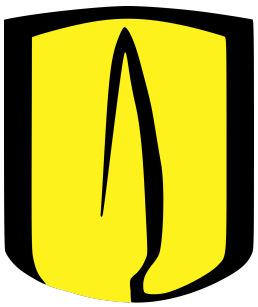

Diseño, implementación y validación de un núcleo de procesador basado en el conjunto de instrucción RISC-V

Por:

Ing. Nicolás Rocha Pacheco

Asesores:

Fredy Enrique Segura Quijano, Ph. D.

Fernando A. Escobar, Ph. D.

Universidad de Los Andes

Facultad de Ingeniería

Departamento de Ingeniería Eléctrica y Electrónica

Bogotá D.C., Colombia

De parvis grandis acervus erit

Agradecimientos

Quisiera agradecer a mis padres, Jaime y Jasmine. Ellos hicieron todo lo que estuvo a su alcance para darme siempre lo mejor, especialmente en lo relacionado a mi educación. Gracias a ellos puedo decir que hablo inglés, que me gradué de un colegio que me permitió aspirar a ingresar a la mejor universidad del país, y que me gradué de esta con dos títulos profesionales y una opción académica sobre aquello que me apasionaba. El impulso que ellos me dieron me permitió aspirar a este nivel de formación profesional y académica, que va más allá de lo aprendido en la Universidad con los estudios de pregrado. Así como mis otros títulos, este es un esfuerzo compartido entre los tres; sin duda alguna ustedes también son artífices de esto. Los amo y gracias por permitirme llegar hasta acá.

Quisiera agradecer a Andrea Carolina Domínguez Gámez, cuyo amor se tradujo en un apoyo que solo pocas personas en el mundo pueden dar. Ella siempre estuvo pendiente de mí, ya sea para consolarme cuando el núcleo no funcionaba, así como para celebrar cuando se aparecían los números de la secuencia de Fibonacci en la pantalla. Durante el 2020 que comprendió el desarrollo de la tesis tu apoyo fue algo que me permitió seguir avanzando y conseguir terminar este proyecto. No sé que habría hecho sin ti, aunque es una cuestión que no me interesa resolver. Te amo Andrea y agradezco el apoyo que me diste infinitamente.

Quisiera agradecer a Fredy Segura, mi asesor, por permitirme desarrollar un proyecto con el cual soné desde antes de iniciar mis estudios profesionales. Predecir el futuro es una tarea imposible, pero espero que este proyecto le permita seguir ahondando en esta área de la Ingeniería tan emocionante no solo a él, sino a su grupo de estudiantes. Espero colaborar con el grupo en un futuro y ojalá, motivar a alguien a conocer esta área, así como Fredy me motivó en su momento. Muchas gracias Fredy por todo lo que colaboramos a lo largo de estos últimos años, y por lo pronto, le deseo lo mejor a usted y su familia. También quisiera agradecer a Fernando Escobar, por compartir su experiencia con nosotros y permitir que este proyecto obtuviera un enfoque que nos permitió aumentar la calidad del mismo. Fernando realizó aportes muy valiosos que, sin duda, alteraron el curso del proyecto para bien. Gracias por compartir su experiencia con nosotros.

Mis agradecimientos van a la Universidad de Los Andes y al Departamento de Ingeniería Eléctrica y Electrónica por darme la oportunidad de alcanzar un nuevo título. Quisiera agradecer a Roberto Bustamante Miller, Profesor del Departamento, por darme la oportunidad laboral que me permitió finalizar mis estudios de maestría. En los últimos años la Universidad de Los Andes ha sido mi hogar y el lugar donde he vivido grandes experiencias. Gracias por permitirme crecer como persona y como profesional, además de darme la oportunidad de conocer a grandes personas dispuestas a ayudarme para hacerme mejor. Gracias a todas aquellas personas que se cruzaron en mi vida universitaria, en especial a Paola por haber cambiado mi forma de ver el mundo y por hacerme entender que uno se mejora día a día con cada pensamiento nuevo. Sin duda alguna dejo la Universidad de Los Andes como aquel hijo que deja su hogar en busca de nuevos horizontes.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En 1991 un estudiante de ciencias de la computación de la Universidad de Helsinki llamado Linus Torvalds aprovechaba su tiempo libre para desarrollar un sistema operativo que le permitiera aprovechar el procesador Intel 80386 que tenía en su computador personal. Para 1994, cuando Torvalds liberó la versión 1.0 de su sistema operativo, Richard Stallman y la Free Software Foundation (FSF) trataban de crear un sistema operativo de código abierto similar a UNIX. Torvalds se enfocó en desarrollar el kernel de su sistema operativo primero y luego las utilidades mientras que Stallman y la FSF desarrollaron primero las utilidades, pensando en desarrollar el kernel después. Esta coincidencia permitió que naciera el sistema operativo GNU/Linux al sumar el kernel de Torvalds y las utilidades de Stallman y la FSF. Una coincidencia que cambió la historia de las ciencias de la computación.

Para el 2020, el sistema operativo Linux ocupaba algo más del $1{,}8\,\%$ de la cuota de mercado de los computadores de escritorio, un valor lejano al $83\,\%$ que ocupa el sistema operativo Windows. Sin embargo, Linux tiene una cuota de mercado considerable para los supercomputadores del mundo, una cuota del $100\,\%$. Otra cuota de mercado donde Linux domina sobre otros sistemas operativos son los servidores de los 25 sitios web más visitados del mundo. De esta lista, solo dos lugares no utilizan Linux como sistema operativo de sus servidores. Continuando con las cuotas de mercado de los servidores, el $93{,}5\,\%$ del primer millón de servidores del mundo utiliza Linux. En otras palabras, de ese millón de servidores, algo más de 9 de cada 10 utilizan Linux.

A la cuestión de cómo llegó Linux a ser tan ampliamente usado en el el campo del Internet y la computación de alto desempeño, la respuesta es que Linux es un sistema operativo ligero, robusto y gratuito, salvo por algunas distribuciones que cobran por el soporte al usuario. A su vez, esta respuesta lleva a la pregunta de cómo llegó Linux a ser un sistema operativo con estas cualidades; cómo se puede desarrollar un sistema operativo robusto, lo cual implica una buena cantidad de tiempo de desarrollo, sin llegar a cobrar un solo dólar por su uso. En este caso la respuesta se debe a la comunidad. Si bien Linus Torvalds fue quien inició el desarrollo, fue la comunidad de desarrolladores a nivel mundial quienes llevaron a este sistema operativo a ser un referente para los sistemas operativos. De igual modo, Linux es el mejor ejemplo de cómo la comunidad puede desarrollar herramientas de forma colaborativa para su propio beneficio. Sin embargo, el caso de Linux no es único y muchos más ejemplos pueden ser extraídos de los anales de la historia de las ciencias de la computación.

Un sistema operativo como Linux hace parte de lo que se conoce como el software de un computador, un concepto abstracto e intangible. Por otra parte, un computador tiene lo que se conoce como hardware, donde los conceptos etéreos de la ciencia de la computación se convierten en señales físicas medibles y observables. El principal componente donde ocurre la interfaz entre el software y el hardware es el procesador. Allí, una serie de señales físicas llamadas bits, que solo pueden tomar dos posibles valores, codifican las instrucciones que representan los deseos de los programadores. Programas complejos como sistemas operativos se traducen en miles de números binarios que pueden ser representados por un conjunto de señales eléctricas. En este sentido, el procesador se convierte en una máquina que presenta un comportamiento determinado por el número que le sea ingresado.

Vale la pena considerar que, en la actualidad, los procesadores son de 32 o de 64 bits, por lo que existen respectivamente 2^{32} y 2^{64} posibles combinaciones de números que pueden ser ingresados a un procesador. Para evitar diseñar máquinas que deban contemplar millones de escenarios de operación diferentes se han especificado lo que se denomina conjunto de instrucción. Un conjunto de instrucción es el vocabulario que entiende un procesador, es decir, un conjunto de números que representan una serie de instrucciones que permiten ejecutar un sinfín de programas. Dependiendo de la filosofía de diseño, un conjunto de instrucción puede tener desde unas decenas hasta miles de instrucciones diferentes. Y es precisamente el número de instrucciones, además de su organización, lo que determina algunos aspectos de los procesadores como lo son su desempeño y su consumo energético.

Dependiendo del número de instrucciones contempladas en un conjunto de instrucción, estos pueden ser divididos en dos categorías: CISC (Computación de Conjunto de Instrucción Complejo, *Complex Instruction Set Computing*) y RISC (Computación de Conjunto de Instrucción Reducido, *Reduced Instruction Set Computing*). La primera categoría hace referencia a conjuntos de instrucción que poseen miles de instrucciones capaces de realizar tareas específicas dentro del procesador. El objetivo de estas instrucciones tan específicas es reducir el tamaño que alberga el código en la memoria y hacer más fácil el proceso de programación. Por otra parte, la segunda categoría contiene conjuntos de instrucción con un número reducido de instrucciones de propósito general, con el fin de que estas puedan ser combinadas con el fin de realizar las tareas complejas que se requieren en la ejecución de programas. Al tener un número limitado de instrucciones se busca que los procesadores sean menos complejos y más eficientes en términos energéticos.

Las diferencias entre los conjuntos de instrucción CISC y RISC pueden ser evidenciados al discriminar las diferentes clases de dispositivos electrónicos en base al tipo de conjunto de instrucción que utilicen. Por una parte, en la mayoría de computadores portátiles, de escritorio y servidores se suelen usar conjunto de instrucción CISC, gracias a que estos suelen ofrecer mayor desempeño que su contraparte reducida. Por el contrario, en dispositivos móviles como celulares y tabletas se suelen usar conjuntos de instrucción RISC, gracias a su eficiencia energética que favorece el uso de baterías. A pesar de la marcada diferencia en los campos de aplicación, los conjuntos de instrucción RISC están ganando terreno sobre los CISC debido a fenómenos físicos que determinan los límites de los procesadores.

Uno de los métodos que permiten desarrollar procesadores más rápidos y eficientes año a año es la reducción en la tecnología de fabricación de los semiconductores. Esta reducción consiste en hacer los transistores que componen un procesador más pequeños con el fin de que conmuten más rápido y consuman menos energía. La idea de que los transistores conmuten más rápido busca que los procesadores puedan operar a frecuencias de reloj más altas, lo que se traduce en que sean más rápidos. Por otra parte, la reducción del consumo energético pretende que se genere menos calor durante el uso del procesador favoreciendo el rendimiento del

mismo. Sin embargo, esta reducción tiene un límite y si bien aún se debate si se ha alcanzado ese límite, es una realidad que tarde o temprano tendrá que ser enfrentada. Una vez se alcance el límite de la reducción, no será posible extraer mas desempeño a partir de este método.

Retomando la idea de la conmutación de los transistores, intuitivamente se podría pensar que esta podría aumentarse arbitrariamente para hacer los procesadores más rápidos. De hecho, este era otro método para extraer desempeño de los procesadores desde la década de los 80s hasta mediado de los 2000. Vale la pena notar que el aumento de la frecuencia de reloj causa que los procesadores consuman más energía y por lo tanto, emitan más calor. En este sentido, a medida que se aumenta la frecuencia de los procesadores la capacidad de disipación de calor debe ser aumentada. Allí es donde se encuentra un límite ya que la capacidad de disipación de calor tiene una cota máxima. Debido a esta cota, desde 2005 se ha notado que el aumento de la frecuencia de reloj ha aumentado de forma mínima y que el límite de diseño térmico de los procesadores ha alcanzado una constante.

Teniendo en cuenta las situaciones descritas anteriormente, el aumento en el desempeño de los procesadores va a estar limitado por dos factores: la disminución en la tecnología de fabricación y la capacidad de disipación térmica. Sin embargo, estos límites están más cercanos para los conjuntos de instrucción CISC que para los RISC, por lo que estos últimos constituyen una oportunidad para satisfacer las demandas de rendimiento de la sociedad. Uno de los ejemplos que demuestra la adopción de los conjuntos de instrucción RISC en áreas donde anteriormente dominaban los CISC son los computadores portátiles. En noviembre de 2019 Microsoft lanzó una versión RISC (Arm) de su tableta Surface Pro, mientras que en Noviembre de 2020 Apple lanzó la versión RISC de su MacBook Air. Esto demuestra la tendencia de los productores de dispositivos electrónicos para incluir procesadores RISC en sus diseños.

Previendo esta tendencia de adoptar sistemas RISC como solución a los límites físicos del desarrollo de procesadores, en el año 2010 nació el conjunto de instrucción RISC-V en la Universidad de California, Berkeley. Para 2020 RISC-V había crecido lo suficiente como para ser considerado el equivalente en hardware de Linux. El objetivo de RISC-V es ser un conjunto de instrucción abierto y libre de sesgos comerciales con el fin de promover el desarrollo de hardware a nivel mundial. Dentro de muchas características que hacen a RISC-V un conjunto de instrucción que rompe con los esquemas del pasado está el hecho de que sea libre, o en otras palabras, gratuito. Resulta curioso pensar que si bien casi todas las interacciones de los humanos hacia sus dispositivos electrónicos implican un uso indirecto de un conjunto de instrucción, estos no son libres para que cualquier persona pueda desarrollar sus propios procesadores. La mayoría de los conjuntos de instrucción son propietarios, por lo que si algún interesado quiere desarrollar su propia implementación deberá pagar millonarias regalías.

A pesar que el uso de RISC-V no genere algún costo, sí es posible generar beneficios económicos a partir de una implementación que utilice este conjunto de instrucción. De hecho, esta opción es tan atractiva que empresas como Google, NVidia, Qualcomm, entre otras, han decidido apostarle a RISC-V con el propósito de desarrollar sus propias implementaciones para sus intereses comerciales. Y es que una de las ventajas que ofrece RISC-V es que si bien el conjunto de instrucción es estándar, las implementaciones de este no lo son. Fácilmente una implementación de RISC-V puede ser usada en un supercomputador donde el desempeño es fundamental, mientras que otra puede ser usada en un sistema embebido con la eficiencia energética en mente.

En términos generales, RISC-V está en buen camino para convertirse en el estándar de hardware a nivel mundial. En primer lugar, es un proyecto de código abierto lo cual permite una evolución constante y sin

presiones comerciales. En segundo lugar, su crecimiento se ha visto acompañado de la inclusión de otros conjuntos de instrucción RISC en el mercado, eliminando las barreras que pudiesen existir para usar esta clase de conjuntos de instrucción en aplicaciones como servidores y computadores de escritorio. En tercer lugar, a nivel tecnológico los conjuntos de instrucción RISC pueden dar cabida a desarrollos que no aplican para conjuntos de instrucción CISC. En cuarto lugar, el hecho de ser libre favorece que empresas inicien el proyecto de desarrollar sus propias implementaciones, lo cual a su vez ampliaría la oferta de mercado de RISC-V. En conclusión, RISC-V tiene todas las oportunidades para convertirse en un referente de hardware así como Linux se convirtió en un referente para el software.

Capítulo 2

Objetivos y Alcance

2.1. Objetivos

2.1.1. Objetivo General

Diseñar, implementar y validar un sistema digital en conformidad con las especificaciones del conjunto de instrucción RISC-V.

2.1.2. Objetivos Específicos

- Diseñar la microarquitectura de una unidad de procesamiento central conforme a los requerimientos de la especificación entera de 32 bits de RISC-V.
- Implementar la unidad de procesamiento central en el lenguaje de descripción de hardware Verilog de forma estructurada.
- Validar el diseño e implementación de la unidad de procesamiento central mediante simulaciones comportamentales, simulaciones estructurales y análisis estáticos de tiempos.
- Sintetizar y programar una tarjeta FPGA la descripción de la unidad de procesamiento central.
- Documentar la configuración y uso de las herramientas de síntesis, simulación y programación relacionadas al proyecto para su aprovechamiento en trabajos futuros.

2.2. Alcance

Considerando que el objetivo principal del proyecto es diseñar un sistema digital basado en RISC-V, en la tabla 2.1 se proponen las especificaciones mínimas y deseadas de la unidad de procesamiento. En ambos escenarios se espera que el sistema ofrezca capacidades de números enteros con anchos de datos mínimo de 32 bits. En general, los dos escenarios apuntan a una implementación en Verilog, un proceso de simulación y una implementación en FPGA.

Adicionalmente, se esperan obtener entregables secundarios que se producen como parte del desarrollo del proyecto. Si bien estos entregables no influyen directamente en el resultado del proyecto, sí pueden influir en

Especificación	Mínimo	Deseado
ISA	RV32I	RV64I
Pipeline (Profunidad)	2	5
¿Ejecución especulativa?	No	Sí
HDL	Verilog	Verilog
Simulación	Sí	Sí
Implementación en FPGA	Sí	Sí

Tabla 2.1: Especificaciones mínimas y deseadas para la unidad de procesamiento vectorial.

el desarrollo de proyectos futuros. La lista de los entregables secundarios se presenta a continuación:

- Documentación de las herramientas de compilación, síntesis, simulación y programación asociadas al desarrollo de hardware y software relacionado con RISC-V.
- Entornos de ejecución estándar para las herramientas mencionadas anteriormente.
- Manual de uso para la implementación y programación de la unidad de procesamiento.

Capítulo 3

Marco Teórico

En este capítulo se describen los fundamentos teóricos necesarios para el desarrollo del proyecto y a su vez, constituye la evidencia de la revisión de literatura planteada en la propuesta del proyecto. Dentro del marco teórico se describen conceptos de RISC-V como conjunto de instrucción, datapath y pipeline, peligros de la ejecución de instrucciones además de algunas técnicas de optimización para un pipeline. Finalmente, se presenta una ejemplificación de la cadena de compilación de software.

3.1. RISC-V

RISC-V es un conjunto de instrucción (ISA, *Instruction Set Architecture*) de código abierto desarrollado inicialmente en la Universidad de California, Berkeley y posteriormente por parte de RISC-V International [11]. Como su nombre lo indica, es un conjunto de instrucción que explota la filosofía de computación de conjunto de instrucción reducido (RISC, *Reduced Instruction Set Computing*), sin embargo, las características relevantes de su diseño no se limitan al número de instrucciones únicamente. A continuación se presentan algunas características del ISA RISC-V junto con el razonamiento que llevó a los diseñadores a incluirlas.

3.1.1. Filosofía de Diseño y Organización

Una de las características principales de RISC-V es la filosofía sigue y que se indica en su nombre. Un ISA RISC define una serie de instrucciones con el fin de realizar operaciones sencillas y de propósito general. Una filosofía opuesta es la computación de conjunto de instrucción complejo (CISC, *Complex Instruction Set Computing*) donde las instrucciones se definen para realizar operaciones específicas. El término *reducido* hace referencia a que la instrucción tiene que realizar una cantidad de trabajo reducida en comparación a la de una instrucción de un conjunto de instrucción *complejo*. La baja carga de trabajo que realiza un procesador RISC se traduce en que este puede ejecutar más instrucciones por ciclo de reloj que uno CISC. Vale la pena notar que otras características pueden ser comunes para los ISAs RISC, aunque esto se debe a similitudes entre los conjuntos de instrucción y no a la filosofía de diseño como tal [9].

RISC-V sigue la filosofía de diseño reducida con el fin de favorecer el tamaño de los circuitos integrados de sus implementaciones. Este argumento se basa en que el costo de un circuito integrado tiene una relación cuadrática con el área que ocupa en la oblea de silicio. Un ejemplo de esta situación es que un procesador que

ocupa la mitad del espacio que otro tendría un costo cuatro veces menor que su competencia. Ahora bien, una implementación de un ISA simple disminuye el área ocupada por el procesador lo cual favorece el costo del mismo. Otros beneficios de un ISA simple son una documentación menos extensiva, lo cual ayuda a que los usuarios pueden usarlo más fácilmente, y un proceso de validación más rápido para las implementaciones.

Otra característica que diferencia a RISC-V de la mayoría de otros ISAs es que este es modular en vez de incremental. Un ISA incremental es uno tal que conforme pasa el tiempo se agregan instrucciones para dar soporte a nuevas características. El problema de los ISAs incrementales es que para mantener la compatibilidad retroactiva es necesario mantener todas las instrucciones, así estas estén obsoletas. RISC-V evita esta situación al organizar sus instrucciones en módulos, donde solo uno de ellos es obligatorio. En este orden de ideas, la cantidad de instrucciones que deben ser soportadas va a estar determinada por las necesidades del diseño y no por la compatibilidad retroactiva [9].

3.1.2. Código Abierto

Si bien RISC-V no es el primer conjunto de instrucción abierto, sí ha sido uno de los máximos exponentes de esta clase de licencia. El objetivo de RISC-V al ser un ISA abierto es promover la adopción de este por parte de los múltiples actores que se pueden ver involucrados. Siendo un conjunto de instrucción abierto RISC-V previene que su comunidad sufra divisiones y aislamientos que terminarían por romper la interoperabilidad entre las herramientas construidas para RISC-V. El principal beneficio de RISC-V como ISA abierto es que cualquier persona, empresa y organización que lo utilice para desarrollar sus propias implementaciones o herramientas de software no deberá pagar ninguna regalía [11].

Otro beneficio de RISC-V como ISA de código abierto es que este permite que un usuario final pueda realizar modificaciones al ISA para incluir sus propias instrucciones. Un sistema basado en RISC-V puede incluir sus propias instrucciones con el fin de acelerar cálculos en áreas como procesamiento de señales, aprendizaje de máquinas y criptografía. No obstante, se están diseñando extensiones de RISC-V para satisfacer las demandas de áreas como las mencionadas anteriormente. Un ejemplo de esto es la extensión vectorial, que busca tener uso en implementaciones especializadas en aprendizaje de máquinas y criptografía [11].

Finalmente, el hecho de que RISC-V sea de código abierto permite la colaboración de cualquier persona en aras de mejorar el ISA. La colaboración abierta es la clave para que RISC-V se mantenga neutral y libre de sesgos hacia un tipo de implementación. Vale la pena mencionar que si bien RISC-V es un ambiente que fomenta la participación abierta, existen comités que deciden la evolución del ISA a partir de los aportes de la comunidad. Si bien hay representantes de organizaciones comerciales dentro de estos comités, la evolución del ISA se basa en decisiones técnicas [11].

3.1.3. Módulos de la Arquitectura

RISC-V es un conjunto de instrucción modular, por lo que sus instrucciones se organizan en módulos según su funcionalidad denominados especificaciones. Con el fin de garantizar la estabilidad del conjunto de instrucción, todas las especificaciones pasan por un proceso de congelamiento y ratificación. El congelamiento de una especificación quiere decir que esta no será objeto de cambios técnicos como lo sería agregar una nueva instrucción o modificar la codificación de una ya existente. Por su parte, la ratificación indica que tanto la definición técnica como la documentación de una especificación están completas. La tabla 3.1 presenta las principales especificaciones de RISC-V con su funcionalidad. Vale la pena resaltar que las especificaciones

presentadas en la tabla 3.1 son aquellas que se encuentran ratificadas [11].

Nombre	Funcionalidad	Versión
RV32I	Computación entera de 32 bits	2.0
RV64I	Computación entera de 64 bits	2.1
M	Multiplicación y división	2.0
A	Operaciones atómicas	2.1
F	Punto flotante de precisión simple	2.2
D	Punto flotante de precisión doble	2.2
Q	Punto flotante de precisión cuádruple	2.2
C	Instrucciones comprimidas	2.0
ZiCSR	Registros de control y estado	2.0
ZiFencei	Protección de búsqueda de instrucción	2.0

Tabla 3.1: Módulos del conjunto de instrucción con su nombre, funcionalidad y versión. En esta tabla sólo se presentan los módulos que se encuentran ratificados [11].

Computación Entera (RV32I y RV64I)

La única especificación obligatoria para cualquier implementación de RISC-V es la de computación entera, tanto para 32 como para 64 bits. La especificación de computación entera de 32 bits está diseñada para soportar sistemas operativos modernos con una implementación mínima de hardware. La especificación entera de 32 bits tiene cuarenta (40) instrucciones, aunque para algunas implementaciones este número puede reducirse a treinta y ocho (38). Por su parte, la variante de 64 bits aprovecha las instrucciones de la variante de 32 bits y agrega quince (15) instrucciones adicionales. Tanto la especificación entera, como el resto de especificaciones, utilizan treinta y dos bits para codificar las instrucciones, con la excepción de la especificación C que utiliza 16 bits [11].

La especificación entera de RISC-V define treinta y dos (32) registros de propósito general (x0-x31) con un ancho (XLEN) dado por la variante que se utilice. Otro registro es definido para albergue la dirección del contador de programa (PC, *Program Counter*). La codificación de las instrucciones se hace con cuatro formatos de instrucción, que son detallados en la sección correspondiente. Dependiendo del formato de instrucción y de la instrucción, se pueden generar seis tipos diferentes de valores inmediatos. Todos los valores inmediatos son extendidos en signo para facilitar la decodificación y ubican el bit de signo en la misma posición dentro de la instrucción [11].

Dado que RISC-V es una arquitectura de carga/almacenamiento, las operaciones están definidas para tener máximo dos registros fuente y un registro destino, aunque las instrucciones pueden codificar valores inmediatos usados en las operaciones. Algunas de las operaciones soportadas por la especificación entera son sumas, resta, and, or, xor y corrimientos aritméticos y lógicos. Para la transferencia de control se definen saltos condicionales y no condicionales. Finalmente, se incluyen instrucciones para gestión de accesos a memoria y periféricos y para realizar llamados al sistema. La especificación entera de RISC-V permite emular prácticamente todas las especificaciones opcionales, siendo la única excepción la especificación atómica [11].

Multiplicación y División (M)

RISC-V define la especificación de multiplicación y división para realizar estas operaciones en hardware. Las instrucciones de multiplicación y división no se encuentran definidas en la especificación de computación

entera ya que su inclusión habría perjudicado implementaciones de bajo nivel. Otro motivo para separar estas operaciones de la especificación base es que RISC-V contempla que la multiplicación y división se pueden realizar en aceleradores. En el caso de la multiplicación es posible multiplicar dos registros y almacenar ya sea los 32 bits menos significativos o los 32 bits más significativos. Para la división se definen comportamientos similares en cuanto a la separación de los bits más y menos significativos. La división también define los comportamientos en caso de overflow o de una división por cero [11].

Operaciones Atómicas (A)

La especificación A define las operaciones atómicas que se pueden realizar en la memoria y que requieren definiciones más allá de la instrucción FENCE de la especificación base. Las operaciones atómicas definidas en esta especificación tienen dos tipos: operaciones de memoria atómicas (AMO, *Atomic Memory Operations*) y carga reservada/almacenamiento condicional. El primer tipo de operaciones recuperar un valor de la memoria, al cual se le aplica un operador binario y se vuelve a almacenar en la memoria. Durante la ejecución de una operación atómica no es posible que otros núcleos o hilos accedan al valor de la memoria. El segundo tipo de instrucciones están diseñadas para operaciones atómicas más complejas. En este caso, en vez de prevenir cualquier acceso a la memoria, las operaciones fijan valores que indican si se ha realizado alguna operación en dicha dirección de memoria [11].

Punto Flotante (F, D y Q)

Para realizar operaciones de punto flotante se han definido tres especificaciones dentro de RISC-V: F para operaciones de precisión simple, D para operaciones de precisión doble y Q para operaciones de precisión cuádruple. La especificación F se considera la base de las especificaciones de punto flotante y se basa en el estándar IEEE 754-2008. Ya que es la especificación base para las operaciones de punto flotante, esta agrega los registros de punto flotante y un registro de control de estado. En lo referente a los registros de punto flotante, la especificación agrega treinta y dos registros de punto flotante con un ancho determinado por la máxima precisión implementada (FLEN). La especificación F agrega un registro de control de estado (FCSR) donde se establece el modo de operación y el estado de excepciones. Ya que la especificación F requiere de un registro de control de estado, esta depende de las instrucciones que realizan estas operaciones (ZiCSR). Con el fin de soportar operaciones de mayor precisión, se definen las especificaciones D y Q para operaciones de precisión doble y precisión cuádruple respectivamente. Estas especificaciones amplían el ancho de los registros de punto flotante y fijan los modos de operación para soportar diferentes precisiones [11].

Registros de Control y Estado (ziCSR)

Los registros de control y estado (CSRs, *Control Status Registers*) están ubicados en un espacio de direccionamiento de doce (12) bits y son accedidos mediante las instrucciones de la especificación ZiCSR. Los registros de control y estado almacenan valores que definen la operación de una implementación en diferentes aspectos. Por ejemplo, para medir el desempeño de una implementación existe un CSR que contiene el número de instrucciones ejecutadas y otro que contiene el número de ciclos de reloj que han pasado. Otro ejemplo del uso de los CSRs esta relacionado a los modos de operación de máquina y supervisor que son usados por sistemas operativos. En general, los CSRs contienen información relevante para el entorno de ejecución de una implementación y su gestión depende de la especificación ZiCSR [11].

3.1.4. Formatos de Instrucción

En la especificación de computación entera se definen cuatro formatos de instrucción. El diseño y uso de cada formato de instrucción depende de el origen de los datos, la operación que se va a realizar y su destino. El primer formato de instrucción corresponde a las operaciones entre registros (formato R) y permite realizar operaciones binarias entre datos almacenados en los registros. El formato R tiene dos registros fuente (*rs1* y *rs2*) y un registro destino (*rd*). Un formato similar (formato I) incluye un valor inmediato en reemplazo de un registro fuente, y aparte de ser usada para operaciones binarias este formato es usado para cargas a memoria. El tercer formato de instrucción (formato S) se usa para los almacenamientos en memoria y saltos condicionales, por lo que debe codificar dos registros fuente y un valor inmediato. El último formato corresponde al formato U que codifica un registro destino y un valor inmediato. La figura 3.1 presenta los formatos de instrucción descritos [11].

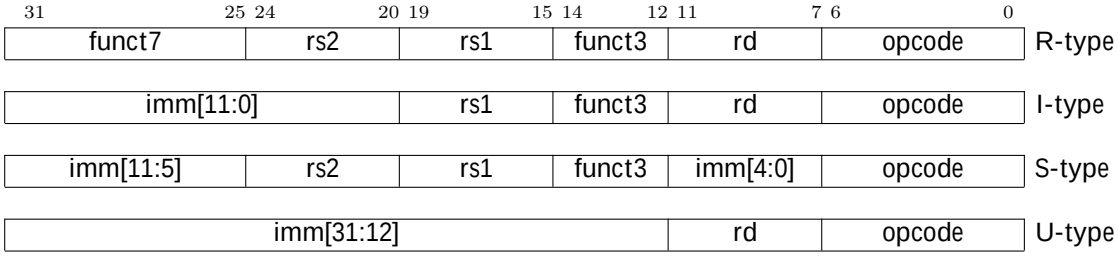

Figura 3.1: Formatos de instrucción definidos para RISC-V. Tomado de [11].

A partir de tres de los cuatro formatos de instrucción se pueden establecer cinco tipos de formatos de valor inmediato. El primer tipo de inmediato es el resultado de decodificar una instrucción de formato I. Note como se realiza la extensión de signo a partir del bit más significativo de la instrucción, esta metodología aplica para todos los tipos de inmediatos. El segundo formato de inmediato corresponde a la decodificación del formato de instrucción S. Un tercer tipo de inmediato (tipo B), usado en los saltos condicionales, se deriva del tipo S donde se agrega un cero en el bit menos significativo y se realiza un desplazamiento del primer campo de inmediato. El cuarto formato de inmediato corresponde a la decodificación de una instrucción de formato U. Para el inmediato de tipo U se fijan los doce (12) bits menos significativos en cero. Finalmente, del inmediato de tipo U se deriva el inmediato J que fija únicamente el bit menos significativo en cero y ejecuta un desplazamiento con un campo del inmediato codificado.

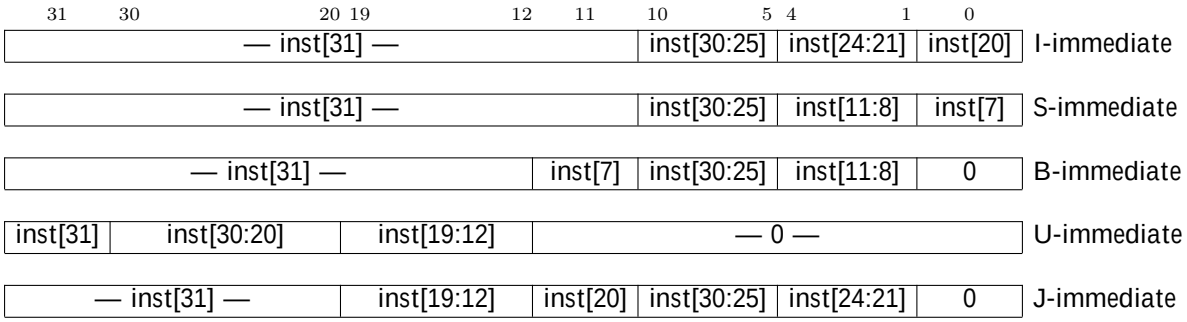

Figura 3.2: Tipos de inmediatos definidos para RISC-V. Tomado de [11].

3.2. Microarquitectura de Procesador

El diseño de una microarquitectura constituye el proceso mediante el cual se define una organización e interconexión de componentes digitales que permita ejecutar el comportamiento definido por un conjunto de instrucción (ISA, *Instruction Set Architecture*). Dentro del proceso de diseño de una microarquitectura se le llama *datapath* al conjunto de componentes que conforman la microarquitectura. En general, un datapath está conformado por componentes digitales que le permiten recuperar una instrucción de la memoria, decodificar la instrucción, ejecutar la operación correspondiente y almacenar el resultado en la memoria o en los registros del procesador [8].

Para recuperar la instrucción de la memoria se necesita de una dirección, que se suele almacenar en un registro de propósito general denominado contador de programa (PC, *Program Counter*). El ISA determina cómo se actualiza el contador del programa con el fin de ir recuperando las instrucciones que conforman el programa en ejecución. En el caso de RISC-V, el contador de programa puede ser actualizado de tres maneras distintas: con un incremento de cuatro unidades, con un desfase, o con una nueva dirección. El incremento de cuatro unidades permite la ejecución de dos instrucciones consecutivas, ya que todas las instrucciones de RISC-V son de 32 bits. El desfase se usa para las transferencias de control como saltos condicionales y no condicionales mientras que la nueva dirección se usa en saltos no condicionales o transferencias de control debido a excepciones [8].

La decodificación de la instrucción corresponde a un proceso en el cual se generan las señales de control necesarias para realizar la operación indicada en la instrucción. Las señales de control pueden ser las direcciones de los registros que deben ser leídas así como la dirección destino, el código de operación de la ALU, o el valor inmediato decodificado, entre otras. Dependiendo de nivel de complejidad de la microarquitectura van a ser necesarias más o menos señales de control. Por lo general, el comportamiento de la microarquitectura para el resto de la ejecución de una instrucción va a estar determinado por las señales de control generadas durante la decodificación [8].

Una vez se ha recuperado la instrucción de la memoria y se ha decodificado, es tarea del procesador llevar a cabo la operación indicada por la instrucción. En base al ISA RISC-V se pueden definir tres tipos de operaciones: aritmético-lógicas, transferencias de control y accesos a memoria. Las operaciones aritmético-lógicas realizan operaciones binarias entre dos operandos y almacenan el resultado en un registro del procesador. Las transferencias de control, en caso de ser condicionales, evalúan una sentencia lógica para determinar si se realizan o no saltos dentro del programa. En caso de ser no condicionales, se realiza un salto sin evaluar ninguna sentencia lógica. Finalmente, los accesos a memoria permiten cargar datos de la memoria y almacenarlos en un registro del procesador o almacenar el valor de un registro en la memoria. Si la operación realizada no requiere de accesos a memoria, los resultados son almacenados en uno de los registros del procesador para ser usados en la ejecución de otras instrucciones [8].

La figura 3.3 presenta un datapath diseñado para un subconjunto de la especificación entera de RISC-V. El subconjunto mencionado previamente corresponde a una instrucción de carga (`ld`), una de almacenamiento (`sd`), cuatro para realizar operaciones aritmético-lógicas (`add`, `sub`, `and` y `or`) y una para evaluar saltos condicionales (`beq`). En la figura se puede identificar el contador de programa, a la izquierda de la misma, y cómo este se actualiza ya sea mediante un aumento en cuatro unidades o mediante un valor que resulta de sumar el PC con el inmediato codificado en el salto condicional [8].

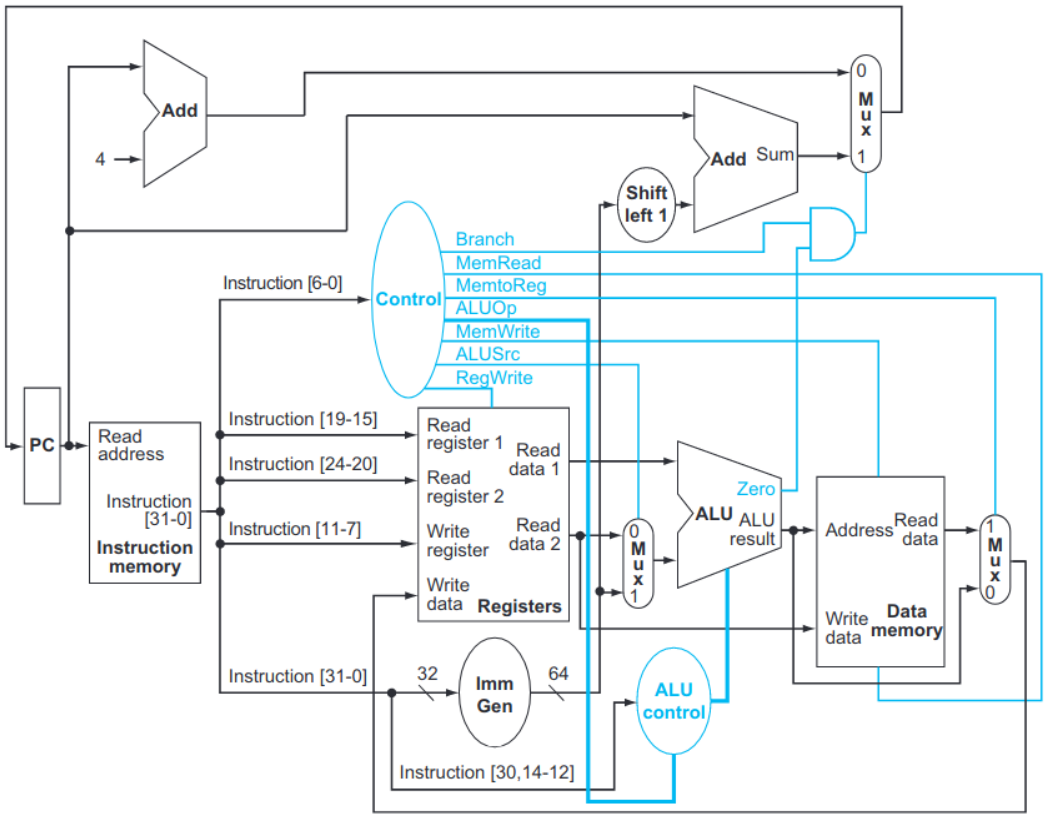

Figura 3.3: Datapath de una implementación de un subconjunto de la especificación entera de RISC-V. Imagen tomada de [8, p. 257].

El siguiente elemento del datapath de la figura 3.3 corresponde a la lectura de la memoria de instrucción. Para la lectura de la memoria se ingresa una dirección y la instrucción es recuperada. Posteriormente la instrucción pasa por la decodificación, donde se leen los registros, se genera el valor inmediato y las señales de control. Para la ejecución se utiliza una ALU cuyo código de operación fue generado en la decodificación. El resultado de la ejecución tiene dos opciones, puede ser usado para acceder a la memoria y realizar una carga o un almacenamiento, o puede ser almacenado en los registros de propósito general [8].

Si bien el datapath de la figura 3.3 implementa un subconjunto de la especificación entera de RISC-V, sirve como referencia para establecer lo que sería una implementación del conjunto de instrucción completo. El datapath utilizaría lo que se conoce como una implementación de un único ciclo. En otras palabras, esta implementación requeriría un ciclo de reloj para resolver todas las instrucciones, lo cual se traduce en que tanto la ejecución más rápida como la ejecución mas lenta tardarían lo mismo, disminuyendo su desempeño [8].

3.2.1. Pipelining

Para optimizar el diseño de un procesador es posible incluir una técnica denominada pipelining que solapa la ejecución de instrucciones dentro del pipeline. Para la técnica de pipelining se requiere dividir la ejecución en diferentes etapas de modo tal que no interfieran entre ellas. A partir del pipeline de la figura 3.3 se pueden identificar cinco etapas de la instrucción:

1. Búsqueda de instrucción de la memoria (IF, *Instruction Fetch*).
2. Decodificación de instrucción y lectura de registros (ID, *Instruction Decode*).
3. Ejecución de la operación (EX, *Execution*).
4. Acceso a la memoria de datos (MEM, *MEMory access*).
5. Escritura en registros (WB, *register WriteBack*).

La división indicada anteriormente corresponde las etapas del pipeline clásico de RISC. El pipeline clásico RISC corresponde a una estrategia común utilizada por las primeras implementaciones de procesadores RISC. Sin embargo, conforme emergieron nuevas técnicas y tecnologías, se realizaron divisiones que permitieron aumentar el número de etapas del pipeline, lo cual aumenta el desempeño del mismo. Ahora, una vez se ha dividido la ejecución en etapas que pueden ser solapadas se puede proceder a su implementación en el hardware del datapath. La implementación física del pipeline consiste en definir la lógica para ejecutar una etapa de la ejecución de la instrucción y almacenar el resultado en un registro, que a su vez alimenta la lógica de la siguiente etapa. La figura 3.4 presenta la versión con pipeline del datapath de la figura 3.3.

La ejecución de diferentes instrucciones puede ser solapada dentro del datapath bajo la idea de que, en un instante de tiempo, la ejecución de una instrucción se realiza en una única etapa. Dado que una instrucción requiere solo de una etapa durante la ejecución, el resto de etapas está libre para ejecutar otras instrucciones, por lo que son aprovechadas para este fin. Por consiguiente, en un instante de tiempo determinado el procesador estará ejecutando, en el caso de la figura 3.4, cinco instrucciones simultáneamente donde una instrucción terminará su ejecución al final de un ciclo de reloj. No obstante, vale la pena aclarar que si bien cada ciclo de reloj termina la ejecución de una instrucción, cada instrucción tarda cinco ciclos de reloj para ser ejecutada en el pipeline. Para visualizar la operación de un pipeline sin presentar toda la microarquitectura se suelen usar diagramas de pipeline. La figura 3.5 presenta un diagrama de pipeline a

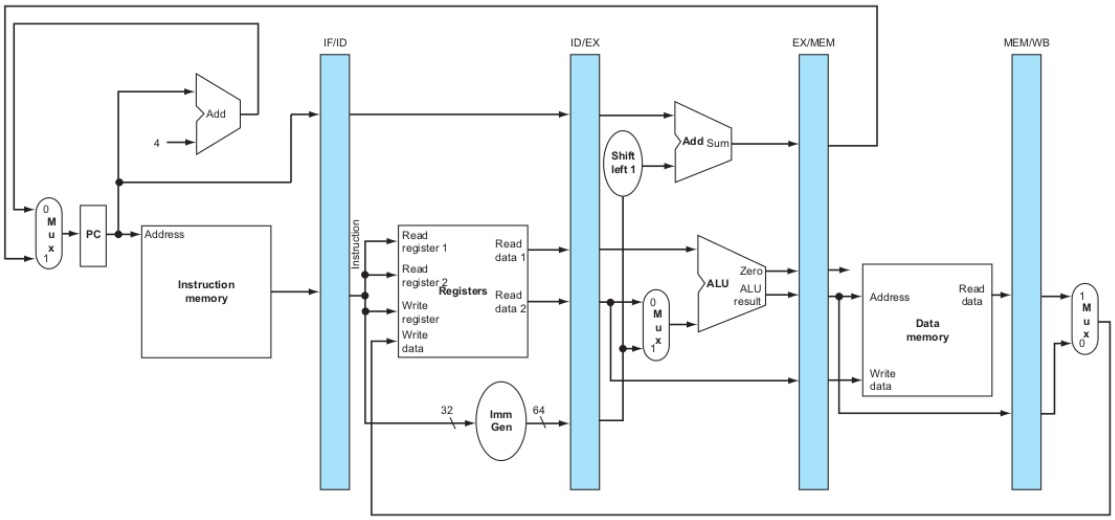

Figura 3.4: Datapath de la figura 3.3 con los registros de pipeline implementados. Figura tomada de [8, p. 279].

Instrucción	**CC0**	**CC1**	**CC2**	**CC3**	**CC4**	**CC5**	**CC6**	**CC7**	**CC8**	**CC9**
I_0	IF	ID	EX	MEM	WB	–	–	–	–	–
I_1	–	IF	ID	EX	MEM	WB	–	–	–	–
I_2	–	–	IF	ID	EX	MEM	WB	–	–	–
I_3	–	–	–	IF	ID	EX	MEM	WB	–	–
I_4	–	–	–	–	IF	ID	EX	MEM	WB	–
I_5	–	–	–	–	–	IF	ID	EX	MEM	WB

Figura 3.5: Diagrama de pipeline para la ejecución de seis instrucciones en el datapath de la figura 3.4. Note como en el quinto ciclo de reloj (CC4) cinco instrucciones eran ejecutadas simultáneamente.

partir de las etapas definidas para el ejemplo de la figura 3.4.

3.2.2. Peligros en el Pipeline

Aunque la técnica de pipelining permita acelerar la ejecución de instrucciones en un datapath, su implementación puede presentar situaciones en las cuales se puede comprometer la ejecución de una instrucción, los datos del procesador o el flujo de ejecución del programa. Dependiendo de la causa del peligro, estos pueden ser divididos en tres tipos: estructurales, de datos y de control [8].

Peligros Estructurales

Los peligros estructurales se presentan cuando dos instrucciones tratan de acceder al mismo recurso físico. Un ejemplo de un peligro estructural es, en el caso de la figura 3.4, que solo se pueda acceder a una memoria. En esta situación una instrucción que requiera un acceso a la memoria entraría en conflicto con la etapa de búsqueda de instrucción, por lo que alguna instrucción no se podría ejecutar. Sin embargo, dado que la aplicación de la técnica de pipelining es casi universal, la mayoría de conjuntos de instrucción están diseñados para evitar esta clase de peligros. RISC-V no es una excepción a esto ya que fue diseñado explícitamente para ser usado con esta técnica [8].

Peligros de Datos

Los peligros de datos son aquellos que comprometen los datos almacenados en los registros del procesador [9]. Los peligros de datos son causados por dependencias entre dos instrucciones subsecuentes que pueden presentarse en tres situaciones diferentes:

1. Lectura después de escritura (RAW, *Read After Write*).
2. Escritura después de lectura (WAR, *Write After Read*).
3. Escritura después de escritura (WAW, *Write After Write*).

Los peligros RAW ocurren cuando la primera instrucción trata de escribir en en registro que es usado como operando por parte de la segunda instrucción. En el listado 3.1 se presentan dos instrucciones que presentan una dependencia RAW con el registro $\mathbf{x1}$. Note como la primera instrucción almacena el resultado de una suma en este registro y la siguiente instrucción utiliza este valor como un operando. Debido a la forma en la cual se estructura el pipeline, la primera instrucción tendrá el valor verdadero de $\mathbf{x1}$ cuando termine la etapa de ejecución, mientras que la segunda instrucción leería el valor de los registros al mismo tiempo. Debido a que la primera instrucción no habrá almacenado el valor de $\mathbf{x1}$ en el registro para cuando la segunda lo lea, esta última operará con el valor erróneo [9].

Los peligros WAR y RAW, por su parte, dependen de las condiciones de ejecución para perjudicar el estado del pipeline. Los listados 3.2 y 3.3 presentan pares de instrucciones que podrían generar peligros de datos WAR y WAW respectivamente. En el primer caso, el registro $\mathbf{x1}$ presenta una dependencia WAR debido a que primero se lee el registro y posteriormente se escribe. Este caso es problemático únicamente en condiciones de ejecución concurrentes cuando la segunda instrucción pueda culminar su ejecución antes que la primera. En el segundo caso, el registro $\mathbf{x1}$ presenta un peligro de escritura después de escritura. Al igual que la dependencia WAR, esta podría ser problemática cuando la ejecución de la segunda instrucción termine antes que la primera [9].

```
add x1, x2, x3
add x4, x1, x5
```

Listing 3.1: Instrucciones con dependencia de datos de tipo RAW.

```
add x1, x2, x3
add x2, x1, x5
```

Listing 3.2: Instrucciones con dependencia de datos de tipo WAR.

```
add x1, x2, x3
add x1, x4, x5
```

Listing 3.3: Instrucciones con dependencia de datos de tipo WAW.

Para prevenir los peligros de datos de tipo RAW es necesario implementar hardware adicional que permita usar un dato una vez haya sido calculado sin necesidad de almacenarlo y leerlo del registro. Para eliminar los peligros de tipo WAR y WAW se debe hacer uso de técnicas como la ejecución fuera de orden, *scoreboarding* o el algoritmo de Tomasulo [9].

Peligros de Control

El último tipo de peligro corresponde a los de control, que pueden alterar el flujo de ejecución de un programa de modo tal que se ejecuten instrucciones que no debían ser ejecutadas. Estos peligros ocurren cuando se presentan transferencias de control tanto condicionales como no condicionales. El listado 3.4 presenta un fragmento de código con un salto condicional que debe ser tomado. Sin embargo, entre el inicio de la ejecución de este salto condicional y su resolución se iniciará la ejecución de las instrucciones subsecuentes al salto, que no deberían ser ejecutadas. En este caso, se ha presentado un peligro de control [8].

```
main:
    addi x1, x0, 3
    addi x2, x0, 5
    bne  x1, x2, target
    addi x3, x0, 1
    addi x4, x0, 2
    addi x5, x0, 3
    addi x6, x0, 4

target:
    ...
```

Listing 3.4: Instrucciones con dependencia de datos de tipo WAW.

Los peligros de control no pueden ser eliminados completamente, aunque su efecto en el desempeño del procesador sí puede ser reducido. Una de las técnicas que pueden permitir reducir el impacto de los saltos condicionales es la predicción dinámica de ramas. La predicción dinámica de ramas trata de anticipar el resultado de un salto condicional sin evaluar la condición lógica, acelerando la ejecución de los programas. En caso tan que el resultado de la predicción no coincida con el resultado de la evaluación lógica, el pipeline debe revertir los cambios que hayan hecho las instrucciones ejecutadas y comenzar a ejecutar las instrucciones correctas. Si bien los predictores actuales tienen tasas de acierto elevadas, no es posible eliminar completamente la posibilidad de predecir erróneamente un salto condicional [8].

3.2.3. Optimización del Pipeline

Ante los posibles peligros que pueden generarse en un pipeline, existen técnicas de optimización que permiten mitigar o eliminar estos peligros. Si bien su implementación depende exclusivamente del pipeline que pretenden mejorar, la idea detrás de estas técnicas es la misma para todas las implementaciones. A continuación se van a presentar dos técnicas de optimización para el pipeline de un procesador: la predicción dinámica de ramas y el adelantamiento de datos. La predicción dinámica de ramas busca mitigar los peligros de control de un procesador mientras que el adelantamiento de datos busca eliminar los peligros de datos [8].

Predicción Dinámica de Ramas

La predicción dinámica de ramas busca anticipar el resultado de un salto condicional sin evaluar la condición lógica asociada a este. Esta clase de predicción de saltos se denomina dinámica porque es realizada durante la ejecución de las instrucciones en el procesador. Por el contrario, la predicción estática de saltos

se realiza durante el proceso de compilación. Existen diferentes tipos de predictores de saltos dependiendo de la lógica seguida para anticipar el resultado del salto. Una de las formas más simples de predicción de saltos es el predictor de dos bits, que almacena los dos últimos resultados de un salto y a partir de estos anticipa el siguiente resultado. La figura 3.6 presenta el diagrama de estados de un predictor de dos bits para determinar el resultado de un salto condicional.

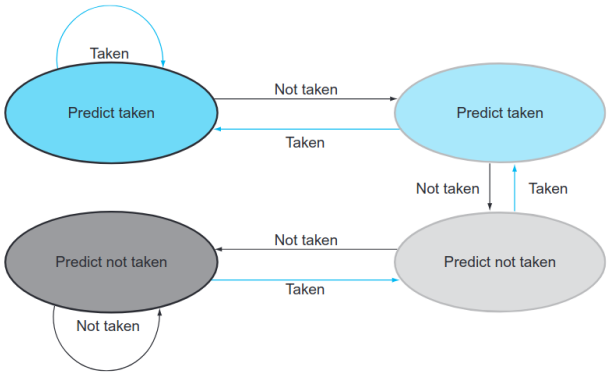

Figura 3.6: Diagrama de estados para un predictor de dos bits. Tomado de [8, p. 313].

A partir de los predictores de dos bits se pueden derivar predictores más avanzados, y complejos, para mejorar la tasa de predicción. Uno de estos predictores derivados, denominado predictor de correlación, mezclan el historial global de las ramas para determinar el resultado de la predicción. Un ejemplo de un predictor de correlación es el predictor *gshare* que mezcla el historial global de las ramas con la dirección del salto mediante un operador binario para acceder a los bits de predicción. Otro tipo de predictor más avanzado es el predictor de torneo, que mezcla un predictor de correlación con un predictor local y decide entre la predicción de cada uno. Las figuras 3.7 y 3.8 presentan diagramas de un predictor gshare y un predictor de torneo respectivamente [8].

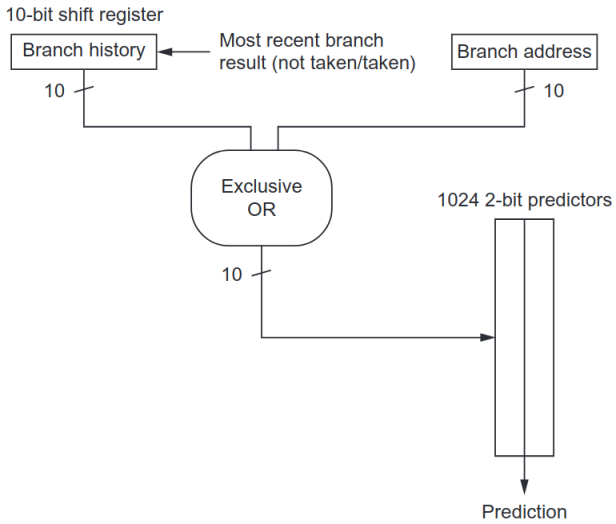

Figura 3.7: Predictor de correlación ghsare. Tomado de [9, p. 186].

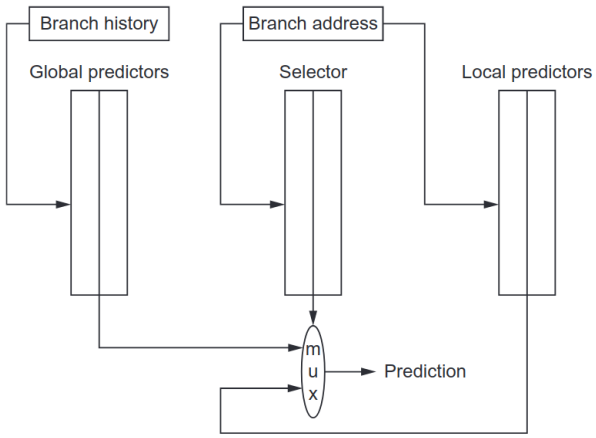

Figura 3.8: Predictor de torneo con un predictor global y un predictor local. Tomado de [9, p. 186].

Adelantamiento de Datos

El adelantamiento de datos es una técnica que le permite a una instrucción usar un dato sin que este haya sido almacenado el los registros o en la memoria. El objetivo del adelantamiento de datos es eliminar los peligros de datos RAW de forma directa. La lógica del adelantamiento de datos consiste en identificar si uno de los registros fuente corresponde al registro destino de la instrucción previa. Si se llega a presentar esta situación, el adelantamiento de datos puede habilitar unos multiplexores que permitan acceder al dato sin que este haya sido escrito en el registro. En general, la implementación del adelantamiento de datos consiste en alimentar multiplexores con los datos calculados por la o las unidades de ejecución y con los datos provenientes de los registros, y seleccionar alguna fuente a partir de la lógica del adelantamiento. la figura 3.9 presenta la implementación de esta técnica en el pipeline de la figura 3.4 [8].

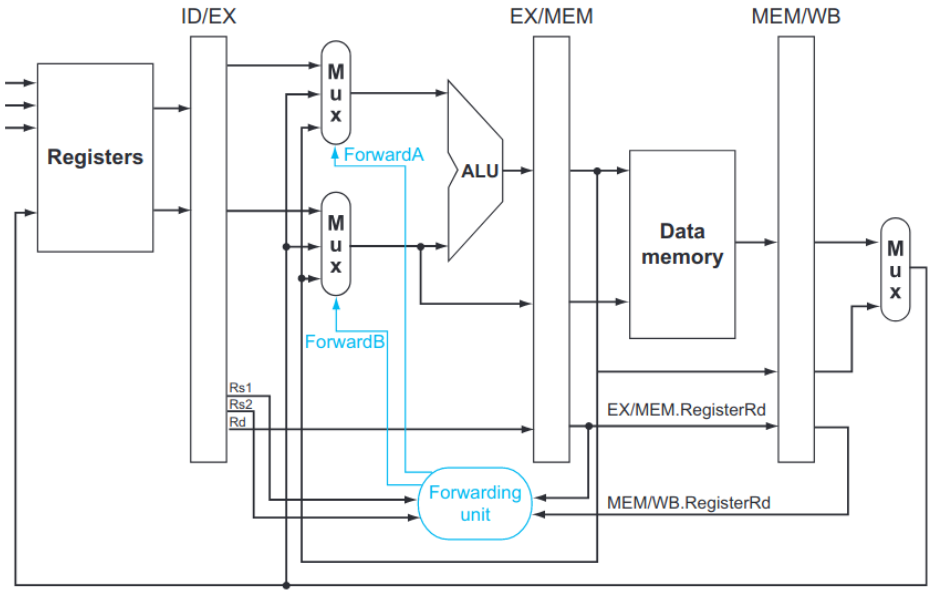

Figura 3.9: Implementación de la técnica de adelantamiento de datos en el pipeline de la figura 3.4. Tomado de [8, p. 299]

Capítulo 4

Metodología

Para llevar a cabo el desarrollo del sistema digital es necesario definir una metodología orientada al diseño de hardware digital. Debido a que el sistema digital va a ser desplegado en una FPGA (*Field-Programmable Gate Array*, Arreglo de compuertas programable en campo), esta metodología debe estar direccionada hacia tal fin. Por otra parte, la metodología debe considerar las etapas para llevar a cabo un proyecto de ingeniería como lo son la revisión de literatura, la etapa de diseño, la etapa de implementación y la etapa de validación. En este orden de ideas, la metodología del proyecto va a estar dada por la unión entre la metodología de diseño de hardware para FPGA y la metodología para el desarrollo de proyectos de ingeniería. Un diagrama general de la metodología que será seguida en el proyecto se presenta en la figura 4.1.

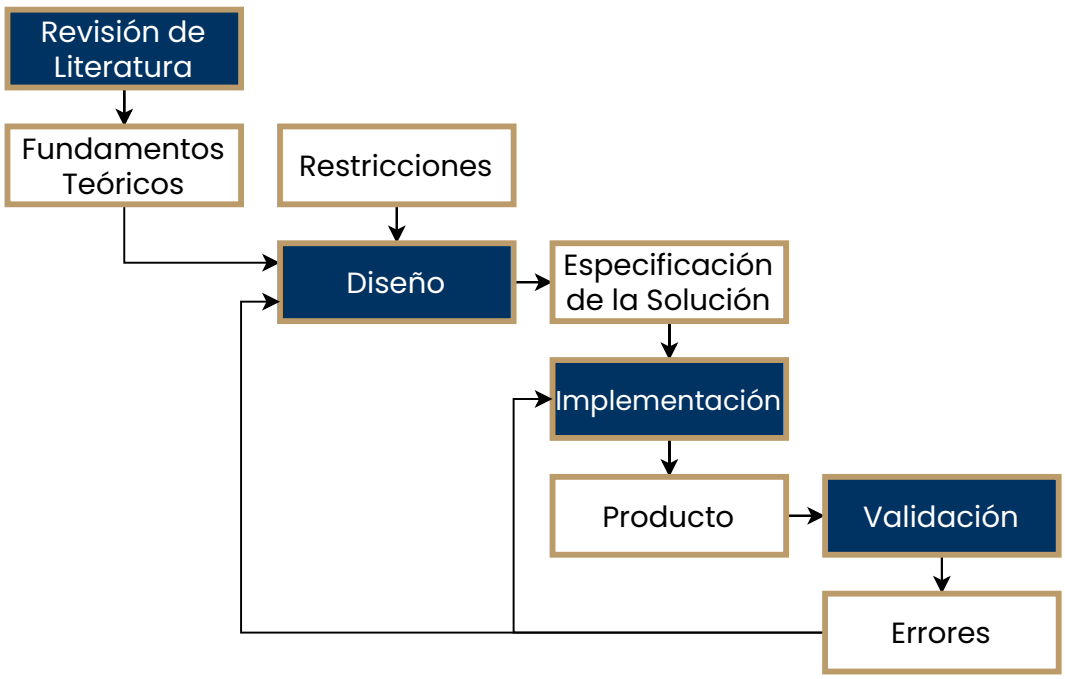

Figura 4.1: Vista general de la metodología de desarrollo del proyecto. En esta vista general se presentan las cuatro etapas: revisión de literatura, diseño, implementación y validación.

Siguiendo las etapas de un proyecto en ingeniería, el desarrollo del sistema digital va a contar con las etapas de revisión de literatura, de diseño, de implementación y validación. La etapa de revisión de literatura corresponde a la primera etapa del proyecto, donde se establecen las bases teóricas para la ejecución del mismo. La etapa de diseño busca producir una especificación de la solución planteada teniendo en cuenta una serie de restricciones dadas por el contexto del proyecto. A partir de la especificación de la solución se procede con su implementación orientada a su despliegue en FPGA. Finalmente, a partir de la implementación y de los resultados obtenidos del despliegue, se lleva a cabo la validación y caracterización de la solución. En caso tal que que el diseño o la implementación tengan errores, estos serán corregidos de forma iterativa en las etapas correspondientes.

4.1. Revisión de Literatura

Teniendo en cuenta la metodología definida para el proyecto, la primera etapa del desarrollo es la revisión de literatura, donde se van a consultar las fuentes bibliográficas para obtener los fundamentos teóricos necesarios para ejecutar el resto del desarrollo. La revisión de literatura se hará con artículos y libros relacionados al diseño de procesadores. El capítulo del marco teórico corresponde a la evidencia de la ejecución de esta etapa de la metodología. Los dos libros principales que fueron usados como referencia para el proyecto son:

- *Computer Organization and Design: The Hardware/Software Interface (RISC-V Edition)* [8]: Este libro de David Patterson y John Hennesy sirve como introducción a la organización y el diseño de computadores. Presenta conceptos como los componentes de un computador, operaciones binarias, la idea de conjunto de instrucción, la construcción de un pipeline y la jerarquía de memoria. Esta edición utiliza el conjunto de instrucción RISC-V como referencia para explicar los conceptos, escribir software y desarrollar hardware por lo que su contenido está relacionado directamente con el diseño del sistema digital.
- *Computer Architecture: A Quantitative Approach* [9]: Constituye una versión avanzada del libro *Computer Organization and Design* de los mismos autores y enfocada en el diseño de unidades de procesamiento, en especial procesadores. Si bien algunos temas están orientados a implementaciones de alto nivel, otros eran relevantes para el diseño del sistema digital. Al igual que su versión más básica, este libro utiliza el conjunto de instrucción RISC-V para la explicación de conceptos.

Algunos proyectos relacionados a RISC-V como implementaciones y software fueron usados como referencia para el desarrollo de este proyecto. A continuación se presentan algunos proyectos analizados durante la revisión de literatura:

- *mriscvcore* [3]: Es una implementación del conjunto de instrucción RISC-V desarrollada en la Universidad Industrial de Santander. El núcleo soporta el conjunto de instrucción RV32IM con un pipeline de tres etapas descrito en Verilog legible. Es un núcleo con características similares al núcleo Cortex M0 de Arm, y sobre el cual se construyó un microcontrolador denominado *mriscv*. A partir del diseño del microcontrolador se han fabricado tarjetas de desarrollo similares a lo que sería un Arduino Uno, solo que utilizando el conjunto de instrucción RISC-V.
- *ibex* [6]: Anteriormente conocido como *zero-riscy* es una implementación de RISC-V desarrollada por el grupo PULP Platform de la Università di Bologna y el ETH de Zúrich. La implementación fue donada a la compañía lowRISC que lo mantiene y continua desarrollando. Esta implementación utiliza el conjunto de instrucción RV32IMC en un pipeline de dos etapas. Actualmente posee diferentes modos de configuración

que le permiten variar su microarquitectura dependiendo de las necesidades del usuario.

- *CV32E40P* [7]: También conocido como RI5CY, es una implementación desarrollada por PULP Platform y posteriormente donada al grupo OpenHW. Tiene un pipeline de cuatro etapas y soporta el conjunto de instrucción RV32IMC al cual se le puede agregar la especificación F, que es opcional. Tanto *ibex* como *CV32E40P* sirven como base para el microcontrolador PULPino, ya que los dos núcleos tienen la misma interfaz.
- *GCC* [5]: Es la versión de RISC-V para el compilador *GNU Compiler Collection*. A pesar que se diseño inicialmente para compilar el lenguaje de programación C, en la actualidad permite compilar C++, Fortran, Ada y otros. Es el compilador que fue adoptado por los sistemas operativos UNIX como Linux y es uno de los proyectos de software libre más reconocidos.

4.2. Diseño

Tras la etapa de revisión de literatura se procede a realizar la etapa de diseño, donde se especifica la microarquitectura del núcleo y los módulos que lo componen. Vale la pena notar que debido al carácter iterativo del diseño, esta etapa estará presente durante toda la ejecución del proyecto. El objetivo de mantener un proceso iterativo es converger a un diseño que cumpla con los requerimientos funcionales dados por el conjunto de instrucción RISC-V. Adicionalmente el diseño iterativo permite optimizar el diseño y aprovechar oportunidades de mejora para el núcleo del procesador.

Como se mencionó previamente, en la etapa de diseño se busca especificar la microarquitectura y sus módulos de una forma funcional. En otras palabras, durante esta etapa no se espera realizar alguna implementación ya sea en el lenguaje de alto nivel ó en el lenguaje de descripción de hardware. La especificación del diseño será presentada en este documento como evidencia del desarrollo del mismo.

4.3. Implementación

La tercera etapa de la metodología corresponde a la implementación en el lenguaje de descripción de hardware y lenguaje de programación en alto nivel del diseño. Esta implementación busca generar una representación del diseño consecuente con la especificación desarrollada en la etapa anterior. La implementación en el lenguaje de programación de alto nivel se realiza en Python mientras que la implementación de lenguaje de descripción de hardware se realiza en Verilog. Vale la pena notar que la implementación en Verilog se realiza siguiendo el estándar IEEE 1364-2005, buscando generar una descripción estructural, sintetizable y legible. De forma similar a la etapa de diseño, la etapa de implementación se encuentra presente durante la mayor parte del proyecto debido al carácter iterativo del mismo. En el caso de la implementación, las iteraciones permiten corregir los errores que se presenten y optimizar el diseño.

Otro aspecto a tener en cuenta durante la etapa de implementación corresponde al despliegue del diseño en las tarjetas de desarrollo FPGA. Este despliegue busca cargar el diseño en la tarjeta a partir de una cadena de herramientas específica a cada tarjeta. En términos generales el proceso de despliegue comprende la etapa de síntesis, *place and route* y análisis de tiempo. La síntesis permite convertir el diseño de una descripción en Verilog a un modelo que use las celdas lógicas disponibles en la FPGA. Por su parte, el *place and route* ubica físicamente las celdas lógicas y las conecta de modo tal que represente adecuadamente el diseño planteado. Finalmente, el análisis de tiempo garantiza que el diseño podrá operar a la frecuencia de reloj disponible en

la tarjeta.

4.4. Validación

La última etapa de desarrollo del proyecto corresponde a la validación, donde se evalúa si el diseño y la implementación son consistentes entre sí. Para la validación se aprovechan las implementaciones en el lenguaje de alto nivel y el lenguaje de descripción de hardware. Para la validación del diseño se tendrán tres instancias, donde dos usaran una validación cruzada entre la implementación en alto nivel y la descripción de hardware. Para la validación cruzada se tendrá como referencia la implementación en alto nivel asumiendo que dicha implementación es congruente con el diseño especificado. La validación cruzada se dará en dos niveles de abstracción: a nivel de módulos y a nivel de microarquitectura. En el primer nivel de abstracción, a nivel de módulos, se evaluará si cada módulo opera de forma independiente según el comportamiento esperado del mismo. Por su parte, a nivel de microarquitectura se evaluará si esta opera correctamente según lo especificado por el conjunto de instrucción. Finalmente, en la tercera instancia de validación se evaluará el funcionamiento del núcleo una vez desplegado en la FPGA.

Capítulo 5

Diseño

Este capitulo presenta el diseño del núcleo de procesador Core101, una implementación de RISC-V. El núcleo Core101 tiene un pipeline de seis etapas y utiliza el conjunto de instrucción RV32I. A lo largo de este capitulo se presenta el núcleo de forma general, se definen los protocolos de comunicación del núcleo con la memoria y periféricos además de los registros de pipeline. La figura 5.1 presenta el logo del Core101.

Figura 5.1: Logo del núcleo de procesador Core101.

El núcleo Core101 es el componente de mayor jerarquía dentro del diseño planteado. En este sentido, el núcleo está constituido por varios módulos de menor jerarquía organizados según una microarquitectura determinada. El funcionamiento y organización de estos módulos busca satisfacer el objetivo fundamental de soportar el conjunto de instrucción RISC-V en su especificación entera (RV32I). La organización de la microarquitectura se presenta en la figura 5.2. La microarquitectura tiene en cuenta dos parámetros

principales: el número de etapas de pipeline y el ancho de datos, que para el Core101 son seis etapas y un ancho de datos de 32 bits respectivamente.

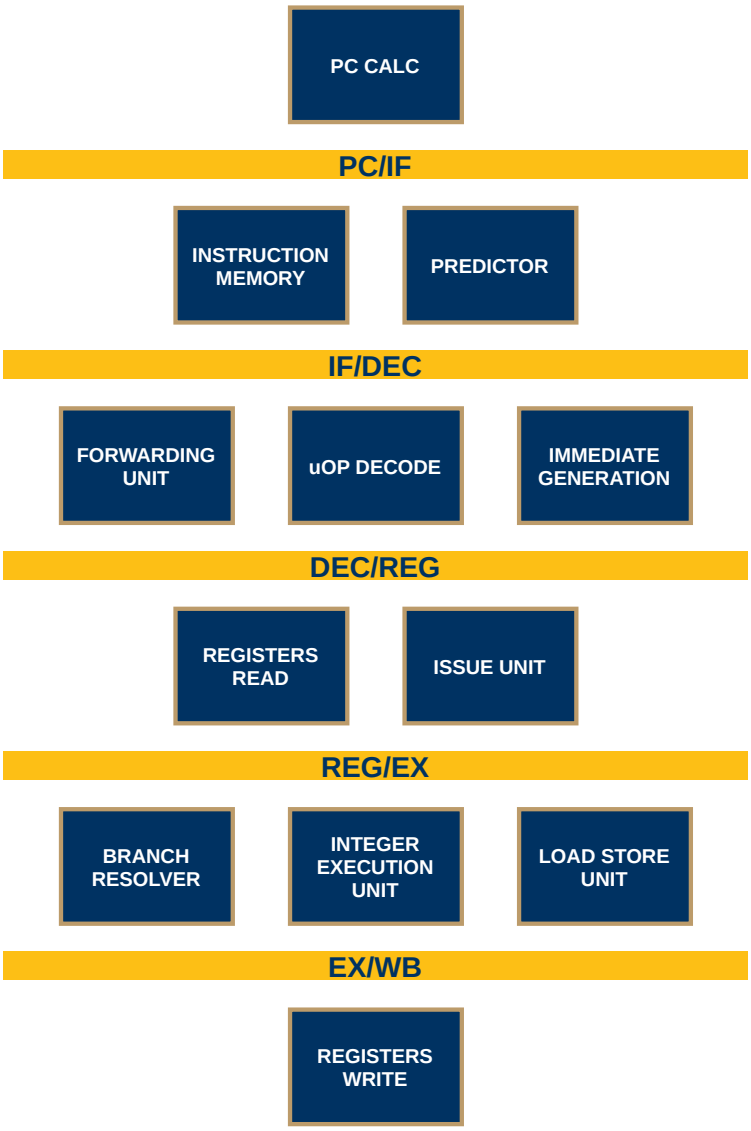

Figura 5.2: Microarquitectura del núcleo Core101.

5.1. Descripción del Pipeline

Core101 es un núcleo que utiliza la especificación entera de 32 bits (RV32I) del conjunto de instrucción RISC-V. El núcleo consta de un pipeline de seis etapas con ejecución en orden y accesos a memoria bajo el concepto de arquitectura Harvard modificada. En este sentido, a nivel de núcleo se hace la distinción entre memoria de datos y memoria de instrucción. Adicionalmente, el pipeline utiliza algunas técnicas de optimización para mejorar su desempeño como lo son la predicción dinámica de ramas y el adelantamiento de datos. Las etapas del pipeline están basadas en el pipeline clásico de las arquitecturas RISC, pero incluye una etapa de lectura de registros y tres unidades de ejecución.

En la primera etapa del pipeline (PC) se genera el valor del contador de programa (PC, *Program Counter*) para recuperar la instrucción de la memoria correspondiente. Este valor puede ser generado de tres formas diferentes. La primera corresponde a un incremento de cuatro unidades en relación al anterior valor del PC, que se almacena en el registro de pipeline PC/IF. La segunda corresponde a un valor generado a partir de una predicción en el caso de un salto condicional. Por último, la tercera corresponde a una corrección en el caso de una predicción errónea. Una vez se ha generado el valor del PC, este se almacena en el registro de pipeline PC/IF junto a una bandera que indica si dicho valor es el resultado de una predicción. La figura 5.3 presenta esta etapa del pipeline en detalle.

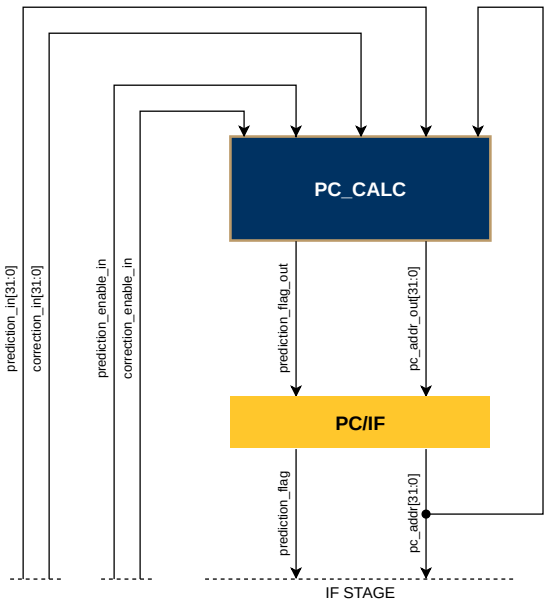

Figura 5.3: Diagrama con las señales asociadas a la primera etapa del pipeline (PC).

La segunda etapa (IF) se encarga de recuperar la instrucción de la memoria y de realizar la predicción de los saltos. Para recuperar la instrucción de la memoria correspondiente se utiliza el protocolo definido para este fin, donde el comportamiento de las señales de control (validez y disponibilidad) es implementado en el control de pipeline. El uso de este protocolo permite que los accesos a memoria puedan tener una latencia arbitraria sin que se afecte la operación del pipeline. Por otra parte, a partir de la dirección del PC se realiza la predicción en el módulo correspondiente de un salto condicional. El registro de pipeline IF/DEC recibe el valor del contador de programa, el dato correspondiente a la instrucción y la bandera de la predicción. La figura 5.4 presenta un diagrama de la segunda etapa del pipeline.

La tercera etapa (DEC), presentada detalladamente en la figura 5.5, es donde se realiza la decodificación de la instrucción para su ejecución. De forma paralela, en esta etapa se realizan tres operaciones: decodificación de instrucción y generación de señales de control, decodificación del valor inmediato y adelantamiento de datos. La decodificación de instrucción genera las señales de control necesarias para la ejecución en el resto del pipeline. Algunas de las señales que se generan son la micro-operación para la unidad de ejecución, la activación de multiplexores para usar valores inmediatos o registros fuentes y la activación de escritura en el registro destino. Por su parte, la decodificación del valor inmediato busca recuperar dicho valor de la codificación de la instrucción. Dependiendo del código de operación de la instrucción se genera el valor

inmediato con un ancho de 32 bits según el formato correspondiente. La tercera y última operación corresponde al adelantamiento de datos. Esta operación genera las señales de control necesarias para, en caso de ser necesario, usar el valor que será escrito en los registros directamente.

La cuarta etapa (REG) corresponde a la lectura de los registros y la emisión de instrucción. La lectura de los registros se hace según los valores codificados en la instrucción, sin importar si estos van a ser reemplazados mediante el adelantamiento de datos. Por otra parte, hay una parte de la lógica dedicada a la emisión de instrucción, donde se asigna una operación a las unidades de ejecución. Si bien esta lógica podría ser realizada directamente en la decodificación, esta organización permite que en desarrollos futuros se pueda implementar alguna clase de planeación para soportar ejecución concurrente. Los módulos y señales de esta etapa se ilustran en la figura 5.6.

La quinta etapa del pipeline (EX) es donde se lleva a cabo a la ejecución de las operaciones codificadas en la instrucción. Las operaciones pueden ser ejecutadas en una de tres unidades de ejecución: la unidad de enteros, la unidad de carga/almacenamiento y la unidad de resolución de ramas. En conjunto con las unidades de ejecución existe lógica combinacional que permite usar datos adelantados y seleccionar entre los valores de los registros, el contador de programa o el valor inmediato. La unidad de enteros es una ALU (Unidad aritmética lógica, *Arithmetic Logic Unit*) donde se llevan a cabo la mayoría de operaciones binarias entre números enteros. La unidad de carga/almacenamiento se encarga de los accesos a la memoria de datos tanto a nivel de comunicación como a nivel de cálculo de la dirección efectiva de acceso. En el caso de la unidad de resolución de ramas, esta se encarga de evaluar las sentencias lógicas de saltos condicionales, verificar si la predicción de estas fue apropiada y generar las señales de retroalimentación para el predictor. El diagrama de la figura 5.7

La sexta y última etapa del pipeline es la escritura en registros (WB). A partir de los valores almacenados en el registro de pipeline (EX/WB) se determina el comportamiento de la lógica de la etapa y la escritura en los registros. La lógica de esta etapa determina si el valor escrito en los registros es el valor calculado por una unidad de ejecución o el valor del contador de programa para la instrucción siendo ejecutada incrementado en cuatro unidades. Por lo demás, el valor es escrito en los registros de propósito general en caso de ser requerido.

5.2. Protocolos de Comunicación

En esta sección se presenta la definición de los protocolos de comunicación usados en el Core101 para llevar a cabo los accesos a las memorias tanto de datos como de instrucciones. El objetivo principal de los protocolos definidos es dotar al núcleo con la capacidad de acceder a las memorias de forma estructurada y no bajo supuestos de retrasos. Si bien los dos protocolos de acceso son similares, estos tienen diferencias que radican en el hecho de asumir la memoria de instrucción como un elemento de lectura y la memoria de datos como uno de lectura y escritura.

5.2.1. Accesos a la Memoria de Instrucción

Una de las operaciones que debe llevar a cabo el núcleo Core101 para ejecutar un programa es recuperar las instrucciones de la memoria correspondiente. Los accesos deben tener en cuenta la posible latencia que exista entre la solicitud de una instrucción y la respuesta de la memoria. Por este motivo se debe definir un protocolo que permita lidiar con los posibles retrasos en los accesos a memoria. Al definir este protocolo se

busca que el núcleo opere según condiciones de la ejecución y no bajo supuestos que pueden comprometer la integridad de la ejecución.

El protocolo que se ha definido para los accesos a la memoria de instrucción utiliza dos señales principales, correspondientes al bus de dirección y el bus de datos, y dos señales de validación, que validan tanto la dirección como el dato. Un resumen de las señales utilizadas por el protocolo se presenta en la tabla 5.1. Las dos señales principales utilizan el ancho de datos definido por la arquitectura (XLEN), que en el caso del Core101 es de 32 bits. Por su parte, las señales de validación tienen un ancho de un bit.

Señal	Ancho	Descripción
ins_mem_addr_out	XLEN	Dirección del acceso
ins_mem_valid_out	1 bit	Señal de validez
ins_mem_data_in	XLEN	Entrada de datos
ins_mem_ready_in	1 bit	Señal de disponibilidad

Tabla 5.1: Descripción de las señales usadas en el protocolo de los accesos a la memoria de instrucción.

La pareja comprendida por el bus de direcciones y la señal de validez se encargan de solicitar a la memoria que se acceda a una dirección. Para que la memoria acceda a una dirección, esta debe verificar que la señal de validez esté activa. La respuesta de la memoria está comprendida por la pareja del bus de datos y la señal de disponibilidad. De forma similar a la señal de validez, el núcleo debe verificar que esta señal esté activa para completar la transacción y que el dato sea leído. La figura 5.9 presenta un acceso normal conforme al protocolo definido en el cual no se presenta retraso alguno.

Si se llegase a presentar un retraso, la señal de disponibilidad no estaría activada, por lo que el núcleo deberá esperar hasta que este se active para completar la transacción. Este comportamiento se presenta en la figura 5.10. Si durante un retraso la dirección a la cual se desea acceder cambia, esta deberá se acompañada de una señal de validez desactivada por un ciclo para indicar que el acceso es cancelado. Posteriormente, se podrá activar la señal de validez para completar la transacción con la nueva dirección. Esta situación se presenta en la figura 5.11.

5.2.2. Accesos a la Memoria de Datos

Los accesos a la memoria de datos utilizan un protocolo basado en el de accesos a la memoria de instrucción. No obstante, el protocolo de los accesos a la memoria de datos presenta diferencias significativas. La primera diferencia consiste en las señales utilizadas, tanto en cantidad como en comportamiento. El cambio en el número de señales utilizadas se debe al hecho que los accesos a la memoria de datos son bidireccionales y pueden acceder a datos de distintos anchos. Esta misma lógica aplica para el cambio de comportamiento en el protocolo.

Para el protocolo de acceso a la memoria de datos se agrega una señal del código de operación y un bus de datos de salida al bus de dirección. El bus de salida de datos se usa como el puerto de salida para la escritura en la memoria. Por su parte, el código de operación codifica si se está realizando una lectura o una escritura y el ancho de datos correspondiente. El protocolo permite acceder a datos con tres anchos diferentes: palabras, medias palabras y bytes. Un resumen de las señales usadas en el protocolo se define en la tabla 5.2. Los buses de dirección y datos siguen usando el ancho de datos de la arquitectura (XLEN), que en el caso del Core101 es de 32 bits.

Señal	Ancho	Descripción
data_mem_addr_out	XLEN	Dirección
data_mem_data_out	XLEN	Salida de datos
data_mem_valid_out	1 bit	Señal de validez
data_mem_code_out	3 bit	Código de operación
data_mem_data_in	XLEN	Entrada de datos
data_mem_ready_in	1 bit	Señal de disponibilidad

Tabla 5.2: Descripción de las señales usadas en el protocolo de los accesos a la memoria de datos.

Para realizar una lectura en la memoria de datos se debe cargar la dirección a la cual se desea acceder en el el bus correspondiente, activar la señal de validez y asignar el código de operación correspondiente. Una vez la memoria verifique las señales y recupere el dato, esta lo cargará en el bus de entrada de datos y habilitará la señal de disponibilidad. Un diagrama de tiempos para la lectura de una dirección en la memoria de datos se presenta en la figura 5.12. La escritura en la memoria sigue un procedimiento similar, salvo que se debe cargar el bus de salida de datos. La memoria activará la señal de disponibilidad una vez se haya escrito el dato en esta. En la figura 5.13 se presenta el diagrama de tiempos para una escritura en la memoria.

Si la memoria está realizando una lectura y la respuesta de la memoria presenta algún retraso, el procesador deberá esperar hasta recibir la señal de disponibilidad para completar la transacción. Este comportamiento se presenta en la figura 5.14. De forma similar, para una escritura la transacción se mantendrá hasta recibir la señal de disponibilidad, como se muestra en la figura 5.15. Al contrario del comportamiento del protocolo de lectura de la memoria de instrucción, en el protocolo de la memoria de datos no se podrá cambiar la dirección y/o el dato durante la transacción. Esto se hace para evitar problemas de integridad de la memoria.

5.3. Interfaz del Núcleo

La tabla 5.3 presenta la descripción de las señales de entrada y salida del núcleo Core101. La tabla hace referencia a la constante XLEN, que corresponde al ancho de datos de la arquitectura que se está utilizando (32 bits).

Señal	**Ancho**	**Descripción**
clock_in	1 bits	Señal de reloj
reset_in	1 bits	Señal de reinicio
ins_mem_valid_out	1 bit	Señal de validez para la memoria de instrucción
ins_mem_addr_out	XLEN	Bus de dirección para la memoria de instrucción
ins_mem_ready_in	1 bit	Señal de disponibilidad para la memoria de instrucción
ins_mem_data_in	XLEN	Bus de datos para la memoria de instrucción
data_mem_valid_out	1 bit	Señal de validez para la memoria de datos
data_mem_code_out	3 bit	Bus de código de operación para la memoria de datos
data_mem_addr_out	XLEN	Bus de dirección para la memoria de datos
data_mem_data_out	XLEN	Bus de datos de salida para la memoria de datos
data_mem_ready_in	1 bit	Señal de disponibilidad para la memoria de datos
data_mem_data_in	XLEN	Bus de datos de entrada para la memoria de datos

Tabla 5.3: Descripción de las señales del núcleo Core101.

En la tabla 5.4 se presentan los rango válidos de las señales, y se indica si se debe tener alguna consideración

respecto a los valores válidos de la señal. Para aquellas señales que utilizan constantes en su definición, se utilizan expresiones en base a dichas constantes.

Señal	Valor Mínimo	Valor Máximo	Consideraciones
clock_in	0	1	Ninguna
reset_in	0	1	Ninguna
ins_mem_valid_out	0	1	Ninguna
ins_mem_addr_out	0	$2^{XLEN} - 1$	Ninguna
ins_mem_ready_in	0	1	Ninguna
ins_mem_data_in	0	$2^{XLEN} - 1$	Ninguna
data_mem_valid_out	0	1	Ninguna
data_mem_code_out	0	7	Ninguna
data_mem_addr_out	0	$2^{XLEN} - 1$	Ninguna
data_mem_data_out	0	$2^{XLEN} - 1$	Ninguna
data_mem_ready_in	0	1	Ninguna
data_mem_data_in	0	$2^{XLEN} - 1$	Ninguna

Tabla 5.4: Rango de las señales del núcleo Core101.

Finalmente, en la tabla 5.5 se relaciona cada señal con el módulo, o módulos, que la utilizan. Dentro de estos módulos no se tienen en cuenta los registros de pipeline sino solo los módulos que hacen uso útil de las señales.

Señal	Tipo	Módulo Relacionado
clock_in	Entrada	CLK
reset_in	Entrada	RESET
ins_mem_valid_out	Salida	INS_MEM
ins_mem_addr_out	Salida	INS_MEM
ins_mem_ready_in	Entrada	PIPELINE_CONTROL
ins_mem_data_in	Entrada	DEC; IMM; FWD;
data_mem_valid_out	Salida	DATA_MEM
data_mem_code_out	Salida	DATA_MEM
data_mem_addr_out	Salida	DATA_MEM
data_mem_data_out	Salida	DATA_MEM
data_mem_ready_in	Entrada	PIPELINE_CONTROL
data_mem_data_in	Entrada	GPR

Tabla 5.5: Relación de las señales del núcleo Core101 con otros módulos.

5.4. Registros de Pipeline

Los registros de pipeline están encargados de almacenar las señales de control y datos que permiten vincular las diferentes etapas de ejecución del pipeline. Cada uno de estos registros almacena diferentes señales conforme a las necesidades de la etapa del pipeline que se alimenta del registro. En algunos casos, estos registros pueden ser reiniciados para prevenir que la ejecución de instrucciones no deseadas altere erróneamente el estado del procesador. Tanto la habilitación de escritura como el reinicio de los registros de pipeline está determinado por el control de pipeline. A continuación se describe la organización de los registros de pipeline a nivel interno.

En la tabla 5.6 se presenta la organización de las señales para el registro de pipeline ubicado entre la

etapa de generación del valor del contador de programa y la etapa de búsqueda de instrucción (PC/IF). Este registro de pipeline tiene un ancho de 33 bits.

Bit Inicio	Bit Final	Descripción
0	31	Valor del PC
32	–	Bandera de predicción

Tabla 5.6: Organización y descripción de las señales para el registro de pipeline PC/IF.

En la tabla 5.7 se presenta la organización de las señales para el registro de pipeline ubicado entre la etapa de búsqueda de instrucción y la etapa de decodificación (IF/DEC). Este registro de pipeline tiene un ancho de datos de 66 bits.

Bit Inicio	Bit Final	Descripción
0	31	Valor del PC
32	63	Dato de la instrucción
64	–	Bandera de predicción
65	–	Bandera de predicción propia

Tabla 5.7: Organización y descripción de las señales para el registro de pipeline IF/DEC.

En la tabla 5.8 se presenta la organización de las señales para el registro de pipeline ubicado entre la etapa de decodificación de instrucción y la etapa de lectura de registros y emisión de instrucción (DEC/REG). Este registro de pipeline tiene un ancho de datos de 95 bits.

Bit inicio	Bit final	Descripción
0	31	Contador de programa (PC)
32	63	Valor inmediato (IMM)
64	68	Registro destino (*rd*)
69	73	Primer registro fuente (*rs1*)
74	78	Segundo registro fuente (*rs2*)
79	82	Código de microinstrucción
83	86	Selección de la unidad de ejecución
87	88	Control de datos adelantados
89	–	Selección de *rs1* ó PC
90	–	Selección de *rs2* ó IMM
91	–	Activación de escritura en *rd*
92	–	Selección de dato ó PC + 4
93	–	Selección de salto no condicional
94	–	Predicción del salto

Tabla 5.8: Organización y descripción de las señales para el registro de pipeline ID/IS.

En la tabla 5.9 se presenta la organización de las señales para el registro de pipeline ubicado entre la etapa de lectura de registros y emisión de instrucción y la etapa de ejecución de instrucción (REG/EX). Este registro de pipeline tiene un ancho de datos de 161 bits.

Finalmente, en la tabla 5.10 se introduce la organización de las señales para el registro de pipeline ubicado entra las etapas de ejecución y escritura en registros (EX/WB). Este registro de pipeline tiene un ancho de datos de 71 bits.

Bit inicio	Bit final	Descripción
0	31	Contador de programa (PC)
32	63	Valor inmediato (IMM)
64	95	Valor del primer registro fuente (*rs1*)
96	127	Valor del segundo registro fuente (*rs2*)
128	132	Dirección del registro destino (*rd*)
133	136	Microinstrucción para la unidad de resolución de ramas
137	140	Reservado
141	144	Microinstrucción para la unidad de carga/almacenamiento
145	148	Microinstrucción para la unidad de ejecución de enteros
149	150	Control de datos adelantados
151	–	Bit de activación de la unidad de resolución de ramas
152	–	Reservado
153	–	Bit de activación de la unidad de carga/almacenamiento
154	–	Bit de activación de la unidad de ejecución de enteros
155	–	Selección entre *rs1* ó PC
156	–	Selección entre *rs2* ó IMM
157	–	Activación de escritura en *rd*
158	–	Selección entre dato ó PC + 4
159	–	Bit de selección para el multiplexor de saltos
160	–	Predicción de salto

Tabla 5.9: Organización y descripción de las señales para el registro de pipeline IS/EX.

Bit inicio	Bit final	Descripción
0	31	Contador de programa (PC)
32	63	Resultado de la etapa de ejecución
64	68	Dirección de registro destino (*rd*)
69	–	Activación de escritura en *rd*
70	–	Selección entre dato o PC + 4

Tabla 5.10: Organización y descripción de las señales para el registro de pipeline EX/WB.

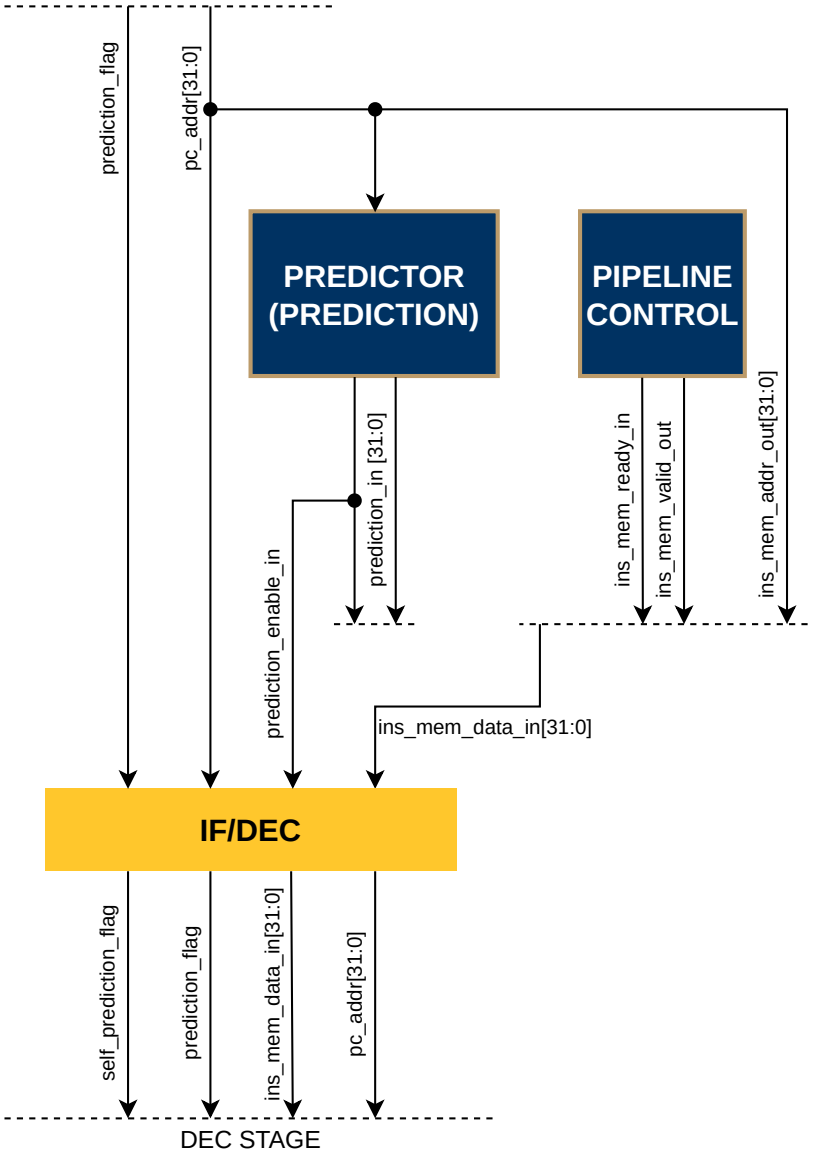

Figura 5.4: Diagrama con las señales asociadas a la segunda etapa del pipeline (IF).

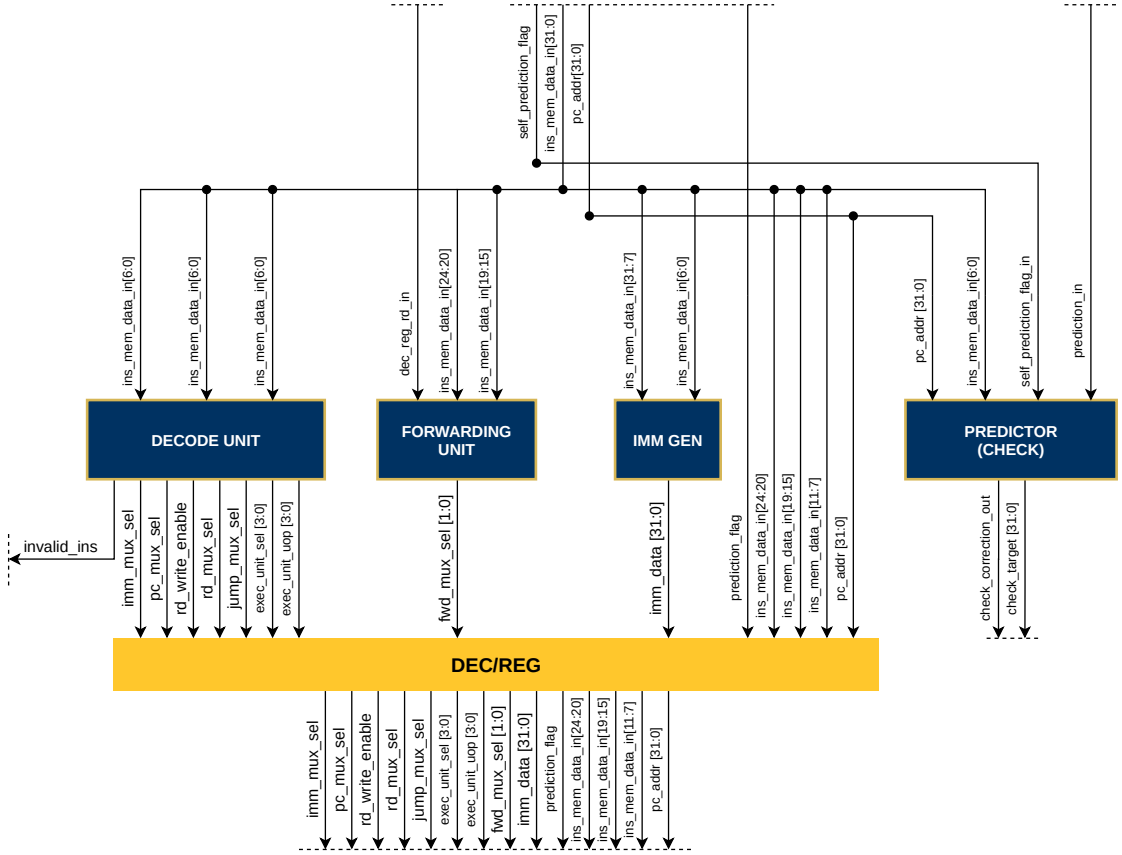

Figura 5.5: Diagrama con las señales asociadas a la tercera etapa del pipeline (DEC).

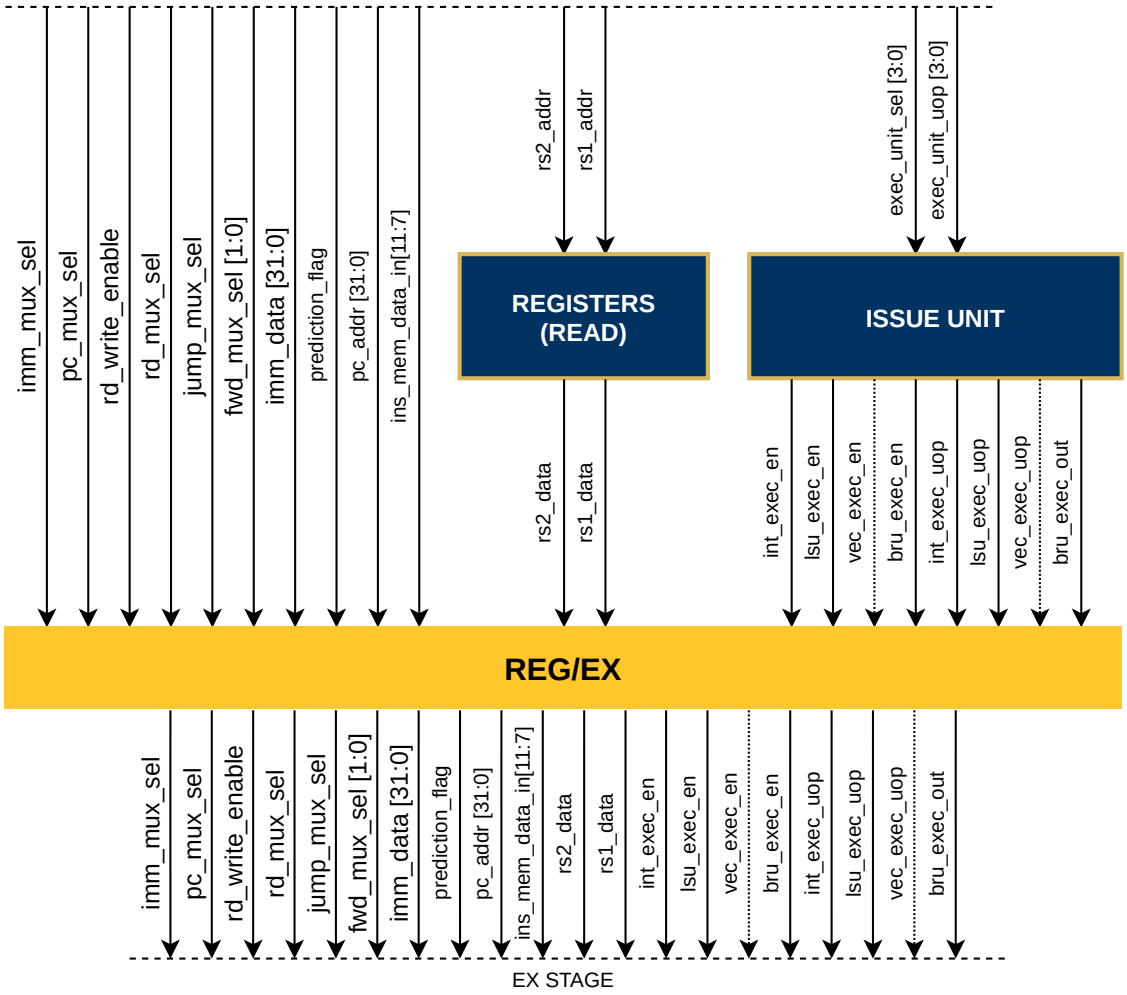

Figura 5.6: Diagrama con las señales asociadas a la cuarta etapa del pipeline (REG).

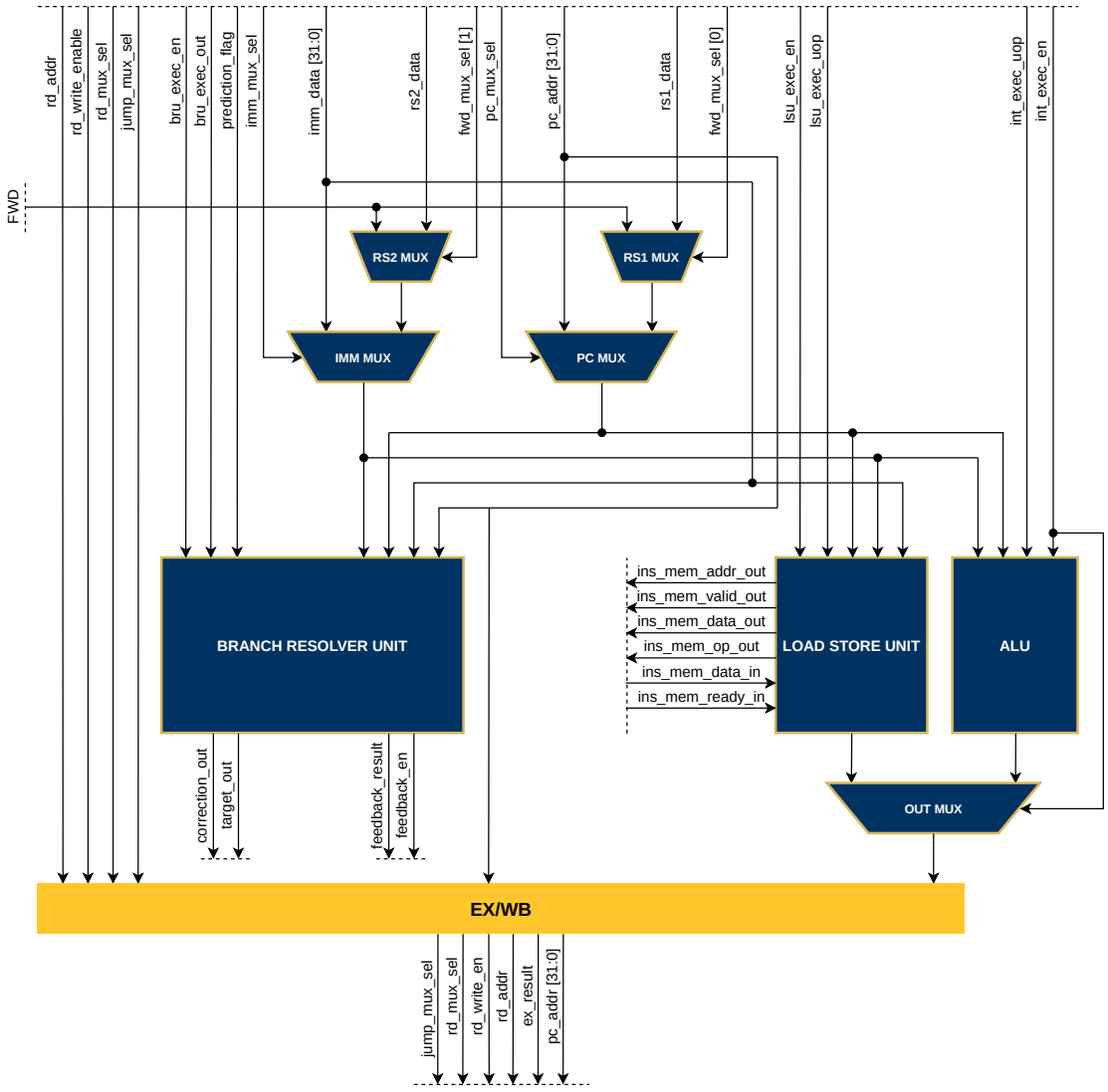

Figura 5.7: Diagrama con las señales asociadas a la quinta etapa del pipeline (EX).

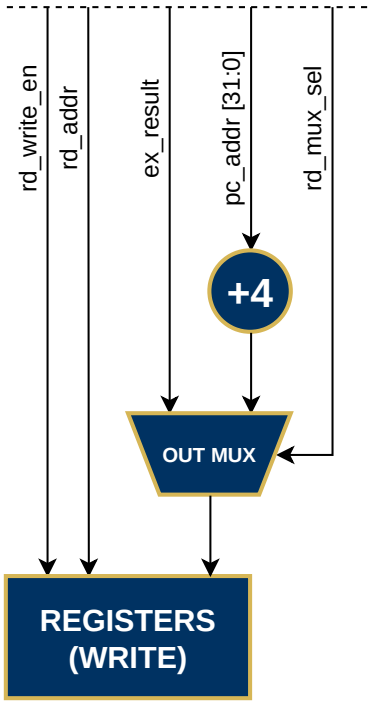

Figura 5.8: Diagrama con las señales asociadas a la sexta etapa del pipeline (WB).

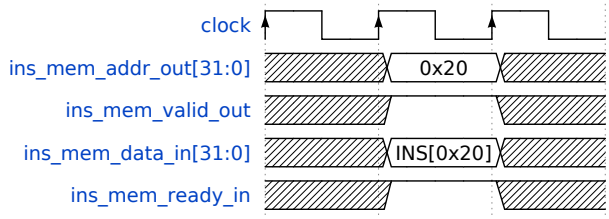

Figura 5.9: Comportamiento del protocolo para un acceso normal a la memoria de instrucción.

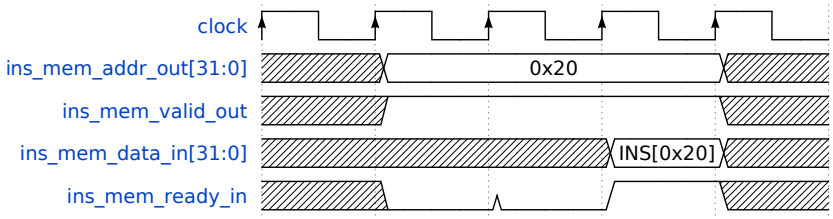

Figura 5.10: Comportamiento del protocolo para un acceso con retraso a la memoria de instrucción.

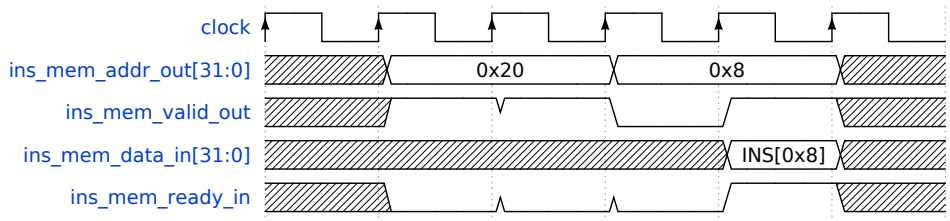

Figura 5.11: Comportamiento del protocolo para un acceso con cambio de dirección a la memoria de instrucción.

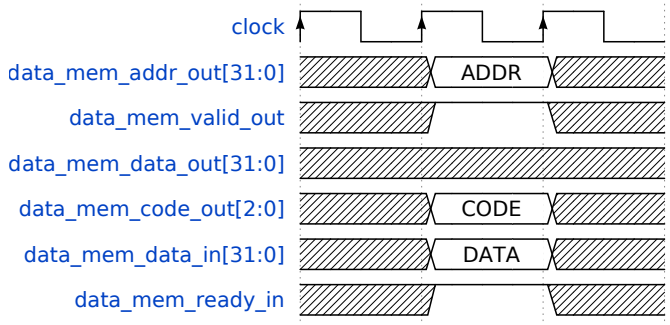

Figura 5.12: Comportamiento del protocolo para la lectura de la memoria de datos

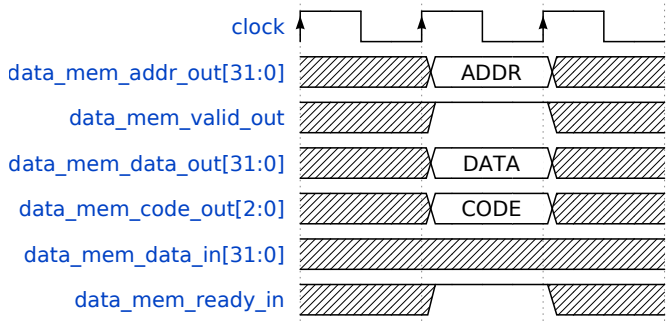

Figura 5.13: Comportamiento del protocolo para la escritura en la memoria de datos.

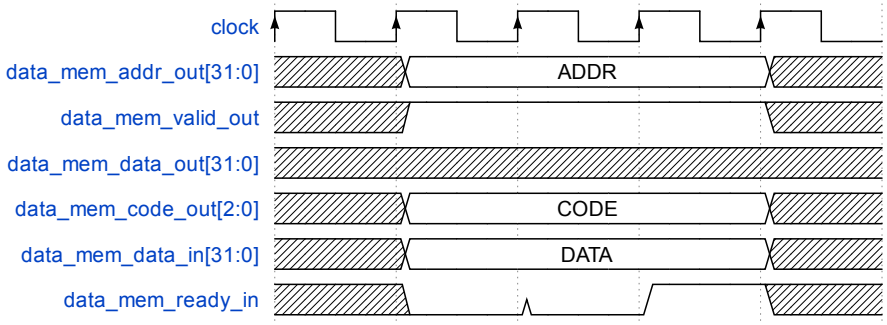

Figura 5.14: Comportamiento del protocolo para una lectura de la memoria de datos con retraso.

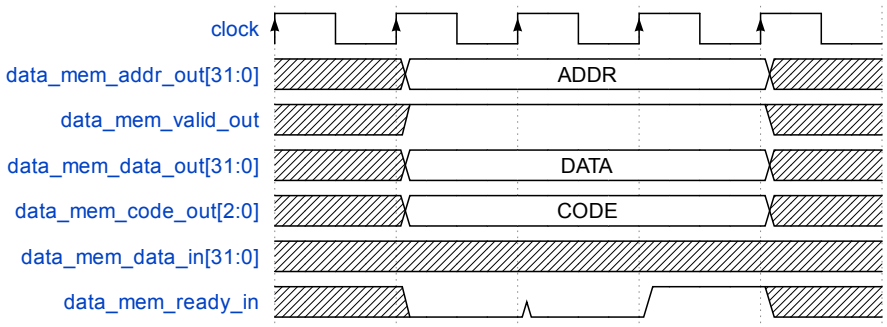

Figura 5.15: Comportamiento del protocolo para escritura en la memoria de datos con retraso.

Capítulo 6

Implementación

Este capitulo presenta los detalles del proceso de implementación así como las actividades que se derivan de la misma. La implementación, según lo definido en la metodología, se realiza en un lenguaje de descripción de hardware y en un lenguaje de programación de alto nivel. Para la implementación se usa Verilog, como lenguaje de descripción de hardware, y Python, como lenguaje de programación de alto nivel. El objetivo de realizar un implementación en Verilog es que sirva para su despliegue en FPGA, mientras que la implementación en Python tiene como objetivo servir de referencia para el proceso de validación.

6.1. Lenguaje de Alto Nivel

La implementación en un lenguaje de alto nivel tiene como objetivo principal servir de referencia para la validación del núcleo Core101. Esta implementación fue realizada en el lenguaje de programación Python con el fin de acelerar su desarrollo. Ahora, si bien el uso de Python favorece el tiempo de desarrollo de esta implementación, presenta el inconveniente de no tener un entorno de desarrollo o librerías para simular o emular hardware. No obstante, dentro del desarrollo del proyecto fue posible construir el marco de desarrollo para la emulación de hardware. La figura 6.1 presenta un diagrama UML del objeto base para implementar módulos de hardware en Python.

En la figura 6.1 se presenta el diagrama UML de la clase `Module` cuyo objetivo es generalizar la implementación de módulos de hardware en Python. La clase `Module` tiene cuatro atributos correspondientes a una cadena de texto para el nombre del módulo (`name`), una lista con los puertos de entrada (`inputs`), una lista con los puertos de salida (`outputs`) y una lista con los archivos de memoria del módulo (`memories`). Los puertos de entrada son almacenados como diccionarios que contienen el nombre del puerto, su valor mínimo, su valor máximo y las restricciones que le aplican, que pueden ser los valores válidos de los campos *opcode*, *funct3* y *funct7*. Los métodos de la clase `Module` se presentan a continuación:

- `execute`: recibe como entrada un diccionario (`inputs`) cuyas llaves son el nombre de los puertos de entrada y se encuentran asociados a los valores que llevan estos. Este es un método virtual en el sentido que debe ser implementado por las clases que hereden la clase `Module`, o de lo contrario levantará una excepción de tipo `NotImplementedError`.
- `add_input`: recibe como entrada el nombre del puerto (`name`), su ancho de bits (`width`) y si presenta

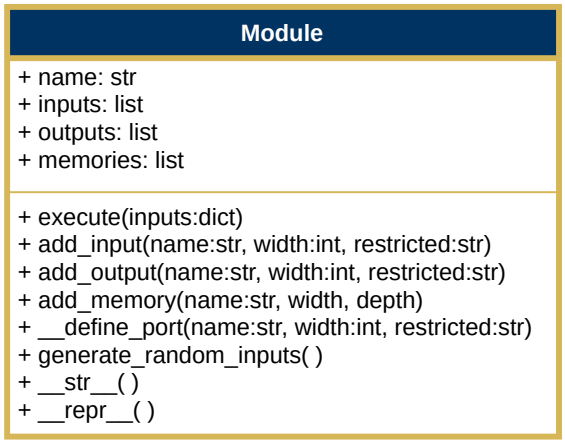

Figura 6.1: Diagrama UML del objeto base para la emulación de hardware.

alguna restricción (`restricted`). Este método agrega el diccionario del puerto, creado mediante el método `__define_port`, a la lista de entradas.

- `add_output`: recibe como entrada el nombre del puerto (`name`), su ancho de bits (`width`) y si presenta alguna restricción (`restricted`). Este método agrega el diccionario del puerto, creado mediante el método `__define_port`, a la lista de salidas.
- `add_memory`: recibe como entrada el nombre de la memoria (`name`), su ancho de bits (`width`) y su profundidad (`depth`). Este método agrega una definición de memoria a la lista correspondiente de la clase. Una memoria se define como un diccionario con el nombre, una lista cuyo número de entradas es definido según la profundidad, y los valores mínimos y máximos según el ancho de bits.
- `__define_port`: recibe como entrada el nombre del puerto (`name`), su ancho de bits (`width`) y si presenta alguna restricción (`restricted`). Este método crea el diccionario con los datos del puerto y lo retorna.
- `generate_random_inputs`: crea y retorna un diccionario con entradas aleatorias según los puertos almacenados en la lista correspondiente.
- `__str__` y `__repr__`: retornan el nombre del módulo cuando se debe representar la clase.

A partir de la definición de la clase `Module` es posible crear la representación de cualquier módulo de hardware en Python. Para representar un módulo se debe construir una clase que herede la clase `Module`, definir sus entradas, salidas y memorias en el constructor, e implementar el comportamiento del mismo en el método `execute`. Las representaciones en Python de los módulos de la microarquitectura del Core101, presentados en el apéndice correspondiente, son asociados a las descripciones funcionales del capitulo de diseño. Para discutir la implementación específica de los módulos en Python se va a tener como referencia la descripción del módulo `IMM_GEN` en el listado 6.1.

```
class IMM_GEN(Module):

  def __init__(self):

    # Initalizes parent class
    Module.__init__(self, "IMM_GEN")
```

```
    # Adds inputs to module
    self.add_input("opcode_in", 7, "OPCODE")
    self.add_input("instruction_in", 25)

    # Adds output to module
    self.add_output("immediate_out", 32)


  def execute(self, inputs):
    """ Defines the behaviour of IMM_GEN modules.

    Args:
      inputs (dict): dictionary with values for inputs.

    Returns:
      dict: dictionary with values for outputs

    """

    # Retrieves opcode from inputs
    opcode = inputs["opcode_in"]

    # Converts instruction (w/o opcode) to binary for bit manipulation
    bin_ins = bin(int(inputs["instruction_in"]))[2:].zfill(25)

    # Depending on opcode, the IMM_VALUE is generated.
    if opcode in [3, 19, 103]:

      # Creates sign extension for I instruction format
      sign_extension = ""

      # Generates bits for sign extension length
      for i in range(21):
        sign_extension += bin_ins[0]

      # Retrieves value from binary instruction
      value = bin_ins[25-24:25-13]

      # Generates immediate
      imm_str = sign_extension + value

    elif opcode in [35]:

      # Creates sign extension for S instruction format
      sign_extension = ""

      for i in range(21):
        sign_extension += bin_ins[0]

      # Retrieves value from binary instruction
      value0 = bin_ins[1:7]
      value1 = bin_ins[20:]

      # Generates immediate
```

```
    imm_str = sign_extension + value0 + value1

  elif opcode in [99]:

    # Creates sign extension for B instruction format
    sign_extension = ""

    for i in range(20):
      sign_extension += bin_ins[0]

    # Retrieves value from binary instruction
    value0 = bin_ins[24]
    value1 = bin_ins[1:7]
    value2 = bin_ins[20:24]

    # Generates immediate
    imm_str = sign_extension + value0 + value1 + value2 + '0'


  elif opcode in [23, 55]:

    # Creates sign extension for U instruction format
    value = bin_ins[0:25-5]

    z_fill = ""

    for i in range(12):
      z_fill += '0'

    #
    imm_str = value + z_fill

  elif opcode in [111]:

    # Creates sign extension for J instruction format
    sign_extension = ""

    for i in range(12):
      sign_extension += bin_ins[0]

    # Retrieves value from binary instruction
    value0 = bin_ins[12:20] # 8
    value1 = bin_ins[11]  # 1
    value2 = bin_ins[1:11]  # 10

    # Generates immediate
    imm_str = sign_extension + value0 + value1 + value2 + '0'

  else:

    # Default immediate is zero
    imm_str = ""

    # R instruction format
```

```
    for i in range(32):

      imm_str += '0'

  imm_value = int(imm_str, 2)

  # Adds output to dictionary
  outputs = {
    "immediate_out": imm_value
  }

  # Returns outputs
  return outputs
```

Listing 6.1: Implementación en alto nivel para el módulo de generación de inmediatos.

La principal característica de la clase `IMM_GEN` es la herencia de la clase `Module`. La herencia se define en la definición de la clase y en la inicialización de la superclase en el método constructor. De igual modo, dentro del método constructor se definen dos puertos de entrada y uno de salida. Los puertos de entrada son el código de operación de la instrucción, que tiene siete (7) bits de ancho y está restringido, y el resto de la instrucción, con un ancho es de veinticinco bits (25) y sin restricciones. El puerto de salida es el valor inmediato ya decodificado con un ancho de treinta y dos (32) bits. Otra característica de la clase `IMM_GEN` es el método `execute` que define el comportamiento del módulo.

La implementación de módulos de hardware como se ha presentado busca aprovechar las características de la programación orientada a objetos y reflejar en la mayor medida posible las características del hardware. La principal característica de la programación orientada a objetos es la herencia que se da entre los módulos y la superclase `Module`. Otra característica es la capacidad de utilizar múltiples instancias de un módulo como suele suceder con multiplexores y otros elementos lógicos básicos. Algunas características del hardware que no fueron implementadas fueron la falta de sincronización mediante una señal temporal de reloj y la operación directa sobre valores binarios.

Una vez definida la implementación de los módulos de la microarquitectura, se procede con la implementación de la microarquitectura para consolidar el Core101. La implementación del Core101 instancia los módulos del pipeline y define el método `execute` para simular la ejecución de un programa. Para cada módulo se definen dos diccionarios: uno con las entradas del módulo y otro las salidas. Adicionalmente se definen diccionarios que representan los registros de pipeline de la microarquitectura. La simulación del programa utiliza un diccionario organizado de modo tal que la llave es una dirección de memoria y el valor una cadena de texto que representa la instrucción en hexadecimal.

Para simular la ejecución de un programa se define el diccionario que lo codifica y un número de ciclos a simular. La simulación itera por el número de ciclos especificado realizando las siguientes operaciones: propagación de las señales por los módulos, actualización de registros de pipeline y actualización de las entradas de los módulos. En la primera operación, la propagación de señales por los módulos, se obtienen las salidas de los módulos para las entradas que son almacenadas en los diccionarios correspondientes. Posteriormente, usando los diccionarios de las salidas, se actualizan los diccionarios de los registros de pipeline. Finalmente, en base a los valores almacenados en los registros de pipeline, se actualizan las entradas de los módulos. Si la simulación va a ser usada como referencia, se genera un archivo con los valores de los

registros de pipeline, es decir, con las señales combinadas y organizadas como irían en el hardware.

6.2. Lenguaje de Descripción de Hardware

Para la implementación en lenguaje de descripción de hardware (HDL, *Hardware Descripction Language*) se utiliza Verilog. Verilog permite modelar el comportamiento definido en las descripciones funcionales y desplegar el diseño en una tarjeta FPGA. Para la implementación en Verilog se tuvieron en cuenta tres consideraciones: mantener la modularidad en la implementación, utilizar un estilo estructurado y no comportamental, y mantener el código legible. La modularidad en la implementación consiste en desarrollar módulos que sean conectados posteriormente mediante instanciaciones en un módulo de mayor nivel. Del mismo modo, se mantuvo un estilo estructurado en todos los módulos desarrollados como parte del proyecto. El estilo estructurado se usó ya que permite la sintetización y despliegue en FPGA. Finalmente, la legibilidad del código se logra mediante la combinación del código estructurado y la inclusión de comentarios.

Otro uso dado al HDL Verilog fue la implementación de bancos de prueba para los módulos de la microarquitectura y el módulo del Core101. Estos bancos de prueba fueron usados para la validación mediante la simulación basada en eventos y su comparación con los resultados de referencia. Para la construcción de estos bancos de prueba se definía la instanciación del módulo bajo prueba y se leían dos archivos: uno con las entradas para el módulo y otro con los resultados de referencia. Las señales de entrada eran evaluadas y comparadas con los resultados de referencia cuyo resultado era escrito en un tercer archivo. Mediante una rutina en Python era posible obtener métricas de la validación.

6.3. Despliegue en FPGA

La última etapa de implementación corresponde al despliegue en FPGA del núcleo en las dos tarjetas de desarrollo consideradas. Se consideran dos tarjetas como objetivo debido a las diferencias en la filosofía de diseño seguida para cada una. Por una parte, se tiene la tarjeta *TinyFPGA BX* orientada a promover la adopción de herramientas de código abierto para la sintetización y despliegue de diseños de hardware. Por la otra, se tiene el kit de evaluación de las FPGAs Cyclone 10 LP de Intel. De forma general, las tarjetas de desarrollo y kits de evaluación de Intel buscan servir como base para desarrollar aplicaciones comerciales de FPGAs mediante herramientas propietarias. Uno de los principales beneficios de usar FPGAs de código abierto y propietarias es evaluar cómo influyen las herramientas para la programación y evaluación de los diseños.

6.3.1. TinyFPGA BX

La tarjeta de desarrollo TinyFPGA BX es el resultado de un proyecto que ha buscado desarrollar tarjetas FPGA de bajo costo, de código abierto y en factores de forma pequeños. La tarjeta TinyFPGA BX está pensada para ser usada con herramientas de síntesis y programación de código abierto como Yosys, Arachne PnR y el proyecto IceStorm. Las especificaciones técnicas de la tarjeta TinyFPGA BX se presentan en la tabla 6.1.

Especificación	Valor
FPGA	ICE40LP8K
Celdas lógicas	7680
RAM	128 Kbit
Flash	6000 Kbit
PLLs	1
GPIOs	31

Tabla 6.1: Especificaciones de la tarjeta de desarrollo TinyFPGA BX.

6.3.2. Cyclone 10 Evaluation Kit

El kit de evaluación de las FPGAs Cyclone 10 es una tarjeta de desarrollo de Intel para su serie de FPGAs Cyclone en para la generación 10. La tarjeta de desarrollo utiliza las herramientas del software de diseño Quartus. Este software de diseño dispone de herramientas para sintetizar, programar y evaluar los diseños para las FPGAs de Intel. Las especificaciones técnicas del kit de evaluación Cyclone 10 se presentan en la tabla 6.2.

Especificación	Valor
FPGA	10CL025YU256I7G
Celdas lógicas	25K
RAM	128 Mbit
Flash	128 Mbit
PLLs	4
GPIOs	36

Tabla 6.2: Especificaciones del kit de evaluación Cyclone 10.

Es posible evaluar el despliegue del diseño dependiendo de la frecuencia de operación de los módulos y el uso de los recursos físicos de la FPGA. Para el caso del Core101, la tabla 6.3 presenta las métricas de uso de recursos y frecuencia máxima de operación. En primer lugar, vale la pena notar que estas métricas son generadas para las dos tarjetas de desarrollo consideradas. En segundo lugar, las métricas son presentadas para cada uno de los módulos que componen el diseño del procesador. Es importante resaltar que no es posible comparar las dos tarjetas de desarrollo ya que estas constituyen dos tecnologías diferentes.

Modulo	TinyFPGA BX			Cyclone 10 Evaluation Kit		
	Elementos Lógicos	Porcentaje de Utilización	Frecuencia Máxima	Elementos Lógicos	Porcentaje de Utilización	Frecuencia Máxima
PC_CALC	122	1.68 %	56.19 MHz	95	1.79 %	125 MHz
PREDICTOR (PRED)	3106	42.69 %	56.95 MHz	2096	39.54 %	125 MHz
PREDICTOR (CHECK)	61	0.83 %	72.75 MHz	32	0.60 %	125 MHz
DECODER	59	0.81 %	110.73 MHz	19	0.36 %	125 MHz
IMM_GEN	48	0.65 %	59.84 MHz	75	1.41 %	125 MHz
FWD	10	0.14 %	128.36 MHz	10	0.19 %	125 MHz
REGISTERS	2467	33.91 %	43.74 MHz	1329	25.07 %	125 MHz
ISSUE	20	0.27 %	161.86 MHz	20	0.38 %	125 MHz
BRU	384	5.28 %	35.04 MHz	254	4.79 %	125 MHz
ALU	681	9.36 %	38.17 MHz	635	11.98 %	125 MHz
LSU	93	1.28 %	72.16 MHz	77	1.45 %	125 MHz
PIPELINE CONTROL	9	0.12 %	210.91 MHz	6	0.11 %	125 MHz

Tabla 6.3: Conteo de celdas lógicas, porcentaje de utilización en relación al conteo de celdas del Core101 y frecuencia máxima de operación para cada uno de los módulos sobre las dos tarjetas de desarrollo FPGA.

Es importante resaltar que los elementos lógicos que más recursos consumen son los módulos que utilizan elementos secuenciales. Esto son los registros y el predictor de saltos condicionales. Estos módulos corresponden a un porcentaje alto de utilización en las dos FPGA de las tarjetas de desarrollo: en el caso de la TinyFPGA BX ocupan más del $75\,\%$ de los recursos utilizados mientras que en el kit de evaluación ocupan cerca del $65\,\%$. En lo que se refiere a los módulos combinacionales, el módulo que más recursos consume es la ALU con un $9{,}36\,\%$ y un $11{,}98\,\%$ de los recursos en la TinyFPGA BX y el kit de evaluación respectivamente. Otro módulo con consumo de recursos considerable es la unidad de resolución de ramas con consumo cercano al $5\,\%$ en las dos tarjetas. Finalmente, vale la pena mencionar que el diseño pudo ser desplegado dentro de los márgenes dados por los recursos disponibles de las tarjetas de desarrollo.

Capítulo 7

Validación

Esta sección presenta el proceso de verificación con sus respectivos resultados tanto a nivel de módulos como de microarquitectura. El proceso de verificación tiene dos propósitos principales: encontrar la mayor cantidad de errores presentes en el diseño y asegurar que el mismo cumple con los requerimientos funcionales del conjunto de instrucción. Para la verificación se va a hacer uso de la implementación en el lenguaje de alto nivel, la descripción en el lenguaje de descripción de hardware y de herramientas disponibles en el software de desarrollo.

7.1. Verificación de Módulos

La verificación de los módulos se lleva a cabo con el fin de validar que la descripción en Verilog sea consistente con el funcionamiento esperado y la implementación en el lenguaje de alto nivel. Como se presentó en la sección de metodología, para validar el funcionamiento de un módulo se realiza una comparación entre la respuesta de implementación en Python y la descripción en Verilog para un mismo conjunto de entradas. Estas entradas son generadas de forma aleatoria en conformidad con los rangos de cada uno de los módulos que componen el procesador. La generación de las entradas aleatorias es responsabilidad de la implementación en el lenguaje de alto nivel, así como su almacenamiento para el uso posterior dado en la descripción de Verilog.

El flujo de verificación para cada módulo se presenta en la figura 7.1. El primer paso realizado en el proceso de validación corresponde a la generación de entradas aleatorias y su evaluación por parte de la implementación en alto nivel. Al finalizar este primer paso se tiene un archivo que almacena las entradas aleatorias del módulo y otro con los resultados generados por la implementación en alto nivel. Tanto el archivo de entradas aleatorias como los resultados generados son leídos por un banco de pruebas que instancia la descripción en Verilog, propaga las entradas en el módulo y compara las salidas con las de la descripción de alto nivel. Esta comparación permite generar un tercer archivo donde se escriben valores binarios dependiendo de si las salidas son iguales o no. Vale la pena notar que este archivo binario contiene una comparación por cada señal de salida del módulo. Finalmente, este archivo es procesado por un *script* de Python que calcula el porcentaje de acierto de las señales, además de indicar cuáles presentaron un error.

Este flujo de evaluación fue seguido para cada uno de los módulos principales que componen la microarquitectura del núcleo. Dentro de cada iteración se procedía a identificar la fuente del error, corregirla y realizar

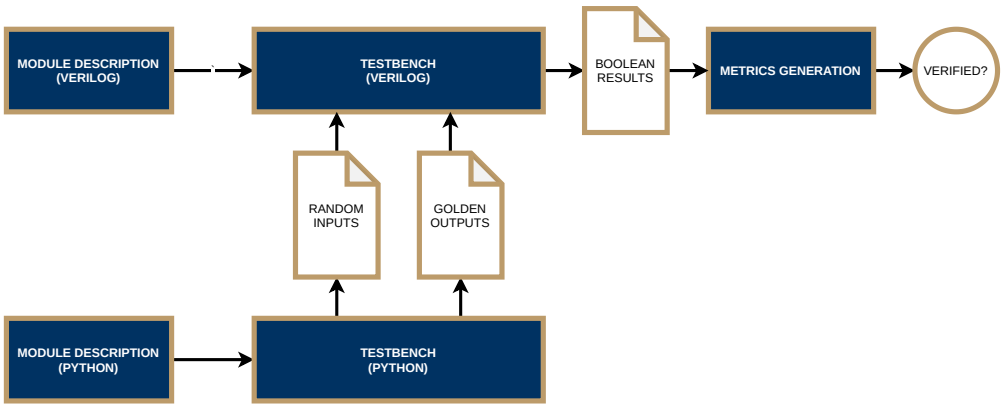

Figura 7.1: Flujo para la verificación de los módulos de la microarquitectura.

la evaluación posterior a la corrección. El ciclo se detenía cuando el módulo presentaba un acierto del $100\,\%$. Los errores identificados en la validación de los módulos fueron de tres clases principalmente: tipográficos, asignación y comportamiento. Los errores tipográficos corresponden a la escritura errónea de nombres de variables o de elementos de la sintaxis del lenguaje de descripción de hardware. Los errores de asignación corresponden a la asignación errónea de valores en las variables y/o salidas de un módulo. Por último, los errores de comportamiento corresponden a que el módulo no describe apropiadamente el comportamiento del módulo.

En términos generales los errores más comunes corresponden a errores tipográficos cuya solución consiste en modificar el nombre la variable y/o salida con el fin de que sea consistente al resto de variables. Por otra parte, los errores de asignación se presentaron de forma esporádica en dos situaciones principales: la asignación de constantes binarias y la concatenación de datos. En el primer caso, fue necesario revisar los valores asignados a las constantes para eliminar las fuentes de error. En el segundo, era necesario revisar el procesamiento que sea hacía al generar datos cuyo origen era la concatenación de otros valores. La situación más común que se daba al presentarse un error de asignación era usar el índice equivocado en la concatenación de valores. El último error consiste en una descripción equivocada del comportamiento del sistema. Esta situación se presentó únicamente en un módulo, específicamente en la unidad de carga/almacenamiento.

Un aspecto importante a tener en cuenta es el tiempo de ejecución del proceso de validación. Esta métrica permite evidenciar que si bien existen contraejemplos, en general Python es más lento en simular los módulos. Por otra parte, es posible evidenciar que los módulos toman más y menos tiempo dependiendo de la complejidad del módulo. Un ejemplo de esta situación es el tiempo de simulación que toma el módulo de decodificación en comparación a la unidad lógico/aritmética. De igual modo, es posible notar que los módulos que contienen registros toman más tiempo en la simulación del lenguaje de alto nivel que en la simulación de Verilog.

7.2. Verificación de la Microarquitectura

Una vez se ha realizado la verificación de los módulos se procede a verificar la microarquitectura a nivel global. En términos generales el proceso es similar a la validación de los módulos salvo porque se evalúa toda la microarquitectura y se usan instrucciones de RISC-V como entradas al modelo. La figura 7.2 presenta el

Módulo	Python	Verilog
PC_CALC	36.0	52
PREDICTOR (PRED)	153.94	58
PREDICTOR (CHECK)	54.13	40
DECODER	95.01	58
IMM_GEN	55.46	26
FWD	24.32	19
REGISTERS	142.55	125
ISSUE	20.20	44
BRU	51.13	70
ALU	26.13	37
LSU	46.67	70
PIPELINE CONTROL	145.11	80

Tabla 7.1: Tiempo de ejecución para simulaciones con cinco millones de pruebas sobre la implementación de Python y Verilog

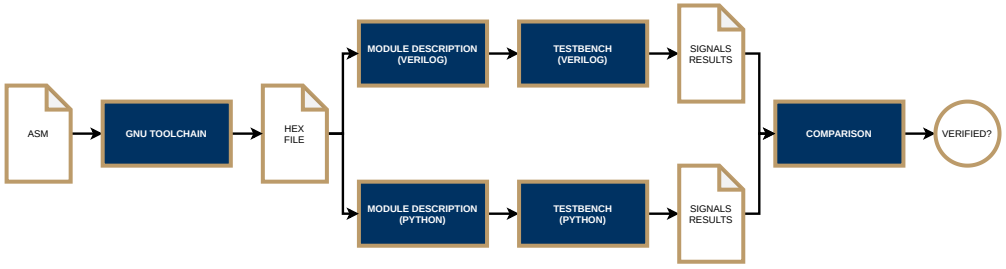

Figura 7.2: Flujo para la verificación de la microarquitectura.

proceso de validación de la microarquitectura. El proceso de validación inicia con un programa que tiene como objetivo evaluar la funcionalidad de una instrucción en la microarquitectura. Para la validación se escribe un programa en ensamblar para cada una de las instrucciones que componen la especificación RV32I de RISC-V. Vale la pena notar que en algunas instrucciones se realiza una verificación extendida debido a que las instrucciones en cuestión son utilizadas frecuentemente por otros programas de validación. Este es el caso para las instrucciones `JAL` (*Jump And Link*, saltar y enlazar), `BEQ` (*Branch on EQual*, tomar rama si es igual) y `ADDI` (*ADD Immediate*, sumar valor inmediato). El listado 7.1 presenta un fragmento del código para el programa de validación para la instrucción `ADDI`. En el caso de las instrucciones con verificación extendida, fueron considerados más casos de evaluación en comparación al resto de instrucciones.

```
# ADDI Validation Program

_preparation:    # Prepares the test
    addi t0, zero, 10
    addi t1, zero, -15
    addi t2, zero, 5

_test:           # Tests the instruction
    add t0, t0, t1
    add t0, t0, t2
    beq zero, t0, _pass
    j _fail

_pass:           # Writes 1 into t6 (x31) if passed
```

```
    addi t6, zero, 1
    j _out

_fail:          # Writes -1 into t6 (x31) if failed
    addi t6, zero, -1

_out:           # Clears the registers
    addi t0, zero, 0
    addi t0, zero, 0
    addi t0, zero, 0
```

Listing 7.1: Program used for testing the $ADDI$ instruction.

Una vez se han definido los programas que serán usados, estos son compilados usando la cadena de herramientas de GNU para RISC-V. Esta cadena de herramientas compila el programa de validación y genera un archivo hexadecimal. Este archivo hexadecimal contiene el equivalente en lenguaje máquina del programa de validación generado para una instrucción en particular. Por otra parte, este archivo hexadecimal es suministrado a la descripción de la microarquitectura en el lenguaje de alto nivel. Esta descripción genera un archivo con los valores de las señales en cada uno de los ciclos de reloj considerados para la simulación. Una vez se ha generado este archivo de señales, se realiza el mismo proceso con la descripción de Verilog para la microarquitectura. Al término de las dos simulaciones se cuenta con dos archivos de los valores de las señales discriminadas por ciclo de reloj. Finalmente, estos archivos son contrastados en un *script* de Python que calcula el porcentaje de acierto además de indicar en qué ciclo y señal se presentaron los errores. Al igual que con la verificación de los módulos, este proceso de validación se repitió hasta que se alcanzara una exactitud del $100\,\%$ para cada una de las instrucciones consideradas. Los resultados de esta validación se presentan en la tabla 7.2.

Inst.	Estado	Inst.	Estado	Inst.	Estado	Inst.	Estado
LUI	V	LB	V	SLTIU	V	SLT	V
AUIPC	V	LH	V	XORI	V	SLTU	V
JAL	E	LW	V	ORI	V	XOR	V
JALR	V	LBU	V	ANDI	V	SRL	V
BEQ	E	LHU	V	SLLI	V	SRA	V
BNE	V	SB	V	SRLI	V	OR	V
BLT	V	SH	V	SRAI	V	AND	V
BGE	V	SW	V	ADD	V	FENCE	V
BLTU	V	ADDI	E	SUB	V	ECALL	V
BGEU	V	SLTI	V	SLL	V	EBREAK	V

Tabla 7.2: Resultados de la validación para cada instrucción del conjunto de instrucción RV32I. El término V corresponde para una instrucción validada, mientras que E corresponde instrucciones que son validadas de forma extensiva.

7.3. Verificación del Despliegue en FPGA

El último proceso de validación considerado en el desarrollo del Core101 consiste en la verificación del funcionamiento correcto con el núcleo desplegado en una tarjeta de desarrollo FPGA. Este proceso de validación busca revisar si el núcleo mantiene el comportamiento una vez ha pasado por el proceso de despliegue

en la FPGA. Esta verificación es relevante ya que el despliegue en FPGA implica que se ha realizado una nueva representación del diseño generada a partir de la descripción en Verilog y los programas de la cadena de síntesis que se esté usando. El proceso de validación para la FPGA se presenta en la figura 7.3.

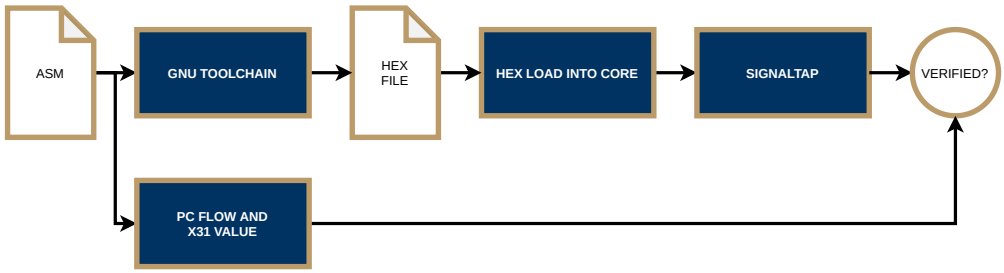

Figura 7.3: Flujo para la verificación del núcleo sobre el núcleo desplegado en la FPGA.

La validación del despliegue en FPGA parte de los programas de validación presentados previamente. Debido a la estructura de estos programas, es posible identificar el flujo de las instrucciones en términos del valor del contador de programa dentro del pipeline del procesador. Vale la pena notar que en este proceso de validación no se realiza una emulación de la ejecución sino que efectivamente se está ejecutando el código en el procesador. Adicionalmente, los programas están estructurados para que escriban valores específicos al registro $\mathbf{x31}$ dependiendo del resultado de la validación, por lo que el valor almacenado en este registro también debe ser tenido en cuenta para la validación. El flujo de los programas y el valor escrito en el registro $\mathbf{x31}$ puede ser adquirido de las señales físicas de la FPGA usando la herramienta SignalTap del software de desarrollo Quartus. En este sentido el despliegue será realizado sobre el Kit de Evaluación Cyclone 10LP debido a que Quartus es compatible únicamente con esta tarjeta de desarrollo. Una vez se ha revisado el flujo de ejecución para todos los programas y se ha identificado que son consistentes con el flujo esperado, se da por finalizada la validación.

Capítulo 8

Conclusiones y Trabajo Futuro

8.1. Conclusiones

A continuación se presentan las conclusiones relacionadas al desarrollo del núcleo Core101. Inicialmente se presentan conclusiones organizadas a partir de los diferentes enfoques de trabajo considerados para el desarrollo del núcleo. Cada conclusión presenta una sentencia, que presenta la idea central de la misma, y una discusión de esta donde se expande y se argumenta. Los enfoques de trabajo considerados para las conclusiones son: la metodología, el diseño del Core101, su implementación en lenguaje de alto nivel y lenguaje de descripción de hardware, su despliegue en tarjetas de desarrollo FPGA, la validación del funcionamiento del núcleo y el uso de herramientas de software para el proceso de desarrollo.

8.1.1. Metodología

1. **Se especificó una metodología para el desarrollo de sistemas digitales con énfasis en la validación:** en proyectos anteriores la metodología seguida para el desarrollo de hardware concebía un diseño, una única implementación en un lenguaje de descripción de hardware y la validación de esta. La metodología definida para este proyecto incluyó una implementación del diseño en un lenguaje de alto nivel con el fin de tener una referencia para la validación, en ese sentido, el proceso de validación se hizo más riguroso. No obstante, la validación aún está sujeta a la restricción de que una misma persona realice la implementación en el lenguaje de descripción de hardware y en lenguaje de alto nivel. Esta situación conlleva a que puede darse la situación en la que se comete el mismo error en las dos implementaciones, por lo que este no puede ser identificado de forma inmediata.
2. **Se definió una implementación en un lenguaje de alto nivel para ser usado como referencia del diseño:** bajo la premisa de que una implementación en alto nivel favorece la calidad de la misma, previniendo la existencia de errores, se definió una implementación del diseño del Core101 en el lenguaje de alto nivel Python. La implementación en Python busca ser una referencia que permita evaluar los resultados de la implementación en el lenguaje de descripción de hardware Verilog. Teniendo en cuenta lo anterior, fue posible ejecutar un mismo conjunto de pruebas con las dos implementaciones y comparar los resultados obtenidos. Este esquema de validación permitió no solo validar la microarquitectura del núcleo sino los módulos que la componen, además de permitir una automatización del proceso.

3. **Se establecieron dos tipos de simulaciones para la validación del Core101:** Teniendo en cuenta las limitaciones de las simulaciones de eventos y de ciclos, se especificaron las situaciones en las cuales se debían usar. Dentro del desarrollo del Core101 se determinó que la validación de los módulos y de la microarquitectura se haría mediante simulaciones. En este sentido, la validación de los módulos y de la microarquitectura tendría un uso de un simulador basado en eventos, esto con el fin de detectar errores que no se pueden identificar en un simulador de ciclos. Una vez se validaron los módulos y la microarquitectura se procedió con el uso de un simulador basado en ciclos, lo cual permite simular el funcionamiento del Core101 por un mayor número de ciclos de reloj, permitiendo simular la ejecución de programas más extensos y complejos.

8.1.2. Diseño

1. **Fue posible implementar una variación del pipeline RISC clásico para la especificación entera de RISC-V:** el pipeline clásico de RISC cuenta con cinco etapas correspondientes a la búsqueda y decodificación de instrucción, ejecución, acceso a memoria y escritura en registros. Por su parte, el pipeline del Core101 tiene seis etapas correspondientes a generación del valor del contador de programa, búsqueda y decodificación de instrucción, lectura de registros, ejecución y escritura en registros. Esta organización del pipeline del Core101 obedece a distintas razones: en primer lugar, el cálculo del valor del contador de programa es más complejo que en el pipeline RISC clásico debido a la predicción de saltos, esto hace que se deba dedicar una etapa a este cálculo. En segundo lugar, con el fin de limitar la lógica de la etapa de decodificación se desplazó la lectura de los registros del procesador a una etapa junto con la emisión de las microinstrucciones. En tercer lugar, se dedicó una etapa a la ejecución la cual incluyó los accesos a memoria, eliminando así esta etapa del pipeline clásico.

2. **Se diseñaron dos optimizaciones para el pipeline del procesador para aumentar su desempeño:** uno de los aspectos del diseño del Core101 fue la inclusión de dos optimizaciones para el pipeline: el adelantamiento de datos y la predicción de saltos. El adelantamiento de datos tiene como finalidad prevenir los peligros de datos en el Core101 y eliminar la latencia que podría existir entre la escritura de un registro y su lectura. De no haber incluido el adelantamiento de datos, se hubiese tenido que diseñar e implementar la lógica para incluir burbujas en el pipeline ante situaciones de lectura después de escritura (RAW, *Read After Write*), disminuyendo el desempeño del núcleo. Por otra parte, la predicción de saltos implicó la elaboración de una lógica que potencialmente elimina una latencia de cuatro ciclos de reloj por salto. Sin embargo, vale la pena mencionar que una latencia de esta magnitud aún es posible en situaciones donde la predicción del salto no sea igual a su resolución, o cuando la dirección de destino no es igual a la dirección predicha. En todo caso, la inclusión de estas dos optimizaciones permitió un aumento en el desempeño del núcleo.

3. **Se especificaron los protocolos para el acceso a la memoria de instrucción y de datos para una gestión apropiada de estos recursos:** Si bien el diseño del Core101 planteó un acceso a la memoria directo, es decir, sin hacer uso de un bus de interconexión, se definió un protocolo para el acceso de estos recursos. El objetivo de la inclusión de este protocolo fue la inclusión de la lógica necesaria para que el núcleo fuera capaz de soportar accesos a memoria con retardos, de forma similar a como lo haría un procesador con memorias cache. En este sentido, el protocolo le permite determinar al núcleo cuando debe parar la ejecución mientras se espera una respuesta de la memoria. Debido a que la arquitectura del núcleo es de tipo Harvard modificada, se establecieron los protocolos para la

memoria de instrucción y de datos. Bajo esta perspectiva, el Core101 es capaz de lidiar con retrasos en los accesos a las dos memorias.

8.1.3. Implementación

1. **Se implementó el diseño del Core101 en el lenguaje de alto nivel Python:** uno de los aspectos que se incluyó en la metodología fue la implementación del diseño en un lenguaje de alto nivel como Python. Si bien el lenguaje de alto nivel favorece la implementación, se deben tener en cuenta algunos aspectos para asemejar una implementación de hardware. En primer lugar, se definió un marco de trabajo para la implementación de módulos en Python. En este sentido, se desarrolló una clase que sirviera de base abstracta para la implementación de distintos módulos con sus señales de entrada, salida y archivos de memoria. En segundo lugar, la implementación de cada módulo tuvo en cuenta la metodología de sincronización del reloj con el fin de reflejar el comportamiento adecuado sin necesidad de generar una señal de reloj. En tercer lugar, se incluyó la capacidad de generar entradas automáticas para las señales con el fin de realizar la validación de cada módulo de forma eficiente. En lo referente a la implementación de la microarquitectura, gracias al uso de diccionarios fue posible realizar la conexión entre distintos módulos. En general, fue posible superar las limitaciones del software para asemejar una implementación de hardware a la vez que se explotaron las capacidades de la programación orientada a objetos.
2. **Se realizó una implementación en Verilog sintetizable del diseño del Core101 que permitió tanto su simulación como su despliegue:** la implementación en el lenguaje de descripción de hardware Verilog tuvo en cuenta dos características principales: el uso de una implementación modular, es decir, constituir el diseño como una interconexión de distintos módulos descritos en Verilog, y realizar todas las implementaciones con Verilog sintetizable. Estas dos características tenían como objetivo que fuera posible simular la implementación a partir del código en Verilog y realizar el despliegue del mismo en una tarjeta de desarrollo FPGA. Adicionalmente, el uso de la implementación modular buscaba favorecer el uso del Core101 en espacios académicos, de modo tal que un estudiante pudiese relacionar los módulos de la microarquitectura con la implementación en Verilog. De igual modo, la implementación modular favorecería el trabajo en caso tal que se requiera modificar el núcleo. En todo caso, la implementación en Verilog de forma modular y con una descripción sintetizable permitió que se realizaran simulaciones a partir de esta y que se lograra desplegar el diseño en una FPGA.

8.1.4. Despliegue

1. **Se evaluaron dos cadenas de herramientas para el despliegue del Core101 en dos FPGAs diferentes:** durante el desarrollo del proyecto se tuvieron a disposición dos tarjetas de desarrollo FPGA: una TinyFPGA BX y un kit de evaluación para una FPGA Cyclone 10. La tarjeta TinyFPGA BX utiliza la cadena de herramientas del proyecto IceStorm y Yosys para la síntesis, evaluación y despliegue de un diseño en esta. Las herramientas para esta FPGA cuentan con una buena documentación por lo que fue posible integrarlas como parte del proyecto. Por otra parte, el kit de evaluación de la FPGA Cyclone 10 utiliza Quartus como plataforma de desarrollo. Una característica utilizada de Quartus, no disponible en el proyecto IceStorm, fue el analizador SignalTap. El analizador SignalTap permite adquirir datos del diseño operando sobre la FPGA por lo que se puede depurar el mismo a partir de estos datos. Si bien las herramientas asociadas a cada tarjeta de desarrollo permitieron el despliegue

del Core101 en estas, el desarrollo se vio favorecido por la disponibilidad de herramientas como el SignalTap en el caso del kit de evaluación. De igual manera, las dos cadenas de herramientas quedaron configuradas para favorecer la adopción del diseño por parte de terceros.

8.1.5. Validación

1. **Fue posible validar el núcleo mediante una combinación de simulaciones aleatorias y software compilado para RISC-V:** la metodología de validación permitió que el diseño del núcleo fuera validado de forma exhaustiva en distintos niveles de abstracción y con diferentes clases de estímulos. Por una parte, los módulos de la microarquitectura fueron validados mediante el uso de señales aleatorias que eran generados desde la implementación en el lenguaje de alto nivel Python. A partir de la implementación en Python se generaban los resultados de referencia para validar la implementación en el lenguaje de descripción de hardware Verilog. Mediante el uso de un banco de prueba en Verilog y el simulador de eventos ModelSim se leían las entradas aleatorias y se evaluaba la implementación en Verilog. Los resultados obtenidos de esta simulación eran comparados con los de referencia para determinar si el módulo operaba correctamente. Para el caso de la validación de la microarquitectura se utilizó la cadena de compilación de RISC-V para generar archivos hexadecimales que podían ser cargados en las dos implementaciones del Core101. Esto permitió obtener una referencia de la ejecución de un programa mediante la implementación de Python, que podía ser usado para validar la implementación en Verilog.

8.1.6. Software

1. **Fue posible utilizar la cadena de herramientas de software de RISC-V para generar programas y archivos hexadecimales usados en el Core101:** aprovechando la existencia de herramientas de compilación de software para la arquitectura RISC-V, se generaron los archivos que eran cargados al Core101 para la ejecución de programas. Estos archivos consistían en una serie de valores hexadecimales que representaban las instrucciones compiladas del programa. Los programas, por su parte, eran escritos en ensamblador de RISC-V. Adicionalmente, fue posible desarrollar un *linker* que permitiera gestionar las memorias disponibles en las tarjetas de desarrollo. Mediante el uso del linker fue posible utilizar las memorias disponibles en las tarjetas de desarrollo para almacenar y ejecutar programas.

8.2. Trabajo Futuro

Teniendo en cuenta el alcance logrado con el desarrollo del Core101 se presentan algunas ideas para que sean desarrolladas en el futuro en torno al presente proyecto. Las ideas presentadas en esta sección se dividen en dos: *inside core* (dentro del núcleo) y *extra core* (fuera del núcleo). Inside core hace referencia a aquellas ideas que podrían modificar algún módulo o módulos de la microarquitectura del Core101 para mejorar su rendimiento, consumo energético o soporte de instrucciones. Por otra parte, extra core hace referencia a las ideas que podrían usar el Core101, como un todo, para llevar a cabo otra clase de proyectos.

8.2.1. Inside Core

Las modificaciones que pueden tener lugar dentro del Core101 a futuro pueden ser divididas en base a mejoras de rendimiento y aumento en el soporte de instrucciones. El primer conjunto de mejoras no buscaría extender el soporte actual sino mejorar la microarquitectura. En otras palabras, buscarían mejorar el desempeño del núcleo sin soportar nuevas instrucciones. Por el contrario, el segundo conjunto de mejoras buscaría extender el número de instrucciones soportadas por el Core101.

Mejoras de rendimiento

1. **Cambio del esquema de predicción de saltos condicionales:** consiste en modificar el predictor de saltos condicionales para utilizar otro esquema que mejore la tasa de predicción. Algunos esquemas que pueden ser utilizados son predictores de correlación (*correlating predictors*) [9, p. 182] o predictores de torneo (*tournament predictors*) [9, p. 184].
2. **Cálculo de valores comunes:** consiste en calcular valores comunes durante la etapa de lectura de registros o de ejecución. Algunos de estos valores comunes son la suma del contador de programa y cuatro unidades, la suma del contador de programa y el valor inmediato, y la suma del primer registro fuente y el valor inmediato. Estas sumas son utilizadas ampliamente a lo largo del núcleo y pueden ser calculadas en una unidad para evitar hardware con funcionalidad duplicada.

Soporte de nuevas instrucciones

1. **Registros de control y estado:** consiste en agregar un módulo correspondiente a los registros de control y estado (CSR, *Control Status Registers*) además de la lógica para decodificar las instrucciones relacionadas a la lectura y escritura de estos registros. Adicionalmente, se deben agregar los registros de contadores de instrucciones y del reloj para soportar completamente estas características. El soporte de estos registros y sus instrucciones se traduce en el soporte para la especificación Zicsr [11, p. 55].
2. **Multiplicación y división:** consiste en dotar al Core101 de la capacidad para realizar operaciones de multiplicación y división en hardware, además de agregar la lógica para decodificar las instrucciones asociadas a estas operaciones. El soporte de estas instrucciones se traduce en el soporte para la especificación RV32M [11, p. 43].
3. **Instrucciones comprimidas:** consiste en agregar la lógica de la decodificación de instrucciones comprimidas. Debido a que la especificación de instrucciones comprimidas (RVC) no agrega nuevas instrucciones, sino que mapea instrucciones de la base entera (RV32I) a sus variantes de 16 bitss, agregar el soporte para estas instrucciones consiste en modificar la lógica de decodificación, ya que el resto del núcleo puede soportar el resto de la ejecución. Soportar las variantes comprimidas de RISC-V dotaría al núcleo del soporte para la especificación RV32C [11, p. 97].
4. **Unidad de punto flotante:** consiste en agregar una unidad de punto flotante a las unidades de ejecución del núcleo para permitir operaciones con esta clase de datos. Agregar esta unidad es una de las modificaciones más grandes que se le pueden hacer al núcleo, ya que se debe agregar lógica para la decodificación de nuevas instrucciones, un conjunto de registros para punto flotante y la unidad de punto flotante como tal. La posibilidad de operar valores con punto flotante dotarían al núcleo de soporte para las especificaciones RV32FD [11, p. 63].

8.2.2. Extra Core

Si bien no hay mucho que modificar fuera del Core101, sí existen algunas ideas para trabajos futuros que pueden ampliar la funcionalidad del núcleo agregando periféricos y buses de interconexión. Por otra parte, se puede realizar trabajo externo relacionado a validaciones adicionales del núcleo o a realizar una descripción en otro lenguaje de descripción de hardware o sistemas. Basado en los diferentes enfoques se propone una diferenciación similar a la hecha en el trabajo futuro inside core.

Periféricos

1. **Protocolos de comunicación serial:** son un conjunto de periféricos que pueden ser desarrollados que permiten la comunicación con otros sistemas, mediante protocolos de comunicación serial como I^2C, UART, SPI, entre otros. Los módulos destinados a esta clase de protocolos de comunicación pueden ser combinados con algún bus de interconexión estándar para ser usado en conjunto con el Core101.
2. **Ethernet:** consistiría en un módulo que permita la conexión mediante Ethernet a otros dispositivos y sistemas. Esta clase de conexiones permitiría que el sistema pueda ser usado a distancia, o que permita el uso de varios sistemas en conjunto. Al igual que los módulos de comunicación serial, este módulo puede ser combinado con un bus de interconexión para ser usado con el Core101.

Buses de Interconexión

1. **Tilelink:** Tilelink es un estándar de conexión para circuitos integrados diseñado para RISC-V por SiFive. Su aplicación con el Core101 consistiría en desarrollar los adaptadores para el núcleo y los periféricos con el fin de usar este estándar como medio de interconexión para los mismos [10].
2. **AMBA:** Es un especificación abierta desarrollada por ARM para conectar y gestionar diferentes bloques funcionales en un sistema en un chip (SoC, *System-on-a-Chip*). A pesar que es propiedad de ARM, su uso no implica el pago de regalías y puede ser usado por cualquier persona interesada. En la actualidad es uno de los protocolos de facto usados en el diseño de SoCs para sistemas embebidos, por lo que existe una documentación completa del protocolo además de módulos de propiedad intelectual para su uso [2].

Validaciones Adicionales

1. **RISC-V Tests:** consiste en compilar y ejecutar las pruebas denominadas RISC-V Tests en el Core101 para llevar a cabo este nivel de validación. La compilación y ejecución de estas pruebas requiere modificar algunas macros para poder interpretar los resultados apropiadamente [4].
2. **RISC-V Benchmark:** consiste en compilar y ejecutar las pruebas de desempeño contenidas dentro de las pruebas de RISC-V para obtener métricas del Core101. Adicionalmente, se pueden ejecutar estas pruebas con otras implementaciones de RISC-V para comparar los resultados con los del Core101 [4].

Lenguajes de Descripción de Hardware

1. **Chisel:** es un lenguaje de implementación de hardware de alto nivel, embebido dentro del lengauje Scala, que permite generar descripciones del sistema en diferentes niveles de abstracción. Por ejemplo, a partir de la descripción de un módulo en Chisel, es posible generar la descripción en Verilog del

mismo para su simulación o despliegue. Adicionalmente, tiene la ventaja que al ser un lenguaje de alto nivel, permite que los módulos y sistemas sean validados utilizando la misma descripción. El Core101 podría ser implementado en este lenguaje con el fin de facilitar su desarrollo futuro [1].

Bibliografía

[1] CHIPS Alliance. Chisel 3 HDL. https://github.com/chipsalliance/chisel3.

[2] Arm Limited. *AMBA AXI and ACE Protocol Specification*, 2020.

[3] Universidad Industrial de Santander (OnChip UIS). mriscvcore. https://github.com/onchipuis/mriscvcore.

[4] RISC-V Foundation. RISC-V Tests. https://github.com/riscv/riscv-tests.

[5] GNU. GNU Compiler Collection. http://gcc.gnu.org/.

[6] lowRISC. Ibex RISC-V core. https://github.com/lowRISC/ibex.

[7] OpenHW Group. CV32E40P RISC-V core. https://github.com/openhwgroup/cv32e40p.

[8] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Elsevier Science, 2017.

[9] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Elsevier Science, 2019.

[10] SiFive Inc. *SiFive TileLink Specification*, 2019.

[11] Andrew Waterman and Krste Asanovic, editors. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 2019121*. RISC-V Foundation, December 2019.