ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DE COMPUTADORES

ANALIZAR Y COMPARAR EL LENGUAJE DE ALTO NIVEL SYSTEMC FRENTE A LOS DE HDL. APLICACIÓN AL DISEÑO DE SISTEMAS DE CÓMPUTO

ANALYZE AND COMPARE HIGH LEVEL LANGUAGE SYSTEMC VERSUS HDL. APPLICATION TO THE DESIGN OF COMPUTER SYSTEMS

Realizado por
Francisco Jiménez Jiménez

Tutorizado por
Rafael Escaño Quero

Departamento
Electrónica

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Diciembre 2014

Fecha defensa:
El Secretario del Tribunal

Resumen: La complejidad de los sistemas actuales de computación ha obligado a los diseñadores de herramientas CAD/CAE a acondicionar lenguajes de alto nivel, tipo C++, para la descripción y automatización de estructuras algorítmicas a sus correspondientes diseños a nivel físico. Los proyectos a realizar se encuadran dentro de una línea de trabajo consistente en estudiar la programación, funcionamiento de los lenguajes SystemC y SystemVerilog, sus herramientas asociadas y analizar cómo se adecuan a las restricciones temporales y físicas de los componentes (librerías, IP's, macro-celdas, etc) para su directa implementación.

En una primera fase, y para este TFG, se estudiarán los componentes que conforman el framework elegido que es SystemC y su inclusión en herramientas de diseño arquitectural. Este conocimiento nos ayudará a entender el funcionamiento y capacidad de dicha herramienta y proceder a su correcto manejo.

Analizaremos y estudiaremos unos de los lenguajes de alto nivel de los que hace uso dicha herramienta. Una vez entendido el contexto de aplicación, sus restricciones y sus elementos, diseñaremos una estructura hardware.

Una vez que se tenga el diseño, se procederá a su implementación haciendo uso, si es necesario, de simuladores. El proyecto finalizará con una definición de un conjunto de pruebas con el fin de verificar y validar la usabilidad y viabilidad de nuestra estructura hardware propuesta.

Palabras claves: SystemC, HDL, VHDL, Verilog, DLX Segmentado

Abstract: The complexity of current computer systems has forced designers of CAD / CAE tools to put high-level languages, C ++ type, for automation and algorithmic description to their corresponding physical layer designs to structures. The projects to be implemented are included in a line of work that involves studying programming, operation of SystemC and SystemVerilog languages, its associated tools and analyze how they adapted to the temporal and physical constraints of the components (libraries, IP's, macro-cells, etc.) for a direct implementation.

In the first phase, and for this TFG, the components that make up the framework that is chosen for inclusion in SystemC and tools for architectural design will be explored. This knowledge will help us understand the functioning and effectiveness of this tool and proceed to their proper management.

We will analyze and study a few of the high-level languages, which use that tool. From here, we will understand the context of the application, its constraints and its elements, so design a hardware structure.

When we get the design, we will proceed to implementation using, if necessary, simulators. The project ends with a definition of a set of tests in order to verify and validate the usability and feasibility of our proposed hardware structure.

Keywords: SystemC, HDL, VHDL, Verilog, Pipelined DLX

ÍNDICE

Capítulo 1: Introducción

1.1 Objetivos del proyecto

- Estudio del lenguaje SystemC y su entorno de programación.
- Investigar la aplicación de este lenguaje para la generación y aplicación de módulos de diseño de sistemas integrados o SOC.
- Diseño de un estructura hardware con dicho lenguaje.
- Implementación y simulación de dicha estructura hardware.
- Compararlos con lenguajes tipo HDL o Verilog.

1.2 Fases del proyecto

Las fases de trabajo se dividen en cinco etapas:

1. Estudio y adecuación de los componentes que conforman la librería SystemC.
2. Análisis y estudio unos de los lenguajes de alto nivel de los que hace uso dicha herramienta.
3. Una vez entendido el contexto de aplicación, sus restricciones y sus elementos:
    1. Diseño de una estructura hardware.
    2. Implementación de dicha estructura.
    3. Simulación.
4. Definición de conjuntos de pruebas y validación.
5. Comparativa con lenguajes tipo VHDL o Verilog.

1.3 Herramientas utilizadas

Como ya fue mencionado, el framework elegido es SystemC (en este trabajo se utiliza la versión 2.3.0 lanzada en Julio de 2012) que en realidad es una librería del lenguaje C++. El entorno de desarrollo para utilizar SystemC es Eclipse CDT que es de dominio público.

La herramienta para visualizar las señales producidas por el modelado de simulación es GTKWave. Es un visualizador de formas de onda basado en GTK+, con el cual se pueden leer archivos tipo Value Change Dump (VCD) y generalmente es usado para depurar modelos de simulación Verilog o VHDL.

También, usaremos un editor de diagramas UML y diagramas de flujo, ya que es una herramienta gráfica apropiada para el diseño de software, tanto para el comportamiento de los componentes modelados, como la topología del sistema, es decir, la forma de conexión entre los componentes.

Por último, todo esto se instalará en una máquina virtual con VirtualBox, con un sistema operativo basado en UNIX como es Ubuntu 10.04, ya que el manejo de los lenguajes C y C++, además de su programación están muy bien integrados en estas distribuciones.

Capítulo 2: Introducción al lenguaje SystemC

En este capítulo se describen las características del lenguaje de descripción de sistemas utilizado en el desarrollo de este TFG: el SytemC. Para ello se ha dividido en seis apartados:

- En el primero se hace una introducción al lenguaje *SystemC* en la que se incluye su historia, la arquitectura y la metodología de diseño.
- En el segundo se describen los tipos de datos del lenguaje *SystemC*.
- En el tercero se describe el núcleo del lenguaje *SystemC* y todos los elementos que lo conforman.
- En el cuarto apartado se describen un conjunto de canales ya incluidos en el lenguaje *SystemC*.
- El quinto apartado está dedicado a la simulación en el lenguaje *SystemC*.
- En el sexto y último se describe el nivel de abstracción utilizado en el desarrollo de este TFG: el *Transaction Level Modeling* o *TLM*.

2.1 ¿Qué es SystemC?

El lenguaje *SystemC* se define como un lenguaje de descripción de sistemas basado en C++. Tiene un conjunto de rutinas y librerías implementadas en C++ que permiten la simulación de procesos concurrentes, comunicaciones en tiempo real y otros procesos presentes en los sistemas. [1]

2.1.1 Un poco de historia

El lenguaje SystemC nace en 1999 como consecuencia de dos proyectos encabezados por los fabricantes de herramientas de CAD (Sinopsis, Infineon, Frontier Design), la Universidad de California en Irving y el IMEC (Inter Universiti Microelectronic Center) cuyo objetivo era la utilización de C++ para el modelado del software y del hardware. [2]

También en el año 1999 nace la OSCI (Open SystemC Initiative) que se encargará del desarrollo del lenguaje y de sus versiones. En la siguiente tabla se resume el desarrollo del lenguaje SystemC llevado a cabo por la OSCI desde 1999:

Fecha	Versión de SystemC	Descripción
Septiembre 1999	0.9	Versión inicial
Febrero 2000	0.9.1	Reparación de errores
Marzo 2000	1.0.0	Flujo de diseño hardware. Primera versión ampliamente usada.
Octubre 2000	1.0.1	Reparación de errores
Febrero 2001	1.0.2	Modelado de tiempo
Agosto 2001	2.0.0	Incluye canales y eventos. Flujo de diseño de sistema.
Abril 2002	2.0.1	Reparación de errores. Segunda versión ampliamente usada.
Septiembre 2005	2.1.0	Versión beta
Julio 2006	2.2.0	Reparación de errores
Julio 2012	2.3.0	Incluye las librerías de TLM

Tabla 2.1 - Historial de las versiones de SystemC

2.1.2 Arquitectura del lenguaje SystemC

En este apartado se hace una introducción a la arquitectura del lenguaje SystemC (se hace una descripción más detallada en los próximos apartados).

La figura 2.1 muestra la arquitectura del lenguaje SystemC, en ella se distinguen tres zonas:

- Una capa central con los bloques sombreados que representan el lenguaje SystemC.
- Una capa inferior que representa el lenguaje C++. El lenguaje SystemC está implementado en C++.
- Una capa superior que representa librerías implementadas en SystemC pero que no pertenecen al lenguaje.

Methodology-Specific Libraries	Layered Libraries
Master/Slave Library, etc	Verification Library Static Dataflow, etc.
Primitive Channels Signal, Mutex, Semaphore, FIFO, etc.	
Core Language Modules Ports Processes Interfaces Channels Events Event-driven simulation	**Data Types** 4-valued Logic type 4-valued Logic Vectors Bits and Bit Vectors Arbitrary Precision Integer Fixed-point types C++ user-defined types
C++ Language Standard	

Figura 2.1 Arquitectura del *SystemC*

Dentro del lenguaje SystemC se distinguen tres bloques: core lenguage, data types y primitive channels. El core lenguage incluye todas las estructuras de datos definidas en el lenguaje SystemC para el modelado de sistemas. En data types se incluyen los tipos de datos del lenguaje SystemC. En primitive channels se incluyen los canales específicos del lenguaje. [1]

En la siguiente figura se muestra un ejemplo de un sistema en SystemC:

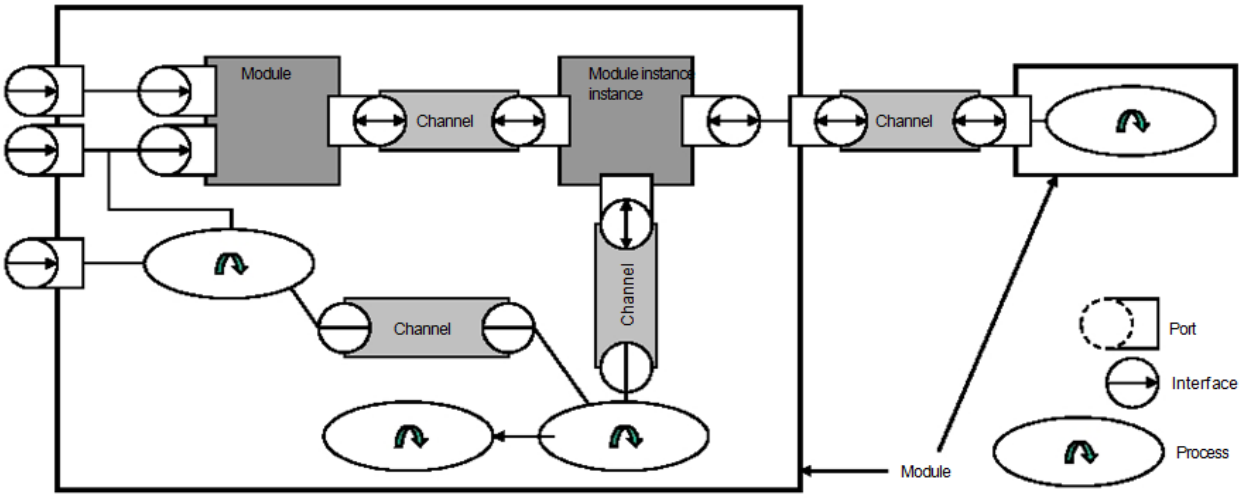


Figura 2.2 Ejemplo de sistema en SystemC

Un sistema en SystemC consta de uno o varios módulos (en el ejemplo de la figura 2.2 hay dos módulos, uno de ellos contiene, a su vez, dos instancias a otros dos módulos), estos módulos describen una estructura y pueden incluir los siguientes elementos:

- Procesos: modelan el funcionamiento de los módulos. Son concurrentes con otros procesos.
- Puertos: son objetos a través de los cuales un módulo se comunica con el exterior.
- Datos internos y canales: utilizados para la comunicación entre distintos procesos dentro de un mismo módulo y para mantener el estado del módulo.
- Instancias de otros módulos: permite la creación de una estructura de jerárquica módulos.

Para comunicar varios procesos de distintos módulos se utiliza la estructura puerto, interfaz y canal. El puerto es el objeto a través del cual el proceso accede a la interfaz de un canal. El interfaz define el conjunto de métodos para acceder al canal y en el canal se implementan estos métodos.

Por último están los eventos, que son los elementos básicos de sincronización, sincronizan procesos y comunicaciones.

2.1.3 Metodología de diseño

En este apartado se va a describir una metodología de diseño a modo de ejemplo. Se parte de una especificación inicial a partir de la cual se desarrolla un modelo funcional ejecutable en SystemC; este es un modelo en el cual las diferentes tareas se llevan a cabo secuencialmente y que se utiliza para realizar una primera evaluación de los algoritmos utilizados. A partir de este modelo se realiza la partición hardware-software, que puede ser manual o asistida por herramientas de co-diseño.

El hardware del sistema consiste en un procesador sobre el que se ejecuta el software y la lógica sobre la que se mapean las tareas que se realizan en hardware.

El banco de test realizado en SystemC permite llevar a cabo el desarrollo y depuración del software a la vez que se progresa en el nivel de abstracción de la descripción del hardware y utilizando un entorno homogéneo de diseño. La síntesis lógica puede llevarse a cabo realizando previamente una descripción RTL en SystemC, VHDL o Verilog o, al menos en teoría, directamente a partir de las descripciones hardware, con alto nivel de abstracción, utilizadas en la especificación funcional.

La ventaja de las metodologías basadas en SystemC estriba en que se utiliza un mismo lenguaje, C++, para describir el hardware y el software en todas las etapas del proceso de diseño, lo cual permite mantener un mismo banco de pruebas sin necesidad de tener que cambiar de nivel de abstracción.

Este planteamiento, que desde el punto de vista teórico es bueno, en la práctica está condicionado por la falta de eficiencia de las herramientas de síntesis lógica para sintetizar descripciones con elevados niveles de abstracción.

La metodología elegida se muestra en la figura 2.3 y está descrita en el recurso [3].

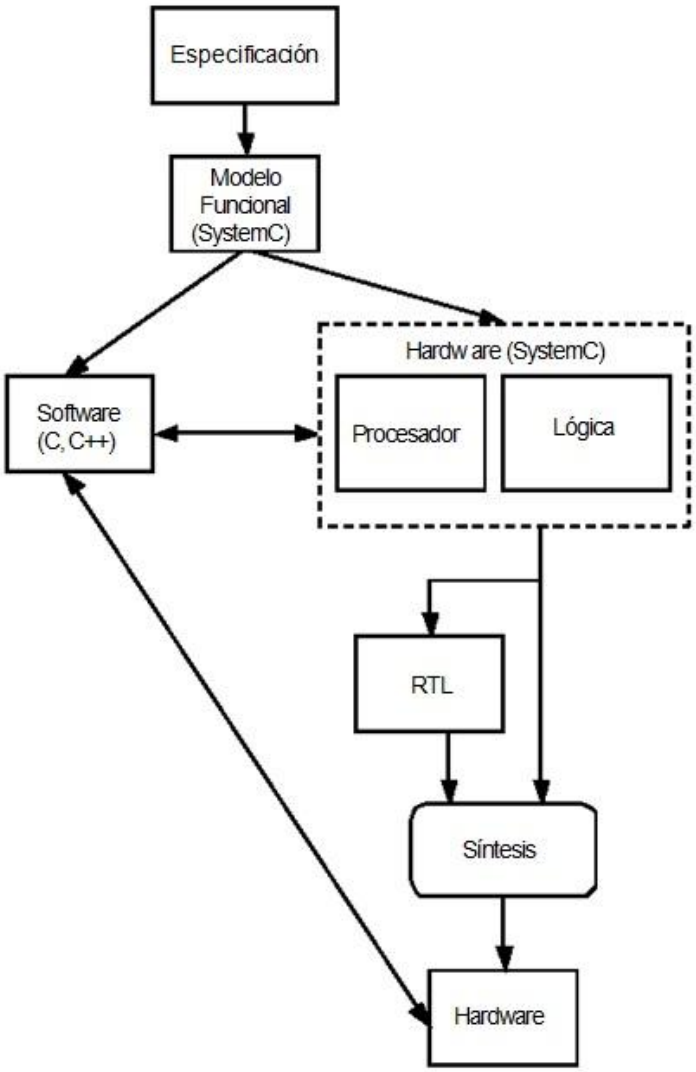
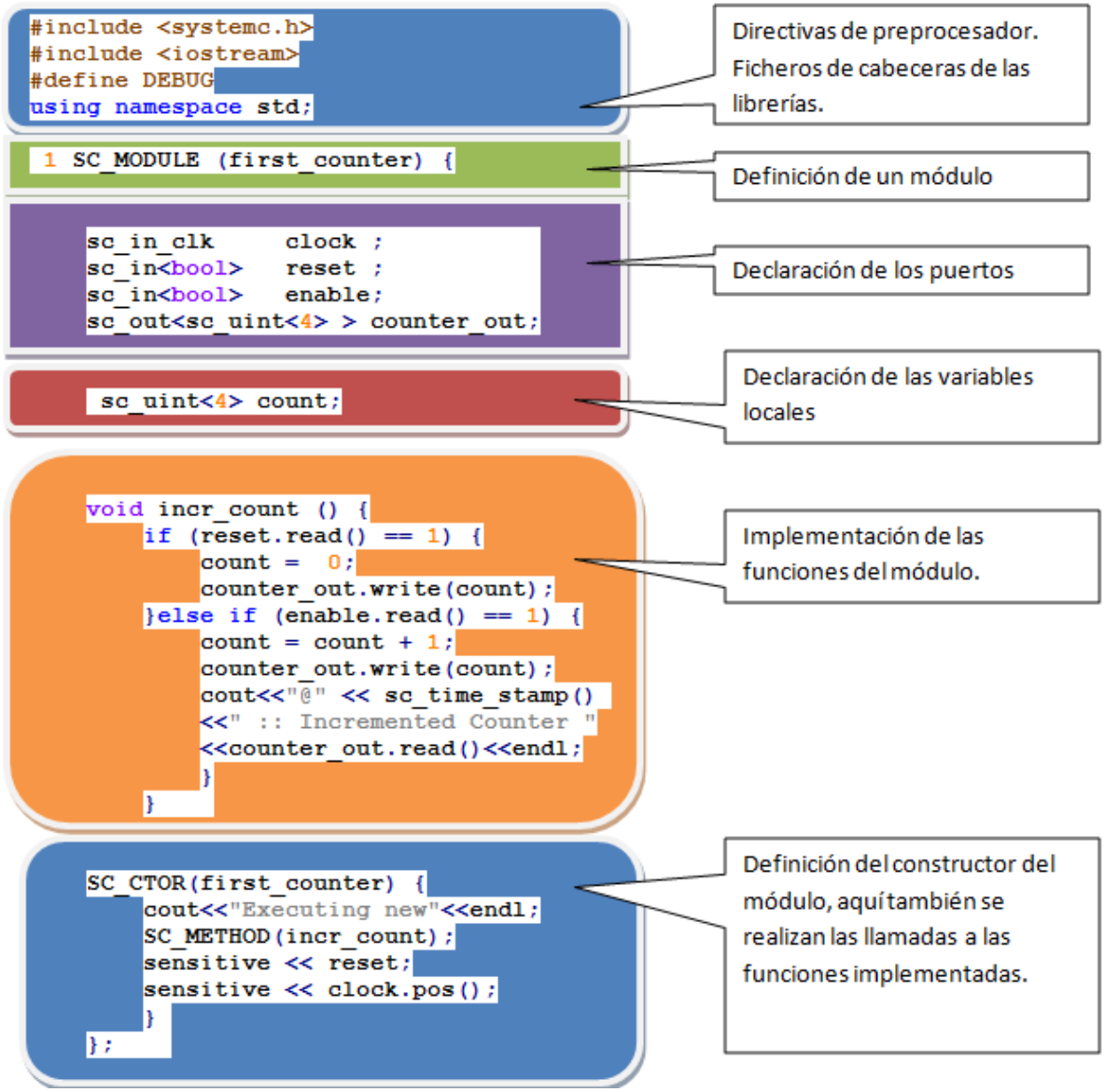


Figura 2.3 Metodología de diseño con SystemC

2.1.4 Ejemplo de un programa en SystemC

Vamos a mostrar un pequeño ejemplo de un programa en SystemC, como primera toma de contacto, no vamos a atender al código en sí, ya que aún no hemos explicado la sintaxis, ni los tipos que existen ni las estructuras que podemos utilizar en SystemC. Lo que vamos hacer es describir los pasos a seguir, como debemos definir y estructurar un fichero que define un módulo, y un fichero principal (main) que ejecuta una simulación:

Fichero que implementa un módulo:



Fichero que implementa un fichero main:

2.2 Tipos de datos

Como se comentó en la introducción, el lenguaje SystemC está implementado en C++, así que todos los tipos de datos disponibles en el lenguaje C++ también están disponibles en el lenguaje SystemC.

Aparte de los tipos de datos de C++, el lenguaje SystemC tiene algunos tipos de datos propios especiales para el modelado de sistemas. En los siguientes apartados se van a describir cada uno de ellos. [4]

2.2.1 Tipos de datos enteros con precisión fija

Su sintaxis es:

```
sc_int<w> // Con signo, se representa en complemento a dos

sc_uint<w> // Sin signo
```

El parámetro "w" indica el número de bits que tiene el entero, su valor tiene que ser mayor que uno y menor o igual a 64.

Se llaman de precisión fija porque todas las operaciones que se realizan con este tipo de datos se hacen en 64 bits y es el resultado de la operación el que se almacena utilizando el número de bits indicados en "w".

2.2.2 Tipos de datos enteros con precisión arbitraria

Su sintaxis es:

```
sc_bigint<w> // Con signo, se representa en complemento a dos

sc_biguint<w> // Sin signo
```

El parámetro "w" indica el número de *bits* que tiene el entero, su valor tiene que ser mayor que uno. Se llaman de precisión arbitraria porque las operaciones se realizan según la precisión determinada en "w".

2.2.3 Bits y vectores de bits

Existen dos tipos de datos para representar los *bits* en *SystemC*:

sc_bit
sc_bgic

La diferencia entre ellos radica en que, en el caso de "sc_bit", sólo puede tomar los valores '0' y '1' lógicos mientras que "sc_logic" puede tomar los valores '0', '1', 'X' y 'Z', estos dos últimos para las situaciones de valor indeterminado y alta impedancia respectivamente. El segundo tipo es menos eficiente en la simulación que el primero.

Para los vectores de *bits* tenemos también dos tipos de datos:

sc_bv<w>
sc_lv<w>

La diferencia entre ambos es la misma que la de "sc_bit" y "sc_logic". El parámetro "w" indica el número de *bits* del vector y tiene que ser mayor que 1. Su precisión es arbitraria.

2.2.4 Tipos de datos con decimales

Los tipos de datos con decimales tienen precisión arbitraria y varios modos de cuantificación y desbordamiento:

sc_fixed <wl, iwl, qmode, o_mode, n_bits> // Con signo

sc_ufixed <wl, iwl, qmode, o_mode, n_bits> // Sin signo

El significado de parámetros es:

- El primer parámetro "wl" indica el número total de *bits*.
- El segundo parámetro "iwl" indica la posición del punto con respecto al *bit* mayor peso. Este parámetro puede ser mayor, igual o menor que cero. Si es positivo indica el número de *bits* de la parte entera.
- El parámetro "qmode" indica el modo de cuantificación (redondeo, truncamiento,…) Este parámetro se tienen en cuenta cuando se realiza una operación aritmética que necesita más *bits* de los que se pueden representar en la parte menos significativa.

- Los parámetros "o_mode" y "n_bits" indican el modo de desbordamiento (saturación, *wrap-around*, …) Estos parámetros se tienen en cuenta cuando se realiza una operación aritmética que necesita más *bits* de los que se pueden representar en la parte más significativa.

También hay tipos de datos con decimales con precisión limitada, cuyo límite de la mantisa es de 53 *bits* y que son más eficientes en la simulación:

sc_fixed_fast <wl,iwl,qmode,o_mode,n_bits> // Con signo

sc_ufixed_fast <wl,iwl,qmode,o_mode,n_bits> // Sin signo

Los parámetros son los mismos que los anteriores.

Existe la posibilidad de crear una variable que contenga los parámetros del tipo y generar varias variables con ella:

sc_fxtype_params config(6, -2, SC_TRN, SC_WRAP);

//6 bits de resolución, dos bits a la derecha del punto, cuantificación por truncamiento //y desbordamiento por wrap around

sc_fix var1(config); // Precisión arbitraria, con signo

sc_ufix var2(config); // Precisión arbitraria, sin signo

sc_fix_fast var3(config); // Precisión limitada, con signo

sc_ufix_fast var4(config); // Precisión limitada, sin signo

2.3 Núcleo del lenguaje SystemC

En este apartado se describen todos los elementos que conforman la estructura y el funcionamiento de un sistema modelado en *SystemC*.

Este apartado se va a divide en seis sub-apartados para describir cada uno de los elementos por separado: módulos; interfaces, canales y puertos; eventos; procesos; instancias a otros módulos y otros elementos. [4]

2.3.1 Módulos

El módulo es el bloque básico de estructura en *SystemC*. Un módulo puede tener los siguientes elementos:

- Constructor (obligatorio)
- Puertos de comunicación
- Variables
- Canales
- Procesos
- Funciones
- Instancias a otros módulos

2.3.1.1 Estructura de un módulo

Un nuevo módulo se crea haciendo una clase que hereda públicamente de la clase "sc_module":

```
class alusc: public sc_module{
// Contenido del módulo
};
```

También se puede utilizar la macro "SC_MODULE", esta macro tiene un único parámetro que es el nombre del módulo:

```
SC_MODULE (alusc){
// Contenido del módulo
};
```

2.3.1.2 Constructor del módulo

Todos los módulos necesitan un constructor. En dicho constructor se inicializan los puertos, las variables y los canales internos, las instancias a otros módulos y otros objetos presentes en el módulo. También se utiliza para declarar las funciones que son procesos y su lista de sensibilidad. Existen tres formas de declarar un constructor:

- La macro "SC_CTOR", esta macro es que solo admite un parámetro de tipo "sc_module_name" que contiene el nombre del módulo:

```
SC_MODULE(alusc){
    //Constructor
    SC_CTOR(alusc){
    // Contenido del constructor
    }
};
```

- También se puede declarar el constructor de forma explícita para que tenga más parámetros:

```
class alusc: public sc_module, public proc_if{
    // Variables, puertos, funciones, etc.
    unsigned int long_img; unsigned int
    alto_img;

    // Constructor
    alusc(sc_module_name name, unsigned int lng_img,
                unsigned int alt_img): sc_module(name),
                                                long_img(lng_img),
                                                alto_img(alt_img){
    // Contenido del constructor
    }
};
```

Siempre es necesario que haya un parámetro del tipo "sc_module_name" que contenga el nombre del módulo.

- Si, además de tener varios parámetros en el constructor, el módulo tiene algún proceso es necesario utilizar la macro "SC_HAS_PROCESS":

```
SC_MODULE (cam){
    // Variables, puertos, funciones, etc.
    unsigned int long_img
    unsigned int alto_img;


    // Constructor

    SC_HAS_PROCESS(cam);
    cam(sc_module_name name,
            unsigned int lng_img,
            unsigned int alt_img): sc_module(name),
                                                long_img(lng_img),
                                                alto_img(alt_img){
    // Contenido del constructor
    }
};
```

Tanto "SC_CTOR" como "SC_HAS_PROCESS" declaran símbolos especiales para usar con las macros que declaran procesos "SC_THREAD" y "SC_METHOD".

2.3.2 Interfaces, canales y puertos

Los elementos básicos para la comunicación entre módulos son los interfaces canales y puertos. Un interfaz define un conjunto de funciones o métodos implementados en un canal y a los que se accede a través de los puertos. En los próximos apartados se describen con más detalle estos tres elementos.

2.3.2.1 Interfaces

El interfaz define un conjunto de funciones o métodos. Estas funciones no se implementan en el interfaz, eso se hará en el canal. En el interfaz se especifica el nombre de la función, sus parámetros y el tipo del valor devuelto. En el ejemplo se muestra una interfaz de una memoria:

```
class mem_if : public virtual sc_interface{

  public:

//Función virtual de lectura de la memoria, la función
//estará implementada en el módulo de memoria
//virtual char read(char *dato, unsigned long dir) = 0;

//Función virtual de escritura en la memoria, la función
//estará implementada en el módulo de memoria
//virtual char write(char *dato, unsigned long dir) = 0;

};
```

2.3.2.2 Canales

Los canales son las estructuras en las que se implementan las funciones definidas en los interfaces. Son las vías de comunicación entre módulos o entre procesos dentro de un mismo módulo.

Un canal puede implementar uno o más interfaces y, a su vez, un mismo interfaz puede ser implementado por varios canales.

Existen dos tipos de canales, los primitivos y los jerárquicos.

2.3.2.2.1 Canales primitivos

Los canales primitivos son canales que heredan de la clase base "sc_prim_channel".
A diferencia del resto de estructuras de *SystemC* estos canales soportan un método especial de petición - actualización implementado en dos funciones:

- "request_update()": es una función no virtual que se llama durante la fase d evaluación de un ciclo delta[1]. Esta función hace que el simulador incluya al canal en la fila de actualización.
- "update()": es una función virtual que tiene que implementarse en el canal. Durante la fase de actualización del ciclo delta[1], el simulador coge todos los canales de la fila de actualización y llama al método "update()" de cada uno de ellos.

[1]. En el simulador de SystemC existe el mismo concepto del ciclo delta que en los simuladores de VHDL. Para más información véase apartado 2.5.2.2.1

En el apartado 2.4 se describen los canales primitivos implementados en la librería del lenguaje SystemC.

2.3.2.2.2 Canales jerárquicos

Los canales jerárquicos se implementan como si fuesen módulos. Estos canales pueden incluir puertos, instancias a otros módulos, otros canales y procesos. El canal tiene apariencia de módulo. Esta estructura tiene una gran flexibilidad en la definición de un canal en comparación con los canales primitivos.

2.3.2.3 Puertos

Los puertos son objetos que ofrecen al módulo un medio de conexión y comunicación con el exterior. A través de un puerto, un módulo se puede conectar con uno o varios canales.

Todos los puertos heredan de la clase base "sc_port":

sc_port<IF,N> port_name;

La clase "sc_port" es una plantilla con dos parámetros:

- IF: interfaz a la que se puede conectar el puerto.
- N: número entero opcional que determina el número máximo de canales a los que se puede conectar el puerto.

El valor por defecto de N es 1, cuando toma ese valor el puerto se puede conectar a un único canal, se define como puerto simple. Si N es mayor que 1 el puerto se puede conectar con varios canales (sin superar el valor de N) y se denomina entonces multipuerto. Si N es igual a 0, el puerto se puede conectar a cualquier número de canales.

Existen dos operadores fundamentales en el manejo de los puertos:

- Operador "->": sirve para llamar a una función definida en la interfaz asociada al puerto:

```
// Declaración de puerto
sc_port<sc_signal_in_if<int> > a
```

```
// Llamada a la función "read" del canal unido al puerto "a"
a->read();
```

- Operador "[]": se utiliza para el acceso individual en un multipuerto:

```
// Declaración de puerto
sc_port<sc_signal_in_if<int>,2> a
// Llamada a la función "read" del segundo canal unido al
// puerto "a"
a[1]->read();
// Llamada a la función "read" del primer canal unido al
// puerto "a"
a[0]->read(); // También a->read();
```

2.3.3 Eventos

Un evento es un objeto de la clase "sc_event" que se utiliza para representar una situación que puede ocurrir durante la simulación. Esta situación se puede utilizar para determinar la ejecución de un proceso:

```
sc_event  nombre_mi_evento; // Declaración de evento
```

Los eventos son los elementos básicos en la sincronización de procesos, la notificación de un evento provoca que el *kernel* de *SystemC* ejecute el código de un proceso que sea sensible a ese evento.

Antes de entrar a describir más en detalle las características de los eventos, es necesario introducir un concepto nuevo, el ciclo delta. El ciclo delta consiste en la ejecución de dos de las fases de la simulación: la de evaluación (en la que se ejecutan los procesos) y la de actualización (en la que se actualizan los valores de los canales) Para más información, véase el apartado 2.5.2.2.1.

2.3.3.1 Ocurrencia de un evento

Hay que distinguir entre evento y ocurrencia, un evento es un objeto que está asociado con el cambio de estado en algún proceso, la ocurrencia de un evento es el hecho de que suceda ese cambio. De esta manera puede haber varias ocurrencias de un mismo evento y estas son únicas.

El módulo propietario del evento es el responsable de informar sobre los cambios en el evento y el evento es el responsable de mantener la lista de procesos sensibles a él. Cuando se notifique, el evento informará al simulador sobre qué proceso o procesos debe ejecutar.

2.3.3.2 Notificación de eventos

Para notificar lo eventos se utiliza la función "notify()", dependiendo de los parámetros que se utilicen en su llamada se puede notificar un evento de tres maneras:

- Notificación inmediata: provoca la ejecución de los procesos sensibles al evento en la fase de evaluación del ciclo delta actual. Una llamada a la función sin argumentos hace una notificación inmediata:

```
sc_event nombre_mi_evento;
nombre_mi_evento.notify(); // Notificación inmediata
```

- Notificación retrasada un ciclo delta: provoca la ejecución de los procesos sensibles al evento en la fase de evaluación del próximo ciclo delta. Una llamada a la función con un cero como argumento provoca una notificación retardada un ciclo delta:

```
sc_event nombre_mi_evento;
nombre_mi_evento.notify(0, SC_NS); // Retrasada un ciclo delta
nombre_mi_evento.notify(SC_ZERO_TIME); // Retrasada un ciclo delta
```

- Notificación temporizada: provoca la ejecución de los procesos sensibles al evento después del tiempo especificado en la llamada a la función. Una llamada a la función con un periodo de tiempo provoca una notificación temporizada:

```
sc_event nombre_mi_evento;
sc_time t(10, SC_NS);
nombre_mi_evento.notify(t); // Temporizada 10 ns
nombre_mi_evento.notify(200, SC_MS); // Temporizada 200 ms
```

2.3.3.3 Múltiples notificaciones de un evento

Los eventos sólo pueden tener una notificación pendiente, si se hacen varias notificaciones de un mismo evento la más próxima en el tiempo es la que se va a imponer a las demás.

Una notificación inmediata es más próxima en el tiempo que una retrasada un ciclo delta y, a su vez, una notificación retrasada un ciclo delta es más próxima en el tiempo que una temporizada.

2.3.3.4 Cancelar notificaciones

Existe la posibilidad de cancelar una notificación pendiente, para ello se usa la función "cancel()":

```
sc_event a, b, c; // Eventos
sc_time t(10, SC_NS);

a.notify();                    // Notificación inmediata
notify(SC_ZERO_TIME, b); // Retrasada un ciclo delta
notify(t, c);                  // Temporizada 10 ns

// Cancelar las notificaciones
a.cancel(); // Error, no se puede cancelar una notificación
// inmediata (ya se ha producido su efecto)
b.cancel();
c.cancel();
```

2.3.4 Procesos

Los procesos se utilizan para describir la funcionalidad de los módulos. Un proceso es una función incluida en el módulo, esta función se declara como proceso en el kernel del SystemC utilizando una macro especial en el constructor del módulo.

La característica principal de los procesos es que no se llaman directamente desde el código, se invocan a partir de la lista de sensibilidad. La lista de sensibilidad es un conjunto de eventos que provocan la ejecución de un proceso. Durante la fase de inicialización se ejecutan todos los procesos. Existe una función "dont_initialize()" para evitar esta ejecución. La función "dont_initialize()" se puede llamar en el constructor del módulo justo después de la declaración del proceso.

2.3.4.1 Declaración de una función como proceso

Para declarar un proceso se necesita, primero, declarar una función en el módulo que no tenga argumentos y que devuelva el tipo "void", esta función va a ser el proceso del módulo. Después, dentro del constructor se utiliza una de las macros para declarar una función como proceso, la "SC_METHOD" o la "SC_THREAD".

Estas macros tienen como único argumento el nombre de la función:

```
SC_MODULE(alusc){

    void lectura_fifo (void); // Declaración de función
    void escritura_ram (void); // Declaración de función
    SC_CTOR(alusc){
        SC_THREAD(lectura_fifo); // Declaración de proceso
        SC_METHOD(escritura_ram); // Declaración de proceso
    }
};
```

2.3.4.2 Sensibilidad estática

En la clase base "sc_module" hay un objeto llamado "sensitive" de la clase "sc_sensitive". Este objeto se utiliza para declarar la sensibilidad estática de los procesos. La llamada a este objeto se hace después de la declaración del proceso y antes de la declaración del siguiente proceso.

La clase "sc_sensitive" sobrecarga los operadores "()" y "<<" para definir la lista de sensibilidad estática (conjunto de eventos a los que el proceso es sensible).

El operador "()" se utiliza en la notación funcional. Dicho operador tiene un único argumento, el evento al que el proceso es sensible. Si el proceso es sensible a varios eventos, se puede llamar al operador "()" varias veces:

```
SC_MODULE(alusc) {
    sc_event clk_int; // Declaración del evento
    void lectura_fifo (void); // Declaración de función
    SC_CTOR(ivideoh){
        SC_THREAD(lectura_fifo); // Declaración de proceso
        // Declaración de sensibilidad estática
        sensitive(clk_int);
    }
};
```

Otra forma de declarar la sensibilidad estática es con notación tipo streaming. Para ello se usa el operador "<<" que soporta varios eventos.

```
SC_MODULE(alusc) {
    sc_event clk_int, n_reset; // Declaración de eventos
    void lectura_fifo (void); // Declaración de función

    SC_CTOR(alusc){

        SC_THREAD(lectura_fifo); // Declaración de proceso

        // Declaración de sensibilidad estática
        sensitive << clk_int << n_reset;
    }
};
```

2.3.4.2.1 Múltiples procesos en un módulo

Si hay varios procesos en un módulo, hay que declarar la lista de sensibilidad de cada proceso justo después de su declaración como proceso en el constructor.

```
SC_MODULE(alusc){
    sc_event clk_int, n_reset; // Declaración de eventos
    void lectura_fifo (void); // Declaración de función
    void escritura_ram (void); // Declaración de función

    SC_CTOR(alusc){

        SC_THREAD(lectura_fifo); // Declaración de proceso

        // Declaración de sensibilidad estática
        sensitive << clk_int << n_reset;

        SC_METHOD(escritura_ram); // Declaración de proceso

        // Declaración de sensibilidad estática
        sensitive(clk_int);
    }
};
```

2.3.4.3 Tipos de procesos

Existen dos tipos de procesos, los métodos y los hilos. En los siguientes apartados se describen ambos tipos de procesos.

2.3.4.3.1 Métodos

Los métodos se caracterizan por:

- Una vez que se inicia su ejecución, se ejecuta todo el código del método. Cuando termina devuelve el control al kernel del SystemC.
- No mantiene su estado de manera implícita, todas sus variables son locales y pierden su valor cuando el método termina. Si hay que guardar el estado del método hay que utilizar variables globales.
- No se puede suspender su ejecución explícitamente esto es, no puede tener llamadas a la función "wait()".

Una función se declara como método utilizando la macro "SC_METHOD" en el constructor del módulo. Esta macro tiene como único argumento el nombre de la función que se declara como método.

```
SC_MODULE(mi_modulo){
    void mi_metodo (void); // Función del proceso
    SC_CTOR(mi_modulo){
        SC_METHOD(mi_metodo); // Declaración del método
    }
}
```

2.3.4.3.1.1 Sensibilidad dinámica

Los métodos pueden utilizar sensibilidad dinámica, estática o ambas. La sensibilidad dinámica se utiliza con la función "next_trigger". Esta función se puede llamar desde el código del método o desde alguna función llamada por él (otra función del módulo o un método de un canal). La función "next_trigger" no detiene la ejecución del método, indica cual es la condición (evento, lista de eventos o retardo) para que se inicie la próxima ejecución del método.

Si se ejecuta varias veces la función "next_trigger" dentro de un mismo método, la última ejecución determinará la condición de la próxima ejecución del método.

La función "next_trigger" puede tener varios parámetros en su llamada, dependiendo del número de parámetros y del tipo de los mismos tendrá distintos significados:

- Si se llama a la función sin argumentos la próxima ejecución del método vendrá determinada por la lista de sensibilidad:

  ```
  next_trigger();
  ```

- Si se llama a la función con un evento como argumento la próxima ejecución del método vendrá se producirá cuando ocurra el evento:

  ```
  sc_event nombre_mi_evento;
  next_trigger(nombre_mi_evento);
  ```

- Si se llama a la función con un periodo de tiempo como argumento, la próxima ejecución del método será después de que pase el periodo de tiempo:

  ```
  sc_time t(200, SC_NS);
  // Próxima ejecución dentro de 200 ns
  next_trigger(t);
  // Próxima ejecución dentro de 10 ms
  next_trigger(10, SC_MS);
  // Próxima ejecución en el próximo ciclo delta
  next_trigger(0, SC_NS);
  // Próxima ejecución en el próximo ciclo delta
  next_trigger(SC_ZERO_TIME);
  ```

- Para que la próxima ejecución sea cuando ocurra uno de los eventos de una lista hay que llamar a la función con un conjunto de eventos separados por el operador *or*:

  ```
  sc_event evento_1, evento_2, evento_3;
  next_trigger(evento_1|evento_2|mevento_3);
  ```

- Para que la próxima ejecución sea cuando ocurran todos los eventos de una lista hay que llamar a la función con un conjunto de eventos separados por el operador *and*:

  ```
  sc_event evento_1, evento_2, evento_3;
  next_trigger(mevento_1&evento_2&evento_3);
  ```

- Existe la posibilidad de combinar la temporización y los eventos:

```
sc_time t(200, SC_NS);
sc_event evento_1, evento_2, evento_3;

// Próxima ejecución dentro de 200 ns o cuando ocurra el
// evento 1, lo que antes suceda
next_trigger(t, evento_1);

// Próxima ejecución dentro de 10 ns o cuando ocurra uno
// de los eventos, lo que antes suceda
next_trigger(10, SC_NS, evento_1|evento_2|evento_3);

// Próxima ejecución dentro de 50 ms o cuando ocurran
// los tres eventos, lo que antes suceda
next_trigger(50, SC_MS, evento_1&evento_2&evento_3);
```

2.3.4.3.2 Hilos

Los hilos se caracterizan por:

- Sólo se ejecutan una vez, al inicio de la simulación.

- El hilo se ejecuta hasta que aparece la función "wait()", entonces el proceso se suspende. Durante la suspensión, el hilo mantiene su estado de manera implícita. Después de la suspensión, el hilo se ejecuta en el punto donde lo dejó, esto es, en la instrucción siguiente al "wait()".

- Para ejecutar el código de un hilo varias veces es necesario implementar un bucle. Normalmente se implementa un bucle infinito para que el hilo se ejecute continuamente. Si se ejecuta un hilo sin bucle infinito y sin llamadas a la función "wait()", el proceso se ejecuta entero y termina en un único ciclo delta. Si se ejecuta un hilo con bucle infinito pero sin llamadas a la función "wait()", el hilo se ejecutará continuamente en un mismo ciclo delta y no se podrá ejecutar ningún otro proceso.

Una función se declara como hilo utilizando la macro "SC_THREAD" en el constructor del módulo. Esta macro tiene como único argumento el nombre de la función que se declara como hilo.

```
SC_MODULE(mi_modulo){
    void mi_hilo (void); // Función del proceso
    SC_CTOR(mi_modulo){
        SC_THREAD(mi_hilo); // Declaración del hilo
    }
}
```

2.3.4.3.2.1 Sensibilidad dinámica

Los hilos pueden utilizar sensibilidad dinámica, estática o ambas. La sensibilidad dinámica se utiliza con la función "wait()". Esta función se puede llamar desde el código del hilo o desde alguna función llamada por él (otra función del módulo o un método de un canal). La función "wait()" detiene la ejecución del hilo e indica cual es la condición (evento, lista de eventos o retardo) para que continúe con la ejecución del hilo.

La función "wait()" puede tener varios parámetros en su llamada, dependiendo del número de parámetros y del tipo de los mismos tendrá distintos significados:

- Si se llama a la función sin argumentos se continuará con la ejecución del hilo según lo determine la lista de sensibilidad:
  ```
  wait();
  ```

- Si se llama a la función con un evento como argumento se continuará con la ejecución del hilo cuando ocurra el evento:
  ```
  sc_event mi_evento;
  wait(mi_evento);
  ```

- Si se llama a la función con un periodo de tiempo como argumento, se continuará con la ejecución del hilo después de que pase el periodo de tiempo:

  ```
  sc_time t(200, SC_NS);
  // Se continua la ejecución dentro de 200 ns
  wait(t);

  // Se continua la ejecución dentro de 10 ms
  wait(10, SC_MS);
  // Se continua la ejecución en el próximo ciclo delta
  wait(0, SC_NS);
  // Se continua la ejecución en el próximo ciclo delta
  wait(SC_ZERO_TIME);
  ```

- Para continuar con la ejecución cuando ocurra uno de los eventos de una lista de eventos hay que llamar a la función con un conjunto de eventos separados por el operador *or*:

```
sc_event mi_evento_1, mi_evento_2, mi_evento_3;
wait(mi_evento_1|mi_evento_2|mi_evento_3);
```

- Para continuar con la ejecución cuando ocurran todos los eventos de una lista de eventos hay que llamar a la función con un conjunto de eventos separados por el operador *and*:

```
sc_event mi_evento_1, mi_evento_2, mi_evento_3;
wait(mi_evento_1&mi_evento_2&mi_evento_3);
```

- Existe la posibilidad de combinar la temporización y los eventos:

```
sc_time t(200, SC_NS);
sc_event evento_1, evento_2, evento_3;

// Continuar la ejecución dentro de 200 ns o cuando
// suceda el evento 1, lo que antes suceda
wait(t, evento_1);

// Continuar la ejecución dentro de 10 ns o cuando
// suceda uno de los eventos, lo que antes suceda
wait(10, SC_NS, evento_1|evento_2|evento_3);

// Continuar la ejecución dentro de 50 ms o cuando
// sucedan los tres eventos, lo que antes suceda
wait(50, SC_MS, evento_1&evento_2&evento_3);
```

2.3.5 Instancias a otros módulos

Los módulos pueden ser instanciados dentro de otros módulos para crear jerarquía entre ellos. Para ello hay que seguir dos pasos, el primero es declarar el módulo y el segundo es inicializarlo. Si el módulo tiene puertos, también hay que conectar dichos puertos.

En la instanciación del módulo es necesario darle un nombre, este nombre lo utiliza SystemC para asignar un nombre jerárquico de forma automática. Este nombre jerárquico se crea concatenando el nombre del módulo padre con nombre del módulo hijo.

2.3.5.1 Declaración de un módulo

La declaración de un módulo se hace en la definición de clase del módulo padre. Se puede hacer de dos formas, usando punteros o no:

```
SC_MODULE(modulo_padre){

    // Instancia a un módulo hijo sin puntero
    modulo_hijo hijo_sin_puntero;

    // Instancia a un módulo hijo con puntero
    modulo_hijo *hijo_con_puntero;

    // Constructor
    SC_CTOR(modulo_padre){
    }
}
```

2.3.5.2 Inicialización de un módulo

La inicialización de los módulos instanciados se hace en el constructor de la clase del módulo padre. Si se trata de un módulo instanciado sin puntero, se hace en la lista de inicialización del constructor, si está instanciado con puntero, se hace en el cuerpo del constructor utilizando la función "new" para la reserva dinámica de memoria:

```
SC_MODULE(modulo_padre){

    // Instancia a un módulo hijo sin puntero
    modulo_hijo hijo_sin_puntero;

    // Instancia a un módulo hijo con puntero
    modulo_hijo *hijo_con_puntero;

    // Constructor, inicialización sin punteros
    SC_CTOR(modulo_padre):hijo_sin_puntero("hijo_sin_puntero"){

        // Inicialización con puntero
        hijo_con_puntero=new modulo_hijo("hijo_con_puntero");
    }
}
```

Los objetos creados utilizando el comando "new" pueden ser borrados cuando hayan terminado de utilizarse, de esta manera, se evita que estén ocupando memoria cuando ya no se necesitan. Esto se puede hacer en el destructor del módulo:

```
~modulo_padre(void){
    delete(hijo_con_puntero);
}
```

2.3.5.3 Conexión de puertos

La conexión de puertos se hace en el constructor del módulo padre. Los puertos del módulo hijo se pueden conectar a un canal interno o a los puertos del módulo padre. Existen dos formas de conectar los puertos, por nombre o por posición.

2.3.5.3.1 Conexión de puertos por nombre

Este método une un puerto del módulo hijo directamente a un puerto o canal del módulo padre utilizando el nombre para identificarlos:

```
SC_MODULE(modulo_hijo){

    sc_fifo_in<int> p_ent; // Puerto de entrada
    sc_fifo_out<int> p_sal; // Puerto de salida
…}
```

```
SC_MODULE(modulo_padre){

    // Declaración de puertos y canales

    // Puerto de entrada
    sc_fifo_in<int> puerto_entrada;

    // Puerto de salida
    sc_fifo_out<int> puerto_salida;

    // Canal interno
    sc_fifo<int> canal_interno;

    // Instancia a un módulo hijo sin puntero
    modulo_hijo hijo_sin_puntero;

    // Instancia a un módulo hijo con puntero
    modulo_hijo *hijo_con_puntero;

    // Constructor, inicialización sin punteros
    SC_CTOR(modulo_padre):

        hijo_sin_puntero("hijo_sin_puntero"){

        // Inicialización con punteros
        hijo_con_puntero=new modulo_hijo("hijo_con_puntero");

        // Conexión de puertos sin puntero
        hijo_sin_puntero.p_ent(puerto_entrada);
        hijo_sin_puntero.p_sal(canal_interno);

       // Conexión de puertos con puntero
        hijo_con_puntero->p_ent(canal_interno);
        hijo_con_puntero->p_sal(puerto_salida);
    }
}
```

2.3.5.3.2 Conexión de puertos por posición

Otra forma de conectar los puertos del módulo hijo es por posición. En este caso se toma la posición en que se declaran los puertos en el módulo hijo para identificarlos y unirlos a los canales y puertos del módulo padre:

```
SC_MODULE(modulo_hijo){

    sc_fifo_in<int> p_ent; // Puerto de entrada sc_fifo_out<int>
    p_sal; // Puerto de salida
…}
SC_MODULE(modulo_padre){

    // Declaración de puertos y canales
    sc_fifo_in<int> puerto_entrada; // Puerto de entrada
    sc_fifo_out<int> puerto_salida; // Puerto de salida
    sc_fifo<int> canal_interno; // Canal interno

    // Instancia a un módulo hijo sin puntero
    modulo_hijo hijo_sin_puntero;

    // Instancia a un módulo hijo sin puntero
    modulo_hijo *hijo_con_puntero;

    // Constructor, inicialización sin punteros
    SC_CTOR(modulo_padre): hijo_sin_puntero("hijo_sin_puntero"){

        // Inicialización con punteros
        hijo_con_puntero=new modulo_hijo("hijo_con_puntero");

      // Conexión de puertos sin puntero:
      // - p_ent -> puerto_entrada
      // - p_sal -> canal_interno

        hijo_sin_puntero(puerto_entrada, canal_interno);

      // Conexión de puertos con puntero
      // - p_ent -> canal_interno
      // - p_sal -> puerto_salida
        (*hijo_con_puntero)(canal_interno, puerto_salida);
    }
}
```

2.3.6 Variables internas y canales internos

Dentro de un módulo pueden existir otros elementos que aún no se han mencionado: las variables internas y los canales internos.

Las variables internas se utilizan para almacenar valores dentro de un módulo, por ejemplo, el estado en que se quedó un método la última vez que se ejecutó. Se declaran como miembros del módulo:

```
SC_MODULE(modulo_1){
    …
    private:
        char estado;
    …
}
```

Se recomienda declarar estas variables como privadas o protegidas en el módulo para que no se pueda acceder a ellas desde fuera de él.

Existe la posibilidad de utilizar el identificador "static" para que una misma variable sea común a todas las instancias del módulo:

```
SC_MODULE(modulo_1){
    …
    private:
        static char estado;
    …
}
```

Los canales internos se utilizan para comunicar dos instancias a módulos o distintos procesos. El acceso al interfaz se realiza sin necesidad de utilizar puertos:

```
SC_MODULE(modulo_1){
    …
    sc_fifo<int> canal_1; // Canal interno
    inta;
    …
    // Lectura de un entero del canal interno
    a=canal_1.read();
}
```

2.4 Canales primitivos

Como se explicó en el apartado 2.3.2.2.1, los canales primitivos son canales que heredan de la clase base "sc_prim_channel", estos canales se caracterizaban por tener un método especial de petición y actualización implementado en dos funciones.

Antes de entrar a describir más en detalle las características de los canales primitivos, es necesario recordar el concepto del ciclo delta. El ciclo delta consiste en la ejecución de dos de las fases de la simulación: la de evaluación (en la que se ejecutan los procesos) y la de actualización (en la que se actualizan los valores de los canales) Para más información, véase el apartado 2.5.2.2.1.

En los siguientes apartados se van a describir los canales primitivos que incluye el lenguaje SystemC. [4]

2.4.1 Canal "sc_signal"

Este canal implementa la interfaz "sc_signal_inout_if", este tipo de canales admite la lectura desde varios procesos pero sólo permiten la escritura desde un único proceso, tanto la lectura como la escritura siguen el proceso de evaluación - actualización descrita en el apartado 3.3.2.2.1

Existen dos parámetros a tener en cuenta en la descripción del canal:

- Valor actual: se refiere al valor que tiene la instancia del canal.
- Valor nuevo: es el valor que se va a escribir en la instancia del canal.

Para describir el comportamiento del canal, se van a explicar cinco situaciones: inicialización, escritura, escrituras múltiples en un ciclo delta, lectura y lecturas y escrituras simultáneas.

2.4.1.1 Inicialización

El valor inicial del valor actual de la instancia al canal es indefinido y se puede inicializar en la función "sc_main" o en el constructor del módulo que crea el canal.

En el ejemplo se muestra como se generan varios canales tipo "sc_signal":

```
// Canal para datos char
sc_signal<char> can_char;
// Canal para datos "sc_logic"
sc_signal<sc_logic> can_sc_logic;
```

2.4.1.2 Escritura

La escritura se realiza en la fase de evaluación del ciclo delta, si el nuevo valor es distinto al valor actual, se solicita una actualización. Durante la fase de actualización el nuevo valor se escribe en el valor actual y se genera un evento.

Tomando como ejemplo los canales de la inicialización:

```
// Escritura usando la función "write"
can_char.write(10);
// Escritura utilizando el operador "="
can_sc_logic=1;
```

2.4.1.3 Escrituras múltiples en un ciclo delta

Si se hacen varias escrituras en la fase de evaluación de un mismo ciclo delta, la última escritura que se ejecute será la que determinará el nuevo valor que se va a escribir en el valor actual durante la fase de actualización. Como no se puede saber el orden de ejecución de los procesos en un ciclo delta, es imposible determinar cuál será el valor que se va a escribir en el canal.

Tomando como ejemplo los canales de la inicialización:

```
// Proceso 1
can_char.write(10);
// Proceso 2
// El proceso que se ejecute el último dentro
// del ciclo delta será el que escriba en el canal.
can_char=15;
```

2.4.1.4 Lectura

La lectura también se ejecuta durante la fase de evaluación del ciclo delta. Esta función devuelve el valor actual y no borra el dato.

Tomando como ejemplo los canales de la inicialización:

```
char val_char; // Valor leído
sc_logic val_sc_logic; // Valor leído

// Lectura usando la función "read"
val_char=can_char.read();

// Lectura utilizando el operador
val_sc_logic=can_sc_logic;
```

2.4.1.5 Lecturas y escrituras simultáneas

Si durante la fase de evaluación de un ciclo delta hay una lectura y una escritura sobre el mismo canal "sc_signal", la lectura devolverá el valor actual del canal. El nuevo valor de la escritura se escribirá en la fase de actualización y estará disponible para ser leído en la fase de evaluación del siguiente ciclo delta.

Tomando como ejemplo los canales de la inicialización:

```
char val_char; // Valor leído

// Proceso 1
can_char.write(10);
// Se completa la escritura

// Proceso 1
can_char.write(15);

// Proceso 2
val_char=can_char.read();
// val_char tomará el valor 10 puesto que la segunda
// escritura no se ha terminado
```

2.4.2 Canal "sc_buffer"

Este canal implementa la interfaz "sc_signal_inout_if", este tipo de canales admite la lectura desde varios procesos pero sólo permiten la escritura desde un único proceso, tanto la lectura como la escritura siguen un proceso de evaluación - actualización que descrito en el apartado 3.3.2.2.1.

Existen dos parámetros a tener en cuenta en la descripción del canal:

- Valor actual: se refiere al valor que tiene la instancia al canal.
- Valor nuevo: es el valor que se va a escribir en la instancia al canal.

En este caso solo se va a describir la escritura ya que la inicialización, la lectura, la escritura de múltiples elementos en un mismo ciclo delta y la lectura y escritura simultáneas son iguales que en los canales "sc_signal".

2.4.2.1 Escritura

La escritura se realiza en la fase de evaluación del ciclo delta cuando se solicita una actualización. Durante la fase de actualización el nuevo valor se escribe en el valor actual y se genera un evento.

2.4.3 Canal "sc_signal_resolved"

Este canal también implementa la interfaz "sc_signal_inout_if". Se comporta como el canal "sc_signal<sc_logic>" con la diferencia de que este permite la escritura desde varios procesos.

En la descripción de estos canales existen tres parámetros a tener en cuenta:

- Valor actual: es el valor que tiene la instancia al canal.
- Valor nuevo: valor que se desea escribir en el canal.
- Valores previos: son cada uno de los valores que quiere escribir cada uno de los procesos en el canal. En cada instante de tiempo se genera el valor nuevo a partir de todos los valores previos.

En este caso sólo se va a describir la inicialización del canal y la escritura desde distintos procesos ya que en el caso de la lectura, la escritura, las múltiples lecturas en un mismo ciclo delta y las lecturas y escrituras simultáneas se comporta igual que el canal "sc_signal".

2.4.3.1 Inicialización

El valor actual del canal se puede inicializar en la función "sc_main" o en el constructor del módulo que crea el canal, si no se inicializa el valor que toma por defecto es "Log_X".

2.4.3.2 Escritura desde múltiples

Al escribir en un canal "sc_signal_resolved", cada proceso genera un valor previo, a partir de esos valores previos se genera el nuevo valor que se escribirá en el valor actual del canal en la fase de actualización. Para resolver los conflictos que se puedan crear a la hora de generar el nuevo valor a partir de los valores previos, el canal utiliza la siguiente tabla:

Valor	0	1	Z	X
0	Log_0	Log_X	Log_0	Log_X
1	Log_X	Log_1	Log_1	Log_X
Z	Log_0	Log_1	Log_Z	Log_X
X	Log_X	Log_X	Log_X	Log_X

Tabla 2.2 Resolución de conflictos en el canal "sc_signal_resolved"

Donde:

- '0' o "Log_0" indica un cero lógico.
- '1' o "Log_1" indica un uno lógico.
- 'X' o "Log_X" indica un valor indeterminado.
- 'Z' o "Log_Z" indica una alta impedancia.

2.4.4 Canal "sc_signal_rv"

Este canal también implementa la interfaz "sc_signal_inout_if". Se comporta como el canal "sc_signal<sc_lv<W> >" con la diferencia de que este permite la escritura desde varios procesos.

En la descripción de estos canales existen cuatro parámetros a tener en cuenta:

- Valor actual: es el valor que tiene la instancia al canal.
- Valor nuevo: valor que se desea escribir en el canal.
- Valor antiguo: valor anterior que tenía la instancia del canal.
- Valores previos: son cada uno de los valores que quiere escribir cada uno de los procesos en el canal. En cada instante de tiempo se genera el valor nuevo a partir de todos los valores previos.

En este caso sólo se va a describir la inicialización del canal y la escritura desde distintos procesos ya que en el caso de la lectura, la escritura, las múltiples lecturas en un mismo ciclo delta y las lecturas y escrituras simultáneas se comporta igual que el canal "sc_signal".

2.4.4.1 Inicialización

El valor actual del canal se puede inicializar en la función "sc_main" o en el constructor del módulo que crea el canal, si no se inicializa el valor que toma por defecto cada uno de los *bits* es "Log_X".

2.4.4.2 Escritura desde múltiples procesos

Al escribir en un canal "sc_signal_rv", cada proceso genera un valor previo, a partir de esos valores previos se genera el nuevo valor que se escribirá en el valor actual del canal en la fase de actualización. Para resolver los conflictos que se puedan crear a la hora de generar el nuevo valor a partir de los valores previos, el canal utiliza la siguiente tabla para cada uno de los *bits* de los valores previos:

Valor	0	1	Z	X
0	Log_0	Log_X	Log_0	Log_X
1	Log_X	Log_1	Log_1	Log_X
Z	Log_0	Log_1	Log_Z	Log_X
X	Log_X	Log_X	Log_X	Log_X

Tabla 2.3 Resolución de conflictos en el canal "sc_signal_resolved"

Donde:

- '0' o "Log_0" indica un cero lógico.
- '1' o "Log_1" indica un uno lógico.
- 'X' o "Log_X" indica un valor indeterminado.
- 'Z' o "Log_Z" indica una alta impedancia.

2.4.5 Canal "sc_fifo"

El canal "sc_fifo" implementa los interfaces "sc_fifo_in_if" y "sc_fifo_out_if" e implementa el comportamiento de una *FIFO* con un número máximo de elementos establecidos en el momento de su creación. Estos canales sólo se pueden conectar a un puerto de salida (escritura) y a uno de entrada (lectura). Varios procesos pueden leer y escribir sobre un mismo canal.

En la descripción de los canales "sc_fifo" hay que tener en cuenta los siguientes parámetros:

- Elemento se refiere a un valor dentro del canal.
- Número máximo se refiere al número máximo de valores que puede almacenar el canal.

Para describir el comportamiento del canal, se van a ver ocho situaciones: inicialización, escritura con bloqueo, escritura sin bloqueo, múltiples escrituras en un mismo ciclo delta, lectura con bloqueo, lectura sin bloqueo, múltiples lecturas en un mismo ciclo delta y lecturas y escrituras simultáneas.

2.4.5.1 Inicialización

El valor del número máximo de elementos se puede inicializar desde la función "sc_main" o desde el constructor que crea el canal. Si no se inicializa el valor que toma por defecto es de 16. En el siguiente ejemplo se muestra como se crea una FIFO con 5 elementos:

```
// Canal FIFO de 5 elementos tipo char
sc_fifo<char> can_fifo(5);
```

2.4.5.2 Escritura con bloqueo

La escritura se realiza en la fase de evaluación del ciclo delta, en este caso nos podemos encontrar con dos situaciones:

1. La FIFO tiene espacio libre, se solicita una actualización y, en la fase de actualización del ciclo delta, se introduce el nuevo elemento en la FIFO.

2. La FIFO está llena, el método suspende el proceso hasta que la FIFO tenga espacio libre para poder escribir.

Tomando como ejemplo la FIFO de la inicialización:

```
// Escribimos en una FIFO con 4 elementos. Se escribe el 5º dato en la FIFO

can_fifo.write(5);

// Se completa la escritura en el canal, la FIFO está llena. La FIFO tiene 5 datos, el
//proceso se bloquea hasta que se libere espacio en la FIFO

can_fifo.write(6);
```

2.4.5.3 Escritura sin bloqueo

Esta escritura se realiza de la misma manera que la escritura con bloqueo si hay sitio en la *FIFO*, si no hay sitio el proceso no hace nada (no suspende el proceso que realiza la escritura).

Tomando como ejemplo la FIFO de la inicialización:

```
bool ret; // Valor retornado

// Escribimos en una FIFO con 4 elementos
ret=can_fifo.nb_write(5);
// Se escribe el 5º dato en la FIFO y la función devuelve un  "true"

// Se completa la escritura en el canal, la FIFO está llena
ret=can_fifo.nb_write(6);
 // La FIFO tiene 5 datos, no se  puede escribir en la FIFO, la  función devuelve un
"false" y el proceso continua
```

2.4.5.4 Múltiples escrituras en un mismo ciclo delta

Si hay varias escrituras sobre un mismo canal "sc_fifo" en la fase de evaluación de un mismo ciclo delta, todos los valores se introducirán en durante la fase de actualización en el orden en que fueron escritos, no se pierden datos. Como no se puede saber qué proceso se ejecuta primero en un mismo ciclo delta, es imposible saber cuál de los procesos va a escribir antes en la *FIFO* y cuál lo hará después.

Tomando como ejemplo la FIFO de la inicialización:

```
bool ret; // Valor retornado
// Escribimos en una FIFO con 3 elementos

// Proceso 1
can_fifo.write(5); // Se escribe un dato en la FIFO
// Proceso 2
can_fifo.write(6); // Se escribe un dato en la FIFO
// Proceso 3
ret=can_fifo.nb_write(7);
// Se escribe un dato en la FIFO.Se completa la escritura
//en el canal, dos de los tres datos se han guardado en la
//FIFO pero el proceso del  tercer dato se bloqueará
//(función "write") o la función devolverá un "false" (función "nb_write")
```

2.4.5.5 Lectura con bloqueo

La lectura se realiza en la fase de evaluación del ciclo delta, en este caso nos podemos encontrar con dos situaciones:

1. La FIFO tiene datos, con lo que el método devuelve el valor del elemento y solicita una actualización, en la fase de actualización del ciclo delta el elemento se borra de la FIFO.

2. La FIFO está vacía, el método suspende el proceso hasta que la FIFO tenga elementos para leer.

Tomando como ejemplo la FIFO de la inicialización:

```
char dat; // Dato leido
// Leemos de una FIFO con 1 elemento. Se lee el último dato de la FIFO
dat=can_fifo.read();

// Se completa la lectura del canal, la FIFO está vacía
dat=can_fifo.read();
// La FIFO no tiene datos, el proceso se bloquea hasta que haya datos en la FIFO
```

2.4.5.6 Lectura sin bloqueo

Si la FIFO tiene datos, se comporta igual en el caso anterior, si la FIFO está vacía, el método no hace nada (no suspende el proceso).

Tomando como ejemplo la FIFO de la inicialización:

```
bool ret; // Valor retornado
char dat; // Dato leido

// Leemos de una FIFO con 1 elemento
ret=can_fifo.nb_read(dat);
// Se lee el último dato de la FIFO y la función devuelve un "true"

// Se completa la lectura del canal, la FIFO está vacía
ret=can_fifo.nb_read(dat);
// La FIFO está vacía, no se puede leer, la función devuelve un "false" y el proceso
//continua.
```

2.4.5.7 Múltiples lecturas en un mismo ciclo delta

Si se hacen varias lecturas en un mismo ciclo delta, todos los valores serán leídos durante la fase de evaluación en el orden en que hayan sido escritos en la FIFO. Durante la fase de actualización se eliminarán todos los elementos que hayan sido leídos. Como no se puede saber qué proceso se ejecuta primero en un mismo ciclo delta, es imposible saber cuál de los procesos va a leer antes en la FIFO y cuál lo hará después.

Tomando como ejemplo la FIFO de la inicialización:

```
// Valor retornado
bool ret;
// Dato leido
char dat;

// Leemos de una FIFO que tiene tres elementos
// Proceso 1
dat=can_fifo.read(); // Se lee un dato de la FIFO

// Proceso 2
dat=can_fifo.read(); // Se lee un dato de la FIFO

// Proceso 3
ret=can_fifo.nb_read(dat); // Se lee un dato de la FIFO
```

Se completa la lectura del canal, dos de los tres procesos habrán conseguido leer un dato de la FIFO. El tercer proceso se bloqueará (función "read") o continuará con sin leer dato (función "nb_read" que devolverá un "false").

2.4.5.8 Lecturas y escrituras simultáneas

Si se realiza una lectura y una escritura sobre un mismo canal FIFO en el mismo ciclo delta, la lectura leerá el último dato que se escribiese en la FIFO antes de el actual ciclo delta, si la FIFO estaba vacía la lectura bloqueará el proceso o le indicará que la FIFO está vacía y no se puede leer. La escritura incluirá el nuevo dato que estará disponible para su lectura en el siguiente ciclo delta, si la FIFO estaba llena, la lectura bloqueará el proceso o le indicará que la FIFO está llena y no se puede escribir.

Tomando como ejemplo la FIFO de la inicialización:

```
// Dato leido
char dat;

// Proceso 1. Escribe un dato. La FIFO está vacía
can_fifo.write(10);
// Se completa la escritura. Pasa un ciclo delta.

// Proceso 1. Escribe otro dato
can_fifo.write(11);

// Proceso 2. Lee un dato
dat=can_fifo.read();
// "dat" tomará el valor 10, que se borrará de la FIFO y se incluirá el dato 11
```

2.4.6 Canal "sc_mutex

Un canal "sc_mutex" se utiliza para bloquear el acceso a un recurso compartido por distintos procesos. Este canal implementa el interfaz "sc_mutex_if".

Un proceso puede bloquear un mutex y, una vez bloqueado, solo ese proceso puede desbloquear ese mutex.

Si varios procesos intentan bloquear un mutex desbloqueado en el mismo ciclo delta, solo uno tendrá éxito. Como el orden de ejecución de los procesos en un ciclo delta es indeterminado, no se sabe cuál es el proceso que va a conseguir bloquear el mutex. Los procesos que no consigan bloquear el mutex quedarán suspendidos.

Si un proceso intenta bloquear un mutex que ya estaba bloqueado, el proceso se suspenderá. Cuando se desbloquee el mutex, el proceso suspendido volverá a intentar bloquear el mutex. Los procesos suspendidos no tienen garantizado bloquear el mutex si hay más de uno intentándolo de nuevo.

En el siguiente ejemplo se crea un mutex y se muestra su funcionamiento:

```
SC_MODULE(modulo){
    sc_mutex mut;
    …}
}
```

```
// Proceso 1 intenta entrar al mutex
mut.lock(); // Proceso 1 entra en el mutex

// Proceso 2 intenta entrar al mutex
mut.lock(); // Proceso 2 se queda bloqueado

// Proceso 3 intenta entrar al mutex
mut.lock(); // Proceso 3 se queda bloqueado

// Proceso 4 intenta entrar al semáforo
mut.trylock();
// Proceso 4 no entra al mutex pero  sigue su ejecución.

// Proceso 1 libera el mutex
mut.unlock(); // Proceso 2 o proceso 3 entra en el mutex
```

2.4.7 Canal "sc_semaphore"

Es similar a un canal "sc_mutex" con la diferencia de que este permite el acceso a un número determinado de procesos de forma concurrente (el *mutex* solo admite uno).

Un canal "sc_semaphore" se tiene que crear con un valor entero que determine el número de accesos concurrentes al semáforo.

Dentro de la clase "sc_semaphore" existe un valor que se llama "semaphore_value", este valor indica el número de accesos concurrentes disponibles. Cuando el valor de "semaphore_value" es mayor que cero se permite el acceso al semáforo.

Cada vez que un proceso bloquea el semáforo, el valor de "semaphore_value" se reduce en uno, cuando este proceso desbloquea el semáforo, el valor de "semaphore_value" se incrementa en uno. El semáforo no comprueba que el proceso que lo desbloquea sea el mismo que anteriormente lo había bloqueado. Tampoco se comprueba si el valor de "semaphore_value" sea mayor que el número de entradas concurrentes.

Si varios procesos intentan bloquear un semáforo con un valor "semaphore_value" igual a 1 en un ciclo delta, solo uno de ellos tendrá éxito. Como el orden de ejecución de los procesos en un ciclo delta es indeterminado, no se puede saber qué proceso conseguirá bloquear el semáforo. Los procesos que no lo consigan se suspenderán.

Si un proceso intenta bloquear un semáforo con un valor de "semaphore_value" menor a igual a cero, el proceso se suspenderá. Cuando el valor de "semaphore_value" aumente, los procesos suspendidos volverán a intentar bloquear el semáforo. Estos procesos no tienen asegurado el bloquear el semáforo si hay más de un proceso intentándolo.

En el siguiente ejemplo se crea un semáforo y se muestra su funcionamiento:

```
SC_MODULE(modulo){

    sc_semaphore sem;
    …
    SC_CTOR(modulo){
        …
        sem(2); // Semáforo que permite dos entradas
        …
    }
}

    // Proceso 1 intenta entrar al semáforo
    sem.wait(); // Proceso 1 entra en el semáforo

    // Proceso 2 intenta entrar al semáforo
    sem.wait(); // Proceso 2 entra en el semáforo

    // Proceso 3 intenta entrar al semáforo
    sem.wait(); // Proceso 3 se queda bloqueado

    // Proceso 4 intenta entrar al semáforo
    sem.trywait();
    // Proceso 4 no entra al semáforo pero  sigue su ejecución.

    // Proceso 1 libera el semáforo
    sem.post(); // Proceso 3 entra en el semáforo
```

2.5 Simulación en SystemC

Una vez conocidas las estructuras de datos y los tipos de datos de SystemC hay que describir cómo se hace la simulación en SystemC. Para ello se ha dividido este apartado en tres sub-apartados:

- En el primero se describe el modelo de tiempo en SystemC y las funciones para manejar variables de tiempo.
- En el segundo se describen las funciones "main" y "sc_main". En este apartado se va a describir como se hace la elaboración de una simulación en SystemC y las etapas de simulación.
- En el tercero se explica el manejo de los ficheros de trazas que soporta SystemC. [4]

2.5.1 Modelo de tiempo

El modelo de tiempo de *SystemC* es absoluto, tiene valor entero e internamente se representa con un entero sin signo de 64 *bits*. El tiempo empieza en 0 y solo puede aumentar durante la simulación.

Dentro de *SystemC* está el tipo "sc_time" que se utiliza para representar intervalos de tiempo dentro de la simulación, un objeto de este tipo está formado por un valor numérico y una unidad de tiempo, si no se especifica la unidad de tiempo se utiliza la unidad de tiempo por defecto.

La unidad de tiempo por defecto es la que se utiliza cuando se genera una variable de tiempo y no se especifica su unidad. Su valor por defecto es de 1 nanosegundo.

- Modificar el valor de la unidad tiempo por defecto:

  void sc_set_default_time_unit(double val, sc_time_unit utime)

  // Cambiar el tiempo por defecto a 1 us
  sc_set_default_time_unit(1, SC_US);

- Consultar el valor de la unidad de tiempo por defecto:

```
sc_time sc_get_default_time_unit()

// Leer la unidad de tiempo por defecto
sc_time tiempo;
tiempo=sc_get_default_time_unit()
```

Otro parámetro importante es la resolución del tiempo, la resolución del tiempo es la unidad más pequeña de tiempo que se puede representar con los objetos "sc_time". El valor por defecto de la resolución del tiempo es de 1 picosegundo. Existen dos funciones para manejar la resolución del tiempo:

- Modificar la resolución del tiempo:

```
void sc_set_time_resolution(double val, sc_time_unit utime)

// Modificar la resolución del tiempo a 1 ns
sc_set_time_resolution(1,SC_NS);
```

- Consultar la resolución del tiempo:

```
sc_time sc_get_time_resolution()

// Leer la resolución del tiempo
sc_time tiempo;
tiempo=sc_get_time_resolution();
```

Cualquier tiempo que sea inferior a la resolución se aproximará al tiempo dentro de la resolución que sea más próximo.

Existe otra función para saber el tiempo que ha transcurrido (en unidades del tiempo por defecto) desde que se inició la simulación:

```
double sc_simulation_time()

// Leer el tiempo desde que se inició la simulación
double tiempo;
tiempo=sc_simulation_time()
```

2.5.2 Funciones "main()" y "sc_main()"

Como se explicó en el apartado 2.1 de introducción, el lenguaje SystemC está basado en C++. Para poder realizar una simulación se necesita una función "main()" que inicie el programa de simulación.

La función "main()" está implementada dentro de la librería de SystemC, si se desea, se puede implementar una función "main()" propia.

La función "main()" llama a la función "sc_main()", dicha función es el punto de entrada desde la librería de SystemC al código de usuario. El prototipo de la función "sc_main()" es:

```
int sc_main(int argc, char *argv[]);
```

Los argumentos "argc" y "argv" son argumentos típicos y se pasan de la función "main" a la función "sc_main".

Dentro de la función "sc_main" se hacen, normalmente, las siguientes operaciones:

- Elaboración de la simulación que incluye:
    - Inicialización de las variables de tiempo (unidad por defecto y resolución)
    - Instanciación de los módulos de más alto nivel dentro de la jerarquía del modelo e instanciación de los canales de comunicación a este nivel.
- Simulación.
- Eliminación de todos los elementos de memoria. (CLEAN-UP).
- Devolución de un código de estatus.

Todas estas operaciones se describen en los siguientes apartados.

2.5.2.1 Elaboración

La elaboración se define como la ejecución de la función "sc_main" desde su inicio hasta la primera llamada a la función "sc_start".

En este código se inicializan las variables de tiempo tales como la resolución y la unidad por defecto (esto ha de hacerse antes de crear objetos del tipo "sc_time").

También en la elaboración se van a instanciar los módulos de mayor nivel en la jerarquía del modelo y los canales por los que se comunican. Durante este proceso, a través de la jerarquía de módulos se van a instanciar todos los módulos del modelo, cuando un módulo se construye, él mismo construye e inicializa todos los módulos que contiene. La instanciación de un módulo se hace de la misma manera que se describió en los apartados 2.3.5.1 y 2.3.5.2.

Durante la elaboración también es necesario conectar los puertos de los módulos de más alto nivel a los canales, esto se hace de la misma manera que se describió en el apartado 2.3.5.3.

En el siguiente ejemplo, se muestran las sentencias que hay que ejecutar antes del "sc_start" necesarias para la fase de elaboración:

```
int sc_main (int argc, char *argv[])
{
    // Inicialización de las variables de tiempo
    sc_set_time_resolution(500,SC_PS); // Resolución a 1/2 ns

    // Unidad de tiempo por defecto 1 ns
    sc_set_default_time_unit(1,SC_NS);

    // Variables de tiempo
    sc_time int_clk; // Reloj interno a 1 ns

    // Módulos de alto nivel
    alusc *filtro_h;

    // Canales de alto nivel
    sc_fifo<char> canal_alusc(LONG_IMG*ALT_IMG);

    // Inicialización de los módulos
    filtro_h = new alusc("filtro_h", int_clk, LONG_IMG, ALT_IMG);
```

```
    // Conexión de los puertos
    filtro_h->dato_in(canal_alusc);

    // Inicio de simulación
    sc_start(120,SC_MS);

    // Borrado de las variables
    delete filtro_h;

    // Se devuelve el control a la función main
    return (0);
}
```

2.5.2.2 Simulación

La segunda fase dentro de la función "sc_main()" es la simulación, en esta fase es donde realmente se ejecuta el código de los procesos de los módulos del modelo que se ha generado.

El simulador de SystemC es un simulador basado en eventos, esto es, solo se ejecutan procesos cuando sucede algún evento. El simulador se encarga de controlar la temporización y el orden de ejecución de los procesos, maneja las notificaciones de los eventos y las actualizaciones de los canales.

El simulador de SystemC soporta la noción de ciclo delta, un ciclo delta consiste en la ejecución de una fase de evaluación y otra de actualización dentro de la ejecución del modelo. El número de ciclos delta dentro de la unidad de tiempo de simulación puede ser variable.

Como se explicó en el apartado 2.3.4 los procesos en SystemC se ejecutan de una vez, esto quiere decir que, una vez que se inicie un proceso tipo hilo, se ejecutará hasta que haya una llamada a la función "wait()" sin que otro proceso le interrumpa.

De la misma manera, un proceso tipo método ejecutará todo su código sin que ningún otro proceso le interrumpa.

El simulador se inicia con la llamada a la función "sc_start()". La simulación puede tener un tiempo de simulación determinado o correr de forma indefinida (hasta que se acaben los eventos, se llame a la función "sc_stop()" o suceda una excepción).

La simulación consta de 3 fases:

1ª Inicialización

En la inicialización todos los métodos del modelo se ejecutan una vez (a no ser que se haya especificado lo contrario en la creación del proceso, apartado 2.3.4).

2ª Evaluación

En la fase de evaluación se decide que proceso, de los que están disponibles para ejecutarse, se ejecuta. El orden de ejecución de los procesos es indeterminado.

La ejecución de un proceso puede contener notificaciones inmediatas de un evento con lo que nuevos procesos se pueden añadir a la lista de disponibles para ejecutarse en esta fase de evaluación.

La ejecución de los procesos también pueden incluir llamadas a las funciones "request_update()" de los canales primitivos. Estas llamadas dejarán pendiente la ejecución de las funciones "update()" de los canales primitivos dentro de la fase de actualización.

La fase de evaluación se termina cuando todos los procesos que estaban disponibles para ejecutarse se ejecutan.

3ª Actualización

En la fase de actualización se ejecutan todas las funciones "update()" de todos loscanales primitivos cuyas funciones "request_update()" fueron llamadas durante la fase de evaluación.

Si había notificaciones de eventos pendientes con retardo de un ciclo delta, se determinan que procesos son los que están listos para ejecutarse y se vuelve a la fase 2 de evaluación.

Si no hay notificaciones de eventos pendientes con retardo de un ciclo delta pero hay notificaciones pendientes temporizadas, se avanza el tiempo de la simulación hasta la más próxima en el tiempo, se determina que procesos están listos para ejecutarse y se vuelve a la fase 2 de evaluación.

Si no quedan más notificaciones de eventos pendientes la simulación ha terminado y se devuelve el control a la función "sc_main()".

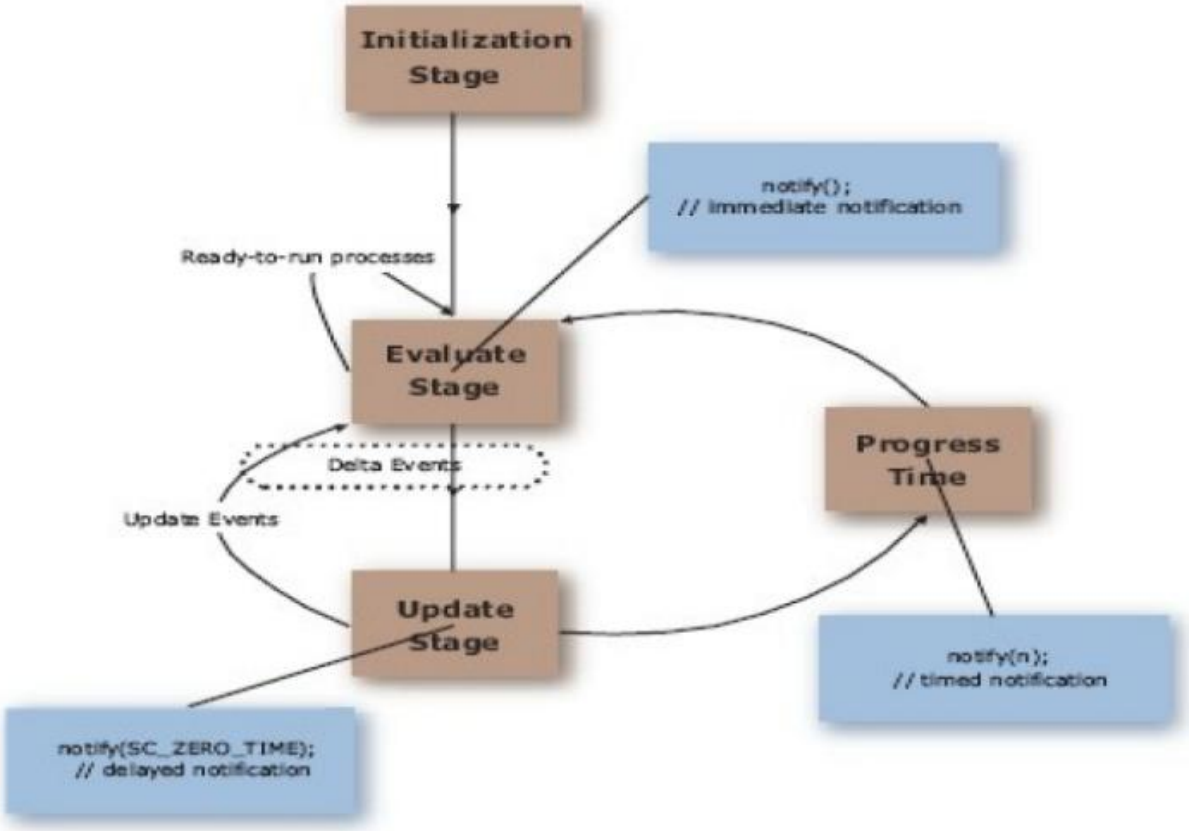


Figura 2.4 Kernel de Simulación de SystemC [7]

2.5.2.2.1 Ciclo delta

Tal y como se ha explicado anteriormente, en SystemC, al igual que en otros lenguajes de diseño de sistemas digitales como VHDL, existe el concepto de ciclo delta.

En SystemC el ciclo delta consiste en la ejecución de las fases de evaluación y actualización explicadas anteriormente:

- Fase de evaluación: en esta fase se ejecutan los procesos que están pendientes de ejecutar, estos procesos pueden, a su vez habilitar la ejecución de otros procesos con eventos de notificación inmediata y llamar a funciones "request_update" de los canales jerárquicos.

- Fase de actualización: en esta fase se ejecutan las funciones "update" de todos los canales jerárquicos a los que se llamó a la función "request_update" en la fase de evaluación. Si hay notificaciones de eventos retardadas un ciclo delta, se determinan qué procesos se deben ejecutar y se vuelve a la fase 2, de esta manera se inicia otro ciclo delta.

Hay que tener en cuenta que cada unidad de tiempo tiene un número indeterminado de ciclos deltas, depende de la ejecución de la ejecución de los procesos y del tipo de notificaciones de los eventos.

2.5.2.2.2 Funciones de control de la simulación

Existen dos funciones para controlar la simulación en *SystemC*:

- "sc_start()" que inicia la simulación, se le puede pasar un valor de tiempo por parámetro para limitar la duración de la simulación. Una vez que termina la simulación se puede volver a llamar a la función "sc_start" para continuar la simulación en el tiempo que se dejó.

- "sc_stop()" esta función termina la simulación y devuelve el control a la función "sc_main()", después de llamar a esta función no se puede volver a llamar a la función "sc_start()" para continuar la simulación.

2.5.2.3 Eliminación de variables y retorno

Una vez que se ha terminado la simulación hay que borrar todos los datos que se hayan creado utilizando la función "new" y hay que devolver el control a la función "main()", un valor devuelto normal es el 0.

2.5.3 Ficheros de trazas

SystemC ofrece la posibilidad de crear ficheros de trazas y de añadir señales para poder verificar el resultado de una simulación. Para ello hay que seguir tres pasos:

1. Crear el fichero de trazas, para ello hay que llamar a la función "sc_create_vcd_trace_file()". Esta función crea el fichero y devuelve un puntero a él. El fichero de trazas puede generarse en la función "sc_main()" o en el constructor de una clase, el único requisito es que se genere antes de registrar las variables que se van a trazar.

2. Registrar las variables que se van a trazar en el fichero, para ello existe la función "sc_trace()". Solamente las variables que se mantengan durante toda la simulación pueden ser registradas (esto excluye a las variables locales) Es posible registrar variables, puertos y algunos canales.

3. Cerrar el fichero, para ello está la función "sc_close_vcd_trace_file()". Esta función ha de llamarse antes de salir de la función "sc_main()"

En el siguiente ejemplo se muestra lo que se ha explicado:

```
SC_MODULE(control){

  public:
        // Señales que se van a mostrar en el fichero de trazas
        sc_in<sc_logic> fin_alusc; // Señal de fin de alusc
        sc_in<sc_logic> fin_dct; // Señal de fin de la DCT
        SC_CTOR(control){
        // Inicializamos las señales de depuración
        s_alusc=0;
        s_dct=0;
```

```
        // Creamos el fichero de trazas
        sc_trace_file *s_control=0;
        s_control=sc_create_vcd_trace_file("control");

        // Señales que vamos a registrar en el fichero de trazas
        sc_trace(s_control,s_alusc,"alusc");
        sc_trace(s_control,s_dct,"dct");
        };
};
```

2.6 Transaction level modeling

En el lenguaje SystemC se pueden implementar modelos en muchos niveles de abstracción, entre ellos está el nivel de abstracción, transaction level modeling o TLM. [5]. Pertenece al nivel de abstracción (AL) Nivel de arquitectura apartado 2.7.

El TLM es una aproximación a un modelado de alto nivel de sistemas donde las comunicaciones entre módulos están separadas de la funcionalidad de los módulos. Las comunicaciones se modelan como canales y se presenta a los módulos como interfaces. Las comunicaciones se realizan llamando a funciones de la interfaz que están modeladas en los canales. Los canales, a su vez, encapsulan los detalles de bajo nivel de las comunicaciones.

En el TLM las comunicaciones se concentran más en la funcionalidad de la comunicación, de dónde a dónde van los datos, que en su implementación exacta.

Esta abstracción permite a los diseñadores implementar distintos modelos de un módulo (todos con el mismo interfaz de comunicación) y probarlos sin necesidad de cambiar el resto del diseño.

Además de poder generar distintos modelos de un módulo y poder probarlos sin cambiar el resto del diseño, el TLM tiene otras ventajas:

- Los diseños son relativamente sencillos de modelar, usar y ampliar.
- En TLM se puede diseñar módulos software y hardware.
- Simulaciones más rápidas ya que se omiten detalles que no se ejecutan en tiempo de simulación.

- Se consiguen módulos con funcionalidad completa muy pronto en el ciclo de diseño, esto permite poder hacer modelos alternativos y poder explorar varias soluciones antes de que sea demasiado tarde para ello.

- Nos permite conseguir módulos lo suficientemente precisos como para poder validar el software antes de tener una implementación hardware más detallada.

En el año 2005, la OSCI publicó un estándar para el modelado a nivel TLM, aunque nuestro objetivo principal no se basa en éste estándar, sino que consiste en explorar las posibilidades del SystemC y no de la librería TLM.

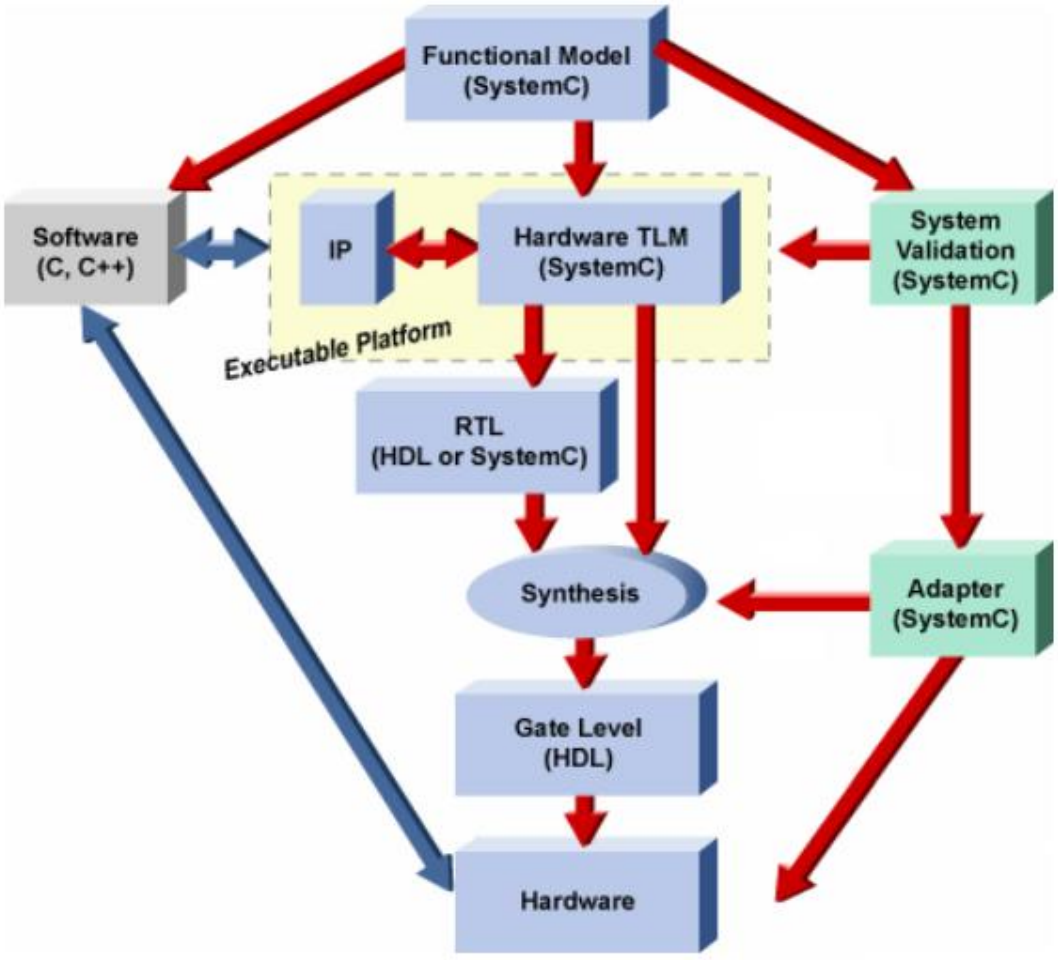


Figura 2.5 Interacción de TLM con otros módulos de SystemC [6]

2.7 Niveles de abstracción en SystemC

Existen varios niveles de abstracción para modelar sistemas. Cada nivel de abstracción maneja distintos tipos de detalles y esconde otros que se consideran sin interés. Así como el nivel de abstracción de los diseñadores se movió desde diseño con células estándares usando técnicas full-custom a lenguajes de descripción hardware usando RTL en los años 90, nuevos métodos de diseño proponen el elevar el nivel de abstracción del diseñador desde RTL hasta nivel de sistema. A nivel de sistema los elementos básicos tienen una mayor granularidad (por ejemplo: bloques IP o subsistemas completos) y están descritos usando lenguajes de modelado de alto nivel como SystemC.

Simulaciones a nivel RTL limitan el tamaño y la complejidad de los sistemas debido a la enorme cantidad de detalles que el diseñador tiene que manejar y las bajísimas prestaciones (en el orden de miles de ciclos por segundo). Varios niveles de abstracción son necesarios para representar los SoC al nivel de detalle requerido para las diferentes actividades que existen en el ciclo de diseño (por ejemplo: si el nivel de detalle es suficientemente detallado para exploración de la arquitectura HW, entonces la velocidad de simulación es demasiada baja para análisis algorítmico). Los tres niveles de abstracción que pueden ser modelados con SystemC son:

- **Nivel funcional (FL).** Este nivel de abstracción solo representa bloques funcionales que componente los algoritmos o aplicaciones. La comunicación entre bloques funcionales se realiza a nivel de mensajes. El nivel de abstracción funcional también se conoce como nivel algorítmico o de procesos comunicantes. A día de hoy no existe un conjunto estándar de interfaces para crear modelos SystemC a este nivel.

- **Nivel de arquitectura (AL).** El nivel de arquitectura tiene como objetivo describir tanto la funcionalidad como la estructura de los bloques IP en una plataforma HW. Los modelos tienen que ser completamente correctos en su funcionalidad, pero los detalles microarquitecturales no tienen por que tenerse en cuenta a este nivel. La comunicación entre los bloques IP se realiza a nivel de transacciones (TL), donde las interfaces representan las interfaces reales de los bloques IP.

- **Nivel de implementación (IL).** Este nivel de abstracción tiene como objetivo el capturar los detalles de implementación de las interfaces y los bloques internos de los bloques IP. La comunicación entre bloques IP se realiza a nivel de señales/pines.

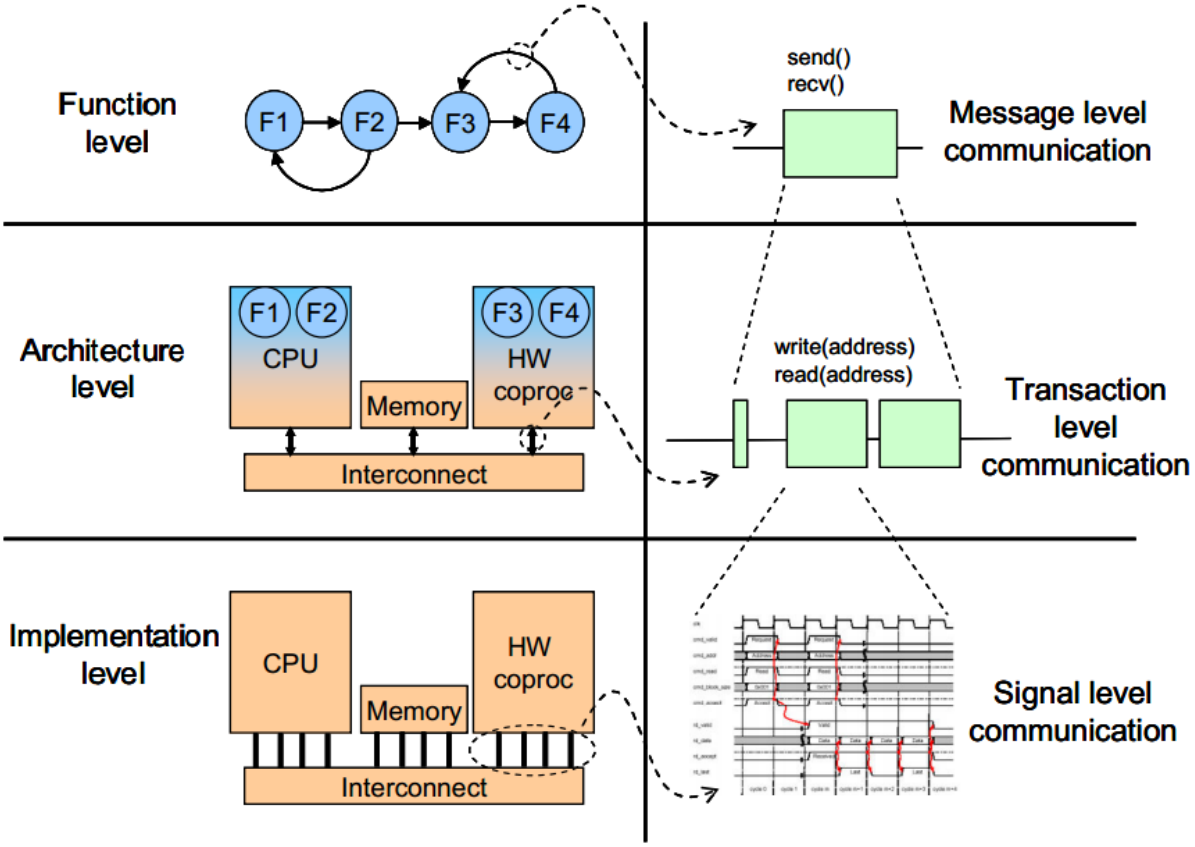


Figura 2.5.1 – Niveles de abstracción y comunicaciones

2.8 Comparativas con otros lenguajes

SystemC no representa un lenguaje en sí mismo, ya que se trata de una serie de librerías dentro de un lenguaje formado como es C++. Sin embargo aporta herramientas externas (del propio equipo de desarrollo de SystemC) que permiten trabajar de forma eficaz.

SystemC agrega al lenguaje C++ las siguientes características:

- Noción de tiempo.
- Concurrencia.
- Tipos de datos especiales para modelado de Hardware.
- Jerarquía modular para manejar la estructura y conectividad.

Además, SystemC nos ofrece ciertas ventajas:

- Abstracción: Ofrece varios niveles de abstracción para el modelado de sistemas.
- Reutilización del diseño: Al igual que C++, el diseño puede ser reutilizado.
- Permite especificar el mismo tiempo el software y el hardware de un sistema electrónico. [7]

Dependiendo del problema a resolver podemos recurrir a lenguajes de programación con distintos grados de complejidad, así adaptarse al diseño especificado, seleccionando algunos de estos lenguajes para partes distintas de un mismo sistema. Por ejemplo C/C++ predomina en el diseño de software embebido, lenguajes de definición de hardware (HDL) como Verilog o VHDL están utilizándose en simulación y sintetizado de elementos hardware, Vera es un lenguaje muy utilizado a la hora de verificar circuitos integrados de tipo ASIC y Matlab y otros lenguajes del estilo son utilizados para capturar los requisitos de los sistemas.

SystemVerilog es un nuevo lenguaje que encierra el lenguaje Verilog para direccionar muchos de los problemas del diseño de sistemas orientados a hardware. Matlab y otras varias herramientas y lenguajes tales como SPW y System Studio son ampliamente usados para requerimiento de sistemas y algoritmos para el procesado de señales. [4]

En la figura 2.6 se puede apreciar cómo SystemC es un lenguaje más completo que cualquiera de los mencionados anteriormente, y que además podría sustituir el uso de varios de ellos a la vez. Ya que llega desde el nivel RTL hasta el de requisitos.

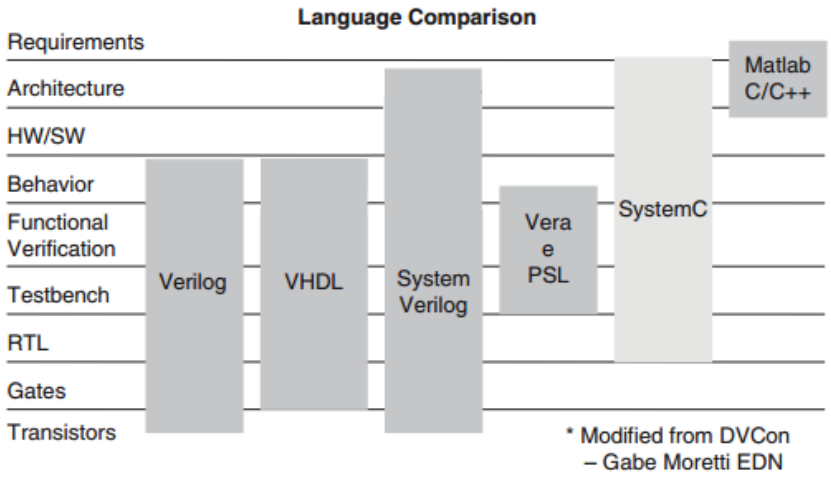


Figura 2.6 Niveles de abstracción manejados por SystemC respecto al de otros lenguajes. [4]

El motor de ejecución de SystemC permite la simulación de procesos concurrentes al igual que Verilog , VHDL o cualquier otro lenguaje de descripción de hardware. Esto es posible gracias al uso de un modelo cooperativo multitarea, el cual corre en un kernel que orquesta el intercambio de procesos y mantiene los resultados y evaluaciones de dichos procesos alineados en tiempo.

2.9 Instalación de la librería SystemC

La instalación de la librería de SystemC no es algo sencillo, por eso se ha incluido este apartado. Aquí se explica cómo se instala la librería de SystemC en el sistema operativo Ubuntu y como se enlaza desde el entorno de desarrollo Eclipse. [8], [9]

Lo primero es instalar la librería en nuestro sistema, para ello:

1. Una vez descomprimido el fichero descargado (en este caso, el systemc-2.3.0.tgz) se abre un terminal y se va al directorio donde se ha descomprimido el fichero (normalmente será el …/systemc-2.3.0)

2. Generar un directorio temporal para realizar la compilación y entrar en dicho directorio:

```
mkdir objdir
cd objdir
```

3. Cambiar la variable de entorno CXX para que encuentre el compilador:

```
setenv CXX g++
```

Es necesario tener instalado todo el entorno g++ en nuestro sistema operativo para poder instalar la librería de SystemC.

4. Ejecutar el fichero "configure". Se recomienda usar la opción "--prefix" para indicar el directorio donde se va a instalar la librería, lo hemos realizado de la siguiente manera:

```
../configure --prefix=/home/fjimenez/SystemC/systemc-2.3.0/lib
```

5. Una vez que ha terminado, se compila la librería:

```
make
```

Al igual que con el entorno g++, es necesario tener instalada la utilidad make en nuestro sistema operativo para poder instalar la librería de SystemC.

6. Una vez que ha terminado la compilación, se instala en el sistema:
   make install

7. Con esto ya estaría instalada, si se desea, se puede comprobar la instalación:
   make check

8. Una vez terminada la instalación, se puede borrar el directorio temporal, no obstante no se recomienda ya que, si se mantiene, se puede reinstalar o limpiar la librería.

A continuación, hay que añadir la librería de *SystemC* que se ha instalado en el sistema al entorno de desarrollo Eclipse, para ello hay que:

1. Dentro de Eclipse, pinchar en Project -> Properties.

2. Desplegar "C/C++ General" y pinchar en "Paths and Symbols".

3. En la pestaña "Includes", seleccionar dentro de los lenguajes el "GNU C++" y pinchar en el botón "Add…" para añadir un nuevo directorio.

4. Habilitar la opción "Add to all configurations" y después pinchar en el botón "File system…". Se abre un explorador de directorios.

5. Dentro del explorador buscar con dicho explorador el directorio donde se encuentran los *includes* de la librería de *SystemC* y pinchar en "Aceptar", se cerrará el explorador y se volverá a la ventana de la figura 2.8.

6. Pinchar en "Ok". Con esto ya se han añadido los includes de la librería.

7. El siguiente paso es incluir el directorio de la librería, para ello hay que ir (desde la pantalla de la figura 2.7) a la pestaña "Library Paths" y pinchar en el botón "Add…" Se abre una ventana similar a la de la figura 2.8.

8. Habilitar la opción "Add to all configurations" y después pinchar en el botón "File system…". Se abre un explorador de directorios.

9. Dentro del explorador buscar con dicho explorador el directorio donde se encuentra la librería de *SystemC* y pinchar en "Aceptar", se cerrará el explorador y se volverá a la ventana de la figura 2.8.

10. Pinchar en "Ok". Con esto ya se han añadido el directorio de la librería.

11. Por último hay que añadir dos librerías, para ello hay que ir (desde la pantalla de la figura 2.7) a la pestaña "Libraries" y pinchar en "Add…" Se verá una pantalla similar a la de la figura 2.8.
12. En el cuadro de texto, se escribe "systemc", se habilita la opción "Add to all configurations" y se pincha en el botón "Ok".

13. Se repite la operación anterior para añadir la librería "m".

14. Una vez realizados todos los pasos, en la venta mostrada en la figura 2.7 se pincha en el botón de "Apply" para aplicar los cambios y en el de "Ok" para salir.

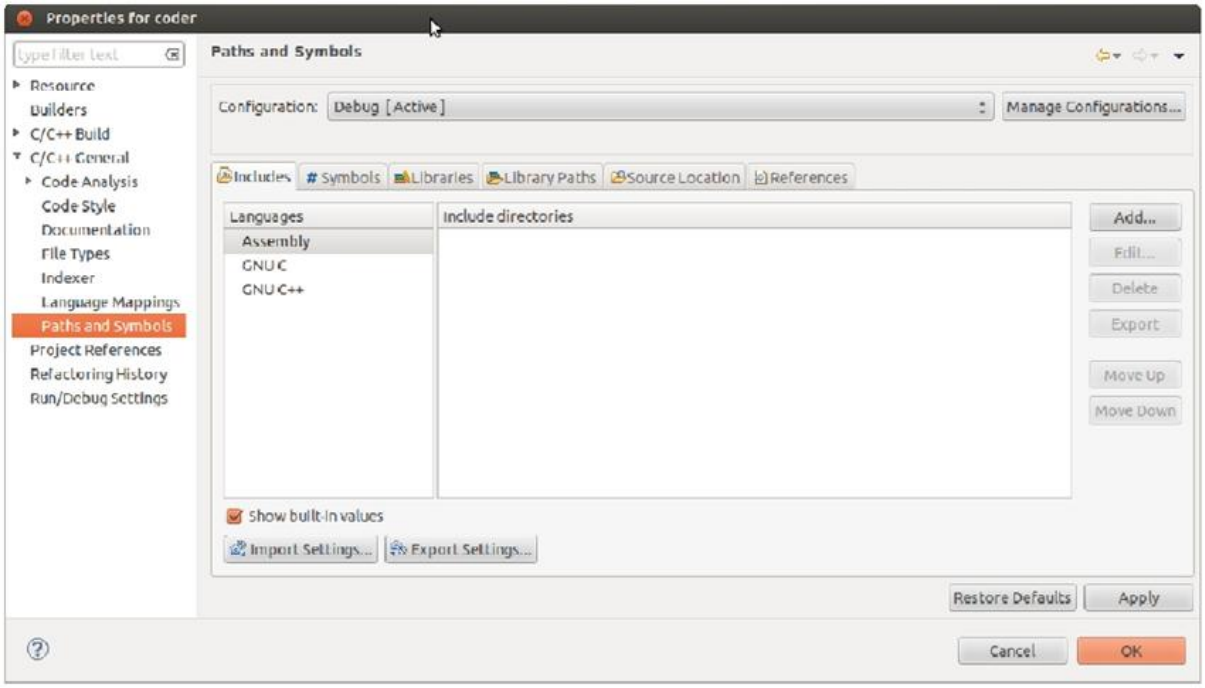


Figura 2.7 Paths y símbolos del proyecto

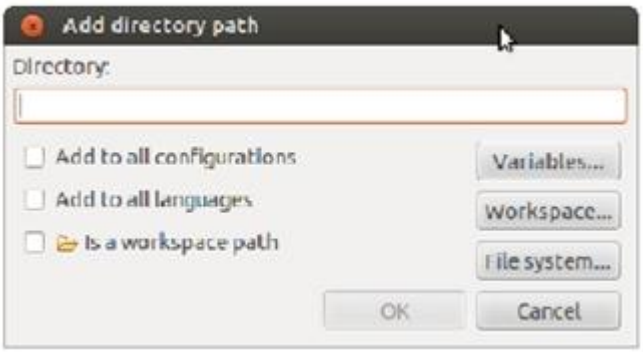


Figura 2.8 Añadir un directorio

Por último, es necesario añadir unas variables de entorno para evitar unos problemas a la hora de ejecutar el código desde Eclipse:

1. Dentro de eclipse pinchar en Run -> Run Configurations… .Figura 2.9
2. Pinchar en la pestaña de "Enviroment" y en el botón de "New…"
3. En el cuadro "Name:" hay que poner el nombre de la variable,"LD_LIBRARY_PATH", y en "Value:" hay que poner el directorio donde se encuentra la librería de *SystemC*, en nuestro caso "/home/fjimenez/SystemC/systemc-2.3.0/lib/lib-linux64".
4. Pinchar en el botón de "Ok" para añadirla.
5. Se vuelve a pinchar en el botón de "New…" para añadir otra variable de entorno más.
6. En "Name:" hay que poner "SC_SIGNAL_WRITE_CHECK" y en "Value:" se pone "DISABLE". Se pincha en "Ok" para añadirla. Se puede ver una pantalla similar a la figura 2.9
7. Se pincha en el botón de "Apply" para guardar los cambios y en "Ok" para salir. Ya están las dos variables de entorno configuradas para evitar los errores en la simulación.

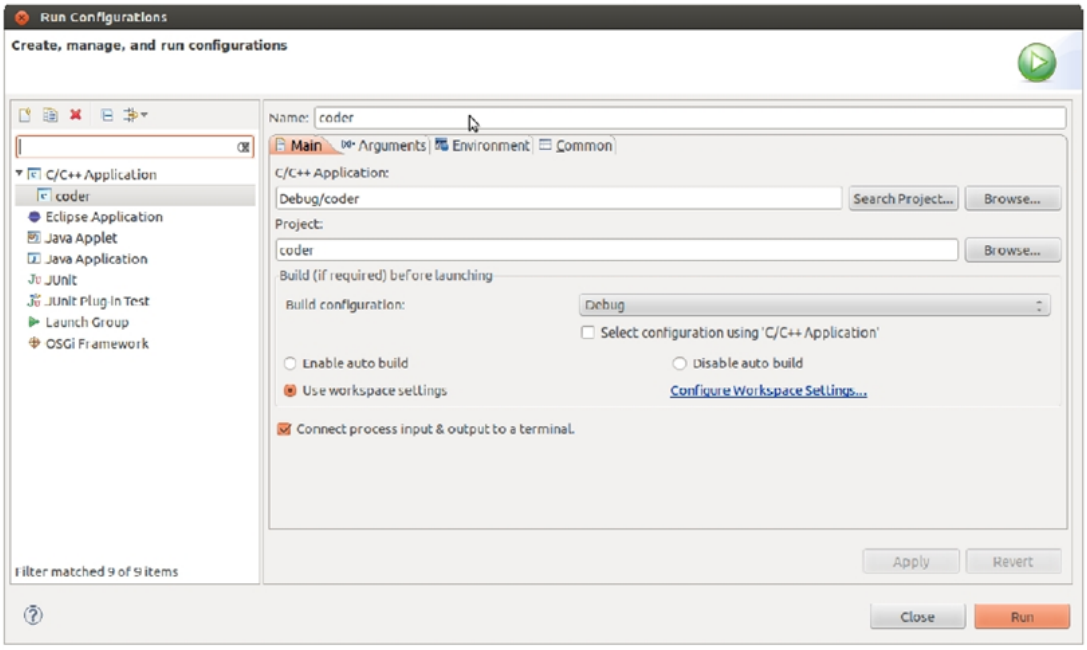


Figura 2.9 Configuración de ejecución

2.10 Instalación y prueba del Simulador de SystemC

2.10.1 Compilando e instalando GTKWave

GTKWave es un visor de simulación de ondas, que utilizaremos para ver la simulación de nuestros programas, a través de los ficheros de traza (apartado 2.5.3) que generamos en nuestro main del programa, la extensión de estos ficheros es .vcd, Si estamos utilizando una versión de Ubuntu 10.04 o superior podemos instalarlo accediendo a Aplicaciones->Ubuntu Software Center, escribir en la barra de búsqueda gtkwave, hacer click en instalar, luego ingresamos nuestra clave de administrador y listo. Figura 2.10.

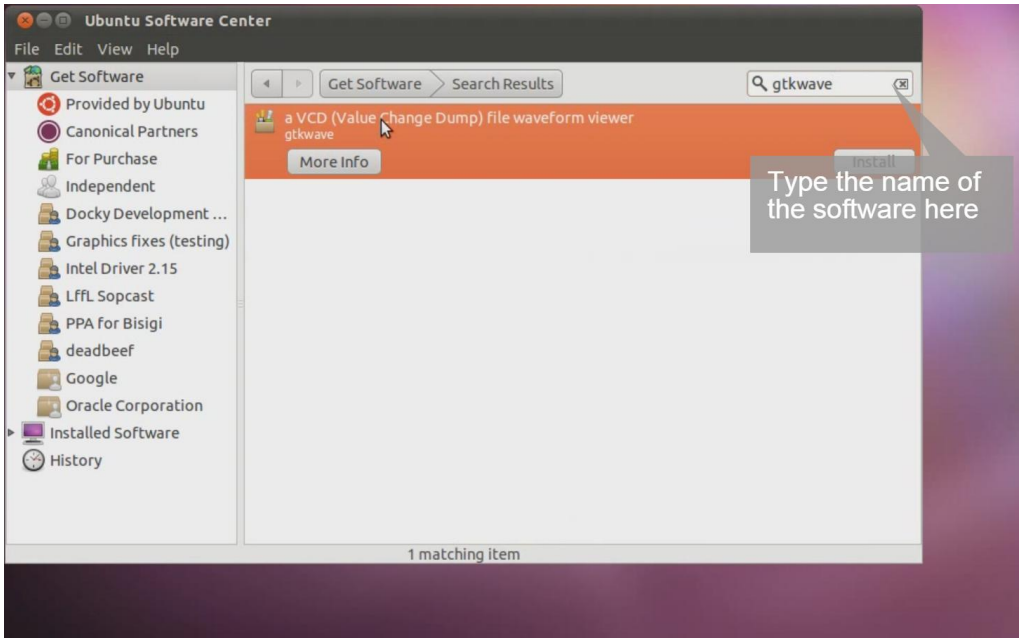


Figura 2.10 Instalación de gtkwave en Ubuntu 10.04

Si no es nuestro caso:

1. A través de un terminal escribimos el siguiente comando:

   sudo apt-get install gtkwave

   Nos descargará todos los paquetes necesarios desde el repositorio, aceptamos con y. Se instalará muy rápidamente.

2. Comprobar su funcionamiento, es sencillo, escribimos gtkwave en un terminal y se nos abrirá el programa. Figura 2.11

3. Tenemos que localizar el fichero de traza .vcd que contiene un seguimiento de onda en respuesta a la simulación. Para poder verlo y verificar los datos ejecuta:

   gtkwave <nombre_del_fichero>.vcd

4. Usa las opciones de menú en Search->Signal Search…para añadir señales y poder verificar las salidas del modelo. Si estás usando la versión 2.0.0pre3 de gtkwave y dispones de un fichero con trazas salvadas de una anterior ejecución (por ejemplo adder.trc) puedes ejecutar la simulación cargando automáticamente las trazas con la llamada:

   gtkwave adder.vcd adder.trc

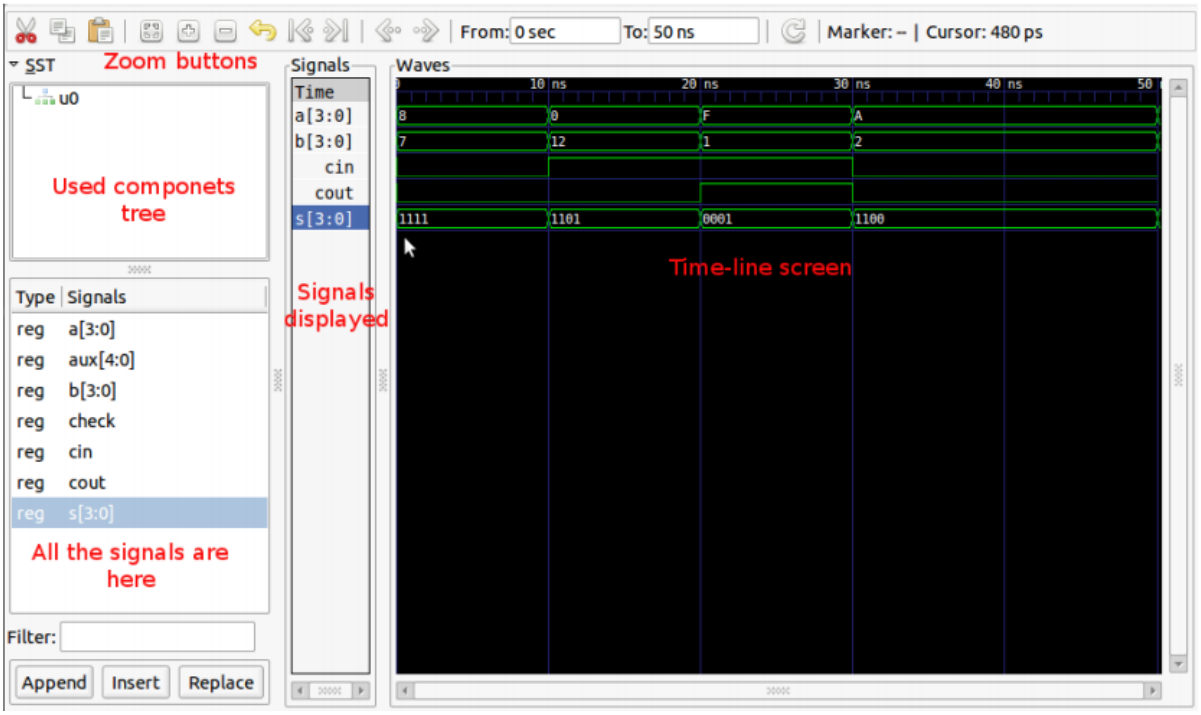


Figura 2.11 Simulador GTKWave

Capítulo 3: Aplicación a una estructura

3.1 Arquitectura del procesador

Este procesador se basa en una arquitectura de memoria tipo Harvard (utilizamos dispositivos de almacenamiento físicamente separados para las instrucciones y para los datos) y en el uso de la segmentación. [10]

El pipeline del procesador estará dividido en cinco etapas. Si cada etapa dura un ciclo, las instrucciones duran cinco ciclos en ejecutarse.

Serán las 5 etapas que normalmente implementa un procesador DLX y las denominaremos a partir de ahora:

- BUSQ (fetch instruction)
- D/LE (Instruction Decodification)
- ALU (Ejecución)
- MEM (Memoria)
- ESCR (Almacenamiento en el registro).

3.1.1 Formatos de instrucción

Se van a codificar tres tipos distintos de formato de instrucción. Estos formatos nos van a indicar que valores hay que poner en la palabra de instrucción y en que bits. Los formatos son R, I y J.

Formato de instrucción R se codifican todas aquellas operaciones en las que hay dos registros fuentes y un registro destino. Se usan para las operaciones de la ALU entre registros. Figura 3.1

.

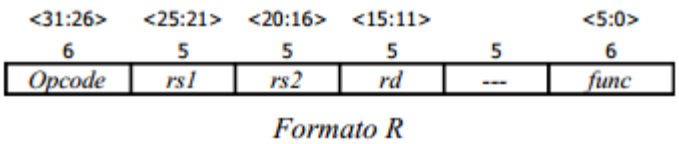


Figura 3.1 - Formato de instrucción R

El formato I se usa para todas aquellas instrucciones donde hay un valor constante implícito. Este valor constante está codificado en complemento a 2 en el campo Inmediate de 16 bits. Figura 3.2

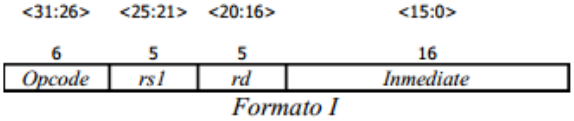


Figura 3.2 Formato de instrucción I

El formato de instrucción J se usa para las instrucciones de salto lejano. Figura 3.3

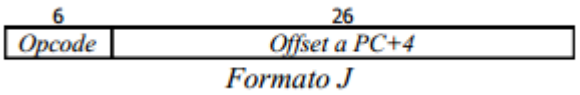


Figura 3.3 Formato de instrucción J

3.1.2 Instrucciones implementadas

En la siguientes tablas aparecen las instrucciones del DLX que se van a implementar en este procesador.

3.1.2.1 Operaciones Aritméticas

Instrucción	Tipo	Cods.Op.	Explicación breve
ADD	R	0 - 0x00	Suma de dos registros fuentes
SUB	R	0 - 0x22	Resta de dos registros fuente
ADDI	I	0x08	Suma de un registro fuente y un inmediato de 16 bits (con el signo extendido)(rd= rs1+ext(inmediato))
ADDUI	I	0x03	Suma de un registro fuente y un inmediato de 16 bits (sin el signo extendido)(rd= rs1+ext(inmediato))
SUBI	I	0x0A	Resta de un registro fuente y un inmediato con signo de 16 bits
SUBUI	I	0x0B	Resta de un registro fuente y un inmediato sin signo de 16 bits

Tabla 3.4 – Conjunto de operaciones aritméticas

3.1.2.2 Operaciones Lógicas

Instrucción	Tipo	Cods.Op.	Explicación breve
SLL	R	0 - 0x04	Desplazamiento lógico de un bit a la izquierda
SRL	R	0 - 0x06	Desplazamiento lógico de un bit a la derecha
AND	R	0 - 0x24	Y lógica de dos registros(rd= ds1 & rs2)
OR	R	0 - 0x25	O lógica de dos registros(rd= rs1 | rs2)
XOR	R	0 - 0x26	O-exclusiva de dos registros (rd = rs1 $\oplus$ rs2)
ANDI	I	0x0C	Y lógica de un registro y un inmediato (rd = rs1 & inmediato)
ORI	I	0x0D	O lógica de un registro y un inmediato (rd = rs1 | inmediato)
XORI	I	0x0E	O-exclusiva de un registro y un inmediato (rd = rs1 $\oplus$inmediato)
LHI	I	0x0F	Carga de una constante en la parte alta (rs = inmediato)

Tabla 3.4 – Conjunto de operaciones lógicas

3.1.2.3 Instrucciones de comparación

Instrucción	Tipo	Cods.Op.	Explicación breve
SEQ	R	0 - 0x28	devuelve 1 si de $R_a = R_b$ En otro caso devuelve 0
SNE	R	0 - 0x29	devuelve 1 si de $R_a \neq R_b$ En otro caso devuelve 0
SLT	R	0 - 0x2A	devuelve 1 si de $R_a < R_b$ En otro caso devuelve 0
SGT	R	0 - 0x2B	devuelve 1 si de $R_a > R_b$ En otro caso devuelve 0
SLE	R	0 - 0x2C	devuelve 1 si de $R_a \leq R_b$ En otro caso devuelve 0
SGE	R	0 - 0X2D	devuelve 1 si de $R_a \geq R_b$ En otro caso devuelve 0
SEQI	I	0x18	devuelve 1 si de $R_a = R_b$ En otro caso devuelve 0
SNEI	I	0x19	devuelve 1 si de $R_a \neq In$ En otro caso devuelve 0
SLTI	I	0x1A	devuelve 1 si de $R_a < In$ En otro caso devuelve 0

SGTI	I	0x1B	devuelve 1 si de $R_a > In$ En otro caso devuelve 0
SLEI	I	0x1C	devuelve 1 si de $R_a \leq In$ En otro caso devuelve 0
SGEI	I	0x1D	devuelve 1 si de $R_a \geq In$ En otro caso devuelve 0

Tabla 3.5 – Conjunto de operaciones de comparación

3.1.2.4 Instrucciones de carga/almacenamiento y especiales

Instrucción	Tipo	Cods.Op.	Explicación breve
LW	I	0x23	Lee una palabra desde memoria (rd=MEM(rs1+extend(inmediato))
SW	I	0x2B	Escribe una palabra en memoria (MEM(rs1+extend(inmediato) =rd)
NOP	I	0x15	No Operación - Añade un ciclo de retraso

Tabla 3.6 – Conjunto de instrucciones de carga/almacenamiento y especiales

3.1.2.5 Instrucciones de salto

Instrucción	Tipo	Cods.Op.	Explicación breve
J	J	0x02	Salto incondicional a un valor inmediato de 26 bits (relativo al PC+4)
JAL	J	0x03	Salto incondicional a un valor inmediato de 26 bits (relativo al PC+4) guardando el calor de PC en R31 (R31 = PC+4; PC = extend(valor))
BEQZ	I	0x04	Bifurcación si un registro es igual a cero (lee z) Desplazamiento de 16bits (relativo a PC+4)
BNEZ	I	0x05	Bifurcación si un registro es distinto a cero (lee z) Desplazamiento de 16bits (relativo a PC+4)
JR	I	0x12	Salto incondicional al contenido de un registro (lee registro B) PC = rs1
JALR	I	0x13	Salto Incondicional al contenido de un registro guardando el valor del pc EN R31 (R31 = PC+4; PC = rs1)

Tabla 3.7 – Conjunto de instrucciones de salto

Las instrucciones JR y JALR solo usan “rs1” y el valor del campo inmediato y “rd” son descartados.

3.1.3 División del diseño (Nivel Funcional)

A continuación se van a presentar las partes en las que quedará dividido el diseño del procesador. Se seguirá una técnica de diseño top-down, partiendo de una visión global hasta llegar a la definición de los distintos bloques funcionales, sus entradas y sus salidas. Estos se revisarán para ver si se adaptan al diseño global y para analizar posibles mejoras tanto de la unidad en cuestión como de todo el diseño completo.

3.1.3.1 Visión global del diseño

Vamos a mostrar una visión global del diseño, al conectar las cinco etapas que implementa un procesador DLX y que describiremos en detalles en los siguientes apartados.

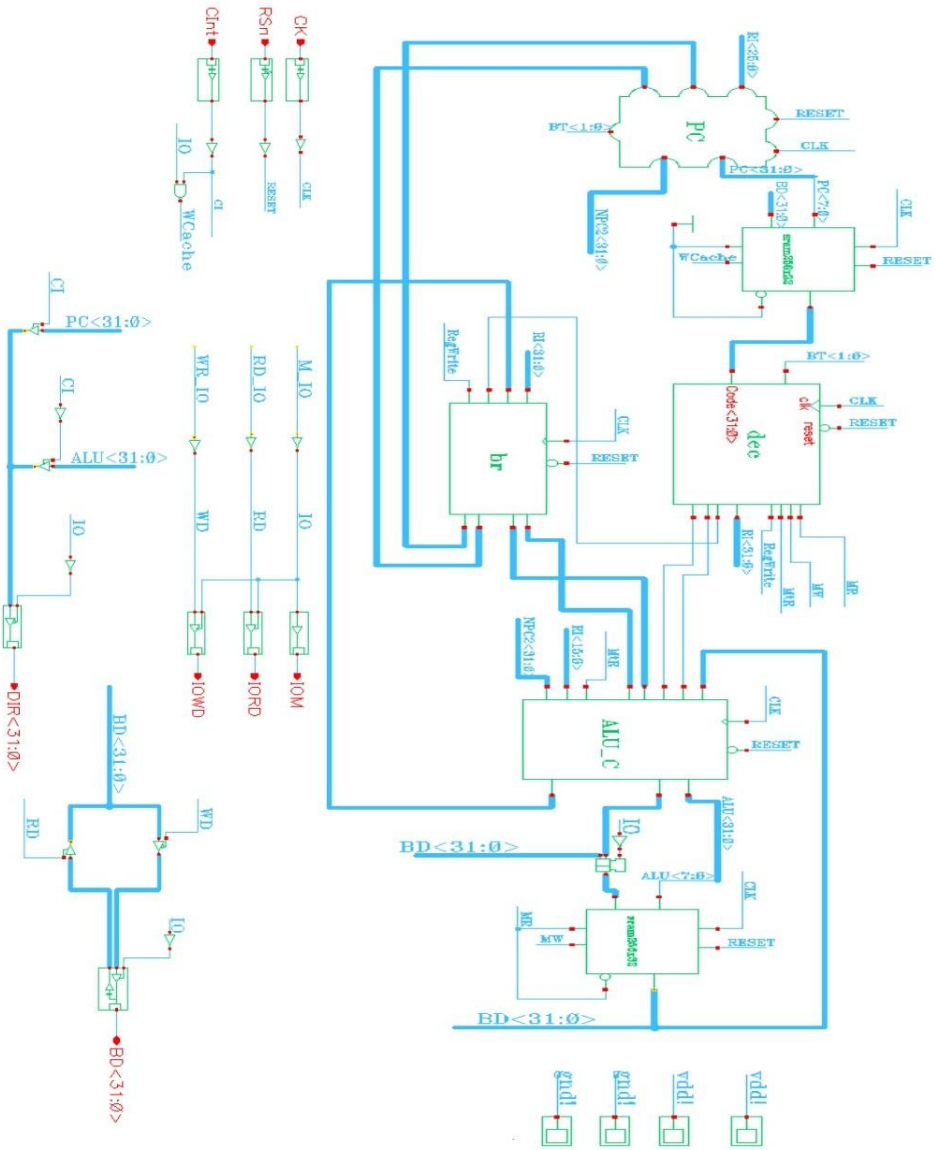


Figura 3.19 – Visión global de la circuitería del diseño

Si nos abstraemos de la circuitería, podremos tener una visión generalizada de las etapas que estarán divididas en módulos independientes de entrada y salida. Lo mostramos en la siguiente figura.

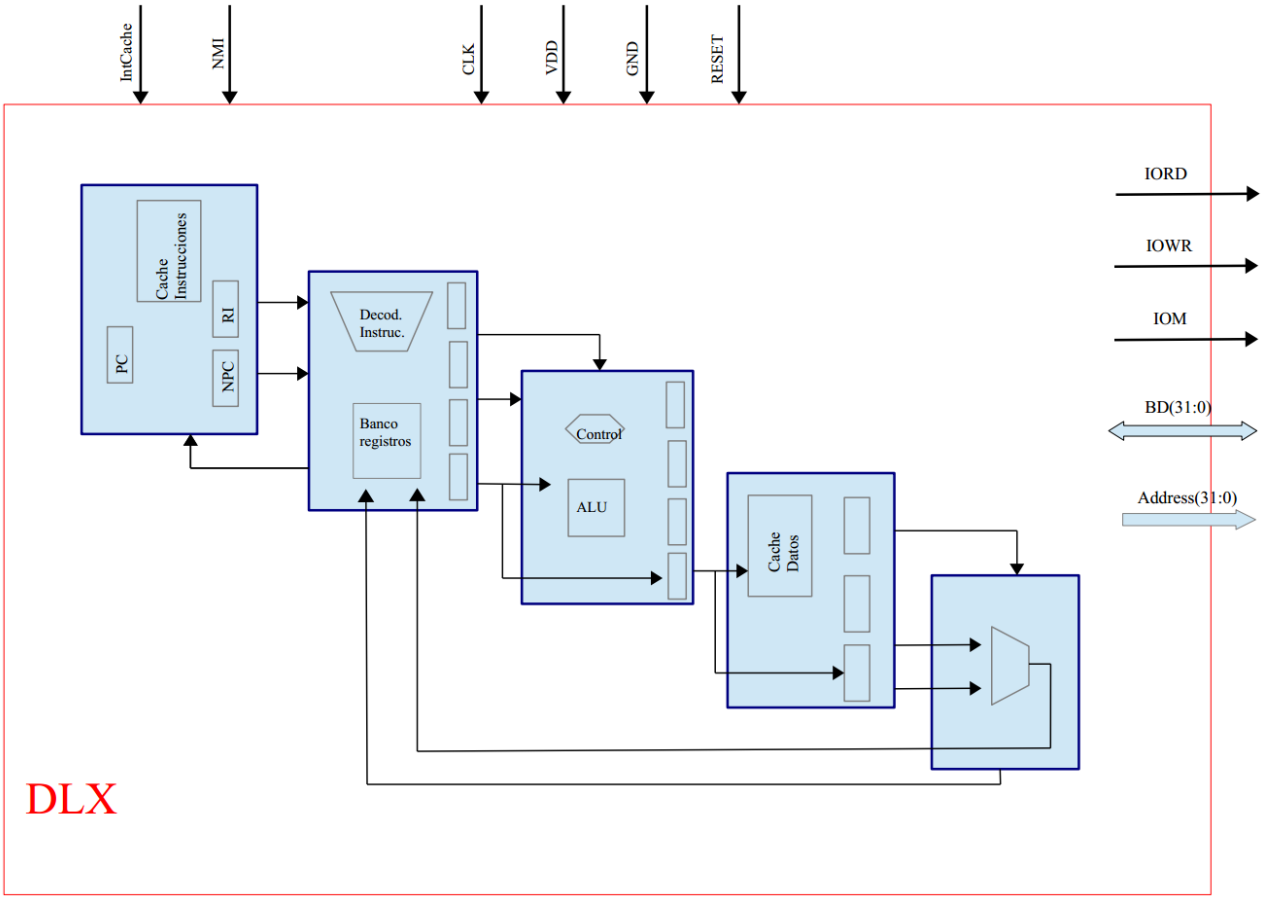


Figura 3.20 – Visión global del diseño por módulos

Una visión un poco más abstracta, a modo de resumen, de nuestro procesador quedaría de la siguiente manera:

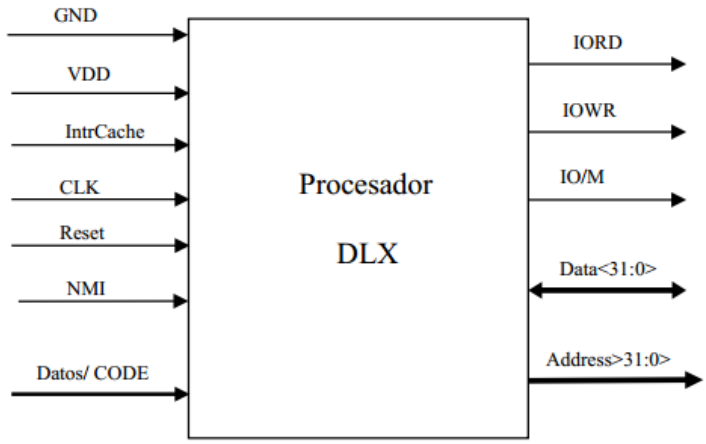


Figura 3.21 – Visión global del diseño

La señal de reloj, se dispara en el flanco de subida, el banco de registros escribe en flanco de bajada y lee en flanco de subida, la señal de reset se activa en alta.

Las señales IntrCache y NMI gestionan las interrupciones y enmascaramiento respectivamente. Para gestionar las interrupciones, deberemos conectar en su caso el chip de interrupciones, eje. 8259, a estas patillas.

Las señales IORD y IOWR indican a la memoria externa/dispositivo si debemos escribir o leer.

La señal IO/M, línea de estados de memoria o E/S: Indica si la dirección es de memoria o de I/O.

El bus de datos Data<31:0> de entrada/salida, posibilita recibir datos (Datos/CODE) de la memoria externa o dispositivos o volcar datos hacia fuera con los distintos dispositivos mediante un puerto E/S.

Un bus de direcciones Address<31:0> que indica la posición para leer o escribir.

3.1.3.2 Etapa de Búsqueda (BUSQ)

Se encarga de la búsqueda de la instrucción en la cache de instrucciones. Además, esta parte se encarga de toda la lógica necesaria para generar el próximo PC (incrementos, saltos condicionales, saltos incondicionales, etc.). La decisión de cuál será el próximo valor del PC se hará en la etapa de D/LE y afectará a la siguiente instrucción capturada. Se ha forzado el diseño para tomar la decisión en la etapa de decodificación /lectura. De esta manera se minimiza el número de huecos que se producirán cuando se ejecute una instrucción de salto, en el detrimento de un aumento considerable de la circuitería necesaria en esta etapa. [10]

El módulo de PC se encargará de seleccionar la instrucción que se va a ejecutar en siguiente lugar, encargándose de los saltos sin ayuda del módulo ALU, y enviándolo a este módulo, cuando su valor cuando sea necesario.

El módulo salto se encargará de generar la señal que nos dicta si debemos de saltar o no, usando como entrada BT (BranchType<1:0>) y Z usando un multiplexor obtendremos como salida el mismo BranchType en la mayoría de los casos, aunque en uno de ellos estemos condicionados a el valor de Z.

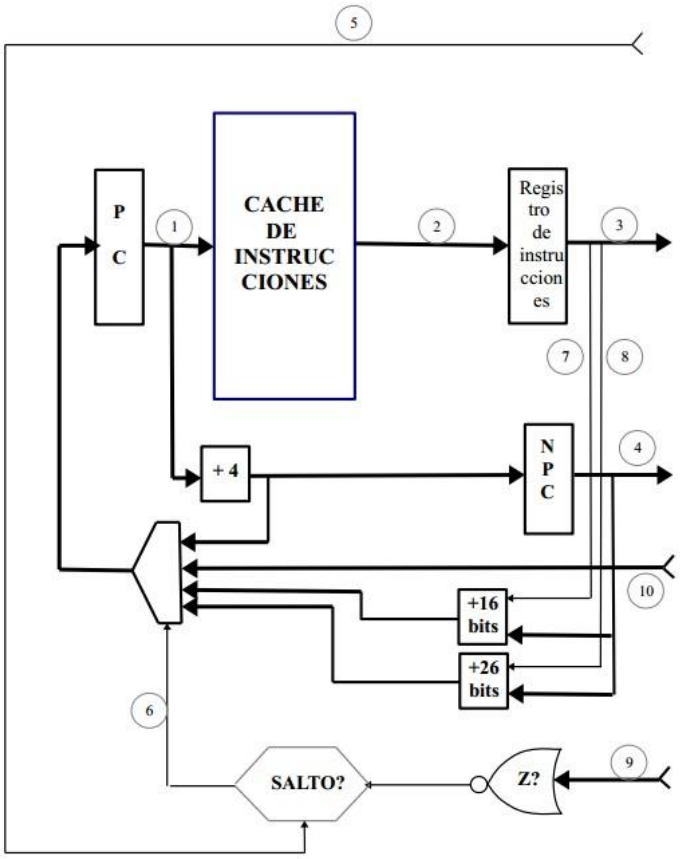


Figura 3.8 – Diseño da la etapa de Búsqueda

Señal	Función	Nº	Tipo (Entrada /Salida/Señal Interna)
clk	Señal de reloj, disparo en el flanco de subida		Entrada
reset	Señal de reset		Entrada
PC<31:0>	Valor del PC para la etapa de búsqueda	1	Señal Interna
Code<31:0>	Salida de la cache de instrucciones	2	Señal Interna
RI<31:0>	Salida del registro de instrucciones	3	Salida
NPC<31:0>	Valor del PC+4 para etapa DEC/LEC	4	Salida
BranchType<1:0>	Tipo de salto que se va a realizar	5	Entrada
BranchSource<1:0>	Determina la carga al registro PC	6	Señal Interna
RI<15:0>	16 bits de RI (hay que extender el signo)	7	Señal Interna
RI<25:0>	26 bits de RI (hay que extender el signo)	8	Señal Interna
BrA<31:0>	Valor del registro A	9	Entrada
BrB<31:0>	Valor del registro B	10	Entrada

Tabla 3.9 – Tabla de Señales de la etapa de búsqueda

3.1.3.3 Etapa de Decodificación/Lectura (DE/L)

El decodificador es la unidad funcional que nos permite reconocer la instrucción que entra en nuestro procesador. Además, se encarga de gestionar todos los puntos de control necesarios que tenemos que activar para la instrucción que estemos procesando en ese momento.

Además, consta de un banco de registros que está compuesto por 32 registros de 32 bits con dos puertos de lectura y un puerto de escritura. El registro cero siempre nos va a devolver un valor cableado igual a cero. Para permitir que se pueda leer el resultado y poder escribir en el mismo ciclo tendremos que adelantar la escritura para que se escriba a mitad de ciclo. Es decir, en el flanco de bajada del ciclo de escritura se debe guardar el resultado en el registro destino. En ese mismo semiciclo se deben propagar los buses internos del banco de registros para permitir la lectura de los valores correctos al final del ciclo en curso. Además, en ese mismo semiciclo se llevará a cabo la comparación para el salto.

También, nos encontramos con el módulo de los extensores de signo con una señal de entrada RI<25:0> que es el valor que tendremos que extender de signo. La salida resultante de este módulo es INM<31:0> que es una de las salidas, la cual representa el valor inmediato de 32 bits.

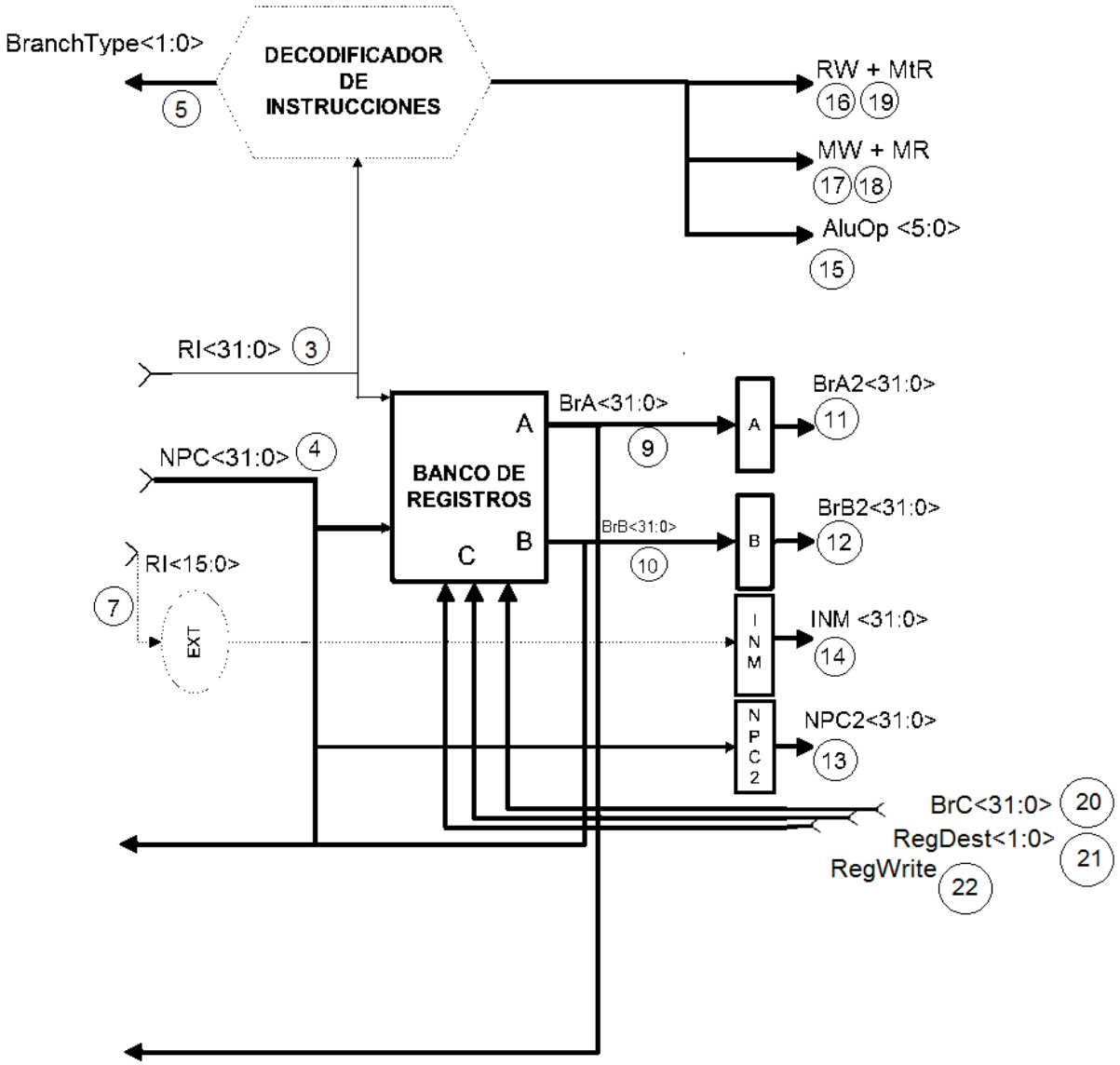


Figura 3.10 – Diseño da la etapa de Decodificación y lectura

Señal	Función	Nº	Tipo (Entrada /Salida/Señal Interna)
clk	Señal de reloj, disparo en el flanco de subida		Entrada
reset	Señal de reset		Entrada
RI<31:0>	Salida del registro de instrucciones	3	Entrada
NPC<31:0>	Valor del PC+4 para etapa DEC/LEC	4	Entrada
BranchType<1:0>	Tipo de salto que se va a realizar	5	Salida
RI<15:0>	16 bits de RI (hay que extender el signo)	7	Señal Interna
BrA<31:0>	Valor del registro A	9	Salida
BrB<31:0>	Valor del registro B	10	Salida
BrA2<31:0>	Valor del registro A para la etapa de la ALU	11	Salida
BrB2<31:0>	Valor del registro B para la etapa de la ALU	12	Salida
NPC2<31:0>	Valor del PC+4 para etapa ALU	13	Salida
INM	Valor inmediato para etapa ALU	14	Salida
AluOp<5:0>	Indica que operación debe hacer la ALU	15	Salida
MemToReg	Selector de la cache de datos (MtR)	16	Salida
MemWrite	Escritura en la caché de datos (MW)	17	Salida
MemRead	Lectura de la caché de datos (MR)	18	Salida
RegWrite	Se deberá escribir en el registro destino (RW)	19	Salida
BrC<31:0>	Valor a escribir en el registro C	20	Entrada
RegDest<1:0>	Fuente para el registro destino (dependiendo del tipo de instrucción)	21	Entrada
RegWrite3	Se deberá escribir en el registro destino después de la etapa ESC (RW3)	22	Entrada

Tabla 3.11 – Tabla de Señales de la etapa de decodificación y lectura

3.1.3.4 Etapa de la ALU (Ejecución)

Es la ALU de 32 bits del microprocesador, hará falta una pequeña unidad de control para generar algunas señales internas, principalmente las necesarios para seleccionar el tipo de operación y la salida en función del código de operación de entrada.

La ALU debe soportar la mayoría de las operaciones que se va a realizar en nuestro procesador. En la siguiente tabla se definen las funciones que debe implementar y el código de función que tiene asociada esa operación:

Función	**Código**
Suma del operando A y B	000
Resta del operando A y B	001
O-lógica del operando A con el B	010
Y-lógica del operando A con el B	011
O exclusiva del operando A con el B	100
Desplazamiento lógico un bit a la izquierda	101
Desplazamiento lógico de un bit a la derecha	110
Carga de cte. en la parte superior (LHI)	111

Tabla 3.12 – Tabla del código de función de la ALU

Debemos implementar la lógica de control para seleccionar la operación de la ALU, multiplexores, registros del pipeline y la extensión del signo del valor inmediato.

También se encargará de las operaciones de comparación, las cuales devuelven 0 o 1 dependiendo de los resultados que de la ALU y de la selección del valor a escribir en el banco de registros.

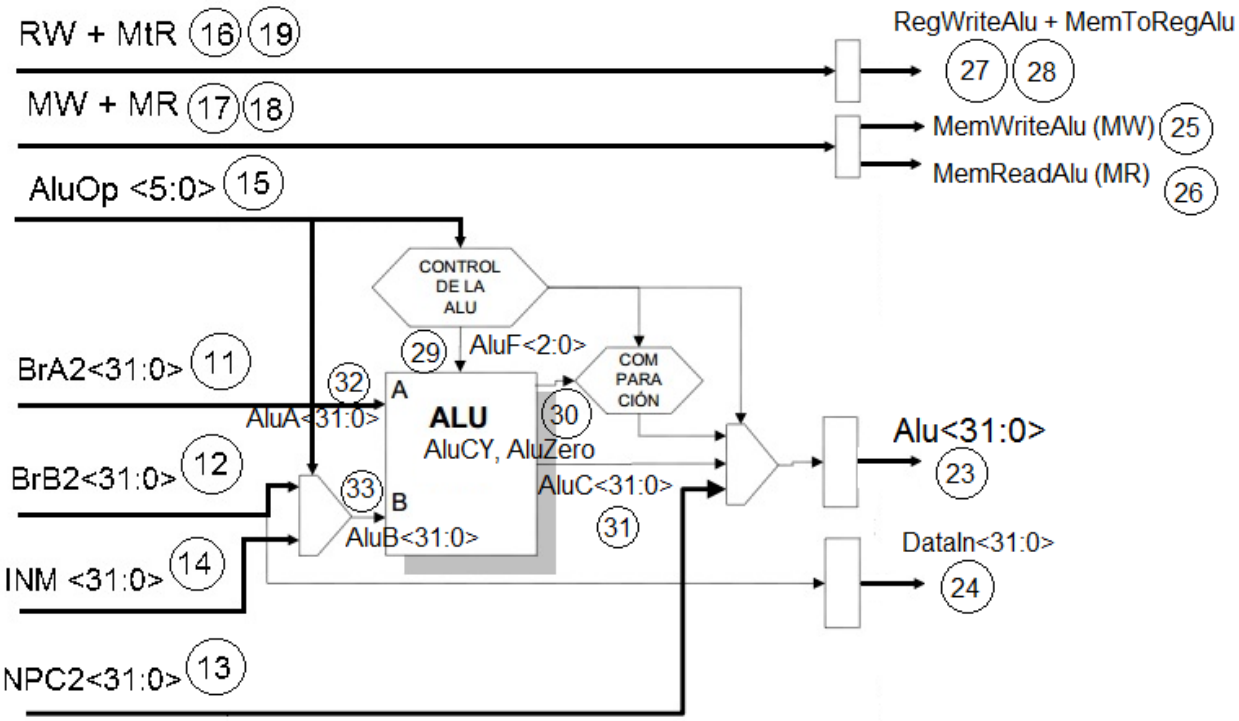


Figura 3.13 – Diseño da la etapa de Decodificación y lectura

Señal	Función	Nº	Tipo(Entrada/Salida/Señal Interna)
clk	Señal de reloj, disparo en el flanco de subida		Entrada
reset	Señal de reset		Entrada
BrA2<31:0>	Valor del registro A para la etapa de la ALU	11	Entrada
BrB2<31:0>	Valor del registro B para la etapa de la ALU	12	Entrada
INM	Valor inmediato para etapa ALU	14	Entrada
NPC2<31:0>	Valor del PC+4 para etapa ALU	13	Entrada
AluOp<5:0>	Indica que operación debe hacer la ALU	15	Entrada
MemToReg	Selector de la cache de datos (MtR)	16	Entrada
MemWrite	Escritura en la caché de datos (MW)	17	Entrada
MemRead	Lectura de la caché de datos (MR)	18	Entrada
RegWrite	Se deberá escribir en el registro destino (RW)	19	Entrada
Alu<31:0>	Salida de la etapa de la ALU	23	Salida
DataIn<31:0>	Entrada en la cache de datos 2	24	Salida
MemWriteAlu	Escritura en la caché de datos (MW) etapa de la ALU	25	Salida
MemReadAlu	Lectura de la caché de datos (MR) etapa de la ALU	26	Salida
MemToRegAlu	Selector de la cache de datos (MtR) etapa de la ALU	27	Salida
RegWriteAlu	Se deberá escribir en el registro destino (RW) etapa de la ALU	28	Salida
AluF<2:0>	Función de la ALU	29	Señal Interna
AluCY, AluZero	Valores de estado de la ALU	30	Señal Interna
AluC<31:0>	Resultado de la ALU	31	Señal Interna
AluA<31:0>	Operando A de la ALU	32	Señal Interna

Tabla 3.14 – Tabla de Señales de la etapa de la ALU

3.1.3.5 Etapa de Memoria (Mem)

La cuarta etapa consiste en un acceso a la unidad de memoria. Dicho acceso estará gobernado (habilitado) por el bloque D/LE, y en caso de habilitación, la dirección de memoria viene dado por el resultado de la ALU, y el dato viene dado por el contenido de uno de los puertos (B) del banco de registros. Al igual que sucede con los bancos de memoria, la escritura en memoria se lleva a cabo en el flanco de bajada para no aumentar el número de ciclos.

Cuando se realice una operación de escritura en memoria, la memoria debe estar habilitada. En dicho caso, la salida de la ALU sería la dirección de escritura, y el dato del banco de registros B sería el dato a escribir.

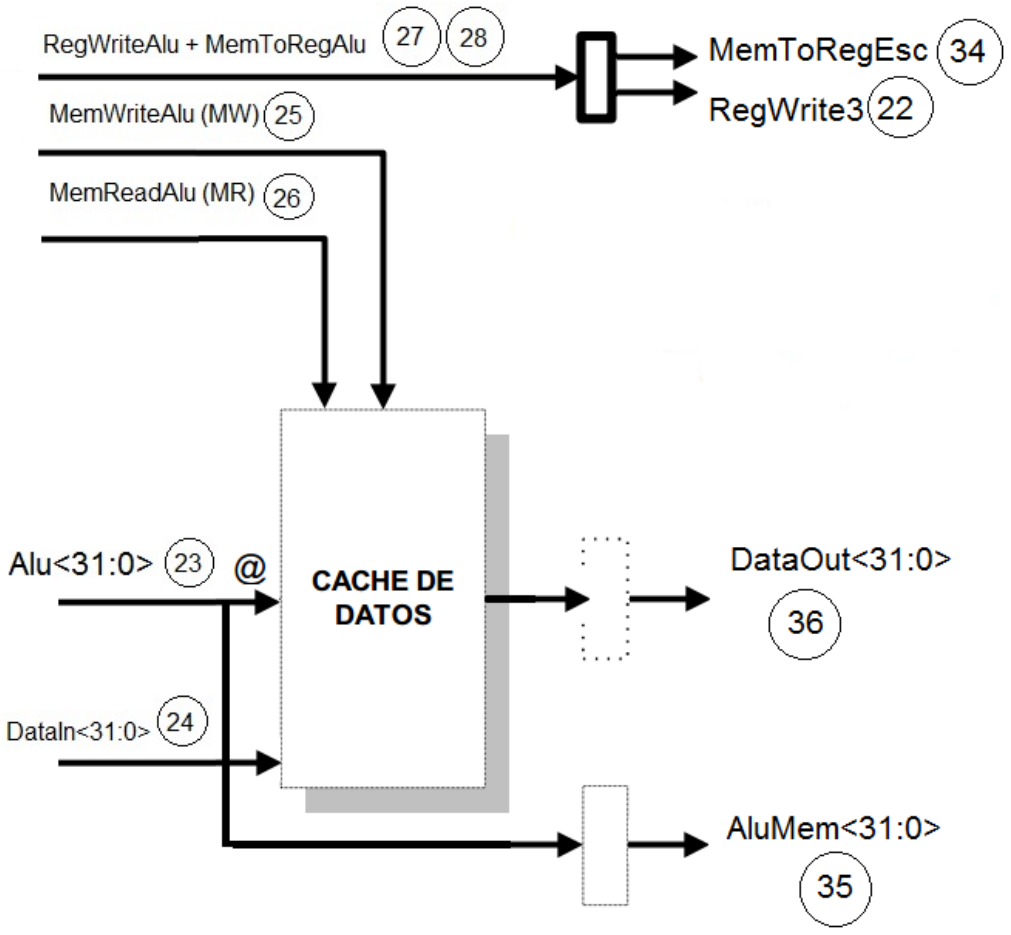


Figura 3.15 – Diseño da la etapa de Memoria

Señal	Función	Nº	Tipo(Entrada/Salida/Señal Interna)
clk	Señal de reloj, disparo en el flanco de subida		Entrada
reset	Señal de reset		Entrada
Alu<31:0>	Salida de la etapa de la ALU	23	Entrada
DataIn<31:0>	Entrada en la cache de datos 2	24	Entrada
MemWriteAlu	Escritura en la caché de datos (MW) etapa de la ALU	25	Entrada
MemReadAlu	Lectura de la caché de datos (MR) etapa de la ALU	26	Entrada
MemToRegAlu	Selector de la cache de datos (MtR) etapa de la ALU	27	Entrada
RegWriteAlu	Se deberá escribir en el registro destino (RW) etapa de la ALU	28	Entrada
RegWrite3	Se deberá escribir en el registro destino después de la etapa ESC (RW3)	22	Salida
MemToRegEsc	Selector de la cache de datos (MtR)	34	Salida
AluMem<31:0>	Salida de la etapa de la ALU en la etapa de MEM	35	Salida

Tabla 3.16 – Tabla de Señales de la etapa de Memoria

3.1.3.6 Etapa de Escritura (Esc)

Por último, la quinta etapa consiste en el almacenamiento en el banco de registros del dato obtenido por la ALU (en el caso de una operación) o por la unidad de memoria (en el caso de un almacenamiento en registro). [10]

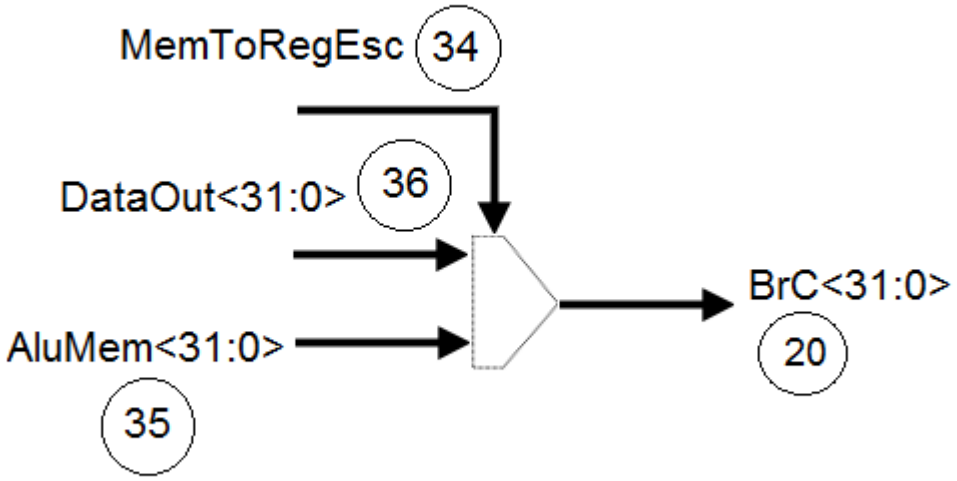


Figura 3.17 – Diseño da la etapa de Escritura

Señal	**Función**	**Nº**	**Tipo (Entrada/Salida/Señal Interna)**
clk	Señal de reloj, disparo en el flanco de subida		Entrada
reset	Señal de reset		Entrada
MemToRegEsc	Selector de la cache de datos (MtR)	34	Entrada
AluMem<31:0>	Salida de la etapa de la ALU en la etapa de MEM	35	Entrada
DataOut<31:0>	Salida de la cache de datos	36	Entrada
BrC<31:0>	Valor a escribir en el registro C	20	Salida

Tabla 3.18 – Tabla de Señales de la etapa de Escritura

Capítulo 4: Modelado del Diseño

Vamos a proceder a comentar los módulos más importantes que hemos implementado en el diseño de nuestro procesador DLX con SystemC. En el apartado 4.3.3 tenemos una visión global del funcionamiento del procesador a partir del código que hemos creado.

4.1 Módulo DLX

Ficheros DLX.cc y DLX.h: consta de cuatro tablas, **DLX::opTable[ ]** y **DLX::specialTable[ ]** se encargan de traducir los bits 31:26 de la instrucción en un valor adecuado para el campo "código de operación", o en un valor especial para su posterior decodificación, **FParithTable[ ]** se utiliza para convertir el campo "func" de las instrucciones y **DLX::operationNames[ ]** es de tipo char e indica el nombre de la operación. Además de cuatro métodos, **void DLX::unimplemented(int opcode)** muestra que la operación no está implementada**, void DLX::decode(...)** este método decodifica la instrucción con los parámetros que se le pasan, es el formato indicado en el capítulo anterior. **Void DLX::execute(…)** básicamente con un switch-case comprueba el nombre de la operación y realiza lo oportuno almacenándolo en las variables alu_output y r32_output, cabe la posibilidad de una futura mejora, implementando mas operaciones, por ello se incluyen todas las operaciones del procesador dlx y las que no hemos implementado, llamará al método unimplemented. Por último, **DLX::print(…)** muestra el nombre de la operación si es conocida. El recurso [11] nos muestra el conjunto completo de instrucciones para el procesador dlx, además de un ejemplo de cada operación, me ha resultado muy útil.

```
static void decode(uint32_t insn,
           int& format,
           OpcodeEnum& opcode,
           int& rs1,
           int& rs2,
           int& rd,
           int& imm);
```

```
static void execute(OpcodeEnum opcode,
            int A,
            int B,
            int imm,
            int PC,
            int NPC,
            int& alu_output,
            int& r31_output);
```

Figura 4.1 – Cabeceras de los métodos decode y execute

4.1.1 Módulo DLXMemory

Ficheros DLXMemory.cc y DLXMemory.h: se define la memoria del procesador y las interfaces de lectura, escritura e internas. En esta unidad se permite acceder a un espacio de direcciones de 32 bits de byte a byte, de media palabra o acceso a la palabra. Hemos implementado un conjunto pequeño de memoria para que pueda funcionar en cualquier ordenador. La estructura del mapping la página tendrá: 2 ^ (32-15) * 4 == 512K de espacio.

```
// Interfaces de lectura
uint8_t read_byte(DLX_addr_t addr);
uint16_t read_half(DLX_addr_t addr);
uint32_t read_word(DLX_addr_t addr);
uint64_t read_qword(DLX_addr_t addr);

// Interfaces de escritura
void write_byte(DLX_addr_t addr, uint8_t value);
void write_half(DLX_addr_t addr, uint16_t value);
void write_word(DLX_addr_t addr, uint32_t value);
void write_qword(DLX_addr_t addr, uint64_t value);
```

Figura 4.2 – Cabeceras para le lectura y escritura a memoria

4.1.2 Módulo MachineState

Ficheros DLXMachineState.cc y DLXMachineState.h: definimos el banco de registro que constará de 32, definimos PC y NPC, inicializamos los registros a cero, además hemos definido los métodos para cambiar y pedir el valor de estos registros, también lo hemos hecho para PC. Por último, hemos realizado un reset de PC y de SP (la pila), hemos definido la dirección 0xdeadbeef que será la que detenga la máquina.

```
static const int ARCH_REGS = 32;
uint32_t regs[ARCH_REGS];
uint32_t PC;
uint32_t NPC;
```

Figura 4.3 – Definición de los registros, PC y NPC

4.1.3 Módulo DLXLoader

Ficheros DLXLoader.cc y DLXLoader.h: el nombre es bastante descriptivo, cargamos un fichero, que será el programa en memoria, definimos los parámetros con variables enteras sin signo (uint32_t) que nos indicará en los métodos definidos posteriormente, el tamaño del texto, su comienzo y su fin.

4.2 Módulo Pipelined

Ficheros Pipelined.cc y Pipelined.h: tiene una función main, que crea los módulos, correspondientes, cada uno de ellos a una etapa del procesador DLX, crea el reloj del sistema, hace un reset de las señales, internconecta los módulos adyacentes y establece la configuración para generar el fichero de traza VCD.

4.2.3 Módulos Pipelined-IF, Pipelined-ID, Pipelined-EX, Pipelined-MEM, Pipelined-WB, Pipelined-RESET

Cada modulo tiene en su fichero .h con la definición de los puertos de entrada y salida, la señal de reloj, la señal de reset, una señal booleana que indica si la próxima etapa está lista para recibir el dato, así como un constructor, y el lanzamiento de una hebra que realiza una llamada al respectivo método que se encuentra en el fichero .cc con el mismo nombre.

```
// Crear los módulos
Pipelined_RESET stage_reset("reset_stage");
stage_reset.debug = debug;
Pipelined_IF stage_if("fetch_stage");
stage_if.debug = debug;
  ...
// Crear el reloj del sistema
sc_clock clk("clock");
stage_reset.clock(clk);
stage_if.clock(clk);
  ...
// Reset signal
sc_signal<bool> reset;
stage_reset.reset(reset);
stage_if.reset(reset);
...
// Internconecta los modulos adyacentes
sc_signal<StateIF2ID> from_if_2_id;
stage_if.to_decode(from_if_2_id);
stage_id.from_fetch(from_if_2_id);
...
// Interconecta las señales de ready
sc_signal<bool> fetch_ready;
stage_if.ready(fetch_ready);
...
// Configuracion de la traza
sc_trace_file *tf = sc_create_vcd_trace_file("Pipeline-Trace");
sc_trace(tf, from_if_2_id, "from_if_2_id");
...
// Inicio de la ejecucion
reset.write(true);
sc_start(cycles);
sc_close_vcd_trace_file(tf);
```

Figura 4.4 –Ejemplo de la creación del módulo de la etapa IF del fichero Pipelined.cc

```
// Definicion de los puertos

sc_out<StateIF2ID> to_decode;
sc_inout<bool> decode_ready;

sc_in<State2IF> from_wb;
sc_inout<bool> reset;
sc_in_clk clock;
sc_inout<bool> ready;
```

Figura 4.5 –Definición de los puertos de la etapa IF del fichero Pipelined-IF.cc

4.3 Jerarquía de los ficheros

4.3.1 Diagrama UML

En el siguiente diagrama observamos una abstracción de las relaciones que existe entre cada uno de los módulos del procesador DLX:

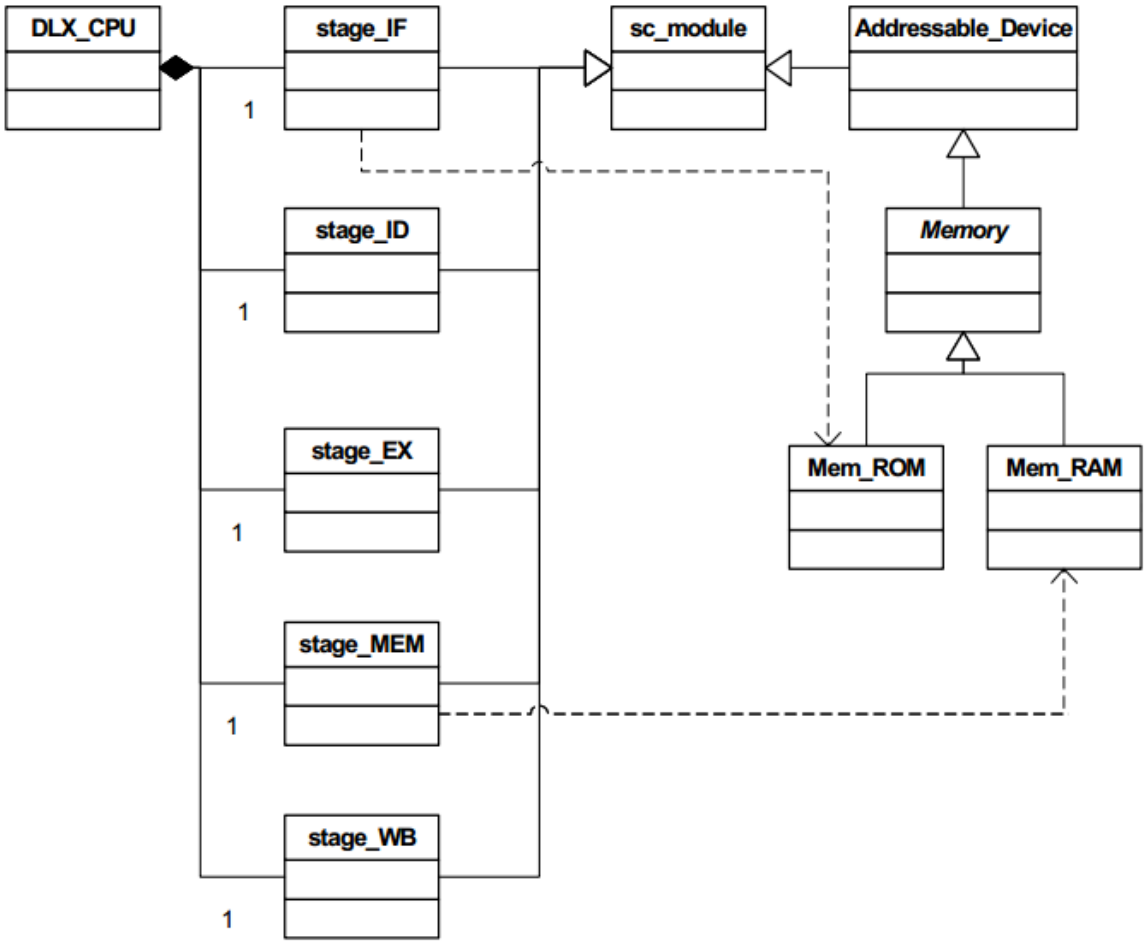


Figura 4.6 –Diagrama UML con las relaciones

4.3.2 Diagrama UML de dependencias

Vamos a mostrar un diagrama UML con las dependencias entre los ficheros explicados anteriormente, así como sus variables y métodos más importantes:

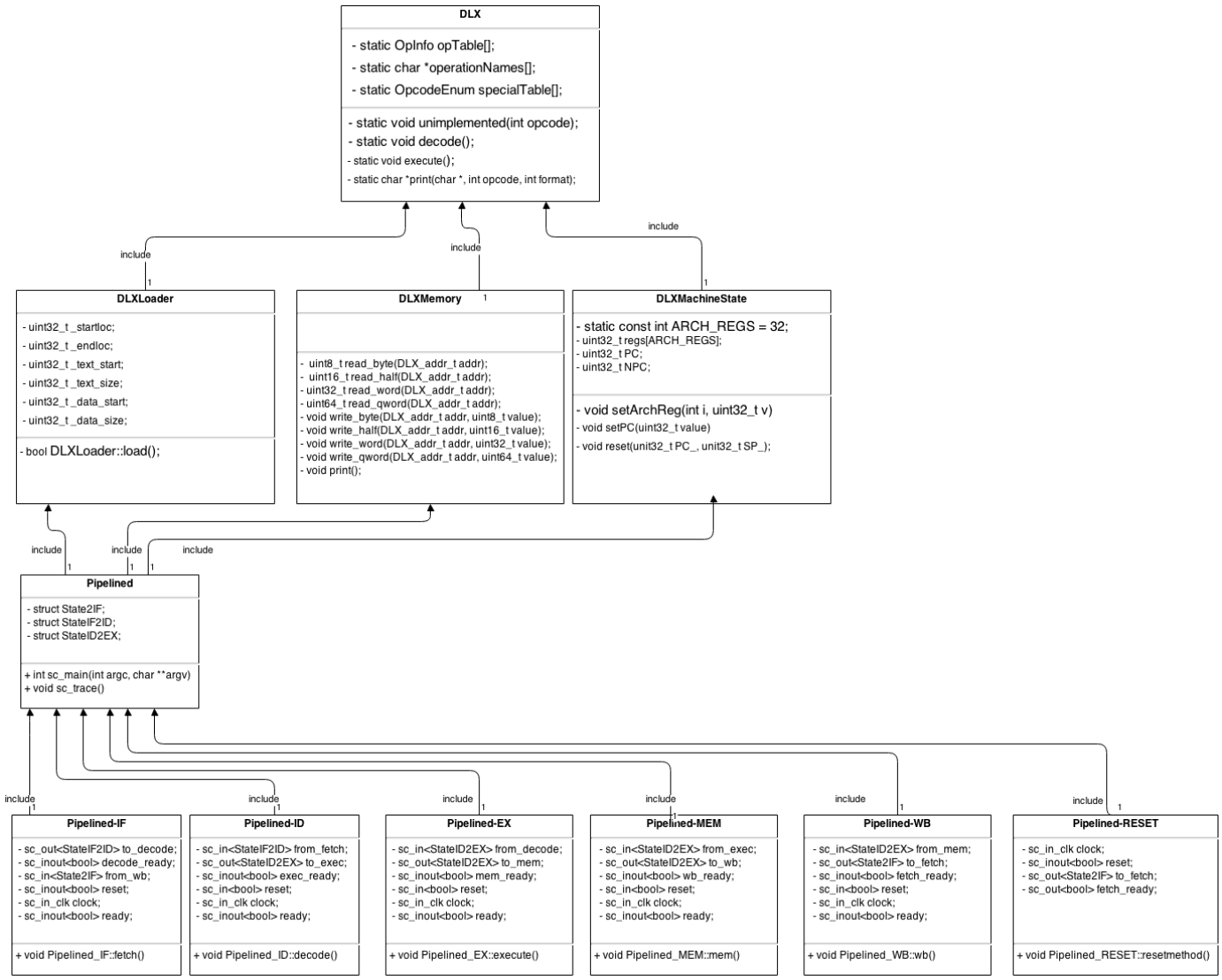


Figura 4.7 –Diagrama UML con sus dependencias

4.3.3 Diagrama de Flujo

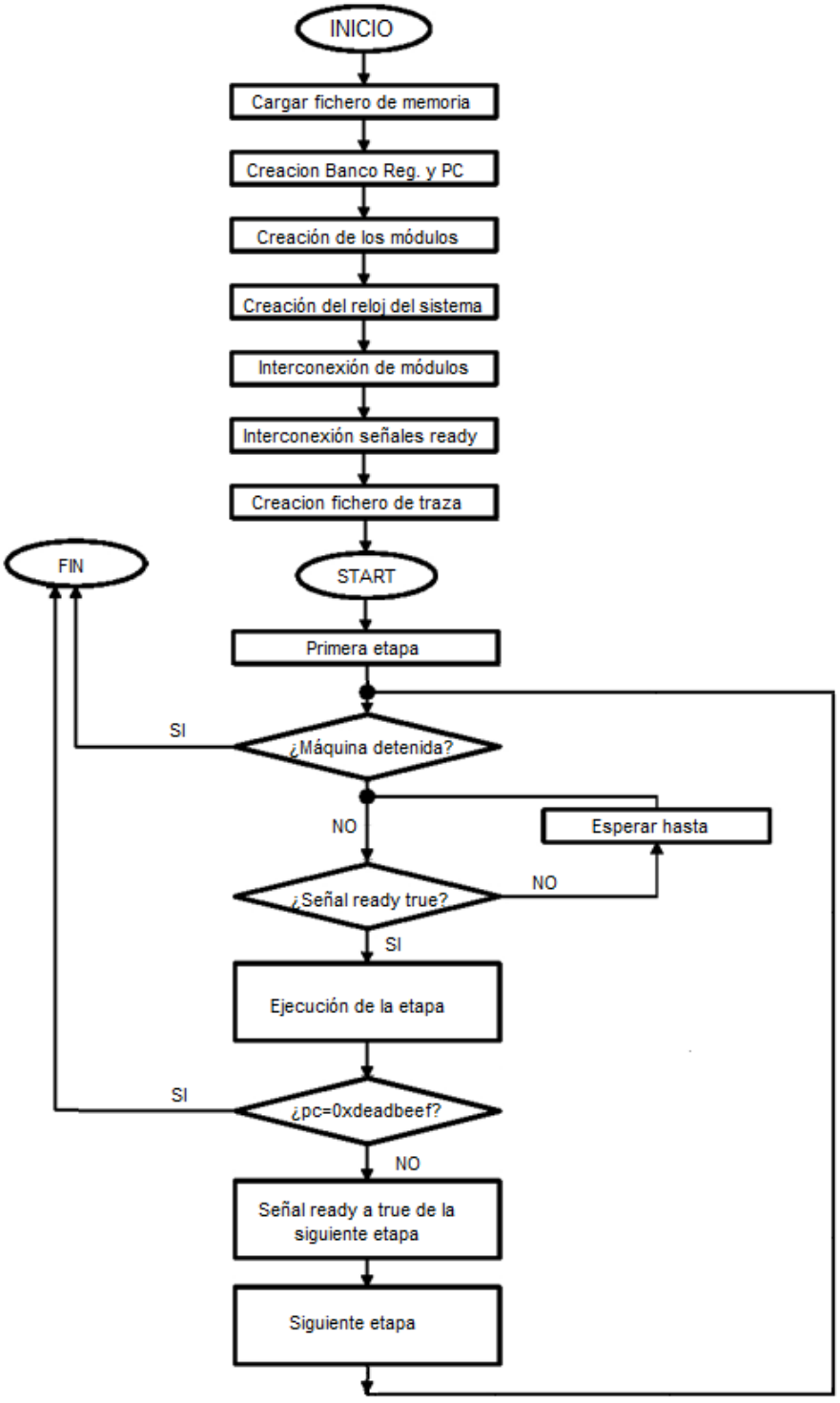
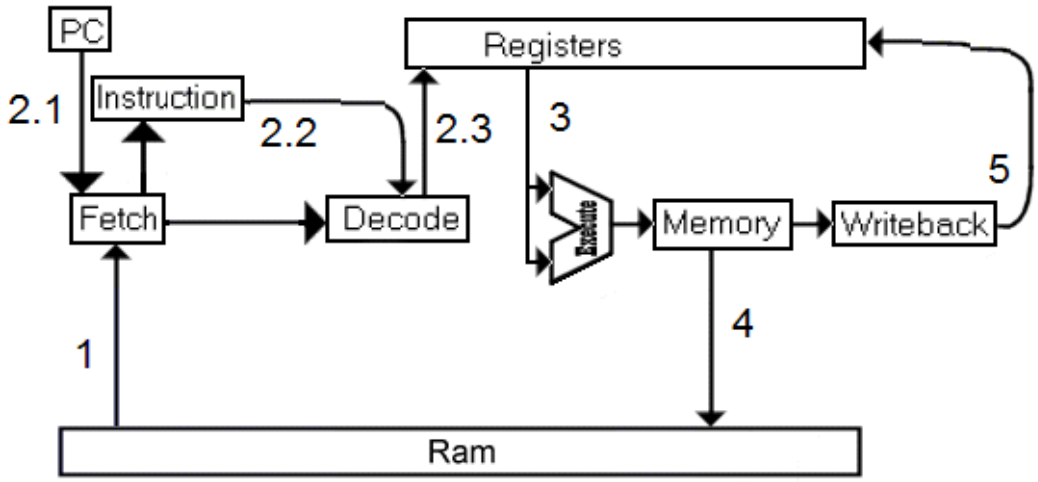


Figura 4.8 –Diagrama de flujo del funcionamiento del procesador

4.3.4 Funcionamiento del procesador DLX

El funcionamiento del procesador DLX se ha explicado etapa por etapa, tanto a nivel de arquitectura, como el diseño en la programación en SystemC, pero es posible que aún hayan dudas acerca del funcionamiento completo de nuestro procesador DLX. Por eso, se muestra el siguiente diagrama:

El procesador DLX segmentado tiene 5 etapas: IF (Búsqueda de la instrucción), ID (Decodificación de la instrucción), EX (Ejecución), MEM (Memoria) y el WB (Escritura).



1.- Obtener las instrucciones de la RAM (memoria de programa),

2.1-Debe conocer la posición de memoria donde se encuentra dicha instrucción (PC–Program Counter).

2.2-Esta etapa es responsable de la decodificación de la instrucción.

2.3 - La evaluación de la condición de ramificación y la selección de los registros de operandos.

3.- La tercera etapa se encarga de la aritmética y cálculos lógicos como así como el cálculo de direcciones de memoria.

4.- La memoria RAM (datos memoria) el acceso se lleva a cabo en la cuarta etapa.

5.- Finalmente los resultados se vuelven a escribir en el registro de destino, si es necesario. Al final del ciclo de cada etapa debe incrementarse PC que apuntará a la siguiente instrucción.

Capítulo 5: Simulación

Para proceder a la simulación previamente hay que generar un fichero .vcd, que es el fichero de traza, el cuál abriremos con la herramienta GTKWave. Para ello creamos un fichero de traza con su nombre, como se muestra en la siguiente sentencia:

```
sc_trace_file *tf = sc_create_vcd_trace_file("Pipeline-Trace");
```

Figura 5.1 – Creación del fichero de traza

Después añadimos al fichero las señales que queremos que aparezcan en el fichero, indicando en los parámetros el fichero, la señal y un nombre descriptivo. En nuestro ejemplo vamos a trazar las salidas de cada una de las etapas del procesador:

```
sc_trace(tf, from_mem_2_wb, "from_wb_2_if");
sc_trace(tf, from_if_2_id, "from_if_2_id");
sc_trace(tf, from_id_2_ex, "from_id_2_ex");
sc_trace(tf, from_ex_2_mem, "from_ex_2_mem");
sc_trace(tf, from_mem_2_wb, "from_mem_2_wb");
```

Figura 5.2 – Añadimos las señales para mostrar en la traza

Por último, tenemos que iniciar la ejecución y al finalizar cerrar el fichero de traza con esta sentencia:

```
sc_close_vcd_trace_file(tf);
```

Figura 5.3 – Cerramos el fichero de traza

5.1 Simulación de la etapa de Búsqueda (BUSQ)

Como ya habíamos comentado, esta etapa se encarga de la búsqueda de la instrucción en la cache de instrucciones. Además, esta parte se encarga de toda la lógica necesaria para generar el próximo PC (incrementos, saltos condicionales, saltos incondicionales, etc.). PC se encargará de seleccionar la instrucción que se va a ejecutar en siguiente lugar, encargándose de los saltos sin ayuda del módulo ALU, y enviándolo a este módulo, cuando su valor cuando sea necesario.

Debemos observar que la búsqueda la instrucción sea la correcta comprobando que las instrucciones que se muestran en hexadecimal, correspondan a las que se ejecutan secuencialmente en el programa que hemos cargado, además de que PC la seleccione correctamente y avanza según lo previsto.

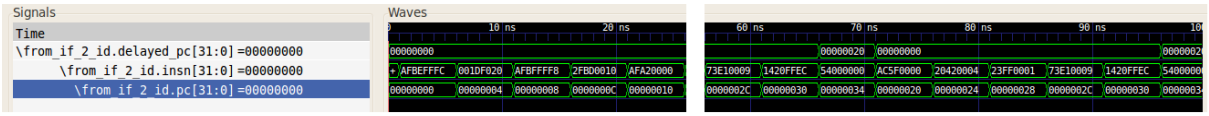


Figura 5.4. –Simulación de la etapa IF

```
        sw    -4(r29),r30
        add   r30,r0,r29
        sw    -8(r29),r31
        subui r29,r29,#16
        sw    0(r29),r2
        sw    4(r29),r3
        addi  r2,r0,#8192
        addi  r31,r0,#0
L11:
        sw    (r2),r31
        addi  r2,r2,#4
        addi  r31,r31,#1
        slei  r1,r31,#9
        bnez  r1,L11
```

Figura 5.5 – Fragmento del comienzo del programa cargado

La primera instrucción es un almacenamiento en memoria, el formato es de tipo I, y la instrucción en hexadecimal corresponde a AFBEFFFC:

101011	11101	11110	1111 1111 1111 1100
OPCODE	r29	r30	-4

La siguiente instrucción es una operación aritmética, la suma (add), el formato es de tipo R, y la instrucción en hexadecimal corresponde a 001DF020:

000000	00000	11101	11110	-	10000
OPCODE	r0	r29	r30		FUNC

Vamos a avanzar hasta los 70 ns, el valor de delayed_pc apunta a 00000020, aquí se ha producido un salto por lo que la siguiente instrucción a ejecutar va a apuntar a esta posición, se ha evaluado la instrucción corresponde a **bnez r1, L11,** como no se cumple la condición se vuelven a ejecutar las instrucciones a partir de la etiqueta L11 con la instrucción **sw (r2),r31.**

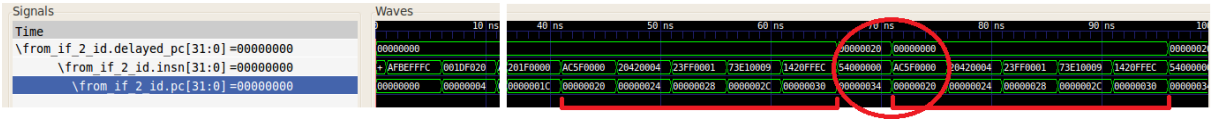


Figura 5.6 - Se produce un salto y se vuelve a ejecutar a partir de la posición de PC –> 00000020

Vamos a mostrar la correcta finalización del programa, se ejecuta la instrucción de salto al registro 31, con la instrucción **jr r31, de tipo J,** y valor hexadecimal 4BE00000, este registro almacena la dirección 0xdeadbeef, que indica la detención del sistema.

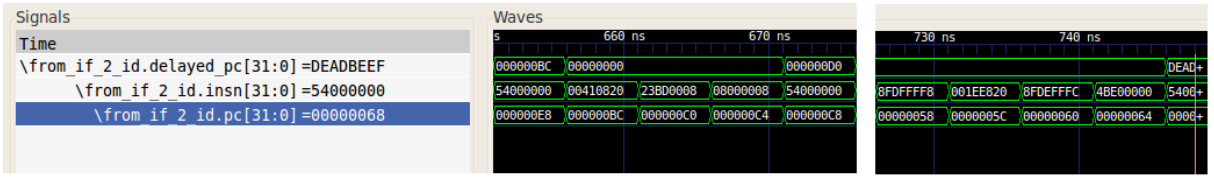


Figura 5.7 - Detención del sistema

5.2 Simulación de la etapa de Decodificación/Lectura (DE/L)

El decodificador es la unidad nos permite reconocer la instrucción que entra en nuestro procesador. Además, se encarga de gestionar todos los puntos de control necesarios que tenemos que activar para la instrucción que estemos procesando en ese momento.

Hemos mostrado todas las señales de salida de esta etapa en la traza de la simulación para comprobar que la instrucción entrante se ha decodificado correctamente.

Vamos a analizar la primera instrucción **sw -4(r29), r30**, a partir de lo que nos muestra la traza. En la Figura 5.8 podemos observar que la señal **format** vale 1, indicando que es de tipo I, la señal **rd** apunta al registro 30 con el valor 0000001E, además la señal **imm**, el valor inmediato vale –4, por último tenemos que tener en cuenta que la señal de **pc** y **delayed_pc**, funcionan correctamente.

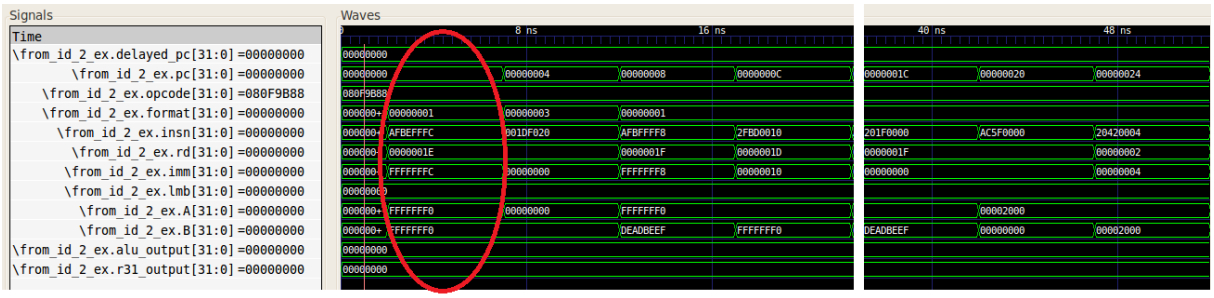


Figura 5.8 - Decodificación de una instrucción de tipo I en la etapa ID

Análogamente, analizamos una instrucción que sabemos que es de tipo R, para comprobar si se ha decodificado correctamente, la siguiente instrucción **add r30, r0, r29,** aparece en la Figura 5.9 la señal format vale 3, indicando que es de tipo R, la señal rd también apunta al registro 30 con el valor 0000001E, el valor inmediato vale 0, ya que no se trata de una instrucción de tipo I.

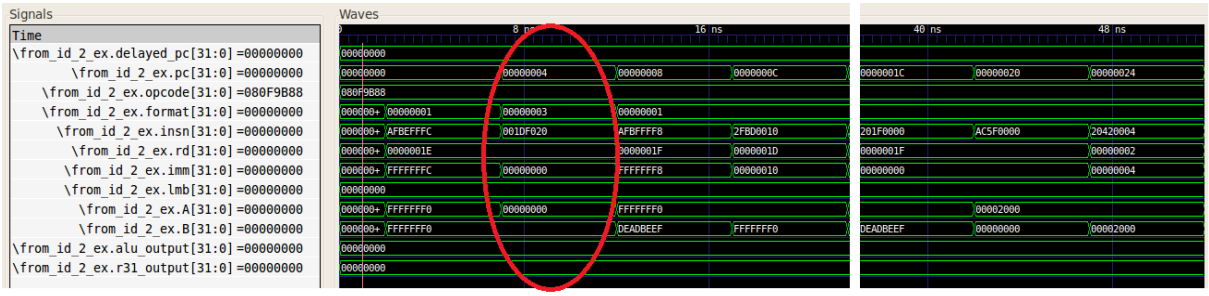


Figura 5.9 - Decodificación de una instrucción de tipo R en la etapa ID

Por último, además de comprobar que el programa termina correctamente en esta etapa, podemos observar como que la ultima instrucción, **jr r31**, su señal format vale 2, por la tanto es de tipo J, tal como esperábamos, la señal imm que es el valor inmediato vale 0, ya que no se trata de una instrucción de tipo I.

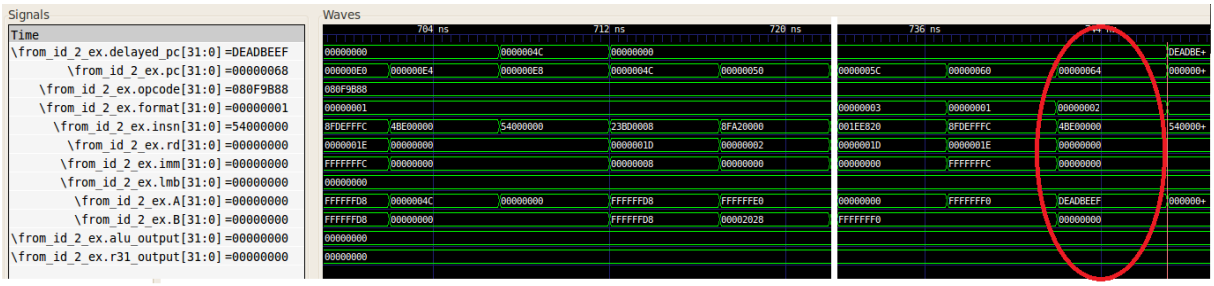


Figura 5.10 - Decodificación de una instrucción de tipo J en la etapa ID

5.3 Simulación de la etapa de la ALU (Ejecución)

La ALU soportar la mayoría de las operaciones que se va a realizar en nuestro procesador. También se encargará de las operaciones de comparación, las cuales devuelven 0 o 1 dependiendo de los resultados que de la ALU y de la selección del valor a escribir en el banco de registros.

Las señales de salida mostradas en esta etapa en la traza de la simulación son las mismas respecto de la etapa anterior, por lo cual algunas señales mostrarán lo mismo, y otras como **alu_output** y **r31_ouput** son las que analizaremos con detenimiento.

Vamos a analizar las mismas instrucciones que en la etapa anterior, como ejemplo de cada uno de los tres formatos de instrucciones.

La instrucción **sw -4(r29),r30** , en la Figura 5.11 observamos que la señal A y B que son los registros donde se cargan los valores de los registros para las operaciones, estos valen 0 debido a que se trata de una operación de almacenamiento, la señal alu_output es el resultado de sumar el valor inmediato y el registro fuente rs1.

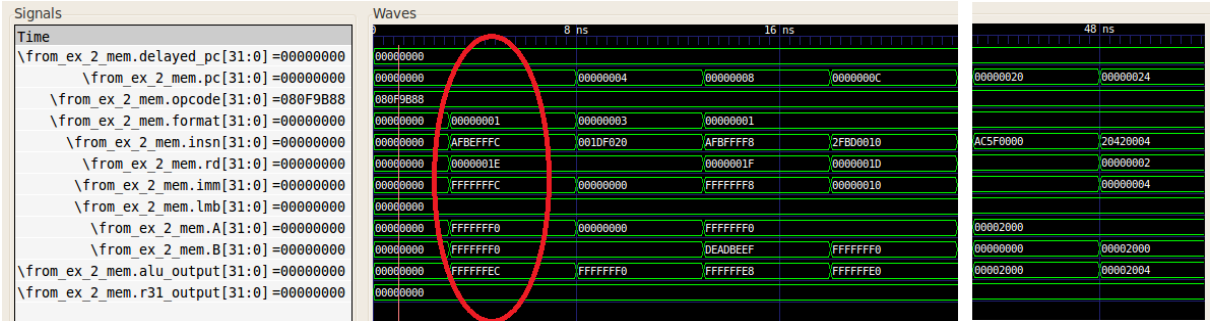


Figura 5.11 - Ejecución de una instrucción de tipo I en la etapa EX

La siguiente instrucción **add r30, r0, r29,** aparece en la Figura 5.12 observamos que la señal A y B que son los registros donde se cargan los valores de los registros para las operaciones, aparecen cargados con valores 00000000 y FFFFFFF0 respectivamente, al tratarse de un operación de suma de tipo R, en la salida de la alu, la señal alu_output, aparece la suma esperada, es decir, el valor FFFFFFF0.

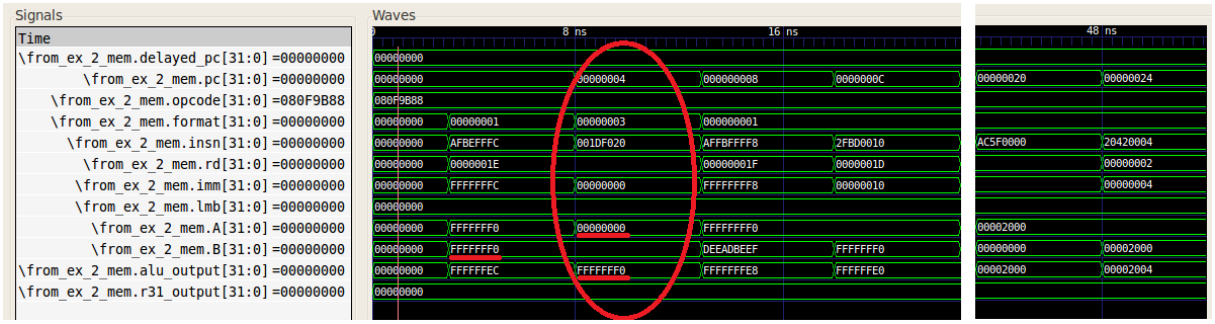


Figura 5.12 - Ejecución de una instrucción de tipo R en la etapa EX

En esta etapa, también termina el programa como se esperaba, la última instrucción a ejecutar es **jr r31**, y podemos apreciar como el valor almacenado en la señal de salida de la alu, la señal alu_output, es la dirección 0xdeadbeef, esta es la posición donde apuntará delayed_pc para que se detenga el sistema.

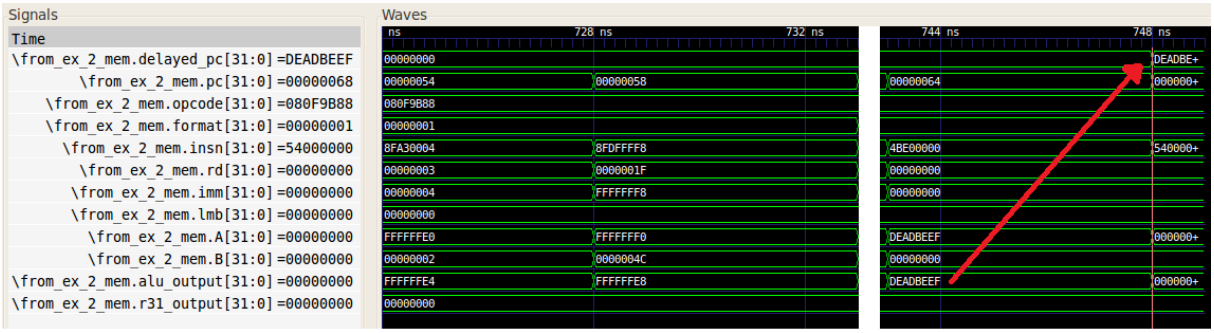


Figura 5.13 - Ejecución de una instrucción de tipo J en la etapa EX

5.4 Simulación de la etapa de Memoria (Mem)

La dirección de memoria viene dado por el resultado de la ALU, la señal alu_output, y el dato viene dado por el contenido de uno de los puertos (B) del banco de registros, la señal B.

En la simulación de esta etapa, vamos a analizar si se ejecuta correctamente la instrucción LW que lee una palabra desde memoria de la siguiente forma rd=MEM (rs1+extend (inmediato)), para ello observamos la instrucción lw r3, (r30) del fragmento siguiente:

```
_fib:
   sw     -4(r29),r30
   add    r30,r0,r29
   sw     -8(r29),r31
   subui  r29,r29,#16
   sw     0(r29),r2
   sw     4(r29),r3
   lw     r3,(r30)
   sgti   r1,r3,#1
   beqz   r1,L17
```

Figura 5.14 – Fragmento de la ejecución de fibonacci

La instrucción de carga en memoria, es de tipo I, y la instrucción en hexadecimal corresponde a 8FC30000:

100011	11110	00011	0000 0000 0000 0000
OPCODE	r30	r3	0

Como el valor inmediato vale 0, el dato contenido en el registro r3 que vale 2, en la posición de memoria FFFFFFD8.

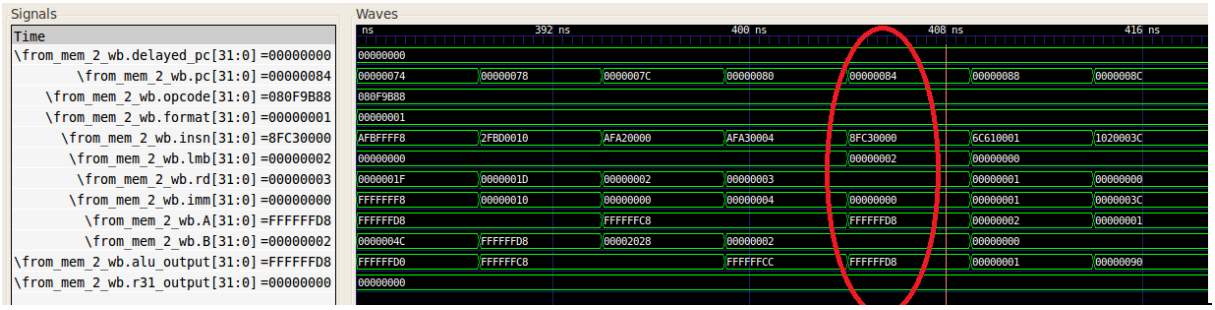


Figura 5.16 - Lectura de una instrucción LW en la etapa MEM

5.5 Simulación de la Etapa de Escritura (Esc)

Por último, la quinta etapa consiste en el almacenamiento en el banco de registros del dato obtenido por la ALU (en el caso de una operación) o por la unidad de memoria (en el caso de un almacenamiento en registro).

Partiendo de la Figura 5.14 vamos a analizar si se ejecuta correctamente la instrucción **SW** que escribe una palabra en memoria (MEM (rs1+extend (inmediato) =rd), para ello observamos la instrucción **sw 4(r29),r3**.

La instrucción de almacenamiento en memoria, es de tipo I, y la instrucción en hexadecimal corresponde a AFA30004:

101011	11101	00011	0000 0000 0000 0100
OPCODE	r29	r3	4

Como el valor inmediato vale 4, se escribe en la posición de memoria FFFFFFC8 + 4, es decir, la posición FFFFFFCC.

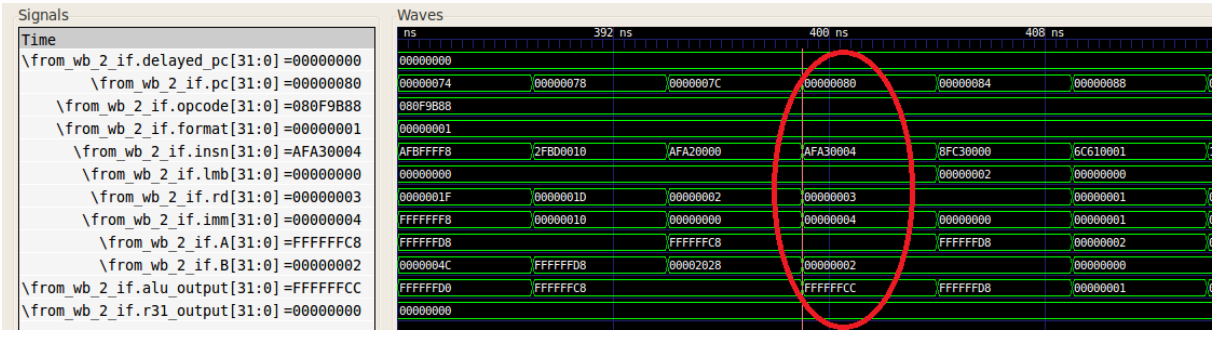


Figura 5.17 - Escritura de una instrucción SW en la etapa WB

```
.text
      .align 2
.proc _entry
.global _entry

_entry:

      sw    -4(r29),r30 ; push fp
      add   r30,r0,r29  ; fp = sp
      sw    -8(r29),r31 ; push ret
      addr
      subui r29,r29,#16 ; alloc
      local storage
      sw    0(r29),r2
      sw    4(r29),r3
      addi  r2,r0,#8192
      addi  r31,r0,#0
L11:
      sw    (r2),r31
      addi  r2,r2,#4
      addi  r31,r31,#1
      slei  r1,r31,#9
      bnez  r1,L11
      nop   ; not filled.
      addi  r29,r29,#-8
      addi  r3,r0,#2
      sw    (r29),r3
      jal   _fib
      nop   ; not filled.
      addi  r29,r29,#8
      lw    r2,0(r29)
      lw    r3,4(r29)
      lw    r31,-8(r30)
      add   r29,r0,r30
      lw    r30,-4(r30)
      jr    r31
      nop
.endproc _entry
```

```
      .align 2
.proc _fib
.global _fib

_fib:
      sw    -4(r29),r30 ; push fp
      add   r30,r0,r29  ; fp = sp
      sw    -8(r29),r31 ; push ret
      addr
      subui r29,r29,#16 ; alloc
      local storage
      sw    0(r29),r2
      sw    4(r29),r3
      lw    r3,(r30)
      sgti  r1,r3,#1
      beqz  r1,L17
      nop   ; not filled.
      addi  r29,r29,#-8
      addi  r1,r3,#-1
      sw    (r29),r1
      jal   _fib
      nop   ; not filled.
      add   r2,r0,r1
      addi  r1,r3,#-2
      sw    (r29),r1
      jal   _fib
      nop   ; not filled.
      add   r1,r2,r1
      addi  r29,r29,#8
      j     L16
      nop   ; not filled.
L17:
      addi  r1,r0,#1
L16:
      lw    r2,0(r29)
      lw    r3,4(r29)
      lw    r31,-8(r30)
      add   r29,r0,r30
      lw    r30,-4(r30)
      jr    r31
      nop
.endproc _fib
```

Figura 5.18 - Código completo del programa de fibonacci

Capítulo 6: Comparativa Verilog y SystemC

Los lenguajes de descripción de hardware (HDL) se han utilizado tanto para diseños de arquitecturas en sistemas digitales como referencia en el análisis y validación de estos circuitos. Ofrecen una metodología de diseño jerárquica en las modalidades de bottom-up y top-down. En particular el lenguaje SystemC, permite programar en diferentes niveles de abstracción: nivel de puertas, nivel trasferencia de registros, nivel comportamiento, nivel transaccional, etc. Debido a que el nivel comportamiento representa un nivel aceptable (ni tan abstracto ni tan detallado) de programación (permite estructuras case, if, for), actualmente la mayoría de diseños de circuitos se realiza utilizando este nivel de abstracción.

Además, cabe recordar que, SystemC es una biblioteca estandarizada por el IEEE portable en C++ y usada para modelar sistemas con comportamiento concurrente. SystemC suministra mecanismos valiosos para modelar hardware mientras se usa un ambiente compatible con el desarrollo de software. Este lenguaje también provee estructuras orientadas a modelar hardware que no están disponibles en los lenguajes de programación normales.

El motor de ejecución de SystemC permite la simulación de procesos concurrentes al igual que Verilog (Lenguaje de descripción de hardware), VHDL(Lenguaje de descripción de hardware) o cualquier otro lenguaje de descripción de hardware. Esto es posible gracias al uso de un modelo cooperativo multitarea, el cual corre en un kernel que orquesta el intercambio de procesos y mantiene los resultados y evaluaciones de dichos procesos alineados en tiempo.

Por otro lado, ya que C++ implementa la orientación a objetos (paradigma que de hecho fue creado para técnicas de diseño de Hardware) la abstracción de datos, propiedades, comportamientos y atributos de un modulo (clase) puede ser el principio para elaborar un diseño jerárquico, que es lo que se ha realizado.

Cuando comparamos SystemC con otros lenguajes HDL, por lo general en relación con VHDL y Verilog, la documentación es limitada y algo escasa. Esto se debe principalmente al hecho de que SystemC es un recién llegado a este campo, pero también porque es meramente una extensión de un lenguaje ya existente.

Algo muy habitual, cuando se trata de comparar dos lenguajes de programación es basarse en el número de líneas de código necesarias y su tiempo de ejecución para que ambos logren una misma tarea. Además, debemos de tener en cuenta otros parámetros, como la optimización del código, sus características, la existencia o ausencia de constructores que facilitan la codificación, etc.

Otros puntos utilizados como base para la comparación son: la eficiencia de los métodos y construcciones del lenguaje, descripción comportamiento de la señal, la semántica de programación y facilidad de implementación. Así pues, vamos a comparar diferentes aspectos entre el Verilog y SystemC, de acuerdo con las medidas mencionadas anteriormente.

6.1 Diferencia en las construcciones

Tanto Verilog y SystemC utilizan módulos como entidades de diseño. Mientras Verilog tiene como palabra reservada **module** para construirlo, SystemC necesita realizar una llamada a un constructor cuya palabra reservada es **sc_module ()** para declarar el cuerpo de la tarea.

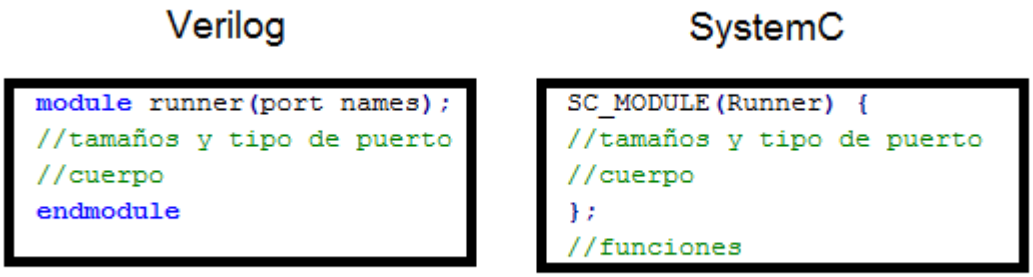


Figura 6.1 – Diferencias sintácticas en la declaración de un módulo

Cabe señalar que Verilog requiere definir los puertos una vez en la línea de declaración del módulo, y de nuevo inmediatamente después, para especificar sus tamaños. SystemC es capaz de hacer esto en un solo paso, además puede declarar una función separada de su cuerpo, como en C. Esta propiedad del lenguaje puede ser visto como una ventaja, ya que un módulo puede llamar a varios procesos diferentes.

Verilog sólo es capaz de llevar a cabo un proceso si se encuentra dentro de un módulo, lo que significa tener que escribir un módulo para cada proceso que necesita ser llamado por módulos externos. La ausencia de una construcción de alto nivel que replique la estructura, a menudo puede conducir a la escritura de código repetido, o difíciles de optimizar.

Al modelar un proceso en Verilog, la práctica común es tener siempre un constructor en el cuerpo de la función a ser evaluado. En SystemC, las funciones se escriben como miembros de la clase del módulo que se diseña, por lo que permite al diseñador integrar fácilmente funcionalidades adicionales en el mismo diseño.

Acerca de la temporización, también tenemos algunas diferencias, SystemC incluye una función para construir una señal de reloj utilizando **sc_clock().** En su lugar, Verilog algo más generalizado, para crear un reloj: definirlo como un módulo. Aunque esta técnica puede parecer ineficiente, en realidad es más natural para un diseñador principiante, ya que no necesita conocer construcciones nuevas.

Verilog

```
module m555 (clock);
    output clock;
    reg clock;
    initial
        #5 clock = 1;
    always
        #50 clock = ~clock;
endmodule
```

SystemC

```
sc_clock m555("m555", 20, 0.5, 5, true);
```

Figura 6.2 – Declaraciones sintácticas de un reloj

SystemC permite utilizar tres tipos de procesos en la descripción de un modelo: métodos, threads y threads sincronizados. Los métodos se ejecutan cuando se producen cambios en las señales que se encuentran en su lista de sensibilidad. Al termina, un método devuelve el control al kernel de simulación.

Los threads son similares a los métodos, pero también pueden ser suspendidos y reactivarse en la ocurrencia de un evento especificado. Por último, los threads síncronos, son un caso especial de threads, en la que la activación se efectúa en un flanco específico de una señal, la programación de estos tres tipos de procesos de realizan dentro de de un constructor, cuya sintaxis en SystemC es sc_ctor().

```
sc_ctor (Runner) {
    sc_method (tick);
    sensitive_pos (clock);
}
```

Figura 6.3 – Declaración de un método en SystemC

Verilog no es capaz de diferenciar entre procesos, pero permite flexibilidad para imitar el comportamiento encontrado en los tres escenarios descritos anteriormente. Esto se consigue por medio de la temporización orientada construcciones tales como:

```
always @ (condition)
```

Aquí, la condición puede ser una señal o un acontecimiento, en cuyo caso se puede encontrar el identificador entre paréntesis. Por esta razón, una función también puede ser disparada por flanco:

```
always @ (posedge clock)
```

Ambos lenguajes permite definir variables, aunque con algunas diferencias, Verilog sólo soporta tipos de vectores de bits y subdivide sus tipos de datos en dos categorías principales: registers y nets.

Un tipo registres implica el almacenamiento y consta de las siguientes variantes:

reg	Variable sin singo de cualquier tamaño en bits.
integer	Variable entera con signo de 32 bits.
time	Variable sin signo de 64 bits
real	Variable punto flotante de doble precisión

SystemC, ofrece un amplio conjunto de tipos de datos, y se han mejorado para admitir varios dominios de diseño y niveles de abstracción. Los tipos fixed-points permiten una simulación rápida. Los tipos de datos enteros con precisión arbitraria se pueden utilizar para los cálculos con números grandes. Además de disponer del tipo Bits, vectores de bits y tipos de datos con decimales. Tales tipos no tienen una limitación de tamaño.

La definición una señal es:

sc_signal <base_type> signal_name;

, donde base_type corresponde a uno de los tipos de base de C++ como entero, real, char, etc. Del mismo modo, los puertos se declaran con la misma sintaxis, pero utilizan los identificadores **sc_in <>**, **sc_out <>** o **sc_inout <>**.

Al igual que en C o C++, SystemC puede incluir las bibliotecas definidas por el usuario que contienen funciones y estructuras de datos, y utilizarlas a lo largo de un programa. Verilog no ofrece tal reutilización, además, en ciertos casos puede ser un inconveniente, que se debe a la naturaleza del lenguaje. [12]

6.2 Descripción de un diseño

Verilog es muy eficiente en diseños estructurales, este lenguaje posee port-mapping y técnicas para instanciar módulos fácilmente. Sin embargo, a la hora de cambiar entre varias capas de abstracción puede resulta problemático, ya que no existen construcciones dedicadas a ayudar a la modelización de los diseños más grandes. Por esta razón, al describir una estructura puede aparecer como una única, independientemente de la jerarquía de componentes.

SystemC le lleva ventaja en el modelado de sistemas grandes, por medio de construcciones especiales. La posibilidad de declarar un **sc_main ()** como rutina que gestiona todos los módulos, en la temporización y esquemas de transferencia de datos, es una gran ventaja. Esto también facilita el proceso de depuración de las mismas descripciones.

A los programadores principiantes les resultará más fácil de aprender Verilog, debido a su sintaxis generalizada. Además, no requiere que el usuario esté familiarizado con otros lenguajes que no sean HDL, como es el caso de SystemC, que se necesita conocimientos previos de C, y conocimientos orientados a objetos de C++.

6.3 Determinismo en ambos lenguajes

El modelo de simulación de Verilog garantiza un cierto nivel de determinismo sobre el orden en que se programa. Las declaraciones ubicadas dentro de un bloque begin-end se ejecutan secuencialmente en el orden en que aparecen en el interior de dicho bloque. A pesar de que un proceso puede suspenderse en un determinado caso y más tarde recuperar el control, sus declaraciones se ejecutarán en el orden indicado en el bloque begin-end.

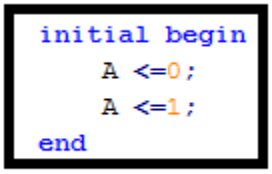


Figura 6.4 – Bloque begin-end

Por ejemplo, cuando se ejecuta el bloque de código de la figura 5.5 habrá dos eventos añadidos a la cola de actualización de asignación no bloqueante. La regla anterior exige que se introduzcan y se llevan a cabo en el mismo orden en que aparece en el código. Por lo tanto al final del paso de tiempo 1, la variable se le asignará un 0, y después se le asignará un 1.

Hay varias cosas que los diseñadores pueden hacer para controlar no-determinismo al programar en SystemC. En primer lugar, el uso de canales, tales como señales hardware (sc_signal, etc.) y fifos, nos ayudarán a definir un comportamiento determinista a nivel global.

En segundo lugar, debe haber una conciencia general entre los diseñadores, para que el orden de ejecución de un thread dentro de una fase de simulación particular no sea específico y dependa de la implementación. Sin embargo, cuando el mismo diseño se simula varias veces utilizando el mismo estímulo y la misma versión del simulador, el orden del thread entre diferentes trazas no varía.

6.4 Diferencias entre Verilog y SystemVerilog

Verilog es un lenguaje HDL, utilizado por los desarrolladores para describir hardware. El nombre de Verilog se deriva de la capacidad del lenguaje para verificar y registrar (log) el desarrollo y la ejecución de los componentes electrónicos.

SystemVerilog es una extensión de Verilog, y se expande en los protocolos HDL, debido a su relación son muy similares. Sin embargo, hay algunos factores clave que permiten distinguir a uno del otro.

SystemVerilog busca enfocar las capacidades de Verilog y mejorar la capacidad del lenguaje para verificar chips basados en IP.

Además, SystemVerilog mejora a Verilog con la implementación del lenguaje de programación C, lo que permite a los desarrolladores implementar los protocolos HDL en los lenguajes de programación más conocidas, como C y C++. Ocurre algo similar con SystemC.

A diferencia de Verilog, el protocolo HDL de SystemVerilog contiene verificación basada en objetos, lo que reduce algunos de los trabajos de implementación de módulos de prueba externo para el proceso de verificación.

6.5 Algunas características de SystemVerilog que mejoran Verilog

Los operadores de asignación de procedimientos (<=, =) ahora pueden operar sobre arrays. Al inicializar una variable, ésta también podrá operar sobre un array.

Las definiciones de los puertos (inout, input, output) se amplían y permiten para una mayor variedad de tipos de datos: struct, enum, real, y tipos multidimensionales.

La construcción del bucle for, permite la declaración de variables dentro de la instrucción for. Se mejora el control del bucle con las sentencias continue y break. Además, se añade el bucle do/while. La construcción fork/join se ha ampliado con join_none y join_any.

Las funciones ahora se pueden declarar como desiertas, es decir, éstas no devuelven nada. Los parámetros pueden ser declarados de cualquier tipo, incluidos los tipos definidos por el usuario.

6.6 Conclusiones

Para programadores principiantes, es conveniente que comiencen con Verilog (o incluso VHDL), ya que tiene una sintaxis menos extensa, y no se requieren conocimientos previos de otro lenguaje. También cuenta con una menor cantidad de construcciones para tareas específicas.

Verilog puede ser considerado como un lenguaje orientado a objetos débiles, SystemC es más adecuado para tal estilo de programación, debido a sus orígenes.

SystemC presenta más potencial para tiempos de simulación cortos, ya que su algoritmo de planificación se basa en el tipo de proceso. Tiempos de simulación más cortos combinados con una etapa de verificación reducida, hacen que SystemC sea favorable entre ambos lenguajes, a la hora de implementar diseños muy grandes. Verilog carece de construcciones de alto nivel que facilitan un proceso de este tipo de diseño.

En general, Verilog es más adecuado para diseños estructurales, ya que permite un mejor control de los módulos dentro de las mismas capas de abstracción, a pesar de que carece de gestión jerarquía de componentes.

	SystemC	Verilog
Creación de módulos	sc_module( )	Module
Creación del reloj	sc_clock	Creación de un módulo específico para el reloj
Procesos	Threads, métodos y threads sincronizados	No diferencia entre procesos, hace uso de la temporización.
Variables	Las propias de SystemC y todas las de C++ (No tiene límite de tamaño)	Registers y nets (Límite de tamaño)
Bibliotecas definidas	Sí	No
Módulos externos	Permite llamadas, ya que es un lenguaje orientado a objetos	Reescritura de código.
Nivel de abstracción	Eventos y mensajes	Estados lógicos y transiciones
Diseño arquitectural	Perspectiva hardware a nivel de sistema y perspectiva software al programador	Perspectiva de la implementación hardware
Diseño de puertas RTL	No hay modelado a nivel de puertas	Diseño lógico
Verificación de puertas RTL	TLM/RTL co-simulación.	Implementación de banco de pruebas, incluido el nivel funcional.

Tabla 6.5 – Tabla comparativa que resume algunas características de ambos lenguajes.

Capítulo 7: Conclusiones

La realización de este proyecto además de la descripción de un lenguaje, un sistema y su simulación incluye una serie de puntos que lo convierten en algo más complejo que la de presentar un ejemplo de uso con el lenguaje SystemC.

Primero habría que tener en cuenta el total desconocimiento previo del autor sobre el lenguaje con que se ha trabajado. En cuanto al lenguaje en sí no se han encontrado muchas dificultades en cuanto a su entendimiento, siendo la programación más fácil de lo esperado, partiendo de la base de unos buenos conocimientos de programación general. Los conocimientos que tenía acerca de C y C++ los cuales aprendí durante la carrera, me han facilitado el trabajo, además de tener que haber vuelto repasar sus sintaxis y el manejo de punteros.

Su estructura es muy comprensible y su lógica de programación también, además de aportar las funciones y herramientas necesarias para la resolución global del sistema descrito.

7.1 Posibles mejoras

Vale remarcar que el trabajo realizado no está cerrado ya que puede ser ampliado en varios aspectos. Aun habiendo conseguido la funcionalidad necesaria para realizar los análisis requeridos del sistema, puede verse que no se han implementado un amplio número de operaciones del conjunto de instrucciones del procesador DLX, como la implementación de las operaciones de punto flotante, o la posible carga en memoria de una imagen para utilizar operaciones de rotación para transformarla. Además, podríamos evaluar estrategias para distintas temporizaciones de reloj para así ajustar el programa de forma óptima y/o asociarle un retraso específico a cada una de las etapas.

En la actualidad la organización Accellera Systems Initiative cuenta con un grupo de investigadores que se encargan de compatiblizar el lenguaje SystemC con posibles traducciones a otros lenguajes HDL.

También es destacable el esfuerzo de distintas empresas como CADENCE y Synopsys de crear herramientas que permitan generar código RTL a partir de código C/SystemC. Esto sería una opción a realizar para una posible mejora, manejar alguna herramienta que genere esquemáticos a nivel de puertas, partiendo del código escrito en SystemC.

Otra posible línea de investigación sería, a partir de la descripción en SystemC, realizar la síntesis de alto nivel del modelado en un field programmable gate array (FPGA) y utilizar técnicas de reúso para su refinamiento. [13]

Capítulo 8: Glosario de términos

- **C++:** es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.

- **SystemC:** es una extensión del C++, que utiliza unas bibliotecas de clase para describir y simular circuitos digitales. Se publicó en 1999.

- **VHDL** (Very High Speed Integrated Circuit Hardware Description Language): Nace como proyecto del Departamento de Defensa (DoD) de EEUU (año 82) para disponer de una herramienta estándar, independiente para la especificación (modelado y/o descripción) y documentación de los sistemas electrónicos. El IEEE lo adopta y estandariza.

- **Verilog:** Software de la firma Gateway y posteriormente de Cadence. Estándar industrial hasta que apareció el VHDL como estándar IEEE. En 1990 Cadence lo hace público y el IEEE lo estandariza en 1995.

- **FIFO (**First in, first out**):** primero en entrar, primero en salir, es un concepto utilizado en estructuras de datos, contabilidad de costes y teoría de colas. Guarda analogía con las personas que esperan en una cola y van siendo atendidas en el orden en que llegaron, es decir, que la primera persona que entra es la primera persona que sale.

- **Kernel:** la mayoría de los lenguajes de modelado, SystemC, por ejemplo, utilizan un kernel de simulación. El propósito del Kernel es para asegurar que las actividades paralelas (concurrencia) se modelan correctamente.

- **RTL** (Resistor Transistor Logic): o lógica de resistencia-transistor. Fue la primera familia lógica en aparecer antes de la tecnología de integración. Pertenece a la categoría de familias lógicas bipolares, o que implican la existencia de dos tipos de portadores: electrones y huecos.

- **TLM** (Transaction Level Modeling): es una forma de modelar los sistemas digitales donde los detalles de las comunicaciones entre los módulos están separados de la implementación.

- **Fichero VCD,** los ficheros VCD , también conocido como Vale Change Dump, son archivos basados en ASCII que describen una forma de onda digital del simulador de VHDL, Verilog, o SystemC . Son archivos flexible y fácil de manejar que puede ser fácilmente manipulados por un programador con experiencia para crear simulaciones o ser utilizados como en una nueva simulación se ejecuta en ModelSim, GTKWave, etc. Se utilizan principalmente para capturar transiciones de la señal en el programa en dichos programas y llevar a cabo análisis adicionales.

- **DLX:** proporciona un buen modelo de arquitectura para estudio, no sólo por el amplio conjunto de instrucciones de este tipo de máquinas, sino también a lo fácil de entender que es su arquitectura. Esta arquitectura se viene utilizando en muchas Universidades en innumerables cursos sobre Arquitectura de Computadores.

- **UML** (Unified Modeling Language): es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad. Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. Es importante remarcar que UML es un "lenguaje de modelado" para especificar o para describir métodos o procesos. Se utiliza para definir un sistema, para detallar los artefactos en el sistema y para documentar y construir. En otras palabras, es el lenguaje en el que está descrito el modelo.

- **FPGA** (Field Programmable Gate Array): es un dispositivo semiconductor que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada 'in situ' mediante un lenguaje de descripción especializado. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinacional hasta complejos sistemas en un chip.

Bibliografía

A continuación se dan las referencias bibliográficas que se han utilizado en la elaboración del presente proyecto:

[1] SystemC 2.0.1 Language Reference Manual

[2] http://www.accellera.org/downloads/standards/systemc

[3] Alan Ma and Allan Zacharda, "Utilizaing SystemC for design and verification", http://www.edadirect.com/docs/Using_SystemC_for_Design_Verification.pdf

[4] SystemC: From the Ground Up / by David C. Black, Jack Donovan, Bill Bunton, Anna Keist. 2010.

[5] Quality-Driven SystemC Design / by Daniel Grose, Rolf Drechsler. 2010.

[6] Introducción a SystemC / by Agustín González http://profesores.elo.utfsm.cl/~agv/elo330/2s03/projects/SystemC/SystemC.pdf

[7] Modelado y simulación de sistemas usando SystemC / by Cesar Armando Fuguet Tortolero.

[8] http://rincon-linuxx.blogspot.com.es/2013/05/instalacion-de-systemc-debian-y-ubuntu.html

[9] http://www.kubuntu-es.org/wiki/howto-instalar-libreria-systemc

[10] Laboratorio de Diseño Microelectrónica / by Rafael Escaño

[11] http://electro.fisica.unlp.edu.ar/arq/downloads/Software/WinDLX/DLXinst.html

[12] Hardware Description Languages Compared: Verilog and SystemC / by Gianfranco Bonanome

[13] VLSI Design: A Practical Guide for FPGA and ASIC Implementations / by Vikram Arkalgud Chandrasetty. 2011.

Agradecimientos:

- A mi Tutor Rafael Escaño Quero por la ayuda y la comprensión recibida en el transcurso de su asignatura y en este proyecto. Al Departamento de Electrónica en general por la concesión de este proyecto.

- A mis padres y mi hermana por el apoyo y la confianza que han puesto en mi a lo largo de mi vida.

- A Cristina por su amor, paciencia y creer en mí.

- A mis compañeros de la Universidad, con los que he estado muchas horas preparando asignaturas, además de los buenos momentos que hemos pasado.