República Bolivariana de Venezuela

Ministerio de la Defensa

Universidad Nacional Experimental

Politécnica de la Fuerza Armada Nacional

Núcleo Maracay

Cátedra de Sistemas Digitales I

ÍNDICE

INTRODUCCIÓN

Los diseñadores tienen una amplia gama de CIs disponibles con numerosas funciones lógicas y arreglos de circuitos sobre el mismo CI. Además, estos CIs son ofrecidos por varios fabricantes y con un costo razonablemente bajo. Por estas razones los diseñadores han interconectado CIs estándares para construir una variedad casi sin fin de circuitos y sistemas diferentes.

Sin embargo, existen algunos problemas con los diseños de circuitos y sistemas que utilizan sólo CIs estándares. Algunos diseños pueden requerir de cientos o miles de estos CIs. A su vez, este gran número de CIs requiere de una tarjeta de circuito impreso de tamaño considerable y de mucho tiempo y esfuerzo para colocar, soldar y probar todos los CIs.

Para reducir el número de CIs utilizados en un diseño, es necesario colocar cada vez más funciones lógicas sobre un CI. Esto, por supuesto, se hace con tecnologías LSI y VLSI para funciones estándar tales como las memorias, microprocesadores, sintetizadores de voz, calculadoras y así sucesivamente. Estos dispositivos tienen cientos y miles de compuertas lógicas interconectadas entre sí dentro del CI de forma tal que estos funcionan de una manera definida de antemano. Sin embargo, hay muchas situaciones de diseño para las que no existe ninguna solución LSI o VLSI. Por lo general, estas situaciones requieren que los diseñadores interconecten varios CIs LSI o MSI para obtener las funciones lógicas necesarias. El reciente desarrollo de los dispositivos lógicos programables (PLDs) ofrece a los diseñadores lógicos una manera de reemplazar varios CIs estándares con un solo CI.

Un PLD es un CI que contiene un número muy grande de compuertas, FFs y registros, que están conectados entre sí dentro del CI; sin embargo, muchas de las conexiones son del tipo fusible. Se dice que el CI es programable porque la función específica que éste realice en una determinada aplicación está determinada por la interrupción selectiva de algunas de las conexiones, mientras que al mismo tiempo se dejan otras intactas.

El proceso de fundir las conexiones puede realizarlo el fabricante de acuerdo con las instrucciones del cliente o bien puede hacerlo este último. El proceso recibe el nombre de programación porque produce el patrón de interconexión de circuito deseado entre las compuertas, FFs, registros, etc.

Al igual que una ROM o PROM, un PLD se programa al especificar un patrón de diodos o de fisibles. Sin embargo, a pocos diseñadores les gusta cortar los fisibles de forma directa y ni siquiera en modo indirecto con un archivo de texto hexadecimal. En su lugar, la mayoría de los diseñadores usan un lenguaje de programación PLD para especificar simbólicamente las funciones lógicas.

Existen diversos paquetes software para implementar los diseños lógicos basados en PLDs. ABEL y CUPL son dos de los lenguajes de descripción de hardware (HDL, Hardware Description Language) más comúnmente utilizados; puesto que ambos son similares y producen el mismo resultado en términos de programación de un PLD, a menudo, su utilización es una cuestión de preferencias y disponibilidad.

El objetivo del presente trabajo es explicar el Lenguaje de Programación de PLD estándar industrial llamado ABEL; para lo cual se indicará su estructura interna, programación, ventajas y desventajas sobre las alambradas, y su aplicación con circuitos combinacionales.

DEFINICIÓN DE LAS SIGLAS ABEL

ABEL es una marca registrada de Data I/O Corporation y es el acrónimo de Advanced Boolean Expression Language, permite implementar diseños lógicos en dispositivos lógicos programables. Puede ser utilizado para programar cualquier tipo de PLD y, por tanto, es un lenguaje independiente del dispositivo. El lenguaje ABEL se ejecuta en un computador conectado a un programador de dispositivos, independiente del lenguaje, en el que se inserta el PLD.

Un lenguaje de programación PLD está respaldado por un procesador de lenguaje PLD denominado compilador. La tarea del compilador es traducir un archivo de texto escrito en el lenguaje en un patrón de fusibles para el PLD físico. Aún cuando la mayoría de los PLD pueden programarse físicamente sólo con expresiones de suma de productos, lenguajes como ABEL permiten que las ecuaciones PLD se escriban casi en cualquier formato; el compilador manipula algebraicamente y minimiza las ecuaciones para ajuste, si es posible, dentro de la estructura PLD disponible.

ESTRUCTURA INTERNA

A continuación se presenta la estructura típica de un programa en el lenguaje ABEL y un ejemplo del mismo:

Estructura:

module *nombre del módulo*

[**title** *string*]

[deviceID **device** deviceType;]

declaraciones de pin

otras declaraciones

equations

ecuaciones

[**Test_Vectors**]

tested vectores

end *nombre de módulo*

En la estructura se identifican las siguientes características:

- Un archivo de programa comienza con el enunciado ***module***, que asocia un nombre (por ejemplo: Decodificador_de_Memoria) con el módulo del programa. Los programas grandes pueden tener múltiples módulos, cada uno con su propio título local, declaraciones y ecuaciones. El nombre del módulo puede ser cualquier identificador válido.

Los identificadores deben comenzar con una letra o un guión, pueden contener hasta 31 letras, dígitos y guiones, y son distinguibles las minúsculas y mayúsculas.

- El enunciado ***title*** especifica una cadena como un título que se insertará en los archivos de documentación que sean creados por el compilador.

Una cadena es una serie de caracteres encerrados entre comillas simples.

- La declaración ***device*** incluye un identificador de dispositivo (por ejemplo: `P16L8' para un PAL16L8). El compilador usa el identificador del dispositivo en los nombres de los archivos de documentación que genera, y usa el tipo de dispositivo para determinar si éste puede en realidad realizar las funciones lógicas requeridas en el programa.
- La directiva *@ALTERNATE* le dice al compilador que reconozca un conjunto alterno de símbolos para denotar las operaciones lógicas; los cuales se presentan a continuación:

Operación Lógica

Símbolo ABEL

Notación Booleana

Notación ABEL

* AND

&

AB

A & B

+ OR

#

A + B

A # B

/ NOT

!

!A

:+: XOR

$

A " B

A $ B

:*: XNOR

!$

A " B

A !$ B

Por ejemplo, la siguiente expresión lógica:

Y= (+ B +   + D) (A + B + E)

en lenguaje ABEL sería:

Y= (!A # B # !E # D) & (A # B # E)

- Los comentarios comienzan con una doble comilla y terminan con otra doble comilla o el fin de la línea, lo que ocurra primero.
- Las declaraciones de terminales le indican al compilador los nombres simbólicos asociados con las terminales externas del dispositivo. Las señales cuyos nombres tienen el prefijo NOT (/) son activas bajas en la terminal externa; las otras son activas altas.
- Otras declaraciones permiten al diseñador definir constantes y macros para simplificar el diseño del programa y mejorar su legibilidad.
- El enunciado ***equations*** indican qué ecuaciones lógicas definen señales de salida como función de las señales de entrada que correspondan.
- El enunciado ***end*** marca el fin del módulo.

PROGRAMACIÓN EN EL LENGUAJE ABEL

ABEL proporciona tres (3) diferentes formatos para describir e introducir un diseño lógico desde el teclado de un computador; los cuales corresponden a: ecuaciones, tablas de verdad y diagramas de estado. Las ecuaciones y las tablas de verdad se usan en los diseños lógicos combinacionales; mientras que los diagramas de estado, así como los otros dos formatos, se pueden utilizar para el diseño de lógica secuencial (máquinas de estado).

Una vez que se ha introducido el diseño lógico, se puede simular su funcionamiento utilizando vectores de

prueba para asegurarse que no existan errores en el diseño. Básicamente, una sección de vectores de prueba enumera todos los posibles valores de entrada y los correspondientes valores de salida. El software, esencialmente, ejercita el diseño lógico para asegurarse de que trabaja como se esperaba, tomando todas las posibles combinaciones de entrada y comprobando los resultados de salida.

El proceso software que convierte la descripción del circuito en formato de ecuaciones, tablas de verdad o diagramas de estado, en el fichero JEDEC estándar requerido para programar un PLD se denomina síntesis lógica. Usualmente, la síntesis lógica incluye la optimización y la minimización del diseño.

Números

Los números pueden estar en cuatro diferentes bases: binario, octal, decimal y hexadecimal. La base predeterminada es la decimal. Se usan los siguientes símbolos (mayúsculas o minúsculas permitidos) para especificar la base. Cuando no se usan símbolos se asume que se usa la base decimal. Se puede cambiar la base con la directiva Radix como se explicará en el próxima sección:

BASE NAME	**BASE**	**Symbol.**
Binary	2	^b
Octal	8	^o
Decimal	10	^d (default)
Hexadecimal	16	^h

Ejemplo:

Specified in ABEL	**Decimal Value**
35	35
^h35	53
^b101	5

Directivas

Las directivas permiten la manipulación avanzada del archivo fuente y del procesado, y puede ser colocado en cualquier parte donde se necesite en el archivo.

@ALTERNATE
Syntax
@alternate

@ALTERNATE habilita alternar sets de operadores. Usando el set operador alternado se evita el uso del operadores ABEL−HDL suma (+), multiplicación (*) y división (/) porque ellos representan lo operadores lógicos OR, AND y NOT en el set alternado. Los operadores estándar permanecen trabajando cuando @ALTERNATE esta en efecto. Los operadores alternados permanecen en efecto hasta que la directiva @STANDARD es usada o el final del módulo es alcanzado.

@RADIX
Syntax
@radix *expr* ;

Expr: Una expresión válida que produce los números 2, 8 10 y 16 para indicar la predeterminación de una nueva base.

La directiva @Radix cambia la base predeterminada. La base predeterminada es 10 (decimal). La nueva base especificada como predeterminada permanece en efecto hasta que otra directiva @radix es colocada o hasta que el final del módulo es alcanzado. Notese que cuando un nuevo @radixes colocado, la especificación de la nueva base debe estar en el formato de la actual base.

Ejemplo

@radix 2; cambia la base predefinida a binario

@radix 1010; devuelve el cambio de binario a decimal.

@STANDARD
Syntax
@standard

La opción @standard hace un reset en los operadores estandar ABEL−HDL. Los set alternados son elegidos con la directiva @alternative.

Sets

Un set es una colección de señales o constantes usadas para referenciar un grupo de señales por un nombre. Un set es conveniente para simplificar expresiones lógicas. Cualquier operación que este aplicada a un set, es aplicada a cada elemento.

Una lista de constantes o señales set separadas por comas o por el operador de rango (..) se coloca entre corchetes (requerido).

Ejemplos:

[D0,D1,D2,D4,D5]	
[D0..D6]	" rango incrementado
[b6..b0]	" rango decrementado
[D7..D15]	
[b1,b2,a0..a3]	" rango entre un set más largo
[!S7..!S0]	"rango decrementado de señales activas−bajas

Sin embargo, la siguiente no es permitido: [D0, X];

Donde la X es tambien un set X = [X3..X0];

En vez de eso, se puede escribir:

[D0, X3..X0];

a. Indexando o accesando a un set

El indexado permite acceder a elementos dentro de un set. Usando valores numéricos para indicar el set index. Los números refieren a la posición del bit en el set de inicio con 0 para el bit menos significativo del set. Aquí se presentan algunos ejemplos.

D1 = [D15..D0]; "declaración set

X2 = [X3..X0]; "declaración set

X2 := D1[3..0]; "hace X2 igual a [D3, D2, D1, D0]

X2 := D1[7..4]; "hace X2 igual a [D7, D6, D5, D4]

Para acceder a un elemento del set, se usa la siguiente sintaxis:

OUT = (X[2] == 1);

Un operador comparador (==) es usado para convertir un elemento (X[2]) en un valor de bit equivalente a X2. El comparador (==) arroja un 1! O 0 dependiendo si la comparación es Verdadera o Falsa. Véase la *diferencia entre el operador de asignación (=) y el operador de igual (==).* El operador de asignación es usado en ecuaciones en vez de expresiones. Las ecuaciones asignan un valor de una expresión a la señales de salida.

b. Operaciones Set

Muchas de las operaciones pueden ser aplicadas a un set y son ejecutadas en cada elemento del set de acuerdo a las reglas del álgebra Booleana. Las operaciones son ejecutadas de acuerdo a las prioridades de operación; los operadores con la misma prioridad son ejecutados de izquierda a derecha (al menos que use paréntesis). Aquí se muestran un par de ejemplos:

Ejemplo 1:

Signal = [D2,D1,D0]; "declaración de Señal set

Signal = [1,0,1] & [0,1,1];" resultados en Señal siendo "igual a [0,0,1]

Ejemplo 2:

[A,B] = C & D;

es equivalente a dos declaraciones:

A = C & D;

B = C & D;

Ejemplo 3:

[A1,B1] = [D1,D2] & [C3,C2];

es equivalente a: [A1,B1] = [D1 & C3, D2 & C2];

así A1 = D1 & C3, y B1= D2 & C2.

Ejemplo 4:

X & [A,B,C];

El cual es equivalente a

[X&A, X&B, X&C];

Sin embargo considerar la siguiente expresión

2 & [A,B,C];

ahora el número 2 es convertido primero en una representación binaria y completado con ceros (0010) si es necesario. De esta manera la siguiente ecuación es equivalente a:

[0 & A, 1 & B, 0 & C];

Ejemplo 5:

A=[A2,A1,A0]; "declaración set

B=[B2,B1,B0]; " declaración set

A # B; es equivalente a [A2 # B2, A1 # B1, A0 # B0];

!A; es equivalente a [!A2,!A1,!A0];

Ejemplo 6:

[b3,b2,b1,b0] = 2;"es equivalente a b3=0,b2=0,b1=1,b0=0.

El número 2 es convertido a binario y completado con ceros (0010).

Ejemplo 7: Los Sets son también convenientes para especificar ecuaciones lógicas. Supóngase que se necesite especificar la ecuación:

Chip_Sel = !A7 & A6 & A5;

Esto puede ser hecho usando sets. Primero se define una constante Addr set:

Addr = [A7,A6,A5];" declares a constant set Addr.

Se puede entonces usar la siguiente ecuación para especificar la dirección:

Chip_Sel = Addr == [0,1,1];

El cual es equivalente a decir:

Chip_Sel = !A7 & A6 & A5;

De manera que: si A7=0, A6=1 y A5=1, la expresión Addr ==[0,1,1] es verdadera (ó 1) y así Chip_Sel será veradero (ó 1). Otra manera de escribir la misma ecuación es:

Chip_Sel = Addr == 3; " decimal 3 is equal to 011.

La siguiente expresión es de mucha ayuda con se trabaja con gran cantidad de variables (e.j: una dirección de

16bit)

Ejemplo 8:

Para la misma constante del ejemplo siguiente, la siguiente,

3 & Addr;

el cual equivale a

[0,1,1] & [A7,A6,A5]

[0 & A7, 1 & A6, 1 & A5]

[0,A6,A5].

Sin embargo. La siguiente declaración es diferente:

3 & (Addr == 1);

el cual es equivalente a:

3 & (!A7 & !A6 & A5).

Sin embargo, el operador relacional (==) arroja un solo bit, así que el resto de la ecuación evalúa también a un bit, y el 3 es truncado a 1. De esta manera la siguiente ecuación es igual a:

1& !A7 & !A6 & A5.

Operadores

Hay cuatro tipos de operadores: lógico, aritmético, relacional y de asignación.

a. Operadores Lógicos

La tabla siguiente muestra operadores lógicos. Estos son ejecutados bit a bit. Con la directiva @ALTERNATIVE, se puede usar el set alternativo de operadores como se indica en la tabla.

Operador (default)	**Descripción**	**Operador alternado**
!	NOT (complemento a uno)	/
&	AND	*
#	OR	+
$	XOR: or exclusiva	:+:
!$	XNOR: nor exclusiva	:*:

b. Operadores Aritméticos

La siguiente tabla muestra los operadores aritméticos. Nótese que los últimos cuatro operadores no están permitidos para los sets. El signo menos puede tener un significado diferente: usado entre dos operandos indica substracción (o suma a complemento a dos), mientras que usado con un operador indica el complemento a dos.

Operador	Ejemplo	Descripción
−	−D1	Complemento a dos (negación)
−	C1−C2	Substracción
+	A+B	Adición
Los siguientes operadores no son usados con los sets:		
*	A*B	Multiplicación
/	A/B	División de enteros sin signo
%	A%B	Módulos: residuo de A/B
<<	A<<B	Intercala A a la izquierda por B bits
>>	A>>B	Intercala A a la derecha por B bits

c. Operadores de Relación

Estos operadores producen un valor Booleano de verdadero (−1) o falso (0). El valor verdadero lógico −1 en complemento a dos esta representado por todos los unos (todos los bits serán uno ej: para una palabra de 16 bits todos los bits son uno: −1 está representado por 1111 1111 1111 1111).

Operador	Ejemplo	Descripción
==	A==B or 3==5 (false)	Igual
!=	A!=B or 3 != 5 (true)	Diferente
<	A<B or 3 < 5 (true)	Menor que
<=	A<=B or 3 <= 5 (true)	Menor o igual que
>	A>B or −1 > 5 (true)	Mayor que
>=	A>=B or !0 >= 5 (true)	Mayor o igual que

Los operadores de relación no tienen signo. Cuidado: ¡0 es el complemento a uno de 0 o 11111111 (8 bit data) el cual es 255 en binario sin signo. Además !0>9 es verdadero. La expresión −1>5 es verdadera por la misma razón.

Una expresión de relación se puede usar donde se sea necesaria por un numero. El −1 o 0 serán sustituidos dependiendo del resultado lógico. Por ejemplo:

A = B !$ (C == D);

A será igual a B si C es igual a D (verdadero o 11111; B XNOR 1 igual B), sino, A será el complemento de B (si C no es diferente a B (falso o =)).

d. Operadores de asignación

Estos operadores son usados en una ecuación para asignar el valor de una expresión a una señal de salida. Hay dos tipos de operadores de asignación: combinacionales y de registro. En un operador de asignación combinacional ocurre inmediatamente sin retardo. La asignación con registro ocurre con el cambio del pulso de reloj asociado a la salida. Por ejemplo, se puede definir un flip−flop con la siguiente declaración:

Q1 pin istype 'reg';

Q1 := D;

La primera declaración define el Q1 del flip−flop para usar el `reg' como istype (salida de registro). La segunda declaración dice que la salida del flip−flop tomará el valor D de la entrada en la próxima transición de reloj.

Operador	**Descripción**
=	Asignación Combinacional
:=	Asignación de Registro

e. Operador de prioridad

La prioridad de cada operador esta dada en la siguiente tabla, con prioridad 1 tiene mayor prioridad y con 4 tiene la menor prioridad. Los operadores con la misma prioridad son ejecutados de izquierda a derecha.

Prioridad	**Operador**	**Descripción**
1	−	Negación (complemento a 2)
1	!	NOT
2	&	AND
2	<<	shift left
2	>>	shift right
2	*	Multiplicar
2	/	división sin signo
2	%	módulo
3	+	Suma
3	−	Resta
3	#	OR
3	$	XOR
3	!$	XNOR
4	==	Igual
4	!=	Diferente
4	<	Menor que
4	<=	Menor o igual que
4	>	Mayor que
4	>=	Mayor o igual que

Descripción Lógica

Un diseño lógico puede ser descrito de la siguiente manera.

- Ecuaciones
- Tablas de la verdad
- Descripción de estados.

a. Ecuaciones

Usar el comando **equations** para empezar la descripción lógica. Equations especifica expresiones lógicas usando los operadores descritos antes, o por declaraciones "When−Then−Else".

Las declaraciones "When−Then−Else" son usadas en ecuaciones para describir funciones lógicas. (Nota: "If −Then−Else" es usado en la sección de los diagramas de estado para describir progresión de estados)

El formato de declaración "When−Then−Else" es el siguiente:

WHEN condition THEN element=expression;

ELSE equation;

or

WHEN condition THEN equation;

Ejemplos de ecuaciones:

SUM = (A & !B) # (!A & B) ;
A0 := EN & !D1 & D3 & !D7;

WHEN (A == B) THEN D1_out = A1;

ELSE WHEN (A == C) THEN D1_out = A0;

WHEN (A>B) THEN { X1 :=D1; X2 :=D2; }

Se usan llaves {} para agrupar secciones en bloques. El texto en bloque puede ser usado en una línea o en muchas líneas. Los bloques son usados en ecuaciones, diagramas de estados y directivas.

b. Tablas de verdad

El commando es **truth−table** y la sintaxis es:

TRUTH_TABLE (in_ids −> out_ids)

inputs −> outputs ;

o

TRUTH_TABLE (in_ids :> reg_ids)

inputs :> reg_outs ;

o

TRUTH_TABLE

(in_ids :> reg_ids −> out_ids)

inputs :> reg_outs −> outputs ;

en donde "−>" es para salidas combinacionales y :> para salidas registradas. La primera línea de la tabla de verdad (entre paréntesis) define las señales de entradas y salida. Cada línea debe finalizar con punto y coma.

Las entradas y salidas pueden ser simple señales o set's. Cuando se usan set's como entradas o salidas, se usa la notación del set normal, ej señales dentro de corchetes y separadas por comandos. Las condiciones irrelevantes don't care se representan con X.

Ejemplo 1: Medio sumador (half adder)

TRUTH_TABLE ([A, B] −> [Sum, Carry_out])

[0, 0] −> [0, 0] ;
[0, 1] −> [1, 0] ;
[1, 0] −> [1, 0] ;
[1, 1] −> [1, 1] ;

Sin embargo, si se define un set IN = [A,B]; y OUT = [Sum, Carry_out]; la tabla de verdad se hace más simple:

TRUTH_TABLE (IN −> OUT)

0 −> 0;
1 −> 2;
2 −> 2;
3 −> 3;

Ejemplo 2: Una OR exclusiva de dos entradas y un habilitador (EN). Este ejemplo ilustra el uso de las condiciones irrelevantes X

TRUTH_TABLE ([EN, A, B] −> OUT)

[0, .X.,.X.] −> .X. ;
[1, 0 , 0] −> 0 ;
[1, 0 , 1] −> 1 ;
[1, 1 , 0] −> 1 ;
[1, 1 , 1] −> 0 ;

Tablas de verdad pueden ser usadas para definir máquinas secuenciales. Ej: implementando un contador ascendente de 3−bit de 000 a 111 y regresando a 0. Llamemos QA, QB y QC las salidas de los flip−flops. Además, generaremos una salida OUP siempre que el contador alcance el estado 111. También se hará un reset al contador al estado 000 cuando la señal de reset sea alta.

MODULE CNT3;

CLOCK pin; " input signal
RESET . pin; " input signal
OUT pin istype 'com'; " output signal (combinational)
QC,QB,QA pin istype 'reg'; " output signal (registered)

[QC,QB,QA].CLK = CLOCK; "FF clocked on the CLOCK input
[QC,QB,QA].AR = RESET; "asynchronous reset by RESET

TRUTH_TABLE) [QC, QB, QA] :> [QC,QB,QA] −> OUT)

[0 0 0] :> [0 0 1] −> 0;

[0 0 1] :> [0 1 0] −> 0;
[0 1 0] :> [0 1 1] −> 0;
[0 1 1] :> [1 0 0] −> 0;
[1 0 0] :> [1 0 1] −> 0;
[1 0 1] :> [1 1 0] −> 0;
[1 1 0] :> [1 1 1] −> 0;
[1 1 1] :> [0 0 0] −> 1;

END CNT3;

c. Descripción de estado

La sección de diagrama de estado contiene la descrición de estado para el diseño lógico. Esta sección usa la sintaxis State_diagram y las declaraciones "If−Then−Else", "Goto", "Case" y "With". Usualmente se declara nombres de estado simbólicos en la sección de Declaración, el cual hace que la lectura se más fácil.

Sintaxis de *Declaración de estado:*

state_id [, state_id ...] STATE ;

Como ejemplo: SREG = [Q1, Q2]; asocial el nombre del estado SREG con el estado definido por Q1 y Q2.

La sintaxis para *State_diagram* es la siguiente:

State_diagram *state_reg*

STATE *state_value* : [equation;]

[equation;]

:

:

trans_stmt ; ...

El comando **state_diagram** indica la descripción del inicio del estado de la máquina.

El comando STATE y las siguientes declaraciones describen un estado de los diagramas de estado e incluyen un valor de inicio o un nombre simbólico de estado, una declaración de transición de estado y una salida de ecuación opcional. Con la siguiente sintaxis,

- state_reg: es un identificador que define las señales que determinan el estado de la máquina. Este puede ser un registro de estado simbólico.
- state_value: puede ser una expresión, un valor o un nombre de estado simbólico de el estado actual.
- equation : una ecuación que define los estados de las salidas de la máquina.
- trans_stmt: las declaraciones "If−Then−Else", CASE o GOTO definen el siguiente estado, seguido de un WITH de ecuaciones de transición opcional.

Declaración **If−Then−Else**:

Esta declaración es usada en la sección state_diagram para describir el próximo estado y especificar

mutualmente las condiciones de transiciones exclusivas.

Sintaxis:

IF expression THEN state_exp

[ELSE state_exp] ;

Donde state−exp puede ser una expresión lógica o un nombre simbólico de estado. Nótese que la declaración "IF−Then−Else" puede ser usada solamente en la sección state_diagram (use los "When−If−Then" para describir funciones lógicas). La cláusula ELSE es opcional. Las declaraciones IF−Then−Else pueden estar seguidas con las declaraciones Goto, Case y With.

Ejemplo:

En la sección de declaración se define primero el estado del registro:

```
SREG = [Q1, Q0]; "definition of state registers
S0 = [0, 0];
S1 = [1, 1];

state_diagram SREG
state S0: OUT1 = 1;

if A then S1
else S0;

state S1: OUT2 =1;

if A then S0
else S1;
```

La declaración "If−Then−Else" puede ser conectada de la siguiente manera. Se asume que se ha definido el registro y los estados en la sección de declaraciones.

```
state_diagram MAK

state INIT: if RESET then INIT else LOOK;

state LOOK: if REST than INIT

else if (X == LASTX) then OK
else LOOK;

state OK: if RESET than INIT

else if Y then OK
else if (X == LASTX) then OK
else LOOK;

state OK: goto INIT;
```

Declaración "**with**":

Sintaxis:

trans_stmt state_exp WITH equation
[equation] ... ;

donde trans_stmt puede ser una declaración "If−then−else", 'Goto" o "Case".
state_exp: es el estado siguiente, y equation es una ecuación para las salidas de la máquina.

Esta declaración puede ser usada con las declaraciones "If−Then−Else", "Goto" o "Case" en lugar de una simple expresión de estado. La declaración "With" permite a las ecuaciones de salida escribirse en términos de transición.

Ejemplo 1:

if X#Y==1 then S1 with Z=1 else S2;

En este ejemplo, la salida Z sera activada tan pronto la expresión después de la declaración If evalúa un 1 lógico (o TRUE). La expresión después del comando "With" puede ser una ecuación que será evaluada tan pronto como la condición If es verdadera como en el ejemplo 2:

Ejemplo 2:

if X&!Y then S3 with Z=X#Y else S2 with Z=Y;

La declaración "With" también sirve para describir el comportamiento de salida de una salida registrada, desde que la salida registrada sea retrasada un ciclo del reloj. Esto permite por un instante especificar que la salida registrada debería tener un valor específico después de una transición en particular. Como en el ejemplo [1],

Ejemplo 3[1]:

state S1:

if RST then S2 with { OUT1 := 1;

Error−Adrs := ADDRESS; }

else if (ADDRESS <= ^hC101)

then S4
else S1;

Nótese que se pueden usar paréntesis para controlar un grupo de salidas y ecuaciones después del comando With como en el ejemplo siguiente.

Ejemplo 3:

state S1: if (A & B) then S2 with TK = 1

else S0 with TK = 0 ;

Se debe tener cuidado del temporizado cuando se use la declaración "With " con salidas combinacionales o asincrónicas

Declaración **Case** :

Sintaxis:

CASE expression : state_exp;

[expression : state_exp;]
:

ENDCASE ;

expresión es cualquier expresión ABEL válida y state_exp es una expresión que indica el próximo estado (opcionalmente seguida por una declaración WITH).

Ejemplo:

State S0:

case (A == 0) : S1;

(A == 1) : S0;

endcase;

La declaración case es usada para enlistar una secuencia de condiciones de transiciones mutualmente−exclusiva y correspondiente al estado siguiente. La declaración CASE debe ser mutualmente exclusiva (no dos condiciones de transición pueden estar al mismo tiempo) o el resultado del estado siguiente es impredecible.

d. Extensiones punto (Dot extensions)

Se usan condiciones punto para describir de manera más precisa el comportamiento del circuito. Las extensiones de las señales son muy apropiadas y proveen un punto para referir específicamente señales internas y nodos asociados con la señal primaria.

La sintaxis es

signal_name.ext

Algunas de las extensiones punto son dadas en la tabla siguiente. Las extensiones no son de argumento sensible. Algunas extensiones punto son de propósito general (también llamadas de arquitectura independiente o pin−a−pin) y pueden ser usadas con variedad de arquitecturas de dispositivos. Otras extensiones punto son usadas para especificar clases de arquitecturas de dispositivos o extensiones punto detalladas. En general, se puede usar cualquier extensión punto.

Extensión Punto	**Descripción**
Arquitectura independiente o extensión pin−a−pin	
.ACLR	Reset de registro asincrónico

.ASET	Preset de registro asincrónico
.CLK	Entrada de reloj en flip−flop disparado por flanco
.CLR	Reset de registro sincrónico
.COM	Retroalimentación combinacional de un flip−flop tipo D
.FG	Retroalimentación de registro
.OE	Habilitador de salida
.PIN	Pin de retroalimentación
.SET	Preset de registro sincrónico
Extensiones específicas de dispositivos (dependiente de arquitectura)	
.D	Entrada de dato a un Flip flop D
.J	Entrada J a un JK flip−flop
.K	Entrada K a un JK flip−flop
.S	Entrada S a un SR flip−flop
.R	Entrada R a un SR flip−flop
.T	Entrada T a un T flip−flop
.Q	Retroalimentación de registro
.PR	Preset de registro
.RE	Reset de registro
.AP	Preset de registro asincrónico
.AR	Reset de registro asincrónico
.SP	Preset de registro sincrónico
.SR	Reset de registro sincrónico

Ejemplo 1:

[S6..S0].OE = ACTIVE;

donde se activa el control de señal tres estados de la señal de salida del buffer S6...S0. Cuando ACTIVE es alto, las señales serán habilitadas, de lo contrario una señal alta es generada en la salida Z.

Ejemplo 2:

Q.AR = reset;

[Z.ar, Q.ar] = reset;

donde se hace un reset en las salidas del registro (flip flops) a cero cuando el reset es alto.

Vectores de prueba

Los vectores de prueba son opcionales y proveen un punto para verificar la operación correcta de una máquina de estado. Los vectores especifican la operación lógica esperada de un dispositivo lógico por explícitamente dando a las salidas funciones de las entradas.

Sintaxis:
Test_vectors [note]

(input [, input].. −> output [, output] ..)

[invalues −> outvalues ;]
:
:

Ejemplo:

Test_vectors

([A, B] −> [Sum, Carry])

[0, 0] −> [0, 0];
[0, 1] −> [1, 0];
[1, 0] −> [1, 0];
[1, 1] −> [1, 1];

También se puede especificar los valores para el set con constantes numéricas como se muestra abajo:

Test_vectors

([A, B] −> [Sum, Carry])

0 −> 0;
1 −> 2;
2 −> 2;
3 −> 3;

Condiciones irrelevantes (.X.), entradas de reloj (.C.) tantas constantes simbólicas sean permitidas, como se muestra en el ejemplo siguiente:

test_vectors

([CLK, RESET, A, B] −> [Y0, Y1, Y3])

[.X., 1, .X.,.X.]−>[S0, 0, 0];
[.C., 0, 0, 1] −> [S0, 0, 0];
[.C., 1, 1, 0] −> [S0, 0, 1];

Propiedades de las declaraciones

ABEL permite dar declaraciones específicas a los dispositivos usando las propiedades de las declaraciones. Esta declaración será pasada por un programa filtro durante la compilación. Para dispositivos CPLD estas propiedades incluyen

- Taza de transferencia (Slew rates)
- Optimizaciones lógicas
- Ubicación lógica
- Configuraciones de Potencia
- Valores precargados

Misceláneos

a. Declaraciones Activas−bajas

señales activas bajas son definidas con el operador "!", como se muestra,

!OUT pin istype 'com' ;

Cuando estas señales son usadas en una descripción de un diseño subsiguiente, será automáticamente completada. Considérese la descripción del ejemplo siguiente,

module EXAMPLE

A, B pin ;

!OUT pin istype 'com';

equations

OUT = A & !B # !A & B ;

end

En este ejemplo, la señal OUT es una XOR de A y B, ej; OUT será 1 (Alta o On) cuando solo una de las entradas es 1, de lo contrario OUT es 0. Sin embargo, el pin de salida es definido como !OUT , ej; en una señal activa baja, lo que significa que el pin irá a 0 (Activa−baja ON) cuando una de las dos entradas sean 1. Se podría obtener el mismo resultado invirtiendo la señal en las ecuaciones y declarándola en la salida del pin que será OUT, como se muestra en el siguiente ejemplo. Esto es llamado *activación−baja pin−apin explicita* (porque usa señales activas bajas en la ecuación).

module EXAMPLE

A, B pin ;

OUT pin istype 'com';

equations

!OUT = A & !B # !A & B ;

end

Activas bajas pueden especificarse para un set también. En el ejemplo se define a sets A, B y C.

A = [A2,A1,A0]; "declaración set
B = [B2,B1.B0]; "declaración set
X = [X2,X1.X0]; "declaración set

!X = A & !B # !A & B;

La última ecuación es equivalente a escribir

!X0 = A0 & !B0 # !A0 & B0;
!X1 = A1 & !B1 # !A1 & B1;

!X2 = A2 & !B2 # !A2 & B2;

APLICACIONES CON CIRCUITOS COMBINACIONALES

BCD Suma 6 con ecuaciones reducidas

MODULE bcdadd6a

TITLE 'Lab1 Demo − Reduced Equations

Todd Morton, 10/19/00'

" Pin Declarations

Bi0,Bi1,Bi2,Bi3 pin;

Bo0,Bo1,Bo2,Bo3 pin istype 'com';

Equations

Bo0 = Bi0;

Bo1 = (!Bi3 & !Bi2 & !Bi1) # (Bi2 & Bi1);

Bo2 = !Bi2 & !Bi1;

Bo3 = !Bi2 & Bi1;

Test_vectors

([Bi3..Bi0] −> [Bo3..Bo0])

[0,0,0,0] −> [0,1,1,0];

[0,0,0,1] −> [0,1,1,1];

[0,0,1,0] −> [1,0,0,0];

[0,0,1,1] −> [1,0,0,1];

[0,1,0,0] −> [0,0,0,0];

[0,1,0,1] −> [0,0,0,1];

[0,1,1,0] −> [0,0,1,0];

[0,1,1,1] −> [0,0,1,1];

[1,0,0,0] −> [0,1,0,0];

[1,0,0,1] −> [0,1,0,1];

END

BCD Suma 6 con Tabla de Verdad sin Condiciones Irrelevantes

MODULE bcdadd6b

TITLE 'Lab1 Demo − Using ABEL Constructs

Todd Morton, 10/19/00'

" Pin Declarations

Bi0,Bi1,Bi2,Bi3 pin;

Bo0,Bo1,Bo2,Bo3 pin istype 'com';

"Sets

Bi = [Bi3..Bi0];

Bo = [Bo3..Bo0];

Equations

```
when (Bi <= 3) then Bo = Bi + 6;

when (Bi > 3) then Bo = Bi − 4;


Test_vectors


([Bi3..Bi0] −> [Bo3..Bo0])

[0,0,0,0] −> [0,1,1,0];

[0,0,0,1] −> [0,1,1,1];

[0,0,1,0] −> [1,0,0,0];

[0,0,1,1] −> [1,0,0,1];

[0,1,0,0] −> [0,0,0,0];

[0,1,0,1] −> [0,0,0,1];

[0,1,1,0] −> [0,0,1,0];

[0,1,1,1] −> [0,0,1,1];

[1,0,0,0] −> [0,1,0,0];

[1,0,0,1] −> [0,1,0,1];


END
```

BCD Suma 6 con Tabla de Verdad con Condiciones Irrelevantes

```
MODULE bcdadd6c


TITLE 'Lab1 Tutorial − Using Truth Table, no don't cares

Todd Morton, 10/19/00'
```

" Pin Declarations

Bi0,Bi1,Bi2,Bi3 pin;

Bo0,Bo1,Bo2,Bo3 pin istype 'com';

Truth_Table ([Bi3..Bi0] −> [Bo3..Bo0])

[0,0,0,0] −> [0,1,1,0];

[0,0,0,1] −> [0,1,1,1];

[0,0,1,0] −> [1,0,0,0];

[0,0,1,1] −> [1,0,0,1];

[0,1,0,0] −> [0,0,0,0];

[0,1,0,1] −> [0,0,0,1];

[0,1,1,0] −> [0,0,1,0];

[0,1,1,1] −> [0,0,1,1];

[1,0,0,0] −> [0,1,0,0];

[1,0,0,1] −> [0,1,0,1];

Test_vectors

([Bi3..Bi0] −> [Bo3..Bo0])

[0,0,0,0] −> [0,1,1,0];

[0,0,0,1] −> [0,1,1,1];

[0,0,1,0] −> [1,0,0,0];

[0,0,1,1] −> [1,0,0,1];

[0,1,0,0] −> [0,0,0,0];

[0,1,0,1] −> [0,0,0,1];

[0,1,1,0] −> [0,0,1,0];

[0,1,1,1] −> [0,0,1,1];

[1,0,0,0] −> [0,1,0,0];

[1,0,0,1] −> [0,1,0,1];

END

BCD Suma 6 con constructores ABEL

MODULE bcdadd6d

TITLE 'Lab1 Tutorial − Using Truth Table, with dont cares

Todd Morton, 10/19/00'

@DCSET

" Pin Declarations

Bi0,Bi1,Bi2,Bi3 pin;

Bo0,Bo1,Bo2,Bo3 pin istype 'com';

Truth_Table ([Bi3..Bi0] −> [Bo3..Bo0])

[0,0,0,0] −> [0,1,1,0];

[0,0,0,1] −> [0,1,1,1];

[0,0,1,0] −> [1,0,0,0];

[0,0,1,1] −> [1,0,0,1];

[0,1,0,0] −> [0,0,0,0];

[0,1,0,1] −> [0,0,0,1];

[0,1,1,0] −> [0,0,1,0];

[0,1,1,1] −> [0,0,1,1];

[1,0,0,0] −> [0,1,0,0];

[1,0,0,1] −> [0,1,0,1];

Test_vectors

([Bi3..Bi0] −> [Bo3..Bo0])

[0,0,0,0] −> [0,1,1,0];

[0,0,0,1] −> [0,1,1,1];

[0,0,1,0] −> [1,0,0,0];

[0,0,1,1] −> [1,0,0,1];

[0,1,0,0] −> [0,0,0,0];

[0,1,0,1] −> [0,0,0,1];

[0,1,1,0] −> [0,0,1,0];

[0,1,1,1] −> [0,0,1,1];

[1,0,0,0] −> [0,1,0,0];

[1,0,0,1] −> [0,1,0,1];

END

BCD Suma 6 con salida habilitada

MODULE badd6oe

TITLE 'Output Enable Example, BCDadd6 w/ OE

Todd Morton, 10/14/99'

```
" Pin Declarations

Bi0,Bi1,Bi2,Bi3 pin;

!OE pin;

Bo0,Bo1,Bo2,Bo3 pin istype 'com';

Bo = [Bo3..Bo0];


Equations


Bo0 = Bi0;

Bo1 = (!Bi3&!Bi2&!Bi1)#(Bi2&Bi1);

Bo2 = !Bi2&!Bi1;

Bo3 = !Bi2&Bi1;


" Output Enables

Bo.oe = OE;


Test_vectors


([Bi3..Bi0,OE] −> [Bo3..Bo0])

[0,0,0,0,0] −> [.z.,.z.,.z.,.z.];

[0,0,0,0,1] −> [0,1,1,0];

[0,0,0,1,1] −> [0,1,1,1];

[0,0,1,0,1] −> [1,0,0,0];

[0,0,1,1,1] −> [1,0,0,1];

[0,1,0,0,1] −> [0,0,0,0];

[0,1,0,1,1] −> [0,0,0,1];
```

[0,1,1,0,1] −> [0,0,1,0];

[0,1,1,1,1] −> [0,0,1,1];

[1,0,0,0,1] −> [0,1,0,0];

[1,0,0,1,1] −> [0,1,0,1];

END

CONCLUSIONES

- Es un lenguaje independiente del dispositivo, por lo que se puede utilizar para programar cualquier tipo de PLD.
- El diseño lógico puede ser descrito he introducido en tres formatos distintos, mediante ecuaciones, tablas de verdad y diagramas de estado.
- El diseño puede ser simulado para asegurarse de que no existan errores en el mismo.
- Mediante el proceso de la síntesis lógica el lenguaje optimiza y minimiza el diseño.
- Se ahorra más espacio y tiempo; ya que con la lógica MSI se utilizan mayor cantidad de compuertas y alambrado que con las aplicaciones del PLD programados con el lenguaje ABEL.

BIBLIOGRAFÍA

- Tocci, R. (1.996). **Sistemas Digitales. Principios y Aplicaciones**. Editorial Pearson Educación. Sexta Edición. México.
- Wakerly, J. (1992). **Diseño Digital Principios y Prácticas**. Editorial Prentice may Hispanoamericana. México.
- **ABEL−HDL Primer** [2000−2001] On−Line. Disponible en:

http://www.ee.upenn.edu

- **ABEL−HDL Examples** [1998−1999] On−Line. Disponible en:

http://eet.etec.wwu.edu/etec373/SynarioExs/

32