



**Universidad de Valladolid**



**ESCUELA DE INGENIERÍAS  
INDUSTRIALES**

**UNIVERSIDAD DE VALLADOLID**

**ESCUELA DE INGENIERIAS INDUSTRIALES**

**Grado en Ingeniería Electrónica Industrial y Automática**

**Desarrollo de un simulador de  
circuitos digitales integrado en un  
compilador de diagramas ASM**

**Autor:**

**Sanz Cabrereros, David**

**Tutor:**

**De Pablo Gómez, Santiago  
Tecnología Electrónica**

**Valladolid, Mayo 2017.**





## Título:

Desarrollo de un simulador de circuitos digitales integrado en un compilador de diagramas ASM.

## Resumen:

El proyecto se centra en el diseño y desarrollo de un simulador de circuitos digitales para su integración en un compilador de diagramas ASM, capaz de simular un circuito digital descrito en lenguaje Verilog, mostrando al usuario tras su ejecución el resultado correcto o incorrecto de una serie de verificaciones descritas en el diagrama.

El objetivo principal es el desarrollo en lenguaje C++ de un simulador integrable en un compilador de diagramas ASM, capaz de a partir de un archivo con extensión .vdo, simular un circuito digital descrito en lenguaje Verilog y realizar una serie de verificaciones.

El objetivo secundario es desarrollar un interfaz de usuario empleando librerías Qt en el cual esté integrado el simulador y que permita al usuario seleccionar el archivo de simulación deseado, comprobar el proceso de simulación y obtener un archivo con extensión .txt con el resultado final de la simulación.

## Palabras clave:

- Simulador de circuitos digitales
- Diagramas ASM y ASM++
- Lenguaje VHDL y Verilog
- Circuitos a nivel RTL
- Dispositivos reconfigurables FPGA



## Title:

Development of a digital circuit simulator integrated in an ASM diagrams compiler.

## Abstract:

The project focuses on the design and development of a digital circuit simulator for its integration into an ASM diagrams compiler, capable of simulating a digital circuit described in verilog language, showing the user after its execution the correct or incorrect result of a list of verifications described in the diagram.

The main objective is the development in C ++ language of a simulator integrable in an ASM diagrams compiler, able to from a file with .vdo extension, simulate a digital circuit described in verilog language and perform a series of verifications.

The secondary objective is to develop a user interface using Qt libraries in which the simulator is integrated and which allows the user to select the desired simulation file, check the simulation process and obtain a file with .txt extension with the final result of the simulation.

## Keywords:

- Digital circuit simulator
- ASM and ASM++ diagrams
- VHDL and Verilog language
- RTL circuits
- Reconfigurable devices FPGA



## Índice:

<b>Capítulo 1: Introducción y objetivos</b> .....	<b>9</b>
1.1. Introducción .....	10
1.2. Objetivos .....	12
<b>Capítulo 2: Estado del arte</b> .....	<b>13</b>
2.1. Circuitos RTL.....	16
2.2. Máquinas de estado algorítmicas tradicionales (ASM charts).....	17
2.3. Máquinas de estado algorítmicas modernas: ASM++.....	18
2.4. Lenguajes HDL: VHDL y Verilog.....	21
2.5. FPGA: Dispositivos reconfigurables.....	22
2.6. Herramientas.....	23
2.6.1. Microsoft Office Visio .....	23
2.6.2. ASM++ Compiler .....	23
2.6.3. ModelSim.....	24
2.6.4. Lenguaje de programación C++.....	24
2.6.5. Microsoft Visual C++ 2010 express.....	25
2.6.6. Librerías Qt .....	26
<b>Capítulo 3: Desarrollo del TFG</b> .....	<b>27</b>
3.1. Teoría del simulador .....	28
3.1.1. Expresiones en lenguaje Verilog .....	34
3.1.2. Estructura de los diagramas ASM++ .....	42
3.1.3. Estructura de los archivos .vdo compilados para simulación.....	64
3.2. Especificaciones iniciales del proyecto .....	67
3.2.1. Expresiones en lenguaje Verilog .....	71
3.2.2. Tipos de cajas simulables de los diagramas ASM++ .....	71
3.2.3. Módulos de los archivos .vdo .....	73



3.3. Desarrollo de la programación del simulador .....	74
3.3.1. Desarrollo del interfaz de usuario.....	74
3.3.2. Desarrollo del analizador de expresiones .....	83
a) Lista enumerada de tokens .....	84
b) Función GetToken.....	86
c) Estructura del analizador de expresiones .....	90
d) Operaciones de las funciones del analizador de expresiones .....	98
3.3.3. Desarrollo del simulador .....	100
a) Función simulación.....	100
b) Lectura del archivo y almacenamiento de información.....	104
c) Preparación de la información .....	109
d) Simulación declarativa .....	113
e) Simulación operativa.....	118
f) Verificaciones.....	128
g) Errores y avisos.....	129
h) Archivo .txt de salida .....	130
<b>Capítulo 4: Conclusiones finales y líneas futuras .....</b>	<b>133</b>
<b>Capítulo 5: Bibliografía .....</b>	<b>137</b>
5.1. Artículos y seminarios.....	138
5.2. Páginas web .....	139
5.3. Proyectos finales de carrera .....	139



**Capítulo 6: Anexos .....141**

6.1. Archivos para la ejecución del proyecto.....142

6.1.1. aplicacion.pro .....142

6.1.2. doQmake.bat.....142

6.1.3. Archivos .qrc de recursos .....142

6.1.4. Archivo Test\_Multiplier.vdx y resultado de su simulación.....142

6.1.5. Bibliotecas .dll .....142

6.1.6. Ejecutable del simulador .....142

6.2. Archivos con extensión .cpp y bibliotecas del simulador .....143

6.2.1. main.cpp .....143

6.2.2. mainwindow.cpp .....143

    a) Funciones del interfaz de usuario.....143

    b) Funciones del simulador .....145

    c) Inicialización de las matrices empleadas para almacenar datos...148

    d) Funciones del analizador de expresiones .....149

6.2.3. simulador.cpp.....149

    a) Funciones para resolver operaciones del simulador.....149

    b) Función para detectar partes de expresiones .....150

6.2.4. shared.h.....150

6.2.5. GlobalDefs.h .....150

6.2.6. mainwindow.h .....150

6.2.7. simulador.h.....151





# Capítulo 1:

## Introducción y objetivos

---

# Capítulo 1: Introducción y objetivos

## 1.1. Introducción:

Un sistema electrónico es un conjunto de circuitos digitales que interactúan entre sí para obtener un resultado. A grandes rasgos, todos los circuitos digitales están formados por una estructura similar que consta de unas señales de entrada (Inputs), unos procesadores de la información y unas salidas (Outputs).

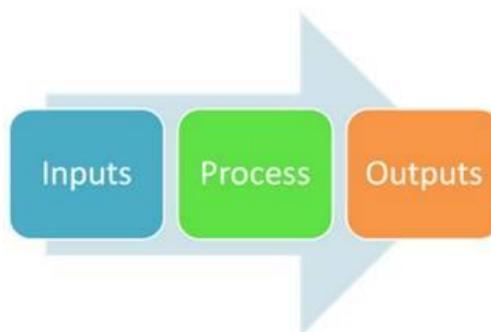


Figura 1: Esquema básico de un circuito digital

Para poder procesar esa información obtenida en la señales de entrada es necesario realizar un diseño adecuado de los circuitos digitales para que al implantarlo en el sistema físico estos realicen correctamente las tareas deseadas.

La realización de este diseño se lleva a cabo a través de una serie de lenguajes de descripción de hardware (HDL) empleados en el modelado de los sistemas electrónicos, entre los cuales los más comunes internacionalmente son los lenguajes VHDL y Verilog.

Estos lenguajes se encargan de definir la estructura, diseño y operación de los circuitos digitales haciendo posible así obtener una descripción formal de los mismos, posibilitando obtener un análisis automático y una simulación previa de los circuitos sin necesidad de esperar a su implantación física.



Pese a que el aprendizaje de estos lenguajes no es excesivamente complicado, dada la complejidad actual de algunos circuitos digitales, realizar el diseño de estos directamente en lenguaje VHDL o Verilog puede volverse en ocasiones tedioso y visualmente poco descriptivo. Para simplificar el diseño de estos circuitos y permitir a personas que no estén suficientemente formadas con estos lenguajes entender la lógica de estos circuitos existe un lenguaje de programación que simplifica la descripción de los circuitos.

A este método gráfico de diseño de circuitos digitales se le denomina ASM++, basado en las máquinas de estado algorítmicas tradicionales (ASM), que incrementa sus posibilidades (para más información ver Capítulo 2, apartado 2.3).

Una vez se tiene definido el diagrama ASM++, a través de un compilador de diagramas ASM++ como el desarrollado por la Universidad de Valladolid, se puede obtener la descripción del circuito en los diferentes lenguajes HDL y a partir de ahí realizar las verificaciones y simulaciones adecuadas para comprobar el funcionamiento real del circuito digital.

Y es en ese último punto donde se enmarca este proyecto: desarrollar un simulador digital para integrarlo en el compilador de diagramas ASM++ que sea capaz de proporcionar una simulación informática de un circuito digital prediciendo su comportamiento en el tiempo y verificando el resultado obtenido en diferentes momentos de la simulación para saber si el diseño es acorde a los resultados esperados.

## 1.2. Objetivos:

A la hora de realizar el diseño de un circuito digital para implementarlo físicamente en una aplicación es fundamental poder saber con antelación a su ejecución física cual va a ser su comportamiento real sin necesidad de probarlo físicamente en la aplicación, pudiendo así evitar posibles fallos en el diseño del circuito, corregirlos rápidamente y poder ahorrar tiempos en el diseño de una aplicación.

El objetivo de este proyecto es desarrollar ese simulador, integrable en un compilador de diagramas ASM++, capaz de obtener a partir de un archivo en lenguaje Verilog la simulación del circuito digital descrito, mostrando si el resultado esperado en determinados momentos del tiempo (mediante verificaciones añadidas en el diseño del diagrama) es el correcto o no.

Para su realización se han marcado una serie de objetivos mostrados a continuación:

- El objetivo principal es la realización de un simulador en lenguaje de programación C++, que a partir de un archivo con extensión .vdo (archivos que describen un conjunto de cajas interconectadas con expresiones descritas en Verilog, o alternativamente en VHDL) sea capaz de interpretar sus diferentes partes, simular el circuito y mostrar como salida si las verificaciones deseadas son correctas o no. Este simulador debe ser integrable en el compilador de diagramas ASM++, permitiendo al diseñador del compilador si lo desea añadir la función del simulador a continuación de la compilación del diagrama ASM++ y saber así si el circuito diseñado cumple con las verificaciones esperadas o no.
- Como objetivo secundario se realizará una aplicación con interfaz gráfico de usuario, empleando las librerías Qt, que permitirá disponer al diseñador de un circuito digital del simulador como una aplicación independiente del simulador de diagramas ASM++.

Esta aplicación, descrita más detalladamente en el Capítulo 2, permitirá seleccionar un archivo con extensión .vdo (lenguaje Verilog), editarlo y simularlo. La aplicación mostrará al usuario los diferentes pasos realizados para obtener la simulación final y creará un fichero con extensión .txt de salida con el valor de las diferentes señales del circuito en los ciclos de simulación, el resultado de las verificaciones indicadas en el diseño y una lista de errores y avisos para ayudar al diseñador a encontrar los posibles fallos en el diseño del circuito.



# Capítulo 2:

## Estado del arte

---

## Capítulo 2: Estado del arte

La electrónica, por definición, es el campo de la ingeniería y de la física aplicada relativo al diseño y aplicación de dispositivos cuyo funcionamiento depende del flujo de electrones para la generación, transmisión, recepción, almacenamiento de información, entre otros.

Dentro de los diferentes elementos que forman todo lo relativo a la electrónica, en la actualidad uno de sus elementos más importantes y fundamentales para el desarrollo de dispositivos electrónicos son los circuitos integrados digitales. Pese a que existen diferentes tipos y modelos, físicamente son similares al que se muestra en la siguiente figura:



Figura 2: Imagen de un circuito integrado digital real

Estos circuitos, formados por diferentes componentes, se encargan de realizar operaciones lógicas, marcadas por los operadores lógicos que lo forman. Estos circuitos se denominan “Circuitos Lógicos” o “Circuitos digitales”.

Los circuitos digitales emplean componentes encapsulados, los cuales pueden contener puertas lógicas o circuitos lógicos más complejos. En la siguiente tabla se muestran algunas de las funciones lógicas más básicas:

Tabla 1: Funciones lógicas básicas

NOMBRE	AND - Y	OR - O	XOR O-exclusiva	NOT Inversor	NAND	NOR																																																																																	
SÍMBOLO																																																																																							
SÍMBOLO																																																																																							
TABLA DE VERDAD	<table border="1"> <tr><th>a</th><th>b</th><th>z</th></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	a	b	z	0	0	0	0	1	0	1	0	0	1	1	1	<table border="1"> <tr><th>a</th><th>b</th><th>z</th></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	a	b	z	0	0	0	0	1	1	1	0	1	1	1	1	<table border="1"> <tr><th>a</th><th>b</th><th>z</th></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	a	b	z	0	0	0	0	1	1	1	0	1	1	1	0	<table border="1"> <tr><th>a</th><th>z</th></tr> <tr><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td></tr> </table>	a	z	0	1	1	0	<table border="1"> <tr><th>a</th><th>b</th><th>z</th></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	a	b	z	0	0	1	0	1	1	1	0	1	1	1	0	<table border="1"> <tr><th>a</th><th>b</th><th>z</th></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	a	b	z	0	0	1	0	1	0	1	0	0	1	1	0
a	b	z																																																																																					
0	0	0																																																																																					
0	1	0																																																																																					
1	0	0																																																																																					
1	1	1																																																																																					
a	b	z																																																																																					
0	0	0																																																																																					
0	1	1																																																																																					
1	0	1																																																																																					
1	1	1																																																																																					
a	b	z																																																																																					
0	0	0																																																																																					
0	1	1																																																																																					
1	0	1																																																																																					
1	1	0																																																																																					
a	z																																																																																						
0	1																																																																																						
1	0																																																																																						
a	b	z																																																																																					
0	0	1																																																																																					
0	1	1																																																																																					
1	0	1																																																																																					
1	1	0																																																																																					
a	b	z																																																																																					
0	0	1																																																																																					
0	1	0																																																																																					
1	0	0																																																																																					
1	1	0																																																																																					



Para poder realizar estas funciones lógicas (y otras funciones más complejas), los circuitos digitales emplean componentes encapsulados, los cuales se programan para que realicen las operaciones deseadas por el diseñador.

Para realizar este diseño, se emplean las máquinas de estados finitos, las cuales permiten modelar los circuitos digitales mediante lenguajes o modelos como el modelo tradicional de máquinas de estados algorítmicas (ASM).

Para ampliar las funcionalidades básicas de estos modelos, como ya se ha mencionado anteriormente existe un modelo extendido denominado ASM++, el cual permite modelar circuitos digitales algo más complejos, en un lenguaje fácilmente comprensible para su posterior instalación en los circuitos integrados.

El proceso de diseño de la lógica de los circuitos digitales puede ser una tarea muy simple o tremendamente complicada y laboriosa, dependiendo de la complejidad de las tareas deseadas para el propio circuito. Esto provoca que puedan aparecer multitud de fallos en la lógica del programa durante su diseño hasta obtener el modelo deseado. Para poder detectar estos fallos en el diseño de los circuitos, sin necesidad de esperar a tenerlo instalado y probado físicamente (lo cual lleva una cantidad importante de tiempo), se emplean aplicaciones informáticas que sirven de apoyo a los diseñadores para poder simular digitalmente los circuitos diseñados rápidamente, sin necesidad de disponer de un entorno físico.

Son diversos los programas, lenguajes y métodos que permiten el diseño, simulación e implantación de circuitos lógicos en circuitos integrados, pero entre toda esa multitud de opciones disponibles, de ahora en adelante se particularizará para la situación que ha llevado a la necesidad de realizar este proyecto.

Para la realización de este proyecto se han empleado una serie de conceptos, bibliotecas, lenguajes de programación y programas ya existentes, los cuales han servido de apoyo a la creación del simulador. En este apartado se va a realizar una explicación de cada herramienta empleada siguiendo el orden seguido desde la creación y diseño del diagrama para obtener el circuito digital, hasta lo empleado para realizar la simulación del mismo.

## 2.1. Circuitos RTL:

Se denomina circuito RTL (“Register Transfer Level”) a los circuitos digitales a nivel de transferencia de registros, empleados para el diseño de circuitos digitales complejos, en los cuales el diseñador ha de especificar todas las operaciones y transferencia entre registros que se suceden a lo largo de varios ciclos de reloj.

Para el diseño de estos circuitos digitales de cierta complejidad, se emplean máquinas de estado (FSM – Finite State Machine), representados con círculos y flechas. Las operaciones que ha de realizar el circuito se van anotando como comentarios cerca de los estados.

Gracias a estos modelos, se pueden diseñar gran cantidad de circuitos digitales, los cuales pueden ser sintetizados para introducirse en dispositivos reconfigurables tipo FPGA y así disponer físicamente del circuito digital diseñado.

A continuación se muestra el diseño de una máquina de estados diseñado a nivel RTL:

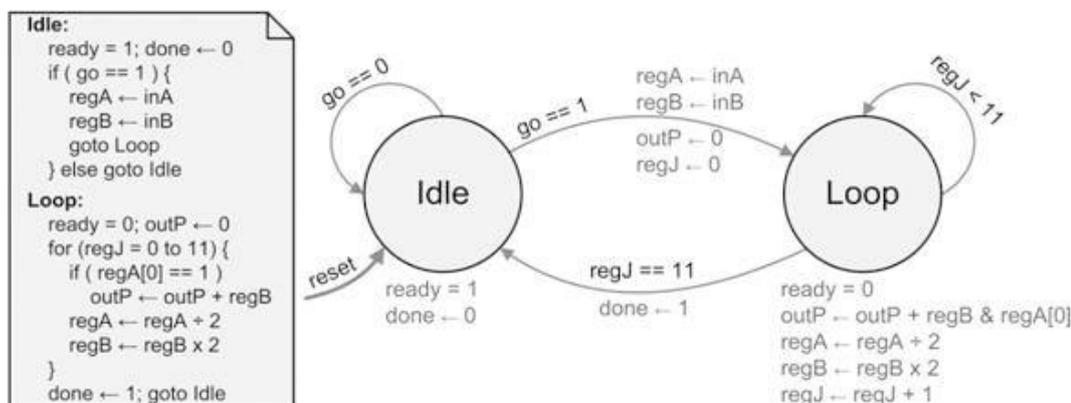


Figura 3: Máquina de estados diseñado a nivel RTL

Las máquinas de estados muestran de forma intuitiva cual será la evolución temporal del circuito, sin embargo, no son suficientemente claros como para expresar de forma clara y concisa las operaciones que realiza. Para solucionar esto último, existen diagramas de grafos como los diagramas ASM, que tienen como finalidad mantener una representación gráfica intuitiva del circuito, expresando con detalle y de forma consistente todas las operaciones que ha de realizar el circuito en cada uno de los ciclos de reloj.

## 2.2. Máquinas de estado algorítmicas tradicionales (ASM charts):

Para modelar los circuitos integrados digitales se emplean las Máquinas de estados finitos o autómatas finitos. Estas máquinas de estado finito son modelos computacionales capaces de procesar automáticamente información recibida por una entrada para producir una salida.

El diseño de estas máquinas de estados finitos se realiza mediante métodos como la Máquina de estados algorítmica (en adelante denominada con la abreviatura ASM). Este método se utiliza para representar los diagramas de los circuitos de una forma menos formal y más fácil de entender, basándose en la apariencia de los diagramas de estados. En resumen, un gráfico de ASM tradicional es un método para describir las operaciones por orden de un sistema digital.

Estos diagramas ASM se componen de la interconexión de tres elementos básicos:

- Los estados: diseñados con forma rectangular corresponden a un estado de un diagrama de estado regular o máquina de estados finitos.
- Los controles condición: con forma de rombo, son cajas encargadas de decidir cuál será el camino del diagrama dependiendo la condición indicada en ellos.
- Las salidas condicionales: son cajas con forma ovalada que indica las señales de salida.

Un ejemplo de un ASM tradicional es el mostrado a continuación:

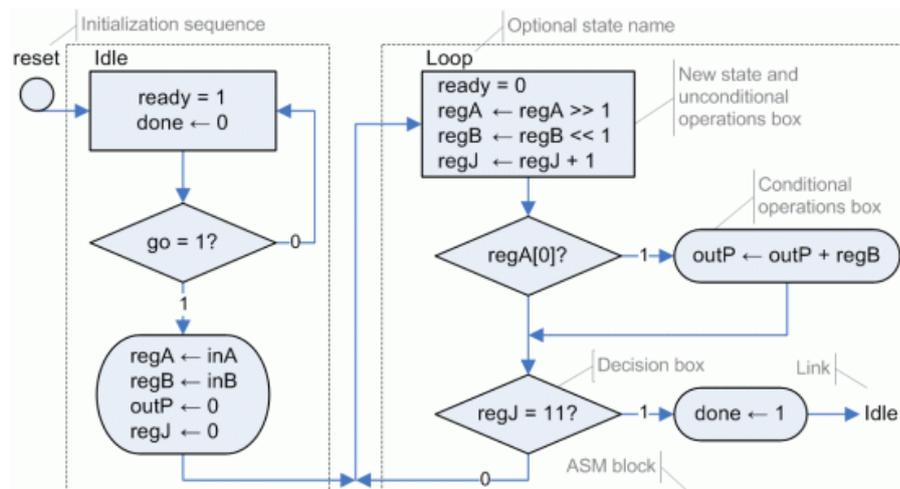


Figura 4: Circuito digital descrito en diagrama ASM

### 2.3. Máquinas algorítmicas de estado modernas: ASM++:

Con el fin de mejorar y ampliar las posibilidades de los ASM tradicionales, se ha creado una versión moderna de estos, denominada ASM++.

ASM++ (Extended Algorithmic State Machine) es un lenguaje gráfico creado con el objetivo de ser usado como un interfaz gráfico para generar circuitos digitales de forma automática.

Para tener una primera idea de la diferencia de estos diagramas con los ASM tradicionales, a continuación se muestra el mismo circuito digital mostrado anteriormente pero descrito en lenguaje ASM++:

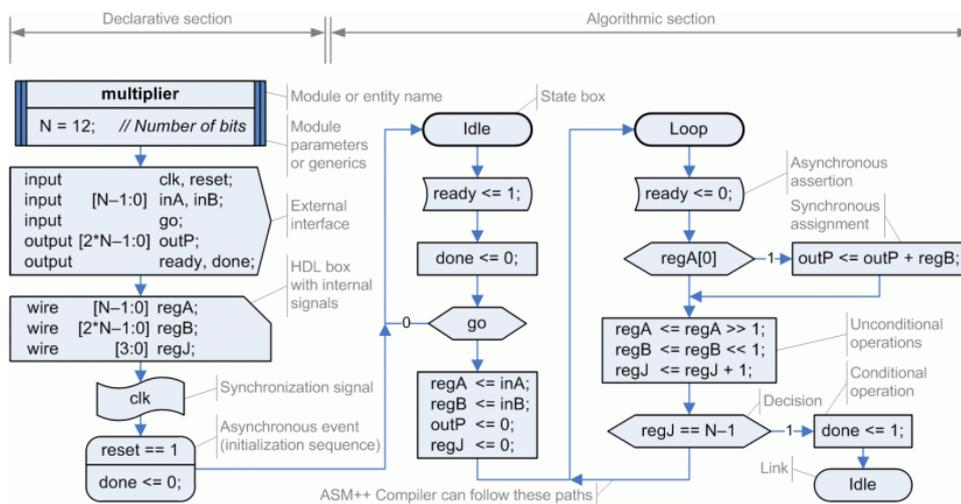


Figura 5: Circuito digital descrito en diagrama ASM++

La principal diferencia con los ASM tradicionales es que los ASM++ tienen unas nuevas cajas ovaladas de estado utilizadas para representar el comienzo y el final de cada ciclo de reloj. Las cajas rectangulares se emplean para describir operaciones síncronas (condicionales o incondicionales). Para describir las operaciones asíncronas se emplean unas cajas rectangulares con el lado izquierdo redondeado.

A parte de estos cambios se añaden una serie de cajas nuevas que se encargan de diferentes cosas, como definir entradas y salidas, definir el reloj con el que está sincronizado un circuito, definir el nombre del circuito y sus parámetros...

Todos estos elementos (denominados boxes o cajas de aquí en adelante) se encuentran descritos en lenguaje Verilog o VHDL y están definidos adecuadamente en la página web de ASM++.

A continuación se muestra una tabla resumen con todos los elementos existentes empleados en los diagramas ASM++:

Tabla 2: Elementos básicos de los diagramas ASM++

	<p>Las cajas “State” (Estado) indican el comienzo y el fin de cada ciclo de reloj. En su interior está indicado el nombre del estado. En ocasiones pueden usarse para unir partes distantes del diagrama.</p>
	<p>En estas cajas se escriben las diferentes operaciones que se realizarán de forma <b>síncrona</b> cuando termine el ciclo de reloj actual.</p>
	<p>En estas cajas se escriben las diferentes operaciones que realizará el circuito de forma asíncrona durante el ciclo del reloj actual.</p>
	<p>Las cajas decisión establecen 2 caminos alternativos basándose en una condición descrita en su interior.</p>
	<p>Estas cajas especifican el nombre del módulo que le precede, con el resto de cajas, y opcionalmente algunos parámetros que podrían ser cambiados a continuación.</p>
	<p>Esta caja es usada para definir el nombre y el tipo de las diferentes entradas y salidas del diseño.</p>
	<p>Estas cajas código “Code” son empleados para escribir directrices de compilación precedidas de #, o código en lenguaje VHDL o Verilog.</p>
	<p>Esta caja establece el nombre de la señal de sincronización usado por un circuito. Afecta a todos los cajas que le preceden.</p>
	<p>Esta caja específica que realiza el circuito cuando hay un evento asíncrono así como las secuencias de inicialización y primer estado de las máquinas de estado.</p>

Los símbolos descritos con anterioridad son los más comunes en los diagramas ASM++ simples. También existen otras cajas adicionales para diagramas más avanzados:

Tabla 3: Elementos avanzados de los diagramas ASM++

	<p>Una caja “conector” une dos partes de un diagrama para facilitar su diseño. Dependiendo si 2 conectores se encuentran en la misma página o no se muestran respectivamente con forma circular o con forma de flecha</p>
	<p>Estas cajas se usan para especificar los valores por defecto de las señales síncronas y asíncronas. Se usan antes de cualquier box STATE.</p>
	<p>Estas cajas se emplean para instanciar un módulo de nivel menor, definido con otro diagrama ASM++ o con código HDL.</p>
	<p>Estas cajas permiten tomar múltiples decisiones simultáneas.</p>
	<p>Este símbolo establece hilos paralelos, que se ejecutaran concurrentemente. Cada flecha de salida permite la descripción de múltiples estados o circuitos, con igual o diferente sincronización</p>
	<p>Estas cajas son una unión de las cajas decisión y asignación síncrona. La asignación descrita se ejecutara solamente si la condición es verdadera. En caso contrario no se ejecutara.</p>
	<p>Estas cajas son una unión de las cajas decisión y asignación asíncrona. La asignación descrita se ejecutara solamente si la condición es verdadera. En caso contrario no se ejecutara.</p>
	<p>Este elemento solo es empleado para facilitar la comprensión del diagrama. No tiene ningún efecto en la generación del circuito.</p>

## 2.4. Lenguajes HDL: VHDL y Verilog:

A la hora de modelar los diferentes sistemas electrónicos, existen una serie de lenguajes encargados de describir su hardware (HDL, Hardware Description Language). Entre los diferentes tipos existentes, los más reconocidos a nivel internacional son los lenguajes VHDL y Verilog.

Pese a que las diferentes herramientas empleadas a lo largo del desarrollo del software del simulador son capaces de emplear ambos lenguajes, el simulador se ha diseñado única y exclusivamente para interpretar circuitos descritos en lenguaje Verilog, debido en cierta manera y como se explicara a continuación a que el lenguaje Verilog se diseñó para ser similar al lenguaje C.

El lenguaje Verilog permite el diseño, la prueba y la implantación de circuitos analógicos, digitales y de señal mixta a diferentes niveles de abstracción.

Los diseñadores de este lenguaje querían un lenguaje con una sintaxis similar a la del lenguaje C empleado en programación. Este lenguaje fue creado para la implementación de Sistemas Operativos. Emplea datos estáticos, esta débilmente tipificado y es de medio nivel, ya que aunque dispone de estructuras empleadas por los lenguajes de alto nivel, dispone de construcciones del lenguaje que permiten controlar a muy bajo nivel.

Verilog emplea palabras similares a las reservadas por el Lenguaje C, variando algunos aspectos como las llaves de inicio y fin de C por las palabras `Begin` y `End`, y añadiendo algunos recursos necesarios como el concepto del tiempo.

A diferencia del resto de lenguajes de programación, Verilog no emplea ejecuciones lineales de sentencias, sino que está estructurado por módulos definidos con puertos de entrada, salida y bidireccionales, los cuales internamente están formados por cables y registros.

Para controlar el orden de ejecución de las sentencias, existen 2 formas de ejecutar los programas: sentencias secuenciales y concurrentes.

## 2.5. FPGA: Dispositivos reconfigurables

Una FPGA (Field Programmable Gate Array) es un dispositivo programable que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada “in situ” mediante un lenguaje de descripción especializado. Su arquitectura interna es como la mostrada en la Figura 6:

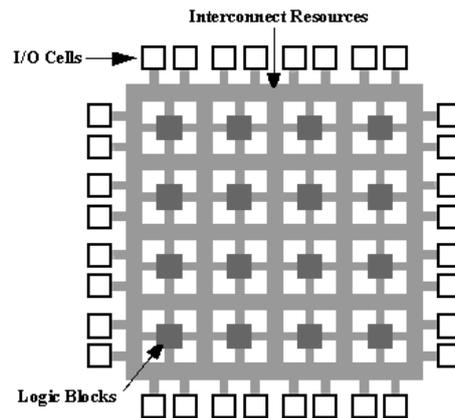


Figura 6: Arquitectura de una FPGA

Este tipo de dispositivos surgieron como una evolución de los CPLDs (Complex Programmable Logic Device).

Las FPGAs, gracias a su jerarquía de interconexiones programables, permiten a los bloques lógicos ser interconectados según desee el diseñador del sistema. Estos bloques e interconexiones se pueden programar después de su manufactura por el diseñador, pudiendo así desempeñar las FPGA cualquier función lógica deseada.

A diferencia de los microcontroladores y otros dispositivos, las FPGA tienen una forma de programación específica, que se basa en permitir al diseñador describir el hardware que tendrá la FPGA. Para realizar esto, el diseñador cuenta con entornos de desarrollo específicos basados en esquemas (ASM) o con lenguajes de programación (HDL) definidos previamente (en el apartado 2.1. Estado del arte).



## 2.6. Herramientas empleadas:

### 2.6.1. Microsoft Office Visio:

La empresa Microsoft dentro de todas las herramientas y aplicaciones que ofrece tiene un paquete de aplicaciones denominadas Microsoft Office. Por definición, Microsoft Office es una suite ofimática (un paquete de oficina) que abarca el mercado completo en internet e interrelaciona aplicaciones de escritorio, servidores y servicios para diferentes sistemas operativos. Esta suite está compuesta por diferentes aplicaciones informáticas empleadas en oficinas para realizar operaciones sobre archivos y documentos.

Dentro de todas las aplicaciones que ofrece Microsoft Office, para poder diseñar gráficamente los diagramas ASM++ que posteriormente se van a simular, se emplea la aplicación denominada Microsoft Office Visio. Esta aplicación se encarga de simplificar y comunicar información compleja con diagramas vinculados a datos.

Gracias a esta aplicación, mediante los diferentes símbolos disponibles podemos diseñar el circuito digital deseado, escribiendo dentro de las cajas la lógica del circuito, obteniendo como salida un archivo con extensión .vdx que posteriormente compilaremos con el compilador de diagramas ASM++ y simularemos con el simulador de circuitos digitales desarrollado en este proyecto. La ventaja de usar los ficheros .vdx frente al formato por defecto .vsd es que el contenido del fichero está en formato XML, similar al HTML, lo que facilita el trabajo a las herramientas que trabajan con estos ficheros cómo será el caso del compilador de ASM++.

### 2.6.2. ASM++ Compiler:

El compilador de diagramas ASM++, es una aplicación de ordenador, desarrollada por epYme workgroup en la Universidad de Valladolid (Uva) de España en colaboración con eZono AG (Alemania) y ISEND S.A. (España). Esta aplicación tiene como fin compilar diagramas ASM++ generando como salida códigos de simulación válidos en lenguaje VHDL y Verilog.

Este compilador, aparte de generar como salidas archivos con circuitos digitales definidos en esos lenguajes para ser simulados, también permite la detección de posibles errores en la descripción del mismo ayudando al usuario a su detección.

La intención de los desarrolladores del compilador es poder tener una aplicación capaz de compilar los diagramas ASM++, detectando los posibles errores, y que también sea capaz de simular los circuitos una vez compilados sin necesidad de emplear otras aplicaciones externas para la comprobación

de la lógica general del diagrama ASM++. Hasta la terminación de este proyecto, esto último no era capaz de hacerlo el simulador, necesitando el apoyo externo de otros simuladores. Para evitar la necesidad de estos simuladores externos es por lo que surgió este proyecto, que tiene como fin desarrollar un primer simulador de circuitos digitales para incorporarse en un compilador de diagramas ASM++.

### 2.6.3. ModelSim:

ModelSim es una entorno de simulación en lenguaje HDL para ordenador, diseñada por Mentor Graphics. Esta aplicación tiene como fin realizar la simulación de circuitos digitales descritos en lenguajes VHDL, Verilog o SystemC, permitiendo además la compilación en lenguaje C de los mismos.

Gracias a este programa, a partir de los archivos obtenidos con el compilador de diagramas ASM++ mencionado en el apartado anterior, se puede realizar una simulación completa del circuito, obteniendo como salidas una visualización completa del circuito a lo largo de los ciclos de tiempo deseados, y el resultado de la comprobación de una serie de verificaciones deseadas para comprobar si el diseño del circuito digital es el correcto.

### 2.6.4. Lenguaje de programación C++:

El lenguaje de programación C++ se creó como una extensión del lenguaje de programación C para permitir la programación orientada a objetos. Actualmente es uno de los lenguajes de programación más populares empleados para la creación de software de sistemas y para aplicaciones.

Este lenguaje de programación dispone de una serie de elementos que permiten la correcta creación de las aplicaciones deseadas por los desarrolladores y programadores. A continuación se hace una breve descripción de los elementos que se han empleado en el simulador para facilitar la comprensión del código diseñado para este:

- **Tipos de datos:** pese a que el lenguaje C++ permite la creación de nuevos tipos de datos al programador, existen una serie de tipos de datos predefinidos que indican al ordenador los atributos de los datos con los que se va a trabajar. Estos tipos de datos fundamentales que emplea el lenguaje C++ son:
  - Caracteres: char
  - Enteros: short, int, long, long
  - Numerus en coma flotante: float, double, long double
  - Booleanos: bool
  - Vacío: void.



La mayoría de estos tipos de datos serán empleados en la definición de las diferentes variables empleadas en el desarrollo del simulador.

- **Funciones:** son un conjunto de instrucciones que realizan una tarea específica. Normalmente son capaces de tomar ciertos valores de entrada y proporcionar unos valores de salida. Estas funciones se declaran mediante uno de los tipos de datos indicados anteriormente y permiten estructurar de una forma más simple al programa global, englobando dentro de cada función una tarea concreta que puede ser empleada en diferentes partes del programa.
- **Función principal:** todo programa en C++ tiene una función principal, denominada main (), que sirve de punto de entrada del programa, a partir de la cual se puede ir accediendo al resto de funciones del programa.
- **Clases:** la programación orientada a objetos, como su propio nombre indica, se basa en la manipulación de objetos. Estos objetos son abstraídos mediante clases. Las clases tienen una serie de propiedades que las diferencia una de otras:
  - **Identidad:** nombre que lleva la clase a la que pertenece un objeto
  - **Método:** las funciones de la clase
  - **Atributos:** variables de la clase
- **Bibliotecas:** definidas al inicio de los diferentes archivos mediante la palabra #include, y son las encargadas de permitir a un programa disponer de las funciones y los elementos ya diseñados incluidos dentro de estas bibliotecas.

### 2.6.5. Microsoft Visual C++ 2010 express:

Visual C++ es una aplicación desarrollada por Microsoft con el objetivo de permitir la creación de aplicaciones orientadas a objetos para desarrollo en lenguajes C, C++ y C++/CLI.

Esta aplicación dispone de un entorno de desarrollo integrado que incluye un editor de textos para la edición de código fuente, un depurador, un compilador, un editor de interfaces gráficas GUI, permitiendo además la creación de aplicaciones de bases de datos.

Microsoft Visual C++ express es la versión gratuita disponible para descarga vía internet, la cual incluye las funcionalidades suficientes para el desarrollo del proyecto.



Visual C++ cuenta además con su propio compilador y otras herramientas como Debug... proveyendo además de diferentes bibliotecas y sockets para los diferentes sistemas operativos, las cuales pueden ser ampliadas.

El lenguaje de programación de esta herramienta está basado en C++, lo que la hace compatible en la mayor parte de su código con este lenguaje, teniendo una sintaxis prácticamente idéntica.

### **2.6.6. Librerías Qt:**

Qt es una infraestructura multiplataforma orientada a objetos empleada en la creación y desarrollo de aplicaciones de software informático. Esta infraestructura permite al desarrollador de software poder diseñar e incluir en su aplicación un interfaz gráfico de usuario.

Esta infraestructura está desarrollada como un software libre y de código abierto. Gracias a esto se ha podido emplear para el desarrollo de la aplicación de usuario del simulador sin necesidad de licencias.

Qt se diseñó originalmente para emplear el lenguaje de programación C++, sin embargo mediante una serie de bibliotecas ya existentes se pueden adaptar otros lenguajes de programación (Python, Java...) para ser empleador en el diseño de las aplicaciones.

Aunque existen más infraestructuras que permiten disponer de interfaces gráficas de usuario, debido a la cantidad de ejemplos ya disponibles para facilitar la creación de la interfaz de usuario que se desea para el simulador, así como por ser una infraestructura de software libre y por disponer de una cantidad muy elevada de funciones incluidas en sus librerías, se ha decidido apostar por esta plataforma para realizar el desarrollo y programación de la aplicación.



# Capítulo 3:

# Desarrollo del TFG

---

## Capítulo 3: Desarrollo del TFG

### 3.1. Teoría del simulador

En este apartado se realizará una descripción teórica de todo lo relativo al proyecto, desde las necesidades previas que motivaron a su desarrollo, hasta la situación final, en la cual se dispone de una aplicación con interfaz de usuario que incluye el simulador, el cual puede integrarse en el compilador de diagramas ASM++ y que es capaz de simular circuitos digitales que cumplan los requisitos descritos en las especificaciones iniciales necesarias para la finalización del proyecto.

Como se ha explicado en el Capítulo 2, para realizar la programación de una FPGA se pueden emplear los lenguajes de programación HDL. Como el diseño de circuitos lógicos complejos directamente en lenguaje HDL puede resultar muy laborioso existen entornos de desarrollo basados en esquemas que simplifiquen la programación como son ASM y ASM++, el cual es una ampliación del primero.

A continuación se muestra un esquema simplificador de lo mencionado anteriormente y en lo que se basaron los diseñadores del lenguaje ASM++ para apostar por su creación:

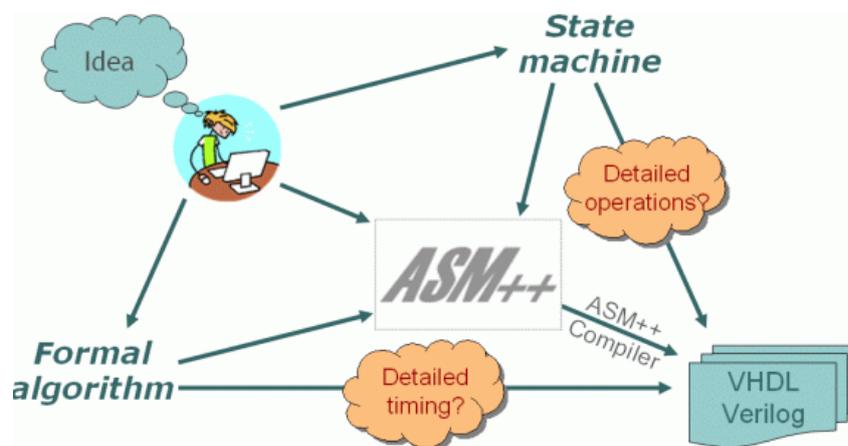


Figura 7: Esquema básico del proceso de creación de un circuito en lenguaje Verilog vía ASM++

En la figura 7 se muestra de forma muy simplificada el proceso de diseño de un circuito descrito en lenguaje HDL partiendo desde una idea presente en la mente del diseñador y mostrando donde se encuentran los diagramas ASM++ en todo este proceso.



Como se puede observar, para llegar a obtener un circuito lógico descrito en lenguaje HDL a partir de una idea existen varios caminos, los cuales, si se desea prescindir de los diagramas ASM++ se vuelven complejos. Sin embargo, estos diagramas permiten la existencia de un camino directo entre el circuito lógico que desea el diseñador y su descripción en lenguaje HDL.

Sin embargo, como se muestra en la figura, para poder convertir los diagramas ASM++ a lenguaje HDL fue necesaria la creación del compilador de diagramas ASM++, del cual se hacía una breve introducción en el Capítulo 2.

Gracias a este compilador, el diseñador puede disponer de un circuito lógico descrito en lenguaje HDL, el cual puede ser programado en una FPGA para realizar las funciones deseadas por el diseñador.

Sin embargo, aunque el compilador permite detectar fallos en el diseño y la descripción de la aplicación, no es capaz de detectar si la lógica del diseño es la deseada. Para poder comprobar el correcto funcionamiento del circuito diseñado, los diseñadores deben apoyarse en algún simulador existente, externo al compilador, que verifique el correcto funcionamiento de la aplicación diseñada sin necesidad de tener que esperar a tenerlo implantado físicamente en la FPGA.

La siguiente cuestión que surge en el paso de la descripción del circuito en lenguaje HDL a su programación en la FPGA es: ¿en qué momento realizar la simulación de la lógica del circuito diseñado?

Para responder a esta pregunta hay que estudiar como es el proceso de programación de la FPGA partiendo de un diseño inicial en lenguaje ASM++. Para simplificar la explicación se muestra la siguiente figura:

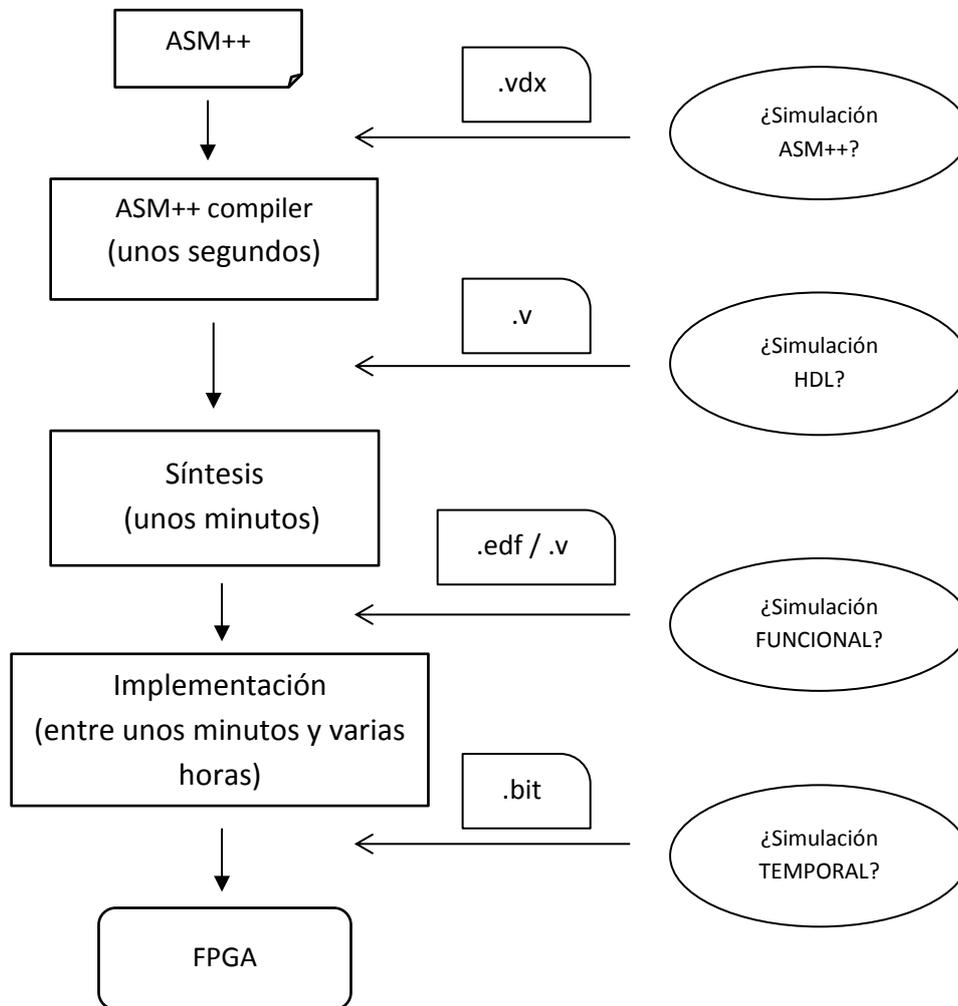


Figura 8: Etapas existentes en el proceso de programación de un diagrama ASM++ en una FPGA

En la figura anterior se muestran las diferentes etapas que aparecen en el proceso de programación de una FPGA a partir de un diagrama ASM++. A continuación se hace un breve análisis de cada parte del diagrama:

- **Diagrama ASM++:**

Hace referencia al archivo con extensión .vdx obtenido empleando Microsoft Office Visio y que sirve de entrada al compilador de diagramas ASM++. En él se describe totalmente tanto el diseño lógico del circuito como el test bench del mismo para realizar la simulación.

- **ASM Compiler:**

Como su propio nombre indica, hace referencia al compilador de diagramas ASM. Este compilador se encarga de transformar el archivo de entrada en extensión .vdx en una serie de lenguajes de programación (Verilog, VHDL...), creando una serie de ficheros de salida con diferentes extensiones (.vdo, .a++, .v, .do) y comprobando si hay algún error en la descripción del diseño.

- **Síntesis:**

El proceso de síntesis es el encargado de traducir el lenguaje HDL en su representación mediante puertas lógicas. Este proceso es relativamente rápido (minutos), siendo mayor o menor en tiempo dependiendo de la complejidad del diseño.

- **Implementación:**

La implementación hace referencia al proceso de programación del diseño en la FPGA, estableciendo la ubicación de cada bloque de puertas lógicas en el bloque lógico de la FPGA que crea conveniente, y realizando las diferentes interconexiones necesarias para el correcto funcionamiento de la FPGA. Simplificando, hace referencia a la instalación del diseño en el sistema físico. Este proceso (dependiendo de la complejidad del diseño) es el más elevado en tiempo, pudiendo tardar aproximadamente 1 hora.

- **FPGA:**

Este bloque hace referencia al dispositivo programable físico en el cual ira instalado el diseño realizado mediante el diagrama ASM++.

Una vez diferenciadas las distintas etapas existentes en el proceso de diseño y programación de un circuito digital, queda definir los tipos de simulaciones que pueden realizarse, comprobando sus ventajas e inconvenientes, y decidiendo así donde es más conveniente realizar la simulación del circuito lógico.

- **Simulación ASM++:**

Este tipo de simulación sería la realizada sobre el propio lenguaje ASM++ sin necesidad de compilación previa para adaptar el archivo .vdx obtenido en el diseño a un lenguaje HDL. Sería una simulación similar a la realizada en el lenguaje HDL, sin retrasos, con un análisis temporal dinámico y cumpliendo con las reglas de diseño de circuitos RTL.

- **Simulación HDL:**

Este tipo de simulación es la realizada sobre la descripción en lenguaje HDL del circuito digital. Al igual que la realizable en lenguaje ASM++, esta es una simulación sin retrasos, con un análisis temporal dinámico y cumpliendo con las reglas de diseño de circuitos RTL, lo cual permite obtener una simulación válida en un tiempo prácticamente insignificante.

Una de las ventajas con respecto a la simulación ASM++ es que los lenguajes HDL están mucho más internacionalizados.

- **Simulación Funcional:**

Se denomina simulación funcional a la realizada entre la síntesis del circuito y su implementación. En este caso, el archivo simulables estaría descrito indicando las diferentes puertas lógicas que lo forman. La síntesis del circuito a partir del lenguaje HDL lleva un cierto tiempo, y a pesar de que su simulación no incluye retrasos y sigue las reglas de diseño RTL tiene la desventaja de que su análisis temporal es estático.

- **Simulación Temporal:**

La simulación temporal, que toma como archivo simulable aquel en el que se define ya como es la programación del circuito en cada celda de la FPGA, indicando para cada una su circuito sintetizado con el diseño de puertas lógicas, es una simulación en tiempo real.

Esto provoca que la simulación temporal incluya retrasos a la simulación, aumentando enormemente su duración.



Como ya se ha comentado con anterioridad, el compilador de diagramas ASM ofrece diferentes archivos de salida con diferentes extensiones para permitir realizar una simulación HDL en diferentes programas o aplicaciones existentes como ModelSim o Simulink.

Como se ha visto, la simulación se puede realizar en diferentes zonas, pero dependiendo el momento en que se realice varía notoriamente tanto el tiempo como la complejidad del diseño. Para obtener una simulación rápida con un diseño sencillo, pero que a la vez sea robusta, la situación ideal para la simulación se encuentra antes y después del compilador de diagramas ASM.

En el caso de realizarse antes, se dispone de un archivo con extensión .vdx, el cual no está compilado para comprobar su correcto diseño. A la salida del compilador obtenemos un diseño en lenguaje HDL, con diferentes extensiones (incluida la extensión .vdo), compilado correctamente. Este es el momento idóneo para realizar la simulación.

La necesidad de emplear aplicaciones externas diseñadas por empresas o grupos diferentes a los creadores del compilador de diagramas ASM, y por lo tanto la dependencia de estos programas externos, es el principal problema que presentaba la situación previa a la realización del simulador.

Cualquier cambio realizado en estos programas externos por sus creadores o por los diseñadores de las empresas encargadas de su desarrollo podría crear fallos en las simulaciones, obligando a los diseñadores del compilador de diagramas ASM a modificar constantemente su programa para poder obtener como salida archivos validos de simulación.

Para evitar estos problemas de dependencia de aplicaciones externas nació este proyecto, permitiendo a los diseñadores del compilador introducir en su propio compilador un simulador capaz de mostrar tras la compilación el resultado de la simulación, en un tiempo bastante reducido (comparado con el resto de simulaciones posibles definidas anteriormente) y permitiendo además tener total control sobre el código del simulador, añadiendo o eliminando las funcionalidades que se estimen oportunas, y realizando la simulación de la manera que estimen más adecuada.

A continuación se va a describir una serie de conceptos con el objetivo principal de conocer completamente el funcionamiento de los diagramas ASM++ y del lenguaje Verilog y así permitir al lector entender porque se ha optado por realizar la programación del simulador de la manera en que se ha desarrollado.

### 3.1.1. Expresiones en lenguaje Verilog:

La mayoría de lenguajes existentes tienen definidas una serie de prioridades para sus elementos, que permiten a unos tener más peso que otros en el propio lenguaje. Por ejemplo, un lenguaje mundialmente conocido como es el lenguaje matemático, como se muestra en la siguiente figura, tiene establecidas una serie de prioridades a sus operadores, indicando con ello que en el caso de aparecer 2 operadores de diferente prioridad en una misma operación, primero deberá ejecutarse el operador con mayor prioridad.

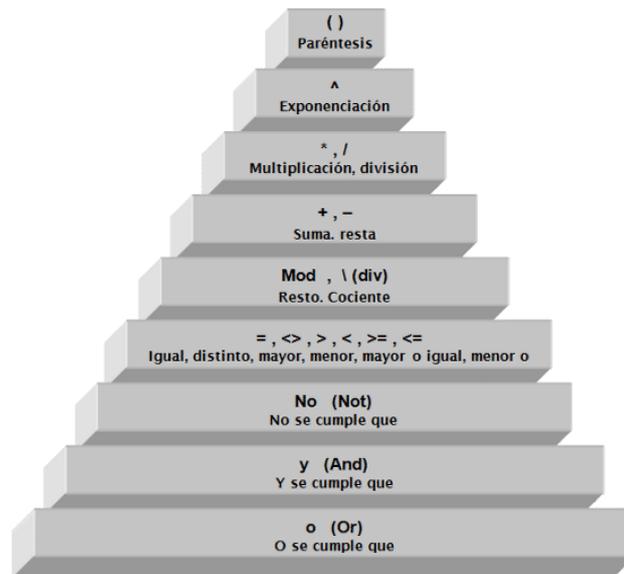


Figura 9: Pirámide de prioridades del lenguaje matemático

En el caso de los lenguajes HDL pasa exactamente lo mismo, existen una serie de reglas o directrices que indican que operadores se deben realizar con más prioridad que otros. Para poder visualizar de una forma más fácil estas reglas, a continuación se muestran 2 esquemas que se encargan de indicar que operaciones prevalecen sobre otras en los 2 principales lenguajes HDL:

○ Verilog:

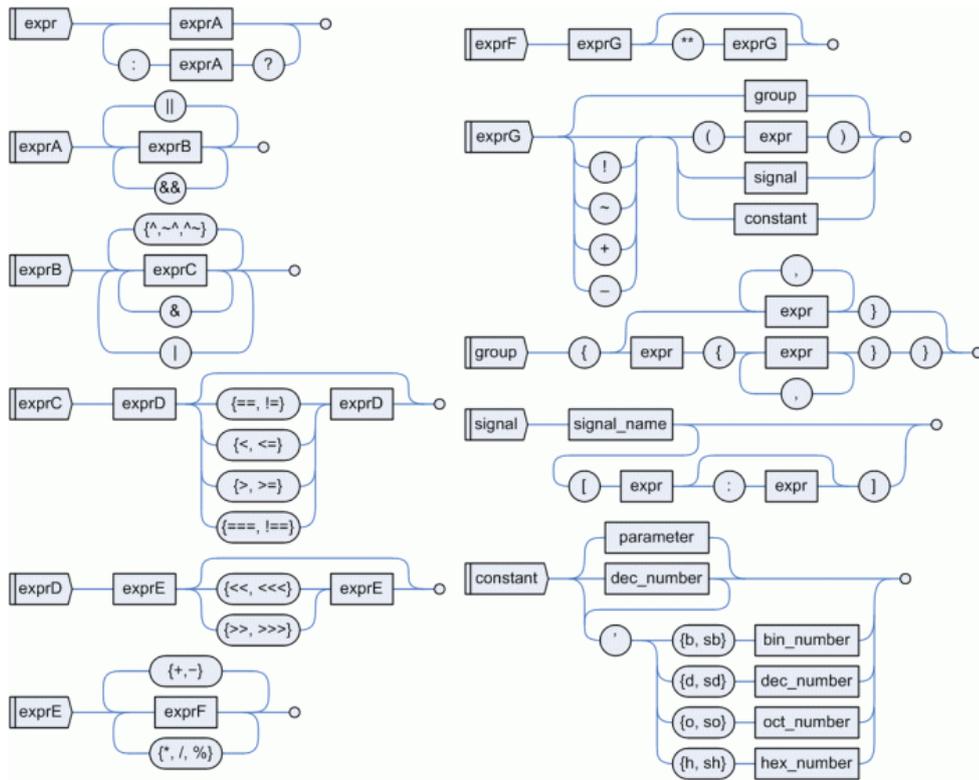


Figura 10: Estructura de las expresiones en lenguaje Verilog.

○ VHDL:

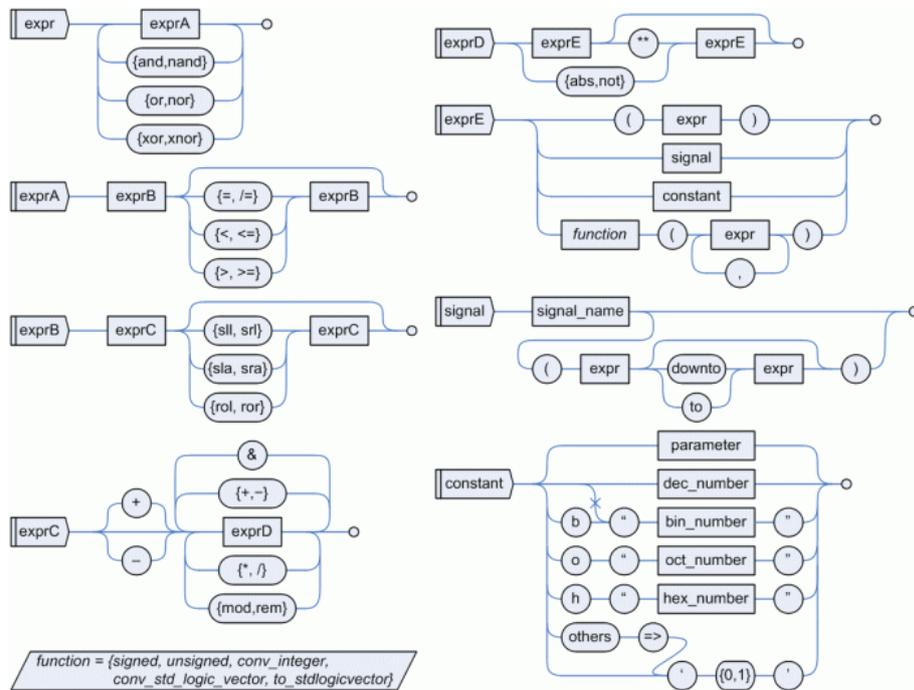


Figura 11: Estructura de las expresiones en lenguaje VHDL



Como se puede observar en las dos figuras anteriores, pese a que existen algunas diferencias tanto en la definición de algunas operaciones y expresiones como en el uso de algunos operadores, la lógica que siguen las expresiones tanto en lenguaje VHDL como Verilog es muy similar, lo cual permite poder pasar de un lenguaje a otro sin excesiva complicación. Es por esto por lo que el desarrollo del simulador se propuso únicamente en lenguaje Verilog, puesto que una vez se disponga de un programa capaz de simular circuitos en lenguaje Verilog, la modificación de este para detectar expresiones en VHDL es bastante sencilla.

Dejando de lado el lenguaje VHDL (que no se va a emplear para el desarrollo de la programación del simulador), en este apartado se va a explicar detalladamente el funcionamiento de la lógica de las expresiones en lenguaje Verilog apoyándose en la Figura 10.

Como se observa en el esquema de la figura 10 una expresión inicial en lenguaje Verilog (Expr) está formada por una expresión secundaria (ExprA) seguida o no de un interrogante “?” y de otra Expresión secundaria (ExprA). A su vez, esas expresiones secundarias (ExprA) están formadas por expresiones terciarias (ExprB) seguidas o no de una serie de operadores lógicos (que se explicaran a continuación) y de otra expresión terciaria (ExprB). Estas expresiones a su vez, están formadas por otras expresiones (ExprC) y así sucesivamente como se indica en el diagrama hasta llegar hasta las expresiones de más prioridad, las cuales no tienen expresiones dentro de ellas (números binarios, decimales, parámetros constantes...).

Aunque se ha dicho que la expresión final es una expresión sin capacidad de desglosarse en otro tipo de expresión, puede darse el caso de que se incluyan grupos de expresiones o expresiones dentro de paréntesis “( )” las cuales incluyan expresiones iniciales repitiendo el proceso entero.

Evidentemente, una expresión en lenguaje Verilog no tiene por qué tener absolutamente todos los operadores y todos los tipos de expresiones (ExprA, ExprB, ExprC...). Una expresión Verilog puede ser simplemente un número decimal, o una suma de Expresiones F sin necesidad de tener expresiones secundarias (ExprA).

Para aclarar un poco más esta explicación se adjunta la siguiente figura en forma de árbol, para demostrar como es el desglose de las expresiones:

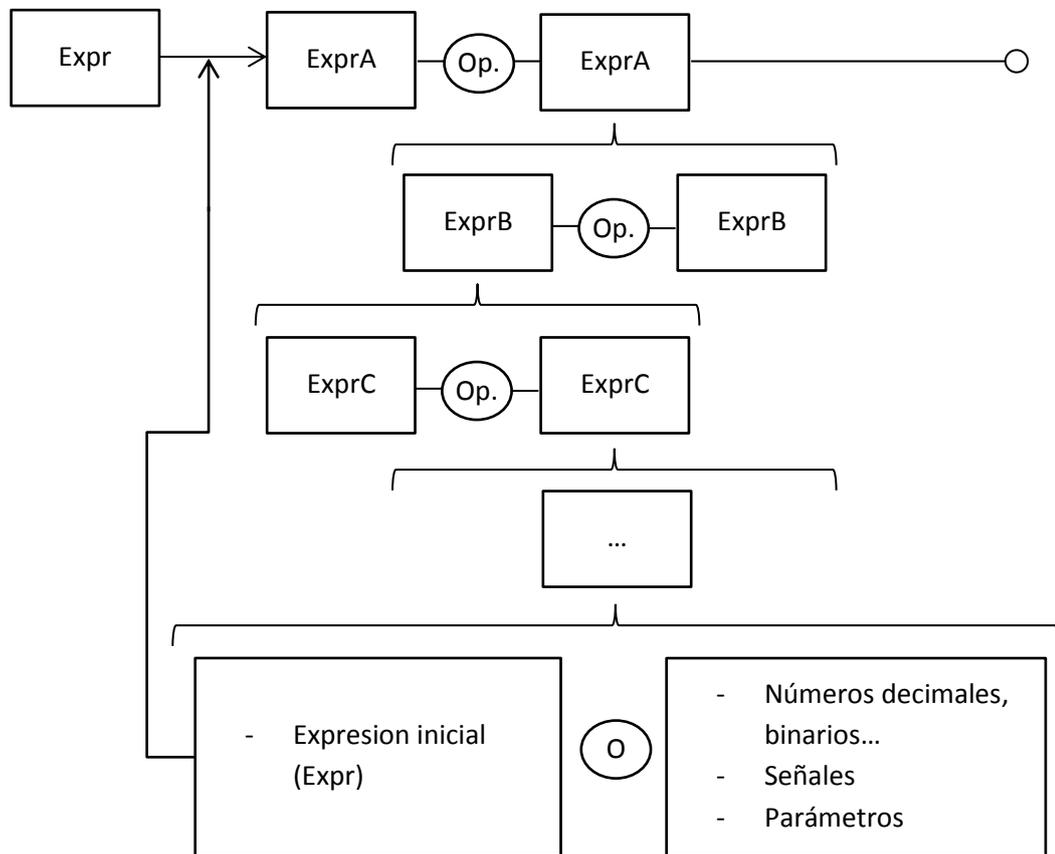


Figura 12: Lógica de las expresiones en el lenguaje Verilog. Diagrama en árbol

Una vez entendida la lógica que emplean las expresiones en lenguaje Verilog, es importante conocer el significado de los siguientes símbolos y operadores que aparecen en él. Para ello, se explicará que tipos de operaciones se realizan en cada nivel de expresiones, partiendo desde las expresiones iniciales (Expr) hasta las expresiones finales (señales, números decimales...).



○ **Expresiones condicionales (Expr):**

Estas expresiones iniciales se encargan de realizar operaciones condicionales de Expresiones A. En este nivel se encuentran las interrogaciones “?” y los dos puntos “:”.

Estas expresiones devuelven un valor u otro dependiendo del resultado lógico (verdadero o falso) de la expresión A que se encuentra antes del interrogante.

Ejemplo:

$$A = (5 > 6) ? 7 : 8 \quad \Rightarrow \quad A = 8$$

$$A = (5 < 6) ? 7 : 8 \quad \Rightarrow \quad A = 7$$

En el primer ejemplo, como la expresión lógica situada antes del interrogante es falsa, el resultado obtenido es el de la expresión situada después de los 2 puntos.

En el segundo ejemplo, al ser una expresión lógica verdadera el resultado es el obtenido antes de los 2 puntos.

○ **Expresiones lógicas con salida lógica (ExprA):**

Estas expresiones incluyen todas las operaciones lógicas que dan como salida un resultado lógico (verdadero o falso).

Emplean 2 tipos de operadores diferentes:

- **&&:** “y” lógica. El resultado será la combinación de los dos operandos lógicos. Para ser verdadero ambos operandos deben serlo.
- **||:** “o” lógica. El resultado será la combinación de los dos operandos lógicos. Para que sea verdadero, simplemente un operando debe serlo.

En el caso de aparecer seguidos ambos operadores, el operador && tiene prioridad sobre el operador ||.



○ **Expresiones lógicas de bit con salida numérica (ExprB):**

Estas expresiones incluyen todas las operaciones lógicas que se pueden realizar con los bits de los operandos. Incluyen los siguientes operadores listados de mayor a menor prioridad:

- **&**: AND bit a bit.
- **^, ~^, ^~**: XOR y NXOR bit a bit.
- **|**: OR bit a bit.

En la tabla 3 de este capítulo está incluida la tabla de la verdad de cada una de las diferentes operaciones lógicas de bit.

○ **Expresiones comparativas (ExprC):**

Estas expresiones incluyen todas las operaciones comparativas de igualdad o relacionales incluidas en el lenguaje Verilog. Devuelven como salida un resultado lógico (verdadero o falso). Pueden aparecer los siguientes símbolos:

- **==, !=** : igualdad y desigualdad simple. El primero devuelve verdadero si los operandos son idénticos, y falso en caso contrario. El segundo indica desigualdad devolviendo como valores los contrarios a los anteriores. Si algún bit es "X" el resultado será "X".
- **<, <=, >, >=** : relacionales. Permiten comparar dos operandos, retornando verdadero o falso dependiendo de la lógica de la operación. Poseen el significado habitual de otros lenguajes.
- **===, !==** : su significado es similar al de igualdad y desigualdad simple, pero además compara los bits indefinidos "X" y de alta impedancia "Z".

- **Expresiones de desplazamiento (ExprD)**

Estas expresiones se encargan de realizar desplazamientos de bits. Emplean los siguientes operandos:

- **<<, >>**: desplazan a la izquierda o a la derecha respectivamente, añadiendo ceros (0), tantos bits como indique la expresión E que les precede.
- **<<<, >>>**: desplazan a la izquierda o a la derecha respectivamente, añadiendo tantos 0 ó 1, dependiendo el ultimo (desplazamiento izquierda) o el primer bit (desplazamiento derecha), como indique la expresión E que le precede.

- **Expresiones aritméticas (ExprE):**

En este nivel se realizan todas las expresiones aritméticas (salvo las exponenciales) que se pueden realizar en el lenguaje Verilog. Se emplean los siguientes símbolos para cada una de ellas:

- **+, -**: suma y resta aritmética.
- **\*, /, %**: multiplicación, división y resto.

Al igual que en el lenguaje matemático, las operaciones de multiplicación, división y resto tienen prioridad sobre la suma y la resta.

- **Expresiones exponenciales (ExprF):**

En este nivel se realizan las operaciones exponenciales. Se identifican con el operador “\*\*” y se encarga de elevar la expresión G previa al operador “\*\*” a la exponencia indicada por la segunda expresión G.



- **Expresiones de signo (ExprG):**

Estas expresiones realizan se encargan de definir el signo del resto de expresión que les precede.

Contienen los operadores siguientes:

- **!:** negación lógica. Cambia el valor lógico del operando que va justo detrás del operador. Devuelve un valor lógico (verdadero o falso).
- **~:** negación bit a bit.
- **+, -:** signo positivo y negativo respectivamente.

Aparezcan o no los operadores indicados, las expresiones G están formadas por diferentes tipos de expresiones, entre las que se incluyen los grupos de expresiones (Group), una nueva expresión inicial entre paréntesis (“(Expr)” ), señales o constantes.

- **Grupos:**

Los grupos (Group) de expresiones, como su propio nombre indica, son agrupaciones de expresiones iniciales separadas por comas o corchetes. Estos grupos tienen como inicio una llave de inicio “{“ y como fin una llave de finalización “}”.

- **Señales:**

Las señales están formadas por el nombre de una señal, seguido o no de un corchete que indica el bit (o conjunto de bits) de la señal a los que se refiere la expresión.

- **Constantes:**

Las expresiones denominadas constantes se pueden separar en dos grupos:

- **Parámetros:** palabras o letras que hacen referencia a constantes o variables.
- **Números** en los diferentes sistemas aceptados por el lenguaje Verilog (decimal, binario, octal o hexadecimal) diferenciados por un apostrofe ‘ seguido de:
  - **b, sb:** binario
  - **d, sd:** decimal
  - **o, so:** octal
  - **h, sh:** hexadecimal.

En este apartado han quedado definidos todos los posibles operadores existentes en el lenguaje Verilog con su estructura. En el caso de que alguna expresión no siguiera el diagrama indicado en la figura 10 no sería una expresión válida en este lenguaje.

### 3.1.2. Estructura de los diagramas ASM++:

Como ya se mencionaba en el capítulo 2, los diagramas ASM++ (Extender Algorithmic State Machine) son un lenguaje gráfico creado con el objetivo de ser usado como un interfaz gráfico para generar circuitos digitales de forma automática.

Estos diagramas están formados por una serie de elementos con la capacidad de interactuar entre sí, mostrados en las tablas 1 y 2 de este capítulo, los cuales se encargan de definir diferentes operaciones de los circuitos lógicos digitales. Sin embargo, estos elementos no tienen libertad total para interconectar entre sí. Los diagramas ASM++ presentan una serie reglas y zonas que se encargan de definir la estructura que deben seguir todos los circuitos digitales diseñados en este lenguaje. A continuación se muestra un diagrama explicativo de esta estructuración:

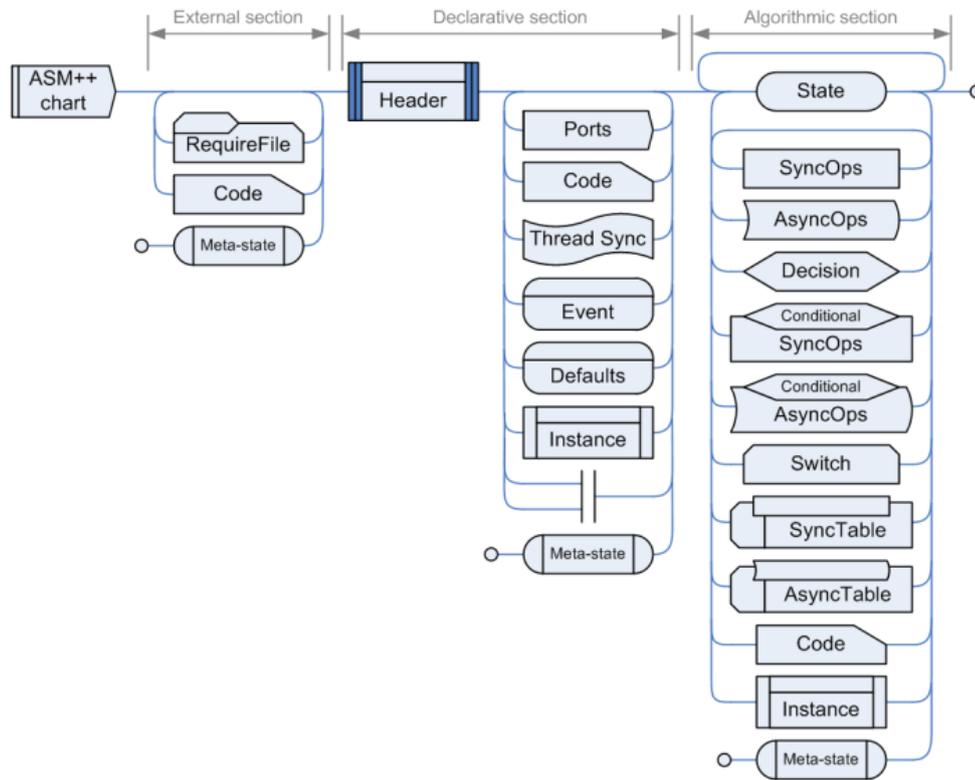


Figura 13: Estructura de los diagramas ASM++

Todos los diagramas ASM++ para ser validos deben cumplir con la estructuración indicada en la figura 13.

En esta estructuración se pueden diferenciar 3 zonas o secciones, con funciones diferentes. A continuación se realiza una explicación detallada de cada una de ellas:

- **External section:**

La sección externa, es la primera sección presente en cada diagrama ASM++, siendo una de las cajas incluidas de esta sección el inicio de todos y cada uno de los diseños presentes en un diagrama ASM++.

Como se muestra en la Figura 13, esta sección no es imprescindible para la correcta definición de un diagrama ASM++, puesto que de existir solo se emplearía para incluir información relativa al diagrama que le precede, sin afectar a la lógica del diseño. También se puede emplear para incluir ficheros con otros diagramas ASM++, o para incluir un final de un diseño.



Las cajas que pueden aparecer en esta sección son las siguientes:

- **RequireFile:** el objetivo de esta caja es hacer una llamada a un fichero con otro diagrama ASM++.
- **Code:** en las cajas code de la sección externa se suelen incluir comentarios informativos acerca del diseño del diagrama ASM++, indicados con una almohadilla inicial en cada línea de texto “#”.
- **MetaState:** un bloque MetaState es un bloque encargado de indicar el fin de un diagrama ASM++.
- **Declarative section:**

La sección declarativa es la zona del diagrama ASM++ encargada de definir todas las señales, parámetros e información necesaria para que la lógica del diagrama ASM++ pueda ejecutar sus operaciones de forma correcta.

Obligatoriamente, en caso de que no tengamos un diagrama con únicamente zona externa, todos los diagramas ASM++ deben incluir un box cabecera denominado “Header”. Esta caja es la encargada de definir el nombre del diagrama que le precede.

Una vez pasada la caja header pueden aparecer una serie de cajas con diferentes funcionalidades, encargadas de definir toda la información necesaria para el resto del circuito. A continuación se muestra un listado de cada tipo de caja con una explicación de su funcionamiento:

- **Header:** indica el inicio de esta sección. Las cajas header se interpretan correctamente, almacenando el nombre del diseño que le precede.  
Aparte de indicar el nombre, estas cajas en su parte inferior pueden realizar la definición de alguna constante que se empleará a lo largo del circuito.

En el diseño del circuito que se ha empleado para verificar el correcto funcionamiento del simulador aparecen 2 cajas de este tipo, indicando tanto el principio del diseño de la lógica y del test bench, marcando su nombre, como la definición de la constante N, que indica el número de bits de las señales. Esta constante se ha dejado en el diseño pese a que no se empleará al no estar implementados los parámetros.



Figura 14: Cajas Header de diagramas ASM++ (DISEÑO Y TEST BENCH)

Una vez definida la caja Header, puede aparecer directamente la sección Algorítmica, o aparecer una o más de una de las cajas mostradas a continuación:

- **Ports:** los bloques Ports se encargan de definir todas las señales de entrada (Inputs) y salida (Outputs) del diseño de un circuito.

Para realizar esta definición se emplean las palabras “Input” o “Output” seguidas o no del rango de bits de la señal, y el nombre de las diferentes señales con estas características que aparecerán en el circuito.

En el diseño realizado para la comprobación del simulador, aparece un bloque Ports, que se encarga de definir 5 señales de entrada y 3 señales de salida. Este bloque no indica el sincronismo que tendrá cada tipo de señal. En la siguiente figura se muestra el bloque Ports del diseño mencionado:

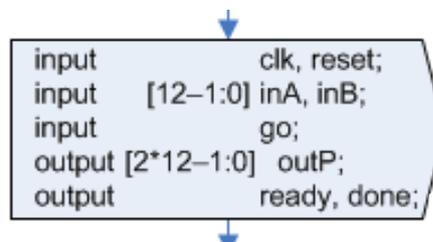


Figura 15: Box Ports

- **Code:** este tipo de caja, que también aparece en el resto de secciones, tiene un uso diferente aquí al que puede tener en otras partes. En esta sección, esta caja es la encargada de inicializar las señales internas tipo “Wire” e “Inout” presentes en el resto del circuito, siguiendo la misma estructura empleada por la caja Ports en la definición de las señales “Input” y “Output”.

También se puede emplear, como en el caso del test bench del diseño empleado para comprobar el funcionamiento del simulador, para establecer variables empleadas en la simulación. Esta última parte no está implementada aún en el simulador.

A continuación se muestran las cajas Code presentes en las secciones declarativas del diseño del circuito:

```
wire [12-1:0]regA;  
wire [2*12-1:0]regB;  
wire [3:0] regJ;  
  
#view -hex $$.*;
```

Figura 16: Box Code de las secciones declarativas

Como se puede observar en la figura, en este caso no aparecen señales tipo “Inout” en el diseño, solamente aparecerán señales tipo “Wire”. En la caja de la derecha se muestra la definición del parámetro \$\$ para hacer referencia a todas las señales del diseño presentes en un test bench.

- **ThreadSync:** este bloque, que de aparecer se encuentra situado siempre en la sección declarativa de los diagramas ASM++, se encarga de establecer el nombre y la polaridad de la señal de sincronización usada por un circuito o por un hilo del mismo. Esta sincronización afecta a todos los bloques que se encuentran situados después de él.

En el diseño empleado para la comprobación del funcionamiento del simulador aparece este bloque indicando que todo el diseño del circuito trabaja síncronamente con la señal de entrada “clk” como se muestra en la siguiente figura:

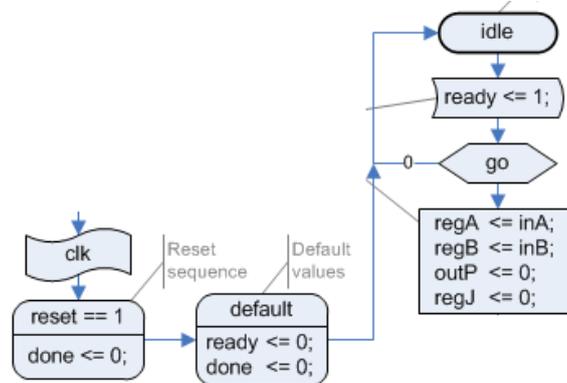


Figura 17: Diseño síncrono la señal clk

- **Event e Initial:** este tipo de cajas, presentes en la sección declarativa, se encargan de establecer un valor concreto a las diferentes señales indicadas cuando ocurre un evento marcado dentro de la propia caja.

Estas cajas están formadas por 2 apartados:

- **Texto superior:** se encargan de indicar la condición que marca la ejecución de la caja. Puede ser una expresión en lenguaje HDL como la mostrada en el bloque de la izquierda de la figura 18 (`reset == 1`) o con la palabra “initial” como en el bloque de la derecha, que indica el valor inicial que tendrán las señales indicadas.
- **Texto inferior:** en esta zona se indica en lenguaje HDL el valor de la señal al ejecutarse la caja correspondiente.

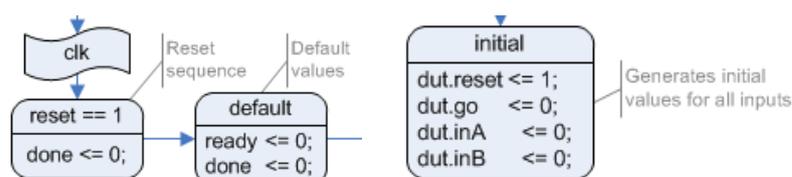


Figura 18: Cajas Event e Initial del diseño



- **Default:** este bloque, mostrado en la figura anterior, es similar al bloque Initial o event, encargándose de establecer el valor por defecto de unas señales indicadas en la parte inferior.
- **Instance:** este tipo de caja se encarga de instanciar un módulo de menor nivel, definiendo otro diagrama ASM++ o código HDL. También se emplea en el diseño del test bench, como es nuestro caso, para instanciar el “Device under test” (dut) necesario en todos los test bench, así como de declarar las variables empleadas por este.
- **||:** este bloque hace referencia a una doble línea, y se emplea para establecer hilos paralelos que serán ejecutados concurrentemente. Con este bloque se pueden describir múltiples circuitos con igual o diferente sincronización.
- **Metastate:** este bloque, presente también en el resto de secciones, es el encargado de indicar el fin de un diagrama ASM++.

En todas estas cajas se realizan inicializaciones o definiciones de señales o de parámetros del diseño del diagrama, pero sin realizar ningún tipo de operación algorítmica.

El fin de esta sección estará marcado por una caja MetaState o por una caja de estado (State), en caso de existir una zona algorítmica en el diseño del diagrama.

A continuación se muestra señalada cual sería la zona declarativa en el diagrama ASM++ que se empleara para el desarrollo y la verificación del correcto funcionamiento del simulador:

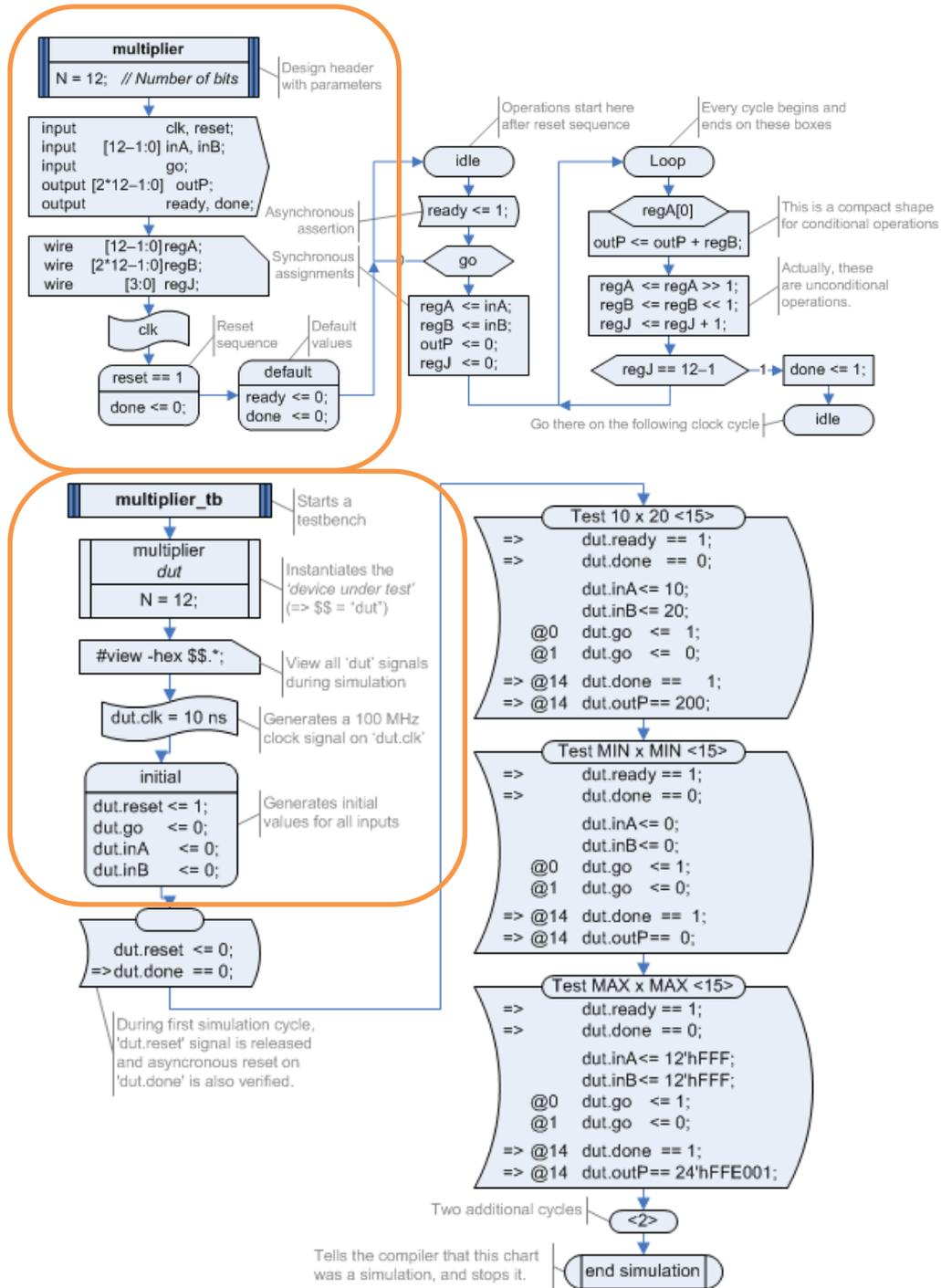


Figura 19: Secciones declarativas en el diagrama ASM++ empleado en la simulación



Como se observa en la Figura 19, los diagramas ASM++ tanto del diseño como del test bench carecen de zona externa, teniendo como inicio la sección declarativa. Esta sección declarativa tiene como inicio una caja Header, y como fin el primer estado del diagrama.

En el diseño superior, cuyo fin es definir la lógica del circuito digital, aparece en primer lugar la caja header, encargada de definir el circuito con el nombre “multiplier”. Además, se encarga de definir en su parte inferior el valor de una constante que se podrá emplear a lo largo del diseño del circuito.

A continuación aparecen las cajas Ports y Code, encargadas de definir todas las señales que se emplearán en el circuito. En estas cajas no se define si las señales tienen carácter síncrono o asíncrono, simplemente se encargan de definir su tipo y las características de cada una de ellas (número de bits).

En las cajas Ports se definen los 2 tipos de señales que interactúan con el exterior como son las señales de entrada (Inputs) y las de salida (Output).

En las cajas Code se definen los 2 tipos de señales internas del circuito, como son las señales Wire e Inout.

Una vez definidas todas las señales empleadas por el circuito, a continuación aparece un bloque Thread Sync que tiene como finalidad indicar la señal que servirá de sincronismo del diagrama que le precede.

Para terminar esta sección, en el diagrama de la figura aparecen los bloques Event y Default, que se encargan de definir el valor de las señales (en la parte inferior) cuando suceden los eventos indicados en su parte superior.

○ **Algorithmic section:**

La sección algorítmica, es la zona de los diagramas ASM++ encargada de realizar todas las operaciones algorítmicas del circuito digital. En esta sección, indicada en la siguiente figura, pueden aparecer las cajas o boxes indicados a continuación de la misma.

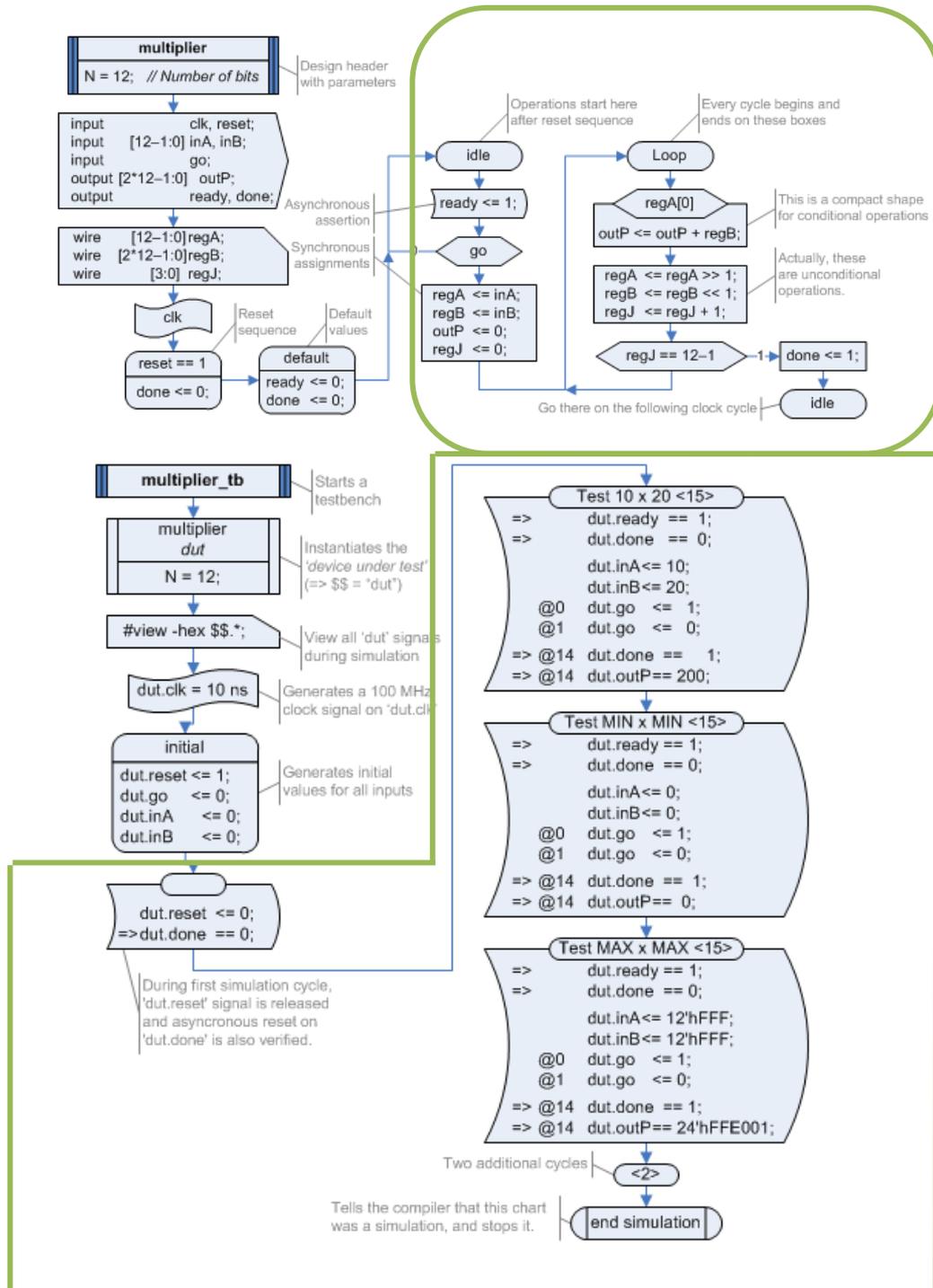


Figura 20: Secciones algorítmicas en el diagrama ASM++ empleado en la simulación

- **State:** como ocurría en la sección declarativa con los bloques Header, en la sección algorítmica siempre aparecerá un bloque State al inicio. Este bloque indica el principio y el final de cada ciclo de reloj. En su interior, en caso de no estar vacío, se indica el nombre del estado.

Cada bloque State tiene un nombre único e irrepetible en el mismo diseño. En caso de aparecer 2 bloques State con el mismo nombre en su interior se estarán empleando para unir sin necesidad de flechas 2 puntos del diagrama.

Una vez definido algún bloque State, después pueden aparecer una serie de bloques que se encargan de realizar tanto el diseño de la lógica como del test bench para la simulación:

- **SyncOps:** este bloque aparece únicamente en la sección algorítmica, y se encarga de realizar todas las operaciones descritas en el lenguaje HDL con la condición de que solo pueden ejecutarse simultáneamente al finalizar el ciclo de reloj. Además este bloque sirve para definir si una señal es síncrona o no puesto que todas las señales presentes en un bloque de este tipo automáticamente son síncronas.
- **AsyncOps:** este bloque tiene una función similar al anterior, pero la única diferencia es que las operaciones descritas en este bloque se realizan asíncronamente durante los ciclos de reloj. Al igual que en el bloque anterior, todas las señales presentes en estos bloques son asíncronas.
- **Decision:** este bloque tiene como finalidad decidir qué camino seguirá un diagrama ASM++ en su ejecución. En su interior aparecerá una expresión lógica en lenguaje HDL, y dependiendo de si su resultado es verdadero o falso el circuito seguirá un trayecto u otro.

Para visualizar mejor estos últimos 3 bloques se muestra en la siguiente figura una parte del diagrama empleado para comprobar el funcionamiento del simulador, en el cual se muestra un bloque State de inicio, 1 bloque Asíncrono y otro síncrono y un bloque decisión:

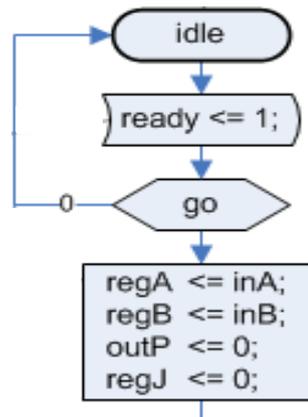


Figura 21: Bloques AsyncOps, SyncOps y Decision

Como se muestra en la figura, en bloque State denominado “idle” se marca el inicio de un ciclo de reloj. Durante ese ciclo se ejecutara el bloque Asíncrono, poniendo la señal ready a 1. Después llegará al bloque decisión, comprobará el valor de la señal “go” y dependiendo de su valor volverá al estado inicial para poner fin al ciclo de reloj actual ( go=0 ) o ejecutara el bloque síncrono al finalizar el ciclo de reloj (go = 1).

- **Conditional SyncOps:** este bloque es una mezcla del bloque decisión y el bloque SyncOps. En su parte superior se encuentra una condición lógica en lenguaje HDL, y dependiendo de su resultado se ejecutará el texto descrito en lenguaje HDL en su parte inferior (condición verdadera) o no (condición falsa) al finalizar el ciclo de reloj.
- **Conditional AsyncOps:** este bloque es una mezcla del bloque decisión y el bloque AsyncOps. Como el bloque anterior, en su parte superior se encuentra una condición lógica descrita en lenguaje HDL y dependiendo de su resultado se ejecutará el texto descrito en lenguaje HDL en su parte inferior durante el ciclo de reloj.
- **Code:** este bloque, similar al que aparecía en el resto de secciones con el mismo nombre, tiene como finalidad permitir escribir en el directrices de compilación (con una # inicial) o describir código en lenguaje HDL.
- **Instance:** al igual que el bloque que aparecía en la sección declarativa con el mismo nombre, este box tiene como finalidad instanciar un bloque de menor nivel usando otro diagrama ASM++ o código VHDL.



- **Metastate:** al igual que pasaba con los bloques de otras secciones con el mismo nombre, este bloque se encarga de indicar el fin de un diagrama ASM++.
  
- **Switch:** este bloque se podría definir como una ampliación del bloque decisión. Este bloque se encarga de elegir el camino que debe seguir un circuito digital, teniendo múltiples decisiones simultáneas.
  
- **SyncTable:** este bloque tiene como finalidad modificar el valor de la señal que aparece en su texto superior, dependiendo del valor indicado en la parte inferior, de forma síncrona con una señal de reloj. Esta parte inferior tiene el mismo funcionamiento que un bloque Switch. Por simplificar su explicación, este bloque es una mezcla entre un box Switch y un box SyncOps.
  
- **AsyncTable:** el funcionamiento de este bloque es similar al del anterior, pero el valor de la señal indicada en el texto superior se modifica asíncronamente, independiente del ciclo de reloj. Por simplificar su explicación, este bloque es una mezcla entre un box Switch y un box AsyncOps.

Para entender mejor el funcionamiento de estas últimas 3 cajas, en la figura 22 se muestra el diseño de un circuito lógico que incluye el bloque Switch y los bloques SyncTable y AsyncTable:

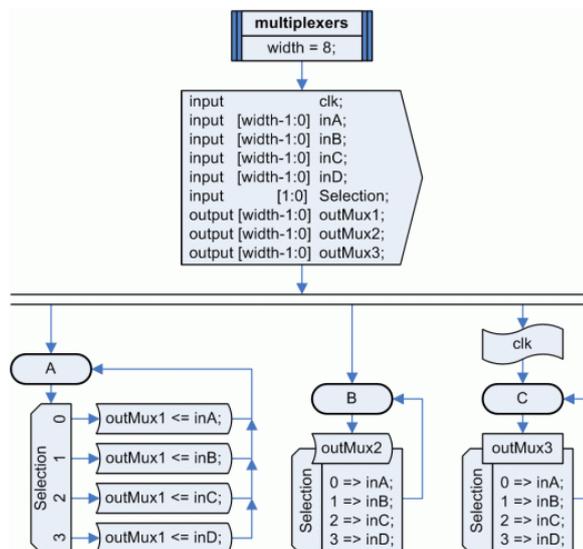


Figura 22: Cajas Switch, AsyncTable y SyncTable

En este diagrama, en primer lugar aparece la sección declarativa, formada por el box Header, el box Ports (encargado de definir las señales de entrada y salida del circuito) y una doble línea para crear 3 hilos diferentes. A continuación aparece la sección algorítmica, que tiene como inicio 3 bloques State (uno para cada hilo).

El diagrama del estado A, está formado por un bloque Switch seguido de 4 bloques AsyncOps. Dependiendo el valor de la señal Selection, la señal outMux1 tendrá un valor u otro indicado por los bloques AsyncOps.

El diagrama del estado B, formado por una tabla Asíncrona, tiene el mismo funcionamiento lógico que el estado A. Dependiendo del valor de la señal Selection, la señal outMux2 tendrá durante el ciclo de reloj un valor u otro.

El diagrama del estado C, formado por una tabla síncrona, tiene el mismo comportamiento lógico que el estado B, salvo que en este caso, al ser un bloque SyncTable, la señal outMux3 solamente obtendrá el valor indicado por la tabla al finalizar el ciclo de reloj, puesto que se encuentra sincronizado con la señal de reloj clk.



A todas estas cajas mostradas hasta ahora, en esta sección habría que añadir 2 tipos de cajas más que no aparecen incluidos en la figura 13. Estas cajas son: **StateSyncOps** y **StateAsyncOps**. Estos 2 bloques son una unión entre las cajas State y Sync-AsyncOps.

Para que un diagrama ASM++ sea válido, debe cumplir la estructuración indicada en este apartado. En caso contrario, el diagrama ASM++ no estaría diseñado correctamente.

En caso de cumplir la estructura indicada anteriormente, otro concepto a tener presente en los diagramas ASM++ es el objetivo de su diseño.

Para el correcto funcionamiento de un diagrama ASM++ al ser programado físicamente en un dispositivo programable como una FPGA, no basta solamente con diseñar la lógica que tendrá el circuito, sino que también es necesario crear lo que se conoce como un test bench (banco de pruebas).

Este test bench, es un diagrama ASM++, tiene como finalidad indicar una serie de directrices empleadas para verificar el correcto funcionamiento de un diseño.

Tanto el diseño lógico del programa como el del test bench deben cumplir unos requisitos imprescindibles para su correcto funcionamiento. A continuación se indican las diferentes partes que deben tener ambos diseños. Para entender mejor su explicación se empleará el diseño empleado para comprobar el correcto funcionamiento del simulador.

○ **Diseño de la lógica del circuito:**

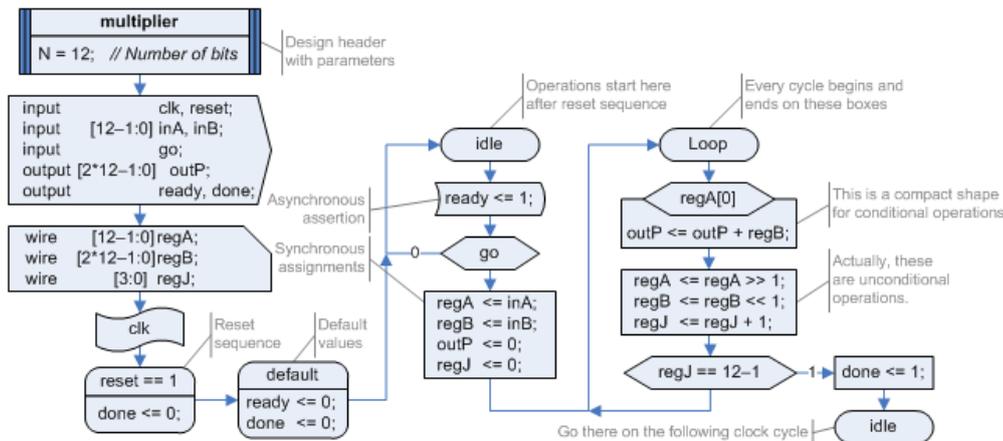


Figura 23: Diagrama ASM++ con el diseño de la lógica del circuito

El diseño de la estructura de la lógica del circuito (el diseño del circuito digital) en los diagramas ASM++ se basa en todo lo explicado anteriormente. Para ser un diseño válido, el diseño tiene que seguir la estructura marcada en la figura 13, contando con una zona externa (opcional), una zona declarativa (con inicio en la caja Header que da nombre al diseño, y con fin en el último box antes del primer box state) y con una zona algorítmica (con inicio en el primer box state y con fin en un box MetaState o sin fin, como ocurre en el diseño de esta figura).

Como hasta ahora ya se ha explicado básicamente todo el diseño con sus diferentes partes, en este apartado se va a explicar únicamente el único concepto que falta por explicar, que es la finalización de un ciclo de reloj y el paso al siguiente, y que es muy importante a la hora de realizar el desarrollo del código del simulador.

A la hora de realizar la simulación, los bloques State se encargan de indicar el inicio y el fin de un ciclo de reloj, pero ¿cuándo se puede dar por finalizado un ciclo de reloj?

Para responder a esta pregunta existe una condición: **solo se puede pasar de un ciclo de reloj a otro cuando se comprueba que el valor de cualquier señal no ha sido modificado durante el ciclo de reloj.** En el caso de haberse modificado alguna señal se debería repetir todo el camino entre el State inicial de este ciclo y el final con los valores actuales de las señales.

Para ayudar a entender cómo cambian las señales y entender mejor esto primero hay que recordar que hay 2 tipos de señales según su sincronismo:

- **Síncronas:** cambian al finalizar un ciclo de reloj.
- **Asíncronas:** cambian durante el ciclo del reloj.

Las señales síncronas no causan prácticamente ninguna complicación: cuando pasamos de un ciclo de reloj a otro (del bloque State con nombre “idle” al bloque con nombre “Loop” por ejemplo) actualizamos el valor de las señales indicadas en el bloque SyncOps. Por lo tanto estas señales no se modifican durante el ciclo de reloj.

El problema aparece con las señales Asíncronas. Estas señales cambian durante el ciclo de reloj cada vez que se encuentra una expresión que la modifique (como en el bloque AsyncOps del estado “idle” con texto “ready <= 1”).

En el diagrama de esta figura, centrándonos en el estado “idle” y suponiendo que la señal GO es siempre igual a 0 comprobamos que aunque la señal ready cambia durante el ciclo de reloj, salvo en el primer ciclo en el cual ready = 0 y por lo tanto la señal ready cambia de valor en el primer ciclo de simulación, en el resto siempre va a ser 1 y no crea ningún problema.

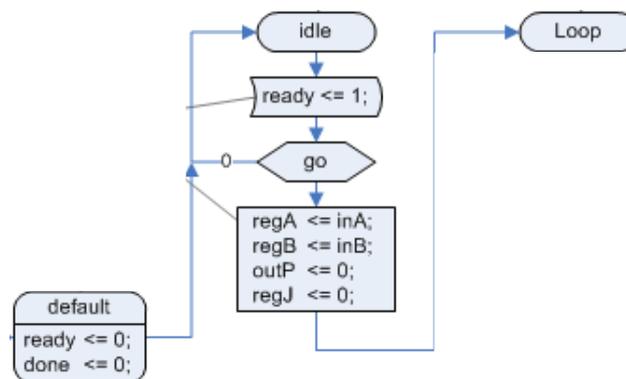


Figura 24: Diagrama explicativo del cambio de ciclo de reloj

Sin embargo, si en el bloque AsyncOps, en vez de tener el texto:

“ready <= 1”

apareciera:

“ready <= ready+1”

al pasar de un estado “idle” al mismo, la variable ready aumentaría, modificándose siempre, incumpliendo la condición necesaria de que **para pasar de un ciclo de reloj al siguiente hay que comprobar que no varía el valor de ninguna señal** y creando así un bucle infinito.

Esto sería un problema de diseño, y debe ser detectado para modificarlo y evitar un mal funcionamiento del circuito. Por ejemplo, si en vez de ser un box AsyncOps fuera AsyncOps, como la señal sería síncrona y solo se modificaría al finalizar un ciclo de reloj, esta no se ve modificada durante el ciclo de reloj y desaparece el problema (que es la situación que aparece en el estado “Loop”).

Por esto, para que el simulador cumpla con la condición necesaria, al llegar a un estado final hay que comprobar siempre si durante ese ciclo se ha modificado alguna señal para poder pasar o no al siguiente ciclo de reloj.

- **Diseño de la simulación del circuito (test bench):**

En primer lugar, un test bench por definición es un diseño que tiene como finalidad verificar el correcto funcionamiento de un módulo. Para describir correctamente un test bench, el diseñador debe tener siempre presente las especificaciones de diseño, quedando reflejadas las funciones del diseño a verificar.

Un test bench se compone de 3 elementos:

- **Módulo DUT:** diseño a verificar.
- **Modulo test:** modulo encargado de activar las señales de entrada al DUT y analizar las salidas que produce.
- **Módulo tb:** modulo que incluye los anteriores, caracterizado por no tener ni entradas ni salidas, y que corresponde a una declaración estructural del conexionado de los módulos.

Los test bench necesitan tener una generación de estímulos, realizada con en el proceso Initial (inicial) y always (siempre). Dependiendo de la naturaleza de las señales a generar, estas pueden crearse síncronamente o de forma asíncrona.

Todo esto mencionado anteriormente, en los diagramas ASM++ emplea una estructuración como la que se ha indicado a lo largo de este capítulo, empleando las cajas mostradas y con las diferentes secciones indicadas. Para poder visualizar como se realiza un test bench empleando los diagramas ASM++ se emplea de apoyo la siguiente figura:

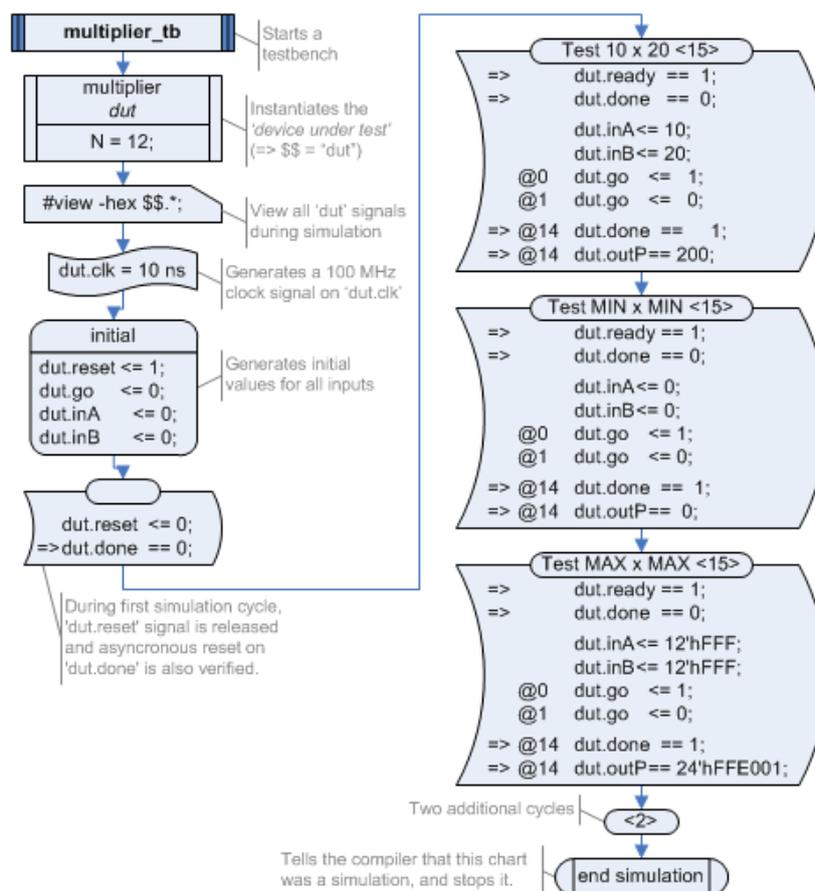


Figura 25: Diseño del Test Bench de un circuito digital

Como se ha mencionado, un test bench está formado por 3 partes diferentes: módulo DUT, modulo test y modulo tb. En el diagrama ASM++ aparecen estas 3 partes siendo las siguientes:

- **Módulo DUT:** definido mediante el box Instance, hace referencia al diseño definido en su cabecera con el nombre “multiplier”.
- **Modulo test:** la inicialización de las señales se realiza con el box Initial y el box ThreadSync.



- **Modulo tb:** hace referencia al diseño completo mostrado en la figura, empezando con un bloque Header y finalizando con un bloque Metastate.

Una vez comprobado a que hace referencia cada apartado del diagrama ASM++ de acuerdo a la teoría básica de un test bench, el siguiente paso es comprender las diferentes partes del módulo tb apoyando la explicación en la figura 25.

Para ello, se va a dividir la explicación de acuerdo a las diferentes secciones presentes:

- **External section:** esta sección, formada normalmente por una llamada a otro diseño o por comentarios dentro de un bloque Code, no aparece en este diseño, pues como ya se ha mencionado anteriormente en el capítulo no es imprescindible para la correcta definición de un diagrama ASM++.
- **Declarative Section:** esta sección tiene como punto de partida el box “Header” y como fin el box “Initial” puesto que el siguiente a este es un box state. En primer lugar, en esta sección se indica el nombre del módulo tb con un box “Header”.

Después, aparece un box “Instance”, representando el módulo DUT, que tiene como finalidad indicar el diseño del circuito lógico al que hace referencia el test bench. En este caso, hace referencia al diseño que tiene en su cabecera el nombre “multiplier”.

A continuación aparece un box Code, cuya finalidad es sustituir la variable dut por otra variable denominada \$\$, para definir todas las señales a las que se hace referencia en el diseño del test bench, y enlazarlas con las señales del diseño. En esa figura la palabra dut no aparece cambiada por la variable \$\$ debido a que el simulador dentro de sus limitaciones no tiene incluido el reconocimiento de parámetros y variables.

Para finalizar esta sección, aparece el modulo test, formado por los box ThreadSync e Initial, cuya finalidad es establecer el valor de las señales indicadas en su interior antes de entrar en la sección algorítmica.

- **Algorithmic Section:** esta sección tiene como punto de partida siempre el primer bloque State presente, y como punto final un box “MetaState” con el texto “End Simulation”.

Como se ha mencionado anteriormente, para simplificar los diagramas, existe un tipo de cajas denominadas **StateAsyncOps** y **StateSyncOps**, que se creó como la unión de un bloque State con un AsyncOps y SyncOps respectivamente.

En el diagrama de la figura, estos bloques son los encargados de indicar que se realiza en cada ciclo de reloj. En su parte superior (la que correspondería al State), aparece un texto con el siguiente formato:

Test NombreBloque <N°Ciclos>

Con este texto se indica en primer lugar el nombre del bloque, y en segundo lugar, el número de ciclos de reloj que se ejecutara dicho box. En el caso de encontrarse vacío (como ocurre en la figura 25 con el primer state, indica que solo se ejecutara durante 1 ciclo de reloj.

En la parte inferior de estas cajas aparece un texto, el cual puede indicar 4 acciones diferentes indicadas cada una con un símbolo distinto en el inicio de cada línea de texto:

- **Verificación en el primer ciclo de reloj ( “=>” ):**

Cuando aparece este símbolo antes de una expresión, se indica que durante el siguiente ciclo de reloj se tiene que realizar la verificación marcada por la expresión de esa línea.

- **Verificación en el ciclo indicado ( “=> @NºCiclo”):**

Cuando aparece este símbolo antes de una expresión se indica que en el ciclo indicado después del símbolo @ se tiene que realizar la verificación indicada por la expresión.

- **Actuar en el ciclo indicado (“@NºCiclo”):**

Cuando aparece solo este símbolo antes de una expresión, se indica que en el ciclo indicado después del símbolo @ se debe ejecutar la expresión indicada

- **Actuar en el primer ciclo:**

En el caso de aparecer una expresión en lenguaje HDL sin ningún símbolo previo se tendrá que ejecutar la expresión durante el primer ciclo de reloj de ese box.

Tanto las verificaciones como la ejecución de las expresiones explicadas se realizará en diferente momento dependiendo de si el box es StateAsyncOps (se realizará durante el ciclo de reloj) o si es StateSyncOps (al finalizar el ciclo de reloj).

Para finalizar la explicación del diagrama de la figura y por lo tanto el test bench, aparece un box state, que no ejecutara expresiones, simplemente tiene como objetivo añadir 2 ciclos de reloj antes de la finalización del test bench, y a continuación el box Metastate con el texto “End Simulation” que indica que se ha finalizado el test bench.



### 3.1.3. Estructura de los archivos .vdo compilados para simulación:

Un archivo con extensión .vdo es un documento obtenido tras la compilación de un diagrama ASM++ con el compilador de diagramas ASM. Este archivo es el encargado de estructurar, de acuerdo a un estándar explicado a continuación, todos los diagramas ASM++. En definitiva se podría decir que es una traducción del diagrama ASM++ a un lenguaje con una estructura estandarizada.

Todos los archivos .vdo tienen una estructura similar, formada por módulos con diferentes atributos, independientemente de las cajas que incluya el diseño (para todos los ejemplos empleados a continuación se emplean capturas del archivo vdo correspondiente al diseño de la figura 28 empleado para comprobar el funcionamiento del simulador):

- **Comentarios:** en primer lugar, pueden aparecer comentarios relativos al programa. Todos estos comentarios siempre tienen como inicio dos barras inclinadas "//":

```
//  
// VDO code generated by ASM++ Compiler 3.00.3 from  
// 'Test_Multiplier.vdx'  
//  
// File name: C:/Almacen/D.Sanz/Clase/Universidad/TFG/
```

- **Páginas (pages):** estos módulos se encargan de indicar a qué página pertenece cada diseño:

```
Pages {  
    Id = 0; Name = "multiplier & multiplier_tb";  
}
```

Estos módulos siempre tienen la misma estructura que la mostrada en la figura anterior:

- Palabra "Pages" seguida de llave que indica el inicio del módulo "{"
- El id y el nombre de los diseños
- Llave que indica el cierre del módulo "}"



- **Cajas (Box):** estos módulos se encargan de describir todas las cajas que aparecen tanto en el diseño como en el test bench de un circuito digital. Su estructura prácticamente igual siempre:
  - Palabra “**Box**” seguida de llave que indica el inicio del módulo “{“
  - El **Id** seguido de un “=” y el numero identificador
  - El tipo del box indicado con la palabra **Type**, seguido de un “=” y el tipo de box.
  - El **texto** de cada box que puede ser:
    - Text: texto normal
    - TextUp: texto situado en la parte superior de un bloque con texto superior e inferior como un box Instance por ejemplo.
    - TextDown: texto situado en la parte inferior de un bloque con texto inferior y superior.
  - Palabra **Next**, que indica el Id del bloque que le precede. Como puede darse el caso de que el box tenga diferentes bloques a continuación como un box Decision para diferenciarlos aparece de la siguiente manera:
    - Next: id del bloque siguiente normal.
    - Next0: id del bloque siguiente si sigue un camino.
    - Next1: id del bloque siguiente si sigue otro camino.
  - Palabra **Comment**, que indica que existe un comentario en ese bloque.

Para verlo mejor se adjuntan a continuación 2 ejemplos en los que se muestran la caja en el diagrama ASM++ y su descripción en el archivo .vdo:

- Ejemplo box StateAsyncOps:



Figura 26: Ejemplo box StateAsyncOps

#### Box

```
Id = 41;  
Type = "StateAsyncOps";  
TextUp = "";  
TextDown = "dut.reset <= 0;%CR%=> dut.done == 0;";  
Next = 90;
```

- Ejemplo box Decision:

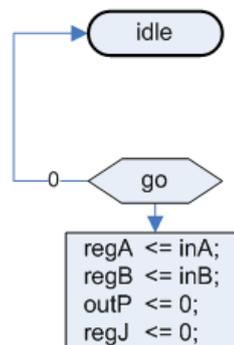


Figura 27: Ejemplo box Decision

#### Box

```
Id = 141;  
Type = "Decision";  
Text = "go";  
Next0 = 156;  
Next1 = 145;
```



### 3.2. Especificaciones del proyecto:

Como se ha explicado a lo largo de este Capítulo, debido a la necesidad de poder simular informáticamente el comportamiento de un circuito lógico diseñado mediante diagramas ASM++, para poder encontrar posibles fallos en el diseño, sin tener que emplear aplicaciones externas a las empleadas para la compilación de los diagramas, surgió la idea de crear un simulador propio para comprobar el comportamiento a lo largo de los ciclos de simulación dentro del propio compilador.

El proyecto de este simulador está marcado por una serie de especificaciones que se indicarán a continuación:

- El objetivo del proyecto es poder disponer de una primera versión de un simulador de circuitos digitales descritos con diagramas ASM++. El simulador no debe poder simular absolutamente todos los diseños posibles que se pueden realizar, sino que debe ser una versión inicial, formando columna vertebral del programa, al cual se podrán ir añadiendo en futuras versiones nuevas funcionalidades. En el apartado 3.4.1 se muestran las cajas que deben poder ser simulables, indicando que tipos de diseños puede simular esta versión inicial del simulador y cuáles no.
- El simulador debe poder ser fácilmente integrable en el compilador de diagramas ASM++, debido a esto su programación tiene que ser desarrollada en lenguaje C++, empleando las librerías Qt y partiendo de un tipo de entrada predefinida (descrita a continuación) para dar una salida específica (descrita más adelante). La integración en el compilador no está incluida en las especificaciones del proyecto.
- El simulador tiene como punto de partida un archivo con extensión .vdo, en el cual aparecerá descrito en lenguaje Verilog el diseño de un circuito digital, con sus diferentes cajas y páginas, así como el diseño de un test bench (banco de pruebas) que indica como debe ser la simulación, marcando el valor de ciertas señales en los diferentes ciclos de simulación, e indicando el final de la simulación y una serie de verificaciones finales.

- Para realizar una simulación válida, el simulador debe poder simular correctamente diseños con un nivel complejidad como el mostrado en la figura 28, en el cual aparece tanto el diseño de la lógica del programa como el test bench.

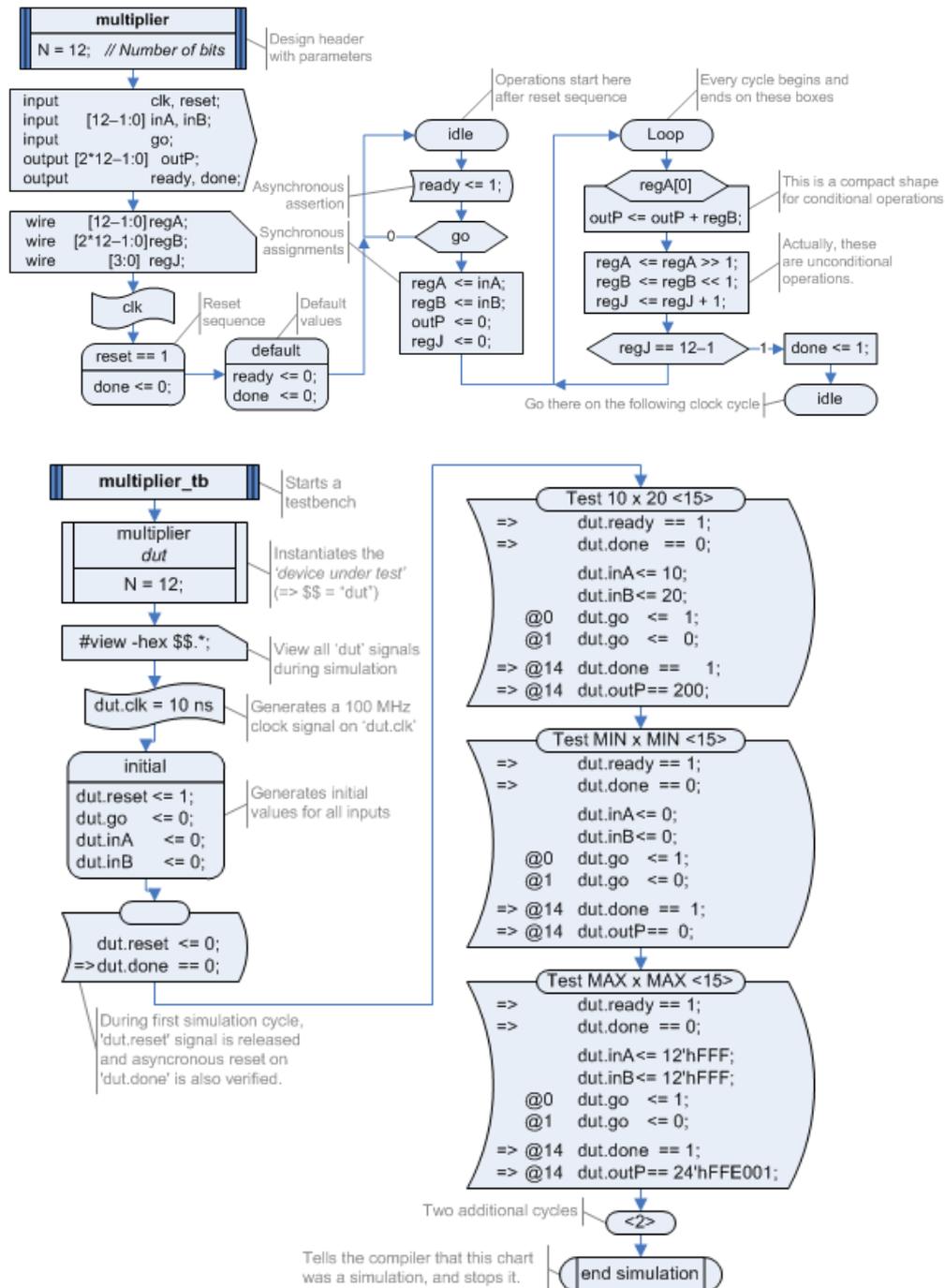


Figura 28: Ejemplo de circuito digital diseñado con diagramas ASM++. Diseño y Test Bench



El simulador tiene que ser capaz de interpretar correctamente todo el circuito, siguiendo una estructura similar a la mostrada a continuación:

1. Leer archivo con extensión .vdo, generando una serie de variables internas a partir de todo lo descrito en el archivo de entrada.
  2. Ejecutar el simulador a partir de toda la información recogida en la lectura del archivo de la siguiente forma:
    - a. Búsqueda de las cajas cabecera (Header) que indican el principio de un bloque (diseño o test bench).
    - b. Búsqueda de las instancias dut dentro del test bench para diferenciar entre un bloque de diseño o test bench. Esto también puede realizarse buscando un box MetaState con la descripción “End Simulation” en su interior.
    - c. Buscar las diferentes señales I/O del circuito, detectando y definiendo el tipo de las señales: diferenciando entre señales Input, Output, e indicando también si son síncronas o asíncronas).
    - d. Búsqueda de las señales internas de los módulos: Wire o Inout, indicando también si son síncronas o asíncronas.
    - e. Capacidad de ejecución del texto indicado dentro de todas las cajas existentes en el archivo de entrada mediante un analizador de expresiones.
    - f. También debía ser capaz de ejecutar ciclo a ciclo la simulación total del circuito digital detectando los posibles errores encontrados al comprobar una serie de verificaciones descritas en el test bench.
- El simulador debe ser capaz de interpretar circuitos descritos únicamente en lenguaje Verilog, por lo tanto en el desarrollo de su programación debe incluir un analizador de expresiones en lenguaje Verilog siguiendo la estructura indicada en la figura 10.
  - La salida del simulador debe ser un archivo .Log o .txt, el cual debe incluir al menos un listado con todos los errores y avisos (warnings) detectados durante la simulación, que facilitará la inclusión del propio simulador en el compilador, el cual mostrará únicamente si aparece o no errores en el archivo.



El simulador no tiene como objetivo solucionar al usuario que ejecute el programa los posibles errores encontrados. Sin embargo, para facilitar su labor al usuario del simulador, este incluye la funcionalidad de mostrar al usuario en los errores y avisos de la salida todos aquellos errores que vaya encontrando, incluidas las posibles cajas encontradas no incluidas en la interpretación. En el caso de encontrar algún box no interpretado durante la lectura del archivo de entrada, muestra por pantalla durante el proceso de lectura la línea del archivo en la cual se encuentra el box no interpretable.

- Para poder visualizar el comportamiento del simulador, y permitir al usuario disponer de únicamente el simulador sin necesidad de emplear el compilador de diagramas ASM++ al completo, se debe desarrollar una aplicación gráfica con interfaz de usuario, la cual tiene que permitir al usuario abrir y editar archivos con extensión .vdo en un layout, permitir la ejecución del simulador al usuario cuando este lo desease, y mostrar en otros 2 layouts la salida y el proceso de la simulación.

Para ello se debe emplear el programa Microsoft Visual Studio 2010 express, así como las librerías Qt 4.8.5.

Una vez cumplidas todas las especificaciones iniciales se puede dar por finalizada esta primera versión del simulador y por lo tanto este proyecto. En los apartados siguientes se muestran más detalladamente alguna de las especificaciones indicadas anteriormente.

### 3.2.1. Expresiones en lenguaje Verilog:

Como ya se ha mostrado con anterioridad en el apartado 2.2., y más concretamente en la figura 10, las expresiones en el lenguaje Verilog están formadas por una estructura siempre igual.

Como se explicara en el siguiente apartado con más detalle, según especificaciones se ha realizado un analizador de expresiones, encargado de realizar correctamente todo lo indicado en la imagen anterior. Sin embargo, esta primera versión del simulador no tiene dentro de sus especificaciones la interpretación de absolutamente todas las expresiones existentes. A continuación se muestra un listado con los tipos de expresiones no incluidos en las especificaciones:

- **Signal con [ : ]:** no incluido en las especificaciones
- **Parameter:** no incluido en las especificaciones
- **Group:** no incluido en las especificaciones

### 3.2.2. Tipos de cajas simulables de los diagramas ASM++:

Como se ha descrito anteriormente, y representado en la figura 29, todos los diagramas ASM++ están formados por una estructura similar, que incluye una sección externa (External section), una sección declarativa (Declarative section) y una sección algorítmica (Algorithmic section). Cada zona incluye una serie de cajas diferentes y realizan una función u otra en el circuito.

En esta primera versión del simulador no se ha incluido la interpretación de todas las posibles cajas existentes en los diagramas ASM++. A continuación se muestra un listado de las diferentes cajas que pueden aparecer (indicados en la figura 29, mostrada de nuevo a continuación) indicando si está incluida o no su interpretación:

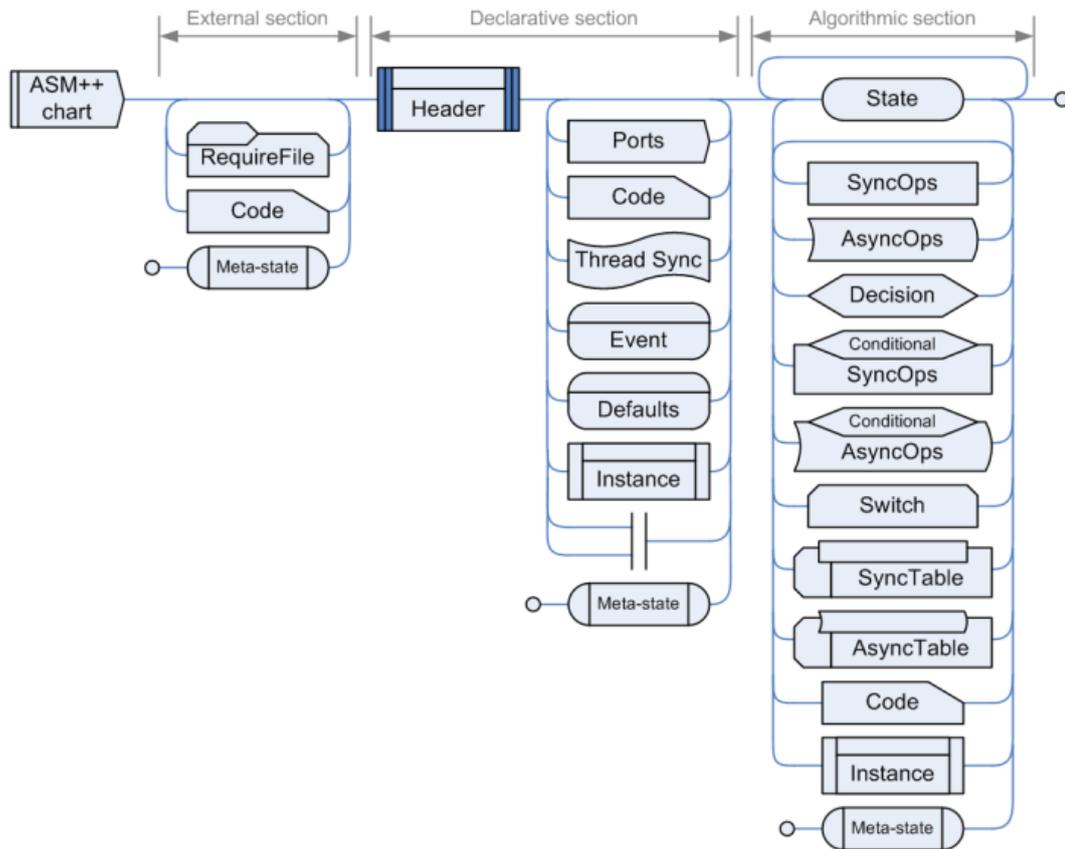


Figura29: Estructura de los diagramas ASM++

- **External Section:** esta primera versión del simulador no tiene que incluir la interpretación específica de ninguna caja incluida antes de una cabecera (Header).
  - o **Require file:** fuera de especificaciones
  - o **Code:** fuera de especificaciones
  - o **MetaState:** fuera de especificaciones
  
- **Declarative Section:** esta sección como se ha mencionado anteriormente, se encarga de definir toda la información que se empleará posteriormente en la sección algorítmica.
  - o **Header:** incluido en las especificaciones
  - o **Ports:** incluido en las especificaciones
  - o **Code:** incluido en las especificaciones
  - o **ThreadSync:** incluido en las especificaciones
  - o **Event e Initial:** incluido en las especificaciones
  - o **Default:** incluido en las especificaciones
  - o **Instance:** incluido en las especificaciones
  - o **||:** fuera de especificaciones
  - o **MetaState:** fuera de especificaciones



- **Algorithmic section:** en esta sección se incluyen todas las cajas incluidos entre 2 cajas state. Se encargan de realizar las operaciones indicadas en cada ciclo.
  - **State:** incluido en las especificaciones
  - **SyncOps:** incluido en las especificaciones
  - **AsyncOps:** incluido en las especificaciones
  - **Decision:** incluido en las especificaciones
  - **CondSyncOps:** incluido en las especificaciones
  - **CondAsyncOps:** incluido en las especificaciones
  - **Switch:** fuera de especificaciones
  - **SyncTable:** fuera de especificaciones
  - **AsyncTable:** fuera de especificaciones
  - **Code:** incluido en las especificaciones
  - **Instance:** incluido en las especificaciones
  - **Metastate:** incluido en las especificaciones

Para facilitar aún más el trabajo a los futuros desarrolladores que quieran incluir la interpretación de cajas no incluidas, el simulador es capaz de mostrar durante el desarrollo de la simulación si la interpretación de un tipo de box no está incluida al ser detectado este durante la simulación.

### 3.2.3. Módulos de los archivos .vdo:

Como se ha mencionado anteriormente, los archivos .vdo están formados por módulos. Existen 2 tipos de módulos presentes: Pages y Box. Esta primera versión del simulador dentro de sus especificaciones no incluye la detección de todas las posibles opciones que pueden aparecer en los módulos de los archivos .vdo. A continuación se muestran que elementos quedan fuera de estas especificaciones muestra el simulador con respecto a estos archivos:

- **Pages:** los módulos que indican las páginas de los diseños no tienen por qué ser implementados. Como sí que aparecerá una zona en el archivo de entrada con este módulo, el simulador debe detectar el tipo de modulo, su id y su nombre, almacenándolo en una clase, pero sin emplear esta información en ningún momento del resto de la simulación debido a que esta primera versión solo debe aceptar archivos y diseños con 1 página.
- **Box:** dentro de los módulos Box aparecen diferentes apartados que indican el Id del box, su tipo... Puesto que no están incluidos todos los tipos de cajas existentes, solo se han incluido en esta versión las



características que aparecen en el archivo vdo empleado para comprobar el funcionamiento del simulador. Las características implementadas son las siguientes:

- **Id:** incluido en las especificaciones
- **Type:** incluido en las especificaciones
- **Text:** incluido en las especificaciones
- **TextUp:** incluido en las especificaciones
- **TextDown:** incluido en las especificaciones
- **Next:** incluido en las especificaciones
- **Next0:** incluido en las especificaciones
- **Next1:** incluido en las especificaciones

### 3.3. Desarrollo de la programación del simulador:

En este apartado se describe por orden cronológico el desarrollo global de la programación del simulador, describiendo detalladamente las diferentes acciones realizadas para su creación, incluyendo imágenes y las partes de código que se creen necesarias para apoyar la descripción de los diferentes apartados del desarrollo.

Al igual que ocurrió para la creación del simulador al completo, el desarrollo de la programación del simulador se separará en 3 apartados que al finalizar quedan interrelacionados formando el conjunto global del proyecto. Estos apartados son: desarrollo del interfaz de usuario, desarrollo del analizador de expresiones, desarrollo del código del simulador.

#### 3.3.1. Desarrollo del interfaz de usuario:

En primer lugar, para poder disponer de un entorno gráfico de usuario, se desarrolla una aplicación que permita seleccionar un archivo con extensión .vdo, muestre el proceso de simulación y el resultado final de la misma.

Para lograrlo, se emplean las bibliotecas Qt, las cuales, aparte de aportar multitud de funciones para simplificar el proceso de creación, incluyen modelos que sirven de base para crear el entorno gráfico final que buscamos.

La aplicación está formada por un entorno gráfico de usuario básico ejecutable en un sistema operativo de Windows, con un aspecto esquemático como el mostrado a continuación:



Figura 30: Esquema del interfaz gráfico de la aplicación

Como se muestra en la imagen, la aplicación cuenta con 3 zonas diferenciadas:

- **Barra menú principal:** esta barra, formada por 4 menús denominados “File”, “Edit”, “Simulator”, “Help”, se encuentra situada en la parte superior de la ventana, y se encarga de contener todos los menús disponibles en la aplicación los cuales contienen diferentes acciones disponibles.
- **Barra de herramientas:** esta barra, situado justo debajo de la barra de menú principal, contiene una serie de iconos o botones que sirven de acceso rápido a una serie de funciones incluidas en la barra de menú principal.
- **Ventanas de simulación:** esta zona, situada debajo de la barra de herramientas, está formada por 3 ventanas las cuales permitirán al usuario ver y editar el archivo de simulación deseado, comprobar el proceso de la simulación y obtener el archivo de salida de simulación.

Antes de entrar más en profundidad en cómo es la realización de cada apartado, se muestra en la siguiente imagen como es el aspecto final de la ventana gráfica una vez finalizada:

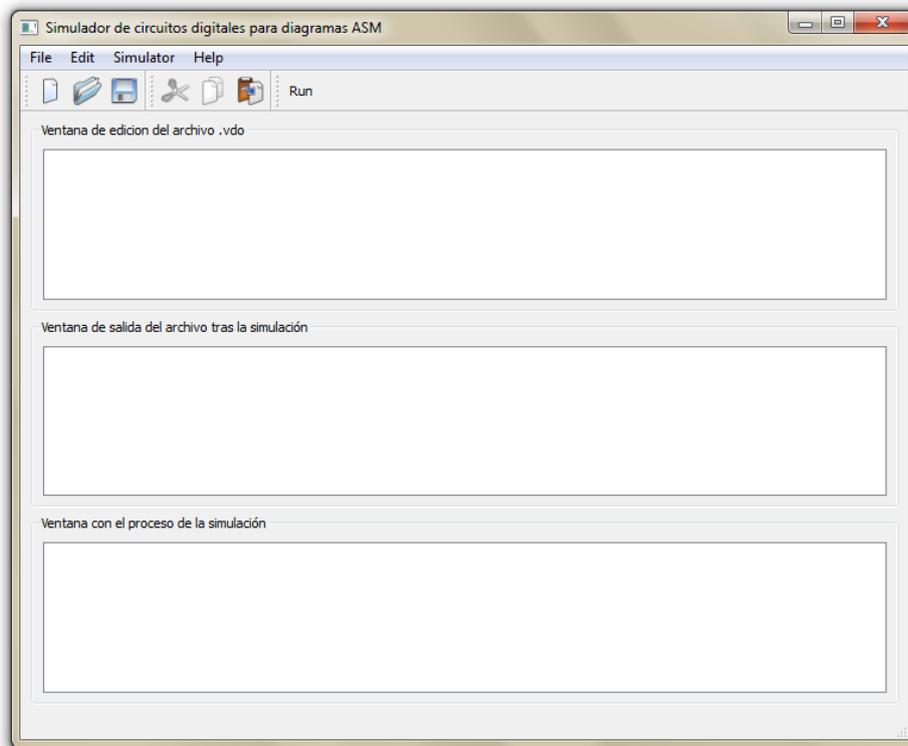


Figura 31: Entorno gráfico de la aplicación

El aspecto final del entorno gráfico es el mostrado en la imagen anterior, donde se puede observar cada una de las partes indicadas en el esquema de la página anterior.

Para poder obtener una ventana gráfica como la anterior, obviando los diferentes menús y cuadros blancos de texto, hacen falta en primer lugar 2 bibliotecas, que incluyen las funciones necesarias para su realización:

- **Biblioteca con la definición de las clases** incluidas en QtCore y QtGui:

```
#include <QApplication>
```

- **Bibliotecas** que incluyen todo lo necesario para **ejecutar una ventana en Windows**:

```
#include <QMainWindow>
```

```
#include <QtGui>
```

Una vez incluidas las bibliotecas necesarias, se procede a la creación del frame que servirá de plantilla para incluir las diferentes funcionalidades, barras y ventanas de texto deseadas. Para ello en primer lugar se crea la clase que maneja el control de las aplicaciones GUI y se accede a la función principal de la ventana de usuario:

```
QApplication app(argc, argv);  
  
MainWindow::MainWindow() {}
```

En la función principal se crean las diferentes partes de la ventana gráfica como se muestra a continuación:

- **Creación de las variables que hacen referencia a las ventanas de texto** en las que se mostrara el archivo de entrada, el de salida y el proceso, indicando los permisos de cada uno de ellos:

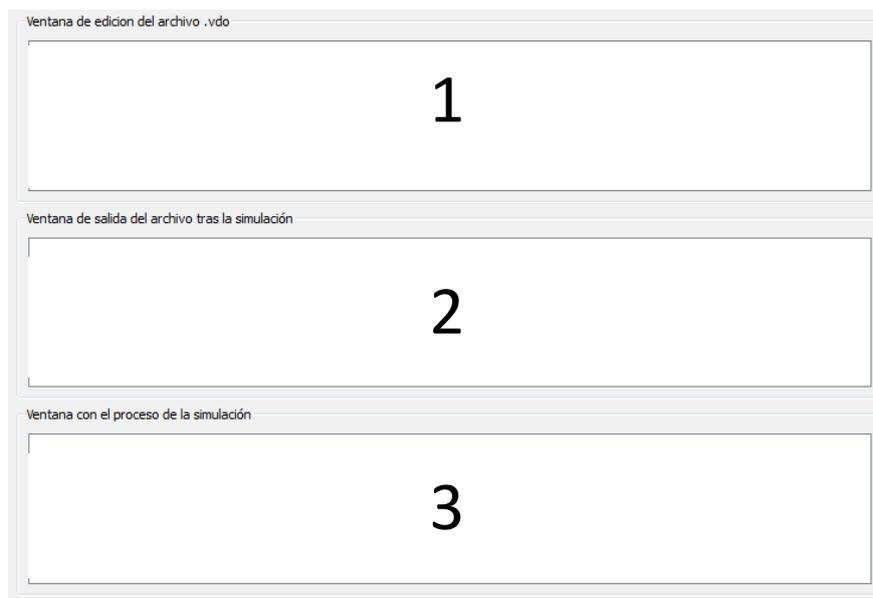


Figura 32: Ventanas de texto

```
1- textEdit = new QPlainTextEdit;  
2- textOut = new QListWidget;  
3- textProcess = new QListWidget;
```

- **Creación del layout de la ventana principal**, situando cada una de las ventanas creadas anteriormente en el orden deseado. Para lograrlo se indica en la función `addWidget` con un número la posición, y con la función `setLayout` se crea:

```
QGridLayout *grid = new QGridLayout;  
grid->addWidget(CreateEditGroup(),1,0)  
grid->addWidget(CreateOutGroup(),2,0);  
grid->addWidget(CreateProcessGroup(),3,0);
```

```
setLayout(grid);
```

- **Colocación e inserción del layout con las 3 ventanas de texto** en el centro de la ventana principal. Para ello se crea un nuevo `Widget`, en el cual se guarda el `GridLayout` creado anteriormente, y con la función `setCentralWidget` se sitúa en el centro de la pantalla:

```
QWidget *widget = new QWidget();  
widget->setLayout(grid);  
setCentralWidget(widget);
```

- **Creación de la función que crea las acciones del programa**. Para que al pulsar una opción del menú o de un icono de la barra de herramientas se ejecute la acción deseada, primero hay que crear las acciones deseadas. A continuación se incluyen las líneas creadas para la acción de crear un archivo nuevo, creando la variable que realiza la acción asociándola a una imagen, estableciendo un nombre para dicha acción, fijando un mensaje para la barra de estado y conectándola con la función `newFile` que se encarga de abrir un nuevo archivo:

```
newAct = new QAction(QIcon(":/images/new.png"),  
tr("&New"), this);
```

```
newAct->setShortcuts(QKeySequence::New);
```

```
newAct->setStatusTip(tr("Crear un archivo nuevo"));
```

```
connect(newAct, SIGNAL(triggered()), this,  
SLOT(newFile()));
```

```
void MainWindow::newFile() {}
```

Al igual que se hace con la acción de nuevo fichero, para cada acción existente aparecen líneas de código similares, guardando una variable como referencia a una acción, y conectándola con una función.

## - Creación de los menús:



Figura 33: Barra de menús

La barra de menús, mostrada en la figura superior, está formada por 4 listas de acciones, incluyendo todas las posibles acciones realizables por el simulador. Para la creación de esta barra de menú se crea la función `createMenus()` que se encarga de ir añadiendo las diferentes listas y acciones deseadas. A continuación se muestra tanto en imagen como en código los diferentes menús (el código solo aparece para el menú "File", puesto que en los siguientes menús es similar):

- **File:** menú archivo, creado con la función `addMenu`, en el que se incluyen con la función `addAction` las diferentes acciones deseadas (las cuales se han creado con anterioridad). Con la función `addSeparator` se incluye una línea que separa unas acciones de otras. En este menú se incluye una lista que muestra los archivos abiertos recientemente, para facilitar al usuario su búsqueda.

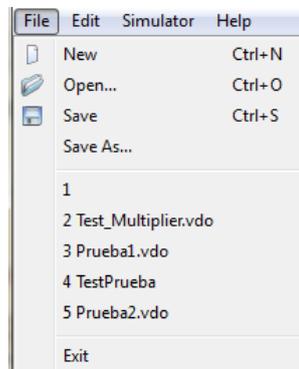


Figura 34: Menu File

```
fileMenu = menuBar()->addMenu(tr("&File"));
fileMenu->addAction(newAct);
fileMenu->addAction(openAct);
fileMenu->addAction(saveAct);
fileMenu->addAction(saveAsAct);
separatorAct = fileMenu->addSeparator();
for (int i = 0; i < MaxRecentFiles; ++i)
    fileMenu->addAction(recentFileActs[i]);
fileMenu->addSeparator();
fileMenu->addAction(exitAct);
updateRecentFileActions();
```

- **Edit:** en este menú se incluyen 3 diferentes funcionalidades disponibles a realizar sobre el texto, como son copiar, cortar y pegar.

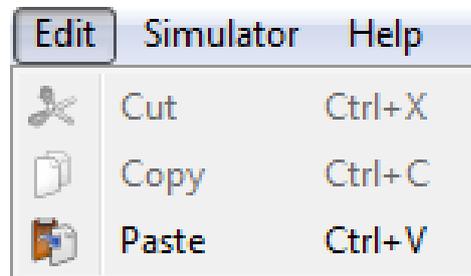


Figura 35: Menú Edit

- **Simulator:** en este menú se incluye la acción de ejecutar el archivo seleccionado para obtener el resultado de la simulación:

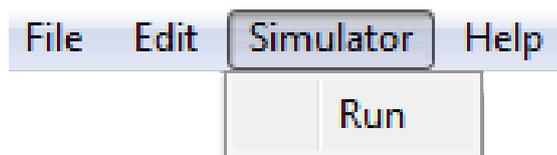


Figura 36: Menú Simulator

Esta acción enlaza con la función Run, que se encuentra descrita a continuación:

```
void MainWindow::run()
{
    if (curFile.isEmpty())
        QMessageBox runBox;
        runBox.setText("No hay ningun archivo");
        runBox.exec();
    else
        QMessageBox runBox;
        runBox.setText("Simulando el archivo.");
        runBox.exec();

    return simulacion(curFile);
}
```

Como se observa, esta función detecta si hay archivo o no seleccionado con la variable `curfile`. De no haberlo, lanza una ventana con el mensaje “No hay ningún archivo”. En caso de detectar el archivo, llama a la función pasándole el archivo “simulación(`curfile`)”, y a partir de ese momento se inicia la simulación.

- **Help:** este menú simplemente muestra información sobre el simulador y sobre las librerías Qt:

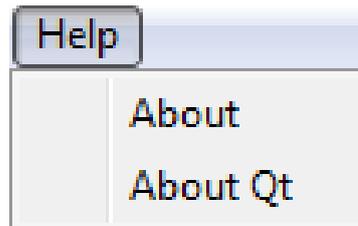


Figura 37: Menú Help

- **Creación de la barra de herramientas:** para que todas las acciones existentes sean más fácilmente accesibles, se incluye una barra de herramientas con imágenes (salvo la acción “Run”).

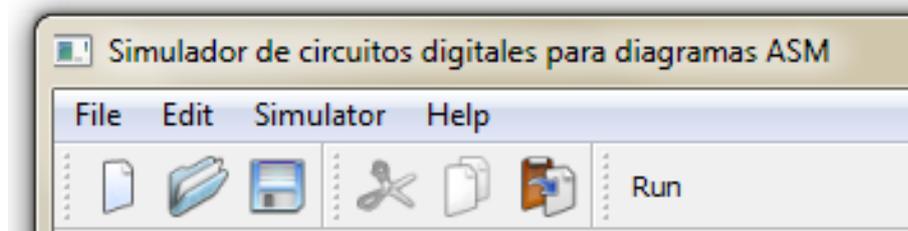


Figura 38: Barra de herramientas

Como se observa en la imagen, las funciones son las mismas que estaban incluidas en los diferentes menús. Para crear esta barra de herramientas se crea la función `createToolBars()` con el siguiente código:

```
void MainWindow::createToolBars()

//Barra de herramientas archivo
fileToolBar = addToolBar(tr("File"));
fileToolBar->addAction(newAct);           //Nuevo
fileToolBar->addAction(openAct);         //Abrir
fileToolBar->addAction(saveAct);         //Guardar

//Barra de herramientas editar
editToolBar = addToolBar(tr("Edit"));
editToolBar->addAction(cutAct);           //Cortar
editToolBar->addAction(copyAct);         //Copiar
editToolBar->addAction(pasteAct);        //Pegar

//Barra de herramientas simulador
editToolBar = addToolBar(tr("Simulator"));
editToolBar->addAction(runAct);           //Run
```

- **Creación de la barra de estado:** también se crea en la parte inferior una barra de estado, cuya finalidad es mostrar en cada momento de la ejecución del simulador que acción se está realizando. Por ejemplo, al ir a pulsar sobre el botón Run, aparece un mensaje en la parte inferior indicando que se va a ejecutar la simulación:

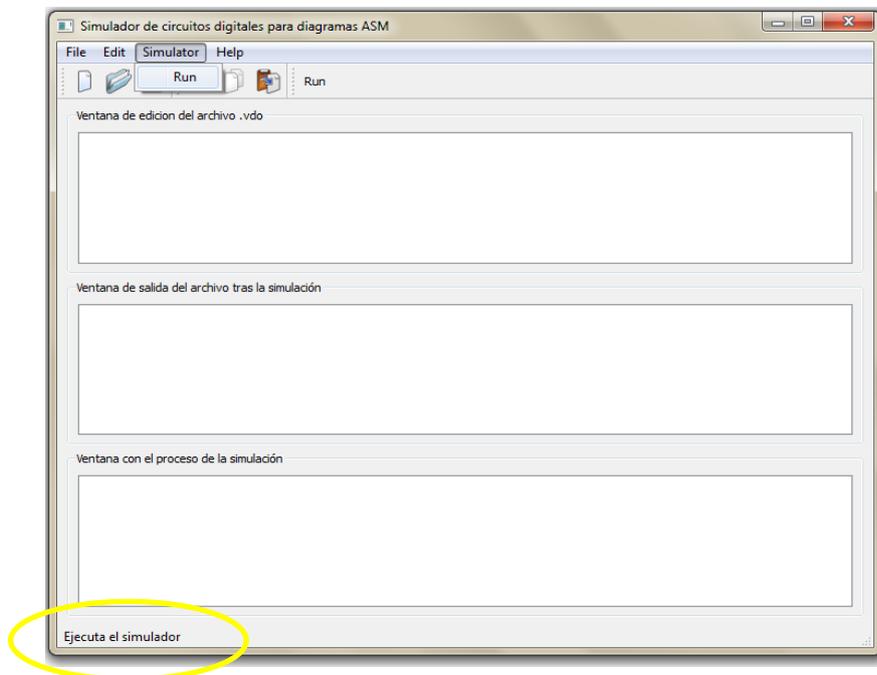


Figura 39: Barra de estado

A parte de todo esto que se ha descrito, también se añaden otras funcionalidades como funciones que leen los ajustes de la aplicación para mantenerlos como se encontraba anteriormente al abrir la aplicación de nuevo, o funciones que se encargan de comprobar si el texto ha sido modificado para avisar al usuario. Con la función `setWindowTitle` se establece el título de la ventana.

Con todo esto queda terminado el diseño de la ventana de interfaz de usuario.

### 3.3.2. Desarrollo del analizador de expresiones:

Una vez realizado el interfaz de usuario, el siguiente paso es crear un analizador de expresiones en lenguaje Verilog, capaz de interpretar expresiones descritas en este lenguaje para invocarlo desde el simulador cada vez que sea necesario analizar una expresión durante la simulación.

Para realizar este analizador de expresiones, se emplea el diagrama de la figura 40, que ya se ha mostrado anteriormente en este capítulo, y que sirve de apoyo para simplificar la estructura de las expresiones Verilog, así como de gran ayuda visual para comprender fácilmente la lógica que siguen las expresiones.

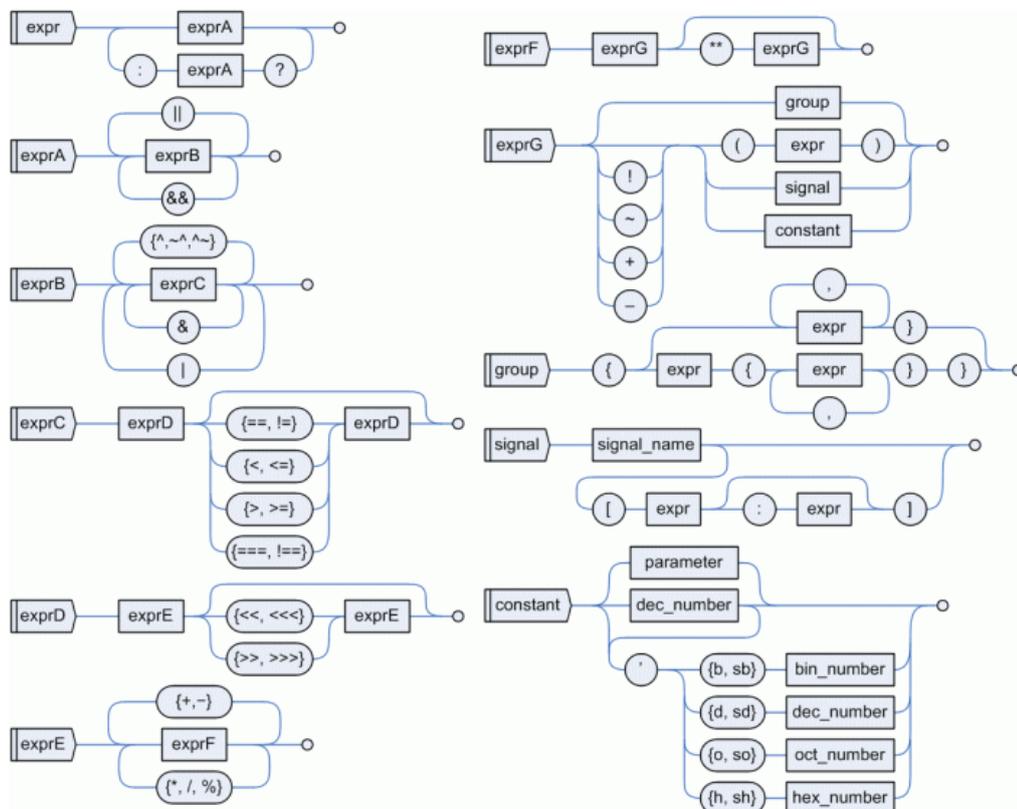


Figura 40: Expresiones en Lenguaje Verilog

Como se mencionaba en el apartado 3.1.1. Expresiones en lenguaje Verilog (página 34), la estructura de las expresiones en este lenguaje se puede desglosar como se muestra en la figura anterior. Cada tipo de expresión está formada por una operación de expresiones de un nivel con prioridad superior, que a su vez está formada por una operación de expresiones de prioridad superior, y así sucesivamente hasta llegar a las expresiones denominadas en la figura anterior ExprG, que están formadas por grupos de expresiones iniciales de menor prioridad, señales, expresiones iniciales de menor prioridad entre paréntesis o constantes.

La metodología seguida para diseñar el analizador de expresiones se basa prácticamente en la estructura marcada en la figura. Partiendo de una expresión en lenguaje verilog almacenada en un tipo de dato String, la cadena de caracteres va pasando por una serie de funciones que se van llamando unas a otras hasta obtener el resultado final de la expresión.

Para simplificar la explicación de cómo se ha realizado el analizador de expresiones en lenguaje verilog se va a dividir la explicación en diferentes apartados:

#### a) Lista enumerada de tokens:

En primer lugar, para poder realizar la detección de cada símbolo presente en las expresiones, se crea una biblioteca (archivo con extensión .h) denominada “shared.h” en la cual se realiza una lista enumerada (enum) que permite asignar a cada variable presente dentro de la lista un número determinado de acuerdo con la posición que ocupe en la lista, convirtiendo cada variable en una variable global del programa.

Para poder diferenciar estas variables de otras, se ha seguido la siguiente estructura:

TOKEN \_ NOMBRE-EXPLICATIVO-DEL-SIMBOLO

Gracias a estos tokens, a la hora de programar y de entender el código fuente del programa para una persona externa se simplifica todo, gracias a que el simple nombre de la variable permite entender prácticamente de golpe a que símbolo hace referencia. Por ejemplo, para identificar el símbolo de suma “+” se ha creado un token denominado “TOKEN\_MAS”, el cual dentro de la lista debido a su posición obtiene el número 29 (aunque esta información es irrelevante a la hora de programar).



La lista de tokens es la siguiente:

```
enum {  
  
    //Bloque de errores  
    TOKEN_ERROR, TOKEN_NULL, TOKEN_EOF,  
  
    //Bloque de simbolos sin operacion  
    TOKEN_RETORNOCARRO, TOKEN_PUNTOCOMA, TOKEN_COMMENT,  
  
    //Expr  
    TOKEN_QUESTION, TOKEN_DOSPUNTOS,  
  
    //ExprA y ExprB  
    TOKEN_OLOGIC, TOKEN_OR, TOKEN_YLOGIC, TOKEN_AND,  
    TOKEN_NXOR, TOKEN_XOR, TOKEN_NEGBIT,  
  
    //ExprC y ExprD  
    TOKEN_TRESIGUAL, TOKEN_DOSIGUAL, TOKEN_IGUAL,  
    TOKEN_NOTDOSIGUAL, TOKEN_NOTIGUAL, TOKEN_NEGLOG,  
    TOKEN_TRESIZQ, TOKEN_DOSIZQ, TOKEN_MENORIGUAL,  
    TOKEN_MENOR, TOKEN_TRESDER, TOKEN_DOSDER,  
    TOKEN_MAYORIGUAL, TOKEN_MAYOR,  
  
    //ExprE y ExprF  
    TOKEN_MAS, TOKEN_MENOS, TOKEN_EXPO, TOKEN_POTENCIA,  
    TOKEN_DIVISION, TOKEN_PORCENTAJE,  
  
    //ExprG y siguientes  
    TOKEN_CORCHOPEN, TOKEN_CORCHCLOSE, TOKEN_PARENOPEN,  
    TOKEN_PARENCLOSE, TOKEN_QPARENOPEN, TOKEN_QPARENCLOSE,  
  
    TOKEN_COMA, TOKEN_APOSTROFE, TOKEN_COMILLAS,  
    TOKEN_ARROBA, TOKEN_VERIFICACION, TOKEN_PUNTO,  
  
    TOKEN_NUM, TOKEN_WORD, TOKEN_INPUT, TOKEN_INOUT,  
    TOKEN_OUTPUT, TOKEN_WIRE,  
  
    TOKEN_BINARIO, TOKEN_DECIMAL, TOKEN_OCTAL,  
    TOKEN_HEXADECIMAL  
};
```

Como se observa en el código, dentro de la lista, cada token esta ordenado junto con el resto de tokens de un mismo nivel con respecto al esquema de la figura 40 para simplificar su comprensión.



### b) Función GetToken:

Una vez se tienen todos los tokens necesarios para la correcta comprensión de los símbolos y expresiones que pueden aparecer a lo largo de la ejecución del simulador, el siguiente paso es crear una función capaz de detectar que símbolo aparece en cada lugar de una expresión.

Para ello se crea la función GetToken, localizada dentro del archivo "Simulador.cpp". Esta función tiene el objetivo de devolver a partir de un String de entrada, un token identificativo del primer carácter que encuentre en la expresión. También tiene la capacidad de borrarlo para poder seguir analizando el resto.

La función queda definida mediante la siguiente cabecera:

```
int GetToken (QString &Buffer, bool removeIt) {}
```

En ella se puede observar en primer lugar que es una función que devuelve un número entero (int). Como entradas a la función tiene 2 variables:

- **QString &Buffer:** contiene la expresión a analizar formada por un conjunto de caracteres.
- **Bool removeIt:** esta variable booleana es la encargada de indicar si tras identificar un conjunto de caracteres y devolver el token a la función que la ha invocado se desea borrar esos caracteres (true) o no (false).

Para simplificar la explicación del funcionamiento de esta función, se empleara como ejemplo la expresión:

```
Expresion = "a + b && 7;"
```

Una vez se dispone de la expresión, se invoca a la función GetToken para detectar el primer carácter presente en la expresión y devolver el token correspondiente. Un ejemplo de invocación sería el siguiente:

```
int token=0;  
token = GetToken(Expresion,TRUE);
```



Mediante esas líneas de código, creamos una variable entera denominada “token” y hacemos una llamada a la función pasándole la expresión definida anteriormente indicando que queremos que al detectar un token lo borre de la expresión.

La expresión dentro de un String se almacena por espacios como se haría un vector, ocupando desde la posición “0” hasta el tamaño total de la cadena - 1. En nuestro caso sería:

Posición:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Valor:	[ ]	[a]	[ ]	[+]	[b]	[ ]	[&]	[&]	[7]

La función con sus primeras líneas, se encarga de borrar todos los espacios presentes inicialmente en la expresión mediante el siguiente código, calculando primero el tamaño de espacios blancos hasta el primer carácter con el while, y borrándolos del String almacenado en la variable Buffer:

```
int i=0;
bool borrarEspacios=false;

while (Buffer[i].isSpace() )
    i++;
    borrarEspacios=true;

if (borrarEspacios)
    //Borra 1 caracter a partir de la posición 0
    Buffer.remove(0,i);
    i=0;
```

Una vez eliminado los espacios, la situación de la expresión sería la siguiente:

Posición:	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Valor:	[a]	[ ]	[+]	[b]	[ ]	[&]	[&]	[7]

A continuación, la función detecta en la posición [0] un valor que no es un espacio. En primer lugar comprueba si es una letra. En caso afirmativo tiene una serie de palabras predefinidas como son “INPUT”, “OUTPUT”... que las detecta y devuelve su token.

En caso de no encontrar una palabra predefinida, como en nuestro ejemplo la “a”, encontraría una letra. Entonces, mediante el while de la parte inferior, detectaría el tamaño de la palabra, y devolvería un token indicativo de que ha encontrado una palabra “TOKEN\_WORD”.

En caso de que la variable booleana de entrada “removeIt” sea TRUE como es el caso, se borraría la palabra de la expresión al completo.

Todo lo descrito anteriormente se realiza con el código mostrado a continuación:

```
//Detectando letras
if ( (Buffer[0].isLetter()) )
int letter=0;
    if ( (Buffer[0] == 'I')    )
        if (Buffer[1] == 'N')
            if ( (Buffer[2] == 'P')
                && (Buffer[3] == 'U')
                && (Buffer[4] == 'T'))
                if (removeIt) Buffer.remove(0,5);
                letter=1;
                return TOKEN_INPUT;
            ...
if (letter==0)
    int k=1;
    while (Buffer[k].isLetter() )
        k++;
    if (removeIt) Buffer.remove(0,k);
    return TOKEN_WORD;
```

Eliminado el carácter “a” y devuelto el valor token, la expresión quedaría como se muestra a continuación:

Posición:	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Valor:	[ ]	[+]	[b]	[ ]	[&]	[&]	[7]

Si se volviera a mandar la expresión de nuevo, borraría el espacio en blanco inicial en primer lugar, y después volvería a detectar cual es el siguiente carácter presente en la expresión.

En esta ocasión, se encontraría con un símbolo “+”. Para detectarlo se emplean las siguientes líneas de código, las cuales en caso de encontrar un símbolo descrito en la condición de la estructura if, devuelve el token correspondiente después de borrarlo o no de la expresión:

```
//Detectando simbolos
if (Buffer[0] == '+')
    if (removeIt)      Buffer.remove(0,1);
    return TOKEN_MAS;
```

En el caso de que aparecieran símbolos que pueden ir seguidos de otros e indicar otra operación, como ocurre en el ejemplo con los símbolos “&&”, el código buscaría en primer lugar el dígito de la posición [0], y luego comprobaría el de la siguiente posición como se muestra a continuación, devolviendo en cada momento un token diferente:

```
if (Buffer[0] == '&')
    if (Buffer[1] == '&')
        if (removeIt)  Buffer.remove(0,2);
        return TOKEN_YLOGIC;
    else
        if (removeIt)  Buffer.remove(0,1);
        return TOKEN_AND;
```

Para terminar ya con la explicación de esta función, en el caso de encontrar un número, el funcionamiento sería similar al descrito en el caso de las letras, cambiando la parte de detección de letras indicada con la función “.isLetter” por la función “.isNumber”.

Si en la expresión apareciera algún carácter que no está incluido dentro de código detector de tokens, y por lo tanto no estaría incluido dentro de los símbolos del lenguaje Verilog, la función devolvería un token indicativo de error mediante el token TOKEN\_ERROR.

### c) Estructura del analizador de expresiones:

Una vez tenemos diseñado un sistema capaz de detectar qué carácter o conjunto de caracteres hay presentes en una expresión, que es capaz de devolvernos el resultado en forma de variables globales gracias a la lista enumerada, y que es capaz de borrar los caracteres para seguir analizando la expresión, solo queda crear la lógica del lenguaje Verilog, para crear las prioridades de los símbolos.

Para ello se crea una función para cada nivel de expresiones de acuerdo a la figura 40 (página 83), las cuales se van llamando recursivamente una a otra. Todas estas funciones se encuentran incluidas dentro de la clase MainWindow, presente dentro del archivo "mainwindow.cpp".

A continuación se va a hacer una breve explicación de la estructura de las expresiones, apoyándonos en el siguiente ejemplo:

Expresion = "inA == 6 || 9 >= 8 ? 10 : 100;"

inA = 12;

Fijándose en la estructura de la expresión (suponiendo que inA es una señal con ese valor), y adecuándola al esquema de las expresiones en lenguaje Verilog de la figura 40, tendríamos lo siguiente:

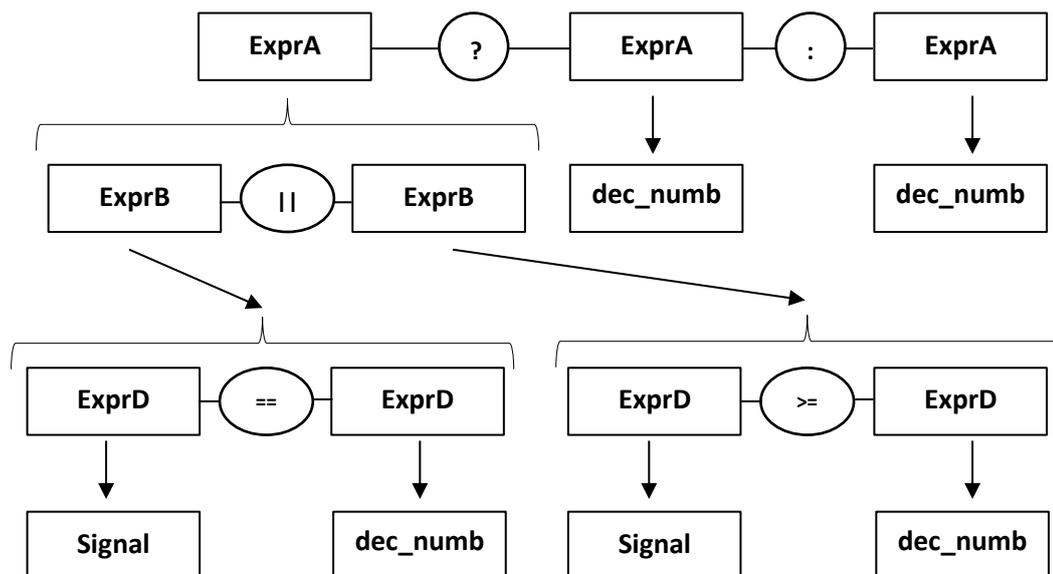


Figura 41: Desglose de la expresión en lenguaje Verilog



En la figura anterior podemos observar el desglose de la expresión que emplearemos de ejemplo de acuerdo a la nomenclatura de la figura 40 que sirve de base para describir la lógica de las expresiones en Verilog.

Para realizar la programación, en primer lugar se crea una expresión que se encarga de identificar frases vacías o de separar varias expresiones guardadas en un mismo String, las cuales pueden estar separadas por retornos de carros “%CR%” o con punto y coma “,”:

```
void MainWindow::ExprPreviaToken(QString &Frase)

Token=GetToken(Frase,FALSE);

if ( Token==TOKEN_NULL )
    TokenProcess.append("Frase vacia");
else
    if ( Token==TOKEN_RETORNOCARRO )
        TokenProcess.append("Retorno de carro");
        GetToken(Frase,TRUE);

    else if ( Token==TOKEN_PUNTOCOMA )
        TokenProcess.append("Fin de frase ");
        GetToken(Frase,TRUE);

    else
        ExprToken(Frase);

Token=GetToken(Frase,FALSE);
```

Como se observa en el código, en caso de no encontrar ninguno de los símbolos indicadores de que hay más de una expresión dentro del String, se manda la expresión al analizador de expresiones iniciales (Expr en la figura 40) la cual realiza lo mostrado en la siguiente imagen con el código que le prosigue:

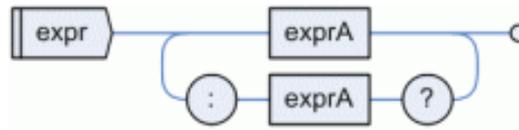


Figura 42: Estructura de las expresiones iniciales en lenguaje Verilog

```
void MainWindow::ExprToken(QString &frase)
while(1)
    if (i>1)

        //Mandamos la expresion A
        ExprAToken(frase);

        //Recibimos frase
        if (GetToken(frase, FALSE) == TOKEN_QUESTION)

            //Elimina ?
            TokenProcess.append("Encontrado ?");
            GetToken(frase, TRUE);
            TokenProcess.append(frase);

            //Vuelve a mandar la frase ya sin ?
            ExprAToken(frase);

            if (GetToken(frase, TRUE) != TOKEN_DOSPUNTOS)
                Error;

            TokenProcess.append("Encontrado :");
            i++;
```

Como se observa en el código, en primer lugar, la expresión se manda a otra función denominada ExprAToken, encargada de analizar expresiones A en busca de encontrar el valor del primer carácter de la expresión. La función se define de la forma siguiente y realiza en su código lo indicado en la figura que le precede:

- **Expresiones A:**

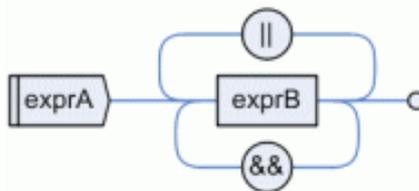


Figura 43: Estructura de las expresiones A en lenguaje Verilog

```
void MainWindow::ExprAToken(QString &frase)
```

Podemos obtener 2 cosas:

- En caso de ser un interrogante lo primero de la expresión (aunque no podría ser porque no sería una expresión válida) no recibiríamos nada y continuaríamos con el código de esta función.
- En caso de tener una señal, un número u otra expresión, la expresión iría pasando de funciones en funciones hasta devolver el valor de la expresión A, y podríamos seguir con esta función. El resto de funciones se declaran de la siguiente forma, mostrándose en primer lugar en imagen la lógica que realiza en su código:

- **Expresiones B:**

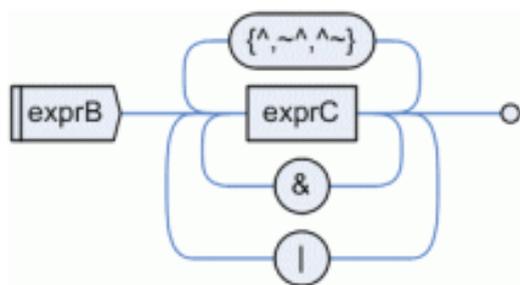


Figura 44: Estructura de las expresiones B en lenguaje Verilog

```
void MainWindow::ExprBToken(QString &frase)
```

○ Expresiones C:

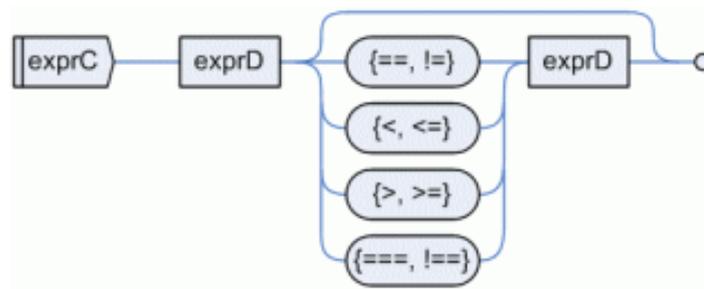


Figura 45: Estructura de las expresiones C en lenguaje Verilog

```
void MainWindow::ExprCToken(QString &frase)
```

○ Expresiones D:

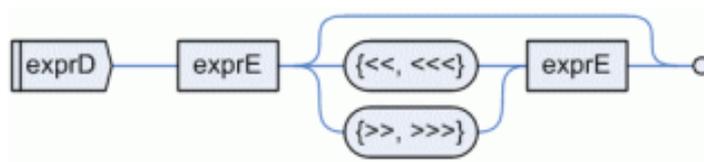


Figura 46: Estructura de las expresiones D en lenguaje Verilog

```
void MainWindow::ExprDToken(QString &frase)
```

○ Expresiones E:

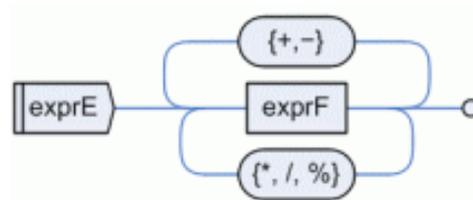


Figura 47: Estructura de las expresiones E en lenguaje Verilog

```
void MainWindow::ExprEToken(QString &frase)
```

- Expresiones F:

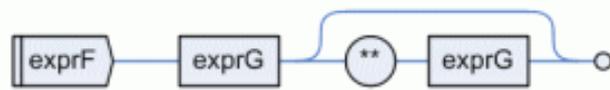


Figura 48: Estructura de las expresiones F en lenguaje Verilog

```
void MainWindow::ExprFToken(QString &frase)
```

- Expresiones G:

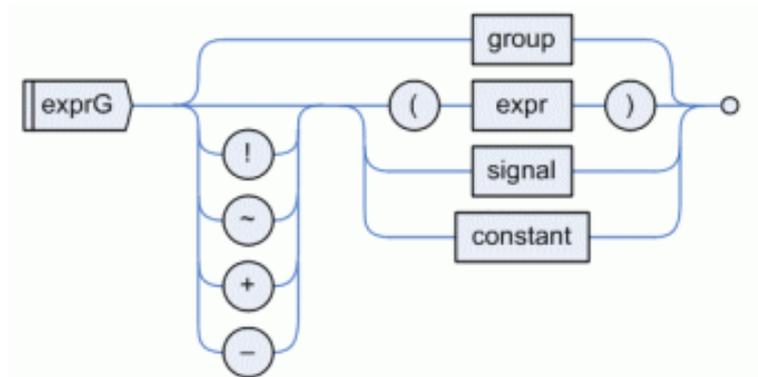


Figura 49: Estructura de las expresiones G en lenguaje Verilog

```
void MainWindow::ExprGToken(QString &frase)
```

Llegado a esta función, una vez detectado el símbolo que puede aparecer en primer lugar indicador de signo, pueden aparecer 4 tipos diferentes de expresiones (un grupo de expresiones, una expresión inicial entre paréntesis, una señal o una constante). Para diferenciar entre los 4 tipos se realiza el siguiente código (salvo los parámetros y los grupos que no entran dentro de las especificaciones):



```
GetToken(frase, TRUE);
Token = GetToken(frase, FALSE);

// ( Expr )
if (Token == TOKEN_PARENOPEN)
    ExprToken(frase);

//SIGNAL
if ( (Token == TOKEN_WORD))
    parametro=1;
    while (frase[j].isLetter())
        NameSignal.push_back(frase[j]);
        j++;

    NameSignal=NameSignal.toUpperCase();

    //Si el nombre es de una señal se manda a la función
    de señales. Sino a la de parametros
    for (i=0; i<Tamaño_Lista_Señales; i++)
        if(Lista[i]==NameSignal)
            SignalToken(frase);

    if (parametro==1)
        ConstantToken(frase);

// CONSTANT
if ( (Token == TOKEN_NUM)
    || (Token == TOKEN_APOSTROFE)
    || (Token == TOKEN_BINARIO)
    || (Token == TOKEN_DECIMAL)
    || (Token == TOKEN_OCTAL)
    || (Token == TOKEN_HEXADECIMAL))

    ConstantToken(frase);
```

Dependiendo del token obtenido en la función ExprGToken, la expresión pasa a una de las funciones subrayadas anteriormente, las cuales se muestran a continuación tanto en imagen como con la declaración de la misma:

- **Expresiones Signal:**

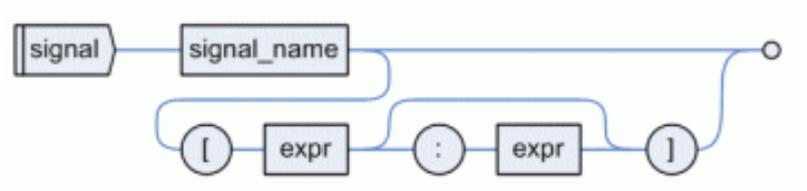


Figura 50: Estructura de las expresiones Signal en lenguaje Verilog

```
void MainWindow::SignalToken(QString &frase)
```

- **Expresiones Constantes:**

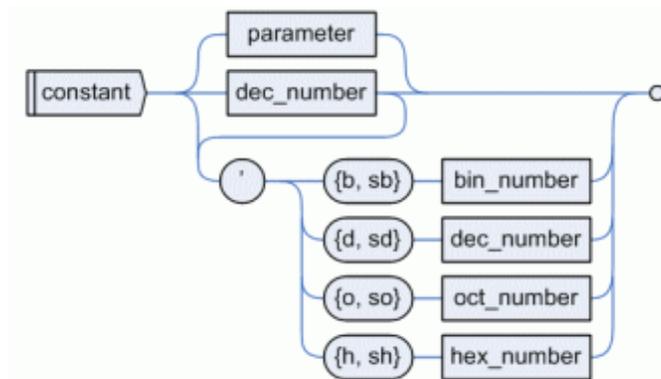


Figura 51: Estructura de las expresiones Constantes en lenguaje Verilog

```
void MainWindow::ConstantToken(QString &frase)
```

Una vez se ha llegado a estas funciones finales y se obtiene un valor, el valor final va volviendo a las funciones que lo han ido llamando, realizándose las operaciones indicadas a los símbolos encontrados en cada función, hasta llegar a la función ExprAToken inicial que lo siguiente que hace es invocar a la función explicada en el apartado B para saber si tiene que realizar ella alguna operación. Si obtenemos un “?” mandaríamos la expresión que le precede realizando el mismo proceso descrito en este apartado.

Al obtener un resultado buscaríamos el símbolo “:” y mandaríamos la expresión que le precede como se indica en la estructura de la figura 41.

Al finalizar este proceso podríamos devolver el valor deseado a la función ExprPreviaToken, que había invocado esta función.

#### d) Operaciones de las funciones del analizador de expresiones:

Cada función explicada anteriormente en el apartado c, al detectar un símbolo en una expresión y calcular el valor de las expresiones que hacen de operandos de esa operación, a continuación llama a una función que realiza la operación que tiene que realizar.

Por ejemplo, la función “ExprFToken”, cuya estructura se muestra en la siguiente imagen, tiene que realizar una operación exponencial:



Figura 52: Estructura de las expresiones G en lenguaje Verilog

Para realizar esta operación, al igual que el resto de funciones del apartado c) que requieren realizar una operación, dentro de esta función aparece la llamada a su función, mostrada subrayada a continuación en el código:

```
void MainWindow::ExprFToken(QString &frase)
    ExprGToken(frase);
    Token = GetToken(frase, FALSE);

    if (Token == TOKEN_EXPO)

        TokenProcess.append("Encontrado **");
        GetToken(frase, TRUE);
        ExprGToken(frase);
        Exponencia(NumAntes, NumAhora, NumDec);
```

Tanto esta operación como todas las realizadas por cada una de las funciones usadas en el analizador de expresiones se encuentran en el archivo “simulador.cpp”, cuyas declaraciones son:

- **Suma:**

```
void Suma(int &constante1, int &constante2, int &SUMA)
```

- **Resta:**

```
void Resta(int &constante1, int &constante2, int &RESTA)
```



- **Multiplicación:**

```
void Multiplicacion(int &constante1, int &constante2, int &MULTIPLICACION)
```

- **División:**

```
void Division(int &constante1, int &constante2, int &DIVISION)
```

- **Porcentaje:**

```
void Porcentaje(int &constante1, int &constante2, int &PORCENTAJE)
```

- **Desplazamiento de bits:**

```
void Desplazamiento(int &Valor, int &Cantidad, int Tipo, int &Salida)
```

- **Comparaciones:**

```
void Comparacion(int &Primero, int &Segundo, int Tipo, bool &Salida)
```

- **Operaciones lógicas de bit:**

```
void LogicaBit(int &Primero, int &Segundo, int Tipo, int &Salida)
```

- **Operaciones lógicas de expresiones:**

```
void LogicaExpr(bool &Primero, bool &Segundo, int Tipo, bool &Salida)
```

- **Operaciones condicionales:**

```
void Condicional(bool &Condicion, int &Cierto, int &Falso, int &Salida)
```

- **Exponencia:**

```
void Exponencia(int &Expresion1, int &Expresion2, int &Salida)
```

- **Función encargada de convertir todos los tipos de formatos permitidos para los números a formato decimal:**

```
int ConvertirUnidades(int TipoConversion, QString &Numero)
```

### 3.3.3. Desarrollo del código del simulador:

Realizada la programación del interfaz de usuario y del analizador de expresiones tal y como se ha descrito en los apartados 3.3.1 y 3.3.2., ya disponemos de los recursos suficientes para poder realizar la programación de la lógica global del simulador, con el fin de obtener el resultado final de la simulación de un diagrama ASM partiendo de un archivo con extensión .vdo.

En la imagen siguiente se indica el camino que seguiremos a continuación y como emplearemos los recursos programados hasta ahora:

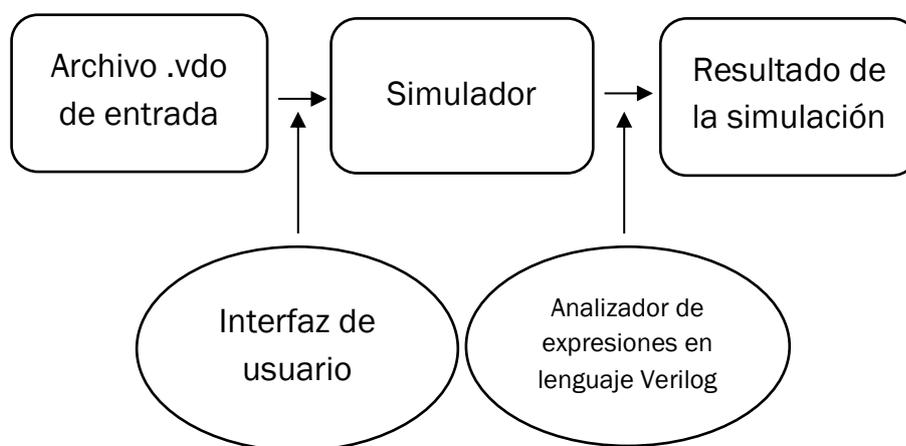


Figura 29: Estructura del simulador empleando los recursos creados anteriormente

En este apartado se va a explicar los pasos realizados para completar el proceso lineal desde que tomamos el archivo con extensión .vdo empleando el interfaz de usuario, realizamos la programación de la lógica del simulador y obtenemos el resultado final empleando el analizador de expresiones donde sea necesario.

#### a) Función simulación:

En primer lugar, para poder realizar todas las operaciones necesarias en el simulador, necesitamos crear una función que sea capaz de llamar al resto de funciones necesarias, en el orden deseado, para realizar la simulación de forma correcta. Para ello se crea la función:

```
void MainWindow::simulacion(const QString &fileName)
```

Como se había visto anteriormente, para poder disponer de un archivo .vdo el primer paso es seleccionar un archivo mediante el interfaz gráfico como se muestra en la siguiente imagen. En toda la explicación de este apartado se empleará como referencia el archivo denominado "Test\_Multiplier.vdo" correspondiente al diagrama de la figura 28 (página 68) que mostraba el diagrama ASM++ empleado para comprobar el correcto funcionamiento del simulador y que nos ha ayudado como ayuda en la explicación teórica.

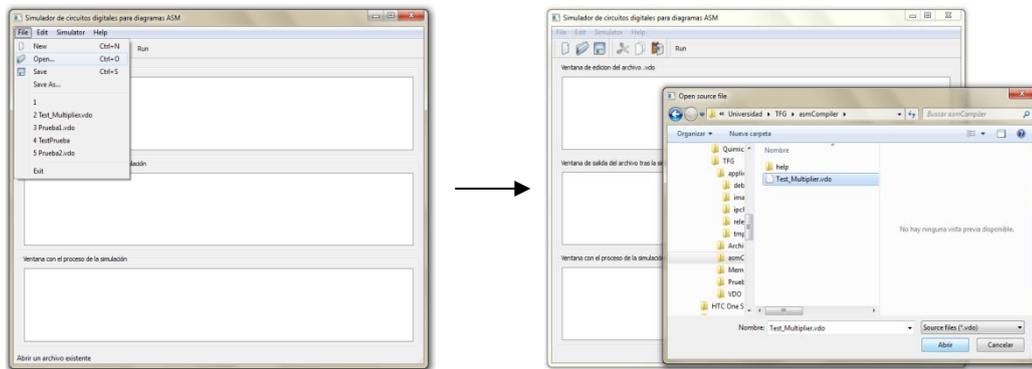


Figura 30: Explicación de la apertura de un archivo

Como se explicó en el apartado 3.3.1., al seleccionar el archivo deseado, este se abre en el editor de textos superior como se muestra en la siguiente imagen:

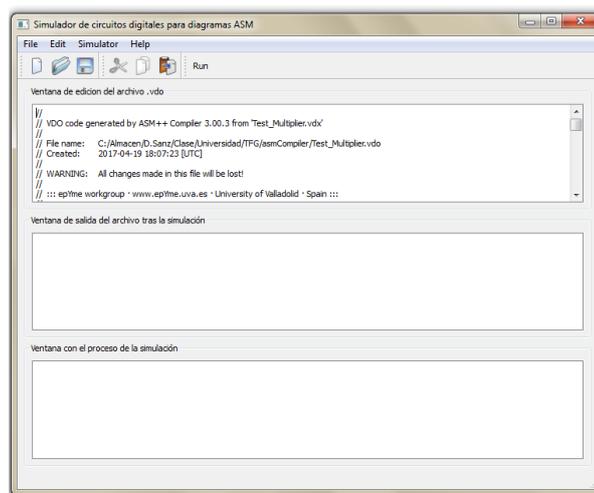


Figura 31: Simulador con el archivo abierto

Pulsando sobre el botón “Run”, como se había indicado en el apartado 3.3.1. al explicar la funcionalidad de esta función, en la última línea de función

```
void MainWindow::run()
```

se hace una llamada a la función “simulación” pasándole el archivo seleccionado en la variable “curfile” como se muestra a continuación

```
return simulacion(curFile);
```

Esta función tiene como objetivo realizar todo el proceso de simulación, llamando a las diferentes funciones creadas para realizar la simulación.

Para simplificar visualmente el proceso de simulación se incluye un esquema mostrando en cada bloque las diferentes funciones a realizar desde esta función simulación:

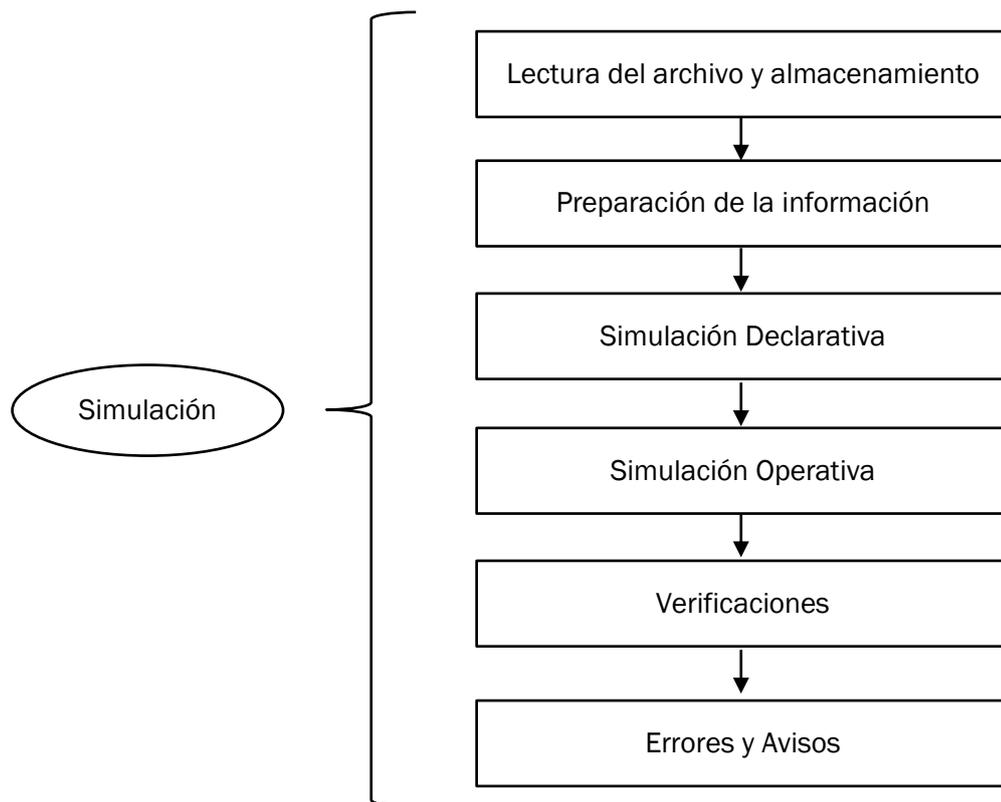


Figura 32: Estructura del código del simulador



A continuación se muestra un resumen del código de la función simulación con las diferentes llamadas a las funciones deseadas:

```
void MainWindow::simulacion(const QString &fileName)
```

- Guardamos el archivo de entrada y lo mandamos a la función que se encarga de leerlo y guardarlo:

```
FILE = fileName;  
LecturaArchivo(FILE);
```

- Preparamos los datos almacenados de forma que nos sea más fácil realizar la simulación:

```
OrdenandoBoxes();  
BoxCabecera();
```

- Realizamos una primera simulación con carácter declarativo, buscando las señales presentes en el diseño:

```
SimulacionDeclarativa();  
MostrarSignals();
```

- Realizamos una segunda simulación con carácter operativo, realizando las operaciones presentes en el diagrama para completar la simulación:

```
SimulacionOperativa();  
MostrarSignalPorCicloSimulacion();
```

- Realizamos las verificaciones finales:

```
MostrarVerificacionesARealizar();  
RealizarVerificaciones();
```

- Mostramos los errores y avisos encontrados a lo largo de la simulación:

```
ErroresYWarnings(NULL, MSG_Lectura);
```



## b) Lectura del archivo y almacenamiento de información:

El objetivo de esta función es recorrer completamente el archivo de entrada, interpretando todo lo que aparece en el archivo y almacenándolo en diferentes variables del sistema para tener toda la información disponible a la hora de simular.

Para leer cada línea del archivo se emplean fundamentalmente las funciones siguientes:

- Abrimos el archivo con la dirección almacenada en la variable FILE:

```
QFile archivo(FILE);
```

- Establecemos la secuencia de texto a leer, haciendo un streaming del archivo:

```
QTextStream textoLeer(&archivo);
```

- Leemos la línea de texto presente, con longitud predefinida con la variable "longline", y la almacenamos en un String:

```
QString LineaTexto;  
LineaTexto=textoLeer.readLine(longline);
```

Una vez disponemos de un sistema capaz de almacenar cada una de las frases del archivo en un String, el siguiente paso es crear una estructura capaz de diferenciar cada uno de los apartados presentes en los archivos .vdo coherente con lo explicado en el apartado 3.1.3. Estructura de los archivos .vdo compilados para simulación (página 64).

A continuación se muestra un esquema indicando los diferentes bloques que pueden aparecer (Pages y Box) así como de los diferentes apartados que contienen estos (incluido solo lo requerido dentro de las especificaciones):

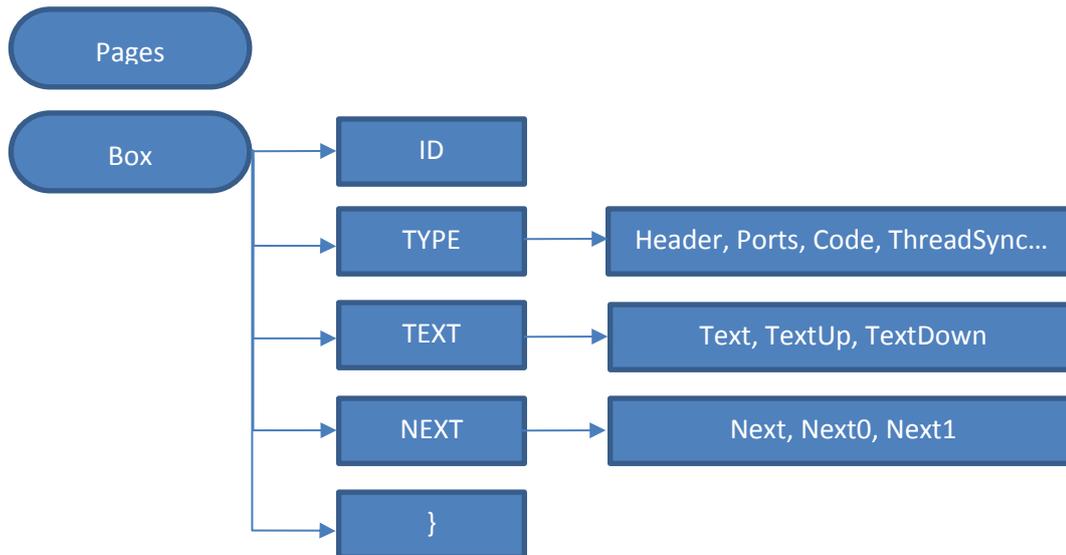


Figura 33: Estructura de los archivos .vdo

Para poder realizar la estructura anterior, en la función LecturaArchivo la implementación se realiza empleando las estructuras de c++ if y Switch como se muestra a continuación:

- Para saber si un String contiene una cadena de texto se emplea la función "QString.indexOf":

```
int page = LineaTexto.indexOf(Pages);  
int box = LineaTexto.indexOf(Box);
```

- Primero se diferencia entre páginas y cajas con las estructuras condicionales IF:

```
if (page != -1)  
if (box != -1)
```



- Después se diferencia entre las diferentes partes presentes dentro de las cajas con las estructuras IF y Switch (se incluye la detección de espacios en blanco):

```
int IDBox=LineaTexto.indexOf("Id");
int TypeBox=LineaTexto.indexOf("Type");
int TextBox=LineaTexto.indexOf("Text");
int NextBox=LineaTexto.indexOf("Next");
int LlaveBox=LineaTexto.indexOf("{}");

int PalabraEnBox=0;
if(IDBox!=-1)           { PalabraEnBox=1; }
else if(TypeBox!=-1)   { PalabraEnBox=2; }
else if(TextBox!=-1)  { PalabraEnBox=3; }
else if(NextBox!=-1)  { PalabraEnBox=4; }
else if(LlaveBox!=-1) { PalabraEnBox=5; }

switch (PalabraEnBox)

    case 0: //Espacio en blanco
    case 1: //ID
    case 2: //TYPE
    case 3: //TEXT
    case 4: //NEXT
    case 5: //}
```

- Dentro de cada caso del Switch, se diferencia entre las diferentes propiedades que pueden aparecer mediante las mismas estructuras IF y Switch. A continuación se muestra como sería para el caso del tipo de box:

```
case 2: //TYPE
    header = LineaTexto.indexOf(Header);
    ports = LineaTexto.indexOf(Ports);
    code = LineaTexto.indexOf(Code);
    ...
    if (header!=-1)
    else
        if (ports!=-1)           { Section=1; }
        else if (code!=-1)       { Section=2; }
        else if (threadsync!=-1) { Section=3; }
    ...

    switch (Section)
        case 0:
        case 1:
    ...
```



Una vez detectado cada parte mencionada anteriormente, se almacena la información en el programa para poder disponer de ella cuando sea necesario. Para almacenar la información se crean una serie de clases en el archivo “simulador.h”.

Para la información recogida de la lectura del archivo y almacenada por la función LecturaArchivo() se emplean las siguientes clases:

- Clase para las páginas (se almacena aunque no entra en las especificaciones): incluye el almacenamiento del Id y del nombre

```
class Pages
public:
    QVector<int>          ID;
    QVector<QString>     NAME;
```

- Clase para las cajas: incluye el almacenamiento del ID, Tipo, Texto (diferenciando entre texto normal, superior e inferior), ID siguiente (diferenciando entre next normal, next0 y next1) y la variable touch (que se empleará más adelante para saber si una caja está tocada o no en la simulación declarativa).

```
class Box
public:
    QVector<int>          ID;
    QVector<QString>     TYPE;
    QVector<QVector<QString>> TEXTMatrix;
    QVector<QVector<QString>> TEXTUPMatrix;
    QVector<QVector<QString>> TEXTDOWNMatrix;
    QVector<int>         NEXT;
    QVector<int>         NEXT0;
    QVector<int>         NEXT1;
    QVector<int>         TOUCH;
```

Al almacenar mediante vectores la información, estamos formando una estructura cuadriculada como la mostrada en la siguiente imagen, permitiéndonos poder acceder a toda la información requerida de un box simplemente sabiendo que posición ocupa dentro de la memoria (iteradorBox):

	iteradorBox	[0]	[1]	[2]	[3]	[...]
Qvector<String>	ID	7	10	28	30	...
Qvector<String>	TYPE	StateAsyncOps	StateAsyncOps	Header	ThreadSync	...
Qvector<Qvector<String>>	TEXTMatrix	null	null	multiplier_tb	dut.clk = 10 ns	...
Qvector<Qvector<String>>	TEXTUPMatrix	Test MIN x MIN <15>	Test MAX x MAX <15>	null	null	...
Qvector<Qvector<String>>	TEXTDOWNMatrix	=> dut.ready == 1;	=> dut.ready == 1;	null	null	...
		=> dut.done == 0;	=> dut.done == 0;	null	null	...
		dut.inA <= 0;	dut.inA <= 12'hFFF;	null	null	...
		...	...	...	...	...
Qvector<String>	NEXT	10	172	86	null	...
Qvector<String>	NEXT0	null	null	null	null	...
Qvector<String>	NEXT1	null	null	null	null	...
Qvector<int>	Touch	0	0	0	0	...

Figura 34: Estructura del almacenamiento de los datos del archivo de simulación

En el caso de que alguna celda este vacía se almacena con el valor null salvo la variable Touch que se almacena un 0 por ser de tipo entero.

Las variables que se encargan de almacenar el texto son de tipo Matriz, pero como no existe este tipo de variable, se emplea crea un tipo nuevo de dato formado por un vector de vectores. En cada posición del vector se guarda otro vector. Con esto logramos separar varias líneas de texto incluidas dentro de un mismo box, y poder ejecutarlas luego de forma más rápida en la simulación operativa y declarativa. La inicialización de estas variables se realiza con las siguientes funciones incluidas dentro de "mainwindow.cpp":

- MatrizTexto();
- MatrizTextoUp();
- MatrizTextoDown();

Una vez realizado todo este proceso, la función se encarga de cerrar el Streaming del archivo .vdo puesto que ya está almacenada toda la información importante para realizar correctamente la simulación.



### c) Preparación de la información:

Como se ha explicado en el punto anterior, ya tenemos almacenada toda la información existente en el archivo .vdo que nos permitirá realizar la simulación. Sin embargo, antes de poder pasar a realizarla, tenemos que ordenarla debidamente para que el proceso de simulación sea más sencillo y visual. Para ello se emplean las siguientes funciones:

#### - OrdenandoBoxes:

```
void MainWindow::OrdenandoBoxes()
```

El objetivo de esta función como su nombre indica es el de ordenar la información necesaria que tenemos almacenada en el sistema para poder realizar la simulación de la forma deseada.

Para poder tener toda la información ordenada como se desea, esta función se encarga de recorrer todo el vector "Type" de la clase "Box" identificando que tipo de caja es y almacenando su posición en memoria (iterador) en un tipo de contenedor denominado "QMultiHash". Este tipo de contenedor permite almacenar información con un parámetro común.

Para poder almacenar toda esta información, y toda la que se crea necesaria se crea la clase "OrdenBox" dentro del archivo "simulador.h". Esta clase contiene los siguientes parámetros:

```
class OrdenBox
public:
    QMultiHash<QString, int> POSICION;
    QVector<int> ItCabeceras;
    QVector<int> IdCabeceras;
    QVector<QString> TypeCabeceras;
    QVector<QVector<int>> IDStatePadresMatrix;
    QVector<QVector<int>> ITStatePadresMatrix;
```

Como se puede observar, en primer lugar dentro de esta clase aparece el contenedor QMultiHash denominado "POSICION". Esta variable almacena datos de tipo String y enteros. En nuestro caso, en el apartado de datos String se almacenará el tipo de la caja presente, y en el dato entero su posición en la memoria. El código diseñado para realizar este proceso es el siguiente:



```
QString Header="Header";
QString Ports="Ports";
QString Code="Code";
...

for(int x=0;x<BOX.TYPE.size();x++)
    TIPO=BOX.TYPE.at(x);

    if (TIPO==Header) { TypeSw=1; }
    else if (TIPO==Ports) { TypeSw=2; }
    else if (TIPO==Code) { TypeSw=3; }
    ...

    switch (TypeSw)

    case 1:

        ORDENBOX.POSICION.insert("Header",ORDENBOX.IterBox);
        ORDENBOX.IterBox++;
        break;
    case 2:

        ORDENBOX.POSICION.insert("Ports",ORDENBOX.IterBox);
        ORDENBOX.IterBox++;
        break;
    case 3:

        ORDENBOX.POSICION.insert("Code",ORDENBOX.IterBox);
        ORDENBOX.IterBox++;
        break;
    case 4:
    ...
```

Como se observa en el código, se recorre el vector TYPE de la clase box identificando el tipo de box encontrado, y dependiendo del tipo se almacena su iterador en el QMapMultiHash POSICION de acuerdo al tipo de caja correspondiente.



Una vez finalizado el resultado de este contenedor sería el mostrado a continuación (ejemplo obtenido del archivo de salida tras simular el archivo .vdo empleado para comprobar el correcto funcionamiento del simulador (figura 28, página 68):

```
//-----Posición en la memoria del "TYPE" de cada BOX -----//
Posiciones de las cajas Header en el archivo:      25,2
Posiciones de las cajas Ports en el archivo:      11
Posiciones de las cajas Code en el archivo:       26,9
Posiciones de las cajas ThreadSync en el archivo:  10,3
Posiciones de las cajas Event en el archivo:      20
Posiciones de las cajas Default en el archivo:    19
Posiciones de las cajas Instance en el archivo:   7
Posiciones de las cajas MetaState en el archivo:  4
Posiciones de las cajas StateAsyncOps en el archivo: 8,5,1,0
Posiciones de las cajas Initial en el archivo:    6
Posiciones de las cajas Decision en el archivo:   13,12
Posiciones de las cajas State en el archivo:      24,21,18,14
Posiciones de las cajas CondSyncOps en el archivo: 23
Posiciones de las cajas SyncOps en el archivo:   17,16,15
Posiciones de las cajas AsyncOps en el archivo:   22
```

En este ejemplo se puede observar la posición en memoria ocupada por cada box presente en el diseño elegido para simulación, separados por su tipo de caja.

#### - **BoxCabecera:**

```
void MainWindow::BoxCabecera()
```

Una de las informaciones principales a la hora de ir a realizar la simulación es tener identificados dentro de toda la información que cajas son padres, para a partir de ellos realizar la simulación de los diferentes caminos.

Para saber que box precede a cada box, existen 2 números ya explicados anteriormente:

- **ID:** se encarga de asignar a cada box un número determinado, que identifica a cada box.
- **Next:** en esta propiedad se incluye el numero ID del box que le precede, indicando por tanto que al salir de un box, el siguiente box del diseño es el que tiene como ID el next del box actual.

A continuación se muestra una imagen de esto último:

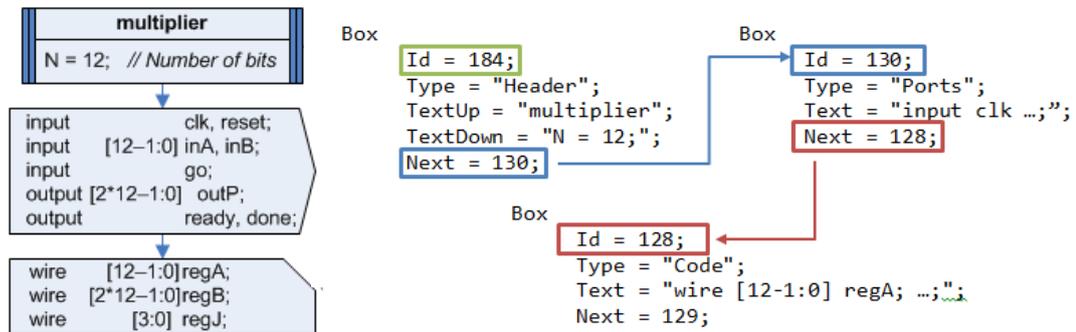


Figura 35: Relación de precedencia Next -> Id

En la imagen superior se puede observar a la izquierda el diseño de 3 cajas mediante diagramas ASM++, y a la derecha su descripción en el formato .vdo empleado para realizar las simulaciones.

Como se observa en la imagen, mediante los parámetros Id y Next se definen las uniones de las cajas, indicando así el camino que debe seguir el diseño. El box Header, identificado con el Id 184, tiene como box siguiente (Next) el box cuyo Id sea 130. En este caso, este es el box tipo "Ports".

Basándose en esto, el objetivo de la función "OrdenandoBoxes()" es recorrer toda la información almacenada en el sistema, box a box, detectando que cajas tienen un Id que no aparece en ningún next del resto de las cajas. Una vez encontrados los almacena en la clase "OrdenBox" del archivo "simulador.h" formada por los siguientes datos:

De todos los datos presentes en esta clase, para almacenar la lista de padres presentes en el diseño se emplean los 2 primeros: "ItCabeceras" e "IdCabeceras". El objetivo de estos 2 vectores es almacenar respectivamente tanto la posición en memoria de las cajas cabeceras, como su Id, para que al saber que Id corresponde a un padre, tengamos directamente la posición en memoria que ocupa ese box de acuerdo a la de la figura 58 (página 109).



#### d) Simulación declarativa:

Hasta ahora se ha estado preparando la información presente en el archivo de entrada para poder simularla. Una vez realizado todo ese proceso ya podemos empezar a realizar la simulación del circuito. A la hora de simular existen 2 pasos o tipos de simulaciones con una finalidad diferente: **Declarativa y Operativa**. En este apartado se va a explicar cómo se ha realizado la primera de ellas. Para ello se crea la siguiente función, incluida dentro del archivo “mainwindow.cpp”:

```
void MainWindow::SimulacionDeclarativa()
```

El objetivo de esta simulación, y por tanto de la función creada para realizarla, es el de recorrer el diagrama por orden, partiendo desde las cajas padre de acuerdo a la relación de precedencia (Next -> Id) explicada anteriormente buscando todas las señales presentes en el diseño (también se buscarían las variables en este apartado, pero no entran dentro de las especificaciones del proyecto). Al encontrar cada señal, se define de cada una los siguientes parámetros:

- Su **tipo**: Input, Output, Wire, Inout
- Sus **características**: número de bits [ : ]
- Su **sincronismo**: se las clasifica como señales Síncronas o Asíncronas.

Para realizarlo, el primer paso es identificar si los padres almacenados en la clase “OrdenBox” son padres de un diseño o de un test bench. A modo de recuerdo:

- **Diseño**: se encarga de definir la “lógica de un circuito”. No tiene fin.
- **Test Bench**: se encarga de definir cómo se va a realizar la simulación del circuito. Su fin es un box “MetaState” con el texto: “end simulation”. **El Test Bench no aporta información necesaria para esta simulación puesto que en el no se definen Señales del sistema.**

Como se acaba de mencionar, el test bench no aporta información necesaria para la simulación declarativa, por lo tanto al detectar que un padre pertenece a un Test Bench se “elimina” de esta simulación.

Para poder detectar si un box padre es de un diseño o de un test bench, la función SimulaciónDeclarativa() en primer lugar recorre Box a Box desde cada padre siguiendo la relación (Next -> Id) dejando una “marca” en cada box pasado hasta llegar a un box de tipo MetaState con texto “end simulation” (siendo el circuito con ese padre un Test Bench) o hasta llegar a un box con la “marca” indicando que por ese box ya se ha pasado.

Como se ha mencionado, a la hora de recorrer un diagrama de diseño en busca de las señales, hay que tener en cuenta que estos diagramas no tienen un box indicador de fin de diagrama como tienen los Test Bench. Por lo tanto, para saber si se ha recorrido todo el diseño sin pasar 2 veces por una misma caja y sin dejarse ninguna sin analizar se emplea el método de “marcas” mencionado anteriormente. El objetivo de este método es dejar una “marca” en cada box analizado para dejar constancia de que hemos pasado ya por ese box. Para entender mejor este método la explicación se apoya en la siguiente figura:

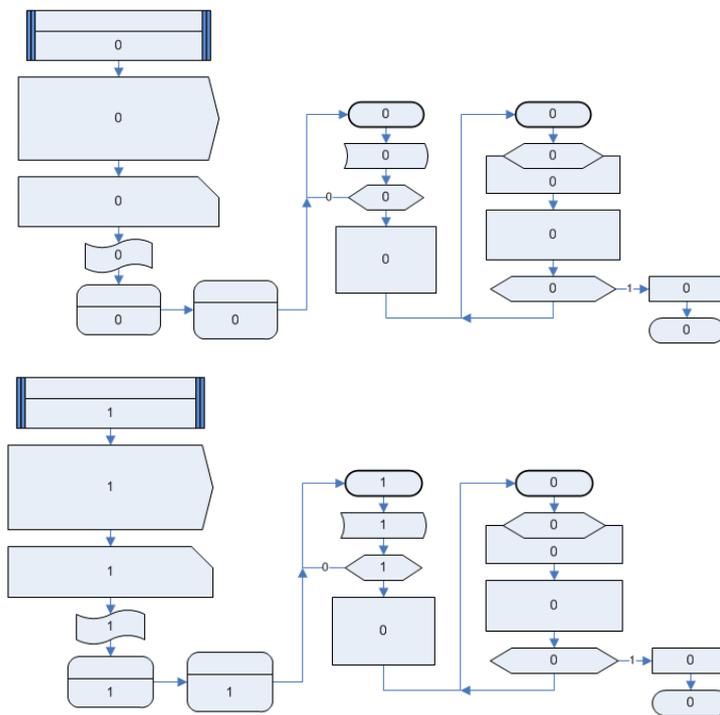


Figura 36: Diagrama explicativo del método de "marcas" empleado en la simulación declarativa

En la imagen anterior, perteneciente a la parte de diseño del diagrama ASM++ que está sirviendo de apoyo en toda la memoria para la explicación del simulador se observa la metodología de marcas empleada. En la imagen se ha suprimido el texto de cada box por simplificar visualmente la explicación.

Como se muestra en la imagen superior, inicialmente cada box se encuentra marcado con el valor “0”. Al pasar por cada uno de ellos, esta marca se pone a “1” indicando que ya se ha pasado por él. Esto se realiza para evitar crear bucles infinitos como puede suceder en el caso de cajas decisión, como se muestra en la imagen inferior, puesto que el diagrama se recorre sin interpretar la lógica interna de los mismos, solo yendo al Id indicado en el parámetro Next de cada box. En el caso del diagrama inferior, al llegar al bloque decisión no tomaría el camino con decisión 0 puesto que le llevaría a un bloque con una marca.

Como ya se había mencionado anteriormente en el apartado de lectura del archivo, esta información se almacena en la variable Touch de la clase "Box":

	iteradorBox	[0]	[1]	[2]	[3]	[...]
Qvector<String>	ID	7	10	28	30	...
Qvector<String>	TYPE	StateAsyncOps	StateAsyncOps	Header	ThreadSync	...
Qvector<Qvector<String>>	TEXTMatrix	null	null	multiplier_tb	dut.clk = 10 ns	...
Qvector<Qvector<String>>	TEXTUPMatrix	Test MIN x MIN <15>	Test MAX x MAX <15>	null	null	...
Qvector<Qvector<String>>	TEXTDOWNMatrix	=> dut.ready == 1;	=> dut.ready == 1;	null	null	...
		=> dut.done == 0;	=> dut.done == 0;	null	null	...
		dut.inA <= 0;	dut.inA <= 12'hFFF;	null	null	...
		...	...	...	...	...
Qvector<String>	NEXT	10	172	86	null	...
Qvector<String>	NEXT0	null	null	null	null	...
Qvector<String>	NEXT1	null	null	null	null	...
Qvector<int>	Touch	0	0	0	0	...

Figura 37: Almacenamiento en memoria de las marcas empleadas en la simulación declarativa

Teniendo en cuenta este método de "marcas" e identificado los padres de los diagramas de diseño, la función "SimulacionDeclarativa()" recorre el diagrama con diferentes modos de búsqueda, buscando en cada una de ellas una información u otra, poniendo a "0" todas las marcas al iniciar un modo de búsqueda. Los modos de búsqueda (ordenados por orden de ejecución) son los siguientes:

- Modo Ports:** en este primer modo de búsqueda, la función se encarga de recorrer el diagrama de diseño buscando únicamente las cajas de tipo "Code". Estas cajas son en las que se indican todas las señales de tipo INPUT y OUTPUT, así como su tamaño de bits.
- Modo Code:** este segundo modo de búsqueda tiene como función encontrar las cajas "Code" presentes antes del primer box tipo State, puesto que en ellas se definen todas las señales internas de tipo WIRE e INOUT y su tamaño de bits.
- Modo Async-Sync:** este último modo de búsqueda se encarga de recorrer todas las cajas del tipo siguiente:

```

if
( (BOX.TYPE.at(IteraPadre) == "AsyncOps")
|| (BOX.TYPE.at(IteraPadre) == "SyncOps")
|| (BOX.TYPE.at(IteraPadre) == "CondSyncOps")
|| (BOX.TYPE.at(IteraPadre) == "CondAsyncOps")
|| (BOX.TYPE.at(IteraPadre) == "StateAsyncOps")
|| (BOX.TYPE.at(IteraPadre) == "Event"))

```



Tras detectar una caja, se llama a la siguiente función:

```
void MainWindow::SaveSignals(int IteradorMemoria, int modo)
```

El objetivo de esta función es guardar las señales correctamente en memoria durante la simulación declarativa. A esta función la llega una variable de tipo entero indicando la posición en memoria a analizar, y una variable entera indicando el modo de búsqueda. Dependiendo del modo de búsqueda realiza una acción u otra:

- 1 **Modo Ports:** interpreta el texto presente en las cajas Ports, identificando mediante la función “GetToken” si hay una señal tipo INPUT u OUTPUT.
- 2 **Modo Code:** interpreta el texto presente en las cajas Code, identificando mediante la función “GetToken” si hay una señal tipo WIRE o INOUT.
- 3 **Modo Async-Sync:** este método, al ser el último de los 3, ya dispone de todas las señales del sistema almacenadas. Su objetivo es recorrer las cajas cuyo tipo se ha indicado anteriormente buscando en su texto si aparece alguna de las señales. En el caso de aparecer, dependiendo del tipo de caja que aparezca el sincronismo de esa señal será uno u otro. Por ejemplo, cualquier señal presente en un box tipo AsyncOps es asíncrona.

Toda la información de las señales se almacena en la clase “Signal” del archivo “simulador.h”. Esta clase está formada por los siguientes datos (solo se muestran los empleados en la simulación declarativa):

```
class Signal
public:
    QVector<QString>      InputList, OutputList,
                        InoutList, WireList;
    QVector<QString>      CaractInput, CaractOutput,
                        CaractInout, CaractWire;
    QVector<QString>      SyncInput, SyncOutput,
                        SyncInout, SyncWire;
```

Al igual que se hacía con la información de las cajas, se emplean vectores para tener almacenada en forma de “tabla virtual” todas las señales, sus características (tamaño de bits) y su sincronismo como se muestra a continuación:

	Iterador	[0]	[1]	[2]	[3]	[...]
Qvector<String>	InputList	clk	reset	inA	inB	...
Qvector<String>	CaractInput			[11:0]	[11:0]	...
Qvector<String>	SyncInput	ASYNC	ASYNC	SYNC	SYNC	...
Qvector<String>	OutputList	outP	ready	done		...
Qvector<String>	CaractOutput	[23:0]				...
Qvector<String>	SyncOutput	SYNC	ASYNC	SYNC		...
Qvector<String>	WireList	regA	regB	regC		...
Qvector<String>	CaractWire	[11:0]	[23:0]	[3:0]		...
Qvector<String>	SyncWire	SYNC	SYNC	SYNC		...
Qvector<String>	InoutList					...
Qvector<String>	CaractInout					...
Qvector<String>	SyncInout					...

Figura 38: Estructura del almacenamiento de las señales en la memoria

Tras realizar la simulación del diagrama de la figura 28 (el mismo que el empleado para el resto de la explicación de la memoria) se obtiene a la salida la siguiente información de las señales:

//----- Lista de señales -----//

- Inputs:
  - CLK ASYNC
  - RESET ASYNC
  - INA SYNC [11:0]
  - INB SYNC [11:0]
  - GO ASYNC
- Outputs:
  - OUTP SYNC [23:0]
  - READY ASYNC
  - DONE SYNC
- Wires:
  - REGA SYNC [11:0]
  - REGB SYNC [23:0]
  - REGJ SYNC [3:0]
- Inouts:

### e) Simulación operativa:

Como se mencionaba anteriormente, el simulador realiza 2 tipos de simulaciones: **Declarativa y Operativa**. En el apartado anterior se ha realizado la explicación de la primera de ellas. En este apartado se indicará como se ha realizado la segunda.

Una vez definidas todas las señales y variables del circuito en los apartados anteriores (variables fuera de especificaciones) el objetivo de la simulación operativa es realizar la simulación total del circuito, recorriendo totalmente tanto el diseño como el test bench, analizando las expresiones, tomando decisiones para decidir que camino seguir y obteniendo los valores de cada señal en cada ciclo de simulación.

Para tener junta toda la simulación operativa se crea la siguiente función, la cual llama a otras dependiendo lo que desee hacer en cada situación como se muestra en la siguiente figura:

```
void MainWindow::SimulacionOperativa()
```

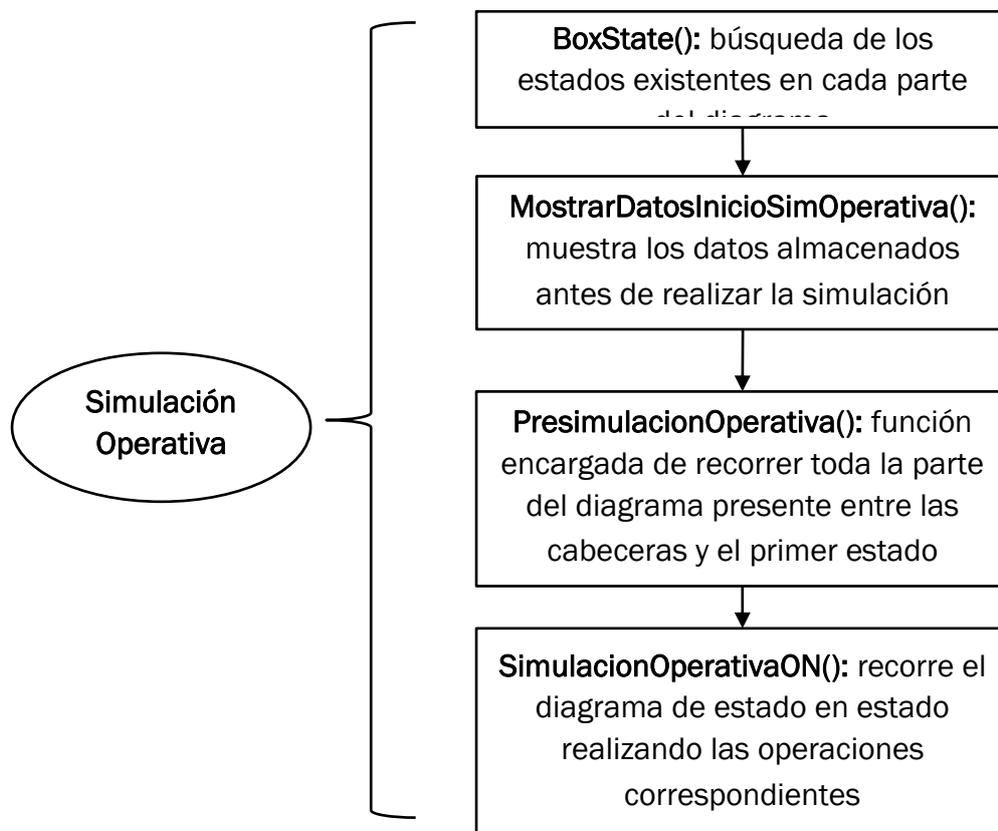


Figura 39: Estructura de la simulación operativa



Como se indica en la figura anterior, para realizar correctamente la simulación esta función se divide en 4 apartados definidos a continuación:

1. **BoxState():** El primer paso consiste en recorrer el diagrama partiendo de los diferentes partes según la relación (Next -> Id) guardando los diferentes estados existentes. Cada estado indica el inicio de un ciclo de reloj, por eso es necesario saber que cajas “State” (o cualquier box similar (“StateAsyncOps”,...)).

Para realizar este proceso se crea la función “BoxState()”:

```
void MainWindow::BoxState()
```

Al igual que hacia la función BoxCabecera() descrita antes de la simulación declarativa, esta función se encarga de buscar cada uno de las cajas “State” y de almacenarlos en la clase OrdenBox, usando las variables mostradas a continuación para guardar su ID y su posición en memoria:

```
class OrdenBox
```

```
public:
```

```
    QVector<QVector<int>> IDStatePadresMatrix;
```

```
    QVector<QVector<int>> ITStatePadresMatrix;
```

2. **MostrarDatosInicioSimOperativa():** El siguiente paso simplemente tiene como objetivo mostrar la información disponible antes de realizar la simulación. Para ello se crea la función:

```
void MainWindow::MostrarDatosInicioSimOperativa()
```



Para el ejemplo de la figura 28, obtenemos la siguiente información:

```
//-----SIMULACION OPERATIVA-----//
                Padre Nº1  Padre Nº2
                -----  -----
- TYPE Cabeceras:   TestBench  Design
- ID Cabeceras:      28      184
- IT Cabeceras:      2       25
- ID States:
    +ID State 1:      41      156
    +ID State 2:      90      143
    +ID State 3:       7      147
    +ID State 4:      10
    +ID State 5:     172
    +ID State 6:      35

- IT States:
    +It State 1:       5      21
    +It State 2:       8      14
    +It State 3:       0      18
    +It State 4:       1       0
    +It State 5:     24       0
    +It State 6:       4       0
```

Como se observa, antes de realizar la simulación operativa como tal disponemos de toda la información necesaria para realizarla correctamente, incluyendo los diferentes padres presentes, el tipo, su Id y su posición en memoria, y todos los estados (Id e iterador en memoria) de cada padre.

**3. PresimulacionOperativa():** El tercer paso tiene como finalidad realizar lo que se ha denominado “Presimulación Operativa”. Este concepto hace referencia a todas las operaciones que se realizan antes del primer estado. Para ello se crea la siguiente función:

```
void MainWindow::PresimulacionOperativa()
```

Dentro de las especificaciones se indica que el simulador solo debe interpretar diagramas con 1 diseño y 1 test bench, por lo tanto la presimulación operativa como la simulación entre estados se realiza de acuerdo a estas especificaciones.

Para entender mejor de que se encarga esta presimulación operativa se muestran las siguientes figuras, correspondientes a las secciones declarativas explicadas en el apartado 3.1.2. de este capítulo (página 42):

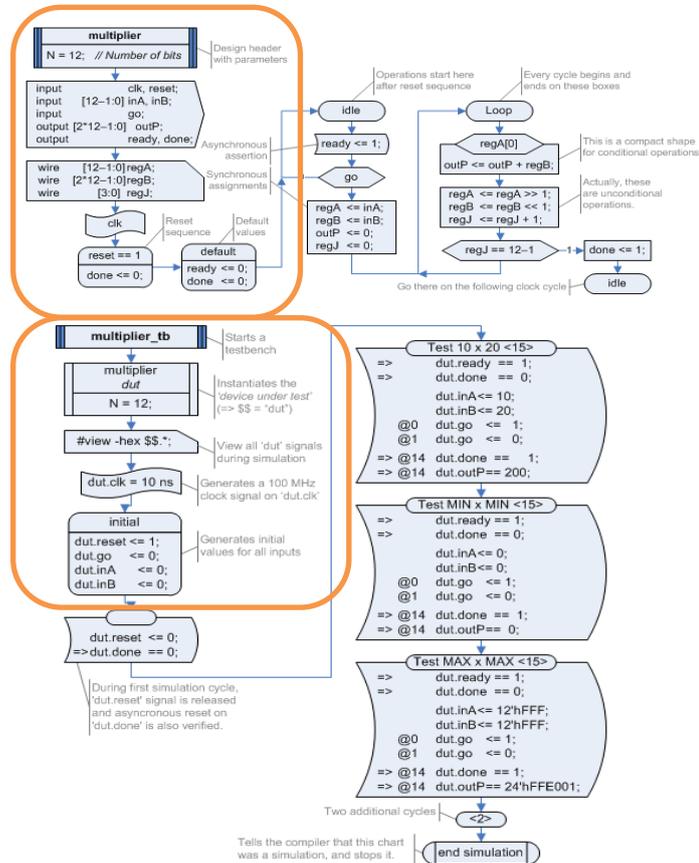


Figura 40: Zona del diagrama en la que se centra la Presimulación Operativa

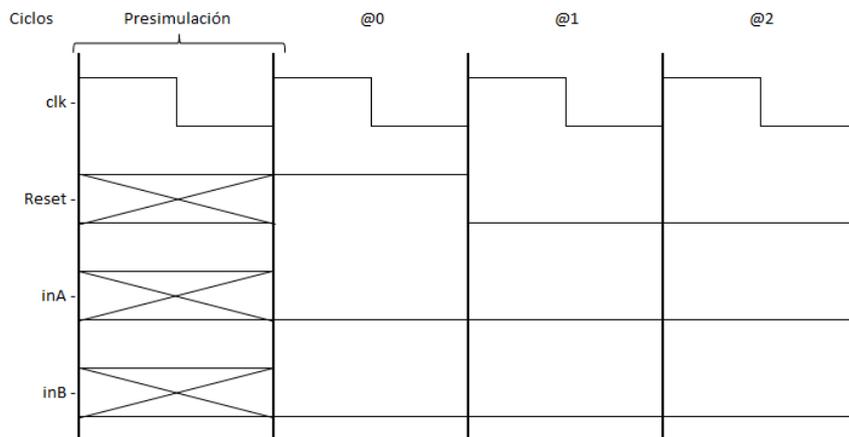


Figura 41: Periodo de tiempo en el que se sitúa la Presimulación Operativa



Como se indica en la figura inferior, esta presimulación se encarga de establecer los valores que tendrán las señales inicialmente antes del primer ciclo de reloj, así como de guardar los posibles eventos o de establecer la señal de reloj que establece el sincronismo.

Para almacenar los eventos se añaden a la clase Signal una serie de parámetros mostrados a continuación:

```
class Signal

public:
    ...
    QVector<QString>    SignalUPEvent,
                       SignalDOWNEvent;
    QVector<int>        SignalUPEventValor,
                       SignalDOWNEventValor;
    QVector<QString>    SignalDefaultName;
    QVector<int>        SignalDefaultValor;
```

Con estas nuevas variables añadidas se almacenan tanto los nombres de las señales que ejecutan eventos y sus valores como las variables que se modifican al ejecutarse sus eventos y sus valores. También se crean las variables para establecer los nombres y valores de las señales por defecto.

La programación de esta función es similar al de la Simulación entre ciclos, por lo tanto su descripción se realiza en el siguiente apartado. La única diferencia es el tipo de cajas que se analiza en esta presimulación, puesto que en la zona declarativa aparecen cajas diferentes a los presentes en la zona operativa como se mostraba en el apartado 3.1.2. (Página 42).

- 4. SimulacionOperativaOn():** realizada la presimulación operativa, encargada de analizar y ejecutar todo lo indicado antes del primer estado (tanto para el diseño como para el test bench del diagrama) esta parte se encarga de realizar la simulación entre ciclos de reloj. Toda la programación de esta simulación entre estados se realiza en la siguiente función:

```
void MainWindow::SimulacionOperativaON()
```

Para entender mejor su estructura se muestra el siguiente diagrama:

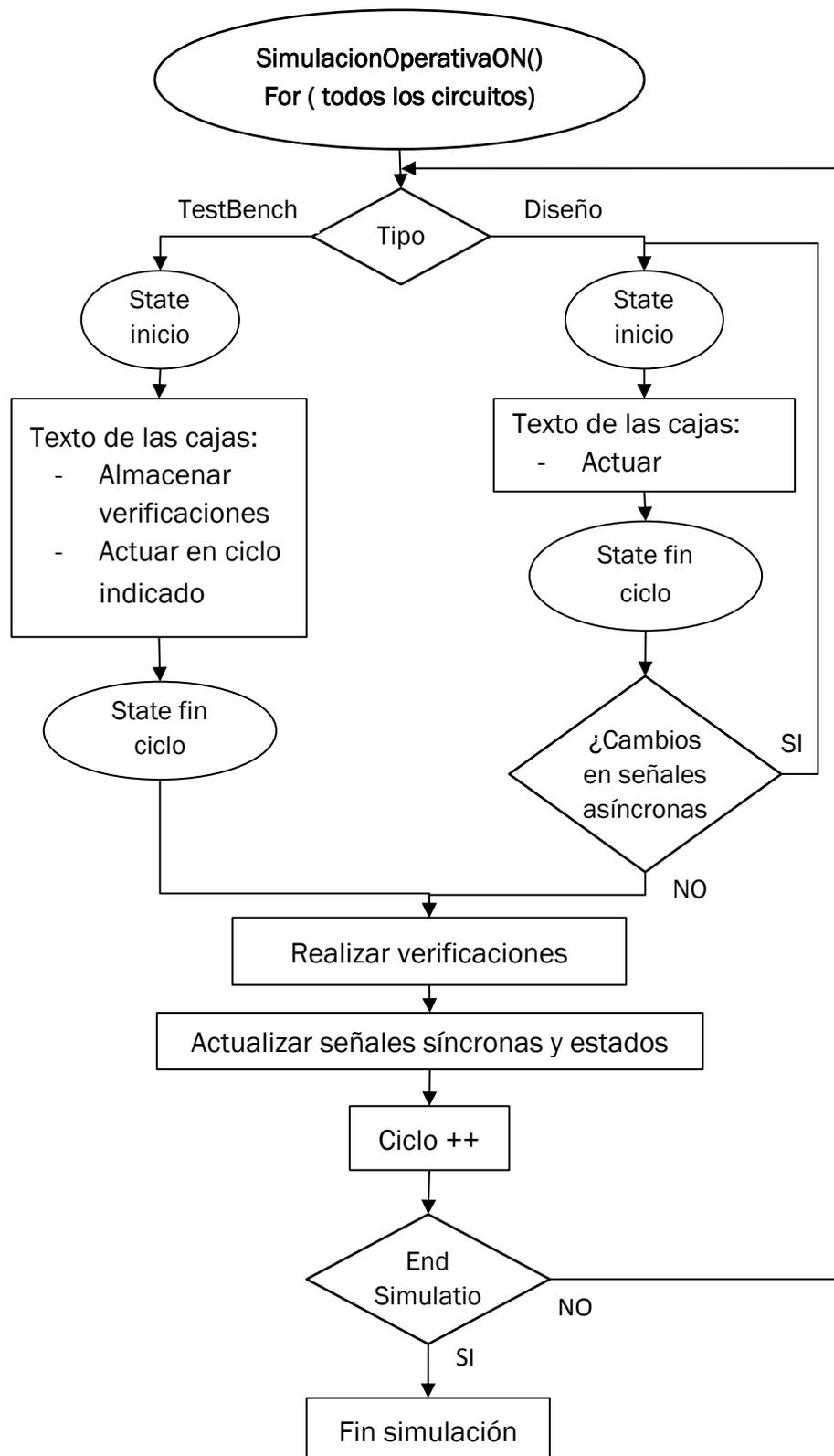


Figura 42: Estructura de la función SimulaciónOperativaOn

Como se observa en la figura, en primer lugar en la función SimulacionOperativaON se parte de los padres del diagrama. Dependiendo del tipo de padre tendremos o un diseño o un test bench. En cada uno de ellos, partiendo desde el estado que toque ejecutarse en ese momento, se realizarán las acciones correspondientes hasta encontrar el siguiente box estado que indica el final.

Al terminar se vuelve al inicio y se ejecuta el padre del otro tipo. Inicialmente siempre se ejecuta primero el test bench, y el ciclo de reloj se aumenta al llegar al estado final del diseño.

- **Test Bench:** como se indicaba en el apartado de teoría del simulador (página 28), en un test bench podemos encontrarnos bloques de estados, en los cuales en la parte superior se indica el número de ciclos que se tiene que ejecutar. En su texto inferior aparecerá lo que se debe realizar en cada ciclo de reloj, diferenciando entre verificaciones y actuaciones sobre señales:
  - **Verificación en el primer ciclo ( “=>” ):** Cuando aparece este símbolo antes de una expresión, se indica que justo antes de finalizar el primer ciclo se debe verificar la condición que le precede.
  - **Verificación en el ciclo indicado ( “=> @N°Ciclo”):** Cuando aparece este símbolo antes de una expresión se indica que en el ciclo indicado después del símbolo @ se tiene que realizar la verificación indicada por la expresión.
  - **Actuar en el ciclo indicado (“@N°Ciclo”):** Cuando aparece solo este símbolo antes de una expresión, se indica que en el ciclo indicado después del símbolo @ se debe ejecutar la expresión indicada
  - **Actuar en el primer ciclo:** En el caso de aparecer una expresión en lenguaje HDL sin ningún símbolo previo se tendrá que ejecutar la expresión durante el primer ciclo en el que se ejecuta el box como si pusiera un @0.



En el caso de encontrarnos verificaciones, todas se almacenan en la clase “Verificaciones” del archivo “simulador.h”. Esta clase está formada por los siguientes datos:

```
class Verificaciones

public:
    QVector<int>          Ciclo;
    QVector<QString>     NameSignal;
    QVector<int>         Valor;
```

Al igual que en el resto de clases, el almacenamiento de estos datos se realiza en vectores para poder acceder a ellos conociendo la posición de la verificación requerida. En la variable Ciclo se almacena el ciclo en el que hay que comprobar la verificación, en la variable NameSignal el nombre de la señal sobre la que hay que realizarla, y en Valor el valor de la señal a verificar.

En el caso de tener que actuar sobre señales, el valor de la nueva señal se almacena en el vector que hace referencia a los valores de las señales de la clase Signal, después de buscar la señal en las diferentes listas para saber su tipo:

```
SignalClass.InputValor.replace(posSignal,Number);
```

o

```
SignalClass.OuputValor.replace(posSignal,Number);
```

o

```
SignalClass.WireValor.replace(posSignal,Number);
```

o

```
SignalClass.InoutValor.replace(posSignal,Number);
```



- **Diseño:** como se indicaba en el apartado de teoría del simulador (página 28), en un diseño se indica la lógica del circuito digital. Al igual que en el test bench, para simular el diseño de un diagrama ASM++, se parte desde un estado, recorriendo todo el diagrama según la relación (Next -> Id) hasta encontrar el siguiente estado.

Al finalizar este proceso, hay que comprobar si algún evento se ha activado. Para eso se emplea la siguiente función:

```
void MainWindow::AnalizadorEventos()
```

Esta función se encarga de comprobar si alguna condición que ejecuta eventos esta activa, y si es así ejecuta los cambios indicados modificando el valor de las señales en la clase Signal.

Como se indicaba en la página 57, a la hora de simular la parte de diseño hay que tener en cuenta la siguiente pregunta: ¿cuándo se puede dar por finalizado un ciclo de reloj?

La respuesta es: solo se puede dar por finalizado un ciclo de reloj, cuando ninguna señal ha sido modificada entre el estado inicial y final de ese ciclo de reloj.

Por lo tanto, a la hora de realizar la simulación de la parte de diseño, tenemos que comparar los valores de las señales antes del estado que inicia el ciclo y en el estado que indica el fin del ciclo. Dependiendo de si las señales son síncronas o asíncronas, su valor cambiara después del ciclo del reloj actual o durante el ciclo de reloj actual respectivamente. Para comprobar si los valores de las señales han modificado durante el ciclo en primer lugar se añaden las siguientes variables a la clase signal:

```
class Signal

public:
    ...
    QVector<int>      InputValorAnt, OutputValorAnt,
                    InoutValorAnt, WireValorAnt;
    QVector<int>      InputValorDesp, OutputValorDesp,
                    InoutValorDesp, WireValorDesp;
```

Con las variables que acaban con la palabra “Desp” se guardan los valores que tendrán en el siguiente ciclo de reloj las señales (señales síncronas). Con las variables que acaban en “Ant”, guardadas al inicio de la simulación del diseño se comparan los vectores de la clase Signal donde están los valores actuales:



```
if (SignalClass.InputValorAnt!=SignalClass.InputValor)
    CAMBIO=1;

if (SignalClass.OutputValorAnt!=SignalClass.OutputValor)
    CAMBIO=1;

if (SignalClass.InoutValorAnt!=SignalClass.InoutValor)
    CAMBIO=1;

if (SignalClass.WireValorAnt!=SignalClass.WireValor)
    CAMBIO=1;
```

En caso de que el resultado de la comparación sea que alguna señal se ha modificado, mediante la función “goto” de C++, se vuelve al inicio de la simulación del diseño. En caso de que ninguna señal se haya modificado se pasa a guardar la simulación, aumentando el ciclo de simulación.

Esto se realiza mediante la función:

```
void MainWindow::GuardarSimulacion(int CYCLE)
```

Esta función se encarga de guardar al final de cada ciclo de reloj todos los valores de la simulación en la clase DatosSimulacion, la cual tiene los siguientes parámetros (para almacenar el nombre de la señal y su valor en el ciclo de la simulación):

```
class DatosSimulacion
public:
    QVector<QString>          SignalSim;
    QVector<QVector<int>>    ValoresSim;
```

Una vez almacenado todo en el ciclo de reloj, como ya se ha aumentado el ciclo, se comprueba si hemos llegado al final del Test Bench, en caso afirmativo, se da por finalizada la simulación. En caso negativo se vuelve al inicio de la función SimulacionOperativaOn() y se analiza el siguiente ciclo de reloj.

Una vez finalizados todos los ciclos de reloj, marcados por el Test Bench, se muestra el resultado final de la simulación mediante la función:

```
void MainWindow::MostrarSignalPorCicloSimulacion()
```

Esta función se encarga de recorrer la clase DatosSimulación, guardando en el archivo de texto de salida el valor de cada señal en su ciclo. El ciclo corresponde con la posición de los valores de las señales almacenados en los vectores de la clase DatosSimulación. El resultado final de la simulación para el diagrama empleado durante la explicación es el siguiente (debido a su largo tamaño solo se incluyen los 8 primeros ciclos de simulación, el archivo final está incluido con el resto del código del simulador en el archivo txt de salida del simulador):

	//-----SIMULACION-----//									
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	...
CLK	10	10	10	10	10	10	10	10	10	...
RESET	1	0	0	0	0	0	0	0	0	...
INA	0	0	10	10	10	10	10	10	10	...
INB	0	0	20	20	20	20	20	20	20	...
GO	0	0	1	0	0	0	0	0	0	...
OUTP	-1	-1	-1	0	0	40	40	200	200	...
READY	0	1	1	1	1	1	1	1	1	...
DONE	0	0	0	0	0	0	0	0	0	...
REGA	-1	-1	-1	10	5	2	1	0	0	...
REGB	-1	-1	-1	20	40	80	160	320	640	...
REGJ	-1	-1	-1	0	1	2	3	4	5	...

Tanto para la parte de test bench como para la de diseño, a la hora de analizar los textos de cada box se emplea la función `GetToken()`, así como el analizador de expresiones, diseñados anteriormente.

#### f) Verificaciones:

Como se ha visto en el apartado anterior, el resultado final de la simulación operativa es una tabla en la cual se muestra el valor de cada señal en los diferentes ciclos de simulación. Con esto se da por finalizada la simulación del circuito.

Sin embargo, el trabajo del simulador aún no ha terminado. El objetivo del simulador no es solo simular un circuito digital, sino comprobar una serie de verificaciones indicadas en el diagrama. Estas verificaciones, como se ha visto en el apartado anterior, se encuentran en la parte del test bench del diagrama. Durante la simulación operativa se han ido almacenando en la clase Verificaciones todas las verificaciones a realizar.

Una vez finalizada la simulación operativa, mediante la función mostrada a continuación, se recorre la clase DatosSimulación, en la que están almacenados los valores de cada señal en cada ciclo de



simulación, y se comprueba el valor de las señales de acuerdo a lo almacenado en la clase Verificaciones:

```
void MainWindow::RealizarVerificaciones()
```

Esta función, en su parte final, llama a la función encargada de mostrar el resultado de las verificaciones por pantalla, y almacenando en el archivo txt de salida el resultado total de las verificaciones. La función encargada de realizar este proceso es la siguiente:

```
void MainWindow::MostrarVerificacionesARealizar()
```

El resultado obtenido para el diagrama simulado a lo largo de la memoria es el siguiente:

```
//-----VERIFICACIONES-----//
Ciclo de actuacion:   1           2           2           15   ...
Señal a verificar:   DONE        READY        DONE        DONE  ...
Valor a verificar:   0           1           0           1     ...
```

```
Verificaciones correctas: 97
```

```
Verificaciones erroneas: 0
```

#### g) Errores y avisos:

Para dar por finalizado el desarrollo del simulador, y antes de escribir todos los datos de la simulación en el archivo de salida, se hace una llamada a una función, encargada de comprobar cuantos errores y avisos ha habido a lo largo de la simulación, desde que se selecciona el archivo a simular hasta que se comprueban las verificaciones. Esta función es la siguiente:

```
void MainWindow::ErroresYWarnings(QString ErrWar, int tipo)
```

A lo largo de la simulación, cada vez que aparece un error o es conveniente realizar un aviso se realiza una llamada a esta función.

A esta función le llegan 2 valores: un String, con el error o el aviso que se quiere almacenar, y un entero. Este entero puede tener 3 valores:

- Valor 0: indica que lo que se quiere es leer todos los errores y avisos y mostrarlos por pantalla.
- Valor 1: se quiere almacenar un error
- Valor 2: se quiere almacenar un aviso

Los valores almacenados y leídos se almacenan en la clase Fallos de “simulador.h”, la cual está formada por los siguientes parámetros:

```
class Fallos  
  
public:  
    QVector<QString> Warnings;  
    QVector<QString> Errores;
```

Para el diagrama simulado a lo largo de la memoria, obtenemos como avisos y errores lo siguiente:

```
//----- WARNINGS Y ERRORES -----//  
Warning!: BOX CODE no implantado en Presimulacion Operativa  
del Test Bench  
Numero de Warnings: 1  
Numero de Errores: 0
```

Con esto se puede comprobar el correcto funcionamiento de la función. Aparece un aviso indicando que las cajas Code no se encuentran implantados dentro de la Presimulación Operativa del Test Bench. Efectivamente, el diagrama ASM++ simulado contiene en su Test Bench un box Code, el cual se encarga de definir una variable para todo el Test Bench:

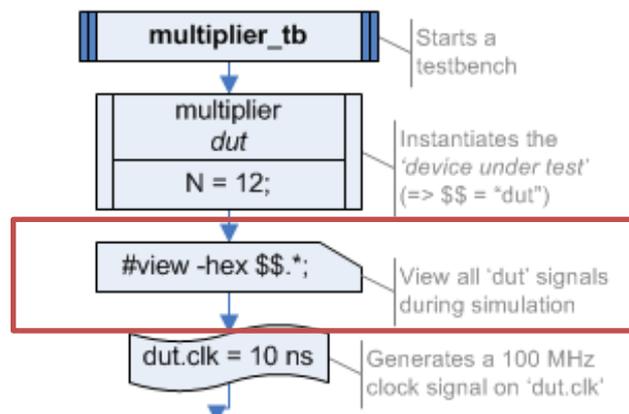


Figura 43: Box Code presente en la sección declarativa del Test Bench

Al no estar incluidas las variables dentro de las especificaciones, esas cajas no se encuentran incluidos en esa parte de la simulación, pero se ha dejado en el diagrama para comprobar que la función de avisos y errores funciona correctamente.



#### h) Archivo .txt de salida:

Una vez finalizada toda la simulación, el último paso para dar por finalizada la función del simulador consiste en cerrar el archivo de salida en el cual se ha ido almacenando todos los datos de la simulación que se han creído necesarios a lo largo de la misma. El archivo de salida, definido como variable global del archivo "mainwindow.cpp" se llena mediante la función "TXTOut()" mostrada a continuación:

```
QFile fOut("OutFileSimulation.txt");
QTextStream salidaTXT(&fOut);

void MainWindow::TXTOut(QStringList ListOut)

    for (int i = 0; i < ListOut.size(); ++i)

        salidaTXT << ListOut.at(i) << "\n";
```





# Capítulo 4:

## Conclusiones finales y líneas futuras

---

## Capítulo 4: Conclusiones finales y líneas futuras

En los capítulos anteriores se ha descrito por completo toda la información relativa al proyecto, tanto teóricamente como mostrando el desarrollo de la programación del simulador.

Además, se han enumerado las diferentes especificaciones marcadas para la realización del proyecto, las cuales marcaban el objetivo final del simulador: obtener una primera versión de un simulador de circuitos digitales para diagramas ASM++, integrado dentro de un interfaz de usuario, y desarrollado en código C++ empleando las librerías Qt, con el objetivo de poder ser integrable posteriormente en el compilador de diagramas ASM++.

Como se comprueba en el desarrollo de la programación del simulador, todas las especificaciones marcadas se cumplen correctamente:

- Se dispone de una primera versión de un simulador de circuitos digitales descritos con diagramas ASM++ fácilmente integrable en un compilador de diagramas ASM++.
- El simulador tiene como punto de partida un archivo con extensión .vdo, y como salida un archivo de texto .txt.
- El simulador es capaz de obtener el resultado de la simulación del diseño descrito en la figura 28.
- El simulador es capaz de interpretar circuitos descritos en lenguaje Verilog.
- El simulador dispone de un interfaz gráfico que permita al usuario interactuar con él.

A partir de ahora, una vez que se han determinado las características finales conseguidas tras la realización del trabajo fin de grado, pueden establecerse las diferentes líneas de actuación futuras, abriendo así la posibilidad de nuevos proyectos. A continuación se muestran algunas de las posibles líneas de actuación futuras que surgen a partir de este trabajo:



- **Integración del simulador en el compilador de diagramas ASM++:**

Tras la realización del proyecto se dispone de un simulador de circuitos digitales descritos mediante diagramas ASM++, diseñado de forma que sea fácil de implementar en el compilador de diagramas ASM++. Por lo tanto, realizar esa inclusión en el compilador de diagramas ASM++ se antoja como la acción más evidente a realizar una vez se dispone del simulador.

- **Ampliación de las funcionalidades del simulador:**

Pese a que las especificaciones del trabajo incluían gran cantidad de funcionalidades disponibles, el simulador tiene ciertas limitaciones al no incluir la totalidad de posibilidades que incluyen los diagramas ASM++. A continuación se indican algunos de las funcionalidades que pueden ampliarse en futuras versiones del simulador:

- Añadir al simulador la capacidad de leer y simular diseños con más de una página.
- Añadir al simulador la capacidad de interpretar los tipos de cajas que no se han incluido en esta primera versión.
- Añadir el análisis de las diferentes expresiones en Lenguaje Verilog que no se encuentran dentro de las especificaciones del trabajo, como variables o grupos de expresiones.
- Ampliar el número de palabras y símbolos detectables por la función GetToken.
- Añadir al simulador la capacidad de interpretar expresiones en lenguaje VHDL.

- **Creación de un entorno gráfico capaz de visualizar el resultado de la simulación:**

Esta primera versión del simulador es capaz de realizar la simulación de circuitos digitales diseñados mediante diagramas ASM++ y obtener el resultado de la simulación, mostrando y guardando el valor de las señales en cada ciclo de simulación. Sin embargo, esta forma de mostrar los datos de la simulación visualmente no es la más adecuada.

Uno de los siguientes proyectos que surgen a raíz de la realización del simulador es la capacidad de realizar un entorno gráfico capaz de mostrar los datos de la simulación de forma visualmente sencilla para el usuario.





# Capítulo 5:

# Bibliografía

---



## Capítulo 5: Bibliografía

En este capítulo se hace referencia a todos libros, páginas web y artículos que se han utilizado como apoyo para obtener la información necesaria durante el desarrollo del TFG, ordenados cronológicamente respecto a su publicación.

### 5.1. Artículos y seminarios:

- **C.R. Clare:**

“Designing Logic Systems Using State Machines” McGraw-Hill, New York, (1973)

- **IEEE Computer Society:**

“IEEE Standard for Verilog Hardware Description Language” (2006)

- **S. de Pablo, S. Cáceres, J.A. Cebrián, and M. Berrocal:**

"A proposal for ASM++ diagrams" IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2007, Cracovia, Polonia (Abril, 2007)

<http://www.epyme.uva.es/Docs/Public/Conferences/Ddecs2007a.pdf>

- **Juan González:**

“Introducción al Lenguaje de Descripción Hardware VHDL” (2007-2008)

<http://arantxa.ii.uam.es/~jgonzale/fco/curso07-08/download/seminarios-vhdl.pdf>

- **S. de Pablo, S. Cáceres, J.A. Cebrián y F. Sanz:**

“Diseño de circuitos digitales a nivel de registro empleando diagramas ASM++” (Marzo, 2010)

[http://www.scielo.cl/scielo.php?script=sci\\_arttext&pid=S0718-07642010000200012&lng=es&nrm=iso&tlng=es](http://www.scielo.cl/scielo.php?script=sci_arttext&pid=S0718-07642010000200012&lng=es&nrm=iso&tlng=es)

- **Paulino Ruiz de Clavijo Vázquez:**

“INTRODUCCIÓN A HDL VERILOG” (Noviembre, 2012)

<https://www.dte.us.es/Members/paulino/Verilog-Intro.pdf>



**5.2. Páginas web:** todas las páginas consultadas entre Septiembre de 2016 y Mayo de 2017

- **ASM++: a modern Algorithmic State Machine methodology for RTL designs**

<http://www.epyme.uva.es/asm++/index.php>

- **Programación en Lenguaje Verilog:**

[https://es.wikibooks.org/wiki/Programaci%C3%B3n\\_en\\_Verilog/Introducci%C3%B3n](https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Verilog/Introducci%C3%B3n)

- **Qt Documentation:**

<http://doc.qt.io/qt-5/classes.html>

- **Introducción al Lenguaje VHDL:**

<http://www.sc.ehu.es/acwarila/LDD/Teoria/VHDL.pdf>

**5.3. Proyectos finales de carrera:**

- **Marcos Alonso Rebollo:**

“Diseño de un entorno de desarrollo de alto y bajo nivel para un procesador de propósito general, programable en C, integrado en FPGA” Universidad de Valladolid, Uva (Diciembre, 2003)





# Capítulo 6:

# Anexos

---



## Capítulo 6: Anexos

En este capítulo se incluye un listado de los diferentes archivos adjuntos a la memoria del TFG, con un pequeño resumen indicativo de la función de cada uno de ellos.

### 6.1. Archivos para la ejecución del proyecto:

#### 6.1.1. aplicación.pro:

Archivo encargado de incluir en su interior la localización de la carpeta con el proyecto del simulador, los diferentes archivos .cpp y bibliotecas que contiene el mismo, los recursos y el resto de información requerida para poder ejecutar correctamente el simulador.

#### 6.1.2. doQmake.bat:

Ejecutable encargado de crear o compilar el proyecto.

#### 6.1.3. Archivos qrc de recursos:

Carpetas y archivos con los recursos requeridos en el diseño del proyecto.

#### 6.1.4. Archivo Test\_Multiplier.vdx y resultado de su simulación:

En los anexos se incluye el archivo .vdx empleado para realizar la simulación y que ha servido de ayuda a la hora de redactar la memoria, así como el resultado de su simulación en el archivo OutFileSimulation.txt.

#### 6.1.5. Bibliotecas .dll:

Se incluyen 4 librerías con extensión .dll necesarias a la hora de querer ejecutar el simulador.

#### 6.1.6. Ejecutable del simulador:

Se incluye en los anexos el ejecutable “miejecutable.exe” que permite acceder al interfaz gráfico del simulador.



## 6.2. Archivos con extensión .cpp y bibliotecas del simulador:

### 6.2.1. main.cpp:

Función main principal del simulador, encargada de inicializar el proyecto. Incluye la inicialización de los recursos, la clase que maneja el control de la aplicación interfaz de usuario (GUI) y la llamada a la ventana principal del interfaz.

### 6.2.2. mainwindow.cpp:

Archivo con extensión .cpp en el cual quedan definidos todos los comandos y funciones encargadas de crear la ventana de simulación, así como diversas funciones encargadas de realizar dicha simulación.

Todas las funciones presentes en este archivo se encuentran dentro de la clase MainWindow. A continuación se muestra un listado con las funciones presentes:

#### a) Funciones del interfaz de usuario:

- **MainWindow::MainWindow**  
Función principal del Main Window.
- **QGroupBox \*MainWindow::CreateEditGroup**  
Box encargado de crear el grupo de edición.
- **QGroupBox \*MainWindow::CreateOutGroup()**  
Box encargado de crear el grupo de salida.
- **QGroupBox \*MainWindow::CreateProcessGroup()**  
Box encargado de crear el grupo de procesamiento.
- **void MainWindow::closeEvent(QCloseEvent \*event)**  
Función que detecta si el usuario quiere cerrar la ventana.
- **void MainWindow::newFile()**  
Función que crea un nuevo archivo.
- **void MainWindow::open()**  
Función que abre un archivo ya existente.



- **bool MainWindow::save()**  
Función que guarda un archivo.
- **bool MainWindow::saveAs()**  
Función que guarda un archivo inexistente dándole un nombre.
- **void MainWindow::openRecentFile()**  
Función que permite abrir archivos recientemente cargados.
- **void MainWindow::run()**  
Función que ejecuta la aplicación.
- **void MainWindow::about()**  
Función que da información acerca de la aplicación.
- **void MainWindow::documentWasModified()**  
Función que comprueba si se ha modificado el documento.
- **void MainWindow::createActions()**  
Función que crea todas las acciones del interfaz.
- **void MainWindow::createMenus()**  
Función que crea la barra de menú principal.
- **void MainWindow::createToolBars()**  
Función que crea la barra de herramientas.
- **void MainWindow::createStatusBar()**  
Función que crea la barra de estado.
- **void MainWindow::readSettings()**  
Función que lee las propiedades y características de la aplicación.
- **void MainWindow::writeSettings()**  
Función que reescribe las propiedades y características de la aplicación.
- **bool MainWindow::maybeSave()**  
Función que comprueba si ha habido cambios en el archivo y muestra si se desea guardar antes de cerrar.



- **void MainWindow::loadFile(const QString &fileName)**  
Función que carga un archivo existente.
- **bool MainWindow::saveFile(const QString &fileName)**  
Función que guarda el archivo abierto.
- **void MainWindow::setCurrentFile(const QString &fileName)**  
Función que establece el archivo abierto actualmente.
- **void MainWindow::updateRecentFileActions()**  
Función que carga los archivos abiertos recientemente.
- **QString MainWindow::strippedName(const QString &fullFileName)**  
Función encargada de obtener la información del nombre del archivo.

#### b) Funciones de la simulación:

- **void MainWindow::simulacion(const QString &fileName)**  
Función con la que llamamos a las funciones del simulador tras apretar el botón RUN.
- **void MainWindow::TXTOut(QStringList ListOut)**  
Función para escribir en el archivo .txt de salida.
- **void MainWindow::ErroresYWarnings(QString ErrWar, int tipo)**  
Función que recibe un String con un error o un warning y los almacena en class Fallos. Tipo 0 = lectura, tipo 1 = Error, tipo 2 = Warning) .
- **void LimpiarMemoria()**  
Función encargada de limpiar la memoria totalmente al iniciar la simulación de un archivo nuevo.
- **void MainWindow::LecturaArchivo(const QString &FILE)**  
Función que recibe un archivo de texto, lo abre, lo recorre interpretando todo su contenido y almacena los valores.



- **void MainWindow::MostrarDatosGuardados()**  
Función encargada de mostrar todos los datos guardados durante la lectura del archivo.
- **void MainWindow::OrdenandoBoxes()**  
Función encargada de recorrer el vector TYPE de la clase Box BOX y guardar en un QMultiHash las posiciones de cada tipo.
- **void MainWindow::BoxCabecera()**  
Función encargada de detectar que ID's pertenecen a padres del diseño y cuáles no. Los junta con los de las cabeceras HEADER y los guarda en la clase ORDENBOX.
- **void MainWindow::SimulacionDeclarativa()**  
Esta función es la encargada de simular el circuito digital de forma declarativa. Partiendo desde las cajas padres, recorre el diagrama por orden NEXT -> ID de caja en caja buscando todas las señales. Solo recorre la parte de diseño del circuito, no del test bench.
- **void MainWindow::MostrarSignals()**  
Función encargada de mostrar todas las señales guardadas durante la simulación declarativa.
- **void MainWindow::NextBox(int NextPadre, int &ItHijo)**  
Función encargada de recorrer la información almacenada. Recibe el NEXT de un padre y devuelve la posición en memoria del hijo.
- **void MainWindow::SaveSignals(int IteradorMemoria, int modo)**  
Función encargada de guardar las señales en la simulación declarativa. Dependiendo del modo de búsqueda que le llegue diferencia entre los diferentes tipos de señales.  
  
En modo 1 = Ports busca inputs y outputs. Si hay otro tipo muestra error.  
En modo 2 = Code busca wire e inouts. Si hay otro tipo muestra error.  
En modo 3 = Async-Sync busca entre las señales que ya hay las que encuentra y decide del tipo que son.
- **void MainWindow::SimulacionOperativa()**  
Esta función es la encargada de simular el circuito digital en forma operativa.



- 1- Llama a la función que recorre cada padre guardando sus estados.
- 2- Recorre todo lo que está entre las cabeceras y el primer estado interpretando la información.
- 3- Va al primer estado de cada cabecera, comprueba la información presente al inicio del ciclo, recorre el diagrama hasta encontrar el siguiente estado, verifica si hay algún cambio en las señales y si no hay pasa al siguiente ciclo.

- **void MainWindow::BoxState()**

Función encargada de recorrer cada padre en busca de todos los estados que tiene y los guarda en la matriz de la clase ORDENBOX.

- **void MainWindow::MostrarDatosInicioSimOperativa()**

Función que muestra los datos que se tienen al inicio de la simulación operativa.

- **void MainWindow::PresimulacionOperativa()**

Función encargada de recorrer todas las cajas presentes entre el inicio de cabeceras y el primer estado.

- **void MainWindow::SimulacionOperativaON()**

Función encargada de ir recorriendo todos los esquemas una vez realizada la presimulación operativa para simular ciclo a ciclo el comportamiento del circuito.

- **void MainWindow::MostrarEventos()**

Función encargada de mostrar todos los eventos presentes en la simulación.

- **void MainWindow::ValoresPorDefecto()**

Función encargada de guardar los valores por defecto de las señales.

- **void MainWindow::AnalizadorEventos()**

Función encargada de comprobar si alguna condición que ejecuta eventos esta activa (como la señal reset) y si es así ejecuta los cambios marcados por ese evento modificando el valor de las señales.



- **void MainWindow::GuardarSimulacion(int CYCLE)**  
Función encargada de guardar al final de cada ciclo todos los valores de la simulación.
- **void MainWindow::GuardarActual(int CYCLE)**  
Función copia de la anterior, pero encargada de guardar los valores de las señales durante la simulación operativa (señales asíncronas).
- **void MainWindow::GuardarDespues(int CYCLE)**  
Copia de la anterior pero esta se encarga de guardar los valores de inicio del siguiente ciclo de la simulación operativa.
- **void MainWindow::MostrarSignalPorCicloSimulacion()**  
Función encargada de mostrar el valor de todas las señales en el ciclo actual.
- **void MainWindow::MostrarVerificacionesARealizar()**  
Función encargada de mostrar todas las verificaciones que hay que realizar.
- **void MainWindow::RealizarVerificaciones()**  
Función encargada de realizar las verificaciones.

**c) Inicialización de las matrices empleadas para almacenar datos:**

- **void MatrizTexto()**  
Inicializa la matriz que guarda los textos de las cajas identificadas como "TEXT".
- **void MatrizTextoUp()**  
Inicializa la matriz que guarda los textos de las cajas identificadas como "TEXT UP".
- **void MatrizTextoDown()**  
Inicializa la matriz que guarda los textos de las cajas identificadas como "TEXT DOWN".



- **void MatrizIDStatesPadres()**  
Inicializa la matriz que guarda los ID's de los estados de los diferentes padres.
- **void MatrizITStatesPadres()**  
Inicializa la matriz que guarda las posiciones en memoria de los estados de los diferentes padres.
- **void MatrizDatosSimulacionFinal(int NumSignals)**  
Función que inicializa la matriz para guardar todos los valores de las señales a lo largo de los ciclos de simulación.

**d) Funciones del analizador de expresiones:**

Todas estas funciones ya están definidas en el Capítulo 3, y hacen referencia a cada uno de los diagramas presentes en la figura 40 (página 83).

- **void MainWindow::ExprPreviaToken(QString &Frase)**
- **void MainWindow::ExprToken(QString &frase)**
- **void MainWindow::ExprAToken(QString &frase)**
- **void MainWindow::ExprBToken(QString &frase)**
- **void MainWindow::ExprCToken(QString &frase)**
- **void MainWindow::ExprDToken(QString &frase)**
- **void MainWindow::ExprEToken(QString &frase)**
- **void MainWindow::ExprFToken(QString &frase)**
- **void MainWindow::ExprGToken(QString &frase)**
- **void MainWindow::ConstantToken(QString &frase)**
- **void MainWindow::SignalToken(QString &frase)**
- **void MainWindow::GroupToken(QString &frase)**

**6.2.3. simulador.cpp:**

Archivo con extensión .cpp en el cual se crean las funciones encargadas de realizar operaciones durante la simulación del circuito digital descrito en diagramas ASM++.

**a) Funciones para resolver operaciones del simulador:**

Las siguientes funciones, como indica el propio nombre de cada una, se encargan de realizar operaciones sobre los datos como sumar, restar, multiplicar... así como operaciones lógicas con los bits y de convertir las unidades de los números enteros.



- void Suma(int &constante1, int &constante2, int &SUMA)
- void Resta(int &constante1, int &constante2, int &RESTA)
- void Multiplicacion (int &constante1, int &constante2, int &MULTIPLICACION)
- void Division(int &constante1, int &constante2, int &DIVISION)
- void Porcentaje(int &constante1, int &constante2, int &PORCENTAJE)
- void Desplazamiento(int &Valor, int &Cantidad, int Tipo, int &Salida)
- void Comparacion(int &Primero, int &Segundo, int Tipo, bool &Salida)
- void LogicaBit(int &Primero, int &Segundo, int Tipo, int &Salida)
- void LogicaExpr(bool &Primero, bool &Segundo, int Tipo, bool &Salida)
- void Condicional(bool &Condicion, int &Cierto, int &Falso, int &Salida)
- void Exponencia(int &Expresion1, int &Expresion2, int &Salida)
- int ConvertirUnidades(int TipoConversion, QString &Numero)

#### b) Función para detectar partes de expresiones:

- int GetToken (QString &Buffer, bool removeIt)

Como ya se ha explicado en el Capítulo 3 (apartado 3.3.2.b) esta función se encarga de detectar cada carácter presente en una cadena de caracteres.

#### 6.2.4. shared.h:

Biblioteca en la cual se crea la lista enumerada de tokens empleados durante todo el simulador para detectar los diferentes caracteres presente dentro de las cadenas de caracteres deseadas a través de la función GetToken.

#### 6.2.5. GlobalDefs.h:

Biblioteca en la cual se definen los nombres de las diferentes clases empleadas durante la simulación para almacenar la información.

#### 6.2.6. mainwindow.h:

Biblioteca correspondiente al archivo mainwindow.cpp, encargada de incluir las cabeceras de sus funciones para poder disponer de ellas en las diferentes funciones en la que esté incluida, así como de definir la clase MainWindow.



### 6.2.7. simulador.h

Biblioteca encargada incluir las cabeceras de las funciones de simulador.cpp para disponer de ellas en cualquier archivo en la que se incluya, así como de definir las diferentes clases empleadas por el simulador. A continuación se muestra un listado de todas ellas:

- Clase para guardar las diferentes señales:

class Signal

- Clase para guardar todo lo de un modulo BOX

class Box

- Clase para guardar todo lo de un módulo PAGES

class Pages

- Clase para ordenar los BOXES, y para guardar variables relativas a este

class OrdenBox

- Clase para guardar todo lo relativo a la simulación de los test bench

class TestBench

- Clase para guardar todas las verificaciones que hay que realizar

class Verificaciones

- Clase para guardar todos los datos de la simulación

class DatosSimulacion

- Clase para guardar todos los warning y errores que aparecen en el programa

class Fallos