RÍO NEGRO
UNIVERSIDAD NACIONAL

Implementación de softcore RISC-V en FPGA

Proyecto Final Integrador

Leandro Andrés Jalil Tomás Kromer Nicolás Luciano Bértolo

6 de junio de 2020
Universidad Nacional de Río Negro

Resumen

Este proyecto final integrador consiste en el desarrollo de un softcore que implementa el conjunto de instrucciones (ISA) RISC-V en una FPGA. La implementación del softcore está basada en una arquitectura de tipo *pipeline* y el sistema completo contiene soporte para dispositivos, interrupciones, memoria DRAM e interconexión a un bus AMBA AXI4. Dos de los periféricos implementados son una MAC Ethernet y una UART 16550 tomando como base un IP core de Xilinx para cada uno. La plataforma es capaz de correr Zephyr, un RTOS (Sistema operativo de tiempo real), para el cual se implementó un driver que maneje el MAC Ethernet, y a través de este, es capaz de conectarse a una red IPv4, solicitar una dirección IP y correr un servidor Telnet en ella.

El trabajo llevado a cabo en este proyecto incluye también la implementación de un sistema de integración continua automatizado, para la ejecución de los pasos necesarios para la compilación, síntesis, implementación y verificación del desarrollo.

Índice general

1. Introducción

Una FPGA es un dispositivo de hardware programable de alta velocidad que se usa para varias aplicaciones como codificación y decodificación de video, comunicaciones, y otros dominios de problema donde sea necesaria una alta velocidad de procesamiento con baja latencia. Una ventaja de un diseño implementado en una FPGA es que está compuesto por hardware programable, de esta forma se aprovecha el paralelismo y la velocidad que puede ofrecer un diseño sin una capa de software intermedio.

Otro uso de una FPGA es para acelerar una parte específica de un diseño, como un algoritmo paralelo, o un componente que tiene que cumplir con requerimientos de latencia muy estrictos. En estos casos se suele incorporar la FPGA como parte de un SoC con un procesador de propósito general. En este procesador se implementan las partes del diseño que no sea necesario implementar en FPGA.

Ademas, se utiliza para realizar prototipado de diferentes sistemas, los cuales luego pueden verse implementados en un ASIC en su etapa final. El prototipado cuenta con la ventaja de poder realizar simulaciones del sistema en hardware y en caso de encontrarse un error, corregirlo sin tener que pagar un alto costo monetario. En nuestro caso, se utilizó para el prototipado de un softcore. Un softcore es un procesador que puede ser implementado de forma completa, mediante una síntesis lógica. Esta síntesis lógica consiste en definir el comportamiento abstracto del circuito deseado, generalmente codificado en un RTL, mediante un lenguaje de descripción de hardware o por sus siglas en ingles HDL (Hardware description language: lenguaje de descripción de hardware), y luego implementarlo en componentes de la FPGA.

Realizar una microarquitectura bajo las especificaciones de un ISA (Set de instrucciones) permite ejecutar binarios destinados y diseñados para ese set de instrucciones. Nosotros elegimos desarrollar el softcore basado en las especificaciones de RISC-V por las simples razones de que es gratuito, nuevo y su popularidad está creciendo mundialmente.

Este softcore nos servirá de experiencia para diseños digitales en lenguaje HDL, y nos ayuda a reafirmar como también adquirir más conocimientos en diseño de sistemas digitales, siendo de mucha importancia debido al crecimiento digital global.

1.1. Objetivos

El objetivo principal de este proyecto es la creación de un softcore funcional, basado en el set de instrucciones RISC-V, en el cual se pueda correr un RTOS (Sistema operativo de tiempo real). El enfoque del proyecto es la implementación dentro de una FPGA, aprovechando las ventajas que permite hacer simulaciones, posibilidad de iteraciones para correcciones de errores, entre otras, sin dejar de lado las desventajas que esto incluye como limites de recursos, timing. Este objetivo está divido en pequeños fragmentos con el fin de lograr un análisis más detallado en cada una de las etapas.

Los objetivos son:

- Realizar un softcore con diseño pipeline.
- Correr los test del ISA.

- Implementar extensiones.
- Implementar sistema de automatización para verificación y validación.
- Implementar memoria caché.
- Implementar un bus de comunicaciones.
- Implementar un PLIC para comunicación con dispositivos externos.

La finalización de cada uno de estos objetivos concluye en la creación de un softcore que a grandes rasgos cumplirá con el objetivo central, el cual se mencionó anteriormente. Además, estos objetivos nos brindan una idea de las tareas que se necesitaran llevar a cabo durante el proyecto, que al finalizarlas nos darán el resultado esperado. En los próximos capítulos, daremos una breve introducción al marco teórico sobre el tema de este proyecto. También, hablaremos de la implementación realizada y finalmente se observarán los resultados obtenidos así como una conclusión de los mismos.

2. Arquitectura de computadoras

La arquitectura de computadoras detalla cómo un conjunto de estándares de HW (Hardware) y SW (Software) interactúan para formar un sistema de cómputo.

Un ejemplo sobre arquitectura de computadoras es la arquitectura von Neumann, que todavía es utilizada por la mayoría de las computadoras en la actualidad. El diseño consiste en una CPU (Central processing unit: Unidad central de procesamiento), la cual incluye la ALU (Arithmetic Logic Unit: Unidad Aritmética Lógica), una unidad de control, registros, memoria para datos e instrucciones. El diseño también incluye una interfaz de entrada una de salida y almacenamiento externo.

Hay tres categorías que abarcan a la arquitectura de computadoras.

- Diseño del sistema: Incluye todos los componentes del HW del sistema, incluidos los procesadores de datos adicionales a la CPU, como la GPU (Graphics processing unit: Unidad de procesamiento gráfico). También se incluye controladores de memoria, de DMA (Direct memory access: Acceso directo a memoria), rutas de datos, virtualización como también multiprocesamiento.
- ISA: Define el código máquina sobre el que lee y actua el procesador, así como el tamaño de palabra, los modos de dirección a memoria , los registros del procesador y el tipo de dato.
- Microarquitectura: Es la forma en que un determinado ISA se implementa en un procesador particular, definiendo el camino de datos, el procesamiento de datos y los elementos de almacenamiento, es decir, incluye las partes constituyentes del procesador y cómo estas se interconectan.

Para determinar la performance del CPU, se calcula cuánto tarda en ejecutarse un programa. Como existen infinitos programas para utilizar, se desarrollaron los Benchmark Los benchmarks más famosos son los benchmarks SPECint y SPECfp desarrollados por Standard Performance Evaluation Corporation y el benchmark ConsumerMark desarrollado por el Embedded Microprocessor Benchmark Consortium EEMBC.

$$Tiempo_{CPU} = \frac{N * CPI}{f}$$

N: *Corresponde al número de instrucciones ejecutadas realmente. La densidad del código del conjunto de instrucciones afectan fuertemente N.*

CPI: *Corresponde al número medio de ciclos de reloj por instrucción en un programa o fragmento.*

f: *Frecuencia del reloj en ciclos por segundo.*

Un error habitual es considerar que uno o dos de estos factores son determinantes de la performance, en realidad ésta depende del tipo de trabajo.

- Caso típico: frecuencia de reloj
- $MIPS = \frac{f}{(CPIx10^6)}$

	N	CPI	f
Programa	X		
Compilador	X	(X)	
ISA	X	X	
Organización		X	X
Tecnología			X

Cuadro 2.1.: Aspectos de la performance de la CPU

2.1. ISA

El ISA (Instruction Set Architecture), también conocido como un conjunto de instrucciones es una especificación que detalla las instrucciones que una unidad de procesamiento puede entender y ejecutar. Este termino describe los aspectos del procesador generalmente visibles para un programador, incluyendo tipos de datos nativos, instrucciones, registros, la arquitectura de memoria y las interrupciones.

Existen en la actualidad tres tipos:

- CISC (Complex Instruction Set Computer).
- SISC (Simple Instruction Set Computer).
- RISC (Reduced Instruction Set Computer).

La primera arquitectura era CISC. Los procesadores con este tipo de arquitectura se caracterizan por tener un set de instrucciones muy amplio que permite realizar operaciones complejas con tan solo una instrucción.

La segunda arquitectura mencionada es SISC. La misma esta orientada al procesamiento de tareas en paralelo. Permite que varios dispositivos de bajo costo se utilicen en conjunto para resolver un problema particular divido en partes disjuntas.

Y finalmente, la arquitectura conocida como RISC. La misma es un subjunto de la antes mencionada SISC. Esta se caracteriza por tener un set de instrucciones reducidos que posibiliten la segmentación y el paralelismo a la hora de ejecución de instrucciones.

Lo que se pretende en un conjunto de instrucciones es que se pueda realizar en un tiempo finito cualquier tarea ejecutable con una computadora, que permita alta velocidad de cálculo sin exigir una elevada complejidad en su UC (Unidad de Control) y ALU (Arithmetic Logic Unit: Unidad Aritmética Lógica) y sin consumir excesivos recursos de memoria. Otro punto a tener en cuenta es que contengan toda la información necesaria para ejecutarse y que no dependan de la ejecución de alguna otra instrucción.

Para el proyecto utilizamos el ISA RISC-V. Este es una arquitectura RISC. Esta tiene la ventaja de ser nueva, gratuita, abierta y de muy alto potencial para el futuro.

2.2. RISC-V

RISC-V es un conjunto de instrucciones de Hardware Libre basado en un diseño RISC. Es totalmente gratuito y abierto, lo que deja la libertad de que quien quiera pueda diseñar, fabricar y vender chips como también SW (Software) destinado para este set de instrucciones. Este ISA en su inicio fue desarrollado para uso interno de la Universidad de California Berkeley para investigación y cursos. Luego de un tiempo los diseñadores comenzaron a recibir quejas de por qué realizaban cambios en el ISA de algunas de sus practicas que se encontraban en internet. Al encontrase frente a esta situación de necesidad decidieron volverlo un estándar abierto.

Su diseño está pensando para implementaciones pequeñas, rápidas y de bajo consumo pero sin una sobre ingeniería excesiva que buscase una microarquitectura concreta. Los objetivos de RISC-V son:

- Debe funcionar bien con una amplia variedad de paquetes de software y lenguajes de programación populares.
- Debe poder implementarse en todo tipo de tecnologías:
  - ASIC (Application Specific Integrated Circuits: Circuitos integrados de propósito especifico)
  - FPGA (Field Programmable Gate Arrays: Arreglos de compuertas programables)
  - Chips personalizados e incluso tecnologías aún no inventadas.
- Debe ser eficiente para todo tipo de micro-arquitectura: microcódigo; distintos tipos de pipelines: en orden, o fuera de orden; emisión de instrucciones secuenciales simple o superescalar; etcétera.
- Debe permitir un alto grado de especialización para utilizarse como base de aceleradores personalizados, los cuales cobran mayor importancia ahora que la Ley de Moore comienza a perder fuerza.
- Debe ser estable, implicando que el ISA base no debe cambiar. Aun más importante, no debe descontinuarse, como ha ocurrido en el pasado con ISAs propietarios como: AMD Am29000; Digital Alpha; Digital VAX, Intel i860, Intel i960, Motorola 88000 y Zilog Z8000.

Lo que se busca generalmente en el diseño de arquitectura de computadoras es desarrollar ISAs incrementales, en donde, los nuevos procesadores no solo implementan las nuevas extensiones, si no que también todas las instrucciones de ISAs anteriores con el objetivo de mantener compatibilidad binaria. Esto provoca que se tenga un incremento sustancial en la cantidad y complejidad del ISA, sin dejar de lado que se van arrastrando errores de extensiones pasadas como también instrucciones obsoletas.

> « *Intel apostó su futuro a un procesador de alto desempeño, sin embargo, el diseño de dicho procesador llevaría años. Para contrarrestar a Zilog, Intel*

desarrolló un procesador temporal llamado 8086. Dicho procesador duraría muy poco tiempo en el mercado y no tendría sucesores, pero la historia no fue esa. El procesador de alto desempeño llegó tarde al mercado y era muy lento. De esta manera, el 8086 siguió en el mercado y evolucionó a un procesador de 32 bits y eventualmente a 64 bits. Los nombres fueron cambiando (80186, 80286, i386, i486, Pentium), pero, por cuestiones de compatibilidad, el set de instrucciones permaneció intacto. [1] »

RISC-V ademas de ser nuevo y gratuito, tiene la particularidad de que, a diferencia de casi todos los ISAs, es *modular*. El ISA base de RISC-V es llamado RV32I, el cual ejecuta un stack de software completo y se encuentra, como la comunidad de desarrolladores lo denomina, *congelado*, por lo cual nunca cambiará. El concepto de modularidad viene de extensiones opcionales estándar que el hardware puede incorporar de acuerdo a las necesidades de cada aplicación.

El estado en el que se encuentra el ISA base y sus extensiones pueden verse en la tabla 2.2.

Nombre	Descripción	Versión	Estado
	Base		
RV32I	Set de instrucciones de enteros, 32 bits	2.2	Congelada
RV32E	Set de instrucciones de enteros (embedded), 32 bits, 16 registros	1.9	Abierta
RV64I	Set de instrucciones de enteros, 64 bits	2.0	Congelada
RV128I	Set de instrucciones de enteros, 128 bits	1.7	Abierta
	Extensiones		
M	Extensión estandar para multiplicación y division de enteros	2.0	Congelada
A	Extensión estandar para instrucciones atómicas	2.0	Congelada
F	Extensión estandar para punto flotante de simple precisión	2.0	Congelada
D	Extensión estandar para punto flotante de doble presición	2.0	Congelado
G	Taquigrafía para la base y extensiones superiores	N/A	N/A
Q	Extensión estándar para punto flotante de cuádruple precisión	2.0	Congelado
L	Extensión estándar para coma flotante decimal	0.0	Abierta
C	Extensión estándar para instrucciones comprimidas	2.0	Congelado
B	Extensión estándar para manipulación de bits	0.90	Abierta
J	Extensión estándar para idiomas traducidos dinámicamente	0.0	Abierta
T	Extensión estándar para transacción de memoria	0.0	Abierta
P	Extensión estándar para instrucciones SIMD empaquetadas	0.1	Abierta
V	Extension estándar para operaciones vectoriales	0.7	Abierta
N	Extensión estándar para interrupciones a nivel de usuario	1.1	Abierta
H	Extensión estándar para hipervisor	0.4	Abierta

Cuadro 2.2.: ISA base y extensiones

Para poder abastecer los objetivos que se planteron en el proyecto, el objetivo más determinante fue el del sistema operativo de tiempo real, el cual nos determinó que el softcore sea del tipo RV32IM. Esto debido a que el set estándar es RV32I y para permitir una mejora en la performance se optó por agregar la extensión M. De esta manera, evitar realizar multiplicaciones y divisiones por software y realizarlas mediante hardware lo cual reduciría el tamaño de instrucciones a ejecutar y, por ende, el tiempo de ejecución.

2.2.1. Modos de operación

RISC-V admite 3 niveles de privilegios: modo de usuario, supervisor y máquina, siendo el modo máquina el utilizado para el diseño del softcore. Existe la posibilidad de agregar un modo más a la especificación de RISC-V pero hasta ahora no está definido.

El modo máquina, abreviado como modo M, es el modo más privilegiado en el que un Hart (Hardware thread: Hilo de ejecución en hardware) de RISC-V puede ser ejecutado. Cada Hart que corre en modo M tiene acceso completo a la memoria, I/O, y funcionalidades del sistema de bajo nivel necesarias para arrancar y configurar el sistema. Como tal, es el único modo de privilegio que implementan todos los procesadores RISC-V estándar. La característica más importante de este modo es la habilidad de interceptar y manejar *excepciones*.

Las *excepciones* pueden ser clasificadas como *excepciones* síncronas o interrupciones. Las *excepciones* síncronas son aquellas que ocurren como resultado de ejecución de instrucciones, como cuando se accede a direcciones inválidas de memoria o ejecutar una instrucción con un *opcode* invalido. Las *interrupciones* son eventos asíncronos al flujo de una instrucción, como por ejemplo hacer click en el mouse.

A pesar de que el modo M es suficiente para sistemas embebidos simples, solo es recomendable cuando todo el código es confiable, dado a que el modo M tiene acceso ilimitado a la plataforma de hardware. Para proteger al sistema del código no confiable, y para proteger procesos no confiables entre ellos, son necesarios los otros modos. La implementación del modo U (Usuario), que se agrega al modo M, incluye la funcionalidad llamada PMP (Phsysical Memory Protection: Protección física de memoria) la cual permite al modo M especificar a cuáles direcciones de memoria puede acceder el modo U.

La funcionalidad de PMP es dar protección de memoria a un costo relativamente bajo, pero tiene varios inconvenientes que limitan su uso en computadoras de propósito general. Dado que PMP solo soporta una cantidad fija de regiones de memoria, no escala a aplicaciones complejas. Y como estas regiones deben ser contiguas en memoria física, el sistema puede sufrir de fragmentación de memoria. Los procesadores más sofisticados de RISC-V tratan estos problemas de la misma manera que casi todas las arquitecturas de propósito general, utilizando memoria virtual basada en páginas.

El modo S (o modo supervisor) tiene más privilegios que el modo U, es decir, a diferencia del modo U este modo puede ejecutar instrucciones privilegiadas. Generalmente, en este modo se ejecutan procesos que pertenecen al sistema operativo, por esta razón, el modo supervisor es también conocido como modo Kernel.

2.3. Microarquitectura

Una microarquitectura, como lo mencionamos anteriormente, es la manera en que una arquitectura en particular implementa el conjunto de instrucciones que se ejecuta en un procesador. Si bien describe el interconexionado de los diferentes elementos como compuertas lógicas, registros y ALU también incluye el camino de datos (datapath) y la trayectoria de control (control path). Entre otras cosas también describe el número y tipo de memorias caché, si existe re nombramiento de registros, la unidad de predictor de saltos (si es que tiene), una unidad para ejecutar fuera de orden (si es que tiene) y el nombre de las etapas de segmentación si es que se utiliza este diseño.

Existen diferentes formas en que se pueden ejecutar las instrucciones. A estas formas las llamaremos diseños de microarquitectura. Los diferentes diseños que se podría

utilizar son Uniciclo: Unicycle, Multiciclo: Multi-cycle, Segmentación: Pipeline, Superescalar: Superscalar (No entraremos en detalle) o Fuera de orden: Out-of-order (No entraremos en detalle)

2.3.1. Uniciclo

Es un diseño que ejecuta una instrucción en un ciclo de clock (CPI = 1). La frecuencia del sistema se ajusta a la ruta mas crítica combinacionalmente como también a la instrucción que mas tarda. Los recursos de memoria, ALU o lógicas deben utilizarse solo una única vez por cada ciclo. Tiene la desventaja de que usa ineficientemente el clock y algunas de las unidades funcionales deben ser duplicadas ya que no se pueden compartir en un ciclo. La ventaja que tiene es su simplicidad y su fácil interpretación.

2.3.2. Multiciclo

Es un diseño que ejecuta una instrucción en uno o varios ciclo de clock (CPI $>$ 1), teniendo por lo general una cantidad de ciclos por instrucción variable, dependiendo de cual es la instrucción a ejecutar. De esta manera, los recursos están utilizados de forma mas equitativa permitiendo aumentar la frecuencia del clock, ajustándose a la etapa de instrucción más lenta. Por esto mismo, es que se debe balancear el trabajo hecho en cada etapa. La desventaja que tiene es que requiere registros de estados adicionales, multiplexores y maquinas de estados mas complejas.

2.3.3. Segmentación o Pipeline

Es un diseño que ejecuta varias instrucciones al mismo tiempo y ejecuta cada una de ellas en varios ciclos de clock (CPI $>$ 1). Este diseño también es conocido como Pipeline. En un diseño de pipeline hay 3 riesgos principales que se deben detectar:

- Riesgos estructurales: Cuando dos instrucciones requieren acceder a un mismo espacio de memoria de forma simultanea y la instrucción superior escriba en la misma, es necesario que la instrucción posterior tome el valor adecuado.
- Riesgos de control: Cuando se tienen acciones inesperadas por la unidad de control como un salto condicional o incondicional es necesario conservar los datos de registros/memoria que podrían afectar una instrucción posterior a la del salto.
- Riesgos de datos: Se pueden tener 3 riesgos diferentes. Uno de ellos es del tipo RAW (Read After Write) en cual consiste en leer después de escribir. Otro es del tipo WAW (Write after Write) en cual consiste en escribir después de escribir y el ultimo es del tipo WAR (Write After Read) en cual consiste en escribir después de leer. Esto quiere decir que estos riesgos ocurren porque una instrucción depende del resultado de una instrucción previa en el pipeline. Todas las operaciones como add(suma) o sub(resta) no escriben el resultado hasta llegar a la ultima etapa.

2.3.4. Fuera de orden

Es un diseño que tiene la particularidad de que ejecuta las instrucciones como su nombre lo menciona, fuera de orden. La idea consiste en permitir al procesador evitar ciertos tipos de burbuja que suceden cuando la información necesaria para realizar una operación no esta disponible.

2.3.5. Superescalar

Es un diseño que es capaz de ejecutar mas de una instrucción por ciclo de reloj. La microarquitectura superescalar utiliza el paralelismo de instrucciones además del parelelismo de flujo, este ultimo gracias a las estructura en pipeline.

2.4. Eleccion de Arquitectura

Para el diseño de nuestro procesador optamos usar el diseño de pipeline por las siguientes razones:

- Es un diseño que utiliza los recursos de una forma mas equitativa, por lo cual nos permitía hacer ajustes en secciones criticas y observar que mejora tenemos en el rendimiento.
- Al ser un diseño bastante modular nos permitía dividir las tareas de forma equitativa ya que eramos 3 personas trabajando en el mismo softcore. Esto también permite un trabajo mas trazable y fácil de seguir.

2.5. Memorias

Una memoria puede ser clasificada por diferentes criterios, como por ejemplo:

- Soporte físico: Puede ser semiconductores, magnético, óptico o magneto-óptico.
- Alterabilidad: Puede ser RAM la cual sirve para leer como para escribir o ROM la cual solo sirve para leer.
- Volatilidad: Existen memorias que solo mantienen información cuando están energizadas y otras que la mantienen cuando no hay presencia de la misma.
- Duración de la Información: Hay dos tecnologías, uno de ellas es estática en la cual se mantiene inalterable mientras están polarizadas y la otra es dinámica en la cual el contenido sólo dura un corto período de tiempo, por lo que es necesario refrescarlas periódicamente.
- Proceso de lectura: Hay procesos que son destructivos y otros que no. Cuando el proceso es destructivo es necesario reescribirla después de una lectura.
- Ubicación en el CPU: Puede estar internamente al CPU como pueden ser los registros, memoria Cache (L1, L2 ,L3), o memoria principal. También puede ser externa como por ejemplo discos, cintas, etc.

- Parámetros de velocidad: Los parámetros que pueden diferir son el tiempo de acceso, tiempo de ciclo y ancho de banda (Frecuencia de acceso).

Las computadoras no utilizan una única memoria ya que no es factible tener una lo suficientemente rápida y suficientemente grande. Por ello mismo es que existe una jerarquía de memoria que se basa en los principios proximidad temporal y proximidad espacial. Esta jerarquía de memoria lo que busca es conseguir un buen rendimiento a un costo reducido.

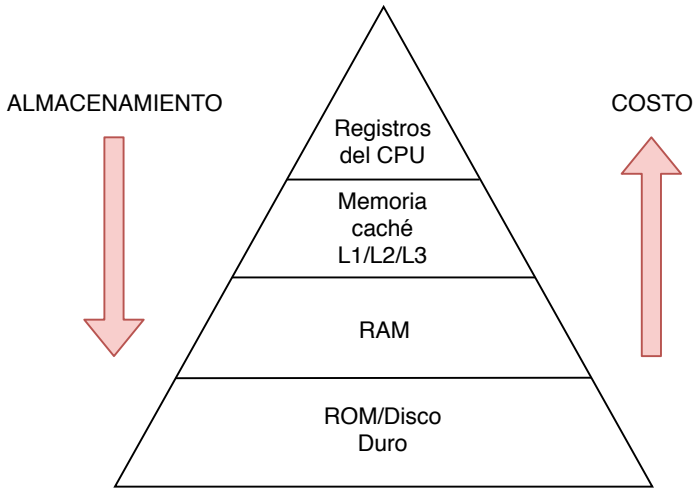

Figura 2.1.: Jerarquia de memoria

2.5.1. Registros del CPU

Los registros del CPU son de muy alta velocidad y se encuentran integrados en el microprocesador. Estos permiten guardar información temporalmente y son usados mayoritariamente en operaciones matemáticas. Su unidad de dimensión es en bits y son implementados generalmente en un banco de registros. Los registros son generalmente indexados como operandos de una instrucción como está definido en el ISA generalmente porque en un microprocesador también existen registros para fines específicos (Ej. Contador).

El ISA de RISC-V define los registros para un conjunto de extensiones, unos para extensiones de punto flotante y otros de estado que son para el conjunto de instrucciones privilegiadas. Los registros para el conjunto de instrucciones sin privilegios pueden observarse en la siguiente tabla.

Registro	Nombre en ABI	Función que lo guarda	Descripción
x0	zero	—	Siempre cero
x1	ra	Llamadora	Dirección de retorno
x2	sp	Llamada	Puntero a la *stack*
x3	gp	—	Puntero global
x4	tp	—	Puntero de *thread*
x5-7	t0-2	Llamadora	Temporarios
x8	s0/fp	Llamada	Registro guardado / Puntero al *frame*
x9	s1	Llamada	Registro guardado
x10-11	a0-1	Llamadora	Argumentos / Valores retornados
x12-17	a2-7	Llamadora	Argumentos de función
x18-27	s2-11	Llamada	Registros guardados
x28-31	t3-t6	Llamadora	Temporarios
f0-7	ft0-7	Llamadora	Temporarios de PF
f8-9	fs0-1	Llamada	Registros de PF guardados
f10-11	fa0-1	Llamadora	Argumentos / Valores retornados de PF
f12-17	fa2-7	Llamadora	Argumentos de PF
f18-27	fs2-11	Llamada	Registros de PF guardados
f28-31	ft8-11	Llamadora	Temporarios de PF

Cuadro 2.3.: Registros del ISA RISC-V

El conjunto de registros f0-f31 pertenece a la extensión de punto flotante con un ancho de precisión mayor. Estos registros no son utilizados en nuestro procesador ya que no tenemos este tipo de extensión.

Los registros para el conjunto de instrucciones privilegiadas que utilizaremos serán los utilizados en el modo M, el cual es el implementado en nuestro sistema. A continuación se observan los mismos en la tabla 2.4

Registro	Nombre del Registro	Privilegio	Descripción
	Machine Information Registers		
xF11	mvendorid	MRO	ID del Vendedor
xF12	marchid	MRO	ID de la Arquitectura
xF13	mimpid	MRO	ID de la Implementación
xF14	mhartid	MRO	ID del Hardware Thread
	Machine Trap Setup		
x300	mstatus	MRW	Estado del core
x301	misa	MRW	ISA y extensiones implementadas
x302	medeleg	MRW	Registro de delegación de excepciones
x303	mideleg	MRW	Registro de delegación de interrupciones
x304	mie	MRW	Registro de habilitación de interrupciones
x305	mtvec	MRW	Address base del trap-hadler
x306	mcounteren	MRW	Enable de contador
	Machine Trap Handling		
x340	mscratch	MRW	Registro de scratch del manejo de interrupciones
x341	mepc	MRW	Registro del PC de excepciones
x342	mcause	MRW	Registro de guardado de causa de excepción
x343	mtval	MRW	Guarda el bad address o instrucción que fallo
x344	mip	MRW	Registro de interrupciones pendientes
	Machine Memory Protection		
x3A0-x3A3	pmpcfg0-3	MRW	Configuración de PMP
x3B0-3BF	pmpaddr0-15	MRW	Registro del address de PMP

Cuadro 2.4.: Registros del CSR del ISA RISC-V

MRO: Machine Read Only.

MRW: Machine Read Write.

2.5.2. Memoria caché

La memoria caché es un componente que almacena datos que fueron y posiblemente serán solicitados en un futuro próximo con el fin de que al momento de solicitarlos se pueda atender con mayor rapidez. Al momento de hacer una petición se puede tener un acierto o un fallo, y a medida que mas grande sea la cache la cantidad de aciertos va a aumentar, por lo tanto el sistema funcionará más rápido. De todos modos hay que evaluar cuál es el tamaño más optimo para utilizar ya que el costo comienza a elevarse rápidamente en términos de los recursos de memoria. Para poder aumentar la cantidad de aciertos una alternativa puede ser con predicciones (*prefetching*) para

adivinar de dónde vendrán las lecturas futuras. Los datos en caché se almacenan en distintos niveles según la frecuencia de uso que tengan. La caché de nivel 1 (L1) se encuentra en el núcleo del microprocesador y se divide en dos subniveles, uno de ellos es para la cache de datos y el otro para la cache de instrucciones. El siguiente nivel de cache es el 2 (L2) en la cual se almacenan los datos de uso frecuente, siendo la misma de mayor tamaño que la caché L1 pero más lenta. Como ultimo nivel está la caché de nivel 3 (L3) la cual es aun más lenta y más grande que la caché L2 pero más chica y más rápida que la RAM.

Las caches L2 y L3 pueden ser inclusivas, esto quiere decir que los datos solicitados se quedan en la memoria caché de procedencia manteniéndose una copia en dos o más niveles. También pueden ser cachés exclusivas, esto quiere decir que los datos solicitados se eliminan de la memoria caché de procedencia una vez transferido al nuevo nivel.

En la memoria caché existen diversas formas de diseños, como por ejemplo:

- Ubicación: Política de decidir en qué lugar se colocara el bloque de memoria que ingresa a la caché.
- Extracción: Política que determina cuándo y qué bloque de memoria principal hay que llevar a memoria de caché.
- Reemplazo: Determina qué bloque de memoria caché se debe sacar cuando no existe espacio disponible para que un nuevo bloque entre.
- Actualización: Determina el instante en que se actualiza la información en memoria principal cuando se hace un cambio en la memoria cache.

2.5.3. Memoria de acceso aleatorio

La memoria de acceso aleatorio (random-access memory) se utiliza como memoria de trabajo para el sistema operativo, los programas y la mayoría del software. Se denominan de acceso aleatorio porque se puede leer o escribir en una posición de memoria con un tiempo de espera igual para cualquier posición, no siendo necesario seguir un orden para acceder a la información de la manera más rápida posible. Dentro de esta categoría las dos mas básicas son la DRAM (Dynamic Random Access Memory) y la SRAM (Static Random Access Memory) diferenciándose por el tipo de tecnología que utilizan. En la memoria dinámica necesita actualizarse miles de veces por segundo mientras que la estática no lo necesita, esto quiere decir que la memoria estática es mucho mas rápida, utiliza menos consumo y es más cara.

2.5.4. Memoria de solo lectura

Los datos almacenados en la ROM no se pueden modificar, o al menos no de manera rápida o fácil. Las ROM más modernas, como EPROM y Flash EEPROM, se pueden borrar y volver a programar varias veces, aun siendo descritos como memoria ROM. La memoria FLASH derivada de la memoria EEPROM permite la lectura y escritura de múltiples posiciones de memoria en la misma operación utilizando impulsos eléctricos permite velocidades de funcionamiento muy superiores frente a la tecnología EEPROM que solo permitía actuar sobre una única celda de memoria en cada

operación de programación. La memoria EEPROM fue el sustituto de las memorias EPROM con la diferencia primordial de que para borrar el contenido de la misma no era necesaria exponerlas a luz ultravioleta como tampoco era necesaria la ventana y el silicio permitiendo que el costo del encapsulado era menor.

2.6. Eleccion de Jerarquia de Memoria

Para nuestro procesador utilizamos los registros del ISA que mencionamos anteriormente, dos memorias caches de nivel 1 siendo una para instrucciones y la otra de ellas para los datos. También utilizamos una memoria DRAM que nos brinda la placa de desarrollo que utilizamos.

2.7. Buses

Todos los procesadores poseen un bus principal o de sistema por el cual se envían y reciben todos los datos, instrucciones y direcciones. Cuando hablamos de buses para microcontroladores podríamos mencionar al bus AMBA (Advanced Microcontroller Bus Architecture: Arquitectura avanzada de bus para microcontroladores) o al bus Wishbone, los cuales permiten mantener un estándar y que diversos dispositivos o procesadores puedan comunicarse entre sí hablando el mismo idioma.

2.7.1. Wishbone

El bus Wishbone es un bus de código abierto pensado como un bus lógico en donde no se especifica su información eléctrica ni su topología. Su especificación está escrita en términos de señales, ciclos de clock, y niveles lógicos (altos o bajos). Wishbone está definido para tener buses de 8, 16, 32 o 64 bits. Todas sus señales son sincrónicas pero algunas respuestas de *slave* deben ser resueltas combinacionalmente para mejorar su rendimiento. La transmisión es paralelizada.

2.7.2. AMBA

El bus AMBA es un bus que se utiliza on-chip generalmente para procesadores de ARM. También hay otros procesadores que lo utilizan como por ejemplo el procesador LEON2 y LEON3 que lo utilizan para interconectar el procesador con los periféricos. AMBA no especifica el nivel de tensión ni los tiempos. Existen diversos tipos de buses.

AMBA especifica 4 tipos de buses.

- AXI (Advanced eXtensible Interface: Bus de interconexión entre varios masters y varios slaves)
- AHB (Advanced High-performance Bus)
- ASB (Advanced System Bus)
- APB (Advanced Peripheral Bus)

2.8. Eleccion de Bus

Nosotros optamos por utilizar el bus AMBA4 bajo el protocolo de AXI4-Lite. Usamos AMBA AXI4 porque es el que está adoptado por Xilinx y es el utilizado para poder comunicarse con los periféricos de la placa de desarrollo que se usó.

3. Organización del proyecto

Para una mejor administración del proyecto, éste fue dividido en pequeñas partes para lograr un mayor nivel de abstracción. Esto dio lugar a desarrollar un plan de trabajo que nos permitió mejorar la división de las tareas y establecer tiempos estimativos para cada una. Al momento de finalizar cada una de las etapas con sus respectivos test de verificación, solo restaría realizar la integración de sus partes para luego somenterlo a los test más generales.

El plan de trabajo fue creado para generar metas, las cuales en su conjunto llevaran a la realización del proyecto. A su vez, permitió una división de tareas entre los integrantes del mismo. También permitió generar interaciones en el proyecto permitiendo agregar funcionalidades al diseño.

Observamos que para poder lograr cumplir dichas metas, se necesitaba agregar herramientas que permitieran una rápida iteración entre las mejoras y correcciones de errores del sistema. Se buscó aprovechar el conjunto de tests ya implementados para este set de instrucciones. También era deseable que el conjunto de tests estuviera automatizado, ya que es frecuente olvidarse de correrlos, o bien, equivocarse. Por esta razón, se pensó en un sistema de CI (Continuous Integration: Integración Continua) en el cual se pudieran ejecutar los test de RISC-V así como los tests diseñados en la herramienta de desarrollo Vivado. De esta manera, podremos saber rápidamente si los cambios aplicados pasan los test de RISC-V y/o el diseño es sintetizable.

La integración continua consiste en llevar a cabo testeos automáticos de un proyecto después de cada cambio, permitiendo detectar errores lo antes posible. Lo habitual es que los tests corran automaticamente después de un cambio subido al repositorio. De esta forma, pudimos obtener un reporte de qué tests son superados por una versión particular y cuáles no.

3.1. Control de versiones

El sistema de control de versiones que elegimos para este proyecto es Git. Es un sistema de control de versiones distribuido con buen soporte para distintas ramas. Esto fue importante ya que facilitaba el trabajo en diferentes partes del sistema sin entorpecernos entre nosotros.

Usamos el proveedor de hosting GitLab porque provee repositorios privados y un sistema de CI que juzgamos adecuado para nuestras necesidades.

3.2. Testing

Es importante garantizar que el core implementa la especificación correctamente. Para este propósito creamos un sistema de testing automático que verifica el funcionamiento adecuado del core ante cada cambio (commit) que hicieramos.

Para realizar las simulaciones del hardware, se utilizaron las herramientas de Verilator y Vivado. Verilator toma los archivos HDL y crea un modelo en lenguaje C++ para luego simularlo, en esta herramienta se realizaron los test generales del ISA. En Vivado, se realizaron los test específicos de partes especificas del sistema diseñado.

Por ultimo, para la prueba en hardware del diseño se utilizo una placa conectada a un servidor, en el cual ante cualquier cambio se iniciliazaba con el camino de desarrollo común de este tipo de proyectos (síntesis, implementación, generación de bitstream y programación).

3.2.1. Tests de componentes

Para los módulos testeables en aislación se implementaron tests en SystemVerilog que instancian el módulo en cuestión, simulan entradas y verifican la salida contra datos correctos.

Los módulos que se testean de esa forma son:

- Unidad de *shift* de ALU (Arithmetic Logic Unit: Unidad Aritmética Lógica)
- Caché de datos
- Caché de instrucciones
- Dispositivo de GPIO
- Puente entre memoria e IO
- Dispositivo de timer
- Platform Layer Interrupt Controller
- Etapa de *ejecución*
- Etapa de generación de *Program Counter*
- Módulo de UART

3.2.2. Tests de RISC-V

El proyecto RISC-V creó un conjunto básico de tests que prueban el funcionamiento de las instrucciones del set base y de algunas extensiones como comprimidas, multiplicación, entre otras. Estos tests están implementados como pequeños programas con una capa de abstracción de la arquitectura sobre la que se corren.

Este conjunto de tests fue de gran utilidad durante el desarrollo del proyecto ya que permitió depurar el funcionamiento de las instrucciones aisladas entre sí.

3.2.3. Modelo de Verilator

El software verilator fue de mucha utilidad para depurar el trabajo. Con verilator es posible producir un modelo que se puede compilar como un programa C++. Este modelo puede producir un archivo de trazas `logs.vcd` que contiene los valores de todas las señales a lo largo de la simulación.

Integración de stream de salida

Es posible integrar el módulo de la UART con el stream `stdout` a través de las funciones de PLI (Programming Language Interface):

```
always @(posedge clk or posedge wb_rst_i)
begin
  if (fifo_write)
    begin
      \$write("%c", wb_dat_i);
  end
end
```

Esta porción de código es parte de la UART que compone el modelo de verilator. Se encarga de imprimir por pantalla cada caracter que recibe.

Integración de stream de entrada

Para que la UART, que compone el modelo de verilator, reciba cada caracter que el usuario ingresa por el teclado es necesario convertir ese caracter en señales eléctricas que sean recibidas por el pin de entrada de la UART. Para que sea más sencillo de programar desde el lado de C++ se optó por colocar un modelo simplificado de una UART conectada a la UART completa. Este modelo se controló a través de señales controladas por el código C++.

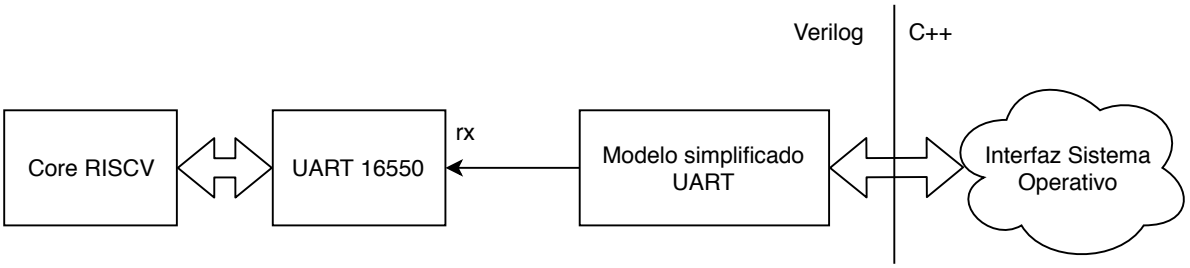

Figura 3.1.: Diagrama de la integración con `stdin`

3.2.4. Sistema de Integración Continua

Armamos un setup de pruebas automáticas que se corre cada vez que se sube un commit al repositorio de GitLab.

En la figura 3.2 se puede observar un ejemplo de una corrida, en la cual se pude apreciar que la primer etapa de testeo pasó exitosamente y la segunda etapa fallo.

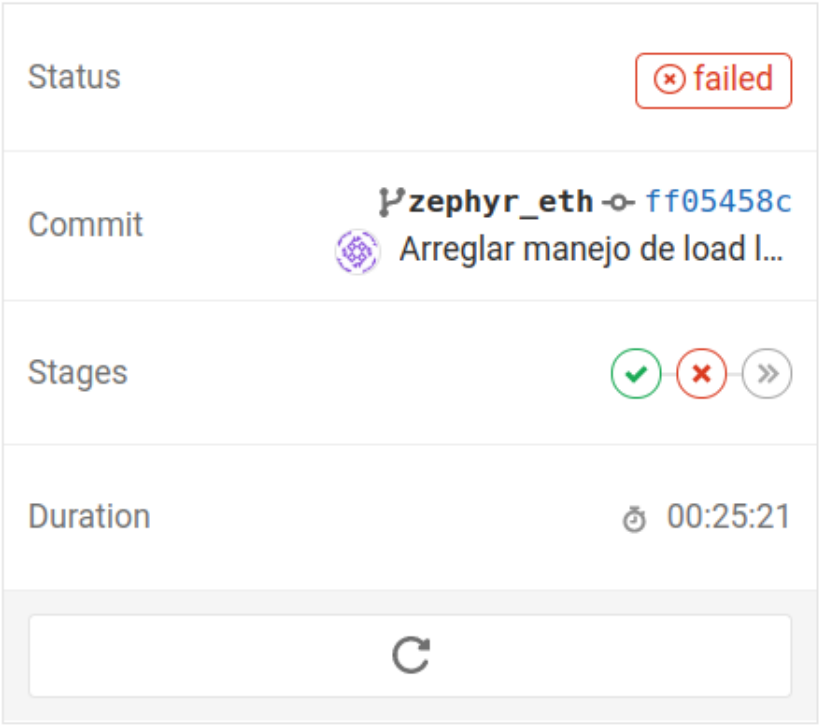

Figura 3.2.: Ejemplo de CI

El servicio de GitLab provee facilidades de integración continua con integración fácil de usar. Los pasos a correr se definen en un archivo `.gitlab-ci.yml` ubicado en la raiz del repositorio. En este archivo se definen las sucesivas etapas y los trabajos asociados a cada una. Los trabajos de una misma etapa pueden correr en paralelo. Cuando todos los trabajos de una etapa se completan exitosamente se ejecuta la siguiente.

Es posible definir varios entornos donde se corren los comandos de cada trabajo. En este trabajo elegimos que se corran en contenedores de Docker. Docker es una tecnología que permite correr servicios aislados entre sí sobre un kernel linux. Es similar a una máquina virtual, pero también permite elegir qué compartir con alta granularidad. En este caso nos interesa que los trabajos estén aislados entre sí y que las modificaciones al sistema de archivos sean revertidas cuando termine el trabajo. Para aprovechar el paralelismo que se describe al definir los trabajos en etapas con dependencias es necesario que cada trabajo corra en un sistema de archivos separado.

Para aprovechar las funcionalidades de Docker se creó una imagen Docker con los componentes de software necesarios para probar el trabajo.

- Toolchain de RISC-V.
- Vivado 2018.3
- Verilator
- Python 3.6 con pytest.

Se definieron dos tipos de ejecutores de trabajos de CI. Uno de ellos corre todos los trabajos que no estén etiquetados como *hardware*. Este ejecutor puede correr varias instancias en paralelo. El otro ejecutor corre solo los trabajos etiquetados como *hardware* y solamente puede haber un trabajo ejecutandose con este ejecutor. De esta forma se aprovechan los recursos del hardware para paralelizar la síntesis y simulaciones del trabajo, pero el acceso a la placa de desarrollo está serializado. El traspaso de información entre etapas, en este caso el resultado de la síntesis e implementación (el bitstream), se hace a través del mecanismo de *artifacts* que provee GitLab.

Las etapas en las que se dividen los trabajos son:

1. Simulaciones y producción de bitstream.

   Tests de Verilator y Vivado: Se corren los tests de los módulos de vivado como también los test de verilator verificando que el resultado sea el esperado.

   Producción de birstream: Se corre la sintesis, implementación para juego generar el bitstream si es que no se detectaron errores.

2. Ejecución en una placa Zybo: Se corren los tests de RISC-V en la placa y se verifica que el resultado sea correcto.
3. Chequeo de formato de código: Se verifica que el código use el formato correcto.

4. Memoria Técnica

4.1. Plataforma de desarrollo elegida: ZYBO, Xilinx

Para este proyecto, se opto por la placa de desarrollo Zybo Zynq-7000 ARM/FPGA. La misma cuenta con las características necesarias para cumplir con los requerimientos pedidos. A continuación una breve descripción de las especificaciones de la placa en cuestión:

- FPGA Xilinx Zynq-7000 (XC7Z010-1CLG400C).
- 28000 celdas lógicas.
- 240KB Block RAM.
- 80 DSP slices.
- Procesador dual-core Cortex^{TM}-A9 650 MHz.
- GPIO: 6 pushbuttons, 4 slide switches, 5 LEDs.
- 128 Mb Serial Flash c/ interfaz QSPI.
- Controlador de memoria DDR3 con 8 canales DMA.
- 512MB x32 DDR3 c/ 1050Mbps de ancho de banda.
- 6 puertos Pmod (1 dedicado al procesador,1 dual analógico/digital)

En cuanto a la herramienta de desarrollo, se utilizo Vivado de Xilinx. Debido a que la placa en cuestión utiliza una FPGA del fabricante Xilinx. Además, ésta es una herramienta muy completa y presenta un entorno de desarrollo amigable. Dentro de sus posibilidades permite crear un diseño de manera fluida y permite automatizar ciertas etapas del proyecto mediante scripts en TCL (Tool Command Language).

4.2. Descripción de la Arquitectura implementada

Para este proyecto se eligió la arquitectura *pipeline*. En nuestro caso esta consta de siete etapas enumeradas a continuación:

- PC Generator.
- Instruction Fetch.
- Instruction Decode.
- Issue.
- Execute.
- Memory.
- Commit.

4.2.1. Descripción de Etapas

Cada una de estas etapas cumple una función particular en el sistema. También interactúan entre ellas para lograr el correcto funcionamiento del sistema. A continuación se encuentra brevemente explicado el funcionamiento de cada etapa. En la figura 4.2 se observa un diagrama general de las etapas.

Para comprender mejor el funcionamiento de las etapas son necesarias algunas definiciones:

Burbuja: Consiste en enviar una operación de *nop* a través del pipeline para hacer tiempo hasta que un proceso se cumpla.

Hit: Consiste en una solicitud a memoria cache de un dato que es encontrado dentro de la misma.

PC Generator

La primer etapa de pipeline es la etapa de generación de *Program Counter* (PCGEN). Es en esta etapa se elige el próximo PC (Program Counter) en función de la información proporcionada por la etapa de *commit*. Este valor de PC es enviado hacia la caché de instrucciones, en donde se buscará la instrucción y luego será enviado a la próxima etapa.

Instrucción Fetch

La segunda etapa llamada etapa de *instruction fetch* (IF), es la encargada de recibir la instrucción desde la caché. Ésta tomara la instrucción, en caso de que la caché haga un *hit*, y la pasará a la siguiente etapa. En el caso de no hacer *hit*, esta etapa enviara una burbuja a la siguiente etapa hasta que reciba la instrucción. El propósito de esta etapa es procesar las instrucciones comprimidas, aunque no están implementadas en esta versión del pipeline.

Instrucción Decode

La tercer etapa nombrada etapa de *instruction decode* (ID) es la encargada de recibir la instrucción y decodificarla. En la misma, se encuentra la unidad de *Instruction Decode* y la *Control Unit* del Pipeline. Esto se refiere a determinar qué tipo de instrucción es, qué operación llevara a cabo, qué datos necesitara, dónde será guardado el resultado (de haber uno), generar el dato inmediato, entre otras cosas. Todos estos datos serán enviados hacia la próxima etapa.

La unidad de *Instruction Decode* toma la instrucción recibida de la etapa anterior y decodifica de manera tal de que la unidad de control pueda utilizar esa información. Luego, la unidad de control recibe estos datos y configura las señales de control que serán utilizadas en las siguientes etapas del pipeline (tales como registro a usar, registro en donde guardar, operación de ALU a utilizar, etc).

Issue

La cuarta etapa llamada etapa de *issue* (ISSUE), es la encargada de recibir la decodificación de la instrucción. En ella, se leen los datos de los registros para ser utilizados en la próxima etapa. En la misma, se encuentran los GPR (General Purpose Register) y los CSR (Control Status Register).

De los GPR, se seleccionan los datos que serán entregados a la siguiente etapa para la operación. También existe la posibilidad de usar en conjunto los CSR con los GPR utilizando una operación particular del ISA, como por ejemplo una operación en la que se intercambian los datos.

A su vez, al estar en esta etapa tanto los CSR como GPR, las señales de control de escritura y de datos de la etapa final del pipeline (etapa de *commit*) deben ser enviadas a esta etapa para realizar la escritura de los datos.

Execute

La quinta etapa, llamada etapa de *ejecución* (EX), toma los datos proporcionados por la etapa anterior y realiza la operación deseada. Estas operaciones pueden ser lógicas (and, or, xor, corrimientos), aritméticas (suma, resta, multiplicación, división) o comparativas (igual, mayor o igual, menor). En ella, se encuentra la ALU (Arithmetic Logic Unit: Unidad Aritmética Lógica), la cual esta subdividida en otras pequeñas unidades encargadas de un conjunto particular de las operaciones mencionadas. Estas unidades son:

- Unidad de Shift: Se encarga de los corrimientos lógicos y aritmético.
- Unidad de Operaciones Rápidas: En esta unidad están las operaciones que se requiere que se realicen instantáneamente (en nuestra implementación suma, and, or y xor).
- Unidad de Comparación: En esta unidad se encuentran todas las comparaciones entre valores.
- Unidad de Multiplicación: En esta unidad se realizan todas las operaciones de multiplicación y se orientó la unidad a ser realizada en DSP (Digital Signal Processor).
- Unidad de División: En la misma, se realizan las operaciones de división y resto de división (módulo).

Este diseño fue pensado para evitar dependencias entre unidades, realizando la activación de una unidad a la vez, exceptuando un único caso. Este caso es una instrucción de salto o branch. Cuando se realiza una instrucción de salto se debe realizar una comparación para determinar si el programa en ejecución deberá realizar un salto o no, ademas se tiene que calcular el PC al cual deberá saltar el procesador en caso de tomar dicho salto. Para esto, se utilizan la unidad de operaciones rápidas y la de comparación. La primera realizara el calculo del próximo PC, mientras que la segunda realiza la comparación que decidirá si se tomara el salto. De esta manera, se evita esperar la comparación para decidir si se deberá calcular el PC al que debe saltar. En la figura 4.1 hay un diagrama de las partes de la ALU.

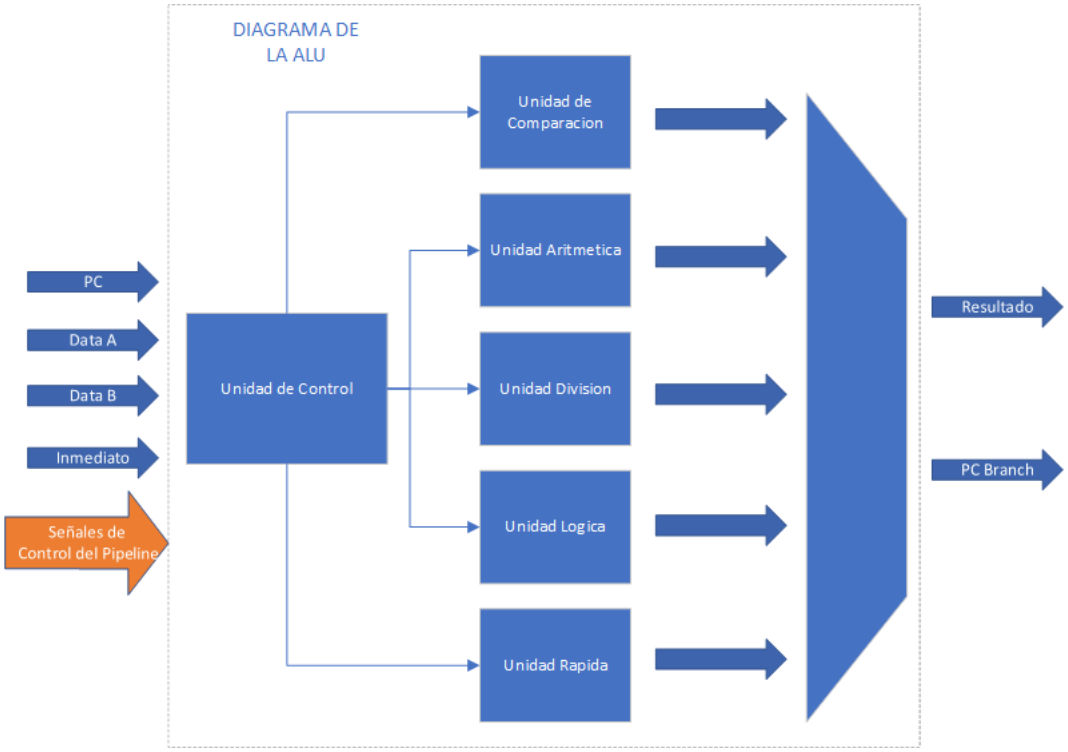

Figura 4.1.: Diagrama de la ALU.

Memory

La sexta etapa llamada etapa de *Memoria*, es en la cual se toman los pedidos de instrucciones de lectura o escritura de memoria o dispositivos y se enviaran al puente entre memoria e IO. Estos pedidos una vez finalizados devolverán el dato y un aviso de finalización a la siguiente etapa (etapa de *commit*).

En el puente entre memoria e IO se recibirán los datos enviados desde la etapa de Memory y se direccionará a la memoria o dispositivos según corresponda. Más adelante, en la sección 4.5, se explica en profundidad la conexión de la unidad mencionada tanto con la interfaz de dispositivos como con la memoria cache.

Commit

La última etapa, llamada etapa de *commit*, en la misma se indica de donde vendrá el resultado, a dónde será guardado, cual sera la fuente del próximo PC y otorgara el próximo pc en caso de ser un salto. En este punto, el valor obtenido de una instrucción realizada será guardado y no se podrá volver atrás.

También se encuentra la lógica de atención de las excepciones y interrupciones del sistema. En la misma se analiza la causa, se determinan los pasos a seguir y se guarda toda la información relevante de la excepción o interrupción según corresponda.

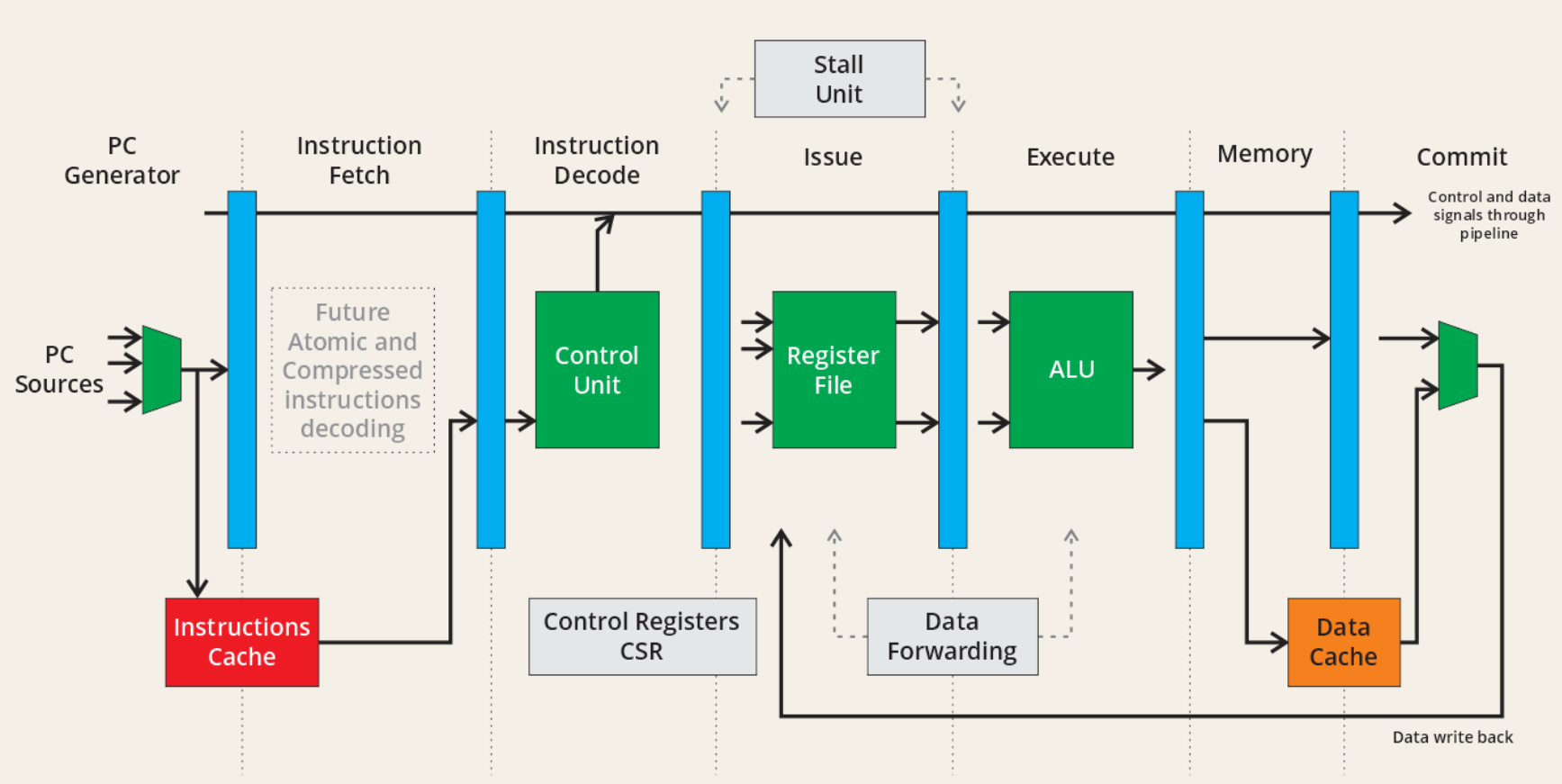

Figura 4.2.: Diagrama de etapas del *pipeline*.

Solución a Dependencias

Estas etapas, que parecen como un único camino recto, necesitan una realimentación debido a las dependencias de datos que generan. Esto se produce cuando dos instrucciones seguidas o muy cercanas usan el mismo registro (por ejemplo, la primera instrucción guarda un dato en el registro `x5` y la segunda o tercera instrucción hace uso de este registro para la operación). Esto genera que, de no realimentarse el dato, la operación se haría con un dato antiguo. Este valor no correspondería al que sigue de la ejecución del programa. Para solucionar esto, se implemento una unidad de *forwarding*. La misma, genera un bypass de la etapa de *Memoria* y etapa de *commit* hacia la etapa de *issue* y etapa de *ejecución*. De esta manera, realimenta dichas etapas con los datos nuevos permitiendo una correcta ejecución del programa. En la figura 4.3 se observan las conexiones de la unidad de *forwarding*.

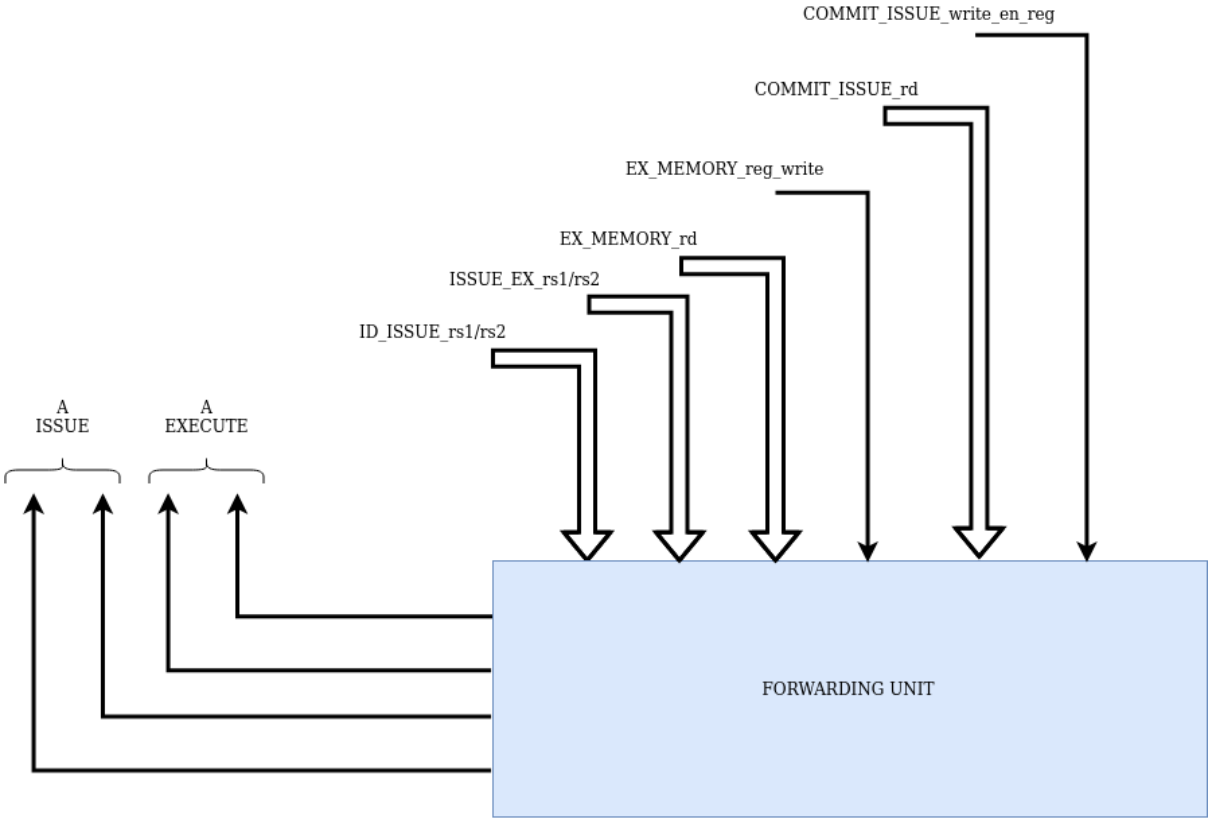

Figura 4.3.: Diagrama de la unidad de *forwarding*.

Peligro de datos

Algunas veces, las dependencias de datos creadas por el pipeline no son salvables. Por esta razón, se agrega una nueva unidad llamada unidad de *stall*. La misma se encarga de parar las etapas anteriores a la etapa de *ejecución* y colocar burbujas hasta que la operación que genero el stall termine y continué la ejecución normal del sistema. Esta unidad no solamente es utilizada en estos casos, también se utiliza para el manejo de interrupciones, cuando en la ejecución del programa se realiza un salto

o cuando el sistema debe esperar por un acceso a memoria. Esto permite parar la ejecución del programa hasta que los datos estén listos para ser usados y realizar un cambio de contexto para las instrucciones de saltos o interrupciones.

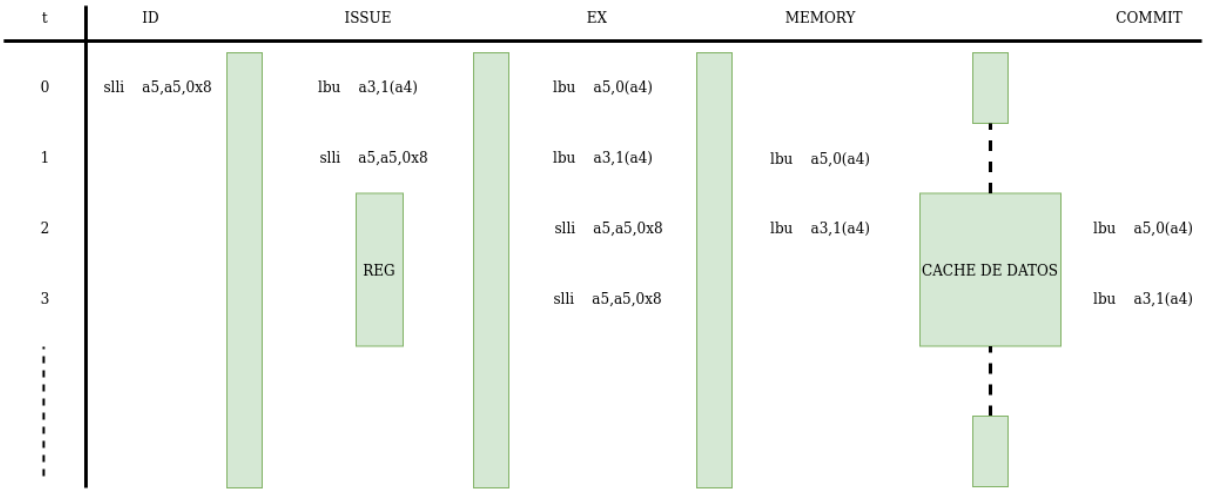

Figura 4.4.: Ejemplo de Read-After-Write.

En la figura 4.4 se observa un ejemplo de RAW (Read-After-Write). El problema es que la instrucción `slli a5,a5,0x8` leería el contenido del registro `a5` antes de que la etapa de *commit* haya escrito en su nuevo contenido proveniente de la memoria gracias a la instrucción `lbu a5,0(a4)`. Para resolver este problema se debe frenar la ejecución de la instrucción `slli` por un ciclo de clock. El responsable de esto es la unidad de *stall*. En la figura 4.5 se observan las conexiones de esta unidad con las etapas del *pipeline*.

Figura 4.5.: Unidad de Stall.

Integración de etapas

Habiendo detallado cada una de las etapas, se puede observar en el anexo A.1 la figura A.1 que muestra a bajo nivel un diagrama de como es que están interconectadas cada unas de las partes conformando así el Softcore.

4.2.2. Comunicación Interna del Softcore

Para realizar un pedido a cualquier dispositivo de desarollo propio o instanciado, se deberá utilizar algún tipo de interfaz. En este softcore, se utilizaron dos tipos de buses, uno de creación propia y el bus AXI.

En la imagen 4.6 se puede observar cual es el diseño de la arquitectura planteada.

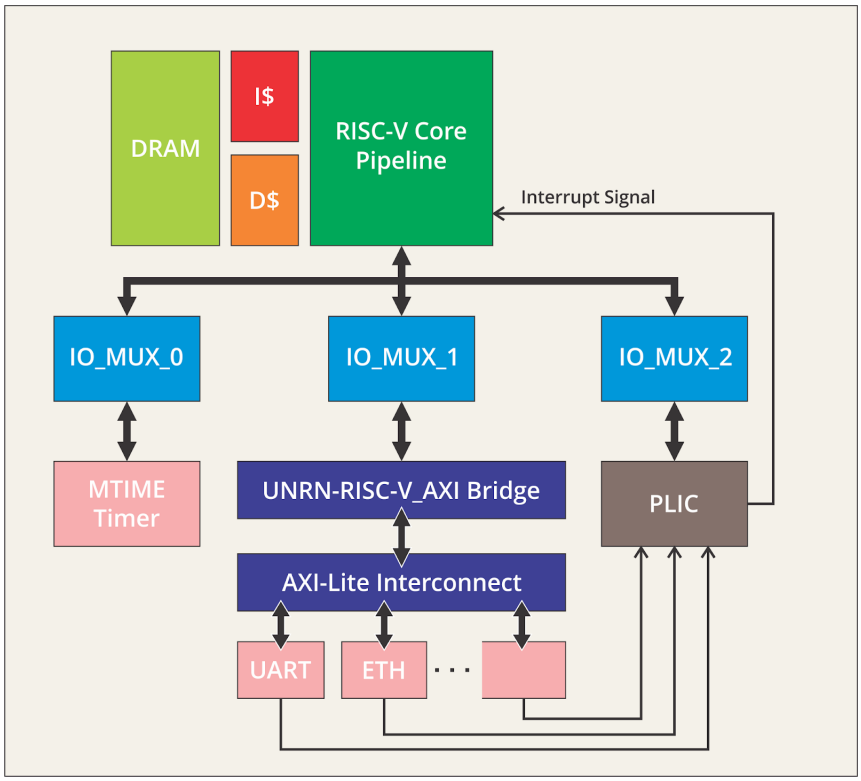

Figura 4.6.: Diagrama de la arquitectura

Bus Nativo del Procesador

Para realizar dispositivos propios nos vimos en la necesidad de tener una forma de interactuar con los mismos. Para esto, nos vimos en la necesidad de crear un bus nativo para la comunicaciones entre N dispositivos y nuestro procesador.

Para realizar la comunicación con dispositivos de nuestra autoría, se creo un modulo llamado *io mux*. Cuando se agregue un dispositivo al sistema y este sera agregado al bus nativo, se deberá crear una instancia del modulo mencionado.

El funcionamiento del modulo es muy sencillo. Este es conectado primero hacia el pipeline donde recibirá la información y enviara las respuestas del dispositivo. Luego, tendrá la comunicación hacia el dispositivo que administre. Además, las respuestas de estos dispositivos serán encadenadas, es decir, la respuesta del dispositivo $N+1$, serán enviadas al dispositivo N el cual la enviara al dispositivo $N-1$, de esta manera hasta llegar al procesador. A continuación se ve un diagrama de la conexión entre distintos de estos módulos.

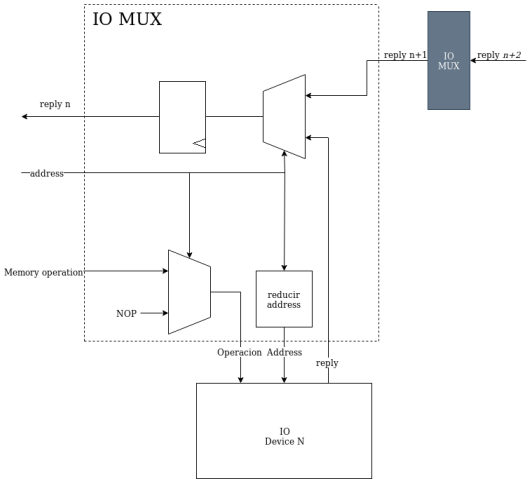

Figura 4.7.: Diagrama de Conexión de los Módulos *IO mux*.

A cada *IO mux*, le llegara de forma directa el *address*, el dato a escribir y el tipo de operación pedido. En cambio, la respuesta de cada dispositivo sera enviada a su correspondiente *IO mux* el cual lo enviara hacia el próximo *IO mux* en la cadena hasta llegar al procesador.

El ciclo de funcionamiento del dispositivo *IO mux* es el siguiente. Primero, recibirá las señales del procesador (address, dato a escribir y tipo de operación) luego, mediante el *address*, determinara si la operación corresponde o no al dispositivo que esta asociado. De corresponder, responderá poniendo en *high* la señal de done, colocando el dato pedido (en caso de ser ese tipo de operación) y informando si hay o no excepción. En el caso de no corresponder, espera la respuesta del próximo dispositivo la cual recibirá y la enviara por la cadena. En la figura 4.8 se esquematiza la conexión con dispositivos.

En caso de que el *address* no pertenezca a ningún dispositivo la señal llegara hasta un modulo llamado “chain end”. Este recibirá una señal de *match address* de todos los *io mux* y en el caso de que ninguna este afirmada le avisará al procesador que ese *address* es incorrecto.

Los dispositivos los cuales utilizan los módulos *io mux* con el correspondiente bus nativo son los siguientes:

- Bridge entre bus nativo y bus AXI.
- Dispositivo de Timer.
- Dispositivo de salida.

- UART para test.
- PLIC.

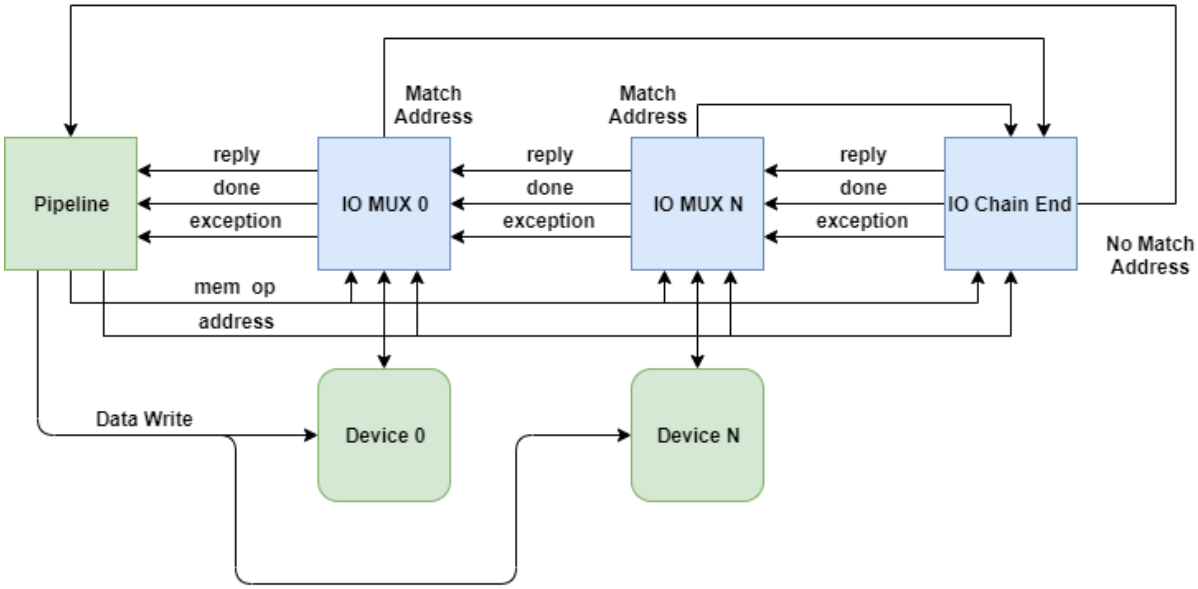

Figura 4.8.: Diagrama de Conexión de los Módulos *IO mux* con dispositivos.

Bus AXI

El bus AXI fue utilizado en dos partes:

- Comunicación con dispositivos.
- Comunicación con memoria DDR.

El bus AXI fue usado exclusivamente para dispositivos IP core de Xilinx, esto debido a que desarrollarlos costaría mucho mas tiempo del planeado. El siguiente uso fue por la comunicación con la memoria DDR que trae la placa, de esta manera lograr que los datos e instrucción del programa se encuentren guardados en la memoria de la PS (Processing System) y no en la parte de la PL (Programmable Logic). Esto nos permite, utilizar mas recursos de la PL para diseño de módulos. Para ambos casos, se utilizo un modulo el cual recibe el protocolo propio del sistema y luego es traducido al protocolo AXI. En el primer caso, el bus nativo del sistema contiene un dispositivo que funciona como bridge entre el bus nativo y el bus AXI. Luego, los dispositivos con protocolo AXI son conectados mediante un modulo llamado *AXI Interconnect*, el cual es provisto por la herramienta Vivado. Este permite conectar distintos dispositivos de tipo *Master* con distintos dispositivos de tipo *Slave* realizando traducción de protocolos como AXI4 a AXI4-lite (de ser necesario) y el manejo de distintos tipos de velocidades de buses.

4.2.3. Almacenamiento interno del Softcore

Para almacenar información, por ejemplo: datos e instrucción, se tienen los registros mencionados en la sección 2.5.1. A un nivel de jerarquia más bajo como se puede ver

en la figura 2.1 se tiene la memoria caché. La misma esta divida en dos partes, caché de instrucciones y caché de datos. Luego, las mismas están conectadas a través de un bus AXI a la memoria DDR de la placa y de este modo se obtiene más espacio de memoria.

4.3. Diseño para la FPGA

El diseño descrito en la sección anterior solamente concierne a las partes centrales del softcore. Para tener una computadora funcional es necesario tener más componentes: el clock, dispositivos de IO, memoria, conexión con el mundo exterior y demás componentes auxiliares necesarios para estos propósitos.

Para conectar estos componentes con el softcore se usó la funcionalidad de diagramas en bloques de IP core de Vivado. Ella permite conectar gráficamente los diferentes bloques que componen un diseño.

Durante el desarrollo fue práctico poder cambiar la velocidad a la que corre el softcore para poder arreglar problemas sin preocuparse por el *timing* y recién una vez que los problemas estuvieran arreglados corregir los tiempos. Para ajustar la velocidad de clock a la que corren los dispositivos es necesario editar varias constantes en cada IP core y en el sistema operativo, como esto no es práctico usamos dos regiones de clock. Una región corre el softcore y los dispositivos más simples, la otra región corre los dispositivos conectados al bus AXI. De esta forma se puede cambiar la frecuencia a la que funciona el softcore sin afectar el funcionamiento del sistema operativo o de los dispositivos.

4.3.1. IP Utilizados en la Arquitectura

Para nuestra diseño se decidió utilizar ciertos IP para facilitar el funcionamiento del sistema y para reducir los tiempos de desarrollo. Esto debido a que de otra manera estos dispositivos extenderían el tiempo desarrollo por su gran complejidad. Los IP fueron provistos por la herramienta de desarrollo de *Xilinx Vivado*. De esta forma, a continuación se nombran:

- Clock Wizard.
- Processor System Reset.
- AXI Interconnect.
- AXI UART16550.
- AXI EthernetLite.
- Ethernet PHY MII to Reduced MII.
- Constant.
- ZYNQ Processor System.

El IP *Clock Wizard*, permite recibir una señal de clock a una frecuencia particular y convertirla en un clock de menor frecuencia a elección. Esto nos da la posibilidad de utilizar el clock del sistema de 125 MHz y convertirlo en clocks de menor frecuencia.

El IP llamado *Processor System Reset* consiste en un sistema que permite generar un reset para un sistema a través de condiciones personalizadas. En nuestro caso, el uso que le daremos será tomar la señal externa de reset y convertirla en una señal sincronizada con el clock del sistema. De esta forma, convertir el reset externo asincrónico en un reset sincrónico.

Para la conexión en distintos dispositivos AXI, se utilizo el IP core *AXI Interconnect*. El mismo permite que varios dispositivos del bus AXI (tanto *master* como *slave*) se comuniquen entre si. Básicamente, este IP se comporta como un árbitro del bus, cuando un dispositivo *master* propone una comunicación a un *address* específico éste lo conecta con el dispositivo *slave* que contiene ese address. Ademas, permite la traducción entre los distintos estándares de bus AXI, es decir, se pueden conectar dispositivos de distintas frecuencias como tanto de distintas estandares (AXI3,AXI4 o AXI4-lite).

Para la interaccion del sistema de manera sencilla con el exterior, se utilizo el IP core *AXI UART16550*. El mismo implementa una UART 16550 completa con todas sus funcionalidades. Se penso utilizar dicha UART (Universal Asynchronous Receiver Transmiter) debido a que el sistema operativo Zephyr utilizado contiene el driver de la misma. De esta manera, se evita tener que lidiar con la creacion de un driver para una UART personalizada, lo cual habria alargado el proceso de diseño. Esta UART es de uso exclusivo del sistema operativo.

El IP llamado *AXI Ethernet Lite* nos da la posibilidad de tener una interfaz MII (Media Independent Interface) la cual nos permite tener la interfaz física MII de Ethernet. De esta forma, nos permite tener una interfaz independiente de comunicación la cual puede ser transformada para ser utilizada en un medio de transmisión a elección.

En el caso del IP *Ethernet PHY MII to Reduced MII*, fue utilizado debido a que este permite reducir la cantidad de señales necesarias para el PHY Ethernet MII. Esto fue utilizado debido a que el protocolo MII utiliza 16 señales las cuales no pueden ser utilizadas en un solo header de la placa de desarrollo. De esta manera, al hacer la traducción se reducen de 16 señales a 8, logrando colocar todas las señales en un solo header.

El IP *Constant* es simplemente un valor que se puede colocar para que una entrada de algún dispositivo tenga un valor fijo. En nuestro caso, fue utilizado para que una señal del IP *Ethernet PHY MII to Reduced MII* tenga un valor especifico.

4.3.2. Dispositivos Adicionales de la Arquitectura

Un dispositivo agregado al sistema es el PLIC (Platform Layer Interrupt Controller). Este dispositivo se encarga de alertar al procesador que algún dispositivo externo al sistema generó una interrupción. Para poder manejar correctamente estas interrupciones es necesario darle algunos datos correctamente. El dispositivo esta conectado al bus del sistema como un dispositivo, y así poder configurar ciertas funcionalidades como habilitar interrupciones, leer información, entre otras cosas.

Para la comunicación externa del sistema, se crearon dos submódulos, uno llamado Ethernet y otro llamado UART 16550 (el cual es simplemente el IP *AXI UART16550*). En el caso del Ethernet, se encuentran los ip AXI EthernetLite, Ethernet PHY MII

to Reduced MII, Clock Wizard y Constant. Los IP AXI EthernetLite, Ethernet PHY MII to Reduced MII y Constant son utilizados como ya se menciono anteriormente mientras que el Clock Wizard es usado para generar un clock de referencia para la comunicación del bus Ethernet.

Un dispositivo particular del sistema, es una uart diseñada en el bus nativo del sistema, la cual es unicamente utilizada para los test del ISA. De esta manera, se evita toda la configuración inicial necesaria para utilizar la UART 16550 dejando los test del ISA sencillos sin programación extra que podrian generar errores innecesarios.

Para tener una posibilidad sencilla de un dispositivo de salida, se creo un pequeño modulo el cual a través del bus nativo del procesador genera un set o un clear de señales especificas. Las mismas, están conectadas a los LEDs de que trae incorporado la placa de desarrollo logrando de esta manera encender o apagar los mismos.

4.3.3. Conexión del softcore con la memoria

Se utiliza la memoria DRAM presente en la PS de la Zynq como memoria principal. Para acceder a ella es necesario conectarse como *master* al bus AXI. Para esto es necesario convertir la interfaz nativa del softcore en el bus AXI. El IP core responsable por esto es el *data_mem_combined*. Debido a esto, la estructura de conexión del softcore con sus dispositivos y memorias es de la siguiente manera.

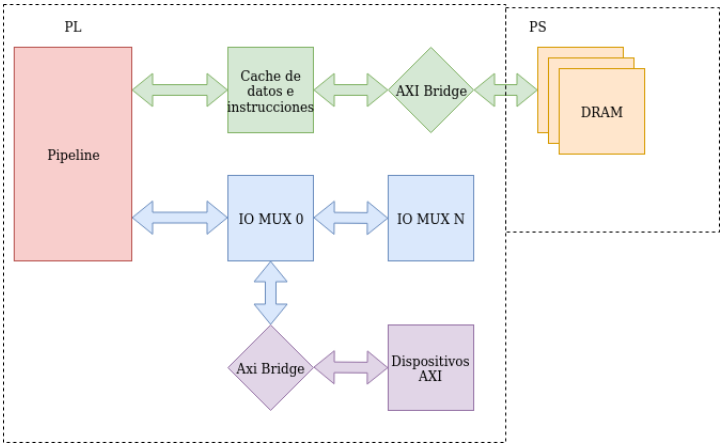

Figura 4.9.: Diagrama de la conexión del softcore con la memoria

Para poder simular el sistema en el simulador de Vivado es necesario evitar almacenar el programa en la memoria DRAM, porque, como se explica en las tablas 4.3 y 4.4, esto requiere correr un programa en el core ARM. El `data_mem_combined` posee un modo alternativo de funcionamiento que se activa cuando no se está sintetizando. Este modo consiste en cargar un archivo `.mem` pasado como parámetro de simulación y llenar una memoria creada en Verilog con su contenido.

En la figura 4.11 se puede observar la conexión a la memoria DRAM a través del bus AXI.

La función de cada IP core se describe a continuación.

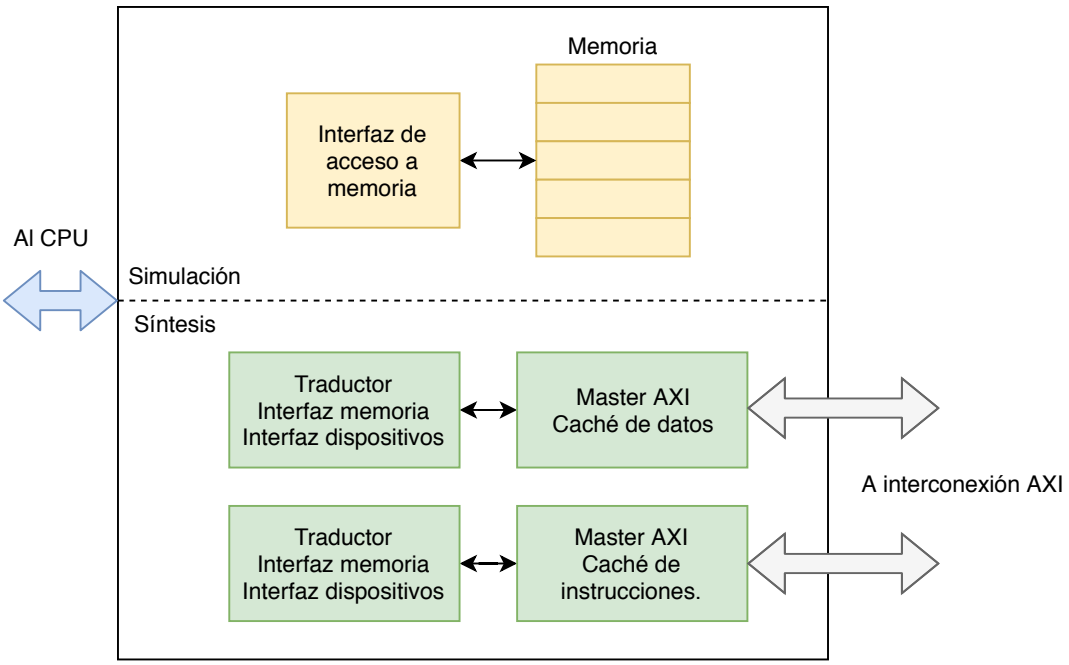

Figura 4.10.: Diagrama de bloques del componente *data_mem_combined*

processing_system7_0: Este IP core *ZYNQ7 Processing System* sirve para conectarse a la parte PS (Processing System) del chip. Esto se hace a través del bus AXI de nombre `S_AXI_GP0`.

data_mem_combined_0: Este IP core *data_mem_combined* traduce entre la interfaz nativa de acceso a memoria (descripta en la sección 4.4) y el bus AXI. Posee dos *masters* AXI y dos conexiones al *pipeline* porque tanto la caché de datos como la de instrucciones acceden a la memoria por separado.

pipeline_wrapper_0: Este IP core contiene el softcore. Dentro de él se encuentran la caché de datos, caché de instrucciones, puente entre memoria e IO.

axi_interconnect_1: Este IP core *AXI Interconnect* conecta los dos masters que tiene el IP core *data_mem_combined_0* con el único *slave* que posee el *processing_system7_0*.

Las señales que entran y salen de esta parte del diagrama son:

PLIC_TARGET_interrupt: Esta señal se activa cuando el PLIC indica que recibió una interrupción para este core. Ver sección 4.6 para más detalles.

MTIMEDEV: Estas señales corresponden al dispositivo que controla el timer.

clocks y resets: Esta parte del diagrama funciona con el clock de la CPU. No hay cruces de dominio de clock. El bus AXI de la PS está comandado por el clock de la cpu para que la interconexión no tenga que generar circuitos de cruce de dominio de clock.

IO_MUX: Estas señales se dirigen a los dispositivos de IO. Se originan en el puente entre memoria e IO. Ver secciones Bus Nativo del Procesador

DDR: Estas señales se dirigen al controlador DRAM.

FIXED_IO: Estas señales se conectan con los pines de MIO (Multiplexed IO) de la FPGA.

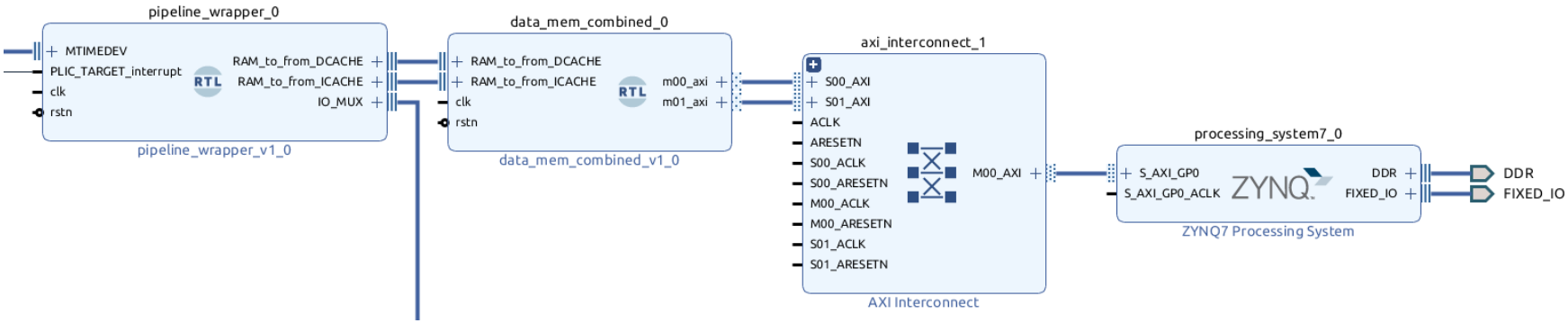

Figura 4.11.: Diagrama de conexión del softcore con la PS para la memoria

4.3.4. Generación de clock y reset

La parte de generación de clock toma un clock de referencia, en este caso el del oscilador de la placa a 125 MHz, y lo convierte en varios clocks para diferentes partes del sistema. Se usan dos regiones de clock: una región a 80 MHz para el softcore y los dispositivos más simples y otra región para los dispositivos conectados al bus AXI que corre a 150 MHz.

En la figura 4.12 se encuentran los componentes de esta parte del diagrama en bloques, que son:

clk_divider: Este módulo toma el clock de la placa a 125 MHz y lo transforma en dos clocks como se explicó anteriormente.

proc_sys_reset: El propósito de este módulo es ordenar la salida del estado de reset de los dispositivos y el softcore. Primero salen los módulos de interconexión AXI, luego los dispositivos y por último el softcore. Este módulo está diseñado para ser usado con procesadores Microblaze, pero se puede adaptar su salida para el core que diseñamos.

reset_mux: Este módulo se usa para elegir entre las señales de reset. El sistema de CI usa la señal `arduino_resetn` para poder resetear el softcore programáticamente. Para el desarrollo era más práctico resetear el core usando un switch de la Zybo. La elección entre ellos se hace a través de una resistencia pullup/pulldown conectada a un pin de la placa.

cross_domain_level_0 y cross_domain_level_1: El `proc_sys_reset` genera las señales de reset sincronizadas al clock más lento, en este caso el del softcore, por ello es necesario implementar un cruce de dominio de clock para llevarlas a los dispositivos que usan el clock AXI. Para ello se usan 3 flip-flops conectados en serie y activados por el clock AXI. Los módulos de tipo `cross_domain_level` encapsulan esta funcionalidad.

not_gate: Esta compuerta *not* es necesaria para adaptar la señal de reset activo alta del Microblaze a la señal activo baja que necesita el core que diseñamos.

Las señales que comunican esta parte del diagrama son:

clk: Clock externo generado por la placa. Su frecuencia es 125 MHz.

arduino_resetn: Entrada externa que depende de una conexión a un header. Sirve para resetear el softcore.

sw_resetn: Entrada externa que depende de un switch de la placa. Sirve para resetear el softcore.

use_switch_resetn: Entrada externa que depende de una conexión a un header. Sirve para elegir entre `arduino_resetn` y `sw_resetn` como señal de reset para el diseño.

cpu_clk: Señal de clock para el softcore.

cpu_interconnect_aresetn: Señal de reset para la interconexión que maneja el acceso a la DRAM.

cpu_rstn: Señal de reset para el softcore.

axi_clk: Señal de clock para dispositivos AXI.

axi_interconnect_aresetn: Señal de reset para la interconexión que maneja los dispositivos AXI.

peripheral_aresetn: Señal de reset para periféricos AXI.

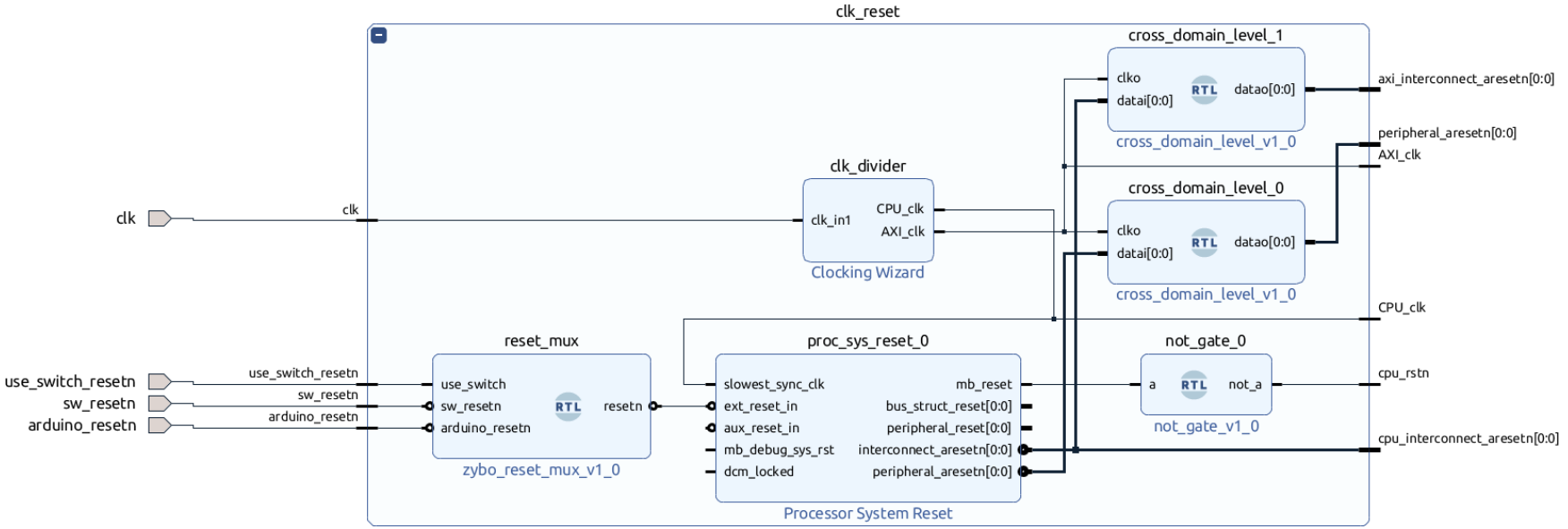

Figura 4.12.: Diagrama de componentes para la generación del clock

4.3.5. Interfaz con dispositivos de IO

Los dispositivos externos al softcore son conectados al módulo *device conexion* excepto la memoria DRAM. Este dispositivo esta compuesto internamente por los modulos *IO mux* explicados anteriormente en la sección 4.2.2. Estas conexiones no representan nada distintivo a diferencia de la que se ve en el modulo mtime device y AXI bridge.

El modulo *mtime device* funciona como una interfaz para los registros del timer del sistema debido a que por especificación los mismos deben encontrarse conjuntamente con los registros CSR pero con la diferencia que deben ser accedidos como si fueran un dispositivo externo. A continuacion se observa una imagen de la conexion del modulo con registros CSR y el *IO mux*.

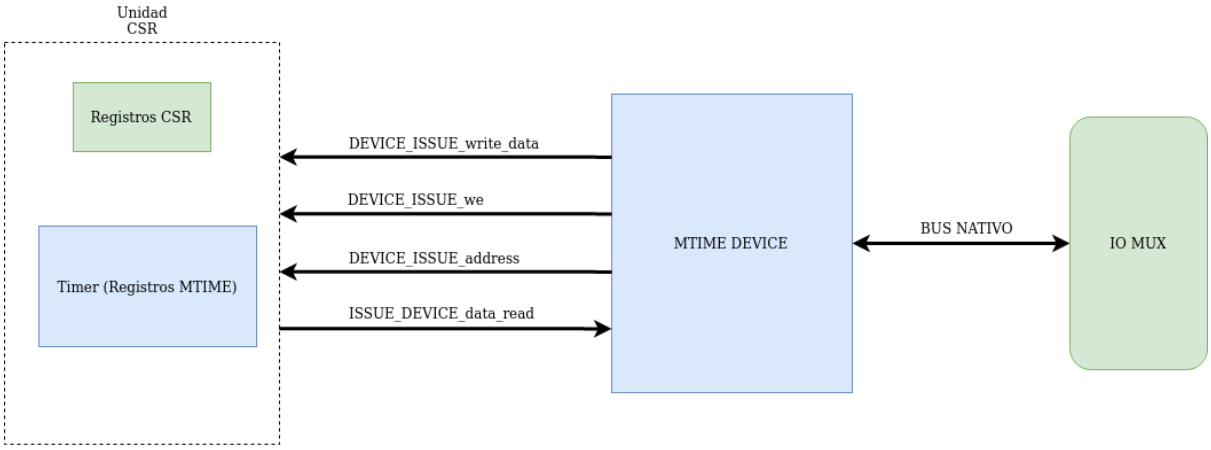

Figura 4.13.: Diagrama del módulo MTime Device y sus conexiones

En la figura 4.14 se observa dos módulos específicos los cuales son el AXI bridge (también mencionado anteriormente), el cual genera la traducción del bus nativo al bus AXI. Y por otro lado, se encuentra el modulo *mtime device*.

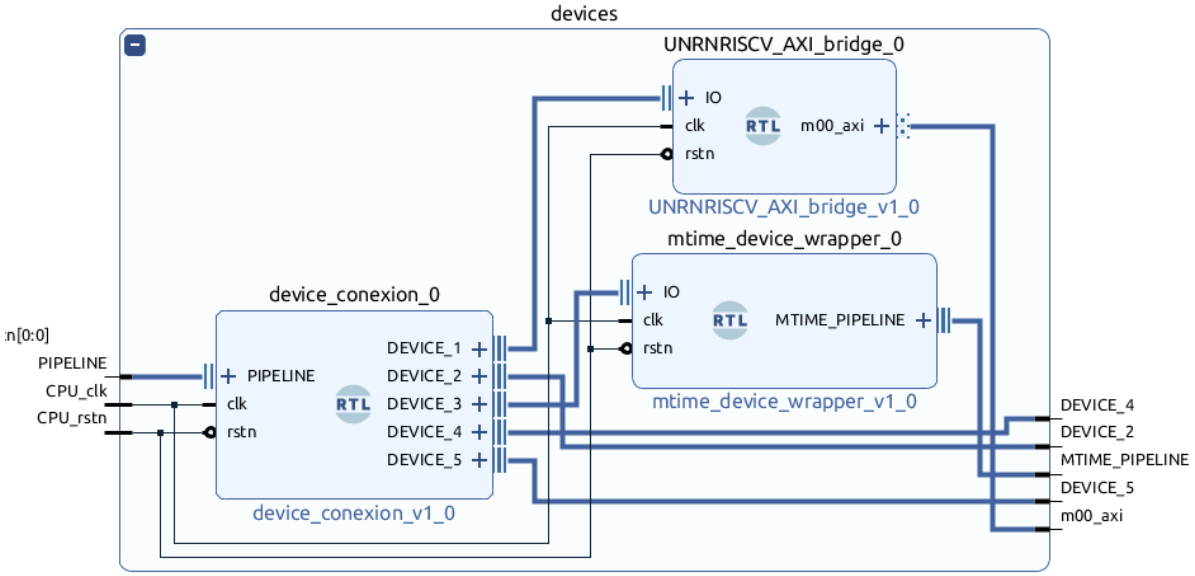

Figura 4.14.: Diagrama del módulo que agrupa los IO mux

Cada uno de los dispositivos tienen asignado un espacio de memoria especifico. El modulo de *device_conexion_0* tiene como entrada ciertos parametros que permite ser configurados facilmente por medio del BD. En la figura 4.15 se observa el método de seteo del address base de los dispositivos como también la profundidad de address.

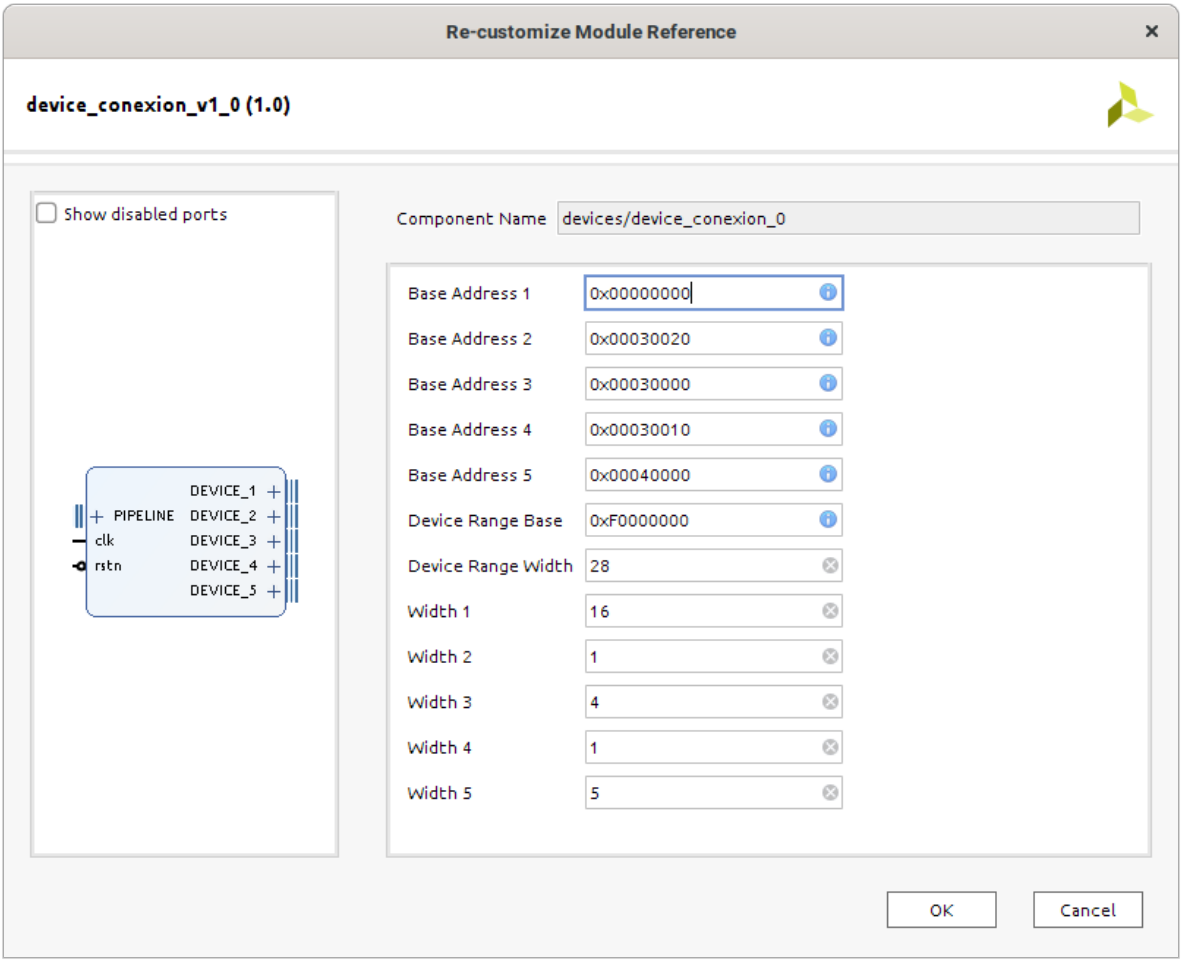

Figura 4.15.: Configuración del conjunto de IO mux

4.3.6. Dispositivos de IO

Los dispositivos de io conectados al bus axi son los que se observan en la imagen. Ademas, cabe destacar que se observa la conexión de las señales de interrupción de cada dispositivo conectadas al PLIC. El cual las tomará y se encargará de las mismas.

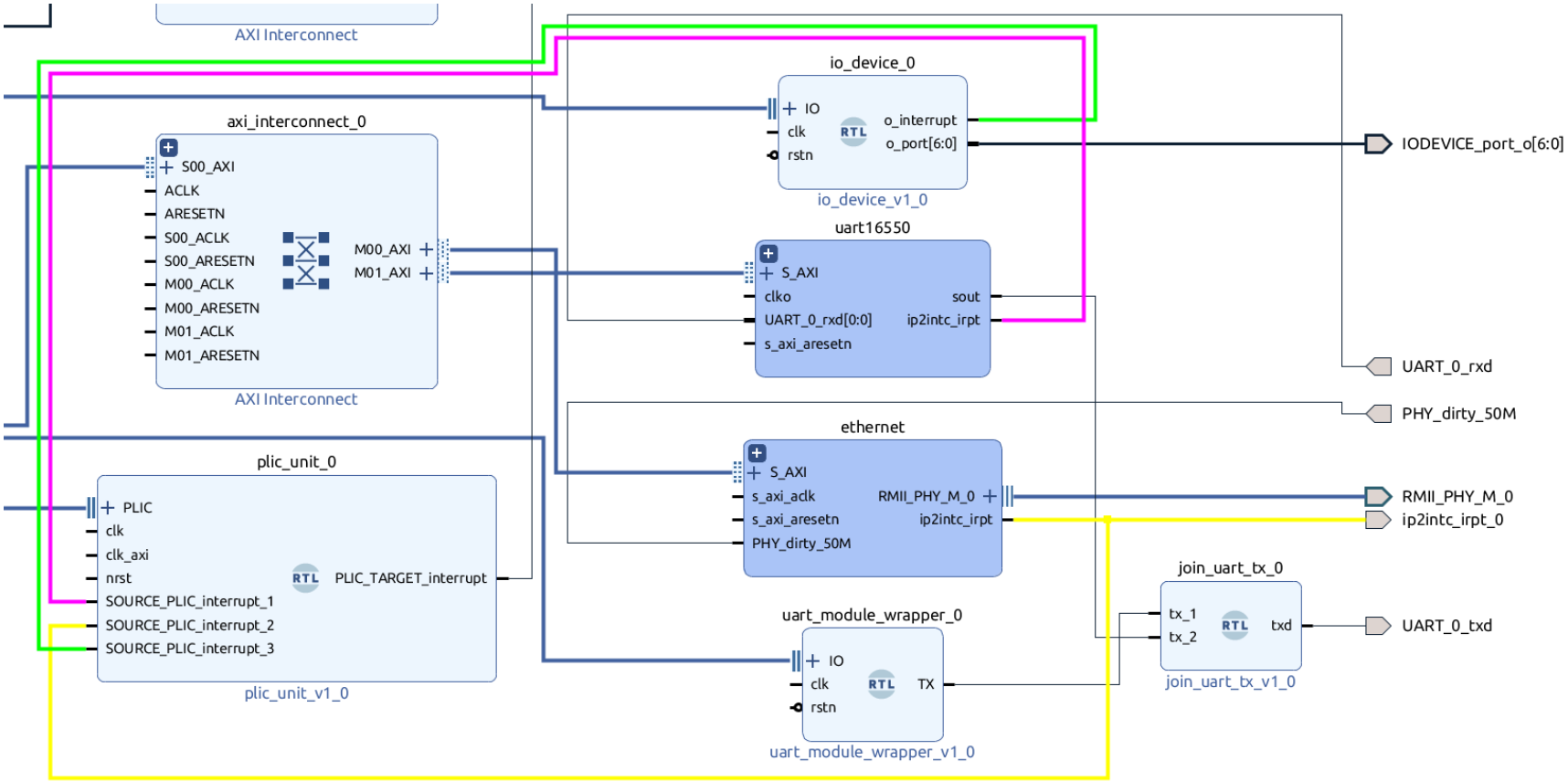

Figura 4.16.: Diagrama de bloques de dispositivos

4.3.7. IP de UART 16550

En la figura 4.17 se encuentra el esquema de conexión de la UART 16550. La parte más importante es que la señal `UART_0_rxd` pasa por un circuito sincronizador para evitar problemas de metaestabilidad en la entrada de la UART.

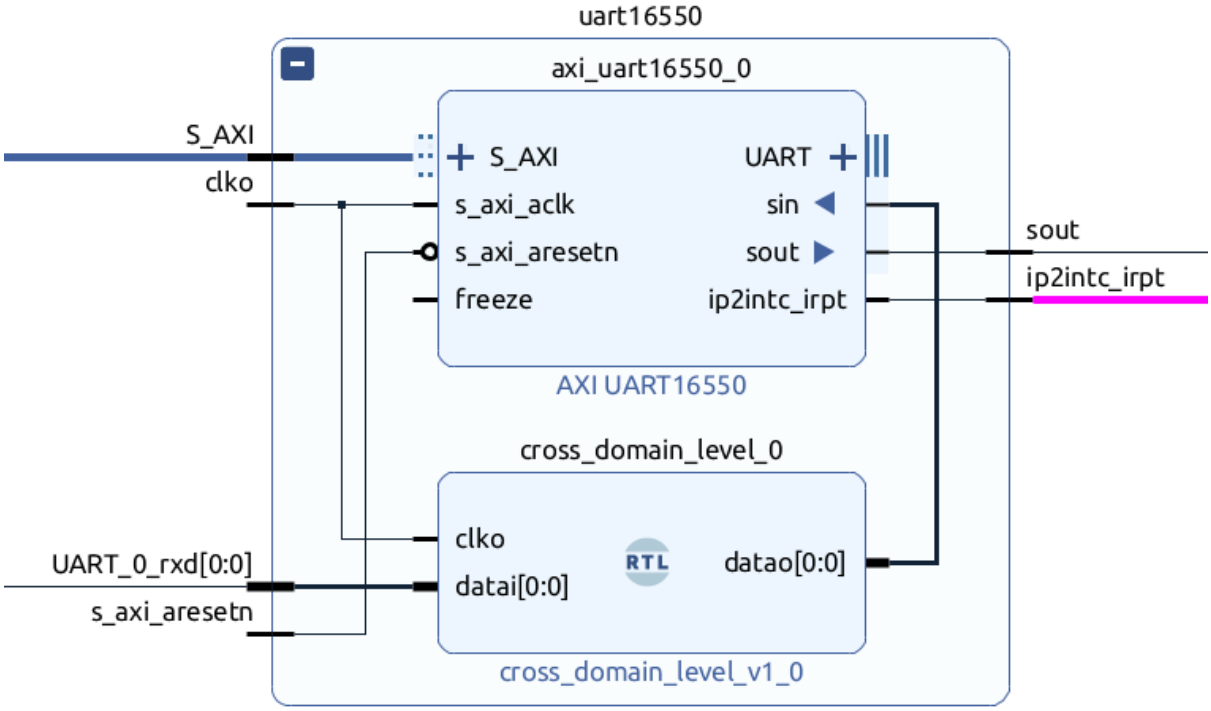

Figura 4.17.: Diagrama de bloques de los componentes de la UART 16550

4.3.8. IP de Ethernet

Para la comunicación por Ethernet se usan tres IPs. El IP de *AXI EthernetLite* es el principal.

axi_ethernetlite_0: Este IP core *AXI EthernetLite* se encarga de presentar una conexión por el bus AXI para que se puedan utilizar sus funcionalidades de comunicación Ethernet. Es un MAC Ethernet. Para ese propósito utiliza una interfaz de tipo MII para comunicarse con el dispositivo PHY Ethernet.

PHY_clock: Este IP core *Clocking Wizard* es el responsable de recuperar el clock de 50MHz generado por el PHY Ethernet. Este clock es el de referencia de las señales del bus RMII.

mii_to_rmii_0: El IP core *Ethernet PHY MII to Reduced MII* es el responsable de convertir el bus MII al bus RMII que se usa el PHY Ethernet LAN8720.

Para que la recuperación del clock funcionara se configuró el `PHY_clock` para que genere un clock con la misma fase que tiene el clock ruidoso `PHY_dirty_50M`, el cual

es proveniente de la placa externa utilizada LAN 8720. Esto es necesario para que el circuito que depende de este clock pueda trabajar correctamente. Además, para compensar la demora que representan las pistas entre el oscilador y la FPGA se agregó un constraint que define la latencia del clock como 3ns. Esto junto con los constraints apropiados según el datasheet del LAN 8720 que asegura que los tiempos de las señales sean correctos.

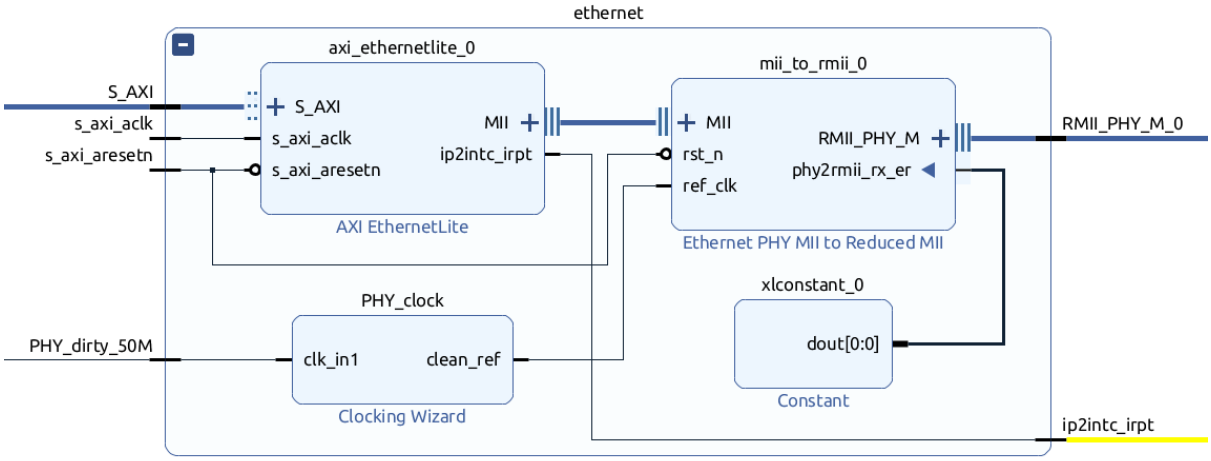

Figura 4.18.: Diagrama de bloques de los componentes para la funcionalidad de Ethernet

Como se menciono anteriormente, la placa externa utilizada fue una LAN 8720 la cual provee el medio fisico de conexion para el servicio de ethernet. A continuacion se observa una imagen de la placa y sus pines.

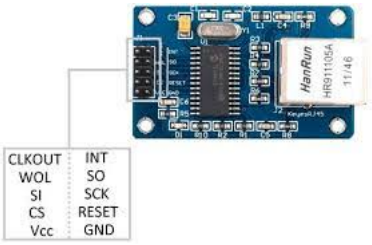

Figura 4.19.: Imagen de la placa utilizada

4.4. Implementación de la caché de datos

Los programas suelen acceder al mismo sector de la memoria varias veces en un periodo corto de tiempo o acceden a direcciones adyacentes en la memoria. Estas propiedades se llaman proximidad temporal y proximidad espacial, respectivamente. Gracias a estas propiedades es posible implementar una caché que acelere el acceso a la memoria.

Una caché es un dispositivo que pretende acelerar el acceso a una memoria lenta. Se obtiene ese objetivo a través de una copia de parte de la memoria en otro medio que

sea más rápido. Para simplificar el diseño de la caché y su costo de implementación se hacen copias de sectores grandes de la memoria, cada sector se conoce como *linea de caché*. Esta copia implica varios accesos a la memoria subyacente, pero se espera que una vez que se haga la copia ésta sea usada varias veces antes de que sea descartada. De esta forma el costo de copiar la linea de caché se amortiza entre varios accesos a la caché.

En el caso de la caché implementada en este proyecto la latencia deseada durante el diseño era de un ciclo de clock. Con este requerimiento se podía garantizar que el pipeline funcionara a máxima velocidad. Este requerimiento condicionó el diseño elegido, así como también fue condicionado por el tipo de recursos que dispone la FPGA.

La caché de datos está implementada como una caché de mapeo directo. En este tipo de caché hay una sola ubicación posible para cada palabra de la memoria dentro de la cache y se determina a través de partes de la dirección. Un ejemplo está en la figura 4.20, en ella la dirección `0xABCD1234` se descompone en los índices correspondientes a linea de caché, índice de bloque e índice dentro de bloque.

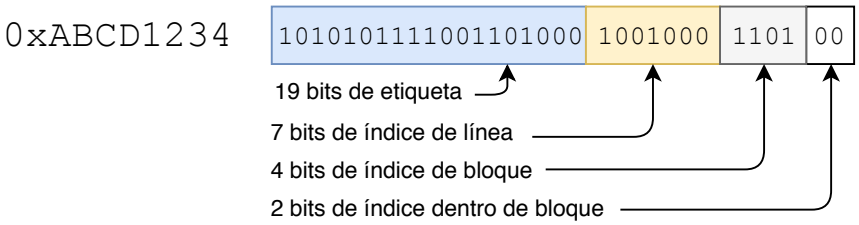

Figura 4.20.: Ubicación de una dirección de memoria en la caché

Si bien una palabra de la memoria solo puede tener una ubicación en la caché, esto no es así para las líneas de cache. Cada linea de caché puede almacenar copias de muchas diferentes regiones. Para conocer cuál es la que actualmente está en la caché se utiliza la memoria de etiqueta y el bit de *válido*. Este bit indica si la linea de caché se encuentra llena o vacía y si estuviera llena la memoria de etiqueta almacena la etiqueta correspondiente. La disposición de los datos en la memoria de la caché se encuentra en la figura 4.21.

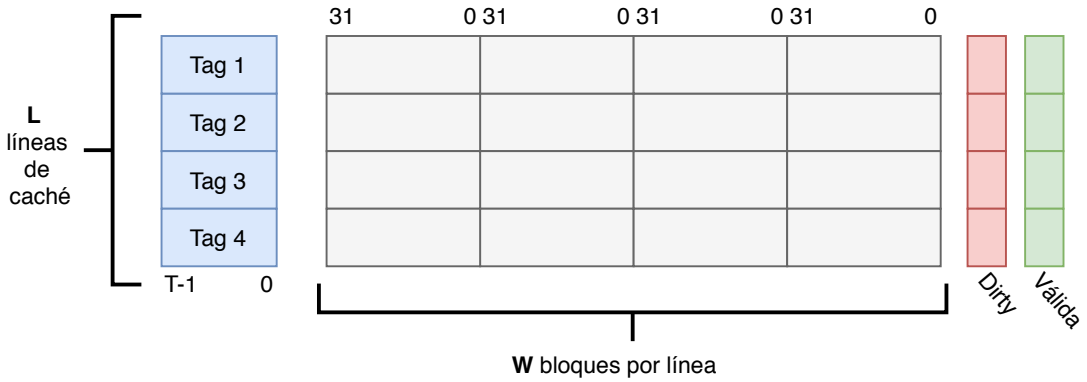

Figura 4.21.: Diagrama esquemático de la caché de datos

Como se puede ver en la figura 4.21 hay un bit más por cada línea llamado *dirty*. Este bit indica si los datos en la caché son más actuales que los que se encuentran en la memoria. En caso de que esté en `1` cambia la acción a realizar cuando haya que cambiar el sector de memoria que se aloja en esa linea de caché.

Existen dos bloques de memoria principales: el bloque de etiquetas o *tags* y el bloque de datos donde se almacenan las lineas de caché. Es posible cambiar su tamaño a través de los parámetros **W** para el ancho de la linea de caché y **L** para la cantidad de líneas.

El conjunto de señales que comunican la caché con el puente entre memoria e IO está en la tabla 4.1. El propósito de cada señal se describe en la tabla 4.2.

Desde	Hacia	Tipo	Nombre
Membridge	Dcache	`logic [RAM_WIDTH-1:0]`	`address`
Membridge	Dcache	`common::mem_op_t`	`mem_op`
Membridge	Dcache	`bit`	`flush`
Membridge	Dcache	`logic [31:0]`	`wdata`
Membridge	Dcache	`logic [1:0]`	`req_tx_id`
Dcache	Membridge	`logic [31:0]`	`data`
Dcache	Membridge	`logic [1:0]`	`reply_tx_id`
Dcache	Membridge	`logic`	`exception`

Cuadro 4.1.: Señales que conectan la caché de datos con el puente entre memoria e IO

`address`	Indica la dirección de memoria sobre la que se va a realizar la operación.
`wdata`	Son los datos a escribir en caso de una operación de escritura.
`mem_op`	Enumeración que indica la operación a realizar
`flush`	Indica que la caché debe escribir todo su contenido en la memoria.
`data`	Contiene los datos que se leyeron o código de excepción.
`exception`	Indica si los datos de la señal `data` son un código de excepción.
`req_tx_id`	Cuenta el id del pedido que se realiza a la caché.
`reply_tx_id`	Cuenta el id del pedido con el que se asocian las señales de salida.

Cuadro 4.2.: Descripción de señales que comunican puente entre memoria e IO y caché de datos

El conjunto de señales que comunican la caché con la memoria RAM está en la tabla 4.3 y el propósito de cada señal se describe en la tabla 4.4

Para cumplir con el requerimiento de respuestas en un ciclo de clock y que la implementación sea sencilla implementamos un mecanismo de solicitud de pedidos que utiliza contadores de 2 bits para asociar los datos con los pedidos a los que corresponden.

Para realizar una solicitud el puente entre memoria e IO debe incrementar el contador `req_tx_id`. La caché de datos compara este contador con el número del último pedido que completó, si es diferente entonces toma los datos de las demás entradas como un pedido nuevo. Cuando éste termina, la caché coloca el valor del contador al inicio del pedido en la señal `reply_tx_id`. De esta forma el puente entre memoria e IO entiende que los datos que se presentan en las señales de salida son válidos y corresponden al último pedido que realizó.

Desde	Hacia	Tipo	Nombre
Dcache	RAM	logic [RAM_WIDTH-1:0]	addr
Dcache	RAM	logic	write_enable
Dcache	RAM	logic	addr_valid
RAM	Dcache	logic	addr_ack
Dcache	RAM	logic [31:0]	write_data
Dcache	RAM	logic	write_data_valid
RAM	Dcache	logic	write_data_ack
RAM	Dcache	logic	write_done
Dcache	RAM	logic	write_done_ack
RAM	Dcache	logic [31:0]	read_data
RAM	Dcache	logic	read_data_valid
Dcache	RAM	logic	read_data_ack

Cuadro 4.3.: Señales que conectan la caché de datos con la memoria RAM

addr	Indica la dirección de memoria.
write_enable	Indica que la operación es de escritura.
addr_ack	Indica que la dirección de memoria ha sido recibida.
addr_valid	Indica que la señal addr contiene datos válidos.
read_data	Datos leidos.
read_data_ack	Indica que los datos leidos han sido recibidos.
read_data_valid	Indica que la señal read_data contiene datos válidos.
write_data	Datos a escribir.
write_data_ack	Indica que los datos a escribir ya han sido recibidos.
write_done	Indica que la escritura ha sido completada.
write_done_ack	Indica que la señal write_done ha sido recibida.
write_data_valid	Indica que la señal write_data contiene datos validos.

Cuadro 4.4.: Descripción de señales que comunican la caché de datos con la memoria RAM

4.4.1. Máquinas de estado de la caché de datos

La caché de datos tiene dos máquinas de estados que trabajan coordinadas. La máquina de estado principal se encarga de atender los pedidos del puente entre memoria e IO y de describir los pedidos a la memoria principal. La máquina de estado secundaria se encarga de tomar estos pedidos, transferirlos a la memoria principal y escribir el resultado en la memoria de bloques de la caché.

Existen dos conjuntos de estados en la máquina de estados principal. Los que están resaltados en celeste en la figura 4.22 son los que se encargan de la operación de *flush*. Estos estados recorren todas las lineas de caché y las que estén *dirty* son copiadas a la memoria principal. El estado *pre flush all* actualiza un contador que recorre todas las lineas de la caché y transiciona al estado *Wait flush all*. En este estado se espera hasta que la máquina de estados secundaria termine de copiar la línea de caché a la memoria principal. Luego regresa al estado *Pre flush all* si quedan lineas para copiar. Los estados resaltados en amarillo son los que están involucrados cuando se reciben pedidos de acceso a direcciones de memoria que están en la caché.

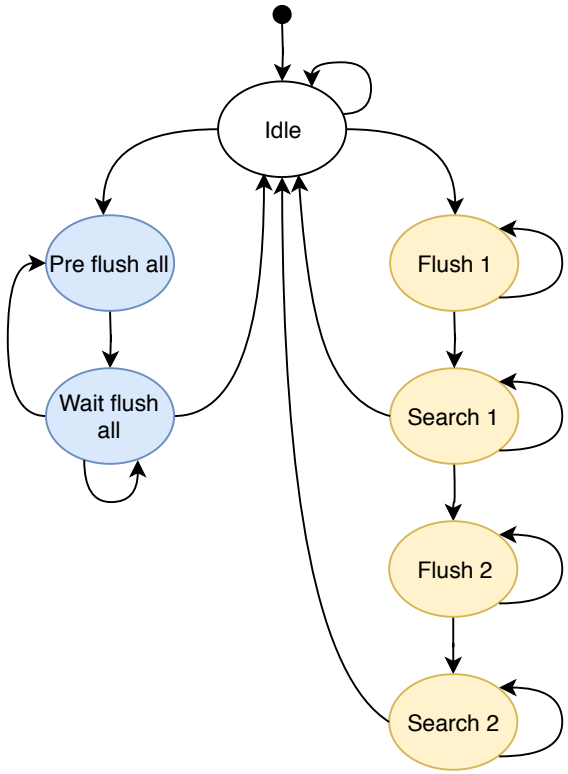

Figura 4.22.: Diagrama de estados de máquina de estados principal

Todos los pedidos de acceso a la memoria tienen que pasar por los estados *Flush 1* y *Search 1*. El *1* hace referencia a que estos estados trabajan sobre la linea de caché en la que se ubica el byte menos significativo del pedido. El estado *Flush 1* se encarga de copiar la linea a la memoria principal en el caso de que esté más actualizada que la memoria principal. El estado *Search 1* se encarga de pedir la linea a la memoria principal.

Es posible que la solicitud a la caché implique una operación sobre dos lineas de caché. Para manejar este estado están los estados *Flush 2* y *Search 2*. El *2* hace referencia a que estos estados trabajan sobre la linea sobre la que se extiende la operación, o en otras palabras, la linea donde se ubica el byte más significativo del pedido. Su comportamiento es análogo a los estados *Flush 1* y *Search 1*.

Los estados de la máquina de estos secundaria están descriptos en la figura 4.23. En ella se puede ver que hay dos clasificaciones diferentes de estados. Los estados relacionados a las lecturas de lineas de caché están representados en celeste y los relacionados a escrituras de lineas de caché están representados en amarillo.

Cuando la máquina de estados secundaria recibe un pedido de parte de la máquina de estados principal transiciona al estado *Do read* o *Do write* según corresponda. Estos estados inician una operación con la memoria principal y transicionan al estado *Wait read* o *Wait write*, respectivamente. Estos estados esperan hasta que la memoria complete la operación. Una vez que la operación haya sido completada se incrementa un contador y se regresa al estado *Do read* o *Do write* según corresponda. Así se transfiere en varias operaciones de memoria una linea de cache desde o hacia la

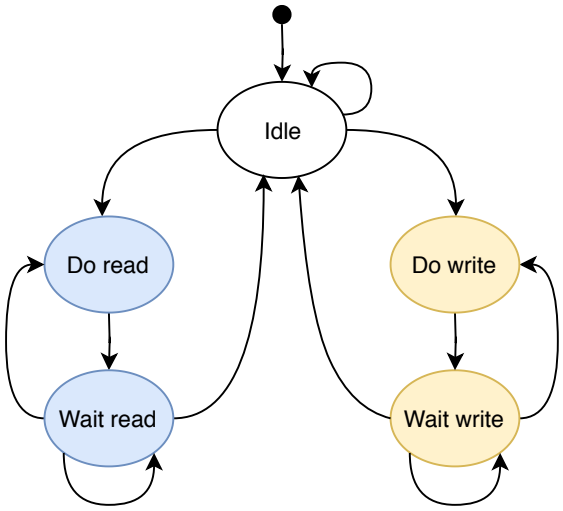

Figura 4.23.: Diagrama de estados de máquina de estados secundaria

caché de datos. Cuando se completan todas las operaciones el estado *Wait read* o *Wait write* transiciona al estado *Idle* e indica a la máquina de estados principal que su pedido ya ha sido completado.

4.4.2. Implementación de la memoria de bloques en la FPGA

Uno de los objetivos de diseño de la caché es poder implementar todas las operaciones de acceso a memoria en un solo periodo de clock en el caso de que los datos se encuentren en la memoria de bloques. Esto incluye las operaciones sobre direcciones desalineadas y escrituras menos anchas que la palabra (32 bits).

Estos requerimientos imponen otros requerimientos sobre la memoria de bloques, en particular: debe poder escribir sobre dos palabras en el mismo ciclo de clock y debe permitir escribir sobre parte de la palabra, también debe permitir la lectura de dos palabras adyacentes en el mismo ciclo de clock. Estos requerimientos no eran compatibles con los primitivos de memorias que provee la FPGA.

Para cumplir estos requerimientos creamos un diseño con varios bloques de memoria de 8 bits de ancho. De esta forma como máximo ocurre una escritura y una lectura en cada memoria por cada ciclo de clock. En la figura 4.24 hay un diagrama de la disposición en bloques de RAM de la memoria de bloques.

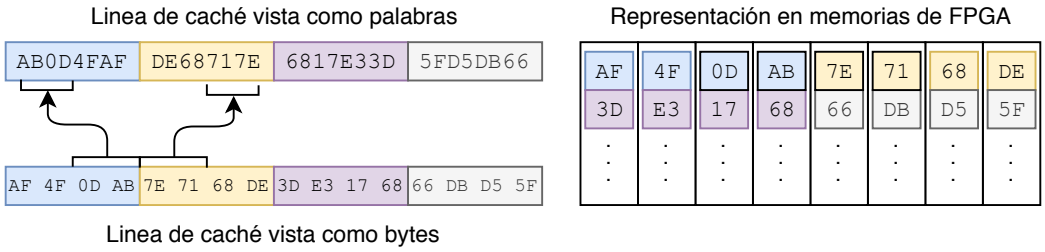

Figura 4.24.: Implementación de la memoria de bloques

4.4.3. Diferencias con la caché de instrucciones

La implementación de la caché de instrucciones es similar a la implementación de la caché de datos, pues está basada en esta. Sin embargo, hay algunas simplificaciones por el hecho de que esta caché no realiza escrituras ni accesos desalineados. Estos dos detalles permiten que la máquina de estados sea mucho más sencilla al carecer de los estados *Pre flush all*, *Wait flush all*, *Search 2* y *Flush 2*.

La memoria de bloques está implementada con un solo bloque de memoria, pues el acceso es a palabras completas tanto para lectura como escritura.

4.5. Implementación del puente entre memoria e IO

Para el acceso a memoria principal se usa un espacio de direcciones distinto al usado para los dispositivos. Esto permite distinguir el tipo de pedidos de acuerdo a las direcciones que acceden y dirigirlos al módulo correcto en función de esto. Para realizar la conexión entre la CPU, la caché de datos y los dispositivos de IO se utiliza un módulo que es el puente entre memoria e IO. En la figura 4.25 hay un diagrama esquemático de este componente.

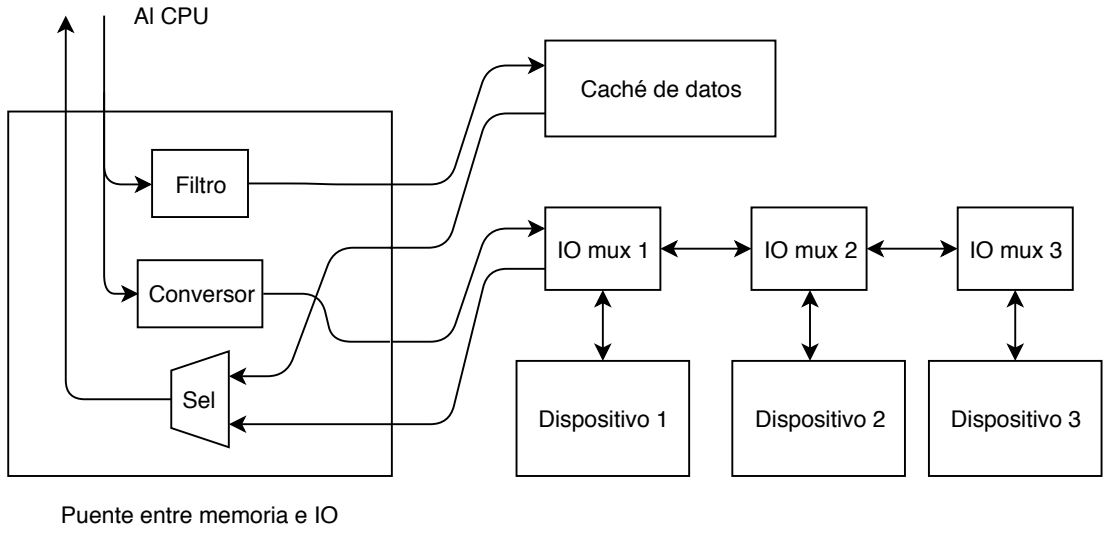

Figura 4.25.: Diagrama interno del puente entre memoria e IO

La interfaz usada para conectarse con los dispositivos de IO es más sencilla que la que tiene la caché de datos. Por una parte, esto es una ventaja ya que facilita la implementación de los dispositivos, pero por otra parte es una desventaja, porque las operaciones con dispositivos demoran varios ciclos de clock, durante los cuales la CPU está frenada.

La interfaz entre el puente entre memoria e IO y la CPU es igual a la que conecta la caché de datos con el puente entre memoria e IO. En la tabla 4.5 se describen las señales que la componen. Las primeras señales son análogas a las que comunican el puente con la caché de datos, para su descripción ver la tabla 4.2. La señal

address_valid cumple la función de indicar a la etapa de *commit* que la dirección usada no es válida, es decir que no está dentro del espacio de memoria de la caché y que no está en el espacio de ningún dispositivo.

Desde	Hacia	Tipo	Nombre
MEMORY	Puente	`logic [RAM_WIDTH-1:0]`	address
MEMORY	Puente	`common::mem_op_t`	mem_op
MEMORY	Puente	bit	flush
MEMORY	Puente	`logic [31:0]`	wdata
MEMORY	Puente	`logic [1:0]`	req_tx_id
Puente	COMMIT	`logic [31:0]`	data
Puente	COMMIT	`logic [1:0]`	reply_tx_id
Puente	COMMIT	logic	exception
Puente	COMMIT	logic	address_valid

Cuadro 4.5.: Señales que conectan la CPU con el puente entre memoria e IO

El puente entre memoria e IO está implementado con una máquina de estados. Es necesario que los pedidos para la caché de datos sean completados en un ciclo de clock, por lo que el estado *idle* debe pasar la operación a la caché de datos y el estado que espera la respuesta de la caché debe ser capaz de iniciar una operación. Como las señales entre la caché de datos y el puente entre memoria e IO son similares, la máquina de estados solamente debe ver que la operación corresponda al espacio de direcciones de la memoria y si es así direccionar los datos hacia ella. Para el caso de operaciones de dispositivos de IO es necesario convertirlas al tipo de bus que se usa para ellos, en la sección 4.2.2 se detalla qué señales lo componen.

4.6. Implementación del PLIC

Todo dispositivo que quiere comunicarse con el procesador, en primera instancia debe mandar un aviso al PLIC. Este luego de haber recibido la señal define cuando será atendido por el procesador. Es posible que se tengan varios dispositivos y cada uno tenga diferentes caracteristicas, como por ejemplo:

- Velocidad: Cada dispositivo puede tener diferentes frecuencias de operación, por lo cual es necesario solventar los cruces de dominio de clock.
- Prioridad: Cada dispositivo puede tener una prioridad diferente, lo cual se requiere saber cual es el dispositivo a atender con mayor urgencia.
- Interrupción: Es posible tener dos tipos de señales de interrupción, una por nivel y la otra de ellas por flanco. Cada una tiene una forma particular de ser atendida.

Dada esta variedad de opciones, al momento que diseñamos y desarrollamos el PLIC (Platform Layer Interrupt Controller) tuvimos que tener en cuenta estas variedades. Frente a este panorama se tomó la decisión de que el PLIC fuera modular, lo cual significa que los parametros sean configurables y que si se llegasen a agregar más dispositivos sea de una forma sencilla y rapida, sin tener que realizar demasiadas modificaciones.

Todas las consideraciones mencionadas anteriormente solamente son para estar al alcance de las necesidades de los dispositivos. Otra de las consideraciones es que el PLIC pueda comunicarse de forma adecuada con el procesador y para ello se utilizo el bus nativo del procesador.

Con estos parametros, el PLIC debía tener las señales de interface de la figura 4.26.

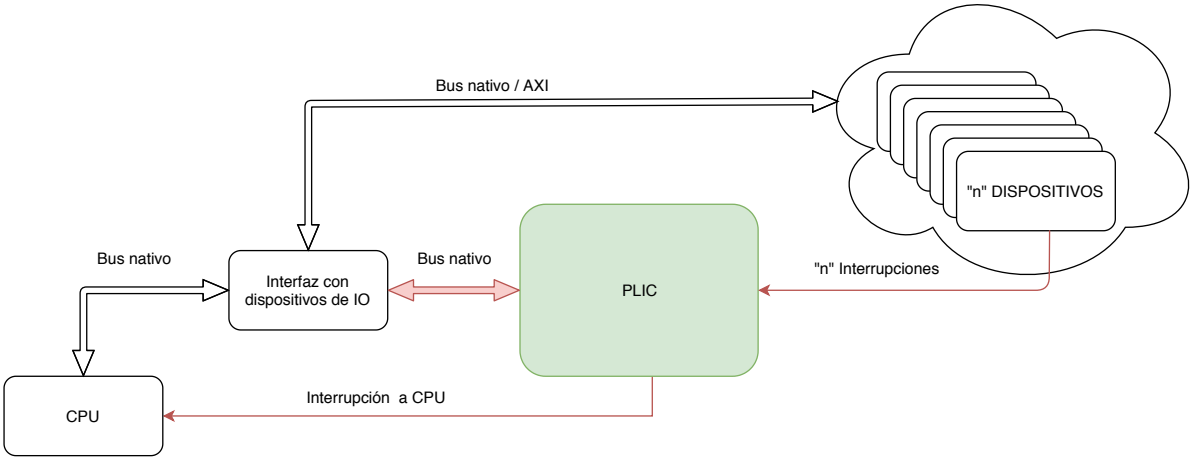

Figura 4.26.: Señales necesarias del PLIC

La señal “interrupción a CPU” como también la señal “n interrupciones” corresponden a una señal de 1 bit en donde la primera sería para alertar al CPU que tiene una interrupción pendiente para atender y la otra señal sirve para avisarle al PLIC que un dispositivo tiene algo para enviar.

Las señales del bus nativo corresponderian a las señales de operación, dirección, dato a escribir, dato a leer, excepción y finalización de operación.

Para que el PLIC pueda comunicarse correctamente con el CPU y pueda tomar las decisiones correctas al momento de recibir una interrupción el ISA recomienda usar una serie de registros que almacenen cierta información. Los registros que se utilizaron fueron:

- Registro IE: Este registro habilita o deshabilita las interrupciones de los dispositivos que estan conectados al PLIC. Si un dispositivo genera una interrupción y tiene deshabilita su interrupción, el PLIC descarta cualquier petición del mismo.
- Registro PRIORITY: Cada dispositivo tiene una prioridad, siendo la misma para definir cual es el dispositivo que se va a tomar al momento de que mas de un dispositivo haya generado una interrupción.
- Registro EDGELEVEL: Este registro, si bien el Driver no lo utiliza, se usa para definir como es la interrupción del dispositivo conectado, la cual podria ser por nivel o por flanco. Esto es porque dependiendo de como es la interrupción a tomar debe ser manejada de formas diferentes.
- Registro THRESHOLD: Uno podría decidir que dispositivos tomar colocando un umbral a nivel de software. Si el dispositivo que genero una interrupción tiene un nivel de prioridad mayor al umbral, la interrupción se tomaría. Esto es

util debido a que uno podria tener multi cores con diferentes umbrales y que cada uno de ellos pueda o no manejar las interrucpiones de los dispositivos compartidos o no.

- Registro COMPLETE: Cuando se termina de atender la interrupción el CPU escribe sobre este registro almacenando cual fue el ID del dispositivo que atendio así como también al momento de escribir sobre este registro permite que pueda tomarse una nueva interrupción.
- Registro CLAIM: Este registro se utiliza en conjunto con el registro COMPLETE para que al momento de leer el mismo se devuelva al CPU el ID (identificador) del dispositivo, entonces cuando el CPU quiere tomar una interrupción sabe cual es el dispositivo a atender. Este registro también se utiliza para unas señales internas que definen si hay o no interrupciones pendientes.

Sabiendo que se tiene registros que almacenan información, que hay una logica para definir cual es el ID a atender y que cada uno de los dispositivos tiene señales de interrupción pendiente o que tiene la información necesaria para avisar cuando recibió una interrupción se tomó la decisión de que se dividiera en 3 bloques y que cada uno de ellos cumpla una función especifica.

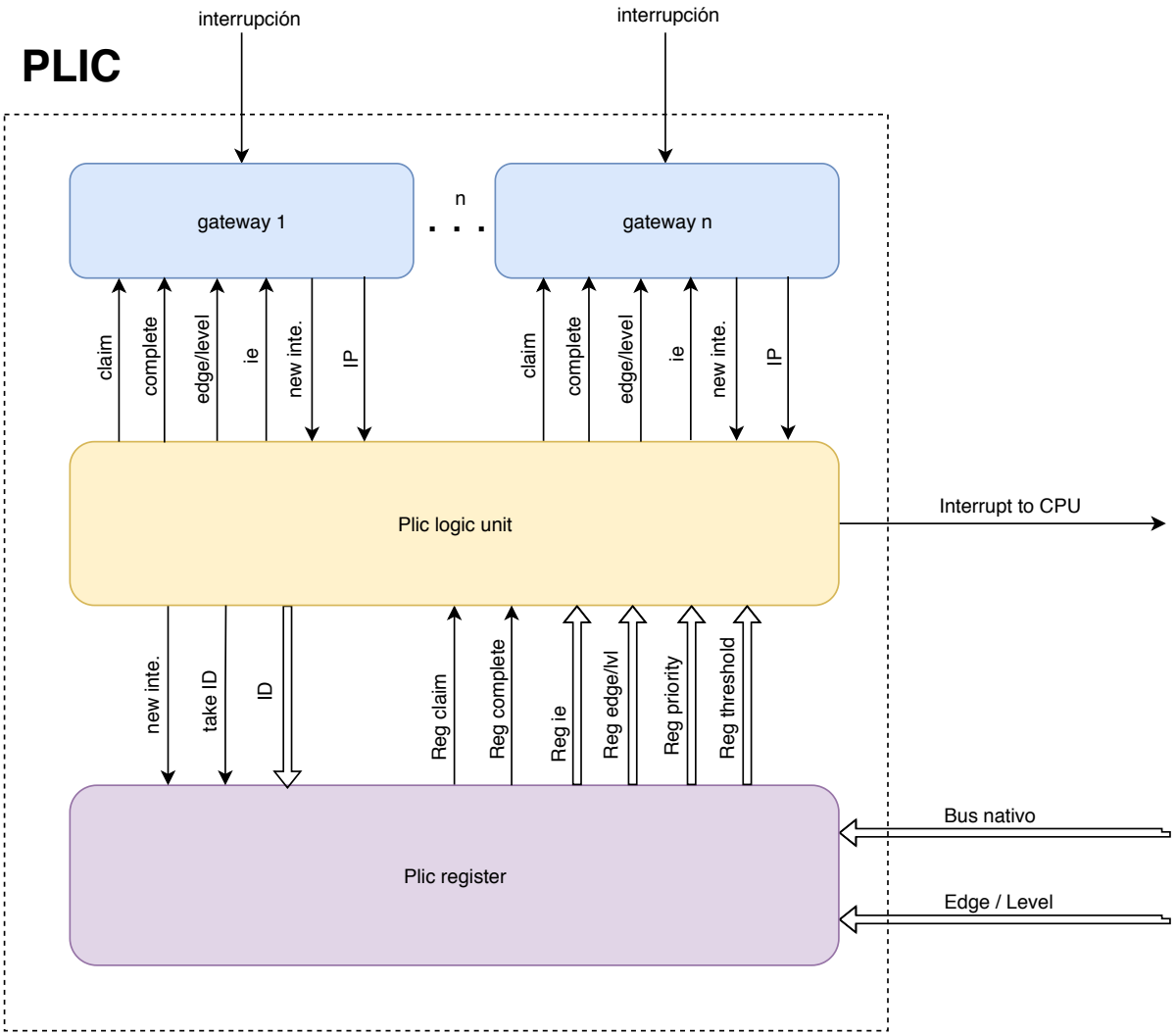

Figura 4.27.: Diagrama en bloques del PLIC

4.6.1. Gateway

El bloque gateway es el encargado de recibir la interrupción externa y definir de alguna manera si se va a tomar o no. Este bloque esta constituido por una maquina de estados que permite determinar en qué estado de transacción se encuentra con el CPU. Los mismos definen cuándo se tiene una IP (interrupción pendiente), cuándo hay una nueva interrupción, como también definen la saturación de la cola de interrupciones, si es que la interrución es por flanco.

A continuación se mostrara un diagrama de la maquina de estados y como afecta cada uno de los estados en algunos parametros importantes del bloque como el contador y el IP.

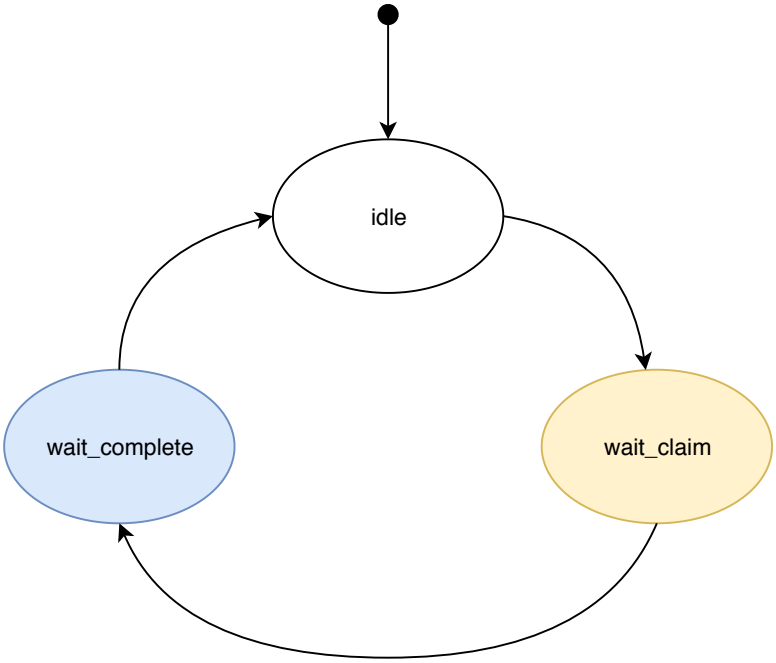

Figura 4.28.: Machina de estados de Gateway

Cuando se inicializa, la maquina de estados comienza en el estado de *idle*. Para pasar al estado de *wait_claim* puede ser por 3 situaciones diferentes, siendo dos de ellas para dispositivos con interrupciones por flanco y una de ellas por dispositivos que generen interrupciónes por nivel.

- Por nivel: Si se recibe una interrupción y el dispositivo este habilitado para interrupciones, directamente en el proximo instante de clock se pasara al estado de *wait_claim*. Cuando este en este nuevo estado se levantara la señal de IP y por un ciclo de reloj se levantara una señal de que se recibio una nueva interrupción. Cuando se recibe un claim, se pasara directamente al estado de *wait_complete*.
- Por flanco: Cuando la maquina esta en el estado de *idle* es posible transicionar al proximo estado por dos motivos. Uno de ellos es que se tenga una interrupción pendiente en el contador. Otra alternativa es que se reciba una interrupción, el dispositivo este habilitado y que no este colapsado el contador de interrupciones. Cuando se recibe una interrupción y no esta colapsado el contador se incrementa en 1 el contador. Cuando la maquina esta en el estado de *wait_complete* sucede exactamente lo mismo que para dispositivo por nivel.

Cuando la maquina ya haya transicionado al estado de *wait_complete* solo pasara al estado de *idle* solo cuando se reciba un complete. El contador decrementara cuando ya este en el nuevo estado, el anterior estado haya sido *wait_complete* y el contador sea diferente de cero.

4.6.2. PLIC Logic

El bloque plic logic unit es el que se encarga principalmente de definir el ID del dispositivo que se atendera si es que hubo una o más interrupciones. Se utilizaron

dos maquinas de estados en donde una de ellas es la encargada de mandarle la señal al CPU de cuando hay una interrupcion para atender y la otra de ellas se usa para hacer una busqueda de cual es el ID que se tomará de una manera optimizada, lo cual explicaremos a continuación.

A continuación se mostrara un diagrama de las maquinas de estados y se dara una explicación de como es el flujo en general.

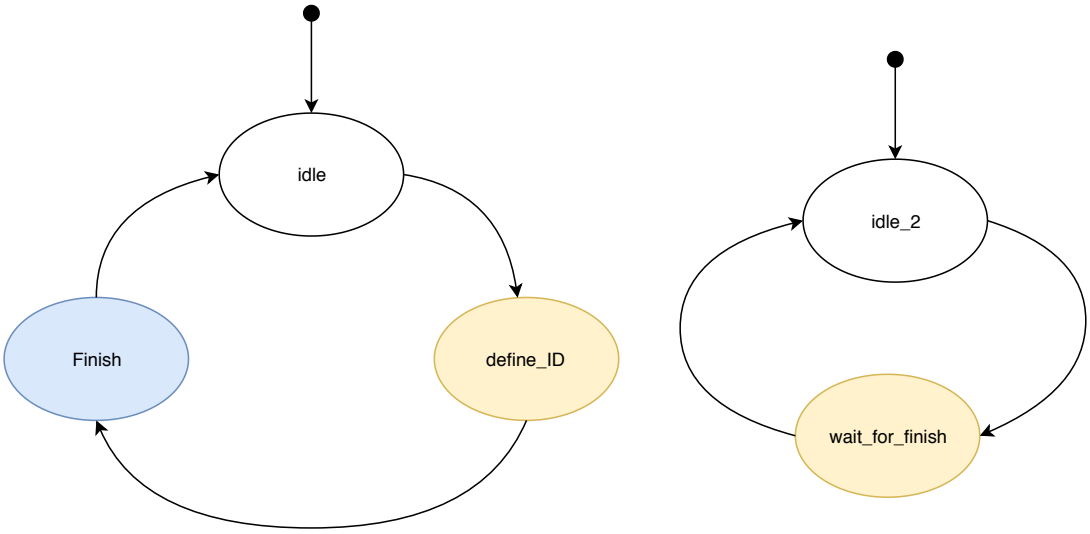

Figura 4.29.: Machina de estados de Plic unit

La maquina de estados de la izquierda es la encargada de definir el ID. Al momento de inicializarse comienza en el estado de *idle* en donde se define que la maxima prioridad y el máximo ID visto son cero. Si la señal de IP se encuentra afirmada,se pasa al estado de *define_ID*. Al momento de definir que el próximo estado es *define_ID* y el estado actual es *idle* se levanta una señal al bloque de Register plic avisandole que hay una nueva interrupción. Otra alternativa para que se levante esta señal es que el estado actual este por varios ciclos de clock en *define_ID* y el bloque de gateway mande una señal de que tiene una nueva interrupción. En este nuevo estado, lo que se hace es que cuando se tiene una nueva interrupción se haga un barrido por todos los dispositivos que estan habilitados (IE) observando cual es la prioridad que tiene cada uno de ellos y calcular si el valor supera el threshold. También, se calcula si la prioridad que se esta viendo es mayor o igual a la prioridad que se tomo anteriormente. A medida que se va avanzando y se van cumpliendo todos los criterios, se va guardando el ID que corresponderia tomar. El mismo valor es calculado en este bloque. Por ello, es necesario tener sincronizado todo con el bloque de Register plic ya que todos estos valores de IE, PRIORITY, THRESHOLS son leidos de los registros. Si no hay sincronizacion lo que puede suceder es que los valores que se estan leyendo sean del dispositivo 2 y este bloque haya calculado que los valores sean del dispositivo 1 provocando que al momento de que el CPU pida el ID se le devuelva uno equivocado, ejecutado el driver de un dispositivo que no genero ninguna interrupción. Al finalizar el barrido de todos los dispositivos conectados, se pasa al estado de *Finish*. Al momento en el que se encuentra en este estado, se levanta una señal al bloque de Register plic para avisarle que ya puede tomar el ID,

entonces si se llegase a recibir un claim se enviara el ID correspondiente. Si llegase a pasar el caso en el que llega una nueva interrupción y aun no haya llegado el claim, se vuelve al estado de *define_ID* para ver si la nueva interrupción tiene mas prioridad que la actual. Cuando llega el claim en el estado de *finish* se pasa al estado de *idle* y no se queda esperando el complete. Esto permite que si hay interrupciones pendientes, se vaya haciendo un barrido de forma prematura para ver cual va a ser el proximo ID a tomar.

La maquina de estados de la derecha es la encargada de mandarle la señal al CPU de que hay una nueva interrupciòn. Al momento de inicialiazarse arranca en el estado *idle_2* y solo pasa al siguiente estado si la maquina de estados de la izquierda se encuentra en un estado diferente a *idle*. La transición de estado a *wait_for_finish* genera un pequeño pulso al CPU para avisarle de la interrupción. Cuando esta en el nuevo estado solo volvería al estado inicial si llegase un complete. Esto quiere decir que si llegase a haber varias interrupciones encoladas a la espera de ser atendidas, al momento de que llegue el complete es muy probable que la otra máquina de estados ya haya determinado cual es el proximo ID, provocando una transición relativamente inmediata del estado *idle_2* al estado *wait_for_finish*.

Lo que se busco en tener dos maquinas de estados, es aumentar un poco la eficiencia en la determinación de que dispositivo se tomará y que no se haga la busqueda solo cuando el CPU haya termino de atender el dispositivo. La latencia que hay para que un dispositivo sea atendido es bastante grande, por lo cual esta metodologia permite de alguna manera ahorrar algunos ciclos de clock aumentando la performance en general.

4.6.3. Register plic

El bloque Register plic funciona como interfaz con el procesador como también con el bloque de plic logic. Internamente, esta compuesto por los registros que se mencionaron anteriormente distribuidos de la siguiente manera.

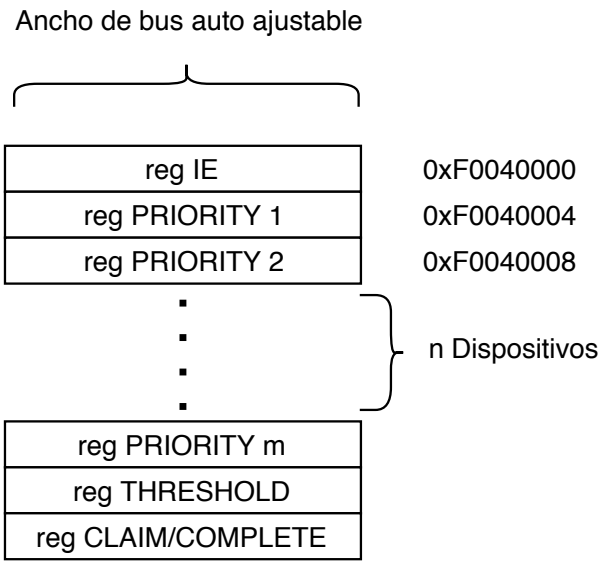

Figura 4.30.: Registros del Plic

Para poder interfacear con el procesador necesita poder comunicarse con el mismo por medio del bus nativo, por lo cual necesita cumplir con cierto requisitos, como por ejemplo que al momento de realizar una lectura sobre los registros se seteen las señales de io_done cuando realmente el dato esta disponible y colocado en el bus.

4.7. Sistema operativo Zephyr

Para verificar el correcto funcionamiento del core junto con los dispositivos se utilizó el sistema operativo Zephyr. Éste es un RTOS (Sistema operativo de tiempo real) que puede correr sobre la arquitectura RISC-V. Para poder correrlo sobre el core que diseñamos fue necesario describir el hardware sobre el que corre: esto significa describir los espacios de direcciones, los dispositivos, cómo están conectados al PLIC, etc.

Para agregar soporte a Zephyr de nuestro proyecto seguimos la guía de portado a otras placas que provee la documentación del sistema operativo. Entre los cambios que se hicieron están:

Archivos `Kconfig`: Estos archivos definen la configuración del sistema operativo. En ellos se definen elementos como el espacio de direccionamiento, características del sistema operativo y los drivers que están habilitados (UART, red, IP, TCP, etc).

Archivo `dts`: En el archivo `dts` correspondiente a nuestro proyecto se describen los dispositivos de hardware disponibles, es decir, en qué dirección se encuentra la UART, el PLIC, cómo están conectados entre sí. También se define la velocidad de clock de la UART, ésta es necesaria para que el driver pueda configurar correctamente el baudrate.

Archivo `dts` fixup: En este archivo se asocian las definiciones que contiene el `dts` con las variables que necesita el sistema operativo y sus drivers, por ejemplo:

```
#define DT_PLIC_MAX_PRIORITY                    \
DT_RISCV_PLIC0_F0040000_RISCV_MAX_PRIORITY
#define DT_PLIC_PRIO_BASE_ADDR                  \
DT_RISCV_PLIC0_F0040000_PRIO_BASE_ADDRESS
#define DT_PLIC_IRQ_EN_BASE_ADDR                \
DT_RISCV_PLIC0_F0040000_IRQ_EN_BASE_ADDRESS
#define DT_PLIC_REG_BASE_ADDR                   \
DT_RISCV_PLIC0_F0040000_REG_BASE_ADDRESS
```

Estas lineas vinculan la definición del PLIC hecha en el `dts` con las variables que usa su driver.

Para la UART implementada con el IP de Xilinx se utilizó el driver de UART 16550 que provee zephyr. No fueron necesarios cambios en el aparte de la configuración básica.

Para el IP *AXI EthernetLite* implementamos un driver simple en C. La escritura de este driver se hizo siguiendo las implementaciones de otros drivers, en particular el de la placa E1000, que era el más sencillo según nuestro entendimiento. También consultamos mucho la documentación del IP. Para este dispositivo también fue necesario agregar configuraciones a los archivos `Kconfig`.

El driver tiene dos funciones principales:

Manejo de interrupciones: En esta parte se reciben los pedidos de interrupción del dispositivo, que ocurren cuando se reciben datos. Aquí se copian los datos a un buffer de tipo `net_buf` asociado a un `net_pkt` y se informa al sistema operativo que llegó un paquete Ethernet. El sistema operativo es el encargado de revisar su validez y transmitirlo a las capas IP, TCP, etc. según corresponda.

Envío de paquetes: En esta etapa se recibe un paquete ya armado desde la capa L2 del sistema operativo, se lo copia a un buffer del dispositivo y se levanta una bandera que solicita su transmisión.

4.8. Uso de la PS (Processing System) de la FPGA

La principal razón para usar la PS de la FPGA es que es allí está el controlador de la memoria DRAM. Poder acceder a la memoria DRAM es una gran ventaja en cuanto al tipo de programas que se pueden correr sobre el procesador, es decir, un mayor espacio de almacenamiento permite correr programas más sofisticados.

Es imposible acceder desde la PL (Programmable Logic) (donde está implementado el core) a la PS si ésta no ha sido inicializada primero. Esto significa cargar un programa simple que escriba en los registros necesarios para habilitar la comunicación entre ambas partes del chip. Afortunadamente el BSP (Board Support Package) generado por el SDK de Xilinx es suficiente para inicializar esos registros, así que el programa solo debe cargar en la memoria el programa para el core RISC-V.

Para aprovechar el proceso de carga de binarios para el procesador para copiar el programa del softcore se almacena el programa como un *array* de C. Este array se coloca en una sección llamada *program_data* y el script de *linker* la coloca en la dirección `0x100000`, que es donde está mapeada la DRAM en ambos procesadores.

El procedimiento de carga de un programa es el siguiente:

1. Se obtiene el archivo ELF del programa a ejecutar. Es crucial que los datos de programa estén almacenados a partir de la dirección `0x100000`, con la primera instrucción exactamente en esa ubicación.
2. Se copia la parte correspondiente a la memoria de programa a un archivo binario.
3. Se utiliza el programa `srecord` para copiar el archivo binario a un archivo que contenga un *array* de C.
4. Se incluye este archivo en el proyecto del SDK para que se compile.
5. Se instruye al *linker* a que coloque este array en la dirección `0x100000`, que es la que espera el softcore.
6. Se compila el binario para el procesador ARM.
7. Se programa el *bitstream* y luego se programa el binario.
8. Se baja la señal de `reset` del softcore.

Un ejemplo de *array* con un programa es este:

```
__attribute__((__section__(".program_data")))
unsigned char eprom[] =
{
  0x97, 0x02, 0x00, 0x00, 0x93, 0x82, 0x02, 0x01, 0x73,
  0x90, 0x52, 0x30, 0x6F, 0x00, 0x50, 0x32, 0x13, 0x01,
  0x01, 0xFB, 0x23, 0x20, 0x11, 0x00, 0x23, 0x22, 0x31,
  0x00, 0x23, 0x24, 0x41, 0x00, 0x23, 0x26, 0x51, 0x00,
  .......
```

El script de linker que lo coloca en la dirección correcta en la DRAM es este:

```
SECTIONS
{
  . = 0x100000;
  .program_data : {KEEP(*(program_data)) }  > ps7_ddr_0
  .......
```

5. Resultados

Cumplimos con los objetivos del proyecto. Escribimos un softcore que puede correr programas para el modo M de las especificaciones del ISA de RISC-V. Pudimos correr un sistema operativo de tiempo real y corrimos programas complejos sobre él como un servidor HTTP, un cliente DHCPv4 y un servidor Telnet.

5.1. Detalle breve de uso del softcore

El módulo que implementa el softcore es configurable. Se puede cambiar el tamaño de las cachés, así como el tamaño de los espacios de memoria.

El acceso a memoria es a través de un bus con dos *slaves* implementados: uno de ellos coloca los datos de la memoria en los BRAM de la placa Zybo y el otro lee de un bus AXI. Este último es el usado para correr los ejemplos más complejos de Zephyr.

5.2. Reporte de recursos y power

Esta información fue extraida de la herramienta que brinda *Xilinx Vivado*. La frecuencia de operación a la cual se sacaron estos valores fue de 80MHz para el *CPU* y 150 MHz para el bus *AXI*. El tamaño de la memoria que se utilizó es la que se puede ver en la figura 5.1.

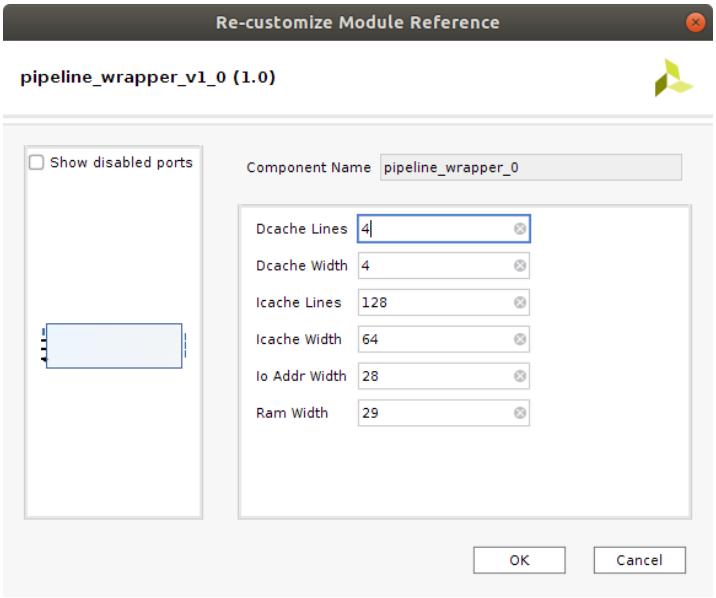

Figura 5.1.: Reporte de Recursos

5.2.1. Reporte de utilización

En la figura 5.2 se observan las proporciones ocupadas del total de recursos de la FPGA.

Resource	Utilization	Available	Utilization %
LUT	6072	17600	34.50
LUTRAM	188	6000	3.13
FF	5182	35200	14.72
BRAM	19	60	31.67
DSP	4	80	5.00
IO	21	100	21.00
MMCM	1	2	50.00
PLL	1	2	50.00

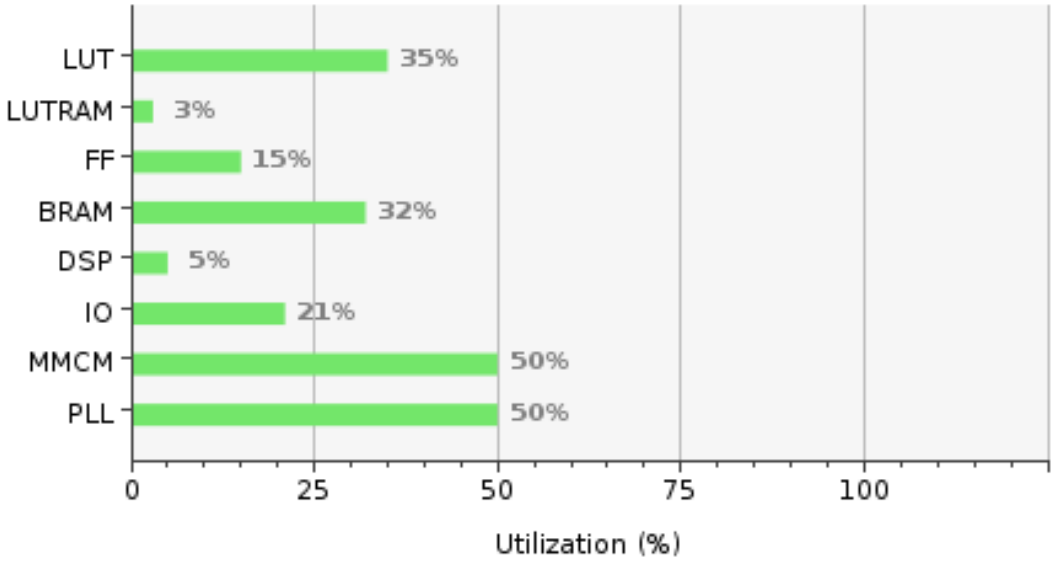

Figura 5.2.: Porcentaje de utilización

En la figura 5.3 se observa la utilización de recursos de la FPGA por parte de cada componente del diagramas en bloques de IP cores. Como es predecible, el IP *pipeline_wrapper_0* que contiene el softcore es el que mayores recursos consume. La caché de datos consume más recursos que la caché de instrucciones porque es más compleja. En las secciones 4.2 y 4.3 se describe la función de cada componente en mayor detalle.

Name	Slice LUTs (17600)	Block RAM Tile (60)	DSPs (80)	Slice Registers (35200)	F7 Muxes (8800)	F8 Muxes (4400)	Slice (4400)	LUT as Logic (17600)	LUT as Memory (6000)
riscv_unrn_wrapper	6072	19	4	5182	89	12	2012	5884	188
riscv_unrn_i (riscv_unrn)	6072	19	4		89	12	2012	5884	188
uart_module_wrapper_0 (riscv_unrn_uart_module_wrapper_0_0)	41	0	0		0	0	17	41	0
uart16550/cross_domain_level_0 (riscv_unrn_cross_domain_level_0_1)	0	0	0		0	0	1	0	0
uart16550/axi_uart16550_0 (riscv_unrn_axi_uart16550_0_0)	401	0	0		0	0	137	390	11
reset_mux (riscv_unrn_reset_mux_0)	1	0	0		0	0	1	1	0
processing_system7_0 (riscv_unrn_processing_system7_0_0)	0	0	0		0	0	0	0	0
plic_unit_0 (riscv_unrn_plic_unit_0_0)	99	0	0		0	0	33	99	0
pipeline_wrapper_0 (riscv_unrn_pipeline_wrapper_0_0)	3665	15	4		89	12	1139	3505	160
inst/i_pipeline/i_stage_ex/i_alu (ALU)	983	0	4		33	0	289	983	0
inst/i_pipeline/i_instruction_cache/i_instruction_cache_mem (instruction_cache_mem)	71	14	0		0	0	36	71	0
inst/i_pipeline/i_data_cache/i_data_cache_mem (data_cache_mem)	194	0	0		0	0	69	130	64
join_uart_tx_0 (riscv_unrn_join_uart_tx_0_0)	1	0	0		0	0	1	1	0
io_device_0 (riscv_unrn_io_device_0_0)	3	0	0		0	0	5	3	0
ethernet/mii_to_rmii_0 (riscv_unrn_mii_to_rmii_0_0)	50	0	0		0	0	30	46	4
ethernet/axi_ethernetlite_0 (riscv_unrn_axi_ethernetlite_0_0)	664	4	0		0	0	231	652	12
ethernet/PHY_clock (riscv_unrn_PHY_clock_0)	0	0	0		0	0	1	0	0
devices/mtime_device_wrapper_0 (riscv_unrn_mtime_device_wrapper_0_0)	40	0	0		0	0	33	40	0
devices/device_conexion_0 (riscv_unrn_device_conexion_0_0)	170	0	0		0	0	67	170	0
devices/UNRNRISCV_AXI_bridge_0 (riscv_unrn_UNRNRISCV_AXI_bridge_0_0)	214	0	0		0	0	83	214	0
data_mem_combined_0 (riscv_unrn_data_mem_combined_0_0)	279	0	0		0	0	177	279	0
clk_reset/proc_sys_reset_0 (riscv_unrn_proc_sys_reset_0_0)	25	0	0		0	0	14	24	1
clk_reset/not_gate_0 (riscv_unrn_not_gate_0_0)	1	0	0		0	0	1	1	0
clk_reset/cross_domain_level_1 (riscv_unrn_cross_domain_level_1_0)	0	0	0		0	0	1	0	0
clk_reset/cross_domain_level_0 (riscv_unrn_cross_domain_level_0_0)	0	0	0		0	0	1	0	0
clk_reset/clk_divider (riscv_unrn_clk_divider_0)	0	0	0		0	0	0	0	0
axi_interconnect_1 (riscv_unrn_axi_interconnect_1_0)	191	0	0		0	0	65	191	0
axi_interconnect_0 (riscv_unrn_axi_interconnect_0_0)	227	0	0		0	0	151	227	0

Figura 5.3.: Reporte de utilización del proyecto

5.2.2. Reporte de power

En los desarrollos en FPGA es necesario prestar cierta atención al consumo de energía del diseño. En la imagen a continuación se encuentra un reporte de la herramienta vivado sobre la potencia utilizada por el diseño realizado.

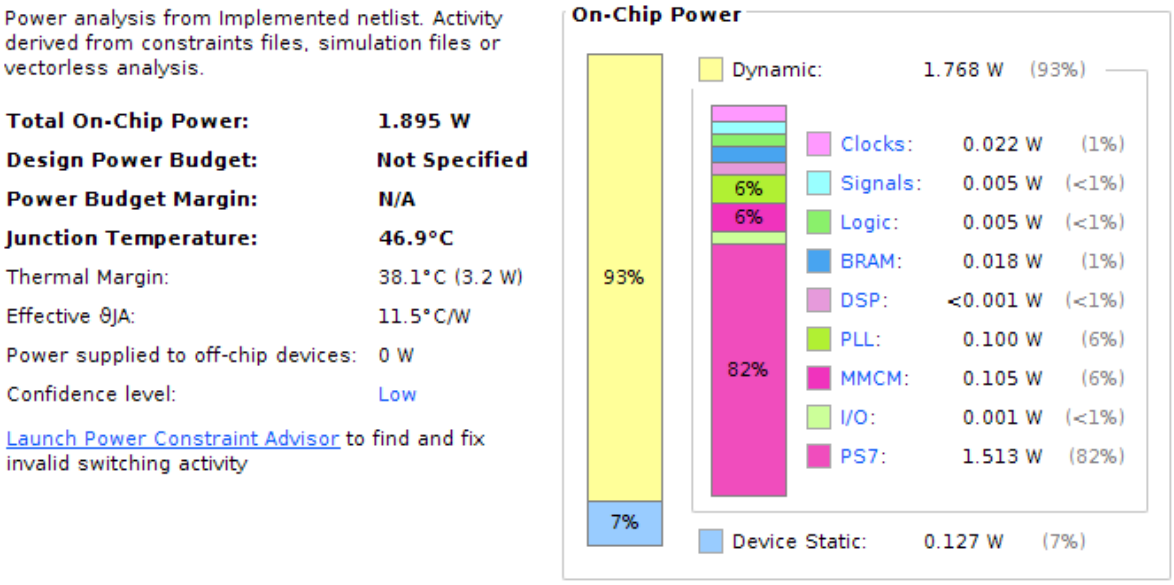

Figura 5.4.: Reporte de power

En este reporte, como era de esperarse, la parte del diseño que mas energía consume en standby es la PS del chip de la placa. Las partes del sistema siguen en consumo al core incorporado dentro del chip son las partes relacionadas con el clock las cuales seria el MMCM (Mixed-Mode Clock Manager), el PLL (Phase-Locked Loop) y el clock debido a que estas están en funcionamiento constantemente. Finalmente, se encuentran las demás partes del sistema en la cual destaca la BRAM que es la que mantiene la información del sistema.

5.3. Demostraciones

Corrimos varios ejemplos que vienen con el sistema operativo Zephyr. Para cada ejemplo verificamos que la funcionalidad descripta en el archivo `README` funcionara bien.

5.3.1. Ping pong de caracteres por UART: `subsys/console/echo`

Este ejemplo corre una aplicación que lee caracteres por `stdin` y los imprime por `stdout`. En ella se verifica que la y el PLIC funcionen correctamente.

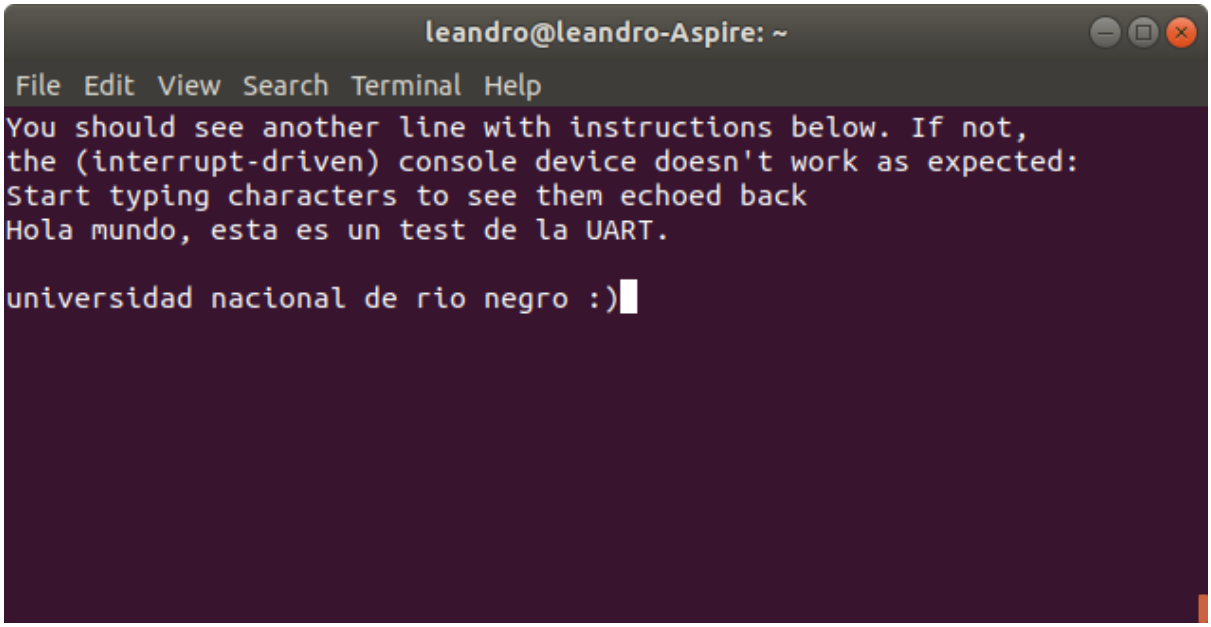

Figura 5.5.: Test UART

5.3.2. Inspección de tramas Ethernet: `net/sockets/packet`

Este ejemplo envía tramas por la placa Ethernet, e imprime la longitud de todas las tramas que recibe. Este ejemplo sirve para probar que la configuración del *AXI EthernetLite* sea correcta, que su driver funcione y que el PLIC funcione correctamente.

En la figura 5.7 se muestran los paquetes que recibe la laptop a la que está conectada la placa Zybo y en la figura 5.6 se ve un registro de los paquetes que se envían y se reciben.

5.3.3. Cliente DHCPv4: `net/dhcpv4_client`

Este ejemplo corre un cliente de DHCPv4 e imprime la dirección IP que recibe del servidor. Este ejemplo demuestra que el sistema operativo Zephyr junto con el driver de *AXI EthernetLite* están funcionando bien.

En la figura 5.8 se observa que Zephyr solicitó una IP a un servidor DHCPv4 y recibió `192.168.1.104`. En la figura 5.9se puede ver que Zephyr responde a *pings* en la IP que recibió.

5.3.4. Servidor HTTP: `net/sockets/dumb_http_server`

Este ejemplo corre una aplicación que responde solicitudes HTTP en el puerto 8080. Este ejemplo muestra que la capa TCP/IP de Zephyr se integra correctamente con el driver de *AXI EthernetLite*.

En la figura 5.10 se ve que el servidor HTTP recibe solicitudes y en la figura 5.11 se observa el documento HTML que envía el servidor.

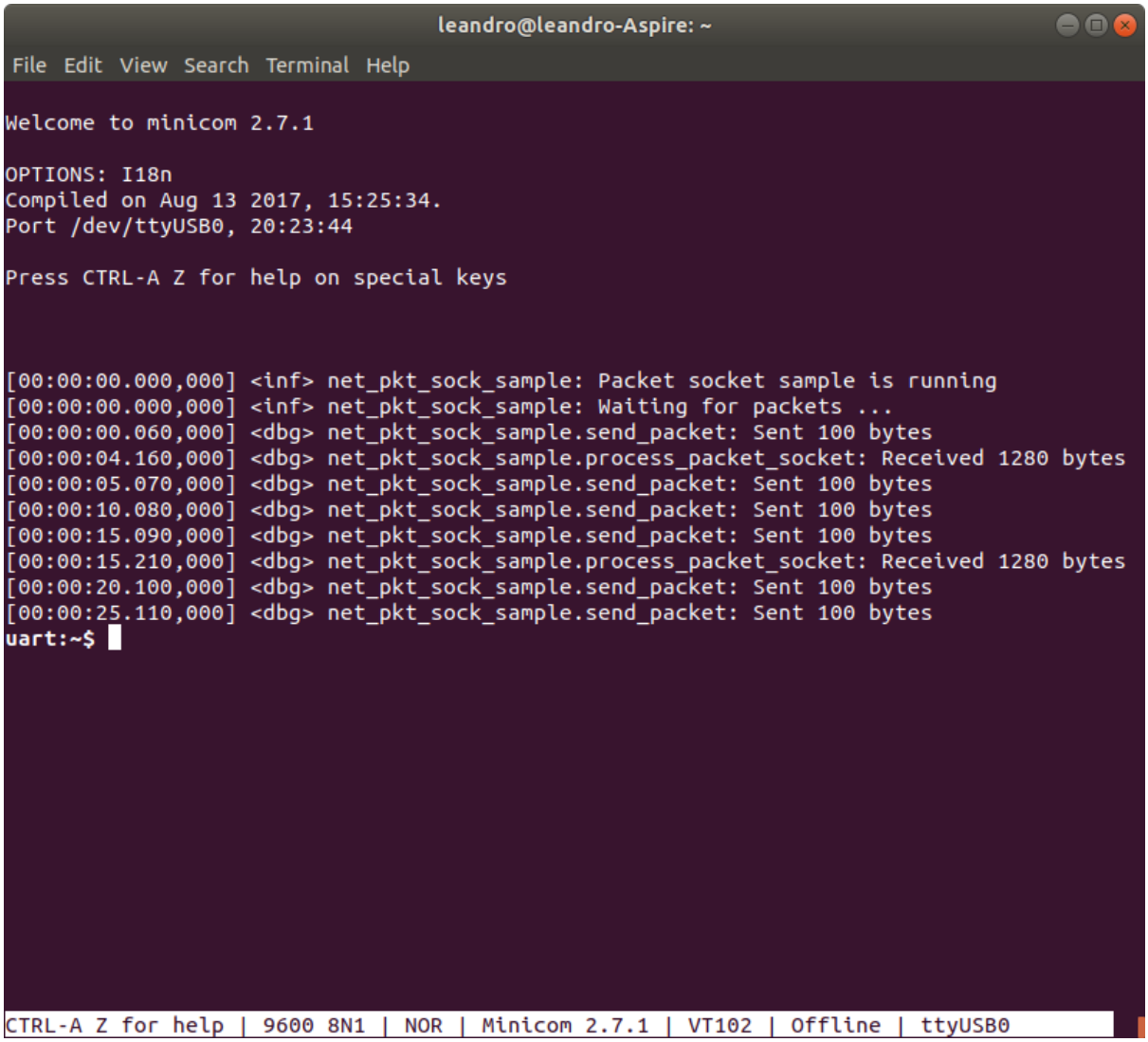

Figura 5.6.: Test Packet en consola

5.3.5. Servidor Telnet: `net/telnet`

Este ejemplo corre un servidor Telnet. Un cliente puede conectarse y enviar comandos de *shell* por él. En la figura 5.13 se observa que un cliente Telnet puede enviar comandos y recibir la respuesta por la red. En la figura 5.12 se observa que también se puede enviar comandos por la UART

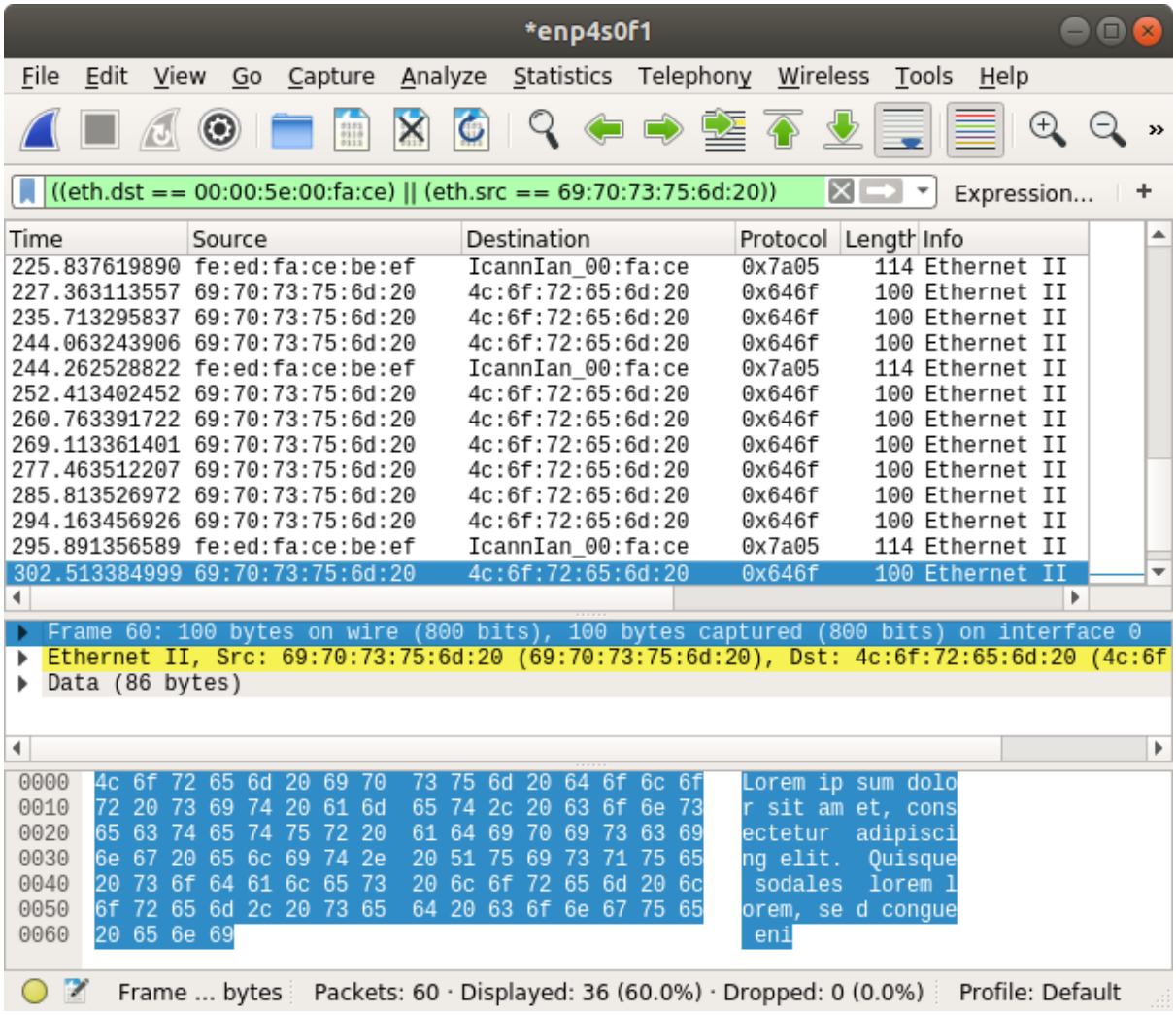

Figura 5.7.: Test Packet en Wireshark

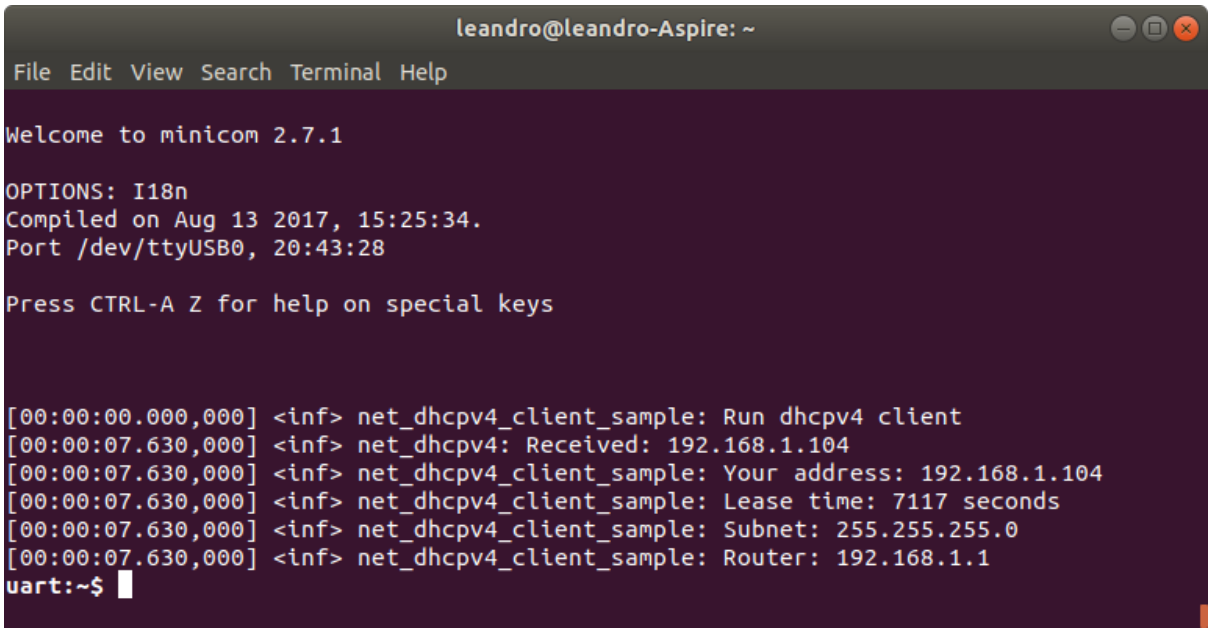

Figura 5.8.: Test cliente DHCPv4

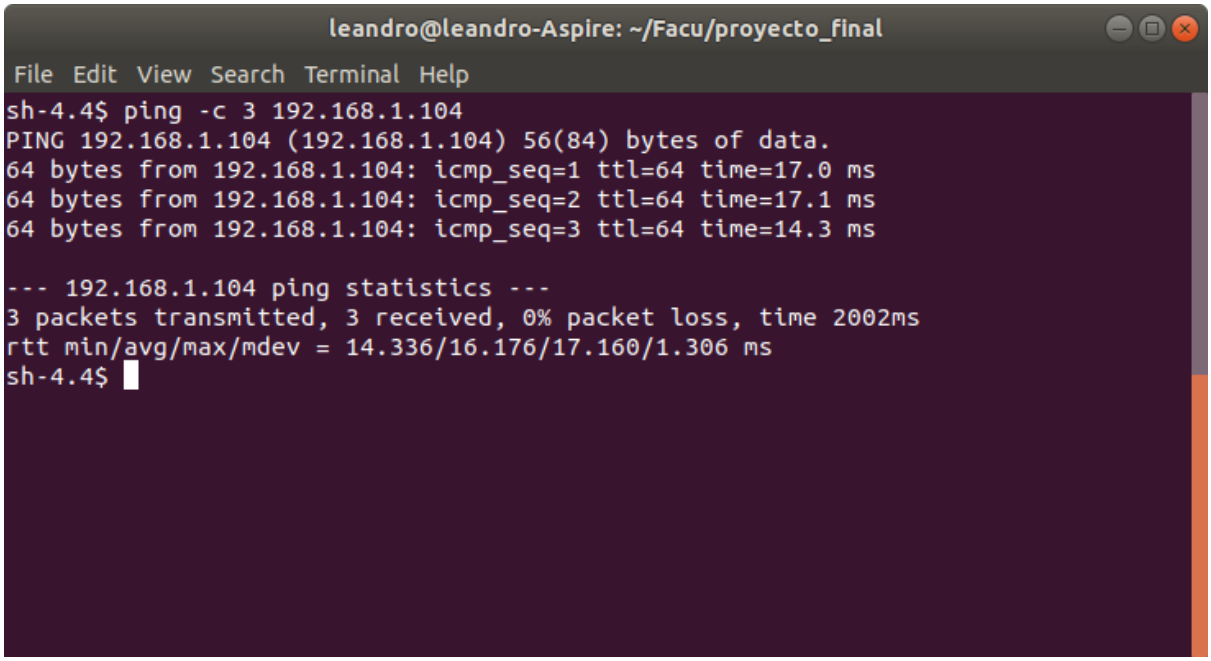

Figura 5.9.: Test ping a cliente DHCPv4

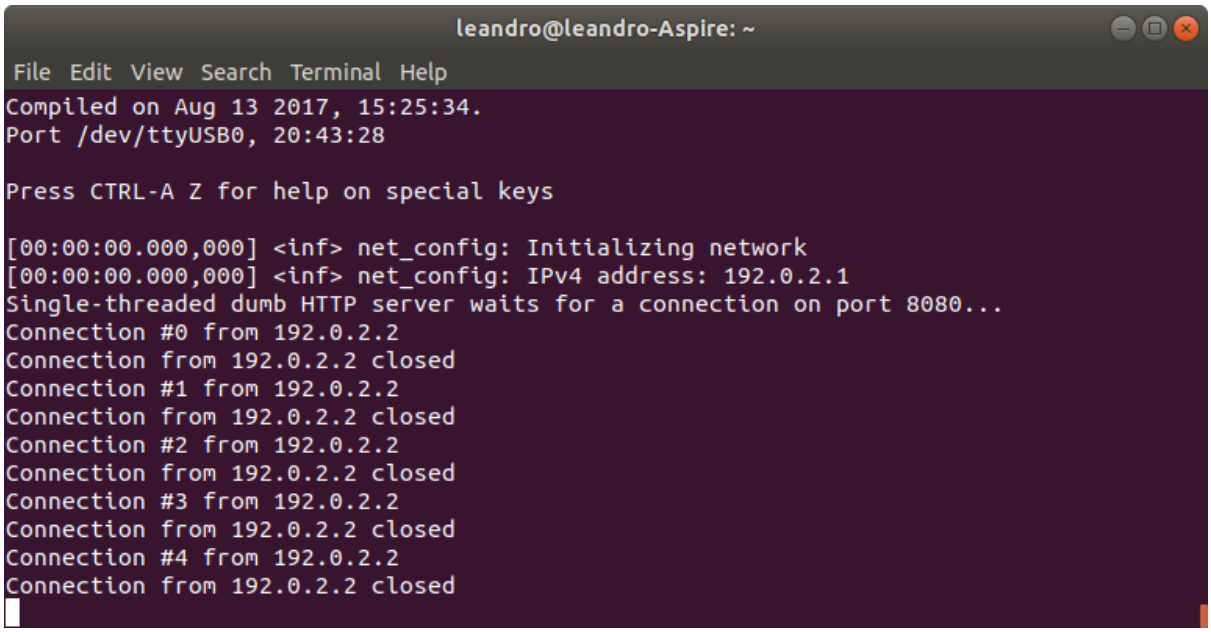

Figura 5.10.: Test de servidor HTTP: registro por consola

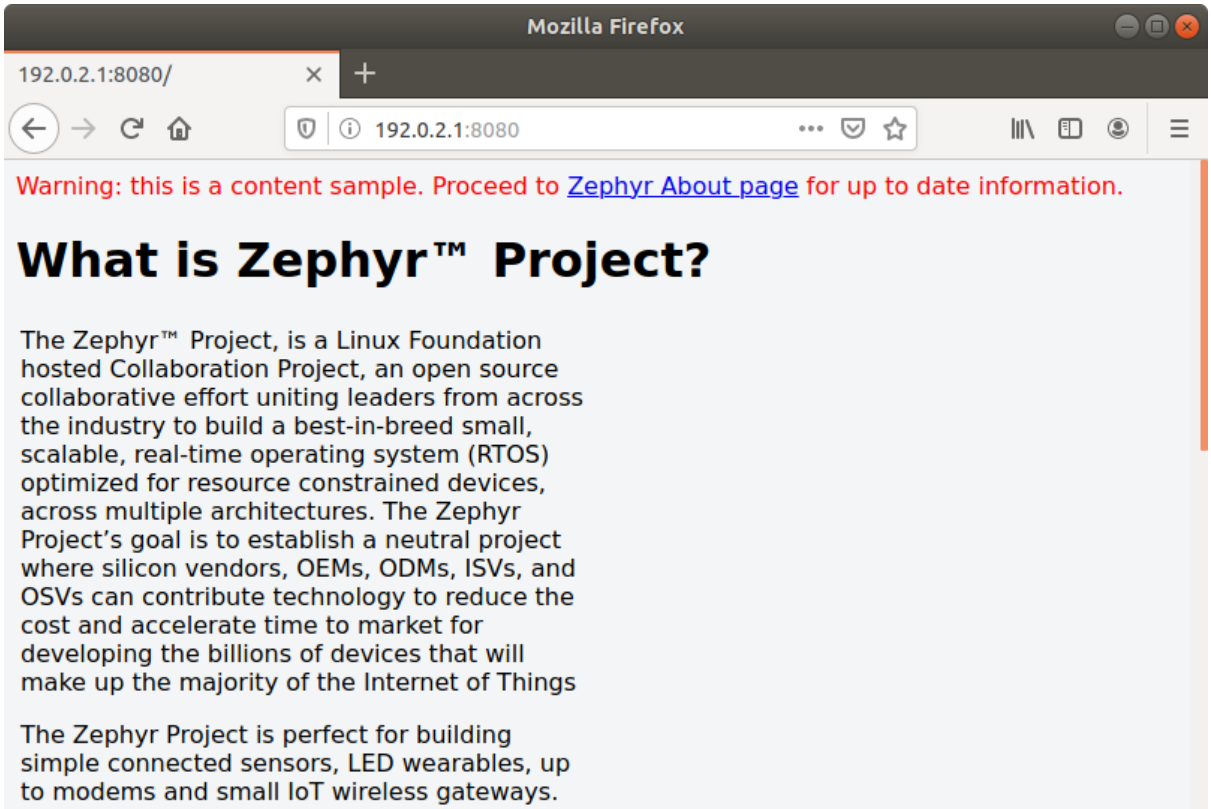

Figura 5.11.: Test de servidor HTTP con Firefox

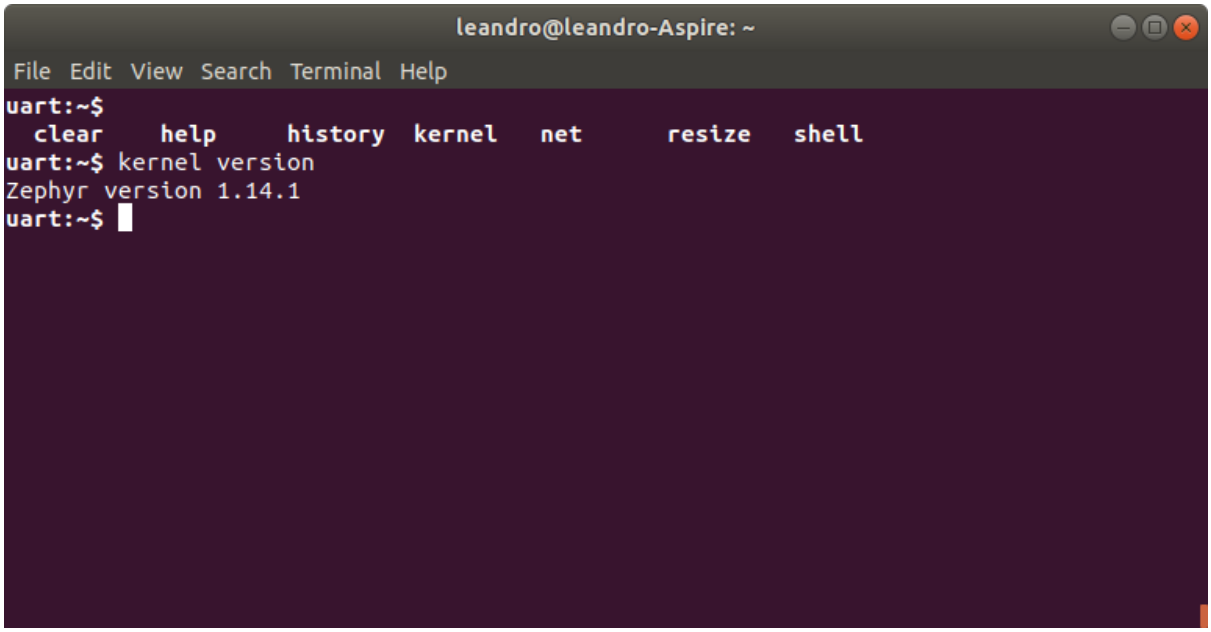

Figura 5.12.: Test de servidor Telnet: registro por consola

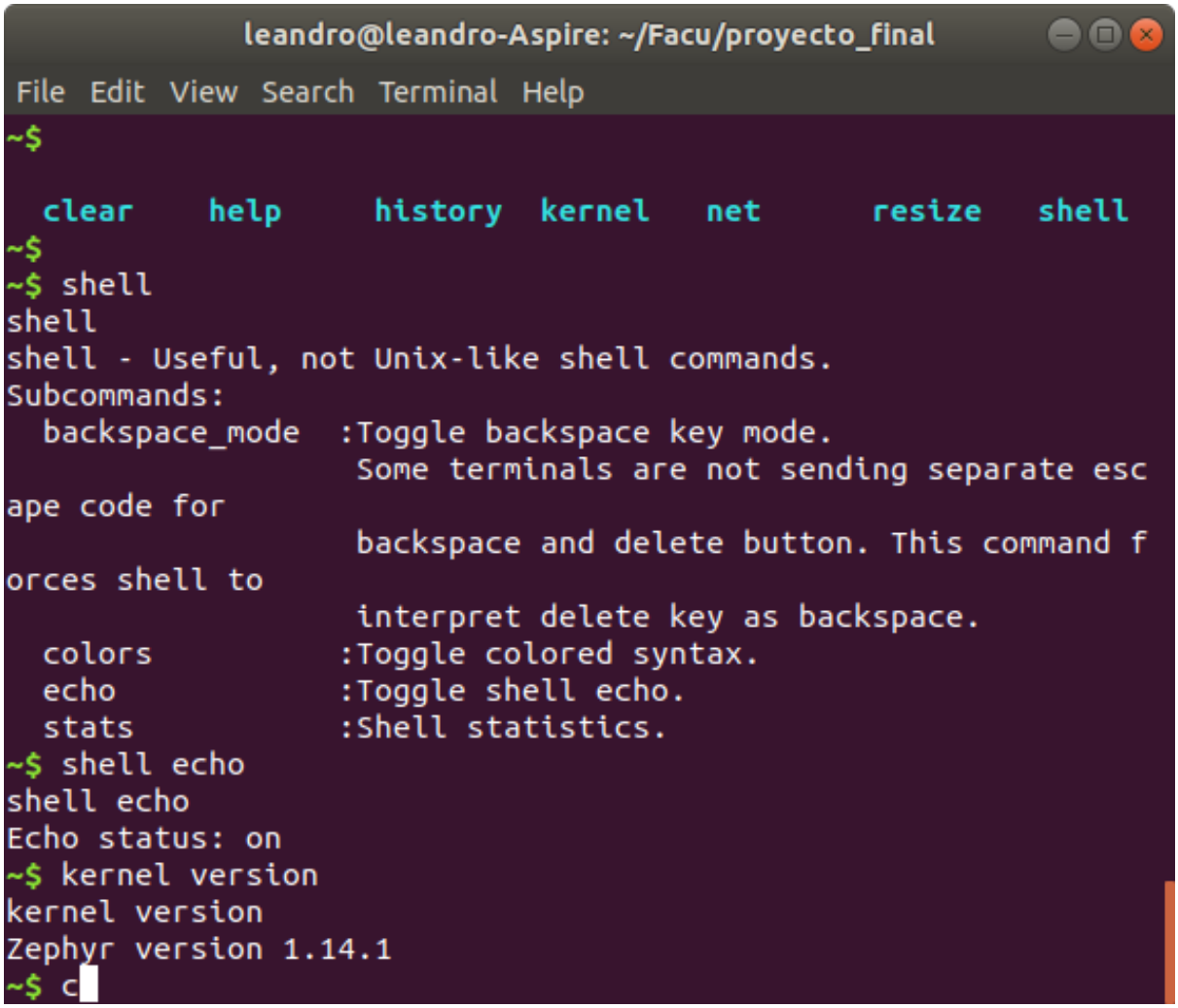

Figura 5.13.: Test de servidor Telnet: cliente Telnet

6. Conclusiones

Este trabajo nos permitió aplicar lo aprendido durante el curso de la carrera como también aprender nuevos contenidos. Tuvimos que acompañar el proyecto desde que fue una idea hasta su concreción, pasando por las etapas de planeamiento, diseño, implementación, testeo y documentación. Durante la ejecución del proyecto tuvimos que aprender a usar herramientas de la industria, como Vivado, el SDK de Xilinx, Verilator. También tuvimos que trabajar en el *kernel* de un RTOS (Sistema operativo de tiempo real) y comprender cómo se porta a una plataforma nueva. Otro aspecto importante fue la planificación que se fue llevando durante todo el transcurso del proyecto, dividiendo tareas para cada uno de los participantes como también poniendo a discución diversos caminos de diseño llegando hasta un acuerdo en común.

Hubo varias dificultades durante el desarrollo del proyecto. Una de ellas fue entender cómo arreglar los problemas de *timing* que ocurrían. Esto requirió entender bien cómo se implementa un diseño de hardware en una FPGA y qué patrones de diseño son problemáticos. También requirió aprender cómo funcionan las herramientas para poder generar reportes que expliquen cuales son los problemas y poder comprenderlos.

Otra dificultad fue arreglar los problemas debidos a *bugs* en el diseño. Para este problema fue de mucha utilidad la herramienta verilator y los tests de RISC-V. Pero hay un conjunto de problemas que no eran depurables a través de verilator, estos se pueden resumir en los que están relacionados con los dispositivos de IO y su integración con Zephyr. Depurar estos problemas fue costoso en términos de tiempo porque se requería reimplementar el diseño en hardware en caso de arreglar algo en el HDL o adivinar dónde está el problema a través de impresiones en pantalla. Una opción se que planteo fue diseñar un sistema de debbuging propio, el cual consistia en tener un historial de cuales fueron las instrucciones que se fueron ejecutando como también se queria tener la libertad de poder parar la ejecución del programa para poder leer ciertos valores a través de un bus diseñado por nosotros. El diseño tenía su complejidad pero como los tiempos que habíamos planteado ya eran cortos, se tomo la decisión de no diseñarlo y tomar otro camino.

El problema con las impresiones en pantalla es que consisten en código que debe ser agregado a las funciones y esto puede cambiar el comportamiento de las funciones depuradas.

También los problemas con el dispositivo de timer fueron dificiles de resolver. Esto fue así porque los problemas surgían cuando ocurría una interrupción del dispositivo de timer justo cuando se ejecutaba una instrucción en particular. Para poder depurar estos errores fue crucial la herramienta verilator.

Una buena decisión que se tomó al comienzo del proyecto fue agregar un sistema de CI para correr todos los tests cada vez que se hacía un cambio. Esto nos permitió

detectar errores rápidamente y mantener el proyecto funcionando a medida que se desarrollaba. Otra buena decisión relacionada al sistema de integración continua fue tomarse el tiempo para diseñar tests que prueben partes del diseño aisladas del resto del sistema o con modelos simulados. Esto fue importante para poder depurar componentes complejos como el PLIC, la caché de datos, la caché de instrucciones o la ALU. Otra buena decisión fue definir un manual de stylo, lo cual nos permitió automatizar ciertas tareas que si uno las tuviese que hacer manualmente, sería demasiado tedioso.

Durante las pruebas en el hardware real no tuvimos ningún problema con los componentes que fueron depurados exhaustivamente con tests en aislación. Todas los problemas estuvieron en las integraciones entre componentes o en la conexión con el mundo exterior.

Dentro de las mejoras posibles para el diseño se pueden enumerar las siguientes:

1. Implementar un *barrel shifter* para la ALU.
2. Mejorar la documentación.
3. Portar el diseño a FPGA de otros fabricantes.
4. Agregar *prefetching* a las cachés.
5. Ensanchar los buses de acceso a memoria para que las cachés sean más rápidas.
6. Agregar una caché L2 entre la caché de datos, caché de instrucciones y la memoria principal.
7. Completar la implementación de los CSR.
8. Hacerlo compatible con muchos cores corriendo en la misma memoria.
9. Agregar un módulo de *debugging* compatible con la especificación de RISC-V.
10. Agregar un predictor de saltos más inteligente.
11. Implementar una caché asociativa de N vías.
12. Implementar los otros modos para poder correr software más complejo.
13. Reducir el consumo de energía.

Glosario

IP core Módulo de hardware empaquetado. 1, 30–33, 42, 60, 72

AXI EthernetLite IP core que implementa una MAC Ethernet. 63

forwarding Realimentación entre etapas del pipeline. 27

ARM Arquitectura de computadoras utilizada en sistemas embebidos. 32, 57, 58

bitstream Diseño de hardware listo para programar en una FPGA. 23

caché de datos Es la caché que ahorra accesos a la memoria para datos. 21, 33, 45, 46, 48, 49, 71

caché de instrucciones Es la caché que ahorra accesos a la memoria para instrucciones. 21, 25, 33, 48, 71

commit Es un conjunto de cambios en diferentes archivos que se agrupan para el propósito de control de versiones. 21, 22

diagramas en bloques de IP core Interconexión de IP core de forma gráfica. 30, 60

dispositivo de GPIO Es un dispositivo que implementa accesos de IO de propósito general. 21

dispositivo de timer Es un dispositivo que permite observar el tiempo y configurar una interrupción interrupción de timer. 21, 70

docker Servicio que permite crear contenedores aislados en un sistema Linux. 23

espacio de direcciones Rango de la memoria que se asigna a un propósito especifico, por ejemplo: acceder a un dispositivo en particular. 49

etapa de commit etapa del pipeline, en ella se esperan las operaciones de memoria y se escriben los registros. 25–27, 49

etapa de ejecución etapa del pipeline, en ella se encuentra la ALU. 21, 26, 27

etapa de instruction decode etapa del pipeline, se encarga de la decodificación de instrucciones. 25

etapa de instruction fetch etapa del pipeline. 25

etapa de issue etapa del pipeline. 26, 27

etapa de Memoria etapa del pipeline, en ella se accede a la caché de datos o a los dispositivos.. 27

etapa de generación de Program Counter etapa del pipeline, en ella se calcula el siguiente PC y se accede a la caché de instrucciones. 21, 25

Git Sistema de control de versiones distribuido. 20, 73

GitLab Servicio de hosting de repositorios Git. 20, 22, 23

interrupción Un pedido de atención externo al procesador. 72

linea de caché Es la unidad mínima de acceso a memoria por parte de una caché. 43, 44

MAC Ethernet Componente que controla el acceso al medio físico de comunicaciones de redes Ethernet. 1, 42

Microblaze Softcore diseñado por Xilinx que implementa una arquitectura propia. 36

módulo de UART módulo que implementa una UART que solo puede transmitir a una velocidad fijada durante la síntesis. 21

PHY Ethernet Circuito integrado que conecta el medio físico de comunicaciones con el dispositivo MAC Ethernet. 32, 42

proximidad espacial Proximidad temporal entre accessos a direcciones cercanas de memoria. 14, 43

proximidad temporal Proximidad temporal entre accesos a una misma dirección de memoria. 14, 43

puente entre memoria e IO módulo que conecta la memoria con los dispositivos de IO y el procesador tal que su acceso por éste sea transparente. 3, 21, 27, 33, 34, 44, 45, 49, 50

pytest Es una librería Python diseñada para hacer facil el testeo de software.. 23

Python Es un lenguaje de programación de alto nivel muy popular con muchas librerías disposibles. 23, 73

RAM Es una memoria de acceso aleatorio de escritura y lectura.. 13, 49

repositorio Conjunto de carpetas y archivos administrados con un sistema de control de versiones. 20, 22

RISC-V Es el ISA implementado en este proyecto. 1, 8, 9, 11, 14, 21, 23, 56, 57, 59, 70, 71

ROM Es una memoria de solamente escritura.. 13

SystemVerilog Es un lenguaje de descripción de hardware con más funcionalidades que Verilog. 21

toolchain Software necesario para crear binarios que corran sobre una plataforma.. 23

UART 16550 Componente que implementa una UART programable similar al circuito integrado de National Instruments con el mismo nombre. 1, 3, 31, 32, 42, 62

unidad de stall unidad del pipeline encargada de frenar las etapas cuando una etapa no terminó. 27

verilator Es una herramienta que permite compilar modelos en Verilog o SystemVerilog a modelos en C++. 2, 21–23, 70

Verilog Es un lenguaje de descripción de hardware. 33

Vivado Es la herramienta de sintesis e implementación utilizada para FPGA de Xilinx. 20, 21, 23, 24, 30, 32, 70

Wishbone Bus de interconexión.. 18

Zephyr Sistema operativo de tiempo real utilizado en este proyecto.. 1, 3, 31, 56, 62, 63, 70

Zybo Placa de desarrollo para FPGA Zynq diseñada por Digilent. 36

Bibliografía

[1] Stephen P Morse. The Intel 8086 Chip and the Future of Microprocessor Design. *IEEE Computer*, 50(4):8–9, 2017.

[2] Editors Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2, May 2017.

[3] Editors Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10, May 2017.

Siglas

AHB Advanced High-performance Bus. 18

ALU Arithmetic Logic Unit: Unidad Aritmética Lógica. 6, 7, 11, 21, 26, 71, 72

AMBA Advanced Microcontroller Bus Architecture: Arquitectura avanzada de bus para microcontroladores. 18

APB Advanced Peripheral Bus. 18

ASB Advanced System Bus. 18

ASIC Application Specific Integrated Circuits: Circuitos integrados de propósito especifico. 8

AXI Advanced eXtensible Interface: Bus de interconexión entre varios masters y varios slaves. 18, 29–33, 36, 37, 42, 57

Benchmark . 6

BSP Board Support Package. 57

CI Continuous Integration: Integración Continua. 20, 23, 36, 70

CISC Complex Instruction Set Computer. 7

CPU Central processing unit: Unidad central de procesamiento. 6, 33, 49, 50

CSR Control Status Register. 26, 71

DMA Direct memory access: Acceso directo a memoria. 6

DRAM Dynamic Random Access Memory. 17

DSP Digital Signal Processor. 26

ELF Executable and Linkable Format. 58

FPGA Field Programmable Gate Arrays: Arreglos de compuertas programables. 1–4, 8, 30, 34, 43, 47, 48, 57, 60, 62, 70, 71, 74

Fuera de orden Fuera de orden: Out-of-order. 12

GPR General Purpose Register. 26

GPU Graphics processing unit: Unidad de procesamiento gráfico. 6

Hart Hardware thread: Hilo de ejecución en hardware. 11

HDL Hardware description language: lenguaje de descripción de hardware. 4, 70

HTTP Hypertext Transport Protocol. 63

HW Hardware. 6

IO Input Output. 3, 21, 27, 30, 33, 34, 39, 40, 44, 45, 49, 50, 70, 72, 73, 77
IP Intellectual property. 1, 30–33, 42, 60, 72
ISA Instruction Set Architecture. 1, 7, 8, 14, 26

MII Media Independent Interface. 31, 42
MIO Multiplexed IO. 34
MMCM Mixed-Mode Clock Manager. 62
Multiciclo Multiciclo: Multi-cycle. 12

PC Program Counter. 21, 24–27, 72
PL Programmable Logic. 30, 57
PLI Programming Language Interface. 22
PLIC Platform Layer Interrupt Controller. 21, 32, 33, 50, 56, 57, 62, 63, 71
PLL Phase-Locked Loop. 62
PMP Phsysical Memory Protection: Protección física de memoria. 11
PS Processing System. 3, 30, 32, 33, 35, 57

RAW Read After Write. 12
RISC Reduced Instruction Set Computer. 7, 8
RMII Reduced Media Independent Interface. 42
RTOS Sistema operativo de tiempo real. 1, 4, 56, 70

SDK Software Development Kit. 57, 58, 70
Segmentación Segmentación: Pipeline. 12
SISC Simple Instruction Set Computer. 7
SRAM Static Random Access Memory. 17
Superescalar Superescalar: Superscalar. 12
SW Software. 6, 8

TCL Tool Command Language. 24

UART Universal Asynchronous Receiver Transmiter. 3, 21, 22, 31, 42, 56, 57, 62, 64, 73
UC Unidad de Control. 7
Uniciclo Uniciclo: Unicycle. 12

WAR Write After Read. 12
WAW Write after Write. 12

A. Anexo

A.1. Diagrama del Softcore

En la figura A.1 se muestra brevemente el diseño que se implemento para el softcore, detallando caminos de datos e instrucciones, señales de control, señales del forwaring unit como también las de stall unit.

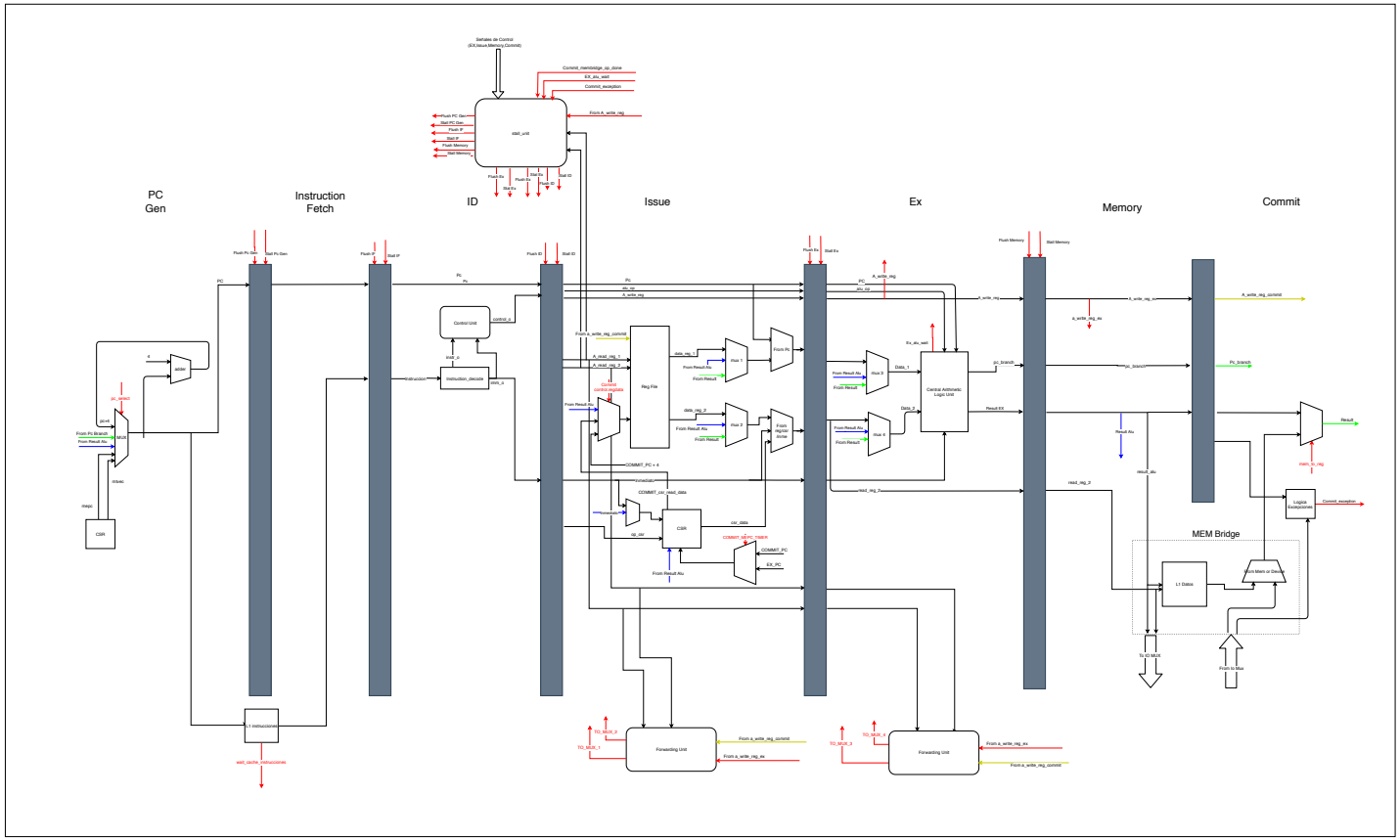

Figura A.1.: Diagrama del softcore