Universidad Zaragoza
1542

Trabajo Fin de Grado

Implementación de un procesador RISC-V con soporte para un sistema operativo de tiempo real

Implementation of a RISC-V processor with support for a real time operating system

Autor

Samuel Pérez Pedrajas

Directores

Javier Resano Ezcaray

Darío Suárez Gracia

Titulación

Grado en Ingeniería Informática

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2022

AGRADECIMIENTOS

Quiero agradecer a Darío y Javier por darme la oportunidad de hacer este trabajo y ayudarme siempre que se lo he pedido.

RESUMEN

El objetivo final de este trabajo es conseguir diseñar un procesador RISC-V capaz de ejecutar un sistema operativo.

El proyecto RISC-V ofrece una ISA (*Instruction Set Architecture*) abierta lo que la sitúa como una buena alternativa frente a otras arquitecturas RISC, como puede ser ARM, a la hora de diseñar nuevos procesadores al no tener que pagar ningún tipo de comisión por utilizarla. Para tener éxito, una arquitectura debe dar un buen soporte a los componentes *software* mas importantes en la actualidad como son los sistemas operativos.

A lo largo de este trabajo se ha diseñado un procesador RISC-V capaz de ejecutar el repertorio de instrucciones básico definido en su ISA. A continuación se han estudiado los requisitos necesarios para poder ejecutar el sistema operativo de tiempo real FreeRTOS (*Free Real Time Operating System*) y se han incluido en el diseño diversas extensiones para cumplirlos. Estas extensiones incluyen tanto nuevas instrucciones y registros de la ISA, como mecanismos de comunicación y de gestión de tiempo.

El funcionamiento del procesador diseñado se ha validado mediante simulación y a continuación se ha implementado sobre *hardware* real utilizando una FPGA (*Field Programable Gate Array*). Las FPGA son una muy buena herramienta para poder validar el funcionamiento de un diseño sobre *hardware* real a un bajo coste, ya que permiten implementar el diseño objetivo en un dispositivo *hardware* configurable desde cualquier ordenador. Este trabajo también documenta la metodología que se ha utilizado para realizar estas pruebas y como se ha desarrollado un entorno para realizarlas.

Índice

Capítulo 1

Introducción

1.1. Motivación

RISC-V[1] es una ISA abierta, lo que implica que no es necesario pagar ningún tipo de comisión por utilizarla en el diseño de un nuevo procesador. Con la escasez actual de microcontroladores y la gran necesidad de estos en mercados como el de la automovilística o en IoT (*Internet of Things*) están surgiendo muchos proyectos para crear nuevas fábricas y desarrollar nuevos diseños. RISC-V se presenta como una posible alternativa para utilizarse en estos nuevos proyectos.

Estudios de mercado valoran el futuro de esta arquitectura de forma muy positiva prediciendo un gran crecimiento en su tasa de mercado, considerando que en 2023 estará cerca de los 800 millones de dólares americanos y en 2024 cerca de los 1000 [1]. Otro buen indicador de un futuro positivo para la arquitectura es el proyecto conjunto de Intel y el BSC (*Barcelona Supercomputing Center*) de crear un laboratorio para diseñar procesadores RISC-V centrados en la computación de altas prestaciones [2]. Como indicador del impacto actual de esta arquitectura la empresa SiFive, siendo una de las principales impulsoras de RISC-V, está valorada en mas de 2500 millones de dolares americanos [3].

Dentro de los elementos software de un sistema, los sistemas operativos son una de las herramientas principales y toda ISA que quiera ser empleada masivamente requiere dar un buen soporte para ellos. Este soporte debido a su complejidad suele ser poco estudiado en los planes de estudio, pero debido al auge de nuevas arquitecturas, como RISC-V, es crítico su estudio. Este trabajo muestra la implementación de un sub-conjunto de la arquitectura RISC-V capaz de ejecutar un sistema operativo. La implementación es sintetizable pudiéndose ejecutar en una FPGA.

[1] `https://riscv.org`

1.2. Objetivos y Alcance

El objetivo final de este trabajo es conseguir implementar en una FPGA un procesador RISC-V capaz de ejecutar un sistema operativo de tiempo real como FreeRTOS[2]. Se van a seguir los siguientes pasos para lograr este objetivo:

- Estudiar la ISA RISC-V.
- Partiendo del diseño de un procesador MIPS estudiado en la asignatura AOC2, diseñar la lógica de un procesador RISC-V básico.
- Implementar el diseño en el lenguaje de especificación hardware VHDL (*Very high speed integrated circuit Hardware Description Language*).
- Validar el diseño mediante simulación.
- Estudiar los requisitos de FreeRTOS para ejecutarse en una plataforma RISC-V.
- Ejecutar FreeRTOS sobre una plataforma RISC-V emulada utilizando QEMU[3].
- Extender el diseño para que cumpla con los requisitos de FreeRTOS.
- Validar de nuevo del diseño mediante simulación.
- Estudiar las herramientas para implementar diseños hardware sobre FPGAs.
- Implementar y validar el diseño sobre una FPGA.

1.3. Estructura del documento

El Capítulo 2 describe el estado del arte de la ISA RISC-V, los sistemas operativos de tiempo real, sus requisitos, los lenguajes de especificación *hardware* y su uso en FPGAs. El Capítulo 3 describe la metodología que se ha seguido durante el desarrollo del proyecto y las herramientas que se han utilizado en el. El Capítulo 4 muestra las decisiones de diseño que se han tomado para el procesador RISC-V diseñado. El Capítulo 5 describe las pruebas que se han realizado para validar el diseño y como se han desarrollado. El Capítulo 6 muestra el proceso y los resultados de la implementación del diseño en una FPGA. Por último el Capitulo 7 contiene las conclusiones del trabajo.

[2] `https://www.freertos.org`
[3] `https://www.qemu.org`

Capítulo 2

Estado del Arte

2.1. ISA RISC-V

El proyecto RISC-V comenzó como un proyecto de investigación en 2010 en la Universidad de Berkeley, California [4]. Cuenta con el apoyo de muchas empresas del sector, como Intel o Qualcomm, y desde entonces han publicado varias versiones y revisiones de la especificación de la ISA.

La especificación de la ISA está divida en 2 documentos. El documento de la ISA no privilegiada [5], el cual se centra en definir los aspectos generales de los procesadores RISC-V, conjuntos de instrucciones base y extensiones de propósito general. Y el de la ISA privilegiada [6], el cual esta centrado en definir el soporte a diferentes modos de ejecución y otras necesidades comunes de los sistemas operativos, explicadas en mas detalle en la Sección 2.2. Algunas de las extensiones definidas en estos documentos aun son borradores, pero los conjuntos de instrucciones base de la ISA no privilegiada y el soporte diferentes modos de ejecución de la privilegiada ya se encuentran ratificados.

Una ISA define las instrucciones de lenguaje maquina que un procesador puede ejecutar. Ofrece las herramientas a los programadores y programas, como puede ser un compilador, para poder crear código para dicho procesador. Una ISA no especifica como se han de implementar las instrucciones pero si como se han de comportar, por lo que pueden existir muchas implementaciones diferentes de una misma ISA.

La ISA RISC-V, como su nombre indica, esta basada en un diseño de tipo RISC (*Reduced Instruction Set Computer*). Este diseño implica que la ISA define instrucciones simples, que tardan poco tiempo en ejecutarse. No permite realizar operaciones aritméticas con operandos en memoria principal, obliga siempre a mover los operandos entre registros y memoria mediante instrucciones específicas. Otro ejemplo de una arquitectura RISC es ARM. Otro tipo de diseño es CISC (*Complex Instruction Set Computer*). Este suele definir un conjunto muy grande de instrucciones con operaciones complejas y específicas. Una de las diferencias principales con RISC que generalmente

permiten realizar operaciones con operandos directamente en memoria principal. Un ejemplo de arquitectura CISC es x86_64.

Una ISA debe estar pensada para poder utilizarse en diseños de sistemas de bajo coste, de propósito general o de altas prestaciones. Definiendo para ello un conjunto de instrucciones base sencillo y varias posibles extensiones estándar. En la actualidad los procesadores pueden ser de 32 bits o de 64 bits, este numero indica el tamaño de los operandos de las instrucciones. RISC-V ofrece conjuntos base de instrucciones para ambos tamaños, permitiendo así crear diseños que se acomoden a las necesidades de diferentes consumidores. También esta diseñada con el propósito de que sea fácil crear extensiones nuevas. Ayudando de esta forma a diseñar nuevas instrucciones para acelerar la ejecución de ciertos esquemas de código, o interactuar con nuevo *hardware*, en caso de que la implementación lo necesite.

2.1.1. Instrucciones RISC-V

Un procesador debe de ser capaz de ejecutar instrucciones. Estas instrucciones realizan diferentes operaciones con datos. Las instrucciones y datos están almacenados en una memoria. En los procesadores RISC-V se define un único espacio de direccionamiento que comparten instrucciones y datos. La ISA define que las direcciones de memoria están definidas a nivel de byte, es decir que cuando se realice un acceso a memoria como mínimo se ha de acceder a un byte.

El tamaño base de una instrucción es de 4 bytes, aunque, también se definen tamaños de instrucción reducidos y extendidos pero no se van a utilizar. Como las direcciones son a nivel de byte se tiene que definir en que orden se colocan los 4 bytes de las instrucciones en memoria. La ISA define que para las instrucciones se tiene utilizar *Little Endian*, es decir, que los bytes se colocan en orden empezando por el byte de menor peso. En la figura 2.1 se puede ver un ejemplo.

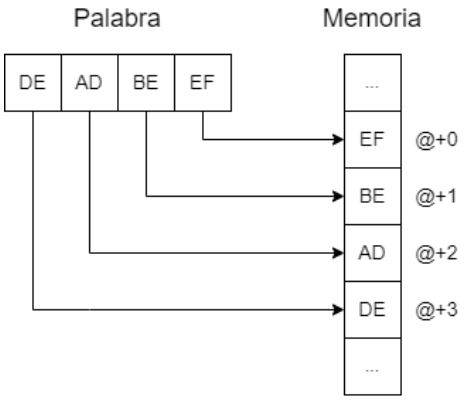

Figura 2.1: Ejemplo *Little Endian*.

Para todas las instrucciones, salvo las de acceso a memoria, los operandos fuente y destino deben ser siempre registros. La ISA define un banco de 32 registros con este propósito. Uno de estos registros, llamado `x0`, siempre toma el valor 0. Para almacenar la dirección de memoria de la instrucción que se esta ejecutando se utiliza un registro extra llamado `PC`.

Los tipos que instrucciones que forman los conjuntos base y como estas se codifican se encuentra detallado en el Anexo B.

2.2. Requisitos básicos de un sistema operativo

Un sistema operativo es un programa, generalmente muy complejo, que se encarga de gestionar y abstraer los recursos de la plataforma y ofrecer diferentes servicios a los programas que se ejecutan en el sistema. En la figura 2.2 se puede ver un diagrama sencillo de donde se sitúa un sistema operativo con respecto al resto de elementos.

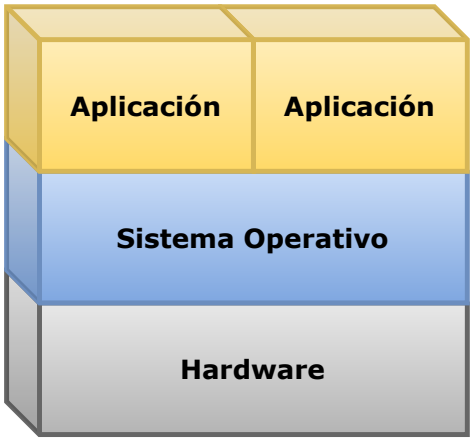

Figura 2.2: Esquema de una plataforma con un sistema operativo.

Los sistemas operativos deben proveer seguridad, es decir, no permitir que una aplicación pueda tomar el control del *hardware*, del sistema o de otras aplicaciones. Para que puedan hacerlo las ISA generalmente definen, como mínimo, un modo de ejecución privilegiado y uno no privilegiado. Los modos no privilegiados no tienen acceso a todas las instrucciones ni a todos los recursos *hardware* de la plataforma, mientras que los modos privilegiados tienen control completo. También se definen mecanismos de protección de memoria para controlar los accesos a diferentes secciones de esta por parte de los modos no privilegiados.

Los sistemas operativos se ejecutan en modo privilegiado y las aplicaciones en modo no privilegiado, cuando se intenta ejecutar una acción privilegiada en modo no privilegiado el modo privilegiado toma el control, en este caso el sistema operativo, pudiendo así implementar mecanismos de seguridad.

Las aplicaciones tienen que ser capaces de solicitar servicios al sistema operativo, estos servicios se denominan llamadas al sistema. Las ISA tienen que ofrecer la capacidad de invocar al modo privilegiado desde el modo no privilegiado para que las aplicaciones puedan hacer esto.

Un sistema operativo debe de de poder ejecutar varias aplicaciones a la vez, por lo que ha de ser capaz de repartir el tiempo de ejecución entre todas ellas. La parte del sistema operativo que se encarga de esto se llama el planificador, este planificador debe de poder ser capaz de medir el tiempo para poder repartirlo.

2.2.1. Sistemas operativos de tiempo real

Los sistemas operativos de tiempo real son sistemas ligeros y con tiempos de respuesta predecibles. El objetivo principal de estos sistemas es que las tareas que ejecutan cumplan plazos temporales. Para ello permiten configurar las tareas con diferentes prioridades.

FreeRTOS es un sistema operativo de tiempo real de código libre. Está escrito en C y ensamblador, tiene diferentes versiones para muchas plataformas, incluidas algunas plataformas RISC-V.

2.3. Lenguajes de especificación *hardware* y FPGAs

Estos lenguajes permiten definir el comportamiento de circuitos lógicos. Se pueden utilizar para definir el comportamiento de los diferentes bloques lógicos que forman un procesador. Algunos ejemplos de lenguajes de este tipo son VHDL y Verilog. Una vez se tiene un diseño definido en estos lenguajes con la ayuda de un simulador se puede validar que su comportamiento es el esperado.

Los circuitos especificados en uno de estos lenguajes luego se pueden implementar en una FPGA utilizando una herramienta de síntesis. Una FPGA es un componente *hardware* en el que se pueden configurar circuitos lógicos. Uno de los usos principales de estos componentes es poder configurar aceleradores para diferentes tareas en ellos y luego que un procesador de propósito general haga uso de ellos para obtener mejores prestaciones. Otro posible uso de las FPGA es que permiten simular en *hardware* real un circuito con un bajo coste.

En la actualidad empresas como Xilinx comercializan plataformas de desarrollo en la que conviven un procesador de propósito general y una FPGA. También ofrecen las herramientas de síntesis necesarias para poder programar las FPGA de estas plataformas.

Capítulo 3

Metodología y herramientas

3.1. Metodología

Se ha seguido un proceso iterativo en el diseño del procesador. Cada vez que se añadía soporte para una instrucción o funcionalidad nueva se validaba que todas funcionalidades previas siguieran funcionando y que las nuevas también mediante simulación. En paralelo al desarrollo del diseño procesador se adaptaban programas de prueba ya existentes o se creaban nuevos en caso de que no hubiera, siempre intentando que el proceso de validación fuera lo mas automático posible. También en paralelo a estas tareas se estudió el comportamiento de FreeRTOS sobre una plataforma RISC-V emulada utilizando QEMU.

Una vez el diseño del procesador cumplía con los requisitos para poder ejecutar FreeRTOS se adaptó una de las versiones de prueba que ofrece para plataformas RISC-V, la misma que se utilizó para estudiar su comportamiento sobre QEMU, y se validó el correcto funcionamiento de esta nueva versión mediante simulación.

Después se implementó el procesador diseñado sobre una placa de desarrollo con una FPGA y una CPU. Para validar que el procesador implementado en la FPGA funcionaba correctamente se desarrolló un entorno de pruebas que permitía programar la memoria de este procesador sin tener que repetir los procesos de síntesis, que son muy costosos en tiempo, cada vez que se quería probar un programa nuevo. Este entorno de pruebas también permitía al procesador implementado en al FPGA comunicarse con la CPU de la plataforma para enviarle información y así poder observar que comportamiento estaba teniendo.

En el Anexo A se puede observar un diagrama de Gantt del proyecto y un diagrama el porcentaje de horas que se han dedicado a cada grupo de tareas.

3.2. Herramientas utilizadas

Para especificar el procesador se ha utilizado el lenguaje VHDL y para validar su comportamiento mediante simulación GHDL[1]. Una de las ventajas de este compilador y simulador es que es de código abierto al igual que GTKWave[2] que se ha utilizado para visualizar el valor de las señales de simulación. Además, se ha automatizado la infraestructura de diseño y pruebas con *scripts* de Bash y Python.

Como no se disponía de una plataforma RISC-V real se ha empleado QEMU para virtualizar y emular una sobre la que ejecutar un sistema operativo.

Para compilar el sistema operativo se ha utilizado la RISC-V GNU *Compiler Toolchain*[3], un conjunto de herramientas para compilar código C y ensamblador RISC-V. El proceso de compilación se ha automatizado mediante Make.

Una vez completado y validado el diseño este se ha sintetizado sobre la plataforma Zedboard[4], la cual cuenta con un procesador ARM y una FPGA. Como herramienta de síntesis se ha utilizado Vivado *Design Suite*[5] y para programar la plataforma Vitis IDE (*Integrated Development Environment*)[6], ambas ofrecidas por Xilinx.

A la hora de redactar este documento se ha utilizado el editor de LaTeX en línea Overleaf[7] y para hacer los diagramas que contiene se ha utilizado el editor de diagramas en línea Diagrams.net[8].

[1] `http://ghdl.free.fr`
[2] `http://gtkwave.sourceforge.net`
[3] `https://github.com/riscv-collab/riscv-gnu-toolchain`
[4] `https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard`
[5] `https://www.xilinx.com/products/design-tools/vivado.html`
[6] `https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html`
[7] `https://www.overleaf.com`
[8] `https://app.diagrams.net`

Capítulo 4

Diseño del procesador

Se ha decidido implementar un procesador de 32 bits, por lo que se va a utilizar el conjunto de instrucciones base RV32I que contiene 40 instrucciones. Como los operandos de las instrucciones son de 32 bits las direcciones para acceder a memoria también lo son, por lo que el espacio de direccionamiento es como máximo de 4 GB.

La especificación da libertad sobre como se han de almacenar los datos en memoria, pero se ha decidido por simplicidad utilizar el mismo formato de las instrucciones, *Little Endian*.

4.1. Ruta de datos

Se emplea una ruta de datos dividida en múltiples etapas. Así aumenta la frecuencia del reloj al reducir el tamaño del camino combinacional más lento. Esto también ha permitido aplicar técnicas de optimización como la segmentación, explicada en el Anexo C.

La contrapartida es un aumento del coste y la complejidad del diseño [7]. Incluso incluir muchas etapas puede degradar el rendimiento. Dada la complejidad de esta ruta de datos se ha decidido utilizar 5 etapas, este número de etapas ya se ha utilizado anteriormente en rutas de datos similares como pueden ser las de los primeros procesadores MIPS [8].

Las etapas son las siguientes:

- ***Fetch***: Lectura de los 4 bytes de la instrucción de la memoria.
- ***Decode***: Extracción de todos los valores codificados en la instrucción. Entre ellos el tipo de la instrucción, el valor inmediato, los registros fuente y el registro destino. Se calculan todas las señales de control para las etapas posteriores y se accede al banco de registros para leer los valores de los operandos fuente. Debido a la segmentación en esta etapa se deben detectar posibles dependencias entre

instrucciones y parar la ejecución si no se pueden adelantar los operandos en la siguiente etapa.

- ***Execution***: Calculo de los resultados de las operaciones aritméticas, las direcciones de salto para las instrucciones de salto, las direcciones para acceder a memoria para las instrucciones *load/store*. Comparaciones para comprobar si se han de realizar los saltos condicionales. Si en la etapa anterior se ha detectado que se ha de adelantar algún operando se adelantan en esta etapa.

- ***Memory***: Las instrucciones *load/store* realizan un acceso a memoria, escribiendo en ella en el caso de los *store* o leyendo en el caso de los *loads*.

- ***Writeback***: Escritura del resultado de la instrucción en el registro destino en el caso de que la instrucción tenga uno.

Se ha decidido que todas las instrucciones pasen por todas las etapas para simplificar la implementación de la segmentación, por lo que todas las instrucciones tardan 5 ciclos en ejecutarse.

La Figura 4.1 muestra un diagrama simplificado del diseño en el que se pueden ver los bloques lógicos asociados a cada etapa.

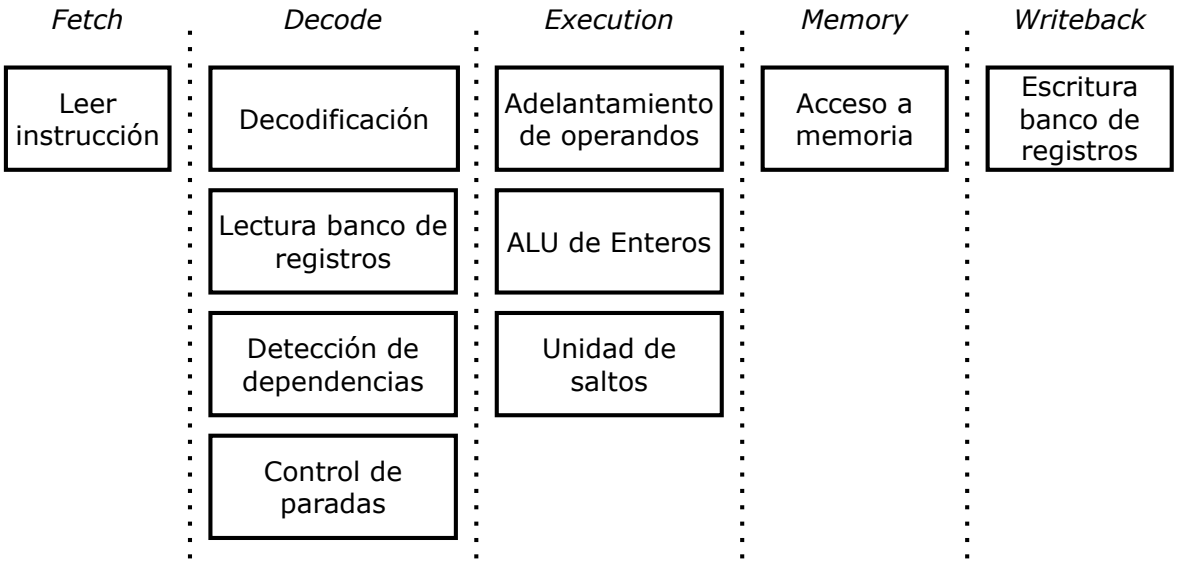

Figura 4.1: Diagrama de bloques lógicos asociados a cada etapa.

4.2. Bancos de memoria

El diseño de la memoria es muy importante ya que dependiendo de como se haga puede que esta sea simulable pero las herramientas de síntesis no sean capaces de

sintetizarla como una memoria RAM. Como el objetivo final de este proyecto es poder implementarlo en una FPGA es necesario diseñar la memoria teniendo esto en cuenta. Además se quiere que se implemente en unos bloques específicos de las FPGA llamados BRAM lo que también se ha de tener en cuenta a la hora de hacer el diseño. El Anexo E explica con mas detalle los tipos de bloques en las FPGA.

Como la ruta de datos esta segmentada la memoria debe de poder soportar 2 accesos simultáneos a 2 direcciones diferentes, ya que tanto la etapa de *Fetch*, que tiene que leer una instrucción, como la de *Memory*, que puede estar ejecutando una instrucción *load/store*, pueden estar realizando un acceso. La latencia de acceso debe de ser de un ciclo, es decir, que al ciclo siguiente de acceder el dato debe estar disponible.

Teniendo todo esto en cuenta se ha decidido implementar la memoria como una memoria RAM dividida en 4 bancos. Cada banco tiene un puerto de lectura y un puerto de lectura/escritura, de esta forma se permite acceder para leer instrucciones y leer o escribir datos.

La ISA define que a la memoria se puede acceder a 4 bytes, lo que implica acceder a los 4 bancos a la vez, a 2 bytes, lo que implica acceder a 2 bancos o a 1 byte que implica acceder a un solo banco. Para leer las instrucciones se accede a los 4 bancos a la vez.

Todos los datos se tienen que guardar en registros de 4 bytes, los valores enteros con signo están codificados en complemento a 2, cuando se lee un dato menor de 4 bytes con signo se tiene que realizar una operación llamada extensión de signo para aumentar su tamaño a 4 bytes. Para los valores sin signo se rellenan con 0.

Diseñar la memoria de esta forma añade una restricción a los accesos, estos solo se pueden hacer alineados al tamaño del dato que se quiera acceder. Para evitar esta limitación hay que complicar mas el diseño pero se ha decidido no hacerlo ya que la ISA permite que los accesos a memoria tengan esta limitación, que es habitual en otras ISAs, por ejemplo en las de ARM.

La figura 4.2 muestra un diagrama del diseño.

4.3. Diferentes modos de ejecución

Los sistemas operativos requieren de varios modos de ejecución con diferentes privilegios para proteger a los usuarios y a las aplicaciones entre sí. En concreto, la ISA RISC-V define hasta 3 modos de ejecución diferentes: maquina, supervisor y usuario. La versión de FreeRTOS para RISC-V solo requiere 2 de ellos, el modo usuario y el modo máquina. La figura 4.3 muestra como el sistema operativo y firmware se ejecutan en modo máquina, mientras que las tareas se ejecutan en modo usuario.

Una vez el diseño soporta el conjunto de instrucciones base se puede empezar a

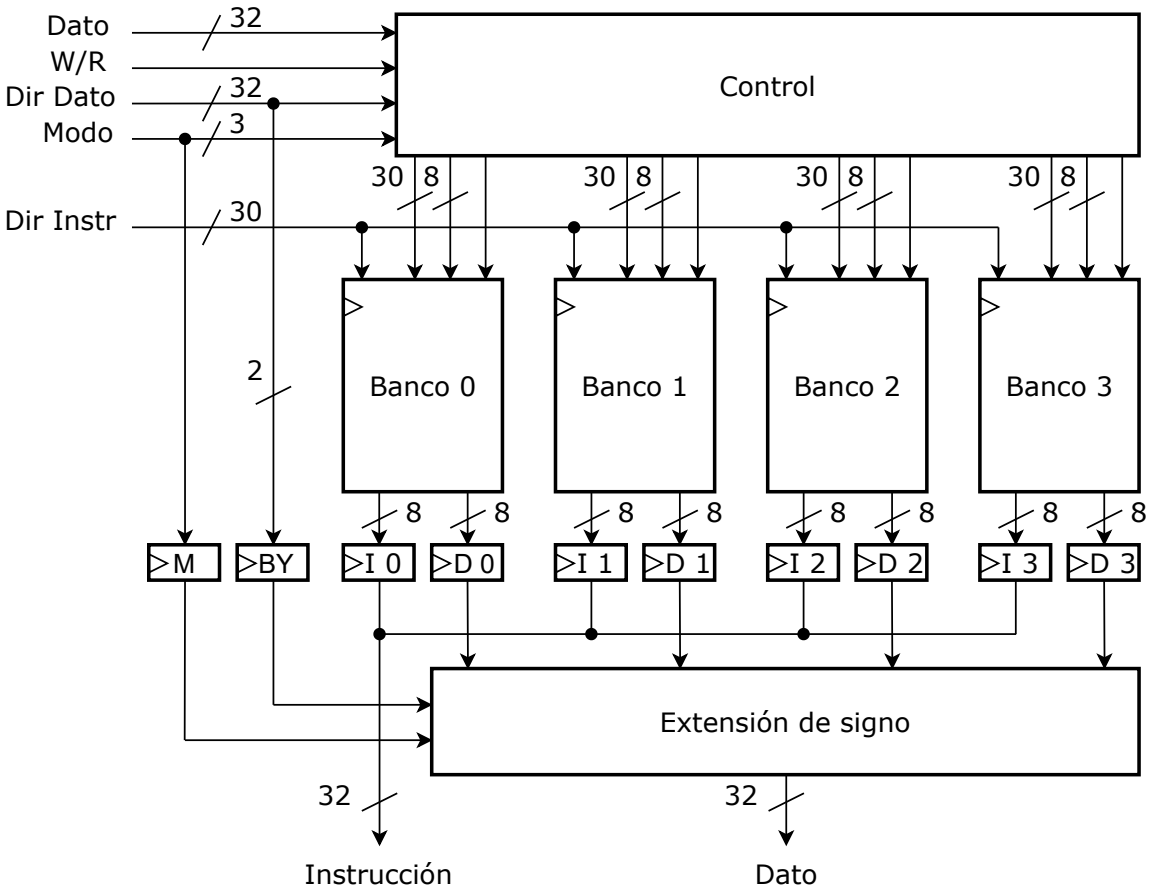

Figura 4.2: Diagrama del diseño de la memoria.

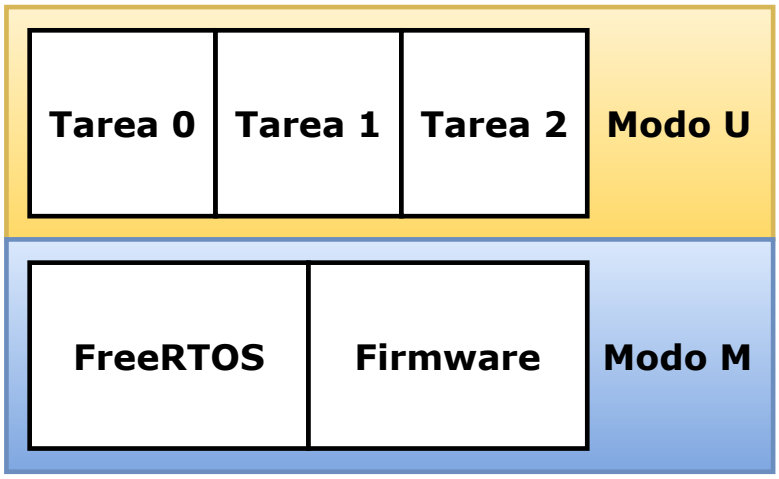

Figura 4.3: Diagrama de los diferentes modos de ejecución.

añadir soporte a diferentes extensiones. Para poder implementar el soporte los modos estándar M (*Machine*) y U (*User*) es necesario añadir nuevos registros, dar soporte a excepciones, interrupciones y a la extensión estándar Zicsr.

4.4. Excepciones e Interrupciones

Una excepción es un evento inesperado que ocurre durante la ejecución de una instrucción y una interrupción es un evento externo al procesador que puede ocurrir en cualquier momento. Añadir soporte para estos eventos es dar las capacidades al procesador de detectarlos y actuar en respuesta a estos, por ejemplo, ejecutar una función cuando ocurra una interrupción.

La ISA define el comportamiento de procesador RISC-V cuando ocurre uno de estos eventos. Se ha de transferir el control a una función, generalmente llamada *trap handler*, y se ha de guardar el estado del procesador en el momento que ha ocurrido el evento para dar la posibilidad de restaurarlo mas adelante. A esta transferencia de control se le llama *trap*, síncrona en el caso de las excepciones y asíncrona en el caso de las interrupciones. Todas las *traps* deben de ser precisas, es decir, que siempre se ha de conocer durante qué instrucción ha ocurrido y poder restaurarse mas adelante.

Las *trap* son el mecanismo principal mediante el cual se transfiere el control desde una aplicación de usuario al sistema operativo, en RISC-V son el único mecanismo para cambiar a modo privilegiado desde un modo no privilegiado. La única manera de cambiar a un modo no privilegiado desde uno privilegiado es mediante la instrucción `mret`.

4.4.1. Registros de control

Los registros que contienen valores que afectan al comportamiento del procesador se llaman CSR (*Control and Status Registers*). La ISA define muchos registros de control para diferentes funcionalidades opcionales que se pueden implementar. Para dar soporte a las excepciones e interrupciones se necesitan registros que almacenen el estado interrumpido (`mstatus` y `mepc`), un registro que almacene la dirección de la función *trap handler* (`mtvec`), un registro que almacene la causa de la interrupción (`mcause`) y un registro que permita bloquear las interrupciones (`mie`). El Anexo D detalla en mas profundidad estos registros y los campos que los forman.

4.4.2. *Traps* soportadas

Se ha decidido que el diseño soporte las siguientes *traps*:

- **Acceso a memoria no alineado**: Por como se ha diseñado la memoria solo se pueden soportar accesos alineados tanto para buscar instrucciones como accesos tanto en lectura como en escritura de datos. Ocurre una *trap* cuando se detecta un acceso no alineado.

– **Instrucción ilegal**: Ocurre una *trap* cuando no se reconoce la instrucción o se intenta ejecutar una instrucción privilegiada en un modo no privilegiado.

– **Instrucción *trap***: Para que una aplicación pueda invocar al sistema operativo de forma voluntaria para solicitar algún servicio existen instrucciones que siempre generan una *trap* al ejecutarse. Estas instrucciones se llaman `ecall` y `ebreak`.

– **Interrupción externa**: Cuando se detecte una interrupción externa y las interrupciones estén habilitadas ocurre una *trap*.

La ISA define los valores que debe tomar el registro `mcause` dependiendo de la causa de la *trap*. La tabla 4.1 muestra los valores que toma este registro con las *traps* que soporta el diseño.

Tabla 4.1: Valor del registro `mcause` en función de la causa de la *trap*.

Causa	Valor `mcause`
Interrupción timer	`0x80000007`
Interrupción externa	`0x8000000b`
Instrucción no alineada	`0x00000000`
Instrucción ilegal	`0x00000002`
`ebreak`	`0x00000003`
Load no alineado	`0x00000004`
Store no alineado	`0x00000006`
`ecall` desde modo U	`0x00000008`
`ecall` desde modo M	`0x0000000b`

4.4.3. Instrucción `mret`

Para restaurar el estado del procesador, la ISA define la instrucción `mret`. Esta instrucción cambia el valor del `PC` por el valor almacenado en `mepc` y restaura el modo de ejecución junto con el estado que se guardó en `mstatus`. Un posible objetivo de restaurar el estado es que un sistema operativo después de haber realizado un servicio pueda devolver el control a la aplicación que se lo ha solicitado. Esta es una instrucción privilegiada ya que puede modificar el modo de ejecución del procesador dependiendo de los valores almacenados en `mstatus`.

4.4.4. Modificaciones al diseño base

Además de añadir los nuevos registros CSR es necesario realizar mas modificaciones a la ruta de datos del procesador. Primero hay que añadir bloques lógicos capaces de detectar los diferentes tipos de *traps* a las que se quiere dar soporte.

Las interrupciones externas son señales de entrada a la ruta de datos, simplemente hay que detectar cuando estén activas. Las *traps* relacionadas con las instrucciones `ecall`, `ebreak` y las instrucciones no reconocidas se pueden detectar en la etapa *Decode* durante la decodificación de la instrucción.

Detectar cuando se esta intentando ejecutar una instrucción privilegiada en modo no privilegiado se podría hacer en la etapa *Decode* pero se va a hacer en la etapa *Memory* para simplificar el diseño de la extensión Zicsr explicada en la Sección 4.5.

Los accesos no alineados a memoria se pueden detectar cuando se han calculado las direcciones que se van a utilizar para acceder a ella. Estas direcciones se calculan todas en la etapa *Execution*. Las *traps* de instrucción no alineada solo se pueden dar cuando se intenta saltar a una dirección mal alineada, con lo cual también se detectan en esa etapa.

Cuando una instrucción genera una excepción esta no debe de terminar de ejecutarse, es decir, no debe escribir ninguno de sus resultados ni en memoria ni en el banco de registros ni en el `PC` en al caso de los saltos. Todas las instrucciones realizan cambios permanentes en a partir de la etapa de *Memory*, salvo los saltos, que pueden modificar el `PC` en *Execute*.

En el diseño actual puede haber hasta 5 instrucciones ejecutándose en paralelo, se ha tomado la decisión de que la instrucción interrumpida sea la que se encuentra en la etapa *Memory*. Es decir, si ocurre una *trap* se guardará la dirección de la instrucción que estuviera en esa etapa. Las instrucciones en etapas anteriores se borraran y la instrucción en la etapa *Writeback* terminará de ejecutarse, esta instrucción no se puede borrar por que puede que ya haya realizado algún acceso a memoria cuando pasó por la etapa *Memory*. Cuando se detecte alguna excepción en alguna de las etapas que atraviesa una instrucción esta sigue avanzado hasta la etapa *Memory* donde ocurre la *trap*. La Figura 4.4 muestra un ejemplo de lo que ocurre en la ruta de datos en esta situación.

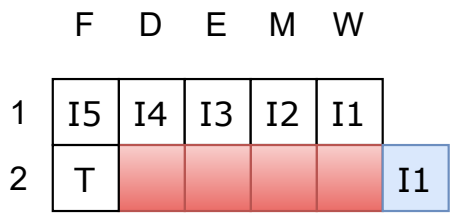

Figura 4.4: En el ciclo 1 se detecta que se ha de realizar una *trap* por lo que se carga la primera instrucción de la función *trap handler* y las instrucciones I2, I3, I4 e I5 son borradas. Para resaltar que la instrucción I1 termina de ejecutarse se ha añadido al ciclo 2 en azul fuera de las etapas.

Al interrumpir un salto en la etapa *Memory* puede que ya haya modificado el `PC`

cuando pasó por la etapa *Execution*, este cambio del `PC` no importa ya que ninguna instrucción posterior a esta ha realizado ningún cambio permanente en el estado del procesador y el `PC` se va a cambiar de nuevo al realizar la *trap*. La Figura 4.5 muestra el comportamiento de la ruta de datos en esta situación.

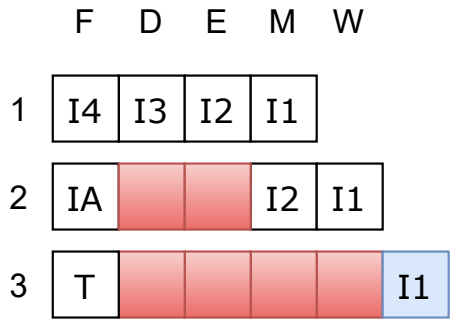

Figura 4.5: En el ciclo 1 la instrucción I2 realiza un salto a la instrucción IA. En el ciclo 2 se detecta que se ha realizar una *trap*, por lo que se borran las instrucciones I2 e IA. Para resaltar que la instrucción I1 termina de ejecutarse se ha añadido al ciclo 3 en azul fuera de las etapas.

Cuando en un mismo ciclo un salto en la etapa *Execution* va a modificar el `PC` y va a ocurrir una *trap* se da prioridad a la *trap* ya que afecta a una instrucción previa al salto.

Al añadir este soporte cuando se borran o se inyectan instrucciones se les debe asignar una dirección válida, de forma que si ocurre una *trap* cuando una de estas instrucciones se encuentre en la etapa *Memory* esta se almacene en `mepc`. Cuando se borran instrucciones como consecuencia de un salto o de un `mret` se les asigna la dirección de la instrucción y cuando se inyectan instrucciones a causa de una dependencia se les asigna la dirección de la instrucción parada en *Decode*.

Si ocurren múltiples causas para una *trap* en un mismo ciclo la ISA define una lista de prioridades para el valor que debe tomar el registro `mcause`. La Tabla 4.2 muestra la lista.

También se tiene que añadir soporte a la instrucción `mret`, se ha decidido que esta instrucción se ejecute en la etapa *Memory*. Esto se ha hecho para simplificar el diseño ya que así se evita que se este ejecutando en paralelo con una *trap*. La Figura 4.6 muestra un diagrama de lo que ocurre cuando se realiza una *trap* y se ejecuta la instrucción `mret` para restaurar el estado.

Realizar una *trap* o ejecutar una instrucción `mret` implica una perdida de 4 ciclos de ejecución debido al borrado de instrucciones de las etapas previas.

La Figura 4.7 muestra la Figura 4.1 actualizada con los nuevos bloques lógicos añadidos. Los nuevos bloques y los antiguos modificados están resaltados en azul.

Tabla 4.2: Prioridades de las posibles causas de una *trap*.

Prioridad	Causa
1	Interrupción *timer*
2	Interrupción externa
3	`ebreak`
4	Instrucción ilegal
5	Instrucción no alineada
6	`ecall` desde modo U
7	`ecall` desde modo M
8	*Load* no alineado
9	*Store* no alineado

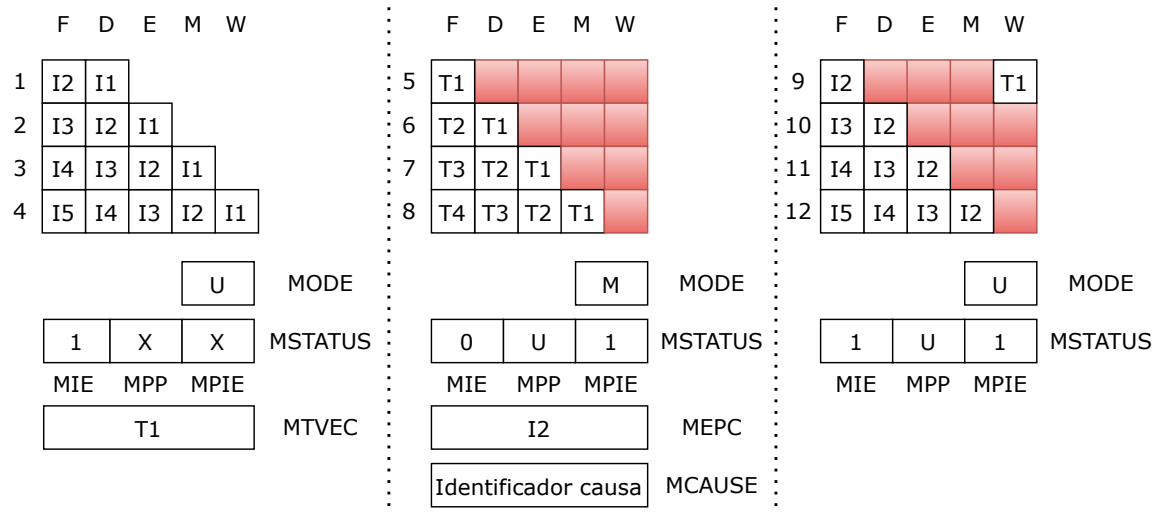

Figura 4.6: En el ciclo 4 se realiza una *trap*, esta interrumpe a la instrucción I2, que se guarda en `mepc`, cambia el modo, los valores de `mstatus` y `mcause` y carga la instrucción T1 indicada por `mtvec`. T1 es un `mret` que se ejecuta en el ciclo 8, restaurando la instrucción I2, el modo y los valores de `mstatus` en el ciclo 9.

4.5. Extensión Zicsr

Los valores que contienen los CSR deben de ser accesibles mediante instrucciones, de esta forma un sistema operativo puede, por ejemplo, configurar la dirección de la función *trap handler* y leer el valor de la causa de la *trap*. La ISA define la extensión estándar Zicsr que añade 6 instrucciones que permiten leer y escribir los valores de estos registros.

En el diseño actual los valores de estos registros solo se modifican y leen en la etapa *Memory* cuando ocurre una *trap* o para detectar los fallos de privilegios. Se ha decidido que estas instrucciones modifiquen los CSR también en la etapa *Memory*, de forma que no pueda darse el caso de 2 accesos en paralelo.

La lectura de los CSR se realiza en la etapa *Decode*. Como la ruta de datos esta segmentada, puede que el valor que se lea no sea el valor correcto por que una instrucción

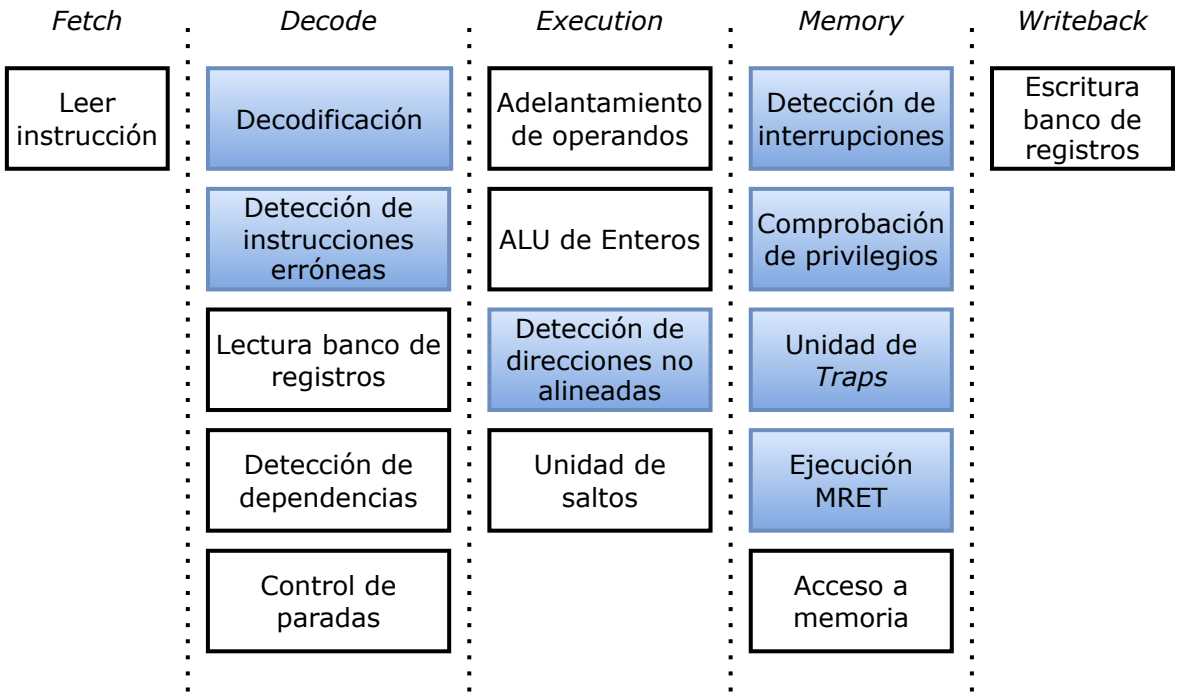

Figura 4.7: Actualización de la Figura 4.1 con los bloques lógicos añadidos o modificados resaltados en azul.

anterior pretenda modificarlo y aun no lo ha hecho. Para simplificar el diseño se ha decidido que no se van a adelantar los valores de los CSR como operandos, es decir, las instrucciones Zicsr no avanzaran de *Decode* si hay otra instrucción Zicsr en las etapas posteriores. La perdida de rendimiento que supone esta decisión es pequeña debido a que los accesos a estos registros no son muy comunes y pocas veces se realizan varios de estos de forma seguida. La Figura 4.8 muestra un ejemplo de como se comporta la ruta de datos cuando se da este caso.

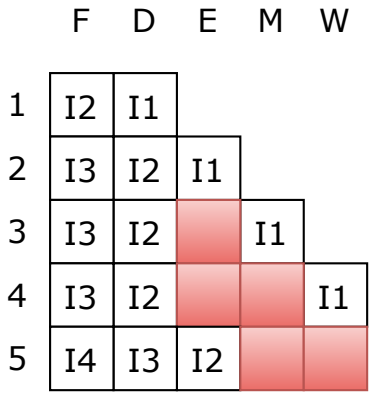

Figura 4.8: Las instrucciones I1 e I2 son Zicsr, I2 en su etapa *Decode* detecta otra instrucción Zicsr en etapas posteriores por lo que para 2 ciclos hasta que termina de ejecutarse.

Para poder añadir al diseño esta extensión, hay que añadir a la etapa *Decode* la lógica necesaria para decodificar estas nuevas instrucciones y para leer los diferentes CSR, a la etapa *Execution* los bloques lógicos necesarios para hacer las operaciones que requieren estas instrucciones y a la etapa *Memory* la lógica de escritura de estos registros y el control de privilegios de estas instrucciones.

La Figura 4.9 muestra la Figura 4.7 actualizada con los nuevos bloques lógicos añadidos. Los nuevos bloques y los antiguos modificados están resaltados en azul.

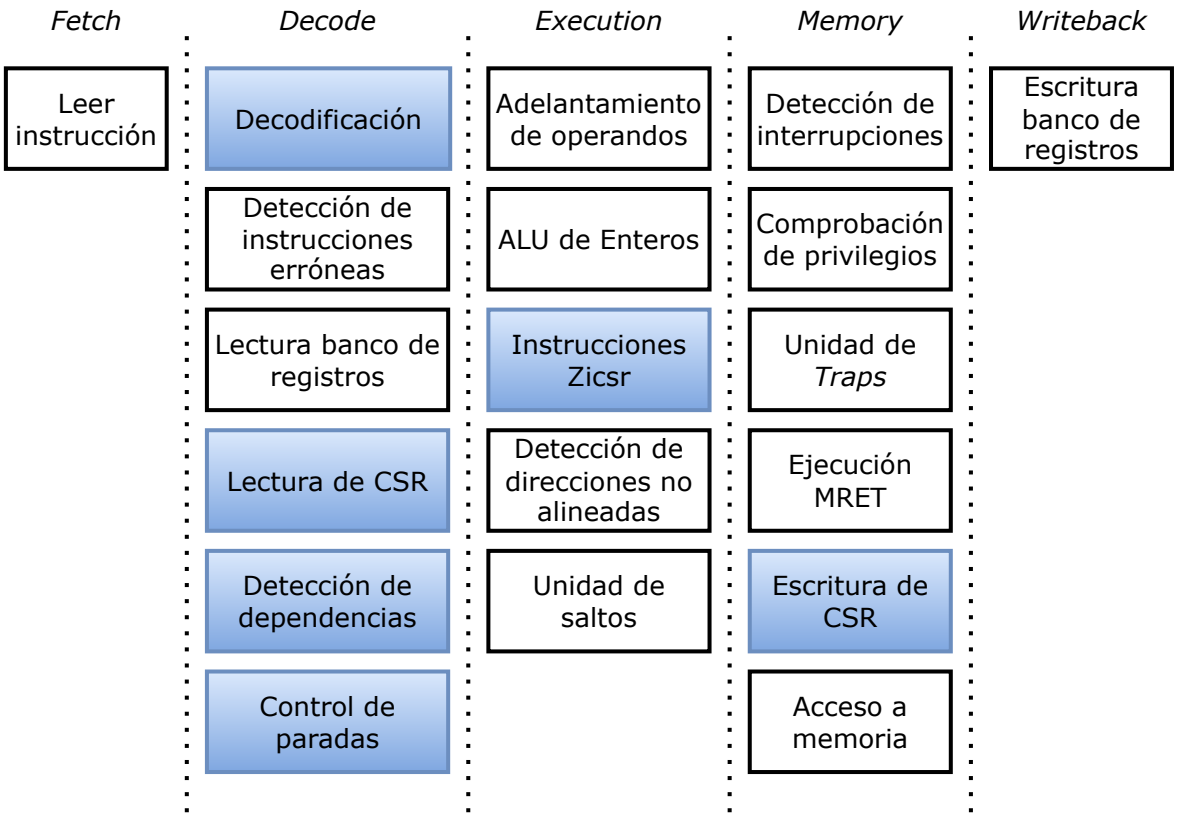

Figura 4.9: Actualización de la Figura 4.7 con los bloques lógicos añadidos o modificados resaltados en azul.

Estas instrucciones son privilegiadas por que modificar el valor de estos registros podría permitir a una aplicación tomar el control completo del sistema, por ejemplo, cambiando el valor de la función *trap handler*.

Si una de estas instrucciones va a acceder a un registro que no esta implementado o va cambiar algún valor que controla una funcionalidad no implementada la ISA define el tipo de valor WARL (*Write Any Read Legal*), permitiendo que en la lectura de estos siempre se devuelva un valor legal para el diseño y la escritura se ignore. Para la mayoría de registros un valor legal siempre es 0, que indica que la funcionalidad no se ha implementado.

4.6. Entrada y salida

Para un procesador, su comunicación con el exterior es fundamental. Ya sea para recibir datos de entrada, compartir resultados o configurar periféricos. Uno de los mecanismos mas utilizados para esto es la entrada y salida mapeada en memoria, MMIO (*Memory Mapped Input Output*).

Este mecanismo consiste en asociar diferentes registros con los que se comunican los periféricos a posiciones de memoria. Es decir, cuando se realice algún acceso a memoria utilizando esas direcciones con una instrucción *load/store* en vez de acceder a memoria se accede al registro asociado a la dirección. Para añadir esta funcionalidad, al diseño se ha decidido implementar un bus compartido por los diferentes periféricos y la memoria. La Figura 4.10 muestra un esquema de la red de interconexión del procesador con el resto de componentes.

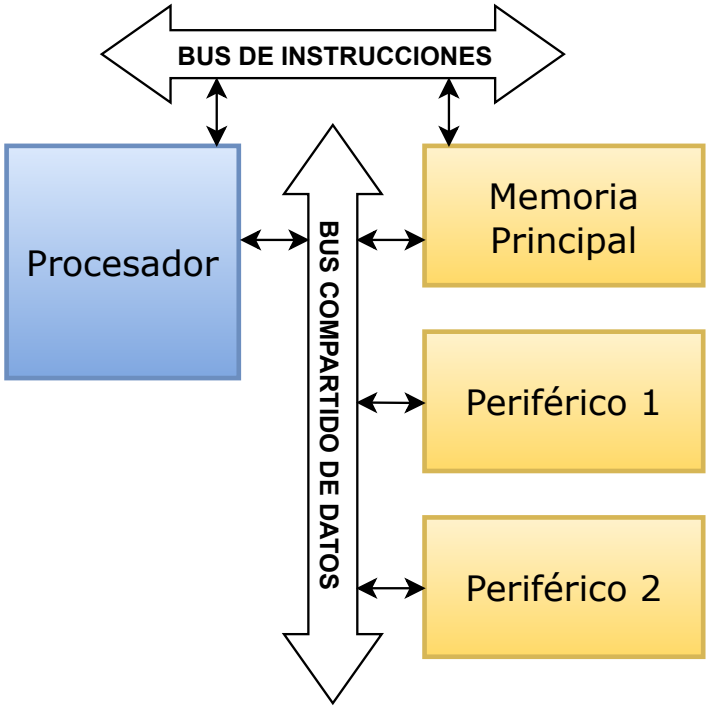

Figura 4.10: Red de interconexión mediante un bus compartido del procesador con el resto de componentes.

El bus compartido sigue el modelo controlador/objetivo, este modelo consiste en que el controlador escribe una orden en el bus y los objetivos la realizan. En este caso el controlador es el procesador y los objetivos son los diferentes registros de los periféricos y la memoria.

El procesador en un primer ciclo escribe una dirección, un tipo de acceso y un dato en caso de que el tipo de acceso sea una escritura. Todos los objetivos comprueban si la dirección de memoria del bus está asociada a ellos y en caso de que lo esté realizan el

acceso. Al ciclo siguiente los objetivos responden con un dato si el tipo de acceso era una lectura.

Al hacer esto el espacio de direccionamiento queda dividido entre las direcciones que pertenecen a memoria principal, con lo cual pueden contener tanto instrucciones como datos, y las direcciones que pertenecen a registros de periféricos que no pueden contener instrucciones.

4.6.1. Temporizador

Uno de los requisitos básicos de un sistema operativo es que pueda comunicarse con un temporizador para poder medir el tiempo. La ISA define un periférico temporizador llamado `MTIMER` el cual se puede configurar mediante 2 registros mapeados en memoria. Se ha diseñado una implementación de este periférico y se ha añadido al sistema.

El objetivo de este periférico es generar interrupciones periódicas para poder medir el tiempo. Consiste en un registro de 64 bits, `mtime` cuyo valor se incrementa cada ciclo y otro registro de 64 bits, `mtimecmp`. Cuando el valor de `mtime` es mayor o igual que el de `mtimecmp` se genera una interrupción. Estos registros se pueden acceder mediante el mecanismo de MMIO para leer o modificar sus valores. Como estos registros son de 64 bits y la ruta de datos y los buses de 32 se tiene que acceder en accesos diferentes a los 32 bits bajos o a los 32 bits altos. La figura 4.11 muestra un diagrama del diseño del temporizador.

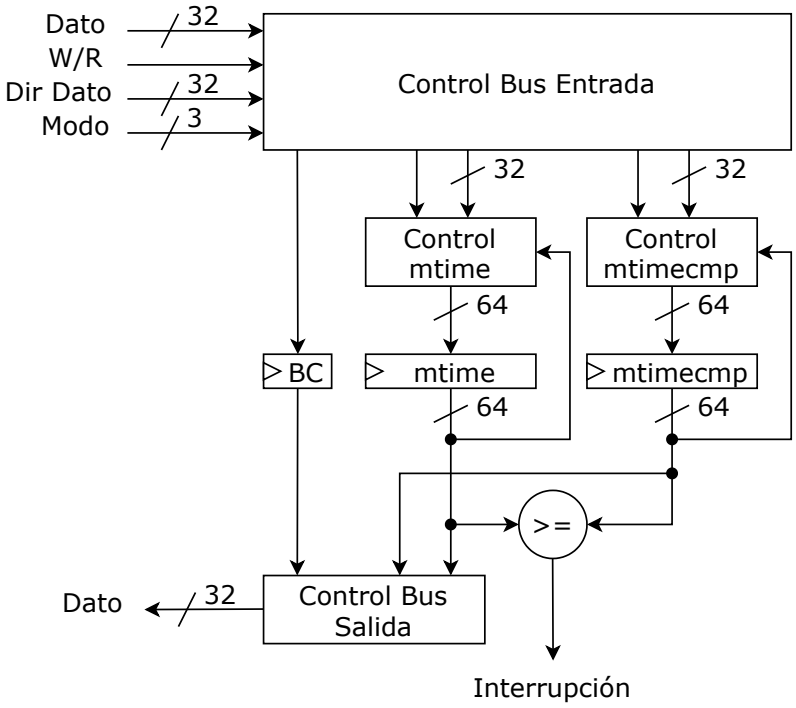

Figura 4.11: Diagrama del diseño del temporizador `mtimer`.

Capítulo 5

Validación del diseño

5.1. Programas de prueba en ensamblador

Un apartado muy importante de este proyecto ha sido validar que el diseño implementado cumple con la especificación. RISC-V ofrece un repositorio de GitHub [9] con un conjunto de test oficiales escritos en ensamblador para probar esto. Hay tests completos para el conjunto de instrucciones base RV32I y la extensión Zicsr. Para los modos privilegiados hay algunos tests pero no son exhaustivos.

Se han adaptado estos tests para poder utilizarlos. Además se diseñaron más para los modos privilegiados de forma que se comprobaran situaciones concretas relativas al comportamiento del diseño. También se diseñaron test para comprobar el sistema de entrada y salida y el temporizador. El total 48 programas de prueba.

Para poder simular la ejecución de uno de estos test primero se tienen que cargar las instrucciones de este en los bancos de memoria. Para ello implementó un pequeño compilador en Python que desde un fichero en ensamblador de RISC-V genera los ficheros necesarios de VHDL para simular el contenido de las memorias.

Primero se realizaron las pruebas para el modo no privilegiado, estas consistían en validar las el conjunto de instrucciones base, después las pruebas con distintos modos de ejecución, estas consistían en comprobar el correcto funcionamiento del mecanismo de *traps* y las instrucciones Zicsr, y por ultimo se validó el sistema de entrada y salida con el periférico temporizador.

Con el objetivo de automatizar el proceso de pruebas se diseñó un periférico con un registro mapeado en memoria. Cuando se escribe en este registro se imprime el valor escrito por la salida estándar del simulador, de forma que mediante un *script* de Bash se puede analizar la salida. Se modificaron los test para que informaran de si se habían superado o no mediante escrituras en este registro.

5.2. Entorno de C

Los programas mas complejos, como puede ser un sistema operativo, no están escritos solamente en ensamblador, están escritos en lenguajes de mas alto nivel como C. Uno de los objetivos finales es poder ejecutar FreeRTOS y parte de este está escrito en C, por lo que primero se va a crear un entorno que permita simular la ejecución de programas escritos en este lenguaje sobre el diseño.

Se ha utilizado la RISC-V GNU *Compiler Toolchain*, que es un conjunto de herramientas de código libre que permite compilar y enlazar código en C y C++ para RISC-V.

5.2.1. Compilación

El primer paso es poder compilar código en C a lenguaje máquina RISC-V. Esta tarea la realiza el compilador, a partir de un fichero de código fuente genera un fichero objeto con el código traducido a lenguaje máquina.

Se tiene que configurar el compilador para que genere código con instrucciones soportadas por la plataforma objetivo, es decir hay que indicarle que extensiones estándar soporta dicha plataforma. Por ejemplo, existe una extensión estándar que ofrece soporte a la multiplicación y a la división añadiendo instrucciones para estas, pero el diseño no la soporta, por lo que si el compilador esta bien configurado en vez de utilizar estas instrucciones añadirá funciones al código que las emulen.

También se tiene que indicar al compilador que ABI (*Application Binary Interface*) ha de utilizar, esta debe de ser compatible con la plataforma objetivo. Las diferentes ABI están definidas por el grupo RISC-V [10]. Entre otras cosas, especifica el tamaño en bytes de los diferentes tipos de datos de alto nivel y que procedimiento de invocación de funciones se ha de utilizar.

5.2.2. Enlazado

Después de generar todos los ficheros objeto estos se han de enlazar para crear un fichero ejecutable en formato ELF (*Executable and Linkable Format*). Esta tarea la realiza el enlazador. Durante el proceso de enlazado se crea el mapa de memoria del programa, lo que implica que se asignan las direcciones de memoria finales para los datos e instrucciones.

Este mapa de memoria debe de ser compatible con el mapa de memoria de la plataforma objetivo, es decir, las direcciones asignadas a código y datos deben de pertenecer a memoria principal y no a registros de periféricos y la primera instrucción a ejecutarse debe encontrarse en la dirección cuyo valor sea el que tome el `PC` al reiniciarse.

En el procesador diseñado esa dirección es la 0, y las direcciones asignadas a memoria principal empiezan siempre desde ella.

Para indicarle al enlazador como crear un mapa de memoria correcto se utiliza un *linker script*, que define como se deben de mapear en memoria las secciones de los distintos ficheros objeto. La Figura 5.1 muestra un ejemplo de un posible mapeado de las diferentes secciones de un programa en el espacio de direccionamiento del procesador diseñado.

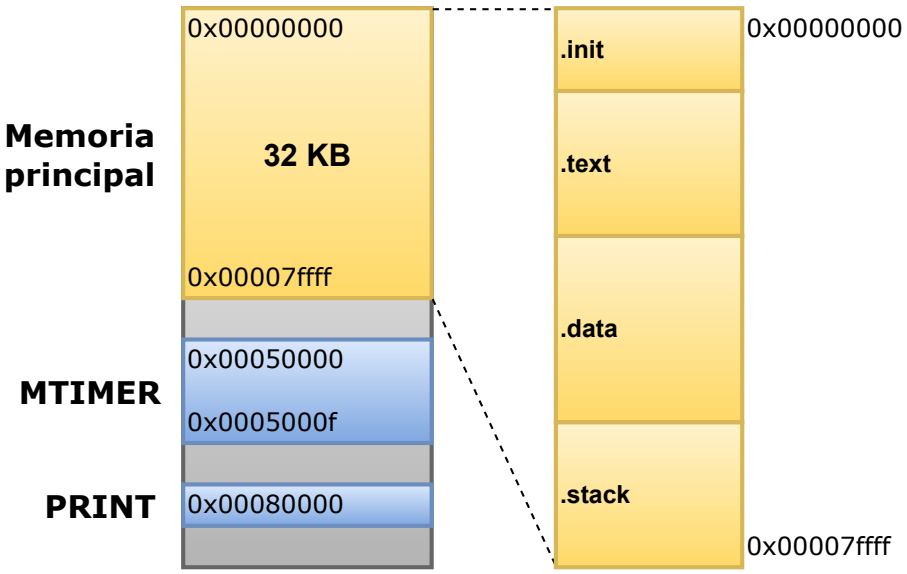

Figura 5.1: Mapeado de las secciones de un programa en el espacio de direccionamiento del procesador diseñado.

En el ejemplo de la Figura 5.1 la sección `.init` contiene las primeras instrucciones que se van a ejecutar. Estas instrucciones puede ser simplemente un salto a la función principal del programa en C o pueden realizar algunas configuraciones previas, la sección `.text` contiene las instrucciones de las distintas funciones del programa, la sección `.data` contiene los datos del programa y reserva espacio par las diferentes variables de este y la sección `.stack` es un espacio reservado para la pila de ejecución.

5.2.3. Formato ELF

Una vez se ha generado el fichero en formato ELF se tienen que generar los ficheros de VHDL necesarios para cargar los bancos de memoria y poder simular la ejecución del programa. Para ello se han de extraer las instrucciones y datos que hay almacenados en el fichero. Esto se ha hecho con la ayuda de un *script* en Python que hace uso de la librería *pyelftools* [11], que permite trabajar de forma cómoda con ficheros en formato ELF.

En los ficheros ELF hay 2 tablas de cabeceras, una de ellas es la *Program Header*

Table, que se utiliza para cargar el programa en memoria, y la otra es la *Section Header Table*, que se utiliza en el proceso de enlazado.

La tabla que se va a utilizar es la *Program Header Table*, que contiene las cabeceras de los segmentos de memoria que forman el programa. Cada cabecera indica si el segmento debe ser cargado para ejecutar el programa o no y donde esta su contenido dentro del fichero. La Figura 5.2 muestra un diagrama del formato ELF.

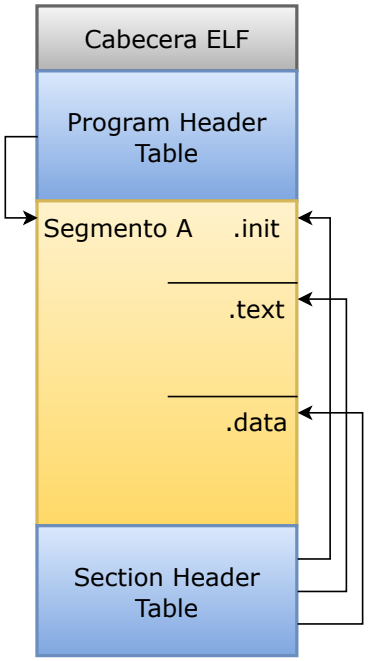

Figura 5.2: Diagrama del contenido de un fichero ELF. El segmento A contiene todas las secciones que se han de cargar del programa.

Lo que hace el *script* es buscar en la *Program Header Table* las cabeceras de los segmentos que se han de cargar y crea los ficheros VHDL necesarios para simular las memorias con el contenido de dichos segmentos.

5.3. FreeRTOS

Una vez se tenia un entorno capaz de compilar y simular la ejecución de un programa escrito en C en el diseño, se adaptó una de las versiones de prueba oficiales que ofrece FreeRTOS para plataformas RISC-V utilizando su guía oficial de como crear una versión del sistema para una nueva plataforma [12].

5.3.1. Requisitos

El objetivo de los sistemas operativos de tiempo real es ser ligeros y poder ejecutarse en plataformas de bajo coste sin problemas. Para poder ejecutar FreeRTOS sobre una

plataforma RISC-V es necesario que esta tenga soporte a la gestión de excepciones e interrupciones y algún tipo de temporizador. En caso de que el temporizador sea el temporizador `MTIMER` especificado por la ISA, FreeRTOS ya tiene implementada la funcionalidad para interactuar con el.

5.3.2. Configuración

FreeRTOS es altamente configurable, permitiendo configurar cuales de sus funcionalidades se van a utilizar y como. Para ello se utiliza un fichero de configuración llamado `FreeRTOSConfig.h` [13]. Además también se puede cambiar el código fuente del propio sistema si se quiere.

Temporizador

Para los procesadores RISC-V que utilicen el `MTIMER` en este fichero se tienen que definir las direcciones en las que se encuentran mapeados sus registros de control, de forma que FreeRTOS pueda configurar sus funciones de control de este periférico para que interactúen con las direcciones adecuadas. También se tienen que definir la frecuencia de reloj del procesador y la frecuencia del *tick* de sistema, es decir, cada cuanto se quiere que el temporizador genere una interrupción. En cada interrupción se tiene que incrementar el valor de `mtimecmp` de forma que el valor de `mtime` no lo supere o iguale hasta que haya transcurrido el periodo del *tick* de sistema. Utilizando los valores la frecuencia del reloj y del *tick* se puede calcular el valor con el que se debe incrementar.

Planificador

Se tiene que configurar el tipo de planificador que se quiere utilizar, cooperativo o preemptivo. Se ha configurado como preemptivo. En cada *tick* del sistema el planificador comprueba si la tarea que tiene el control del procesador en ese momento es la tarea activa mas prioritaria, en caso de que no lo sea cambia la tarea en ejecución a la que lo sea. Las tareas también pueden ceder el control al planificador de forma voluntaria si necesitan bloquearse para esperar a algún evento.

Gestión de memoria

También se tiene que configurar como se asignan en memoria los diferentes objetos que el sistema crea durante sus ejecución. Esto se puede hacer de forma estática, teniendo que definir en tiempo de compilación donde se van a asignar, o de forma dinámica. La asignación dinámica consiste en que el sistema decide donde asignar los

objetos utilizando un conjunto de funciones en tiempo de ejecución. FreeRTOS ofrece distintos modelos de gestión de memoria dinámica [14]. Se ha utilizado el modelo de memoria dinámica `heap_4`.

Además se tienen que configurar el tamaño máximo y mínimo de las pilas para las diferentes tareas e interrupciones y el tamaño de la zona reservada para memoria dinámica. A la hora de asignar el tamaño a estas regiones se tienen que tener en cuenta que junto con el código del sistema y las tareas todo debe de poderse almacenar en la memoria principal. También hay que tener cuidado de no hacer las regiones demasiado pequeñas o puede que desborden en tiempo de ejecución, el sistema incluye una funcionalidad para detectar errores de desbordamiento.

Funciones opcionales

Si se quiere se puede configurar que se invoquen funciones opcionales definidas por la aplicación en algunos eventos. Un ejemplo de esta funcionalidad es que se puede configurar una función para que se invoque en cada *tick* del sistema o cuando ocurre un desbordamiento.

5.3.3. Aplicación de pruebas y simulación

Para probar la corrección del sistema se ha diseñado una pequeña aplicación de prueba que funciona sobre este. La aplicación consiste en 2 tareas que se envían un valor a través de una cola gestionada por el sistema cada segundo. Una de las tareas se ejecuta periódicamente cada segundo y escribe valor incremental en la cola. La otra lee los valores de la cola, esta lectura bloquea a la tarea si no hay ningún valor en ella, y comprueba que los valores siguen una secuencia incremental.

Se tienen que simular varios segundos para poder comprobar que la aplicación está funcionando correctamente. Varios segundos pueden llegar a ser muchos millones de ciclos de reloj dependiendo de la frecuencia de este. Simular muchos ciclos de reloj es muy costoso ya que requiere simular todos los cambios de todas las señales internas del diseño. Para intentar reducir este problema se utilizaron frecuencias bajas de reloj y se aumento la frecuencia de la tarea de escritura aunque aun así el proceso de simulación era muy largo hasta que se podía obtener un resultado suficiente para validar el comportamiento.

Capítulo 6

Implementación en FPGA

6.1. Adaptación del diseño

Se ha utilizado la plataforma de desarrollo Zedboard para el SoC (*System on Chip*) Xilinx Zynq-7000. Este SoC esta formado por un procesador de propósito general ARM y una FPGA. La Figura 6.1 muestra un diagrama de bloques de la plataforma.

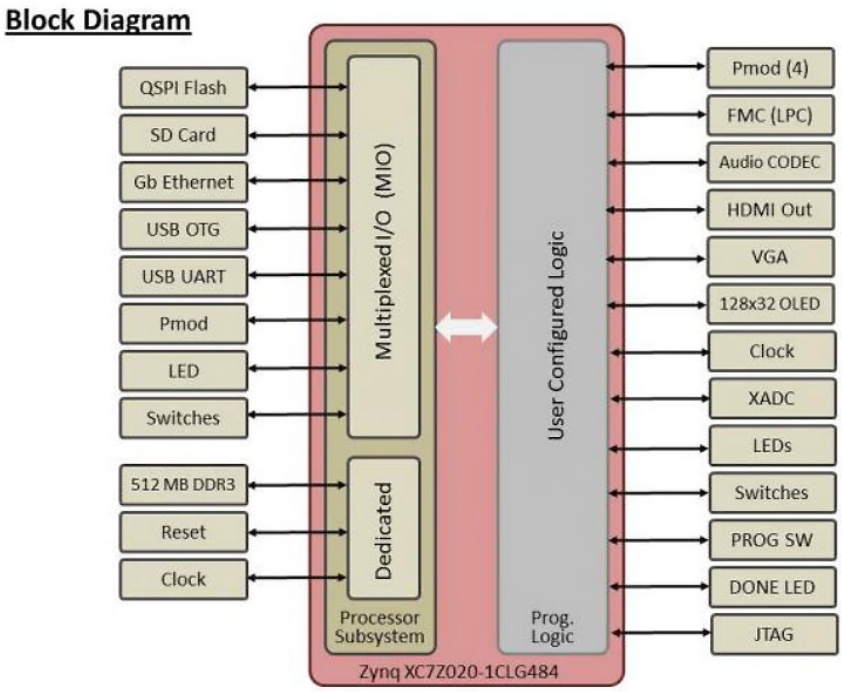

Figura 6.1: Diagrama de bloques de la plataforma Zedboard. Figura de `https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/zedboard/`

El procesador de propósito general ARM puede programarse de forma sencilla utilizando las herramientas que ofrece Xilinx para ello. Además los drivers para interactuar con los periféricos como la UART ya están implementados. Por lo tanto se

ha decido utilizar los drivers de la UART del procesador ARM para obtener una salida que se pueda visualizar.

Para mostrar lo que esta ocurriendo en el procesador RISC-V este tiene que comunicarse con el procesador ARM. Para esto, se ha añadido un periférico con 3 registros mapeados en memoria al procesador RISC-V, estos registros también se mapean en la memoria del procesador ARM, esto se hace utilizando un bus AXI4-Lite [15]. Este bus permite al procesador ARM escribir y leer registros que se encuentren en la FPGA.

También se ha diseñado un bloque de control que permite al procesador ARM parar el reloj del procesador RISC-V y tomar control de su bus compartido de datos. A través de este bloque de control el procesador ARM puede escribir en la memoria principal del procesador RISC-V para cambiar el programa que este ejecuta. Este bloque de control esta formado por registros que tambien se acceden mediante un bus AXI4-Lite. La Figura 6.2 muestra el diagrama de bloques del diseño que se ha implementado en la plataforma.

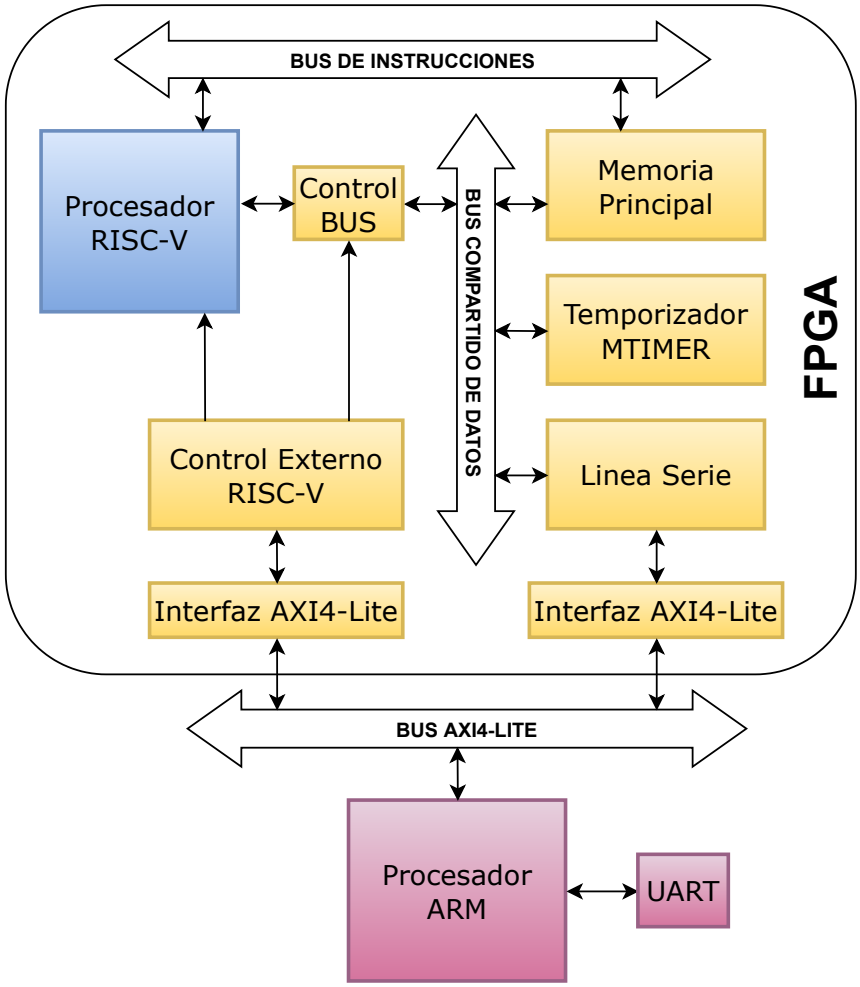

Figura 6.2: Diagrama de bloques y comunicación del diseño implementado en la FPGA.

Se han tenido que diseñar 2 protocolos sencillos de comunicación. El primero de ellos es para que el procesador ARM pueda escribir en la memoria principal del procesador RISC-V. Como la frecuencia de reloj del procesador ARM es diferente a la del reloj de la FPGA y los datos tienen que atravesar la red de interconexión hasta llegar a su destino, el procesador ARM debe esperar a una confirmación de que los datos se han escrito. La Figura 6.3 muestra un diagrama de comunicación del protocolo.

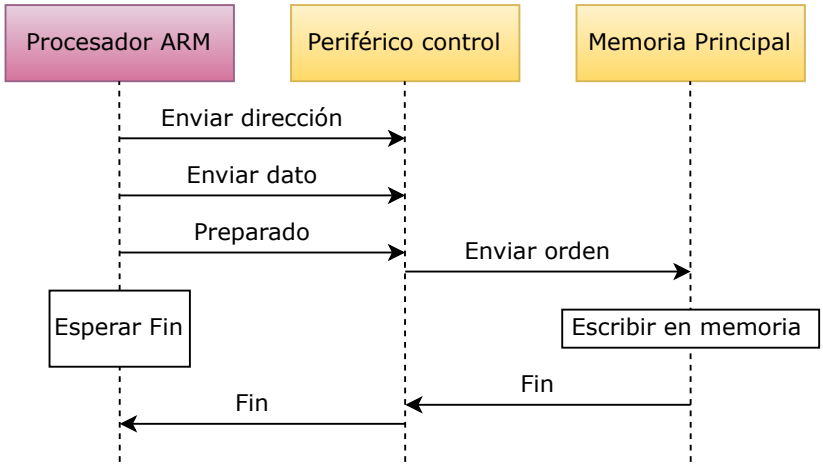

Figura 6.3: Diagrama de comunicación del procesador ARM con la memoria principal del procesador RISC-V mediante un periférico de control.

El siguiente protocolo que se ha diseñado se ha utilizado para que el procesador RISC-V envié valores al procesador ARM y este los muestre a través de la UART. Para ello se ha tenido que escribir una función en ambos procesadores y hacer que estos se sincronicen. La Figura 6.4 muestra un diagrama de comunicación del protocolo.

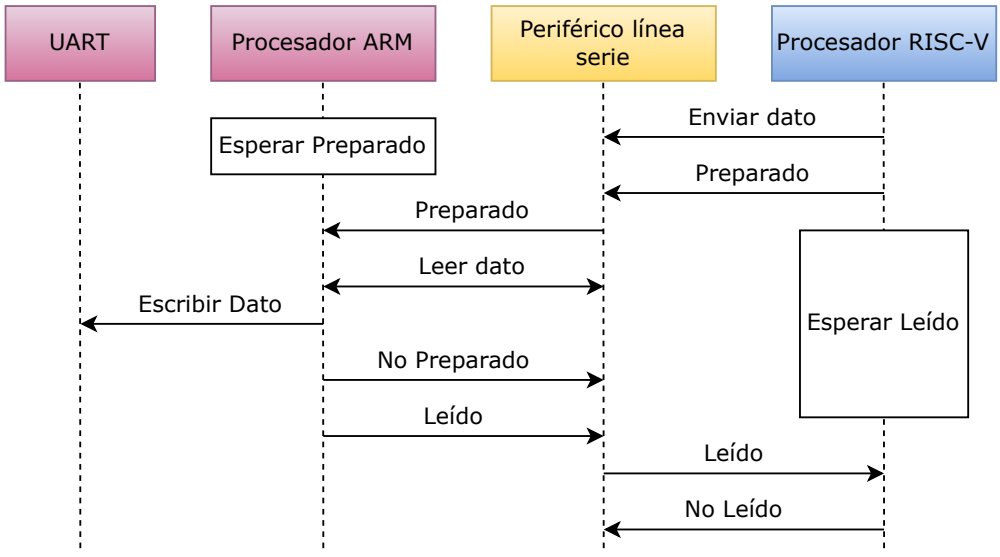

Figura 6.4: Diagrama de comunicación del procesador ARM con el procesador RISC-V mediante un periférico.

6.2. Proceso de síntesis

Las herramientas de síntesis durante este proceso generan la configuración necesaria para que los bloques lógicos y la red de interconexión entre bloques de una FPGA se comporten de acuerdo a un circuito especificado. El proceso de sintetizado es bastante costoso en tiempo por lo que se ha intentado reducir al mínimo el número de veces que se ha de realizar añadiendo al diseño el soporte de programar la memoria principal del procesador RISC-V.

6.2.1. Resultados obtenidos

La Tabla 6.1 muestra los componentes en los que se ha sintetizado el diseño.

Tabla 6.1: Resultados de la síntesis.

Componente	Cantidad
Sumadores	27
XORs	1
Registros	202
RAMs 8 kB	4
Multiplexores	234
PS7	1

La Figura 6.5 muestra la utilización de los bloques de la FPGA. Como se puede apreciar es muy baja, es decir el diseño es de muy bajo coste. Este se podría extender con distintos aceleradores o con extensiones estándar de la propia ISA RISC-V.

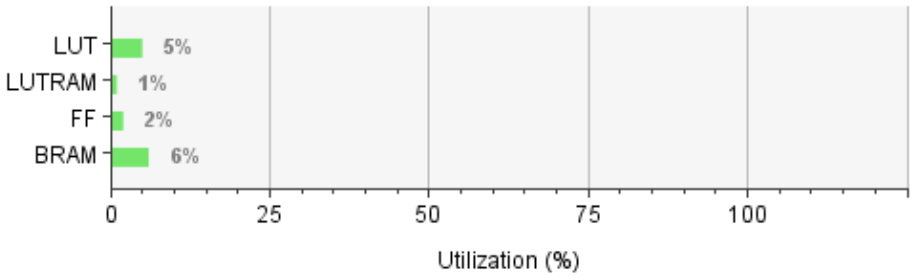

Figura 6.5: Utilización de los bloques de la FPGA Xilinx Zynq-7000.

La frecuencia del reloj de la FPGA de la plataforma utilizada es de 100 MHz. Para esta frecuencia la herramienta no es capaz de dar una configuración para el diseño para la que este completamente segura de que se cumplen los requisitos temporales de las señales. A pesar de esto el diseño funciona correctamente una vez implementado en la FPGA, esto se debe a que las herramientas toman precauciones de más a la hora de asegurar la temporización. Si se quiere eliminar el aviso de fallo en la temporización

una posible solución sería bajar la frecuencia de reloj o cambiar la especificación para reducir los caminos que estén generando mas retardo en las señales.

La Figura 6.6 muestra los bloques lógicos ocupados por el diseño remarcados en azul.

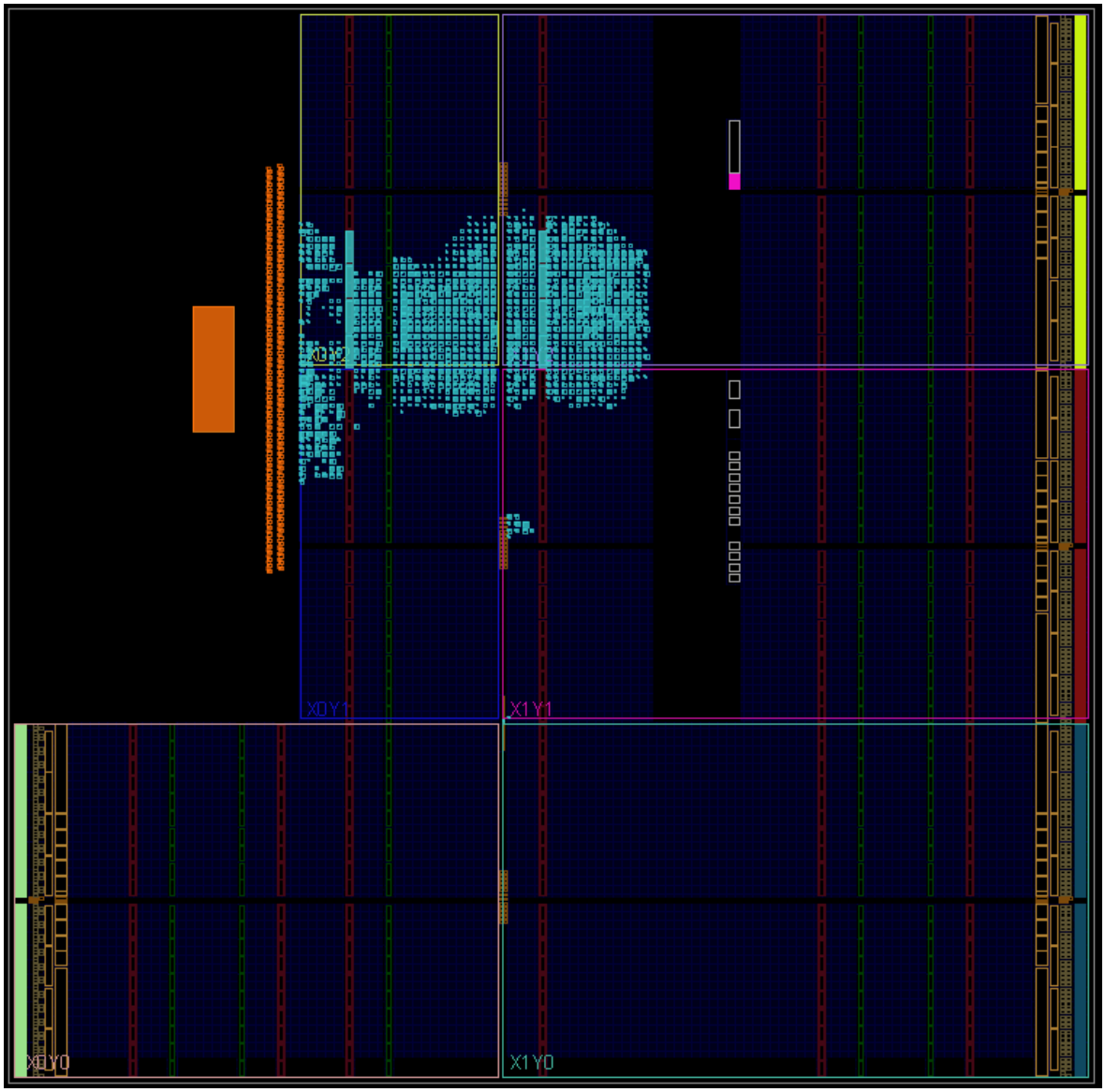

Figura 6.6: Diagrama con los bloques utilizados remarcados en azul en la FPGA Xilinx Zynq-7000.

Capítulo 7

Conclusiones y trabajo futuro

Se ha conseguido cumplir con el objetivo principal del proyecto: el procesador RISC-V diseñado es capaz de ejecutar el sistema operativo de tiempo real FreeRTOS tanto en simulación como una vez implementado en una FPGA. Además junto con el procesador se ha desarrollado un entorno robusto de pruebas para validar el diseño tanto mediante simulación como mediante ejecución en FPGA. Este proyecto demuestra que un procesador RISC-V de muy bajo coste es capaz de ofrecer el soporte necesario para ejecutar un sistema operativo ligero.

Aparte de diseño *hardware* una gran parte de este trabajo también ha sido diseño de *software* para arranque de plataformas, interacción con periféricos, rutinas de gestión de interrupciones, llamadas al sistema y tests.

Este trabajo cubre una gran parte de las asignaturas de la especialidad y del grado relacionadas con la arquitectura de computadores, sistemas operativos y programación de sistemas. Me ha permitido afianzar los conocimientos que ya había adquirido durante el grado, entender mejor el soporte que dan las ISA para los sistemas operativos y formarme en tecnologías con gran proyección de futuro como RISC-V o las FPGA.

Este proyecto da pie a poder extenderse en el futuro. Se podría ampliar el diseño del procesador con mas extensiones estándar de RISC-V o aplicar técnicas de optimización mas agresivas a la ruta de datos como puede ser la ejecución fuera de orden.

Bibliografía

[1] Duncan Stewart y col. *RISC-V business: Could open chip standard RISC-V gain traction against dominant incumbents?* Dic. de 2021. URL: https://www2.deloitte.com/xe/en/insights/industry/technology/technology-media-and-telecom-predictions/2022/risc-v-open-source-cpu.html (visitado 16-06-2022).

[2] EUROPA PRESS. *El BSC-CNS e Intel crearán un laboratorio pionero de diseño de procesadores*. Mayo de 2022. URL: https://www.europapress.es/catalunya/noticia-bsc-cns-intel-crearan-laboratorio-pionero-diseno-procesadores-20220524133615.html (visitado 16-06-2022).

[3] Business Wire. *SiFive Leadership in RISC-V Powers $2.5B+ Company Valuation*. Feb. de 2022. URL: https://www.bloomberg.com/press-releases/2022-03-16/sifive-leadership-in-risc-v-powers-2-5b-company-valuation (visitado 16-06-2022).

[4] Andrew Waterman y col. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Inf. téc. UCB/EECS-2011-62. EECS Department, University of California, Berkeley, mayo de 2011. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html.

[5] Editores Andrew Waterman y Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation. Dic. de 2019.

[6] Editores Andrew Waterman, Krste Asanović y John Hauser. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*. RISC-V International. Dic. de 2021.

[7] D. Koufaty y D.T. Marr. «Hyperthreading technology in the netburst microarchitecture». En: *IEEE Micro* 23.2 (2003), págs. 56-65. DOI: 10.1109/MM.2003.1196115.

[8] W. Stallings. «Reduced instruction set computer architecture». En: *Proceedings of the IEEE* 76.1 (1988), págs. 38-55. DOI: 10.1109/5.3287.

[9] Tim Newsome, Andrew Waterman, Yunsup Lee y col. *riscv-tests*. URL: https://github.com/riscv-software-src/riscv-tests (visitado 17-06-2022).

[10] Kito Cheng, Andrew Waterman y Jessica Clarke. *RISC-V Calling Conventions*. URL: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc (visitado 17-06-2022).

[11] Eli Bendersky, Yann Rouillard y col. *pyelftools*. URL: https://github.com/eliben/pyelftools (visitado 17-06-2022).

[12] FreeRTOS. *Using FreeRTOS on RISC-V Microcontrollers*. URL: https://www.freertos.org/Using-FreeRTOS-on-RISC-V.html (visitado 18-06-2022).

[13] FreeRTOS. *Customisation*. URL: https://www.freertos.org/a00110.html (visitado 18-06-2022).

[14] FreeRTOS. *Memory Management*. URL: https://www.freertos.org/a00111.html (visitado 18-06-2022).

[15] *AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite, Revision E*. ARM. Feb. de 2013.

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Gestión del proyecto

A.1. Diagrama de Gantt

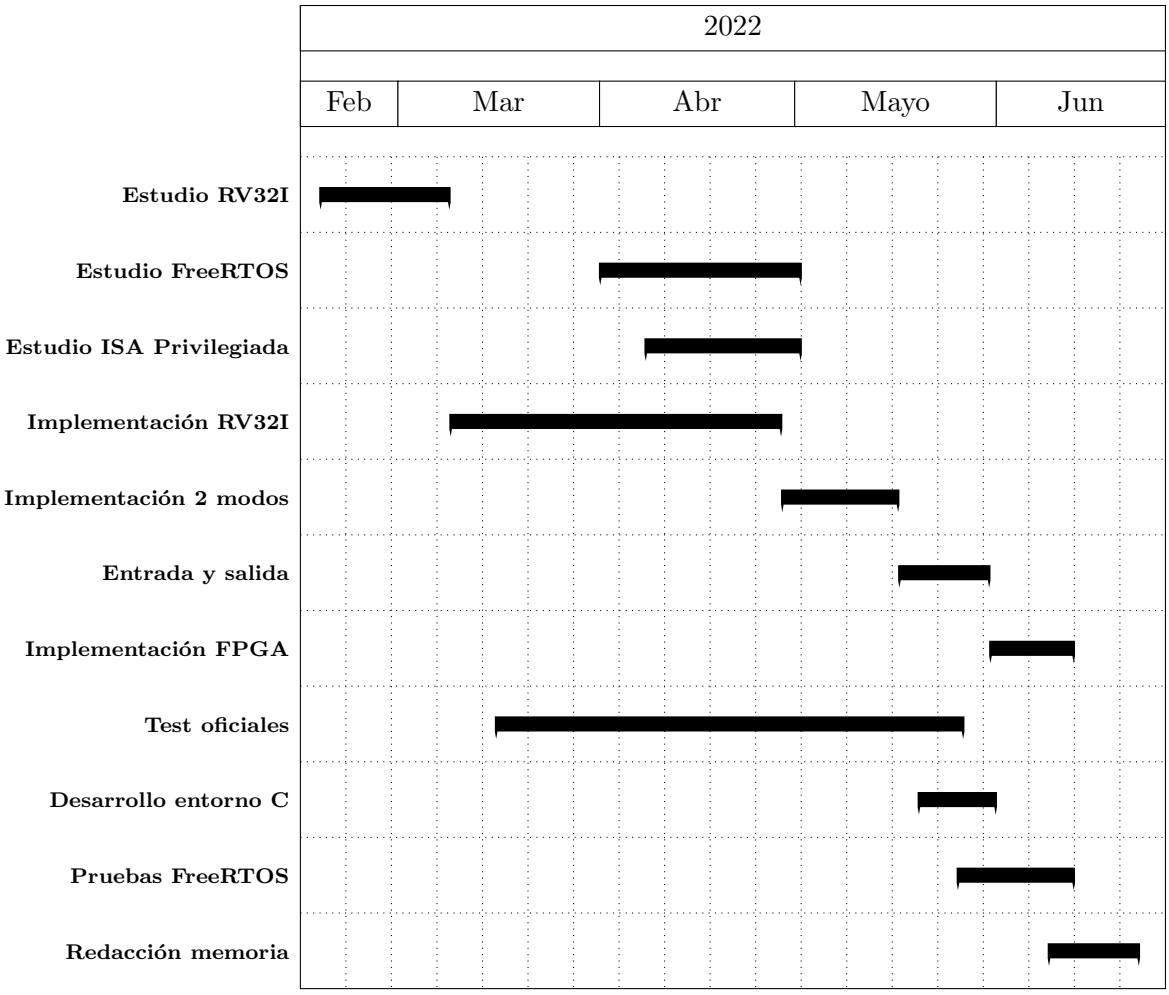

Figura A.1: Diagrama de Gantt del trabajo.

A.2. Dedicación a cada tarea

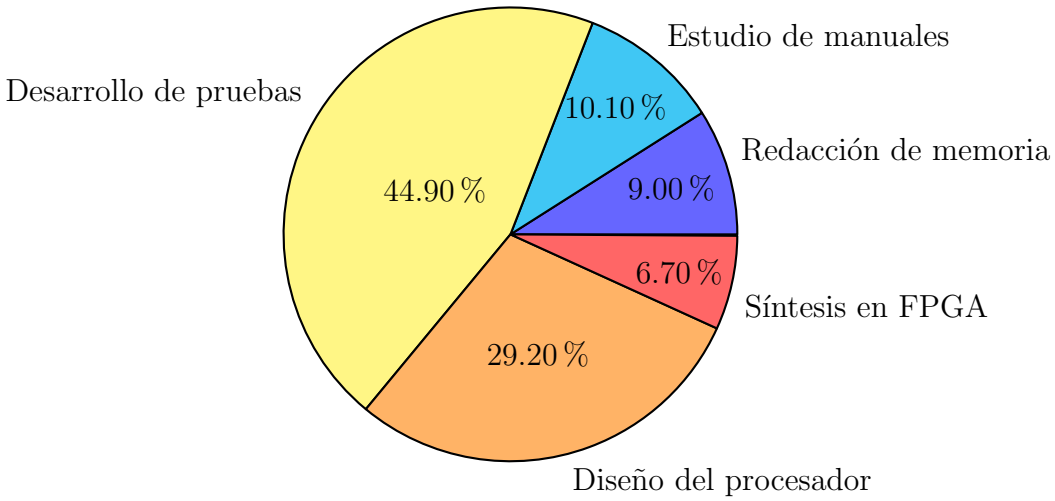

Figura A.2: Porcentaje de tiempo dedicado a cada grupo de tareas.

Anexos B

Codificación y tipos de instrucciones RISC-V

B.1. Tipos de codificación

La ISA define como se codifican las instrucciones, para ello define 6 formatos diferentes que se utilizan dependiendo de la instrucción codificada. La Figura B.1 muestra los diferentes y los campos que contienen.

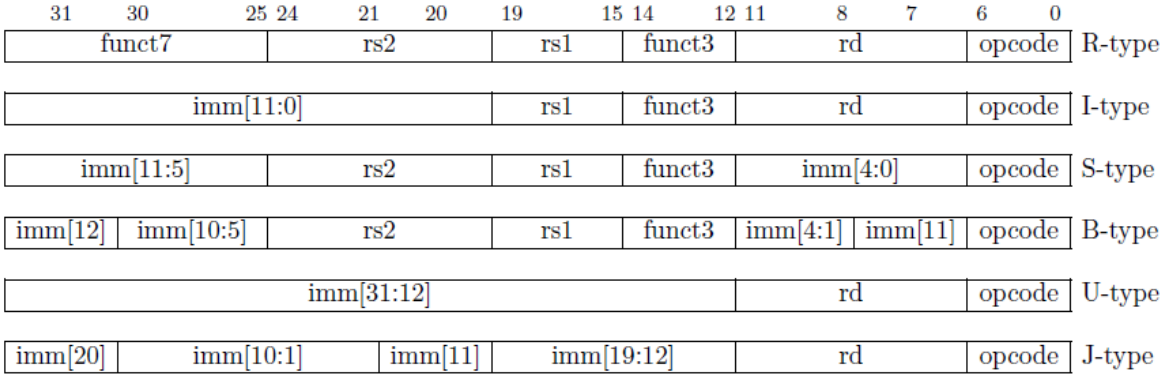

Figura B.1: Formatos de instrucción RISC-V. Figura de [5].

- El campo `opcode` contiene un valor que permite identificar el tipo de instrucción. Siempre se encuentra en los 7 bits de menor peso para poder averiguar el tipo de codificación que está utilizando la instrucción.
- El campo `rd` es de 5 bits y contiene el identificador del registro destino.
- Los campos `rs1` y `rs2` son de 5 bits y contiene los identificadores de los registros fuente.
- Los campos `funct7` y `funct3` se utilizan para diferenciar entre instrucciones que utilizan la misma codificación.

– Muchas instrucciones pueden utilizar como operandos fuente valores codificados en la propia instrucción, a estos se les llaman valores inmediatos y se encuentran en los campos denominados como `imm`. La Figura B.2 muestra como se forman estos valores a partir de los bits de la instrucción.

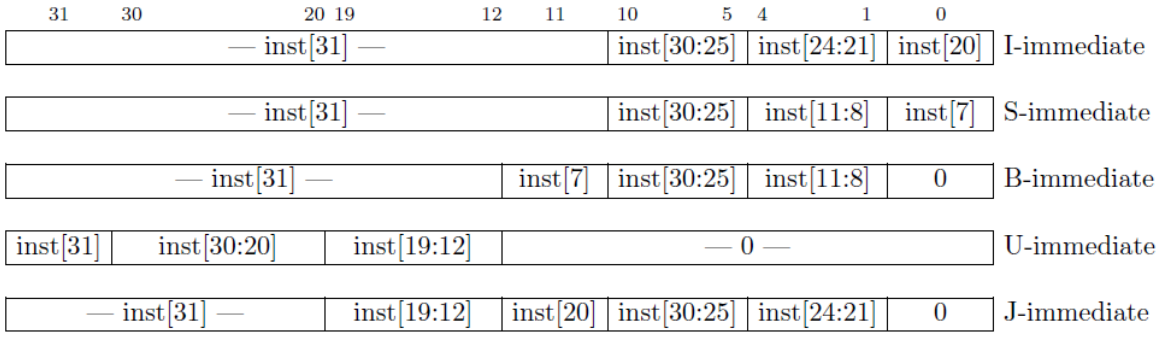

Figura B.2: Tipos de inmediatos RISC-V. Figura de [5].

B.2. Tipos de instrucciones base

Los conjuntos de instrucciones base definen instrucciones de los siguientes tipos:

– **Instrucciones aritméticas**: realizan una operación con 2 valores, estos pueden ser ambos del banco de registros (tipo R) o uno de ellos puede ser un valor inmediato (tipo I o U), o el valor del `PC` (tipo U). Después guardan en el banco de registros el resultado.

– **Instrucciones de acceso a memoria**: calculan una dirección utilizando 2 valores, uno del banco de registros y otro inmediato. En el contexto de estas instrucciones se les llama dirección base y desplazamiento respectivamente. Pueden leer el dato que se encuentre en la dirección calculada y guardarlo en el banco de registros (instrucción *load*, tipo I) o guardar un dato procedente del banco de registros en ella (instrucción *store*, tipo S).

– **Instrucciones de salto**: modifican el valor del `PC`, esto permite cambiar la instrucción que se va a ejecutar a continuación. Calculan el nuevo valor del `PC` utilizando su valor actual o el valor de un registro y un valor inmediato. Dependiendo de la instrucción estos saltos puede ser condicionales (tipo B) o no (tipo J o I), en caso de que lo sean se comparan 2 valores del banco de registros y se salta solo si se cumple la condición. Algunas de estas instrucciones también guardan en el banco de registros la dirección de la siguiente instrucción antes de que se modifique el `PC`, esto permite mas adelante poder restaurarla.

- **Instrucciones *trap***: Estas instrucciones se utilizan para implementar llamadas al sistema (Tipo I). Siempre generan una excepción cuando se ejecutan.

- **Instrucciones barrera**: Estas instrucciones se utilizan para gestionar el orden en los accesos a memoria (Tipo específico para estas instrucciones). La ISA define un modelo de consistencia de memoria débil para permitir técnicas agresivas de optimización en los accesos a memoria. Para poder controlar el impacto de estas técnicas se definen estas instrucciones. Se ha decidido no implementar ninguna de estas técnicas en el diseño implementado por lo que las instrucciones barrera no tienen ningún efecto.

Anexos C

Segmentación de la ruta de datos

La segmentación es un mecanismo de optimización que permite ejecutar varias instrucciones en paralelo. En una ruta de datos dividida en diferentes etapas consiste en tener una instrucción diferente en cada etapa. La Figura C.1 muestra como la segmentación reduce el número de ciclos por instrucción medio.

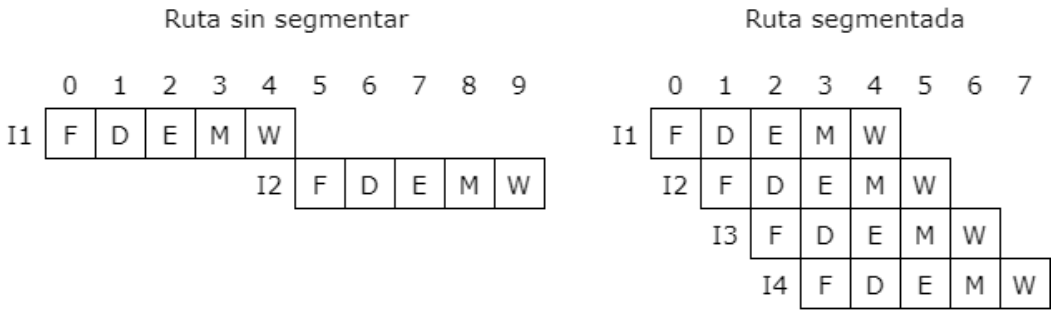

Figura C.1: Comparativa de ejecución de varias instrucciones en una ruta de datos sin segmentar y segmentada.

C.1. Dependencias entre instrucciones

Al segmentar la ruta de datos surgen diferentes problemas que se han tenido que solucionar. Uno de ellos es que las instrucciones tienen dependencias entre ellas, una instrucción puede escribir sus resultados en un registro que una instrucción posterior utilice como registro fuente. Si ambas instrucciones se están ejecutando en paralelo puede que el valor que lea la segunda instrucción no sea correcto.

Para poder solucionar este problema primero se tiene que poder detectar. Cuales son los operandos fuentes de una instrucción se calcula en la etapa *Decode*, por lo que se ha añadido a esta etapa la funcionalidad de detectar si alguno de estos registros fuente va a ser escrito por alguna instrucción en una etapa posterior.

Una vez se ha detectado la dependencia una posible solución es parar la ejecución de la instrucción hasta que sus operandos fuente estén preparados. Las dependencias entre

instrucciones son muy comunes, por lo que esto implicaría realizar muchas paradas lo que generaría una gran perdida de rendimiento. Para evitar esta perdida de rendimiento se ha optado por otra posible solución, esta consiste en adelantar los resultados de las operaciones previas, si ya han sido calculados, antes de que se escriban en el banco de registros.

Todos los posibles operandos fuente se utilizan en la etapa *Execute*, en esta etapa es donde también se generan todos los resultados, salvo en las instrucciones *load* que ocurre en *Memory*. Cualquier resultado calculado en una etapa *Execute* previa se puede adelantar a una instrucción en su etapa *Execute* sin tener que parar la ejecución.

En el caso de los *load* se tiene que parar la ejecución de la instrucción que se encuentra en *Decode* un ciclo. Esto se hace inyectando una instrucción que no hace nada en la etapa posterior en vez de avanzar la instrucción. La Figura C.2 muestra un ejemplo del comportamiento de la ruta de datos en esta situación.

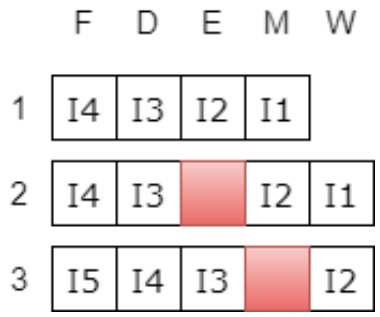

Figura C.2: En la etapa *Decode* se detecta una dependencia *load-use* entre las instrucciones I2 e I3, ocurre una parada de un ciclo hasta que se puede adelantar el operando.

Otro problema surge con las instrucciones de salto. Cuando se toma un salto se tienen que anular todas las instrucciones leídas de memoria posteriormente ya que el valor del `PC` que se ha utilizado para ello no ha tenido en cuenta el salto. Los saltos se pueden tomar una vez se ha calculado la dirección destino y se ha comprobado que la condición de los saltos condicionales se cumple, todo esto se realiza en la etapa *Execute*. Para reducir el número de instrucciones que se tienen que anular los saltos se van a tomar lo antes posible, es decir en la etapa *Execute*, teniendo así solo que borrar 2 instrucciones de las etapas anteriores. La Figura C.3 muestra un ejemplo de esta situación.

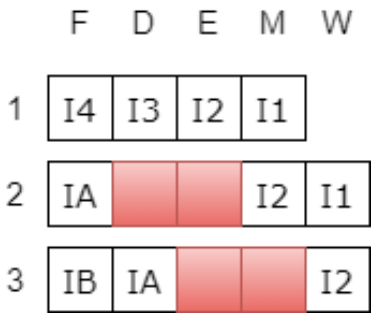

Figura C.3: La instrucción I2 en la etapa *Execution* va a realizar un salto a la instrucción IA, por lo que las instrucciones I3 e I4 han de ser borradas perdiendo así 2 ciclos de ejecución.

Anexos D

Registros de control RISC-V

`mstatus`: Este registro contiene información del estado del procesador. La figura D.1 muestra todos los campos y bits que lo forman.

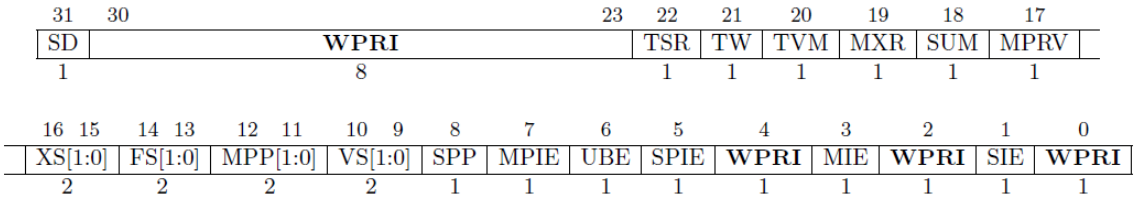

Figura D.1: Registro de estado `mstatus`. Figura de [6].

De este registro solo se van a utilizar 3 campos, el resto toman siempre el valor 0.

- `MIE` (*Machine Interrupt Enable*): este bit controla cuando las interrupciones esta activas. Si esta activo se debe realizar una *trap* cuando ocurra una.
- `MPIE` (*Machine Previous Interrupt Enable*): este bit se utiliza para almacenar el valor de `MIE` cuando ocurra una *trap* para así poder restaurarlo luego. Es necesario guardar el estado de las interrupciones por que siempre que ocurre una *trap* estas se desactivan de forma automática para que se pueda procesar la que ha ocurrido. Si no se hiciera esto el estado guardado de la primera podría sobre escribirse por el estado de la segunda y ya no se podría restaurar el estado inicial.
- `MPP` (*Machine Previous Privilege*): este campo se utiliza para almacenar el modo de ejecución cuando ocurra una *trap* para así poder restaurarlo luego.

`mie`: Este registro contiene la información de qué interrupciones están activas. Su tamaño es de 32 bits por lo que se pueden diferenciar entre 32 tipos de interrupciones diferentes. Solo se debe de realizar una *trap* cuando ocurra una interrupción y tenga su bit asociado de este registro activo.

`mtvec`: Este registro se utiliza para almacenar la dirección de la función *trap handler*. La figura D.2 muestra todos los campos y los bits que lo forman.

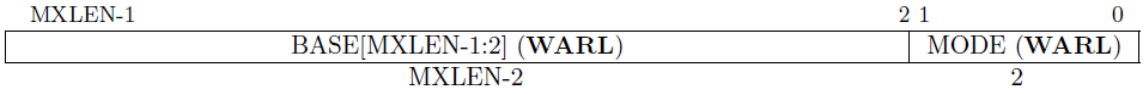

Figura D.2: Registro de estado `mtvec`. Figura de [6]

- `BASE`: este campo contiene los 30 bits que se necesitan para almacenar la dirección de la función.
- `MODE`: dependiendo del valor de este campo las *trap* pueden ser vectoriales o directas. Solo se va a dar soporte al modo directo, lo que implica que para todas las *traps* se salta al valor de `BASE` y que este campo siempre contiene el valor `00`.

`mepc`: Este registro se utiliza para almacenar el valor del `PC` de la instrucción en la cual ha ocurrido una *trap* para así poder restaurarlo luego.

`mcause`: Este registro se utiliza para almacenar un valor que permita identificar desde la función *trap handler* cual es el motivo por el que ha ocurrido la *trap*.

Anexos E

FPGAs y proceso de síntesis

Una FPGA esta compuesta de diferentes bloques lógicos genéricos programables y de una red de interconexión que también se puede programar para configurar como se conectan dichos bloques. La red de interconexión es muy extensa y compleja ya que tiene que dar soporte a muchas posibles configuraciones. Al ser tan compleja atravesarla añade retardos grandes, por lo que las frecuencias de reloj en las FPGA no es tan alta como en otros componentes lógicos.

El tipo de bloque que se utiliza principalmente para configurar la lógica de un circuito es el LUT (*Lookup Table*), este consiste en una memoria RAM que se programa con una tabla de verdad. Las FPGA también tienen bloques especializados que se utilizan para implementar funcionalidades mas específicas como pueden ser bloques de memoria RAM o celdas dedicadas a la entrada y salida. La Figura E.1 muestra un diagrama de una FPGA.

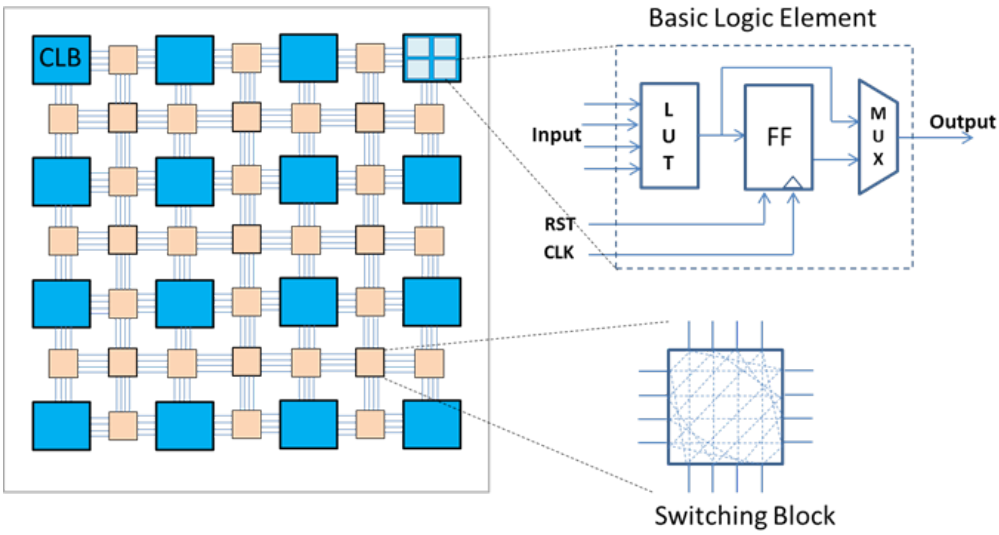

Figura E.1: Diagrama de componentes de una FPGA. Figura de `https://evergreen.loyola.edu/dhhoe/www/hoeresearchfpga.htm`

Dependiendo de como se ha especificado el circuito puede que las herramientas de

síntesis generen configuraciones imposibles de implementar. Por ejemplo, si una memoria RAM no esta bien especificada y la herramienta de síntesis detecta que tiene mas de 2 puertos de escritura puede que intente implementarla toda en registros, ocupando mas recursos de los que ofrece la FPGA. Es importante que la especificación este hecha teniendo en cuenta que bloques específicos tiene la plataforma objetivo y de forma que se amolde lo máximo posible a ellos si necesita utilizarlos, de esta forma se facilita el trabajo de las herramientas de síntesis.

Otra de las tareas que realizan las herramientas de síntesis es un análisis temporal de la configuración de la FPGA que generan. Como la red de interconexión entre los bloques lógicos añade mucho retardo se tiene que verificar que las rutas que la herramienta ha elegido para el diseño soporta la frecuencia de reloj. Es decir, que las señales de entrada a los registros están disponibles con suficiente antelación en un ciclo de reloj. Las herramientas son muy conservadoras en esta verificación intentando dejar bastante margen entre el momento en el que el valor esta disponible y el flanco de reloj.

Las herramientas de síntesis utilizan diferentes heurísticas a la hora de definir la distribución de bloques y su conexionado. Se pueden configurar para que intenten optimizar mas en concreto algun aspecto, como puede ser el área que ocupa el circuito, la temporización o el consumo de este.