UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Carrera de Ingeniero Informático

APLICACIÓN DEL FORMALISMO DEVS AL MODELADO DE CIRCUITOS DIGITALES

VICENTE REIG MOLLÁ

Dirigido por: ALFONSO URQUIA MORALEDA

Curso: 2013 / 2014 (marzo 2014)

APLICACIÓN DEL FORMALISMO DEVS AL MODELADO DE CIRCUITOS DIGITALES

Proyecto de Fin de Carrera de modalidad *oferta específica (tipo B)*

Realizado por: VICENTE REIG MOLLÁ (firma)

Dirigido por: ALFONSO URQUIA MORALEDA (firma)

Tribunal calificador:

Presidente: D. /Da..... (firma)

Secretario: D. /Da..... (firma)

Vocal: D. /Da..... (firma)

Fecha de lectura y defensa: .....

Calificación: .....

Resumen

En este proyecto se ha realizado una implementación software de un circuito electrónico consistente en una CPU conectada a una memoria. Para el desarrollo de la implementación, nos hemos basado en la descripción que se realiza de dicho circuito en el texto "VHDL Programming by Example" de Douglas L. Perry. En dicho texto se describe una implementación elemental y muy básica de dicho circuito.

Se han realizado y simulado dos modelos diferentes del circuito: uno mediante el formalismo DEVS y la herramienta PowerDEVS, y el otro mediante VHDL y la herramienta ModelSim PE Student Edition. Esto nos permite describir diferencias entre ambos, a la vez que nos permite validar los resultados de las distintas implementaciones. La validación se realiza mediante una simple comparación de los resultados obtenidos de las simulaciones de ambas implementaciones, con los resultados teóricos esperados.

Mediante la comparación de los resultados obtenidos, estableceremos la validez de los resultados obtenidos de ambas simulaciones, pues a implementaciones equivalentes, se deben obtener resultados equivalentes en las simulaciones. Para poder establecer dichas comparaciones se han establecido cuatro bancos de pruebas, sobre la simulación de los cuales se establecerá la validez de las implementaciones, así como las comparaciones y mejoras establecidas.

Listas de palabras claves

Circuito digital; simulación; VHDL; DEVS; PowerDEVS; ModelSim PE Student Edition.

Abstract

This project was for the implementation of an electronic circuit consisting of a CPU connected to a memory. The development of the implementation was based on a description of the circuit contained in “VHDL Programming by Example” by Douglas L. Perry, which describes the simplified, basic implementation of such a circuit.

There have been two different models and simulated the circuit: one using DEVS formalism and PowerDEVS tool, and the other using VHDL and ModelSim PE Student Edition tool.

By comparing the results obtained, the validity of the results will be established for both simulations as similar implementations should yield similar simulated results. In order to establish the comparisons and verify the implementations, four test beds were assembled.

Once the implementations were validated the differences between them could be established, allowing improvements to be introduced to both systems.

Key words

Electronic circuit; simulationVHDL; DEVS; PowerDEVS; ModelSim PE Student Edition.

Tabla de contenido

Lista de figuras:

Página

Lista de código:

Página

Lista de tablas:

Página

1

INTRODUCCIÓN, OBJETIVOS Y ESTRUCTURA

1.1 INTRODUCCIÓN

El presente documento se inscribe en el ámbito de los estudios de Ingeniería Informática de la Universidad Nacional de Educación a Distancia y representa el colofón de dichos estudios, al suponer los últimos créditos necesarios para la obtención del título, de Ingeniero Informático, tendente a desaparecer por la implantación del plan Bolonia.

Este proyecto que solo tiene carácter académico y no se presenta como un proyecto para desarrollo comercial de software, aunque intenta reproducir las características típicas de este tipo de proyectos, presenta características diferenciadas de los proyectos profesionales de gran escala.

Las características del mismo, viene limitada forzosa y necesariamente por el tamaño del proyecto, que se realiza de manera unipersonal (por necesidades académicas), por tanto los objetivos del mismo, no son necesariamente los habituales en los proyectos de Ingeniería Informática.

En esta sección meramente introductoria, vamos a tratar de introducir las herramientas que se van a utilizar y el propósito de las mismas. Por tanto vamos a tratar de introducir el lenguaje VHDL, además de un entorno de simulación de

dicho lenguaje, en nuestro caso hemos elegido ModelSim PE Student Edition. A continuación, describiremos el formalismo DEVS y un entorno para su desarrollo y simulación, que en nuestro caso será PowerDEVS.

1.1.1 El lenguaje VHDL

Lenguaje de programación que pertenece al grupo de lenguajes HDL (Hardware Description Language), al igual que otros como Verilog y Abel. El propósito de estos lenguajes, es su utilización como lenguajes de descripción de hardware digital y analógico en circuitos electrónicos.

Promovido en 1981 por el Departamento de Defensa de los Estados Unidos, su desarrollo fue paralelo con el lenguaje ADA, con el que comparte ciertas características; al igual que ocurre con Verilog que comparte características de lenguaje C. Este lenguaje, surge ante la necesidad de disponer de un idioma con una amplia capacidad descriptiva, capaz de funcionar con cualquier simulador e independiente de la tecnología o metodología del diseño.

En 1987 se publica como norma IEEE, siendo a partir de entonces un estándar, que se revisara por primera vez en 1994, dando lugar a la versión VHDL 1076-1993.

Este lenguaje adquiere una importancia en el desarrollo y diseño de hardware, solo comparable con el lenguaje Verilog, el cual se conforma como alternativa al lenguaje VHDL.

Dentro del diseño lógico con VHDL, existen varias formas de describir y simular un circuito lógico, lo cual vendrá determinadas por el tipo de secuencia

utilizado. Los modelos básicos de diseño son tres, dos dentro del dominio del comportamiento que son: “modelo Algorítmico” y “modelo Flujos de Datos”. Mientras que en el domino estructural, solo disponemos de un modelo de comportamiento, que es el “modelo Estructural”.

Destacar que en el desarrollo de nuestro PFC, se ha utilizado el modelo Algorítmico para el comportamiento y el modelo Estructural. Entrar en una descripción de los diferentes modelos, escapa de los objetivos de este PFC y resultaría demasiado prolijo (necesidad de ejemplos, esquemas y diagramas…), pero esto no es óbice, para realizar una explicación somera de ambos modelos.

El modelo Algorítmico básicamente, se encarga de realizar una simulación de la relación de entradas con las correspondientes salidas, de un componente hardware. Realizando una descripción de su funcionamiento, sin entrar en la estructura interna del componente. Esto se realiza, mediante una descripción de programación imperativa, que nada tiene que ver con la realidad física del componente hardware. Por contrapartida, dentro del mismo dominio del comportamiento, el modelo de Flujo de Datos, realiza a su vez una simulación del funcionamiento del mismo, con la diferencia respecto al modelo Algorítmico, que este trata de mantener una estructura interna más próxima al del elemento hardware real.

En lo que afecta al modelo Estructural, este consiste en, realizar una representación biunívoca de los elementos software, descritos en el dominio de comportamiento (se podría decir que este modelo realiza la conexión física de los elementos diseñados).

1.1.2 ModelSim

Ni que decir cabe, que un lenguaje de diseño como es VHDL no habría experimentado un notable desarrollo y popularidad, sin la existencia de entornos de desarrollo y simulación (IDE's, Simuladores, etc.) de los diseños realizados por los ingenieros.

Cabe decir, que este lenguaje es muy potente y no todos los entornos soportan todas las posibles funcionalidades y usos del mismo, esto también ha derivado en una cierta especialización de los entornos de desarrollo y simulación. Orientándolos a usos más específicos, dentro de las distintas funcionalidades del lenguaje. De los muchos entornos de desarrollo que podríamos haber utilizado: Bluepc [Blue Pacific, 2013]; Entorno Aliance de Altera [ModelSim Altera Started Edition, 2013]; Xiling [ISE, 2013], nos hemos decantado por la utilización de ModelSim PE Student Edition de la empresa MentorGrafics en su versión 10.2. A continuación, pasamos a realizar una reseña del entorno, así como a describir brevemente el proceso de instalación del mismo.

La versión que hemos utilizado en la ejecución del presente proyecto, es la ModelSim PE Student Edition 10.2. Si bien, con el entorno de PowerDEVS, hemos tratado de profundizar en la explicación de la herramienta (Capítulo 2), en el caso de ModelSim nos parece una acción innecesaria, pues la misma herramienta la incorpora en su directorio Modeltech_pe_edu_10.2\docs. Esta documentación, nos parece más que suficiente para la utilización del entorno, resultando de gran utilidad, ademas que por mucho que nos esforzáramos, no podríamos superar en exactitud y rigurosidad; esto sumado a la gran cantidad de tutoriales que existen

en la red sobre este entorno en Castellano, nos induce a pensar que resulta superflua la explicación del funcionamiento del entorno.

En cambio, puesto que en el apartado de simulación, se explica los pasos que debemos realizar para proceder a la simulación de nuestro proyecto y que son válidos para cualquier proyecto. Nos parece oportuno, realizar una breve explicación del proceso de instalación del entorno, que nos puede ser de ayuda para poder comprobar cuantos extremos necesitemos del proyecto. Pasamos por tanto a realizar la descripción breve del proceso de instalación del entorno.

Comenzaremos presentando la página, desde donde se iniciará la descarga del entorno de simulación, que es la que mostramos en la Figura 1.1.

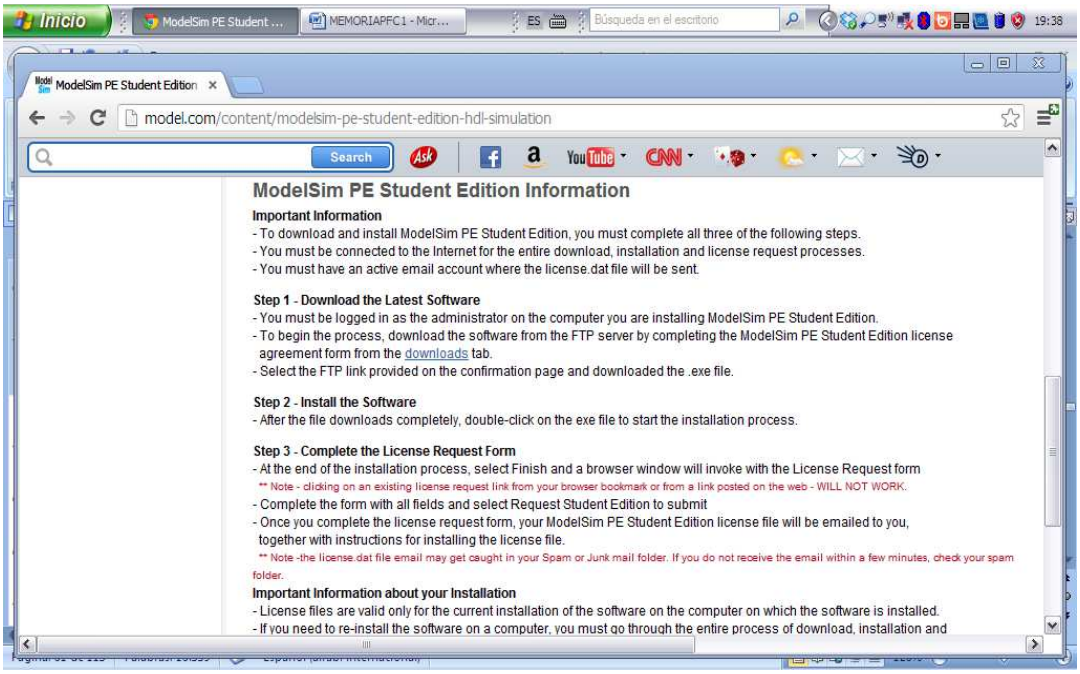


Figura 1.1: Pagina inicio para la descarga de ModelSim

En esta pantalla, deberemos seleccionar “downloads” tras lo cual llegaremos a la pantalla mostrada en la Figura 1.2.

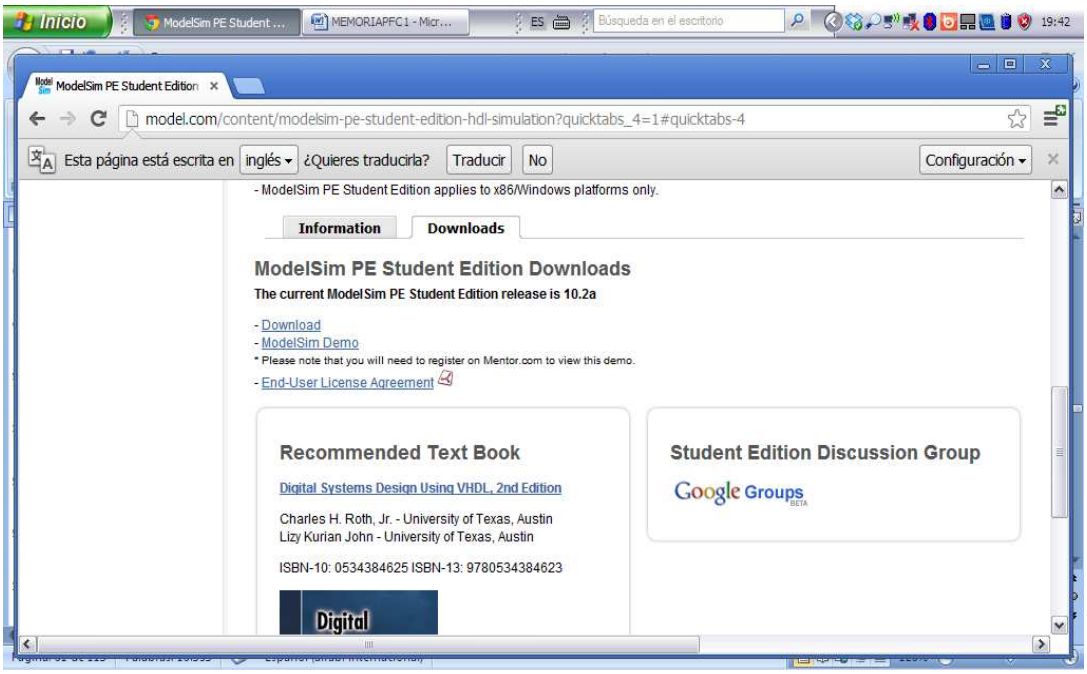


Figura 1.2: Segundo paso para la descarga de ModelSim

Volveremos a seleccionar “downloads” y nos llevará a la pantalla siguiente que se muestra en la Figura 1.3.

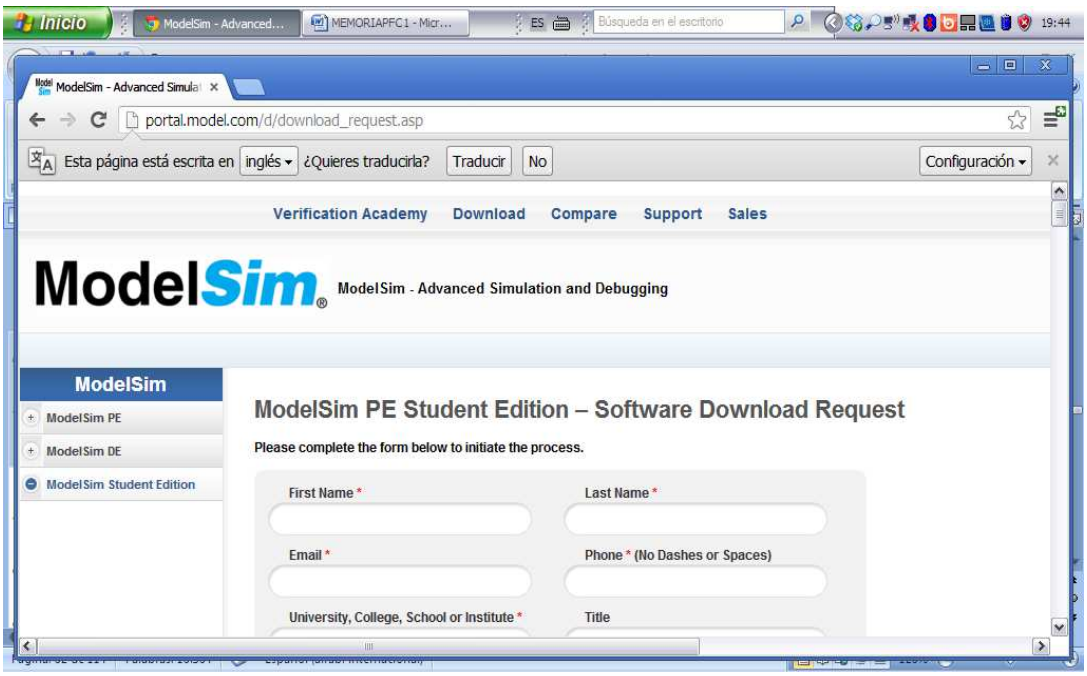


Figura 1.3: Tercer paso para la descarga de ModelSim; ingreso de datos personales

La aplicación nos exige, como es habitual en este tipo de aplicaciones, que aceptemos los términos del contrato.

Una vez ingresados nuestros datos deberemos ir al final de la pantalla para pulsar sobre el botón “Request Download”, tal y como se nos muestra en la Figura 1.4.

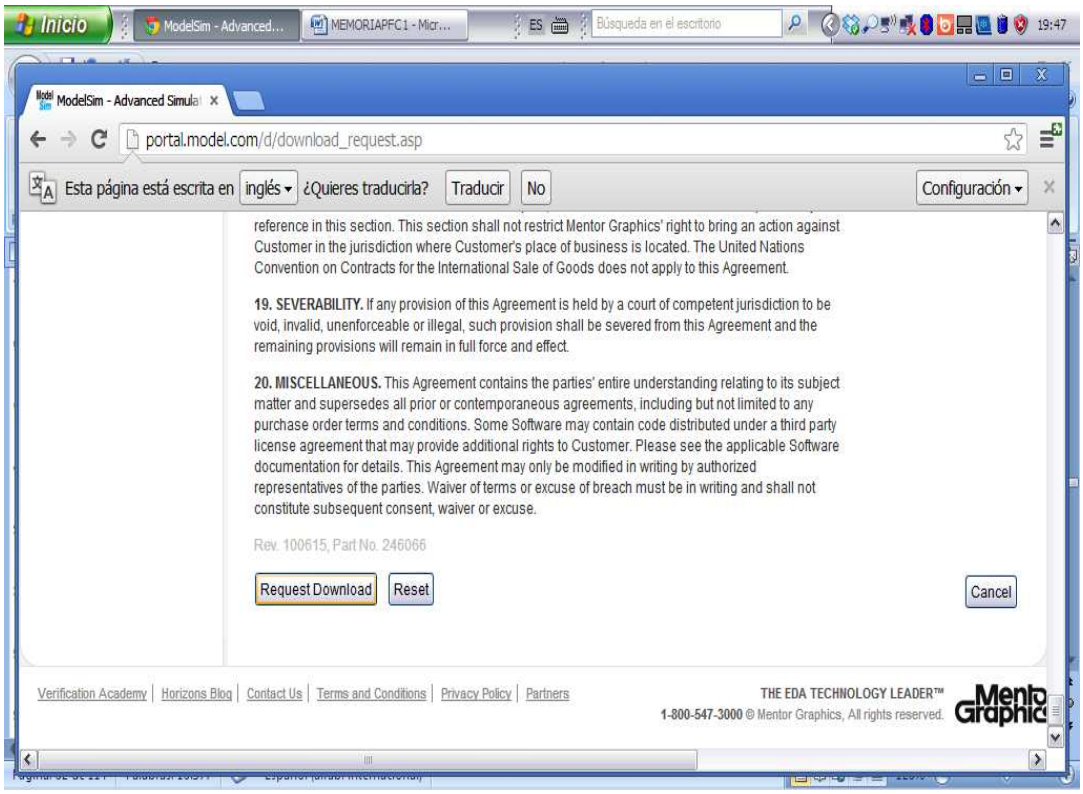


Figura 1.4: Cuarto paso para la descarga de ModelSim

Tras pulsar sobre el botón, nos lleva a otra pantalla, de la cual no entendemos bien su cometido, pero en la que aparece un enlace, que es el que nos llevara a la pantalla de descarga propiamente dicha, tal y como podemos ver en la imagen de la Figura 1.5.

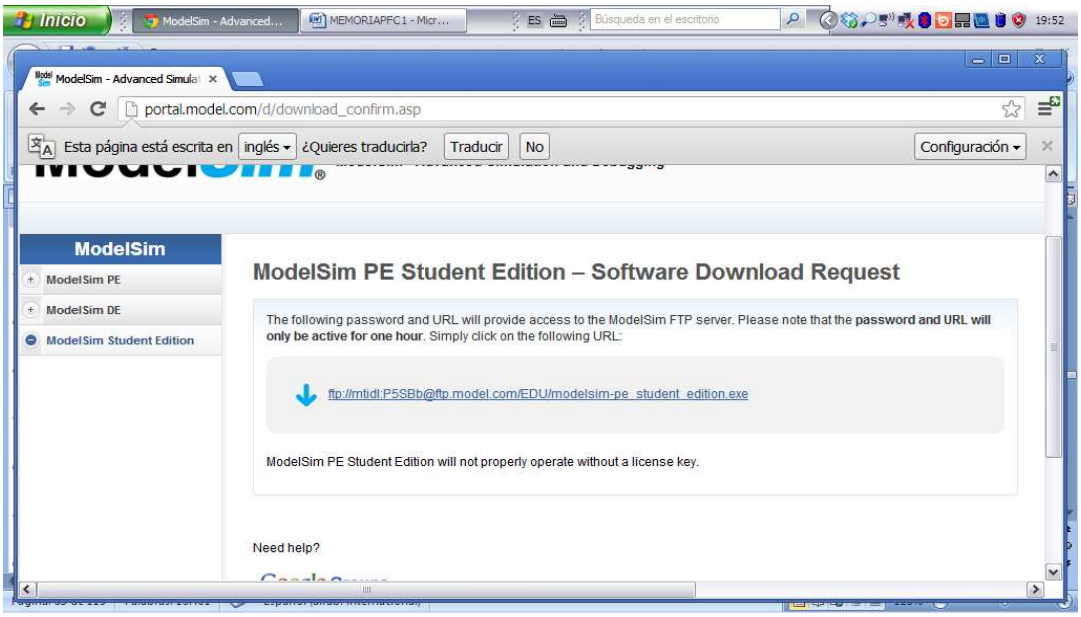


Figura 1.5: Quinto paso para la descarga de ModelSim

Tras pulsar sobre el enlace, se nos descargara el programa de instalación, apareciendo la pantalla que se muestra en la Figura 1.6.

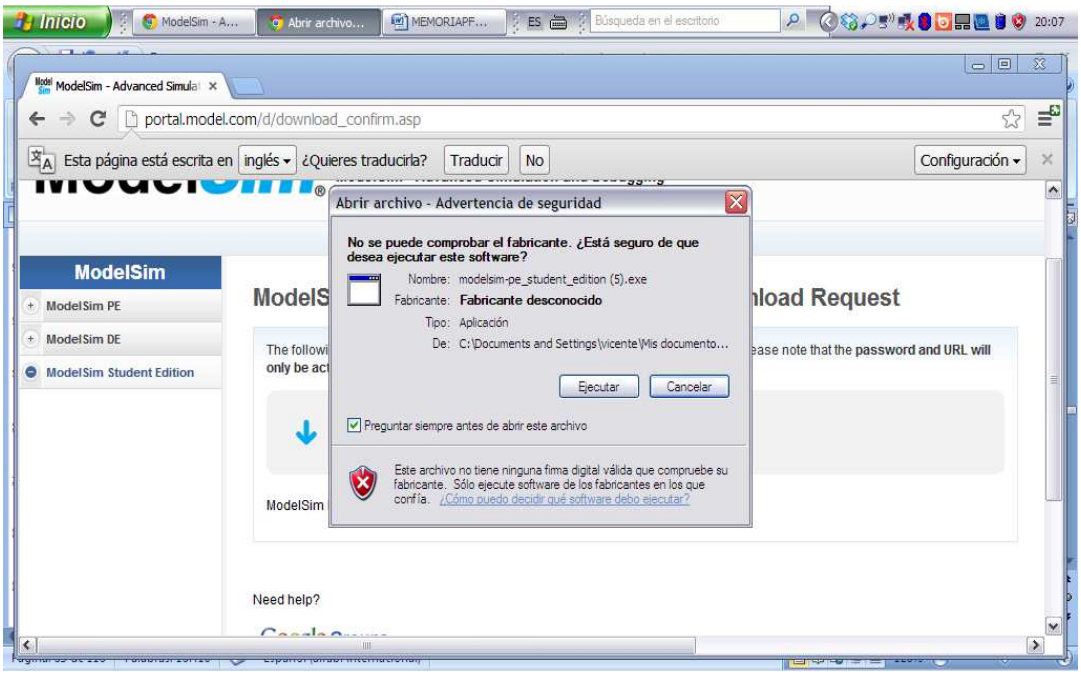


Figura 1.6: Sexto paso para la descarga de ModelSim; ejecutar archivo descarga

Pulsaremos sobre “Ejecutar” y nos llevará al menú de instalación, que se muestra en la Figura 1.7.

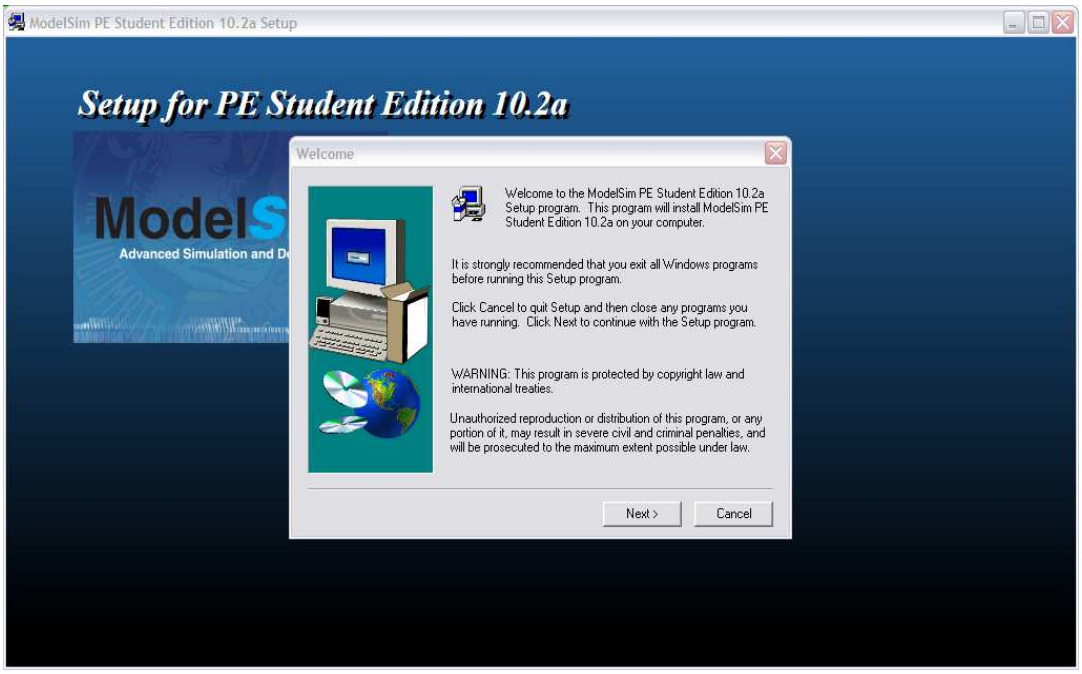


Figura 1.7: Séptimo paso para la descarga de ModelSim; seguimiento de la guía de instalación

A partir de aquí, lo que se nos presenta, es un típico menú de instalación, en el que avanzaremos aceptando las condiciones que se nos pide. No siendo este menú diferente de tantos otros, que nos podemos encontrar al realizar descargas de programas en la red, procederemos a explicar el paso final, pues es el único que puede resultar diferente de otros menús de instalación.

Una vez finalizada la instalación, recibiremos un mensaje en nuestro correo con un archivo y las instrucciones para su instalación, pues de no hacerlo de la manera correcta, el entorno no funcionará. El correo que recibiremos será similar al mostrado en la Figura 1.8.

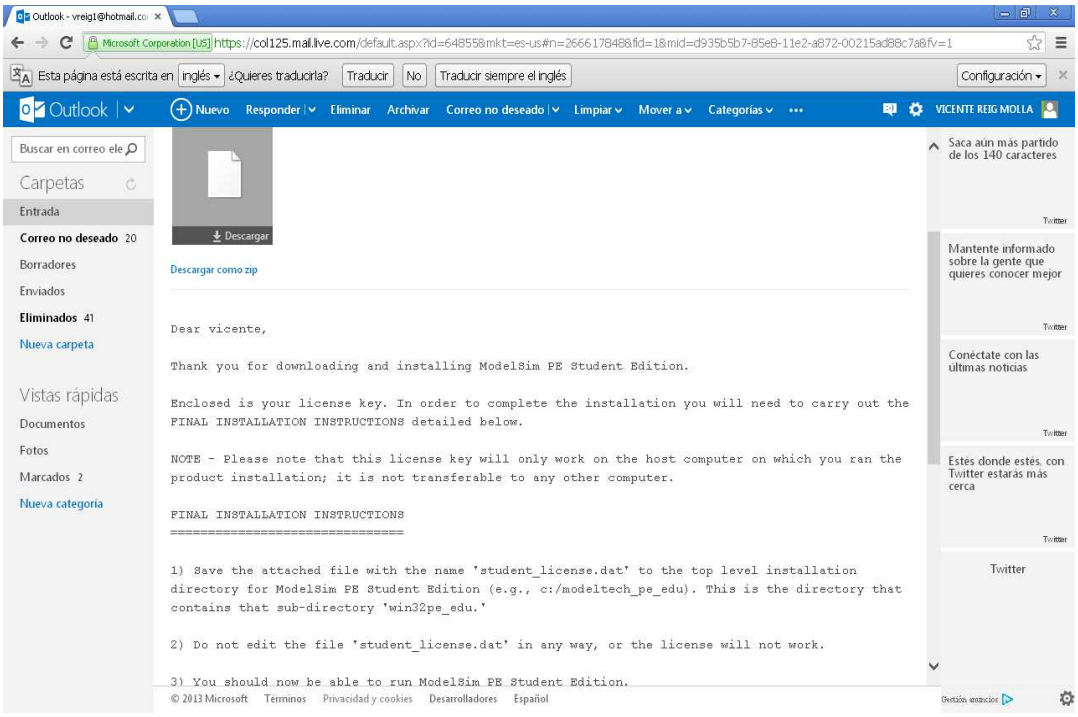


Figura 1.8: Correo recibido de ModelSim con el archivo para funcionamiento del entorno

Como podemos apreciar en la imagen, en el correo se nos envía un archivo adjunto y unas instrucciones de instalación de dicho archivo. En el caso que nos ocupa y si no dice lo contrario, la instalación de dicho archivo solo consistirá en colocarlo en el directorio que se nos indica tal y como se muestra en la Figura 1.9.

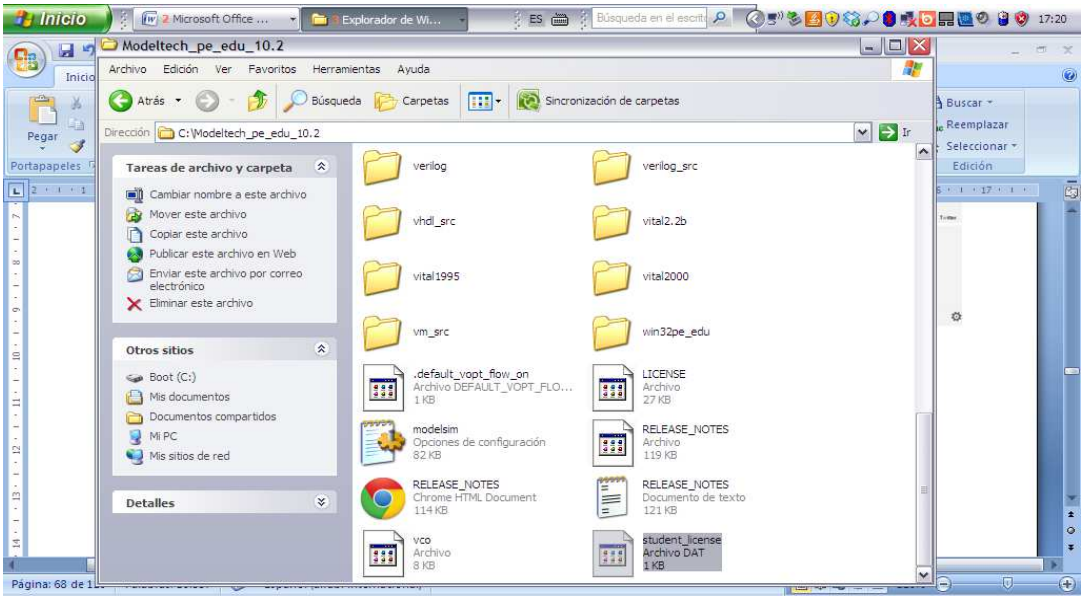


Figura 1.9: Instalación del archivo recibido en el correo de ModelSim para funcionamiento del entorno

En el mismo correo, se nos indica donde podemos conseguir tutoriales de ModelSim.

El cómo crear nuestro proyecto y proceder a su simulación, se explica en el Capítulo 3 y 4.

1.1.3 DEVS Clásico

Hemos incluido el formalismo DEVS, como la base de la herramienta PowerDEVS, pues DEVS representa la base formal, mientras que PowerDEVS se corresponde con la implementación algorítmica de dicho formalismo.

Por tanto, realizamos una introducción del formalismo DEVS, como base para comprender la implementación algorítmica que vamos a realizar, mediante el entorno PowerDEVS; éste ha sido el elegido para la implementación y simulación del modelo. Pasaremos a continuación a realizar una descripción breve del formalismo, para lo cual hemos utilizado e incluso trascrito literalmente parte del punto 1.1 del PFC (Ibáñez, 2010) así como el texto (Urquia, 2008); con los cuales hemos confeccionado esta breve descripción remitiendo al texto de (Zeigler, Praehofer y Kim, 2000) para aclaraciones y ampliación de conocimientos.

DEVS es un formalismo universal, que nos permite representar cualquier sistema, con tal que satisfaga la condición, que en cualquier intervalo acotado de tiempo, el sistema experimente un número finito de cambios (eventos). El sistema se describe como un conjunto compuesto de

una base de tiempo, entradas, salidas y funciones, para calcular los siguientes estados y salidas. El formalismo define cómo generar nuevos valores para las variables, y los momentos en los que estos cambios se producen.

Un modelo DEVS, se construye en base a un conjunto de modelos básicos, llamados atómicos, que se combinan para formar modelos acoplados. Los modelos atómicos son objetos independientes modulares, con puertos de entrada y puertos de salida, variables de estado y parámetros, funciones de transición externa, de transición interna, de salida y avance de tiempo.

Si el modelo que necesitamos modelar, se compone de varias entradas y salidas, con la condición de que no se produzcan varios eventos de entrada o de salidas simultáneos en una misma entrada o salida, podremos utilizar el modelo MIMO. La descripción del modelo MIMO es muy similar a la del modelo SISO, con la diferencia que hay que integrar en el modelo las múltiples entradas o salidas. Realizamos a continuación la descripción formal del modelo MIMO.

$$DEVS = < X, S, Y, \delta int, \delta ext, \lambda, ta >$$

Donde:

$\mathbf{X= \{(p,v) \mid p \in InPorts, v \in X_p)\}}$	Conjunto de entrada compuesto por todas las posibles parejas (p, v), donde p es el nombre de un puerto de entrada y v es un

	posible dato recibido en dicho puerto.
S	Conjunto de posibles estados secuenciales.
$Y= \{(p,v) \mid p \in OutPorts, v \in X_p)\}$	El conjunto de salida compuesto por todas las posibles parejas (p,v), donde p es el nombre de un puerto de salida y v es un posible dato enviado por dicho puerto.
δ**int**: **S → S**	Función de transición interna. Describe el estado que sucederá al estado actual.
δ**ext**: **Q x X → S**	Función de transición externa. Describe la respuesta del modelo a un evento de entrada, mostrando cuál será el nuevo estado, teniendo en cuenta la entrada producida, el estado actual y el tiempo que ha transcurrido desde la última transición de estado. $Q = \{(s,e) \mid s \in S, e \in [0, t_a(s)]\}$.
λ: **S → Y**	Función de salida, que admite como argumento un estado del modelo y devuelve un elemento (p, v) del conjunto Y. También puede devolver el conjunto vacío.

ta: $S \rightarrow R^{+}_{0,\infty}$	Función de avance de tiempo. Muestra el tiempo que permanecerá el modelo en el estado s, en ausencia de eventos de entrada.

Hay dos variables que se emplean en todos los modelos DEVS para representar intervalos de tiempo: la variable **e**, que almacena el tiempo transcurrido desde la anterior transición de estado, ya sea interna o externa, y la variable **σ**, que almacena el tiempo que resta hasta el instante en que está planificada la siguiente transición interna.

Para comprender el mecanismo de funcionamiento de un modelo DEVS, es importante resaltar que durante la transición externa no se produce ningún evento de salida. Cuando tiene que realizarse una transición interna, se produce un evento de salida justo antes de efectuarse la transición interna.

En la Figura 1.10 se muestra un ejemplo del funcionamiento de un modelo DEVS con la secuencia de eventos y sus trayectorias:

Figura 1.10: Ejemplo de funcionamiento de un modelo DEVS; Entradas

En la Figura 1.11 se nos muestra los estados del modelo

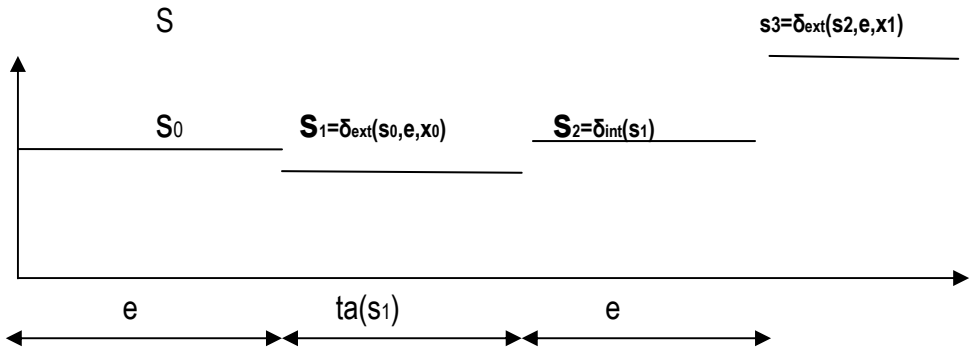


Figura 1.11: Ejemplo de funcionamiento de un modelo DEVS; estados

En la Figura 1.12 se nos muestra el grafico de salida:

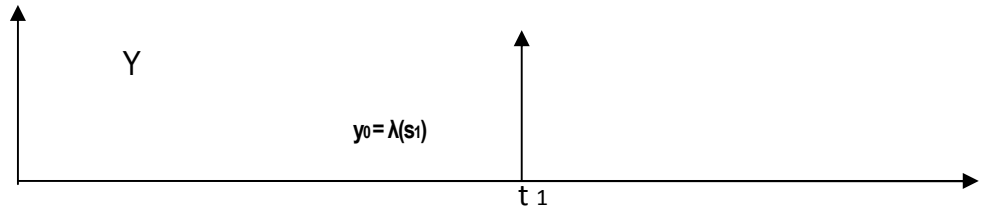


Figura 1.12: Ejemplo de funcionamiento de un modelo DEVS. Salidas

Los eventos de entrada ocurren en los instantes **t0** y **t2**. En **t1** sucede un evento interno. En cada uno de los tres eventos se produce un cambio en el estado del sistema. La salida **y0** se produce justo antes de que se produzca el evento interno.

El sistema se encuentra inicialmente en el estado **s0**, el cual tiene planificada una transición interna para cuando hayan transcurrido $\mathbf{t_a}$ (**s0**) unidades de tiempo. Antes de que se alcance ese momento, se produce un evento de entrada en el instante **t0**. Como **t0** < $\mathbf{t_a}$ (**s0**), la transición interna prevista para el instante $\mathbf{t_a}$(**s0**) no llega a producirse.

Como resultado del evento de entrada en **t0**, se produce un cambio de estado. El nuevo estado se calcula mediante la función de transición externa: **s1** = δext (**s0, e, x0**), donde **s0** es el estado anterior, **e** es el tiempo que el sistema ha permanecido en dicho estado y **x0** es el valor del evento de entrada.

Ahora se planifica una transición interna para dentro de $\mathbf{t_a}$ (**s1**) unidades de tiempo, es decir, para el instante **t0** + $\mathbf{t_a}$ (**s1**). Como no se produce ningún evento externo antes de ese instante, la transición interna ocurre cuando estaba planificada. En ese instante se genera la salida, que se calcula con la función de salida, a partir del valor del estado: **y0** = **λ** (**s1**). A continuación se produce la transición de estado. El nuevo estado se calcula mediante la función de transición interna, pasándole como parámetro el estado anterior:

s2 = δint (s1).

En este nuevo estado, se planifica la siguiente transición interna para el instante **t1** + $\mathbf{t_a}$(**s2**), pero no llega a producirse porque lo impide la aparición de un nuevo evento externo en el instante **t2** < **t1** + $\mathbf{t_a}$(**s2**).

El nuevo estado se calcula con la función de transición externa, se planifica la próxima transición interna, y así sucesivamente.

Con la parte descrita del formalismo ya estamos en condiciones de entender la descripción de todos los componentes individuales que conforman el proyecto. Con esto poco podríamos hacer, pues el presente PFC tiene que describir el funcionamiento conjunto de todos los elemento funcionando coordinadamente,

para este menester, usaremos la descripción del modelo DEVS Acoplados Modularmente; por tanto vamos a realizar una presentación formal del método.

Un modelo acoplado describe los modelos que lo componen y cómo conectarlos entre sí, ya sean atómicos o acoplados, para formar un nuevo modelo. Este modelo puede ser empleado como componente, permitiendo una construcción jerárquica. La conexión de los modelos se realiza mediante los puertos de entrada y de salida. La especificación del modelo se realiza con la tupla siguiente:

N= ‹X, Y, D, {Md|dϵ D}, EIC, EOC, IC, select›

Donde:

- X: Conjunto de entrada.
- Y: Conjunto de salida.
- X e Y conforman la interfaz del modelo.
- D y Md definen los componentes. EIC, EOC, IC Definen la conexión entre los componentes y también la conexión de éstos con la interfaz del modelo compuesto.
- La función select establece la prioridad en caso de que varios componentes tengan planificada una transición interna para el mismo instante.

La descripción matemática del modelo es la siguiente:

X = {(p, v) |p ϵ InPorts, v ϵ Xp} Conjunto de entradas del modelo compuesto.

Y = {(p, v) |p ϵ OutPorts, v ϵ Yp} Conjunto de salidas del modelo compuesto.

D Conjunto de nombres de los componentes.

Los componentes son modelos DEVS. Para cada d ϵ D se verifica:

DEVS = < X, S, Y, δint, δext, λ, ta > Es un modelo DEVS clásico, que en general tendrá varios puertos de entrada y salida.

Xd= {(p, v) |p ϵ InPorts, v ϵ Xp)}; Yd= {(p,v)|p ϵ OutPorts, v ϵ Xp)}.

En la Figura 1.13 se nos muestra un ejemplo de conexión entre modelos DEVS atómicos.

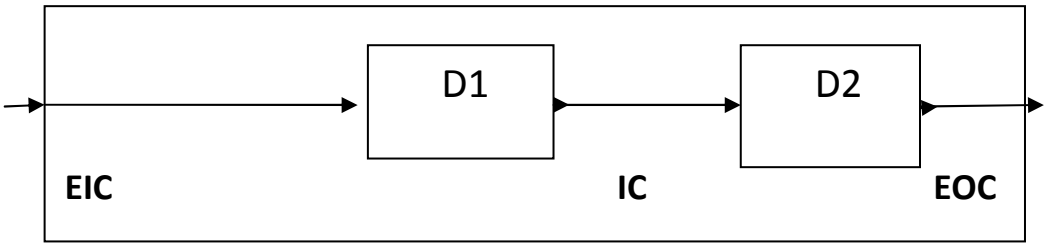


Figura 1.13: Tipos de conexión entre modelos DEVS con puertos

EIC: Entrada externa-Entrada de un componentes. Se trata de una entrada al sistema que a su vez es la entrada a un componente del sistema. La definición del mismo es la siguiente:

EIC = {((N, ipn), (d, ipd)) | ipn ϵ InPorts, d ϵ D, ipd ϵ InPorts}

Donde:

(N, ipn) representa el puerto de entrada ipn del modelo compuesto.

(d, ipd) representa el puerto de entrada ipd del componente d.

((N, ipn), (d, ipd)) representa la conexión entre ambos puertos.

EOC: Salida de un componente–Salida externa. Se trata de la salida de un componente del sistema que a su vez corresponde a la salida del sistema. La definición del mismo es la siguiente:

EOC = {((d, opd), (N, opn)) | opn ϵ OutPorts, d ϵ D, opd ϵ OutPorts}

Donde:

(d, opd)	representa el puerto de salida opd componente d.
(N, opn)	representa el puerto de salida opn del modelo compuesto.
((d, opd), (N, opn))	representa la conexión entre ambos puertos.

IC: Salida de un componente-Entrada de un componente. Se trata de una conexión entre la salida de un componente y la entrada de otro componente del mismo sistema. La definición es la siguiente:

IC = {((a, opa), (b, ips)) | a, b ϵ D con a≠b, opa ϵ OutPortsa, ipb ϵ InPortsb}

Donde:

(a, opa)	representa el puerto de salida opa del componente a.
(b, ipb)	representa el puerto de entrada ipb del componente b.
((a, opa), (b, ipb))	representa la conexión entre ambos puertos.

Hay que observar que DEVS no permite que una salida de un componente se conecte a una entrada del mismo componente (realimentación de la señal) esto queda expresado con la condición a, b ϵ D con a≠b.

Select: se define como una función del tipo:

Select: $2^D \rightarrow D$ La función select, establece la prioridad en caso de que varios componentes tengan un evento interno planificado para el mismo instante.

Para ampliar la información sobre DEVS acoplados modularmente remitimos al texto Modelado de Sistemas Mediante DEVS (Zeigler, Praehofer y Kim, 2000); de donde se ha extractado esta información de manera casi literal.

1.1.4 PowerDEVS

Existen varios entornos para simulación y desarrollo de aplicaciones mediante DEVS de los cuales destacamos DEVSJAVA, desarrollado en el ámbito de la universidad de Arizona por Zeigler & Sarjoughian en 2005 siendo propiedad intelectual de Regents of the State of Arizona siendo éste uno de los más utilizados, en parte por estar escrito en Java. Existen otros entornos como pueden ser JDEVS (Filippi et al, 2002) desarrollado en la Universidad de Córcega y CD++ (Wainer et al., 2001) desarrollado por la Universidad de Carleton (Ottawa, Canadá). En el caso que nos ocupa hemos optado por utilizar el entorno PowerDEVS que pasamos a explicar a continuación.

PowerDEVS: como se ha explicado anteriormente, se trata de un entorno de simulación de propósito general para simulación de modelos DEVS. PowerDEVS (Pagliero et al, 2003) desarrollado originalmente en la Facultad de Ciencias

Exactas, Ingeniería y Agrimensura de la Universidad de Rosario (Argentina), este software está publicado bajo licencia GNU y LPGL por tanto es de uso libre.

Existe poca documentación acerca de este entorno (especialmente escasa en español), por lo cual hemos considerado de utilidad el realizar un tutorial, aprovechando de esta forma la experiencia adquirida en el desarrollo de este PFC. Entendiendo que esta tarea, tiene la suficiente entidad como para que le dediquemos un capítulo de esta memoria; en el capítulo siguiente, se amplía la información de este entorno.

1.2 **OBJETIVOS**

Para iniciar este punto, entendemos que nada mejor que transcribir los objetivos literalmente, para poder desarrollarlos con posterioridad.

1. Comprender el funcionamiento del circuito digital consistente en una CPU conectada a una memoria, que está descrito en el Capítulos 12 y sucesivos del texto base (Perry, 2002).

2. Proponer un banco de pruebas para dicho circuito.

3. Aplicar el formalismo DEVS a la descripción de un sistema de complejidad media, como es el circuito digital de una CPU conectada a una memoria y su banco de pruebas.

4. Aprender a manejar un entorno de simulación para DEVS como puede ser PowerDEVS, y aplicarlo a la simulación del modelo DEVS del circuito y su banco de pruebas.

5. Aprender a describir circuitos de complejidad media usando VHDL y sus bancos de pruebas. Describir usando VHDL el circuito consistente en una CPU y la memoria, y su banco de pruebas. Simular el circuito y su banco de pruebas usando un simulador de VHDL, como puede ser ModelSim PE Student Edition.

6. Validar el modelo DEVS comparando los resultados obtenidos de su simulación con los resultados obtenidos de simular el banco de pruebas y el circuito descritos en VHDL.

Estos objetivos son los que aparecen en el documento consensuado con el director del proyecto, en el anteproyecto del PFC y que se han transcrito literalmente, como aparecen en dicho documento. Durante el desarrollo del proyecto, se ha detectado la falta de manuales de PowerDEVS en español, esto nos ha hecho pensar, como forma de añadir valor al proyecto, en desarrollar un pequeño tutorial de la herramienta, pasando a ser un objetivo que se ha incluido tácitamente en el proyecto, por aportar valor al mismo. En el Capítulo 2, pasaremos a realizar una explicación tutorial de la herramienta PowerDEVS.

Pasamos por tanto, a desarrollar y detallar los objetivos del presente proyecto, que consiste básicamente, en implementar un circuito consistente en una CPU conectada a una memoria, siguiendo el patrón marcado en el texto de D.L. Perry “VHDL Programming by example” que en los Capítulos 12 y 13 (algo

menos en los sucesivos), realiza una somera descripción del funcionamiento de una CPU conectada a una memoria.

Por tanto, ese es el fondo documental básico, sobre el que debemos realizar el modelado del circuito. Para cualquier cosa que no quede recogida en el texto o no se explique con meridiana claridad su comportamiento, el autor de este PFC se ha reservado el derecho, no violando el espíritu fundamental de la obra, de decidir por sí mismo o en base a otros textos la implementación final del mismo, insisto, de aquellos pormenores que no están reflejados o suficientemente explicados. Cabe resaltar que por desgracia, estos aspectos son muy abundantes, pues no se explica suficientemente (por ejemplo), como funciona cada instrucción, simplemente explica el funcionamiento de una de ellas, de una manera muy superficial.

1.2.1 Instrucciones de la CPU

Tal y como se ha indicado en el apartado anterior, una característica de este proyecto es, que un aspecto fundamental como la especificación del funcionamiento básico de cada instrucción, no se ha especificado claramente, teniendo que obtenerse a través del funcionamiento del código VHDL que se presenta como ejemplo, en el libro de D. I. Perry.

Esto puede ocasionar, que se hayan originado pequeñas discrepancias, en cuanto al funcionamiento de las instrucciones, no siendo estas importantes, ni afectan al funcionamiento de más alto nivel, siendo transparentes para el usuario y tratándose solo de pequeñas variaciones en los destinos de las ejecuciones de algunas instrucciones.

Lo reseñado anteriormente tiene su máxima expresión en las instrucciones de salto “Branch less than”, “Branch not equal”, “Branch if equal”, “Branch greater than”, “branch all the time”, “Branch if less or equal”, pues aunque a priori, su ejecución es la misma o muy similar para todas (la misma ruta de ejecución) no explica en ningún momento el funcionamiento de las mismas. En cambio si lo hace con las instrucciones equivalentes, que son aquellas en las que, la dirección de salto aparece en la segunda palabra de la instrucción (ejemplo: “Branch to immediate address”, etc.).

Entendemos por tanto, que en un caso como este, de un ejemplo sencillo de CPU, y puesto que sí aparece explicado con claridad el funcionamiento de las instrucciones equivalentes (“Branch……inmediate”) donde la dirección de salto queda especificada en la segunda palabra de instrucción.

No se ha procedido a implementar las instrucciones de salto condicional de una sola palabra, siendo usadas en los casos de prueba, las equivalentes que especifican la dirección de salto, en la segunda palabra, por los motivos antes mencionados.

Por tanto las instrucciones de salto, “Branch less than”, “Branch not equal”, “Branch if equal”, “Branch greater than”, “branch all the time”, “Branch if less or equal”, no se han implementado en nuestro proyecto, siendo sustituidas por las equivalentes de doble palabra.

Adjuntamos la tabla de instrucciones del texto base que se muestra en Tabla 1.1.

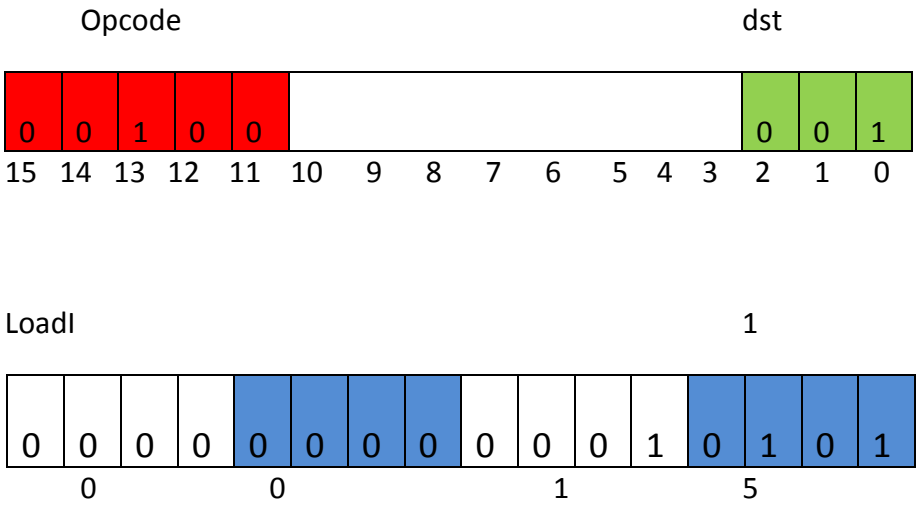
OPCODE	INSTRUCTION	NOTE
00000	NOP	No operation
00001	LOAD	Load register
00010	STORE	Store register
00011	MOVE	Move value to register
00100	LOADI	Load register with immediate value
00101	BRANCHI	Branch to immediate address
00110	BRANCHGTI	Branch greater than to immediate address
00111	INC	Increment
01000	DEC	Decrement
01001	AND	And two registers
01010	OR	Or two registers
01011	XOR	Xor two registers
01100	NOT	Not a register value
01101	ADD	Add two registers
01110	SUB	Subtract two registers
01111	ZERO	Zero a register
10000	BRANCHLTI	Branch less than to immediate address
10001	BRANCHLT	Branch less than
10010	BRANCHNEQ	Branch not equal
10011	BRANCHNEQI	Branch not equal to immediate address
10100	BRANCHGT	Branch greater than
10101	BRANCH	Branch all the time
10110	BRANCHEQ	Branch if equal
10111	BRANCHEQI	Branch if equal to immediate address
11000	BRANCHLTEI	Branch if less or equal to immediate address
11001	BRANCHLTE	Branch if less or equal
11010	SHL	Shift left
11011	SHR	Shift right
11100	ROTR	Rotateright
11101	ROTL	Rotate left

Tabla 1.1: Tabla de instrucciones del libro VHDL Programming by Example

Tal y como se explica en el libro base (Perry,2002), las instrucciones pueden ser de simple palabra o de doble palabra; en la Figura 1.14 se nos muestra un ejemplo de formato de la instrucción LoadI # en hexadecimal. No se nos presentan ejemplos de otras instrucciones, con lo cual la interpretación del formato de las instrucciones que restan, ha sido obtenido o bien mirando el escaso código que aparece en el texto base o bien criterio propio.

Reproducimos en la Figura 1.14 la información extractada del formato de doble palabra, seguida de la instrucción de simple palabra, donde se nos especifica el formato de las instrucciones de una sola palabra.

Instrucción: LoadI, 16#15



Para el formato de simple palabra el libro apenas aporta claridad limitándose a decir:

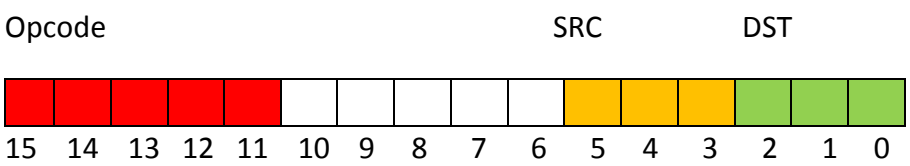


Figura 1.14: Diagrama posiciones en vector de instrucción

Llegado a este punto, pasamos a describir brevemente, qué realiza cada operación, junto con la secuencia de microinstrucciones o pasos que va a ir simulando cada programa para completar cada instrucción.

(01) Load :(00001) Esta instrucción es single word, carga en el registro indicado en la dirección 2 a 0, el valor almacenado en la posición de memoria que se guardada en el registro de la dirección, que se indica en 5 a 3.

A continuación detallamos los pasos que se realizan para ejecutar la instrucción:

Vector registros->bus datos->registro direcciones->bus direcciones -> memoria (lee los datos)->bus datos-> vector registros-> [incPc…].

(02) Move:(00011) Esta instrucción es single word y su función es mover el valor almacenado en el registro cuya dirección se indica en 5 a 3 a otro registro cuya dirección se indica en 2 a 0.

A continuación detallamos los pasos que se realizan para ejecutar la instrucción:

Vector de registros ->bus datos-> ALU-> registro desplazamiento -> registro de salida ->bus datos-> vector de registros.

(03) LoadI:(00100) Esta instrucción es doble word y su función consiste en cargar en el registro, cuya dirección se indica en las posiciones 2 a 0, el valor binario o hexadecimal que se indica en la palabra siguiente.

A continuación detallamos los pasos (ruta de datos) que se realizan para ejecutar la instrucción:

Contador programa-> ALU (inc)-> registro desplazamiento -> registro de salida -> contador programa, registro de direcciones->bus direcciones ->memoria->bus datos -> vector registros.

(04) Store:(00010) Esta instrucción es single word y su función es, almacenar en la dirección de la memoria que se indica en el registro cuya dirección

está en 2 a 0, el valor que se almacena en el registro cuya dirección se indica en 5 a 3.

A continuación detallamos los pasos (ruta de datos) que se realizan para ejecutar la instrucción:

Vector registros->bus datos->registro direcciones->bus de direcciones -> | ->memoria

vector registros->bus datos-> | - >memoria

(05) BranchI:(00101) Esta instrucción es doublé word y su función es la de realizar un salto incondicional, es decir saltara a la dirección de programa que se indica en la palabra que le sigue.

A continuación detallamos los pasos (ruta de tatos) que se realizan para ejecutar la instrucción:

Contador programa->bus datos-> ALU (inc) -> registro desplazamiento (pass)-> registro de salida ->Contador de programa(+1)->registro direcciones->bus direcciones->memoria->bus datos->load pc

(06) BranGTI(00110),BranchLTI(10000),BranchNEQI(10011),BranchEQI(10111),BranchLTEI(11000) : todas estas instrucciones son de doble palabra y su función es, realizar un salto a la dirección de programa que se indica en la palabra siguiente, siempre y cuando se cumpla la condición, que se expresa en su nombre de instrucción.

A continuación detallamos los pasos (ruta de datos) que se realizan para ejecutar la instrucción:

Vector registros->bus datos->registro operaciones-> comparador->|

Vector registros ->bus datos-> comparador->|

|->control (si cumple condición) -> incPC…

|->control (si no cumple condición) ->contador programa->

Bus datos-> ALU (inc) -> registro desplazamiento -> registro de salida ->bus datos ->contador programa (+1) -> incPC

(07) BranGT(10100),BranchLT(10001),BranchNEQ(10010),BranchEQ(10110),BranchLTE(11001) : Estas instrucciones no se especifica si son de simple palabra o doble y su función entendemos que es la de realizar un salto condicionado a que se cumpla una condición que se expresa en el propio nombre de la instrucción. No se ha implementado por motivos que se han explicado con anterioridad.

(08) Inc(00111), Dec (01000), And (01001); Or (01010), Xor (01011), Not (01100), Add (01101), Sub (01110), Zero (01111): Estas instrucciones son single word y su función son realizar las operaciones unarias o binarias en la ALU, con los valores que se almacenan en los registros cuya dirección se indica en 5 a 3 y 2 a 0 guardando el resultado en el registro que se indica en 2 a 0.

A continuación detallamos los pasos (ruta de datos) que se realizan para ejecutar la instrucción:

Operaciones binarias.

Vector registros->bus datos->registro operaciones-> ALU->|

Vector registros->bus datos-> ALU->|

|-> ALU-> registro desplazamiento->registro salida ->bus datos-> vector registros->inc PC…

Operaciones unarias.

Vector registros->bus datos -> ALU->registro desplazamiento->registro salida->bus datos->vector registros->incPc…..

(09) Shl (11010), Shr (11011): Operaciones unarias que no se realizan en la ALU, sino en el registro de desplazamiento, se realizan sobre el valor almacenado en el registro que se indica en 2 a 0 volviendo a almacenar el resultado en el mismo registro.

A continuación detallamos los pasos (ruta de datos) que se realizan para ejecutar la instrucción:

Vector registros->bus datos -> ALU (pass)-> registro desplazamiento(realiza operación)-> registro de salida->bus datos->vector registros->incPC…

(10) Nop (00000): No realiza ninguna función.

Existen dos operaciones, que no se pueden invocar por parte del programador. Estas tiene como misión, agrupar operaciones repetitivas que se realizan a menudo en la ejecución del programa estas operaciones son.

IncPc: Operación que se encarga de incrementar el contador de programa y cargar la siguiente instrucción, se realiza cada vez que finaliza la ejecución de una instrucción.

La ruta de datos se indica a continuación:

Contador programa ->bus datos-> ALU (inc)->registro desplazamiento->registro de salida-> bus datos->contador programa->registro direcciones->bus direcciones->memoria (lee instrucción)->bus datos->registro de instrucciones ->control->…

LoadPC: Operación cuya misión es cargar la siguiente instrucción en las instrucciones de salto incondicional

Contador programa->registro direcciones->memoria (lee instrucción) ->bus datos-> Execute…

Es importante destacar, que esta ruta de datos corresponde al esquema original del texto base, que también se corresponde con el programa en versión VHDL. Esta ruta de datos arroja unas pequeñas diferencias con la versión del programa realizado para el entorno PowerDEVS. Esto es debido a las pequeñas modificaciones, que se han tenido que realizar en el esquema eléctrico original, las cuales se comentan en el Capítulo 5.

Estas modificaciones, vienen impuestas, por la necesidad de implementar una serie de registros, que impidan la realimentación directa de algunos elementos del diseño y que por tanto aumentaran la ruta de datos. Estos cambios, no son significativos, pues básicamente no alteran la ruta de datos, solo que la aumentan al existir más elementos en el esquema. Pasamos a detallar un ejemplo del caso referido:

Como se comentara más adelante, en el elemento “Regarray” en la versión PowerDEVS se añadirá un registro para evitar la realimentación de datos directa, entonces esto aumenta la ruta de datos por ejemplo en el caso original:

vector registros->bus datos->etc.

Cambia en la versión para PowerDEVS a:

vector registros ->registro salida-> bus datos->etc.;

Tal y como se puede apreciar, la variación en la ruta de datos no es significativa, y por tanto nos parece que no es interesante (pues no representa un cambio sustantivo) volver a repetir la rutas de datos para el caso de PowerDEVS. Remitimos a aquellas personas interesadas en las diferencias, en revisar el código fuente (que se adjunta en los anexos), pues obligatoriamente en este código se han reflejado las modificaciones oportunas, como no podría ser de otra manera, para el correcto funcionamiento del programa.

1.2.2 Diagrama de bloques de la CPU

Como base para la realización del PFC en el libro (Perry, 2002), aparece un esquema de conexión de todos los elementos y que reproducimos a continuación en la Figura 1.15.

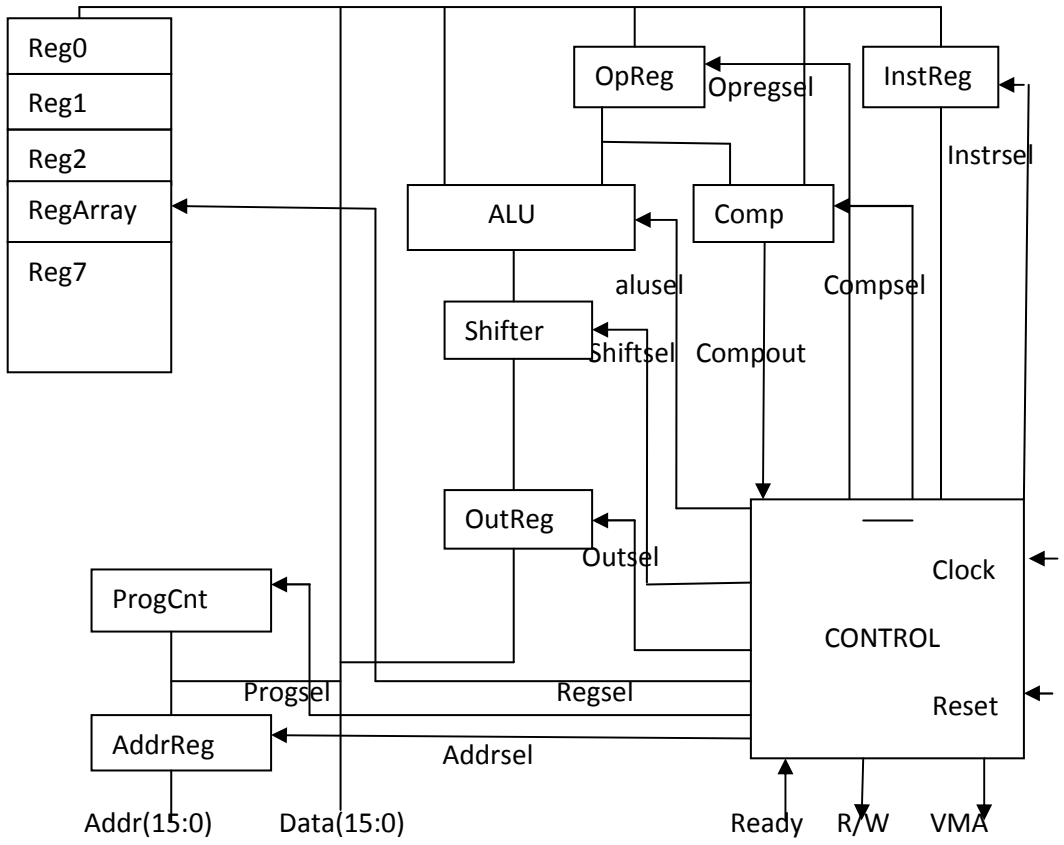


Figura 1.15: Esquema de conexión elementos de la CPU

Este esquema, es la guía sobre la cual se ha realizado en PFC , que con pequeñas modificaciones, ha sido usado en el desarrollo del mismo.

En el esquema que se nos muestra en la Figura 1.16 se nos muestra las conexiones que van a enlazar nuestra CPU con la memoria que es el objetivo de nuestro PFC, pues para que podamos ejecutar un programa en la CPU necesitamos de una memoria donde se almacenara nuestro programa.

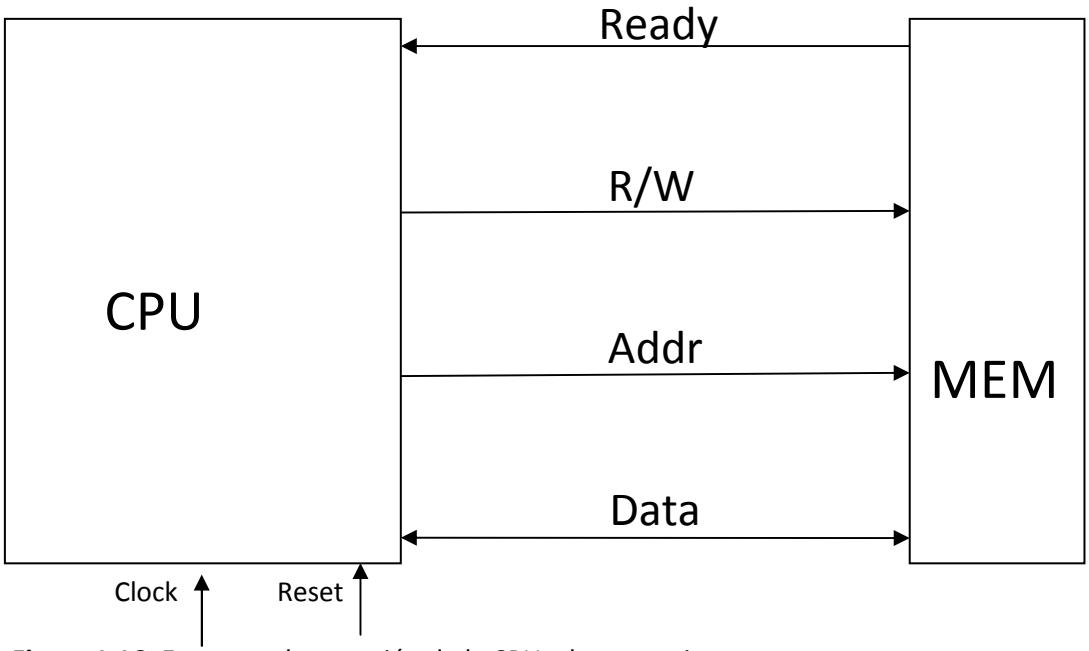


Figura 1.16: Esquema de conexión de la CPU a la memoria

1.2.3 Modelado del circuito

El circuito propuesto, se va a modelar con dos metodologías diferentes, una mediante el lenguaje VHDL. Realizando la simulación de un banco de pruebas mediante un entorno que soporta dicho lenguaje como es ModelSim y otra metodología basada en DEVS. Realizando la simulación del banco de pruebas sobre el entorno PowerDEVS.

El objetivo de realizar el circuito con dos metodologías distintas, no puede ser otro, que el tener una referencia de comparación, de los resultados obtenidos. Con los cuáles procederemos a la validación de los modelos que hemos realizado, mediante la simulación de un banco de pruebas que implementaremos, para poder simular los circuitos obtenidos y cuyas directrices pasamos a explicar.

1.2.4 Programación del banco de pruebas

Para la realización del banco de pruebas, no tenemos directrices en el texto base, pues en él, se nos presenta un pequeño programa en VHDL que consiste en copiar los datos existentes en una parte de la memoria a otra zona de la memoria.

Nos ha parecido correcto el introducir dicho programa como parte del banco de pruebas, y como no, hemos procedido a ampliarlo con nuestras aportaciones. Tal y como hemos explicado con anterioridad el circuito a implementar consiste en una CPU conectada a una memoria, lo cual sería el precursor de un ordenador moderno. Nos ha parecido, que la mejor prueba que se podría realizar sobre dicho circuito, consistirá en introducir un pequeño programa en su memoria y ejecutarlo. Comprobando, si los resultados obtenidos concuerdan, primero con los resultados esperados de una ejecución manual del mismo, comparándolos a su vez, entre las dos implementaciones del circuito. Entendiendo que a programas equivalentes deberán proporcionar resultados equivalentes.

Para desarrollar esos programas, nos hemos establecido dos premisas. Primero, que los programas a ejecutar sean sencillos y no muy extensos, pues esto facilitara el seguimiento de la ejecución de instrucciones, así como la detección de los errores si los hubiera. La otra premisa es, que en pocos programas se prueben

la práctica totalidad de las instrucciones que hemos implementado. Entendemos, que si un programa es una acumulación de instrucciones más sencillas, si estas funcionan correctamente de manera simple; también lo harán cuando estén implementadas todas juntas en un programa más complejo.

La explicación detallada de los programas del banco de pruebas, nos ha parecido más correcto incluirla en el apartado dedicado a la simulación de dichos programas, pero esto no es óbice, para que realicemos una breve explicación del banco de pruebas.

El banco, se compone de cuatro programas, que tal y como hemos explicado anteriormente, uno es el que hemos denominado “Copiar de memoria a memoria” este programa lo que hace es copiar los datos incluidos por nosotros en una parte de la memoria a otra parte de la memoria sin borrar los anteriores.

Otro programa lo hemos denominado “bucle while y alternativa if/else”, consiste como su propio nombre indica en realizar una bucle while y una alternativa if /else, pues con esos se pueden realizar cualquier tipo de bucle que deseemos. Si se prueba el funcionamiento correcto de estos dos tipos, estamos en condiciones de desarrollar cualquiera de los bucles que deseemos.

Un tercer programa que hemos denominado “Bucle interior a un bucle”, implementa un bucle interior a otro bucle, con lo cual, el bucle interno se ejecutara tantas veces como indique el externo.

En un cuarto programa, denominado “operaciones matemáticas”, implementamos todas las operaciones lógica y matemáticas que hemos

implementado en nuestra ALU, tratando así de demostrar la corrección de la implementación de esta parte muy importante de la CPU.

El citado banco de pruebas, no está configurado como pruebas de software tradicional (pruebas de caja blanca, de caja negra, en los limites, etc.…) pues no se trata en este caso de probar un software comercial. En este caso nos parecido más importante, el probar primero el funcionamiento de las instrucciones individualmente, para posteriormente probar el comportamiento de éstas, cuando se encuentran funcionando, agrupadas en un pequeño programa.

1.3 ESTRUCTURA

El presente proyecto, presenta dos partes claramente diferenciadas, la implementación mediante la herramienta PowerDEVS; la otra parte seria la implementación de dicho circuito, mediante una herramienta diferente, en este caso ModelSim (VHDL), que nos servirá como demostración del circuito y comparación de resultados obtenidos.

Por tanto es lógico, que el empleo de herramientas diferentes, para construir las diferentes partes del proyecto, provoque que el citado proyecto se nos presente, con dos partes diferenciadas en la estructura del mismo. En consecuencia, en el CD donde se entregan los resultados del PFC, aparecen dos carpetas una dedicada cada implementación, según se aprecia en la Figura1.17 que se presenta a continuación.

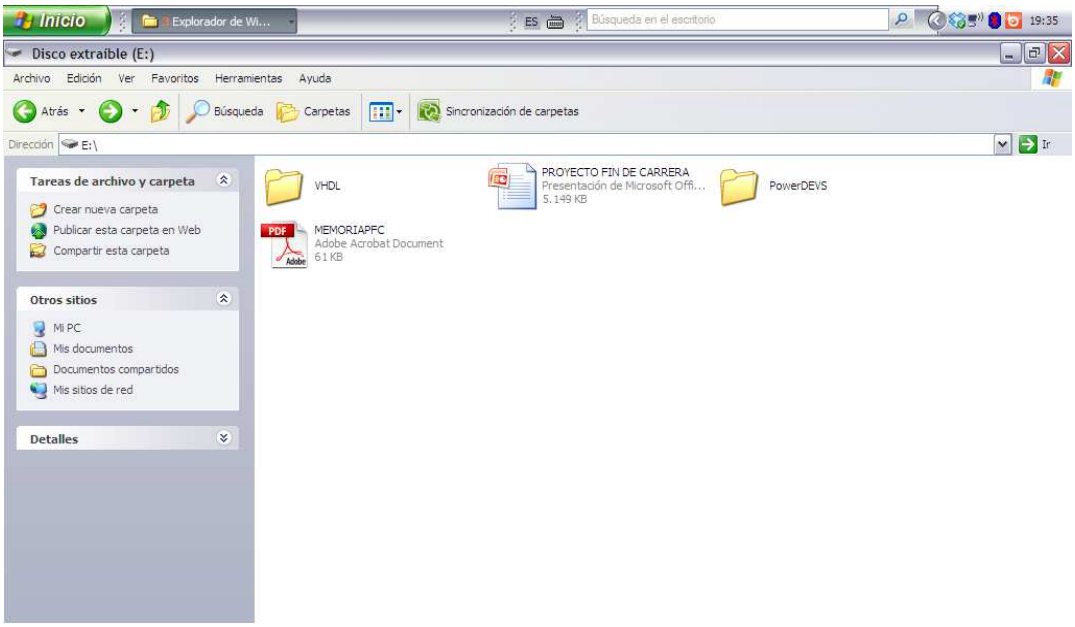


Figura 1.17: Estructura de archivos presentes en el CD del PFC

Como se aprecia en la figura, en el CD del proyecto, encontramos una carpeta con el nombre PowerDEVS, en esta carpeta encontraremos toda la herramienta completa, con lo cual la comprobación de los resultados obtenidos en dicho entorno, no requerirán de la instalación previa de dicha Herramienta.

Podremos utilizar el entorno simplemente copiándolo en el disco duro de nuestro ordenador. Entendemos que esto facilita la realización de pruebas, tanto las que hemos desarrollado como parte del proyecto, así como aquellas que el tribunal considerase a bien en realizar.

La utilización de la herramienta, esta descrita en el Capítulo 2. Por tanto de la estructura de directorios del entorno, nos parece importante resaltar solo dos directorios concretos, el primero es el que aparece en la Figura 1.18.

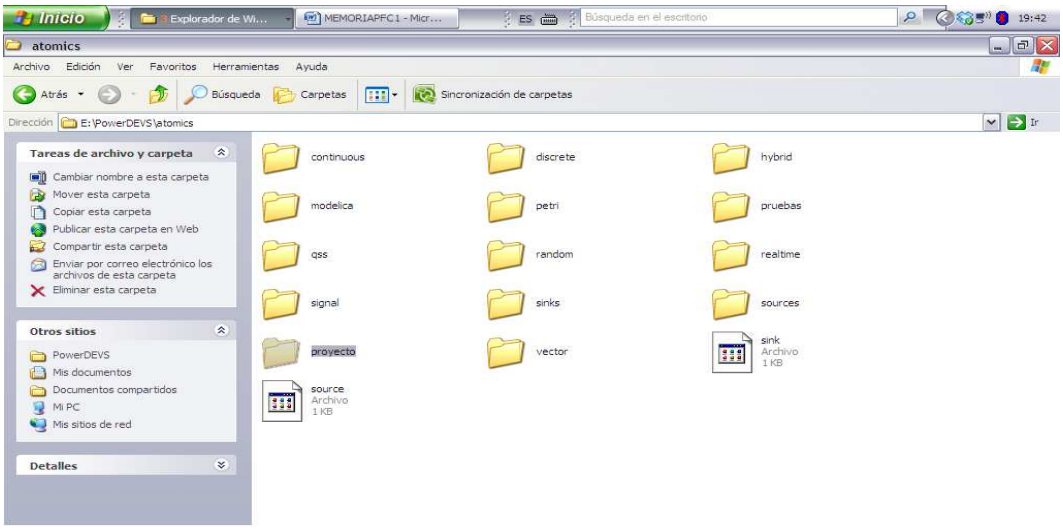


Figura 1.18: Estructura de archivos presentes en el CD del PFC dentro del directorio PowerDEVS

En la carpeta nombrada como “proyecto”, es donde están situados los archivos desarrollados dentro del entorno PowerDEVS para el PFC, por tanto en el encontraremos prácticamente todo el código relacionado con el desarrollo del circuito propuesto, que ha sido desarrollado por nosotros.

Otra parte importante del proyecto, consiste en la implementación del circuito en lenguaje VHDL usando ModelSim PE Student Edition. Todo lo referente a la implementación del circuito en dicho entorno, aparece en la carpeta con el nombre VHDL.

Cabe resaltar, que en este caso en dicho directorio, no se encuentra la herramienta o entorno, lo cual obligará a disponer del mismo o proceder a su instalación, para poder realizar simulaciones del mismo. Tanto las implementadas por nosotros en el ámbito de este PFC, como aquellas que se consideran oportunas, por parte del tribunal.

En el directorio reseñado, encontraremos los archivos creados para la implementación del presente proyecto, en la parte concerniente a VHDL tal y como se muestra en la Figura 1.19.

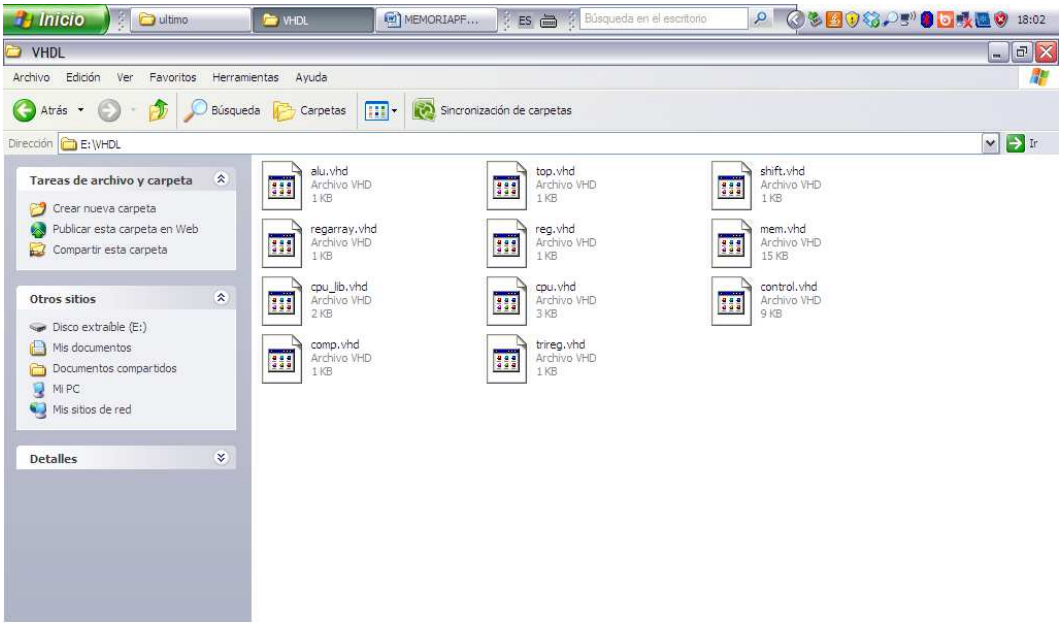


Figura 1.19: Estructura de archivos presentes en el CD del PFC dentro del directorio VHDL

Siguiendo las instrucciones de presentación del PFC, en el CD se presenta una presentación en PowerPoint, que es con la que se defenderá el PFC ante el tribunal, el archivo aparece en el CD con el nombre “PROYECTO FIN DE CARRERA”.

Hasta aquí, se ha explicado la configuración del CD que se entrega como resultado del PFC; pero una parte importante del citado PFC está constituida por la “memoria” del mismo, entendemos por tanto que también procede que se realice una breve explicación de la misma.

La memoria propiamente dicha está compuesta por ocho capítulos, de los cuales el lector está a punto de concluir el primero titulado “introducción,

objetivos y estructura", en él cómo habrá podido comprobar, se trata de establecer un marco sobre el cual se describirá y desarrollara la memoria del PFC.

En el segundo Capítulo, titulado "Introducción tutorial a PowerDEVS", presentamos un breve tutorial de uno de los entornos, que se va a utilizar en el presente PFC, tratando de cubrir un hueco o necesidad detectado durante la elaboración del mismo, pues nos ha sido muy difícil encontrar documentación del entorno "PowerDEVS", (Tutoriales, ejemplos, etc.) en español. Por tanto en este capítulo, pretendemos, a la vez que facilitamos la comprensión del presente PFC por parte del lector; rellenar en parte el hueco documental del entorno.

No ocurre lo mismo con el entorno "ModelSim", por lo que no hemos desarrollado un tutorial para dicho entorno. Limitándonos a documentar donde podemos encontrar información de dicho entorno.

En el Capítulo 3, titulado "Diseño VHDL del circuito", describimos la implementación desarrollada del circuito propuesto (CPU conectada a una memoria), mediante el lenguaje VHDL. Realizando una explicación detallada de dicha implementación, pues esta será la base sobre la que desarrollaremos posteriormente la implementación "PowerDEVS" del mismo circuito.

En el Capítulo 4 titulado "Programación VHDL del banco de pruebas", desarrollamos el banco de pruebas en VHDL. Con él, posteriormente trataremos de demostrar la validez del circuito implementado. Realizando una simulación del citado banco de pruebas en el entorno ModelSim y presentando los resultados obtenidos de dicha simulación. Estos resultados serán retomados en el Capítulo 6,

para comparación de los mismos, con la simulación PowerDEVS del banco de pruebas.

En el Capítulo 5 titulado “Modelado mediante PowerDEVS”, se presenta una explicación del código implementado para el desarrollo del circuito (CPU conectado a una memoria), así como la transposición del banco de pruebas desarrollado en VHDL, al entorno PowerDEVS.

En el Capítulo 6, titulado “Validación del modelo DEVS del circuito”, tratamos de demostrar la validez del circuito implementado en DEVS, mediante el entorno PowerDEVS. La comparación de resultados obtenidos por simulación de los bancos de prueba del circuito, mediante ModelSim y PowerDEVS, junto con la ejecución a mano del pseudocódigo equivalente, nos aportaran los datos para la comparación, que establecerán la validación de ambos modelos.

En el Capítulo 7, hemos tratado de establecer una pequeña memoria económica de los costes del proyecto. En dicho capítulo no se ha tratado de hacer un estudio riguroso de costes, sino mas bien, reflejar en dicho capítulo, que los costes, forman parte importante de un proyecto de desarrollo de software, limitándonos por tanto a realizar una planificación para el desarrollo del proyecto y un somero estudio de los costes.

En el Capítulo 8, se reflejan nuestras conclusiones, obtenidas de la realización del presente PFC, así como una breve explicación de los campos de mejora que podrían aplicarse en el ámbito en el que se ha desarrollado el PFC.

2

INTRODUCCIÓN TUTORIAL A POWERDEVS

2.1 INTRODUCCIÓN

Dentro de las herramientas utilizadas en este proyecto, podemos establecer un apartado de entornos de desarrollo y simulación. El que nos ocupa principalmente, es PowerDEVS en la version2.2. Existen sin duda varios entornos de desarrollo y simulación del formalismo DEVS, de entre ellos PowerDEVS, nos parece interesante por su facilidad de uso y por la interfaz gráfica que dispone el entorno, que es una de sus mejores características, por lo menos a nuestro entender.

Cabe destacar, la poca información que se dispone de esta herramienta, estando mucha de ella escrita en inglés. Nos parece interesante, que lo que hemos aprendido en el desarrollo del presente proyecto, se plasme en una especie de tutorial, que recopile nuestra experiencia en la utilización de esta herramienta.

No puede ser un documento exhaustivo, pues no somos los desarrolladores de la herramienta, además muchas de sus potencialidades, no han sido usadas en la elaboración del presente proyecto. Debido por otra parte a la citada escasez de información no estamos en condiciones de explicar el funcionamiento exhaustivo de todo el entorno, así como todas sus funcionalidades. Esto no nos debe apartar

de nuestro objetivo, que es plasmar en este apartado, toda la experiencia que hemos acumulado en el uso de la herramienta.

2.2 INSTALACIÓN

Qué duda cabe, que para poder entender el presente PFC, así como los pasos seguidos en este tutorial, lo primero que necesitamos es disponer de la herramienta en cuestión, en este caso PowerDEVS. Podemos descargar la herramienta desde varios entornos, nosotros hemos descargado la herramienta en varias ocasiones y en distintos ordenadores, usando en cada ocasión un enlace diferente [Sourceforge, 2013] y [Fceia, 2012].

En el caso del primer enlace, describimos brevemente los pasos a seguir para su instalación, que por otra parte resulta sencilla y muy similar a la de otras descargas que realizamos a través de la red. Situados en el sitio del enlace, encontraremos una pantalla similar a la mostrada en la Figura 2.1.

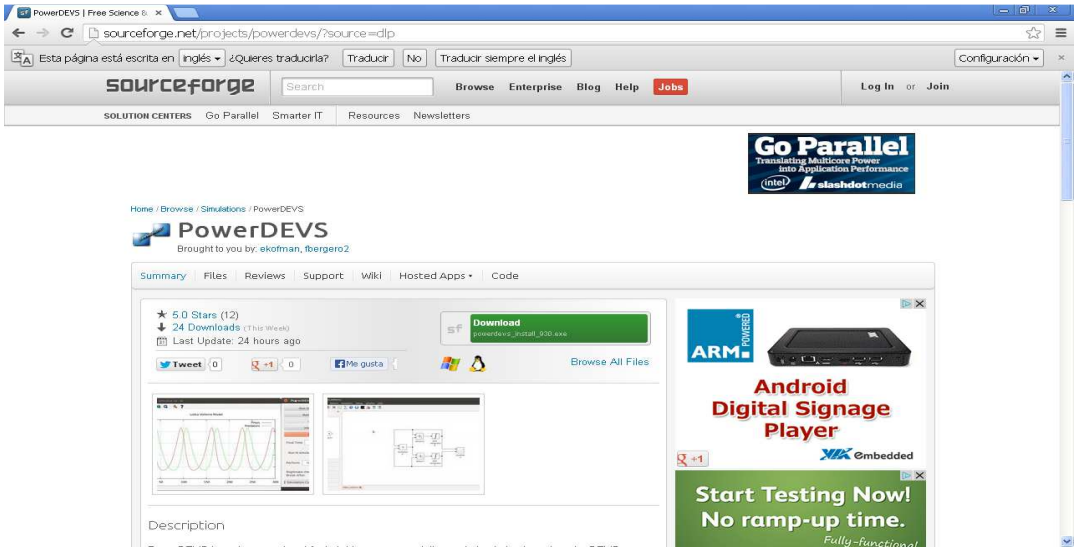


Figura 2.1: Instalación PowerDEVS paso 1

Pulsando sobre el icono verde “Download”, nos llevara a una pantalla donde se nos pide que ingresemos unos datos personales, pero que en nuestro caso, nos ha dejado continuar sin ingresarlos. Esta pantalla es similar a la mostrada en la Figura 2.2.

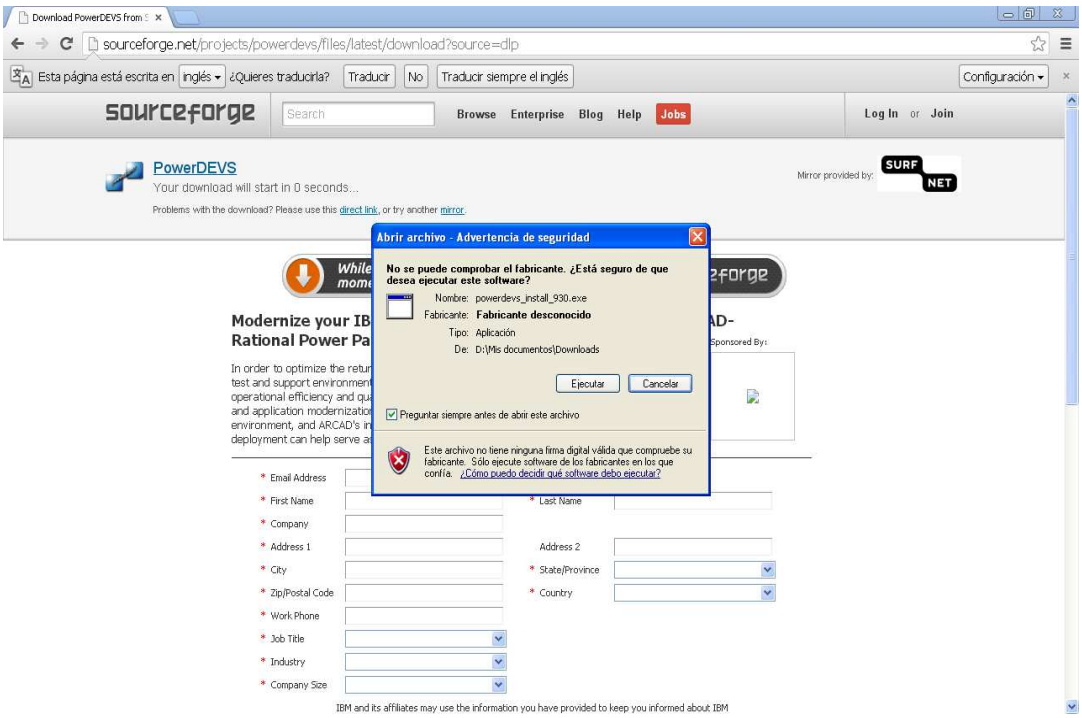


Figura 2.2: Instalación PowerDEVS paso 2

Una vez pulsamos sobre el botón ejecutar, se nos desplegará el menú de instalación, éste es muy similar a los menús de instalación de otros programas, el cual básicamente nos pedirá un directorio donde instalar el programa.

El programa de instalación, en nuestro caso no nos preguntó, si deseábamos crear un acceso directo y por tanto, lo creamos nosotros sobre el archivo “pdme.exe”; que encontraremos en el directorio “bin” de PowerDEVS.

2.3 ESTRUCTURA DE FICHEROS

Entendemos que la mejor manera de comenzar este tutorial, sería explicar la estructura de directorios (carpetas en adelante) del entorno, por tanto vamos a comenzar por mostrar una imagen de los directorios, que encontraremos cuando abramos la carpeta PowerDEVS. En la cual, el programa de instalación habrá colocado todo lo referente al entorno.

En la Figura 2.3 se nos muestra una imagen de la estructura inicial del entorno (a), junto con la estructura de la capeta “atomics” (b) en ella se muestran, agrupados por tipos en cada carpeta, los tipos de elementos simples que tiene el entorno.

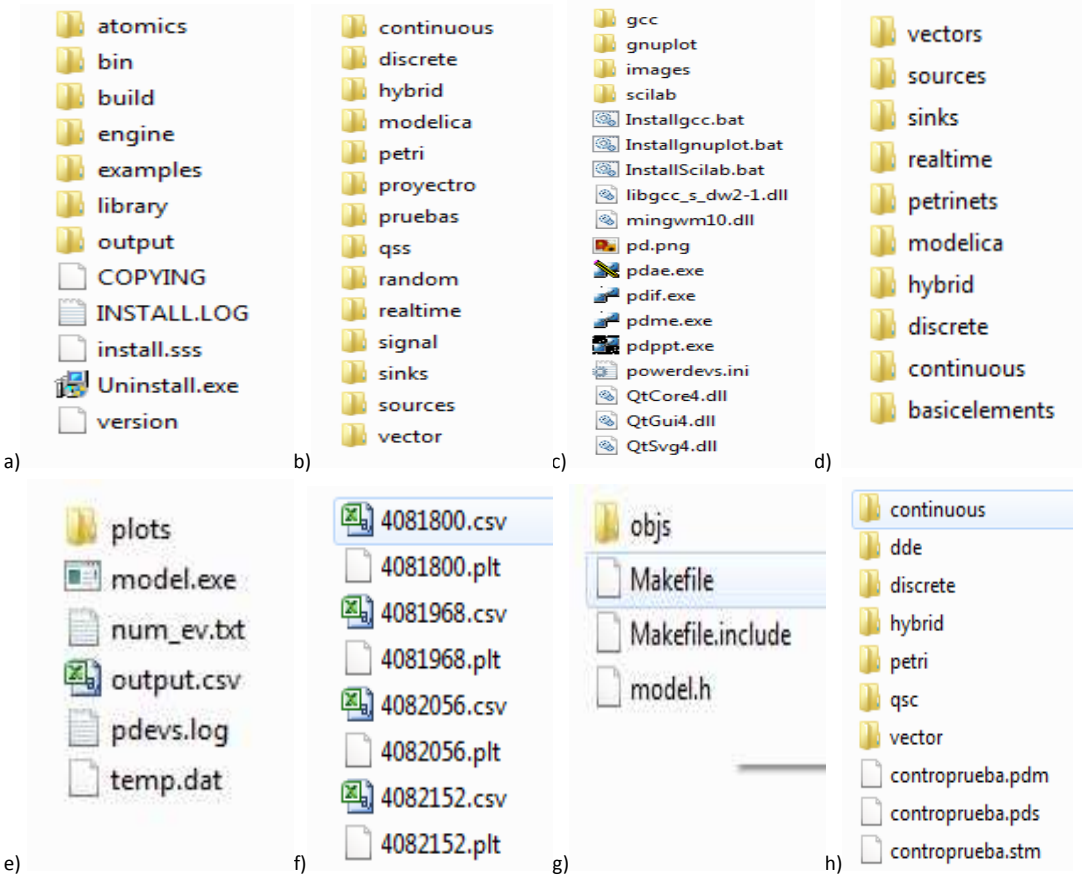


Figura 2.3: Estructura de archivos a)DEVS, b)atomics, c)bin, d)library e)output, f)output/plots, g)built, h)example

Estos elementos se detallaran más adelante, pues en definitiva son los elementos más simples, que podemos encontrarnos en la estructura de un proyecto, a partir de los cuales se construye el mismo.

En la Figura 2.3(c) se nos muestra el contenido de la carpeta “bin”, tal y como se aprecia en la imagen, en ella encontrarnos los archivos y carpetas de sistema del entorno, así como los ejecutables del mismo. Destacamos el archivo “pdme.exe”, pues es el que al pulsar sobre él, abrirá nuestro entorno y sobre el cual podremos realizar un acceso directo, para poder iniciar desde nuestro escritorio el entorno.

El siguiente directorio, es el nominado como “library”, como se aprecia en la Figura 2.3 (d). En él encontraremos, organizados por una estructura de directorios, organizada de manera similar a la del directorio atomics, las imágenes que utiliza el entorno para su funcionamiento e instalación.

Pasamos a explicar el directorio siguiente, nominado como “output”. Figura 2.3 (e), una vez abierto el directorio nos encontraremos los archivos y directorios, relacionados con las salidas del entorno de simulación, debemos resaltar el archivo “output.csv” y el directorio “plots”.

En el archivo “outpup.csv” Figura 2.3(f), tendremos disponibles al abrirlo, los pares de datos (tiempo de simulación, valor obtenido de la simulación) en un archivo de formato Excel, siempre y cuando conectemos la salida de la que queremos obtener los datos, a un objeto del menú “sinks”, denominado como “toDisk”, con el siguiente icono  . Sin embargo, si el elemento utilizado para la

obtención de datos es el nominado como “gnuPlot”, con el siguiente icono  , los datos además de presentarse inmediatamente por pantalla, mediante un grafico, estarán disponibles en el directorio nominado como “plots”. El resto de archivos que se presentan son ejecutables y demás archivos auxiliares para la simulación.

El siguiente directorio que se nos presenta es el directorio “build” Figura 2.3 (g); en el cual en el directorio “objs”, al abrirlo encontramos los archivos con extensión “.o” que son los archivos compilados de los código fuente; es decir los archivo que contienen el código objeto de los correspondientes “cpp”, por tanto, aquí se guardan los archivos automáticamente tras la compilación.

El siguiente directorio es el denominado como “Example”, Figura 2.3 (h), como ya habremos imaginado, es el directorio donde se presentan los proyectos compilados de los ejemplos, organizados por directorios; así como los de los proyectos desarrollados por nosotros, que en nuestro caso han sido colocados en un nuevo directorio nominado “proyecto”.

Resaltar que este directorio, es el que presenta el entorno por defecto, para que guardemos nuestros nuevos proyectos.

El directorio nominado como "engine"(no se encuentra representado en la Figura 2.3), en el encontraremos los archivos de C++ necesarios para las funcionalidades del entorno. Es un directorio del entorno en el cual nosotros no necesitamos actuar y por tanto no nos merece más explicación.

2.4 INTERFAZ DE USUARIO

Continuamos la explicación de la herramienta, con una presentación un tanto inusual. Cuando tenemos abierto en entorno PowerDEVS 2.2 (versión para Windows, en nuestro caso) al pulsar sobre el menú “help” obtenemos la siguiente información que se muestra en la Figura 2.4.

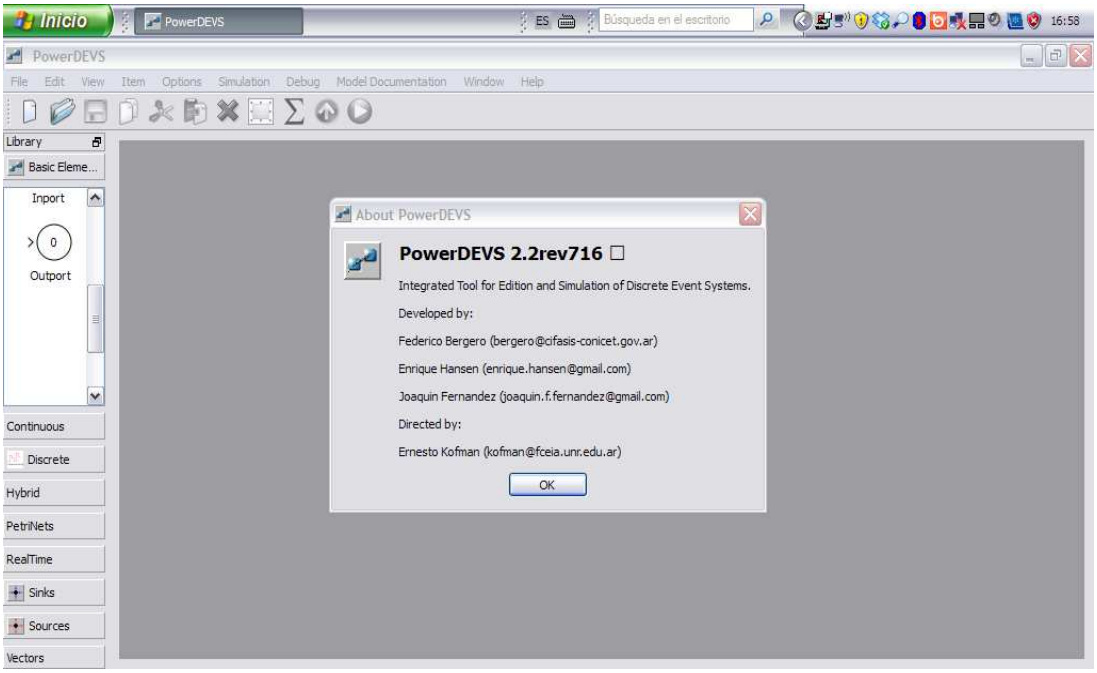


Figura 2.4: Pantalla inicial entorno PowerDEVS

En la Figura 2.4, se puede observar con claridad que se nos dice que se trata de una herramienta que integra la edición y simulación de sistemas de eventos discretos, a la vez, nos indica las personas que han desarrollado la herramienta; también se nos indica la versión, como no podía ser de otra manera.

Nos parece una buena manera de comenzar, la ayuda de la aplicación, pero el problema, es que será la única información que obtendremos del menú help. Esto es sin duda un debe de la herramienta. Quizá confiados sus autores, con la

más que aceptable interfaz gráfica y su sencillez de uso, podrían haber considerado innecesaria el desarrollo profundo de este apartado.

Para explicar el uso de la herramienta, vamos a basarnos en el desarrollo de un pequeño proyecto, explicando paso a paso, cómo se va haciendo este proyecto. Por desgracia, no podemos explicar mucho más, pues es lo que hemos utilizado en el desarrollo del presente proyecto.

Procederemos a abrir la herramienta, bien desde el acceso directo o desde el menú “inicio”, nos aparecerá la pantalla que se muestra en la Figura 2.5.

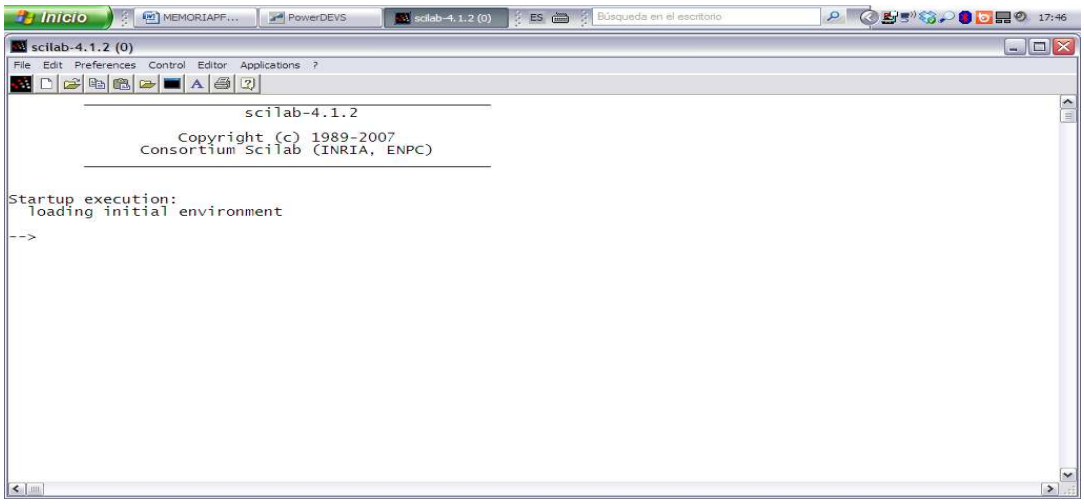


Figura 2.5: Pantalla Scilab entorno PowerDEVS

Deberemos cerrarla o bien minimizarla tras lo cual aparecerá la pantalla de PowerDEVS. Este es el entorno que vamos a utilizar para crear nuestro proyecto, que llamaremos “tutorial”; procedemos a la explicación de los pasos del mismo. La primera acción será pulsar sobre el menú “File->New” tal y como se indica en la Figura 2.6 (izquierda).

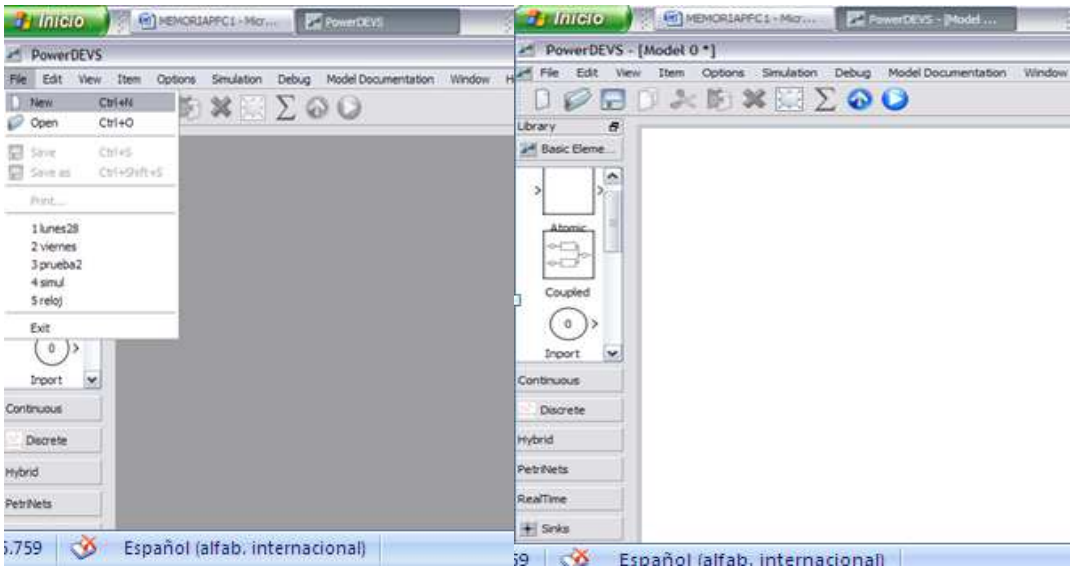


Figura 2.6: Pantalla menú "File" (izquierda) y el nuevo proyecto (derecha)

Obtendremos una plantilla para el desarrollo de nuestro proyecto que será la que se muestra en Figura 2.6 (derecha), cuando guardemos el proyecto nombraremos el mismo con el nombre elegido que es "tutorial".

La parte más importante del entorno, es sin duda las librerías que aparecen en el margen izquierdo, que se refieren a elementos ya creados, que son de uso generalizado. Estos están agrupados en función de su uso o características.

2.5 PANELES DE ELEMENTOS PREDEFINIDOS

El entorno, nos proporciona de inicio unos elementos predefinidos, que podemos utilizar tal cual están en las librerías, o bien son susceptibles de modificación por parte del usuario aprovechando de esta manera la estructura existente.

Estos elementos se nos muestran en la interfaz, agrupados según su función. Quizá los elementos más importantes para el diseño, sean los elementos que se agrupan en la pestaña denominada “Basic Elements”.

Pues con esto elementos, que ejercen como plantillas, podremos definir nuestros nuevos elementos, sin necesidad de modificar los ya existentes. Pasamos a describir los diferentes grupos de elementos, en la Figura 2.7.

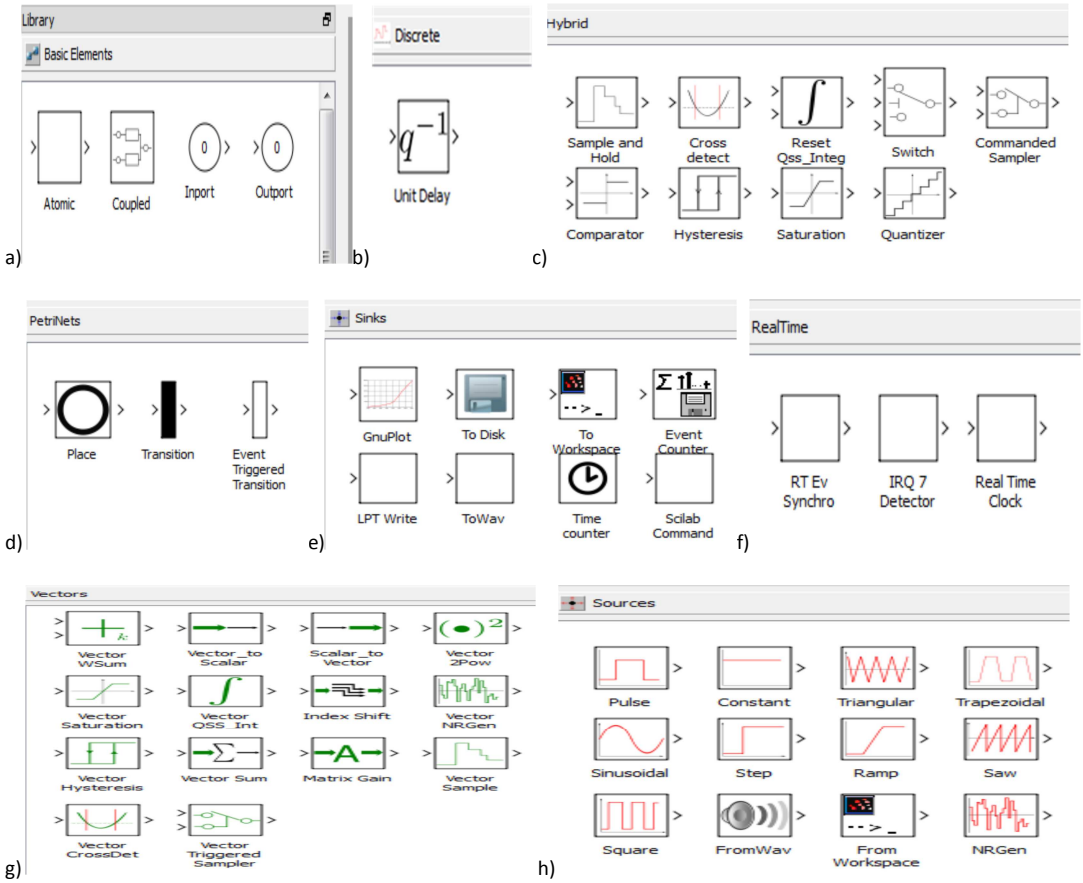


Figura 2.7: a)Basic Elements, b)Discrete, c)Hybrid, d)PetriNets, e)Sinks, f)RealTime, g)Vectors, h)Source

Basic Elements: Como se aprecia en la imagen Figura 2.7 (a), está compuesto de cuatro elementos, que son los más importantes, pues se trata de las plantillas que utilizaremos para crear nuevos elementos, que serán los nuestros propios y que podremos guardar tal y como explicaremos más adelante.

Estos elementos, dada su importancia son explicados y desarrollados en los apartados siguientes, por lo cual no vamos a incidir de momento sobre los mismos. Continuamos por tanto con el siguiente grupo que aparece en la interfaz con el nombre “Discrete".

Discrete: En este menú de librerías, Figura 2.7 (b), se nos muestran los elementos cuyo uso, está relacionado con el manejo de señales discretas.

Hybrid: Se nos muestra los elementos que pueden ser usados tanto para señales continuas como discretas tal y como se aprecia en la Figura 2.7 (c).

PetriNets: Elementos para su uso en redes de Petri, en la Figura 2.7 (d), se nos muestra los elementos que se encuentran en dicha librería.

Sinks: Aquí encontramos agrupados, aquellos elementos disponibles para visualización de señales y valores de entrada y/o salida; en la Figura 2.7 (f), se nos muestran los elementos que conforman el grupo.

RealTime: Elementos para manejo de señales de tiempo, como vemos en la Figura 2.7 (e).

Sources: Se nos muestran en la Figura 2.7 (g) los elementos que podemos usar como generadores de distintos tipos de señal (cuadrada, rectangular, triangular, etc.).

Vectors: Se nos muestran en la Figura 2.7 (h), los elementos disponibles para el manejo de vectores.

Tras esta descripción somera de las librerías disponibles desde la interfaz, procederemos a retomar la explicación del primer grupo denominado “Basic Elements”.

2.6 DESCRIPCIÓN DE MODELOS ATÓMICOS

Entendemos que por su importancia, se debe explicar más detenidamente los elementos básicos, y en concreto el elemento “atomic”.

Estos elementos básicos, son la plantilla sobre la cual podemos crear nuestros propios elementos. Recordemos, que existen en las librerías unos elementos predefinidos, pero obviamente estos no cubren todas las necesidades, por lo tanto disponemos de una plantilla, sobre la cual podemos definir e implementar los elementos que deseemos, ajustándolos así a nuestras necesidades.

Mediante los elementos básicos, podremos crear nuestras propias librerías de elementos, que podrán ser incluidos dentro de las clases o tipos que viene predefinidos y que podemos ver en la interfaz, o bien agruparlos en directorios creados por nosotros para tal fin.

Basic Elements/ atomic: comenzaremos clicando sobre el elemento y arrastrándolo a la plantilla del nuevo proyecto, que está a la derecha de las librerías tal y como se aprecia en la Figura 2.8.

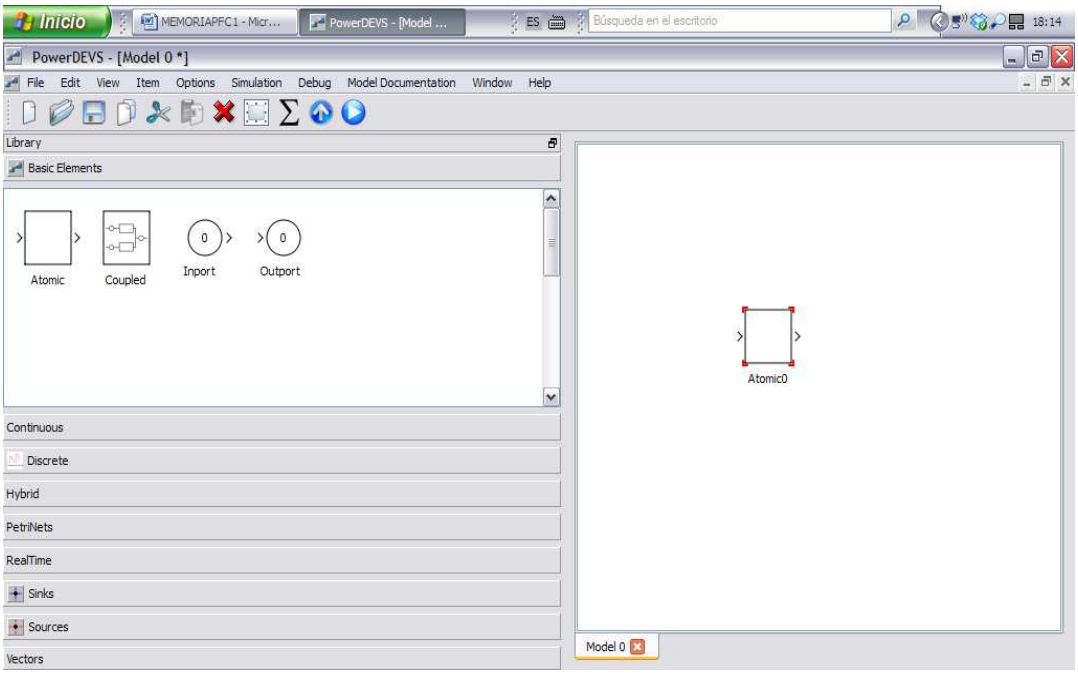


Figura 2.8: Pantalla con un elemento insertado en la plantilla del entorno PowerDEVS

Al pulsar sobre el elemento "Atomic0" con el botón derecho, se nos desplegara el menú tal y como se muestra en la Figura 2.9.

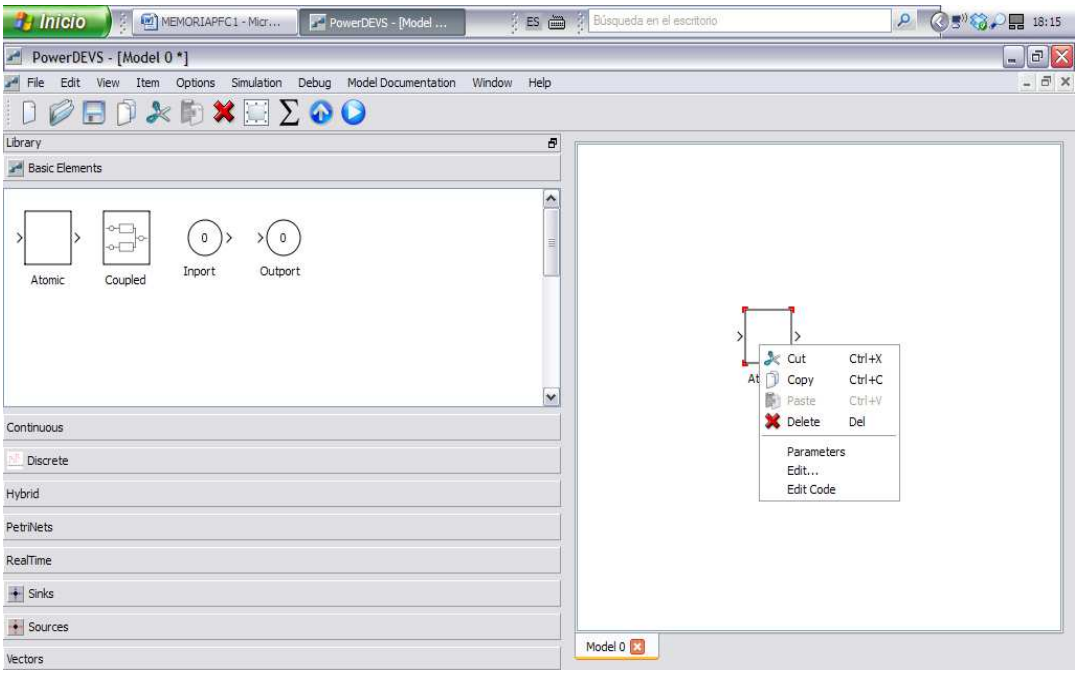


Figura 2.9: Pantalla con el menú disponible de un elemento insertado en la plantilla del entorno

Del menú, entendemos que no necesita explicación las funciones “Cut, Copy y Delete”, pues son las habituales, de los distintos sistemas operativos, por lo que pasaremos a describir las restantes, comenzaremos por “Edit…”, que nos mostrara la pantalla que se muestra en la Figura 2.10, tras clicar sobre él.

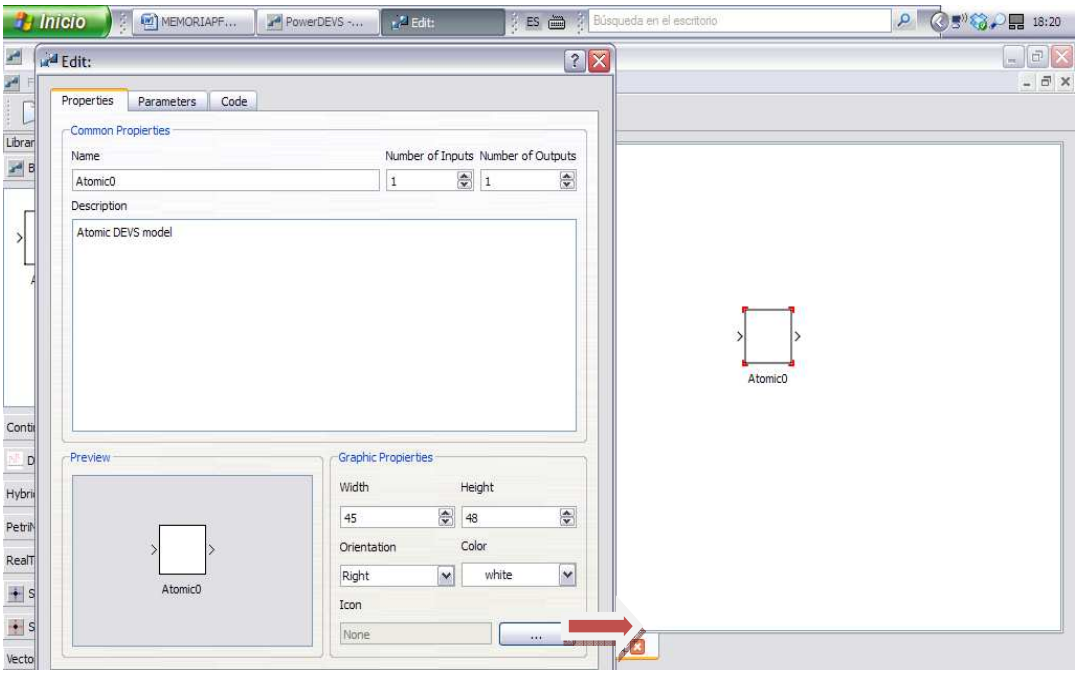


Figura 2.10: Pantalla con la opción “Edit” del menú del elemento insertado en la plantilla

Vemos en la Figura 2.10, que el submenú dispone de tres pestañas, la primera que es la que se ve en la imagen. Se refiere principalmente, a lo que se va ver del elemento es decir a la imagen que veremos del mismo. En ella podemos especificar el nombre del elemento, el número de puertos de entrada y el de puertos de salida. En la casilla central, podemos realizar una descripción del elemento, en la parte inferior podemos cambiar tanto el tamaño del icono como su color y orientación.

También se podrá seleccionar la imagen del icono al pulsar sobre el botón inferior derecho nominado "….." que nos llevará a un menú de exploración, donde se podrá seleccionar la imagen que veremos del elemento.

La siguiente pestaña es la nominada como "Parameters", en ella se nos propone un menú, que nos permitirá establecer los parámetros que nos sean necesarios. Cabe recordar, que los parámetros podrán formar parte de las funciones de transición y que se podrán establecer sus valores al inicio de cada simulación, sin necesidad de que accedamos a cambiar el código del elemento. Como ejemplo vamos por tanto a establecer uno, pulsando sobre "new" tal y como se aprecia en la Figura 2.11.

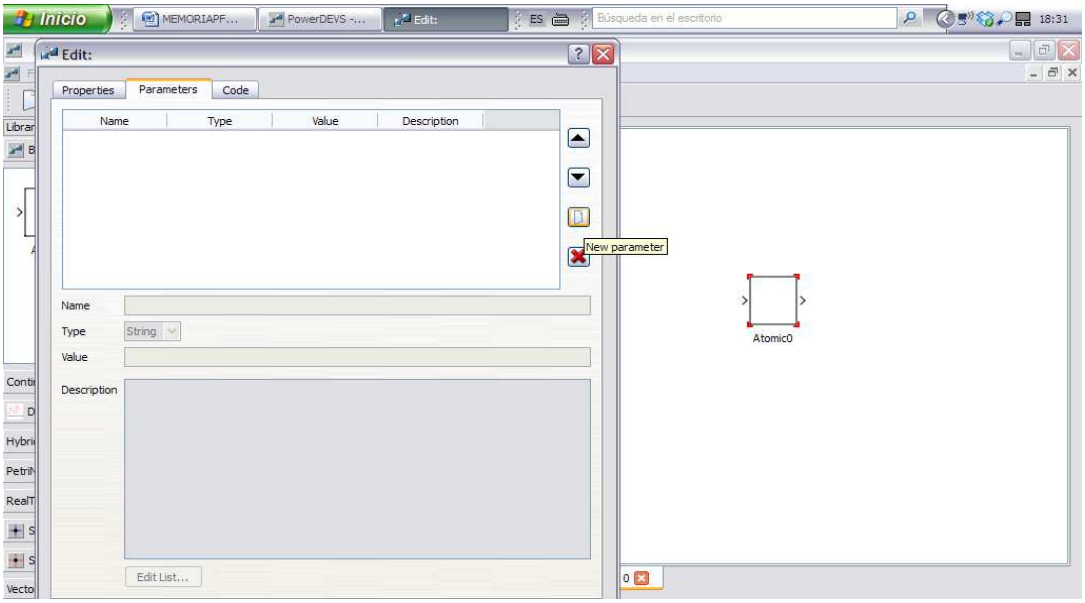


Figura 2.11: Pantalla con la pestaña "Parameters" del menú "Edit"

Establecemos un parámetro que nominaremos como "parametro1", del tipo "Value" y con valor inicial "0". En la Figura 2.12, se aprecia el parámetro ya definido.

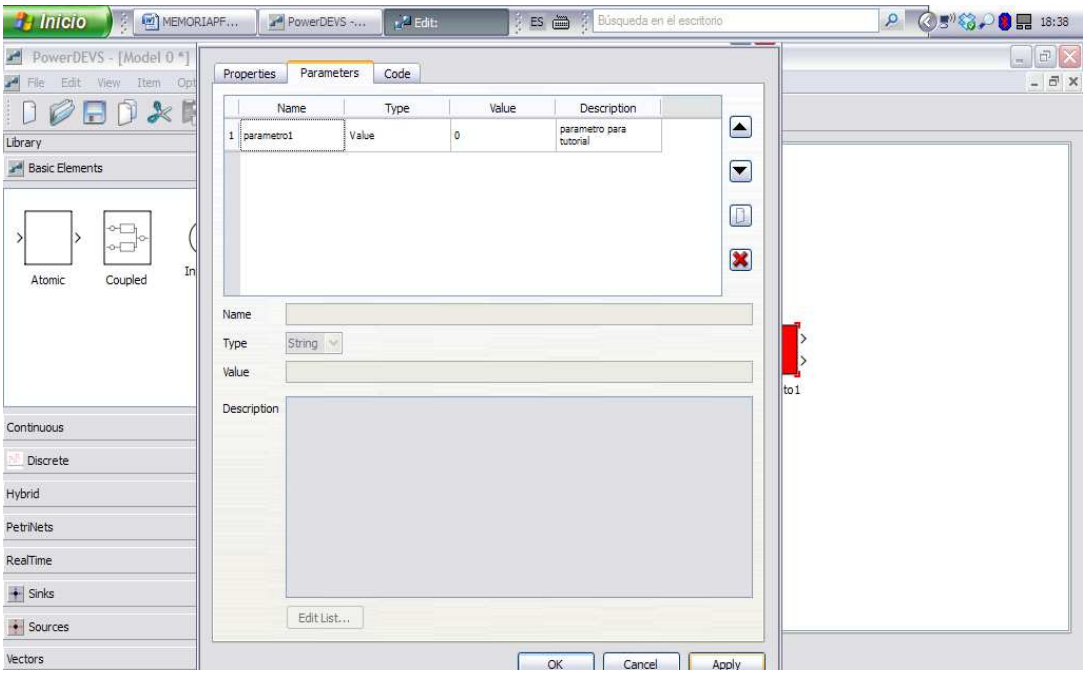


Figura 2.12: Pantalla con la pestaña "Parameters" del menú "Edit" donde se ha establecido un parámetro

Al pulsar sobre el botón nominado como "Apply", éste se incorporara al código de nuestro elemento. La siguiente pestaña es la nominada como "Code", al seleccionarla se nos muestra una pantalla similar a la que se muestra en la Figura 2.13.

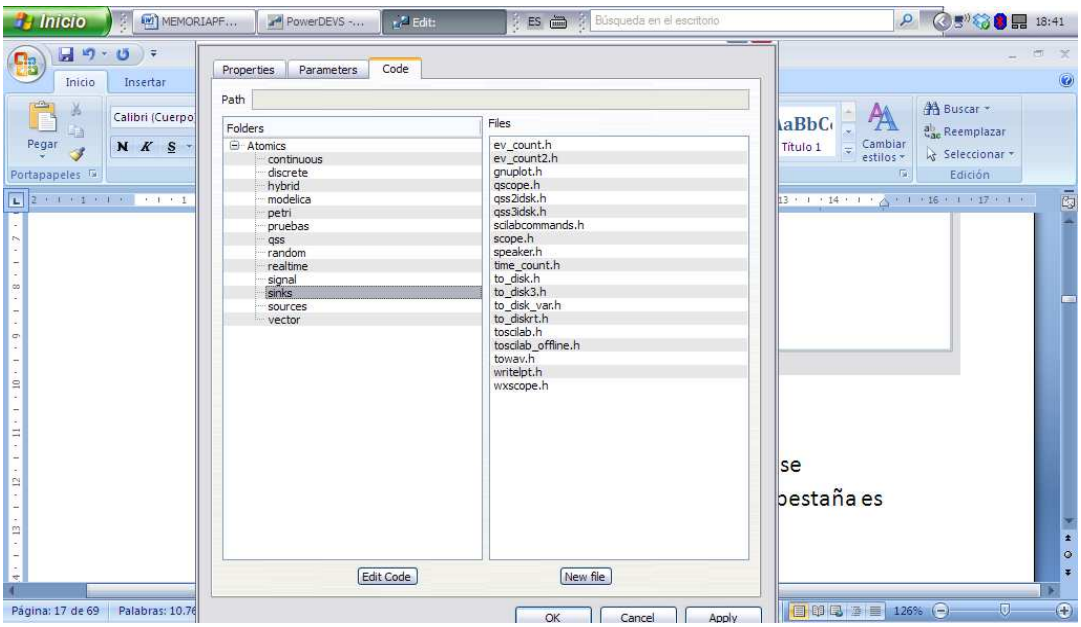


Figura 2.13: Pantalla con la pestaña "Code" del menú "Edit"

La utilidad de este submenú, es la de crear código C++ o la de aplicar uno ya existente al elemento que hemos creado. Vamos por tanto a mostrar ambas opciones.

La primera opción, consiste en aplicar el código de una librería que está creada ya, a nuestro elemento; para ello se nos muestra agrupadas en la parte izquierda por tipos (se corresponden con carpetas físicas) y al pulsar sobre uno de estos elementos, se nos mostrara a su derecha, los archivos de descripción (.h) que están disponibles tal y como se ve en la Figura 2.14. Si deseamos incorporar dicho código a nuestro elemento, lo deberemos seleccionar y luego pulsar sobre "Apply", tras lo cual quedará incorporado a nuestro elemento (Atomic0, en nuestro caso).

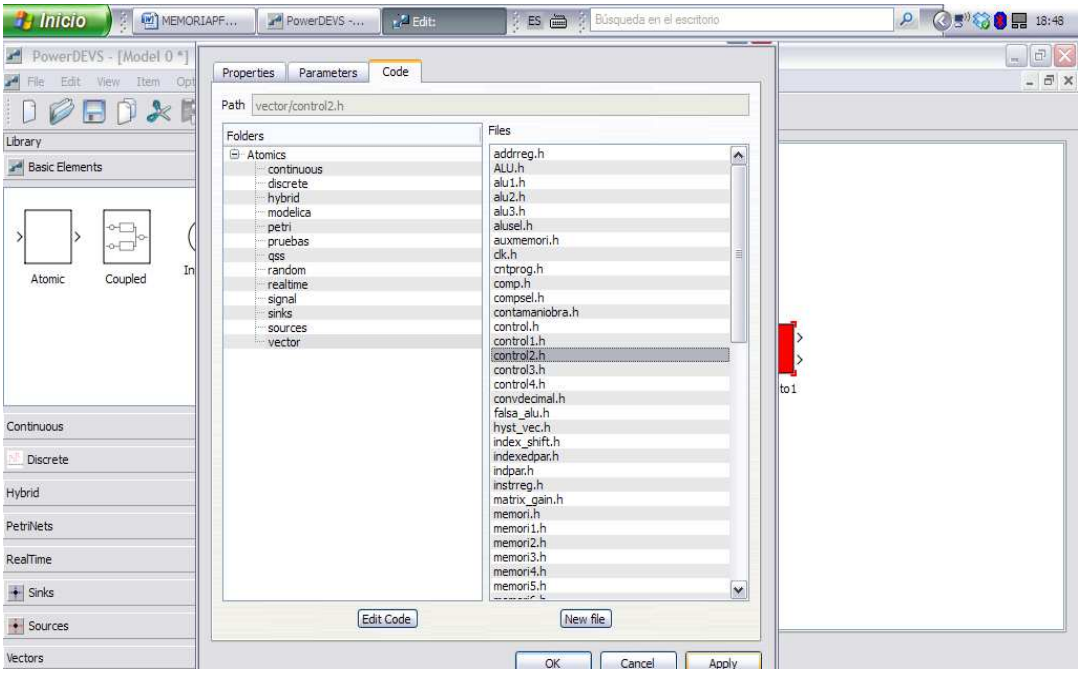


Figura 2.14: Pantalla con la pestaña "Code" del menú "Edit" donde ya se ha establecido el Path

Observemos en la Figura 2.14, que se ha establecido el "Path" al inicio de la pestaña, en nuestro caso "vector/contro2.h". Si deseamos visualizar el código del

archivo seleccionado deberemos pulsar sobre el botón nominado como “Edit Code” y se no mostrara una pantalla similar a la de la Figura 2.15, que explicaremos más adelante.

Figura 2.15: Pantalla en la que se muestra el código del archivo establecido en el Path

Debemos observar, que si necesitamos realizar algún cambio sobre el código de nuestro elemento, las modificaciones que hagamos sobre éste, se aplicaran a cuantos elementos utilicen el mismo código. Por tanto, si deseamos realizar cambio en el código, será conveniente establecer un nuevo código, pulsando sobre el botón nominado como “New file”, con lo cual crearemos un nuevo archivo de definición (.h) y de código (.cpp).

El archivo que hemos creado, lo podremos aplicar a nuestro elemento pulsando sobre el botón “Apply”. Este caso sería el comentado anteriormente, para aplicar un nuevo código a un elemento y que una vez realizados estos pasos nos aparecerá una pantalla como la de la Figura 2.16.

Esta pantalla, dispone del código base para crear un nuevo elemento, esta pantalla es la misma que se nos mostrara tanto para la creación de nuevos elementos, como para los ya existente.

Observamos que en la pantalla, se dispone de dos zonas una a la izquierda, (parte declarativa), en la que se nos indica, que deberemos declarar ahí tanto las variables como los parámetros y las variables de estados, lógicamente también las constantes (todo lo declarado en esta zona aparecerá en el archivo de declaración (".h").

La zona de la derecha, es la parte donde deberemos colocar el código, observando que cada pestaña pertenece a una función diferente dentro del archivo ".cpp". El código que coloquemos en la pestaña "External transition" se integrará dentro de una función con nombre "dext", en el archivo que se creará al compilar el elemento.

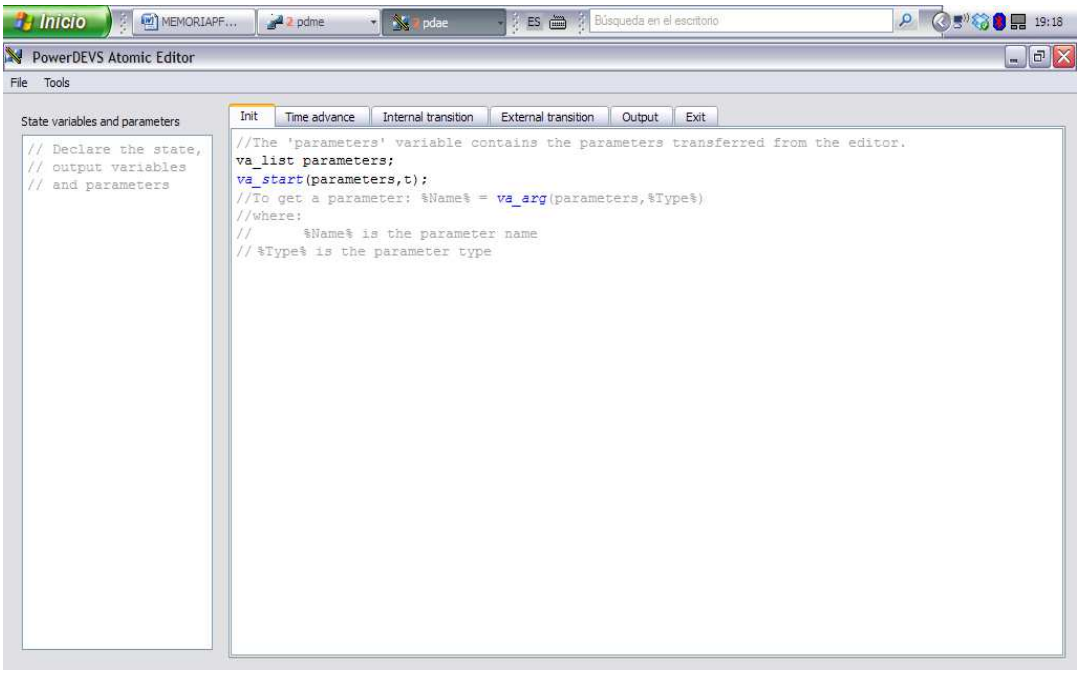


Figura 2.16: Pantalla en la que se muestra la plantilla para crear un nuevo archivo

Quizá la forma más fácil de explicar cómo se crea o modifica el código de un elemento, sea mediante un ejemplo, lo cual pasamos a explicar en la sección siguiente.

2.7 EJEMPLO DE DESCRIPCIÓN DE MODELO ATÓMICO

Vamos por tanto, a implementar un ejemplo sencillo, con el que poder ilustrar la forma de desarrollar nuevos elementos. Para dicho menester, hemos elegido un ejemplo que llamaremos “tutorial”, éste se compone principalmente de un elemento que hemos denominado “normaeventos”.

Este elemento, tiene como función principal, el normalizar el valor de los eventos de entrada, a un valor que predefiniremos mediante un parámetro, en cada simulación.

Es decir, sea cual sea el valor del evento de entrada, en su salida correspondiente se entregará un valor fijo de salida para todos los eventos registrados. Por tanto normalizaremos el evento de entrada a un valor fijo de salida, que será independiente del valor de entrada del evento.

Para crear el nuevo modelo atómico, crearemos un nuevo proyecto al que llamaremos “tutorial”. Una vez creado el proyecto pasaremos a definir el elemento al que denominaremos “normaeventos”.

Para ello comenzaremos clicando sobre el elemento “Basic Elements / atomic” y arrastrando el icono hacia la parte derecha de la interfaz.

Una vez tengamos el icono de “Atomic” en nuestro nuevo proyecto, procedemos a definir el elemento. Comenzaremos por darle un nombre, en nuestro caso “normaeventos”.

A continuación, definiremos un parámetro que llamaremos “valor”, mediante el cual normalizaremos el valor de salida de los eventos, además fijaremos el nº de puertos de entrada y salida. Dos en nuestro caso, tanto para las entradas como para las salidas.

Procederemos a crear el código que soporta estas funciones, por el procedimiento expresado en el apartado anterior para crea un nuevo código.

Una vez realizados los pasos anteriores, se no presentará una pantalla similar a la que se muestra en la Figura 2.17, en la que se muestra la declaración de variables y la implementación del parámetro de entrada valor.

Figura 2.17: Pantalla en la que se muestra la plantilla para crear el elemento “normaeventos”

Se puede apreciar, que en la parte izquierda de la Figura 2.18, es la parte declarativa del elemento. Lugar donde se declaran las variables y constantes que formaran parte del elemento.

En el recuadro de la parte derecha se puede apreciar, en la pestaña “Init”, que es donde procedemos a recoger el valor del parámetro de entrada. En la misma plantilla, se nos indica como comentario, la forma en que podemos obtener el valor de un parámetro que definamos en la interfaz.

La explicación de cómo hacerlo, se nos indica en los siguientes comentarios extraídos de la plantilla de elemento atomic; “//To get a parameter: %Name% = va_arg (parameters, %Type%)”. Por tanto tal y como se aprecia en la Figura 2.17 para recuperar el valor del parámetro definido en la interfaz, simplemente declaramos un puntero de tipo “char” y con el método “getscilabVar” obtenemos el valor del parámetro que hayamos ingresado a través de la interfaz.

Ambos métodos son heredados y por tanto ya están definidos y simplemente, debemos usarlos en la forma en que se nos indica en los comentarios.

A continuación mostramos el resto de pestañas, que tal y como se ha comentado corresponde a la traslación de código, al formalismo DEVS. En el Capítulo 5 realizamos una explicación más profunda de dicha traslación de código, por lo que nos limitaremos a describir superficialmente dicha traslación.

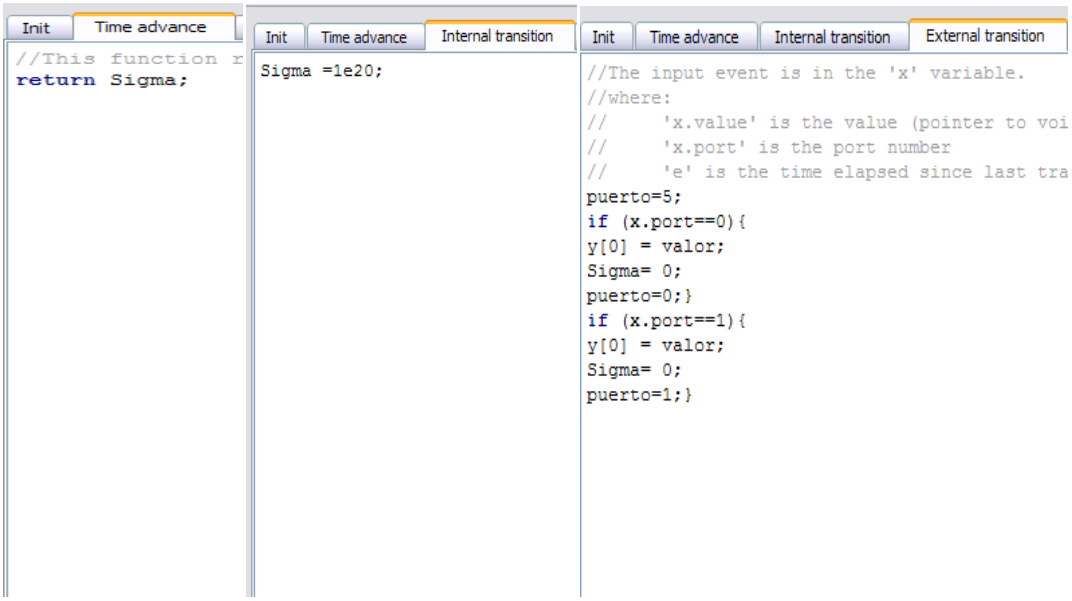


Figura 2.18: Pestañas “Time advance”, “Internal transition” y “External transition”

En la Figura 2.18, se puede observar el código que corresponde a cada pestaña. Destacar que el nombre que aparece en cada pestaña, se traducirá en el código fuente de cada función, que tiene correspondencia con la misma función en DEVS. Es decir el código que escribimos en la pestaña “Internal Transition”, aparecerá en el archivo del elemento dentro de una función que se llama “dint” y que se encarga de realizar las transiciones internas correspondientes, siguiendo el formalismo DEVS.

En la pestaña “External transition” (Figura 2.18 derecha) ocurrirá lo mismo, el código que escribamos aparecerá dentro de la función “dext” del código fuente. Recordemos que la función de transición externa responde a los eventos externos recibidos por los puertos de entrada del elemento.

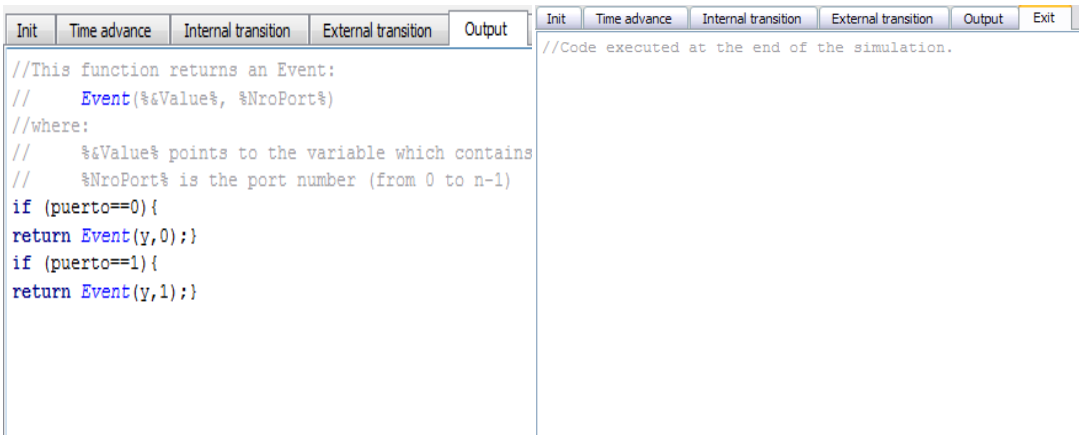


Figura 2.19: Pestañas “Output” y “Exit”

En cuanto a la pestaña “Output” (Figura 2.19, Izquierda), poco hay que explicar, pues como se puede apreciar la función de salida encamina el valor normalizado del evento, a la salida con el mismo índice que la entrada. Es decir, los eventos que han entrado por el puerto 0, salen por el puerto 0 y los que han entrado por la puerto 1 salen por el puerto 1; eso sí, con un valor normalizado y que hemos definido en el parámetro de entrada “valor”.

En la Figura 2.19 (derecha), se aprecia el código que implementara la función “Exit” como una pestaña denominada como “Exit”. Explicaremos que la pestaña "Exit" no tiene correspondencia directa con el formalismo DEVS tal y como sí ocurre con las pestañas, “Time Advance”, “Internal Transition”, “External Transition” y “Output”.

La función de dicha pestaña, es la de ejecutar acciones que pudieran quedar pendientes tras la ejecución del código del elemento, como podría ser por ejemplo, cerrar un archivo que previamente se ha abierto por parte del elemento. Por supuesto una vez utilizado, procederemos a cerrar, mediante el código que implementemos en la pestaña “Exit”. En nuestro ejemplo no necesitamos de ejecutar nada en dicha pestaña.

Una vez hemos escrito el código fuente que deseamos implementar, en las pestañas correspondientes, al pulsar sobre el menú “Tools/Check”, nos pedirá que guardemos el archivo en una carpeta antes de proceder a su compilación tal y como se muestra en la Figura 2.20. En nuestro caso guardamos en la carpeta denominada “tutorial”.

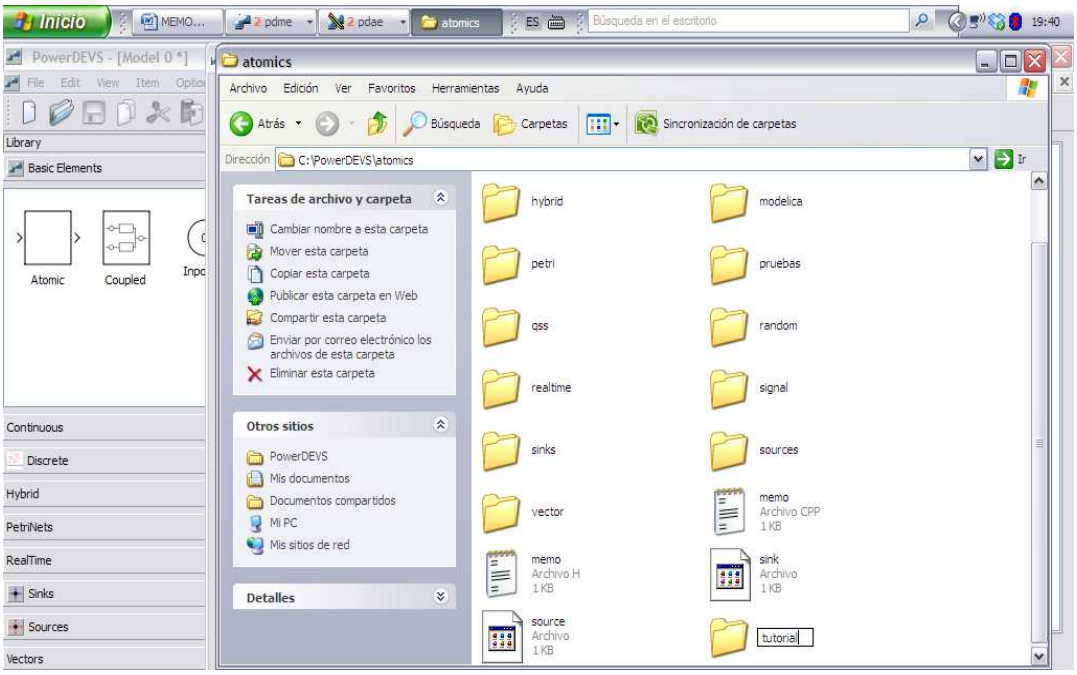


Figura 2.20: Pantalla en la que se ve como guardamos el archivo que hemos creado

Una vez creada la carpeta, donde guardaremos los nuevos archivos que creemos, podremos proceder a realizar la compilación del nuevo archivo. En este caso hemos introducido un error a propósito, tal y como se nos muestra en la Figura 2.21 (derecha), donde se nos indica que existe un error de compilación. Frente a la imagen de la parte izquierda, que nos muestra el mensaje de compilación correcta.

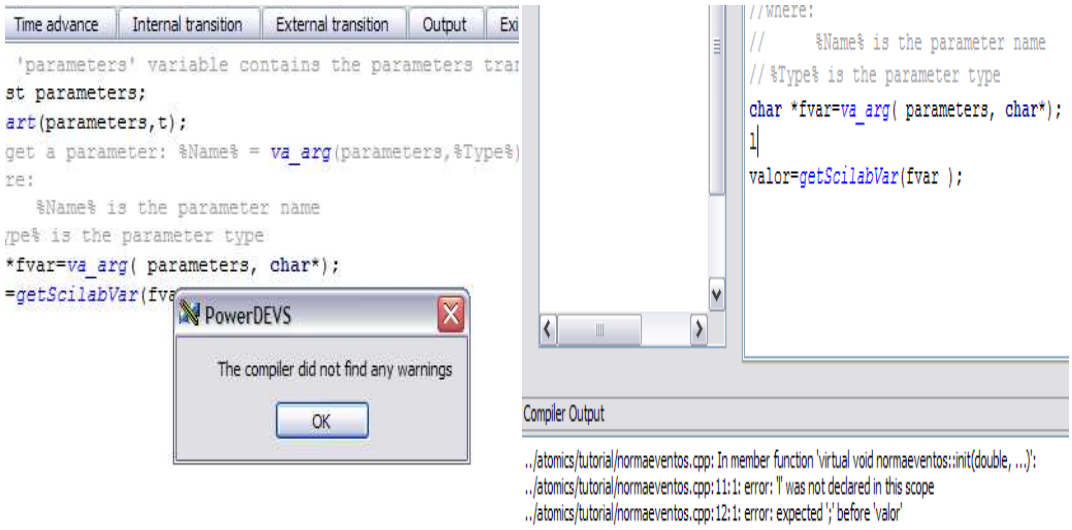


Figura 2.21: Resultado de compilar el archivo correctamente e incorrectamente (derecha)

Conexión de componentes: Los distintos elementos individuales se pueden conectar entre sí para formar un elemento de orden superior.

La conexión entre elementos es muy sencilla, simplemente se trata de pulsar con el botón izquierdo, sobre un puerto de entrada o salida y sin dejar de pulsar el botón, arrastrar el ratón hasta el otro extremo que deseamos unir (veremos que aparece una línea de trazos discontinuos), una vez situado en el otro extremo, al soltar el botón quedara establecida la conexión (con una línea de trazo continuo).

La conexión también se puede establecer entre líneas de conexión y puertos, no necesariamente entre puertos. En este caso, cuando la conexión se realiza entre líneas, veremos que aparece un punto en el lugar de conexión lo cual confirma que la conexión se ha efectuado; si en lugar del punto lo que aparece es una cruz, indicara que no existe conexión ente una línea y otra.

El programa realiza una traza de la línea de conexión, una vez establecida por nosotros, podemos modificar la traza simplemente pulsando con el botón izquierdo sobre la traza y sin dejar de pulsar, al arrastrar el ratón en cualquier dirección, veremos como la traza de la conexión, se desplaza en el mismo sentido que el desplazamiento del ratón, sin alterar los punto inicio y fin de la conexión.

En el caso, que lo que deseemos sea borrar la línea de conexión, simplemente deberemos seleccionarla, pulsando sobre ella con el botón izquierdo, tras lo cual veremos que aparecen unos pequeños puntos rojos en los cambios de dirección de la traza (esto indica que se encuentra seleccionada) y pulsando sobre el botón de cabecera nominado como "X" ("delete" al situar el ratón sobre él) desaparecerá la línea de conexión que deseábamos borrar.

Mostramos en la Figura 2.22 la imagen de nuestro ejemplo ya conectado y a punto de realizar la simulación.

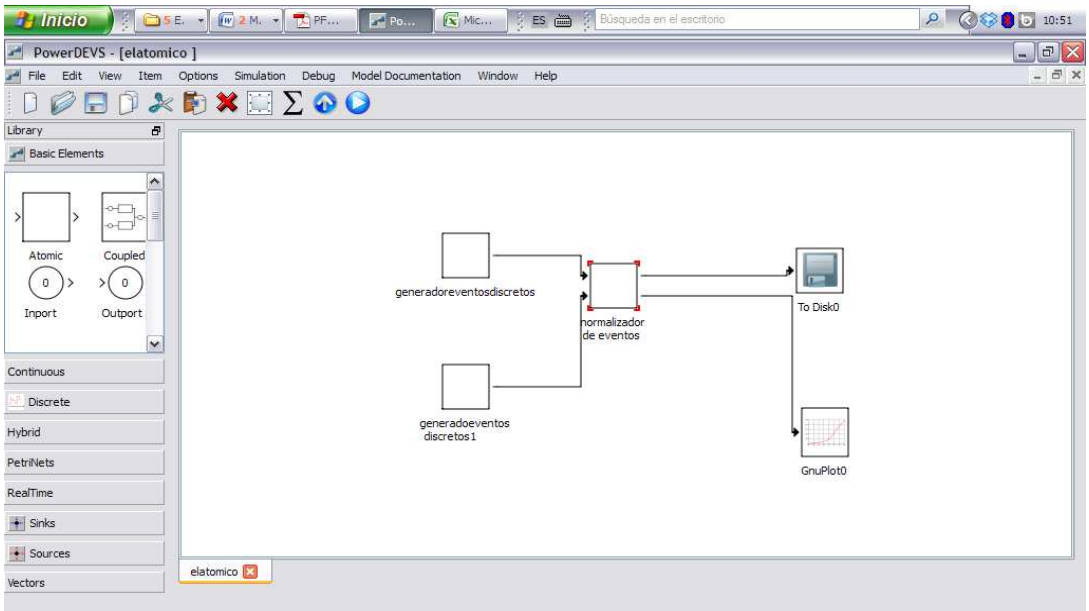


Figura 2.22: Pantalla conexión de los elementos que componen el ejemplo

2.8 DESCRIPCIÓN DE ELEMENTOS ACOPLADOS

Otro elemento que nos encontraremos en la carpeta “Basic Elements” es el elemento “Coupled”. Para realizar una explicación del mismo, procederemos a seleccionarlo y arrastrarlo a la plantilla de nuestro proyecto, situándolo en el lugar que deseemos. Tras pulsar con el botón derecho sobre el elemento se nos despliega el siguiente menú tal y como se nos muestra en la Figura 2.23.

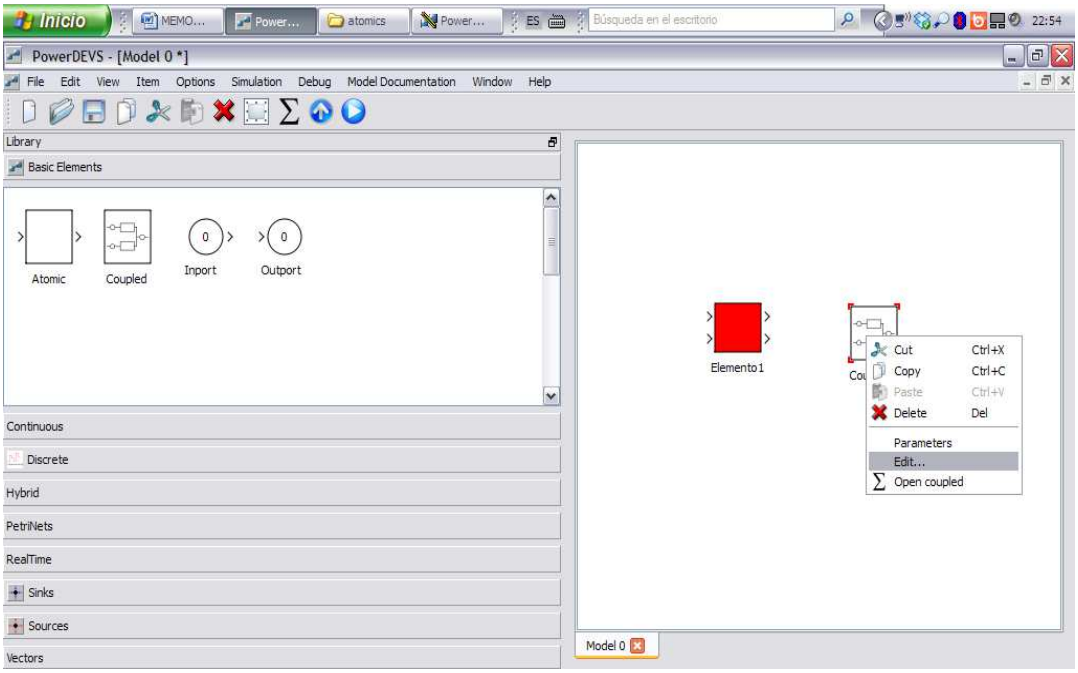


Figura 2.23: Pantalla de menú para un elemento compuesto

Como podemos observar, el menú es muy similar al menú obtenido cuando hemos realizado la misma operación sobre el elemento “atomic0”. Destaca una única diferencia visible, y es el último elemento del menú nominado como “Open coupled”, que pasaremos a explicar a continuación.

Tras pulsar sobre esa parte del menú, nos aparece la siguiente pantalla.

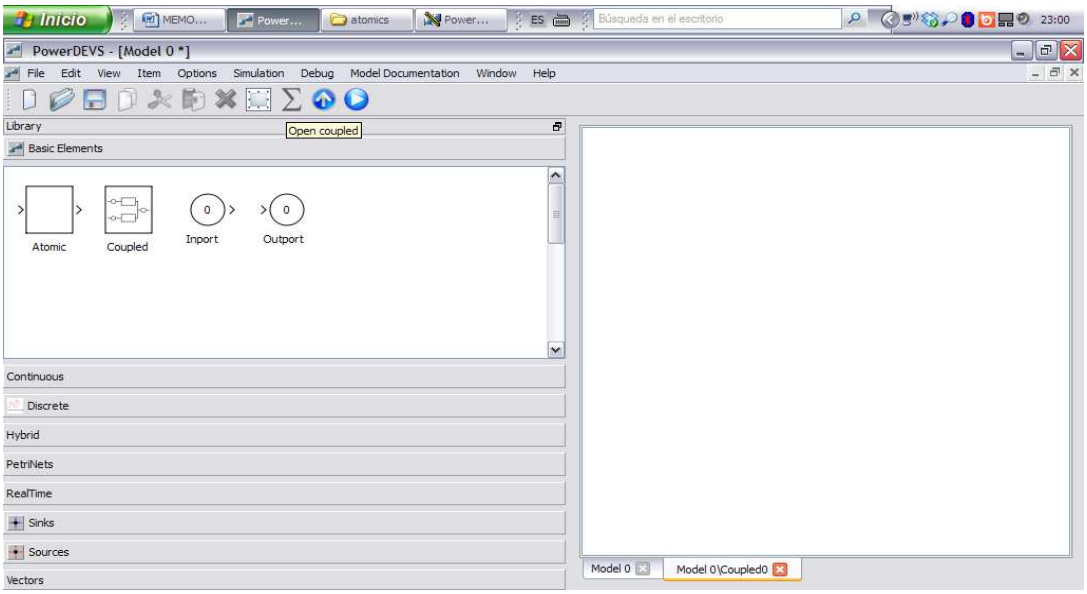


Figura 2.24: Pantalla de la plantilla de un elemento compuesto

Como se puede apreciar en la Figura 2.24, se nos vuelve a abrir una nueva pestaña, que es un subprograma del anterior tal y como se indica en la pestaña "Model0\Coupled0", este elemento tiene como utilidad, el posibilitar agrupar una determinada cantidad de elementos más o menos complejos, bajo la interfaz de un solo elemento.

Se nos mostrara en la plantilla general, como un único elemento; se trata pues, de una manera de agrupar unos elementos, que unidos, realizan una única tarea más general y que al mostrarse como un único elemento, simplifica el esquema general.

A modo de información, adjuntamos una imagen Figura 2.25, de un circuito donde se hace uso de este elemento, sin pasar a más explicaciones, pues dentro de esta pestaña el programa se comporta como si fuera la pestaña principal y se trabaja con ella de la misma manera.

El uso de los elementos “Inport” y “Outport” como su propio nombre indica son puertos de entrada y salida que se pueden utilizar como entradas generales de un subsistema o de un elemento.

El subelemento “copled0” sería el que se ve en la figura (no tiene ninguna función específica, en este caso solamente se trata de una visualización de ejemplo).

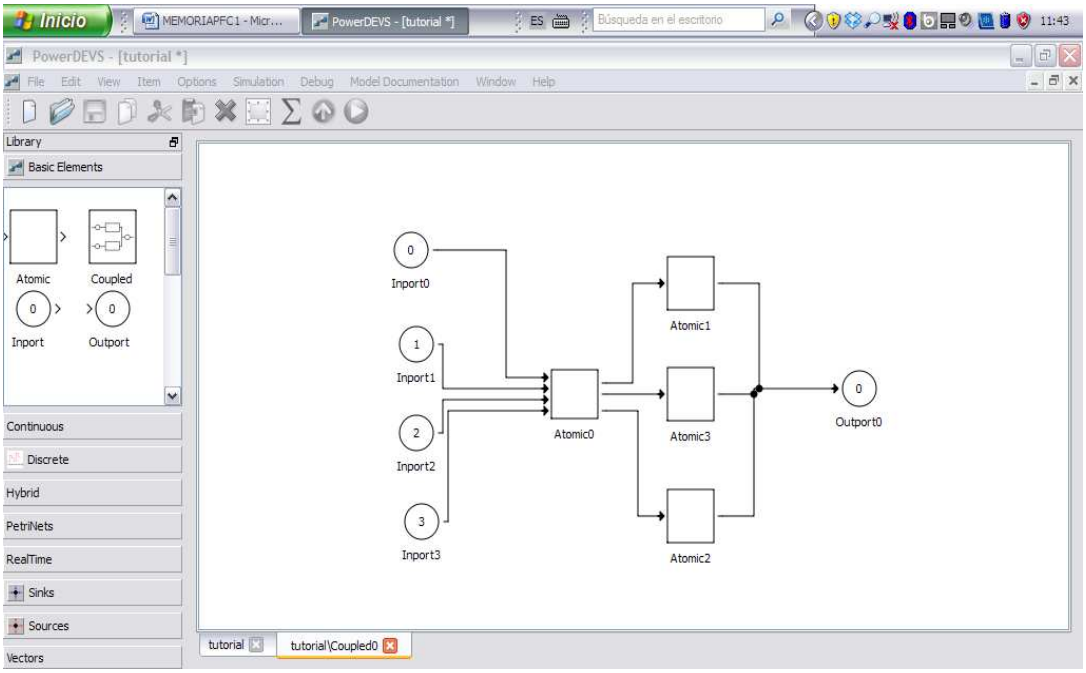


Figura 2.25: Pantalla de la plantilla de un elemento compuesto, integrado a su vez de elementos simples

Sin embargo en la 1ª plantilla Figura 2.26, solo veríamos el elemento “coupled0” con cuatro entradas y una salida quedando oculta la implementación del elemento “coupled0”.

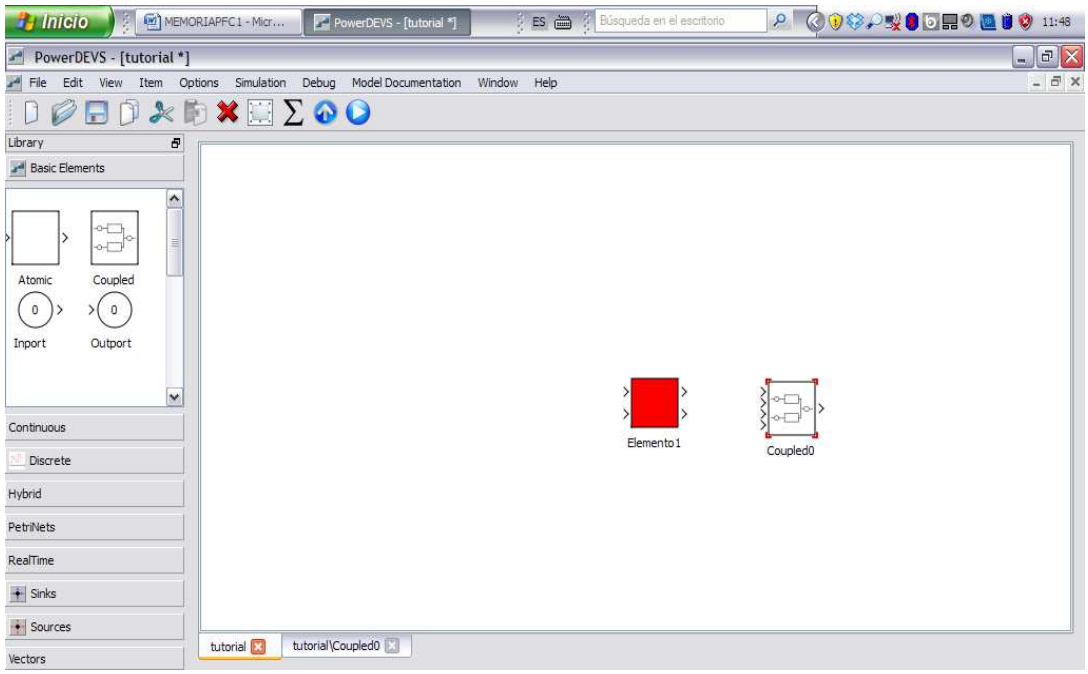


Figura 2.26: Pantalla del proyecto con un elemento simple (rojo) y un elemento compuesto

Del menú del elemento "Coupled", Figura 2.27, cabe destacar que el menú "Edit" no dispone de la pestaña "Code" pues en el elemento "coupled", no se inserta ningún código, solo se trata de la interfaz que mostrara el elemento, el código ira instalado en los elementos de la plantilla "tutorial/coupled0".

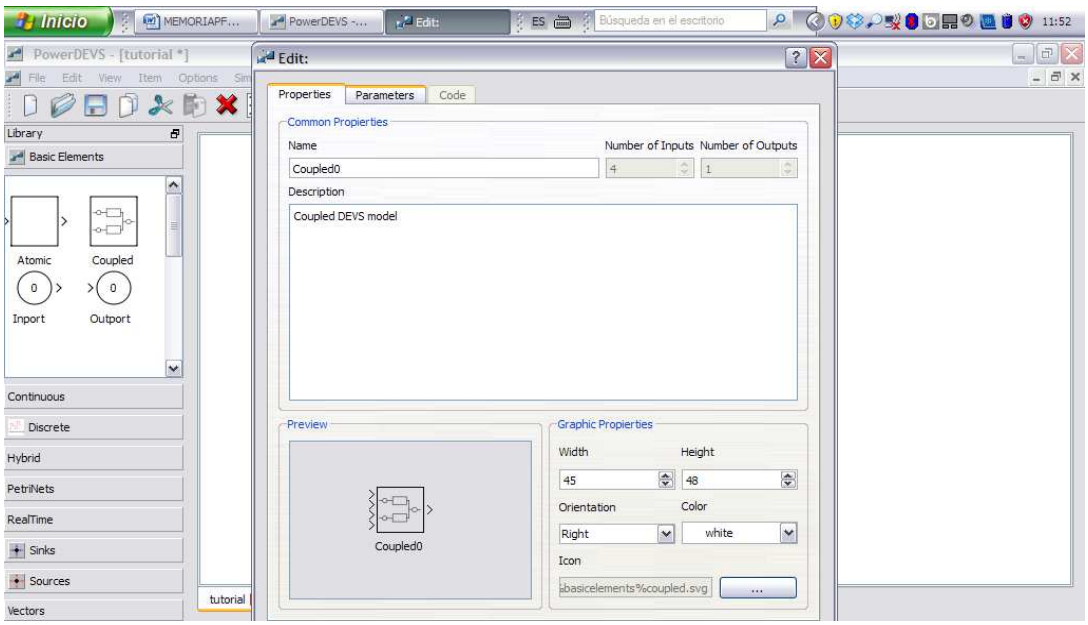


Figura 2.27: Pantalla del menú "Edición" para el elemento compuesto

2.9 SIMULACIÓN

Para pasar a explicar cómo realizar la simulación, nos ha parecido interesante completar el ejemplo inicial con más elementos intentando a la vez, conjugar la sencillez del ejemplo con el usar un máximo de las funciones que hemos explicado.

El esquema que proponemos, se compone de nuestro "normaeventos" que es el elemento que hemos creado y descrito en la Sección 2.7. A demás, hemos colocado dos generadores de eventos discretos a las entradas, en las salidas hemos colocado dos elementos para poder visualizar los valores de la salida.

Los elementos conectados a las salidas se diferencian básicamente, en la forma en que nos muestran los resultados.

El elemento "GnuPlot", presenta un gráfico donde se nos muestran los valores que va tomando las salidas en cada instante de tiempo de simulación.

En cambio "To Disk", muestra los valores (pares, tiempo-valor) en un archivo Excel, que encontraremos en la ruta PowerDEVS\output\out.exc.

Pasamos a mostrar la imagen del esquema en la Figura 2.28, donde se aprecian los elementos que se han empleado en nuestro ejemplo.

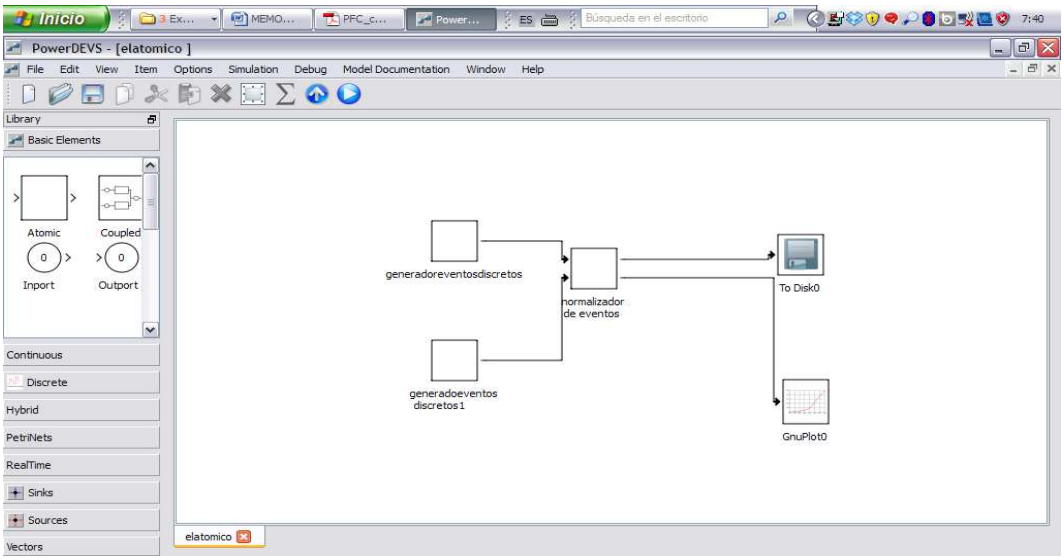


Figura 2.28: Elementos que configuran nuestro proyecto ejemplo

Antes de iniciar la simulación, pasaremos a ajustar los parámetros de los distintos elementos, de la siguiente manera. Pulsando con el botón derecho sobre el elemento, se nos desplegará un menú, pulsando sobre la opción “Parameters” se nos desplegará una pantalla similar a cualquiera de las tres que se nos muestra en la Figura 2.29.

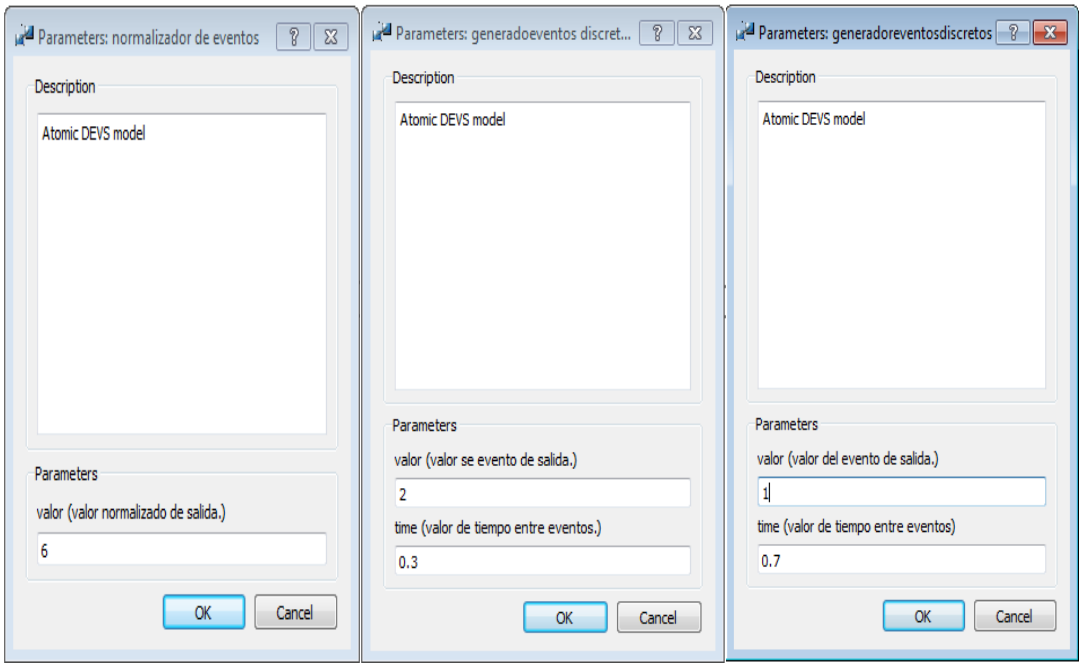


Figura 2.29: Establecimiento de los parámetros para los elementos “normalizador de eventos”,”generadoreventos discretos” y “generadoeventosdiscretos1”

Colocados los valores deseados, en las correspondientes casillas, pulsaremos sobre el botón “OK” con lo que desaparecerá esta pantalla quedando establecidos los parámetros del elemento.

Podemos apreciar en la imagen de la Figura 2.29 (izquierda y centro), que hemos establecido para los generadores de eventos discretos dos parametros, “valor “ y “time”.

la funcion del parámetro “valor” es definir el valor que tomara el evento de la función salida en nuestro caso 2 y 1 respectivamente. La funcion del parametro “time”, es establecer cada cuánto tiempo se generará un evento en nuestro caso hemos elegido los valores 0.3 y 0.7 intervalos de simulacion.

Para el elemento “normalizador de eventos”, hemos establecido el parametro “valor”, a 6 unidades. Por tanto, todos los eventos que se produzcan en las entradas, apareceran en la salida correspondiente, pero con un valor normalizado a 6.

Ya estamos dispuestos para iniciar la simulación pulsando sobre el icono  , o bien en el menú “Simulation->Simulate”.

Nos aparecerá una pantalla similar a la mostrada en la Figura 2.30, donde podemos apreciar que nos aparece un menú emergente, que es el que utilizaremos para realizar la simulación.

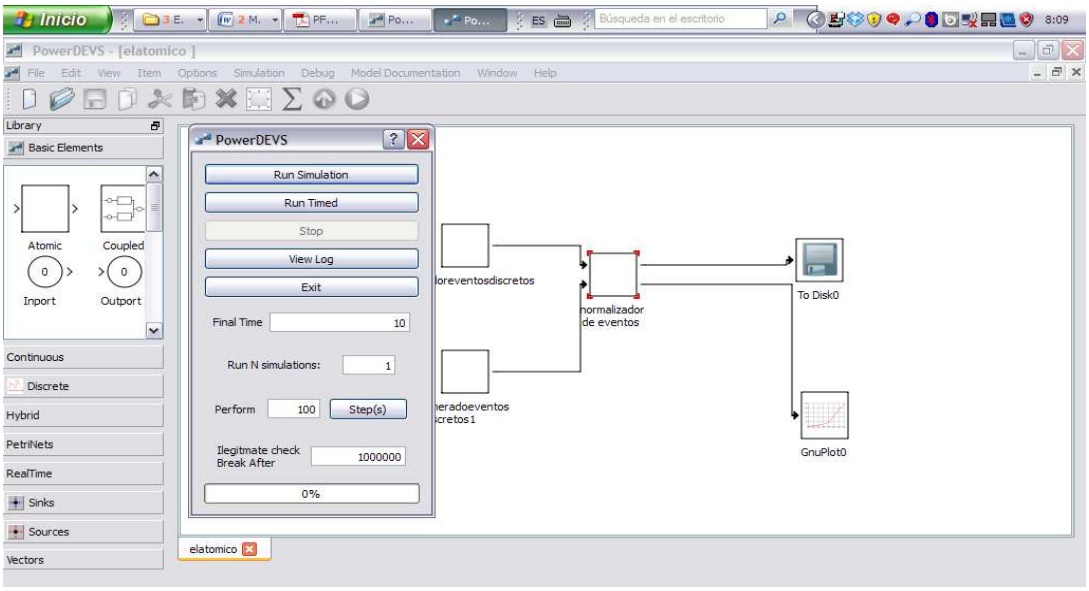


Figura 2.30: Pantalla donde se inicia la simulación paso nº 1

Vamos a explicar el menú que se nos muestra en la Figura 2.30, vemos que disponemos de dos modos de simulación que elegiremos mediante los botones "Run Simulation" y "Run Timed". En ambos casos, la simulación termina cuando el tiempo simulado alcanza el valor del tiempo especificado en la casilla "Final time". La diferencia entre ambas simulaciones, estriba en que, mientras "Run Simulation" realiza la simulación en el mínimo tiempo necesario, mientras que "Run Timed" está sincronizada la simulación con el tiempo real seleccionado en la casilla "Final time".

Más abajo, vemos una casilla nominada como "Run N simulations", donde podemos seleccionar el nº de simulaciones que se va a realizar, aparece una por defecto. La siguiente casilla en orden descendente es la casilla "Perform" donde podemos ingresar el nº de pasos que queremos simular. A su derecha el botón "Step (s)" nos permitirá la simulación del nº de pasos ingresados en la casilla "Perform".

Continuando más abajo, aparece la casilla “Ilegitmate check Break After”, esta casilla nos permitirá ingresar el valor numérico (tiempo), después del cual abortará la simulación, para aquellos casos en que exista algún problema que impida la correcta realización.

Vemos que aparece una casilla al final del menú, donde se nos indica o bien el tiempo de simulación (si elegimos la opción “Run Simulate”), o % de simulación que se ha realizado, si elegimos la opción “Run Timed” o la opción “Step (s)”. El botón “Exit”, nos devuelve a la pantalla anterior y el botón “View Log” nos llevara al archivo de depuración.

Una vez que hemos pulsado sobre el botón “Run Simulate”, se inicia la simulación del esquema implementado. En nuestro caso una vez finalizada la misma nos aparece una imagen como la que nos muestra la Figura 2.31. Esta imagen es el gráfico correspondiente al elemento “GnuPlot0”, cuya función es mostrar en un gráfico, los valores que recibe en su entrada. La grafica en si, no aporta información, pues a nuestro modesto entender no realiza correctamente la función para la que ha sido diseñado, siendo poco útil para la simulación de eventos discretos.

Decir que esta tabla, se confecciona con los datos que el elemento “GnuPlo0”, guarda en un archivo Excel y que encontraremos en la carpeta PowerDEVS\output\plots, si previamente hemos tenido la precaución de limpiarla de archivos, pues en ella se van acumulando todas las simulaciones que se realizan a través de elementos de tipo “GnuPlot”.

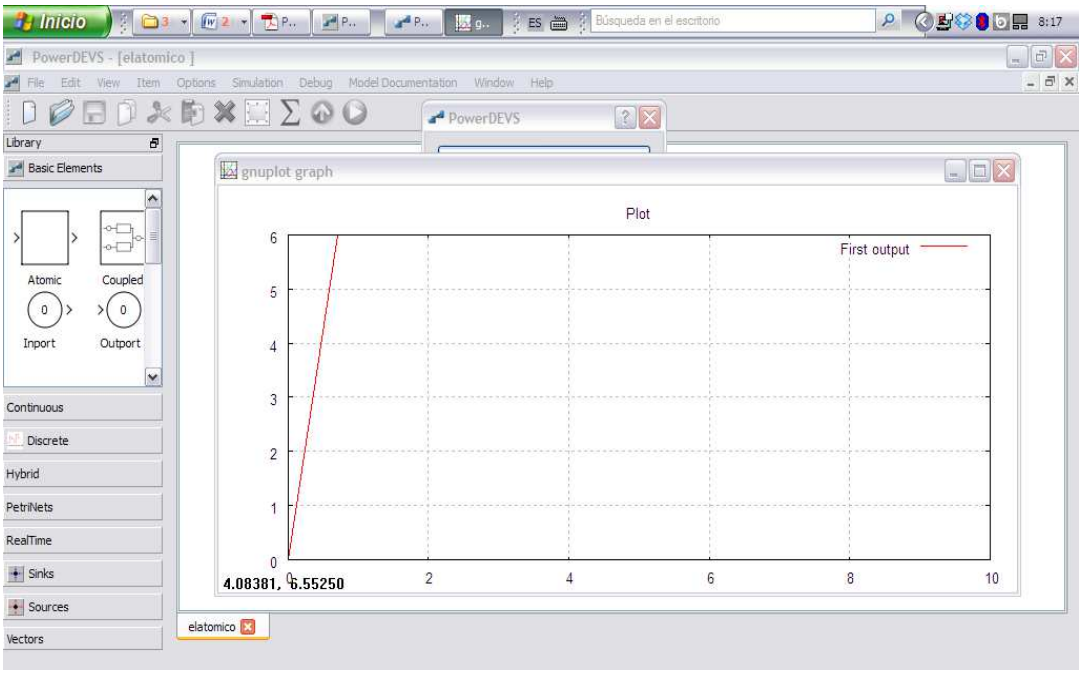


Figura 2.31: Pantalla donde se muestra un resultado de la simulación a través de un grafico

Entendemos que los resultados mostrados a través del archivo Excel, son más descriptivos y útiles que los mostrados a través de la gráfica. Por tanto procederemos a mostrar los resultados obtenidos en las salidas en el modo que se guarda en los archivos Excel, cuyo formato es un par de valores separados por ",". El primer valor indica el instante de tiempo de simulación y el segundo valor corresponde al valor de entrada. Tal y como se nos muestra en la Figura 2. 32.

	A
1	0.7, 6
2	1.4, 6
3	2.1, 6
4	2.8, 6
5	3.5, 6
6	4.2, 6
7	4.9, 6
8	5.6, 6
9	6.3, 6
10	7, 6
11	7.7, 6
12	8.4, 6
13	9.1, 6
14	9.8, 6
15	
16	
17	
18	

	A
1	0
2	0, 6
3	0.3, 6
4	0.6, 6
5	0.9, 6
6	1.2, 6
7	1.5, 6
8	1.8, 6
9	2.1, 6
10	2.4, 6
11	2.7, 6
12	3, 6
13	3.3, 6
14	3.6, 6
15	3.9, 6
16	4.2, 6
17	4.5, 6
18	4.8, 6

18	4.8, 6
19	5.1, 6
20	5.4, 6
21	5.7, 6
22	6, 6
23	6.3, 6
24	6.6, 6
25	6.9, 6
26	7.2, 6
27	7.5, 6
28	7.8, 6
29	8.1, 6
30	8.4, 6
31	8.7, 6
32	9, 6
33	9.3, 6
34	9.6, 6
35	9.9, 6
36	

Figura 2.32: Resultados de la simulación obtenidos en los archivos Excel de salida (izquierda) "Output.xls" salida nº 0; (centro y derecha) salida nº 1 elemento "GnuPlot0"

Entendemos que estos resultados merecen una explicación. Comenzaremos por los resultados de la imagen en la parte izquierda de la Figura 2.32. Explicar que se puede observar con claridad que el valor de todas las salidas es 6, que es el valor al cual hemos normalizado para las salidas (iniciación de parámetros Figura 2.28).

El dato más interesante de lo mostrado en la Figura 2.32 no es tanto el valor de normalización (pues es el esperado), sino el tiempo de simulación, este nos dará pistas sobre lo que está ocurriendo en el circuito.

Como se puede apreciar en la imagen de la izquierda, el tiempo de simulación se incrementa en intervalos de valor 0.7. Lo cual es consistente con la iniciación de parámetros, en concreto del elemento “generador eventosdiscretos” que está conectado al puerto 0. Por tanto entregara su salida por el puerto 0, al que está conectado el elemento “ToDisk0”  , los valores que mostramos se obtienen de la ruta \PowerDEVS\output\output.xls.

En la parte central y derecha de la Figura 2.32, observamos los resultados de la simulación correspondientes a la salida del puerto 1; obtenidos de la simulación mediante el elemento de salida “GnuPlot0”  , que tal y como se comentado con anterioridad, se visualizan a través de una gráfica, pero que también pueden consultarse en un archivo de extensión Excel, como es el caso que se nos presenta. Los resultados obtenidos son consistentes con lo esperado, pues cada 0.3 intervalos de tiempo, el generador de eventos, genera un evento y éste es entregado en la salida, con el valor normalizado 6. Recordar que los parámetros de

iniciación del elemento “generadoreventosdiscretos1” se pueden observar en la Figura 2.29.

2.10 CONCLUSIONES

En este capítulo, hemos tratado de realizar un simple tutorial sobre el entorno PowerDEVS. Entendemos que este tutorial, es sencillo y no cubre todas las posibilidades que nos ofrece el entorno, de alguna manera es lógico que así sea, pues la documentación que hemos encontrado sobre este entorno es muy escasa.

Esto unido, a que no somos los creadores de la herramienta, hace que nuestra visión del entorno no sea completa, sino mas bien parcial. Por tanto el objetivo de este modesto tutorial, se ha basado en aprovechar la experiencia que hemos adquirido en el desarrollo de nuestro PFC. Aportando nuestro conocimiento de la herramienta, que puede ayudar a las personas que deben evaluar el mismo, a entenderlo y poder simularlo, sin necesidad de recurrir a otros textos.

Entendemos por tanto, que aun no siendo exhaustivo el tutorial, si resulta suficiente para acercarnos al funcionamiento del entorno. Además, resultará útil también para poder comprender el trabajo realizado en el PFC, así como para poder probar y evaluar el software desarrollado.

3

DISEÑO VHDL DEL CIRCUITO

3.1 INTRODUCCIÓN

Tras la introducción, que hemos realizado en la Sección 1.2 sobre los objetivos del proyecto, hemos de pasar a una descripción mucho más completa del circuito, que sirve de base para el desarrollo del presente proyecto. Dedicando por completo este capítulo, a sentar claramente las bases, sobre las que debe funcionar el circuito a implementar, así como su descripción mediante VHDL. Esta descripción, constituye la base para la posterior implementación en PowerDEVS así como su posterior validación.

Primero procederemos a explicar cada elemento que compone de manera individual la CPU, para a continuación explicar la conexión de la CPU a la memoria. Por último, procederemos a explicar el funcionamiento general del circuito, mediante la descripción de una instrucción de cada uno de los tipos de instrucción más importantes, con lo que podremos seguir de manera dinámica como se va comportando el circuito.

3.2 COMPONENTES DE LA CPU

El circuito a implementar consiste en un circuito compuesto por una CPU conectada a una memoria. Procederemos primero a explicar el funcionamiento de la CPU, para lo cual entendemos que resulta interesante el disponer de un esquema de conexionado de los diferentes elementos que componen la CPU.

La CPU que se va proceder a describir es la que aparece en el libro "VHDL Programming by Example" (Perry, 2003), que es posiblemente, la más básica que se puede describir y que a la vez resulte funcional. Básicamente su función es ejecutar un programa en lenguaje máquina (como funciona cualquier computador), en nuestro caso los modelos que vamos a ejecutar constituirán lo que hemos denominado "banco de pruebas", que describiremos en el capítulo siguiente. A continuación presentamos el esquema mostrado en la Figura 3.1.

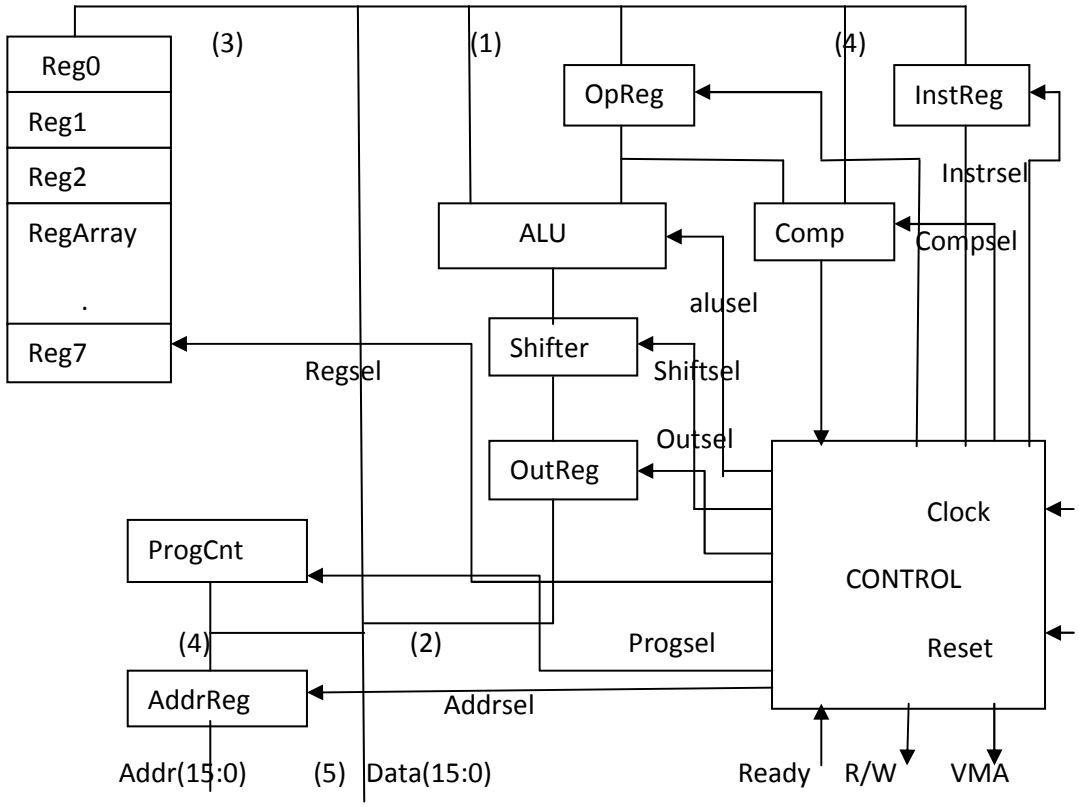


Figura 3.1: Esquema de conexión elementos de la CPU

Procederemos a continuación a describir lo distintos elementos que conforman el circuito de la CPU.

3.2.1 Control

El circuito que presentamos en la Figura 3.1, posee un elemento que constituye el alma del circuito y que es el elemento denominado como “Control”. La función que realiza este elemento, es la de realizar la lectura de las instrucciones que le entrega la memoria y realizar las operaciones/funciones implícitas en cada instrucción. Para ello, generará todas las señales de control necesarias, para que el resto de los elementos de la CPU ejecuten la función para la que han sido diseñados, en el momento adecuado. Lo que facilitara el funcionamiento del conjunto, ejecutando el trabajo para el que han sido requeridos.

Por tanto podemos decir que el elemento control es el cerebro del conjunto, como hemos comentado este elemento generara las señales adecuadas y las encaminara al elemento correspondiente de manera secuencial.

Para su funcionamiento, el elemento dispone de 5 puertos de entrada y de 11 puertos de salida. En cuanto a las entradas, comenzaremos comentando la entrada que en el esquema aparece como “Clock”, por esta entrada “Control” recibe señal de reloj (que no está en el esquemático), esta señal impondrá la velocidad final del sistema y marcará el comienzo de cada paso de las instrucciones. Es decir, los pasos se irán ejecutando guiados por la señal del reloj.

Otra entrada es la señal de “Reset”, cuya función es la de inicializar el elemento control y por tanto a todo el sistema.

Otra entrada es la señalada como “Ready”, que se utilizará para que la “Memoria” indique a “Control” que ha depositado un dato (en el bus), para que este actúe de manera adecuada según la instrucción a ejecutar.

Otra entrada es la que aparece nominada como “Compout”, cuya función es la de recibir la señal desde el comparador, cuando proceda la misma. Es decir, cuando realicemos una comparación de dos valores, a través de esta entrada, recibiremos el resultado de la comparación.

Otra entrada es la de “InstrugReg”, pues a través de esta entrada es donde recibirá “Control” las instrucciones a ejecutar, es decir es el puerto donde recibe dichas instrucciones.

En cuanto a las salidas, todas ellas tienen una función de control sobre los elementos que se conectan a las mismas. Es decir, estas salidas que conectan con elementos de la CPU, son utilizadas por “Control” para enviar señales, que el elemento que las recibe, interpreta para realizar una función para la que ha sido diseñado. De todas las señales de salida (Addrsel, Progsel, Regsel, Outsel, Shiftsel, Alusel, OpRegsel, R/W, VMA) solo R/W y VMA conectan con la “Memoria” a la cual debemos conectar para que el sistema pueda funcionar. De estas dos señales, la segunda se usa para habilitar a la “Memoria” y la primera se encarga de decirle que escriba en el bus un dato, o que lo lea del mismo.

3.2.2 Bus

Este elemento como tal, aparece en el esquemático como una simple línea marcada como (5) en la Figura 3.1, pero que, duda cabe que, constituye un elemento indispensable en el circuito. Pues por él, circularan tanto los datos, como las instrucciones (en este caso, no siempre es así respecto a las instrucciones), pues todos los elementos están conectados a él, bien de manera directa o de manera interpuesta a través de otro elemento.

Es un elemento indispensable por tanto, para el funcionamiento del circuito, pues es a través de él, como se intercambian datos e instrucciones. Anticipamos que este elemento como tal no existe en su implementación software (si en la Hardware) pues su función es asumida por diferentes instrucciones.

3.2.3 ALU

Este elemento, es el encargado de realizar todas las operaciones lógico/aritméticas, para ello dispone de dos entradas, una conectada directamente al “Bus” y otra que recibe datos desde el elemento “Opreg” (que es un registro de almacenamiento intermedio).

A través de estas dos entradas, recibirá los datos para realizar las operaciones, en el caso de operaciones binarias. Cuando las operaciones sean unarias, los datos los recibirá a través del puerto conectado directamente con el bus marcado como (1) en la Figura 3.1.

Disponemos de otra entrada de control (como el resto de elementos que configuran la CPU), denominada en este caso como “Alusel”, a través de esta

recibirá todas las señales de control, que le indicaran en todo momento que acción debe realizar y en que instante. El resultado obtenido de la operación, se entregará en la salida que conecta con el elemento “Shifter”.

3.2.4 Registros

Los registros básicamente son, elementos de almacenamiento temporal de datos. Su principal característica es la rapidez de acceso a los mismos, por lo tanto será utilizado como almacenamiento intermedio, en operaciones y acciones que realicen en la CPU. Siendo sus funciones muy variadas existirán por tanto varios tipos de registros según su función. Describimos a continuación los utilizados en el PFC.

OutReg: Se trata de un registro intermedio, cuya función es almacenar un dato cuando se le indique y entregarlo cuando se le solicite. Este tipo de registro tiene una entrada de datos en este caso conectada a “Shifter” y una salida de datos conectada al bus, marcado como (2) en la Figura 3.1. También dispone (como en todos los casos siguientes), de una entrada de control conectada al elemento Control y que en este caso se denomina “Outsel”, a través de la cual, recibirá la señal para guardar un dato o para entregarlo en su salida.

Regarray: Constituye un conjunto de ocho registros nominados desde Reg0 a Reg7. Constituye un elemento de memoria rápida (supuestamente mucho más rápida que la memoria principal, a la que conectaremos la CPU), cuyo principal uso es realizar un almacenamiento intermedio o temporal de los datos, mientras estamos trabajando con ellos. Con esta práctica, lo que se consigue, es reducir los

accesos a la memoria principal, que resultan más costoso en términos temporales y de cómputo.

Para realizar esta función, el elemento dispone de una única entrada/salida que está conectada al “Bus” marcado como (3) en la Figura 3.1, a través de la cual recibe datos y entrega los datos que se le solicitan, también disponemos de una entrada conectada al elemento “Control”, de donde recibirá datos de la operación a realizar (guardar, entregar) y la dirección de los mismos.

ProgCnt: Es el elemento contador de programa y se encarga de almacenar el número de instrucción que se está ejecutando. Dispone de una única entrada/salida donde entregará o recibirá el valor numérico de la instrucción que se está ejecutando, también tiene una entrada de control nominada en este caso “Progsel”.

AddrReg: Registro de direcciones de memoria, su función es almacenar las direcciones de memoria a las que va acceder, recibe entradas desde el “Bus” y desde “ProgCnt”, a través de una única entrada marcada como (4) en la Figura 3.1 y entrega el valor almacenado a la memoria en su salida. También dispone de una entrada de control nominada “Addrsel”, desde donde recibe información de la acción a realizar.

IntrugReg: El registro de instrucciones, es un registro de almacenamiento intermedio, que tal y como se ha explicado en casos anteriores, se dedica a almacenar un valor (una instrucción en este caso) para dejar libre al bus y lo entrega al elemento “Control”, cuando este se lo requiere, indicándoselo a través de la entrada de control nominada como “Instrsel”.

OpReg: Este elemento es también un registro de almacenamiento intermedio, que recibe datos del bus a través de su entrada y su función es almacenar el dato y entregarlo cuando se lo requiera el elemento "Control" a través de la entrada "OpRegsel". Este dato será recogido o bien por la "ALU" o por "Comp" según el tipo de instrucción que se esté ejecutando.

3.2.5 Comp

Su función es la de realizar comparaciones entre dos valores que recibe a través de dos entradas, una conectada directa al "Bus "marcado como (4) en la Figura 3.1 y otra conectada al "OpReg" (igual que la ALU). También dispone de una entrada de control nominada "Compsel, desde donde recibe el tipo de comparación a realizar, el resultado de la comparación se entrega a su salida, que está directamente conectada con el elemento "Control".

3.3 CONEXIÓN ENTRE CPU Y LA MEMORIA

El otro elemento de importancia crucial, es la "Memoria", a la cual va conectada la "CPU". Es el elemento que va a soportar físicamente el programa así como los datos del mismo. Para que nos hagamos una idea de la conexión y su relación con la CPU, adjuntamos un esquema en la Figura 3.2, de la manera en que están interconectadas, para pasar a explicar su función.

La memoria es el elemento encargado de almacenar datos de manera permanente. Entre esos datos que almacena de manera permanente, se incluye el

programa (banco de pruebas en nuestro caso), que también está almacenado en este elemento.

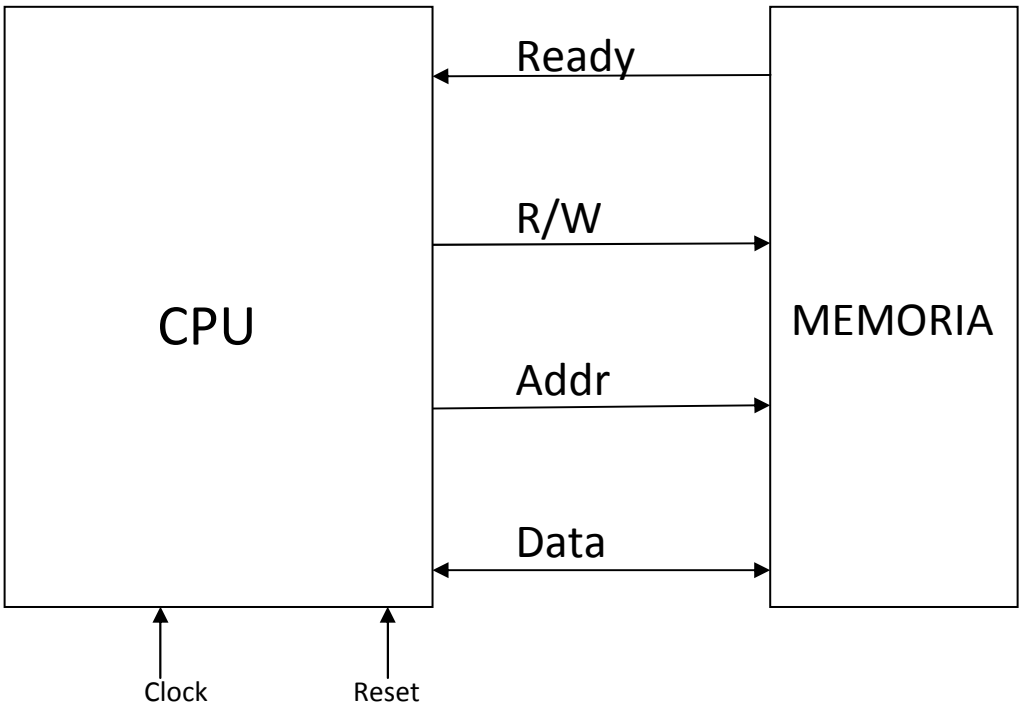


Figura 3.2: Esquema de conexión de la CPU a la memoria

Hay que recordar que en la "CPU", también existe una memoria (Regarray) pero se diferencia de ésta, en que ésta es volátil (desaparecen los datos con cada ejecución del programa), mientras que los datos que tengamos almacenados en la "Memoria" permanecen constantes en diferentes ejecuciones del programa.

La "Memoria" dispone de dos entradas, una salida y una conexión al bus denominada como "Data". En este caso, la conexión al bus es la vía de entrada y salida de datos, así como salida de instrucciones.

Claro está, que estos datos no entran y salen de manera indiscriminada, para poder hacer algo útil con ellos, necesitamos señales de control, que nos indiquen qué debemos hacer con estos datos. De esto se encargan las entradas señaladas

como “R/W”, que se encargan de seleccionar la operación de lectura o escritura. Es decir, si lo que desea es que los datos los entregue la “Memoria” al “Bus” o que los datos presentes en el “Bus” se almacenen en la “Memoria”.

La otra entrada nominada como “Addrs”, indica en qué dirección de la memoria se debe leer o escribir los datos. En una operación de lectura, en la que la “Memoria” escribe unos datos en el “Bus”, de alguna manera ésta deberá indicarle a el elemento “Control” que los datos están disponible en el “Bus”, pues esto se realiza con la salida nombrada como “Ready” cuya misión es simplemente indicar a “Control“ que tiene un dato (o instrucción) disponible en el “Bus” para que actúe en consecuencia.

3.4 DESCRIPCIÓN GENERAL DEL FUNCIONAMIENTO

Entendemos que una vez hecha una descripción general de los componentes, pasaremos a describir su funcionamiento de una manera general. Una vez reseteada la “CPU”, ésta se encargara de iniciar el contador de programa. Ordenará a “AddrReg” que entregue el valor del “ProgCnt” (lo habrá recibido desde el bus) a la “Memoria”. A su vez “Control” mediante señales en las salidas “R/W” y “VMA” ordenara a memoria que lea el dato existente en la posición de memoria que coincida con el valor del contador de programa y lo ponga en el “Bus”. Después la “Memoria” le indicará a “Control” que el dato está disponible en el “Bus”, mediante una señal en la salida “Ready”.

Una vez sabe “Control”, que el dato de la instrucción está en el bus, ordenará a “InstrReg”, mediante una señal en la salida “Instrsel”, que cargue el dato y se lo entregue. Con lo que “Control” ya tendrá a su disposición la instrucción a ejecutar, y comenzará a ejecutar los pasos que tenga programada esa instrucción de manera secuencial.

Una vez ejecutada la instrucción completamente, volverá a actualizar el contador de programa, sumándole una unidad al valor que tenga el contador de programa, volviendo a comenzar el ciclo otra vez, hasta la finalización de todas las instrucciones. Esta explicación será detallada más adelante, explicando la ruta de ejecución de cada instrucción en el momento en que expliquemos el código VHDL, usando éste para entender las instrucciones y la secuencia de las señales que se van produciendo.

3.5 INSTRUCCIONES

En esta explicación, hemos hablado repetidamente de las instrucciones que ejecuta el circuito, esto hace que la máquina que estamos simulando sea una máquina programable y esto ofrece la posibilidad de que el circuito realice funciones diferentes con solo cambiar el programa, o sea las instrucciones del mismo (lógica programada). Por tanto debemos explicar cómo son estas instrucciones y qué función y formato tienen.

Las instrucciones son por tanto, lo que realmente hace que el circuito tenga una utilidad y constituirá la base del “banco de pruebas”, que describiremos en el

siguiente capítulo, pues pensamos que la manera más fácil de probar el diseño es ponerla a funcionar.

Existen gran cantidad de instrucciones, tanto de lenguajes de bajo nivel (ensamblador, etc.), así como de lenguajes de alto nivel (java, C++, Pascal…), de hecho existen tantas instrucciones como seamos capaces de definir, si existe un equipo que las pueda ejecutar. En nuestro caso las instrucciones que ejecutará nuestro circuito, así como su formato, nos viene impuesto por la descripción que se realiza de las mismas en el libro base sobre el que se desarrolla este documento "VHDL Programming by Example" (Douglas L. Perry) que en su página 291 en la Sección "Instruction", definen las instrucciones que deberá poder ejecutar nuestro circuito, así como su composición. Por tanto, poco margen nos queda, debiendo ceñirnos, a lo que el autor propone para el desarrollo de las mismas, que pasamos a explicar.

El autor comienza describiendo los tipos de instrucción, agrupadas según su función que serán las siguientes:

Load: Esta instrucción carga el valor existente en un registro que indica la propia instrucción a otro registro que se indica en la propia instrucción.

Store: Esta instrucción, se encarga de almacenar datos en la "Memoria".

Branch: Esta instrucción, se encarga de cambiar el rumbo de ejecución de las instrucciones.

ALU: Aquí se agrupan todas aquellas instrucciones, que realizan alguna interacción con el elemento "ALU", es decir realizan operaciones con ésta.

Shift: Estas instrucciones se encargan de actuar sobre el “Shifter”, es decir se utilizan para operaciones de desplazamiento de bits.

En el siguiente Apartado, “Sample Instruction Representation”, el autor define mediante un ejemplo, el formato de las instrucciones.

Tal y como se explica en el libro base, las instrucciones pueden ser de simple palabra o de doble palabra; cada palabra tendrá una longitud de 16 bits (0 a 15), cada bit según su posición, ejercerá una función por tanto la posición del mismo es crucial para el bit (que solo podrá tener dos valores “0” y “1”) tenga un significado u otro. En el texto base, se nos muestra un ejemplo de formato de la instrucción LoadI # en hexadecimal. No se nos presentan ejemplos de otras instrucciones, con lo cual la interpretación del formato de las instrucciones que restan, ha sido obtenido o bien mirando el escaso código que aparece en el libro base, o bien criterio propio, reproducimos aquí la información extractada del formato de la simple y doble palabra.

En la Figura 3.3 se nos muestra la composición de la instrucciones de simple palaba y doble palabra, vemos como el código de operación (Opcode en la Figura 3.3), ocupa las primeras posiciones de la instrucción, mientras que la dirección del dato de entrada (SRC en la Figura 3.3) y la dirección de destino del dato de salida (DST en la Figura 3.3.) ocupan respectivamente las posiciones más bajas de la instrucción (marcadas de 5 a 3 para SRC y de 2 a 0 para DST).

En cuanto al formato de doble palabra, indicar que en la primera palabra aparece el código de operación, en la misma posición que en instrucciones de una palabra, diferenciándose de ésta, el hecho que solo aparece en la primera palabra

el destino de la operación que se realiza. En el caso de la Figura 3.3, es la instrucción LoadI 1#15 la cual cargará el valor “15” en el registro que se indica “1”, en este caso.

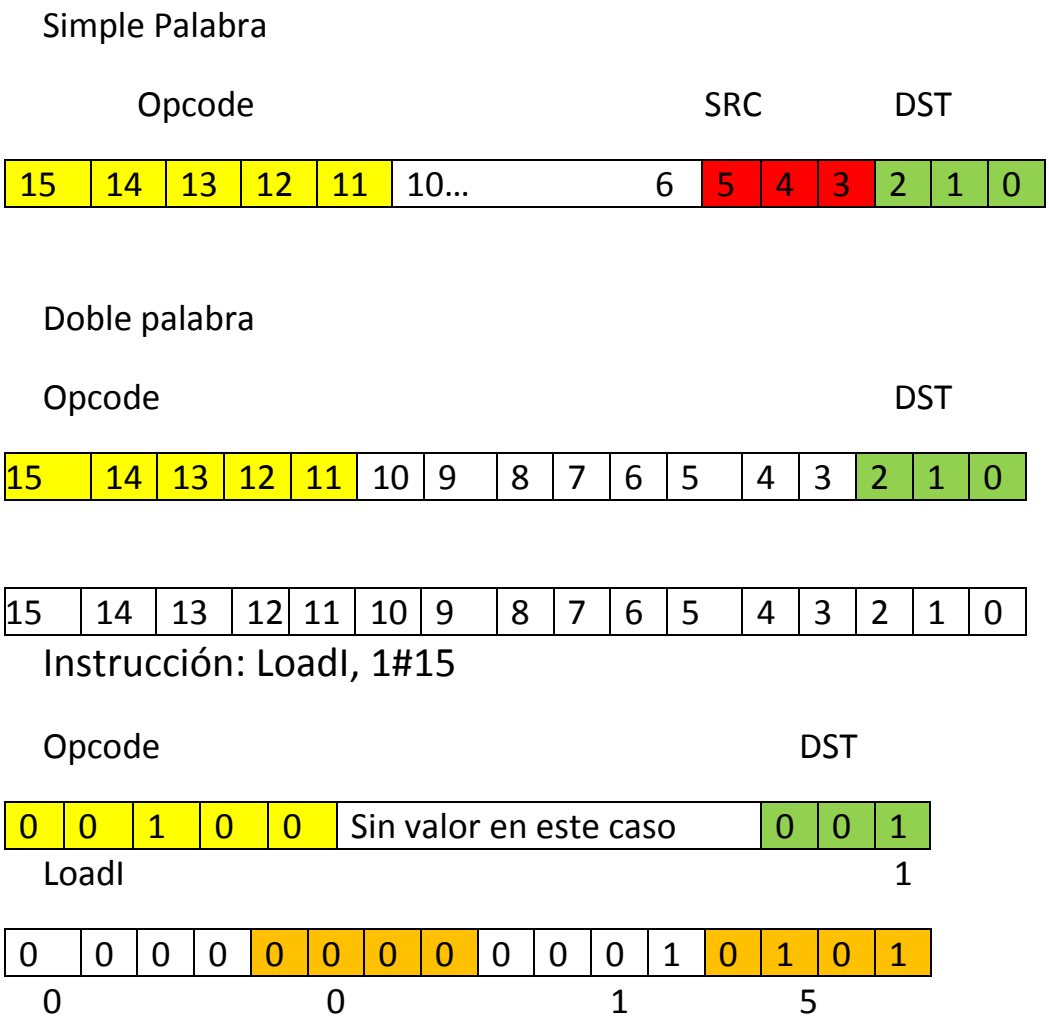


Figura 3.3: Formato de las instrucciones simples y dobles palabras

Todo el conjunto de instrucciones, que el autor define para el circuito del texto base, queda resumido en la Tabla 3.1, que adjuntamos y que es copia literal de la que aparece en el texto base.

Esta tabla constituye la base sobre la que definiremos el circuito, y constituye un buen resumen de las instrucciones que podremos implementar en el “banco de pruebas”, que tenemos que desarrollar.

OPCODE	INSTRUCTION	NOTE
00000	NOP	No operation
00001	LOAD	Load register
00010	STORE	Store register
00011	MOVE	Move value to register
00100	LOADI	Load register with immediate value
00101	BRANCHI	Branch to immediate address
00110	BRANCHGTI	Branch greater than to immediate address
00111	INC	Increment
01000	DEC	Decrement
01001	AND	And two registers
01010	OR	Or two registers
01011	XOR	Xor two registers
01100	NOT	Not a register value
01101	ADD	Add two registers
01110	SUB	Subtract two registers
01111	ZERO	Zero a register
10000	BRANCHLTI	Branch less than to immediate address
10001	BRANCHLT	Branch less than
10010	BRANCHNEQ	Branch not equal
10011	BRANCHNEQI	Branch not equal to immediate address
10100	BRANCHGT	Branch greater than
10101	BRANCH	Branch all the time
10110	BRANCHEQ	Branch if equal
10111	BRANCHEQI	Branch if equal to immediate address
11000	BRANCHLTEI	Branch if less or equal to immediate address
11001	BRANCHLTE	Branch if less or equal
11010	SHL	Shift left
11011	SHR	Shift right
11100	ROTR	Rotateright
11101	ROTL	Rotate left

Tabla 3.1: Tabla de instrucciones del libro VHDL Programming by Example

3.6 DISEÑO VHDL DE LOS REGISTROS

Comenzaremos la descripción del circuito mediante VHDL, explicando la implementación de los elementos más simples, que son los distintos tipos de registros. Esto nos servirá de base para ir entendiendo y familiarizándonos con el lenguaje, para entender con más facilidad los elementos más complejos.

Reg: La definición de este circuito como se aprecia en la Código 3.1 es bien simple, consiste en la declaración de la entidad donde se declara los puerto y se define su función, así como los valores que admiten (1). En este caso un puerto de entrada para las palabras que como hemos explicado es un tipo de dato declarado “bit16” y que consiste en un vector de 16 posiciones del tipo “std_logic”; un

puerto de salida del mismo tipo y otro puerto de entrada, desde donde recibirá las señales de control. En este caso el puerto se denomina “in” y es del tipo std_logic. El circuito se encargará de presentar por su puerto de salida la señal que recibe en la entrada con un retardo de un nanosegundo, si previamente ha recibido un bit de valor “1” (3), por el puerto de entrada clk. Resaltar que como veremos más adelante esta entidad se usará para la implementación de varios elementos.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.cpu_lib.all;
entity reg is port( a : in bit16;clk : in std_logic;q : out bit16);  --(1)
end reg;
architecture rtl of reg is
begin
regproc: process
begin
wait until clk'event and clk = '1';  --(3)
q <= a after 1 ns;  --(2)
end process;
end rtl;
```

Código 3.1: Código “reg.vhd”

Trireg:

```
library IEEE;
use IEEE.std_logic_1164.all; use work.cpu_lib.all;
entity trireg is
port( a : in bit16; en : in std_logic; clk : in std_logic; q : out bit16);  --(1)
end trireg;
architecture rtl of trireg is
signal val : bit16;
begin
triregdata: process
begin
wait until clk'event and clk = '1';
val <= a;  --(2)
end process;
trireg3st: process(en, val)
begin
if en = '1' then
q <= val after 1 ns;  --(3)
elsif en = '0' then  --(4)
q <= "ZZZZZZZZZZZZZZZZ" after 1 ns;
else
q <= "XXXXXXXXXXXXXXXX" after 1 ns;
end if; end process; end rtl;
```

Código 3.2: Código “trireg.vhd”

Este registro dispone de una entrada más, que el registro de tipo “Reg”, esta entrada nominada como “en” que es del mismo tipo que la entrada “clk” (std_logic) marcada como (1) en Código 3.2. Como su propio nombre indica tiene tres estados. Un primer estado, que consiste en almacenar el valor que tenga presente en su entrada “a” (2), si recibe un 1 en la entrada “clk”. Otro estado, que consiste en entregar en la salida “q”, el valor que previamente tenga almacenado el registro (variable “val”) (3), si recibe un 1 por la entrada “en”. El otro estado posible consiste en, una salida de alta impedancia si recibe un 0 por la entrada “en” (4). Por lo demás se comporta como el elemento “Reg”.

Shift: Este elemento dispone los mismos puertos de entrada y salida que un elemento “Reg”; una entrada, una salida y una entrada para control marcada como (1) en Código 3.3.

La diferencia principal respecto al elemento “Reg”, estriba en las funciones que realiza el elemento. Es este caso “Shift” tiene cinco posibilidades, estas están definidas en un tipo de dato declarado por nosotros llamado “t_shift”, que puede tomar los siguiente valores.

“Shftpass” (2), cuando este valor sea entregado en la entrada “in”, el registro entregara en su salida el mismo valor que tenga en su entrada.

“Shl” (3) este valor hará que el registro presente en su salida el valor de entrada, pero en el cual habrá hecho previamente un desplazamiento de bits a izquierdas (equivalente a multiplicar por dos).

"Shr" (4), este valor hará que el registro presente en su salida el valor de entrada, pero en el cual habrá hecho previamente un desplazamiento de bits a derechas (equivalente a la división entera por 2).

"Rotl" (5), este valor hará que el registro presente en su salida el valor de entrada, pero en el cual habrá hecho previamente un desplazamiento circular a izquierdas (sin pérdida de bits).

"rotr" (6), este valor hará que el registro presente en su salida el valor de entrada, pero en el cual habrá hecho previamente un desplazamiento circular a derechas (sin pérdida de bits).

```
library IEEE;use IEEE.std_logic_1164.all; use work.cpu_lib.all;
entity shift is
port ( a : in bit16; sel : in t_shift; y : out bit16);  --(1)
 end shift;
architecture rtl of shift is
begin
shftproc: process(a, sel)
begin
case sel is
when shftpass =>
y <= a after 1 ns;  --(2)
when shl =>
y <= a(14 downto 0) & '0' after 1 ns;  --(3)
when shr =>
y <= '0' & a(15 downto 1) after 1 ns;  --(4)
when rotl =>
y <= a(14 downto 0) & a(15) after 1 ns;  --(5)
when rotr =>
y <= a(0) & a(15 downto 1) after 1 ns;  --(6)
end case;
end process;
end rtl;
```

Código 3.3: Código "shift.vhd"

Regarray: Este elemento constituye una memoria rápida y volátil, está compuesta por un grupo de 8 registros, que se implementado mediante una matriz de vectores del tipo de dato "bit16", marcado como (1) en Código 3.4. Los cuales utilizaremos para almacenamiento, su utilidad es proveer de una memoria

rápida, para almacenamiento intermedio para uso de “Control”, así como de la “ALU”.

El elemento dispone de cuatro entradas y una salida, la primera de ellas “data”, constituye la entrada de datos que deseamos almacenar, las otras tres entradas se utilizan para el control del elemento. Una de ellas, “clk”, es la entrada de señal de reloj. Otra denominada, “en”, selecciona la operación bien de lectura o bien de escritura. Mientras que “sel”, que ha sido declarada como un tipo de datos “t_reg”, que es un vector de tres posiciones de tipo lógico, mediante el cual seleccionamos en cuál de las 8 posiciones deseamos leer o escribir. La salida de datos se produce a través de la salida “q”.

El proceso de guardar un dato en memoria, se produce en el instante en que se recibe una señal 1 a través de la entrada “clk”, siempre y cuando se halla introducido previamente un dato en la entrada “sel” (posición de la matriz donde guardar el dato) con lo que se guardara el dato que tengamos en la entrada (conectada al bus) (2).

El proceso de lectura es muy similar al de escritura, la diferencia estriba en que utiliza una entrada diferente para habilitar la lectura en este caso “en”. Cuando se recibe un 1 en dicha entrada, entregara en la salida el dato que previamente habremos seleccionado mediante la entrada "sel"(3).

```
library IEEE;
use IEEE.std_logic_1164.all; use IEEE.std_logic_unsigned.all; use work.cpu_lib.all;
entity regarray is
port( data : in bit16; sel : in t_reg; en : in std_logic; clk : in std_logic; q : out bit16);(1)
end regarray;
architecture rtl of regarray is
type t_ram is array (0 to 7) of bit16;
signal temp_data : bit16;
begin
process(clk,sel)
variable ramdata : t_ram;
begin
if clk'event and clk = '1' then
ramdata(conv_integer(sel)) := data;(2)
end if;
temp_data <= ramdata(conv_integer(sel)) after 1 ns;
end process;
process(en, temp_data)
begin
if en = '1' then
q <= temp_data after 1 ns;(3)
else
q <= "ZZZZZZZZZZZZZZZZ" after 1 ns;
end if;
end process;
end rtl;
```

Código 3.4: Código “regarray.vhd”

3.7 DISEÑO VHDL DEL COMPARADOR

Comp: Este elemento se dedica, a comparar dos valores presentes en sus entradas “a”, “b” y mostrar el resultado de la comparación en su salida “compout”. Para saber qué tipo de comparación debe establecer, tiene una entrada adicional conectada con “Control” llamada “sel” señalada como (1) en Código 3.5. Esta entrada es de un tipo de datos definido por nosotros como “t_comp” y que puede tomar los siguientes valores: “EQ”(2) realiza una comparación entre las entradas “a” y “b”, presentando en su salida “compout” un valor 1 sí a=b. “NEQ” (3) realiza una comparación entre las entradas “a” y “b”, presentando en su salida “compout” un valor 1 sí a≠b. “GT”(4) realiza una comparación entre las entradas “a” y “b”, presentando en su salida “compout” un valor 1 sí a>b. “GTE”(5) realiza una comparación entre las entradas “a” y “b”,

presentando en su salida “compout” un valor 1 sí a>=b. “LT”(6) realiza una comparación entre las entradas “a” y “b” presentando en su salida “compout” un valor 1 sí a<b. “LTE”(7) realiza una comparación entre las entradas “a” y “b”, presentando en su salida “compout” un valor 1 sí a<=b. Cabe destacar que en el resto de casos devolverá un cero en la salida “compout” (8).

```
library IEEE;
use IEEE.std_logic_1164.all; use IEEE.std_logic_arith.all;
use work.cpu_lib.all;
entity comp is
port( a, b : in bit16;
sel : in t_comp;  --(1)
compout : out std_logic);
end comp;
architecture rtl of comp is begin
compproc: process(a, b, sel) begin
case sel is
when eq =>
if a = b then  --(2)
 compout <= '1' after 1 ns;
else
 compout <= '0' after 1 ns;  --(8)
 end if;
when neq =>
if a /= b then  --(3)
 compout <= '1' after 1 ns;
else
   compout <= '0' after 1 ns;
 end if;
when gt =>
 if a > b then  --(4)
 compout <= '1' after 1 ns;
 else
 compout <= '0' after 1 ns;
end if;
when gte =>
if a >= b then  --(5)
 compout <= '1' after 1 ns;
else
   compout <= '0' after 1 ns;
end if;
when lt =>
if a < b then  --(6)
 compout <= '1' after 1 ns;
 else
 compout <= '0' after 1 ns;
 end if;
when lte =>
if a <= b then  --(7)
   compout <= '1' after 1 ns;
    else
   compout <= '0' after 1 ns;
 end if; end case;
 end process;
end rtl;
```

Código 3.5: Código “comp.vhd”

3.8 DISEÑO VHDL DE LA ALU

En este elemento, es donde se van a realizar todas las operaciones lógico/matemáticas de nuestros programas. Es por tanto una pieza fundamental en toda "CPU". Para cumplir con su cometido, este elemento dispone de dos entradas, "a" y "b", por donde recibirá los datos sobre los que tendrá que realizar las operaciones, una salida "c", por donde entregará el resultado de las mismas y una entrada de control llamada "sel", marcado como (1) en Código 3.6, a través de la cual, le indicaremos a la "ALU" que operación deseamos que nos realice.

Como en casos anteriores, esta entrada admite un tipo de datos que hemos declarado previamente, al que hemos llamado "t_alu", es un vector de 4 posiciones (3 a 0) de tipo "unsigned". Reproducimos en Tabla 3.2 los valores que admitirá o reconocerá la "ALU" así como la operación correspondiente a dicho valor.

Sel Input	Operacion	Nombre
0000	C = A	alupass
0001	C = A AND B	andOp
0010	C = A OR B	orOp
0011	C = NOT A	notOp
0100	C = A XOR B	xorOp
0101	C = A + B	plus
0110	C = A – B	alusub
0111	C = A + 1	inc
1000	C = A – 1	dec
1001	C = 0	zero

Tabla 3.2: Tabla de instrucciones para la "ALU"

Destacar, que como se aprecia en la tabla, realiza tanto operaciones unarias como binarias. La única diferencia entre ambas estribará en la forma en que cargará los datos.

En la tercera columna vemos que aparece un nombre, en este caso y tal como se apreciará en el código, se ha hecho una traslación del valor del vector

(dato tipo t_alu) a una constante, que es el nombre de operación y que es el que usaremos en el “case”, que discriminará qué operación debe realizar (2).

```
library IEEE;
use IEEE.std_logic_1164.all; use IEEE.std_logic_unsigned.all; use work.cpu_lib.all;
entity ALU is
port( a, b : in bit16; sel : in t_alu; c : out bit16);  --(1)
 end ALU;
architecture rtl of ALU is
begin
ALUproc: process(a, b, sel)
begin
case sel is
when alupass =>  --(2)
c <= a after 1 ns;
when andOp =>  --(2)
c <= a and b after 1 ns;
when orOp => (2)
c <= a or b after 1 ns;
when xorOp =>  --(2)
c <= a xor b   after 1 ns;
when notOp =>  --(2)
c <= not a after 1 ns;
when plus =>  --(2)
c <= a + b after 1 ns;
when alusub =>  --(2)
c <= b - a after 1 ns;
when inc =>  --(2)
c <= a + "0000000000000001" after 1 ns;
when dec =>  --(2)
c <= a - "0000000000000001" after 1 ns;
when zero =>  --(2)
c <= "0000000000000000" after 1 ns;
when others =>   c <= "0000000000000000" after 1 ns;
end case; end process;
end rtl;
```

Código 3.6: Código “alu.vhd”

3.9 DISEÑO VHDL DE CONTROL

Es el elemento, que se encarga de controlar todos los otros elementos, mediante señales que enviará a los elementos correspondientes. Esto conseguirá, que la acción conjunta de todos estos elementos, produzca unos resultados que son los que el programa requiere. Para efectuar el control del resto de elementos, el elemento dispone de 5 entradas y de 17 salidas tal y como podemos apreciar en

Código 3.7. La explicación pormenorizada de cada una, resultaría muy prolija, por tanto adjuntamos el fragmento de código donde se declaran las mismas.

Recordar que el lector, dispondrá del código completo que aquellos archivos que solo se muestran fragmentos (como por ejemplo control.vhd), en el Anexo A.

```
port( clock : in std_logic; reset : in std_logic; instrReg : in bit16; compout : in std_logic;
 ready : in std_logic;  --(entradas)

 progCntrWr : out std_logic;progCntrRd : out std_logic;
addrRegWr : out std_logic; addrRegRd : out std_logic; outRegWr : out std_logic;
 outRegRd : out std_logic; shiftSel : out t_shift; ALUSel : out t_ALU; compSel : out t_comp;
 opRegRd : out std_logic; opRegWr : out std_logic; instrWr : out std_logic; regSel : out t_reg;
 regRd : out std_logic; regWr : out std_logic; rw : out std_logic; vma : out std_logic);
--(salidas)
);
```

Código 3.7: Fragmento de código del archivo “control.vhd” donde se declaran las entradas y salidas

Para describir el elemento control, nos parece adecuado describir el funcionamiento de una instrucción de cada tipo que presentamos al inicio de este capítulo, cuando describimos las instrucciones agrupadas por tipos, según su funcionamiento.

Debido a que el funcionamiento de todas las instrucciones de un mismo tipo, es muy similar, con pequeñas diferencias, es lógico explicar solo la instrucción más significativa de cada tipo.

El código comienza inicializando las variables con sus valores iniciales, se nos muestra en Código 3.8 el fragmento donde se realiza esta operación.

```
begin
progCntrWr <= '0';
progCntrRd <= '0';
addrRegWr <= '0';
outRegWr <= '0';
outRegRd <= '0';
shiftSel <= shftpass;
alusel <= alupass;
compSel <= eq;
opRegRd <= '0';
opRegWr <= '0';
instrWr <= '0';
regSel <= "000";
regRd <= '0';
regWr <= '0';
rw <= '0';
vma <= '0';
```

Código 3.8: Fragmento de código de "control.vhd" donde se inicializan las variables

La primera acción que ejecuta el elemento control, es un reset de la CPU. Decir que todas las instrucciones del programa a ejecutar, como las internas (como "reset") se ejecutan de manera secuencial mediante sentencias "case". El código de la acción de reset es el que se muestra en Código 3.9.

```
case current_state is when reset1 => --(1)
aluSel <= zero after 1 ns;  --(11)
shiftSel <= shftpass;  --(12)
next_state <= reset2;  --(2)
when reset2 =>
aluSel <= zero;
shiftSel <= shftpass;
outRegWr <= '1';  --(21)
next_state <= reset3;  --(3)
when reset3 =>
outRegRd <= '1';
next_state <= reset4;
when reset4 =>  --(4)
outRegRd <= '1';
progCntrWr <= '1';  -(41)
addrRegWr <= '1';  --(42)
next_state <= reset5;
when reset5 =>  --(5)
vma <= '1';
rw <= '0';
next_state <= reset6;
when reset6 =>  --(6)
vma <= '1';
rw <= '0';
if ready = '1' then instrWr <= '1';  --(61)
next_state <= execute;
else
next_state <= reset6;  --(62)
end if;
```

Código 3.9: Fragmento de código de "control.vhd", donde se realiza el reset de la CPU

En "reset1"(1) se obtiene un 0 de la ALU(11), éste se pasa sin modificar por el registro de desplazamiento "Shifter"(12) entregándolo a "OutReg", se cambia el

valor de la variable del case a “reset2”(2): aquí se pasa el dato desde la salida de “Shifter” a “OutReg”(21) que la almacena, se cambia el valor de la variable del case a “reset3”(3): aquí el valor almacenado anteriormente en “OutReg” se vuelcan al bus(31), se cambia el valor de la variable a “reset4”(4): aquí se carga el valor que habíamos volcado al bus anteriormente en “ProgCnt”(41) y en “AddrReg” y éste lo entrega a la memoria(42), se cambia el valor de la variable del case a “reset5”(5): aquí se le pide a la memoria el dato (instrucción) que se encuentra en la dirección suministrada anteriormente por “AddrReg” y se cambia el valor de la variable a “reset6”(6): aquí, si la memoria contesta a control que tiene el dato disponible en el bus, control se cargara el dato tomándolo de “InstrReg”(61), cambiará el valor de la variable del case a “execute”; en caso contrario(62) entrará en un bucle, esperando a que conteste la memoria.

Una vez en el valor de “next_state” es “execute” se crea otro case dentro de este estado del case anterior, con la variable “instrReg” que tomará como valor las posiciones 15 a 11 de las instrucciones que cargue, Código 3.10.

```
when execute =>
case instrReg(15 downto 11) is
when "00000" =>       --- nop
next_state <= incPc;
```

Código 3.10: Fragmento de código de “control.vhd” alternativa case

Load: Vamos a explicar una instrucción del tipo load, concretamente la instrucción “load” cuyo “Opcode” es “00001”. Mostramos el código que ejecuta la instrucción en Código 3.11. Esta instrucción se encarga de cargar un dato en un registro, indicado en las posiciones 2 a 0 de la instrucción, de “Regarray”; que previamente habrá obtenido de la dirección de memoria, que se encuentra en un registro indicado en los bits de 5 a 3 de la instrucción.

```
when "00001" =>       --- load  (1)
regSel <= instrReg(5 downto 3);  --(11)
regRd <= '1';  --(12)
next_state <= load2;
end case;
when load2 =>  --(2)
regSel <= instrReg(5 downto 3);
regRd <= '1';
addrregWr <= '1';  --(21)
next_state <= load3;
when load3 =>  --(3)
vma <= '1';  --(31)
rw <= '0';  --(32)
next_state <= load4;
when load4 =>  --(4)
vma <= '1';
rw <= '0';
regSel <= instrReg(2 downto 0);  --(41)
regWr <= '1';  --(42)
next_state <= incPc;
```

Código 3.11: Fragmento de código “control.vhd”, instrucción “load

Cuando “Control” recibe una instrucción, comprueba los valores de los bits 15 a 11 de la instrucción y los asigna a la variable “instrugReg”. En nuestro caso la instrucción es “00001” con lo que comenzará la ejecución en el case de ese valor (1). Aquí envía a “Regarray” el valor del número de registro que obtiene de las posiciones 5 a 3 de la instrucción (11), le dice que lo lea y lo ponga en el bus (12). A continuación se cambia el valor de la variable del primer case (nex_state) y se pone al valor “load2”(2): aquí ordena a “AddrReg” que mande el dato (posición de la memoria) que ha leído del bus a la memoria(21), a continuación cambia el valor de la variable a “load3”: aquí control habilita la memoria (31) y le dice que lea la memoria en la posición que le ha entregado “AddrReg” y lo ponga en el bus(32). A continuación cambia el valor de la variable, a “load4” (4): aquí control manda la dirección donde deberá guardar el dato que hay en el bus, a “Regarray” (41) y a continuación le dice que lo guarde (42). A continuación cambia el valor de la variable nex_state a, “incPc”: aquí se incrementa el valor del contador del programa y se continúa con la ejecución de la siguiente instrucción.

Store: Vamos a explicar una instrucción de este tipo, concretamente la que tiene un código de operación “00010” llamada “store”. El código que se ejecuta en esta instrucción se muestra en Código 3.1. La función que realiza esta instrucción, es la de almacenar en una dirección de la memoria, que se encuentra especificada en un registro del “Regarray”, y que se especifica en la instrucción en los valores de los bits de las posiciones 2 a 0, el dato que se encuentra en otro registro del “Regarray” y que en este caso se especifica el los bits de las posiciones 5 a 3.

Básicamente lo que hace, es guardar un valor que se encuentra en los registros de almacenamiento, en la memoria principal. Se puede decir que es la instrucción inversa a la de “Load” que realizaba el trabajo en sentido inverso.

Cuando se carga la instrucción en el elemento “Control”, en los valores de los bits de las posiciones 15 a 11, se discrimina que instrucción se va a ejecutar (1), por tanto comienza la ejecución de la instrucción cuya primera acción consiste en dar a “Regarray” el número del registro donde se encuentra almacenado el dato de la dirección de memoria (11). Acto seguido le pide que lo lea y lo deposite en el bus (12), cambia el estado de la variable (next_state) a “store2”.

```
when "00010" =>       --- store (1)
regSel <= instrReg(2 downto 0); (11)
regRd <= '1'; (12)
next_state <= store2;
when store2 =>
regSel <= instrReg(2 downto 0);
regRd <= '1';
addrregWr <= '1'; (2)
next_state <= store3;
when store3 =>
regSel <= instrReg(5 downto 3);
regRd <= '1';
next_state <= store4;
when store4 =>
regSel <= instrReg(5 downto 3); (3)
regRd <= '1';
vma <= '1'; (4)
rw <= '1'; (41)
```

Código 3.12: Fragmento de código de “control.vhd”, instrucción “store”

En “store2” se limita a decirle al registro de direcciones “AddrReg” que tome el dato del bus y lo se lo entregue a la memoria (2), cambiando a continuación el valor de la variable, a “store3”. Aquí “Control” le indica a “Regarray” el nº de registro donde se encuentra el dato (3), indicado en los valores de los bits de las posiciones 5 a 3 de la instrucción. En este caso, será el dato que queremos guardar en la memoria. Acto seguido le indica que lo lea y por tanto lo deposite en el bus (31). Cambia el valor de la variable, a “store4”: aquí se habilita a la memoria (4) y se le dice que lo guarde (41).

Branch: Este es quizás, el grupo más complejo, por su variedad de instrucciones. Vamos a describir, el funcionamiento de la más básica de todas, que es el salto incondicional a una dirección.

```
when "00101" =>       ---- BranchImm (1)
progcntrRd <= '1';  --(11)
alusel <= inc;  --(12)
shiftsel <= shftpass;  --(13)
next_state <= bral2;
when bral2 =>
progcntrRd <= '1';
alusel <= inc;
shiftsel <= shftpass;
outregWr <= '1';
next_state <= bral3;  --(2)
when bral3 =>
outregRd <= '1';   --(3)
next_state <= bral4;
when bral4 =>
outregRd <= '1';
progcntrWr <= '1';   --(4)
addrregWr <= '1';  --(41)
next_state <= bral5;
next_state <= bral5;
when bral5 =>
vma <= '1';  --(5)
rw <= '0';  --(51)
next_state <= bral6;
when bral6 =>
vma <= '1';
rw <= '0';
if ready = '1' then  --(6)
progcntrWr <= '1';  --(61)
next_state <= loadPc;  -- (62)
else
next_state <= bral6;  --(63)
end if;
```

Código 3.13: Fragmento de código de “control.vhd”, instrucción “branch”

La función de esta instrucción, es la de realizar un salto a una dirección del programa que viene indicada en la segunda palabra de la instrucción. Por tanto estamos hablando de una instrucción de doble palabra. Esta instrucción comienza, tal y como se puede apreciar en Código 3.13, con el reconocimiento de que instrucción se trata por parte de control, mediante los valores de los bits de las posiciones 15 a 11. En este caso esos valores son “00101” (1). Con esto control ya sabe de qué instrucción se trata y comienza por tanto su ejecución, cuyo primer paso consiste en leer el valor del contador de programa y volcarlo al bus (11). Con el dato en el bus, el siguiente paso que se realiza es indicar a la “ALU” que incremente el valor que tiene disponible en el bus en una unidad (12). Una vez hecho esto el valor está disponible para “Shift” al cual, se le indica que lo entregue al siguiente elemento “OutReg”, sin realizar ninguna operación de desplazamiento (13). A continuación se cambia el valor de la variable “nex_state”, a “braI2”: aquí simplemente se le indica a “OutReg” que guarde el dato que tiene disponible a su entrada (2), se cambia la variable, a, “braI3”. Aquí se le indica a “OutReg”, que el dato que tenía guardado lo entregue en su salida, volcándolo de esta manera en el bus (3). A continuación cambia el valor de la variable, a “braI4”. Aquí como el valor del contador de programa incrementado en una unidad ya está disponible en el bus, lo que hace “Control” es ordenar guardarlo de nuevo en el “ProgCnt”(4), y le dice a “AddrReg” que entregue a memoria esa dirección (41). A continuación se cambia el valor de la variable, a “braI5”: aquí lo que hace la instrucción es habilitar la memoria (5) y leer la segunda parte de la instrucción (51), que recordemos es donde se encuentra la dirección de salto, se cambia el valor de la variable, a “braI6”. Aquí lo que hacemos es un bucle, en el que “Control” espera a que la

memoria le indique que ha puesto el dato en el bus(6) en cuyo caso actualiza el valor del contador de programa con el dato disponible en el bus(61) y actualiza la variable nex_state al valor “loadPc” (62). Con lo que sigue la ejecución del programa. En el caso, que no reciba confirmación de la memoria de que el dato está disponible en el bus se actualiza la variable nex_state al valor que ya tenía, “brai6”(63) creando así un bucle de espera.

ALU y Shift: Presentamos estos dos tipos de instrucciones en un mismo grupo, pues sus rutas de ejecución son prácticamente la misma, siendo la diferencia principal, en qué elemento se realiza la operación. Unas instrucciones realizan la operación en la ALU, mientras que las de desplazamiento la operación se realiza en el registro de desplazamiento “Shift”.

Sin embargo la ruta de ejecución es la misma y solo difieren entre sí en unas pocas señales que indicaremos en su momento para distinguirlas. Mostramos el código que ejecuta la instrucción en Código 3.14.

Explicaremos por tanto, la operación de suma, que en el caso que nos ocupa tiene un código de operación “01101”, que consiste en la suma de dos datos, se trata pues de una operación binaria, pues implica el uso de dos datos.

Destacar que existen operaciones unarias, que sin pretenderlo hemos explicado en la instrucción anterior (branchI), cuando incrementábamos el valor del contador de programa.

El código de dicha operación es el que se nos muestra en Código 3.14.

```
when "01101" =>  ----sumar (1)
operacion := plus;  --(11)
 regSel <= instrReg(5 downto 3);  --(12)
regRd <= '1';  --(13)
next_state <= sum2;
when sum2 =>
 regSel <= instrReg(5 downto 3);
regRd <= '1';
opRegWr <= '1';  --(2)
next_state <= sum3;
when sum3 =>
opRegRd <= '1';
regSel <= instrReg(2 downto 0);  --(3)
regRd <= '1';  --(31)
alusel <= operacion;  --(32)
next_state <= sum4;
when sum4 =>
opRegRd <= '1';
regSel <= instrReg(2 downto 0);
regRd <= '1';
alusel <= operacion;
shiftsel <= shftpass;  --(4)
outregWr <= '1';  --(41)
next_state <= sum5;
when sum5 =>
outregRd <= '1';  --(5)
next_state <= sum6;
when sum6 =>
outregRd <= '1';
 regSel <= instrReg(5 downto 3);  --(6)
regWr <= '1';  --(61)
next_state <= incPc;
```

Código 3.14: Fragmento de código de “control.vhd”, instrucciones “ALU” o “Shift”

Como en todas las instrucciones, ésta comienza también detectando que código de operación tiene, para saber de qué operación se trata (1). Una vez discriminada la operación, el siguiente paso consiste en guardar la operación en la variable “operación” (11). A continuación le indicamos al “Regarray” dónde está el dato del primer operando, mediante los bits de las posiciones 5 a 3 (12) y a continuación le pedimos que lo lea y lo vuelque en el bus (13). Después, cambia la variable next_state, al valor “sum2”. Aquí lo que hacemos es cargar el valor del primer operando en el registro “OpReg” (2) dejando así libre el bus para el siguiente operando, cambia el valor de la variable a “sum3”. Aquí indicamos a “Regarray” donde se encuentra el siguiente operando (3) mediante los bits de 2 a 0, a continuación le pedimos a “Regarray” que lo lea y lo vuelque en el bus (31), le indicamos a la “ALU” que operación tiene que realizar, la realiza y la presenta en la

salida (32). Después cambia el valor de la variable a “sum4”. Aquí es donde difieren las instrucciones de “ALU” respecto de las de “Shift”.

En las operaciones de “ALU”, el dato de salida pasa simplemente por el registro de desplazamiento sin realizar ningún cambio en el dato (4). Mientras que en las de desplazamiento, es en la “ALU” donde pasa sin cambiar el dato, y en cambio es en el registro de desplazamiento donde realiza una operación de desplazamiento.

Seguidamente se indica al “OutReg” que guarde el dato que tiene en la entrada (salida de “Shift”)(41), y se cambia a continuación el valor de la variable, a “sum5”.Aquí le indicamos a “OutReg” que entregue en su salida el dato que había almacenado anteriormente y por tanto lo entregue en el bus (5). A continuación se cambia la variable al valor “sum6”. Aquí se le indica a “Regarray” el número del registro donde debe guardar el dato que vendrá especificado en la instrucción mediante los valores de los bits de las posiciones de 2 a 0 (6). A continuación se le pide que lo guarde en dicho registro. Continúa así la ejecución del programa incrementando el contador de programa y siguiendo con la ejecución de la siguiente instrucción.

3.10 DISEÑO VHDL DE LA MEMORIA

Memoria: Este elemento, es el que actúa como memoria permanente y no volátil del circuito, no forma parte de la “CPU” y por tanto se describe aparte, pero está íntimamente relacionada con la misma, pues son partes necesarias para que

el circuito tenga una utilidad real. Mostramos el código que ejecuta este elemento en Código 3.15.

Este elemento, se explicara con mayor detalle en el banco de pruebas, pues es el elemento principal del mismo. Vamos a dar una explicación, un tanto somera del elemento, lo que nos ayudara a tener una visión global y de conjunto del circuito a implementar. Pues no constituye en sí, una parte aislada del mismo, sino que ambas "CPU" y "Memoria" están íntimamente relacionadas. Describiremos primero la estructura del elemento.

```
port (addr : in bit16;
sel, rw : in std_logic; ready : out std_logic; data : inout bit16);  --(1)
(.....)
 memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;  --(2)
```

Código 3.15: Fragmento de código de "memory.vhd", estructura y definición

Podemos apreciar en Código 3.15 que el elemento dispone de dos entradas y una salida de control así como una entrada y salida de datos hacia/desde la memoria (1). La cuestión es cómo se almacenan los datos en la memoria. En este caso tenemos un array de 64 vectores de 16 posiciones (longitud de palabra previamente establecida) (2). En la cual, guardaremos tanto el programa como los datos que deseemos almacenar en la memoria principal. Destacar que en este caso, solo serán permanentes entre las simulaciones, los datos del programa que se guardan mediante código. No aquellos que guardamos durante la ejecución de ese código.

Pasamos a explicar, cómo se introducen o se leen datos de la memoria. El código que realiza las operaciones de escritura/lectura se muestra en Código 3.16, que pasamos a explicar a continuación.

```
begin
data <= "ZZZZZZZZZZZZZZZZ";
ready <= '0';
if sel = '1' then  --(1)
if rw = '0' then  --(2)
data <= mem_data(CONV_INTEGER(addr(15 downto 0))) after 1 ns;  -- (3), (4)
ready <= '1';  --(5)
elsif rw = '1' then  --(2)
mem_data(CONV_INTEGER(addr(15 downto 0))):= data;  --(3), (1)
end if;
else
data <= "ZZZZZZZZZZZZZZZZ" after 1 ns;
end if;
end process;
```

Código 3.16: Fragmento de código “memory.vhd”, lectura/escritura de datos

Para leer un dato de la memoria, deberemos realizar desde la “CPU” las siguientes operaciones. Primero habilitar la memoria (1), a continuación indicar qué operación deseamos realizar en este caso lectura (2). Previamente habremos ingresado la dirección de la memoria a leer (3). Tras lo cual, la memoria entregará el dato requerido en su salida al bus (4), e indicará a “Control” que el dato está disponible en el bus (5).

Para escribir un dato el proceso es muy similar la diferencia estriba en que el dato tiene que estar disponible al principio (1) y no al final como sucedía en la operación de lectura. Por tanto una vez disponible el dato en el bus indicamos que operación vamos a realizar (2). Procedemos a guardar en la matriz el dato en la posición indicada en la entrada de direcciones (3).

Bien, hasta aquí hemos definido los tipos o modelos de fichas, que componen nuestro puzle (circuito). Pero aún no hemos definido ni la cantidad de

fichas, ni cómo van a conectarse entre ellas; esta labor es la que vamos a realizar a continuación con la descripción de los siguientes archivos.

3.11 DECLARACIÓN DE LOS TIPOS DE DATOS

cpu_lib: En este archivo se realiza la descripción de los tipos de datos que deseamos declarar (1), tal y como se muestra en Código 3.17, así como de sus valores o tipos, y se especifican el valor que pueden tomar ciertas variables.

Podemos ver por ejemplo, cómo hemos declarado los valores que puede tomar la variable "next_state", declarando un tipo de datos de tipo "state" (2), que puede tomar los valores que definimos nosotros con anterioridad. Estos valores son los que nos ayudaran a recorrer los "case" del elemento control. Pues aquí se ha definido los valores que pueden tomar.

En este archivo, en definitiva, se especifican los tipos de señal que vamos a utilizar para comunicar los distintos elementos del circuito, entre ellos.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
package cpu_lib is  --(1)
type t_shift is (shftpass, shl, shr, rotl, rotr);
subtype t_ALU is unsigned(3 downto 0);
constant ALUpass : unsigned(3 downto 0) := "0000";
constant andOp : unsigned(3 downto 0) := "0001";
constant orOp : unsigned(3 downto 0) := "0010";
constant notOp : unsigned(3 downto 0) := "0011";
constant xorOp : unsigned(3 downto 0) := "0100";
constant plus : unsigned(3 downto 0) := "0101";
constant ALUsub : unsigned(3 downto 0) := "0110";
constant inc : unsigned(3 downto 0) := "0111";
constant dec : unsigned(3 downto 0) := "1000";
constant zero : unsigned(3 downto 0) := "1001";
type t_comp is (eq, neq, gt, gte, lt, lte);
subtype t_reg is std_logic_vector(2 downto 0);  --(1)
type state is (reset1, reset2, reset3, reset4, reset5, reset6, execute, nop, load, store,  --(2)
 move,load2, load3, load4, store2, store3, store4, move2, move3, move4,incPc, incPc2, incPc3, incPc4,
incPc5, incPc6, loadPc, loadPc2,loadPc3, loadPc4, bgtI2, bgtI3, bgtI4, bgtI5, bgtI6, bgtI7,bgtI8,
bgtI9, bgtI10, braI2, braI3, braI4, braI5, braI6, loadI2,loadI3, loadI4, loadI5, loadI6, inc2, inc3, inc4,
sum2, sum3, sum4, sum5, sum6, shl2, shl3, shl4, shr2, shr3, shr4, parar);  --(2)
subtype bit16 is std_logic_vector(15 downto 0);
end cpu_lib;
```

Código 3.17: código de "cpu_lib.vhd"

3.12 DISEÑO VHDL DE LA ESTRUCTURA DEL CIRCUITO

El siguiente archivo que vamos a describir es el que se encarga de describir la conexión de los dos subcircuitos implicados en el circuito general, por un aparte la "CPU" (1) y por otra parte la "Memoria" (2), el código se describe en Código 3.18.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.cpu_lib.all;
entity top is end top;
architecture behave of top is
component mem1 port (addr : in bit16;
sel, rw : in std_logic;
ready : out std_logic;
data : inout bit16);
end component;
component cpu
port(clock, reset, ready : in std_logic;
addr : out bit16;
rw, vma : out std_logic;
data : inout bit16);
end component;
signal addr, data : bit16;
signal vma, rw, ready : std_logic;
signal clock, reset : std_logic := '0';
begin
clock <= not clock after 50 ns;
reset <= '1', '0' after 100 ns;
m1 : entity  work.mem (whileyif) port map (addr, vma, rw, ready, data);  --(2)
u1 : cpu port map(clock, reset, ready, addr, rw, vma, data);  --(1)
end behave;
```

Código 3.18: Código de "top.vhd"

CPU: La descripción de cuántos componentes tiene la "CPU", así como de sus tipos y cómo están conectados estos entre sí; corresponde al archivo "cpu.vhd". En Código 3.19, se nos muestra un fragmento de dicho archivo. Comenzaremos por explicar, cómo se define que componente pertenece a qué tipo de los que hemos descrito con anterioridad (1). En esta parte, también se declara explícitamente el nombre de los puertos (2) y por tanto un mismo nombre de puerto en dos componentes distintos indicará que están unidos eléctricamente.

Este archivo por tanto, describe los tipos de elementos que hemos definido (Reg, ALU, etc.), define los componentes de la “CPU” que pertenecerán a uno de los tipos que hemos definido y con posterioridad establece la conexión eléctrica ente elementos, tal y como hemos descrito en el párrafo anterior.

```
ra1 : regarray port map(data, regsel, regRd, regWr, data); opreg: trireg port map (data, opregRd, opregWr, opdata);
ALU1: ALU port map (data, opdata, ALUsel, ALUout); shift1: shift port map (ALUout, shiftsel, shiftout); outreg: trireg
port map (shiftout, outregRd, outregWr,
data);
addrreg: reg port map (data,addrregWr, addr);  --(1), data(2),
progcntr: trireg port map (data,progcntrRd, progcntrWr,data); -- data(2),
comp1: comp port map (opdata, compsel, compout); instr1: reg port map  (data, instrregWr, instrOutReg);
con1: control port map (clock, reset, instrOutReg,
compout, ready, progcntrWr, progcntrRd, addrregWr, outregWr, outregRd, shiftsel, ALUsel, compsel, opregRd,
opregWr, instrregWr, regsel, regRd, regWr, rw, vma);
end rtl
```

Código 3.19: Fragmento de código “cpu.vhd”

mem: Es el elemento que se conecta a la CPU, la conexión del mismo queda especificada en Código 3.18 (2). El comportamiento del elemento “mem” queda definido en la Sección 3.9, por tanto no procede el volver a explicar este elemento y remitimos a dicha sección para aclarar las dudas que se presenten.

3.13 CONCLUSIONES

Debemos señalar, que para aquellos archivos que se han mostrado solo fragmentos de su código, podrán visualizarse con el código completo en el Anexo "A", donde podrá ser consultado en toda su extensión. Entendemos, que para realizar explicaciones sobre puntos concretos de archivos muy extensos, es mejor presentar solo el fragmento concreto, sobre el que nos referimos, que presentar en varias páginas todo el código de dicho archivo. Pues entendemos que esto no aportara claridad en esos casos.

4

PROGRAMACIÓN VHDL DEL BANCO DE PRUEBAS

4.1 INTRODUCCIÓN

El diseño del circuito expresado en el capítulo anterior, qué duda cabe, requiere de un "banco de pruebas", que demuestre la corrección del mismo. Es decir, demuestre su validez así como su correcto funcionamiento. Por tanto explicamos aquí un breve esbozo de cómo se realiza la "V y V" validación y verificación de los diseños hardware.

El primer proceso de prueba, que se realiza en el desarrollo del producto software, es sin duda la verificación, que de acuerdo con las bases sentadas con anterioridad consiste en comprobar que el software que se va produciendo es correcto. En la presente memoria la documentación existente de este proceso es casi nula, no significando esto, que este proceso de verificación no se ha producido, pues es un proceso que de manera consciente o inconsciente, está presente en el desarrollo de todos los productos software. Esto hace más necesario si cabe, introducir una pequeña explicación del procedimiento seguido. Entendemos pues, que en un proyecto tan pequeño no quedaría justificado el esfuerzo de desarrollar una documentación de dicho proceso.

El proceso de verificación, en este proyecto ha consistido básicamente en unas pruebas de unidad y unas pruebas de integración que pasamos a desarrollar brevemente.

Pruebas de unidad: Consistentes en la prueba funcional de las unidades más básicas construidas; en nuestro caso la unidad queda establecida a nivel de clase, que representa la implementación software de un elemento o unidad hardware básica. En este caso se puede identificar funcionalmente, como un elemento electrónico concreto.

Pruebas de Integración: El siguiente paso lógico a las pruebas de unidad, son las pruebas de integración. Al contrario que las anteriores, éstas admiten varias estrategias de integración y pruebas, siendo el criterio del programador (o jefe del proyecto) el elegir la estrategia más adecuada de entre las más usuales.

En el caso que nos ocupa, nuestra elección ha consistido en una estrategia de integración incremental. Ésta, como su propio nombre indica, consiste en líneas generales, en ir integrando pequeñas unidades progresivamente, en función de cómo avanza el desarrollo. Evitando de esta manera, los inconvenientes de usar una estrategia “Big Bang”, beneficiándonos así de la ventaja, de la detección temprana de errores graves, que pudieran afectar a la viabilidad del proyecto.

Dentro del enfoque incremental existen varias estrategias de integración que básicamente son: integración ascendente e integración descendente. En nuestro caso se ha optado por una estrategia que incorpora características de ambos procedimientos, conocida como integración en sándwich. Ésta, consiste

básicamente en usar integración ascendente en los niveles bajos, e integración descendente en los niveles más altos de la arquitectura.

En nuestro caso, los niveles de arquitectura no son significativos pues la dependencia entre módulos es pequeña, pero sí se establece una clara dependencia de uso, para la realización de pruebas. Es decir, para cualquier prueba de un nivel superior al de la unidad, prácticamente todas las pruebas dependen del elemento “control”. Podríamos situar a este elemento, en la cúspide de la jerarquía de uso, pues prácticamente todas las operaciones son controladas por dicho elemento, que es el corazón del funcionamiento del proyecto, junto con el elemento “Memoria”.

En cuanto al proceso de validación, que recordaremos hemos establecido anteriormente, consiste en comprobar si el producto se ajusta a los requisitos del cliente. Diversos autores coinciden en establecer la frontera de los requisitos del cliente, en el documento “especificación de requisitos del software”. En nuestro caso, al tratarse de un proyecto académico, este documento no se presenta con tal nombre. Pero si podríamos decir que sus funciones y utilidad, son las que realiza el documento del anteproyecto. En dicho documento, en su apartado de “objetivos”, establece que requisitos finales, debe cumplir el producto software a desarrollar, por tanto ese va a ser el documento base sobre el que se debe desarrollar todas las pruebas de validación.

En este proceso, cabe destacar que si se ha aportado documentación de las pruebas realizadas, que básicamente han ido encaminado a demostrar que los requisitos del anteproyecto, se cumplen en el software desarrollado. Las pruebas

típicas (aunque no únicas) de la validación, son conocidas como pruebas alfa y pruebas beta.

Las pruebas alfa, consisten en pruebas que realiza el usuario en el lugar o ámbito donde se ha producido el software. Estas pruebas, obviamente en nuestro caso, han sido completadas por los desarrolladores del proyecto software. Siendo la documentación que se aporta en esta memoria la demostración del cumplimiento de estos requisitos.

En cuanto a las pruebas beta, éstas les corresponden al tribunal el realizarlas para producir como resultado de este proceso de prueba, una nota final que no olvidemos es el objetivo finalista del proyecto.

El motivo académico del proyecto, es sin duda, el que ha de guiar todo el proceso. Por tanto se impone, que debamos realizar una explicación sobre la motivación y objetivos perseguidos al desarrollar la batería de pruebas que se ha implementado.

No está de más por tanto, el recordar los objetivos que persiguen el presente PFC. En su primer punto habla de “Comprender el funcionamiento del circuito digital consistente en una CPU conectada a una memoria…”, nos parece que la demostración del cumplimiento de este punto estará realizado con la demostración de los demás objetivos, pues este no es mas, que un requisito previo para la realización del software.

En su segundo punto dice, “Proponer un banco de pruebas para dicho circuito”, en este punto el banco de pruebas propuesto, que se desarrolla con más

extensión a lo largo de este capítulo y que entendemos es la base de la validación del proyecto. Consiste en cuatro pequeños subprogramas, que tratan de abarcar todos los tipos de posibilidades, que se presentan en un circuito tan simple, como el propuesto para este PFC.

Por tanto nuestro primer subprograma consiste en un bucle while seguido de una alternativa if/else. Además se realiza, una operación matemática resta y la comparación necesaria para los bucles.

El segundo subprograma de pruebas que se ha propuesto, consiste en un bucle while que tiene anidado en su interior un bucle for. A su vez se realiza una operación matemática, consistente en la suma, además de las comparaciones propias de los bucles.

El tercer subprograma de prueba, consiste en realizar una copia de una parte de la memoria a otra parte de la memoria. Cabe destacar que esta prueba está implementada en el texto base y nos ha parecido obligado a la vez que interesante el incluirla en la batería de pruebas.

El cuarto subprograma, consiste en realizar todas las operaciones matemáticas que se han implementado en la ALU, con lo cual queda demostrado su funcionamiento correcto de esta parte importante de la CPU. Cabe destacar que otra parte importante de la CPU es el comparador, éste se ha utilizado en los bucles de forma que se han implementado las diferentes comparaciones que se pueden establecer (<, >, =, ¬).

Como se puede comprobar, no se ha buscado una batería de pruebas extensa, sino mas bien, una batería que pruebe el máximo de operaciones y sobre todo las más significativas. Con el fin de que sirva como demostración, que la CPU desarrollada realiza las mismas funciones que una CPU de similares características, implementada con un método diferente.

El banco de pruebas desarrollado, está compuesto por cuatro subprogramas, cuya misión principal es demostrar la validez del desarrollo implementado, vamos por tanto a realizar una explicación de los subprogramas, que hemos desarrollado con dicho fin. Como se ha comentado con anterioridad estos no son programas complejos, sino mas bien sencillos, tratando de demostrar, que si funciona el circuito implementado en sus estructuras más básicas, también lo hará cuando el programa a probar sea un agrupación múltiple de dichas estructuras.

Los bancos de pruebas implementados se encuentran como no podía ser de otra manera, dentro del archivo “mem.vhd”. Destacar que cada implementación o unidad del banco de pruebas, se presenta o describe como una arquitectura diferente del elemento “mem”. En este elemento y concretamente en su matriz de vectores es donde se encuentra codificado los diversos subprogramas del “banco de pruebas”.

4.2 BANCO DE PRUEBAS “BUCLE WHILE Y ALTERNATIVA IF/ELSE”

Este subprograma realiza la ejecución de un bucle while y de una alternativa if/else, junto con la operación matemática suma.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	descripción
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	LoadI en reg 0--a=2;
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	--2
2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	LoadI en reg 1--b=15;
3	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	--15
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	loadI en reg 2--c=7;
5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	--7
6	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	LoadI en reg 3--d=10;
7	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	--10
8	0	0	1	1	0	0	0	0	0	0	0	1	1	0	1	0	BranchI reg2 reg3;
9	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	14
10	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	Sub reg 1 reg 0 b=b-a
11	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	inc reg 3—C++;
12	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	BranchI ;
13	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	--8
14	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	Branch not equal
15	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	Dirección de salto
16	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	branchI
17	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	Dirección de salto 20
18	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	a or b
19	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	end

Tabla 4.1: Tabla de instrucciones código máquina subprograma y su equivalencia pseudocódigo

En la Tabla 4.1 mostramos el código máquina de cada instrucción junto con el tipo de la instrucción que implementa.

Tras mostrar el código máquina presentamos en Código 4.1, adjuntamos el código en lenguaje de alto nivel, que generaría el código máquina que se muestra en la Tabla 4.1. El código, se encuentra comentado para que se pueda realizar una traslación sencilla de un código a otro para poder hacerse así una idea sencilla de cómo funciona el programa.

```
Programa.
        a=2;  --Linea 0 tabla 4.1
        b=15; --Linea 2 tabla 4.1
        c=7;  --Linea 4 tabla 4.1
        d=10;  --Linea 6 tabla 4.1
        while (c<=d){  --Lineas de 8 a 13 tabla 4.1
        b =b-a;  --Linea 10 tabla 4.1
        c++;  --Linea 11 tabla 4.1
         }
        If  (a != b){  --Linea 14 tabla 4.1
        }else{  --Linea 16 tabla 4.1
        a =a or b;  --Linea 18 tabla 4.1
        }
        End;  --Linea 19 tabla 4.1
```

Código 4.1: Código alto nivel del banco de pruebas “bucle while y alternativa if/else”

Seguidamente explicamos el código que implementa este programa en VHDL.

El programa se implementa dentro de la matriz de vectores declarada como “mem_data” y se puede apreciar en Código 4.2, que cada instrucción corresponde con un (instrucciones de una palabra) vector de dicha matriz o con dos para el caso de instrucciones de dos palabras.

El programa en sí trata de demostrar el funcionamiento de la alternativa “if/else” y de un bucle que en este caso es “while”. Anteriormente hemos presentado tanto el código de alto nivel de un programa equivalente que realizaría las mismas funciones en lenguaje C, así como el código máquina, que es el que ejecuta nuestro circuito y que hemos presentado en la tabla precedente.

Es fácil comprobar, que el código máquina descrito en dicha tabla, coincide exactamente con el código implementado en la arquitectura “whileyif” del archivo “mem”. Por tanto los resultados obtenidos de la simulación (ejecución) de dicho programa, son los que deben aportar la validez del circuito implementado.

Existe una correspondencia línea a línea entre la Tabla 4.1 y el Código 4.2 que hace innecesaria la explicación de dicho código, pues como se aprecia, son iguales los datos de instrucción, en ambos casos. Solo comentar, que las instrucciones se almacenan en un matriz de vectores de 16 posiciones, que es donde queda implementado el código a ejecutar.

El funcionamiento del elemento “mem” ya se explicó en la sección anterior (pág. 110 y siguientes) por tanto solo mostraremos de aquí en adelante el código que almacena el subprograma a ejecutar.

```
architecture whileyif of mem is
begin
memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;
variable mem_data : t_mem :=
        ("0010000000000000",    ---    0 loadI reg 0
         "0000000000000010",    ---    1 a=2
         "0010000000000001",    ---    2 loadI reg 1,
         "0000000000001111",    ---    3 15
         "0010000000000010",    ---    4 loadI reg2
         "0000000000000111",    ---    5 c=7
         "0010000000000011",    ---    6 load reg 3
         "0000000000001010",    ---    7 10
         "0011000000010011",    ---    8 bgtI reg 3 reg2
         "0000000000001110",    ---    9 E
         "0111000000001000",    ---    A b=b-a
         "0011100000000010",    ---    B inc c++
         "0010100000001111",    ---    C braI
         "0000000000001000",    ---    D 8
         "1011100000010011",    ---    E braI  if reg 3 es != a reg 2
         "0000000000010010",    ---    F 18
         "0010100000000000",    ---    10 branchI (else)
         "0000000000010100",    ---    11 20
         "0101000000000001",    ---    12 or reg 0 reg 1
         "1111100000000000",    ---    13 END
```

Código 4.2: Fragmento de código “mem.vhd” arquitectura “whileyif”

4.3 BANCO DE PRUEBAS “BUCLE INTERIOR A UN BUCLE”

En este caso, demostrado anteriormente el funcionamiento del bucle y la alternativa “if”. Pasamos a presentar un caso de prueba, en el que se ejecuta un bucle en el interior de un bucle. Tratamos de probar que es posible ejecutar un bucle interior a otro bucle, para lo cual usamos un bucle “for”, que se ejecuta en el interior de un bucle “while”.

En la Tabla 4.2, se nos muestra el código máquina junto con pequeño comentario, qué nos dice que acción corresponde a cada línea de código, entendemos que se comenta por sí mismo cuando lo comparamos con Código 4.3, que también se encuentra comentado.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	descripción
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	LoadI en reg 0 -- a=2;
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	--2
2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	LoadI en reg 1 b=15;
3	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	--15
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	loadI en reg 2--c=7;
5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	--7
6	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	LoadI en reg 3--d=10;
7	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	--10
8	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	LoadI en reg 4--f=2;
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	--2
10	1	0	1	1	1	0	0	0	0	0	0	1	0	0	0	1	branchI si equal
11	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	24
12	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	d=d-a;
13	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	loadI --f=0,
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	1	0	1	1	1	0	0	0	0	0	1	0	1	1	0	0	Branch I if equal -for
16	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	21
17	0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	Add 3 0--d=d+a;
18	0	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	Inc 4--f++;
19	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	branchI
20	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	15
21	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	Inc 2--c++;
22	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	branchI
23	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	10
24	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	End.

Tabla 4.2: Tabla de instrucciones código máquina subprograma y su equivalencia en pseudocódigo

```
a= 2;  --Linea 0 tabla 4.2
b=15;  --Linea 2 tabla 4.2
c= 7;  --Linea 4 tabla 4.2
d=10;  --Linea 6 tabla 4.2
while (c!= b){  --Linea 8 tabla 4.2
d= d-a;  --Linea 12 tabla 4.2
for (f=0; f<2;f++){  --Linea 13 tabla 4.2
d= d+a;}  --Linea 17 tabla 4.2
c++;  --Linea 18 tabla 4.2
}
End; --Linea 24 tabla 4.2
```

Código 4.3: Código alto nivel banco de pruebas “Bucle interior a un bucle”

Una vez descrito el código máquina del caso de prueba, así como su traslación a código de alto nivel. Mostramos el correspondiente código en VHDL, en Código 4.4 que al fin y al cabo no es mas, que una copia del código máquina expresado en Tabla 4.2, trasladado a la posición correspondiente de la matriz de vectores de la memoria.

```
architecture bucleinterno of mem is
begin
memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;
variable mem_data : t_mem :=
        ("0010000000000000",    ---    0 loadI reg 0
         "0000000000000010",    ---    1        a=2
         "0010000000000001",    ---    2 loadI reg 1,
         "0000000000001111",    ---    3 15
         "0010000000000010",    ---    4 loadI reg2
         "0000000000000111",    ---    5 c=7
         "0010000000000011",    ---    6 load reg 3
         "0000000000001010",    ---    7 10
         "0010000000000100",    ---    8 loadI reg 4
         "0000000000000010",    ---    9 f=2
         "1011100000010011",    ---    A branch is equal reg2 reg 3
         "0000000000011000",    ---    B 24
         "0111000000011000",    ---    C d=d-a
         "0010000000000101",    ---    D f=0
         "0000000000000000",    ---    E 0
         "1011100000101100",    ---    F branch if equal (for
         "0000000000010101",    ---    21
         "0110100000011000",    ---    11 d=d+a
         "0011100000000101",    ---    12 f++
         "0010100000000000",    ---    13 branchI
         "0000000000001111",    ---    14 f
         "0011100000000010",    ---    15 inc c++
         "0010100000000000",    ---    16 branchI
         "0000000000001010",    ---    17 10
         "1111100000000000",    ---    18 end
```

Código 4.4: Fragmento de código “mem.vhd” arquitectura “bucleinterno”

4.4 BANCO DE PRUEBAS “COPIAR DE MEMORIA A MEMORIA”

En este caso de prueba, vamos a realizar la copia de una parte de la memoria en otra parte de la memoria. Este subprograma, está descrito en el texto base y se muestra en la Tabla 4.3, este código lo que realiza es la copia de un sector de datos de la memoria, en otro sector de la memoria. Presentamos a continuación la tabla que presenta el código máquina que se ejecuta en el subprograma.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	descripción
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	LoadI --a=32;
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	32
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	LoadI 2-- b=16;
3	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	16
4	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	LoadI 6-- c=36;
5	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	36
6	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	Load 2 – 4;
7	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	Store 4 1
8	0	0	1	1	0	0	0	0	0	0	0	1	0	1	1	0	BGT 2 6
9	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	56
10	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	1	Load 1 –5
11	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	Inc 2 --b++;
12	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	Inc 1 -- a++;
13	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	Branch
14	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	6
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	ELEMENTOS A COPIAR.
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	“
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	“
18	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	“
19	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	“
20	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	“
21	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	“
22	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	“
23	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	“
24	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	“
25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	“
26	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	“
27	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	“
28	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	“
29	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	“
30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	“

Tabla 4.3: Tabla donde se presenta el código máquina que se ejecuta en el subprograma

El código de alto nivel correspondiente al código máquina de la Tabla 4.3 de este programa sería el que se muestra en Código 4.5.

```
a = 32;  --Linea 0 tabla 4.3
b = 16;  --Linea 2 tabla 4.3
c = 36;  --Linea 4 tabla 4.3
while ( c>b){ --Linea 8 tabla 4.3
copiar (*a,*b)  --Linea 7 a 15 tabla 4.3
a++;  --Linea 12 tabla 4.3
b++;  --Linea 11 tabla 4.3
}
```

Código 4.5: Código alto nivel banco de pruebas “copiar de memoria a memoria”

El código que se muestra en la Tabla 4.5, implementado en cada posición de la matriz de vectores “mem_data” se nos muestra en Código 4.6.

```
architecture copiarmem of mem is
begin
memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;
variable mem_data : t_mem :=
        ("0010000000000001",    ---     0 loadI address
         "0000000000010000",    ---     1       10
         "0010000000000010",    ---     2 loadI 2,
         "0000000000110000",    ---     3 30
         "0010000000000110",    ---     4 loadI 6,
         "0000000000100000",    ---     5 20
         "0000100000001011",    ---     6 load 1, 3
         "0001000000011010",    ---     7 store 3
         "0011000000001110",    ---     8 bgtI 1, 6
         "0000000000001110",    ---     9 E
         "0011100000000001",    ---     A inc 1
         "0011100000000010",    ---     B inc 2
         "0010100000001111",    ---     C braI
         "0000000000000110",    ---     D 06
         "1111100000000000",    ---     E parar
         "1111100000000000",    ---     F
         "0000000000000001",    ---     10
         "0000000000000010",    ---     11
         "0000000000000011",    ---     12
         "0000000000000100",    ---     13
         "0000000000000101",    ---     14
         "0000000000000110",    ---     15
         "0000000000000111",    ---     16
         "0000000000001000",    ---     17
         "0000000000001001",    ---     18
         "0000000000001010",    ---     19
         "0000000000001011",    ---     1A
         "0000000000001100",    ---     1B
         "0000000000001101",    ---     1C
         "0000000000001110",    ---     1D
         "0000000000001111",    ---     1E
         "0000000000010000",    ---     1F
         "0000000000000000",    ---     20
         "0000000000000000",    ---     21
         "0000000000000000",    ---     22
         "0000000000000000",    ---     23
         "0000000000000000",    ---     24
         "0000000000000000",    ---     25
         "0000000000000000",    ---     26
         "0000000000000000",    ---     27
         "0000000000000000",    ---     28
         "0000000000000000",    ---     29
         "0000000000000000",    ---     2A
```

Código 4.6: Fragmento código “mem.vhd”, arquitectura “copiarmem”

Las restantes posiciones de memoria están vacías, por tanto están inicializadas con todos sus bits a cero. Como es el caso de las líneas a partir de 1F. En las posiciones 32 en adelante, es donde se copiaran las posiciones desde la F a la 1F.

4.5 BANCO DE PRUEBAS “OPERACIONES MATEMÁTICAS”

En este caso de prueba, se va a realizar la prueba de todas las operaciones matemáticas y lógicas, que se han implementado. Sirviendo esto, como demostración del correcto funcionamiento de las instrucciones que actúan sobre la “ALU” y sobre el registro de desplazamiento. Se presenta a continuación en Tabla 4.4 del código máquina del subprograma.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	descripción
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	LoadI reg0 -a=178
1	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	0	--178
2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	LoadI reg 1-b=409
3	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	1	--409
4	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 -reg 2
5	0	1	1	0	1	0	0	0	0	0	0	0	1	0	1	0	add 2 1--c=b+a;
6	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 ,reg 2
7	0	1	0	0	1	0	0	0	0	0	0	0	1	0	1	0	and 2 1-c=a and b
8	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 ,reg 2
9	0	1	0	1	0	0	0	0	0	0	0	0	1	0	1	0	or 2 1 --c=a or b;
10	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 ,reg 2
11	0	1	0	1	1	0	0	0	0	0	0	0	1	0	1	0	Xor 2 1-c= a xor b;
12	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 reg 2
13	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	Shift left c--c<<
14	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 reg 2
15	1	1	0	1	1	0	0	0	0	0	0	0	1	0	1	0	Shift right c--c>>
16	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 ,reg 2
17	0	1	1	1	0	0	0	0	0	0	0	0	1	0	1	0	Sub 2 1 -- c=b-a;
18	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	END
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

Tabla 4.4: Tabla de instrucciones código máquina subprograma y su equivalencia pseudocódigo

En Código 4.7 tenemos el código de alto nivel que generaría el código máquina que hemos presentado en la Tabla 4.4. Entendemos que por su sencillez requiere pocos comentarios, pues se encuentra comentada su correspondencia entre la línea de código máquina y la línea de código de alto nivel.

Básicamente, se compone de las operaciones lógico matemáticas que se han implementado en la “ALU” y en el registro de desplazamiento, nominado como “Shifter” en el esquema presentado en la sección anterior en la Figura 3.1.

```
a= 178;  --Linea 0 tabla 4.4
b=409;  --Linea 2 tabla 4.4
c= a + b;  --Linea 5 tab5a 4.4
c= a and b;  --Linea 7 tabla 4.4
c= a or b;  --Linea 9 tabla 4.4
c= a xor b;  --Linea 11 tabla 4.4
c= <<a ;  --Linea 13 tabla 4.4
c= a >>;--Linea 11 tabla 4.4
c= b - a;  --Linea 17 tabla 4.4
end;
```

Código 4.7: Código alto nivel banco de pruebas “operaciones matemáticas”

El código VHDL que implementa el caso de prueba que hemos descrito se nos muestra en Código 4.8, es bastante sencillo y se limita a la traslación del código máquina de la tabla a la matriz de vectores “mem_data”.

```
architecture operbasicas of mem is
begin
memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;
variable mem_data : t_mem :=
        ("0010000000000000",    ---    0 loadI address
         "0000000010110010",    ---    1          a= 178
         "0010000000000001",    ---    2 loadI 1,
         "0000000110011001",    ---    3 b= 409
         "0001100000000010",    ---    4 move reg0 a reg 2
         "0110100000010001",    ---    5 c= b+a
         "0001100000000010",    ---    6 move reg0 a reg 2
         "0100100000010001",    ---    7 c=a and b
         "0001100000000010",    ---    8 move reg0 a reg 2
         "0101000000010001",    ---    9 c= a or b
         "0001100000000010",    ---    A move reg0 a reg 2
         "0101100000010001",    ---    B c= a xor b
         "0001100000000010",    ---    C move reg0 a reg 2
         "1101000000000010",    ---    D shift left a
         "0001100000000010",    ---    E move reg0 a reg 2
         "1101100000000010",    ---    F shift right a
         "0001100000000010",    ---    10 move reg0 a reg 2
         "0111000000001010",    ---    11 c=b-a
         "1111100000000000",    ---    12 end
         "0000000000000000",    ---    13
```

Código 4.8: Fragmento de código “mem.vhd” arquitectura “operbasicas”

4.6 SIMULACIÓN DEL BANCO DE PRUEBAS VHDL

Para la simulación del banco de pruebas, deberemos disponer de un entorno de simulación adecuado para el lenguaje VHDL. En el Capítulo 1, hemos introducido cómo instalar el entorno de simulación ModelSim PE Student Edition, no necesariamente debe de ser éste, pudiendo utilizar cualquier otro entorno a condición que soporte el código desarrollado.

En este caso, la simulación ha sido realizada con dicho entorno y los resultados mostrados a continuación, no deben variar más que en el formato de la imagen, no así en los valores obtenidos, que necesariamente han de ser los mismos. Procedemos primero a explicar cómo debemos compilar los archivos que presentamos en el CD, para a continuación poder realizar las simulaciones del "banco de pruebas".

Es necesario resaltar, que en contra de lo que sucede con PowerDEVS, para poder simular el proyecto será necesario disponer o tener instalado el entorno ModelSim Pe Student Edition versión 10.2 o superiores, no teniendo constatado el funcionamiento en versiones inferiores.

En el CD del proyecto se adjunta una carpeta nominada como VHDL, en la que se encontraran los archivos que conforman nuestro proyecto tal y como podemos apreciar en la Figura 4.1.

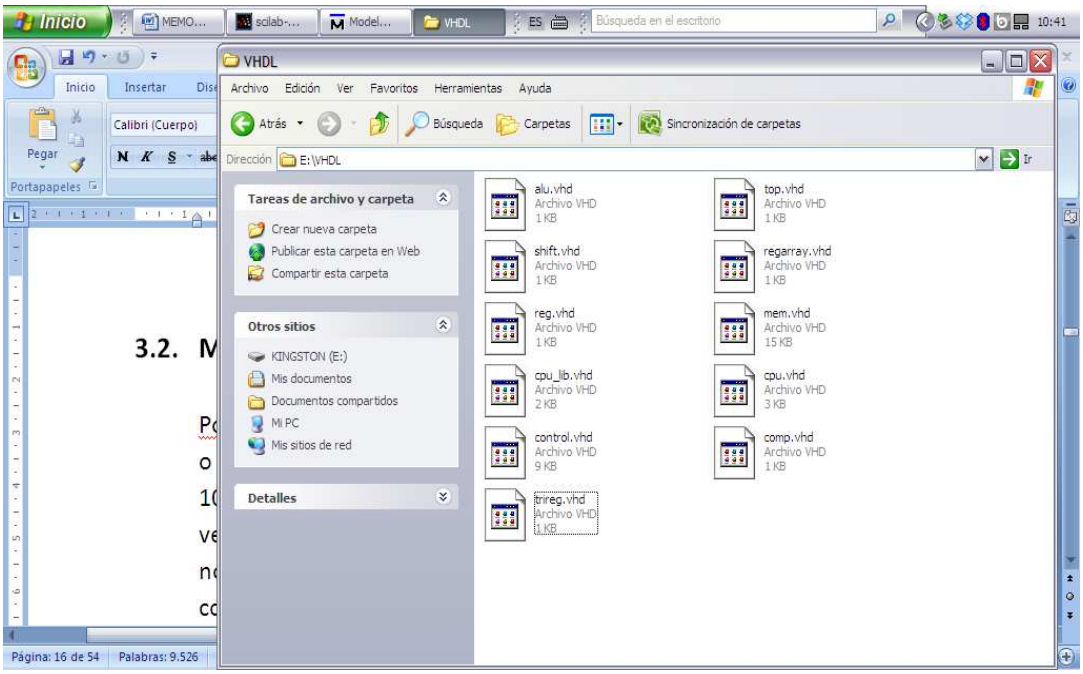


Figura 4.1: Simulación proyecto VHDL paso 1

Estos archivos deberemos incorporarlos a un nuevo proyecto que deberemos crear de la manera que pasamos a mostrar en la Figura 4.2. En la pantalla inicial seleccionaremos el menú new/project...

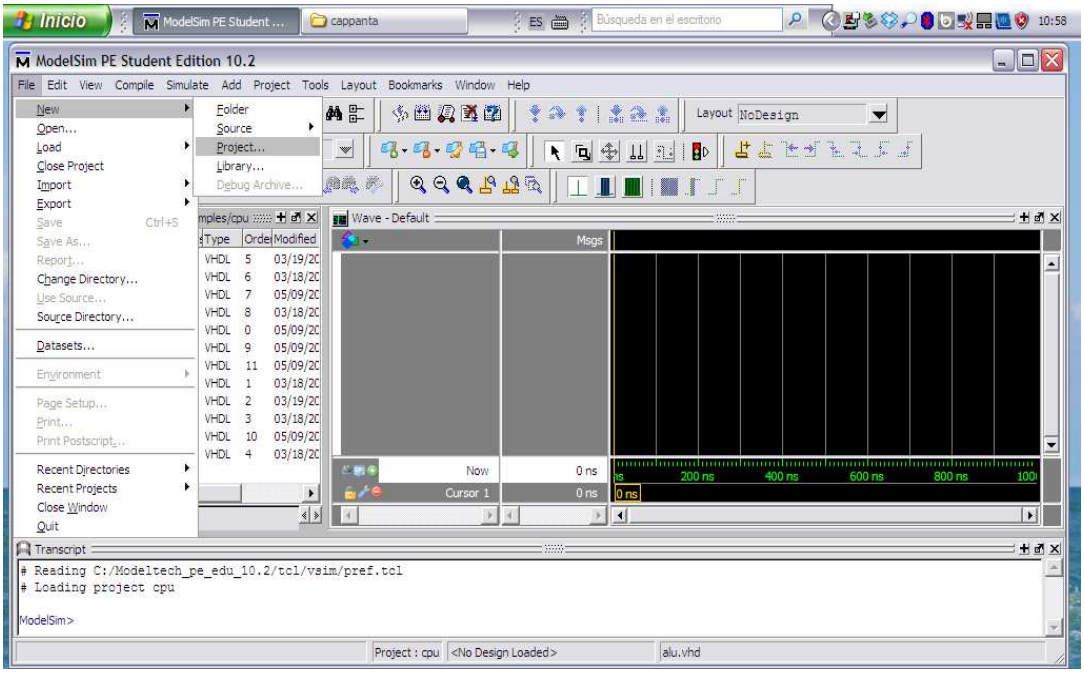


Figura 4.2: Simulación proyecto VHDL paso2

Nos aparecerá la pantalla que mostramos en la Figura 4.3 donde deberemos ingresar el nombre del proyecto (el que queramos) así como la ruta donde se guardará.

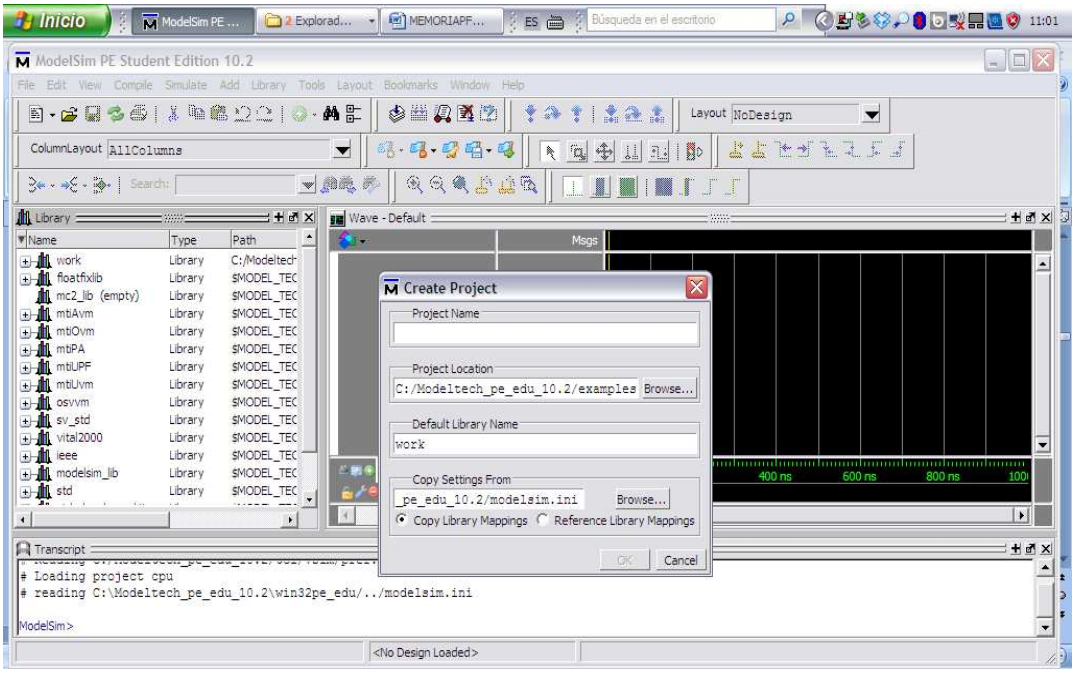


Figura 4.3: Simulación proyecto VHDL paso 3

Una vez rellenados estos datos se pasará a la pantalla siguiente mostrada en la Figura 4.4, donde deberemos seleccionar la opción “add existing file”.

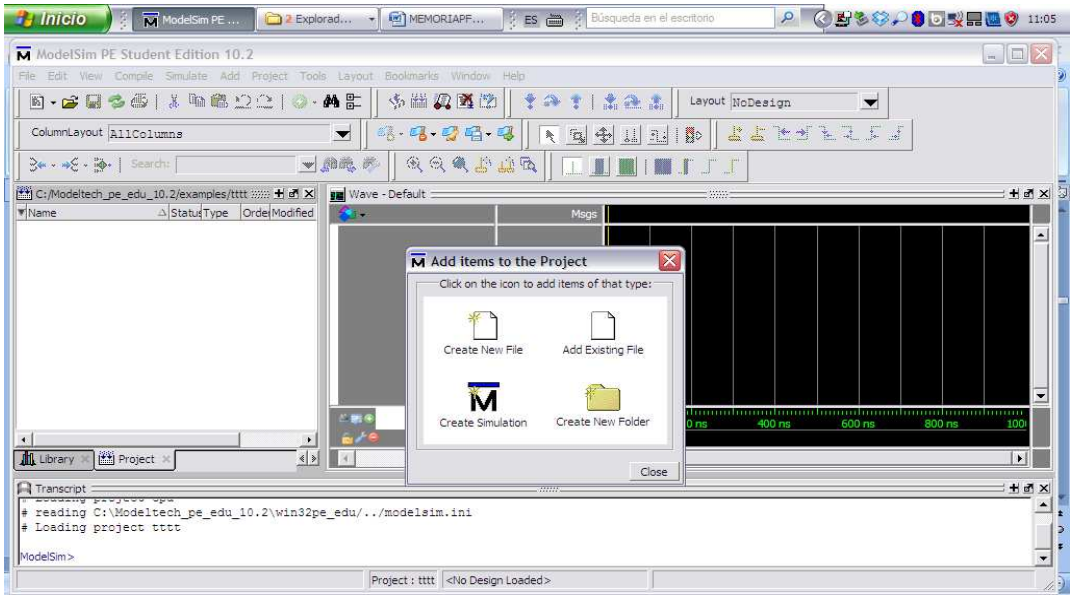


Figura 4.4: Simulación proyecto VHDL paso 4

En la siguiente pantalla, mostrada en la Figura 4.5, podremos seleccionar todos los archivos que configuran este proyecto e ingresarlos en el nuevo proyecto, para poder proceder a su simulación.

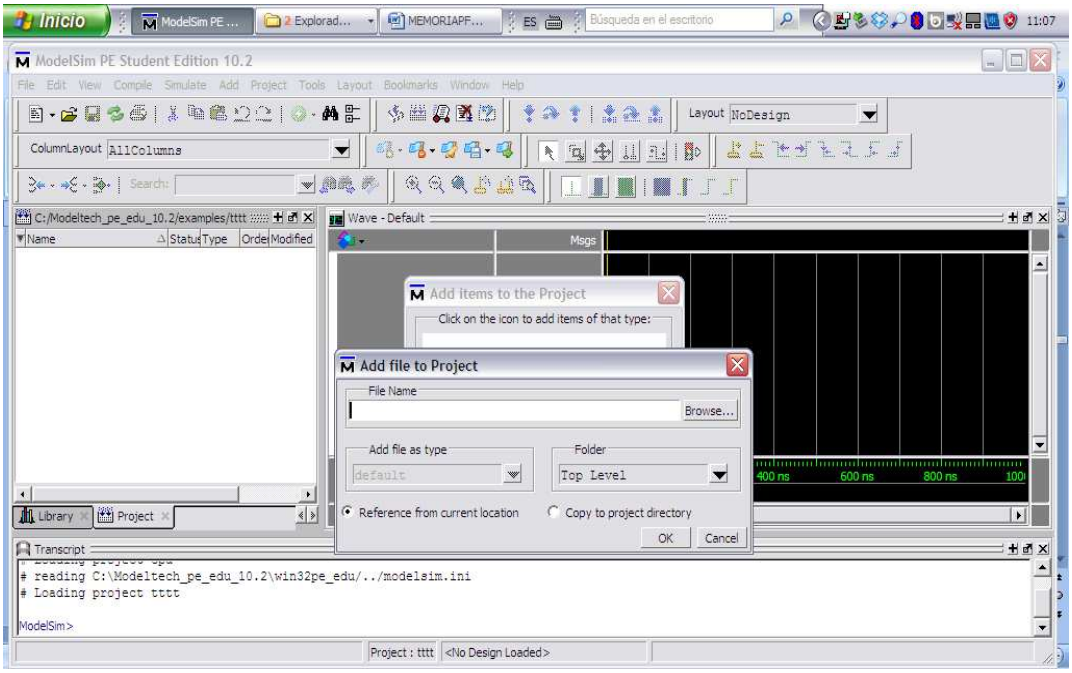


Figura 4.5: Simulación proyecto VHDL paso 5

A continuación deberemos compilar nuestro proyecto pudiendo establecer su orden de compilación de manera automática, de todas formas se presenta en la Figura 4.6, el orden utilizado en este proyecto.

El orden de compilación utilizado en este caso, no significa que sea el único posible. Una vez compilado el proyecto, si éste es correcto, podemos proceder a la simulación del mismo.

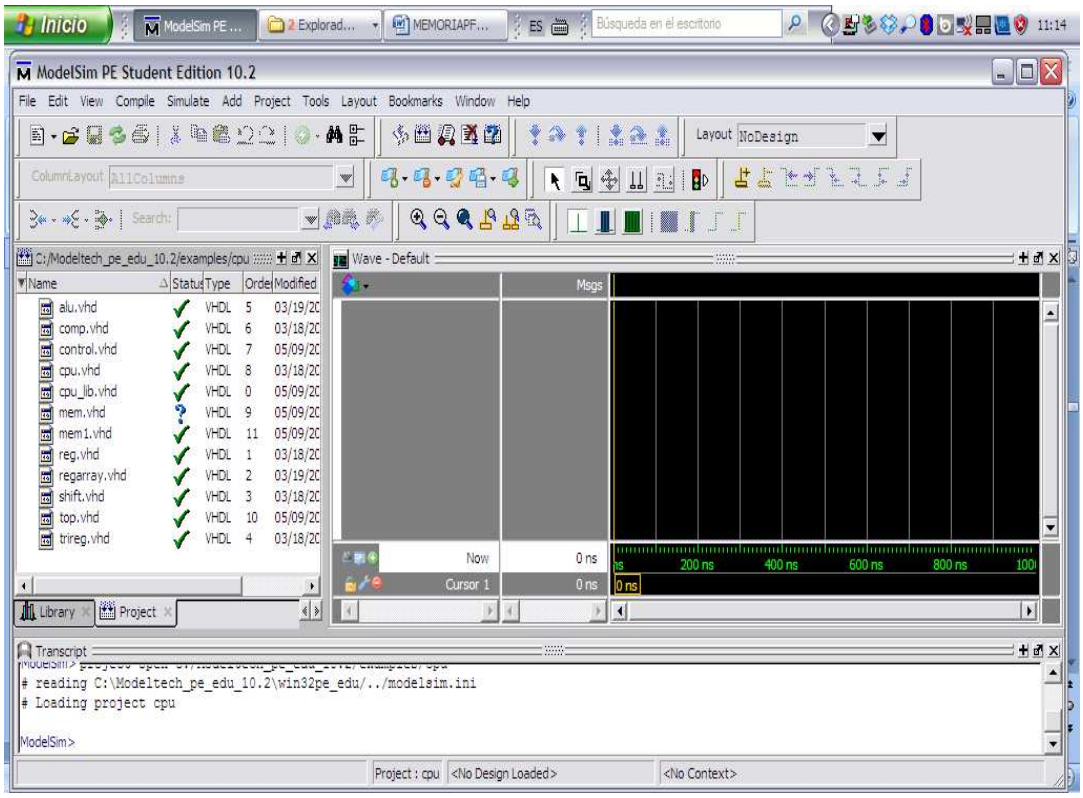


Figura 4.6: Simulación proyecto VHDL, orden de simulación

Para la simulación del banco de pruebas con el entorno ModelSim, al igual que hemos hecho en el entorno PowerDEVS, vamos solo a centrarnos en las acciones básicas. Estas nos conducirán a realizar con éxito la simulación del banco de pruebas, que hemos diseñado para el PFC, sin entrar a fondo, en cómo se utiliza el entorno, pues existen manuales y tutoriales que lo explican con mucha mayor precisión y rigurosidad que podríamos hacerlo nosotros.

En ModelSim todos los bancos de prueba (que son iguales a los de PowerDEVS en cuanto a resultados que nos debe ofrecer, no tanto en cuanto a la forma), están agrupados en el archivo “mem”, en forma de diferentes arquitecturas de una misma entidad.

Estas arquitecturas (primera, whileyif, etc.) para poder ser simuladas en el entorno, deberán ser seleccionadas en el archivo top.vhd de la manera que indicamos en la Figura 4.7.

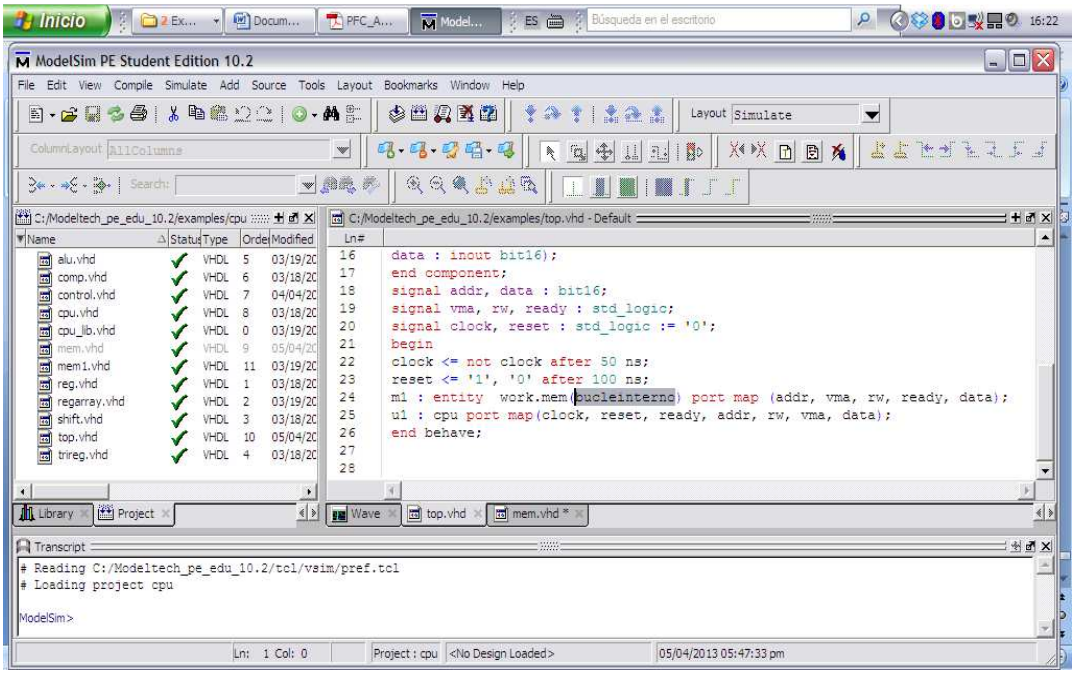


Figura 4.7: Simulación proyecto VHDL, selección de arquitectura a simular

La parte que aparece seleccionada del archivo top.vhd, es donde debemos indicar sobre qué arquitectura deseamos realizar la simulación, en este caso sobre la arquitectura “bucleinterno”, a continuación deberemos volver a realizar la compilación del proyecto como se muestra en la Figura 4.8.

Recordemos que disponemos de cuatro arquitecturas, cada una de las cuales se corresponde con un banco de pruebas, por tanto cada vez que simulemos un banco distinto deberemos repetir esta operación.

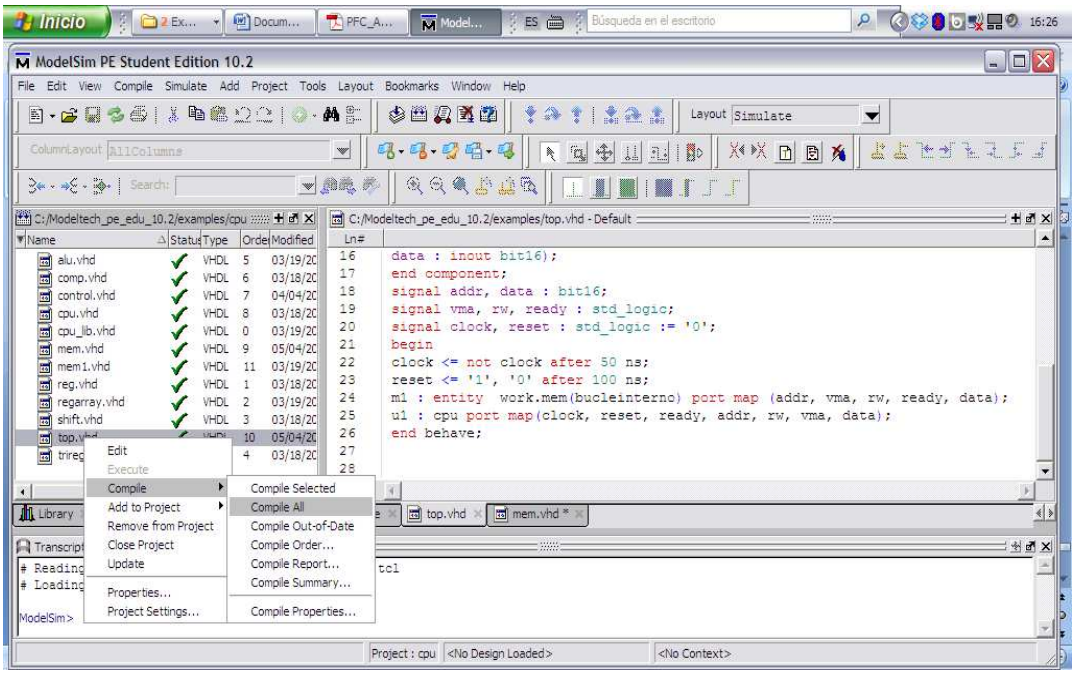


Figura 4.8: Simulación proyecto VHDL, compilación de arquitectura seleccionada

Una vez realizada y si todo está correcto, aparecerá un mensaje en la consola de salida en el que se nos indica, si existe algún error de programación o de lo contrario la compilación es correcta, como el caso mostrado en la Figura 4.9.

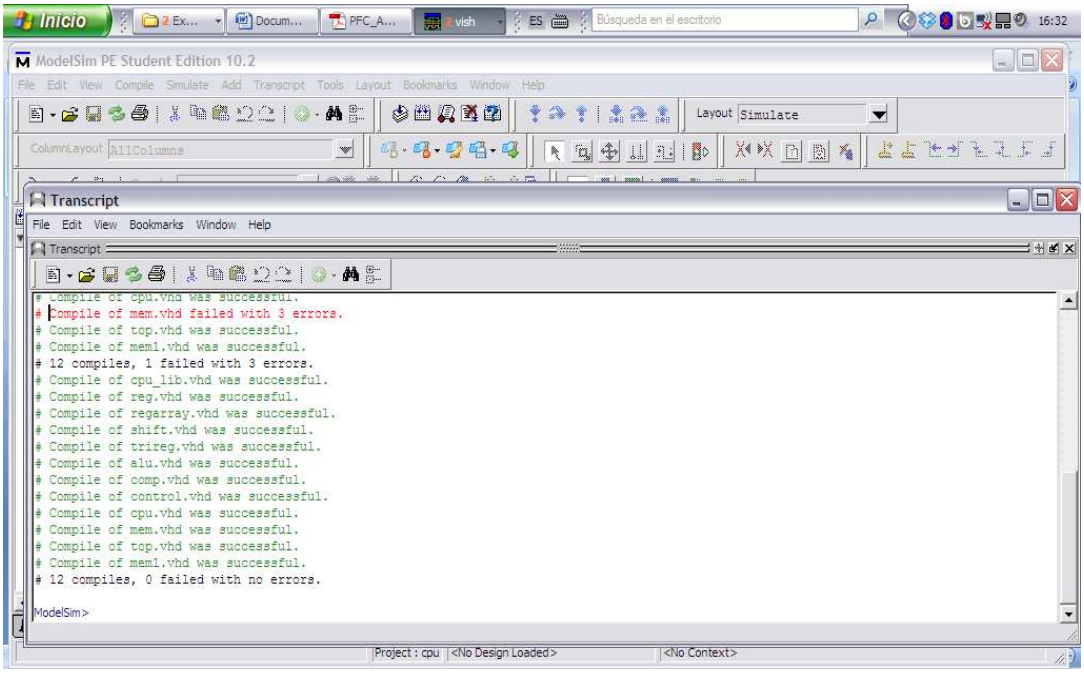


Figura 4.9: Simulación proyecto VHDL, resultado de la simulación

Después de esto vamos a proceder a simular realmente la arquitectura seleccionada, y para eso seleccionaremos simulate→ start simulation; seleccionaremos top dentro del directorio "work" tal y como se indica en la Figura 4.10.

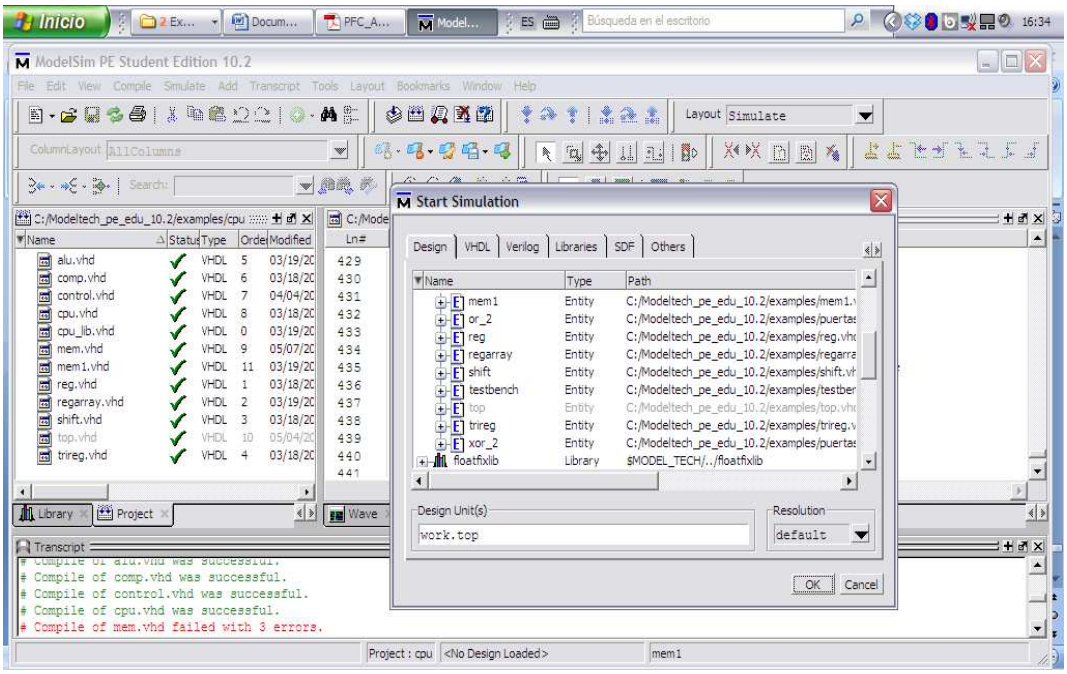


Figura 4.10: Simulación proyecto VHDL, selección de archivo para simulación

Una vez hecho esto, pulsamos sobre el "OK" y seleccionando la pestaña nominada como Wave, con el botón derecho, seleccionaremos qué señales o variables queremos que aparezcan en la simulación. En este caso como son muchas y como solo nos interesan los resultados finales; aconsejamos se utilicen aquellos registros o posiciones de memoria donde se prevea, que se encuentran alojados los resultados de la simulación. De lo contrario puede resultar muy farragoso el tener tantas variables en pantalla.

Por ejemplo en la imagen de la Figura 4.11, solo se han añadido las variables que se corresponden con las posiciones del registro de memoria, donde seguramente se depositarán los resultados de la simulación.

Según en qué caso, nos puede interesar que nos aparezcan las posiciones de la memoria o cualquier otra variable, por tanto procederemos a seleccionar aquellas que nos interesen.

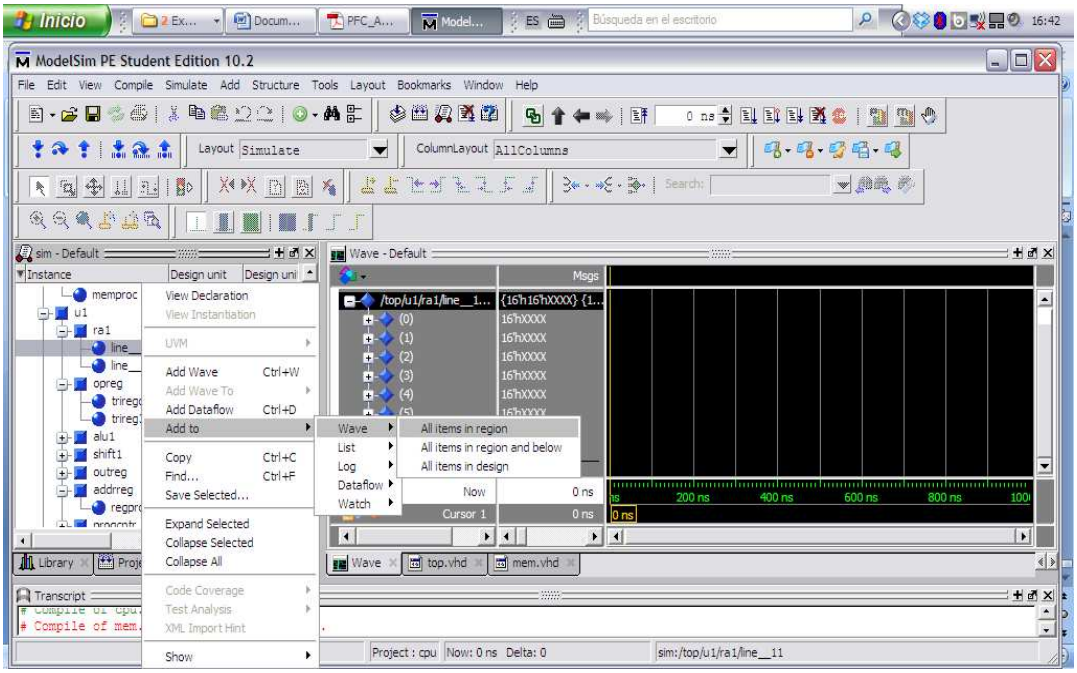


Figura 4.11: Simulación proyecto VHDL, selección de variables y señales

Llegado hasta aquí, solo nos queda pulsar sobre simule-> Run-> RunAll. Como se muestra en la Figura 4.12, comenzara la simulación, si pulsamos sobre "end simulation" ésta se detendrá, al desplazar el cursor veremos los valores que toman las señales y variables, que hayamos seleccionado previamente, como varían en función del tiempo base de simulación.

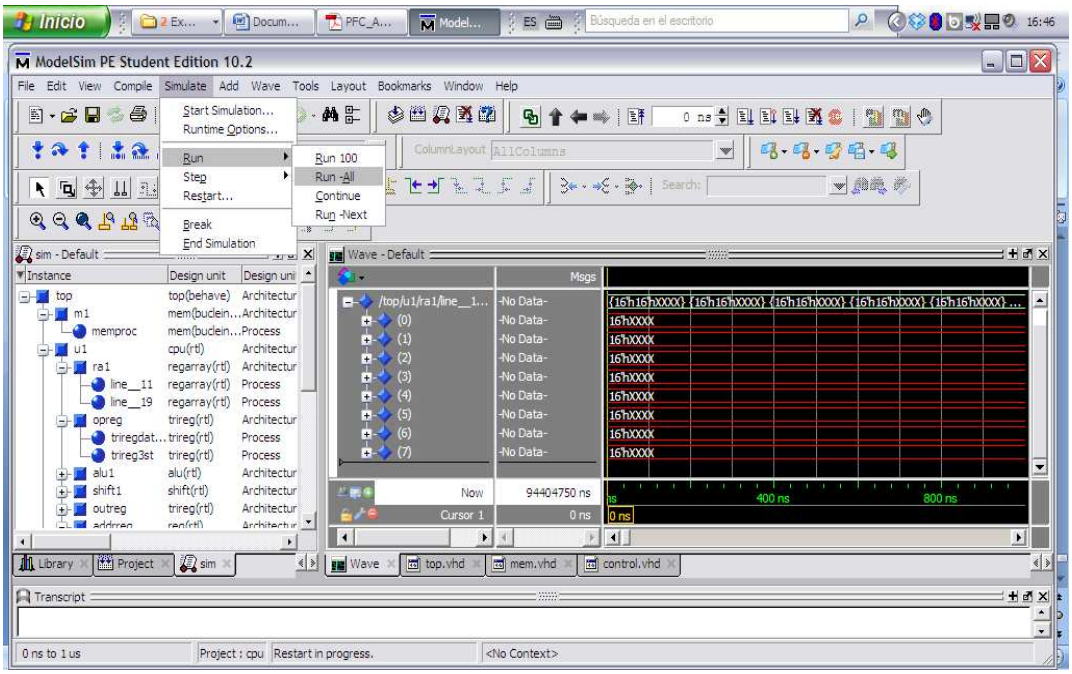


Figura 4.12: Simulación proyecto VHDL, selección de tiempo de simulación

Podremos iniciar, cuantas veces deseemos la simulación y variar las señales seleccionadas en función de lo que nos interese conocer. Como se puede apreciar es una forma muy cómoda, de conocer en todo momento, qué está haciendo nuestro programa y qué valores toman tanto las señales como las variables, lo cual representa un mecanismo de depuración de errores muy potente.

4.7 ANÁLISIS DE LOS RESULTADOS DE LA SIMULACIÓN

Explicado cómo podemos realizar nuestras propias simulaciones del proyecto. Procedemos a continuación, a mostrar los resultados obtenidos de nuestras propias simulaciones. Resultados que por otra parte, servirán por comparación con los resultados obtenidos mediante PowerDEVS, para la validación del circuito implementado. Desglosamos por tanto según cada caso de prueba, los resultados obtenidos.

Bucle “while” y alternativa “if/else”:

El código fuente que vamos a simular, desprende de su ejecución a mano el resultado siguiente. Al finalizar el programa; a =2, b= 7, c= 11, d= 10 (recordemos que la alternativa “else” no procede pues solo procederá en el caso de que sean iguales a y b). El código está disponible en Código 4.1, invitamos al lector a ejecutar dicho código a mano o implementarlo en un ordenador en el lenguaje que se desee, para comprobar la veracidad de los mismos.

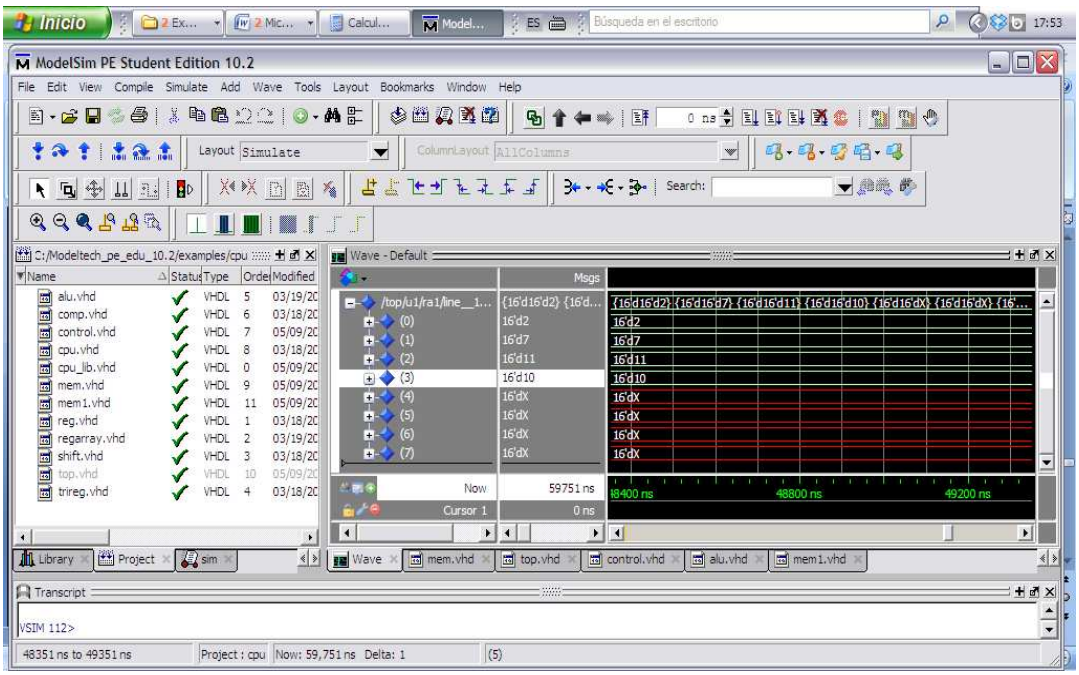


Figura 4.13: Imagen de resultados ejecución subprograma en ModelSim

El resultado se obtiene en los registro de 0 a 3 (resaltado en la Figura 4.13), el registro 0 corresponde con la variable “a” y es igual a 2; el registro 1 corresponde con la variable “b” y es igual a 7; el registro 2 corresponde con la variable “c” y es igual a 11; el registro 3 corresponde con la variable “d” y es igual a 10; valor de 10 corresponde con d= 10. Si ejecutamos el código a mano, son los valores que deberíamos haber obtenido, como hemos indicado en el párrafo anterior.

Bucle interior a un bucle:

Si procedemos a la ejecución a mano (o mediante un programa en lenguaje de alto nivel) de Código 4.3. Obtendríamos los resultados siguientes: a=2, b = 15 (ambos valores no cambian en la ejecución), c= 15, y d= 26 estos resultados los obtenemos tal y como se aprecia en la Figura 4.12 en los registros 0-> a, 1 ->b, 2->c y 3->d cuya correspondencia con las variables hemos indicado y se puede comprobar en la Tabla 4.2 o en Código 4.4. Esto concuerda con los datos que mostramos de la ejecución en ModelSim y que son fácilmente comprobables.

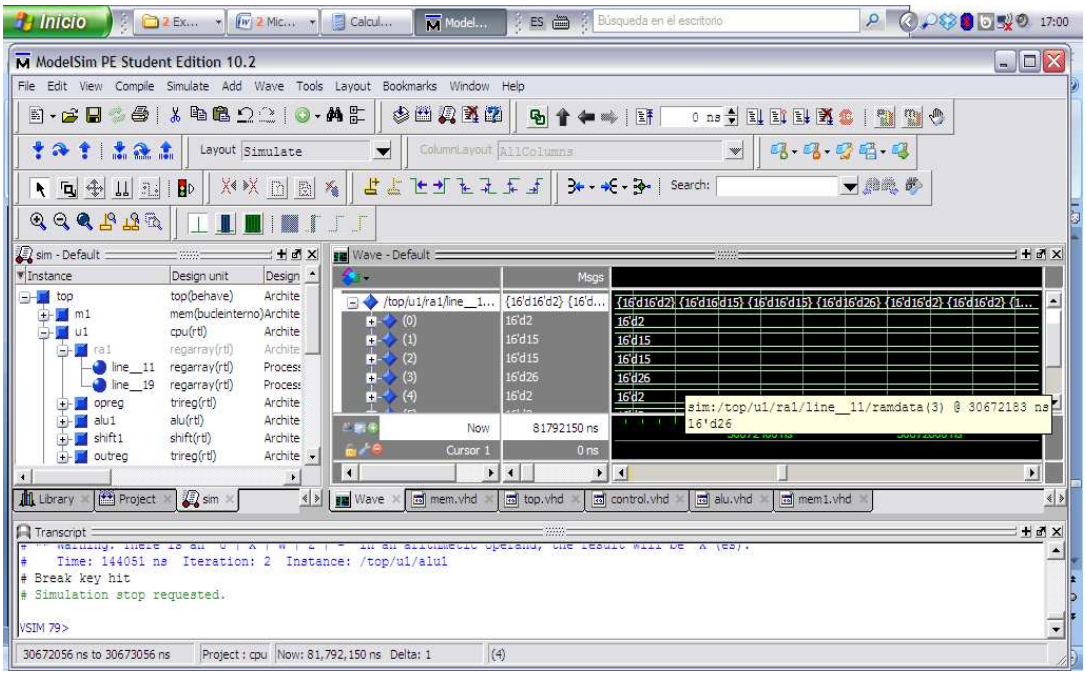


Figura 4.14: Imagen de resultados ejecución subprograma en ModelSim "bucle interior a un bucle"

Como se puede apreciar en la Figura 4.14, el resultado de la variable "d", se encuentra en el registro nº 3 y arroja un valor 26. El resto de variables se muestran del reg 0 a reg 2.

Copiar de memoria a memoria:

A continuación, se muestran los resultados obtenidos de la simulación del programa equivalente (archivo “mem.vhd” arquitectura “copiarmem”) para VHDL. Presentamos en varias imágenes, el valor de las posiciones de memoria al inicio del subprograma y con los valores finales, después de la ejecución del mismo.

Inicialmente las posiciones de memoria referidas están a 0, tal y como podemos apreciar en la Figura 4.15.

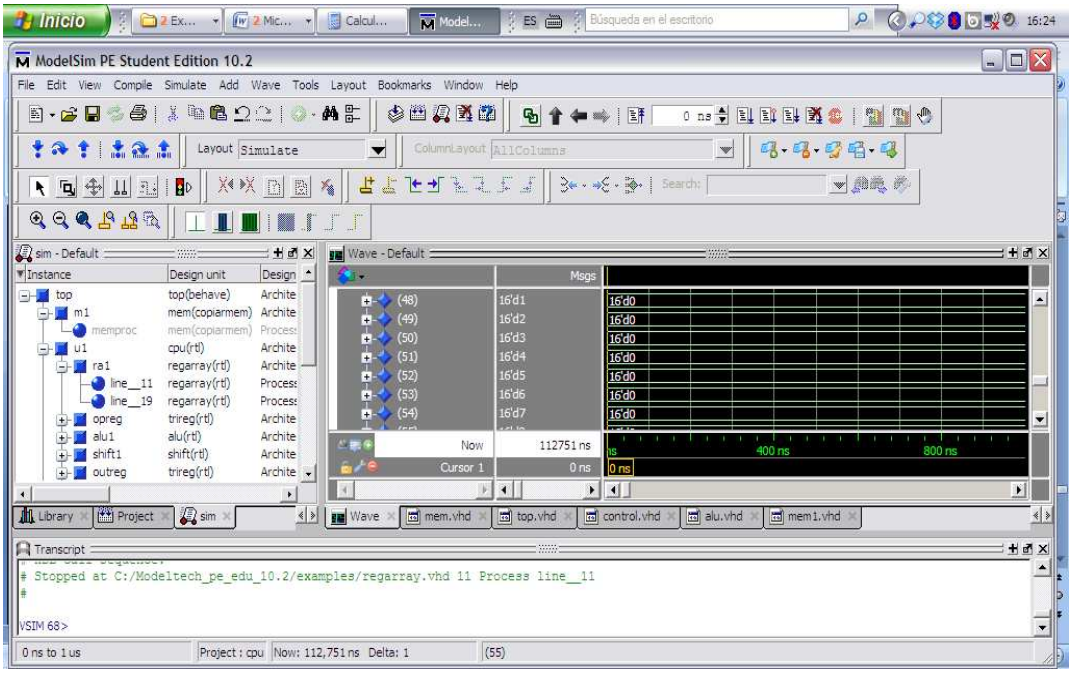


Figura 4.15: Imagen de resultados ejecución subprograma en ModelSim, datos de inicio

Al final de la ejecución del subprograma, tienen los valores que se deseaban copiar en dichas posiciones, como podemos observar en la Figura 4.16 y 4.17. En la Figura 4.15, podemos observar que inicialmente se encontraban inicializadas al valor “0”.

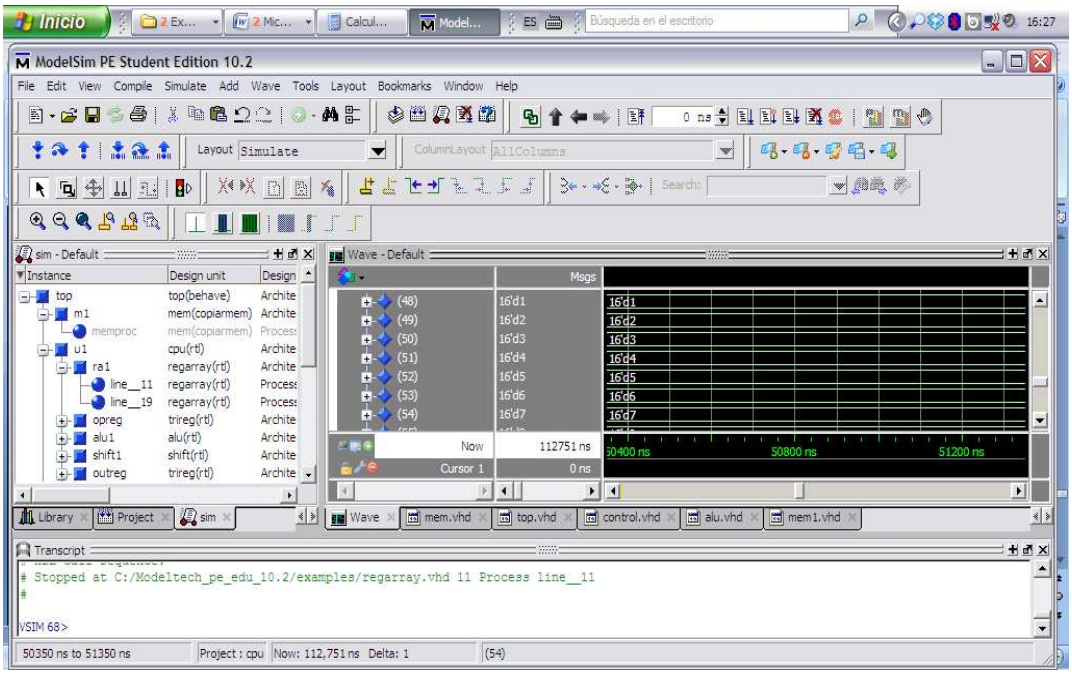


Figura 4.16: Imagen de resultados ejecución subprograma en ModelSim, resultado final, parte 1

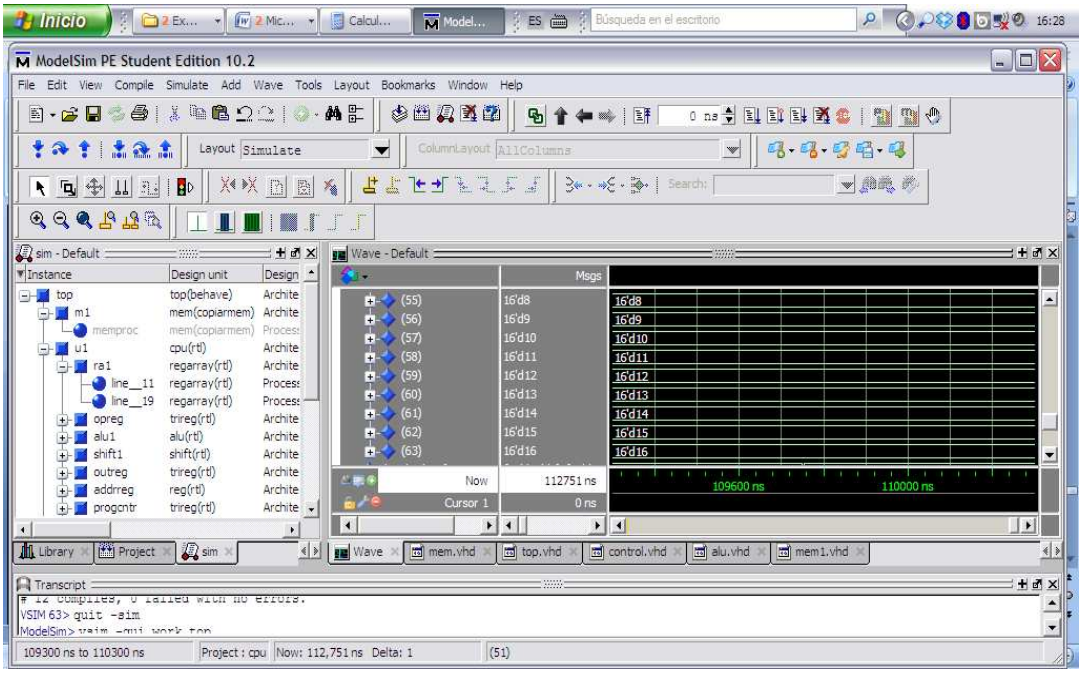


Figura 4.17: Imagen de resultados ejecución subprograma en ModelSim, resultado final, parte 2

Operaciones básicas:

Los resultados que obtenemos al realizar la simulación del subprograma equivalente, que se encuentra en el archivo “mem.vhd” y en la arquitectura

"operbasicas", se muestran en las Figuras de 4.18 a 4.24. Recordamos que realizamos la simulación en decimal para realizar la comparación con los resultados obtenidos mediante PowerDEVS (datos decimales), los resultados de la simulación son los que aparecen en el registro nº 2, tal y como podemos apreciar en la Figura 4.18.

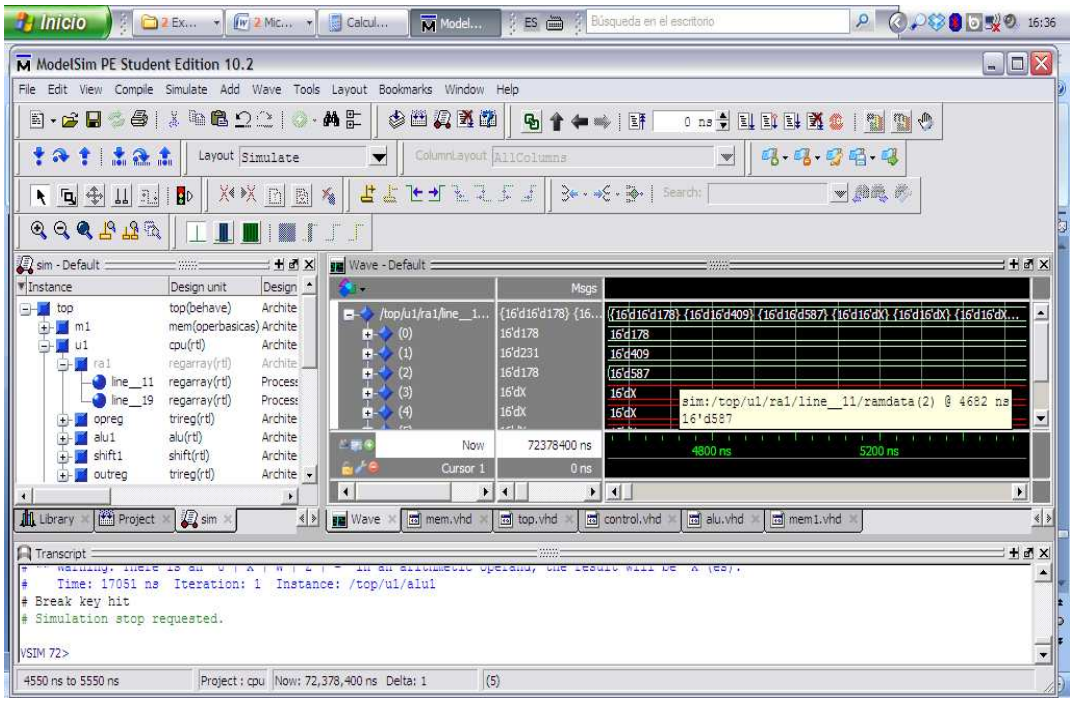


Figura 4.18: Imagen de resultados ejecución subprograma en ModelSim

El resultado, en este caso de la suma, corresponde con el valor 587, que es el valor que observamos en el registro numerado como "2" y en la etiqueta blanca.

El resultado obtenido para la operación "AND" es el valor 144, tal y como se muestra en la Figura 4.19 (registro nº 2).

La siguiente simulación, corresponde con la operación "OR", que arroja el siguiente resultado que mostramos a Figura 4.20.

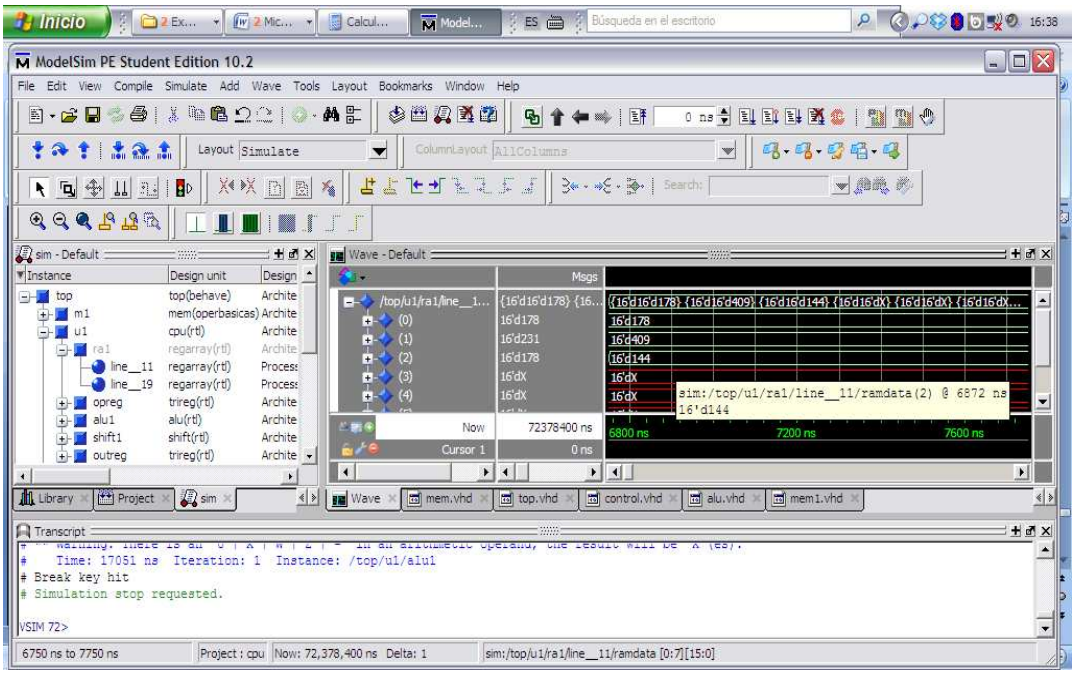


Figura 4.19: Imagen de resultados (2) ejecución subprograma en ModelSim

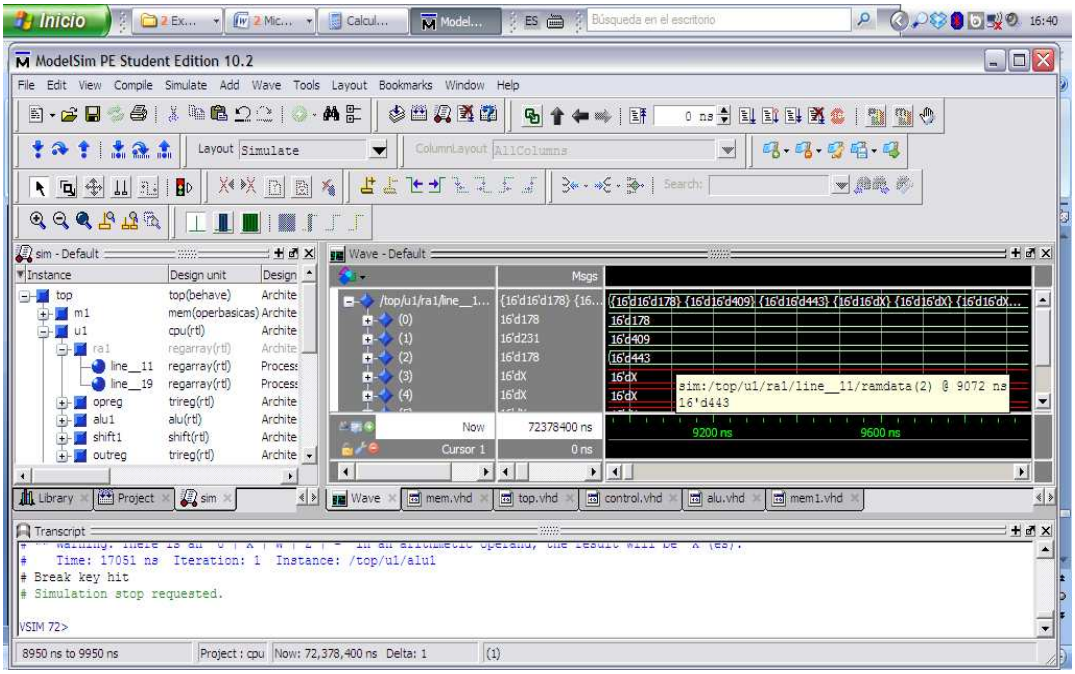


Figura 4.20: Imagen de resultados (3) ejecución subprograma en ModelSim

El valor que obtenemos es el de 443 (registro nº 2 y en la etiqueta blanca).

El siguiente resultado, corresponde con la operación “XOR”, que arroja el resultado mostrado en la Figura 4.21.

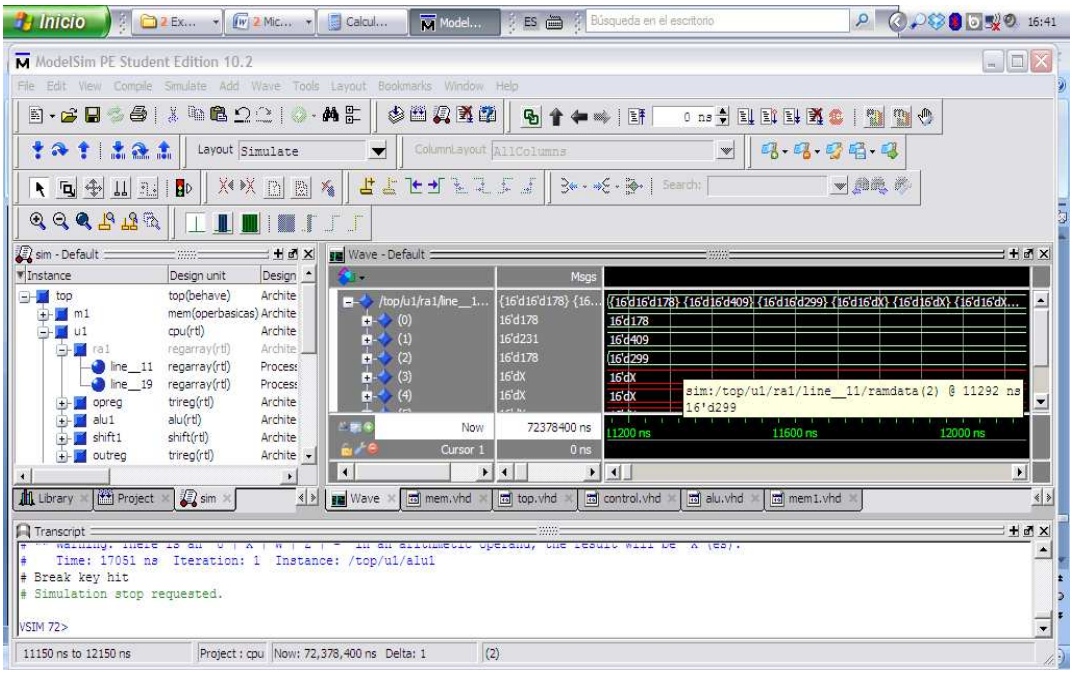


Figura 4.21: Imagen de resultados (4) ejecución subprograma en ModelSim

El resultado obtenido es el valor 299 (registro nº 2 y en la etiqueta blanca). La siguiente operación que vamos a realizar es “shifleft”, que arroja el resultado que podemos apreciar en la Figura 4.22.

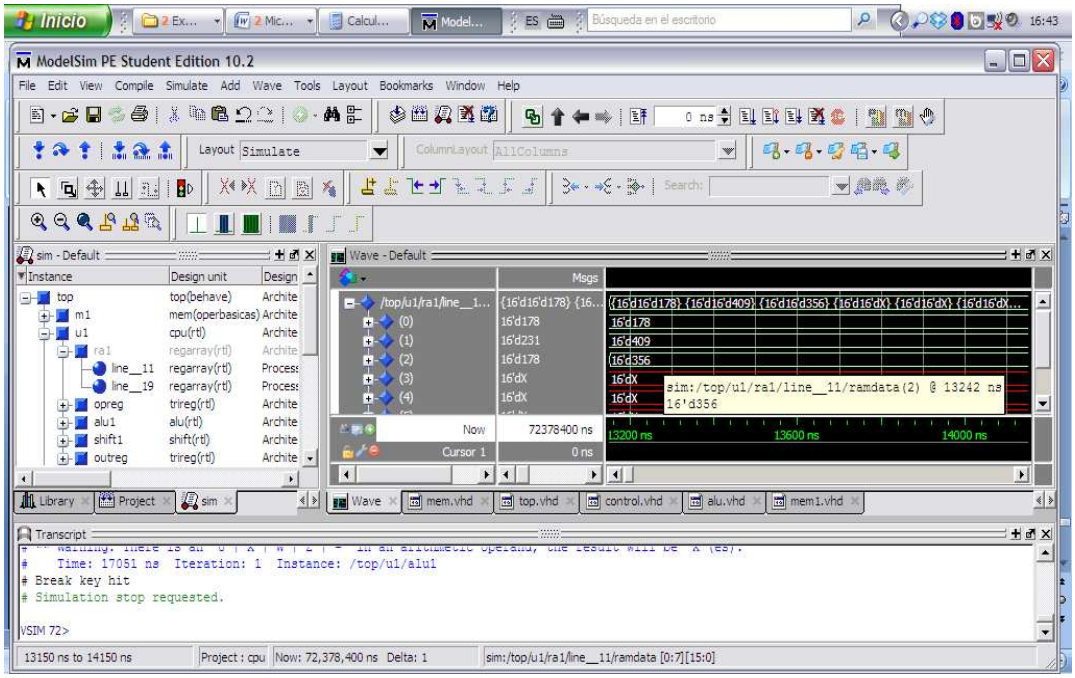


Figura 4.22: Imagen de resultados (5) ejecución subprograma en ModelSim

El resultado que obtenemos es el valor 356 (registro nº 2 y en la etiqueta blanca), que concuerda con la ejecución en una calculadora con los datos de inicio que presentamos.

El siguiente resultado se refiere a la operación “SHR”, que arroja el resultado que podemos apreciar en la Figura 4.23 en el registro nº 2, o en la etiqueta blanca.

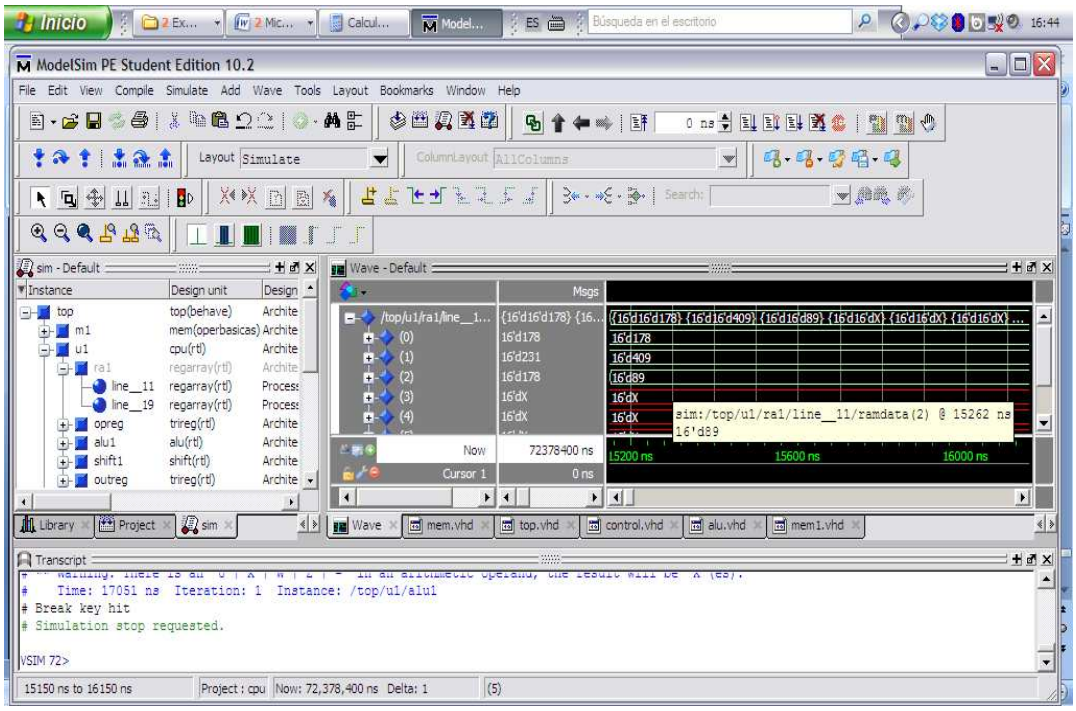


Figura 4.23: Imagen de resultados (6) ejecución subprograma en ModelSim

El valor obtenido corresponde con el valor 89. Que es el que obtenemos de realizar la operación con una calculadora.

Por último presentamos el resultado obtenido en la operación “resta”, que arroja el resultado que podemos apreciar en la Figura 4.22 (registro nº 1 y en la etiqueta blanca).

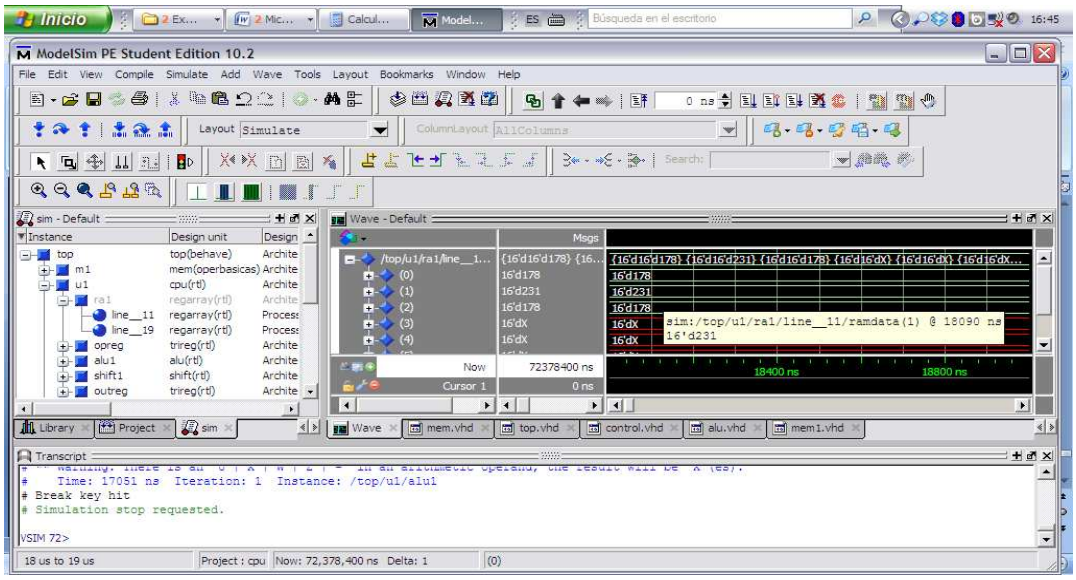


Figura 4.24: Imagen de resultados (7) ejecución subprograma en ModelSim

El valor obtenido, corresponde con el valor 231 (registro nº 1 en este caso). Consistente con el valor de la resta realizada con calculadora.

4.8 CONCLUSIONES

A la hora de analizar los resultados de la simulación, se impone explicar el método usado para la comparación de resultados. El análisis de los resultados así como la corrección de los mismos, se han realizado por comparación de resultados obtenidos, esta comparación de resultados nos servirá de guía de la corrección del código ejecutado, pudiendo por tanto establecer un análisis de los mismos.

Dado que se parte en cada caso de prueba de un código de alto nivel será éste por tanto, el que nos dará una parte de la comparación de resultados. La otra parte de la comparación, la obtenemos de la ejecución del código objeto que hemos desarrollado.

Lo que pretendemos establecer con la documentación presentada, es demostrar la corrección de la traslación del código de alto nivel, con el código máquina obtenido. En un programa comercial este trabajo lo realizaría en automático un procesador de lenguaje, denominado comúnmente como compilador. Con esto se garantizaría, si el mismo es de suficiente calidad, que el código fuente es correcto y a su vez realizaría una traslación optimizada a código objeto, que es el que ejecutaríamos en nuestro programa.

En nuestro caso no se ha contemplado la construcción de dicho compilador, lo cual hubiera resultado muy laborioso y por tanto, la corrección de la traslación código fuente a código objeto la establecemos nosotros basándonos en el conocimiento del funcionamiento de un compilador.

Este hecho condiciona de una manera importante los casos de prueba que hemos desarrollado, pues ha impuesto de facto, que los bancos de prueba sean ejemplos sencillos y poco extensos. Casos de prueba muy extensos y enrevesados, serían muy susceptibles de contener errores, que deberíamos detectar nosotros al no poder hacerlo en automático, Lo cual no es tarea fácil, lo cual pondría en entredicho, la corrección de la correspondencia del código fuente, a código objeto.

Entendemos por tanto que los bancos de prueba seleccionados, dada su sencillez, y tras un análisis de los mismos, se cumple la corrección de la traslación del código fuente al código objeto. No resulta difícil para una persona con conocimientos informáticos, comprobar la corrección de ambos código.

Una vez establecido que dichos códigos son equivalentes, debemos concluir que ambos códigos deben arrojar los mismos resultados, y será la comprobación

de que esto es así, la confirmación de que no es incorrecto el supuesto que hemos establecido.

Los resultados obtenidos de la simulación a mano del código fuente son iguales y consistentes con los resultados obtenidos de la simulación en ModelSim del código objeto implementado en cada caso de prueba. Por tanto entendemos que los datos obtenidos son correctos y están dentro de lo que cabría esperar de los datos de partida.

El código completo de “mem.vhd” (donde están implementados los 4 bancos de prueba), está disponible en el Anexo A. También está disponible el archivo completo, en la carpeta “VHDL” del CD del proyecto.

5

MODELADO MEDIANTE POWERDEVS

5.1 INTRODUCCIÓN

En este capítulo, trataremos de explicar, cómo hemos implementado el circuito propuesto, en PowerDEVS. Para ello comenzaremos por realizar una explicación general del circuito, donde se indicaran los elementos y sus conexiones. A continuación realizaremos una descripción de la implementación PowerDEVS, de un elemento ejemplo, para a continuación describir cada uno de los elementos que hemos implementado. Cabe destacar que la implementación PowerDEVS de los elementos aparece especificada en un archivo de texto plano. Con lo que entendemos, que la explicación de cómo se traduce la implementación de un elemento desde la interfaz, a como se traduce a código fuente nos parece de vital importancia, para la comprensión de las implementaciones de los distintos elementos que componen el PFC.

Cada elemento que hemos implementado, se muestra su código en cuadros de texto, acompañada de una somera explicación de sus funciones. Para aquellos elementos de código, en que se especifica, que es un fragmento de código, debemos indicar que el código completo se muestra en estos casos en el Anexo B.

En la explicación de cada elemento comenzaremos por los elementos más sencillos y por tanto seremos más rigurosos en su explicación, el objetivo que se persigue, es que cuando se llegue a los elementos más complejos gran parte de los pequeños detalles hayan quedado ya explicados (muchos se repiten elemento a elemento), pudiendo por tanto, centrarnos en detalles novedosos o importantes.

Por tanto comenzaremos por presentar el circuito general, recordamos que ya se ha explicado su funcionamiento en el Capítulo 3, para posteriormente pasar al detalle de los elementos que componen el circuito.

5.2 CIRCUITO GENERAL

Antes de pasar a explicar el circuito general, debemos explicar un poco cuál es el circuito base del cual partimos, que mostramos en, Figura 5.1.

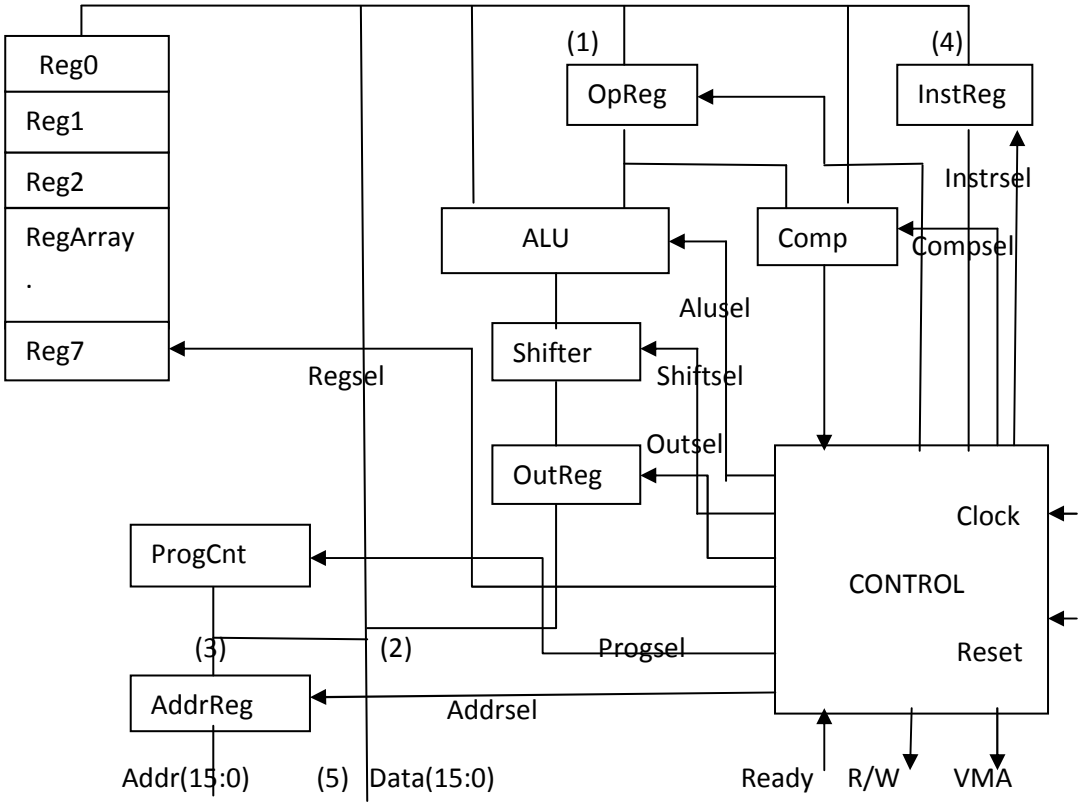


Figura 5.1: Esquema de conexión del circuito "CPU"

Este es el circuito objetivo, es decir el circuito que debemos implementar al cual solo se le debe añadir una memoria, para que funcione dentro de los parámetros requeridos por el PFC.

A continuación en la Figura 5.2, presentamos el circuito con la “CPU” conectada a la “Memoria” que constituye el circuito base del PFC.

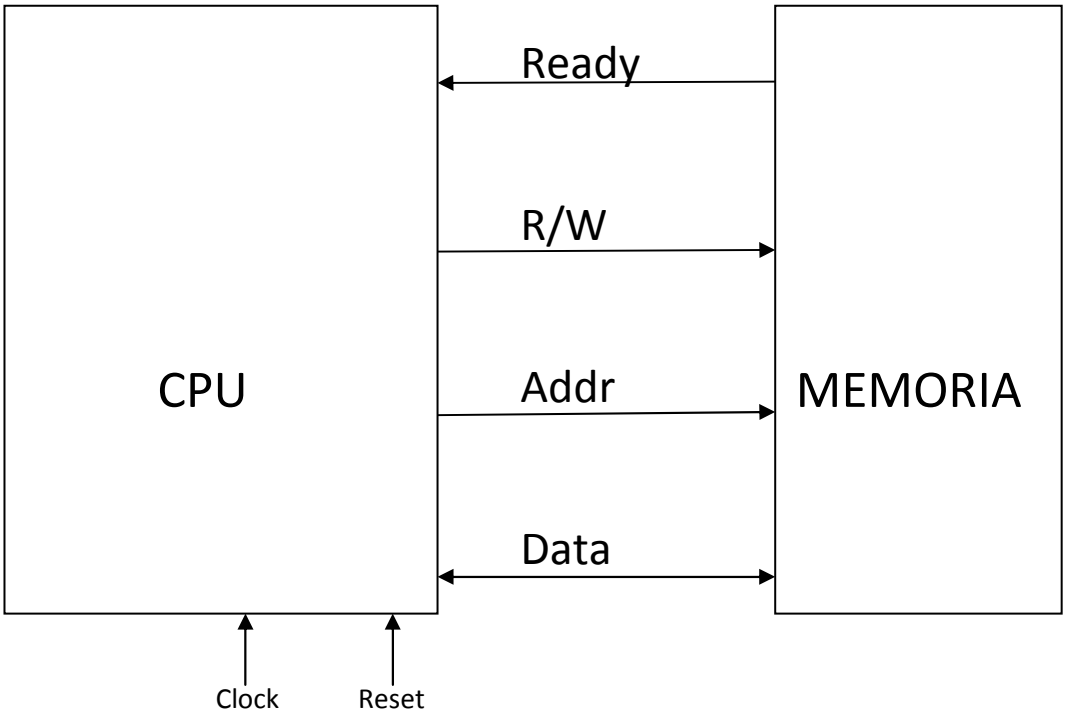


Figura 5.2: Esquema de conexión del circuito “CPU” Conectado a una “memoria”

La traslación a la implementación del circuito mediante PowerDEVS, no resulta inmediata, por las características del formalismo DEVS. Éste contiene algunas restricciones, que necesariamente modifican el esquema de conexión, para poder cumplir con dichas restricciones.

Estas restricciones hacen necesario la aplicación de nuevos elementos, que no aparecen en el diseño inicial (definido ya mediante VHDL). La principal

restricción que afecta al circuito y a la cual nos referimos, es que en un modelo DEVS, la salida de un elemento no puede ser entrada del mismo elemento (realimentación). Esto nos ha obligado a redefinir el circuito base, implementando en aquellos casos, en los cuales ha sido necesario, de un registro intermedio, que actuará como separación de la entrada y de la salida. Entendemos que esto no afecta al funcionamiento general del circuito, si bien supone una pequeña variación en la implementación del mismo.

Presentamos el circuito modificado que corresponde con la implementación DEVS y PowerDEVS del circuito inicial.

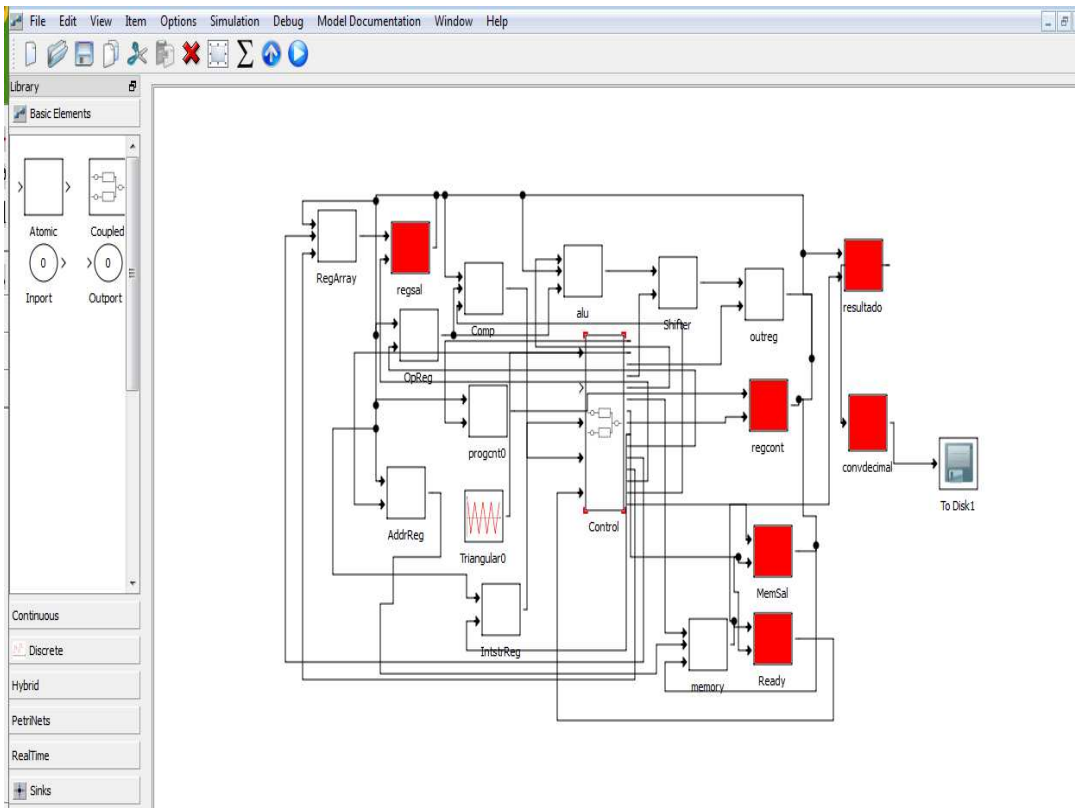


Figura 5.3: Esquema de conexión del circuito “CPU” Conectado a una “memoria”, implementación PowerDEVS

Todos los elementos que aparecen marcados en rojo en la Figura 5.3, no están en el esquema original y sí en el esquema que hemos implementado. Si bien

su aparición en este esquema, se deben a causas bien distintas que pasamos a explicar.

El caso de los elementos “regsal”, ”regcont”, “Memsal” y “Ready” nos hemos visto obligados a su implementación en el proyecto, pues se producía un error en la compilación del programa. Los elementos a los que están conectados tenían a su vez entrada y salida al bus con lo cual el programa interpretaba una realimentación de señal, por tanto abortaba la compilación del mismo.

En consecuencia, para evitar esa realimentación directa de la señal de salida al bus y del bus a la entrada del mismo elemento, hemos intercalado un registro, cuya única misión es proveer de una separación de señales de entrada y salida de los elementos (obligatoria según la definición DEVS) a los que están conectados. En este caso son “Regarray”, “progcont0”, “memory”. Este hecho aunque complica un tanto las señales (añade más señales al sistema), no nos parece de especial relevancia, pues no afecta al funcionamiento básico del mismo y su implementación está plenamente justificada.

Otro caso aparte, son los elementos marcados con el nombre “resultado” y “convdecimal”. Estos elementos se han añadido para comodidad en la presentación de resultados, en la salida por archivo de los mismos (se aprecia que están conectados al elemento que imprime en el archivo out.exe, que es el señalado como “To Disk1”). Básicamente su función es la siguiente; el elemento “resultado” es un registro que se encarga de extraer y seleccionar del bus de datos un dato que previamente habremos seleccionado con la instrucción "print" (no implementada en la especificación inicial). Este dato en binario se lo pasa a

"convdecimal" que lo convierte a decimal y se lo entrega a "To Disk" para que lo imprima en el archivo "out.exe". Esto sucederá, con todos aquellos datos que deseemos, aparezcan como resultado en el archivo de salida.

Esta pequeña modificación, se ha añadido simplemente por comodidad en la presentación del resultado. No se ha implementado en VHDL, simplemente por la facilidad que presenta el entorno ModelSim, para seguir paso a paso todas las señales, cosa que no sucede en PowerDEVS.

Si bien, ambos circuitos parecen distintos (según las imágenes que hemos presentados de ellos). Un seguimiento detallado de los mismos demostrara que son circuitos eléctricamente equivalentes (con las salvedades que hemos explicado), cosa que dejamos al lector.

5.3 IMPLEMENTACIÓN DE ELEMENTOS EN POWERDEVS

Pasamos a detallar elemento a elemento, comenzando primero por una explicación de qué se espera que haga el elemento. Explicando a continuación la implementación PowerDEVS del mismo, mostrando y explicando el código desarrollado y la función que cumple. También nos parece importante recordar, que comenzaremos con los elementos más sencillos, realizando una explicación más detallada de los mismos para ir conforme se avance en las descripciones siendo cada vez menos exhaustivos centrándonos básicamente en los aspectos más importantes y novedosos.

Nos parece lógico, realizar una pequeña explicación de cómo se va a presentar el código, y su correlación con lo que se podrá ver en la interface de PowerDEVS.

Presentar el código mediante imágenes de capturas de pantalla, donde se nos muestra la interfaz, nos parece altamente ineficiente pues ocupa mucho espacio y puede llevar a confusión, debido al alto número de imágenes, que serían necesarias, para desarrollar una explicación que resultase medianamente clara.

Por tanto, entendemos que lo correcto, es mostrar en cuadros de texto plano el código y explicarlo mediante etiquetas. Entendemos que PowerDEVS es un entorno con una buena interfaz gráfica, y de hecho el código se desarrolla mediante dicha interfaz, por tanto debemos establecer la relación que existe entre el código en texto plano y la interfaz para que quede claro lo que se pretende explicar.

Recurriremos a un primer ejemplo, en el cual solo incluiremos comentarios para explicar dicha correspondencia. Con este ejemplo, solo pretendemos establecer la relación entre lo que vemos en la interfaz y el archivo en texto plano, que será el código fuente del elemento. En la Figura 5.4 mostramos la interfaz para implementación de nuestro ejemplo al que hemos denominado “BORRAR”.

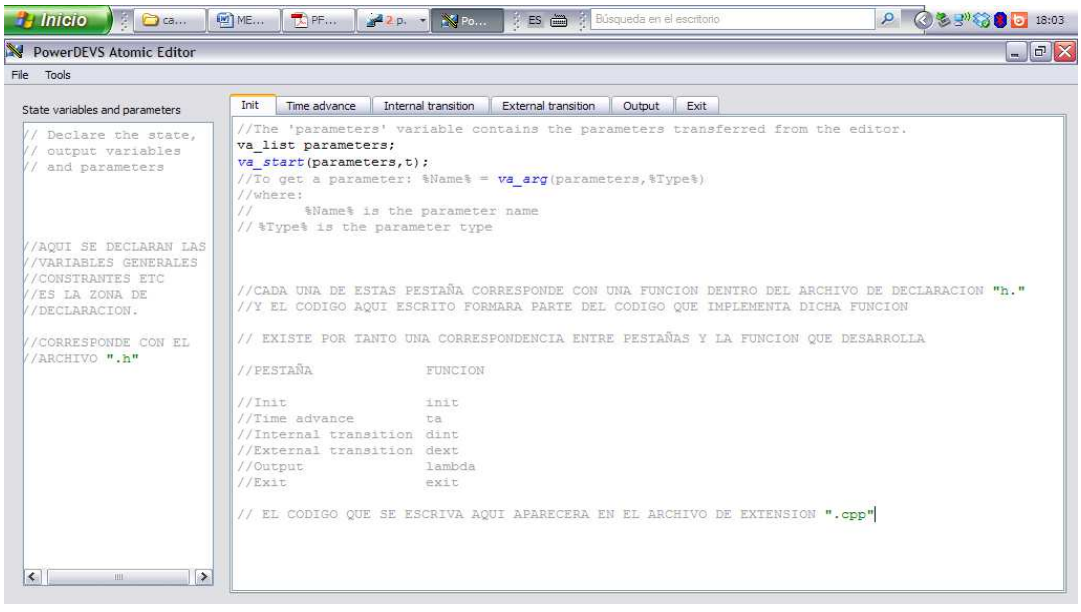


Figura 5.4: Interfaz ejemplo de “BORRAR”

Recordemos, que en la interfaz no hemos declarado código, solo hemos escrito comentarios. El código que aparezca en los archivos que se muestran en Código 5.1 y Código 5.2, ha sido creado de manera automática por el entorno (esto entendemos que facilitara nuestra explicación) y por tanto aparecerá en todos los archivos que se creen mediante el entorno.

```
//CPP:vector/BORRAR.cpp
#if !defined BORRAR_h
#define BORRAR_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
class BORRAR: public Simulator {
// Declare the state,
// output variables
// and parameters

//AQUI SE DECLARAN LAS
//VARIABLES GENERALES
//CONSTRANTES ETC
//ES LA ZONA DE
//DECLARACION.

//CORRESPONDE CON EL
//ARCHIVO ".h"
public:
        BORRAR(const char *n): Simulator(n) {};
        void init(double, ...);// (1)
        double ta(double t); // (1)
        void dint(double); // (1)
        void dext(Event , double ); // (1)
        Event lambda(double); // (1)
        void Exit();// (1)
};
```

Código 5.1: Código “BORRAR.h”

En Código 5.1 vemos resaltado en azul los comentarios que aparecen en Figura 5.4 (parte izquierda), que han sido escritos por nosotros, junto con comentarios que realiza el entorno automáticamente. Por tanto, todo lo que aparece en negro en ese Código 5.1 es declarado automáticamente por el entorno.

En las explicaciones que se sucederán posteriormente, en la descripción de los elementos, veremos que en el archivo de extensión “.h”, aparecerán todas las declaraciones de variables constantes y parámetros que realicemos y que aparecerán en la parte donde están implementados nuestros comentarios (en

azul). Resaltar que en Código 5.1 (1), se declaran las funciones que corresponden con cada pestaña que se muestra en la interfaz.

```
#include "BORRAR.h"
void BORRAR::init(double t,...) {  // (1)
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//      %Type% is the parameter type

//CADA UNA DE ESTAS PESTAÑA CORRESPONDE CON UNA FUNCION DENTRO DEL ARCHIVO DE
DECLARACION "h."
//Y EL CODIGO AQUI ESCRITO FORMARA PARTE DEL CODIGO QUE IMPLEMENTA DICHA FUNCION

// EXISTE POR TANTO UNA CORRESPONDENCIA ENTRE PESTAÑAS Y LA FUNCION QUE
DESARROLLA

//PESTAÑA                                           FUNCION

//Init                                              init
//Time advance                                      ta
//Internal transition                               dint
//External transition                               dext
//Output                                            lambda
//Exit                                              Exit

// EL CODIGO QUE SE ESCRIVA AQUI APARECERA EN EL ARCHIVO DE EXTENSION ".cpp"
}
double BORRAR::ta(double t) {  // (2)
//This function returns a double.

}
void BORRAR::dint(double t) {  // (3)

}
void BORRAR::dext(Event x, double t) {  // (4)
//The input event is in the 'x' variable.
//where:
//      'x.vALUe' is the vALUe (pointer to void)
//      'x.port' is the port number
//      'e' is the time elapsed since last transition

}
Event BORRAR::lambda(double t) {  // (5)
//This function returns an Event:
//      Event(%&VALUe%, %NroPort%)
//where:
//      %&VALUe% points to the variable which contains the vALUe.
//      %NroPort% is the port number (from 0 to n-1)

}
void BORRAR::Exit() {  // (6)
//Code executed at the end of the simulation. }
```

Código 5.2: Código “BORRAR.cpp”

En Código 5.2, podemos ver el código que aparece en el archivo de implementación “BORRAR.cpp”. Resaltado en azul está todo lo que hemos escrito nosotros en la interfaz que en este caso son solo comentarios.

Explicar que en (1) función “init”, aparecerá el código que implementemos en la primera pestaña nombrada como “Init” en la Figura 5.4. En (2) la función “ta”, tendremos el código que hayamos escrito en la segunda pestaña nombrada como “Time advance”. En la función “dint” (3), aparecerá el código que hayamos escrito en la tercera pestaña nombrada como “Internal transition”. En la función “dext” (4), tendremos el código que hayamos escrito en la cuarta pestaña nombrada como “External transition”. En la función “lambda” (5), tendremos el código que implementemos en la quinta pestaña nombrada como “Output”. En la función “Exit” (6), tendremos el código que se implemente en la sexta pestaña, nombrada como “Exit”. Debemos indicar que en nuestro proyecto no hemos escrito código en dicha pestaña.

Por tanto, éste será el patrón que seguiremos en este capítulo, para realizar una explicación del funcionamiento del código implementado en cada elemento. Debemos recordar, que existirán elementos, sobre todo los registros, que aunque sean distintos los elementos, tendrán implementado el mismo código es decir estos elementos compartirán el código. Lo cual se refleja en que existen muchos más elementos que código explicado.

Explicar, que en aquellos recuadros de código, en que se especifica que es un fragmento, el código completo de dicho archivo se podrá consultar en el Anexo B. Por tanto el código que aparezca en un recuadro, donde se indique que es “Código”, corresponderá por tanto al código completo del archivo.

5.4 IMPLEMENTACIÓN DE LOS ELEMENTOS PFC EN POWERDEVS

5.4.1 Reg: El funcionamiento requerido de este circuito, consiste en almacenar en una variable interna la última entrada que le haya llegado, cuando así se lo indique el elemento "control". Entregando el dato almacenado, cuando se le requiera por parte de "control".

Para realizar estas funciones, el elemento dispone de dos estradas, una destinada a datos (puerto 0), por donde recibirá valores de entrada (tipo vector16) y otro puerto, por donde recibirá señales del elemento "control", de tipo double. Dispone a su vez de una salida (puerto 0) por donde entregara los datos que le son requeridos por control, mediante un vector de salida que denominaremos "y" en todos los casos.

Explicitamos (solo en este caso y para que sirva de ejemplo) mediante la Figura 5.5, donde se aprecia en el margen superior unas pestañas cuyos nombres corresponden a funciones del formalismo DEVS. En su traslación a PowerDEVS se corresponden con funciones en código C++ (explicadas con anterioridad), podemos apreciar que la pestaña "Init" de la Figura 5.4, corresponde con la declaración de la función "init" en el Código 5.3, (1).

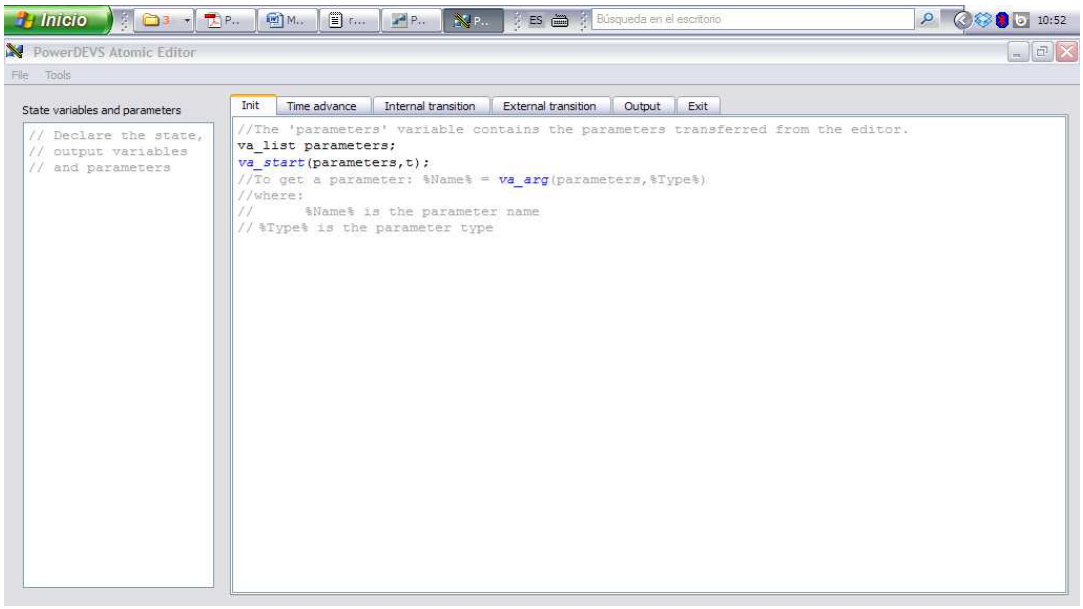


Figura 5.5: Pantalla para la creación de código de elementos nuevos

Lo mismo sucede en Código 5.3, con las pestañas “Time advance” “ta” (2), “Internal Transition” “dint”(3), “External Transition” “dext”(4), “Output” “lambda”(5), “Exit” “Exit”(6).

```
//CPP:vector/reg.cpp
#if !defined reg_h
#define reg_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector.h"
#include "vector/vector16.h"
class reg: public Simulator {
// Declare the state,
// output variables
// and parameters
vector16 vec,vec1;  --(7)
double Sigma;  --(8)
double y[16];  --(9)
#define INF 1e20  --(10)
public:
        reg(const char *n): Simulator(n) {};
        void init(double, ...);  --(1)
        double ta(double t);  --(2)
        void dint(double);  --(3)
        void dext(Event , double );  --(4)
        Event lambda(double);  --(5)
        void Exit();  --(6)
};
#endif
```

Código 5.3: archivo de definición elemento “reg.h”

Presentamos en la Código 5.3 el archivo de definición (extensión “.h”), en el que se nos muestra la definición de las variables generales del elemento y de sus

funciones. Destacaremos por ser de utilidad para el resto de elemento que en todos los archivos de definición, se crea en automático, al definir un nuevo elemento de código, ya aparecen declaradas las funciones que aparecen en la interface de los elementos.

La declaración de las variables y constantes del elemento se nos muestra en Código 5.3 (7) a (10). Resaltar que en (7), vemos que se declara una variable de tipo “vector16” este tipo de datos ha sido creado por nosotros para que las instrucciones sean de este tipo, que es un vector de 16 posiciones, en (8) se aprecia la declaración de un dato de tipo doble y en (9) se aprecia la declaración de un vector de 16 posiciones de tipo doble (que utilizaremos en la salida). En (10) se declara una constante que simula un valor infinito mediante su declaración a un valor muy elevado.

En el archivo de implementación que mostramos en la Código 5.4, vemos cómo se almacena el último dato entrado en el puerto 0, en la variable en este caso llamada “vec” (1), si recibe un 0 por el puerto 1 (2), el valor recibido por el puerto 0, se almacenará en la variable de salida “y” y estará dispuesto para ser entregado en la salida (3). Lo cual ocurrirá cuando se reciba un 1 por el puerto 1 pues como se aprecia en (4), Sigma pasa a valer 0 y se producirá una transición interna y se sacará por la salida (puerto 0 en este caso), el último valor almacenado, que estará en la variable de salida “y”.

```
#include "reg.h"
void reg::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
}
double reg::ta(double t) {
//This function returns a double.
return Sigma; }
void reg::dint(double t) {
Sigma=INF;}
void reg::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
if (x.port == 0){
vec =*(vector16*) x.vALUe;}  --(1)
if (x.port ==1){
vec1 =*(vector16*) x.vALUe;
if (vec1.vALUe[0]==0){  --(2)
for (int i=0;i<16;i++) {
y[i]=vec.vALUe[i];  --(3)
}}
if (vec1.vALUe[0]==1){
Sigma=0;}  --(4)
}
}
Event reg::lambda(double t) {
return Event(y,0)};  --(3)
void reg::Exit() {
//Code executed at the end of the simulation.
}
```

Código 5.4: Código “reg.cpp”

5.4.2 Reg_a: El caso de este tipo de registro es un tanto diferente al anterior, pues solo almacena el dato último que ha llegado. Cada vez que le llega un dato al puerto 0, éste se almacena en una variable, borrando el dato existente anteriormente. Este dato solo se presentará en la salida del registro cuando reciba una señal de control a través del puerto 1, con lo que presentara en la salida el dato último que tenga almacenado.

Por definirlo de una manera, este registro no selecciona el dato a almacenar sino que selecciona, cuando se muestra un dato en la salida.

```
//CPP:vector/reg_a.cpp
#if !defined reg_a_h
#define reg_a_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"
class reg_a: public Simulator {
// Declare the state,
// output variables
// and parameters
//out
vector16 vec,vec1;
double y[16];
double Sigma;
#define INF 1e20
public:
        cntprog(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

Código 5.5: Código “reg_a.h”

En Código 5.5 se nos muestra la definición de variables, que básicamente se repetirá en todos los elementos registro, así como de la contante INF, que ya hemos explicado anteriormente, simula un valor infinito mediante una declaración a un valor muy alto. Por otra parte las variables “vec” o “vec1”, se declaran de tipo “vector16”, que como hemos dicho anteriormente es un tipo de datos declarado por nosotros.

En Código 5.6 se nos muestra el código de “reg_a.cpp”. Como hemos comentado anteriormente, vemos que los valores de entrada se reciben en el puerto 0 y se almacenan en la variable vec (1). Cada vez que llegue un nuevo valor al puerto 0, éste se almacenará en dicha variable sobrescribiendo el anterior valor. Cuando recibimos una señal por el puerto 1 y solo si el valor es 1, entonces el valor almacenado en “vec”, se entregará en la salida (2) al realizarse de inmediato una transición interna, puesto que el valor de sigma pasa a valer 0 (3).

```
#include "reg_a.h"
void reg_a::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
for(int i=0;i<16;i++){
y[i]=10;
}
}
double reg_a::ta(double t) {
//This function returns a double.
return Sigma;
}
void reg_a::dint(double t) {
Sigma=INF;
}
void reg_a::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
if (x.port==0){
vec=*(vector16*)x.value;}  //(1)
if (x.port==1){
vec1=*(vector16*)x.value;
if (vec1.value[0]==1){
for (int i=0;i<16;i++){
y[i]=vec.value[i];  //(2)
}
sigma=0;}}  //(3)
}
event reg_a::lambda(double t) {
//this function returns an event:
//     event(%&value%, %nroport%)
//where:
//     %&value% points to the variable which contains the vALUe.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(&y,0);  //(2)
}
void reg_a::Exit() {
//Code executed at the end of the simulation.
}
```

Código 5.6: Código “reg_a.cpp”

5.4.3 ProgCnt: El contador de programa tiene como misión, almacenar el número de instrucción que se está ejecutando, por tanto deberá ir variando conforme avance la ejecución del programa.

Para realizar su cometido, el elemento dispone de dos entradas y una salida. En la entrada 0, se recibe el valor que deberá ser almacenado; por su salida se nos mostrará el valor almacenado con anterioridad y por el puerto 1, recibirá la señal de control, que nos indica, si debemos almacenar un valor determinado o bien entregarlo por la salida.

```
//CPP:vector/cntprog.cpp
#if !defined cntprog_h
#define cntprog_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"

class cntprog: public Simulator {
// Declare the state,
// output variables
// and parameters

vector16 vec1,vec2;
//out
double y[16];
double Sigma;

#define INF 1e20
public:
        cntprog(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

Código 5.7: Código “cntprog.h”

Entendemos, que la declaración que se nos muestra en Código 5.7, no merece más explicación, pues es igual a la declaración del archivo “reg.h” en Código 5.3.

El código del archivo “contprog.ccp” que mostramos en Código 5.8 es igual al de Código 5.4, entendemos por tanto que no merece más explicación, pues remitimos al lector a dicha explicación. De hecho podríamos usar uno u otro indistintamente para el contador de programa, pero nos ha parecido más correcto dejar un archivo solo, para el contador de programa.

```
#include "cntprog.h"
void cntprog::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
for (int i=0;i<16;i++){
y[i]=10;}

}
double cntprog::ta(double t) {
//This function returns a double.
return Sigma;
}
void cntprog::dint(double t) {
Sigma=INF;
}
void cntprog::dext(event x, double t) {
//the input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
if (x.port==0){
vec1=*(vector16*) x.value;}

if (x.port==1){
vec2=*(vector16*) x.value;

if (vec2.value[0]==1){
for (int i=0;i<16;i++){
y[i]=vec1.value[i];}
}
if (vec2.value[0]==0){
sigma=0;}}
}
event cntprog::lambda(double t) {
//this function returns an event:
//     event(%&value%, %nroport%)
//where:
//     %&value% points to the variable which contains the value.
//     %nroport% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void cntprog::Exit() {
//Code executed at the end of the simulation.
}
```

Código 5.8: "cntprog.cpp"

5.4.4 Comp: Este elemento, se encarga de realizar las comparaciones entre dos valores, devolviendo en su salida un valor determinado según sea el resultado de la comparación. Decir que este elemento es clave en las bifurcaciones y bucles pues el resultado de la comparación determina la alternativa a tomar.

En Código 5.9 se nos muestra el archivo de declaración del elemento “comp”.

```
//CPP:vector/comp.cpp
#if !defined comp_h
#define comp_h

#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector.h"
#include "vector/vector16.h"
class comp: public Simulator {
// Declare the state,
// output variables
// and parameters
vector16 vec1,vec2,vec3;
int restot,restot1;
double y[16];
double Sigma;
#define INF 1e20
public:
        cntprog(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

Código 5.9: “comp.h”

En cuanto a la implementación en PowerDEVS, en este caso por comodidad nuestra, pues el lenguaje C tiene un tratamiento de bit bastante pobre, hemos optado por convertir los valores de entrada que son binarios puros a decimal. Realizar las comparaciones en decimal, nos resulta más cómodo.

Veremos en Código 5.10 y en la continuación que es Código 5.11, que las comparaciones se producen en valores decimales. La comparación (1) si resulta correcta, con el tipo de comparación que se nos pide a través de “compsel”, “vec3” (2), la salida valdrá 1 y en caso contrario 0.

El valor de tiempo para la siguiente transacción interna pasará a 0 en ambos casos, con lo que la transición se realizara inmediatamente. Presentado el resultado correspondiente en el puerto de salida, en este caso, puerto “0” (3).

```
#include "comp.h"
void comp::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type

}
double comp::ta(double t) {
//This function returns a double.
return Sigma;
}
void comp::dint(double t) {
Sigma=INF;
}
void comp::dext(event x, double t) {
//the input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
int res1,res2,res3,res4,res11,res12,res13,res14,oper,resto;
if (x.port==0){
vec1=*(vector16*)x.value;
for (int i=0;i<16;i++){
if (vec1.value[i]==1){
}else{
vec1.value[i]=0;}
}  //
res1 =
(int)((vec1.value[0]*8)+(vec1.value[1]*4)+(vec1.value[2]*2)+(vec1.value[3]));
res2 =
(int)((vec1.value[4]*8)+(vec1.value[5]*4)+(vec1.value[6]*2)+(vec1.value[7]));
res3 =
(int)((vec1.value[8]*8)+(vec1.value[9]*4)+(vec1.value[10]*2)+(vec1.value[11]));
res4 =
(int)((vec1.value[12]*8)+(vec1.value[13]*4)+(vec1.value[14]*2)+(vec1.value[15]));
restot = res1*1000+res2*100+res3*10+res4;
}
if (x.port==1){
vec2=*(vector16*)x.value;
for (int i=0;i<16;i++){
if (vec2.value[i]==1){
}else{
vec2.value[i]=0;}
}  //
res11 =
(int)((vec2.value[0]*8)+(vec2.value[1]*4)+(vec2.value[2]*2)+(vec2.value[3]));
res12 =
(int)((vec2.value[4]*8)+(vec2.value[5]*4)+(vec2.value[6]*2)+(vec2.value[7]));
res13 =
(int)((vec2.value[8]*8)+(vec2.value[9]*4)+(vec2.value[10]*2)+(vec2.value[11])
);
res14 =
(int)((vec2.value[12]*8)+(vec2.value[13]*4)+(vec2.value[14]*2)+(vec2.value[15]));
restot1 = res11*1000+res12*100+res13*10+res14;
}
if (x.port==2){

vec3=*(vector16*)x.value;
```

Código 5.10: “comp.cpp” (parte 1)

```
if (vec3.value[0]==0&&vec3.value[1]==0&&vec3.value[2]==1){  //(2)
//neq
if (restot!=restot1){  //(1)
y[0]=1;}else{
y[0]=0;}
sigma=0;}
if (vec3.value[0]==0&&vec3.value[1]==1&&vec3.value[2]==0){  //(2)
//gt
if (restot<restot1){  //(1)
y[0]=1;}else{
y[0]=0;}
sigma=0;}
if (vec3.value[0]==0&&vec3.value[1]==1&&vec3.value[2]==1){  //(2)
//gt or eq
if (restot<=restot1){  //(1)
y[0]=1;}else{
y[0]=0;}
sigma=0;}
if (vec3.value[0]==1&&vec3.value[1]==0&&vec3.value[2]==0){  //(2)
//less
if (restot>restot1){  //(1)
y[0]=1;}else{
y[0]=0;}
sigma=0;}
if (vec3.value[0]==1&&vec3.value[1]==0&&vec3.value[2]==1){  //(2)
//less or eq
if (restot>=restot1){  //(1)
y[0]=1;}else{
y[0]=0;}
sigma=0;}  //(3)
}

}
event comp::lambda(double t) {
//this function returns an event:
//     event(%&value%, %nroport%)
//where:
//     %&value% points to the variable which contains the value.
//     %nroport% is the port number (from 0 to n-1)
return Event(y,0);  //(3)
}
void comp::Exit() {
//Code executed at the end of the simulation.

}
```

Código 5.11: “comp.cpp” (parte 2)

5.4.5 Shift: La función de este elemento es, la de realizar los desplazamientos de bits, tiene por tanto dos entradas y una sola salida donde presentará el resultado de la operación de desplazamiento, que se le indique a través del puerto 1. En la mayoría de ocasiones, será presentar en la salida el mismo valor que recibió en la entrada, puerto 0, esto ocurrirá en los casos que no tenga que realizar el desplazamiento de ningún dato.

El archivo de definición, es igual a los anteriores archivos que describen registros y por tanto entendemos que su explicación se encuentra descrita anteriormente en “reg”. Mostramos en Código 5.12 el código del archivo de definición del elemento “shift”.

```
//CPP:vector/shift1.cpp
#if !defined shift1_h
#define shift1_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"

class shift1: public Simulator {
// Declare the state,
// output variables
// and parameters
vector16 vec1,vec2;
int oper,restot;
double y[16];
double Sigma;
#define INF 1e20
public:
        shift1(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

Código 5.12: Código “shift1.h”

El desplazamiento de bits, se realiza mediante una conversión a decimal del valor binario de la entrada (1) en Código 5.14. Procediendo con posterioridad a realizar un desplazamiento decimal (operación <<) en el caso de desplazamiento a izquierdas (2) o un desplazamiento decimal (operación>>) en el caso del desplazamiento a derechas (3). Una vez realizados los desplazamientos, el valor decimal obtenido, es vuelto a convertir a binario y presentado en la salida. El código de “shifter1” comienza en Código 5.13.

```
#include "shift1.h"
void shift1::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//       %Type% is the parameter type
}
double shift1::ta(double t) {
//This function returns a double.
return Sigma;
}
void shift1::dint(double t) {
Sigma=INF;
}
void shift1::dext(Event x, double t) {
//the input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition

int res1,res2,res3,res4,resto;
if (x.port==0){
vec1=*(vector16*)x.value;
for (int i=0;i<16;i++){
if (vec1.value[i]==1){
}else{
vec1.value[i]=0;}
}
res1 =
((vec1.value[0]*32768)+(vec1.value[1]*16384)+(vec1.value[2]*8192)+(vec1.value[3]*4
096));
res2 =
((vec1.value[4]*2048)+(vec1.value[5]*1024)+(vec1.value[6]*512)+(vec1.value[7]*256)
);
res3 =
((vec1.value[8]*128)+(vec1.value[9]*64)+(vec1.value[10]*32)+(vec1.value[11]*16));
res4 =
((vec1.value[12]*8)+(vec1.value[13]*4)+(vec1.value[14]*2)+(vec1.value[15]));
restot = res1+res2+res3+res4;  // (1)
}
if (x.port==1){
vec2=*(vector16*)x.value;
for (int i=0;i<16;i++){
if (vec2.value[i]==1){
}else{
vec2.value[i]=0;}
}
  if (vec2.value[0]==1&vec2.value[1]==1&vec2.value[2]==1){  // (3)
  for (int i=0;i<16;i++){
  y[i]= vec1.value[i];}
  sigma=0;}
  if (vec2.value[0]==0&&vec2.value[1]==1&&vec2.value[2]==0){
  //desplazamiento a derecha
  oper=restot>>1;
  for (int i=15;i>=0;i--) {
  resto= oper % 2;
  oper=(oper-resto)/2;
  y[i]=resto;
  }
  sigma=0;
  }
```

Código 5.13: Código “shift1.cpp” (parte1)

```
if (vec2.value[0]==0&&vec2.value[1]==0&&vec2.value[2]==1){  // (2)
//desplazamiento a izquierdas
oper=restot<<1;
for (int i=15;i>=0;i--) {
resto= oper % 2;
oper=(oper-resto)/2;
y[i]=resto;
}

sigma=0;
}
}}
Event shift1::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&value% points to the variable which contains the vALUe.
//     %NroPort% is the port number (from 0 to n-1)
Sigma=INF;
return Event(y,0);
}
void shift1::Exit() {
//Code executed at the end of the simulation.
}
```

Código 5.14: Código “shift1.cpp” (parte 2)

5.4.6 Regarray: la función de este elemento, es la de ser una memoria de almacenamiento rápido para uso intermedio entre control y la memoria principal. Se compone por definición, en este caso de 8 registros. En cada uno de los cuales se almacena 1 palabras de 16 bits. Cabe destacar que en la definición del circuito en la Figura 5.1, se puede apreciar como la entrada y la salida de datos es única. Hemos comentado anteriormente, que las restricciones del formalismo DEVS, impiden que eso sea así. Por lo tanto tendremos una entrada de datos y una salida de datos. Además dispondremos de una entrada de control denominada “Regsel” (en el esquema Figura 5.1), por la cual indicaremos el número de registro sobre el que se desea actuar, y otra entrada de control, donde se indicará que operación deseamos realizar (lectura o escritura).

A continuación presentamos en la Figura 5.15 el archivo de definición.

```
//CPP:vector/reagarray1.cpp
#if !defined reagarray1_h
#define reagarray1_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector.h"
#include "vector/vector16.h"


class reagarray1: public Simulator {
// Declare the state,
// output variables
// and parameters
vector16 ent,v1;
vector dir;
int d,i;
double memoria1[16][8];  // (1)
//out
double y[16];
double Sigma;

#define INF 1e20
public:
       reagarray1(const char *n): Simulator(n) {};
       void init(double, ...);
       double ta(double t);
       void dint(double);
       void dext(Event , double );
       Event lambda(double);
       void Exit();
};
#endif
```

Código 5.15: Código “rearray1.h”

Cabe destacar las diferencias básicas, respecto a las definiciones de registros efectuadas con anterioridad.

La existencia de un matriz de 16x8 elementos de tipo doble (1), que actuará como la memoria física de los registros, efectuándose las operaciones de entrada y salida de datos sobre dicha matriz de vectores.

En la Figura 5.16, mostramos el código que implementa el elemento “regarray1”, apreciamos que se inicializa la matriz al valor cero (1).

```
#include "reagarray1.h"
void reagarray1::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
for (int i=0;i<8 ;i++){  // (1)
for (int j=0;j<16;j++){
memoria1[k][i]=0;
}}

double reagarray1::ta(double t) {
//This function returns a double.
return Sigma;

}
void reagarray1::dint(double t) {
Sigma=INF;

}
void reagarray1::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
/* mapa de puertos
port 0 = entrada de datos
port 1 entrada de direcciones
port 2 entrada de lectura almacena el valor del bus
port 3 entrada de escritura saca por el bus el valor de memeoria elegido
*/
vector v2;
if (x.port==0){
ent=*(vector16*)x.value;  // (2)
}
if (x.port==1){
v1=*(vector16*)x.value;
i=(v1.value[0]*4)+(v1.value[1]*2)+(v1.value[2]);  // (3)
}
if (x.port==2){  // (4)
v2=*(vector*)x.value;
if (v2.value[0]==0){
for (int k=0;k<16;k++){   // (5)
memoria1[k][i]=ent.value[k];}
}

if (v2.value[0]==1){   // (6)
for (int k=0;k<16;k++){
y[k]=memoria1[k][i];}
sigma=0;}}
}
Event reagarray1::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the vALUe.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);

}
void reagarray1::Exit() {
//Code executed at the end of the simulation.

}
```

Código 5.16: Código “regarray1.cpp”

Podemos apreciar, que cuando se recibe un dato de entrada, éste se carga en la variable “ent” (2). Cuando tenemos una dirección para guardarlo, que se recibe por el puerto 1, ésta se convierte al valor decimal correspondiente (3) y según el valor que se reciba por el puerto 2 (4), si es cero se guardará el dato existente en la variable en la posición correspondiente de la matriz vector1 (5).

En el caso de la lectura, no existirá dato de entrada, solo las señales de dirección de memoria y la de tipo de operación, en este caso su valor deberá ser 1 con lo que se entregará en la salida el valor solicitado (6).

5.4.7 Convdecimal: Este elemento, no está en la descripción del circuito inicial de la Figura 5.1. Por tanto es un elemento que no está presente en la descripción VHDL del circuito. La implementación de dicho elemento, se justifica por la forma de presentar los datos de salida, que tiene el entorno PowerDEVS.

Recordemos que en ModelSim, podemos ver con toda facilidad el valor que toman las variables, en cada instante de simulación (seleccionándolas previamente para su visualización).

Pero esta funcionalidad no está presente en el entorno PowerDEVS, dicho entorno presenta una librería, con los distintos tipos de salida disponibles. Pero está funcionalidad no se encuentra implementada.

Esto nos ha hecho reflexionar, sobre la manera más adecuada de mostrar los datos de salida. Al final, nos hemos decantado por implementar una nueva instrucción, que hemos denominado “print”. Esta instrucción, nos muestra en la salida, el dato que hemos seleccionado en dicha instrucción convertido a su valor

decimal, a través de “Convdecimal”. Con lo cual resultará más fácil la comparación de resultados, con las ejecuciones que se realicen del banco de pruebas con otros entornos (VHDL).

El circuito en sí, lo que realiza es la conversión a decimal de todos los resultados que le llegan a su entrada, es decir entrega en la salida el valor de entrada convertido a su valor decimal.

```
//CPP:vector/convdecimal.cpp
#if !defined convdecimal_h
#define convdecimal_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"
class convdecimal: public Simulator {
// Declare the state,
// output variables
// and parameters
double y[16];
double Sigma;
#define INF 1e20
public:
       convdecimal(const char *n): Simulator(n) {};
       void init(double, ...);
       double ta(double t);
       void dint(double);
       void dext(Event , double );
       Event lambda(double);
       void Exit();
};
```

Código .5.17: Código “convdecimal.h”

En la Figura 5.17 podemos ver el código del archivo de definición del elemento “Convdecimal”, entendemos que de este código poco hay que comentar pues solo dispone de una entrada y una salida y simplemente su función es la conversión automática a un valor decimal de la señal que se entregue en su entrada tal y como explicaremos en Código 5.18.

```
#include "convdecimal.h"
void convdecimal::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
for(int i=0;i<16;i++){
y[i]=0;}}
double convdecimal::ta(double t) {
//This function returns a double.
return Sigma;
}
void convdecimal::dint(double t) {
Sigma=INF;}
void convdecimal::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
vector16 vec1,vec2,vec3;
double res1,res2,res3,res4,restot;
for (int i=0;i<16;i++){
vec1.value[i]=0;}
if (x.port==0){
vec1=*(vector16*)x.value;
for (int i=0;i<16;i++){
if (vec1.value[i]==1){
}else{
vec1.value[i]=0;}}
res1 =
((vec1.value[0]*32768)+(vec1.value[1]*16384)+(vec1.value[2]*8192)+(vec1.value[3]
*4096));
res2 =
((vec1.value[4]*2048)+(vec1.value[5]*1024)+(vec1.value[6]*512)+(vec1.value[7]*25
6));
res3 =
((vec1.value[8]*128)+(vec1.value[9]*64)+(vec1.value[10]*32)+(vec1.value[11]*16
));
res4 =
((vec1.value[12]*8)+(vec1.value[13]*4)+(vec1.value[14]*2)+(vec1.value[15]));
restot = res1+res2+res3+res4;  // (1)
//y[0]=((vec1.value[0]*32768)+(vec1.value[1]*16384));
//y[0]=vec1.value[1];
y[0]=restot;
sigma=0;  // (2)
}}
Event convdecimal::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the vALUe.
//     %NroPort% is the port number (from 0 to n-1)
Sigma=INF;
return Event(y,0);}
void convdecimal::Exit() {
//Code executed at the end of the simulation.
}
```

Código 5.18: Código “convdecimal.cpp”

Del código presentado en Código 5.18, entendemos que poco se tiene que explicar de dicho código simplemente se hace una conversión automática al valor

decimal(1), valor binario que se recibe en la única entrada, presentando dicho valor en la salida a continuación(2).

5.4.8 Memory: La función que debe realizar este elemento, es la de una memoria principal. Es por tanto, donde se guardaran los programas, que en nuestro caso, serán los bancos de pruebas. En nuestro caso se han implementado cuatro bancos de prueba. Por facilidad y claridad, hemos decidido implementar los mismos en diferentes archivos, en lugar de implementarlos todos en un único archivo. Esto nos hubiera obligado, a establecer un mecanismo de selección, de cuál es el programa que queremos simular.

Por tanto a la hora de simular cada banco de prueba deberemos previamente establecer sobre que archivo se va a realizar la simulación. Este extremo, se explica en el siguiente capítulo, que entendemos es el lugar apropiado para dicha explicación.

Debemos resaltar, que cada uno de los cuatro bancos de prueba es idéntico en el código, con solo la diferencia, del código máquina establecido en la matriz de memoria. Pues cada uno, corresponde con un banco de pruebas. Por tanto pasamos a detallar el código de uno de los banco de pruebas, resaltando que el resto de los bancos de pruebas, son iguales exceptuando el código máquina de la matriz de memoria.

```
//CPP:vector/memori4.cpp
#if !defined memori4_h
#define memori4_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"

class memori4: public Simulator {
// Declare the state,
// output variables
// and parameters

vector16 vec,vec1,vec2;
double memoria [64][16];  // (1)
double Sigma;
int adress,i;
double y[16];
#define INF 1e20


public:
       memori4(const char *n): Simulator(n) {};
       void init(double, ...);
       double ta(double t);
       void dint(double);
       void dext(Event , double );
       Event lambda(double);
       void Exit();
};
#endif
```

Codigo5.19: Código “memori4.h”

```
#include "memori4.h"
void memori4::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
for (int j=0;j<64;j++){
for (int i=0;i<16;i++){
memoria[j][i]=0;  // (2)
}}
// load en reg 0
memoria[0][2]=1;
// a=2
memoria[1][14]=1;
// load en reg 1
memoria[2][2]=1;
memoria[2][15]=1;
// b=15
memoria[3][12]=1;
memoria[3][13]=1;
memoria[3][14]=1;
memoria[3][15]=1;
// load en reg 2
memoria[4][2]=1;
memoria[4][14]=1;
// c=7
memoria[5][13]=1;
memoria[5][14]=1;
memoria[5][15]=1;
//load en reg 3
memoria[6][2]=1;
memoria[6][14]=1;
memoria[6][15]=1;
```

Código 5.20: Código “memori4.cpp” (parte1)

```
// d=10
memoria[7][12]=1;
memoria[7][14]=1;
// BranchI c d
memoria[8][2]=1;
memoria[8][3]=1;
memoria[8][11]=1;
memoria[8][14]=1;
memoria[8][15]=1;
//14
memoria[9][12]=1;
memoria[9][13]=1;
memoria[9][14]=1;
//Sub b-a
memoria[10][1]=1;
memoria[10][2]=1;
memoria[10][3]=1;
memoria[10][15]=1;
//inc d
memoria[11][2]=1;
memoria[11][3]=1;
memoria[11][4]=1;
memoria[11][14]=1;
//BranchI
memoria[12][2]=1;
memoria[12][4]=1;
// 8
memoria[13][12]=1;
memoria[14][0]=1;
memoria[14][1]=1;
memoria[14][2]=1;
memoria[15][0]=1;
memoria[15][1]=1;
memoria[15][2]=1;
memoria[15][15]=1;
memoria[16][0]=1;
memoria[16][1]=1;
memoria[16][2]=1;
memoria[16][14]=1;
//BRACNH NOT EQUAL
memoria[17][0]=1;
memoria[17][3]=1;
memoria[17][4]=1;
memoria[17][12]=1;
memoria[17][15]=1;
//DIRECCION DEL SALTO
memoria[18][11]=1;
memoria[18][13]=1;
memoria[18][14]=1;
//PRINT
memoria[19][0]=1;
memoria[19][1]=1;
memoria[19][2]=1;
memoria[19][15]=1;
//BRANCHI
memoria[20][2]=1;
memoria[20][4]=1;
//DIRECCION
memoria[21][11]=1;
memoria[21][13]=1;
memoria[21][14]=1;
memoria[21][15]=1;
//PRINT
memoria[22][0]=1;
memoria[22][1]=1;
memoria[22][2]=1;
memoria[22][14]=1;
memoria[22][15]=1;
// FIN
memoria[23][0]=1;
memoria[23][1]=1;
memoria[23][2]=1;
memoria[23][4]=1;}
```

Código 5.21: Código “memori4.cpp” (parte 2)

```
double memori4::ta(double t) {
//This function returns a double.
return Sigma;
}
void memori4::dint(double t) {
Sigma=INF;
}
void memori4::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the vALUe (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
/* Mapeo de puertos
port 0 read/write
port 1 direcciones
port 2 datos
port 3 vma*/
int res4;
if (x.port==1){  // (5)
vec1=*(vector16*)x.value;  //
res4 =
(int)((vec1.value[10]*32)+(vec1.value[11]*16)+(vec1.value[12]*8)+(vec1.value[
13]*4)+(vec1.value[14]*2)+(vec1.value[15]));
i=res4;  // (6)
//adress
}
if (x.port==2){  // (3)
//data
vec2=*(vector16*)x.value;  // (4)
}
if (x.port==0){
vec=*(vector16*)x.value;
if (vec.value[0]==1){  // (7)
//escritura
for (int j=0; j<16;j++){
memoria[i][j]=vec2.value[j];
}
}
if (vec.value[0]==0){  // (8)
//lectura
for (int k=0;k<16;k++){
y[k]=memoria[i][k];}
//y[k]=i;}
//y[16]=1;
sigma=0;
}}
}
Event memori4::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the vALUe.
//     %NroPort% is the port number (from 0 to n-1)
Sigma=0;
return Event(y,0);  // (8)
}
void memori4::Exit() {
//Code executed at the end of the simulation.
}
```

Código 5.22: Código “memori4.cpp” (parte 3)

En Código 5.19 debemos resaltar, que se declara una matriz de 64 x 16 elementos de tipo “double”, que corresponde a la simulación de una memoria física. Es por tanto en la matriz de nombre memoria (1) donde se almacenarán datos del programa a ejecutar tanto las instrucciones como los datos iniciales.

En Código 5.20, así como en parte de Código 5.21, mostramos el programa o código máquina a ejecutar, que como hemos indicado con anterioridad se encuentra almacenado en la matriz “memoria”. Debemos resaltar, que para escribir el código de las instrucciones, así como de los datos que se almacenan en la matriz, hemos recurrido a inicializar la matriz, en todas sus posiciones a un valor igual a 0(2), por ser este el valor más repetido en todas las instrucciones, sobrescribiendo en las posiciones que corresponda el valor 1, con posterioridad. Ese es el motivo, por el cual el lector verá que solo aparecen declarados las posiciones que tiene como valor 1.

En Código 5.21 y Código 5.22 podemos apreciar la forma para proceder a la escritura de un dato así como a la lectura del mismo.

Cabe destacar, que la forma utilizada es prácticamente la misma que la empleada en “Regarray”. En el puerto 2 se reciben los datos a escribir en la memoria (3), que al igual que en el caso de “Regarray”, por imperativo del formalismo DEVS no puede ser la misma entrada que salida de datos en PowerDEVS, separando la entrada y salida de datos de la memoria.

Nos parece adecuado resaltar, que para proveer del necesario aislamiento de entrada y salida tal y como se ha explicado con anterioridad, se ha tenido que proveer en la salida de la memoria de un registro que actúa separando la entrada del “bus” a “memora” de la salida “memory” al “bus”.

Una vez el dato está disponible en la entrada, se almacena en una variable intermedia que hemos denominado “vec2” (4). En el puerto 1, recibiremos la dirección binaria de la posición de la matriz donde deseamos escribir el dato (5),

procediendo a continuación a convertir la posición que hemos recibido en dato binario a dato decimal (6), cuando en el puerto 0 recibimos un 1 (7), procederemos a guardar en la matriz “memoria”, el dato almacenado en la variable “vec2”.

En cambio, si la señal recibida es un cero, procederemos a la lectura del dato guardado en la posición recibida en el puerto 1, y disponerlo en la salida de la memoria (8). La entrada denominada como “VMA” no es utilizada en este caso.

5.4.9 ALU: Este elemento, es el encargado de realizar las operaciones lógico/matemáticas. Para realizar su función, dispone de dos entradas (puede realizar tanto operaciones unarias, como binarias) de datos, y una entrada de control. Ésta es por donde se le indica al elemento, que operación es la que debe realizar. Dispone de una salida de datos por donde entrega los resultados.

```
//CPP:vector/ALU3.cpp
#if !defined ALU3_h
#define ALU3_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"
class ALU3: public Simulator {
// Declare the state,
// output variables
// and parameters
int sal,restot,restot1;
//double oper[4];
vector16 result,vec1,vec2;
double y[16];
double Sigma;
#define INF 1e20
public:
       ALU3(const char *n): Simulator(n) {};
       void init(double, ...);
       double ta(double t);
       void dint(double);
       void dext(Event , double );
       Event lambda(double);
       void Exit();
};
#endif
```

Código 5.23: Código “alu3.h”

En Código 5.23 se nos muestra el Archivo de definición.

```
#include "ALU3.h"
void ALU3::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
}
double ALU3::ta(double t) {
//This function returns a double.
return Sigma;
}
void ALU3::dint(double t) {
Sigma=INF;
}
void ALU3::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
vector16 vec3;
int res1,res2,res3,res4,res11,res12,res13,res14,oper,resto,a,b;
for (int i=0;i<16;i++){
y[i]=0;}
if (x.port==1){  // (1)
vec1=*(vector16*)x.value;
for (int i=0;i<16;i++){
if (vec1.value[i]==1){
}else{
vec1.value[i]=0;}
}
res1 =
((vec1.value[0]*32768)+(vec1.value[1]*16384)+(vec1.value[2]*8192)+(vec1.value[3]*
4096));
res2 =
((vec1.value[4]*2048)+(vec1.value[5]*1024)+(vec1.value[6]*512)+(vec1.value[7]*
256));
res3 =
((vec1.value[8]*128)+(vec1.value[9]*64)+(vec1.value[10]*32)+(vec1.value[11]*16));
res4 =
((vec1.value[12]*8)+(vec1.value[13]*4)+(vec1.value[14]*2)+(vec1.value[15]));
restot = res1+res2+res3+res4;  // (2)
}
if (x.port==2){  // (3)
vec2=*(vector16*)x.value;
for (int i=0;i<16;i++){
if (vec2.value[i]==1){
}else{
vec2.value[i]=0;}
}
res11 =
((vec2.value[0]*32768)+(vec2.value[1]*16384)+(vec2.value[2]*8192)+(vec2.value[3]*
4096));
res12=
((vec2.value[4]*2048)+(vec2.value[5]*1024)+(vec2.value[6]*512)+(vec2.value[7]*256
));
res13=
((vec2.value[8]*128)+(vec2.value[9]*64)+(vec2.value[10]*32)+(vec2.value[11]*16));
res14 =
((vec2.value[12]*8)+(vec2.value[13]*4)+(vec2.value[14]*2)+(vec2.value[15]));
restot1 = res11+res12+res13+res14;
}
if (x.port==0){  // (4)
vec3=*(vector16*)x.value;
if (vec3.value[0]==0&&vec3.value[1]==0&&vec3.value[2]==0&&vec3.value[3]==0){
//c=a
for (int i=0;i<16;i++){
y[i]=vec1.value[i];
}
```

Código 5.24: Código “alu3.cpp” (parte1)

```
sigma=0;
}
if (vec3.value[0]==0&&vec3.value[1]==0&&vec3.value[2]==0&&vec3.value[3]==1){
//c=a and b;
oper =restot & restot1;
for (int i=15;i>=0;i--) {
       resto= oper % 2;
       oper=(oper-resto)/2;
       y[i]=resto;
}
sigma=0;
}
if (vec3.value[0]==0&&vec3.value[1]==0&&vec3.value[2]==1&&vec3.value[3]==0){
//c=a or b;
oper =restot | restot1;
for (int i=15;i>=0;i--) {
       resto= oper % 2;
       oper=(oper-resto)/2;
       y[i]=resto;
}
sigma=0;
}
if (vec3.value[0]==0&&vec3.value[1]==0&&vec3.value[2]==1&&vec3.value[3]==1){
//c=not b;
oper=1;
//oper=8;
for (int i=15;i>=0;i--) {
       resto= oper % 2;
       oper=(oper-resto)/2;
       y[i]=resto;
}
sigma=0;
}
if
(vec3.value[0]==0&&vec3.value[1]==1&&vec3.value[2]==0&&vec3.value[
3]==0){
//c= a xor b;
oper =restot ^ restot1;
for (int i=15;i>=0;i--) {
       resto= oper % 2;
       oper=(oper-resto)/2;
       y[i]=resto;
}
sigma=0;
}
if (vec3.value[0]==0&&vec3.value[1]==1&&vec3.value[2]==0&&vec3.value[3]==1){
//c= a+b;
oper=restot1+restot;
for (int i=15;i>=0;i--) {
       resto= oper % 2;
       oper=(oper-resto)/2;
       y[i]=resto;}
sigma=0;
}
if (vec3.value[0]==0&&vec3.value[1]==1&&vec3.value[2]==1&&vec3.value[3]==0){
//c= a-b;
oper=restot-restot1;
//oper=8;
for (int i=15;i>=0;i--) {
       resto= oper % 2;
       oper=(oper-resto)/2;
       y[i]=resto;
}
sigma=0;
}
if (vec3.value[0]==1&&vec3.value[1]==0&&vec3.value[2]==0&&vec3.value[3]==0){
//c= a-1;
oper=restot-1;
//oper=8;
for (int i=15;i>=0;i--) {
       resto= oper % 2;
       oper=(oper-resto)/2;
       y[i]=resto;
```

Código 5.25: Código “alu3.cpp” (parte2)

```
}
sigma=0;
}
if (vec3.value[0]==0&&vec3.value[1]==1&&vec3.value[2]==1&&vec3.value[3]==1){
//c= a +1;
oper=restot+1;
//oper=8;
for (int i=15;i>=0;i--) {
        resto= oper % 2;
        oper=(oper-resto)/2;
        y[i]=resto;
}
sigma=0;
}
if (vec3.value[0]==1&&vec3.value[1]==0&&vec3.value[2]==0&&vec3.value[3]==1){
for (int i =15;i>=0;i--){
y[i]= 0;}
sigma=0;
}
}
}
Event ALU3::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the vALUe.
//     %NroPort% is the port number (from 0 to n-1)
Sigma=INF;
return Event(y,0);
}
void ALU3::Exit() {
//Code executed at the end of the simulation.
}
```

Código 5.26: Código “alu3.cpp” (parte 3)

En Código 5.24 se puede apreciar que en puerto 1 se recibe un dato de entrada (1), este es convertido a continuación a un valor decimal. Pues al igual que hemos hecho con los elementos “Shifter” y “Comp” las operaciones lógico/matemáticas se realizan en decimal. Por tanto cuando recibimos un dato realizamos su conversión a decimal (2). En puerto 2 (3) recibimos el 2 º dato, en el caso de operaciones binarias, procedemos al igual que en el caso anterior a su conversión a un valor decimal equivalente.

La indicación sobre la operación, que debemos efectuar sobre los datos recibidos, llega a través del puerto 0(4), una vez recibido el dato de operación, ésta se realiza y a continuación se vuelve a convertir a binario y es entregada en la salida.

En Código 5.25 y 5.26, se puede apreciar lo que se realiza en cada operación. Es indicada en binario por cuatro bits de operación, es decir las operaciones se indican en función del valor de cuatro bits, en este caso, no realizamos conversión a decimal para discriminar la operación que se indica.

5.4.10 Control: Este elemento, es sin duda el más complejo de la "CPU", es el encargado de decir, qué debe hacer el resto de elementos y en qué momento deben de hacerlo. Su funcionamiento ha quedado explicado en el Capítulo 3, pero entendemos que procede su descripción en PowerDEVS.

El elemento control dispone para realizar su función de 11 salidas, a través de las cuales, indicará a los elementos que están conectadas a dichas salidas, que es lo que deben hacer y cuándo. Estos por así decirlo, reaccionarán a las señales que les envía control.

En cuanto a las entradas, este elemento dispone de 5 entradas, una de ellas es la dedicada al reloj, por donde recibirá la señal de reloj, que es el que sincronizará el funcionamiento del elemento. Existe otra entrada, que se dedica a la función de reset. Otra entrada, es por donde cargará las instrucciones para ejecutarlas. Otra entrada es la de señal del comparador, es por esta puerta por donde recibirá el resultado de las operaciones de comparación (para las instrucciones de salto por ejemplo). Dispone además, de otra entrada "Ready", que es por donde recibirá la señal de la memoria, que le indica que ésta, ha depositado un dato en el bus.

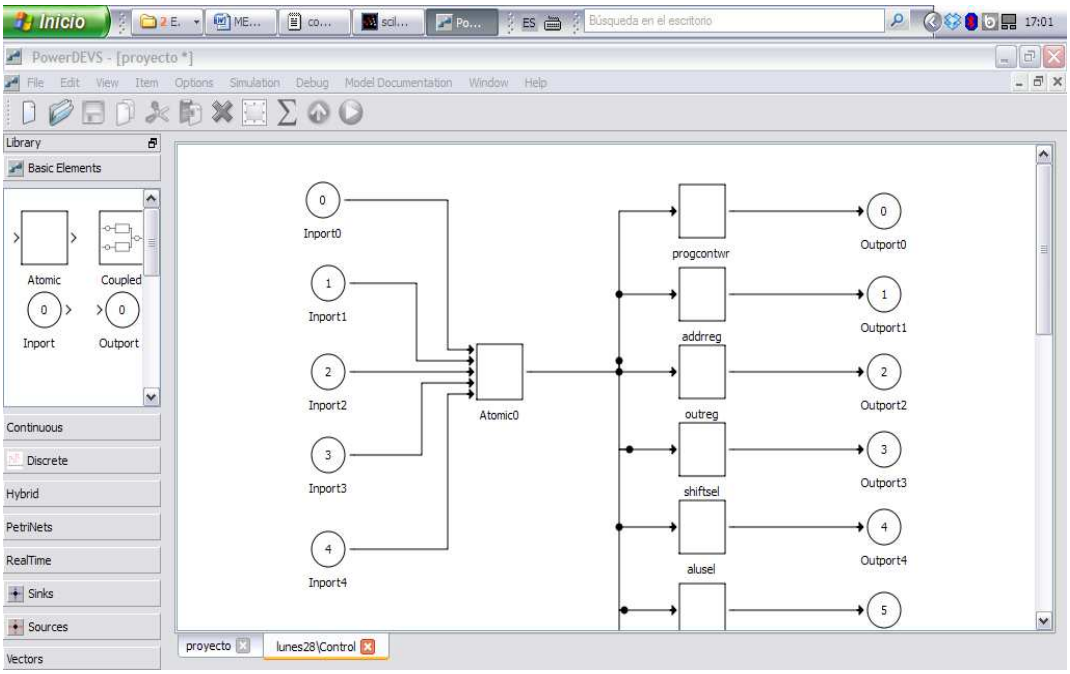


Figura 5.6: Fragmento del esquema de conexión del elemento “control”

Aunque en la imagen de Figura 5.6, no se ve el circuito completo, es más que suficiente para explicar el código que vamos a mostrar a continuación. Es el código del elemento nominado como “atomic0”. El cual se aprecia que en sus salidas, aparecen unos elementos con una única entrada y una única salida, que están nombrados como los nombres de las salidas de la Figura 5.1.

El funcionamiento de estos elementos es el siguiente, atomic0 presenta en su salida un vector de 24 posiciones. Cada posición del vector corresponde a un elemento de la salida, así por ejemplo, la posición 0 del vector, define la señal correspondiente a “Progcont”, en la posición 1 corresponde a “AddrReg” y así sucesivamente como se muestra en Tabla 5.1.

Este vector llega a todos los elementos, pero cada elemento mira si la posición que le corresponde (posición del vector), viene con un 0 o un 1 en cuyo caso repetirá en su salida la señal de entrada, en caso de tener otro valor es ignorado el valor de la entrada. Esta implementación, permite, que varias salidas

sean enviadas simultáneamente (en un mismo instante de simulación) a distintos elementos.

Posición vector	Señal a la que representa
0	ProgCnt
1	AddrReg
2	OutReg
3	ShiftSel
4	“
5	“
6	alusel
7	“
8	“
9	“
10	CompSel
11	“
12	“
13	OpRegSel
14	InstrSel
15	RegSel (1)
16	“ (1)
17	“ (1)
18	Reg Rd/wr
19	RD/Wr
20	VMA
21	RegCont
22	Regsal
23	Result

Tabla 5.1: Vector de salida del elemento “control”

En la Tabla 5.1 podemos establecer la correspondencia entre la posición del vector y la salida que representa.

Para explicar el código que implementa este elemento, entendemos que por su extensión, no procede mostrarlo completo en los diferentes cuadros de texto. Por tanto en estos cuadros de texto, iremos mostrando parte del código que por su importancia merezca ser explicado.

Remitimos al Anexo B, donde tendremos el código completo del elemento control.

```
//CPP:Projectro/control4.cpp
#if !defined control4_h
#define control4_h

#include "simulator.h"
#include "event.h"
#include "stdarg.h"

#include "vector/vector16.h"
#include "vector/vector24.h"


class control4: public Simulator {
// Declare the state,
// output variables
// and parameters
int valor,res1,fase;
int steep,instr;
double e[2],n,l,p,f,g,h,m;
vector16 vec3,vec1;
//out
double y[24];  // (1)
double Sigma;
#define INF 1e20
public:
        control4(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

Código 5.27: Código “control4.h”

Cabe destacar en Código 5.27, que en la salida se declara un dato de tipo doble que es un vector de 24 posiciones (1) y que corresponde con la Tabla 5.1. Es en este vector de salida, donde se indica las señales de control que deben de estar activas en cada instante de simulación. Es decir los elementos de Figura 5.5 que están conectados a la salida de “atomic0”, vigilan sobre la posición del vector que le corresponde, para saber si deben de repetir la señal de entrada o no. La implementación de este sistema de señales, permite el envío simultáneo de señales, a varios elementos en un mismo instante de simulación, esto da una mayor versatilidad al diseño.

Debemos reconocer, que esta característica no ha sido utilizada en la implementación del circuito, pues al optar por una implementación puramente

secuencial, nos hemos limitado a seguir una secuencia de ejecución, en la que no se han simultaneado las señales de salida. Es decir, hemos realizado una sola función por cada señal de reloj, esto es claramente ineficiente y en la actualidad no se utiliza en ningún hardware, pero en el caso que nos ocupa, dada su sencillez, este modo de funcionamiento parece el más lógico, por facilitar la implementación del circuito.

```
if (x.port==0){

a[0]=*(double*) x.vALUe;
if (a[0]==1){
if (steep==0)
steep = 11;
}}
//////////
/////////
switch(steep){
case 11:
y[6]=1;
y[7]=0;
y[8]=0;
y[9]=1;
Sigma=0;
steep=12;
break;
case 12:
// continua
```

Código 5.28: Fragmento de código de “control4.cpp”

El funcionamiento del elemento control en PowerDEVS, es básicamente el mismo que la implementación en VHDL, no tanto así en sus formas (cada uno está implementado en un lenguaje diferente), sino en su funcionamiento básico.

En Código 5.28, se nos muestra la forma en que inicia su funcionamiento el elemento control. Podemos apreciar claramente que su funcionamiento se inicia cuando por el puerto 3, llega una señal con lo que comienza el reset del circuito, a través de un case con la variable “steep”.

Aunque no se nos muestra el código completo, la variable “steep” va subiendo de valor y completando pasos en el reset del circuito. Una vez

completados los pasos de la función reset. Se continua en otro case, con la variable “instr” (1) que se muestra en Código 5.29.

```
case 19:
switch(instr){
// NO OPERACION.
case 0:
instr=400;
break;

//load.
case 10:
y[15]=vec1.vALUe[10];
y[16]=vec1.vALUe[11];
y[17]=vec1.vALUe[12];
Sigma=0;
instr=11;
```

Código 5.29: Fragmento de código de “control4.cpp”

La variable “instr”, toma el valor de la instrucción que ha recibido por el puerto 2(1) como podemos apreciar en Código 5.30.

El valor binario se traduce en un valor decimal y se multiplica por diez, para obtener el siguiente paso de instrucción (2).

```
if (x.port==2){  // (1)
vec1=*(vector16*) x.value;
fase =
(int)((vec1.value[0]*16)+(vec1.value[1]*8)+(vec1.value[2]*4)+(vec1.value[3]*2)+
(vec1.value[4]));
}
////////

case 18:
instr=fase*10;  // (2)
//instr=1;
steep=19;
break;
```

Código 5.30: Fragmento de código de “control4.cpp”

```
//inc
case 70:  // (1)
f=0;
g=1;
h=1;
m=1;
n=1;
l=1;
p=1;
instr=71;
break;
case 71:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
sigma=0;
instr=72;
break;
case 72:
y[18]=1;
Sigma=0;
instr=73;
break;
case 73:
y[22]=1;
Sigma=0;
instr=74;
break;
case 74:
y[22]=0;
y[6]=f;
y[7]=g;
y[8]=h;
y[9]=m;
Sigma=0;
instr=75;
break;
case 75:
y[3]=n;
y[4]=l;
y[5]=p;
Sigma=0;
instr=76;
break;
case 76:
y[2]=1;
Sigma=0;
instr=77;
break;
case 77:

y[15]=vec1.value[13];  // (3)
y[16]=vec1.value[14];  // (3)
y[17]=vec1.value[15];  // (3)
sigma=0;  // (2)
instr=78;
break;
case 78:
y[18]=0;
Sigma=0;
instr=310;
break;
```

Código 5.31: Fragmento de código de “control4.cpp”

En el ejemplo que nos ocupa, la instrucción corresponde con el código de instrucción “00111”, cuya función es incrementar el valor del dato de entrada en una unidad. Vemos por tanto que en Código 5.31 el valor de “instr” es de 70 (1).

Recordemos que es el valor binario del código de la instrucción, multiplicado por 10. Se aprecia a continuación, que se van dando órdenes de ejecución, a los distintos elementos, siguiendo un patrón que corresponde con las posiciones de la tabla 5.1 (1). Es decir, en Código 5.31 las posiciones marcadas como (3) corresponden con la señal “regsel”, de la tabla 5.1 (1) que indican una dirección donde escribir o leer un dato en el “Regarray”.

Vemos que en cada ejecución de un paso, el valor de Sigma pasa a valer cero y por tanto el dato cargado en el vector, es enviado a la salida. Tenemos que explicar, que el dato es puesto en la salida del elemento del “Atomic0”, de la Figura 5.5. Luego, el dato todavía no se ha enviado al elemento “Regarray”, antes pasara por el elemento señalado en la Figura 5.5 como “addrreg”, que tal y como se aprecia en la salida, está conectado a la salida 2 del elemento “Lunes28\control”.

```
//CPP:vector/regsel.cpp
#if !defined regsel_h
#define regsel_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class regsel: public Simulator {
// and parameters
//out
double y[3];  // (1)
double Sigma;
#define INF 1e20
public:
       regsel(const char *n): Simulator(n) {};
       void init(double, ...);
       double ta(double t);
       void dint(double);
       void dext(Event , double );
       Event lambda(double);
       void Exit();
};
#endif
```

Código 5.32: Código “regsel.h”

En Código 5.32, podemos ver el archivo de declaración del elemento “Regsel”. Resaltar que la salida se declara de un tipo “doublé”, en un vector de 3 posiciones (1). Es mediante ese vector como se transmitirá la dirección a

“Regarray”. En Código 5.33 mostramos como el elemento “regsel”, vigila la posición 15 del vector de salida de “element0”. En el caso de tener un valor < 2 (típicamente será un valor 1 ó 0), repetirá el valor que tenga en las posiciones 15, 16 y 17. Esas posiciones en la salida (marcada como 2), si seguimos el esquema de la Figura 5.3 veremos conecta con “Regarray”.

Como se aprecia en la Figura 5.5, cada salida de control, estará conectada a un elemento similar, al presentado. Este vigila cada una de las posiciones del vector, que le corresponden. Quedando conectado al elemento, al que provee de señal.

```
#include "regsel.h"
void regsel::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the
editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
}
double regsel::ta(double t) {
//This function returns a double.
return Sigma;
}
void regsel::dint(double t) {
Sigma=INF;
}
void regsel::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.vALUe' is the vALUe (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
vector24 ent;
ent=*(vector24*)x.value;
if (ent.value[15]<2){  // (1)
y[0]=ent.value[15];  // (2)
y[1]=ent.value[16];  // (2)
y[2]=ent.value[17];  // (2)
sigma=0;}
}
Event regsel::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&VALUe% points to the variable which contains the vALUe.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void regsel::Exit() {
//Code executed at the end of the simulation.
}
```

Código 5.34: Código “regsel.cpp”

5.5 CONEXIÓN DE LOS ELEMENTOS POWERDEVS

En el Capítulo anterior, hemos descrito el código que compone los distintos elementos, de los que va a estar formada nuestra implementación DEVS del circuito. Queda por tanto para que estos elementos formen verdaderamente un circuito el interconectar los elementos entre sí.

El Capítulo 3, pudimos comprobar que la conexión (su equivalente eléctrica), de los distintos elementos en VHDL, se determinaba mediante software que debíamos definir nosotros en nuestro caso en el archivo “cpu.vhd”.

En el caso de la implementación en PowerDEVS, la conexión de los distintos elementos se realiza de manera gráfica, mediante la interfaz de nuestro proyecto. Por tanto, para realizar la conexión de los distintos elementos, no necesitamos, tener que implementar ningún software por nuestra parte pues esto lo hace el entorno de manera transparente para el usuario. Por nuestra parte, solo es necesario, realizar una unión gráfica entre los distintos puertos, de los distintos elementos.

Esta característica de PowerDEVS, además de facilitar la implementación de circuitos, presenta además la ventaja, que a la par que se conecta el circuito, se está realizando un croquis del circuito que estamos implementando. Si, además, deseamos que el croquis represente un esquema eléctrico, solo tendremos que sustituir el icono del elemento, por el icono normalizado para el mismo.

Dentro de las distintas conexiones que se establecen en el circuito, cabe destacar una, que aun no siendo implementada como un elemento software más del circuito, si tiene una crucial importancia, pues es el elemento sobre el cual se comunican entre sí los distintos elementos. Esta conexión es la que denominamos "bus".

Por el "bus", es donde circulan tanto datos como instrucciones, estando accesibles a todos los elementos conectados a dicho bus. Las otras conexiones que aparecen en el circuito, son las que se dedica a señales. Éstas simplemente, se usan para dar órdenes desde control a los distintos elementos y viceversa. Las señales de control, generalmente enlazan solo dos elementos y son direccionales mientras que el "bus", conecta múltiples elementos y los datos e instrucciones no tiene un sentido definido, simplemente están disponibles para todos los elementos que están conectados al "bus", una vez han sido conectados con éste.

5.6 CONCLUSIONES

La implementación DEVS del circuito, presenta unas características muy definidas, que parece encorsetar, un tanto, la forma de desarrollar las implementaciones de los diferentes circuitos. Quizá, en diferentes casos, podrían requerir pautas diferentes, que facilitaran cada caso en particular. Pero en cambio perderíamos lo que supone de normalización el formalismo DEVS. Pues éste, unifica la forma de definición de todos los circuitos que podamos implementar.

Por tanto, en todas las implementaciones que desarrollemos mediante el formalismo DEVS, tendrán las mismas funciones y se probarán siguiendo un mismo patrón en todos los casos. Esto supone una normalización en la implementación y desarrollo.

La implementación que hemos realizado del circuito, en el entorno PowerDEVS, implica la garantía que la implementación sigue las norma y criterios del formalismo DEVS. Con la simulación mediante PowerDEVS, queda garantizado que los resultados corresponden al comportamiento algorítmico de la misma implementación, mediante la descripción matemática en DEVS.

6

VALIDACIÓN DEL MODELO DEVS DEL CIRCUITO

6.1 INTRODUCCIÓN

Vamos a realizar una pequeña explicación, a nivel teórico, a modo de introducción, para establecer qué se entiende como pruebas de validación y el enfoque que en el ámbito de la ingeniería de software se sigue en este proceso. Nuestro objetivo con esta pequeña introducción es situar las bases teóricas del proceso realizado.

En el ámbito de la ingeniería del software las pruebas del software suelen denominarse como "Verificación" y "Validación" (VyV); Pressman (2006: 384) define, la "Verificación es el conjunto de actividades que aseguran que el software implementa correctamente una función específica. Validación, es el conjunto de actividades, que aseguran que el software construido corresponde con los requisitos del cliente". Otra definición a la que recurre el mismo autor es, [BOE81] Verificación: "¿Estamos construyendo el producto correctamente?". Validación: "¿Estamos construyendo el producto correcto?".

Bien, es en este capítulo, donde desarrollaremos toda la información que pueda aportar luz a todo lo concerniente con el proceso realizado de validación y verificación del producto software producido.

Recordamos, que estas bases teóricas han sido explicadas en el Capítulo 4.1 (pág. 137). Por tanto no procede recordarlas, pero sí procede retomar el tercer punto del anteproyecto para reflexionar sobre lo que estamos tratando de realizar en este capítulo y en los sucesivos.

En el tercer punto dice, “Aplicar el formalismo DEVS a la descripción de un sistema de complejidad media…”. Bien para el cumplimiento de este punto, consideramos que se debe realizar la demostración, que el circuito se ha realizado siguiendo los criterios del formalismo DEVS, lo cual queda garantizado al usar un entorno de desarrollo que implementa dicho formalismo, como es PowerDEVS. Esta implementación, y el correcto funcionamiento de nuestro desarrollo, hacen suponer que el circuito implementado es correcto.

En el cuarto punto dice, “Aprender a manejar un entorno de simulación para DEVS, como puede ser PowerDEVS…”, este punto igual que el siguiente se puede considerar alcanzado, con la terminación correcta del PFC.

En el punto quinto nos dice “Aprender a describir circuitos de complejidad media usando VHDL y sus banco de pruebas…” se considera validado en el Capítulo 4, pues es necesario su cumplimiento para poder desarrollar con éxito este PFC.

El punto sexto dice, “Validar el modelo DEVS comparando los resultados obtenidos de su simulación con la simulación en PowerDEVS y la comparación de los resultados obtenidos, de simular el banco de pruebas y el circuito descritos en VHDL”. Este objetivo, es el que se presenta en este capítulo. Con los resultados obtenidos con la simulación de programas equivalentes en ambos entornos, que

por supuesto deben arrojar los mismos resultados independientemente del entorno utilizado, pues el circuito a simular es el mismo. Entendemos que esto queda demostrado con la documentación aportada y que sometemos a consideración del director del PFC y del tribunal.

En este capítulo, se va a desarrollar una breve descripción del funcionamiento básico, para la simulación del entorno PowerDEVS, utilizado en el desarrollo del presente PFC. Cabe destacar, que no se pretende en este capítulo, hacer una descripción exhaustiva del funcionamiento de dicho entorno de simulación, pues ya existen manuales y tutoriales que están reflejados en la bibliografía del presente documento. Sino que más bien lo que se pretende, es ser una guía rápida, para que una persona que no haya tenido oportunidad de utilizar dichos entornos, pueda realizar las simulaciones oportunas, para la comprobación del cumplimiento de los objetivos del PFC. Sin la necesidad de tener que recurrir a los citados tutoriales, mucho más completos sin duda, pero más farragosos para el objetivo tan simple que se persigue.

6.2 PROCEDIMIENTO DE SIMULACIÓN EN POWERDEVS

Para poder hacer uso del entorno que se proporciona en el “CD” del PFC, bastará solo con copiarlo en un directorio de nuestro disco duro que deseemos. Una vez copiado, o bien se podrá realizar un acceso directo o bien abrirlo directamente pulsando dos veces sobre el archivo ejecutable “pdme” que se encuentra en la ruta “PowerDEVS/bin”, tal y como se nos muestra en la Figura 6.1.

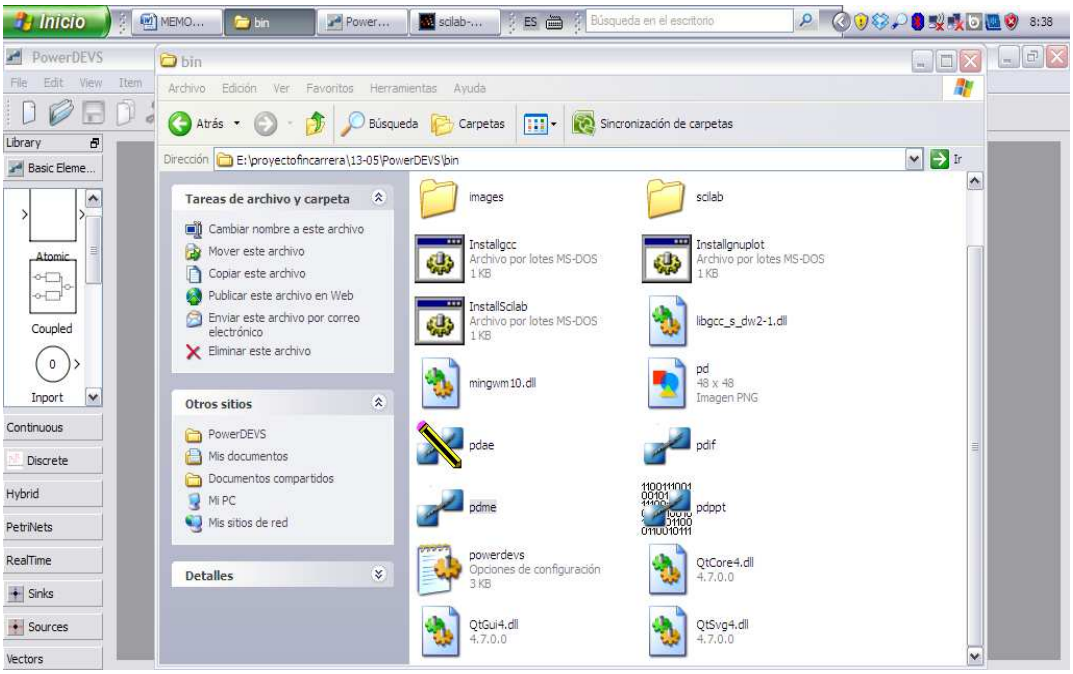


Figura 6.1: Iniciar entorno PowerDEVS paso1

Tras lo cual se nos abrirá el entorno, y aparecerá la pantalla de “Scilab” si la minimizamos nos aparecerá la interface de PowerDEVS, que es la que se muestra en la Figura 6.2.

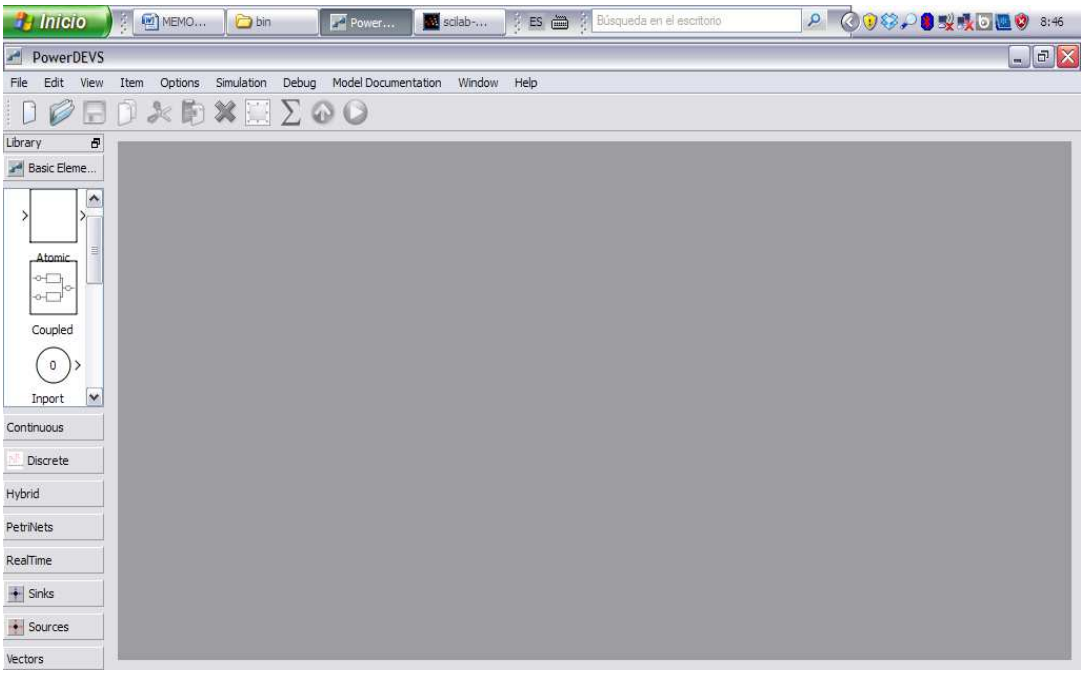


Figura 6.2: Iniciar entorno PowerDEVS paso 2

Para abrir nuestro programa, procederemos pulsando sobre “File/ Open”, pues si pulsamos directamente sobre los últimos proyectos, utilizados recibiremos un mensaje de error que no encuentra el archivo pues la ruta absoluta en que se guardó el proyecto. Pues no tiene por qué coincidir, con la que tengamos en ese momento. En la Figura 6.3 se muestra la forma correcta como cargar el proyecto.

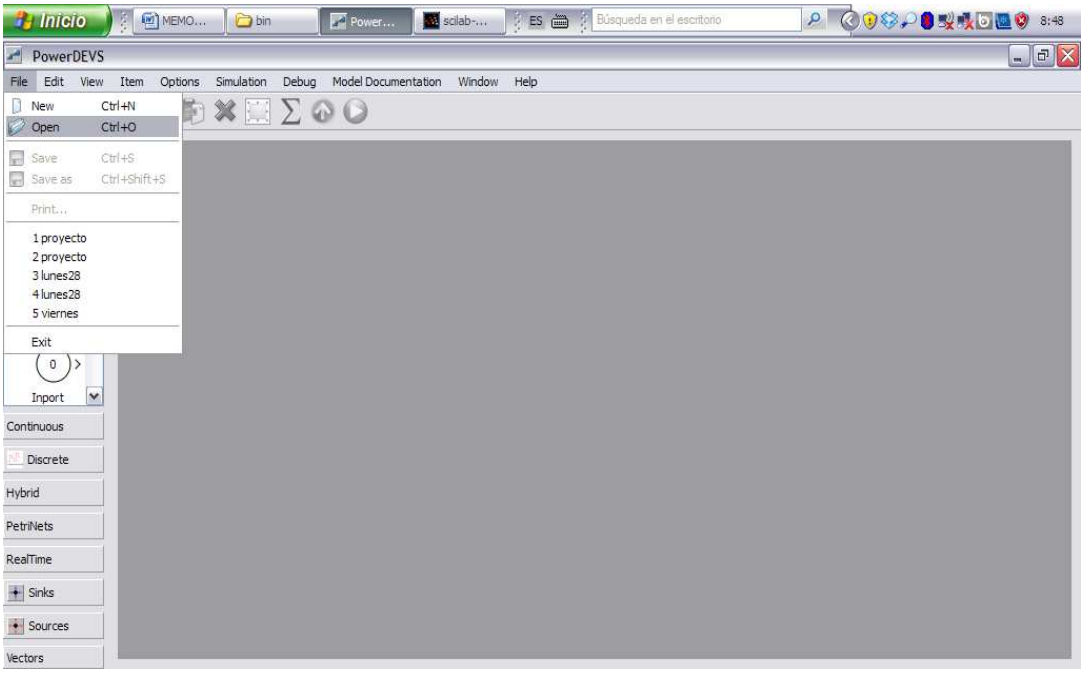


Figura 6.3: Iniciar entorno PowerDEVS paso 3.

Deberemos situarnos sobre la carpeta “example” y seleccionar el archivo “proyecto.pdm” tras lo cual, se nos presentará nuestro proyecto en pantalla, estando ya dispuesto para nuestra manipulación.

En la Figura 6.4 mostramos la selección del archivo que procederá a mostrar en la interfaz nuestro proyecto. Recordar que los códigos fuente de todos los elementos que conforman el proyecto se encuentran en un mismo directorio con dicho nombre, que podremos consultar en la ruta PowerDEVS\atomic\proyecto.

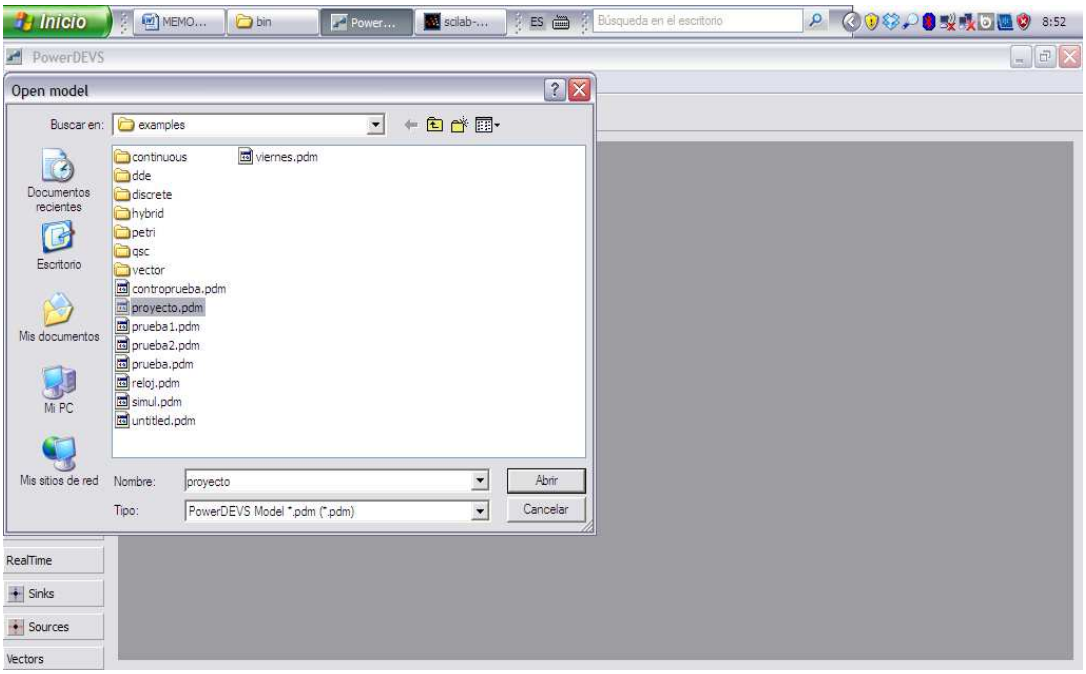


Figura 6.4: Cargar proyecto en el entorno

Cabe destacar, que cada uno de los casos de prueba que conforman el banco de pruebas de nuestro proyecto, debe ser cargado en el elemento nombrado como “memory” según la que se cargue, se realizará la simulación de un subprograma en la CPU diferente. Los elementos que podemos cargar para la correcta simulación son, los denominados “memori”, seguidos de un numeral como por ejemplo “memori5”. En el apartado correspondiente de esta memoria, se reflejaran las correspondencia entre los nombres y lo que realiza dicho subprograma.

Para cargar un nuevo subprograma deberemos seguir los siguientes pasos, consistentes en editar el elemento “memory” tal y como se nos muestra en la Figura 6.5.

Seleccionamos el elemento memory y con el botón derecho seleccionamos la opción Edit y en la pantalla.

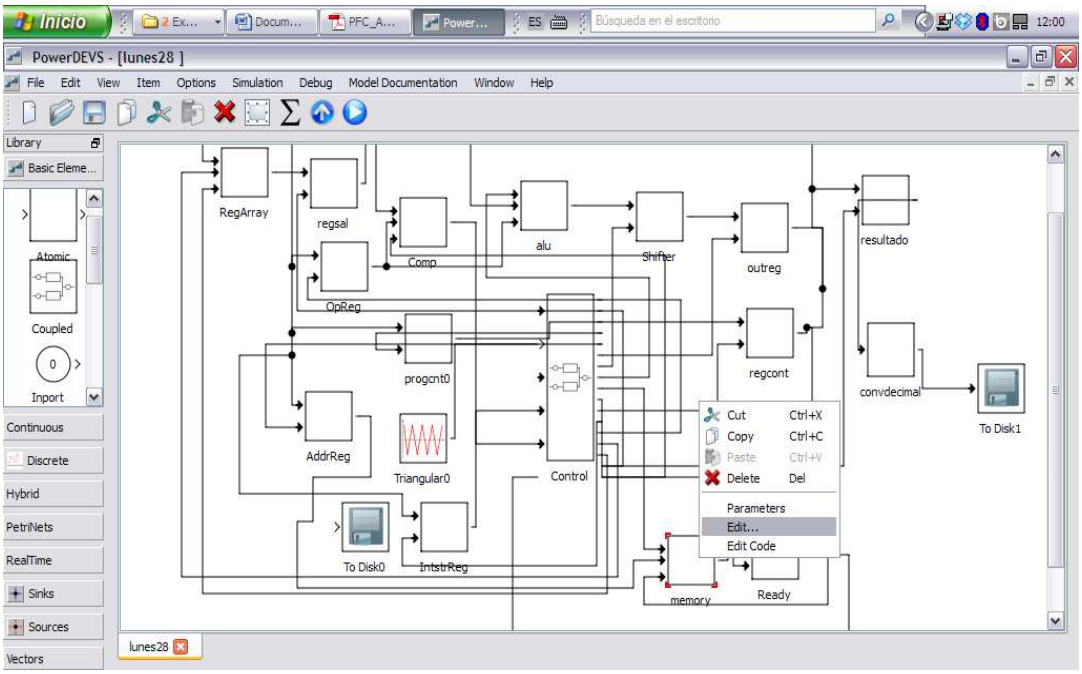


Figura 6.5: Edición de un elemento

Tras lo cual pasaremos a cargar el elemento “memory” deseado, tal y como se nos muestra en la Figura 6.6.

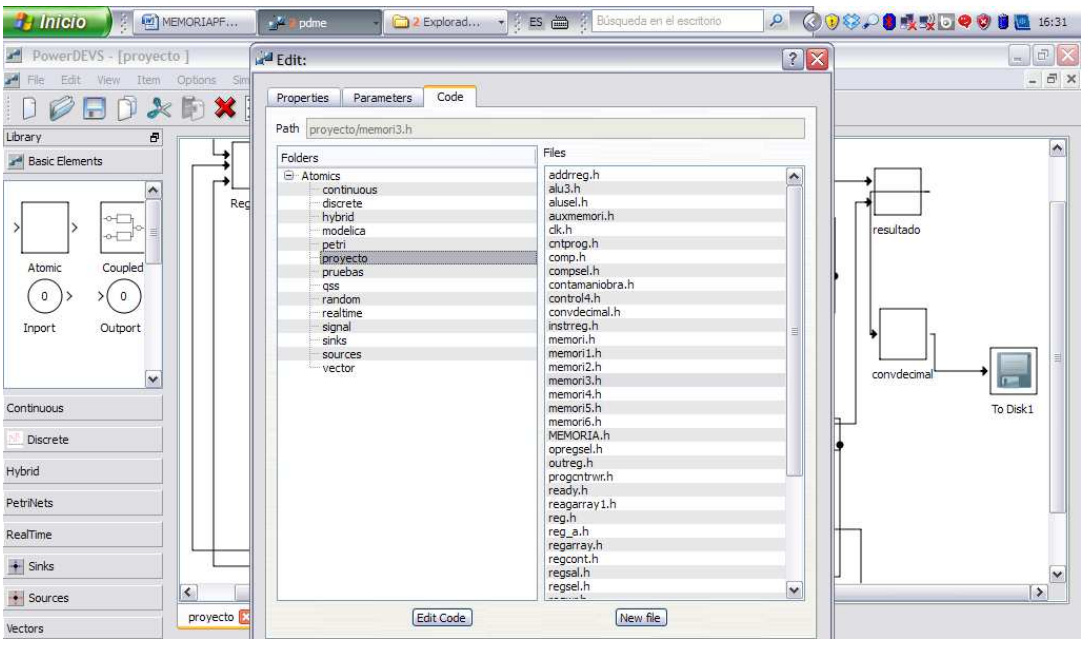


Figura 6.6: Selección de archivo para elemento

Seleccionaremos la carpeta adecuada (“proyecto” en nuestro caso) y el elemento que queremos probar. En nuestro caso los archivos nombrados con

"memori" seguidos de un numeral, posteriormente deberes pulsar sobre el botón "Apply" y sobre "Ok" y quedara cargado el archivo que hemos seleccionado.

Cabe destacar, que este proceso, se puede realizar con cualquier elemento gráfico. En nuestro caso, cada archivo nombrado como "memori" representa cada caso de prueba que podemos simular, no siendo necesario actuar sobre el resto de elementos del programa para realizar una correcta simulación.

Si deseamos comprobar o modificar el código del archivo que hay cargado sobre cada elemento, deberos pulsar con el botón derecho, sobre el elemento y seleccionar la opción "EditCode", como se nos muestra en la Figura 6.7

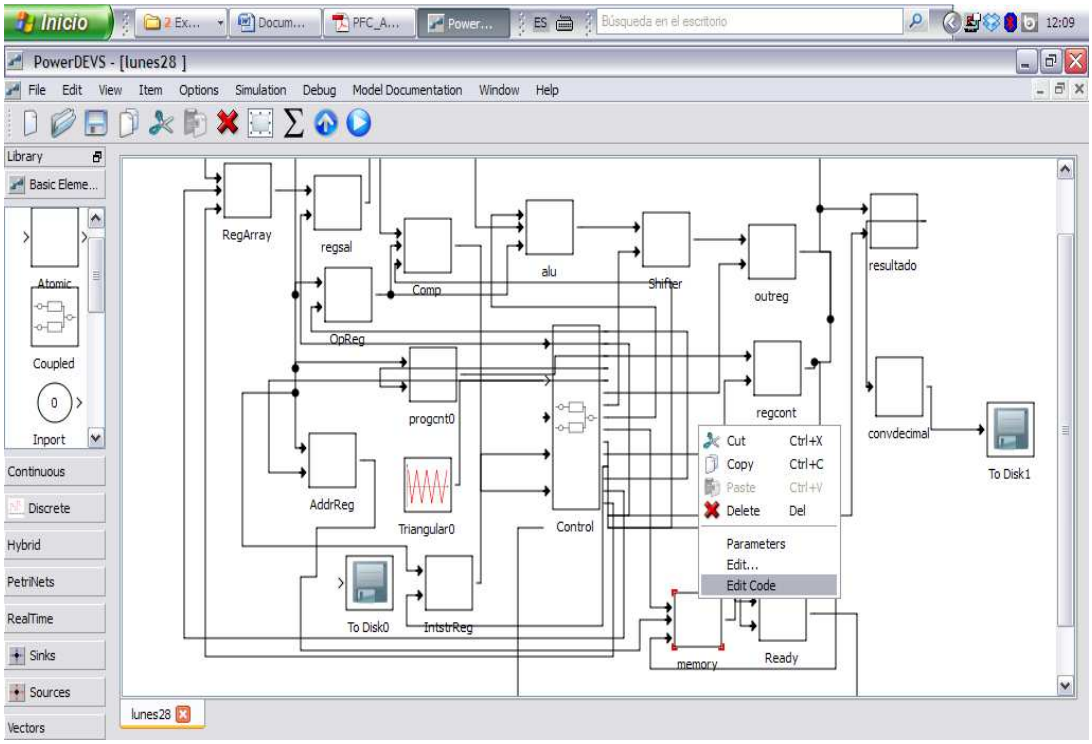


Figura 6.7: Edición del código del elemento seleccionado

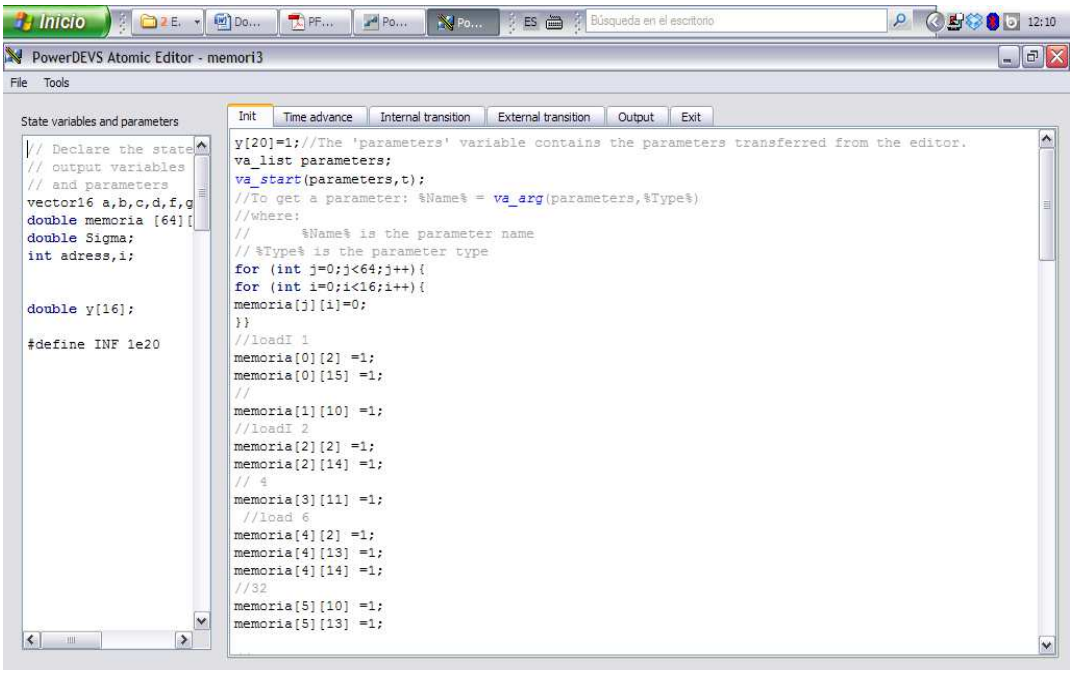


Figura 6.8: Pantalla de edición y visualización de código de elemento DEVS

En la Figura 6.8, se nos muestra la pantalla donde podremos comprobar el código que vamos a ejecutar. Éste se encuentra en la pestaña “Init” cargada en la matriz nominada como “memoria”.

Como se explicó en el anterior capítulo, la matriz “memoria” almacena en código máquina, las instrucciones que se cargarán durante la ejecución del subprograma en el elemento “Control”, por tanto serán los que se ejecute. Entendemos, que la forma en que se almacena es un tanto confusa de seguir para el lector, pues para el lector sería más cómodo la definición completa de cada vector que implementa una instrucción tal y como se nos muestra en la Tabla 6.1.

Sin embargo, por comodidad nuestra y debido a la elevada cantidad de posiciones de la matriz cuyo valor es “0”, nos ha resultado más sencillo inicializar la matriz con todos sus elementos al valor “0”, para posteriormente declarar al valor “1”, en las posiciones correspondientes.

En la Figura 6.9 se nos muestra el primer paso para la simulación del proyecto tras pulsar sobre el icono .

Una vez pulsado sobre el botón “Play”, (recordemos que también se puede realizar la simulación mediante el menú general) se nos presentará la pantalla que se muestra en la Figura 6.9, en la cual podemos elegir el tipo de simulación y la duración de la misma. Una vez hecho esto, al realizar la simulación, veremos que en la parte de abajo del menú emergente, aparece un mensaje que indica que la simulación ha sido completada.

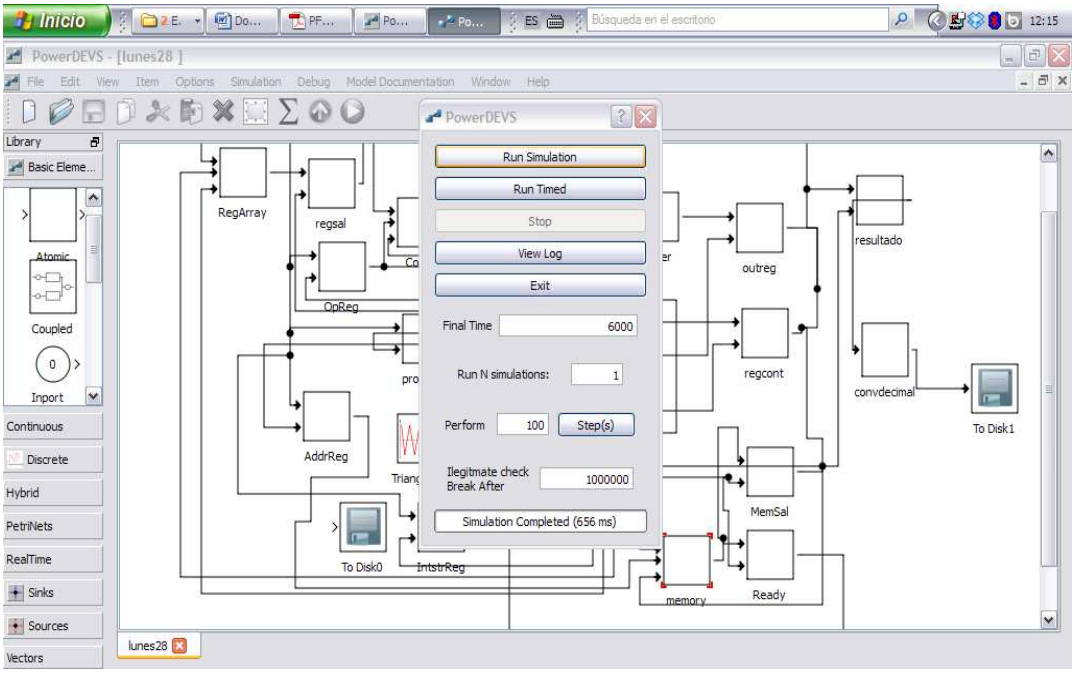


Figura 6.9: Simulación del proyecto DEVS paso1; parámetros de simulación

En nuestro caso (por haber elegido “To disk”), para poder acceder a los resultados deberemos acceder a un archivo de tipo Excel, en el cual el programa ha ido depositando los valores que ha obteniendo durante la simulación.

En cada celda del documento Excel, aparecen un par de valores separados por una coma, el primer valor es el tiempo de simulación transcurrido, y el segundo es el valor obtenido en la simulación en ese instante.

La ruta para acceder a dicho archivo, es la indicada en la Figura 6.10 \PowerDEVS\output. Evidentemente la ruta absoluta cambiará según hayamos guardado la carpeta nombrada como PowerDEVS, no así su ubicación dentro de dicha carpeta. Cabe destacar, que una vez abierto el archivo Excel, deberemos proceder a cerrarlo antes de iniciar una nueva simulación, de lo contrario el programa lanzara un error, diciendo que no ha podido abrir dicho archivo.

En la Figura 6.10, también se aprecia la carpeta “plots”, en ella encontraremos archivos de salida de tipo Excel, que son los resultados de simulación de circuitos, cuyo dispositivo de salida es el “Gnuplot”. Este elemento nos muestra por pantalla un gráfico, éste se construye con los valores guardados en dichos archivos en formato de par de valores (tiempo de simulación, valor de salida). Entendemos que las simulaciones que se realizan de esta forma, pueden ser consultadas por tanto en ambos formatos, lo cual puede clarificar bastante los gráficos que obtenemos.

En nuestro caso, no hemos utilizado dicho dispositivo para nuestra salida, por tanto esta información se añade como un complemento más del proyecto. El archivo es igual al mostrado en la Figura 6.11 lo único que cambiará es el nombre del archivo, pues en estos casos será un numero de simulación que asignara el entorno.

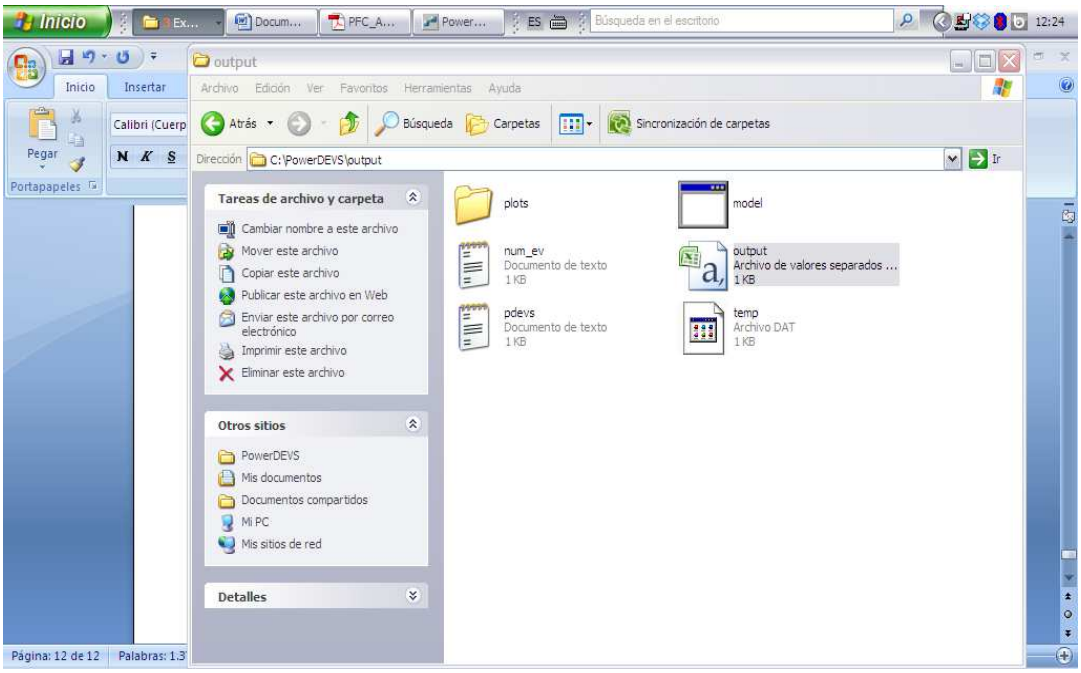


Figura 6.10: Simulación proyecto DEVS paso2; abrir archivo Excel de salida de datos

Y se nos mostrará un archivo similar al de la Figura 6.11.

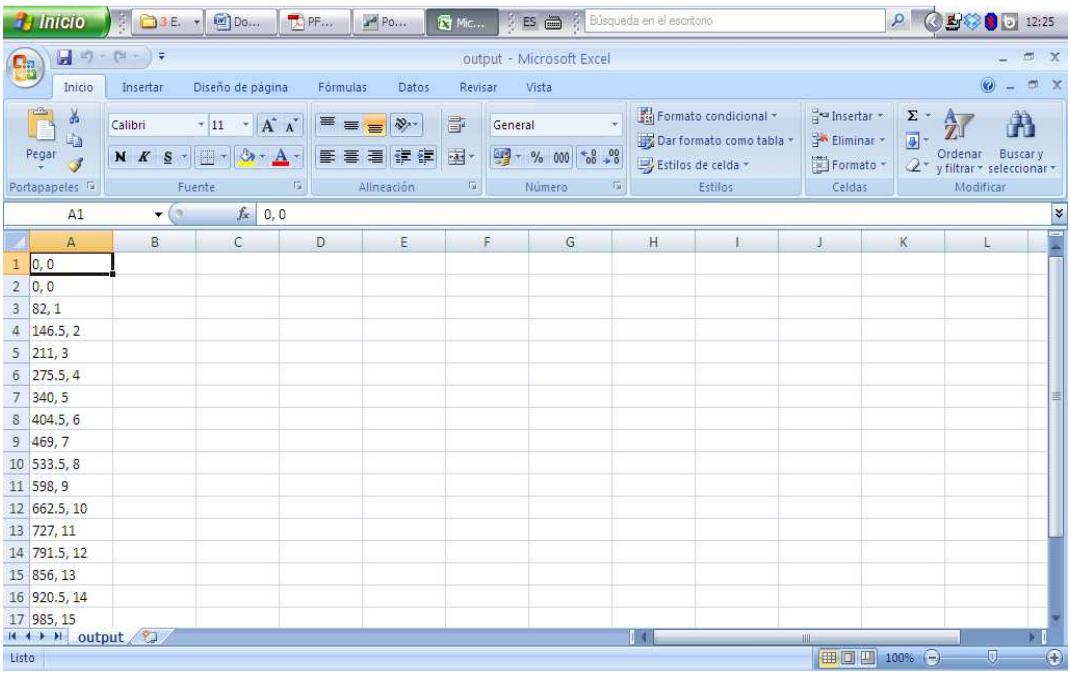


Figura 6.11: Simulación proyecto DEVS paso 3; resultados obtenidos en formato "par de valores"

Hasta aquí, se muestran los pasos básicos para realizar la simulación de nuestro proyecto. Evidentemente el entorno dispone de muchas más

funcionalidades, pero el objetivo que se persigue, es el de ser una guía rápida para la simulación sin más pretensiones.

6.3 SIMULACIÓN Y VALIDACIÓN DEL BANCO DE PRUEBAS

Cabe destacar, que las diferencias en cuanto a la simulación de cada entorno, hace que los subprogramas del banco de pruebas no sean completamente idénticos, línea de código a línea. En el caso de PowerDEVS, ha sido necesaria la implementación de una nueva instrucción, la cual se ha justificado con anterioridad su necesidad, que hemos denominado “print”. Ésta se dedica, a mostrar por pantalla (mediante el archivo Excel en nuestro caso), los datos que nosotros indiquemos, para de esta manera conseguir mayor claridad del funcionamiento del código que hemos implementado en PowerDEVS.

Esta instrucción, en el caso de ModelSim pierde su sentido, por la posibilidad de seleccionar señales y saber en todo momento cuál es su valor. Esta característica hace innecesaria dicha instrucción, por tanto aun siendo programas equivalente en cuanto a los resultados que deben arrojar, no son exactamente iguales, al no necesitar de dicha instrucción; que en el caso de PowerDEVS, si he utilizado, en aras de una mayor claridad en las simulaciones.

En la elección del banco de pruebas se han establecido dos criterios, que he considerado importantes, en pos de una facilidad de comprobación de resultados. Por tanto, hemos pretendido por una parte realizar subprogramas de prueba que fueran por una parte claros y cortos (una cosa va en función de la otra) y que a su vez abarcara la mayoría, si no todos, los tipos de instrucciones que se ha

implementado, no todos a la vez, sino en el cómputo total de todos los subprogramas que componen la batería de pruebas.

Nos parece importante resaltar qué se entiende, o a qué nos referimos como subprograma. Nos ha parecido que la mejor manera de probar un software que imita el comportamiento de una CPU conectada a una memoria (objetivo fundamental del presente proyecto), sería la ejecución en el ámbito de este software, de un subprograma escrito en código máquina, puesto que nuestro objetivo es probar el funcionamiento del software que da soporte a la simulación de dicha CPU. Por tanto, como subprograma para este caso en concreto, es el programa escrito en código máquina, que es el que va a ser interpretado dentro de la simulación del software creado a tal efecto. Quizá la elección de la palabra no sea del todo afortunada, pero nos parece que mediante la explicación que hemos proporcionado, así como de los casos a los que me voy a referir en adelante, dejaran claro el concepto de la misma, así como su función dentro del banco de pruebas.

Como hemos comentado con anterioridad, los subprogramas que vamos a probar dentro de cada simulación del software del proyecto (banco de pruebas). No son idénticos línea a línea si se formulan para PowerDEVS como si son formulados para su ejecución en VHDL. Estas pequeñas diferencias (aunque ya las he explicado con anterioridad volveremos en cada caso sobre ellas). También se manifiestan a su vez, en la forma de presentación de los resultados, que no resultarán relevantes en absoluto, una vez sean explicadas, para comprobar la correcta validez de los resultados.

Como metodología de prueba, para seguir un mismo procedimiento para todos los casos de prueba. Comenzaremos al inicio de cada caso de prueba, con una tabla que mostrará el código máquina que se va a probar. En dicha tabla, se podrá observar el código binario de cada instrucción y a su vez su correspondiente traslación a una instrucción de "Ensamblador". A continuación, al finalizar la tabla, se presentará en pseudocódigo (mas bien código C incompleto), el código equivalente, que implementa lo que realmente se espera de la ejecución del programa. Aunque seguramente, en un compilador comercial no se obtendría un programa idéntico (código máquina) a nuestro subprograma, si serían equivalentes en cuanto a los resultados de su ejecución.

Con posterioridad presentaremos los resultados de la ejecución del mismo, en PowerDEVS. Nos parece importante resaltar, que la validación de los resultados que obtengamos con la simulación del banco de pruebas en el entorno PowerDEVS, se validara su corrección por comparación con los resultados obtenidos en la simulación del mismo banco de pruebas mediante ModelSim, (recordemos se realizó en el Capítulo 4). Por tanto, indicaremos en cada caso de prueba en qué sección y figura se pueden comprobar el resultado correspondiente de la simulación ModelSim del banco de prueba equivalente. Con el objetivo, que el lector pueda establecer la correspondencia de resultados de ambas simulaciones. Estos valores obtenidos en ambas simulaciones, junto con los resultados que esperamos de la ejecución a mano (o implementando el código en cualquier lenguaje de programación) del pseudocódigo correspondiente, que se muestra a continuación de cada tabla, constituirán el proceso de validación y verificación de resultados.

Entendemos por tanto, que la corrección de los resultados de la implementación PowerDEVS queda demostrada por la comparación de los resultados de la simulación mediante ModelSim y la ejecución del pseudocódigo, que deben arrojar los mismos resultados. De no ser así, se pondría de manifiesto el funcionamiento irregular de la implementación.

Tras finalizar la ejecución, de cada caso de prueba, mostraremos dónde se puede consultar el código completo de este caso de prueba en PowerDEVS, por si se desea verificar el mismo o realizar cambios para comprobar la veracidad de dicha ejecución. Por ejemplo proponemos al lector que se muestre interesado, que cambie los valores de entrada del código máquina y compruebe que los resultados obtenidos, corresponden con la ejecución del pseudocódigo correspondiente, con los nuevos valores de entrada.

6.4 BANCO DE PRUEBAS "BUCLE WHILE Y ALTERNATIVA IF/ELSE"

Este subprograma, realiza la ejecución de un bucle while y de una alternativa if/else, junto con la operación matemática resta. Presentamos a continuación la tabla con el código máquina en Tabla 6.1.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	descripción
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	Loadl en reg 0--a=2;
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	--2
2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	Loadl en reg 1-b=15;
3	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	--15
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	loadl en reg 2--c=7;
5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	--7
6	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	Loadl en reg 3-d=10;
7	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	--10
8	0	0	1	1	0	0	0	0	0	0	0	1	1	0	1	0	Branchl reg2 reg3;
9	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	14
10	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	Sub reg1–reg0b=b-a
11	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	dec reg 3--C--;
12	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	Branchl ;
13	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	--8
14	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Print a
15	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	Print b
16	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	Print c
17	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	Branch not equal
18	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	Dirección de salto
19	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	Print b
20	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	branchl
21	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	Dirección de salto
22	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	Print d
23	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	fin

Tabla 6.1: Tabla de instrucciones código máquina subprograma y su equivalencia pseudocódigo

En Código 6.1 se nos muestra el subprograma equivalente al de la Tabla 6.1.

```
a=2;
b=15;
c=7;
d=10;
while (c<=d){
b-a;
c++;}
Print a;
Print b;
Print c;
If  (a != b){
print b;
}else{
Print d;}
End;
```

Código 6.1: Seudocódigo del caso de prueba bucle “while” y alternativa “if/else”

La ejecución de dicho subprograma, que encontraremos en la ruta E:\PowerDEVS\atomics\Proyecto\memori4.ccp con el nombre “memori4”, del cual aportamos el código escrito en la matriz de la memoria en Código 6.2 y 6.3. Este código se encuentra en la pestaña de “Init” por tanto esto se ejecutara siempre al principio de la ejecución del archivo.

Es fácil comprobar con la tabla anterior, la correspondencia entre códigos, simplemente comprobando las posiciones en las que aparece un 1. Resaltar antes de la ejecución del programa en PowerDEVS, que la instrucción “print”, la hemos añadido nosotros, para facilitar la presentación de resultados en el caso de PowerDEVS.

```
for (int j=0;j<64;j++){
for (int i=0;i<16;i++){
memoria[j][i]=0;
}}
// load en reg 0
memoria[0][2]=1;
// a=2
memoria[1][14]=1;
// load en reg 1
memoria[2][2]=1;
memoria[2][15]=1;
// b=15
memoria[3][12]=1;
memoria[3][13]=1;
memoria[3][14]=1;
memoria[3][15]=1;
// load en reg 2
memoria[4][2]=1;
memoria[4][14]=1;
// c=7
memoria[5][13]=1;
memoria[5][14]=1;
memoria[5][15]=1;
//load en reg 3
memoria[6][2]=1;
memoria[6][14]=1;
memoria[6][15]=1;
// d=10
memoria[7][12]=1;
memoria[7][14]=1;
// BranchI c d
memoria[8][2]=1;
memoria[8][3]=1;
memoria[8][11]=1;
memoria[8][14]=1;
memoria[8][15]=1;
//14
memoria[9][12]=1;
memoria[9][13]=1;
memoria[9][14]=1;
//Sub b-a
memoria[10][1]=1;
memoria[10][2]=1;
memoria[10][3]=1;
memoria[10][15]=1;
//inc d
memoria[11][2]=1;
memoria[11][3]=1;
memoria[11][4]=1;
memoria[11][14]=1;
//BranchI
memoria[12][2]=1;
memoria[12][4]=1;
// 8
```

Código 6.2: Fragmento de código de “memori4”, parte 1

```
// 8
memoria[13][12]=1;

memoria[14][0]=1;
memoria[14][1]=1;
memoria[14][2]=1;

memoria[15][0]=1;
memoria[15][1]=1;
memoria[15][2]=1;
memoria[15][15]=1;

memoria[16][0]=1;
memoria[16][1]=1;
memoria[16][2]=1;
memoria[16][14]=1;
//BRACNH NOT EQUAL
memoria[17][0]=1;
memoria[17][2]=0;
memoria[17][3]=1;
memoria[17][4]=1;
memoria[17][12]=1;
memoria[17][15]=1;
//DIRECCION DEL SALTO
memoria[18][11]=1;
memoria[18][13]=1;
memoria[18][14]=1;
//PRINT
memoria[19][0]=1;
memoria[19][1]=1;
memoria[19][2]=1;
memoria[19][14]=0;
memoria[19][15]=1;
//BRANCHI
memoria[20][2]=1;
memoria[20][4]=1;
//DIRECCION
memoria[21][11]=1;
memoria[21][13]=1;
memoria[21][14]=1;
memoria[21][15]=1;

//PRINT
memoria[22][0]=1;
memoria[22][1]=1;
memoria[22][2]=1;
memoria[22][14]=1;
memoria[22][15]=1;
// FIN

memoria[23][0]=1;
memoria[23][1]=1;
memoria[23][2]=1;
memoria[23][4]=1;
}
```

Código 6.3: Fragmento de código de “memori4”, parte 2

En la Figura 6.12, se nos muestran los resultados de la simulación en PowerDEVS. Recordar que en cada casilla el valor de la izquierda, corresponde al tiempo de simulación, mientras que el valor que aparece a la derecha de la coma corresponde al valor de salida del circuito. Por tanto, vemos que se realiza la inicialización del circuito en el instante 0 y que el instante 199 la instrucción “print a” (Tabla 6.1, fila 14) produce el valor 2; en el instante 205,5 la instrucción “print

b" (Tabla 6.1, fila 15) produce el valor 7, en el instante 212 la instrucción "print c" (Tabla 6.1, fila 16) produce el valor 11, y en el instante 230 la instrucción "print b" (Tabla 6.1, fila 20) produce el valor 7. Recordemos que existe otra instrucción "print d" (Tabla 6.1, fila 22), pero ésta no se ejecuta porque no procede la alternativa "else" y por tanto no se ejecuta.

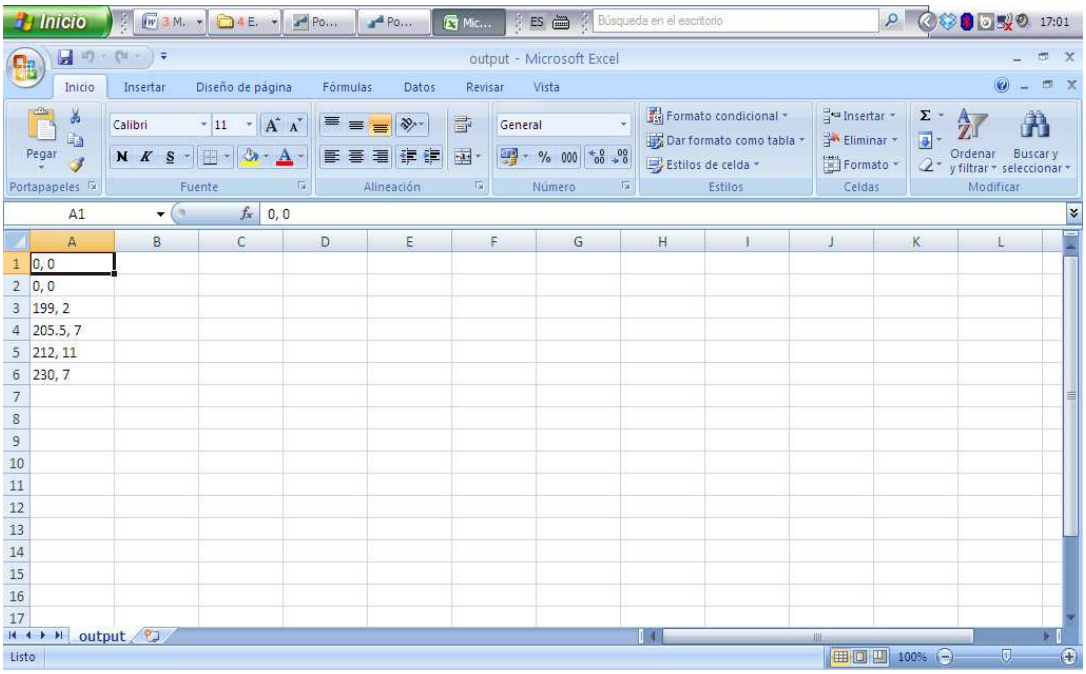


Figura 6.12: Resultados ejecución subprograma en PowerDEVS

Al inicio de este capítulo, hemos comentado la estructura que vamos a seguir para la validación de la implementación en "PowerDEVS".

Debemos recordar que la simulación del banco de pruebas de VHDL, han sido realizadas y comprobadas en el Capítulo 4. En dicho capítulo se ha realizado la simulación ModelSim del banco de pruebas VHDL siguiendo los mismos criterios que hemos hecho con PowerDEVS, hasta este momento. Es decir, primero hemos mostrado la tabla del código máquina correspondiente al circuito, para a continuación mostrar el subprograma equivalente en pseudocódigo. A

continuación se muestra el código que implementa dicho subprograma en VHDL, para finalizar mostrando el resultado de la simulación, de dicho banco de pruebas.

La comparación de resultados podrá establecerse con los resultados obtenidos en la Sección 4.7, Figura 4.13 y en los párrafos anexos a dicha figura, que es donde se explican los resultados de la simulación ModelSim.

Como se aprecia en la Figura 4.13 (pág. 162), los valores obtenidos al final de la ejecución, en los registros de 0 a 2, es idéntico a los mostrados en la imagen del archivo de salida “out.exe”(Figura 6.12). Cabe explicar, que en la versión PowerDEVS aparece impreso el valor de b que es 7, en dos ocasiones, esto es consecuencia del bucle if que nos dice: si a!= b, print b. En el caso VHDL, se ha añadido para comprobación del else, la siguiente línea de código ("0101000000000001", --- 12 or reg 0 reg 1) Código 4.2 (pág. 145), cosa que no sucede a tenor de los resultados, pues la alternativa else no procede con estos datos, recordemos que no se ha implementado la instrucción “print” para el circuito VHDL.

6.5 BANCO DE PRUEBAS “BUCLE INTERIOR A UN BUCLE”

En este caso, demostrado el funcionamiento del bucle y la alternativa if, pasamos a presentar un caso de prueba, en la que se ejecuta un bucle “for” en el interior de un bucle “while”. En la Tabla 6.1, se presenta explicado el código máquina que se va a ejecutar.

En Código 6.5 se muestra el pseudocódigo equivalente al código máquina que se nos muestra en la Tabla 6.2, es decir la ejecución de ambos códigos debe

de arrojar el mismo resultado. Esto se comprobará mediante la simple ejecución mental del pseudocódigo, pues se ha intentado realizar un banco de pruebas de fácil seguimiento.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	descripción
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	LoadI en reg 0 -- a=2;
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	--2
2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	LoadI en reg 1 -b=15;
3	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	--15
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	loadI en reg 2--c=7;
5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	--7
6	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	LoadI en reg 3--d=10;
7	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	--10
8	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	LoadI en reg 4--f=2;
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	--2
10	1	0	1	1	1	0	0	0	0	0	0	1	0	0	0	1	branchI si equal
11	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	24
12	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	d=d-a;
13	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	loadI --f=0,
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	1	0	1	1	1	0	0	0	0	0	1	0	1	1	0	0	BranchI if equal --(for
16	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	21
17	0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1	Add 3 0--d=d+a;
18	0	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	Inc 4--f++;
19	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	branchI
20	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	15
21	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	Inc 2--c++;
22	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	branchI
23	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	10
24	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	Print d
25	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	End.
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
27																	
28																	

Tabla 6.2: Tabla de instrucciones código máquina subprograma y su equivalencia pseudocódigo

```
Programa equivalente en lenguaje de alto nivel .
a= 2;
b=15;
c= 7;
d=10;
while (c!= d){
d= d-a;
for (f=0; f<2;i++){
d= d+a;}
c++;
}
Print d;
End;
```

Código 6.4: pseudocódigo “Bucle interior a un bucle”

El código del subprograma a ejecutar en nuestro circuito, en nuestro caso se

encuentra en E:\PowerDEVS\atomics\Proyecto\memori5.cpp. Como hemos comentado con anterioridad al respecto de la instrucción “print”, aplica en los cuatro casos de prueba.

```
for (int j=0;j<64;j++){
for (int i=0;i<16;i++){
memoria[j][i]=0;
}}
// load en reg 0
memoria[0][2]=1;
// a=2
memoria[1][14]=1;
// load en reg 1
memoria[2][2]=1;
memoria[2][15]=1;
// b=15
memoria[3][12]=1;
memoria[3][13]=1;
memoria[3][14]=1;
memoria[3][15]=1;
// load en reg 2
memoria[4][2]=1;
memoria[4][14]=1;
// c=7
memoria[5][13]=1;
memoria[5][14]=1;
memoria[5][15]=1;
//load en reg 3
//load en reg 3
memoria[6][2]=1;
memoria[6][14]=1;
memoria[6][15]=1;
// d=10
memoria[7][12]=1;
memoria[7][14]=1;
//load en reg 4
memoria[8][2]=1;
memoria[8][13]=1;
//2
memoria[9][14]=1;
//branchI reg 2---1
memoria[10][0]=1;
memoria[10][2]=1;
memoria[10][3]=1;
memoria[10][4]=1;
memoria[10][11]=1;
memoria[10][15]=1;
//24
memoria[11][11]=1;
memoria[11][12]=1;
// d=d-a;
memoria[12][1]=1;
memoria[12][2]=1;
memoria[12][3]=1;
memoria[12][14]=1;
memoria[12][15]=1;
// loadI
memoria[13][2]=1;
memoria[13][13]=1;
memoria[13][15]=1;
//0
//branchI if equal
memoria[15][0]=1;
memoria[15][2]=1;
memoria[15][3]=1;
memoria[15][4]=1;
memoria[15][10]=1;
memoria[15][12]=1;
memoria[15][13]=1;
```

Código 6.5: Fragmento de código del archivo “memori5” (parte 1)

```
// 21
memoria[16][11]=1;
memoria[16][13]=1;
memoria[16][15]=1;
//print d
memoria[17][1]=1;
memoria[17][2]=1;
memoria[17][4]=1;
memoria[17][14]=1;
memoria[17][15]=1;
//f++
memoria[18][2]=1;
memoria[18][3]=1;
memoria[18][4]=1;
memoria[18][13]=1;
memoria[18][15]=1;
//branchI
memoria[19][2]=1;
memoria[19][4]=1;
//15
memoria[20][12]=1;
memoria[20][13]=1;
memoria[20][14]=1;
memoria[20][15]=1;
//inc c
memoria[21][2]=1;
memoria[21][3]=1;
memoria[21][4]=1;
memoria[21][14]=1;
//branchI
memoria[22][2]=1;
memoria[22][4]=1;
//10
memoria[23][12]=1;
// print d
memoria[24][0]=1;
memoria[24][1]=1;
memoria[24][2]=1;
memoria[24][14]=1;
memoria[24][15]=1;
//end
memoria[25][0]=1;
memoria[25][1]=1;
memoria[25][2]=1;
memoria[25][3]=1;
memoria[25][5]=1;}
```

Código 6.6: Fragmento de código del archivo “memori5” (parte 2)

Como vemos en Código 6.5 y 6.6, la disposición de la memoria es igual que en el caso anterior y que el resto de las implementaciones del banco de prueba. La disposición de la memoria es la misma en todos los casos (solo se implementan las posiciones de valor “1”) y por tanto no procede mayor explicación.

Los resultados obtenidos de la ejecución del archivo, se encuentran en la ruta donde tengamos instalado PowerDEVS, raíz…\PowerDEVS\output y que se muestran en la Figura 6.13. En la fila 3 se muestra el valor del tiempo de

simulación 1179,5 y el valor de salida “26” corresponde con la instrucción “print d” Tabla 6.2 línea 24.

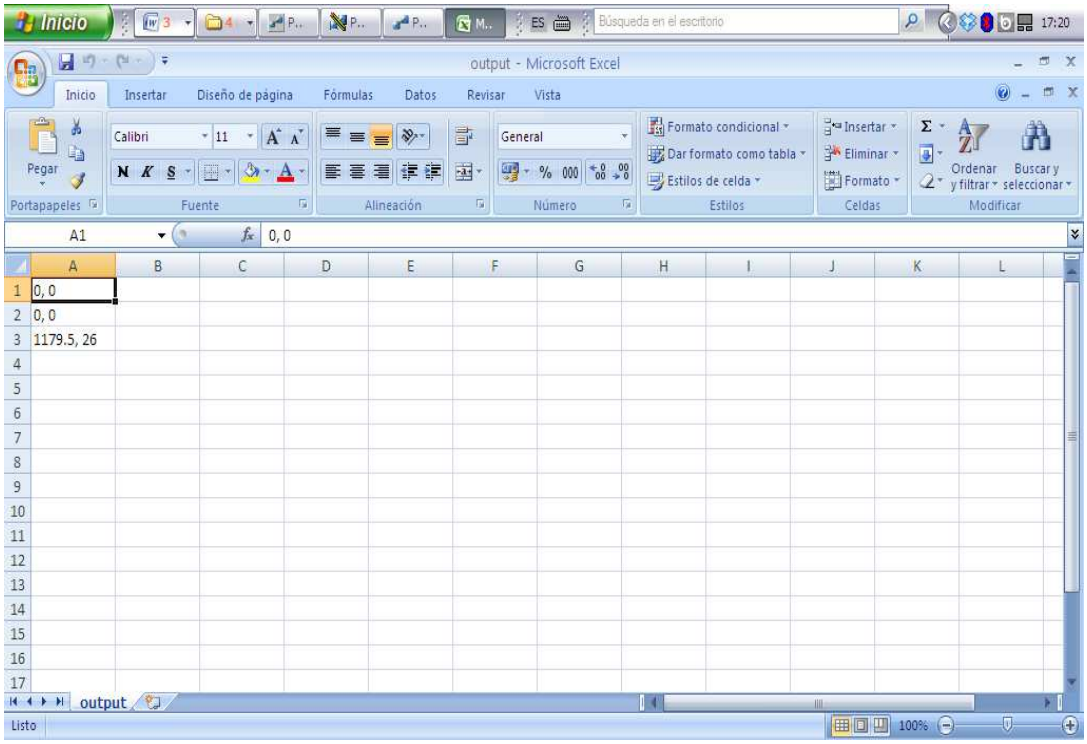


Figura 6.13: Resultados ejecución subprograma en PowerDEVS

La ejecución del subprograma equivalente en VHDL que encontraremos en la Sección 4.7 Figura 4.14 (pág. 163) arroja los mismos resultados tal y como se explica en los párrafos anexos a dicha figura.

Como se puede apreciar en la Figura 4.14, el resultado se encuentra en el registro nº 3 y arroja un valor 26, que es el resultado obtenido en la simulación en PowerDEVS.

Por tanto ambos resultados son iguales y concuerdan con la ejecución a mano del código mostrado en Código 6.4.

6.6 BANCO DE PRUEBAS "COPIAR DE MEMORIA A MEMORIA"

En este caso de prueba, vamos a realizar la copia de una parte de la memoria en otra parte de la memoria.

Este subprograma está descrito en el texto base, aunque aquí se presenta con algunas diferencias. En nuestro caso las posiciones de memoria no se pueden comprobar directamente como en el caso de ModelSim y por tanto hemos añadido varias instrucciones, cuya función es imprimir el valor existente de las posiciones que se supone, se han acabado de copiar.

En ambos casos, tanto en el banco de pruebas de VHDL como en el de PowerDEVS, lo que se realiza es la copia de un sector de datos de la memoria, en otro sector de la memoria.

Para demostrar que se ha realizado la copia, en el caso PowerDEVS, procedemos una vez copiados en el nuevo sector de la memoria a imprimir el valor que se ha almacenado en la nueva posición de memoria, aprovechando el bucle de copia (líneas 10 y 13 de la Tabla 6.3). Presentamos a continuación el código objeto que se ejecuta en el subprograma.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	descripción
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	LoadI --a=32;
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	32
2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	LoadI 2-- b=16;
3	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	16
4	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	LoadI 6-- c=36;
5	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	36
6	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	Load 2 – 4;
7	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	1	Store 4 1
8	0	0	1	1	0	0	0	0	0	0	0	1	0	1	1	0	BGT 2 6
9	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	56
10	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	1	Load 1 –5
11	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	Inc 2 --b++;
12	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	Inc 1 -- a++;
13	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	Print 5
14	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	Branch
15	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	6
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	ELEMENTOS A COPIA
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	“
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	“
19	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	“
20	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	“
21	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	“
22	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	“
23	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	“
24	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	“
25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	“
26	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	“
27	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	“
28	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	“
29	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	“
30	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	“
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	“

Tabla 6.3: Tabla donde se presenta el código máquina que se ejecuta en el subprograma

```
PROGRAMA QUE COPIA DE UNA POSICION DE MEMORIA A OTRA.
a = 32;
b = 16;
c = 36;
while ( c>b){
Load b, d; // (1)
Store d, a;// (2)
Load a, f;// (1)
a++;
b++;
print f;}
```

Código 6.7: pseudocódigo “Copiar memoria a memoria”

En Código 6.7 se nos muestra el pseudocódigo en lenguaje de alto nivel, que correspondería al código objeto que se presenta en Tabla 6.3. Entendemos que

debemos explicar las instrucciones marcadas como (1) y (2). No es habitual que los lenguajes de alto nivel indiquen en qué posiciones de memoria principal se debe guardar los datos, esto se hace en el código objeto con direcciones relativas y el sistema operativo, es el que dirá en qué posición guarda esos datos en la memoria principal.

Por tanto no tenemos instrucciones de alto nivel que especifiquen esas direcciones, con lo cual, nosotros hemos optado por utilizar las instrucciones de ensamblador que realizaran esas funciones entre la memoria principal y el banco de registros, para la especificación del pseudocódigo.

El código que ejecuta este supuesto, es el que encuentra en la ruta E:\PowerDEVS\atomics\Proyecto\memori3.cpp, donde encontraremos el código completo. Limitándonos a continuación a presentar en Código 6.8 y 6.9, la estructura de la memoria, con la implementación del código propuesto en la tabla anterior.

En este caso, no es posible visualizar el valor de las posiciones de la matriz de memoria, cosa que si sucede en el caso de ModelSim, en que tenemos a nuestra disposición el valor que van tomando todas las variables en el instante que deseemos.

Como hemos indicado anteriormente aprovechamos el bucle de copia para volver a cargar el último valor copiado en la memoria, en el registro nº 5 de “Regarray”, de donde realizamos la impresión en el registro de salida con la instrucción print.

```
for (int j=0;j<64;j++){
for (int i=0;i<16;i++){
memoria[j][i]=0;
}}
//loadI 1
memoria[0][2] =1;
memoria[0][15] =1;
// 32
memoria[1][10] =1;
//loadI 2
memoria[2][2] =1;
memoria[2][14] =1;
// 16
memoria[3][11] =1;
 //load 6
memoria[4][2] =1;
memoria[4][13] =1;
memoria[4][14] =1;
//36
memoria[5][10] =1;
memoria[5][13] =1;
//load
memoria[6][4] =1;
memoria[6][11] =1;
memoria[6][13] =1;
//store
memoria[7][3] =1;
memoria[7][10] =1;
memoria[7][15] =1;
//bgt
//bgt
memoria[8][2] =1;
memoria[8][3] =1;
memoria[8][11] =1;
memoria[8][13] =1;
memoria[8][14] =1;
//
memoria[9][10] =1;
memoria[9][11] =1;
memoria[9][12] =1;
//load
memoria[10][4] =1;
memoria[10][12] =1;
memoria[10][13] =1;
memoria[10][15] =1;
//inc
memoria[11][2] =1;
memoria[11][3] =1;
memoria[11][4] =1;
memoria[11][14] =1;
//inc
memoria[12][2] =1;
memoria[12][3] =1;
memoria[12][4] =1;
memoria[12][15] =1;
//print
memoria[13][0] =1;
memoria[13][1] =1;
memoria[13][2] =1;
memoria[13][13] =1;
memoria[13][15] =1;
// branch
memoria[14][2] =1;
memoria[14][4] =1;
//6
memoria[15][13] =1;
memoria[15][14] =1;
```

Código 6.8: Fragmento de código de “memori3.cpp”, parte 1

```
//elementos a copiar
memoria[16][15] =1;
memoria[17][14] =1;
memoria[18][14] =1;
memoria[18][15] =1;
memoria[19][13] =1;
memoria[20][13] =1;
memoria[20][15] =1;
memoria[21][13] =1;
memoria[21][14] =1;
memoria[22][13] =1;
memoria[22][14] =1;
memoria[22][15] =1;
memoria[23][12] =1;
memoria[24][12] =1;
memoria[24][15] =1;
memoria[25][12] =1;
memoria[25][14] =1;
memoria[26][12] =1;
memoria[26][14] =1;
memoria[26][15] =1;
memoria[27][12] =1;
memoria[27][13] =1;
memoria[28][12] =1;
memoria[28][13] =1;
memoria[28][15] =1;
memoria[29][12] =1;
memoria[29][13] =1;
memoria[29][14] =1;
memoria[30][12] =1;
memoria[30][13] =1;
memoria[30][14] =1;
memoria[30][15] =1;
}
```

Código 6.9: Fragmento de código de “memori3.cpp”, parte 2

La ejecución del archivo, que también podremos encontrar en el Anexo B como “memori3”, arroja los resultados que se muestran en Figura 6.4. Como podemos apreciar lo que hacemos es imprimir los valores copiados en las nuevas posiciones de memoria.

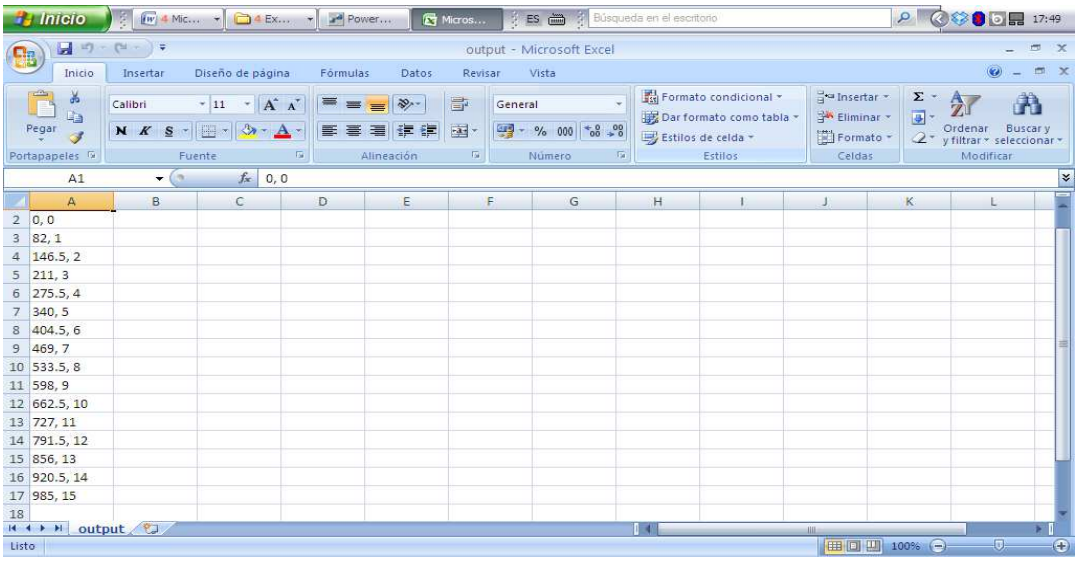


Figura 6.14: Imagen de resultados ejecución subprograma en PowerDEVS

En la ejecución del programa en ModelSim, cuyo código presenta las discrepancias explicadas con anterioridad. Las instrucciones de las posiciones 10 y 13 de la Tabla 6.3, no se han implementado pues no son necesarias en absoluto por la particularidad del entorno ModelSim tal y como hemos comentado anterioridad. Pues podemos consultar el valor, que contienen las posiciones de memoria.

Por tanto los resultados que aparecen en la Figura 6.14 son los datos que se han copiado a otra posición de la memoria, pero aprovechando el bucle de copia. Una vez un dato se ha copiado en su nueva posición de memoria este dato se vuelve a cargar en el registro 5 (línea 10 de Tabla 6.3) y se procede a imprimirlo (línea 13 de la Tabla 6.3).

Para la comparación de resultados, con la ejecución del mismo banco de pruebas en VHDL. Los resultados obtenidos de la simulación del programa equivalente (archivo “mem.vhd” arquitectura “copiarmem”) para VHDL, se presentan, en varias imágenes el valor de las posiciones de memoria al inicio del subprograma Figura 4.15 (pág. 164), como comprobación de que no se encuentran escritas, es decir con todo sus valores a cero. Con posterioridad a la ejecución del código, presentaremos el valor de las mismas posiciones, con los valores finales después de la ejecución del mismo, Figuras 4.16, y 4.17 (pág. 165). Inicialmente las posiciones de memoria referidas estaban a 0.

Al final de la ejecución del subprograma, se obtienen los valores que se deseaban copiar en dichas posiciones.

6.7 BANCO DE PRUEBAS "OPERACIONES MATEMÁTICAS"

En este caso de prueba, se va a realizar la prueba de todas las operaciones matemáticas y lógicas que se han implementado, como demostración del correcto funcionamiento de las instrucciones, que actúan sobre la ALU.

Se presenta a continuación en Tabla 6.4, el código máquina del subprograma a ejecutar, con su correspondiente traslación a ensamblador.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	descripción
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	LoadI reg 0 – a=178
1	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	0	--178
2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	LoadI reg 1 –b=409
3	0	0	0	0	0	0	0	1	1	0	0	1	1	0	0	1	--409
4	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 a reg 2
5	0	1	1	0	1	0	0	0	0	0	0	0	1	0	1	0	add 2 1--c=b+a;
6	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	Print c;
7	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 a reg 2
8	0	1	0	0	1	0	0	0	0	0	0	0	1	0	1	0	and 2 1 --c=a and b
9	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	Print c;
10	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 a reg 2
11	0	1	0	1	0	0	0	0	0	0	0	0	1	0	1	0	or 2 1 --c=a or b;
12	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	Print c;
13	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 a reg 2
14	0	1	0	1	1	0	0	0	0	0	0	0	1	0	1	0	Xor 2 1 --c= a xor b;
15	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	Print c;
16	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 a 2- c=a
17	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	Shift left c--c<<
18	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	Print c;
19	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 a 2 c=a
20	1	1	0	1	1	0	0	0	0	0	0	0	1	0	1	0	Shift right c--c>>
21	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	Print c;
22	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	Move reg 0 a reg 2
23	0	1	1	1	0	0	0	0	0	0	0	0	1	0	1	0	Sub 2 1 -- c=b-a;
24	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	Print c;
25	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	END
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
27																	
28																	

Tabla 6.4: Tabla de instrucciones código máquina subprograma y su equivalencia pseudocódigo

A continuación en Código 6.11 mostramos el pseudocódigo que correspondería al código máquina de la Tabla 6.4. Como se puede apreciar, éste consiste en la ejecución de una operación matemática o lógica y la impresión del resultado de la misma.

Esto se hace con todas las instrucciones implementadas en la ALU y en el registro de desplazamiento.

```
OPERACIONES MATEMATICAS
a= 178;
b=409;
c= a + b;
print c;
c= a and b;
print c;
c= a or b;
print c;
c= a xor b;
print c;
c= b - a;
print c;
c= a >>;
print c;
c= <<a ;
print c;
end;
```

Código 6.10: Seudocódigo de "operaciones matemáticas"

El código de este subprograma, lo encontraremos en la ruta E:\PowerDEVS\atomics\Proyecto\memori6.ccp. En Código 6.12, 6.13 y 6.14, se presenta extractado, el contenido de las posiciones de la memoria donde se incluye el subprograma, pues el resto del archivo es igual en todos los casos de prueba y la parte más importante es por tanto, el contenido de la matriz que constituye la memoria.

```
for (int j=0;j<64;j++){
for (int i=0;i<16;i++){
memoria[j][i]=0;
}}
// load en reg 0
memoria[0][2]=1;
// a=178
memoria[1][8]=1;
memoria[1][10]=1;
memoria[1][11]=1;
memoria[1][14]=1;
// load en reg 1
memoria[2][2]=1;
memoria[2][15]=1;
// b=409
memoria[3][7]=1;
memoria[3][8]=1;
memoria[3][11]=1;
memoria[3][12]=1;
memoria[3][15]=1;
// move reg 0 a reg 2
memoria[4][3]=1;
memoria[4][4]=1;
memoria[4][14]=1;
// c=b+a
```

Código 6.11: Fragmento de código "memori6" (parte 1)

```
// c=b+a
memoria[5][1]=1;
memoria[5][2]=1;
memoria[5][4]=1;
memoria[5][12]=1;
memoria[5][14]=1;
//print c
memoria[6][0]=1;
memoria[6][1]=1;
memoria[6][2]=1;
memoria[6][14]=1;
// move reg 0 a reg 2
memoria[7][3]=1;
memoria[7][4]=1;
memoria[7][14]=1;
// c= a and b;
memoria[8][1]=1;
memoria[8][4]=1;
memoria[8][12]=1;
memoria[8][14]=1;
//print c
memoria[9][0]=1;
memoria[9][1]=1;
memoria[9][2]=1;
memoria[9][14]=1;
// move reg 0 a reg 2
memoria[10][3]=1;
memoria[10][4]=1;
memoria[10][14]=1;
// c= a 0r b;
memoria[11][1]=1;
memoria[11][3]=1;
memoria[11][12]=1;
memoria[11][14]=1;
//print c
memoria[12][0]=1;
memoria[12][1]=1;
memoria[12][2]=1;
memoria[12][14]=1;
// move reg 0 a reg 2
memoria[13][3]=1;
memoria[13][4]=1;
memoria[13][14]=1;
// c= a xor b;
memoria[14][1]=1;
memoria[14][3]=1;
memoria[14][4]=1;
memoria[14][12]=1;
memoria[14][14]=1;
//print c
memoria[15][0]=1;
memoria[15][1]=1;
memoria[15][2]=1;
memoria[15][14]=1;
// move reg 0 a reg 2
memoria[16][3]=1;
memoria[16][4]=1;
memoria[16][14]=1;
// c= shift left a;
memoria[17][0]=1;
memoria[17][1]=1;
memoria[17][3]=1;
memoria[17][14]=1;
//print c
memoria[18][0]=1;
memoria[18][1]=1;
memoria[18][2]=1;
memoria[18][14]=1;
// move reg 0 a reg 2
memoria[19][3]=1;
memoria[19][4]=1;
memoria[19][14]=1;
// c= shift right a;
```

Código 6.12: Fragmento de código “memori6” (parte 2)

```
// c= shift right a;
memoria[20][0]=1;
memoria[20][1]=1;
memoria[20][3]=1;
memoria[20][4]=1;
memoria[20][14]=1;
//print c
memoria[21][0]=1;
memoria[21][1]=1;
memoria[21][2]=1;
memoria[21][14]=1;
// move reg 0 a reg 2
memoria[22][3]=1;
memoria[22][4]=1;
memoria[22][12]=1;
memoria[22][14]=1;
// c=b-a;
memoria[23][1]=1;
memoria[23][2]=1;
memoria[23][3]=1;
memoria[23][14]=1;
//print c
memoria[24][0]=1;
memoria[24][1]=1;
memoria[24][2]=1;
memoria[24][14]=1;
//end
memoria[25][0]=1;
memoria[25][1]=1;
memoria[25][2]=1;
memoria[25][3]=1;
memoria[25][4]=1;
}
```

Código 6.13: Fragmento del archivo “memori6” (parte 3)

En este caso, pretendemos probar el funcionamiento de las instrucciones que trabajan sobre operaciones de la ALU. Se presentan funcionando todas en un mismo programa, con instrucciones que son todas del mismo tipo. Recordamos que en los programas anteriores ya se han realizado operaciones de suma, resta e incremento en unión de otras operaciones de carga de registro, copia etc. El objetivo, entendemos queda cubierto con las principales operaciones de la ALU, pues estas instrucciones son todas iguales en cuanto su ejecución (solo difieren las unarias de las binarias). Siendo su diferencia principal, la operación que realizan. En este caso no es relevante, pues la ALU, funciona mediante conversión a decimal y realiza las operaciones con operaciones en código c.

La ejecución del archivo que podemos encontrar en el anexo con el nombre “memori6” arroja los resultados que se muestran en Figura 6.15.

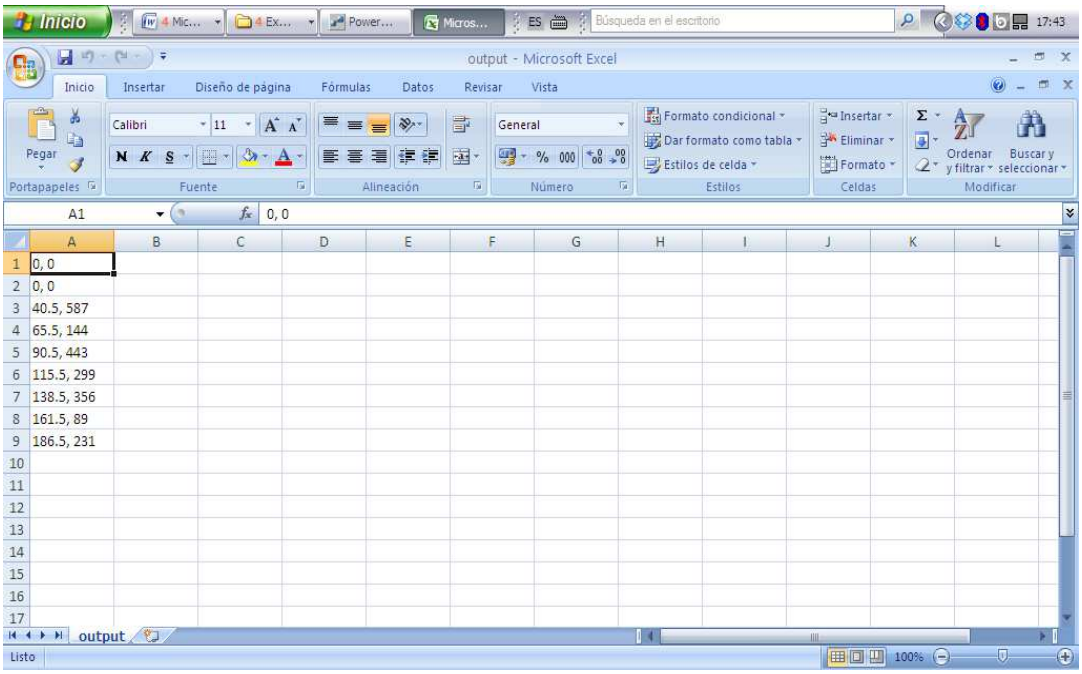


Figura 6.15: Imagen de los resultados ejecución subprograma en PowerDEVS

Entendemos que estos resultados no requieren de muchos comentarios, pues la comprobación de su corrección, podría realizarse simplemente calculadora en mano. En este caso seguiremos el procedimiento habitual que consiste en comparar estos resultados, con los que obtenemos en la simulación mediante ModelSim, entendiendo de esta manera que si ambos son conformes en valor, el circuito se comporta de manera correcta.

Los resultados que obtenemos al realizar la simulación del subprograma equivalente, que se encuentra en el archivo “mem.vhd” y en la arquitectura “operbasicas”, los encontraremos en la Figuras 4.18 a 4.24 (págs. de 166 a 170) en la Sección 4.7. Recordamos que realizamos la simulación en decimal, para realizar la comparación con los resultados obtenidos mediante PowerDEVS, (datos decimales). En los párrafos anexos a dichas figuras, encontraremos la explicación de los resultados.

El resultado, en este caso de la suma, corresponde con el valor 587 Figura 4.18, que como observamos es el valor obtenido en la casilla 3 del archivo "output" (Figura 6.15), continuamos mostrando los resultados de cada operación.

En la Figura 4.19, se nos muestra en el registro nº 2 el resultado que corresponde con la operación "AND" que arroja el resultado "144", corresponde con casilla nº 4 de Figura 6.15.

El resultado obtenido para la operación "OR", es el valor 443 Figura 4.20, que es el valor obtenido en la casilla 5 del archivo "output" Figura 6.15

El siguiente resultado corresponde con la operación "XOR", que arroja el siguiente resultado que se nos muestra en la Figura 4.21 registro nº 2 y que tiene el valor "299".El resultado obtenido, se corresponde con el valor obtenido en la casilla 6 del archivo "output" Figura 6.15.

La siguiente operación, que vamos a analizar es "shifleft" (desplazamiento a izquierda), que arroja el resultado que se nos muestra en la Figura 4.22, registro nº 2.El resultado que obtenemos es el valor 356 que coincide con el valor obtenido en la casilla nº 7 del archivo "output", Figura 6.15.

La siguiente operación, es la denominada "SHR" (desplazamiento a derecha) que arroja el resultado que se nos muestra en la Figura 4.23, registro nº 2. El valor obtenido corresponde con el valor 89, que es coincidente con el valor obtenido en la casilla 8 del archivo "output" Figura 6.15.

Por último, presentamos el resultado obtenido en la operación "resta" que arroja el resultado que se muestra en la Figura 4.24 registro nº 1.El valor obtenido

corresponde con el valor 231 que coincide con el valor obtenido en la casilla nº 9 del archivo “output” Figura 6.15.

6.4 CONCLUSIONES

Como se puede comprobar del resultado de las pruebas de validación, éstas arrojan los mismos resultados, en la ejecución de los banco de pruebas de ambos entornos (PowerDEVS y ModelSim). Esto en sí mismo no constituye ninguna prueba de nada, pues es obvio, que se podrían inducir los resultados para que así resultase. De otra manera se puede realizar la ejecución a mano del código correspondiente y anticipar los resultados esperados.

Tenemos por tanto que resaltar, que las pruebas de validación se corresponden con los resultados esperados de la ejecución del código de alto nivel de los subprogramas. Con lo cual se establece una primera prueba de validación de los resultados obtenidos; esta validación, podría ser que hiciese superflua, el volver a realizar una validación con los resultados obtenido en el validación con código VHDL, pero a nuestro entender constituyen las “patas de una misma mesa”, es decir no son redundantes. Pues la ejecución del código a mano, no implica ninguna comparación, solo la validez de los resultados.

Los resultados de la comparación de las distintas ejecuciones del “banco de pruebas” según su implementación, si permiten una comparación de los mismos, que aunque deben arrojar resultados iguales, sí permiten establecer diferencias en cuanto, a la sencillez de la presentación de resultados, facilidad de las pruebas,

cronología de los mismo, etc. y es lo que al fin y al cabo permitirá en el siguiente capítulo el establecer comparaciones y por tanto áreas de mejora.

La validez de los resultado tal y como se ha explicado con anterioridad, queda un tanto establecida, porque los resultados sean coherentes con la ejecución a mano de los distintos subprogramas, y con el hecho que en la ejecución del banco de pruebas en códigos diferentes (VHDL, C++) los resultados concuerdan a su vez, con las distintas ejecuciones del mismo subprograma.

La implementación PowerDEVS del circuito, con la simulación del “banco de pruebas”, constituyen la prueba de la validez del circuito definido mediante el formalismo DEVS, que por otra parte no es mas, que otra fórmula para definir implementaciones de circuitos reales, que como se ha demostrado, se pueden definir de distintas maneras, siendo todas correctas en su formulación, si bien presentan características diferentes en cuanto a sencillez, idoneidad, facilidad, etc.

7

PLANIFICACIÓN Y COSTES DEL PROYECTO

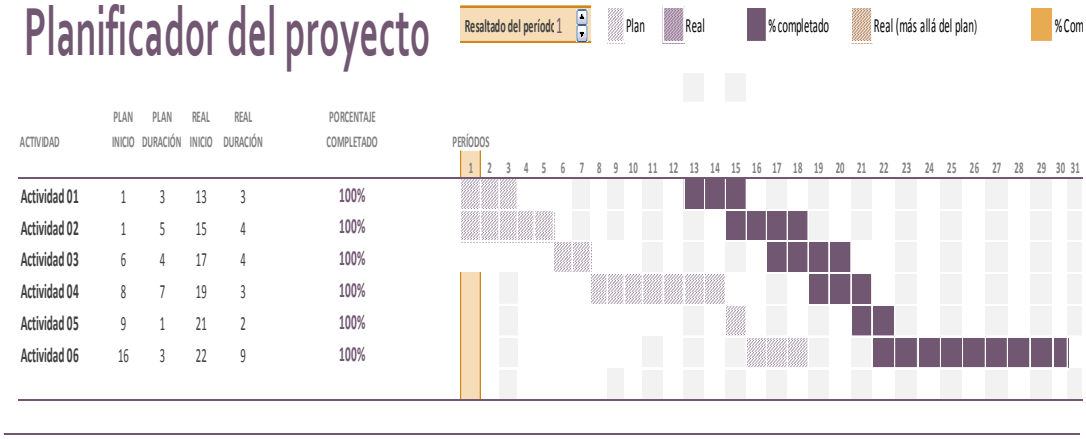
7.1 PLANIFICACIÓN

En esta sección, trataremos de mostrar la planificación seguida en el proyecto, así como el desarrollo del mismo, mostrando sus discrepancias y explicando en la medida de lo posible las divergencias de la planificación, al desarrollo final seguido.

Entendemos que el carácter académico del presente PFC, así como la escasa disponibilidad de medios humanos (carácter unipersonal del PFC), condiciona en gran manera, tanto la planificación del mismo, así como el cumplimiento de los hitos de la planificación. Esta característica, aunque dificulta en gran manera la planificación y cumplimiento de la misma, no es motivo suficiente para que ésta no exista y no se trate de cumplir en la medida de lo posible.

Pasamos por tanto, a mostrar la planificación inicial presentada en el anteproyecto, la cual si bien no nos ha sido posible seguir de la manera que nos hubiera gustado, en cuanto a la referencia temporal del mismo (cumplimientos de hitos y plazos), sí se ha seguido escrupulosamente en cuanto a lo que se refiere al cumplimiento de hitos y orden de tareas del mismo.

Como se puede observar en la Figura 7.1 existe un desfase temporal de un año entre la planificación recogida en el anteproyecto y la real. Este desfase está motivado por necesidades académicas de tiempo, que nos obligó a retrasar el inicio del PFC, por dicho periodo y es por tanto la explicación a dicho desfase.



Actividad 01	Aprender a manejar el entorno de modelado PowerDEVS
Actividad 02	Describir uno o varios bancos de prueba para el circuito propuesto
Actividad 03	Codificar y simular los modelos del banco de pruebas en PowerDEVS
Actividad 04	Describir el circuito y los bancos de prueba mediante vhdl y simularlos usando ModelSimPEStudentEdition
Actividad 05	Validar el modelo comparando los resultados obtenidos mediante PowerDEVS , con los resultados obtenidos usando ModelSim
Actividad 06	Redaccion de la memoria, presentación y demas documentos técnicos.

INICIO 01.10.2011- FIN 31.03.2014 : cada periodo comprende a un mes natural

Figura 7.1: Imagen de la tabla de Gantt con la planificación y el desarrollo real del proyecto

7.2 COSTES DEL PFC

A la hora de estimar los costes materiales del presente PFC, debemos tener muy encuentra el carácter académico del mismo, pues esto condiciona en gran manera los resultados del análisis de costes.

Para realizar el análisis de los costes, pasaremos a explicar los mismos, separando lo que consideramos costes en material y lo que consideramos costes en mano de obra, que sin duda son los más importantes. Nos gustaría resaltar, que

los resultados obtenidos constituyen una mera aproximación a la realidad, pues es muy difícil valorar los costes en mano de obra así como los costes en materiales o bienes de equipos. Al contrario que en una empresa, en nuestro caso no disponemos de contabilidad oficial (tanto financiera, como de costes).

Costes Materiales y Bienes de equipo: Como se ha explicado con anterioridad, a continuación describiremos una aproximación de costes materiales del PFC, siendo éstos una mera especulación, de unos valores aproximados que pueden divergir de la realidad.

Amortización Bienes de equipo ……100

Gastos electricidad y consumibles……50

Total……150 €.

Procederemos a explicar ambas partidas de gastos, en "amortización de bienes de equipo" hemos hecho una estimación del valor de un ordenador junto con sus periféricos de un tipo medio. Estimando un valor inicial de unos 800 €, estimando una vida útil de 4 años, teniendo en cuenta la duración real del proyecto de unos 17 meses, dado que el ordenador también se ha utilizado paralelamente para otros fines ajenos al PFC. Estimamos un coeficiente de utilización de 0,3, arrojaría un valor aproximado de 100 €.

En la partida de gastos de electricidad, entendemos dicho gasto como una estimación del alumbrado necesario para la realización de proyecto; tanto para la alimentación propia del ordenador, así como la iluminación necesaria para el

personal. Los consumibles, nos referimos a la estimación del coste de impresión del PFC así como de los CD y gastos de envío del mismo.

Apuntar simplemente, que no se ha pretendido ser exhaustivos en esta valoración, pues de lo contrario se deberían haber tenido en cuenta costos adicionales como podría ser los relativos al local o lugar de desarrollo, calefacción etc.

En cuanto a los programas utilizados, estos son gratuitos y la documentación adicional o bien, se ha encontrado en los fondos bibliográficos de la Uned o en Internet, con lo cual no ha sido necesario desembolso alguno en este capítulo.

Costes de personal: Si la valoración de los costes materiales ha resultado un tanto complicada, la valoración de los costes en personal, lo es mucho más. La dificultad, no viene dada tanto por la cantidad de horas hombre utilizada, lo cual es bastante fácil de controlar, sino viene determinada por la calidad de las mismas.

Entendemos que la característica académica del PFC, así como el hecho que el autor tenga un trabajo a tiempo completo, hace que el tiempo invertido en el desarrollo del mismo sea de una calidad variable. Entendemos que el rendimiento/ hora durante la jornada laboral es superior, que el obtenido después de haber realizado ya la misma. Esto unido a que los periodos invertidos generalmente no son continuos, ni de una duración apropiada, hacen que el rendimiento hora, sea inferior al esperado dentro de una jornada laboral normal.

Estimamos correcto la aplicación de un coeficiente reductor por calidad de las mismas de 0,6; tras lo cual obtenemos un resultado de 360 horas. Si se estima

un valor aproximado de las mismas de 60 €, tendremos unos costes de personal próximos a los 21600€. En Tabla 7.1 presentamos un pequeño desglose de horas por hitos conseguidos. Estimamos un coste de hora hombre de 60 €.

ACTIVIDAD	Nº H	COSTE
Aprender el manejo entorno PowerDEVS	20	1200
Describir banco de pruebas del circuito propuesto	40	2400
Codificar y simular banco de pruebas en PowerDEVS	90	5400
Describir el circuito y banco de pruebas en VHDL	80	4800
Validación del modelo por comparación VHDL y PowerDEVS	30	1800
Redacción de memoria y documentos técnicos	100	6000
TOTAL	360	21600

Tabla 7.1: Tabla con desglose de horas por hito

7.3 CONCLUSIONES

Como se puede apreciar, las características del desarrollo de software son muy intensivas en mano de obra y no tanto en cuanto a lo que se refiere a bienes de equipo y gasto corriente. Esta constate también se ha reproducido en el desarrollo del presente PFC, pues en la estimación de costos del total valorado 21750€ los costes materiales solo representan el 0,6 % siendo el resto 99,4 % el coste de la mano de obra, representando este el capítulo más importante de gastos en el desarrollo del PFC.

8

CONCLUSIONES Y TRABAJOS FUTUROS

8.1 CONCLUSIONES

A la hora de establecer conclusiones, no debemos olvidar los objetivos del presente proyecto. Las conclusiones deben establecerse, según estos objetivos. Por tanto entendemos que lo correcto sería establecer un recorrido por dichos objetivos estableciendo las conclusiones pertinentes si las hubiese.

En el punto uno se establece como objetivo el conocimiento de circuito digital consistente en una CPU conectada a una memoria. De este punto, poco hay que añadir, pues entendemos, que se trata de un circuito que se explica con mucho detenimiento en varias asignaturas de la carrera de Ingeniería Informática, siendo el circuito propuesto, uno de los más básico de los tratados en los textos de dichas asignaturas, siendo su simplicidad su principal virtud.

En el punto dos, se expone la necesidad de un banco de pruebas para dicho circuito. El criterio establecido por nuestra parte para dicho circuito, ha sido el de desarrollar pequeños fragmentos de código que cumplan con el criterio de ser código, que se puede encontrar en cualquier programa comercial. Por otra parte utilice la máxima diversidad de instrucciones disponibles, para así, con un número reducido de pruebas, se puedan probar todas las instrucciones disponibles, sin necesidad de aumentar innecesariamente el número de pruebas.

Cabe resaltar, que entendemos, por una forma correcta de probar una CPU conectada a una memoria. Entendemos que consiste en desarrollar un programa y hacerlo correr en dicho circuito. Siendo la correcta ejecución del mismo, junto con la corrección de los resultados obtenidos, una prueba de su correcto funcionamiento.

En tercer punto, se establece el objetivo de realizar la descripción formal del circuito empleando DEVS. Este punto se ha completado con la implementación algorítmica del circuito propuesto, usando un entorno de desarrollo que implementa dicho formalismo.

Por tanto, la conclusión evidente que se desprende de lo dicho anteriormente, es la dificultad de entender el formalismo, que siendo sencillo en su definición (lo cual no impide la potencia del mismo, sino todo lo contrario), no nos ha resultado fácil la traslación del circuito base a la implementación formal del mismo.

En el cuarto punto, el objetivo es la comprensión de entorno de simulación de DEVS, como es PowerDEVS. En este punto pensamos que se impone opinar sobre dicho entorno, que cuenta entre sus inputs, con la interfaz gráfica que aun no siendo vistosa ni brillante en cuanto a su estética (recuerda en cierto modo a otros entornos de diseño antiguo), sí nos ha parecido muy intuitiva y de fácil comprensión, en lo que afecta a su funcionamiento más básico pensamos que es suficientemente potente para el objeto que nos ocupa.

En cuanto a sus inconvenientes entendemos que el más importante es, la escasa documentación existente sobre dicho entorno y sobre todo en lo que se refiere a documentación en castellano.

El quinto punto, hace referencia a la utilización de un entorno de simulación en VHDL. Entendemos que poco hay que añadir al respecto, pues el lenguaje VHDL ha demostrado sobradamente su utilidad en la descripción y simulación de diversos tipos de estructuras especialmente en circuitos electrónicos digitales, así como últimamente en los analógicos donde ha sido utilizado ampliamente.

En el punto sexto, se refiere a la comparación de resultados de ambas simulaciones. Cabe destacar que la naturaleza matemática de las pruebas desarrolladas en el banco de pruebas, ha facilitado dicha comparación de resultados. Pues se ha reducido en muchas ocasiones a una comparación numérica de ambos resultados, comprobándose en todos los casos, que los resultados obtenidos en las simulaciones, coincidían plenamente. Prueba inequívoca que ante programas idénticos se obtienen resultados idénticos.

Las conclusiones a un nivel de detalle más general, son positivas pues el proyecto por su marcado carácter académico, nos ha servido de recordatorio de temario y de asignaturas que teníamos ya olvidadas, debido a su carácter específico. Como es el caso del lenguaje VHDL, con el que no habíamos vuelto a tener contacto desde segundo curso de la carrera; no ha ocurrido lo mismo con DEVS, pues su estudio se ha realizado en quinto curso de la carrera, siendo por tanto más reciente.

8.2 TRABAJOS FUTUROS

En cuanto a trabajos futuros, entendemos que el formalismo está suficientemente explicado así como su desarrollo; sin embargo hemos echado en falta una herramienta que entendemos debería estar integrada en PowerDEVS (como en cualquier otro entorno), con el fin de poder realizar una traslación más

sencilla (automatizada) desde la descripción formal, al desarrollo algorítmico, con el entorno. Con lo que se nos asegura, que la descripción formal que se realice, es consecuente con su implementación y con el resultado obtenido de la simulación. Entendemos que este objetivo no es sencillo de realizar, pues no tenemos constancia de la existencia de dicho algoritmo de conversión, e incluso podría requerir de modificaciones del entorno importantes.

Otro campo de trabajo, podría ser, el incorporar documentación de ayuda al entorno PowerDEVS. Como se ha comentado anteriormente, ésta es muy escasa y no se encuentra fácilmente, lo que añade si cabe más dificultad al aprovechamiento máximo del entorno de simulación. Pues muchas de sus potencialidades, quedan un tanto ocultas para los usuarios.

Bibliografía y lista de referencias

- Blue Pacific, 2013: http://www.bluepc.com.

- Chu P. P.: "RTL Hardware Design using VHDL", Wiley-Interscience, 2006.

- Fceia, 2012: http://www.fceia.unr.edu.ar/lsd/powerdevs

- García J. y otros: "Aprenda C++ como si estuviera en primero". Universidad de Navarra, 1998.

- García J. y otros: "Aprenda lenguaje ANSI C como si estuviera en Primero". Universidad de Navarra 1998.

- Ibáñez, A.: "Estudio Comparativo para la Simulación de Modelos DEVS", Proyecto Fin de Carrera, E.T.S. Ingeniería Informática, UNED, Madrid, 2010.

- IEEE-SA Standards Board: "IEEE Standard VHDL Language Reference Manual", IEEE, 200.

- ISE, 14.7, 2013: http://www.xilinx.com/.

- Kernighan B.W.; D. M. Ritchie: "The C programming language", Prentice Hall, 1998.

- Martín C.; A. Urquia; M. A. Rubio: "Lenguajes de programación", Unidad Didáctica Colección Grado, Editorial UNED, 2011.

・ Mentor Graphics Corporation: "ModelSim LE/PE User's Manual", 2013.

http://www.model.com/resources/resources manuals.asp.

・ ModelSim Altera started edition, 13.1, 2013: http://www.altera.com.

・ Pagliero E. y otros: "PowerDEVS. Una Herramienta Integrada de Simulación por Eventos Discretos", Departamento de Electrónica, FCEIA –UNR CONICET.

・ Pedroni V. A.: "Circuit Design with VHDL", MIT Press, 2004.

・ Perry D.: "VHDL Programming by Example".4ª Edición, McGraw-Hill, 2002.

・ Ruz J.: "VHDL: de la tecnología a la arquitectura de computadores". Ed. Síntesis 2003.

・ Sourceforge, 2013: http://sourceforge.net/projects/powerdevs/?source=dlp

・ Stroustrup B.: "The C++ programming language", Addison-Wesley.

・ Urquia A.: "Modelado de Sistemas mediante DEVS. Teoría y práctica". Texto base de la asignatura Modelado de Sistemas Discretos, 5 curso ETS Ingeniería Informática. UNED. Curso 2008/09 (2008).

www.euclides.dia.uned.es/aurquia/Files/textoBase_MSD_2008_09.pdf

・ Urquia A.; C. Martín: "Diseño y análisis de circuitos digitales con VHDL", Unidad Didáctica Colección Grado, Editorial UNED, 2011.

・ Urquia A.; C. Martín: "Modelado y simulación de eventos discretos", unidad didáctica colección Grado, Editorial UNED, 2013.

· Wainer G. A.: "Discrete-Event Modeling and Simulation", CRC Press, 2009.

· Zeigler B. P.; H. Praehofer; T.G. Kim: "Theory of Modeling and Simulation", Academic Press.

ANEXO A

CÓDIGO VHDL

Archivo “control.vhd”.

```
library IEEE;
use IEEE.std_logic_1164.all; use work.cpu_lib.all;
entity control is
port( clock : in std_logic; reset : in std_logic; instrReg : in bit16; compout : in
std_logic; ready : in std_logic; progCntrWr : out std_logic;
progCntrRd : out std_logic;
addrRegWr : out std_logic; addrRegRd : out std_logic; outRegWr : out std_logic;
outRegRd : out std_logic; shiftSel : out t_shift; aluSel : out t_alu; compSel : out
t_comp; opRegRd : out std_logic; opRegWr : out std_logic; instrWr : out std_logic;
regSel : out t_reg; regRd : out std_logic; regWr : out std_logic; rw : out
std_logic; vma : out std_logic
);
end control;
architecture rtl of control is
signal current_state, next_state : state;
begin
nxtstateproc: process( current_state, instrReg, compout,
ready)
variable operacion : t_alu;
variable compara : t_comp;
begin
progCntrWr <= '0';
progCntrRd <= '0';
addrRegWr <= '0';
outRegWr <= '0';
outRegRd <= '0';
shiftSel <= shftpass;
aluSel <= alupass;
compSel <= eq;
opRegRd <= '0';
opRegWr <= '0';
instrWr <= '0';
regSel <= "000";
regRd <= '0';
regWr <= '0';
rw <= '0';
vma <= '0';
case current_state is when reset1 =>
aluSel <= zero after 1 ns;
shiftSel <= shftpass;
next_state <= reset2;
when reset2 =>
aluSel <= zero;
shiftSel <= shftpass;
outRegWr <= '1';
next_state <= reset3;
when reset3 =>
outRegRd <= '1';
next_state <= reset4;
when reset4 =>
outRegRd <= '1';
progCntrWr <= '1';
addrRegWr <= '1';
next_state <= reset5;
when reset5 =>
vma <= '1';
rw <= '0';
next_state <= reset6;
when reset6 =>
vma <= '1';
rw <= '0';
if ready = '1' then instrWr <= '1';
next_state <= execute;
else
next_state <= reset6;
```

```
end if;
when execute =>
case instrReg(15 downto 11) is
when "00000" =>       --- nop
next_state <= incPc;
when "00001" =>       --- load
regSel <= instrReg(5 downto 3);
regRd <= '1';
next_state <= load2;
when "00010" =>       --- store
regSel <= instrReg(2 downto 0);
regRd <= '1';
next_state <= store2;
when "00011" =>  --- move
regSel <= instrReg(5 downto 3);
regRd <= '1';
aluSel <= alupass;
shiftSel <= shftpass;
next_state <= move2;
when "00100" =>        --- loadI
progcntrRd <= '1';
alusel <= inc;
shiftsel <= shftpass;
next_state <= loadI2;
when "00101" =>       ---- BranchImm
progcntrRd <= '1';
alusel <= inc;
shiftsel <= shftpass;
next_state <= braI2;
when "00110" =>       ---- BranchGTImm
regSel <= instrReg(5 downto 3);
regRd <= '1';
compara := gt;
next_state <= bgtI2;
when "00111" =>   ----inc
regSel <= instrReg(2 downto 0);
regRd <= '1';
alusel <= inc;
shiftsel <= shftpass;
next_state <= inc2;
when "01001" =>   ----and
operacion := andOp;
regSel <= instrReg(5 downto 3);
regRd <= '1';
next_state <= sum2;
when "01010" =>   ----or
operacion := orOp;
 regSel <= instrReg(5 downto 3);
regRd <= '1';
next_state <= sum2;
when "01011" =>   ----xor
operacion := xorOp;
regSel <= instrReg(5 downto 3);
regRd <= '1';
next_state <= sum2;
when "01101" =>   ----sumar
operacion := plus;
 regSel <= instrReg(5 downto 3);
regRd <= '1';
next_state <= sum2;
when "01110" =>   ----resta
operacion := alusub;
 regSel <= instrReg(5 downto 3);
regRd <= '1';
next_state <= sum2;
when "10000" =>       ---- BranchLessImm
regSel <= instrReg(5 downto 3);
regRd <= '1';
compara := lt;
next_state <= bgtI2;
when "10011" =>       ---- BranchNEQImm
regSel <= instrReg(5 downto 3);
regRd <= '1';
compara := neq;
next_state <= bgtI2;
when "10111" =>       ---- BranchEQImm
regSel <= instrReg(5 downto 3);
regRd <= '1';
```

```
compara := eq;
next_state <= bgtI2;
when "11000" =>       ---- BranchLEQImm
regSel <= instrReg(5 downto 3);
regRd <= '1';
compara := lte;
next_state <= bgtI2;
when "11010" =>   ----SHL
regSel <= instrReg(2 downto 0);
regRd <= '1';
alusel <= alupass;
shiftsel <= shl;
next_state <= shl2;
when "11011" =>   ----SHR
regSel <= instrReg(2 downto 0);
regRd <= '1';
alusel <= alupass;
shiftsel <= shr;
next_state <= shr2;
when "11111" =>   ----parar
  next_state <= parar;
when others =>
next_state <= incPc;
end case;
when load2 =>
regSel <= instrReg(5 downto 3);
regRd <= '1';
addrregWr <= '1';
next_state <= load3;
when load3 =>
vma <= '1';
rw <= '0';
next_state <= load4;
when load4 =>
vma <= '1';
rw <= '0';
regSel <= instrReg(2 downto 0);
regWr <= '1';
next_state <= incPc;
when store2 =>
regSel <= instrReg(2 downto 0);
regRd <= '1';
addrregWr <= '1';
next_state <= store3;
when store3 =>
regSel <= instrReg(5 downto 3);
regRd <= '1';
next_state <= store4;
when store4 =>
regSel <= instrReg(5 downto 3);
regRd <= '1';
vma <= '1';
rw <= '1';
next_state <= incPc;
when move2 =>
regSel <= instrReg(5 downto 3);
regRd <= '1';
aluSel <= alupass;
shiftsel <= shftpass;
outRegWr <= '1';
next_state <= move3;
when move3 =>
outRegRd <= '1';
next_state <= move4;
when move4 =>
outRegRd <= '1';
regSel <= instrReg(2 downto 0);
regWr <= '1';
next_state <= incPc;
when loadI2 =>
progcntrRd <= '1';
alusel <= inc;
shiftsel <= shftpass;
outregWr <= '1';
next_state <= loadI3;
when loadI3 =>
outregRd <= '1';
next_state <= loadI4;
```

```
when loadI4 =>
outregRd <= '1';
progcntrWr <= '1';
addrregWr <= '1';
next_state <= loadI5;
when loadI5 =>
vma <= '1';
rw <= '0';
next_state <= loadI6;
when loadI6 =>
vma <= '1';
rw <= '0';
if ready = '1' then
regSel <= instrReg(2 downto 0);
regWr <= '1';
next_state <= incPc;
else
next_state <= loadI6;
end if;
when braI2 =>
progcntrRd <= '1';
alusel <= inc;
shiftsel <= shftpass;
outregWr <= '1';
next_state <= braI3;
when braI3 =>
outregRd <= '1';
next_state <= braI4;
when braI4 =>
outregRd <= '1';
progcntrWr <= '1';
addrregWr <= '1';
next_state <= braI5;
when braI5 =>
vma <= '1';
rw <= '0';
next_state <= braI6;
when braI6 =>
vma <= '1';
rw <= '0';
if ready = '1' then
progcntrWr <= '1';
next_state <= loadPc;
else
next_state <= braI6;
end if;
when bgtI2 =>
regSel <= instrReg(5 downto 3);
regRd <= '1';
opRegWr <= '1';
next_state <= bgtI3;
when bgtI3 =>
opRegRd <= '1';
regSel <= instrReg(2 downto 0);
regRd <= '1';
compsel <= compara;
next_state <= bgtI4;
when bgtI4 =>
opRegRd <= '1' after 1 ns;
regSel <= instrReg(2 downto 0);
regRd <= '1';
compsel <= compara;
if compout = '1' then
next_state <= bgtI5;
else
next_state <= incPc;
end if;
when bgtI5 =>
progcntrRd <= '1';
alusel <= inc;
shiftSel <= shftpass;
next_state <= bgtI6;
when bgtI6 =>
progcntrRd <= '1';
alusel <= inc;
shiftsel <= shftpass;
outregWr <= '1';
next_state <= bgtI7;
```

```
when bgtI7 =>
outregRd <= '1';
next_state <= bgtI8;
when bgtI8 =>
outregRd <= '1';
progcntrWr <= '1';
addrregWr <= '1';
next_state <= bgtI9;
when bgtI9 =>
vma <= '1';
rw <= '0';
next_state <= bgtI10;
when bgtI10 =>
vma <= '1';
rw <= '0';
if ready = '1' then
progcntrWr <= '1';
next_state <= loadPc;
else
next_state <= bgtI10;
end if;
when inc2 =>
regSel <= instrReg(2 downto 0);
regRd <= '1';
alusel <= inc;
shiftsel <= shftpass;
outregWr <= '1';
next_state <= inc3;
when inc3 =>
outregRd <= '1';
next_state <= inc4;
when inc4 =>
outregRd <= '1';
regsel <= instrReg(2 downto 0);
regWr <= '1';
next_state <= incPc;
when loadPc =>
progcntrRd <= '1';
next_state <= loadPc2;
when loadPc2 =>
progcntrRd <= '1';
addrRegWr <= '1';
next_state <= loadPc3;
when loadPc3 =>
vma <= '1';
rw <= '0';
next_state <= loadPc4;
when loadPc4 =>
vma <= '1';
rw <= '0';
if ready = '1' then
instrWr <= '1';
next_state <= execute;
else
next_state <= loadPc4;
end if;
when incPc =>
progcntrRd <= '1';
alusel <= inc;
shiftsel <= shftpass;
next_state <= incPc2;
when incPc2 =>
progcntrRd <= '1';
alusel <= inc;
shiftsel <= shftpass;
outregWr <= '1';
next_state <= incPc3;
when incPc3 =>
outregRd <= '1';
next_state <= incPc4;
when incPc4 =>
outregRd <= '1';
progcntrWr <= '1';
addrregWr <= '1';
next_state <= incPc5;
when incPc5 =>
vma <= '1';
rw <= '0';
```

```
next_state <= incPc6;
when incPc6 =>
vma <= '1';
rw <= '0';
if ready = '1' then
instrWr <= '1';
next_state <= execute;
else
next_state <= incPc6;
end if;
when sum2 =>
 regSel <= instrReg(5 downto 3);
regRd <= '1';
opRegWr <= '1';
next_state <= sum3;
when sum3 =>
opRegRd <= '1';
regSel <= instrReg(2 downto 0);
regRd <= '1';
alusel <= operacion;
next_state <= sum4;
when sum4 =>
opRegRd <= '1';
regSel <= instrReg(2 downto 0);
regRd <= '1';
alusel <= operacion;
shiftsel <= shftpass;
outregWr <= '1';
next_state <= sum5;
when sum5 =>
outregRd <= '1';
next_state <= sum6;
when sum6 =>
outregRd <= '1';
 regSel <= instrReg(5 downto 3);
regWr <= '1';
next_state <= incPc;
when shl2 =>
regSel <= instrReg(2 downto 0);
regRd <= '1';
alusel <= alupass;
shiftsel <= shl;
outregWr <= '1';
next_state <= shl3;
when shl3 =>
outregRd <= '1';
next_state <= shl4;
when shl4 =>
outregRd <= '1';
regsel <= instrReg(2 downto 0);
regWr <= '1';
next_state <= incPc;
when shr2 =>
regSel <= instrReg(2 downto 0);
regRd <= '1';
alusel <= alupass;
shiftsel <= shr;
outregWr <= '1';
next_state <= shr3;
when shr3 =>
outregRd <= '1';
next_state <= shr4;
when shr4 =>
outregRd <= '1';
regsel <= instrReg(2 downto 0);
regWr <= '1';
next_state <= incPc;
when parar =>
 null;
when others =>
--next_state <= incPc;
end case;
end process;
controlffProc: process(clock, reset)
begin
if reset = '1' then
current_state <= reset1 after 1 ns;
elsif clock'event and clock = '1' then
```

```
current_state <= next_state after 1 ns;
end if;
end process;
end rtl;
```

Archivo “cpu.vhd”.

```
library IEEE;
use IEEE.std_logic_1164.all; use work.cpu_lib.all;
entity cpu is
port(clock, reset, ready : in std_logic;
addr : out bit16;
rw, vma : out std_logic;
data : inout bit16);
end cpu;
architecture rtl of cpu is component regarray
port( data : in bit16; sel : in t_reg; en : in std_logic; clk : in std_logic; q
: out bit16);
end component;
component reg
port( a : in bit16;
clk : in std_logic;
q : out bit16);
end component;
component trireg
port( a : in bit16; en : in std_logic; clk : in std_logic; q : out bit16);
end component;
component control
port( clock : in std_logic; reset : in std_logic; instrReg : in bit16; compout
: in std_logic; ready : in std_logic; progCntrWr : out std_logic; progCntrRd :
out std_logic; addrRegWr : out std_logic; outRegWr : out std_logic; outRegRd :
out std_logic; shiftSel : out t_shift; aluSel : out t_alu; compSel : out
t_comp; opRegRd : out std_logic; opRegWr : out std_logic; instrWr : out
std_logic; regSel : out t_reg; regRd : out std_logic; regWr : out std_logic; rw
: out std_logic; vma : out std_logic );
end component;
component alu
port( a, b : in bit16; sel : in t_alu; c : out bit16);
end component;
component shift
port ( a : in bit16;
sel : in t_shift; y : out bit16);
end component;
component comp
port( a, b : in bit16;
sel : in t_comp;
compout : out std_logic);
end component;
signal opdata, aluout, shiftout, instrOutReg : bit16;
signal regsel : t_reg;
signal regRd, regWr, opregRd, opregWr, outregRd, outregWr,
addrregWr, instrregWr, progcntrRd, progcntrWr,
compout : std_logic;
signal alusel : t_alu;
signal shiftsel : t_shift;
signal compsel : t_comp;
begin
ra1 : regarray port map(data, regsel, regRd, regWr, data); opreg: trireg port
map (data, opregRd, opregWr, opdata); alu1: ALUport map (data, opdata, alusel,
aluout); shift1: shift port map (aluout, shiftsel, shiftout); outreg: trireg
port map (shiftout, outregRd, outregWr,
data);
addrreg: reg port map (data, addrregWr, addr);
progcntr: trireg port map (data, progcntrRd, progcntrWr, data);
comp1: comp port map (opdata, data, compsel, compout); instr1: reg port map
(data, instrregWr, instrOutReg);
con1: control port map (clock, reset, instrOutReg,
compout, ready, progcntrWr, progcntrRd, addrregWr, outregWr, outregRd,
shiftsel, alusel, compsel, opregRd, opregWr, instrregWr, regsel, regRd, regWr,
rw, vma);
end rtl;
```

Archivo “mem.vhd”.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.cpu_lib.all;
entity mem is
port (addr : in bit16;
sel, rw : in std_logic; ready : out std_logic; data : inout bit16);
end mem;
architecture behave of mem is
begin
memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;
variable mem_data : t_mem :=
       ("0010000000000001",  ---    0 loadI address
        "0000000000000001",  ---    1       10
        "0010000000000010",  ---    2 loadI 2,
        "0000000000110000",  ---    3 30
        "0010000000000110",  ---    4 loadI 6
        "0000000000000101",  ---    5 5
        "0010000000000011",  ---    6 loadI 3
        "0000000000000000",  ---    7 0
        "0110100000001010",  ---    8 add 1 2
        "0011000000011110",  ---    9 bgtI 3 - 6
        "0000000000001110",  ---    A E
        "0011100000000011",  ---    B inc 3
        "0010100000000000",  ---    C braI
        "0000000000001000",  ---    D 08
        "1000000000001010",  ---    E bgtI 2 - 1
        "0000000000010010",  ---    F 12
        "0010000000000001",  ---    10 loadI
        "0000000000011000",  ---    11 18
        "1111100000000000",  ---    12
        "0000000000000000",  ---    13
        "0000000000000101",  ---    14
        "0000000000000110",  ---    15
        "0000000000000111",  ---    16
        "0000000000001000",  ---    17
        "0000000000001001",  ---    18
        "0000000000001010",  ---    19
        "0000000000001011",  ---    1A
        "0000000000001100",  ---    1B
        "0000000000001101",  ---    1C
        "0000000000001110",  ---    1D
        "0000000000001111",  ---    1E
        "0000000000010000",  ---    1F
        "0000000000000000",  ---    20
        "0000000000000000",  ---    21
        "0000000000000000",  ---    22
        "0000000000000000",  ---    23
        "0000000000000000",  ---    24
        "0000000000000000",  ---    25
        "0000000000000000",  ---    26
        "0000000000000000",  ---    27
        "0000000000000000",  ---    28
        "0000000000000000",  ---    29
        "0000000000000000",  ---    2A
        "0000000000000000",  ---    2B
        "0000000000000000",  ---    2C
        "0000000000000000",  ---    2D
        "0000000000000000",  ---    2E
        "0000000000000000",  ---    2F
        "0000000000000000",  ---    30
        "0000000000000000",  ---    31
        "0000000000000000",  ---    32
        "0000000000000000",  ---    33
        "0000000000000000",  ---    34
        "0000000000000000",  ---    35
        "0000000000000000",  ---    36
        "0000000000000000",  ---    37
        "0000000000000000",  ---    38
        "0000000000000000",  ---    39
        "0000000000000000",  ---    3A
        "0000000000000000",  ---    3B
```

```
         "0000000000000000",  ---     3C
         "0000000000000000",  ---     3D
         "0000000000000000",  ---     3E
         "0000000000000000"); ---     3F

         begin
data <= "ZZZZZZZZZZZZZZZZ";
ready <= '0';
if sel = '1' then
if rw = '0' then
data <= mem_data(CONV_INTEGER(addr(15 downto 0))) after 1 ns;
ready <= '1';
elsif rw = '1' then
mem_data(CONV_INTEGER(addr(15 downto 0))) := data;
end if;
else
data <= "ZZZZZZZZZZZZZZZZ" after 1 ns;
end if;
end process;
end behave;

architecture primera of mem is
begin
memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;
variable mem_data : t_mem :=
        ("0010000000000001",  ---     0 loadI address
         "0000000000010000",  ---     1       10
         "0010000000000010",  ---     2 loadI 2,
         "0000000000000011",  ---     3 3
         "0010000000000011",  ---     4 loadI 3
         "0000000000000000",  ---     5 0
         "0010000000000100",  ---     6 load 4
         "0000000000000101",  ---     7 5
         "0011000000010001",  ---     8 bgtI 1
         "0000000000001110",  ---     9 E
         "0110100000001010",  ---     A b=b+a
         "0011100000000011",  ---     B inc 3
         "0010100000001111",  ---     C braI
         "0000000000001000",  ---     D 08
         "0011000000000000",  ---     E braI greater than
         "0000000000001100",  ---     F 12
         "0000100000000001",  ---     10 LoadI 1
         "0000000000010010",  ---     11 18
         "0000000000000000",  ---     12
         "0000000000000000",  ---     13
         "0000000000000000",  ---     14
         "0000000000000000",  ---     15
         "0000000000000000",  ---     16
         "0000000000000000",  ---     17
         "0000000000000000",  ---     18
         "0000000000000000",  ---     19
         "0000000000000000",  ---     1A
         "0000000000000000",  ---     1B
         "0000000000000000",  ---     1C
         "0000000000000000",  ---     1D
         "0000000000000000",  ---     1E
         "0000000000000000",  ---     1F
         "0000000000000000",  ---     20
         "0000000000000000",  ---     21
         "0000000000000000",  ---     22
         "0000000000000000",  ---     23
         "0000000000000000",  ---     24
         "0000000000000000",  ---     25
         "0000000000000000",  ---     26
         "0000000000000000",  ---     27
         "0000000000000000",  ---     28
         "0000000000000000",  ---     29
         "0000000000000000",  ---     2A
         "0000000000000000",  ---     2B
         "0000000000000000",  ---     2C
         "0000000000000000",  ---     2D
         "0000000000000000",  ---     2E
         "0000000000000000",  ---     2F
         "0000000000000000",  ---     30
         "0000000000000000",  ---     31
         "0000000000000000",  ---     32
         "0000000000000000",  ---     33
```

```
         "0000000000000000",  ---     34
         "0000000000000000",  ---     35
         "0000000000000000",  ---     36
         "0000000000000000",  ---     37
         "0000000000000000",  ---     38
         "0000000000000000",  ---     39
         "0000000000000000",  ---     3A
         "0000000000000000",  ---     3B
         "0000000000000000",  ---     3C
         "0000000000000000",  ---     3D
         "0000000000000000",  ---     3E
         "0000000000000000"); ---     3F
         begin
data <= "ZZZZZZZZZZZZZZZZ";
ready <= '0';
if sel = '1' then
if rw = '0' then
data <= mem_data(CONV_INTEGER(addr(15 downto 0))) after 1 ns;
ready <= '1';
elsif rw = '1' then
mem_data(CONV_INTEGER(addr(15 downto 0))) := data;
end if;
else
data <= "ZZZZZZZZZZZZZZZZ" after 1 ns;
end if;
end process;
end primera;

architecture whileyif of mem is
begin
memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;
variable mem_data : t_mem :=
        ("0010000000000000",  ---     0 loadI reg 0
         "0000000000000010",  ---     1       a=2
         "0010000000000001",  ---     2 loadI reg 1,
         "0000000000001111",  ---     3 15
         "0010000000000010",  ---     4 loadI reg2
         "0000000000000111",  ---     5 c=7
         "0010000000000011",  ---     6 load reg 3
         "0000000000001010",  ---     7 10
         "0011000000010011",  ---     8 bgtI reg 3 reg2
         "0000000000001110",  ---     9 E
         "0111000000001000",  ---     A b=b-a
         "0011100000000010",  ---     B inc c++
         "0010100000001111",  ---     C braI
         "0000000000001000",  ---     D 8
         "1011100000010011",  ---     E braI  if reg 3 es != a reg 2
         "0000000000010010",  ---     F 18
         "0010100000000000",  ---     10 branchI (else)
         "0000000000010100",  ---     11 20
         "0101000000000001",  ---     12 or reg 0 reg 1
         "1111100000000000",  ---     13 END
         "0000000000000000",  ---     14
         "0000000000000000",  ---     15
         "0000000000000000",  ---     16
         "0000000000000000",  ---     17
         "0000000000000000",  ---     18
         "0000000000000000",  ---     19
         "0000000000000000",  ---     1A
         "0000000000000000",  ---     1B
         "0000000000000000",  ---     1C
         "0000000000000000",  ---     1D
         "0000000000000000",  ---     1E
         "0000000000000000",  ---     1F
         "0000000000000000",  ---     20
         "0000000000000000",  ---     21
         "0000000000000000",  ---     22
         "0000000000000000",  ---     23
         "0000000000000000",  ---     24
         "0000000000000000",  ---     25
         "0000000000000000",  ---     26
         "0000000000000000",  ---     27
         "0000000000000000",  ---     28
         "0000000000000000",  ---     29
         "0000000000000000",  ---     2A
         "0000000000000000",  ---     2B
         "0000000000000000",  ---     2C
```

```
         "0000000000000000",  ---     2D
         "0000000000000000",  ---     2E
         "0000000000000000",  ---     2F
         "0000000000000000",  ---     30
         "0000000000000000",  ---     31
         "0000000000000000",  ---     32
         "0000000000000000",  ---     33
         "0000000000000000",  ---     34
         "0000000000000000",  ---     35
         "0000000000000000",  ---     36
         "0000000000000000",  ---     37
         "0000000000000000",  ---     38
         "0000000000000000",  ---     39
         "0000000000000000",  ---     3A
         "0000000000000000",  ---     3B
         "0000000000000000",  ---     3C
         "0000000000000000",  ---     3D
         "0000000000000000",  ---     3E
         "0000000000000000"); ---     3F
         begin
data <= "ZZZZZZZZZZZZZZZZ";
ready <= '0';
if sel = '1' then
if rw = '0' then
data <= mem_data(CONV_INTEGER(addr(15 downto 0))) after 1 ns;
ready <= '1';
elsif rw = '1' then
mem_data(CONV_INTEGER(addr(15 downto 0))) := data;
end if;
else
data <= "ZZZZZZZZZZZZZZZZ" after 1 ns;
end if;
end process;
end whileyif;

architecture bucleinterno of mem is
begin
memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;
variable mem_data : t_mem :=
       ("0010000000000000",  ---     0 loadI reg 0
         "0000000000000010",  ---     1       a=2
         "0010000000000001",  ---     2 loadI reg 1,
         "0000000000001111",  ---     3 15
         "0010000000000010",  ---     4 loadI reg2
         "0000000000000111",  ---     5 c=7
         "0010000000000011",  ---     6 load reg 3
         "0000000000001010",  ---     7 10
         "0010000000000100",  ---     8 loadI reg 4
         "0000000000000010",  ---     9 f=2
         "1011100000010011",  ---     A branch is equal reg2 reg 3
         "0000000000011000",  ---     B 24
         "0111000000011000",  ---     C d=d-a
         "0010000000000101",  ---     D f=0
         "0000000000000000",  ---     E 0
         "1011100000101100",  ---     F branch if equal (for
         "0000000000010101",  ---     21
         "0110100000011000",  ---     11 d=d+a
         "0011100000000101",  ---     12 f++
         "0010100000000000",  ---     13 branchI
         "0000000000001111",  ---     14 f
         "0011100000000010",  ---     15 inc c++
         "0010100000000000",  ---     16 branchI
         "0000000000001010",  ---     17 10
         "1111100000000000",  ---     18 end
         "0000000000000000",  ---     19
         "0000000000000000",  ---     1A
         "0000000000000000",  ---     1B
         "0000000000000000",  ---     1C
         "0000000000000000",  ---     1D
         "0000000000000000",  ---     1E
         "0000000000000000",  ---     1F
         "0000000000000000",  ---     20
         "0000000000000000",  ---     21
         "0000000000000000",  ---     22
         "0000000000000000",  ---     23
         "0000000000000000",  ---     24
         "0000000000000000",  ---     25
```

```
        "0000000000000000",  ---    26
        "0000000000000000",  ---    27
        "0000000000000000",  ---    28
        "0000000000000000",  ---    29
        "0000000000000000",  ---    2A
        "0000000000000000",  ---    2B
        "0000000000000000",  ---    2C
        "0000000000000000",  ---    2D
        "0000000000000000",  ---    2E
        "0000000000000000",  ---    2F
        "0000000000000000",  ---    30
        "0000000000000000",  ---    31
        "0000000000000000",  ---    32
        "0000000000000000",  ---    33
        "0000000000000000",  ---    34
        "0000000000000000",  ---    35
        "0000000000000000",  ---    36
        "0000000000000000",  ---    37
        "0000000000000000",  ---    38
        "0000000000000000",  ---    39
        "0000000000000000",  ---    3A
        "0000000000000000",  ---    3B
        "0000000000000000",  ---    3C
        "0000000000000000",  ---    3D
        "0000000000000000",  ---    3E
        "0000000000000000"); ---    3F
        begin
data <= "ZZZZZZZZZZZZZZZZ";
ready <= '0';
if sel = '1' then
if rw = '0' then
data <= mem_data(CONV_INTEGER(addr(15 downto 0))) after 1 ns;
ready <= '1';
elsif rw = '1' then
mem_data(CONV_INTEGER(addr(15 downto 0))) := data;
end if;
else
data <= "ZZZZZZZZZZZZZZZZ" after 1 ns;
end if;
end process;

end bucleinterno;


architecture operbasicas of mem is
begin
memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;
variable mem_data : t_mem :=
       ("0010000000000000",  ---    0 loadI address
        "0000000010110010",  ---    1       a= 178
        "0010000000000001",  ---    2 loadI 1,
        "0000000110011001",  ---    3 b= 409
        "0001100000000010",  ---    4 move reg0 a reg 2
        "0110100000010001",  ---    5 c= b+a
        "0001100000000010",  ---    6 move reg0 a reg 2
        "0100100000010001",  ---    7 c=a and b
        "0001100000000010",  ---    8 move reg0 a reg 2
        "0101000000010001",  ---    9 c= a or b
        "0001100000000010",  ---    A move reg0 a reg 2
        "0101100000010001",  ---    B c= a xor b
        "0001100000000010",  ---    C move reg0 a reg 2
        "1101000000000010",  ---    D shift left a
        "0001100000000010",  ---    E move reg0 a reg 2
        "1101100000000010",  ---    F shift right a
        "0001100000000010",  ---    10 move reg0 a reg 2
        "0111000000001010",  ---    11 c=b-a
        "1111100000000000",  ---    12 end
        "0000000000000000",  ---    13
        "0000000000000000",  ---    14
        "0000000000000000",  ---    15
        "0000000000000000",  ---    16
        "0000000000000000",  ---    17
        "0000000000000000",  ---    18
        "0000000000000000",  ---    19
        "0000000000000000",  ---    1A
        "0000000000000000",  ---    1B
        "0000000000000000",  ---    1C
```

```
         "0000000000000000",  ---    1D
         "0000000000000000",  ---    1E
         "0000000000000000",  ---    1F
         "0000000000000000",  ---    20
         "0000000000000000",  ---    21
         "0000000000000000",  ---    22
         "0000000000000000",  ---    23
         "0000000000000000",  ---    24
         "0000000000000000",  ---    25
         "0000000000000000",  ---    26
         "0000000000000000",  ---    27
         "0000000000000000",  ---    28
         "0000000000000000",  ---    29
         "0000000000000000",  ---    2A
         "0000000000000000",  ---    2B
         "0000000000000000",  ---    2C
         "0000000000000000",  ---    2D
         "0000000000000000",  ---    2E
         "0000000000000000",  ---    2F
         "0000000000000000",  ---    30
         "0000000000000000",  ---    31
         "0000000000000000",  ---    32
         "0000000000000000",  ---    33
         "0000000000000000",  ---    34
         "0000000000000000",  ---    35
         "0000000000000000",  ---    36
         "0000000000000000",  ---    37
         "0000000000000000",  ---    38
         "0000000000000000",  ---    39
         "0000000000000000",  ---    3A
         "0000000000000000",  ---    3B
         "0000000000000000",  ---    3C
         "0000000000000000",  ---    3D
         "0000000000000000",  ---    3E
         "0000000000000000"); ---    3F
         begin
data <= "ZZZZZZZZZZZZZZZZ";
ready <= '0';
if sel = '1' then
if rw = '0' then
data <= mem_data(CONV_INTEGER(addr(15 downto 0))) after 1 ns;
ready <= '1';
elsif rw = '1' then
mem_data(CONV_INTEGER(addr(15 downto 0))) := data;
end if;
else
data <= "ZZZZZZZZZZZZZZZZ" after 1 ns;
end if;
end process;

end operbasicas;

architecture copiarmem of mem is
begin
memproc: process(addr, sel, rw)
type t_mem is array(0 to 63) of bit16;
variable mem_data : t_mem :=
       ("0010000000000001",  ---    0 loadI address
         "0000000000010000",  ---    1       10
         "0010000000000010",  ---    2 loadI 2,
         "0000000000110000",  ---    3 30
         "0010000000000110",  ---    4 loadI 6,
         "0000000000100000",  ---    5 20
         "0000100000001011",  ---    6 load 1, 3
         "0001000000011010",  ---    7 store 3
         "0011000000001110",  ---    8 bgtI 1, 6
         "0000000000001110",  ---    9 E
         "0011100000000001",  ---    A inc 1
         "0011100000000010",  ---    B inc 2
         "0010100000001111",  ---    C braI
         "0000000000000110",  ---    D 06
         "1111100000000000",  ---    E parar
         "1111100000000000",  ---    F
         "0000000000000001",  ---    10
         "0000000000000010",  ---    11
         "0000000000000011",  ---    12
         "0000000000000100",  ---    13
         "0000000000000101",  ---    14
```

```
        "0000000000000110",  ---    15
        "0000000000000111",  ---    16
        "0000000000001000",  ---    17
        "0000000000001001",  ---    18
        "0000000000001010",  ---    19
        "0000000000001011",  ---    1A
        "0000000000001100",  ---    1B
        "0000000000001101",  ---    1C
        "0000000000001110",  ---    1D
        "0000000000001111",  ---    1E
        "0000000000010000",  ---    1F
        "0000000000000000",  ---    20
        "0000000000000000",  ---    21
        "0000000000000000",  ---    22
        "0000000000000000",  ---    23
        "0000000000000000",  ---    24
        "0000000000000000",  ---    25
        "0000000000000000",  ---    26
        "0000000000000000",  ---    27
        "0000000000000000",  ---    28
        "0000000000000000",  ---    29
        "0000000000000000",  ---    2A
        "0000000000000000",  ---    2B
        "0000000000000000",  ---    2C
        "0000000000000000",  ---    2D
        "0000000000000000",  ---    2E
        "0000000000000000",  ---    2F
        "0000000000000000",  ---    30
        "0000000000000000",  ---    31
        "0000000000000000",  ---    32
        "0000000000000000",  ---    33
        "0000000000000000",  ---    34
        "0000000000000000",  ---    35
        "0000000000000000",  ---    36
        "0000000000000000",  ---    37
        "0000000000000000",  ---    38
        "0000000000000000",  ---    39
        "0000000000000000",  ---    3A
        "0000000000000000",  ---    3B
        "0000000000000000",  ---    3C
        "0000000000000000",  ---    3D
        "0000000000000000",  ---    3E
        "0000000000000000"); ---    3F

        begin
data <= "ZZZZZZZZZZZZZZZZ";
ready <= '0';
if sel = '1' then
if rw = '0' then
data <= mem_data(CONV_INTEGER(addr(15 downto 0))) after 1 ns;
ready <= '1';
elsif rw = '1' then
mem_data(CONV_INTEGER(addr(15 downto 0))) := data;
end if;
else
data <= "ZZZZZZZZZZZZZZZZ" after 1 ns;
end if;
end process;
end copiarmem;
```

ANEXO B

CÓDIGO PARA ENTORNO POWERDEVS

Código C++.

Archivos .ccp y .h “addrreg”

```
//CPP:vector/addrreg.cpp
#if !defined addrreg_h
#define addrreg_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class addrreg: public Simulator {
// Declare the state,
// output variables
// and parameters
//out
double y[1];
double Sigma;
#define INF 1e20
public:
        addrreg(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
#include "addrreg.h"
void addrreg::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
}
double addrreg::ta(double t) {
//This function returns a double.
return Sigma;
}
void addrreg::dint(double t) {
Sigma=INF;
}
void addrreg::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
vector24 ent;
ent=*(vector24*)x.value;
if (ent.value[1]==1){
y[0]=ent.value[1];
ent.value[1]=0;
Sigma=0;}
}
Event addrreg::lambda(double t) {
//This function returns an Event:
```

```
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void addrreg::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “alusel”

```
//CPP:vector/alusel.cpp
#if !defined alusel_h
#define alusel_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class alusel: public Simulator {
// Declare the state,
// output variables
// and parameters
vector24 ent;
//out
double y[4];
double Sigma;
#define INF 1e20
public:
       alusel(const char *n): Simulator(n) {};
       void init(double, ...);
       double ta(double t);
       void dint(double);
       void dext(Event , double );
       Event lambda(double);
       void Exit();
};
#endif
```

```
#include "alusel.h"
void alusel::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
}
double alusel::ta(double t) {
//This function returns a double.
return Sigma;
}
void alusel::dint(double t) {
Sigma=INF;
}
void alusel::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
ent=*(vector24*)x.value;
if(ent.value[6]<9){
y[0]=ent.value[6];
y[1]=ent.value[7];
y[2]=ent.value[8];
y[3]=ent.value[9];
Sigma=0;
```

```
}
}
Event alusel::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void alusel::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “compsel”

```
//CPP:vector/compsel.cpp
#if !defined compsel_h
#define compsel_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class compsel: public Simulator {
// Declare the state,
// output variables
// and parameters
vector24 ent;
//out
double y[3];
double Sigma;
#define INF 1e20
public:
        compsel(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "compsel.h"
void compsel::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
}
double compsel::ta(double t) {
//This function returns a double.
return Sigma;
}
void compsel::dint(double t) {
Sigma=INF;
}
void compsel::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
ent=*(vector24*)x.value;
if (ent.value[10]<10){
y[0]=ent.value[10];
y[1]=ent.value[11];
y[2]=ent.value[12];
```

```
Sigma=0;}
}
Event compsel::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void compsel::Exit() {
//Code executed at the end of the simulation.
}
```

Archivo .ccp y .h “control4”

```
//CPP:Projectro/control4.cpp
#if !defined control4_h
#define control4_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"
#include "vector/vector24.h"
class control4: public Simulator {
// Declare the state,
// output variables
// and parameters
int valor,res1,fase;
int steep,instr;
double e[2],n,l,p,f,g,h,m;
vector16 vec3,vec1;
//out
double y[24];
double Sigma;
#define INF 1e20
public:
       control4(const char *n): Simulator(n) {};
       void init(double, ...);
       double ta(double t);
       void dint(double);
       void dext(Event , double );
       Event lambda(double);
       void Exit();
};
#endif
```

```
#include "control4.h"
void control4::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
for (int i=0;i<24;i++){
y[i]=10;
}
}
double control4::ta(double t) {
//This function returns a double.
return Sigma;//This function returns a double.
}
void control4::dint(double t) {
Sigma=INF;
}
void control4::dext(Event x, double t) {
//The input event is in the 'x' variable.
```

```
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
/*mapeo de puertos de entrada
in0 reset
in1 clok
in2 instrReg (memoria);
in3 Compout  (comparador);
in4 Ready
*/
double a[2];
vector16 vec2;
int res1=0;
vector24 vex;
//
for (int i=0;i<22;i++){
y[i]=10;
}
if (x.port==0){
a[0]=*(double*) x.value;
//y[0]= vec1.value[0];
if (a[0]==1){
if (steep==0)
steep = 11;
}}
if (x.port==2){
vec1=*(vector16*) x.value;
fase =
(int)((vec1.value[0]*16)+(vec1.value[1]*8)+(vec1.value[2]*4)+(vec1.value[3]*2)+(vec1
.value[4]));
}
if(x.port==4){
vec3=*(vector16*)x.value;
e[0]==*(double*) x.value;
}
if(x.port==3){
vec2=*(vector16*)x.value;
//e[0]==*(double*) x.value;
}
switch(steep){
case 11:
y[6]=1;
y[7]=0;
y[8]=0;
y[9]=1;
Sigma=0;
steep=12;
break;
case 12:
y[3]=1;
y[4]=1;
y[5]=1;
Sigma=0;
steep=13;
break;
case 13:
y[2]=1;
Sigma=0;
steep=14;
break;
case 14:
y[0]=1;
y[1]=1;
Sigma=0;
steep=15;
break;
case 15:
y[19]=0;
Sigma=0;
steep=16;
break;
case 16:
y[20]=1;
Sigma=0;
steep=17;
break;
case 17:
```

```
//steep=18;
if (vec3.value[0]==1){
//vec3.value[0]=0;
y[14]=1;
Sigma=0;
steep=18;
break;
}else{
steep=17;
break;
}
case 18:
instr=fase*10;
//instr=1;
steep=19;
break;
case 19:
switch(instr){
// NO OPERACION.
case 0:
instr=400;
break;
//load.
case 10:
y[15]=vec1.value[10];
y[16]=vec1.value[11];
y[17]=vec1.value[12];
Sigma=0;
instr=11;
break;
case 11:
y[18]=1;
Sigma=0;
instr=12;
break;
case 12:
y[22]=1;
Sigma=0;
instr=1121;
break;
case 1121:
y[22]=0;
y[1]=1;
Sigma=0;
instr=13;
break;
case 13:
//y[22]=0;
y[19]=0;
Sigma=0;
instr=14;
break;
case 14:
y[20]=1;
Sigma=0;
instr=15;
break;
case 15:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=16;
break;
case 16:
y[18]=0;
Sigma=0;
instr=310;
break;
//store.
case 20:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=21;
break;
case 21:
```

```
y[18]=1;
Sigma=0;
instr=22;
break;
case 22:
y[22]=1;
Sigma=0;
instr=23;
break;
case 23:
y[22]=0;
y[1]=1;
Sigma=0;
instr=24;
break;
case 24:
y[15]=vec1.value[10];
y[16]=vec1.value[11];
y[17]=vec1.value[12];
Sigma=0;
instr=25;
break;
case 25:
y[18]=1;
Sigma=0;
instr=26;
break;
case  26:
y[22]=1;
Sigma=0;
instr=27;
break;
case 27:
y[22]=0;
y[1]=0;
y[19]=1;
Sigma=0;
instr=310;
break;
//move
case 30:
y[15]=vec1.value[10];
y[16]=vec1.value[11];
y[17]=vec1.value[12];
Sigma=0;
instr=31;
break;
case 31:
y[18]=1;
Sigma=0;
instr=32;
break;
case 32:
y[22]=1;
Sigma=0;
instr=33;
break;
case 33:
y[22]=0;
y[6]=0;
y[7]=0;
y[8]=0;
y[9]=0;
Sigma=0;
instr=34;
break;
case 34:
y[3]=1;
y[4]=1;
y[5]=1;
Sigma=0;
instr=35;
break;
case 35:
y[2]=1;
Sigma=0;
instr=36;
break;
```

```
case 36:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=37;
break;
case 37:
y[18]=0;
Sigma=0;
instr=310;
break;
//loadI
case 40:
y[0]=0;
Sigma=0;
instr=410;
break;
case 410:
y[21]=1;
Sigma=0;
instr=41;
break;
case 41:
y[6]=0;
y[7]=1;
y[8]=1;
y[9]=1;
Sigma=0;
instr=42;
break;
case 42:
y[3]=1;
y[4]=1;
y[5]=1;
Sigma=0;
instr=43;
break;
case 43:
y[2]=1;
Sigma=0;
instr=44;
break;
case 44:
y[0]=1;
y[1]=1;
Sigma=0;
instr=45;
break;
case 45:
y[19]=0;
Sigma=0;
instr=46;
break;
case 46:
y[20]=1;
Sigma=0;
instr=47;
break;
case 47:
if (vec3.value[0]==1){
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=48;
break;}else{
instr=47;}
case 48:
y[18]=0;
Sigma=0;
instr=310;
break;
//branI
case 50:
y[0]=0;
Sigma=0;
instr=510;
```

```
break;
case 510:
y[21]=1;
Sigma=0;
instr=51;
break;
case 51:
y[6]=0;
y[7]=1;
y[8]=1;
y[9]=1;
Sigma=0;
instr=52;
break;
case 52:
y[3]=1;
y[4]=1;
y[5]=1;
Sigma=0;
instr=53;
break;
case 53:
y[2]=1;
Sigma=0;
instr=54;
break;
case 54:
y[0]=1;
y[1]=1;
Sigma=0;
instr=55;
break;
case 55:
y[19]=0;
Sigma=0;
instr=56;
break;
case 56:
y[20]=1;
Sigma=0;
instr=57;
break;
case 57:
if (vec3.value[0]==1){
y[0]=1;
Sigma=0;
instr=300;
break;}else{
instr=57;
break;}
//bgtI
//bgtI
case 60:
n=0;
l=1;
p=0;
instr= 161;
break;
case 161:
y[0]=0;
Sigma=0;
instr=61;
break;
case 61:
y[21]=1;
Sigma=0;
instr=62;
break;
case 62:
//y[21]=0;
y[6]=0;
y[7]=1;
y[8]=1;
y[9]=1;
Sigma=0;
instr=63;
break;
case 63:
```

```
y[3]=1;
y[4]=1;
y[5]=1;
Sigma=0;
instr=64;
break;
case 64:
y[2]=1;
Sigma=0;
instr=650;
break;
case 650:
y[0]=1;
Sigma=0;
instr=65;
break;
case 65:
y[0]=0;
y[15]=vec1.value[10];
y[16]=vec1.value[11];
y[17]=vec1.value[12];
Sigma=0;
instr=66;
break;
case 66:
y[18]=1;
Sigma=0;
instr=67;
break;
case 67:
y[22]=1;
Sigma=0;
instr=68;
break;
case 68:
y[22]=0;
y[13]=0;
Sigma=0;
instr=69;
break;
case 69:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=690;
break;
case 690:
y[18]=1;
Sigma=0;
instr=691;
break;
case 691:
y[22]=1;
y[13]=1;
Sigma=0;
instr=692;
break;
case 692:
y[22]=0;
y[10]=n;
y[11]=l;
y[12]=p;
Sigma=0;
instr=693;
break;
case 693:
if (vec2.value[0]==1){
y[0]=0;
instr=694;
break;
}else{
instr=310;
break;}
case 6941:
y[21]=1;
instr=694;
break;
```

```
case 694:
y[21]=1;
Sigma=0;
instr=695;
break;
case 695:
y[1]=1;
Sigma=0;
instr=696;
break;
case 696:
y[19]=0;
Sigma=0;
instr=697;
break;
case 697:
y[20]=1;
Sigma=0;
instr=698;
break;
case 698:
if (vec3.value[0]==1){
y[0]=1;
Sigma=0;
instr=300;
break;
}else{
instr=698;
//instr=300;
break;
}
//bgtI if less than
case 160:
n=1;
l=0;
p=0;
instr=161;
break;
//bhtI if not equal
case 190:
n=0;
l=0;
p=1;
instr=161;
break;
//bgtI if equals
case 230:
n=0;
l=0;
p=0;
instr=161;
break;
//bgtI if lessor equals
case 240:
n=1;
l=0;
p=1;
instr=161;
break;
//alu
//operaciones unarias
//not a
case 120:
f=0;
g=0;
h=1;
m=1;
n=1;
l=1;
p=1;
instr=71;
break;
//c=0
case 150:
f=1;
g=0;
h=0;
m=1;
```

```
n=1;
l=1;
p=1;
instr=71;
break;
//decrement
case 80:
f=1;
g=0;
h=0;
m=0;
n=1;
l=1;
p=1;
instr=71;
break;
//inc
case 70:
f=0;
g=1;
h=1;
m=1;
n=1;
l=1;
p=1;
instr=71;
break;
case 71:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=72;
break;
case 72:
y[18]=1;
Sigma=0;
instr=73;
break;
case 73:
y[22]=1;
Sigma=0;
instr=74;
break;
case 74:
y[22]=0;
y[6]=f;
y[7]=g;
y[8]=h;
y[9]=m;
Sigma=0;
instr=75;
break;
case 75:
y[3]=n;
y[4]=l;
y[5]=p;
Sigma=0;
instr=76;
break;
case 76:
y[2]=1;
Sigma=0;
instr=77;
break;
case 77:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=78;
break;
case 78:
y[18]=0;
Sigma=0;
instr=310;
break;
// and
```

```
case 90:
f=0;
g=0;
h=0;
m=1;
instr=131;
break;
//or
case 100:
f=0;
g=0;
h=1;
m=0;
instr=131;
break;
//xor
case 110:
f=0;
g=1;
h=0;
m=0;
instr=131;
break;
//sub
case 140:
f=0;
g=1;
h=1;
m=0;
instr=131;
break;
//add
case 130:
f=0;
g=1;
h=0;
m=1;
instr=131;
break;
case 131:
y[15]=vec1.value[10];
y[16]=vec1.value[11];
y[17]=vec1.value[12];
Sigma=0;
instr=132;
break;
case 132:
y[18]=1;
Sigma=0;
instr=133;
break;
case 133:
y[22]=1;
Sigma=0;
instr=134;
break;
case 134:
y[22]=0;
y[13]=0;
Sigma=0;
instr=135;
break;
case 135:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=136;
break;
case 136:
y[18]=1;
Sigma=0;
instr=137;
break;
case 137:
y[22]=1;
y[13]=1;
Sigma=0;
```

```
instr=138;
break;
case 138:
y[22]=0;
y[6]=f;
y[7]=g;
y[8]=h;
y[9]=m;
Sigma=0;
instr=139;
break;case 139:
y[3]=1;
y[4]=1;
y[5]=1;
Sigma=0;
instr=141;
break;
case 141:
y[2]=1;
Sigma=0;
instr=142;
break;
case 142:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=143;
break;
case 143:
y[18]=0;
Sigma=0;
instr=310;
break;
//branch
// salto a direccion contenida en un registro
case 200:
n=0;
l=1;
p=0;
instr= 201;
break;
case 201:
y[0]=0;
Sigma=0;
instr=202;
break;
case 202:
y[21]=1;
Sigma=0;
instr=203;
break;
case 203:
//y[21]=0;
y[6]=0;
y[7]=1;
y[8]=1;
y[9]=1;
Sigma=0;
instr=204;
break;
case 204:
y[3]=1;
y[4]=1;
y[5]=1;
Sigma=0;
instr=205;
break;
case 205:
y[2]=1;
Sigma=0;
instr=206;
break;
case 206:
y[0]=1;
Sigma=0;
instr=207;
break;
```

```
case 207:
y[0]=0;
y[15]=vec1.value[10];
y[16]=vec1.value[11];
y[17]=vec1.value[12];
Sigma=0;
instr=208;
break;
case 208:
y[18]=1;
Sigma=0;
instr=209;
break;
case 209:
y[22]=1;
Sigma=0;
instr=221;
break;
case 221:
y[22]=0;
y[13]=0;
Sigma=0;
instr=222;
break;
case 222:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=223;
break;
case 223:
y[18]=1;
Sigma=0;
instr=224;
break;
case 224:
y[22]=1;
y[13]=1;
Sigma=0;
instr=225;
break;
case 225:
y[22]=0;
y[10]=n;
y[11]=l;
y[12]=p;
Sigma=0;
instr=226;
break;
case 226:
if (vec2.value[0]==1){
y[0]=0;
instr=227;
break;
}else{
instr=310;
break;}
case 227:
y[21]=1;
Sigma=0;
instr=228;
break;
case 228:
y[1]=1;
Sigma=0;
instr=229;
break;
case 229:
y[19]=0;
Sigma=0;
instr=251;
break;
case 251:
y[20]=1;
Sigma=0;
instr=252;
break;
```

```
case 252:
if (vec3.value[0]==1){
y[14]=1;
Sigma=0;
instr=253;
break;
}else{
instr=252;
//instr=300;
break;
}
case 253:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=254;
break;
case 254:
y[18]=1;
Sigma=0;
instr=255;
break;
case 255:
y[22]=1;
Sigma=0;
instr=256;
break;
case 256:
y[22]=0;
y[0]=1;
Sigma=0;
instr=300;
break;
//bgt if less than
case 170:
n=1;
l=0;
p=0;
instr=201;
break;
//bhtI if not equal
case 180:
n=0;
l=0;
p=1;
instr=201;
break;
//bgtI if equals
case 220:
n=0;
l=0;
p=0;
instr=201;
break;
//bgtI if lessor equals
case 250:
n=1;
l=0;
p=1;
instr=201;
break;
//shift left
case 260:
f=0;
g=0;
h=0;
m=0;
n=0;
l=0;
p=1;
instr=71;
break;
//shift rigth
case 270:
f=0;
g=0;
h=0;
```

```
m=0;
n=0;
l=1;
p=0;
instr=71;
break;
// resultado
case 280:
y[15]=vec1.value[13];
y[16]=vec1.value[14];
y[17]=vec1.value[15];
Sigma=0;
instr=281;
break;
case 281:
//y[22]=1;
y[18]=1;
Sigma=0;
instr=282;
break;
case 282:
y[22]=1;
Sigma=0;
instr=283;
break;
case 283:
y[22]=0;
y[23]=1;
Sigma=0;
instr=284;
break;
case 284:
y[23]=0;
Sigma=0;
instr=310;
break;
//LoadPc
case 300:
y[0]=0;
Sigma=0;
instr=301;
break;
case 301:
y[21]=1;
Sigma=0;
instr=302;
break;
case 302:
y[1]=1;
Sigma=0;
instr=303;
break;
case 303:
y[19]=0;
Sigma=0;
instr=304;
break;
case 304:
y[20]=1;
Sigma=0;
instr=305;
break;
case 305:
if (vec3.value[0]==1){
y[14]=1;
instr=306;
Sigma=0;
instr=306;
break;
}else{
instr=305;
//instr=310;
break;
}
case 306:
//instr=310;
instr=fase*10;
break;
```

```
//instr=310;
instr=fase*10;
break;
case 310://IncPc.
y[0]=0;
Sigma=0;
instr=311;
break;
case 311:
y[21]=1;
Sigma=0;
instr=312;
break;
case 312:
y[6]=0;
y[7]=1;
y[8]=1;
y[9]=1;
Sigma=0;
instr=313;
break;
case 313:
y[3]=1;
y[4]=1;
y[5]=1;
Sigma=0;
instr=314;
break;
case 314:
y[2]=1;
Sigma=0;
instr=315;
break;
case 315:
y[0]=1;
y[1]=1;
Sigma=0;
instr=316;
break;
case 316:
y[19]=0;
Sigma=0;
instr=317;
break;
case 317:
y[20]=1;
Sigma=0;
instr=318;
break;
case 318:
//instr=300;
if (vec3.value[0]==1){
y[14]=1;
Sigma=0;
instr=319;
break;
}else{
instr=318;
//instr=300;
break;
}
case 319:
//instr=300;
instr=fase*10;
break;}}
}
Event control4::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma=INF;
return Event(y,0);
}
void control4::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “instrreg”

```
//CPP:vector/instrreg.cpp
#if !defined instrreg_h
#define instrreg_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class instrreg: public Simulator {
// Declare the state,
// output variables
// and parameters
vector24 ent;
//out
double y[1];
double Sigma;
#define INF 1e20
public:
	instrreg(const char *n): Simulator(n) {};
	void init(double, ...);
	double ta(double t);
	void dint(double);
	void dext(Event , double );
	Event lambda(double);
	void Exit();
};
#endif
```

```
#include "instrreg.h"
void instrreg::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//      %Type% is the parameter type
}
double instrreg::ta(double t) {
//This function returns a double.
return Sigma;
}
void instrreg::dint(double t) {
Sigma=INF;
}
void instrreg::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//      'x.value' is the value (pointer to void)
//      'x.port' is the port number
//      'e' is the time elapsed since last transition
ent=*(vector24*)x.value;
if (x.port==0){
if (ent.value[14]<9){
y[0]=1;
Sigma=0;}}
}
Event instrreg::lambda(double t) {
//This function returns an Event:
//      Event(%&Value%, %NroPort%)
//where:
//      %&Value% points to the variable which contains the value.
//      %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void instrreg::Exit() {
//Code executed at the end of the simulation.
```

```
}
```

Archivos .ccp y .h “memori3”

```
//CPP:vector/memori3.cpp
#if !defined memori3_h
#define memori3_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"
class memori3: public Simulator {
// Declare the state,
// output variables
// and parameters
vector16 a,b,c,d,f,g,h,vec,vec1,vec2,vec3;
double memoria [64][16],memoria1[16][32];
double Sigma;
int adress,i;
double y[16];
#define INF 1e20
public:
        memori3(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "memori3.h"
void memori3::init(double t,...) {
y[20]=1;
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
for (int j=0;j<64;j++){
for (int i=0;i<16;i++){
memoria[j][i]=0;
}}
//loadI 1
memoria[0][2] =1;
memoria[0][15] =1;
// 32
memoria[1][10] =1;
//loadI 2
memoria[2][2] =1;
memoria[2][14] =1;
// 16
memoria[3][11] =1;
 //load 6
memoria[4][2] =1;
memoria[4][13] =1;
memoria[4][14] =1;
//36
memoria[5][10] =1;
memoria[5][13] =1;
//load
memoria[6][4] =1;
memoria[6][11] =1;
memoria[6][13] =1;
//store
memoria[7][3] =1;
memoria[7][10] =1;
```

```
memoria[7][15] =1;
//bgt
memoria[8][2] =1;
memoria[8][3] =1;
memoria[8][11] =1;
memoria[8][13] =1;
memoria[8][14] =1;
//
memoria[9][10] =1;
memoria[9][11] =1;
memoria[9][12] =1;
//load
memoria[10][4] =1;
memoria[10][12] =1;
memoria[10][13] =1;
memoria[10][15] =1;
//inc
memoria[11][2] =1;
memoria[11][3] =1;
memoria[11][4] =1;
memoria[11][14] =1;
//inc
memoria[12][2] =1;
memoria[12][3] =1;
memoria[12][4] =1;
memoria[12][15] =1;
//print
memoria[13][0] =1;
memoria[13][1] =1;
memoria[13][2] =1;
memoria[13][13] =1;
memoria[13][15] =1;
// branch
memoria[14][2] =1;
memoria[14][4] =1;
//6
memoria[15][13] =1;
memoria[15][14] =1;
//elementos a copiar
memoria[16][15] =1;
memoria[17][14] =1;
memoria[18][14] =1;
memoria[18][15] =1;
memoria[19][13] =1;
memoria[20][13] =1;
memoria[20][15] =1;
memoria[21][13] =1;
memoria[21][14] =1;
memoria[22][13] =1;
memoria[22][14] =1;
memoria[22][15] =1;
memoria[23][12] =1;
memoria[24][12] =1;
memoria[24][15] =1;
memoria[25][12] =1;
memoria[25][14] =1;
memoria[26][12] =1;
memoria[26][14] =1;
memoria[26][15] =1;
memoria[27][12] =1;
memoria[27][13] =1;
memoria[28][12] =1;
memoria[28][13] =1;
memoria[28][15] =1;
memoria[29][12] =1;
memoria[29][13] =1;
memoria[29][14] =1;
memoria[30][12] =1;
memoria[30][13] =1;
memoria[30][14] =1;
memoria[30][15] =1;
memoria[31][11] =1;
}
double memori3::ta(double t) {
//This function returns a double.
return Sigma;
}
void memori3::dint(double t) {
```

```
Sigma=INF;
}
void memori3::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
/* Mapeo de puertos
port 0 read/write
port 1 direcciones
port 2 datos
port 3 vma*/
int res1,res2,res3,res4,restot,res11,res12,res13,res14,restot1,oper,resto;
if (x.port==1){
vec1=*(vector16*)x.value;
res4 =
(int)((vec1.value[10]*32)+(vec1.value[11]*16)+(vec1.value[12]*8)+(vec1.value[13]*4)+
(vec1.value[14]*2)+(vec1.value[15]));
i=res4;
//adress
}
if (x.port==2){
//data
vec2=*(vector16*)x.value;
}
if (x.port==0){
//r/w
vec=*(vector16*)x.value;
if (vec.value[0]==1){
//escritura
for (int j=0; j<16;j++){
memoria[i][j]=vec2.value[j];
}
}
if (vec.value[0]==0){
//lectura
for (int k=0;k<16;k++){
y[k]=memoria[i][k];}
//y[k]=i;}
//y[16]=1;
Sigma=0;
}}
}
Event memori3::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma=0;
return Event(y,0);
}
void memori3::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “memori5”

```
//CPP:vector/memori5.cpp
#if !defined memori5_h
#define memori5_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"
class memori5: public Simulator {
// Declare the state,
// output variables
// and parameters
vector16 vec,vec1,vec2;
double memoria [64][16];
double Sigma;
```

```
int adress,i;
double y[16];
#define INF 1e20
public:
        memori5(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "memori5.h"
void memori5::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//      %Type% is the parameter type
for (int j=0;j<64;j++){
for (int i=0;i<16;i++){
memoria[j][i]=0;
}}
// load en reg 0
memoria[0][2]=1;
// a=2
memoria[1][14]=1;
// load en reg 1
memoria[2][2]=1;
memoria[2][15]=1;
// b=15
memoria[3][12]=1;
memoria[3][13]=1;
memoria[3][14]=1;
memoria[3][15]=1;
// load en reg 2
memoria[4][2]=1;
memoria[4][14]=1;
// c=7
memoria[5][13]=1;
memoria[5][14]=1;
memoria[5][15]=1;
//load en reg 3
memoria[6][2]=1;
memoria[6][14]=1;
memoria[6][15]=1;
// d=10
memoria[7][12]=1;
memoria[7][14]=1;
//load en reg 4
memoria[8][2]=1;
memoria[8][13]=1;
//2
memoria[9][14]=1;
//branchI reg 2---1
memoria[10][0]=1;
memoria[10][2]=1;
memoria[10][3]=1;
memoria[10][4]=1;
memoria[10][11]=1;
memoria[10][15]=1;
//24
memoria[11][11]=1;
memoria[11][12]=1;
// d=d-a;
memoria[12][1]=1;
memoria[12][2]=1;
memoria[12][3]=1;
memoria[12][14]=1;
memoria[12][15]=1;
```

```
// loadI
memoria[13][2]=1;
memoria[13][13]=1;
memoria[13][15]=1;
//0
//branchI if equal
memoria[15][0]=1;
memoria[15][2]=1;
memoria[15][3]=1;
memoria[15][4]=1;
memoria[15][10]=1;
memoria[15][12]=1;
memoria[15][13]=1;
// 21
memoria[16][11]=1;
memoria[16][13]=1;
memoria[16][15]=1;
//print d
memoria[17][1]=1;
memoria[17][2]=1;
memoria[17][4]=1;
memoria[17][11]=0;
memoria[17][14]=1;
memoria[17][15]=1;
//f++
memoria[18][2]=1;
memoria[18][3]=1;
memoria[18][4]=1;
memoria[18][13]=1;
memoria[18][15]=1;
//branchI
memoria[19][2]=1;
memoria[19][4]=1;
//15
memoria[20][12]=1;
memoria[20][13]=1;
memoria[20][14]=1;
memoria[20][15]=1;
//inc c
memoria[21][2]=1;
memoria[21][3]=1;
memoria[21][4]=1;
memoria[21][14]=1;
//branchI
memoria[22][2]=1;
memoria[22][4]=1;
//10
memoria[23][12]=1;
memoria[23][14]=0;
// print d
memoria[24][0]=1;
memoria[24][1]=1;
memoria[24][2]=1;
memoria[24][14]=1;
memoria[24][15]=1;
//end
memoria[25][0]=1;
memoria[25][1]=1;
memoria[25][2]=1;
memoria[25][3]=1;
memoria[25][5]=1;
}
double memori5::ta(double t) {
//This function returns a double.
return Sigma;
}
void memori5::dint(double t) {
Sigma=INF;
}
void memori5::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
/* Mapeo de puertos
port 0 read/write
port 1 direcciones
```

```
port 2 datos
port 3 vma*/
int res4;
if (x.port==1){
vec1=*(vector16*)x.value;
res4 =
(int)((vec1.value[10]*32)+(vec1.value[11]*16)+(vec1.value[12]*8)+(vec1.value[13]*4)+
(vec1.value[14]*2)+(vec1.value[15]));
i=res4;
//adress
}
if (x.port==2){
//data
vec2=*(vector16*)x.value;
}
if (x.port==0){
//r/w
vec=*(vector16*)x.value;

if (vec.value[0]==1){
//escritura
for (int j=0; j<16;j++){
memoria[i][j]=vec2.value[j];
}
}
if (vec.value[0]==0){
//lectura
for (int k=0;k<16;k++){
y[k]=memoria[i][k];}
//y[k]=i;}
//y[16]=1;
Sigma=0;
}}
}
Event memori5::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma=0;
return Event(y,0);
}
void memori5::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “memori6”

```
//CPP:vector/memori6.cpp
#if !defined memori6_h
#define memori6_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"
class memori6: public Simulator {
// Declare the state,
// output variables
// and parameters
vector16 vec,vec1,vec2;
double memoria [64][16];
double Sigma;
int adress,i;
double y[16];
#define INF 1e20
public:
        memori6(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
```

```
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "memori6.h"
void memori6::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
for (int j=0;j<64;j++){
for (int i=0;i<16;i++){
memoria[j][i]=0;
}}
// load en reg 0
memoria[0][2]=1;
// a=178
memoria[1][8]=1;
memoria[1][10]=1;
memoria[1][11]=1;
memoria[1][14]=1;
// load en reg 1
memoria[2][2]=1;
memoria[2][15]=1;
// b=409
memoria[3][7]=1;
memoria[3][8]=1;
memoria[3][11]=1;
memoria[3][12]=1;
memoria[3][15]=1;
// move reg 0 a reg 2
memoria[4][3]=1;
memoria[4][4]=1;
memoria[4][14]=1;
// c=b+a
memoria[5][1]=1;
memoria[5][2]=1;
memoria[5][4]=1;
memoria[5][12]=1;
memoria[5][14]=1;
//print c
memoria[6][0]=1;
memoria[6][1]=1;
memoria[6][2]=1;
memoria[6][14]=1;
// move reg 0 a reg 2
memoria[7][3]=1;
memoria[7][4]=1;
memoria[7][14]=1;
// c= a and b;
memoria[8][1]=1;
memoria[8][4]=1;
memoria[8][12]=1;
memoria[8][14]=1;
//print c
memoria[9][0]=1;
memoria[9][1]=1;
memoria[9][2]=1;
memoria[9][14]=1;
// move reg 0 a reg 2
memoria[10][3]=1;
memoria[10][4]=1;
memoria[10][14]=1;
// c= a 0r b;
memoria[11][1]=1;
memoria[11][3]=1;
memoria[11][12]=1;
memoria[11][14]=1;
//print c
```

```
memoria[12][0]=1;
memoria[12][1]=1;
memoria[12][2]=1;
memoria[12][14]=1;
// move reg 0 a reg 2
memoria[13][3]=1;
memoria[13][4]=1;
memoria[13][14]=1;
// c= a xor b;
memoria[14][1]=1;
memoria[14][3]=1;
memoria[14][4]=1;
memoria[14][12]=1;
memoria[14][14]=1;
//print c
memoria[15][0]=1;
memoria[15][1]=1;
memoria[15][2]=1;
memoria[15][14]=1;
// move reg 0 a reg 2
memoria[16][3]=1;
memoria[16][4]=1;
memoria[16][14]=1;
// c= shift left a;
memoria[17][0]=1;
memoria[17][1]=1;
memoria[17][3]=1;
memoria[17][14]=1;
//print c
memoria[18][0]=1;
memoria[18][1]=1;
memoria[18][2]=1;
memoria[18][14]=1;
// move reg 0 a reg 2
memoria[19][3]=1;
memoria[19][4]=1;
memoria[19][14]=1;
// c= shift right a;
memoria[20][0]=1;
memoria[20][1]=1;
memoria[20][3]=1;
memoria[20][4]=1;
memoria[20][14]=1;
//print c
memoria[21][0]=1;
memoria[21][1]=1;
memoria[21][2]=1;
memoria[21][14]=1;
// move reg 0 a reg 2
memoria[22][3]=1;
memoria[22][4]=1;
memoria[22][12]=1;
memoria[22][14]=1;
// c=b-a;
memoria[23][1]=1;
memoria[23][2]=1;
memoria[23][3]=1;
memoria[23][14]=1;
//print c
memoria[24][0]=1;
memoria[24][1]=1;
memoria[24][2]=1;
memoria[24][14]=1;
//end
memoria[25][0]=1;
memoria[25][1]=1;
memoria[25][2]=1;
memoria[25][3]=1;
memoria[25][4]=1;
}
double memori6::ta(double t) {
//This function returns a double.
return Sigma;
}
void memori6::dint(double t) {
Sigma=INF;
}
void memori6::dext(Event x, double t) {
```

```
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
/* Mapeo de puertos
port 0 read/write
port 1 direcciones
port 2 datos
port 3 vma*/
int res4;
if (x.port==1){
vec1=*(vector16*)x.value;
res4 =
(int)((vec1.value[10]*32)+(vec1.value[11]*16)+(vec1.value[12]*8)+(vec1.value[13]*4)+
(vec1.value[14]*2)+(vec1.value[15]));
i=res4;
//adress
}
if (x.port==2){
//data
vec2=*(vector16*)x.value;
}
if (x.port==0){
//r/w
vec=*(vector16*)x.value;

if (vec.value[0]==1){
//escritura
for (int j=0; j<16;j++){
memoria[i][j]=vec2.value[j];
}
}
if (vec.value[0]==0){
//lectura
for (int k=0;k<16;k++){
y[k]=memoria[i][k];}
//y[k]=i;}
//y[16]=1;
Sigma=0;
}}
}
Event memori6::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma=0;
return Event(y,0);
}
void memori6::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “opregsel”

```
//CPP:vector/opregsel.cpp
#if !defined opregsel_h
#define opregsel_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class opregsel: public Simulator {
// Declare the state,
// output variables
// and parameters
//out
double y[1];
double Sigma;
#define INF 1e20
public:
```

```
        opregsel(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "opregsel.h"
void opregsel::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
}
double opregsel::ta(double t) {
//This function returns a double.
return Sigma;
}
void opregsel::dint(double t) {
Sigma=INF;
}
void opregsel::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
vector24 ent;
ent=*(vector24*)x.value;
if (ent.value[13]<10){
y[0]=ent.value[13];
Sigma=0;}
}
Event opregsel::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void opregsel::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “outreg”

```
//CPP:vector/outreg.cpp
#if !defined outreg_h
#define outreg_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class outreg: public Simulator {
// Declare the state,
// output variables
// and parameters
vector24 ent;
//out
double y[1];
double Sigma;
#define INF 1e20
public:
        outreg(const char *n): Simulator(n) {};
```

```
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "outreg.h"
void outreg::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
}
double outreg::ta(double t) {
//This function returns a double.
return Sigma;
}
void outreg::dint(double t) {
Sigma=INF;
}
void outreg::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
ent=*(vector24*)x.value;
if (ent.value[2]<10){
y[0]=ent.value[2];
Sigma=0;}
}
Event outreg::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void outreg::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “ready”

```
//CPP:vector/ready.cpp
#if !defined ready_h
#define ready_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
#include "vector/vector16.h"
class ready: public Simulator {
// Declare the state,
// output variables
// and parameters
vector24 ent;
//out
double y[1];
double Sigma;
#define INF 1e20
public:
```

```
        ready(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "ready.h"
void ready::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
}
double ready::ta(double t) {
//This function returns a double.
return Sigma;
}
void ready::dint(double t) {
Sigma=INF;
}
void ready::dext(Event x, double t) {
vector16 vec;
if (x.port==1){
vec=*(vector16*)x.value;
if (vec.value[0]==1){
y[0]=1;
Sigma=0;}}
}
Event ready::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void ready::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “regcont”

```
//CPP:vector/regcont.cpp
#if !defined regcont_h
#define regcont_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector16.h"
#include "vector/vector24.h"
class regcont: public Simulator {
// Declare the state,
// output variables
// and parameters
//out
vector24 vec;
double y[1];
double Sigma;
#define INF 1e20
public:
        regcont(const char *n): Simulator(n) {};
        void init(double, ...);
```

```
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "regcont.h"
void regcont::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//      %Type% is the parameter type
}
double regcont::ta(double t) {
//This function returns a double.
return Sigma;
}
void regcont::dint(double t) {
Sigma=INF;
}
void regcont::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//      'x.value' is the value (pointer to void)
//      'x.port' is the port number
//      'e' is the time elapsed since last transition
vec=*(vector24*)x.value;
if (vec.value[21]<10){
y[0]=vec.value[21];
Sigma=0;}
}
Event regcont::lambda(double t) {
//This function returns an Event:
//      Event(%&Value%, %NroPort%)
//where:
//      %&Value% points to the variable which contains the value.
//      %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void regcont::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “regsal”

```
//CPP:vector/regsal.cpp
#if !defined regsal_h
#define regsal_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class regsal: public Simulator {
// Declare the state,
// output variables
// and parameters
//out
double y[1];
double Sigma;
#define INF 1e20
public:
        regsal(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
```

```
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "regsal.h"
void regsal::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//      %Type% is the parameter type
}
double regsal::ta(double t) {
//This function returns a double.
return Sigma;
}
void regsal::dint(double t) {
Sigma=INF;
}
void regsal::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//      'x.value' is the value (pointer to void)
//      'x.port' is the port number
//      'e' is the time elapsed since last transition
vector24 ent;
ent=*(vector24*)x.value;
if (ent.value[22]==1){
y[0]=ent.value[22];
Sigma=0;}
}
Event regsal::lambda(double t) {
//This function returns an Event:
//      Event(%&Value%, %NroPort%)
//where:
//      %&Value% points to the variable which contains the value.
//      %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void regsal::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “regwr”

```
//CPP:vector/regwr.cpp
#if !defined regwr_h
#define regwr_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class regwr: public Simulator {
// Declare the state,
// output variables
// and parameters
vector24 ent;
//out
double y[1];
double Sigma;
#define INF 1e20
public:
        regwr(const char *n): Simulator(n) {};
        void init(double, ...);
```

```
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "regwr.h"
void regwr::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//      %Type% is the parameter type
}
double regwr::ta(double t) {
//This function returns a double.
return Sigma;

}
void regwr::dint(double t) {
Sigma=INF;
}
void regwr::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//      'x.value' is the value (pointer to void)
//      'x.port' is the port number
//      'e' is the time elapsed since last transition
ent=*(vector24*)x.value;
if (ent.value[18]<10){
y[0]=ent.value[18];
Sigma=0;}
}
Event regwr::lambda(double t) {
//This function returns an Event:
//      Event(%&Value%, %NroPort%)
//where:
//      %&Value% points to the variable which contains the value.
//      %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void regwr::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “resut”

```
//CPP:vector/resut.cpp
#if !defined resut_h
#define resut_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class resut: public Simulator {
// Declare the state,
// output variables
// and parameters
//out
double y[1];
double Sigma;
#define INF 1e20
public:
        resut(const char *n): Simulator(n) {};
        void init(double, ...);
```

```
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "resut.h"
void resut::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//      %Type% is the parameter type
}
double resut::ta(double t) {
//This function returns a double.
return Sigma;
}
void resut::dint(double t) {
Sigma=INF;
}
void resut::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
vector24 ent;
ent=*(vector24*)x.value;
if (ent.value[23]==1){
y[0]=ent.value[23];
Sigma=0;}
}
Event resut::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void resut::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “rw”

```
//CPP:vector/rw.cpp
#if !defined rw_h
#define rw_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class rw: public Simulator {
// Declare the state,
// output variables
// and parameters
vector24 ent;
//out
double y[1];
double Sigma;
#define INF 1e20
public:
        rw(const char *n): Simulator(n) {};
```

```
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "rw.h"
void rw::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//      %Type% is the parameter type
}
double rw::ta(double t) {
//This function returns a double.
return Sigma;
}
void rw::dint(double t) {
Sigma=INF;
}
void rw::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//      'x.value' is the value (pointer to void)
//      'x.port' is the port number
//      'e' is the time elapsed since last transition
ent=*(vector24*)x.value;
if (ent.value[19]<10){
y[0]=ent.value[19];
Sigma=0;}
}
Event rw::lambda(double t) {
//This function returns an Event:
//      Event(%&Value%, %NroPort%)
//where:
//      %&Value% points to the variable which contains the value.
//      %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void rw::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “shiftsel”

```
//CPP:vector/shiftsel.cpp
#if !defined shiftsel_h
#define shiftsel_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class shiftsel: public Simulator {
// Declare the state,
// output variables
// and parameters
vector24 ent;
//out
double y[3];
double Sigma;
#define INF 1e20
public:
```

```
        shiftsel(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "shiftsel.h"
void shiftsel::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//     %Type% is the parameter type
}
double shiftsel::ta(double t) {
//This function returns a double.
return Sigma;
}
void shiftsel::dint(double t) {
Sigma=INF;
}
void shiftsel::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
//     'e' is the time elapsed since last transition
ent=*(vector24*)x.value;
y[0]=ent.value[3];
y[1]=ent.value[4];
y[2]=ent.value[5];
Sigma=0;
}
Event shiftsel::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void shiftsel::Exit() {
//Code executed at the end of the simulation.
}
```

Archivos .ccp y .h “triangular_sci”

```
//CPP:source/triangular_sci.cpp
#if !defined triangular_sci_h
#define triangular_sci_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "string.h"
class triangular_sci: public Simulator {
// Declare the state, output
double sigma;
double y[10];
// variables and parameters
double a,f,dQ,pte;
double sgn;
char* Method;
double state;
```

```
public:
        triangular_sci(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "triangular_sci.h"
void triangular_sci::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
char *fvar=va_arg( parameters, char*);
a=getScilabVar(fvar );
fvar=va_arg( parameters, char*);
f=getScilabVar(fvar );
Method=va_arg(parameters,char*);
fvar=va_arg( parameters, char*);
dQ=getScilabVar(fvar );
sgn=1;
pte=4*a*f;
sigma=0;
state=-a;
y[0]=state;
for(int i=0;i<10;i++){y[i]=0;};
}
double triangular_sci::ta(double t) {
//This function returns a double.
return sigma;
}
void triangular_sci::dint(double t) {
if(strcmp(Method,"QSS")==0){
    if(pte>0){
          if((a-y[0]-dQ)>0){
                sigma=dQ/pte;
                state=state+dQ;
          } else {
                sigma=(a-y[0])/pte;
                state=a;
                pte=-pte;
          };
    } else{
          if((y[0]+a-dQ)>0){
                sigma=-dQ/pte;
                state=state-dQ;
          } else{
                sigma=-(a+y[0])/pte;
                pte=-pte;
                state=-a;
          };
    }
} else{
    sgn=sgn*(-1);
    sigma=1/(2*f);
};
}
void triangular_sci::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//     'x.value' is the value (pointer to void)
//     'x.port' is the port number
}
Event triangular_sci::lambda(double t) {
//This function returns an Event:
//     Event(%&Value%, %NroPort%)
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
if(strcmp(Method,"QSS")==0){
    y[0]=state;
```

```
    y[1]=0;   //pte
} else{
    y[0]=-a*sgn;
    y[1]=pte*sgn;
};
return Event(&y[0],0);
}
void triangular_sci::Exit() {

}
```

Archivos .ccp y .h “vma”

```
//CPP:vector/vma.cpp
#if !defined vma_h
#define vma_h
#include "simulator.h"
#include "event.h"
#include "stdarg.h"
#include "vector/vector24.h"
class vma: public Simulator {
// Declare the state,
// output variables
// and parameters
vector24 ent;
//out
double y[1];
double Sigma;
#define INF 1e20

public:
        vma(const char *n): Simulator(n) {};
        void init(double, ...);
        double ta(double t);
        void dint(double);
        void dext(Event , double );
        Event lambda(double);
        void Exit();
};
#endif
```

```
#include "vma.h"
void vma::init(double t,...) {
//The 'parameters' variable contains the parameters transferred from the //editor.
va_list parameters;
va_start(parameters,t);
//To get a parameter: %Name% = va_arg(parameters,%Type%)
//where:
//      %Name% is the parameter name
//      %Type% is the parameter type
}
double vma::ta(double t) {
//This function returns a double.
return Sigma;
}
void vma::dint(double t) {
Sigma=INF;
}
void vma::dext(Event x, double t) {
//The input event is in the 'x' variable.
//where:
//      'x.value' is the value (pointer to void)
//      'x.port' is the port number
//      'e' is the time elapsed since last transition
ent=*(vector24*)x.value;
if (ent.value[20]<9){
y[0]=1;
Sigma=0;}
}
Event vma::lambda(double t) {
//This function returns an Event:
//      Event(%&Value%, %NroPort%)
```

```
//where:
//     %&Value% points to the variable which contains the value.
//     %NroPort% is the port number (from 0 to n-1)
Sigma = INF;
return Event(y,0);
}
void vma::Exit() {
//Code executed at the end of the simulation.
}
```

Archivo .h “VECTOR16”

```
#ifndef VECTOR16_H
#define VECTOR16_H
class vector16
{
public:
  double value[16];
  int index;
  };
#endif
```

Archivo .h “Vector_24”

```
#ifndef VECTOR24_H
#define VECTOR24_H
class vector24
{
public:
  double value[24];
  int index;
  };
#endif
```