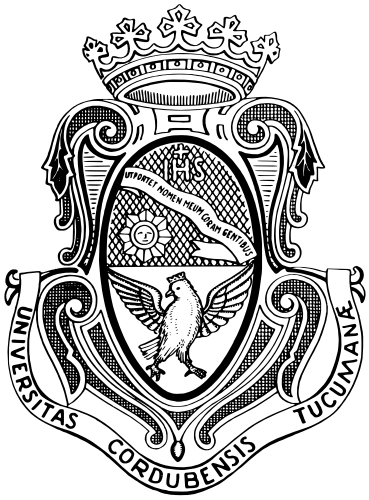
Universidad Nacional de Córdoba



Trabajo especial de la
Licenciatura en Ciencias de la Computación

Inyección de fallas en procesadores RISC-V para caracterizar nodos DTN

Autor: Edelstein Adrian Marcelo *Director: Dr. Ferreyra Pablo*

Facultad de Matemática, Astronomía, Física y Computación

septiembre de 2023

Resumen

Los procesadores RISC-V de código abierto se están popularizando cada vez más aceleradamente en diversos campos de aplicación. El hecho de que son procesadores de arquitectura y de código abierto permite implementar diversas formas de caracterizar la confiabilidad y disponibilidad de los sistemas basados en ellos.

Una forma de caracterización se basa en la inyección de fallas, técnica que se aplica ampliamente para evaluar la sensibilidad de los circuitos integrados frente a los efectos de la radiación. En este caso nos interesa simular alteraciones aleatorias de la información (AAI) almacenada en los registros internos del procesador.

En este trabajo se propone desarrollar primero una implementación simplificada de un procesador RISC-V. Luego planteamos el desarrollo de una herramienta de inyección de fallas del tipo AAI en sus registros internos. El parámetro que se busca obtener es la tasa de fallas que permite, mediante simples cálculos, obtener la confiabilidad del sistema, asumiendo ciertas condiciones de base. El siguiente paso consiste en aplicar dicha herramienta para caracterizar algoritmos simples programados en el procesador RISC-V previamente desarrollado para poder verificar el correcto funcionamiento de dicho inyector. El último paso del trabajo consiste en aplicar la herramienta de inyección de fallas para caracterizar un algoritmo para la generación de tablas de ruteo propios de un nodo DTN.

La relevancia de este trabajo radica en el uso creciente de los nodos DTN como módulos constructivos básicos para el desarrollo de Servicios y Sistemas Espaciales Distribuidos.

Índice de figuras

Índice de tablas

Índice general

Glosario

ALU Arithmetic/Logic Unit

BRAM Block RAM

CLB Configurable Logic Block

CSR Control and Status Register

DTN Delay Tolerant Network

DUT Device Under Test

DUT Instruction Set Architecture

FF Flip Flop

FPGA Field-Programmable Gate Array

GPR General Purpose Register

JTAG Joint Test Action Group

LSB Least Significant Bit

LUT Lookup Table

PC Program Counter

RAM Random Access Memory

ROM Read Only Memory

SEE Single Event Effect

SEL Single Event Latchup

SET Single Event Transient

SEU Single Event Upset

UART Universal Asynchronous Receiver Transmitter

VHDL VHSIC Hardware Description Language

Capítulo 1

Introducción

1.1. Motivación

Los procesadores RISC-V de código abierto se están popularizando cada vez más aceleradamente en diversos campos de aplicación, tales como electrodomésticos y sistemas de automatización [2–5]. En particular su uso en sistemas espaciales es cada vez más común [6]. Dichos sistemas exigen un cierto nivel de tolerancia a fallas para poder soportar la hostilidad del ambiente espacial. El hecho de que son procesadores de arquitectura y de código abierto permite implementar diversas formas de caracterizar la confiabilidad y disponibilidad de los sistemas basados en ellos. Una de estas formas de caracterización se basa en la inyección de fallas para simular alteraciones aleatorias de la información (AAI) almacenada en los registros internos del procesador. Diversos trabajos tratan sobre la forma de inyectar este tipo de fallas en procesadores [7–10]. En este trabajo se presenta el desarrollo de una implementación simplificada de un procesador RISC-V y una herramienta para simular AAI en el mismo. Para ello se utiliza una plataforma de desarrollo basada en FPGA, de manera que el procesador es descripto en un lenguaje de descripción de hardware y luego embebido en dicha plataforma. El siguiente paso realizado en este trabajo consistió en el desarrollo de una herramienta de inyección de fallas del tipo AAI para los registros internos del procesador. El uso de la FPGA y sus recursos internos facilitan mucho el trabajo. De allí la justificación de utilizar una plataforma basada en una FPGA. La herramienta de inyección de fallas tiene por objetivo obtener un parámetro conocido en la literatura como tasa de error. Dicho parámetro a su vez permite, mediante la aplicación de simples cálculos obtener la confiabilidad del sistema bajo estudio, asumiendo ciertas condiciones de contorno [11]. El siguiente paso realizado en este trabajo consistió en la aplicación de la herramienta de inyección de fallas para caracterizar algoritmos simples programados en el procesador RISC-V previamente desarrollado. Con ello se pudo verificar el correcto funcionamiento tanto cualitativo como cuantitativo de dicho inyector. El objetivo final planteado para este trabajo consistió en la aplicación de la herramienta de inyección de fallas para caracterizar un algoritmo para la generación de tablas de ruteo propios de un nodo DTN. La relevancia de este trabajo radica en el uso creciente de los nodos DTN como módulos constructivos básicos para el desarrollo de Servicios y Sistemas Espaciales Distribuidos.

En efecto, las Redes Tolerantes a Demoras, (DTNs) [12–18] por sus siglas en inglés, son hoy en día un campo muy activo de investigación y desarrollo. Las DTNs son principalmente concebidas para largas distancias entre nodos y en escenarios donde otros servicios de redes regulares son altamente disruptivas o no pueden aplicarse debido a la falta de la infraestructura requerida, [19–21].

1.2. Organización y Contenido

En el capitulo 2 se introduce la plataforma de hardware sobre la que se trabajo, RISC-V y el diseño del procesador trabajado. En el capitulo 3 se discuten distintas metodologías de simulación de fallas tipo AAI y se presenta la forma desarrollada en este trabajo y la manera en que se integra con el procesador mencionado. En el capitulo 4 se presentan una serie de casos de estudio utilizados en el desarrollo y verificación experimental del diseño del procesador y la herramienta de AAI. Los primeros dos casos son algoritmos simples que utilizamos para estudiar y verificar el correcto funcionamiento de la herramienta de AAI y el tercero es nuestro caso de estudio principal, siendo este un algoritmo de enrutamiento de una DTN particular que estudiamos para ver como se comporta el procesador implementado mientras esta corriendo un programa mas complejo dentro de un entorno hostil emulado y verificar su robustez. Por ultimo se cierra el trabajo con una breve conclusión.

Capítulo 2

Preliminares de hardware y software

2.1. FPGA

Una matriz de puertas lógicas programable en campo o FPGA por sus siglas en inglés, es un dispositivo programable que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada en el momento mediante un lenguaje de descripción de hardware como SystemVerilog o VHDL. Como muestra la Fig. 2.1, dichos bloques de lógica configurable (CLB) típicamente consisten en una tabla de búsqueda (LUT) y un flip-flop tipo D que permite implementar lógica secuencial pero a su vez también pueden usarse con otros fines para incrementar la velocidad de ciertos diseños, como por ejemplo implementar *"pipelines"* en sumadores que permiten optimizar la velocidad de los mismos.

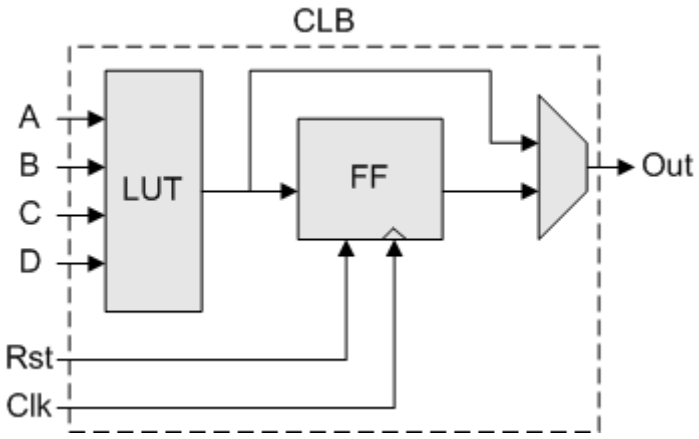


Figura 2.1: Diagrama de un bloque lógico configurable básico.

Implementar nuestro diseño en una FPGA permite que podamos iterar sobre el diseño del hardware con mayor facilidad, velocidad y menor costo. Sin embargo también existen varias limitaciones que hacen que no siempre sea conveniente diseñar utilizando exclusivamente una FPGA. Una limitación en particular es si nuestro diseño contiene una memoria grande o necesita ser optimizado en la relación entre su velocidad y su consumo. Haciendo uso de CLBs podríamos hacer la implementación de una *memoria de acceso aleatorio* (RAM) usando los flip-flops presentes. La implementación de una *memoria*

de solo lectura (ROM) se podría hacer haciendo uso de LUTs. Sin embargo, en ambos casos la solución no es fácilmente escalable ya que requiere de un gran numero de CLBs para implementar una memoria de tamaño limitado. Además en caso de implementarse desaprovecharía muchos recursos de la FPGA y tendría un consumo muy elevado y una performance muy baja. En resumen y en general, si se necesita una memoria ROM o RAM grande, deberá ser externa a la FPGA. Para combatir este problema, las FPGAs modernas proveen bloques de memoria RAM, a las que se conoce como *memoria de acceso aleatorio de bloque* (BRAM). Por otro lado, el uso de BRAM en nuestro diseño permite que el proceso de sintetización, es decir la traducción de nuestro lenguaje de descripción de hardware a una secuencia de bits que se cargan en la FPGA para su configuración, se acelere por varios ordenes de magnitud ya que simplifica la interconexión entre los componentes. Sin embargo, para los estándares modernos estas memorias son limitadas en su capacidad, ya que los sistemas operativos actuales requieren grandes espacios de memorias. Por lo tanto el uso de BRAM soluciona parcialmente la necesidad de memoria en sistemas basados en un microprocesador, proscribiendo en general a la mayoría de los sistemas operativos actuales. Es por esto que en las placas de desarrollo además de una FPGA, también existen otros componentes como por ejemplo una memoria de acceso aleatorio dinámica o puertos de entrada/salida. De todos modos se requiere de descripciones de hardware adicionales para que la FPGA pueda interactuar con estos componentes.

Para implementar el *"softcore"* del procesador se decidió optar por utilizar la placa DE0_NANO [22] fabricada por Terasic que contiene una FPGA Cyclone IV de Intel (previamente Altera). En las Fig. 2.2 y 2.3 podemos observar los distintos componentes presentes en la placa de desarrollo utilizada.

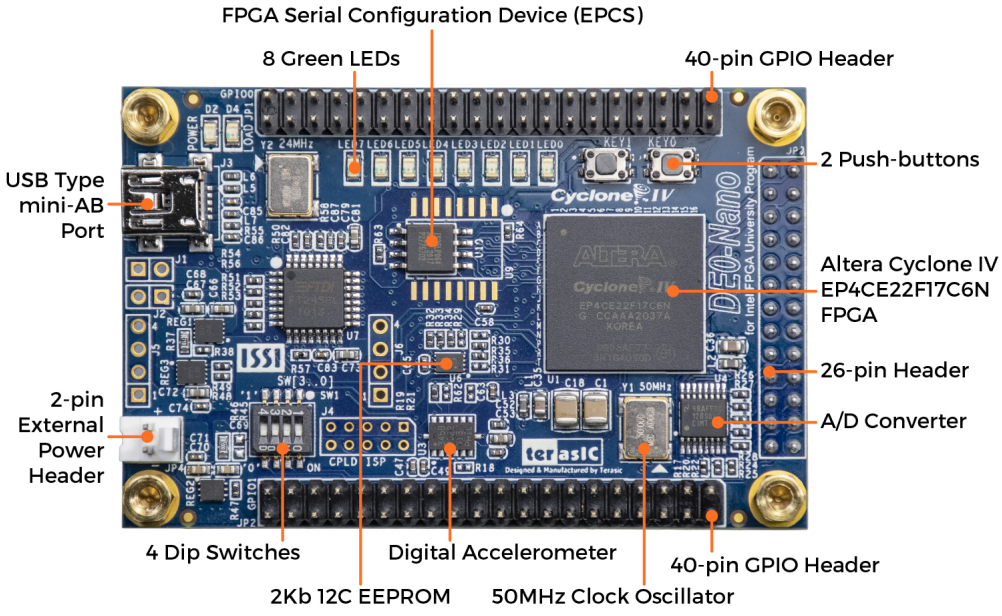


Figura 2.2: Organización de la placa de desarrollo DE0_NANO (Vista superior)

Dicha FPGA (modelo EP4CE22F17C6N) contiene 22320 elementos lógicos, 594 Kbits de memoria embebida, 66 multiplicadores embebidos 18x18, 4 PLL de propósito

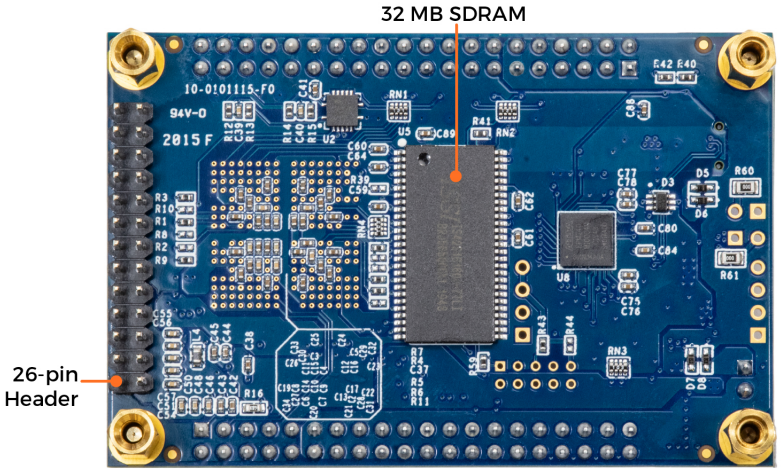


Figura 2.3: Organización de la placa de desarrollo DE0_NANO (Vista inferior)

general y un máximo de 153 pines para entrada/salida [23]. En la Fig. 2.4 se muestra como se conectan los distintos componentes de la placa de desarrollo con la FPGA. De esta forma, se le permite al usuario la mayor flexibilidad para implementar el diseño que necesite. La razón por la cual se opto por este dispositivo descansa en el hecho de que se

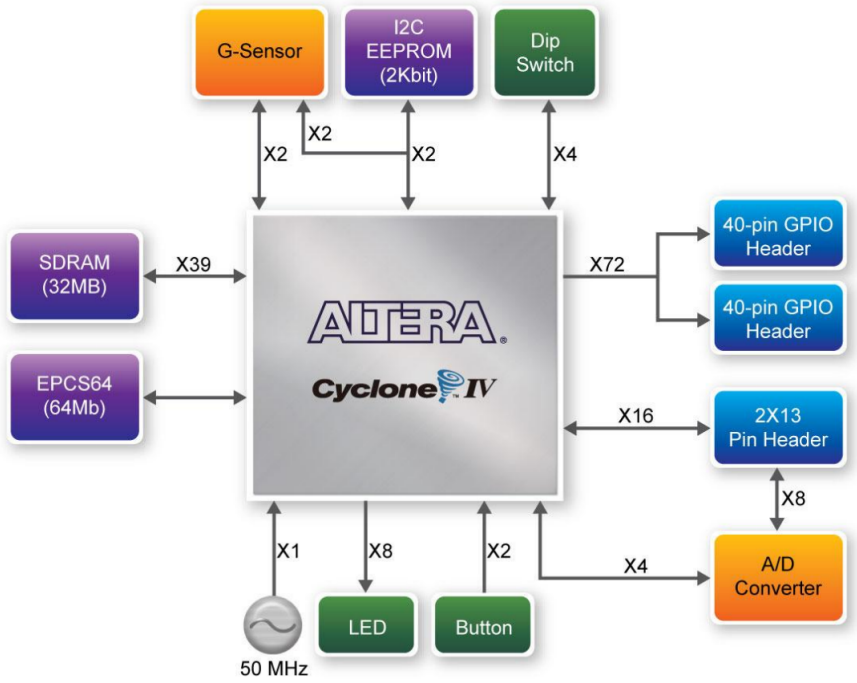


Figura 2.4: Organización de la placa de desarrollo DE0_NANO (Vista inferior)

dispone un acceso fácil a la placa de desarrollo DE0_NANO previamente mencionada. Si bien el diseño implementado requiere de algunas funcionalidades especificas del dispositivo elegido, implementar el diseño en otra placa de desarrollo o con otra FPGA es trivial siempre y cuando se disponga de la suficiente cantidad de elementos lógicos necesarios y si no hay problemas de consumo o de velocidad a considerar.

2.2. RISC-V

RISC-V es un estándar abierto de Arquitectura de Conjunto de Instrucciones o más conocido como *"Open ISA"* por sus siglas en inglés *"Open Instruction Set Architecture"*. La ISA es una interfaz abstracta entre el hardware y el nivel mas bajo de software que engloba la información necesaria para escribir un programa en código de maquina que se ejecutara correctamente en un procesador.

La naturaleza de carácter modular y el hecho de que la ISA de RISC-V este libre de regalías implica que cualquiera puede beneficiarse de la propiedad intelectual contribuida y producida por la comunidad de RISC-V. Mas aun la naturaleza no propietaria de RISC-V permite de la existencia de simuladores, compiladores y depuradores, entre otras herramientas, de fuente abierta.

La ISA de RISC-V y sus especificaciones asociadas son desarrolladas, ratificadas y mantenidas principalmente por el grupo RISC-V International, [24]. Dentro del conjunto de instrucciones, se definen dos subconjuntos que la especificación denomina bases y extensiones [25]. El subconjunto de las instrucciones denominadas *"bases"* es un conjunto de instrucciones reducido típico (tipo RISC) cuyo propósito principal es ser suficientes para que se pueda hacer uso de un compilador. A su vez también determinan la cantidad de registros de propósito general que se requieren, el tamaño de estos y tamaño del espacio de direccionamiento de memoria. Toda implementación de RISC-V requiere implementar al menos un conjunto de instrucciones de base. Las extensiones son conjuntos de instrucciones disjuntos a las bases cuyo objetivo es optimizar algún aspecto del diseño para el problema que se desea resolver, como por ejemplo acelerar o reducir el costo energético de ciertas operaciones. Salvo que el problema a resolver lo requiera, implementar extensiones es opcional. A su vez podemos categorizar a las extensiones en tres grupos por su nomenclatura: estándar, estándar extendido y no estándar. Los primeros dos grupos los regula RISC-V International y el ultimo corresponde a instrucciones que implementa el diseñador. Dado que es posible tener una gran variedad de arquitecturas que implementan distintos conjuntos de instrucciones de RISC-V, se define una nomenclatura para distinguir distintos diseños según la funcionalidad que implementa. Primero se especifica el ancho de bits de los registros de propósito general *"General Purposes Registers"* (GPR) junto con la base, luego le siguen las extensiones estándar, después le siguen las extensiones estándar extendidas y por ultimo las extensiones no estándar. Algunas extensiones también tienen el prefijo **Z** seguido de una letra asociada a la extensión estándar mas cercana a su funcionalidad, como por ejemplo la extensión *Zam* que extiende el conjunto de instrucciones definido en la extension **A**. El orden en que se especifica cada extension es el orden en el que aparecen en la tabla 2.1. En el caso de las extensiones estándar extendidas y no estándar, es decir las que tienen los prefijos **S, H, Z** y **X**, se requiere que los nombres estén separados usando un guión bajo y ordenados primero por categoría y luego en orden alfabético por categoría. A su vez si esta presente una extensión que tiene una dependencia, no se especifica el nombre de dicha dependencia. Por ejemplo, la nomenclatura RVI64AMSbSaZstoZicsr no es valida porque primero se especifica el ancho de los registros, el orden de las extensiones no es el correcto ni hay separación entre extensiones de prefijo **Z**, pero RV64IMAZicsr_Zsto_Sa_Sb si lo es.

Subconjunto	Nombre	Dependencias
Base		
Numeros enteros	I	
Numeros enteros (reducido)	E	
Estandar		
Multiplicacion de numeros enteros	M	
Instrucciones Atomicas	A	
Punto flotante (32bit)	F	Zicsr
Punto flotante (64bit)	D	F
General	G	MAFDZifencei
Punto flotante (128bit)	Q	D
Punto flotante (decimal)	L	
Instrucciones comprimidas (16bit)	C	
Manipulacion de bits	B	
Lenguajes dinamicos	J	
Memoria transaccional	T	
SIMD empaquetado	P	
Instrucciones vectoriales	V	
Interrupciones de modo usuario	N	
Acceso a Registros de Control y Estado	Zicsr	
Barrera de memoria (instrucciones)	Zifencei	
Alineamiento de instrucciones atomicas	Zam	A
Total Store Ordering (modelo de memoria)	Ztso	
Extendido y no estandar		
Extension de modo supervisor "ext_s"	Sext_s	
Extension de modo hipervisor "ext_h"	Hext_h	
Extension de Modo maquina "ext_m"	Zxmext_m	
Extension No estandar "ext_x"	Xext_x	

Tabla 2.1: Extensiones de RISC-V.

2.3. El lenguaje de un microprocesador

Dado que el código de maquina, en formato binario o hexadecimal, no es fácilmente legible por los programadores, nos abstraemos haciendo uso del lenguaje ensamblador. Dicho lenguaje contiene conjuntos de nemónicos que son mucho más fácil de recordar y manejar por el humano, aunque el uso de buenos entornos de desarrollo y compiladores se hace casi imprescindible para poder trabajar. Haciendo uso de un entorno ensamblador de instrucciones podemos traducir las sentencias de un programa escrito en este lenguaje a código maquina con un esfuerzo razonable. Es importante notar que dichas sentencias de lenguaje ensamblador están compuestas tanto por instrucciones reales como también instrucciones simbólicas que pueden ser una secuencia o casos particulares de instrucciones reales. Estas ultimas se denominan pseudoinstrucciones.

A continuación detallaremos las instrucciones implementadas. Todas las instrucciones de RISC-V son representadas como un numero binario de 32 bits y presentan un formato distinto dependiendo de su función. En la tabla 2.2 podemos ver como se organizan los bits de la instrucción según su tipo y a continuación detallaremos sobre el significado de cada campo.

Tipo R

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
funct7	rs2	rs1	funct3	rd	opcode

Tipo S

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

Tipo I

[31:20]	[19:15]	[14:12]	[11:7]	[6:0]
imm[11:0]	rs1	funct3	rd	opcode

Tipo J

[31]	[30:20]	[19:15]	[14:12]	[11:7]	[6:0]
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

Tipo U

[31:20]	[11:7]	[6:0]
imm[11:0]	rd	opcode

Tipo B

[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

Tabla 2.2: Formatos de instruccion de RISC-V.

El campo *OpCode* representa el código de operación y corresponde a los 7 bits menos significativos, es decir los bits que representan el menor valor. Su función es

especificar el tipo de operación que representa la instrucción. En nuestro caso los dos bits menos significativos siempre tienen el valor 11_2 en todas las instrucciones. La razón de esto radica en que valores distintos a 11_2 le corresponden a OpCodes de instrucciones de 16 bits, especificadas en la extensión **C** [25]. Los 5 bits que le siguen pueden variar y terminan de definir el tipo de operación. Esto se puede observar mas en detalle en la tabla 2.4. A su vez, como podemos observar en la tabla 2.3, las instrucciones que tengan presente los campos *funct* terminan de determinar la función que deberá cumplir la *Unidad aritmético-lógica* (ALU) o la *Unidad de saltos* (Branching Unit) en el caso de que la instrucción sea de salto. El campo *imm* representa un valor constante signado

	OpCode			
funct3	**BRANCH**	**LOAD**	**STORE**	**OP(-IMM)(-32)**
000	beq	lb	sb	add(i)(w) / sub(w)
001	bne	lh	sh	sll(i)(w)
010	—	lw	sw	slt(i)
011	—	ld	sd	sltu
100	blt	lbu	—	xor(i)
101	bge	lhu	—	srl(i)(w)/sra(i)(w)
110	bltu	lwu	—	or(i)
111	bgeu	—	—	and(i)

Tabla 2.3: Instrucción en función del campo *funct3* (Solo instrucciones implementadas).

(complemento a 2). En el caso de las instrucciones de carga que terminen en *u*, como *lbu, lhu*, etc; no se realiza una extensión de signo al mover el valor de memoria al registro. Por ultimo, los campos *rd, rs1* y *rs2* representan una dirección en el banco de *registros de propósito general*, que representan nuestro registro de destino en donde se guardara el resultado de la operación y nuestros operandos respectivamente. La forma en que la implementación trata cada campo se detalla en la siguiente sección.

Instr[6:2]	**OpCode**	**Tipo**	**Función**
00000	LOAD	I	Cargar desde memoria
01000	STORE	S	Guardar en memoria
11000	BRANCH	B	Saltos condicionales
11011	JAL	J	Saltos incondicionales
11001	JALR	I	Saltos incondicionales con registro
00100	OP	R	Aritmética/Lógica con registros
01100	OP-IMM	I	Aritmética/Lógica con registro e inmediato
00110	OP-32	R	Aritmética/Lógica con registros (32 bits)
01110	OP-IMM-32	I	Aritmética/Lógica con registro e inmediato (32 bits)
11100	SYSTEM	I	Operaciones privilegiadas
00101	AUIPC	U	Calculo de saltos relativos al PC
01101	LUI	U	Generación de inmediatos
00011	MISC-MEM (FENCE)	I	Ordenamiento de memoria (equivalente a NOP en nuestro caso)

Tabla 2.4: OpCodes de RISC-V implementados. Instr[1:0] := 11_2

2.4. Implementación de un microprocesador RISC-V de un solo ciclo

En esta sección detallaremos la implementación del procesador sobre la cual se trabajo [26]. El requerimiento inicial especifica que sólo es necesario implementar el conjunto de instrucciones de la base **I** (en nuestro caso de 64 bits) para hacer uso de un compilador y sus herramientas asociadas [25]. Por ello nos limitamos inicialmente a implementar este conjunto de instrucciones sumado a algunas funcionalidades adicionales que fueron útiles a la hora de depurar y que nos permitirán cargar un sistema operativo en un futuro con un poco más de desarrollo adicional. El diseño en cuestión es un procesador de un solo ciclo, es decir un procesador que procesa una instrucción por ciclo y todas las instrucciones demoran un ciclo de reloj. Este diseño fue elegido ya que permite en un futuro ser mejorado mediante una técnica llamada *"pipeline"* que consiste en separar las etapas de ejecución de una instrucción en varias partes más cortas y en ejecutar en paralelo varias instrucciones parcialmente solapadas como en una lína de montaje de una fábrica de productos en serie. Es importante notar que en un *"softcore"* que utiliza pipeline es mas complejo de implementar una forma de inyección de fallas [8], el cual es parte de nuestro objetivo final. Por lo tanto limitamos el alcance de nuestro trabajo al procesador de un ciclo, (*"pipeline"*) dejando para el siguiente trabajo la extensión del mismo a procesador con *"pipeline"*. Podemos observar en la Fig. 2.5 un diagrama simplificado de la implementación de nuestro procesador en el cual no esta presente la funcionalidad de una arquitectura con privilegio. Esto lo veremos detalladamente en la próxima sección. Es importante notar que si bien en la Fig. 2.5 se denotan distintas etapas, estas ocurren en paralelo y no secuencialmente por cada instrucción en cada ciclo de reloj. Esto se debe a que no guardamos el resultado de cada etapa en una memoria,

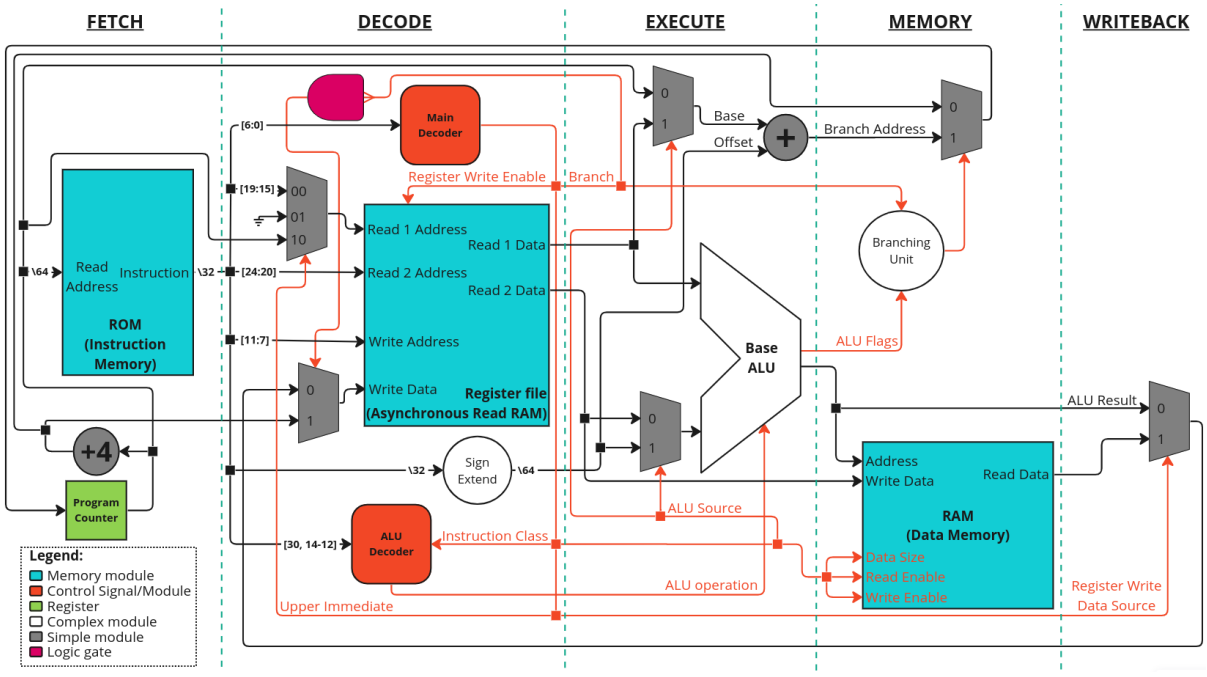


Figura 2.5: Diagrama de procesador de un solo ciclo.

como puede ser un (*"flip-flop"*). Sin embargo distinguiremos entre etapas ordenadas para simplificar la explicación de la operación del procesador. Podemos pensar que el flujo de datos es de la siguiente manera:

En la etapa **Fetch** se destacan un registro especial, el *contador del programa* (PC) y la *memoria de instrucciones* (normalmente una variante de ROM, pero puede contener también RAM). El PC da como salida una dirección (64 bits) en la memoria de instrucciones. El dato de salida de la memoria de programa es una palabra de 32 bits que representa una de las instrucciones de las cuales mencionamos en la sección anterior, correspondientes a nuestro programa. En nuestro diseño, cada dirección de memoria se corresponde con un byte, por lo tanto el PC debe incrementarse en múltiplos de 4 para leer siempre una instrucción valida. En el caso de saltos también nos moveremos en instrucciones que se alojan en direcciones que son múltiplos de 4.

En la etapa **Decode** se decodifica la instrucción, específicamente se establecen las señales de control, se leen los registros que serán los operandos y en caso de estar presente en la instrucción, se establecen el o los valores inmediatos asociados a dicha instrucción. La decodificación de registros de propósito general es directa dado que el *banco de registros* (Register File) es una memoria de acceso aleatorio de 31 palabras de 64 bits y 1 palabra que siempre toma el valor 0. Dichos registros que se interpretan como valores booleanos y de enteros signados o sin signo, tienen el prefijo **x**. Por otra parte la construcción de inmediatos se puede observar en la tabla 2.5 y en la tabla 2.7 podemos observar las señales de control según la instrucciones implementadas.

Tipo I

[64:11]	[10:5]	[4:1]	[0]
instr[31]	instr[30:25]	instr[24:21]	instr[20]

Tipo S

[64:11]	[10:5]	[4:1]	[0]
instr[31]	instr[30:25]	instr[11:8]	instr[7]

Tipo B

[64:12]	[11]	[10:5]	[4:1]	[0]
instr[31]	instr[7]	instr[30:25]	instr[11:8]	0

Tipo I

[64:31]	[30:12]	[11:0]
instr[31]	instr[30:12]	0

Tipo J

[64:20]	[19:12]	[11]	[10:5]	[4:1]	[0]
instr[31]	instr[19:12]	instr[20]	instr[30:25]	instr[24:21]	0

Tabla 2.5: Construccion de inmediatos.

Señal de control	
Descripcion	**Bit(s)**
ALU32	[16]
Upper Immediate	[15:14]
ALU Source	[13]
Register Write Data Source	[12]
Register Write Enable	[11]
Memory Read Unsigned Enable	[10] [9]
Memory Write Enable	[8]
Memory Data Size	[7:5]
Branch	[4:2]
Instruction Class	[1:0]

OpCode	**Señal de control**
BRANCH	000000000000_$f3\{3\}$_01
LOAD	000111_$f3\{1\}$_10_$f3\{3\}$_000_00
STORE	000100001_$f3\{3\}$_000_00
OP	0_00_001_000000000_10
OP-IMM	0_00_101_000000000_11
OP-32	1_00_001_000000000_10
OP-IMM-32	1_00_101_000000000_11
JAL	000_001_000000111_01
JALR	000_101_000000111_01
LUI	0_01_101_000000000_00
AUIPC	0_10_101_000000000_00

Tabla 2.6: Composicion de la señal de control y su valor según el OpCode.
Utilizamos '_' para delimitar cuando sea relevante y $f3\{n\}$ para indicar que el campo varia según el campo funct3 donde n representa la cantidad de bits variables.

Dado que los valores signados son complementados a 2, esto significa que el bit mas significativo de nuestra instrucción representa el signo del inmediato y si lo repetimos hasta llenar los 64 bits no cambiamos el valor que representa en base decimal.

Señal de control		
Descripcion	**Bit(s)**	**OP**
ALU32	[16]	0
Upper Immediate	[15:14]	00
ALU Source	[13]	0
Register Write Data Source	[12]	0
Register Write Enable	[11]	1
Memory Read Unsigned Enable	 [10] [9]	 0 0
Memory Write Enable	[8]	0
Memory Data Size	[7:5]	000
Branch	[4:2]	000
Instruction Class	[1:0]	10

Tabla 2.7: Composicion de la señal de control y su valor según el OpCode.
Utilizamos '_' para delimitar cuando sea relevante y $f3\{n\}$ para indicar que el campo varia según el campo funct3 donde n representa la cantidad de bits variables.

Por ultimo, podemos notar que en la Fig. 2.5 la señal *ALU32* correspondiente al selector de operaciones de 32 bits, no esta presente en la dado a que estructuralmente es muy similar en 64 bits. El comportamiento que tiene dicha señal lo podemos observar en la Fig. 2.6.

En la etapa **Execute** se computa el resultado de la instrucción y la dirección de salto. En la tabla 2.8 se observa detalladamente como opera la ALU en base a sus señales de control. En las instrucciones relevantes, solo consideramos *funct7[5]*, es decir el segundo bit mas significativo del campo *funct7* de la instrucción, porque dentro de las instrucciones implementadas es el único bit que varia. En la Fig. 2.7 podemos ver un diagrama de como fue implementada la unidad de saltos.

El flag *Zero* indica si el resultado de la operación de la ALU fue 0, el flag *Sign* indica si el resultado fue negativo y el flag *Overflow* indica si el resultado requiere mas bits de los que hay disponibles.

En la etapa **Memory** se decide si la próxima dirección de la memoria de instrucciones sera $PC + 4$ o la dirección de salto calculada en la etapa anterior. A su vez se lee o se escribe la memoria de datos (RAM) en caso de que lo indiquen las señales *Read/Write Enable* de la RAM. La señal *Data Size* indica la cantidad de bits que se van a leer o escribir, siendo 8, 16, 32 y 64 bits las opciones. Un detalle sobre la implementación sobre la FPGA es que utilizamos la memoria de bloque previamente mencionada que esta

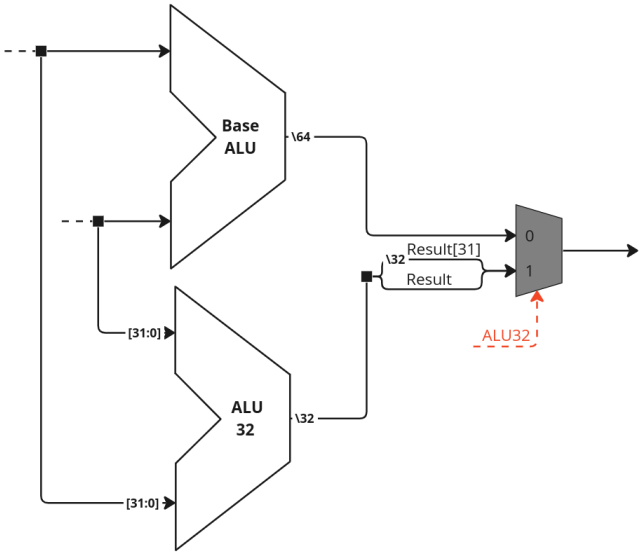


Figura 2.6: Diagrama del comportamiento de la señal *ALU32*

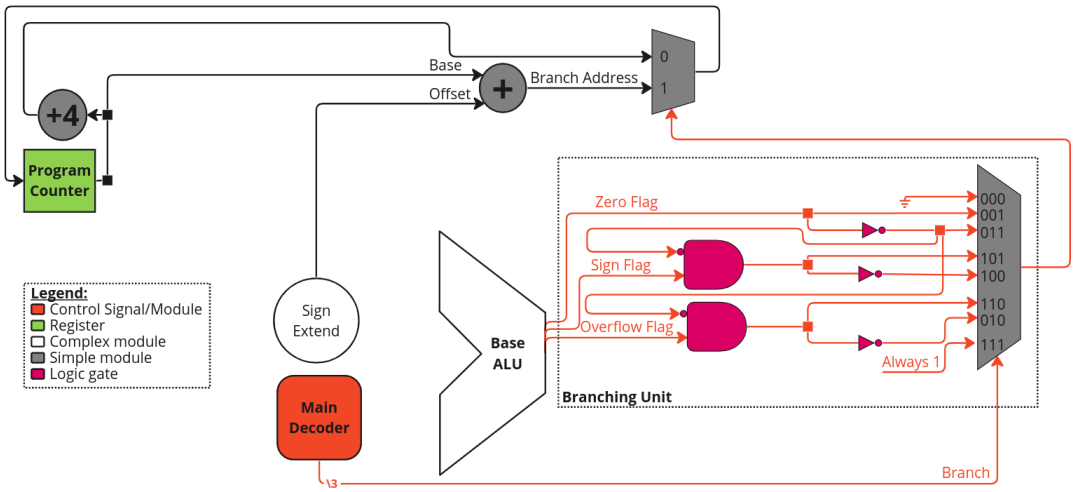


Figura 2.7: Diagrama de Branching Unit

presente en toda FPGA moderna. Esto trae consigo una desventaja con nuestro diseño debido a la forma en como se suelen implementar las memoria de bloque (BRAM). En efecto la BRAM requiere de que haya un registro para todas las entradas a nuestro modulo de memoria. La entradas a dichos registros están dadas por la señal de permiso de escritura, la dirección y el dato a escribir. Este tipo de memoria se conoce como *memoria síncrona* y aumenta la latencia de acceder a la memoria en 2 ciclos en vez de 1. Este problema se resuelve al considerar una arquitectura pipeline ya que se sitúan naturalmente registros entre cada etapa. En nuestro caso decidimos solucionarlo por software en el momento de compilación, repitiendo una vez las instrucción de lectura, en los casos de escritura no es necesario ya que la memoria implementa una técnica conocida como *data forwarding*, que consiste en leer el dato de escritura mas reciente de la dirección que se esta leyendo mientras esta siendo escrita. En caso de no poder hacer forwarding, la

I. Class	**funct3**	**funct7[5]**	**ALU Op.**	**OpCodes**	**Operación ALU**
00	—	—	0010	LOAD/STORE/-LUI/AUIPC	Suma
01	—	—	0110	BRANCH/JAL(R)	Resta
10/11	000	0	0010	OP(-IMM)(-32)	Suma
10	000	1	0110	OP(-IMM)(-32)	Resta
10/11	111	0	0000	OP(-IMM)(-32)	Conjunción lógica
10/11	110	0	0001	OP(-IMM)(-32)	Disyunción lógica
10/11	100	0	1001	OP(-IMM)(-32)	Disyunción lógica exclusiva
10/11	000	0	0111	OP(-IMM)(-32)	Desplazamiento lógico (izquierda)
10/11	101	0	0011	OP(-IMM)(-32)	Desplazamiento lógico (derecha)
10/11	101	1	1011	OP(-IMM)(-32)	Desplazamiento aritmético (derecha)
10/11	010	0	1110	OP(-IMM)(-32)	Comparación aritmética $<$
10/11	011	0	1010	OP(-IMM)(-32)	Comparación aritmética $<$ (Sin signo)

Tabla 2.8: Decodificación de la ALU Base.

solución seria agregar una instrucción NOP. Es importante destacar que estos problemas se solucionarán al implementar al procesador con pipeline. Por otra parte observamos experimentalmente al implementar la solución anterior que este tipo de memoria puede introducir lo que se conoce como un *riesgo de datos* ya que si el registro de destino es un operando también, puede suceder que al repetir la instrucción se lean valores que no son correctos. Cuando se presentan estas situaciones lo solucionamos guardando el valor del registro en riesgo en un registro que no utiliza el programa previo a las operaciones de lectura.

Por ultimo en la etapa **Writeback** se escribe en el registro de destino si lo indica la señal *Register Write Enable* el resultado de la ALU o de la RAM según indique la señal *Register Write Data Source*.

2.5. Registros de control y excepciones

Debido a que un procesador como el que describimos en la sección anterior no tiene forma de saber por si solo si sucedió en algún momento una falla, se decidió implementar una unidad de manejo de excepciones/interrupciones. La función de este componente es detectar si ocurre cierto comportamiento inusual y en cuyo caso saltar a una parte del programa donde se determinara que debe hacer el procesador dependiendo de la causa de dicho comportamiento. Podemos destacar tres tipos de comportamientos inusuales dependiendo de la severidad de cada caso:

- **Falla**: Comportamiento corregible que no impide que el programa siga corriendo.
- **Trampa**: Comportamiento que es notificado inmediatamente al ejecutar la instrucción responsable.
- **Aborto**: Error severo e irrecuperable.

A su vez, RISC-V hace la distinción entre excepciones e interrupciones dependiendo de como y cuando ocurre el comportamiento inusual. Tales eventos se los denomina excepciones en los casos donde el comportamiento inusual se debe a la instrucción que se esta procesado e interrupciones cuando la causa del comportamiento inusual se debe a factores externos o asíncronos. Debido a que no se implementaron interrupciones nos enfocaremos en excepciones. Las razones por las cuales un programa puede lanzar una excepción están detalladas en la tabla 2.9. Debemos notar que las excepciones implementadas son relevantes en los casos en que nuestro procesador se puede comportar de manera indefinida o son parte del comportamiento de las instrucciones *ecall* y *ebreak*, que forman parte del conjunto de instrucciones de base. En el caso de *Dirección desalineada (Instrucción)* omitimos hacer una implementación ya que los dos bits menos significativos siempre son ignorados al leer de la memoria de instrucciones con nuestro diseño, por lo que la dirección de la instrucción siempre esta alineada a 4 bytes, aunque podría no ser el caso si tuviéramos soporte para la extensión **C**. Tampoco tenemos presente en nuestro diseño sistemas de paginación, sistemas de protección de memoria o distintos modos de privilegio por lo que dichas excepciones no se implementaron aunque si detectamos alineamiento en las direcciones de memoria.

A su vez para toda esta funcionalidad se requiere del uso de registros adicionales denominados ***Registros de Control y Estado (CSR)***. Dichos registros operan de manera distinta a los GPR ya que pueden ser leídos/escritos implícitamente por la implementación y se deben acceder mediante instrucciones especiales. Para el manejo de excepciones debemos tener presente los siguientes registros:

- **mstatus**: Contiene el habilitador global de interrupciones y otras señales de manejo de excepciones, sumado a señales de estado que son irrelevantes en este momento.
- **mtvec**: Contiene la dirección a la cual salta el procesador cuando ocurre una excepción. Puede ser un valor fijo aunque en nuestro caso no lo es.
- **mepc**: Apunta a la instrucción donde ocurrió la excepción.
- **mcause**: Indica la causa de la excepción.

Codigo de excepciones	Descripción	Implementado?
0	Dirección desalineada (Instrucción)	X
1	Fallo de acceso (Instrucción)	X
2	Instrucción ilegal	✓
3	Punto de interrupción (Breakpoint)	✓
4	Dirección desalineada (Lectura)	✓
5	Fallo de acceso (Lectura)	X
6	Dirección desalineada (Escritura/operación atómica)	✓
7	Fallo de acceso (Escritura/operación atómica)	X
8	Llamada de entorno (Modo-U)	X
9	Llamada de entorno (Modo-S)	X
10	Reservado	—
11	Llamada de entorno (Modo-M)	✓
12	Fallo de pagina (Instrucción)	X
13	Fallo de pagina (Lectura)	X
14	Reservado	—
15	Fallo de pagina (Escritura/operación atómica)	X
16-23	Reservado	—
24-31	Designado para uso personalizado	X
32-47	Reservado	—
48-63	Designado para uso personalizado	X
64≥	Reservado	—

Tabla 2.9: Tabla de excepciones de RISC-V

Si bien la especificación [27] nos detalla otros CSR para el manejo de excepciones, para un sistema simple los que acabamos de nombrar son suficientes y son los que se implementaron. A su vez también hacemos la distinción de la forma en que ciertas regiones de los CSR se leen/escriben:

- **WARL**: Se permite escribir cualquier valor pero solo se leen valores permisibles. Nunca se lanza una excepción por escribir un valor invalido.
- **WPRI**: Intentar escribir al registro no afecta su valor y las lecturas por software deben ser ignoradas. En los casos de que no se utilicen estos campos del registro siempre devuelven el valor 0.
- **WLRL**: Se permite escribir solo ciertos valores y solo se leen valores permisibles. En caso de una escribir un valor invalido se permite lanzar una excepción o devolver un patrón dependiente de la ultima escritura realizada al registro.

En el caso de los registros anteriormente mencionados, *mepc*, *mtvec* y *mcause* son WARL. En el caso de *mstatus* solo consideramos que el bit [3] (MIE o habilitador global de excepciones de modo M) es WARL y el resto de bits son WPRI. Notemos también que las instrucciones de CSR, ebreak y ecall, entre otras instrucciones tienen el mismo OpCode: **SYSTEM**. Las instrucciones SYSTEM tienen el mismo formato que las instrucciones

de tipo I previamente mencionadas. Una vez mas, dependiendo del campo funct3 se determinara la función que debe cumplir la nueva ALU. En el caso de las instrucciones CSR (funct3 $\neq 000_2$) los 12 bits mas significativos representan la dirección del registro mientras que si funct3 $= 000_2$ dichos bits representan el campo funct12, es decir se utiliza para determinar alguna otra funcionalidad.

Para implementar estas instrucciones junto a algunas mas se agregaron nuevas señales de control y se agrego una ALU especial de dos entradas y dos salidas. Este cambio se ve reflejado en la tablas 2.10 y 2.11. Análogamente a la Fig. 2.6, la señal

I. Class	**funct3**	**ALU Op.**	**Operacion ALU**
00	—	1111	Reservado para Load Reserved / Store Conditional. Actualmente no intercambia valores ni los modifica
01	—	1111	Reservado para AMO. Actualmente no intercambia valores ni los modifica
10	001/101	0000	Leer CSR y escribir CSR con GPR.
10	010/110	0001	Leer CSR y fijar bits en CSR según mascara de GPR
10	011/111	0011	Leer CSR y limpiar bits de CSR según mascara en GPR
11	—	1111	Indefinido, no intercambia valores ni los modifica

Tabla 2.10: ALU de instrucciones CSR/Atomicas.

AMO/CSR ALU habilita el uso de la nueva ALU. Por otra parte, también debemos considerar como diseñamos el circuito para que se pueda leer/escribir implícita y explícitamente cada CSR. En principio esto varia según cada caso, pero a modo general lo podemos ver ilustrado en la Fig. 2.8.

Si bien bastaría con un solo multiplexor para diferenciar los dos tipos de escritura, se desea que tomen prioridad las escrituras implícitas en los casos que ocurran ambos tipos de escritura en simultaneo. Un ejemplo de este comportamiento es en el manejo de excepciones, si al procesar una instrucción *csrrx* ocurre un comportamiento inusual que lanza una excepción y a su vez el CSR de destino es uno de los registros

Señal de control	
Descripcion	**Bit(s)**
Retorno de trampa	[22]
AMO/CSR ALU	[21]
CSR Write Enable	[20]
Excepción	
Instrucción ilegal	[19]
ecall	[18]
ebreak	[17]

Instrucción	**Señal de control**
csrrx	011_000_00000100000000010
csrrxi	011_000_00010100000000010
mret	100_000_00000000000000000
fence	000_000_00000000000000000
ebreak	000_001_00000000000000000
ecall	000_010_00000000000000000
Ilegal	000_100_00000000000000000

Tabla 2.11: Nuevas señales de control para instrucciones privilegiadas.

de manejo de excepciones, el comportamiento deseado es que se escriban los registros relevantes implícitamente.

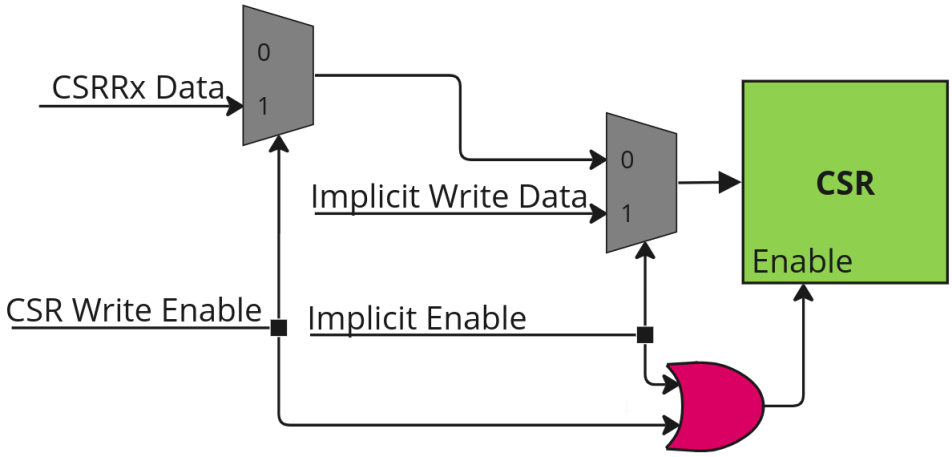


Figura 2.8: Diagrama de bloque de implementación general de CSR

Capítulo 3

Métodos de inyección de fallas

3.1. Motivación

A lo largo de las ultimas décadas se ha podido observar el incremento de la cantidad y la disminución en el tamaño de los transistores dentro de los circuitos integrados que utilizamos a diario. Estos avances tecnológicos trajeron consigo un aumento en el desempeño de dispositivos como por ejemplo microprocesadores. Sin embargo el menor tamaño de los transistores provoca que estos dispositivos sean más susceptibles a alteraciones aleatorias de información. Dichas alteraciones ocurren debido a partículas ionizantes descargando su energía en el circuito integrado, hecho que sucede comúnmente en el entorno espacial y que es conocido desde hace mucho tiempo [28]. Cuando ocurre que una sola partícula ionizante introduce un cambio en el estado de un circuito integrado se denomina al suceso Efecto de Evento Singular (SEE por sus siglas en ingles). Dentro de estos eventos podemos distinguir dos tipos de errores, suaves y duros. Los errores suaves son no destructivos y normalmente se manifiestan como pulsos transitorios en la lógica combinacional o como una inversión de bits en los registros/memoria. A ambos eventos se los conoce por sus siglas en ingles como SEU y SET respectivamente. Los errores duros son potencialmente destructivos, dentro de los cuales podemos distinguir por su denominación en inglés algunos como por ejemplo los Single Event Latch-up (enclavamiento), Single Event Gate Rupture (ruptura de compuertas lógicas) [29].

3.2. Esquemas de inyección de fallas

Es importante notar que cuando ocurre un SEE de manera natural se comporta en el circuito de la misma manera que si ocurriera de manera artificial. Por otra parte forzar que un sistema se comporte de manera incorrecta permite verificar la corrección y robustez de un sistema como el que estamos tratando con nuestro procesador en una etapa muy temprana de la resolución del problema que se desee tratar. A este proceso de introducir errores en un sistema se lo denomina inyección de fallas. Para circuitos digitales complejos se utilizan dos tipos de pruebas de inyección de fallas para evaluar que tan sensibles son frente a un SEE, las cuales se denominan pruebas estáticas y pruebas dinámicas. El objetivo de ambas pruebas es el mismo, obtener el promedio de partícu-

las que pueden provocar un funcionamiento incorrecto. Las pruebas estáticas, utilizadas típicamente cuando el sistema a probar es o contiene bloques de memoria, consisten en escribir un patrón en una región de memoria estudiada, inyectar las fallas, esperar un cierto tiempo, leer los resultados y finalmente comparar dichos resultados con lo que se escribió originalmente. Las pruebas dinámicas consisten en ejecutar una serie de comandos/instrucciones mientras el dispositivo bajo prueba se encuentra bajo los efectos de la radiación. Los distintos enfoques propuestos en la literatura se pueden clasificar en dos grandes categorías: métodos de hardware y métodos de software. A su vez, los métodos de software se pueden dividir en dos subclases, inyecciones por software (SWIFI) e inyecciones de fallas basadas en simulación [30–32]. En el caso de las simulaciones, las fallas se inyectan usando un Lenguaje de Descripción de Hardware (HDL), como por ejemplo VHDL o SystemVerilog, y programas de simulación como ModelSim o ActiveHDL. La principal ventaja de este tipo de enfoques respecto a otros tipos de métodos de inyección de fallas es la facilidad de observación y control que se pueden ejercer sobre el circuito bajo prueba. Sin embargo una gran desventaja de simular es que puede llevar mucho tiempo dependiendo del dispositivo bajo prueba y el hardware utilizado para simular. Por otra parte, mediante emulación usando de FPGAs se hace posible realizar mas pruebas sobre circuitos complejos en menos tiempo. Otra ventaja significativa de esta metodología es la capacidad que se tiene de inyectar fallas sin la necesidad de considerar la complejidad de los circuitos, tales como procesadores. La única limitación es que se debe proveer el código HDL correspondiente al circuito y tener una metodología que abarque emulaciones. Los métodos de inyección de fallas basados en FPGAs se han vuelto frecuentes, permitiendo estudiar los efectos de las fallas y la probabilidad de errores consecuentes a las fallas, incluso antes de su fabricación. A continuación detallaremos la metodología que se empleó para probar el procesador descrito anteriormente.

3.3. NoRTHFISch

Para probar la robustez del diseño del softcore implementamos la herramienta NoRTHFISch (*Non Real Time Hardware Fault Injection Scheme*). En la Fig. 3.1 podemos

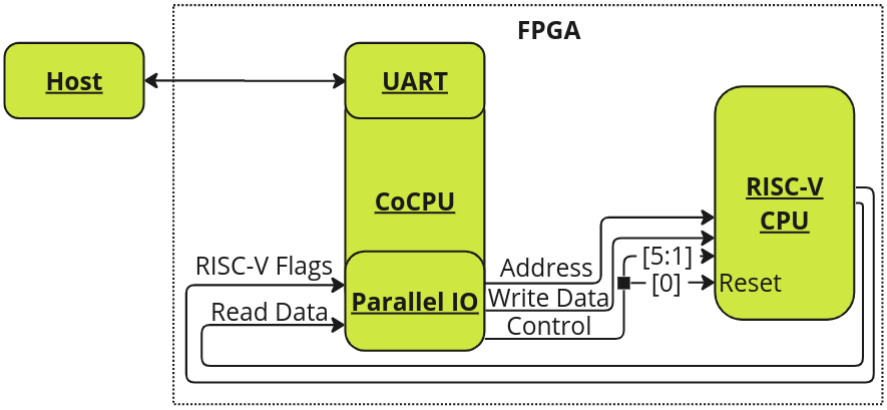


Figura 3.1: Diagrama de bloque del esquema de inyección de fallas implementado.

observar la arquitectura a nivel simplificado de dicha herramienta. Esta herramienta

consiste en el uso de un softcore adicional (**coprocesador**) que se comunica a través de un bus serial y registros compartidos con el softcore de RISC-V implementado (**procesador**). Mediante el protocolo UART el coprocesador se comunica con la computadora encargada de configurar la FPGA (**anfitrión**).

En nuestro caso el coprocesador que elegimos fue el softcore Nios-II de Intel. La razón por la cual este softcore fue elegido como coprocesador es por la facilidad de uso que nos proveen las herramientas de la plataforma sobre la que estamos trabajando, sin embargo no es difícil ver que esta elección es arbitraria ya que lo único que un coprocesador necesita es poder leer/escribir los datos de un bus y poder transmitir por otro medio los resultados de la computación hacia el anfitrión mediante un programa, por lo tanto no se requerirían mayores cambios a la arquitectura o en el proceso de programación e inyección de fallas ya que el coprocesador puede operar independientemente del procesador. A su vez definimos un bus partido en secciones de hasta 32 bits que utilizamos para cambiar y comunicar el estado del proceso de inyección de fallas, los detalles del mismo se observan en la tabla 3.1. Dado que con la arquitectura del coprocesador elegido solo podemos

Sección	**Ancho**	**Tipo**
Flags	2	Entrada
Control	6	Salida
Direccion	15	Salida
Datos	64	Bidireccional

Tabla 3.1: Bus de inyección usando Nios-II

acceder a secciones de hasta 32 bits, el bus de datos que ocupa 64 bits esta separado en 2. Debido a que nuestra inyección de fallas solo consiste en inversión de bits es posible codificar el bus de datos en 6 bits, sin embargo decidimos optar por un bus de 64 bits que actúa directamente como una mascara de bits para permitir invertir mas de un bit por escritura al bus de datos. A su vez dicho bus también es utilizado para leer/escribir directamente a la memoria del procesador y los registros compartidos.

La herramienta funciona en cuasi-tiempo real ya que cuando el coprocesador necesita utilizar el bus de control, el procesador tiene que frenar su procesamiento para que la inyección de fallas tenga el comportamiento deseado. En la tabla 3.2 podemos observar las señales de control que entran y salen de cada softcore. Decimos que una señal es de **Control** si sale del coprocesador y un **Flag** en caso contrario. Para evitar usar hardware adicional se opto por usar JTAG-UART [33,34], que hace uso de la conexión presente entre la FPGA y el dispositivo anfitrión para comunicarse usando el protocolo UART. Por otra parte al usar Nios-II como coprocesador podemos hacer uso de una *capa de abstracción de hardware* (HAL), un sistema en tiempo de ejecución que nos provee de los drivers necesarios para comunicarse con los bus de datos de entrada salida del coprocesador. Esto permite que nos abstraigamos de la implementación del hardware y se pueda escribir programas en C como si hubiera un sistema operativo presente. En particular facilita la comunicación entre el coprocesador y el anfitrión ya que no necesitamos implementar un driver de UART.

Para el hardware descrito y los dispositivos de la FPGA, JTAG-UART se com-

Bit	Nombre	Descripción
Control		
0	Processor Soft Reset	Reinicia el procesador
1	Register Write	Permite escribir un GPR del procesador
2	Memory Write	Permite escribir la memoria del procesador
3	Memory Read	Permite leer la memoria del procesador
4	CSR Enable	Permite leer registros de RISC-V y escribir compartidos
5	Fault Delay	Habilita el temporizador de inyección de fallas
Flags		
0	Delay Timeout	Notifica que el tiempo de demora transcurrió.
1	Breakpoint	Notifica al coprocesador que el procesador termino de ejecutar exitosamente

Tabla 3.2: Señales de Control y Flags de NoRTHFISch

porta de la misma manera que UART, solo que en vez de tener que describir el hardware necesario para la comunicación en HDL, la comunicación se hace a través de la interfaz de JTAG. Por el lado del anfitrión, tenemos dos componentes principales, el driver que habilita la interfaz con JTAG y software de alto nivel para que el usuario se comunique con la FPGA. Usar JTAG-UART nos abstrae de implementar hardware adicional y del costo que representa, sin embargo cabe destacar que su uso tiene sus desventajas. No existe un estándar para el protocolo de JTAG-UART y la implementación entre FPGAs de los distintos fabricantes que lo implementan varia lo suficiente como para que sean incompatibles entre si, lo cual implica que se requiere del uso de herramientas de fuente cerrada.

El proceso de inyección de fallas por parte del coprocesador luego de ser instanciado en la FPGA en cada corrida es el siguiente:

1. Por defecto el procesador inicializa con la señal Processor Soft Reset en alto. En simultaneo ponemos en alto las señales Processor Soft Reset para reiniciar el procesador, Fault Delay para habilitar que se inyecten de fallas después de un periodo de tiempo y CSR Enable para escribir los registros de control compartidos.

2. Asignamos un valor aleatorio a la demora usando el bus de datos. Dado que nuestros programas no son complejos ni correrán por un tiempo prolongado podemos limitarnos a usar solo los 32 bits menos significativos.

3. Despejamos el valor del bus de datos y dejamos en bajo las señales CSR Enable para finalizar la configuración del temporizador y Processor Soft Reset para arrancar el procesador.

4. Esperamos hasta que transcurra el tiempo de demora consultando mediante un bucle el valor de Delay Timeout. Cuando este flag este en alto significa que el procesador pauso su computación y es momento inyectar una falla.

5. Una vez transcurrido el tiempo de demora, fijamos la dirección del registro que queremos modificar, deshabilitamos la señal Delay Timeout y habilitamos la señal Register Write para poder modificar el registro deseado. Luego usando el bus de datos fijamos que bits deberán ser invertidos.

6. Deshabilitamos la señal Register Write para que reanude el programa en el procesador y esperamos leyendo en un bucle a que se ponga en alto la señal Breakpoint. En caso de que dicho flag no se halla habilitado en un cierto tiempo, consideramos que ocurrió una falla.

7. Por ultimo, leemos la memoria y los registros del procesador para análisis posterior.

A su vez, también fue necesario modificar el procesador. En la Fig. 3.2 se muestra la forma en que se organiza la inyección de fallas en el banco de registros, en donde se resaltan las nuevas señales que se debieron agregar al diseño.

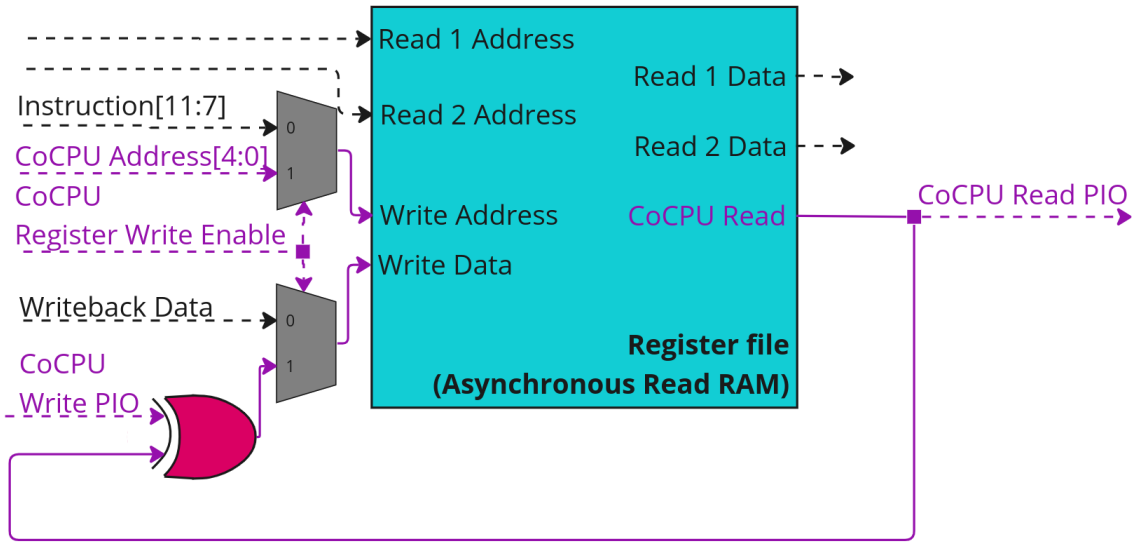


Figura 3.2: Diagrama de bloque del nuevo banco de registros.

Por otra parte en la Fig. 3.3 podemos observar el hardware necesario para que el esquema de inyección de fallas funcione sobre el procesador según lo que describimos anteriormente respecto al funcionamiento de las señales de control y flags. A modo de

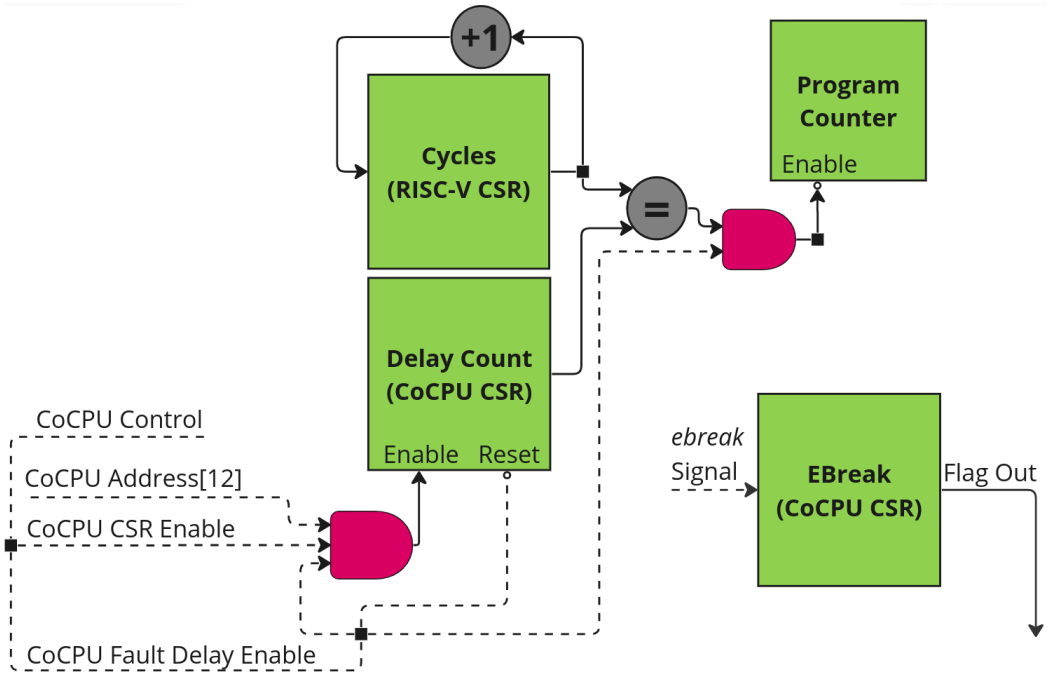


Figura 3.3: Diagrama de bloque de los registros compartidos.

ejemplo, podemos observar en el repositorio lo anteriormente explicado en los componentes *core.sv* y *decode.sv*.

3.4. Simulación

Como ya mencionamos previamente, una alternativa para la probar la robustez y verificar nuestro diseño es la inyección de fallas mediante simulación. Esta consiste en aplicar distintos valores a las señales de entrada y verificar que el comportamiento de nuestro diseño es el deseado al estudiar las señales internas y de salida. Dado que las ideas de inyección de fallas de NoRTHFISch no son especificas al coprocesador ni al modo en que se comunican los resultados, también utilizamos la simulación a la hora de desarrollar la implementación que se instancia en la FPGA. El único requisito para esto es que el procesador tenga el hardware y los puertos de entrada para que cualquier coprocesador pueda operar como mencionamos previamente. En nuestro caso hicimos uso del programa ModelSim, aunque esta elección es arbitraria. Primero para simular es necesario proveer una descripción del comportamiento de las señales de entrada a lo largo del tiempo, lo que se conoce como *testbench*. En dicha descripción declaramos las señales de entrada y salida de nuestro diseño bajo prueba (DUT) y lo instanciamos. Luego especificamos como cambian las señales de entrada a lo largo de la simulación hasta cierto punto. Por ultimo se realiza el análisis de los resultados y se comprueba si son adecuados al diseño. Adicionalmente, si bien podemos observar en ModelSim el valor de las señales a lo largo de la ejecución, también es posible guardar dichos resultados en un archivo. Esto nos resultó conveniente a la hora de automatizar el proceso de simulación corriendo desde la terminal el comando `quartus_map` para sintetizar el diseño y luego simular corriendo `vish -- -vsim -i -do "do wave.do"` (Ver [26]). Por ultimo en la Fig. 3.4 podemos observar un ejemplo de una corrida junto a la inyección donde las lineas amarillas denotan las distintas etapas de inyección de fallas siendo estas las siguientes:

- #**1** es el comienzo de la simulación, se caracteriza por la inicialización del temporizador, la configuración de que registro se inyectara y que bit se invertirá.
- #**2** representa el comienzo de la corrida del procesador
- #**3** es el punto donde el procesador se frena e inyecta la falla. El tiempo que se demora es configurable y depende del programa del coprocesador que se desea simular.
- #**4** marca el fin de la corrida del procesador y el inicio de la escritura de los resultados en un archivo.

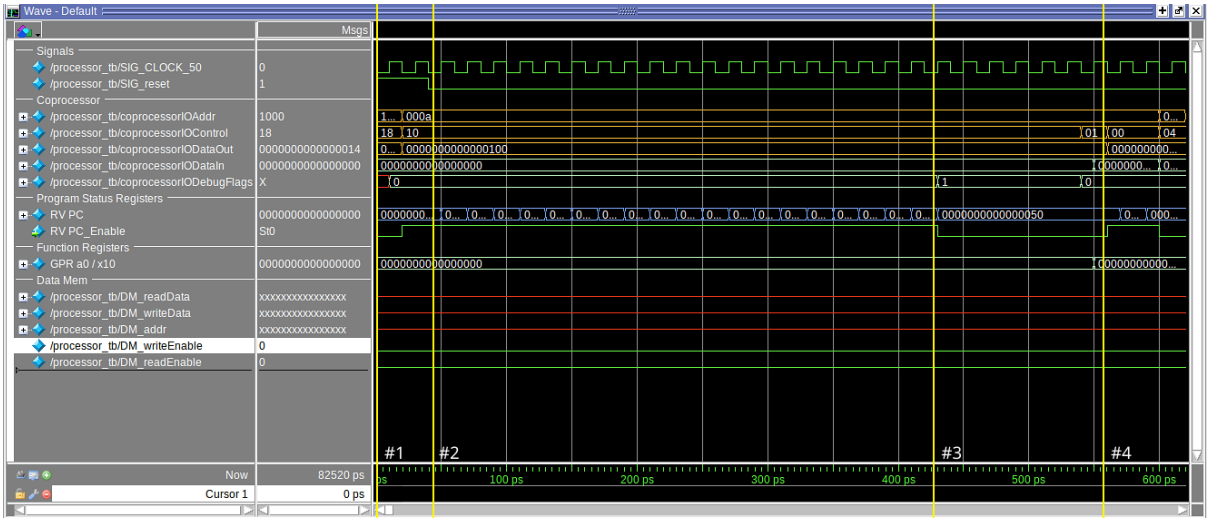


Figura 3.4: Ejemplo de corrida de simulación usando ModelSim. Explicación detallada en apéndice A

Capítulo 4

Casos de estudio

4.1. Preliminares de compilación

En esta sección detallaremos algunos casos de estudio que fueron utilizados para determinar la corrección y robustez del procesador. En todos los casos consideramos que ocurrió un error si el resultado difiere del esperado al concluir el programa, no concluye en un tiempo determinado u ocurre una excepción. Esto significa que computamos la tasa de error como la cantidad de veces que la región de memoria donde se aloja el resultado del programa difiere del resultado esperado. En las primeras subsecciones presentaremos cada caso de estudio junto con los resultados que obtuvimos luego de realizar las pruebas de inyección de fallas y en la ultima subsección haremos un análisis de los resultados.

Limitándonos a inyectar fallas en los registros de propósito general, la metodología para inyectar fallas fue la siguiente:

1. Una vez configurada la FPGA, como primer paso corremos el programa sin ninguna alteración, esperamos a que termine y observamos un patrón en la memoria o en un registro de propósito general del procesador RISC-V. También tomamos nota de cuanto demora.
2. Por cada GPR elegimos un tiempo de espera aleatorio para inyectar una falla y un bit aleatorio que deseamos alterar durante una cantidad fija de veces por cada registro.
3. Leemos la memoria/registros resultantes del procesador RISC-V una vez terminado el programa bajo un tiempo acotado.
4. Hacemos un análisis cuantitativo de los resultados comparando cada corrida en donde se inyecta una falla con la primera en la que el programa opera bajo la suposición de que no hay ninguna falla presente.

Para entender mejor el efecto que pueden producir las fallas, detallaremos un poco de como se traduce nuestro código escrito en C a lenguaje ensamblador de RISC-V. Todo programa escrito en C es una composición de bloques/subrutinas de lo que se denominan estructuras de control (secuencia, selección y bucle), es decir es un *lenguaje*

Tipo	**Tamaño (Bits)**
char	8
short	16
int	32
long / puntero	64
long long	64
float	32
double	64
long double	128

Tabla 4.1: Tamaño de tipos de datos en C con ABI=LP64

estructurado. Para que nuestro procesador pueda interpretar un programa, primero debe ser compilado. Muy resumidamente, este proceso consiste en interpretar cada archivo de código fuente (*unidades de traducción*) que compone nuestro programa, traducirlo a código maquina a través de código ensamblador produciendo *archivos objeto* y luego enlazarlos a un solo archivo que puede ser un programa ejecutable, biblioteca de funciones u otro archivo objeto. Dado que hay varias formas de traducir una sentencia de C en una secuencia de instrucciones de lenguaje ensamblador y en particular de manipular estructuras de dato abstractas, se define en la especificación de RISC-V la interfaz binaria de aplicaciones (ABI) para estandarizar esta interacción entre procesador y abstracción, ver [35]. En particular, algunas de las cosas que son definidas mediante convenciones por la ABI son como el procesador accede a las estructuras abstractas de datos, como se llaman a las subrutinas de nuestro programa, como el compilador debería tratar a los registros y las funciones que deberían cumplir. Esto ultimo lo podemos observar en la tabla 4.2. También se hace una distinción de cuando se preserva el valor de cada registro. Decimos que el registro se preserva si podemos garantizar que su valor antes y después del llamado de una subrutina es el mismo.

Si bien un compilador podría no respetar la ABI, esto implicaría que los archivos objeto resultantes por parte de este compilador podrían no ser compatibles con los archivos objeto resultantes de otro compilador. Por otra parte en los lenguajes ensambladores al no ser estructurados no tienen presente el concepto de bloque/subrutina. Dichos bloques/subrutinas se ven reflejados en la *pila de llamadas*. Una pila es una estructura abstracta que contiene una colección de elementos y se accede primero al ultimo elemento que se guardo. Contiene dos operaciones fundamentales, *apilar* (guardar un elemento) y *desapilar* (quitar un elemento). El uso de una pila es crucial para el llamado de subrutinas dentro de otras subrutinas ya que solo disponemos de un solo registro designado por la ABI para la dirección de retorno de una subrutina ($x1$ o ra por su denominación). La especificación de RISC-V no hace mención sobre instrucciones especiales para el manejo de la pila, si no que utiliza dos registros para indicar los limites de la pila respecto a cada función. El registro $x2$, denominado sp, es utilizado como puntero de pila y guarda la ultima dirección del marco de la pila durante el transcurso de la subrutina. El registro $x8$, denominado fp o $s0$ según el contexto, guarda la dirección del comienzo del marco de la pila. Al llamar una subrutina, el procesador apila lo que se denomina un *marco* que

Registro	Nombre ABI	Descripción	Preservación a traves de llamadas
x0	zero	Valor constante 0	—
x1	ra	Dirección de retorno de función	No
x2	sp	Puntero de pila	Si
x3	gp	Puntero global	—
x4	tp	Puntero de hilos	—
x5-7	t0-2	Registros temporales	No
x8	s0/fp	Registro guardado/Puntero de marco de pila	Si
x9	s1	Registro guardado	Si
x10-11	a0-1	Argumentos/valor de retorno de función	No
x12-17	a2-7	Argumentos de función	No
x18-27	s2-11	Registros guardados	Si
x28-31	t3-6	Registros temporales	No

Tabla 4.2: Registros de propósito general de RISC-V.

deberá contener el espacio disponible para guardar la dirección de retorno de la subrutina, el puntero del marco de donde se llama la subrutina que deberá ser restaurado una vez concluya su operación, los argumentos de la subrutina y las variables locales a dicha subrutina. En concreto, sucede lo siguiente: Previo al llamado de una subrutina cargamos en los registros denominados $a10 - 17$ los argumentos de la subrutina y saltamos guardando en el registro ra la dirección de retorno utilizando la pseudo instrucción *call* (dependiendo de la distancia relativa al PC se traduce a una instrucción *jalr* si el offset es menor a ± 2048 o una secuencia de la instrucciones *auipc;jalr* en caso contrario). Si la subrutina contiene mas de 8 argumentos se utiliza adicionalmente la pila para guardarlos. Después de saltar decrementamos el registro *sp* en un múltiplo de 16 que dependerá de la cantidad de variables presentes en nuestro contexto y su tamaño, luego cargamos en fp el valor previo a la resta de *sp*. A medida en que se requiera se cargan las variables/argumentos en los registros a partir de fp. Al concluir la subrutina se restaura ra al valor guardado y si debe retornar un valor se aloja en los registros $a0/a1$. Un detalle que cabe destacar, es que un compilador podría por predeterminado no guardar en la pila la dirección de retorno salvo en los casos que una subrutina llame a otra y ahorrar una instrucción. Esto se debe a que típicamente un compilador no va a generar instrucciones que modifiquen el registro ra. Dada nuestra metodología de inyección de fallas, la implicación de esto es

```
1  void* foo(void* arg_0,
2            ...,
3            void* arg_n) {
4  →   void* var_0;
5      ...
6      void* var_m;
7      ...
8  }
9
10 int main() {
11     ...
12     foo(arg_0,
13         ...,
14         arg_n);
15     ...
16     return 0;
17 }
```

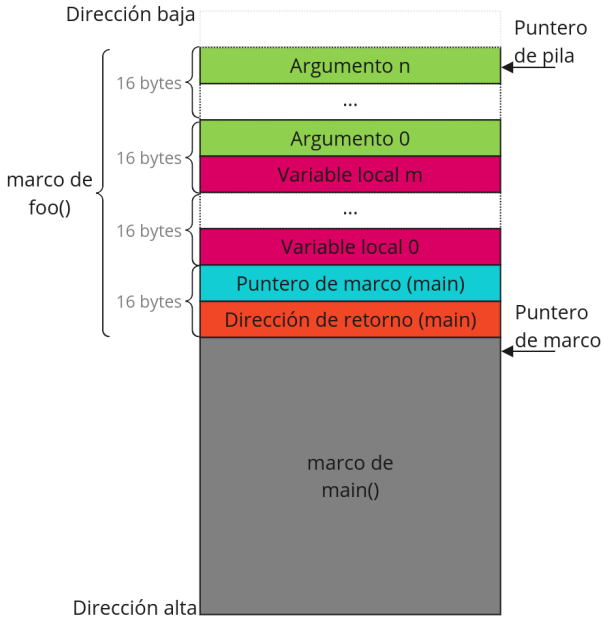


Figura 4.1: Ejemplo genérico de organización de la pila en memoria dentro de una subrutina

que podría suceder que al inyectar una falla en el registro *ra* nunca logremos retornar al punto anterior a la llamada de la subrutina. En la Fig. 4.1 podemos observar como se organizan generalmente los datos en la pila.

Los programas fueron escritos mayormente en C. Para compilar y luego configurar la FPGA se utilizaron las herramientas de GNU Compiler Toolchain versión 12.2.0 (referido como GCC de ahora en mas), configurado para la arquitectura RV64IA y la ABI LP64. En términos coloquiales esto significa que el compilador solo soporta instrucciones de las extensiones **I** y **A** (Aunque en nuestro diseño no esten presentes) de RISC-V (64 bits), determina el tamaño de los tipos de C (ver la tabla 4.1) y emula el sistema de punto flotante mediante las instrucciones soportadas. La primera función del programa, lo que se conoce como punto de entrada fue escrita exclusivamente en lenguaje ensamblador. Esta función se encarga de cargar en los registros *sp* y *gp* sus valores iniciales, siendo estos la dirección del fondo de la pila y la dirección de los datos de inicialización/variables globales. Luego habilita las excepciones y salta hacia la función principal (main) del programa que se encargara de computar el resultado de nuestro programa. Al finalizar su función, se retorna al punto de entrada y retornamos el valor 0 si la subrutina concluyo exitosamente. En el caso que haya ocurrido una excepción, guardamos en memoria los valores de los CSR *mcause* y *mepc* y retornamos el valor -1. En ambos casos utilizamos la instrucción *ebreak* para notificar al coprocesador que el programa concluyo su operación. Dado que también causa una excepción, en los casos en que se haya ejecutado dicha instrucción no cambiamos el valor de retorno ni guardamos.

Los archivos resultantes por parte del proceso de compilación pueden tener distintos formatos, pero nos centraremos en el formato ELF [36]. Como podemos observar en la Fig. 4.2, los archivos ejecutables ELF se organizan en distintas regiones:

- **Cabecera ELF**: La cabecera del archivo indica el formato del mismo y contiene un resumen de la organización del resto del archivo.
- **Tabla de Cabecera de Programa**: Describe los segmentos del programa.
- **Tabla de Cabecera de Secciones**: Describe las secciones del archivo.

A su vez podemos ver que se hace una distinción entre secciones y segmentos dependiendo si el contexto de la región de datos es de enlazado a tiempo de compilación o ejecución respectivamente. La razón de esta diferencia es debido a que en dichos contextos las regiones de datos se indexan de distinta manera, sin embargo la información es la misma en ambos casos. A su vez destacamos dos secciones al ser las únicas que son necesarias para inicializar las memorias de la FPGA:

- **.text**: Contiene las instrucciones del programa. Presente en el segmento de instrucciones/código ejecutable. Se escribe a la ROM.
- **.rodata**: Datos de solo lectura, contiene información de inicialización de variables. Presente en el segmento de datos en caso de existir. También puede presentarse como varias secciones con el prefijo .rodata o .srodata. Se escribe a la RAM.

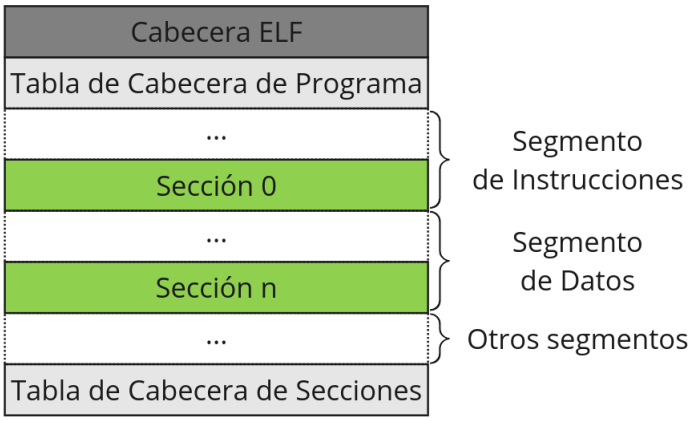


Figura 4.2: Organización de un archivo ELF.

Por ultimo para que sea mas fácil la tarea de análisis de los resultados decidimos no optimizar las funciones principales en ninguno de los casos pero si para funciones auxiliares.

4.2. Sumatoria hasta n

Este programa fue utilizado para verificar que la inyección de fallas en los registros de propósito general tenga el comportamiento adecuado y consiste en calcular $\sum_{n=1}^{N} n$. A su vez al ser un programa muy simple también decidimos que se calcule la sumatoria con un N grande para probar como se comporta el procesador prolongado mientras se le inyecta una falla en un programa que demora en calcular su resultado. Por otra parte es importante notar que al compilar no le indicamos al compilador que optimice el código. Sumado a la razón anterior, dependiendo del compilador y el nivel de optimización elegido se podría precomputar el resultado de la sumatoria cuando nuestro objetivo es que nuestro procesador la compute. La tasa de error resultante es de 0,1617 si consideramos solo los registros que utiliza el programa y de 0,0313 si consideramos todos los registros del procesador. En la Fig. 4.3 podemos observar un histograma de cuales registros fueron mas susceptibles a error.

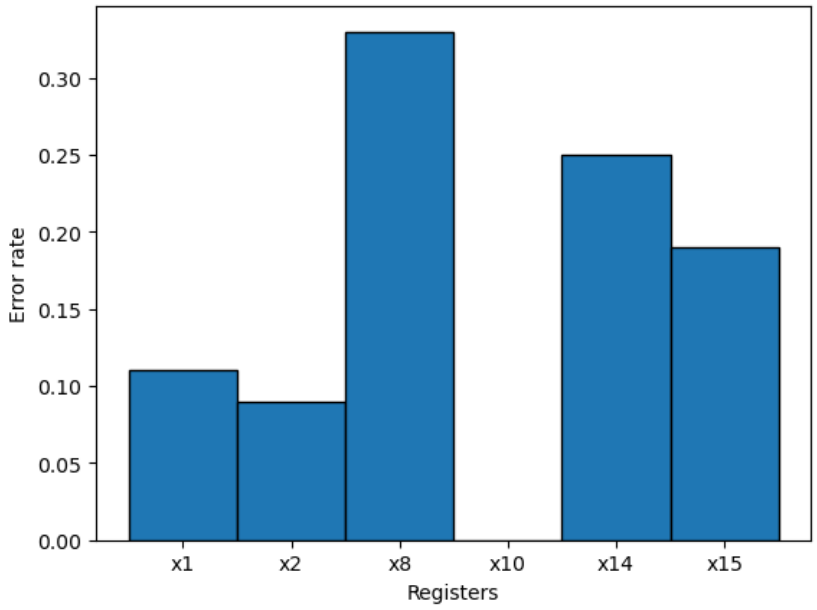


Figura 4.3: Tasa de error de los registros utilizados por el programa (Sumatoria).

4.3. Multiplicación de matrices

Dado que es una operación muy común en nuestro software de uso diario, decidimos también probar como se comporta el procesador al multiplicar dos matrices $N \times N$ de enteros. El programa en cuestión inicializa dos arreglos unidimensionales con los elementos de dos matrices. Recordemos que un arreglo unidimensional puede ser tratado como un arreglo bidimensional, es decir una matriz, si lo indexamos de la forma $i \times N + j$ donde i es el índice de filas, j el índice de columnas, $N \in \mathcal{N}_N = \{x \in \mathcal{N} \mid x < N\}$ y $i, j \in \mathcal{N}_N \cup \{0\}$. Una de las grandes ventajas de hacer esto es que nos asegura que los datos de cada matriz se van a encontrar siempre en memoria contigua. Si bien la inicialización es siempre con los mismos elementos para poder comparar con mayor facilidad los resultados a lo largo de la campaña de inyección de fallas, es importante notar que dichos elementos son arbitrarios. Un método alternativo de inicialización es usando un generador de números pseudoaleatorios con una semilla fija. Indistinto del método utilizado para lograr que el programa compile correctamente sobre la FPGA fue necesario inicializar la memoria de datos durante la programación de la FPGA. Esto es porque el compilador prefiere utilizar una región especifica de la memoria para guardar los datos de la inicialización en vez de aumentar la cantidad de instrucciones del programa en el primer método y necesita de una variable global en el segundo. Podemos ver como se

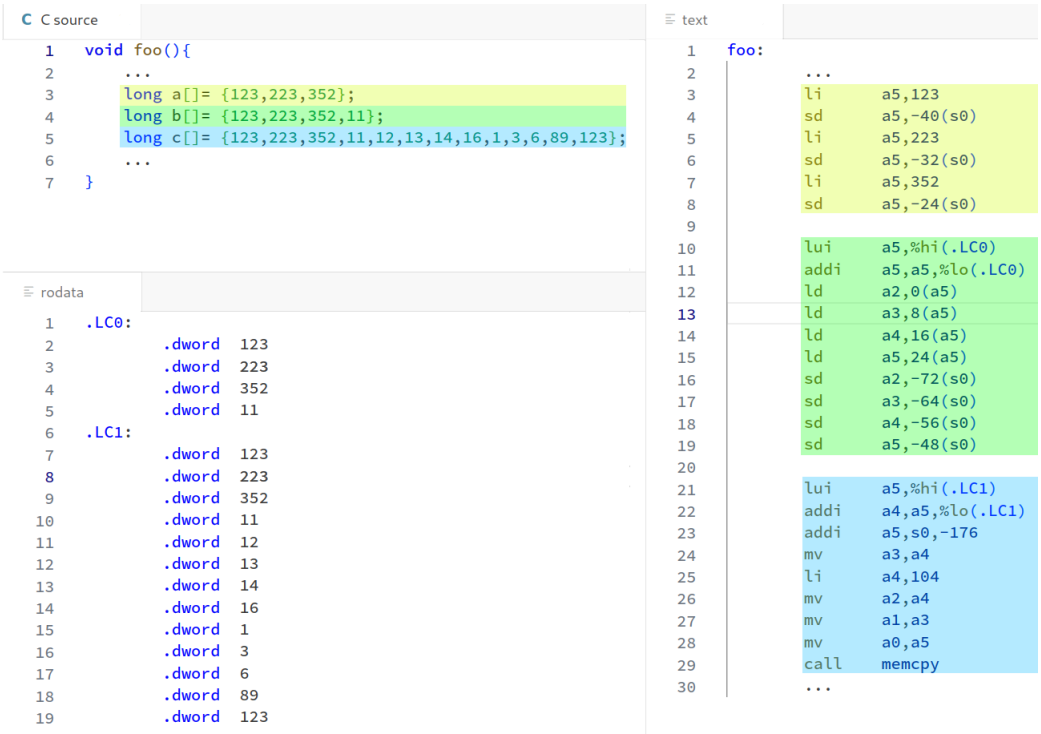


Figura 4.4: Compilación de arreglos inicializados usando GCC.

comporta el compilador para inicializar arreglos en la Fig.4.4 dependiendo de su tamaño, hasta tres elementos genera instrucciones *li* (Load Immediate), entre 4 y 12 los lee directamente de la región de memoria inicializada y a partir de los 13 hace uso de la función *memcpy*. Dicha función copia de una región de memoria a otra y toma tres parámetros, el primero es la dirección de memoria de destino, el segundo es la dirección de memoria

de donde se copia y el tercero es la cantidad de bytes que se copian. Por ultimo, como resultado de la función se retorna un puntero hacia el primer argumento. En la Fig. 4.5 se observa el histograma de error en los registros que utiliza el programa. La tasa de error resultante es de 0,294 si consideramos solo los registros que utiliza el programa y de 0,0948 si consideramos todos los registros del procesador.

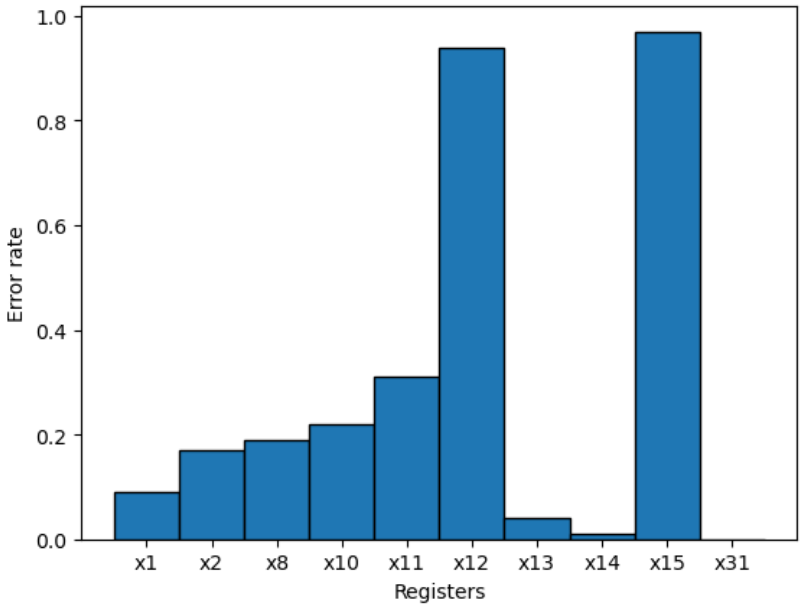


Figura 4.5: Tasa de error de los registros utilizados por el programa (Multiplicación de matrices).

4.4. Redes tolerantes a demoras

Actualmente la comunicación entre sistemas de computación modernos se realiza usando Internet que se basa en el modelo TCP/IP. Este modelo supone algunas cosas de la red sobre la que opera: existe un camino punto a punto entre el transmisor y el receptor, el tiempo máximo en que un mensaje se manda y se recibe (RTT) entre nodos en la red no es excesivo y la probabilidad de que el mensaje se pierda la comunicación punto a punto es baja. Desafortunadamente existen redes que no pueden operar bajo estas suposiciones y que no funcionan bien bajo este modelo. Para resolver esto en 2003 se propuso a partir de las ideas de Internet Interplanetario las redes tolerantes a demoras o DTN por sus siglas en ingles [12].

Una DTN es una red cuyos nodos garantizan que el mensaje se transmita de un nodo a otro aun en el caso de que hayan interrupciones. Podemos encontrar ejemplos de DTNs en escenarios donde las distancias son relativamente largas y con interrupciones también proporcionalmente larga entre los enlaces de sus nodos, como puede ser el caso de las comunicaciones interplanetarias e intersatelitales que se caracterizan por conectividad intermitente pero programada y largas distancias entre nodos. Para resolver el problema del contacto momentáneo entre nodos se han propuesto diversos algoritmos que explotan las características de la red de distintas formas [37–40] Sin embargo, es imposible para un nodo DTN diferenciar entre las altas demoras o disrupciones debidos a las largas distancias o a las oclusiones planetarias y la falla de un nodo vecino. De hecho, un nodo que no está respondiendo debido a una falla transitoria, produce el mismo efecto que un nodo volando en el lado opuesto de un planeta remoto. En la Fig. 4.6 podemos observar el comportamiento deseado de dichas redes, donde en un tiempo $t + t1$ un nodo móvil intercambia paquetes con un nodo emisor, en $t + t2$ intercambia paquetes con otro nodo móvil y en $t + t3$ intercambia paquetes con un nodo receptor.

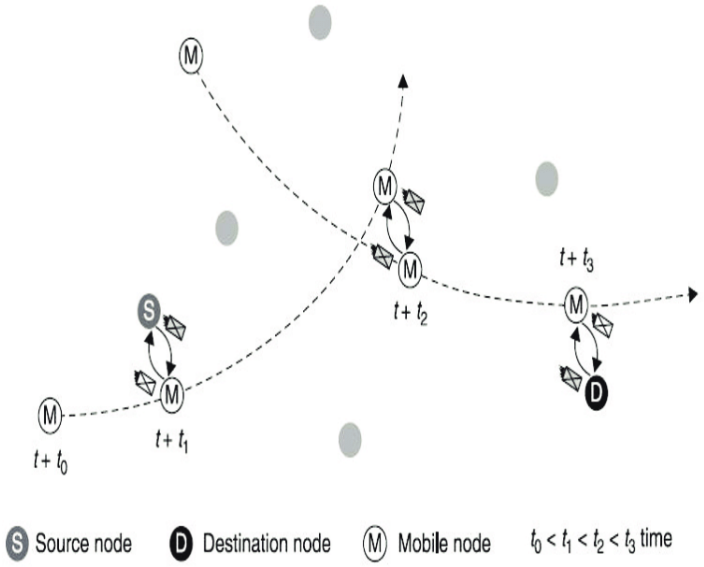


Figura 4.6: Comportamiento de una DTN. Fuente: [1]

De acuerdo con esta visión, se puede afirmar que las principales características

y servicios de los protocolos DTN se pueden aplicar también a escenarios de muy cortas distancias y muy rápida respuesta donde los nodos componentes evidencian una alta tasa de fallas. Es por esto que recientemente se ha propuesto el estudio de las plataformas que permitan desarrollar redes DTN operando en distancias entre nodos de varios ordenes de magnitud menor: DTN System On Chip (SOC). Para estas redes se ha propuesto el nombre WDTNOC (Wireless Delay Tolerant Networks On Chips), para describir DTNs en una escala a nivel de chip, donde las demoras y los intervalos de tiempo son reducidos en concordancia a este nuevo dominio de aplicación. También se ha propuesto el nombre WDTNBC para representar un concepto similar de redes entre chips, (Wireless Delay Tolerant Networks Between Chips) [41]. En este contexto de DTN-SOC podemos ver que las soluciones DTN se pueden adaptar e implementar a escala de chip como un paradigma de conectividad confiable y de soporte de las denominadas redes en chips (NOC). El paradigma NOC se está convirtiendo rápidamente en la infraestructura de comunicación estándar para proporcionar comunicación escalable entre núcleos. Sin embargo, la investigación ha demostrado que las interconexiones metálicas causan una alta latencia y consumen un exceso de energía en las arquitecturas NOC. Las tecnologías emergentes, como las interconexiones inalámbricas en chip, pueden aliviar los problemas de potencia y ancho de banda de los NOC metálicos tradicionales. En los NOC inalámbricos, las interfaces de red deben implementar antenas miniaturizadas y equipos electrónicos compatibles, incluidos protocolos de control de acceso medio en una escala muy pequeña dentro de un diseño de chip único. Aunque los NOC inalámbricos permiten una flexibilidad sin precedentes y transmisiones de un solo salto a larga distancia, su complejidad supera significativamente la de los NOC cableados tradicionales. Como resultado, los NOC inalámbricos son potencialmente poco confiables y propensos a fallas transitorias o permanentes provocadas no solo por errores de diseño sino también por fenómenos externos no deseados, como el efecto de la radiación que afecta actualmente a la electrónica moderna. Es por esto que este caso de estudio se centra en el problema que mencionamos recién al ver como se comporta un algoritmo de enrutamiento sencillo bajo mientras ocurren SEUs aleatoriamente. Primero asumiremos ciertas cosas sobre la red sobre la que trabajaremos:

- Los nodos no son demasiado distantes entre si y su distancia no varia.
- La arquitectura de la red esta dividida por niveles similar a como muestra la Fig. 4.7 donde los nodos que están en un mismo nivel representan cierta cercanía. En nuestro caso concreto por restricciones de memoria y hardware de computo, optamos por estudiar una red con 4 niveles y 4 nodos por nivel.
- Siempre podemos enviar un mensaje a un vecino del mismo nivel.
- A pesar de las posibles disrupciones un nodo que esta caído volverá a estar disponible durante el enrutamiento para que se pueda incluir en una ruta.

Luego construimos un modelo simple de la red que usaremos con nuestro algoritmo. Cada nodo solo tiene en cuenta su estado (Disponible/No disponible) y una lista de costo de transmitir a sus vecinos del próximo nivel. Los costos de transmisión hacia el próximo nivel son valores generados pseudoaleatoriamente y consideramos que el costo de transmitir a un vecino del mismo nivel es un valor fijo. El algoritmo no tiene en cuenta a los vecinos del nivel anterior por lo cual no guardamos dicha información en la lista de costos. El

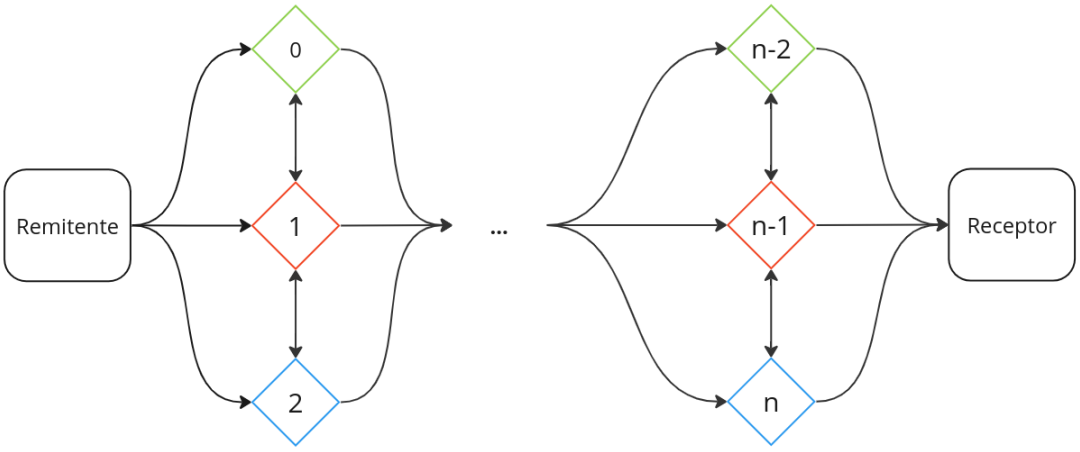


Figura 4.7: Diagrama genérico de la arquitectura de la DTN estudiada (3 nodos por nivel).

algoritmo de generación de números pseudoaleatorios (PRNG) elegido es de la familia **xorshift** [42], dado que es bastante simple y rápido en relación al hardware que tenemos disponible. El valor inicial (semilla) del PRNG no lo cambiamos entre distintas corridas del algoritmo para que los resultados sean comparables bajo el esquema de inyección de fallas actual. Por ultimo el enrutamiento se hace de forma *greedy*, es decir, siempre se busca desde el nodo actual el vecino disponible de menor costo. En caso de no tener ningún vecino disponible, esperamos saltando secuencialmente entre los nodos del mismo nivel hasta que haya algún nodo del próximo nivel que este disponible.

En este caso tenemos en cuenta dos resultados, el menor costo de envío y el camino resultante del emisor al receptor. En la Fig. 4.8 se observa el histograma de error en los registros que utiliza el programa. La tasa de error resultante es de 0,0591 si consideramos solo los registros que utiliza el programa y de 0,021 si consideramos todos los registros del procesador.

4.5. Análisis de los resultados

Al observar el código ensamblador generado por la compilación, podemos ver en detalle las razones por las cuales se produce un resultado erróneo en cada registro dados el PC donde ocurre la inyección y los valores del registro previo y posterior a la inyección. Si bien no entraremos en detalle sobre como fallaron nuestros programas a lo largo de cada ejecución en esta sección, podemos destacar a modo general de cada caso las situaciones donde puede fallar. En el apéndice B se detalla mas el análisis mediante un ejemplo. Como observamos en los histogramas anteriores, podemos ver que un registro que exhibe una tasa de error alta es *x8/fp*, es decir nuestro puntero de marco. Esto se debe a que constantemente estamos leyendo/escribiendo memoria utilizando direcciones relativas al valor guardado en este registro durante todas las subrutinas. En comparación el registro *x2/sp*, es decir el puntero de pila, no falla tan seguido ya que se guarda al principio de la llamada de una subrutina y se restaura al concluir, pero podría no ser el caso si tuviéramos mas llamadas a funciones durante las corridas de nuestros programas,

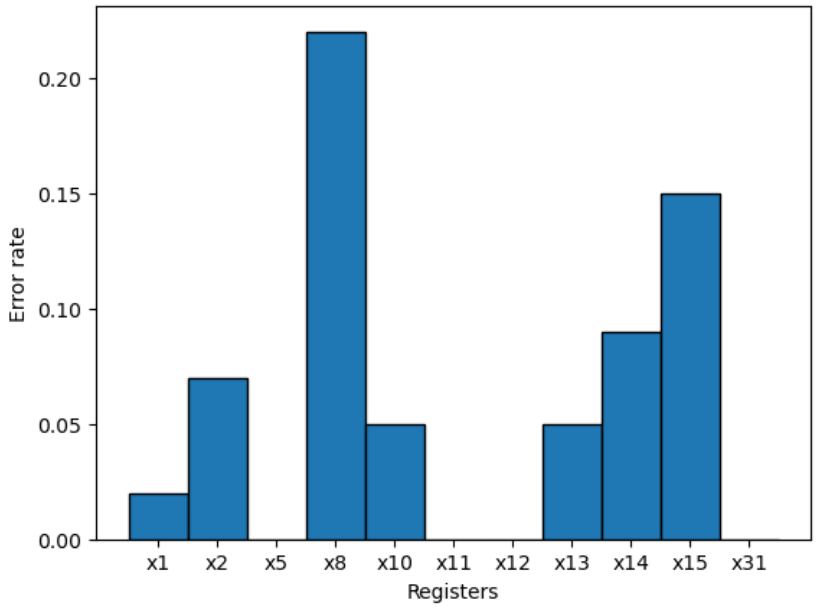


Figura 4.8: Tasa de error de los registros utilizados por el programa (DTN).

funciones recursivas o si no utilizáramos el puntero de marco. Por otra parte podemos observar que los registros *x10-15/a0-5* frecuentemente también presentan errores, dado que se utilizan principalmente como argumentos y operandos en las operaciones distintas al manejo de la pila. Tampoco hemos visto errores al inyectar en el registro x31/t6 aunque es posible que falle dado que lo utilizamos para evitar el riesgo de datos previamente mencionado. Por ultimo podemos ver que el registro *x1/ra* no presenta errores por lo general dado que la gran mayoría de los bits de direccionamiento de la memoria de datos se ignoran y en las subrutinas que no llaman a otras subrutinas dicho registro se guarda en la pila y se restaura al concluir. Esto podría no ser el caso, pero dado que no tenemos en cuenta las excepciones que pueden llegar a ocurrir por tratar de saltar a una dirección que esta fuera del rango o que desalinean escribiendo los 2 bits menos significativos la memoria de instrucciones, consideramos que dichas en situaciones no ocurre un error ya que y no se ven afectados el resultado final ni el tiempo de ejecución.

Capítulo 5

Conclusiones y trabajos futuros

Este trabajo propone el diseño de un microprocesador RISC-V básico sobre el cual uno puede modificarlo y expandirlo dependiendo del tipo de problema que se desee tratar. A su vez propone una herramienta para la verificación su funcionamiento correcto y su robustez frente a las AAI. El diseño básico del procesador y del inyector de fallas se validaron mediante algoritmos simples que pudieron ser analizados teniendo en cuenta su compilación. Una vez validados ambos se procedió a aplicarlo a la obtención de la tasa de fallas de un algoritmo de ruteo para una red DTN. De esta manera se demostró tanto la factiblidad de calificar nodos DTN basados en procesadores RISCV. Éste era el objetivo principal del trabajo y fué cumplido muy satisfactoriamente.

Sin embargo es importante destacar algunas tareas y mejoras a ser realiadas en futuros trabajos relacionados a éste:

Respecto al esquema de inyección de fallas se desearía que operara en tiempo real, que sea posible emular fallas en otras regiones del procesador y estudiar el comportamiento del procesador mientras se inyectan fallas en componentes que no sean el banco de registros de propósito general.

Respecto al procesador, una gran limitación del diseño actual es la carencia de memoria de acceso aleatorio, memoria de almacenamiento de datos permanente y la falta de herramientas de entrada/salida para interactuar directamente con el procesador. A su vez el diseño actual si bien tiene presente el manejo de algunos tipos de excepciones, hecho por el cual seria posible cambiar de un modo de privilegio a otro, no existe ningún tipo de mecanismo que distinga entre estos. Resolver estos problemas permitiría que el procesador pueda ser utilizado como cualquier otro mediante un sistema operativo, como por ejemplo Linux.

Mas aun, el diseño tampoco cuenta con ningún tipo de sistema de detección y/o corrección de errores. Resolver esto es crucial si se desea que sea utilizado para cualquier sistema critico. A su vez el conjunto de instrucciones implementado si bien permite que se pueda escribir cualquier programa Turing Completo al emular funcionalidad presente en otros procesadores, no es óptimo para ciertas operaciones en las que se requiere que el rendimiento sea mayor, como por ejemplo si el programador necesitara hacer uso de números flotantes. Por otra parte también se debería mencionar el deseo de explorar

otras arquitecturas y de mejorar la inyección de fallas. En particular una arquitectura con pipeline permitiría mejorar el rendimiento del procesador considerablemente sin mayores cambios y considerar toda la memoria disponible, no solo los registros de propósito general, permitiría probar la robustez del procesador de manera mas completa.

Para concluir, también deseamos destacar que una versión preliminar de este trabajo se presento este año en el Congreso Argentino de Sistemas Embebidos (CASE) [43]. También se encuentra presente una presentación del mismo en formato de video.

Bibliografía

[1] Nimish Ukey and Lalit Kulkarni. Implementation of energy efficient algorithm in delay tolerant networks. In *2017 2nd International Conference for Convergence in Technology (I2CT)*. IEEE, apr 2017.

[2] Eric Matthews and Lesley Shannon. TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, sep 2017.

[3] C. Duran, L. Rueda, G. Castillo, A. Agudelo, C. Rojas, L. Chaparro, H. Hurtado, J. Romero, W. Ramirez, H. Gomez, H. Hernandez, J. Amaya, and E. Roa. A 32-bit microcontroller featuring a RISC-V core. `https://github.com/onchipuis/mriscv`, 2016.

[4] M. Samsoniuk. Open source RISC-V DarkRISCV. `https://github.com/darklife/darkriscv`, 2019.

[5] Wietse F. Heida. Towards a fault tolerant RISC-V softcore. Master's thesis, 2016.

[6] Douglas Almeida Santos, Lucas Matana Luza, Cesar Albenes Zeferino, Luigi Dilillo, and Douglas Rossi Melo. A Low-Cost Fault-Tolerant RISC-V Processor for Space Systems. In *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE, apr 2020.

[7] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. On the Importance of Analysing Microarchitecture for Accurate Software Fault Models. In *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, aug 2018.

[8] M. Rebaudengo, M. Sonza Reorda, and M. Violante. Analysis of SEU effects in a pipelined processor. In *Proceedings of the Eighth IEEE International On-Line Testing Workshop (IOLTW 2002)*. IEEE.

[9] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. FISSC: A Fault Injection and Simulation Secure Collection. In *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, pp. 3–11, 2016.

[10] D. Gil, J. Gracia, J.C. Baraza, and P.J. Gil. Analysis of the influence of processor hidden registers on the accuracy of fault injection techniques. In *Proceedings. Ninth IEEE International High-Level Design Validation and Test Workshop (IEEE Cat. No.04EX940)*. IEEE, 2004.

[11] P.A. Ferreyra, G. Viganotti, C.A. Marques, R. Velazco, and R.T. Ferreyra. Failure and Coverage Factors Based Markoff Models: A New Approach for Improving the Dependability Estimation in Complex Fault Tolerant Systems Exposed to SEUs. *IEEE Trans. Nucl. Sci. Transactions on Nuclear Science*, 54(4):912–919, aug 2007.

[12] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 27-34, August 2003.

[13] Juan A. Fraire and Jorge M. Finochietto. Design challenges in contact plans for disruption-tolerant satellite networks. *IEEE Communications Magazine*, 53(5):163–169, May 2015.

[14] J. Fraire and J.M. Finochietto. Routing-aware fair contact plan design for predictable delay tolerant networks. *Ad Hoc Networks*, 25:303–313, feb 2015.

[15] Carlo Caini, Haitham Cruickshank, Stephen Farrell, and Mario Marchese. Delay- and Disruption-Tolerant Networking (DTN): An Alternative Solution for Future Satellite Networking Applications. *Proc. IEEE*, 99(11):1980–1997, nov 2011.

[16] Carlo Caini and Rosario Firrincieli. DTN for LEO Satellite Communications. In *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 186–198. Springer Berlin Heidelberg, 2011.

[17] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss. Delay-tolerant networking architecture. `http://www.rfc-editor.org/rfc/rfc4838.txt`, April 2007.

[18] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, B. Durst, K. Scott, and H. Weiss. Delay-tolerant networking: an approach to interplanetary Internet. *IEEE Communications Magazine*, 41(6):128–136, 2003.

[19] Juan A. Fraire and Pablo A. Ferreyra. Assessing DTN architecture reliability for distributed satellite constellations: Preliminary results from a case study. In *2014 IEEE Biennial Congress of Argentina (ARGENCON)*. IEEE, jun 2014.

[20] J. A. Fraire, P. Madoery, S. Burleigh, M. Feldmann, J. Finochietto, A. Charif, N. Zergainoh, and R. Velazco. Assessing Contact Graph Routing Performance and Reliability in Distributed Satellite Constellations. *Journal of Computer Networks and Communications*, 2017:1–18, 2017.

[21] Carlos Barrientos, Anabella Ferral, Leandro Cara, Juan Fraire, Raoul Velazco, P.G. Madoery, and Pablo Ferreyra. A Segmented Architecture Approach to Provide a Continuous, Long-Term, Adaptive and Cost-effective Glaciers Monitoring System Based on DTN Communications and Cubesat Platforms. 10 2017.

[22] Terasic Technologies. DE0_NANO User Manual. https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=593&FID=75023fa36c9bf8639384f942e65a46f3.

[23] Altera Corporation. Cyclone IV Device Handbook, Volume I. https://www.intel.com/content/www/us/en/docs/programmable/767845/current/cyclone-iv-featured-documentation-quick.html, 2016.

[24] RISC-V International. About RISC-V. https://riscv.org/about/, 2021.

[25] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213. *RISC-V Foundation*, December 2019.

[26] Adrian Edelstein. PassiFloRisc - V - A single cycle RV64IZicsr softcore. https://github.com/AGemstone/pfr-v, 2022.

[27] Andrew Waterman, Krste Asanović, and John Hauser. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203. *RISC-V International*, December 2021.

[28] D. Binder, E. C. Smith, and A. B. Holman. Satellite Anomalies from Galactic Cosmic Rays. *IEEE Transactions on Nuclear Science*, 22(6):2675–2680, 1975.

[29] Analysis Group Radiation Effects and & Radiation Physics Office (NASA). Single Event Effects. https://radhome.gsfc.nasa.gov/radhome/see.htm, 2021.

[30] Peter Folkesson, Sven Svensson, and Johan Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pp. 284–293. IEEE, 1998.

[31] Henrique Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. RIFLE: A general purpose pin-level fault injector. In *European Dependable Computing Conference*, pp. 197–216. Springer, 1994.

[32] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, J-C Fabre, J-C Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.

[33] Intel Corporation. *Embedded Peripherals IP User Guide*. March 28, 2022.

[34] Advanced Micro Devices. *Vitis Unified Software Platform Documentation, Embedded Software Development*. May 16, 2023.

[35] Kito Cheng and Jessica Clarke. RISC-V ABIs Specification, Document Version 1.0. https://github.com/riscv-non-isa/riscv-elf-psabi-doc, November 2022.

[36] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. Book I: Executable and Linking Format (ELF). https://refspecs.linuxfoundation.org/elf/elf.pdf, 1995.

[37] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. Spray and wait. In *Proceeding of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking - WDTN '05*. ACM Press, 2005.

[38] Fani Tsapeli and Vassilis Tsaoussidis. Routing for Opportunistic Networks Based on Probabilistic Erasure Coding. In *Wired/Wireless Internet Communication*, pp. 257–268. Springer Berlin Heidelberg, 2012.

[39] Anders Lindgren, Avri Doria, and Olov Schelén. Probabilistic routing in intermittently connected networks. *SIGMOBILE Mobile Computing and Communications Review*, 7(3):19–20, jul 2003.

[40] Michael Demmer and Kevin Fall. DTLSR. In *Proceedings of the 2007 workshop on Networked systems for developing regions*. ACM, aug 2007.

[41] Pablo Ferreyra, Juan Fraire, Fabian Gomez, R. Velazco, Daniel Sanchez, and Dardo Viscardi. Delay-Tolerant Wireless Networks on Chip: Preliminary Analysis and Results. pp. 1–6, 03 2019.

[42] George Marsaglia. Xorshift RNGs. *J. Stat. Soft.*, 8(14), 2003.

[43] Adrian M. Edelstein and Pablo A. Ferreyra. A New Fault Injection Scheme Optimized for RISC-V Soft Processors. In *Congreso Argentino de Sistemas Embebidos (CASE 2023)*, aug 2023.

Apéndice A

Explicación detallada de Fig. 3.4

Las señales que aparecen en las figuras a continuación representan los siguientes valores:

- **SIG**: Señales de control del procesador.
  - **CLOCK_50**: Nuestra señal de reloj.
  - **reset**: Señal de reinicialización.
- **coprocessorIO**: Señales del coprocesador.
  - **DataIn**: Dato de entrada/lectura hacia el coprocesador.
  - **DataOut**: Dato de salida / escritura desde el coprocesador.
  - **Addr**: Dirección de memoria del procesador, utilizado para memoria de datos y banco de registros.
  - **Control**: Señales de control, para el manejo de inyección de datos.
  - **DebugFlags**: Flags de RISC-V, para notificar al coprocesador.
- **RV PC (enable)**: El contador del programa (y su habilitador).
- **GPR a0/x10**: El registro de proposito general a0/x10.
- **DM**: Memoria de datos
  - **readData**: Dato de lectura.
  - **writeData**: Dato de escritura.
  - **addr**: Dirección de memoria.
  - **writeEnable**: Habilitador de escritura.
  - **readEnable**: Habilitador de lectura.

En la Fig. A.1 vemos la inicialización donde la señal *reset* esta en alto y se fijan los valores de las señales del coprocesador. En la etapa que le sigue no cambia nada en nuestras señales salvo la señal *RV PC* por lo que no se incluye una figura.

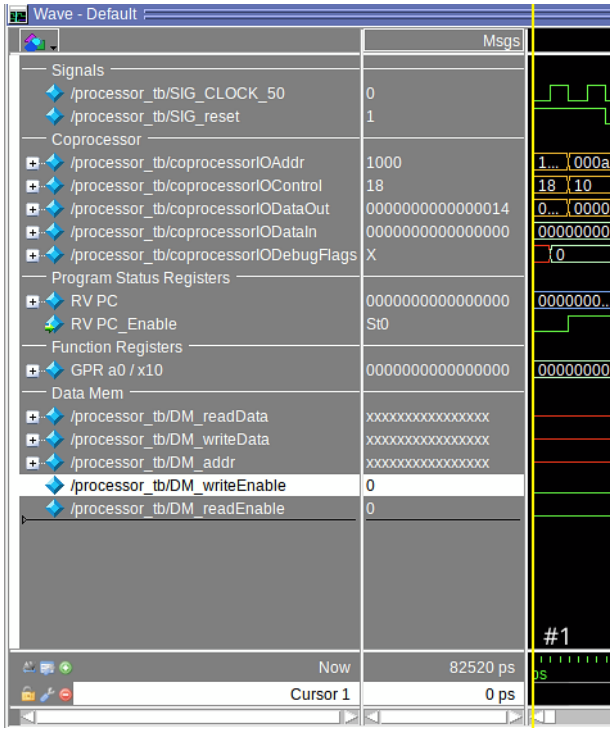


Figura A.1: Señales de la simulación, etapa 1.

En la Fig. A.2 es el momento en que el procesador frena su ejecución y el coprocesador efectúa la inyección de fallas detallada en 3.

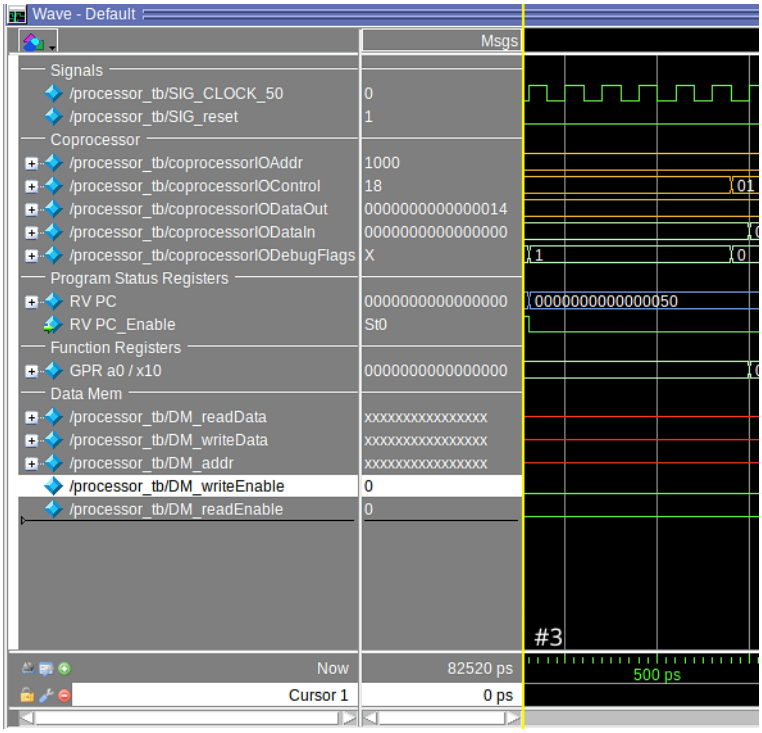


Figura A.2: Señales de la simulación, etapa 3.

Por ultimo en la Fig. A.3 observamos que el valor del *GPR a0* cambio, el fin de la simulación de la corrida del programa y el comienzo de la escritura de la memoria de datos del procesador a un archivo con el cambio de la señal de *Control* del coprocesador.

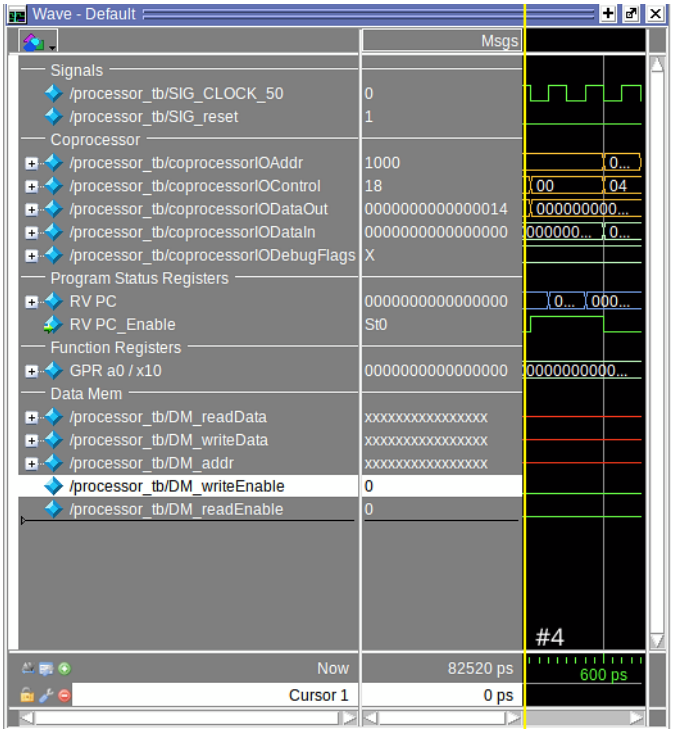


Figura A.3: Señales de la simulación, etapa 4.

Apéndice B

Análisis detallado

Para dar mayor claridad al análisis de los resultados generados por la inyección de fallas, consideremos a modo de ejemplo el caso de estudio *Multiplicación de Matrices*. Podemos ver que la siguiente función cuya funcion es realizar la multiplicacion matricial:

```
#define N 5
void matrix_mul(const long *a, const long *b, long *mul) {
  for (unsigned long i = 0; i < N; i++) {
    for (unsigned long j = 0; j < N; j++) {
      mul[i * N + j] = 0;
      for (unsigned long k = 0; k < N; k++) {
        mul[i * N + j] += a[i * N + k] * b[k * N + j];
      }
    }
  }
}
```

Se ensambla a el siguiente código ensamblador de RISC-V al compilar utilizando GCC sin ninguna optimización:

```
matrix_mul:
        addi    sp,sp,-80
        sd      s0,72(sp)
        addi    s0,sp,80
        sd      a0,-56(s0)
        sd      a1,-64(s0)
        sd      a2,-72(s0)
        sd      zero,-24(s0)
        j       .L2
.L7:
        sd      zero,-32(s0)
        j       .L3
.L6:
        ld      a4,-24(s0)
        mv      a5,a4
        slli    a5,a5,2
        add     a4,a5,a4
        ld      a5,-32(s0)
        add     a5,a4,a5
        slli    a5,a5,3
        ld      a4,-72(s0)
        add     a5,a4,a5
        sd      zero,0(a5)
        sd      zero,-40(s0)
        j       .L4
.L5:
        ld      a4,-24(s0)
        mv      a5,a4
        slli    a5,a5,2
        add     a4,a5,a4
        ld      a5,-32(s0)
        add     a5,a4,a5
        slli    a5,a5,3
        ld      a4,-72(s0)
        add     a5,a4,a5
        ld      a2,0(a5)
        ld      a4,-24(s0)
        mv      a5,a4
        slli    a5,a5,2
        add     a4,a5,a4
        ld      a5,-40(s0)
        add     a5,a4,a5
        slli    a5,a5,3
        ld      a4,-56(s0)
        add     a5,a4,a5
        ld      a3,0(a5)
        ld      a4,-40(s0)
        mv      a5,a4
        slli    a5,a5,2
        add     a4,a5,a4
        ld      a5,-32(s0)
        add     a5,a4,a5
        slli    a5,a5,3
        ld      a4,-64(s0)
        add     a5,a4,a5
        ld      a5,0(a5)
        mul     a3,a3,a5
        ld      a4,-24(s0)
        mv      a5,a4
        slli    a5,a5,2
        add     a4,a5,a4
        ld      a5,-32(s0)
        add     a5,a4,a5
        slli    a5,a5,3
        ld      a4,-72(s0)
        add     a5,a4,a5
        add     a4,a2,a3
        sd      a4,0(a5)
        ld      a5,-40(s0)
        addi    a5,a5,1
        sd      a5,-40(s0)
.L4:
        ld      a4,-40(s0)
        li      a5,4
        bleu    a4,a5,.L5
        ld      a5,-32(s0)
        addi    a5,a5,1
        sd      a5,-32(s0)
.L3:
        ld      a4,-32(s0)
        li      a5,4
        bleu    a4,a5,.L6
        ld      a5,-24(s0)
        addi    a5,a5,1
        sd      a5,-24(s0)
.L2:
        ld      a4,-24(s0)
        li      a5,4
        bleu    a4,a5,.L7
        nop
        nop
        ld      s0,72(sp)
        addi    sp,sp,80
        jr      ra
```

Por lo que nuestra pila la podemos representar de la forma en que se observa en la Fig. B.1

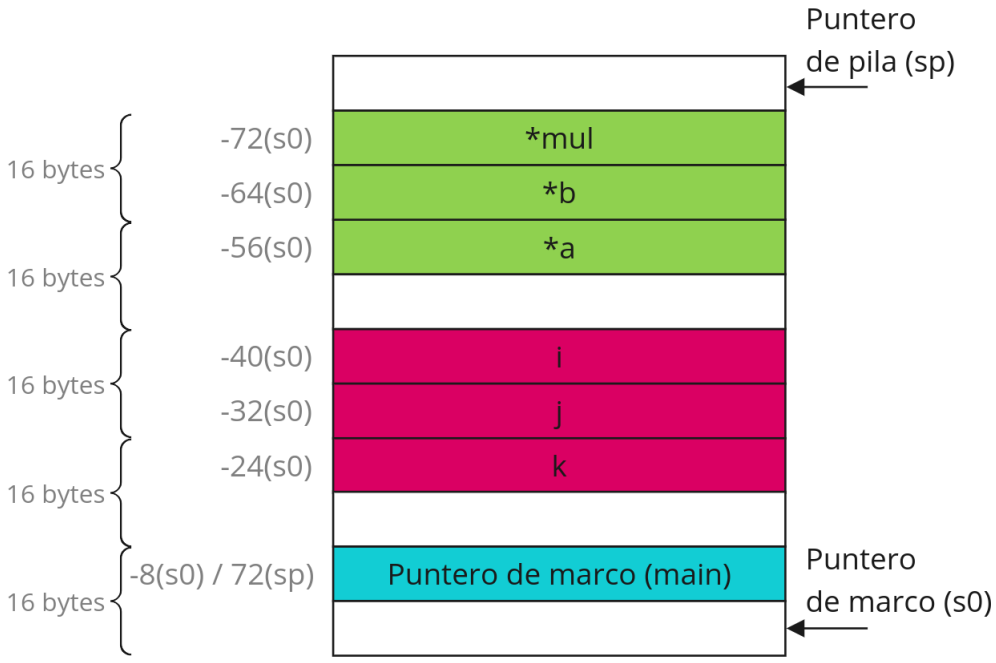


Figura B.1: Formato del marco de la función `matrix_mul`

A su vez destacamos lo siguiente:

- Como no se llama a ninguna subrutina dentro de esta función, no se guarda el registro *ra* en la pila.
- Para mantener el alineamiento de 16 bytes especificado por la ABI y el orden de los argumentos / variables debemos dejar espacios en la pila sin utilizar.
- Los únicos puntos donde podría fallar al inyectar en *sp* son en las primeras tres instrucciones instrucciones al comienzo de la función cuando se inserta el marco a la pila o en las ultimas tres cuando se restaura su valor y se retorna de la función.
- Los registros que mas se utilizan y son mas los susceptibles a las fallas son los registros a4 y a5 que están presenten en 38 y 56 respectivamente de las 87 instrucciones presentes en la función. Sin embargo hay que considerar que también estamos duplicando las instrucciones de lectura a memoria para evitar el riesgo de datos. Por lo tanto el numero de instrucciones nuevo es 123 y tenemos que los registros a4 y a5 están presentes en 51 y 70 instrucciones respectivamente.
- Podemos cambiar en varios puntos el valor de un registro y no va a fallar ya que su valor se restaura al leer de la pila.

Por otra parte en la Fig. 4.5 observamos que la mayoría de los errores que ocurrieron se debieron a fallas en los registro a2 y a5. Esto se debe a que la mayoría de las veces, la falla ocurría durante la inicialización de las matrices, es decir, durante la ejecución de la operación *memcpy* cuya descompilación se encuentra a continuación:

```
# Descompilación de memcpy, al optimizar evitamos
# movimientos inecesarios de memoria, pero aumenta
# la suceptibilidad a fallas
# (optimizamos la función, no el llamado)

# Computamos valores de los argumentos
lui         a5,0x3
mv          a4,a5
addi        a5,s0,-232
mv          a3,a4
li          a4,200
# Argumentos de memcpy
### Bytes a copiar
mv          a2,a4
### Direccion fuente
mv          a1,a3
### Direccion de destino
mv          a0,a5
call        memcpy
...
memcpy:
    li        a5,0
.L1:
    bne       a5,a2,.L0
    ret
.L0:
    add       a4,a1,a5
    lbu       a3,0(a4)
    lbu       a3,0(a4)
    add       a4,a0,a5
    addi      a5,a5,1
    sb        a3,0(a4)
    j         .L1
```

Como una ultima observación podemos ver que que al optimizar evitamos escribir / leer de la pila innecesariamente e interactuamos con los registros directamente lo cual reduce la cantidad de instrucciones y acelera nuestro programa, pero esto trae consigo el problema de que aumentan los puntos donde falla el programa ya que no hay posibilidad alguna de restaurar los valores en caso de que haya ocurrido una falla.

El análisis del resto del programa es análogo a lo anteriormente expuesto. Los datos que obtuvimos para generar los gráficos de la sección 4 se encuentran aquí (cada caso se encuentra en su directorio correspondiente), el script generador es generate_graph.py (ver codigo fuente).