NG RISC-V ASIC Design

SCALER: Side Channel Leakage Reduction in RISC-V Core

Sponsored by Northrop Grumman Corporation

Group 13 Authors

Anna Craig	Alexander DeFalco	Joshua Joseph	Daniel Odi	Alyssa Pinnock
Electrical Engineering	*Computer Engineering*	*Computer Engineering*	*Computer Engineering*	*Computer Engineering*

Review Committee

Dr. Chung Yong Chan	UCF ECE	Advisor/Professor
Dr. Mike Borowczak	UCF ECE	Advisor/Professor
Dr. Fan Yao	UCF ECE	Associate Professor
Brian McNulty	NG Sponsor	Program Sponsor
Steve Leonard	NG Sponsor	Program Manager
Marlon J Fuentes	NG Sponsor	Principal Investigator
Aaron Lingerfelt	NG Sponsor	Technical Lead
Alberto Reategui	NG Sponsor	Technical Lead

Chapter 2 - Project Description

2.1 Background & Motivation

2.1.1 ASIC Design Process with EDA Tools and Cadence

EDA tools play a critical role in the ASIC development process. These tools transform high-level hardware descriptions to mitigate Side Channel Attacks into a chip implementation, proving that it is a hardware-ready design that can be fabricated. Given the main goal of this project is to gain familiarity with ASIC Design processes, it is imperative to utilize EDA tools such as the Cadence Design Suite.

Cadence is an automated flow tool that streamlines the process of converting behavioral-level description of your circuit design into GDS format (which is used to fabricate your circuit on silicon). This project will require us to use various tools in the Cadence suite, including Cadence's Genus, Xcelium, Simvision, and Innovus. These tools are able to automate steps in the design process, as the tools work significantly faster and more efficient than hand-placing gates. Cadence Xcelium is used to simulate the design along with a testbench as a way of debugging and ensuring the functionality expressed in the RTL code is the same as intended. Once waveforms are generated through simulations, they can be pulled up on Simivision, which is Cadence’s waveform viewer. In introductory digital logic classes, testbenches are created for very small designs so it's possible to test all the inputs and possible states very easily, but as you increase the complexity and size of the design in the case of this project simple functional verification will not hold. Thus, the team will be verifying the design using the Universal Verification Methodology (UVM) which is a verification standard to test digital logic designs and confirm that it is working according to the specifications. In the context of Side Channel Analysis Attack, through verification techniques we can confidently confirm that RTL descriptions used to implement algorithms that are confirmed to mitigate Side Channels are providing the correct output and functionality.

Once verified, the design is taken to Cadence Genus, which synthesizes the design from RTL into a netlist that contains the logical components making up the circuit. There

are three steps to synthesis: syn_generic, syn_map, and syn_opt. These steps are responsible for outputting the initial design netlist, mapping the cells in the netlist to a standard cell library specified by your Process Design Kit (PDK), and optimizing the synthesized netlist to meet timing and area constraints. In the context of Side Channel Analysis Attacks, the synthesis process confirms that the RTL can successfully be converted into an equivalent gate representation. If synthesis is unsuccessful, such as in the case where the RTL written is unsynthesizable, errors will pop in flow logs showing there are implementation errors present.

Once complete, the design is taken to Innovus, where you floorplan the design (set size of chip, pin placements, and macro placements), place gates, and route gates for the design. Innovus automates the place-and-route process, with the assistance of the user to validate the result and manually fix any design rule violations. These design rule violations will give insight into problems of the layout to the chip, which might catch issues related to our implementation of these Side Channel Analysis Attacks.

Overall, Cadence is required to help to mitigate side channel attacks in any microarchitecture that is developed in this project. Its set of tools allow for designs to verify implementations of functions in circuits that mitigate Side Channel Analysis Attacks through simulations are correct, synthesizes the HDL description into gates and catches unsynthesizable code, and helps with place and route of the circuit in order to prepare it for tapeout. Without this, using classical methods like hand placing is much more prone to error, more inefficient, and much more difficult to accomplish.

2.1.2 Importance of Hardware Security

With the increasing presence of IoT and connectivity in our electronics, more resources have been allocated to securing our devices from malicious entities. As a result, we have made many impressive advancements in the field of cybersecurity, yielding substantial results. Although this is great, more emphasis needs to be put on defending hardware-focused attacks as electronics are just as, if not more, susceptible to them. These attacks are very detrimental because once the physical components are compromised, the entire system becomes vulnerable or even unusable. Once the attacker is successful, they are usually able to bypass any kind of software security already in place completely. Successful hardware attacks can cause things like stolen IP, identity theft, and exposure of customer data. It gets even more serious once you start to consider the implications in the Department of Defense (DoD), where each of their devices is responsible for saving lives and can cost tens of millions of dollars. Defending these devices has proven to be extremely difficult due to their sheer complexity, so many talented engineers and developers are needed to secure these technologies. This is especially true when considering their enemies have an abundance of time, money, and techniques to find and exploit these weaknesses. The issue is further amplified by the fact that the attacker only needs one vulnerability to be effective, while the defender needs to be conscious of all the possibilities. Once compromised, the effects can be catastrophic, as it is very difficult to find out that information has been leaked or make changes once the devices have been released.

2.1.3 Types of Hardware Attacks **[5]**

Unlike how software attackers need access to a device's software/network, their hardware counterparts usually have direct access to the chip or signal wires. These attacks are defined as. In order to carry out their disruption, they use a variety of tools to monitor and tamper with electronics, like electromagnetic probes and logic analyzers to view signals, as well as heat guns to remove components. Physical attacks can be classified into two main types: passive and active. Passive attacks involve the malicious entity gaining access to information that the creator did not intend to be read, while active ones attempt to alter the normal functions of a system. Hardware attacks can be further grouped as invasive or non-invasive, with the former being when the attacker has access to the inside of the chip and the latter utilizing the device's peripheral characteristics, leaving very little tamper evidence.

2.1.4 Side-Channel Attacks **[13]**

One of the most common methods of attacking hardware is the side-channel attack, where the attacker exploits unintended information leakage. The sources of these leakages can originate from a chip's power usage, temperature, or even the timing of different instructions. These are very tricky to defend because they are taking advantage of the normal function of the system itself. A common method for side channel attacks involves using time-driven attacks to predict a device's control flow and functionality. For example, a malicious attacker can observe the time it takes for instructions after branches to determine whether the long or short execution path was taken. The aggressor is also able to track how long the chip takes for memory access to determine whether there was a cache hit or miss, allowing them to infer what data is being used.

On top of time-driven side channel attacks, attackers can analyze the leakage power associated with the different instructions of a circuit to infer its functions and other sensitive information. A basic form of power attacks is the Simple Power Analysis (SPA), where the attacker can directly observe the device's power consumption over time to interpret data. Although this is a much simpler route of attack, it can predict things like encryption keys for different algorithms or whether the device is doing addition or multiplication. A step above that is Differential Power Analysis (DPA), where multiple traces are collected and statistically analyzed to extract hidden patterns. Although DPA is more computationally expensive, this method is especially potent because it has high resistance against random noise and other preventative techniques. While SPA takes advantage of lower-level defenses with minimal protection, DPA uses statistical methods like correlation to get past many countermeasures, treating them as mere delays instead of prevention.

2.1.5 Spectre and Meltdown [5][14]

To maintain speed, modern CPUs require efficient methods for handling branches. Speculative execution allows a CPU to predict the outcome of a branch and start the next instruction before the result of the branch is known. If the guess is correct, the results of the instruction can be updated in memory; if it isn't, the result is discarded, and the processor can roll back to the correct instruction. This allows processors to maintain speed while minimizing hardware idle time, unlike other methods like pipeline stalling.

Although speculative execution is beneficial for overall CPU performance, it can cause some security vulnerabilities

One of the main attacks for targeting speculative execution is Spectre. This method focuses on the branch prediction mechanisms present in the CPU. Spectre takes advantage of the fact that even though certain instructions can be discarded, there are still traces left behind, like in the cache, for example. The malicious entities taking this route of attack can train the branch predictor to take the route that accesses sensitive information, allowing the attacker to then use techniques like Flush+Load to recover the data. Flush and Load refers to the process of an attacker removing a line from the cache(flush) only to execute the same instruction again. Once the instruction is done, they can check how long it takes to reference the line of memory to see if it ends up back in the cache(reload). This allows the attacker to predict things like cryptographic keys or passwords.

Attackers can also opt for the Meltdown attack, which has proven to be just as devastating. While Spectre focuses on manipulating the branch predictor to go down the desired path, Meltdown is based on the fact that CPUs can speculatively execute instructions before permissions are enforced. This attack allows the malicious entity to effectively bypass hardware-enforced memory permission checks to read sensitive information. Combined with side channel attacks like Prime+Probe and the Flush+Load mentioned previously, Meltdown can access memory outside of the current process, and even the kernel's memory space. To perform the Prime+Probe technique, the attacker must fill the cache with their data(prime), then have the victim access their memory to see which of the primed lines of memory is evicted. The attacker can see which lines have been evicted by referencing their data again and seeing the hit/miss times for each line(probe). The scope of Meltdown is also massive, as it works on every Intel processor since at least 2010.

2.1.6 Trojans [15]

Under some circumstances, attackers are also able to implant their own electronics into the victims' original system in an attack known as hardware trojans. As with most physical attacks, trojans can leak data, alter regular functions, and even completely shut off a device. To set themselves apart from other hardware attacks, they are extremely hard to detect due to their very low footprint compared to the rest of the device or system. The fact that they are not always active also makes them even harder to find, as the right trigger needs to occur for them to activate and carry out their intended task. Trojans also have multiple possible points of insertion(design IP, fabrication masks, PCB assembly, firmware/bitstreams, or post-market hardware modifications), which makes it more difficult to predict the culprit. These triggers can be based on time, a sequence of instructions, or external factors, making trojans even bigger of a threat. After the corresponding trigger, it is then able to slightly degrade the performance of the victim's device or simply analyze the data going through the host system.

2.2 Existing Products and Work

With the introduction of side-channel attacks, implementations to mitigate their effects and create more secure processor designs have emerged. This section will go over products that have been successful in implementing features that make the processor more secure.

2.2.1 Mitigating Side Channel Attacks on RISC-V **[5][6]**

To mitigate the vulnerabilities presented from Side Channel Attacks, it is necessary to look at the prior work that has been done in the space and see what techniques have been used. Giving both the context and motivation for choosing the project that is being implemented.

2.2.1.1 INVITED: Protecting RISC-V against Side-Channel Attacks

In research presented by Mulder et al, a RISC-V processor is implemented to tackle vulnerabilities during cryptographic operations in the ALU as well as memory accesses of these secret values in the effort to put less responsibility on software developers to make secure code. Their solution was to use masking techniques that made the operated or read/write data be independent from the secret.

For ALU operations, Boolean masking, specifically d+1 Threshold Implementation (TI), is performed where data get encrypted/combined with a random value (the mask) to produce masked data (which tracks as to be resistant to first order power attacks you must split data into two shares). The masks and masked data are operated on individually resulting in the intermediate values being independent of the secret and are only decrypted after the completion of the operation. Splitting into two shares creates two domains of operation obfuscating the dynamic power consumption and preventing data-dependent leakages as a result of glitching.

The next solution for more secure processing was to secure memory accesses as leakages can occur when the mask and masked data are transferred over the same data bus, so simply storing the mask and masked data in memory is not sufficient. In an effort to create a memory agnostic solution, a second seed was generated with a combination of the address and a selection of set seeds that can be chosen by the software developer. The seeds used to generate the new masked data can be constructed/reconstructed for encryption/decryption respectively with its hardware eliminating the potential leakage from mask and masked data overwrites as the second mask does not need to be written to memory. To eliminate other rewrite leakages at a specific address, reseeding initiated by the software developer rendering the content invalid. The content being invalid isn't an issue as the presence of multiple seeds allows for reseeding to occur after the secret is not relevant.

These two features are tested and provide power traces proving its functionality. Thus, showcasing two methods that mitigate Side Channel Attacks and create a more secure processor. The main drawback is the processor being closed source as it makes it hard to have a complete understanding of how the internal components are working. Additionally, the need for secure software to control reseeding might prove too difficult to create tests for given the timeline. Though Boolean Masking does prove to be a

potential avenue for this project's goals as it did not leak any information excluding at the beginning and end of 2 million power traces measurements.

2.2.1.2 PARAM: Power Attack ResistAnt Microprocessor

PARAM is an optimization of an existing open-source RISC-V called Shackti-C. This was done by using a methodology called PLAN to assess the security vulnerabilities through the Side Channel Vulnerability Factor (SVF) which quantified the strength of the side channel. Through this errors were found in the register bank, cache memory, control status registers, execute stage units and pipeline buffers. Additionally, vulnerabilities were found as a result of synthesis algorithms creating functionally correct translation of RTL but resulted in a higher susceptibility to side channels. This provided the basis for the improvement of the processor to make it more secure.

PARAM uses a similar method as the previous core in section 2.2.1.1 where it obfuscates the data by encrypting it. Similar to the boolean masking, it splits the data into two shares where it uses a seed to encrypt the data such that it is independent of the secret. Unlike boolean masking, these two 16 bit values are only used to encrypt the data by using the 16 bit seed. Using the 16 bit seed and the 16-bit values as inputs for a 4-round Feistel structure which uses the Affine transformation to combine the key with the input giving your encrypted data. Additionally, unlike boolean masking, the data stays encrypted while in the CPU except for operations which requires the data to be unencrypted unlike boolean masking where data is only unencrypted after cryptographic operations.

Similar to the previous implementation, PARAM stores the encrypted data as memory cells but doesn't use a different seed when storing the encrypted data. Additionally, both implementations have features where the seed used for encryption can be changed from the software invalidating all the data in the cache prior to the change. In this implementation, it specifically mentions that prior to the key change all dirty bits are written back to the on-chip RAM which might be part of but is not explicitly mentioned in the previous paper.

Finally, after looking at the synthesized netlist, the research team was able to identify that the automated flow tools had synthesized the RTL to create unwanted leakages. This is because of an attempted optimization by the synthesis tool to reduce cell count but failed to factor its effect on the security of the processor. To resolve this, the RTL had to be modified so that when synthesized it would keep this functionality.

These changes were very effective as the processor was significantly more secure. The Shackti-C processor when running an AES-128 software takes 62K power traces to recover the secret key while PARAM wasn't able to recover a single portion of the key after one million traces. Thus showcasing the effectiveness of the techniques used in the PARAM processor, but that it is not as effective as the RTL and techniques used in the first paper presented as resistance to attacks for over 2 million traces over this processor's

1 million traces. Thus, supporting the encryption of secret data while being in a processor and its memory is important for hardware security but there needs to be more done as this was not as efficient as the prior product which had encrypted cryptographic operations.

2.2.1.3 Secure RISC-V Microprocessor based on SERV Architecture

This paper presents a secure RISC-V microprocessor which was built of the open-source SERV RISC-V processor. In the presented implementation, the research team implemented many different techniques to implement the most robust and secure processor that when run with software for AES was able to withstand Differential Power Analysis for over 20 million traces.

This RISC-V core combines multiple different concepts including boolean masking, clock randomization, Random Number Generators (RNGs), and Differential Domino Logic methods in their processor. Clock randomization focuses on making it harder to pull information from Side Channel Attacks. By applying delays to the input clock, outputs are not consistent with the standard clock causing misaligned power traces. A mix of RNG modules and LFSRs are used to delay the clock's delivery to components and other modules in the circuit.

The Boolean Masking in this project is different than in the previous version as the splitting of shares is done through scripts in a netlist. After the initial synthesis, a script is used to replace all basic operations with masked counterparts. This is also the case for Differential Domino Logic as post-synthesis scripts are used to replace CMOS gates with special DDL gates which reduce glitching because of the precharge/evaluation phases that they have.

While this processor is the most secure out of all of them and highlights the importance of combining different methods in different portions of the processor to enhence security. The main drawback with this product is the use of specialized cells which is not accessible to us and thus is a limitation preventing us from implementing it. Although this is the case, unlike the other processors, this processor is open source which gives us a reference and will allow us to have a better understanding of the functionality behind the security features in the processor.

In conclusion, all these projects showcased the importance of how certain techniques especially integrated with each other in a processor's functions can mitigate and make it much harder to gain information from Side Channel Attacks such as Differential Power Analysis. Understanding what worked in these products is important as it gives guidance as to what secures future implementations of processors (which is out goal). One thing that was not mentioned in these projects was the use of extraneous logic to generate noise in the circuit. For all types of side channel attacks, including extra hardware to generate noise within the circuit obscures the signals related to secure operations, raises the difficulty of finding and listening to the signals necessary to extract valuable information. Hardware such as a Gaussian Noise Generator can be used as a random number generator to generate n-bit values over a normal distribution when certain processes are implemented in the chip. Effectively making it much harder to extract information from both power and timing analysis attacks. Seeing a combination this along with the techniques above is worth exploring to see if there are any possible security gains. Overall, now we understand what should be the opted features for

processors to make it more secure which includes the encryption of secrets during operations and storage to make it such that the encrypted data is completely independent of the secret as well as the inclusion of extraneous noise generators.

2.2.2 RISC-V Core Selection

As our design is intended to be an extension of an existing RISC-V core, we have explored various open-source RISC-V core designs to determine a sufficient foundation to expand upon:

2.2.2.1 Rocket Chip (CHIPS Alliance) **[1][2][3]**

The Rocket Chip generator is an open source framework developed at UC Berkeley that allows for the instantiation of a RISC-V core built around the Rocket CPU, a parameterizable, in-order, single-issue core. The framework supports both RV32 and RV64 ISAs, defaulting to RV64GC (64-bit with general-purpose, optional floating-point instructions).

The framework also demonstrates features to integrate larger SoC infrastructure(s) (caches, interconnects, peripheral interfaces). It has been purposely designed to support integration into the Chipyard ecosystem (also a framework from the CHIPS Alliance at UC Berkeley), meaning that this core has extensive room for expansion and scaling.

The main issue we would face using this design is that the RTL design is built using a hardware *construction* language (Chisel), as opposed to a traditional hardware *description* language (Verilog, SystemVerilog, VHDL). This construction language code then gets translated into synthesizable Verilog, but that translation layer adds extra abstraction that may make things difficult to design around. For this reason, the Rocket Chip core (framework) is one we may refer to when making design choices, though it is not what we will base our project on.

Along with these concerns, Rocket Chip provinces support for caches and various memory management unit (MMU) implementations, which are natural attack surfaces for side channels (timing attacks, microarchitectural attacks/cache-based leakage). Reiterating on its abstraction being a reason to not select this core, we can study Rocket Chip's SoC-level features to better understand how to introduce noise injection into cache access timings or even handle masking within MMU-managed load/store operations.

2.2.2.2 NeoRV32 **[4]**

NeoRV32 is a customizable SoC built around a 32-bit RISC-V CPU core, oriented towards beginners as an auxiliary controller in either larger SoC designs or compact customized microcontrollers. It is designed for ease of use and to fit into low-power and low-density FPGA devices.

It emphasizes execution safety to provide predictable behavior (stability) at all times. Precise and resumable hardware exceptions are presented whenever unexpected states occur. This makes NeoRV32 attractive for safety-critical or real-time embedded contexts.

This core seems to incorporate more than what is necessitated by our design goals, and is also written in VHDL, which is not our HDL of choice. They provide tools and steps to translate the design into Verilog (compatible with SystemVerilog) though,

like with Rocket Chip, this raises concerns of the readability and long-term maintainability of the codebase.

NeoRV32's design goal of execution safety and predictable behavior is particularly relevant to our mission. Its precise and resumable hardware exceptions highlight an architectural approach to ensuring deterministic responses/behavior(s) to unexpected states. This raises emphasis on constant-time execution and leak-resilient error reporting, in which exceptions are handled in ways that prevent data-dependent timing variations. Like mentioned prior, the VHDL implementation makes NeoRV32 insufficient to use as a baseline core, but we can most definitely leverage its approach to exception determinism to better mitigate side-channels attacks.

2.2.2.3 XiangShan **[7][8]**

XiangShan is a high-performance, out-of-order, superscalar RISC-V CPU design developed by the Institute of Computing Technology, Chinese Academy of Sciences. It is designed to function as a research-grade equivalent to industrial core designs, with an outright goal of approaching the performance of modern ARM and x86 cores.

XiangShan is implemented in Chisel and is much larger in scope than the other discussed cores. It includes advanced features like speculative execution, multi-level caches, and complex pipeline stages. It is designed around the RV64GCV (64-bit with vector extensions) ISA and is intended to be extremely scalable, often being used as a baseline for exploring server-class RISC-V CPU designs.

The overall complexity of XiangShan is one of the biggest drawbacks, as it seems far more complex than what is needed for our design. While some features such as speculative execution and out-of-order execution would allow us to explore vulnerabilities such as Spectre **[9]**, the overall size and layers of abstraction may prove unsuitable for our more lightweight modifications. It is also worth recognizing that much of the documentation for this design is natively written in Chinese. English translations exist, though only 6% of the translated work has been officially approved as accurate. Also, much like Rocket Chip, being written in Chisel adds further abstraction that may prove problematic even with translation layers. For these reasons, XiangShan will stay within our toolkit as a useful reference point for our selected design and implementations.

XiangShan's inclusion of speculative execution and out-of-order execution are prime vectors for side-channel attacks (such as Spectre[9] and Meltdown[14]). That makes the microarchitecture implementation a critical one for us to explore as a reference point as we explore extension implementations designed to disable, mask, or obfuscate speculative dataflow in a more basic design.

2.2.2.4 PicoRV32 **[11]**

PicoRV32 is a CPU core that implements the RV32IMC ISA (and is configurable to various permutations of the ISA), which includes an optional built-in interrupt controller. It is designed to be logically compact to fit on resource-constrained FPGA targets. It is also written in Verilog (specifically Verilog-2001).

PicoRV32's is particularly optimized for Xilinx 7-series FPGAs, achieving operating frequencies exceeding 250MHz on these platforms. Its efficiency allows it to act as an auxiliary processor in FPGA and ASIC designs, where its maximum frequency

ensures that clock domain crossing can often be avoided (high frequency provides plenty of timing slack when operated at lower clock speeds, easing timing closure).

PicoRV32 trades off its compactness through simplicity; it does not provide multi-level caches or advanced SoC features out of the box, like many of the other cores described, and is not intended to be used as a general-purpose CPU. It is best suited for control logic, small embedded platforms, or lightweight resource/accelerator management within larger designs. The overall simplicity of the design makes PicoRV32 compelling to use for our ISA extensions and architectural modifications, though we would prefer slightly more infrastructure regarding privileged memory modes, speculation, or OS-managed memory (PicoRV32 is not natively Linux-class).

By design, PicoRV32's omission of multi-level caches and speculative execution inherently reduce its attack surface/susceptibility for timing and speculation-based side channels. This simplicity makes the PicoRv32 an attractive platform for demonstrating mitigation techniques such as boolean masking in arithmetic instructions as well as randomized instruction latency injection. The lack of speculation or cache-based variability means that these modifications can be introduced and evaluated in an isolated context, simplifying design and verification stages as leakage trends may come off as easier to measure and can be more directly related to the extension implementations.

2.2.2.5 Ibex **[12]**

Ibex is a compact 32-bit, in-order RISC-V core released by the lowRISC project. It implements the RV32IMCB ISA (configurable between RV32EC, RV32IMC, RV32IMCB on micro, small/maxperf, and maxperf-pmp-bmfull configurations, respectively), with optional extensions and configurable features such as performance counters, debug support, and interrupt handling. Ibex emphasizes clarity, verifiability, and flexibility. It is written in SystemVerilog and includes a fleshed-out verification environment.

Ibex focuses heavily on industry-grade verification. The design has been formally verified with open-source frameworks, and the UVM environments, assertations, and test suites seem to be well-maintained.

Much like PicoRV32, Ibex is not natively Linux-class. It does not provide multi-level caches or a memory management unit out of the box. One stark contrast is that it does support Physical Memory Protection (PMP), which allows for memory access regions and privilege enforcement to be configurable. This makes Ibex an ideal candidate for exploring secure memory implementations, which is outlined as one of our stretch goals.

Ibex is somewhat more complex than PicoRV32, but simpler than the other designs discussed. For our purposes, Ibex's combination of SystemVerilog as the chosen HDL (without translation layer), configurable ISA support, and existing verification infrastructure makes it an ideal starting point for our proposed modifications/design. If our stretch goals seem reachable, then the existing architecture for privileged execution modes and memory protections will prove to be a beneficial leap forward in progress.

Ibex's support for PMP allows for privileged software to define fine-grained access regions in memory, which can be considered directly relevant to the goal of mitigating information through side channels and microarchitectural leakage. PMP can enable isolation between these trusted memory regions, ensuring untrusted code cannot

probe or infer properties of sensitive data through memory access patterns. PMP can be leveraged as a baseline enforcement mechanism that complements ISA-level modifications (constant time arithmetic or load/store operations), providing both spatial isolation and temporal uniformity against side-channel attacks. The implementation of PMP is dependent on the configuration/size of Ibex that we choose to move forward with, so at the very least we can use its principles in order to enact better architectural design choices, and at best we can leverage PMP as recently described.

2.2.2.6 SERV **[10]**

SERV (the SErial RISC-V core) is an extremely compact, bit-serial RISC-V CPU core. It implements the RV32I base instruction set and is widely regarded as the smallest RISC-V CPU in the world, with synthesis results showing it can fit into only a few hundred LUTs when programmed onto an FPGA. Unlike traditional cores that process entire words in parallel, SERV executes one bit per cycle,t reading off raw performance for dramatic reductions in area and power.

SERV is written in Verilog, intended for scenarios in which the constraints of area and power dominate over the needs of performance. This might include IoT sensor nodes, embedded controllers, or research/educational proof-of-concepts. Its simplicity makes it straightforward to understand and modify, though the single-bit execution model results in very long execution times for arithmetic and memory (load/store) operations.

From the perspective of side-channel analysis and our overall design goals, SERV is highly relevant despite (but also thanks to) its minimalistic approach. The bit-serial datapath inherently enforces a uniform execution structure, which naturally reduces opportunities for data-dependent timing variation. This makes SERV an ideal platform for conceptually exploring constant-time arithmetic and execution. Also, as every instruction gets decomposed into a fixed sequence of bitwise micro-operations, SERV provides a unique framework to study where leakage attack surfaces may arise, allowing us to explore how boolean masking or randomized instruction timing could be inserted at varying levels of granularity.

SERV comes off as too simple of a design as a candidate for our baseline implementation, partially due to its impractical performance limitations, though it makes a valuable reference point for our implementation. Its design philosophy demonstrates how architectural simplicity and serialized execution can be the key to mitigating certain classes of side-channel attacks, translating to the insertion of masking, noise injection, or instruction-latency randomization in more complex cores.

2.3 Goals and Objectives

As things currently stand, the undergraduate talent pipeline tends to be lacking in candidates that understand the full-flow digital ASIC design process. The overall initiative of this project is to expand upon an existing RTL design, applying industry-standard practices and tools to develop skills in the full digital ASIC design flow

whilst developing a RISC-V core that is able to mitigate side-channel attacks. This effort not only targets the creation of a valid GDS-II file/ASIC design (mock-tapeout), but also seeks to strengthen the undergraduate talent pipeline in semiconductor design. Goals are divided into basic, advanced, and stretch categories to help guide our progression:

2.3.1 Basic Goals

- Gain familiarity with the digital ASIC design process: RTL Design → Logic Synthesis → Physical Design → Signoff → Physical Verification → GDS-II tapeout
- Learn to effectively use Cadence EDA tools for ASIC development
- Develop extensions to a baseline RISC-V core RTL design with a custom instruction set extension, demonstrating mitigation to timing and power-based side channel attacks
- Synthesize modified RISC-V core using a 45nm process node
- Perform functional verification of RTL designs through directed testbenches and UVM-based environments
- Develop and execute RISC-V assembly programs to validate ISA compliance and system-level functionality
- Produce clear documentation describing design methodology, flow, and encountered challenges/troubleshooting steps

2.3.2 Advanced Goals

- Develop a custom interpreter or compiler that can interface hardware and software with custom instruction extensions
- Optimize the RTL design for area, power, and timing closure using industry practices
- Explore security-aware design considerations, such as fault tolerance or side-channel resistance, during verification

2.3.3 Stretch Goals

- Insert design for test (DFT) structures to validate functionality of a manufactured IC
- Perform gate-level simulation with back-annotated delays (SDF) to verify correct operations under realistic timing conditions
- Prototype Trusted Access Memory Region-like extensions for enforcing memory security policies in hardware
- Utilize open-source workflows to explore the integration of a lightweight embedded FPGA (eFPGA) fabric within the SoC to demonstrate reconfigurability for IP redaction use cases

2.3.4 Basic Objectives

- Instantiate the baseline RISC-V core to ensure we have a functional foundation
- Write RTL modules to integrate custom extensions that enforce noise injection to mitigate prospective side-channels targeting the modified core

- Run the official RISC-V compliance test suite in assembly to confirm ISA correctness
- Write custom RISC-V assembly and corresponding test suites to confirm validity of ISA extensions
- Document tool usage and workflow in a step-by-step manner
- Simulate the modified RTL design in Cadence Xcelium using representative workloads
- Complete various Cadence training courses pertaining to the ASIC design flow

2.3.5 Advanced Objectives

- Modify the RISC-V GCC (or LLVM) compiler to recognize and generate code for custom instructions; validate correctness by running C programs compiled with new extensions
- Regularly run Cadence Genus and Innovus to achieve PPA reports that demonstrate optimizations compared to the unmodified/previous iterations of the core both pre- and post-layout, respectively
- Develop testbenches to evaluate susceptibility to timing/power-based side channels, and validate effectiveness of custom instruction extension

2.3.6 Stretch Objectives

- Develop a regression testing framework to automate functional verification across RTL, synthesized, and post-layout designs, ensuring consistency after design changes
- Insert scan chains and boundary scan logic for DFT and verify functionality with fault simulation
- Explore Trusted Memory Access Region enforcement in RTL design using the existing PMP module that the Ibex core provides
- Integrate an open-source eFPGA fabric using OpenFPGA, VPR/VTR, and other encompassing workflows. This defaults to a 130nm process node (skywater), so also explore this process using the selected 45nm process node

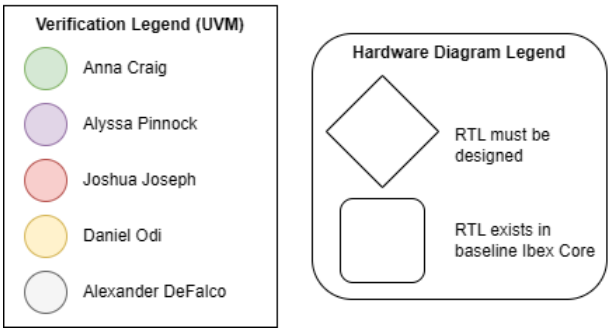
2.4 Engineering Specification Tables

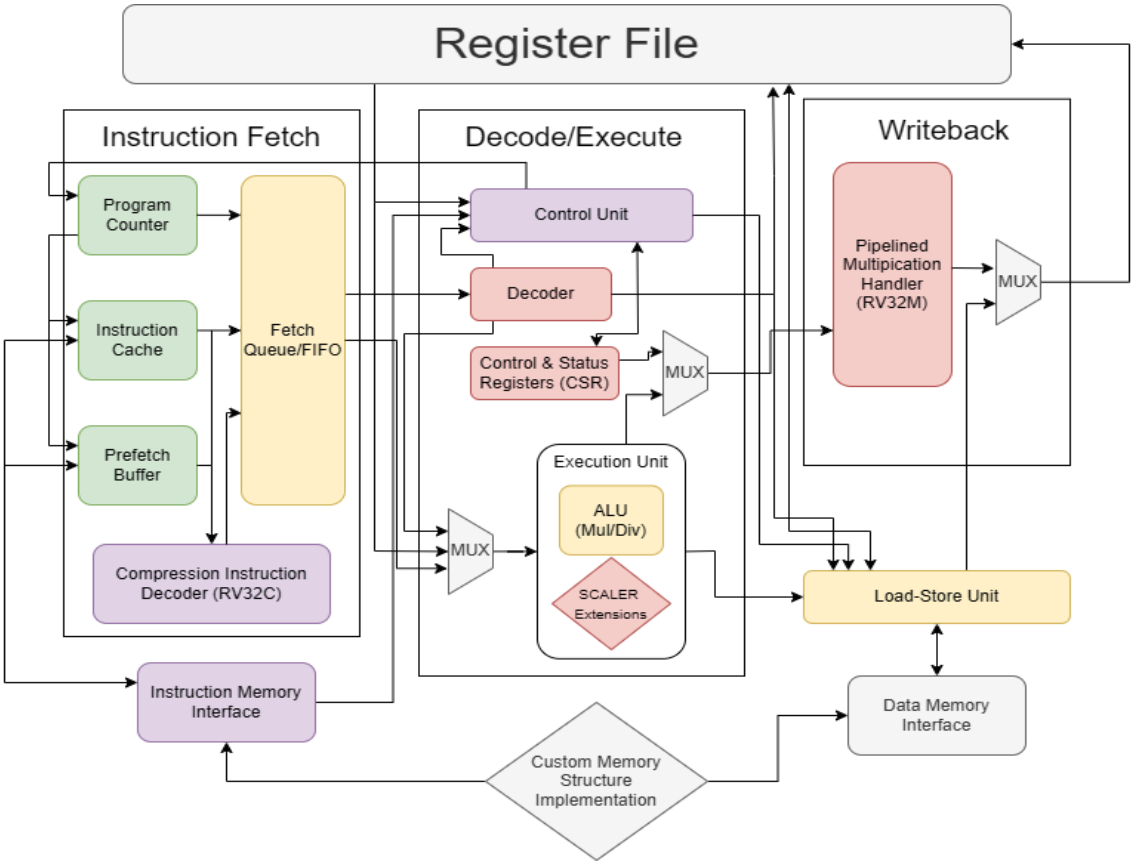
Engineering Specifications		
Parameter	**Specification**	**Priority**
Instruction Set	RV32I	High
Operating Temperature Range	0 °C - 120 °C	Low
Operating Voltage Range	0.95V - 1.25V	High

Engineering Specifications		
Process node	~45nm	High
IR Drop	≤ 5%	Low
Max Frequency	≥ 250 MHz	High
Masking Order	First-order (d=1) minimum	High
Instructions Per Cycle (IPC)	Average CPI ≤ 1.2	Medium
Cache Size	≥ 16 MB	High
Timing Failure Count	0 Faults	High

2.5 Hardware & Software Block Diagram

The Ibex core provides a strong foundation for us to build our design off of. Due to this, much of the delegated work boils down to verifying functionality of the design, implementing custom extensions to achieve SCALER's goals, and attaching a memory system to the instruction and data memory interfaces provided in the design. For that reason, the hardware and software block diagrams have been combined into one:





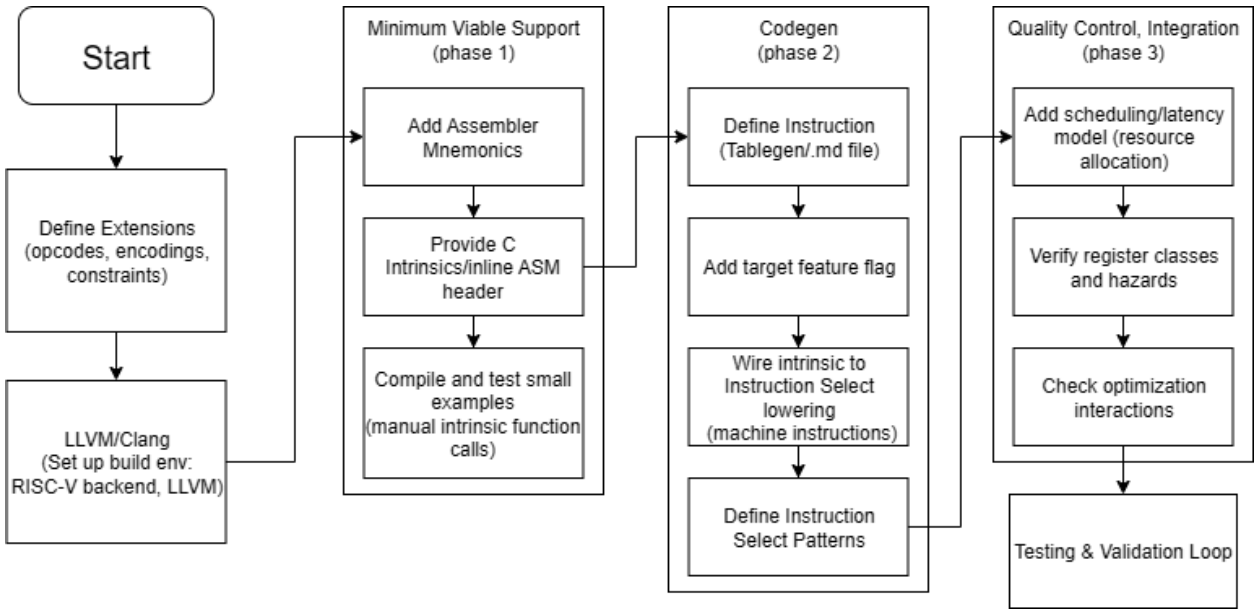
2.5.1 Figure 1: Design Verification and Development Flowchart with Legend

It is worth recognizing that the added SCALER extensions are lumped into the Execution Unit alongside the existing ALU, and additional connections from this module may be necessitated as the design flow carries on (especially once PPA and layout optimizations become a concern), and can truly only be addressed at that stage in development.

2.6 Compiler/Interpreter Software Diagram / Flowchart

A separate Software Diagram has been created to reflect the abstraction of our interpreter and compiler goals for executing RISC-V Assembly on the core. As the creation of assembly code can be done manually (with some effort), this goal is not being delegated to the entire team. Instead, this goal will be a joint objective of Alexander DeFalco and Joshua Joseph.

Compiler design has been broken up into three phases to abstract the feasibility and potential milestone points:

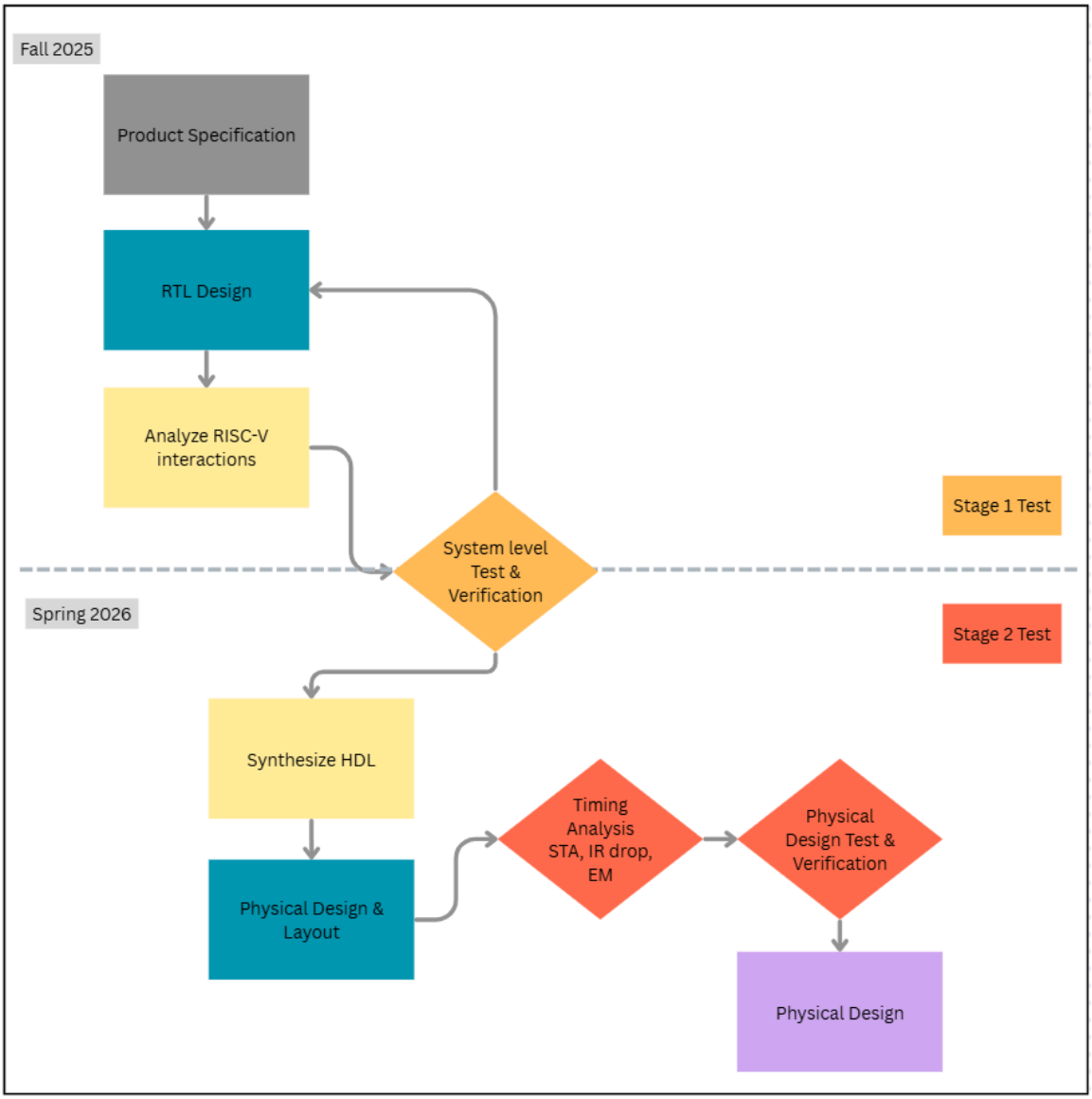


2.6.1 Figure 2: Compiler Development Flow

2.7 Prototype Illustration

For the purposes of our project scope, the prototype will be illustrated primarily through high-level representational models, rather than through direct hardware fabrication. This includes the development and presentation of high level system block diagrams (Figure 3) and our Software/Hardware diagrams.

Together, these artifacts will allow us to effectively convey the structure, function, and performance characteristics of our design, considering the lack of physical implementation we will have.



2.7.1 Figure 3: ASIC Design Flow Process

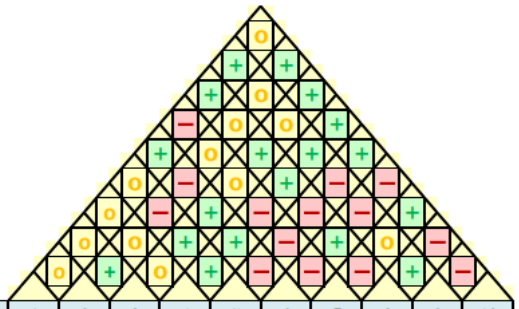
2.8 House of Quality

Our House of Quality (HoQ) reflects a structured framework to link potential customer requirements with corresponding engineering requirements that will guide our design process. This approach highlights trade-offs and prioritizes design objectives to balance performance, cost, and maintainability within the ASIC design flow.

Correlations	
Positive	+
Negative	–
No Correlation	o

Relationships	
Strongly Positive	↑ ↑
Weakly Positive	↑
No Relation	○
Weakly Negative	↓
Strongly Negative	↓ ↓

Direction of Improvement	
High	▲
Low	▼



Column #		1	2	3	4	5	6	7	8	9	10
Engineering Requirements / Customer Requirements	Direction of Desirability	Instruction Set	Operating Temperature Range	Operating Voltage Range	Process Node	IR Drop	Max Frequency	Masking Order	Instructions Per Cycle (IPC)	Cache Size	Timing Failure Count
Direction of Desirability		▲	▲	▲	▲	▲	▲	▲	▲	▲	▲
Affordability	▲	○	○	○	↓ ↓	↓	↓	↓ ↓	↓	↓	○
Durability	▲	○	↑	↑	○	↑	↓	○	○	○	↑ ↑
Expandability	▲	↑ ↑	○	○	○	○	↑ ↑	○	↑	↑	↑
Reliability	▲	○	↑	↑	↓	↑ ↑	↑ ↑	○	○	↑	↑ ↑
System Compatibility	▲	↑ ↑	↑ ↑	↑ ↑	○	○	↑	↓	↑ ↑	↑ ↑	↑ ↑
Forward Compatability	▲	↑ ↑	○	○	○	○	↑	↑	↑	↑	↑ ↑
Ease of Use	▲	↑ ↑	○	○	↑ ↑	○	○	↓	↑	↑	↑ ↑
Ease of Maintenance	▲	↑ ↑	○	↑	○	○	↓	↓	○	↑	↑ ↑
Security-Oriented Extension	▲	↑ ↑	○	○	○	↓	○	↑ ↑	○	○	↑ ↑
Target		RV32I	0 °C - 120 °C	0.95V - 1.25V	~ 45nm	≤ 5%	≥ 250 MHz	First-order (d=1) minimum	Average CPI ≤ 1.2	≥ 16 MB	0

2.8.1 Figure 4: House of Quality

2.9 Budget and Financing

Our project requires minimal physical implementation, as the majority of the work will focus on simulation, development, and validation in controlled environments. A summary of the tools and resources we plan to utilize is as follows:

- Cadence Xcelium- A professional-grade simulation environment used for verifying RTL designs.
- Development boards such as the BASYS-3- These boards will serve as prototyping platforms for testing and demonstrating hardware functionality.
- Lab Test Equipment- Standard bench equipment, including oscilloscopes, logic analyzers, and power supplies may be used if needed to verify functionality.
- Code Development Environments such as VSCode- Integrated development environments will support the creation, testing, and refining the software and HDL.

All of the above software and hardware resources are already secured and funded through Northrop Grumman's sponsorship of the project. UCF provides necessary lab equipment and infrastructure, including access to development boards and test equipment.
Additional software licenses and specialized tools such as Cadence Xcelium have been procured by Northrop Grumman. Open-source tools are also available for use in our project.

Given this agreement, we do not anticipate the need to allocate a separate budget for tool usage. All products are either free to use, provided by UCF, or already purchased and funded by Northrop Grumman.

2.10 Project Milestones

The project milestones listed below outline the design and documentation process for SCALER. This outline will keep the team on track with project requirements and deadlines to properly meet our core goals and objectives. The milestones are organized by the task, start date, end date, duration, and description.

2.10.1 Senior Design 1

Senior Design 1 Timeline				
Task	**Start Date**	**End Date**	**Duration**	**Description**
Group Formation	8/18/25	8/25/25	1 week	Create project group and assign roles
Project Formation	8/26/25	9/2/25	1 week	Form project idea (SCALER), design scope, and goals
Complete Divide & Conquer Document	8/28/25	9/4/25	1 week	Draft and complete project proposal/Divide & Conquer document
D&C Review	9/10/25	9/10/25	1 day	Meet with advisors and Dr. Chan for proposal review

Senior Design 1 Timeline				
Start 40-Page Draft	9/10/25	9/18/25	1 week	Write 40-page draft (¼ of final paper)
High-Level Design & Instruction Assembly	9/11/25	10/2/25	3 weeks	Define the SCALER custom instructions, RTL design, and instruction encoding. Justify the workloads.
40-Page Draft Revision	9/18/25	9/25/25	1 week	Revise and organize 40-page draft for final paper
Start 75-Page Draft	9/26/25	10/9/25	2 weeks	Write 75-page draft (½ of final paper)
RTL Functional/Formal Verification	10/3/25	10/24/25	3 weeks	Test each instruction module and ensure correctness with simulation/UVM
75-Page Draft Revision	10/10/25	10/17/25	1 week	Revise and organize 75-page draft for final paper
Final Report Draft	10/18/25	11/7/25	3 weeks	Write remaining pages for final report (up to 150 pages)
System-Level Testing	10/25/25	11/7/25	2 weeks	Write RISC-V assembly to test instructions
Final Report Revision & Submission	11/8/25	11/20/25	2 weeks	Revise final report and submit

2.10.2 Senior Design 2

Senior Design 2 Timeline				
Task	**Start Date**	**End Date**	**Duration**	**Description**
Verification wrap-up	1/6/26	1/20/26	2 weeks	Complete any remaining RTL/system-level verification from SD1

Senior Design 2 Timeline				
HDL Synthesis & Netlist Generation	1/21/26	2/3/26	2 weeks	Convert RTL to gate-level netlist (target ~45nm technology)
Physical Design & Layout	2/4/26	2/24/26	3 weeks	Create floorplan, placement, and routing
Static Timing Analysis (STA) & Power Analysis	2/25/26	3/10/26	2 weeks	Perform STA, IR drop, and electromigration analysis to meet constraints (5% with no timing failures)
DRC/LVS & Design Signoff	3/11/26	3/24/26	2 weeks	Verify layout vs schematic (LVS) and manufacturability (DRC)
Mock Tapeout & Verification Metrics	3/25/26	4/7/26	2 weeks	Generate mock GDSII file and document final verification results
Final ASIC Design Completion	TBD	TBD	1 week	Review and wrap up final verification for ASIC design

2.10.3 Distribution of Work

Work Distribution	
Team Member	Task(s)
Anna Craig	Verification of program counter, instruction cache, and prefetch buffer
Alexander DeFalco	Design of Custom Memory Structure, in addition to verifying the Data Memory Interface

Work Distribution	
Joshua Joseph	SCALER Extension design, with decoder, CSR, and multiplication handler verification
Daniel Odi	FIFO, ALU, and Load-Store Unit verification
Alyssa Pinnock	Control unit, Instruction Memory Interface, and Compression Instruction Decoder verification

2.11 Appendices

2.11.1 Citations

[1] chipsalliance, “rocket-chip (Rocket Chip Generator),” GitHub, Accessed: Sep. 4, 2025. [Online]. Available: https://github.com/chipsalliance/rocket-chip

[2] Chipyard, “3.2 Rocket Core — Chipyard 1.11.0 documentation,” Chipyard Documentation, accessed Sep. 4, 2025. [Online]. Available: https://chipyard.readthedocs.io/en/stable/Generators/Rocket.html

[3] Chipyard, “Welcome to Chipyard’s documentation (version “1.11.0”),” Chipyard Documentation, accessed Sep. 4, 2025. [Online]. Available: https://chipyard.readthedocs.io/en/stable/index.html

[4] stnolting, “neorv32-verilog: Convert the NEORV32 processor into a synthesizable plain-Verilog netlist module using GHDL,” GitHub, access date: Sep. 4, 2025. [Online]. Available: https://github.com/stnolting/neorv32-verilog

[5] J. Mishra and S. K. Sahay, “Modern hardware security: A review of attacks and countermeasures,” *arXiv preprint*, arXiv:2501.04394v1 [cs.CR], Sept. 4 2025. https://doi.org/10.48550/arXiv.2501.04394

[6] E. D. Mulder, S. Gummalla and M. Hutter, "INVITED: Protecting RISC-V against Side-Channel Attacks," 2019 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, 2019, pp. 1-4.

[7] OpenXiangShan, “XiangShan: Open-source high-performance RISC-V processor,” GitHub, accessed Sep. 4, 2025. [Online]. Available: https://github.com/OpenXiangShan/XiangShan

[8] XiangShan, “香山官方文档 (最新版本),” XiangShan Documentation, accessed Sep. 4, 2025. [Online]. Available: https://docs.xiangshan.cc/zh-cn/latest/

[9] P. Kocher et al., “Spectre Attacks: Exploiting Speculative Execution,” 2018. [Online]. Available: https://spectreattack.com/spectre.pdf (accessed Sept. 4, 2025).

[10] O. Kindgren, “SERV – The SErial RISC-V CPU,” GitHub repository, Sept. 4, 2025. [Online]. Available: https://github.com/olofk/serv

[11] YosysHQ, “picoRV32 – A size-optimized RISC-V CPU,” GitHub, accessed Sept. 4, 2025. [Online]. Available: https://github.com/YosysHQ/picorv32

[12] lowRISC, “Ibex: A small 32-bit RISC-V CPU core previously known as zero-riscy,” GitHub, accessed Sept. 4, 2025. [Online]. Available: https://github.com/lowRISC/ibex

[13] Ahmadi, M. M., Khalid, F., & Shafique, M. (2021, June 16). *Side-Channel Attacks on RISC-V Processors: Current Progress, Challenges, and Opportunities*. arXiv. https://arxiv.org/abs/2106.08877]

[14] M. Lipp *et al.*, “Meltdown,” 2018. [Online]. Available: https://arxiv.org/pdf/1801.01207 (accessed Sept. 11, 2025).

[15] M. Xue et al., “Ten Years of Hardware Trojans: A Survey from the Attacker’s Perspective,” IET Computers & Digital Techniques, vol. 14, no. 6, pp. XXX-XXX, Sept. 2020. [Online]. Available: https://doi.org/10.1049/iet-cdt.2020.0041 (accessed Sept. 11, 2025).