RISCV Hardware Security Improvements for Embedded Software

University of the West of England
BEng(Hons) Computing for Embedded Systems

Final Year Project 2020 Report

Andrew Belcher
StudentID:17010347

April 23, 2020

NOTICE

The full project files and code discussed in this report can be found at..

https://github.com/AndrewGBelcher/FYP_2020

Acknowledgments

This project would not have been possible if it wasn't for my Wife, who was a constant source of support throughout my academic life, especially during this projects development.

I would also like to thank my project's supervisor Martin Serpell, who was a continual force in its development helping keeping it on track through his feedback, direction, and consistant review of the various iterations this project had gone through.

I also cannot forget to thank Craig Duffy, especially for the suggestions and discussions had regarding this project which proved helpful in informing me of similar projects to gain inspiration and what aspects to keep in mind.

Contents

List of Figures

Chapter 1

Abstract

This report conducts a review of existing control-flow integrity enforcement schemes as well as discusses the development of a CFI enforcement design for the RISCV open-source architecture. Various changes are outlined to produce the support needed in the form of a full software stack comprised of a compiler, instruction set simulator, RTL core design, and demo application. The efforts gathered in this report coalesce into a complete project demonstrating control flow integrity through a custom RISCV extension suitable for all its architectural forms.

Chapter 2

Introduction

In modern computer security, one of the most targeted vectors for an attacker to gain control of a system is through memory access. There has been a cat and mouse game between system designers and attackers over this area and we have seen much exploitation and mitigation techniques employed. Methods such as ASLR or (address space layout randomization), a technique used to randomize memory addressing to prevent attackers from knowing where things are. Another method DEP or data execution prevention marks pages in memory as non-writeable in the effort to make code injection more difficult on a system. Many other methods are used; however, the overall goal is to protect control flow integrity or CFI.

Nevertheless, with an increase of security, comes new inventive ways around it. Modern exploitation techniques use what is known as code reuse attacks like ROP (return-oriented programming) or JOP(jump oriented programming) in the effort to chain together pieces of

existing code to derive the desired execution flow on a system. Both previously mentioned code reuse methods target two different data structures in memory, the stack, and the heap. ASLR attempts to make things for these types of attacks more difficult by changing the position of data within memory to be at random locations set on the initial allocation of that memory space. Additional mitigations within the stack are also employed such as stack canaries, a token at the stacks base which prevents stack modification from a buffer overflow.

2.1 Research and Questions

Based on the research, the majority of current software protection schemes developed for the Control Flow Attacks are for 64bit platforms. The reason for this is based on the partial representation of the full address space with 64bit numbers leaving room to store pointer authentication information in the upper bits of a pointers data. From what was gathered it seems there is barely any support for 32bit architectures in protecting against CFA geared towards return address modification (ROP).

Several questions were posed during research such as "How does pointer authentication work on 64bit architectures?" and "Is it possible to perform authentication the same way on other architectures like 32bit?". Based on the research gathered on pointer authentication codes otherwise known as PAC implementations, it is seen that the solution won't likely solve JOP at-

tacks due to the unpredictability of pointer usage. But will focus on preventing ROP based attacks or similar attacks that modify the return address within a stack frame. To deliver a 32bit solution the research points to additional memory structures acting as internal storage akin to a shadow stack design.

2.2 Aims and objectives

The project intends to deliver an extension to the RISCV 32bit instruction set that provides the ability to authenticate return addresses stored on the stack. The deliverables will be broken down into three phases, a modified simulator, a modified core design, and a software suite made up of a modified toolchain and demo program to showcase the extension's functionality. The project will aim to thwart return address modification on the stack by what would potentially be a control flow attack in the form of ROP or buffer overflow and stack modification.

2.3 Outline

To contextualise the reports information, the following structure will be applied:

- **Research** – This chapter will give an overview of the research conducted in the literature review, and reveal how the project's requirements, design, and implementation will be informed. Justification will be established on why certain choices will be made

throughout the project's development.

- **Requirements** – In this chapter, the findings of the research will initially inform a set of requirements. However, throughout the project's development, they will have been expanded into a detailed set of both functional and non-functional requirements.
- **Methodology** – This chapter will outline the chosen development methodology used in the project, and describe how the project's requirements, design, and implementation will be carried out to deliver the end product.
- **Design** – This chapter will display the various designs for each of the project's components with comprehensive breakdowns on how and why they were developed in such a manner.
- **Implementation** – Details regarding issues encountered, paths taken during development, and overall testing of the project will be outlined in this chapter through screenshots, code snippets, and descriptions.
- **Evaluation &Conclusion** – Reflections on the overall project will be discussed in this chapter, explaining what its findings were, as well as an evaluation of what the project did and didn't accomplish and where it can go beyond its current implementation.

Chapter 3

Research

3.1 Introduction

Research for this project started with the premise of investigating control flow integrity solutions on embedded processors with a focus on security rather than resiliency. The initial research highlights several CFI designs on both ARM and RISCV architectures. The latter is used more often for verification due to it being open source. The majority of the papers used for informing the research section are based on 64bit modifying architectures and maintain CFI via pointer authentication codes. Based on the information gathered, it was determined that this project could fill a gap in providing a CFI solution to a 32bit architecture without the use of pointer authentication codes (PAC instructions).

3.2 Control Flow Attacks

A particularly important area of modern computer security is in dealing with what is known as control flow attacks or CFAs. These are attacks that gain control of the execution flow of a system through breaking control-flow integrity (CFI). This is usually the goal of an attacker, even though they might be only interested in leaking data, hijacking control of a system's execution proves to be one of the best ways of doing so. For this project, research will look into how we might maintain CFI on an embedded system.

3.2.1 Attacks

To first understand how CFI can be maintained it is important to establish how it can be exploited. A nice overview presentation regarding control flow hijacking from Stanford University computer science course details various attacks such as buffer overflows, integer overflows, and format string vulnerabilities **[20](Stanford University, 2020)**. All of which either target two data structures that make up a program's memory space, the stack, and the heap.

3.2.2 Stack Based

The stack is a location within the memory of a program that acts as a LIFO or last in first out data structure. This data structure consists of what are known as stack frames, which are data sets linked to particular function calls, with each frame being pushed and popped off the stack according to the LIFO model. In each frame exists function argument data in the form of literal data, and/or pointers to argument data. A particularly targeted area of the stack frame, however, is the return address. The return address is a pointer that will direct the CPUs program counter to its next location upon exiting the function and restoring the stack frame data to the CPUs internal registers. Generally, most attack methods rely on unchecked bounds of memory manipulation operations that allow write ability to the stack or heap of a program's memory **[20](Stanford University, 2020)**.

3.2.3 Heap Based

Regarding the heap, it is a much more chaotic data structure than the stack. The heap of a program is generally larger and visible to any function. The heap is a much more complex data structure than the stack in that it can be dynamically allocated and deallocated unlike the LIFO structure of the stack **[20](Stanford University, 2020)**. This makes it difficult to manage, especially regarding CFI enforcement. Heap-based exploits such as heap overflow make use of what is known as heap spraying. A technique used to increase the chance

of overwriting a functions pointer stored in what is known as a vtable or virtual table. These tables are used within C++ programs to store an object's functions pointers to their various methods.

3.2.4 Memory Protection

In modern systems such as x86 or ARM architectures, program code is protected in execute-only memory. This restriction is very important to CFI and protects a program's machine code from being altered during runtime. An article that helped inform how embedded devices tackle this problem in contrast to PC architectures was found in an article on security from USENIX **[21](Wetzels, 2017)**. It notes how architectures built according to the Harvard architecture (AVR, PIC, RISCV) take care of this problem as a side effect of their design with separate program ROMs instead of having unified memory like within von Neumann architectures (ARM, X86, MIPS). Which makes it clear that a good handful of embedded architectures will be able to protect against code injection through a side effect of their Harvard architectural design rather than using hardware-specific features to enforce machine code write protection within a program's memory space.

3.2.5 Code Reuse Attacks

However, protecting the memory mappings containing a program's machine code is not an end to control flow hijacking. There exist various techniques to circumvent this through what is known as code reuse attacks (CRAs). A significant change from Code Injection attacks which can directly rewrite a program during runtime or direct execution to injected code in data mappings. CRAs can break CFI through directing execution to specific places in existing code to produce the desired computations. A great article from MDPI has a comprehensive description of CFI exploitation techniques that informed this research **[17](Sayeed, 2019)**. In that article, two important techniques are identified as Return Oriented Programming (ROP) and Jump oriented Programming (JOP).

3.2.6 Return-Oriented Programming

Starting with ROP, this technique is based upon manipulating the return address within a stack frame. The return address can be overwritten with an address that points to a tail of a functions assembly so that only a few instructions will be executed before the program counter it's the return instruction and then moves execution to the next pointer written to the controlled return address **[3](Buchanan, 2008)**. These endpoints are known as gadgets and can be chained together to effectively write something similar to what the attacker would inject if they could. ROP only works however if the programs machine code has enough unique as-

sembly before its return instructions, that can produce near Turing complete functionality.

3.2.7 Jump-Oriented Programming

When looking at JOP, things become even more complex, due to the mitigations against the ROP technique, JOP seeks to remove the reliance on both the stack and existing return opcodes. Instead, JOP uses a single gadget as a dispatcher which then will use indirect jumps instead of returns to direct the program counter to various other gadgets via a jump table loaded into the heap **[2](Bletsch, 2011)**. The benefit of JOP is that stack protections are avoided as it uses corruption of the heap to gain execution and pivot execution with a table located in the heap.

3.3 Mitigations

3.3.1 Shadow Stacks

One solution to maintaining CFI through authenticating stack contents is known as a shadow stack. In a research paper from Purdue University (Burow, 2019) various implementations of shadow stacks are reviewed and analysed including an extension to the X86 Instruction Set Architecture developed by Intel, and known as CET or Control Enforcement Technology. This extension uses both extended instructions to the hardware as well as changes to both compilers GCC and Clang. The majority of implementations focus on either signing the return address as it is stored onto the stack and

authenticating it as it is popped off, or simply XORing it with an internally held copy to aid performance. These shadow stack solutions center around creating control flow resiliency as well as protecting against ROP attacks. However Intel's CET does offer protection against JOP through indirect branch tracking, but this functionality is not part of the shadow stack design and is a separate implementation.

3.3.2 Address Space Layout Randomisation

Another mitigation technique that works against ROP/JOP is what is known as ASLR, or address space layout randomization. This technique works alongside virtual memory management to produce a randomised layout of the program's memory space. This randomization occurs for different mappings such as a program's data section or its code section and will be established on mapping the program upon launch via the virtual memory manager or memory management unit **[7](GRSecurity, 2020)**. The mitigation technique makes it more difficult for an attacker to execute their attack since things like gadgets will be harder to locate due to the change of the program's layout every time it is run. However, with a bit of work, the attacker can still overcome this barrier through brute-forcing the ASLRs slide value (the difference between its true location and layout location).

3.3.3 Pointer Authentication

As seen when researching shadow stacks for CFA mitigation, the use of a signature on a pointer such as the return address can be employed. A very good paper regarding how pointer authentication can be achieved is by **[9](Liljestrand, 2018)**, it illustrates and details how pointer authentication can be achieved far better than an alternate paper **[19](Schilling,2018)**. In this paper, it is explained how the use of MACs or message authentication codes, which are cryptographically generated values, can be used to authenticate pointers. These MACs are named appropriately as PAC or pointer authentication codes and can be generated with custom PAC instructions as extensions to an ISA. The use of PAC instructions in this context help in the mitigation of CFAs like ROP/JOP or similar attacks through authenticating the upper unused bits of a 64bit pointer to produce a verification process on the lower used bits of the pointer. Since 64bit systems don't use the full address space representable by 64bit numbers, information can be stored in a pointer value, maintaining the link between the PAC and the pointer without an internal data structure like a shadow stack. Due to this flexibility in storing information, pointers on the heap can be easily authenticated as well, producing powerful mitigation against both ROP/JOP attacks.

3.3.4 Implementations

With research performed over CFI and CFA mitigations, several problem areas can be identified such as additional control schemes enforced on stack and heap contents, and adjustment to instruction emission patterns to add a layer against code reuse attacks. However, with just the knowledge of these mitigations and the theory of their behaviour, this project still needs inspiration towards how various protection schemes can be implemented to derive a custom scheme.

3.3.5 ARMv8.3 PAC

A standout design with minimal overhead was spotted during research to be the use of PAC instructions. These instructions as outlined in a good overview whitepaper by Qualcomm **[15](Qualcomm Technologies INC, 2017)** describe their usage on an assembly level as additionally added instructions to the ARMv8.3 specification. The whitepaper lacks hardware implementation details due to the designs being developed by ARM which would require purchasing the license for their current RTL designs. However, the details they cover can inform us of its implementation sufficiently enough. Essentially, the usage of 2 important instructions in the prologue and epilogue of a function can replace the need for stack canaries with much less overhead. Stack canaries are tokens that are checked upon entering the functions epilogue, the token is validated to ensure that a buffer adjacent to the stack return address has not been overflown thus modifying the return address. In-

stead on ARMv8.3 PAC instructions can validate the return address in near enough one cycle instead of multiple instructions being executed to produce an equivalent of validation. The PAC extension also includes an instruction for verifying branch pointer values, which would mitigate against JOP. The basis of these instructions rely on 64bit pointers that can be only used on a 64bit architecture, a 32bit architecture would require a drastically different design to enforce pointer authentication it seems.

3.3.6 HardScope

During the research for this project, an interesting implementation outlined in a paper from researchers at Aalto University was sighted **[11](Nyman, 2017)**. In this paper, attacks known as Data-Oriented Programming or DOP can be thwarted by using custom instructions via an extension to the RISCV instruction set architecture. DOP attacks are based on the manipulation of data only rather than code, ROP and JOP can be classed as DOP techniques since they rely on manipulating data to break CFI, however, DOP techniques offer a broader scope in that a DOP attack can be conducted without breaking CFI through simply manipulating function variables to produce the desired execution by the attacker. To solve this problem, the paper presents an implementation known as HardScope or "Hardware-Assisted Run-Time Scope Enforcement". This implementation demonstrates how new custom instructions that maintain runtime scope on the vari-

ables of a function can be supported through changes to the GCC backend as well as to a RISCV instruction set simulator other-wise known as Spike.

3.3.7 RISCV ISA Extension

It was seen that quite a few research projects/papers have been using the RISCV architecture for prototyping various CFI protection schemes. One paper, from the Technical University of Munich, as a bachelor thesis by Leander Seidlitz, demonstrates developing a custom extension to the RISCV 64bit architecture to support PAC instructions like used on ARMv8.3 **[18](Seidlitz, 2019)**. This paper is incredibly useful for showcasing how the GCC backend might be modified to support such instruction emission in forming a function's prologue and epilogue. The detail in this paper regarding the design of the opcodes themselves is helpful in contrast to how the Hard Scope paper detailed them. The paper lacked other sufficient details regarding RTL implementations in an HDL language.

3.3.8 Project Solution Scope

Based on the research carried out for this project ranging from CFI attacks, mitigations, and various implementations we can determine an area that this project will deliver a solution in. The vast majority of CFI mitigation implementations have been performed upon 64bit architectures most notably RISCV and ARMv8. RISCV proves to be a good candidate for the project due to it being open source and the designs available in many HDL languages. An interesting area for the project's design to be based on would be that of a 32bit architecture. This eliminates traditional PAC methods, but something like a shadow stack unit might work well.

3.4 RISCV

3.4.1 Archetecture

RISCV is a completely open-source instruction set architecture originally developed to support research and education, however, it is now a standard free and open architecture for industry implementations. The architecture comes in 4 variants, 32bit base integer set RV32I, RV32E, RV64I, and RV128I. Specifically, the 32bit variants feature 2 kinds, a normal base integer set that has 32 integer registers, as well as an embedded variant that reduces the number of integer registers to that of 16 to strive to be super small base core for embedded microcontrollers.

The nice part of RISCV's design is that the base instruction set can be easily extended to support custom instructions through what are known as extensions. These extensions are split into two broad categories, "standard" and "non-standard". When compiling for RISCV the machine arch prefix can be extended with various standards using the represented letters for those extensions. This will enable the compiler to produce the associated instructions. The convenient factor about using extensions is that if a RISCV core doesn't support the instructions, they can be simply turned off by omitting the extension letter. This maintains compatibility throughout RISCV development and support. For this project, we will be creating a non-standard extension, which can only be set by using the letter 'X' in

the arch name followed by a name and a version number. Multiple non-standard extensions can be chained together even, by using the '_' character to separate them i.e. "`RV64GXargle_Xbargle`" **[16](Riscv.org, 2020)**.

3.4.2 Software Tools

When dealing with developing an extension to the RISCV instruction set, various software tools are necessary so the changes can be tested. Aside from the compiler and the programs you would run on the hardware, an important tool to have is an instruction set simulator. An instruction set simulator is a piece of software that effectively emulates the instructions for a computer architecture. Currently, there are quite a few options regarding ISS, like Gem5 and even QEMU. For this project, however, the most straightforward simulator to use is the official RISCV ISS known as Spike.

3.4.3 Hardware Implementations

Going into the project research there was still enough information about what hardware was needed. A determining factor in this decision was that of selecting an appropriate RTL design of a RISC-V 32bit processor. There are quite a few cores out there ranging from large developers to weekend projects from capable hobbyists. Yet to narrow the search a few requirements helped make a clear decision, one requirement is that the core would need to be written in Verilog. Verilog, when compared to SystemVerilog and especially VHDL, tends to be easier to write and was a personal preference. Another requirement in the location of a design core was that it needed reliable I/O already implemented. Ideally, this would be that of a serial signal so that the code running on the design can be interfaced with without needing multiple inputs through switches or buttons.
A handful of designs were gathered according to these requirements but all but one were eliminated due to ease of use, quality of I/O, 32bit architecture, and written in Verilog. Ironically the chosen core DarkRiscV had been initially written in a night, surprisingly its functionality and practicality in implementing on an FPGA proved to be the best. For the actual Hardware itself for the project, an ARTY S7-50 board was acquired due to its support on windows 10 through the Vivado IDE. A Spartan 3E was initially considered but would require the use of a windows 7 virtual machine to use the compatible legacy IDE from Xilinx ISE.

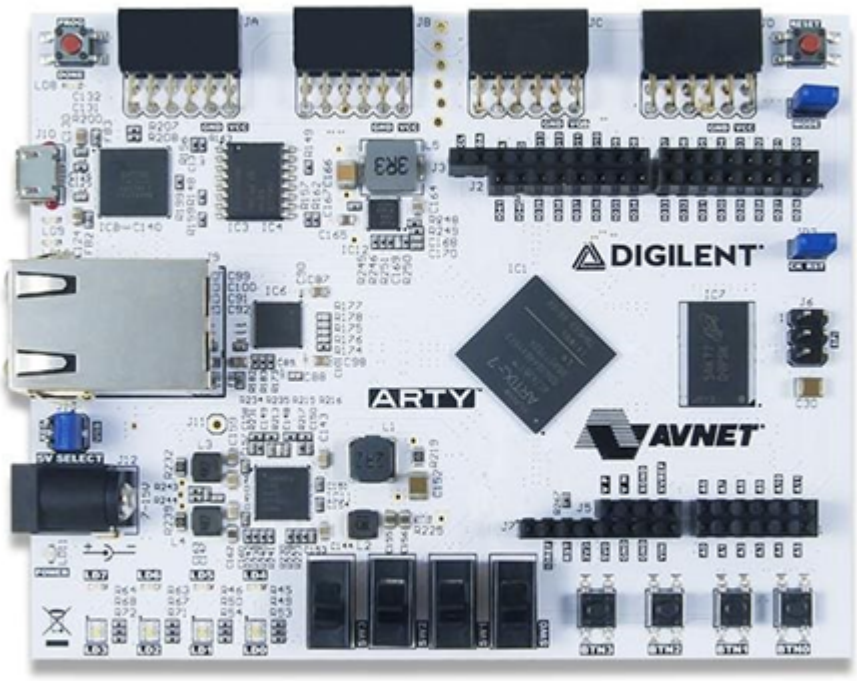


Figure 3.1: Research: Arty-S7-50 FPGA by DILIGENT featuring a Spartan 7 XC7S50-CSGA324 from Xilinx. The chosen hardware for the integration and modification of the DarkRiscv core.

Chapter 4

Requirements

A very important key factor in the success of any project is the development of a set of requirements. Requirements act as short descriptions about what a project will create, and how its products will perform. They can be high or low-level descriptions of how the system will behave and how it will work.

In refining the list of the final requirements, considerations towards attributes such as traceability, testability, and consistency were given. Some of the requirements will represent different functionality while only being possible through the completion of other requirements, they will feature a set of pre-requisites as one of their attributes to make note of this. The rest of this chapter will outline how the requirements will be structured, separated into non-functional/functional requirements, and demonstrate a use case diagram and table. For the full requirements list of project, it can be found in the appendix section A.1.

4.1 Requirments Structure

The completed requirements specification will feature a set of attributes for each requirement:

- **Unique ID** – This ID will be set for a requirement depending on which component it describes. For this project, there are 3 components or sub-projects. HW refers to the hardware, such as the core design and its implementation on an FPGA. ISS references the changes made to Spike, the instruction set simulator used in this project. And SW represents the software elements of the project, such as the demo program for both HW/ISS, as well as the changes to the compiler.
- **Description** – A simple description of the requirement, what it will do, or how the associated component will need to perform.
- **Pre-requisites** – Requirements referenced by their unique ID that are essential to have completed to complete the requirement the pre-requisites refer to.
- **Requirement Type** – The requirement type attribute will reference whether the requirement is of a non-functional or functional type.
- **Priority** – This attribute will dictate what level of importance a requirement has to the overall success of the project. Using the MoSCoW prioritisation methodology, the project's requirements will be or-

dered according to levels of critical impact [**14**](**Qiao, 2019**).

- **Must-Have** – These requirements need to be met for the project to be a success since they are critical to the projects deliver.
- **Should-Have** – Requirements that the project should have are determined to be non-critical to the time window in which the project is delivered, however, they will affect the usability of the product.
- **Could-Have** – These requirements are possible to complete if time and budget are in excess after meeting Must and Should requirements. They are desirable requirements but not necessary to the project's success.
- **Would-Have** – Requirements such as these are reserved for the least critical to the project's success. They are less desirable than could have requirements but are still valuable in their own right.

4.2 Functional Requirements

Functional requirements are statements on what the system should do and/or not do, or how it should react under certain scenarios. They need to be testable, and singular, meaning they describe one function of a system. This doesn't mean you can't have a functional requirement that is testing something that is doing a lot under the hood, it just specifies that it should be written in a way that it can either pass or fail according to the description **[5](Cal Poly, 2020)**. For the requirements in this project, both non-functional and functional sets are split according to each subproject or deliverable as set by each of their unique IDs.

4.2.1 Hardware

The functional requirements for the DarkRiscv core design of this project are quite simple. They just state that it should support all 3 of our instructions (**HW-3**, **HW-4**, **HW-5**) as well as the internal stacks(**HW-6**, **HW-7**), and the internal signal for when something goes wrong in the verification of stack contents(**HW-8**). The requirements for how this hardware will behave have been allocated to the software component as it will be easier to test against those requirements. All of these requirements are critical to the success of the project except for **HW-5** and **HW-7** which deal with multi-core support. So these last two meet the Should priority whereas the rest are deemed as Must.

4.2.2 Instruction Set Simulator

For the 2nd component of the project, the instruction set simulator modifications, much of the same that was set for the hardware component applies here as well. Requirements **ISS-2**, **ISS-3**, **ISS-4**, each is defined around supporting the 3 new opcodes, `ssst`, `ssld`, and `ssth`. Along with a software implementation supporting a shadow stack and additional stacks for multi-core support (**ISS-5**, **ISS-6**). Leaving **ISS-7** defining the use of an internal interrupt signal to process errors from the shadow stack. Again like the hardware component, **ISS-4** and **ISS-6** can be regarded as Should priority requirements as they deal with multi-core support whereas the remaining are critical to the overall success of the project.

4.2.3 Software

This component of the project naturally has the most functional requirements since it represents the majority of the functionality of both of the other components. This is due to the software component being made up of changes to the compiler, assembler, test programs for both components, as well as 2 demo applications that run on both components. Starting with the 1st Software requirement **SW-1**, this requirement is based on whether or not the compiler can emit the opcodes. Requirements then branch into whether the assembler supports each opcode(**SW-2** to **SW-4**). All of which mentioned are must priority except **SW-4**, which deals with the thread selection opcode making it non-essential as a should priority. **SW-5** through **SW-8** focus on

whether the compiler inserts the SSST and SSLD opcode at all, and if correctly within the function prologue/epilogue. **SW-9** through **SW-13** all deal with the functionality of the test program for the core, with the multithread related requirements set to Should, and all others designated as Must. Requirements **SW-14** through **SW-18** parallel those set for the core design, but this time for the Instruction set simulator. Finally for the remaining functional requirements of the software component, **SW-19** through **SW-24** outlines what the demo application will need to do on both the core and ISS components of the project. More specifically, the usage of shadow stack instructions in two scenarios where CFI is enforced, and when it is not, showcasing the exploitation or thwarting of a ROP based exploit on both systems. The requirements here follow Must for the core, and Should for the ISS.

4.3 Non-Functional Requirements

Contrasted from Functional requirements, Non-Functional Requirements deal with the characteristics of the system they are outlined for. Generally, they are harder to evaluate since they can be subjective in their definition. These types of requirements can outline things like what type of programming language or technology used in the project **[6](Chung, 2020)**. They are rather thought of as what constraints will be applied to the system for it to be deemed successful, not what the system will do but instead how it will do it.

4.3.1 Hardware

For the hardware component, the Non-Functional requirements mainly dictate things like what language will be used or architecture. **HW-1** states it needs to be based on a 32bit arch, not a 64bit arch, this informed the correct selection of a core design. **HW-2** also states that it needs to be written in Verilog, coupled with **HW-1** this narrowed the search down and eventually Dark-Riscv was chosen to be the most suitable core for developing the hardware component, based on the project's requirements. Both are essential to the success of the project so they are set as Must requirements. **HW-9** and **HW-10** deal with the core's performance in regards to the Verilog written. They mainly state that a definition shall enable the inclusion of all shadow stack related functionality and that if not defined, the core will function correctly regardless. This isn't essential to the success, it just helps ease of use.

4.3.2 Instruction Set Simulator

The Instruction Set Simulator's non-functional requirements range from what simulator will be chosen to how the various instructions will perform. Requirement **ISS-1**, dictates the selection of an instruction set simulator, whereas requirements **ISS-8** through **ISS-10** outline how the simulator should perform when dealing with multithread usage of the shadow stack instructions. Finally, **ISS-11** outlines the instruction set simulators restriction on internal shadow stack prints to be only performed if a verbosity flag has been set.

4.3.3 Software

The only Non-Functional requirement outlined for the software component is that it must only enable the emission of shadow stack instructions in the prologue and epilogue if the correct arch prefix has been used in configuring the compiler. This isn't critical to the project's success and can be added at a later date in the project's development making it a could priority requirement.

4.4 Requirements Use-Case

Good practice in both informing and developing a set of requirements is to create a use case delivered system. A use case is a description of how the system will be used, by who it will be used, and outline the sequences and structures in its usage [**10**](**Marchese, 2020**). For this project, a use case diagram has been created that can be paired with a use case table in the appendix section A.3. For the diagram, we can illustrate 3 different actors, which in this case are people, but could be various systems or organizations. In the scenario for the project, we will have a developer who uses the systems 3 deliverables to develop a solution for the customer. This solution in the form of code run through the compiler GCC will be enabled for the custom extension. The scenario accounts for either the testing of this solution on Spike the instruction set simulator that is modified to use the custom extension, or they can load it onto the actual core design, implemented on a system-on-chip. At this point, the scenario will have

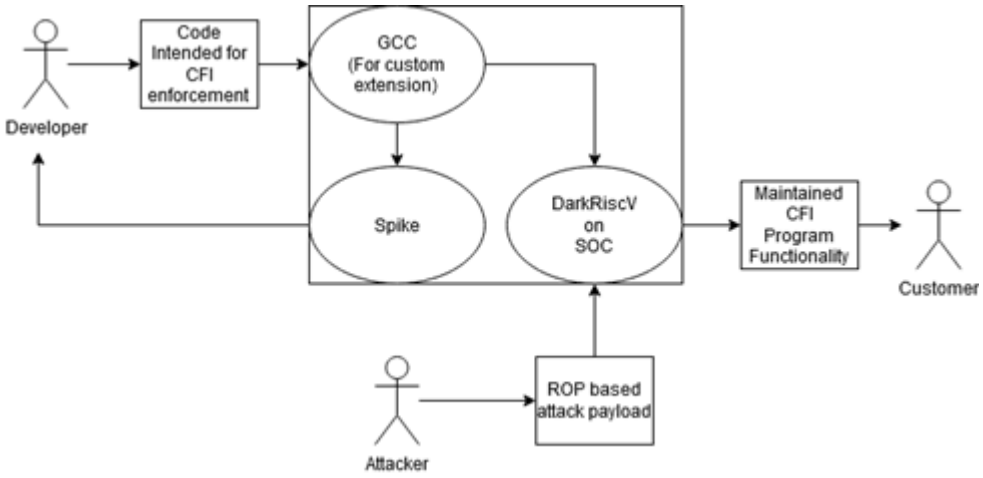


Figure 4.1: Requirements: Use Case diagram based on the project's requirements

either a failure or success outcome depending on the correct insertion of shadow stack instructions loaded onto the SOC. The scenario regardless of the outcome will have an attacker attempt to break CFI through their payload on the system. If instructions are correctly inserted, the customer will benefit from a maintained CFI, this acts as the success outcome.

Chapter 5

Methodology

The choice and correct implementation of a software development life cycle model is essential to the success of any large software development project. Even though the project is relatively small, it will still greatly benefit from enforcing a development methodology. For this project, a type of prototyping model will be applied in the form of evolutionary prototyping. Evolutionary prototyping is a software engineering prototyping model used to conserve efforts in the form of changes to software or hardware aka the prototype. This is contrasted against other prototyping models such as throw-away prototyping wherein at the end of the development of the prototype, any hardware or software artefacts are discarded and only used to inform the development process of the final product starting from scratch once again [**8**](**Heimdahl, 2020**).

The reasoning behind choosing this development methodology was in knowing that the design of the project wasn't well outlined initially, and using a ridged design methodology may help reduce development time or improve quality, it wouldn't guarantee that the design would likely work unless it had been established properly early in development. Ideally using this model allows for some exploration of designs and implementations which help inform further revisions to the overall design and specifications of the project. Using this methodology when working on a project that is based on customer feedback would be problematic, as with evolutionary prototyping it can take quite long to finish as requirements and design revisions can be indefinite as long as the customer isn't satisfied. In this case, the project isn't user-focused, so this allows us to estimate how many revisions will be needed for the project to reach all of its requirements.

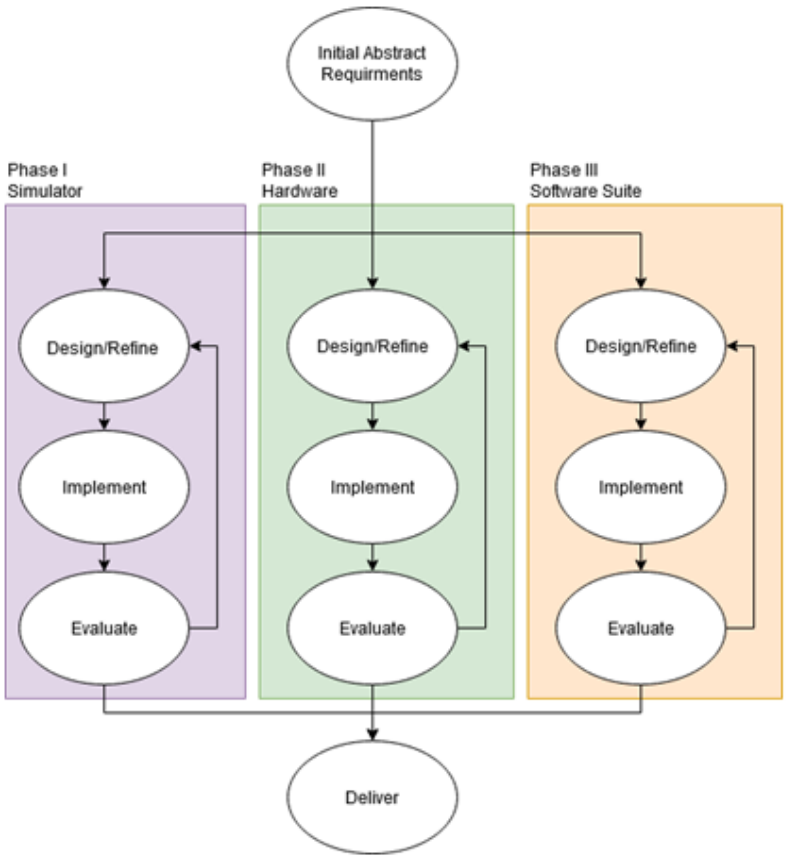


Figure 5.1: Methodology: Development Methodology Diagram for each of the 3 subprojects.

Using this development methodology on the project we can update the evolutionary prototyping model to accommodate multiple phases. The project is broken into 3 phases of development..

- **Phase 1:** Support of the shadow stack instructions within the Spike simulator.
- **Phase 2:** Support of the shadow stack instructions within the DarkRiscv core design.
- **Phase 3:** A software suite made up of changes to GCC along with demonstration code used to showcase and test out the functionality of the project designs.

The reasoning behind breaking out the model into 3 phases is that most of the work involved can be done in parallel as there is quite some work that can be done in parallel. Also, the project will attempt to create 3 main deliverables at the end of development.

Chapter 6

Design

The following designs due to using an evolutionary prototyping development methodology have been subject to multiple revisions throughout the project. Various changes have been made due to issues and blind spots experienced in the initial design revisions. The design documentation includes the initial instruction extension design along with the simulator and hardware designs. The use of top-level design is needed to illustrate the software stack necessary to produce software for both implementations within the project.

6.1 Top-Level Design

In a top-level design, the goal is to illustrate any major components by decomposing any of its subcomponents into single modules. For our top-level project design, one way to display all the components of the project is within a software stack. A software stack is a collection of components that will help deliver software functionality, and can be displayed as an architectural hierarchy or "stack" **[13](Ou, 2015)**. This stack does not exclusively show each phase of the project but outlines the components involved, the focus, however, will just be on the compiler/assembler, bare-metal application, and both implementations.

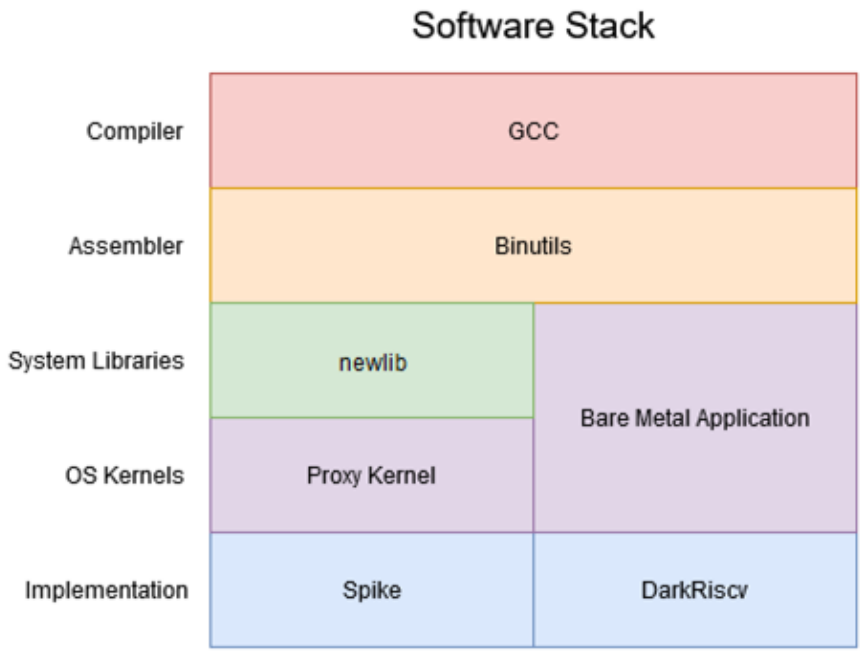


Figure 6.1: Design: Top-Level full software stack outline showcasing each component of the project.

Starting at the top, the project will be using GCC to generate instruction patterns and pass them off to the assembler Binutils to generate the correct machine code. Modifying both GCC to insert the new instructions as part of prologue and epilogue instruction patterns, as well as Binutils to translate instruction names into machine code. System libraries won't need modifying in this project but will be using Newlib as on the simulator via the proxy kernel. For the DarkRiscv core, the application is minimal and all the IO is custom written without a library involved, no changes will be needed for this section. The OS kernel on the DarkRiscv core, however, will be changed to test the new instructions, but aside from that, it will remain much of the same. Regarding the implementation, both the ISS Spike and the DarkRiscv core design will be heavily modified to implement the instructions, they will act as phase 1 and 2 of the project whereas GCC and the Bare Metal applications will make up phase 3.

6.2 ISA Extension

To extend the RISCV instruction set with the new proposed instructions, we need to develop designs for each to determine how they will be decoded. Both SSST and SSLD the push and pop functioning instructions of the shadow stack are very much hardcoded in their opcode design. No immediate data is loaded and no destination register is needed. They act very much like system opcodes but without any privileges needed. Any bits beyond the opcode region from 0-6 won't be decoded so we leave them blank. The bottom 2 bits of the opcode need to be set to 1s as this is standard for any opcode for them to be decoded. Using a simple bit pattern of 1010 and 0101 for each instruction is suitable as it does not match any existing instruction.

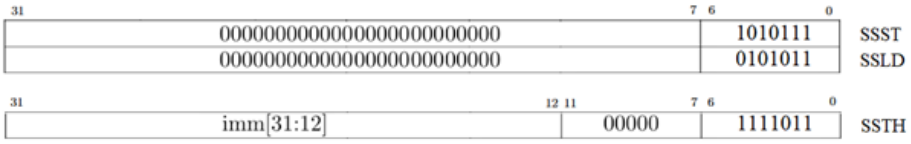


Figure 6.2: Design: ISA extension, design for 3 shadow stack opcodes according to RISCV ISA formats.

The SSTH instruction, on the other hand, works differently. It will need some kind of input data to set which internal shadow stack will be used for a different thread. This instruction won't ignore the upper bits during decoding from positions 12-31. Again the register designation at position 7-11 will be set to 0 since it will be ignored anyway. Using a pattern of 11110 as the opcode additionally is a gap in the existing opcodes and is suitable for a new instruction.

6.3 Simulator

In designing the changes to the simulator, all shadow stack functionality was developed in a separate program for testing. After evaluating that the functionality had been reached, a UML class diagram was developed with the shadow stack class and merged with some of the UML class diagram contents. For the simulator, we are basing its implementation on how HardScope for runtime evaluation was developed on Spike [**12**]**(Nyman, 2018)**. In that implementation, their class for runtime monitoring is constructed in the MMU class and gained reference to using the MMU for the rest of the simulators runtime. Taking that into account we can develop a UML class diagram that represents how the shadow stack class will work within the simulator, it will create on construction from the MMU class, 4 instances of a shadow stack set in a defines for stack count. Using the MMU class we can gain a reference to the shadow stack class and call its public methods to interact with any of its shadow stack internal data structures.

6.4 Hardware

For designing the hardware adjustments to the DarkRiscv core we can use data flow and simplified logic element designs to summarize the instructions and their related signal behaviour. The majority of the instructions design will be based around the use of multiplexers or "mux" for short which act as controlled switches. These elements will take an input signal which will be represented as a multi-bit or single bit selector value depending on the number of inputs or outputs being switched upon. In the design scheme, rectangles will represent registers as well as the use of common logic gate designs such as AND, OR, and XOR gates. For simplification, the shadow stack data structures which will be internally arrays of registers will be drawn as a stack of registers.

The hardware modifications can be broken down into 4 components, a dedicated reset signal for improper use of the shadow stack, the index control for maintaining correct reading and writing to the shadow stacks, and both designs for the functionality of the push/pop instructions in the form of store and load. The third instruction thread select, which acts as an index between internal stacks will allow for multi-core support on the hardware, and since DarkRiscv only supports 2 threads, we will only design in 2 stacks. The stack selection or thread selection instruction won't need to have a separate design since it can be implemented as part of each design as an additional signal.

6.4.1 Shadow Stack Reset Signal

The design of a shadow stack reset signal in hardware is important to establish how errors can be detected during shadow stack instruction execution. In the design, we need a 1-bit signal state which we can use in the rest of the DarkRiscv design. This is labeled as "`SS Reset`" and will be a register for holding that signal. "`SSsel`" acts as a 1-bit register that is loaded by the stack selector instruction using its immediate data in the opcode. This data won't need to be checked as part of the selector signal for the multiplexer as even if it is incorrect it is checked as either input to the mux which drives the reset signal register. Using this selector value, we can switch between 2 banks of multiplexers that each check bounds conditions. When any of these conditions are true upon their coupled selector signal then a "high" signal will be loaded into the reset signal at the end. The use of an XOR gate at the bottom of the multiplexer bank ensures on a global reset that the stack selector is cleared to 0 and the shadow stack reset signal is also set to 0. The full shadow stack reset signal design can be seen in the appendix section B.5.

6.4.2 Shadow Stack Store Instruction (SSST)

The basis around designing the store instruction consists of moving data from the return address register into one of the shadow stacks using the currently set index. This instruction is pretty easy to design in hardware since it only relies on 3 signals to function correctly. The stack selector signal, as seen in the appendix

section B.5 via "`SSsel`", the corresponding stack index register, and the SSST signal which is set during the decode cycle of the pipeline. It's important that we also maintain the currently indexed stack register to be its previous value when not executing the SSST opcode. The SSST design can be seen in appendix section B.4.

6.4.3 Shadow Stack Load Instruction (SSLD)

When designing the load instruction initially authentication was based around setting the reset signal seen in the appendix section B.5, however, this would have required designing an interrupt routine as part of phase 3 of the project. Since the design attempts to limit any external access to the shadow stack it didn't seem viable to design the instruction this way. Instead, attempts were made for authentication failure to update the return address register with the internally held return address in the form of a recovery RA register. This way CFI could be maintained in the event of corrupting the return address or an attacker modifying it on the stack. The design for this now outlines only reading from the 2 stacks using their corresponding indexes and switching the output based on which stack is being used via the stack selector register. To authenticate the address, the return address register is fed into an XOR with the output of the currently selected stack. If the result is not 0, which will be the result if both addresses match, then the shadow stack value will be output. Upon the use of the load instruction this shadow stack data will

be sent into the recover return address register. For it to be restored to the original return address register, we will just feed it with the recovery return address register when the load opcode is executed. The SSLD design can be seen in the appendix section B.3.

6.4.4 Shadow Stack Indexing

The core functionality of the 2 store/load instructions needs the ability to move about each stack and retain its position via indexing. We can accomplish this with the use of a register for each stack that stores the current index otherwise known as the stack pointer. Since the DarkRiscv core design only supports 2 cores we just need to design the indexing logic for 2 stacks. The end design yielded 4 multiplexers that fed into each other starting with the shadow stack internal reset signal, then the global reset signal, the load instruction decode signal, and the store instruction decode signal. Upon both reset signals, the index will be set to 0, however, if not triggered it will be fed the output of either an increment or decrement to its previously held value. Both the load opcode and the store opcode should only trigger selection if they are coupled with the corresponding stack selection signal. If both signals are 1 then the index will be incremented or decremented correctly. The design for which can be seen in the appendix section B.2.

Chapter 7

Implementation

7.1 Development Enviornment Setup

To start things off, the development of the ISA extension needs the correct environment. This requires both an IDE for the modification of the DarkRiscv core design in the form of Vivado as well as the instruction set simulator Spike to be installed and working with basic functionality. Both the core and the simulator require the riscv-gnu-toolchain, specifically for DarkRiscv it only needs the toolchain if modifications are needed to the current program and not the design. To acquire the toolchain it was simple as cloning the repo from GitHub and recursively downloading the various submodules it incorporates.

Since we are using the 32bit architecture of RISCV we will need to compile the toolchain for 32bit RISCV cross-compile support. We do this using an option of `-with-arch=rv32i` to build the base integer set ISA although alternatively, we can use 'e' in replace of 'i' for the embedded ISA which the only difference is less general-purpose registers. For the purposes, it won't make a difference which we select so 'i' was used.

Passing both options to the configure script will generate us a make file that can build the toolchain, we just need to select the prefix which will be the install path on the system. We are using the Linux subsystem that is supported on Windows 10 as the workspace environment, so effectively we can work on a Linux workstation with the support of the windows OS; this provides optimal workflow since any tool is easily available.

```
./configure -prefix=$RISCV --with-arch=rv32i --with-abi=ilp32d
```

We just need to run make now, with the option of `-j7` for 7 jobs to be set up which will run on the 7 threads leaving one thread for the OS since the CPU used to develop on has a max of 2 threads for each core.

```
make -j7
```

With a working toolchain setup, we can now add it to the environmental variables using export and the variable of `RISCV` which points to `/opt/riscv` on the Linux directory.

```
Export PATH=$PATH:$RISCV
```

A bit of info on the instruction set simulator Spike, to use standard LibC functions like `printf`, etc, we need to run the programs on a proxy kernel, this can be found in the `riscv-tools` repo on GitHub. Both compiling the Spike simulator and proxy kernel is pretty simple, just need to create a build directory and call the configure script one directory up from within the build directory using the prefix which is the previous install path via the RISCV environmental variable. When done we can install both the PK (proxy kernel) and the Spike simulator to `/opt/riscv/bin` that is also linked to the PATH variable.

Effectively at this point, we have a full software stack setup for RISCV development. We just need to test if everything is working. To do so we can compile a test program that just acts as a hello world program.

```
riscv64-unknown-elf-gcc -o hello hello.c
```

And run it on Spike via the proxy kernel..

Figure 7.1: Implementation: ISS running hello world test program to test the working development enviornment.

All that is left to test the software stack on is the DarkRiscv core. DarkRiscv implements its bootloader and main program via a make file and linker script so that all code lands within a ROM, and any data is inserted into RAM. Input and output related code such as for UART is set up so that when called it will effect upper memory at `0x80000000` which is mapped to actual pins on the FPGA. No changes are needed to this code for it to work, but to prove we can effect it a print statement was inserted after the initial setup and compiled with the toolchain.

```
      INSTRUCTION SETS WANT TO BE FREE

boot0: text@0 data@4096 stack@8192
board: simulation only (id=0)
build: darkriscv fw build Mon, 20 Apr 2020 16:56:48 +0100
core0: darkriscv@100.0MHz with rv32i+MT+MAC
uart0: 115200 bps (div=868)
timr0: periodic timer=100000000Hz (io.timer=0)

Welcome to DarkRISCV!
```

Figure 7.2: Implementation: DarkRiscv bare metal application successfully running on FPGA hardware. Serial data captured with Arduino Serial Monitor.

To load the program on the FPGA we simply both synthesise the core design as well as implement it. During implementation, the code compiled into RAM and ROM files will be set into the design and be used in the final generated bitstream. Since the code is part of the actual RTL design making changes and testing them will be quite the process since the time to process all the RTL into a final bitstream is around 5 minutes on the machine used for development. However, with a bit of patience, we can confirm that the core has implemented the modified code which completes the setup of the development environment. The next step will be to implement the new instructions into an assembler.

7.2 Binutils ISA Extension Support

To enable support for the newly designed instructions in the toolchains assembler, we only need to modify two files. This requires creating a mask that will be used to perform an AND operation against the opcode, and in this case we know that the opcode will be static every time with the exception for the SSTH instruction. Since SSTH was modeled after the load upper immediate instruction, we use the same bitmask for that instruction. As well as a full mask for SSST and SSLD as they will act as system-related opcodes. Regarding changes to `riscv-opc.c` we don't need to set any operand rules such as 'd' for destination or 's' for source since the instructions won't be working with any data aside from the SSTH instruction that only needs data from the upper bits of the instruction. We can simply copy the format used for load upper immediate but remove the d since we will know the destination internally.

`riscv-gnu-toolchain\riscv-binutils\include\opcode\riscv-opc.h`

```
#define MATCH_SSST 0x2b
#define MASK_SSST 0xffffffff
#define MATCH_SSLD 0x57
#define MASK_SSLD 0xffffffff
#define MATCH_SSTH 0x7b
#define MASK_SSTH 0xffffffff
```

```
DECLARE_INSN(ssld, MATCH_SSST, MASK_SSST)
DECLARE_INSN(ssst, MATCH_SSLD, MASK_SSLD)
DECLARE_INSN(ssth, MATCH_SSTH, MASK_SSTH)
```

riscv-gnu-toolchain\riscv-binutils\opcodes\riscv-opc.c

```
{"ssst", 0, INSN_CLASS_I, "", MATCH_SSST, MASK_SSST, match_opcode,
    0 },
{"ssld", 0, INSN_CLASS_I, "", MATCH_SSLD, MASK_SSLD, match_opcode,
    0 },
{"ssth", 0, INSN_CLASS_I, "u", MATCH_SSTH, MASK_SSTH, match_opcode,
     0 },
```

With those changes made, that is all that this toolchain will need to encode the instructions into assembly via Binutils. However at this point, we will not be able to dynamically insert them via instruction patterns, we will just rely on the use of inline assembly in the code to invoke the assembler to insert the instructions. We can add 3 lines of inline assembly to a program and compile it, using Objdump and passing the `-d` command to tell it to generate a disassembly of the program we can inspect to see if the instructions have been processed correctly. At this point the simulator and core design still need to be changed to decode and execute them.

7.3 Simulator Support

As noted in the design section, we can implement the shadow stack functionality through a class and both instantiate and reference it through the MMU as inspired by HardScopes implementation on Spike. This leaves us with several files that need changing. For starters, we need to add support to decode the instructions, this is done in the same fashion as we did for Binutils, however, we only need to add the match/mask defines and declare them within the encoding header file. In the

Spike simulator, each instruction's functionality is outlined within the `insns` directory via a header file that contains a small amount of C needed to simulate the actions of the instructions. We will create three of these files for each of the instructions, `ssth.h`, `ssld.h` and `ssst.h`. To access the shadow stack component as part of the simulated CPU extension we need to gain reference to its class that was constructed when the MMU class was. Easily enough the processor class contains a method to obtain a reference to the MMU class via `get_mmu()`, we then simply call the public method of `get_sstack` within it to gain reference to the instance of the shadow stack. From here we can call any of its public methods which grants us the full functionality needed to simulate the instructions.

```
1         sstack = new sstack_t();
```

```
1         sstack_t* get_sstack() { return sstack; }
```

```
1     sstack_t *ss;
2
3     ss = p->get_mmu()->get_sstack();
```

For the store instruction, we just need to load the value from the return address register internally using the macro function of `READ_REG` and passing it 1 (the return address register number). With the value read from the return address register, we can pass it to the shadow stack method 'push' to load it at the internally held indexed position on the stack.

```
1     reg_t ra = READ_REG(1);
2     printf("ssst ra=%lx pc=%lx\n", ra, pc);
3     sstack_t *ss;
4     ss = p->get_mmu()->get_sstack();
5     ss->push(ra);
```

Loading needs a bit more functionality. It still gains reference to the shadow stack but now it needs to use the popped value from the shadow stack as input to the authenticate method. The output of the authenticate method will set a condition whether or not we should restore the return address register with the internal return addressed popped of the shadow stack or perform no additional operation.

```
1  sstack_t *ss;
2  ss = p->get_mmu()->get_sstack();
3  int dout = ss->pop();
4  int result = ss->authenticate(READ_REG(X_RA), dout);
5  printf("ssld ra=%lx pc=%lx\n", READ_REG(X_RA), pc);
6  if(result != 0)
7  {
8      printf("dout:%x ra_reg:%x result:%x\n",dout, READ_REG(X_RA),
           result);
9      WRITE_REG(X_RA, dout);
10 }
```

Regarding the thread selection instructions, all we need to do after gaining reference to the shadow stack class instance is to call the select method with the upper immediate data of the instruction. Since the upper immediate data will retain its bit position when we pull it from the instruction, we just need to shift it 12 bits to the right to align it to 0.

```
1 reg_t ra = READ_REG(1);
2 printf("ssst ra=%lx pc=%lx\n", ra, pc);
3 sstack_t *ss;
4 ss = p->get_mmu()->get_sstack();
5 ss->push(ra);
```

To implement the shadow stack class, we need two files in the `riscv` directory of `riscv-isa-sim`, `sstack.cc` and `sstack.h`. The `c++` code will implement the push/pop functionality along with authentication and stack selection. Additionally, we can use an assert macro to print an error and abort the program if a fault is detected. We must also outline the class constructor so that it initialises all the variables such as the stack selector and both stack contents/stack pointer for each created internal shadow stack.

```
/** @brief Shadow Stack Constructor
 *
 *  @param void
 *  @return Shadow Stack Class instance
 */
sstack_t::sstack_t(void)
{
  stack_selector = 0;
  verbose_mode = false;

  for(int i = 0; i < SHADOW_STACK_COUNT; i++)
  {
    ss_banks[i].sp = 0;

    for(int p = 0; p < SHADOW_STACK_SIZE; p++)
    {
      ss_banks[i].SSTACK[p] = 0;
    }
  }
}


/** @brief Sets whether the Shadow Stack module prints internal
    info
 *
 *  @param bool mode, true allows it, false denies it
 *  @return void
 */
void sstack_t::set_verbose_mode(bool mode)
{
  verbose_mode = mode;
}


/** @brief Pushes the return address to the Shadow Stack selected
 *
 *  @param int ra, the return address
 *  @return void
 */
void sstack_t::push(int ra)
{
  shadow_stack* ss = &ss_banks[stack_selector];

  if(ss->sp >= SHADOW_STACK_SIZE)
  {
    SSTACK_ASSERT("stack is full!");
  }

  else
  {
    ss->SSTACK[ss->sp++] = ra;

    if(verbose_mode)
    {
      printf("ssst sp:%d ra:%x\n", ss->sp-1, ss->SSTACK[ss->sp-1]);
    }
  }
```

```
57 }
58
59 /** @brief Pushes the return address to the Shadow Stack selected
60  *
61  *  @param void
62  *  @retrun int pop_val, the return address
63  */
64 int sstack_t::pop(void)
65 {
66   shadow_stack* ss = &ss_banks[stack_selector];
67   int pop_val = 0x33;
68
69   if(ss->sp < 1)
70   {
71     SSTACK_ASSERT("stack is empty!");
72   }
73
74   else
75   {
76     pop_val = ss->SSTACK[--ss->sp];
77
78     if(verbose_mode)
79     {
80       printf("ssld sp:%d ra:%x\n", ss->sp, pop_val);
81     }
82   }
83
84   return pop_val;
85 }
86
87 /** @brief Pushes the return address to the Shadow Stack selected
88  *
89  *  @param int ra, the return address to authenticate
90  *  @param int din, the data popped off the stack used to
       auhthenticate with
91  *  @retrun int result, the output from XORing the 2 inputs, needs
       to be 0 to be successful
92  */
93 int sstack_t::authenticate(int ra, int din)
94 {
95   int result = din ^ ra;
96
97   if(verbose_mode)
98   {
99     printf("auth ra=%lx datain=%lx\n", ra, din);
100  }
101
102  return result;
103 }
104
105
106 /** @brief Changes the Shadow Stack selecter value
107  *
108  *  @param int selector, a value that represents which stack to use
       , 0-3(max is set by SHADOW_STACK_COUNT
109  *  @retrun void
110  */
```

```
111 void sstack_t::select(int selector)
112 {
113
114   if(verbose_mode)
115   {
116     printf("ssth %d\n", selector);
117   }
118
119   if((selector < 0) || (selector >= SHADOW_STACK_COUNT))
120   {
121     SSTACK_ASSERT("selector is not valid!");
122   }
123
124   else
125   {
126     stack_selector = selector;
127   }
128 }
129
130
131 /** @brief Clears all Shadow Stacks
132  *
133  *  @param void
134  *  @retrun void
135  */
136 void sstack_t::clear(void)
137 {
138   for(int i = 0; i < SHADOW_STACK_COUNT; i++)
139   {
140     shadow_stack* ss = &ss_banks[i];
141
142     for(int p = 0; p < SHADOW_STACK_SIZE; p++)
143     {
144       ss->SSTACK[p] = 0;
145     }
146
147     ss->sp = 0;
148   }
149 }
```

The header file for the shadow stack component will outline a struct for a stack, which is made up of a stack pointer and a stack size array. The size and amount of stacks to be allocated are set in a definition.

```
1
2  #define SHADOW_STACK_SIZE 0x100
3  #define SHADOW_STACK_COUNT 4
4
5  struct shadow_stack {
6    int sp;
7    int SSTACK[SHADOW_STACK_SIZE];
8  };
9
10 class sstack_t {
11 private:
12
13   shadow_stack ss_banks[SHADOW_STACK_COUNT];
14   int stack_selector;
15   bool verbose_mode;
16
17 public:
18
19   /** @brief Pushes the return address to the Shadow Stack selected
20    *
21    *  @param int ra, the return address
22    *  @return void
23    */
24   void push(int ra);
25
26   /** @brief Pushes the return address to the Shadow Stack selected
27    *
28    *  @param void
29    *  @retrun int pop_val, the return address
30    */
31   int pop(void);
32
33   /** @brief Pushes the return address to the Shadow Stack selected
34    *
35    *  @param int ra, the return address to authenticate
36    *  @param int din, the data popped off the stack used to
          auhthenticate with
37    *  @retrun int result, the output from XORing the 2 inputs,
          needs to be 0 to be successful
38    */
39   int authenticate(int ra, int din);
40
41   /** @brief Changes the Shadow Stack selecter value
42    *
43    *  @param int selector, a value that represents which stack to
          use, 0-3(max is set by SHADOW_STACK_COUNT
44    *  @retrun void
45    */
```

```
46    void select(int selector);
47
48    /** @brief Clears all Shadow Stacks
49     *
50     *  @param void
51     *  @retrun void
52     */
53    void clear(void);
54
55    /** @brief Sets whether the Shadow Stack module prints internal
          info
56     *
57     *  @param bool mode, true allows it, false denies it
58     *  @return void
59     */
60    void set_verbose_mode(bool mode);
61
62    /** @brief Shadow Stack Constructor
63     *
64     *  @param void
65     *  @return Shadow Stack Class instance
66     */
67    sstack_t();
68
69  };
```

Now with all of the code in place, we can amend the Makefile include referencing both headers and sources for the shadow stack component as well as its related instructions.

```
1  sstack.h \
```

```
1  sstack.cc \
```

```
1  riscv_insn_sstack = \
2  ssst \
3  ssld \
4  ssth \
```

7.4 Core Design

Using DarkRiscv as a base we only need to target one file for the extension to be implemented into. In `darkriscv.v` there exist only 2 main events that form the pipeline, decode and execute. Initially, we need to define what the new opcodes will be by writing them as a variable of 7bits wide according to their opcode pattern. We will also need to establish what registers and signals the instructions will need to be controlled correctly. This includes the shadow stacks which act as 32bit wide registers in an array 256 deep.

A few reset signals will be used to drive the interrupt if a shadow stack instruction has been executed illegally. There is also the need for a few registers to store the value of the stack pointers as well as their selection to support multithreading functionality. Additionally, we need a register for the return address that will be loaded from the shadow stack into the real return address register. The need for this is since if we wire the return address to the shadow stack it becomes more difficult to control through how it gets implemented. So a register to buffer the output from the selected shadow stack was used in the end to help write things more simply.

```
1  `ifdef __SSTACK__
2
3      reg SS_RES = 0;
4
5    reg [31:0] RECOVERY_RA = 32'd0;
6
7      reg [31:0] SHADOW_STACK_1 [0:255];
8      reg [31:0] SHADOW_STACK_2 [0:255];
9
10     reg [31:0] SS1_INDEX_VAL = 32'd0;
11     reg [31:0] SS2_INDEX_VAL = 32'd0;
12
13     reg [31:0] SS_THREAD_SEL = 32'd0;
14
15     wire RES = IRES || SS_RES;
16
17     reg XSSLD, XSSST, XSSTH;
18
19 `else
20     wire RES = IRES;
21 `endif
```

Further on during the decoding event, we use 3 new signals that are set when the corresponding opcode is detected within the lower 7bits of the instruction data. Next in the execution event portion, we set up the majority of the logic needed to drive the shadow stacks. To start we can base the shadow stack selector primarily on the signal from the use of the thread selection opcode. If high it will load the selection register value to that of the upper immediate data. Else it will be determined by either its current value or that of 0 if the reset signal is high.

To drive storing values on the shadow stacks a case statement is used that updates the corresponding stack to the return address register if the store opcode is detected else it will default to its current value. All of which are switched upon using the stack selection register value. The next step after a store is executed is to update the stack pointer, but first, we set the shadow stack reset signal based on whether the index will go out of bounds. If it will be within range after changing according to the operation it won't set the internal reset signal. This reset signal will also prevent the stack index values from applying the change and instead reset it. Just after we set the recovery return address register to be the output of the corresponding indexed shadow stack based on if general return address register XORed with the shadow stacks return address value is not equal to 0. If it is 0 then it will just be set to the general return address value instead. Effectively this drives the recovery register to be updated with the previous return address if the general return address register has been corrupted, recovering execution flow.

```
1  `ifdef __SSTACK__
2
3          // execute shadow stack thread select, set the selector to
               the upper immediate value
4          SS_THREAD_SEL = SSTH ? IDATA[31:12] : RES ? 0 :
               SS_THREAD_SEL;
5
6          // switch upon which stack to use based on the shadow stack
                selector value
7          case (SS_THREAD_SEL)
8              1: SHADOW_STACK_2[SS2_INDEX_VAL] = SSST ? REG1[1] :
                   SHADOW_STACK_2[SS2_INDEX_VAL];
9
10             default: SHADOW_STACK_1[SS1_INDEX_VAL] = SSST ? REG1[1]
                    : SHADOW_STACK_1[SS1_INDEX_VAL];
11         endcase
12
13   // handle exeptions and trigger reset signal
14         case (SS_THREAD_SEL)
15
16             1: SS_RES = SSTH ? (SS_THREAD_SEL > 1) :
17                         SSST ? (SS2_INDEX_VAL > 254) :
18                         SSLD ? (SS2_INDEX_VAL < 1)  :
19                         SS_RES ^ RES;
20
21             default: SS_RES = SSTH ? (SS_THREAD_SEL > 1) :
22                               SSST ? (SS1_INDEX_VAL > 254) :
23                               SSLD ? (SS1_INDEX_VAL < 1)  :
24                               SS_RES ^ RES;
25         endcase
26
27
28         // set the shadow stack stack pointer
29         case (SS_THREAD_SEL)
30
31             1:
32             begin
33             SS2_INDEX_VAL = SS_RES ? 0 :                  // shadow
                   stack related reset clears sp
34                             RES ? 0 :                     // normal
                                reset clears sp
35                             SSLD ? (SS2_INDEX_VAL - 1) : // shadow
                                stack load opcode dec the sp
36                             SSST ? (SS2_INDEX_VAL + 1) : // shadow
                                stack store opcode inc the sp
37                             SS2_INDEX_VAL;               // in all
                                other cases keep the sp the same
38
39             // if selecting other stack only proccess reset signals
40             SS1_INDEX_VAL = SS_RES ? 0 :
41                             RES ? 0 :
42                             SS1_INDEX_VAL;
43             end
44
45
46
47
```

```
48
49              default:
50              begin
51              SS1_INDEX_VAL = SS_RES ? 0 :                  // shadow
                    stack related reset clears sp
52                              RES ? 0 :                      // normal
                                    reset clears sp
53                              SSLD ? (SS1_INDEX_VAL - 1) : // shadow
                                    stack load opcode dec the sp
54                              SSST ? (SS1_INDEX_VAL + 1) : // shadow
                                    stack store opcode inc the sp
55                              SS1_INDEX_VAL;                // in all
                                    other cases keep the sp the same
56
57              // if selecting other stack only proccess reset signals
58      SS2_INDEX_VAL = SS_RES ? 0 :
59                      RES ? 0 :
60                      SS2_INDEX_VAL;
61              end
62          endcase
63
64
65          // execute the authentication of the return address agaisnt
                the current value in the shadow stack upon loading
66    if(SSLD)
67    begin
68
69      // based on which stack is selected, the recovery address is
            set to the current value in the stack, or the ra register
70      // if values mismatch recovery ra becomes the value in the
            shadow stack, else it will be the ra value in the ra
            register
71      case (SS_THREAD_SEL)
72       1: RECOVERY_RA = (REG1[1]^SHADOW_STACK_2[SS2_INDEX_VAL]) != 0
              ? SHADOW_STACK_2[SS2_INDEX_VAL] : REG1[1];
73
74                  default: RECOVERY_RA = (REG1[1]^SHADOW_STACK_1[
                        SS1_INDEX_VAL]) != 0 ? SHADOW_STACK_1[
                        SS1_INDEX_VAL] : REG1[1];
75      endcase
76    end
77  `endif
```

However, an additional step to implement this recovery is needed in the logic that drives the general-purpose registers. In both banks we set the return address register to be loaded with the recovery registers value when the load instruction is executed.

```
1  `ifdef __RV32E__
2  REG1[DPTR] <=   RES ? (RESMODE[3:0]==2 ? `__RESETSP__ : 0)  :
               // reset sp
3  `else
4  REG1[DPTR] <=   RES ? (RESMODE[4:0]==2 ? `__RESETSP__ : 0)  :
               // reset sp
5  `endif
6                  HLT ? REG1[DPTR] :        // halt
7                  !DPTR ? 0 :                // x0 = 0, always!
8                  AUIPC ? PC+SIMM :
9                  JAL||
10                 JALR ? NXPC :
11                 LUI ? SIMM :
12                 LCC ? LDATA :
13                 MCC||RCC ? RMDATA:
14 `ifdef __MAC16X16__
15                 MAC ? REG2[DPTR]+KDATA :
16 `endif
17
18 // if using a shadow stack and executing the ssld opcode, the ra
       register will be set to the recovery ra register
19 `ifdef __SSTACK__
20                 SSLD ? RECOVERY_RA :
21 `endif
22                 //MCC ? MDATA :
23                 //RCC ? RDATA :
24                 //CCC ? CDATA :
25                       REG1[DPTR];
```

Overall these were the only changes needed to implement the design into the core, along with adding a definition for enabling the shadow stack logic in the synthesized design.

7.5 GCC Backend Support

Now that the instructions work within the simulator as well as on an FPGA running the extension to the DarkRiscv core, we need to develop the means to dynamically insert these instructions. The load and store instructions are only suitable in functions prologues and epilogues before and after the return address has been saved and restored. Ideally, we need to have "store" be generated in the prologue as the 1st instruction when entering the assembly of a function call. Also, for the load instruction to happen just before a return.

Within GCC's backend, there are only 2 main files that are the concern regarding instruction emission patterns. Riscv.md and Riscv.c inside the configuration folder for the RISCV architecture outline both the machine definitions for this architecture as well as c code for generating them.

```
riscv-gnu-toolchain\riscv-gcc\gcc\config\riscv
```

This portion of development proved to be quite the challenge since machine definitions were pretty cryptic to understand. Instruction emission was also hit or miss and seemly would break after a change of position within the code. Eventually, after lots of time testing various placements and definition formats, a solution was found using a diff constant in both the store and load definitions.

```
    (define_insn "ssst"
    [(unspec_volatile [(const_int 0)] UNSPECV_SSST)]
    ""
    {
        return "ssst";
    })

    (define_insn "ssld"
    [(unspec_volatile [(const_int 1)] UNSPECV_SSLD)]
    ""
    {
        return "ssld";
    })
```

The instruction generation for the prologue is invoked before the stack frame is constructed.

```
void
riscv_expand_prologue (void)
{
    struct riscv_frame_info *frame = &cfun->machine->frame;
    HOST_WIDE_INT size = frame->total_size;
    unsigned mask = frame->mask;
    rtx insn;


    if(TARGET_SSTACK)
        emit_insn(gen_ssst());
```

For the epilogue, we need to cover 3 types of returns, normal, returns from library functions, and "sibcalls" which are for direct function call returns.

```
    if ((style == NORMAL_RETURN) && riscv_can_use_return_insn ())
    {
        if(TARGET_SSTACK)
            emit_insn(gen_ssld());

        emit_jump_insn (gen_return ());
        return;
    }
```

```
if (use_restore_libcall)
{
    rtx dwarf = riscv_adjust_libcall_cfi_epilogue ();
    insn = emit_insn (gen_gpr_restore (GEN_INT (
        riscv_save_libcall_count (mask))));
    RTX_FRAME_RELATED_P (insn) = 1;
    REG_NOTES (insn) = dwarf;

    if(TARGET_SSTACK)
        emit_insn(gen_ssld());

    emit_jump_insn (gen_gpr_restore_return (ra));
    return;
}
```

```
else if (style != SIBCALL_RETURN)
{
    if(TARGET_SSTACK)
        emit_insn(gen_ssld());

    emit_jump_insn (gen_simple_return_internal (ra));
}
```

7.6 Issues Encountered

During implementing the project, a multitude of issues was experienced. Starting with setting up the environment, establishing a working proxy kernel to run programs on in 32bit mode was a struggle. Compiling riscv-pk in a 32bit arch would cause it to crash upon launching in Spike. The solution for the project was to run everything in the simulator in 64bit mode as it would not matter since the design could work in both 32bit and 64bit architectures, testing for 32bit mode would be solely conducted on the hardware.

Issues with modifying Binutils were minimal but it was overlooked a few times when testing out different versions of the riscv toolchain that the format of how opcodes were defined in riscv-opc.c changed, and replacing the file rather than modifying it was always important to remember. Compared to the hardware component of the project, the simulator experienced barely any issues in implementing the shadow stack design. Since it was prototyped and unit tested outside of the simulator, its integration went smoothly. The fact that it could be tested and iterated quickly played a big role in its smooth transition.

Regarding the core design implementation, development time for this component took the longest. Due to the length of time needed to synthesise, implement, and then both generate a bitstream as well as write it to the board. Getting the core design to work was a

painful process, and in hindsight should have been prototyped through simulation rather than working with the full core design for its entire implementation. Initially, the design was going to be adjusted to support loading the boot loader and main program from external flash so that any change to the code could just be programmed on a chip rather than run through the steps needed to integrate it into the bitstream. But the number of problems that needed to be solved such as the design and implementation of an SPI flash sequencer was hard to gauge and the project dropped this idea in favour for prioritising the core design development.

Towards the end of the project's development, an important issue was detected with how Newlib would compile. The assembly behind internal functions called using `printf` would break CFI through the uses of some indirect jumps. Through evaluating the execution flow of the call to `printf`, it was seen that the issue stemmed from `_flush_r` not containing a ssst opcode due to how it directed execution to `__sflush_r`. this would then be detected when swrite would call write˙r and after `_write`, whereupon returning the shadow stack would not be inline with the return address stored on the stack and end up triggering a segmentation fault. A workaround at the time of writing this report for Newlib would be to write a wrapper for the write syscall that produced the functionality of `printf`. Currently, the exact issue of why this is generated in assembly is being looked into, with the expectation of modifying Newlib to maintain CFI when a shadow stack is being used. Oddly

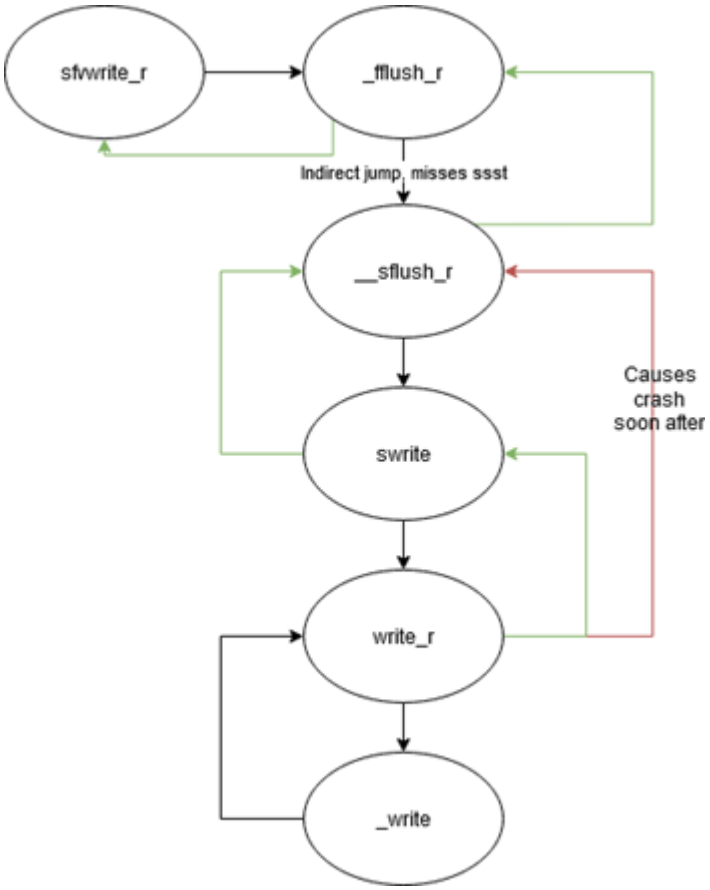


Figure 7.3: Implementation: Newlib printf Execution Flow Diagram

enough, quite a few other Libc functions such as sprint, scanf etc have been used and haven't encountered an issue.

7.7 Testing

Testing the various components of the implementation was procedural. Due to the complexity of code or designs being modified, functionality always had to be confirmed whether or not the changes were to blame or it being something altogether. This would be classed as unit testing after each component had been modified and then verified for the correct operation. Normally a project testing life cycle would progress from unit testing to integration testing, system testing, and finally acceptance testing.

However, due to the nature of this project, acceptance testing will cover all previous testing areas through a series of white-box tests and black-box tests on both a component level and system level. Each of the acceptance tests will cover several requirements as well as sometimes needing other tests to be completed to be performed, these will be outlined as the prerequisite for the test. All acceptance tests can be found in the appendix section C.1.

Each test will also require a series of steps that form the actions and a description of the expected outcome after each step has been produced. If all the expected observations are witnessed then the test should be deemed as a PASS rather than a FAIL. The project's acceptance tests will also be matched against its functional and non-functional requirements list. Using a traceability matrix in appendix section C.2 to display how each

test will cover different requirements, and ultimately show how the acceptance test results will determine the overall success of the project's delivery.

7.7.1 White-Box Testing

The white-box tests created for this project will require access to the project's code and designs to observe the desired expectations. These tests will be designed around the internal functionality of the project's components, such as if the implementations are being applied correctly and if they are being used correctly.

7.7.2 Black-Box Testing

For the project's black-box testing, tests will be focused on both forms of the completed implementations running the demo program, deliverables 1 and 2. This demo application is considered to be part of the 3rd deliverable along with the changes to the compiler, however, it will be the interface for the black box testing that will be conducted upon the 1st two deliverables. The PASS of all black-box tests along with the additional white-box tests will confirm the success of each deliverable with the requirements.

Chapter 8

Evaluation &Conclusion

At the end of this project's development for the timescale given, there were a lot of insights gathered regarding how it could be been developed differently. The project's research was staggered throughout development and because of this, all other phases of the project's development were affected. A few papers discovered during the research section outlined some implementations that were similar or useful for the project after they had been developed iteratively on their own. This knowledge would have sped up the time needed to develop the changes to the compiler primarily.

Throughout the project, the requirement set changed as well. Initially, the design was much similar to that of how Pointer Authentication instructions were used with cryptographic material. This was leftover from initial research notes and both informed the initial requirements as well as the initial design. Upon further refinement, both the design and requirements were adjusted to meet the new optimizations which yielded a

simpler shadow stack design. In hindsight, if the problem was understood more initially, the design would not have been affected since it would be informed by a requirement set more in line with what the end version looked like.

Reflecting on implementing the core design, time could have been significantly reduced if the shadow stack component had been developed in its separate module for easy testing through simulation.

Overall, the initial work on the design and implementation would not have panned out as it would have so early if it wasn't for the feedback and suggestions from the project supervisor. During a brainstorming session, the notion of using a shadow stack was discussed and the workings of the 3 instructions were determined.

8.1 Future Work

Looking towards how the project can progress beyond its current state at the time of writing this report, there still are quite a few things to nail down. For starters, the DarkRiscV core on the FPGA hardware struggled to perform correctly in a multicore mode, so the shadow stack component couldn't be tested properly on the end hardware for multicore support. Changes to the core can still be carried out to get this working correctly and perform the multicore tests. Additionally, code within riscv Newlib specifically in the execution flow starting with printf, needs to be examined for how it might have

broken CFI with mismatched shadow stack instructions. Whether this will inform additional tweaks to how the compiler detects instruction insertion, or simply creating alternate code in Newlib that is toggled with a shadow stack flag.

LLVM can also be modified for support of these instructions going forward with the project. In terms of testing the implementation, Newlib was chosen as it was the simplest to get working, as Glibc introduced several difficulties in getting set up within the development environment. Further work on the project would most likely entail both testing Glibc with these instructions through simulating a Linux kernel, as well as running Glibc linked code on the actual FPGA hardware. The projects further work might also modify QEMU to support this shadow stack and its instructions. A lot more testing can be done as well, specifically looking at the performance overhead of using this component to maintain CFI. As much as there is already that can be expanded on with the project, a more ambitious undertaking is to add support for CFI enforcement against JOP techniques. Of course, this would require a full new design to go alongside the shadow stack as it has completely different requirements, however, the changes to the compiler would be very similar.

Bibliography

[1] Abadi, Martín &Budiu, Mihai &Erlingsson, Úlfar &Ligatti, Jay. (2009). *Control-flow integrity: Principles, implementations, and applications.* ACM Trans. Inf. Syst. Secur.. 13. 10.1145/1609956.1609960.

[2] Bletsch, Tyler &Jiang, Xuxian &Freeh, Vincent &Liang, Zhenkai. (2011). *Jump-oriented programming: a new class of code-reuse attack..* 30-40. 10.1145/1966913.1966919.

[3] Buchanan, Erik &Roemer, Ryan &Shacham, Hovav &Savage, Stefan. (2008). *When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC.*

[4] Burow, Nathan &Zhang, Xinping &Payer, Mathias. (2019). *SoK: Shining Light on Shadow Stacks.* 985-999. 10.1109/SP.2019.00076.

[5] Cal Poly College of Engineering. (2020). *Functional Requirements [online] Available at: ¡http://users.csc.calpoly.edu/ csturner/courses/308w09/FunctionalRequirements.pdf [Accessed 22 April 2020].*

[6] University of Texas at Dallas. Lawrence Chung. (2020). *Non-Functional Requirements [online] Available at: ¡https://personal.utdallas.edu/ chung/SYSM6309/NFR-18.pdf [Accessed 22 April 2020].*

[7] grsecurity. (2020). *[online] Available at: https://pax.grsecurity.net/docs/aslr.txt [Accessed 8 April 2020].*

[8] The University of Edinburgh. Prof. Mats Heimdahl. (2020). *Sommerville Chapter 2 and 3, The Process. [online] Available at: http://www.inf.ed.ac.uk/teaching/courses/inf2c-se/Lectures/Lectures-2014/lecture-14-process.pdf [Accessed 22 April 2020].*

[9] Liljestrand, Hans &Nyman, Thomas &Wang, Kui &Perez, Carlos &Ekberg, Jan-Erik &Asokan, N. (2018). *PAC it up: Towards Pointer Integrity using ARM Pointer Authentication*

[10] PACE University. Prof. Francis T. Marchese. (2020). *Use Case Diagrams Tutorial. [online] Available at: http://csis.pace.edu/ marchese/CS389/L9/Use%20Case%20Diagrams.pdf [Accessed 22 April 2020].*

[11] Nyman, Thomas &Dessouky, Ghada &Zeitouni, Shaza &Lehikoinen, Aaro &Paverd, Andrew &Asokan, N. &Sadeghi, Ahmad-Reza. (2017). *HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement*

[12] Nyman, Thomas. HardScope. (2018). GitHub repository. *[online] Available at: https://github.com/runtime-scope-enforcement/riscv-isa-sim*

[13] Albert Ou UC Berkely (2015). *Software Tools Bootcamp - RISC-V ISA Tutorial —HPCA-21. [online] Available at: https://riscv.org/wp-content/uploads/2015/02/riscv-software-stack-tutorial-hpca2015.pdf [Accessed 22 April 2020].*

[14] Ma Qiao. (2019). *The effectiveness of requirements prioritization techniques for a medium to large number of requirements: a systematic literature review*

[15] QualcommTechnologies, INC. (2017). *Pointer authentication on ARMv8.3. [online] Available at: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf*

[16] riscv.org. (2020). *[online] Available at: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf [Accessed 8 April 2020].*

[17] Sayeed, Sarwar &Marco-Gisbert, Hector &Ripoll, Ismael &Birch, Miriam. (2019). *Control-Flow Integrity: Attacks and Protections.* Applied Sciences. 9. 4229. 10.3390/app9204229.

[18] Leander Seidlitz. (2019). *RISC-V ISA Extension for Control Flow Integrity*

[19] Schilling, Robert &Werner, Mario &Nasahl, Pascal &Mangard, Stefan. (2018). *Pointing in the Right Direction - Securing Memory Accesses in a Faulty World.* 10.1145/3274694.3274728.

[20] Stanford University. (2020). *[online] Available at: https://crypto.stanford.edu/cs155old/cs155-spring16/lectures/02-ctrl-hijacking.pdf [Accessed 8 April 2020].*

[21] Wetzels, J. (2017). *Internet of Pwnable Things: Challenges in Embedded Binary Security.* login Usenix Mag., 42.

Appendix

Appendix A

Requirements

A.1 Requirements List

Unique ID	Description	Pre-requisites	Type	Priority (MoSCoW)
HW-1	Core design is based on a 32bit arch	X	N	M
HW-2	RISCV core written in Verilog to be used	HW-1	N	M
ISS-1	RISCV ISA instruction set simulator to be used	HW-1	N	S
SW-1	Compiler supports the new instructions	X	F	M
ISS-2	ISS code supports shadow stack store	X	F	M
ISS-3	ISS code supports shadow stack load	X	F	M
ISS-4	ISS code supports thread selection via thread select opcode	X	F	S

ISS-5	ISS code supports an internal stack for shadow stack instruction use	X	F	M
ISS-6	ISS code supports multiple internal stacks for shadow stack instruction use via multiple threads	ISS-2, ISS-3, ISS-4, ISS-5	F	S
ISS-7	ISS code supports an interrupt signal for shadow stack authentication failure	ISS-3	F	S
HW-3	Core supports shadow stack store instruction	HW-2, SW-2, SW-11	F	M
HW-4	Core supports shadow stack load instruction	HW-2, SW-3, SW-11	F	M
HW-5	Core supports shadow stack thread selection instruction	HW-2, SW-4, SW-9	F	S
HW-6	Core supports an internal shadow stack for shadow stack instructions	HW-2, HW-3, HW-4	F	M
HW-7	Core supports multiple internal shadow stacks for shadow stack Instruction use via multiple threads	HW-2, HW-6	F	S
HW-8	Core features a dedicated interrupt signal for when shadow stack verification fails	HW-2, HW-3, HW-4, SW-11	F	M

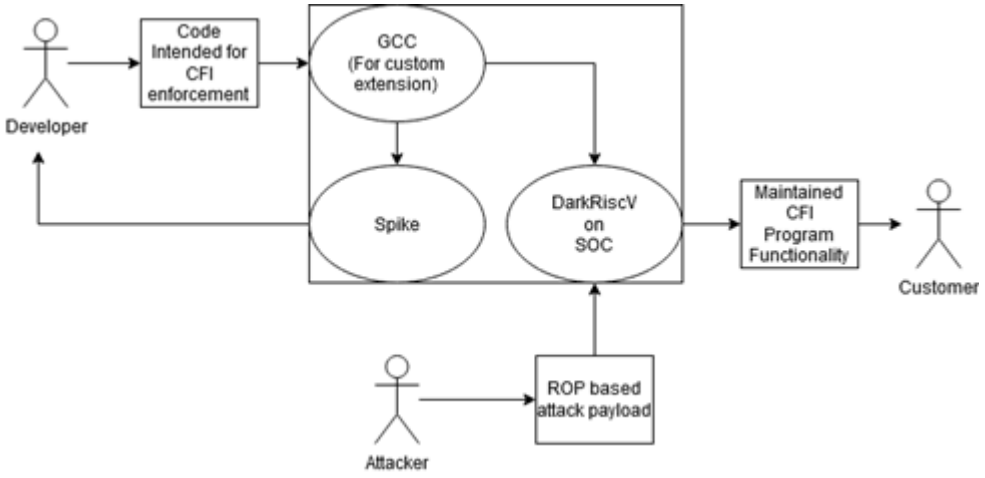
HW-9	The core design can only be generated to include a shadow stack component if the definition is set.	HW-6	N	S
HW-10	The core design will run correctly if the shadow stack definition is not set.	HW-9	N	S
SW-2	Assembler supports shadow stack store opcode	X	F	M
SW-3	Assembler supports shadow stack load opcode	X	F	M
SW-4	Assembler supports thread selection opcode	X	F	S
SW-5	Compiler produces shadow stack store opcode	SW-1	F	M
SW-6	Compiler produces shadow stack store opcode at the end of function call prologue.	SW-5	F	M
SW-7	Compiler produces shadow stack load opcode	SW-1	F	M
SW-8	Compiler produces shadow stack load opcode at the beginning of function call epilogue.	SW-7	F	M
SW-9	Test software for core supports use of thread selection opcode	HW-5	F	S

SW-10	Test software for core supports use of Shadow Stack load opcode	HW-4	F	M
SW-11	Test software for core supports use of Shadow Stack store opcode	HW-3	F	M
SW-12	Test software for core supports use of scheduler	X	F	S
SW-13	Test software for core supports interrupt service routine for shadow stack interrupt signal	HW-3, HW-4	F	M
SW-14	Test software for instruction set simulator supports use of shadow stack load opcode	ISS-3	F	S
SW-15	Test software for instruction set simulator supports use of shadow stack store opcode	ISS-2	F	S
SW-16	Test software for instruction set simulator supports use of shadow stack thread selection opcode	ISS-4	F	S
SW-17	Test software for instruction set simulator supports use of a scheduler	ISS-4, ISS-6	F	S

SW-18	Test software for instruction set simulator supports interrupt service routine for shadow stack interrupt signal	ISS-2, ISS-3, ISS-5	F	S
ISS-8	Instruction set simulator test software scheduler runs multiple test programs as threads with no issue.	SW-15, SW-16, SW-17	N	C
ISS-9	shadow stack store opcode works with multiple threads on instruction set simulator test software via use of a scheduler	SW-2, ISS-5	N	S
ISS-10	shadow stack load opcode works with multiple threads on instruction set simulator test software via use of a scheduler	SW-3, ISS-5	N	S
ISS-11	ISS shadow stack module should not print internal prints until told to do so with a verbose flag.	ISS-1	N	S
SW-19	Demo code for performing a stack return address targeted attack(ROP) developed for instruction set simulator	SW-14	F	S

SW-20	Demo code for performing a stack return address targeted attack(ROP) developed for the core design	SW-11	F	M
SW-21	Demo code successfully exploits system when scheduler/demo code isn't compiled with shadow stack support.(ISS)	SW-19	F	S
SW-22	Demo code fails to exploit system when scheduler/demo code is compiled with shadow stack support.(ISS)	ISS-2, ISS-3, ISS-5, SW-17	F	S
SW-23	Demo code successfully exploits system when scheduler/demo code isn't compiled with shadow stack support.(core design)	SW-20	F	M
SW-24	Demo code fails to exploit system when scheduler/demo code is compiled with shadow stack support.(core design)	SW-20, HW-3, HW-4, HW-5, HW-8	F	M
SW-25	The compiler should only insert shadow stack instructions when the corresponding extension name is set in the arch.	SW-6, SW-8	N	C

A.2 Use Case Diagram



A.3 Use Case Table

Goal	Generate a program or OS with CFI enforcement using a shadow stack which maintains CFI against ROP based attacks.
Primary Actor	Developer
Scope	Software + Compiler + ISS and DarkRiscv core
Level	Developer
Precondition	Written code ready, toolchain environment setup, machine arch includes custom extension name.
Success end condition	Machine code generated contains correct placement of shadow stack instructions, CFI is maintained against vulnerability exploited by an attacker.
Failure end condition	Machine code generated does not contain shadow stack instructions or they are not inserted correctly, CFI breaks due to incorrect insertion of instructions resulting in the attacker's execution of their payload.
Trigger	Developer compiles their code with the correct GCC machine arch and loads it onto the SOC.

Main Success Scenario	1. Developer writes operating system or program to run on an existing OS for the RISCV architecture. 2. Developer compilers program/OS with custom extension set in the machine arch flag. 3. Developer programs the software onto a SOC based on DarkRiscv featuring the shadow stack design. 4. Developer offers the product to the user. 5. User is protected against ROP based attacks, or the system they use is resilient to return address corruption.
Error Scenario	• 2.A Compiler fails to generate/insert shadow stack instructions in programs output machine code. • 5.A The user experiences return address corruption and crashes when exposed to ROP based exploits.
Variations	3. Developer tests program on Spike instruction set simulator

Appendix B

Designs

B.1 ISS Shadow Stack UML

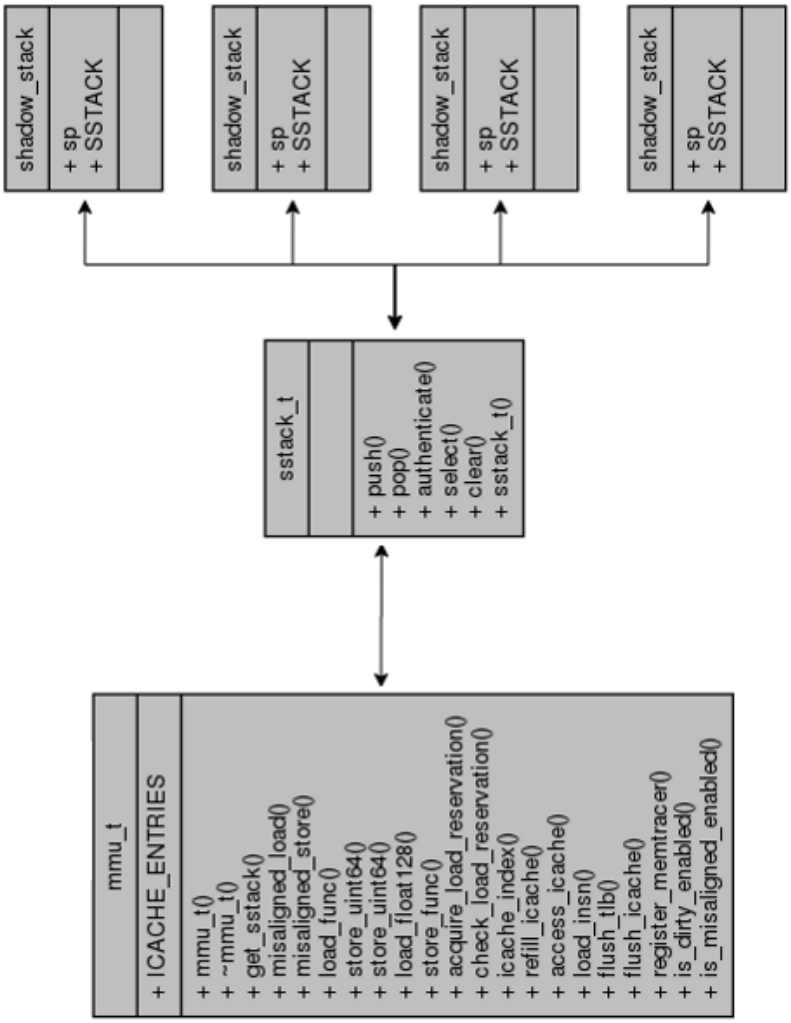


Figure B.1: ISS Design: ISS UML Class design

B.2 Shadow Stack Indexing Design

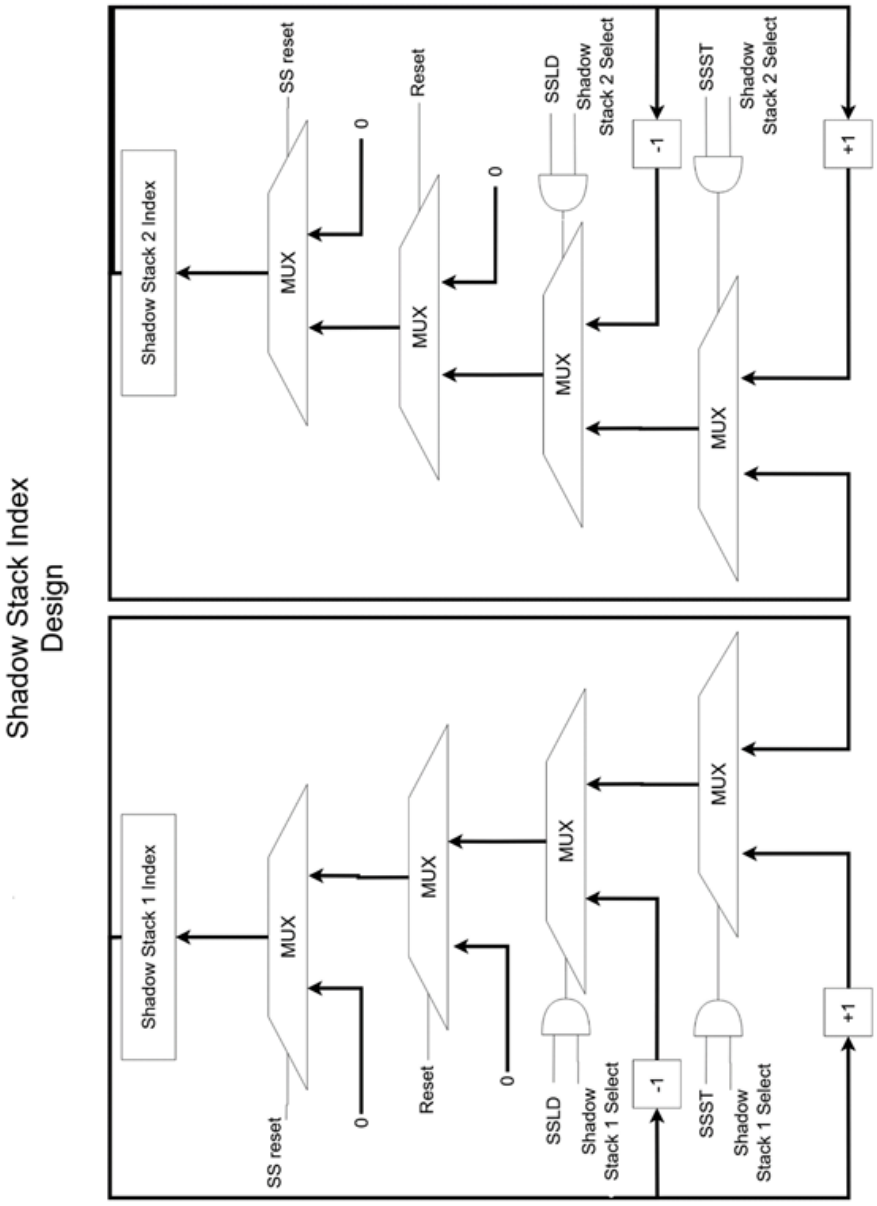


Figure B.2: Core Design: Shadow Stack Indexing Design

B.3 Shadow Stack Load Opcode Design

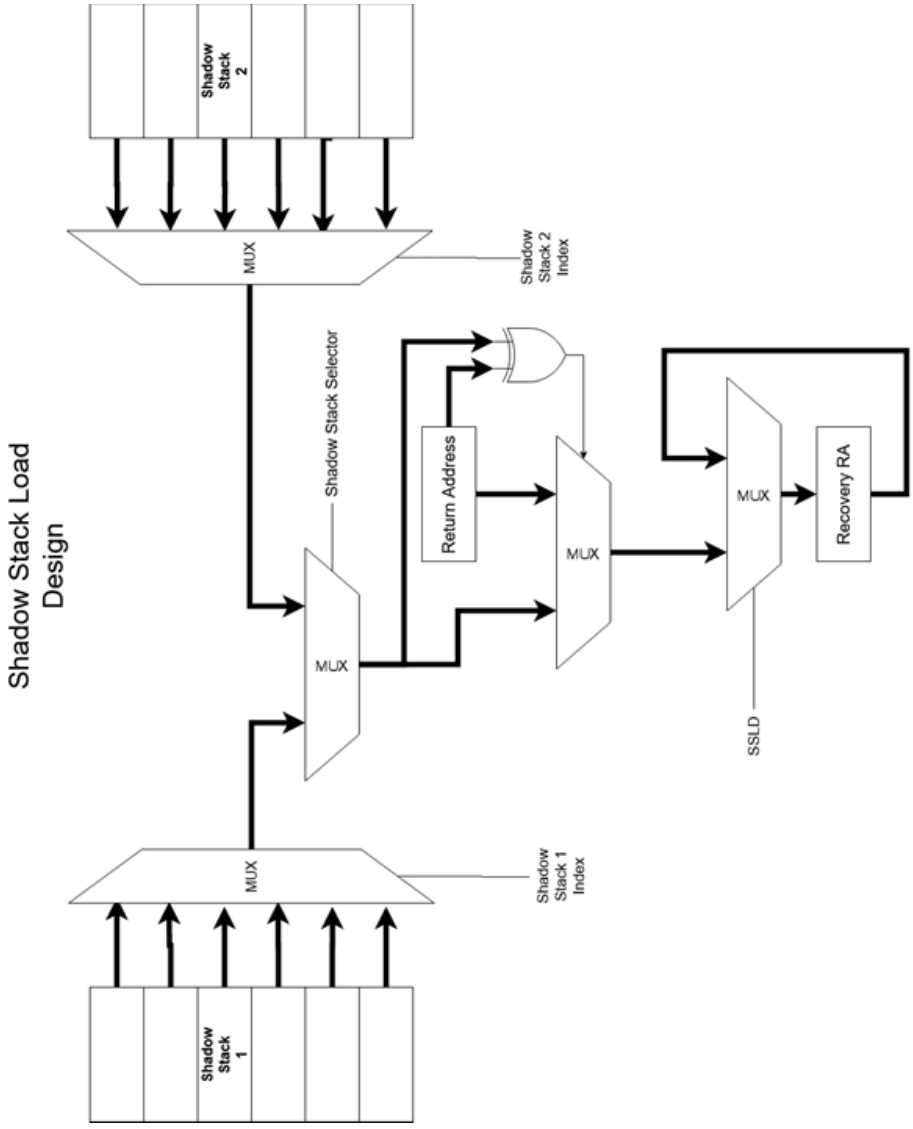


Figure B.3: Core Design: Shadow Stack Load Opcode Design

B.4 Shadow Stack Store Opcode Design

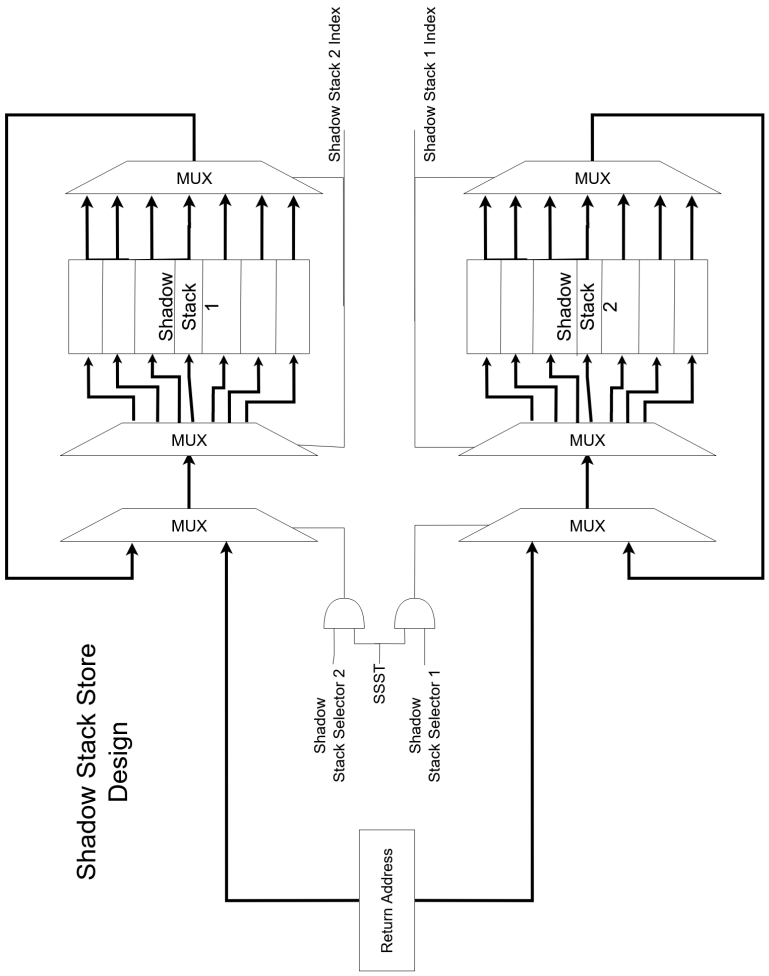


Figure B.4: Core Design: Shadow Stack Store Opcode Design

B.5 Shadow Stack Reset Signal Design

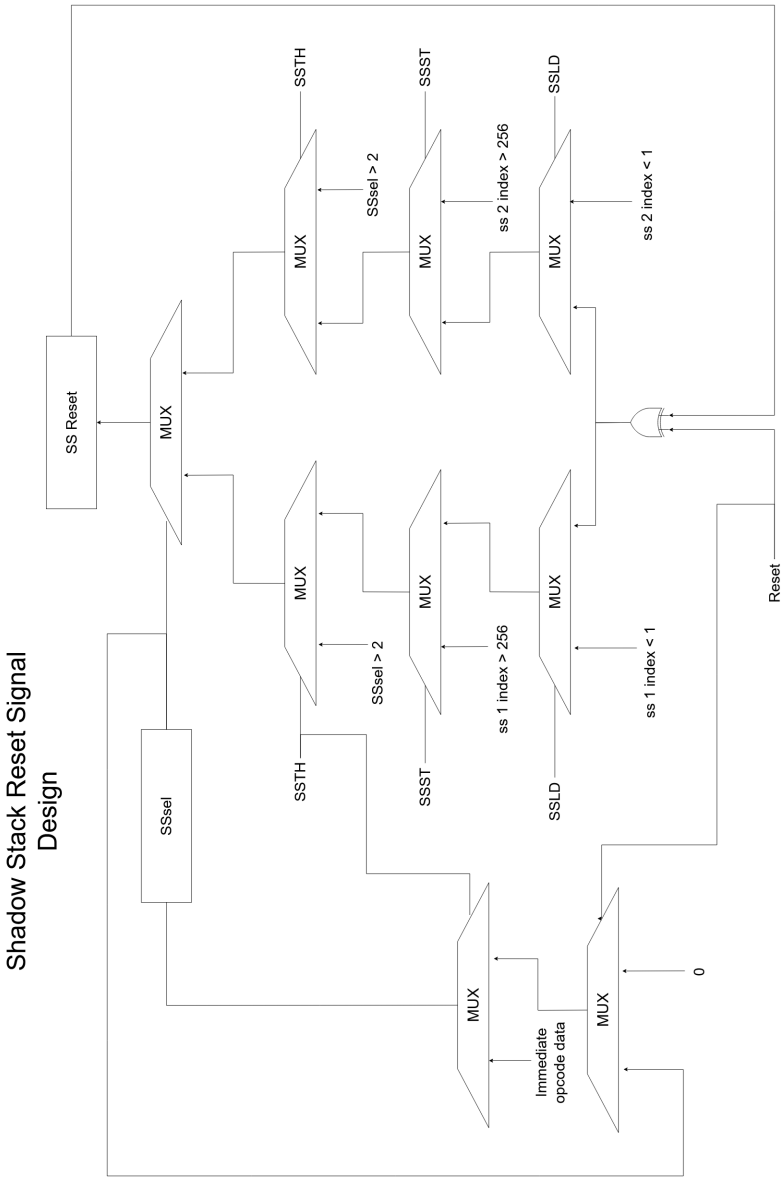


Figure B.5: Core Design: Shadow Stack Reset Signal Design

Appendix C

Acceptance Tests

C.1 Accpetance Test List

Test Name	BB-01
Target requirements	HW-1, HW-2
Outline	Core on FPGA hardware boots
Pre-requisites	Have FPGA board connected to the PC and the core design built and ready to program the bit-stream.

Step	**Action**	**Expected Observation**
1	Attach FPGA to pc.	Should hear it connect
2	Open Arduino serial monitor and select the correct com port with baud set to 115200	The com port exists and the terminal is blank
3	Program the bit stream on Vivado	The terminal displays the correct boot message.

Test Name	BB-02
Target requirements	HW-1, HW-2
Outline	Core on FPGA hardware responds to serial input with serial output
Pre-requisites	Passed test BB-01 and a serial monitor listening with correct COM port/baud rate selected.

Step	**Action**	**Expected Observation**
1	Perform BB-01 steps	Serial output is seen on monitor
2	Type “help” into the serial monitor and hit send	Serial output responds to input by printing a CMD list

Test Name	BB-03
Target requirements	HW-3, HW-4, HW-5, HW-6, HW-7, SW-2, SW-3, SW-4, SW-9, SW-10, SW-11
Outline	Core on FPGA hardware serial handler parses test commands correctly
Pre-requisites	Test BB-02 passes

Step	**Action**	**Expected Observation**
1	Perform BB-02 steps	Serial output is seen and we have interaction on serial
2	Type “sstest” in serial monitor	Serial output replies with “sstests completed!”

Test Name	BB-04
Target requirements	SW-12
Outline	Core on FPGA hardware serial handler shows that more than 1 thread is running
Pre-requisites	Test BB-01 passes, the core has been built and the board is programmed with a multithread supporting core.

Step	**Action**	**Expected Observation**
1	Perform BB-01 steps	We see the boot message print the number of threads running.

Test Name	BB-05
Target requirements	HW-3, HW-4, HW-6, HW-8, SW-2, SW-3, SW-10, SW-11, SW-13
Outline	Core on FPGA hardware does not reset when told to run a return address corruption test.
Pre-requisites	Core built with test program and board programmed with it.

Step	**Action**	**Expected Observation**
1	Type and send “sstest corrupt” on serial to board	“test passed!” is printed

Test Name	BB-06
Target requirements	HW-3, HW-4, HW-5, HW-6, HW-7, SW-2, SW-3, SW-4, SW-9, SW-10, SW-11, SW-12
Outline	Core on FPGA hardware does not reset when told to run multiple test programs on diff threads.
Pre-requisites	Test BB-05 passed, scheduler implemented in the test program and programmed on the board.

Step	**Action**	**Expected Observation**
1	Type and send "sstest sched" on serial to board	Led 0/1 alternate on/off on the board

Test Name	BB-07
Target requirements	HW-3, HW-4, HW-5, HW-6, HW-7, SW-2, SW-3, SW-4, SW-5, SW-6, SW-7, SW-8, SW-9, SW-10, SW-11, SW-12, SW-20, SW-23, SW-24
Outline	Core on FPGA hardware runs demo program without crashing
Pre-requisites	BB-01 up to BB-06 all pass, board programed with core design using Demo program binary.

Step	**Action**	**Expected Observation**
1	Type and send "start" on serial to board	Serial prints all test results up until it says, "Done!"

Test Name	BB-08
Target requirements	ISS-1, ISS-2, ISS-3, ISS-4, ISS-5, ISS-6, SW-2, SW-3, SW-4, SW-14, SW-15, SW-16, SW-17, ISS-8, ISS-9, ISS-10
Outline	ISS runs multiple tests programs on different simulated threads.
Pre-requisites	ISS tested with shadow stack instructions and passes, test program for ISS written/compiled with use of a scheduler.

Step	**Action**	**Expected Observation**
1	Run ISS on CMD line with the corresponding test program.	CMD line output displays each program with each allocated thread number, and no error messages seen.

Test Name	BB-09
Target requirements	ISS-1, ISS-2, ISS-3, ISS-5, SW-2, SW-3, SW-14, SW-15, SW-18
Outline	ISS prints "recovered!" and doesn't crash when running return address corruption test program
Pre-requisites	ISS tested with shadow stack instructions and passes, test program with return address corruption code compiled.

Step	**Action**	**Expected Observation**
1	Run ISS on CMD line with the corresponding test program.	ISS prints "recovered!" on prompt window and completes the rest of the program without crashing.

Test Name	BB-10
Target requirements	ISS-1, ISS-2, ISS-3, ISS-5, ISS-11, SW-2, SW-3, SW-14, SW-15
Outline	ISS doesn't crash when running a normal test program
Pre-requisites	ISS tested with shadow stack instructions and passes, and a test program that uses shadow stack instructions compiled.

Step	**Action**	**Expected Observation**
1	Run ISS on CMD line with the corresponding test program using –vss for verbose printing.	No error message or crash is seen on CMD prompt, but shadow stack internal prints are seen.

Test Name	BB-11
Target requirements	ISS-1, ISS-2, ISS-3, ISS-4, ISS-5, ISS-6, SW-2, SW-3, SW-4, SW-5, SW-6, SW-7, SW-8, ISS-9, ISS-10, SW-19, SW-21, SW-22
Outline	ISS runs demo program without crashing
Pre-requisites	Tests BB-08 to BB-10 all pass and demo code for ISS compiled.

Step	**Action**	**Expected Observation**
1	Run ISS on CMD line with the ISS demo program.	No crash is seen and "Done!" is seen at the end of the demo on CMD prompt.

Test Name	WB-01
Target requirements	SW-2, SW-3, SW-4, SW-9, SW-10, SW-11
Outline	Linker accepts custom instructions via inline assembly
Pre-requisites	Shadow stack store, load, and thread select instructions implemented in linker code. Test program source ready to compile.

Step	**Action**	**Expected Observation**
1	Run make on CMD line with source code that uses shadow stack instructions via inline assembly.	The program builds without error.

Test Name	WB-02
Target requirements	SW-2, SW-3, SW-4, SW-9, SW-10, SW-11
Outline	Linker generates the correct binary for the custom instruction
Pre-requisites	Test WB-01 passes

Step	**Action**	**Expected Observation**
1	Run ricv32 objdump on the test program elf and passing it arg –d and pipe its output to a text file with `>program_dis.txt`	The CMD runs without error
2	Open text file in notepad and search for SSLD, SSST, and SSTH.	The file shows all three mnemonics in the file.
3	Take the binary for each opcode and compare binary to their designs	The hex values match each of the opcodes expected binary representations.

Test Name	WB-03
Target requirements	SW-2, SW-5, SW-6, SW-25
Outline	Compiler inserts SSST correctly in prologue
Pre-requisites	Compiler source modified for SSST, and WB-02 passes

Step	**Action**	**Expected Observation**
1	Run ricv32 objdump on the test program elf and passing it arg –d and pipe its output to a text file with `>program_dis.txt`	The CMD runs without error
2	Open text file in notepad and search for SSST.	SSST instruction has been correctly inserted in the start of the prologue of each normal function.

Test Name	WB-04
Target requirements	SW-3, SW-7, SW-8, SW-25
Outline	Compiler inserts SSLD correctly in epilogue
Pre-requisites	Compiler source modified for SSLD, and WB-02 passes

Step	**Action**	**Expected Observation**
1	Run ricv32 objdump on the test program elf and passing it arg –d and pipe its output to a text file with `>program_dis.txt`	The CMD runs without error.
2	Open text file in notepad and search for SSLD.	SSLD instruction has been correctly inserted in the epilogue of a normal function just before the “ret” instruction.

Test Name	WB-05
Target requirements	HW-3, HW-4, HW-6, HW-8, SW-2, SW-3, SW-10, SW-11
Outline	Core recovers return address when authentication is passed an incorrect return address.
Pre-requisites	Core supports SSST/SSLD along with a single shadow stack. Recovery return address is implemented. Test program compiled and loaded on in the core.

Step	Action	Expected Observation
1	Type and send “sstest corrupt” on serial to board	No reset is detected and “test passed!” is seen.

Test Name	WB-06
Target requirements	HW-3, HW-4, HW-6, HW-8, SW-2, SW-3, SW-10, SW-11
Outline	Core does not reset/crash when multiple SSST set and matched with SSLD, for the purpose of testing multiple stack frames being pushed and popped.
Pre-requisites	WB-05 passes, test program with recursive code compiled and loaded on the core.

Step	Action	Expected Observation
1	Type and send “sstest recu” on serial to board	“sstests completed!” printed on serial.

Test Name	WB-07
Target requirements	ISS-1, ISS-2, SW-2, SW-15
Outline	ISS decodes SSST instruction
Pre-requisites	WB-02 passes and test program compiled.

Step	Action	Expected Observation
1	Run ISS on CMD line with the corresponding test program.	No error message is seen on CMD prompt.

Test Name	WB-08
Target requirements	ISS-1, ISS-3, SW-3, SW-14
Outline	ISS decodes SSLD instruction
Pre-requisites	WB-02 passes and test program compiled.

Step	**Action**	**Expected Observation**
1	Run ISS on CMD line with the corresponding test program.	No error message is seen on CMD prompt.

Test Name	WB-09
Target requirements	ISS-1, ISS-4, SW-4, SW-16
Outline	ISS decodes SSTH instruction
Pre-requisites	WB-02 passes and test program with scheduler compiled.

Step	**Action**	**Expected Observation**
1	Run ISS on CMD line with the corresponding test program.	No error message is seen on CMD prompt.

Test Name	WB-10
Target requirements	ISS-1, ISS-2, ISS-3, ISS-5, ISS-7, SW-2, SW-3, SW-14, SW-15, SW-25
Outline	ISS recovers return address when authentication is passed an incorrect return address.
Pre-requisites	ISS code supports SSST/SSLD along with a single shadow stack. Recovery return address is implemented. Test program compiled.

Step	**Action**	**Expected Observation**
1	Run ISS on CMD line with the corresponding test program.	No crash or error message is printed, the program runs to completion.

Test Name	WB-11
Target requirements	ISS-1, ISS-2, ISS-3, ISS-5, ISS-7, SW-2, SW-3, SW-14, SW-15
Outline	ISS does not reset/crash when multiple SSST set and matched with SSLD, for the purpose of testing multiple stack frames being pushed and popped.
Pre-requisites	WB-10 passes, and a recursive test program compiled.

Step	**Action**	**Expected Observation**
1	Run ISS on CMD line with the corresponding test program.	No error message or crash is seen on CMD prompt.

Test Name	WB-12
Target requirements	ISS-1, ISS-2, ISS-3, ISS-4, ISS-5, ISS-6, SW-2, SW-3, SW-4, SW-14, SW-15, SW-16
Outline	ISS scheduler test program sets thread index for shadow stack correctly
Pre-requisites	WB-02 passes, ISS supports multiple shadow stacks, and a test program that uses the SSTH instruction has been compiled.

Step	**Action**	**Expected Observation**
1	Run ISS on CMD line with the corresponding test program.	No error message or crash is seen on CMD prompt.

Test Name	WB-13
Target requirements	HW-3, HW-4, HW-5, HW-6, HW-7, SW-2, SW-3, SW-4, SW-9, SW-10, SW-11
Outline	Core scheduler test program sets thread index for shadow stack correctly
Pre-requisites	Core that supports SSTH and multiple shadow stacks has been loaded along with the test program. Core modified to map index to an I/O

Step	**Action**	**Expected Observation**
1	Type and send “sstest sched” on serial to board	Led 0/1 alternate on/off on the board

Test Name	WB-14
Target requirements	HW-9, HW-10
Outline	Core design encounters illegal instruction behavior if SSTACK defines has not been set.
Pre-requisites	Full Shadow Stack supporting core is synthesized and implemented on FPGA hardware without setting the SSTACK definds in config.vh

Step	**Action**	**Expected Observation**
1	Comment out “‘define SSTACK” in config.vh	
2	Synth, implement, generate bitstream, and program the board.	All succeed, and board is programmed. Responds normally on uart.
3	Enter “sstest st”	The board will reset. Meaning ssst was detected as an illegal instruction since the design wont support shadow stack opcodes.

C.2 Acceptance Test Tracability Matrix

Req \ Test	BB-01	BB-02	BB-03	BB-04	BB-05	BB-06	BB-07	BB-08	BB-09	BB-10	BB-11	WB-01	WB-02	WB-03	WB-04	WB-05	WB-06	WB-07	WB-08	WB-09	WB-10	WB-11	WB-12	WB-13	WB-14
HW-1	X	X																							
HW-2	X	X																							
HW-3			X		X	X	X									X	X							X	
HW-4			X		X	X	X									X	X							X	
HW-5			X			X	X																	X	
HW-6			X		X	X	X									X	X							X	
HW-7			X			X	X																	X	
HW-8					X											X	X								
HW-9																									X
HW-10																									X
ISS-1								X	X	X	X							X	X	X	X	X	X		
ISS-2								X	X	X	X							X			X	X	X		
ISS-3								X	X	X	X								X		X	X	X		
ISS-4								X			X									X			X		

Req \ Test	BB-01	BB-02	BB-03	BB-04	BB-05	BB-06	BB-07	BB-08	BB-09	BB-10	BB-11	WB-01	WB-02	WB-03	WB-04	WB-05	WB-06	WB-07	WB-08	WB-09	WB-10	WB-11	WB-12	WB-13	WB-14
ISS-5								X	X	X	X										X	X	X		
ISS-6								X			X												X		
ISS-7																					X	X			
ISS-8								X																	
ISS-9								X			X														
ISS-10								X			X														
ISS-11										X															
SW-1				X																					
SW-2			X	X	X	X	X	X	X	X	X	X	X	X		X	X	X			X	X	X	X	
SW-3			X		X	X	X	X	X	X	X	X	X			X	X		X		X	X	X		
SW-4			X			X	X	X			X	X	X							X			X	X	
SW-5							X				X			X											
SW-6							X				X			X											
SW-7							X				X				X										
SW-8							X				X				X										

Req \ Test	BB-01	BB-02	BB-03	BB-04	BB-05	BB-06	BB-07	BB-08	BB-09	BB-10	BB-11	WB-01	WB-02	WB-03	WB-04	WB-05	WB-06	WB-07	WB-08	WB-09	WB-10	WB-11	WB-12	WB-13	WB-14
SW-9			X			X	X					X	X											X	
SW-10			X		X	X	X					X	X			X	X							X	
SW-11			X		X	X	X					X	X			X	X							X	
SW-12						X	X																		
SW-13					X																				
SW-14								X	X	X									X		X	X	X		
SW-15								X	X	X								X			X	X	X		
SW-16								X											X	X			X		
SW-17								X																	
SW-18									X																
SW-19											X														
SW-20							X																		
SW-21											X														
SW-22											X														
SW-23							X																		

Req \ Test	SW-24	SW-25
BB-01		
BB-02		
BB-03		
BB-04		
BB-05		
BB-06		
BB-07	X	
BB-08		
BB-09		
BB-10		
BB-11		
WB-01		
WB-02		
WB-03		X
WB-04		X
WB-05		
WB-06		
WB-07		
WB-08		
WB-09		
WB-10		
WB-11		
WB-12		
WB-13		
WB-14		

C.3 Acceptance Test Results

Test Name	Step #	Expected Observation	Status
BB-01	1	Should hear it connect	PASS
	2	The com port exists and the terminal is blank	PASS
	3	The terminal displays the correct boot message.	PASS
BB-02	1	Serial output is seen on monitor	PASS
	2	Serial output responds to input by printing a CMD list	PASS
BB-03	1	Serial output is seen and we have interaction on serial	PASS
	2	Serial output replies with "sstests completed!"	PASS
BB-04	1	We see the boot message print the number of threads running.	TO BE TESTED
BB-05	1	"test passed!" is printed	TO BE TESTED
BB-06	1	Led 0/1 alternate on/off on the board	PASS
BB-07	1	Serial prints all test results up until it says, "Done!"	TO BE TESTED
BB-08	1	CMD line output displays each program with each allocated thread number, and no error messages seen.	PASS
BB-09	1	ISS prints "recovered!" on prompt window and completes the rest of the program without crashing.	PASS

BB-10	1	No error message or crash is seen on CMD prompt, but shadow stack internal prints are seen.	PASS
BB-11	1	No crash is seen and "Done!" is seen at the end of the demo on CMD prompt.	PASS
WB-01	1	The program builds without error.	PASS
WB-02	1	The CMD runs without error	PASS
	2	The file shows all three mnemonics in the file.	PASS
	3	The hex values match each of the op-codes expected binary representations.	PASS
WB-03	1	The CMD runs without error	PASS
	2	SSST instruction has been correctly inserted in the start of the prologue of each normal function.	PASS
WB-04	1	The CMD runs without error	PASS
	2	SSLD instruction has been correctly inserted in the epilogue of a normal function just before the "ret" instruction.	PASS
WB-05	1	No reset is detected and "test passed!" is seen.	PASS
WB-06	1	"sstests completed!" printed on serial.	PASS
WB-07	1	No error message is seen on CMD prompt.	PASS
WB-08	1	No error message is seen on CMD prompt.	PASS
WB-09	1	No error message is seen on CMD prompt.	PASS

WB-10	1	No crash or error message is printed, the program runs to completion.	PASS
WB-11	1	No error message or crash is seen on CMD prompt.	PASS
WB-12	1	No error message or crash is seen on CMD prompt.	PASS
WB-13	1	Led 0/1 alternate on/off on the board	PASS
WB-14	1	N/A	PASS
	2	All succeed, and board is programmed. Responds normally on uart.	PASS
	3	The board will reset. Meaning "ssst" was detected as an illegal instruction since the design won't support shadow stack opcodes.	PASS