Technical University of Crete

Diploma Thesis

Forward-Edge Control Flow Integrity ISA Extensions for RISC-V Architecture

Author:
Argyro Palli

Thesis Committee:
Assoc. Prof. Sotiris Ioannidis
Prof. Apostolos Dollas
Dr. Angelos Ioannou (FORTH and LBNL)

A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

October 16, 2023

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Forward-Edge Control Flow Integrity ISA Extensions for RISC-V Architecture

by Argyro Palli

Jump Oriented Programming attacks pose a security threat against modern processors. In order to enhance the security of designs based on the RISC-V architecture, a set of ISA extensions, namely the Forward-Edge Control Flow Integrity (CFI) ISA extension, have been proposed and are currently under evaluation by the relevant RISC-V committees, with the intention of becoming a standardized ISA extension in the RISC-V architecture. This work is also intended for publication. Within this context, this thesis presents the implementation and evaluation of the CFI ISA extension on the CVA6 soft processor core. The study leveraged the Genesys2 board as the hardware platform and the Verilator simulator for experimentation. The CFI extension was integrated into the CVA6 core, ensuring robust protection against potential security vulnerabilities stemming from JOP attacks. Notably, the performance overhead incurred by this security enhancement was limited to a mere 2%, demonstrating its minimal impact on system efficiency. Furthermore, the area utilization overhead on the Genesys2 board was effectively managed, with an minimal impact compared to the original design. There was a mere 1.04% increase in LUTs, a 2.14% increment in FFs, no discernible effect on estimated power consumption, and only a slight impact on the critical path, all without any timing violations. These adjustments were made while still adhering to the core's constraints, with a positive slack of 1.302 ns. This research underscores the feasibility of bolstering system security through ISA extensions, while preserving system performance and resource utilization at highly acceptable levels.

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Electrical and Computer Engineer

Forward-Edge Control Flow Integrity ISA Extensions for RISC-V Architecture

by Argyro Palli

Οι επιθέσεις αλματώδους προγραμματισμού (JOP) αποτελούν απειλή για τους μοντέρνους επεξεργαστές. Για να ενισχυθεί η ασφάλεια των σχεδιασμών που βασίζονται στην αρχιτεκτονική RISC-V, προτάθηκε ένα σύνολο επεκτάσεων εντολών αριχτεκτονικής (ISA), που ονομάζεται ¨Επεκτάσεις της αρχιτεκτονικής συνόλου εντολών RISC-V για υποστήριξη ελέγχου ακεραιότητας ροής έμμεσων διακλαδώσεων¨, η οποία αυτήν τη στιγμή βρίσκεται υπό αξιολόγηση από τις σχετικές επιτροπές του RISC-V, με τον σκοπό να γίνουν βασικός τύπος ISA στην αρχιτεκτονική RISC-V. Αυτή η εργασία προορίζεται επίσης για δημοσίευση. Εντός αυτού του πλαισίου, αυτή η διατριβή παρουσιάζει την υλοποίηση και αξιολόγηση των επεκτάσεων της αρχιτεκτονικής συνόλου εντολών RISC-V για υποστήριξη ελέγχου ακεραιότητας ροής εκτέλεσης (CFI) έμμεσων διακλαδώσεων στον πυρήνα επεξεργαστή CVA6. Η μελέτη χρησιμοποίησε την πλακέτα Genesys2 ως πλατφόρμα υλικού και τον προσομοιωτή Verilator για πειράματα. Η επέκταση της Ακεραιότητα Ροής Εκτέλεσης(CFI) ενσωματώθηκε στον πυρήνα CVA6, εξασφαλίζοντας αξιόπιστη προστασία από δυνητικά προβλήματα ασφάλειας που προκύπτουν από επιθέσεις προγραμματισμού με άλματα (JOP). Είναι σημαντικό να σημειωθεί ότι η επιβράδυνση που προκλήθηκε από αυτήν την ενίσχυση της ασφάλειας περιορίστηκε σε μόλις 2%, καταδεικνύοντας την ελάχιστη επίδρασή της στην απόδοση του συστήματος. Επιπλέον, η αύξηση χρήσης των πόρων στην πλακέτα Genesys2 διαχειρίστηκε αποτελεσματικά, με ελάχιστη αύξηση σε σύγκριση με τον αρχικό σχεδιασμό.Παρατηρήθηκε μόνο μια αύξηση 1.04% στους LUTs, αύξηση 2.14% στα FFs, καμία αντίκτυπος στην εκτιμώμενη κατανάλωση ενέργειας και ελάχιστη επίπτωση στον κρίσιμο μονοπάτι χωρίς καμία παραβίαση χρονισμού. Όλες αυτές οι προσαρμογές πραγματοποιήθηκαν διατηρώντας τους περιορισμούς του πυρήνα, με θετικό slack 1.302 ns. Αυτή η έρευνα υπογραμμίζει οτι είναι εφικτή η ενίσχυση της ασφάλειας του συστήματος μέσω επεκτάσεων εντολών αρχιτεκτονικής (ISA), διατηρώντας ταυτόχρονα την απόδοση του συστήματος και τη χρήση πόρων σε πολύ αποδεκτά επίπεδα.

Acknowledgements

I would like to express my heartfelt gratitude to my family, friends, and professors at Technical University of Crete for their unwavering support throughout my academic journey. I extend my deepest thanks to my supervisors, George Christou, Andreas Brokalakis, and Sotiris Ioannidis, for their guidance and help. Their expertise and unwavering support have been crucial in shaping my research and enhancing my skills. I am also grateful to RIVOS Inc. for their assistance and provision of tools that have greatly contributed to the completion of this thesis. Thank you all.

Contents

List of Figures

List of Tables

List of Abbreviations

ISA	**I**nstruction **S**et **A**rchitecture
RISC	**R**educed **I**nstruction **S**et **C**omputer
RVWMO	**W**eak **M**emory **O**rdering
RV32I	Base **I**nteger Instruction Set, **32-bit**
RV64I	Base **I**nteger Instruction Set, **64-bit**
M	Standard Extension for Integer **M**ultiplication and Division
A	Standard Extension for **A**tomic Instructions
F	Standard Extension for Single-Precision **F**loating-Point
D	Standard Extension for **D**ouble-Precision Floating-Point
Zicsr	Control and Status Register (CSR)
Zifencei	Instruction-Fetch Fence
C	Standard Extension for **C**ompressed Instructions
B	Standard Extension for **B**it Manipulation
K	Standard Extension for Scalar Cryptography
H	Standard Extension for **H**ypervisor
S	Standard Extension for **S**upervisor-level Instructions
CPU	**C**entral **P**rocessor **U**nit
ISA	**I**nstruction **S**et **A**rchitecture
OS	**O**perating **S**ystem
MMU	**M**emory **M**anagment **U**nit
PC	**P**rogram **C**ounter
FIFO	**F**irst **I**n **F**irst **O**ut
FU	**F**unctional **U**nit
OP	**OP**eration
RS1/2	**R**egister **S**ource address 1/2
RD	**R**egister **D**estination address
ALU	**A**rithmetic **L**ogic **U**nit
LSU	**L**oad **S**tore **U**nit
CSR	**C**ontrol and **S**tatus **R**egister
B	**B**uffer
FPGA	**F**ield **P**rogrammable **G**ate **A**rray

CNA	**C**omputer **N**etwork **A**ttack
CRA	**C**ode **R**euse **A**ttack
ROP	**R**eturn **O**riented **P**rogramming
JOP	**J**ump **O**riented **P**rogramming
COP	**C**all **O**riented **P**rogramming
DEP	**D**ata **E**xecution **P**revention
CFI	**C**ontrol **F**low **I**ntegrity
CFG	**C**ontrol **F**low **G**raph
WARL	**W**rite **A**ny Value **R**ead **A**ny Value
LPSLL	**L**anding **P**ad **S**et **L**ower **L**abel
LPSML	**L**anding **P**ad **S**et **M**iddle **L**abel
LPSUL	**L**anding **P**ad **S**et **U**pper **L**abel
LPCLL	**L**anding **P**ad **C**heck **L**ower **L**abel
LPSML	**L**anding **P**ad **C**heck **M**iddle **L**abel
LPSUL	**L**anding **P**ad **C**heck **U**pper **L**abel
LPLR	**L**anding **P**ad **L**abel **R**egister
ELP	**E**xpected **L**anding **P**ad
MPELP	**M**achine **P**revious **E**xpected **L**anding **P**ad
SPELP	**S**upervisor **P**revious **E**xpected **L**anding **P**ad
FCFIE	**F**orward - Edge **C**ontrol **F**low **I**ntegrity **E**nable
MFCFIE	**M**achine **F**orward - Edge **C**ontrol **F**low **I**ntegrity **E**nable
SFCFIE	**S**upervisor **F**orward - Edge **C**ontrol **F**low **I**ntegrity **E**nable
UFCFIE	**U**ser **F**orward - Edge **C**ontrol **F**low **I**ntegrity **E**nable
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh- **L**evel **S**ynthesis
RTL	**R**egister **T**ransfer **L**evel
IDE	**I**ntegrated **D**evelopment **E**nvironment
JTAG	**J**oint **T**est **A**ction **G**roup
ssp	**s**hadow **s**tack **p**ointer
CET	**C**ontrol-flow **E**nforcement **T**echnolog
IBT	**I**ndirect **B**ranch **T**racking
BTI	**B**ranch **T**target **I**nstruction
RELRO	**REL**ocation **R**ead-**O**nly
GOT	**G**lobal **O**ffset **T**able
PLT	**P**ocedure **L**inkage **T**able
DOP	**D**ata-**O**riented **P**rogramming

Dedicated to my family and friends...

Chapter 1

Introduction

1.1 Introduction

Code re-use attacks, such as Return-Oriented Programming (ROP) and Jump/ Call-Oriented Programming (JOP/COP)[1], are state-of-the-art exploits that allow attackers to execute arbitrary code on a vulnerable machine. These attacks do not require any code injections, instead, they re-use existing code fragments of a program to build the necessary functionality without violating Data Execution Prevention (DEP) [2]. According to a recent report, more than 80% of the vulnerabilities are exploited using code-reuse attacks[3].

Protecting against such attacks is necessary, in order to preserve program execution flow, but using software-only solutions is not sufficient, since advanced attacks can modify even the security software itself, thus bypassing any restrictions posed. In addition, the performance overheads of software-based solutions are non-negligible in many cases. This is particularly true in constrained environments (such as embedded devices) where there are intrinsic limitations in the amount of available compute and memory resources [4].

A way to defend against code re-use attacks is by employing Control Flow Integrity (CFI) techniques. These techniques focus on preventing code injection and any new functionality that is not part of the legitimate control-flow graph (CFG). They can be distinguished to Forward-Edge CFI and to Backward-Edge CFI, which are sets of rules that regard jumps and returns, respectively.

This work aims to enforce Forward-Edge Control Flow Integrity by extending the Instruction Set Architecture (ISA) of a popular RISC-V soft core, the CVA6 [5]. This core is a 6-stage, single issue, in-order CPU which implements the 64-bit RISC-V instruction set. New landing pad instructions are introduced, that enable software to indicate valid targets for indirect jumps in a program. The

Forward-Edge Control Flow Integrity supports labeling the indirect jumps and can encode up to 25-bit wide labels. It's important to note that this project doesn't encompass Backward-Edge Control Flow Integrity, as the specification for this ISA has not been finalized at the time.

The integration of the landing pad extension, signifying the introduction of a new set of instructions into a program's codebase, guarantees that in case of a control flow subversion, the hardware thread (hart) will promptly detect it and halt program execution, effectively fortifying the system against potential JOP attacks. Crucially, this security enhancement is achieved with minimal performance impact, as the average execution time increases by a mere 2.1%. Furthermore, when employed on the Genesys2 board, it introduces minimal area utilization overhead and no impact on worst case power consumption estimation.

1.2 Thesis Outline

- **Chapter 2 - Theoretical Background:** In this chapter, some basic information is provided in order to understand the theoretical basis of this work. A reference is made in RISC-V instruction set architecture, in code re-use attack and in control flow integrity technique.
- **Chapter 3 - Related Work:**This chapter explores related work.It includes a discussion of "Branch Regulation: Low-overhead Protection from Code Reuse Attacks","Hard-edges: Hardware-based Control-Flow Integrity for Embedded Devices","Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity" and "Branch Target Instructions (BTIs)".
- **Chapter 4 - Implementation:** This chapter provides a detailed account of the implementation of the Forward Edge Control Flow Integrity (CFI) ISA extension. It explains the necessary additions and modifications made to support this extension, as well as the specific changes implemented in the CVA6 core. This chapter also lists all the tools used to verify the functionality of the extension and to obtain measurements for the performance.
- **Chapter 5 - Results:** This chapter provides an overview of the findings related to performance, code size, area utilization, power consumption estimation and timing analysis.

- **Chapter 6 - Conclusions and Future Work:** This chapter serves as a conclusion, summarizing the research conducted and outlining future directions for this thesis.

Chapter 2

Theoretical Background

This work implements, investigates and evaluates a RISC-V ISA architecture in order to incorporate a security technique against a type of code reuse attack. The following sections will provide a brief presentation of the RISC-V architecture, the Jump - Oriented Programming (JOP) attack, which is the code re-use attack this work aims to mitigate, and the basic scheme that has been proposed to secure applications from this kind of attacks.

2.1 RISC-V

RISC-V[6] is an open-source instruction set architecture (ISA), first introduced in 2010 by researchers from the University of California, Berkeley. The design philosophy of RISC-V is based on the principles of simplicity, modularity, and scalability.

The RISC-V ISA is a reduced instruction set architecture (RISC) that provides a streamlined set of instructions and encoding formats to minimize complexity and improve performance. It uses a fixed-length instruction format, which simplifies the instruction fetch and decoding process, and provides uniformity across different implementations.

One of the key features of RISC-V is its modular design, which allows for easy customization and extension. The ISA is organized into a base instruction set and a number of optional extensions, which can be added or removed depending on the application requirements. This modularity also makes it easier for designers to optimize the ISA for different target markets, such as microcontrollers, embedded systems, or high-performance computing.

- **ISA Principles:**

As a RISC architecture, the RISC-V ISA is a load–store architecture, which means that it divides instructions into two categories: memory access and arithmetic/logical operations. The base instruction set has a fixed length of 32-bit naturally aligned instructions, however the ISA also supports compressed instructions.

Base	
Name	Description
RVWMO	Weak Memory Ordering
RV32I	Base Integer Instruction Set, 32-bit
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers
RV64I	Base Integer Instruction Set, 64-bit
RV64E	Base Integer Instruction Set(embedded), 64-bit
Extension	
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
Zicsr	Control and Status Register (CSR) Instructions
Zifencei	Instruction-Fetch Fence
Q	Standard Extension for Quad-Precision Floating-Point
C	Standard Extension for Compressed Instructions
B	Standard Extension for Bit Manipulation
Zk	Standard Extension for Scalar Cryptography
H	Standard Extension for Hypervisor
S	Standard Extension for Supervisor-level Instructions
Zihintpause	Pause Hint
Zfh	Half-Precision Floating-Point
Zfhmin	Minimal Half-Precision Floating-Point
Zfinx	Single-Precision Floating-Point in Integer Register
Zdinx	Double-Precision Floating-Point in Integer Register
Zhinx	Half-Precision Floating-Point in Integer Register
Zhinxmin	Minimal Half-Precision Floating-Point in Integer Register
Zmmul	Multiplication Subset of the M Extension
Ztso	Total Store Ordering

Table 2.1: ISA base and Extensions [6].

RISC-V has a modular design, consisting of different base parts, with optional extensions that can be added. The base architecture specifies instructions and their encoding, control flow, registers and their sizes, memory and addressing, logic manipulation, and more, which can implement a simplified general-purpose computer, with full software support,

32-bit RISC-V instruction formats

Format	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- *opcode* (7 bits): Partially specifies which of the 6 types of *instruction formats*.
- *funct7*, and *funct3* (10 bits): These two fields, further than the *opcode* field, specify the operation to be performed.
- *rs1*, *rs2*, or *rd* (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.

FIGURE 2.1: 32-bit Instruction Format

including a general-purpose compiler. The ratified base ISA and extension are presented in Table 2.1 and the 32-bit instruction format is listed in Figure 2.1.

Concerning the control transfers in RISC-V instruction set, the RV32I/RV64I provide two types of control transfer instructions - unconditional jumps and conditional branches. Conditional branches encode an offset in the immediate field of the instruction and are thus direct branches that are not susceptible to control flow subversion. Unconditional direct jumps using JAL transfer control to a target that is in a +/- 1 MiB range from the current PC. Unconditional indirect jumps using the JALR obtain their branch target by adding the sign extended 12-bit immediate encoded in the instruction to the rs1 register. The RV32I/RV64I does not have a dedicated instruction for calling a procedure or returning from a procedure. A JAL or JALR may be used to perform either a procedure call or a return from a procedure. The RISC-V application binary interface (ABI) however defines the convention that a JAL/JALR where rd (i.e. the link register) is x1 or x5 is a procedure call, and a JAL/JALR where rs1 is the conventional link register (i.e. x1 or x5) is a return from procedure. The architecture allows for using these hints and conventions to support return address prediction and the hints are specified in Table 2.1 of the unprivileged ISA specifications. The RISC-V CFI extension builds on these conventions and hints.

RISC-V has 32 (or 16 in the embedded variant) integer registers, and, when the floating-point extension is implemented, separate 32 floating-point registers are added to the system. All instructions address registers except for those of memory access. The first integer register is a zero register, and the remainder are general-purpose registers.

In terms of software support for RISC-V, there are available tools such as

the GNU Compiler Collection (GCC) toolchain (with the GDB debugger), an LLVM toolchain, the OVPsim simulator [7] (and library of RISC-V Fast Processor Models), the Spike simulator[8], and a simulator in QEMU (RV32GC /RV64GC) [9].

Operating system support for RISC-V is available (including the Linux kernel, FreeBSD, NetBSD and OpenBSD), however it is in early stages. Ports of the Debian and Fedora Linux distributions, and a port of Haiku, are stabilizing but they only support 64-bit version of RISC-V, with no plans to support the 32-bit version.

There are many examples of RISC-V IP cores available in the market such as the[10], a highly configurable, 32-bit RISC-V core that is designed for embedded and IoT applications, the OpenHW Group CORE-V-MCU[11], which is an open-source RISC-V core that is optimized for microcontroller applications, the Codasip Bk3[12] which is a high-performance, 64-bit RISC-V core that is designed for use in computing and networking applications.

Furthermore, RISC-V CPU and SoC implementations exist in the RISC-V ecosystem providing a wide range of applications that RISC-V can be used for. Some examples [13] of these implementations are SiFive Freedom U540, AndesCore N25F, OpenTitan, Berkeley Out-of-Order Machine (BOOM), Syntacore SCRx, Codasip H50X, Rocket Chip, LowRISC, Chipyard, Ariane SoC etc.

This work utilises CVA6 [14], previously known as Ariane, which is a 6-stage, single issue, in-order CPU which implements the 64-bit RISC-V instruction set. It fully implements I, M and C extensions. It, also, implements three privilege levels M (machine), S (supervisor), U (user) to fully support a Unix-like operating system.

2.2 Attacks

A security attack is an attempt to exploit vulnerabilities or weaknesses in a system or network in order to gain unauthorized access, steal data, disrupt services, or cause damage to the system or its users. There are many different types of security attacks and several ways that they can be categorized. As proposed in [15], security attacks can be classified according to their characteristics, severity and impact. As such the authors distinguish seven different categories:

1. Physical attacks: this category includes attacks that involve physical access, such as tampering, theft, or destruction.
2. Network attacks: this category includes attacks that target the communication networks, such as man-in-the-middle attacks, denial-of-service attacks and network scanning.
3. Malware attacks: this category includes attacks that involve the installation of malicious software, such as viruses, worms, and Trojan horses.
4. Device attacks: this category includes attacks that exploit vulnerabilities in the hardware or firmware, such as buffer overflow or code injection attacks.
5. Application attacks: this category includes attacks that target the software applications, such as SQL injection, cross-site scripting, and buffer overflow attacks.
6. Cloud attacks: this category includes attacks that target the cloud infrastructure, such as data breaches, data leakage, and unauthorized access.
7. Social engineering attacks: this category includes attacks that exploit human vulnerabilities, such as phishing, baiting, or pretexting.

2.2.1 Code Reuse Attacks - CRA

Code reuse attacks are a type of security attack that exploit vulnerabilities in software by reusing existing code segments to execute malicious code. The basic idea behind a code reuse attack is to take advantage of legitimate code segments in a program, such as function calls or libraries, and modify them to execute malicious code instead.According to the aforementioned attack categorization, code reuse attacks may be considered application attacks.

Code reuse attacks can be particularly effective because they bypass traditional security measures, such as address space layout randomization (ASLR) [16] and data execution prevention (DEP) [2], which are designed to prevent the execution of malicious code. Code reuse attacks can also be difficult to detect, as the attacker is not injecting new code into the program, but rather reuse existing code snipets.

To prevent code reuse attacks, developers can implement various security measures, such as control flow integrity (CFI), which verifies that only legitimate

code paths are executed, and stack canaries, which detect buffer overflow attacks. Additionally, developers can use code analysis tools to identify potential vulnerabilities and implement coding best practices to minimize the risk of code reuse attacks.

The two most common types of code reuse attacks are Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP) attacks. Both attacks work by chaining together small code segments to perform a series of operations that ultimately execute the attacker's code.

1. **Return-Oriented Programming - ROP**

   A return-oriented programming (ROP) [17] attack is a type of code reuse attack where an attacker uses existing code fragments in a program's memory, known as "gadgets," to execute malicious actions. These attacks stem from stack overflow attacks.

   The technique involves constructing a "ROP chain" by chaining together the return addresses of these gadgets to create a new sequence of instructions that perform the desired actions. The attacker may use this technique to bypass security defenses, such as non-executable memory or address space layout randomization (ASLR).

   ROP attacks have become increasingly prevalent in recent years, as they can be used to exploit a wide range of software vulnerabilities, including buffer overflows and format string vulnerabilities. Defending against ROP attacks requires a combination of techniques, including code signing, control-flow integrity, and runtime defenses such as stack canaries and shadow stacks.

2. **Jump-Oriented Programming - JOP**

   Jump-Oriented Programming (JOP) is a novel technique employed in code-reuse attacks, as described in [1]. Jump-Oriented Programming (JOP), is similar to Return-Oriented Programming (ROP). In an ROP attack, the software stack is scanned for gadgets that can be strung together to form a new program. ROP attacks look for sequences that end in a function return (RET). In contrast, JOP attacks target sequences that end in other forms of indirect (absolute) branches, like function pointers or case statements [18]. JOP leverages the fundamental building blocks of a computer program, namely, the instructions responsible for altering the

control flow, to construct malicious payloads that redirect program execution to arbitrary locations within the existing code base by corrupting function pointers residing in data segments.

Unlike traditional code injection attacks that rely on injecting new code into a target program, JOP operates by rearranging and repurposing existing code fragments, known as gadgets, to achieve malicious objectives. Gadgets are short sequences of instructions that end with a control flow transfer instruction. By combining these gadgets through careful manipulation of the stack and registers, JOP constructs a chain of instructions that deviates from the original program logic.

The essence of JOP lies in identifying and chaining together suitable gadgets that collectively perform the desired functionality. These gadgets may be sourced from various locations within the program, such as libraries or the program's own code. The attacker constructs a payload that populates the stack and registers with the necessary addresses of gadgets, thus forming a sequence of jumps that ultimately executes the attacker's malicious code.

JOP attacks are particularly challenging to detect and mitigate due to their reliance on legitimate code fragments. Traditional security mechanisms, such as address space layout randomization (ASLR) and data execution prevention (DEP), are less effective against JOP, as the attacker reuses existing code rather than injecting new code.

By leveraging JOP, attackers can bypass security measures, subvert control flow integrity, and execute arbitrary code within a compromised program. This technique has highlighted the need for new defense mechanisms and countermeasures to mitigate the threat posed by code-reuse attacks.

2.3 Control Flow Integrity - CFI

Control-Flow Integrity (CFI) [19] is a security mechanism designed to defend against code-reuse attacks, including Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP). CFI aims to ensure that the control flow of a program follows a predetermined legitimate path, preventing deviations caused by attackers attempting to hijack the control flow.To counter this, CFI employs two primary strategies: forward edge CFI and backward edge CFI.

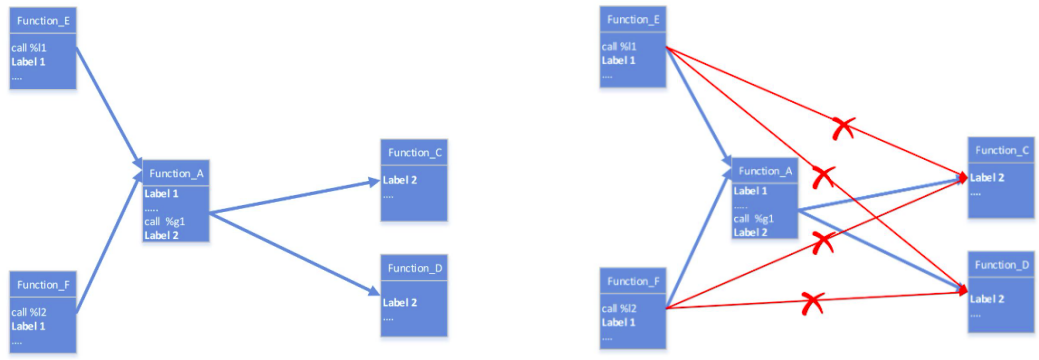


FIGURE 2.2: Forward Edge CFG and the restrictions that occur

Forward edge CFI focuses on protecting against attacks that exploit indirect control transfers, such as function pointers or virtual function calls. It enforces a strict policy that validates the targets of these indirect jumps or calls, allowing only authorized targets to be executed. By maintaining a comprehensive list of valid targets for each indirect control transfer site, forward edge CFI thwarts attempts by attackers to redirect control flow to unauthorized code.

In contrast, backward edge CFI aims to safeguard control flow transfers involving direct control flow instructions, such as function calls and returns. It focuses on verifying the integrity of return addresses stored on the stack. By maintaining a set of approved return addresses and validating them before executing a return instruction, backward edge CFI detects any alterations made by attackers. This prevents control flow hijacking by detecting unauthorized modifications to return addresses and halting execution in such cases.

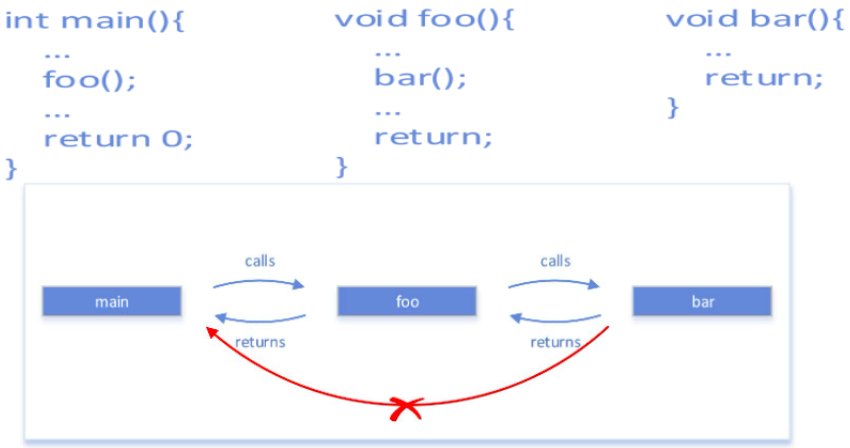


FIGURE 2.3: Backward Edge CFG example

When combined, forward and backward edge CFI techniques provide a comprehensive defense mechanism against control flow hijacking attacks. By ensuring that control flow adheres to expected paths and validating both indirect and direct control transfers, CFI significantly raises the bar for attackers attempting to exploit vulnerabilities.

CFI works by enforcing constraints on the program's control transfers, such as function calls, returns, and indirect jumps. It establishes a set of valid targets for each control transfer instruction and verifies that the actual target matches the expected set of targets. If an attacker attempts to redirect control flow to an invalid or unauthorized location, the CFI mechanism detects the deviation and raises an alert or terminates the program.

There are several approaches to implementing CFI [20], including:

1. Static CFI: Static CFI performs control-flow analysis at compile time, utilizing static program analysis techniques. It analyzes the program's source code or compiled binary to determine valid control flow targets and inserts runtime checks to ensure compliance with the predefined control flow graph.

2. Dynamic CFI: Dynamic CFI performs control-flow checks at runtime during the execution of a program. It uses runtime monitoring techniques to verify the legitimacy of control transfers by maintaining a runtime control flow graph. Dynamic CFI introduces runtime checks to validate control transfers and detect any unauthorized deviations.

3. Hybrid CFI: Hybrid CFI combines both static and dynamic analysis techniques. It performs static analysis to establish an initial control flow graph and inserts runtime checks based on this analysis. During program execution, dynamic analysis further refines the control flow graph and enables more precise monitoring.

CFI can significantly enhance software security by providing protection against control-flow hijacking attacks. However, implementing CFI may incur performance overhead due to the additional runtime checks and analysis. Various research efforts aim to optimize CFI techniques to minimize the impact on performance while maintaining robust security.

Chapter 3

Related Work

Chapter 3 provides valuable insights into related research that contextualizes Forward-Edge Control-Flow Integrity (CFI). It commences with an in-depth exploration of branch regulation methods, exemplified in [21]. These techniques leverage hardware-based approaches to dynamically control indirect branch instructions, effectively fortifying defenses against code reuse attacks while maintaining minimal performance overhead. Subsequently, the focus shifts to the pioneering work of Hard-Edges [4], which introduces a hardware-centric CFI enforcement strategy designed explicitly for embedded devices. A cornerstone of this research is the development of an ISA extension, empowering resource-constrained devices to efficiently thwart control flow hijacking attacks. Additionally, Intel's Control-flow Enforcement Technology (CET) [22] is highlighted for its introduction of instructions aimed at tracking indirect calls and jumps, offering a robust defense against code reuse attacks. Furthermore, ARM's contribution is noted, featuring landing pads in conjunction with Branch Target Instructions (BTIs) to enhance protections against Jump Oriented Programming (JOP) attacks. This comprehensive examination of related research optimally establishes the backdrop for the exploration of Forward-Edge CFI ISA extension in RISC-V architecture.

3.1 Branch regulation

As mentioned in the previous section, ROP and JOP attacks exploit vulnerabilities in a program by reusing existing code fragments. To mitigate the impact of these attacks, the branch regulation mechanism proposed in [21] leverages a small amount of hardware support to track and verify the target addresses of indirect branches. The proposed approach aims to mitigate the impact of such attacks with minimal performance overhead.

The main idea behind branch regulation is to dynamically regulate the execution of indirect branch instructions. These instructions are commonly targeted by code reuse attacks as they allow control flow transfers to non-sequential addresses. By monitoring and regulating these branch instructions, the proposed technique prevents attackers from manipulating control flow to execute malicious code fragments.

The branch regulation mechanism leverages a small amount of hardware support to track and verify the target addresses of indirect branches. This hardware component checks the target addresses against a set of allowed destinations, ensuring that control flow transfers only occur within the permitted range. If an unauthorized target is detected, the mechanism interrupts the execution and triggers an appropriate security response.

The hardware maintains a control flow map that contains information about the valid target addresses for each indirect branch instruction. This map is typically generated during a program's initialization phase or dynamically updated as the program executes.The hardware maintains a control flow map that contains information about the valid target addresses for each indirect branch instruction. This map is typically generated during a program's initialization phase or dynamically updated as the program executes. If the target address of an indirect branch is not within the allowed range, indicating a potential code reuse attack, the hardware interrupts the execution and triggers an appropriate security response. This response may involve raising an exception, terminating the program, or invoking specific security mitigation techniques.

The branch regulation mechanism, with its low overhead and hardware-based approach, is highly relevant to the goals of the Forward-Edge Control-Flow Integrity (CFI) ISA extension. Both approaches share the common objective of enhancing control flow integrity and providing defense against code reuse attacks. By dynamically regulating the execution of indirect branches and verifying the target addresses, the branch regulation mechanism aligns with the principles of the Forward-Edge CFI ISA extension. It strives to ensure that control flow transfers occur within the expected range of valid destinations, preventing attackers from manipulating the control flow and executing malicious code fragments.

3.2 Hard edges

In [4], the authors introduce a hardware-based approach to enforce Control-Flow Integrity (CFI) specifically tailored for embedded devices. Their primary focus (as well as the focus of this work), is the proposed ISA extension that incorporates additional instructions and registers to support CFI enforcement at the hardware level.

By extending the instruction set architecture, the authors provide hardware support for maintaining and protecting the shadow stack. The ISA extension introduces instructions for push and pop operations on the shadow stack and modifies existing instructions to interact with the shadow stack appropriately.

The ISA extension aims to prevent control flow hijacking attacks by ensuring that program execution follows a valid control flow graph. It introduces new instructions that enable the verification of control transfers and the maintenance of integrity checks. The hardware support provided by the ISA extension allows for efficient and low-overhead enforcement of CFI policies, which is crucial for resource-constrained embedded devices.

Overall, the paper presents a comprehensive exploration of hardware-based CFI for embedded devices through the proposed ISA extension. It provides valuable insights into the design, implementation, and evaluation of an ISA extension focused on strengthening control flow integrity.

3.3 Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity

Another closely related work to this thesis revolves around Intel's Control-flow Enforcement Technology (CET) [22], with a particular focus on its significant component known as Indirect Branch Tracking (IBT). IBT is a critical security feature within CET, primarily designed to identify and thwart any attempts to redirect control flow to unintended destinations. It achieves this by introducing novel branch termination instructions: ENDBR32 for 32-bit programs and ENDBR64 for 64-bit programs.

CET, through its Indirect Branch Tracking mechanism, effectively identifies and prevents unauthorized control flow redirection within a program. This

safeguard is enacted by triggering an exception if the instruction at the destination of an indirect call or jump does not match the expected branch termination instruction. This exception mechanism serves as a protective barrier, ensuring that control flow remains within the prescribed boundaries of the program and doesn't deviate towards unauthorized or potentially malicious code segments.

In summary, CET, with its prominent feature IBT, plays a pivotal role in maintaining control flow integrity and security within software programs. By introducing specialized branch termination instructions and employing exception handling, CET effectively mitigates the risks associated with unauthorized control flow redirection.

3.4 Branch Target Instructions (BTIs)

The Arm architecture uses landing pads. Armv8.5-A introduced Branch Target Instructions (BTIs) [18], also referred to as landing pads, as a security measure to defend against Jump-Oriented Programming (JOP) attacks. These BTIs allow the processor to be configured in such a way that indirect branches (specifically, BR and BLR instructions) are only permitted to target landing pad instructions. By restricting the potential destinations for indirect branches to these landing pads, the attack surface is significantly reduced, making it considerably more challenging for attackers to chain together gadgets and create malicious code sequences. The implementation of BTIs is page-specific, controlled by a new bit (GP bit) in the translation tables. This per-page control enables a file system to contain a mix of landing pad-protected code and legacy code. It's important to note that BTI-protected code can still operate on older processors lacking BTI support or when GP=0, albeit without the additional protective measures in place.

3.5 Thesis Approach

This work primarily focuses on Forward-Edge Control Flow Integrity (CFI) and advocates harnessing the capabilities inherent in the CPU's Instruction Set Architecture (ISA) to fortify defenses against control-flow subversion attacks in the style of Jump-Oriented Programming (JOP). The foundation of this approach draws from the RISC-V CFI specification, as proposed by the RISC-V foundation [23]. More specifically, the proposal introduces six novel landing pad instructions geared towards tracking indirect calls and jumps, thereby thwarting

any illicit control flow subversion within a program. The ultimate objective is to secure the approval and ratification of this extension as an integral part of the RISC-V architecture, providing an optimal defense mechanism against such attacks.

Chapter 4

Implementation

This chapter provides an insight into the threat model and outlines the various modifications and extensions required for the successful deployment of Forward-Edge Control-Flow Integrity (CFI) within the CVA6 architecture. Serving as a foundational guide, the chapter commences with an introduction to the threat model adopted for guidance. Subsequently, a comprehensive examination is undertaken, detailing the essential adaptations and additions necessary to seamlessly integrate Forward-Edge CFI ISA extension into the CVA6 architecture.

4.1 Concepts

Here are some important concepts to clarify before the implementation chapter.

- Privilege modes [24]: At any time, a RISC-V hardware thread (hart) operates at a specific privilege level, encoded as a mode in one or more control and status registers (CSRs). RISC-V defines three privilege levels, as outlined in Table 4.1. These privilege levels serve to establish protection boundaries within the software stack. Any attempt to perform operations not allowed by the current privilege mode results in an exception. Typically, these exceptions lead to traps into an underlying execution environment.

Level	Name	Abbreviation
0	User/Application	U
1	Supervisor	S
2	Reserved	
3	Machine	M

Table 4.1: Privilege levels

The machine level has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Machine-mode (M-mode) code is generally considered trustworthy, given its deep access to the machine's implementation. M-mode is often used to manage secure execution environments in the context of RISC-V. User-mode (U-mode) and supervisor-mode (S-mode) are designed for conventional application and operating system usage, respectively. Each privilege level comes with a fundamental set of privileged ISA extensions, along with optional extensions and variants. For instance, machine-mode may support an optional standard extension for memory protection.

Implementations can offer anywhere from 1 to 3 privilege modes, trading off reduced isolation for lower implementation costs, as demonstrated in Table 4.2. M-mode is a mandatory mode in all hardware implementations, as it grants unrestricted access to the entire machine. Simpler RISC-V implementations might solely provide M-mode, which, however, offers no protection against incorrect or malicious application code. Many RISC-V implementations will also include user mode (U-mode) to safeguard the rest of the system from application code. Supervisor mode (S-mode) can be introduced to enforce isolation between a supervisor-level operating system and some execution environment.

Number of levels	**Supported Modes**	**Intended Usage**
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

TABLE 4.2: Privilege levels combinations

Typically, a hart runs application code in U-mode until a trap, such as a supervisor call or a timer interrupt, forces a switch to a trap handler, often operating in a more privileged mode. The hart then executes the trap handler, which eventually resumes execution at or after the original trapped instruction in U-mode. Traps that elevate privilege levels are referred to as vertical traps, while traps that remain at the same privilege level are known as horizontal traps. The RISC-V privileged architecture offers flexible routing of traps to different privilege layers.

- XLEN: In RISC-V, XLEN stands for "Register Width" or "Integer Register Width," and it represents the width (in bits) of the general-purpose integer registers in the RISC-V processor architecture. The XLEN value determines the native word size of the processor and indicates the maximum

bit width that the processor can use for integer operations. RISC-V is designed to be highly customizable, so the specific value of XLEN can vary depending on the implementation and target application. Different implementations of RISC-V processors may support different XLEN values to meet the requirements of various computing environments and performance needs. Common XLEN values include 32, 64, and 128, but other values are possible based on the design choices of the processor. The effective XLEN in M-mode, S-mode and U-mode are termed MXLEN, SXLEN and UXLEN, respectively [25].

- WARL: Certain read/write CSR (Control and Status Register) fields in the RISC-V architecture are designed to be flexible, permitting writes of various values while ensuring that they always return a valid value when read. These fields are denoted as WARL, which stands for "Write Any Values, Reads Legal Values" [26].

  In the case of WARL fields, implementations won't trigger an exception when receiving unsupported values during writes. Instead, they may return any valid value when these fields are read after an illegal write. However, the specific legal value returned should depend in a deterministic manner on both the illegal value that was written and the architectural state of the hardware thread (hart).

- Vtables: In computer programming, a vtable [27], short for "virtual function table", serves as a critical mechanism within programming languages to facilitate dynamic dispatch, often referred to as runtime method binding.

  When a class defines a virtual function or method, many compilers introduce a concealed member variable to the class. This hidden variable points to an array of function pointers, which are essentially pointers to virtual functions. This collection of pointers is known as the virtual method table (vtable). During runtime, these pointers come into play to invoke the appropriate function implementations dynamically. This dynamic invocation is necessary because, during compile time, it may not be determined whether the base function or a derived one implemented by a class inheriting from the base class should be called.

  Vtables are commonly employed in object-oriented programming, and since they essentially are tables with function pointers, they can potentially introduce vulnerabilities susceptible to code reuse attacks.

- RELRO: Relocation Read-Only, commonly known as RELRO [28], is a security measure aimed at rendering specific binary sections as read-only.

  RELRO operates in two distinct modes: partial and full. By default, GCC (GNU Compiler Collection) employs partial RELRO, and it's the configuration you'll encounter in most binaries.

  From a security perspective, partial RELRO introduces minimal changes, but it ensures that the Global Offset Table (GOT) [29] precedes the Block Started by Symbol (BSS) in memory layout. This arrangement eliminates the risk of buffer overflows causing global variable overwrites that could affect GOT entries.

  On the other hand, full RELRO takes the security measures further by enforcing read-only status for the entire GOT. This effectively thwarts "GOT overwrite" attacks, where an attacker attempts to overwrite the GOT address of a function with the location of another function or a Return-Oriented Programming (ROP) gadget they intend to execute.

  It's worth noting that full RELRO is not set as the default compiler option because it can significantly extend program startup times. This delay arises because all symbols must be resolved before the program commences execution. In larger programs containing thousands of symbols requiring linkage, this delay can be noticeable during startup.

- Data-Oriented Programming (DOP) attack: The term "Data-Oriented Programming" (DOP) [30] refers to an attack technique where an attacker manipulates or exploits non-control data in a program to achieve malicious goals. In DOP attacks, the attacker focuses on modifying or controlling data structures and data flow within a program rather than attempting to alter the control flow or execution path of the program.

  Unlike traditional control-flow attacks that target control structures like function calls or branch instructions, DOP attacks specifically aim to manipulate data in such a way that they can achieve unintended behavior within a program. This can include changing the values of variables, modifying data structures, or manipulating input data to trigger vulnerabilities.

  DOP attacks can be challenging to detect and defend against because they do not involve altering the program's control flow, making them less susceptible to traditional control-flow integrity (CFI) defenses.

4.2 Threat Model

The software and firmware threat model in use assumes that an attacker can exploit vulnerabilities found in the target software binary. These vulnerabilities, such as use-after-free issues, and similar weaknesses, could allow the attacker to manipulate critical components like function pointers or VTable pointers. Additionally, it is assumed that the attacker has successfully bypassed Address Space Layout Randomization (ASLR) and possesses comprehensive knowledge of the memory layout. It's important to note that CFI (Control Flow Integrity) operates independently of ASLR and does not interfere with it in any manner.

However, the system enforces certain security measures: (i) the .text segment is non-writable, preventing any modification to the application's code, and (ii) the data segments are non-executable, thereby preventing the execution of injected data, provided it adheres to proper CFI annotations. Regarding linking, it is assumed that the binaries are configured with RELRO [28] to protect the Global Offset Table (GOT) [29] and Procedure Linkage Table (PLT) [31] sections from being tampered with.

For privileged modes, there is no extension of the kernel/firmware using unverified or untrusted kernel/firmware extensions like drivers or modules, as these may disable the aforementioned protections. Additionally, the toolchain used for compiling the binary and the libraries loaded by the binary are trusted to be reliable and free from intentional malicious behavior. At each privilege level, it is assumed that higher privilege levels are not exploited, and the code is trustworthy, free from malware.

It's important to note that this threat model does not encompass Data-Oriented Programming (DOP) attacks, where attackers manipulate non-control data to alter the behavior of a CFI-compliant application. In this context, attackers do not need to deviate from the predefined control flow graph but can change the application's behavior by altering data, like arguments passed to a system call. While Control Flow Integrity can bolster application security, it cannot entirely mitigate DOP attacks. Additionally, techniques based on debugging, emulation, and code injection using hooking techniques are excluded from consideration. Lastly, side-channel techniques allowing arbitrary data accesses fall outside the scope of CFI's intended problem-solving domain.

4.3 Control Transfer Conventions in RISC-V architecture

The RISC-V control-flow integrity (CFI) extension proposal utilizes the conventions and hints provided by the RISC-V architecture to application resilience. The CFI extension takes advantage of the different control transfer instructions available in RV32/RV64 architectures.

In RV32/RV64, there are two types of control transfer instructions: unconditional jumps and conditional branches. Conditional branches are direct branches that are not susceptible to control-flow subversion, as they encode an offset in the immediate field of the instruction.

Unconditional direct jumps (JAL) transfer control to a target within a range of +/- 1 MiB from the current program counter (pc). Unconditional indirect jumps (JALR) calculate their branch target by adding a sign-extended 12-bit immediate, encoded in the instruction, to the value in the rs1 register.

While the RV32I/RV64I instruction set does not have dedicated instructions for procedure calls and returns, the RISC-V ABI (Application Binary Interface) defines conventions for using JAL and JALR instructions. According to the ABI, a JAL or JALR with the rd (link register) set to x1 or x5 is considered a procedure call, and a JALR with rs1 set to the conventional link register (x1 or x5) is considered a return from a procedure. These conventions allow for utilizing hints and supporting return address prediction, as specified in Table 2.1 of the Unprivileged ISA specifications [32].

The RISC-V C extension (RVC) introduces compressed instructions that include unconditional jump (C.J) and conditional branch (C.JAL) instructions. Similar to conditional branches in the base ISA, C.J and C.JAL encode an offset in the immediate field, making them resilient to control-flow subversion.

Additionally, the RISC-V RVC instructions C.JR and C.JALR enable unconditional control transfers within the instruction sequence. C.JR facilitates procedure returns when employed with regular link registers such as x1 or x5. If the instruction uses any other register, it results in an indirect jump. On the other hand, C.JALR not only performs an unconditional jump but also serves as a procedure call by saving the address of the subsequent instruction (pc+2) to the link register x1. This provides an efficient means for managing procedure calls and returns in RISC-V architectures.

The RISC-V control-flow integrity (CFI) extension builds upon these established conventions and hints to further enhance control-flow security. By leveraging the architectural features and specifications, CFI aims to mitigate control-flow hijacking attacks and improve the overall integrity of program execution.

4.4 Forward Edge Control-Flow Integrity

The Forward-Edge Control-Flow Integrity (CFI) mechanism introduces a set of new landing pad instructions that allow software to specify valid targets for indirect jumps within a program. These instructions provide enhanced security by enabling precise control over program execution flow. Six new landing pad instructions have been defined for this purpose, namely LPSLL, LPCLL, LPSML, LPCML, LPSUL, and LPCUL.

Instructions	**Encoded lable**
LPSLL/LPCML	instruction[23:15] (9-bits label)
LPSML/LPCML/LPSUL/LPCUL	instruction[22:15] (8-bits label)

TABLE 4.3: Encoded label in landing pad instructions

Each landing pad is associated with a label, which can be up to 25 bits wide. When the label is 9 bits or less, the LPSLL and LPCLL instructions are used. In cases where a label greater than 9 bits is required, the LPSML and LPCML instruction are utilized too. These instructions encode the middle label as an 8-bit immediate value. Similarly, if a label greater than 17 bits is needed, the LPSUL and LPCUL instructions are, also, employed 4.3. The compiler is responsible for emitting set instructions before an indirect jump takes place, and check instructions at the target address of the indirect jump.

In order to ensure that the target of an indirect call or jump is a valid landing pad instruction, the hardware maintains an expected landing pad (ELP) state. This state helps determine whether a landing pad instruction is required at the target location. When the landing pad feature is active, the hardware maintains an ELP state. This state is updated with the expected landing pad instruction upon encountering indirect calls and jumps. An indirect call or jump updates the ELP state to require an LPCLL instruction at the target. If the instruction at the target does not match LPCLL, an illegal instruction exception is raised, indicating a potential control flow integrity violation. To ensure label matching, a landing pad label register (LPLR) is set up before initiating an indirect call or jump. The LPLR holds the expected landing pad label, and if the label of the

landing pad does not match the one in LPLR, an illegal instruction exception is raised.

4.4.1 Expected Landing Pads - ELP

To track landing pads, the core maintains an expected landing pad (ELP) state to determine the landing pad instruction that is required at the target of a JALR, C.JALR or C.JR instruction. The ELP state can be in one of two states:

1. NO_LP_EXPECTED: This state indicates that a landing pad instruction is not expected at the target location and so, the target can be any valid instruction other than a landing pad instruction.
2. LP_EXPECTED: This state signifies that a landing pad instruction is expected at the target location. The hardware enforces that the target must be a valid landing pad instruction in order to comply with this state.

An indirect jump occurs when a JALR/C.JALR/C.JR instruction is encountered, and the destination register (rd) is not equal to x1 or x5, and the source register (rs1) is not equal to x1 or x5. In such cases, the expected landing pad (ELP) field is updated to LP_EXPECTED. If the ELP is set to LP_EXPECTED and the next instruction decoded is not LPCLL, an illegal instruction exception is raised. This exception indicates that a valid landing pad instruction was expected, but a different instruction type was encountered. If the target of the indirect call/jump is a valid landing pad instruction, the expected label established in the LPLR is matched with the target's label. If a mismatch is detected then the label check instruction causes an illegal instruction exception.

4.4.2 Landing Pad Label Register - LPLR

The LPLR CSR 4.1 (Control and Status Register) is a user-mode read-write (URW) register and it is responsible for holding the expected label at the target of an indirect jump. The label is split into a lower label (LL) - 9 bits wide, a middle label (ML) - 8 bits wide and an upper label (UL) - 8 bits wide. To ensure label matching, the landing pad label register (LPLR) is set up before initiating an indirect call or jump, by the set landing pad instructions (LPSLL, LPSML and/or LPSUL). The LPLR holds the expected landing pad label, and if the label embedded in the check landing pad instructions (LPCLL, LPCML and/or

LPCUL) does not match the one in LPLR, an illegal instruction exception is raised.

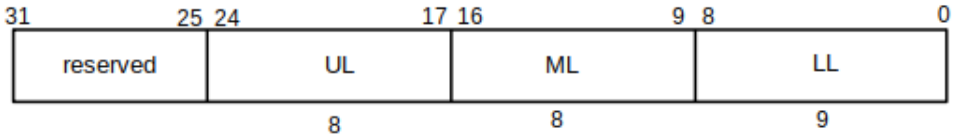


FIGURE 4.1: LPLR CSR for RV32 and RV64

4.4.3 Landing Pad CSRs & FCFIE bit

Machine environment configuration registers (menvcfg and menvcfgh) 4.2

The menvcfg [32] CSR is an MXLEN-bit read/write register, that controls certain characteristics of the execution environment for modes less privileged than M.

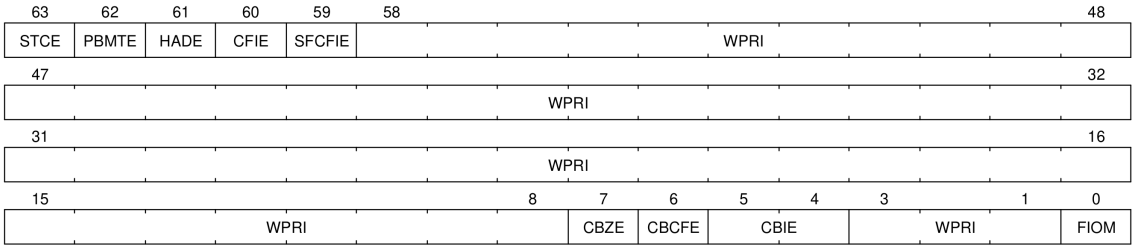


FIGURE 4.2: Machine environment configuration register (menvcfg) for MXLEN=64

The availability of the Forward-Edge CFI extension in modes lower privileged than M is controlled by the CFIE field, located at bit 60. When CFIE is set to 1, the Forward-Edge CFI functionality can be utilized in S-mode by enabling the SFCFIE field, located at bit 59. When the menvcfg. CFIE bit is set to 0, certain rules apply to privilege modes lower than M:

- Instructions related to the Forward-Edge CFI extension will behave according to the defined behavior of Zimops instructions.
- The SFCFIEN and SPELP fields in mstatus are read-only zero.

Machine Security Configuration (mseccfg) 4.3

mseccfg [32] is an optional MXLEN-bit read/write register, formatted as that controls security features. When MXLEN=32 only, mseccfgh is a 32-bit read-/write register that contains the same fields as mseccfg bits 63:32 when MXLEN=64.

The MFCFIE (bit 10) is a WARL field that when set to 1 enables forward-edge CFI at M-mode.

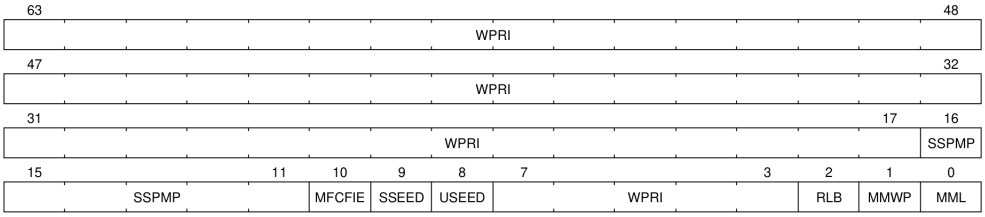


FIGURE 4.3: Machine security configuration register (mseccfg) when MXLEN=64

Machine status registers (mstatus) 4.4

The mstatus [32] register is an MXLEN-bit read/write register that keeps track of and controls the hart's current operating state. A restricted view of mstatus appears as the sstatus register in the S-level ISA.

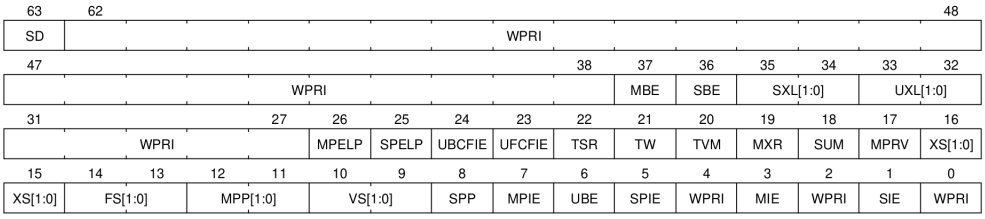


FIGURE 4.4: Machine-mode status register (mstatus) for RV64

The UFCFIE (bit 23) is a WARL field that, when set to 1, enable forward-edge CFI in U-mode.This means that the system will enforce the presence of landing pad instructions for indirect calls and jumps in U-mode, ensuring the integrity of control flow.

The SPELP (bit 25) and MPELP (bit 26) are also WARL fields that hold the previous Expected Landing Pad (ELP) value. These fields are updated according to the specifications outlined in Section 4.1.5 of the documentation.

The xPELP fields, including SPELP and MPELP, are encoded with the following values:

- 0 - NO_LP_EXPECTED: This indicates that no landing pad instruction is expected at the current point in the control flow.
- 1 - LP_EXPECTED: This indicates that a landing pad instruction is expected at the current point in the control flow.

Supervisor status registers (sstatus) 4.5

The sstatus [32] register is an SXLEN-bit read/write register. The sstatus register keeps track of the processor's current operating state.

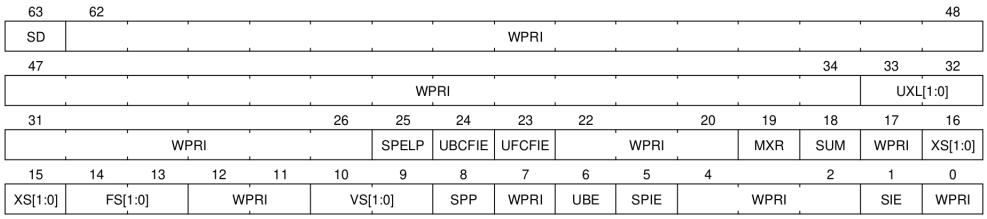


FIGURE 4.5: Supervisor-mode status register (sstatus) when SXLEN=64

When menvcfg.CFIE is 1, access to the following fields accesses the homonymous field of the mstatus register. When menvcfg.CFIE is 0, these fields are read-only zero.

- UFCFIE (bit 23).
- SPELP (bit 25).

FCFIE bit

When the privilege mode is M, the activation of Forward-Edge Control Flow Integrity (CFI) depends on the value of the MFCFIE bit in the mseccfg register. If MFCFIE is set to 1, Forward-Edge CFI is active in M-mode. However, when the menvcfg.CFIE bit is 0, the Forward-Edge CFI extension is not enabled for privilege modes lower than M. If menvcfg.CFIE is set to 1, Forward-Edge CFI is active in S-mode only if the menvcfg.SFCFIE bit is also set to 1. Similarly, in U-mode, forward-edge CFI is active if the mstatus.UFCFIE bit is set to 1.

The FCFIE bit is used to determine if Forward-Edge CFI is active at privilege level x and is defined as follows:

```
if ( privilege == M-mode )
        FCFIE = mseccfg.MFCFIE
else if ( menvcfg.CFIE == 1 && privilege == S-mode )
        FCFIE = menvcfg.SFCFIE
else if ( menvcfg.CFIE == 1 && privilege == U-mode )
        FCFIE = mstatus.UFCFIE
else
        FCFIE = 0
```

4.4.4 Landing Pad Instructions

The forward-edge CFI introduces the following instructions for landing pad operations 4.6. All instructions are encoded using the SYSTEM major opcode.

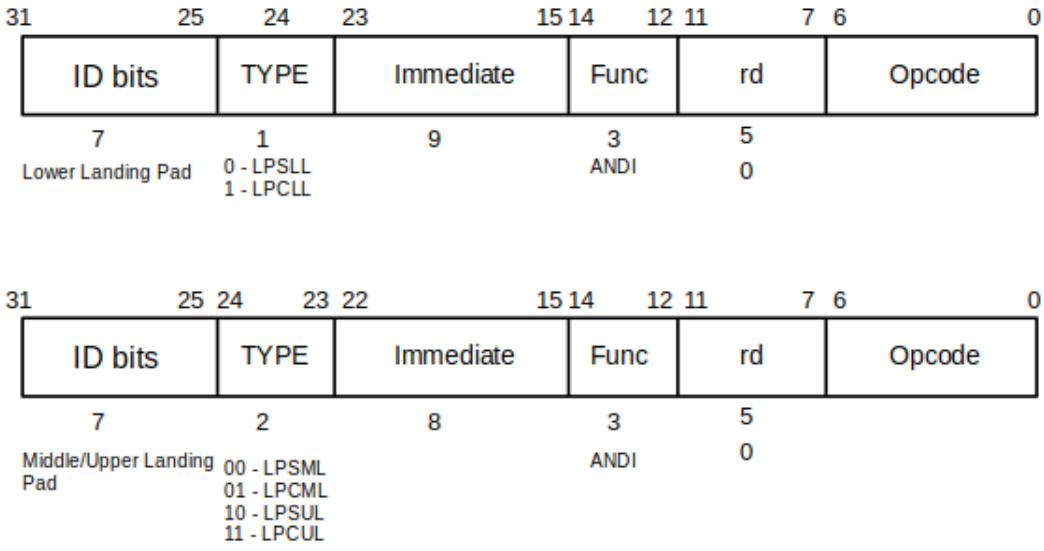


FIGURE 4.6: Format of CFI Instructions

Before performing an indirect call or indirect jump to a labeled landing pad, the LPLR is loaded with the expected landing pad label - a constant determined at compilation time. The LPSLL instruction is used for this purpose.

```
If FCFIE != 0
        LPLR.LL = LPSLL.immediate
else
        [rd] = 0;
endif
```

The LPCLL instruction serves as a valid landing pad for indirect jumps and indirect calls. However, there are specific requirements and behaviors associated with this instruction.The LPCLL instruction includes a lower landing pad label embedded in the immediate field. This label indicates the destination of the landing pad. It is important to note that the label needs to match the LPLR.LL field, otherwise, executing the LPCLL instruction will trigger an illegal instruction exception. In the case of an illegal instruction exception caused by executing LPCLL instructions, the ELP remains unchanged.

```
If FCFIE != 0 // If lower landing pad label not matched -> illegal-instruction
        if (LPCLL.immediate != LPLR.LL)
                Cause illegal-instruction exception
        else
                ELP = NO_LP_EXPECTED
        else
                [rd] = 0;
endif
```

Similarly, the LPSML and LPSUL instructions are used to set the corresponding fields ML and UL of the LPLR CSR and the the LPCML and LPCUL instructions, perform a check operation on the corresponding fields. A summary of the functionality of Forward-Edge CFI instructions can be found in Table 4.4.

Instruction	**Functionality**
LPSLL (set lower landing pad)	LPLR.LL = immediate
LPCLL (check lower landing pad)	If LPLR.LL!=LPCLL.immediate; exception
LPSML (set middle landing pad)	LPLR.ML = immediate
LPCML (check middle landing pad)	If LPLR.ML!=LPCML.immediate; exception
LPSUL (set upper landing pad)	LPLR.UL = immediate
LPCUL (check upper landing pad)	If LPLR.UL!=LPCUL.immediate; exception

Table 4.4: Functionality of CFI instructions.

The compiler may emit the following instruction sequence at indirect call sites to set up the Landing Pad Label Register (LPLR) when using labels that are up to 17 bits wide:

```
:
# x10 is expected to have address of function bar()
lpsll $0x1de
# setup lplr.LL with value 0x1de
lpsml $0x17
# setup lplr.ML with value 0x17
jalr %ra, %x10
:
```

The following instruction sequence may be emitted at indirect call sites by the compiler to set up the landing pads at entrypoint of function bar():

```
bar:
        lpcll $0x1de
        lpcml $0x17
        :
                # Verifies that LPLR.LL matches 0x1de
                # Verifies that LPLR.ML matches 0x17
                # continue if landing pad checks succeed
```

4.4.5 Preserving expected landing pad state on traps

A trap may need to be delivered to the same or higher privilege level on completion of JALR but before the instruction at the target of JALR was decoded.To ensure proper trap handling and preserve the ELP state during the execution of JALR instructions, the mstatus CSR (Control and Status Register) includes the MPELP and SPELP fields. In M-mode and S-mode respectively, these bits indicate the ELP of the current privilege level and are used to deliver traps to the same or higher privilege level on completion of JALR instructions. The xPELP fields in mstatus are WARL fields, meaning they can be written with any value, but the read value is subject to certain restrictions. When a trap is taken into privilege mode x, the trap handler should preserve the LPLR CSR, which holds the Landing Pad Label and the xELP bits are updated with current ELP and ELP is set to NO_LP_EXPECTED. This ensures that the current landing pad label and associated information are not lost during the trap handling process.To return from a trap in M-mode or S-mode, the MRET and SRET instructions are used, respectively.When executing an xRET instruction, where x represents M or S, the ELP is set to the corresponding xPELP value, and

xPELP is set to NO_LP_EXPECTED. This update ensures that the correct ELP is restored after returning from the trap.

4.5 CVA6 & Implementation

The CVA6 4.7 [14] core was used as a reference design upon which the CFI logic described in the previous section was implemented. The core's datapath consists of 6 stages, the PC generation stage and instruction fetch (IF) stage,both included on the front-end region, the instruction decode (ID) stage, the issue stage, the execution stage and the commit stage.

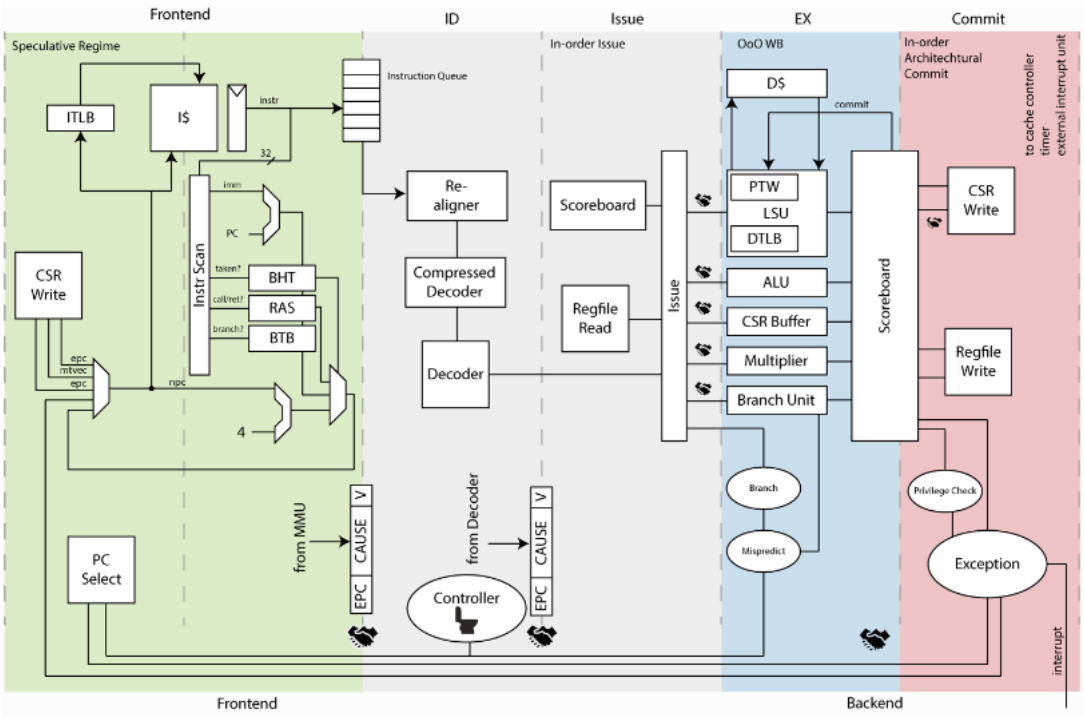


FIGURE 4.7: Datapath of CVA6 core

- Front-end:

  The front-end 4.8 region of the core is responsible for generating the next program counter (PC) in a computer system. The PC represents the logical address of the next instruction to be executed. This stage includes the PC Generation unit, which interacts with the Instruction Fetch (IF) stage and the Memory Management Unit (MMU) for address translation.

  The PC Generation unit incorporates speculation on the branch target address, meaning it predicts the target address of a branch instruction before it is resolved. It also determines whether a branch is taken or not. The IF stage communicates with the PC Generation unit and requests

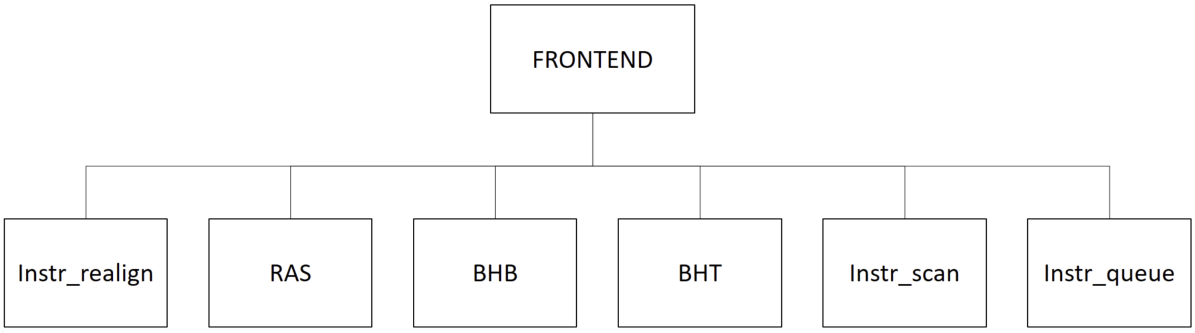


FIGURE 4.8: Front-end high level block diagram

address translation from the MMU for the desired PC. It also controls the interface to the instruction memory (I$).

When the IF stage wants to fetch an instruction from memory, it signals the I$ interface. However, the request may or may not be granted depending on the cache's state. If the request is granted, the instruction fetch stage places the request in an internal First-In-First-Out (FIFO) buffer. This FIFO buffer has two ports, allowing a maximum of two outstanding transactions at a time. If there are already two outstanding transactions, the IF stage will not acknowledge any new requests from the PC Generation unit. Once a valid response is received from memory, the fetched instruction, along with its fetch address and branch prediction information, is stored in the FIFO buffer.

Before advancing to the next stage, a check is performed to determine if a Jump and Link Register (JALR), a C.JR or a C.JALR instruction with rs1 equal to one of the conventional long registers, x1 or x5, occurs. If this condition is true, the core enters an architectural state called the Expected Landing Pad (ELP). It signals the CSR Register file and sets the expected landing pad to LP_EXPECTED. While CFI (Control Flow Instructions) instructions continue to be processed, the architectural state remains the same. In this way, all labels are included in the control flow graph. The next instruction is expected to be a CFI instruction; otherwise, an illegal instruction exception will occur.

The Finite State Machine (FSM) 4.9 described above is implemented in the Instruction Queue module and is responsible for managing the Expected Landing Pad (ELP) state in a program. The FSM consists of six states, which are as follows:

– 000 - NO_LP_EXPECTED : The initial state of the FSM is the idle

state, during which instructions are fetched and the FSM remains in the NO_LP_EXPECTED state. When an indirect jump occurs, the CFI process begins, transitioning the FSM to the WAIT_STATE.

- 001 - WAIT_STATE : In CVA6, all jumps are treated as conditional jumps, which means that the processor requires some cycles to determine whether the jump will be taken or not. By default, unconditional jumps are taken, but due to the processor's design, irrelevant instructions are still being fetched, until the jump is resolved. This creates a stall state until the jump is resolved. Once the jump is resolved and the program counter (PC) points to the correct address, the FSM transitions to the L_LP_EXPECTED_STATE.

  If an interrupt occurs the ELP state is updated to NO_LP_EXPECTED.

- 010 - L_LP_EXPECTED_STATE : In this state, the Forward-Edge Control Flow Integrity (CFI) checks start. The initial comparison occurs between the instruction located at the target address of the indirect jump and an LPCLL format. If the fetched instruction does not match the LPCLL format, the next state will be ILLEGAL_INSTRUCTION_STATE. However, if the instruction at the target address is an LPCLL instruction, the immediate value of this instruction is compared against the lower label (9 bits) stored in LPLR CSR. If there is a match between the immediate value and the label, the next state will be M_LP_EXPECTED. On the other hand, if there is no match, the next state will be ILLEGAL_INSTRUCTION_STATE.

- 011 - M_LP_EXPECTED_STATE : In this state, the comparison is directed towards the subsequent instruction following LPCLL. If this instruction is not LPCML, it implies that only the lower label is in use. As a result, the system transitions to the NO_LP_EXPECTED_STATE, signifying a successful completion of the control flow. On the other hand, if the instruction is indeed a LPCML instruction, the 8-bit label embedded in the instruction's raw bits is matched against the 8-bit middle label stored in the LPLR CSR. A match directs the next state of the FSM to U_LP_EXPECTED_STATE, while a mismatch leads to a transition to the ILLEGAL_INSTRUCTION_STATE.

- 100 - U_LP_EXPECTED_STATE : Similarly, in this state, the

comparison is directed towards the subsequent instruction following LPCML. If this following instruction is not LPCUL, it implies that the 17-bit label (comprising the lower and middle labels) is exclusively in use. As a result, the system seamlessly transitions to the NO_LP_EXPECTED_STATE, marking the successful conclusion of the control flow process. Conversely, if the instruction is indeed LPCUL, the instruction's raw bits containing the 8-bit label are matched against the 8-bit upper label stored in the LPLR CSR. A successful match guides the finite state machine (FSM) into the NO_LP_EXPECTED_STATE, thereby signifying the continuation of the control flow, now employing a 25-bit label. However, in cases where the labels do not align, a transition occurs to the ILLEGAL_INSTRUCTION_STATE, indicating a divergence from the anticipated control flow.

– 101 - ILLEGAL_INSTRUCTION_STATE : In this state an illegal instruction exception is raised is raised.

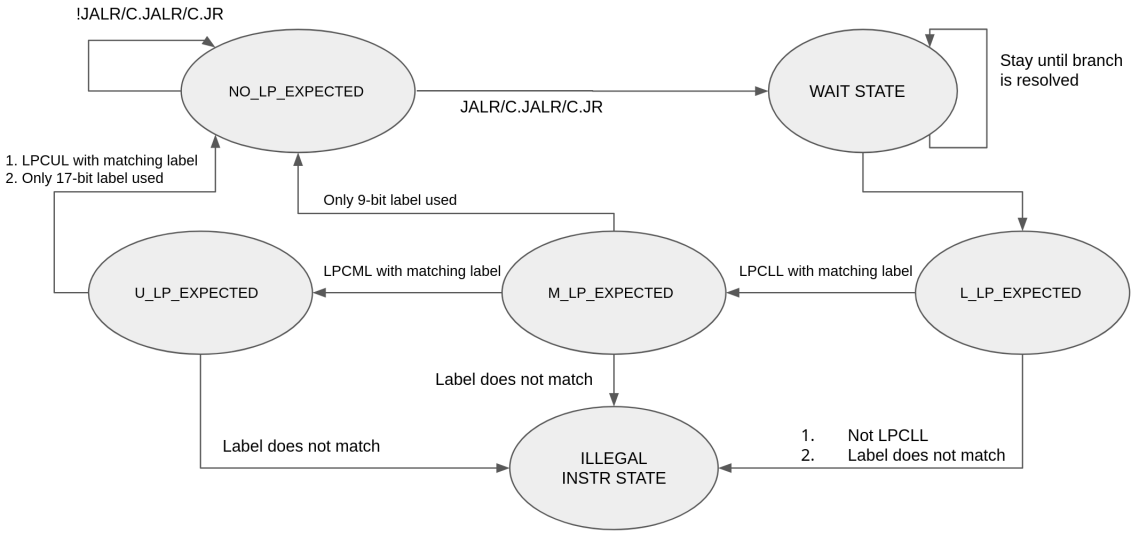


Figure 4.9: Landing Pad FSM

- ID Stage:

  The Instruction Decode (ID) stage is the first pipeline stage in the processor's back-end. Its primary objective is to receive instructions from the data stream provided by the Instruction Fetch (IF) stage, decode them, and then forward them to the issue stage. Additionally, this stage handles the realignment and decompression of potentially compressed 16-bit instructions.

During the decoding procedure, the decoder converts the raw instruction bits into a scoreboard entry. In Ariane, the scoreboard entry is a control structure that contains information about the Program Counter (PC), Functional Unit (FU), Operation (OP), Register Source 1 (RS1), Register Source 2 (RS2), Register Destination (RD), type of immediate value, exception handling, and branch prediction information, among other details.

The CFI (Control Flow Integrity) instructions have the major opcode "SYSTEM" and are classified as CSR (Control and Status Register) instructions. These instructions are responsible for updating CSRs and controlling the core's architectural state. To implement the functionality of each instruction, all the fields of these instructions are appropriately assigned in the control structures of the core, extending down to the datapath.

- Issue Stage:

  The Issue stage is responsible for managing data information after the ID stage. Its main objective is to receive decoded instructions and distribute them to various functional units. Additionally, the Issue stage keeps track of all issued instructions, monitors the status of functional units, and receives write-back data from the execute stage using a scoreboard. It also contains the CPU's register file.

  During the Issue stage, the relevant parts of the instruction are assigned to the operands (operand A and/or operand B) that will be utilized in subsequent stages of the datapath to perform the operation specified by each instruction. In the case of CFI instructions, the label is assigned to operand A.

- EX Stage:

  The execute stage is a critical stage in the processor pipeline that houses various functional units (FUs). Each unit operates independently, performing its specific operation. The execute stage consists of an Arithmetic Logic Unit (ALU), a branch unit, a load-store unit (LSU), a multiply/divide unit, and a CSR buffer. For the purpose of this work, the CSR buffer has been modified.

  The CSR buffer serves as a dedicated unit to store the address of the CSR register that the instruction intends to read from or write to. This

buffering is necessary because CSR instructions modify the architectural state, so the instructions must be buffered and executed only when the commit stage decides to commit them, considering the structure of the scoreboard entry. Typically, CSR instructions include the address of the CSR they are going to modify within their raw bit stream. However, in the CFI (Control Flow Integrity) instructions, this information is not present, and the address of the LPLR CSR is hard-wired. To avoid cluttering the scoreboard with special case bit fields, the CSR buffer is introduced. It simply holds the address, and if the CSR instruction is scheduled for execution, it uses the stored address from the buffer.

One clear disadvantage of the CSR buffer is that it consists of only a single element, preventing the execution of consecutive CSR instructions without a pipeline stall. However, since CSR instructions are relatively rare, this limitation is not a significant problem. Additionally, certain CSR instructions may cause a pipeline flush regardless.

- Commit Stage:

  The Commit Stage is the final stage in the processor's pipeline and plays a crucial role in updating the architectural state. It receives incoming instructions and is responsible for carrying out actions that affect the architectural state, such as writing to CSR registers, committing store operations, and updating the register file. Importantly, only the Commit Stage is allowed to modify the architectural state of the core.

  In addition to handling state updates, the Commit Stage also has control over the overall stalling of the processor. If the halt signal is asserted, indicating a need to halt the pipeline, the Commit Stage will not commit any new instructions. This generates back-pressure and eventually leads to a stall in the pipeline, allowing for proper synchronization and handling of exceptional conditions.

  The Commit Stage also maintains extensive communication with the controller. It works closely with the controller to execute fence instructions, which are used for cache flushes and other pipeline resets. By coordinating with the controller, the Commit Stage ensures the proper execution of these instructions and the overall synchronization and consistency of the pipeline.

- CSR Register File:

The CSR Register file serves as a critical module responsible for manipulating the architectural state of the core and managing updates to the Control and Status Registers (CSRs). In order to support the Forward-Edge CFI extension, the CVA6 core, which implements the draft privilege extension 1.10, is undergoing specific modifications. To begin with, the read and write processes of the CSR Register File now include the addition of machine configuration registers (menvcfg, menvcfgh, and mseccfg) as outlined in the current Privileged Specification version 20211203 [32]. These registers exclusively implement and manage the CFI relevant bits to enable the privileged functionality of the Forward-Edge CFI extension.

Furthermore, the LPLR CSR, responsible for holding the landing pad label, has also been integrated into the read/write processes of this module. Additionally, the implementation of the xFCFIE bit takes place within this module, as it relies on information from the machine configuration registers and the privilege level of the core. This bit plays a crucial role in restricting the execution of CFI instructions if there is no valid extension of CFI supported on the core or if such instructions are attempted to be executed at an unauthorized privilege level.

Moreover, this module takes on the responsibility of handling traps efficiently. Several statements have been carefully added to ensure the correct behavior of the core when operating in the ELP state, guaranteeing its overall reliability and functionality. These modifications collectively enhance the core's capabilities, enabling seamless support for the Forward-Edge CFI extension and ensuring the desired system behavior and privilege management.

4.6 Tools Used

4.6.1 Hardware Tools

To test and verify the Forward-Edge CFI ISA extension in the CVA6 core, a variety of hardware tools were utilized. These tools included Vivado, which is a widely used FPGA design and implementation tool, Verilator, a cycle-accurate simulator for verifying the hardware design, OpenOCD, a debugger commonly used for debugging and testing embedded systems, GTKwave, a waveform viewer for analyzing simulation results, and RISC-V PK, a minimal operating system or runtime environment that serves as a proxy for running RISC-V programs on a host machine.

Vivado Design Suite

Vivado Design Suite HLx Editions 2018.3 [33] is a software suite developed by Xilinx for designing and implementing digital circuits using their FPGAs and SoCs. Key components of Vivado Design Suite include design entry methods such as schematic-based entry, HDL entry, and high-level synthesis (HLS). The IP Integrator tool simplifies design integration using pre-designed IP blocks. The suite performs synthesis and optimization to convert RTL into gate-level netlists, and handles implementation including place and route.

In this project, Vivado Design Suite HLx Editions 2018.3 was utilized for various tasks. Firstly, Vivado was employed for the implementation and synthesis of the core, including the integration of the Forward Edge CFI ISA extension. It facilitated the creation of the bitstream required for programming the FPGA. Additionally, Vivado was utilized for JTAG programming of the FPGA with the generated bitstream. Furthermore, the suite was instrumental in generating reports related to the new design, providing valuable information on power consumption, area utilization, place and route, and other relevant metrics.

Verilator

Verilator [34] is an open-source simulator and synthesizer for digital designs described in Verilog and SystemVerilog. It provides high-speed simulation by generating optimized C++ or SystemC code. Verilator supports standard Verilog and SystemVerilog language constructs and offers cycle-accurate simulation for accurate verification and debugging. It can seamlessly integrate with C++ and SystemC code for easy hardware-software co-design. Verilator includes a static analyzer for detecting RTL issues, supports code coverage and functional coverage analysis, and is cross-platform compatible. As an open-source tool, it encourages community collaboration and customization.

In this work, Verilator version 4.110 was utilized as the primary tool for simulation and ensuring the functionality of the core, through a test harness, which is a structured environment for executing test cases and capturing the results for analysis. Verilator 4.110 provided a reliable and efficient platform for simulating the digital design described in SystemVerilog. By running the design through Verilator's simulation engine, the core's behavior and functionality could be thoroughly verified. Verilator's features, including high-speed simulation and cycle-accurate modeling, contributed to the accurate assessment of the core's performance and functionality.

RISCV-PK and Bootloader

The RISC-V Proxy Kernel and Boot Loader [35] is a software framework that enables the execution of user-space binaries on RISC-V processors. It provides an interface between the user application and the underlying hardware, handling system calls, memory management, and other operating system-like functions. The Proxy Kernel acts as a bridge between the user application and the host operating system, facilitating the execution of user programs.

The RISC-V Proxy Kernel and Boot Loader were utilized to validate the proper functionality of the core in conjunction with an operating system-like environment. By employing the RISC-V Proxy Kernel and Boot Loader, the core's behavior and performance could be thoroughly assessed within a comprehensive operating system framework.

GTKWave

GTKWave [36] is an open-source waveform viewer used for visualizing and analyzing digital simulation waveforms. It offers features such as waveform visualization, hierarchical viewing, zoom and navigation, signal analysis, cross-probing, and customization. In this work, GTKWave was utilized as a waveform viewer to analyze the behavior of the core when running applications with or without an operating system.

4.6.2 Software tools

RISC-V Tests

RISC-V tests [37] are a collection of software programs designed to verify the correctness and functionality of RISC-V processors and related systems. They serve as a suite of test cases that cover various aspects of the RISC-V architecture, including instruction set implementation, system calls, memory operations, interrupts, and more. CVA6 core has been verified by the dhrystone, median, mm, mt-matmul, mt-vvadd, multiply, pmp, qsort, rsort, spmv, towers and vvadd benchmarks. Those test were used to verify the functionality of the Forward Edge CFI ISA Extension and measure the overhead in performance.

Compiler

The benchmarks in this thesis were compiled using a RISC-V GNU Compiler Toolchain that includes the Control Flow Integrity (CFI) extension. This compiler was provided by Rivos Inc. [38]. During my internship at Rivos Inc., which

lasted for three months, I worked on implementing the Forward and Backward Edge CFI ISA extension based on the specification proposed by the company within the RISC-V community. This thesis follows the guidelines and specifications outlined in that proposal.

The compiler enforces the Forward Edge CFI ISA extension by inserting the set labels instructions before the indirect jumps in a program and the check labels instructions in the prologue of every function.

4.7 FPGA Platform

FPGAs (Field-Programmable Gate Arrays) [39] are reprogrammable integrated circuits used in digital design. They offer flexibility, allowing for the implementation of custom digital logic circuits through programmable logic blocks and interconnects. FPGAs provide advantages such as reconfigurability, customizability, parallelism, prototyping capabilities, and hardware acceleration. They find applications in various fields and enable the development of optimized and specialized hardware solutions. This work utilises the Genesys II FPGA board.

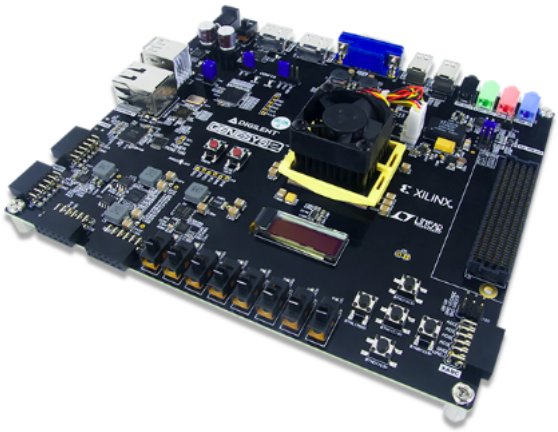


FIGURE 4.10: Genesys II board

4.7.1 Genesys II

The Genesys 2 board is a development platform by Digilent, featuring a powerful Xilinx Kintex-7 FPGA. It offers extensive connectivity options, including Ethernet, USB, HDMI, and Pmod connectors. With DDR3 and Flash memory resources, high-speed transceivers, and programmability options, the Genesys 2 board provides a versatile platform for FPGA-based projects. Its user-friendly

interface, expansion capabilities, and rich feature set make it well-suited for complex digital design development and prototyping.

Chapter 5

Evaluation

This chapter presents an extensive description and analysis of all benchmarks executed in Verilator 4.110 with and without an operating system, alongside their results. Additionally, it conducts a thorough analysis of the static results generated by Vivado 2018.3. The primary focus lies on revealing the minimal performance impact introduced by the Forward-Edge CFI ISA extension on CVA6 RISC-V softcore and validating its effectiveness.

5.1 Case Study

To evaluate the Forward-Edge Control Flow Integrity (CFI) ISA extension's effectiveness, a simple test incorporated two indirect function calls. This test underwent compilation using the riscv64-unknown-elf-gcc compiler, which produced a statically linked assembly file. Subsequently, this generated file underwent another compilation step to produce an executable file. The dual-step compilation was necessary because the riscv64-unknown-elf-gcc compiler does not support the CFI ISA extension. Instead, the landing pad instructions were inserted at the correct location in the code using the ".long" assembly directive.

To assess the Forward-Edge CFI ISA extension's effectiveness, the testbench rigorously examines various scenarios. These encompass verifying label matching with 9-bit, 17-bit, and 25-bit labels to ensure consistency in control flow. It also scrutinizes cases of label mismatch, where control flow may diverge unexpectedly. Furthermore, the absence of an LPCLL instruction at the target address location is investigated, potentially indicating an invalid redirection of execution flow. Lastly, the testbench includes a test for interrupt handling, ensuring that the CFI extension can maintain control flow integrity even when interrupts occur, thus providing a comprehensive evaluation of the extension's capabilities.

test.c

```
int main() {
        add(8,3);
        return foo();
}
int bar(int a){ return a; }
int foo(){
        int (*fp)(int) = bar;
        int res = fp(4);
        return res;
}
int add(int a, int b){
        int (*fp)(int,int) = sub;
        int res = fp(a,b);
        a = res + b;
        return a;
}
int sub(int a, int b){
        a= a - b;
        return a;
}
```

test.s in RISC-V assembly

```
main:
        :
        call add
        call foo
        :
bar:
        :
         jr ra
foo:
        :
        jalr a5 #bar
        :
add:
        :
        jalr a5 #sub
        :
sub:
        :
         jr ra
```

The test encompasses two indirect jumps, addressing two scenarios in each run. In the initial run, it validates the control flow for matching labels to guarantee consistency. This occurs during the first indirect call (from "add" to "sub", utilizing only a 9-bit label. In the second indirect jump, there's an absence of an LPCLL instruction at the target address location (from "foo" to "bar"), indicating a potential redirection of the execution flow.

```
add:
    :
    .long 0x82044073 #lpsll 8
    jalr a5 #sub
    :
sub:
    .long 0x83044073 #lpcll 8
    :
     jr ra
```

```
bar:
    :
     jr ra

foo:
    :
    .long 0x82014073 #lpsll 2
    jalr a5 #bar
    :
```

The waveform 5.1 depicts the flow of an indirect jump from the "add" function to the "sub" function. The initial six signals pertain to the instruction queue module, while the last two belong to the CSR register file module. Within the waveform, red boxes represent the LPSLL, JALR, and LPCLL instructions, respectively. The fetch_entry_valid_o signal identifies instructions advancing to the next pipeline stages.

The curr_state and next_state signals indicate the current and upcoming states of the CFI FSM. Beginning in the NO_ELP_EXPECTED_STATE(000), both the LPSLL and JALR instructions pass through the instruction queue (since fetch_entry_valid_o is set to 1). A JALR instruction triggers a transition to the WAIT_STATE(001), awaiting resolution of the branch as taken.

Upon a branch taken resolution (signaled by the move signal turning to one, depicted in yellow), the FSM enters the L_LP_EXPECTED_STATE(010). At this point, the label(LL signal) is written into the LPLR CSR , and the lp_exp register stores the landing pad's expected architectural state within the CSR register file.

When fetch_entry_valid_o again indicates an instruction entering the instruction queue, it is expected to be an LPCLL instruction with a matching label. Since,it is an LPCLL instruction performed. the label comparison occurs at this stage, transitioning the FSM to the M_LP_EXPECTED_STATE(011). In this state, since only the 9-bit label is utilized, the FSM moves forward to the NO_ELP_EXPECTED_STATE(000), completing the CFI process. Both the label and the lp_exp register are cleared in this final phase.

The waveform 5.2 depicts the flow of an indirect jump from the "foo" function to the "bar" function. The analysis includes the same signals, along with the signals

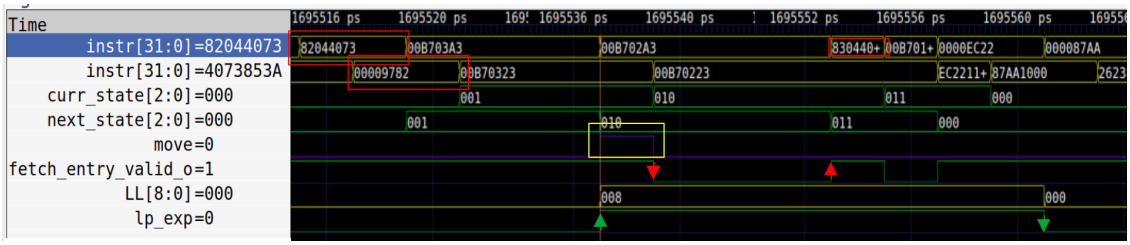


FIGURE 5.1: Test 1: Lower Label match

instruction and exception cause that tag instructions leaving the instruction queue and proceeding through the pipeline.

Following the indirect jump, the FSM enters to WAIT_STATE (001) where it waits for branch resolution. When the move signal becomes one, the FSM transitions to the L_LP_EXPECTED_STATE (010). In this state, the label is written into the LPLR CSR, and the lp_exp register retains the expected architectural state for the landing pad in the CSR register file.

When fetch_entry_valid_o equals to one, the incoming instruction is examined to determine if it is a landing pad instruction. In the scenario at hand, the absence of the LPCLL instruction prompts the FSM to shift to the ILLEGAL_INSTRUCTION_STATE (101). Both the architectural state of the core and the content of the LPLR CSR (with LL=002), remain unaltered. The instruction departing from the instruction queue carry an exception cause of 2 (as shown by the white box). This indicates an illegal instruction exception, and the instruction proceeds through the pipeline tagged as an illegal instruction. Given that the test operates within the proxy kernel, the exception will be managed by the trap handler.

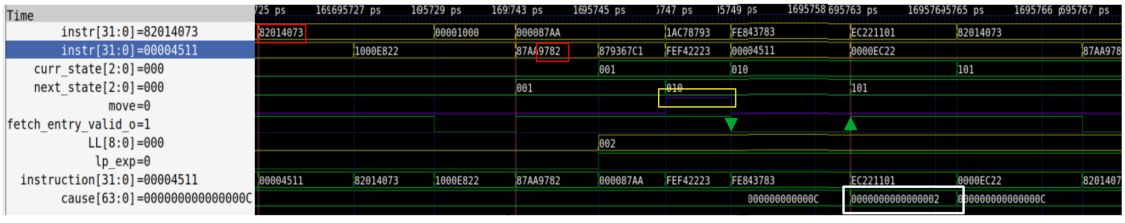


FIGURE 5.2: Test 1: Absence of LPCLL instruction in target address

In the second test run, the analysis encompasses two specific scenarios. The first scenario involves the validation of control flow integrity when labels match within a 17-bit wide context. This occurs during the transition from the "add" function to the "sub" function. The second scenario focuses on a label mismatch,

particularly concerning a 9-bit wide label, as the control flow proceeds from the "foo" function to the "bar" function.

```
add:
 :
 .long 0x82044073 #lpsll 8
 .long 0x8600C073 #lpsml 1
 jalr a5 #sub
 :
sub:
 .long 0x83044073 #lpcll 8
 .long 0x8680C073 #lpcml 1
 :
  jr ra
```

```
bar:
 .long 0x8301C073 #lpcll 3
 :
foo:
 :
 .long 0x82014073 #lpsll 2
 jalr a5 #bar
 :
```

In this waveform 5.3 analysis, the focus lies on examining the behavior of the core during the execution of an indirect jump secured by a 17-bit label. Following the entry of LPSLL and LPSML instructions (indicated by the red boxes) into the pipeline, an indirect jump (also marked in red) initiates a shift in the FSM to the WAIT_STATE (001). Upon the resolution of the branch, the move signal activates (highlighted in yellow), prompting the FSM to advance to the L_LP_EXPECTED_STATE (010). At this stage, updates occur for both the lower and middle label, as well as the lp_exp register, within the CSR register file.

When fetch_entry_valid_o equals 1, and the current instruction corresponds to an LPCLL instruction, label comparisons are carried out. In the event of a match, the FSM progresses to the M_LP_EXPECTED_STATE (011). Similarly, upon fetch_entry_valid_o being set to 1 once more, and the current instruction is recognized as an LPSML instruction, label comparisons are executed and matched. Consequently, the FSM transitions to the U_LP_EXPECTED_STATE (100). Since there is no upper label to validate, the CFI process concludes, and the FSM shifts to the NO_LP_EXPECTED_STATE (000). Once again, both the label and the lp_expected register are cleared within the CSR register file.

Following the execution of the indirect jump from "foo" to "bar" 5.4, the anticipation is for an illegal exception to occur. As previously detailed, following the resolution of the indirect jump (indicated by move=1), the FSM transitions to the L_LP_EXPECTED_STATE (010).

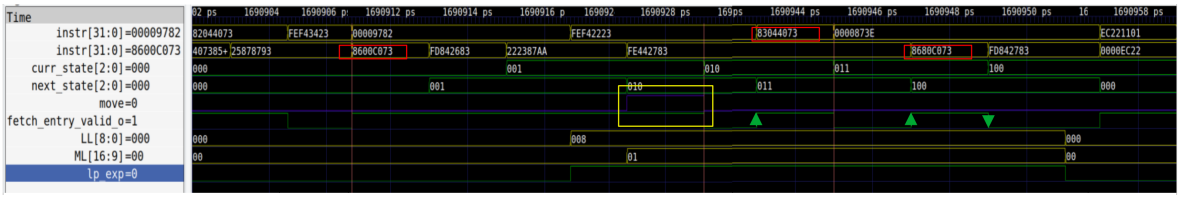


FIGURE 5.3: Test 2: Matching 17-bit label

Subsequently, since the current instruction is identified as an LPCLL instruction, the label comparison is conducted. The label stored in the LPLR CSR, originating from the LPSLL instruction, equals to 2, while the label encoded in the LPCLL instruction equals to 3. This label mismatch triggers the FSM's transition to the ILLEGAL_INSTRUCTION_STATE (101). The instruction emerging from the instruction queue is marked as illegal, accompanied by an exception cause of 2.

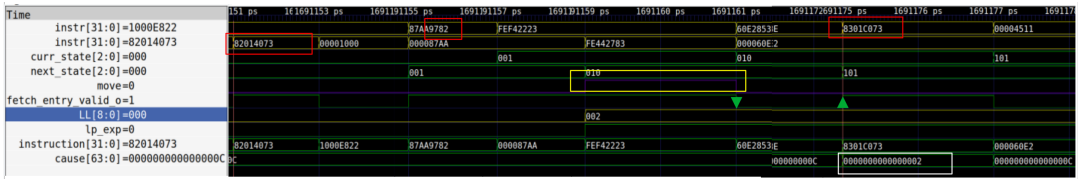


FIGURE 5.4: Test 2: Mismatch of 9-bit label

In the third phase of the study, the focus shifts to examining the core's response when an interrupt occurs during the resolution of an indirect branch. To replicate this scenario, the ecall instruction is utilized to trigger an environmental call exception. This exception is strategically executed just before the indirect jump, effectively interrupting both the jump resolution process and the CFI mechanism.

Following the described CFI FSM sequence, when an indirect jump enters the instruction queue, the FSM transitions to the WAIT STATE (001), the label is set to the LPLR CSR and a landing pad is expected. This phase investigates two distinct scenarios: first, the core's behavior when an interrupt takes place and matching labels are present, and second, the core's behavior when an interrupt occurs and there is a label mismatch.

Interrupt with matching labels

```
add:
 :
 .long 0x82044073 #lpsll 8
 ecall
 jalr a5 #sub
 :
sub:
 .long 0x83044073 #lpcll 8
 :
  jr ra
```

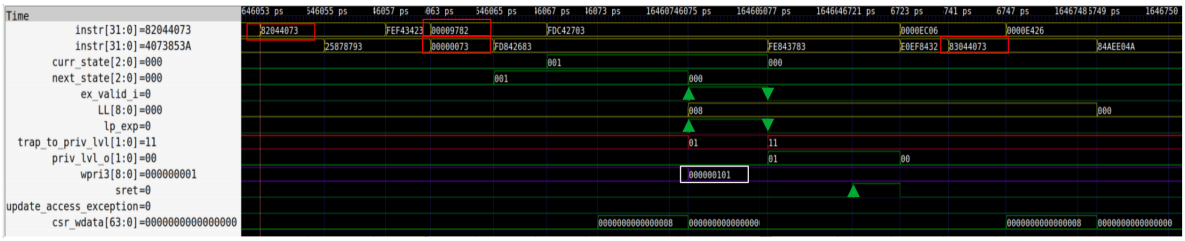
Interrupt with label mismatch

```
foo:
 :
 .long 0x82014073 #lpsll 2
 ecall
 jalr a5 #sub
 :
bar:
 .long 0x83044073 #lpcll 8
 :
  jr ra
```

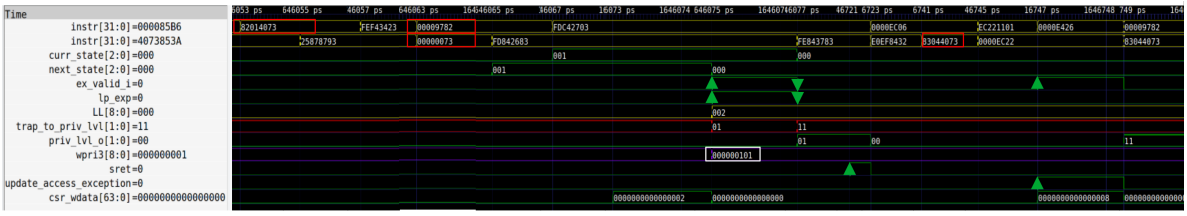
When an interrupt occurs the FSM transitions to NO_LP_EXPECTED_STATE (000). This prompts an immediate halt in the regular execution flow, with control shifting to the trap handler. Prior to the commencement of the trap handler's execution routine, the core's architectural state is carefully preserved in the mstatus csr. This preservation ensures that once the routine concludes, the core's architectural state can be seamlessly restored to its initial configuration.

Upon analyzing the waveforms 5.5, the CSR Register File detects the interrupt or exception, leading to an update in the trap_to_priv_lvl signal. In this scenario, it transitions to "01", indicating a change in the core's privilege level from U-mode to S-mode. The SPELP field of the mstatus csr (wpri3[2]), denoted by the white box, is adjusted to match the expected landing pad, which, in this case, is "1".

Once the trap handler completes its routine, the sret signal is set to "1", enabling the normal program execution flow to resume. In this sequence, where the FSM is in the NO_LP_EXPECTED_STATE (000) , the verification checks occur in the core's backend region. If label matching is confirmed, the label csr is cleared, and no exceptions arise 5.5a. However, if a label mismatch occurs, it triggers an update_access_exception 5.5b.



(A) Matching label



(B) Label mismatch

FIGURE 5.5: Interrupt and CFI process

Middle label mismatch

```
add:
  :
  .long 0x82014073 #lpsll 2
  .long 0x8600C073 #lpsml 1
  jalr a5 #sub
  :
sub:
  .long 0x82014073 #lpcll 2
  .long 0x8682C073 #lpcml 5
  :
   jr ra
```

In the fourth run, the focus of examination centers on the core's behavior in the presence of a mismatch in the middle label. As anticipated, the flow proceeds smoothly until the lower label check. However, when the middle label comparison is conducted, the FSM transitions from the M_LP_EXPECTED_STATE (011) to the ILLEGAL_INSTRUCTION_STATE (101), since the middle label stored in the LPLR CSR is one and the label encoded in the LPCML instruction equals to five.

Similarly, when a mismatch occurs in the upper label, the core's behavior follows a similar pattern. The flow advances until the U_LP_EXPECTED_STATE

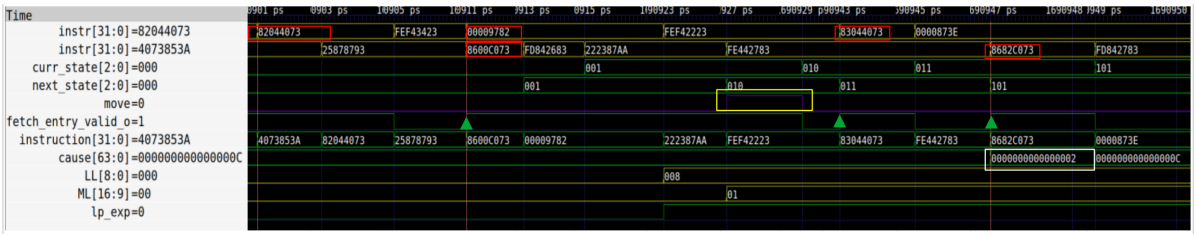


FIGURE 5.6: Test 4: Mismatch of the middle label

(100). Upon label comparison, and in the event of a mismatch, the FSM transitions to the ILLEGAL_INSTRUCTION_STATE (101).

After concluding all the experiments for the case study,it is important to highlight that with the Forward-Edge CFI extension enabled, if there is not a landing pad instruction at the target address or there is a label mismatch, an illegal subversion of the program's control flow occurred.

5.2 Performance

To evaluate performance, the study employed CVA6's proposed microbenchmarks, including Dhrystone, Towers, VVADD, QSort, Multiply, SPMV, RSort, Median, MT-Matmul, and MT-VADD. These benchmarks were compiled using the Rivos Inc. RISC-V GNU Compiler, which includes the Forward-Edge CFI extension. Importantly, each test incorporates landing pad instructions, when needed.Note, that all jumps performed by JALR, C.JALR or C.JR instructions, incorporate landing pad instructions. The execution of these tests took place in a bare-metal environment, utilizing Verilator 4.110 within the Variane Testharness testbench framework.

In Figure 5.7, the overhead of each benchmark is illustrated, showcasing the CPU time used for 9, 17, and 25-bit labels in comparison to the CPU time utilized by the same tests without the inclusion of landing pads. Notably, as the label size increases, the performance overhead experiences a corresponding increment. However, it's crucial to emphasize that for these relatively small programs, a label size of 9 bits suffices. The most precise measurement, therefore, centers on the 9-bit label, indicating an average overhead of 2.1%. The other measurements, utilizing wider labels of 17 and 25 bits, serve as illustrative examples, relevant in scenarios where a program necessitates a broader label than the standard 9 bits.

In another case study, the evaluation involved running a test with varying numbers of indirect jumps. To achieve this, modifications were made to the Median

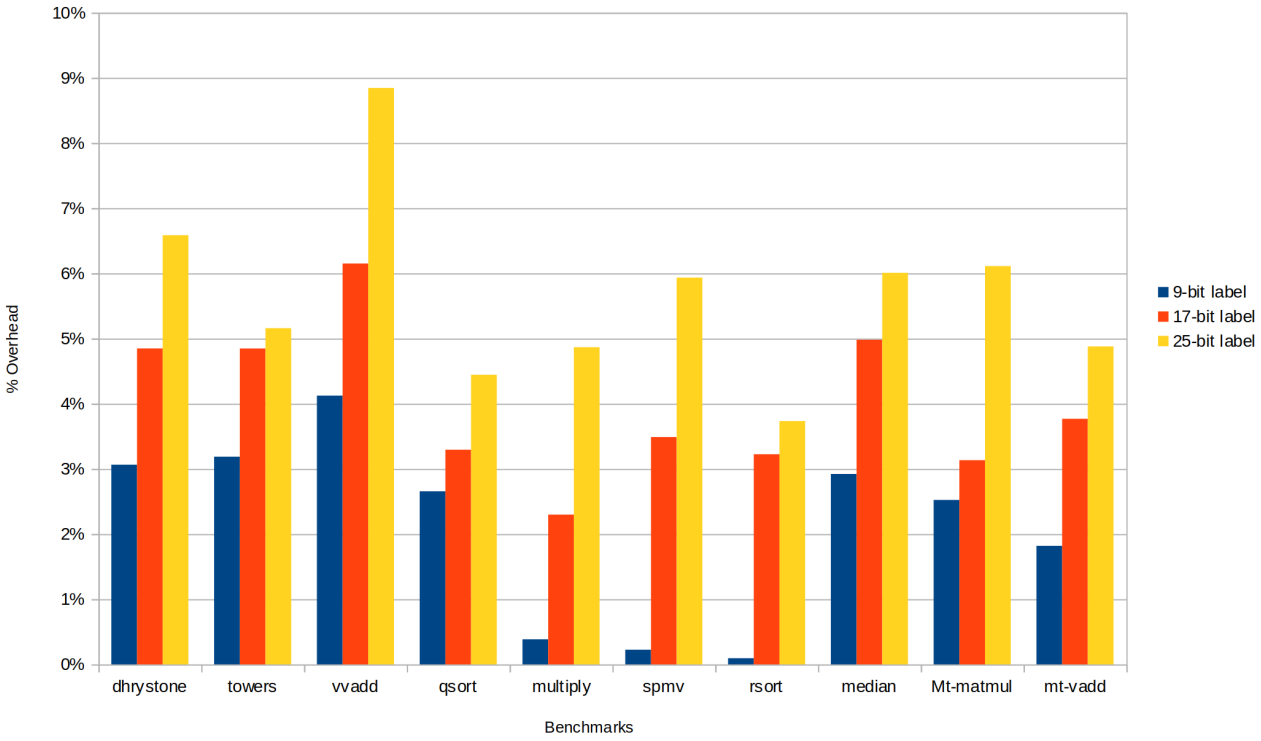


FIGURE 5.7: RISC-V Benchmarks' overhead with Forward-Edge CFI enable for different label width

microbenchmark by introducing a while loop that indirectly invokes an empty function. By adjusting the loop's iteration count, it became possible to obtain results and observe the core's behavior across a range of repetitions, spanning from 0 to 2000 iterations. The outcomes of this study are visually presented in the accompanying diagram 5.8. To attain these results, a comparative analysis was conducted by measuring the CPU time for varying iterations against the CPU time required for a single iteration. As illustrated in the diagram, an observable trend emerges: as the number of iterations increases, there is a corresponding rise in performance overhead. Notably, the results obtained for the 9-bit label exhibit a closer alignment with real-world scenarios, emphasizing their relevance in practical applications.

It's essential to note that using a 9-bit label requires two landing pad instructions per indirect jump, while a 17-bit label demands four, and a 25-bit label necessitates six. So, the overhead that accompanies this extension stems from the fact that, depending on the width of the landing pad, an additional 2, 4, or 6 instructions will be present with every indirect call or jump.

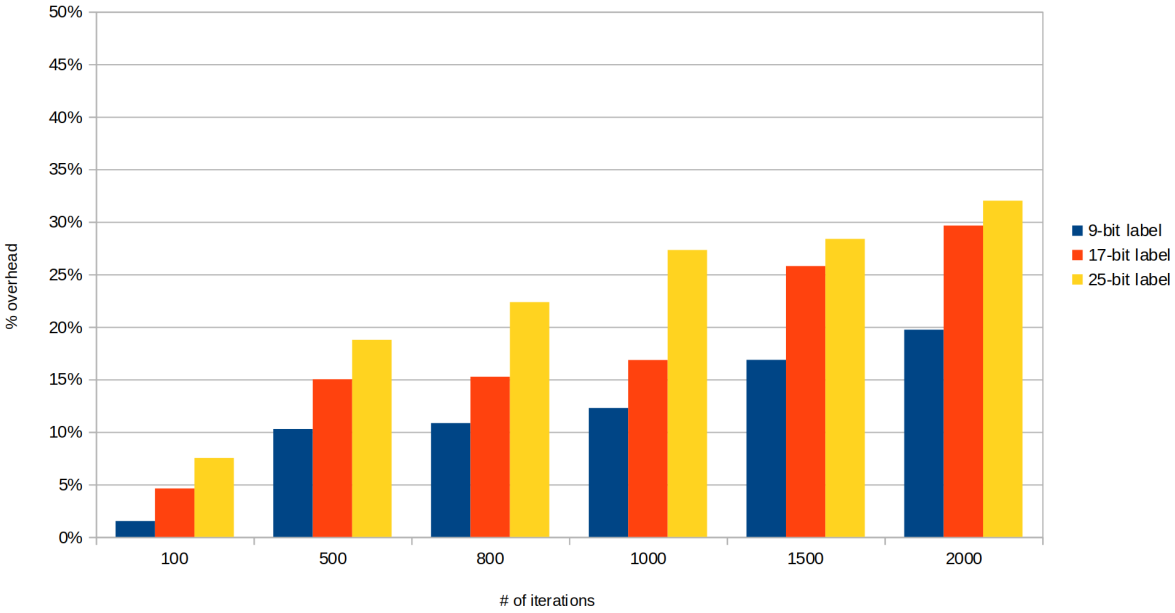


FIGURE 5.8: Median's overhead for different iterations with Forward-Edge CFI enable for different label width

5.2.1 Text area

Another significant metric under consideration pertains to the code size, denoting the volume of code in the benchmarks. To quantify this aspect, the "wc" command was utilized to enumerate the code size across three distinct label sizes: 9, 17, and 25 bits.

As it is already mentioned, a 9-bit label necessitates the inclusion of two landing pad instructions for each indirect jump. In contrast, the utilization of a 17-bit label escalates this requirement to four landing pad instructions, while a 25-bit label mandates the incorporation of six landing pad instructions. Notably, the analysis reveals that when employing a 9-bit label, the text area registers an average increase of 2.71%. With a 17-bit label, this elevation in code size averages 4.33%, and with a 25-bit label, the average expansion amounts to 5.78%.

5.3 Vivado Reports

The next step of the evaluation is to analyze the generated reports from Vivado Hlx Design Suite 2018.3 of the RTL design of the CVA6 core with the Forward-Edge CFI ISA extension, mapped on a Genesys 2 (Kintex-7 FPGA) board. Following synthesis and implementation, the tool provided results for power consumption, area utilization, and timing analysis.

To measure the impact of the Forward-Edge CFI ISA extension, a comparison was made between the CVA6 core with the extension and the CVA6 core without the extension within the integrated SoC. Additionally, measurements were taken for the two instances of the entire SoC. The SoC includes several peripherals such as ethernet, uart, plic etc. as shown in figure 5.9 [14]. The measurements for area utilization are, also provided in comparison to the FPGA's available resources.

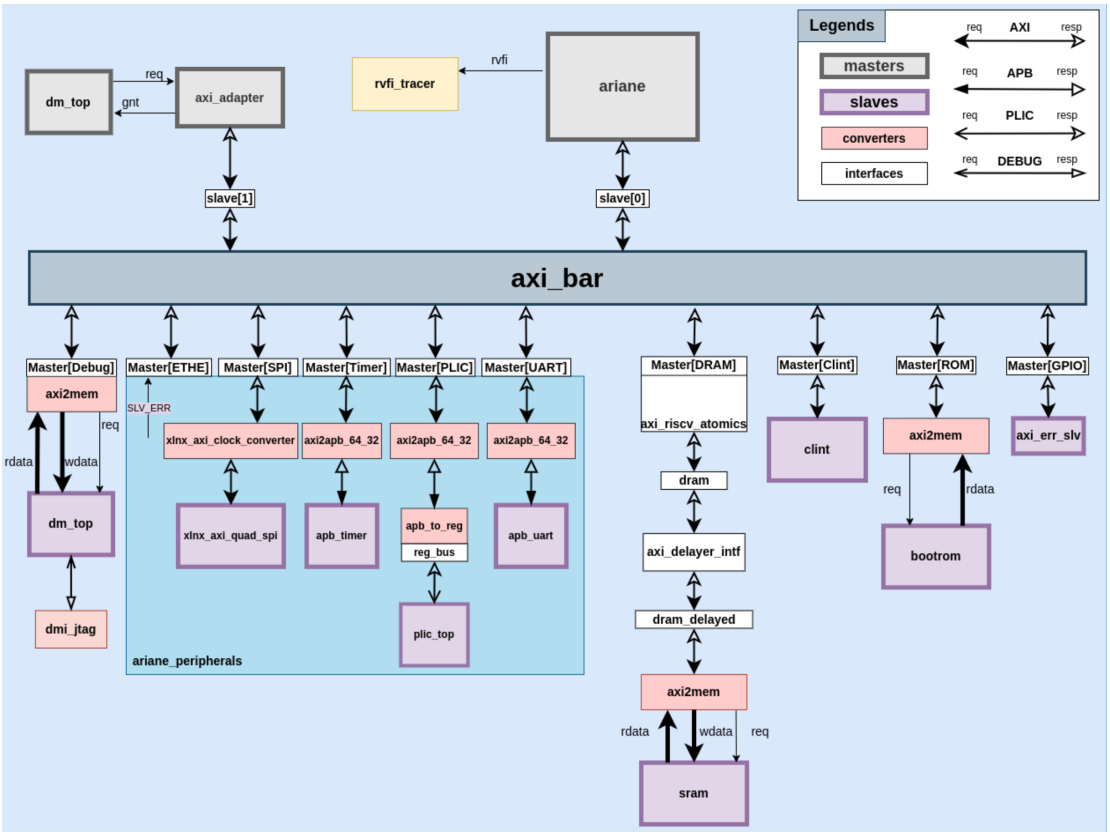


FIGURE 5.9: Ariane Soc

5.3.1 Area Utilization

To gain insight into the area utilization impact resulting from the addition of the CFI ISA extension to the CVA6 core, a comparison was made using the hierarchical utilization report generated by Vivado with the *report_utilization -hierarchical* command. This report contains resource utilization details at various hierarchical levels within the design, offering a breakdown of how resources are allocated across different design modules, submodules, and the overall design hierarchy.

Upon comparing CVA6 with and without the extension, it is observed, that there was a 0.18% increase in the total number of LUTs within the CVA6

core and a 0.10% increase in the Ariane SoC. Additionally, the number of flip-flops (FFs) saw an increase of 2.14% within the CVA6 core and 1.04% in the Ariane SoC. Table 5.1 also,provides a detail area utilisation report for several components. As expected, other resources such as RAM and DSP blocks appear to remain unaffected by the extension.

	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48 Blocks
Ariane SoC	73719	72082	1264	373	48283	49	2	27
CVA6 core	47406	47406	0	0	23563	36	0	27
frontend	3089	3089	0	0	4134	0	0	0
id_stage	235	235	0	0	303	0	0	0
ex_stage	17816	17816	0	0	7683	0	0	0
issue_stage	17218	17218	0	0	6796	0	0	0
csr_regfile	2607	2607	0	0	1706	0	0	0

(A) Area Utilization of CVA6 without the FCFI extension

	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48 Blocks
Ariane SoC	73796	72159	1264	373	48786	49	2	27
CVA6 core	47491	47491	0	0	24068	36	0	27
frontend	3244	3244	0	0	4145	0	0	0
id_stage	439	439	0	0	318	0	0	0
ex_stage	18191	18191	0	0	7689	0	0	0
issue_stage	17413	17413	0	0	6796	0	0	0
csr_regfile	2638	2638	0	0	1870	0	0	0

(B) Area Utilization of CVA6 with the FCFI extension

	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48 Blocks
Ariane SoC	0.10%	0.10%	0.00%	0.00%	1.04%	0.00%	0.00%	0.00%
CVA6 core	0.18%	0.18%	0.00%	0.00%	2.14%	0.00%	0.00%	0.00%
frontend	0.001%	0.001%	0.00%	0.00%	0.01%	0.00%	0.00%	0.00%
id_stage	85.5%	85.5%	0.00%	0.00%	4.7%	0.00%	0.00%	0.00%
ex_stage	2.1%	2.1%	0.00%	0.00%	0.07%	0.00%	0.00%	0.00%
issue_stage	1.1%	1.1%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
csr_regfile	1.06%	1.06%	0.00%	0.00%	9.6%	0.00%	0.00%	0.00%

(C) Area Utilization Impact in percentages

TABLE 5.1: Area Utilization

To further gain insight into the area utilization impact of the design, particularly focusing on the placed design, a comparison was made between CVA6 with and without the ISA extension using the reports generated after the completion of place and route using the command *report_utilization -pb ariane_xilinx_utilization_placed.pb*. These reports rely on a placed design database, specified by the -pb flag, to provide comprehensive details about resource utilization after the design has undergone placement and routing. This level of reporting offers a more precise view of how the design utilizes resources on the Genesys 2 FPGA device, including specifics about placement and routing.

Table 5.2 presents the results as percentages, indicating the utilization of various components (e.g., FFs, LUTs) relative to the available resources on the FPGA device. It is observed that with the extension, the core requires an additional 0.04% of LUTs, 0.13% more registers, 0.92% more F7 muxes, and 0.32% more F8 muxes compared to the available resources.

Site Type	Used (Baseline)	Used (FCFI)	Available	Util% (Baseline)	Util% (FCFI)
Slice LUTs	73719	73796	203800	36.17	36.21
LUT as Logic	72082	72159	203800	36.36	35.40
LUT as Memory (LUTRAMs/SRLs)	1637 (1264/373)	1637 (1264/373)	64000	2.55	2.55
Slice Registers	48283	48786	407600	11.84	11.97
Register as Flip Flop	48269	48764	407600	11.84	11.97
Register as Latch	0	8	407600	0.00	<0.01
Register as AND/OR	14	14	407600	<0.01	<0.01
F7 Muxes	2315	3251	101900	2.27	3.19
F8 Muxes	359	521	50950	0.70	1.02

TABLE 5.2: Slice Logic: FPGA resourses after place and route for CVA6 without and with FCFI

On-Chip	**Power (W)**	
	CVA6 (baseline)	CVA6 with FCFI
Clocks	0.199	0.199
Slice Logic	0.079	0.079
LUT as Logic	0.072	0.072
CARRY4	0.004	0.004
Register	0.002	0.002
LUT as Distributed RAM	<0.001	<0.001
F7/F8 Muxes	<0.001	<0.001
LUT as Shift Register	<0.001	<0.001
Others	<0.001	<0.001
Signals	0.097	0.097
Block RAM	0.039	0.039
MMCM	0.324	0.324
PLL	0.133	0.133
DSPs	<0.001	<0.001
I/O	0.623	0.623
PHASER	0.456	0.456
XADC	0.004	0.004

TABLE 5.3: On Chip components: Power Consumption

5.3.2 Power Consumption

Results for power consumption were obtained through an analysis of power reports generated by Vivado with the *report_power -verbose* command. These reports provide insights into worst-case power estimations. In both cases (CVA6 with and without the extension), the total On-Chip Power remains constant at 2.13 W. Static power remains consistent at 0.17 W, with the estimated dynamic power at 1.95 W. For a more comprehensive perspective on power consumption related to on-chip components, please refer to Table 5.3. Regarding the RTL design, the CVA6 core contributes 0.106 W to the total estimated dynamic power.

5.3.3 Timing Analysis

Concerning the timing analysis, the results of the Vivado reports that were examined, were generated from the command *report_timing_summary -max_paths 10 -file -pb ariane_xilinx_timing_summary_routed.pb -rpx ariane_xilinx _timing_summary_routed.rpx -warn_on_violation*. This command generates a concise summary of the most critical timing paths in the routed design and highlights any timing violations as warnings.

The Ariane SoC, inclusive of the CVA6 core, was configured to operate with a clock period set at 20ns, corresponding to a 50MHz frequency. Under these established timing constraints, the CVA6 core, without the Forward-Edge CFI ISA extension, exhibits a critical path with a positive slack of 2.167ns. However, when the FCFI extension is enabled within the CVA6 core, the critical path shows a slightly reduced positive slack of 1.302ns.

To provide additional context, delays were observed in situations involving interrupts, wherein updates to the Control and Status Registers (CSRs) related to the CFI extension, specifically menvcfg and the label csr, were necessary both before the initiation of the interrupt handler and after the conclusion of the interrupt handler routine. Those delays occur in issue stage and are highlighted with red color in Figure 5.10.

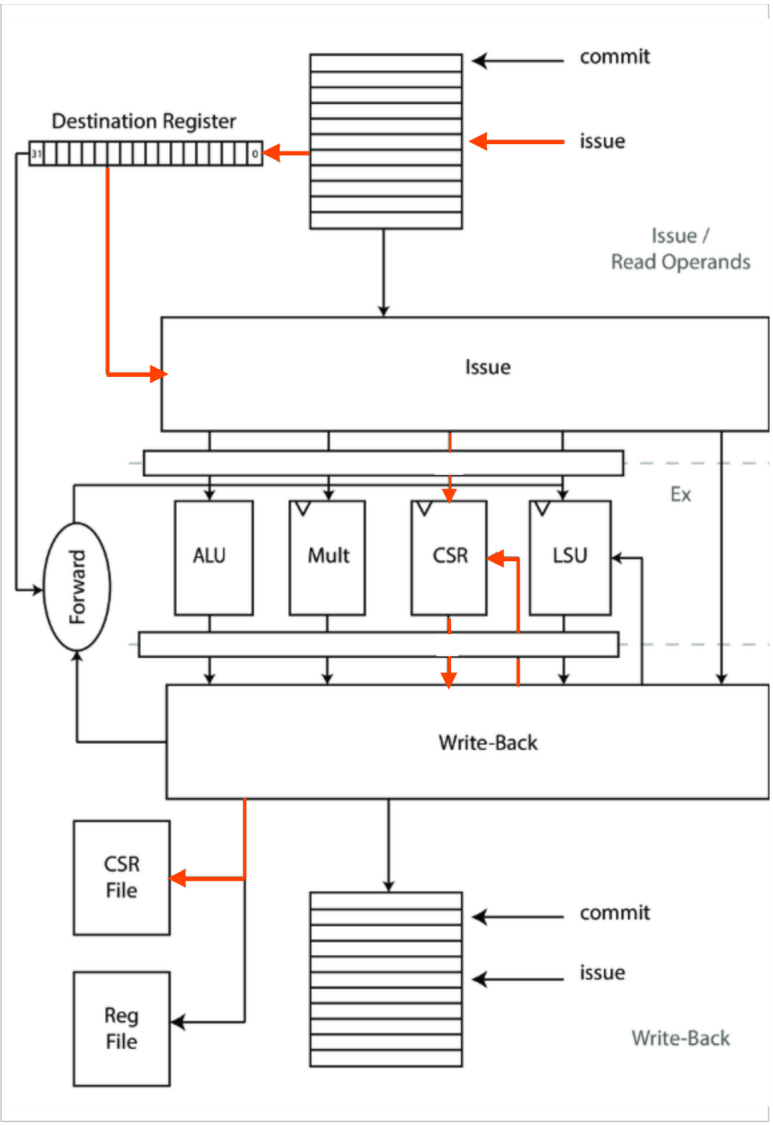


FIGURE 5.10: Critical path in CVA6, with the Forward-Edge CFI extension enabled, focused on Issue Stage

5.4 Performance Evaluation Conclusions

In summary, the performance evaluation of the Forward-Edge CFI ISA extension entailed conducting various experiments to demonstrate its effectiveness. The proposed benchmarks were executed to quantify the performance overhead. Additionally, an analysis was performed on the reports generated from Vivado when mapping the core onto the Genesys 2 FPGA board, with a focus on area utilization, power estimation, and timing.

Regarding effectiveness, the results of the experiments reveal that in the case of an illegal subversion of the control flow of a program (such as the absence of a landing pad instruction or a label mismatch), an exception will be raised,

halting the program's execution. The tagging of all instructions indicating indirect calls or jumps with labels significantly restricts the potential attack surface for a JOP (Jump-Oriented Programming) attack. This outcome aligns with the findings of IBTs (Indirect Branch Tracking) in CET [22] (Control-Flow Enforcement Technology), which is a similar implementation on Intel processors. Moreover, considering the insights from paper [40], it can be confidently stated that forward-only approaches provide a sufficient level of precision but are limited to partial security, as they do not protect the stack.

As for the performance analysis, in Figure 5.7, the depicted performance overhead for benchmarks using 9, 17, and 25-bit labels compared to those without landing pads shows an increase as label size grows. Notably, for relatively small programs, a 9-bit label suffices, resulting in an average overhead of 2.1%. The measurements for 17 and 25-bit labels serve as illustrative examples, especially relevant when broader labels are required. Given the hardware-assisted approach of the Forward-Edge CFI extension, it outperforms software-based CFI implementations, which can incur up to a 14% performance overhead [40]. It is worth noting that a wider label results in increased performance overhead because more instructions are needed to tag the indirect calls and jumps. It's essential to note that using a 9-bit label requires two landing pad instructions per indirect jump, while a 17-bit label demands four, and a 25-bit label necessitates six. The analysis emphasized that employing a 9-bit label led to an average text area increase of 2.71%. With a 17-bit label, this expansion averaged 4.33%, and with a 25-bit label, it reached 5.78%.

In another crucial case study, regarding performance analysis, depicted in Figure 5.8, the core's performance under varying numbers of indirect jumps was examined. Modifications to the Median microbenchmark allowed the analysis of performance across iterations from 0 to 2000. The results unveiled a clear trend: an increase in iterations resulted in a corresponding rise in performance overhead. Significantly, the results for the 9-bit label closely mirrored real-world scenarios, highlighting their practical significance.

Upon reviewing the Vivado reports, it becomes evident that the Forward-Edge CFI ISA extension has a negligible impact on area utilization. Following the placement and routing processes, there is a slight uptick in the utilization of available FPGA resources. When comparing this to the initial design lacking the Forward-Edge CFI ISA extension, we observe the need for additional resources: 0.04% more LUTs, 0.13% more registers (including Flip-Flops, LATCH registers as AND/OR), 0.92% more F7 multiplexers and 0.32% more F8 multiplexers.

Notably, this impact doesn't affect BRAMs (Block RAMs), SRLs (Shift Register LUTs), or DSPs (Digital Signal Processors).

In terms of timing analysis, with the Forward-Edge CFI ISA extension enabled within the CVA6 core, the critical path exhibits a slightly reduced positive slack but still manages to stay within the originally configured timing constraints of 50MHz.

Regarding the worst-case power consumption estimation, no significant variations are apparent. Power levels remain constant at 2.13 W, with static power staying at 0.17 W, and the estimated dynamic power at 1.95 W, when compared to the initial design.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In conclusion, this thesis aimed to implement the Forward-Edge CFI ISA extension, following the guidelines outlined in the RISC-V CFI [23] specification proposal by the RISC-V Foundation. The objective of this ISA extension is to enhance the security of the CVA6 core against jump-oriented programming attacks while minimizing performance overhead. To achieve this, the core's security was bolstered by monitoring all indirect calls and jumps (e.g., jalr, c.jalr, c.jr) using labels. Six new instructions were introduced to set and check these labels during program execution. The "set landing pad" instructions assign labels before indirect calls or jumps, and the "check landing pad" instructions verify label matches. A label mismatch indicates an illegal subversion of the control flow, a possible JOP attack, triggering an illegal instruction exception.

All experiments conducted as proof-of-concept demonstrated the extension's successful operation, with performance overhead averaging at 2.10% for a 9-bit label, 4% for a 17-bit label, and 5.66% for a 25-bit label. Corresponding code size increases were 2.71%, 4.33%, and 5.78%, respectively, based on label width.

Regarding the design, a comparison between the CVA6 core with and without the Forward-Edge CFI ISA extension revealed minimal area utilization overhead, with an increase of 2.14% within the CVA6 core and 1.04% in the Ariane SoC. Similarly, the worst-case power consumption estimation remained unchanged. Additionally, a delay was observed in the critical path of the CVA6 core, without violating any timing constraints.

This thesis serves as a demonstration of the Forward-Edge CFI ISA extension within the RISC-V architecture, with the ultimate goal of seeking ratification

from the RISC-V organization as a standard ISA extension in the RISC-V architecture.

6.2 Future Work

6.2.1 Forward-Edge CFI ISA Extension

In response to recommendations from the RISC-V organization, the landing pad instructions were categorized as "maybe operations" (Zimops). This classification implies that in systems lacking support for the CFI extension, these instructions will function as no-operations (nops). Conversely, when the CFI extension is available, they will operate as outlined in the thesis. A debate arose regarding whether to label these instructions as Zimops or HINTs [41]. HINTs represent a class of instructions that should not modify the core's architectural state, which does not align with this extension's behavior. Initially defined as Zimops, all six landing pad instructions were required for full implementation of the Forward Edge CFI logic. However, the latest update leans towards categorizing the landing pad instructions as HINTs, streamlining implementations to employ two instruction (set and check) with 20-bit labels. In this context, the performance overhead corresponds to the additional overhead and code size associated with the 9-bit label (averaging 2.1% and 2.14%, respectively), while concurrently reducing FPGA resource demands.

6.2.2 Backward-Edge CFI ISA Extension

Without the implementation of Backward-Edge Control Flow Integrity (CFI) mechanisms, a system cannot achieve complete protection from code reuse attacks, even if it already employs Forward-Edge protection. To enhance security further, future work needs to be done to incorporate the Backward-Edge CFI ISA extension as outlined in the specification [23]. The proposed solution involves the use of a shadow stack.

In its current state with only forward edge protection, the system can mitigate certain code reuse attacks that rely on altering the control flow at the point of calling functions (forward edges). However, it remains vulnerable to attacks that target the return addresses or function pointers (backward edges) where malicious code can hijack the execution flow.

To address this limitation, the future work aims to implement the Backward-Edge CFI ISA extension, which involves the utilization of a shadow stack. The

shadow stack acts as a separate stack from the regular stack used for storing return addresses. Each function call will have its return address stored both in the regular stack and the shadow stack. During the function's return, the system verifies that the return address matches the value on the shadow stack, ensuring that the execution flow remains intact and untampered.

By combining forward edge protection with the proposed Backward Edge CFI mechanism using the shadow stack, the system will be able to more comprehensively safeguard against code reuse attacks, providing a stronger defense against potential security threats.

References

[1] Vince W. Freeh Tyler Bletsch Xuxian Jiang and Zhenkai Liang. "Jump-Oriented Programming: A New Class of Code-Reuse Attack". In: (2011). DOI: 10.1145/1966913.1966919.

[2] C. E Nwokeji. *An Overview of Buffer Overflow and Network Intrusion Detection and Prevention Systems*. Vol. 2. 2019, pp. 39–46.

[3] T. Rains, M. Miller, and D. Weston. *Exploitation trends: From potential risk to actual risk. In: RSA Conference (2015).*

[4] Giorgos Vasiliadis George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. "Hard edges: Hardware-based Control-Flow Integrity for Embedded Devices". In: (2022). DOI: 10.1007/978-3-031-04580-6_18.

[15] Naimah Yaakob TMukrimah Nawir Amiza Amir and Ong Bi Lynn. "Internet of Things (IoT): Taxonomy of Security Attacks". In: (2016). DOI: 10.1109/ICED.2016.7804660.

[16] Byoungyoung Le eand Simon P. Chung Kangjie Lu Chengyu Song, Taesoo Kim, and Wenke Lee. "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks". In: (2015). DOI: 10.1145/2810103.2813694.

[17] Marco Prandini and Marco Ramilli. "Return-Oriented Programming". In: (2012). DOI: 10.1109/MSP.2012.152.

[19] Ulfar Erlingsson Mart´ın Abadi Mihai Budiu and Jay Ligatti. "Control-Flow Integrity". In: (2009). DOI: 10.1145/1609956.1609960.

[20] Ismael Ripoll Sarwar Sayeed Hector Marco-Gisbert 1 and Miriam Birch. "Control-Flow Integrity: Attacks and Protections". In: (2019). DOI: 10.3390/app9204229.

[21] Nael Abu-Ghazaleh Mehmet Kayaalp Meltem Ozsoy and Dmitry Ponomarev. "Branch regulation: Low-overhead protection from code reuse attacks". In: (2012). DOI: 10.1109/ISCA.2012.6237009.

[22] Deepak Gupta Vedvyas Shanbhogue and Ravi Sahita. "Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity". In: (2019). DOI: 10.1145/3337167.3337175.

[30] S. Adrian H. Hu S. Shinde, P. Saxena Z. L. Chua, and Z. Liang. "Data-Oriented Programming:On the Expressiveness of Non-Control Data Attacks". In: (2016). DOI: 10.1109/SP.2016.62.

[40] Joseph Nash Nathan Burown Scott A. Carr et al. "Control-Flow Integrity: Precision, Security, and Performance". In: (2018). DOI: 10.1145/3054924.

External Links

[5] OpenHW Group. *CVA6 RISC-V CPU*. URL: https://github.com/openhwgroup/cva6.

[6] *RISC-V*. Apr. 2023. URL: https://en.wikipedia.org/wiki/RISC-V.

[7] *OPV Smulator*. Sept. 2023. URL: https://www.ovpworld.org/info_riscv.

[8] *Spike Smulator*. Sept. 2023. URL: https://chipyard.readthedocs.io/en/latest/Software/Spike.html.

[9] *QEMU Emulator*. Sept. 2023. URL: https://www.qemu.org/docs/master/system/target-riscv.html.

[10] *E31*. Apr. 2023. URL: https://www.sifive.com/cores/e31.

[11] *CORE-V-MCU*. Apr. 2023. URL: https://docs.openhwgroup.org/projects/core-v-mcu/doc-src/overview.html.

[12] *BK3*. Apr. 2023. URL: https://codasip.com/2018/02/28/trinamic-licenses-codasips-bk3-risc-v-processor-for-next-generation-motion-control-applications/.

[13] *RISC-LIST*. Apr. 2023. URL: https://github.com/riscvarchive/riscv-cores-list/blob/master/README.md.

[14] OpenHW Group. *CVA6 User Manual*. URL: https://cva6.readthedocs.io/en/latest/.

[18] *Branch Target Instructions (BTIs)*. URL: https://developer.arm.com/documentation/102433/0100/Jump-oriented-programming.

[23] *RISC-V CFI specification*. Jan. 2023. URL: https://github.com/riscv/riscv-cfi.

[24] *Privilege Levels*. URL: https://five-embeddev.com/riscv-isa-manual/latest/priv-intro.html#privilege-levels.

[25] *XLEN*. URL: https://five-embeddev.com/riscv-isa-manual/latest/machine.html#xlen-control.

[26] *WARL*. URL: https://five-embeddev.com/riscv-isa-manual/latest/priv-csrs.html.

[27] *Vtables*. URL: https://en.wikipedia.org/wiki/Virtual_method_table.

[28] *RELRO*. URL: https://ctf101.org/binary-exploitation/relocation-read-only/.

[29] *GOT*. URL: https://ctf101.org/binary-exploitation/what-is-the-got/#got.

[31] *PLT*. URL: https://ctf101.org/binary-exploitation/what-is-the-got/#plt.

[32] Five EmbedDev. *Five EmbedDev RISC-V Manual*. URL: https://five-embeddev.com/riscv-isa-manual/latest.

[33] *Vivado Design Suite Documentation*. URL: https://docs.xilinx.com/r/2021.1-English/ug896-vivado-ip/Vivado-Design-Suite-Documentation.

[34] *VERILATOR*. URL: https://verilator.org/guide/latest/.

[35] *RISC-V Proxy Kernel and Boot Loader*. URL: https://github.com/riscv-software-src/riscv-pk.

[36] *GTKWave*. URL: https://gtkwave.sourceforge.net/.

[37] *RISC-V Tests*. URL: https://github.com/riscv-software-src/riscv-tests/tree/master.

[38] *Rivos Inc.* URL: https://www.rivosinc.com.

[39] *Field Programmable Gate Array (FPGA)*. URL: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[41] *HINTs*. URL: https://five-embeddev.com/riscv-isa-manual/latest/rv32.html#sec:rv32i-hints.