



UNIVERSITAT  
ROVIRA I VIRGILI

Departament d'Enginyeria Electrònica Elèctrica i Automàtica

## **Implementación de una ALU de 8 bits en lenguaje VHDL**

AUTORS: Jordi Pons Albalat.  
DIRECTORS: Enric Cantó Navarro.

DATA: Febrer / 2002.

<b>1. Introducción .....</b>	<b>4</b>
<b>2. El lenguaje VHDL .....</b>	<b>5</b>
2.1 VHDL describe estructura y comportamiento .....	6
2.1.1 Ejemplo básico y estilos de descripción VHDL .....	7
2.1.2 Descripción algorítmica .....	7
2.1.3 Descripción flujo de datos .....	9
2.1.4 Descripción estructural .....	10
2.2 Unidades básicas de diseño .....	14
2.2.1 Como se declara una entidad .....	14
2.2.2 Como se declara una arquitectura .....	19
2.3 VHDL para síntesis .....	22
2.4 Construcciones básicas .....	23
2.5 Objetos .....	24
2.6 Identificadores .....	24
2.7 Palabras reservadas .....	25
2.8 Símbolos especiales .....	26
<b>3. Explicación ALU de 8 bits.....</b>	<b>27</b>
3.1 Explicación paso a paso .....	27
3.2 Esquema interno bloque AND/OR .....	30
3.3 Esquema interno bloque suma_resta .....	32
3.4 Esquema interno bloque multiplexor .....	34
3.5 Tabla y simplificaciones .....	35
3.5.1 Karnaught CLRA .....	35
3.5.2 Karnaught INVB .....	36
3.5.3 Karnaught MUX .....	37
3.5.4 Karnaught CIN .....	38
3.5.5 Karnaught Funcio .....	39
<b>4. Guía del programa eProduct Designer .....</b>	<b>41</b>
4.1 Funcionamiento del programa paso a paso .....	41

4.1.1 Compilar .....	41
4.1.2 Simulación .....	53
4.1.3 Sintetizar .....	63
<b>5. Conclusiones .....</b>	<b>72</b>
<b>Anexos .....</b>	<b>73</b>
Anexo 1. Programa ALU .....	73
Anexo 2. Programa de simulación .....	80
Anexo 3. Fichero de retardos después de la síntesis .....	82
Anexo 4. Disposición de las patas de la FPGA .....	83
Anexo 5. Gráficas de salida .....	84
<b>Bibliografía .....</b>	<b>88</b>

## 1 Introducción

En este proyecto lo que se trata es de hacer una unidad aritmético lógica de 8 bits, es decir lo que solemos conocer con el nombre de ‘**ALU**’, y la implementaremos a partir del lenguaje VHDL, es decir diseñaremos un algoritmo que implemente una ALU. En el algoritmo se ha hecho de forma que todo se tiene que hacer a través de puestas lógicas es decir un lenguaje puramente sintetizable ya que sólo utilizaremos puertas. Esta ALU tiene que hacer las siguientes operaciones o funciones:

- Inversión del registro B
- Suma
- Resta
- AND
- OR

Cuando tengamos el código VHDL hecho tendremos que compilarlo, simularlo, y posteriormente tendremos que sintetizarlo. Para hacer todo esto que he comentado anteriormente utilizaremos el programa **eProduct Designer** y con él lo haremos todo.

Este programa se ha hecho con un lenguaje RTL es decir transferencia de registros. En el capítulo 2 se explicará como hacer un programa mediante RTL, y aparte también se explicará de forma introductoria como se pueden hacer programas en VHDL mediante otro tipo de descripciones.

En el capítulo 3 explicaremos los pasos que he seguido para hacer la ALU, primero mostraré los esquemas de como se ha hecho paso a paso, y en el apartado de anexos se mostrará el código.

En el capítulo 4 se explicará como hacer funcionar el programa **eProduct Designer** mediante un tutorial que te irá diciendo como compilar, simular y sintetizar un programa VHDL.

En el capítulo 5 explicaremos las conclusiones a que hemos llegado después de hacer el proyecto, y las cosas que podríamos mejorar.

Por último al final pondremos los anexos, allí pondremos los programas VHDL que hemos hecho para hacer la ALU. En definitiva pondremos dos programas VHDL, uno que será la ALU y el otro que será el fichero de simulación con el cual verificamos que el programa de la ALU funciona bien. A parte también pondremos ahí los ficheros que encontremos interesantes poner en el apartado de anexos, como el fichero de retardos que se obtiene después de realizar la síntesis del programa de VHDL.

Este proyecto será una práctica para los alumnos de sistemas digitales para que vean como se programa en VHDL y también para que vean como funciona este programa nuevo el **eProduct Designer**. Antes se hacían las prácticas con el programa **SYNARIO** y ahora se pasará a utilizar este nuevo programa y a utilizar el lenguaje VHDL que como se verá es un lenguaje bastante sencillo de hacer.

Ahora pasaremos a explicar como se hace un programa VHDL y como usar el programa para poder hacer nuestras pruebas.

## 2. El lenguaje VHDL

El significado de las siglas VHDL es VHSIC(*Very High Speed Integrated Circuit Hardware Description Language*), es decir, lenguaje de descripción hardware de circuitos integrados de muy alta velocidad. VHDL es un lenguaje de descripción y modelado diseñado para describir, en una forma en que los humanos y las máquinas puedan leer y entender la funcionalidad y la organización de sistemas hardware digitales, placas de circuitos y componentes.

VHDL fue desarrollado como un lenguaje para el modelado y simulación lógica dirigida por eventos de sistemas digitales, y actualmente se utiliza también para la síntesis automática de circuitos. El VHDL fue desarrollado de forma muy parecida al ADA debido a que el ADA fue también propuesto como un lenguaje que tuviera estructuras y elementos sintácticos que permitieran la programación de cualquier sistema hardware sin limitación de la arquitectura. El ADA tenía una orientación hacia sistemas en tiempo real y al hardware en general, por lo que se lo escogió como modelo para desarrollar el VHDL.

VHDL es un lenguaje con una sintaxis amplia y flexible que permite el modelado estructural, en flujo de datos y de comportamiento hardware. VHDL permite el modelado preciso, en distintos estilos, del comportamiento de un sistema digital conocido y el desarrollo de modelos de simulación.

Uno de los objetivos del lenguaje VHDL es el modelado. Modelado es el desarrollo de un modelo para simulación de un circuito o sistema previamente implementado cuyo comportamiento, por tanto, se conoce. El objetivo del modelado es la simulación.

Otro de los usos de este lenguaje es la síntesis automática de circuitos. En el proceso de síntesis se parte de una especificación de entrada con un determinado nivel de abstracción y se llega a una implementación más detallada, menos abstracta. Por tanto, la síntesis es una tarea vertical entre niveles de abstracción, del nivel mas alto en la jerarquía de diseño hacia el más bajo nivel de la jerarquía.

El VHDL es un lenguaje que fue diseñado inicialmente para ser usado en el modelado de sistemas digitales. Es por esta razón que su utilización en síntesis no es inmediata, aunque lo cierto es que la sofisticación de las actuales herramientas de síntesis es tal que permiten implementar diseños especificados en un alto nivel de abstracción.

La síntesis a partir de VHDL constituye hoy en día una de las principales aplicaciones del lenguaje con una gran demanda de uso. Las herramientas de síntesis basadas en el lenguaje permiten en la actualidad ganancias importantes en la productividad de diseño.

Algunas ventajas del uso de VHDL para la descripción hardware son:

- ✓ VHDL permite diseñar, modelar y comprobar un sistema desde un alto nivel de abstracción bajando hasta el nivel de definición estructural de puertas.

- ✓ Circuitos descritos utilizando VHDL, siguiendo unas guías para síntesis, pueden ser utilizados por diversas herramientas de síntesis para crear e implementar circuitos.
- ✓ Los módulos creados en VHDL pueden utilizarse en diferentes diseños, lo que permite la reutilización del código. Además, la misma descripción puede utilizarse para diferentes tecnologías sin tener que rediseñar todo el circuitos.
- ✓ Al estar basado en un estándar(IEEE Std 1076-1987, IEEE Std 1076-1993) los ingenieros de toda la industria e diseño pueden usar este lenguaje para minimizar errores de comunicación y problemas de compatibilidad.
- ✓ VHDL permite diseño *Top-Down*, esto es, describir(modelar) el comportamiento de los bloques de alto nivel, analizarlos(simularlos) y refinar la funcionalidad en alto nivel requerida antes de llegar a niveles más bajos de abstracción de la implementación del diseño.
- ✓ Modularidad: VHDL permite dividir o descomponer un diseño hardware y su descripción VHDL en unidades más pequeñas.

## 2.1 VHDL describe estructura y comportamiento

Existen dos formas de describir un circuito. Por un lado se puede describir un circuito indicando los diferentes componentes que lo forman y su interconexión, de esta manera se tiene especificado un circuito y se sabe cómo funciona. Esta es la forma habitual en que se han venido describiendo circuitos, siendo las herramientas utilizadas para ello las de captura de esquemas y las de descripción *netlist*.

La segunda forma consiste en describir un circuito indicando lo que hace o cómo funciona, es decir, describiendo su comportamiento. Naturalmente esta forma de describir un circuito es mucho mejor para un diseñador puesto que lo que realmente le interesa es el funcionamiento del circuito más que sus componentes. Por otro lado, al encontrarse lejos de lo que es realmente un circuito, se pueden plantear algunos problemas a la hora de implementarlo a partir de la descripción de su comportamiento.

El VHDL va a ser interesante puesto que va permitir los dos tipos de descripciones:

**Estructura:** VHDL puede ser usado como un lenguaje de *Netlist* normal y corriente donde se especifican por un lado los componentes del sistema y por otro sus interconexiones.

**Comportamiento:** VHDL también se puede utilizar para la descripción comportamental o funcional de un circuito. Esto es lo que lo distingue de un lenguaje de *Netlist*. Sin necesidad de conocer la estructura interna de un circuito es posible describirlo explicando su funcionalidad. Esto es especialmente útil en simulación, ya que permite simular un sistema sin conocer su estructura interna. Así, este tipo de descripción se está volviendo cada día más importante porque las actuales herramientas de síntesis permiten la creación automática de circuitos a partir de una descripción de su funcionamiento.

Muchas veces la descripción comportamental se divide a su vez en dos, dependiendo del nivel de abstracción y del modo en que se ejecutan las instrucciones. Estas dos formas comportamentales de describir circuitos son la de *flujo de datos* y la *algorítmica*.

### 2.1.1 Ejemplo básico y estilos de descripción en VHDL

VHDL presenta tres estilos de descripción de circuitos dependiendo del nivel de abstracción. El menos abstracto es una descripción puramente estructural. Los otros dos estilos representan una descripción comportamental o funcional, y la diferencia viene de la utilización o no de la ejecución serie.

Ahora mediante un ejemplo de un multiplexor explicaré la forma en que se escriben descripciones en VHDL, se van a mostrar tres estilos de descripción.

**Ejemplo. Describir en VHDL un circuito que multiplexe dos líneas de entrada como el de la siguiente figura. Figura 1**

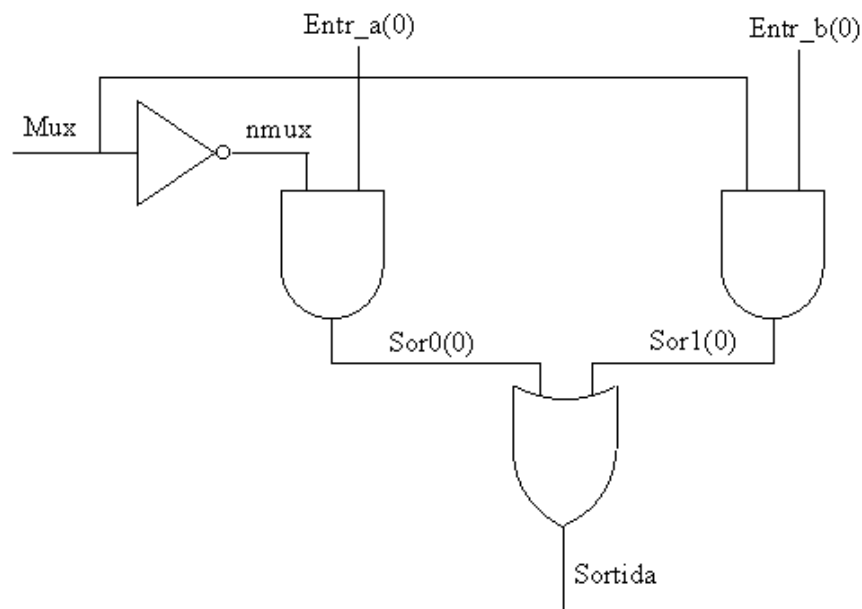


Figura 1. Multiplexor

### 2.1.2 Descripción algorítmica

Lo se va a realizar a continuación es la descripción comportamental algorítmica del circuito de la figura anterior, después se realizará la transferencia entre registros, que sigue siendo comportamental, y por último se verá la descripción estructural mostrando así las diferencia.

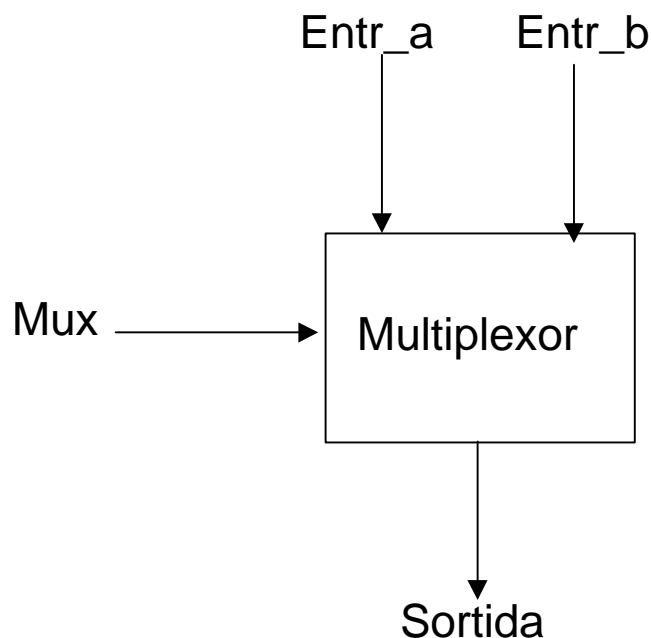
La sintaxis del VHDL no es sensible a mayúsculas o minúsculas, por lo que se puede escribir como se prefiera. A lo largo de las explicaciones se pondrán siempre las palabras clave del lenguaje en mayúsculas para distinguirlas de las variables y otros elementos. Esto no significa que durante la descripción de diseños se tenga que hacer así,

de hecho resulta más rápido escribir siempre en minúsculas. Se ha hecho así en todos los ejemplos para mayor claridad del código.

En primer lugar, sea el tipo de descripción que sea, hay que definir el símbolo o ENTIDAD del circuito. En efecto, lo primero es definir las entradas y salidas del circuito, es decir, la caja negra que lo define. Se llama entidad porque en la sintaxis de VHDL esta parte se declara con la palabra clave ENTITY. Esta definición de entidad, que suele ser la primera parte de toda descripción VHDL, se expone a continuación. En esta parte pondremos un ejemplo de lo que es una

entidad para poder hacer una descripción de cada tipo especificado, después en temas posteriores definiremos con claridad que es una ENTIDAD y también una arquitectura, y para que sirven.

Primero antes de pasar a la declaración pondremos un dibujo de lo que es la caja negra es decir la ENTIDAD de este multiplexor, que será la siguiente. **Figura 2**



**Figura 2.** Bloque multiplexor

Ahora declararemos la ENTIDAD con lenguaje VHDL es decir las entradas y salidas que tenemos.

```
ENTITY multiplexor IS --Bloque multiplexor

    Port( entr_a:in std_logic  --Entrada a
          entr_b:in std_logic  --Entrada b
          mux :in std_logic;   --Seleccionamos la entrada
          sortida: out std_logic --Salida

END multiplexor;
```



Esta porción del lenguaje indica que la entidad *multiplexor* (que es el nombre que se le ha dado al circuito) tiene tres entradas de tipo *bit* y una salida también del tipo *bit*.

Los tipos de las entradas y salidas se verán más adelante. El tipo *bit* simplemente indica una línea que puede tomar valores '0' o '1'.

Ahora en lo que llamamos ARQUITECTURA definiremos lo que hace esta caja negra que llamamos ENTIDAD es decir definiremos la función que hace, diremos que hacen estas entradas y salidas que hemos definido antes. En la ARQUITECTURA pondremos como se hace el multiplexor. Se muestra a continuación la descripción comportamental del multiplexor:

```
ARCHITECTURE comportamental OF muxtplexor IS
BEGIN
  IF (mux='0') THEN
    Sortida<=entr_a;
  ELSE
    Sortida<=entr_b;
  END IF;
END comportamental
```

Esta descripción comportamental es muy sencilla de entender, ya que sigue una estructura parecida a los lenguajes de programación convencionales. Es por lo que se dice que se trata de una descripción comportamental algorítmica. Lo que se está indicando es simplemente que si la señal *mux* es cero, entonces la entrada es *entr\_a*, y si *mux* es uno, entonces la salida es la entrada *entr\_b*. Esta forma tan sencilla de describir el circuito permite a ciertas herramientas sintetizar el diseño a partir de una descripción comportamental como la que se acaba de mostrar. La diferencia con un *Netlist* es directa: en una descripción comportamental no se están indicando ni los componentes ni sus interconexiones, sino simplemente lo que hace, es decir, su comportamiento o funcionamiento.

### 2.1.3 Descripción flujo de datos

La descripción anterior era puramente comportamental, de manera que con una secuencia sencilla de instrucciones se podría describir el circuito. Naturalmente, a veces resulta más interesante describir el circuito de forma que esté más cercano a una posible realización física del mismo. En este sentido VHDL posee una forma de describir circuitos que además permite la paralelización de instrucciones, y que se encuentra más cercana a una descripción estructural del mismo, siendo todavía una descripción funcional. A continuación se muestra una descripción de *flujo de datos o de transferencia entre registros (RTL)*.

ARCHITECTURE flujo OF multiplexor IS

```
SIGNAL nmux: std_logic;  
SIGNAL sor0: std_logic;  
SIGNAL sor1: std_logic;
```

BEGIN

```
nmux<=not mux;  
sor0<=nmux and entr_a;  
sor1<=mux and entr_b;  
sortida<= sor0 or sor1;
```

END multiplexor;

#### 2.1.4 Descripción estructural

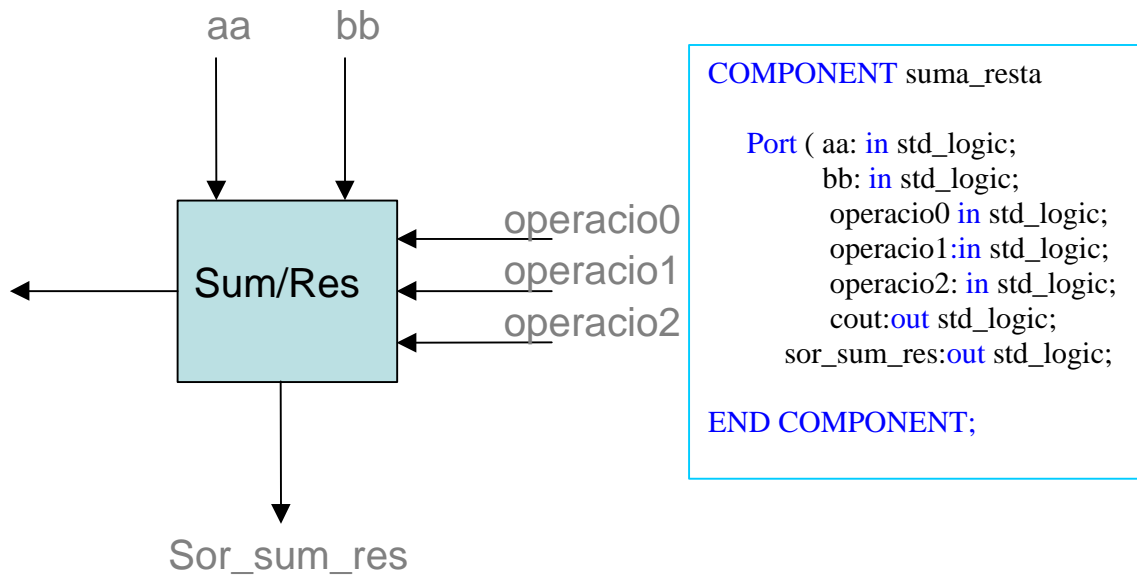
Aunque no es la característica más interesante del VHDL, también permite ser usando como *Netlist* o lenguaje de descripción de estructura. En este caso esta estructura también estaría indicada dentro de un bloque de arquitectura, aunque la sintaxis interna es completamente diferente.

Ahora pondremos un ejemplo para que se vea más claro lo que es una descripción estructural, definiremos una ALU. Esto lo interpretaremos como una caja donde dentro habrá un bloque que sume y reste, otro bloque que haga las operaciones and y or, y por último otro bloque que sea un multiplexor que nos escoja la salida que queremos en función de si en las operaciones **Op0**, **Op1**, **Op2** le hemos dicho que función nos tiene que hacer si suma o resta, o and o or. Para este ejemplo tendremos dos registros de entrada que serán el registro **a** y el registro **b** que serán con los que haremos las operaciones, tendremos una salida del carry llamada **Cout** y otra salida que será el **resultado** de la operación hecha. Aparte como se verá en la unión de todos los bloques crearemos unas señales para unir las diferentes entradas y salidas que serán **x** e **y**.

Cada uno de los tres bloques, haremos una descripción por componentes, es decir tendremos tres componentes, que serán:

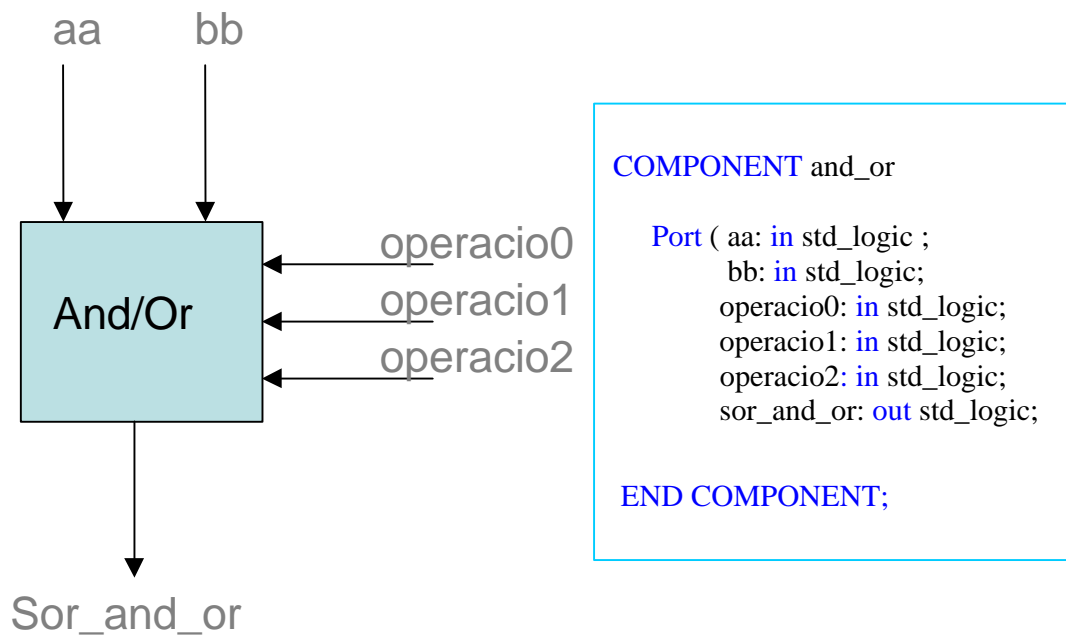
1. Componente Suma/Resta
2. Componente And/Or
3. Componente Multiplexor

1. Primero definiremos los bloques por separado, y luego los uniremos, empezaremos por el bloque de sumar y restar. **Figura 3**



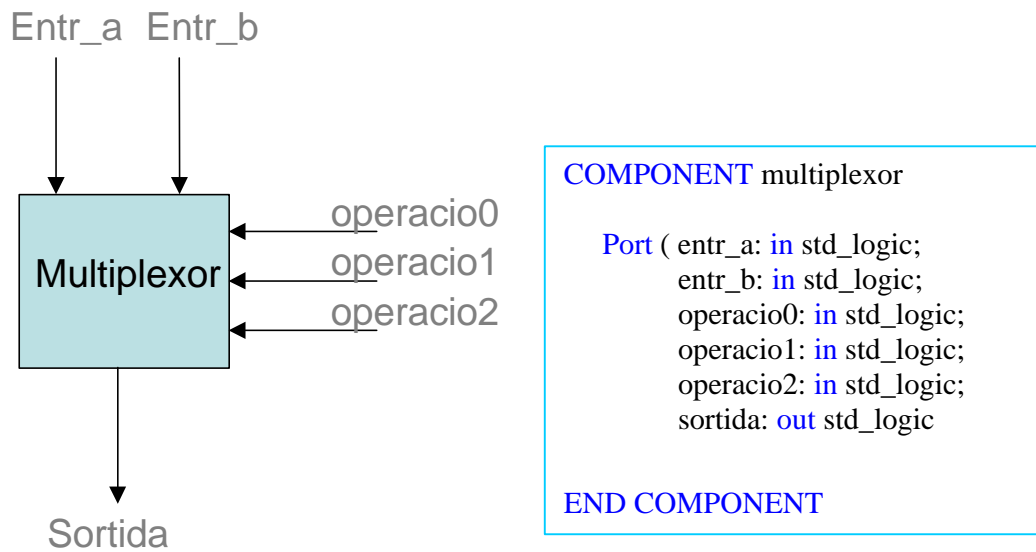
**Figura 3.** Componente suma\_resta

2. Ahora haremos el bloque que nos hará la función AND y la función OR. **Figura 4**



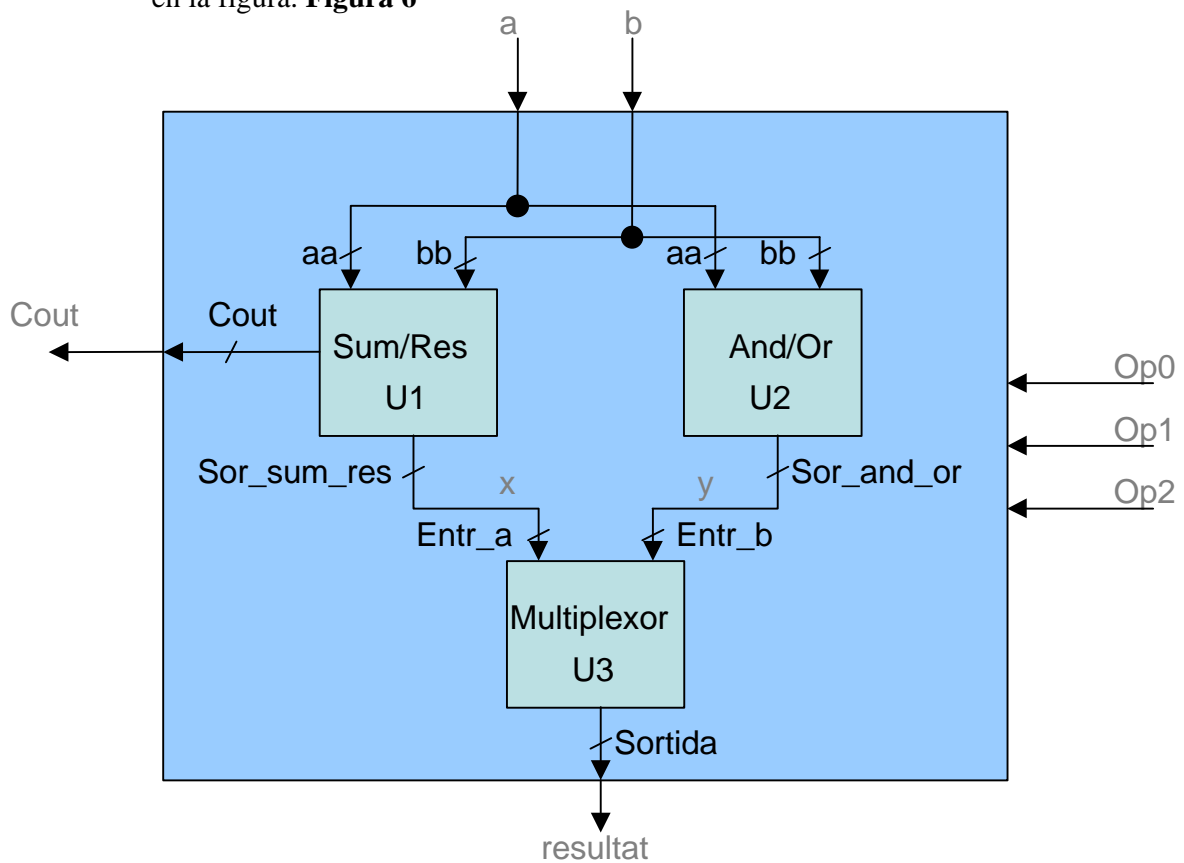
**Figura 4.** Componente AND/OR

3. Por último haremos el bloque multiplexor que será el siguiente. **Figura 5**



**Figura 5.** Componente multiplexor

4. Ahora uniremos todos los bloques y nos quedará la ALU tal y como se puede ver en la figura. **Figura 6**



**Figura 6.** ALU

Seguidamente haremos el programa para unir los diferentes componentes y así se verá como se ha hecho para hacer una descripción estructural.

Primero mediante una entidad general tal y como habíamos definido en el enunciado pondremos las diferentes entradas y salidas de las que está compuesta nuestra ALU, que serán, las siguientes.

ENTITY alu IS

Port ( a: in std_logic	--Entrada de 1 bit
b: in std_logic	--Entrada de 1 bit
resultat: out std_logic	--Resultado de la operación
op0: in std_logic;	--Op0,Op1,Op2 sirven
op1: in std_logic;	--para seleccionar la operación
op2: in std_logic;	--que tiene que realizar la ALU
cout: out std_logic);	--Carry de salida

END alu;

Ahora en el siguiente trozo de programa haremos las uniones de los diferentes bloques mediante una arquitectura como la siguiente

ARCHITECTURE estructura OF alu IS

COMPONENT and\_or

--Bloque AND y OR

Port ( aa: in std_logic ;	--Entrada del bloque
bb: in std_logic;	--Entrada del bloque
operacio0: in std_logic;	
operacio1: in std_logic;	--operación a realizar
operacio2: in std_logic;	
sor_and_or: out std_logic;	--salida

END COMPONENT;

COMPONENT suma\_resta

--Bloque suma y resta

Port ( aa: in std_logic;	--Entrada del bloque
bb: in std_logic;	
operacio0 in std_logic;	
operacio1: in std_logic;	
operacio2: in std_logic;	--Selección de operación
cout: out std_logic;	--Carry de salida
sor_sum_res: out std_logic	--Salida

END COMPONENT;

COMPONENT multiplexor

--Bloque multiplexor

Port ( entr_a: in std_logic;	
entr_b: in std_logic;	--Entrada del bloque
operacio0: in std_logic;	

```

operacio1: in std_logic;
operacio2: in std_logic;
sortida: out std_logic;

--Selección de operación
--Salida

END COMPONENT;

SIGNAL x: std_logic;
SIGNAL y: std_logic;
BEGIN

U1 :and_or port map (a,b,op0,op1,y);
U2: suma_resta port map (a,b,op0,op1,op2,cout,x);
U3: multiplexor port map (x,y,op0,op1,op2,resultat);

--Paso de parámetros
--Paso de parámetros
--Paso de parámetros

END estructura;
```

Como se puede ver en el ejemplo anterior en las últimas líneas hacemos el paso de parámetros de los componentes a las variables de la **ENTIDAD** principal que es la entidad ALU, lo único que se tiene que tener en cuenta es que al hacer el paso de parámetros lo hagamos en igual orden con el que hemos definido las variables, ya que si no al compilar nos dará error si asignamos variables de tipo **in** a tipo **out**, o igual no nos da error pero si invertimos variables el programa no nos va a funcionar i será un auténtico desastre.

## 2.2 Unidades básicas de diseño

### 2.2.1 Cómo se declara una entidad

En la declaración de entidades, se definen las entradas y salidas de nuestro chip, diciendo cuántas son, de qué tamaño (de 0 a n bits), modo (entrada, salida, ...) y tipo (integer, bit,...) . Las entidades pueden definir bien las entradas y salidas de un diseño más grande o las entradas y salidas de un chip directamente. La declaración de entidades es análoga al símbolo esquemático de lo que queremos implementar, el cual describe las conexiones de un componente al resto del proyecto, es decir, si hay una entrada o puerto de 8 bits, o dos salidas o puertos de 4 bits, etc. La declaración de entidades tiene la siguiente forma:

**ENTITY** programa **IS**

Cabecera del programa

**port**(

Se indica que a continuación viene los puertos (o grupos señales) de entrada y/o salida

```
-- puertos de entradas
-- puertos de salidas
-- puertos de I/O
-- puertos de buffers
```

Aquí se declaran las entradas y/o salidas con la sintaxis que se verá a continuación. Las líneas empezadas por dos guiones son ignoradas por el compilador. Así mismo, recordamos que el compilador no distingue las mayúsculas de las minúsculas

);  
**END** programa;

Se indica que se ha acabado la declaración de puertos de entrada y/o salida, y que se ha acabado la entidad

Como hemos dicho, cada señal en una declaración de entidad está referida a un puerto (o grupo de señales), el cual es análogo a un(os) pin(es) del símbolo esquemático. Un puerto es un objeto de información, el cual, puede ser usado en expresiones y al cual se le pueden asignar valores. A cada puerto se le debe asignar un nombre válido, Un ejemplo de declarar los puertos es el siguiente:

nombre_variable: modo tipo;	Forma genérica de designar un puerto
puertoa: <b>in</b> bit;	El primer puerto es un bit de entrada, y su nombre es "puertoa"
puertob: <b>in</b> bit_vector(0 to 7);	El segundo puerto es un vector de 8 bits de entrada siendo el MSB el puertob(0) y el LSB el puertob(7)
puertoc: <b>out</b> bit_vector(3 downto 0);	El tercer puerto es un vector de 4 bits de salida siendo el MSB el puertoc(3) y el LSB el puertoc(0)
puertod: <b>buffer</b> bit;	El cuarto puerto es un buffer de un solo bit, cuyo nombre es "puertod"
puertoe: <b>inout</b> std_logic;	El quinto puerto es una entrada/salida del tipo estándar logic de un solo bit

Seguido del nombre del puerto y separado de éste por dos puntos, viene el tipo de puerto que va a ser. El modo describe la dirección en la cual la información es transmitida a través del puerto. Éstos sólo pueden tener cuatro valores: **in**, **out**, **buffer** e **inout**. Si no se especifica nada, se asume que el puerto es del modo **in**.

- ✓ **Modo in:** Un puerto es de modo **in** si la información del mismo, solamente debe entrar a la entidad, soliendo ser usado para relojes, entradas de control (como las típicas load, reset y enable), y para datos de entrada unidireccionales.
- ✓ **Modo out:** Un puerto es de modo **out** si la información fluye hacia fuera de la entidad. Este modo no permite realimentación, ya que al declarar un puerto como **out**, estamos indicando al compilador que el estado lógico en el que se encuentra no es leíble. Esto le da una cierta desventaja, pero a cambio consume menos recursos de nuestros dispositivos lógicos programables.
- ✓ **Modo buffer:** Es usado para una realimentación interna (es decir, para usar este puerto como un driver dentro de la entidad). Este modo es similar al modo **out**, pero además, permite la realimentación. Este puerto no es bidireccional, y solo puede ser conectado directamente a una señal interna, o a un puerto de modo **buffer** de otra entidad. Una aplicación muy común de este modo es la de

salida de un contador, ya que debemos saber la salida en el momento actual para determinar la salida en el momento siguiente.

- ✓ Modo **inout**: Es usado para señales bidireccionales, es decir, si necesitamos que por el mismo puerto fluya información tanto hacia dentro como hacia afuera de la entidad. Este modo permite la realimentación interna. Este modo puede reemplazar a cualquiera de los modos anteriores, pudiéndose usar este modo para todos los puertos, pero reduciremos la lectura posterior del código por otra persona, y reduciendo los recursos disponibles de la cápsula.

Como se ha comentado arriba, VHDL sólo admite cuatro modos para los puertos, pero puede haber tantos tipos de señales como queramos, ya que las podemos crear nosotros mismos. VHDL incorpora varios tipos de forma estándar (por haber sido creado así), pudiendo usar otros mediante librerías normalizadas, y los creados por nosotros. La norma internacional IEEE 1076/93 define cuatro tipos que son nativos para VHDL como son:

- ✓ Tipo **boolean**: puede tomar dos valores: verdadero/true o falso/false. Un ejemplo típico es la salida de un comparador que da verdadero si los números comparados son iguales y falso si no lo son:

equal: <b>out boolean</b> ;	Sólo puede tomar dos valores: verdadero o falso, y es de salida (darle mas operatividad a la salida de un comparador sería superfluo)
-----------------------------	---

- ✓ Tipo **bit**: Puede tomar dos valores: 0 ó 1 ( o también "low" o "high", según se prefiera). Es el tipo más usado de los nativos.
- ✓ Tipo **bit\_vector**: Es un vector de bits. Debemos tener cuidado al definir el peso de los bits que lo integran, ya que según pongamos la palabra reservada **downto** o **to** estaremos diciendo que el bit más significativo es el número más alto o el más bajo del vector, respectivamente.

numero : <b>bit_vector</b> (0 <b>to</b> 7);	En este caso el MSB es numero(0) y numero(7) el LSB
numero : <b>bit_vector</b> (7 <b>downto</b> 0);	En este caso el MSB es numero(7) y numero(0) el LSB

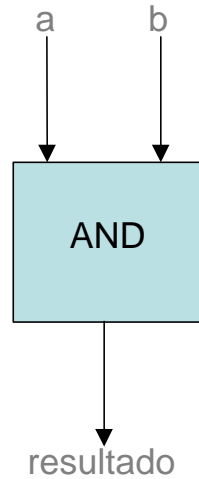
- ✓ Tipo **integer**: Para manejar números enteros. Hay que advertir que el uso de enteros consume muchos recursos de cápsula, siempre y cuando sea sintetizable, ya que está prácticamente creado para la simulación.

Pero ante la necesidad de ampliar la operatividad del tipo bit, la norma IEEE 1164, definió un nuevo tipo llamado std\_logic, y sus derivados tales como std\_logic\_vector y



std\_logic\_vector. Como su nombre pretende indicar, es el tipo de tipo lógico estándar, que es el más usado en la actualidad.

Primero pondremos un ejemplo sencillo de una entidad que consiste en un bloque que nos haga la función AND de dos entradas a y b. **Figura 7**



**Figura 7.** Bloque AND

**ENTITY** and **IS port** (

a,b: **in** std\_logic;  
resultado: **out** bit;

);

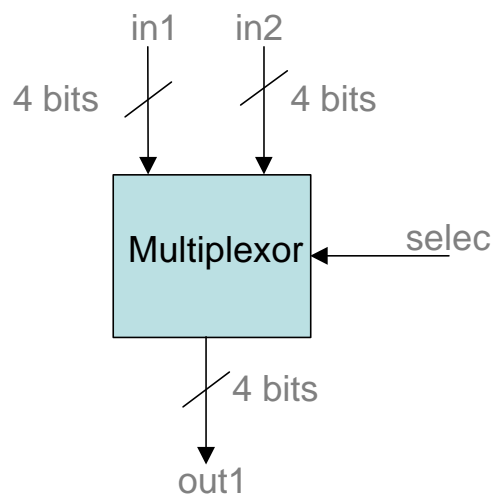
**END** and;

Cabecera de la entidad, cuyo nombre es **and**

- **a** y **b** son las entradas de 1 bit
- **resultado** es la salida de un sólo bit

Se finaliza la entidad con la palabra clave **end** y el nombre de la misma and).

Como ejemplo, a continuación se incluye la declaración de entidades de un multiplexor de 2x1 de cuatro bits, con entrada de habilitación o enable. El multiplexor necesita las entradas de información, la señal de selección, la de enable y las salidas de información. **Figura 8**



**Figura 8.** Multiplexor

**ENTITY** multi **IS port** (

```
    selec: in bit;  
    in1: in std_logic_vector(3 downto 0);  
    in2: in std_logic_vector(3 downto 0);  
    out1: out std_logic_vector(3 downto 0));
```

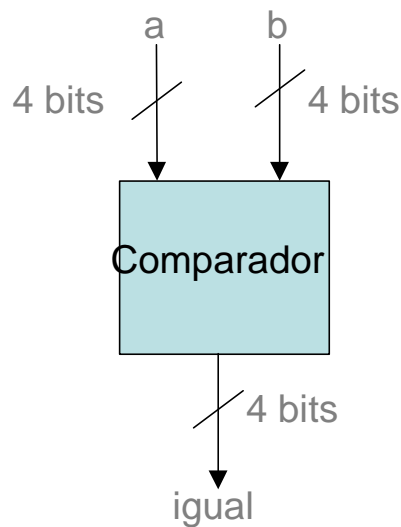
**END** multi;

Cabecera ya estudiada arriba, en la que **multi** es el nombre de la entidad

- **selec** es otro bit de entrada, que selecciona la entrada **in1** o **in2**, ambas de 4 bits
- **out1** es de salida, que lógicamente, debe ser de la misma longitud que **in1** e **in2**

Otro ejemplo se muestra a continuación como es la entidad para un comparador.

**Figura 9**



**Figura 9.** Comparador

**ENTITY** compa **IS port** (

```
    a,b: in std_logic_vector(3 downto 0);  
    igual: out bit;
```

```
);
```

**END** compa;

Cabecera de la entidad, cuyo nombre es **compa**

- **a** y **b** son las entradas de cuatro bits
- **igual** es la salida de un sólo bit

Se finaliza la entidad con la palabra clave **end** y el nombre de la misma (**compa**).

Debemos recordar dos puntos más a la hora de dar el nombre a algún puerto, que se tratarán más adelante en el apartado de objetos:

- ✓ VHDL no distingue las letras mayúsculas de las minúsculas, por lo que un puerto llamado por nosotros "EnTraDA" será equivalente a otro que se llame "ENTRADA" o "entrada".

- ✓ El primer carácter de un puerto sólo puede ser una letra, nunca un número. Así mismo, no pueden contener caracteres especiales como \$, %, ^, @, ... y dos caracteres de subrayado seguidos.

Estos dos detalles a tener en cuenta surgieron del comité que creó este lenguaje, por lo que no se debe considerar como un fallo de nuestra herramienta, sino como una característica más del lenguaje.

## 2.2.2 Cómo se declara una arquitectura

La arquitectura es lo que nos dice que debemos hacer con los puertos 8o grupos de señales de entrada, declarados previamente en la entidad) para llegar a tener los puertos de salida (también declarados en la entidad). En la declaración de entidades es donde reside todo el funcionamiento de un programa, ya que es ahí donde se indica que hacer con cada entrada, para obtener la salida. Si la entidad es vista como una "caja negra", para la cual lo único importante son las entradas y las salidas, entonces, la arquitectura es el conjunto de detalles interiores de la caja negra.

La declaración de arquitecturas debe constar de las siguientes partes como mínimo, aunque suelen ser más:

<b>ARCHITECTURE</b> archpro <b>OF</b> programa <b>IS</b>	Cabecera de la arquitectura. En ésta, <b>archpro</b> es un nombre cualquiera (suele empezar por "arch", aunque no es necesario) y <b>programa</b> es el nombre de una entidad existente en el mismo fichero
-- declaración de señales y otros accesorios	Declaraciones de apoyo, que se verán en la página siguiente
<b>BEGIN</b>	Se da comienzo al programa
-- núcleo del programa	Conjunto de sentencias, bucles, procesos, funciones,... que dan operatividad al programa.
<b>END</b> archpro;	Fin del programa

Como podemos apreciar, es una estructura muy sencilla, y que guarda alguna relación con Turbo Pascal. Las sentencias entre **begin** y **end** son las que realmente "hacen algo". A continuación, pondremos primero la arquitectura de la AND la cual debe ir unido a la entidad expuesta en el apartado de la declaración de entidades, ya que una parte sin la otra carecen de sentido. **Figura 10**

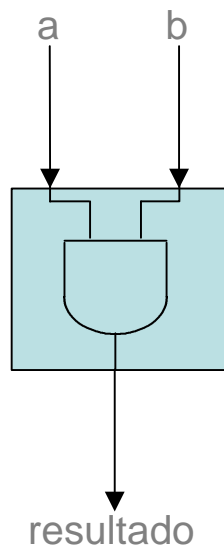


Figura 10. AND

**ARCHITECTURE** archiand **OF** and **IS**

Cabecera de la arquitectura. En esta ocasión el nombre de la arquitectura es **archiand**, y el de la entidad es **and**, la cual está definida anteriormente.

-- señales

En este programa no vamos a necesitar señales

**BEGIN**

Se da comienzo al programa

resultado<=a and b;

Asignación para hacer la función AND

**END**archiand;

Fin del programa

Después del anterior ejemplo que muy sencillo pasaremos hacer el ejemplo del multiplexor, que también en el apartado anterior especificamos su identidad, ahora definiremos su función mediante la arquitectura siguiente, primero haremos el dibujo del multiplexor con su puertas lógicas, y después pasaremos a implementarlo. **Figura 11**

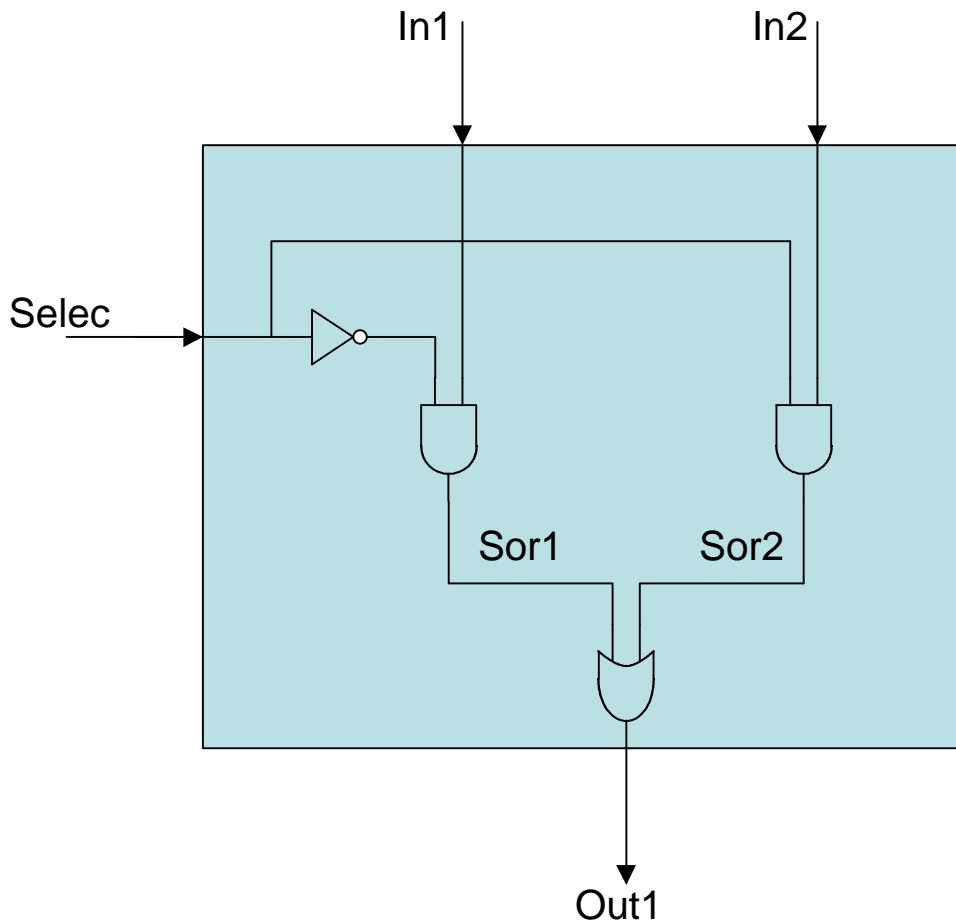


Figura 11. Multiplexor

Ahora haremos el programa con VHDL, que será el siguiente:

**ARCHITECTURE** archimulti **OF** multi **IS**

**SIGNAL** sor1 : std\_logic\_vector (3 **downto** 0);

**SIGNAL** sor2 : std\_logic\_vector (3 **downto** 0);

**SIGNAL** selecnot : std\_logic;

**BEGIN**

Selecnot<= **not** selec;

Sor1(0)<=selecnot **and** in1(0);

Sor2(0)<=selec **and** in2(0);

Cabecera de la arquitectura. En esta ocasión el nombre de la arquitectura es **archimulti**, y el de la entidad es **multi**, la cual está definida anteriormente.

Señales necesarias tal como se puede observar en el dibujo anterior

Se da comienzo al programa

Operaciones con cada uno de los 4 bits. Para cada bit se tiene que implementar el circuito del dibujo anterior.

```
Out(0)<=sor1(0) or sor2(0);
```

```
Sor1(1)<=selecnot and in1(1);
```

```
Sor2(1)<=selec and in2(1);
```

```
Out(1)<=sor1(1) or sor2(1);
```

```
Sor1(2)<=selecnot and in1(2);
```

```
Sor2(2)<=selec and in2(2);
```

```
Out(2)<=sor1(2) or sor2(2);
```

```
Sor1(3)<=selecnot and in1(3);
```

```
Sor2(3)<=selec and in2(3);
```

```
Out(3)<=sor1(3) or sor2(3);
```

```
END multi;
```

Fin del programa

## 2.3 VHDL para síntesis

La síntesis de un circuito, a partir de una descripción VHDL, consiste en reducir el nivel de abstracción de la descripción del circuito hasta convertirlo en una definición puramente estructural cuyos componentes son elementos de una determinada biblioteca. Esta biblioteca dependerá del circuito que se quiera realizar, la herramienta de síntesis, etc. Al final del proceso de síntesis se debe obtener un circuito que funcionalmente se comporte igual que la descripción que de él se ha hecho.

En un principio cualquier descripción en VHDL es sintetizable, no importa el nivel de abstracción que la descripción pueda tener. Esto, que en principio puede parecer sorprendente no lo es en absoluto, ya que cualquier descripción en VHDL se puede simular, y si se puede simular, el propio simulador(en general un ordenador ejecutando un programa) es un circuito que funcionalmente se comporta tal y como se ha descrito, por lo tanto, desde este punto de vista es una síntesis del circuito que se ha diseñado. Es evidente que no será el circuito más optimizado para realizar la tarea que se pretende, ni lo hará a la velocidad que se requiere, pero seguro que funcionalmente se comporta tal y como se ha descrito.

La complejidad del circuito resultante y también incluso la posibilidad o no de realizar el circuito, va a depender sobre todo del nivel de abstracción inicial que tenga la descripción. Como primera solución se puede utilizar un ordenador que ejecute la

simulación y ya se tiene la síntesis. A partir de este primer intento, hay que ir optimizando el circuito. En realidad las herramientas de síntesis siguen una aproximación distinta, ya que de otra manera el circuito acabaría siendo algo parecido a un microprocesador cuando quizá en realidad sólo se pretende, eventualmente, implementar una puerta lógica.

La aproximación de las herramientas de síntesis consiste en, partiendo de la descripción original, reducir el nivel de abstracción hasta llegar a un nivel de descripción estructural. La síntesis es por tanto una tarea vertical entre los niveles de abstracción de un circuito. Así, una herramienta de síntesis comenzaría por la descripción comportamental abstracta algorítmica e intentaría traducirla a un nivel de transferencia entre registros descrita con ecuaciones de conmutación. A partir de esta descripción se intenta transformarla a una descripción estructural donde se realiza, además, lo que se llama el mapeado tecnológico, es decir, la descripción del circuito utilizando los componentes de una biblioteca concreta que depende de la tecnología con la cual se quiera implementar el diseño.

Las herramientas de síntesis actuales cubren a la perfección la síntesis a partir de descripciones RTL y estructurales, pero no están tan avanzadas en el manejo de diseños descritos en un nivel de abstracción más alto. No es que no se pueda sintetizar a partir de un nivel alto de abstracción, lo que ocurre es que la síntesis obtenida no es quizá la más óptima para el circuito que se pretende realizar.

## **2.4 Construcciones básicas**

El primer paso es ver si un circuito describe lógica combinacional o secuencial. Un circuito describe lógica combinacional si la salida depende únicamente de la entrada en ese instante, y no de la entrada que hubiera tenido en un pasado, es decir, ante una entrada dada la salida es siempre la misma. Un circuito describe lógica secuencial cuando la salida depende de la entrada actual y de las entradas anteriores, o dicho de otra forma, la salida depende de la entrada y del estado del sistema. Esto introduce un nuevo elemento dentro del sistema que será la memoria. Normalmente el elemento de memoria será una señal que frente a unos estímulos captura otra señal y en caso contrario permanece igual. Esto nos da una pista de si un circuito es secuencial y si se realizará por tanto a partir de elementos de memoria como pueden ser cerrojos o registros.

En este proyecto solo estudiaremos la lógica combinacional ya que solo interviene este tipo, la idea básica es que si en la estructura del lenguaje no se introducen “elementos de memoria” entonces se está delante de una descripción combinacional. Se va a mostrar entonces cómo evitar que aparezcan elementos de memoria para que el circuito se realice sólo con puertas lógicas.

## 2.5 Objetos

En un lenguaje de descripción de software (SDL) una variable contiene un valor y puede aceptar un nuevo valor a través de una asignación secuencial. Por otro lado, las constantes tienen valores prefijados a lo largo de toda la ejecución del programa. Sin embargo, en VHDL se hace necesaria la utilización de un nuevo tipo de objeto que puede emular las asignaciones concurrentes propias de los circuitos eléctricos reales; este nuevo tipo de objeto son las señales.

Un objeto en VHDL es un elemento que tiene asignado un valor de un tipo determinado. Según sea el tipo de dato, el objeto poseerá un conjunto de operaciones que se le podrán aplicar. En general, no será posible realizar operaciones entre dos objetos de distinto tipo, a menos que definamos previamente un programa de conversión de tipos.

## 2.6 Identificadores

Los identificadores son un conjunto de caracteres dispuestos de una forma adecuada y siguiendo unas normas propias del lenguaje, para dar un nombre a los elementos en VHDL, por lo que es aconsejable elegir un nombre que sea representativo y que facilite la comprensión del código.

Las reglas a tener en cuenta a la hora de elegir un identificador son:

- ✓ Los identificadores deben empezar con un carácter alfabético, no pudiendo terminar con un carácter subrayado, ni tener dos o más de estos caracteres subrayados seguidos.
- ✓ VHDL identifica indistintamente tanto las mayúsculas como las minúsculas, pudiéndose emplear por igual el identificador "sumador" o "SUMADOR".
- ✓ El tamaño o extensión del identificador no está fijado por VHDL, siendo recomendable que el usuario elija un tamaño que confiera sentido y significado al identificador, sin llegar a alcanzar longitudes excesivamente largas.
- ✓ Los identificadores pueden contener caracteres numéricos del '0' al '9', sin que éstos puedan aparecer al principio.
- ✓ No puede usarse como identificador una palabra reservada por VHDL.



## 2.7 Palabras reservadas

Las palabras reservadas son un conjunto de identificadores que tienen un significado específico en VHDL. Estas palabras son empleadas dentro del lenguaje a la hora de realizar un diseño. Por esta razón y buscando obtener claridad en el lenguaje, las palabras reservadas no pueden ser empleadas como identificadores definidos por el usuario.

Las palabras reservadas por VHDL son:

abs	else	nand	return
access	elsif	new	select
after	end	next	severity
alias	entity	nor	signal
all	exit	not	subtype
and	file	null	then
architecture	for	of	to
array	function	on	transoprt
asser	generate	open	type
attribute	generic	or	units
begin	guarded	others	until
block	if	out	use
body	in	package	variable
buffer	inout	port	wait
bus	is	procedure	when
case	label	process	while
component	library	range	with
configuration	linkage	record	xor
constant	loop	register	
disconnect	map	rem	
downto	mod	report	

## 2.8 Símbolos especiales

Además de las palabras reservadas empleadas como identificadores predefinidos, VHDL utiliza algunos símbolos especiales con funciones diferentes y específicas, tales como el símbolo "+" se utiliza para representar la operación suma y, en este caso, es un operador. El símbolo "--" es empleado para los comentarios realizados por el usuario, de tal forma que el programa al encontrar una instrucción precedida por "--" la saltará ignorando su contenido. De esta forma, el programador puede hacer más comprensible el código del programa.

Los símbolos especiales en VHDL son:

( ) . , : ; & ' < > = | # <= => := -- + - /

Para finalizar, recordar el símbolo más empleado por un programador que es el ";", símbolo que debe finalizar todas y cada una de las líneas del código dando por terminada dicha sentencia en el programa.

### 3 Explicación de la ALU de 8 bits

#### 3.1 Explicación paso a paso

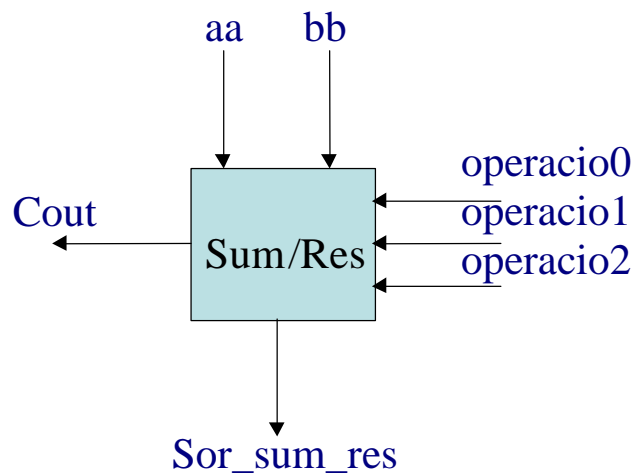
En este capítulo explicaremos paso a paso como se ha hecho la ALU de 8 bits, mediante los esquemas. Cuando tenemos todos los esquemas hechos, entonces haremos el programa en VHDL que está expuesto en el **anexo 1**.

Una vez se tienen claros todos los esquemas, es bastante sencillo programar en VHDL.

Primero ahora definiremos los diferentes bloques que tendrá la ALU, que serán tres:

- ✓ Bloque que ara la suma y la resta y también el inverso del registro B
- ✓ Bloque que ara las operaciones AND i OR
- ✓ Bloque multiplexor

Todos estos bloques estarán dentro de nuestra ALU Ahora definiremos por pasos todos los bloques y el último será hacer un esquema general de todo unido. Primero

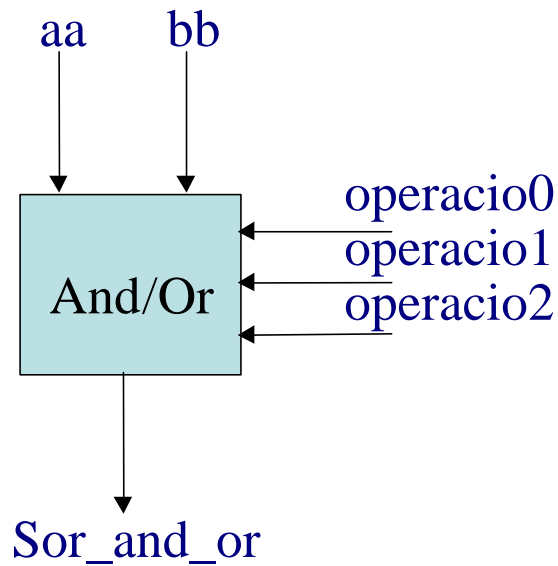


empezaremos por el bloque suma\_resta que será el siguiente tal y como se puede observar. en la figura siguiente. **Figura 12.**

**Figura 12.** Esquema general bloque suma\_resta

En la **figura 12** hemos definido el bloque las entradas y salidas que tendrá, posteriormente cuando hagamos definido todos los bloques y también los hagamos interconexionado todos, haremos los esquemas interiores de cada bloque por separado.

Ahora pasaremos a hacer el esquema del bloque AND/OR tal y como se muestra en

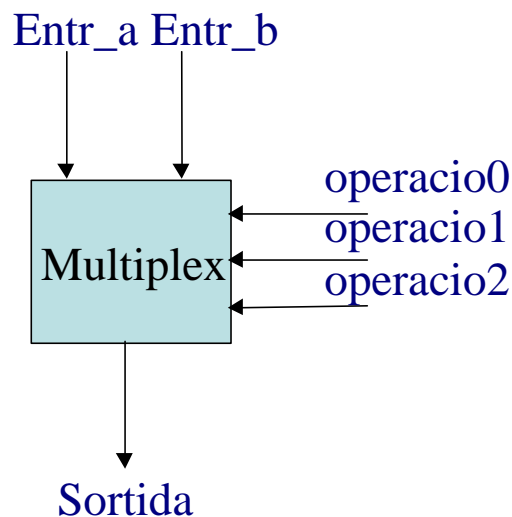


la siguiente figura. **Figura 13.**

**Figura 13.** Esquema general bloque AND/OR

En este bloque de la **figura 13** anterior también podemos observar las entradas y salidas que tendremos.

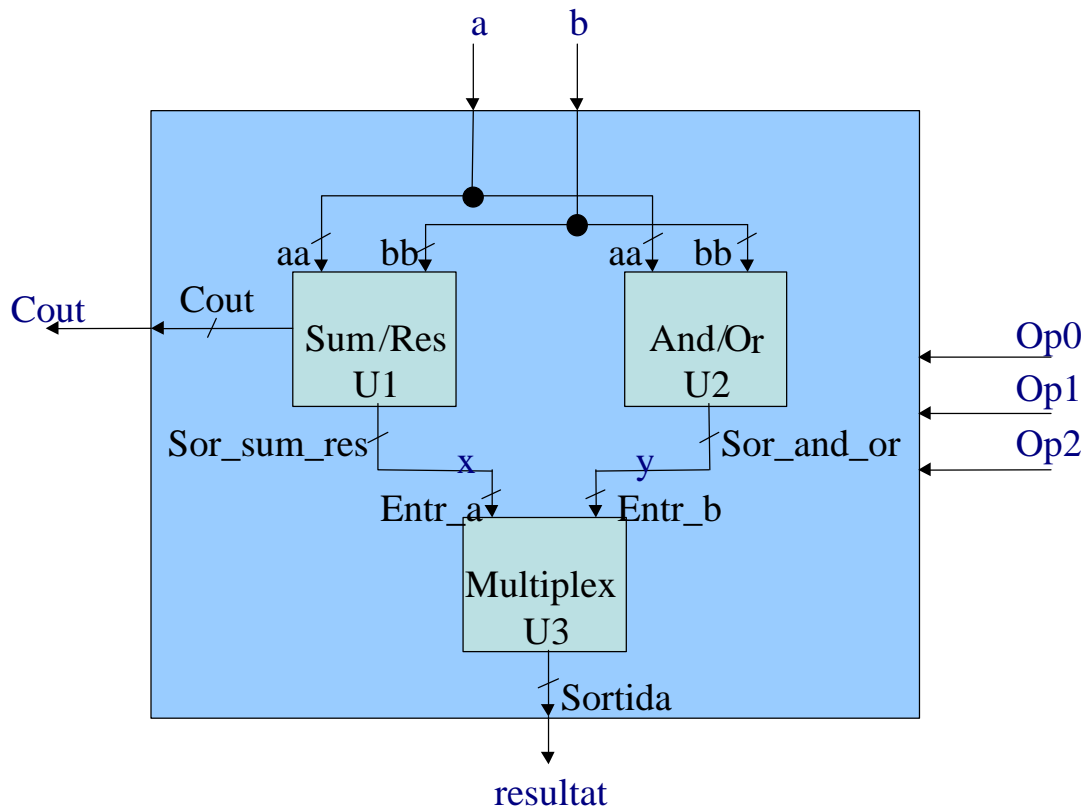
Ahora haremos el bloque multiplexor que lo que nos hará este bloque es escoger si tenemos que coger el resultado del bloque suma\_resta o del bloque AND/OR, y ponerlo a la salida, dependiendo de que hagamos puesto en las operaciones escogeremos un



resultado, u otro. Seguidamente pasaremos a ver la figura del multiplexor. **Figura 14**

**Figura 14.** Esquema general bloque multiplexor

Ahora lo que haremos es unir todos los bloques anteriores, y entonces ya tendremos



definida nuestra ALU, en la siguiente figura. **Figura 15** pondremos el esquema general de la ALU con todas sus entradas y salidas.

**Figura 15.** ALU

Como se puede observar en la figura anterior. **Figura 15** tendremos nuestras entradas y salidas generales que serán, las que están en azul que están fuera del cuadrado son las entradas y salidas generales, mientras que las que están dentro del cuadrado, que están en negro serán variables internas que he creado para después poder diseñar el programa, después aparte de estas entradas y salidas que tenemos internamente como ya se verá posteriormente iré creando nuevas variables y señales para que todo quede más claro.

Variables generales de entrada y salida de la ALU:

1. A) Entrada de 8 bits
2. B) Entrada de 8 bits
3. Cout) Salida de 1 bit
4. Resultat) Salida de 8 bits
5. Op0) Entrada de 1 bit
6. Op1) Entrada de 1 bit

## 7. Op2) Entrada de 1 bit

Como se puede observar tendremos dos entradas de 8 bits que serán con las que haremos las operaciones, y después tres bits Op0, Op1, Op2, que nos dirán que operaciones queremos hacer: suma, resta, inversa de b, AND, OR. Por último tendremos el resultado de 8 bits, y también tendremos el carry de salida en caso de que hagamos una suma o una resta, en el caso de que hagamos otras operaciones no lo tendremos en cuenta.

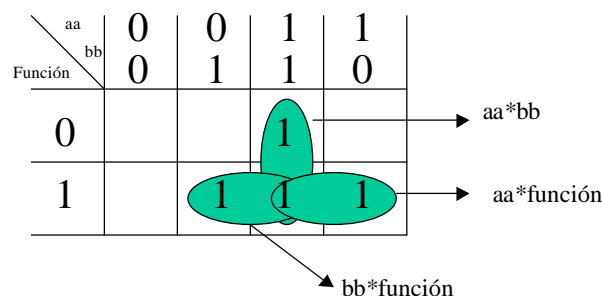
### 3.2 Esquema Interno Bloque AND/OR

Ahora primero haremos la tabla de la verdad. **Figura 16**

AA	BB	Función	SALIDA
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	1

**Figura 16.** Tabla 1

La variable función sirve para decidir si queremos hacer una AND o una OR. Como se puede observar si ponemos un 0 en función haremos la función AND, y si por el contrario ponemos un 1 en función haremos la función OR. Ahora pasaremos hacer la simplificación por Karnaugh. **Figura 17**

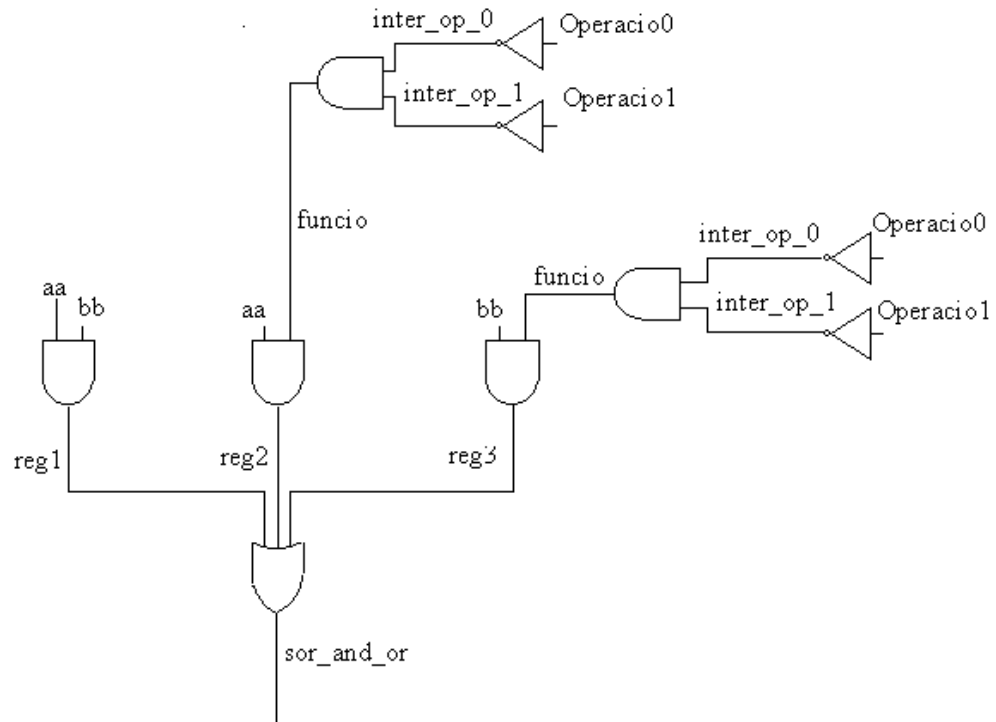


$$\text{Salida} = aa*bb + aa*función + bb*función$$

**Figura 17.** Simplificación AND/OR

(1)

Entonces como las funciones yo las elegiré con Operacion0, Operacion1, Operacion2, esto tengo que adaptar la variable función a estas operaciones, y entonces nos quedará ya la figura del bloque interno AND/OR que será el siguiente. **Figura 18.**



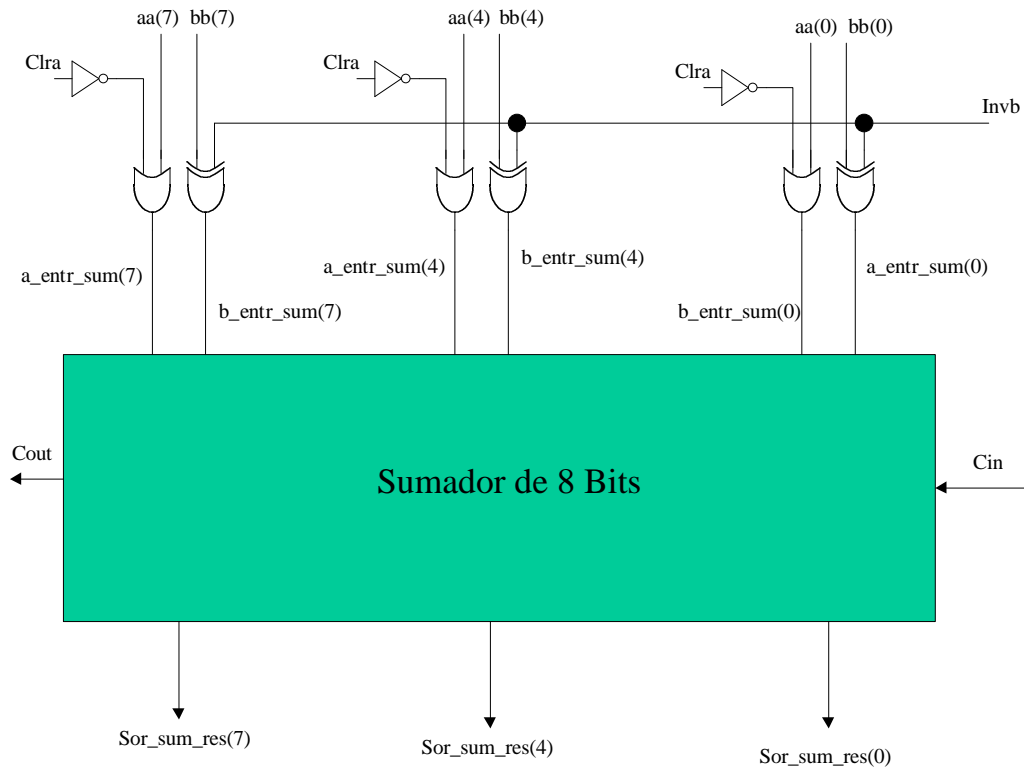
**Figura 18.** Esquema interno AND/OR

Las variables intermedias que he puesto y que no he comentado, son variables que necesitaré para hacer el programa en VHDL, si se quieren ver para que sirven estas variables consultar el código del programa VHDL en el **anexo 1**, y mirar en el apartado del bloque AND/OR donde se verán estas señales.

Para sacar la variable funcio de las operacio0, operacio1, operacio2, mas adelante pondré una tabla y todas las simplificaciones donde se podrá apreciar como se han obtenido, mirar **figura 23**.

### 3.3 Esquema interno bloque suma\_resta

Ahora en este apartado pondremos los dibujos necesarios para que se entienda el bloque que nos hará la suma y la resta, y también la inversión de la señal B. **Figura 19**

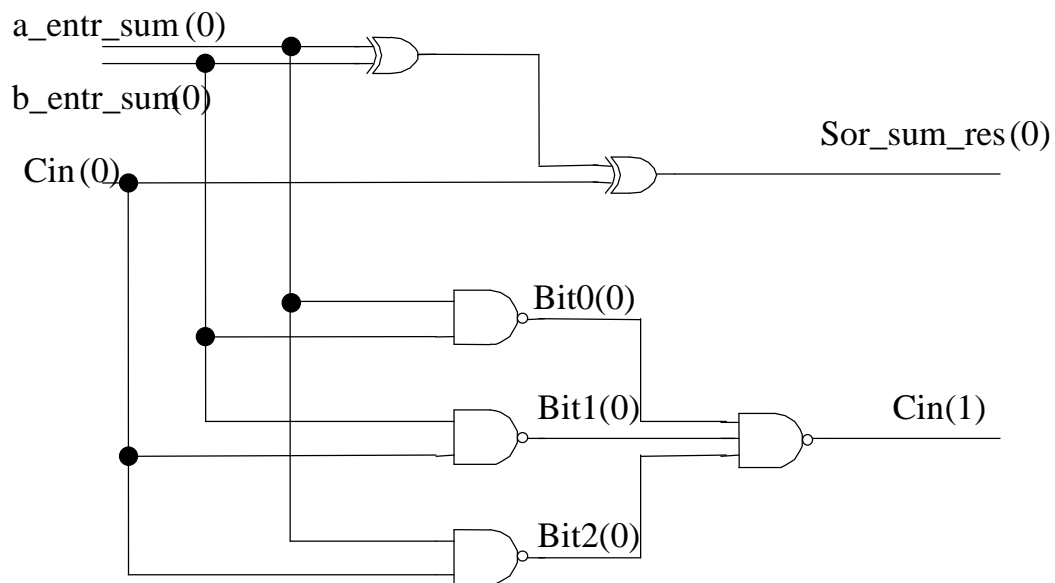


**Figura 19.** Sumador

En la **figura 19** podemos ver que solo hemos implementado el bit 0, el bit 4 y el bit 7, pero para hacer los otros que faltan en medio es lo mismo. También podemos observar que ponemos unas señales que aquí no nos interesan, pero veremos que para hacer el programa en VHDL serán necesarias, mirar **anexo 1** en el bloque suma\_resta y las veremos identificadas.

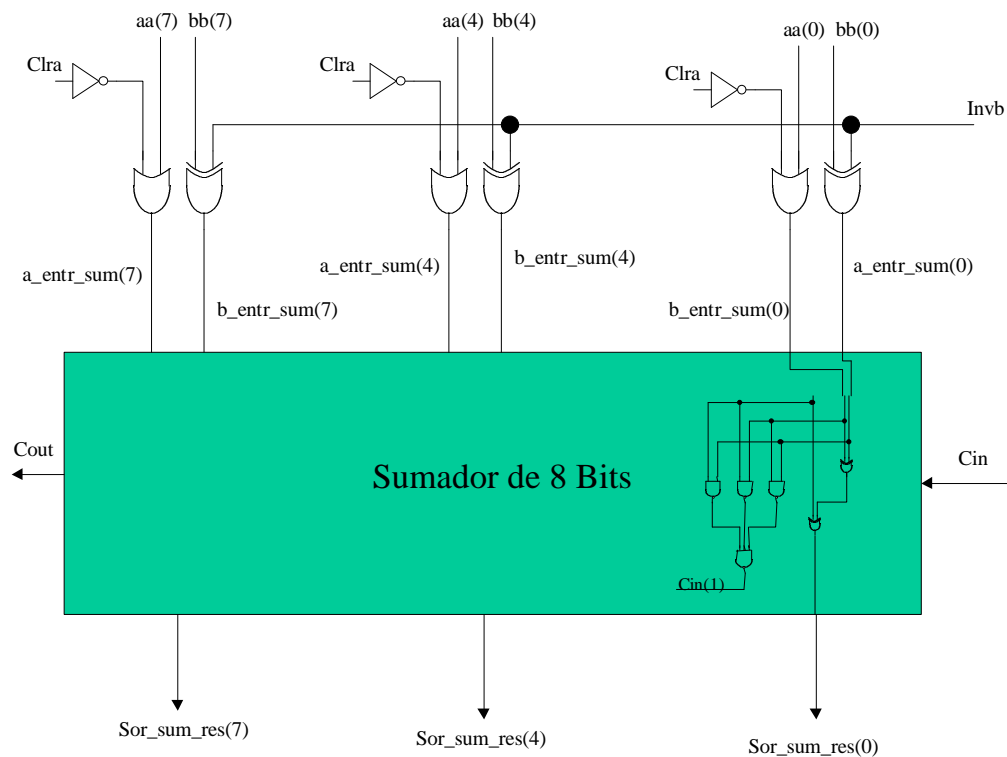


En la figura 19 Dentro del bloque de color verde que pone sumador de 8 bits dentro habrá ocho sumadores como los que se muestran en la siguiente figura. **Figura 20.**



**Figura 20.** Sumador interno

Ahora con las dos figuras anteriores pondremos como quedaría el dibujo final en la siguiente figura. **Figura 21.**



**Figura 21.** Sumador completo

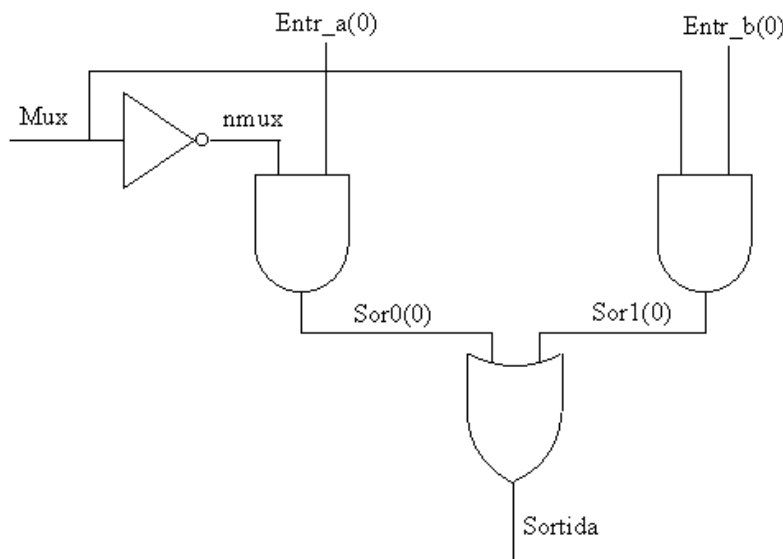
En la **figura 21**, se puede observar como quedaría el sumador con las **figuras 19 y 20** juntas. Para cada bit se tendrá que poner un sumador total igual que el de la **figura 20**.

Ahora después de estas especificaciones anteriores ya tenemos el bloque que suma, resta, y invierte b. Mas adelante pondremos una tabla, **figura 23**, donde especificaremos como se hacen cada una de las operaciones, dependiendo de que valor pongamos en CLRA o en INVB haremos la suma la resta o la inversión, esto ya lo observaremos más adelante.

Después de la **figura 23** pondremos las simplificaciones de CLRA, INVB, CIN, para ponerlas en funcion de operacio0, operacion1, operacio2, que son las que nos tienen que decir que operación tenemos que hacer.

### 3.4 Bloque multiplexor

En este apartado pondremos como hemos hecho el bloque multiplexor, seguidamente pondremos un esquema de un multiplexor de 1 bit, y lo que se tendrá que hacer es lo mismo pero para ocho bits, por lo tanto explicaremos uno y entonces solo será cuestión de poner 7 más. En la siguiente figura pondremos un multiplexor. **Figura 22**



**Figura 22.** Multiplexor

En la **figura 22** se observa un multiplexor, las señales que de más las necesitaremos para hacer el programa como se puede observar en el **anexo 1**.

La entrada mux es la que nos decide que salida tenemos que poner en sortida. Esta entrada mux nos dependerá de operacio0, operacio1, operacio2, que más adelante en la **figura 23** pondremos como se llega a mux y sus simplificaciones.

### 3.5 Tabla y simplificaciones

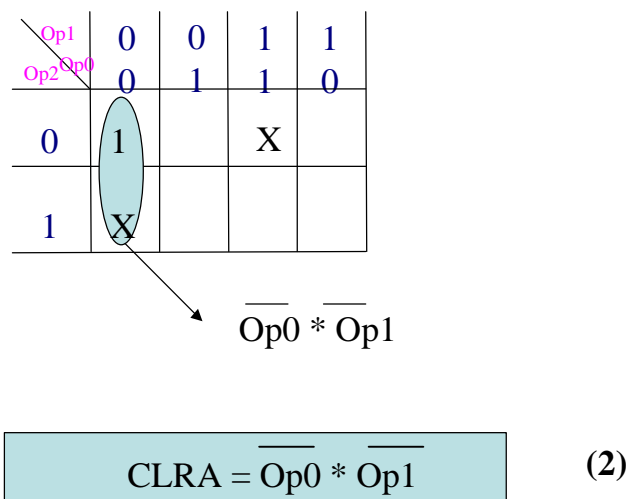
Una vez tenemos claro todos los dibujos anteriores tendremos que dibujar una tabla con las operaciones que tenemos que hacer, y según las entradas Op que pongamos tenemos que definir que salida queremos. Esta tabla también nos servirá para simplificar como veremos próximamente. **Figura 23**

Op2	Op1	Op0	Operación	CLRA	INVB	MUX	CIN	Funcio
0	0	0	Inversion B	1	1	0	0	X
0	0	1	Suma	0	0	0	0	X
0	1	0	Resta	0	1	0	1	X
0	1	1	AND	X	X	1	X	0
1	0	0	OR	X	X	1	X	1

**Figura 23.** Tabla 2

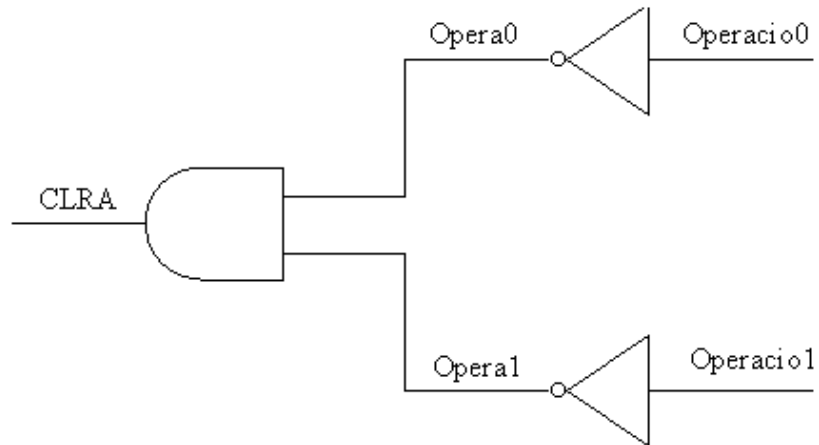
Aquí ahora pondremos todas las tablas de Karnaught de cada variable.

#### 3.5.1. Karnaught de CLRA. Figura 24



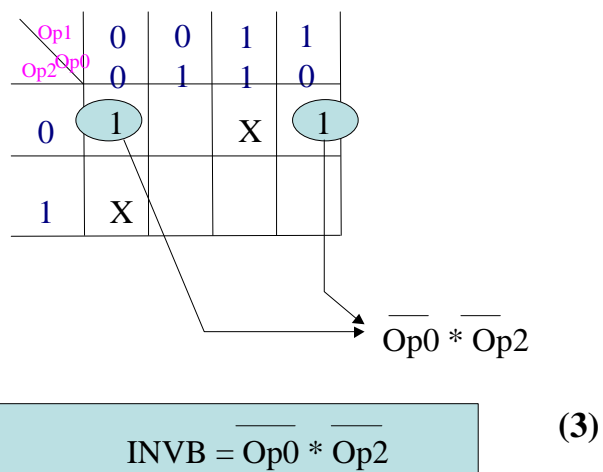
**Figura 24.** Simplificación CLRA

Con la función anterior entonces tendremos el siguiente esquema, **Figura 25.**



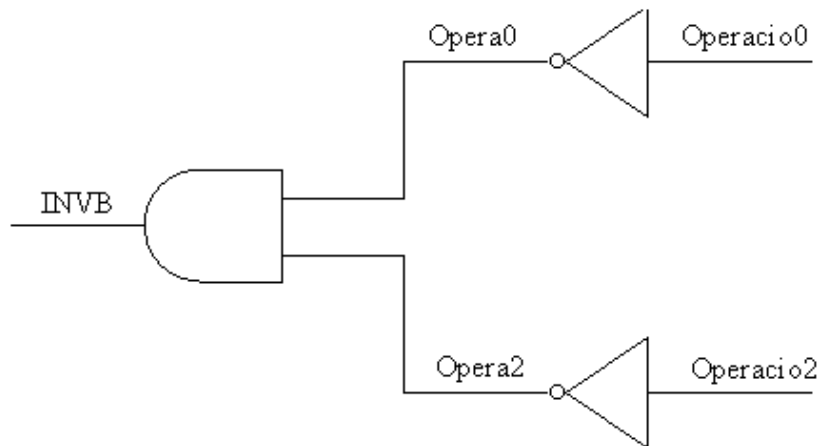
**Figura 25.** Esquema CLRA

### 3.5.2. Karnaught de INVB que será. Figura 26



**Figura 26.** Simplificación INVB

Seguidamente pondremos el esquema. **Figura 27**



**Figura 27.** Esquema INVB

### 3.5.3. Simplificación de Mux. Figura 28

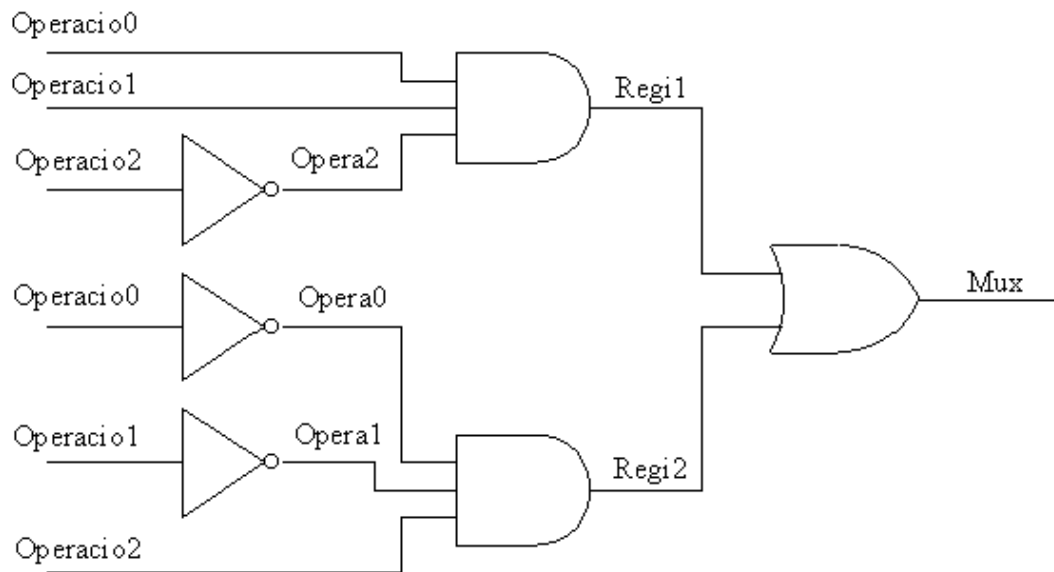
Op1 Op2	0	0	1	1
Op0	0	1	1	0
0			1	
1	1			

$\overline{\text{Op0}} * \overline{\text{Op1}} * \text{Op2}$        $\text{Op0} * \text{Op1} * \overline{\text{Op2}}$

$$\text{MUX} = \overline{\text{Op0}} * \overline{\text{Op1}} * \text{Op2} + \text{Op0} * \text{Op1} * \overline{\text{Op2}} \quad (4)$$

**Figura 28.** Simplificación Mux

**Figura 29.**



**Figura 29.** Esquema Multiplexor

### 3.5.4. Simplificación de Cin.Figura 30

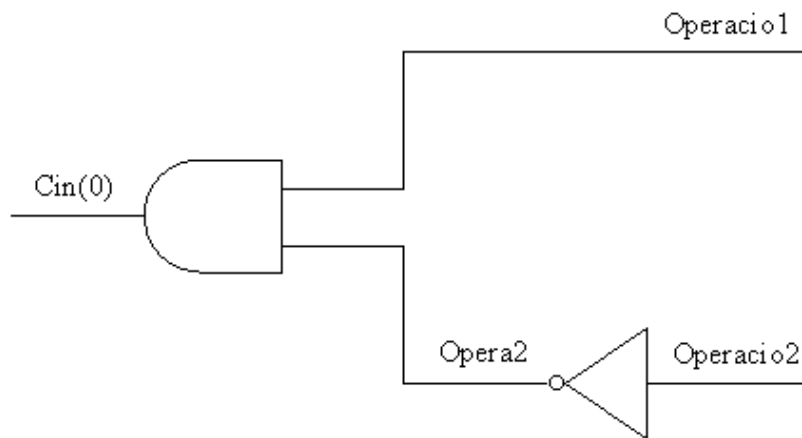
Op1 Op2	0	0	1	1
	0	1	1	0
0			X	1
1	X			

$$\text{Op1} * \overline{\text{Op2}}$$

$$\text{CIN} = \text{Op1} * \overline{\text{Op2}} \quad (5)$$

**Figura 30. Simplificación Cin**

Aquí pondremos su esquema que será el de la figura siguiente. **Figura 31**



**Figura 31.** Esquema Cin

### 3.5.5. Simplificación de función. Figura 32

Op1	0	0	1	1
Op0	0	1	1	0
Funcio	X	X		X
	1			

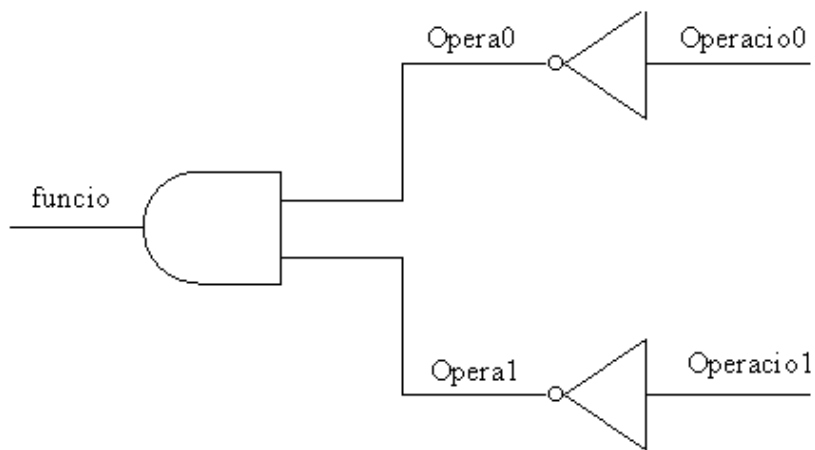
$\text{FUNCIO} = \overline{\text{Op1}} * \overline{\text{Op0}}$

$$\text{FUNCIO} = \overline{\text{Op1}} * \overline{\text{Op0}}$$

(6)

**Figura 32.** Simplificación función

Aquí pondremos el dibujo de la funcio. **Figura 33**



**Figura 33.** Esquema Funcio

Cuando tengo todos los esquemas hechos lo que tendré que hacer es implementarlos en código VHDL, y ponerlos en su lugar adecuado, para que quede más claro, mirar el **anexo 1** donde aparece el código del programa y se podrá ver en los diferentes apartados todas las variables que hemos visto en las simplificaciones.

Como se puede observar en el **anexo 1**, el código una vez tienes hecho todos los esquemas, y lo tienes todo claro resulta bastante evidente como se hace, cuando tienes claro como se programa en VHDL.



## 4 Guia del programa eProduct Designer

El programa que utilizaremos para compilar el código VHDL, para probar que funciona, para sintetizar, para sacar el pin-out de la ALU y para todo será el **eProduct Designer** también dicho **Dashboard**.


### 4.1 Funcionamiento paso a paso del programa

#### 4.1.1 Como hacer para compilar

Aquí pondremos los pasos a seguir para comenzar a utilizar el programa, lo haremos de la siguiente manera:

1) Abrir en programas **eProduct Designer**.

2) **Programmable Design**.

3) Escogemos el icono siguiente  y nos aparecerá la siguiente **figura 34**.

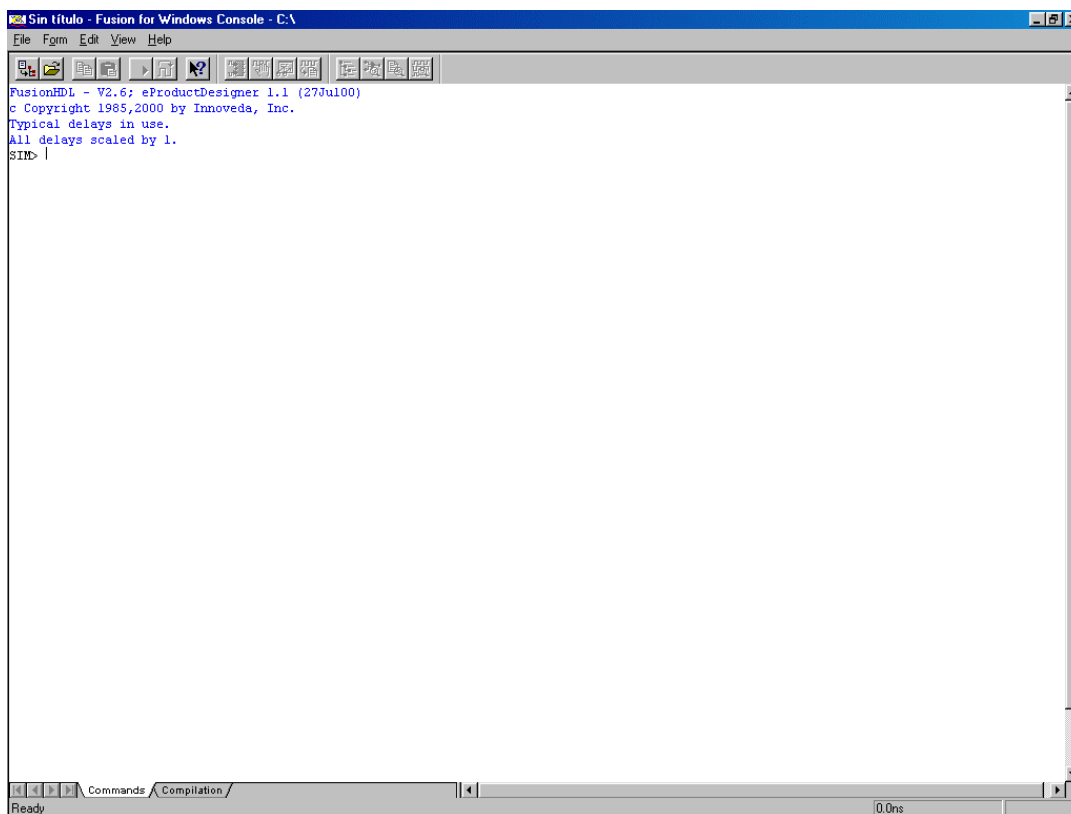
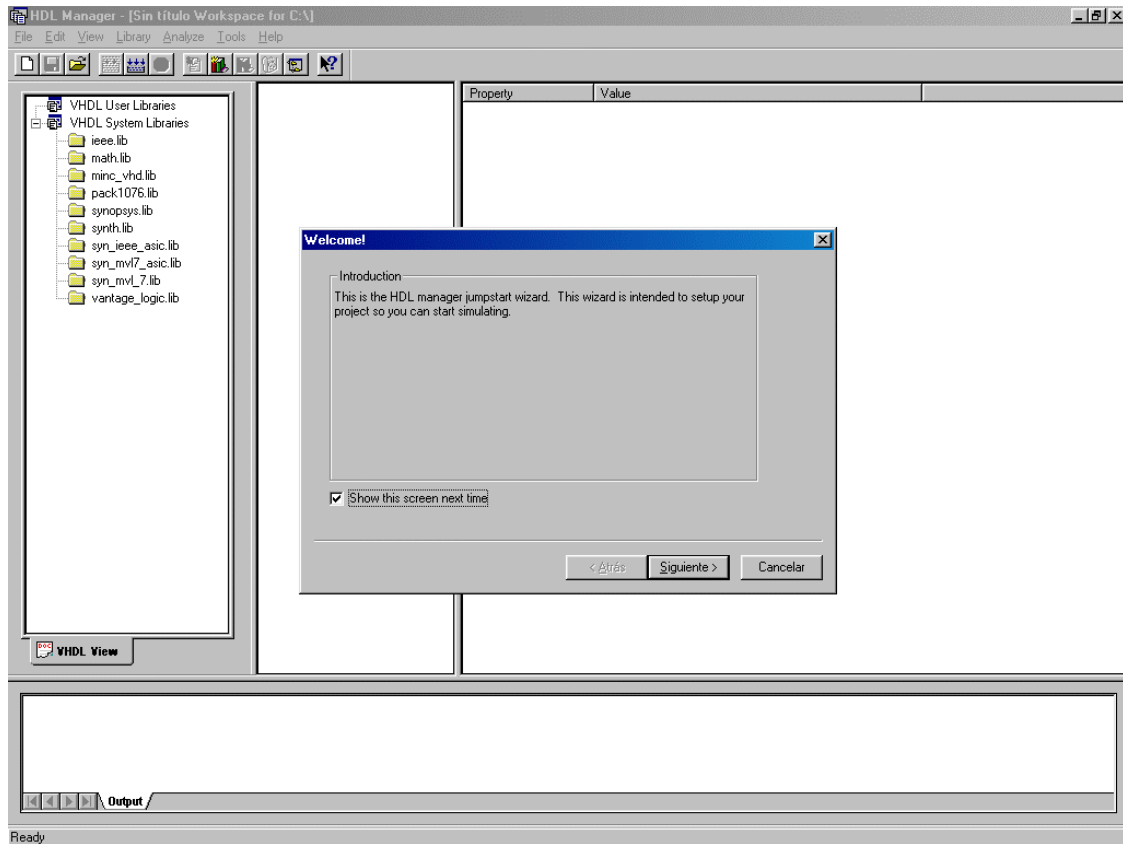


Figura 34. Menú 1

4) Abrimos **File > Analyze VHDL Design** , y entonces podremos ver la **figura 35**.



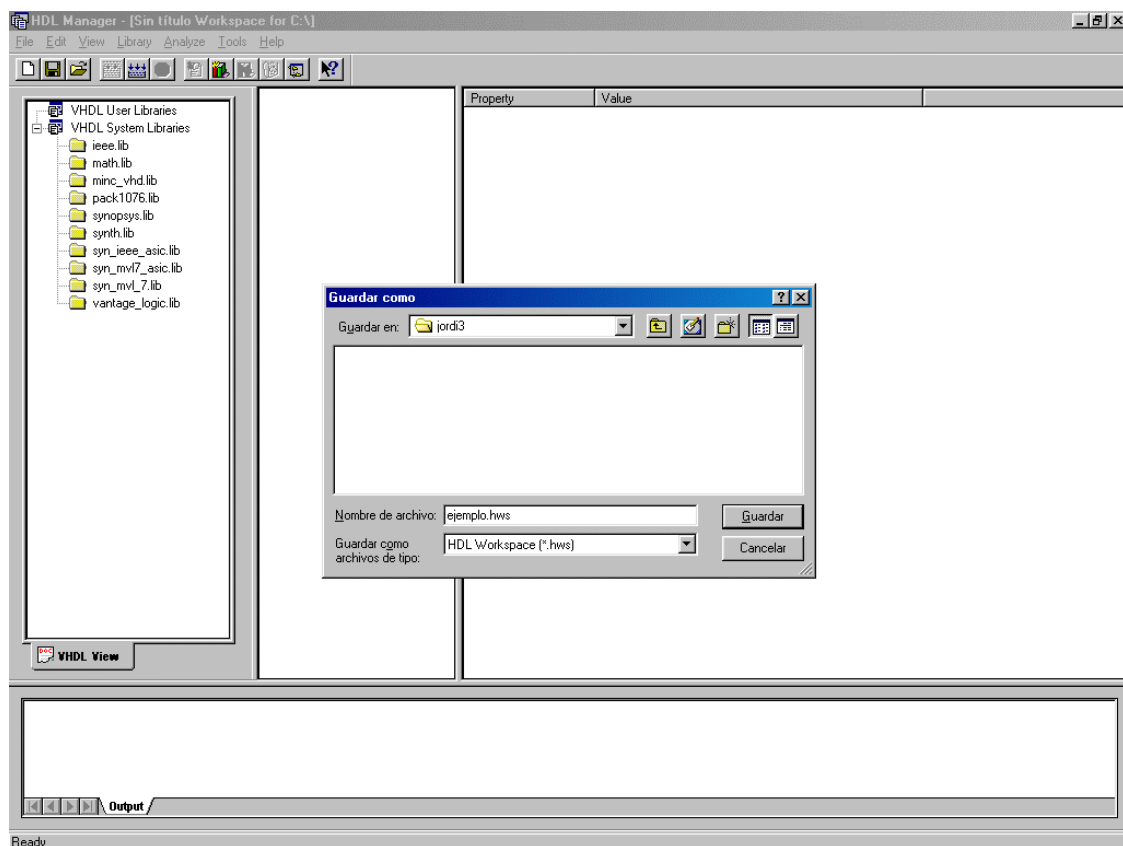
**Figura 35.** Menú 2

5) En el submenú de la figura anterior, **figura 35** tenemos que elegir la opción **Cancelar**.

6) Ahora tendremos que crear un VHDL Workspace para incluir los archivos que nosotros queramos compilar, con las librerías que necesitemos en cada caso.

7) Seleccionar **File > New**.

8) Seleccionar **File > Save as**. Le ponemos un nombre y lo guardamos dentro del directorio que nosotros tenemos los archivos VHDL que queremos compilar. Podemos observarlo en la **figura 36**.



**Figura 36.** Menú 3

9) En el submenú de la figura anterior, **figura 36** poner un nombre donde pone nombre de archivo y después seleccionar **Guardar**.

10) Ahora lo que tenemos que hacer es seleccionar las librerías que necesitamos para compilar nuestro programa.

11) Para seleccionar una determinada librería lo que tenemos que hacer es lo siguiente, ponernos encima de la que queramos y pulsar el botón derecho del ratón y entonces nos aparecerá el siguiente menú, **figura 37**





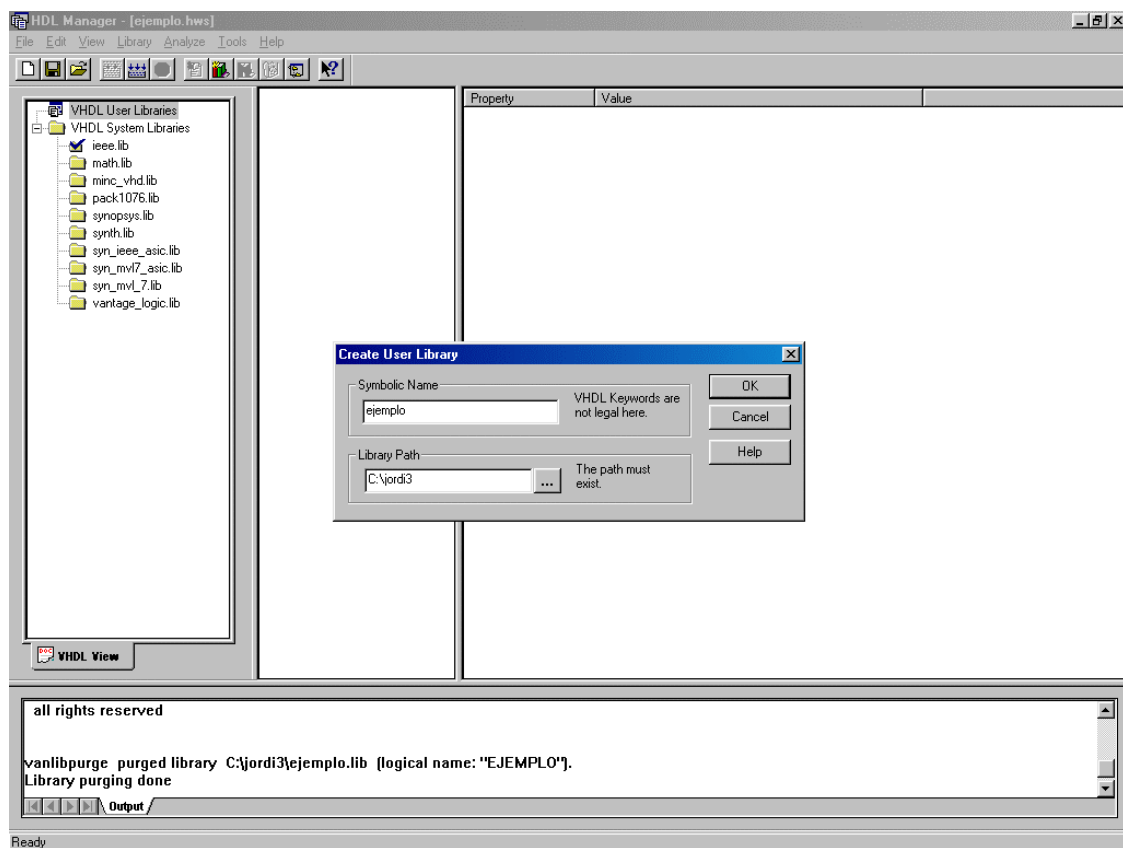


Figura 39. Menú 6

Tenemos que tener cuidado, en el submenú de la figura anterior, **figura 39** donde nos aparece **Symbolic Name** pondremos el nombre que nosotros queramos. En **Library Path** tenemos que estar en el directorio donde nosotros tengamos los archivos VHDL que queremos simular. Esto de estar en el directorio donde tengamos los archivos VHDL es importante de lo contrario si ponemos otro directorio el programa nos dará error y no nos funcionará.

16) Cuando tengamos los nombres correctos hacemos click en **OK**, y entonces nos aparecerá la siguiente figura. **Figura 40**.

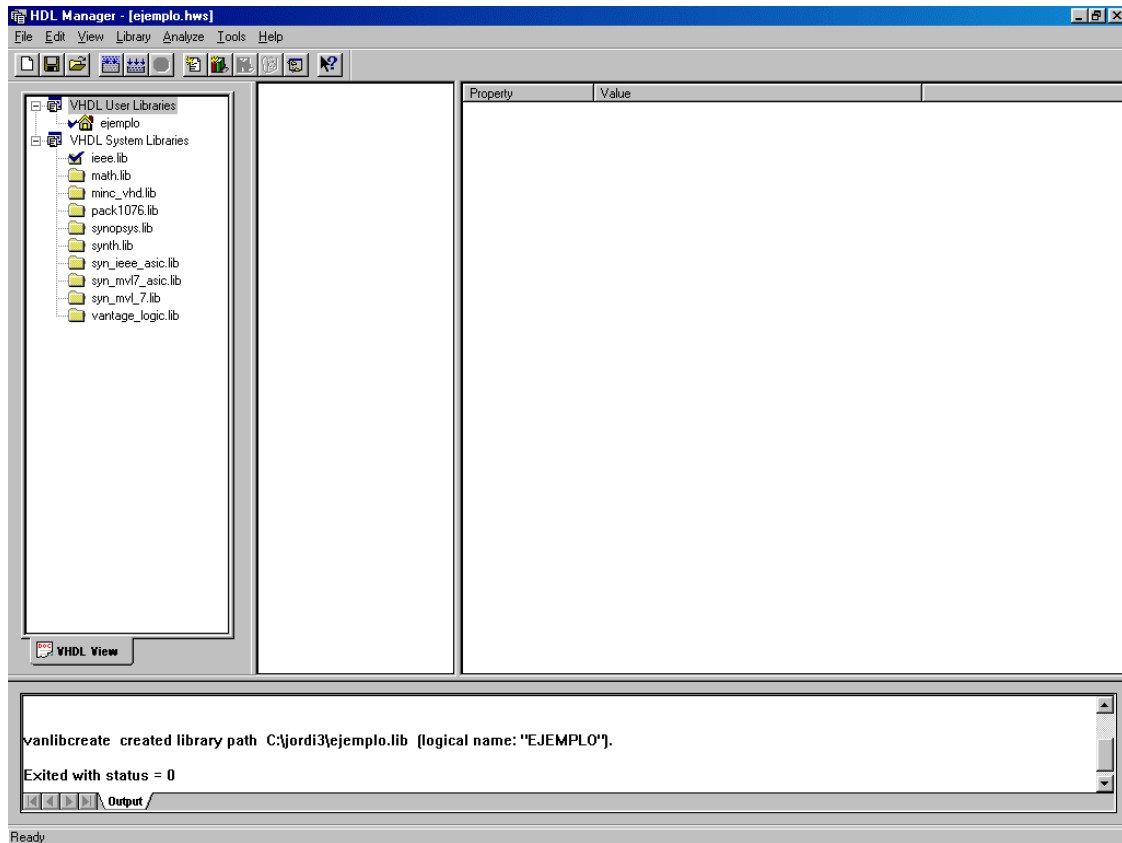


Figura 40. Menú 7

17) Ahora el siguiente paso será incluir los ficheros que nosotros queremos compilar.

18) Lo haremos de la siguiente manera, iremos a la barra de menús y seleccionaremos **Library > Add source files**, como aparece en la siguiente figura. **Figura 41.**

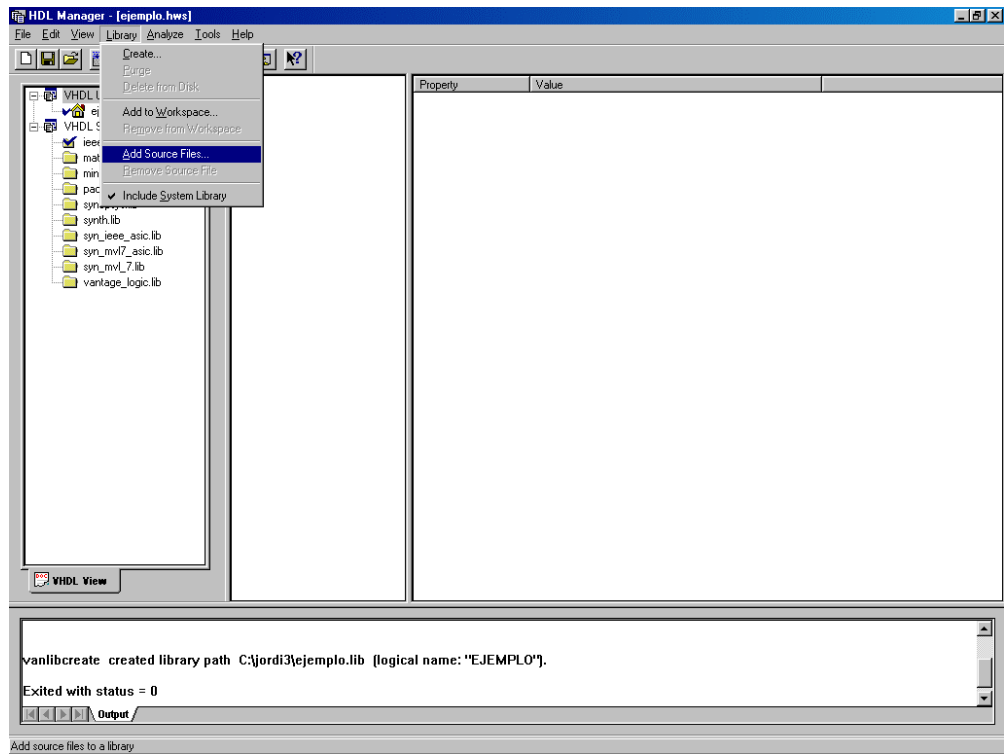


Figura 41. Menú 8

Al hacer click nos aparecerá la siguiente figura. **Figura 42**

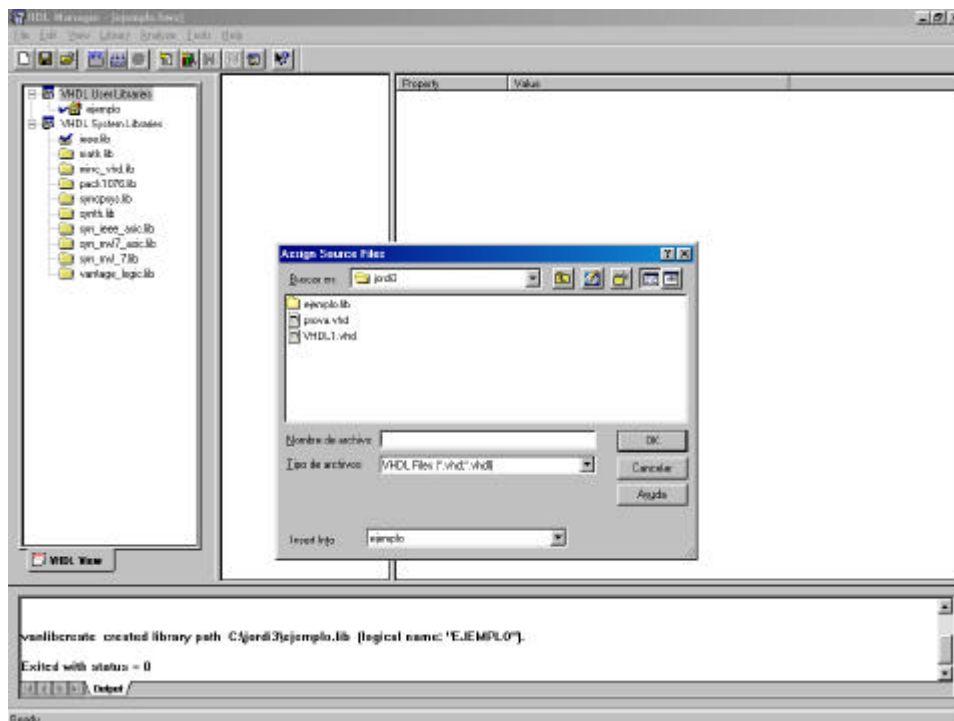
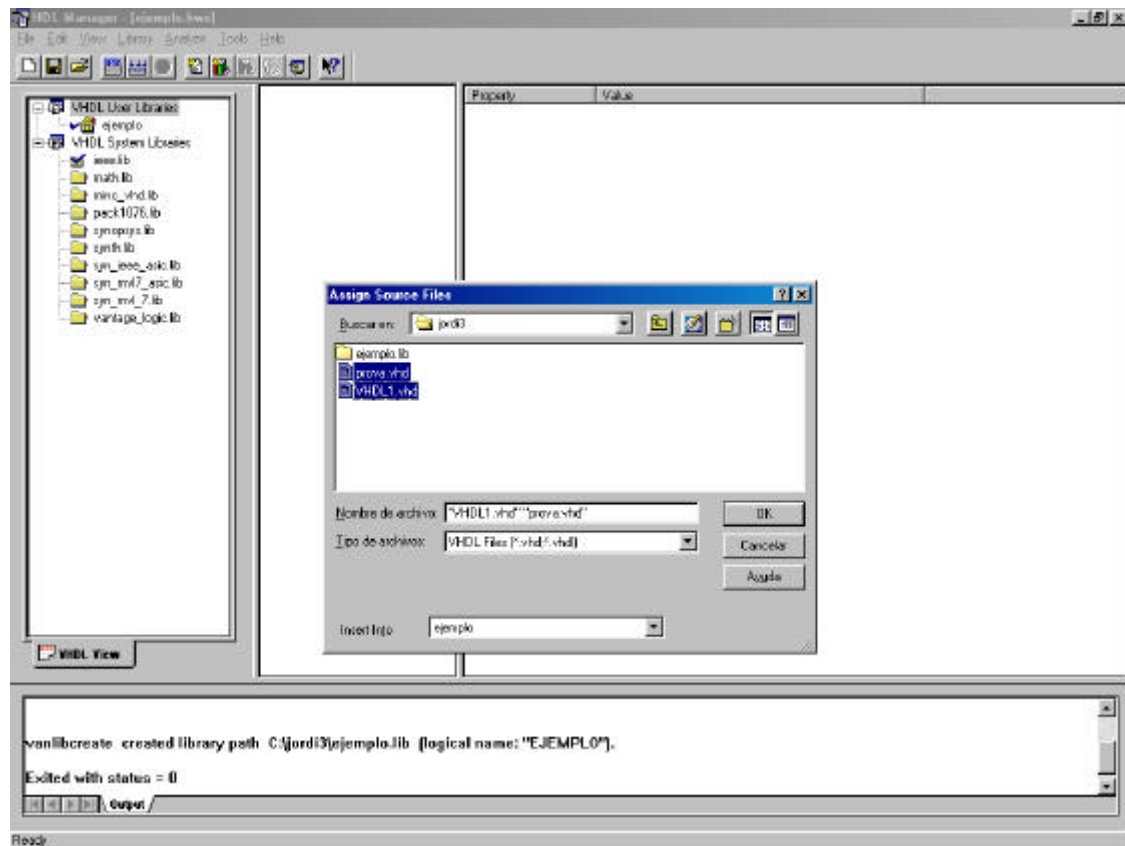


Figura 42. Menú 9



19) En la figura anterior **Figura 42** seleccionamos los ficheros VHDL que queremos compilar, en mi caso pondré dos uno el programa en sí de la ALU, y el otro que será el fichero de simulación. En la próxima figura , **figura 43** podremos observar que he seleccionado los ficheros.



**Figura 43.** Menú 10

En la figura anterior, **figura 43** cuando vemos que tenemos seleccionados los archivos, y se observa porque los tenemos de color azul, haremos un click en el botón OK, y entonces nos aparecerá la siguiente figura. **Figura 44.**

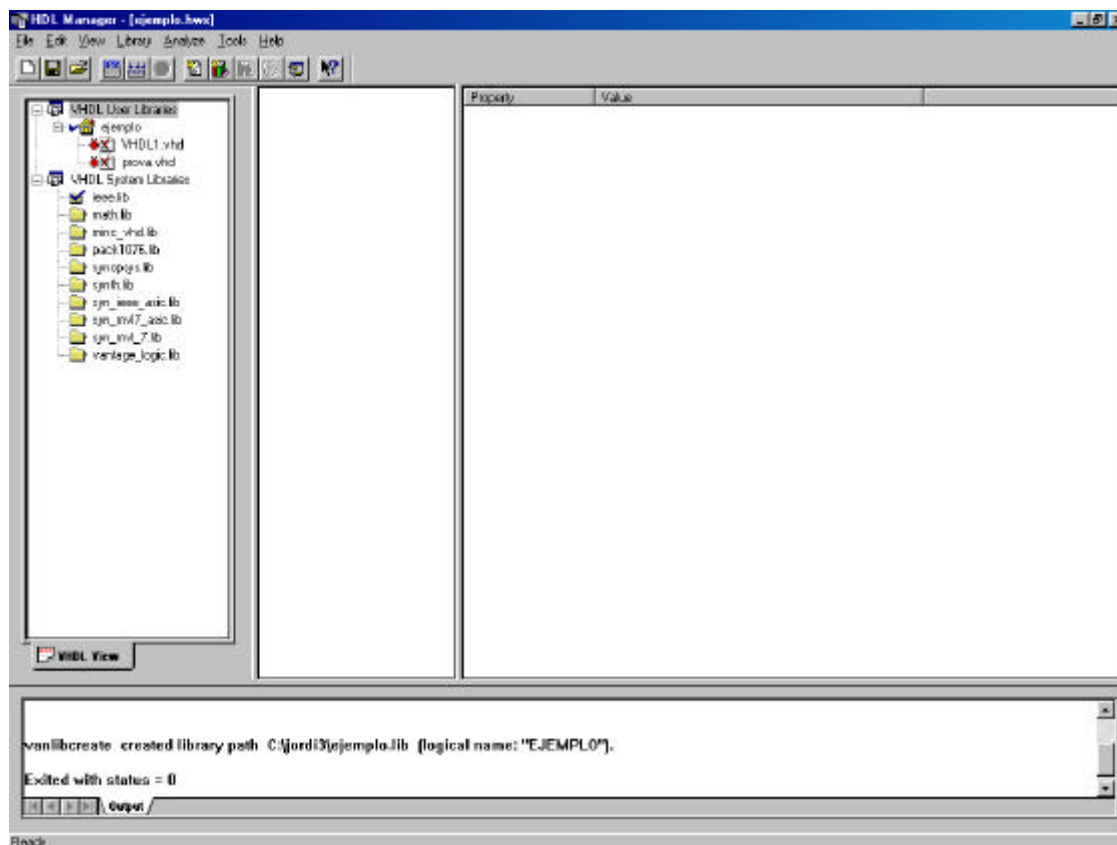


Figura 44. Menú 11

Como se puede observar en la figura anterior **figura 44**, en la parte izquierda hay un caseta dibujada llamada ejemplo y de ella cuelgan dos archivos con un punto rojo y tachados. Estos dos archivos, son los que he seleccionado anteriormente para después compilarlos. La señal x tachada de color rojo antes de los nombres de los archivos, significa que aún no los hemos compilado.

20) El siguiente paso será guardar los cambios **File > Save**.

21) Ahora compilaremos los ficheros de la siguiente manera. Nos ponemos encima de **ejemplo** en nuestro caso. Si hubiéramos puesto otro nombre nos aparecería el que hubiéramos puesto. Hacemos un **click** encima y entonces nos aparecerá de color azul tal y como se puede observar en la siguiente figura. **Figura 45**

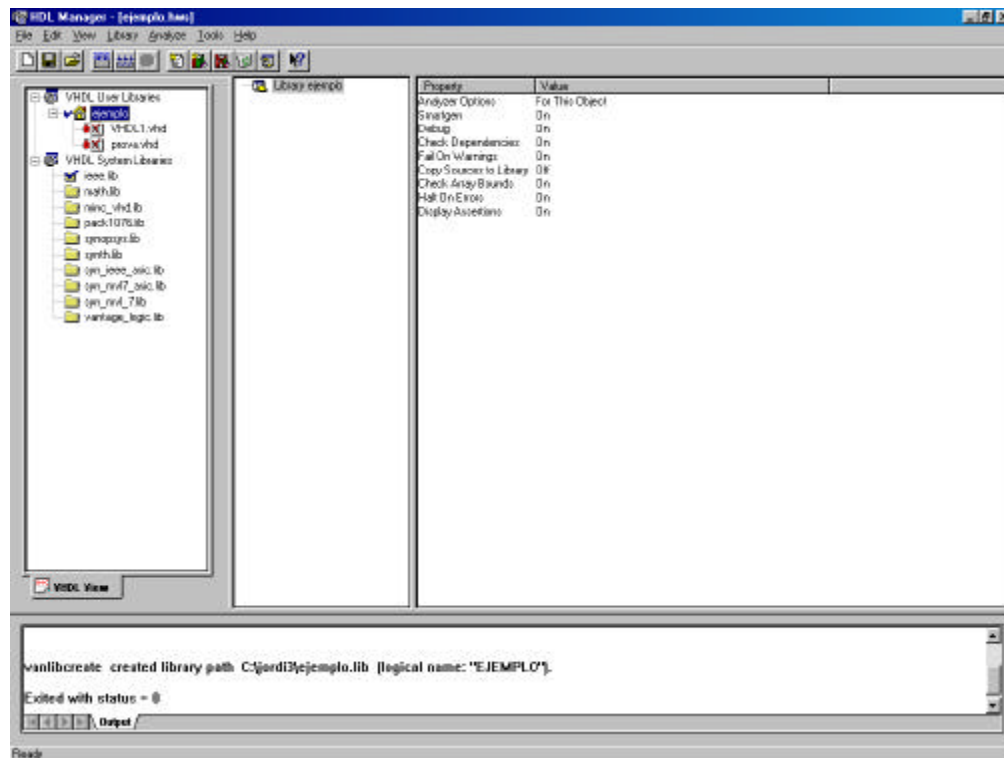


Figura 45. Menú 12

22) Entonces ahora tendremos que ir a **Analyze > Analyze User Library**. Figura 46

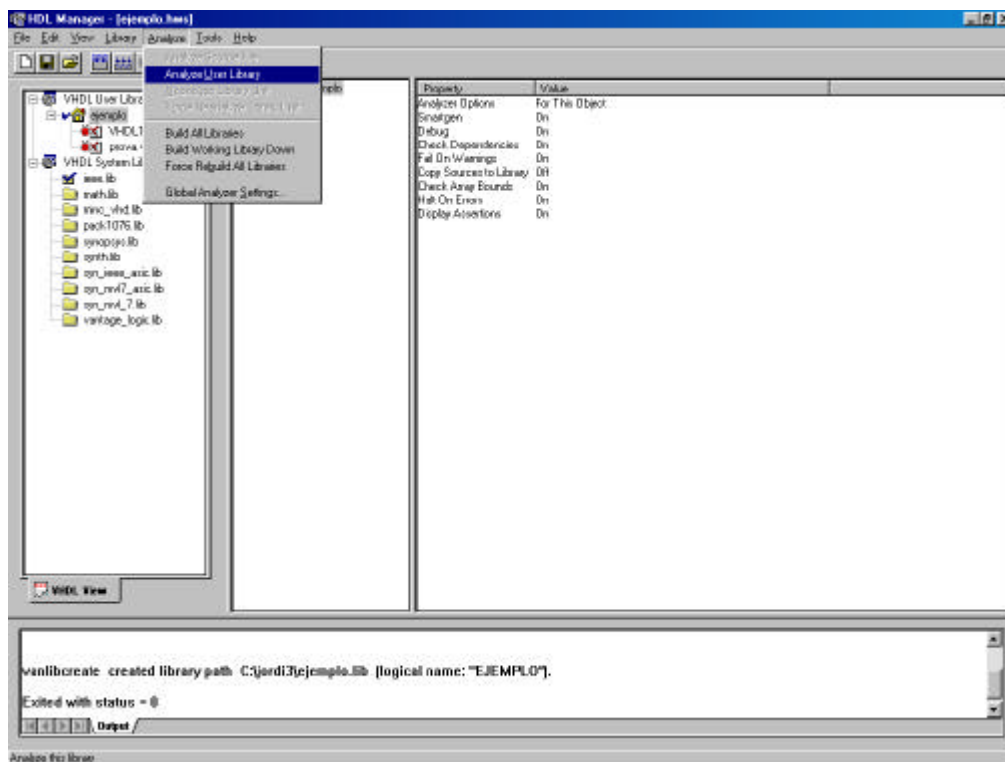
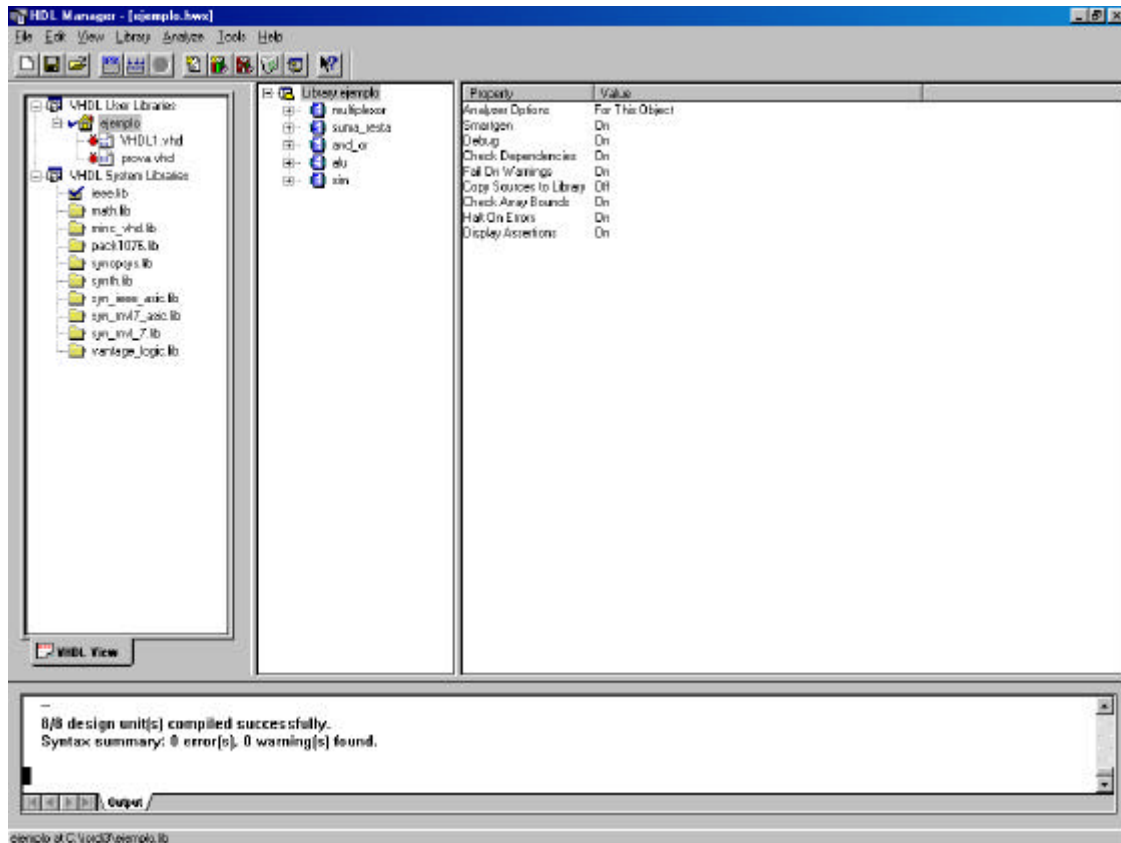


Figura 46. Menú 13

Con el paso anterior lo que hemos hecho es compilar los ficheros, si vemos como en este caso que no nos da error entonces, pasaremos al siguiente paso. Si por el contrario nos diera algún tipo de error tendríamos que corregirlo, y después volver a compilar.

23) Ahora una vez compilados los ficheros nos aparecerá como sigue. **Figura 47**



**Figura 47.** Menú 14

24) Una vez esto tenemos que guardar **File > Save**.

25) El siguiente paso será salir de este programa **File > Exit**.

Una vez hecho esto ya tenemos los ficheros compilados, ahora tenemos que pasar al siguiente paso que será la simulación de nuestro código VHDL. Ahora con el programa siguiente lo que haremos es sacar las gráficas de nuestro programa.

#### 4.1.2. Simulación de ficheros

Cuando hayamos cerrado el programa anterior, nos quedaremos en la ventana de **Fusion**, si no la tenemos abierta, la tendremos que volver a abrir, tal y como lo hemos hecho antes.

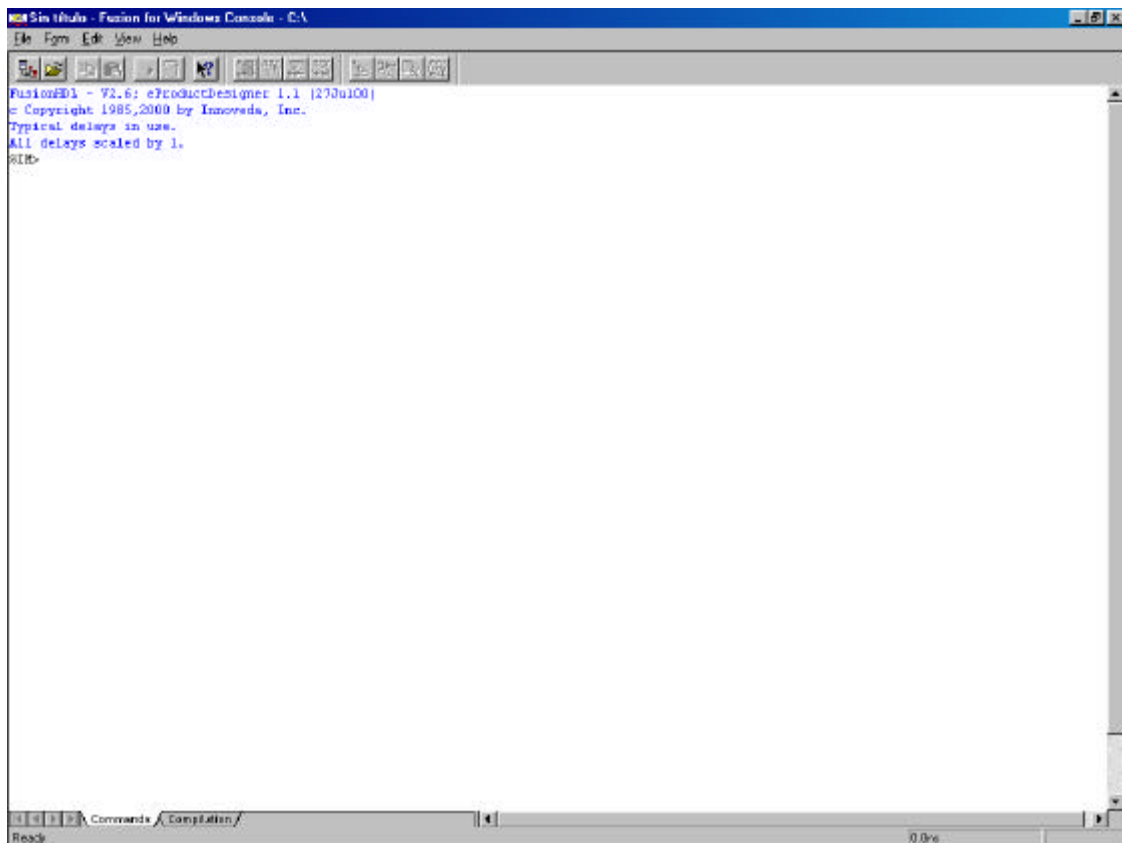
1) Abrir en programas **eProduct Designer**.

2) **Programable Design**.

3) Escogemos el icono siguiente

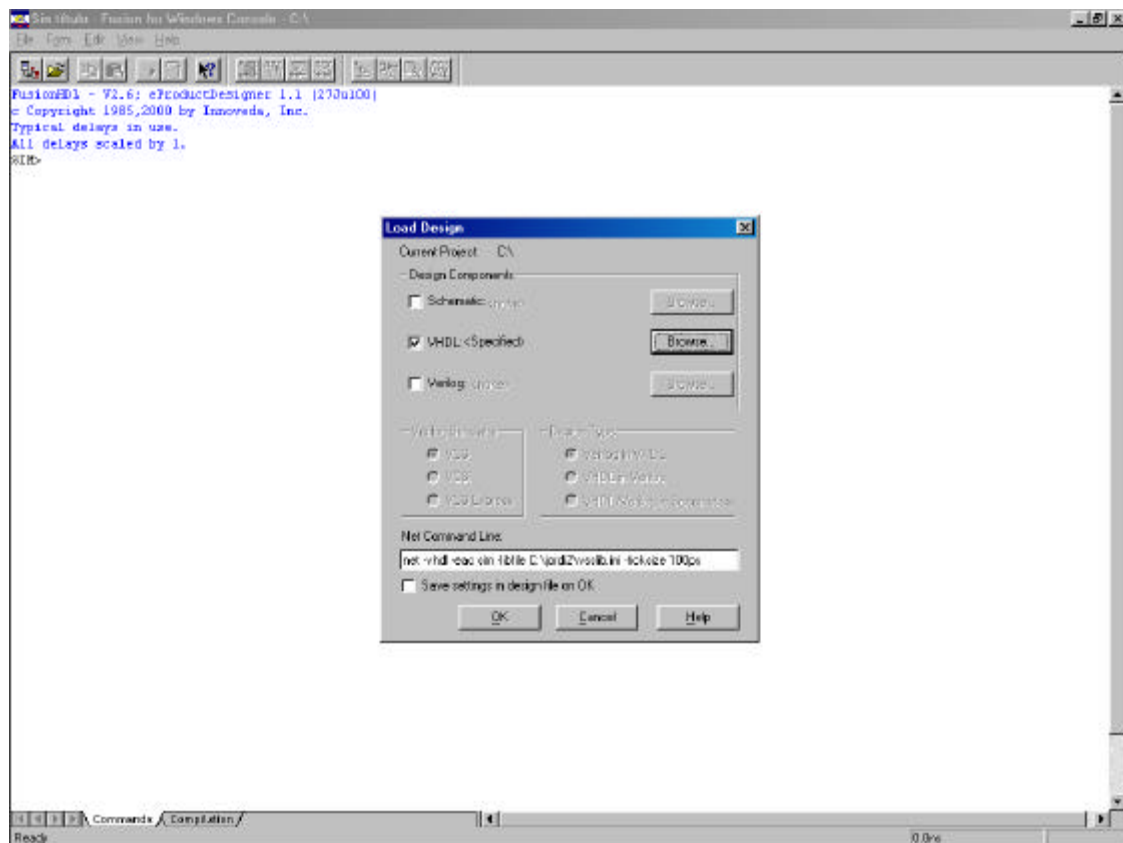


Al pulsar en este icono entonces nos aparecerá la siguiente figura. **Figura 48**



**Figura 48.** Menu 15

4) Ahora haremos **File > Load Design**, y nos aparecerá la siguiente figura. **Figura 49.**



7

Figura 49. Menú 15

5) Como nosotros queremos simular un fichero VHDL pondremos un ralla donde la vemos en la figura anterior, entonces nosotros haremos **click** en **Browse** y seleccionaremos el directorio donde nosotros tenemos los archivos que deseamos compilar, como se puede observar en la figura siguiente. **Figura 50**

Si en vez de querer simular un fichero VHDL, quisiéramos simular un fichero en formato *schematico* o un fichero *Verilog* entonces tendríamos que poner un cruz donde sea necesario.

Como nuestro caso es VHDL la pondremos ahí y entonces nos aparecerá la siguiente figura. **Figura 50.**

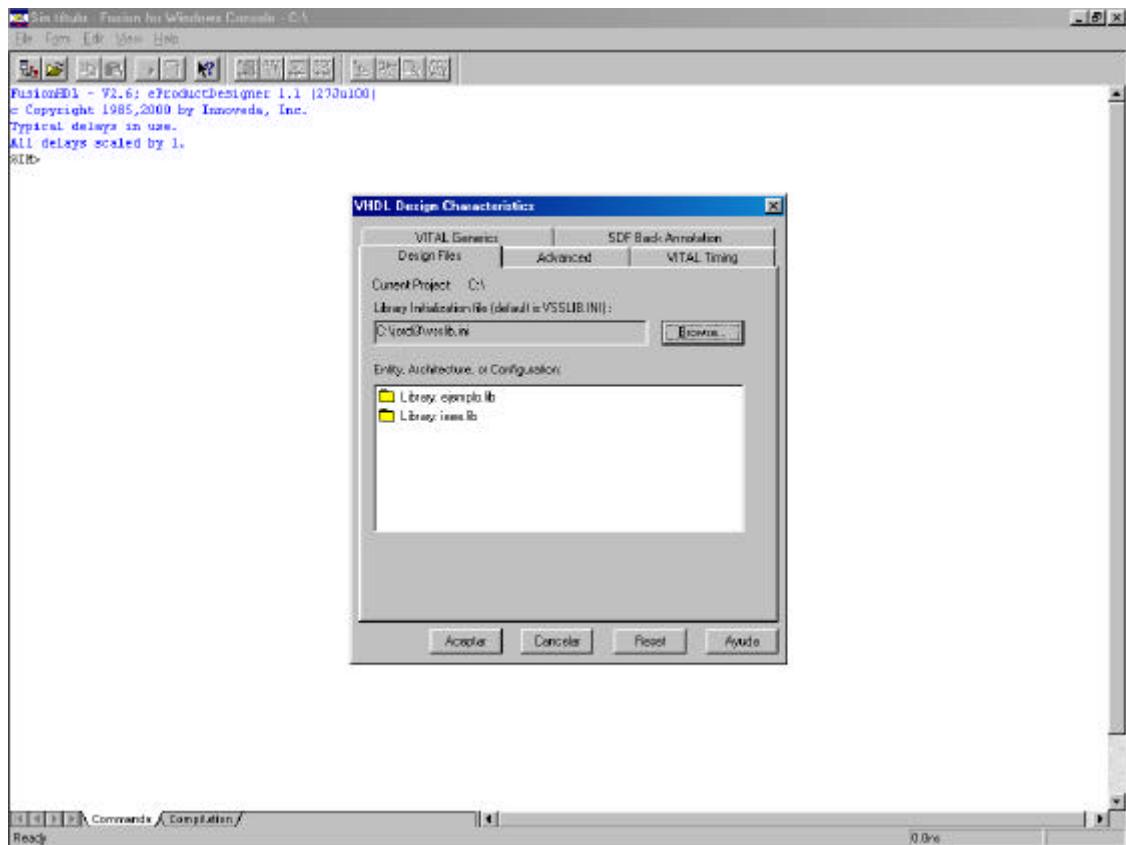


Figura 50. Menú 16

6) Ahora tendremos que escoger la entidad de nuestro programa, que en este caso estará dentro de **Library ejemplo.lib**. haremos **click**, y se nos abrirá la carpeta, dentro de esta carpeta, estarán todas las entidades, seleccionaremos la de mayor jerarquía. Como se puede apreciar en la figura siguiente. **Figura 51**

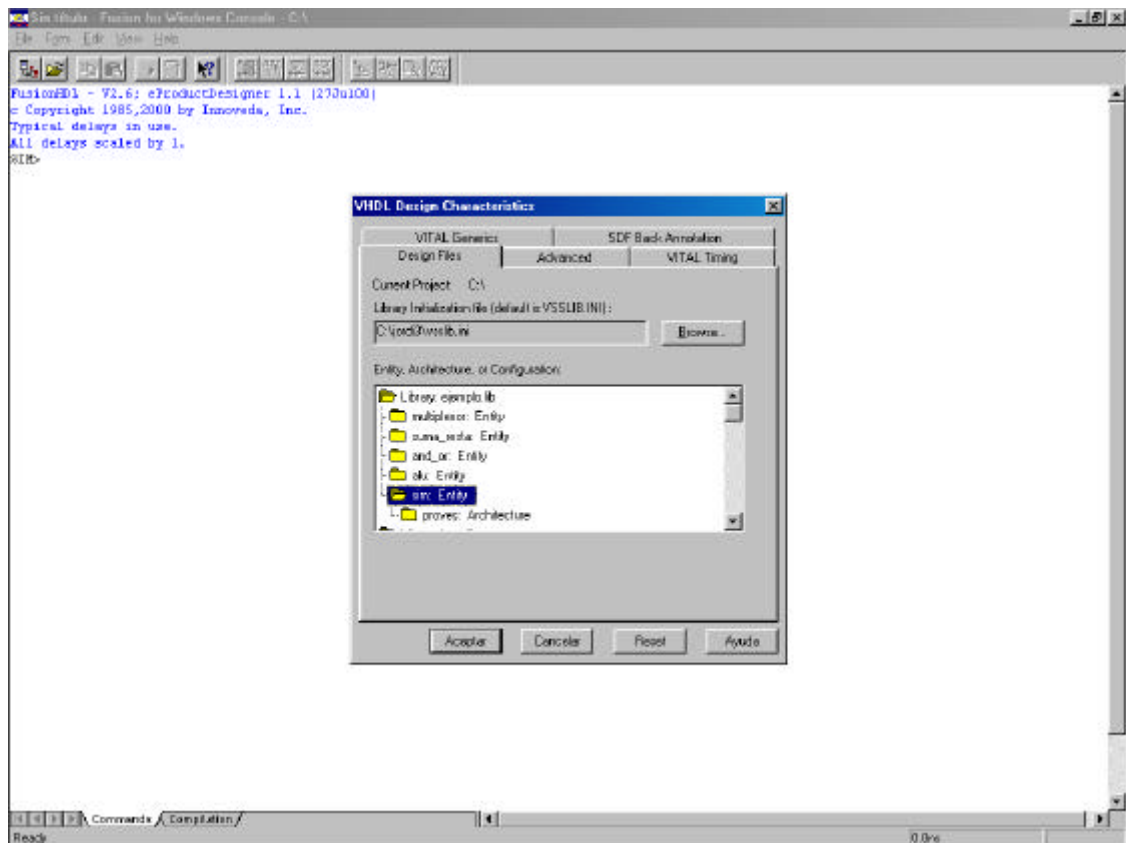


Figura 51. Menú 17

7) Aquí ahora una vez seleccionada le daremos a **Aceptar**, nos volveremos a la **figura 48**, pero entonces ya tendremos seleccionada nuestra carpeta y también el nombre de la entidad que en este caso la de mayor jerarquía es **sim** que es la entidad de simulación. Como volvemos a la **figura 48** no la volveremos a poner ahora, lo que tendremos que hacer cuando volvamos a estar en la **figura 48**, es ponerle **OK** y entonces nos aparecerá la siguiente figura. **Figura 52**.



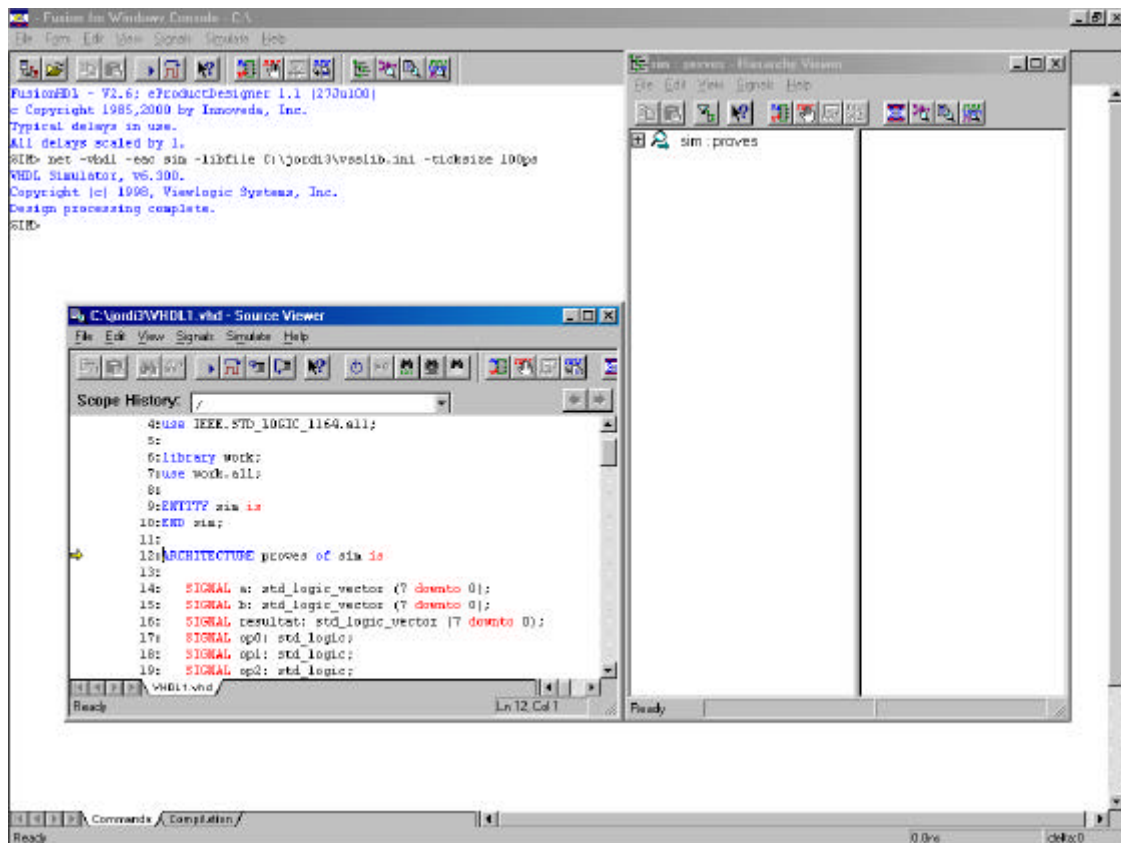


Figura 52. Menú 18

8) Una vez aquí nos ponemos en la ventana **Hierachy Viewer**, y le damos con el botón derecho del ratón **click** a el cuadrado que aparece con un signo +, y entonces nos aparecerá la siguiente figura. **Figura 53**

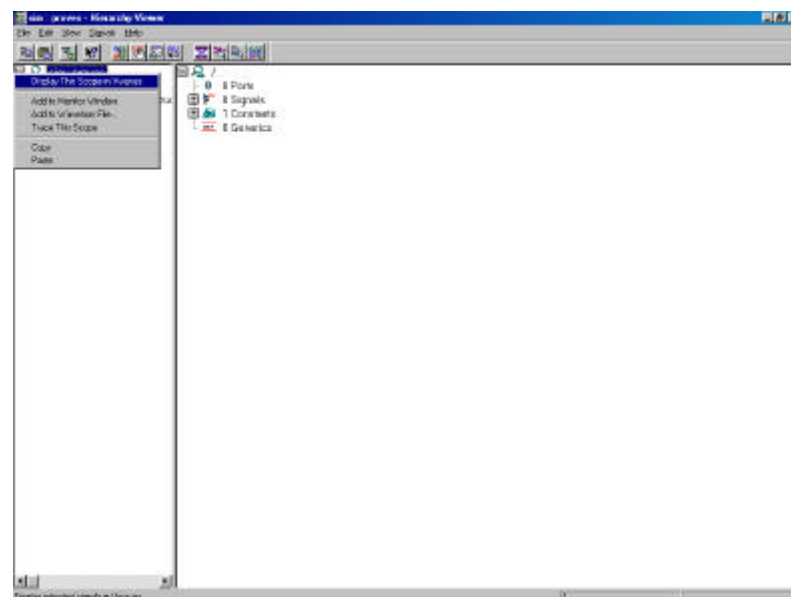
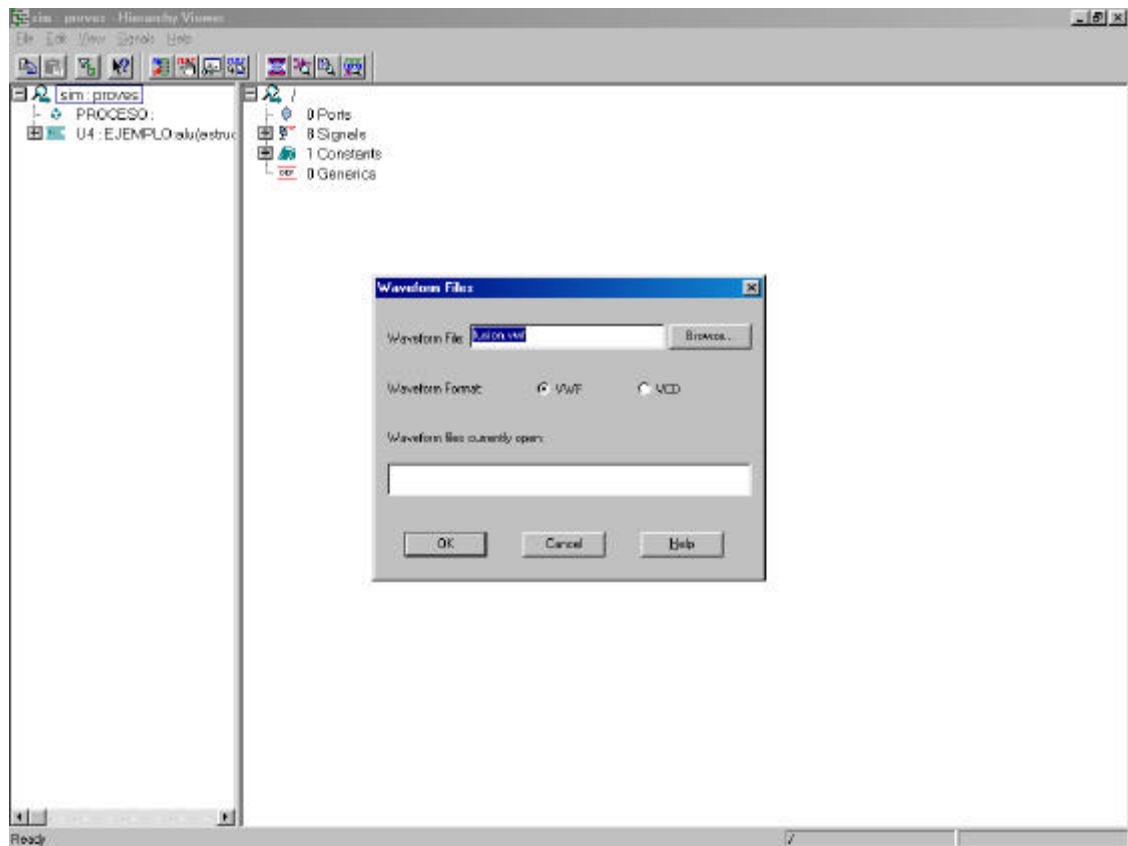


Figura 53. Menú 19

9) Seleccionaremos la que hemos visto seleccionada en la **figura 53. Display This Scope in Vwave**, y nos aparecerá la siguiente figura. **Figura 54**



**Figura 54.** Menú 20

10) Le daremos **click** al **OK**, y entonces el programa nos abrirá automáticamente el **Vwaves**. Tendremos la siguiente figura. **Figura 55**

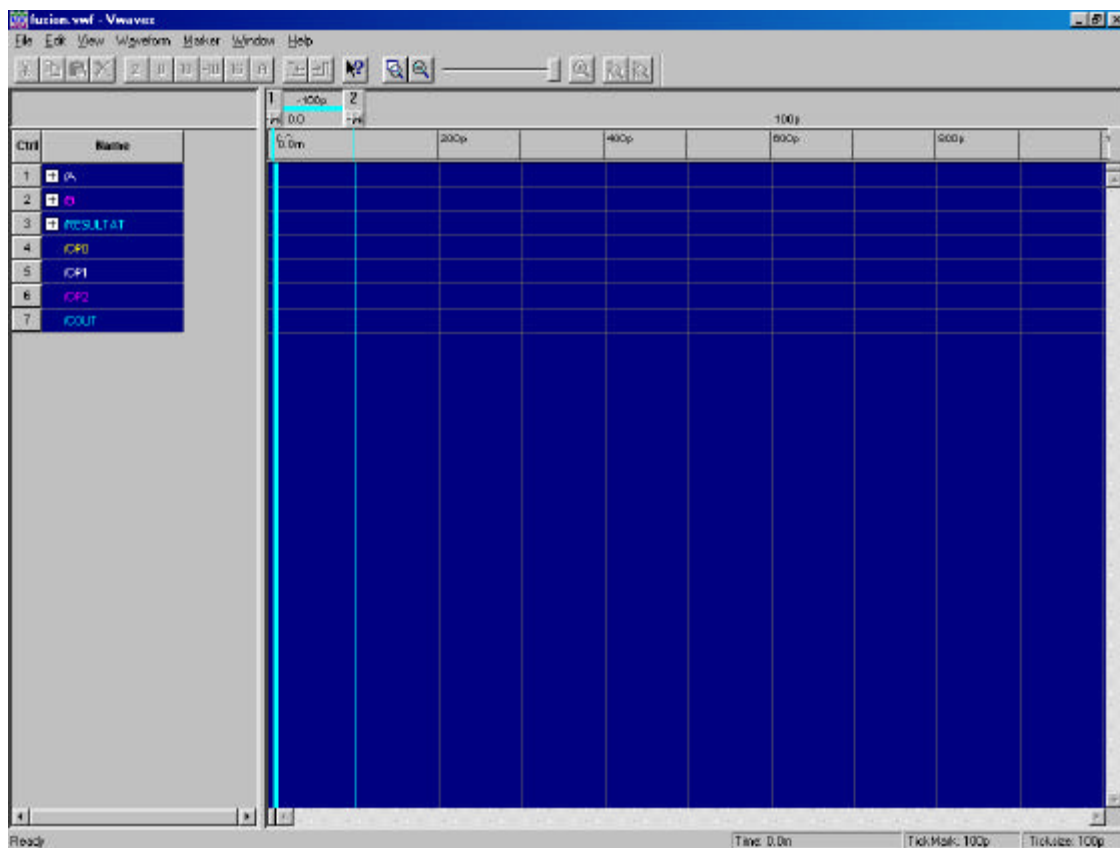
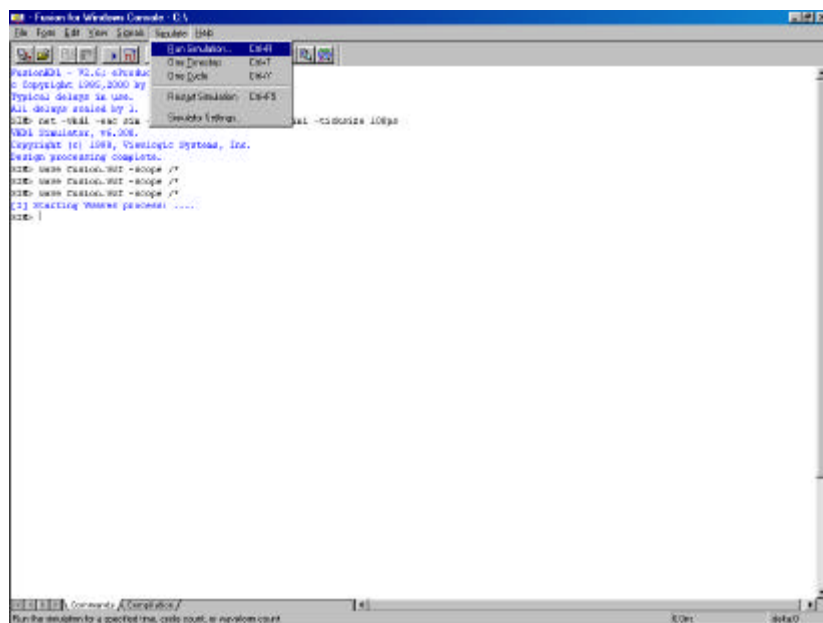


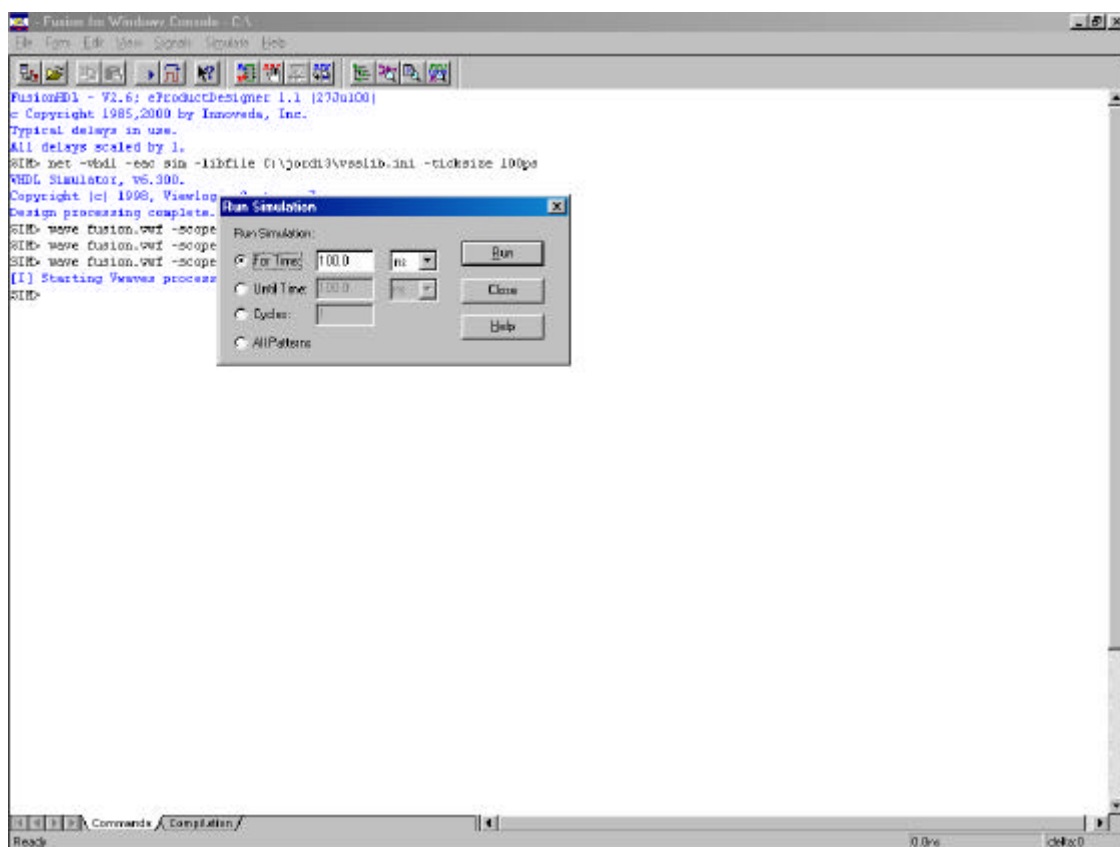
Figura 55. Menú 21

11) Ahora como podemos observar en la izquierda tenemos el nombre de nuestras señales. Ahora lo que tenemos que hacer es que nos salgan las señales dibujadas, y lo haremos de la siguiente manera.

12) Nos iremos al programa **Fusion** que tenemos abierto y seleccionaremos **Simulate > Run** como podemos observar en la siguiente figura. **Figura 56**

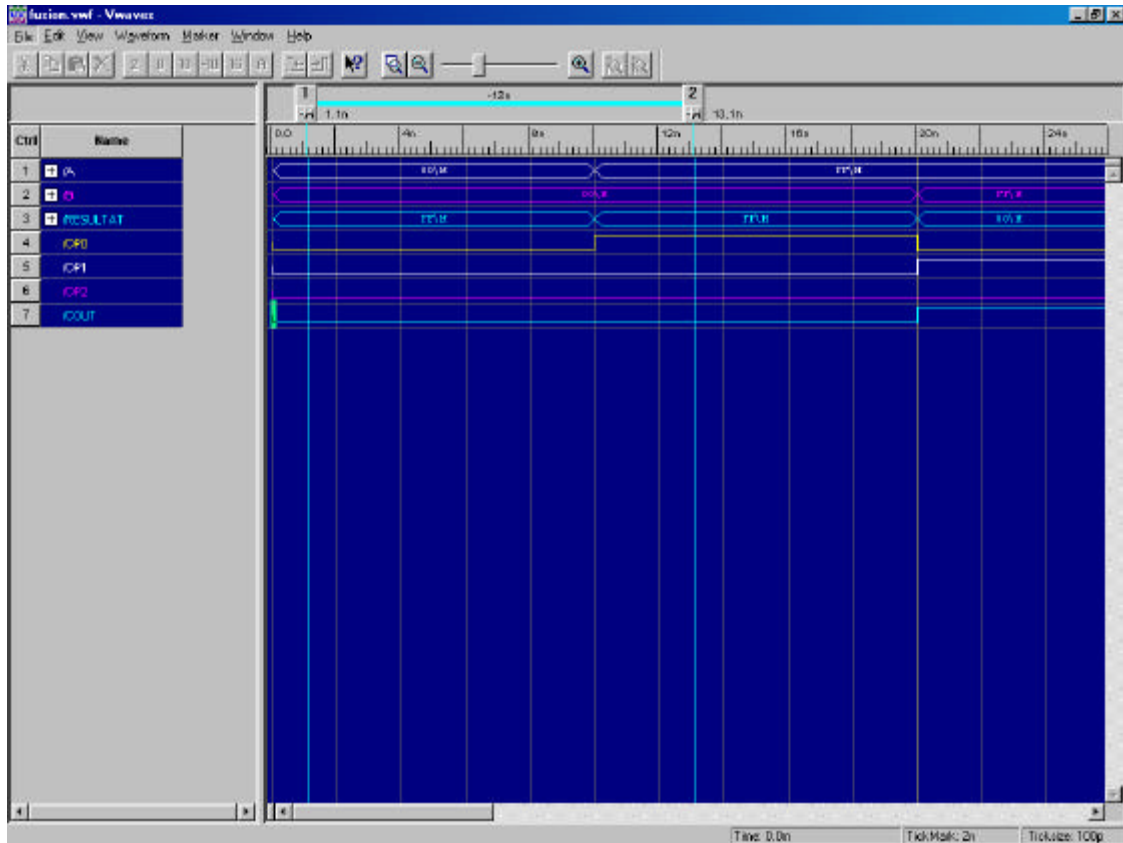


Entonces nos aparecerá la siguiente figura. **Figura 57**



13) En la figura anterior **figura 57** nos pone el tiempo que queremos simular nuestro código, en este caso pondremos 100ns y le daremos al botón **Run**

14) Cuando ya hemos dado al botón Run nos tenemos que ir al programa que tenemos abierto **Vwaves**, y allí observaremos que ya tenemos las señales, como se muestra en la siguiente figura. **Figura 58.**



**Figura 58.** Menú 24

En la izquierda que aparecen las señales, hay unas que aparecen con un signo +, esto quiere decir que son vectores de bits, si queremos ver cada uno de los bits por separado haremos click encima, entonces todas las señales y nos quedará el dibujo como el de la figura siguiente. **Figura 59.**

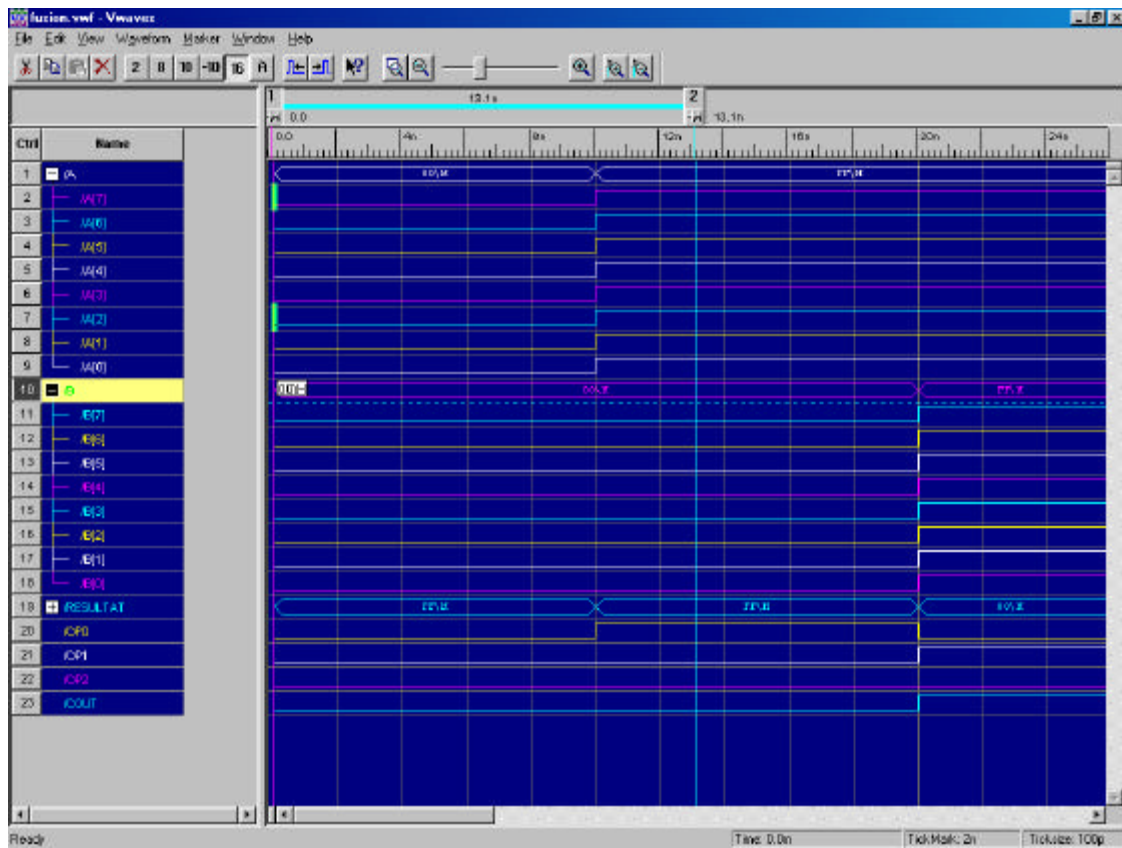


Figura 59. Menú 25

Ahora ya hemos podido observar como simular con este programa. Ahora solo nos quedará hacer la síntesis, que se verá a continuación.

#### 4.1.3. Sintetizar archivos

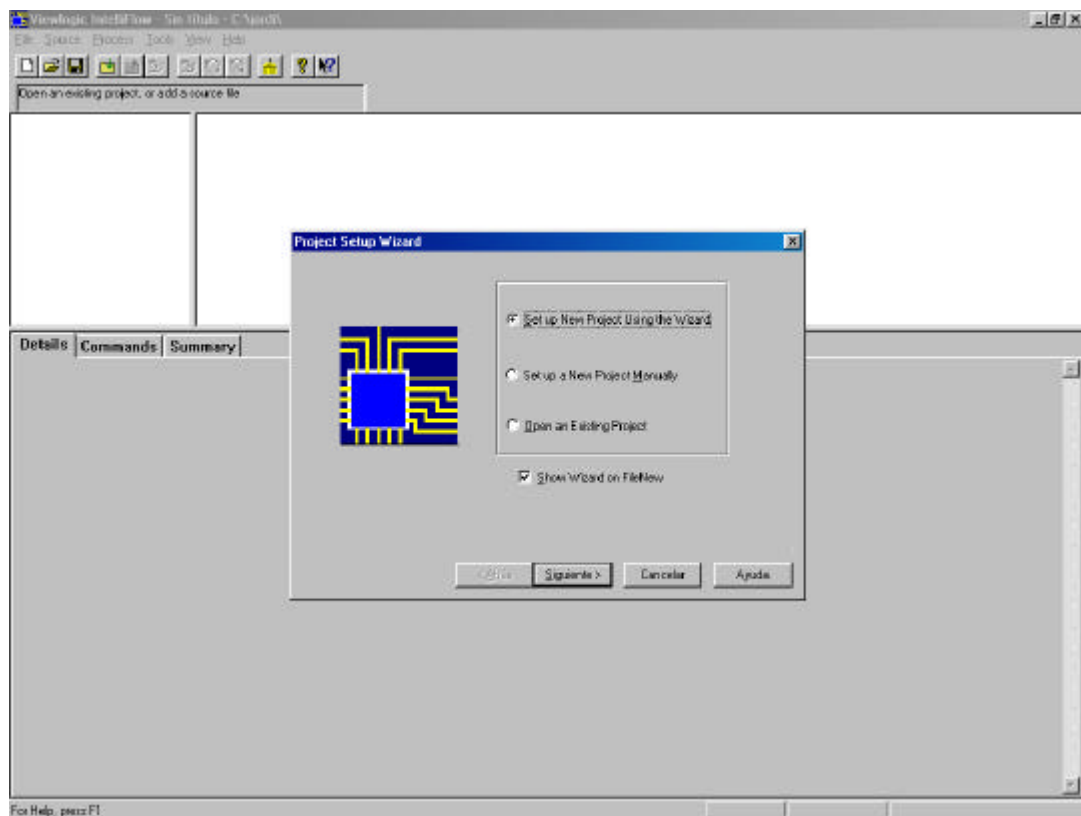
Par sintetizar archivos seguiremos los siguientes pasos:

- 1) Abriremos **eProduct Designer**.
- 2) Seguidamente **Programable Design**.

- 3) Ahora Intelliflow



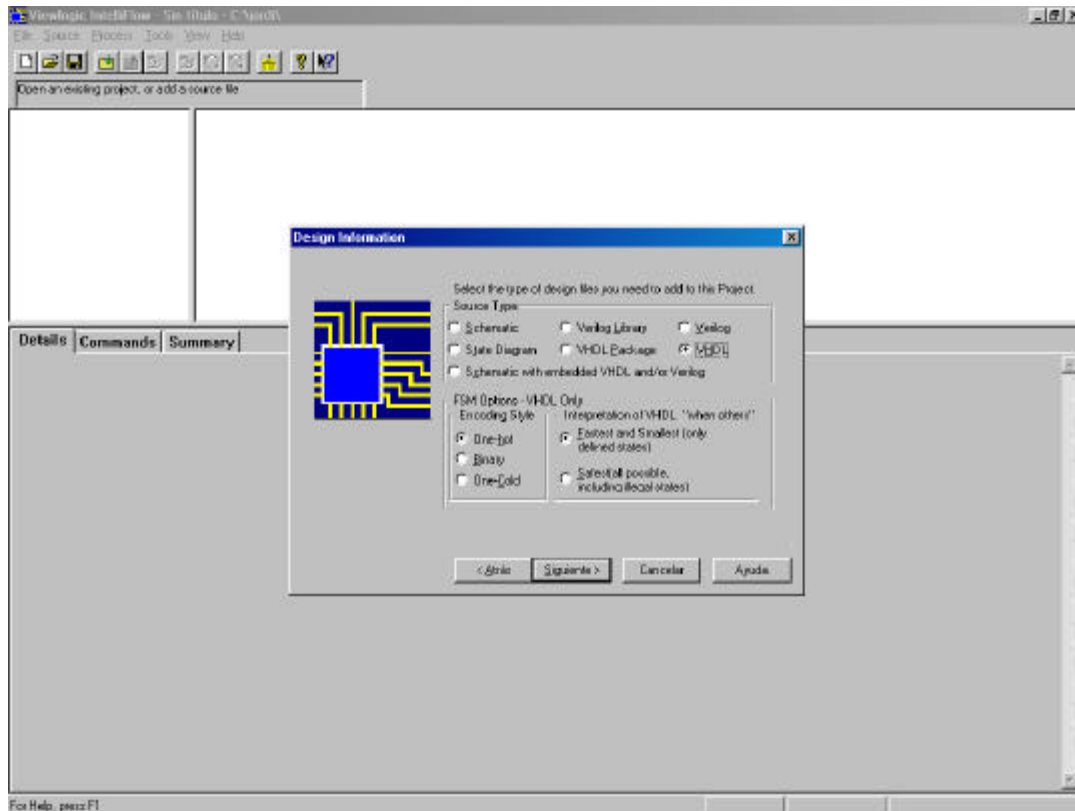
- 4) Ahora se nos abrirá la siguiente ventana. **Figura 60**



**Figura 60.** Menú 26

5) Seleccionaremos **Set Up New Project Using The Wizard** de la **figura 60** anterior.

6) Le daremos al botón siguiente. Nos saldrá el siguiente menú. **Figura 61.**



**Figura 61.** Menú 27

7) Ahora seleccionaremos VHDL de la **figura 61** y también le daremos al botón siguiente.

8) Ahora nos aparecerá otro menú dónde tendremos que escoger el tipo de chip que tenemos que utilizar, le pondremos el que nosotros tengamos que utilizar. En este caso es para el fabricante LATTICE. Entonces nos aparecerá la siguiente figura. **Figura 62.**



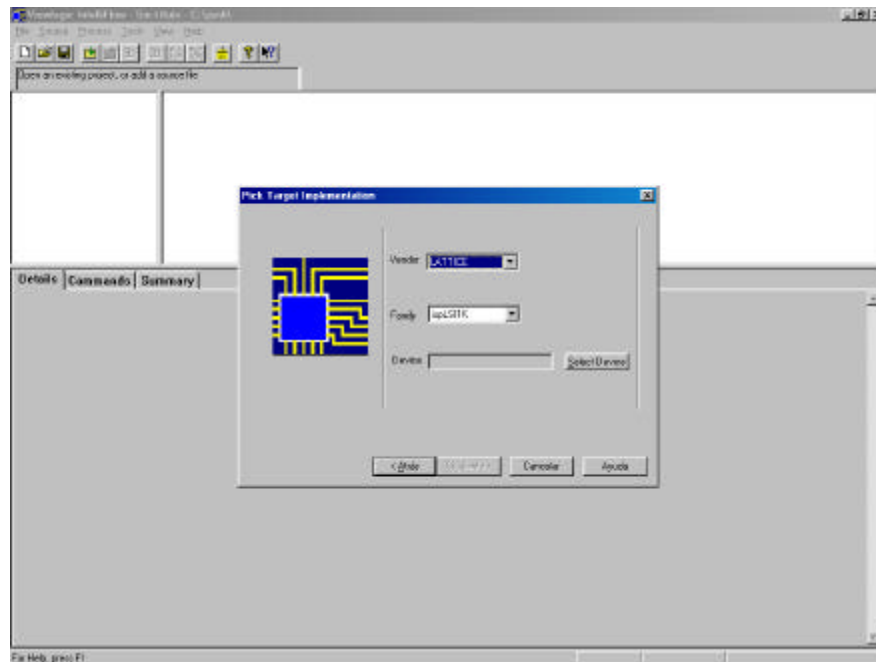


Figura 62. Menú 28

9) En la **figura 62** donde pone **select device** haremos click y entonces entraremos en un menú y seleccionaremos el dispositivo que queremos, en nuestro caso el 1024 Ahora seguidamente haremos **click** en **Siguiente**, nos aparecerá la siguiente figura. **Figura 63**

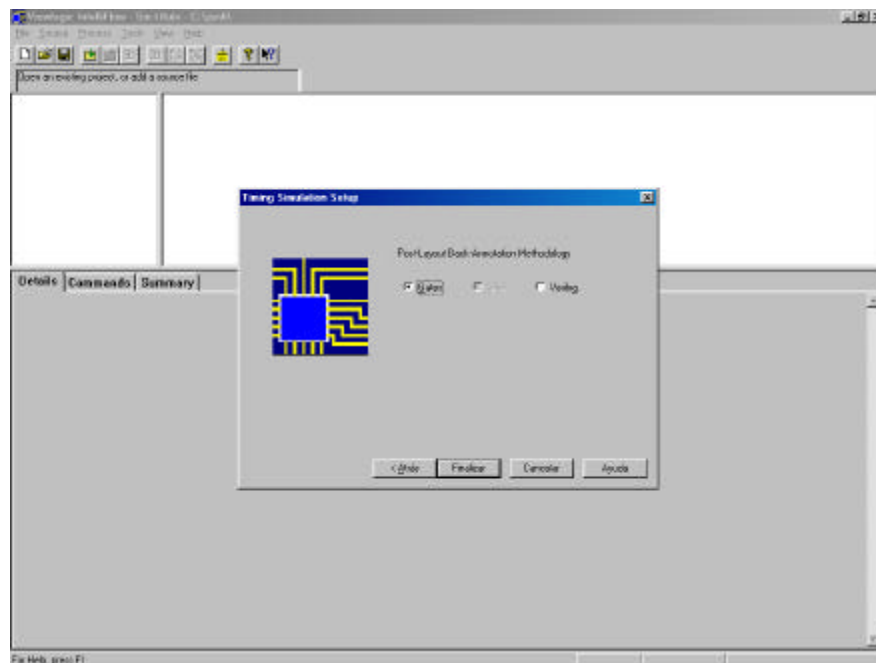
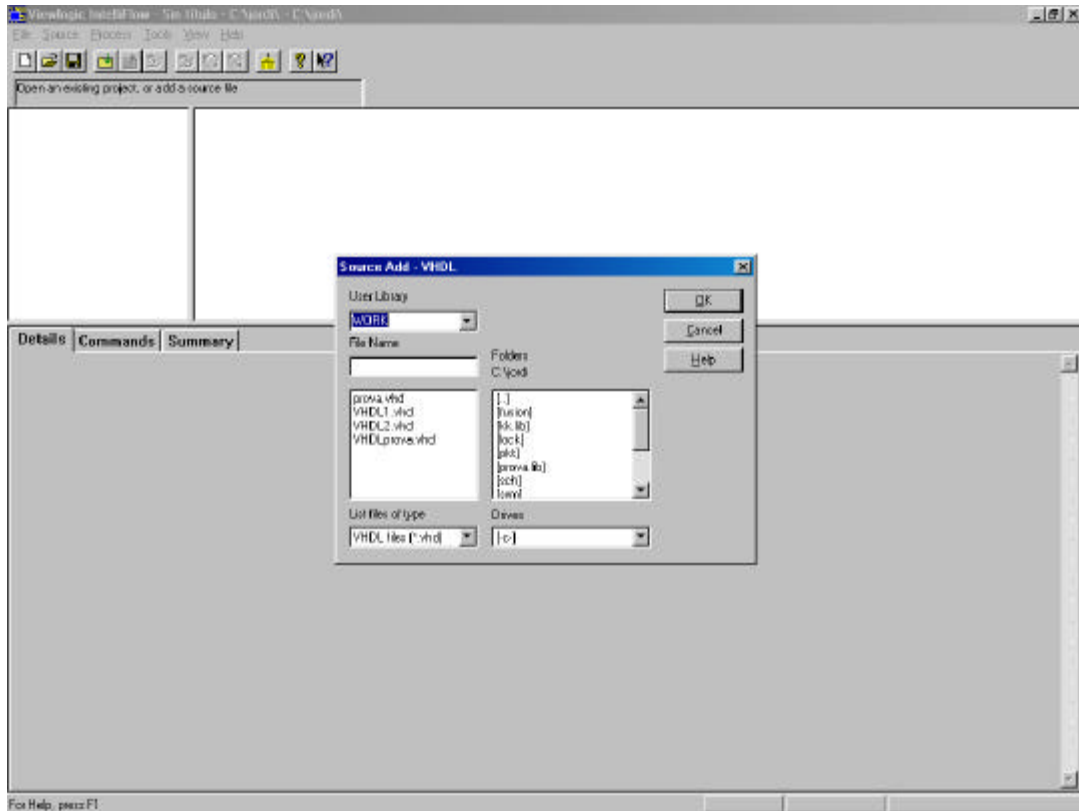


Figura 63. Menú 29

10) Seleccionaremos **Gates** en la **figura 63** que quiere decir puertas y le daremos al botón **Finalizar**

11) Ahora en el siguiente menú tendremos que ponernos en el directorio donde tenemos los archivos VHDL, y seleccionar el archivo que tengamos que sintetizar. Será la figura siguiente. **Figura 64.**



**Figura 64.** Menú 30

12) Lo que tendremos que hacer en la **figura 64** es poner el fichero que queremos sintetizar. Solo se tiene que sintetizar el fichero de la ALU, y no el de simulación, lo que tenemos que hacer es ir a nuestro directorio, elegir el fichero y cuando lo tengamos elegido, le damos al botón OK. Entonces nos aparecerá la siguiente figura. **Figura 65.**

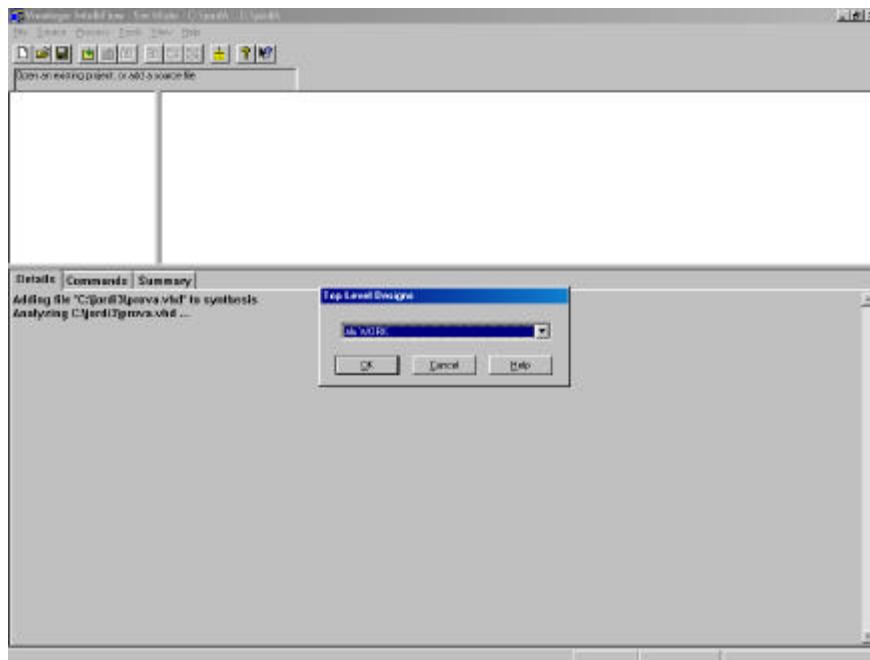


Figura 65. Menu 31

13) Ahora tenemos que seleccionar la entidad de mayor nivel que en nuestro caso será **alu WORK** y le daremos al botón **OK**, con lo cual nos quedará la siguiente figura. **Figura 66.**

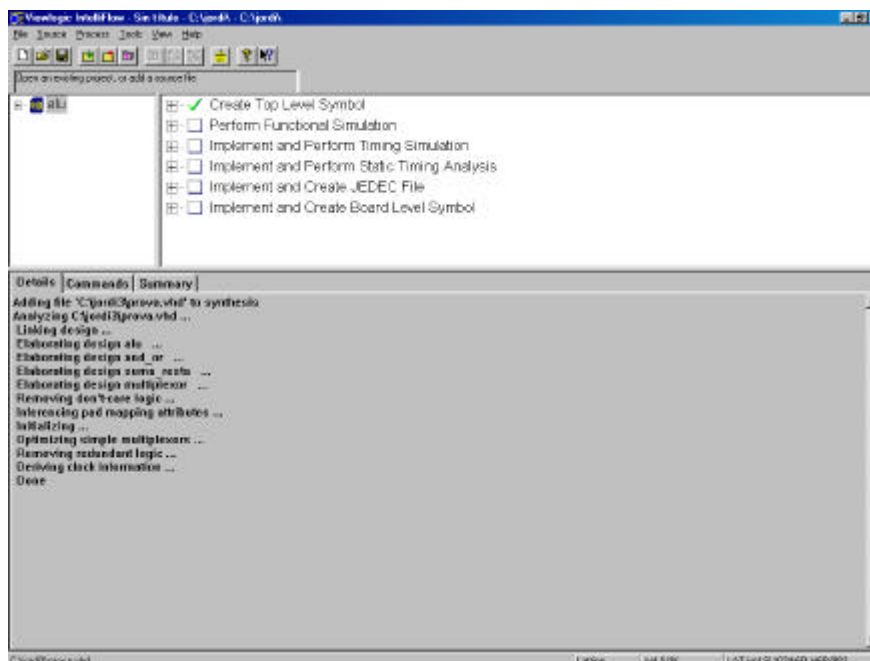
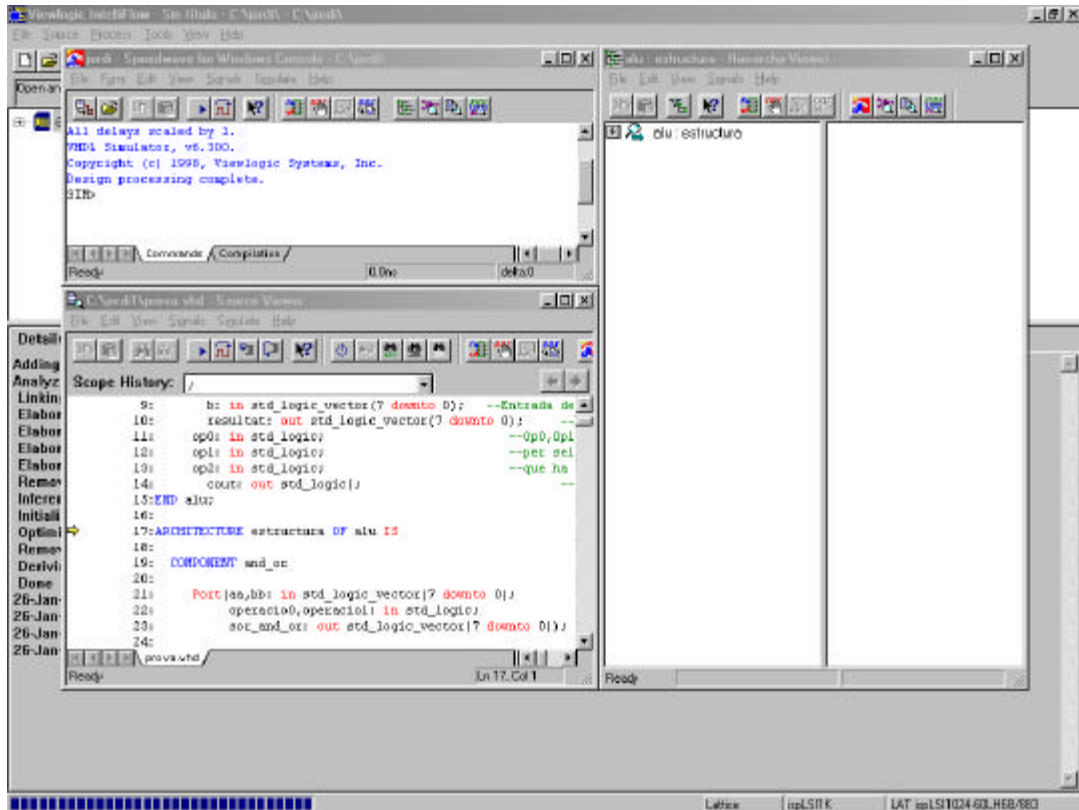


Figura 66. Menú 32

14) Ahora lo que tendremos que hacer es ir haciendo los **clicks** a cada una de las opciones como iremos viendo a continuación hasta que terminemos. En la siguiente le daremos a **Perform Functional Simulation**, y si todo va bien nos tienen que salir como arriba el símbolo este verde que quiere decir que la cosa está bien, y si no tendríamos que arreglar lo que nos dijera. Ahora al hacer nos quedará la siguiente figura. **Figura 67.**



**Figura 67.** Menú 33

El programa lo que nos ha hecho es abrirnos una serie de ventanas, ya que este programa puede hacer muchas cosas, el te las abre por si quieres utilizar las distintas opciones que tiene.

Lo que tendremos que hacer es cerrar todas las ventanas estas que nos aparecen encima de la ventana principal, y solo quedarnos con la principal donde podremos observar que nos a puesto el símbolo de correcto, esto nos dice que podremos pasar a la siguiente. Tal y como se puede observar en la siguiente figura. **Figura 68.**



16) Ahora haremos **Implement and Perform Static Timing Analysis**. Tendremos la siguiente figura. **Figura 70.**

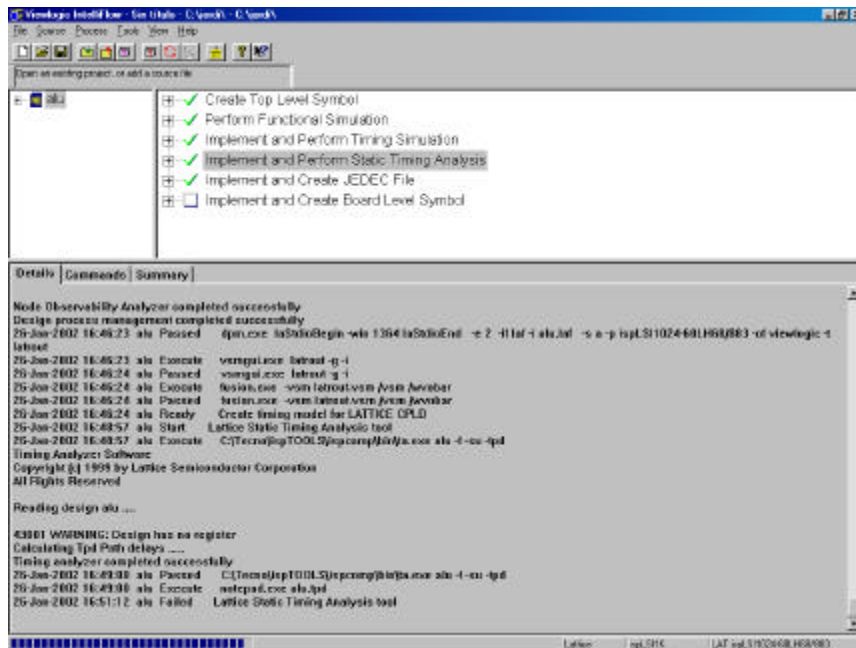


Figura 70. Menú 36

Cuando esto nos lo haya hecho también nos abrirá un fichero que son los retardos que tiene el programa después de sintetizarlo. Es un fichero como el de la siguiente figura. **Figura 71.**

The screenshot shows the 'Tpd Path Report' window. The report contains the following information:

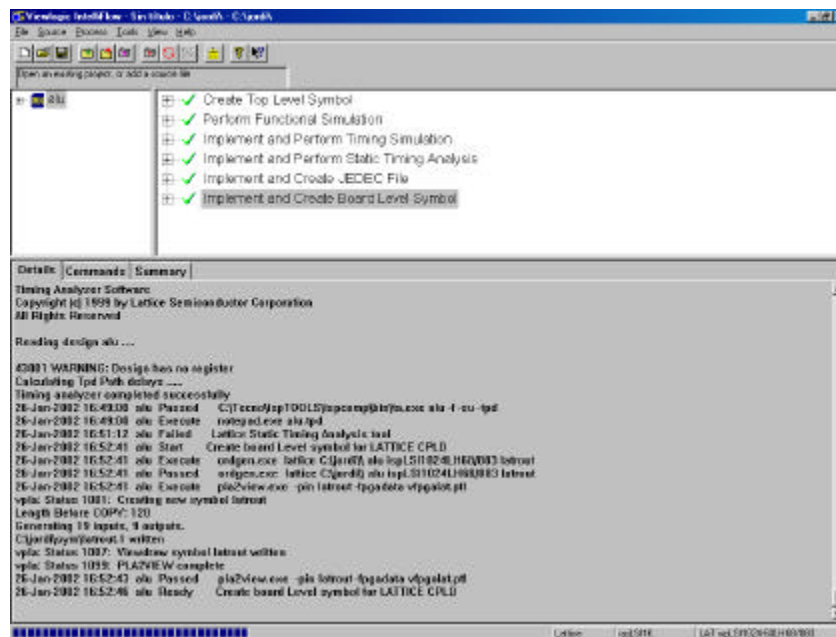
Design Name: ALU  
Part Name: ispLS1024-60LH60/803  
This report contains the path delays between the primary inputs and primary outputs of the design with no register in the path.

Tpd Paths:

Source (Input)	Destination (Output)	Path Delay [ns]
OP0	RESULT_0	26.30
OP1	RESULT_0	25.50
O_0	RESULT_0	25.40
A_0	RESULT_0	25.20
OP2	RESULT_0	24.00
OP0	RESULT_7	125.10
OP0	RESULT_6	109.20
OP0	RESULT_5	95.30
OP0	RESULT_4	85.30
OP1	RESULT_3	82.50
OP1	RESULT_2	69.80
OP2	RESULT_2	66.30
O_0	RESULT_2	67.50
OP0	RESULT_2	55.70
OP1	RESULT_2	54.90
OP2	RESULT_2	50.20
O_0	RESULT_2	50.40
A_0	RESULT_2	52.60
OP0	RESULT_1	39.60
OP1	RESULT_1	38.00
OP2	RESULT_1	38.10
O_0	RESULT_1	37.30
A_0	RESULT_1	36.50
O_1	RESULT_1	34.70
A_1	RESULT_1	25.20

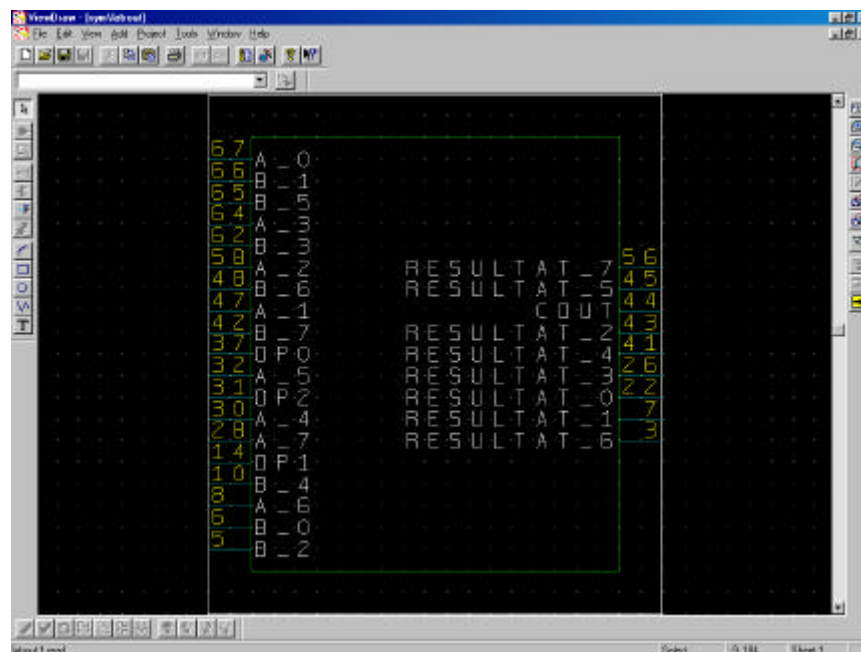
Figura 71. Menu 38

17) Por último haremos el **Implement and Create Board Level Symbol**, y nos aparecerá la siguiente pantalla **.Figura 72**



**Figura 72. Menú 39**

Cuando hayamos hecho esto también nos abrirá un fichero donde nos pondrá la disposición de las patas de la FPGA, es decir donde están nuestras entradas y salidas. Será un fichero como el de la figura siguiente. **Figura 73**



**Figura 73.** Menú 40

## 5 Conclusiones

Las conclusiones que se pueden sacar después de haber hecho proyecto son, la primera y más importante que he aprendido un nuevo lenguaje de programación, que es muy interesante y por lo que he leído se utiliza mucho para programar circuitos digitales.

En mi proyecto he encontrado muy interesante sobretodo programar en RTL, transferencia entre registros, una vez tienes claros todos los esquemas combinacionales, es relativamente sencillo programar en VHDL cuando lo tienes claro.

Por lo que hace referencia al programa **eProduct Designer** ha sido un poco difícil de hacer funcionar, todo y que es un programa muy completo donde se pueden hacer multitud de cosas como por ejemplo puedes ponerle la disposición de patas que más te interesa y el te diseña el circuito para que tengas aquella disposición. Le puedes poner que te diga la máxima frecuencia a que funciona tu circuito etc. Todo esto antes citado no lo he hecho porque no era el objetivo de mi proyecto pero he visto que se puede hacer a parte de otras cosas que no he tenido ni tiempo de mirarlo pero que seguro que son muy interesantes.

En el circuito puedes simular antes de sintetizar, y después de sintetizar el circuito te genera otro fichero VHDL que puedes volver a simular y aquí observas los retardos obtenidos como consecuencia de la implementación en puertas, que es donde aparecen los retardos.

Lo que se tendría que hacer ahora es grabarlo a un micro y mirar si realmente funciona, pero esto no se ha podido realizar ya que la aplicación no funciona. Supongo que sería muy interesante poderlo hacer.

En definitiva ha sido una experiencia muy interesante desde los dos puntos de vista primero porque he aprendido a programar con un lenguaje distinto, y segunda porque he hecho funcionar un programa bastante completo aunque no he podido mirar todas las opciones que tiene, aunque puedo decir que tiene muchas.

Espero que haya sido un proyecto provechoso para la Universidad y para todas las personas que estén interesadas en este lenguaje y en este programa.



## Anexos

### Anexo 1. Programa VHDL de la ALU

```
Library ieee;
Use ieee.std_logic_1164.ALL;

Library work;
Use work.ALL;

ENTITY alu IS
Port( a: in std_logic_vector(7 downto 0);      --Entrada de 8 bits
      b: in std_logic_vector(7 downto 0);      --Entrada de 8 bits
      resultat: out std_logic_vector(7 downto 0); --Resultado de
                                              --la operación

      op0: in std_logic;                       --Op0,Op1,Op2 sirven
      op1: in std_logic;                       --para seleccionar la operación
      op2: in std_logic;                       --que queremos realizar a la ALU
      cout: out std_logic);                   --Carry de salida
END alu;

ARCHITECTURE estructura OF alu IS

    COMPONENT and_or                                --Bloque AND/OR

        Port(aa,bb: in std_logic_vector(7 downto 0); --Entradas del bloque
              operacio0,operacio1: in std_logic;      --operación a realizar
              sor_and_or: out std_logic_vector(7 downto 0)); --salida

    END COMPONENT;

    COMPONENT suma_resta                            --Bloque suma y resta

        Port(aa,bb: in std_logic_vector(7 downto 0); --Entradas del bloque
              operacio0,operacio1,operacio2: in std_logic; --Selecciona
                                                              --la operación
              cout: out std_logic;                       --Carry de salida
              sor_sum_res: out std_logic_vector(7 downto 0)); --Salida

    END COMPONENT;

    COMPONENT multiplexor                            --Bloque multiplexor

        Port(entr_a,entr_b: in std_logic_vector(7 downto 0); --Entradas del
                                                              --bloque
              operacio0,operacio1,operacio2: in std_logic;  --Selecciona la
                                                              --operación
              sortida: out std_logic_vector(7 downto 0));    --Salida

    END COMPONENT;

    SIGNAL x,y: std_logic_vector (7 DOWNT0 0);          --Variables intermedias
                                                         --para unir
                                                         --los 3 bloques anteriores

    BEGIN

        U1: and_or port map(a,b,op0,op1,y);             --Paso de parámetros
        U2: suma_resta port map(a,b,op0,op1,op2,cout,x); --Paso de parámetros
        U3: multiplexor port map(x,y,op0,op1,op2,resultat); --Paso de
                                                         --parámetros
```

```

END estructura;

Library ieee;
  Use ieee.std_logic_1164.ALL;

Library work;
  Use work.ALL;

ENTITY and_or IS  --Bloque que define las entradas salidas de la AND y la
                  OR
  Port( aa:in std_logic_vector(7 downto 0); --Vector de 8 bits de entrada
        bb:in std_logic_vector(7 downto 0); --Vector de 8 bits de entrada
        operacio0:in std_logic;             --Seleccionamos si queremos hacer
                                           la operación AND
        operacio1:in std_logic;             --la OR o ninguna de las dos.
        sor_and_or: out std_logic_vector(7 downto 0));--Salida del bloque

END and_or;

Library ieee;
  Use ieee.std_logic_1164.ALL;

Library work;
  Use work.ALL;

ENTITY suma_resta IS  --Bloque que define las entradas y salidas de la
                      suma y la resta
  Port( aa:in std_logic_vector(7 downto 0); --Vector de entrada de 8 bits
        bb:in std_logic_vector(7 downto 0); --Vector de entrada de 8 bits
        operacio0:in std_logic             --En estas tres variables seleccionamos
        operacio1:in std_logic;            --la operación que tenemos que realizar
        operacio2:in std_logic;
        cout:out std_logic;                --Carry de salida
        sor_sum_res: out std_logic_vector(7 downto 0));--Salida del bloque

END suma_resta;

Library ieee;
  Use ieee.std_logic_1164.ALL;

Library work;
  Use work.ALL;

ENTITY multiplexor IS  --Bloque multiplexor
  Port( entr_a:in std_logic_vector(7 downto 0); --Vector de entrada de 8
                                           bits
        entr_b:in std_logic_vector(7 downto 0); --Vector de entrada de 8
                                           bits
        operacio0:in std_logic;                --Seleccionamos la
                                           operación que queremos
                                           realizar

```

```

        operacio1:in std_logic;
        operacio2:in std_logic;
        sortida: out std_logic_vector(7 downto 0)); --Salida

END multiplexor;

ARCHITECTURE andor OF and_or IS --Definición de la función del bloque

    SIGNAL inter_op_0:std_logic; --Guarda el valor de la operacio0 negada
    SIGNAL inter_op_1:std_logic; --Guarda el valor de la operacio1 negada
    SIGNAL funcio:std_logic;      --Le decimos si tiene que hacer la AND o
                                --la OR

BEGIN

    inter_op_0<=not operacio0;
    inter_op_1<=not operacio1;
    funcio<=inter_op_0 and inter_op_1;

    reg1(0)<=aa(0) and bb(0);    --Realizamos la AND y la OR del bit 0
    reg2(0)<=aa(0) and funcio;
    reg3(0)<=bb(0) and funcio;
    sor_and_or(0)<=reg1(0) or reg2(0) or reg3(0);

    reg1(1)<=aa(1) and bb(1);    --Realizamos la AND y la OR del bit 1
    reg2(1)<=aa(1) and funcio;
    reg3(1)<=bb(1) and funcio;
    sor_and_or(1)<=reg1(1) or reg2(1) or reg3(1);

    reg1(2)<=aa(2) and bb(2);    --Realizamos la AND y la OR del bit 2
    reg2(2)<=aa(2) and funcio;
    reg3(2)<=bb(2) and funcio;
    sor_and_or(2)<=reg1(2) or reg2(2) or reg3(2);

    reg1(3)<=aa(3) and bb(3);    --Realizamos la AND y la OR del bit 3
    reg2(3)<=aa(3) and funcio;
    reg3(3)<=bb(3) and funcio;
    sor_and_or(3)<=reg1(3) or reg2(3) or reg3(3);

    reg1(4)<=aa(4) and bb(4);    --Realizamos la AND y la OR del bit 4
    reg2(4)<=aa(4) and funcio;
    reg3(4)<=bb(4) and funcio;
    sor_and_or(4)<=reg1(4) or reg2(4) or reg3(4);

    reg1(5)<=aa(5) and bb(5);    --Realizamos la AND y la OR del bit 5
    reg2(5)<=aa(5) and funcio;
    reg3(5)<=bb(5) and funcio;
    sor_and_or(5)<=reg1(5) or reg2(5) or reg3(5);

    reg1(6)<=aa(6) and bb(6);    --Realizamos la AND y la OR del bit 6
    reg2(6)<=aa(6) and funcio;
    reg3(6)<=bb(6) and funcio;
    sor_and_or(6)<=reg1(6) or reg2(6) or reg3(6);

    reg1(7)<=aa(7) and bb(7);    --Realizamos la AND y la OR del bit 7
    reg2(7)<=aa(7) and funcio;
    reg3(7)<=bb(7) and funcio;
    sor_and_or(7)<=reg1(7) or reg2(7) or reg3(7);

```

```
END andor;
```

```
ARCHITECTURE sum_rest OF suma_resta IS
```

```
SIGNAL opera0:std_logic; --Guardamos el valor de la operacio0 negada
SIGNAL opera1:std_logic; --Guardamos el valor de la operacio1 negada
SIGNAL opera2:std_logic; --Guardamos el valor de la operacio2 negada
SIGNAL invb:std_logic;  --Bit para saber si tenemos que hacer la
                        inversa de B
SIGNAL clra:std_logic;  --Bit para poner el vector A a 0
SIGNAL clranot:std_logic; --Guardamos el valor del clra negado
SIGNAL cin:std_logic_vector(7 downto 0); --Carry de entrada a cada
                        uno de los 7 sumadores

SIGNAL a_entr_sum:std_logic_vector(7 downto 0); --Entrada del vector
                        A al sumador
SIGNAL b_entr_sum:std_logic_vector(7 downto 0); --Entrada del vector
                        B al sumador
SIGNAL bit0:std_logic_vector(7 downto 0);      --Bits para realizar
                                                una NAND de tres
                                                entradas

SIGNAL bit1:std_logic_vector(7 downto 0);
SIGNAL bit2:std_logic_vector(7 downto 0);
```

```
BEGIN
```

```
opera0<=not operacio0;
opera1<=not operacio1;
opera2<=not operacio2;
clra<=opera0 and opera1;
clranot<=not clra;
invb<=opera0 and opera2;
cin(0)<=opera2 and operacio1;
```

```
--OPERACIONES ANTES DE ENTRAR LOS DATOS AL SUMADOR
```

```
a_entr_sum(0)<=clranot and aa(0);
b_entr_sum(0)<=invb xor bb(0);

a_entr_sum(1)<=clranot and aa(1);
b_entr_sum(1)<=invb xor bb(1);

a_entr_sum(2)<=clranot and aa(2);
b_entr_sum(2)<=invb xor bb(2);

a_entr_sum(3)<=clranot and aa(3);
b_entr_sum(3)<=invb xor bb(3);

a_entr_sum(4)<=clranot and aa(4);
b_entr_sum(4)<=invb xor bb(4);

a_entr_sum(5)<=clranot and aa(5);
b_entr_sum(5)<=invb xor bb(5);

a_entr_sum(6)<=clranot and aa(6);
b_entr_sum(6)<=invb xor bb(6);

a_entr_sum(7)<=clranot and aa(7);
b_entr_sum(7)<=invb xor bb(7);
```

## --OPERACIONES DENTRO DEL SUMADOR

```
sor_sum_res(0)<=cin(0) xor (a_entr_sum(0) xor b_entr_sum(0));
bit0(0)<=a_entr_sum(0) nand b_entr_sum(0);
bit1(0)<=b_entr_sum(0) nand cin(0);
bit2(0)<=a_entr_sum(0) nand cin(0);
cin(1)<=not(bit0(0) and bit1(0) and bit2(0));

sor_sum_res(1)<=cin(1) xor (a_entr_sum(1) xor b_entr_sum(1));
bit0(1)<=a_entr_sum(1) nand b_entr_sum(0);
bit1(1)<=b_entr_sum(1) nand cin(1);
bit2(1)<=a_entr_sum(1) nand cin(1);
cin(2)<= not(bit0(1) and bit1(1) and bit2(1));

sor_sum_res(2)<=cin(2) xor (a_entr_sum(2) xor b_entr_sum(2));
bit0(2)<=a_entr_sum(2) nand b_entr_sum(2);
bit1(2)<=b_entr_sum(2) nand cin(2);
bit2(2)<=a_entr_sum(2) nand cin(2);
cin(3)<= not(bit0(2) and bit1(2) and bit2(2));

sor_sum_res(3)<=cin(3) xor (a_entr_sum(3) xor b_entr_sum(3));
bit0(3)<=a_entr_sum(3) nand b_entr_sum(3);
bit1(3)<=b_entr_sum(3) nand cin(3);
bit2(3)<=a_entr_sum(3) nand cin(3);
cin(4)<= not(bit0(3) and bit1(3) and bit2(3));

sor_sum_res(4)<=cin(4) xor (a_entr_sum(4) xor b_entr_sum(4));
bit0(4)<=a_entr_sum(4) nand b_entr_sum(4);
bit1(4)<=b_entr_sum(4) nand cin(4);
bit2(4)<=a_entr_sum(4) nand cin(4);
cin(5)<= not(bit0(4) and bit1(4) and bit2(4));

sor_sum_res(5)<=cin(5) xor (a_entr_sum(5) xor b_entr_sum(5));
bit0(5)<=a_entr_sum(5) nand b_entr_sum(5);
bit1(5)<=b_entr_sum(5) nand cin(5);
bit2(5)<=a_entr_sum(5) nand cin(5);
cin(6)<= not(bit0(5) and bit1(5) and bit2(5));

sor_sum_res(6)<=cin(6) xor (a_entr_sum(6) xor b_entr_sum(6));
bit0(6)<=a_entr_sum(6) nand b_entr_sum(6);
bit1(6)<=b_entr_sum(6) nand cin(6);
bit2(6)<=a_entr_sum(6) nand cin(6);
cin(7)<= not(bit0(6) and bit1(6) and bit2(6));

sor_sum_res(7)<=cin(7) xor (a_entr_sum(7) xor b_entr_sum(7));
bit0(7)<=a_entr_sum(7) nand b_entr_sum(7);
bit1(7)<=b_entr_sum(7) nand cin(7);
bit2(7)<=a_entr_sum(7) nand cin(7);
cout<= not(bit0(7) and bit1(7) and bit2(7));
```

END sum\_rest;

ARCHITECTURE multi OF multiplexor IS

```

SIGNAL regi1:std_logic;  --Guardamos el valor de una AND de tres
                           entradas correspondiente a la operación a
                           realizar
SIGNAL regi2:std_logic;  --Guardamos el valor de una AND de tres
                           entradas correspondiente a la operación a
                           realizar
SIGNAL mux:std_logic;    --Bit que dice que entrada tiene que coger el
                           multiplexor
SIGNAL nmux:std_logic;   --Negación de la variable mux
SIGNAL sor0:std_logic_vector(7 downto 0); --Canal 0 del multiplexor
SIGNAL sor1:std_logic_vector(7 downto 0); --Canal 1 del multiplexor
SIGNAL opera0:std_logic; --Canal que tiene que coger el multiplexor
                           dependiendo de la operación que se tenga
                           que realizar

SIGNAL opera1:std_logic;
SIGNAL opera2:std_logic;

BEGIN

    opera0<=not operacio0;
    opera1<=not operacio1;
    opera2<=not operacio2;
    regi1<=operacio0 and operacio1 and opera2;
    regi2<=opera0 and opera1 and operacio2;
    mux<=regi1 or regi2;
    nmux<=not mux;

    sor0(0)<=nmux and entr_a(0);
    sor1(0)<=mux and entr_b(0);
    sortida(0)<=sor0(0) or sor1(0);  --Salida del multiplexor
                                       correspondiente al bit 0

    sor0(1)<=nmux and entr_a(1);
    sor1(1)<=mux and entr_b(1);
    sortida(1)<=sor0(1) or sor1(1);  --Salida del multiplexor
                                       correspondiente al bit 1

    sor0(2)<=nmux and entr_a(2);
    sor1(2)<=mux and entr_b(2);
    sortida(2)<=sor0(2) or sor1(2);  --Salida del multiplexor
                                       correspondiente al bit 2

    sor0(3)<=nmux and entr_a(3);
    sor1(3)<=mux and entr_b(3);
    sortida(3)<=sor0(3) or sor1(3);  --Salida del multiplexor
                                       correspondiente al bit 3

    sor0(4)<=nmux and entr_a(4);
    sor1(4)<=mux and entr_b(4);
    sortida(4)<=sor0(4) or sor1(4);  --Salida del multiplexor
                                       correspondiente al bit 4

    sor0(5)<=nmux and entr_a(5);
    sor1(5)<=mux and entr_b(5);
    sortida(5)<=sor0(5) or sor1(5);  --Salida del multiplexor
                                       correspondiente al bit 5

```

```

sor0(6)<=nmux and entr_a(6);
sor1(6)<=mux and entr_b(6);
sortida(6)<=sor0(6) or sor1(6);    --Salida del multiplexor
                                   correspondiente al bit 6

sor0(7)<=nmux and entr_a(7);
sor1(7)<=mux and entr_b(7);
sortida(7)<=sor0(7) or sor1(7);    --Salida del multiplexor
                                   correspondiente al bit 7

END multi;

```

## Anexo 2. Programa de simulación de la ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
library work;
use work.all;

ENTITY sim IS
END sim;                                --Entidad simulación

ARCHITECTURE proves OF sim IS

    SIGNAL a: std_logic_vector (7 downto 0);
    SIGNAL b: std_logic_vector (7 downto 0);
    SIGNAL resultat: std_logic_vector (7 downto 0);
    SIGNAL op0: std_logic;
    SIGNAL op1: std_logic;
    SIGNAL op2: std_logic;
    SIGNAL cout: std_logic;

    COMPONENT alu

        Port (a: in std_logic_vector(7 downto 0);
              b: in std_logic_vector(7 downto 0);
              resultat: out std_logic_vector(7 downto 0);  --Entradas y salidas de la ALU
              op0: in std_logic;
              op1: in std_logic;
              op2: in std_logic;
              cout: out std_logic);

    END COMPONENT;

    SIGNAL variable end_test: boolean:=false;
    constant T:time:=10ns;    --Freq=(1/T)=100Mhz

    BEGIN

        U4: alu port map(a,b,resultat,op0,op1,op2,cout);  --Paso de parámetros al Bloque ALU

        proceso: process BEGIN

            a<="00000000";
            b<="00000000";
            op0<='0';      --Sucesión de pruebas para mirar la salida final
            op1<='0';
            op2<='0';

            wait for T;

            a<="11111111";
            b<="00000000";
            op0<='1';
            op1<='0';
            op2<='0';

            wait for T;

            a<="11111111";
```



```

    b<="11111111";
    op0<='0';
    op1<='1';
    op2<='0';

wait for T;

    a<="11111111";
    b<="00011100";
    op0<='1';
    op1<='1';
    op2<='0';

wait for T;

    a<="11111111";
    b<="11111111";
    op0<='0';
    op1<='0';
    op2<='1';

wait for T;

    a<="00001101";
    b<="00110011";
    op0<='0';
    op1<='0';
    op2<='0';

wait for T;

    a<="01010101";
    b<="10101010";
    op0<='1';
    op1<='0';
    op2<='0';

wait for T;

    a<="01010101";
    b<="10101010";
    op0<='1';
    op1<='1';
    op2<='0';

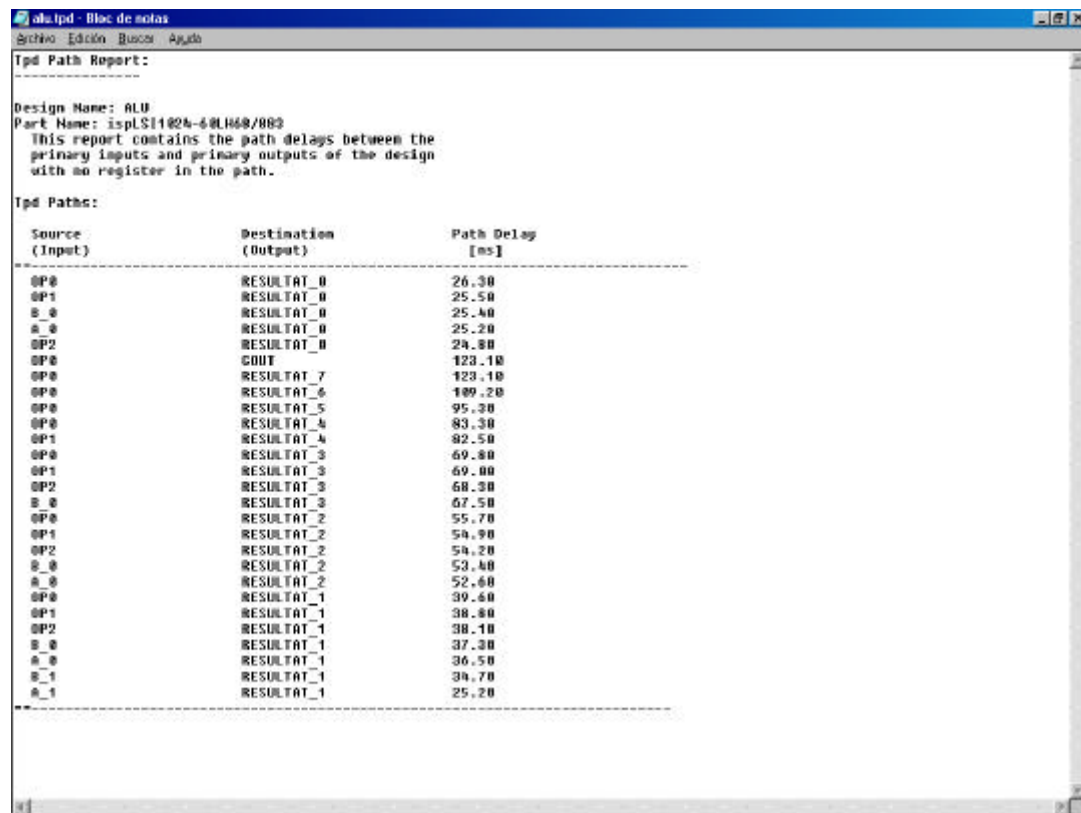
wait for T;
end_test:=true;
wait;

END process;

END proves;

```

### Anexo 3. Fichero de retardos después de la síntesis



alutpd - Bloc de notas

Archivo Edición Búsqueda Ayuda

Tpd Path Report:

Design Name: ALU  
Part Name: ispLSI1024-40LH60/800

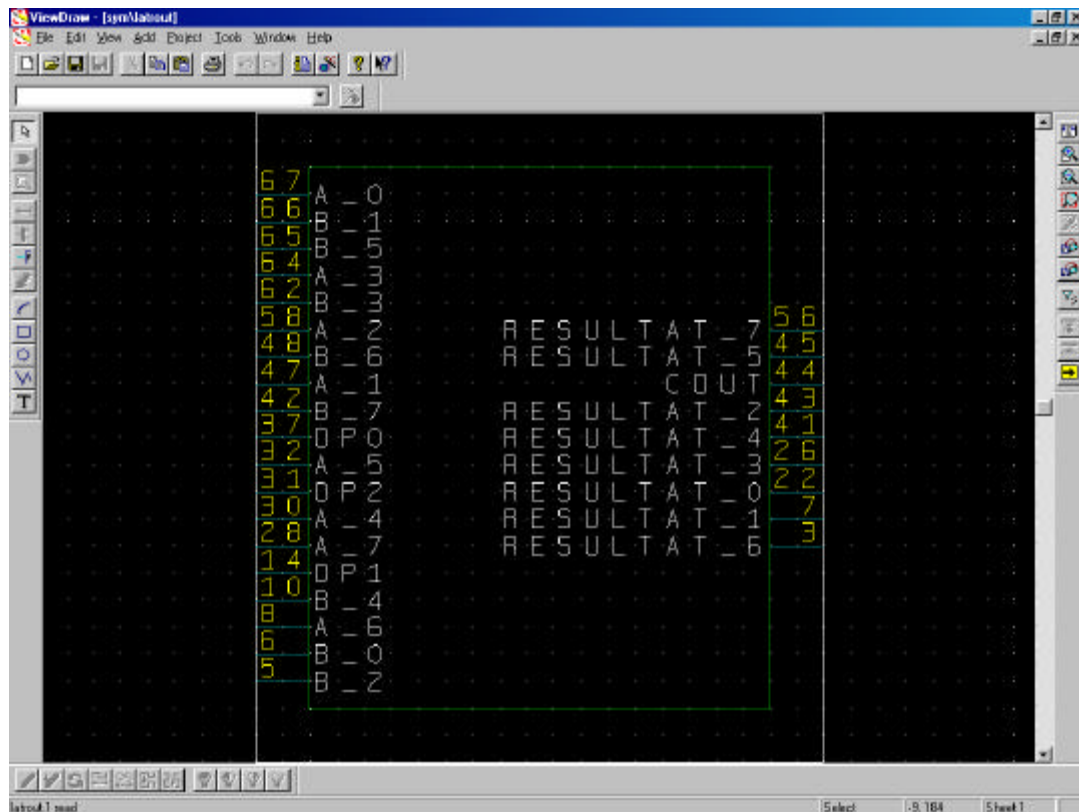
This report contains the path delays between the primary inputs and primary outputs of the design with no register in the path.

Tpd Paths:

Source (Input)	Destination (Output)	Path Delay [ns]
OP0	RESULTAT_0	26.30
OP1	RESULTAT_0	25.50
B_0	RESULTAT_0	25.40
A_0	RESULTAT_0	25.20
OP2	RESULTAT_0	24.80
OP0	COOUT	123.10
OP0	RESULTAT_7	123.10
OP0	RESULTAT_6	109.20
OP0	RESULTAT_5	95.30
OP0	RESULTAT_4	83.30
OP1	RESULTAT_4	82.50
OP0	RESULTAT_3	69.80
OP1	RESULTAT_3	69.00
OP2	RESULTAT_3	68.30
B_0	RESULTAT_3	67.50
OP0	RESULTAT_2	55.70
OP1	RESULTAT_2	54.90
OP2	RESULTAT_2	54.20
B_0	RESULTAT_2	53.40
A_0	RESULTAT_2	52.60
OP0	RESULTAT_1	39.60
OP1	RESULTAT_1	38.80
OP2	RESULTAT_1	38.10
B_0	RESULTAT_1	37.30
A_0	RESULTAT_1	36.50
B_1	RESULTAT_1	34.70
A_1	RESULTAT_1	25.20

En este fichero podemos observar los retardos que obtendré dependiendo de las entradas que yo le ponga

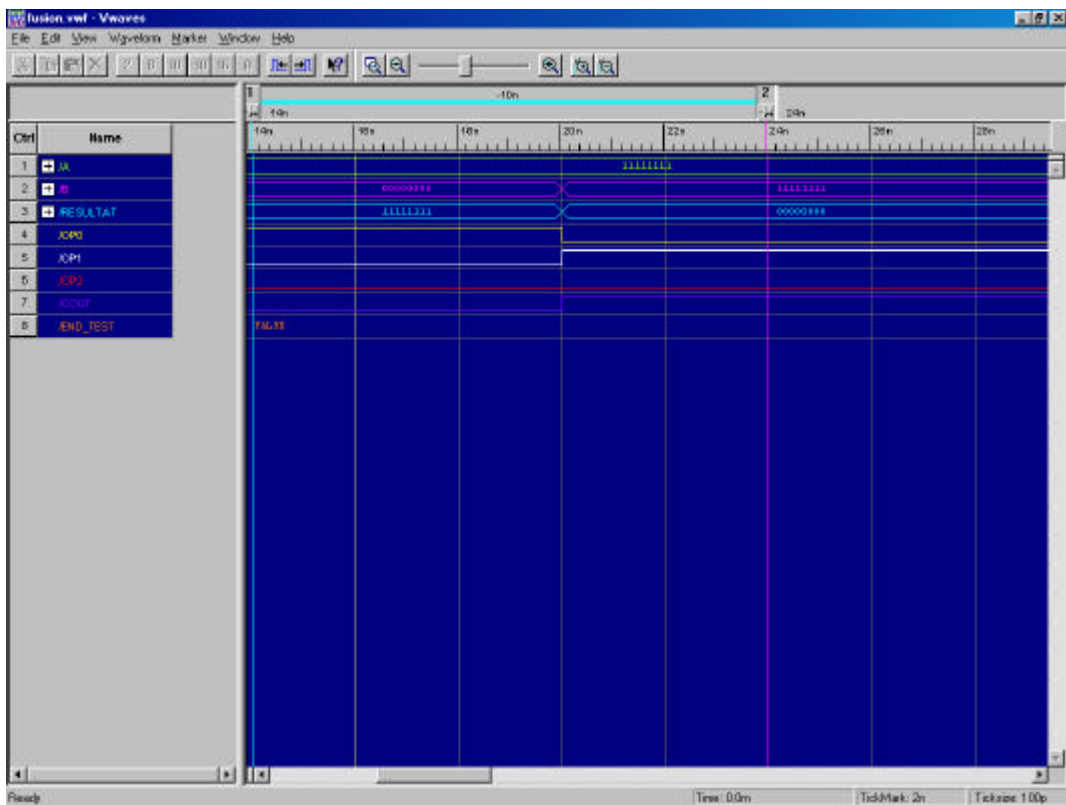
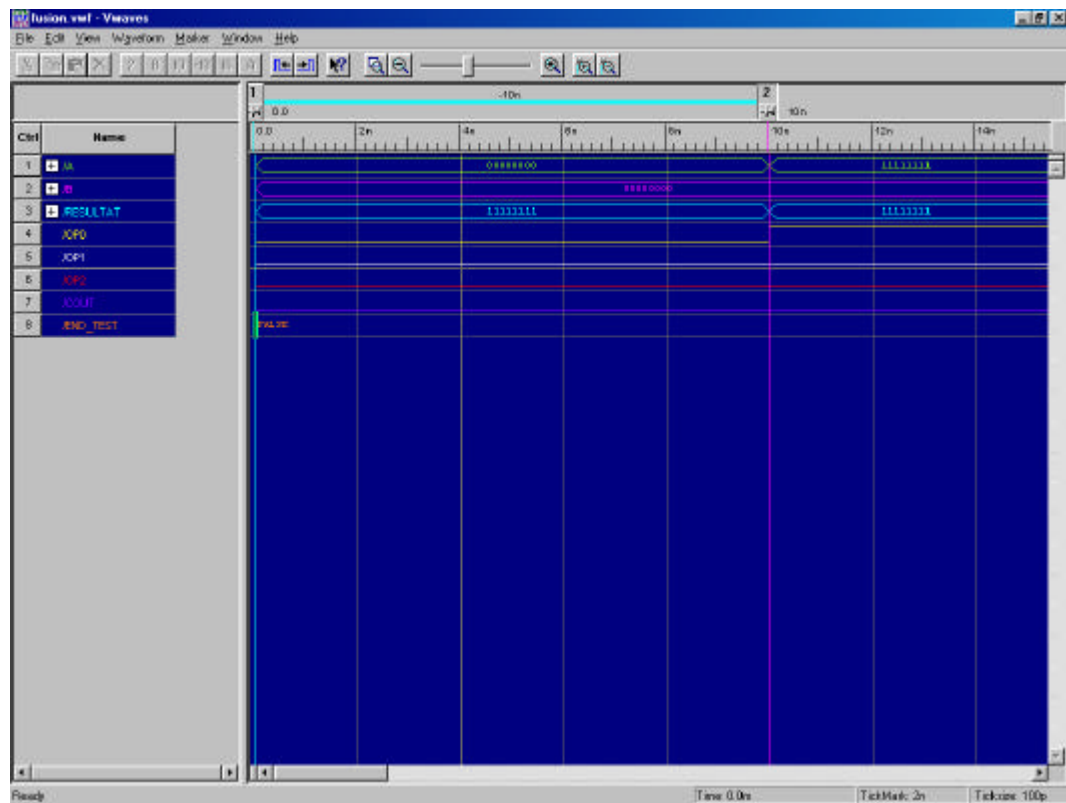
#### Anexo 4. Disposición de las patas de la FPGA

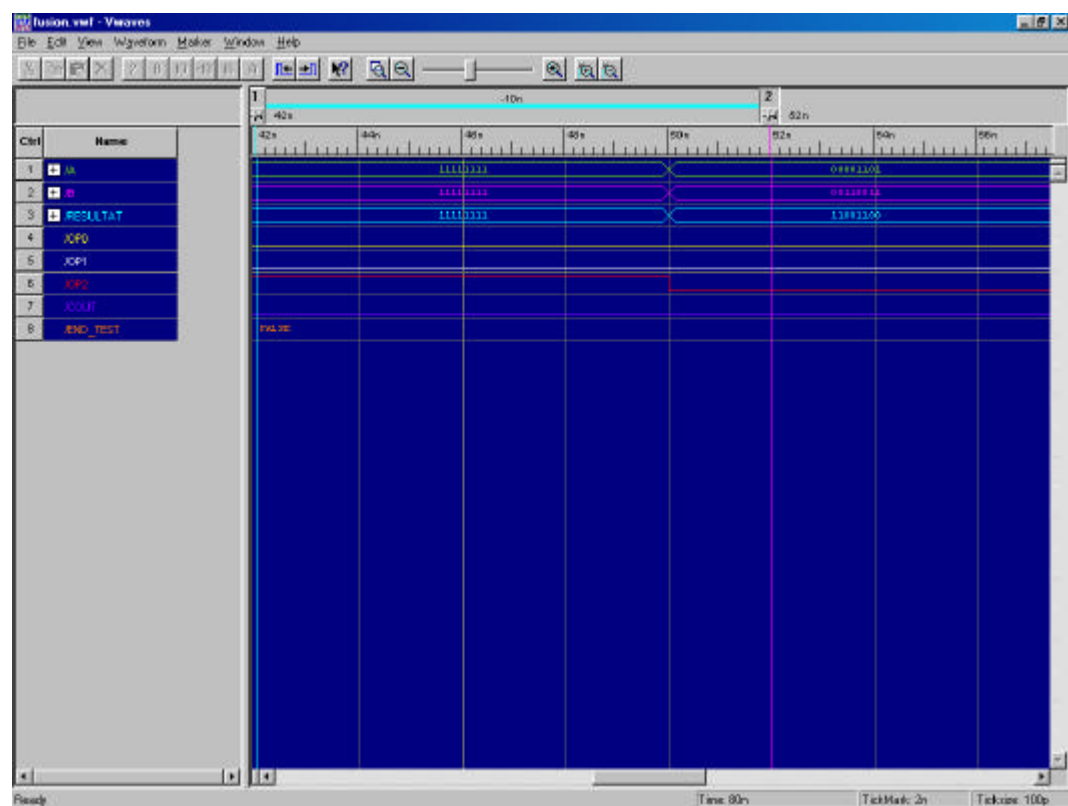
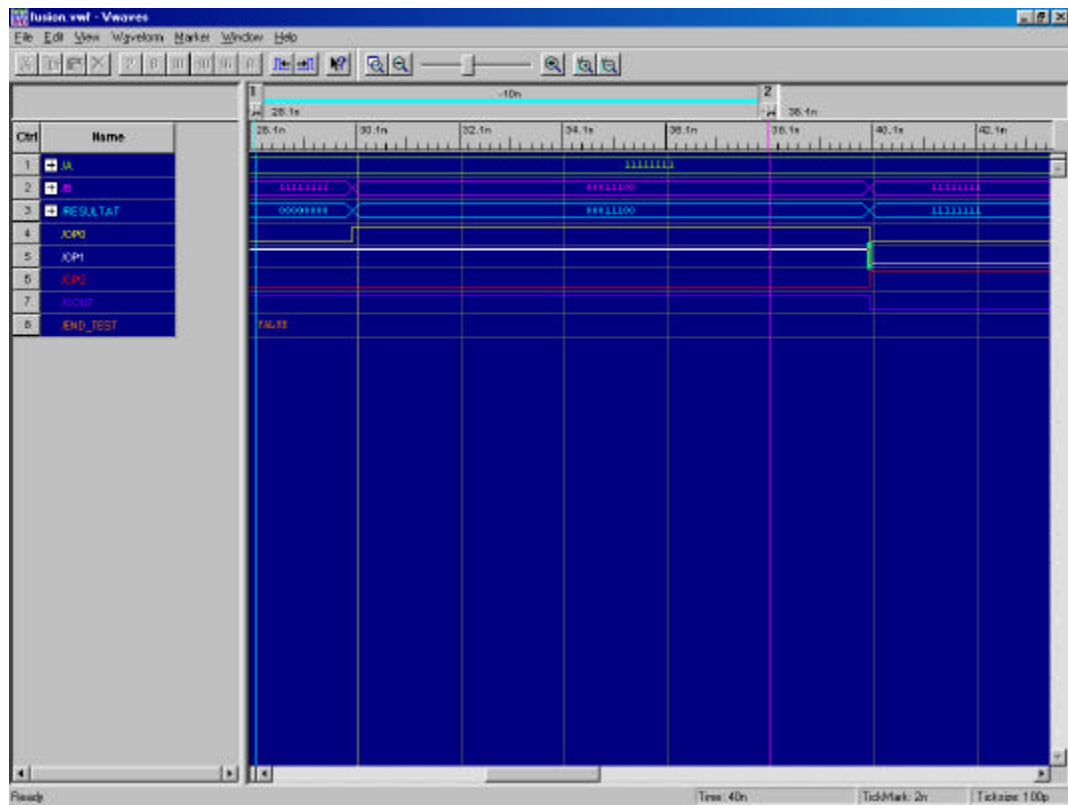


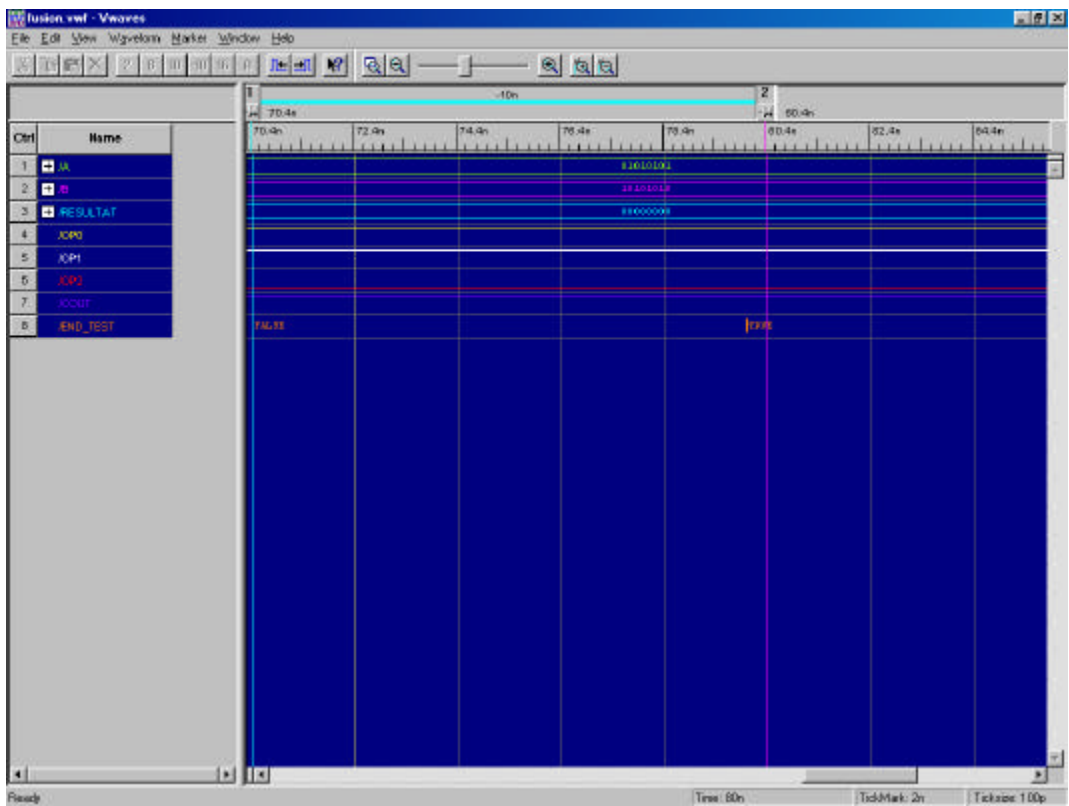
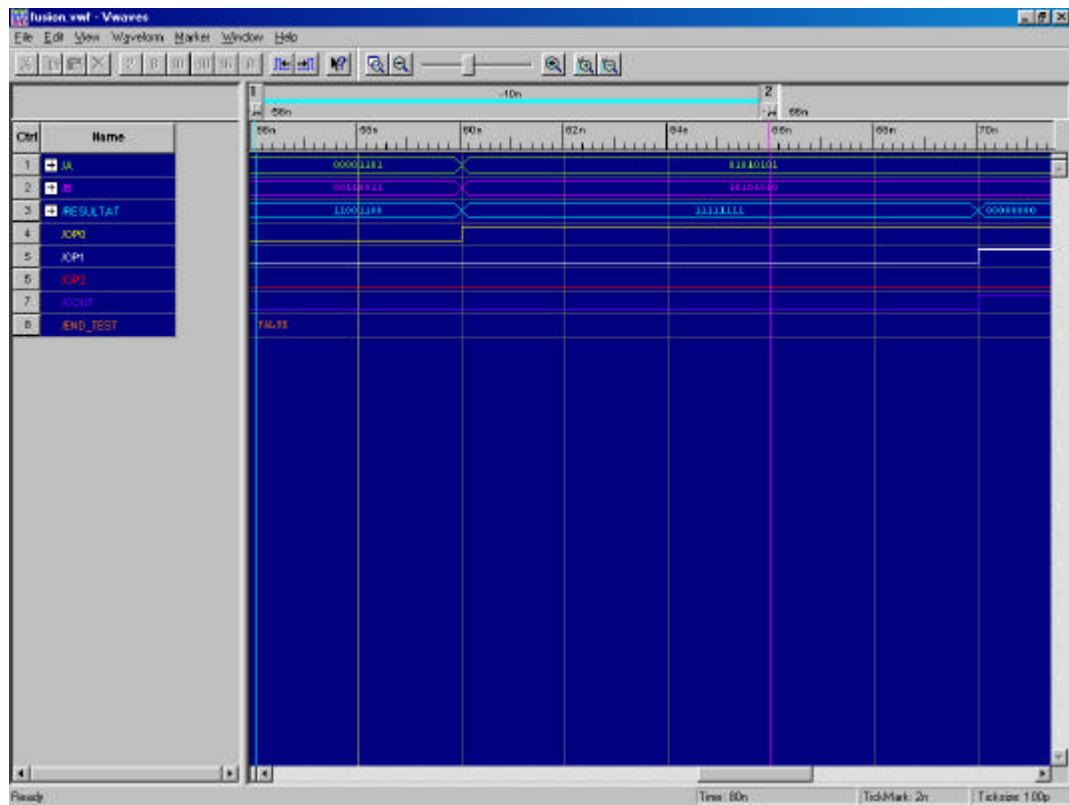
De esta forma es tal y como ha quedado nuestra ALU. La disposición de las patas es la que se indica en la figura.

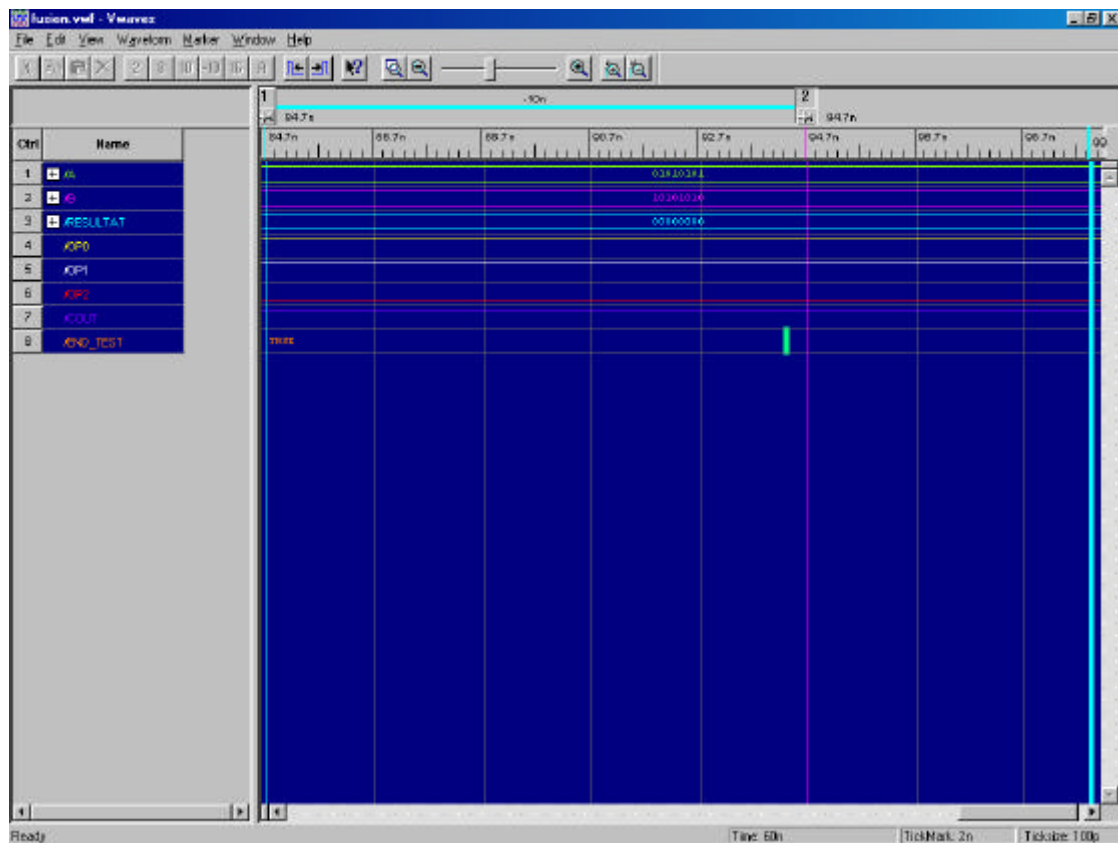
Con este programa también podemos poner con unas opciones si nos interesa por algún motivo la disposición de las patas según nos convenga, pero en nuestro caso como no lo necesitamos, lo hemos puesto tal y como nos lo ha dado el programa.

## Anexo 5. Gráficas de Salida









Como se puede observar en las gráficas, el resultado es correcto de acuerdo con las entrada a y b junto con las operaciones que queremos que haga dependiendo de lo que ponemos en las variables Op.

También se puede observar que cambiamos de valor cada 10 ns.

## **Bibliografía**

1. Lenguaje para la síntesis y modelado de circuitos VHDL. Autores : Fernando Pardo y Jose A. Boluda. Editorial RA-MA.
2. Página Web de la Universidad de Guadalajara
3. Página Web de la Universidad Politécnica de Valencia
4. Tutoriales y ayudas del programa eProduct Designer