Universidad Zaragoza

Trabajo Fin de Grado en Ingeniería Informática
Especialidad en Ingeniería de Computadores
2017

Generación automática de bancos de registros en VHDL mediante una herramienta de traducción en JavaCC

Automatic generation of register banks in VHDL with a translation tool in JavaCC

Adrián Alcolea Moreno

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Departamento de Ingeniería Electrónica y Comunicaciones
Universidad Zaragoza

Director:
Javier Resano Ezcaray
Codirector:
Denis Navarro

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Adrián Alcolea Moreno,

con nº de DNI 72990964G en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado en Ingeniería Informática, (Título del Trabajo)

Generación automática de bancos de registros en VHDL mediante una herramienta de traducción en JavaCC.

Automatic generation of register banks in VHDL with a translation tool in JavaCC.

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 03 de Febrero de 2017

Fdo:

Dedicado a Bea
Gracias
por estar ahí
todo este tiempo

Generación automática de bancos de registros en VHDL mediante una herramienta de traducción en JavaCC

Resumen

La empresa de electrodomésticos Balay utiliza el lenguaje de descripción de hardware VHDL para realizar el diseño de sus circuitos integrados. En el desarrollo de los mismos, la necesidad de modificar habitualmente los bancos de registros acaba suponiendo una sobrecarga de trabajo.

Para solucionarlo se ha planteado este proyecto desde la empresa, cuyo objetivo principal es *desarrollar un software para la generación automática de bancos de registros en lenguaje VHDL a partir de unas especificaciones.*

En el desarrollo de este proyecto se han llevado a cabo tres tareas principales. Por un lado, se ha definido un formato sencillo mediante el cual realizar las especificaciones de los bancos de registros en una hoja de cálculo de manera muy visual. Este formato permite detallar todas las características de los registros que la empresa requiere.

Por otro, se han diseñado los bancos de registros VHDL que deben generarse. Se ha tenido en cuenta la importancia de hacer un código claro y legible, aunque vaya a ser generado de manera automática. Para ello se han utilizado estrategias como definir ciertas especificaciones en un fichero de librería separado, o la creación de registros genéricos a partir de los cuales se realizan las instancias dentro de los bancos.

Finalmente, se ha desarrollado el programa de traducción que, tomando las especificaciones como entrada, genera automáticamente los ficheros VHDL que contienen la definición de los bancos de registros especificados, respetando siempre el formato y comportamiento descritos en la entrada.

Para el desarrollo del traductor se ha utilizado el lenguaje Java junto con JavaCC, un lenguaje específico para la generación de analizadores sintácticos. Esto ha facilitado el diseño y organización del programa, y ha permitido adaptar el análisis de la entrada de forma iterativa, conforme se concretaban aspectos de las especificaciones.

Automatic generation of register banks in VHDL with a translation tool in JavaCC

Abstract

The home appliances company Balay employs VHDL, a hardware description language, to carry out the design of their integrated circuits. In this process arises the necessity of frecuently modifying the register banks, which implies a work overload.

As a solution, the company has proposed this project, the principal objective of which is *to develop a software tool for the automatic generation of register banks in VHDL language based on particular specifications.*

In order to fulfill this project, three main tasks have been performed. On the one hand, we defined a simple format for the specification of a register bank in a spreadsheet to ease its presentation. This format allows us to define all the characteristics of the registers that the company requires.

On the other hand, we designed the VHDL register banks format, taking into account the importance of creating a clear and legible code, even though the latter is to be automatically generated. For this purpose, some strategies were used such as the definiton of a specification library in a separate file, and the creation of a generic register from which the instances are to be defined in the bank.

Finally, we developed the translation program that, using the specifications as the input, automatically generates the VHDL files which contains the definition of the register banks specified, with respect to the format and the behaviour defined in the input.

For the implementation of the translator, we used Java language alongside JavaCC, the latter being a specific language for the generation of parsers. This has eased the design and the organisation of the program, and has facilitated the adaptation of the analysis of the input, as some aspects of the specification changed.

Índice general

Índice de Figuras y Tablas

1. Introducción

1.1. Fundamentación

Este es un trabajo final de grado propuesto por parte de Balay, marca de la empresa BSH, a la Universidad de Zaragoza como fruto de su estrecha colaboración en diversas materias.

Dada la complejidad del diseño de circuitos integrados, así como el elevado coste de su fabricación final, es habitual el uso de un lenguaje de descripción de hardware para esta tarea, lo que permite realizar pruebas y verificar el correcto funcionamiento de un determinado diseño antes de llevarlo a producción. El lenguaje de descripción de hardware utilizado en Balay para este propósito es VHDL.

Durante el proceso de desarrollo de circuitos integrados en Balay, se ha hecho patente la necesidad de modificar a menudo los bancos de registros que comunican el micro y el resto de componentes como se ve en el esquema de la Figura 1. Realizar muchas modificaciones sobre el código supone una sobrecarga de trabajo.

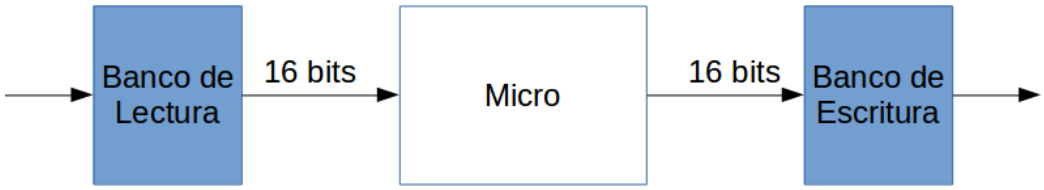

Figura 1: Conexiones entre el micro y los bancos de registros

El presente trabajo surge como respuesta a esta necesidad, con el fin de facilitar la tarea, automatizando la generación del código VHDL de especificación de dichos bancos de registros a partir de una sencilla especificación de los mismos en una hoja de cálculo.

1.2. Objetivos

El planteamiento inicial del trabajo tal y como la empresa lo propuso era muy general, con lo que quedaban abiertas diferentes decisiones, desde la estandarización de las especificaciones de entrada, hasta la propia estructura final de los bancos de registros en VHDL. Así mismo se dejó completa libertad sobre los lenguajes y tecnologías utilizados para el desarrollo de la propia herramienta de traducción.

Teniendo esto en cuenta, quedan bien definidas una serie de tareas a realizar para la consecución del objetivo general de este trabajo, quedando cada una de ellas definida como un objetivo específico del mismo.

El objetivo principal de este trabajo es *desarrollar un software para la generación automática de bancos de registros en lenguaje VHDL a partir de unas especificaciones*. Para ello, se llevarán a cabo los siguientes objetivos específicos:

- Definir las especificaciones de la entrada.
- Definir la estructura y funcionamiento de los bancos de registros VHDL.
- Diseñar e implementar la herramienta de traducción.

1.3. Desarrollo del proyecto

A la vista de estos objetivos específicos, se observan dos grandes bloques de los que se compone este proyecto. Por un lado, el trabajo referente a la definición de especificaciones, tanto de entrada como de salida, que se recoge en los dos primeros objetivos específicos. Y por otro, el diseño e implementación del software de traducción.

La definición de las especificaciones ha sido un trabajo llevado a cabo junto con los representantes de BSH, a través de una serie de reuniones. Partiendo de las primeras aproximaciones surgidas en las mismas, se realizaban pruebas que llevaban a nuevas propuestas, y estas a su vez se discutían en sucesivas reuniones.

De esta forma, las problemáticas encontradas al realizar las primeras aproximaciones de los formatos de entrada y salida han servido de retroalimentación para la definición de las especificaciones, dando lugar a numerosas cuestiones que en un principio no se habían contemplado.

Estos dos primeros objetivos constituyen a su vez un punto de partida necesario para el desarrollo de la herramienta de traducción, ya que su diseño depende ampliamente de las especificaciones de entrada y salida.

El proceso de desarrollo del software ha seguido un enfoque iterativo, en el que pueden destacarse al menos tres grandes iteraciones. Por un lado estarían el análisis léxico y sintáctico, por otro el análisis semántico y en tercer lugar la generación de código. Pero hay que tener en cuenta que estas iteraciones no se han producido de forma independiente, una tras otra.

El desarrollo del análisis léxico y sintáctico ha quedado solapado en el tiempo con la definición de las especificaciones, compartiendo muchos aspectos y realimentándose entre sí. A su vez, el análisis semántico y la generación de código quedan también solapadas y comparten, por ejemplo, la estructura de clases definida en su diseño.

1.4. Estructura de la memoria

El siguiente capítulo expone las ideas recogidas a partir de las reuniones y el formato final de las especificaciones de entrada y salida. Estas constituyen el punto de partida necesario para el desarrollo de la herramienta de traducción.

Posteriormente se abordarán las decisiones de diseño tomadas a partir de dichas especificaciones y de toda la información extraída de las reuniones. Este capítulo sirve para exponer y justificar algunas de las bases sobre las que se ha planteado el desarrollo del software.

En el capítulo sobre el programa de traducción se describe el funcionamiento de la propia herramienta, se utilizan ejemplos para mostrar su diseño interno y se recogen las verificaciones finales llevadas a cabo.

Por último se ha realizado un capítulo de conclusiones, donde se abordan la consecución de los objetivos, las incidencias y aprendizajes realizados, así como las posibilidades de continuación de este trabajo.

2. Especificaciones

2.1. Desarrollo de las reuniones y toma de decisiones

La idea planteada para este proyecto desde la empresa no recogía en un principio una descripción formal, ni del formato de entrada, ni de la propia descripción del código VHDL de salida. Esto dejaba un amplio espacio para plantear dichos formatos desde cero, pero lógicamente existían ciertos aspectos a tener en cuenta sobre la forma de trabajo ya utilizada anteriormente en la empresa.

En la primera reunión se pudieron establecer unos puntos de partida. Por ejemplo, la información referente a los bancos de registros se venía recogiendo en unas hojas de cálculo de forma poco estandarizada, pero con ciertas características comunes. Esto sirvió para observar cuáles podían ser algunas de las informaciones necesarias sobre cada registro, además de para plantear una primera aproximación sobre el formato que tendría la especificación de los mismos.

Por otro lado, se estableció la forma de comunicación desde y hacia el micro, a partir de la cual pudieron tomarse decisiones sobre determinadas líneas de entrada y salida que debían tener los bancos de registros, así como la necesidad de trabajar con diferentes fases y diferentes líneas de reloj.

No obstante, el planteamiento principal por parte de la empresa era el de generar y aplicar desde cero una nueva forma de trabajo, ya que anteriormente no se había seguido ningún tipo de estandarización. Así, fruto de las pruebas y discusiones, acabaron por generarse los modelos que se describen a continuación, que determinarán en adelante la forma de especificación de los registros por un lado, y la forma de comunicación y conexionado con los bancos VHDL por otro.

2.1.1. Principales decisiones tomadas y características del sistema

Antes de describir las especificaciones de entrada y salida como tales, se presentan de forma general algunas de las principales decisiones tomadas sobre el sistema, a partir de las cuales se han detallado los modelos posteriormente descritos.

Estas decisiones, extraídas de las discusiones llevadas a cabo en las reuniones, se describen aquí a modo de "Requisitos Funcionales"del sistema.

- El formato de entrada debe ser sencillo, estándar y fácilmente generable desde una hoja de calculo.
- Se generarán dos bancos de registros, uno de escritura y otro de lectura, siempre desde el punto de vista del micro.
- Cada registro del banco de escritura funciona con su propio reloj y fase. Este banco deberá contener tantas entradas de fase y reloj como diferentes fases y relojes aparezcan en las especificaciones.
- Al activarse la fase, reloj y dirección de un registro, su valor será almacenado de forma temporal. Un reloj y una línea de activación generales determinarán el momento en que todos los valores temporales deben volcarse, en bloque, a la salida.
- Queda garantizado por hardware, de forma externa al banco de escritura, que sus diferentes líneas de entrada, reset, fases y relojes, se activan de forma excluyente como convenga, sin producir colisiones.
- El banco de lectura es puramente combinacional.
- Queda garantizado por hardware, de forma externa al banco de lectura, que el micro respeta los tiempos necesarios en las transiciones de salida del mismo, leyendo así de forma síncrona valores estables.
- Los bancos, como forma de organización, se especificarán mediante bloques de registros, cada uno de los cuales pertenecerá al banco de lectura o al de escritura y podrá repetirse varias veces.
- Los registros se componen de diferentes campos con una posición fija dentro del registro, pudiendo quedar bits vacíos en el mismo.

- Estas agrupaciones de campos en un registro ocuparán la mínima cantidad posible de espacio de almacenamiento, sin desperdiciar bits, reproduciendo sus posiciones al recuperar el contenido de forma transparente.
- Si bien los registros ocuparán en el banco la cantidad de bits necesarios, la especificación de todos los registros será de 16 bits, en los que se incluirán, en su caso, los bits vacíos.
- Para facilitar el conexionado tanto en la salida del banco de escritura como en la entrada del banco de lectura, este se realizará a nivel de campo y mediante nombres, haciendo transparentes las posiciones de bits internas.
- Debe ser posible repetir el nombre de campos en diferentes registros, así como el nombre de registros en diferentes bloques y de bloques en diferente banco.
- En determinados registros se podrán especificar valores máximos y mínimos, con la opción de que unos registros trunquen y otros ignoren el valor en caso de ser superados dichos límites.
- Se podrá definir un valor de reset para cada campo de forma independiente.
- Para garantizar el correcto funcionamiento del reset y de los máximos y mínimos, debe ser posible especificar el tipo de dato (binario, signed o unsigned) para cada campo de forma independiente.
- La cantidad de registros de cada banco se tomará de las especificaciones. En ningún caso se llegará a tener una línea de direcciones de 32 bits, por restricciones de los tipos de datos, aunque lo normal es que no llegue a haber ni 200 registros, con lo que el margen es muy amplio.

2.2. Especificaciones de la entrada

El formato seleccionado para los ficheros de entrada es csv. En la empresa utilizan hojas de cálculo para facilitar la visibilidad de los registros existentes, así que se ha optado por utilizar un formato estándar y fácil de generar a partir de las hojas de cálculo. La separación de campos se realiza mediante comas.

En cuanto a la definición formal de la gramática (A.2) así como las expresiones regulares de los tokens utilizados (A.1) se incluyen como anexos al documento, junto con una explicación detallada de las mismas (A.3) para facilitar su comprensión. Aquí se va a realizar una explicación informal a partir de un ejemplo para clarificar el formato esperado.

Se han definido unas palabras clave para el inicio y fin de las especificaciones, de manera que pueden utilizarse cabeceras que no serán tenidas en cuenta por el traductor. En la Figura 2 puede verse el formato de la hoja de cálculo, con cabeceras, colores y celdas agrupadas, con lo que ofrece una buena visibilidad de los bloques y registros.

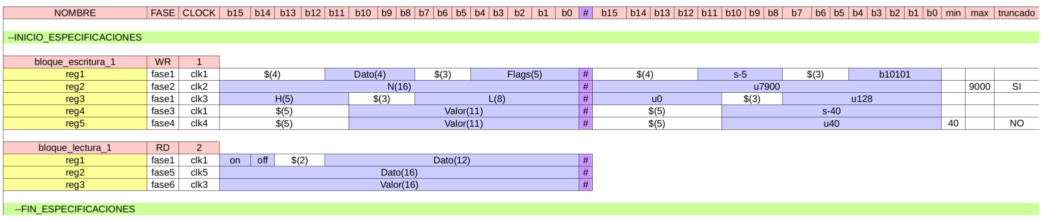

Figura 2: Ejemplo de especificaciones de entrada

Una vez guardado este mismo documento en formato csv, el resultado sería el siguiente. Se han obviado las cabeceras y únicamente se muestra la parte que corresponde a las especificaciones como tales.

```
--INICIO_ESPECIFICACIONES,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
bloque_escritura_1,WR,1,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
reg1,fase1,clk1,$(4),,,,Dato(4),,,,$(3),,,Flags(5),,,,,#,$(4),,,,s-5,,,,$(3),,,b10101,,,,,,,
reg2,fase2,clk2,N(16),,,,,,,,,,,,,,,,,#,u7900,,,,,,,,,,,,,,,,,,9000,SI
reg3,fase1,clk3,H(5),,,,,$(3),,,L(8),,,,,,,,#,u0,,,,,$(3),,,u128,,,,,,,,,,
reg4,fase3,clk1,$(5),,,,,Valor(11),,,,,,,,,,,#,$(5),,,,,s-40,,,,,,,,,,,,,
reg5,fase4,clk4,$(5),,,,,Valor(11),,,,,,,,,,,#,$(5),,,,,u40,,,,,,,,,,,40,,NO
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
bloque_lectura_1,RD,2,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
reg1,fase1,clk1,on,off,$(2),,Dato(12),,,,,,,,,,,,#,,,,,,,,,,,,,,,,,,,
reg2,fase5,clk5,Dato(16),,,,,,,,,,,,,,,,,#,,,,,,,,,,,,,,,,,,,
reg3,fase6,clk3,Valor(16),,,,,,,,,,,,,,,,,#,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
--FIN_ESPECIFICACIONES,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
```

Tomando el ejemplo anterior, pueden verse algunas de las principales características del formato de entrada. Los registros se especifican agrupados por bloques, que deben ir separados entre sí con al menos un salto de línea. Y la primera línea de cada grupo es la especificación del bloque en sí, donde dice su nombre, si pertenece al banco de lectura o escritura, y la cantidad de veces que debe repetirse.

Para definir cada registro se escribe su nombre, fase y reloj y, a continuación, se detallan los campos que lo componen, marcando su tamaño entre paréntesis. Puesto que los registros pueden contener espacios vacíos, estos se marcan con el símbolo "$" seguido del tamaño entre paréntesis.

Los registros del banco de escritura tienen además la posibilidad de establecer valores de reset, valores máximos y mínimos del registro, y la opción de truncar o no el valor en caso de recibir una entrada que se salga de el rango definido. Para definir los valores de reset se utilizan tres posibles tipos de datos, signed, unsigned y máscaras de bits, para lo cual el valor viene precedido del caracter 's', 'u' o 'b' respectivamente.

Debe tenerse en cuenta que los registros que tengan valor máximo o mínimo deben estar compuestos por un único campo, y que el flag de truncado únicamente tiene algún efecto en caso de que se haya definido al menos un valor máximo o mínimo.

2.3. Especificaciones de la salida

En el caso de la salida, se trata de especificar a alto nivel las características de ambos bancos, el de escritura y el de lectura, describiendo las líneas de entrada y salida que tendrán y cuál será su funcionamiento interno. En la Figura 3 puede verse un esquema de los mismos.

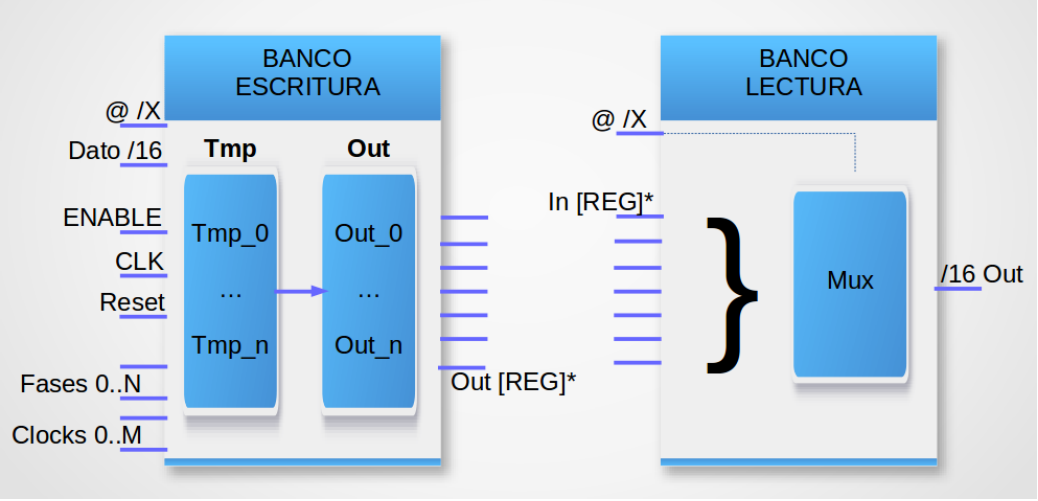

Figura 3: Esquema de los bancos de registros

A continuación se describirá su funcionamiento y las diferentes líneas de

entrada y salida, dando especial importancia de las estructuras definidas para la salida del banco de escritura y la entrada del de lectura.

2.3.1. El banco de escritura

Como puede verse, el banco de escritura está compuesto internamente por dos bloques, uno temporal y otro de salida, de forma que la modificación del valor de cada registro de forma individual se realiza sobre el temporal, mientras que al de salida se vuelcan todos los valores temporales en bloque al activar la señal de enable global.

Las líneas de entrada del banco de escritura se corresponden con las que pueden observarse en el lateral izquierdo del mismo, en la Figura 3. A continuación se describen, de arriba a abajo, especificando tanto sus características como su funcionalidad:

- La línea de direcciones servirá para seleccionar los registros internos del banco. Su tamaño se definirá atendiendo a la cantidad de registros que aparezcan en las especificaciones de entrada, que en ningún caso podrán ser más de $2^{30} - 1$.
- La línea de entrada de datos siempre tendrá 16 bits. Cuando un registro tenga espacios vacíos entre sus diferentes campos, el banco de registros se encargará de guardar únicamente las posiciones válidas, es decir, las correspondientes a los campos.
- La línea de enable global es un bit que activa el volcado de los valores temporales a la salida.
- La línea de reloj global es un bit que controla el volcado de los valores temporales a la salida de forma síncrona en su flanco de subida.
- La línea de reset global es un bit que reinicia, de forma asíncrona y con prioridad sobre el resto de señales de control, los valores de todos los registros del banco, devolviéndolos a su valor inicial y volcándolos así mismo directamente a la salida. Es una línea activa en cero.
- Las siguientes N líneas son las líneas de fase particulares de los registros. Habrá tantas como diferentes nombres de fase aparezcan en las especificaciones de entrada y cada registro estará asociado a la que le corresponda. Su función es activar el volcado del valor de entrada sobre el registro correspondiente.

- Las siguientes M líneas son las líneas de reloj particulares de los registros. Habrá tantas como diferentes nombres de reloj aparezcan en las especificaciones de entrada y cada registro estará asociado a la que le corresponda. Su función es controlar el volcado del valor de entrada sobre el registro correspondiente de forma síncrona en su flanco de subida.

En la Figura 4 se puede observar un ejemplo en el que se modifica el valor de dos registros en el bloque temporal, sin que esto modifique la salida y, posteriormente, se vuelcan estos valores temporales a la salida del banco.

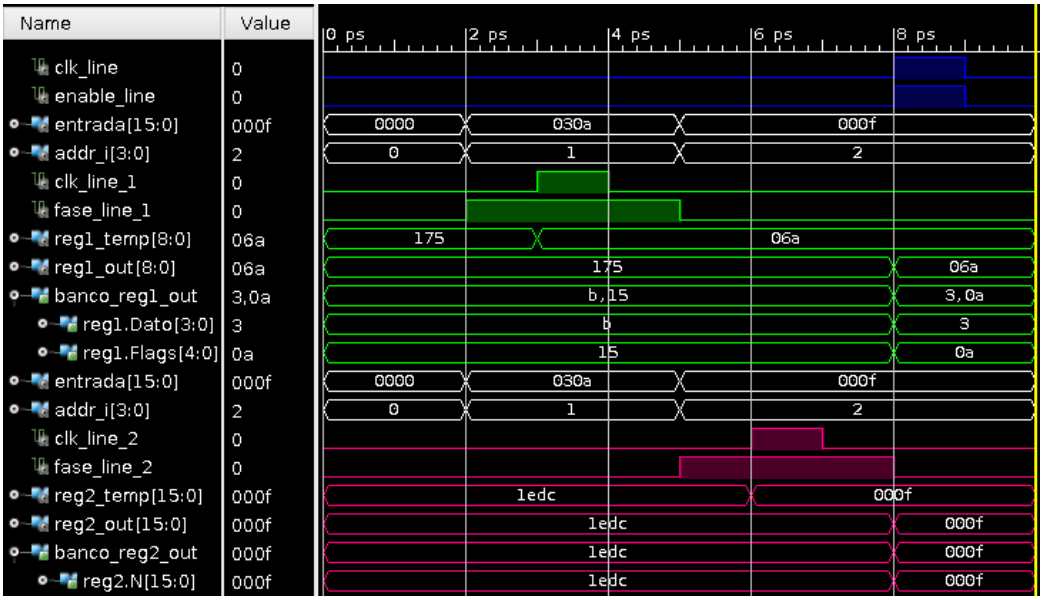

Figura 4: Diagrama temporal del funcionamiento del banco de escritura

En un primer momento se selecciona el valor 1 en la línea de direcciones "addr_i" y se activa la fase "fase_line_1" correspondiente al registro "reg1", de color verde en la imagen. Al activarse el flanco de subida de su reloj, "clk_line_1", el valor de la entrada se vuelca en el valor temporal "reg1_temp", pero no sobre la salida del banco. Seguidamente se siguen los mismos pasos con las señales correspondientes al registro "reg2", en color magenta.

Por último se activan la línea de enable y reloj globales "enable_line" y "clk_line", de color azul en la imagen. Es en ese momento cuando los valores temporales de los registros se vuelcan a la salida, con lo que se puede observar que se modifican simultáneamente "reg1_out" y "reg2_out", tomando sus respectivos valores temporales.

Hay un detalle que no debería pasarse por alto. El registro “reg1” está formado por los campos “Dato”, de 4 bits, y “Flags”, de 5 bits, con lo que el valor realmente guardado no se corresponde con la entrada. La salida “reg1_out” se corresponde con el valor de los 9 bits devueltos por el registro. Sin embargo al interpretarla como campos en la salida real del banco, “banco_reg1_out”, toma cada uno su valor correspondiente. La Figura 5 muestra un esquema de lo ocurrido.

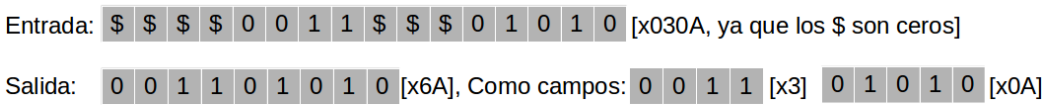

Figura 5: Ejemplo de adaptación de la entrada a los campos de un registro

Mientras la línea de enable global esté activa no debería activarse la fase particular de ninguno de los registros, para evitar la posibilidad de tomar valores inconsistentes. De esto se encarga el hardware de control del micro, de forma independiente al banco de escritura.

Para la línea de salida se ha definido una estructura de datos con la intención de facilitar el conexionado con el resto de componentes, que se detallará más adelante en este mismo capítulo.

2.3.2. El banco de lectura

El banco de lectura es puramente combinacional. Se trata pues de un multiplexor cuya línea de direcciones hace de señal de control para seleccionar de la entrada el registro correspondiente y volcarlo en la salida.

El formato de la línea de entrada se corresponde con la misma estructura de datos utilizada en la salida del banco de escritura, que se describirá en el punto siguiente. No obstante, cabe destacar aquí el hecho de que dicha estructura está direccionada a nivel de campo dentro de cada registro.

Como ya se ha visto, los campos de cada registro no tienen por qué ser consecutivos, quedando en ocasiones espacios inutilizados. El multiplexor del banco de lectura debe encargarse de reconstruir el formato completo de cada registro, introduciendo ceros en las posiciones vacías de los mismos. Su salida tendrá siempre 16 bits.

En el momento en que se introduce un nuevo valor en la línea de direcciones, debe garantizarse que el valor de las entradas correspondientes al registro

seleccionado sea estable. Así mismo, tras haber seleccionado el registro correspondiente, debe esperarse un tiempo de delay antes de leer la salida del banco. De estas sincronizaciones se encarga el hardware de control del micro, de forma independiente al banco de lectura que, como se ha dicho, es combinacional.

2.3.3. Conexionado con el resto de componentes

Tanto la salida del banco de escritura como la entrada del de lectura tendrán un formato jerarquizado para permitir el acceso a nivel de campo y consistirán en una estructura compleja compuesta de tantas líneas como campos contenga el banco, agrupadas entre sí.

Para referenciar un campo determinado será necesario explicitar su nombre, el registro al que pertenece y el bloque al que pertenece el registro. Esto permite que en diferentes registros se repitan los nombres de campos e, igualmente, que en diferentes bloques se repitan los nombres de registros. Por ejemplo, podríamos tener el "campo1" del "registro1" del "bloque1", y el "campo1" del "registro1" del "bloque2".

En cuanto a la entrada del banco de escritura y la salida del de lectura tienen 16 bits, donde el micro lee o escribe los valores correspondientes a la estructura completa de registro de 16 bits que él maneja.

En las Figuras 6 y 7 se puede observar la correspondencia entre las respectivas entradas y salidas de ambos bancos con un ejemplo genérico de registro formado por dos campos y espacios vacíos.

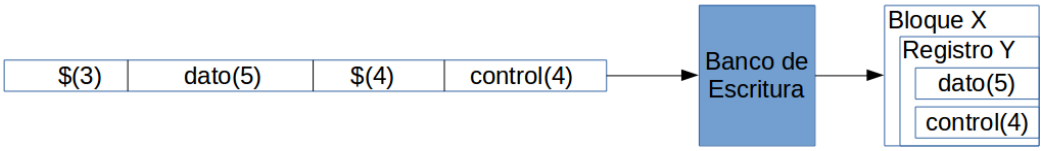

Figura 6: Correspondencia de E/S en el banco de escritura

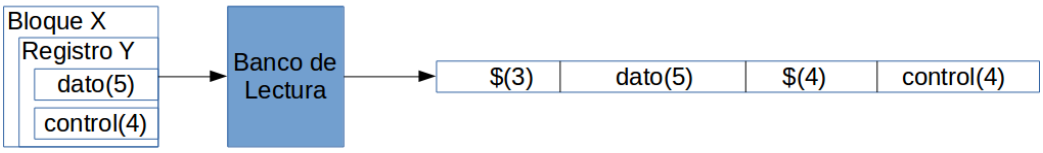

Figura 7: Correspondencia de E/S en el banco de lectura

Debe tenerse en cuenta que esto no significa que dichas líneas de ambos bancos estén conectadas entre sí. Estas estructuras sirven para realizar el conexionado con el resto de componentes de forma transparente a las posiciones de los campos a nivel de bit dentro de los registros.

3. Decisiones de Diseño

3.1. Decisiones tomadas a partir de las especificaciones

Durante el propio proceso de definición de las especificaciones, y a través de las pequeñas pruebas y aproximaciones al problema, iban apareciendo cuestiones directamente relacionadas con la implementación del traductor en sí.

De esta forma fueron tomando cuerpo otras decisiones independientes de las especificaciones, algunas totalmente internas del traductor, sobre las que la empresa dejó total libertad, y otras relacionadas con el código final en VHDL que se plantearon también en las reuniones para definir detalles sobre la forma del código, más allá de la propia funcionalidad de los bancos.

3.1.1. El generador de analizadores sintácticos JavaCC

Una de las principales decisiones tomadas fue el hecho de utilizar un generador de analizadores sintácticos. En este sentido, estar cursando en aquel momento la asignatura de "Procesadores de Lenguajes" como optativa supuso una gran ventaja para poder tomar esta decisión, ya que la propuesta desde la empresa era utilizar un lenguaje de propósito general, como Python o C.

Se ha elegido JavaCC porque ofrece gran funcionalidad para un diseño como este, en el que las especificaciones de entrada están siendo definidas mediante una gramática formal.

JavaCC es básicamente un metacompilador que genera automáticamente un analizador léxico y sintáctico escrito en java, a partir de la definición de los tokens, en forma de expresiones regulares, y de la gramática, en un formato

específico mediante llamadas a funciones sucesivas.

La posibilidad de incluir código java en cualquier punto y acceder como variables a los tokens leídos permite realizar de forma sencilla la comunicación con el analizador semántico y el generador de código, creados directamente como otras clases en java que se comunicarán con las generadas desde JavaCC.

La principal ventaja de utilizar un generador de analizadores sintácticos es la sencillez de trasladar las especificaciones de entrada a un código capaz de analizarla. Pero esto supone además una gran utilidad a la hora de lidiar con especificaciones que están en proceso de construcción, ya que facilita enormemente las modificaciones posteriores.

El análisis es dirigido por el analizador sintáctico, que va pidiendo tokens al léxico. En caso de no haber error, en ese momento el token queda disponible para ser utilizado como se desee desde otras clases como el analizador semántico o el generador de código. En la Figura 8 puede observarse un esquema de su funcionamiento.

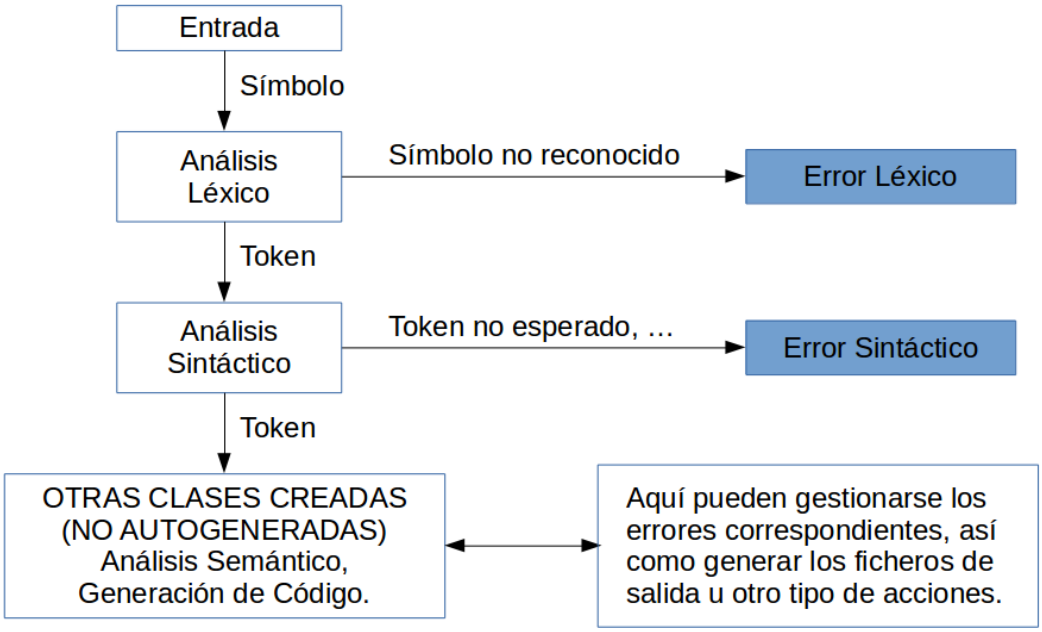

Figura 8: Esquema de funcionamiento de JavaCC

3.1.2. Implementación VHDL de los bancos de registros

Una de las preocupaciones que surgieron durante las reuniones a cerca del resultado en VHDL es el hecho de que un código generado automáticamente tiende a ser enrevesado o poco legible, así que se ha procurado dar mucha importancia a la organización del mismo.

Para facilitar la generación de forma lo más organizada posible, se tomaron dos decisiones sobre el diseño final. Por un lado, el hecho de generar una librería de especificaciones separada de los propios bancos, donde definir macros y tipos de datos. Por otro lado, hacer uso de la cláusula "generic" que permite parametrizar la generación de "entidades".

Las "entidades" son módulos de vhdl a partir de los que declarar instancias concretas de un componente del sistema, tal y como ocurre con las clases de un lenguaje orientado a objetos. Cada una de ellas se comunica con el resto mediante líneas de entrada y salida.

La cláusula "generic" permite que una "entidad" reciba parámetros a partir de los cuales puede definir su comportamiento, e incluso el tamaño de las entradas y salidas. Esto permite generar un "Registro Genérico" a partir del cual se realizarán instancias con parámetros diferentes para la creación de todos los tipos de registros.

3.1.2.1. Macros y tipos de datos

Tanto el tamaño de los registros, que siempre es 16, como el tamaño de la línea de direcciones, que se calcula a partir de la cantidad de registros de la entrada, se definen como macros. Así mismo se definen macros para referenciar las direcciones de cada registro.

Para ello se tomó la decisión, junto con los responsables de la empresa, de asignar las direcciones de forma consecutiva en el orden de aparición en el fichero de entrada. De esta forma se facilita la asociación a la hora de hacer pruebas sobre los bancos.

La definición de estas macros facilita la legibilidad del código, pero además simplifica la propia generación. Por ejemplo la dirección de cada registro se nombra añadiendo "addr_" delante del nombre del registro, de forma que resulta muy sencillo utilizarla durante la generación de código.

En cuanto los tipos de datos de salida del banco de escritura y entrada del

de lectura, se definen a partir de estructuras de tipo "record", lo que serían "struct" en C, anidando unas sobre otras. Esto permite tener un único dato como entrada o salida del banco correspondiente, que es instancia de un tipo de dato complejo definido en el fichero externo. Sin embargo VHDL no permite anidar estructuras "record" directamente, sino que hay que definirlas sucesivamente como tipos de datos separados.

Por tanto se define un "record" para cada registro, cuyos campos son de tipo "std_logic_vector", es decir, un vector de bits. Posteriormente, cada bloque se define igualmente como un "record" y sus campos, que se corresponden con registros, toman los tipos anteriormente creados. Por último, se crea otro "record" para cada uno de los dos bancos y se repite lo anterior, sus campos, que representan bloques, toman el tipo correspondiente.

Para el nombrado de tipos ha habido que tener en cuenta que los nombres de los registros se pueden repetir en diferentes bancos, con lo que se debe utilizar tanto el nombre de banco como el de registro, evitando así repeticiones.

3.1.2.2. El registro genérico

Crear un único "Registro Genérico" facilita enormemente la generación de código, pero especialmente la legibilidad del fichero del banco de escritura. Este contiene únicamente las instancias correspondientes de cada registro. Al instanciar un registro se le pasan los siguientes parámetros, tomados de las especificaciones, que definen sus características y su comportamiento:

- Tamaño del registro, de tipo "natural". Aunque el tamaño completo de cada registro "lógico" siempre es 16 bits, estos tienen bits inutilizados que se evita guardar físicamente. Así pues se le pasa la cantidad de bits útiles. El valor por defecto es 16.
- Dirección, de tipo "natural". Para este valor se utiliza la macro anteriormente definida en el "package" de definiciones. Es el único valor obligatorio para todos los registros.
- Mínimo, de tipo "integer". En caso de que se defina un mínimo valor posible del registro. Por defecto no tiene ningún efecto.
- Máximo, de tipo "integer". En caso de que se defina un máximo valor posible del registro. Por defecto no tiene ningún efecto.
- Truncado, de tipo "boolean". Marca si el comportamiento frente a valores fuera de rango debe ser el de truncar, y solo tiene efecto si hay valor

mínimo o máximo. El valor por defecto es false, que se corresponde con ignorar el valor.

- Valor de reset, de tipo "integer". Define el valor de reset del registro. El valor por defecto es 0.
- Flag de "signed", de tipo "boolean". Marca si el tipo del registro es "signed", y solo tiene efecto si hay valor mínimo o máximo. El valor por defecto es false.

En la implementación del registro genérico como tal se ha utilizado la cláusula de precompilación "generate", de tal forma que, una vez tomados los valores de los parámetros, únicamente se genera y compila la parte de código correspondiente a su comportamiento, con lo que se evita la posibilidad de generar hardware inútil.

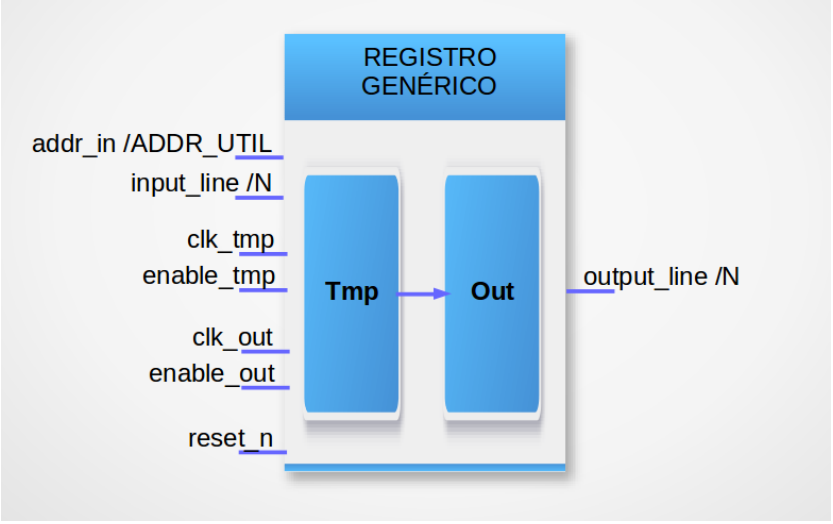

Figura 9: Esquema del registro genérico

Así, cada registro individual responde al esquema de la Figura 9. Sin embargo el tamaño "N" de la línea de entrada y salida es diferente, e internamente varían comportamientos de unos a otros registros.

Vamos a ver este comportamiento comparando dos ejemplos de declaración de la cláusula "generic" en dos registros diferentes. El primero con tamaño por defecto, valores máximo y mínimo y sin truncado. El segundo con un tamaño de 5 bits, sin restricciones de máximo pero sí de mínimo y con el flag de truncado activo.

Ejemplo1:

```
generic map(
    addr => 23,
    min => 200,
    max => 9000,
    reset_value => 7900)
```

Ejemplo2:

```
generic map(
    n => 5,
    addr => 14,
    min => -5,
    truncar => true,
    reset_value => 3,
    es_signed => true)
```

El registro con la cláusula "generic" del ejemplo1 tendrá líneas de entrada y salida de datos de 16 bits, que es el valor por defecto, y un comportamiento que responderá al diagrama de flujo de la Figura 10. Debe tenerse en cuenta que en cada registro hay dos procesos paralelos, uno que controla el valor temporal y otro la línea de salida.

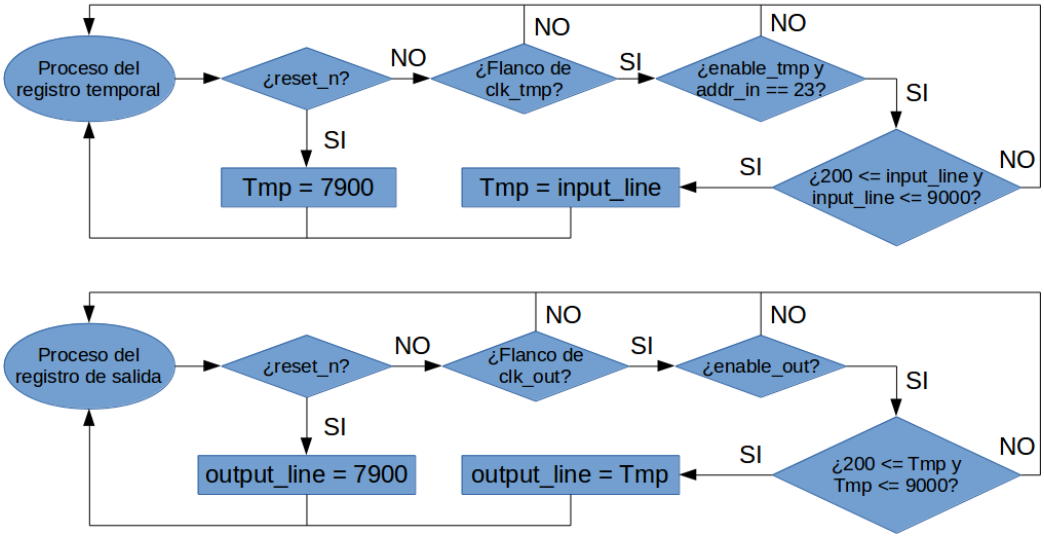

Figura 10: Diagrama de flujo del registro de ejemplo 1

Sin embargo, la declaración de la cláusula "generic" del ejemplo 2, generará un registro con líneas de entrada y salida de datos de 5 bits y un comportamiento diferente de sus procesos, que responderá al diagrama de flujo de la Figura 11.

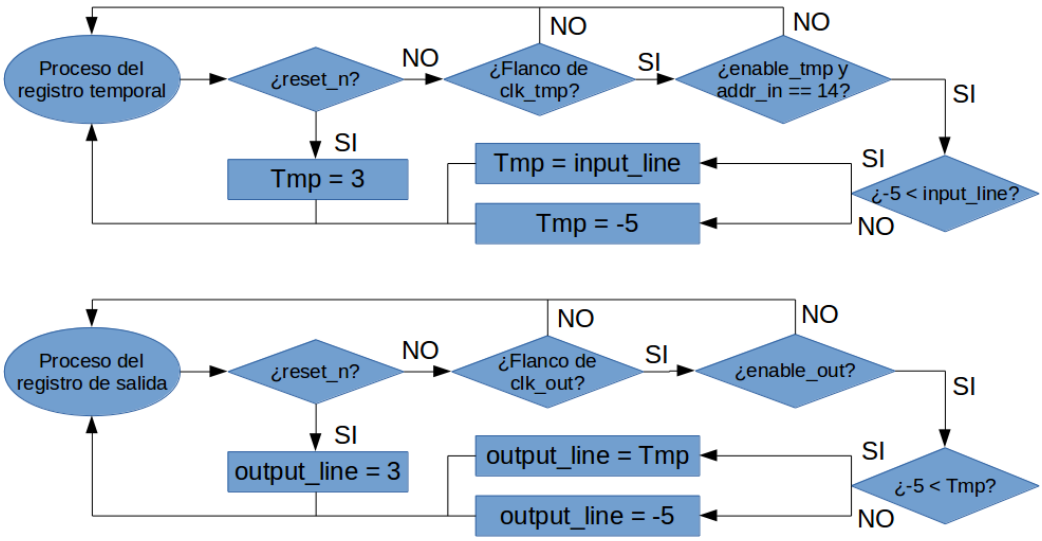

Figura 11: Diagrama de flujo del registro de ejemplo 2

3.2. Otras decisiones sobre la implementación del traductor

Una vez tomada la decisión de utilizar JavaCC quedan algunas cuestiones que plantearse sobre la implementación. Destaca por un lado el enfoque desde el que plantear las clases a utilizar, adaptando el modelo genérico de un compilador al problema de traducción en cuestión. Por otro, se plantean aquí cuestiones sobre la organización y claridad del código, así como la importancia dada a la documentación.

3.2.1. Las clases utilizadas

Posteriormente se hará una descripción detallada de la estructura de clases planteada en las fases de análisis y diseño. En este punto se pretende poner de manifiesto ciertas reflexiones previas, de carácter más general, para contextualizar el uso de una herramienta como JavaCC fuera del ámbito de la creación de compiladores, en el cual el tipo de estructuras a utilizar está más estandarizado.

Generalmente el análisis semántico viene apoyado por una estructura de datos, la "Tabla de Símbolos", done se va recogiendo la información necesaria sobre los tokens recibidos. En el caso de un compilador, por ejemplo,

podríamos guardar información sobre si un símbolo es un procedimiento o una variable, o de qué tipo es. Si se trata además de un lenguaje de bloques, guardaríamos información sobre el ámbito de actuación de cada símbolo.

En cuanto a la generación de código, en ocasiones puede ir realizándose conforme se lee la entrada, pero también es habitual el uso de estructuras de datos, en este caso en forma de árbol, que permiten tener bien organizadas las expresiones a escribir y así poder volcarlas sobre el fichero en el orden y momento deseado.

Pues bien, teniendo en cuenta la utilidad de estas estructuras de datos, y la propia estructura de los bancos de registros, formados por bancos, que a su vez se componen de registros, que están divididos en campos; se ha decidido utilizar una estructura en forma de árbol para representar la información recogida, como la representada en la Figura 12.

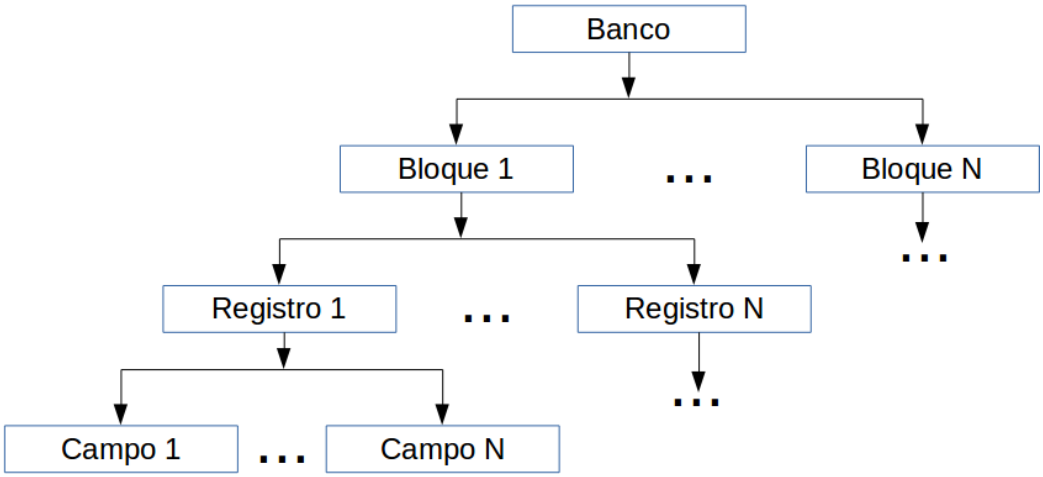

Figura 12: Diagrama del árbol de clases utilizado

Además, en este caso no existen diferentes elementos, como funciones, variables o constantes, ni tampoco diferentes ámbitos de actuación. La relación entre bancos, bloques, registros y campos es una relación de composición total, es decir, cada uno se compone a partir del siguiente, y de nada más. Aunque para asumir esto último, hay que tomar los espacios vacíos de los registros como "campos vacíos".

Esto garantiza por ejemplo que algunas propiedades, como el tamaño, pueden deducirse partiendo de la base del árbol hacia arriba. También ocurre que entre ramas hermanas comparten estructura, lo que facilita otros aspectos como por ejemplo la forma de garantizar que entre bloques no se repitan

nombres, ni entre registros dentro de un bloque, pero estos últimos sí puedan repetirse con registros de otro bloque.

En general lo que ofrece esta estructura es una perfecta compartimentación, de forma que cada nivel del árbol se pueda ocupar de lo que le concierne, siendo completamente transparente a lo que concierne a otros niveles.

Esta propiedad es perfecta para el análisis semántico, permitiendo analizar los errores en su contexto y propagarlos posteriormente hacia la cima del árbol. Pero además, con un poco de imaginación y una buena organización de las estructuras en VHDL, puede resultar tremendamente útil también para estructurar la generación de código.

3.2.2. Organización, claridad y documentación

La última de las cuestiones a tratar en este capítulo se refiere a la organización del propio código en Java. Desde el inicio del trabajo se ha dado gran importancia a una clara organización del código, que facilite su comprensión, una completa documentación así como un código convenientemente comentado.

La buena organización del código ha resultado ser otra de las ventajas de la estructura de clases planteada, por favorecer la compartimentación de las diferentes funcionalidades de cada parte. Esto supone que aquellas cuestiones que quedan fuera del ámbito de actuación de una de las clases resultan ser completamente transparentes a la misma.

Por supuesto este planteamiento tiene también sus desventajas, ya que genera cierta sobrecarga de trabajo por las sucesivas llamadas a funciones entre las clases. Además en algunos sentidos el sistema parece sobredimensionado, llevando a cabo varios pasos sucesivos para acabar realizando una pequeña acción.

Pero esto responde a la decisión de dar prioridad absoluta a un código legible y un sistema fácilmente ampliable, no en cuanto al tamaño de los bancos, sino en cuanto a la posibilidad de añadir funcionalidades en un futuro. Desde la empresa se ha insistido en que la entrada rara vez sobrepasará los doscientos o trescientos registros, con lo que los problemas de escalabilidad no van en ese sentido.

Por otro lado, en lo que respecta a los comentarios y la documentación, se ha decidido utilizar “javadocs”. Esta herramienta genera automáticamente una página web al estilo de la documentación oficial de Java a partir de

interpretar una serie de comentarios con un formato específico que deben integrarse en el código.

Así pues, todo el código está comentado con el formato de "javadocs", con los enlaces pertinentes entre unas funciones y otras, y unido además a la documentación oficial de Java allí donde aparecen sus tipos de datos o funciones de librerías.

4. El programa de traducción

4.1. Funcionamiento y estructura del sistema

Es momento de retomar todo lo tratado hasta ahora para describir, de forma global, la organización y funcionamiento de la herramienta de traducción. Algunas de sus partes vienen definidas por la propia estructura de JavaCC. Otras, como la estructura de clases utilizada, han sido planteadas para garantizar la modularidad y facilitar el análisis semántico y la generación de código.

Recordemos que los analizadores léxico y sintáctico son automáticamente generados por JavaCC. Como se muestra en la Figura 13, una vez recibido el fichero de entrada con las especificaciones, es el analizador sintáctico el encargado de dirigir el funcionamiento, solicitando tokens al léxico y esperando que coincidan con la estructura gramatical. Este último, por su parte, se encarga de leer la entrada símbolo a símbolo hasta completar el siguiente token válido para entregarle.

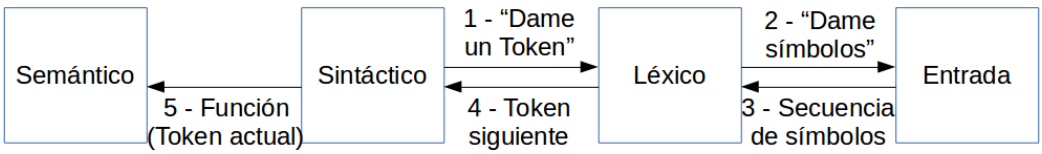

Figura 13: Esquema del funcionamiento de la generación de tokens

Según el punto de la gramática en el que nos encontremos, el analizador sintáctico llama a una u otra función del semántico, pasándole el token correspondiente. Así, cada función del analizador semántico realiza las acciones correspondientes en comunicación con los bancos, que se corresponden con la raíz del árbol de clases definido.

Tal y como se ha descrito, esto inicia las comunicaciones entre las clases del

árbol, es decir, bancos, bloques registros y campos. Un ejemplo bastante completo puede ser la introducción del valor de reset de un campo. Planteemos que dentro de la especificación de un bloque del banco de escritura se acaba de leer el valor de reset de un campo.

Ya se ha creado en el banco de escritura el bloque correspondiente y en este el registro y cada uno de sus campos. Ahora el analizador sintáctico llama a la función necesaria para añadir el valor de reset del campo, llamémosla símplemente "reset()", y se va propagando por el árbol. Tras llegar al campo, se devuelve el estado de salida hasta el analizador semántico. En la Figura 14 se esquematizan las sucesivas comunicaciones.

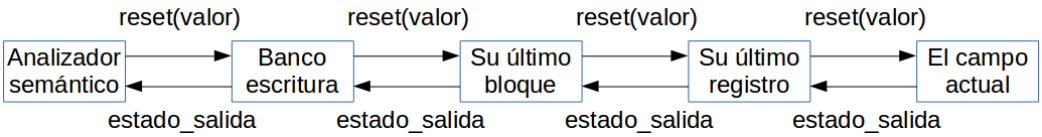

Figura 14: Esquema del funcionamiento del análisis semántico

Una vez leída la entrada entera, la estructura de clases está completamente formada y cada elemento contiene toda la información necesaria para generar el código en VHDL. En ese momento, si no se ha producido ningún error léxico ni sintáctico, el analizador sintáctico avisa del fin de la entrada al semántico. Este a su vez, si no se ha producido ningún error semántico, inicia el funcionamiento del generador de código.

El generador de código se encarga de crear los ficheros de salida y maneja la escritura en los mismos. Además genera la estructura general de cada uno de los ficheros, pero para ir rellenando cada parte específica, se comunica con los bancos de la misma manera que hacía el analizador semántico, pero a través de otras funciones específicas para la generación de código.

Tomemos por ejemplo una función "direccion()" para generar las macros de la dirección interna de cada registro. El esquema de comunicación de la Figura 15 es muy similar al del análisis semántico, pero ahora el valor devuelto es una variable de tipo String a partir del cual se va conformando la salida.

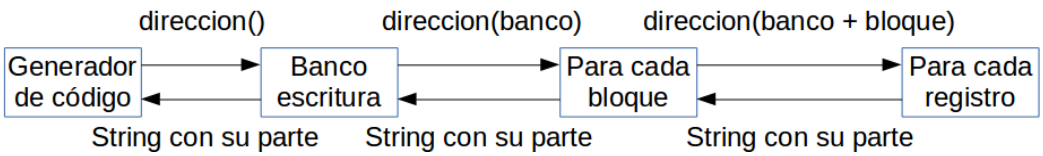

Figura 15: Esquema del funcionamiento de la generación de código

Algo que se observa en este ejemplo es que, en ocasiones, la comunicación de datos para generar el código también puede ir hacia abajo. El registro es el único que conoce su propio nombre, mientras que por el camino se le pasan el nombre del banco y del bloque.

Debe tenerse en cuenta que cada nodo del árbol recibe información de todas sus ramas, ver que en el ejemplo dice "para cada bloque/registro". Así, cada parte concatena de la forma necesaria lo recibido de todos sus componentes y, finalmente, el generador de código se encarga de colocar el resultado en el lugar correspondiente y volcarlo en los ficheros.

4.2. Diseño del sistema

Se ha procurado en todo momento realizar las explicaciones lo más claras posible, evitando los aspectos más formales. En este apartado se recopilan algunos diagramas representativos del código, expuestos de forma exhaustiva pero evitando entrar en un alto nivel de detalle. El diagrama de clases de la Figura 16 muestra las relaciones y cardinalidades entre las clases generadas.

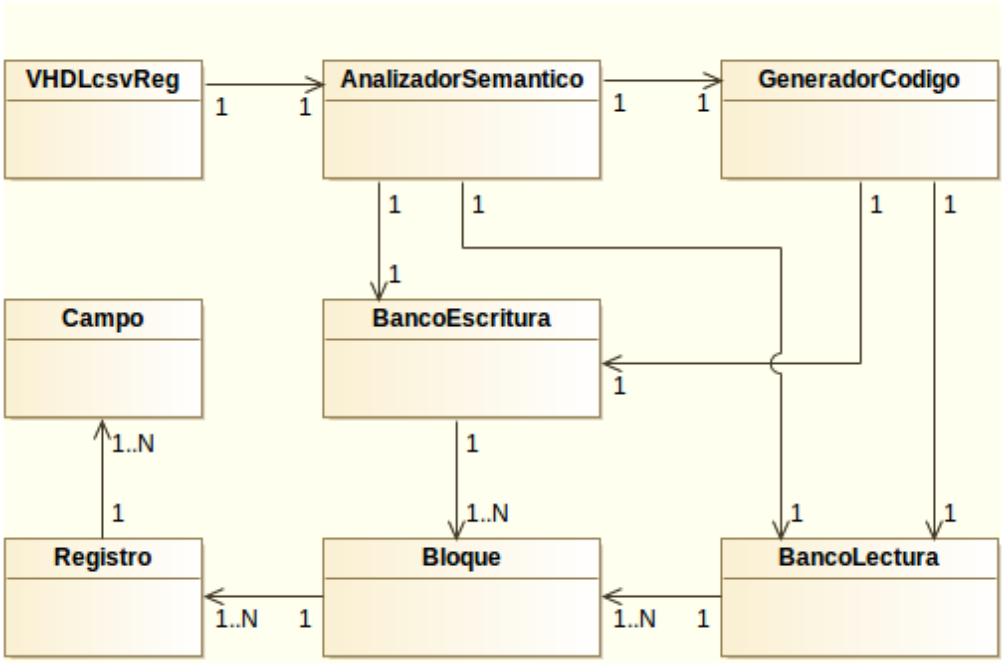

Figura 16: Diagrama de Clases

A continuación se exponen una serie de diagramas de secuencia que describen momentos puntuales del comportamiento del código a modo de ejemplo. Se han seleccionado únicamente algunos considerados suficientemente representativos para describir, de forma más detallada y precisa, el funcionamiento real de la aplicación.

La Figura 17 se corresponde con la secuencia de inicialización, en la que unas clases generan a otras y se prepara el sistema para comenzar el análisis de la entrada. Nótese que la clase "VHDLcsvReg" es la clase principal del paquete de análisis léxico y sintáctico automáticamente generado por JavaCC y encaja con lo que en esquemas anteriores se ha denominado el "analizador sintáctico".

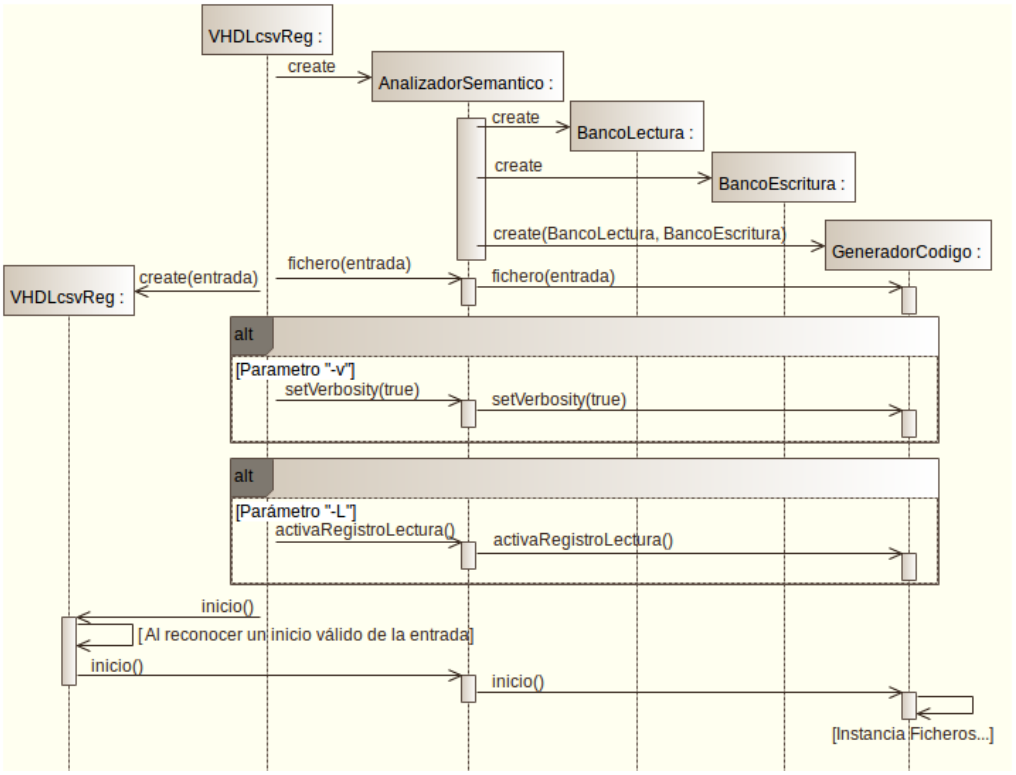

Figura 17: Diagrama de Secuencia: secuencia de inicialización

En el punto anterior se ha descrito, como ejemplo del análisis semántico, la introducción del valor de reset en un campo. En el caso de la Figura 18 se plantea como ejemplo la generación de un nuevo campo dentro de un registro, para mostrar, con las clases y funciones realmente implementadas, las acciones llevadas a cabo.

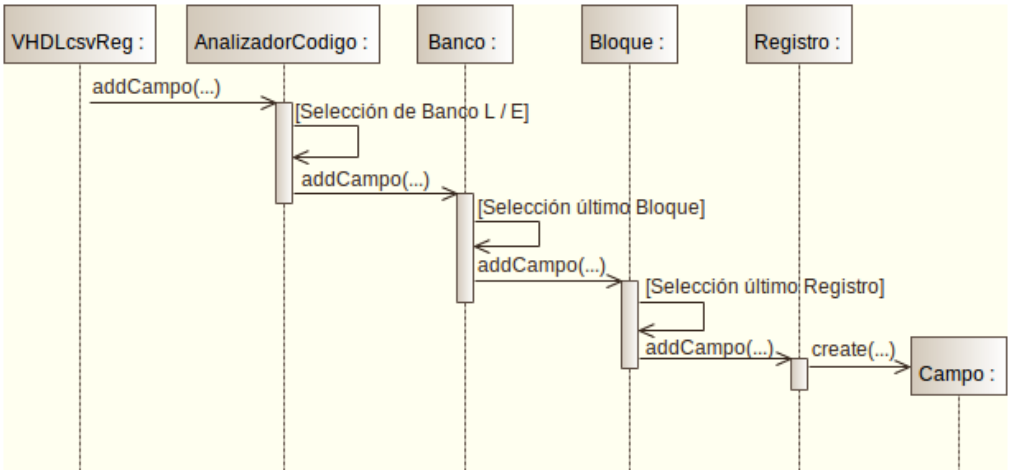

Figura 18: Diagrama de Secuencia: añadir un campo

Otro de los momentos claves descritos de forma genérica en el apartado anterior es el final del análisis e inicio de la generación de código. Su diagrama de secuencia se recoge en la Figura 19, incluyendo la generación del fichero de especificaciones, pero sin entrar en las sucesivas llamadas a las clases del árbol. La función "_generaFicheros()" continúa repitiendo un esquema similar para cada uno de los fichero a generar.

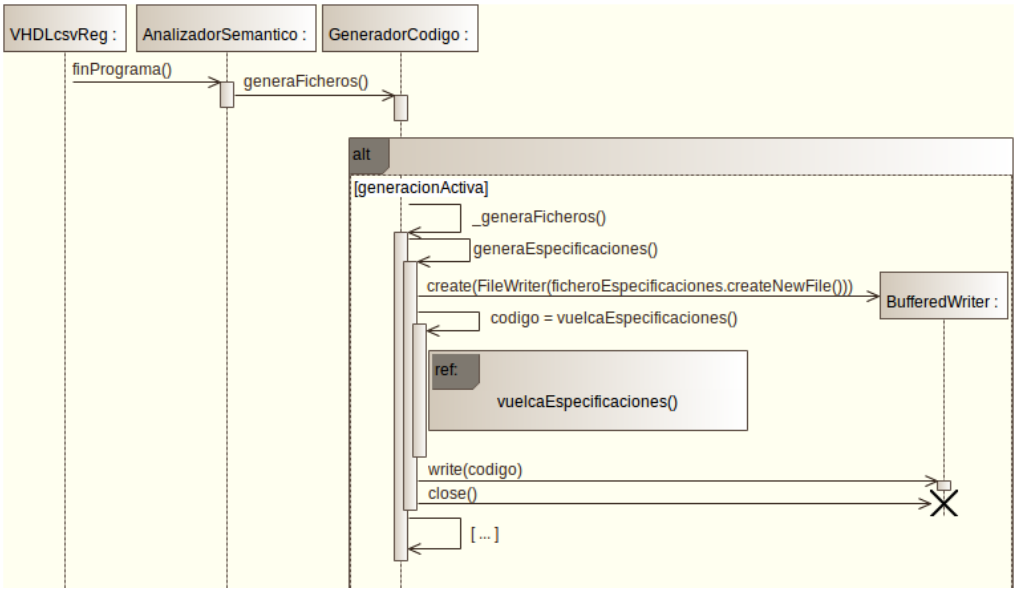

Figura 19: Diagrama de Secuencia: inicio de la generación de código

Por último, el diagrama de la Figura 20 muestra el mismo ejemplo utilizado anteriormente para describir la generación de código. Se trata de especificar las macros que corresponden a las direcciones de cada registro. Se utiliza este ejemplo porque es uno de los más claros en la generación de código.

De hecho se trata de una parte de la función "vuelcaEspecificaciones()" que se ha dejado sin especificar en el diagrama anterior de la Figura 19. Se plantea aquí como un pequeño ejemplo de las sucesivas llamadas a funciones realizadas en el árbol de clases.

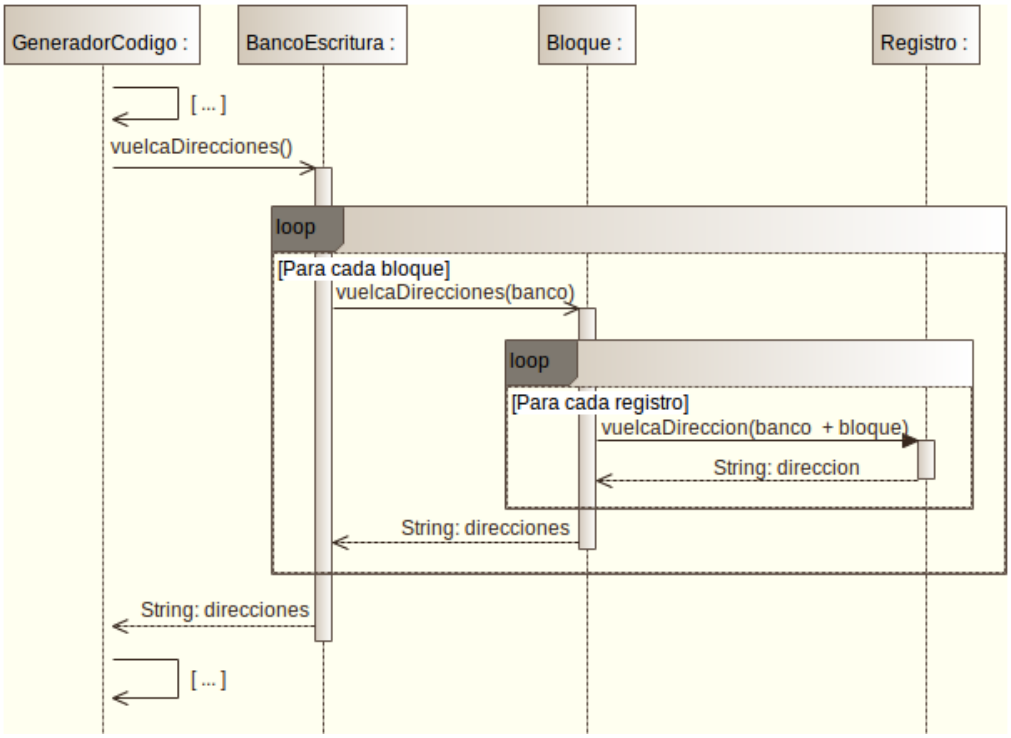

Figura 20: Diagrama de Secuencia: ejemplo de llamadas sucesivas en la generación de código

4.3. Verificación

Las verificaciones aquí realizadas son pruebas de caja negra sobre la herramienta de traducción. Se dividen en dos grupos. En primer lugar, dos pruebas para verificar la corrección del código generado, una centrada en la asignación de nombres y la otra en la asignación de valores. Posteriormente, una

serie de pruebas para verificar el correcto funcionamiento frente a entradas inválidas, con errores léxicos y sintácticos por un lado, y errores semánticos por otro.

4.3.1. Comprobaciones sobre el correcto nombrado

Una importante cuestión sobre la que se han realizado sucesivas verificaciones es el correcto nombrado de las variables y tipos. Las repeticiones de nombres en distintos contextos, así como la posibilidad de repetir un bloque varias veces, supone que aparecen gran cantidad de variables y tipos de datos que en principio se llamarían igual.

La solución adoptada para la repetición de nombres ha sido utilizar prefijos. En el fichero de especificaciones el nombre de los tipos y direcciones utiliza la estructura "banco_bloque_registro", mientras que en los ficheros de los propios bancos, se utiliza únicamente "bloque_registro". Para permitir la repetición de un mismo bloque varias veces se ha optado por utilizar sufijos numéricos.

Para realizar estas comprobaciones se ha definido un fichero de entrada con las siguientes características:

- Nombres de bloque repetidos en bloques de distinto tipo.
- Nombres de registro repetidos en diferentes bloques.
- Nombres de campo repetidos en diferentes registros.
- Las tres anteriores en bloques con y sin repeticiones.

La comprobación se ha realizado sobre los ficheros de salida, asegurando que cada tipo y cada variable tenga un identificador único en el código y que se garantiza además la correcta correspondencia con las estructuras definidas en la entrada.

Gracias a estas pruebas de verificación se han corregido errores, ya que en las primeras aproximaciones se habían pasado por alto aspectos que generaban tipos y variables repetidas. Como, por ejemplo, que un bloque de lectura y otro de escritura se llamasen igual, o no haber tenido en cuenta los prefijos en el multiplexor de salida del banco de lectura.

4.3.2. Comprobaciones sobre la asignación de valores

Otra de las principales preocupaciones era la correcta conversión y asignación de valores, especialmente al crear las instancias de los registros genéricos. Esto supone varios aspectos a tener en cuenta:

- En registros de un solo campo, la correcta asignación de los valores de reset, máximos y mínimos, tanto para registros signed como unsigned como máscaras de bits.
- La correcta activación del flag de truncado, únicamente en registros de un solo campo con máximos o mínimos y truncado.
- La correcta activación del flag de signed, únicamente en registros signed de un solo campo y con máximos o mínimos.
- En registros de varios campos, la correcta composición del valor final de reset con respecto a los valores de reset, tamaños y posiciones de cada uno de sus campos tanto de tipo signed como unsigned como máscaras de bits.
- La correcta asignación de los valores de las direcciones en orden de aparición de los registros, independientemente de que haya bloques de tipos diferentes intercalados.

Para realizar estas comprobaciones se ha definido un fichero de entrada con las siguientes características:

- Registros con huecos.
- Registros de un solo campo con mínimo.
- Registros de un solo campo con máximo.
- Registros de un solo campo con mínimo y máximo.
- Las tres anteriores con truncado.
- Las cuatro anteriores con signed.
- Valores de reset signed, unsigned y mascaras de bits.
- Bloques de diferente tipo intercalados.

Gracias a estas pruebas de verificación se han detectado y solucionado errores de diversos tipos. Algunos ejemplos representativos serían los siguientes:

- Se estaban teniendo en cuenta los campos vacíos al comprobar la cantidad de campos, con lo que no se permitía poner máximos ni mínimos si el campo tenía menos de 16 bits.
- A la hora de calcular los valores de reset en registros con espacios vacíos, no se estaban tomando bien los desplazamientos.
- A la hora de calcular los valores de reset en registros con campos de diversos tipos, no se estaba teniendo en cuenta a la hora de concatenarlos que los campos signed rellenaban con unos los bits de mayor peso más allá del tamaño del campo, siendo necesario utilizar una máscara para eliminarlos.

4.3.3. Comprobaciones sobre la generación de errores

Los errores sintácticos no son fáciles de definir claramente en un conjunto de pruebas, ya que pueden responder a gran cantidad de cuestiones. Un error sintáctico viene a ser que, en cualquier momento, el analizador encuentre cualquier cosa que no esperaba.

Los errores léxicos por su parte también son muy amplios, pero podríamos reducirlos a dos grupos básicos. Un símbolo que no se reconozca, o un patrón de símbolos que no se corresponda con ningún token.

Para las verificaciones léxicas se han reproducido estas dos situaciones de error léxico descritas. Mientras que para las sintácticas, se han realizado un conjunto de pruebas que caractericen algunos de los posibles errores sintácticos más fáciles de cometer en las especificaciones de entrada. Los resultados de unas y otras se exponen de forma detallada en los anexos (B.1).

La comprobación de errores semánticos está más claramente definida. Se ha realizado una o más pruebas específicas para cada uno de los errores definidos en la clase de análisis semántico. En los anexos (B.2) se recoge cada una de las pruebas realizadas y el correspondiente comportamiento de la salida.

Los resultados recogidos responden al comportamiento del sistema final, una vez solucionados los errores encontrados. Estas pruebas se realizaron paralelamente a las anteriormente descritas, y se iban solucionando errores conforme se encontraban, tanto de unas como de otras.

En el caso de los errores semánticos se encontraron algunos problemas como, por ejemplo, que al tomar un reset de tipo mascara de bits se comparaba su valor decimal con el rango del registro, pero no se contaba la cantidad de unos y ceros, con lo que se aceptaba tanto un valor más corto como uno más largo con ceros en las posiciones de más peso.

4.3.4. Comprobaciones sobre el código generado

Desde la empresa se consideró que las verificaciones de funcionamiento de los bancos de registros debían realizarse de forma externa a este proyecto. De esta forma se garantiza la independencia entre la generación de los test y el proceso de diseño. Para ello definieron un proyecto paralelo, llevado a cabo por otro compañero, para la verificación automática de bancos de registros VHDL utilizando el lenguaje SistemVerilog.

Al inicio de este trabajo, en la fase de definición de las especificaciones, sí se realizaron de manera no formalizada pruebas sobre código VHDL, ya que debía definirse un modelo para generar los bancos de registro en la salida.

Sin embargo las pruebas formales de verificación de la salida se llevaron a cabo de forma externa, desde este otro proyecto. La realimentación recibida fue muy positiva para la mejora del diseño y, finalmente, todos los tipos de bancos se comportaron conforme a las especificaciones, superando todas las pruebas.

5. Conclusiones

5.1. Desarrollo del trabajo y consecución de los objetivos

En términos generales, considero el desarrollo de este trabajo muy satisfactorio, tanto en lo referente a la realización del proyecto en sí, como en el ámbito de mi propia formación académica.

La parte que más tiempo se ha llevado ha sido el desarrollo del programa, que se corresponde con el tercer objetivo específico, teniendo en cuenta tanto la organización y diseño como la implementación y verificaciones. Especialmente ha supuesto un reto tener que modificar las estructuras normalmente utilizadas para el desarrollo de un compilador, con las que había trabajado anteriormente, para adaptarlas a un problema con ciertas características similares pero con muchos otros aspectos muy particulares.

Aproximadamente esta parte ha supuesto unas 160 horas de trabajo. Buena parte de este tiempo ha tenido que ver con el diseño y la organización, pero posiblemente la mayoría ha sido un trabajo mecánico, de implementación, documentación y verificación.

Aunque el producto final de este proyecto sea el programa en sí, una vez realizado el trabajo considero que el corazón del mismo ha estado en la primera fase, la correspondiente con los dos primeros objetivos específicos, es decir, la generación de las especificaciones.

Esta parte del proyecto abarca unas 30 horas de reuniones y otras 80 de trabajo individual, sin embargo ha sido la parte más creativa y ha supuesto mucho más trabajo intelectual. En el desarrollo de las reuniones es donde he podido demostrar conocimientos muy específicos de mi rama, y donde he apreciado realmente la importancia de dominar determinados detalles relacionados con la implementación a bajo nivel.

La Figura 21 muestra las horas finales invertidas. Para ofrecer una visión completa de las tareas realizadas se adjuntan como anexo los diagramas de Gantt (B.2) detallados de todo el proceso.

	reuniones y presentaciones	especificaciones	desarrollo	TOTAL
HORAS	30	81,5	174	285,5

Figura 21: Resumen de horas invertidas

5.2. Incidencias y aprendizajes

El hecho de haber realizado este trabajo a petición de una empresa ha sido posiblemente uno de los aspectos más interesantes del mismo de cara a mi aprendizaje personal. He podido ver la importancia de la coordinación entre diferentes partes externas al desarrollo del proyecto como tal.

Además se trata de un proyecto inmerso en un momento de modificaciones en la forma de trabajo de la empresa, con lo que he podido vivir sucesivas modificaciones de la idea inicial, e incluso importantes cambios durante el propio desarrollo del programa.

Esto ha dado si cabe más valor al hecho de estar utilizando unas tecnologías y un modelo de desarrollo que me han facilitado la adaptación a los cambios, que de otra forma habría sido mucho más tediosa.

Un segundo aspecto que me gustaría destacar es la importancia de las verificaciones. Esta es una cuestión que a veces se deja de lado en muchas asignaturas de la carrera, ya que normalmente las prácticas realizadas tienen un contexto muy concreto o un tamaño muy reducido.

Gracias a las pruebas de verificación realizadas he podido solucionar gran cantidad de problemas. Algunos de los errores encontrados han sido pequeños detalles que había pasado por alto en el diseño.

Pero lo más curioso es que muchos otros han sido cuestiones que conceptualmente quedaban muy claras, y que había tenido en cuenta en mi diseño, pero que luego a la hora de la implementación habían quedado mal definidos. Desde problemas con los tipos de datos, hasta directamente olvidos y despistes.

Esto me da cierta idea de lo que debe suponer la realización de un gran proyecto software, con un equipo de personas encargadas de diferentes partes

que posteriormente hay que unir y comunicar. Desde luego las pruebas y verificaciones deben suponer gran cantidad de tiempo y llevarse un porcentaje tremendo del trabajo.

5.3. Perspectivas de continuación

Como ya se ha sostenido en varias ocasiones, una de las prioridades que ha estado presente en todo el desarrollo del proyecto ha sido la claridad y modularidad del código. Una de sus principales ventajas es, por ejemplo, facilitar la posible ampliación de las funcionalidades de la herramienta.

Esa podría ser una continuación lógica de este trabajo. Añadir funcionalidades y opciones, ya de forma independiente a los requisitos concretos de este caso particular, para que el traductor sea capaz de generar bancos de muchos tipos. Esto lo convertiría en una herramienta más genérica para la generación automática de bancos de registros, pudiendo resultar útil para diferentes propósitos.

Por otro lado, este proyecto podría suponer un punto de partida para realizar experiencias similares con otras partes del hardware además de los bancos de registros. De esta manera, un amplio conjunto de herramientas de traducción específicas para diferentes componentes podría acabar constituyendo una capa de abstracción sobre VHDL para facilitar la generación de determinadas partes genéricas de un diseño hardware.

A. Especificaciones formales

A.1. Patrones y tokens

A continuación se muestra la definición de tokens tal y como se ha especificado en la definición del programa en JavaCC.

```
<DEFAULT> SKIP : {
  <~[]>
}

<DEFAULT> TOKEN : {
  <tINICIO: "--INICIO_ESPECIFICACIONES">: CUERPO
}

<CUERPO> SKIP : {
  " "
| "\t"
}

<CUERPO> TOKEN : {
  <tCOMA: ",">
| <tPARAB: "(">
| <tPARCE: ")">
| <tNULL: "$">
| <tSEPARACION: "#">
| <tSI: "si">
| <tNO: "no">
| <tRD: "rd">
| <tWR: "wr">
| <tFIN: "--FIN_ESPECIFICACIONES">: DEFAULT
| <#L: ["a"-"z"]>
```

```
| <#D: ["0"-"9"]>
| <tLINEA: ("\n" | "\r" | "\r\n")>
| <tBINARIO: "b" ("0" | "1")+>
| <tNATURAL: (<D>)+>
| <tNEGATIVO: "-" <tNATURAL>>
| <tUNSIGNED: "u" <tNATURAL>>
| <tSIGNED: "s" (<tNATURAL> | <tNEGATIVO>)>
| <tIDENTIFICADOR: <L> | (<L> | <D> | "_" )+ (<L> | <D>)>
}
```

A.2. Sintáxis

A continuación se muestra la definición de la gramática a partir de la cual se han definido las funciones del programa en JavaCC.

```
inicio ::= <tINICIO> (fin_linea)+ cuerpo <tFIN>

fin_linea ::= (<tCOMA>)* <tLINEA>

cuerpo ::= (declaracion_bloque (fin_linea)+)+

declaracion_bloque ::= <tIDENTIFICADOR> <tCOMA>
  (<tRD> <tCOMA> <tNATURAL> fin_linea lista_registros_lectura
| <tWR> <tCOMA> <tNATURAL> fin_linea lista_registros_escritura)

lista_registros_lectura ::= (registro_lectura fin_linea)+

lista_registros_escritura ::= (registro_escritura fin_linea)+

registro_lectura ::= <tIDENTIFICADOR> <tCOMA>
  <tIDENTIFICADOR> <tCOMA>
  <tIDENTIFICADOR> <tCOMA>
  campos_registro
  <tSEPARACION>

registro_escritura ::= <tIDENTIFICADOR> <tCOMA>
  <tIDENTIFICADOR> <tCOMA>
  <tIDENTIFICADOR> <tCOMA>
  campos_registro
```

```
  <tSEPARACION> <tCOMA>
  reset_registro
  (<tNATURAL> | <tNEGATIVO>)? <tCOMA>
  (<tNATURAL> | <tNEGATIVO>)? <tCOMA>
  (<tSI> | <tNO>)?

campos_registro ::= ((identificador_registro)? <tCOMA>){16}

identificador_registro ::= (<tNULL> | <tIDENTIFICADOR>)
  (<tPARAB> <tNATURAL> <tPARCE>)?

reset_registro ::= ((valor_registro)? <tCOMA>){16}

valor_registro ::= <tNULL> (<tPARAB> <tNATURAL> <tPARCE>)?
| <tBINARIO>
| <tUNSIGNED>
| <tSIGNED>
```

A.3. Explicación detallada de la sintáxis y la gramática

A continuación se detalla el formato final que debe cumplir el fichero csv que recibirá como entrada la herramienta de traducción. Para ello se ha tratado de realizar una serie de definiciones con un compromiso entre formalidad y claridad, procurando no recurrir a la utilización de gramáticas u otras especificaciones formales, pero manteniendo a su vez el máximo rigor posible.

En dichas definiciones se habla de "filas" y de "celdas" atendiendo al formato de hoja de cálculo en el que presumiblemente serán escritas. A efectos del documento csv que se generará como entrada, cada fila constituye una línea y los contenidos de cada celda dentro de la misma van separados por comas. Al final de una fila, antes del salto de línea, puede haber comas de sobra, lo que equivale a celdas vacías en el caso de la hoja de cálculo.

El orden seguido en las explicaciones va de lo general a lo particular, de forma que se facilita la visión global del documento y, sucesivamente, de cada una de sus partes. Esto supone que aparecen constantemente referencias entrecomilladas a determinadas definiciones que se detallan siempre posteriormente a su aparición.

- Pueden incluirse cabeceras con cualquier formato y contenido. Estas no se tendrán en cuenta como parte de lo que llamaremos "especificaciones formales", las cuales deben cumplirse para que la entrada sea válida.

- **Especificaciones formales.** Comenzarán con una fila que únicamente contendrá la palabra clave "--INICIO_ESPECIFICACIONES" en su primera celda y terminarán con una fila que contendrá la palabra clave "--FIN_ESPECIFICACIONES" en su primera celda. El "resto de las especificaciones formales" estará contenido entre estas dos líneas.

- **Resto de las especificaciones formales.** Estará formado por una o más "especificaciones de bloques", las cuales pueden ir precedidas y seguidas de tantas filas vacías como se desee, debiendo estar separadas entre sí por al menos una fila vacía.

- **Especificación de bloque.** Estará formada por una fila, la primera, que llamaremos la "definición de bloque", y seguidamente una fila de "definición de registro" por cada registro que contenga el bloque. Dentro de una "especificación de bloque" no debe haber ninguna fila vacía.

- Fila de **Definición de bloque.** Únicamente tendrá contenido en sus tres primeras celdas, por este orden:

  - Nombre del bloque, sin repetición entre bloques y cumpliendo la "sintaxis de los identificadores".
  - Tipo de bloque, indicando si pertenece al banco de lectura o escritura mediante los indicadores "RD" o "WR" respectivamente.
  - Cantidad de repeticiones del bloque, que será un entero mayor que cero.

- **Definición de registro.** Su formato será diferente en caso de tratarse de un bloque perteneciente al banco de lectura o de escritura.

- **Definición de registro en el caso de un bloque perteneciente al banco de lectura.** Contendrá, por orden y empezando desde el principio de la fila, las siguientes 20 celdas:

  - Nombre del bloque, sin repetición dentro del mismo bloque y cumpliendo la "sintaxis de los identificadores".
  - Nombre de la línea de fase asociada al registro, cumpliendo la "sintaxis de los identificadores".

- Nombre de la línea de reloj asociada al registro, cumpliendo la "sintaxis de los identificadores".
- 16 celdas con la "definición de campos" del registro.
- Símbolo "#" indicando el final de la "definición de campos".

- **Definición de registro en el caso de un bloque perteneciente al banco de escritura.** Comenzará con el mismo contenido anterior y además, tras el símbolo "#", contendrá las siguientes celdas:

  - 16 celdas con la "inicialización de campos" del registro.
  - Un entero con el valor mínimo aceptado por el registro. Este valor es opcional, pero debe respetarse su posición aunque se deje vacío. Únicamente está permitido en registros con un solo campo y debe respetar el tipo de dato, "signed" o "unsigned", y rango de valores del mismo.
  - Un entero con el valor máximo aceptado por el registro. Este valor es opcional, pero debe respetarse su posición aunque se deje vacío. Únicamente está permitido en registros con un solo campo y debe respetar el tipo de dato, "signed" o "unsigned", y rango de valores del mismo.
  - Indicador de truncado. Este valor es opcional y únicamente tiene efecto en caso de que se haya establecido un máximo o un mínimo en las anteriores celdas. El valor "SI" indica que, en caso de que la entrada recibida supere dichos límites, el registro lo sustituirá por el límite más cercano. El valor "NO" es la opción por defecto e indica que, en dicho caso, el registro ignorará la entrada recibida.

- **Definición de campos.** Estará formada por 16 celdas siguiendo las siguientes indicaciones:

  - Cada celda representa un bit del registro, comenzando de izquierda a derecha por el bit más significativo.
  - La definición de cada campo estará únicamente en la celda correspondiente al bit más significativo del mismo, debiendo quedar vacías, a su derecha, el resto de celdas correspondientes a dicho campo. En el caso de la hoja de cálculo, se permite agrupar dichas celdas por visibilidad, manteniendo el valor solo en la primera, teniendo en cuenta que la conversión a csv respetará las comas necesarias.

  - Cada campo se definirá mediante su nombre y seguidamente, en caso de ocupar más de un bit, la cantidad de bits del mismo entre paréntesis.
  - Los espacios vacíos del registro se tratarán como campos, respetando las mismas normas en su definición y pudiendo agrupar aquellos que sean consecutivos, con la única diferencia de que en lugar de un nombre se utilizará la marca "$" como identificador.

- **Inicialización de campos.** Estará formada por 16 celdas y deberá corresponderse con la definición de campos del registro, siguiendo las siguientes indicaciones:

  - Cada celda representa un bit del registro, comenzando de izquierda a derecha por el bit más significativo.
  - La inicialización de cada campo deberá ocupar la misma celda que en su correspondiente "definición de campos", respetando así mismo sus celdas vacías. Se permite igualmente la agrupación en la hoja de cálculo.
  - Cada campo se inicializará mediante el "tipo del campo" seguido inmediatamente por el "valor del campo".
  - **Tipo del campo.** Se especifica mediante el carácter "b", "u" o "s" para definirlo respectivamente como "máscara binaria", "unsigned" o "signed".
  - **Valor del campo.** Tanto en los campos de tipo "signed" como "unsigned" será un entero, pudiendo tomar valores negativos únicamente en el caso de ser "signed". El valor de un campo con tipo "máscara binaria" se compondrá de tantos unos o ceros como bits tenga el campo. Independientemente del tipo, deberá respetar los límites del registro en caso de que este los tenga.
  - Los espacios vacíos deberán rellenarse igual que en la "definición de campos" del registro.

- **Sintaxis de los identificadores.**

  - Los únicos símbolos válidos son letras, dígitos y "_", no pudiendo terminar en "_".
  - No se hace diferenciación entre mayúsculas y minúsculas.
  - Si el identificador está compuesto por un único carácter, este debe ser una letra.

B. Verificaciones formales

B.1. Comprobación de errores léxicos y sintácticos

Un error léxico finaliza automáticamente la ejecución, mientras que tras error sintáctico la ejecución continúa para tratar de encontrar más. Esto produce que un error sintáctico genere otros errores "falsos". Además el resultado varía según esté o no activo el "panic mode".

En las pruebas recogidas en la tabla de la Figura 22 solo se recoge el primer error, correspondiente con la situación forzada.

Figura 22: Tabla de comprobación de errores léxicos y sintácticos

Prueba realizada	**Resultado obtenido**
lex01.csv Identificador con un símbolo no permitido	ERROR LÉXICO (5, A): símbolo no reconocido: ?
lex02.csv Un símbolo "_" donde no corresponde	ERROR LÉXICO (3, A): patrón o símbolo erróneo. Encontrado "," despues de "_"
lex03.csv Un símbolo "_" como nombre de registro	En este caso no se produce un error léxico, pero se pierde la declaración del registro, lo que provoca errores semánticos: ERROR SEMANTICO (6, D): No se ha declarado ningún registro en el bloque actual.
sin01.csv No dejar una línea entre dos declaraciones de bloques	ERROR SINTÁCTICO (11, B) Se encontró : <tRD> "RD" Se esperaba : <tIDENTIFICADOR> Se encuentra la declaración de bloque cuando se esperaba otro registro.

Continuación: Tabla de comprobación de errores léxicos y sintácticos

Prueba realizada	**Resultado obtenido**
sin02.csv Declarar nombres con espacios	ERROR SINTÁCTICO (6, A) Se encontró : <tIDENTIFICADOR> “reg” Se esperaba : “,” Se espera la “,” del siguiente campo porque se considera que el identificador ya ha terminado.
sin03.csv Valor del mínimo colocado en la celda anterior	ERROR SINTÁCTICO (10, AJ) Se encontró : <tNATURAL> “40” Se esperaba : “,”, “$”, <tBINARIO>, ... Al corresponderse con una celda de la declaración del reset no acepta un número sin más.
sin04.csv Máscara de bits con números distintos de unos y ceros	ERROR SINTÁCTICO (8, U) Se encontró : <tIDENTIFICADOR> “b13” Se esperaba : “,”, “$”, <tBINARIO>, ... Al corresponderse con el tipo binario lo interpreta como un identificador.
sin05.csv Mínimo con el formato de los valores de reset	ERROR SINTÁCTICO (7, AL) Se encontró : <tUNSIGNED> “u9000” Se esperaba : “,”, <tNATURAL>, ... Se espera un valor numérico sin la declaración del tipo.

B.2. Comprobación de los errores semánticos

La tabla de la Figura B.2 recoge la salida recibida para cada error semántico analizado. Se comentan además posibles diferencias que geneara la activación del “panic mode”.

Figura 23: Tabla de comprobación de los errores semánticos

Prueba realizada	**Resultado obtenido**
sem01.csv Añadir registros sin bloque (sin declarar)	ERROR SINTÁCTICO (5, B) Se encontró : tIDENTIFICADOR “fase1” Se esperaba : “rd”, “wr” Como era de esperar, se obtiene error sintáctico porque se espera la declaración del bloque, no la del registro.

Continuación: Tabla de comprobación de los errores semánticos

Prueba realizada	**Resultado obtenido**
sem02.csv Añadir registros sin bloque (sin nombre)	ERROR SINTÁCTICO (5, B) Se encontró : <tWR> “WR” Se esperaba : “,”, <tLINEA> Si se ejecuta sin el “panic mode”, después de los errores sintácticos aparece además, para cada registro: ERROR SEMANTICO (6, D): No se ha declarado un bloque en el banco correspondiente.
sem03.csv Nombre de bloque repetido (mismo tipo de bloque)	ERROR SEMANTICO (12, A): Ya se ha declarado un bloque con nombre bloque_1.
sem04.csv Añadir campos sin nombre de registro	ERROR SINTÁCTICO (6, C) Se encontró : <tNULL> “$” Se esperaba : <tIDENTIFICADOR> Esta vez el error semántico aparece tanto con “panic mode” como sin él: ERROR SEMANTICO (6, C): No se ha declarado ningún registro en el bloque actual.
sem05.csv Nombre de registro duplicado en el mismo banco	ERROR SEMANTICO (7, D): Ya se ha declarado un registro con nombre reg1.
sem06.csv Nombre de campo en el espacio del campo anterior	ERROR SEMANTICO (8, I): El campo c1 se superpone con el anterior.
sem07.csv Campo que se sale de los 16 bits del registro	ERROR SEMANTICO (8, L): El campo L se sale del registro.
sem08.csv Nombre de campo repetido en el mismo registro	ERROR SEMANTICO (8, L): Ya se ha declarado un campo con nombre H.
sem09.csv Más comas de las esperadas al final del registro	ERROR SINTÁCTICO (6, U) Se encontró : <tCOMA> “,” Se esperaba : “#” En este caso los errores semánticos solo aparecen con el “panic mode” activo para registros posteriores:

Continuación: Tabla de comprobación de los errores semánticos

Prueba realizada	**Resultado obtenido**
	ERROR SEMANTICO (7, F): Se esperaba un nuevo campo, o bien un '$'. ERROR SEMANTICO (8, Z): El campo no coincide con los previamente declarados.
sem10.csv Más comas de las esperadas entre dos campos (numero mal)	ERROR SEMANTICO (8, J): Se esperaba un nuevo campo, o bien un '$'. ERROR SEMANTICO (8, Z): El campo no coincide con los previamente declarados. Y posteriormente aparece el error sintáctico: ERROR SINTÁCTICO (8, A) Se encontró : <tLINEA> " " Se esperaba : ",", <tNATURAL>, ...
sem11.csv Más comas de las esperadas entre dos campos (coma extra)	ERROR SEMANTICO (8, J): Se esperaba un nuevo campo, o bien un '$'. ERROR SEMANTICO (8, M): El campo L se sale del registro. Y posteriormente aparece el error sintáctico: ERROR SINTÁCTICO (8, U) Se encontró : <tCOMA> "," Se esperaba : "#"
sem12.csv Más comas de las esperadas entre dos campos (desplazado)	ERROR SEMANTICO (8, J): Se esperaba un nuevo campo, o bien un '$'. ERROR SEMANTICO (8, L): El campo L se superpone con el anterior. ERROR SEMANTICO (8, L): El campo L se sale del registro. ERROR SEMANTICO (8, Z): El campo no coincide con los previamente declarados.
sem13.csv Diferente estructura de campos en el reset	ERROR SEMANTICO (8, W): El campo no coincide con los previamente declarados.
sem14.csv Valor mínimo en registro con más de un campo	ERROR SEMANTICO (6, AK): Solo se admite un mínimo en registros con un único campo.

Continuación: Tabla de comprobación de los errores semánticos

Prueba realizada	**Resultado obtenido**
sem15.csv Valor máximo en registro con más de un campo	ERROR SEMANTICO (6, AL): Solo se admite un máximo en registros con un único campo.
sem16.csv Valor de reset menor que el mínimo (mínimo válido)	ERROR SEMANTICO (10, AK): El valor de reset es inferior al mínimo.
sem17.csv Valor de reset superior al máximo (máximo valido)	ERROR SEMANTICO (7, AL): El valor de reset es superior al máximo.
sem18.csv Valor máximo menor que el valor mínimo válido del registro	ERROR SEMANTICO (10, AL): El valor máximo es inferior al mínimo.
sem19.csv Reset mayor que el rango del campo (unsigned)	ERROR SEMANTICO (8, AC): El valor de reset no cabe en el campo correspondiente.
sem20.csv Reset mayor que el rango del campo (signed)	ERROR SEMANTICO (8, AC): El valor de reset no cabe en el campo correspondiente.
sem21.csv Máscara de bits de reset con más bits que el campo	ERROR SEMANTICO (8, AC): La máscara de bits de reset no tiene el tamaño adecuado.
sem22.csv Reset menor que el rango del campo (unsigned)	ERROR SINTÁCTICO (8, AC) Se encontró : <tIDENTIFICADOR> “u” Se esperaba : “,”, “$”, <tBINARIO>, ... Se obtiene un error sintáctico, como era de esperar, ya que esto supone ponerle valor negativo a un unsigned.
sem23.csv Reset menor que el rango del campo (signed)	ERROR SEMANTICO (8, AC): El valor de reset no cabe en el campo correspondiente.
sem24.csv Máscara de bits de reset con menos bits que el campo	ERROR SEMANTICO (8, AC): La máscara de bits de reset no tiene el tamaño adecuado.
sem25.csv Mínimo menor que el rango del campo (único y unsigned)	ERROR SEMANTICO (10, AK): El valor del mínimo no cabe en el campo correspondiente.

Continuación: Tabla de comprobación de los errores semánticos

Prueba realizada	**Resultado obtenido**
sem26.csv Mínimo menor que el rango del campo (único y signed)	ERROR SEMANTICO (10, AK): El valor del mínimo no cabe en el campo correspondiente.
sem27.csv Máximo mayor que el rango del campo (único y unsigned)	ERROR SEMANTICO (7, AL): El valor del máximo no cabe en el campo correspondiente.
sem28.csv Máximo mayor que el rango del campo (único y signed)	ERROR SEMANTICO (7, AL): El valor del máximo no cabe en el campo correspondiente.
sem29.csv Valor "SI" en truncado sin haber mínimo ni máximo	WARNING (9, AM): No se ha declarado valor mínimo ni máximo para el registro. El flag de truncado no tendrá efecto. Únicamente avisa por si se trata de un error en las especificaciones, pero se generan los ficheros de salida.

C. Diagramas de Gantt

2016/2017	ABRIL																												
	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
RESUMEN EVENTOS:																														
Reuniones														3											3					
Presentación parcial																														
TAREAS REALIZADAS:																														
Revisión VHDL y primeras pruebas				2			4				2																			
Primera reunión con Denis														2																
Aproximaciones al Registro Genérico														4	4															
Primera especificación de la entrada																		1												
Primera Gramática y analizador																		2		2										
Pruebas en un banco de escritura																				1	2									
Primera reunión BSH																									4					
Implementación del Registro Genérico																											4			
Generación de un Banco de Prueba																												3		
Generación de un programa de test																												1	1	
Realización de test con varios supuestos																													1	

Figura 24: Diagrama de Gantt de abril

2016/2017	MAYO																													
	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESUMEN EVENTOS:																															
Reuniones					3												3														
Presentación parcial																										3					
TAREAS REALIZADAS:																															
- - Bits constantes y signed/unsigned		2																													
Mini reunión con Denis					0,5																										
Análisis cláusula “Generate”					2																										
Modificación del Registro Genérico					1				2																						
Modificación del Banco de Prueba												2																			
Modificación del programa de test												1																			
Realización de test con varios supuestos												1																			
Segunda reunión BSH																	4														
Modificación de las especificaciones																		1													
Generación de ejemplo de especificación																		1													
Modificación del Registro Genérico																			2												
Modificación del Banco de Prueba																			2												
Modificación del programa de test																				2											
Realización de test iniciales																				1											
Preparación de verificaciones																				1	1										
Preparar presentación																							4	6							

Figura 25: Diagrama de Gantt de mayo

2016/2017	JUNIO																													
	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
RESUMEN EVENTOS:																														
Reuniones																														
Presentación parcial																														
TAREAS REALIZADAS:																														
Especificación de la gramática final																												1	2	
Generación de la gramática final																													2	
Organización de las estructuras																														2
Generación de clases y funciones																														2

Figura 26: Diagrama de Gantt de junio

2016/2017	JULIO																													
	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESUMEN EVENTOS:																															
Reuniones													3																		
Presentación parcial																															
TAREAS REALIZADAS:																															
Registro genérico de lectura					1																										
Pruebas sobre un banco de lectura					1																										
Generación de clases y funciones					5	2																									
Reunión con Denis (detalles de espec.)													2																		
Reorganización final del proyecto													1																		
Adaptación de la gramática													1																		
Implementación de funciones																				4	1										
Adaptación de un ejemplo real																						1									
Implementación de funciones																							1			4					

Figura 27: Diagrama de Gantt de julio

2016/2017	AGOSTO																														
	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESUMEN EVENTOS:																															
Reuniones													3																		
Presentación parcial																															
TAREAS REALIZADAS:																															
Funciones de Gen. Cod. (Espec. Y B.L.)			4		3																										
Tratamiento de extensiones y rutas						2																									
Funciones de Gen. Cod. (B. L.)						2																									
Funciones de Gen. Cod. (B. E.)						2			1																						
Documentación (comentarios javadoc)									3	2	1		6	2																	
Depuración de max, min (y warning)														2																	

Figura 28: Diagrama de Gantt de agosto

2016/2017	OCTUBRE																														
	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESUMEN EVENTOS:																															
Reuniones													3																		
Presentación parcial																															
TAREAS REALIZADAS:																															
Solución errores en las conexiones E/S					2																										
Ejemplos para verificación											2																				
Actualización de nombres y direcciones											2																				
Modificación del tipo de dato dirección																		1													
Solución errores reset																		1													
Implementación BL Combinacional																										2					

Figura 29: Diagrama de Gantt de octubre

2016/2017	NOVIEMBRE																														
	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
RESUMEN EVENTOS:																															
Reuniones																															
Presentación parcial																															
TAREAS REALIZADAS:																															
Generación automática registros genéricos		3																													
Preparar org.apache.commons.cli			1																												
Gestión de parámetros de entrada			2																												
Presentación inicial BSH				1																											
Solución nombres registro duplicados														1																	
Gestión de la ayuda y parámetros erróneos																	5														

Figura 30: Diagrama de Gantt de noviembre

2016/2017	DICIEMBRE																													
	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESUMEN EVENTOS:																															
Reuniones													3																		
Presentación parcial																															
TAREAS REALIZADAS:																															
Preparación de plantilla LaTex						2																									
Detalles finales de javadoc									2					4																	
Organización Memoria																						4	3								
Salida en directorios																												1			
Formatos y estandarización Memoria																													4	2	

Figura 31: Diagrama de Gantt de diciembre

2016/2017	ENERO																													
	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RESUMEN EVENTOS:																															
Reuniones													3																		
Presentación parcial																															
TAREAS REALIZADAS:																															
Contenido y formato de Memoria			2						2	4	5	7	3																		
Formalización de las verificaciones																										6	3			1	
Contenido y formato de Memoria																										2	2			7	8

Figura 32: Diagrama de Gantt de enero

2016/2017	FEBRERO																											
	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M	X	J	V	S	D	L	M
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
RESUMEN EVENTOS:																												
Reuniones																												
Presentación parcial																												
TAREAS REALIZADAS:																												
Contenido y formato de Memoria	8	8	3																									

Figura 33: Diagrama de Gantt de febrero