
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN INGENIERÍA INDUSTRIAL

# EXTENSIÓN DEL JUEGO DE INSTRUCCIONES DE LA IP “NPROC” DE AIRBUS-CRISA: DISEÑO, VERIFICACIÓN Y VALIDACIÓN EN FPGA

Autor: Rita de Miguel Fernández
Director: Álvaro Pérez de la Fuente

Madrid
Junio 2018

**AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESINAS O MEMORIAS DE BACHILLERATO**

***1º. Declaración de la autoría y acreditación de la misma.***

El autor D. Rita de Miguel Fernández DECLARA ser el titular de los derechos de propiedad intelectual de la obra: "Extensión del Juego de Instrucciones de la IP "nProc" de Airbus-Crisa: Diseño, Verificación y Validación en FPGA", que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

***2º. Objeto y fines de la cesión.***

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

***3º. Condiciones de la cesión y acceso***

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar "marcas de agua" o cualquier otro sistema de seguridad o de protección.
b) Reproducirla en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
e) Asignar por defecto a estos trabajos una licencia Creative Commons.
f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente).*

***4º. Derechos del autor.***

El autor, en tanto que titular de una obra tiene derecho a:

a) Que la Universidad identifique claramente su nombre como autor de la misma
b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
c) Solicitar la retirada de la obra del repositorio por causa justificada.
d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

***5º. Deberes del autor.***

El autor se compromete a:

a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e intereses a causa de la cesión.

d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

***6º. Fines y funcionamiento del Repositorio Institucional.***

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a 15 de Junio de 2018

**ACEPTA**

Fdo. Rita de Miguel Fernández

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título "Extensión del Juego de Instrucciones de la IP "nProc" de Airbus-Crisa: Diseño, Verificación y Validación en FPGA" en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2017/2018 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada

de otros documentos está debidamente referenciada.

Fdo.: Rita de Miguel Fernández Fecha: 14/06/2018

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Álvaro Pérez de la Fuente Fecha: ..16./ ..06../ ..2018


ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN INGENIERÍA INDUSTRIAL

# EXTENSIÓN DEL JUEGO DE INSTRUCCIONES DE LA IP “NPROC” DE AIRBUS-CRISA: DISEÑO, VERIFICACIÓN Y VALIDACIÓN EN FPGA

Autor: Rita de Miguel Fernández
Director: Álvaro Pérez de la Fuente

Madrid
Junio 2018

# EXTENSIÓN DEL JUEGO DE INSTRUCCIONES DE LA IP "NPROC" DE AIRBUS-CRISA: DISEÑO, VERIFICACIÓN Y VALIDACIÓN EN FPGA

**Autor: de Miguel Fernández, Rita.**

Director: Pérez de la Fuente, Álvaro.

Entidad Colaboradora: Crisa (Computadoras, Redes e Ingeniería, S.A.U.).

## RESUMEN DEL PROYECTO

### Introducción

Crisa (Computadoras, Redes e Ingeniería, S.A.U.) es una empresa española que se encarga de producir equipos electrónicos con aplicaciones aeroespaciales [1]. Cuando se trata de este tipo de aplicaciones, lo más común es la utilización de FPGAs (*Field Programmable Gate Array*), debido a que son reprogramables y los costes de desarrollo de las mismas son más bajos que los de otros dispositivos válidos para esta función.

Este proyecto se centra en mejorar una IP (Propiedad Intelectual) de la empresa, para así conseguir una mayor versatilidad de uso de la misma, ofreciendo un mayor abanico de posibilidades a futuros usuarios. Esta IP, llamada **CRIP nProc**, es un nanoprocesador que se emplea principalmente para *command and control*, así como para implementación de bucles de control. Esto lo hace procesando microinstrucciones. Inicialmente, el sistema es capaz de reconocer y procesar un set de 32 instrucciones en ensamblador, entre las que se incluyen *branches*, copia de registros, operaciones lógicas y aritméticas, operaciones de un bit y accesos a memorias externas.

Durante el desarrollo de este proyecto, se ampliará este set de instrucciones, incluyendo operaciones direccionadas mediante punteros – con y sin *offset* –, comandos para poder acceder al *offset* y modificarlo, y desplazamientos de registros. De esta manera, se incluirán 13 nuevos comandos, obteniendo así un total de 45 instrucciones que el nanoprocesador reconoce. Además, se otorgará una gran versatilidad al sistema, añadiendo la posibilidad de escoger entre una memoria de instrucción interna o externa. También se podrá diferenciar entre el tamaño máximo que se puede utilizar de las memorias, y el tamaño real que se utiliza en una implementación concreta. Por último, se ampliará el tamaño de la memoria del nanoprocesador, cambiando de una capacidad de almacenamiento de $2^{16}$ registros a una de $2^{32}$.

## CRIP nProc

Como se ha comentado previamente, el sistema procesa microinstrucciones, es decir, comandos en ensamblador [2]. Esto lo hace a través de una estructura en pipeline [3]. Esta estructura se compone de las fases ilustradas en la Figura 1.


*Figura 1. Etapas del pipeline*

Estas etapas consisten en:

- **Lectura**: introducción de la microinstrucción en el sistema y preparación de la misma para la siguiente etapa. Esta etapa alberga la memoria de instrucción interna del sistema.
- **Decodificación**: separación de los distintos campos de la microinstrucción (código de operación y los dos registros con los que se desea operar). En eta etapa se sitúan las memorias de registros.
- **Ejecución**: ejecución de la operación indicada por la microinstrucción. Desde esta etapa se puede acceder a una memoria de almacenamiento externa, pudiendo así almacenar los valores de los registros fuera del sistema.

Es importante tener una mayor variedad de microinstrucciones, ya que la longitud de los programas que el sistema puede procesar es limitada. Así, se podrá reducir el tamaño de los mismos si existe la posibilidad de combinar operaciones que requieren varias microinstrucciones en una sola.

El nanoprocesador puede funcionar mediante dos implementaciones diferentes, en función de su velocidad: en la primera de ellas (implementación rápida), las lecturas de las memorias se pueden realizar en los flancos de bajada del reloj, de tal manera que en un ciclo completo de pipeline, es decir, en tres ciclos de reloj, se puede completar la ejecución de la mayoría de microinstrucciones (algunas microinstrucciones requerirán más ciclos de reloj para finalizar por razones ajenas a la velocidad de lectura). En el segundo caso (implementación lenta), las lecturas solamente se pueden realizar en los flancos de subida del reloj, necesitando un ciclo más que el caso anterior para poder completar la ejecución de una microinstrucción. Esto es debido a que se tarda un ciclo más en obtener el dato leído.

Una característica interesante de la IP nProc es que contiene un sistema de **autoverificación** que es capaz de corregir un registro cuando en éste se produce un error que provoca el cambio de valor de uno de sus bits. Además, en el caso de que el cambio de valor se produzca en dos bits de un registro, el sistema será capaz de detectarlo. En este caso se detendrá el sistema, ya que se considera que los datos están corruptos y no son fiables.

## Metodología

Los pasos a seguir para el desarrollo de este proyecto son los mostrados en la Figura 2.


*Figura 2. Flujo de desarrollo del proyecto*

El proceso de **diseño** implica el desarrollo del código VHDL de todas las características mencionadas (implementación de las nuevas instrucciones, ampliación del tamaño de la memoria, etc.). Para esto, ha sido necesario tener especial cuidado en la integración del nuevo código con el antiguo, de tal manera que no existan incompatibilidades entre ellos.

Por otro lado, el proceso de **verificación** se compone de dos partes principales. En la primera, se comprueba que los cambios realizados en el código VHDL tengan un correcto funcionamiento. Para ello, se ha creado un test en lenguaje ensamblador, que comprueba el resultado de todos los comandos al menos una vez, y generalmente combinando varias posibilidades del cada uno (por ejemplo: resultado positivo y negativo, con y sin *overflow*, etc.). Ese test tiene la capacidad de verificarse a sí mismo, de tal manera que si alguna de las instrucciones no devuelve el resultado esperado saltará a una rutina de error para que el usuario sea capaz de identificar con facilidad dónde se ha producido el fallo.

La segunda fase de la verificación consiste en el proceso de ***code coverage***. Esto es, la comprobación de que todas las líneas del código VHDL se hayan ejecutado al menos una vez durante el transcurso de los tests implementados. De esta manera, si todo el código se ha verificado, existe un riesgo menor de que el código diseñado sea el responsable de que se produzca algún error en la implementación final del mismo en una FPGA. Este proceso realiza diferentes tipos de comprobaciones para reducir en la medida de lo posible este riesgo, como, por ejemplo, son la comprobación de que todos los *branches* (if/else, case…) se hayan ejecutado, o que dentro de una expresión combinacional se hayan probado todas las combinaciones significativas.

Por último, el proceso de **síntesis** consiste en la generación automática de una representación de las puertas lógicas que serían necesarias para poder implementar el código VHDL creado en una FPGA real. De la misma manera, ofrece información del porcentaje de ocupación que tendrá la FPGA en función de las puertas lógicas disponibles, y una estimación del tiempo que tardará el ejecutarse el código.

En primer lugar, se han diseñado las nuevas instrucciones, ya que ofrecían una menor complejidad a la hora de integrar el nuevo código con el ya existente, aprovechando a la vez el proceso de implementación para comprender mejor el funcionamiento del nanoprocesador. Para ello, se ha partido de la implementación rápida (lectura de datos en ambos flancos de reloj), para, una vez funcionando el comando en cuestión de esta manera, poder hacer las modificaciones necesarias para que también lo haga en la implementación lenta.

A continuación, se ha procedido a diferenciar el tamaño máximo de las memorias (tanto la de instrucción como las memorias de registros) del tamaño real que se utiliza. Los tamaños máximos de las mismas están predefinidos por unas constantes (la memoria de instrucción es de 4k – 12 bits – y las de registros pueden guardar hasta 512 – 9 bits), mientras que el tamaño a utilizar lo decidirá el usuario a través de la asignación de un valor a unos genéricos. Como el tamaño de las memorias no varía, es necesario que, en caso de que se pida un acceso fuera de rango, esta petición se detecte y se asigne como un error de tal manera que el sistema se detenga, ya que no sería posible acceder a él. Por ejemplo, si el tamaño que se está utilizando de la memoria de instrucción es de 512 instrucciones (9 bits), en caso de que los bits 9, 10 o 11 de la dirección de la instrucción sea distinto de 0 se deberá de reportar como un error, ya que no corresponde con el tamaño determinado por el usuario.

Por otro lado, se ha creado la posibilidad de utilizar una memoria externa para almacenar las instrucciones del microcódigo cargado en la FPGA en lugar de la memoria interna del propio sistema. De esta manera, aunque los accesos a la memoria de instrucción serán más lentos que utilizando la memoria interna, se podrá aumentar el tamaño de la misma hasta el deseado, no teniendo así la limitación de 4096 microinstrucciones que se tiene al utilizar la memoria interna (con una capacidad de 4k). De igual manera que para la memoria interna, para la externa también será necesario realizar el control de accesos de memoria fuera de rango.

Por último, se ha procedido a aumentar el tamaño de la memoria del sistema. Así, se podrán manejar registros de 32 bits, cuando hasta el momento el tamaño de los mismos era de 16 bits. Se modificará también el tamaño de la memoria externa, pudiendo ahora almacenar un total de $2^{32}$ registros, en lugar de los $2^{16}$ que se podían guardar previamente.

## Resultados y conclusiones

Durante el desarrollo de este proyecto se ha modificado la IP nProc de Crisa para otorgarle diversas funciones, a mayores de las que tenía inicialmente. De esta manera, se han cumplido las expectativas iniciales de este proyecto.

Se ha proveído al sistema de un set de instrucciones más amplio, así como de una serie de características que ofrecen una mayor versatilidad en el uso del mismo, pudiendo adaptarse de manera más precisa a las necesidades de los usuarios.

Aunque las posibilidades del sistema son ahora considerablemente mayores que las disponibles antes de la realización del proyecto, este dispositivo todavía tiene un amplio abanico de posibilidades de mejora, como puede ser la implementación de las operaciones de la ALU en punto flotante, Este sistema, permite el uso de cifras decimales, además de cantidades de un valor mucho más grande que el modelo actual, obteniendo así un sistema de una mayor potencia.

## Referencias


[1] «Crisa,» [En línea]. Available: http://www.crisa.es/.

[2] «Microcódigo,» [En línea]. Available: http://zbc.uz.zgora.pl/Content/27955/Remigiusz_Wisniewski_Synthesis_of_CMCUs_for_Programmable_Devices.pdf.

[3] «PPU NG HKISeq Module Datasheet».

# EXTENSION OF THE INSTRUCTION SET OF THE IP "NPROC" OF AIRBUS-CRISA: DESIGN, VERIFICATION AND FPGA VALIDATION

**Author: de Miguel Fernández, Rita.**

Director: Pérez de la Fuente, Álvaro.

Associated Entity: Crisa (Computadoras, Redes e Ingeniería, S.A.U.).

## ABSTRACT OF THE PROJECT

### Introduction

Crisa (*Computadoras, Redes e Ingeniería, S.A.U.*) is a Spanish company dedicated to produce electronic equipment with aeroespacial applications [1]. When it is about this kind of applications, the device most used is an FPGA (Field Programmable Gate Array), because they are reprogrammable, and the development costs are lower than the associated to other devices appropriated to this function.

This project is centered in improve an IP (Intellectual Property) of this company, due to achieve a higher versatility in its use, offering a largest range of possibilities to future users od the system. This IP, named **CRIP nProc**, is a nanoprocessor that it is used principally in command and control applications and in control loops implementation. This is possible processing microinstrucions. Initially, the system is capable of recognize and process a set of 32 microinstructions in assembler code, among which are branches, register content copy, logical and arithmetical operations, bit operations and external memory accesses.

During the development of this project, this set of instructions will be enhanced, including pointer addressing operations – with and without an offset –, commands to be able to access the offset and modify it, and register shifting. In this way, there will be included 13 new commands, obtaining a total of 45 instructions that the nanoprocessor recognizes. Furthermore, a grand versatility of the system will be granted, adding the possibility of select between an internal or an external instruction memory. It also will be possible to distinguish between the maximum size of the memories and the real size that it is been used in a concrete implementation. Last but not least, the size of the memory of the nanoprocessor will be augmented, been able to store $2^{32}$ registers instead of the initial $2^{16}$.

## CRIP nProc

As it has been said before, the system processes microinstructions, that is, commands in assembler code [2]. The nanoprocessor does this through a pipeline structure [3]. This structure is composed by the stages illustrated in Figure 1.


*Figure 1. Pipeline stages*

These stages consist on:

- **Fetch**: introduction of the microinstruction into the system and preparation to the next stage. This stage includes the internal instruction memory of the system.
- **Decoding**: separation of the different microinstruction fields (operation code and the addresses of the registers that are going to be used). This stage holds the register file memories.
- **Execution**: execution of the operation indicated by the microinstruction. From this stage the accesses to the external memory can be made, being able to store and load the register values out of the system.

It is important to have a higher variety of microinstructions, since the size of the microcodes (sequence of microinstructions) that the system can process is limited. In this way, there exists the possibility of reducing the extent of the microcode combining operations that require several microinstructions into only one.

The nanoprocessor can work in two different implementations, in function of its speed: in the first one (fast implementation), the register can be read in the falling clock edges. Thanks to this, the ejecution of the majority of the microinstructions cam be completed in one complete cycle of the pipeline structure, this is, in three clock cycles. Some microinstructions will require more clock cycles due to reasons unrelated to the implementation mode. In the second case (slow implementation), the reading can be made only in the rising clock edges, needing one more clock cycle to finish the execution of a microinstruction. This is because there is needed one more cycle to be able to obtain the read data.

An interesting feature of the IP nProc is that it contains an **autoverification** system that is able to correct the value of a register whenever it is produced a single error in it, that is, when, for external reasons, a bit of a register changes its value. Moreover, in the case that there is a double error (2 bits of a register have the wrong value), the system will be able to detect it too. In this case the system will be stopped, due to it is considered that the data is corrupted, and the information is trustless.

## Methodology

The steps to follow in the development of this project are the shown in Figure 2.


*Figure 2. Development flow of the project*

The **design** process involves the development of the VHDL code of all the features mentioned above (implementation of new instructions, increasing of the memory size, etc.). To make this, it was necessary to pay special attention to the integration of the new code with the old one, in a way that there were not incompatibilities between them.

On the other hand, the **verification** process is composed by two principal phases. In the first one, there is checked that the changes made in the VHDL code have a correct performance. For this reason, there was created a test in assembler code, that checks the result of all the commands at least once, and, generally, combining several possibilities of each one (for example: positive and negative result, result with and without overflow, etc.). This test has the capability of verify itself, in a way that, if any of the instructions do not return the expected result, it will be a branch to an error routine. Because of this, the user will be able to identify in an easy way where the error was produced.

The second phase of the verification consists in a process of **code coverage**. This means, the verification that all the VHDL code lines are executed at least once during the course of the implemented tests. In this manner, if every line of the code has been tests, there exists a lower risk that the designed code will be the responsible of failure in the final implementation of the design in a FPGA. This process performs different types of verification in order to reduce this risk as far as possible, as, for example, the check that all branches (if/else, case…) were executed, or that inside a combinational expression were tested all the significative combinations.

By last, the **synthesis** process consists in the automatic generation of a representation of the logic gates that will be necessary to be able to implement the created VHDL code in a real FPGA. In the same way, it offers information of the occupation percentage that will have the FPGA in function of the available logic gates in it, and an estimation of the time that the code needs to be executed.

In the first place, the new instructions were designed, since they offered les complexity when integrating the new code with the already existing one, taking advantage of this to understand in a better way the nanoprocessor behavior. For this purpose, the fast implementation (data reading in both, rising and falling, clock edges) of these instructions was designed. When the command was working using this implementation, the needed modifications were applied to make the system able to operate also in slow implementation.

Then, it has been proceeded to differentiate between the maximum size of the memories (both instruction memory and register file memories) and the real size that is being used. The maximum sizes are predefined by constants (the instruction memory has a capacitance of 4k – addressed by 12 bits – and the register files can address 512 registers – 9 bits), and the real size will be decided by the user through the assignment of a value to generics. Since the memory size does not vary, it is necessary to check that, in case that an out of range access is requested, this demand will be detected and assigned as an error in a way that the system will be stopped, due to there will be no manner to access to the requested data. For example, if the instruction memory is being used with a capacitance of 512 instructions (addressed by 9 bits), in the case that the bits 9, 10 or 11 of the address of the instruction is other that 0, this access should be reported as an error, since there is no match using the size determined by the user.

On the other hand, there was created the possibility of using an external memory to store the microcode instructions instead of the internal memory that the system has for this purpose. In this way, although the accesses to the instruction memory will be slower than the ones using the internal memory, there exists the option to increase the size of the instruction memory, not having now the limitation of 4096 microinstructions that happens using the internal instruction memory (with a capacitance of 4k instructions). In the same way as with the internal memory, it is needed to implement the out of range access control.

Last but not least, it has been proceeded to enlarge the size of the memory of the system. In this way, it is possible to manage registers with a width of 32 bits (at the moment, the size of the registers was 16 bits long). It also was modified the external memory size, being able to store now up to $2^{32}$ registers instead of the $2^{16}$ that the system could hold previously.

## Results and conclusions

During the development of the project, the Crisa IP nProc has been modified to grant it with some new characteristics, adding them to the ones that it already had. In this way, the initial expectations of the project were accomplished.

The system was provided with a more complete set of instructions, as well as a series of features that offer a higher versatility at the time of using it. With these properties, the system will be able to adapt itself in a more precise way to the needs of the users.

Even though the possibilities of the system are now considerably higher than the available before the elaboration of the project, this device still has a huge range of possibilities of improvement, as it is the implementation of the ALU operations in floating point. This system will allow the use of decimal numbers, besides quantities with much bigger values than the actual model, obtaining in this way a system with a higher power that the actual one.

## References


[1] «Crisa,» [Online]. Available: http://www.crisa.es/.

[2] «Microcódigo,» [Online]. Available: http://zbc.uz.zgora.pl/Content/27955/Remigiusz_Wisniewski_Synthesis_of_CMCUs_for_Programmable_Devices.pdf.

[3] «PPU NG HKISeq Module Datasheet».

# *Índice de la memoria*

*Parte I Memoria* ..... *7*

*Capítulo 1 Introducción* ..... *9*

**1.1 Estudio de los trabajos existentes / tecnologías existentes** ..... **11**

**1.2 Motivación del proyecto** ..... **13**

**1.3 Objetivos** ..... **13**

**1.4 Metodología** ..... **14**

**1.5 Herramientas empleadas** ..... **15**

*Capítulo 2 Estado inicial del proyecto* ..... *17*

**2.1 Descripción del diseño** ..... **17**

**2.2 Instrucciones existentes** ..... **20**

**2.3 Diagramas de bloques** ..... **21**

**2.4 Verificación** ..... **26**

*Capítulo 3 Implementación de nuevas instrucciones* ..... *37*

**3.1 Punteros** ..... **38**

**3.2 Punteros con offset** ..... **41**

**3.3 Shiftado** ..... **56**

*Capítulo 4 Tamaño real vs. tamaño máximo de memorias* ..... *59*

*Capítulo 5 Memoria de instrucción externa* ..... *63*

*Capítulo 6 Ampliación del tamaño de la memoria* ..... *69*

*Capítulo 7 Conclusiones y futuros desarrollos* ..... *71*

*Capítulo 8 Referencias ..... 75*

*Parte II Presupuesto ..... 77*

*Capítulo 1 Mediciones ..... 79*

**1.1 Material ..... 79**

**1.2 Ingeniería ..... 80**

*Capítulo 2 Precios unitarios ..... 81*

**2.1 Material ..... 81**

**2.2 Ingeniería ..... 82**

*Capítulo 3 Presupuestos parciales ..... 83*

**3.1 Material ..... 83**

**3.2 Ingeniería ..... 84**

*Capítulo 4 Presupuesto general ..... 85*

*Parte III Manual de usuario ..... 87*

*Capítulo 1 Configuración del dispositivo ..... 89*

**1.1 Señal enable ..... 89**

**1.2 Señal ext_jump (pulso) ..... 90**

**1.3 Bus dbg_out ..... 90**

**1.4 Señales im_*(_ext), rf_* and dm_* ..... 91**

*Capítulo 2 Flujo de diseño del microcódigo ..... 93*

**2.1 Estructura ..... 93**

**2.2 Ensamblaje ..... 94**

*Capítulo 3 Operación nominal ..... 95*

*Capítulo 4 Requerimientos de arquitectura externos ..... 97*

**4.1 Registro externo de los flags de error ..... 97**

**4.2 Control de scrubbing de las memorias internas ..... 97**

***Parte IV*** ***Diagramas de Bloques..... 99***

***Parte V*** ***Comandos ..... 113***

# *Índice de figuras*

Figura 1. Diagrama funcional del módulo PPU-NG [3] ..... 10
Figura 2. Flujo de diseño del microcódigo ..... 11
Figura 3. Interfaz I/O del nP – inicial ..... 17
Figura 4. Simulación - implementación rápida ..... 29
Figura 5. Simulación - implementación lenta ..... 30
Figura 6. Code coverage ..... 34
Figura 7. Code coverage con exclusiones ..... 35
Figura 8. MVR y MVW - implementación rápida ..... 40
Figura 9. MVR y MVW - implementación lenta ..... 41
Figura 10. MVRO y MVWO - implementación rápida ..... 44
Figura 11. MVRO y MVWO - implementación lenta ..... 45
Figura 12. LRL - implementación rápida ..... 46
Figura 13. LRL - implementación lenta ..... 46
Figura 14. SRL - implementación rápida ..... 48
Figura 15. SRL - implementación lenta ..... 48
Figura 16. SRLI - implementación rápida ..... 49
Figura 17. SRLI - implementación lenta ..... 50
Figura 18. ADDRL - implementación rápida ..... 51
Figura 19. ADDRL - implementación lenta ..... 52
Figura 20. ADDRLI - implementación rápida ..... 53
Figura 21. ADDRLI - implementación lenta ..... 53
Figura 22. SUBRL - implementación rápida ..... 54

Figura 23. SUBRL - implementación lenta ..... 55
Figura 24. SUBRLI - implementación rápida ..... 56
Figura 25. SUBRLI - implementación lenta ..... 56
Figura 26. SHLI y SHRI – implementación rápida ..... 58
Figura 27. Memoria de instrucción interna – implementación rápida ..... 66
Figura 28. Memoria de instrucción externa - implementación lenta ..... 67
Figura 29. Interfaz I/O del nP – final ..... 72
Figura 30. Ciclo de scrubbing con errores de EDAC simples ..... 98
Figura 31. crip_np - diagrama de bloques ..... 101
Figura 32. i_ifetch - diagrama de bloques ..... 102
Figura 33.i_imem – diagrama de bloques ..... 103
Figura 34. i_imem_ctrl - diagrama de bloques ..... 104
Figura 35. i_idec - diagrama de bloques ..... 105
Figura 36. i_reg - diagrama de bloques ..... 106
Figura 37. i_reg_ctrl - diagrama de bloques ..... 107
Figura 38. i_exec - diagrama de bloques ..... 108
Figura 39. i_alu - diagrama de bloques ..... 109
Figura 40. Testbench inicial - diagrama de bloques ..... 110
Figura 41. Testbench final - diagrama de bloques ..... 111

# *Índice de tablas*

Tabla 1. Entradas y salidas nP inicial ..... 19
Tabla 2. Genéricos del nP ..... 20
Tabla 3. Codificación de las microinstrucciones ..... 21
Tabla 4. Señales simulación ..... 29
Tabla 5. MVR y MVW - valores de los registros ..... 40
Tabla 6. MVRO y MVWO - valores de los registros ..... 44
Tabla 7. SRL - valores de los registros ..... 47
Tabla 8. ADDRL - valores de los registros ..... 51
Tabla 9. SUBRL - valores de los registros ..... 54
Tabla 10. Señales simulación - memoria de instrucción externa ..... 68
Tabla 11. Mediciones. Material ..... 79
Tabla 12. Mediciones. Ingeniería ..... 80
Tabla 13. Precios unitarios. Material ..... 81
Tabla 14. Precios unitarios. Ingeniería ..... 82
Tabla 15. Presupuestos parciales. Material ..... 83
Tabla 16. Presupuestos parciales. Ingeniería ..... 84
Tabla 17. Presupuesto general ..... 85
Tabla 18. Genéricos ..... 89
Tabla 19. dbg_out - señales concatenadas ..... 91
Tabla 20. Comandos ..... 117

# *Parte I MEMORIA*

# Capítulo 1 INTRODUCCIÓN

Crisa (Computadoras, Redes e Ingeniería, S.A.U.) es una empresa española fundada en 1985 para diseñar y producir equipos electrónicos y software para aplicaciones espaciales: satélites, lanzadores, infraestructuras orbitales y vehículos de transporte espaciales. También se realizan proyectos de ingeniería para estaciones en tierra.

Crisa está completamente integrada en Airbus Defence and Space, la empresa número uno en Europa en materia de defensa y espacio. La empresa ha jugado un papel importante en muchas misiones científicas y de observación de la Tierra de la ESA, desarrollando procesadores de carga útil y controladores software y hardware: unidades de procesamiento de datos, unidades de control, acondicionamiento de potencia y unidades de distribución, y equipos de bancos de pruebas. Estos proyectos se utilizan para monitorización de la Tierra, observación de eventos naturales, así como investigación científica del Universo. Crisa también trabaja en satélites de telecomunicaciones desarrollando equipos para plataformas de satélites de comunicaciones [1].

La IP *nProc* es una máquina microprogramada que se emplea, entre otras cosas, para *command and control* e implementación de bucles de control. En esta IP las microinstrucciones se almacenan en una memoria síncrona interna llamada memoria de instrucción [2].

**Microinstrucción**: cada una de las operaciones o acciones que componen un microprograma.

**Microprograma o microcódigo**: secuencia de microinstrucciones.

La ejecución de este microcódigo tiene una estructura en pipeline, y consta de las siguientes fases:

- Lectura de la microinstrucción
- Decodificación de los campos de la microinstrucción

- Ejecución de la operación indicada por la microinstrucción

Un ejemplo de aplicación de la IP *nProc* se encuentra en el módulo PPU-NG [3]. Las funciones de este módulo son las siguientes:

- Comunicaciones con la plataforma a través de una interfaz 1553B.
- Monitorización y control del calentamiento, ignición y disparo (*firing up*) de hasta cuatro propulsores.
- Comunicaciones con hasta cuatro módulos *Anode* a través de un puesto serie CAN a 125 Kbps (bitrate nominal).
- Monitorización y control de dos fuentes de alimentación.
- Monitorización y control de módulo *Ignitor*.
- Telemetría y control del módulo PPU-NG y de los módulos *Anode*.

El diagrama funcional del módulo PPU-NG se muestra en la Figura 1. La IP *nProc* se sitúa en el bloque *"Thruster Controller"* de dicho diagrama.

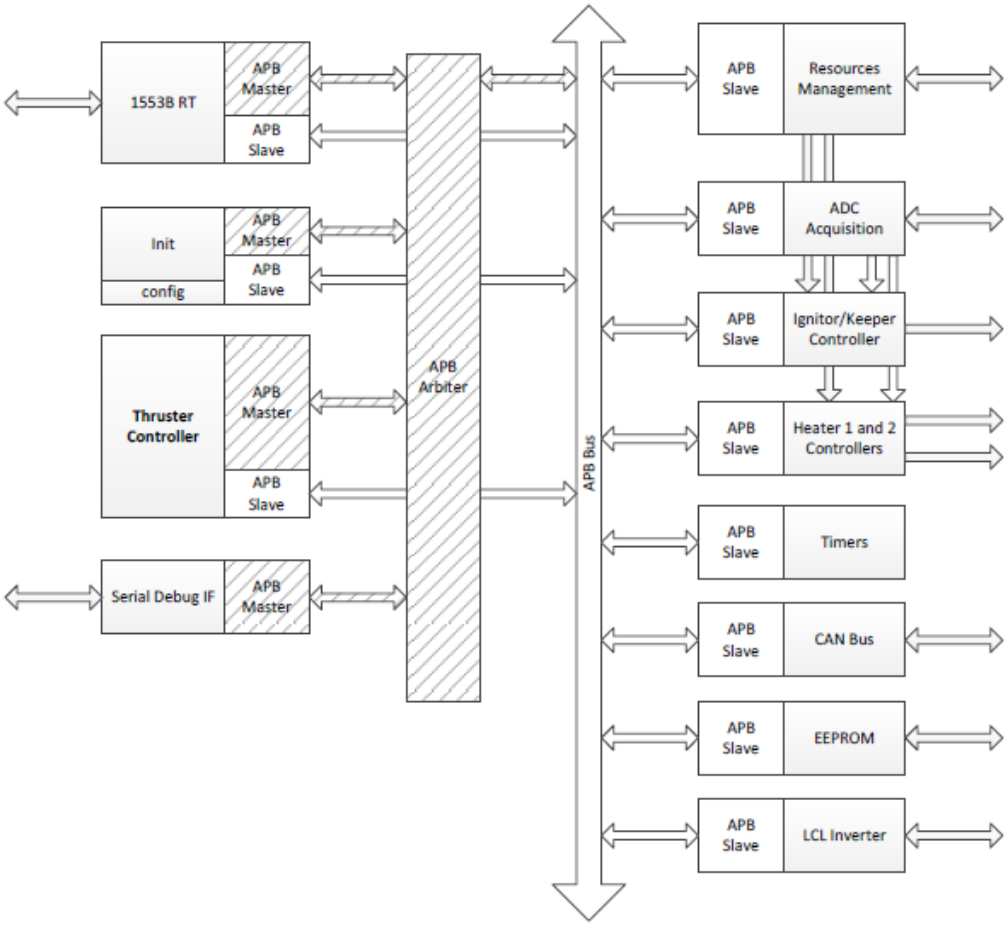


*Figura 1. Diagrama funcional del módulo PPU-NG [3]*

## *1.1 ESTUDIO DE LOS TRABAJOS EXISTENTES / TECNOLOGÍAS EXISTENTES*

El flujo de diseño del microcódigo está compuesto por tres fases, mostradas en la Figura 2.

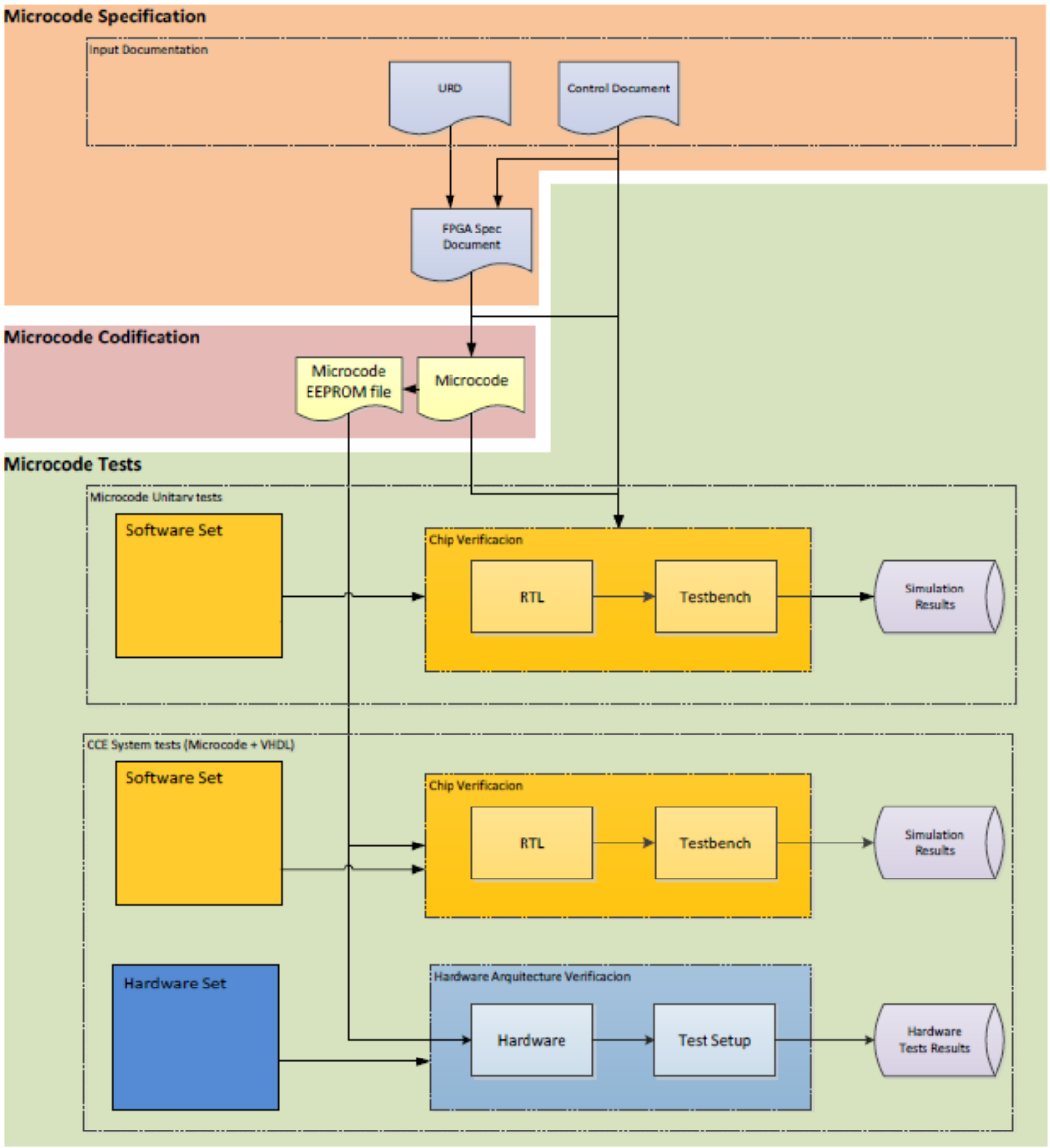


*Figura 2. Flujo de diseño del microcódigo*

### 1.1.1 ESPECIFICACIÓN DEL MICROCÓDIGO

Durante esta etapa se identifican los requisitos del sistema; las especificaciones del microcódigo se corresponderán con las de la FPGA, puesto que el microcódigo es considerado una parte de la FPGA. Aquí, las diferentes funcionalidades a ser implementadas por el microcódigo se subdividen en tareas más simples.

Realizar en este punto la división de tareas permite definir la arquitectura del sistema y dividir la carga de trabajo de codificación y verificación.

### 1.1.2 CODIFICACIÓN DEL MICROCÓDIGO

El microcódigo es codificado utilizando un lenguaje tipo ensamblador.

Las tareas de codificación están incluidas en las funciones divididas en la etapa anterior para que puedan ser codificadas en una interfaz bien definida y testeadas poco a poco, de tal manera que haya menos elementos implicados en el proceso de depuración.

Una vez el microcódigo ha sido codificado, se generan los datos de la EEPROM asociados. Estos datos se utilizarán durante las simulaciones del sistema de la FPGA así como en las pruebas sobre el prototipo (es la misma EEPROM que se incluirá en el dispositivo final).

### 1.1.3 TEST DEL MICROCÓDIGO

Para las simulaciones del sistema, se utiliza Questa Sim. Las simulaciones realizadas contienen un modelo bastante preciso del sistema real, incluyendo modelos de las memorias externas de la FPGA, links de comunicación, el modelo de la cadena de adquisición… Estos modelos se gestionan usando TCL, y los resultados de la simulación se comprueban automáticamente.

Para asegurar que el microcódigo se ejecuta correctamente, las simulaciones del sistema se ejecutan con el microcódigo real. Esto asegura que el microcódigo y el VHDL se simulen en las condiciones más realistas posibles, es decir, que, durante las simulaciones, el sistema testeado es lo más parecido posible al que se incluirá en el modelo de vuelo.

## *1.2 MOTIVACIÓN DEL PROYECTO*

El motivo de realización de este proyecto es evolucionar la IP *nProc* para su implementación en futuros proyectos, de tal manera que les proporcione una mayor flexibilidad en la codificación que la existente hasta el momento mediante la ampliación de instrucciones, como serán el direccionamiento a través de punteros y las instrucciones de shiftado.

Además, es igualmente importante ofrecer una mayor versatilidad en cuanto a las posibilidades del nanoprocesador. Para ello, se ofrecerá la posibilidad de utilizar la memoria de instrucción como una memoria interna del sistema o como una externa al mismo. También se podrá diferenciar entre el tamaño máximo de las memorias a utilizar y el tamaño real utilizado, algo que hasta ahora no se ofrece en el dispositivo.

## *1.3 OBJETIVOS*

Los objetivos del proyecto consisten en añadir las siguientes funcionalidades al nanoprocesador:

- **Punteros**: Instrucciones de lectura y escritura indexada por registros:
  - mvr RT, RS → RT = R(RS)
  - mvw RT, RS → R(RT) = RS
- **Punteros con offset**.
  - Instrucciones de lectura y escritura indexada por registros con offset.

  - mvro RT, RS → RT = R(RS + RL)
  - mvwo RT, RS → R(RT + RL) = RS
- Instrucciones para acceder al registro que almacena este offset (RL: Link Register)
  - lrl RT → RT = RL
  - srl RS → RL = RS
  - srli imm9 → RL = imm9
- Instrucciones para operar sobre el RL.
  - addrl RS → RL = RL + RS, update ZNV
  - addrli imm9 → RL = RL + signext(imm9), update ZNV
  - subrl RS → RL = RL – RS, update ZNV
  - subrli imm9 → RL = RL – signext(imm9) update ZNV

- **Instrucciones de shiftado**.
  - shli RT, imm4→ RT = (RT << imm4)
  - shri RT, imm4→ RT = (RT >> imm4)
- Diferenciación del **tamaño de memoria máximo y tamaño de memoria real**, tanto en la memoria de instrucción como en la memoria de registros.
- Posibilidad de utilización de la **memoria de instrucción** como **interna** (ubicada dentro del nanoprocesador) **o externa** (como un periférico al mismo).
- **Ampliación del tamaño de la memoria** del sistema, pudiendo utilizar registros de 32 bits en lugar de 16 bits.

## *1.4 METODOLOGÍA*

Las tareas a realizar durante el transcurso del proyecto son las siguientes:

1. Comprensión del funcionamiento de la IP *nProc*
2. Modificación del código VHDL
3. Verificación por simulación
4. *Code coverage*
5. Síntesis; chequear errores
6. Redacción de la memoria

## *1.5 HERRAMIENTAS EMPLEADAS*

### 1.5.1 HDL DESIGNER

Es un programa que combina una gran capacidad de análisis, editores y una completa administración de proyectos y flujo para ofrecer un ambiente agradable de diseño HDL [4]. Algunas de sus principales características son:

- Administración de diseños complejos en VHDL, Verilog y SystemVerilog.
- Visualización HDL y herramientas de diseño interactivas.
- Generación automática de documentación e informes.
- Sistema inteligente de depuración y análisis.
- Creación eficiente de diseños en RTL utilizando texto, tablas y gráficos.

### 1.5.2 MENTOR QUESTA SIM

Questa Sim es una herramienta de verificación y simulación para VHDL, Verilog, SystemVerilog, SystemC y diseños con leguajes mixtos. Ofrece numerosas herramientas para depuración y análisis [5].

### 1.5.3 SYNPLIFY PRO

Es un software de síntesis encargado de producir diseños para una FPGA de alto rendimiento y económicos. Soporta las últimas versiones de VHDL y lenguaje Verilog, así como arquitecturas de FPGA de diversos proveedores (Altera, Achronix, Lattice…) El programa utiliza una única interfaz, fácil de utilizar, y tiene la habilidad de realizar síntesis incremental y un análisis de código HDL intuitivo [6].

# Capítulo 2 ESTADO INICIAL DEL PROYECTO

En este apartado se detallan las características iniciales del nanoprocesador. Para ello, se explican las señales de entrada y de salida del nivel superior, así como los genéricos a definir para poder utilizarlo. También se comentarán los diferentes comandos que el dispositivo es capaz de identificar, así como la estructura interna del mismo. Por último, se explicarán los diferentes métodos de verificación que se utilizan para comprobar el correcto funcionamiento del mismo.

## *2.1 DESCRIPCIÓN DEL DISEÑO*

### 2.1.1 ENTRADAS Y SALIDAS

La interfaz de entradas y salidas del nanoprocesador está ilustrada en la Figura 3.

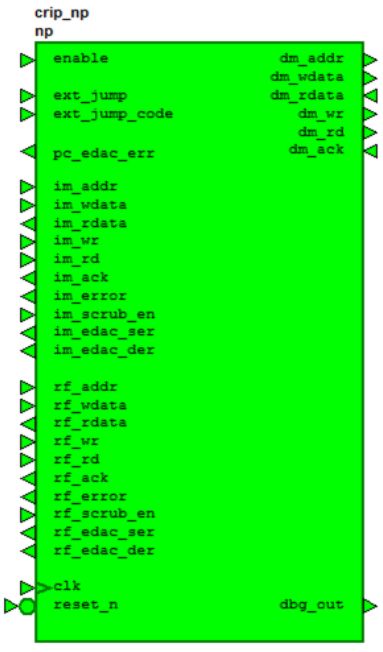


*Figura 3. Interfaz I/O del nP – inicial*

Todas estas señales se describen en la Tabla 1.

| **SEÑALES GENERALES** | | |
|---|---|---|
| **Nombre de la señal** | **Tipo** | **Descripción** |
| ***enable*** | Entrada | Enable |
| ***clk*** | Entrada | Reloj del sistema |
| ***reset_n*** | Entrada | Reset asíncrono, active a nivel bajo |
| ***ext_jump*** | Entrada | Comando de salto externo |
| ***ext_jump_code(1:0)*** | Entrada | Código de salto externo |
| ***pc_edac_err(1:0)*** | Salida | Error en la EDAC en elPC |
| ***dbg_out(46:0)*** | Salida | Señales de depuración concatenadas |
| **MEMORIA DE DATOS EXTERNA** | | |
| **Nombre de la señal** | **Tipo** | **Descripción** |
| ***dm_addr(15:0)*** | Salida | Bus de direcciones |
| ***dm_wdata(15:0)*** | Salida | Bus de datos de escritura |
| ***dm_rdata(15:0)*** | Entrada | Bus de datos de lectura |
| ***dm_wr*** | Salida | Comando de escritura |
| ***dm_rd*** | Salida | Comando de lectura |
| ***dm_ack*** | Entrada | Flag ACK |
| **MEMORIA DE INSTRUCCIÓN** | | |
| **Nombre de la señal** | **Tipo** | **Descripción** |
| ***im_addr(11:0)*** | Entrada | Bus de direcciones |
| ***im_wdata(23:0)*** | Entrada | Bus de datos de escritura |
| ***im_rdata(23:0)*** | Salida | Bus de datos de lectura |
| ***im_wr*** | Entrada | Comando de escritura |
| ***im_rd*** | Entrada | Comando de lectura |
| ***im_ack*** | Salida | Flag ACK |
| ***im_error*** | Salida | Flag de error |

| | | |
|---|---|---|
| ***im_scrub_en*** | Entrada | Enable de scrubbing |
| ***im_edac_ser*** | Salida | Flag de corrección de error simple en la EDAC |
| ***im_edac_der*** | Salida | Flag de detección de error doble en la EDAC |
| **MEMORIA DE REGISTROS** | | |
| **Nombre de la señal** | **Tipo** | **Descripción** |
| ***rf_addr(8:0)*** | Entrada | Bus de direcciones |
| ***rf_wdata(15:0)*** | Entrada | Bus de datos de escritura |
| ***rf_rdata(15:0)*** | Salida | Bus de datos de lectura |
| ***rf_wr*** | Entrada | Comando de escritura |
| ***rf_rd*** | Entrada | Comando de lectura |
| ***rf_ack*** | Salida | Flag ACK |
| ***rf_error*** | Salida | Flag de error |
| ***rf_scrub_en*** | Entrada | Enable de scrubbing |
| ***rf_edac_ser*** | Salida | Flag de corrección de error simple en la EDAC |
| ***rf_edac_der*** | Salida | Flag de detección de error doble en la EDAC |

*Tabla 1. Entradas y salidas nP inicial*

### 2.1.2 GENÉRICOS

Además, es necesario asignar un valor a una serie de genéricos asociados al bloque para poder trabajar con el nanoprocesador. Estos genéricos son los listados en la Tabla 2.

| GENÉRICO | TIPO | DESCRIPCIÓN |
|---|---|---|
| ***g_fast_exec*** | Booleana | Si es TRUE: se usa el flanco de bajada del reloj para leer las memorias internas (doble flanco) |
| ***g_ext_jump_code_w*** | Entero | Tamaño del código de salto externo |
| ***g_pc_edac_err_w*** | Entero | Tamaño del error en la EDAC en el PC |
| ***g_inst_edac_w*** | Entero | Redundancia Hamming para la EDAC-16 de la ESA |

| | | |
|---|---|---|
| ***g_data_w*** | Entero | Tamaño de buses de datos |

*Tabla 2. Genéricos del nP*

## *2.2 INSTRUCCIONES EXISTENTES*

En su estado inicial, el nanoprocesador ejecuta los tipos de microinstrucciones explicados a continuación:

- **Salto**: saltos incondicionales y condicionales a otra parte del microcódigo (br, bi, be, bne, bgt, bge).
- **Acceso a memoria externa**: escribir y leer palabras en la memoria externa (lw, lwi, sw, swi).
- **Mover**: leer y escribir registros (mv, mvi, mva).
- **Operaciones aritméticas**: suma, resta y comparación de dos elementos (add, addi, sub, subi, cmp, cmpi).
- **Operaciones lógicas**: operaciones lógicas entre dos elementos (and, andi, or, ori, xor, xori).
- **Operaciones de un bit**: set, reset y test del valor de un bit (btest, btesti, bset, bseti, brst, brsti).

Todas las microinstrucciones, tanto las iniciales como las creadas durante el desarrollo del proyecto, están enumeradas y explicadas en la Parte V, Tabla 20.

Todas las microinstrucciones se codifican siguiendo alguno de los patrones mostrados en la Tabla 3.

| BITS | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Código de operación | | | | | | RT | | | | | | | | | RS / imm9 | | | | | | | | |
| | | | | | | RT | | | | | | | | | - | | | | | | | | |
| | | | | | | - | | | | | | | | | RS / imm9 | | | | | | | | |

| | | | |
|---|---|---|---|
| | RT | imm12 | |
| | - | imm12 | |
| | RT | - | imm4 |

*Tabla 3. Codificación de las microinstrucciones*

- **Bits (23:18)**: siempre el código de operación.
- **Bits (17:9)**: generalmente corresponden con la dirección del registro RT; ocasionalmente, la dirección del registro RT es más pequeña (cuando el segundo operando es un inmediato de 12 bits), o incluso no necesaria (por ejemplo, en una instrucción de salto).
- **Bits (8:0)**: generalmente corresponden con la dirección del registro RS o un inmediato de 9 bits; ocasionalmente, estas posiciones corresponden a un inmediato de diferente longitud (4 o 12 bits).

Siendo:

- **RT, RS**: direcciones de registros
- **imm4, imm9, imm12**: inmediatos de longitudes 4, 9 y 12 bits respectivamente.

## *2.3 DIAGRAMAS DE BLOQUES*

En este apartado se explicará el contenido de cada uno de los bloques y sub-bloques del sistema. Todos los diagramas de bloques se encuentran adjuntos en la Parte IV.

Es importante señalar que las imágenes adjuntas corresponden con el estado final del proyecto, por lo que algunos de los bloques mostrados en las mismas no formaban parte del nanoprocesador al inicio. Estos bloques no se explicarán en este apartado, sino que se hará más adelante, en el momento en el que se implemente cada uno de ellos.

El nivel superior, **crip_np** (Figura 31), consiste en tres bloques principales, cada uno de los cuales representa una de las etapas del pipeline:

- **i_ifetch**: primera etapa del pipeline. Esta entidad introduce las instrucciones entrantes en el sistema. En ella se incluye la memoria de instrucción.
- **i_idec**: esta entidad decodifica las instrucciones en sus diferentes campos y las prepara para ser ejecutadas. En ella se incluye la memoria de registro.
- **i_iexec**: esta entidad ejecuta las instrucciones.

### 2.3.1 ARQUITECTURA INTERNA DE I_IFETCH

El diagrama de bloques correspondiente a **i_ifetch** se encuentra en la Figura 32.

- **i_pc**: esta entidad gestiona el contador del programa (PC).
- **i_imem**: esta entidad gestiona la memoria de instrucción.
- **p_inst**: este bloque registra las señales de control que serán utilizadas en la siguiente etapa del pipeline (**i_idec**).

#### *2.3.1.1 Arquitectura interna de i_imem*

La Figura 33 muestra el diagrama de bloques de **i_imem**, donde:

- **i_imem_ctrl**: esta entidad controla la memoria de instrucción. Multiplexa la petición de introducir una nueva instrucción en el nanoprocesador con la interfaz de memoria de instrucción externa. La primera se utiliza para leer las instrucciones de la memoria **i_imem**, y la segunda para rellenar la memoria de instrucción **i_imem**. Para los accesos externos, detecta el flanco de subida de los comandos de escritura y genera el flag ACK.
- **p_scrub**: este bloque controla el multiplexor ara el scrubbing.
- **p_edac**: este bloque gestiona la EDAC de la memoria de instrucción.
- **i_imem**: esta entidad es un modelo de memoria de doble puerto paramétrica para inferencias automáticas en síntesis en FPGAs de Actel.

Uno de los puertos es de sólo escritura y el otro de sólo lectura, con diferentes relojes, señales de enable y buses de dirección.

- **p_ro_clk**: este bloque selecciona el valor efectivo del reloj de lectura para el modelo de la memoria de instrucción. Este valor efectivo irá relacionado con el genérico *g_int_inst_mem*.

#### 2.3.1.1.1 Arquitectura interna de i_imem_ctrl

La arquitectura de este bloque se puede observar en la Figura 34, siendo:

- **mem_write_iface**: este bloque detecta el flanco del comando de escritura dado por la interfaz de memoria externa.
- **mem_read_iface**: este bloque detecta el flanco del comando de lectura dado por la interfaz de memoria externa.
- **inst_buff**: este bloque contiene un buffer del dato de lectura.
- **ack**: este bloque genera el flag ACK para la interfaz de memoria externa y registra el dato de lectura.

### 2.3.2 ARQUITECTURA INTERNA DE I_IDEC

La segunda etapa del pipeline, **i_idec**, se muestra en la Figura 35.

- **p_idec**: este bloque registra la instrucción proveniente de la etapa anterior (**i_ifetch**) para la siguiente etapa del pipeline (**i_iexec**). También separa los campos de la instrucción en los campos necesarios para su uso posterior.
- **i_reg**: esta entidad gestiona la memoria de registros del decodificador de instrucciones.
- **rs_addr_selection**: este bloque no figura al principio del proyecto.
- **add_rl**: este bloque no figura al principio del proyecto.
- **ph2_reg**: este bloque no figura al principio del proyecto.
- **rs_selection**: este bloque selecciona el valor del registro RS (registro, posición de memoria, inmediato…) necesario para cada instrucción.

- **hazards**: este bloque detecta cuando una instrucción necesita usar un valor que está siendo actualizado (en la etapa de pipeline **i_iexec**).
- **p_idec_reg**: este bloque registra las señales de control que serán usadas en la siguiente etapa del pipeline (**i_iexec**).

#### *2.3.2.1 Arquitectura interna de i_reg*

La arquitectura interna de **i_reg** se puede observar en la Figura 36, donde:

- **flags_gen**: este bloque genera los flags de error de la EDAC asociada a la memoria de registros.
- **i_reg_ctrl**: esta entidad controla la memoria de registros del decodificador de instrucciones. Multiplexa la petición de los registros RT y RS con la interfaz de memoria externa. La primera se utiliza para leer o escribir las memorias **i_rt_mem** (registro RT) e **i_rs_mem** (registro RS), y la segunda para rellenar la memoria de registro **i_rt_mem** e **i_rs_mem**. Para los accesos externos, detecta el flanco de subida de los comandos de escritura y genera el flag ACK.
- **p_edac_rt**: este bloque gestiona la EDAC asociada al banco de registros RT.
- **i_rt_mem**: esta entidad es un modelo de memoria de doble puerto paramétrica para inferencias automáticas en síntesis en FPGAs de Actel. Uno de los puertos es de sólo escritura y el otro de sólo lectura, con diferentes relojes, señales de enable y buses de dirección. Almacena los valores de los registros RT.
- **p_edac_rs**: este bloque gestiona la EDAC asociada al banco de registros RS.
- **i_rs_mem**: esta entidad es un modelo de memoria de doble puerto paramétrica para inferencias automáticas en síntesis en FPGAs de Actel. Uno de los puertos es de sólo escritura y el otro de sólo lectura, con diferentes relojes, señales de enable y buses de dirección. Almacena los valores de los registros RS.

- **p_ro_clk**: este bloque selecciona el valor efectivo del reloj de lectura para los modelos de las memorias de registro. Este valor efectivo irá relacionado con el genérico *g_int_inst_mem*.

#### 2.3.2.1.1 Arquitectura interna de i_reg_ctrl

El bloque **i_reg_ctrl** se puede ver en la Figura 37.

- **write**: este bloque detecta el flag del comando de escritura de la interfaz de memoria externa.
- **ack**: este bloque genera el ACK para la interfaz de memoria externa y registra el dato de lectura.
- **read**: este bloque detecta el flag del comando de lectura de la interfaz de memoria externa.
- **p_scrub**: este bloque gestiona el *scrubbing* asociado a la memoria de registro. Se accede a los dos bancos de registros de manera concurrente.

### 2.3.3 ARQUITECTURA INTERNA DE I_EXEC

La arquitectura interna de **i_exec** se muestra en la Figura 38, siendo sus bloques internos:

- **b_hazards**: este bloque detecta los *hazards* (uso de un valor que se está actualizando) en saltos.
- **r_hazards**: este bloque detecta los *hazards* (uso de un valor que se está actualizando) en RT y RS.
- **branch**: este bloque procesa las instrucciones de saltos.
- **data_memory**: este bloque accede a la memoria externa.
- **i_alu**: esta entidad las operaciones de la ALU.
- **l_s_rl**: este bloque no figura al principio del proyecto.
- **shift**: este bloque no figura al principio del proyecto.
- **writeback**: este bloque registra las señales de salida del proceso completo.

### *2.3.3.1 Arquitectura interna de i_alu*

El diagrama de bloques de **i_alu** se puede observar en la Figura 39.

- **sel_op_a**: este bloque no figura al principio del proyecto.
- **sel_op_b**: este bloque no figura al principio del proyecto.
- **i_addsub**: esta entidad ejecuta las operaciones matemáticas de suma o resta, dependiendo de un flag.
- **math**: este bloque prepara el flag de resta para la entidad **i_addsub**, además de detectar el flag 'Z' de las operaciones matemáticas.
- **logic**: este bloque ejecuta las operaciones lógicas, dependiendo del código de la operación que se esté procesando.
- **bit**: este bloque ejecuta las operaciones de bit, dependiendo del código de la operación que se esté procesando.
- **out_sel**: estes bloque es un multiplexor que selecciona el valor pedido de la ALU (en función de la instrucción que se esté ejecutando) y determina el valor de los flags 'ZNV'.

## *2.4 VERIFICACIÓN*

Para la verificación del nanoprocesador, existe un testbench sobre los que se realizan una serie de tests para comprobar el correcto funcionamiento de todas las instrucciones existentes, así como de diferentes combinaciones entre ellas para reducir en la medida de lo posible el riesgo de fallo. Para este proceso de verificación se ha utilizado la herramienta Questa Sim.

Los diferentes tests se combinan para realizar el *code coverage,* es decir, la comprobación de que durante la verificación se ejecutan todas las líneas del código VHDL al menos una vez.

Por último, se ejecuta el proceso de síntesis (mediante el programa Synplify Pro), consistente en la verificación de que no existen problemas en la programación que no sean compatibles con la futura implantación del mismo en la FPGA elegida.

### 2.4.1 TESTBENCH

El diagrama de bloques del testbench en su estado inicial se adjunta en la Parte IV, Figura 40. A continuación, se explica el funcionamiento del mismo:

- Existen dos instancias del nanoprocesador: una para funcionamiento rápido, es decir, el que permite hacer lecturas de datos en los flancos de bajada del reloj, y otro para funcionamiento lento, es decir, el que sólo permite hacer lecturas y escrituras en los flancos de subida del reloj. De esta manera se puede verificar el correcto funcionamiento de ambos modos de manera paralela, sin tener que hacer los tests dos veces. Como se ha indicado en el apartado 2.1.2, esta característica viene determinada por el genérico **g_fast_exec**.
- Las memorias internas se cargan mediante una interfaz genérica APB (una para la memoria de instrucción y otra para la memoria de registros). El maestro que gestiona los esclavos de estas interfaces es el mismo para ambas, ya que en ningún caso es necesaria la utilización de las dos simultáneamente. Las memorias internas de ambas instancias del nanoprocesador se cargan a la vez.
- El bloque p_setup escoge las señales de la implementación rápida o lenta.
- Cada una de las instancias del nanoprocesador tiene asociada una memoria externa, a la que se conecta mediante una interfaz genérica APB.

A modo de ejemplo se incluyen unas imágenes en las que se muestra una sección de una simulación, así como algunas de las señales principales, que se explicarán posteriormente. En la Figura 4 se puede observar el funcionamiento del procesador utilizando el doble flanco del reloj para hacer lecturas (implementación rápida), mientras que en la Figura 5 se representa el proceso

utilizando solamente el ciclo de subida (implementación lenta). Como curiosidad, cabe destacar que, aunque se considere a la primera como implementación rápida, la frecuencia del reloj en este caso es más baja que en el segundo supuesto. Como la frecuencia se autoajusta para que la ejecución sea lo más rápida posible, la frecuencia es aproximadamente la mitad en el primer caso. Esto es debido a que tiene que hacer los mismos procesos que en la implementación lenta, pero en sólo medio ciclo de reloj mientras que en la lenta tiene el ciclo de reloj completo para realizar el mismo procedimiento.

Todas las señales expuestas en las simulaciones se explican a continuación, en la Tabla 4.

| SEÑAL | DESCRIPCIÓN |
|---|---|
| ***clk*** | Reloj del sistema |
| ***reset_n*** | Reset (negado) |
| ***enable*** | *Enable* |
| ***if_dbg_cnt (15:0)*** | FETCH – Número de ciclo de *debug* |
| ***if_opcode (5:0)*** | FETCH – Código de operación de la instrucción en proceso |
| ***id_dbg_cnt (15:0)*** | DEC – Número de ciclo de *debug* |
| ***id_opcode (5:0)*** | DEC – Código de operación de la instrucción en proceso |
| ***id_rs_imm (15:0)*** | DEC – Valor de RS |
| ***id_rs_addr (8:0)*** | DEC – Dirección de RS |
| ***id_rs_sel (2:0)*** | DEC – Fuente de RS |
| ***id_rt (15:0)*** | DEC – Valor de RT |
| ***id_rt_sel (3:0)*** | DEC – Fuente de RT |
| ***id_rt_addr (8:0)*** | DEC – Dirección de RT |
| ***id_rt_wr*** | DEC – Orden de escritura en RT |
| ***id_mem_rd*** | DEC – Orden de lectura en memoria externa |
| ***id_mem_wr*** | DEC – Orden de escritura en memoria externa |
| ***id_ready*** | DEC – Flag de *ready* |

| | |
|---|---|
| ***ex_dbg_cnt (15:0)*** | EXEC – Número de ciclo de *debug* |
| ***ex_opcode (5:0)*** | EXEC – Código de operación de la instrucción en proceso |
| ***ex_ready*** | EXEC – Flag de *ready* |
| ***ex_rt_addr (8:0)*** | EXEC – Dirección de RT |
| ***ex_rt_wr*** | EXEC – Orden de escritura en RT |
| ***ex_rt (15:0)*** | EXEC – Valor de RT |
| ***ex_rl (8:0)*** | EXEC – Valor de RL |
| ***ex_znv (2:0)*** | EXEC – Valor de los flags ZNV |

*Tabla 4. Señales simulación*

La principal diferencia entre ambas implementaciones reside en que, en la segunda, se necesita un ciclo de reloj más en la etapa de decodificación. Esto se debe a que los valores de lectura de los registros no se obtienen hasta finalizar el primer ciclo de esta fase. Como se necesita registrar estos datos para poder transferirlos a la siguiente etapa del pipeline, se necesita un segundo ciclo. Para esto existe una etapa de pre-decodificación, que se encarga de preparar los datos de lectura. En el caso de la implementación rápida, esta etapa se ejecuta a la vez que la de decodificación; en la implementación lenta, se ejecuta en el ciclo anterior, por lo que se necesitan dos ciclos de reloj.

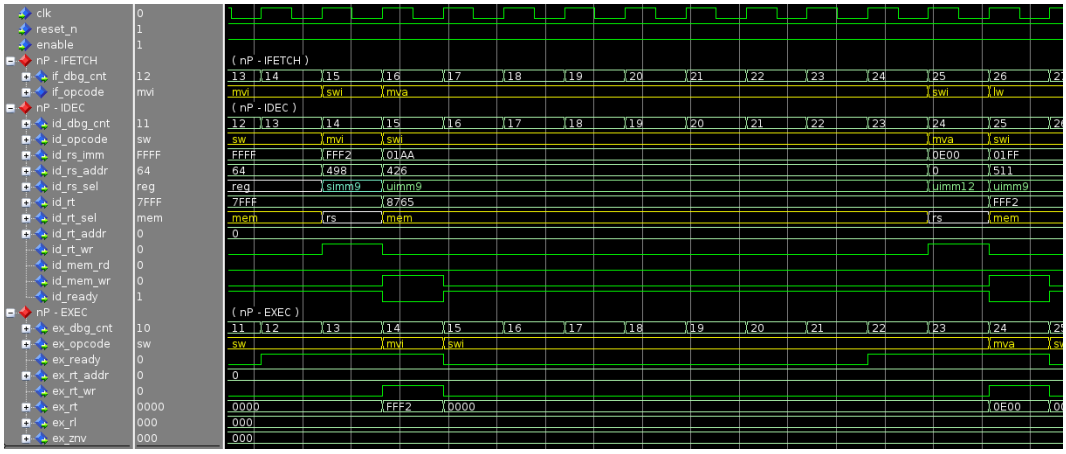


*Figura 4. Simulación - implementación rápida*

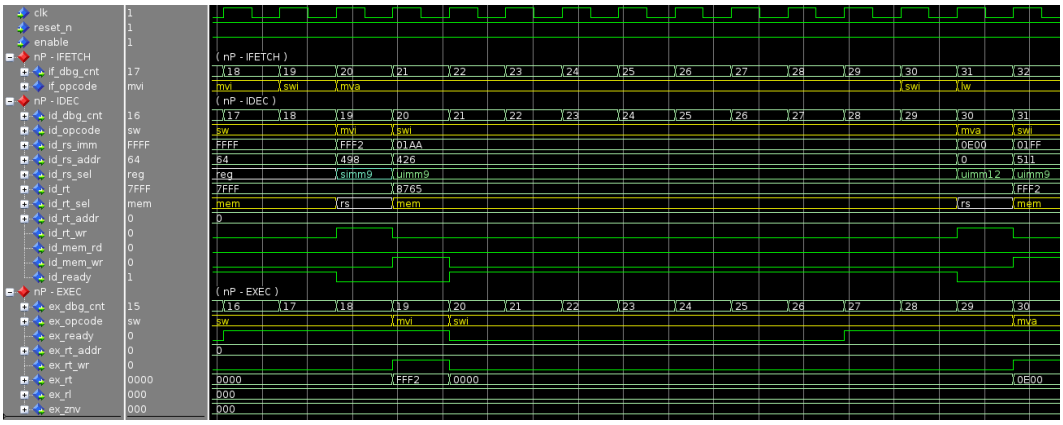


*Figura 5. Simulación - implementación lenta*

En ambas imágenes se muestra el mismo fragmento de la simulación, compuesto por tres microinstrucciones que se explican a continuación. Entre paréntesis se indica en qué ciclo comienza a ejecutarse dicha instrucción, tanto en la implementación rápida como en la implementación lenta.

- **mvi %r0, 0x01F2** (*dbg_cnt* = 14/18) → La etapa de IFETCH introduce la instrucción que se va a ejecutar. En el IDEC, lo primero que se hace es extender el signo del inmediato de 9 bits que se utiliza, ya que es con signo (como se muestra en *id_rs_sel* – simm9), hasta 16 bits, ya que este es el tamaño de los registros (se puede ver el resultado en *id_rs_imm*). El flag *id_rt_wr* indica que en esta instrucción se requiere la escritura del registro RT, en este caso el registro 0 (como se observa en la señal *id_rt_addr*). Por último, en la etapa de EXEC se escribe el registro indicado anteriormente; su nuevo valor se puede observar en la señal *ex_rt*.
- **swi %r0, 0x01AA** (*dbg_cnt* = 15/19) → Como esta instrucción requiere la intervención de una memoria externa al nanoprocesador, dura más ciclos de reloj que las instrucciones normales. Esta memoria externa se comunica a con el dispositivo a través de la etapa de ejecución. Como es necesario que el fetch se detenga y no envíe nuevas instrucciones hasta que las demás etapas estén listas para poder procesarlas, hay que indicárselo: esto se hace mediante las señales *id_ready* y *ex_ready*. Esta última, que indica que la etapa de ejecución no ha terminado su tarea, permanece a '0' hasta

que la memoria externa devuelve un flag de ACK (que no se muestra en las imágenes). El *flag id_ready* se pone a '0' durante un ciclo (dos ciclos en el caso de la implementación lenta), no porque esta etapa lo necesite, sino para que no se manden nuevas instrucciones durante este tiempo en el que todavía no se puede manipular la señal *ex_ready*.

- **mva %r0, 0x0E00** (*dbg_cnt* = 16/20) → Esta instrucción es similar a la primera, sólo que permite la utilización de un inmediato de mayor tamaño (hasta 12 bits). En este caso, el inmediato es sin signo (*id_rs_sel* – uimm12), así que no es necesario realizar la extensión del mismo hasta ocupar los 16 bits, sino rellenarlo con ceros. Cabe destacar que, aunque esta instrucción entre en el sistema en el ciclo 16 (20 para implementación lenta), no se transfiere a las etapas de decodificación y ejecución hasta varios ciclos después. Esto es debido a que va precedida por una instrucción de acceso a memoria externa, y permanece en espera hasta que ésta pone los *flags* de *ready* a '1' de nuevo.

### 2.4.2 TESTS

Para comprobar el funcionamiento del *testbench*, existe una lista de regresión compuesta por los siguientes scripts:

- Un microcódigo que contiene todas las distintas microinstrucciones, con distintas variaciones en caso necesario (resultado positivo/negativo, *overflow*…). Es el test principal de la serie, en el que se comprueba que funciona cada cambio que se realiza en el diseño. Este microcódigo se verifica a si mismo, comparando el valor de las operaciones realizadas con el resultado esperado y saltando a un bucle de error en caso de fallo.
- Un microcódigo que comprueba una vez todas las instrucciones.
- Un microcódigo que contiene la lógica de las Torres de Hanoi. Cada una de las torres está representada por un registro, y cada una de las piezas está representada por un bit en los registros, siendo el MSB la pieza más

grande y el LSB la pieza más pequeña. Si el bit está a '1', significa que la pieza está en esa torre.

- Un microcódigo que ordena una lista de números en orden ascendente. Compara dos números correlativos de la lista; si el primero es más grande que el segundo, los cambia de sitio. A continuación, compara el segundo con el tercero, y así sucesivamente.
- Un microcódigo que utiliza el mismo registro siempre que es posible. Este test es útil para comprobar que no hay peligro de que ocurran *hazards* en la ejecución del código.
- Un microcódigo que busca un número en una lista de 20 elementos. Guarda la posición o posiciones de memoria del número en la memoria externa.
- Un microcódigo en el que se implementan algunas operaciones matemáticas.
- Un microcódigo que calcula el mayor común divisor de dos números naturales.
- Un microcódigo que ejecuta cada comando antes y después de cada comando. De esta manera se reduce significativamente la probabilidad de fallo del nanoprocesador.

A grandes rasgos, la secuencia utilizada para la verificación del test principal (el primero de la serie, siendo el que se utiliza para comprobar que los cambios que se hacen en el diseño funcionan correctamente) consta de los siguientes pasos:

1. Inicializar las memorias de instrucción y de registros, rellenando todos los valores a cero para evitar detecciones de errores en los campos no utilizados.
2. Rellenar la memoria de instrucción con el microcódigo correspondiente al test en cuestión.
3. Forzar errores simples y dobles en la memoria de instrucción para su posterior comprobación.

4. Rellenar la memoria de registros con los datos indicados por el test.
5. Forzar errores simples y dobles en las memorias de registros (RT y RS) para su posterior comprobación.
6. Ejecutar el microcódigo del test, y posteriormente comprobar el valor de ciertas posiciones de la memoria externa para verificar que se ha ejecutado correctamente.
7. Comprobar que los errores simples de la memoria de instrucción y las memorias de registros se han corregido.
8. Comprobar que los errores dobles de la memoria de instrucción y las memorias de registros se han corregido.
9. Ejecutar el microcódigo paso a paso (microinstrucción a microinstrucción), y posteriormente comprobar el valor de ciertas posiciones de la memoria externa para verificar que se ha ejecutado correctamente.
10. Ejecutar el test de cobertura del microcódigo.
11. Forzar errores simples en el PC y comprobar que se han detectado.
12. Forzar errores dobles en el PC y comprobar que se han detectado.

### 2.4.3 CODE COVERAGE

El *code coverage* o test de cobertura consiste en comprobar qué cantidad de código VHDL de un programa se ha ejecutado, al menos una vez, tras correr un test o un conjunto de tests sobre éste [7]. El resultado viene dado en porcentaje, tal y como se muestra en el ejemplo de la Figura 6.

```
================================================================================
=== File: ./np_iexec_rtl.vhd
================================================================================
    Enabled Coverage              Active      Hits    Misses % Covered
    ----------------              ------      ----    ------ ---------
    Stmts                            128       124         4      96.8
    Branches                         101        99         2      98.0
    FEC Condition Terms               20        18         2      90.0
    FEC Expression Terms               8         7         1      87.5
    FSMs                                                        100.0
        States                         0         0         0     100.0
        Transitions                    0         0         0     100.0
    Toggle Bins                      634       596        38      94.0
```

*Figura 6. Code coverage*

Es importante el cumplimiento de este tipo de verificación, ya que, si alguna línea de un código no se ha ejecutado nunca, no se puede comprobar si existe algún error en ella. Como se puede observar en la imagen, mediante la implementación del *code coverage* se realizan diferentes tipos de comprobaciones:

- ***Statement***: comprueba que se hayan ejecutado todas las líneas de código.
- ***Branch***: comprueba que todos los saltos existentes en el programa (if/else, case…) se hayan ejecutado tanto con resultado positivo como con resultado negativo.
- ***FEC***: comprueba que se hayan ejecutado todas las combinaciones significativas dentro de una condición o expresión.
    - ***Condition***: expresiones combinacionales dentro de una condición. Por ejemplo, en la condición: if(a and (not(b) or c)). Se comprobarían las combinaciones:
        - a = ‘0’
        - a = ‘1’, (not(b) or c) = ‘0’
        - a = ‘1’, (not(b) or c) = ‘1’
    - ***Expression***: expresiones combinacionales dentro de una expresión de asignación de un valor a una variable. Por ejemplo, en la expresión: a <= b and (c or d);
- ***FSM***: comprueba que se ejecuten todas las posibilidades dentro de una máquina de estados.

- ***States***: comprueba que, en la máquina de estados, se haya pasado por todos los estados.
- ***Transitions***: comprueba que, en la máquina de estados, se haya pasado por todas las transiciones.

- ***Toggle bins***: comprueba que se hayan dado todas las posibilidades de transición en todos los bits de todas las señales (de '0' a '1' y de '1' a '0').

En este caso resultan interesantes los tres primeros tipos, es decir, *statement*, *branch*, y *FEC*. Esto es debido a que no existe ninguna máquina de estados en este proyecto (por lo que las verificaciones tipo *FSM* no aplican), y los *toggle bins* sólo interesan en el *top level* del mismo, ya que hay muchas señales que tienen algunos bits que no se modifican, o señales que no se espera que cambien su valor por tratarse, por ejemplo, de señales de error.

Además, es posible la creación de un script de casos que están excluidos de este tipo de verificación. Puede resultar conveniente su uso si existen partes del código que no se deben ejecutar en funcionamiento normal, como saltos que sólo se hacen cuando existe algún error en el sistema, un *case-others*… En el caso de este proyecto, el *code coverage* aplicado sobre el mismo código, con los mismos tests, pero con un archivo de exclusiones (todas las exclusiones deben estar perfectamente justificadas), obtendría como resultado los porcentajes mostrados la Figura 7.

```
================================================================================
=== File: ./np_iexec_rtl.vhd
================================================================================
    Enabled Coverage              Active      Hits    Misses % Covered
    ----------------              ------      ----    ------ ----------
    Stmts                            125       123         0      100.0
    Branches                          98        98         0      100.0
    FEC Condition Terms               20        20         0      100.0
    FEC Expression Terms               8         8         0      100.0
    FSMs                                                           100.0
        States                         0         0         0      100.0
        Transitions                    0         0         0      100.0
    Toggle Bins                      634       596        38       94.0
```

*Figura 7. Code coverage con exclusiones*

### 2.4.4 SÍNTESIS

El proceso de síntesis consiste en la representación automática con puertas lógicas del código VHDL creado. Esto es útil para conocer la cantidad de puertas lógicas que necesita el diseño creado, así como las que utiliza cada proceso del mismo. También se utiliza para comprobar que no existan conflictos entre las puertas requeridas y las disponibles en la FPGA que se desea utilizar, así como para obtener una estimación del tiempo que tardarán los diferentes procesos del código diseñado en ejecutarse. Para este fin se ha utilizado el programa Synplify Pro.

# Capítulo 3 IMPLEMENTACIÓN DE NUEVAS INSTRUCCIONES

La primera modificación que se ha realizado en el nanoprocesador consiste en el añadido de instrucciones nuevas, que permiten realizar operaciones que hasta el momento no se podían realizar, o se podían realizar utilizando una combinación de las instrucciones existentes previamente. Es interesante la ampliación de posibilidades en cuanto instrucciones a utilizar, debido a que los programas que el dispositivo puede procesar están limitados a 4k instrucciones.

Como se ha explicado previamente, los diferentes tipos de instrucciones a implementar son:

- Punteros
- Punteros con offset
- Shiftado

Para cada uno de los comandos se ha procedido en primer lugar a la implementación de la misma en el funcionamiento rápido (lectura de datos tanto en flanco de subida como en flanco de bajada), para posteriormente hacer las correcciones necesarias para conseguir también su correcta ejecución en el funcionamiento lento. Es importante tener en cuenta la posibilidad de apariciones de *hazards* en todas las instrucciones.

Para todo el proceso de diseño se ha utilizado el software de diseño HDL Designer, así como la ayuda de diferentes editores de texto.

A continuación, se explica en detalle cada una de las nuevas operaciones disponibles en el nanoprocesador, así como el diseño necesario para el funcionamiento en cada una de ellas, y el proceso de verificación a seguir para comprobarlo.

## *3.1 PUNTEROS*

En primer lugar, se van a añadir las instrucciones necesarias para leer y escribir en un registro indexado mediante otro registro, es decir, mediante un puntero. Estas instrucciones son:

- Instrucción de lectura: mvr RT, RS → RT = R(RS)
- Instrucción de escritura: mvw RT, RS → R(RT) = RS

### 3.1.1 MVR

El comando MVR sirve para leer el contenido del registro al que apunta el registro RS, y guardar su valor en el registro RT.

Como se ha indicado anteriormente, la mayoría de las instrucciones solamente necesitan un ciclo de reloj en cada una de las etapas del pipeline. No es el caso de MVR, ya que se necesitarán ciclos extra en la etapa de decodificación. Como se puede observar en el diagrama de bloques de la etapa de decodificación **i_idec** (Figura 35), durante esta fase se lee el valor tanto del registro RT como del registro RS (en los casos en los que sea necesario). En este proceso, no se necesita leer el valor de RT, pero sí dos veces el valor de RS: en el primer ciclo de la decodificación se leerá por primera vez el registro RS, y en el segundo se leerá por segunda vez, siendo el contenido de la primera lectura la dirección del registro en la segunda. Esto se indicará creando un nuevo bloque en el diagrama (**rs_addr_selection**), desde el que se escogerá la dirección a introducir en la memoria de registros para obtener el resultado correcto en cada caso.

En esta etapa de decodificación es necesario indicar si se está ejecutando el primer ciclo de esta instrucción o el segundo. Esto se ha conseguido mediante la creación de la señal *mvr_ph2*, cuyo valor dependerá del código de operación y de la señal intermedia *ready*, que indica si el dispositivo está listo para proceder a la siguiente fase del pipeline; como el proceso de pipeline es secuencial, en el segundo ciclo

de la decodificación esta señal tendrá un valor de '1' (en el primer ciclo será de '0'). Esta nueva señal (*mvr_ph2*) tomará un valor de '1' durante el segundo ciclo de decodificación de la instrucción MVR, permaneciendo a '0' el resto del tiempo. El valor de esta señal se regulará en un nuevo bloque dentro de la etapa de decodificación, llamado **ph2_reg**.

En cuanto a la etapa de ejecución, en el primer ciclo de la misma se ha creado una burbuja (es decir, se ha insertado una instrucción que no modifica ninguno de los valores de las señales existentes, correspondiente a un NOP), mientras que en el segundo ciclo es donde se realiza la escritura del registro RT.

En la implementación lenta, el decodificador tarda todavía un ciclo más que en la implementación rápida, teniendo que esperar un total de tres ciclos de reloj a que esta etapa termine su tarea. Los pasos que sigue esta fase del pipeline para tardar este número de ciclos son los siguientes:

- Pre-decodificación: prepara los datos que se quieren leer.
- Decodificación – primer ciclo: lee por primera vez el registro RS.
- Decodificación – segundo ciclo: prepara el dato de la segunda lectura de RS (esto se puede hacer en el mismo ciclo que el paso anterior en el caso de la implementación rápida).
- Decodificación – tercer ciclo: lee por segunda vez el registro RS.

### 3.1.2 MVW

La instrucción MVW se utiliza para leer el registro RS y guardar su valor en el registro al que apunta el contenido del registro RT.

La implementación de este comando es bastante sencilla, puesto que la única diferencia con un MV normal es que la dirección de escritura será el contenido de RT en lugar del propio registro RT. Por lo tanto, en el bloque **writeback** de la etapa de ejecución (mostrado en la Figura 38 de la Parte IV), donde se registran las señales de salida y se escriben los registros con los nuevos valores, será necesario tener esto en cuenta.

En la Figura 8 y la Figura 9 se puede observar un ejemplo de simulación de las instrucciones direccionadas por punteros, en implementación rápida y lenta respectivamente. En concreto, las instrucciones son:

- mvw %r250, %r253 (*dbg_cnt* = 1301/1433)
- mvr %r250, %r253 (*dbg_cnt* = 1304/1436)

Siendo los valores iniciales y finales de los registros implicados los representados en la Tabla 5.

| REGISTRO | VALOR INICIAL | VALOR FINAL |
|---|---|---|
| **%r250** | 0x00FB | 0x0011 |
| **%r251** | 0xFFFF | 0x00FE |
| **%r253** | 0x00FE | 0x00FE |
| **%r254** | 0x0011 | 0x0011 |

*Tabla 5. MVR y MVW - valores de los registros*

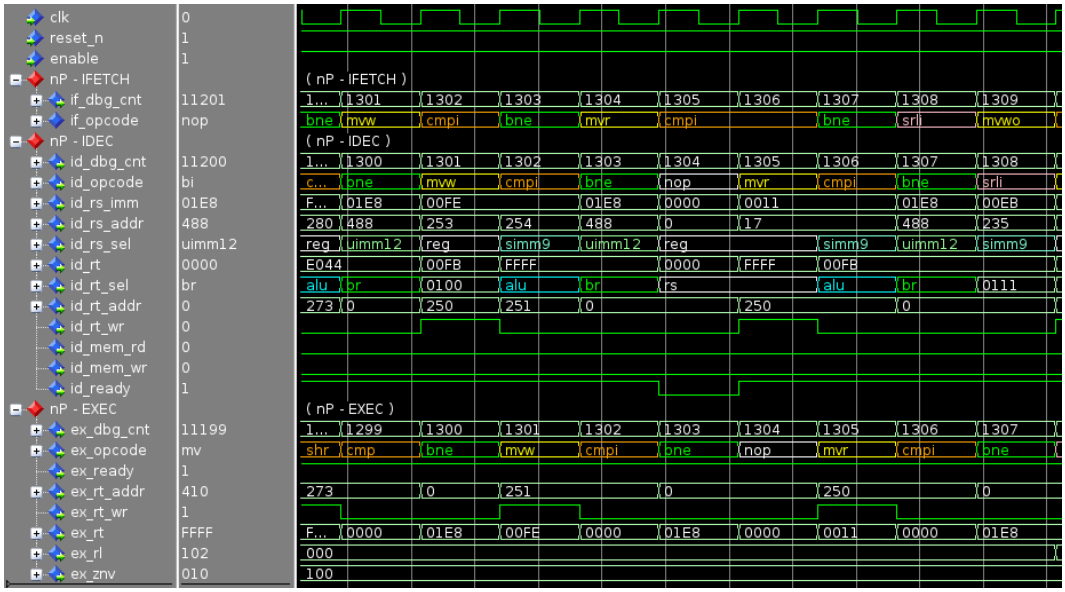


*Figura 8. MVR y MVW - implementación rápida*

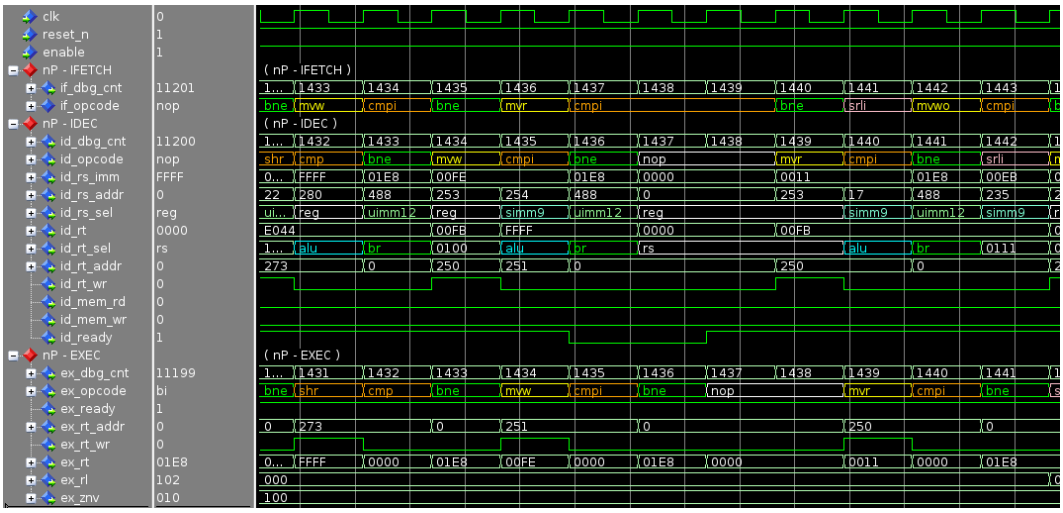


*Figura 9. MVR y MVW - implementación lenta*

## 3.2 *PUNTEROS CON OFFSET*

Después de implementar las instrucciones de punteros, se ha diseñado el código para las correspondientes a punteros con offset.

En primer lugar, se requieren instrucciones de lectura y escritura indexada mediante registros con *offset*:

- mvro RT, RS → RT = R(RS + RL)
- mvwo RT, RS → R(RT + RL) = RS

Para poder acceder a este registro que almacena el *offset* (RL: *Link Register*) se necesitan también algunas instrucciones:

- lrl RT → RT = RL
- srl RS → RL = RS
- srli imm9 → RL = imm9

Por último, se ha decidido conveniente diseñar comandos que permitan operar sobre este registro para poder modificarlo de manera más directa:

- addrl RS → RL = RL + RS, update ZNV
- addrli imm9 → RL = RL + signext(imm9), update ZNV

- subrl RS → RL = RL – RS, update ZNV
- subrli imm9 → RL = RL – signext(imm9) update ZNV

El registro RL almacena un número de 9 bits, ya que las instrucciones para manipularlo utilizan inmediatos de esta longitud.

### 3.2.1 MVRO

La instrucción MVRO sirve para leer el registro cuya dirección se corresponde con el contenido del registro RS más el valor de RL, y almacenarlo en el registro RT. Este comando es análogo a MVR, con la diferencia de que hay que sumarle el valor de RL al contenido de RS para obtener la dirección del registro de lectura.

De igual manera que en el caso anterior, se creará la señal *mvro_ph2*, regulada también desde el bloque **ph2_reg**. En función de esta señal, se escogerá el valor adecuado de la dirección de RS en el bloque **rs_addr_selection**. Para este comando, antes de introducir en valor de la dirección en la memoria de registros, es necesario realizar una suma.

Como el único sumador existente hasta el momento dentro del proyecto está situado en la fase de ejecución, utilizarlo implicaría un retardo en el procesado de esta instrucción de al menos otros dos ciclos de reloj (uno para registrar el valor antes de enviarlo del decodificador a la etapa de ejecución, y otro para hacer el camino inverso). Por este motivo, se ha considerado conveniente la introducción de un nuevo sumador en esta etapa del pipeline, alojado dentro de un nuevo bloque llamado **add_rl**. De esta manera, no existirá ningún retardo adicional en la ejecución del comando MVRO; el único que se seguirá sufriendo será el asociado a la doble lectura del registro RS (un ciclo en la implementación rápida y dos ciclos en la lenta).

Como las longitudes de RL y de RS no coinciden, antes de sumar estos dos valores será necesaria la extensión del signo de RL para poder realizar la operación, volviendo a reducir posteriormente el número de bits del resultado para poder utilizarlo como una dirección de un registro. Se ha decidido hacerlo de esta

manera, y no reduciendo directamente el tamaño de RS, ya que de esta forma se reduce la posibilidad de que exista *overflow* (especialmente en caso de que ambos operandos tengan distinto signo).

El resto del procedimiento es similar al del comando MVR, teniendo que introducir las burbujas dentro del proceso de decodificación, y finalmente actualizando el valor de los registros.

### 3.2.2 MVWO

El comando MVWO se utiliza para leer el registro RS, y guardar su valor en el registro cuya dirección coincide con la suma del valor de RL y el contenido del registro RT.

Esta instrucción es análoga a MVW. La diferencia reside en que, antes de indicar la dirección de RT sobre la que se quiere escribir, hay que añadir el valor de RL al resultado de la lectura del registro RT. A diferencia que en el caso de MVRO, en este caso sí que se puede utilizar el sumador existente en la etapa de ejecución (dentro de la ALU), ya que el proceso de escritura del registro se hace posteriormente.

Para realizar la suma, ha sido necesario crear un bloque para escoger los operadores que se introducirán dentro del sumador. En este caso, el operador que varía es el ‘b’, por lo que se ha creado el bloque **p_sel_op_b** para seleccionar la entrada oportuna (RS o RL). Se escogerá el valor necesario en función del código de operación de la instrucción a ejecutar.

De igual manera que en el caso de MVRO, se extenderá el signo de RL antes de realizar la operación. Además, es necesario tener en cuenta que, en este caso, el signo del resultado no vendrá dado por el último bit del mismo (bit 15), sino por el bit 8. Esto es debido a que este resultado se utilizará como dirección de registro, por lo que el valor válido viene dado por los 9 bits de posición más baja. Para controlar el *overflow* de la operación, se ha de tener en cuenta esto mismo; de esta

manera, todos los bits más significativos deben de coincidir entre ellos, y también coincidir con el bit de signo.

En la Figura 10 y la Figura 11 se puede observar un ejemplo de ejecución de los comandos MVRO y MVWO, en implementación rápida y lenta respectivamente. En concreto, las instrucciones procesadas son:

- mvwo %r250, %r253 (*dbg_cnt* = 1309/1442)
- mvro %r250, %r254 (*dbg_cnt* = 1312/1445)

Siendo los valores iniciales y finales de los registros implicados los representados en la Tabla 6.

| REGISTRO | VALOR INICIAL | VALOR FINAL |
|---|---|---|
| **%r250** | 0x0011 | 0x00FE |
| **%r252** | 0x01FF | 0x00FE |
| **%r253** | 0x00FE | 0x00FE |
| **%r254** | 0x0011 | 0x0011 |
| **RL** | 0x0EB | 0x0EB |

*Tabla 6. MVRO y MVWO - valores de los registros*

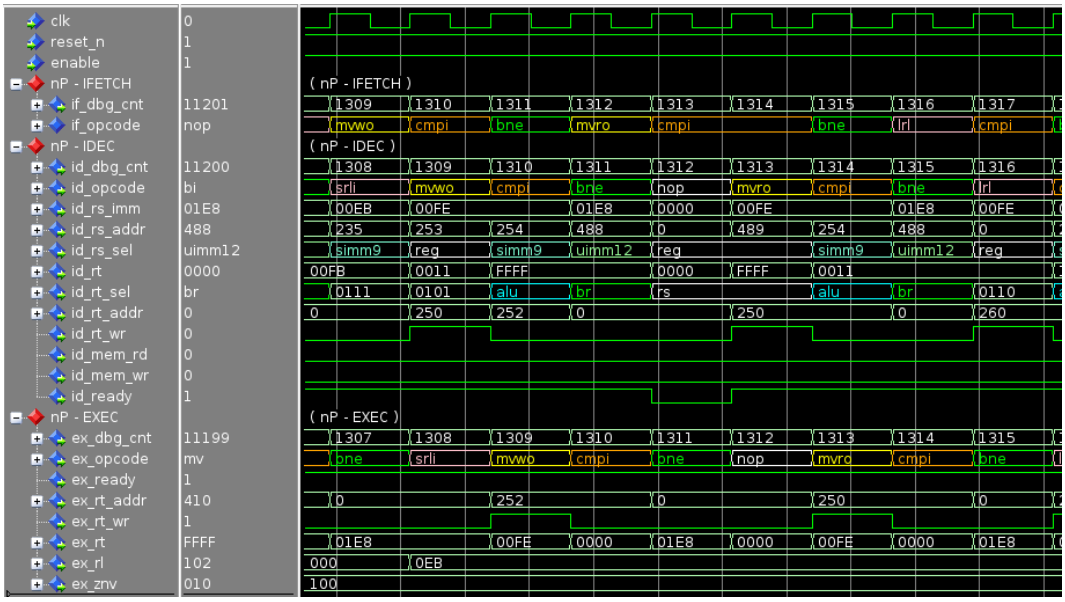


*Figura 10. MVRO y MVWO - implementación rápida*

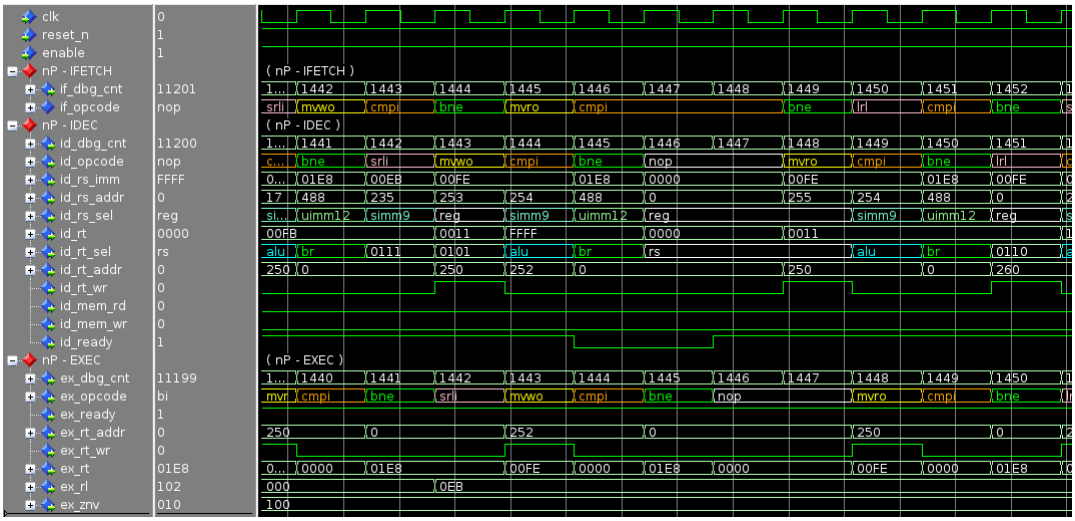


*Figura 11. MVRO y MVWO - implementación lenta*

## 3.2.3 LRL

La instrucción LRL sirve para poder recuperar el valor de RL y guardarlo en el registro RT.

La diferencia con un MV normal (escribir en el registro RT el contenido del registro RS) es que, en este caso, no es necesario leer el registro RS, simplemente se almacena el valor de RL, que, al ser directamente el valor de una señal, no es necesario leerla de nuevo. En el bloque **writeback** se indicará que el valor a escribir en el registro RT indicado será el contenido de RL, extendiendo su signo hasta rellenar los 16 bits. La preparación previa del valor de RT (extensión de signo de RL) se ha realizado en el nuevo bloque **l_s_rl** (dentro de la etapa de ejecución), para mantener el bloque que contiene el proceso de escritura y registrado lo más limpio posible.

En la Figura 12 y la Figura 13 se muestra un ejemplo de ejecución de esta instrucción, tanto con lectura en doble flanco de reloj como en un único flanco. La instrucción implementada es:

- lrl %r260

Siendo el valor de RL = 0x0EB.

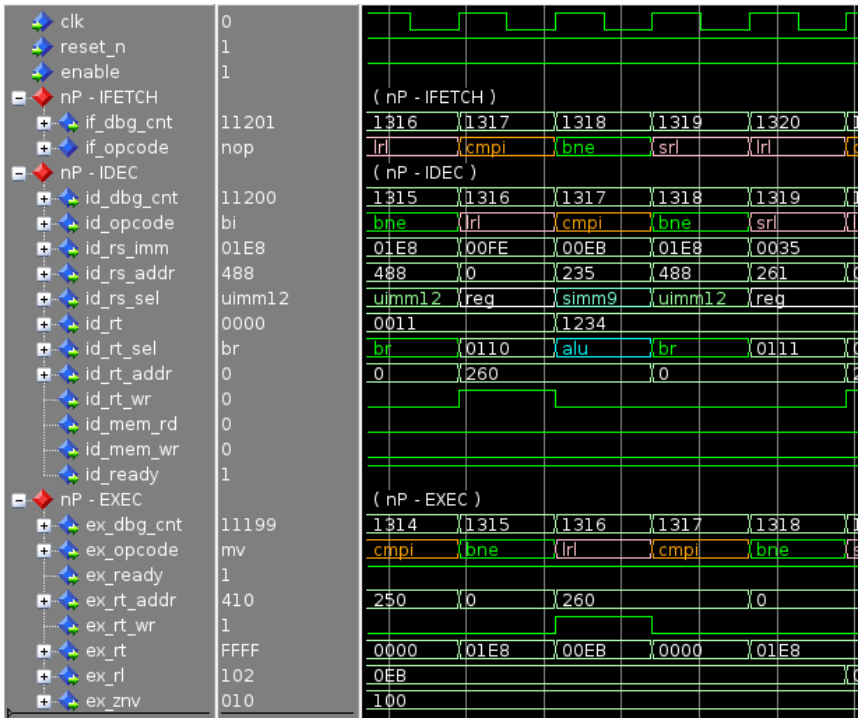


*Figura 12. LRL - implementación rápida*

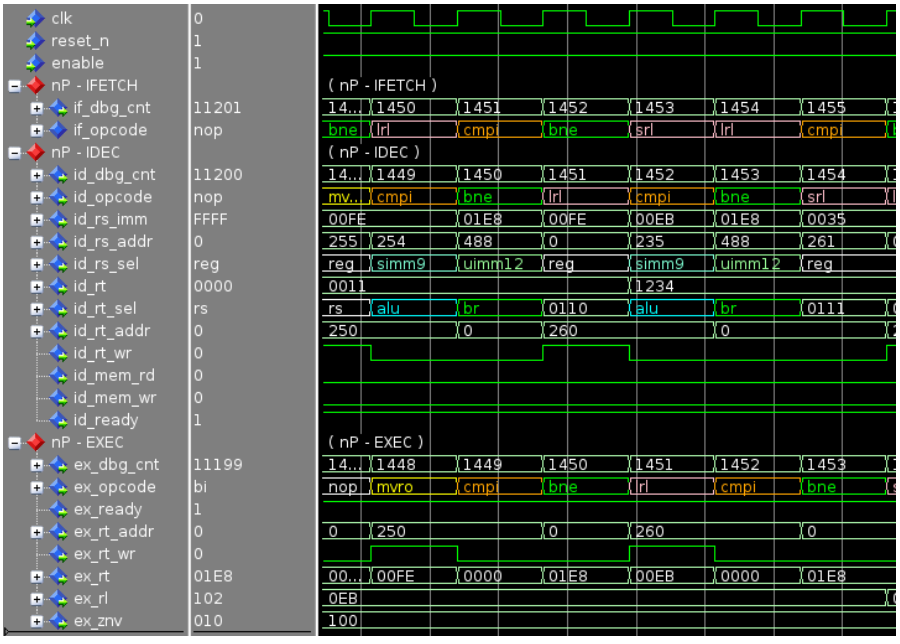


*Figura 13. LRL - implementación lenta*

### 3.2.4 SRL

El comando SRL se utiliza para cargar en RL el valor del registro RS. Es necesario tener en cuenta que hay que reducir el tamaño del contenido de RS para poder almacenarlo en RL, ya que éste disminuye de 16 a 9 bits. Para esto, es importante observar el signo de RS, así como si existe *overflow*, es decir, si los bits (8 to 15) no coinciden. La preparación del valor a escribir en RL se hace en el mismo bloque que en el caso de la instrucción LRL (en **l_s_rl**).

En este caso será necesaria una lectura del registro RS, pero no se escribirá en ningún momento un registro RT, sino que el valor se almacenará simplemente en RL.

En la Figura 14 y la Figura 15 se puede observar una simulación que contempla dos posibilidades de resultado de este comando:

- srl %r261 (*dbg_cnt* = 1319/1453)
- srl %r262 (*dbg_cnt* = 1323/1457)

Donde los valores iniciales y finales de los registros implicados en este proceso son los reflejados en la Tabla 7.

| REGISTRO | VALOR INICIAL | VALOR FINAL |
|---|---|---|
| **%r261** | 0x0035 | 0x0035 |
| **%r262** | 0x0457 | 0x0457 |
| **RL** | 0x00EB | 0x00FF |

*Tabla 7. SRL - valores de los registros*

Se puede observar que, en el primer caso, el resultado de RL (*ex_rl* = 0x035) es precisamente el contenido del registro RS escogido (%r261). En cambio, en el segundo intento, el valor de RL no es el mismo que el del registro (0x0457), ya que existe *overflow*; el valor que se intenta almacenar en RL ocupa más de los 9 bits permitidos, por lo que se mantiene el signo del número en cuestión y se trunca al valor máximo posible (0x0FF). No se actualizan los flags ZNV con el uso de

estos comandos (aunque exista un *overflow*) porque éstos están asociados al resultado de la ALU.

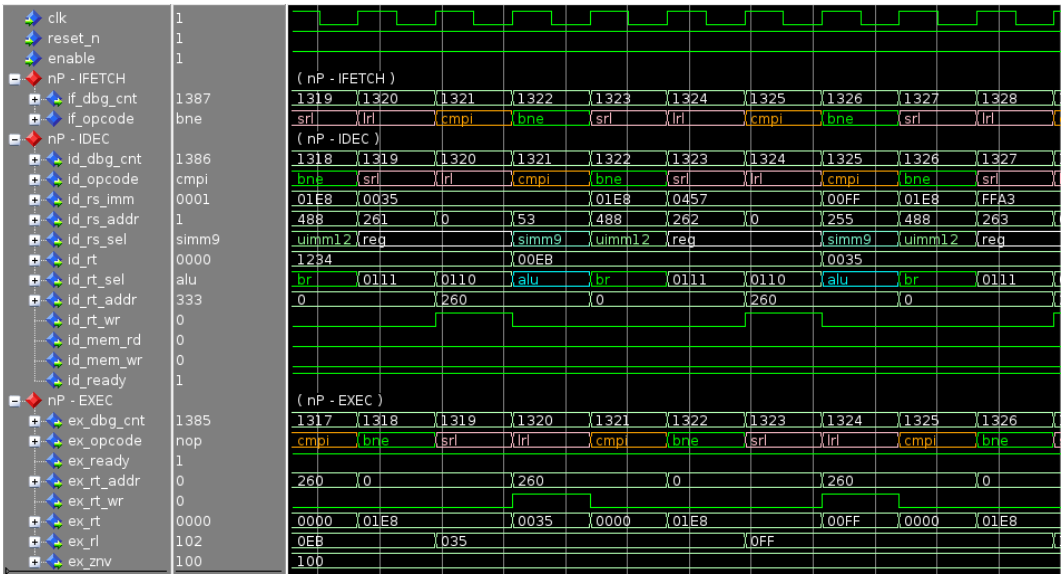


*Figura 14. SRL - implementación rápida*

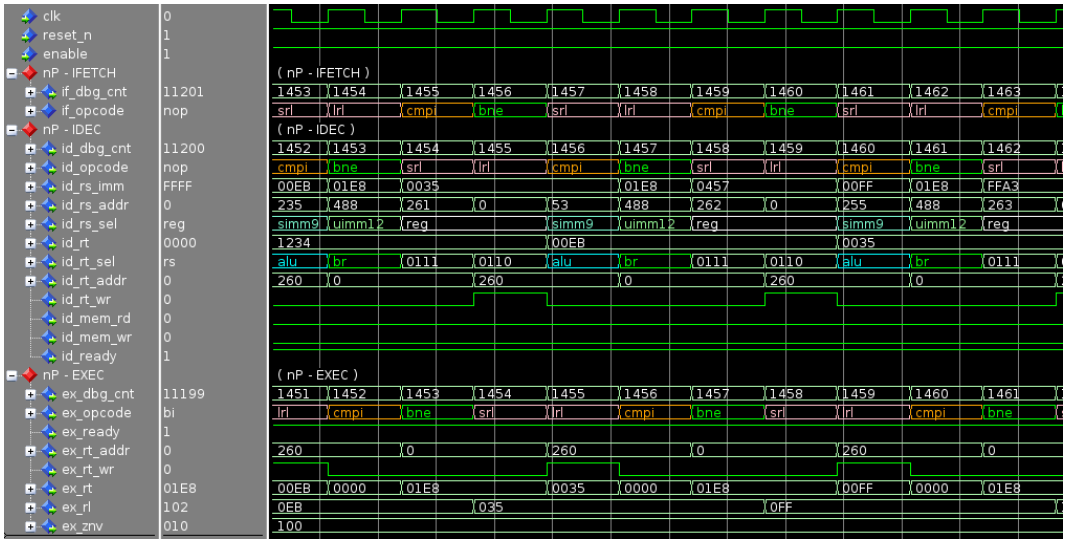


*Figura 15. SRL - implementación lenta*

## 3.2.5 SRLI

La instrucción SRLI sirve para guardar en RL el valor de un inmediato de 9 bits. En este caso no es necesaria la reducción de tamaño del número a guardar, ya que

su longitud ya es de los bits adecuados. Teniendo esto en cuenta, el proceso de implementación de este comando es análogo al de la instrucción SRL.

En la Figura 16 y la Figura 17 se muestran ejemplos de este nuevo comando:

- srli 0x045 (*dbg_cnt* = 1335/1469)
- srli 0x1A3 (*dbg_cnt* = 1339/1473)

Se puede observar que el comando funciona tanto con un inmediato positivo como con uno negativo. El valor de RL se puede observar en la señal *ex_rl* para cada caso.

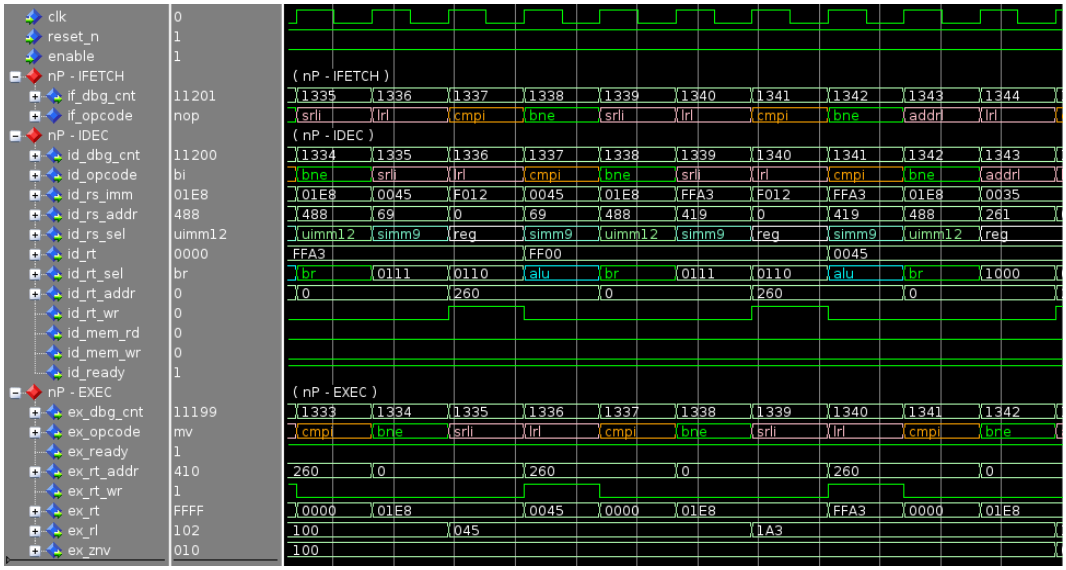


*Figura 16. SRLI - implementación rápida*

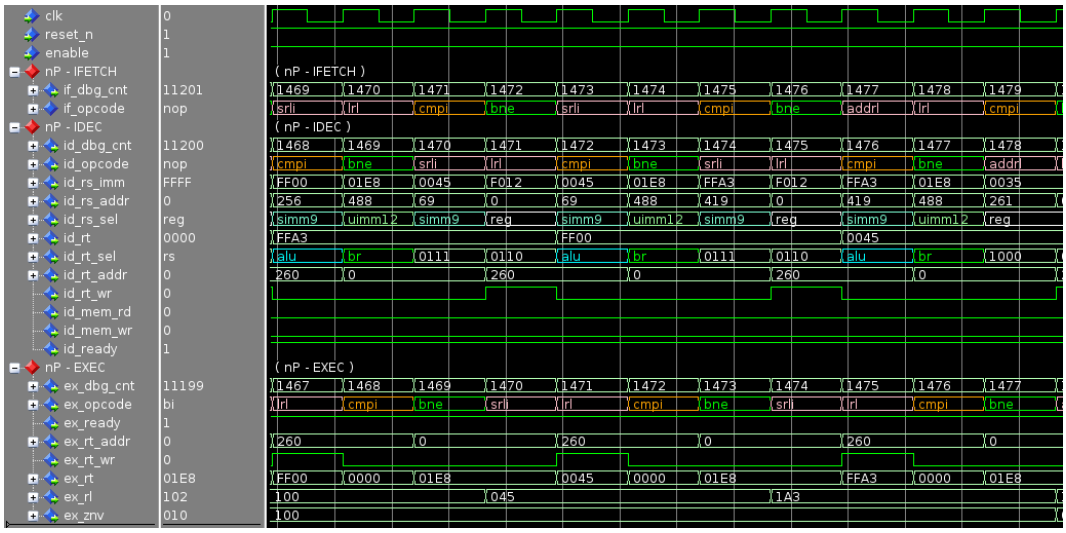


*Figura 17. SRLI - implementación lenta*

## 3.2.6 ADDRL

El comando ADDRL se utiliza para sumar el valor de RL y el contenido del registro RS, y guardar el resultado en RL. En este caso, sí que se actualizarán los flags ZNV en función del resultado.

La operación se realiza en el sumador dispuesto dentro de la ALU, en la etapa de ejecución. Para ello, es necesario tener en cuenta el tamaño de RL (9 bits). En primer lugar, se ha creado el bloque **sel_op_a**, justo antes de la entrada del sumador (de igual manera que el bloque **sel_op_b**). En este bloque se escogerá la entrada del operando 'a' del sumador entre el valor de RL y el registro RT, siendo este último la única posibilidad de entrada antes de la inclusión de estas instrucciones. Se elegirá el operando necesario en función del código de operación.

En caso de necesitar utilizar RL, como es el caso de esta instrucción, en primer lugar, se extenderá el bit de signo del registro hasta ocupar los 16 bits que requiere la entrada al sumador. Posteriormente a la realización de la suma, se reducirá el tamaño del resultado de nuevo a 9 bits, ya que ésta es la longitud requerida para poder almacenar el valor de nuevo en RL. Los flags ZNV se actualizarán en función del resultado en 9 bits, no en 16.

A continuación, en la Figura 18 (implementación rápida) y en la Figura 19 (implementación lenta) se muestra un ejemplo de funcionamiento de este nuevo comando. Las instrucciones mostradas son:

- addrl %r261 (*dbg_cnt* = 1343/1477)
- addrl %r264 (*dbg_cnt* = 1347/1481)

Siendo los valores de los registros implicados los mostrados en la Tabla 8.

| REGISTRO | VALOR INICIAL | VALOR FINAL |
|---|---|---|
| **%r261** | 0x0035 | 0x0035 |
| **%r264** | 0xF012 | 0xF012 |
| **RL** | 0x1A3 | 0x1EA |
| **ZNV** | 100 | 011 |

*Tabla 8. ADDRL - valores de los registros*

En el primer ejemplo, se realiza una suma entre un número negativo (0x1A3) y uno positivo (0x0035) dando como resultado un valor negativo (0x1D8), sin provocar *overflow*, como se refleja en los flags ZNV (010). En el segundo caso sí se produce *overflow*.

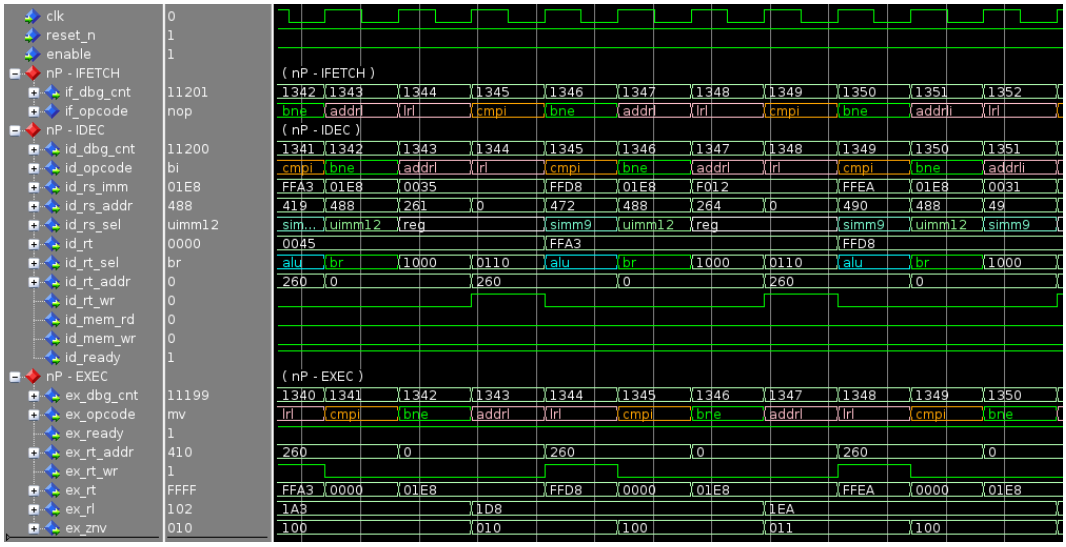


*Figura 18. ADDRL - implementación rápida*

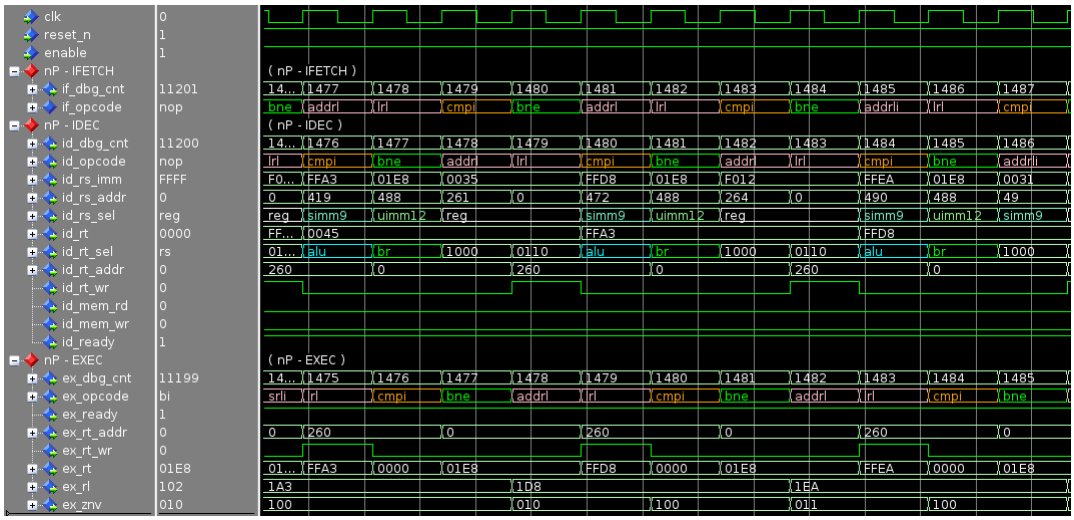


*Figura 19. ADDRL - implementación lenta*

## 3.2.7 ADDRLI

La instrucción ADDRLI sirve para sumar el valor del registro de *offset* RL y un inmediato de 9 bits, actualizando así el valor de RL. También se actualizan los flags ZNV. Como esta operación se realiza en el sumador integrado en la ALU, antes de hacerla es necesario extender tanto el signo de RL como el del inmediato.

De igual manera que en el caso anterior, el operando se escoge dentro del bloque **sel_op_a**, siendo todo el proceso completamente análogo al de la instrucción ADDRL.

En la Figura 20 y en la Figura 21 se puede observar un fragmento de la simulación en la que se representan algunas instrucciones que utilizan este comando. En concreto:

- addrli 0x031 (*dbg_cnt* = 1351/1485)
- addrli 0x0FF (*dbg_cnt* = 1355/1489)

En el primer caso, no se produce *overflow* en la suma, teniendo como resultado de RL el valor 0x01B (y los flags ZNV = 000). En cambio, en la segunda instrucción sí se produce este *overflow*, ya que se suman dos números positivos y se obtiene como resultado uno negativo (ZNV = 011).

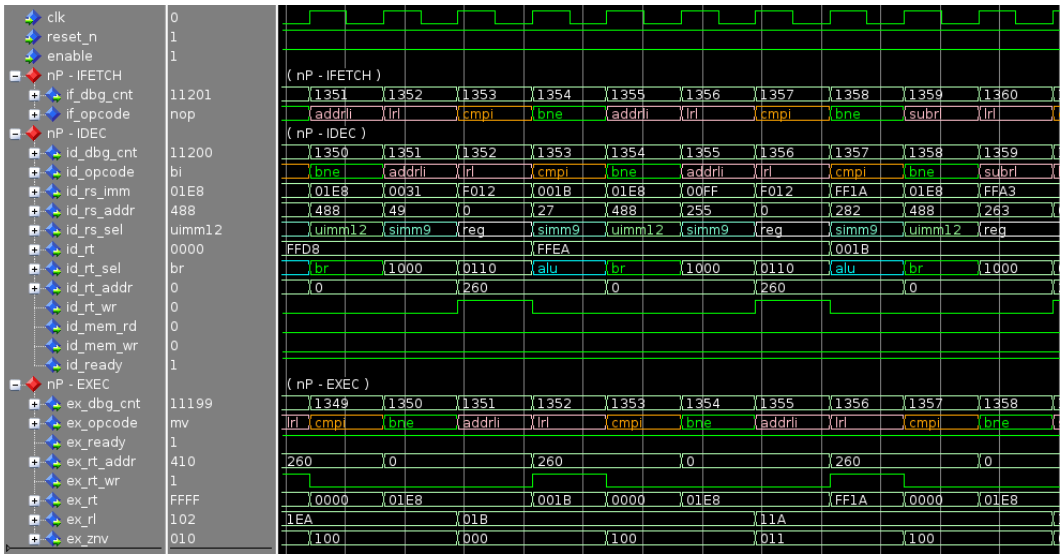


*Figura 20. ADDRLI - implementación rápida*

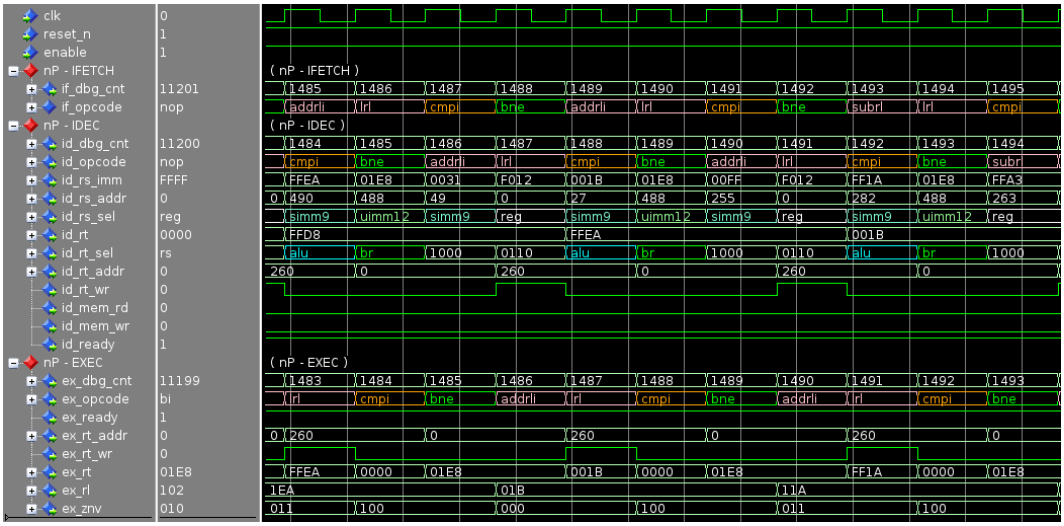


*Figura 21. ADDRLI - implementación lenta*

## 3.2.8 SUBRL

La instrucción SUBRL se utiliza para restar el contenido del registro RS al valor de RL, y guardar el resultado en RL. Además, se actualizan también los flags ZNV.

El proceso de diseño es similar al seguido para generar la instrucción ADDRL, teniendo en cuenta que en este caso se trata de una resta en lugar de una suma.

En la Figura 22 y la Figura 23 se representa la implementación rápida y lenta de esta instrucción.

- subrl %r263 (*dbg_cnt* = 1359/1493)
- subrl %r265 (*dbg_cnt* = 1363/1497)

Los valores de los registros utilizados en este ejemplo son los observados en la Tabla 9.

| REGISTRO | VALOR INICIAL | VALOR FINAL |
|---|---|---|
| **%r263** | 0xFFA3 | 0xFFA3 |
| **%r265** | 0x0088 | 0x0088 |
| **RL** | 0x11A | 0x1EA |
| **ZNV** | 100 | 001 |

*Tabla 9. SUBRL - valores de los registros*

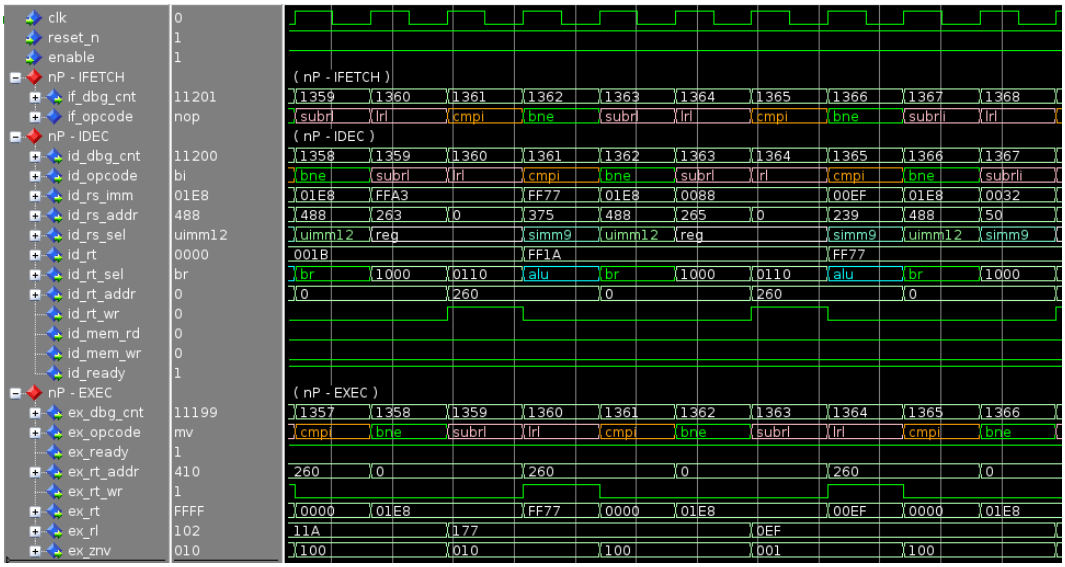


*Figura 22. SUBRL - implementación rápida*

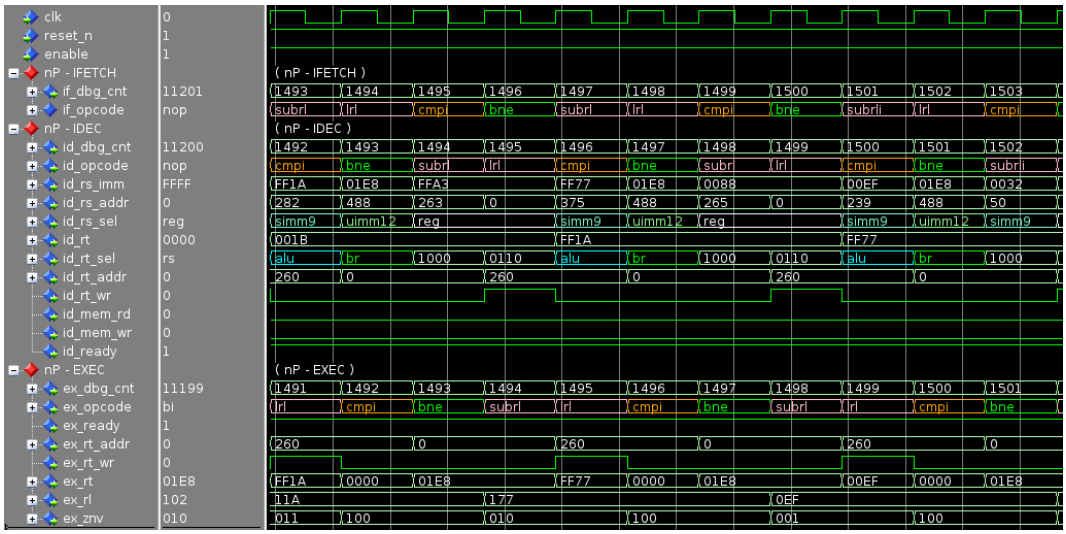


*Figura 23. SUBRL - implementación lenta*

## 3.2.9 SUBRLI

La última instrucción para la manipulación de RL es SUBRLI, utilizada para restar un inmediato de 9 bits al valor de RL y así actualizarlo. De igual manera que en las instrucciones anteriores, también se modifica el valor de los flags ZNV.

Para la creación de este comando, se han seguido los mismos pasos que para las anteriores.

En la Figura 24 y la Figura 25 se muestran las simulaciones correspondientes a la instrucción SUBRLI, tanto en implementación con lectura en doble ciclo de reloj como solamente en el ciclo de subida.

- subrli 0x032 (*dbg_cnt* = 1367/1501)
- subrli 0x1BB (*dbg_cnt* = 1371/1505)

En el primer ejemplo mostrado, la operación se realiza sin *overflow* (ZNV = 000), obteniendo como resultado un número positivo (RL = 0x0BD), mientras que en el segundo caso sí existe *overflow*, como se muestra en la señal *ex_znv*.

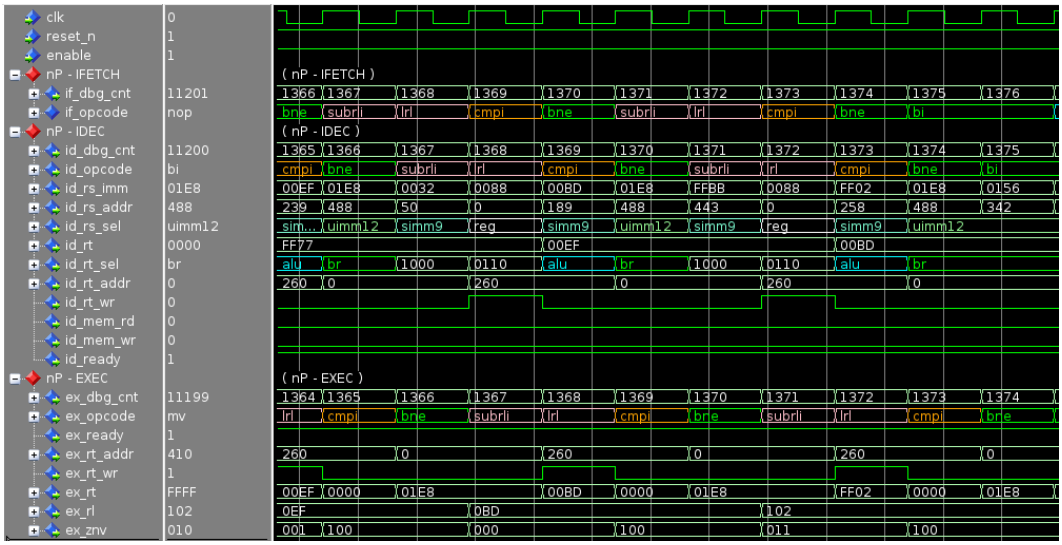


*Figura 24. SUBRLI - implementación rápida*

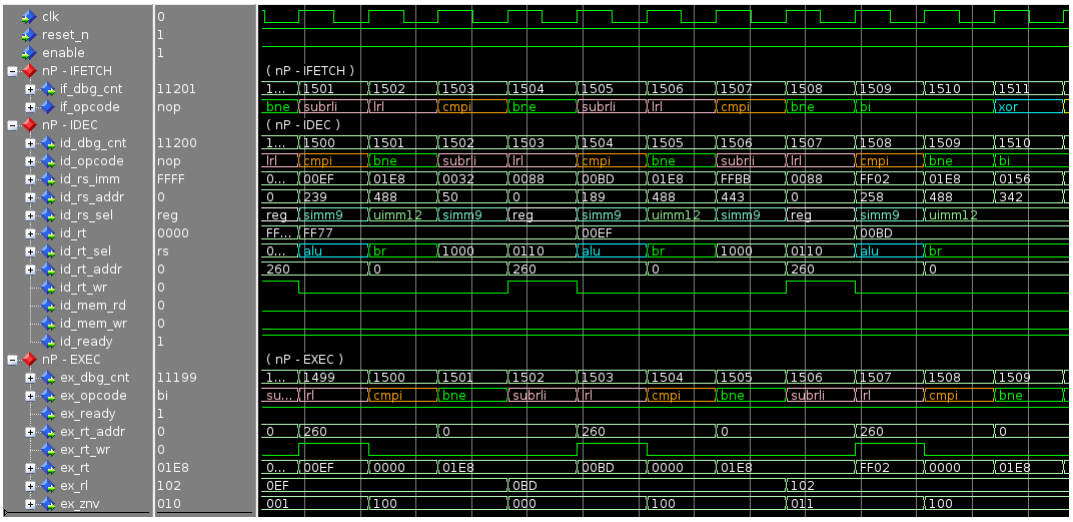


*Figura 25. SUBRLI - implementación lenta*

## 3.3 SHIFTADO

Las últimas instrucciones que se han añadido al set son las de shiftado:

- Shiftado a la izquierda: shli RT, imm4→ RT = (RT << imm4)
- Shiftado a la derecha: shri RT, imm4→ RT = (RT >> imm4)

Para diseñar estas instrucciones se ha creado el bloque **p_shift**, de tal manera que se prepara el dato antes de entrar en el proceso de **writeback**, dejándolo lo más limpio posible.

### 3.3.1 SHLI

La instrucción SHLI sirve para desplazar el contenido del registro RT hacia la izquierda tantas veces como indique el número del inmediato de 4 bits.

Es importante destacar que, para estos comandos (SHLI y SHRI), se han tomado los números como sin signo (tanto el registro RT como el inmediato de 4 bits). Esto se debe a que lo único que se pretende conseguir es el desplazar el contenido del registro (añadiendo ceros), y no multiplicarlo o dividirlo por dos. Esto se conseguiría manteniendo el signo del registro RT.

Mediante la creación de una variable de tipo *unsigned*, se asigna a ésta el resultado del desplazamiento del registro RT, pudiendo así posteriormente almacenar esta variable en una señal, transformando antes su tipo a *std_logic_vector*. Por último, en el bloque de **writeback** se escribirá el nuevo valor del registro RT.

Aunque se podría utilizar un inmediato de hasta 9 bits, se ha decidido utilizar uno de 4 bits porque este número (hasta 15) es suficiente para desplazar el registro lo máximo posible, dejándolo completamente lleno de ceros si se utiliza el máximo valor.

### 3.3.2 SHRI

El comando SHRI sirve para desplazar hacia la derecha (introduciendo ceros) el contenido del registro RT tantas veces como indique el valor del inmediato de 4 bits.

El proceso de implementación es análogo al de la instrucción SHLI.

En la Figura 26 se muestra el funcionamiento del sistema utilizando los comandos SHLI y SHRI en su implementación rápida.

***NOTA**: este test se ha realizado tras la implementación de la ampliación del tamaño de la memoria, por lo que los registros ya no tienen 16 bits, sino 32. Asimismo, el inmediato utilizado es de 5 bits en lugar de 4, ya que es lo que necesita para albergar el valor máximo de desplazamiento (31).*

En concreto, las instrucciones que se pueden observar son:

- shli %r270, 16 (*dbg_cnt* = 1281)
- shri %r270, 16 (*dbg_cnt* = 1284)

Antes de ejecutarse estas instrucciones, el valor del registro %r270 era 0x00002345. Tras producirse la primera de ellas, un desplazamiento hacia la izquierda de 16 bits, el registro pasa a tener el contenido 0x2345000, de tal manera que se han introducido 16 ceros por la derecha. Finalmente, el desplazamiento hacia la derecha de 16 bits hace que el registro vuelva a tomar su valor inicial, es decir, 0x00002345.

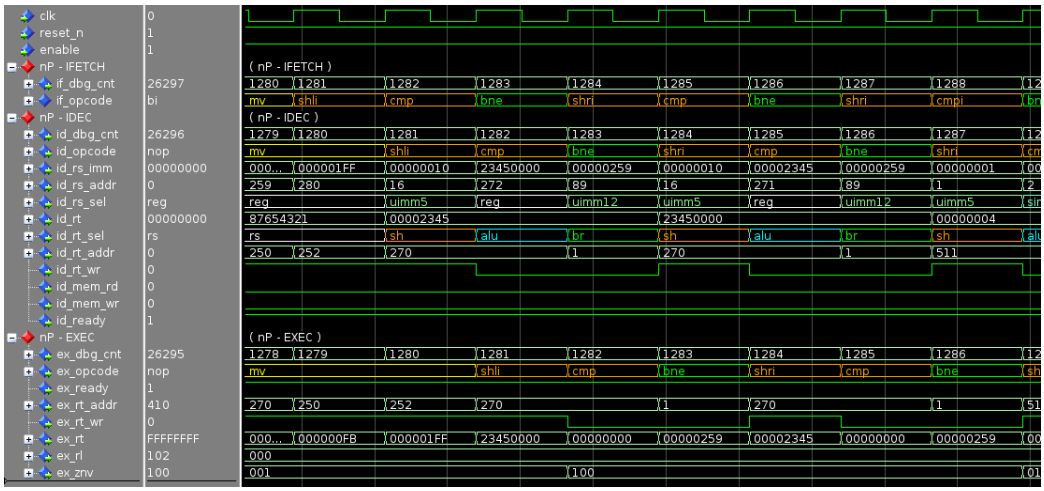


*Figura 26. SHLI y SHRI – implementación rápida*

# Capítulo 4 TAMAÑO REAL VS. TAMAÑO MÁXIMO DE MEMORIAS

Tanto la memoria de instrucción como la memoria de registros tienen un tamaño fijo. Ambas memorias son almacenamiento ya existente en la FPGA, por lo que no se pueden modificar. La memoria de instrucción es de 4k, de tal manera que se pueden cargar programas en el nanoprocesador de hasta 4096 instrucciones, necesitando para direccionarlas 12 bits. La memoria de registros puede albergar hasta 512, utilizando un total de 9 bits.

Estos tamaños estaban recogidos por las constantes *c_inst_addr_w* (entero de valor 12) y *c_reg_addr_w* (entero de valor 9), respectivamente. Estas constantes se utilizaban a lo largo de todo el diseño para definir las longitudes de los vectores de dirección.

Se ha decidido conveniente añadir una característica al dispositivo consistente en utilizar una memoria de menor tamaño a la existente. Para ello, se han sustituido las constantes arriba mencionadas por otras llamadas *c_inst_addr_max_w* y *c_reg_addr_max_w*. Aunque su valor no haya sido modificado, el significado que tienen ahora estas constantes es diferente: ya no definen el tamaño de la memoria a utilizar, sino el tamaño de la memoria que se puede utilizar como máximo.

Además, es necesario asignar el tamaño de la memoria que se desea utilizar, teniendo siempre en cuenta que éste debe ser menor o igual que la constante asociada a su tamaño máximo. Para ello, se han creado dos genéricos, cuyo valor ha de definirse en el bloque superior del proyecto, de la misma manera que los demás genéricos:

- *g_inst_addr_w*: integer range 1 to 12. Contiene el valor que se utiliza realmente para definir el tamaño de la memoria de instrucción.

- *g_reg_addr_w*: integer range 1 to 9. Contiene el valor que se utiliza realmente para definir el tamaño de la memoria de registros.

Ahora que se dispone de dos parámetros diferentes para almacenar el tamaño real y el tamaño máximo, es necesario averiguar dónde se necesita utilizar cada uno de ellos. Las entradas al dispositivo seguirán teniendo el mismo número de bits, sea cual sea el tamaño de la memoria a utilizar. Por ello, la mayoría del diseño seguirá adoptando el valor máximo (*c_inst_addr_max_w* y *c_reg_addr_max_w*) en las señales que tengan el tamaño de la dirección de una instrucción (*c_inst_addr_max_w-1 downto 0* = 12 bits) o el tamaño de la dirección de un registro (*c_reg_addr_max_w-1 downto 0* = 9 bits).

Dentro de los diagramas de bloques, las memorias vienen representadas por:

- Memoria de doble puerto **i_imem**: memoria de instrucción, dentro del bloque **i_imem** (Figura 33 de la Parte IV).
- Memoria de doble puerto **i_rt_mem**: memoria de registros RT, dentro del bloque **i_reg** (Figura 36 de la Parte IV).
- Memoria de doble puerto **i_rt_mem**: memoria de registros RT, dentro del bloque **i_reg** (Figura 36 de la Parte IV).

En estas memorias, su tamaño debe estar definido por el parámetro que contiene el tamaño real que se desea utilizar, es decir, por el genérico correspondiente.

Para no tener que modificar el exceso el código existente, se ha decidido proceder al cambio de longitud de las variables lo más cerca posible de las memorias, es decir, justo antes de la lectura de datos. De esta manera, el único lugar donde se utilizarán los genéricos será en las propias memorias.

Para poder hacer el cambio con éxito y asegurarse de que no se pierde información por el camino, hay que comprobar que los MSB que exceden del tamaño real con respecto al máximo no contengan ningún dato. Por ejemplo, suponiendo que se requiere un acceso a la memoria de instrucción con las siguientes características:

- *c_inst_addr_max_w* = 12

- *g_inst_addr_w* = 9

Y sabiendo que:

- *waddr_to_mem (c_inst_addr_max_w-1 downto 0)*: contiene el dato de la dirección de la memoria de instrucción que se quiere leer introducido por el usuario, es decir, con una longitud de 12 bits.
- *wo_addr (g_inst_addr_w-1 downto 0)*: contiene el dato de la dirección de la memoria de instrucción que se quiere leer una vez modificado su tamaño, es decir, con una longitud de 9 bits.

Hay que asegurarse de que la petición no implica una dirección cuyo tamaño en supere los 9 bits, ya que, aunque no hay nada que en principio restrinja esta petición, una posición de la memoria por encima de ese tamaño no estaría disponible. De esta manera:

- **Caso 1**: no existe conflicto en el tamaño de la dirección
  - *waddr_to_mem* = 0001 1001 0011
  - *wo_addr* = 1 1001 0011
  - No existe pérdida de información, por lo que la lectura se puede realizar sin ningún problema.
- **Caso 2**: existe conflicto en el tamaño de la dirección.
  - *waddr_to_mem* = 0101 1001 0011
  - *wo_addr* = 1 1001 0011
  - Existe pérdida de información, ya que los MSB que exceden el tamaño requerido no son ceros.

Para evitar que sucedan casos como el segundo, es necesario detectar si el valor excede el tamaño dictado por *g_inst_addr_w*. Para ello, se ha seguido el procedimiento explicado a continuación.

En caso de que este acontecimiento ocurra, se pondrá a ‘1’ un flag de error. Por ejemplo, en caso de que la falta se produjese en la dirección de escritura de la memoria de instrucción, este flag sería *mem_error_waddr*. Existe un flag de error para cada uno de los posibles fallos de este tipo, de tal manera que se pueda

detectar fácilmente el origen del mismo en caso de que alguno ocurra. Detectando el flanco de subida de estos flags, se crea una señal que permanezca a '1' durante el resto de la ejecución del programa (*int_mem_error* en el caso de la memoria de instrucción, o *reg_error* en caso de la memoria de registros). Finalmente, estas señales se extraerán hasta el nivel superior del sistema. Una vez se tengan en el nivel superior, se comprobará su valor. En caso de que alguna de ellas sea '1', el sistema se detendrá poniendo a '0' la señal de *enable*, ya que se ha intentado hacer un acceso a una posición de memoria que no existe.

Por último, es necesario añadir también la selección de los genéricos en el testbench, para poder asignar el tamaño de las memorias sin necesidad de cambiar el código interno.

# Capítulo 5 MEMORIA DE INSTRUCCIÓN EXTERNA

Otra característica requerida era la posibilidad de utilizar una memoria externa al sistema como la memoria de instrucción. De esta manera, se podrá escoger entre usar la memoria interna ya existente o una memoria externa, conectada al nanoprocesador mediante una interfaz genérica de memoria con un adaptador APB. Para ello, se incluirá un nuevo genérico en el nivel superior, llamado **g_int_inst_mem**, que:

- Cuando se desee utilizar la memoria de instrucción interna, se pondrá su valor a TRUE.
- Cuando se desee utilizar la memoria de instrucción externa, se pondrá su valor a FALSE.

Este genérico se propagará desde el nivel superior, donde se escogerá su valor, hasta la etapa de *fetch* y sus sub-bloques. En función del valor de este parámetro, se activarán las señales que se dirigen a la memoria de instrucción interna (la que ya existía previamente), o unas nuevas señales que se dirigirán a la memoria de instrucción externa, situada fuera de sistema. Esta elección se hará dentro del nuevo bloque **sel_imem**, que se puede observar en el diagrama de bloques de **i_imem_ctrl** (Figura 34).

La memoria externa (**i_ext_imem**) se creará en el *testbench* como una memoria genérica de doble puerto, conectada al nanoprocesador mediante una interfaz APB, como se puede observar en la Figura 41 de la Parte IV. A esta interfaz se conectan las señales generadas en el bloque **sel_imem**.

Como ahora existe la posibilidad de seleccionar la memoria de instrucción como una memoria externa al dispositivo, los accesos a la misma serán más lentos que

utilizando la interna; lo que antes era leer un dato de una memoria, ahora consistirá en:

- Sacar las señales fuera del nanoprocesador.
- Transformar las señales que se tienen en las necesarias para poder acceder a la memoria externa mediante la interfaz APB.
- Leer el dato requerido.
- Volver a convertir las señales en el formato adecuado para que el sistema las entienda.
- Introducir las señales de respuesta de nuevo en el nanoprocesador.

Por ello, con esta implementación se tardarán varios ciclos de reloj en leer la nueva instrucción.

Por defecto, el sistema pedirá la lectura de una nueva instrucción en cada ciclo de reloj. Si esto sucediera en el caso de estar utilizando la memoria externa, se perderían las peticiones realizadas en el intervalo en el que el sistema está esperando el dato de lectura. Por este motivo, se ha decidido crear la señal *if_ready*. Esta señal se generará en el bloque **if_ready**, dentro de **i_imem_crtl** (Figura 34 de la Parte IV). El valor de esta nueva señal variará en función del flag de ACK recibido por la interfaz de memoria genérica que conecta la memoria (a través de un adaptador APB). Esta señal afectará posteriormente en la instrucción de lectura de dato, no permitiendo que éste se realice hasta que *if_ready* = '1'. De esta manera no se perderá información en los ciclos en los que no se puede realizar una nueva lectura.

También es necesario tener en cuenta que, de igual forma que para la memoria interna, hay que controlar que la dirección de lectura no tenga un mayor tamaño que el dictado por el genérico *g_inst_addr_w*, característica añadida en el Capítulo 4. Esta comprobación se hará en el mismo bloque que en el caso de utilizar la memoria por defecto (la interna). Sin embargo, en este caso, las señales a comprobar son directamente las correspondientes a la entrada de la interfaz APB, ya que la memoria externa no comparte el *scrubbing* y la EDAC aplicados

sobre la interna. Una vez comprobado que no se sobrepasa el tamaño requerido, se pueden extraer las señales de la memoria hasta el nivel superior.

Para la implantación de esta nueva característica, se ha decidido cambiar el *testbench* del sistema. Ahora, para poder comprobar todas las combinaciones de implementación simultáneamente, sería necesario disponer de cuatro instancias del nanoprocesador:

- Implementación con lectura en doble flanco de reloj y memoria de instrucción interna.
- Implementación con lectura en doble flanco de reloj y memoria de instrucción externa.
- Implementación con lectura en flanco de reloj de subida y memoria de instrucción interna.
- Implementación con lectura en flanco de reloj de subida y memoria de instrucción externa.

Como se trataría de un *testbench* muy engorroso, y, además, la simulación de un test que verificase estos cuatro modelos a la vez sería bastante larga, se ha decidido disponer de una sola instancia del nanoprocesador. Esta instancia tendrá unos valores por defecto de implementación rápida y uso de la memoria de instrucción interna.

En lugar de un test que compruebe todas las combinaciones, se realizarán cuatro tests, cada uno de los cuales verificará el correcto funcionamiento de una de las posibilidades explicadas anteriormente. Para ello, dentro del script de características y parámetros de cada uno de los tests, se asignarán también los valores de los genéricos deseados para cada caso. Se ha decidido que lo más adecuado es la creación de cuatro tests diferentes que utilicen una misma secuencia, para no tener que simular todas las posibilidades de funcionamiento cunado sólo se desee verificar una de ellas.

Como secuencia, sea utilizado la existente hasta el momento, con las modificaciones necesarias para que se adecúe a las diferentes implementaciones.

Los aspectos más importantes a tener en cuenta a la hora de adaptar la secuencia a todos los modos de funcionamiento han sido:

- Aumento de los tiempos de simulación, ya que el uso de la memoria de instrucción externa requiere de ciclos de reloj extra para terminar de ejecutar el mismo microcódigo que la memoria interna.
- Al utilizar la memoria externa no se debe hacer la comprobación de los errores en la EDAC de la memoria de instrucción, ya que ésta sólo aplica a la memoria interna.

A continuación, en la Figura 27 y en la Figura 28 se muestra la diferencia entre el uso de la memoria de instrucción interna y externa. Debajo de las imágenes se explican las señales utilizadas para estas simulaciones, ya que no son las mismas que las que aparecen en las demás.

En la Figura 27 se puede observar el funcionamiento del sistema utilizando la memoria de instrucción interna. Como se puede comprobar, las señales asociadas a la memoria externa (las agrupadas bajo el nombre **nP – IMEM Ext**) están desactivadas. La señal *rd_int*, permanece a '1' durante toda la ejecución del microcódigo, excepto en los ciclos en los que se está esperando a que termine de realizarse el acceso a la memoria de datos externa (asociada a la instrucción SWI), a la que se accede desde la etapa de ejecución del *pipeline*. Esta señal varía su valor en función de *ex_ready* y de *id_ready*, pasando a valer '0' en caso de que alguna de estas dos señales también valga '0'.

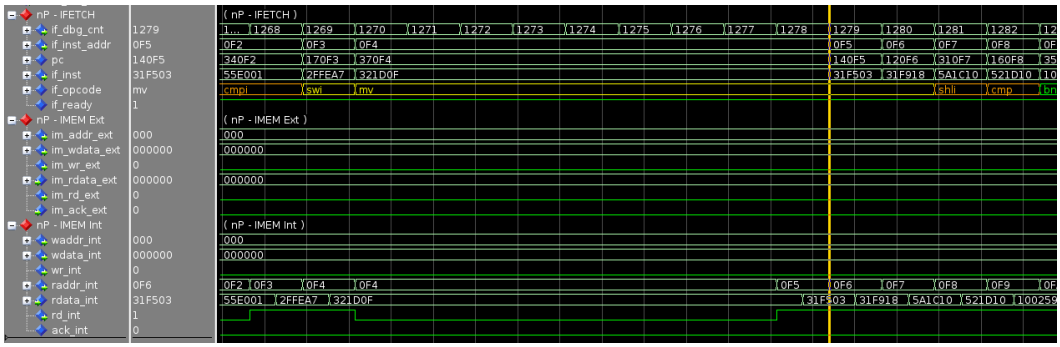


*Figura 27. Memoria de instrucción interna – implementación rápida*

En la Figura 28 se muestra la implementación del nanoprocesador con el uso de la memoria de instrucción externa. En este caso, las señales asociadas a la memoria interna son las que permanecen desactivadas (agrupadas en **nP – IMEM Int**). Se puede ver que, a diferencia del caso anterior, la señal de lectura (*im_rd_ext*) no permanece a '1' durante todos los ciclos. Cuando el flag *im_ack_ext* se pone a '1', también lo hace la señal *if_ready*. El valor de esta última afectará en este caso a la instrucción de lectura *im_rd_ext* de la misma manera que lo hacen *id_ready* y *ex_ready*. En el caso de las señales que no necesitan acceso a la memoria de datos externa, se comenzará el proceso de lectura de la instrucción en el ciclo inmediatamente posterior al flanco de subida del flag de ACK. Cuando sí se requiere este acceso, también es necesario esperar a que éste termine; por ello no se empieza la lectura en el ciclo en el que el *if_ready* se pone a '1', sino que también hay que esperar al *ex_ready*.

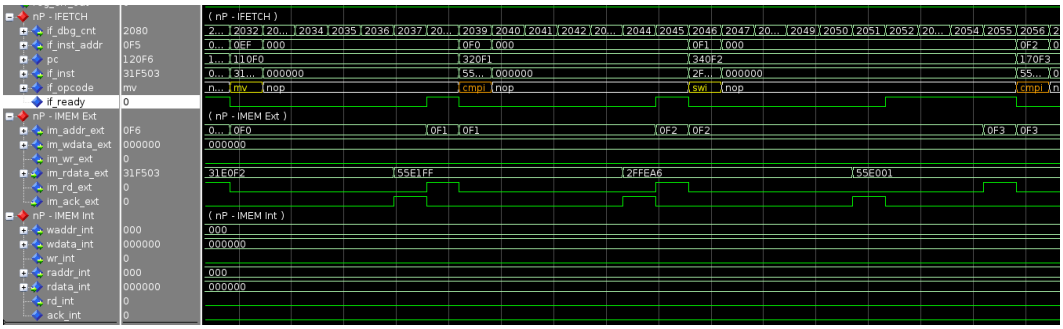


*Figura 28. Memoria de instrucción externa - implementación lenta*

Las señales utilizadas en estas simulaciones se listan y detallan en la Tabla 10.

| SEÑAL | DESCRIPCIÓN |
|---|---|
| ***if_dbg_cnt (15:0)*** | Número de ciclo de *debug* |
| ***if_inst_addr(11:0)*** | Dirección de la siguiente instrucción |
| ***pc(17:0)*** | *Program Counter* |
| ***if_inst(23:0)*** | Siguiente instrucción |
| ***if_opcode(5:0)*** | Código de operación de la instrucción que se está ejecutando |
| ***if_ready*** | Flag de ready de FETCH |
| ***im_addr_ext(11:0)*** | Memoria de instrucción externa – bus de dirección |

| | |
|---|---|
| ***im_wdata_ext(23:0)*** | Memoria de instrucción externa – bus de datos de escritura |
| ***im_wr_ext*** | Memoria de instrucción externa – comando de escritura |
| ***im_rdata_ext(23:0)*** | Memoria de instrucción externa – bus de datos de lectura |
| ***im_rd_ext*** | Memoria de instrucción externa – comando de lectura |
| ***im_ack_ext*** | Memoria de instrucción externa – flag ACK |
| ***waddr_int(11:0)*** | Memoria de instrucción interna – bus de dirección de escritura |
| ***wdata_int(23:0)*** | Memoria de instrucción interna – bus de datos de escritura |
| ***wr_int*** | Memoria de instrucción interna – comando de escritura |
| ***raddr_int(11:0)*** | Memoria de instrucción interna – bus de dirección de lectura |
| ***rdata_int(23:0)*** | Memoria de instrucción interna – bus de datos de lectura |
| ***rd_int*** | Memoria de instrucción interna – comando de lectura |
| ***ack_int*** | Memoria de instrucción interna – flag ACK |

*Tabla 10. Señales simulación - memoria de instrucción externa*

Por último, como se han creado nuevos tests, también es necesario crear los archivos de síntesis correspondientes a cada uno de ellos.

# Capítulo 6 AMPLIACIÓN DEL TAMAÑO DE LA MEMORIA

Hasta ahora, los registros utilizados tenían una longitud de 16 bits. Se pretende que se puedan manejar registros de 32 bits, por lo que se requiere realizar algunos cambios en el sistema. La cantidad de registros seguirá siendo la misma, por lo que no es necesario aumentar el tamaño de la dirección de los mismos, sino solamente su contenido.

Por otro lado, también se modificará el tamaño de la memoria de datos externa en el *testbench*, pasando a tener un tamaño de $2^{32}$ en lugar de $2^{16}$, como era hasta ahora.

Para efectuar este cambio, en primer lugar, hay que tener en cuenta las instrucciones que necesitan acceso a un bit concreto, o que necesitan conocer el número de bits que tiene un registro. Estas instrucciones son:

- **Instrucciones de desplazamiento**: SHLI, SHRI.
- **Instrucciones de acceso directo a un bit**:
    - Comprobar el valor de un bit: BTEST, BTESTI
    - Poner un bit a ‘1’: BSET, BSETI
    - Poner un bit a ‘0’: BRST, BRSTI

Las instrucciones de desplazamiento necesitan un inmediato que pueda albergar un número de hasta la longitud de los registros, ya que ese será el número máximo de posiciones que se puede desplazar un registro. Ahora, en lugar de utilizar un inmediato de 4 bits, utilizarán uno de 5. De esta manera, se podrán desplazar los registros hasta 31 posiciones.

Las instrucciones de acceso directo a un bit pueden ser de dos tipos:

- Aquellas que utilizan el contenido de un registro RS para indicar el bit al que se quiere acceder (BTEST, BSET, BRST).
- Aquellas que utilizan un inmediato para indicar el bit al que se quiere acceder (BTESTI, BSETI, BRSTI).

En el primero de estos casos, simplemente hay que tener en cuenta que ahora el contenido del registro será válido si almacena un número menor de 32, en lugar de menor de 16. En el segundo caso, las consideraciones necesarias son similares a las comentadas para las instrucciones de desplazamiento: como se requiere que el inmediato pueda acceder a todas las posiciones, se tendrá que utilizar uno de 5 bits en lugar de 4. Todos los comandos, así como su implementación y una breve descripción de los mismos, se muestran en su versión final en la Parte V de este documento.

Otro punto a tener en cuenta es que habrá que modificar el tamaño de los registros a lo largo de todo el código. Como durante todo el proceso de diseño se han utilizado constantes y genéricos para definir este tipo de parámetros, el único cambio necesario será el de la constante que almacena este valor, es decir, *c_data_w*.

Por otro lado, también hay que modificar el *testbench*. Inicialmente existía una memoria de datos externa de $2^{16}$, es decir, con una capacidad para almacenar 65536 valores. Ahora se quiere que esta memoria pueda almacenar hasta $2^{32}$ registros, por lo que es necesario aumentar su tamaño.

En la verificación del sistema no es necesario rellenar todas las posiciones de la memoria externa para comprobar el correcto funcionamiento. Por ello, se ha decidido reducir la carga computacional del proceso de verificación utilizando dos memorias de $2^{16}$ cada una en lugar de una de $2^{32}$. De esta manera, una de ellas representará la parte más baja de la memoria y la otra la parte alta, no disponiendo de las posiciones intermedias de la misma. Esto no es un problema, puesto que se dispone de espacio suficiente para almacenar los datos necesarios a la hora de ejecutar los tests. La implementación final del *testbench* puede observarse en la Figura 41 de la Parte IV.

# Capítulo 7 CONCLUSIONES Y FUTUROS DESARROLLOS

Durante el desarrollo del proyecto se han realizado diversos cambios a un nanoprocesador, que se encarga de procesar microinstrucciones. Las funciones añadidas al mismo, cumpliendo así con los objetivos propuestos para este proyecto, son las comentadas a continuación.

En primer lugar, se ha ampliado el juego de instrucciones que el nanoprocesador conoce. De esta manera, además del set inicial, ahora es capaz de procesar instrucciones referenciadas por punteros, tanto con *offset* como sin él, así como las instrucciones necesarias para poder acceder y operar sobre el registro que almacena este *offset*. También se han añadido instrucciones para poder desplazar el contenido de los registros.

Por otro lado, se ofrece una mayor versatilidad de uso del sistema que la existente anteriormente. Tras la finalización del proyecto, es posible variar el tamaño de las memorias de los registros y de instrucción, así como escoger entre utilizar la memoria de instrucción acoplada dentro del nanoprocesador o conectar una memoria externa para utilizar con este propósito. Por último, se ha modificado el tamaño de los registros, de tal manera que se puedan utilizar registros del doble de longitud que la que se ofrecía inicialmente.

La interfaz del nivel superior del nanoprocesador es muy similar a la que se tenía al comenzar el proyecto, por lo que no se ha añadido complejidad al uso del sistema. La única modificación significativa es la existencia de la interfaz de conexión de la memoria de instrucción externa, como se puede observar en la Figura 29.

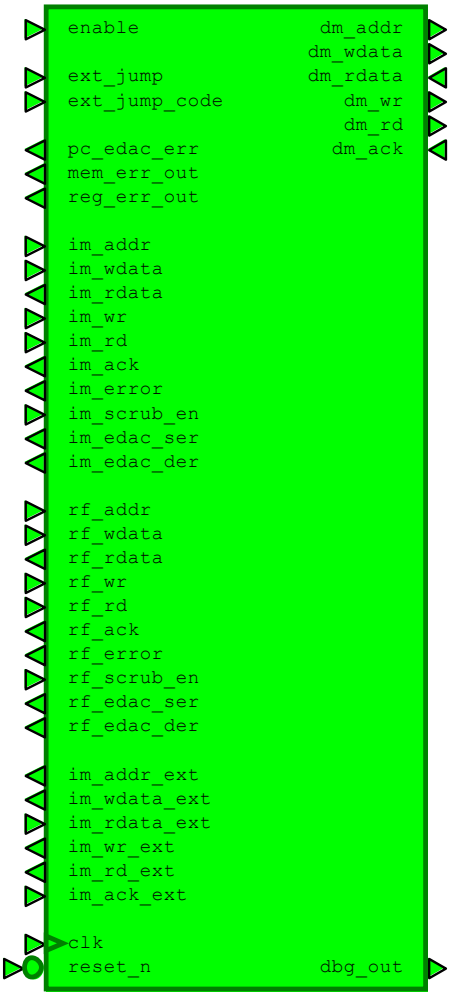


*Figura 29. Interfaz I/O del nP – final*

Sin embargo, sí se han producido cambios relevantes en la estructura interna del mismo, como pueden ser la existencia de un registro que almacena el valor de RL (registro de *offset*) o el cambio de longitud de gran parte de las señales internas.

Por otra parte, también se ha modificado significativamente el *testbench* del sistema, en el que existe solamente una instancia del nanoprocesador. De esta manera, los procesos de verificación serán más rápidos cuando solamente se requiera revisar una configuración concreta del sistema. Así, se logra reducir la carga computacional del sistema con respecto a comprobar todas las combinaciones simultáneamente.

En avances futuros de este producto, se pretende proveer a la máquina de una mayor potencia, ofreciendo la posibilidad de realizar las operaciones de la ALU en punto flotante. También se diseñará un *wrapper* APB que recoja el sistema completo, así como un *wrapper* AHB, añadiendo un nivel más a la jerarquía y proveyendo al nanoprocesador de más posibles futuras aplicaciones gracias a la versatilidad ofrecida.

# Capítulo 8 REFERENCIAS

[1] «Crisa,» [En línea]. Available: http://www.crisa.es/.

[2] «Microcódigo,» [En línea]. Available: http://zbc.uz.zgora.pl/Content/27955/Remigiusz_Wisniewski_Synthesis_of_CMCUs_for_Programmable_Devices.pdf.

[3] «PPU NG HKISeq Module Datasheet».

[4] «HDL Designer,» [En línea]. Available: https://www.mentor.com/products/fpga/hdl_design/hdl_designer_series/.

[5] «Questa Sim,» [En línea]. Available: https://www.mentor.com/products/fv/questa/

[6] «Synplify Pro,» [En línea]. Available: https://www.synopsys.com/implementation-and-signoff/fpga-based-design/synplify-pro.html.

[7] K. B. W. Young, «Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage,» 1998.

# *Parte II PRESUPUESTO*

# Capítulo 1 MEDICIONES

En este apartado se listan todos los aspectos del proyecto que implican una aportación económica, así como la cantidad de la que se ha requerido para la consecución de los objetivos.

## *1.1 MATERIAL*

En la Tabla 11 se incluyen todos los costes relacionados con la compra de utensilios necesarios para realizar el proyecto.

| MATERIAL | UNIDADES |
|---|---|
| Licencia HDL Designer | 1 |
| Licencia Mentor Questa Sim | 1 |
| Licencia Synplify Pro | 1 |

*Tabla 11. Mediciones. Material*

## *1.2 INGENIERÍA*

A continuación, en la Tabla 12, se presentan las tareas de ingeniería realizadas y las horas necesarias para su consecución.

| TAREA | HORAS |
|---|---|
| Comprensión del proyecto existente | 70 |
| Modificación del código VHDL | 200 |
| Verificación por simulación | 100 |
| Code coverage | 50 |
| Síntesis | 80 |
| Escritura de la memoria | 50 |

*Tabla 12. Mediciones. Ingeniería*

# Capítulo 2 PRECIOS UNITARIOS

En este apartado se listan los precios unitarios del material necesario para la realización del proyecto, así como del precio por hora para cada tipo de tarea realizada.

## 2.1 MATERIAL

En cuanto a los materiales, los precios de las licencias de los programas ofrecidos a Airbus son información confidencial, por lo que se ha decidido aplicar al presupuesto el precio que tendrían estas licencias compradas individualmente.

| MATERIAL | PRECIO (€/ud) |
|---|---|
| Licencia HDL Designer | 9.600 |
| Licencia Mentor Questa Sim | 24.000 |
| Licencia Synplify Pro | 19.000 |

*Tabla 13. Precios unitarios. Material*

## *2.2 INGENIERÍA*

| TAREA | PRECIO (€/h) |
|---|---|
| Comprensión del proyecto existente | 40 |
| Modificación del código VHDL | 50 |
| Verificación por simulación | 50 |
| *Code coverage* | 50 |
| Síntesis | 50 |
| Escritura de la memoria | 40 |

*Tabla 14. Precios unitarios. Ingeniería*

# Capítulo 3 PRESUPUESTOS PARCIALES

A continuación, se multiplican las mediciones por el precio unitario de cada uno de los elementos. De esta forma se obtiene el presupuesto de cada una de las secciones utilizadas en los apartados anteriores.

## *3.1 MATERIAL*

| MATERIAL | UNIDADES | PRECIO (€/ud) | TOTAL (€) |
|---|---|---|---|
| Licencia HDL Designer | 1 | 9.600 | 9.600 |
| Licencia Mentor Questa Sim | 1 | 24.000 | 24.000 |
| Licencia Synplify Pro | 1 | 19.000 | 19.000 |
| **Total** | | | **52.600** |

*Tabla 15. Presupuestos parciales. Material*

## *3.2 INGENIERÍA*

| TAREA | HORAS | PRECIO (€/h) | TOTAL (€) |
|---|---|---|---|
| Comprensión del proyecto existente | 70 | 40 | 2.800 |
| Modificación del código VHDL | 200 | 50 | 10.000 |
| Verificación por simulación | 100 | 50 | 5.000 |
| *Code coverage* | 50 | 50 | 2.500 |
| Síntesis | 80 | 50 | 4.000 |
| Escritura de la memoria | 50 | 40 | 2.000 |
| **Total** | | | **26.300** |

*Tabla 16. Presupuestos parciales. Ingeniería*

# Capítulo 4 PRESUPUESTO GENERAL

Por último, se presenta el presupuesto general como la suma de todas sumas parciales obtenidas en el apartado anterior. Además, se incluirán los gastos generales, que abarcarán todo lo que no se ha contenido en ninguna sección, pero también debe ser tenido en cuenta (luz, mantenimientos, material de oficina, etc.). También se incluirá el beneficio industrial, así como el IVA.

| APARTADO | SUMA PARCIAL (€) |
|---|---|
| Materiales | 52.600 |
| Ingeniería | 26.300 |
| Suma | 78.900 |
| Gastos generales (13%) | 10.257 |
| Beneficio industrial (6%) | 4.734 |
| Subtotal | 93.891 |
| IVA (21%) | 19.717,11 |
| **Total** | **113.608,11** |

*Tabla 17. Presupuesto general*

# *Parte III MANUAL DE USUARIO*

# Capítulo 1 CONFIGURACIÓN DEL DISPOSITIVO

Al menos los siguientes requerimientos de configuración deben de ser incluidos por el usuario:

- **Frecuencia**: considerando las limitaciones dadas por la síntesis del nanoprocesador.
- **Genéricos**: decidir los valores de todos los genéricos listados en la Tabla 18.

| GENÉRICO | TIPO | DESCRIPCIÓN | VALOR POR DEFECTO |
|---|---|---|---|
| ***g_fast_exec*** | booleano | Si es TRUE: se usa el flanco de bajada del reloj para leer las memorias internas (doble flanco). | TRUE |
| ***g_ext_jump_code_w*** | entero [1 a 4] | Tamaño del código de salto externo | 2 |
| ***g_inst_addr_w*** | entero [1 a 12] | Tamaño efectivo de la memoria de instrucción. El máximo es 4k microinstrucciones. | 12 |
| ***g_reg_addr_w*** | entero [1 a 9] | Tamaño efectivo de la memoria de registros. El máximo es 512 registros. | 9 |
| ***g_int_inst_mem*** | booleano | Si es TRUE: se usa la memoria de instrucción interna. Si es FALSE: se una la memoria de instrucción externa. | TRUE |

*Tabla 18. Genéricos*

## *1.1 SEÑAL* ENABLE

Mientas la señal de *enable* esté activa, el programa funcionará con normalidad. Cuando esta señal se desactive, el programa se pausará (no se reseteará); esto

quiere decir que el valor del contador del programa (PC) se mantendrá hasta que se vuelva a activar el *enable*.

Esto permite al usuario ejecutar el código microinstrucción a microinstrucción, generando secuencias de pulsos en el *enable*.

El usuario debe garantizar un espacio mínimo entre los flancos de bajada de la señal de *enable* para dar tiempo al nanoprocesador de terminar el ciclo completo de pipeline (preparación, decodificación y ejecución de una microinstrucción). Este tiempo debe ser de tres veces el retardo máximo de los buses de datos externos.

## *1.2 SEÑAL* EXT_JUMP *(PULSO)*

Cuando esta señal esté activa, y estando el nanoprocesador funcionando con normalidad (*enable* = '1'), el puntero del programa saltará a una de las primeras $2^{g\_ext\_jump\_code\_w}$ líneas de microcódigo (seleccionada por la señal de entrada *ext_jump_code*). Estas líneas están reservadas específicamente para este propósito, y deben contener un salto incondicional a alguna rutina creada por el usuario.

## *1.3 BUS* DBG_OUT

La señal *dbg_out* es la concatenación de las señales internas mostradas en la Tabla 19.

| SEÑAL | ANCHO | POSICIÓN | DESCRIPCIÓN |
|---|---|---|---|
| ***ex_rt_wr*** | 1 | (46) | RT write command |
| ***ex_opcode*** | 6 | (45:40) | Operation code |
| ***ex_rt_addr*** | 9 | (39:31) | RT write address bus |
| ***ex_rt*** | 32 | (30:15) | RT write data |
| ***ex_znv*** | 3 | (14:12) | 'ZNV' flags register |

| | | | |
|---|---|---|---|
| ***ex_inst_addr*** | 12 | (11:0) | Next nanoInstruction address |

*Tabla 19. dbg_out - señales concatenadas*

Esta señal solo tiene propósitos informativos para el proceso de *debug*. Puede ser una salida abierta cuando no se utilice y puede ser eliminada por la herramienta de síntesis.

## *1.4 SEÑALES* IM_*(_EXT), RF_* *AND* DM_*

Estas señales son las correspondientes a las interfaces de memoria externa:

- *im_** y *rf_**: estas interfaces se utilizan para rellenar la memoria de instrucción y la memoria de registros respectivamente con el microcódigo que se desea ejecutar y los correspondientes valores iniciales de los registros internos.
- *im_*_ext*: cuando el genérico *g_int_inst_mem* tenga como valor FALSE, el nanoprocesador leerá el microcódigo de una memoria externa a través de esta interfaz. Debido al retardo de los accesos a la memoria externa a través de un bus compartido, la ejecución del microcódigo será mucho más lenta. Todas las microinstrucciones esperarán al flag de ACK de la lectura (*im_ack_ext*).
- *dm_**: interfaz de memoria genérica usada para la ejecución del microcódigo para leer y guardar datos.

# Capítulo 2 FLUJO DE DISEÑO DEL MICROCÓDIGO

## *2.1 ESTRUCTURA*

El microcódigo es un fichero de texto en ensamblador (*.asm) con una estructura predefinida.

- Declaración de constantes y registros:

```
        .header
; constants
c_zero:           .equ 0x00000000
c_one_n:          .equ 0xFFFFFFFF
addr_1:           .equ 0x00000010
; registers
addr_1_reg:       .reg %r65  addr_1
data_1_reg:       .reg %r211 0x00000000
rs_math_one_n:    .reg %r245 c_one_n
rt_error:         .reg %r410 c_zero
```

- Secuencia de microinstrucciones:

```
; main sequence
        .data
        .text

; first lines MUST contain uncontitional brances to
"ext_jump"
; related routines:
        ; #1 : main routine
        bi p_main
        ; #2 : emergency stop
        bi e_end_error_loop
        ...

p_main:
        lw data_1_reg, addr_1_reg
        lwi data_1_reg, addr_1
        addi addr_1_reg, 0x1000
        lw data_1_reg, addr_1_reg
        ...

e_end_error_loop:
        ; Endless loop after an error:
        mv rt_error, rs_math_one_n     ; ERROR : %r410 =
0xFFFFFFFF
        bi e_end_error_loop
```

- Final del fichero:

```
; finish
        .end
```

## *2.2 ENSAMBLAJE*

EL **CRIP nProc** viene con su propio programa de ensamblaje, escrito en C, FLEX y BISON. Utiliza el archivo *.asm como una entrada y genera un set de ficheros:

- **result.obj**: recoge el contenido de la memoria de instrucción en dos columnas, en las que se muestran los pares de dirección y dato (12 y 24 bits respectivamente):

```
0x000  0x080003
0x001  0x080490
0x002  0x080085
0x003  0xa61200
0x004  0x0c0007
0x005  0x33ff14
...    ...
```

- **result.reg**: recoge el contenido de la memoria de registros en dos columnas, en las que se muestran los pares de dirección y dato (9 y 32 bits respectivamente):

```
0x000  0x7fffffff
0x001  0x87654321
0x002  0x80000000
0x040  0xffffffff
0x00b  0x00000000
...    ...
```

- Otros archivos auxiliares, incluyendo el microcódigo con anotaciones.

# Capítulo 3 OPERACIÓN NOMINAL

El funcionamiento nominal del nanoprocesador consta de los siguientes pasos:

- Inicializar la memoria de instrucción interna y la memoria de registros, todo con ceros, para evitar errores en la EDAC cuando se realicen los ciclos de *scrubbing* en posiciones de memoria no utilizadas.
- Cargar el microcódigo en la memoria de instrucción.
- Cargar la memoria de registros con los valores iniciales correspondientes.
- Activar la señal de *enable*.

# Capítulo 4 REQUERIMIENTOS DE ARQUITECTURA EXTERNOS

## *4.1 REGISTRO EXTERNO DE LOS FLAGS DE ERROR*

Las señales de error (*pc_edac_err*, *mem_error* y *reg_error*) son salidas combinacionales, por lo que se deben de registrar fuera del **CRIP nProc**. Cuando se da alguno de estos errores, el control hardware externo debe desactivar la señal de *enable* para que se detenga la ejecución.

## *4.2 CONTROL DE* SCRUBBING *DE LAS MEMORIAS INTERNAS*

El *scrubbing* de las memorias sólo es una característica accesible cuando el nanoprocesador está desactivado. Por este motivo, la señal de *enable* debe de desactivarse periódicamente para activar el proceso de *scrubbing* y así garantizar la protección de las memorias.

Cuando las señales *im/rf_scrub_en* emiten un pulso, se leerá un grupo de 8 palabras consecutivas, empezando por la posición de memoria más alta y descendiendo, para así verificar la redundancia de la EDAC.

Los errores simples se reportarán a través de la salida im/rf_edac_ser, y la palabra correspondiente se actualizará con el valor correcto.

Los errores dobles se reportarán a través de la salida im/rf_edac_der. En este caso, el nanoprocesador no debe activarse de nuevo, ya que el contenido de la memoria es incorrecto y no se puede corregir.

Las palabras que no contengan errores (o que contengan múltiples errores indetectables) no se pueden corregir.

La Figura 30 muestra un ciclo de *scrubbing*:

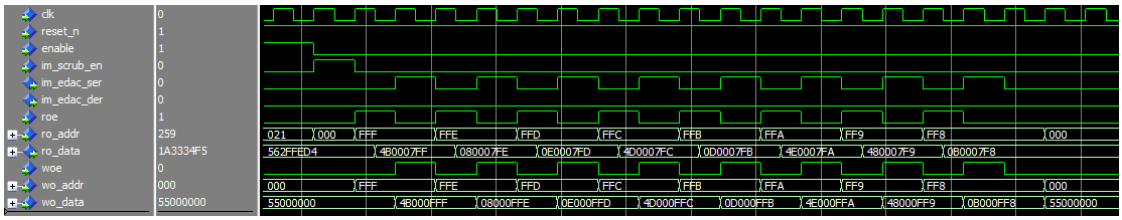


*Figura 30. Ciclo de* scrubbing *con errores de EDAC simples*

# *Parte IV DIAGRAMAS DE BLOQUES*

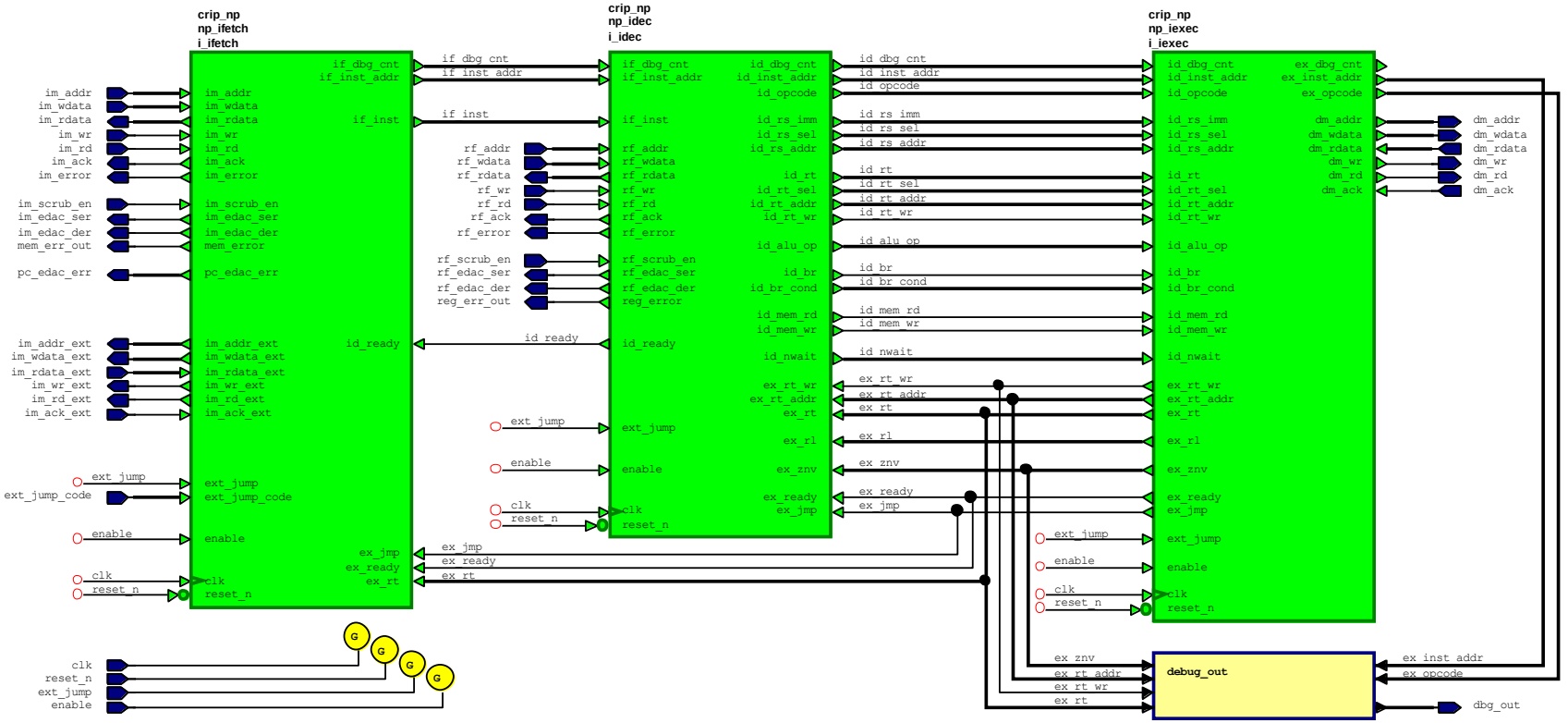


*Figura 31. crip_np - diagrama de bloques*

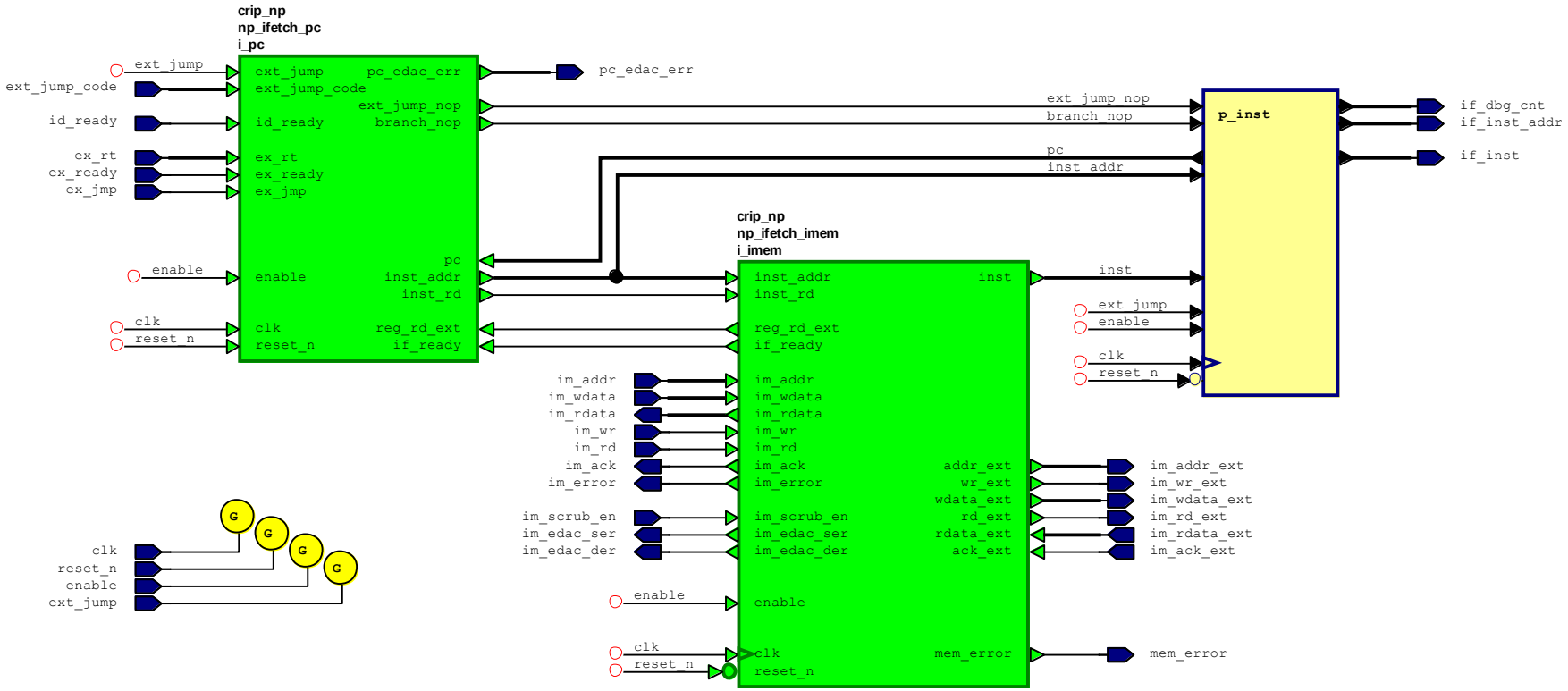


*Figura 32. i_ifetch - diagrama de bloques*

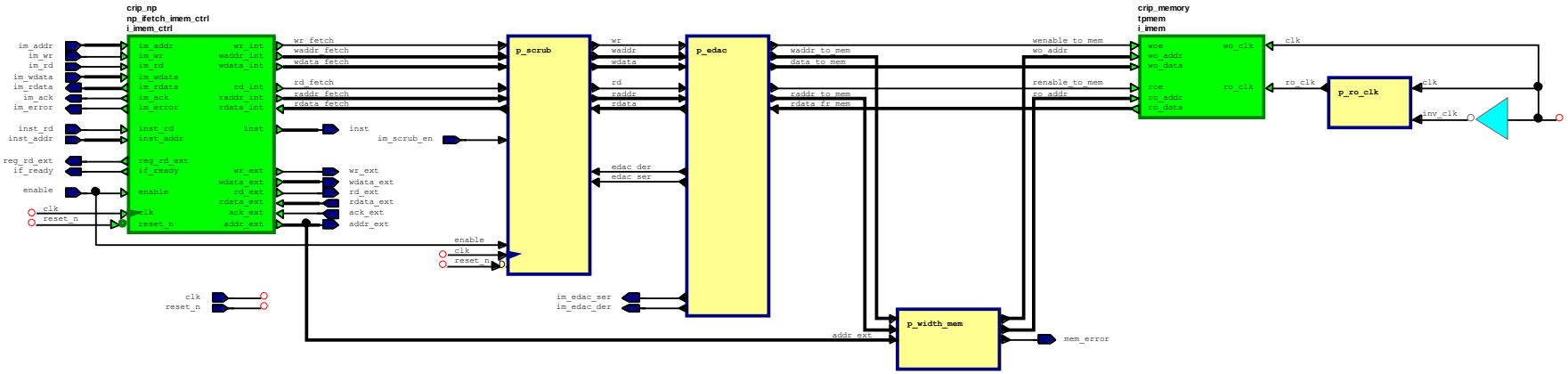


*Figura 33.i_imem – diagrama de bloques*

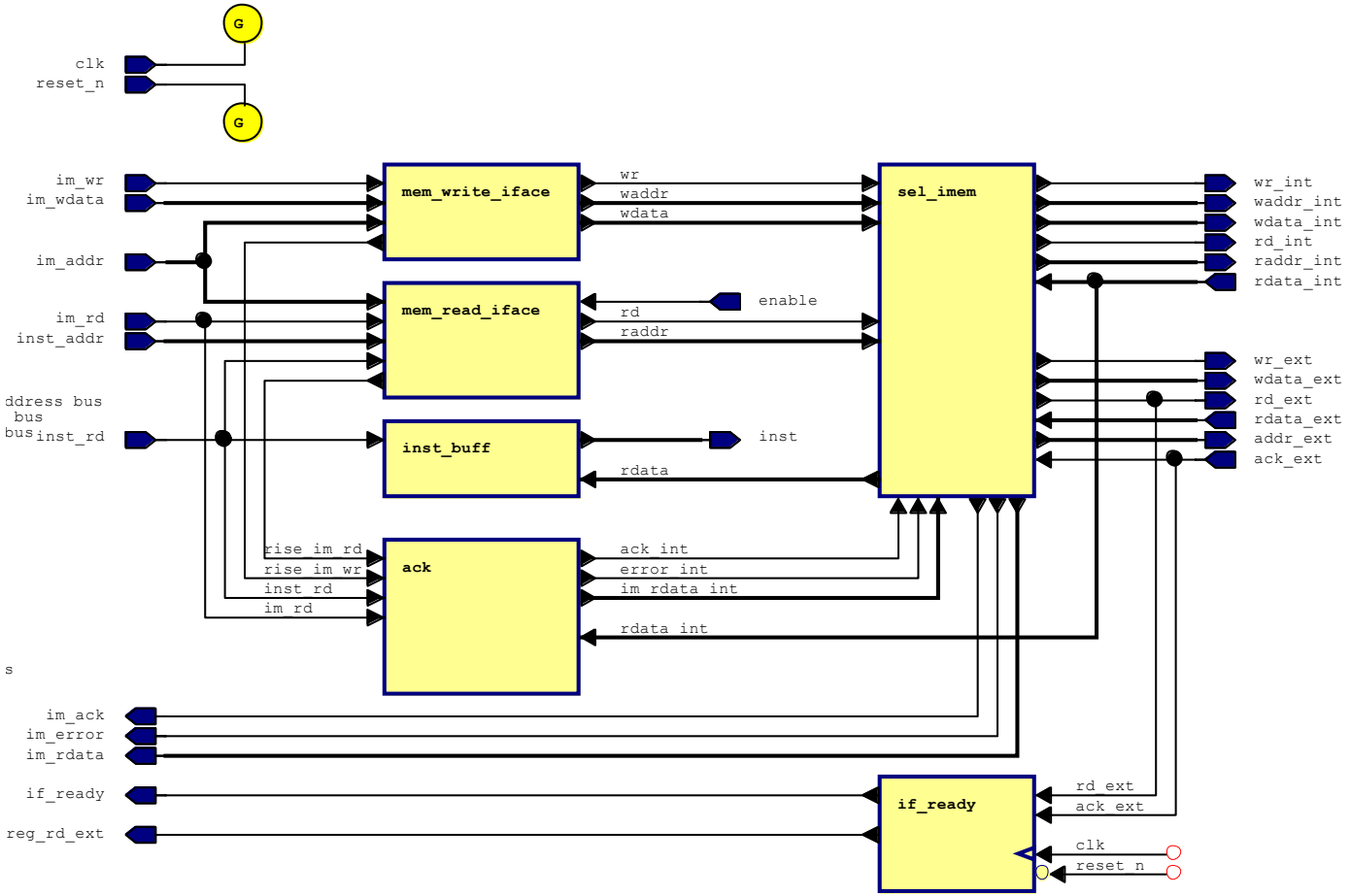


*Figura 34. i_imem_ctrl - diagrama de bloques*

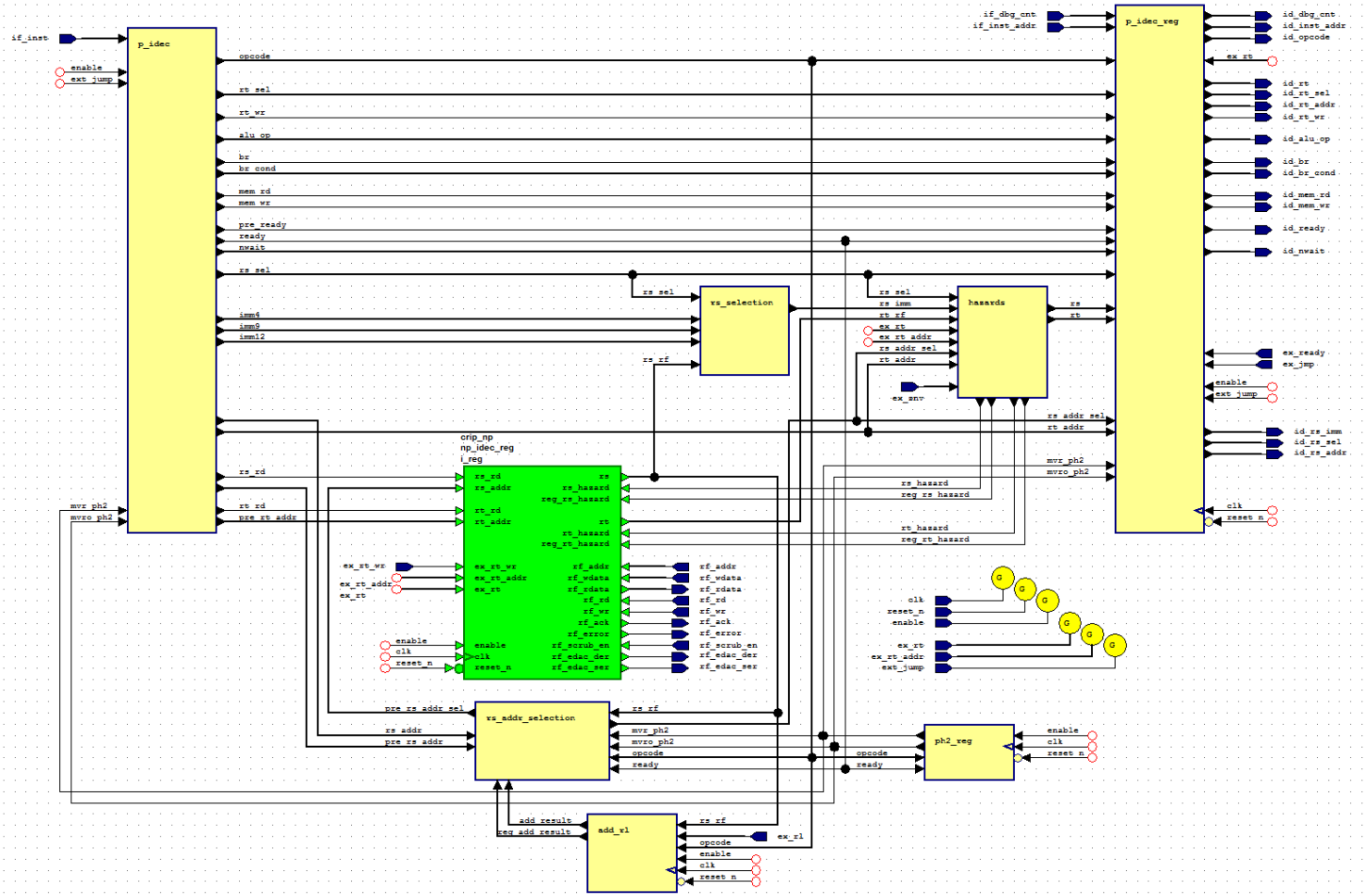


*Figura 35. i_idec - diagrama de bloques*

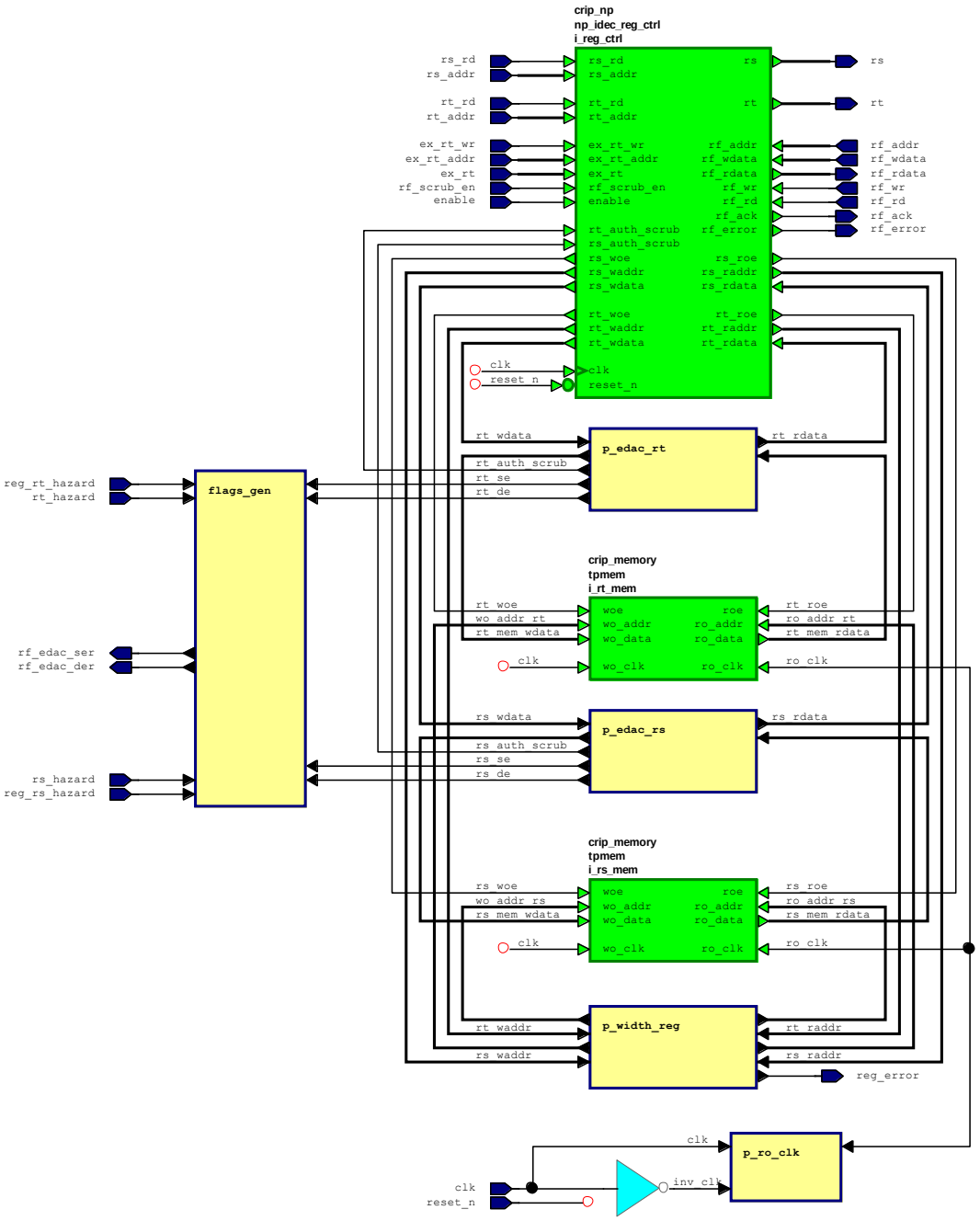


*Figura 36. i_reg - diagrama de bloques*

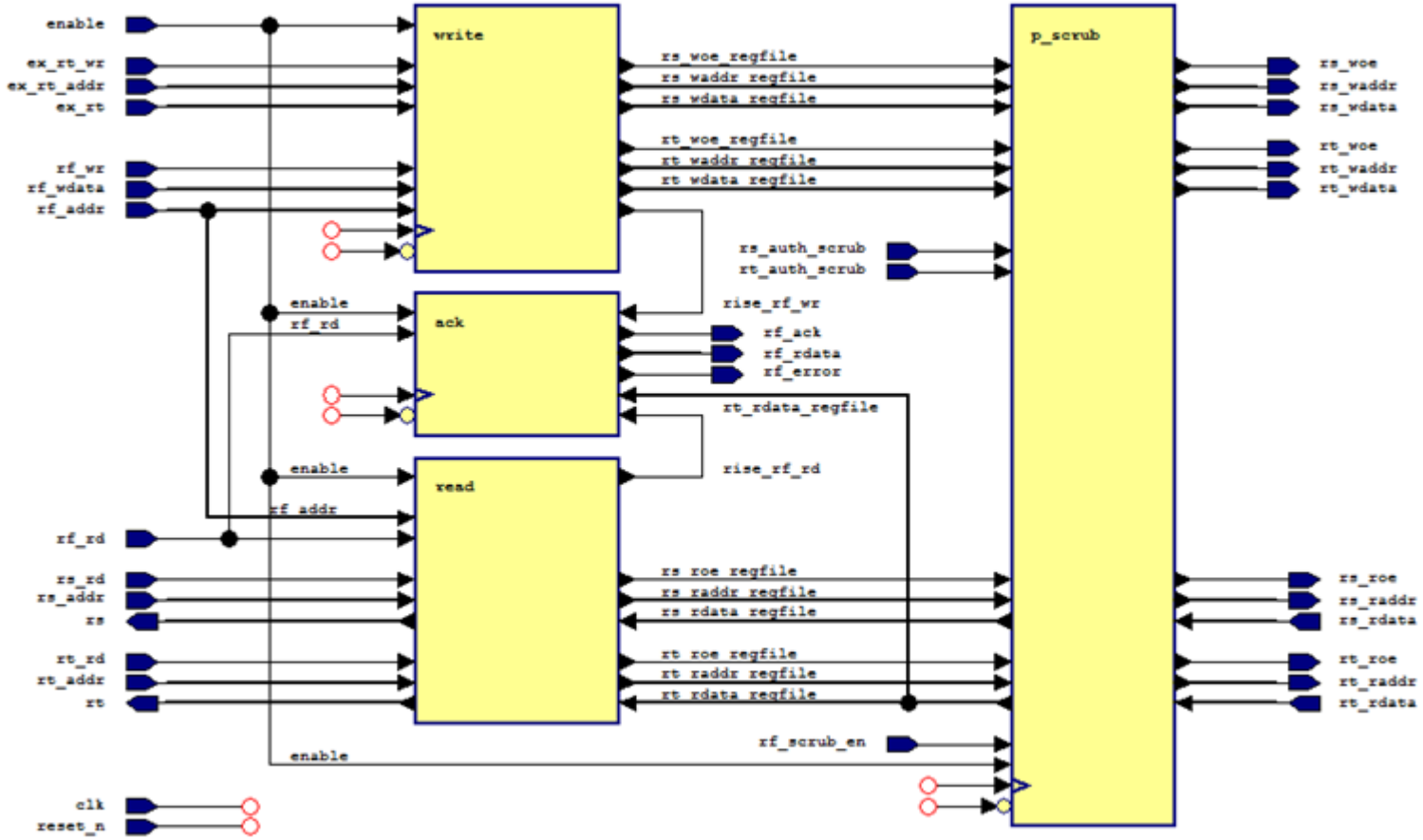


*Figura 37. i_reg_ctrl - diagrama de bloques*

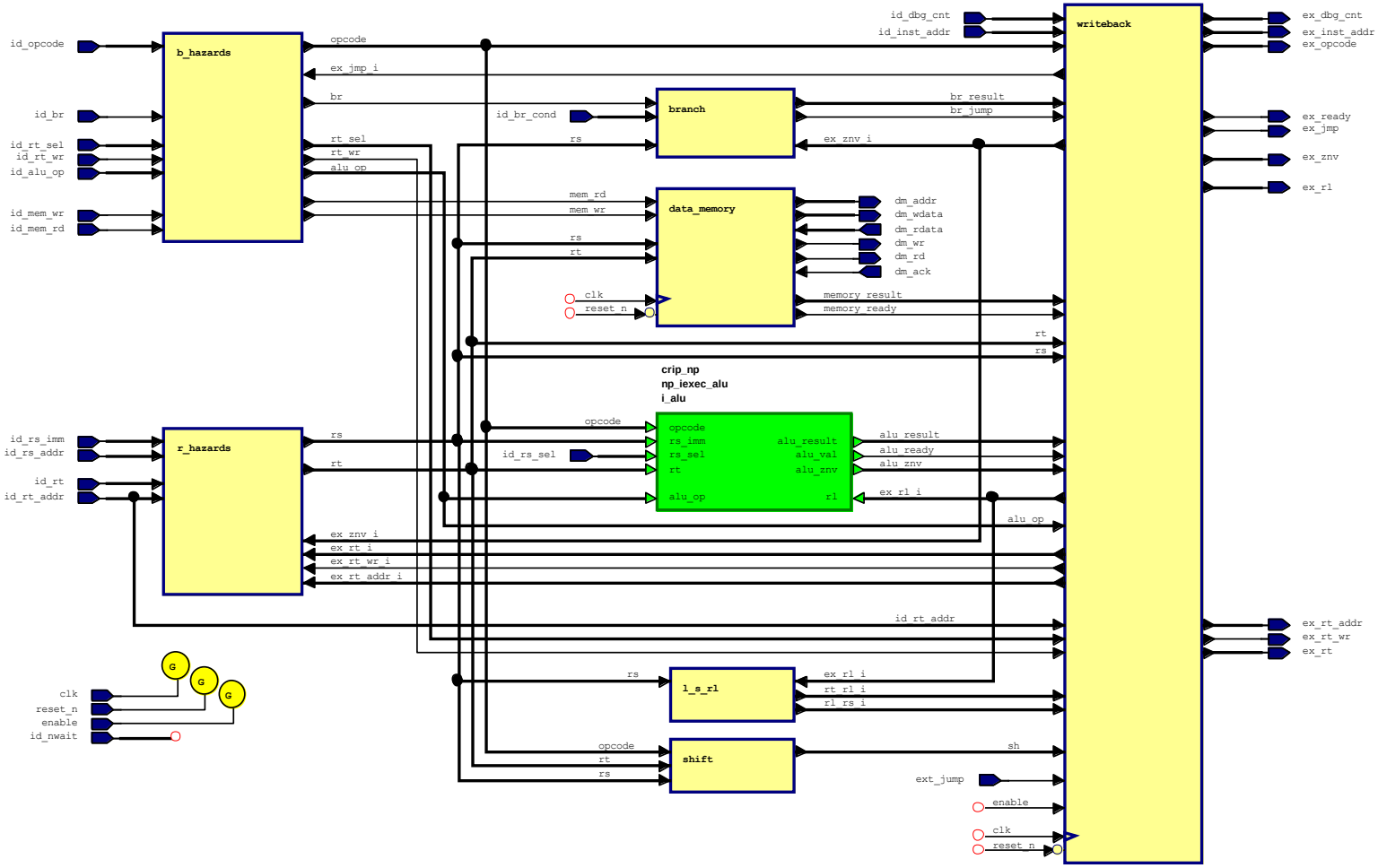


*Figura 38. i_exec - diagrama de bloques*

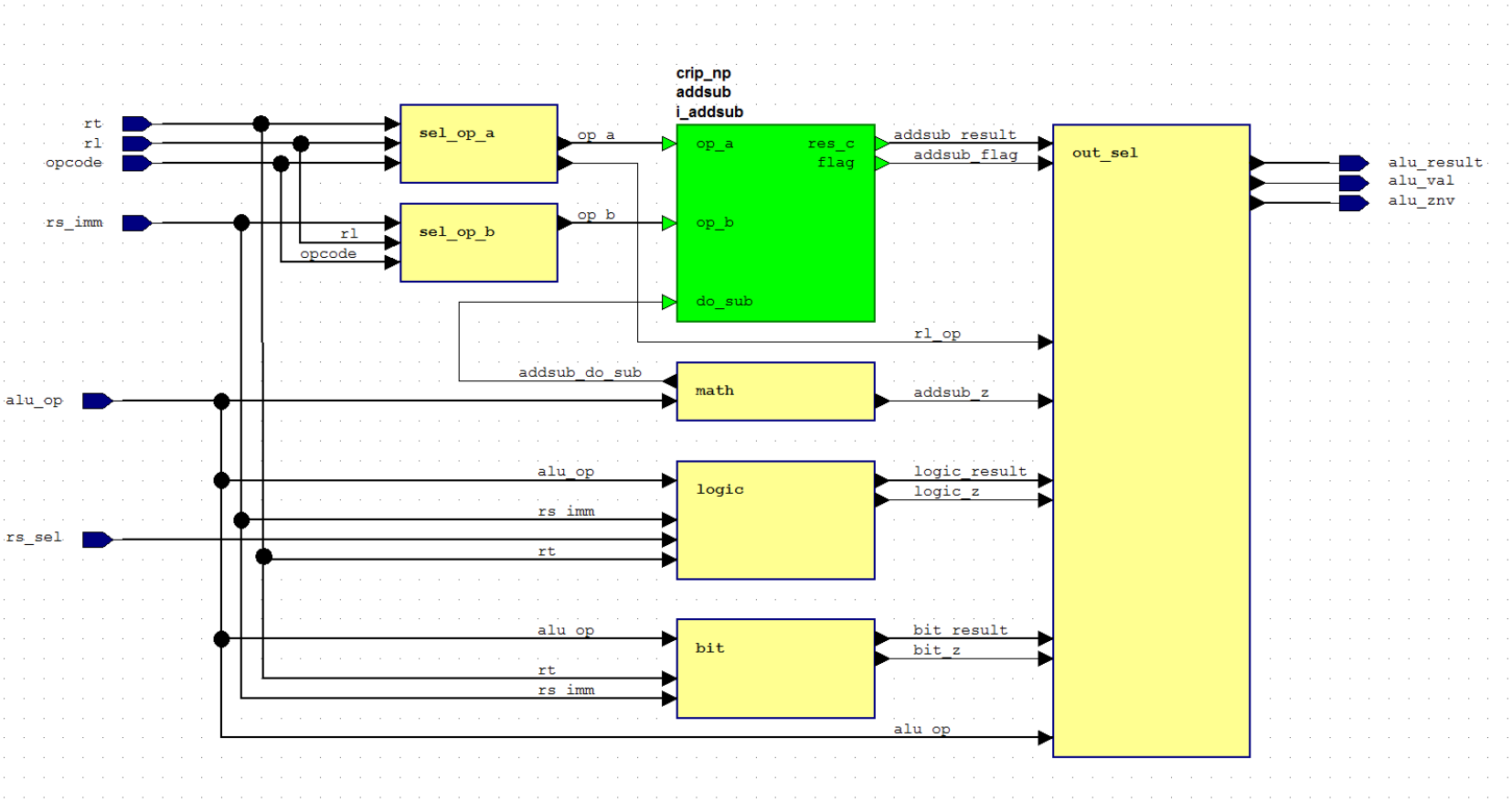


*Figura 39. i_alu - diagrama de bloques*

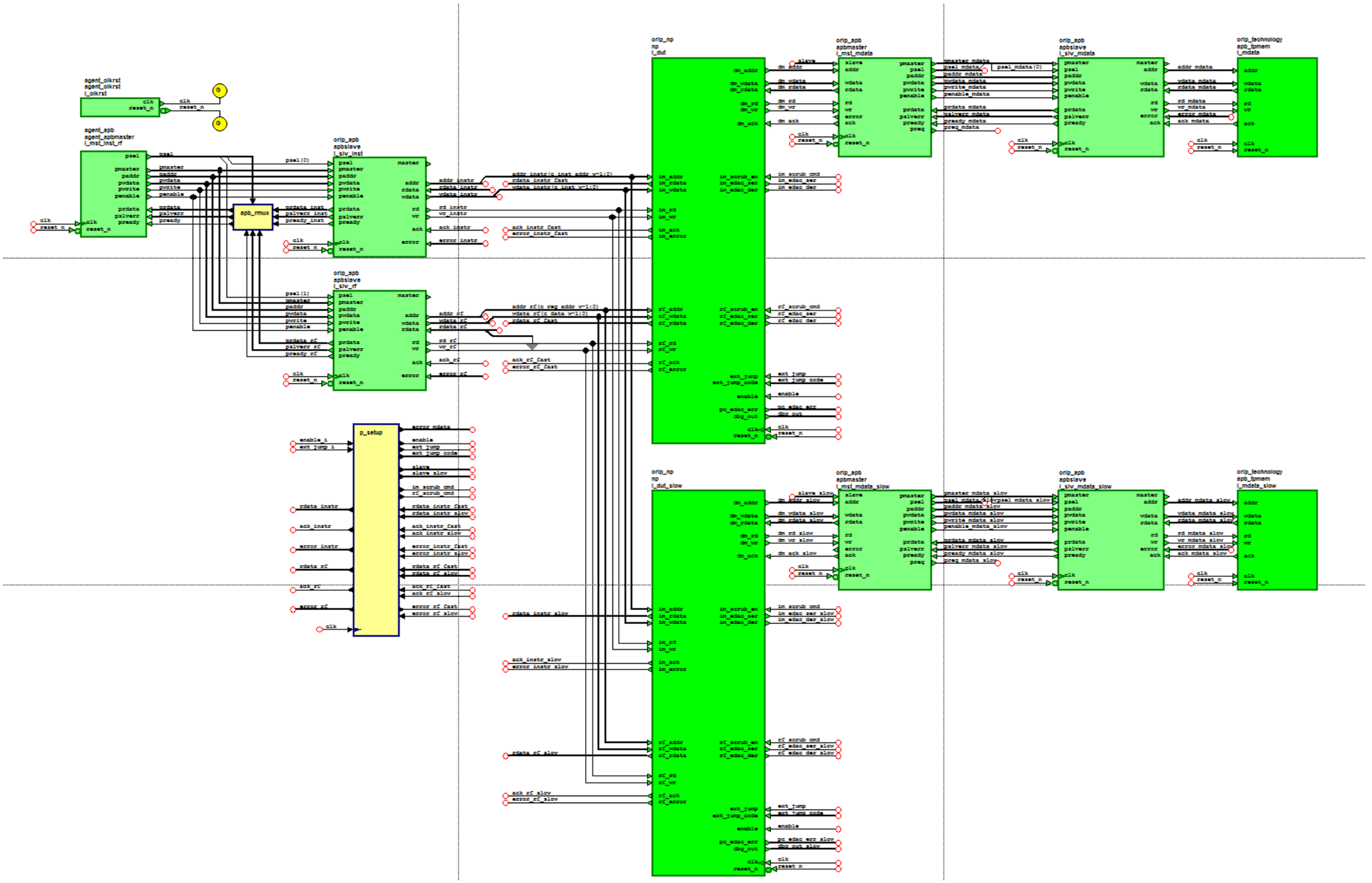


*Figura 40. Testbench inicial - diagrama de bloques*

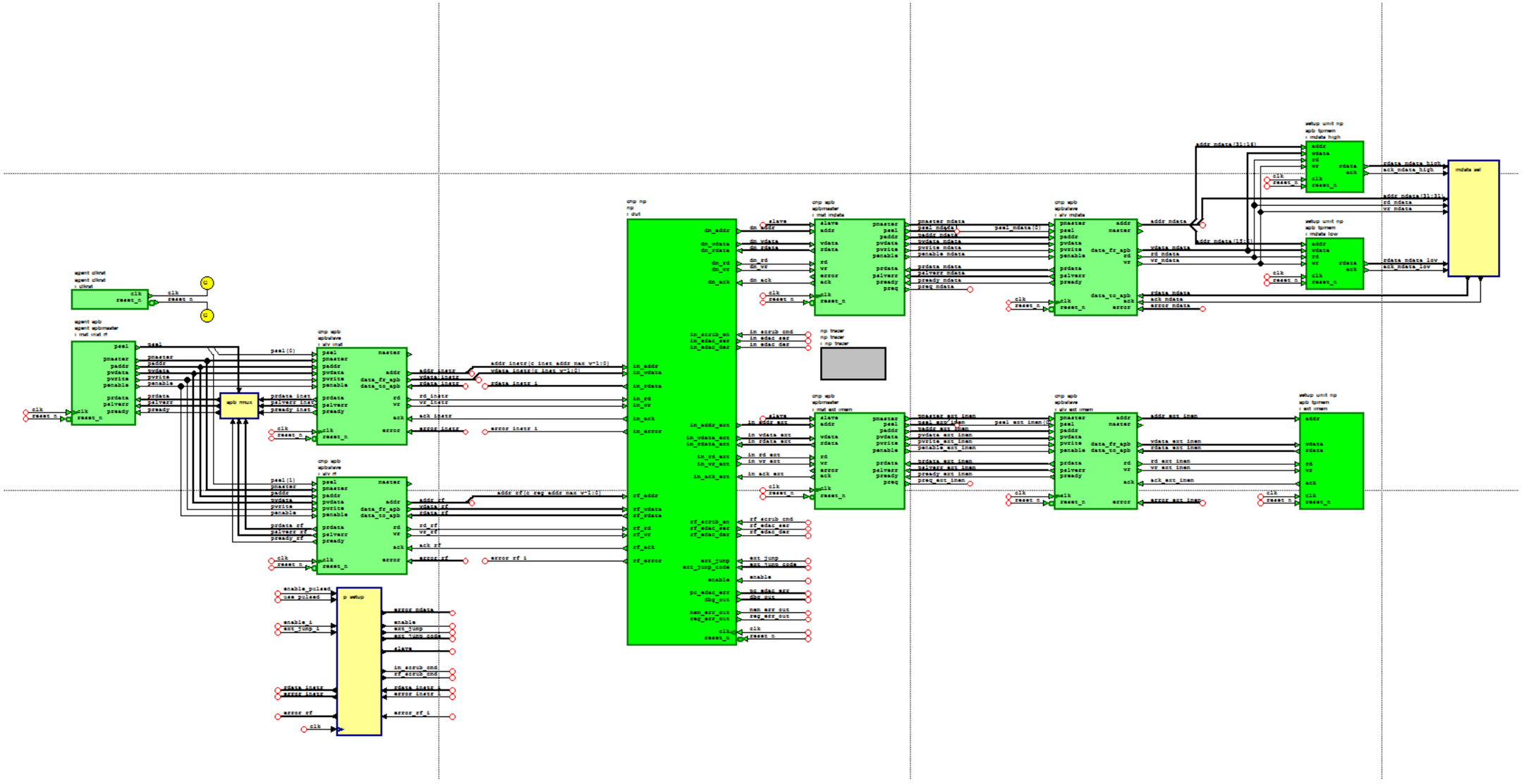

*Figura 41. Testbench final - diagrama de bloques*

# *Parte V COMANDOS*

| Comando | Bits | | | | | | | | | | | | | | | | | | | | | | Implementación | Descripción |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
| **nop** | 0 | 0 | 0 | 0 | 0 | 0 | - | | | | | | | | | | | | | | | | | | nop | PC = PC + 1 |
| **br** | 0 | 0 | 0 | 0 | 0 | 1 | - | | | | | | | | | RS | | | | | | | | | br RS | PC = RS |
| **bi** | 0 | 0 | 0 | 0 | 1 | 0 | - | | | | | | aimm12 | | | | | | | | | | | | bi imm12 | PC = imm12 |
| **be** | 0 | 0 | 0 | 0 | 1 | 1 | - | | | | | | aimm12 | | | | | | | | | | | | be imm12 | if Z then PC = imm12 |
| **bne** | 0 | 0 | 0 | 1 | 0 | 0 | - | | | | | | aimm12 | | | | | | | | | | | | bne imm12 | if !Z then PC = imm12 |
| **bgt** | 0 | 0 | 0 | 1 | 0 | 1 | - | | | | | | aimm12 | | | | | | | | | | | | bgt imm12 | if !(Z or (N xor V)) then PC = imm12 |
| **bge** | 0 | 0 | 0 | 1 | 1 | 0 | - | | | | | | aimm12 | | | | | | | | | | | | bge imm12 | if !(N xor V) then PC = imm12 |
| **-** | 0 | 0 | 0 | 1 | 1 | 1 | - | | | | | | | | | | | | | | | | | | | |
| **lw** | 0 | 0 | 1 | 0 | 0 | 0 | RT | | | | | | | | | RS | | | | | | | | | lw RT, (RS) | RT = mem(RS) |
| **lwi** | 0 | 0 | 1 | 0 | 0 | 1 | RT | | | | | | | | | uimm9 | | | | | | | | | lwi RT, (imm9) | RT = mem(imm9) |
| **sw** | 0 | 0 | 1 | 0 | 1 | 0 | RT | | | | | | | | | RS | | | | | | | | | sw RT, (RS) | mem(RS) = RT |
| **swi** | 0 | 0 | 1 | 0 | 1 | 1 | RT | | | | | | | | | uimm9 | | | | | | | | | swi RT, (imm9) | mem(imm9) = RT |
| **mv** | 0 | 0 | 1 | 1 | 0 | 0 | RT | | | | | | | | | RS | | | | | | | | | mv RT, RS | RT = RS |
| **mvi** | 0 | 0 | 1 | 1 | 0 | 1 | RT | | | | | | | | | simm9 | | | | | | | | | mvi RT, imm9 | RT = signext(imm9) |
| **mva** | 0 | 0 | 1 | 1 | 1 | 0 | RT | | | | | | aimm12 | | | | | | | | | | | | mva RT, imm9 | RT = imm12, 0 ≤ RT ≤ 63 |
| **-** | 0 | 0 | 1 | 1 | 1 | 1 | - | | | | | | | | | | | | | | | | | | | |
| **add** | 0 | 1 | 0 | 0 | 0 | 0 | RT | | | | | | | | | RS | | | | | | | | | add RT, RS | RT = RT + RS, update ZNV |
| **addi** | 0 | 1 | 0 | 0 | 0 | 1 | RT | | | | | | | | | simm9 | | | | | | | | | addi RT, imm9 | RT = RT + signext(imm9), update ZNV |
| **sub** | 0 | 1 | 0 | 0 | 1 | 0 | RT | | | | | | | | | RS | | | | | | | | | sub RT, RS | RT = RT – RS, update ZNV |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **subi** | 0 | 1 | 0 | 0 | 1 | 1 | RT | simm9 | | subi RT, imm9 | RT = RT – signext(imm9), update ZNV |
| **cmp** | 0 | 1 | 0 | 1 | 0 | 0 | RT | RS | | cmp RT, RS | RT – RS, update ZNV |
| **cmpi** | 0 | 1 | 0 | 1 | 0 | 1 | RT | simm9 | | cmpi RT, imm9 | RT – signext(imm9), update ZNV |
| **shli** | 0 | 1 | 0 | 1 | 1 | 0 | RT | - | uimm5 | shli RT, imm5 | RT = (RT << imm5) |
| **shri** | 0 | 1 | 0 | 1 | 1 | 1 | RT | - | uimm5 | shri RT, imm5 | RT = (RT >> imm5) |
| **and** | 0 | 1 | 1 | 0 | 0 | 0 | RT | RS | | and RT, RS | RT = bitwise RT and RS, update Z, NV = 0 |
| **andi** | 0 | 1 | 1 | 0 | 0 | 1 | RT | uimm9 | | andi RT, imm9 | RT = bitwise RT and signext(imm9), upd. Z, NV = 0 |
| **or** | 0 | 1 | 1 | 0 | 1 | 0 | RT | RS | | or RT, RS | RT = bitwise RT or RS, update Z, NV = 0 |
| **ori** | 0 | 1 | 1 | 0 | 1 | 1 | RT | uimm9 | | ori RT, imm9 | RT = bitwise RT or signext(imm9), upd. Z, NV = 0 |
| **xor** | 0 | 1 | 1 | 1 | 0 | 0 | RT | RS | | xor RT, RS | RT = bitwise RT xor RS, update Z, NV = 0 |
| **xori** | 0 | 1 | 1 | 1 | 0 | 1 | RT | uimm9 | | xori RT, imm9 | RT = bitwise RT xor signext(imm9), upd. Z, NV = 0 |
| - | 1 | 0 | 1 | 1 | 1 | X | - | | | | |
| **mvr** | 1 | 0 | 0 | 0 | 0 | 0 | RT | RS | | mvr RT, RS | RT = R(RS) |
| **mvw** | 1 | 0 | 0 | 0 | 0 | 1 | RT | RS | | mvw RT, RS | R(RT) = RS |
| **mvro** | 1 | 0 | 0 | 0 | 1 | 0 | RT | RS | | mvro RT, RS | RT = R(RS + RL) |
| **mvwo** | 1 | 0 | 0 | 0 | 1 | 1 | RT | RS | | mvwo RT, RS | R(RT + RL) = RS |
| - | 1 | 0 | 0 | 1 | X | X | - | | | | |
| **btest** | 1 | 0 | 1 | 0 | 0 | 0 | RT | RS | | btest RT, RS | if RT(RS mod 32) = 1 then Z = 0 else Z = 1 |
| **btesti** | 1 | 0 | 1 | 0 | 0 | 1 | RT | - | uimm5 | btesti RT, imm5 | if RT(imm5) = 1 then Z = 0 else Z = 1 |
| **bset** | 1 | 0 | 1 | 0 | 1 | 0 | RT | RS | | bset RT, RS | RT(RS mod 32) = 1 |
| **bseti** | 1 | 0 | 1 | 0 | 1 | 1 | RT | - | uimm5 | bseti RT, imm5 | RT(imm5) = 1 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **brst** | 1 | 0 | 1 | 1 | 0 | 0 | RT | RS | | brst RT, RS | RT(RS mod 32) = 0 |
| **brsti** | 1 | 0 | 1 | 1 | 0 | 1 | RT | - | uimm5 | brsti RT, imm5 | RT(imm5) = 0 |
| - | 1 | 0 | 1 | 1 | 1 | X | - | | | | |
| - | 1 | 1 | 0 | X | X | X | - | | | | |
| **lrl** | 1 | 1 | 1 | 0 | 0 | 0 | RT | - | | lrl RT | RT = RL |
| **srl** | 1 | 1 | 1 | 0 | 0 | 1 | - | RS | | srl RS | RL = RS |
| **srli** | 1 | 1 | 1 | 0 | 1 | 0 | - | simm9 | | srli imm9 | RL = imm9 |
| **addrl** | 1 | 1 | 1 | 0 | 1 | 1 | - | RS | | addrl RS | RL = RL + RS, update ZNV |
| **addrli** | 1 | 1 | 1 | 1 | 0 | 0 | - | simm9 | | addrli imm9 | RL = RL + signext(imm9), update ZNV |
| **subrl** | 1 | 1 | 1 | 1 | 0 | 1 | - | RS | | subrl RS | RL = RL – RS, update ZNV |
| **subrli** | 1 | 1 | 1 | 1 | 1 | 0 | - | simm9 | | subrli imm9 | RL = RL – signext(imm9), update ZNV |
| - | 1 | 1 | 1 | 1 | 1 | 1 | - | | | | |

*Tabla 20. Comandos*

- **Bits (23:18)**: código de operación
- **Bits (17:9)**: generalmente, registro RT
- **Bits (8:0)**: generalmente, registro RS o imm9

- **aimm12**: inmediato de 12 bits
- **uimm9**: inmediato sin signo de 9 bits
- **simm9**: inmediato con signo de 9 bits; se necesita utilizar los inmediatos con signo en las operaciones aritméticas
- **uimm4**: inmediato sin signo de 4 bits

- **RT**: registro RT
- **RS**: registro RS

- **bitwise**: operación bit a bit (para las operaciones lógicas)
- **mem(*)**: dato contenido en la posición de memoria de la dirección indicada.
- **signext(*)**: necesaria la extensión del signo del parámetro entre los paréntesis para corregir la longitud del mismo para poder operar matemáticamente (misma longitud que el otro operando).
- **R(*)**: registro cuya dirección es el contenido del registro indicado.

- **Z**: flag de cero (SET cuando resultado = 0)
- **N**: flag de número negativo (SET cuando resultado < 0)
- **V**: flag de *overflow* (SET cuando resultado < min o resultado > max)