

MANUAL DE *ispLEVER*

1. Introducció

ispLEVER és un programa enfocat cap al disseny i simulació de circuits digitals, tant a partir de components discrets com aplicat a dispositius lògics programables (PLD's, CPLD, FPGA etc.). Aquest programa permet introduir el disseny tant a partir d'esquemàtics (portes, biestables, etc.) com mitjançant llenguatges d'alt nivell (ABEL-HDL, Verilog, VHDL).

Aquest manual explica bàsicament quins són els passos que s'han de seguir per a introduir esquemàtics, simular-los i incorporar el disseny global en dispositius programables del fabricant Lattice (<http://www.latticesemi.com/>).

2. *ispLEVER Project Navigator*

Un cop que s'és a dins del Windows cal executar (doble clic) l'enllaç al programa *ispLEVER*. La finestra que s'obra és la de la figura 1.

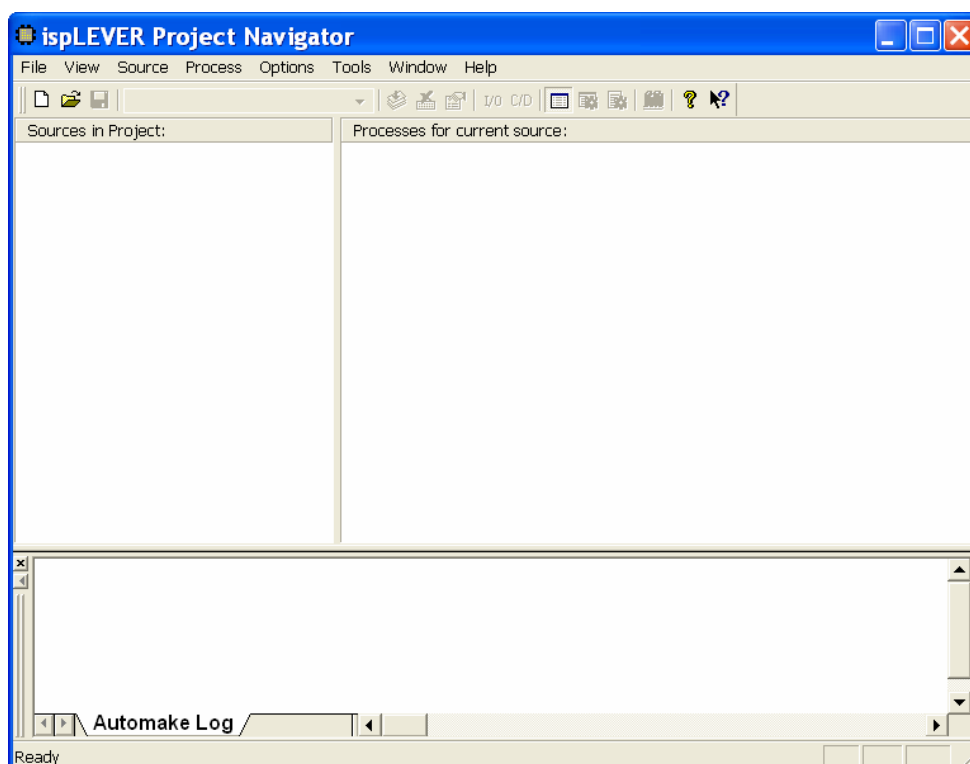


Fig. 1: *ispLEVER Project Navigator* un cop dins el programa.

Aquesta finestra es correspon amb el nucli del programa: *ispLEVER Project Navigator*. Serà aquesta finestra la que servirà d'enllaç per totes les parts del procés de disseny. Està dividida en dues subfinestres:

a) *Sources in Project*: Ens indicarà quins són els fitxers que estan referits al projecte en curs i que en general haurà generat el mateix usuari (esquemàtics, fitxers HDL i fitxers de simulació).

b) *Processes for current source*: Un cop assenyalat amb el ratolí un dels fitxers de la subfinestra *Sources in Project* automàticament apareixeran tots els ítems relacionats amb el mateix. Aquests ítems poden ser fitxers de text, fitxers de dades o executables. Els fitxers de text tindran una icona amb unes ratlles (simulant text). Els fitxers de dades tindran una icona diferent (per exemple un cilindre) i els executables dues fletxes girant sobre elles mateixes. En general, tant per a obrir un fitxer de text o executar un subprograma caldrà fer doble click sobre l'ítem en qüestió. Els fitxers estan ordenats per jerarquia de tal manera que cal obrir-los o executar-los de manera successiva. En qualsevol cas, si s'executa un ítem que necessita de passos previs que encara no s'han fet el programa els executarà fins a arribar a donar el resultat esperat. Si sorgeix un error ho indicarà posant una 'X' al costat del procés problemàtic i obrint una finestra on s'indicarà quin tipus d'error s'ha donat.

Com es pot veure a la figura 1 hi ha una icona petita simulant un disquet que serveix per gravar el disseny (qualsevol part inclosa a dins). **Es molt recomanable anar gravant el disseny de tant en tant de manera d'evitar futurs problemes. Feu a més còpies de seguretat quan finalitzeu la sessió.**

3. Creació d'un nou projecte

Dins el menú *File* s'ha de triar l'opció *New Project*. Apareixerà una finestra on podrem triar la unitat de disc i directori de treball (aquest s'ha de crear prèviament des de Windows). A la finestra que us apareixerà (el *Project Wizard*) heu de dir que el projecte es dirà *pract3* i que el disseny serà entrat mitjançant captura d'esquemàtics i ABEL-HDL (veure figura 2). Un cop fet això aneu picant a *Següent* acceptant els valors per defecte que us aparegui a pantalla fins finalitzar la creació del projecte.

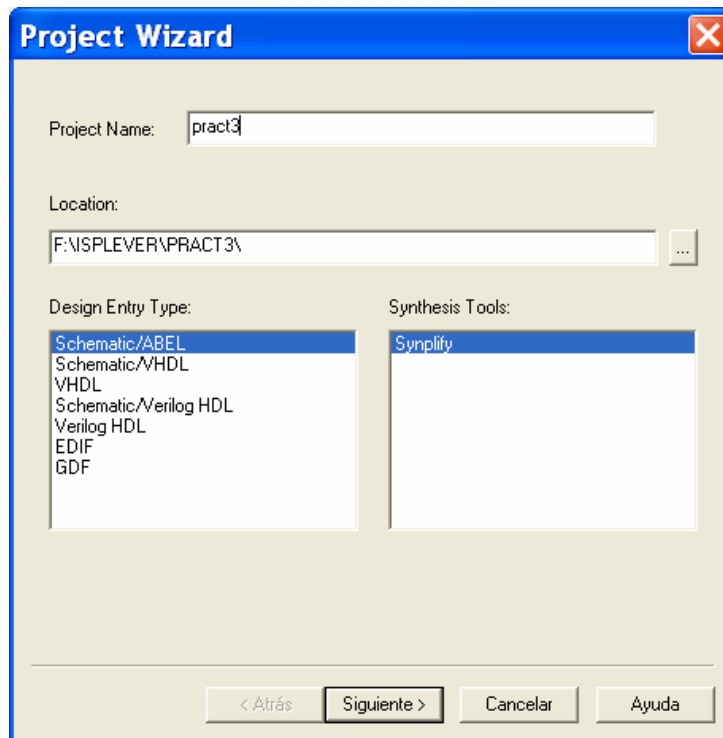


Fig. 2: Creació del nou projecte amb el *ispLEVER Project Wizard*.

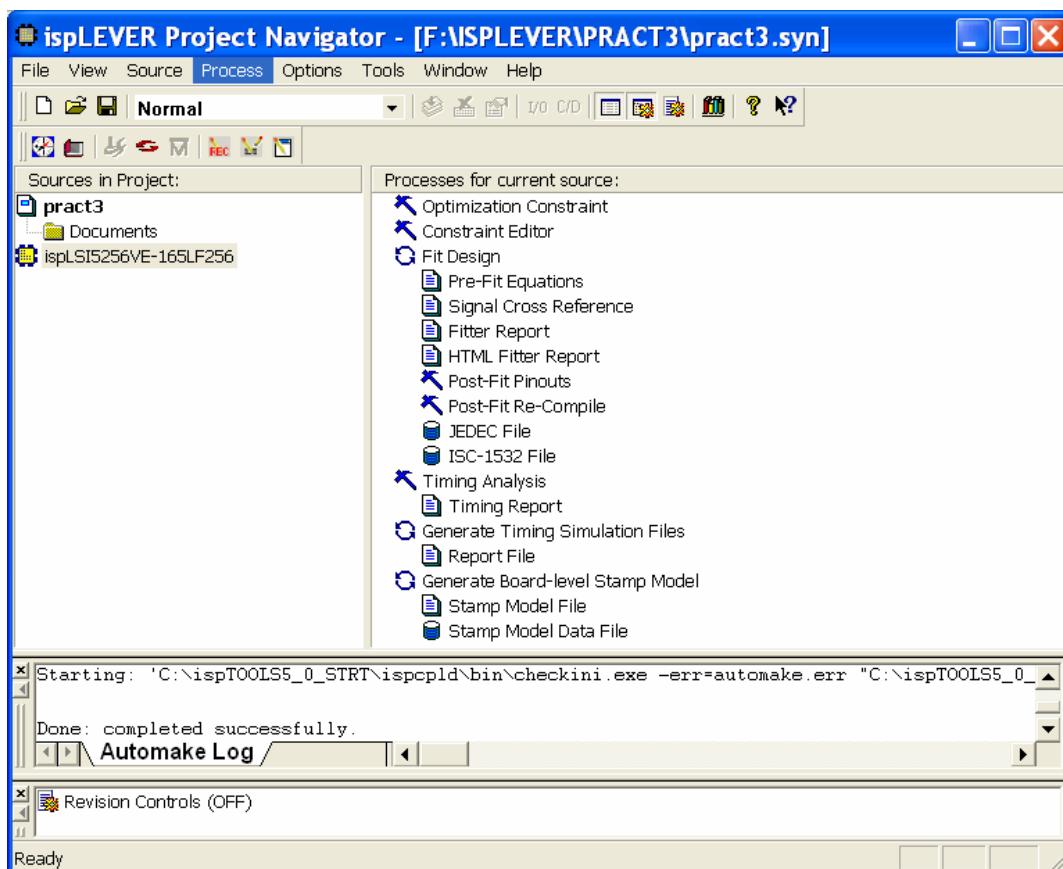


Fig. 3: *ispLEVER Project Navigator* un cop creat el nou projecte (*pract3*).

4. Disseny d'un nou esquemàtic

Per generar un nou esquemàtic heu de fer al *Project Navigator: Source* → *New* i a la finestra que apareix triar *Schematic* (fixeu-vos en els altres formats que podeu fer servir com a entrada pel *ispLEVER*), triant a continuació com a nom *Modul*. Un cop fet això s'obrirà la finestra del *Schematic Editor* (Fig. 4).

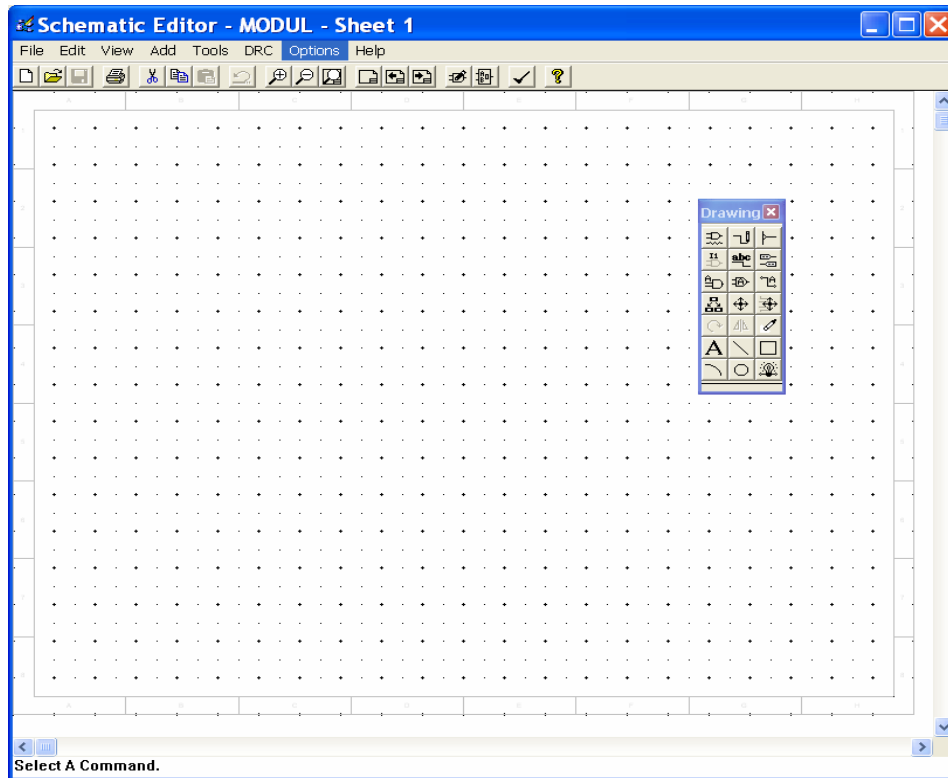


Fig. 4: *Schematic Editor*.

La seqüència de passos necessària per a dissenyar un esquemàtic és la següent:

a) Introduir components i cables que els uneixin segons l'esquema que haguem predefinit. Per a introduir components s'ha de triar el menú *Add* i després *Symbol*. Llavors apareix una finestra amb totes les llibreries que hi són instal·lades (*ARITHS.LIB*: Full Adders i Half Adders, *GATES*: portes, *IOPADS*: Buffers d'entrada i sortida, *REGS.LIB*: flip-flops, *MISC.LIB* : per afegir títol i explicacions a l'esquemàtic, i (*Local*): la llibreria que inclou el projecte actual). Es tria el component necessari i es situa a la finestra d'esquemàtics. Per a afegir cables s'ha de triar el menú *Add* i després *Add wire*.

Per a dibuixar un *wire* en un punt concret o posar en un determinat lloc un component s'ha de prémer el botó de l'esquerra del ratolí i per a cancel·lar l'ordre el de la dreta.

b) Donar nom als *wires* d'entrades i sortides o qualsevol altre del qual volguéssim veure l'evolució. Per això cal anar al menú *Add* i escollir *Add Net Name*. Escriviu el nom contestant a la pregunta *Net Name* que us fa l'editor (observeu la figura 5) i, després de picar a l'enter, amb el ratolí piqueu a l'extrem lliure del cable corresponent.

c) Especificar entrades i sortides: *Add* → *Add I/O marker*: Sortirà una finestra petita demanant si són sortides o entrades. És necessari situar de nou el ratolí sobre l'extrem del *wire*.

Com a exemple referència, a partir d'ara es parlarà del disseny d'un comptador binari *ripple adder* de 4 bits fet a partir de bistables tipus D que inclourà 3 mòduls: MODUL (conversió de bistable tipus D a T), COMP1BIT (comptador d'un bit) i COMPT (comptador total).

El resultat de dibuixar el component MODUL (a falta d'afegir el senyal CLK) es pot veure a la fig. 5.

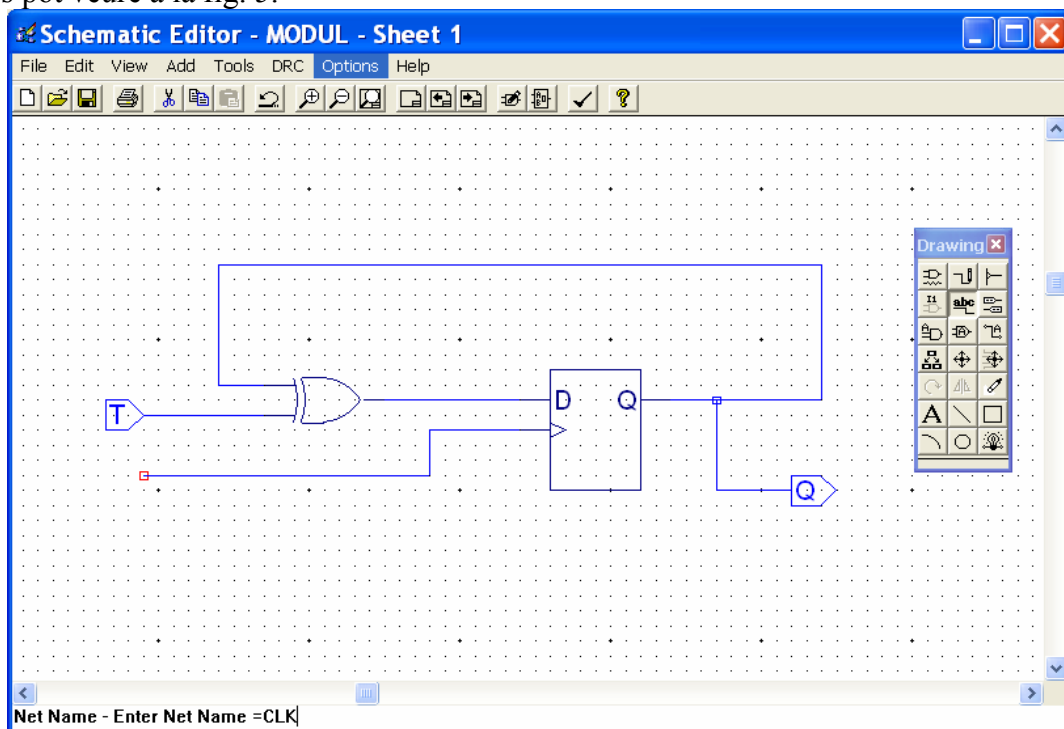


Fig. 5: Conversió d'un *flip-flop* tipus D en un T (esquemàtic MODUL).

Un cop dibuixat l'esquemàtic cal generar el seu símbol. Per això cal anar a *Add* → *New Block Symbol*. Sortirà una finestra on és molt aconsellable fer un *click*

sobre *Use data from this block* per a que detecti automàticament les entrades i sortides, llavors es fa *RUN* i es generarà el nou símbol. A continuació tanqueu la finestra del *Schematic Editor*, gravant abans si no s'ha fet encara.

L'aspecte que tindrà el *ispLEVER Project Navigator* es pot veure a la figura 6.

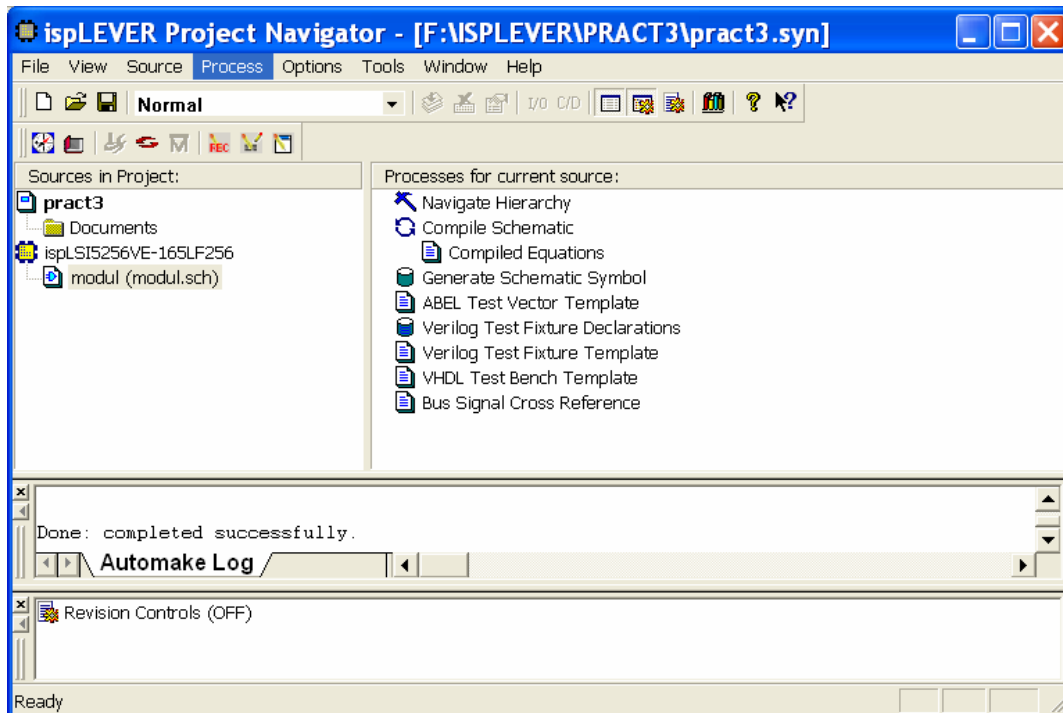


Fig. 6: Aspecte del *ispLEVER Project Navigator* després d'haver creat MODUL.

Selecciónant a dins de la finestra *Sources in Project* l'esquemàtic que acabem de generar tal i com es pot veure a la figura 6, apareixen tota una sèrie de processos associats a MODUL a dins de la finestra *Processes for current Source*. En general serà convenient executar tots els ítems. Per exemple, ara s'ha de compilar el disseny (per a tal de detectar errors com deixar entrades sense connectar, etc.). Un cop compilat apareix una marca (“tick”) de color verd al costat de la compilació com es pot veure a la figura 7.

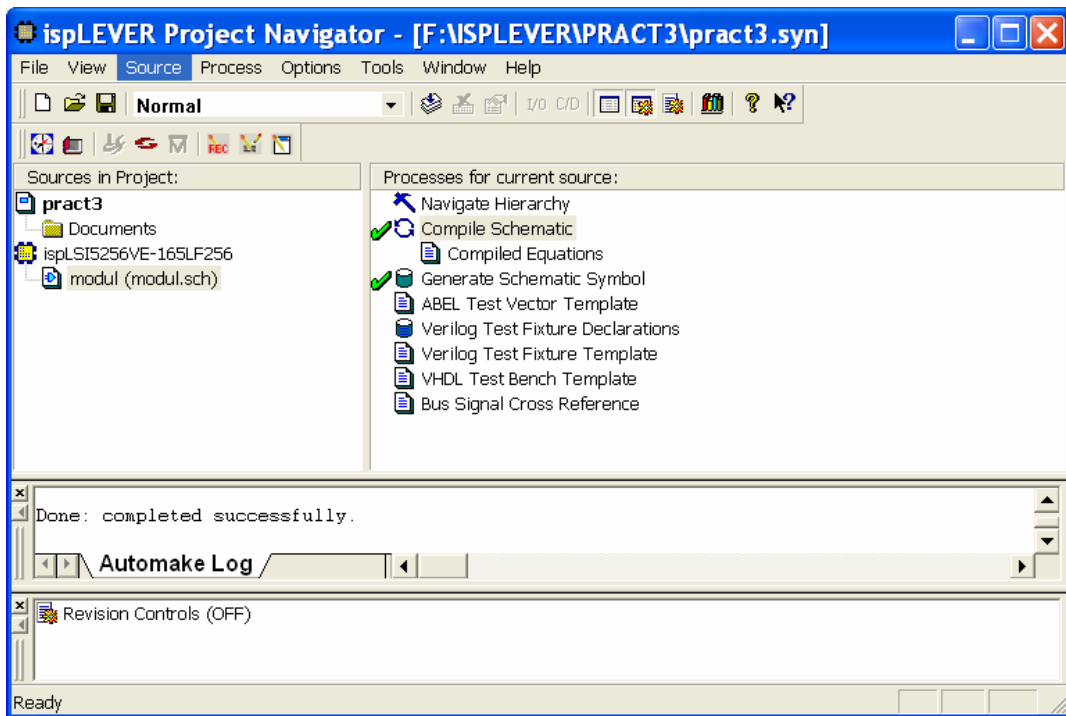


Fig. 7: *ispLEVER Project Navigator*, un cop compilat MODUL.

Ara cal continuar dibuixant COMP1BIT (fig. 8 i 9) i COMPT (fig. 10).

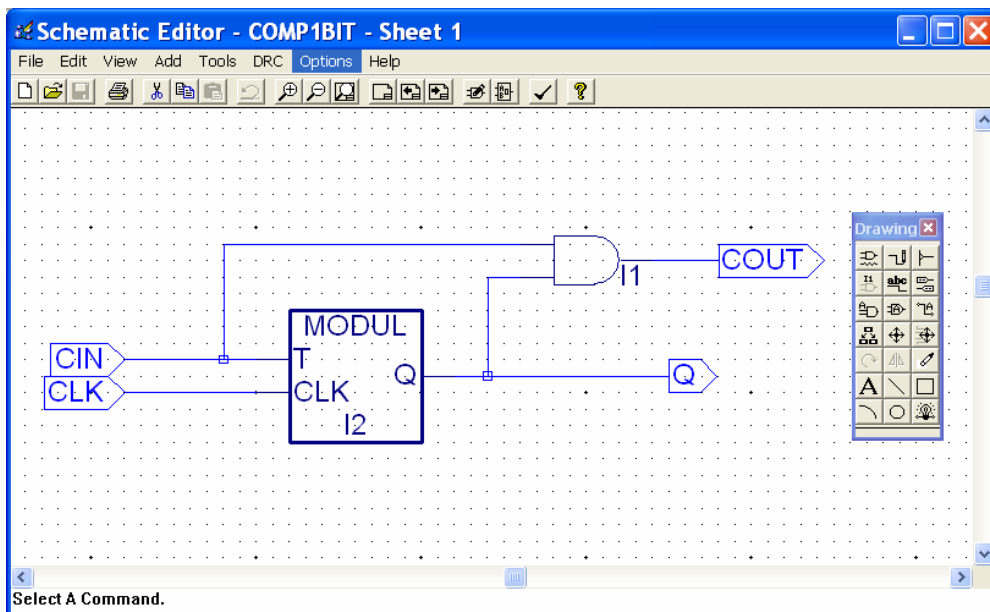


Fig. 8: Esquemàtic de COMP1BIT.

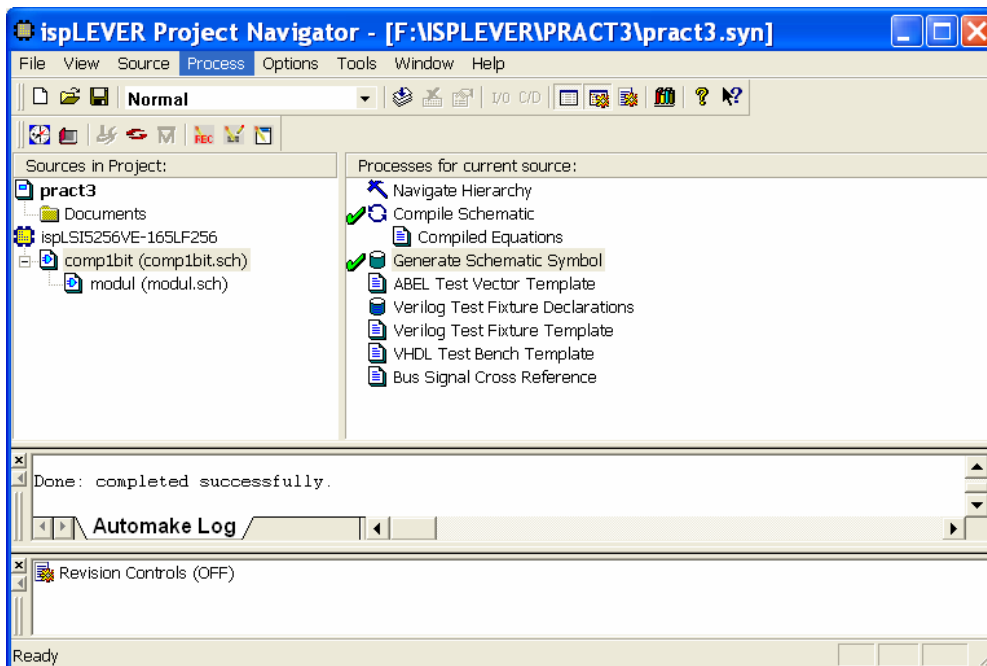


Fig. 9: *ispLEVER Project Navigator* un cop generat l'esquemàtic i el símbol de COMP1BIT. Com es pot veure ha detectat automàticament la jerarquia (COMP1BIT té a dins MODUL).

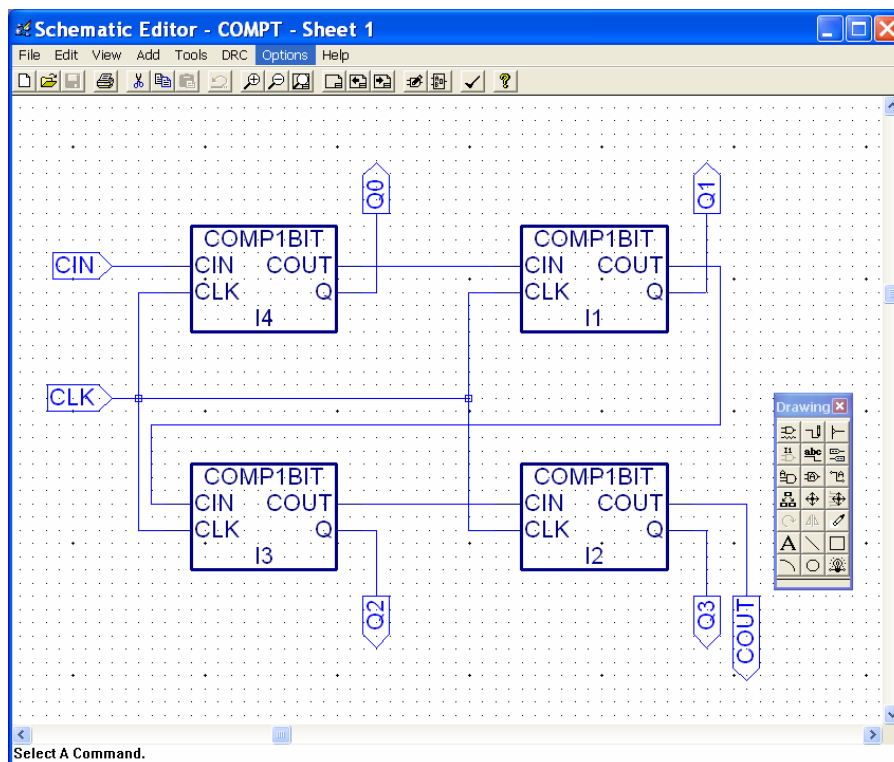


Fig 10: Esquemàtic de COMPT (comptador de 4 bits).

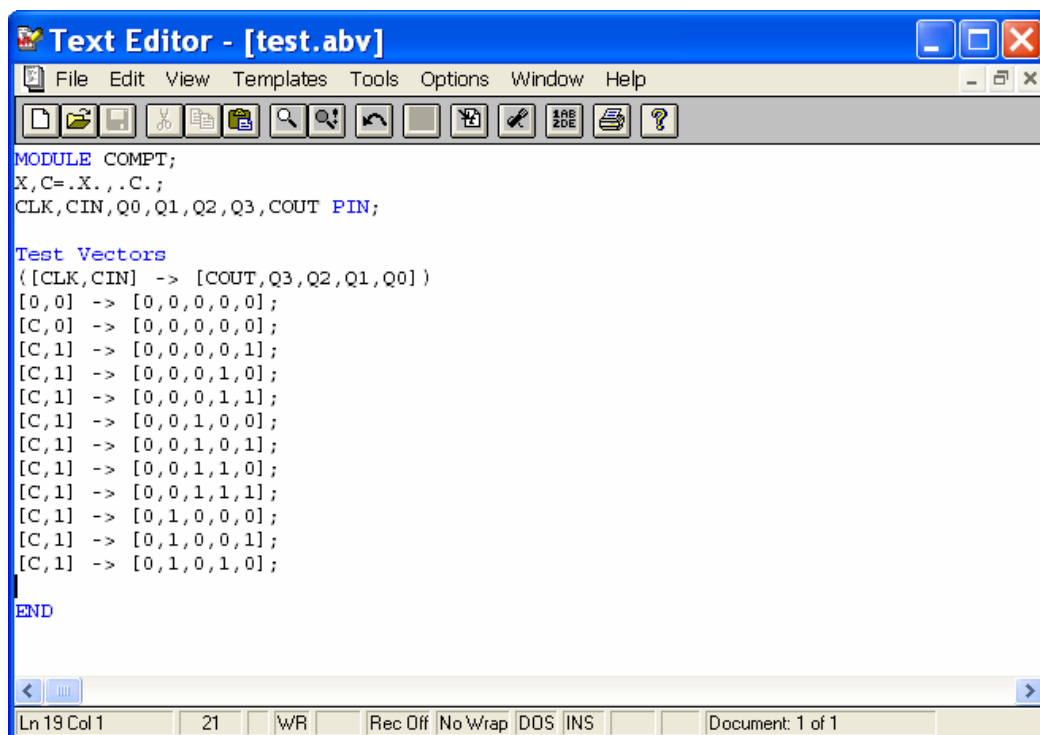
Un cop generat el símbol el *ispLEVER Project Navigator* haurà detectat automàticament la jerarquia i per tant el *top symbol* serà ara COMPT.

5. Simulació d'un projecte

Un cop dissenyades totes les parts i mòduls d'un projecte s'ha de simular per a verificar que efectivament compleix les especificacions de partida. El procés de simulació inclou bàsicament tres parts:

- Creació del fitxer de simulació (llenguatge ABEL-HDL).
- Execució del simulador.
- Lectura de dades (*Wave editor*).

El llenguatge necessari per a generar els estímuls és molt ampli (ABEL). Només tractarem aquelles qüestions necessàries per a simular de manera simple. A la figura 11 es pot veure el que serà el fitxer *TEST.ABV*. Per escriure aquest fitxer s'ha de crear un nou ítem (*New*) a *Sources in Project* i llavors indicar que es vol crear un fitxer *ABEL Test Vectors*, al qual direm *test.abv*.



```
MODULE COMPT;
X, C= .X. , .C. ;
CLK, CIN, Q0, Q1, Q2, Q3, COUT PIN;

Test Vectors
( [CLK, CIN] -> [COUT, Q3, Q2, Q1, Q0] )
[0, 0] -> [0, 0, 0, 0, 0];
[C, 0] -> [0, 0, 0, 0, 0];
[C, 1] -> [0, 0, 0, 0, 1];
[C, 1] -> [0, 0, 0, 1, 0];
[C, 1] -> [0, 0, 0, 1, 1];
[C, 1] -> [0, 0, 1, 0, 0];
[C, 1] -> [0, 0, 1, 0, 1];
[C, 1] -> [0, 0, 1, 1, 0];
[C, 1] -> [0, 0, 1, 1, 1];
[C, 1] -> [0, 1, 0, 0, 0];
[C, 1] -> [0, 1, 0, 0, 1];
[C, 1] -> [0, 1, 0, 1, 0];

END
```

Fig. 11: Fitxer d'estímuls (TEST.ABV) del comptador binari.

A continuació tractem cada ordre per separat.

- Module COMPT;* -> Indica que el mòdul a simular és COMPT.

b) $C, X = .C., .X.$ → Definicions. $.C.$ equival a un pols de rellotje. $.X.$ vol dir indeterminació. Aquesta definició equival a definició de constants (com, per exemple, a C: "#define C .C.").

c) $CLK, CIN, COUT, Q0, Q1, Q2, Q3 PIN$; → Definició de pins d'entrada i sortida. Amb aquesta instrucció indiquem quins són els senyals que tractarem a la simulació com a entrades o sortides. **Es important que estiguin en majúscules o minúscules segons haguem definit els senyals a l'esquemàtic.**

d) $TEST_VECTORS$ → Indicarem quina seqüència de vectors de test aplicarem. Primer indiquem la implicació de les entrades vers les sortides: $([CLK, CIN] \rightarrow [COUT, Q3, Q2, Q1, Q0])$. A continuació posem els valors dels senyals d'entrada i quins són els valor esperats a les sortides. Si no volem especificar cap valor, posarem X (equivalent a $.X.$). Quan es vulgui posar un flanc de rellotge s'ha de posar C (equivalent a $.C.$). Es molt convenient posar el valor esperat dels senyals de sortida, ja que així a l'hora de simular el programa ens dirà si els vectors de test han estat correctes o no.

e) END .

Advertències:

- 1) Només es pot tenir un fitxer d'estímuls per projecte (ABEL Test Vectors).**
- 2) El fitxer de simulació sempre ha de referir-se (instrucció *module*) al bloc de jerarquia màxima.**
- 3) Només pot haver-hi un bloc de jerarquia màxima (un bloc que englobi tots els altres).**

A la figura 11 es pot veure el que serà el significat d'aquest fitxer ABEL. A la part de dalt hi ha les entrades tal com les hem definit i a baix les sortides. Les parts ombrejades són les que el programa no avalua des del punt de vista de vectors de test. És a dir, les parts ombrejades són simulades però no es comprova si el valor que prenen els senyals són els definits en els vectors de test.

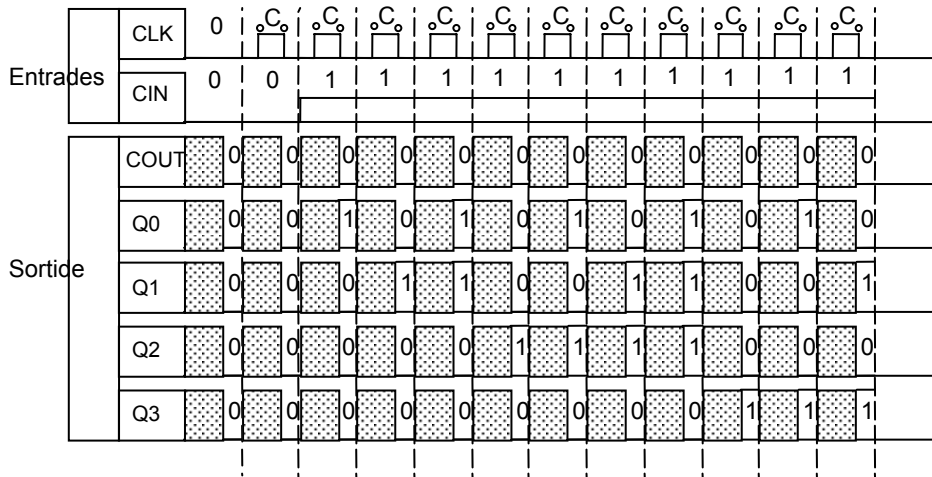


Fig. 12: Significat del fitxer TEST.ABV (ABEL Test Vectors).

Un cop gravat i compilat el fitxer d'estímuls, l'aspecte del *ispLEVER Project Navigator* serà el de la figura 13.

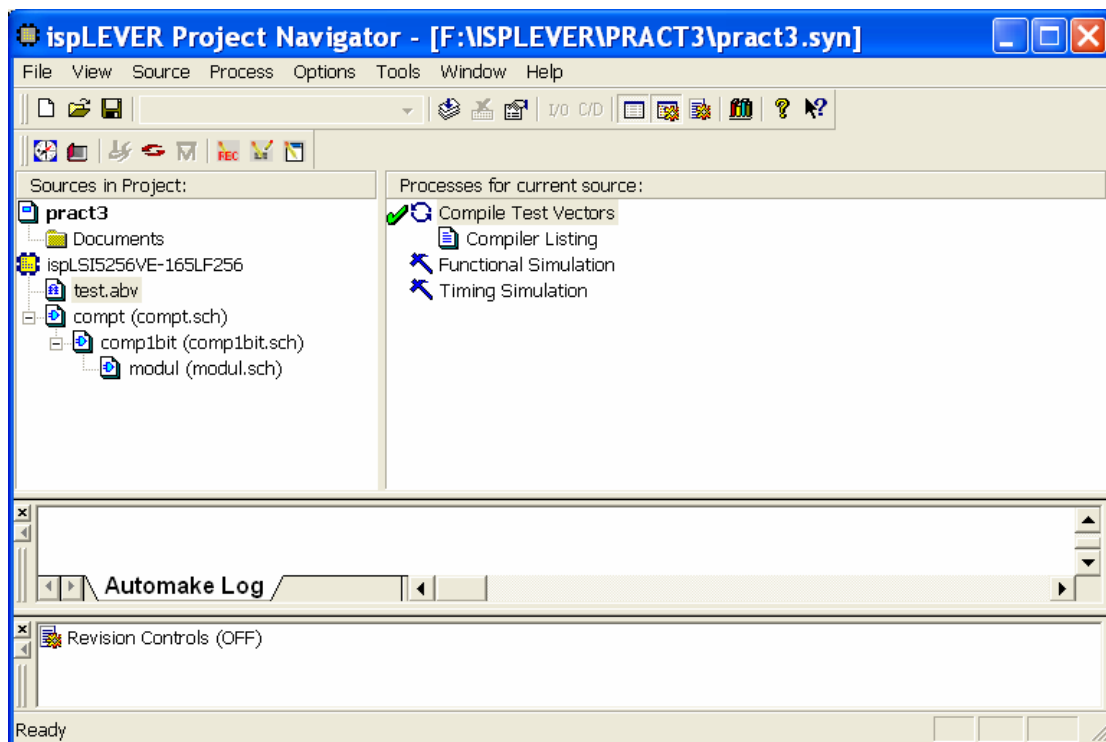


Fig. 13: *ispLEVER Project Navigator* un cop compilat TEST.ABV.

Si fem doble click sobre *Functional Simulation* apareixerà el *Simulator Control Panel* (figura 14). Fent *Simulate*→*Run* es simularà el circuit dissenyat amb els vectors de test descrits al fitxer *test.abv*. Automàticament s'obre el *Waveform Viewer*, amb tots els senyals de la simulació (figura 15). Per veure altres possibles senyals cal fer *Edit*→*Show*→ indicar els senyals que es volen veure i *Show* (proveu

d'agrupar els senyals $Q3$ a $Q0$ en un bus amb la pestanya *Bus*). La simulació funcional considera retards, però ja que la simulació es fa abans de fer el *fitting* dins del dispositiu definitiu que es farà servir per implementar el disseny, són retards estimats. L'avantatge és que es pot fer la simulació i detectar possibles errors sense haver de fer el *fitting*, assignació de pins etc. Una vegada fet el *fitting* definitiu (el fem més endavant) s'ha de fer una simulació temporal, *Timing Simulation*.

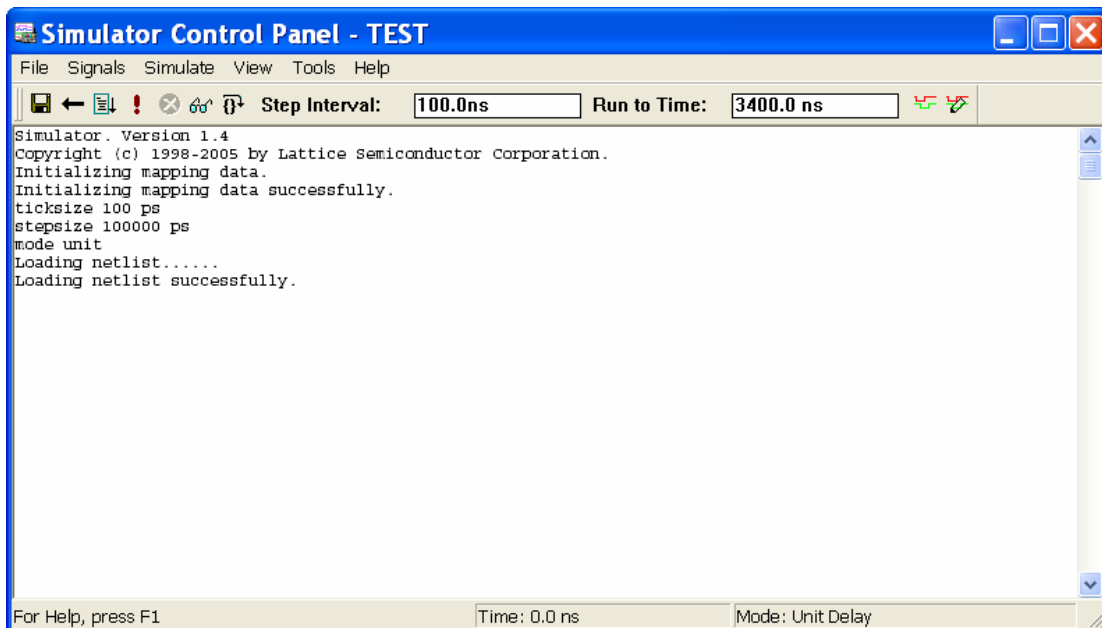


Fig. 14: Finestra de control del simulador del *ispLEVER*.

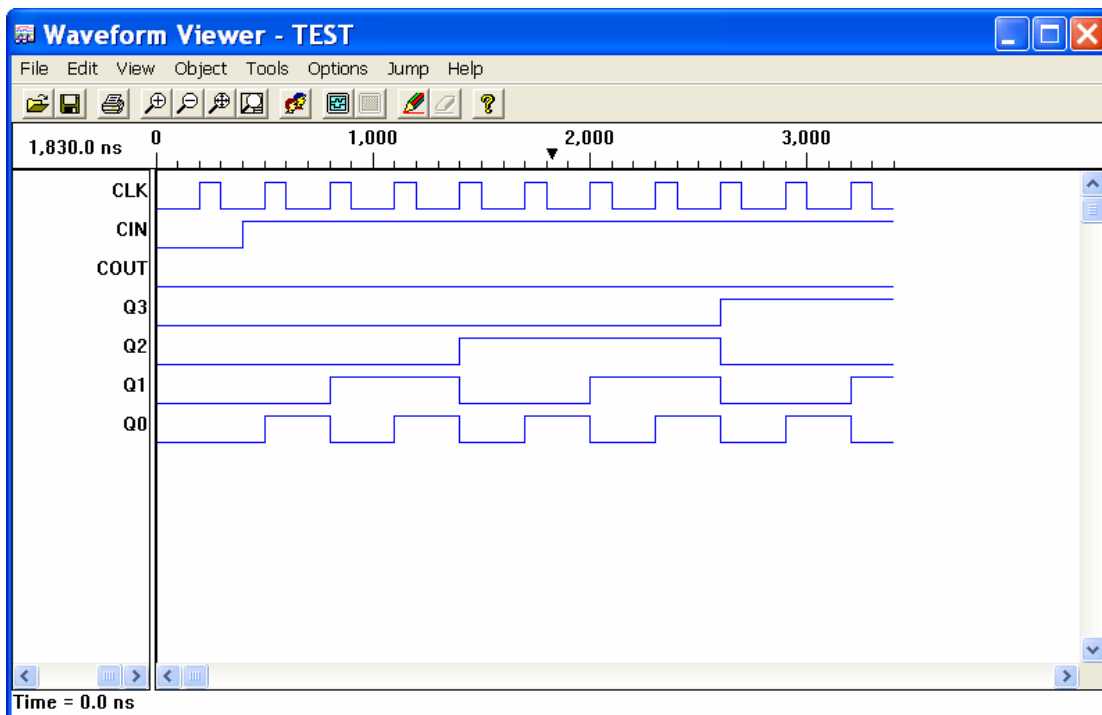


Fig. 15: Formes d'ona de la simulació del circuit COMPT segons els vectors de test descrits al fitxer TEST.ABV.

6. Inclusió del disseny dins d'un PLD

Un cop verificat el bon funcionament del circuit s'ha de procedir a especificar un dispositiu lògic sobre el qual gravarem el disseny. Com a exemple utilitzem el PLD ispLSI 2032-110. Per a altres dispositius el procediment és anàleg. Per canviar el dispositiu s'ha de fer doble clic sobre el dispositiu que havíem triat per defecte (al costat de la icona amb forma de “xip”). Seleccionarem la família **ispLSI 2K**, dispositiu **2032**, versió (velocitat) **110**, i encapsat **44PLCC**, és a dir, el **ispLSI 2032-110 PLCC44** (en realitat el dispositiu amb el que treballarem és el **ispLSI 2032-80**, que no és més que una versió més antiga, a 80MHz, però funciona tot igual). L'aspecte del *ispLEVER Project Navigator* serà el de la figura 16.

Els processos associats a ispLSI 2032-110 són els que haurem d'executar per obtenir finalment el fitxer JEDEC que servirà per gravar definitivament el disseny sobre el PLD.

Un cop executats tots els ítems relacionats amb el dispositiu l'aspecte del *ispLEVER Project Navigator* serà el que es pot veure a la figura 17.

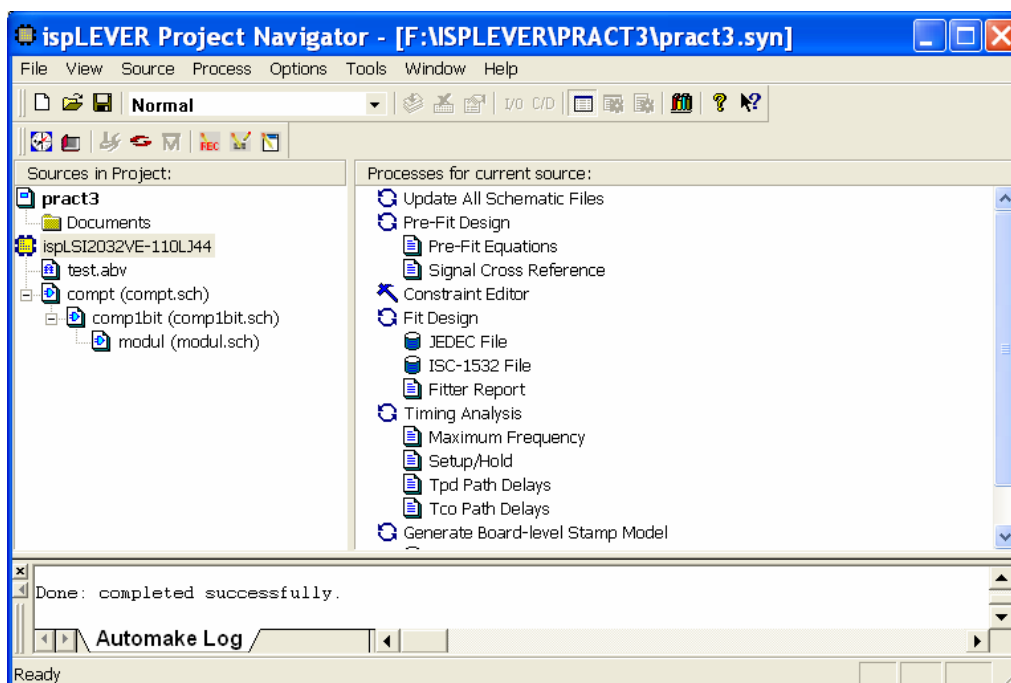


Fig. 16: *ispLEVER Project Navigator* un cop definit el nou dispositiu.

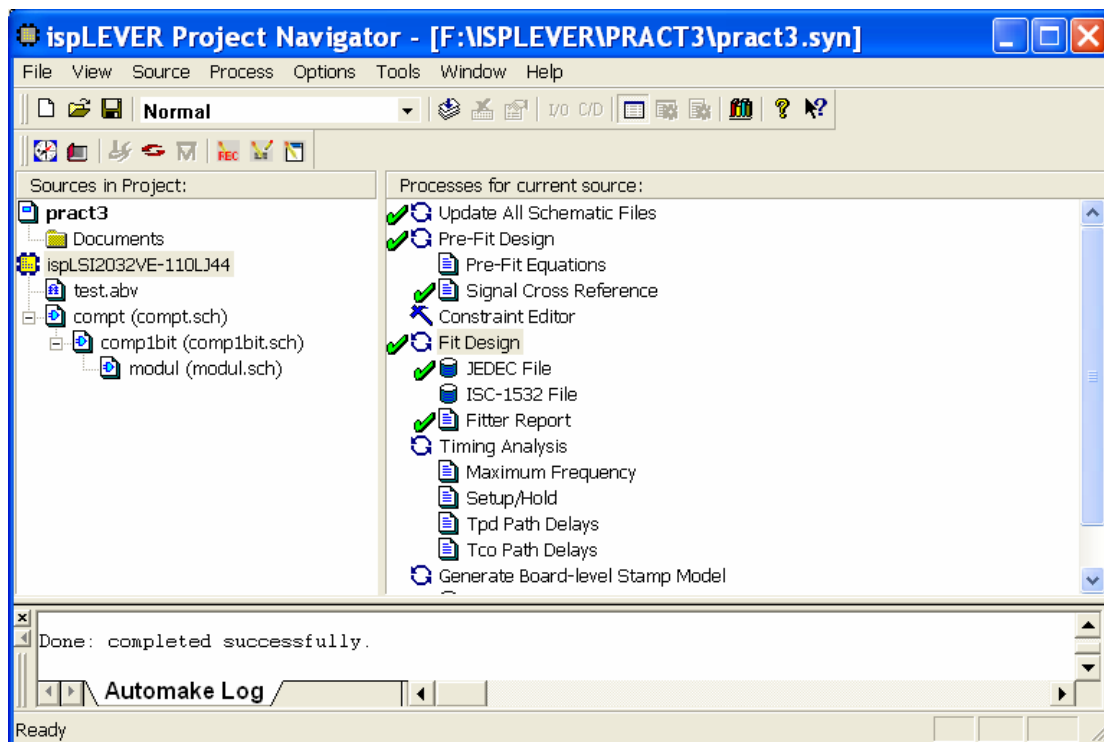


Fig. 17: Projecte un cop generat el fitxer JEDEC.

Amb aquests passos s'ha acabat el procés complet. És important tenir en compte que ara es possible simular el fitxer JEDEC considerant, per tant, els retards exactes deguts a la implementació concreta sobre el PLD sobre el que farem l'experimentació. El fitxer JEDEC conté els vectors de test que haguem definit en el fitxer ABEL. El gravador de PLD's comprova que efectivament es compleixen. És convenient, per tant, fer els vectors de test de l'últim mòdul tant exhaustius com sigui possible.

Fent doble clic sobre l'ítem *Fitter report* podrem veure l'assignació de pins.

6. Estratègia general per a la realització d'un projecte

Convé crear cada projecte en un directori dedicat. En un projecte hi poden conviure diversos dissenys. Es recomana crear un projecte per a cada pràctica. Per importar un esquemàtic d'un altre projecte (o del projecte actiu) s'ha d'anar al menú *Source* del *ispLEVER Project Navigator* i importar el fitxer del directori corresponent.

Donat que en un projecte només es pot simular el bloc de jerarquia superior (només es pot tenir un fitxer de vectors de test *.abv actiu), per simular un bloc dins

d'una jerarquia primer cal eliminar els de jerarquia superior. Per recuperar la jerarquia, s'importen els blocs corresponent per ordre jeràrquic. Degut a això, és convenient anar simulant a mida que es van creant els dissenys, en ordre creixent de jerarquia.

S'ha comentat en aquesta introducció a *ispLEVER* la importància de fer vectors de test exhaustius. Per tal de facilitar la tasca repetitiva d'introduir un mateix vector, el programa disposa de la directiva **@Repeat**, que permet repetir *n* cops un mateix bloc de text, en particular un vector de test. La sintaxi d'aquesta directiva és la següent: *@repeat expression { block }* on *expression* és una expressió numèrica, i *block* un bloc de text. Per exemple, escriure la línia *@repeat 200 { [C]->[X] }* equival a escriure dos-cents cops la línia *[C]->[X]*.

Alguns cops els senyals d'entrada a un bloc seran valors constants, '1' o '0'. Quan això s'escaigui caldrà introduir directament a l'esquemàtic corresponent els símbols *Vcc* o *Gnd* (fent *Add->Net Name->Vcc* (o *Gnd*)).

INTRODUCCIÓN AL ABEL – HDL

ABEL – HDL es un lenguaje jerárquico de descripción lógica de circuitos electrónicos (*Hardware Description Language*) que soporta distintos formatos de entrada, como por ejemplo ecuaciones lógicas, diagramas de estado y tablas de verdad. ABEL (*Advanced Boolean Equation Language*) permite describir el comportamiento de un diseño circuital así como realizar su verificación sin necesidad de especificar el dispositivo donde se realizará, lo cual permite centrarse en el diseño. Una vez el diseño ha sido entrado y verificado funcionalmente, puede ser optimizado para su implementación en un dispositivo específico.

Esta introducción al lenguaje ABEL se divide en dos partes: en la primera se dan las bases de la sintaxis y de la estructura del lenguaje, y en la segunda se muestran unos ejemplos de descripción circuital.

1. ESTRUCTURA DEL LENGUAJE ABEL–HDL

1.1. SINTAXIS BÁSICA

La descripción de un circuito debe cumplir las siguientes reglas sintácticas:

- a) Una línea no puede tener más de 150 caracteres.
- b) Una línea debe acabar con un retorno (*Enter*) del teclado.
- c) Las palabras clave, los identificadores (ambos tratados más adelante) y los números deben de estar separados por un espacio. Excepciones a esta regla son las listas de identificadores (que pueden separarse con comas), así como expresiones cuyos identificadores o números vayan separados por operadores o por paréntesis.
- d) Ninguna palabra clave, identificador, número u operador puede incluir espacios en blanco ni comas o puntos y comas (en algún caso específico, tratado más adelante, pueden incluirse puntos).
- e) Soporta caracteres **ASCII**, y distingue letras mayúsculas de minúsculas.

1.2 ELEMENTOS USADOS EN EL LENGUAJE ABEL-HDL

La descripción de un circuito puede incluir algunos de los siguientes elementos:

1.2.1 Identificadores. Son nombres que sirven para identificar:

- * Dispositivos
- * Pines de dispositivo o nodos
- * Señales de entrada y salida
- * Constantes

Las reglas sintácticas específicas de los identificadores son:

- * Los identificadores pueden tener hasta un máximo de 31 caracteres.
- * Deben comenzar con una letra o con un *underscore*, "_".

1.2.2. Palabras clave. Son identificadores reservados que no pueden usarse para identificar dispositivos, pines, bloques, etc. Son:

| | | |
|------------------|-----------|----------------|
| asyn_reset | fuses | state |
| case | goto | state_diagram |
| declarations | if | state_register |
| device | in | sync_reset |
| else | interface | test_vectors |
| enable | istype | then |
| end | library | title |
| endcase | macro | trace |
| endwith | module | truth_table |
| equations | node | when |
| external | options | with |
| flag | pin | |
| functional_block | property | |

1.2.3. Constantes. Se utilizan para definir el valor de identificadores, de las variables de una tabla de verdad y de las de los vectores de test. También se dispone de constantes especiales como las de la tabla 1. Al utilizar tales constantes deben escribirse como muestra la tabla (incluyendo los puntos).

| Constante | Descripción |
|-----------|---|
| .C. | Reloj (transición bajo - alto - bajo) |
| .D. | Reloj (transición alto - bajo) |
| .K. | Reloj (transición alto - bajo - alto) |
| .U. | Reloj (transición bajo - alto) |
| .X. | Condición de 'desconocido' o 'no importa' |
| .Z. | Estado de alta impedancia |

Tabla 1. Constantes especiales

1.2.4. Bloques. Son secciones de texto entre corchetes, "{" y "}". El texto puede extenderse en una o en varias líneas. Un ejemplo de utilización de bloques es en expresiones lógicas, y también en diagramas de estado, como se muestra en el siguiente ejemplo:

```
If (Hold) THEN
{
    IF (!Reset) THEN State1;
    IF (Error) THEN State2; }
ELSE State3;
```

1.2.5. Comentarios. Sirven para aclarar o hacer más inteligible el código. Hay dos formas de introducir comentarios: comenzando con dobles comillas " y acabando o con dobles comillas o con fin de línea, o bien comenzando con // y acabando con fin de línea.

1.2.6. Cadenas de caracteres (*Strings*): Son series de caracteres ASCII entre signos de apóstrofe. Se usan en el elemento TITLE, y en las declaraciones de atributos de señales, como se indica más adelante.

1.2.7. Operadores. En ABEL-HDL los operadores pueden dividirse en: lógicos, aritméticos, de relación y de asignación:

Lógicos (se realizan bit a bit)

| | |
|-----|------|
| ! | NOT |
| & | AND |
| # | OR |
| \$ | XOR |
| !\$ | XNOR |

Aritméticos (para cualquier tipo de operandos)

| | |
|---|---|
| - | complementario (de un operando p.e. -A) |
| - | diferencia (de dos operandos p.e. A-B) |
| + | suma (de dos operandos p.e. A+B) |

(para operandos que no sean *sets*)

| | |
|----|--|
| * | producto (p.e. A*B) |
| / | división entera (p.e. A/B) |
| % | módulo (p.e. A%B) |
| << | desplazar hacia la izq B bits (p.e. A<<B) |
| >> | desplazar hacia la dcha B bits (p.e. B>>A) |

De relación (estos operadores comparan dos miembros de una expresión lógica y dan como resultado Verdadero o Falso):

| | |
|----|-------------------|
| == | igual que |
| != | distinto que |
| < | menor que |
| <= | menor o igual que |
| > | mayor que |
| >= | mayor o igual que |

De asignación. Se usan en ecuaciones algebraicas y en ecuaciones lógicas:

| | |
|----|--|
| = | Combinacional |
| := | Secuencial (la salida se actualizará cuando llegue la señal de sincronización del reloj) |

Prioridad entre operadores

En la siguiente tabla se especifica la prioridad entre operadores valorada entre la mayor (1) y la más baja (4). Los operadores con la misma prioridad se ejecutan de izquierda a derecha.

| Operador | Descripción | Prioridad |
|----------|---------------------|-----------|
| - | Complemento | 1 |
| ! | NOT | 1 |
| & | AND | 2 |
| << | Desplazamiento izq | 2 |
| >> | Desplazamiento dcha | 2 |
| * | producto | 2 |
| / | División entera | 2 |
| % | Módulo | 2 |
| + | Suma | 3 |
| - | Diferencia | 3 |
| # | OR | 3 |
| \$ | XOR | 3 |
| !\$ | XNOR | 3 |
| == | Igual que | 4 |
| != | Distinto que | 4 |
| < | Menor que | 4 |
| <= | Menor o igual que | 4 |
| > | Mayor que | 4 |
| >= | Mayor o igual que | 4 |

1.2.8 Sets: Son grupos de señales y/o constantes al conjunto de los cuales se asigna un nombre. En su definición se delimitan por corchetes, [y], y pueden utilizar los dos puntos (..) para indicar un rango entero, incremental o decremental. Una operación aplicada a un set se aplica a cada elemento del set. Como ejemplo, las señales de entrada y salida de un decodificador BCD–decimal se pueden representar por dos sets:

El de entrada : $BCD = [A0,A1,A2,A3]$

o

$BCD = [A0 .. A3]$

y el de salida: $SAL = [S0,S1,S2,S3,S4,S5,S6,S7,S8,S9]$

o

$SAL = [S0 .. S9]$

1.3.- ESTRUCTURA BÁSICA DE UN FICHERO ABEL

Un fichero ABEL puede contener módulos independientes. Cada módulo debe contener una descripción lógica de un circuito y estos módulos pueden combinarse para ser procesados a la vez. Un módulo se divide en las siguientes secciones:

1.3.1. Encabezamiento

Esta sección debe ser la que comience un módulo ABEL. Consta de los siguientes elementos:

1. Obligatorio: **Module** (se requiere esta palabra antes del nombre del módulo (identificador))
2. Opcional: **Title** (título) debe indicarse encerrado entre comillas simples (*string*).

Advertencia: no utilizar diéresis ni acentos ni en el nombre del módulo ni en el título

1.3.2. Declaraciones

En esta sección, que debe ir encabezada por la palabra clave **declarations**, se especifican las señales usadas en el diseño. Además se pueden definir constantes, estados, y opcionalmente el dispositivo en el que implementar dicho diseño. Las señales de entrada y salida del módulo se declararán mediante la palabra clave **pin**, y algún nodo interno de interés (esto es opcional) mediante **node**. A cualquier señal se le pueden añadir ciertas propiedades o atributos (enumerados más adelante) mediante la palabra **istype**.

1.3.3. Descripción lógica

La descripción de un circuito electrónico mediante ABEL puede hacerse mediante:

- Ecuaciones lógicas (utilizando operadores lógicos) (ver ejemplo 1) o algebraicas (utilizando when-then-else) (ver ejemplo 4)
- Diagramas de estado (utilizando if-then-else) (ver ejemplo 3)
- Tablas de verdad (ver ejemplo 2)

siendo esta sección encabezada en cada uno de los anteriores casos por la palabra clave:

1. **equations**
2. **state_diagram**
3. **truth_table**

cuyos tipos de declaraciones son:

1. (lógicas) Ejemplos:

SUM = (A & !B) # (!A & B) ;

A0 := EN & !D1 & D3 & !D7;

(algebraicas) WHEN condición THEN asignación;

ELSE ecuación;

o

WHEN condición THEN ecuación; Ejemplos:

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

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

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

Se pueden utilizar corchetes {} para agrupar instrucciones en un único bloque.

2. Descripción de la máquina de estados mediante transiciones condicionadas

If-Then-Else :

IF expresión THEN expresión o nombre de un estado

[ELSE expresión o nombre de un estado] ;

y que puede venir acompañado de una declaración "**with**":

```
IF expresión THEN expresión o nombre de un estado WITH ecuación  
[ ecuación ] . . . ;
```

Ejemplos:

- a. if X#Y==1 then S1 with Z=1 else S2;
- b. if X&!Y then S3 with Z=X#Y else S2 with Z=Y;
- c. if RST then S2 with { OUT1 := 1;
Error-Adrs := ADDRESS; }
else if (ADDRESS <= ^hC101)
then S4
else S1;
- d. state S1: if (A & B) then S2 with TK = 1
else S0 with TK = 0 ;

En lugar de la transición "If-then-else" también se pueden utilizar la "**Goto**" o la "**Case**" y que pueden estar acompañados por un "**With**":

```
GOTO nombre de estado WITH ecuación ;
```

Ejemplo:

```
Goto F With RS=1;
```

```
CASE expresión : expresión o nombre de estado ;
```

```
[ expresión : expresión o nombre de estado ; ] ...
```

```
ENDCASE ;
```

Ejemplo:

```
State S0:
```

```
case ( A == 0 ) : S1;  
      ( A == 1 ) : S0;  
endcase;
```

- 3. TRUTH_TABLE (identificadores_de_entradas -> identificadores_de_salidas)
entradas -> salidas ;
o
TRUTH_TABLE (identificadores_de_entradas :> idents_sal_secuenciales)
entradas :> salidas_secuenciales ;
or
TRUTH_TABLE
(ident_entradas :> ident_sal_sec -> ident_salidas)
entradas :> sal_sec -> salidas ;

A continuación se reproduce un ejemplo de sistema secuencial descrito con tabla de verdad donde las salidas toman los nuevos valores en el momento en que el reloj cambia:

MODULE te08

TITLE 'taula de veritat per secuencial taula4'

```
rellotge pin;  
in1,in0 pin;  
out1,out0 pin istype 'reg';  
inputs=[in1,in0];  
outputs=[out1,out0];  
r=.C.;
```

equations

```
outputs.clk=rellotge;
```

TRUTH_TABLE (inputs :> outputs)

```
0 :> 1 ;  
1 :> 2 ;  
2 :> 3 ;  
3 :> 0 ;
```

test_vectors ([rellotge,inputs] -> outputs)

```
[r,0] -> 1 ;  
[r,1] -> 2 ;  
[r,2] -> 3 ;  
[r,3] -> 0 ;
```

END

En la segunda parte de este manual se presentan más ejemplos de cada caso.

1.3.4. Vectores de test

Se utilizan para simular y verificar el correcto funcionamiento del diseño realizado. No es más que la enumeración de los vectores de entrada a simular, y las salidas válidas correspondientes.

1.3.5. End

Un módulo siempre debe acabar con la palabra **End**.

1.4 ATRIBUTOS Y EXTENSIONES DE LAS SEÑALES

Una vez comentadas brevemente las diferentes secciones que componen un fichero ABEL, en la **figura 1** se muestra un nuevo ejemplo. Utilizando dicho ejemplo como base, a continuación se realizan algunos comentarios:

La declaración **istype**, a continuación de la palabra clave **pin** o **node**, define **atributos** (características) de señales. Algunos de estos atributos son:

| | |
|----------|---|
| 'buffer' | No inversor |
| 'com' | Salida combinacional |
| 'invert' | Inversor |
| 'reg' | Salida secuencial (será salida de un flip-flop) |
| 'reg_d' | Idem, y el FF será tipo D |
| 'reg_jk' | Idem FF JK |
| 'reg_sr' | Idem FF SR |
| 'reg_t' | Idem FF T |

En la descripción del diseño pueden utilizarse, para una mejor descripción del circuito, **extensiones** en las señales, tal como en **out.clk** del ejemplo mencionado. Las extensiones más importantes son:

| Extensión | Descripción |
|------------------|--|
| .ACLR | Reset asíncrono |
| .ASET | Preset asíncrono |
| .CLK | Entrada secuencial |
| .CLR | Reset síncrono |
| .OE | Habilitación de la salida (<i>Output Enable</i>) |
| .SET | Preset síncrono |
| .FB | Retroalimentación (<i>FeedBack</i>) |

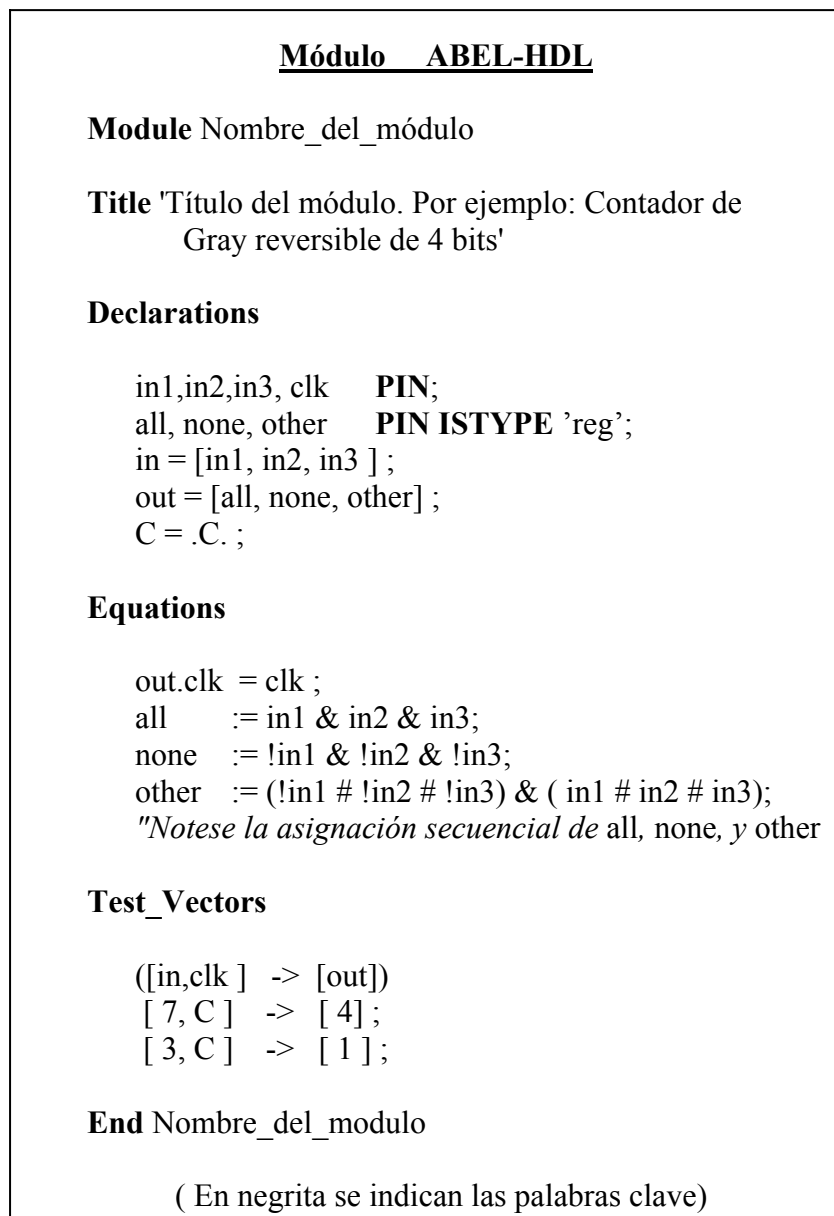


Figura 1. Esquema de un fichero ABEL.

En la figura 2 puede verse un circuito del cual se muestra a continuación su correspondiente fichero **ABEL**, en el que se incluyen los vectores de test:

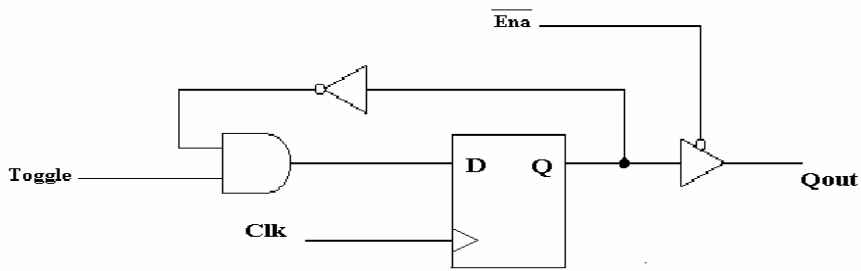


Figura 2.

```

module FF-T_D
title 'Flip-flop tipo T usando uno tipo D'

    Clk, Toggle, Ena pin;
    Qout pin istype 'reg';

equations

    Qout := !Qout.FB & Toggle;
    Qout.CLK = Clk;
    Qout.OE = !Ena;

test_vectors ([Clk,Ena,Toggle] -> [Qout])

    [.c., 0 , 0 ] -> 0;
    [.c., 0 , 1 ] -> 1;
    [.c., 0 , 1 ] -> 0;
    [.c., 0 , 1 ] -> 1;
    [.c., 0 , 1 ] -> 0;
    [.c., 1 , 1 ] -> .Z.;
    [ 0 , 0 , 0 ] -> 1;
    [.c., 1 , 1 ] -> .Z.;
    [.c., 0 , 1 ] -> 1;

end

```

1.5.- MÁQUINAS DE ESTADO

Un caso especial de nomenclatura en la descripción ABEL son las máquinas de estados finitos (FSM). Una máquina de estados es un circuito digital que recorre una secuencia de estados predeterminada y se utiliza usualmente para el control lógico secuencial.

En ABEL cada estado de la máquina se debe identificar con un nombre, y en él se describe la salida, y las transiciones que se producen desde él a otros estados. El formato de la descripción de una máquina de estados en ABEL tiene el siguiente aspecto:

Module Maquina_de_estados
Title 'Ejemplo de maquina de estados'

a, b, c **pin**;

"Además de declarar las señales de entrada y salida del sistema, en el caso de las FSM, se declaran los bits y el registro de estado. Por ejemplo, en una de 4 o menos estados:

q1, q0 **pin** **istype** 'reg';

sreg = [q1,q0];

"También se da nombre a los estados, y se les asigna su codificación. Por ejemplo, para la misma máquina, y suponiendo 3 estados (A, B y C):

A = 0; B = 1; C = 2;

equation

"Ecuaciones entre variables de entrada y salida para definir el sistema. Respecto a la máquina de estados a describir, esta sección del fichero se suele utilizar para declarar las relaciones de las señales de control (clock, reset, enable, etc) del registro de estado. Por ejemplo:

sreg.clk = clock;

sreg.clr = !reset; *"(reset definido activo bajo)*

*"También se pueden definir las transiciones entre estados mediante ecuaciones (ver el cuarto ejemplo), si bien es más usual hacerlo mediante el **state_diagram**, como se muestra a continuación:*

state_diagram 'sreg'

*"Aquí se describe propiamente la máquina de estados. Se hace mediante expresiones del tipo IF-THEN-ELSE y IF-THEN-WITH-ELSE-WITH, para transiciones condicionales, o mediante el término **GOTO**, utilizado para especificar transiciones incondicionales. El funcionamiento de estas estructuras lógicas puede obtenerse analizando el ejemplo que se muestra más adelante.*

State A:

IF-THEN-WITH;

ELSE-WITH;

state B:

IF-THEN-WITH;

ELSE-WITH;

·
·
·

Test_vectors

"Se describen los vectores de entradas y de salida con los que realizar la simulación, y comparar el resultado, tal y como se ha mostrado anteriormente.

End

2.- EJEMPLOS DE DESCRIPCIÓN CIRCUITAL MEDIANTE ABEL-HDL

Se presentan 4 ejemplos de descripción mediante ABEL, dos sistemas combinacionales (uno descrito mediante ecuaciones algebraicas y otro mediante tabla de verdad), y dos sistemas secuenciales (uno definido a partir del diagrama de estados y otro de ecuaciones lógicas).

EJEMPLO 1

Sistema combinatorial (definido mediante ecuaciones algebraicas). Supongamos el sistema combinatorial de cuatro entradas (a, b, c, d) y dos salidas (u, v) que responden a las siguientes ecuaciones algebraicas:

$$u = abc' + ad + bc'd' + acd'$$

$$v = ab + a'd + bd + c'd + ac$$

y al que denominamos **combi**. Su descripción en ABEL es:

Module combi

Declarations

a, b, c, d **pin**;

u, v **pin** **istype 'com'**;

"Se impone que sean salidas combinatoriales"

H = 1;

L = 0;

X = .X.;

equations

u = a&b!c#a&d#b!c!d#a&c&d;

v = a&b#!a&d#b&d#!c&d#a&c;

test_vectors

([a,b,c,d] -> [u,v])

[L,L,L,L] -> [L,L];

[X,X,L,H] -> [L,H];

[L,L,H,L] -> [L,L];

[X,X,H,H] -> [L,H];

[X,H,L,L] -> [H,L];

[L,H,H,L] -> [L,L];

[L,X,X,H] -> [L,H];

[H,L,L,L] -> [L,L];

[H,L,L,H] -> [H,H];

[H,L,H,L] -> [H,H];

[H,X,X,H] -> [H,H];

[H,H,L,X] -> [H,H];

[H,X,H,L] -> [H,H];

[H,H,H,H] -> [H,H];

end combi

EJEMPLO 2

Sistema combinacional (definido mediante tabla de verdad). Supongamos el sistema combinacional de cuatro entradas (i3, i2, i1, i0) y cuatro salidas (f3, f2, f1, f0) del cual conocemos su tabla de verdad. La palabra clave para introducir el comportamiento de un sistema combinacional mediante su tabla de verdad es **truth_table**. En caso de no especificar las salidas de algunas entradas, se ha de indicar en la declaración de las señales de salida que tengan inespecificaciones como se debe realizar la optimización. Para ello se utilizan las palabras **dc**, **neg**, o **pos**. La primera indica que se debe optimizar asignando a cada salida no especificada un 1 o un 0 de forma que la simplificación algebraica sea máxima (mínimo número de términos producto/suma, tamaño de cada término mínimo, etc.). La segunda, **neg**, indica que las inespecificaciones sean asignadas a 0, y la última, **pos**, que sean asignadas a 1.

```
module combi2
```

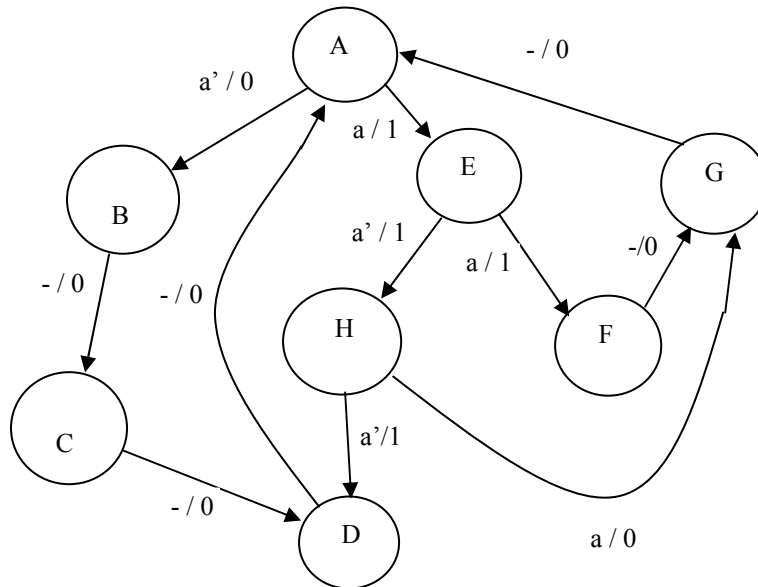
```
    i3,i2,i1,i0 pin;  
    f3,f2,f1,f0 pin istype 'dc,com';
```

```
    truth_table ([i3,i2,i1,i0] -> [f3,f2,f1,f0])  
                [ 0, 0, 0, 0] -> [ 0, 0, 0, 1];  
                [ 0, 0, 0, 1] -> [ 0, 0, 1, 1];  
                [ 0, 0, 1, 1] -> [ 0, 1, 1, 1];  
                [ 0, 1, 1, 1] -> [ 1, 1, 1, 1];  
                [ 1, 1, 1, 1] -> [ 1, 1, 1, 0];  
                [ 1, 1, 1, 0] -> [ 1, 1, 0, 0];  
                [ 1, 1, 0, 0] -> [ 1, 0, 0, 0];  
                [ 1, 0, 0, 0] -> [ 0, 0, 0, 0];
```

```
end combi2
```

EJEMPLO 3

Sistema secuencial (definido a partir de su diagrama de estados). Sea el sistema secuencial, al que denominamos *seqesta*, y que viene definido por el siguiente diagrama de transiciones de estado:



El sistema anterior tiene 8 estados, por tanto puede implementarse con 3 biestables; una entrada **a** y una salida que denominamos **z**. Se puede observar que se trata de un sistema de Mealy. La señal de reloj la llamaremos **clock** y el sistema tendrá una entrada de reset asíncrono que llamaremos **reset** y que lleva al sistema al estado **A**. Para implementarlo impondremos biestables tipo J-K y seleccionamos como codificación de estados la siguiente:

| | Q2 | Q1 | Q0 |
|---|----|----|----|
| A | 0 | 0 | 0 |
| B | 0 | 0 | 1 |
| C | 0 | 1 | 0 |
| D | 0 | 1 | 1 |
| E | 1 | 0 | 0 |
| F | 1 | 0 | 1 |
| G | 1 | 1 | 0 |
| H | 1 | 1 | 1 |

Fichero ABEL

module seqesta

```

clock, a, reset  pin;
z                pin istype 'com';
Q2,Q1,Q0        pin istype 'reg_jk';
sreg           = [Q2,Q1,Q0];
A = 0; B = 1; C = 2; D = 3; E = 4; F = 5; G = 6; H = 7;
Cl, X = .C., .X.;

```

equations

```
sreg.clk = clock;  
sreg.ar = reset;
```

state_diagram sreg

```
state A:  
IF (a) THEN E WITH z = 1;  
ELSE B WITH z = 0;
```

```
state B:  
GOTO C WITH z = 0;
```

```
state C:  
GOTO D WITH z = 0;
```

```
state D:  
GOTO A WITH z = 0;
```

```
state E:  
IF (!a) THEN H WITH z = 1;  
ELSE F WITH z = 1;  
"Este estado también se podría describir de la siguiente forma:"  
" state E:  
" z = 1;  
" IF (!a) THEN H ELSE F ;
```

```
state F:  
z = 0;  
GOTO G;  
"O lo que sería equivalente:"  
" GOTO G WITH z = 0;
```

```
state G :  
GOTO A WITH z = 0;
```

```
state H:  
IF (!a) THEN D WITH z = 1;  
ELSE G WITH z = 0;
```

test_vectors

```
([clock, reset, a] -> [sreg, z ])  
[ Cl, 1, X ] -> [ A, X ];  
[ Cl, 0, 1 ] -> [ E, 1 ];  
[ Cl, 0, 0 ] -> [ H, 1 ];  
[ Cl, 0, 1 ] -> [ G, 0 ];  
[ Cl, 0, X ] -> [ A, 0 ];  
[ Cl, 0, 0 ] -> [ B, 0 ];  
[ Cl, 0, X ] -> [ C, 0 ];
```

```

[ Cl, 0, X ] -> [ D, 0 ];
[ Cl, 0, X ] -> [ A, 0 ];
[ Cl, 0, 0 ] -> [ B, 0 ];
[ Cl, 0, X ] -> [ C, 0 ];
[ Cl, 0, 0 ] -> [ D, 0 ];
[ Cl, 0, X ] -> [ A, 0 ];
[ Cl, 0, 1 ] -> [ E, 1 ];
[ Cl, 0, 0 ] -> [ H, 1 ];
[ Cl, 0, 1 ] -> [ G, 0 ];

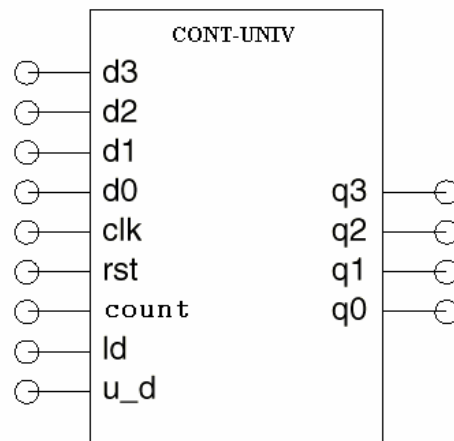
```

end

EJEMPLO 4

Sistema secuencial (definido mediante ecuaciones lógicas): Contador universal de 4 bits

En el diseño de este sistema secuencial utilizaremos las ecuaciones que interrelacionan la salida con la entrada, las cuales se muestran respectivamente a la izquierda y derecha del bloque de la siguiente figura. El significado de las mismas está sobreentendido por el funcionamiento del sistema. El reset (**rst**)del contador es asíncrono.



```

module cont_univ

```

```

title 'contador universal de 4bits con carga en paralelo';

```

```

"Constantes

```

```

    X, C, Z = .X., .C., .Z. ;

```

```

"Entradas
  d3 .. d0   pin ;           "Entrada de datos
  clk       pin ;           "Entrada de reloj
  rst       pin ;           "Reset asíncrono
  count     pin ;           "Habilitación de conteo
  ld        pin ;           "Habilitación de carga
  u_d       pin ;           "Conteo arriba/abajo

```

```

"Salidas

  q3 .. q0   pin istype 'reg'; "Salidas contador

```

"Sets

```

  data = [d3..d0];           "Conjuntos de datos de entrada
  count = [q3..q0];         "Conjunto de datos de salida

```

"Modo de actuación del contador

```

  MODE = [count, ld, u_d];   "Pines de control de funcionamiento.
  LOAD = (MODE == [ X , 1, X ]); "Los distintos modos de funcionamiento se
  HOLD = (MODE == [ 0 , 0, X ]); "definen por los valores aplicados a los pines .
  UP    = (MODE == [ 1 , 0, 1 ]); "El nombre simbólico puede igualarse a
  DOWN = (MODE == [ 1 , 0, 0 ]); "un valor.

```

"Si [count, ld, u_d] = [1, 0, 1] quiere decir que ha de contar hacia arriba, con lo que UP = 1, y el resto de variables valdrán 0. Nótese que cargar tiene precedencia respecto a contar.

Equations

```

when LOAD then count := data           "El contador carga los datos de entrada
else when UP then count := count + 1   "El contador cuenta hacia arriba
else when DOWN then count := count - 1 "El contador cuenta hacia abajo
else when HOLD then count := count ;   "El contador se mantiene; no cuenta

```

*"Nótese como la asignación se hace mediante := , ya que count es la salida de los biestables q3..q0 (son señales definidas como 'reg')
"La estructura WHEN-THEN-ELSE es equivalente a la IF-THEN-ELSE utilizada
"con el comando **state_reg**, pero en el comando **equations**, y válida tanto en sistemas
"combinacionales como secuenciales.*

```

count.clk = clk;           "Entrada de reloj al contador (sistema)
count.ar = rst;           "Entrada de reset al contador (sistema)

```

End