Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería Mecánica Eléctrica

AUMENTO DEL RENDIMIENTO DE UN PROCESADOR *RISC* DE 32 *BITS*, UTILIZANDO *PIPELINE* DE CINCO ETAPAS Y MEMORIA CACHÉ

Ottoniel Abisaí Sierra Madrid
Asesorado por el MSc. Ing. Iván René Morales Argueta

Guatemala, noviembre de 2021

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

AUMENTO DEL RENDIMIENTO DE UN PROCESADOR *RISC* DE 32 *BITS*, UTILIZANDO *PIPELINE* DE CINCO ETAPAS Y MEMORIA CACHÉ

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

OTTONIEL ABISAÍ SIERRA MADRID
ASESORADO POR EL MSC. ING. IVÁN RENÉ MORALES ARGUETA

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN ELECTRÓNICA

GUATEMALA, NOVIEMBRE DE 2021

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANA	Inga. Aurelia Anabela Cordova Estrada
VOCAL I	Ing. José Francisco Gómez Rivera
VOCAL II	Ing. Mario Renato Escobedo Martínez
VOCAL III	Ing. José Milton de León Bran
VOCAL IV	Br. Kevin Vladimir Armando Cruz Lorente
VOCAL V	Br. Fernando José Paz González
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANA	Inga. Aurelia Anabela Cordova Estrada
EXAMINADOR	Ing. José Aníbal Silva de los Angeles
EXAMINADOR	Ing. Hugo Leonel Tiul Valenzuela
EXAMINADOR	Ing. Miguel Ventura Pérez
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

AUMENTO DEL RENDIMIENTO DE UN PROCESADOR *RISC* DE 32 *BITS*, UTILIZANDO *PIPELINE* DE CINCO ETAPAS Y MEMORIA CACHÉ

Tema que me fuera asignado por la Dirección de la Escuela de Ingeniería Mecánica Eléctrica, con fecha 02 de septiembre del 2020.

Ottoniel Abisaí Sierra Madrid

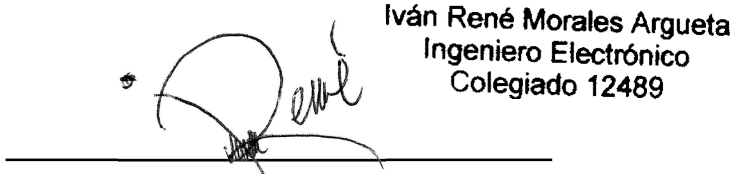
Guatemala, 29 de mayo de 2021

Ingeniero
Julio Solares Peñate
Coordinador de Área de Electrónica
Facultad de Ingeniería
Universidad de San Carlos de Guatemala

Señor coordinador:

Por este medio tengo el gusto de informarle que he concluido con el asesoramiento y revisión del trabajo de graduación con titulo: **Aumento del rendimiento de un procesador RISC de 32 bits, utilizando pipeline de cinco etapas y memoria cache,** desarrollado por el estudiante Ottoniel Abisaí Sierra Madrid con carne 201403847. Después de revisar su contenido final doy mi entera aprobación al mismo.

Atentamente,



MSc. Ing. Iván René Morales Argueta
Ingeniero Electrónico
Colegiado Activo No. 12489

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERIA

Guatemala, 7 de junio de 2021

Señor director
Armando Alonso Rivera Carrillo
Escuela de Ingeniería Mecánica Eléctrica
Facultad de Ingeniería, USAC

Estimado Señor director:

Por este medio me permito dar aprobación al Trabajo de Graduación titulado: **AUMENTO DEL RENDIMIENTO DE UN PROCESADOR RISC DE 32 BITS, UTILIZANDO PIPELINE DE CINCO ETAPAS Y MEMORIA CACHE,** desarrollado por el estudiante **Ottoniel Abisaí Sierra Madrid**, ya que considero que cumple con los requisitos establecidos.

Sin otro particular, aprovecho la oportunidad para saludarlo.

Atentamente,

ID Y ENSEÑAD A TODOS

Ing. Julio César Solares Peñate
Coordinador de Electrónica

UNIVERSIDAD DE SAN CARLOS
DE GUATEMALA

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA MECÁNICA ELÉCTRICA
FACULTAD DE INGENIERÍA
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

REF. EIME 114. 2021.

El Director de la Escuela de Ingeniería Mecánica Eléctrica, después de conocer el dictamen del Asesor, con el Visto Bueno del Coordinador de Área, al trabajo de Graduación del estudiante; OTTONIEL ABISAÍ SIERRA MADRID titulado: AUMENTO DEL RENDIMIENTO DE UN PROCESADOR RISC DE 32 BITS, UTILIZANDO PIPELINE DE CINCO ETAPAS Y MEMORIA CACHE, procede a la autorización del mismo.

Ing. Armando Alonso Rivera Carrillo

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DIRECCION ESCUELA DE INGENIERIA MECANICA ELECTRICA
FACULTAD DE INGENIERIA

GUATEMALA, 10 DE AGOSTO 2,021.

USAC
TRICENTENARIA
Universidad de San Carlos de Guatemala

Decanato
Facultad de Ingeniería
24189101 – 24189102
secretariadecanato@ingenieria.usac.edu.gt

DTG. 590-2021

La Decana de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Ingeniería Mecánica Eléctrica, al Trabajo de Graduación titulado: **AUMENTO DEL RENDIMIENTO DE UN PROCESADOR RISC DE 32 BITS, UTILIZANDO PIPELINE DE CINCO ETAPAS Y MEMORIA CACHÉ**, presentado por el estudiante universitario: **Ottoniel Abisaí Sierra Madrid,** y después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, autoriza la impresión del mismo.

IMPRÍMASE:

Inga. Anabela Cordova Estrada
Decana

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANO
FACULTAD DE INGENIERIA

Guatemala, noviembre de 2021

AACE/cc

ACTO QUE DEDICO A:

Dios	Por ser una importante influencia en mi carrera, y en mi vida.
Mis padres	Otoniel Sierra, que en paz descanse y Telma Madrid de Sierra. Por su amor incondicional y apoyo constante.
Mi tía	Justina Madrid Herrera. Por su amor y apoyo incondicional.
Mi hermano	Abner Azael Sierra Madrid, con amor fraternal.

AGRADECIMIENTOS A:

Universidad de San Carlos de Guatemala	Por permitirme recibir el conocimiento profesional en sus aulas.
Facultad de Ingeniería	Por ser la facultad que desarrolló mis habilidades profesionales.
MSc. Ing. Iván Morales	Por su invaluable colaboración y asesoría para la realización de este trabajo de tesis.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

LISTA DE SÍMBOLOS

Símbolo	Significado
·	Función lógica *AND*
Hz	Hertz
=	Igualdad
log_2	Logaritmo base dos
>	Mayor que
MHz	Megahertz
<	Menor que
ms	Milisegundos
*****	Multiplicación
ns	Nanosegundos
0b_	Número binario
0x_	Número hexadecimal
%	Porcentaje
-	Resta
s	Segundos
+	Suma
∑	Sumatoria

GLOSARIO

Aleatorio	Que depende del azar.
Bit	Unidad de medida de cantidad de información, equivalente a la elección entre dos posibilidades igualmente probables.
Bus	Conjunto de conductores común a varios dispositivos que permite distribuir corrientes de alimentación.
Byte	Unidad de información compuesta generalmente de ocho *bits*.
C	Lenguaje de programación de computadora imperativo, de procedimiento y de propósito general desarrollado en 1972 por Dennis M. Ritchie en Bell.
CMOS	Semiconductor complementario de óxido metálico es una de las familias lógicas empleadas en la fabricación de circuitos integrados.
CPI	Ciclos Por Instrucción, mide la cantidad de ciclos necesarios para ejecutar una instrucción.
Concatenar	Unir o enlazar.

DDR SDRAM	Clase de memoria de acceso aleatorio dinámica síncrona, SDRAM, de doble velocidad de datos, DDR, de circuitos integrados de memoria utilizados en computadoras.
Digital	Dicho de un dispositivo o sistema: Que crea, presenta, transporta o almacena información mediante la combinación de *bits*.
DL	Abreviación de Decodificación y Lectura.
EJ	Abreviación de Ejecución de Instrucciones.
ER	Abreviación de Escritura de Registros.
FIFO	Primero en Entrar Primero en Salir.
FPGA	Circuito integrado con la capacidad de ser configurado posterior a su manufactura.
Hardware	Equipo físico asociado directamente en el desempeño de la función de procesamiento de datos o comunicaciones.
ISA	Modelo abstracto de una computadora.
LRU	Menos Recientemente Utilizado.
MEM	Abreviación de Acceso a Memoria.

MIPS	Millones de Instrucciones Por Segundo.
Micro-arquitectura	Forma en que se implementa una *ISA* dada en un procesador en particular.
OB	Abreviación de Obtención de Instrucciones.
Pila	Lista ordenada o estructura de datos que permite almacenar y recuperar datos.
Procesador	Unidad funcional de una computadora u otro dispositivo electrónico, que se encarga de la búsqueda, interpretación y ejecución de instrucciones.
Python	Lenguaje de programación interpretado de alto nivel de propósito general.
RAM	Memoria de acceso aleatorio
RISC-V	*ISA* de libre uso.
ROM	Memoria de solo lectura.
Retroalimentación	Retorno de parte de la energía o de la información de salida de un circuito o un sistema a su entrada.
Software	Conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora.

VHDL	Lenguaje de descripción de hardware con la capacidad de modelar el comportamiento y estructura de sistemas digitales.

RESUMEN

El trabajo consiste en aumentar el rendimiento de un procesador *RISC*, aumentando su frecuencia de reloj y reduciendo su tiempo de acceso promedio. Para ello, en el primer capítulo se implementa un procesador basado en la *ISA* RV32I de la familia RISC-V el cual es utilizado como control. Además, se presenta el entorno utilizado para medir su rendimiento y comprobar su correcto funcionamiento.

El segundo capítulo se enfoca en el aumento de la velocidad de reloj. En consecuencia, se presenta la idea de *pipeline*; cómo es aplicado en un procesador, las consecuencias negativas que implica su implementación y se presentan soluciones para mitigar sus efectos negativos. Como resultado se logra obtener un aumento de hasta 2.03 veces el rendimiento respecto al procesador de control.

En el tercer capítulo se introduce a la brecha que existe entre la velocidad de operación de procesadores y memorias *RAM*, y cómo esto se convierte en un impacto negativo en el rendimiento de un procesador. Como solución se presenta a las memorias caché y sus parámetros de diseño. Se realiza el diseño de cuatro tipos de memoria caché, y se calculan las cinco configuraciones de parámetros que presentan el menor tiempo de acceso promedio para una memoria de instrucciones y para una memoria de datos. Como resultado se logra eliminar el impacto negativo causado por la diferencia entre la velocidad del procesador y memorias *RAM*.

OBJETIVOS

General

Aumentar el rendimiento de un procesador *RISC* de 32 *bits*, reduciendo el periodo de reloj utilizando *pipeline* de cinco etapas y disminuir el tiempo promedio de acceso a memoria utilizando memorias caché.

Específicos

1. Identificar y mitigar los diferentes riesgos funcionales en un procesador al implementar pipeline.

2. Implementar en VHDL un procesador *RISC* de 32 *bits* con *pipeline* de cinco etapas.

3. Verificar el funcionamiento correcto del procesador *RISC* de 32 *bits* con *pipeline* de cinco etapas utilizando herramientas de simulación.

4. Obtener la configuración óptima de parámetros en un caché de datos y un caché de instrucciones.

INTRODUCCIÓN

Actualmente vivimos en una cultura de alta demanda de cómputo, en nuestros celulares lo utilizamos para navegar en la web, encriptar, desencriptar comunicaciones, al ver un video se utiliza cómputo para decodificar la información. Los servidores a los cuales nuestros celulares se conectan necesitan cómputo para procesar la información de cientos, miles o millones de usuarios que requieren de sus servicios. En los vehículos que cuentan con sistemas de control digitales, entre otros.

Todo este poder de cómputo es entregado por lo general en la unidad central de procesamiento o *CPU*, por ello este trabajo pretende presentar dos métodos, utilizados en la arquitectura de sistemas de cómputo, para mejorar el desempeño de un *CPU*, *pipeline* y memorias caché como lo demuestra SRINIVASAN, Viji. *Optimizing pipelines for power and performance* y CARVALHO, Carlos. *The gap between processor and memory speeds*.

Se guiará al lector en el proceso de diseño, implementación y simulación en el lenguaje de descripción de hardware VHDL, con el objetivo final de obtener la descripción hardware de un procesador *RISC* de 32 *bit* con *pipeline* de cinco etapas y memoria caché.

Se asume que el lector posee conocimientos de electrónica digital y diseño de circuitos digitales, de no ser el caso se recomienda la lectura de los capítulos uno al diez de MORRIS, Mano, *Diseño digital*.

1. PROCESADOR *RISC* DE 32 *BITS*

Un procesador es un circuito digital diseñado para ejecutar instrucciones que componen un programa de computadora. Estas instrucciones indican al procesador realizar operaciones aritméticas, lógicas, de control, de acceso de memoria o de entrada y salida, I/O.

En simples términos un procesador es el encargado de leer y ejecutar instrucciones, pero se necesita algo que defina de forma clara y precisa, dichas instrucciones, por esta razón un procesador es construido basado en un *ISA*.

1.1. *ISA*

Instruction Set-Architecture, ISA, o arquitectura del conjunto de instrucciones en español, es una abstracción que representa la arquitectura de una computadora, además describe la funcionalidad de todas las instrucciones que debe ejecutar un procesador.

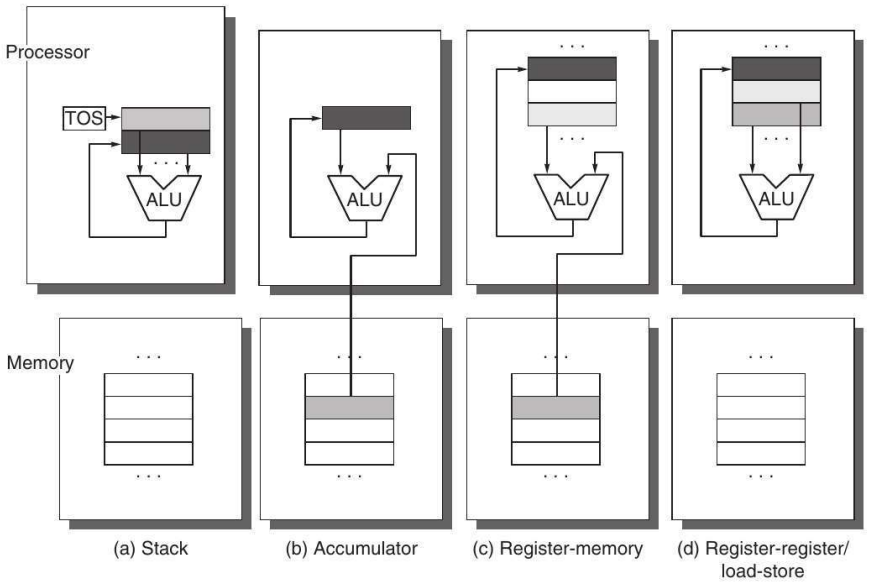
Adicionalmente de proveer a un arquitecto de computadores con la funcionalidad y diseño general que debe poseer el procesador, un *ISA* es utilizado por programadores ya que indica cuales son las instrucciones que el procesador es capaz de ejecutar. De esta manera un *ISA* cubre los aspectos físicos y abstractos, hardware y software, de un sistema computo.

Al implementar la funcionalidad de un *ISA* se presenta una amplia variedad de alternativas de diseño, las cuales se pueden utilizar para clasificarlo.

1.1.1. Clasificación de *ISA*

Tipo de almacenamiento interno y argumentos.

Figura 1. **Clasificación de *ISA* por tipo de almacenamiento interno y argumentos**



Fuente: HENESSY, Jhon, PATTERSON, David. *Computer Architecture A Quantitative Aproach*. p. A-4.

Stack, a, se caracteriza por utilizar una pila o *stack* como almacenamiento interno, sus argumentos son implícitos, el primer operando es el valor al tope de la pila, indicado por *Top Of Stack, TOS*, el cual es combinado con el argumento debajo de él, al finalizar una operación el primer argumento es removido de la pila, el resultado es escrito en la posición del segundo operando, el tope de la pila, *TOS*, es actualizado para apuntar al resultado, necesita instrucciones especiales para realizar transferencias entre memoria y la pila.

Acumulador, b, el único almacenamiento interno es el acumulador, sus argumentos son el acumulador y un valor localizado en memoria, el resultado de toda operación es escrito en el acumulador.

Registro-memoria, c, utiliza registros como almacenamiento interno, sus argumentos pueden ser registro-registro, registro-memoria o memoria-memoria, el resultado de las operaciones es escrito en un registro o en memoria.

Registro-registro/cargar-guardar, d, utiliza registros como almacenamiento interno, sus argumentos únicamente son registros, el resultado de una operación solo puede ser escrito en un registro, para realizar transferencias entre registros y memoria se utilizan instrucciones cargar-guardar.

1.1.2. Codificación de instrucciones

La codificación de instrucciones puede ser constante o variable. Una codificación constante implica que todas las instrucciones poseen el mismo tamaño, esto genera la posibilidad que la información necesaria para su decodificación se encuentre ubicada en la misma posición en todas las instrucciones, simplificando la lógica digital necesaria para la decodificación.

Una codificación variable implica un tamaño variable de instrucciones, diferentes instrucciones pueden poseer diferentes tamaños, su desventaja es el aumento de complejidad en la lógica digital necesaria para la decodificación de instrucciones, su ventaja es que al poseer un tamaño variado de instrucciones el tamaño de los programas en general es menor comparado con una codificación constante.

Independiente si una codificación es constante o variable, cada *ISA* define su propio formato de instrucciones y su interpretación al momento de ser decodificados.

1.1.3. Interpretación de memoria

Una *ISA* puede definir la estructura de la memoria, por ejemplo, puede definir si la memoria utilizará *little endian* o *big endian*, el tamaño en *bytes* de un acceso a memoria. Adicionalmente define como una instrucción que realiza un acceso a memoria calcula la dirección objetivo.

1.1.4. *RISC* y *CISC*

Una forma común de clasificar un procesador es refiriéndose a un procesador *Reduced Instruction Set Computer, RISC*, computadora de instrucciones reducidas, o un procesador *Complex Instruction Set Computer, CISC*, computadora de instrucciones complejas.

En realidad, la distinción entre ambos puede darse a la diferencia en su *ISA*, por ejemplo, un procesador *RISC* se caracteriza por ser registro-registro/cargar-guardar, poseer codificación de instrucciones constante, en su interpretación de memoria se realizan cálculos simples para obtener la dirección objetivo.

Un procesador *CISC* se caracteriza por ser registro-memoria, poseer codificación de instrucciones variable, realizar cálculos complejos para obtener la dirección objetivo al ejecutar instrucciones de acceso a memoria. Pero la distinción más grande entre *RISC* y *CISC* es la complejidad de las instrucciones que el procesador ejecuta, un procesador RISC ejecuta instrucciones simples y más fáciles de ejecutar comparado a un procesador *CISC*.

“Debido a la complejidad de sus instrucciones, *CISC* posee una eficiencia menor a *RISC*, por lo tanto, a partir del procesador Intel P6 los procesadores de la familia x86 al momento de leer instrucciones, *CISC*, son convertidas a micro-operaciones, *µOP*, que son instrucciones más fáciles de ejecutar con un estilo similar a *RISC*”[1].

Por esta razón se considera que los métodos y técnicas de diseño de procesadores *RISC* poseen mayor relevancia, ya que al diseñar un procesador *CISC* sus instrucciones serán convertidas en micro-operaciones.

1.1.5. *ISA* y micro-arquitectura

Se debe tomar en consideración que un *ISA* no restringe como debe ser implementada su funcionalidad a nivel de circuitos, por ejemplo si se compara el procesador Intel Core i3-8100 y su equivalente en AMD Ryzen 3 1300x, se obtienen diferencias en su desempeño[2], a pesar que ambos procesadores utilizan el mismo *ISA*, x86-64, su diferencia se encuentra en la forma que fue implementado la funcionalidad, el diseño de los circuitos, esto es llamado micro-arquitectura, la micro-arquitectura es lo que diferencia un procesador Intel Pentium 4, de un procesador Intel Core i9, ya que ambos implementan el *ISA* x86-64.

[1] DE GELAS, Johan. *Decoding Instructions - Intel Core versus AMD’s K8 architecture.* https://www.anandtech.com/show/1998/3. *Consulta: octubre de 2020.*

[2] cpu.userbenchmark.com. *AMD Ryzen3 1300X vs Intel Core i3 8100*. https:// /Compare/AMD-Ryzen-3-1300X-vs-Intel-Core-i3-8100/3930vs3942. Consulta: octubre de 2020.

1.1.6. Uso de un *ISA* existente vs., diseño de un nuevo *ISA*

Desde el punto de vista de un arquitecto de computadoras el diseño de un nuevo *ISA* puede parecer atractivo ya que se posee el control absoluto de las instrucciones que ejecuta el procesador, conociendo la complejidad de la implementación en circuitos digitales puede decidir incluir, eliminar o modificar instrucciones del *ISA*. Adicionalmente diseñar un nuevo *ISA* elimina la necesidad de obtener licencias de uso, requerido por ciertas *ISA* comerciales.

Al tomar esta perspectiva se ignora completamente la parte de software, si se desea crear un *ISA* realmente útil y funcional, se debe considerar la totalidad de un sistema computacional, por ejemplo, considerar la utilidad y facilidad en la implementación de sistemas operativos, capacidad de virtualización, escalabilidad en diseño de procesadores multi-núcleo, entre otros. Adicionalmente se considera el uso objetivo, por ejemplo, si el *ISA* será utilizado en aplicaciones científicas, de propósito general como ordenadores personales, servidores, dispositivos móviles, o será utilizado en sistemas embebidos. De forma similar se debe tomar en cuenta la tecnología y técnicas en el diseño de circuitos. Encima de todo esto debe ofrecer beneficio o utilidad que otros *ISA* no posean, de lo contrario se diseña un *ISA* redundante.

Por estas razones en este trabajo el procesador implementado utiliza el *ISA* RISC-V.

1.2. RISC-V

RISC-V es un *ISA* diseñada originalmente para soportar la investigación en el área de arquitectura de computadoras y para educación, RISC-V *Foundation*[3]. Es completamente libre de uso en el área académico y comercial, uno de sus objetivos es ser un *ISA* apta para implementación nativa en hardware y no solamente para su uso en simulaciones o traducciones binarias.

RISC-V está constituido por un conjunto de pequeños *ISA* de enteros utilizados como base de la arquitectura, los cuales son capaces de realizar computación por sí mismos. Y extensiones opcionales utilizadas para ampliar la funcionalidad del procesador.

Los *ISA* base de enteros son RV32I, RV32E, RV64I, y RV128I. Los cuales definen instrucciones para arquitecturas de 32, 64 y 128 *bits*.

A continuación, se presenta la nomenclatura del *ISA* base y sus exenciones lo cuales conforman RISC-V:

[3] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft.* p. 1.

Tabla I. **Nomenclatura de *ISA* base y extensiones**

Subset	Name	Implies
Base ISA		
Integer	I	
Reduced Integer	E	
Standard Unprivileged Extensions		
Integer Multiplication and Division	M	
Atomics	A	
Single-Precision Floating-Point	F	Zicsr
Double-Precision Floating-Point	D	F
General	G	IMADZifencei
Quad-Precision Floating-Point	Q	D
Decimal Floating-Point	L	
16-bit Compressed Instructions	C	
Bit Manipulation	B	
Dynamic Languages	J	
Transactional Memory	T	
Packed-SIMD Extensions	P	
Vector Extensions	V	
User-Level Interrupts	N	
Control and Status Register Access	Zicsr	
Instruction-Fetch Fence	Zifencei	
Misaligned Atomics	Zam	A
Total Store Ordering	Ztso	
Standard Supervisor-Level Extensions		
Supervisor-level extension "def"	Sdef	
Standard Hypervisor-Level Extensions		
Hypervisor-level extension "ghi"	Hghi	
Standard Machine-Level Extensions		
Machine-level extension "jkl"	Zxmjkl	
Non-Standard Extensions		
Non-standard extension "mno"	Xmno	

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p 152.

Para fines de este trabajo se desarrolla el *ISA* base RV32I.

1.2.1. RV32I

A continuación, se presenta un resumen del *ISA* relevante para este trabajo:

Se define una arquitectura y un espacio de direcciones de 32 *bits*, en otras palabras, las diferentes instrucciones realizan operaciones a conjuntos de valores booleanos, enteros con signo o enteros sin signo de 32 *bits*.

El espacio de direcciones de memorias es circular, el *byte* correspondiente a la dirección 2^{32} - 1 es adyacente al *byte* de la dirección 0, en consecuencia, si una computación calcula una dirección de memoria mayor a 2^{32}, se aplica el módulo de 2^{32} a dicha dirección.

Posee 32 registros, *x0* a *x31*, donde *x0* es un registro dedicado a cero, es decir el único valor que puede tener *x0* es cero y el resto son registros de propósito general.

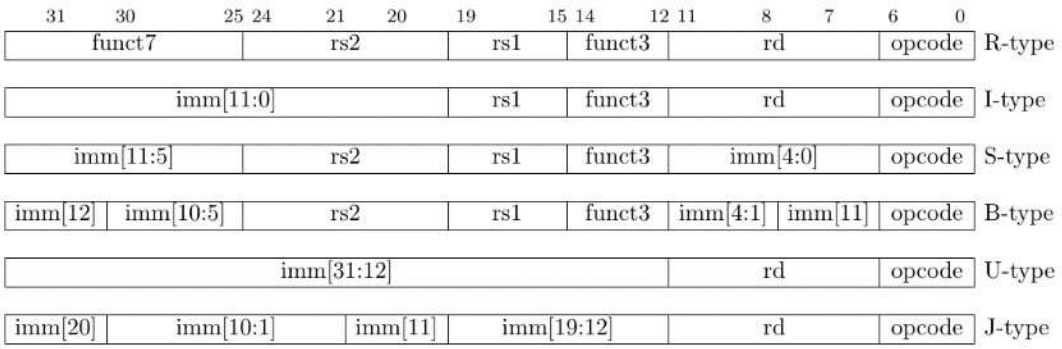
Es una arquitectura registro-registro/cargar-guardar.

Para representar números enteros con signo se utiliza complemento a dos.

1.2.1.1. Formato de instrucciones

RV32I posee cuatro formatos principales, R/I/S/U y dos formatos adicionales, B/J, que varían en el manejo de valores inmediatos, todos estos formatos poseen una codificación constante de 32 *bits*.

Figura 2. **Formato base de instrucciones RISC-V**



Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 16.

Donde:

Rd = registro destino.

rs1, rs2 = registros utilizados como argumentos u origen.

opcode = define el tipo de instrucción.

funct3, funct7 = define la acción a realizar dependiendo del tipo de instrucción.

imm = utilizado para decodificar valores inmediatos.

1.2.1.2. Decodificación de valores inmediatos

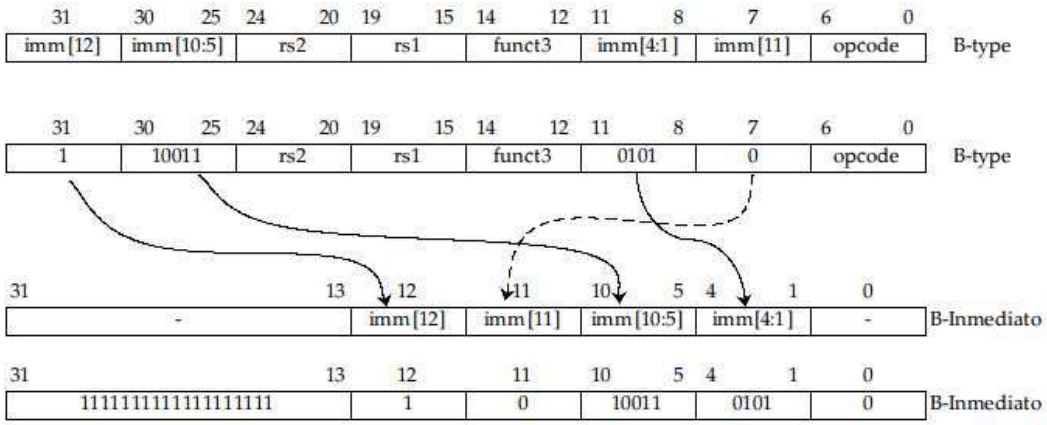
En algunos casos se desean realizar operaciones entre un registro y una constante, por ejemplo, se desea sumarle la constante 1 al registro 3, *x3*, o hacer una operación booleana *AND* entre un registro y el equivalente binario del número 255.

Estas constantes están codificadas dentro de las instrucciones en los *bits* correspondientes a *imm,* como se muestra en la figura 2. Debido a que se

implementa una arquitectura de 32 *bits*, al momento de realizar una operación dichas constantes deben de ser extendidas a su equivalente binario de 32 *bits* en representación complemento a dos.

Al decodificar un valor inmediato se utilizan los valores dentro de *imm*, ya sea un valor único *imm[n]* o un rango *imm[n:k]*, para indicar la posición que ocupan en la representación de 32 *bits* final de la constante. Si la representación de-codificada no abarca el *bit* 31, se utiliza el valor del *bit* más significativo y se extiende en todos los *bits* vacíos más significativos hasta completar 32 *bits*, esto se conoce como extensión de signo en la representación complemento a dos, si la representación de-codificada no abarca el *bit* 0, se debe llenar todos los *bits* vacíos menos significativos con el valor de 0.

Figura 3. **Ejemplo decodificación de inmediatos**

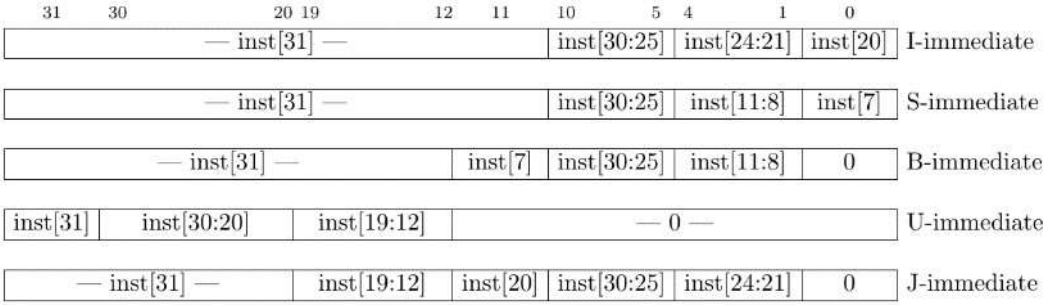


Fuente: elaboración propia, empleando FreeOffice PlanMaker 2018.

Por ejemplo, utilizando la figura 3 como referencia, se utiliza un formato de instrucción tipo B, se asignan los valores de *imm* en las posiciones correspondientes del número inmediato, se logra observar que los *bits* 31 al 13,

en el inmediato decodificado, se encuentran vacíos, por lo tanto el valor del *bit* más significativo es extendido hasta el *bit* 31, en este caso el *bit* más significativo es el *bit* 12 con un valor de 1, asimismo se observa que el *bit* 0 se encuentra vació, por lo tanto todos los *bits* vacíos menos significativos son rellenados con el valor cero, en este caso particular sólo el *bit* 0 se encuentra vació, dando como resultado el número binario 0b11111111111111111111101001101010 o el número entero -1 430.

Figura 4. **Decodificación valores inmediatos**



Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 17.

La figura 4 ilustra la decodificación de todos los formatos de instrucción que poseen valores inmediatos.

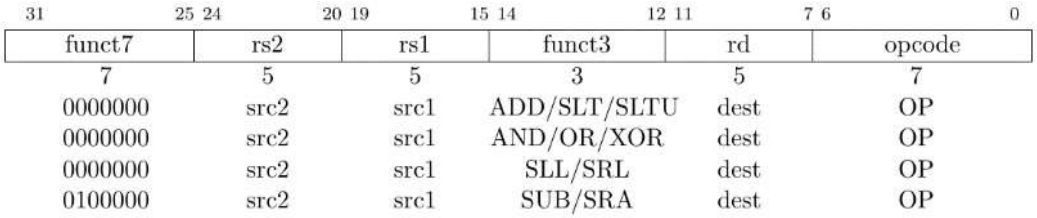
1.2.1.3. Instrucciones de computación de enteros

Estas instrucciones se caracterizan por realizar operaciones lógicas y aritméticas entre registros y, entre registro y valores inmediatos.

1.2.1.3.1. Instrucciones registro - registro

Utilizan el formato tipo R, todas las operaciones utilizan *rs1* y *rs2* como argumentos, escriben el resultado de la operación en el registro *rd*, *funct7* y *funct3* seleccionan el tipo de operación a realizar, su *opcode* es *OP*.

Figura 5. **Instrucciones registro-registro**



31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 19.

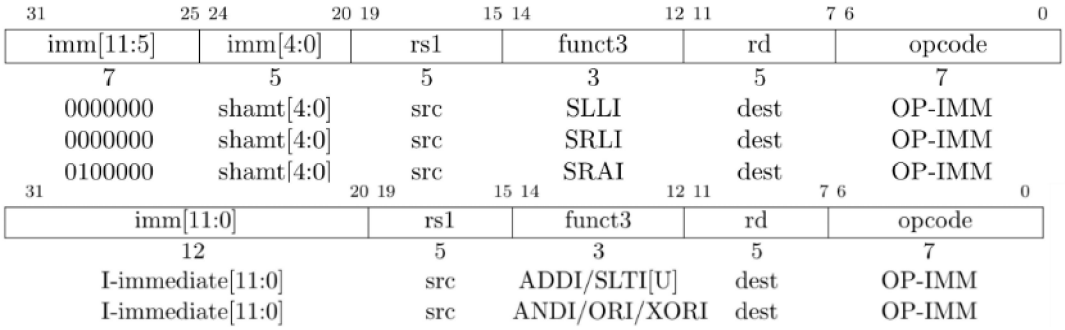
ADD realiza adición entre *rs2* y *rs1*. *SUB* realiza la resta entre *rs2* y *rs1*. *AND*, *OR*, *XOR* realizan la función booleana correspondiente entre *rs2* y *rs1*. *SLT* y *SLTU* realizan comparación con signo y sin signo, escribiendo 1 en *rd* si *rs1* < *rs2,* 0 en caso contrario.

- *SLL* realiza el desplazamiento lógico hacia la izquierda en el valor del registro *rs1* por la cantidad descrita en los primeros 5 *bits* de *rs2*.
- *SRL* realiza el desplazamiento lógico hacia la derecha en el valor del registro *rs1* por la cantidad descrita en los primeros 5 *bits* de *rs2*.
- *SRA* realiza el desplazamiento aritmético hacia la derecha en el valor del registro *rs1* por la cantidad descrita en los primeros 5 *bits* de *rs2*.

1.2.1.3.2. Instrucciones registro - inmediato tipo I

Utilizan el formato tipo I, utilizan *rs1* como primer operando, el valor decodificado de *imm* es utilizado como segundo operando, escriben el resultado de la operación en el registro *rd*, *funct3* selecciona el tipo de operación a realizar, su *opcode* es *OP-IMM*

Figura 6. **Instrucciones registro - inmediato tipo I**



Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 18.

ADDI, *SLTI*, *SLTIU*, *ANDI*, *ORI*, *XORI*, poseen la misma función que sus equivalentes registro - registro, con la única excepción de no usar *rs2* sino el valor inmediato decodificado como segundo operando.

SLLI, *SRLI*, *SRAI* realizan la misma función que sus equivalentes registro-registro, debido a utilizar una arquitectura de 32 *bits* el desplazamiento más grande posible es de 31 *bits*, los números del 0 al 31 puede ser codificados en los primeros 5 *bits* del valor inmediato, por lo tanto, los *bits* 25 a 31 no son

utilizados en su ejecución con la excepción del *bit* 30 que es utilizado para diferenciar *SLRI* y *SRAI*.

1.2.1.3.3. Instrucciones *LUI* y *AUIPC*

Estas instrucciones son utilizadas para escribir en registros, valores inmediatos relativamente grandes.

Figura 7. **Instrucciones *LUI* y *AUIPC***

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 19.

LUI, utiliza el formato tipo U, escribe el valor inmediato en el registro *rd*.

AUIPC, utiliza el formato tipo U, escribe la suma entre *PC* y el valor inmediato en el registro *rd*.

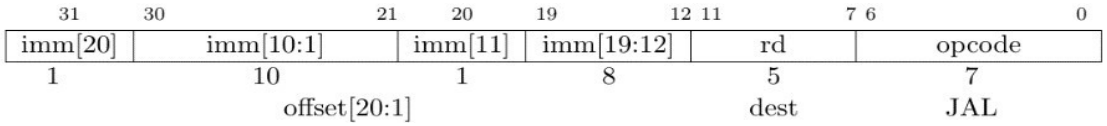
1.2.1.4. Instrucciones de transferencia de control

RV32I establece dos tipos de instrucciones utilizados en la transferencia de control, saltos condicionales y saltos incondicionales.

1.2.1.4.1. Saltos incondicionales

Estas instrucciones son utilizadas para realizar saltos y retornos de subrutinas.

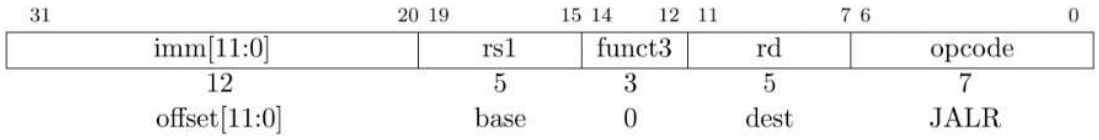
Figura 8. **Instrucción *JAL***



Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 21.

JAL, utiliza el formato tipo J, su *opcode* es *JAL*, escribe la dirección de la siguiente instrucción, *PC* + 4, en el registro *rd*, escribe la suma entre *PC* y el valor inmediato en *PC*.

Figura 9. **Instrucción *JALR***



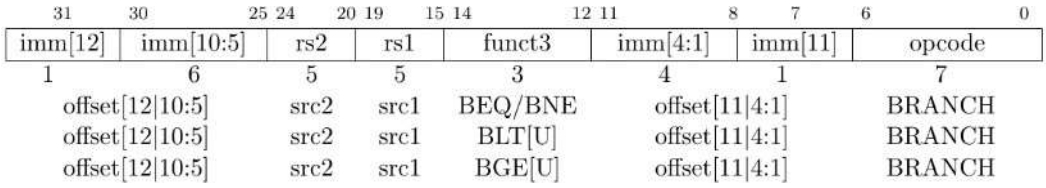
Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 21.

JALR, utiliza el formato tipo I, su *opcode* es *JALR*, escribe la dirección de la siguiente instrucción, *PC* + 4, en el registro *rd*, suma *rs1* y el valor inmediato, luego establece el *bit* menos significativo a cero para ser escrito en *PC*.

1.2.1.4.2. Saltos condicionales

Todas las instrucciones de salto condicional utilizan el formato tipo B. Su *opcode* es *BRANCH*. Comparan los registros *rs1* y *rs2*, si la comparación es verdadera escribe la suma entre *PC* y el valor inmediato en *PC*, en caso contrario *PC* es la siguiente instrucción. Si comparación verdadera: *PC* = *PC* + *imm*, de lo contrario *PC* = *PC* + 4.

Figura 10. **Instrucciones de salto condicional**



31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

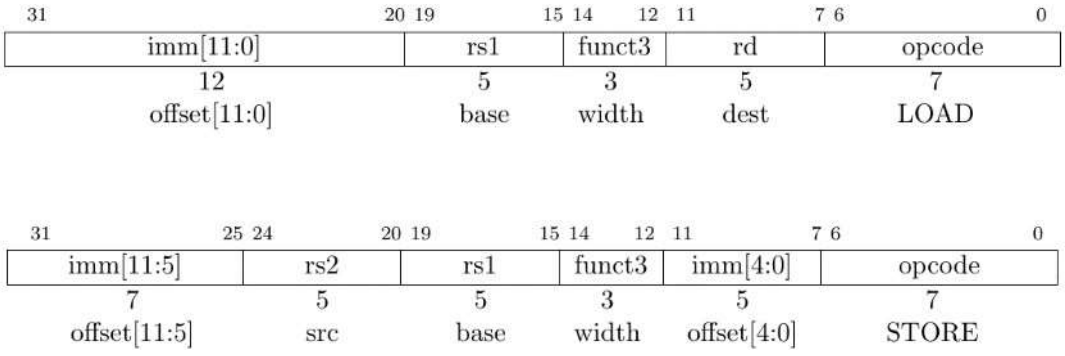
Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 22.

BEQ y *BNE*, su comparación es verdadera si *rs1* y *rs2* son iguales o diferentes, respectivamente. *BLT* y *BLTU*, su comparación es verdadera si *rs1* es menor que *rs2* utilizando valores con signo o sin signo, respectivamente. *BGE* y *BGEU*, su comparación es verdadera si *rs1* es mayor o igual que *rs2* utilizando valores con signo o sin signo, respectivamente.

1.2.1.5. Instrucciones de acceso de memoria

Estas instrucciones son utilizadas para realizar transacciones entre registros y memoria.

Figura 11. **Instrucción *LOAD* y *STORE***



Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 24.

LOAD utiliza el formato tipo I. Su *opcode* es *LOAD*. Transfieren un valor de memoria al registro *rd*. La dirección de memoria a acceder es calculada sumando *rs1* y el valor inmediato. *funct3* define si el valor extraído de memoria es de 32, 16 ó 8 *bits*, además para valores de 16 y 8 *bits*, define si debe realizar una extensión de signo o extender el valor con ceros.

STORE utiliza el formato tipo S. Su *opcode* es *STORE*. Transfieren el valor del registro *rs2* a memoria. La dirección de memoria a acceder es calculada sumando *rs1* y el valor inmediato. *funct3* define si los primeros 32, 16 ó 8 *bits* del registro *rs2* son transferidos a memoria.

1.2.2. RISC-V arquitectura privilegiada

"RISC-V describe una arquitectura privilegiada, que cubre aspectos adicionales a los mencionados en la arquitectura sin privilegios, que es la que se ha descrito hasta el momento, estos aspectos incluyen instrucciones privilegiadas

y funciones adicionales necesarias para ejecutar sistemas operativos y la conexión de dispositivos externos"[4].

RISC-V define tres modos en que un procesador puede operar, modo máquina, M-*mode*, modo usuario, U-*mode*, y modo supervisor, S-*mode*, cada modo posee diferentes privilegios en el acceso del hardware. Dependiendo de la intención del uso del procesador es como se deben de implementar los modos de operación, por ejemplo, para aplicaciones en sistemas de embebidos simples únicamente debe de implementarse M-*mode*, para sistemas embebidos seguros debe de implementarse M-*mode* y U-*mode*, por último, para aplicaciones que deseen ejecutar sistemas operativos tipo Unix, como Linux, OpenBSD, Mac, deben ser implementados los modos, M-*mode*, S-*mode* y U-*mode*.

M-*mode* es el modo con la mayor cantidad de privilegios, posee el acceso total de todos los recursos del procesador, para cualquier implementación de RISC-V es obligatorio poseer M-*mode*.

Debido a que este trabajo será desarrollado en un procesador para sistemas embebidos simples, únicamente se presentará la arquitectura privilegiada mínima que define el *ISA* RISC-V del modo de operación M-*mode*.

1.2.2.1. Registros de control y estado, *CSR*

RISC-V define un espacio separado de direcciones para 4 096 registros de control y estado, en otras palabras, es un espacio de direcciones independiente

[4] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 1.

de memoria central, utilizados para observar el estado del procesador y modificar su funcionamiento.

1.2.2.1.1. *Misa*

Es un registro de solo de lectura de 32 *bits* de dirección 0x301. Utilizado para indicar la arquitectura implementada 32, 64 o 128 *bits*, adicionalmente contiene información de extensiones presentes en la implementación.

Figura 12. **Registro *Machine ISA*, *misa***

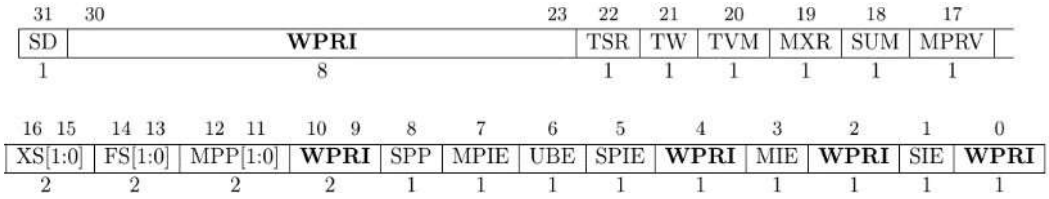
Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 15.

Por ser una implementación RV32I, es decir arquitectura de 32 *bits* y sin extensiones adicionales, este registro debe de poseer el valor de 0x4000100

1.2.2.1.2. *Mstatus*

Es un registro de lectura y escritura de 32 *bits,* el cual es accedido a través de la dirección 0x300.

Figura 13. **Registro *Machine Status*, *mstatus***



Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 20.

Los *bits MPP* son utilizados para especificar el modo de operación actual, debido a que únicamente se implementara M-*mode*, *MPP* debe poseer un valor constante de 0b11.

MIE es utilizado para habilitar interrupciones globalmente, 0 corresponde a interrupciones deshabilitadas, 1 corresponde a interrupciones habilitadas. Al momento de activarse una interrupción o excepción *MIE* debe de ser establecido a 0, y el valor previo de *MIE* debe ser guardado en *MPIE*. Al retornar de la interrupción utilizando la instrucción *MRET*, *MIE* debe ser establecido al valor de *MPIE* adicionalmente *MPIE* debe ser establecido a 1.

El resto de *bits* no son utilizados, por lo tanto, deben de poseer un valor constante de 0.

1.2.2.1.3. *Mtvec*

Es un registro de lectura y escritura con 32 *bits*, el cual es accedido a través de la dirección 0x305.

Figura 14. **Registro *Machine Trap-Vector Base-Address*, *mtvec***

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 28.

BASE, guarda una dirección de memoria.

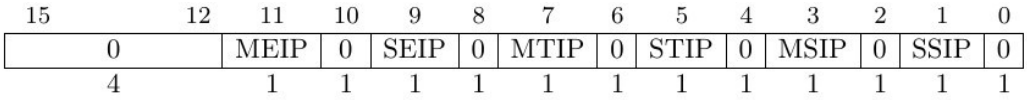
Cuando *MODE* posee el valor de 0b00, excepciones e interrupciones provocan que *PC* tome el valor de *BASE*. Cuando *MODE* posee el valor de 0b01, excepciones provocan que *PC* tome el valor de *BASE*, interrupciones provocan que *PC* tome el valor de la suma entre *BASE* y el número de la causa de interrupción multiplicada por cuatro.

Los valores de *MODE* 0b10 y 0b11, no se encuentran definidos por lo tanto no deben de ser utilizados.

1.2.2.1.4. *Mip* y *mie*

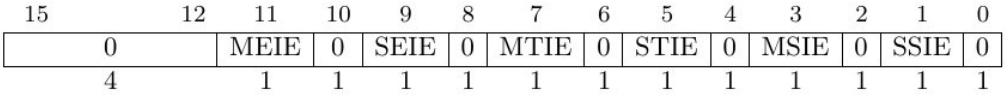
Mie es un registro de lectura y escritura de 15 *bits* de dirección 0x304. *mip* es un registro solo de lectura de 15 *bits* de dirección 0x344.

Figura 15. **Registro *Machine Interrupt-Pending*, *mip***



Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 32.

Figura 16. **Registro *Machine Interrupt-Enable*, *mie***



Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 32.

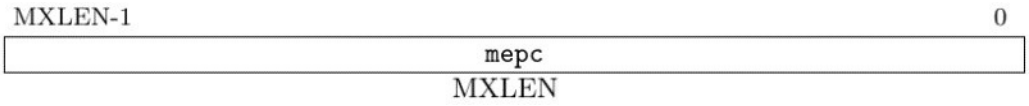
Bits mip.MEIP y *mie.MEIE* son los *bits* de interrupción pendiente e interrupción habilitada, para interrupciones externas. *MEIP* del registro *mip* es un registro de solo lectura, y solamente puede ser modificado por un controlador de interrupciones externo.

El resto de *bits* no son utilizados y poseerán el valor constante de 0.

1.2.2.1.5. *Mepc*

Es un registro de lectura y escritura con 32 *bits,* el cual es accedido a través de la dirección 0x341.

Figura 17. **Registro *Machine Exception Program Counter*, *mpec***



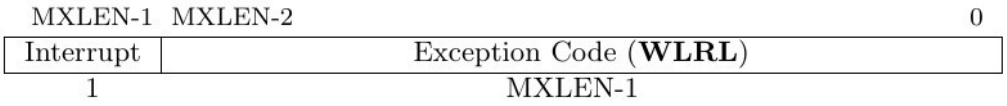
Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 38.

Cuando una excepción o interrupción ocurre, *mepc* es escrito con el valor de *PC* de la instrucción que fue interrumpida o generó la excepción.

1.2.2.1.6. *Mcause*

Es un registro de lectura y escritura con 32 *bits,* el cual es accedido a través de la dirección de 0x342.

Figura 18. **Registro *Machine Cause*, *mcause***



Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 39.

Cuando una excepción o interrupción ocurre, el registro *mcause* es escrito con la representación numérica de su causa, *bit* 31 es utilizado para diferenciar entre excepción o interrupción, 0 y 1 respectivamente.

1.2.2.2. Excepciones e interrupciones

RISC-V define como excepción a una condición inusual asociada a una instrucción al momento de ser ejecutada. Una interrupción es un evento asíncrono externo capaz de realizar una inesperada transferencia de control.

A continuación, se presentan las diferentes causas que pueden causar una excepción o interrupción:

Tabla II. **Causas de excepciones e interrupciones**

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Available for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault

Continuación tabla II.

0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Available for custom use*
0	32–47	*Reserved*
0	48–63	*Available for custom use*
0	>64	*Reserved*

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 40.

La única causa de interrupciones, con el código 11, es una interrupción externa de M-*mode*, es causada cuando un dispositivo externo cambia el valor de entrada de un pin llamado *IRQ*.

Por el contrario, en la implementación realizada en este trabajo, existen diferentes causas que pueden generar una excepción.

Excepción por desalineación en dirección de instrucción, con código 0, es causada por instrucciones de control de flujo, instrucciones de salto incondicional generan esta excepción si la dirección objetivo no es un múltiplo de cuatro. Para instrucciones de salto condicional únicamente se genera esta excepción si la dirección objetivo no es múltiplo de cuatro y la condición a evaluar es verdadera.

Excepción por instrucción ilegal, con código 2, es causado cuando se intenta ejecutar una instrucción no implementada o no existente en el *ISA*.

Excepción *breakpoint*, con código 3, es una excepción generada al ejecutar la instrucción *EBREAK*.

Excepción por desalineación en dirección de instrucción *LOAD*, con código 4, es causado si la dirección destino de una transferencia de 32 o 16 *bits* no es múltiplo de 4 o 2 respectivamente.

Excepción por desalineación en dirección de instrucción *STORE*, con codigo 6, es causado si la dirección destino de una transferencia de 32 o 16 *bits* no es múltiplo de 4 o 2 respectivamente.

Excepción llamada de entorno desde M-*mode*, con código 11, es una excepción generada al ejecutar la instrucción *ECALL*.

El resto de las excepciones no serán mencionadas debido a que no pueden ser generadas en la implementación de este trabajo.

Si una instrucción genera dos o más excepciones la causa que será reportada será escogida basado en un sistema de prioridad ejemplificado en el siguiente cuadro:

Tabla III. **Prioridad de excepciones**

Priority	Exception Code	Description
Highest	3	Instruction address breakpoint
	12	Instruction page fault
	1	Instruction access fault
	2	Illegal instruction
	0	Instruction address misaligned
	8, 9, 11	Environment call
	3	Environment break
	3	Load/Store/AMO address breakpoint
Optionally, these may have lowest priority instead.	6	Store/AMO address misaligned
	4	Load address misaligned
	15	Store/AMO page fault
	13	Load page fault
	7	Store/AMO access fault
	5	Load access fault

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 41.

Al momento de generarse una excepción o interrupción el procesador debe de realizar las siguientes acciones:

- Escribir la dirección de la instrucción que generó la excepción o fue interrumpida, en el registro *CSR mepc*.

- Escribir la causa de la excepción o interrupción en el registro *CSR mcause.*

- Escribir *PC* con el valor del registro *CSR mtvec*, dependiendo del valor escrito en los *bits MODE* de *mtvec*.

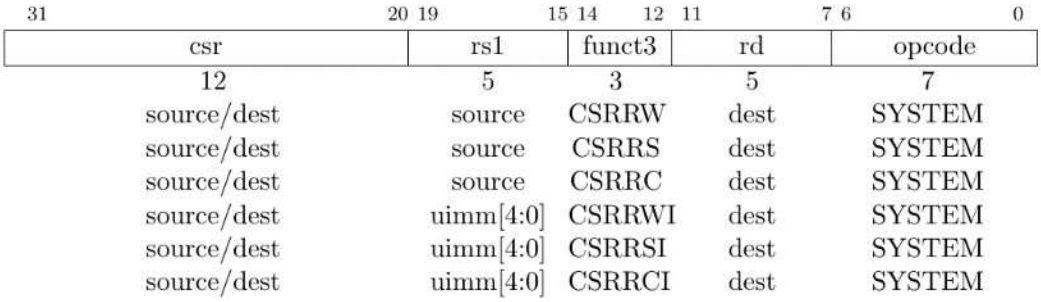
- Escribir el *bit MPIE* con el valor de *MIE* y cambiar el valor de *MIE* a 0, del registro *mstatus.*

Para que una interrupción sea generada el *bit MIE* del registro *mstatus*, debe poseer el valor de 1, adicionalmente en el registro *mie* y *mip* los *bits* correspondientes al tipo de interrupción deben poseer el valor de 1.

1.2.2.3. Instrucciones *CSR*

Utilizan el formato tipo I, utilizan *csr* para especificar la dirección del registro *CSR* a modificar/leer, *funct3* indica qué función ejecutar, su *opcode* es *SYSTEM*.

Figura 19. **Instrucciones *CSR***



31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 55.

CSRRW escribe el registro *csr* con el valor del registro *rs1*, simultáneamente *rd* es escrito con el valor del registro *csr*.

CSRRS lee el valor del registro *csr* para ser escrito en el registro *rd.* El valor del registro *rs1* es utilizado como una máscara de *bits* que indica la posición de los *bits* del registro *csr* que deben ser establecidos a 1. Cualquier *bit* que posea el valor 1 en el registro rs1 causará que el *bit* correspondiente del registro *csr* sea escrito con el valor de 1.

CSRRC lee el valor del registro *csr* para ser escrito en el registro *rd.* El valor del registro *rs1* es utilizado como una máscara de *bits* que indica la posición de los *bits* del registro *csr* que deben ser establecidos a 0. Cualquier *bit* que posea el valor 1 en el registro *rs1* causará que el *bit* correspondiente del registro *csr* sea escrito con el valor de 0.

Si *rs1* es igual al registro cero, *CSRRC* y *CSRRS* no modificarán el valor del registro *csr*.

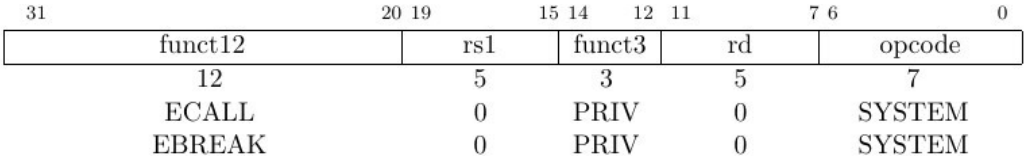
CSRRWI, *CSRRSI* y *CSRRCI* realizan la misma función que *CSRRW*, *CSRRS*, *CSRRC*, respectivamente, con la única diferencia que utilizan el valor inmediato extendido con ceros de *uimm*, contrariamente de utilizar el valor del registro *rs1*.

Si *uimm* es igual a cero, *CSRRCI* y *CSRRSI* no modificarán el valor del registro *csr*.

1.2.2.4. Instrucciones privilegiadas

Utilizan el formato tipo I, utilizan *funct12* para diferenciar la función a ejecutar, *rs1* y *rd* no son utilizados, su *opcode* es *SYSTEM.*

Figura 20. **Instrucción *ECALL* y *EBREAK***

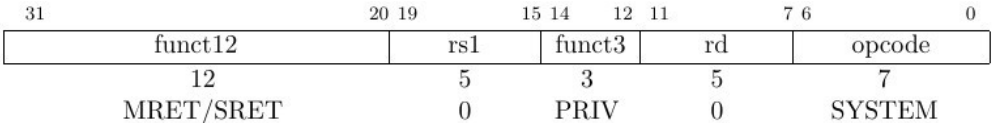


31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 43.

ECALL genera una excepción de causa “llamada de entorno desde M-*mode*”, código 11. *EBREAK* genera una excepción de causa *breakpoint*, código 3.

Figura 21. **Instrucción *MRET***



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
MRET/SRET	0	PRIV	0	SYSTEM

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 43.

MRET, escribe *PC* con el valor del registro *CSR mepc*. Adicionalmente, en el registro *mstatus*, escribe el *bit MIE* con el valor del *bit MPIE*, y establece *MPIE* con el valor de 1.

La instrucción *SRET* no es implementada por lo tanto debe generar una excepción de causa instrucción ilegal.

1.2.3. Lista de Instrucciones

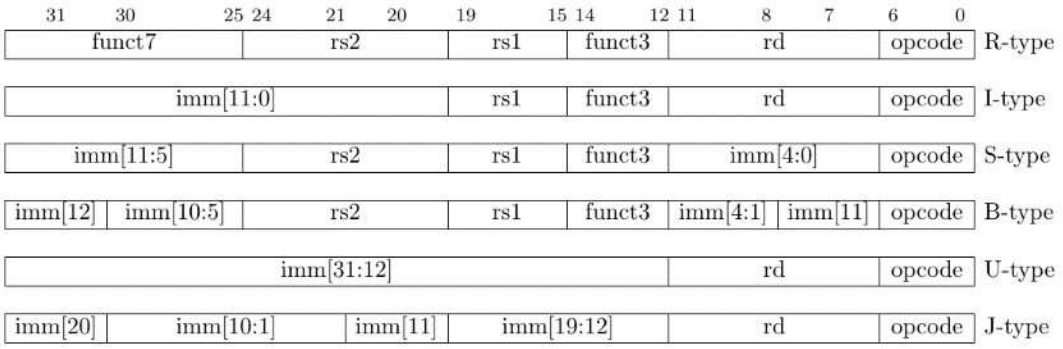
A continuación, se muestra todos los *opcodes* utilizados y su representación binaria, adicionalmente de una lista con todas las instrucciones implementadas:

Tabla IV. **Representación binaria de *opcodes***

opcode	Valor binario
LOAD	0000011
OP-IMM	0010011
AUIPC	0010111
STORE	0100011
OP	0110011
LUI	0110111
BRANCH	1100011
JALR	1100111
JAL	1101111
SYSTEM	1110011

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2018.

Figura 22. **Formatos de instrucciones RISC-V**



Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 16.

Tabla V. **Lista de Instrucciones RV32I**

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 130.

Tabla VI. **Lista de instrucciones privilegiada**

0000000	00010	00000	000	00000	1110011	URET
0001000	00010	00000	000	00000	1110011	SRET
0011000	00010	00000	000	00000	1110011	MRET

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. p. 118.

Tabla VII. **Lista de instrucciones *CSR***

csr	rs1	001	rd	1110011	CSRRW
csr	rs1	010	rd	1110011	CSRRS
csr	rs1	011	rd	1110011	CSRRC
csr	uimm	101	rd	1110011	CSRRWI
csr	uimm	110	rd	1110011	CSRRSI
csr	uimm	111	rd	1110011	CSRRCI

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 131.

1.3. Implementación

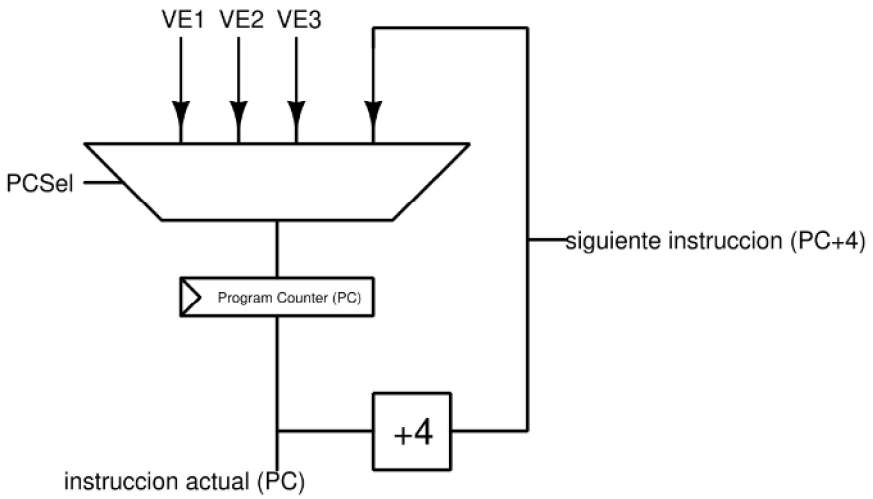
A continuación, se implementa la micro-arquitectura la cual ejecutará las instrucciones del *ISA* previamente mencionado.

1.3.1. Contador de programa

Las instrucciones para ejecutar de un programa de computadora se encuentran alojadas en memoria, por lo tanto, se necesita un circuito que señale la dirección de memoria de la instrucción a ejecutar, este mecanismo posee el nombre de contador de programa o *program counter, PC*, en inglés.

Al finalizar la ejecución de una instrucción, una instrucción finaliza su ejecución en el flanco de subida del reloj, el contador de programa debe de actualizar su valor automáticamente a la siguiente instrucción a ejecutar, asimismo debe ser capaz de ser modificado con valores externos, por ejemplo, al ejecutar instrucciones de transferencia de control.

Figura 23. **Contador de programa**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Se utiliza un registro el cual guarda la dirección de memoria de la instrucción, como lo define la *ISA*, las instrucciones ocupan un tamaño de 4 *bytes* o 32 *bits*, esto quiere decir que cada instrucción se encuentra separada por un espacio 4 *bytes*, por lo tanto, si a la instrucción actual se le suma el valor de cuatro, el resultado es la dirección de la siguiente instrucción a ejecutar, el cual es representado por el bloque *+4*.

Si se desea modificar el valor de *PC*, como en instrucciones de transferencia de control, se utiliza un multiplexor para seleccionar el valor que

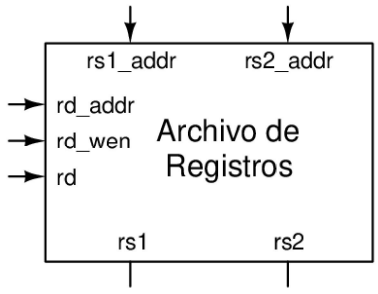
será escrito *PC*, siendo *PCSel* la señal que indicará al multiplexor el valor a utilizar.

Por ejemplo, se sabe que un programa inicia en la localidad de memoria 0x200, por lo tanto, *PC* tendrá un valor inicial de 0x200, al finalizar la ejecución de dicha instrucción, *PCSel* dictará el valor a ser escrito, en este caso *PCSel* escoge siguiente instrucción, *PC*+4, por lo tanto, *PC* poseerá el valor de 0x204. Si no se ejecuta ninguna instrucción de transferencia de control *PC* cambiará su valor en incrementos de cuatro, 0x208, 0x20C, 0x0210, 0x214, ..., *PC*+4. En caso contrario, suponiendo que Valor Externo 1, *VE1*, posee el valor de 0x1BA8 y se ejecuta una instrucción de transferencia de control, *PCSel* escoge *VE1* y *PC* poseerá el valor de 0x1BA8.

1.3.2. Archivo de registros

El archivo de registros es el conjunto de los treinta y dos registros descritos por el *ISA*, cada registro posee la capacidad de ser leído en cualquier momento, asimismo, con excepción del registro cero, cada registro puede ser escrito en el flanco de subida del reloj.

Figura 24. **Archivo de registros**



Fuente: elaboración propia, empleando Xcircuit v3.10.

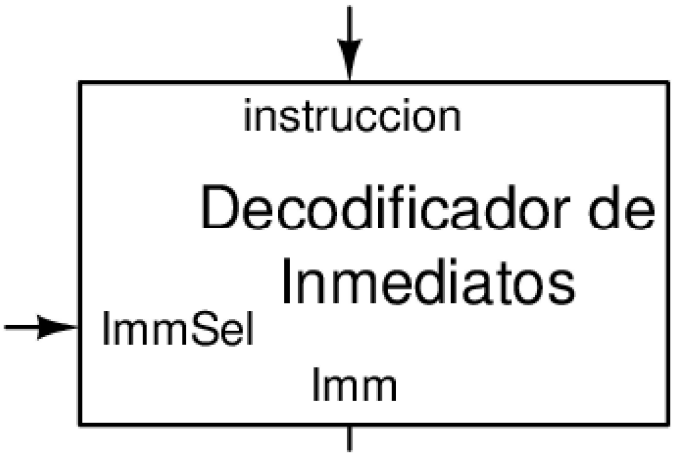
Como el mayor número de argumentos en una instrucción son dos registros, el archivo de registros posee dos puertos de lectura, donde *rs1_addr* y *rs2_addr* se utilizan para seleccionar uno de los treinta y dos registros para ser desplegado en *rs1* y *rs2* respectivamente.

Se utiliza un único puerto de escritura ya que, como máximo se modifica el valor de un registro por instrucción, *rd_addr* selecciona el registro a escribir, *rd* posee el valor a ser escrito. En algunas instrucciones no es necesario modificar el valor de un registro, por lo tanto, *rd_wen* es utilizado para habilitar o deshabilitar la escritura de un registro.

1.3.3. Decodificador de inmediatos

Como su nombre lo indica es utilizado para la decodificación de valores inmediatos.

Figura 25. **Decodificador de inmediatos**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Utiliza los *bits* 7 al 31 de la instrucción, los cuales poseen el valor inmediato codificado.

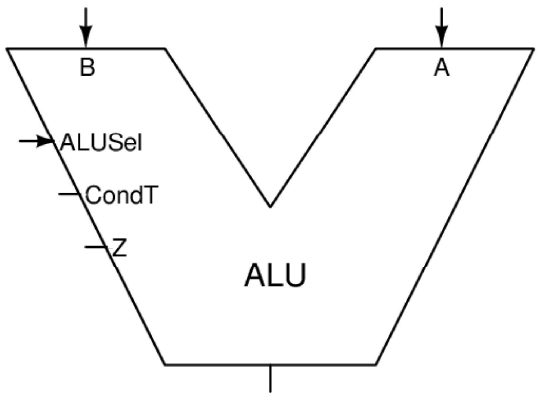
Adicional a los cinco tipos de inmediatos descritos, se agregó un sexto tipo de inmediato “Tipo-*CSR*”, este tipo de inmediato es utilizado en las instrucciones *CSR*, su valor inmediato es decodificado utilizando los *bits* 19 al 15 de la instrucción, *instr[19:15]*, en los *bits* 4 al 0 en el valor decodificado, *imm[4:0] = instr[19:15]*, los *bits* 31 al 5 son escritos con ceros, *imm[31:5] = 0*.

Se utiliza la señal *ImmSel* para seleccionar el tipo de inmediato a utilizar.

1.3.4. Unidad Lógica Aritmética

La unidad lógica aritmética o *Aritmetic Logic Unit, ALU*, en inglés es el encargado de realizar todas las operaciones aritméticas y lógicas en el procesador.

Figura 26. ***ALU***



Fuente: elaboración propia, empleando Xcircuit v3.10.

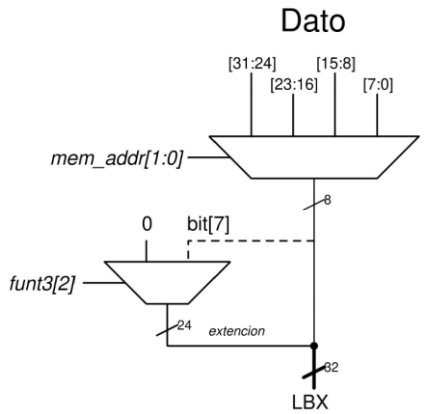
Se utiliza un multiplexor controlado por la señal *ALUSel*, para seleccionar el resultado de las diferentes operaciones aritméticas y lógicas disponibles, adicionalmente a las operaciones mencionadas en las instrucciones registro-registro, se agregó la operación A el cual devuelve el valor intacto del argumento A. B devuelve el valor intacto del argumento B. *CLEAR* utiliza los *bits* que posean el valor de uno en el argumento A para provocar que los *bits* correspondientes del argumento B sean escritos con el valor de cero.

La señal *Z* es utilizada para indicar si el resultado posee el valor de cero, *CondT* posee el resultado de ejecutar las instrucciones *SLT* y *SLTU*.

1.3.5. Interfaz de memoria

Para poder realizar transferencias de memoria de tamaños diferentes a cuatro *bytes*, se implementó una interfaz de memoria, el cual realiza la manipulación de *bits* necesaria para ejecutar instrucciones como *LB*, *LBU*, *LH*, *LHU*, *SB*, *SH*.

Figura 27. ***LB* y *LBU***



Fuente: elaboración propia, empleando Xcircuit v3.10.

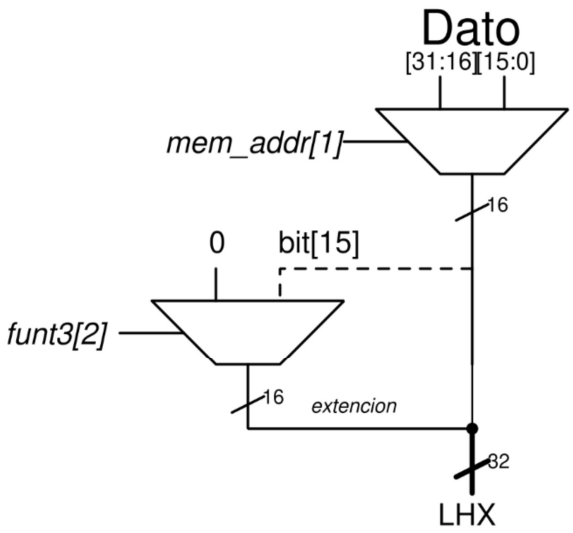
LB y *LBU* son instrucciones que realizan transferencias de un *byte*, pero el dato proveniente de memoria posee cuatro *bytes*, por lo tanto, *LB* y *LBU* poseen la capacidad de seleccionar cualquiera de los cuatro *bytes* provenientes de memoria. Para definir qué *byte* será extraído, se utiliza los *bits* uno y cero de la dirección de memoria, *mem_addr[1:0]*, como selector en un multiplexor.

Observando el formato de las instrucciones *LB*, *LBU*, *LH* y *LHU*, se logra apreciar que, en las instrucciones con extensión de signo, *LB* y *LH*, *funct3[2]* posee el valor de cero y en instrucciones sin extensión de signo, *LBU* y *LHU*, *funct3[2]* posee el valor de uno. Por lo tanto, *funct3[2]* es utilizado por un multiplexor para escoger el tipo de extensión, el cual es el *bit* más significativo del valor extraído en las instrucciones *LB* y *LH* o la constante cero en las instrucciones *LBU* y *LHU*.

Como se extrae un *byte* del dato proveniente de memoria, la extensión debe poseer un tamaño de tres *bytes*, con el propósito de completar los cuatro *bytes* o treinta y dos *bits* requeridos por el *ISA*.

Para completar el resultado, se concatena la extensión y el *byte* extraído.

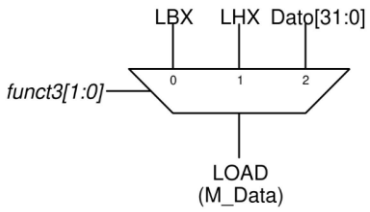
Figura 28. ***LH* y *LHU***



Fuente: elaboración propia, empleando Xcircuit v3.10.

Su funcionamiento es similar a *LB* y *LBU*, con la única diferencia siendo el tamaño del valor a extraer, en este caso dos *bytes*, por lo tanto, únicamente se utiliza el *bit* uno de la dirección de memoria para escoger el dato a extraer, debido a que este *bit* indica direcciones en múltiplos de dos *bytes*. Como el tamaño del valor extraído es mayor, el tamaño de la extensión es reducido, utilizando dos *bytes* y dos *bytes* respectivamente.

Figura 29. ***LOAD***



Fuente: elaboración propia, empleando Xcircuit v3.10.

El tamaño de la transferencia de memoria será determinado por los *bits* uno y cero de *funct3*, *funct3[1:0]*, los cuales escogerán *LBX*, *LHX* o los cuatro *bytes* del dato proveniente de memoria. El valor 0b00 escogerá *LBX*, 0b01 escogerá *LHX*, por último, 0b10 escogerá los cuatro *bytes* del dato proveniente de memoria, correspondiente a la instrucción *LW*.

Figura 30. **Ejemplo instrucciones *LOAD***

Memoria	
Dirección	Datos
...	-
0x200	0x12345678
0x204	0xDEADBEEF
0x208	0xBA987654
...	-

Dirección	0x200	0x201	0x202	0x203
LW	0x12345678	No Permitido	No Permitido	No Permitido
LH	0x00005678	No Permitido	0x00001234	No Permitido
LHU	0x00005678	No Permitido	0x00001234	No Permitido
LB	0x00000078	0x00000056	0x00000034	0x00000012
LBU	0x00000078	0x00000056	0x00000034	0x00000012

Dirección	0x204	0x205	0x206	0x207
LW	0xDEADBEEF	No Permitido	No Permitido	No Permitido
LH	0xFFFFBEEF	No Permitido	0xFFFFDEAD	No Permitido
LHU	0x0000BEEF	No Permitido	0x0000DEAD	No Permitido
LB	0xFFFFFFEF	0xFFFFFFBE	0xFFFFFFAD	0xFFFFFFDE
LBU	0x000000EF	0x000000BE	0x000000AD	0x000000DE

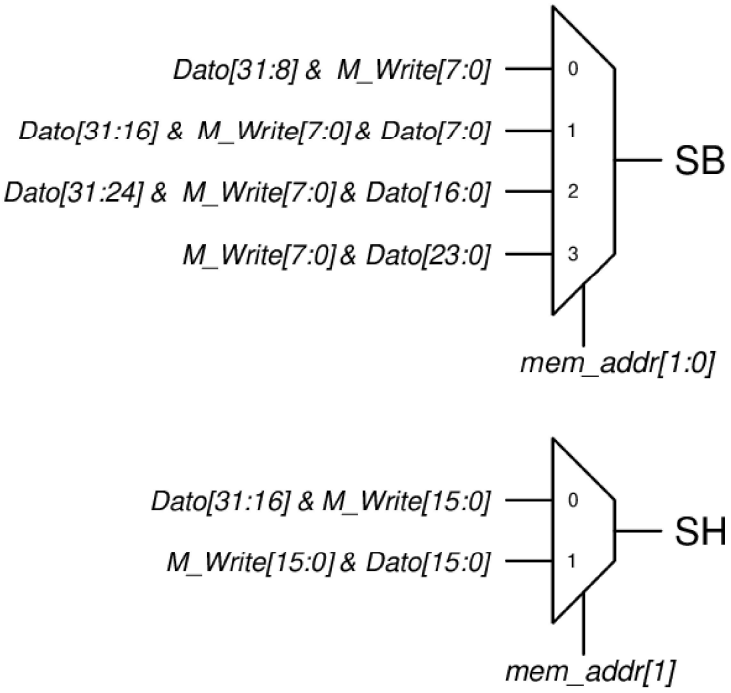
Dirección	0x208	0x209	0x20A	0x20B
LW	0xBA987654	No Permitido	No Permitido	No Permitido
LH	0x00007654	No Permitido	0xFFFFBA98	No Permitido
LHU	0x00007654	No Permitido	0x0000BA98	No Permitido
LB	0X00000054	0X00000076	0XFFFFFF98	0XFFFFFFBA
LBU	0X00000054	0X00000076	0X00000098	0X000000BA

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

En las casillas que poseen el valor de “No Permitido”, se refiere a las transferencias de memoria no posibles debido a que en esos casos se debe de acceder a dos ubicaciones diferentes de memoria simultáneamente o requiere lógica compleja, este tipo de acceso es el causante de excepciones por desalineación mencionados en la arquitectura privilegiada del *ISA*.

A diferencia de las instrucciones *LOAD*, las instrucciones *STORE* transfieren información de un registro hacia memoria, para transferencias de cuatro *bytes* esto no genera algún problema, como en la instrucción *SW*, pero para transferencia de tamaños menores a cuatro *bytes*, como las instrucciones *SB* y *SH*, se presenta el problema de alterar información ajena a la transferencia. Por lo tanto, se debe implementar un mecanismo que altere únicamente la información perteneciente a la transferencia.

Figura 31. ***SB, SH***



Fuente: elaboración propia, empleando Xcircuit v3.10.

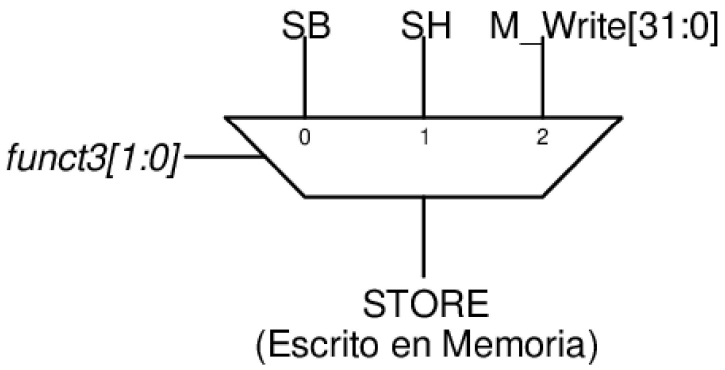
Similar a la instrucción *LB*, *SB* utiliza *mem_addr[1:0]* para seleccionar un *byte* pero en este caso no es la posición del *byte* a extraer sino la posición del

byte a insertar, como es mencionado en el *ISA* el *byte* a insertar son los primeros ocho *bits* del registro, *M_Write[7:0]*.

Las diferentes opciones del multiplexor son concatenaciones del *byte* a insertar y el valor del dato proveniente de memoria.

SH funciona de una manera similar, diferenciándose por el tamaño de la información a insertar dos *bytes*, dado por los primeros dieciséis *bits* del registro, *M_Write[15:0]*. Como su contraparte *LH*, *SH* utiliza el *bit* uno de la dirección de memoria para seleccionar la posición de inserción de los *bytes*.

Figura 32. ***STORE***



Fuente: elaboración propia, empleando Xcircuit v3.10.

El valor por ser escrito en memoria es escogido por, *funct3[1:0]*, el cual determinará si se utilizará el valor de *SB*, *SH* o el valor completo del registro, en la instrucción *SW*.

Figura 33. **Ejemplo instrucciones *STORE***

Memoria	
Dirección	Datos
...	-
0x200	0x12345678
0x204	0xDEADBEEF
0x208	0xBA987654
...	-

Dato a almacenar
0xDDCCBBAA

Dirección	0x200	0x201	0x202	0x203
SW	0xDDCCBBAA	No Permitido	No Permitido	No Permitido
SH	0x1234BBAA	No Permitido	0xBBAA5678	No Permitido
SB	0x123456AA	0x1234AA78	0x12AA5678	0xAA345678

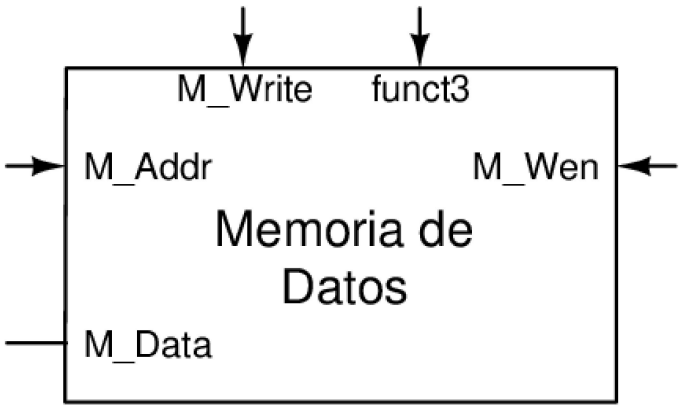
Dirección	0x204	0x205	0x206	0x207
SW	0xDDCCBBAA	No Permitido	No Permitido	No Permitido
SH	0xDEADBBAA	No Permitido	0xBBAABEEF	No Permitido
SB	0xDEADBEAA	0xDEADAAEF	0xDEAABEEF	0xAAADBEEF

Dirección	0x208	0x209	0x20A	0x20B
SW	0xDDCCBBAA	No Permitido	No Permitido	No Permitido
SH	0xBA98BBAA	No Permitido	0xBBAA7654	No Permitido
SB	0xBA9876AA	0xBA98AA54	0xBAAA7654	0xAA987654

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Nuevamente las casillas que poseen el valor de “No Permitido”, son operaciones que generan excepciones por desalineación.

Figura 34. **Interfaz de memoria**



Fuente: elaboración propia, empleando Xcircuit v3.10.

M_addr es la dirección de memoria en la cual se realizará la transferencia. *funct3* determinara el tamaño de la transferencia, en otras palabras, es utilizado para seleccionar la instrucción a ejecutar *LB*, *LBU*, *LH*, *LHU*, *LW*, *SB*, *SH* o *SW*. *M_Data* es la información proveniente de memoria al ejecutar alguna instrucción *LOAD*. *M_Write* es la información proveniente de un registro para ser escrita en memoria. *M_Wen* es utilizado para habilitar la escritura de memoria, utilizado en instrucciones *STORE*.

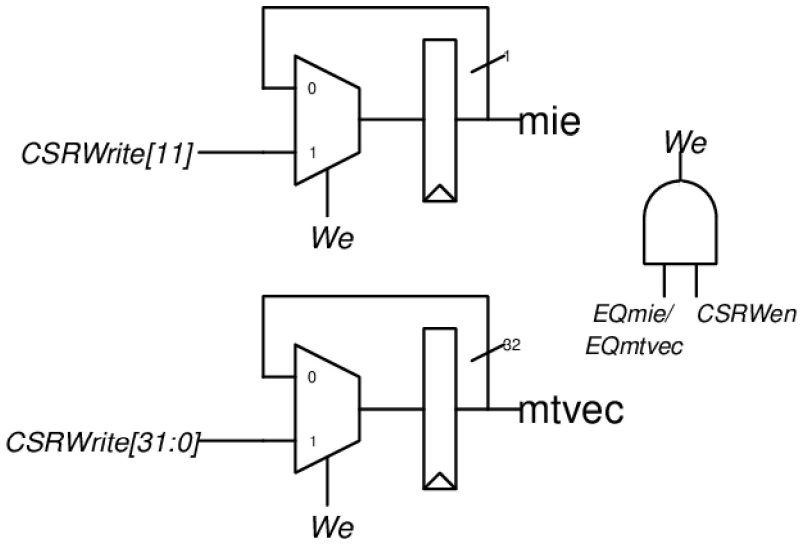
1.3.6. *CSR*

La unidad de registros de estado y control, posee la función de leer y escribir registros *CSR*, habilitar o deshabilitar interrupciones, guardar y modificar el valor de *PC* en el caso de interrupciones o excepciones.

Un registro *CSR* puede ser modificado por tres causas diferentes, la ejecución de una instrucción *CSR*, una excepción o interrupción y la ejecución de la instrucción *MRET*.

En la implementación presentada en este trabajo solo el registro *mstatus* puede ser modificado por estos tres factores, otros registros pueden ser escritos por una interrupción o excepción y por una instrucción *CSR*, por último, como los registros *mie* y *mtvec* únicamente pueden ser escritos por una instrucción *CSR*.

Figura 35. **Escritura de registros *mie* y *mtvec***



Fuente: elaboración propia, empleando Xcircuit v3.10.

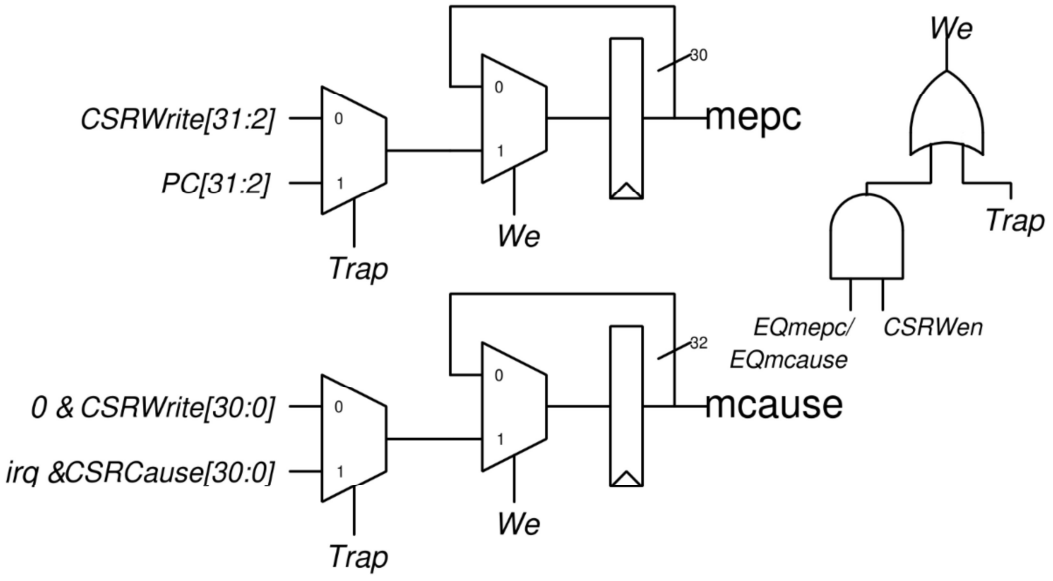
Se utiliza un multiplexor, en su primera entrada se conecta una retroalimentación del valor actual del registro, en su segunda entrada se conecta el valor se desea escribir, denotado como *CSRWrite*. Su selector es *We*. En los casos que *We* posea el valor de cero el registro mantendrá su valor, en caso contrario el registro será escrito con *CSRWrite*.

We únicamente tendrá el valor de uno cuando se cumplan dos condiciones, la dirección del registro proveniente de la instrucción *CSR*, es igual a dirección del registro que se desea escribir, en este caso 0x304 o 0x305 para el registro *mie* o *mtvec* respectivamente, la igualdad entre la dirección del registro y la dirección del registro que se desea escribir es representada por las señales *EQmie* y *EQmtvec* respectivamente. Adicionalmente la señal *CSRWen* debe poseer un valor de uno, utilizada para indicar globalmente la habilitación de la

escritura de registros *CSR*. El cumplimiento de estas dos condiciones es representado por la operación lógica *AND* entre *CSRWen* y *EQmie/EQmtvec.*

Para formar la señal *EQmie* o *EQmtvec*, o cualquier señal que indique igualdad a un número constante, se debe de convertir dicho número en su representación binaria, a las posiciones de *bits* correspondientes a ceros se les agrega una compuerta *NOT*, posteriormente se hace una operación *AND* de todos los *bits* que representan el número, esta función únicamente resultará en uno cuando el número a comparar sea igual al número deseado. Cabe mencionar que el número debe ser representado con la misma cantidad de *bits* que el número a comparar, en este caso doce *bits* de *CSRAddr*, señal utilizada para indicar la dirección del registro *CSR* que se desea escribir.

Figura 36. **Escritura de registros *mepc* y *mcause***



Fuente: elaboración propia, empleando Xcircuit v3.10.

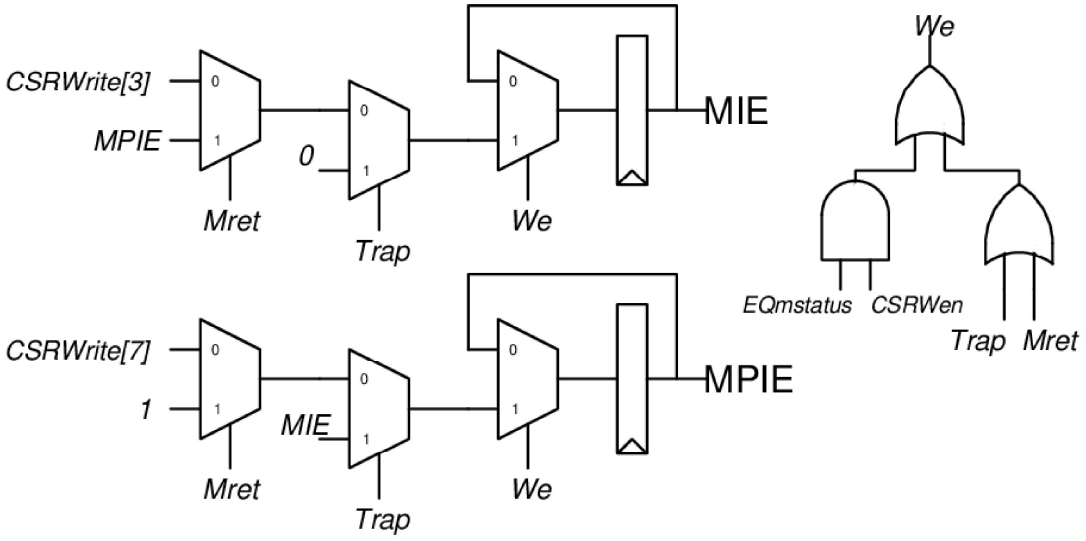
Los registros *mepc* y *mcause* pueden ser modificados por instrucciones *CSR* y por interrupciones o excepciones, la forma de ser escritos es similar a los registros *mie* y *mtvec*, se agrega un multiplexor adicional que es conectado a la segunda entrada del multiplexor controlado por *We*, el valor proveniente de la instrucción *CSR* es colocado en la primera entrada del nuevo multiplexor, a su segunda entrada se conecta la señal *CSRCause*/*PC*, que son los valores que deben ser escritos al momento de una interrupción o excepción, su seleccionador es la señal *Trap*.

Trap representa una interrupción o excepción, como lo menciona el *ISA* al momento de que el valor de *Trap* sea uno *mcause* debe guardar la representación numérica de la causa de la interrupción/excepción, asimismo *mepc* debe guardar la dirección de la instrucción interrumpida, *PC*.

We posee el valor de uno cuando cumple con las condiciones de escritura de una instrucción *CSR* o cuando *Trap* posee el valor de uno. Esto es representado por la función lógica *OR* entre *Trap* y *CSRWen AND EQmepc/EQmcause.*

Debido a que el *bit* más significativo de *mcause* es utilizado para diferenciar entre interrupciones y excepciones, no puede ser escrito por una instrucción *CSR*, por lo tanto, únicamente se utiliza los *bits* cero a treinta de *CSRWrite*, el *bit* treinta y uno de *mcause* es escrito con cero. Al momento que *Trap* sea uno, el *bit* treinta y uno de mcause es escrito con el valor de *irq*, señal utilizada para representar una interrupción.

Figura 37. **Escritura de *mstatus***



Fuente: elaboración propia, empleando Xcircuit v3.10.

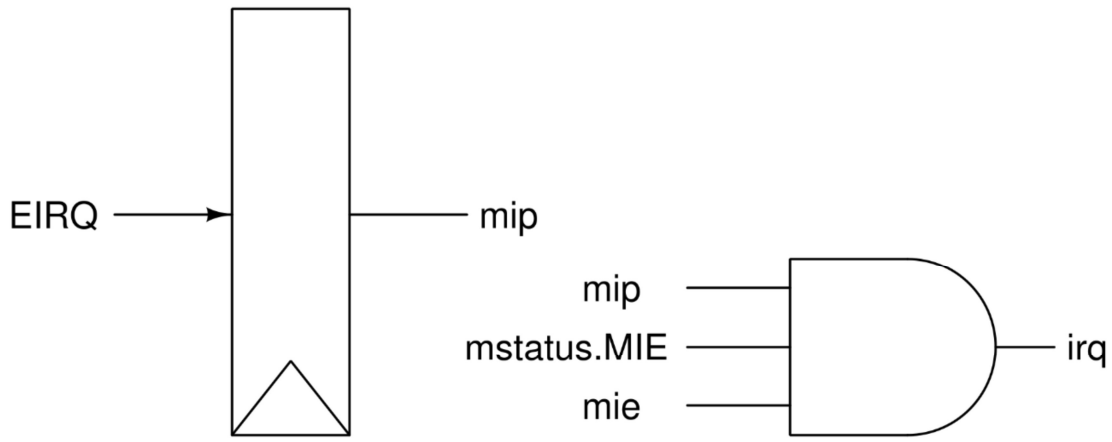
Adicional a poder ser escrito por tres causas, *mstatus* posee la peculiaridad que el *bit MIE* y *MPIE* son escritos de manera diferente en la misma situación, por ende, el circuito de escritura es diseñado por *bit*, al contrario de otros registros que un circuito abarca la totalidad del registro a escribir.

Se utiliza un multiplexor controlado por la señal *Mret*, utilizada para indicar la ejecución de la instrucción *MRET*, este nuevo multiplexor es conectado a la primera entrada del multiplexor controlado por *Trap*, su segunda entrada es conectada con el valor cero/*MIE* en el *bit MIE* y *MPIE* respectivamente. En el multiplexor controlador por *Mret* su primera entrada se encuentra conectada con *CSRWrite*, su segunda entrada es conectada con el valor *MPIE*/uno, en el *bit MIE* y *MPIE* respectivamente.

We poseerá el valor de uno si alguna de las tres señales, *Mret, Trap* o *EQmstatus AND Wen,* es verdadera.

Este diseño cumple con el comportamiento descrito por el *ISA*.

Figura 38. **Escritura registro *mip* y *irq***



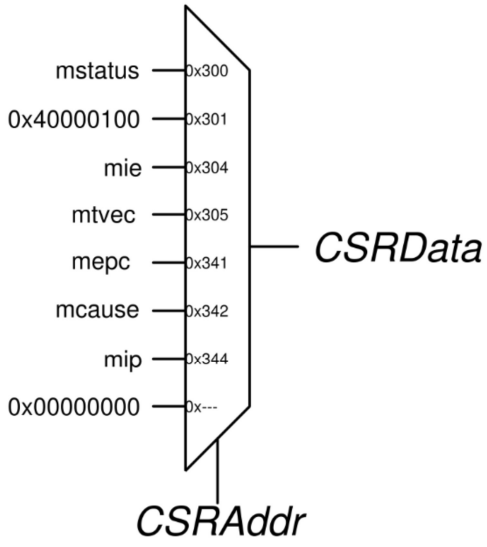
Fuente: elaboración propia, empleando Xcircuit v3.10.

El registro *mip* no puede ser escrito por el procesador, únicamente por un controlador externo de interrupciones, en esta implementación *mip* será escrito por un pin externo a el procesador llamado *EIRQ*, *mip* se encuentra conectado directamente con *EIRQ*.

El procesador detectara una interrupción, *irq*, cuando *mip*, *mie*, y el *bit MIE* del registro *mstatus* posean el valor de uno. Esto es representado con la operación lógica *AND*.

irq es el valor utilizado en la escritura del registro *mcause*.

Figura 39. **Lectura registros *CSR***



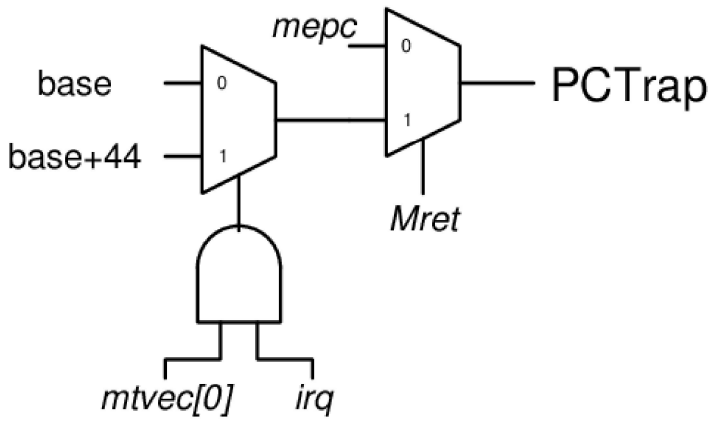
Fuente: elaboración propia, empleando Xcircuit v3.10.

Para la lectura de los registros *CSR* se conectan todos los registros a un multiplexor, el selector del multiplexor es el valor de *CSRAddr.* La dirección 0x301 corresponde al registro *misa*, como no puede ser modificado su valor es una constante, para el resto de las direcciones de registros *CSR* no implementados, representado por 0x---, se debe retornar ceros.

Los registros que no posean un tamaño de treinta y dos *bits*, son concatenados el valor de cero en los *bits* vacíos.

Adicionalmente a la lectura y escritura de registros *CSR*, se debe modificar el valor de *PC* al momento de activarse una interrupción o excepción y al ejecutar la instrucción *MRET*.

Figura 40. **Cálculo valor *PC* objetivo.**

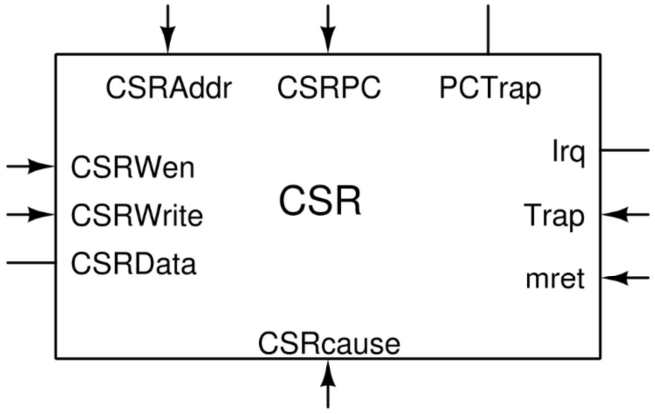


Fuente: elaboración propia, empleando Xcircuit v3.10.

Al momento de activarse una interrupción el *bit* cero del registro *mtvec* decide si el valor a escribir en *PC* es *base, mtvec[31:2]*, o es la suma entre *base* y la causa de la interrupción multiplicada por cuatro. En esta implementación solo existen interrupciones externas con una representación numérica de once, por lo tanto, en caso que *mtvec[0]* sea igual a uno y una interrupción sea generada, el valor a escribir en *PC* será la suma entre *base* y cuarenta y cuatro.

Si se ejecuta la instrucción *MRET*, *PC* será escrito con el valor del registro *mepc*.

Figura 41. ***CSR***



Fuente: elaboración propia, empleando Xcircuit v3.10.

CSRAddr es utilizado para seleccionar el registro a leer o escribir, *CSRData* es la información obtenida al leer un registro CSR, *CSRWen* habilita o deshabilita la escritura de un registro *CSR*, *CSRWrite* es el valor a escribir en el registro escogido por *CSRAddr. CSRcause* es utilizado para escribir la causa de una interrupción o excepción al momento de ser generados. *CSRPC*, es el valor del contador de programa, *mret* indica la ejecución de la instrucción *MRET*, *Trap* representa la activación de una interrupción o excepción, *PCTrap*, es el valor que debe ser escrito *PC* al momento de activarse una interrupción/excepción o al ejecutarse la instrucción *MRET*. *Irq* es la señal utilizada para indicar la presencia de una interrupción.

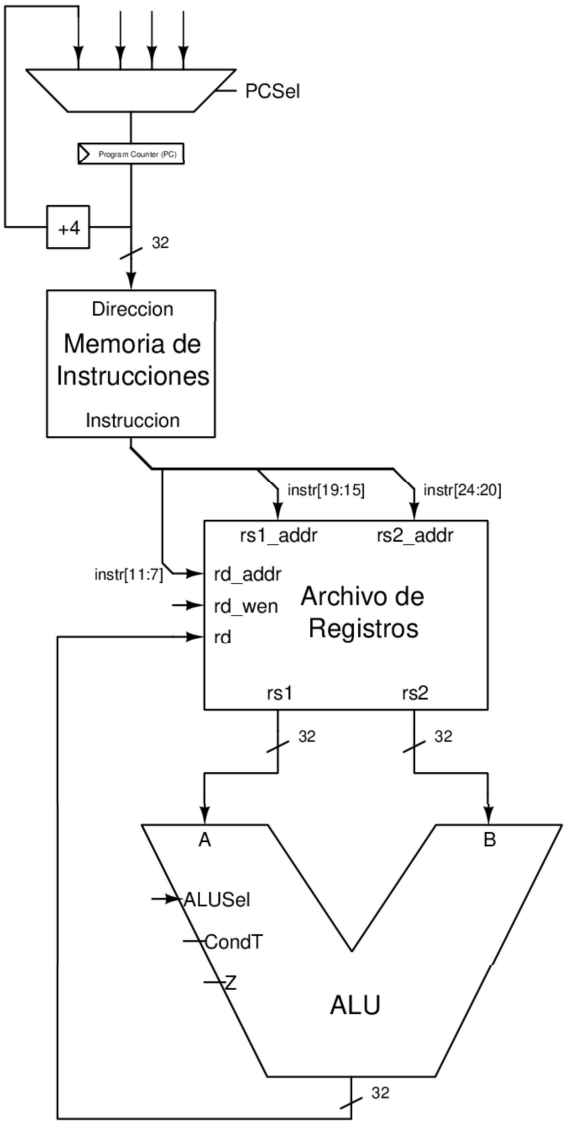
1.3.7. *Datapath*

El *datapath* es el recorrido que la información debe realizar entre los componentes descritos con anterioridad para ejecutar instrucciones.

1.3.7.1. Ejecución instrucciones registro-registro

A continuación, se presenta la figura 42 la cual contiene la organización de componentes necesarios para ejecutar instrucciones registro-registros.

Figura 42. ***Datapath*** **instrucciones registro-registro**



Fuente: elaboración propia, empleando Xcircuit v3.10.

El valor de *PC* es utilizado por la memoria de instrucciones para obtener la instrucción a ejecutar, los *bits* diecinueve al quince, *instr[19:15],* son utilizados para obtener el número de registro que la instrucción utilizará como primer argumento *rs1*, para seleccionar registro del segundo argumento *rs2*, se utilizan los *bits* veinticuatro al veinte, *instr[24:20]*.

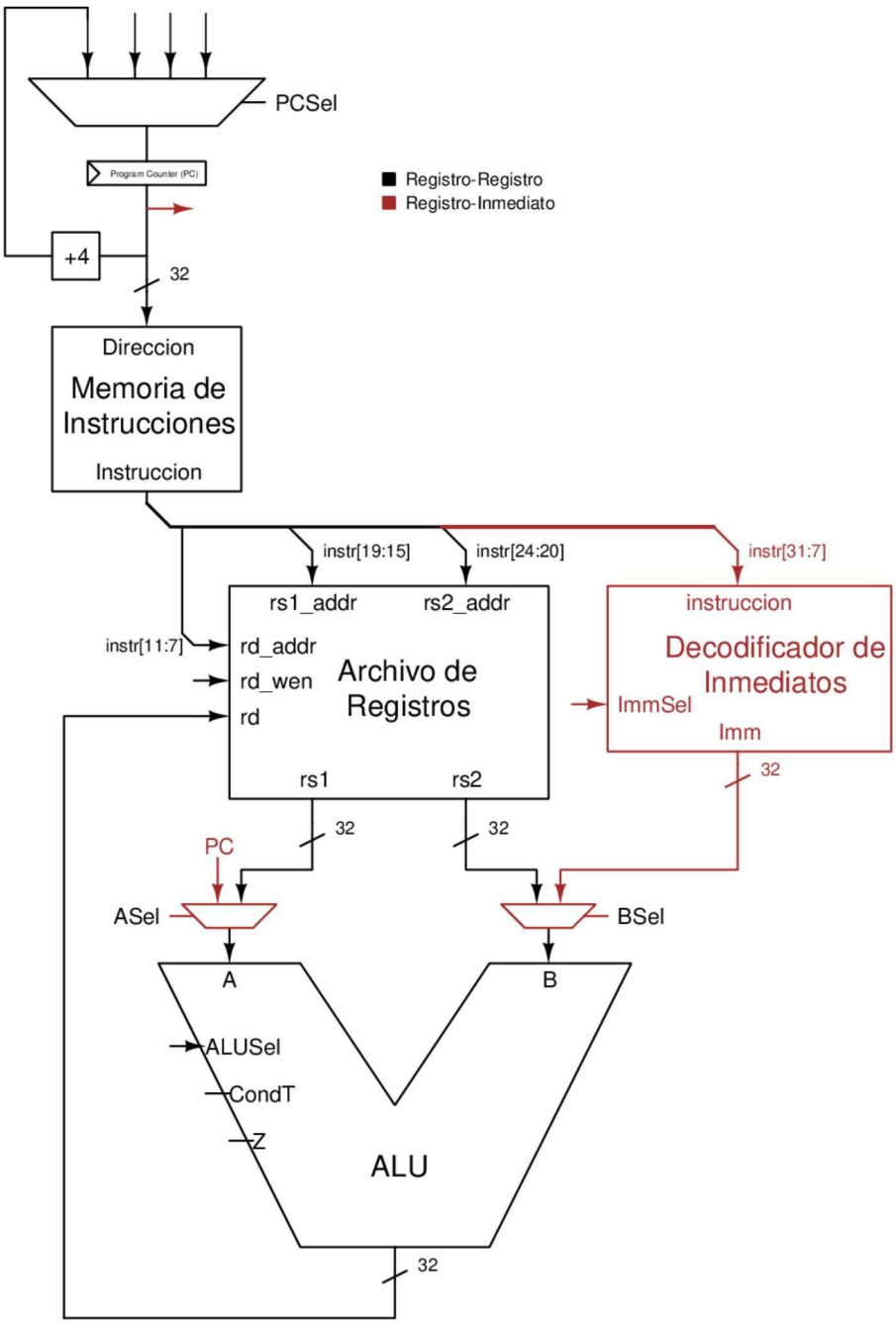
rs1 y *rs2*, son conectados a las entradas del *ALU*, *ALUSel* es utilizado para seleccionar la operación a realizar entre *rs1* y *rs2* dependiendo de la instrucción a ejecutar. El resultado es escrito en el registro *rd* el cual es seleccionado por los *bits* siete al once de la instrucción, *instr[11:7]*. Por lo tanto, *rd_wen* poseerá el valor de uno.

PC es escrito con el valor *PC*+4, lo que ejecutará la siguiente instrucción en el siguiente ciclo de reloj.

1.3.7.2. Ejecución instrucciones registro-inmediato

Al *datapath* existente se agrega el módulo decodificador de inmediatos, adicionalmente se añaden dos multiplexores, uno para cada entrada del *ALU*. Los *bits* siete al treinta y uno de la instrucción, *intr[31:7],* son conectados al módulo decodificador de inmediatos.

Figura 43. ***Datapath* instrucciones registro-inmediato**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Al observar los formatos de instrucciones, cuando se utilizan valores inmediatos generalmente se utiliza *rs1* y el valor inmediato, en otras palabras, no se utiliza el valor del registro *rs2*, por lo tanto, el valor inmediato es conectado al multiplexor que selecciona el valor del argumento *B* del *ALU*.

Al ejecutarse una instrucción registro-inmediato se utilizará *BSel* para seleccionar el valor inmediato como segundo operando, *ASel* seleccionará el valor del registro *rs1*.

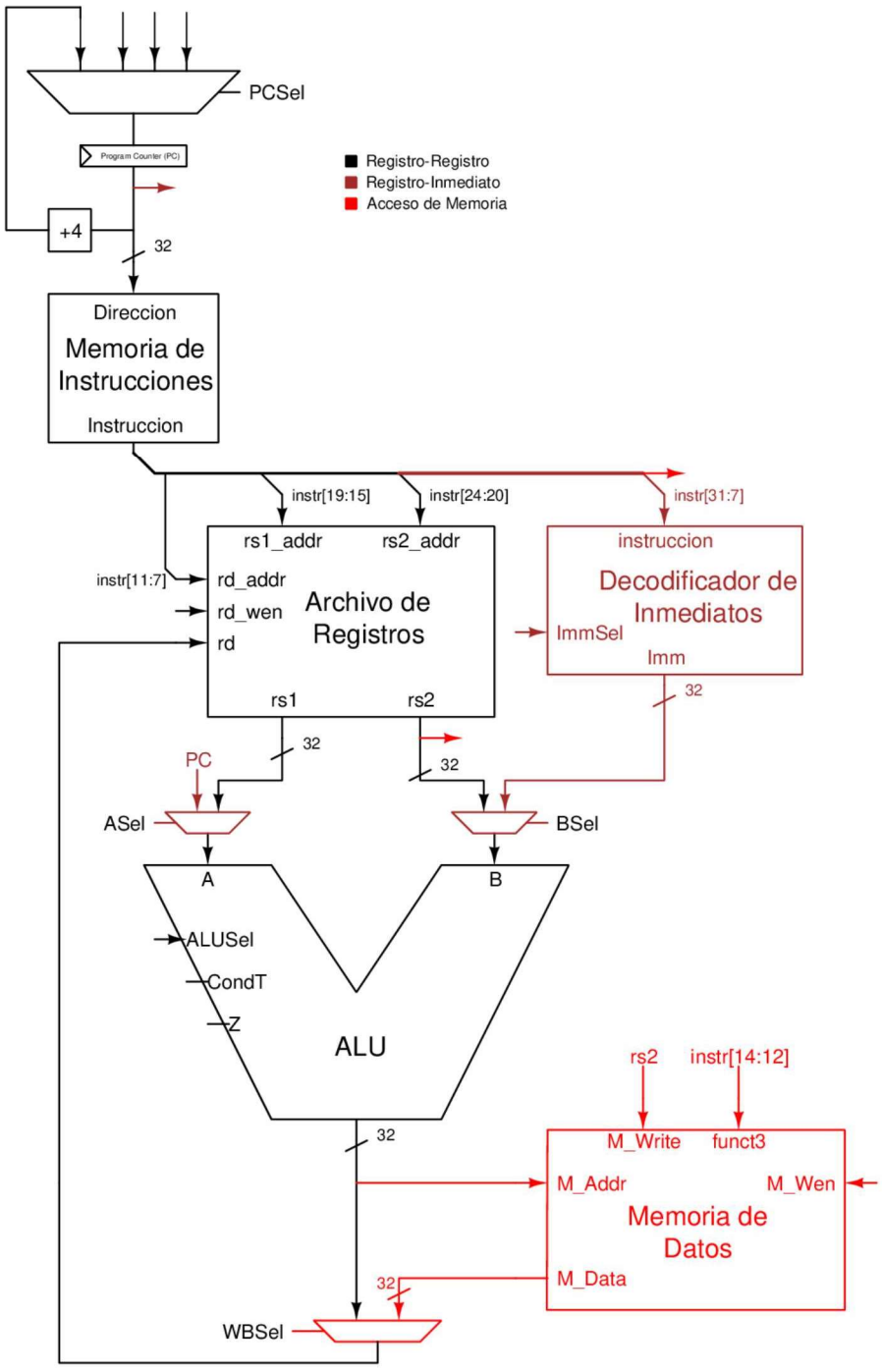
La instrucción *AUIPC* realiza la suma entre *PC* y el valor inmediato, como el valor inmediato se encuentra conectado en el multiplexor de la entrada *B* del *ALU*, la única opción viable es agregar un multiplexor en la entrada *A* del *ALU*. *ASel* seleccionará el valor de PC y *BSel* seleccionará el valor inmediato.

ALUSel escogerá la operación dependiendo de la instrucción, *rd_wen* poseerá el valor de uno para habilitar la escritura del registro *rd*, *PCSel* seleccionará *PC+4* para ser escrito en *PC*. *ImmSel* seleccionará el inmediato correspondiente a la instrucción a ejecutarse, por ejemplo, al ejecutar instrucciones de *opcode OP-IMM*, *ImmSel* escogerá el valor inmediato Tipo-I, para las instrucciones *AUIPC* y *LUI ImmSel* seleccionará el valor inmediato Tipo-U.

1.3.7.3. Ejecución instrucciones de acceso de memoria

Al *datapath* existente se agrega el módulo interfaz de memoria, adicionalmente se añade un multiplexor para seleccionar el valor a escribir en el registro *rd*. Los *bits* doce al catorce de la instrucción, *intr[14:12]*, son conectados en la entrada *funct3* del módulo interfaz de memoria.

Figura 44. ***Datapath* instrucciones de acceso de memoria**



Fuente: elaboración propia, empleando Xcircuit v3.10.

En todas las instrucciones de acceso de memoria, la dirección de la transferencia es calculada con la suma de *rs1* y el valor inmediato. Por lo tanto, *ASel* seleccionará *rs1*, *BSel* seleccionará el valor inmediato, *ALUSel* seleccionará la operación suma. El resultado de la suma es conectado a *M_addr* del módulo interfaz de memoria.

Al ejecutar instrucciones de *opcode LOAD*, el dato proveniente de memoria, *M_Data,* es escrito en el registro *rd*, por lo tanto, *WBSel* escogerá *M_Data* para ser escrito en *rd.* Adicionalmente *rd_wen* habilitará la escritura del registro *rd*. La señal *M_Wen* deshabilita la escritura de memoria.

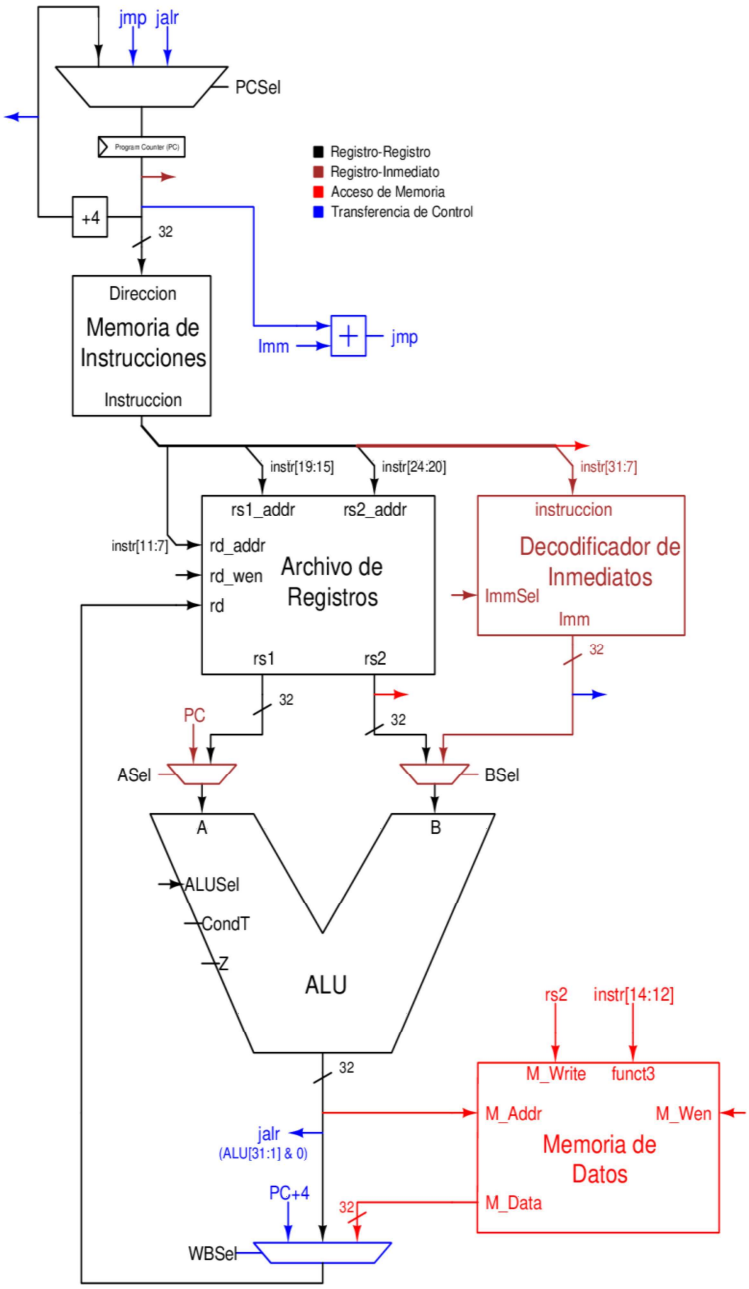
Al ejecutar instrucciones de *opcode STORE*, el dato a escribir a memoria se encuentra en el registro *rs2*, por lo tanto, *rs2* es conectado directamente en *M_Write. M_Wen* habilita la escritura de memoria, como estas instrucciones no modifican el valor de ningún registro, *rd_wen* deshabilitará la escritura de *rd*, por lo tanto, el valor de *WBSel* es irrelevante.

PCSel seleccionará *PC+4* para ser escrito en *PC*.

1.3.7.4. Ejecución instrucciones de transferencia de control

A continuación, se presenta la figura 45 la cual contiene las modificaciones realizadas en el *datapath*.

Figura 45. ***Datapath* instrucciones de transferencia de control**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Las instrucciones de salto condicional realizan una comparación entre *rs1* y *rs2*, si la comparación es verdadera, se debe modificar *PC* con la suma entre el valor inmediato y *PC*, esto presenta un problema, el *ALU* se encuentra ocupado realizando la comparación de *rs1* y *rs2*, en otras palabras, no es posible realizar la comparación y al mismo tiempo realizar la suma del inmediato y *PC*. Para solucionar esto, se agrega un sumador al *datapath*, donde sus dos entradas se encuentran conectadas a *PC* y el valor inmediato, respectivamente, el resultado de este sumador, *jmp*, es conectado con el multiplexor del contador de programa.

Si la comparación es verdadera *PCSel* seleccionará el valor de *jmp* para ser escrito en *PC*, en caso contrario se seleccionará la siguiente instrucción, *PC+4*. *ASel* escogerá *rs1*, *BSel* escogerá *rs2*, estas instrucciones no modifican el valor de ningún registro, por lo tanto, *rd_wen* deshabilitará la escritura del registro rd, *WBSel* es irrelevante.

La instrucción *BEQ* compara si *rs1* y *rs2* son iguales, *ALUSel* seleccionará la función de resta, *SUB*, si dos números son iguales su resta dará como resultado cero, por lo tanto, si *Z* posee el valor de uno, la comparación es verdadera.

La instrucción *BNE* compara si *rs1* y *rs2* son diferentes, *ALUSel* seleccionará la función de resta, *SUB*, si dos números son diferentes; su resta dará como resultado un valor distinto a cero, por lo tanto, si *Z* posee el valor de cero, la comparación es verdadera.

Las instrucciones *BLT* y *BLTU* comparan si el registro *rs1* es menor al registro *rs2*, *ALUSel* seleccionará la función *SLT* y *SLTU* respectivamente, por lo tanto, si *CondT* posee el valor de uno, la comparación es verdadera.

Las instrucciones *BGE* y *BGEU* comparan si el registro *rs1* es mayor o igual al registro *rs2*, *ALUSel* seleccionará la función *SLT* y *SLTU* respectivamente, por lo tanto, si *CondT* posee el valor de cero, la comparación es verdadera.

Las instrucciones de salto incondicional deben escribir *rd* con la dirección de la siguiente instrucción, *PC+4*, *rd_wen* habilitará la escritura del registro *rd*, y *WBSel* escogerá el valor de *PC+4*.

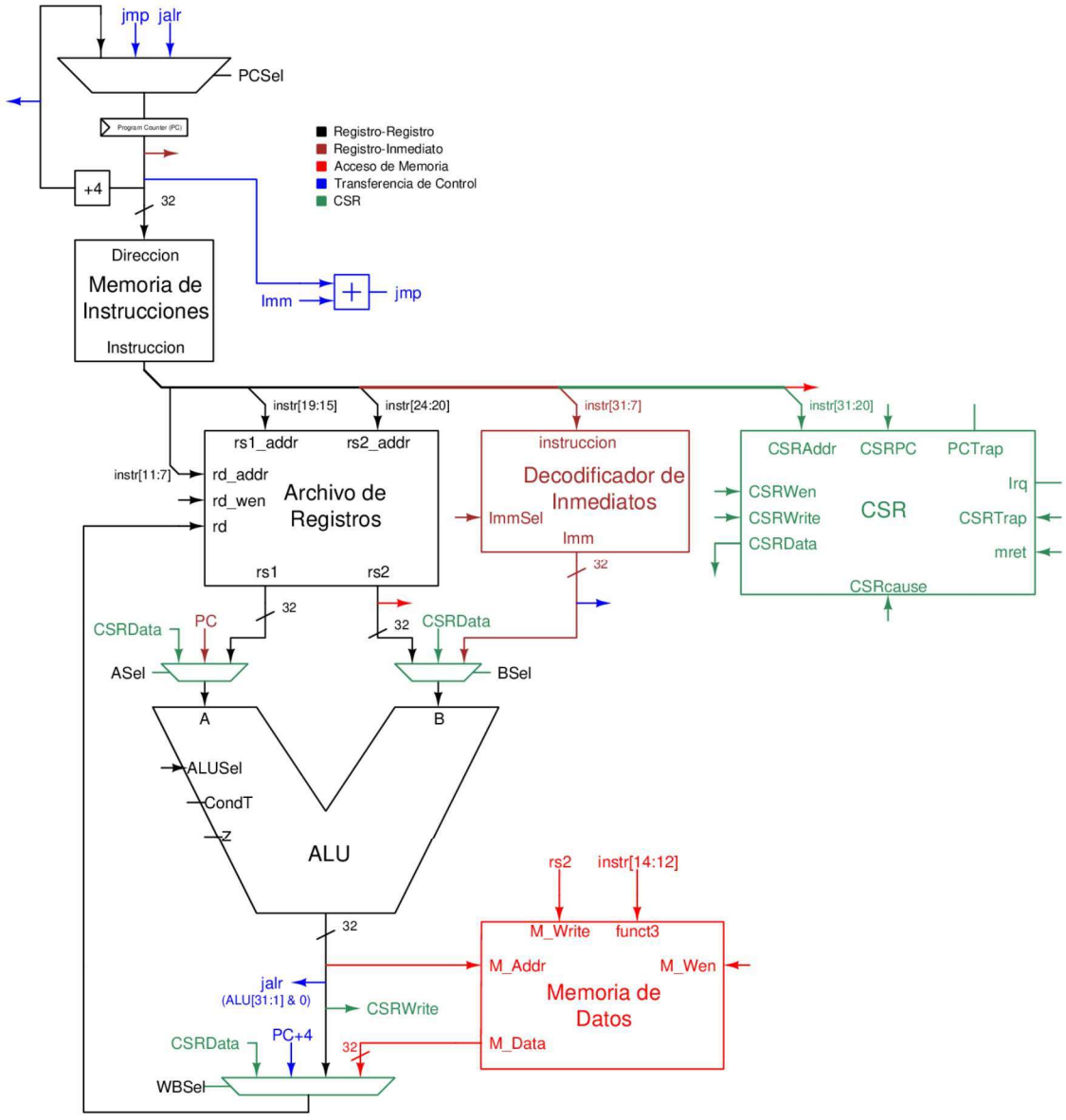
Al ejecutar la instrucción *JAL*, *PCSel* seleccionará el valor de *jmp*, los valores de *ASel*, *Bsel* y *ALUSel* son irrelevantes.

La instrucción *JALR*, realiza la suma entre *rs1* y el valor inmediato, luego establece el *bit* cero del resultado de la suma a cero, este resultado es escrito en *PC*. Para realizar esta acción *Asel* selecciona *rs1*, *Bsel* selecciona el valor inmediato, *ALUSel* escoge la operación de suma. Los *bits* uno al treinta y uno del resultado son concatenados con el valor de cero, señal *jalr*, y es conectado con el multiplexor del contador de programa. *PCSel*, seleccionará el valor *jalr* para ser escrito en *PC*.

1.3.7.5. Ejecución instrucciones *CSR*

Al *datapath* existente se agrega el módulo *CSR*, los *bits* veinte al treinta y uno de la instrucción, *intr[31:20]*, son conectados a *CSRAddr.*

Figura 46. ***Datapath* instrucciones *CSR***



Fuente: elaboración propia, empleando Xcircuit v3.10.

Una instrucción *CSR* realiza una operación entre el registro *csr* seleccionado y un segundo operando, el resultado de la operación es escrito en el registro *csr* y el valor original del registro *CSR* es escrito en *rd*. El resultado de

la operación se encuentra en la salida del *ALU*, el cual está conectado a *CSRWrite.* Para escribir el valor original del registro *CSR* en *rd*, se conecta *CSRData* en el multiplexor controlado por *WBSel*.

En todas las instrucciones *CSR*, *WBSel* seleccionará el valor de *CSRData* para ser escrito en *rd*, y *rd_wen* habilitará su escritura.

Para las instrucciones *CSR* sin inmediato, se realiza una operación entre el registro *csr* y *rs1*, por tal motivo *CSRData* es conectado al multiplexor controlado por *BSel*. La instrucción *CSRRW* únicamente intercambia valores por lo que no debe realizarse alguna operación, *ALUSel* escogerá la operación *A* que retornará el valor de *rs1* intacto para ser escrito en *csr*.

Para las instrucciones *CSR* con inmediato, se realiza una operación entre el registro *CSR* y el valor inmediato, como el valor inmediato es seleccionado por *BSel*, se debe conectar *CSRData* al multiplexor controlado por *ASel*. La instrucción *CSRRWI* únicamente intercambia valores por lo que no debe realizarse alguna operación, *ALUSel* escogerá la operación *B* que retornará el valor de inmediato intacto para ser escrito en *csr*.

Para ejecutar la instrucción *CSRRS* y *CSRRSI*, *ALUSel* seleccionará la operación *OR* y para la instrucción *CSRRC* y *CSRRCI*, *ALUSel* seleccionará la operación *clear*.

1.3.7.6. Arquitectura privilegiada

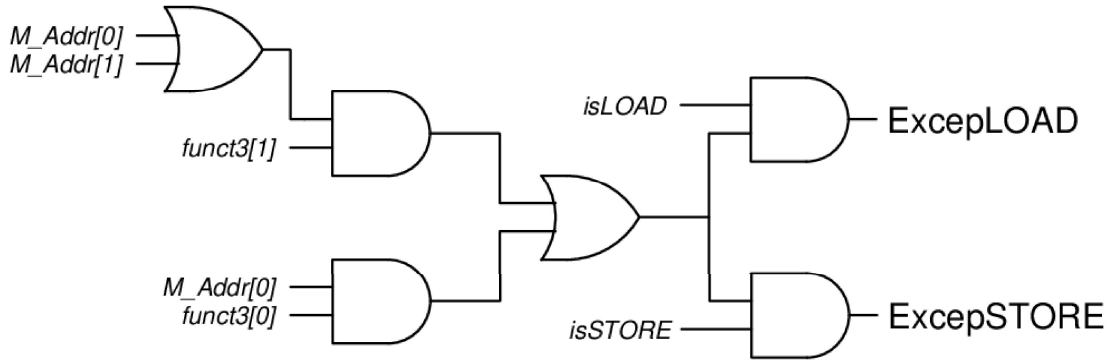
A continuación, se describe la implementación de la arquitectura privilegiada.

1.3.7.6.1. Detección de excepciones por desalineación

La forma de detectar una desalineación es comprobar si la dirección de memoria a acceder es múltiplo del tamaño de la transferencia, por ejemplo, las instrucciones *LH*, *LHU* y *SH* realizan transferencias de dos *bytes*, por lo tanto, todos los accesos de direcciones que no son múltiplos de dos, direcciones impares, son accesos desalineados.

De la misma manera las instrucciones *LW* y *SW* realizan transferencias de cuatro *bytes*. Todo intento de realizar una transferencia a direcciones que no sean múltiplo de cuatro deben generar excepciones por desalineación. Las instrucciones *LB*, *LBU*, *SB*, realizan transferencias de un *byte*, todo número entero es múltiplo de uno, por lo tanto, estas instrucciones no deben generar excepciones por desalineación.

Figura 47. **Detección de *excepción* por desalineación en instrucción *LOAD*/*STORE***



Fuente: elaboración propia, empleando Xcircuit v3.10.

En representación binaria todos los números impares en su *bit* menos significativo, *bit* cero, poseen un valor de uno. Por lo tanto, en las instrucciones *LH*, *LHU* y *SH* se puede detectar accesos desalineados verificando el *bit* cero de *M_Addr*. Los números múltiplo de cuatro, poseen un valor de cero en sus dos *bits* menos significativos. Por consiguiente, las instrucciones *LW* y *SW* detectan accesos desalineados si el *bit* cero o uno de *M_Addr* poseen el valor de uno.

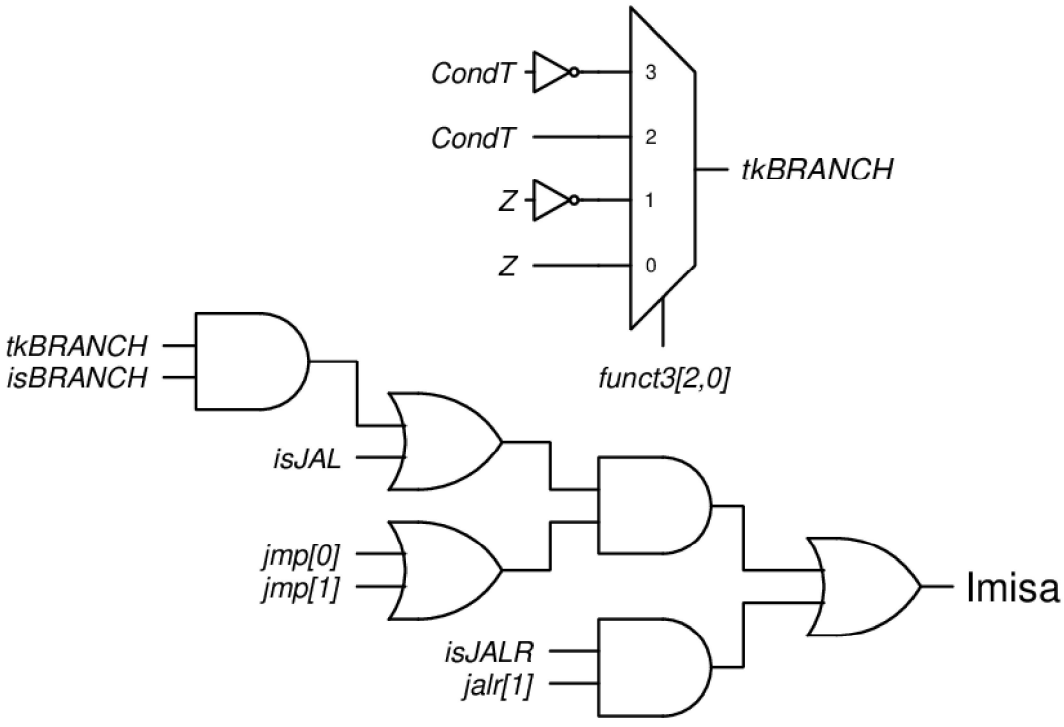
Las instrucciones de *opcode LOAD* y *STORE* utilizan *funct3* para especificar el tamaño de la transferencia y en su codificación surge un patrón, para transferencias de cuatro *bytes*, el *bit* uno de *funct3* es igual a uno, *funct3[1] = 1*. Para transferencias de dos *bytes*, el *bit* cero de *funct3* es igual a uno, *funct3[0] = 1*.

Uniendo la detección de desalineación y el tamaño de transferencia se puede detectar si la transferencia genera una excepción.

Estas excepciones únicamente son generadas al ejecutar instrucciones *LOAD* o *STORE*, para ello se utilizan las señales *isLOAD* y *isSTORE*, los cuales poseerán el valor de uno cuando el *opcode* de la instrucción sea *LOAD* y *STORE* respectivamente. De lo contrario existe la posibilidad que otras instrucciones cumplan con las condiciones necesarias y erróneamente se generé una excepción por desalineación.

Las señales resultantes *ExcepLOAD* y *ExcepSTORE*, son utilizadas para detectar la presencia de una excepción por acceso desalineado en instrucciones *LOAD* y *STORE* respectivamente.

Figura 48. **Excepción por desalineación en dirección de instrucción**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Una excepción por desalineación en dirección de instrucción es producida si la dirección objetivo-generada por instrucciones de transferencia de control no es múltiplo de cuatro. Esto es porque la lectura de una instrucción es una transferencia de memoria de cuatro *bytes*.

Para las instrucciones con *opcode JAL* y *BRANCH* la dirección objetivo está contenida en la señal *jmp*, por lo que se usan los *bits* cero y uno de *jmp* para detectar una desalineación.

Las instrucciones con *opcode BRANCH* poseen un requerimiento adicional, solo se generan excepciones si las condiciones son verdaderas y la dirección objetivo es desalineada.

Al observar *funct3* en instrucciones *BRANCH*, surge un patrón respecto a los *bits* cero y dos, *funct3[2,0]*, si *funct3[2,0]* es igual a 0b00 se ejecuta la instrucción *BEQ* la cual es verdadera si *Z* es igual a uno, si *funct3[2,0]* es igual a 0b01 se ejecuta la instrucción *BNE* la cual es verdadera si *Z* es igual a cero, si *funct3[2,0]* es igual a 0b10 se ejecuta la instrucción *BLT* y *BLTU* las cuales son verdaderas si *CondT* es igual a uno, si *funct3[2,0]* es igual a 0b11 se ejecuta la instrucción *BGE* y *BGEU* las cuales son verdaderas si *CondT* es igual a cero, *tkBRANCH* es utilizado para señalar si la condición es verdadera.

La instrucción *JALR* por otra parte utiliza el *ALU* para calcular la dirección objetivo, esta dirección se encuentra representada por la señal *jalr*, como esta instrucción automáticamente escribe el *bit* cero de *jalr* con un valor de cero, únicamente el *bit* uno es utilizado para calcular desalineación.

Estas excepciones únicamente son generadas al ejecutar instrucciones *JAL, JALR* o saltos condicionales, para ello se utilizan las señales *isJAL*, *isJALR*

y *isBRANCH*, las cuales poseerán el valor de uno cuando el *opcode* de la instrucción sea *JAL, JALR, BRANCH* respectivamente.

La señal resultante que se utiliza para detectar la presencia de una excepción por desalineación en dirección de instrucción es *Imisa*.

1.3.7.6.2. Codificación causa de excepción o interrupción.

Como lo menciona el *ISA*, al generarse una interrupción o excepción el registro *mcause* debe ser escrito con la causa de la interrupción o excepción, así mismo si más de una excepción es generada se debe reportar la causa de la excepción de mayor prioridad.

Por lo tanto, se construyó un circuito codificador de prioridad y causa, el cual cumple con la siguiente tabla de verdad:

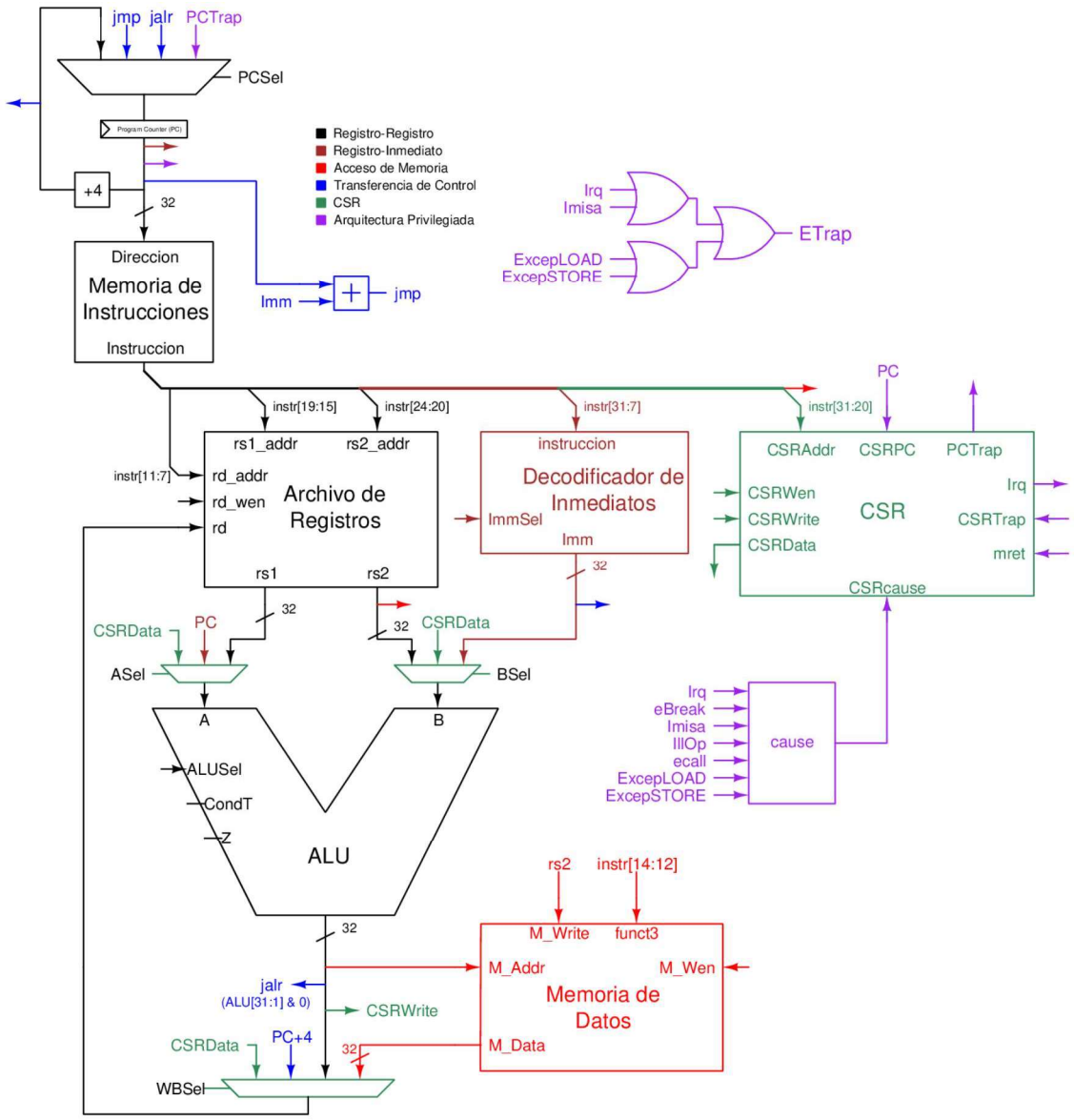
Tabla VIII. **Tabla de verdad codificador de prioridad y causa**

Irq	eBreak	Imisa	IllOp	eCall	ExcepSTORE	ExcepLOAD	cause
0	0	0	0	0	0	0	-
0	0	0	0	0	0	1	4
0	0	0	0	0	1	-	6
0	0	0	0	1	-	-	11
0	0	0	1	-	-	-	2
0	0	1	-	-	-	-	0
0	1	-	-	-	-	-	3
1	-	-	-	-	-	-	11

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Los valores “-” representan no importa. Las señales *eBreak* y *eCall* poseen el valor de uno al ejecutarse las instrucciones *EBREAK* y *ECALL* respectivamente. *IllOp* es generado cuando se trata de ejecutar una instrucción ilegal o no implementada. *cause* es la señal que poseerá la representación numérica de la causa de la excepción o interrupción.

Figura 49. ***Datapath* con arquitectura privilegiada**



Fuente: elaboración propia, empleando Xcircuit v3.10.

CSRcause es conectado con *cause*. *PCTrap* es conectado al multiplexor de contador de programa. *PC* es conectado con *CSRPC. ETrap* es conectado a la señal *Irq* y todas las señales que representan una excepción que no es

causada por una instrucción privilegiada. *ETrap* poseerá el valor de uno si cualquiera de estas señales posee el valor de uno.

Si *ETrap* posee el valor de uno, *PCSel* debe seleccionar el valor de *PCTrap*, *CSRTrap* debe ser establecerse a uno, *rd_wen*, *M_Wen* y *CSRWen* deben ser establecidos a cero para garantizar que no haya cambios en el estado del procesador al momento de la excepción o interrupción. Los registros *mstatus*, *mcause*, *mepc* serán escritos con los valores establecidos por el *ISA*. *PCTrap* tendrá el valor de *base* o *base+44* dependiendo del valor de *mtvec[0]* y *Irq* .

Si *MRet* posee el valor de uno, *PCSel* escogerá el valor de *PCTrap* para ser escrito en *PC*. El registro *mstatus* será modificado como lo define el *ISA*, *PCTrap* poseerá el valor del registro *mepc*.

1.3.8. Unidad de control

Ahora que se tiene lógica capaz de ejecutar instrucciones, se necesita un circuito que seleccione los valores correctos de todos los multiplexores introducidos y todas las señales de habilitación de escritura, dependiendo de la instrucción que se desea ejecutar, adicionalmente *Illop* debe ser establecido a uno cuando se trate de ejecutar una instrucción ilegal, por último, debe establecer las señales *MRet*, *EBreak*, *ECall* y *CSRTrap* cuando sea necesario. El componente encargado de esto es la unidad de control.

Para identificar la instrucción que se desea ejecutar se utilizan los valores de *opcode*, *funct3* y *funct7*, equivalentes a *instr[6:0]*, *instr[14:12]* y *instr[31:25]* respectivamente, adicionalmente se utilizan las señales *Z*, *CondT*, para la verificación de condiciones en instrucciones de salto condicional y cambiar *PCSel*

con el valor correspondiente. *ETrap* es utilizado para indicar la presencia de una excepción o interrupción.

Tabla IX. **Tabla de verdad unidad de control**

Instruccion	Entradas						Salidas													
	Instruccion			DataPath			DataPath													
	opcode	funct3	funct7	ETrap	Z	CondT	PCSel	ASel	BSel	ImmSel	WBSel	ALUSel	rd_wen	M_Wen	CSRWen	CSRTrap	mret	ecall	eBreak	IllOp
LUI	LUI	-	-	0	-	-	PC+4	-	imm	utype	alu	b	1	0	0	0	0	-	-	-
AUIPC	AUIPC	-	-	0	-	-	PC+4	PC	imm	utype	alu	add	1	0	0	0	0	-	-	-
JAL	JAL	-	-	0	-	-	jmp	-	-	jtype	PC+4	-	1	0	0	0	0	-	-	-
JALR	JALR	000	-	0	-	-	jalr	rs1	imm	itype	PC+4	add	1	0	0	0	0	-	-	-
BEQ	BRANCH	000	-	0	0	-	PC+4	rs1	rs2	btype	-	sub	0	0	0	0	0	-	-	-
BEQ	BRANCH	000	-	0	1	-	jmp	rs1	rs2	btype	-	sub	0	0	0	0	0	-	-	-
BNE	BRANCH	001	-	0	0	-	jmp	rs1	rs2	btype	-	sub	0	0	0	0	0	-	-	-
BNE	BRANCH	001	-	0	1	-	PC+4	rs1	rs2	btype	-	sub	0	0	0	0	0	-	-	-
BLT	BRANCH	100	-	0	-	0	PC+4	rs1	rs2	btype	-	slt	0	0	0	0	0	-	-	-
BLT	BRANCH	100	-	0	-	1	jmp	rs1	rs2	btype	-	slt	0	0	0	0	0	-	-	-
BGE	BRANCH	101	-	0	-	0	jmp	rs1	rs2	btype	-	slt	0	0	0	0	0	-	-	-
BGE	BRANCH	101	-	0	-	1	PC+4	rs1	rs2	btype	-	slt	0	0	0	0	0	-	-	-
BLTU	BRANCH	110	-	0	-	0	PC+4	rs1	rs2	btype	-	sltu	0	0	0	0	0	-	-	-
BLTU	BRANCH	110	-	0	-	1	jmp	rs1	rs2	btype	-	sltu	0	0	0	0	0	-	-	-
BGEU	BRANCH	111	-	0	-	0	jmp	rs1	rs2	btype	-	sltu	0	0	0	0	0	-	-	-
BGEU	BRANCH	111	-	0	-	1	PC+4	rs1	rs2	btype	-	sltu	0	0	0	0	0	-	-	-
LB	LOAD	000	-	0	-	-	PC+4	rs1	imm	itype	mem	add	1	0	0	0	0	-	-	-
LH	LOAD	001	-	0	-	-	PC+4	rs1	imm	itype	mem	add	1	0	0	0	0	-	-	-
LW	LOAD	010	-	0	-	-	PC+4	rs1	imm	itype	mem	add	1	0	0	0	0	-	-	-
LBU	LOAD	100	-	0	-	-	PC+4	rs1	imm	itype	mem	add	1	0	0	0	0	-	-	-
LHU	LOAD	101	-	0	-	-	PC+4	rs1	imm	itype	mem	add	1	0	0	0	0	-	-	-
SB	STORE	000	-	0	-	-	PC+4	rs1	imm	stype	-	add	0	1	0	0	0	-	-	-
SH	STORE	001	-	0	-	-	PC+4	rs1	imm	stype	-	add	0	1	0	0	0	-	-	-
SW	STORE	010	-	0	-	-	PC+4	rs1	imm	stype	-	add	0	1	0	0	0	-	-	-
ADDI	OP-IMM	000	-	0	-	-	PC+4	rs1	imm	itype	alu	add	1	0	0	0	0	-	-	-
SLTI	OP-IMM	010	-	0	-	-	PC+4	rs1	imm	itype	alu	slt	1	0	0	0	0	-	-	-
SLTIU	OP-IMM	011	-	0	-	-	PC+4	rs1	imm	itype	alu	sltu	1	0	0	0	0	-	-	-
XORI	OP-IMM	100	-	0	-	-	PC+4	rs1	imm	itype	alu	xor	1	0	0	0	0	-	-	-
ORI	OP-IMM	110	-	0	-	-	PC+4	rs1	imm	itype	alu	or	1	0	0	0	0	-	-	-
ANDI	OP-IMM	111	-	0	-	-	PC+4	rs1	imm	itype	alu	and	1	0	0	0	0	-	-	-
SLLI	OP-IMM	001	0000000	0	-	-	PC+4	rs1	imm	itype	alu	sll	1	0	0	0	0	-	-	-
SRLI	OP-IMM	101	0000000	0	-	-	PC+4	rs1	imm	itype	alu	srl	1	0	0	0	0	-	-	-
SRAI	OP-IMM	101	0100000	0	-	-	PC+4	rs1	imm	itype	alu	sra	1	0	0	0	0	-	-	-
ADD	OP	000	0000000	0	-	-	PC+4	rs1	rs2	-	alu	add	1	0	0	0	0	-	-	-
SUB	OP	000	0100000	0	-	-	PC+4	rs1	rs2	-	alu	sub	1	0	0	0	0	-	-	-
SLL	OP	001	0000000	0	-	-	PC+4	rs1	rs2	-	alu	sll	1	0	0	0	0	-	-	-
SLT	OP	010	0000000	0	-	-	PC+4	rs1	rs2	-	alu	slt	1	0	0	0	0	-	-	-

Continuación tabla IX.

SLTU	OP	011	0000000	0	-	-	PC+4	rs1	rs2	-	alu	sltu	1	0	0	0	0	-	-	-
XOR	OP	100	0000000	0	-	-	PC+4	rs1	rs2	-	alu	xor	1	0	0	0	0	-	-	-
SRL	OP	101	0000000	0	-	-	PC+4	rs1	rs2	-	alu	srl	1	0	0	0	0	-	-	-
SRA	OP	101	0100000	0	-	-	PC+4	rs1	rs2	-	alu	sra	1	0	0	0	0	-	-	-
OR	OP	110	0000000	0	-	-	PC+4	rs1	rs2	-	alu	or	1	0	0	0	0	-	-	-
AND	OP	111	0000000	0	-	-	PC+4	rs1	rs2	-	alu	and	1	0	0	0	0	-	-	-
FENCE	0001111	000	-	0	-	-	PC+4	-	-	-	-	-	0	0	0	0	0	-	-	-
ECALL	SYSTEM	000	instr(20)=0	0	-	-	PCTrap	-	-	-	-	-	0	0	0	1	0	1	0	0
EBREAK	SYSTEM	000	instr(20)=1	0	-	-	PCTrap	-	-	-	-	-	0	0	0	1	0	0	1	0
MRET	SYSTEM	000	0011000, bit(21)=1	0	-	-	PCTrap	-	-	-	-	-	0	0	0	0	1	0	0	0
SRET	SYSTEM	000	0001000, instr(21)=1	0	-	-	PCTrap	-	-	-	-	-	0	0	0	1	0	0	0	1
URET	SYSTEM	000	0000000, instr(21)=1	0	-	-	PCTrap	-	-	-	-	-	0	0	0	1	0	0	0	1
CSRRW	SYSTEM	001	-	0	-	-	PC+4	rs1	-	-	csrdata	a	1	0	1	0	0	-	-	-
CSRRS	SYSTEM	010	-	0	-	-	PC+4	rs1	csr	-	csrdata	or	1	0	1	0	0	-	-	-
CSRRC	SYSTEM	011	-	0	-	-	PC+4	rs1	csr	-	csrdata	clear	1	0	1	0	0	-	-	-
CSRRWI	SYSTEM	101	-	0	-	-	PC+4	-	imm	csrtype	csrdata	b	1	0	1	0	0	-	-	-
CSRRSI	SYSTEM	110	-	0	-	-	PC+4	csr	imm	csrtype	csrdata	or	1	0	1	0	0	-	-	-
CSRRCI	SYSTEM	111	-	0	-	-	PC+4	csr	imm	csrtype	csrdata	clear	1	0	1	0	0	-	-	-
ILEGAL	ILEGAL	-	-	0	-	-	PCTrap	-	-	-	-	-	0	0	0	1	0	0	0	1
-	-	-	-	1	-	-	PCTrap	M	M	M	-	M	0	0	0	1	0	0	0	0

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Las casillas con el valor “-” son utilizados para indicar no importa, por ejemplo, las señales *Z* y *CondT* son utilizados únicamente en instrucciones de salto condicional, para el resto de las instrucciones *Z* y *CondT* son irrelevantes, sin importar qué valor poseen no cambiarán el valor de ninguna señal saliente. De igual manera estas instrucciones no modifican el valor de ningún registro, por lo que el valor de *WBSel* es irrelevante.

ILEGAL se refiere a instrucciones no descritas en el *ISA*, causante de la excepción instrucción ilegal, adicionalmente se observa que, si *ETrap* posee el valor de uno, sin importar qué instrucción se ejecute, *PCSel* debe seleccionar *PCTrap* y se debe deshabilitar todo tipo de escritura, con el propósito de no alterar el estado del procesador al momento de ser interrumpido, adicionalmente se debe establecer el valor de *CSRTrap* a uno.

En las casillas con el valor “M” se debe de mantener el valor correspondiente a la instrucción interrumpida, por ejemplo si *ETrap* es activado por una excepción de desalineación en una instrucción *LOAD*, *ALUSel*, *ImmSel*, *ASel* y *BSel* deben mantener su valor, ya que cambiar alguno de estos valores cambian las condiciones necesarias para detectar la excepción, por consiguiente *ETrap* cambiará su valor a cero, restableciendo los valores originales de *ALUSel*, *ImmSel*, *ASel* y *BSel*, lo que causará nuevamente que *ETrap* posea el valor de uno, en otras palabras de no mantener su valor, se genera una inestabilidad en el procesador.

La unidad de control será el circuito que describe la tabla de verdad, la implementación más sencilla es la utilización de una memoria de solo lectura, *ROM*. Donde las entradas del circuito son utilizadas como dirección de memoria y las salidas son los valores escritos en el *ROM*.

Tabla X. **Representación numérica de constantes utilizadas en *PCSel, ASel, BSel* y *WBSel***

PCSel	
Constante	Valor binario
PC+4	00
jmp	01
jalr	10
PCTrap	11

WBSel	
Constante	Valor binario
alu	00
mem	01
PC+4	10
csrdata	11

ASel	
Constante	Valor binario
rs1	00
PC	01
csr	10

BSel	
Constante	Valor binario
rs2	00
imm	01
csr	10

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Tabla XI. **Representación numérica de constantes utilizadas en *ALUSel* y *ImmSel***

ALUSel	
Constante	Valor Binario
add	0000
sub	1000
sll	0001
slt	0010
sltu	0011
xor	0100
srl	0101
sra	1101
or	0110
and	0111
a	1100
b	1110
clear	1111

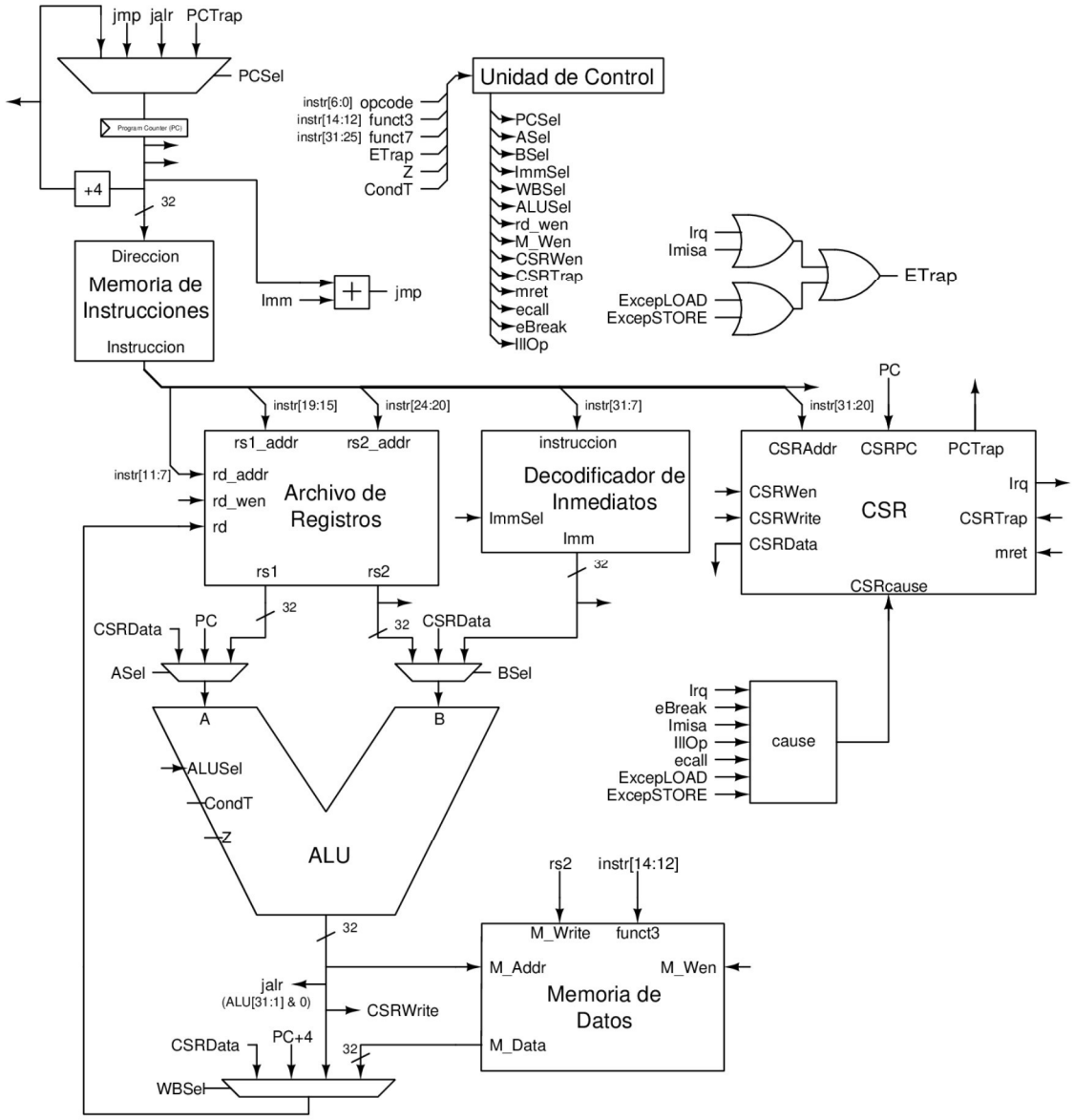
ImmSel	
Constante	Valor Binario
itype	000
stype	001
btype	010
utype	011
jtype	100
csrtype	101

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

1.3.9. Micro-arquitectura procesador RV32I

A continuación, se presenta la figura 50, en cual se expone la micro-arquitectura completa del procesador:

Figura 50. **Micro-arquitectura procesador RV32I**



Fuente: elaboración propia, empleando Xcircuit v3.10.

El código fuente en VHDL de todos los componentes y la micro-arquitectura[5].

1.4. Comprobación y validación

A continuación, se presenta el método de comprobación de componentes y comprobación del correcto funcionamiento del procesador.

1.4.1. Comprobación de componentes

VHDL adicional a describir *hardware*, posee ciertas funciones similares a lenguajes de programación de propósito general, las cuales habilitan la posibilidad de escribir código con el objetivo de simular el comportamiento de un componente.

Una metodología para comprobar y validar el comportamiento deseado de un componente es escribir código para simular su funcionamiento, donde se estimulan las entradas del componente con valores específicos, posteriormente utilizando un software capaz de simular VHDL, observar gráficos de forma de onda de todas las señales internas y externas del componente, y de forma visual verificar que las señales tomen los valores esperados.

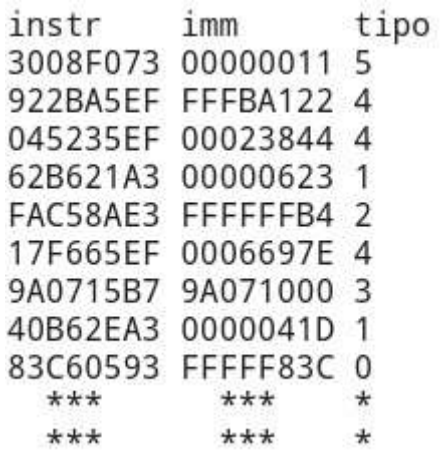
Utilizar esta metodología presenta varias desventajas, dependiendo del componente puede consumir una cantidad considerable de tiempo, en sistemas complejos con muchas señales resulta complicado y tedioso el análisis, además esta metodología puede poseer bajos niveles de portabilidad, entre otros, pero la

[5] SIERRA, Ottoniel. *RV32I/hardware/unpipelined.* https://github.com/oasm95/RV32I/tree/main/hardware-/unpipelined. Consulta: agosto 2021.

desventaja más importante es la dependencia de observaciones humanas, lo que reduce de gran manera la fiabilidad de dichas comprobaciones.

La metodología que se usará es la de utilizar vectores de prueba, esta metodología consiste en leer archivos que contienen valores utilizados para estimular las entradas del componente y valores que contienen el resultado esperado. Estos archivos son generados por programas que recrean el comportamiento deseado del componente, escritos en un lenguaje de programación que facilita la generación de valores de estímulo y la recreación del comportamiento de un componente.

Figura 51. **Ejemplo archivo vector de prueba**



Fuente: elaboración propia, empleando Python 3.

Como ejemplo se utiliza una porción del archivo vector de prueba del componente decodificador de inmediatos, la primera columna contiene la instrucción que contiene el valor a decodificar, la segunda columna contiene el valor inmediato decodificado, este es el valor que se utilizará para comprobar el correcto funcionamiento del componente, la tercera columna indica el tipo de

inmediato a utilizar. Todos los valores están escritos en representación hexadecimal.

El código en VHDL consiste en leer una línea del archivo vector de prueba, asignar los valores a las entradas correspondientes, comprobar si el valor generado por el componente es igual al valor esperado en el vector de prueba, de lo contrario mostrar una advertencia que contenga la señal y la línea donde los valores discrepan. Si al finalizar la simulación no se generó alguna advertencia se puede validar el correcto funcionamiento del componente.

Figura 52. **Código de lectura y comprobación**

```
Stimulous_fromfile:process is
    file text_file : text open read_mode is "testvectorImmDecoder.txt";
    variable reportval: boolean:= false;
    variable text_line : line;
    variable linecnt: integer :=0;
    variable data: std_logic_vector(BUS_Width-1 downto 0);
    variable sel: std_logic_vector(3 downto 0);
begin
    --ciclo utilizado para leer linea por linea la totalidad del archivo
    while not endfile(text_file) loop
        readline(text_file, text_line);

        linecnt := linecnt + 1;
        --asignacion de valores de estimulo a entradas de componente
        readdata(text_line,data,linecnt,reportval);
        instr <= data(31 downto 7);
        readdata(text_line,data,linecnt,reportval);
        fileimm <= data;-- valor esperado
        readdata(text_line,sel,linecnt,reportval);
        immsel <= sel;

        wait for Tclk/4;
        --Comprobacion valor generado por componente y valor esperado en archivo
        assert (imm = fileimm)
            report "'inmediato' no coincide en linea: " & integer'image(linecnt)
            severity warning;

        wait for 3*Tclk/4;
    end loop;
    --Se ha leido el archivo en su totalidad se finaliza la simuacion
    assert false report "Simulacion Finalizada" severity failure;
end process;
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Para diferentes componentes el código debe cambiar en la cantidad de señales a asignar y la cantidad de señales a comprobar. De igual manera a los archivos con el vector de prueba se agrega o reduce la cantidad de columnas dependiendo la cantidad de señales de entrada y salida que posea el componente.

El código para la generación de los archivos vectores de prueba y el código para la lectura y comprobación de todos los componentes[6].

A pesar de que el código fue escrito con la intención de poder ser ejecutado por cualquier software con la capacidad de simular VHDL, se utilizó el programa GHDL por su característica de generar archivos ejecutables a partir de código VHDL, al contrario de interpretar el código como la mayoría de los simuladores, con el objetivo de reducir el tiempo que toma realizar una simulación, GHDL[7].

1.4.2. Comprobación de procesador

Para comprobar el correcto funcionamiento del procesador se utilizaron programas de computadora auto-verificables escritos por RISC-V *Fundation*. Los programas fueron obtenidos en RISC-V [8].

Para poder simular la ejecución de dichos programas primero se debe convertir el código escrito en el lenguaje C en instrucciones que pueda ejecutar el procesador para ello se utiliza la herramienta riscv64-elf-gcc, el cual convierte programas escritos en C en archivos ejecutables *elf*.

---

[6] SIERRA, Ottoniel. *RV32I/hardware/unpipelined*. https://github.com/oasm95/RV32I/tree/main/hardware/unpipelined. Consulta: agosto 2021.

[7] GHDL. *1.0 dev documentation*. https://ghdl.github.io/ghdl/about.html. Consulta: Julio 2020.

[8] RISC-V. *riscv-tests*. https://github.com/riscv/riscv-tests. Consulta: octubre 2020.

Como se observa en el *datapath* el procesador posee dos memorias una para instrucciones y otra memoria para datos. Afortunadamente los archivos *elf* pueden ser separados en secciones, .*text*, .*data* y .*bss*, dónde .*text* representa la sección que contiene el código ejecutable, .*data* posee los datos y variables del programa y .*bss* es una sección utilizada para variables inicializadas con el valor de cero.

Por lo tanto el contenido de la sección .*text* del archivo *elf* será escrito en la memoria de instrucciones, el contenido de .*data* y .*bss* será escrito en la memoria de datos. Para ello se utiliza la herramienta riscv64-elf-objcopy que puede ser utilizada para extraer el contenido de dichas secciones. Pero se presenta un problema debido a que .*bss* es una sección llena de ceros, los archivos *elf* realizan la optimización de únicamente indicar el tamaño de .*bss* y no ocupar ese espacio, con el objetivo de reducir el tamaño del archivo. Por lo tanto, se utiliza la herramienta riscv64-elf-size para obtener el tamaño en *bytes* de la sección .*bss*.

Por último se utiliza un programa escrito en Python el cual utiliza el tamaño de .*bss* en conjunto a los archivos binarios de .*text* y .*data* y los convierte en un formato similar a los vectores de prueba para ser leído por la simulación en VHDL.

Figura 53. **Extracción y conversión de instrucciones y datos**

```
#Compilar codigo con objetivo un procesador rv32i
riscv64-elf-gcc -march=rv32i -mabi=ilp32 -Os -Wl,-Ttext=0x0 -Bstatic -o $1.elf init.s intr.c $1.c
#Separacion de codigo ejecutable y datos
riscv64-elf-objcopy  --dump-section .text=$1.text $1.elf
riscv64-elf-objcopy -S -O binary -R .text* -R .comment* -R .riscv* -g --gap-fill 0 $1.elf $1.data
#Obtencion tamaño de seccion .bss
BSSSIZE=$(riscv64-elf-size -G $1.elf | tail -n 1 | awk -F ' ' '{print $3}')
#Combierte el codigo ejecutable y datos
#a un archivo que puede leer la simulacion en VHDL
python dumpsoftware.py $1 $BSSSIZE
#Mueve archivos resultantes a carpeta de la  simulacion
mv programData.txt ../unpipelined/
mv programText.txt ../unpipelined/
```

Fuente: elaboración propia, empleando Geany 1.37.1.

El símbolo $1 es utilizado para indicar el nombre del programa que se ejecutara, adicionalmente a el programa a ejecutar se agrega un archivo, *init.s*, el cual contiene instrucciones para inicializar el procesador antes de ejecutar el programa, además contiene instrucciones utilizadas para escribir la dirección de memoria que debe poseer el registro *CSR mvect* el cual será utilizado al momento de activarse una interrupción o excepción, por último posee instrucciones utilizadas para el manejo de interrupciones y excepciones.

Figura 54. **Código de inicialización, *init.s***

```
    .section .text.startup
    .align  4
    .globl  _startup
    .type   _startup, @function
_startup:
#Desabilita Interrupciones
    csrwi   mie, 0
    csrwi   mstatus, 0
#Guarda la direccion del codigo que maneja
#interrupciones/excepciones en el registro CSR mtvec
    la      t0 ,_trap
    csrw    mtvec, t0
#Indica el inicio del Stack
    li      sp, 0x80000
    mv      s0, sp
#Salta al inicio del programa
    j    _start
    .size   _startup, .-_startup

_trap:
    j       _trapEntry

#Si mtvec[0] = 1
#PC tomara el valor de _trapintr
#en caso de una interrupciones externa
    .= _trap+0x2c
_trapintr:
    j       _trapEntry
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 55. **Código controlador de interrupción/excepción**

```
_trapEntry:
#Se guardan todos los registros
#para conservar el estado del programa
#interrumpido.
    addi    sp, sp,-132
    sw      x1, 1*4(sp)
    sw      x2, 2*4(sp)
    ...
    sw      x30, 30*4(sp)
    sw      x31, 31*4(sp)
#Guarda el valor del registro mcause y mepc
#para usar como argumento en la funcion
# _Ehandler
    csrr    a0, mcause
    csrr    a1, mepc
    mv      a2,sp
#Salta a la funcion _Ehandler
    jal _Ehandler
#Escribe el valor retornado por _Ehandler
#en el registro CSR mepc
    csrw    mepc,a0

#Se restauran todos los registros
#para restablecer el procesor con
#el estado del programa antes
#de la Interrupcion/Excepcion
    lw      x1, 1*4(sp)
    lw      x3, 3*4(sp)
    ...
    lw      x31, 31*4(sp)
    lw      x2, 2*4(sp)

    addi    sp, sp,132
#Se restablece PC a la direccion de la
#instruccion que fue interrumpida
    mret
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Se logra observar nombres de registros no descritos en el *ISA* por ejemplo *t0* o *sp*, en realidad son sobrenombres para los registros *x5* y *x2*. A continuación, se presenta un listado de los sobrenombres de registros utilizados por los programadores de procesadores RISC-V.

Tabla XII. **Sobrenombre de registros**

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 137.

De igual manera se ven instrucciones no descritas en el *ISA* como *mv* o *li*, estas instrucciones son llamadas pseudoinstrucciones, en realidad "*mv rd, rs"* es la instrucción "*addi rd, rs, 0"* y es utilizado para copiar el contenido del registro *rs* al registro *rd*, de igual manera "*li rd, num"* es la ejecución de las instrucciones "*lui rd, num[31:12]; addi rd, rd, num[11:0]"* utilizado para escribir valores inmediatos de treinta y dos *bits* en el registro *rd*.

Figura 56. **Pseudoinstrucciones**

`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`jalr rs`	`jalr x1, 0(rs)`	Jump and link register
`ret`	`jalr x0, 0(x1)`	Return from subroutine
`call offset`	`auipc x1, offset[31:12] + offset[11]` `jalr x1, offset[11:0](x1)`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12] + offset[11]` `jalr x0, offset[11:0](x6)`	Tail call far-away subroutine
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate

Fuente: RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. p. 139.

Con el objetivo de no utilizar un método visual para verificar el funcionamiento del procesador, se modificaron los programas de comprobación agregando la función *printf*, que en general es utilizada para desplegar texto en consola, en varias partes del código para desplegar el progreso y resultado de la verificación.

Para desplegar texto en consola *printf* hace un llamado a entorno indicando que se desea escribir contenido a un archivo llamado *Standard Output* o *stdout*, en el caso de RISC-V para realizar un llamado a entorno se debe ejecutar la instrucción *ecall* y utilizar los registros *x10* a *x15* para indicar el archivo y contenido a escribir. Estas llamadas a entorno poseen el nombre de *Syscalls*, existen varios tipos de *Syscalls* para realizar diferentes funciones, en este caso en particular el *Syscall* solicitado es *Write* utilizado para escribir archivos. Para diferenciar los diferentes tipos de *Syscalls* se utiliza el registro *x17*.

Se sabe que la instrucción *ECALL* generará una excepción con causa "llamada de entorno", por lo tanto, al momento de generarse una excepción por la instrucción *ECALL* se debe de utilizar el registro *x17* para identificar si el *Syscall* solicitado es *Write*.

Con este conocimiento, la implementación de *Write* escribirá a una dirección de memoria específica con el contenido que se desea escribir, el código del simulador en VHDL puede detectar que se desea escribir a memoria utilizando la señal *M_Wen*, adicionalmente si *M_addr* coincide con dicha dirección de memoria, se guardará el contenido en un archivo de texto. Al finalizar la simulación puede leerse este archivo de texto y observar el resultado de la ejecución del programa.

El código que realiza el manejo de excepciones, *Syscalls* y comunicación con el simulador se encuentra en el archivo *intr*.c, el cual es incluido al generarse el archivo *elf*.

Figura 57. **Código manejo de excepciones, *intr.c***

```
unsigned int _Ehandler(unsigned int mcause,unsigned int mepc, int regs[32])
{
    if (mcause & 0x80000000) //Si fue una interrupcion
    {
        intrhandler();
    }
    else if (mcause == 0xB) // Si fue la instruccion ecall
    {
        //Argumentos syscall a0-a5,a7 registros x10-x15,x7
        regs[10] = syscall(regs[10],regs[11],regs[12],regs[13],regs[14],\
                regs[15],regs[17]);
        //syscall completado retornar a la instruccion despues de ecall
        mepc +=4;
    }
    else //Fue otro tipo de Excepcion
    {
        //Desplegar el tipo de Excepcion y enciclar programa
        volatile int *ErrorAddr = (int*)0xDEADBEEC;
        *ErrorAddr = mcause;
        for(;;);
    }
    // si fue una interrupción se debe ejecutar nuevamente
    // la instruccion interrumpida por lo tanto mepc no cambia
    return mepc;
}
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 58. **Código manejo *Syscalls*, *intr.c***

```
static int sys_write(int fd, const char* buf, int size)
{
    volatile char *printAddr = (char*)0x7000BEEF;
    if (fd == 1) // Si se desea escribir a consola
    {
        //Escribir todos los datos a consola
        for (int i=0; i<size;i++)
            *printAddr = buf[i];
        return size;
    }
    return -1;
}

static int syscall(int a0,int a1,int a2,int a3,int a4,int a5,int syscalltype)
{
    int res = 0;
    if(syscalltype == 64)// Si fue sys_write
    {
        res = sys_write(a0,(const char*)a1,a2);
    }
    else //Si fue otro tipo de sys_call
    {
        //Desplegar tipo de Syscall
        volatile int* othersyscall = (int*) 0x7001BEEC;
        *othersyscall = syscalltype;
    }

    return res;
}
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 59. **Código simulación, comunicación con programa**

```
outputfromRISCV:process(clk)
--archivo con el resultado
    file printoutput: text open write_mode is "stdout.txt";
    variable txt : line;
    variable int: integer;
    variable chr: character;
begin
    if rising_edge(clk) then
        if dmemwen = '1' then
        --Dependiendo la direccion realizar diferente accion
            case dmem_addr is
                when PrintAddr => --Escribe a archivo
                    int := to_integer(unsigned(dmem_data_s(31 downto 24)));
                    chr:=character'val(int);
                    --escribe contenido recibido en archivo
                    if chr = lf then
                        writeline(printoutput,txt);
                    else
                        write(txt,chr);
                    end if;
                when ErrorAddr => -- Despliega Causa de Excepsion
                    int := to_integer(unsigned(dmem_data_s));
                    report "Exception Cause: " & integer'image(int);
                when OtherSyscall => -- Despliega tipo de Syscall Solicitado
                    int := to_integer(unsigned(dmem_data_s));
                    report  "SYSCALL: " & integer'image(int);
                when others => NULL;
            end case;
        end if;
    end if;
end process;
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 60. **Código simulación, inicialización memoria de instrucción y datos**

```
--*****Memoria de Instrucciones****
--declaracion tipo memoria de instrucciones
type Imem_type is array (0 to Imemdepth-1) of std_logic_vector(31 downto 0);

--lectura de archivo y llenado de memoria
impure function fillImem return Imem_type is
    file text_file : text open read_mode is "programText.txt";
    variable text_line: line;
    variable data: Imem_type;
begin
    for i in Imem_type'range loop
        readline(text_file, text_line);
        readdata(text_line,data(i));
    end loop;
    return data;
end;
--Inicializacion de datos
constant Imem: Imem_type := fillImem;

--*****Memoria de Datos****
--declaracion tipomemoria de datos
type Dmem_type is array (0 to Dmemdepth-1) of std_logic_vector(BUS_Width-1 downto 0);

--lectura de archivo y llenado de memoria
impure function fillDmem return Dmem_type is
    file text_file : text open read_mode is "programData.txt";
    variable text_line: line;
    variable data: Dmem_type;
begin
    for i in Dmem_type'range loop
        readline(text_file, text_line);
        readdata(text_line,data(i));
    end loop;
    return data;
end;
--Inicializacion de datos
signal Dmem: Dmem_type := fillDmem;
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Con el entorno de ejecución descrito, se ejecutaron diez diferentes programas:

- *dhrystone*, es un “programa de prueba utilizado para medir el desempeño general de un procesador, desarrollado originalmente en 1984”[9].

[9] WEICKER, Reinhold. *The Dhrystone Benchmark.* https://www.keil.com/benchmarks/dhrystone.asp. Consulta: agosto de 2020.

- *median*, realiza un filtro de mediana en un set de 750 datos enteros.

- *mt-matmul*, realiza la multiplicación matricial entre dos matrices de tamaño 16x16, llenas de enteros.

- *mt-vvadd*, realiza 1 000 sumas entre datos tipo *double*.

- *multiply*, realiza 100 multiplicaciones entre enteros.

- *qsort* y *rsort,* ordena en forma ascendente un set de 2 048 datos enteros utilizando dos algoritmos diferentes.

- *spmv,* realiza multiplicación dispersa matriz-vector de datos tipo *double*.

- *towers*, realiza la resolución del rompecabezas de las torres de Hanoi con siete discos.

- *exceptions*, un programa para verificar el correcto comportamiento de excepciones e interrupciones, se utilizó el programa *dhrystone* como base y se le agregaron instrucciones causantes de excepciones.

A pesar de que el procesador no posee instrucciones que realicen multiplicación y tampoco está diseñado para ejecutar instrucciones con datos tipo *double*, en el caso de RISC-V, *double* son datos de ocho *bytes* utilizados para representar números con exponente variable y decimales, el programa gcc es capaz de convertir estas operaciones en algoritmos que contienen únicamente instrucciones descritas en el *ISA* base, por esta razón se dice que un *ISA* base es capaz de realizar computación por sí misma.

En el programa *exceptions* para el manejo de interrupciones se crearon funciones que habilitan o deshabilitan interrupciones modificando los registros *CSR mstatus* y *mie*, modifican el valor de *mtvec* y se comunican con el simulador con el objetivo de que activar o desactivar la señal *IRQ* responsable de causar interrupciones. Estas funciones se encuentran en el archivo *intr.c*.

Figura 61. **Código generación y manejo de interrupciones**

```
void _setvectored()
{
    //modifica mtvec(0)=1 por lo que las interrupciones
    //cambiaran el valor de: PC = mtvec.base + 4*cause
    //en caso se interrupciones el programa saltara a
    //_trapintr de archivo init.s
    asm("csrsi\tmtvec,1");
}
void _enableintr()
{
    //habilita interrupciones escribiendo 1 en
    //mie.MEI
    //mstatus.MIE
    volatile int mie = 0x800;
    asm volatile ("csrw mie,%0\n\t"
        "csrwi mstatus,8"
        : "=r" (mie));
}
void _disableintr()
{
    //deshabilita interrupciones escribiendo 0 en
    //mie.MEI
    //mstatus.MIE
    asm("csrwi mie,0\n\t"
        "csrci mstatus,8");
}
void interruptCPU()
{
    volatile int *SetIRQ = (int *) 0x90000004;
    *SetIRQ = 1;
}
void intrhandler()
{
    volatile int *disableIRQ = (int *) 0x90000000;
    *disableIRQ = 1;
    int pendingIRQ;
    do{
        asm volatile ("csrr %0,mip"
            : "=r" (pendingIRQ));
    }while(pendingIRQ != 0);
}
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 62. **Código controlador señal *IRQ***

```
IRQcontrol:process(clk)
begin
    if rising_edge(clk) then
        if dmemwen = '1' then
            case dmem_addr is
                when DisableIRQ =>
                    irq <= '0';
                    report "Se detecto y manejo Interrupcion";
                when SetIRQ =>
                    irq <= '1';
                    report "Se genero Interrupcion";
                when others => NULL;
            end case;
        end if;
    end if;
end process;
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Con el fin de validar la detección y manejo de excepciones e interrupciones el programa se modificó, en partes arbitrarias, para ejecutar las siguientes acciones:

- Habilitar, generar y manejar una interrupción.
- Ejecutar instrucción *LH* con desalineación.
- Ejecutar instrucción *LW* con desalineación.
- Modificar *mtvect*, generar y manejar una interrupción.
- Ejecutar instrucción *SH* con desalineación.
- Ejecutar instrucción *SW* con desalineación.
- Ejecutar instrucción *EBREAK*.

- Ejecutar instrucción *SRET*, una instrucción ilegal.

- Deshabilitar y generar una interrupción, el procesador no debe de detectarla por lo tanto no debe ser manejada.

- Ejecutar instrucción *JAL* con desalineación.

- Ejecutar instrucción *JALR* con desalineación.

- Ejecutar instrucción de salto condicional con desalineación y condición verdadera.

- Ejecutar instrucción de salto condicional con desalineación y condición falsa, de acuerdo con el *ISA* no debe generar excepción.

Al momento de ejecutar el programa el simulador desplegó en consola la siguiente información.

Figura 63. **Información desplegada en consola del simulador al ejecutar programa *exceptions***

```
tb_RISCV32I.vhdl:196:25:@107756fs:(report note): Se genero Interrupcion
tb_RISCV32I.vhdl:193:25:@108116fs:(report note): Se detecto y manejo Interrupcion
tb_RISCV32I.vhdl:173:25:@249276fs:(report note): Exception Cause: 4
tb_RISCV32I.vhdl:173:25:@409636fs:(report note): Exception Cause: 4
tb_RISCV32I.vhdl:196:25:@428900fs:(report note): Se genero Interrupcion
tb_RISCV32I.vhdl:193:25:@429260fs:(report note): Se detecto y manejo Interrupcion
tb_RISCV32I.vhdl:173:25:@570588fs:(report note): Exception Cause: 6
tb_RISCV32I.vhdl:173:25:@730980fs:(report note): Exception Cause: 6
tb_RISCV32I.vhdl:173:25:@891956fs:(report note): Exception Cause: 3
tb_RISCV32I.vhdl:173:25:@1052380fs:(report note): Exception Cause: 2
tb_RISCV32I.vhdl:196:25:@1054620fs:(report note): Se genero Interrupcion
tb_RISCV32I.vhdl:173:25:@1212948fs:(report note): Exception Cause: 0
tb_RISCV32I.vhdl:173:25:@1373972fs:(report note): Exception Cause: 0
tb_RISCV32I.vhdl:173:25:@1534508fs:(report note): Exception Cause: 0
```

Fuente: elaboración propia, empleando GHDL 0.37.

Para validar el manejo de excepciones e interrupciones estas no deben de alterar el funcionamiento del programa ejecutado.

Figura 64. **Resultado de ejecución de programa *exceptions***

```
Dhrystone Benchmark, Version C, Version 2.2
Program compiled without 'register' attribute

Trying 500 runs through Dhrystone:
Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    510
        should be:   510
Ptr_Glob->
  Ptr_Comp:          78696
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          78696
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING
```

Fuente: elaboración propia, empleando GHDL 0.37.

Cómo se logra observar el resultado del programa no fue alterado, asimismo se detectaron y manejaron las excepciones e interrupciones de forma correcta, por lo que se valida su correcto funcionamiento.

El código fuente de *init.s, intr.c* y el simulador en VHDL,[10].

El código fuente de cada programa ejecutado, así mismo con su respectivo resultado,[11].

1.5. Métricas de rendimiento

Para medir el rendimiento de un procesador se mide el tiempo que tarda en ejecutar un programa. De este concepto se pueden definir tres métricas de rendimiento, el tamaño del programa medido en cantidad de instrucciones, la cantidad de ciclos necesarios para ejecutar una instrucción y, por último, el tiempo que dura un ciclo,[12].

De los tres términos, el tamaño del programa no depende de la micro-arquitectura del procesador, sino del *ISA*, por lo tanto, no será evaluado ni tomado en cuenta como métrica de rendimiento.

El procesador ejecuta una instrucción por ciclo, es decir posee un *CPI*, Ciclos por Instrucción, de uno.

El tiempo mínimo por ciclo, o frecuencia máxima de reloj, depende de la micro-arquitectura del procesador y de la tecnología en la que será implementado, por ejemplo, un circuito implementado en dos modelos diferentes de *FPGA* obtendrán diferentes frecuencias máximas de operación, aunque la micro-arquitectura sea la misma. Así mismo la frecuencia máxima variará si un

[10] SIERRA, Ottoniel. *RV32/software.* https://github.com/oasm95/RV32I/tree/main/software. Consulta: agosto 2021.

[11] SIERRA, Ottoniel. *RV32I/software/benchmarks*. https://github.com/oasm95/RV32I/tree/main/software/benchmarks. Consulta: agosto 2021.

[12] TERMAN, Chris. *6.004 Computation Structures*. https://computationstructures.org/. Consulta: junio 2020.

circuito es implementado en una *FPGA* o un circuito integrado construido con *CMOS*.

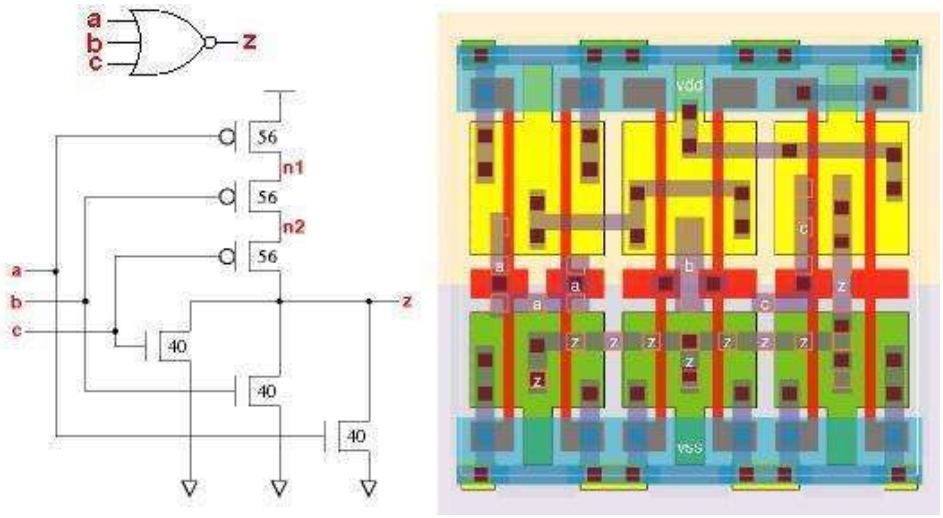
La tecnología de implementación que se utilizará como referencia en este trabajo es de circuitos integrados construidos con *CMOS*, se utiliza Qflow para obtener resultados respecto a los recursos utilizados en el diseño del procesador y su máxima frecuencia de operación.

"Qflow es un conjunto de herramientas utilizadas en el flujo de síntesis digital, para la conversión de circuitos digitales escritos en lenguajes de descripción de *hardware* como VHDL y Verilog a un circuito físico caracterizado por una librería de células estándar"[13].

Una librería de células estándar es un conjunto de abstracciones utilizadas para representar funciones lógicas, una librería difiere de otra en la implementación a nivel de *CMOS* de las funciones lógicas, por consiguiente, cada librería posee propiedades y características físicas particulares. Por ejemplo, se presenta la función lógica *NOR* de tres entradas de las librerías *rgalib01*3 y *vgalib013*, a pesar de implementar la misma función lógica, a nivel de *CMOS*, dichas funciones son construidas de manera diferente.

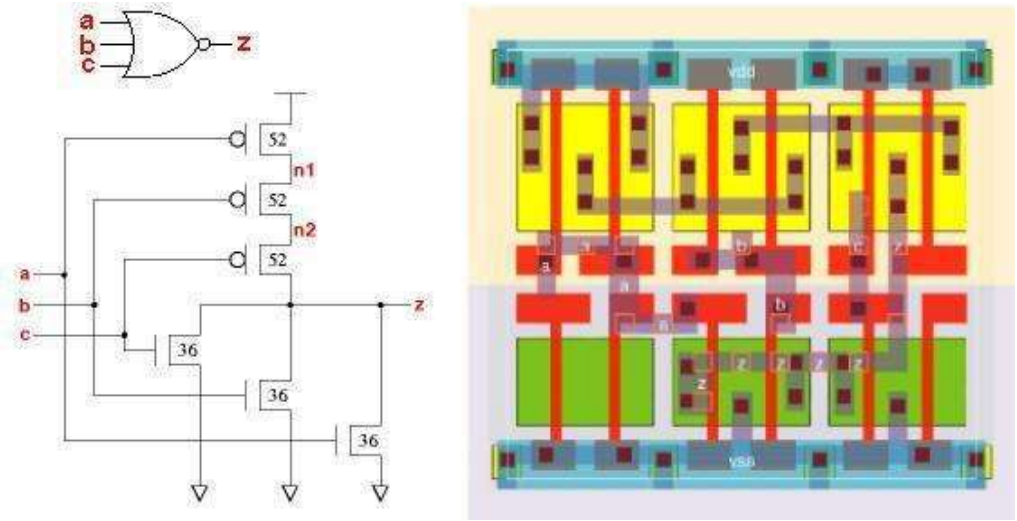
[13] TIMOTHY, Edwards. *Qflow.* http://opencircuitdesign.com/qflow/. Consulta: octubre 2020.

Figura 65. **Función *NR3*, libreria *rgalib013***



Fuente: PETLEY, Graham. *nr3 rgalib013 standard cell family*. http://vlsitechnology.org/html/cells/rgalib013/nr3.html. Consulta: 25 de octubre de 2020.

Figura 66. **Función *NR3*, libreria *vgalib013***



Fuente: PETLEY, Graham. *nr3 vgalib013 standard cell family*. http://vlsitechnology.org/html/cells/vgalib013/nr3.html. Consulta: 25 de octubre de 2020.

1.5.1. Resultados Qflow

Qflow provee varias librerías de células estándar, de las cuales se utilizaron *osu018* y *osu035*, para obtener los recursos utilizados y frecuencia máxima de reloj.

Figura 67. **Resultados *osu035***

```
osu035:
   Number of cells:              10811
     AND2X2                         85
     AOI21X1                      1341
     AOI22X1                       263
     BUFX2                         129
     DFFPOSX1                     1155
     INVX1                         741
     MUX2X1                        418
     NAND2X1                      2007
     NAND3X1                       864
     NOR2X1                       1103
     NOR3X1                         25
     OAI21X1                      2264
     OAI22X1                       282
     OR2X2                          72
     XNOR2X1                        52
     XOR2X1                         10
-------------------------------------------
Computed maximum clock frequency (zero margin) = 95.8497 MHz
```

Fuente: elaboración propia, empleando Qflow v1.4.

Figura 68. **Resultados *osu018***

```
osu018:
   Number of cells:              10353
     AND2X2                        113
     AOI21X1                      1349
     AOI22X1                       179
     BUFX2                         129
     DFFPOSX1                     1155
     INVX1                         737
     MUX2X1                       1233
     NAND2X1                      1810
     NAND3X1                       424
     NOR2X1                        902
     NOR3X1                         34
     OAI21X1                      2115
     OAI22X1                        45
     OR2X2                          47
     XNOR2X1                        52
     XOR2X1                         29
--------------------------------------------
Computed maximum clock frequency (zero margin) = 151.553 MHz
```

Fuente: elaboración propia, empleando Qflow v1.4.

El número total de células estándar utilizadas son 10 811 y 10 353 para las librerías *osu035* y *osu018*, respectivamente, un valor a tomar en cuenta es *DFFPOSX1* ya que representa la cantidad de *flip-flops* utilizados en el procesador, si se describió correctamente el *hardware* en VHDL, la cantidad de *flip-flops* no debe cambiar independientemente de la tecnología en la que sea implementado, ya que se sabe la cantidad y tamaño de los registros utilizados en el procesador, por lo tanto la cantidad de *flip-flops*.

Se utilizaron 32 registros de 32 *bits* en archivo de registros, 1 registro de 32 *bits* en *PC*, en *CSR*, *mvect* es un registro de 32 *bits*, *mcause* es un registro de 32 *bits*, *mepc* es un registro de 30 *bits*, *mstatus* es un registro de 2 *bits, mip* es un registro de 1 *bit*, *mie* es un registro de 1 *bit* y *irq* es un registro de 1 *bit*, dando un total de 1 155 *flip-flops*.

El valor de frecuencia máxima de reloj es de 95,8497 MHz y 151,553 MHz para las librerías *osu035* y *osu018*, hay que tomar en consideración que este valor asume que no existe retraso en el viaje de los cables que conectan las diferentes células estándar, lo cual es falso, por lo tanto, el valor real es ligeramente menor.

1.5.2. Tiempo de ejecución

Para obtener el tiempo de ejecución de un programa, en el simulador se creó un contador para obtener la cantidad de ciclos que toma su ejecución. La cantidad de ciclos es dividida por la frecuencia del reloj para obtener el tiempo de ejecución de cada programa.

Tabla XIII. **Tiempo de ejecución**

Programa	Ciclos	Tiempo de ejecución (ms)	
		osu035	osu018
dhrystone	266 660	2,782	1,760
median	4 224 930	44,079	27,878
mt-matmul	730 880	7,625	4,823
mt-vvadd	261 006	2,723	1,722
multiply	887 306	9,257	5,855
qsort	38 247 921	399,041	252,373
rsort	38 298 441	399,568	252,707
spmv	1 973 309	20,588	13,021
towers	374 523	3,907	2,471
exceptions	283 341	2,956	1,870

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

2. PROCESADOR RISC-V DE 32 *BITS* CON CINCO ETAPAS DE *PIPELINE*

Para aumentar el desempeño del procesador se modificará su micro-arquitectura implementando *pipeline* con el objetivo de aumentar la frecuencia máxima del reloj.

2.1. *Pipeline*

A continuación, se presenta la manera que se calcula el periodo de reloj de un circuito secuencial, el concepto de *pipeline* y como este ayuda a disminuir el periodo de reloj.

2.1.1. Cálculo frecuencia de reloj

Todo circuito digital posee especificaciones que describen su comportamiento a lo largo del tiempo y especificaciones que describen condiciones relacionadas al tiempo que deben ser cumplidas para garantizar un correcto funcionamiento.

Una de estas especificaciones es el tiempo de propagación, denotado como T_{PD}, el cual indica el tiempo que le toma a una señal transportarse desde las entradas de un circuito hasta sus salidas.

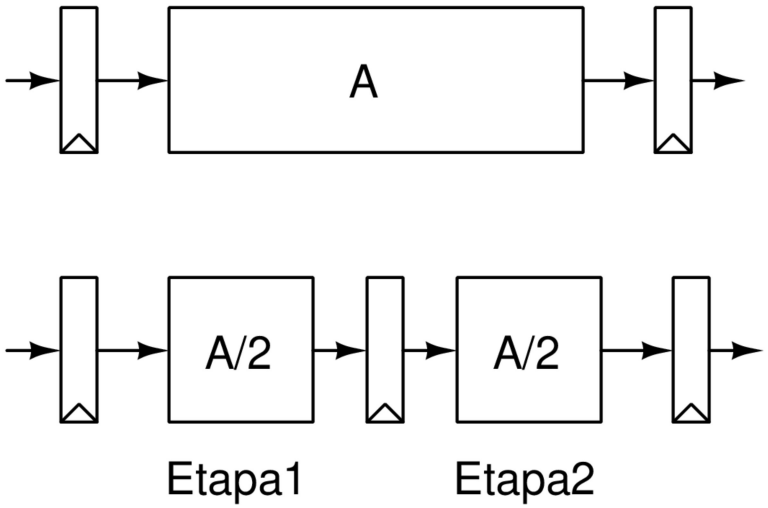
Una especificación referente a los registros es el tiempo de configuración, denotado como T_{SETUP}, el cual indica el tiempo que las entradas de un registro deben estar estables antes del flanco de subida de la señal de reloj.

Si un circuito secuencial está compuesto de un elemento de memoria, un registro, que conecta sus salidas a un circuito combinacional y su resultado es utilizado para sobrescribir el elemento de memoria.[14] el periodo del reloj debe ser mayor o igual que la suma del tiempo de propagación del registro y lógica combinacional, más el tiempo de configuración del registro.

2.1.2. Aumento frecuencia de reloj utilizando *pipeline*

Pipeline es el proceso de separar un circuito combinacional en partes independientes llamadas etapas, con el objetivo de reducir el tiempo de propagación.

Figura 69. **Ejemplo *pipeline***



Fuente: elaboración propia, empleando Xcircuit v3.10.

[14] TERMAN, Chris. *6.004 Computation Structures*. https://computationstructures.org/. Consulta: junio 2020.

En este ejemplo la lógica combinacional es separada en dos diferentes etapas, se agrega un registro entre etapas el cual almacena la salida de la etapa uno para ser utilizada como entrada en la etapa dos. Como resultado el tiempo de propagación es reducido a la mitad, idealmente aumentado la frecuencia de reloj en dos. En caso de que no sea posible separar un circuito de manera que sus etapas posean el mismo tiempo de propagación, la frecuencia de reloj debe ser calculada utilizando la etapa con el mayor tiempo de propagación.

Como consecuencia de agregar *pipeline*, el nuevo circuito posee una latencia de dos ciclos, la cantidad de ciclos necesarios para que una señal atraviese todas las etapas. Pero ahora es posible procesar dos señales diferentes al mismo tiempo, una señal por cada etapa. Como resultado cada ciclo de reloj es procesado por completo una señal, igual que el circuito original, pero con la ventaja de operar a una mayor frecuencia de reloj.

2.2. Agregar *pipeline* al procesador

Conociendo el proceso de ejecución de instrucciones de un procesador, se pueden obtener cinco pasos necesarios para ejecutar una instrucción, los cuales son:

- Obtener instrucción desde memoria con la dirección del contador de programa

- Decodificar la instrucción y obtener los valores de los argumentos, registros o valores inmediatos.

- Ejecutar la instrucción con los argumentos en la unidad lógica aritmética.

- Si se ejecuta una instrucción *LOAD/STORE* acceder a memoria y realizar una transferencia de datos.

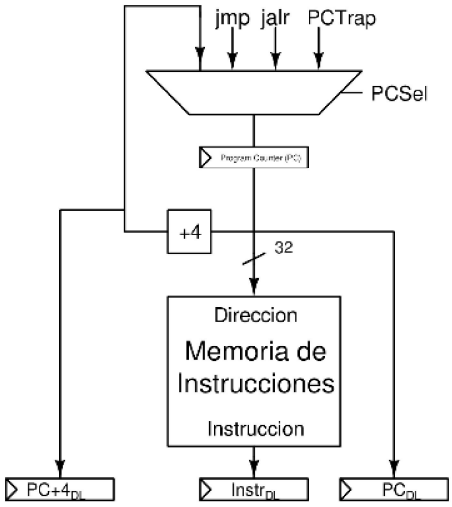
- Seleccionar el valor a ser escrito en el registro *rd*.

Para aplicar *pipeline* de cinco etapas se debe de separar el procesador en cinco etapas representadas por los pasos de ejecución.

2.2.1. Obtención de instrucción, OB

En esta etapa se obtienen las instrucciones utilizando la dirección de memoria almacenada en *PC*.

Figura 70. **Etapa OB**



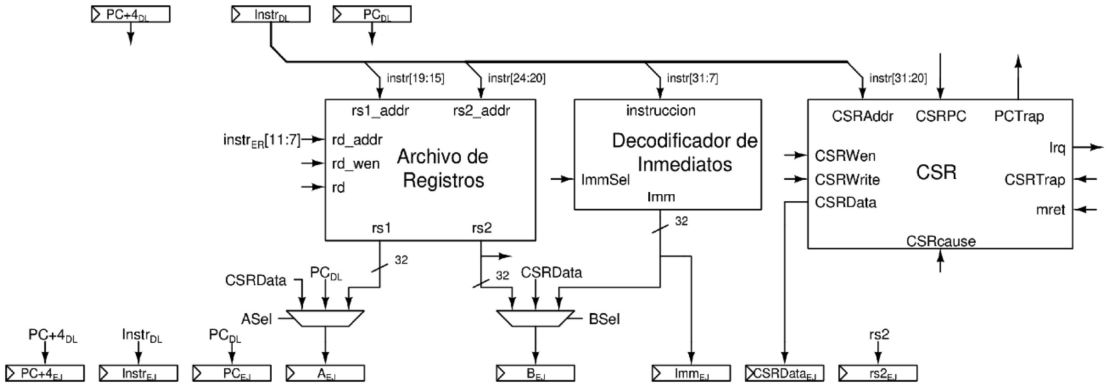
Fuente: elaboración propia, empleando Xcircuit v3.10.

Las señales *PC*, *PC+4* e *instr* son conectados a registros, PC_{DL}, $PC+4_{DL}$, $instr_{DL}$ para ser utilizados en la siguiente etapa.

2.2.2. Decodificación y lectura de registros, DL

En esta etapa se obtienen los valores de los registros de propósito general y *CSR*, adicionalmente se decodifican los valores inmediatos. En el caso de un procesador *CISC* en esta etapa es donde las instrucciones son decodificadas y convertidas en micro-operaciones.

Figura 71. **Etapa DL**



Fuente: elaboración propia, empleando Xcircuit v3.10.

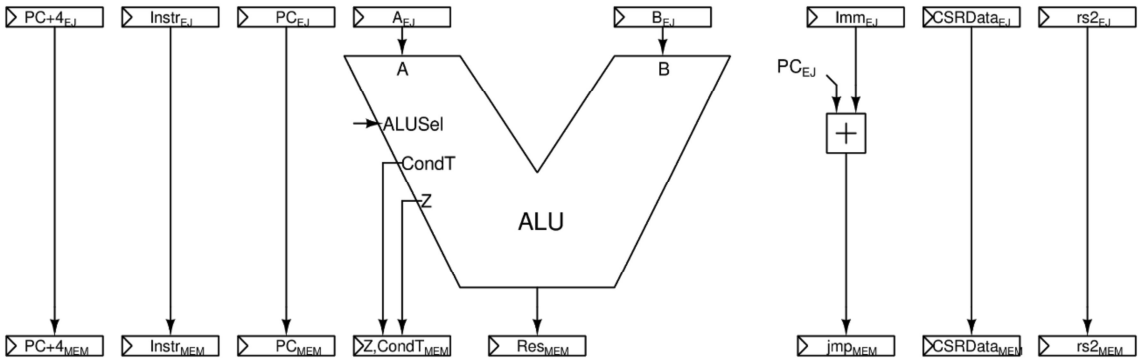
Utilizando la señal $instr_{DL}$ se obtienen los valores de los registros *rs1* y *rs2*, adicionalmente, $instr_{DL}$ es utilizada para decodificar el valor inmediato y obtener el valor del registro *CSR*.

En el multiplexor controlado por la señal *ASel*, la señal *PC* es reemplazada por PC_{DL}, Los valores seleccionados como argumentos para ser operados son escritos en los registros A_{Reg} y B_{Reg} para ser utilizados en la siguiente etapa, adicionalmente el valor del registro *rs2* es escrito en el registro $rs2_{EJ}$ debido a que las instrucciones *STORE* utilizan este valor en su ejecución.

2.2.3. Ejecución de instrucción, EJ

En esta etapa se realizan todas las operaciones lógicas o aritméticas necesarias para ejecutar una instrucción.

Figura 72. **Etapa EJ**



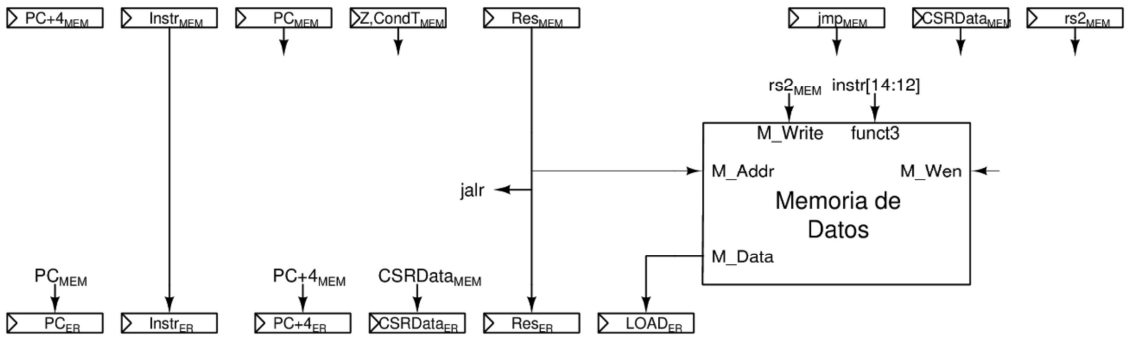
Fuente: elaboración propia, empleando Xcircuit v3.10.

Se utiliza los valores de los registros A_{Reg} y B_{Reg} para ser operados por el *ALU*, el resultado es escrito en el registro Res_{MEM}. El valor del registro PC_{EJ} y Imm_{EJ} son utilizados para obtener el valor de la señal *jmp*, utilizada en instrucciones *BRANCH* y *JAL*, *jmp* es escrito al registro jmp_{MEM}.

2.2.4. Acceso a memoria, MEM

En esta etapa se realizan las transacciones de datos entre registros y memoria de datos.

Figura 73. **Etapa MEM**



Fuente: elaboración propia, empleando Xcircuit v3.10.

En el módulo Memoria de Datos se reemplaza la señal *rs2* por el valor del registro $rs2_{MEM}$, el valor del registro Res_{MEM} es conectado a *M_Addr*, el valor de *M_DATA* es escrito en el registro $LOAD_{ER}$. Para determinar el tamaño de la transferencia se conectan los *bits* 12 al 14 del valor del registro $instr_{MEM}$, en *funct3* del módulo Memoria de Datos.

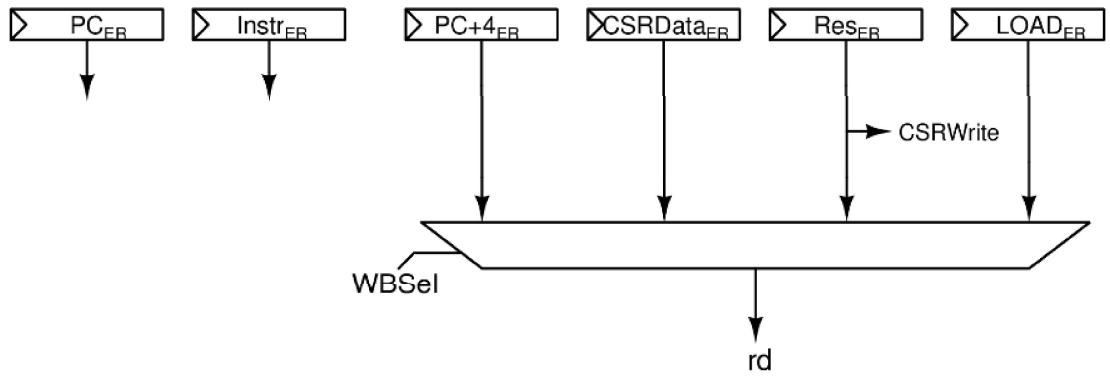
Adicionalmente en esta etapa utilizando las señales Z_{MEM} y $CondT_{MEM}$ se realiza el cálculo para determinar si una instrucción *BRANCH* cumple su condición de salto. Las direcciones destino utilizadas en las instrucciones *JAL*, *BRANCH* son tomadas de la señal jmp_{MEM} y la dirección destino de la instrucción *JALR* es tomada de la señal $Res_{MEM.}$

En esta etapa se modifica el valor de *PCSel,* correspondiente a la ejecución de instrucciones de transferencia de control.

2.2.5. Escritura de registros, ER

En esta etapa se escribe el valor resultante de la instrucción en el registro *rd* y/o *CSR*.

Figura 74. **Etapa ER**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Los *bits* 7 al 11 del registro $instr_{ER}$ son utilizados para seleccionar el registro *rd* a ser escrito, de igual manera $instr_{ER}$ es utilizado para seleccionar el registro *CSR* a ser escrito, los valores de los registros $CSRData_{ER}$, $PC+4_{ER}$, Res_{ER} y $LOAD_{ER}$, son conectados al multiplexor controlado por la señal *WBSel* para seleccionar el valor a ser escrito en *rd*.

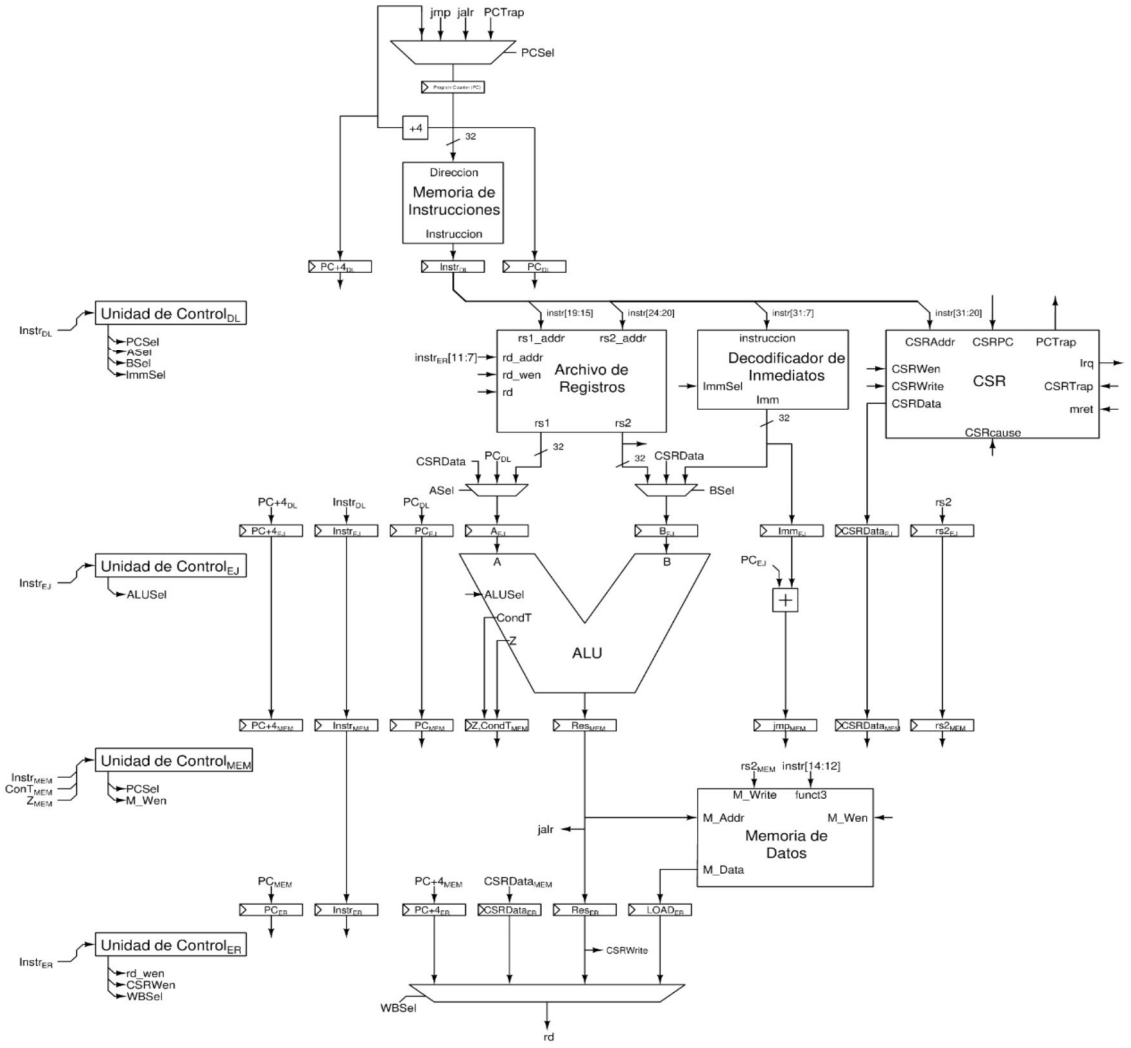
2.2.6. Unidad de control

La unidad de control es separada en cuatro porciones correspondientes a las señales de control en las etapas, DL, EJ, MEM y ER. En estas etapas se utiliza el valor del registro $instr_{Etapa}$, para generar las señales de control respectivas a cada etapa.

2.2.7. Micro-arquitectura preliminar procesador con cinco etapas de *pipeline*

A continuación, se presenta la micro-arquitectura preliminar del procesador con cinco etapas de *pipeline*.

Figura 75. **Micro-arquitectura preliminar procesador con cinco etapas de *pipeline***



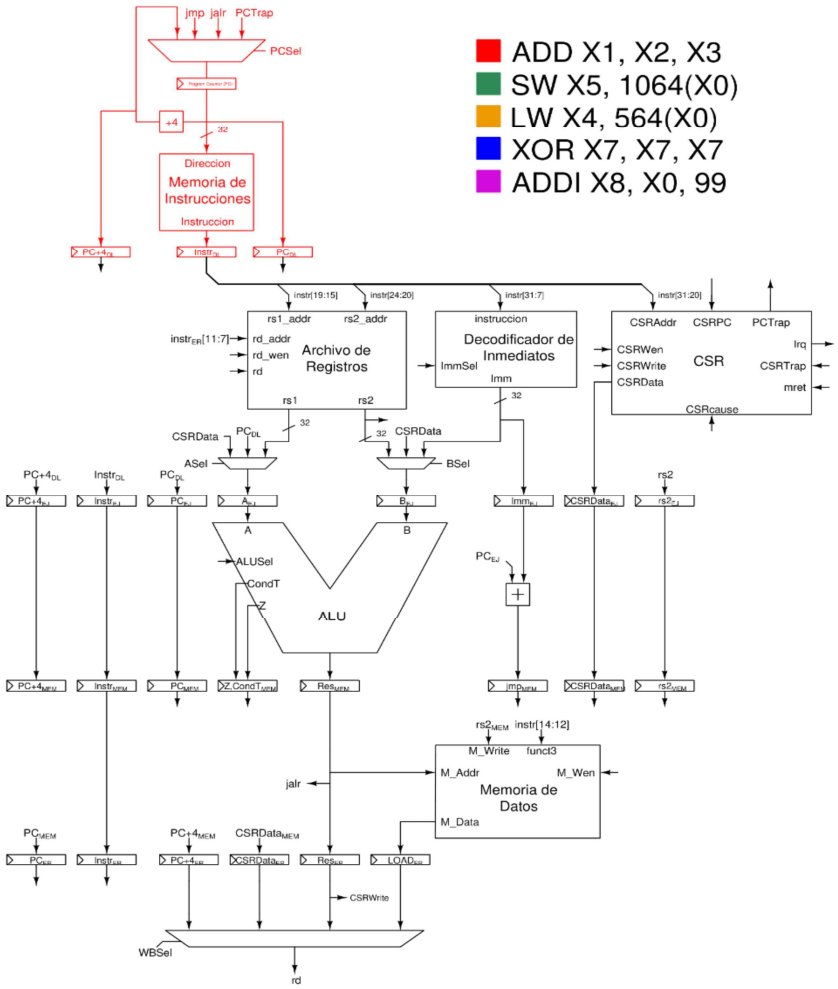
Fuente: elaboración propia, empleando Xcircuit v3.10.

Por el momento se obvia la detección y manejo de excepciones e interrupciones.

2.2.8. Ejecución de instrucciones en procesador con *pipeline*

A continuación, se presenta la figura 76, la cual contiene el estado del procesador en el primer ciclo de la ejecución de un programa.

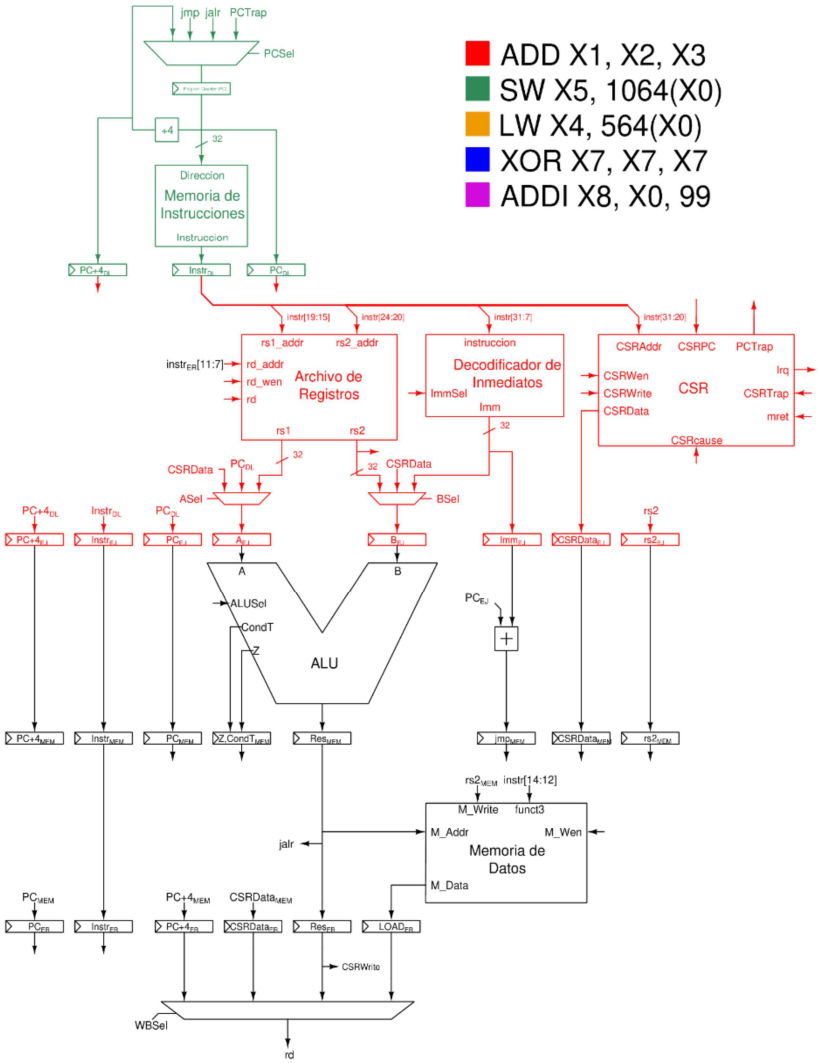
Figura 76. **Ejemplo ejecución ciclo 1**



Fuente: elaboración propia, empleando Xcircuit v3.10.

En el primer ciclo de ejecución del procesador, la primera instrucción, *ADD*, entra en el procesador en la etapa OB en donde es extraída de memoria utilizando el valor de *PC*. Al finalizar el ciclo los valores de las señales *PC+4*, *instr*, y *PC* son escritos en los registros $PC+4_{MEM}$, $instr_{MEM}$, y PC_{MEM}, para ser utilizados en la siguiente etapa.

Figura 77. **Ejemplo ejecución ciclo 2**

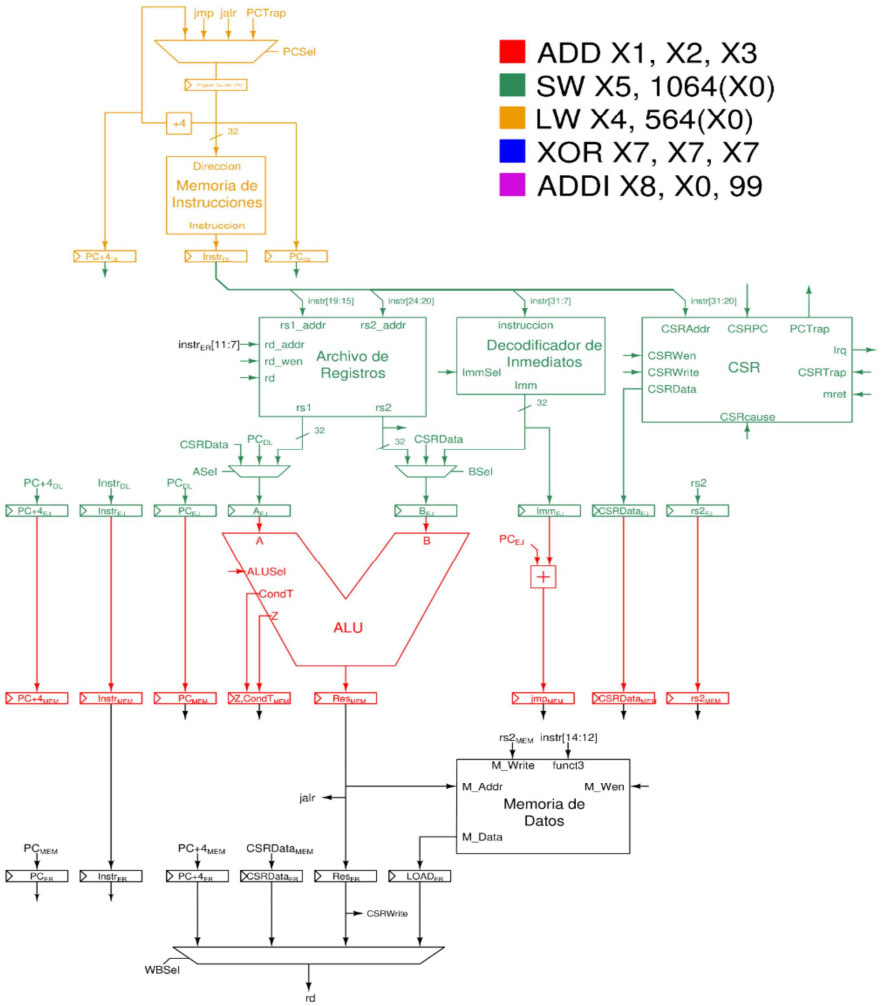


Fuente: elaboración propia, empleando Xcircuit v3.10.

En el segundo ciclo la instrucción *ADD*, entra en la etapa DL, y la instrucción *SW*, ingresa al procesador en la etapa OB.

Al mismo tiempo que se extrae de memoria la instrucción *SW* en la etapa OB, en la etapa DL utilizando la señal $instr_{DL}$ se obtienen los valores de los registros *x2* y *x3*, utilizados en la instrucción *ADD*, los cuales serán escritos en los registros A_{EJ} y B_{EJ} al finalizar el ciclo de reloj.

Figura 78. **Ejemplo ejecución ciclo 3**



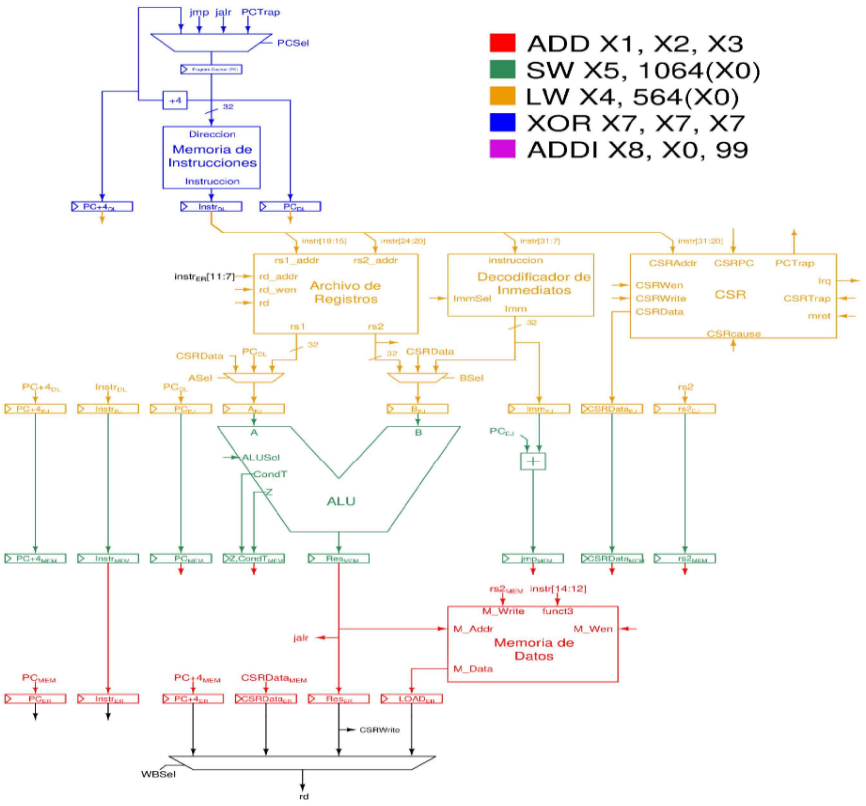
Fuente: elaboración propia, empleando Xcircuit v3.10.

En el tercer ciclo, la instrucción *ADD* entra en la etapa EJ, donde se realiza la suma entre *x2* y *x3*, representado por los registros A_{EJ} y B_{EJ}, el resultado de la operación es escrita en el registro Res_{MEM}.

Así mismo, la instrucción *SW* entra en la etapa DL, donde se obtiene el valor de los registros *x5* y *x0*, además se decodifica el valor inmediato 1 064 de la señal $instr_{DL}$. Al finalizar el ciclo, *x5* es escrito en el registro $rs2_{EJ}$, *x0* es escrito en A_{EJ} y el valor inmediato es escrito en B_{EJ}.

La instrucción *LW* es extraída de memoria, en la etapa OB.

Figura 79. **Ejemplo ejecución ciclo 4**



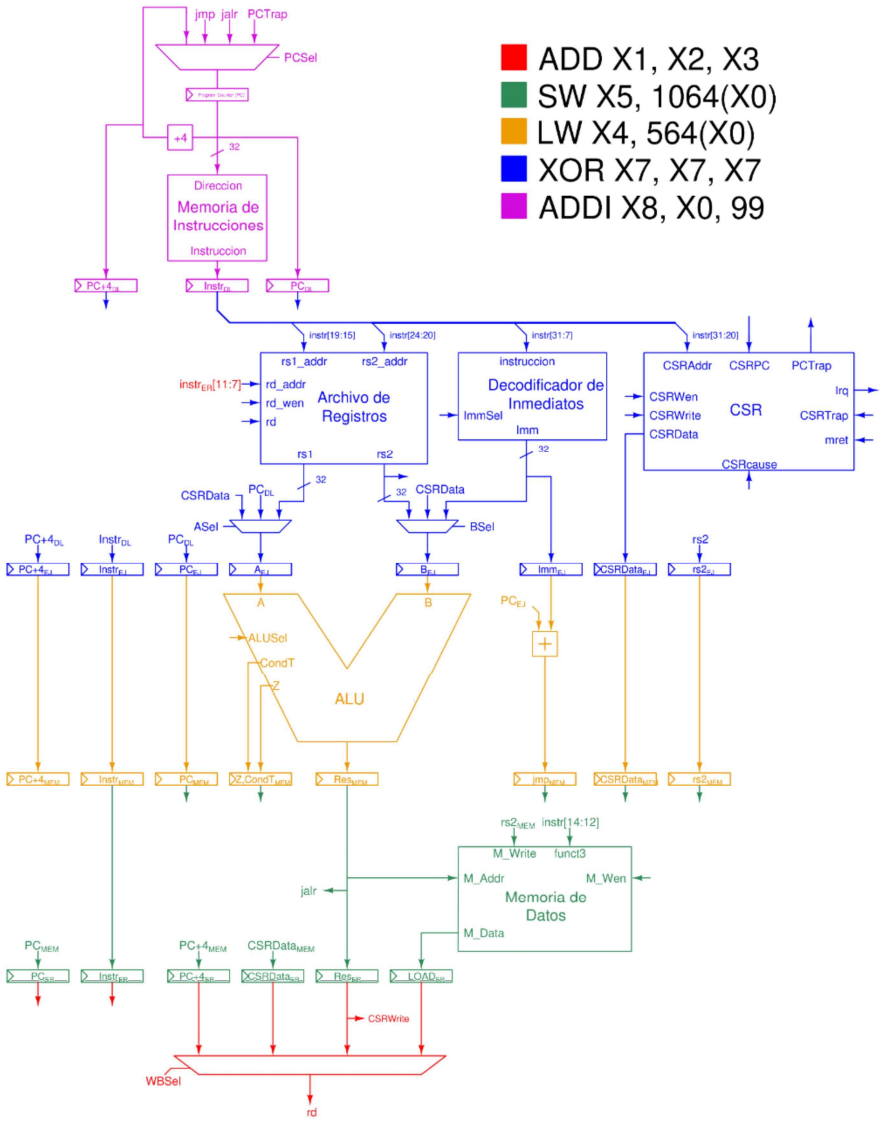
Fuente: elaboración propia, empleando Xcircuit v3.10.

En el cuarto ciclo, la instrucción *ADD* entra en la etapa MEM, la instrucción *SW* entra en la etapa *EJ,* y la instrucción *XOR* entra en la etapa *OB*.

Como la instrucción *ADD* no realiza transferencias a memoria en esta etapa únicamente se transmiten los valores de las señales hacia la etapa ER.

En la etapa EJ se calcula la dirección destino de la instrucción *SW*. En la etapa DL se obtiene el valor de los registros *x4* y *x0*, y se decodifica el valor inmediato 564, de la instrucción *LW*. En la etapa OB se extrae de memoria la instrucción *XOR*.

Figura 80. **Ejemplo ejecución ciclo 5**



Fuente: elaboración propia, empleando Xcircuit v3.10.

En el quinto ciclo, la instrucción *ADD* entra en la etapa ER, donde *WBSel* selecciona el valor de Res_{ER}, para ser escrito en el registro *x1*, por lo tanto, se conectan los *bits* 7 al 11 del registro $instr_{ER}$ en el puerto *rd_addr* del archivo de

registros. Al finalizar el ciclo la instrucción *ADD* finalizará su ejecución y saldrá del *pipeline*.

En la etapa MEM se encuentra la instrucción *SW*, la cual utiliza el valor del registro Res_{MEM}, para seleccionar la dirección destino en la transferencia de memoria, se utiliza el valor de la señal $rs2_{MEM}$, la cual contiene el valor del registro *x5* para ser almacenado en memoria, y se utilizan los *bits* 12 a 14 de la señal $instr_{MEM}$ para seleccionar el tamaño de la transferencia.

En la etapa EJ se realiza la suma entre el valor del registro *x0* y el valor inmediato 564 para obtener la dirección destino de la instrucción *LW*.

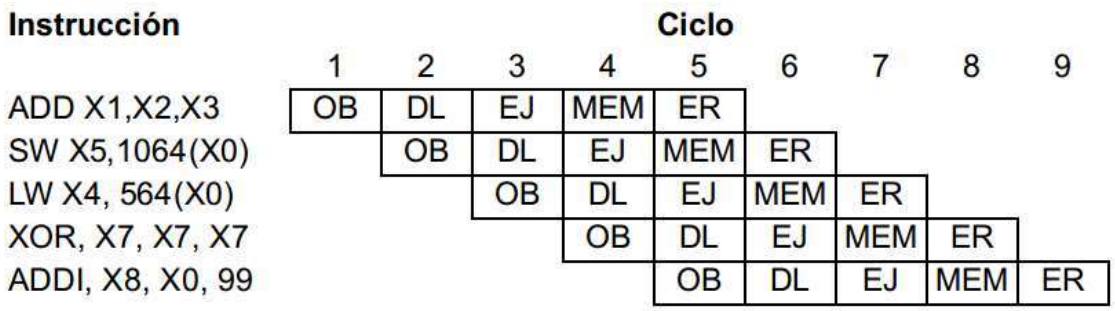
En la etapa DL se obtiene el valor del registro *x7* para ser utilizado en ambos argumentos de la instrucción *XOR*.

En la etapa OB la instrucción *ADDI* entra al procesador al momento de ser extraída de memoria.

2.2.9. Diagramas de *pipeline*

Como se puede observar, visualizar el estado y la ejecución de instrucciones en el procesador puede resultar tedioso y exhaustivo, afortunadamente existe otra manera de visualizar el estado del procesador a través de diagramas de *pipeline*.

Figura 81. **Diagrama de *pipeline***



Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

En la parte superior se encuentran los ciclos de reloj transitados, en la parte izquierda se encuentran las instrucciones que el procesador ejecuta, conforme el avance del tiempo se puede observar que la instrucción transita las diferentes etapas del *pipeline*. Cada ciclo una nueva instrucción inicia el proceso de ejecución en la etapa de Obtención de Instrucción, y como la instrucción antes de ella, transitará por las cinco diferentes etapas hasta concluir su ejecución.

A partir del ciclo número cinco, se observa que el procesador ejecuta cinco instrucciones simultáneamente, aclarando que cada instrucción se encuentra en diferentes etapas de ejecución en el *pipeline*.

Otra observación que es relevante mencionar, es que cada instrucción posee una latencia de cinco ciclos, es decir desde que entra al procesador hasta que finaliza de ejecutarse, se requiere una cantidad de cinco ciclos, sin embargo, se logra observar que a partir del quinto ciclo el procesador finaliza de ejecutar una instrucción en cada ciclo, es decir que del quinto ciclo en adelante el procesador despacha una instrucción por ciclo de reloj. Desde el punto de vista de un programador se podría asumir que el procesador no realiza alguna acción

los primeros cuatro ciclos de ser iniciado, y que a partir del quinto ciclo se ejecuta una instrucción por ciclo.

2.3. Riesgos estructurales

¿Qué sucede si una instrucción *CSR* se encuentra en la etapa DL al mismo tiempo que otra instrucción *CSR* se encuentra en la etapa ER? Ambas instrucciones desean acceder a diferentes registros para ser escritos o leídos, pero el módulo *CSR* únicamente posee un puerto para seleccionar un registro, ya sea para lectura o escritura, por lo tanto, existe conflicto entre estas instrucciones.

Cuando ocurre un conflicto donde dos o más instrucciones desean utilizar el mismo recurso físico se dice que ocurre un riesgo estructural. Para solucionar este riesgo existen dos soluciones, esperar y detener el procesador o añadir hardware adicional.

Para detener y esperar, se debe crear un circuito capaz de detectar el riesgo estructural y un circuito para detener la ejecución de instrucciones del procesador. Al utilizar este método como consecuencia la cantidad de ciclos para ejecutar una instrucción es incrementada, por consiguiente, reduciendo el rendimiento del procesador.

Añadir *hardware* adicional se refiere a duplicar el recurso o rediseñar el recurso para evitar conflictos. A diferencia de detener y esperar, la cantidad de ciclos necesarios para ejecutar una instrucción no es afectada.

Por ejemplo, el procesador posee un riesgo estructural en el módulo *CSR*, en la etapa DL el puerto *CSRAddr* es utilizado para escoger el registro *CSR* que

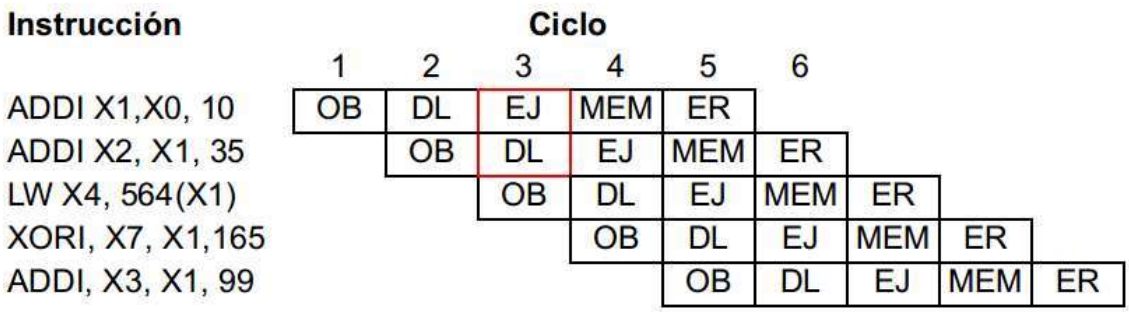
se desea leer, de manera similar en la etapa ER se utiliza el puerto *CSRAddr* para seleccionar el registro *CSR* a escribir, cómo se logra observar hay un conflicto ya que, en dos etapas dos instrucciones acceden el mismo recurso físico.

Para solucionar este riesgo estructural se utilizará el método de añadir *hardware* adicional, se duplicará el circuito selector de registro, con el objetivo de poseer un selector de escritura, puerto *CSRAddrW,* y un selector de lectura, puerto *CSRAddrR.* En la etapa DL se utilizará el puerto *CSRAddrR* y en la etapa ER se utilizará el puerto *CSRAddrW*. Con esto se soluciona el riesgo estructural del procesador.

2.4. Riesgos de datos

Aunque al implementar *pipeline* se separó el procesador en circuitos combinacionales independientes entre sí, a nivel de sistema existe dependencia entre diferentes etapas en casos específicos, por ejemplo:

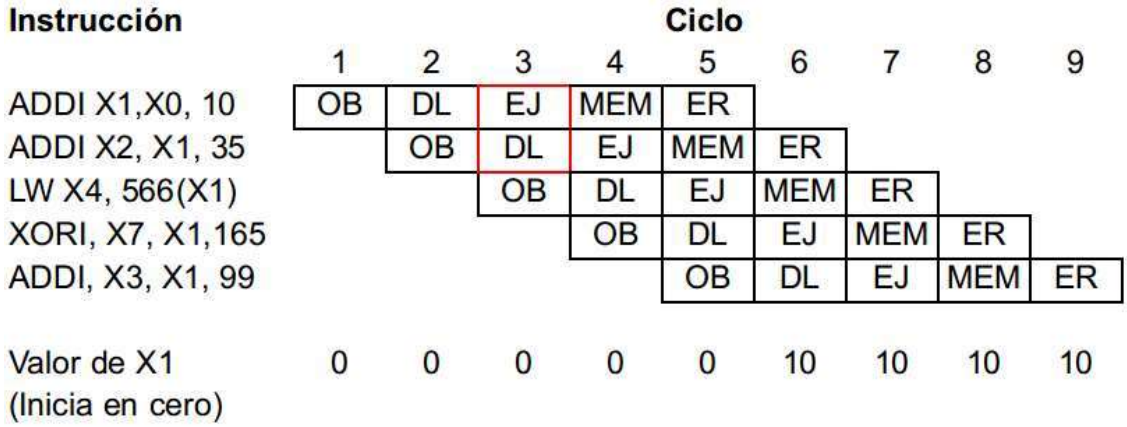
Figura 82. **Ejemplo riesgo de datos**



Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Como se logra observar la primera instrucción modifica el valor del registro *x1* y la segunda instrucción utiliza el valor de *x1* como argumento en su ejecución. En el ciclo tres en la etapa DL ¿Qué valor de *x1* obtendrá la segunda instrucción?

Figura 83. **Ejemplo riesgo de datos con valor de registro *x1***



Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

La última etapa, ER, es la encargada de la escritura de registros, en otras palabras, la primera instrucción no modificará el registro *x1* hasta la finalización del quinto ciclo de reloj. Por lo tanto, el valor de *x1* que obtendrá la segunda instrucción es el valor de *x1* previo a la ejecución de la primera instrucción.

Este problema no es único a la segunda instrucción, sino también a la tercera y cuarta instrucción, la quinta instrucción en adelante no sufre de este problema ya que en el sexto ciclo de reloj donde la quinta instrucción se encuentra en la etapa DL el valor de *x1* ya fue escrito en el archivo de registros.

Aunque cada etapa es independiente entre sí, la ejecución de instrucciones no produce valores correctos, esto es debido a la dependencia que existe entre

instrucciones, en este ejemplo la segunda, tercera y cuarta instrucción, dependen del valor producido por la ejecución de la primera instrucción. Este tipo de dependencia entre instrucciones es conocido como riesgos de datos.

2.4.1. Mitigar riesgo de datos

Para mitigar los riesgos de datos y no alterar la correcta ejecución de instrucciones existen tres diferentes medidas que se pueden tomar, detener el procesador, predecir valores y traspaso de datos.

Detener el procesador como su nombre lo indica, el procesador es detenido hasta que el valor es escrito en el archivo de registros, esto aumenta la cantidad de ciclos necesarios para ejecutar una instrucción, reduciendo el rendimiento del procesador, aunque el costo en *hardware* puede ser bajo.

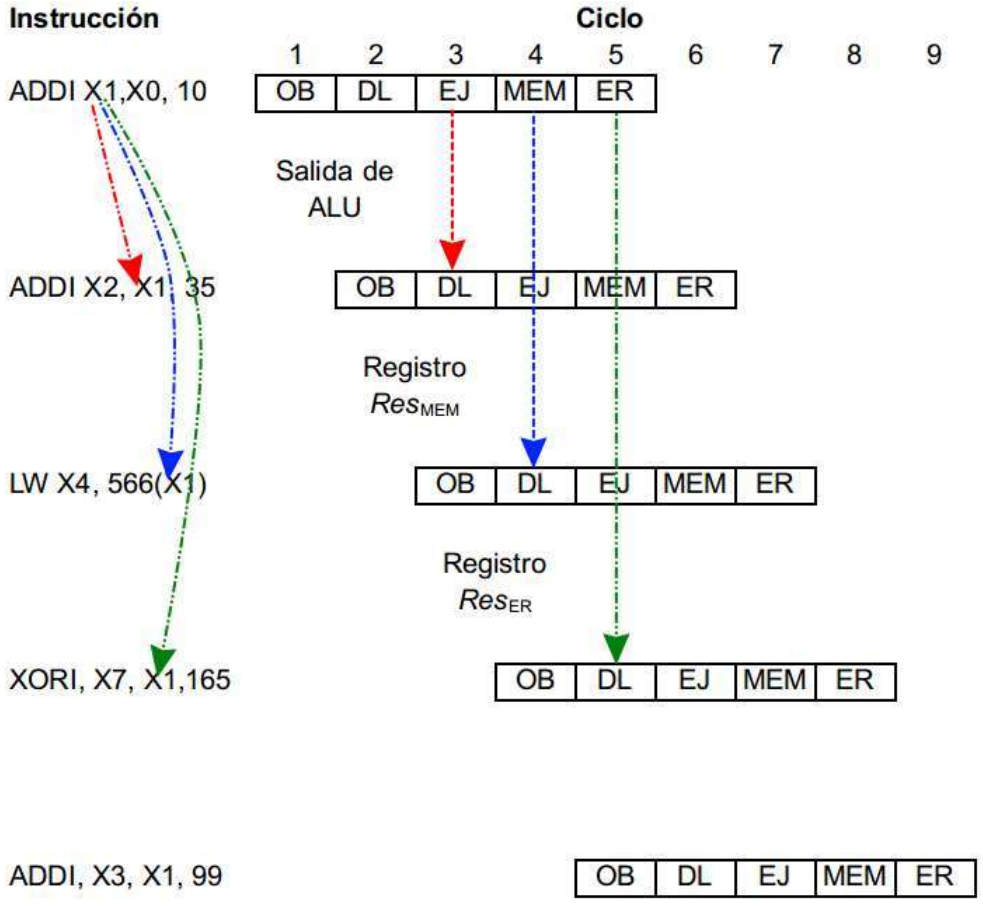
La predicción de valores, como su nombre lo indica se predice el valor esperado por lo tanto el rendimiento se mantiene o es levemente reducido. Para utilizar este método se debe construir un circuito capaz de predecir el valor esperado, un circuito para verificar la predicción y en caso de una predicción equivocada un circuito capaz corregir el error sin alterar la correcta ejecución del programa. Debido a la complejidad de estos circuitos se requiere de una gran cantidad de recursos.

Por último, el traspaso de datos, al momento de existir una dependencia, los datos son transportados de una etapa a otra, esto requiere de un circuito capaz de detectar el riesgo de dato y un circuito que transporte el valor correcto a la etapa que solicita el valor. De forma similar a predicción de valores, mantiene o reduce levemente el rendimiento del procesador, pero se diferencia en que el traspaso de datos requiere menos recursos en hardware.

2.4.2. Traspaso de datos

Como fue discutido previamente los registros son escritos al finalizar la etapa ER, pero esto no significa que el valor a ser escrito no se encuentra presente en el procesador, por ejemplo:

Figura 84. **Traspaso de datos**



Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

El valor actualizado de *x1* solicitado por la segunda instrucción se puede obtener al finalizar la etapa EJ de la primera instrucción, es decir en la salida del

ALU. De igual manera en el cuarto ciclo del reloj el valor actualizado se encuentra en el registro Res_{MEM}. En el quinto ciclo del reloj, el valor actualizado de *x1* se encuentra en el registro Res_{ER}. El traspaso de datos aprovecha estos lugares donde se puede obtener los valores solicitados.

En el caso de presentarse un riesgo de datos en la etapa DL se detecta la presencia de un riesgo de datos, se observa en qué etapa se encuentra el valor solicitado y por último el valor solicitado es transportado hacia la etapa DL y escrito en el registro A_{EJ}, B_{EJ} y/o $rs2_{EJ}$, dependiendo donde sea necesario.

2.4.3. Detener el procesador

Aunque el traspaso de datos es capaz de satisfacer la mayoría de las dependencias en los riesgos de datos, hay ciertos casos que no se pueden resolverse de otra manera más que deteniendo el procesador momentáneamente, por ejemplo:

Figura 85. **Dependencia en instrucciones *LOAD***

Instrucción	Ciclo								
	1	2	3	4	5	6	7	8	9
LW X1, 256(X0)	OB	DL	EJ	MEM	ER				
ADDI X2, X1, 35		OB	DL	EJ	MEM	ER			
ANDI X4, X1, 511			OB	DL	EJ	MEM	ER		
XORI, X7, X1,165				OB	DL	EJ	MEM	ER	
ADDI, X3, X1, 99					OB	DL	EJ	MEM	ER

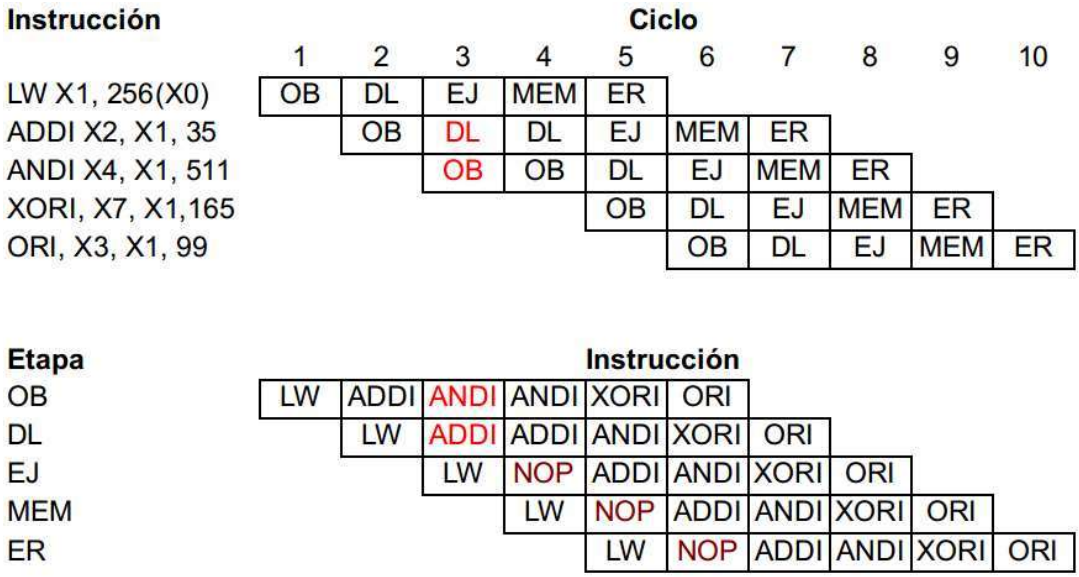
Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

A diferencia de instrucciones registro-registro o registro-inmediato, una instrucción *LOAD* obtiene su valor de una trasferencia de memoria hacia un

registro, es decir obtienen su valor al finalizar su ejecución en la etapa MEM, por lo tanto si una instrucción depende del valor de una instrucción *LOAD* y la instrucción *LOAD* se encuentra en la etapa EJ, donde es calculada la dirección de memoria a ser accedida, el valor solicitado del registro no puede ser traspasado, ya que el valor solicitado no se encuentra presente en ninguna etapa del procesador.

Para solucionarlo la instrucción dependiente de la instrucción *LOAD* debe detener su ejecución hasta que el valor solicitado se encuentre presente en el procesador, es decir debe detener su ejecución hasta que la instrucción *LOAD* ingrese a la etapa *MEM*, que es donde el valor solicitado ingresa al procesador. Esto aumenta la cantidad de ciclos de reloj necesarios para su ejecución, reduciendo el rendimiento, a pesar de ello debe ser incluido para garantizar la correcta ejecución de programas.

Figura 86. **Detener procesador por instrucción *LOAD***



Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

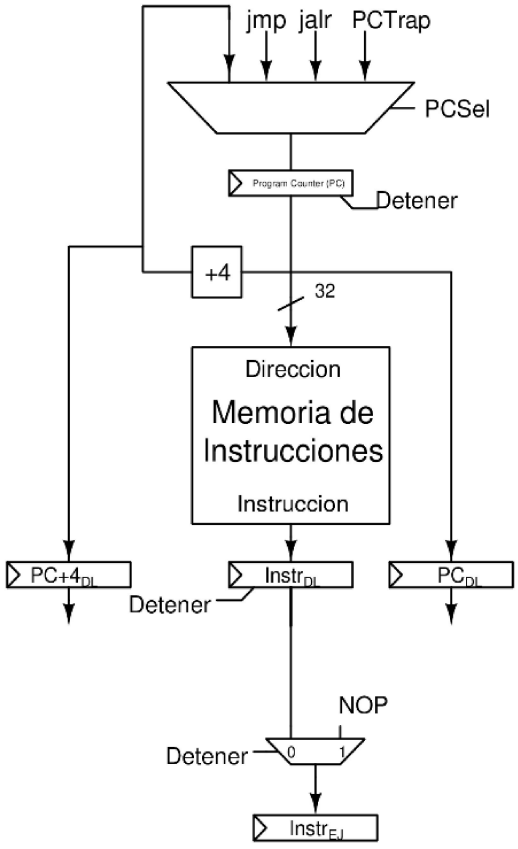
2.4.4. Implementación

A continuación, se presentan los cambios en la micro-arquitectura destinados a mitigar riesgos de datos.

2.4.4.1. Cambios en el *datapath*

Lo primera modificación en el *datapath* es la inserción un circuito capaz de detener el procesador.

Figura 87. **Implementación detener procesador**

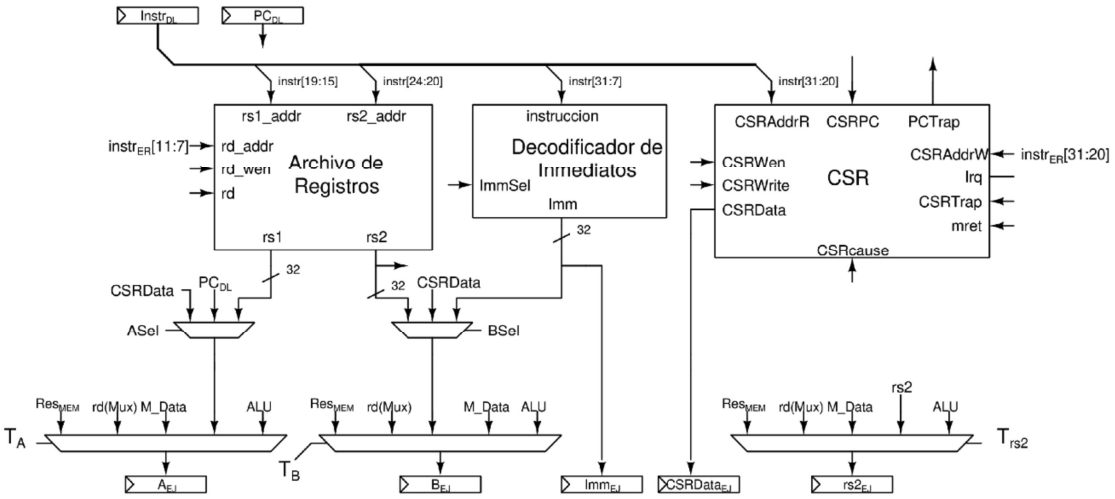


Fuente: elaboración propia, empleando Xcircuit v3.10.

El registro *PC* y el registro $instr_{DL}$ es sustituido por un registro con deshabilitación de escritura, controlado por la señal *Detener*. Cuando *Detener* posee el valor de uno, el registro $instr_{DL}$ y *PC* deshabilitan su escritura, es decir no cambian su valor en el siguiente flanco de subida del reloj. Adicionalmente se añade un multiplexor, controlado por la señal *Detener*, que es conectado al registro $instr_{EJ}$, al cual se conecta el valor del registro $instr_{DL}$ y el valor constante de una instrucción *NOP*, una instrucción *NOP* es una instrucción que al ser ejecutada no altera de ninguna manera el estado del procesador salvo de cambiar el valor de PC a la siguiente instrucción. Una instrucción que cumple con estas características es: *ADD x0, x0, x0*.

Al mantener el valor de *PC*, el procesador no avanza a la siguiente instrucción, y al mantener el valor de $instr_{DL}$, la instrucción que sufre de un riesgo de datos puede intentar obtener los valores en el siguiente ciclo de reloj. El multiplexor es utilizado para sustituir la instrucción detenida en la etapa DL con una instrucción *NOP* en la etapa EJ, de lo contrario se duplicaría la instrucción en el procesador. Con este diseño el procesador puede ser detenido los ciclos que sean necesarios, sin afectar el resultado del programa a ser ejecutado.

Figura 88. **Implementación traspaso de datos**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Para el traspaso de datos es agregado tres nuevos multiplexores en la etapa DL, los cuales son utilizados para traspasar valores a los argumentos A_{EJ}, B_{EJ}, que son utilizados en la etapa EJ, y $rs2_{EJ}$ que es utilizado es la etapa MEM en la ejecución de instrucciones *STORE*.

Los tres multiplexores obtienen sus valores de traspaso de los mismos lugares, en la salida del *ALU*, el registro Res_{MEM}, el valor resultante del multiplexor controlado por *WBSel*, y *M_Data*.

De existir un riesgo de datos, los multiplexores seleccionan el valor necesario para resolver correctamente la dependencia entre instrucciones, de no existir un riesgo de datos, los multiplexores no alterarán los valores a ser escritos en A_{EJ}, B_{EJ} y $rs2_{EJ}$.

2.4.4.2. Detección de riesgos de datos y generación de señales de control

Para que los cambios realizados en el *datapath* sean efectivos en mitigar riesgos de datos que se pueden presentar en la ejecución de un programa, se debe de diseñar un circuito capaz de detectar la presencia de riesgo de datos.

Como un inicio, un riesgo de datos puede ser detectado cuando el registro a leer como argumento en una instrucción en la etapa DL, es el mismo registro destino utilizado en instrucciones en etapas posteriores, con excepción del registro *x0*, ya que es un registro con valor constante cero.

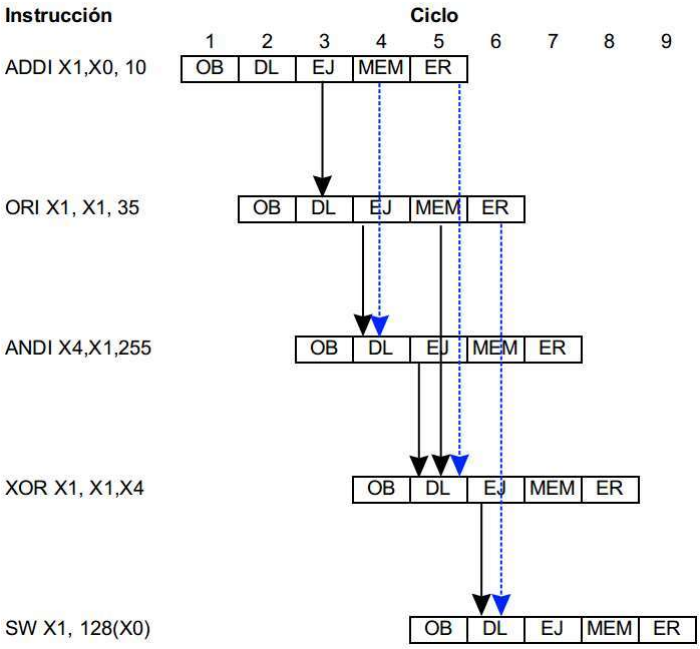
Desafortunadamente comparar si el registro argumento es igual al registro destino de una instrucción anterior no es suficiente, ya que se pueden detectar falsas dependencias, por ejemplo, las instrucciones *LUI* y *AUIPC* no utilizan como argumentos registros de propósito general, en cambio los *bits* que normalmente son utilizados para codificar los registros argumentos son utilizados para codificar valores inmediatos. Por lo tanto, existe la posibilidad que los *bits* 15 a 19 y 20 a 24 del registro $instr_{DL}$, los cuales son utilizados para decodificar *rs1* y *rs2* respectivamente, sean iguales a los *bits* 7 a 11 de los registros $instr_{EJ}$, $instr_{MEM}$ o $instr_{ER}$, utilizados para decodificar el registro destino *rd*, activando un falso positivo, ya que una instrucción *LUI* o *AUIPC* no puede generar riesgos de datos. El problema surge debido a los diferentes formatos de codificación que existen entre instrucciones, en el cual los *bits* utilizados para codificar los registros argumento en un tipo de instrucción son utilizados para codificar valores inmediatos en otro tipo de instrucción.

Adicionalmente, deben ser consideradas las instrucciones que no modifican el valor de ningún registro, como *STORE* o *BRANCH*, las cuales sufren del mismo

problema, por tener diferentes formatos de codificación pueden provocar falsos positivos.

Otro aspecto para tomar en cuenta es, si las condiciones de traspaso se cumplen en dos o más etapas, el valor a traspasar debe ser obtenido de la etapa de mayor proximidad a la etapa DL, en otras palabras, si dos o más etapas poseen un valor a traspasar debe priorizarse el valor de la etapa EJ, seguido de MEM y por último ER.

Figura 89. **Ejemplo traspaso con prioridad**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Las flechas solidas color negro señalan los valores que deben ser traspasados, las flechas punteadas de color azul representan los valores que

cumplen con las condiciones de traspaso, pero debido a la prioridad no son y no deben ser seleccionados para el traspaso de datos.

Por último, no todas las instrucciones utilizan dos registros como argumentos, por tal motivo debe diseñarse diferentes circuitos de detección y generación de señales de control para cada multiplexor de traspaso, ya que las condiciones de traspaso son diferentes en los tres multiplexores.

Tabla XIV. **Tabla de verdad control de traspaso T_A**

noJAL & noLUI & noAUIPC & noCSRI (DL)	noBRANCH & noSTORE (EJ)	$Rd_{EJ} = RS1_{DL}$ & Rd_{EJ} =/ 0	noBRANCH & noSTORE (MEM)	$Rd_{MEM} = RS1_{DL}$ & Rd_{MEM} =/ 0	noBRANCH & noSTORE (ER)	$Rd_{ER} = RS1_{DL}$ & Rd_{ER} =/ 0	$isLOAD_{MEM}$	Traspaso
1	1	1	-	-	-	-	-	ALU
1	0	-	0	0	0	0	-	No
1	-	0	0	0	0	0	-	No
1	0	0	1	1	-	-	1	M_DATA
1	0	0	1	1	-	-	0	Res_{MEM}
1	0	0	0	-	0	0	-	No
1	0	0	-	0	0	0	-	No
1	0	0	0	0	1	1	-	Rd(Mux)
1	0	0	0	0	0	-	-	No
1	0	0	0	0	-	0	-	No
0	-	-	-	-	-	-	-	No

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Tabla XV. **Tabla de verdad control de traspaso T_B**

isBRANCH | isOP (DL)	noBRANCH & noSTORE (EJ)	$Rd_{EJ} = RS2_{DL}$ & Rd_{EJ} =/ 0	noBRANCH & noSTORE (MEM)	$Rd_{MEM} = RS2_{DL}$ & Rd_{MEM} =/ 0	noBRANCH & noSTORE (ER)	$Rd_{ER} = RS2_{DL}$ & Rd_{ER} =/ 0	$isLOAD_{MEM}$	Traspaso
1	1	1	-	-	-	-	-	ALU
1	0	-	0	0	0	0	-	No
1	-	0	0	0	0	0	-	No
1	0	0	1	1	-	-	1	M_DATA
1	0	0	1	1	-	-	0	Res_{MEM}
1	0	0	0	-	0	0	-	No
1	0	0	-	0	0	0	-	No
1	0	0	0	0	1	1	-	Rd(Mux)
1	0	0	0	0	0	-	-	No
1	0	0	0	0	-	0	-	No
0	-	-	-	-	-	-	-	No

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Tabla XVI. **Tabla de verdad control de traspaso T_{rs2}**

isSTORE (DL)	noBRANCH & noSTORE (EJ)	$Rd_{EJ} = RS2_{DL}$ & Rd_{EJ} =/ 0	noBRANCH & noSTORE (MEM)	$Rd_{MEM} = RS2_{DL}$ & Rd_{MEM} =/ 0	noBRANCH & noSTORE (ER)	$Rd_{ER} = RS2_{DL}$ & Rd_{ER} =/ 0	$isLOAD_{MEM}$	Traspaso
1	1	1	-	-	-	-	-	ALU
1	0	-	0	0	0	0	-	No
1	-	0	0	0	0	0	-	No
1	0	0	1	1	-	-	1	M_DATA
1	0	0	1	1	-	-	0	Res_{MEM}
1	0	0	0	-	0	0	-	No
1	0	0	-	0	0	0	-	No
1	0	0	0	0	1	1	-	Rd(Mux)
1	0	0	0	0	0	-	-	No
1	0	0	0	0	-	0	-	No
0	-	-	-	-	-	-	-	No

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Las columnas que poseen señales con el nombre *noOPCODE* en conjunto del símbolo “&” se refieren a que el *opcode* de dicha etapa, debe ser diferente a todos los enunciados en dicha columna para que la condición sea verdadera. Por ejemplo, la primera columna de la tabla de verdad de control de traspaso T_A, para que esa columna sea verdadera se requiere que la instrucción en la etapa DL, posea un *opcode* diferente a *JAL*, *AUIPC*, *LUI* y no se una instrucción *CSRI*, esto

es debido a que ninguna de estas instrucciones utiliza el valor del registro *rs1*, por lo tanto, no generan riesgo de datos.

Las columnas que poseen señales con el nombre *isOPCODE* en conjunto del símbolo "|", requieren que la instrucción sea una de las enmarcadas en dicha columna para que su valor posea el valor de uno. Por ejemplo, la primera columna de la tabla de verdad del control de traspaso T_B, requiere que la instrucción en la etapa DL posea un *opcode BRANCH* o *OP*, para que su condición sea verdadera, esto debido a que únicamente estos dos tipos de instrucciones pueden ser causante de un riesgo de datos en el registro B_{EJ}.

Las columnas que poseen el enunciado "$Rd_{Etapa} = RS\#_{DL}$ & Rd_{Etapa} =/ 0" se refieren a que el registro destino de dicha etapa debe ser igual al registro origen en la etapa DL y el registro destino de dicha etapa debe ser distinto a cero.

2.4.4.3. Generar señal Detener

La señal de $Detener_{LOAD}$ debe de poseer el valor de uno al momento de detectar un riesgo causado por la dependencia a una instrucción *LOAD* que se encuentre en la etapa EJ.

Tabla XVII. **Tabla de verdad señal Detener*Load***

isLOAD (EJ)	noJAL & noLUI & noAUIPC & noCSRI (DL)	$Rd_{EJ} = RS1_{DL}$ & Rd_{EJ} =/ 0	isBRANCH | isSTORE | isOP (DL)	$Rd_{EJ} = RS2_{DL}$ & Rd_{EJ} =/ 0	$Detener_{LOAD}$
1	1	1	-	-	1
1	-	0	0	0	0
1	0	-	0	0	0
1	-	-	1	1	1
1	0	0	-	0	0
1	0	0	0	-	0
0	-	-	-	-	0

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

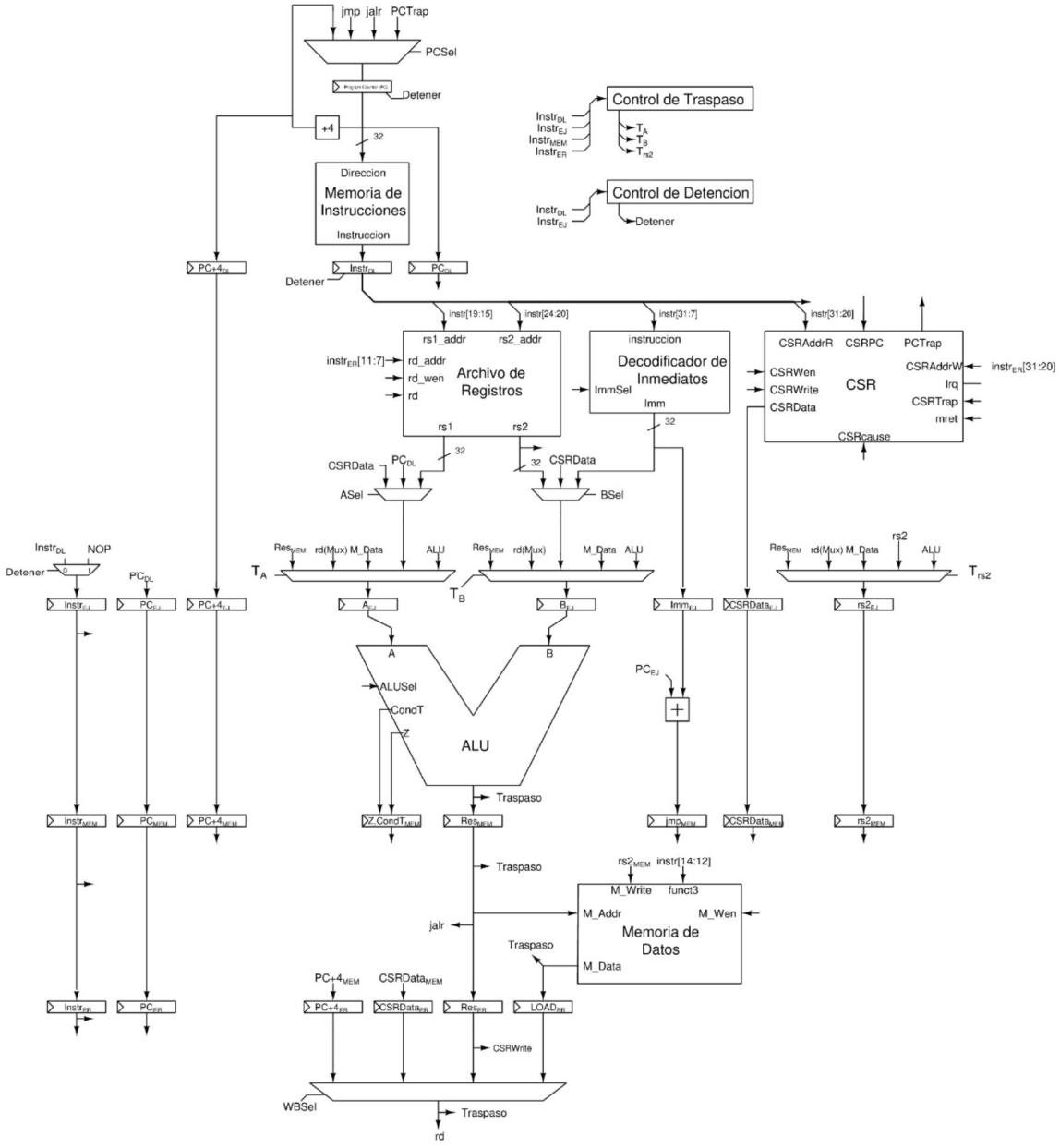
Por último, la señal *Detener* será activada al ejecutarse instrucciones *CSR*, con el objetivo obviar el diseño de circuitos adicionales que eviten riesgos de datos en registros *CSR*, esto quiere decir que el procesador será detenido hasta que la instrucción *CSR* termine de ser ejecutada. Para ello se utiliza el *opcode* en la etapa EJ, MEM o ER para comprobar si posee el valor de *SYSTEM*, representado en las señales $isSYSTEM_{EJ}$, $isSYSTEM_{MEM}$ y $isSYSTEM_{ER}$, estas señales son utilizadas como entradas en una compuerta *OR* dando como resultado la señal $Detener_{SYSTEM}$

La señal *Detener* es generada de la función *OR* entre $Detener_{SYSTEM}$ y $Detener_{Load}$. Con esta señal se mitigan los riesgos de datos generados por las instrucciones *LOAD* y *CSR*.

2.4.5. Micro-arquitectura sin riesgos de datos

A continuación, se presenta la figura 90, la cual contiene la micro-arquitectura sin riesgos de datos.

Figura 90. **Micro-arquitectura sin riesgos de datos**

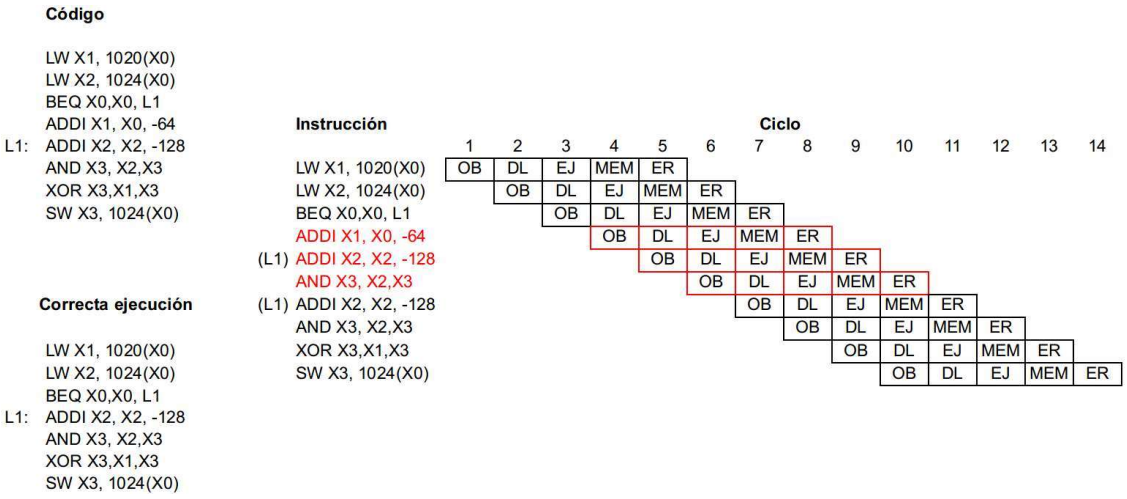


Fuente: elaboración propia, empleando Xcircuit v3.10.

2.5. Riesgos de control

Otro tipo de riesgo causado por *pipeline* son los riesgos de control, los cuales se presentan en la ejecución de instrucciones de control de flujo, por ejemplo:

Figura 91. **Ejemplo riesgo de control**



Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Según el programa la instrucción *beq* debe realizar un salto a la etiqueta *L1* si *x0* es igual a *x0*, obviamente esta condición es verdadera pero al observar el diagrama de *pipeline* el salto ocurre tres ciclos después que la instrucción entra al procesador, como consecuencia el procesador ejecuta tres instrucciones que de acuerdo a la correcta ejecución del programa, no debieron ser ejecutadas, esto ocurre debido a que el valor de *PC* es modificado hasta la etapa MEM, ya que es donde se obtiene la dirección destino y se decide si el salto debe ser tomado.

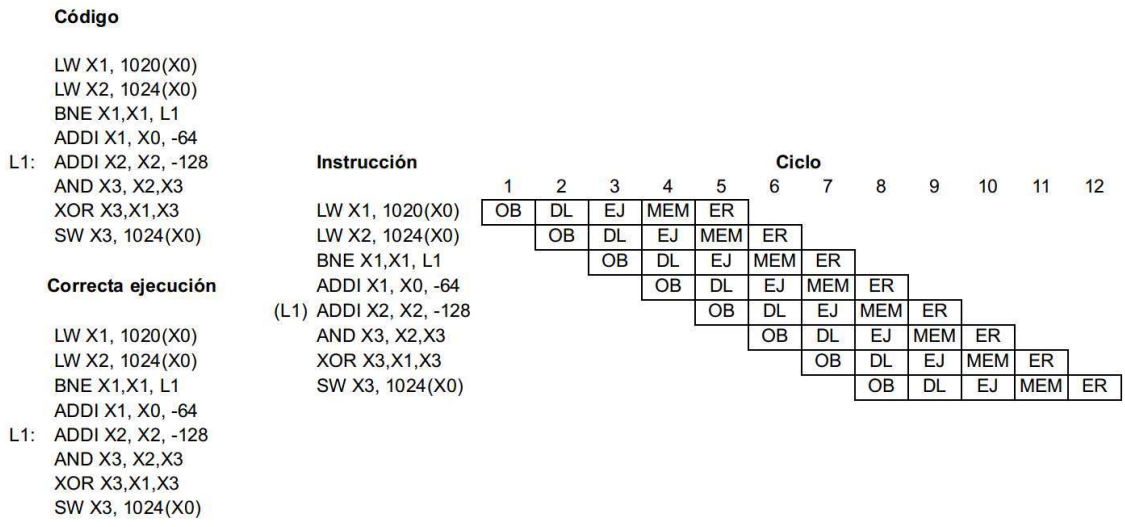
La ejecución de instrucciones adicionales o la modificación no deseada del flujo de ejecución de un programa, debido al retardo del cálculo de condiciones y direcciones objetivos al ejecutar una instrucción de transferencia control, es conocido como riesgo de control.

2.5.1. Mitigar riesgos de control

Una opción para mitigar los riesgos de control es detener el procesador hasta que la condición y dirección destino sea calculada en la etapa MEM. Esto eliminaría la ejecución de instrucciones adicionales.

Como segunda opción y el método a utilizar es especulación, esto surge de la observación de que el programa no es alterado al ejecutarse una instrucción de salto condicional con una condición falsa.

Figura 92. **Instrucción *bne* con condición falsa**



Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

A pesar del retardo al ejecutar instrucciones de transferencia de control, como la condición de la instrucción *bne* es falsa, el valor de *PC* no debe ser alterado, como se logra observar el programa no fue alterado y mantiene su correcta ejecución.

La especulación proviene de predecir que todas las instrucciones *BRANCH* poseerán condiciones falsas. Si esta predicción es correcta el programa se ejecutará correctamente.

Al comparar la especulación con la detención del procesador, la especulación obtiene una ventaja en rendimiento, al utilizar el método de detener el procesador, todas las instrucciones de transferencia de control sufren un retardo de tres ciclos en su ejecución, sin embargo, si se predice que todas las instrucciones *BRANCH* poseen condiciones falsas, las instrucciones en las que esta predicción sea correcta no sufrirán de retardos.

Por ejemplo, si un programa ejecuta mil instrucción *BRANCH* de las cuales un cuarenta por ciento poseen condiciones falsas, es decir saltos no tomados, un procesador que utiliza detención del procesador para mitigar riesgos de control, ejecutará esas mil instrucciones en cuatro mil ciclos, un ciclo por la instrucción en sí misma y tres ciclos por la detención del procesador, y un procesador que utiliza especulación las ejecutará en dos mil ochocientos ciclos, 1 000x(0,4x1 + 0,6x4).

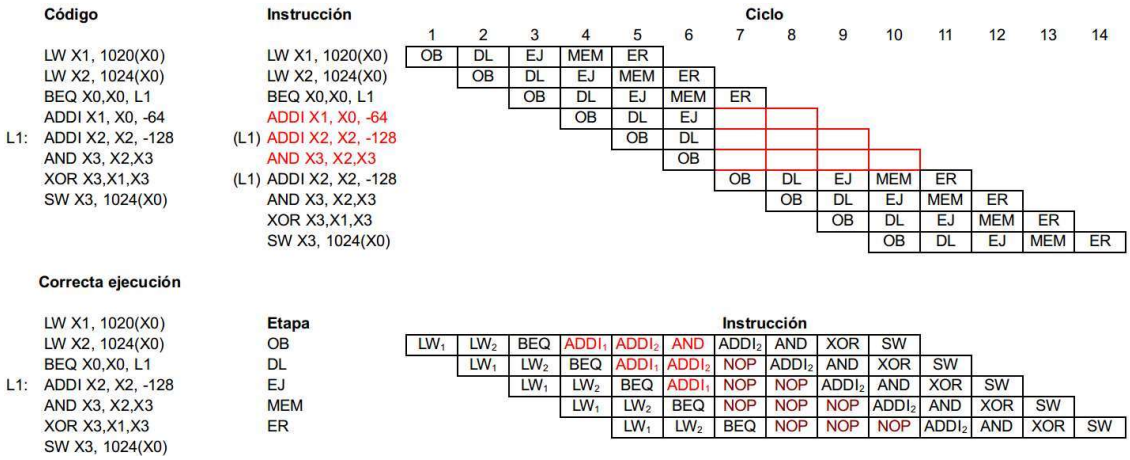
2.5.2. Limpieza de *pipeline*

Se presentó la idea de aumento en el rendimiento al utilizar especulación al contrario de detener el procesador, esto es cierto cuando la predicción de una condición es correcta, pero desafortunadamente este no es siempre el caso, para los casos en que la predicción es incorrecta es necesario eliminar las

instrucciones adicionales producidas por la ejecución de una instrucción de transferencia de control. En otras palabras, se necesita limpiar las etapas de *pipeline* del procesador que contengan instrucciones problemáticas.

Para eliminar instrucciones problemáticas se sustituyen dichas instrucciones por instrucciones *NOP*, en las etapas que sea necesario, particularmente antes de que las instrucciones problemáticas entren en las etapas MEM o ER, ya que en estas etapas se realizan cambios permanentes en el estado del procesador, los cuales son escritura de memoria y escritura de registros, respectivamente.

Figura 93. **Limpieza de *pipeline***



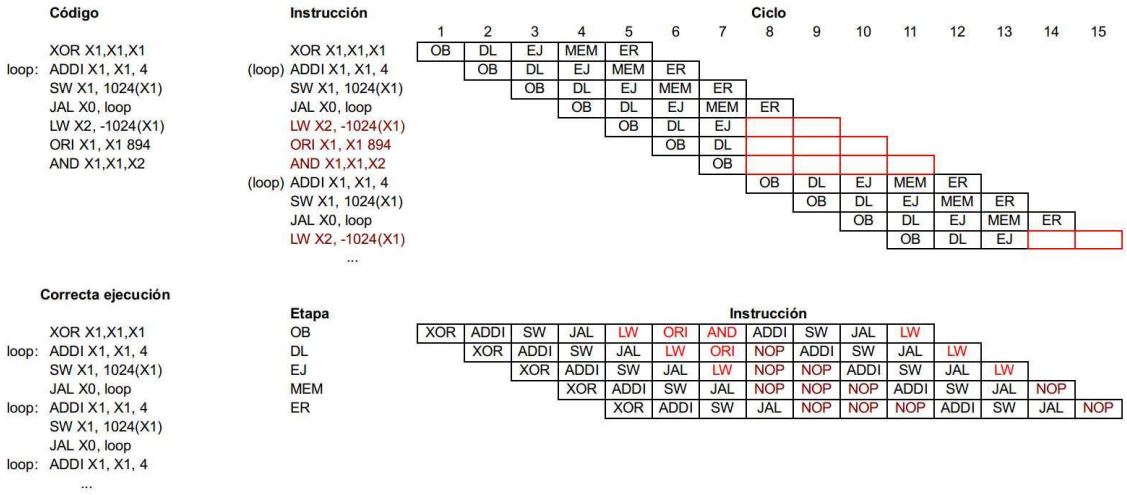
Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Como se puede observar al finalizar la etapa MEM de la instrucción *beq*, las instrucciones *ADDI* y *AND*, fueron reemplazadas por *NOP*.

Aunque solo se han mencionado instrucciones *BRANCH* este método es utilizado por las instrucciones *JAL* y *JALR*, ya que se pueden interpretar, dentro

del contexto de riesgos de control, como instrucciones *BRANCH* con condiciones siempre verdaderas, en otras palabras, al finalizar la etapa MEM de una instrucción *JAL* o *JALR* se realiza una limpieza en el *pipeline*.

Figura 94. **Limpieza de *pipeline* instrucción *JAL***



Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

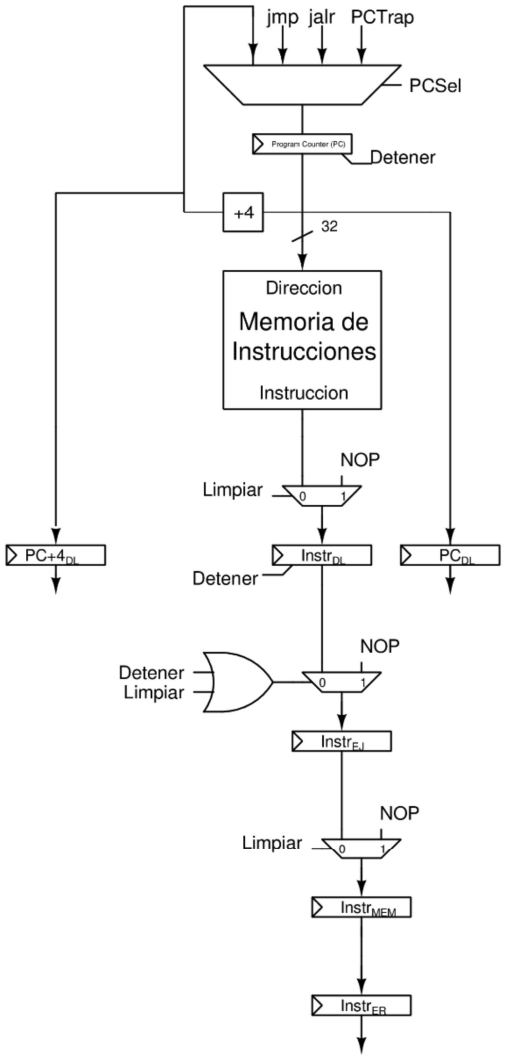
2.5.3. Implementación

A continuación, se presentan los cambios en la micro-arquitectura destinados a mitigar riesgos de control.

2.5.3.1. Cambios en el *datapath*

Si se ejecutan instrucciones *BRANCH* con condiciones falsas no es necesario modificar el *datapath*, pero para las instrucciones *JAL*, *JALR* y *BRANCH* con condiciones verdaderas el *datapath* debe ser modificado para poder realizar una limpieza en las etapas necesarias.

Figura 95. **Implementación limpieza de *pipeline***



Fuente: elaboración propia, empleando Xcircuit v3.10.

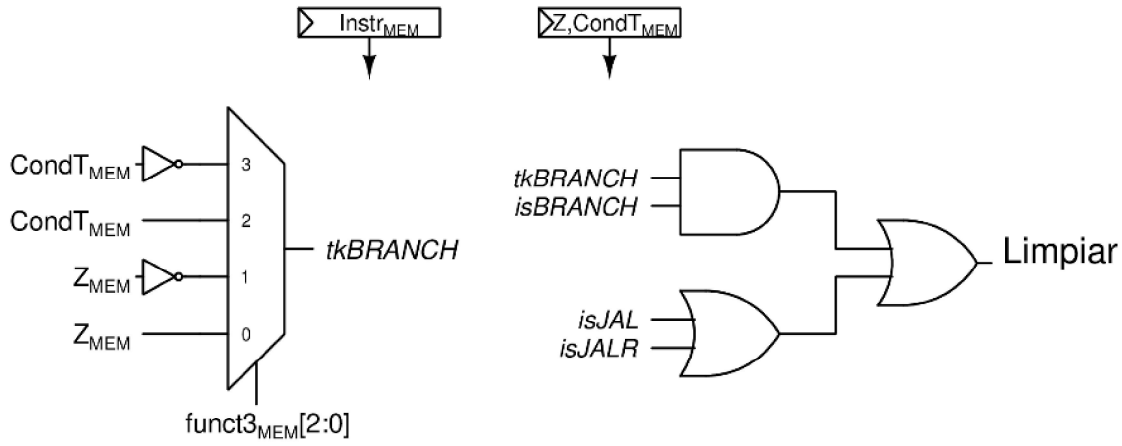
Para eliminar instrucciones problemáticas, el valor de los registros $instr_{ID}$, $instr_{EJ}$ y $instr_{MEM}$, debe ser reemplazado por una instrucción *NOP*. Para ello se agregaron multiplexores controlados por la señal *Limpiar.* Al detectar un riesgo de control, la señal *Limpiar* poseerá el valor de uno eliminando las instrucciones problemáticas, sustituyéndolas por instrucciones *NOP*.

El multiplexor que controla la escritura del registro $instr_{EJ}$, es utilizado en mitigación de riesgos de control y en mitigación de riesgos de control, por lo tanto, es controlado por las señales *Detener* y *Limpiar*.

2.5.3.2. Detección de riesgos de control y generación de señales de control

A continuación, se presentan los cambios en el *datapath* referentes a la detección de riesgos de control y generación de señales de control.

Figura 96. **Implementación señal limpiar**

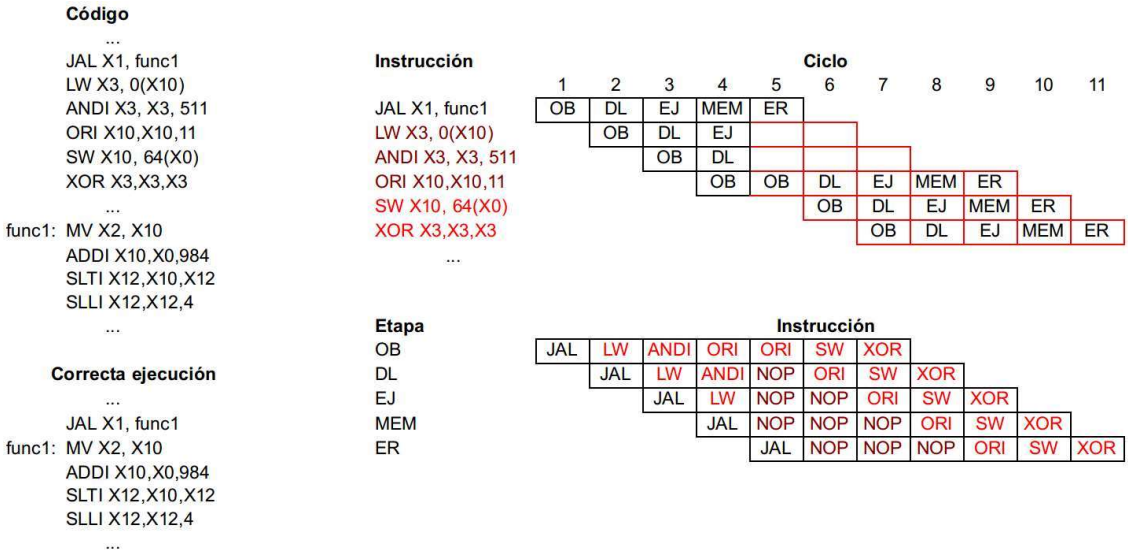


Fuente: elaboración propia, empleando Xcircuit v3.10.

Se debe limpiar el *pipeline* al ejecutar una instrucción *JAL*, *JALR* o *BRANCH* con condición verdadera, la señales *isJAL*, *isJALR* y *isBRANCH* son utilizadas para detectar la presencia de dichas instrucciones, y la señal *tkbranch* es utiliza para indicar una condición verdadera.

Por último, se debe resolver un problema que surge en la presencia de un riesgo de datos y un riesgo de control.

Figura 97. **Ejemplo riesgo de datos y control**

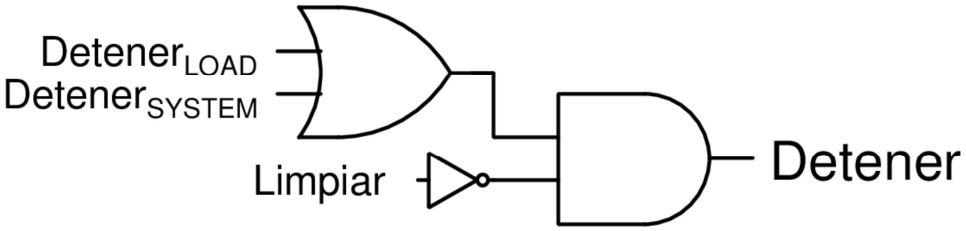


Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Como se puede observar, si una instrucción problemática de un riesgo de control genera un riesgo de datos que debe ser resuelto con la detención del procesador al mismo tiempo que una instrucción de transferencia de control realiza un cambio en *PC*, dicho cambio no será registrado, esto es debido a que la señal detener deshabilita la escritura del registro *PC*.

Como una instrucción problemática no debe ser ejecutada el riesgo de dato generado es un falso positivo, para eliminar este problema la señal *Detener* debe poseer el valor de cero cuando la señal *Limpiar* posea el valor de uno, que es el momento que una instrucción de transferencia de control cambia el valor de *PC*.

Figura 98. **Nueva señal detener**

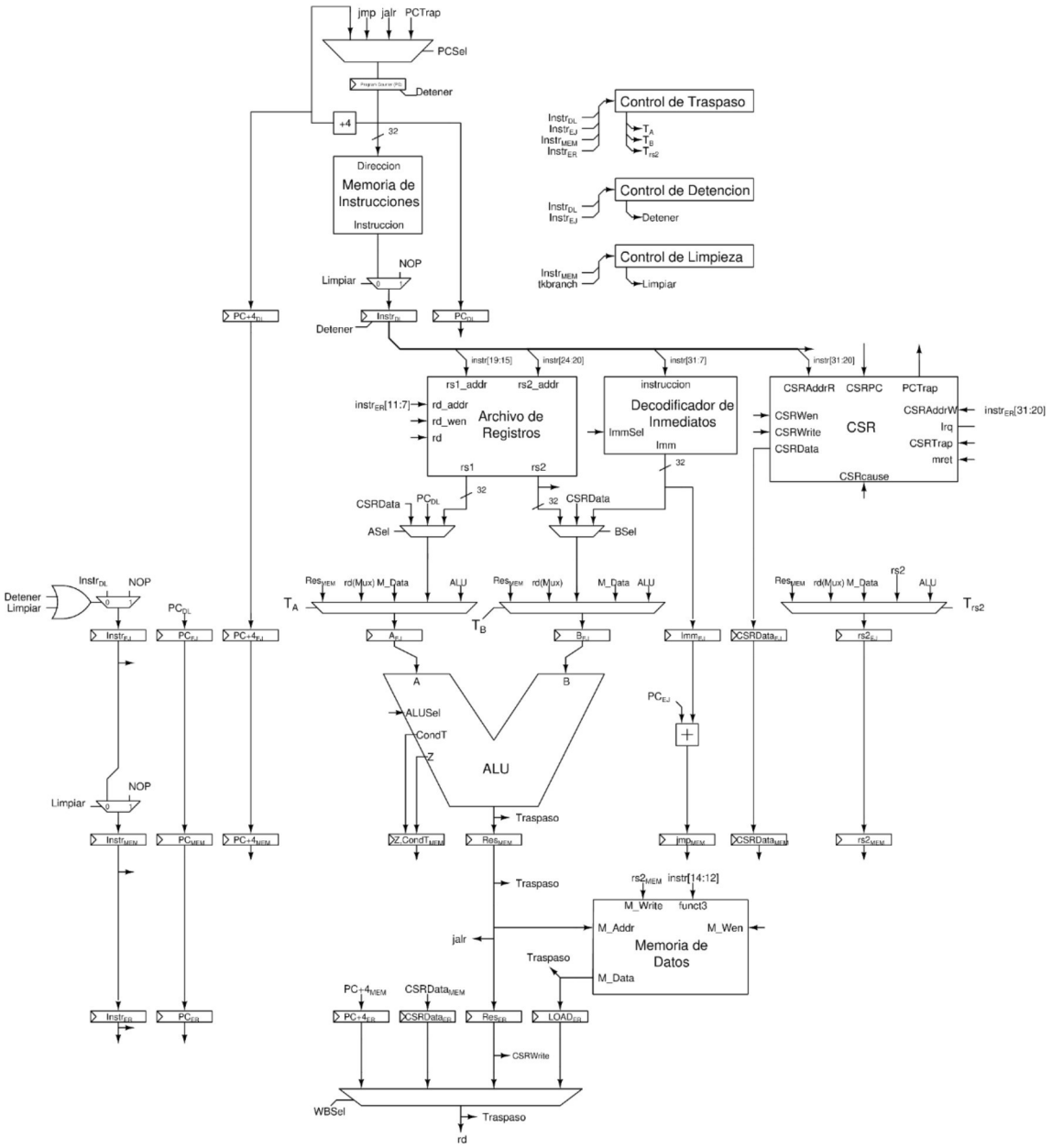


Fuente: elaboración propia, empleando Xcircuit v3.10.

2.5.4. Micro-arquitectura sin riesgos de control

A continuación, se presenta la micro-arquitectura del procesador sin riesgos de control.

Figura 99. **Micro-arquitectura sin riesgos de control**

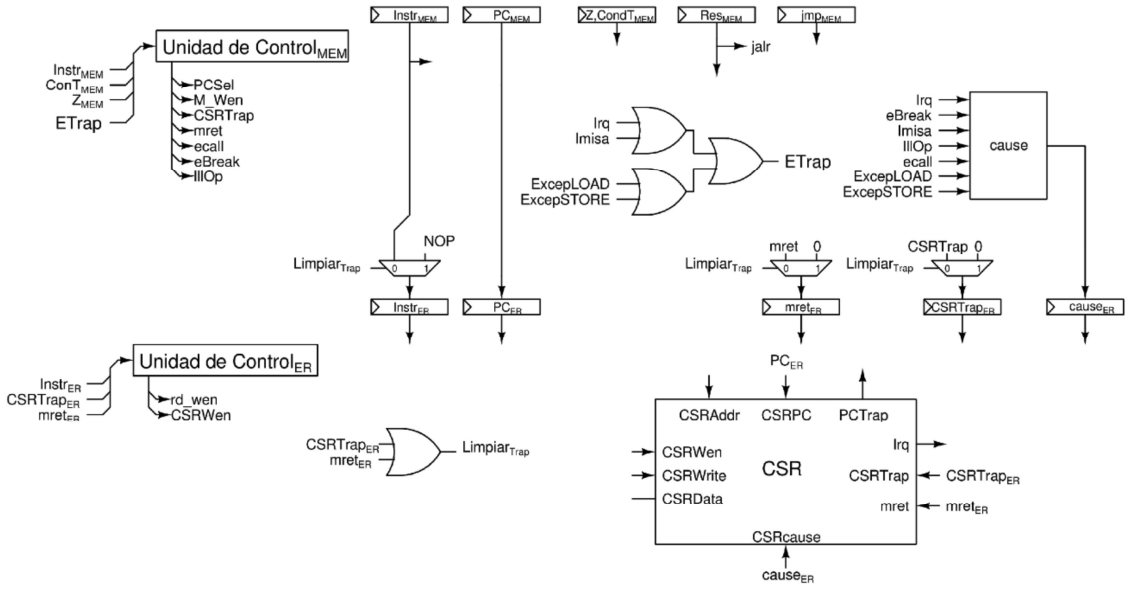


Fuente: elaboración propia, empleando Xcircuit v3.10.

2.6. Excepciones e interrupciones

Debido al tiempo de propagación las interrupciones y excepciones fueron separadas en dos etapas, detección y manejo.

Figura 100. **Detección y manejo de excepciones e interrupciones**



Fuente: elaboración propia, empleando Xcircuit v3.10.

La detección se encuentra en la etapa MEM del *pipeline*, en esta etapa se generan señales que representan la presencia de una interrupción o excepción, y señales utilizadas en el manejo de estas, como *cause*. Al finalizar el ciclo de reloj estas señales son escritas en registros para ser utilizadas en la etapa ER, que es donde se encuentra en manejo de excepciones e interrupciones.

Como se discutió con anterioridad el módulo *CSR* es el encargado del manejo de excepciones e interrupciones, modificando los registros *CSR*

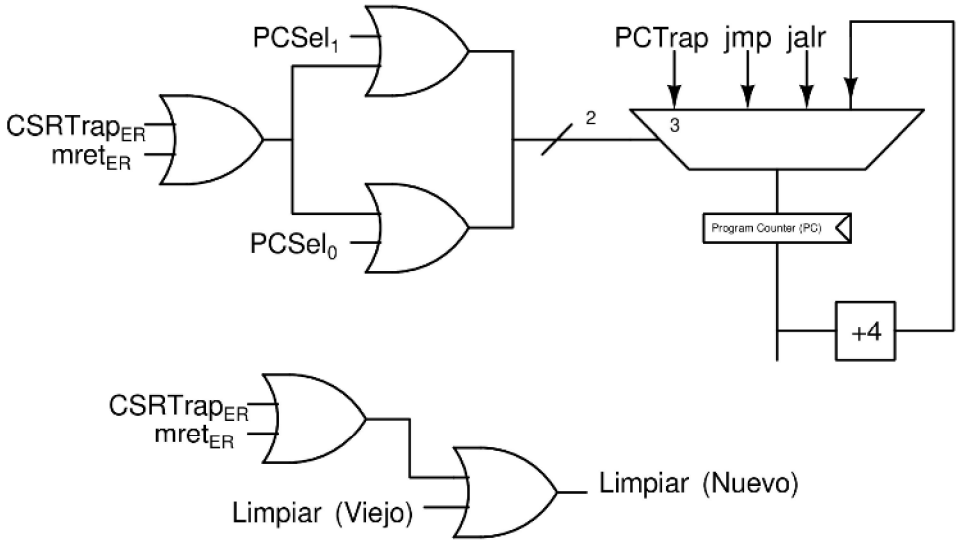
correspondientes y seleccionado la correcta dirección de memoria para ser escrita en *PC*.

Al momento de generarse una interrupción o excepción, el *pipeline* debe ser limpiado en su totalidad. Por tal motivo se agregan tres multiplexores nuevos encargados de la limpieza de la etapa ER, eliminando señales que pueden resultar problemáticas, como $mret_{ER}$ y $CSRTrap_{ER}$, ya que, si ocurren dos excepciones seguidas, como podría ser el caso de una instrucción *ECALL* seguida de una instrucción *EBREAK*, *EBREAK* interrumpiría el manejo de *ecall*, lo que provocaría la omisión de la instrucción *ECALL*.

Por lo tanto, la señal *Limpiar* debe ser actualizada para incluir limpieza de del *pipeline* en los casos de interrupciones y excepciones. Para ello simplemente se agrega una nueva señal $Limpiar_{Trap}$, la cual depende de $mret_{ER}$ y $CSRTrap_{ER}$, y se aplica una función lógica *OR* con la señal *Limpiar* descrita con anterioridad.

Adicionalmente se debe modificar el control del multiplexor de *PC*, debido a que las excepciones e interrupciones poseen prioridad sobre toda instrucción en el procesador, se debe de sobrescribir el valor de *PCSel* para que sea seleccionado *PCTrap* como la próxima instrucción en ejecutarse.

Figura 101. **Cambios en señales Limpiar y *PCSel***



Fuente: elaboración propia, empleando Xcircuit v3.10.

2.7. Criterios de diseño

Se hace la observación, con la nueva micro-arquitectura del procesador, diferentes instrucciones poseen diferentes tiempos de ejecución, dependiendo de la presencia de riesgo de datos o riesgo de control, por ejemplo, al ejecutar una instrucción *JAL* se requiere de una demora de cuatro ciclos para que la siguiente instrucción inicie su ejecución, de igual manera todas las instrucciones con *opcode SYSTEM* demoran cinco ciclos. Por lo tanto, el valor de *CPI* varía con la instrucción a ejecutarse.

Para obtener el *CPI* promedio del procesador se utiliza la siguiente ecuación:

$$CPI_{Promedio} = \sum CPI_{instr} * \%_{instr}$$

Donde:

CPI_{instr} = cantidad de ciclos de retardo por tipo de instrucción.

$\%_{instr}$ = porcentaje del tipo de instrucción que se ejecuta en un programa.

Debido a que las instrucciones *BRANCH* pueden ser ejecutadas de dos maneras diferentes, el *CPI* de las instrucciones *BRANCH* posee dos diferentes valores, uno cuando se toma o no un salto, siendo uno ciclo y cuatro ciclos respectivamente. Por lo tanto, para obtener el *CPI* promedio de una instrucción *BRANCH* se utiliza la siguiente ecuación:

$$CPI_{BRANCH} = SaltosTomados * Costo_{Tomados} + (1 - SaltosTomados) * Costo_{NoTomados}$$

Donde:

SaltosTomados = porcentaje de saltos tomados en la ejecución de un programa.

$Costo_{Tomado}$ = cantidad de ciclos de retardo en un salto tomado.

$Costo_{NoTomado}$ = cantidad de ciclos de retardo en un salto no tomado.

Con la información previa se obtuvieron los siguientes valores de *CPI*:

Tabla XVIII. ***CPI* programas**

Programa	LOAD	CPI_{LOAD}	STORE	CPI_{STORE}	JAL-JALR	$CPI_{JAL-JALR}$	BRANCH	Tomados	$CPI_{Tomados}$	$CPI_{NoTomados}$	SYSTEM	CPI_{SYSTEM}	Registro - Aritmetico	CPI_{R-A}	$CPI_{Promedio}$
dhrystone	19,13 %	1	16,86 %	1	8,14 %	4	14,28 %	35,60 %	4	1	0,11 %	5	41,48 %	1	1,40
median	9,17 %	1	7,74 %	1	7,35 %	4	22,69 %	50,02 %	4	1	0,03 %	5	53,03 %	1	1,56
mt-matmul	11,62 %	1	8,74 %	1	8,43 %	4	19,64 %	46,08 %	4	1	0,04 %	5	51,53 %	1	1,53
mt-vvadd	10,96 %	1	7,13 %	1	3,82 %	4	14,51 %	57,58 %	4	1	0,01 %	5	63,58 %	1	1,37
multiply	5,68 %	1	5,00 %	1	5,48 %	4	27,60 %	55,84 %	4	1	0,02 %	5	56,21 %	1	1,63
qsort	3,81 %	1	3,41 %	1	4,09 %	4	31,69 %	57,09 %	4	1	0,01 %	5	56,99 %	1	1,67
rsort	3,82 %	1	3,45 %	1	4,08 %	4	31,57 %	57,13 %	4	1	0,01 %	5	57,08 %	1	1,66
spmv	3,10 %	1	2,25 %	1	4,79 %	4	21,47 %	71,79 %	4	1	0,00 %	5	68,38 %	1	1,61
towers	21,53 %	1	17,33 %	1	5,81 %	4	18,49 %	51,94 %	4	1	0,26 %	5	36,57 %	1	1,47
exceptions	18,11 %	1	15,80 %	1	7,68 %	4	15,72 %	30,27 %	4	1	0,12 %	5	42,57 %	1	1,38

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Es preferible poseer un valor bajo de *CPI*, lo cual indica una cantidad menor de ciclos necesarios para ejecutar instrucciones, lo cual se traduce a un menor tiempo de ejecución.

La columna Registro - Aritmético es la combinación de las instrucciones que únicamente usan como argumentos registros o argumentos y únicamente modifican el valor de registros, que son las instrucciones con *opcode OP*, *OP-IMM*, *LUI* y *AUIPC*.

Observando la columna de instrucciones de *opcode SYSTEM* se puede apreciar que estas equivalen una fracción mínima de instrucciones que se ejecutan en un programa, por esta razón se decidió detener el procesador para mitigar riesgos de datos, en instrucciones *CSR* ya que esto genera bajo impacto en el rendimiento.

Adicionalmente a *CPI*, se utilizó el valor de Millones de Instrucciones Por Segundo, *MIPS*, el cual divide la velocidad del reloj entre el *CPI* promedio del procesador. Esto da una idea general del rendimiento del procesador ya que toma en cuenta el *CPI* y la frecuencia del reloj.

Tabla XIX. ***MIPS* preliminar de programas**

Programa	$CPI_{Promedio}$	Frecuencia de reloj (MHz)	MIPS
dhrystone	1,40	201,16	143,57
median	1,56	201,16	128,79
mt-matmul	1,53	201,16	131,81
mt-vvadd	1,37	201,16	147,30
multiply	1,63	201,16	123,59
qsort	1,67	201,16	120,76
rsort	1,66	201,16	120,92
spmv	1,61	201,16	125,24
towers	1,47	201,16	136,58
exceptions	1,38	201,16	145,96

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Los valores de frecuencia de reloj fueron obtenidos utilizando la tecnología *osu035* y la micro-arquitectura con *pipeline*, pero sin mitigación de riesgos, es decir la micro-arquitectura preliminar de cinco etapas de *pipeline*. Se obtuvieron estos valores preliminares para poseer una idea general del rendimiento sin necesidad de comprometerse al diseño completo. Por ejemplo, la micro-arquitectura expuesta en este trabajo es la mejor de tres micro-arquitecturas propuestas.

Tabla XX. **Comparación *MIPS* diferentes micro-arquitecturas**

	Propuesta 1			Propuesta 2			Propuesta 3		
Programa	**$CPI_{Promedio}$**	**Frecuencia de reloj (MHz)**	**MIPS**	**$CPI_{Promedio}$**	**Frecuencia de reloj (MHz)**	**MIPS**	**$CPI_{Promedio}$**	**Frecuencia de reloj (MHz)**	**MIPS**
dhrystone	1,27	143,43	113,14	1,40	180,38	128,84	1,40	201,16	143,57
median	1,37	143,43	104,34	1,56	180,38	115,50	1,56	201,16	128,79
mt-matmul	1,35	143,43	106,18	1,53	180,38	118,22	1,53	201,16	131,81
mt-vvadd	1,24	143,43	115,32	1,37	180,38	132,10	1,37	201,16	147,30
multiply	1,42	143,43	101,12	1,63	180,38	110,84	1,63	201,16	123,59
qsort	1,44	143,43	99,34	1,67	180,38	108,29	1,67	201,16	120,76
rsort	1,44	143,43	99,44	1,66	180,38	108,43	1,66	201,16	120,92
spmv	1,40	143,43	102,15	1,61	180,38	112,30	1,61	201,16	125,24
towers	1,32	143,43	108,98	1,47	180,38	122,69	1,47	201,16	136,58
exceptions	1,25	143,43	114,52	1,38	180,38	131,01	1,38	201,16	145,96

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

La primera micro-arquitectura propuesta consistía en realizar los saltos de las instrucciones *JAL*, *JALR* y *BRANCH* en la etapa *EJ*, reduciendo el *CPI* de dichas instrucciones a tres. Cómo se logra observar esta micro-arquitectura posee un *CPI* más bajo, pero en consecuencia su frecuencia de reloj es menor. Esto se debe al elevado tiempo de propagación entre los argumentos de entrada del *ALU*, el resultado Z/CondT, la decodificación de *PCSel* y el multiplexor seleccionador de *PC*.

La segunda micro-arquitectura propuesta consistía en realizar los saltos de las instrucciones *JAL*, *JALR* y *BRANCH* en la etapa MEM, pero la detección y manejo de excepciones e interrupciones es realizado en la etapa MEM. Por lo tanto, las instrucciones *SYSTEM* poseen un *CPI* de cuatro ciclos. Debido al elevado tiempo de propagación entre la generación de la señal *ETrap*, la decodificación del valor *CSRPC* y el multiplexor de *PC*, la velocidad del reloj es menor.

La tercera micro-arquitectura propuesta es la descrita en este trabajo, la cual posee el mejor desempeño preliminar.

Tabla XXI. ***MIPS* de programas, micro-arquitectura completa**

		osu035		osu018	
Programa	**$CPI_{Promedio}$**	**Frecuencia de reloj (MHz)**	**MIPS**	**Frecuencia de reloj (MHz)**	**MIPS**
dhrystone	1,40	189,35	135,15	338,73	241,76
median	1,56	189,35	121,23	338,73	216,87
mt-matmul	1,53	189,35	124,07	338,73	221,95
mt-vvadd	1,37	189,35	138,66	338,73	248,04
multiply	1,63	189,35	116,34	338,73	208,12
qsort	1,67	189,35	113,67	338,73	203,35
rsort	1,66	189,35	113,82	338,73	203,62
spmv	1,61	189,35	117,89	338,73	210,89
towers	1,47	189,35	128,56	338,73	229,99
exceptions	1,38	189,35	137,40	338,73	245,80

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Cómo se logra observar en las columnas correspondientes a la tecnología *osu035*, la cual fue utilizada como referencia en tablas previas, los circuitos adicionales necesarios para mitigar riesgos de datos y control, poseen un costo en términos de velocidad del reloj, ya que sin ellos la frecuencia de reloj estimada es de 201,16 Mhz.

2.8. Predictor de saltos dinámico.

Uno de los factores que atribuyen a la reducción de rendimiento del procesador es el alto *CPI* de instrucciones de transferencia de control ya que estas pueden conformar hasta un poco más de un tercio la de las instrucciones

ejecutadas en un programa. En el caso de instrucciones *BRANCH*, cuando un salto debe ser tomado su *CPI* aumenta debido a la necesidad de limpiar el procesador para corregir la predicción errónea. Por lo tanto, se plantea añadir un predictor de saltos dinámico de dos *bits*.

En la micro-arquitectura presentada hasta el momento, se detalló el funcionamiento de un predictor estático, es decir que su micro-arquitectura está diseñada para realizar un solo tipo de predicción para todas las instrucciones *BRANCH*. Un predictor dinámico es capaz de predecir si un salto debe ser tomado o no, además es capaz de adaptarse a la ejecución de un programa y cambiar dinámicamente la predicción de una instrucción con el objetivo de obtener un alto porcentaje de aciertos en predicciones, en consecuencia, aumentando el rendimiento del procesador.

El encargado de realizar las predicciones es una máquina de estados finita que utiliza dos *bits* de estados.

Figura 102. **Predictor de saltos de dos *bits*, diagrama máquina de estados**

Fuente: NAIR, Ravi, *Optimal 2-bit branch predictors*. p. 701.

Esta máquina posee cuatro estados, 0b00, 0b01, 0b10 y 0b11 representado por cuatro círculos en orden izquierda a derecha, esta máquina de estados predice que una instrucción *BRANCH* debe tomar un salto si se encuentra en los

círculos en negrilla, es decir estados 0b10 y 0b11, en caso contrario si se encuentran en los estados representados por los círculos sin negrilla, estados 0b00 y 0b01, la máquina predecirá que no debe tomarse un salto. Esta máquina de estados actualiza su estado al momento que el procesador obtiene el resultado de la condición y decide si debe tomarse o no un salto, si debe tomarse el salto su memoria de estado cambiará al siguiente estado en el diagrama, representado por flechas en negrilla, si no debe tomarse un salto su memoria de estado cambiará al estado anterior en el diagrama, representado por flechas punteadas.

Por ejemplo, si la memoria de estado se encuentra en el estado 0b01 y el procesador concluye que debe tomarse el salto, su estado será actualizado a 0b10, en caso contrario su estado será actualizado a 0b00.

Se determina que esta máquina de estados finita posee un buen desempeño en todas las aplicaciones en las que fue sometida, por esta razón es la utilizada en este trabajo[15].

Debido a que se puede predecir tomar saltos, se debe conocer de antemano la dirección destino del salto, por lo tanto, adicional a la máquina de estados se requiere de una memoria la cual almacena la dirección objetivo de cada instrucción *BRANCH* y su estado.

2.8.1. Implementación

El funcionamiento del predictor es separado en dos etapas, el circuito de predicción y el circuito de actualización de estado.

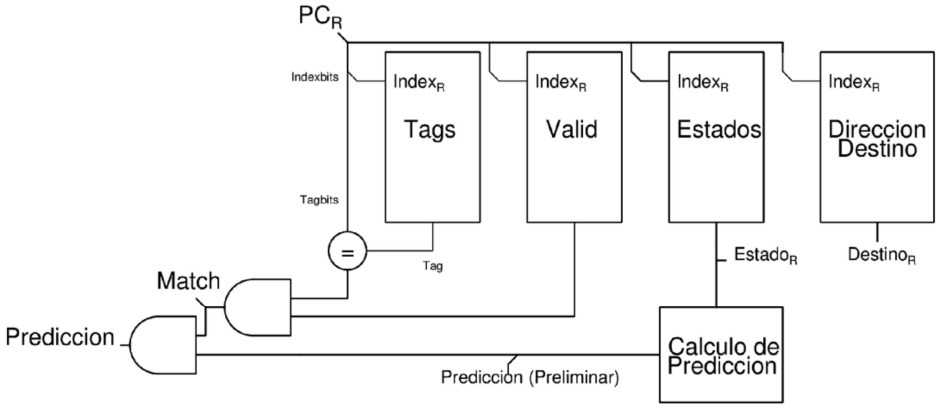
[15] NAIR, Ravi. *Optimal 2-bit branch predictors*. p. 701.

2.8.1.1. Predicción

El circuito de predicción es el encargado de obtener el estado actual y utilizar su valor para predecir si un salto debe ser tomado o no, este circuito opera en la etapa OB del procesador. Debido a que en esta etapa no se ha decodificado la instrucción el circuito debe de operar únicamente con la dirección de memoria de la instrucción. Esto puede presentar un problema, ¿Cómo distinguir una instrucción de transferencia de control del resto de instrucciones?

En primer lugar, se debe aclarar que el predictor únicamente puede mantener el registro de estado de una cantidad finita de instrucciones, la cual es determinada por la capacidad de memoria del predictor.

Figura 103. **Predicción**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Utilizando los *bits* 2 a $\log_2(N)$ de PC_R, *Indexbits*, donde N es la cantidad de instrucciones que puede almacenar el predictor, se indexa en la memoria de *tags* para extraer el *tag* almacenado el cual es comparado con los *bits* $\log_2(N)+1$ a 31 de PC_R, *Tagbits*, si ambos son iguales se sabe que la dirección de memoria en

PC_R corresponde a una instrucción de transferencia de control. Por lo tanto, en la memoria de *tags*, únicamente se guarda el *tag* de instrucciones de transferencia de control.

El principio de este circuito es almacenar la dirección de memoria de instrucciones de transferencia de control, con el objetivo que la siguiente vez que se acceda a la misma dirección de memoria se sepa de antemano que es una instrucción de transferencia de control. Con el objetivo de economizar recursos y no realizar comparaciones de direcciones de treinta y dos *bits*, una porción de los *bits* de la dirección de memoria, *Indexbits*, es utilizada para indexar a una única localidad en memoria. En ella se almacena el resto de *bits* que no fueron utilizados para indexar memoria, *Tagbits*.

Existe la posibilidad que exista un falso positivo en la comparación entre direcciones, por ejemplo, al iniciar el procesador la memoria puede estar escrita con valores aleatorios que coinciden con la dirección a comparar. Por lo tanto, se añade un *bit* de validez que es escrito con el valor de uno solamente cuando se actualice el estado de una correcta instrucción de transferencia de control dentro del predictor.

En la máquina de estados a implementar se debe predecir tomar un salto en los estados 0b10 y 0b11, y se debe predecir no tomar un salto en los estados 0b00 y 0b01. Si se decide representar tomar un salto con el valor de uno, se logra observar que, bajo este criterio, el cálculo de predicción simplemente es el valor del *bit* más significativo del estado.

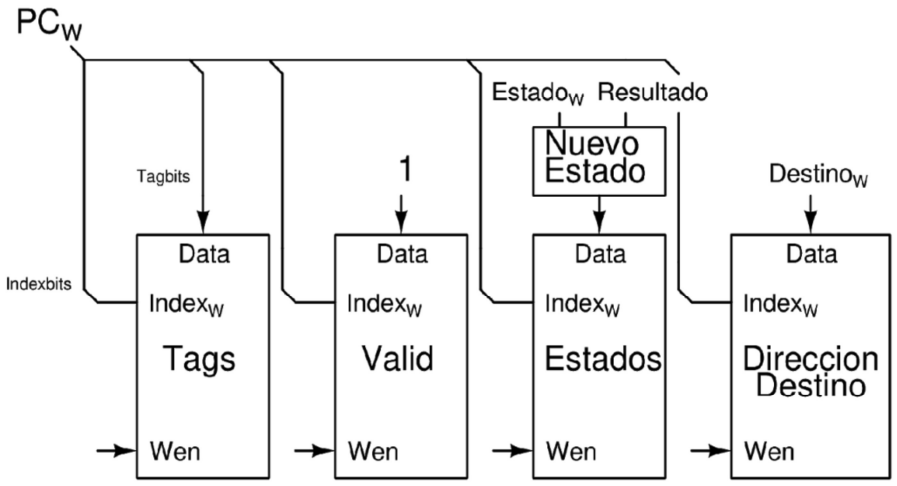
Por lo mencionado anteriormente, el predictor funciona gracias a datos almacenados previamente, por lo tanto, solo es funcional para instrucciones de transferencia de control que se ejecutan más de una vez, adicionalmente la

primera vez que una instrucción de transferencia de control es ejecutada el predictor no tomará efecto.

2.8.1.2. Actualización de estado

El estado es actualizado cuando se obtiene el resultado de la condición, es decir cuando el procesador decide tomar o no tomar un salto, este resultado es obtenido en la etapa MEM. Adicionalmente en esta etapa se escribe en memoria los valores de *tag*, dirección destino y *bit* de validez.

Figura 104. **Actualización de estado**



Fuente: elaboración propia, empleando Xcircuit v3.10.

El nuevo estado depende de las señales *Estado*$_W$ y *Resultado*, que es el estado al momento que se realizó la predicción y el resultado que el procesador obtuvo respecto a realizar un salto o no, respectivamente.

Tabla XXII. **Tabla de verdad señal NuevoEstado**

Resultado	$Estado_W$	Nuevo Estado
0	00	00
0	01	00
0	10	01
0	11	10
1	00	01
1	01	10
1	10	11
1	11	11

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Figura 105. **Módulo Predicción de Saltos**

Fuente: elaboración propia, empleando Xcircuit v3.10.

2.8.2. Modificaciones en *datapath*

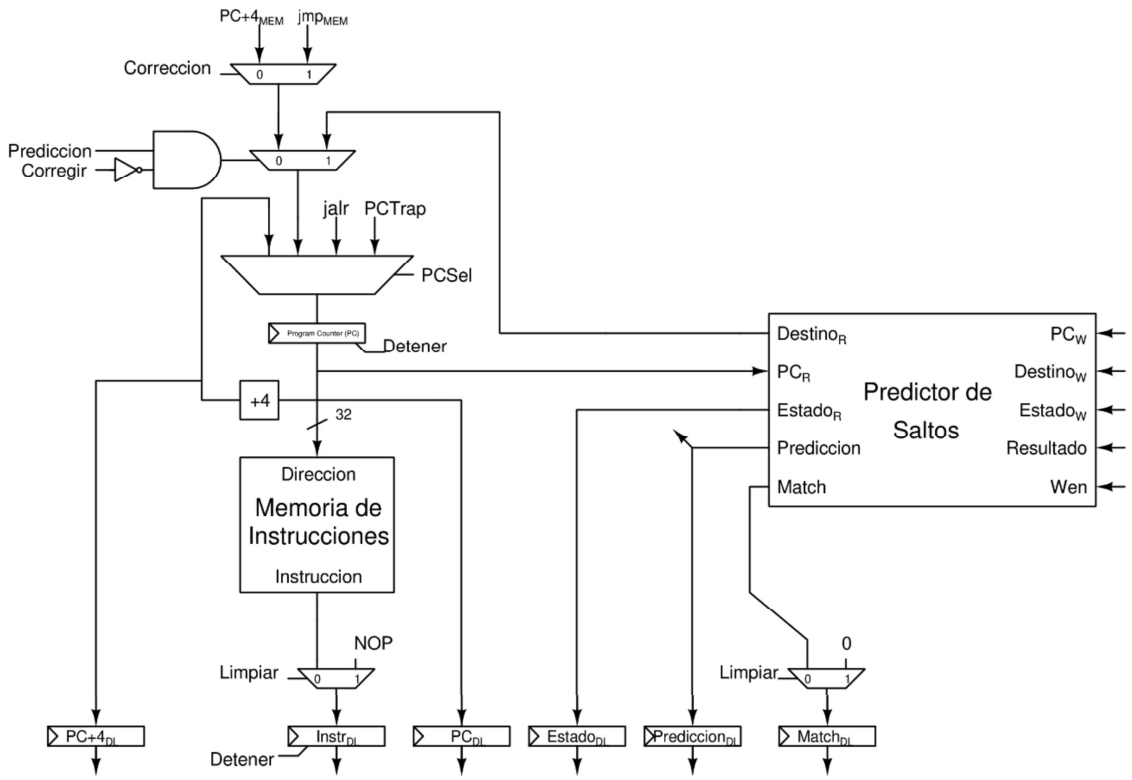
Adicionalmente a agregar el módulo de predicción de saltos, se debe modificar el *datapath* para comprobar la veracidad de las predicciones, corregir predicciones erróneas y modificar el traspaso de datos.

Los cambios presentados a continuación únicamente afectan a instrucciones *JAL* y *BRANCH*, ya que el predictor de saltos presentado únicamente funciona para saltos con destino estático, como es el caso de las instrucciones *JAL* y *BRANCH*. Es decir, la dirección destino de dichas instrucciones es constante durante la ejecución de un programa, ya que esta se encuentra codificada en el formato de instrucción. En cambio, la dirección destino de instrucciones *JALR* varía durante la ejecución de un programa como consecuencia a su dependencia de valores de registros.

2.8.2.1. Cambios en etapa OB

A continuación, se presenta la figura 106, la cual posee los cambios realizados en la etapa OB del *datapath*.

Figura 106. **Cambios en etapa OB**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Se agregan registros $Estado_{DL}$, $Prediccion_{DL}$ y $Match_{DL}$, los cuales almacenan los valores de las señales correspondientes del predictor de saltos, con el objetivo de transportar estos valores hasta la etapa MEM, en donde son utilizados para comprobación y corrección. Por lo tanto, son agregados los registros $Estado_{EJ}$, $Estado_{MEM}$, $Prediccion_{EJ}$, $Prediccion_{MEM}$, $Match_{EJ}$ y $Match_{MEM}$.

Se modifica la manera en que es escrito el registro *PC*, cuando se predice que debe tomarse un salto, la señal *Predicción* selecciona el valor $Destino_R$, que es la dirección objetivo del salto, para ser escrito en *PC*. Por lo tanto, *PCSel* debe

seleccionar la entrada correspondiente a la predicción de saltos. En caso contrario *PCSel* seleccionará *PC*+4 a ser escrito en *PC*.

Cuando se deba corregir una predicción errónea, la señal *Corrección*, es utilizada para seleccionar el valor a corregir, si se predijo erróneamente que debe tomarse un salto, se debe corregir seleccionando el valor *PC*+4 de la instrucción predicha, es decir se corrige con el valor de *PC*+4 de la etapa MEM. Sí se predijo erróneamente que no debe tomarse un salto, se debe corregir seleccionando el valor destino del salto que es la señal jmp_{MEM}.

Dado a que existe la posibilidad que al mismo tiempo que debe ser corregida una predicción errónea el predictor de saltos intente realizar un nuevo salto, la corrección posee prioridad ya que con ella se mantiene la correcta ejecución de programas, se utiliza la señal *Corregir* para deshabilitar la señal *Predicción*, lo cual permite que *PC* sea escrito con el valor correcto al momento de realizar una corrección.

En consecuencia, a las nuevas señales agregadas por el predictor de saltos, se cambia en la forma que la señal *PCSel* es calculada.

Tabla XXIII. **Tabla de verdad señal *PCSel***

$Limpiar_{Trap}$	Corregir	Predicción	$isJALR_{MEM}$	PCSel
0	1	-	0	Jmp_{PS}
0	0	1	0	Jmp_{PS}
0	0	-	1	jalr
0	0	0	0	PC+4
1	-	-	-	PCTrap

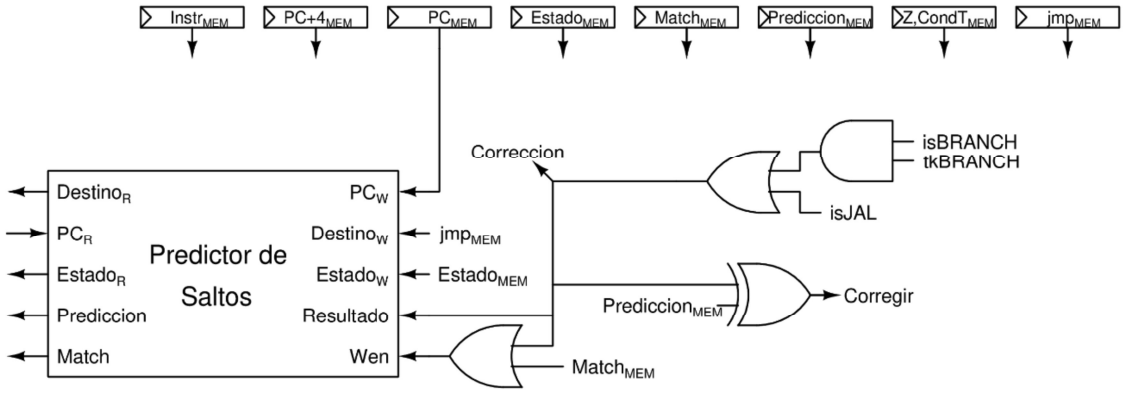
Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Jmp$_{PS}$ es la entrada del multiplexor que posee la señal correspondiente al predictor de saltos.

2.8.2.2. Cambios en etapa MEM

A continuación, se presenta la figura 107, la cual contiene los cambios realizados en el *datapath*.

Figura 107. **Cambios en etapa MEM**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Si la conclusión del procesador respecto a un salto, representado por la señal *Corrección*, difiere con la predicción realizada se debe corregir el flujo del programa, para ello se utiliza una compuerta *XOR* la cual posee el valor de uno cuando sus entradas son diferentes, a la cual es conectada la predicción realizada, *Predicción*$_{MEM}$, y la conclusión del procesador, *Corrección*. El resultado es la señal *Corregir*.

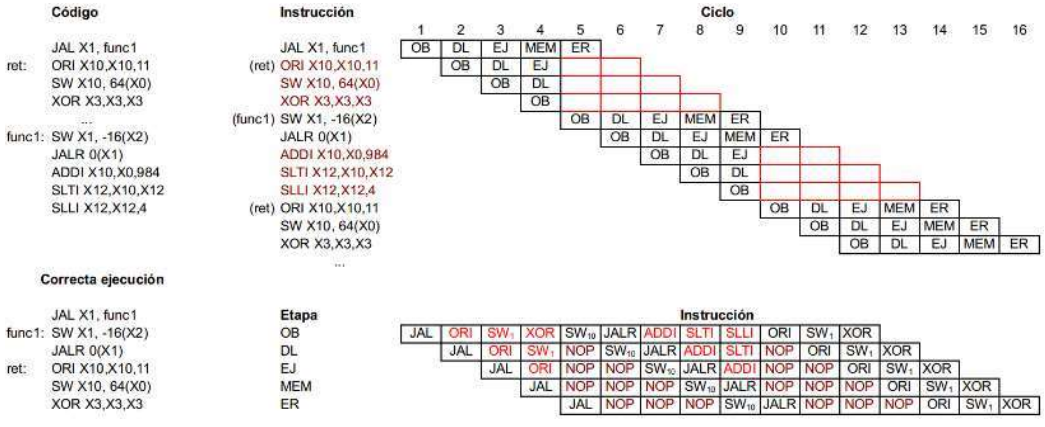
Se habilita la escritura en el módulo, cuando se debe actualizar el estado en una instrucción presente en el predictor de saltos, representado por la señal

Match$_{MEM}$ o cuando una instrucción deba tomar un salto. Se utiliza este enfoque con el objetivo de utilizar eficientemente el espacio de almacenamiento, a diferencia de almacenar todas las instrucciones de transferencia de control, únicamente se almacenan instrucciones que realizan saltos, obviando instrucciones *BRANCH* que no realizan saltos en la ejecución del programa.

2.8.2.3. Cambios en traspaso de datos

Gracias al predictor de saltos las instrucciones *JAL* pueden ser ejecutadas con retardo de un ciclo, pero esto puede generar riesgos en la ejecución de programas. Por ejemplo, se introduce un riesgo de datos al ejecutar una instrucción *JAL* seguido de una instrucción que dependa del valor del registro rd, un ejemplo es llamada y retorno de una subrutina, por lo general se llama a una subrutina utilizando la instrucción JAL la cual almacena el valor de *PC*+4 en el registro *ra*/*x1*, al retornar de una subrutina se utiliza la instrucción *JALR* para modificar el valor de *PC* con el valor del registro *ra*/*x1*.

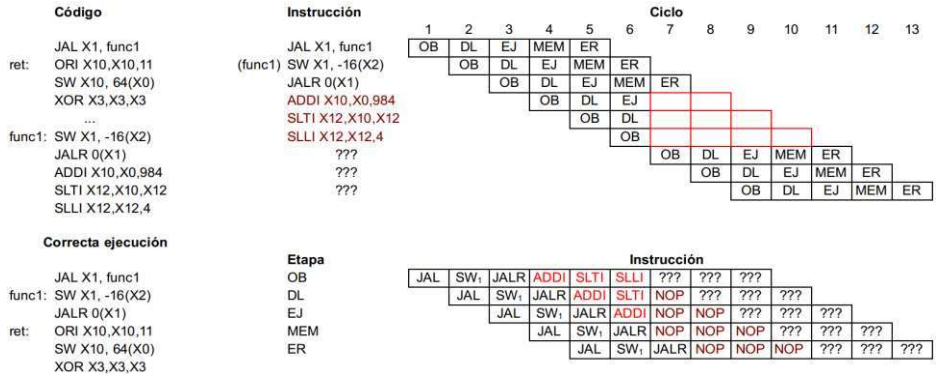
Figura 108. **Ejecución instrucción *JAL* sin predicción de saltos**



Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Debido a que la instrucción *JAL* posee un retraso de cuatro ciclos, esto le permite escribir en el archivo de registros antes que la siguiente instrucción pase por la etapa DL. Por este motivo no se agregó el valor de *PC*+4, el valor que la instrucción *JAL* escribe en el archivo de registros, en el circuito de traspaso de datos.

Figura 109. **Ejecución instrucción *JAL* con predicción de saltos**



Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Cómo se logra observar si el salto es predicho correctamente *JAL* posee retraso de un ciclo, por lo tanto, el valor de *PC*+4 no es escrito en el archivo de registros al momento que la instrucción *JALR* solicita este valor en la etapa DL, causado un riesgo de datos, en este ejemplo el riesgo de datos provoca que la dirección destino de la instrucción *JALR* sea generada erróneamente, perdiendo por completo el control de ejecución del programa. Por tal razón se modificó el circuito de traspaso de datos para agregar el valor de *PC*+4 de la etapa EJ y MEM, en los multiplexores de traspaso y se cambió el controlador de traspaso para cumplir con las siguientes tablas de verdad:

Tabla XXIV. **Tabla de verdad control de traspaso A_{Reg}**

noJAL & noLUI & noAUIPC & noCSRI (DL)	noBRANCH & noSTORE (EJ)	$Rd_{EJ} = RS1_{DL}$ & Rd_{EJ} =/ 0	noBRANCH & noSTORE (MEM)	$Rd_{MEM} = RS1_{DL}$ & Rd_{MEM} =/ 0	noBRANCH & noSTORE (ER)	$Rd_{ER} = RS1_{DL}$ & Rd_{ER} =/ 0	$isLOAD_{MEM}$	$isJAL_{EJ}$	$isJAL_{MEM}$	Traspaso
1	1	1	-	-	-	-	-	0	-	ALU
1	1	1	-	-	-	-	-	1	-	$PC+4_{EJ}$
1	0	-	0	0	0	0	-	-	-	No
1	-	0	0	0	0	0	-	-	-	No
1	0	0	1	1	-	-	1	-	0	M_DATA
1	0	0	1	1	-	-	0	-	1	$PC+4_{MEM}$
1	0	0	1	1	-	-	0	-	0	Res_{MEM}
1	0	0	0	-	0	0	-	-	-	No
1	0	0	-	0	0	0	-	-	-	No
1	0	0	0	0	1	1	-	-	-	Rd(Mux)
1	0	0	0	0	0	-	-	-	-	No
1	0	0	0	0	-	0	-	-	-	No
0	-	-	-	-	-	-	-	-	-	No

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Tabla XXV. **Tabla de verdad control de traspaso B_{Reg}**

isBRANCH | isOP (DL)	noBRANCH & noSTORE (EJ)	$Rd_{EJ} = RS2_{DL}$ & Rd_{EJ} =/ 0	noBRANCH & noSTORE (MEM)	$Rd_{MEM} = RS2_{DL}$ & Rd_{MEM} =/ 0	noBRANCH & noSTORE (ER)	$Rd_{ER} = RS2_{DL}$ & Rd_{ER} =/ 0	$isLOAD_{MEM}$	$isJAL_{EJ}$	$isJAL_{MEM}$	Traspaso
1	1	1	-	-	-	-	-	0	-	ALU
1	1	1	-	-	-	-	-	1	-	$PC+4_{EJ}$
1	0	-	0	0	0	0	-	-	-	No
1	-	0	0	0	0	0	-	-	-	No
1	0	0	1	1	-	-	1	-	0	M_DATA
1	0	0	1	1	-	-	0	-	1	$PC+4_{MEM}$
1	0	0	1	1	-	-	0	-	0	Res_{MEM}
1	0	0	0	-	0	0	-	-	-	No
1	0	0	-	0	0	0	-	-	-	No
1	0	0	0	0	1	1	-	-	-	Rd(Mux)
1	0	0	0	0	0	-	-	-	-	No
1	0	0	0	0	-	0	-	-	-	No
0	-	-	-	-	-	-	-	-	-	No

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Tabla XXVI. **Tabla de verdad control de traspaso *rs2$_{EJ}$***

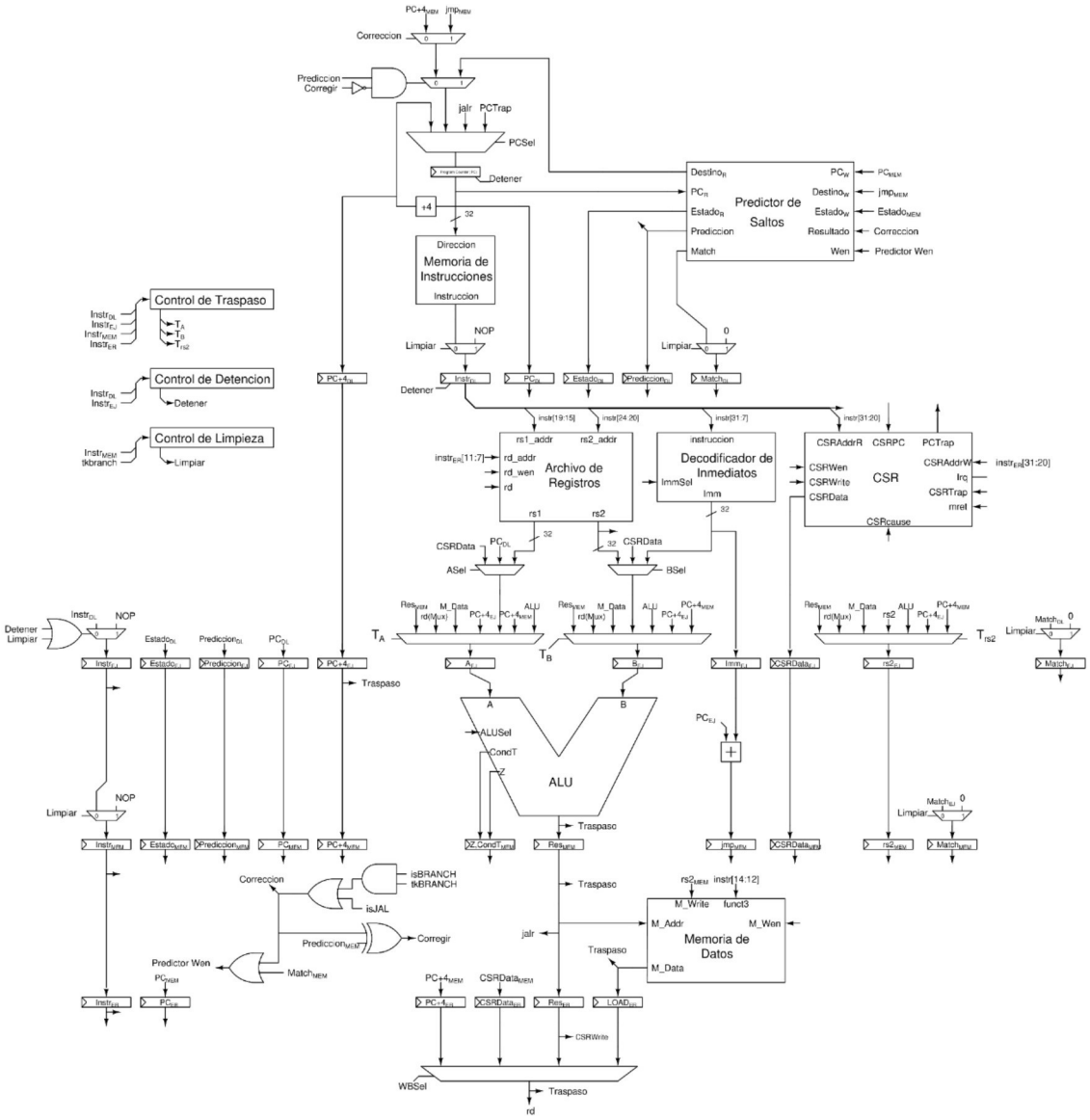
isSTORE (DL)	noBRANCH & noSTORE (EJ)	$Rd_{EJ} = RS2_{DL}$ & Rd_{EJ} =/ 0	noBRANCH & noSTORE (MEM)	$Rd_{MEM} = RS2_{DL}$ & Rd_{MEM} =/ 0	noBRANCH & noSTORE (ER)	$Rd_{ER} = RS2_{DL}$ & Rd_{ER} =/ 0	$isLOAD_{MEM}$	$isJAL_{EJ}$	$isJAL_{MEM}$	Traspaso
1	1	1	-	-	-	-	-	0	-	ALU
1	1	1	-	-	-	-	-	1	-	$PC+4_{EJ}$
1	0	-	0	0	0	0	-	-	-	No
1	-	0	0	0	0	0	-	-	-	No
1	0	0	1	1	-	-	1	-	0	M_DATA
1	0	0	1	1	-	-	0	-	1	$PC+4_{MEM}$
1	0	0	1	1	-	-	0	-	0	Res_{MEM}
1	0	0	0	-	0	0	-	-	-	No
1	0	0	-	0	0	0	-	-	-	No
1	0	0	0	0	1	1	-	-	-	Rd(Mux)
1	0	0	0	0	0	-	-	-	-	No
1	0	0	0	0	-	0	-	-	-	No
0	-	-	-	-	-	-	-	-	-	No

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

2.9. Procesador RISC-V con cinco etapas de pipeline y predicción dinámica de saltos

A continuación, se presenta la micro-arquitectura de un procesador RV32Il con cinco etapas de *pipeline* y predicción dinámica de saltos.

Figura 110. **Procesador RISC-V con cinco etapas de *pipeline* y predicción dinámica de saltos**



Fuente: elaboración propia, empleando Xcircuit v3.10.

2.10. Resultados

El código fuente VHDL de las diferentes micro-arquitecturas puede ser obtenido[16].

2.10.1. Efectividad de predicción dinámica de saltos

La efectividad de un predictor de saltos de dos *bits*, es decir la precisión en sus predicciones, depende de dos parámetros, la máquina de estados, en este trabajo se implementó la máquina de estados que presentó mejores resultados en usos de propósito general[17], por tal razón, no se discutirá su impacto.

El segundo parámetro es el tamaño o la cantidad de instrucciones que puede almacenar, para medir su impacto se creó un simulador en el lenguaje de programación Python, el cual toma rastros de ejecución de instrucciones de transferencia de control, los cuales poseen información como la dirección de memoria de la instrucción, el tipo de instrucción y el resultado del procesador respecto a tomar o no un salto. El programa toma estos datos y realiza una predicción de la misma manera que fue descrita en este trabajo. Cuando la predicción y el resultado del procesador discrepan se toma un registro del número de fallos para obtener la precisión del predictor.

[16] SIERRA, Ottoniel. *RV32I/hardware.* www.github.com/OASM/RV32I/tree/main/hardware. Consulta: agosto 2021.

[17] NAIR, Ravi. *Optimal 2-bit branch predictors*. p. 701.

Figura 111. **Código, obtención de rastros de ejecución, VHDL**

```
BRANCHS: process(clk)
        file printoutput: text open write_mode is "brach.txt";
        variable addr: integer;
        variable txt : line;
        variable opcode: integer;
    begin

        if rising_edge(clk) then
            addr := to_integer(unsigned(imem_addr));
            opcode := to_integer(unsigned(imem_data(6 downto 0)));
            case opcode is
                when iOPCODE_BRANCH =>
                    write(txt,string'("0 "));
                    write(txt,addr);
                    if pcsel = PC_JMP then
                        write(txt,string'(" 1"));
                    else
                        write(txt,string'(" 0"));
                    end if;
                    writeline(printoutput,txt);
                when iOPCODE_JAL =>
                    write(txt,string'("1 "));
                    write(txt,addr);
                    writeline(printoutput,txt);
                when others => NULL;
            end case;
        end if;
    end process;
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 112. **Código, clase predictor de saltos, Python**

```
from math import log2
class BrachPredict:
    def __init__(self,size,fsm,initialstate=0):
        self.fsm = fsm
        self.indexbits = int(log2(size))
        self.indexmask = size-1
        self.currentstate = [initialstate]*size
        self.initialstate = initialstate
        self.tags = [0]*size
        self.misses = 0
        self.jalmisses = 0
        self.branchmisses = 0

    #Metodo utilizado para obtener predicción y actualizar estados
    #isjal: utilizado para diferenciar entre instrucciones BRANCH Y JAL
    #addr: La direccion de memoria de la instruccion BRANCH/JAL
    #taken: Resultado del procesador. True, si el procesador decide tomar salto.
    def predict(self,isjal,addr,taken):
        tableindex = self.indexmask & (addr>>2)
        tag = addr >> (self.indexbits+2)
        state = self.currentstate[tableindex]

        if self.tags[tableindex] == tag:
            prediction, nextstate = self.fsm[state]
            if isjal:
                prediction = 1
        else:
            prediction = 0
            nextstate = self.fsm[state][1]
        #si la prediccion es erronea registrar el numero de fallos
        if prediction != taken:
            self.misses += 1
            if isjal:
                self.jalmisses+=1
            else:
                self.branchmisses+=1
        if taken:
            self.tags[tableindex] = tag
            self.currentstate[tableindex] = nextstate[taken]

    def getAccuracy(self,Total,branches):
        jals = Total - branches
        tacc = (1-(self.misses/Total))*100
        jacc = (1-(self.jalmisses/jals))*100
        bacc = (1-(self.branchmisses/branches))*100
        return tacc,jacc,bacc

    #Metodo para desplegar los resultados de la configuración
    def out(self,Total,branches):
        tacc,jacc,bacc = self.getAccuracy(Total,branches)
        end = '\n' if self.indexmask+1 == 8192 else ''
        return f'\t{self.indexmask+1}:\t\tTotal: {tacc:.4}%\t\
        Jal: {jacc:.4}%\tBranchs: {bacc:.4}%{end}'
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 113. **Código, ejecución principal de simulador, Python**

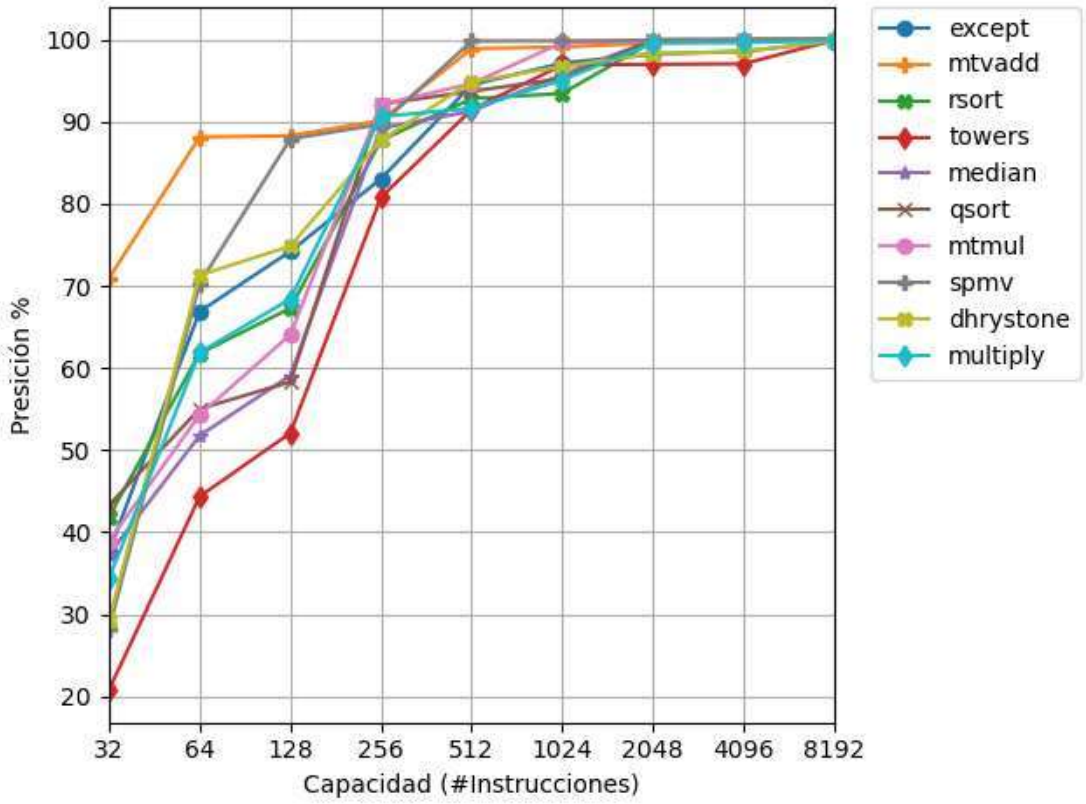
```
#fms[0] = (predicción,(NuevoEstado no tomado,NuevoEstado tomado))
fsms = [((0,(0,1)),(0,(0,2)),(1,(1,3)),(1,(2,3)))]
sizes = (32,64,128,256,512,1024,2048,4096,8192)

for fname in os.listdir():
    if fname.endswith(".txt"):
        info={"total":0,"taken":0,"branchs": 0,'name':fname.replace(".txt",'')}
        predictors = []
        for fsm in fsms:
            for size in sizes:
                predictors.append(BrachPredict(size,fsm,int(argv[1])))
#formato archivo con rastros de ejecucion: isjal addr, taken. Ej (0 2640 0)
        with open(fname,'r') as f:
            for line in f:
                isjal, addr, *taken = line.split(" ")
                info['total']+= 1
                addr = int(addr.strip())
                isjal = int(isjal)
                if taken:
                    taken = int(taken[0].strip())
                else:
                    taken = 1

                if not isjal:
                    info['branchs']+= 1
                    if taken:
                        info['taken']+= 1
                for predictor in predictors:
                    predictor.predict(isjal,addr,taken)

        print(info['name'])
        for predictor in predictors:
            print(predictor.out(info['total'],info['branchs']))
```

Fuente: elaboración propia, empleando Geany 1.37.1.

El código fuente completo y los datos generados pueden ser obtenidos[18].

[18] SIERRA, Ottoniel. *RV32I/software/branch*. https://github.com/oasm95/RV32I/tree/main/software/branch. Consulta agosto 2021.

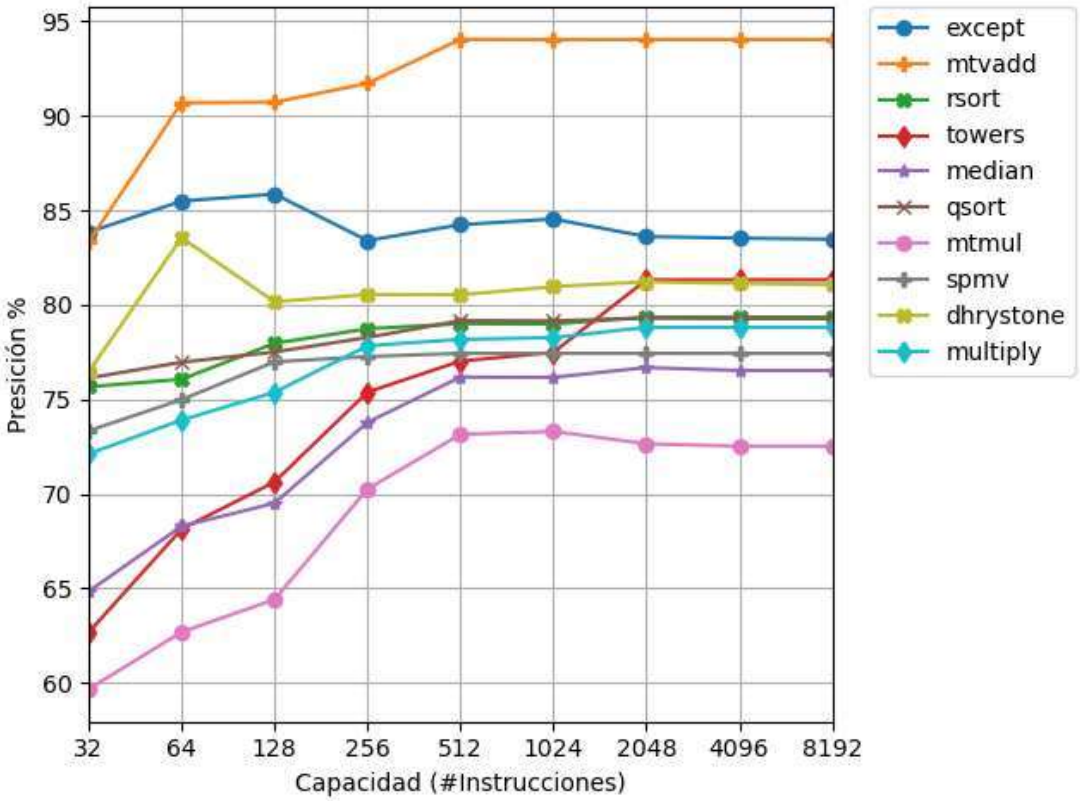
Figura 114. **Capacidad vs., precisión, instrucciones *JAL***



Fuente: elaboración propia, empleando Python 3.

Como se puede observar en las instrucciones *JAL* al aumentar la capacidad del predictor, aumenta la precisión hasta obtener valores cercanos al 100 %. Esto se debe a que las instrucciones *JAL* siempre toman saltos, por lo tanto, no es necesario predecir tomar un salto o no, en este caso el predictor funciona como un registro de instrucciones previamente ejecutadas. Por lo tanto, al aumentar la capacidad del predictor en algún punto poseerá la capacidad para almacenar la mayoría de las instrucciones *JAL* ejecutadas, obteniendo una precisión cercana al 100 %.

Figura 115. **Capacidad vs., precisión, instrucciones *BRANCH***



Fuente: elaboración propia, empleando Python 3.

A pesar de que la capacidad del predictor juega un papel importante en la precisión de las predicciones, en cierto punto a pesar de aumentar la capacidad del predictor la precisión se mantiene constante, representado por la forma de líneas horizontales, en este punto se muestra las limitaciones de un predictor de dos *bits*, en instrucciones de salto condicional *BRANCH*. Este tipo de predictor es limitado a saltos condicionales con patrones simples, por ejemplo, la condición de un ciclo *for* que recorre un vector de cien posiciones, en noventa y nueve ejecuciones de la instrucción, el salto debe ser tomado y solo al finalizar el ciclo este salto no debe ser tomado. Adicionalmente, este tipo de predictor sufre en su

precisión en saltos condicionales que dependen de datos generados de una forma dinámica, por ejemplo, un algoritmo para ordenar valores en un arreglo, aunque se ejecute la misma porción de código, en general los datos a ser ordenados son diferentes en cada ejecución.

Aunque puede ser tentador implementar un predictor con una capacidad relativamente grande, hay que tomar en consideración las compensaciones que se deben asumir al construir una memoria de alta capacidad. Primero, se debe estar dispuesto a utilizar una mayor cantidad de recursos, para ser más precisos debido al requisito de utilizar capacidad con valores en potencia de dos, si se desea aumentar la capacidad como mínimo se debe duplicar la cantidad de recursos. Segundo, se debe considerar como la capacidad de una memoria afecta la velocidad en la que pueden ser extraídos los datos. Para ello se utilizó la herramienta CACTI.

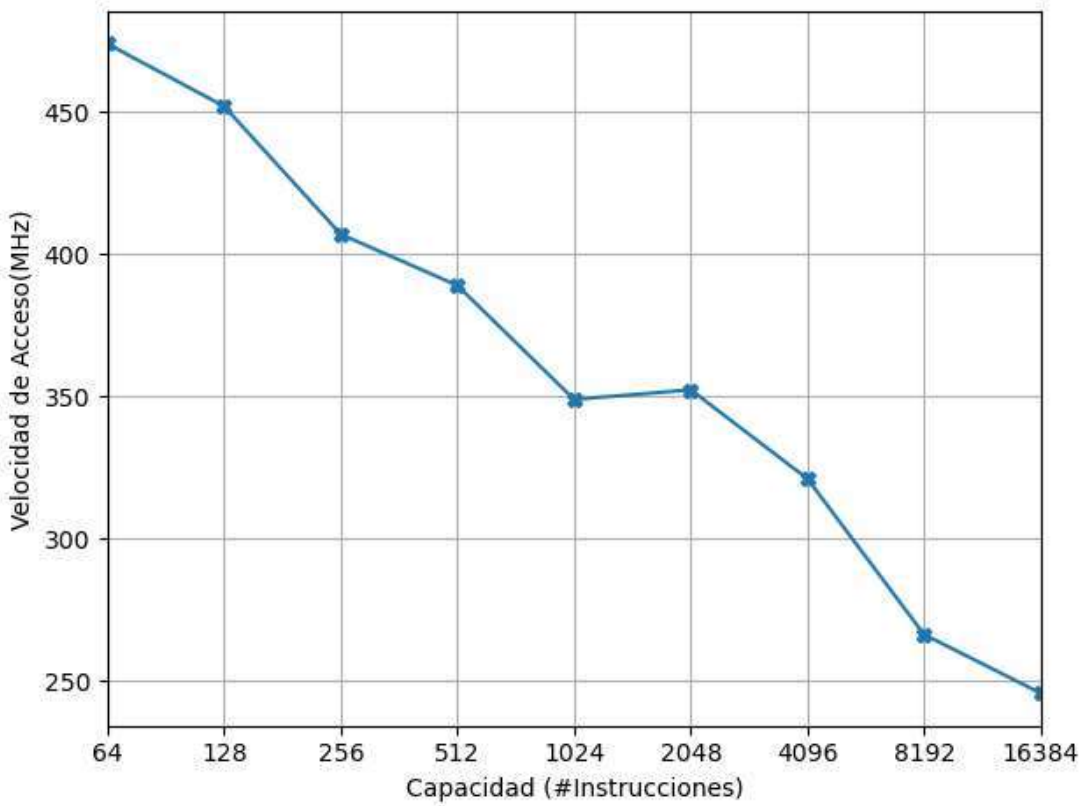
> CACTI es un modelo integrado de tiempo de acceso a memoria y caché, tiempo de ciclo, área, fugas y energía dinámica. Al integrar todos estos modelos juntos, los usuarios pueden tener la confianza de que las compensaciones entre tiempo, potencia y área se basan todas en los mismos supuestos y, por lo tanto, son mutuamente consistentes. CACTI está diseñado para que lo utilicen arquitectos de computadora para comprender mejor las compensaciones de rendimiento inherentes a las organizaciones de sistemas de memoria[19].

La cual puede obtenerse en HEWLETTPACKARD[20].

[19] HP LABS. *CACTI.* https://hpl.hp.com/research/cacti/. Consulta: enero 2021.

[20] HEWLETTPACKARD. *cacti*. https://github.com/HewlettPackard/cacti. Consulta: enero 2021.

Figura 116. **Capacidad vs., velocidad de acceso, tecnología 180 nm**



Fuente: elaboración propia, empleando CACTI 7 y Python 3.

Se observa una tendencia de disminuir la velocidad de acceso en una memoria al aumentar su capacidad. Estos datos son utilizados como aproximaciones y valores esperados, para determinar la configuración a ser implementada.

2.10.2. Tiempos de ejecución

Respecto a la configuración del predictor de saltos se implementó con una capacidad de doscientas cincuenta y seis entradas, con una velocidad de acceso

de aproximadamente 400 Mhz para la tecnología 180 nm similar a *osu018*. Debido a que esta velocidad únicamente toma en consideración la extracción de datos, se dejó un margen suficientemente ancho para acoplar los circuitos de comparación, cálculo de predicción y validación, con el objetivo de no afectar la velocidad de ejecución del procesador.

Tabla XXVII. **Tiempo de ejecución en diferentes micro-arquitecturas**

Programa	Control			Pipeline de cinco etapas			Pipeline y Predictor de saltos (256)		
	Ciclos	Tiempo de ejecución (ms)		Ciclos	Tiempo de ejecución (ms)		Ciclos	Tiempo de ejecución (ms)	
		osu035 (95,850 MHz)	osu018 (151,553 MHz)		osu035 (189,354 MHz)	osu018 (338,733 MHz)		osu035 (189,354 MHz)	osu018 (338,733 MHz)
dhrystone	266 660	2,78	1,76	377 290	1,99	1,11	322 534	1,70	0,95
median	4 224 930	44,08	27,88	6 639 432	35,06	19,60	5 598 855	29,57	16,53
mt-matmul	730 880	7,63	4,82	1 123 797	5,93	3,32	956 559	5,05	2,82
mt-vvadd	261 006	2,72	1,72	356 466	1,88	1,05	286 602	1,51	0,85
multiply	887 306	9,26	5,85	1 448 368	7,65	4,28	1 158 973	6,12	3,42
qsort	38 247 921	399,04	252,37	63 803 865	336,96	188,36	49 248 461	260,09	145,39
rsort	38 298 441	399,57	252,71	63 815 294	337,02	188,39	48 941 887	258,47	144,49
spmv	1 973 309	20,59	13,02	3 170 498	16,74	9,36	2 684 905	14,18	7,93
towers	374 523	3,91	2,47	560 783	2,96	1,66	466 655	2,46	1,38
exceptions	283 341	2,96	1,87	394 162	2,08	1,16	341 176	1,80	1,01

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Tabla XXVIII. **Aumento en rendimiento**

Programa	Pipeline de cinco etapas		Pipeline y Predictor de saltos (256)	
	Aumento en desempeño		Aumento en desempeño	
	osu035	osu018	osu035	osu018
dhrystone	39,63 %	57,97 %	63,33 %	84,79 %
median	25,71 %	42,23 %	49,07 %	68,66 %
mt-matmul	28,48 %	45,36 %	50,94 %	70,78 %
mt-vvadd	44,65 %	63,65 %	79,91 %	103,55 %
multiply	21,03 %	36,93 %	51,25 %	71,12 %
qsort	18,43 %	33,98 %	53,43 %	73,58 %
rsort	18,56 %	34,14 %	54,59 %	74,90 %
spmv	22,96 %	39,11 %	45,19 %	64,27 %
towers	31,94 %	49,27 %	58,55 %	79,38 %
exceptions	42,01 %	60,67 %	64,06 %	85,62 %

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

El aumento en desempeño es medido respecto al tiempo de ejecución de la micro-arquitectura de control.

2.10.3. Recursos utilizados

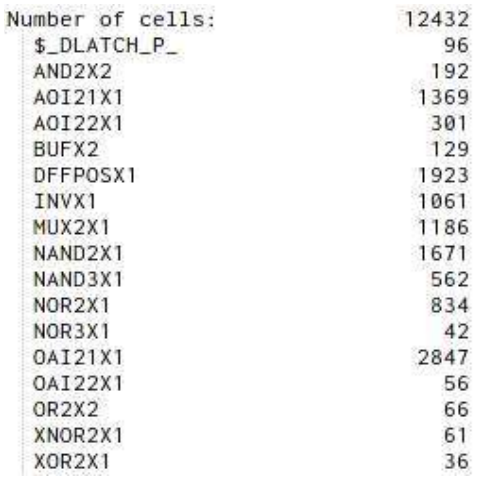
Cantidad de recursos utilizados:

Figura 117. **Recursos utilizados *pipeline* de cinco etapas, *osu035***

Number of cells:	12853
$_DLATCH_P_	96
AND2X2	202
AOI21X1	1298
AOI22X1	454
BUFX2	129
DFFPOSX1	1923
INVX1	1086
MUX2X1	409
NAND2X1	1861
NAND3X1	970
NOR2X1	959
NOR3X1	31
OAI21X1	3022
OAI22X1	284
OR2X2	68
XNOR2X1	45
XOR2X1	16

Fuente: elaboración propia, empleando Qflow v1.4.

Figura 118. **Recursos utilizados *pipeline* de cinco etapas, *osu018***



Number of cells:	12432
$_DLATCH_P_	96
AND2X2	192
AOI21X1	1369
AOI22X1	301
BUFX2	129
DFFPOSX1	1923
INVX1	1061
MUX2X1	1186
NAND2X1	1671
NAND3X1	562
NOR2X1	834
NOR3X1	42
OAI21X1	2847
OAI22X1	56
OR2X2	66
XNOR2X1	61
XOR2X1	36

Fuente: elaboración propia, empleando Qflow v1.4.

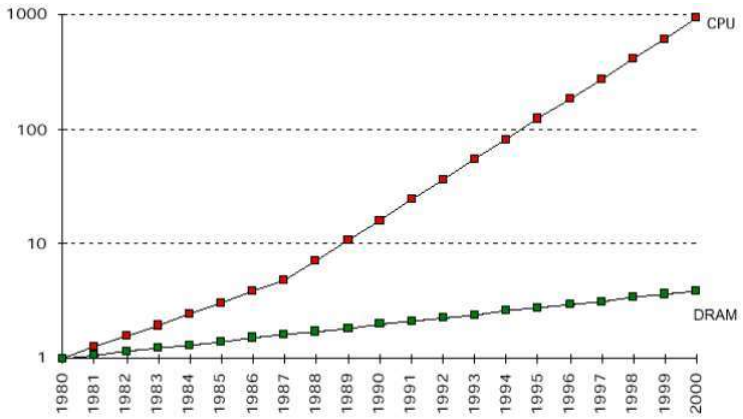
Respecto a la librería *osu035*, se utilizó un 18,88 % más de células estándar, para obtener un aumento en el rendimiento de hasta un 44,65 %.

Respecto a la librería *osu018*, se utilizó un 20,08 % más de células estándar, para obtener un aumento en el rendimiento de hasta un 63,65 %.

3. MEMORIA CACHÉ

En el transcurso de los años se ha creado una brecha en aumento entre la velocidad de los procesadores y memorias[21].

Figura 119. **Velocidad memorias *DRAM* vs., velocidad procesadores**



Fuente: CARVALHO, Carlos. *The gap between processor and memory speeds*. http://gec.di.uminho.pt/discip/minf/ac0102/1000gap_proc-mem_speed.pdf. Consulta: 18 de junio de 2020.

Si se toma en cuenta que toda instrucción que ejecuta el procesador requiere por lo menos un acceso a memoria se puede deducir un impacto negativo significativo. Por ejemplo, en simples términos, se posee un procesador que opera a una frecuencia de 800 MHz y se utiliza una memoria *DRAM* que

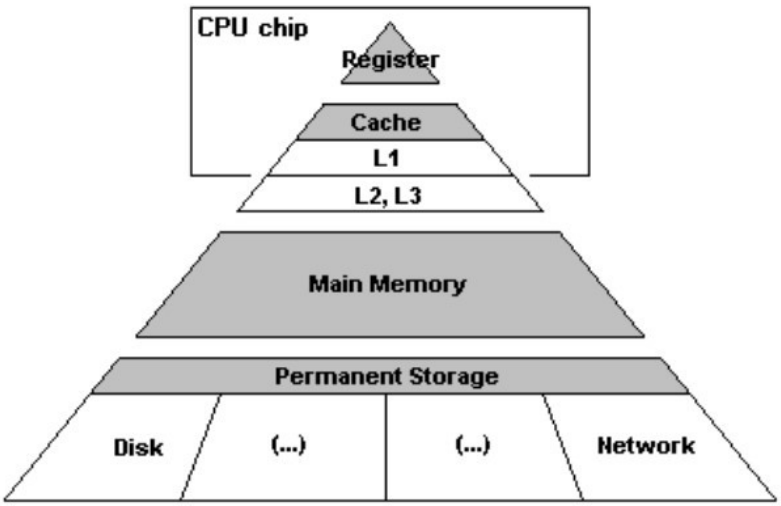
[21] CARVALHO, Carlos. *The gap between processor and memory speeds*. http://gec.di.uminho.pt/discip/minf/ac0102/1000gap_proc-mem_speed.pdf. Consulta: 18 de junio de 2020.

opera a 100 Mhz, el procesador únicamente realizara trabajo útil en uno de cada ocho ciclos.

3.1. Jerarquía de memoria

Un método para mitigar este problema es utilizar una jerarquía de memoria la cual consiste en una organización jerárquica multicapa de memorias en donde memorias de mayor proximidad al procesador poseen mayor velocidad, menor capacidad y mayor costo. Con el objetivo de obtener un sistema de memoria costo efectivo, de alto rendimiento y almacenamiento[22].

Figura 120. **Jerarquía de memoria**



Fuente: CARVALHO, Carlos. *The gap between processor and memory speeds*. http://gec.di.uminho.pt/discip/minf/ac0102/1000gap_proc-mem_speed.pdf. Consulta: 18 de junio de 2020.

[22] CARVALHO, Carlos. *The gap between processor and memory speeds*. http://gec.di.uminho.pt/discip/minf/ac0102/1000gap_proc-mem_speed.pdf. Consulta: 18 de junio de 2020.

Tabla XXIX. **Ejemplo jerarquía de memoria en computadora AlphaServer 8200**

Processor	Alpha 21164	
Machine	AlphaServer 8200	
Clock Rate	300 MHz	
Memory Performance	Latency	Bandwidth
I Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
D Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
L2 Cache (96KB on chip)	20 ns (6 clocks)	4800 MB/sec
L3 Cache (4MB off chip)	26 ns (8 clocks)	960 MB/sec
Main Memory Subsystem	253 ns (76 clocks)	1200 MB/sec
Single DRAM component	≈60ns (18 clocks)	≈30–100 MB/sec

Fuente: PATTERSON, David, ANDERSON, Thomas. *A Case for Intelligent RAM*. p. 3.

3.2. Principio de localidad

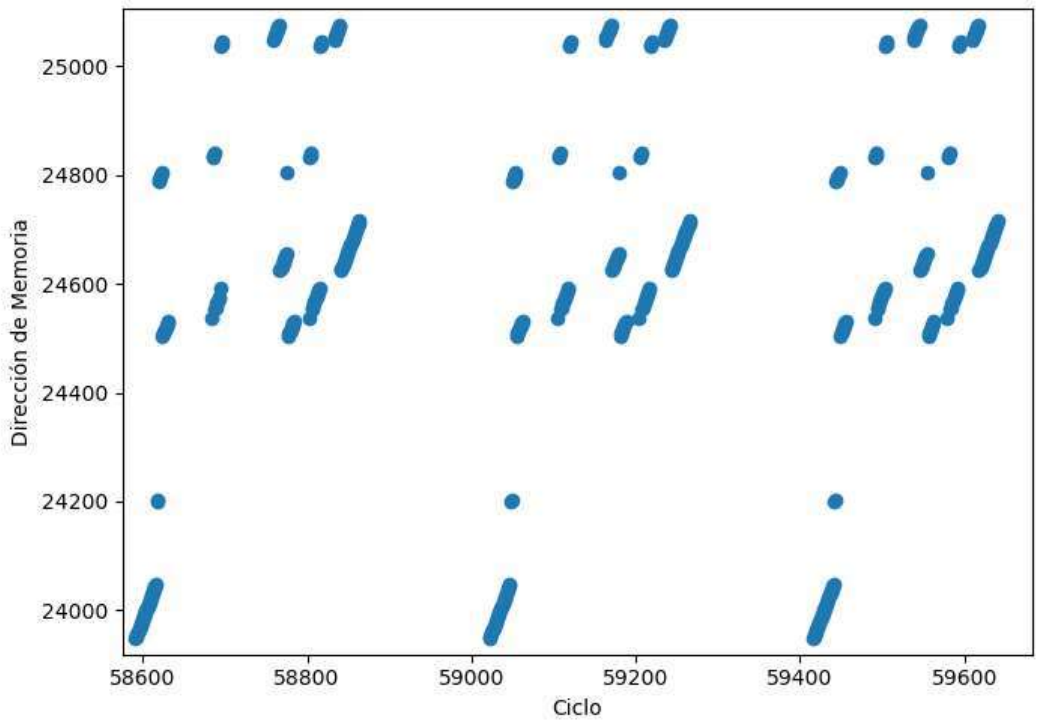
El principio de localidad o localidad de referencia, describe la tendencia de programas de computadoras a acceder la misma o cercanas localidades de memoria frecuentemente y repetidamente. Este principio puede ser separado en localidad espacial y localidad temporal[23].

Localidad temporal se refiere al acceso a la misma localidad de memoria en un periodo corto de tiempo. Un ejemplo de esto son los ciclos, se ejecutan las mismas instrucciones repetidamente en un periodo corto de tiempo. De igual manera subrutinas utilizadas frecuentemente en un programa.

[23] SHEN, Jhon Paul; LIPASTI, Mikko. *Modern Processor Design: Fundamentals of Superscalar Processors.* p. 113-114.

Respecto a memoria de datos, la localidad temporal se presenta en el acceso de variables de frecuente uso y acceso frecuente a la pila.

Figura 121. **Ejemplo localidad temporal, *towers***



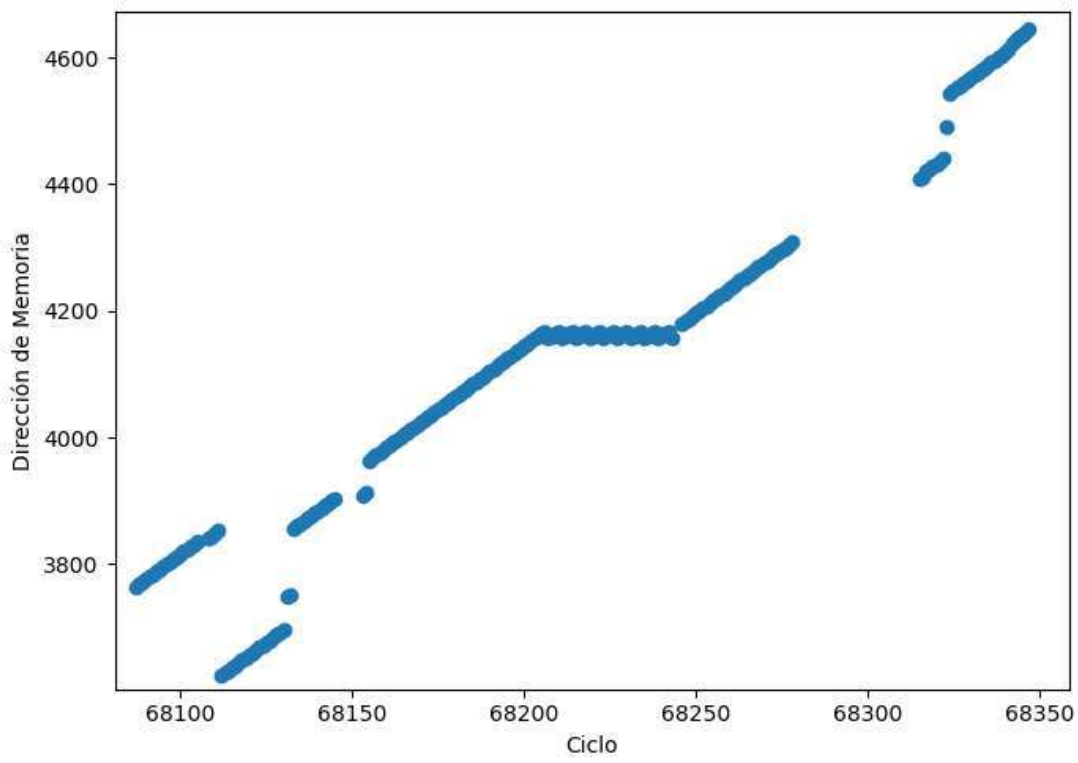
Fuente: elaboración propia, empleando Python 3.

Como se puede observar el programa accede a las mismas direcciones de memoria en corto tiempo.

Localidad espacial se refiere al acceso de localidades de memoria cercanas entre sí durante un periodo corto de tiempo. Esto es evidente debido a que los programas son ejecutados de manera secuencial, con excepción de la ejecución de instrucciones de transferencia de control.

Referente a memorias de datos, la localidad espacial ocurre en aplicaciones que recorren matrices de datos, o en aplicaciones donde se procesan datos en forma lineal como en la codificación y decodificación MP3.

Figura 122. **Ejemplo localidad espacial, *towers***



Fuente: elaboración propia, empleando Python 3.

3.3. Memoria caché

Una memoria caché es una memoria de alta velocidad, de alto precio y relativa baja capacidad de almacenamiento, que es conectada directamente al

procesador y memoria principal. Esta memoria aprovecha el principio de localidad para crear la ilusión de una memoria rápida y con basta capacidad,[24].

Una memoria caché entrará en funcionamiento al momento que el procesador solicite un acceso a memoria, si los datos de la localidad de memoria solicitada se encuentran almacenados dentro del caché, se dice que el caché acertó, por consiguiente, se consigue un acceso rápido a los datos. En caso contrario se dice que el caché falló, como resultado el caché realiza una solicitud a la memoria superior en la jerarquía, como consecuencia el acceso de memoria es lento. Al obtener los datos, son entregados al procesador y almacenados en el caché para su uso en posteriores accesos.

Si un programa posee altos índices de localidad espacial y temporal, la memoria caché acertará una gran porción de los accesos, en consecuencia, se obtendrá, en promedio, accesos rápidos.

El rendimiento de una memoria caché puede ser medido por el tiempo de acceso promedio, TAP, dado por la siguiente ecuación:

$$TAP = TiempoAcierto + \%Fallos * TiempoFallo$$

Donde:

TiempoAcierto = tiempo que le toma a la memoria caché despachar los datos en un acierto.

%Fallos = porcentaje de accesos que resultan en fallos.

[24] SHEN, Jhon Paul; LIPASTI, Mikko H. *Modern Processor Design: Fundamentals of Superscalar Processors.* p. 115.

TiempoFallo = tiempo que le toma a la memoria caché despechar los datos en un fallo, lo cual implica realizar una transacción con la siguiente memoria en la jerarquía.

Estos valores son dependientes de la organización y los parámetros de diseño de las memorias caché. Uno de estos parámetros de diseño es el tamaño de bloque.

3.3.1. Tamaño de bloque

El tamaño de bloque se refiere a la cantidad de *bytes* que almacena una línea de almacenamiento. Al aumentar la cantidad de *bytes* que almacena una línea, se puede tomar provecho de la localidad espacial. Por ejemplo, se posee un cache con un tamaño de bloque de treinta y dos *bytes* y se realiza un acceso a memoria en la localidad 0x4000 y resulta en un fallo, al llenar la memoria caché se almacenarán los valores de la localidad 0x4000 hasta la localidad 0x4020, como resultado accesos posteriores dentro de este rango resultarán en aciertos.

3.3.2. Política de escritura

Política de escritura se refiere al manejo de las peticiones de escritura del contenido de la memoria caché, por ejemplo, al ejecutar una instrucción *STORE*. Existen dos políticas principales, *write-back*, el cual almacena los cambios en el caché para ser escritos en memoria principal posteriormente y *write-through* donde todos los cambios son escritos en caché y en memoria principal, deteniendo el procesador hasta finalizar la transferencia.

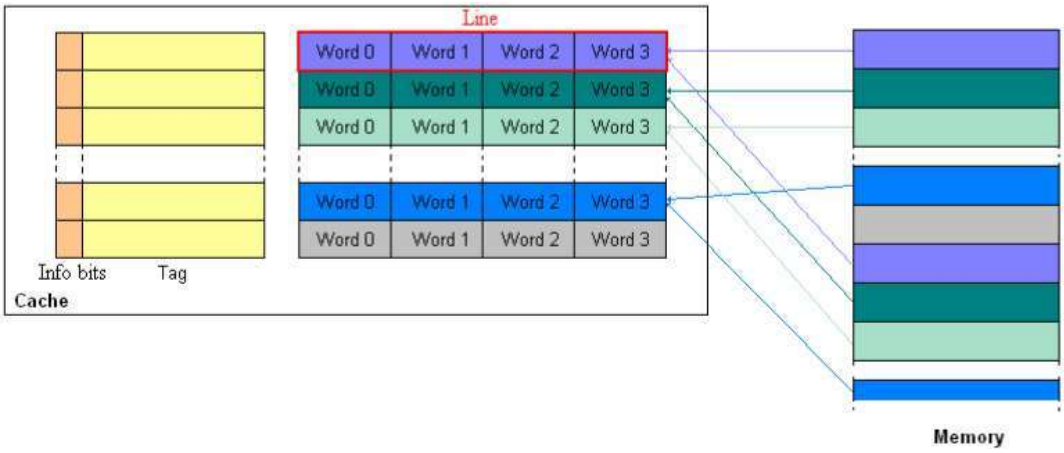
3.3.3. Organización

La organización del cache se referirse a la organización de los bloques de memorias y a sus políticas de llenado y reemplazo, entre ellos se encuentra el mapeo directo, completamente asociativo y conjunto asociativo de N vías

3.4. Caché de mapeo directo

Al momento de llenar una memoria caché existe una política la cual describe en qué línea debe ser situada la información, el mapeo directo, es un mapeo muchos-a-uno entre localidades de memoria y líneas de almacenamiento. En otras palabras, los datos de una localidad o dirección de memoria en particular sólo pueden ser colocados en una línea específica de almacenamiento.

Figura 123. **Caché de mapeo directo**



Fuente: GUILLE, Damien. *Study of Different Cache Line Replacement Algorithms in Embedded Systems*. p. 7.

Para seleccionar una línea de almacenamiento se utiliza una porción de *bits* de la dirección de memoria la cual es determinada por la capacidad del caché y el tamaño de bloque.

Otra porción de *bits* de dirección de memoria es utilizada para seleccionar una palabra específica en una línea de almacenamiento. En RV32I una palabra equivale a cuatro *bytes*.

El resto de los *bits* son llamados *tag*, el cual es utilizado como identificador para ser comparado con un *tag* previamente almacenado en la línea, si estos iguales se dice que el caché acertó, si no coinciden se dice que el caché falló.

Adicionalmente cada línea almacena un *bit* de validez, utilizado para verificar la confiabilidad de los datos y evitar falsos aciertos, por ejemplo, al iniciar el procesador existe la posibilidad que exista coincidencia en la comparación de *tag* sin haber llenado previamente el caché, este *bit* es escrito al momento de reemplazar una línea.

Debido a la política muchos-a-uno en caso de un fallo, sin importar si la línea se encuentre ocupada o no, dicha línea será reemplazada con los datos del acceso de memoria que resultó en fallo.

Si se utiliza política de escritura *write-back* se agrega un *bit*, llamado *dirty*, en cada línea del caché el cual poseerá el valor de uno cuando dicha línea sea modificada por instrucciones *STORE*. Al momento de que una línea sea reemplazada se leerá este *bit* para identificar si dicha línea posee datos para ser escritos en memoria principal, de ser así se debe iniciar un proceso de escritura a memoria principal en consecuencia realizando dos transferencias a memoria, una para escritura y otra para lectura. Al finalizar el proceso de reemplazo este

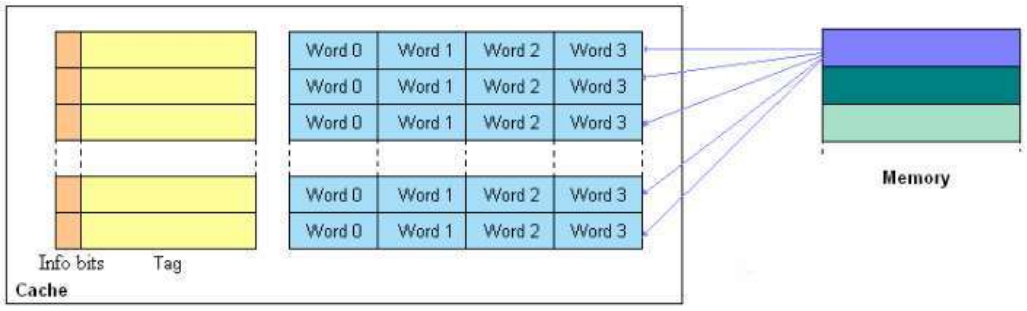
bit es escrito con el valor de cero, para indicar que los datos se encuentran limpios, sin ninguna modificación de parte del procesador. De no poseer este *bit*, por cada reemplazo de línea se debería escribir los datos a memoria provocando un uso innecesario del ancho de banda.

Una de las mayores ventajas del mapeo directo es su simple implementación, un problema que surge de este mapeo es el conflicto entre direcciones, es posible que dos o más direcciones de memoria de uso frecuente sean mapeados a la misma línea en el caché, provocando que la línea sea reemplazada frecuentemente eliminando cualquier beneficio de una memoria caché.

3.5. Caché completamente asociativo

El caché completamente asociativo utiliza un mapeo cualquiera-a-cualquiera entre localidades de memoria y líneas de almacenamiento. En otras palabras, los datos de cualquier localidad o dirección de memoria pueden ser colocados en cualquier línea de almacenamiento.

Figura 124. **Caché completamente asociativo**



Fuente: GUILLE, Damien. *Study of Different Cache Line Replacement Algorithms in Embedded Systems*. p. 6.

Como los datos pueden guardarse en cualquier línea de almacenamiento no se utiliza ningún *bit* de la dirección para indexar en el caché. Para comprobar si el caché acertó en un acceso de memoria, se debe comparar el *tag* de todas las líneas de almacenamiento con la dirección del acceso, por lo tanto, cada línea de almacenamiento debe poseer un comparador, si uno de estos comparadores presenta una respuesta positiva, se dice que el caché acertó.

Debido al mapeo cualquiera-a-cualquiera surge la pregunta, ¿Cómo seleccionar qué línea debe ser reemplazada? Como respuesta a esta pregunta surgen las políticas de reemplazo, las cuales dictan que línea de almacenamiento debe ser reemplazada. En este trabajo se presentarán tres políticas de reemplazo: aleatorio, *FIFO* y *LRU*

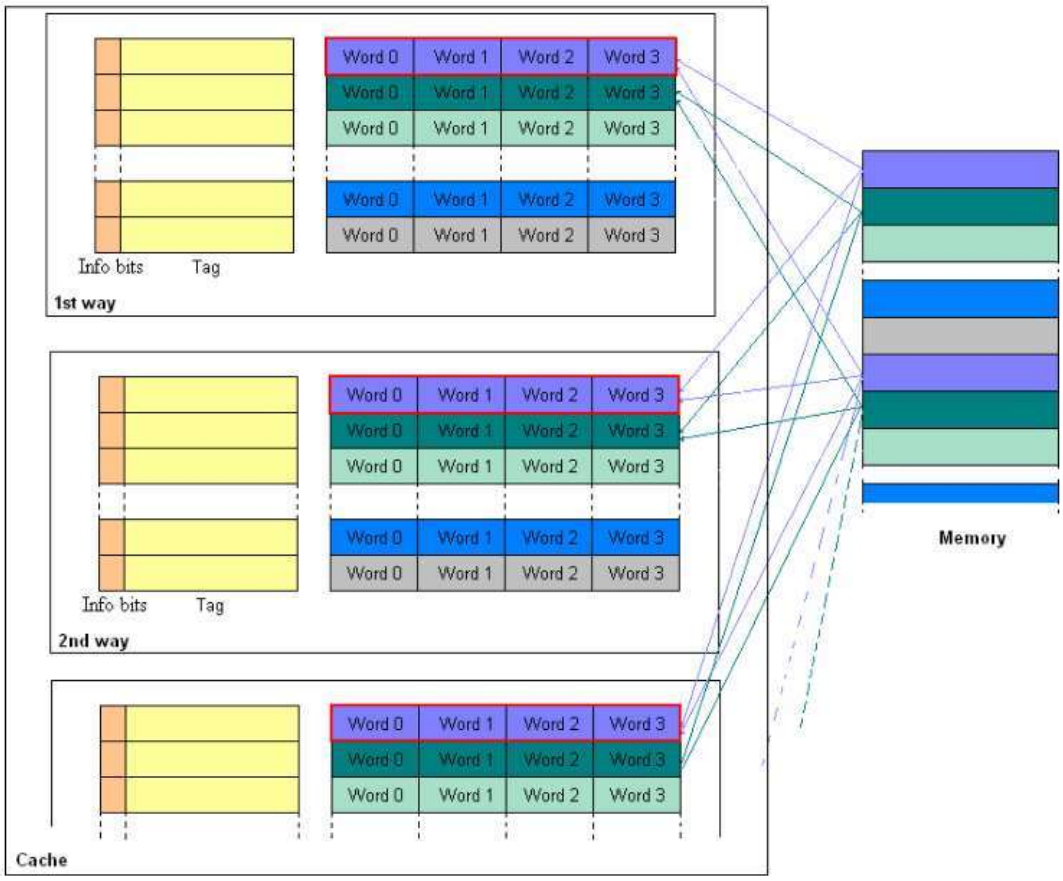
La política de reemplazo aleatorio define que la línea a ser reemplazada es seleccionada aleatoriamente. *First In First Out, FIFO*, primero en entrar primero en salir, define que la línea a ser reemplazada debe ser la de mayor antigüedad, es decir la primera línea que fue ingresada, al momento del que el caché se llene, debe ser la primera línea en ser reemplazada. Por último, *Least Recently Used*, *LRU*, menos recientemente usado, define que la línea a ser reemplazada debe ser la menos utilizada en el caché.

Este método de mapeo no sufre de conflicto de direcciones, a deferencia del mapeo directo, ya que no hay una línea determinada en donde almacenar los datos, pero este beneficio se le agregan inconvenientes, cada línea en el cache debe poseer un circuito comparador, aumentando el costo del caché, adicionalmente se debe implementar una política de reemplazo.

3.6. Caché conjunto asociativo de N vías

El caché completamente asociativo utiliza un mapeo muchos-a-pocos entre localidades de memoria y líneas almacenamiento. Utiliza una porción de los *bits* de la dirección para indexar a una línea, de la misma manera que en mapeo directo, pero con la diferencia que esa línea puede ser utilizada por una cantidad arbitraria de bancos de almacenamiento llamados vías.

Figura 125. **Cache conjunto asociativo de N vías**



Fuente: GUILLE, Damien. *Study of Different Cache Line Replacement Algorithms in Embedded Systems*. p. 8.

La selección de la línea depende del tamaño del cache, el tamaño del bloque y el número de vías.

El reemplazo de línea se realiza de la misma manera que en mapeo directo, pero utiliza políticas de reemplazo para seleccionar la vía en donde se reemplazará la línea.

Este tipo de mapeo reduce la cantidad de conflicto de direcciones. Únicamente se necesitan la misma cantidad de comparadores que vías, reduciendo la cantidad de recursos necesarios respecto a un cache completamente asociativo, aunque se requiere circuitos adicionales para implementar la política de reemplazo deseada, la cantidad de recursos adicionales es menor ya que estas políticas trabajan con la cantidad de vías y no cantidad de líneas de almacenamiento.

3.7. Implementación

En este trabajo se implementaron cuatro tipos de cache, cache de mapeo directo, cache conjunto asociativo de N vías con política de reemplazo *FIFO*, aleatorio y *LRU*.

Dichos caches fueron implementados como caches de datos, por lo tanto, poseen la capacidad de ser leídos y escritos, al poder ser escritos se implementó la política de escritura *write-back*. Esto es con el fin de demostrar todos los principios planteados, ya que si se desea implementar otra configuración solo deben eliminarse la porción no requerida, por ejemplo, si se desea implementar una memoria caché de instrucciones, la cual únicamente es utilizada para lectura, no es necesario el circuito de escritura.

Adicionalmente, para la comunicación con el nivel superior en la jerarquía de memoria, se asume que:

- La cantidad de *bits* del bus de datos es mayor o igual que a treinta y dos.

- Se realiza transferencia de datos únicamente cuando una señal, en este caso *EMemAccess*, se encuentre en alto.

- Se posee un puerto de escritura y un puerto lectura, *EDataStore* y *EDataLoad,* respectivamente.

- El nivel superior notificará cuando una transacción de lectura o escritura se complete satisfactoriamente al poner en alto las señales *EDoneLoad* y *EDoneStore*, respectivamente.

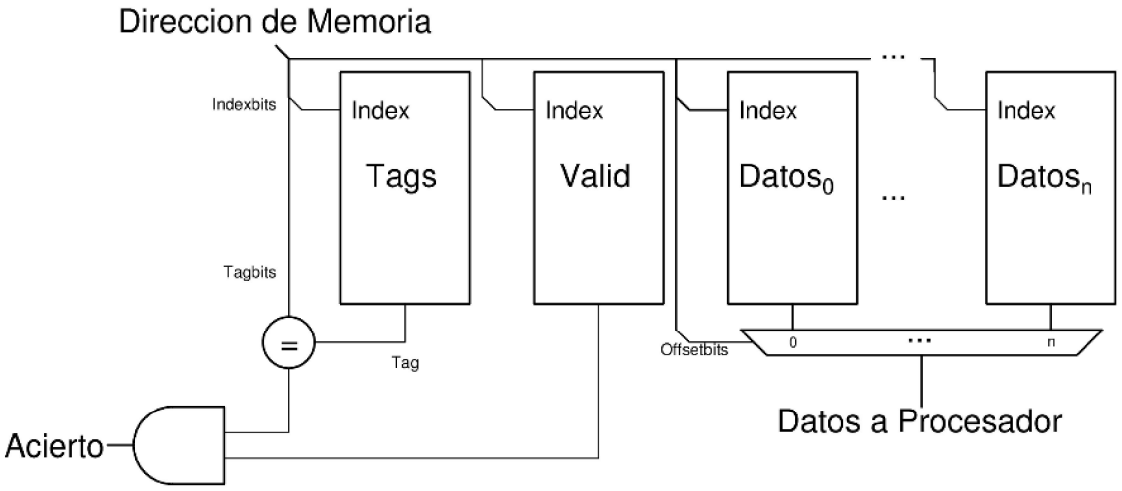
- Sí el tamaño de bloque es mayor al tamaño de bus, se realizan transacciones consecutivas hasta llenar el bloque.

- Se utiliza la señal *EWen* para diferenciar entre transferencias de lectura y escritura, cero y uno respectivamente.

3.7.1. Mapeo directo

A continuación, se presenta la implementación de un cache de mapeo directo.

Figura 126. **Cálculo de acierto en accesos y obtención de datos**



Fuente: elaboración propia, empleando Xcircuit v3.10.

$$\#lineas = \frac{M}{N}$$

$$offsetbits = Dirección\ de\ memoria[log_2(N) - 1 : 2]$$

$$indexbits = Dirección\ de\ memoria[log_2(N * \#lineas) - 1 : log_2(N)]$$

$$tagbits = Dirección\ de\ memoria[31 : log_2(N * \#lineas)]$$

Donde:

M = tamaño de la memoria caché, medido en *bytes*.

N = tamaño de bloque, medido en *bytes*.

#lineas = número de líneas de almacenamiento que posee la memoria caché.

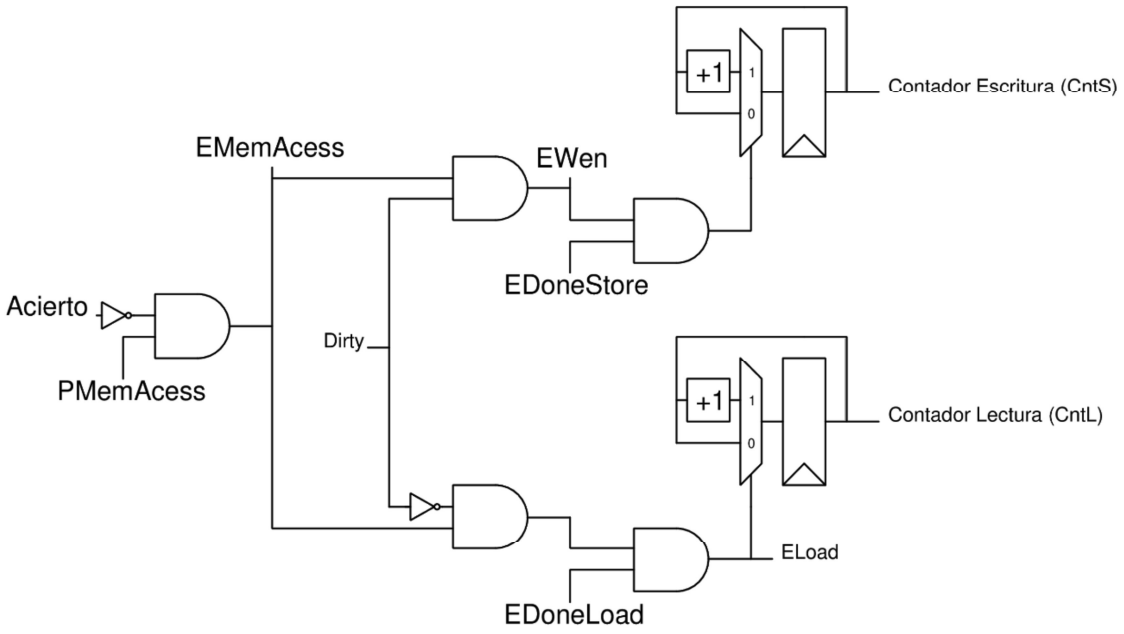
Dirección de memoria = señal que contiene la dirección de memoria a la que se desea acceder.

offsetbits = porción de la dirección de memoria utilizada para seleccionar una palabra en el bloque de datos.

indexbits = porción de la dirección de memoria utilizada para indexar en los diferentes bancos, seleccionan la línea de almacenamiento solicitada
tagbits = porción de la dirección de memoria utilizada como identificador.

En el caso de un fallo, se debe reemplazar la línea, pero antes debe de verificarse si dicha línea ha sido modificada con anterioridad, de ser así debe iniciarse una transferencia de escritura en el nivel superior en la jerarquía de memoria. Por último, iniciar una transferencia de lectura para reemplazar los datos de la línea.

Figura 127. **Control transferencias de lectura y escritura**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Nuevamente se utiliza una porción de la dirección de memoria para indexar en la memoria que almacena el *bit dirty*. La señal *EMemAccess* es generada al momento que un acceso resulte en fallo, es decir en un acierto negado.

Adicionalmente se utiliza la señal *PMemAccess*, la cual es generada por el procesador al momento solicitar acceso al caché, con el objetivo de evitar falsos accesos a cache los cuales pueden generar falsos fallos, en consecuencia, evitar transferencias innecesarias, esto se debe a que el procesador conecta el resultado del *ALU* con la dirección de memoria a solicitar, por lo tanto, instrucciones diferentes a *LOAD* o *STORE* generan valores que el caché puede malinterpretar como accesos a memoria.

EWen es generado utilizando el *bit dirty* y *EMemAccess.*

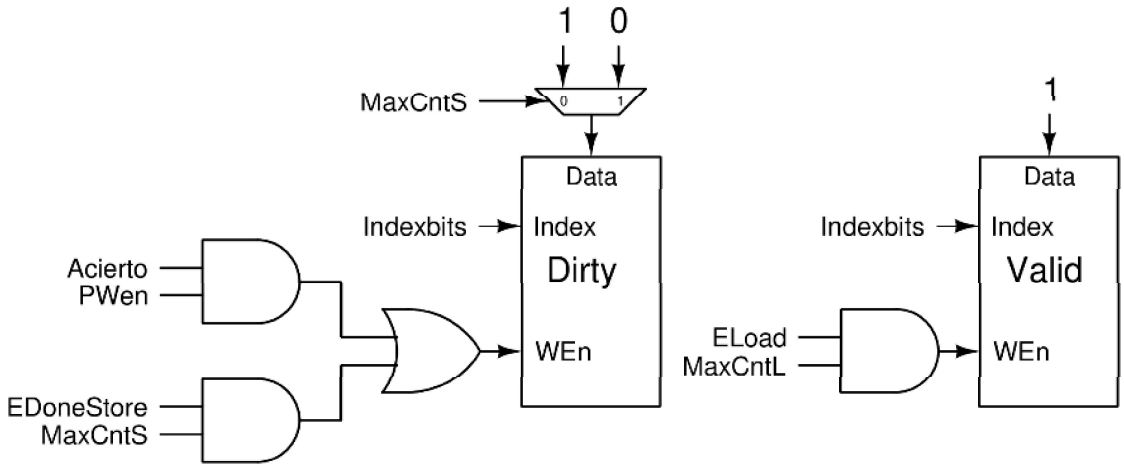
Al momento de realizar una transferencia de escritura, si el tamaño de bloque es mayor al tamaño del bus, se deben realizar transferencias consecutivas, para ello se utiliza un contador el cual seleccionará los *sub-bloques* que transmitirán sus datos, el valor máximo de este contador está dado por la relación entre el tamaño del bloque y el tamaño del bus. Por ejemplo, con un tamaño de bloque de treinta y dos *bytes* y un tamaño de bus de ocho *bytes*, el contador poseerá los valores de cero a tres, en otras palabras, se realizarán cuatro transacciones consecutivas de los datos de dos *sub-bloques* a la vez, debido a que cada *sub-bloque* posee un tamaño de bus de cuatro *bytes*.

Este contador es un contador cíclico, es decir al momento de poseer su valor máximo, el siguiente valor en la secuencia es cero. El contador incrementará su valor únicamente, cuando las señales *EDoneStore* y *EWen* se encuentren en alto, ya que ambas indican una transferencia exitosa, por ende, solicitando los datos de la siguiente transferencia.

De la misma manera, para transferencias de lectura se posee un contador de lectura que posee todas las características del contador de escritura. Para incrementar su valor, la señal *ELoad* debe poseer el valor de uno.

Como se puede observar ambas operaciones, escritura y lectura, son mutuamente exclusivas por medio del *bit dirty*. Si una línea que se desea reemplazar ha sido escrita con anterioridad por el procesador, como lo indica el *bit dirty*, primero se iniciará una transferencia de escritura para almacenar los datos modificados en la siguiente memoria en la jerarquía, al finalizar se iniciará una trasferencia de lectura. En caso de que el procesador no haya modificado los datos, se iniciará una transferencia de lectura, sin necesidad de realizar una transferencia de escritura.

Figura 128. **Escritura *bit dirty* y validez**



Fuente: elaboración propia, empleando Xcircuit v3.10.

El *bit dirty* es modificado en dos posibles escenarios, cuando el procesador modifica una línea en el cache y al momento de finalizar una transferencia de escritura, ya que se debe escribir el *bit dirty* en cero para habilitar la transferencia de lectura.

Por consiguiente, el habilitador de escritura, es controlado por dos diferentes escenarios, representado por la compuerta *OR*. El primero cuando el

procesador modifica una línea, esto únicamente es posible si la línea se encuentra en el caché, es decir en un acierto y la señal *PWen* se encuentra en alto, la cual es controlada por el procesador para indicar una operación de escritura.

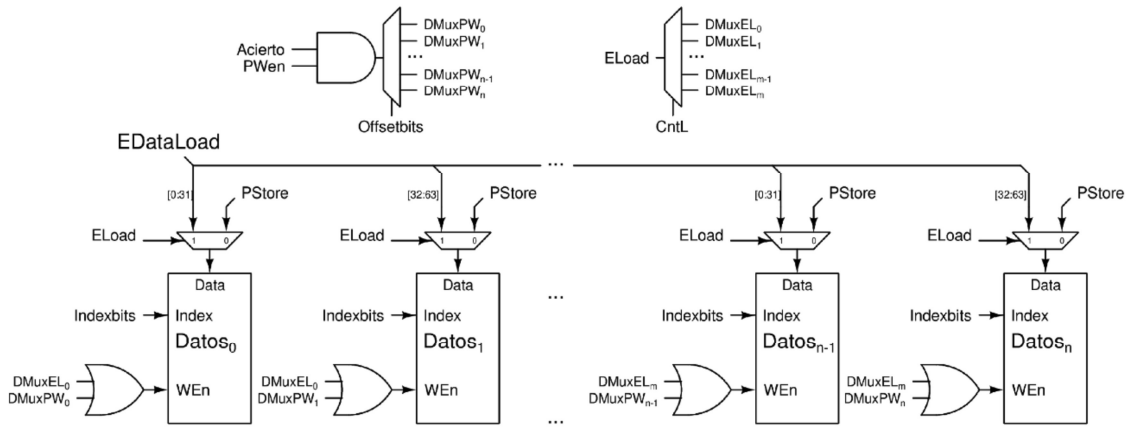
El segundo escenario es al finalizar una transferencia de escritura, es decir cuando se realiza la última transferencia de escritura, el cual puede ser representado cuando el contador de escritura posee el valor máximo posible, es decir cuando todos los *bits* del contador poseen el valor de uno y la memoria en el nivel superior notifica una transacción satisfactoria a través de la señal *EDoneStore.*

La señal *MaxCntS* es la función lógica *AND* aplicada entre todos los *bits* del contador de escritura.

El valor a ser escrito es controlado por un multiplexor que utiliza la señal *MaxCntS* para seleccionar entre cero y uno, es decir cuando se ha finalizado una transferencia de escritura en una línea modificada o cuando el procesador modifica la línea.

El *bit* de validez y la memoria de *tags* son escritos cuando una línea ha sido reemplazada con éxito, es decir al finalizar la última transferencia de lectura. La memoria de *tags* es escrita con el valor de *tagbits*, para poder ser identificado en posteriores accesos.

Figura 129. **Escritura de bloque de datos**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Los bloques pueden ser escritos en dos situaciones, cuando el procesador ejecuta una instrucción *STORE* o cuando se reemplaza una línea, en una transferencia de lectura. Adicionalmente, se debe tomar en consideración la diferencia en tamaño del bus del procesador y bus de memoria.

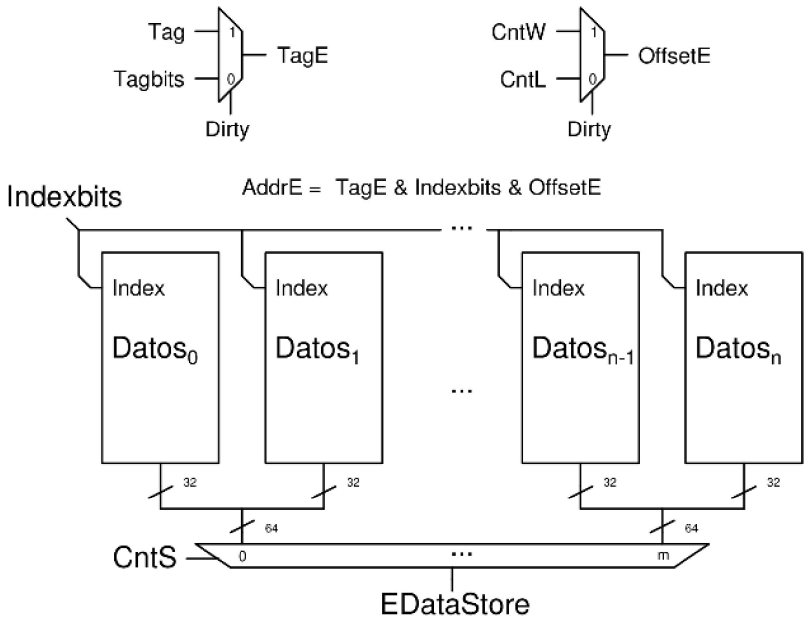
Para ello se utilizan demultiplexores para transmitir la señal habilitadora de escritura a los *sub-bloques* correspondientes. En el caso de transferencias de lectura, se utiliza el valor de contador de lectura para escribir al conjunto de *sub-bloques* correspondientes. Por último, al momento que el procesador ejecute una instrucción de escritura, los *bits* de *offset* son utilizados para seleccionar el *sub-bloque* correspondiente.

Los valores por ser escritos en los bloques de datos son seleccionados por un multiplexor controlado por la señal *ELoad,* de los cuales *PStore*, son los datos que el procesador desea almacenar, como se puede observar todos los *sub-*

bloques están conectados a esta señal, la señal habilitadora de escritura de-multiplexada *DMuxPW* seleccionará el correcto *sub-bloque* a escribir.

La distribución de los datos de una transferencia de lectura depende de la relación entre el tamaño de bus de memoria y el tamaño de bus de un *sub-bloque*. Por ejemplo, como se puede observar el tamaño de bus de memoria es de sesenta y cuatro *bits* y el tamaño de bus de un sub-bloque es de treinta y dos *bits*, por lo tanto, los datos de una transferencia de lectura, *EDataLoad*, son distribuidos en dos *sub-bloques* de datos, el primer bloque almacena los primeros treinta y dos *bits*, es decir los *bits* cero al treinta y uno, y el segundo bloque almacena los últimos treinta y dos bloques, los *bits* treinta y dos al sesenta y tres. Por tal motivo la señal habilitadora de-multiplexada *DMuxEL* es compartida por dos *sub-bloques*, la cual escogerá la pareja de sub-bloques correspondientes.

Figura 130. **Selección *EDataStore* y manejo de dirección de transacción**



Fuente: elaboración propia, empleando Xcircuit v3.10.

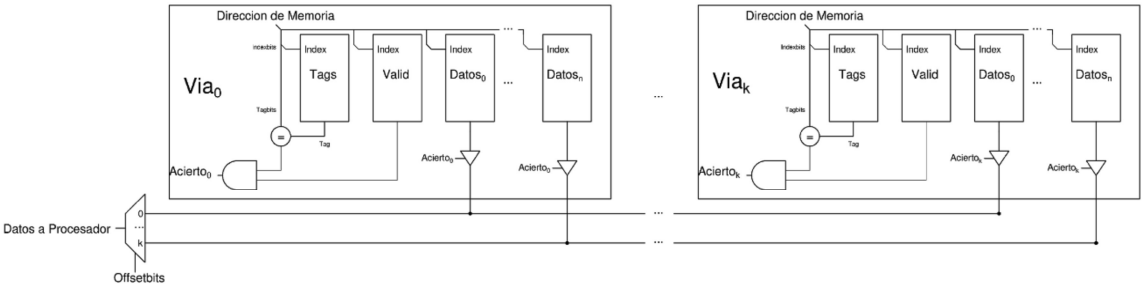
Al igual que los datos de una transferencia de lectura, la distribución de los datos de una transferencia de escritura depende de la relación tamaño de bus de memoria y el tamaño de bus de un bloque.

Dependiendo del tipo de transferencia que se realice la dirección de la transferencia debe cambiar, por ejemplo la dirección de una transferencia de lectura es la dirección que el procesador solicitó, pero la dirección de una transferencia de escritura, debe ser la dirección de los datos que fueron modificados con anterioridad, afortunadamente dicha dirección puede ser reconstruida utilizando el *tag*, almacenado en la línea, concatenado con los *bits* de indexación y el valor del contador de escritura o lectura. Ya que el *bit dirty* controla el tipo de transacción, este es utilizado para controlar la dirección de la transferencia.

3.7.2. Conjunto asociativo de N vías

A continuación, se presenta la implementación de un cache conjunto asociativo de N vías.

Figura 131. **Cálculo de acierto en accesos y obtención de datos**



Fuente: elaboración propia, empleando Xcircuit v3.10.

El circuito de acierto y obtención de datos es idéntico en cada vía del cache, y cómo se logra observar es similar al circuito de un cache de mapeo directo, con diferencia que los datos no son multiplexados individualmente en cada vía, sino que todos los datos de las vías son conectados a un bus el cual es multiplexado para ser entregar al procesador. Adicionalmente, las porciones de *bits* para formar las señales *offsetbits*, *indexbits* y *tagbits* son diferentes.

$$\#lineas = \frac{M}{N * K}$$

Donde:

M = tamaño de la memoria caché, medido en *bytes*.

N = tamaño de bloque, medido en *bytes*.

K = cantidad de vías.

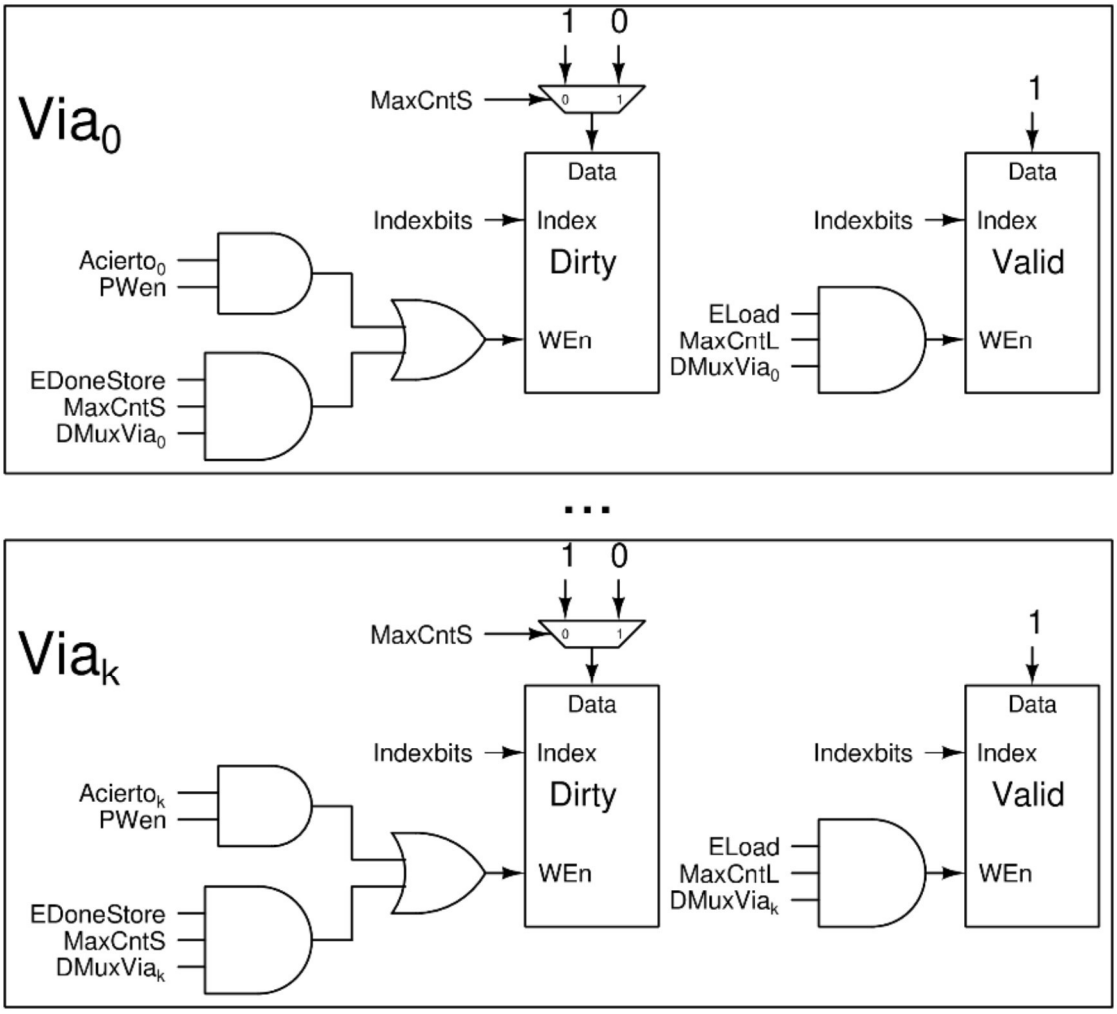
#lineas = número de líneas de almacenamiento que posee cada vía en la memoria caché.

El resto de señales son calculadas de la misma manera que en mapeo directo.

Para evitar conflictos, los datos de las vías son conectados a *buffers* de tres estados, los cuales serán activados por las señales de acierto correspondiente de cada vía.

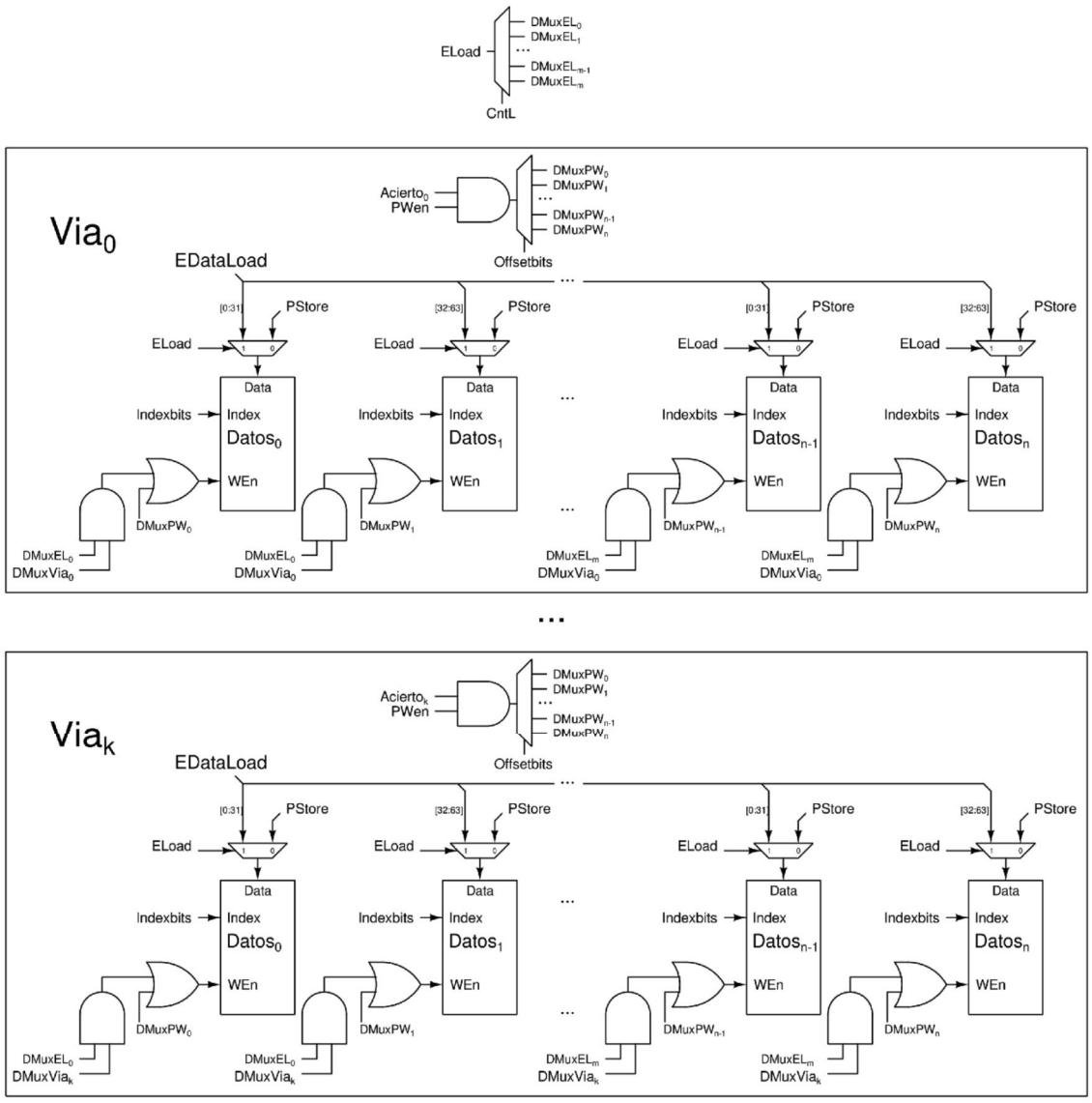
La señal de acierto global del cache es generada al aplicar la función lógica *OR* a las señales de acierto de todas las vías, en otras palabras, el cache acertara si una de sus vías acierta.

Figura 132. **Escritura *bit dirty* y validez por vía**



Fuente: elaboración propia, empleando Xcircuit v3.10.

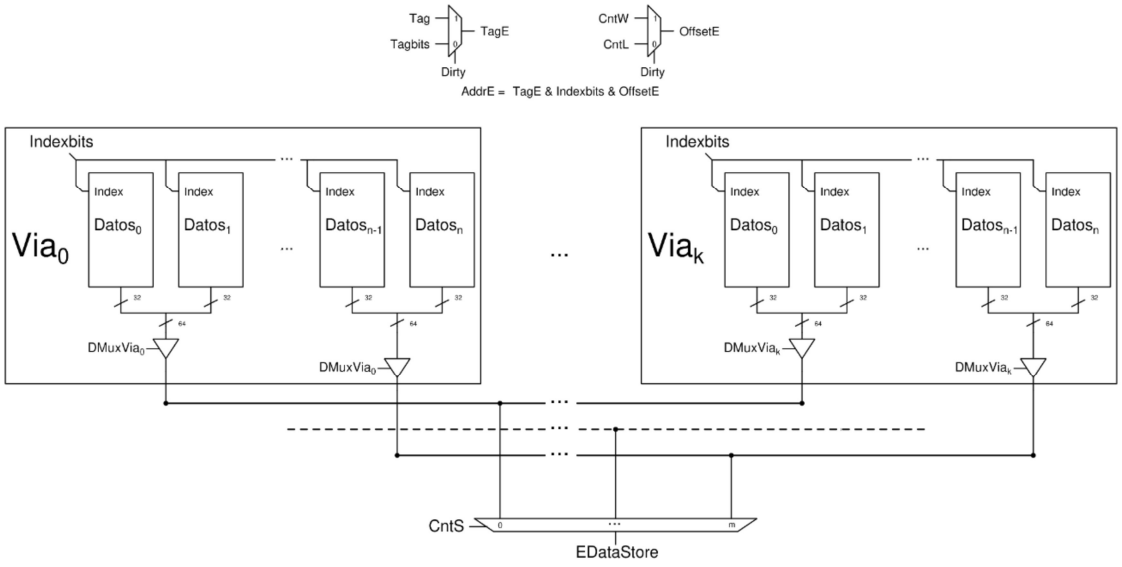
Figura 133. **Escritura de bloque de datos**



Fuente: elaboración propia, empleando Xcircuit v3.10.

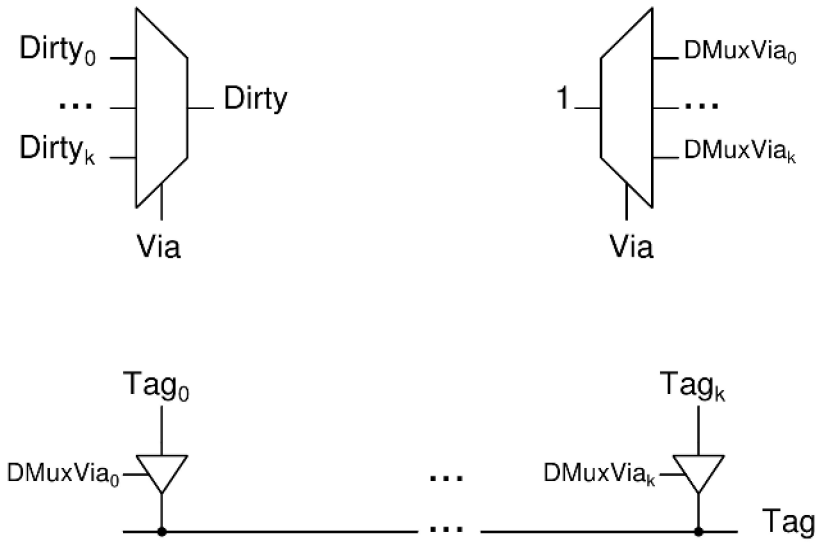
Como se puede observar los circuitos son similares a un cache de mapeo directo, con la diferencia de añadir la señal de-multiplexada *DMuxVia*, utilizada para habilitar la escritura en la vía correspondiente.

Figura 134. **Selección *EDataStore* y manejo de dirección de transacción**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Figura 135. **Señal *DMuxVia*, *tag* y *dirty* global**



Fuente: elaboración propia, empleando Xcircuit v3.10.

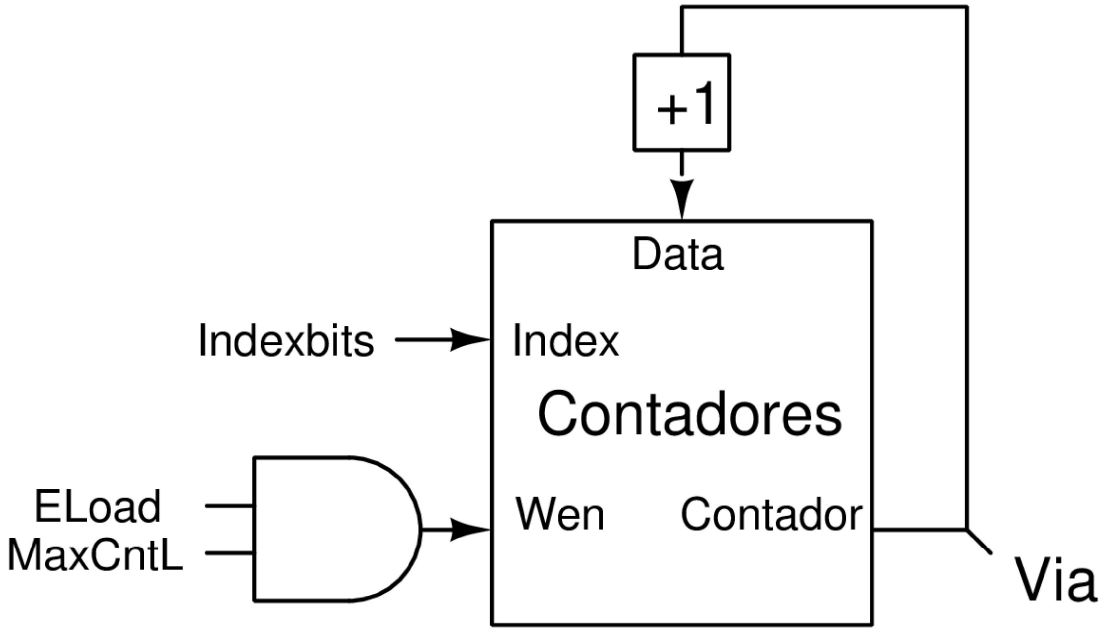
El circuito de manejo de control de transferencias de lectura y escritura es idéntico a un cache de mapeo directo, por lo tanto, no es presentado.

La señal *Vía* es responsable por la mayoría de la lógica de control del caché conjunto asociativo de N vías. Ya que esta señal dicta qué vía debe ser reemplazada, para dar lugar a nuevos datos. Como se mencionó con anterioridad la decisión de qué vía debe ser reemplazada es dictada por la política de reemplazo.

3.7.2.1. Remplazo *FIFO*

A continuación, se presenta la implementación de la política de reemplazo *FIFO*.

Figura 136. **Implementación política *FIFO***



Fuente: elaboración propia, empleando Xcircuit v3.10.

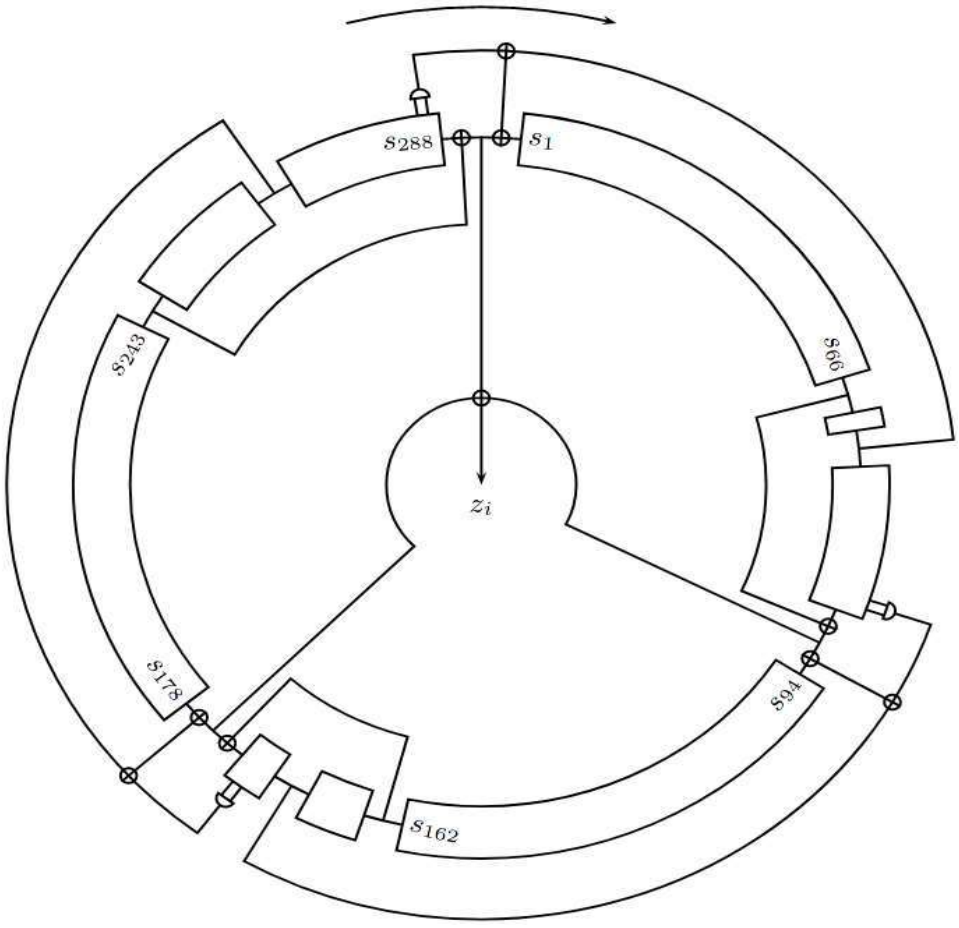
Esta política de reemplazo es relativamente simple de implementar, la cual consta de usar un contador cíclico el cual dicta que vía debe ser reemplazada, este contador únicamente incrementará su valor al ser reemplazada una línea, y su valor máximo es el número de vías menos uno.

El principio de esta implementación es que el contador siempre apunta a la vía que posee el dato más viejo, debido a que cuando el contador completa un ciclo todas las vías diferentes al valor actual del contador ya han sido reemplazadas una vez, esto quiere decir que el valor actual del contador representa la vía que posee el mayor tiempo sin ser reemplazada, la vía con mayor antigüedad.

3.7.2.2. Reemplazo aleatorio

Como su nombre lo indica, se reemplaza la vía de forma aleatoria, para ello se utilizó Trivium, un cifrador de flujo, pero en este trabajo se utilizó como un generador pseudo-aleatorio de números.

Figura 137. **Diagrama Trivium**



Fuente: DE CANNIÉRE, Christophe, PRENEEL,Bart. *Trivium, A Stream Cipher Construction Inspired by Block Cipher Design Principles*. https://www.ecrypt.eu.org/stream/papersdir/2006/021.pdf. Consulta: enero de 2021.

Figura 138. **Algoritmo Trivium**

$$
\begin{aligned}
&\textbf{for } i = 1 \text{ to } N \textbf{ do} \\
&\quad t_1 \leftarrow s_{66} + s_{93} \\
&\quad t_2 \leftarrow s_{162} + s_{177} \\
&\quad t_3 \leftarrow s_{243} + s_{288} \\
&\quad z_i \leftarrow t_1 + t_2 + t_3 \\
&\quad t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171} \\
&\quad t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264} \\
&\quad t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69} \\
&\quad (s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92}) \\
&\quad (s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176}) \\
&\quad (s_{178}, s_{279}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287}) \\
&\textbf{end for}
\end{aligned}
$$

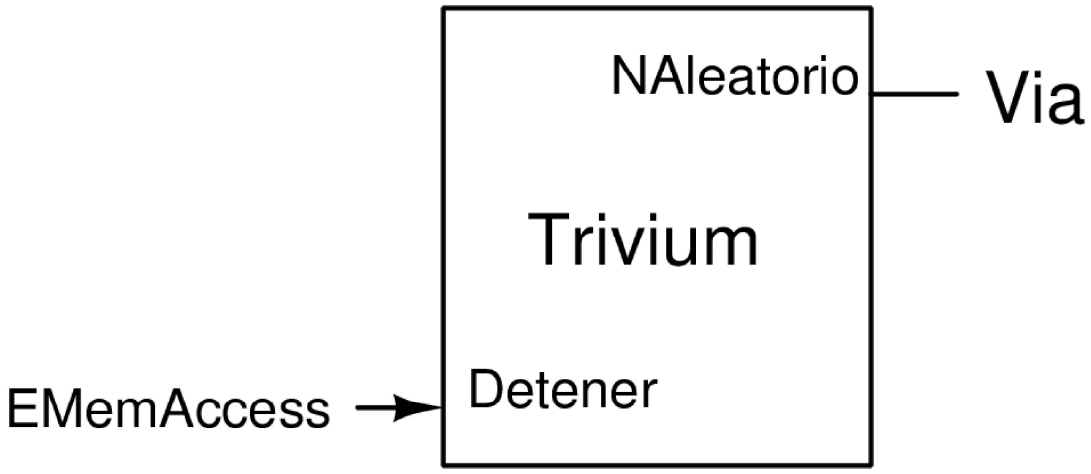
Fuente: DE CANNIÉRE, Christophe, PRENEEL,Bart. *Trivium, Specifications.* https://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf. Consulta: enero de 2021.

Las operaciones con el símbolo "+" se refieren a compuertas *XOR* y las operaciones con el símbolo "·" se refieren a compuertas *AND*. Las señales s_1 a s_{288} se refieren a *flip-flops* que almacenan el estado interno de Trivium. La señal Z_i es utilizada para generar la señal vía. Para una detallada explicación del diseño[25] y sus especificaciones[26].

[25] DE CANNIÉRE, Christophe, PRENEEL, Bart. *Trivium, A Stream Cipher Construction Inspired by Block Cipher Design Principles*. https://www.ecrypt.eu.org/stream/papersdir/2006/021.pdf. Consulta: enero de 2021.

[26] DE CANNIÉRE, Christophe, PRENEEL, Bart. *Trivium, Specifications.* https://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf. Consulta: enero de 2021.

Figura 139. **Trivium como política de reemplazo**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Cada ciclo del reloj el estado interno en Trivium será actualizado generando un nuevo número. Al momento de realizar una transferencia de escritura o lectura, la señal *Vía* debe ser estable durante toda esta operación, para ello se utiliza la señal *EMemAccess* como deshabilitador de escritura en el módulo Trivium, el cual al poseer el valor de cero, los estados de Trivium podrán cambiar libremente, mientras que cuando la señal *EMemAccess* posea el valor de uno, se deshabilitará la escritura en los estados internos del módulo Trivium, por lo tanto, la señal *Vía* no cambiará de valor hasta finalizar todas las transferencias de lectura y escritura.

3.7.2.3. Reemplazo *LRU*

Esta política reemplaza la vía menos recientemente utilizada. Una manera de implementarlo es utilizar una estructura de memoria tipo pila, en donde el tope de la pila almacena la vía más recientemente accedida, y el fondo de la pila se

encuentra la vía menos recientemente accedida. Al acertar, la vía que posee los datos es extraída de la pila y es colocada en el tope. De haber un fallo, la vía en el fondo de la pila es reemplazada. Con el siguiente pseudocódigo se plantea dicho algoritmo.

Figura 140. **Pseudocódigo algoritmo *LRU***

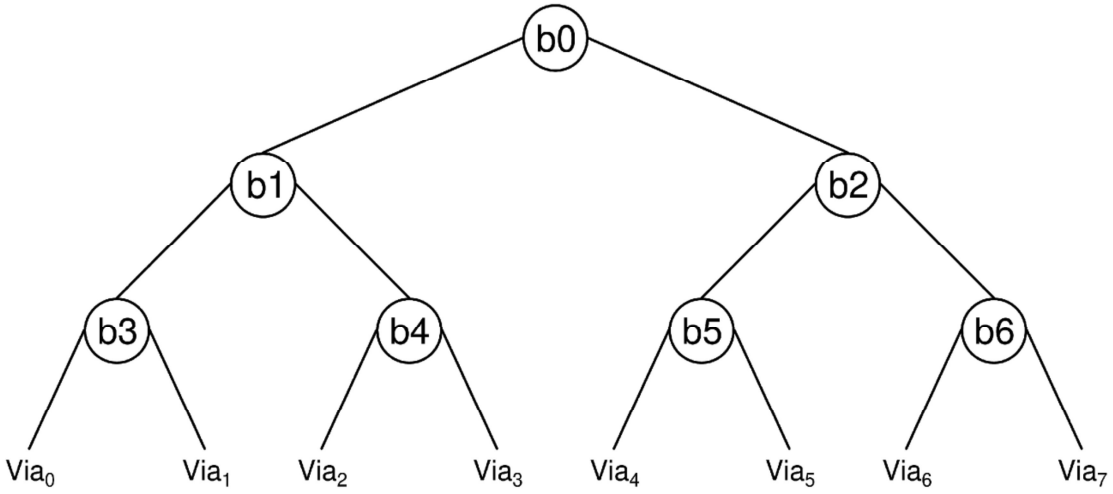
```
Si acierto:
    remover(pila,ViaAccedida)
    agregar_tope(pila,ViaAccedida)
De lo contrario:
    ViaReemplazar = pila.pop(-1) //Extraer valor de fondo
    agregar_tope(pila,ViaReemplazar)
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Aunque este algoritmo es relativamente sencillo, implementarlo en *hardware* requiere una gran cantidad de recursos, adicional a un sistema complejo de control. Por lo tanto, en lugar de implementar *LRU* directamente, se implementó una aproximación llamada *Tree-LRU*.

Tree-LRU utiliza un árbol binario con N-1 *bits* para representar el estado de acceso en N vías. Por ejemplo:

Figura 141. **Árbol binario de ocho vías**

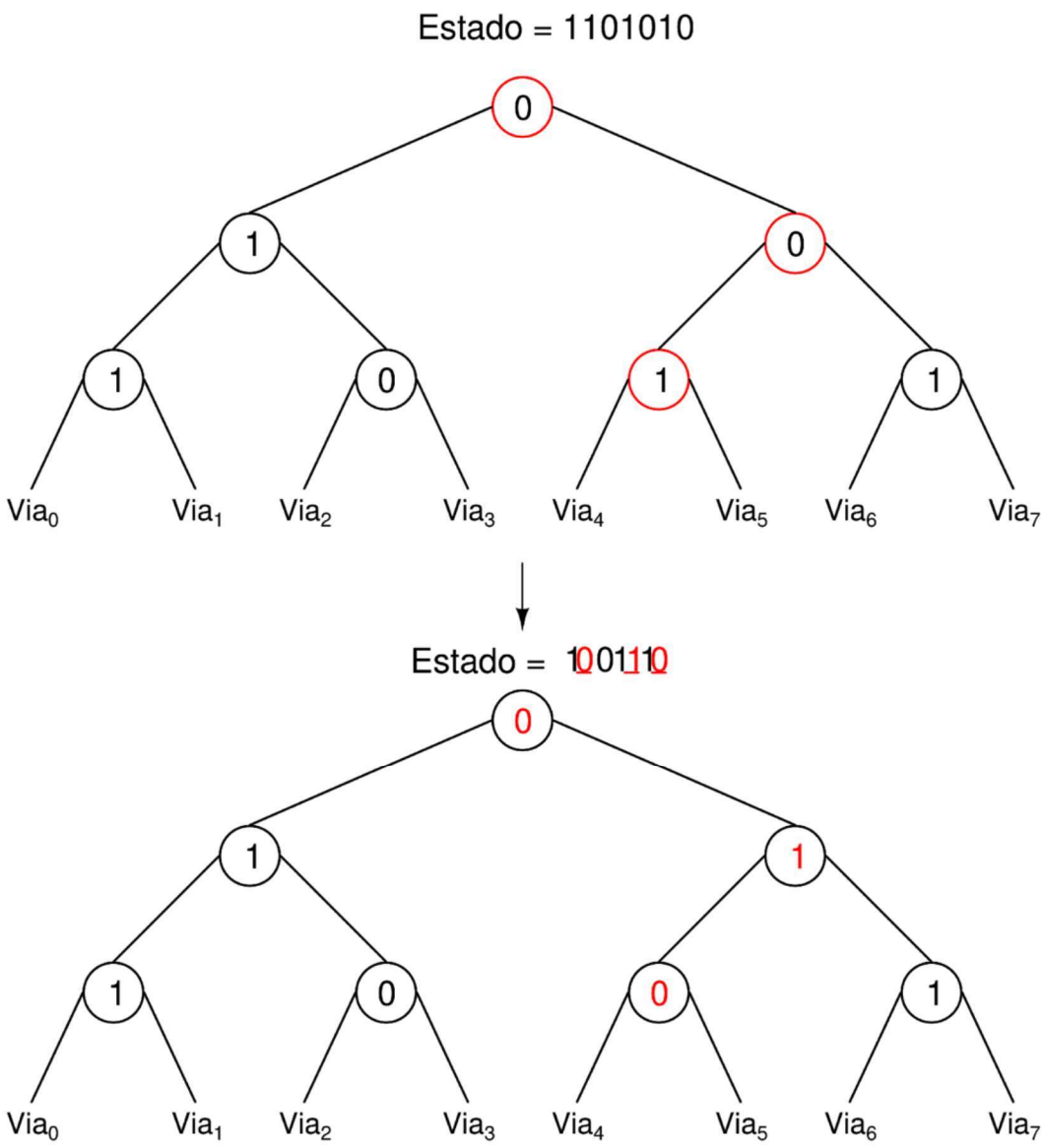


Fuente: elaboración propia, empleando Xcircuit v3.10.

El *bit* cero del estado, es la raíz del árbol, cada *bit* de estado es un nodo que es utilizado para descender en el árbol por la derecha o por la izquierda, dependiendo si su valor es uno o cero respectivamente, hasta llegar a la vía que será reemplazada.

Los estados del árbol se actualizan en dos diferentes escenarios, cuando se realiza un acceso con acierto y un acceso en fallo.

Figura 142. **Actualización estado árbol binario en acierto**

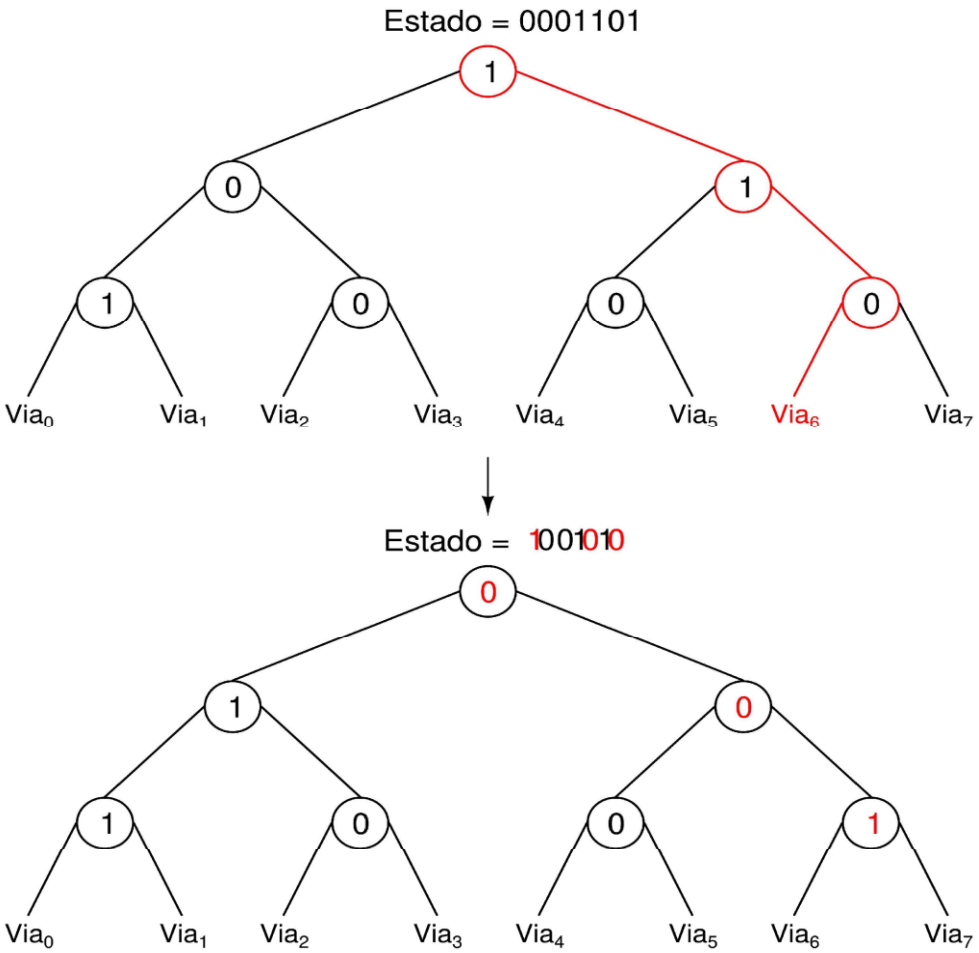


Fuente: elaboración propia, empleando Xcircuit v3.10.

Al momento de un acierto se deben modificar únicamente los *bits* de estado que pueden ser recorridos hacia la vía que acertó el acceso, cambiado su valor de tal forma que apunten en dirección opuesta a dicha vía. Por ejemplo, si acierta

la vía cinco, los *bits* de estado cero, dos y cinco son los únicos capaces de ser recorridos y terminar en la vía cinco, por lo tanto, estos *bits* deben ser cambiados de tal forma que apunten en dirección opuesta a la vía accedida, en este caso el *bit* cero debe apuntar a la izquierda, el *bit* dos a la derecha y por último el *bit* cinco la izquierda, es decir deben poseer el valor de cero, uno y cero, respectivamente. En concreto, si la vía cinco es accedida, el estado debe ser modificado de la siguiente forma: X0XX1X0, donde "X" significa no modificar el valor.

Figura 143. **Actualización estado árbol binario en fallo**



Fuente: elaboración propia, empleando Xcircuit v3.10.

Cuando hay un fallo en el acceso del cache, se recorre el árbol binario para seleccionar la vía a reemplazar. Cuando el cache finalice el proceso de reemplazo la vía reemplazada se convierte en la vía más recientemente accedida. En el árbol binario esto significa que se deben invertir todos los *bits* de estados que fueron utilizados para encontrar la vía a ser reemplazada. En este ejemplo, la vía a ser reemplazada es la vía seis y se deben invertir los valores de los *bits* cero, dos y seis.

Este algoritmo funciona actualizando los estados del árbol binario de tal forma que apunten en dirección opuesta a la última vía accedida, por lo tanto, la vía que menos actualice el árbol binario será la vía menos recientemente utilizada. Aunque existe la posibilidad de que diferentes combinaciones de accesos resulten en el reemplazo de una vía diferente a la menos recientemente utilizada, *Tree-LRU* es una aproximación muy cercana a *LRU*.

3.7.2.3.1. Actualización de estado en acierto

Debido a la complejidad de implementar directamente este algoritmo en VHDL, se realizó un programa en Python capaz de generar la tabla de verdad de la actualización del estado del árbol binario en aciertos.

Primero se realizó una función que retorna los *bits* que son utilizados para acceder a una vía específica y los valores para apuntar en dirección opuesta a dicha vía, donde el usuario puede seleccionar la cantidad de vías que el árbol binario posee.

Figura 144. **Código, función *getTouchedBits***

```
def getTouchedBits(way,ways):
    bitstouched = {}
    nodebits = ways - 1 - ways//2
    bitcnt = (way)//2 + nodebits
    mask = 0
    andmask = (1&way)<<bitcnt
    while True:
        mask|= 1<<bitcnt
        tmp = bitcnt
        bitcnt = (bitcnt-1)//2
        if bitcnt >=0:
            andmask|= (1&(~tmp))<<bitcnt
        else:
            break
    for _ in range(ways-1):
        if mask & (1<<_):
            bitstouched[_] = 0 if andmask & (1<<_) else 1

    return bitstouched
```

Fuente: elaboración propia, empleando Geany 1.37.1.

La función *getTouchedBits* es utilizada como base en la función *genprocessHitOut*, la cual se encarga de manipulación de texto para generar código VHDL sintetizable, el cual contiene la tabla de verdad deseada.

Figura 145. **Código, función *genprocessHitOut***

```
def getbinary(size,num):
    n = bin(num)[2:]
    return ('0'*(size-len(n))) + n

def genprocessHitOut(nways,fstream):
    txt = ""
    for a in range(nways):
        dic = getTouchedBits(a,nways)
        txt += f"\t\t\t\twhen \"{getbinary(nways,1<<a)}\" =>\n"
        for bit, value in dic.items():
            txt += f"\t\t\t\t\tHitOutState({bit}) <= '{value}';\n"

    print(_template.format(txt,waybits=nways-1,ways=nways),file=fstream)
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 146. **Código VHDL generado por función *genprocessHitOut***

```
HitOut8:process(InState,Match_vec)
    variable match: std_logic_vector(7 downto 0);
begin
    HitOutState <= InState;
    match := Match_vec;
    case (match) is
        when "00000001" =>
            HitOutState(0) <= '1';
            HitOutState(1) <= '1';
            HitOutState(3) <= '1';
        when "00000010" =>
            HitOutState(0) <= '1';
            HitOutState(1) <= '1';
            HitOutState(3) <= '0';
        when "00000100" =>
            HitOutState(0) <= '1';
            HitOutState(1) <= '0';
            HitOutState(4) <= '1';
        when "00001000" =>
            HitOutState(0) <= '1';
            HitOutState(1) <= '0';
            HitOutState(4) <= '0';
        when "00010000" =>
            HitOutState(0) <= '0';
            HitOutState(2) <= '1';
            HitOutState(5) <= '1';
        when "00100000" =>
            HitOutState(0) <= '0';
            HitOutState(2) <= '1';
            HitOutState(5) <= '0';
        when "01000000" =>
            HitOutState(0) <= '0';
            HitOutState(2) <= '0';
            HitOutState(6) <= '1';
        when "10000000" =>
            HitOutState(0) <= '0';
            HitOutState(2) <= '0';
            HitOutState(6) <= '0';
        when others => HitOutState <= (others => '-');
    end case;
end process;
```

Fuente: elaboración propia, empleando Geany 1.37.1.

El código VHDL generado es la actualización de estado de un árbol binario de ocho vías en un acierto, representado por la señal *HitOutState*, el estado del árbol a actualizar es representado por la señal *InState*. La señal *Match_vec*, es

la concatenación de las señales acierto de cada vía, donde el *bit* menos significativo corresponde a la señal acierto de la vía cero y el *bit* más significativo corresponde a la señal acierto de la vía siete.

3.7.2.3.2. Actualización de estado en fallo

A diferencia de un acierto, la actualización de estado en un fallo es independiente de señales externas, por lo tanto, el estado debe ser leído para determinar los *bits* que deben ser invertidos. Debido a la naturaleza del algoritmo cada *bit* debe ser analizado para determinar las condiciones en las que su valor debe ser invertido.

El *bit* cero por ser la raíz del árbol binario, siempre es utilizado para determinar la vía a ser reemplazada. Por lo tanto, cuando se actualiza el estado en un fallo este *bit* siempre debe ser invertido. El *bit* uno debe ser invertido cuando el *bit* cero posea el valor de cero, esto forma una función *XNOR* entre el *bit* cero y *bit* uno. El *bit* dos debe ser invertido cuando el *bit* cero posee el valor de uno, formando una función *XOR* entre el *bit* cero y *bit* uno.

Para el resto de *bits* se creó la función *getcondsforbit* que retorna las condiciones de inversión, esta función toma como argumentos el bit que se desea obtener su condición de inversión y el número de vías que posee el árbol binario. La función *grenprocessMissOut* se encarga de la manipulación de texto para generar código VHDL sintetizable el cual contiene todas las condiciones de inversión de todos los *bits* de estados del árbol binario.

Figura 147. **Código, función *getcondsforbit* y *grenprocessMissOut***

```
def getcondsforbit(bit,ways):
    for _ in range(ways):
        dic = getTouchedBits(_,ways)
        if bit in dic:
            break
    newdic= {}
    for d_bit, value in dic.items():
        if d_bit < bit:
            newdic[d_bit] = value
    return newdic

def grenprocessMissOut(nways,fstream):
    if nways == 4:
        print(_templateMiss.format(ways=4,vars='',v_asign='',\
            conds=''),file=fstream)
    else:
        vars=''
        v_asign=''
        conds=''
        for statusbit in range(3,nways-1):
            dic = getcondsforbit(statusbit,nways)
            t_asign=''
            cond=''
            for bit in sorted(dic):
                t_asign+= f" I({bit}) &"
                cond+= f"{(~dic[bit])&1}"

            vars+= f"\t\t\tvariable cond{statusbit}:\
                std_logic_vector({(len(dic)-1)} downto 0);\n"
            v_asign+= f"\t\t\tcond{statusbit}:={t_asign[:-2]};\n"
            conds += _templatecond.format(bit=statusbit,val=cond)

        print(_templateMiss.format(ways=nways,vars=vars,v_asign=v_asign,\
            conds=conds),file=fstream)
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 148. **Código VHDL generado por función *grenprocessMissOut***

```
MissOut8:process(Instate)
    alias I: std_logic_vector(WAYS-2 downto 0) is InState;
    variable cond3: std_logic_vector(1 downto 0);
    variable cond4: std_logic_vector(1 downto 0);
    variable cond5: std_logic_vector(1 downto 0);
    variable cond6: std_logic_vector(1 downto 0);
begin
    cond3:= I(0) & I(1);
    cond4:= I(0) & I(1);
    cond5:= I(0) & I(2);
    cond6:= I(0) & I(2);
    MissOutState(0) <= not I(0);
    MissOutState(1) <= I(0) xnor I(1);
    MissOutState(2) <= I(0) xor I(2);
    if cond3 = b"00" then MissOutState(3)<= not I(3);
    else MissOutState(3)<= I(3); end if;
    if cond4 = b"01" then MissOutState(4)<= not I(4);
    else MissOutState(4)<= I(4); end if;
    if cond5 = b"10" then MissOutState(5)<= not I(5);
    else MissOutState(5)<= I(5); end if;
    if cond6 = b"11" then MissOutState(6)<= not I(6);
    else MissOutState(6)<= I(6); end if;
end process;
```

Fuente: elaboración propia, empleando Geany 1.37.1.

3.7.2.3.3. Determinar vía a reemplazar

Se creó la función *genProcessVictim*, la cual trabaja basada en la observación que la selección del valor de un *bit* de la señal *Vía* es dependiente del valor de los *bits* de estado de un nivel en específico. Por ejemplo, en un árbol binario de ocho vías, el *bit* más significativo de la señal *Vía* depende del nivel cero del árbol binario, es decir del *bit* cero de estado. El *bit* menos significativo depende del nivel más bajo en el árbol, es decir el nivel dos, que contiene los *bits* tres al seis del estado. Para decidir cuál de los *bits* de estado de un nivel debe ser utilizado para asignar el valor a la señal *Vía* se utiliza su condición de inversión, la cual indica si un *bit* en el árbol de estado ha sido utilizado al momento de seleccionar una vía a reemplazar.

Figura 149. **Código, función *genProcessVictim***

```
def genProcessVictim(nways,fstream):
    s = ('I({})\t','not I({})')
    treedepth = int(log2(nways))
    logic = ''
    vardeclaration = ''
    assign = ''
    for lvl in range(1,treedepth):
        vardeclaration+= f"\t\t\tvariable v{treedepth-1-lvl}: std_logic:='0';\n"
        assign+= f'v{treedepth-1-lvl} & '
        for bit in range((2**lvl)-1,(2**(lvl+1)-1)):
            conds = getcondsforbit(bit,nways)
            logic+= f'\t\t\tv{treedepth-1-lvl}:= v{treedepth-1-lvl} or '
            for condbit in sorted(conds):
                logic+= s[conds[condbit]].format(condbit) + '\tand\t'
            logic+= f'I({bit});\n'
        logic+= '\n'
    assign = f'\t\t\tvia <= v{treedepth-1} & {assign[:-3]};\n'
    print(_templateVictim.format(ways=nways,msb=treedepth-1,\
                                 vars=vardeclaration,logic=logic,\
                                 assign=assign),file=fstream)
```

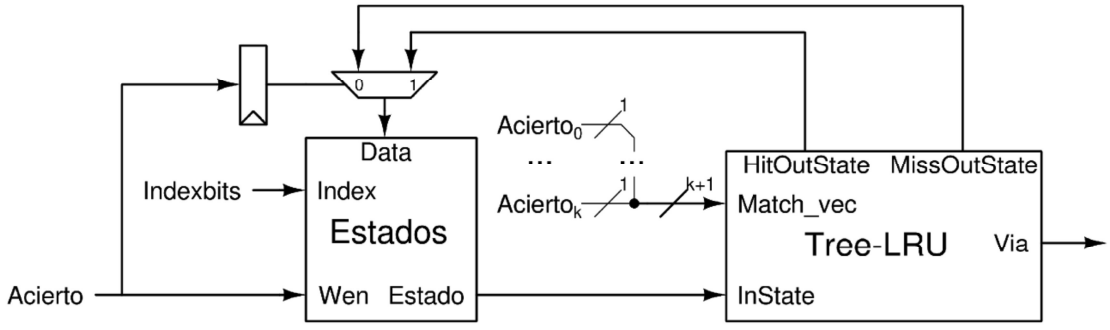
Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 150. **Código VHDL generado por función *genProcessVictim***

```
Victim8:process(InState)
    alias I: std_logic_vector(WAYS-2 downto 0) is InState;
    variable v2: std_logic;
    variable v1: std_logic:='0';
    variable v0: std_logic:='0';
begin
    v2:= I(0);

    v1:= v1 or not I(0) and I(1);
    v1:= v1 or I(0)     and I(2);

    v0:= v0 or not I(0) and not I(1)    and I(3);
    v0:= v0 or not I(0) and I(1)        and I(4);
    v0:= v0 or I(0)     and not I(2)    and I(5);
    v0:= v0 or I(0)     and I(2)        and I(6);

    via <= v2 & v1 & v0;
end process;
```

Fuente: elaboración propia, empleando Geany 1.37.1.

Figura 151. ***Tree-LRU* como política de reemplazo**



Fuente: elaboración propia, empleando Xcircuit v3.10.

La escritura en la memoria de estados es habilitada cuando el caché acierta en un acceso a memoria, debido a que los estados deben ser actualizados en cada acceso. Para determinar qué valor debe ser utilizado para actualizar el estado, se utiliza un registro para almacenar el valor de *Acierto*, este registro posee el valor de *Acierto* un ciclo de reloj atrás.

Cuando el registro tenga el valor de cero y *Acierto* posea el valor de uno, significa que en el ciclo anterior se realizó la última porción del proceso de reemplazo, por lo tanto, se tuvo un acceso fallido y se debe seleccionar el valor de la señal *MissOutState*. Si el registro posee el valor de uno al igual que la señal *Acierto*, en el ciclo anterior no hubo un reemplazo, por lo tanto, el acceso resultó en un acierto y se debe seleccionar el valor de la señal *HitOutState*. Si la señal *Acierto* posee el valor de cero, se deshabilita la escritura de la memoria de estados, lo que hace imposible la modificación del estado durante el reemplazo de una línea, de tal forma el valor de *Vía* es constante durante todo el proceso.

3.8. Resultados

El código fuente en VHDL de las memorias caché presentadas puede ser obtenido[27].

3.8.1. Efectos de parámetros de diseño en porcentaje de aciertos

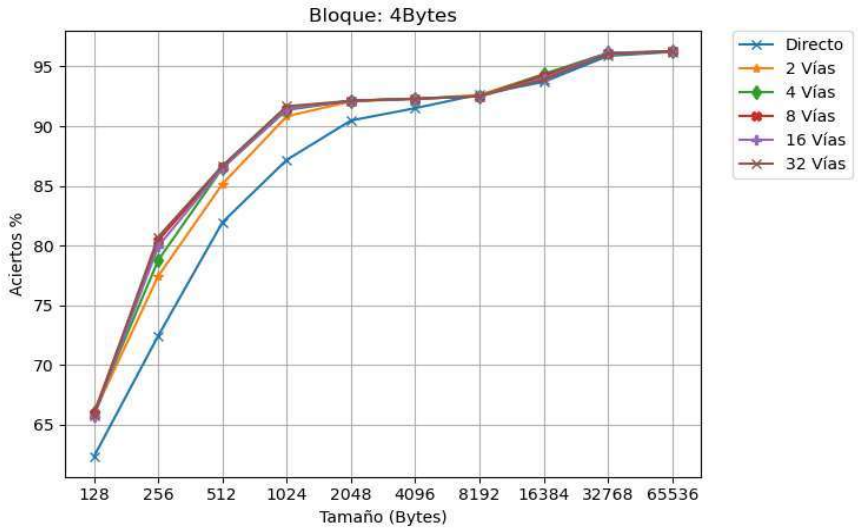
Para observar los efectos de los diferentes parámetros en el porcentaje de aciertos, se creó un simulador en el lenguaje Python que lee los trazos de ejecución de los diferentes programas y simula el comportamiento del caché descrito en este trabajo, además posee un registro de todas las veces que un acceso a memoria resultó en fallo que es utiliza en conjunto con el número total de accesos para obtener el porcentaje de aciertos de una configuración específica. El código fuente y los datos generados[28].

Las gráficas presentadas utilizan los valores promedio entre los diez programas.

[27] SIERRA, Ottoniel. *RV32I/hardware/cache*. https://github.com/oasm95/RV32I/tree/main/hardware/cache. Consulta: agosto 2021.

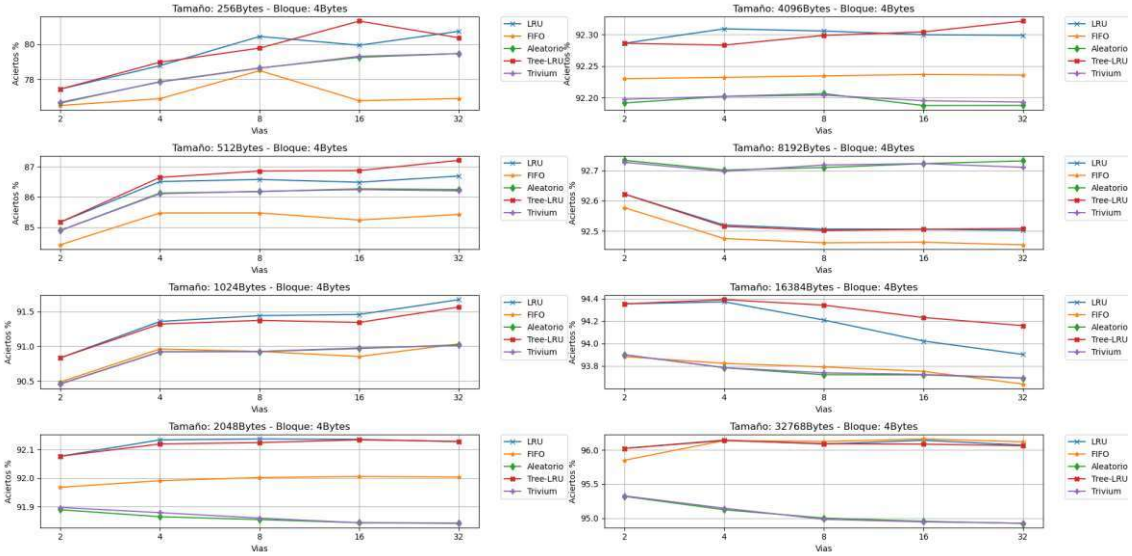
[28] Ibíd.

Figura 152. **Tamaño vs., porcentaje de aciertos**



Fuente: elaboración propia, empleando Python 3.

Figura 153. **Número de vías vs., porcentaje de aciertos**

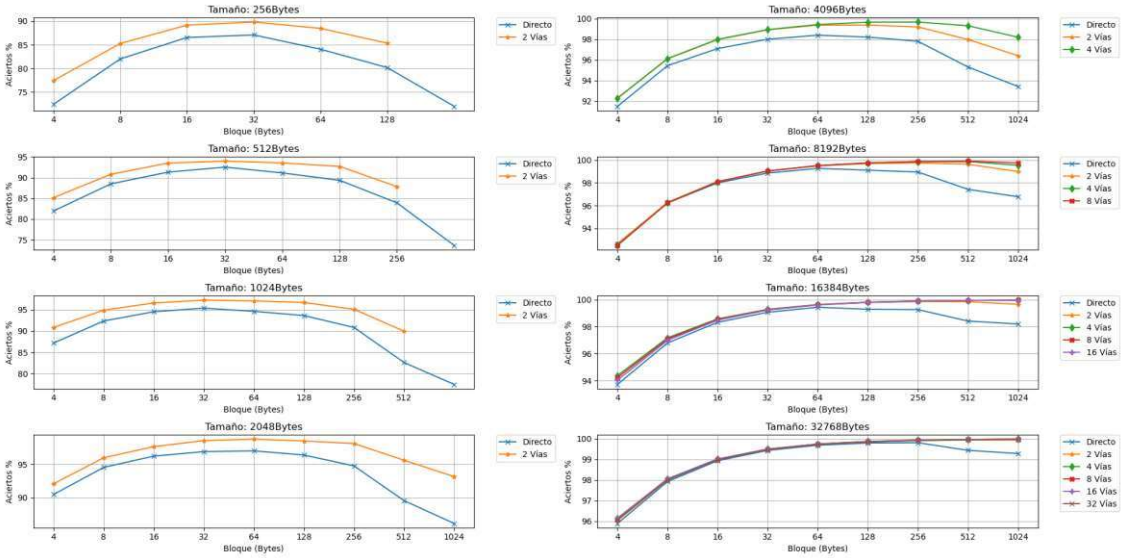


Fuente: elaboración propia, empleando Python 3.

En estas gráficas se incluyen los diferentes tipos de políticas de reemplazo y entre ellas se encuentran *LRU* y aleatorio, con el objetivo de ser comparados con el porcentaje de aciertos de *Tree-LRU* y Trivium, respectivamente. Aleatorio es la generación de números aleatorios a través de software utilizando la función en Python "*numpy.randint*". *LRU* es la implementación en software que utiliza una estructura de memoria tipo pila.

Trivium y aleatorio poseen valores muy cercanos con mínimas variaciones en su porcentaje de aciertos. *LRU* y *Tree-LRU* presentan diferencias más obvias, pero no significativas, adicionalmente ambas se encuentran entre las dos mejores políticas de reemplazo. Con lo que se puede confirmar que *Tree-LRU* es una buena aproximación a *LRU*.

Figura 154. **Tamaño de bloque vs., porcentaje de aciertos**



Fuente: elaboración propia, empleando Python 3.

El tamaño de bloque muestra ser el parámetro que produce un mayor impacto positivo en el porcentaje de aciertos. Una ventaja de modificar este parámetro es que, al ser una reestructuración de la manera de almacenar datos, cambiar este parámetro no significa un aumento significativo en la utilización de recursos.

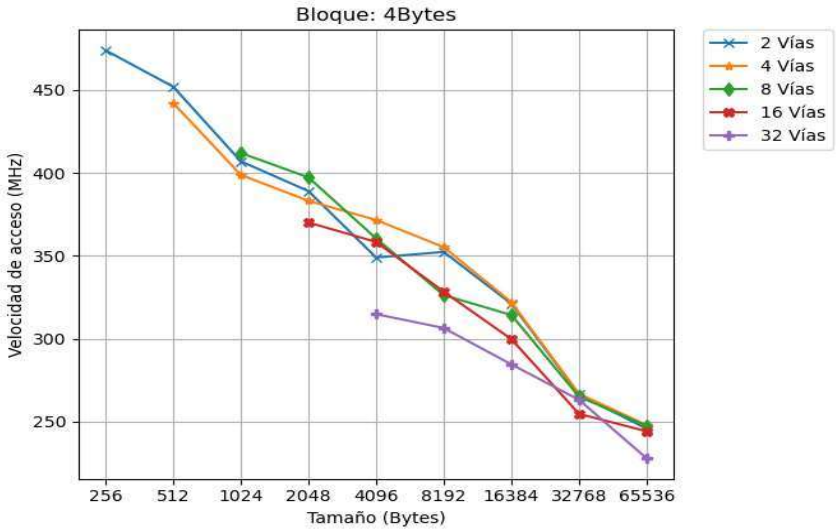
Una de las tendencias que se pueden observar es que al aumentar el tamaño del cache disminuye el impacto de la asociatividad, es decir el número de vías. Esto puede deberse a que al aumentar la capacidad del caché hay más líneas donde pueden ser mapeados los datos, reduciendo los conflictos entre localidades de memoria. En los tamaños que la asociatividad es efectiva, se puede observar que el mapeo directo presenta el menor porcentaje de aciertos.

3.8.2. Efectos de parámetros de diseño en velocidad de acceso

Todos los valores de velocidad de acceso[29] fueron obtenidos de la herramienta CACTI.

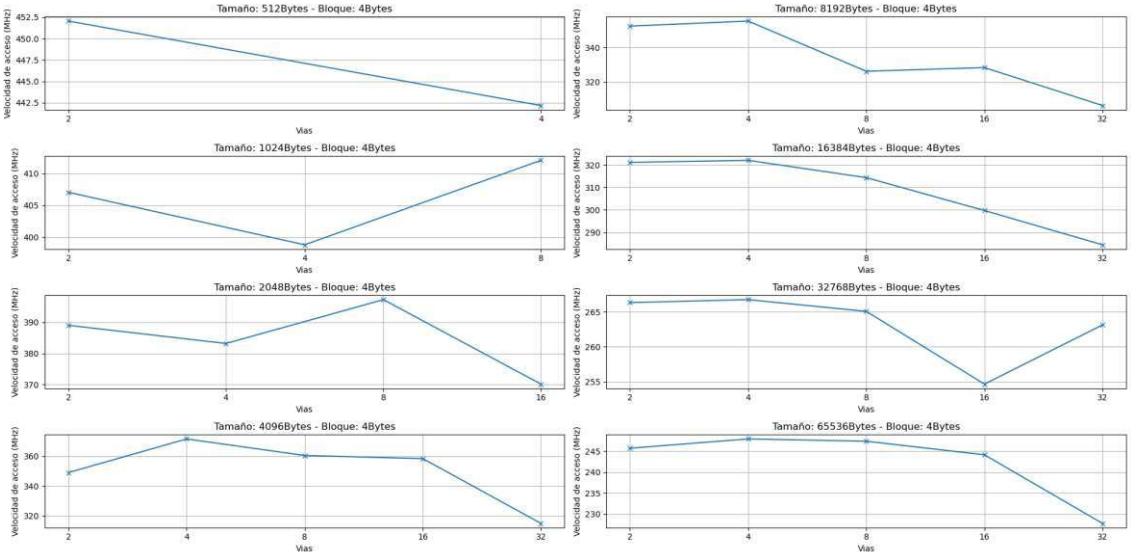
[29] SIERRA, Ottoniel. *RV32I/cacti*. https://github.com/oasm95/RV32I/tree/main/cacti. Consulta: agosto 2021.

Figura 155. **Tamaño vs., velocidad de acceso**



Fuente: elaboración propia, empleando CACTI 7 y Python 3.

Figura 156. **Número de vías vs., velocidad de acceso**

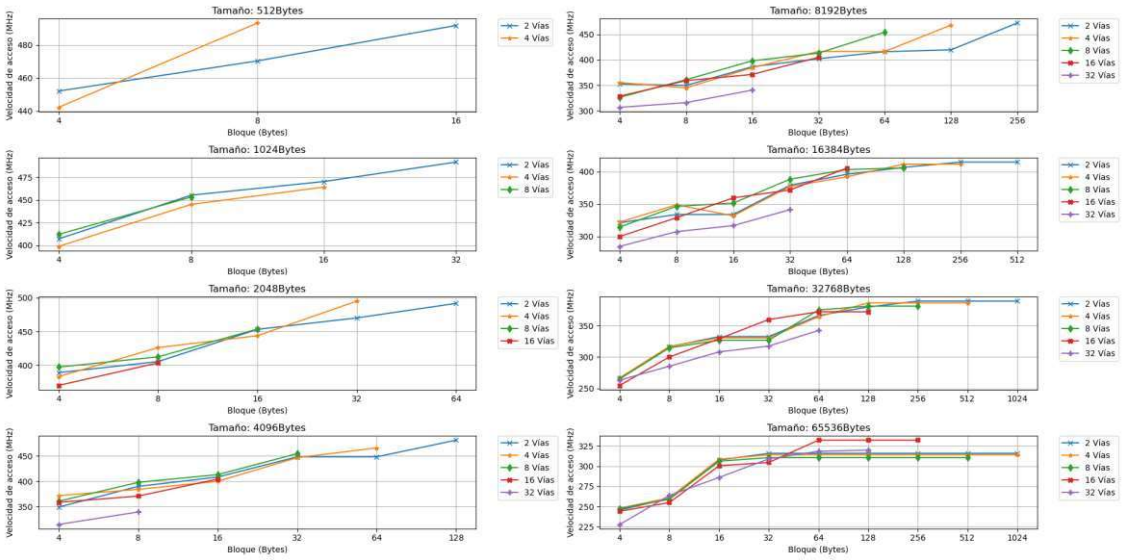


Fuente: elaboración propia, empleando CACTI 7 y Python 3.

La tendencia general en el tamaño es que al aumentar este parámetro disminuye la velocidad que pueden ser extraídos los valores del caché.

El número de vías aparenta poseer una tendencia cuadrática negativa, es decir que posee un punto máximo y al alejarse de este punto se reduce la velocidad de acceso. Este punto aparenta estar entre cuatro y ocho vías.

Figura 157. **Tamaño de bloque vs., velocidad de acceso**



Fuente: elaboración propia, empleando CACTI 7 y Python 3.

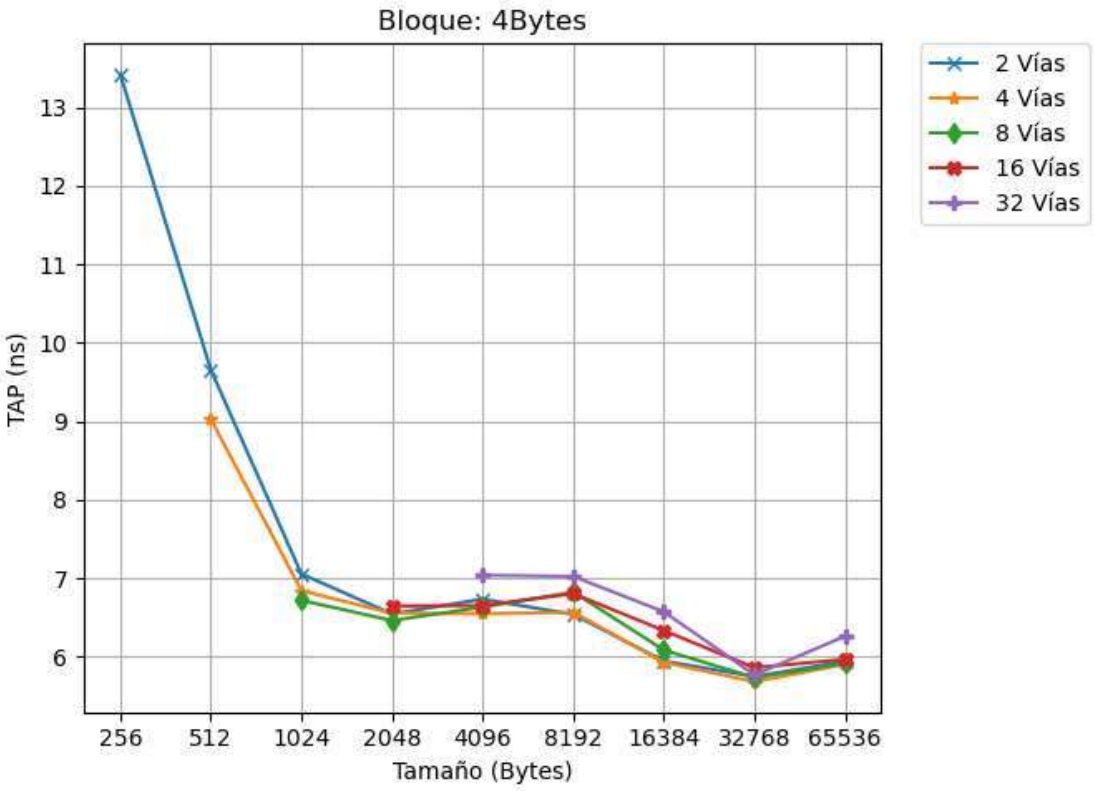
A diferencia del tamaño del caché, al aumentar el tamaño del bloque aumenta la velocidad en la que los datos pueden ser extraídos.

3.8.3. Efectos de parámetros de diseño en Tiempo de Acceso Promedio, TAP

Para calcular el tiempo de acceso promedio son necesarios tres valores, el tiempo de acierto, que es el inverso de la velocidad de acceso, el porcentaje de fallos y el tiempo de fallo. Este último valor depende del nivel superior en la jerarquía de memoria. Para este trabajo se asume que el nivel superior es la memoria principal.

El tiempo de fallo consiste en una transacción que inicia con un retardo de once ciclos de reloj antes de obtener el primer valor, donde se realizan cuatro transacciones en ráfaga, es decir cada valor es recibido un ciclo de reloj después de otro, cada transacción es de un tamaño de ocho *bytes*. Esto se hizo como una sobre-simplificación de una transacción a una memoria *DDR SDRAM*, los primeros once ciclos son el equivalente a la escritura de comandos de escritura/lectura, envió de dirección, el tiempo de viaje de los datos, entre otros.

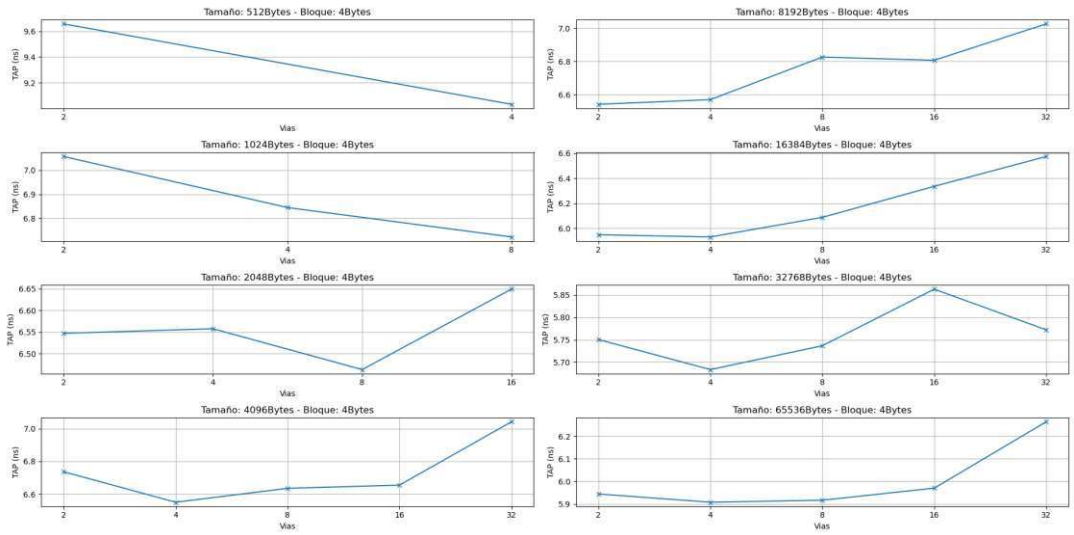
Figura 158. **Tamaño vs., TAP**



Fuente: elaboración propia, empleando CACTI 7 y Python 3.

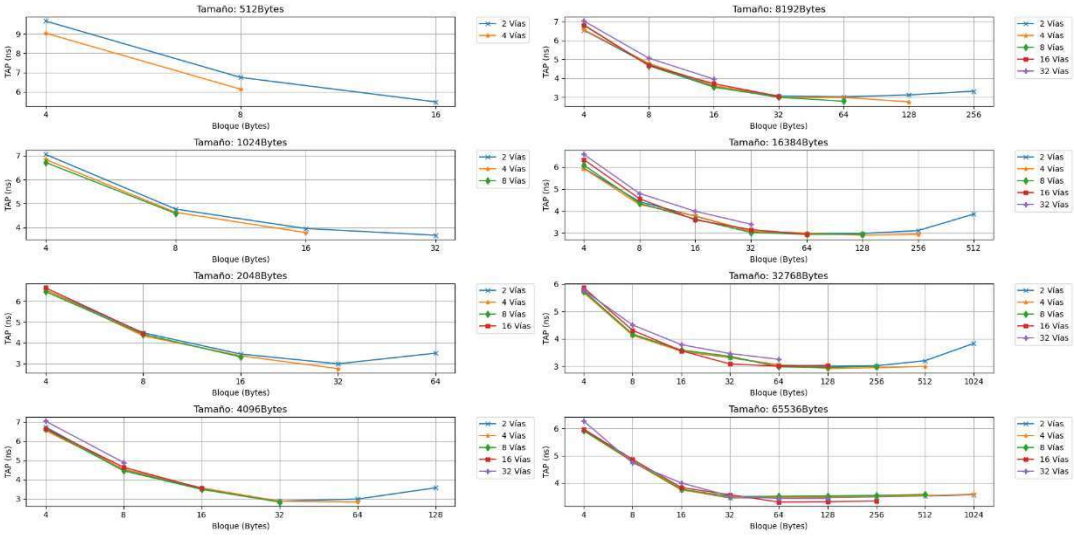
A pesar de que a menor tamaño la velocidad de acceso es mayor, el tiempo de acceso promedio es mayor, esto se debe al bajo índice de aciertos en memorias con menor capacidad, provocando una mayor cantidad de transferencias a niveles superiores, aumentando el tiempo de acceso promedio. De igual manera, el aumento en el tamaño no parece afectar de manera significativa al tiempo de acceso promedio ya que, aunque la velocidad de acceso disminuye, el porcentaje de aciertos aumenta, lo cual reduce la cantidad de transferencias a niveles superiores.

Figura 159. **Número de vías vs., TAP**



Fuente: elaboración propia, empleando CACTI 7 y Python 3.

Figura 160. **Tamaño de Bloque vs., TAP**



Fuente: elaboración propia, empleando CACTI 7 y Python 3.

Cuando se deba reemplazar una línea, entre más grande sea el tamaño del bloque se requiere una mayor cantidad de transferencias para llenarlo, como consecuencia aumentando el tiempo de fallo.

El tiempo de acceso promedio es mínima en una porción de tamaños de bloques, al alejarse de esta porción el tiempo de acceso promedia aumenta. Similar a la tendencia presente en el porcentaje de aciertos. Esto puede deberse a que al aumentar el tamaño de bloque se reducen la cantidad de líneas de almacenamiento, provocando conflictos entre direcciones, causando una mayor cantidad de reemplazos que resultan en tiempo alto de fallo.

3.8.4. Mejores configuraciones de memorias

A continuación, se presentan las mejores cinco configuraciones para memorias cache de datos e instrucciones.

Tabla XXX. **Mejores cinco configuraciones para caché de datos**

Frecuencia Procesador 338,73 Mhz (2,952 ns)				
TAP (ns)	Politica de Reemplazo	Tamaño (Bytes)	Tamaño de Bloque (Bytes)	Vías
2,756 (<1 Ciclo)	Tree-LRU	8 192	128	4
2,763 (<1 Ciclo)	Tree-LRU	2 048	32	4
2,786 (<1 Ciclo)	FIFO	8 192	128	4
2,787 (<1 Ciclo)	Tree-LRU	8 192	64	8
2,797 (<1 Ciclo)	FIFO	2 048	32	4

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Tabla XXXI. **Mejores cinco configuraciones para cache de instrucciones**

Frecuencia Procesador 338,73 Mhz (2,952 ns)				
TAP (ns)	Politica de Reemplazo	Tamaño (Bytes)	Tamaño de Bloque (Bytes)	Vías
2,315 (<1 Ciclo)	Tree-LRU	8 192	64	8
2,322 (<1 Ciclo)	Trivium	8 192	64	8
2,329 (<1 Ciclo)	FIFO	8 192	64	8
2,474 (<1 Ciclo)	Tree-LRU	8 192	32	8
2,479 (<1 Ciclo)	FIFO	8 192	32	8

Fuente: elaboración propia, empleando FreeOffice PlanMaker 2019.

Se aprecia que el tiempo de acceso promedio es menor que el periodo de reloj del procesador, lo que significa que respecto al procesado en promedio todo acceso a memoria tomará un ciclo. En otras palabras, el procesador en promedio no sufrirá algún retardo por accesos a memoria y en la mayoría del tiempo realizará trabajo útil. De no utilizar una memoria caché el tiempo de acceso promedio sería de once ciclos, es decir el procesador únicamente haría trabajo útil en uno de once ciclos.

Se debe tomar en consideración que estas configuraciones son válidas en respecto a los programas que se utilizaron como referencia y en la configuración del nivel superior en la jerarquía de memoria. Por ejemplo, la configuración ideal de caché cambiaría si se probaran en aplicaciones de base de datos, sistemas de tiempo real, entre otros. Por lo tanto, se recomienda realizar las pruebas y simulaciones necesarias con el tipo de aplicación que se desea utilizar para obtener la configuración óptima.

CONCLUSIONES

1. Al aplicar *pipeline* de cinco etapas, se identificaron tres tipos diferentes de riesgos: riesgos estructurales, riesgos de datos y riesgos de control.

2. Los riesgos estructurales fueron mitigados al duplicar los circuitos del módulo *CSR*, específicamente los puertos de escritura y lectura.

3. Los riesgos de datos fueron mitigados al implementar traspaso de datos entre diferentes etapas en el *pipeline*. Para los riesgos en que el traspaso no es suficiente, se detiene la ejecución del procesador hasta ser resuelto el riesgo de datos.

4. Los riesgos de control fueron mitigados al implementar especulación estática y dinámica, en conjunto de un circuito de limpieza de etapas.

5. Se logró implementar en VHDL un procesador *RISC* de treinta y dos *bits*, el cual fue utilizado para obtener cantidad de ciclos de ejecución y rastros de ejecución de diferentes programas.

6. Utilizando GHDL como simulador de descripciones de hardware escritas en VHDL, se ejecutaron diez diferentes programas auto verificables, utilizados para comprobar el correcto funcionamiento del procesador.

7. Respecto a la librería de células estándar *osu035* se aumentó la frecuencia de reloj de 95,85 MHz a 189,35 MHz.

8. Respecto a la librería de células estándar *osu018*, se aumentó la frecuencia de reloj de 151,55 MHz a 338,73 MHz.

9. Al aplicar *pipeline* de cinco etapas al procesador se logró un aumento de desempeño dentro del rango de 18,43 % al 44,65 % utilizando la librería de células estándar *osu035* y un aumento en el desempeño entre 33,98 % y 63,65 % utilizando la librería de células estándar *osu018*.

10. Al aplicar *pipeline* de cinco etapas y agregar un predictor de saltos dinámico de dos *bits* con capacidad para 256 instrucciones se logró un aumento de desempeño dentro del rango de 45,19 % al 79,91 % utilizando la librería de células estándar *osu035* y un aumento en el desempeño entre 64,27 % y 103,55 % utilizando la librería de células estándar *osu018*.

11. Al utilizar memorias caché se redujo el tiempo promedio de acceso a 2,75 ns en caché de datos y 2,31 ns en caché de instrucciones, ambas son menores al periodo de reloj del procesador. Por lo tanto, se eliminó el efecto negativo por la diferencia entre velocidad del procesador y memoria.

12. La configuración óptima de parámetros en una memoria caché de datos respecto a los programas de prueba utilizados en este trabajo, es un cache conjunto asociativo de 4 vías, utiliza política de reemplazo *Tree-LRU*, tiene un tamaño de 8 192 *bytes* y posee un tamaño de bloque de 128 *bytes*.

13. La configuración óptima de parámetros en una memoria caché de instrucciones respecto a los programas de prueba utilizados en este trabajo es un cache conjunto asociativo de 8 vías, utiliza política de

reemplazo *Tree-LRU*, tiene un tamaño de 8 192 *bytes* y posee un tamaño de bloque de 64 *bytes*.

RECOMENDACIONES

1. Tomar en consideración otros factores como el precio, cantidad de recursos, tamaño, eficiencia energética, al realizar el diseño de un procesador, dependiendo de los requerimientos de la implementación.

2. La configuración óptima de una memoria caché es dependiente del tipo de aplicación que ejecuta un procesador, por lo tanto, las pruebas y simulaciones deben ser ejecutadas utilizando el tipo de aplicación deseada.

3. Implementar *pipeline* procurando conectar la menor cantidad de componentes en serie por etapa, ya que al calcular periodo de reloj se utiliza la etapa con el mayor tiempo de propagación.

4. Sintetizar en una *FPGA* un procesador destinado a obtener rastros de ejecución, ya que al utilizar un simulador recabar esta información toma una considerable cantidad de tiempo.

5. Implementar un simulador en software de estructuras como un predictor de saltos o una memoria caché, debido a su alta demanda computacional, utilizar lenguajes de programación compilados como C o C++, ya que el tiempo que es ahorrado en su implementación en un lenguaje interpretado de alto nivel como Python, es eliminado al momento de ejecutar el simulador.

6. Utilizar el lenguaje de descripción de *hardware* Verilog, debido a la alta disponibilidad de herramientas de uso libre que soportan su uso. A diferencia de VHDL.

BIBLIOGRAFÍA

1. CARVALHO, Carlos. *The gap between processor and memory speeds*. [en linea]. <http://gec.di.uminho.pt/discip/minf/ac0102/1000gap_proc-mem_speed.pdf>. [Consulta: 18 de junio de 2020].

2. DE CANNIÉRE, Christophe; PRENEEL, Bart. *Trivium, A Stream Cipher Construction Inspired by Block Cipher Design Principles* [en linea]. <https://www.ecrypt.eu.org/stream/papersdir/2006/021.pdf>. [Consulta: enero de 2021].

3. _____. *Trivium, Specifications* [en linea]. <https://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf>. [Consulta: enero de 2021].

4. DE GELAS, Johan. *Decoding Instructions - Intel Core versus AMD's K8 architecture.* [en linea]. <https://www.anandtech.com/show/1998/3>. [Consulta: octubre de 2020].

5. EDWARDS, Timothy. *Qflow* [en linea]. <http://opencircuitdesign.com/qflow/>. [Consulta: octubre de 2020].

6. GHDL. *1.0-dev documentation*. [en linea]. <https://ghdl.github.io/ghdl/about.html>. [Consulta: julio de 2020].

7. GUILLE, Damien. *Study of Different Cache Line Replacement Algorithms in Embedded Systems*. Suecia: Master Thesis. Kungliga Tekniska Högskolan University. 2007. 94 p.

8. HENESSY, Jhon L.; PATTERSON, David A. *Computer Architecture A Quantitative Aproach*. 5a ed. Estados Unidos: Elsevier. 2012. 857 p.

9. HEWLETTPACKARD. *Cacti.* [en linea]. <https://github.com/HewlettPackard/cacti>. [Consulta: enero de 2021].

10. HP LABS. *Cacti.* [en linea]. <https://hpl.hp.com/research/cacti/>. [Consulta: enero de 2021].

11. KEIL.COM. *The Dhrystone Benchmark* [en linea] <https://www.keil.com/benchmarks/dhrystone.asp> [Consulta: agosto de 2020].

12. MORRIS, Mano, M. *Diseño Digital*. 3a ed. Mexico: Pearson Educación. 2003. 538 p.

13. NAIR, Ravi. *Optimal 2-bit branch predictors*. Estados Unidos: IEEE Transactions on Computers, vol. 44, no. 5. 1995. 5 p.

14. PATTERSON, David, ANDERSON, Thomas. et al. Damien. *A Case for Intelligent RAM*. Estados Unidos: IEEE Micro. 1997. 10 p.

15. RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*. Estados Unidos: Editores Andrew Waterman y Krste Asanović. 2019. 238 p.

16. RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*. Estados Unidos: Editores Andrew Waterman y Krste Asanović. 2019. 135 p.

17. SHEN, Jhon Paul, LIPASTI, Mikko H. *Modern Processor Design: Fundamentals of Superscalar Processors.* Estados Unidos: 1a ed. John Paul Shen y Mikko H. Lipasti. 2005. 658 p.

18. SRINIVASAN, Viji. *Optimizing pipelines for power and performance.* 35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings., Istanbul, Turkey, 2002. 11 p.

19. TERMAN, Chris. *6.004 Computation Structures*. [en línea]. <https://computationstructures.org/>. [Consulta: junio 2020].