clock del sistema
-- Receptor
Rx : in STD_LOGIC; -- Linea de recepcion
Read_Data : in STD_LOGIC; -- '1' : Lectura del dato recibido
Data_Ready : out STD_LOGIC; -- '1' : Dato disponible
Data_Rx : out STD_LOGIC_VECTOR ( 7 downto 0 );
-- Transmisor
Data_Tx : in STD_LOGIC_Vector( 7 downto 0 ); -- Dato a transmitir
Start_Tx : in STD_LOGIC; -- '1' : Inicio de la transmision
Tx : Out STD_LOGIC; -- Linea de transmision
Tx_Empty : Buffer STD_LOGIC -- '1' Transmisor disponible.
);
end component;
SIGNAL Reset : STD_LOGIC;
SIGNAL Clk : STD_LOGIC;
SIGNAL Rx : STD_LOGIC;
SIGNAL Read_Data : STD_LOGIC;
SIGNAL Data_Ready : STD_LOGIC;
SIGNAL Data_Rx : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL Data_Tx : STD_LOGIC_VECTOR (7 downto 0);
SIGNAL Start_Tx : STD_LOGIC;
SIGNAL Tx : STD_LOGIC;
SIGNAL Tx_Empty : STD_LOGIC;
constant ncycles : integer := 999999999;
constant halfperiod : time := 101.72 ns;
constant tbit : integer := 512;
BEGIN
-- Component Instantiation
uut: Uart PORT MAP(
Reset => Reset,
Clk => Clk,
Rx => Rx,
Read_Data => Read_Data,
Data_Ready => Data_Ready,
```

```
Data_Rx => Data_Rx,
Data_Tx => Data_Tx,
Start_Tx => Start_Tx,
Tx => Tx,
Tx_Empty => Tx_Empty
);
-- Generacion del Reloj
Clock_Source: process
begin
for i in 0 to ncycles*10 loop -- Genera ncyclos de periodo 10 ns
clk <= '0';
wait for halfperiod;
clk <= '1';
wait for halfperiod;
end loop;
wait;
end process Clock_Source;
Receiver : PROCESS
BEGIN
Reset <= '1';
Rx <= '1';
Read_Data <= '0';
wait until clk'event and clk = '1';
wait until clk'event and clk = '1';
Reset <= '0';
wait until clk'event and clk = '1';
wait until clk'event and clk = '1';
wait until clk'event and clk = '1';
-- BIT DE START
Rx <= '0';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D0
Rx <= '1';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D1
Rx <= '1';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D2
Rx <= '0';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D3
Rx <= '1';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D4
Rx <= '0';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
```

```
-- D5
Rx <= '0';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D6
Rx <= '1';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D7
Rx <= '1';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- BSTOP
Rx <= '1';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- SEGUNDO BIT ENVIADO
-- BIT DE START
Rx <= '0';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D0
Rx <= '0';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D1
Rx <= '0';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- LECTURA DEL PRIMER BIT RECIVIDO
Read_Data <= '1';
wait until clk'event and clk = '1';
Read_Data <= '0';
-- D2
Rx <= '1';
for i in 0 to tbit loop
wait until clk'event and clk = = '1';
end loop;
-- D3
Rx <= '0';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D4
Rx <= '1';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- D5
Rx <= '1';
for i in 0 to tbit loop
```

```
wait until clk'event and clk = '1';
end loop;
-- D6
Rx <= '0';
for i in 0 to tbit loop
wait until clk'event and clk = = '1';
end loop;
-- D7
Rx <= '0';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
-- BSTOP
Rx <= '1';
for i in 0 to tbit loop
wait until clk'event and clk = '1';
end loop;
wait; -- will wait forever
END PROCESS;
Transmiter : Process
Begin
for i in 0 to 10 loop
wait until clk'event and clk ='1';
end loop;
Data_ Tx <= "10101010";
wait until clk'event and clk = '1';
Start_ Tx <= '1';
wait until clk'event and clk = '1';
Start_ Tx <= '0';
wait until clk'event and clk = = '1';
-- Envía un nuevo dato antes de terminar la transmisión
for i in 0 to 1000 loop
wait until clk'event and clk = '1';
end loop;
Data_ Tx <= "11100011";
wait until clk'event and clk = '1';
Start_ Tx <= '1';
wait until clk'event and clk = '1';
Start_ Tx <= '0';
wait until clk'event and clk = '1';
-- Espera a que el transmisor este listo para la transmision
wait until Tx_Empty = '1';
for i in 0 to 1000 loop
wait until clk'event and clk = '1';
end loop;
Data_ Tx <= "11100011";
wait until clk'event and clk = '1';
Start_ Tx <= '1';
wait until clk'event and clk = '1';
Start_ Tx <= '0';
wait until clk'event and clk = '1';
wait;
End Process;
END;
```

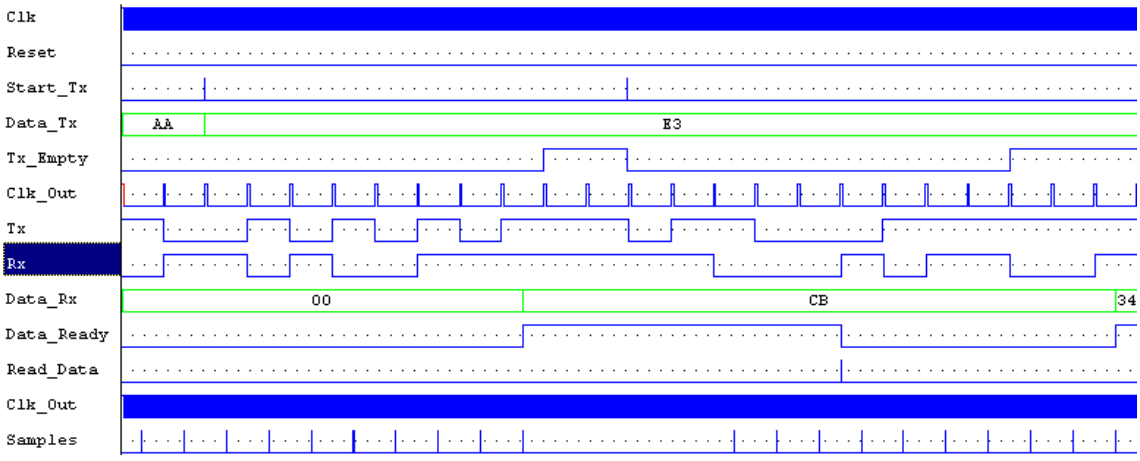
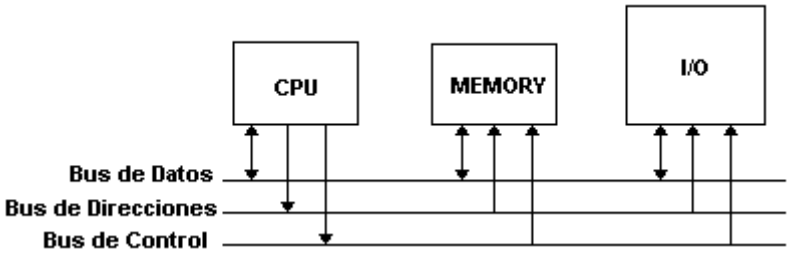


Figura 45. Simulación de la UART.

EJEMPLO 7: Diseño de un Computador Sencillo

Un computador está formado principalmente por tres unidades: La Unidad Central de Procesamiento; La memoria que está encargada de almacenar las instrucciones del programa y datos; Y las unidades de entrada salida cuya función es permitir el intercambio de información con el exterior.



Internamente la CPU está dividida en: El camino de datos o *Datapath* el cual a su vez está formado por:

PC: (Program Counter) Encargado de almacenar la dirección de memoria de la instrucción que se está ejecutando actualmente.

IR: (Instruction Register) Encargado de almacenar el código de la instrucción en ejecución.

ACC: (Acumulator) Utilizado para realizar cálculos y como almacenamiento temporal de datos de programa.

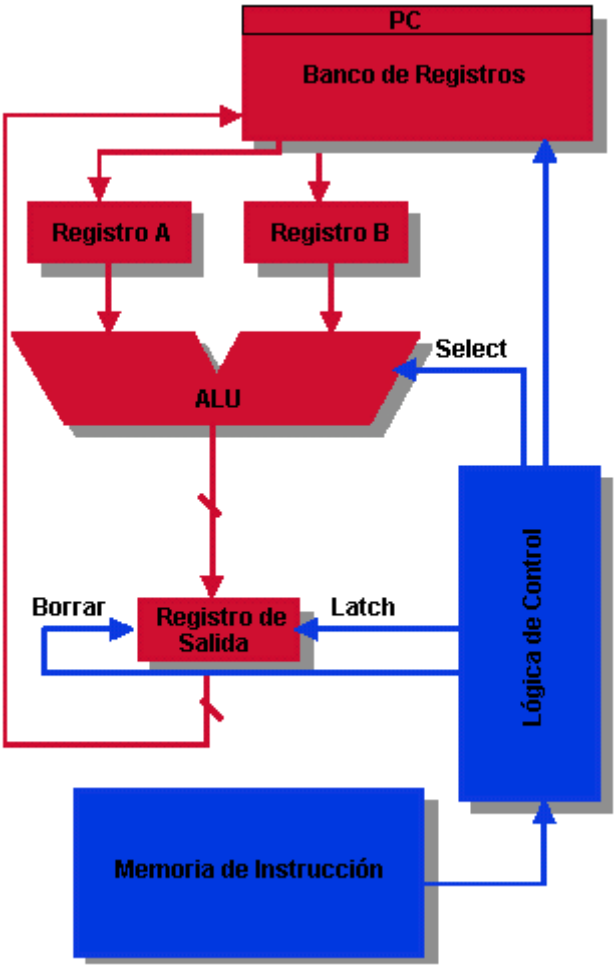
MAR: (Memory Address Register) Utilizado para el direccionamiento de la memoria de programa.

y la lógica de control que está encargada de manejar el *Datapath* dependiendo de la instrucción en ejecución.

El primer paso en el diseño de un computador es definir las operaciones que se podrán realizar, este conjunto de operaciones recibe el nombre de *Set de Instrucciones*. Nuestro computador será capaz de realizar operaciones de Suma (ADD) , Resta (SUB), LOAD y STORE; Debido a que tenemos 4 instrucciones serán necesarios dos bits para su codificación (opcode), es importante notar que cada instrucción debe tener un código único:

Operación	Código
ADD	00
SUB	01
LOAD	10
STORE	11

La siguiente figura muestra una arquitectura simplificada del Computador:



La función LOAD carga el contenido de un registro del Banco en el Registro A o en el Registro B, razón por la cual su codificación debe indicar en cual registro se desea almacenar la información para esto debemos utilizar un tercer bit que nos indique el registro:

LOAD REG A = 100

LOAD REG B = 101

Por otro lado las funciones STORE y LOAD deben indicar hacia y de donde se debe transferir la información por lo que estas funciones deben indicar el registro origen o fuente, esto se logra asignándole a cada registro una dirección de memoria e indicando la dirección del mismo en el código de la instrucción. La siguiente figura muestra la forma de decodificar la dirección dentro de la instrucción.

ADD	0	0	-	-	-	-	-	-
SUB	0	1	-	-	-	-	-	-
LOAD	1	0	X	X	X	X	X	X
STORE	1	1	-	X	X	X	X	X

Donde - indica Don´t Care y **X** 0 o 1.

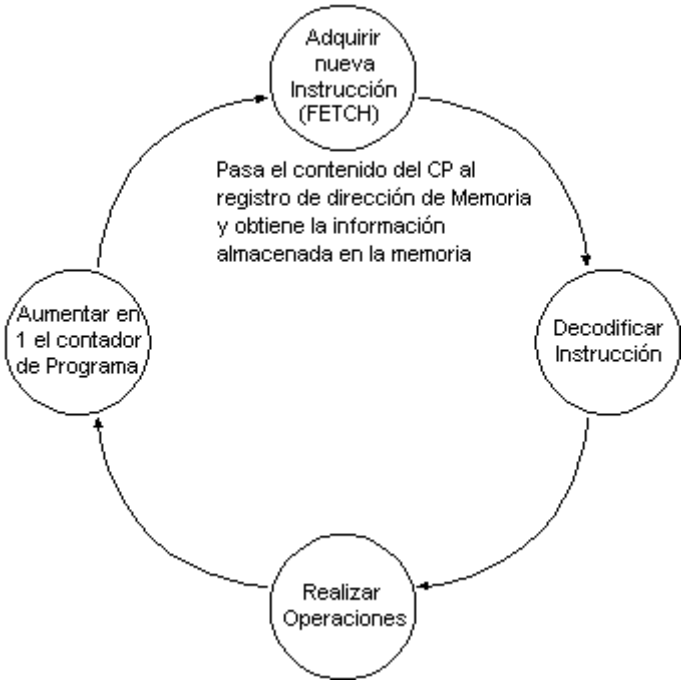
Para comprender mejor el funcionamiento de nuestro computador consideremos el siguiente ejemplo:

Load regA, 0 ; regA [F0DF?] Reg[0]

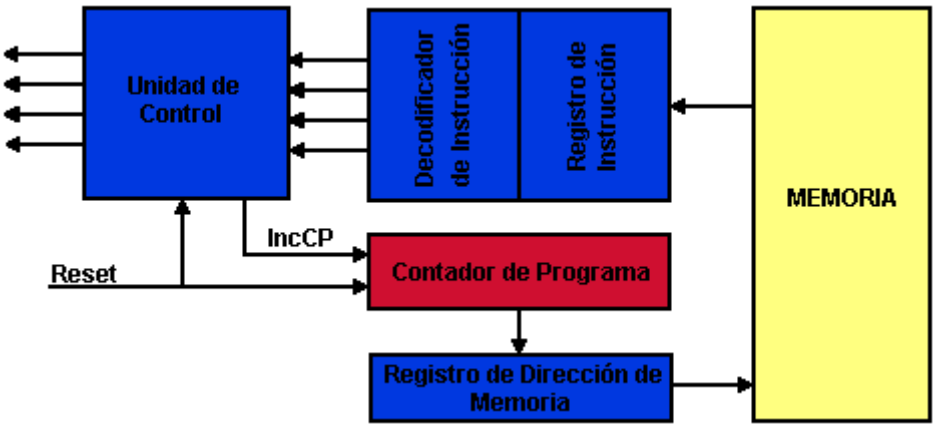
Load regB, 1 ; regB [F0DF?] Reg[1]

Add ; result [F0DF?] regA + regB
Store 2 ; reg[2][F0DF?] resultReg
En binario el código es:
1 0 0 0 0 0 0 0 *;Carga el contenido del registro 0 en regA.*
1 0 1 0 0 0 0 1 *;Carga el contenido del registro 1 en regB.*
0 0 0 0 0 0 0 0 *;Realiza regA + regB y lo almacena en result.*
1 1 0 0 0 0 1 0 *;Carga result en el registro 2.*
Con la codificación anterior no se podría cargar datos desde la memoria de programa al Banco de Registro, y nuestro computador estaría aislado del mundo externo. Para solucionar esto debemos incluir otra instrucción que nos permita el ingreso de datos desde el exterior. Podemos ampliar la función STORE de la siguiente forma:
1 1 0 X X X X X Almacena el contenido del registro de salida de la ALU en la dirección XXXXX
1 1 1 X X X X X Almacena el contenido de la memoria a la dirección XXXXX.
Para mayor claridad en la codificación de las instrucciones es mejor dividir la instrucción STORE en STORE (110) y STOREX (111) para indicar que la fuente es externa.
Para entender el funcionamiento de esta instrucción consideremos el siguiente programa:
Load regA, 0 ; regA [F0DF?] Reg[0]
Storex 1 ; reg1 [F0DF?] PC+1
Load regB, 1 ; regB [F0DF?] Reg[1]
Add ; result [F0DF?] regA + regB
Store 2 ; mem[2][F0DF?] resultReg
En binario el código es:
1 0 0 0 0 0 0 0 *;Carga el contenido del registro 0 en regA.*
1 1 1 0 0 0 0 1 *;Carga 10101010 (PC+1: línea siguiente)en reg1*
1 0 1 0 1 0 1 0 *;Dato a almacenar*
1 0 1 0 0 0 0 1 *;Carga el contenido del registro 1 en regB.*
0 0 0 0 0 0 0 0 *;Realiza regA + regB y lo almacena en result.*
1 1 0 0 0 0 1 0 *;Carga result en el registro 2.*
Para poder realizar estas operaciones la lógica de control debe ser capáz de distinguir las diferentes instrucciones y actuar consecuentemente, es decir asignar los valores lógicos adecuados para cada función. Los pasos que se deben realizar para la lectura de una instrucción son los siguientes:

1. Inicializar el Contador de Programa (PC = Dirección base) para leer la primera instrucción.
2. Transferir el contenido del Contador de Programa al Registro de Dirección de Memoria (MAR)
3. El contenido de la Memoria debe ser transferido al Registro de Instrucción (IR).
4. Decodificar la Instrucción y realizar las operaciones adecuadas.
5. Aumentar en 1 el contenido del contador de Programa en uno y repetir el proceso para la siguiente instrucción.



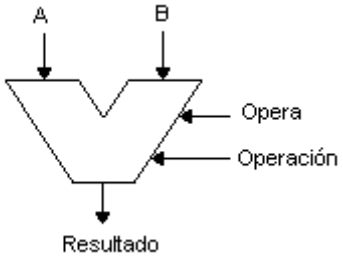
La siguiente figura muestra una arquitectura que nos servirá para realizar las operaciones de la lógica de control. La unidad de control es la encargada de sincronizar las tareas que deben realizarse para ejecutar las instrucciones sobre el *datapath*.



Implementación del *Datapath*

Una vez diseñada la arquitectura de nuestro computador procedemos a la implementación de sus componentes, iniciando por el *datapath*; El cual está compuesto de:

ALU: Esta unidad está encargada de realizar las operaciones aritméticas y lógicas requeridas por las instrucciones, de acuerdo a nuestro set de instrucciones, necesitamos únicamente de la SUMA y de la RESTA (para set de instrucciones más complicados es necesario incluir las operaciones necesarias tales como: Complemento, corrimientos, etc). La siguiente figura muestra la interfaz de la ALU.



Y su código correspondiente en VHDL es:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity ALU is
Port ( A : in std_logic_vector(7 downto 0);
B : in std_logic_vector(7 downto 0);
Opera : in std_logic;
Operacion : in std_logic_vector(1 downto 0);
Resultado : out std_logic_vector(7 downto 0));
end entity ALU;
architecture RT of ALU is
begin
Process ( A, B, Opera, Operacion )
Begin
If Opera = '1' then
Case Operacion is
When "00" => -- Suma
Resultado <= A + B;
When "01" => -- Resta
Resultado <= A - B;
When "10" => -- And logico
Resultado <= A and B;
When "11" => -- Or logico
Resultado <= A or B;
When Others => -- Error
Resultado <= "XXXXXXXX";
end case;
else -- Si no hay operación el resultado es A
Resultado <= A;
end if;
End Process;
end architecture RT;
```

Banco de Registros: El banco de registro posee 4 registros de propósito general en los que se almacenan datos provenientes de la memoria y de resultados temporales de las operaciones. Este módulo tiene la interfaz que se muestra en la siguiente figura.

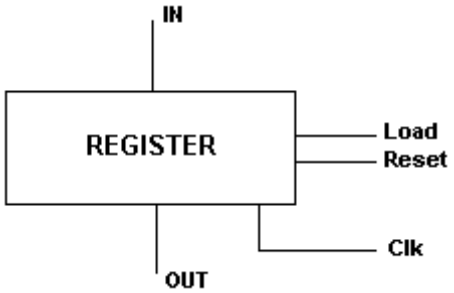
Para realizar una escritura se debe colocar la señal *Write* en '1' e indicar el registro destino asignándole el valor adecuado a *SelWriteReg* y esperar un flanco de subida en la señal del reloj. El proceso de lectura es bastante sencillo, basta con indicar el registro a leer (asignándole un valor a *SelReadReg*). El código en VHDL del Banco de Registros se muestra a continuación.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity RagBank is
Port ( Clk : in std_logic;
Init : in std_logic;
DataWrite : in std_logic_vector(7 downto 0);
SelWriteReg : in std_logic_vector(1 downto 0);
Write : in std_logic;
```

```
SelReadReg : in std_logic_vector(1 downto 0);
DataRead : out std_logic_vector(7 downto 0));
end RagBank;
architecture RT of RagBank is
begin
Process(Clk, Init, Write)
type RegBank is array ( 0 to 3 ) of std_logic_vector( 7 downto 0 );
variable Registers: RegBank;
Begin
If Init = '1' then
for i in 0 to 3 loop
Registers( i ) := "00000000";
end loop;
elsif Write = '1' then
if Clk'event and clk = '1' then
Registers( conv_integer( SelWriteReg ) ) := DataWrite;
end if;
end if;
DataRead <= Registers( conv_integer( SelReadReg ) );
End Process;
end RT;
```

Nótese que se utilizó la función *conv_integer* la cual convierte a entero un valor dado en *std_logic_vector*, esto se debe hacer ya que el subíndice de un arreglo debe ser de tipo entero.

Registros A, B y de salida: Estos registros están encargados de almacenar los operandos y el resultados de las operaciones de la ALU. Su interfaz se muestra en la siguiente figura.



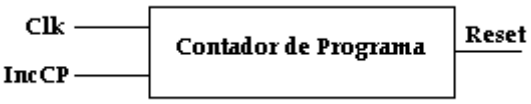
El código en VHDL de este registro es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity Reg is
Port ( Clk : in std_logic;
INR : in std_logic_vector(7 downto 0);
Reset : in std_logic;
Load : in std_logic;
OUTR : out std_logic_vector(7 downto 0));
end Reg;
architecture RT of Reg is
begin
Process( clk, load, reset )
Begin
if Reset = '1' then
OUTR <= "00000000";
else
if clk'event and clk = '1' then
if load = '1' then
OUTR <= INR;
end if;
end if;
```

```
end if;
End Process;
end RT;
```

Implementación de la lógica de control:

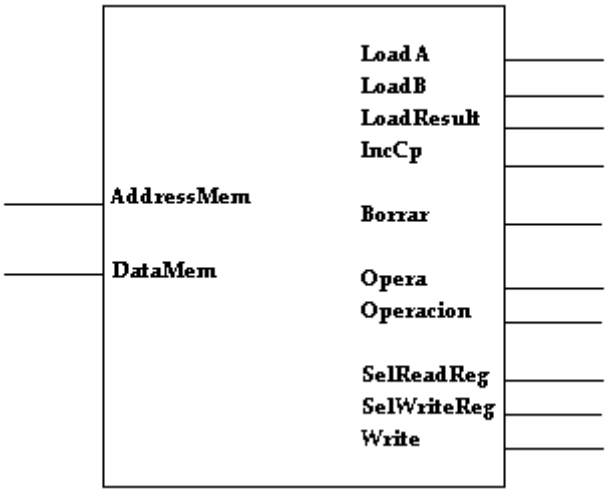
Contador de Programa: El contador de programa está encargado de mantener la dorección de memoria donde se encuentra la última instrucción ejecutada.



El código en VHDL del Contador de Programa es el siguiente:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity ProgCount is
Port ( Clk : in std_logic;
Reset : in std_logic;
IncCP : in std_logic;
OutCP : buffer std_logic_vector(7 downto 0));
end ProgCount;
architecture Behavioral of ProgCount is
begin
Process( Clk, Reset, IncCP )
Begin
if Reset = '1' then
OutCP <= "00000000";
else
if IncCP = '1' then
if clk'event and clk = '1' then
OutCP <= OutCP + "00000001";
end if;
else
OutCP <= OutCP;
end if;
end if;
End Process;
end Behavioral;
```

Registro de Instrucciones Decodificador y Unidad de Control: Estos módulos están encargados de realizar todas las microinstrucciones necesarias para realizar las instrucciones. Este módulo tiene la interfaz mostrada en la siguiente figura.



Y el código en VHDL de la lógica de control se muestra a continuación.

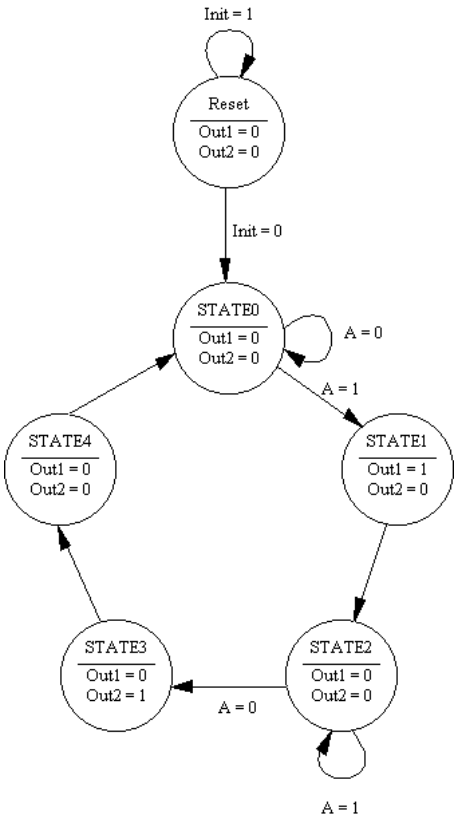
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity LogControl is
Port ( --
Clk : in std_logic;
SysReset : in std_logic;
-- Memory Interface
DataMem : in std_logic_vector(7 downto 0);
-- Specific Register Control
LoadA : out std_logic;
LoadB : out std_logic;
LoadResult : out std_logic;
LoadMAR : out std_logic;
IncCP : out std_logic;
Erase : out std_logic;
-- ALU Control
Operate : out std_logic;
Operation : out std_logic_vector(1 downto 0);
-- Register Bank Control
SelReadReg : out std_logic_vector(1 downto 0);
SelWriteReg : out std_logic_vector(1 downto 0);
Write : out std_logic);
end LogControl;
architecture Behavioral of LogControl is
type state_type is ( ResetPC, Fetch, Decode, PCOperation, IncrementPC );
Signal State : State_type;
type Operation_type is ( OPAdd, OPSub, OPLoad, OPStore, OPStorex );
Signal Oper : Operation_type;
Signal IR : std_logic_vector( 7 downto 0 );
type Inter_type is ( onein, twoin, threein );
Signal Inter : Inter_type;
begin
Process ( Clk, SysReset )
Begin
if SysReset = '1' then
state <= ResetPC;
Inter <= onein;
else
if ( Clk'event and Clk = '1' ) then
Case State is
when ResetPC =>
Erase <= '1'; LoadA <= '0';
LoadB <= '0'; LoadResult <= '0';
IncCp <= '0'; Operate <= '0';
Write <= '0'; LoadMAR <= '1';
State <= Fetch;
when Fetch =>
Erase <= '0'; LoadA <= '0';
LoadB <= '0'; LoadResult <= '0';
IncCp <= '0'; Operate <= '0';
Write <= '0'; LoadMAR <= '1';
state <= Decode;
IR <= DataMem;
when Decode =>
Erase <= '0'; LoadA <= '0';
```

```
LoadB <= '0'; LoadResult <= '0';
IncCp <= '0'; Operate <= '0';
Write <= '0'; LoadMAR <= '0';
Case IR( 7 downto 5 ) is
when "00X" =>
Oper <= OPAdd;
when "01X" =>
Oper <= OPSub;
when "10X" =>
Oper <= OPLoad;
when "110" =>
Oper <= OPStore;
when "111" =>
Oper <= OPStorex;
when others =>
end case;
State <= PCOperation;
Inter <= onein;
when PCOperation =>
Case Oper is
when OPAdd=>
Erase <= '0'; LoadA <= '0';
LoadB <= '0'; LoadResult <= '1';
IncCp <= '0'; Operate <= '1';
Write <= '0'; LoadMAR <= '0';
Operation <= "00";
State <= IncrementPC;
when OPSub=>
Erase <= '0'; LoadA <= '0';
LoadB <= '0'; LoadResult <= '1';
IncCp <= '0'; Operate <= '1';
Write <= '0'; LoadMAR <= '0';
Operation <= "01";
State <= IncrementPC;
when OPLoad =>
Erase <= '0'; LoadResult <= '0';
IncCp <= '0'; Operate <= '0';
Write <= '0'; LoadMAR <= '0';
Case Inter is
when onein =>
SelReadReg <= IR( 1 downto 0 );
Inter <= twoin;
when twoin =>
if IR( 5 ) = '1' then
LoadB <= '1';
LoadA <= '0';
else
LoadB <= '0';
LoadA <= '1';
end if;
Inter <= threein;
when threein =>
LoadB <= '0';
LoadA <= '1';
State <= IncrementPC;
Inter <= onein;
End Case;
when OPStore=>
```

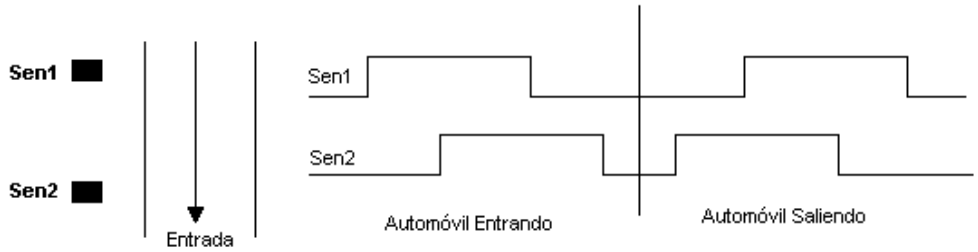
```
Erase <= '0'; LoadA <= '0';
LoadB <= '0'; LoadResult <= '0';
IncCp <= '0'; Operate <= '0';
Write <= '0'; LoadMAR <= '0';
Case Inter is
when onein =>
SelWriteReg <= IR( 1 downto 0 );
write <= '1';
Inter <= twoin;
when twoin =>
Write <= '0';
State <= IncrementPC;
Inter <= onein;
when others =>
Inter <= onein;
End Case;
when OPStorex=>
Erase <= '0'; LoadA <= '0';
LoadB <= '0'; LoadResult <= '0';
IncCp <= '0'; Operate <= '0';
Write <= '0'; LoadMAR <= '0';
end case;
when IncrementPC =>
Erase <= '0'; LoadA <= '0';
LoadB <= '0'; LoadResult <= '0';
IncCp <= '1'; Operate <= '0';
Write <= '0'; LoadMAR <= '1';
state <= Fetch;
end case;
end if;
end if;
end Process;
end Behavioral;
```

2.3. EJERCICIOS

1. Escribir el código en VHDL de la máquina de estados representada por el siguiente diagrama:

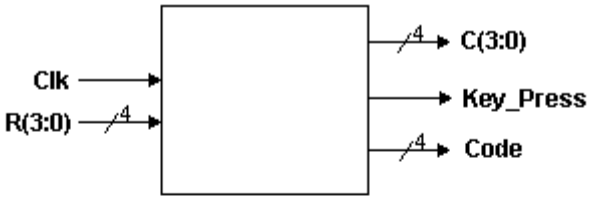


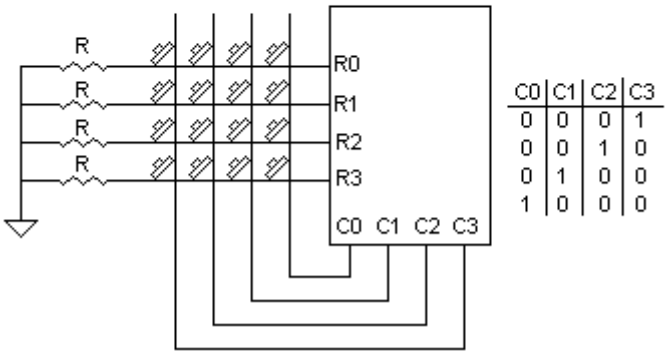
1. En un estacionamiento se cuenta con dos sensores que deben detectar el ingreso o salida de vehículos. Cuando un vehículo se encuentra frente a uno de los sensores genera un nivel lógico '1'. En la siguiente figura se muestra el diagrama de tiempos de las señales generadas por los sensores al entrar y al salir un vehículo.



El sistema debe entregar dos señales UP y DOWN las cuales se colocan en un nivel lógico alto cuando se detecta la secuencia completa de entrada y salida de un automóvil respectivamente.

3. Diseñar el decodificador del teclado matricial que se muestra en la siguiente figura. Cuando no se tiene ninguna tecla presionada, en las entradas R0 a R3 se tiene una lectura de "0000". Si por ejemplo se presiona la tecla superior izquierda, se recibirá un '1' lógico en R0 cuando la secuencia de las columnas C0 a C3 sea "1000". Por lo tanto un valor lógico alto en las entradas R0 a R3 indica la fila de la tecla oprimida, pero no sabemos cual de las cuatro teclas de la fila está presionada. Para averiguarlo debemos colocar un '1' lógico en solo una de las columnas y rotarlo por todas ellas. R0 será igual a '1' sólo cuando se coloque el '1' lógico en la columna de la tecla oprimida y '0' en los otros casos.

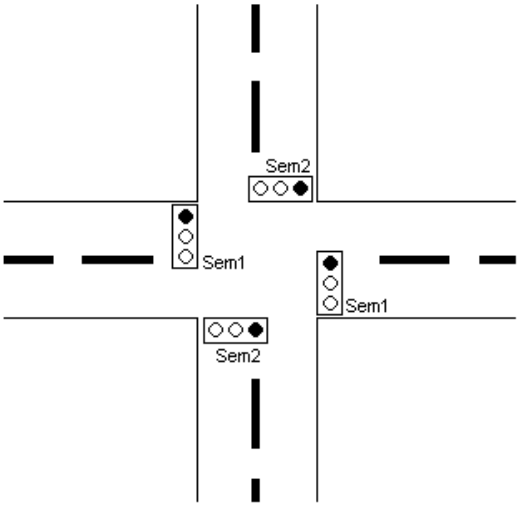




C0	C1	C2	C3
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0

Una vez detectada la tecla oprimida el decodificador debe generar un código único para cada una de ellas y debe indicar el evento haciendo *key_press = '1'*.

1. Diseñar el sistema de control de los semáforos de un cruce de vehículos.



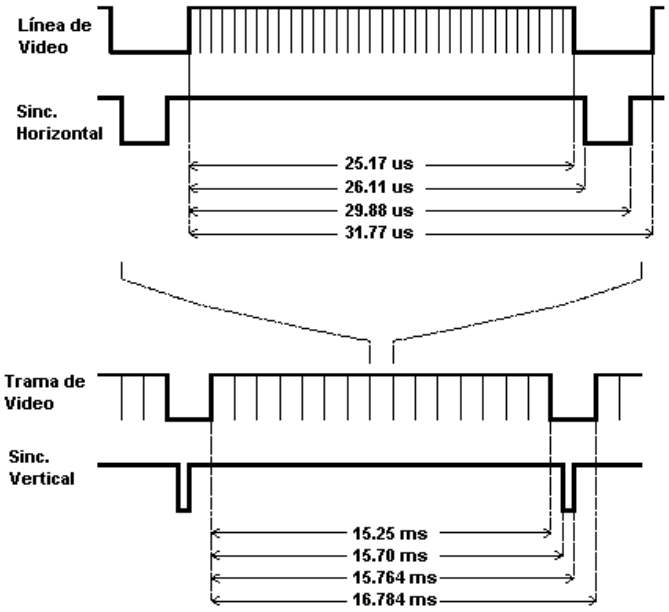
Sem1			Sem2		
Green	Yellow	Red	Green	Yellow	Red
1	0	0	0	0	1
1	1	0	0	0	1
0	1	0	0	0	1
0	0	1	0	0	1
0	0	1	0	1	1
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	0
0	0	1	0	1	0
0	0	1	0	0	1
0	1	1	0	0	1
0	1	0	0	0	1
1	0	0	0	0	1

1. Diseñar un sistema que sea capáz de generar caracteres ASCII en un monitor VGA. Una trama de video VGA está compuesta por 480 líneas de 640 pixels. El monitor cuenta con las siguientes señales de control:

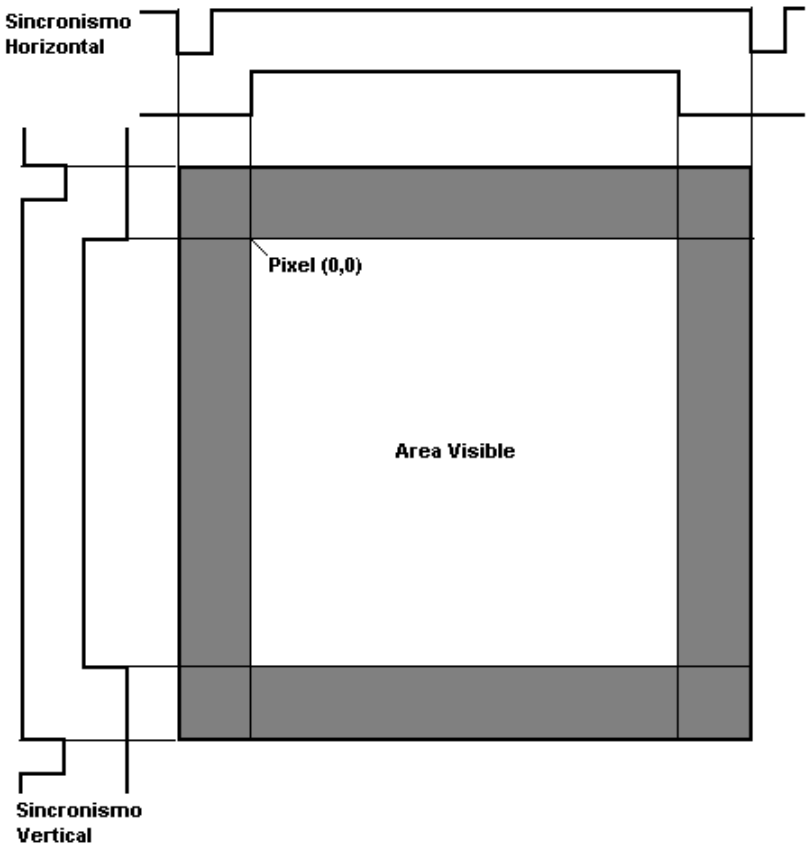
1.

   - Red, Green, Blue : Entradas de señal análoga de color, estas entradas tienen un rango de 0 a 0.7 V
   - Sincronismo Horizontal : Indica el principio y fín de cada línea de video (480 líneas).
   - Sincronismo Vertical : Indica el principio y fin de cada trama de video.

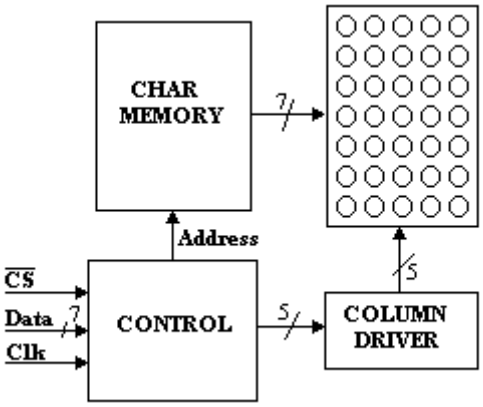
El flanco de bajada de la señal de sincronismo horizontal indica el inicio de la línea y al colocar en alto esta señal se asegura que la información contenida en la línea de video (Red, Green, Blue) se despliegue en la parte visible de la pantalla. En la siguiente figura puede verse la forma de onda y el diagrama de tiempos de esta señal.



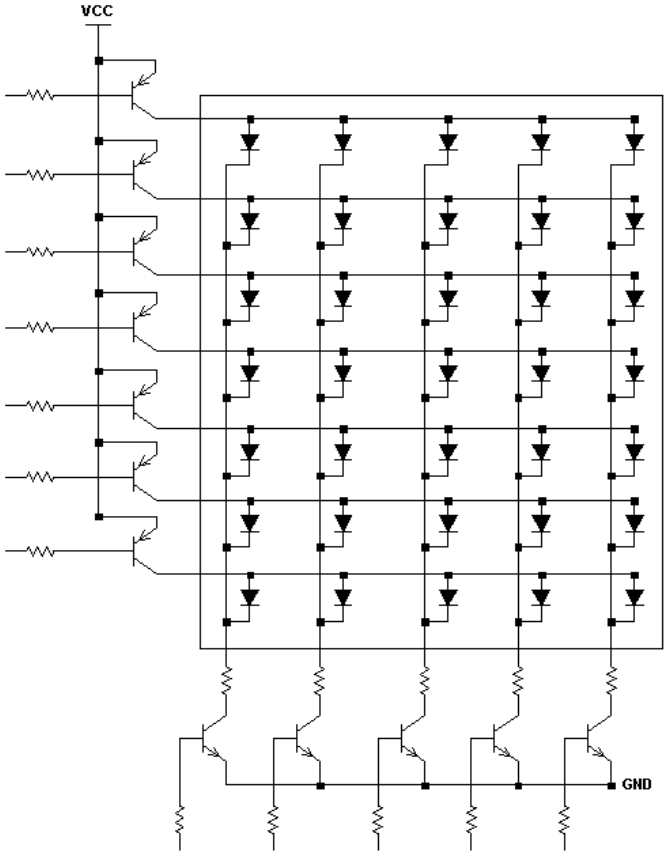
De igual forma el flanco de bajada en la señal de sincronismo vertical indica el inicio y fin de la trama de video formada por 480 líneas. Como puede verse en la figura anterior una línea tiene una duración de 31.77 us y las 480 líneas tiene una duración de 31.77 us x 480 = 15.25 ms.
En la siguiente figura se ilustra el funcionamiento del monitor VGA y se indican las zonas no visibles de la pantalla.



1. Diseñar el control de un display inteligente. Este display tiene como elemento de visualización una matriz de leds de 7x5. Se desea visualizar los caracteres ASCII desde el 20H ‘ ‘ hasta el 7EH ‘ ~ ’. La siguiente figura muestra el diagrama de bloques propuesto.



Debido a que los dispositivos lógicos programables no pueden suministrar la corriente necesaria para manejar los leds, es necesario utilizar el siguiente circuito para manejar la matriz:



El contenido de la memoria debe ser el siguiente:

' '000h,000h,000h,000h,000h
'!'000h,000h,0FAh,000h,000h
'"'000h,0E0h,000h,0E0h,000h
'#'044h,0FEh,044h,0FEh,044h
'$'048h,054h,0FEh,054h,024h
'%'046h,026h,010h,0C8h,0C4h
'&'00Ah,044h,0AAh,092h,06Ch
'''000h,0C0h,0A0h,000h,000h
'('000h,082h,044h,038h,000h
')'000h,038h,044h,082h,000h
'*'028h,010h,07Ch,010h,028h
'+'010h,010h,07Ch,010h,010h
','000h,00Ch,00Ah,000h,000h
'-'010h,010h,010h,010h,010h
'.'000h,006h,006h,000h,000h

'/'040h,020h,010h,008h,004h
'0'07Ch,0A2h,092h,08Ah,07Ch
'1'000h,002h,0FEh,042h,000h
'2'062h,092h,08Ah,086h,042h
'3'05Ch,0A2h,0A2h,082h,044h
'4'008h,0FEh,048h,028h,018h
'5'09Ch,0A2h,0A2h,0A2h,0E4h
'6'04Ch,092h,092h,092h,07Ch
'7'0C0h,0A0h,090h,08Eh,080h
'8'06Ch,092h,092h,092h,06Ch
'9'07Ch,092h,092h,092h,064h
':'000h,06Ch,06Ch,000h,000h
';'000h,06Ch,06Ah,000h,000h
'<'082h,044h,028h,010h,000h
'='014h,014h,014h,014h,014h
'>'010h,028h,044h,082h,000h
'?'060h,090h,08Ah,080h,040h
'@'07Ch,082h,09Eh,092h,04Ch
'A'03Eh,048h,088h,048h,03Eh
'B'06Ch,092h,092h,092h,0FEh
'C'044h,082h,082h,082h,07Ch
'D'07Ch,082h,082h,082h,0FEh
'E'082h,082h,092h,092h,0FEh
'F'080h,090h,090h,090h,0FEh
'G'05Eh,092h,092h,082h,07Ch
'H'0FEh,010h,010h,010h,0FEh
'I'000h,082h,0FEh,082h,000h
'J'080h,0FCh,082h,002h,004h
'K'082h,044h,028h,010h,0FEh
'L'002h,002h,002h,002h,0FEh
'M'0FEh,040h,030h,040h,0FEh
'N'0FEh,008h,010h,020h,0FEh
'O'07Ch,082h,082h,082h,07Ch
'P'060h,090h,090h,090h,0FEh
'Q'07Ah,084h,08Ah,082h,07Ch
'R'062h,094h,098h,090h,0FEh
'S'04Ch,092h,092h,092h,064h
'T'080h,080h,0FEh,080h,080h
'U'0FCh,002h,002h,002h,0FCh
'V'0F8h,004h,002h,004h,0F8h
'W'0FCh,002h,01Ch,002h,0FCh
'X'0C6h,028h,010h,028h,0C6h
'Y'0E0h,010h,00Eh,010h,0E0h
'Z'0C2h,0A2h,092h,08Ah,086h
'['000h,082h,082h,0FEh,000h
'\'004h,008h,010h,020h,040h
']'000h,0FEh,082h,082h,000h
'^'020h,040h,080h,040h,020h
'_'002h,002h,002h,002h,002h
'`'000h,020h,040h,080h,000h
'a'01Eh,02Ah,02Ah,02Ah,004h
'b'01Ch,022h,022h,012h,0FEh
'c'004h,022h,022h,022h,01Ch
'd'0FEh,012h,022h,022h,01Ch
'e'018h,02Ah,02Ah,02Ah,01Ch
'f'040h,080h,090h,07Eh,010h
'g'07Ch,04Ah,04Ah,04Ah,030h
'h'01Eh,020h,020h,010h,0FEh

'i'000h,002h,0BEh,022h,000h
'j'000h,0BCh,022h,002h,004h
'k'000h,022h,014h,008h,0FEh
'l'000h,002h,0FEh,082h,000h
'm'01Eh,020h,018h,020h,03Eh
'n'01Eh,020h,020h,010h,03Eh
'o'01Ch,022h,022h,022h,01Ch
'p'020h,050h,050h,050h,07Eh
'q'07Eh,030h,050h,050h,020h
'r'010h,020h,020h,010h,03Eh
's'004h,02Ah,02Ah,02Ah,012h
't'004h,002h,012h,07Ch,010h
'u'03Eh,004h,002h,002h,03Ch
'v'038h,004h,002h,004h,038h
'w'03Ch,002h,00Ch,002h,03Ch
'x'022h,014h,008h,014h,022h
'y'03Ch,00Ah,00Ah,00Ah,030h
'z'022h,032h,02Ah,026h,022h
'{'082h,082h,06Ch,010h,000h
'|'000h,000h,0FEh,000h,000h
'}'000h,010h,06Ch,082h,082h
'~'080h,040h,0C0h,080h,040h

1. Se desea diseñar un display formado por 5 matrices de Leds de 7x5 que sea capáz de almacenar mensajes de 70 caracteres. El mensaje a desplegar debe ser introducido via serial o por medio de un teclado.

1. Diseñar un circuito que permita detectar la tecla oprimida en un teclado:

Cada vez que se oprime una tecla en in teclado IBM el teclado envía un código, este código es único para cada tecla. Por ejemplo si se oprime la tecla 'A' se enviará el código 1CH. Si se mantiene oprimida esta tecla el código se envía contínuamenta hasta que se suelte o se oprima otra tecla. Cuando se suelta una tecla el teclado envía el código F0H para indicar que se liberó una tecla y a continuación envía el código de la tecla liberada en nuestro ejemplo cuando se libera la tecla 'A' el teclado envía los códigos F0H 1CH.
El teclado siempre envía el mismo código para cada letra, por lo tanto se debe tener en cuenta que cada vez que se oprima la tecla 'SHIFT' la siguiente letra debe cambiarse de mayúsculas a minúsculas o viceversa. También se debe tener en cuenta el estado de la tecla 'CAPS'.

Host Commands

[Warning: Draw object ignored]

ED	**Set Status LED's - This command can be used to turn on and off the Num Lock, Caps Lock & Scroll Lock LED's. After Sending ED, keyboard will reply with ACK (FA) and wait for another byte which determines their Status. Bit 0 controls the Scroll Lock, Bit 1 the Num Lock and Bit 2 the Caps lock. Bits 3 to 7 are ignored.**
EE	Echo - Upon sending a Echo command to the Keyboard, the keyboard should reply with a Echo (EE)
F0	**Set Scan Code Set. Upon Sending F0, keyboard will reply with ACK (FA) and wait for another byte, 01-03 which determines the Scan Code Used. Sending 00 as the second byte will return the Scan Code Set currently in Use**

F3 Set Typematic Repeat Rate. Keyboard will Acknowledge command with FA and wait for second byte, which determines the Typematic Repeat Rate.

F4 **Keyboard Enable - Clears the keyboards output buffer, enables Keyboard Scanning and returns an Acknowledgment.**

F5 Keyboard Disable - Resets the keyboard, disables Keyboard Scanning and returns an Acknowledgment.

FE **Resend - Upon receipt of the resend command the keyboard will re- transmit the last byte sent.**

FF Reset - Resets the Keyboard.

Commands

[Warning: Draw object ignored]

Now if the Host Commands are send from the host to the keyboard, then the keyboard commands must be sent from the keyboard to host. If you think this way, you must be correct. Below details some of the commands which the keyboard can send.

FA **Acknowledge**

AA Power On Self Test Passed (BAT Completed)

EE **See Echo Command (Host Commands)**

FE Resend - Upon receipt of the resend command the Host should re-transmit the last byte sent.

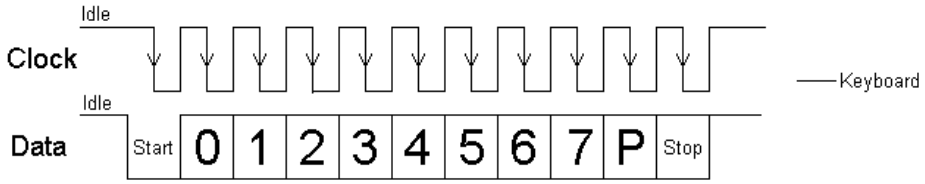
00 **Error or Buffer Overflow**

FF Error or Buffer Overflow

[Warning: Draw object ignored]

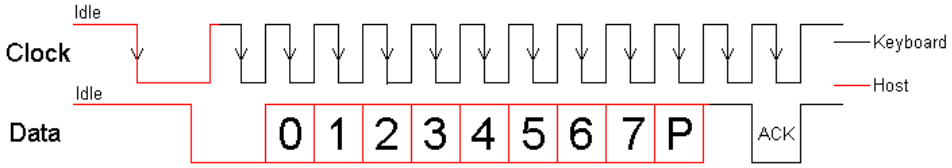
Sca

El teclado riene una interfaz PS2 la cual cuenta con la disposición de pines y el diagrama de tiempos indicados en la siguiente figura:



La figura anterior muestra la comunicación entre el teclado y el Host. Como puede verse se trata de una comuniacación serial sincrónica en la cual el teclado genera las señales Data y Clock. Solo durante el flanco de bajado se debe leer el estado de la señal de datos.

La comunicación entre el Host y el teclado se muestra en la siguiente figura. Como puede verse el Host debe iniciar la transmisión generando un flanco de bajada en la señal clock y un tiempo después generar el Bit de Start, después de esto el teclado genera la señal de reloj y el Host debe enviar serialmente el comando deseado, si el teclado recibe correctamente la trama responde con ACK.



El conector PS2 tiene 6 pines los cuales tiene las siguientes funciones:

PS/2

1. KBD Clock
2. GND
3. KBD Data
4. N/C
5. +5V (VCC)
6. N/C

Se desea que el driver de teclado tenga una salida ASCII, para poder ser utilizado con el Display inteligente del ejercicio 6. La siguiente tabla da una relación entre el código de la tecla y el código ASCII de la letra o carácter correspondiente.

NC		**F9**		**NC**		**F5**		**F3**		**F1**		**F2**		**F12**		**NC**		**F10**		**F8**		**F6**		**F4**		**Tab**			
00	FF	01	FF	02	FF	03	FF	04	FF	05	FF	06	FF	07	FF	08	FF	09	FF	0A	FF	0B	FF	0C	FF	OD	09	0E	7C
NC		**Alt**		**SHL**		**NC**		**CTL**		**q**		**l**		**NC**		**NC**		**NC**		**z**		**s**		**a**		**w**		**2**	
10	FF	11	FF	12	FF	13	FF	14	FF	15	71	16	31	17	FF	18	FF	19	FF	1A	7A	1B	73	1C	61	1D	77	1E	32
NC		**c**		**x**		**d**		**e**		**4**		**3**		**NC**		**NC**				**v**		**f**		**t**		**r**		**5**	
20		21		22		23		24		25		26		27		28		29		2A		2B		2C		2D		2E	
NC		**n**		**b**		**h**		**g**		**y**		**6**		**NC**		**NC**		**NC**		**m**		**j**		**u**		**7**		**8**	
30		31		32		33		34		35		36		37		38		39		3A		3B		3C		3D		3E	
NC		**,**		**k**		**i**		**o**		**0**		**9**		**NC**		**NC**		**·**		**-**		**l**		**ñ**		**p**		**'**	
40		41		42		43		44		45		46		47		48		49		4A		4B		4C		4D		4E	
NC		**NC**		**{**		**NC**		**´**		**¿**		**NC**		**NC**		**CAP**		**SHR**		**ENT**		**+**		**NC**		**}**		**NC**	
50		51		52		53		54		55		56		57		58		59		5A		5B		5C		5D		5E	
NC		**<**		**NC**		**NC**		**NC**		**NC**		**BKS**		**NC**		**NC**		**1**		**NC**		**4**		**7**		**NC**		**NC**	
60		61		62		63		64		65		66		67		68		69		6A		6B		6C		6D		6E	
0		**·**		**2**		**5**		**6**		**8**		**ESC**		**NUM**		**F11**		**+**		**3**		**-**		*****		**9**		**BD**	
70		71		72		73		74		75		76		77		78		79		7A		7B		7C		7D		7E	

;SHIFT + KEY

;00 NC 01 F9 02 NC 03 F5 04 F3 05 F1 06 F2 07 F12 08 NC 09 F10 0A F8 0B F6 0C F4 0D TAB OE ° 0F NC

db 0FFH, 04300H, 0FFH, 03F00H, 03D00H, 03B00H, 03C00H, 08600H, 0FFH, 04400H, 04200H, 04000H, 03E00H, 009H, 0A7H, 0FFH

;10 NC 11 ALT 12 SHL 13 NC 14 CTL 15 Q 16 ! 17 NC 18 NC 19 NC 1A Z 1B S 1C A 1D W 1E @ 1F NC

db 0FFH, 0FFH, 0FFH, 0FFH, 0FFH, 051H, 021H, 0FFH, 0FFH, 0FFH, 05AH, 053H, 041H, 057H, 040H, 0FFH

;20 NC 21 C 22 X 23 D 24 E 25 $ 26 # 27 NC 28 NC 29 SPC 2A V 2B F 2C T 2D R 2E % 2F NC

db 0FFH, 043H, 058H, 044H, 045H, 024H, 023H, 0FFH, 0FFH, 0FFH, 056H, 046H, 054H, 052H, 025H, 0FFH

;30 NC 31 N 32 B 33 H 34 G 35 Y 36 & 37 NC 38 NC 39 NC 3A M 3B J 3C U 3D / 3E (3F NC

db 0FFH, 0FFH, 042H, 048H, 047H, 059H, 026H, 0FFH, 0FFH, 0FFH, 04DH, 04AH, 055H, 02FH, 028H, 0FFH

;40 NC 41 ; 42 K 43 I 44 O 45 = 46) 47 NC 48 NC 49 : 4A _ 4B L 4C Ñ 4D P 4E ? 4F NC

db 0FFH, 03BH, 04BH, 049H, 04FH, 03DH, 029H, 0FFH, 0FFH, 03AH, 05FH, 04CH, 0A5H, 050H, 03FH, 0FFH

;50 NC 51 NC 52 [53 NC 54 ¨ 55 ¡ 56 NC 57 NC 58 CAP 59 SHR 5A ENT 5B * 5C NC 5D] 5E NC 5F NC

db 0FFH, 0FFH, 05BH, 0FFH, 0B0H, 0ADH, 0FFH, 0FFH, 0FFH, 0FFH, 013H, 02AH, 0FFH, 05DH, 0FFH, 0FFH

;60 NC 61 > 62 NC 63 NC 64 NC 65 NC 66 BKS 67 NC 68 NC 69 END 6A NC 6B <- 6C HM 6D NC 6E NC 6F NC

db 0FFH, 03EH, 0FFH, 0FFH, 0FFH, 0FFH, 08H, 0FFH, 0FFH, 0FFH, 0FFH, 0FFH, 0FFH, 0FFH, 0FFH, 0FFH

;70 0 71 DEL 72 AD 73 5 74 -> 75 AUP 76 ESC 77 NUM 78 F11 79 + 7A PDN 7B - 7C * 7D PUP 7E B D 7F NC

db 030H, 0FFH, 0FFH, 035H, 0FFH, 0FFH, 0FFH, 0FFH, 0FFH, 02BH, 0FFH, 02DH, 02AH, 0FFH, 0FFH, 0FFH

;83 F7

;E011 ALT GR E014 CTRR E069 END E06B LEFT E06C HOME E070 INS E071 DEL

;E072 DOWN E074 RIGHT E075 UP E07A P DN E07D P UP E05A INTRO

;E11477E1F014F077 PAUSE

;E012E07C PRINT SCREEN