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

Contents

List of Tables

List of Figures

Acknowledgements

(Optional acknowledgements go here.)

Chapter 1 - Executive Summary

Many undergraduates in Electrical and Computer Engineering do not get hands-on experience with the ASIC design flow process, however industry has increasingly desired engineers who are familiar with RTL verification and design procedure. SCALER aims to address this gap through creating an extension for a RISC-V processor that can mitigate side channel attacks through effective leakage reduction techniques. Our proposal frames the effort through gaining familiarity with ASIC design processes and utilizing Cadence EDA tools to develop side-channel-aware RTL into an ASIC design suitable for fabrication.

The SCALER extension is based on an open-source RISC-V core that can defend against timing and power side-channel leakage, while pushing the design through a professional ASIC tool flow to a mock tapeout. This project builds on Ibex, an open RISC-V core chosen for its strong verification history and practical features. Ibex's SystemVerilog and UVM environments are well maintained, and its support for Physical Memory Protection (PMP) provides a natural hook for defining memory regions and enforcing privilege. This is useful when designing constant-time behaviors and reducing microarchitectural leakage. These characteristics make Ibex a realistic and modifiable baseline for modification and extension in our senior design project.

The methodology for our project follows the process of a full-flow ASIC design: simulation with Cadence Xcelium/SimVision, synthesis with Genus, and floorplan/place-and-route with Innovus. The resources, licenses, and specialized tools are provided through Northrop Grumman sponsorship, with UCF supplying lab infrastructure and development boards. This sponsorship allows for the work to focus on engineering execution and on preserving the intent of side-channel countermeasures through optimization and layout.

Given our scope and timeline, we will document the prototype using simple high-level models, including block diagrams, basic software and hardware flows, and an overview of the ASIC process, along with real tool outputs that lead to a mock GDSII. A clear milestone plan across Senior Design I and II takes the project from proposal and design to verification, testing, and the final report, so the results match both the security goals and the core steps of a full digital design flow.

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 that the main goal of this project is to gain familiarity with the 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 a behavioral-level description of your circuit design into a 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

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 physical attacks, which target the hardware implementation

rather than the software. In order to carry out their disruption, they use a variety of tools to monitor and tamper with electronics [2], 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 [2]. 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.

To expand further, Invasive attacks typically involve some form of physical intrusion into the die or package, such as through delidding or laser fault injection. Non-invasive attacks do not require any physical hardware access and instead use logical access or external observations (observations from user space) to be performed.

2.1.3.1 Side-Channel Attacks

Side-channel attacks are hardware-based attacks that exploit systems by taking advantage of unintended information leaks rather than stealing secret information directly. These unintended leaks occur through "side channels", which includes information such as timing for a system's operations, electromagnetic (EM) attacks, monitoring power consumption, and various other forms of information that can be observed when a system is running. Side-Channel Attacks (SCAs) are powerful hardware system attacks for many reasons, the most common ones being that they are hard to prevent and difficult to detect. Since they exploit unintended leakages, it is difficult to mitigate what information about the system and its operations may be leaked. They are also difficult to monitor and detect because SCAs are passive and non-intrusive, so it is difficult to determine if or when an attack is occurring. Developments in machine learning, statistical analysis, and AI also contribute to making SCAs more powerful, and therefore require better prevention techniques for hardware security.

In our project, we plan to modify the existing Ibex core to develop an extension to mitigate side-channel leakage and make the RISC-V core more resilient to side-channel attacks. However, this requires first analyzing the different types of side-channel attacks and determining which type(s) of attacks to prevent against. The two main types of SCAs we analyze are timing-based attacks and power-based attacks. Timing attacks exploit timing variations when specific operations are occurring on the system. Power attacks involve an attacker that monitors a device's power rails during operation, and they can steal information through observing the current draw and voltage fluctuations.

2.1.3.2 Timing Attacks

The basis of a timing-based attack is that the attacker can steal a user's data through exploiting variations in the timing of operation execution. They are able to measure the execution time of specific operations under different inputs and perform

statistical analysis to deduce data like encryption keys. Operations can be especially vulnerable to timing attacks if they involve cache accesses, RAM, branching, and similar functions. One example of a timing attack includes Paul Kocher's timing attack in 1996, where he analyzed how square-and-multiply exponentiation in RSA can run faster when processing a 0 bit rather than a 1 bit in the key (Kocher, 1996). This allows the key bits to be revealed by the timing operation, and the leakage of the timing details from the processor. Measuring the differences in execution time and small changes in computation times can expose information about the system that was not previously available.

A common subtype of timing attacks is the cache timing attack, where an attacker can exploit how cache hits and misses influence the overall execution time. Flush + Reload and Prime + Probe techniques can be used by an attacker to determine cache lines that are being accessed to steal critical data. Attackers can also analyze the cache timings to reveal where in the table is being accessed. Additional timing differences in the microarchitecture of a system can still be vulnerable to side-channel leakage, and this provides the attacker more information to perform additional attacks that go beyond SCAs.

Why we aren't using timing: We decided not to test timing-based attacks due to the setup of the small in-order Ibex core. We would require special on-chip timers and crafted microbenchmarks to measure the timing differences externally. The AES workload also involves the design to be constant-time and does not include secret-dependent branching or table lookups. Timing guidance will be used while focusing experimental effort on power analysis, and this will provide clear and trace-based signals that are straightforward to capture and analyze in the lab.

2.1.3.3 Power Attacks

In power-based side-channel attacks, it involves the attacker monitoring the power consumption and power rails on a device while it operates. It also analyzes the repeated patterns in current draw and voltage fluctuations to predict the type of data that may be processed by the system. It takes advantage of the fact that in CMOS devices, there is a correlation between the power consumption and the switching activity of the transistors. The switching operation of the transistors involves the bits changing between 0 and 1, so a cryptographic operation will draw different amounts of current for different inputs and key bits.

- Simple Power Analysis (SPA): SPA involves visually inspecting power traces to find clear patterns . If the device's operations are clearly distinct in terms of power consumption, an attacker can sometimes recognize the cryptographic sub-operations and decrypt the classified data. One example is RSA on a smart card, where squaring vs. multiplication operations have different power traces or signatures. By looking at one trace of an RSA decryption, the attacker can spot various lower power spikes for square operations and higher spikes whenever a multiplication operation is executed . This allows the SPA attack on RSA to immediately reveal the bits of the private key. Simple Power Analysis is one of

the simplest forms of power analysis attacks, and it requires prior knowledge of the implementation of the system and its operations. However, it only works on relatively simple or unprotected systems like microcontrollers rather than more complex systems that have multiple power traces for different operations.

- Differential Power Analysis (DPA): DPA is one of the most powerful techniques, and they utilize statistical analysis on power traces to determine differences in power draw. The attacker first has to collect numerous power traces from the device that each correlate with an operation within the system. They have to predict a portion of that data and utilize a select function to divide the traces into two different groups, where one predicts a key-bit and the other group would not. Through prediction and analyzing the correct traces, the attacker can infer that one group would draw slightly more power. They repeat this operation and rule out noise from the measurements to derive a clear prediction for which operations are used based on the power draw and differences in power traces.

- Correlation Power Analysis (CPA): CPA is a form of DPA that utilizes the correlation coefficients instead of the difference between the means, and the results require less traces. In a CPA attack, the attacker assumes a specific power consumption model for the device. The model will assume that the power is drawn proportional to the specific types of bits being processed. Using the specific power consumption model, the attacker will then utilize the statistical analysis to correlate their predicted values with the measured trace samples they gathered from the victim. Guessing the correct key during processing will reveal the device's power at that moment, which exploits the power consumption information. This is considered a form of DPA, but CPA's use of an analog correlation metric typically makes it more sensitive and requires fewer traces than classic DPAs. CPA is very effective against AES and similar algorithms where a linear relation between bit transitions and power can be assumed, and it has been a core technique in academic SCA experiments due to its relatively low trace count requirements.

2.1.4 Spectre and Meltdown

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 [2]. 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 [3], 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.

2.1.5 RowHammer

RowHammer is a non-invasive, active fault-injection attack on DRAM: by rapidly and repeatedly activating "aggressor" rows within the same bank, an attacker can induce bit flips in physically adjacent "victim" rows. The phenomenon was first characterized on commodity systems by Kim et al. (Carnegie Mellon & Intel Labs) at ISCA 2014 [4], who showed that as few as 139k row activations within a refresh window can cause flips. Subsequent work demonstrated practical exploits that escalate privileges from user space on real machines, as well as the ability to profile or perform "memory templating" to record predictably flippable bits [5].

Subsequent work has turned RowHammer into more practical exploits that impact web browsers, mobile applications, and perhaps most credibly cloud computing [6]. This has been done using placement control and eviction strategies to strategically induce bit-flips among security-critical data. The deterministic templating of data on mobile platforms or the cross-VM placement of flips through deduplication has made these types of attacks much more reliable in underprivileged contexts. A recent implementation of RowHammer through a JavaScript web interface only highlights this.

While RowHammer itself is not a side-channel attack, it is an active fault-injection technique that showcases the breaking of data integrity through induced bit flips. It holds relevance to side-channel attacks as an enabler; the disturbances it causes through these bit flips can serve as the active induction phase that, when paired with a passive measurement phase, yields a read-based side-channel. Discussed

later is an attack known as RAMBleed, which demonstrates this hybrid functionality enabled through RowHammer. In short, RowHammer can be considered a catalyst or enabler to many potential side-channel attacks.

2.1.6 RAMBleed

As discussed earlier, the RowHammer exploit is a reliability issue within modern DRAM (DDR4, DDR5) implementations that can enable underprivileged users to induce bit-flips within a memory module [4]. Within those discussed works, it was assumed that RowHammer on its own is not a significant concern, as induced bit-flips within one's own memory space is of minimal concern, as the modified data was already data that the given user (or attacker) had access to. RAMBleed breaks this paradigm by showcasing RowHammer being used to implement a read side channel attack, highlighting concerns with both data integrity and confidentiality [7].

Within RAMBleed, DRAM disturbance is considered data-dependent. That implies that the probability of a bit flip within a cell depends on the values above it and below it within the same column. If the cells above and below are the opposite value of the center bit (010 or 101), then this is considered a "striped" configuration. If they are the same (111 or 000), this is considered a "uniform" configuration.

The RAMBleed attack assumes no physical access to a given system, going beyond the classification of being non-invasive. Without physical access, many cold boot attack methods are unavailable. Instead, the properties of Pseudo-Random Number Generation (PRNG) is able to be exploited. Modern memory controllers typically apply a form of memory scrambling by XORing data with a PNRG mask derived at boot-time. At boot-time of a system, the random seed is identical for all rows, and the index of physical addresses within the seed are generated with the same mask applied. This allows for striped rows to remain striped after boot, which is important for the preservation of data-dependent property, allowing for attackers to work without physical access.

The authors of RAMBleed were able to leak a 2048-bit RSA encryption key on a consumer platform, whilst adjacently analyzing RAMBleed's impact on ECC platforms, showcasing how the attack holds merit on various forms of computing.

In the end, RAMBleed is a side-channel attack that uses RowHammer-style disturbances for the active setup phase, with a passive readout via bit flips and/or timing attacks in attacker-owned pages to violate confidentiality within a system. It demonstrates that DRAM disturbance does not just highlight data integrity, as even ECC platforms hold vulnerabilities through information leakage paths after correcting these disturbances.

2.1.7 RowPress

The RowHammer attack highlighted how frequent access of adjacent memory rows can induce bit-flips in memory systems, which can then be used to enable read-based side-channel attacks [4]. This isn't the only vulnerability that exists in memory as far as read disturbances go; Researchers from ETH Zürich discovered that

if any memory rows are accessed and left open for too long, these rows can have data "expire" and have bit flips occur [8] [9]. Their work discovered that this effect is only compounded as process nodes improve (shrink) and that all popular DRAM vendors face the same vulnerability, albeit at different rates. This attack has been referred to as RowPress.

RowPress shows that keeping a DRAM row open for a long time can disturb adjacent rows with enough strength to induce bit flips, without the rapid open/close/toggling characteristic of RowHammer. In other words, RowPress is a read-disturb phenomenon driven by time, through activation as opposed to activation count.

Experiments find that RowPress is able to reduce the minimum activations to first bit-flip (ACmin) by roughly 1-2 orders of magnitude as compared to the original RowHammer experiment results. This broadens the circumstances under which disturbance errors can appear. Under extreme conditions, a single adjacent-row activation can suffice to induce bit-flips, only compounding this further.

As briefly mentioned earlier, this vulnerability is present across all major DRAM vendors, and there is minimal overlap between DRAM cells vulnerable to both RowHammer and RowPress, which indicates that there is a complementary error/attack surface. Further works showcase systems that have RowHammer oriented protections built in that are still susceptible to RowPress, despite them both being DRAM-based attacks. Not only are they susceptible to RowPress, but hybrid attacks have been designed that characterize combined access patterns of both RowHammer and RowPress [10], resulting in even faster induction of bit flips than either attack alone.

Like RowHammer, RowPress is not a side-channel attack on its own. Instead, it is an active-fault mechanism. It holds significance to side-channel attacks by lowering the bar for causing disturbance (through the long row-open times) and revealing a largely disjoint set of vulnerable cells than just RowHammer, widening the attack space for hybrid attacks pairing active disturbance with passive readout. It also brings to light that many attacks that appear to have overlapping attack surfaces may in fact have minimal overlap, compounding security concerns.

2.1.8 Trojans

Under some circumstances, attackers are also able to implant their own electronics into the victims' original system in an attack known as hardware trojans [11]. 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

The rise of side-channel attacks has encouraged the development of new techniques and processor designs to reduce their impact. This section reviews existing research and products that have successfully introduced hardware features to make processors more secure.

2.2.1 Mitigating Side-Channel Attacks on RISC-V

To address vulnerabilities caused by side-channel attacks, it is important to review previous work in this field and understand which methods have been used. This background helps explain the design choices made in the project described in this paper.

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

In the study by Mulder et al. [12], a RISC-V processor was designed to protect against vulnerabilities that appear during cryptographic operations in the Arithmetic Logic Unit (ALU) and in memory accesses to secret data. The goal was to reduce the need for software developers to write additional secure code by providing hardware-level protection. Their approach used masking techniques to ensure that the data being processed or transferred was independent of the secret values.

For ALU operations, they used Boolean masking based on the $d + 1$ Threshold Implementation (TI) method. This technique combines the data with a random value (called a mask) to produce masked data. To resist first-order power attacks, the data are divided into two parts, or "shares," which are processed separately. The intermediate results remain independent of the secret and are only combined after the operation is complete. This two-share design hides data-dependent power consumption and helps prevent leaks caused by signal glitches.

To secure memory operations, the researchers developed a memory-independent masking system. Storing both the mask and masked data together could leak information when they share the same data bus. To avoid this, a second seed is created using the memory address and several predefined seeds chosen by the software developer. These seeds allow the system to reconstruct the masked data during encryption or decryption without storing the mask in memory. The software can also trigger reseeding, which invalidates memory contents once the secret data are no longer needed.

Testing confirmed that these methods significantly reduced power-based leakages. However, since the processor design is closed source, it is difficult to fully understand its internal mechanisms. Additionally, relying on secure software to manage reseeding could make testing more complex. Even so, Boolean masking proved

to be a strong technique, as no meaningful information leaked during more than two million power trace measurements, except at the very beginning and end of execution.

2.2.1.2 PARAM: Power Attack ResistAnt Microprocessor

PARAM is an enhanced version of the open-source RISC-V processor Shakti-C. Its development used a framework called PLAN, which identifies side-channel weaknesses using the Side Channel Vulnerability Factor (SVF). This process revealed security issues in several parts of the processor, including the register bank, cache, control registers, execution units, and pipeline buffers. It also showed that synthesis tools—although functionally correct—sometimes produced hardware that was more vulnerable to side-channel attacks. These findings guided PARAM's improvements.

PARAM uses a data encryption method similar to Boolean masking. Data are divided into two 16-bit shares and encrypted using a 16-bit seed. The encryption process applies a four-round Feistel structure with affine transformations to mix the key and input. Unlike Boolean masking, the data remain encrypted while inside the CPU and are only decrypted when necessary for computation.

Like the previous design, PARAM stores encrypted data in memory but does not generate a new seed for each storage operation. Both systems allow the software to change the encryption seed, which invalidates all cached data. Before a seed change, PARAM ensures that all modified cache lines are written back to on-chip memory—a step that was implied but not clearly stated in the earlier research.

The team also discovered that automatic synthesis tools introduced unintended leakages by optimizing the hardware in ways that reduced security. To fix this, the RTL code was adjusted to preserve security features during synthesis.

These improvements made PARAM much more secure. The original Shakti-C processor required around 62,000 power traces to recover an AES-128 key, while PARAM did not reveal any part of the key even after one million traces. Although PARAM's protection was slightly weaker than the first study's RTL-level countermeasures (which stayed secure past two million traces), it still showed that encrypting data within the processor is an effective way to improve resistance to side-channel attacks.

2.2.1.3 Secure RISC-V Microprocessor Based on SERV Architecture

This research describes a secure RISC-V microprocessor built from the open-source SERV processor. The design combines several security techniques to create a processor capable of withstanding Differential Power Analysis (DPA) for over 20 million traces during AES encryption tests.

This RISC-V core uses a mix of Boolean masking, clock randomization, Random Number Generators (RNGs), and Differential Domino Logic (DDL). Clock randomization introduces variable timing delays, making it harder for attackers to align power traces and extract useful information. RNG modules and Linear Feedback Shift Registers (LFSRs) are used to generate these random delays across the circuit.

Unlike previous designs, Boolean masking and DDL were applied after synthesis.

Scripts were used to replace basic operations with masked equivalents and to substitute regular CMOS gates with DDL gates. DDL gates help prevent signal glitching by using precharge and evaluation phases, reducing opportunities for power-based data leaks.

Among all the designs reviewed, this processor showed the strongest protection against side-channel attacks, demonstrating that combining multiple techniques provides significant security benefits. The main limitation, however, is its reliance on specialized logic cells, which are not easily accessible for practical use. Despite this, its open-source nature makes it an excellent reference for understanding secure hardware design.

In conclusion, these projects highlight how combining several hardware-level techniques—such as masking, randomization, and encryption—can make it much harder for attackers to extract information through side-channel analysis. Studying these approaches provides valuable insight into which features should be prioritized in future secure processors. One potential improvement not explored in these works is the use of extra logic to create random noise within the circuit. Adding noise, such as through a Gaussian Noise Generator, can obscure the signals that attackers depend on, making both power and timing analysis more difficult. Exploring a combination of noise generation with the methods discussed above could lead to further security gains. Overall, secure processors should prioritize encrypting data during both computation and storage, ensuring that all processed values remain independent of the original secret, while also considering hardware-based noise generation to increase resistance to attacks.

2.2.2 Side-Channel Attacks

One of the most common methods of attacking hardware is the side-channel attack, where the attacker exploits unintended information leakage [13]. 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.

In general, side-channel attacks are passive- the attacker observes unintended leakage (e.g., timing, cache behavior, power/EM emanations) produced by normal execution (implying a non-invasive approach). However, several modern microarchitectural exploits (most notably Spectre and Meltdown) actively induce transient or otherwise favorable states and then passively measure the resulting leakage, such as through cache-timing channels. While no direct modifications of the microarchitecture are necessary for the attack to be executed, it causes the side-channel attack to involve some active phases. For this reason, we can consider many attacks to be a "hybrid", with active setup phases and passive readout phases.

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 Specifications Table

Table 2.1: *Engineering Specifications*

Engineering Specifications		
Parameter	**Specification**	**Priority**
Instruction Set	RV32I	High
Operating Temperature Range	0 °C – 120 °C	Low
Operating Voltage Range	0.95 V – 1.25 V	High
Process Node	∼45 nm	High
IR Drop	$\leq 5\%$	Low
Average Frequency	≥ 250 MHz	High
Masking Order	First-order minimum	High
Instructions per Cycle (IPC)	≤ 1.2	Medium
Cache Size	≥ 16 MB	High
Timing Failure Count	0 Faults	High
Interrupt Latency	≤ 12	Medium
Leakage Power (25 °C)	$\leq 5mW$	High

2.5 House of Quality

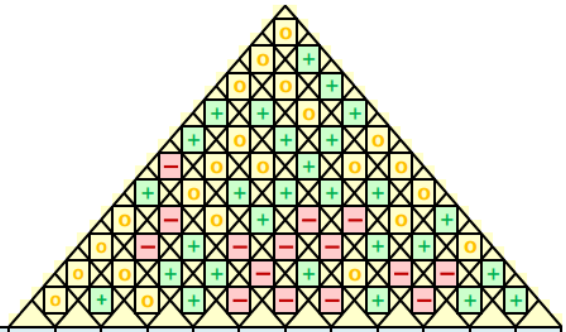
Our House of Quality (HoQ) reflects a structured framework to link potential customer requirements with corresponding engineering requirements that will guide

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	11	12
Customer Requirements \ Engineering Requirements	Direction of Desirability	Instruction Set	Operating Temperature Range	Operating Voltage Range	Process Node	IR Drop	Average Frequency	Masking Order	Instructions Per Cycle (IPC)	Cache Size	Timing Failure Count	Interrupt Latency (cycles)	Leakage Power (25°C)
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 minimum	> .5	≥ 16 MB	0	≤ 12	≤ 5mW

Figure 2.1: *NG-RAD: SCALER House of Quality with Legend*

2.6 Hardware & Software Block Diagrams

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:

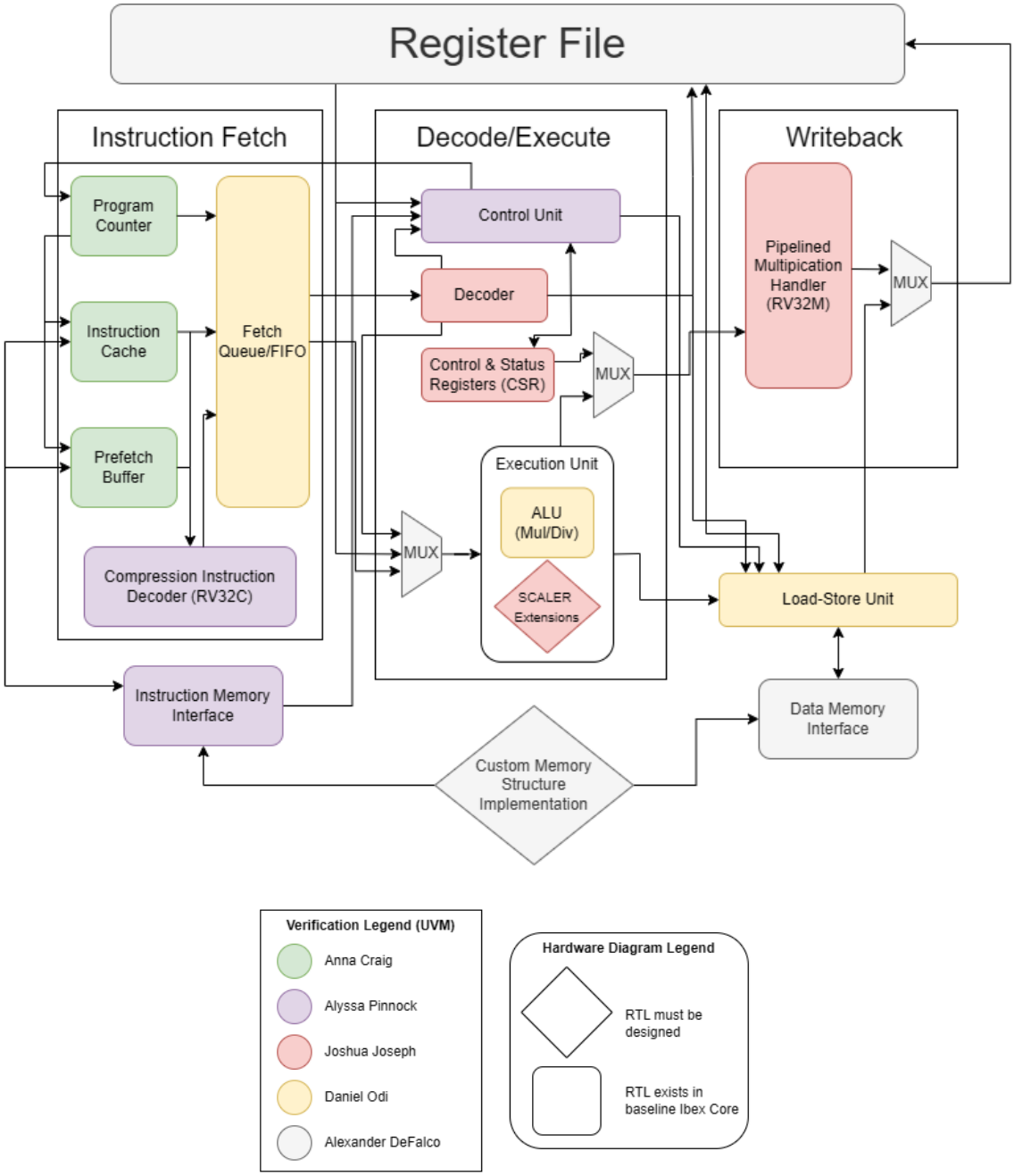


Figure 2.2: *Ibex Core diagram with color-coded assignments*

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.

2.7 Compiler/Interpreter Software Diagram

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:

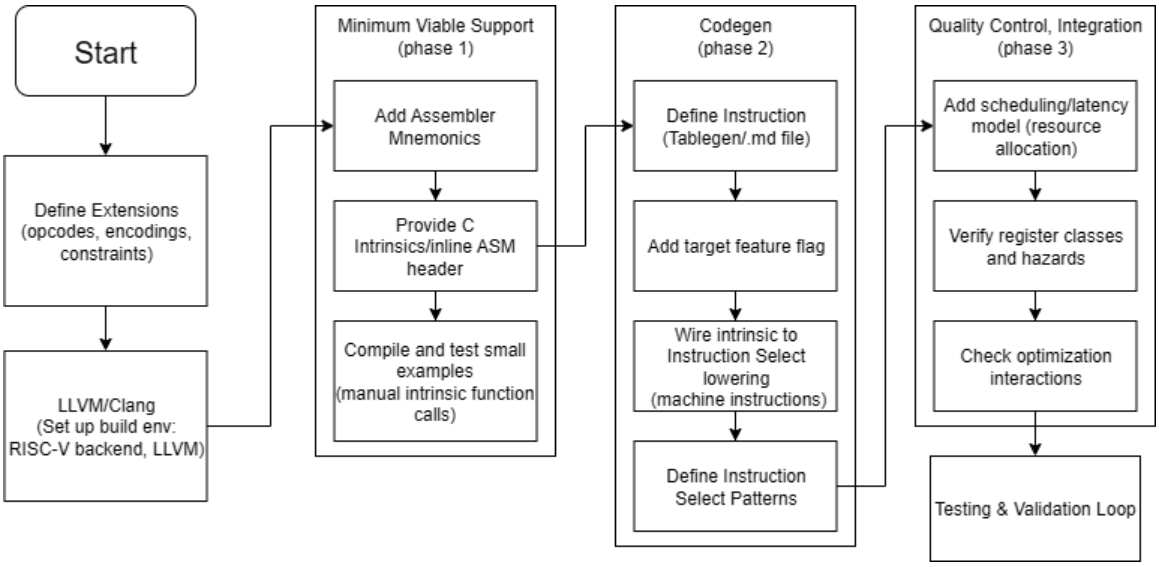


Figure 2.3: *RV32 Compiler Development Flowchart*

2.8 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.

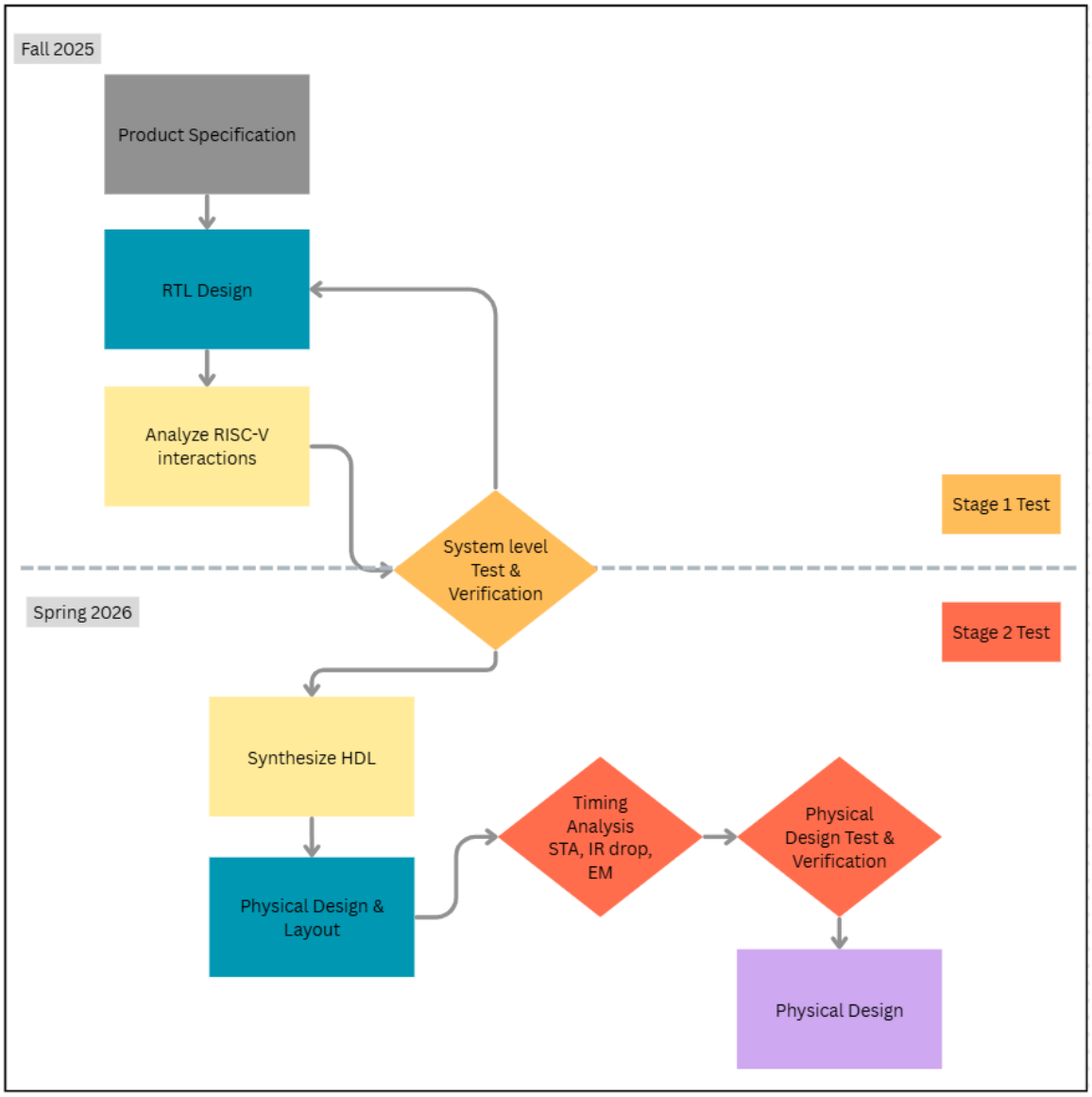


Figure 2.4: *ASIC Design Flow Process*

Chapter 3 - Research and Investigation

3.1 ASIC Design Flow

The Cadence ASIC design flow will be utilized to follow a structured plan for creating our project. The design flow will allow for transformation of a chip design from initial specs to the final design, along with describing the general verification steps within the process. This structured process moves from the high-level plan and description for the chip design to detailed RTL models, which is the core of synthesis and physical design.

The complete ASIC design flow includes the core steps of taking the specifications, developing the microarchitecture for the design, mapping the RTL, synthesis, place-and-route, signoff, and fabrication.These stages form the foundation of a successful project, and the steps must be followed carefully due to the risk and cost of any unresolved errors that may be found later in the process. Many digital design teams rely on frameworks such as the Cadence IC Design Suite, where tools like Xcelium are essential for simulation and verification.

3.1.1 Specifications

The first core step within the ASIC design process is the definition of specifications for the design. These specifications lay out the expectations for performance and functionality of the resulting chip, decided by the customer in addition to design and verification engineers. During this phase, the chip's constraints must be defined clearly and each remaining stage during this process will work together to achieve the required specifications.

Core functional requirements for the ASIC design are defined in the specification document. Some of these functional requirements can include a description of the protocols, behavioral details, and external interfaces. Additional constraints that are essential to include in the specifications might be performance, power, and area constraints. These specifications outline any design trade-offs that may need to be determined later in the process. Designers need to specify the maximum operating frequency, allowed latency, required throughput, etc. These constraints become the baseline for the synthesis constraints, and these are later evaluated against the simulation files.

These specifications also define what should be verified and how it should be verified, which is why planning the constraints is a core part of the design cycle. Each functional requirement should be linked to a verification activity to ensure that all specification needs are being met throughout the design process.

The final specification document should capture both the functional and non-functional requirements, in addition to a plan for verification. This document should be referenced throughout the design process to ensure that the main design goals are being met.

3.1.2 Microarchitecture

After defining the required design specifications, it is necessary to create an outline for the microarchitecture of the design. The microarchitecture breaks up the specs into pipelines, units, and datapaths that can be converted into a block diagram. The functionality of these separate subsystems should each be clearly defined and converted from the design constraints defined in the specifications.

The microarchitecture stage also defines the intercommunications between blocks, along with the bus communications between blocks and other components. Input and output signals for each block would be included in the microarchitecture to clearly outline how they should receive and output data or signals. A table with values needed to run the block are also included in the design of a microarchitecture block.

This step in the design bridges the gap between the abstract specifications and the RTL design implementation, where a clear layout of the specifications within the design architecture is needed before implementing the RTL code.

3.1.3 RTL Design

The Register Transfer Level (RTL) phase is one of the core steps in the ASIC design process. The system design occurs here based on the previous specifications, and this code will be later verified and synthesized later in the design process.

RTL describes how a digital circuit operates, and the specific flow of signals between registers. This step represents how the specifications and microarchitecture will translate to the final design implementation. The RTL is created by design engineers, and is based on both the microarchitecture and specifications of the chip design.

In the microarchitecture, the design was outlined with block diagrams and bus signals that would route the blocks to other components. Each of these blocks are coded separately with HDL based on the needs of that block. The code for each block would be separated into different files, then joined as modules into another file that connects the functionality of the blocks together.

3.1.4 Verification

Verification is implemented throughout various stages in the design process, and is necessary to ensure that the design functions as desired. The first phase of simulation includes system simulation, which verifies at the behavioral or system level. Logic simulation occurs at the RTL level of the design process, before gates are synthesized. After the gates are synthesized, then gate-level simulation occurs for simulating the discrete logic. Physical verification includes design rule checking, and layout versus schematic checking. Formal verification is implemented for logical equivalence checking, and timing signoff is for static timing analysis near the end of the design process.

The strategies for verification and simulation can be complex and intricate, since it's crucial to properly verify the design before tapeout and fabrication. One method of verifying a design requires verification engineers to create environments that will test and verify the behavior of the system design with testbenches. Developing testbenches involves having a complete understanding of how the system works, and any exception cases that need to be tested.

Cadence Xcelium is utilized by many engineers to run RTL simulations and properly verify their designs. Testbenches are also built using the Universal Verification Methodology (UVM) framework to automatically check the outputs of their tests. A verification engineer also defines whether specific constraints defined in the specifications have been tested or not, which is crucial to detecting any possible hidden flaws in the design. The design needs to be thoroughly tested and verified so that there is little room for error.

3.1.5 Implementation

Implementation runs in parallel with verification throughout the design. This stage of the design flow refers to the translation of RTL into gates, placement, and preparation for the final layout at the end of the process. These implementation steps follow the main synthesis process, where RTL is transformed into a gate-level netlist that can be placed and routed into a physical design tool. The steps need to align with the original provided constraints within the specification phase, such as timing, area, and power requirements.

One of the challenges with implementation is considering the size and complexity of the design, where it may take longer to complete the process if the design is larger and more complex. Power requirements are also important to be addressed at different levels during implementation to meet the power requirements.

3.1.6 Logic Synthesis

The first step in implementation is logic synthesis, where the RTL code is translated into discrete logic gates. Cadence Genus is used to create a netlist that maps the design into standard cells from the technology library. During this step, constraints such as maximum frequency, power budgets, and area limits are applied

to guide optimization. The synthesis tool creates a report that highlights the area usage, timing paths and any potential violations. If the constraints aren't met, then adjustments should be made to the RTL or synthesis constraints.

Synthesis can also include Design-for-Test (DFT) insertion, where scan chains and test structures are added to make the chip testable after fabrication. DFT also ensures that the final manufactured chip can be validated on silicon without requiring complete functional testing of every possible state.

3.1.7 Floorplanning

After a netlist is created from synthesis, floorplanning defines the physical layout and partitions of the design on the chip. Blocks like memory and IPs are placed, I/O pins are defined, and power distribution networks are created. Floorplanning is the layout that will be utilized for placement and routing later in the process. Including a well-structured floorplan is essential to prevent issues later in the design process.

3.1.8 Place and Route

Placement occurs in the next step, which places standard cells from the synthesized netlist into the floorplan. The provided placement tool optimizes the design and can reduce congestion while meeting the timing requirements. After placement, Clock Tree Synthesis (CTS) occurs to evenly distribute the clock signal across the design.

Once placement and CTS is performed, routing of the proper interconnects is carried out. Detailed routing involves various layers of routing nets to their specific corresponding channel segments. After this final step in the PnR step, design rule checking (DRC) is performed to ensure that the design fits timing, power, and other critical constraints for the functionality of the design.

3.1.9 Static Timing Analysis

One of the last steps in the ASIC design process is timing signoff along with Static Timing Analysis. Static Timing Analysis (STA) is performed throughout the implementation process to ensure that the design is meeting timing constraints. STA is the preferred method for timing signoff, and is used to determine whether timing goals of the design are met. If not, then the RTL may be modified before moving on. STA is also used during synthesis to both select the correct implementation and confirm whether that implementation achieves the desired results.

3.1.10 Tapeout

Tapeout is the final step in the design process, where the finalized and completed design is sent to a foundry for fabrication. The design is condensed into a GDSII format, and allows the layout to convert from design to production.

3.2 Verilog Hardware Description Language for ASIC Design

For this ASIC Design Project, it is most fitting to use the Verilog hardware description language (HDL). As part of the ASIC design process, it is important to understand the background of Verilog and why it is useful for the purposes of this project.

For a long time, there was no standardized computer language to describe digital circuits. In 1985, companies such as IBM and Texas Instruments collaborated to release the first version of VHDL (Very High-Speed Integrated Circuit Hardware Description Language). This push for development was led by the US Department of Defense around 1981. VHDL was used for a few years before Verilog HDL was created. Originally developed by Gateway Design Automation and later standardized as IEEE 1364, Verilog gained traction in the commercial semiconductor industry. Verilog is a less strongly typed HDL, with features and syntax similar to C programming language. Verilog was designed to be much faster and efficient than VHDL. Verilogs syntax offers more flexibility, but can also be more prone to errors if not handled carefully. The C-like syntax of Verilog also makes it more familiar to software developers, eliminating some of the learning curve [14].

In practice, Verilog has become the dominant HDL in the U.S. commercial ASIC and FPGA design market, while VHDL retains popularity in academic and European contexts, as well as in military projects. These factors support the choice to use Verilog in this ASIC design project.

3.2.1 Purpose of a Hardware Description Language

HDL's are used to model, design, simulate, and synthesize complex digital circuits before committing to physical hardware. This process allows for improved productivity, facilitating automation, and enabling verification and reuse of designs. In this project, it is important to extensively simulate and verify the features implemented. Using an HDL allows for the circuit to be testable, editable, and verifiable before it is ever physically implemented.

HDL's allow for behavioral, register transfer, gate, and switch level logic. This allows designers to define these levels in detail. The language allows for engineers to design digital circuits that meet a specified set of requirements, simulate those designs, verify them, and synthesize them.

It is for these reasons Verilog is chosen as the language for this project's ASIC design, as it encapsulates much of the requirements and goals set out for this design.

3.2.2 Key features of Verilog

Behavioral Modeling: Behavioral modeling describes what the hardware should do, without explicitly detailing how it is built. This aspect of Verilog is useful for abstract, high-level design. Engineers can write algorithms that look almost like

software, but are later transcribed into hardware. An example of this would be designing a rule that follows "Every time the clock ticks, update x to match y."

Structural Modeling: Structural modeling describes hardware in terms of how individual components are connected, much like drawing a schematic. The purpose of this is to give the designer control over how modules and gates are wired together. This aspect is closest to physical hardware. In Verilog, the designer can assign which wires it wants connected to which gates, using its syntax.

Dataflow Modeling: Dataflow modeling describes how data moves and transforms between signals, often using continuous assignments. Dataflow modeling is how combinational logic is designed in a module, rather than sequential steps. This would be when a designer uses logic to describe adders, multiplexers, or arithmetic operators.

Concurrency: In Verilog, multiple processes run at the same time, just like signals in actual hardware. This is essential for modeling real-world circuits, since hardware does not execute line by line like software does. Unlike other languages, variables can update simultaneously, which is important when modeling systems that mimic physical hardware.

Hierarchical Design: Hierarchical design is breaking a large system into smaller modules that can be reused. Large CPU designs can be made up of reusable blocks like adders, registers, and multiplexers all combined together. Instead of rewriting the logic for every bit in a ripple-carry adder, a small 1-bit adder can be designed, then reused and connected to other copies of itself [15].

3.2.3 Verilog in the Design Workflow

In the context of ASIC design, Verilog plays a role in every stage of the workflow. The process begins with the design entry, where the desired functionality of the circuit is written in Verilog code. This code can describe both the high-level behavior of the system, and its low-level implementation details.

Next, simulation ensures that the design behaves as expected before it is synthesized into hardware. During simulation, test benches written in Verilog allow us to apply inputs and monitor outputs, verifying that the design meets requirements.

Once verified, the design is passed through synthesis, where Verilog is translated into a gate-level netlist. This step transforms human-readable code into a form that can be implemented in silicon.

Finally, in implementation, the synthesized design is mapped into either an FPGA for prototyping, or prepared for fabrication as an ASIC.

For this project, this workflow is a critical aspect. By writing the SCA security extension for the Ibex core in Verilog, we gain practical experience in each step: coding the feature, simulating its behavior, synthesizing it into a form compatible with the core, and eventually verifying its integration into the larger ASIC design.

3.2.4 Applications of Verilog

Verilog is applied broadly across digital design, from simple logic circuits to full-scale processors. Some of these applications include ASIC Development used

in commercial CPUs, FPGA prototyping for implementing features into physical implementations, processor extensions for integrating new modules into an existing core, or even verification and testing to validate a design.

In the scope of this project, we are augmenting the IBEX RISC-V core with a security feature for Side-Channel Attacks. Verilog allows us to directly describe the additional logic needed to mitigate side-channel attacks while ensuring compatibility with the existing core. Through simulation and verification, we can confirm that the new feature strengthens the system and achieves the required engineering specifications, while not compromising its baseline functionality.

3.2.5 Advantages and Limitations

There are many advantages to using Verilog in this project. Verilog is widely used in the U.S. semiconductor industry, making its skills highly transferable for engineers who are capable with it. Its C-like syntax allows it to be easier to learn for individuals with software experience, reducing the barrier to entry. Verilog is scalable as it is effective for small designs like finite state machines, as well as large-scale systems like CPUs. Verilog is also supported by major synthesis and simulation tools, ensuring its compatibility in ASIC projects.

With any HDL, there are certain limitations to its applications and functionality. Verilog is described as "weakly typed," meaning its flexibility in syntax could lead to errors if the designer is not careful throughout the code. Its application in analog and mixed-signal modeling is limited when compared to digital systems. Additionally, large designs can become difficult to read and maintain without strict modularity.

For this project, these trade-offs are important. Verilogs strengths in rapid modeling, simulation, and integration make it the right choice for implementing a security feature on the IBEX core. At the same time, awareness of the language's limitations will remind us to enforce careful design practices to avoid errors.

3.2.6 Importance of Verilog in the Industry

Verilog remains one of the most important skills for engineers entering the fields of digital design, ASIC development, and FPGA prototyping. Modern companies rely on Verilog to design and verify everything from microcontrollers to advanced AI accelerators.

Understanding Verilog provides direct exposure to how hardware design, requires concurrency, timing considerations, and the ability to think at multiple abstraction levels. By mastering Verilog, students and engineers are prepared to contribute to the full ASIC workflow, from coding through fabrication.

For this project, this knowledge directly applies to implementing a Side-Channel Attack (SCA) security feature. Not only are we learning how to extend an existing IBEX RISC-V core, but we are also gaining hands-on experience with the industry-standard methods for designing, verifying, and synthesizing ASIC modules. This prepares us for careers in both academia and industry, where hardware security is increasingly critical.

3.2.7 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:

3.2.7.1 Rocket Chip (CHIPS Alliance)

The Rocket Chip generator [16] 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 [17]. 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 provides 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.

3.2.7.2 NEORV32

NeoRV32 [18] 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.

3.2.7.3 XiangShan

XiangShan is a high-performance, out-of-order, superscalar RISC-V CPU design developed by the Institute of Computing Technology, Chinese Academy of Sciences [19]. 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 [20], 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 [21]. 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 [20] and Meltdown [3]). 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.

3.2.7.4 PicoRV32

PicoRV32 [22] 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.

3.2.7.5 Ibex

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 [23].

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.

3.2.7.6 SERV

SERV (the SErial RISC-V core) is an extremely compact, bit-serial RISC-V CPU core [24] [1]. 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.

Table 3.1: *Core Feature Comparison Table*

RISC-V Core Selection Comparison					
Core	**µarch**	**Language Implemented In**	**Verification Maturity**	**Strengths**	**Drawbacks**
Rocket Chip (Rocket CPU)	In-order, single-issue (RV32/64)	Chisel (generates Verilog)	Mature via Chipyard regressions; widely used in research & education	Strong SoC integration; caches/MMU; scalable ecosystem	Chisel abstraction layer; large codebase; overkill for lightweight mods
NEORV32	In-order, 32-bit MCU-class	VHDL	Good self-tests / examples; emphasis on precise exceptions	Beginner-friendly; safety / predictability; fits small FPGAs	VHDL (team prefers SV); no MMU; not Linux-class
XiangShan	Out-of-order, superscalar, speculative (RV64GCV)	Chisel	Active research CI/sims; comprehensive micro-arch testing	High performance; multi-level caches + MMU; vector support	Very complex; heavy deps; partial English docs; Chisel abstraction
PicoRV32	In-order, minimalist RV32IMC	Verilog-2001	Community-mature; straightforward FPGA test flows	Tiny area; high F_{max} on 7-series; easy integration	No caches / MMU / privileged modes; not Linux-class
Ibex	In-order, 2-stage RV32IMC (configurable)	SystemVerilog	Strong UVM env; formal proofs; OpenTitan DV	Clear SV RTL; PMP support; flexible configs; security-minded	No MMU/caches; modest perf; not Linux-class

3.3 RSA/AES Encryption

3.4 EDA Tool and Technology Comparison

Part of the design process for any RISC-V extension is choosing what technologies to utilize. The project includes the guarantee of Cadence Design Suite licensing. Though this technology meets many industry standards, it is important to compare alternate technologies that may pertain to steps throughout the full design process. Additionally, it is important to understand in depth the requirements constraints and how they affect system performance.

3.4.1 Cadence Design Suite

Cadence provides one of the most widely adopted EDA flows, integrating tools such as Xcelium for simulation, SimVision for waveform analysis, Genus for synthesis, and Innovus for place-and-route. Its strength lies in the seamless integration between tools, making it easier to move from RTL verification through to final GDSII. Xcelium supports advanced verification methodologies such as UVM, which is particularly relevant when verifying designs intended to mitigate side-channel attacks. Innovus automates physical design with strong timing closure features, reducing the complexity of floorplanning and routing. The primary drawbacks of Cadence are its high licensing cost and steep learning curve, though these are outweighed in many academic and industrial settings by its reliability and breadth of features.

3.4.2 Synopsys Tools

Synopsys offers an alternative industry-standard toolchain, with Design Compiler as a flagship synthesis tool, VCS for simulation, IC Compiler II for place-and-route, and PrimeTime for static timing analysis. These tools are regarded as benchmarks in their respective domains, particularly Design Compiler and PrimeTime, which are often favored for timing-critical and large-scale designs. Synopsys tools are modular, requiring more expertise to integrate effectively compared to Cadence's unified environment, but they offer strong performance optimizations in terms of area, power, and timing. Like Cadence, Synopsys tools are costly, but they remain a staple in industry flows due to their high-quality results [25].

3.4.3 Mentor / Siemens EDA

Mentor, now part of Siemens EDA, is best known for its strengths in verification and physical verification. Questa is a robust simulator that supports UVM and formal verification methods, making it valuable for confirming functionality in complex designs. Calibre is the industry standard for design rule checking (DRC) and layout versus schematic (LVS) verification, and it is often used alongside Cadence or Synopsys flows as a final signoff step. While Mentor also offers Olympus-SoC for place-and-route, it is

less commonly adopted as the primary synthesis and implementation flow compared to Cadence or Synopsys. Its tools are often used in complement rather than replacement, and adoption in academic projects is less widespread.

3.4.4 Open-Source Tools

Open-source alternatives such as Yosys for synthesis, Verilator for simulation, GTKWave for waveform viewing, and OpenROAD for automated place-and-route have gained attention in recent years. They significantly reduce the barrier to entry because they are free and accessible, and they have enabled projects such as the Google/SkyWater open-source PDK initiative. These tools are particularly useful for education, research, and prototyping at mature nodes such as 130nm. However, they are less mature than their commercial counterparts, with limited support for advanced verification frameworks, fewer PDK options, and less predictable optimization during synthesis and physical design. Although open-source tools are promising and continue to improve, they have not yet been considered practical replacements for Cadence or Synopsys in advanced ASIC design flows.

In summary, while there are a range of EDA technologies with their own strengths and trade-offs, Cadence Design Suite remains the most practical choice for this project. Its integrated toolchain, strong support for verification standards, and established role in both academia and industry provide a reliable foundation for developing and validating ASIC designs. Using Cadence ensures that the project can focus on its core goal of exploring side-channel attack mitigation while relying on a proven and complete design environment.

Table 3.3: *EDA Tool Comparison Table*

EDA Tool Comparison & Selection				
Tool	**PDK Compatibility**	**Learning Resources**	**Full ASIC Design Flow**	**Timing, Power, QoR**
Cadence Design Suite	✓	✓	✓	✓
Synopsys Tools	✗	✓	✓	✓
Mentor / Siemens EDA	✓	✓	✗	✓
Open-Source Tools	✗	N/A	✗	✗

The EDA Tool comparison table offers 4 primary considerations when selecting which EDA tool is best fit for this project. This helps visualize why Cadence Design Suite was the primary option for this projects purposes.

3.5 Ibex Verification Specifics - OpenTitan

OpenTitan is an open-source silicon Root of Trust (RoT) project [26] hosted by lowRISC CIC with major contributions from organizations such as Google, ETH Zürich, G+D Mobile Security, Nuvoton, Seagate, Western Digital, and others. The goal is a trustworthy, vendor-agnostic RoT that any chipmaker can adopt. OpenTitan pursues this by publishing its RTL, firmware, and verification collateral openly, implementing strong logical integrity guarantees in hardware/firmware, and protecting the "OpenTitan" name via a trademark policy and licensing that requires conformance to the project's standards.

As an R&D platform, OpenTitan demonstrates state-of-the-art, openly verifiable security within silicon: a modern RoT microcontroller, "Earl Grey", built around the Ibex RISC-V core [27]. It includes comprehensive documentation and workflows for UVM-based DV (with full-coverage Ibex verification and RISC-V-DV based checking). Its openness enables external auditing, reproducibility, and community contributions across architectures, firmware, and verification.

3.6 Design Parameter Investigation

3.6.1 Masking Order

Masking is one of the most widely used countermeasures against side-channel attacks (SCAs), which exploit correlations between physical leakages (such as power consumption or electromagnetic emissions) and secret data. The general principle of masking is to randomly assign intermediate variables in a cryptographic computation so that no single intermediate value reveals information about the secret. A sensitive variable is divided into several random shares, and operations are performed on these shares independently. The number of random shares determines the masking order, which defines how many leakage probes an attacker would need to successfully recover secret data. There are several types of masking techniques, each tailored to different cryptographic implementations which will be detailed in the following sections [28].

3.6.1.1 First-Order Boolean Masking

First-order Boolean Masking focuses on mitigating the exploitation of first-order leakages, such as in Correlation Power Analysis, by representing the secret data as two shares. This is achieved by combining the data with random values using an XOR operation and operating on the newly generated shares. When computations are complete, these shares can be used to rebuild the original secret. With new representations of the secret data, certain operations must be modified to preserve the masking relation: applying the operation independently to each share produces a correctly masked output (so that the original data can be derived from the shares).

Linear operations, such as XOR, shift, or rotation, are straightforward because linear functions preserve the masking relation: applying the operation independently to

each share produces a correctly masked output. Nonlinear operations, however, require additional care. For instance, an AND or multiplication between two masked variables involves cross-terms that can mix shares, which could leak sensitive information. To handle this, researchers have created secure versions of AND and OR gates comprised on standard AND, ORs, and INVs to hold up the masking relation and ensure that the output shares remains independent of the secret given any single observation of the circuit. These secure gates typically consume additional random values at runtime to refresh the masking and maintain statistical independence. Though, work by Biruykov found that optimal versions of secure AND (SecAND) and secure OR (SecOR) gate that eliminate the need for random numbers in the operation[29]. This method implemented on the SERV Core along with other masking and hiding techniques showed efficient mitigation of Correlation Power Analysis Attacks [1].

First-Order Boolean Masking can be expanded to multiple orders, where each additional masking stage corresponds to an increased security level. For example, a first-order masking system (d = 1) protects against single-point leakage attacks, while a second- or third-order system extends protection to attackers capable of correlating multiple measurements. However, higher-order masking introduces performance trade-offs, including increased latency, silicon area, and random number generation requirements. Additionally, in general, Boolean masking can be imperfect when realized in physical hardware. Glitches in the hardware may cause temporary correlations between shares reducing the effectiveness of the masking. These glitches occur when signals propagate through logic gates with different delays, causing unintended intermediate combinations of masked values. Thus, the masking order must be carefully balanced against processing speed and hardware resource constraints to ensure the design meets both performance and security specifications.

In this project, bit-level masking is an essential tool to improve the security features of the core. By incorporating bit masking directly into the processor's instruction set or hardware data path, sensitive computations can be randomized at the lowest operational level, reducing the likelihood of side-channel leakage. Previous research has outputted actual implementations of boolean masking on RISC-V cores by synthesizing the design and then using scripts during the syn_map to replace standard gates with secure versions of the gates that use boolean masking [1].

3.6.1.2 Higher-Order Boolean Masking

Boolean Masking can be extended to mitigate higher-order attacks by increasing the number of shares used to represent the data. For a n^{th}-order attack, $n + 1$ shares must be used to represent the data. Higher-Order masking falls into the same pitfalls as Boolean Masking where it leakages can happen as a result of glitching as well as increased area, power, and gate counts as the number of shares to represent data increases.

3.6.1.3 Arithmetic Masking

Arithmetic masking uses modular addition and is better suited for algorithms involving arithmetic operations that rely on modular addition and multiplication over bitwise operations, such as RSA or elliptic curve cryptography. Thus, it is better than boolBooleanking in these instances. In some designs, hybrid masking combines Boolean and arithmetic techniques, with secure conversion circuits used to transition between the two representations. Similarly to Boolean Masking, these methods can be used for protection against more sophisticated attacks by increasing the number of shares.

Although integrating Arithmetic and Boolean Masking for hybrid methods does seem promising, current methods have only been showcased on less optimal implementations of Boolean masking as it requires more fresh random numbers which have a much larger overhead[30, 31]. Thus, other methods that detailed in later sections were explored and chosen as they were easier to integrate with Biruykov's [29] optimal boolean masking techniques.

3.6.1.4 Threshold Implementation

Threshold Implementation (TI) is Boolean Masking and enforcing the properties of correctness, non-completeness, and uniformity.

Correctness ensures that when all shares are combined (for example, by XORing them), they produce the correct result as if the operation had been done on the unshared data. This guarantees functional equivalence to the original circuit.

Non-completeness means that no single function or logic component within the circuit may depend on all the shares of any sensitive variable. In other words, every gate or sub-function only operates on a subset of the shares. This property ensures that even if glitches occur, they can never combine all shares of a secret in one place.

Uniformity ensures that all shares are distributed uniformly at random, regardless of the original secret value. This prevents statistical bias that could leak information to an attacker. Achieving uniformity often requires the use of additional random values or correction terms.

Overall, Threshold Implementation provides a systematic and provably secure method to design side-channel-resistant hardware as it offers strong protection even in the presence of glitches making it a reliable countermeasure for secure VLSI cryptographic implementations. The drawback is it increases latency and area of circuits, where classical multipliers that require 4 ANDs and 4 XORs now need 13 ANDs, 12 XORs, and three registers when trying to abide by the properties of Threshold Implementation[32].

3.6.1.5 Domain-Oriented Masking

Domain-Oriented Masking aims to prevent leakage from hardware boolean-masking implementations due toult of glitches. These glitches can cause different masked shares to interact within a single gate, unintentionally recreating dependencies on the secret data and thereby re-introducing exploitable side-channel leakage. Domain-

Oriented Masking was proposed to eliminate this weakness by redesigning how masked computations are organized and synchronized at the hardware level.

Similar to First-Order Boolean masking, data is represented as shares, but DOM takes it a step further by classifying these shares into domains where each is responsible for processing one masked share. Each domain contains its own logic and operates only on its assigned share, preventing simultaneous data-dependent switching between multiple shares. Prioritizing the output of intermediate values to come from operations inside each of the domains that make up the data. Though this cannot be applied to non-linear functions that require shares from different domains to interact. This is accomplished by adding fresh random numbers into the computation and using registers to prevent glitching that can happen when operating with values from different domains [32].

In the context of RISC-V processors, DOM is often integrated into custom cryptographic co-processors or instruction extensions, providing a hardware-level safeguard that complements higher-level masking or hiding defenses. By combining theoretical rigor with practical feasibility, Domain-Oriented Masking represents a critical step toward robust, physically secure computing architectures resistant to power and electromagnetic side-channel analysis.

Table 3.5: *Comparison of Masking Techniques for Side-Channel Attack Resistance*

Comparison of Masking Techniques				
Technique	**Leakage Resistance Level**	**Glitch Resistance**	**Hardware Overhead (Area/Power)**	**Randomness Requirement**
First-Order Boolean Masking	First-order (protects against single-point leakages)	✗	Low to Moderate	Moderate (requires fresh randoms for secure gates)
Higher-Order Boolean Masking	Up to n^{th}-order (depending on number of shares)	✗	High (scales with number of shares)	High (more random shares needed)
Arithmetic Masking	First- to higher-order (depends on share count)	✗	Moderate to High	High (requires many fresh randoms, especially in hybrid schemes)

Continued on next page

Comparison of Masking Techniques (continued)				
Technique	**Leakage Resistance Level**	**Glitch Resistance**	**Hardware Overhead (Area/Power)**	**Randomness Requirement**
Threshold Implementation (TI)	High (provably secure against glitch-induced leakage)	✓	High (increased gate count and latency)	Moderate (additional randoms for uniformity)
Domain-Oriented Masking (DOM)	High (resists higher-order and glitch-induced leakages)	✓(separates domains to prevent share interaction)	Moderate to High (extra registers and logic per domain)	Moderate (fresh randoms used for cross-domain operations)

3.6.1.6 Implementation of Masking Techniques

For each of these masking techniques, the prior sections give a high-level overview of each technique, it's advantages and drawbacks. Another factor to assess how these masking techniques are implemented in open-source RISC-V designs.

The first method would be to rewrite RTL in existing RISC-V designs. In this approach, the designer explicitly instantiates masked logic modules at the register-transfer level. Sensitive operations such as AND, OR, and XOR are redefined in their masked forms. Linear operations, such as XOR, can be applied independently on each share. This method offers full design-time control and transparent functional verification. The main drawback when implementing this method. Having full control of the design results in the task being significantly more intensive. Once this is achieved, designers face new challenges in running their flows through the EDA tools. During synthesis, EDA tools optimize the generic netlist based on the given RTL and this can result in netlist that are optimal for certain metrics outside of security. For example, a design's netlist may optimize a portion of secure operation where it reduces the gate count but results in the operation now being insecure and capable of leakages that can be taken advantage of for SCA.

The second method is to make changes to the design post-synthesis. Masking can also be introduced automatically on a synthesized netlist. Scripts identify logic cones driven by sensitive nets, duplicate them into share domains, and substitute masked cell templates from a custom standard-cell library. This can be accomplished by creating TCL scripts that can be integrated during the synthesis process. Thus, instead of having to redesign the RTL all that would be needed to create the custom secure cells using the designs in the existing cell library but also TCL scripts to be integrated into the synthesis process. However, the only pitfall to doing these gate-level

transformations is that it provides less algorithmic transparency and may obscure functional intent, making equivalence checking and verification essential. Given that this process is less labor intensive and prone to error, it was the selected method to implement masking techniques in this project

3.6.2 Hiding Techniques

The second set of techniques that will be covered are hiding techniques which like masking make extracting information from the correlation of power dissipation and data being process obscure, making it more difficult to perform side-channel analysis attacks on the design. In the case of hiding, the purpose is to obfuscate the correlation between the power dissipation and data being processed, which is used to extract secret key information during correlation power analysis. This section will cover the various hiding techniques researched, as well as the motivation behind the chosen techniques implemented in this project.

3.6.2.1 Random Noise Injection

Random Noise Injection is a hiding-based countermeasure where it works by obscuring the leakage itself that is, by increasing the randomness or unpredictability of the device's instantaneous power consumption. The goal is to reduce the signal-to-noise ratio (SNR) observed by an attacker, making it statistically difficult to correlate measured power traces with internal data transitions. Noise injection increases the background noise or variability of power traces can dramatically reduce an attacker's ability to extract secrets.

Random noise can be introduced in several ways, broadly divided into hardware-level and software-level implementations. In hardware, one common method is to integrate noise-generating circuits such as ring oscillators, linear feedback shift registers (LFSRs), or pseudo-random toggle logic that continuously switch transistors to consume additional current. This creates random variations in overall power consumption, effectively masking the smaller fluctuations caused by sensitive computations. Another hardware approach involves random current injection, where controlled analog noise sources (for example, digitally modulated current mirrors or resistor networks) are added to the power distribution network to blur measurable differences in current draw. Some RISC-V-based system-on-chips (SoCs) use dedicated noise cores that execute meaningless instructions or toggle logic lines in parallel with cryptographic operations to achieve a similar effect.

The advantages of random noise injection are its simplicity, low design complexity, and broad applicability. It can be added to both existing hardware and software systems without redesigning functional units or cryptographic algorithms. Noise injection also complements other countermeasures such as masking or dynamic frequency scaling, collectively making side-channel leakage more difficult to exploit. Additionally, it offers probabilistic protection as even if an attacker collects many traces, the added randomness makes statistical averaging less effective.

However, noise injection has notable drawbacks. The injected noise is indepen-

dent of the secret data, it does not eliminate leakage it only conceals it statistically. Given enough traces and advanced filtering or averaging techniques, an attacker may still recover the underlying signal. Furthermore, hardware noise generators and dummy operations introduce significant power and energy overhead, which can be problematic for resource-constrained embedded systems. Excessive noise also complicates on-chip power delivery and can interfere with circuit stability or timing closure.

After evaluating the aforementioned techniques to implement random noise injection, hardware level noise-injection seems to be the most feasible and optimal route to for this project. Specifically, the inclusion of extraneous logic blocks (like a random noise generator, LFSRs, and ring oscillators) as this would require minimal changes to the RTL as these new noise modules would only have to be connected and controlled with existing control signals in the processor.

3.6.2.2 Dynamic Frequency Scaling

The central idea of RDFS is to randomly vary the processor's operating frequency during execution so that the timing and power characteristics of each clock cycle become unpredictable to an external observer. Because CPA and similar attacks rely on aligning thousands of power traces to corresponding operations across multiple encryptions, randomizing the system's frequency effectively misaligns these traces, disrupting the correlation between power consumption and processed data.

In a typical digital processor, power consumption and timing are tightly linked to the clock frequency: higher frequencies produce shorter cycles and higher dynamic current draw, while lower frequencies slow operations and reduce instantaneous power. Frequency hopping exploits this relationship by introducing temporal randomness. During protected computations, the processor dynamically changes its frequency according to a pseudo-random sequence. These changes can occur on a per-cycle, per-block, or per-round basis, depending on the desired balance between security and performance. As a result, each execution produces power traces that differ in both amplitude and temporal alignment, making it significantly harder for an attacker to synchronize or average traces for correlation analysis.

RDFS can be implemented using several hardware architectures. A common method involves a programmable phase-locked loop (PLL) or digitally controlled oscillator (DCO) that can adjust the system clock in real time based on random control bits generated by a hardware random number generator (RNG) or linear feedback shift register (LFSR). In RISC-V-based SoCs, RDFS is often integrated into the power management unit or clock generator, allowing frequency modulation to occur transparently to the processor core[1].

The advantages of frequency hopping are its hardware simplicity, low area cost, and high effectiveness in misaligning traces. Unlike data-masking approaches, RDFS does not require modifying cryptographic algorithms or duplicating data paths. It can be applied globally to the processor, protecting all workloads executed during variable-frequency operation. Studies on RISC-V implementations have shown substantial increases in the number of traces required for successful CPA, sometimes by orders of magnitude, without heavy hardware overhead. Moreover, because frequency

modulation is already supported in many SoCs for power management, adapting it for security can reuse existing infrastructure.

However, RDFS also has important limitations. It primarily provides hiding rather than elimination of leakage; given enough traces and appropriate signal processing, attackers can still recover correlation. Additionally, frequent changes in frequency and voltage can introduce timing instability, potential synchronization issues with peripherals, and degraded real-time determinism. Power efficiency may also decline, as random frequency variations can prevent optimal operation points. Furthermore, too narrow a frequency range reduces security, while too wide a range may violate timing constraints or cause system errors.

Despite these trade-offs, Random Dynamic Frequency Scaling remains a promising, lightweight, and hardware-friendly approach to mitigating side-channel leakage in modern processors. When combined with other techniques such as masking or noise injection, frequency hopping provides an effective additional layer of defense that significantly complicates power analysis attacks on RISC-V and embedded cryptographic systems.

3.6.2.3 Clock Jitter

One issue with the use of LFSR and CASR is they are pseudo-random number generators which calculate based on a starting seed. This proves to be problematic as masking requires the use of fresh random numbers to keep the masked data independent from the data. To accomplish this, prior implementations have used ring oscillators whose thermal noise over time generates jitter that can be used to introduce randomness into computations which is great for the purposes of this project[1]. Outside of the generation of fresh random numbers for boolean masking, it can also be used to for the dynamic frequency scaling to randomly change the frequency of the circuit by skipping clock edges.

3.6.2.4 Random Delay Instruction Insertion

Random Delay Instruction Insertion (RDII) introduces temporal randomness into the execution of sensitive operations by inserting random delays—implemented as no-operation (NOP) instructions, dummy computations, or variable-length stall cycles. RDII effectively misaligns the timing of power traces, reducing the correlation between trace samples and specific data-dependent events.

These instructions can be inserted into compiled cryptographic routines using compiler instrumentation or inline assembly, requiring minimal hardware modification. At the microarchitectural level, RDII can be implemented through a randomized stall generator within the pipeline or memory controller that periodically halts instruction issue or data access for a random number of cycles. Hardware implementations can leverage pseudo-random number generators (PRNGs) or linear feedback shift registers (LFSRs) to determine delay durations dynamically during runtime.

RDII are simple, have low hardware overhead, and compatible with existing RISC-V designs. It provides a flexible trade-off between security and performance, as

the amount of randomness can be tuned to match system constraints. However, RDII introduces execution time variability and may not prevent attacks that use advanced alignment or averaging techniques. Additionally, inserting random operations may degrade performance and timing determinism, reducing real-time reliability. Though the main issue is that to implement RDII would require making changes to the compiler. Modifying RISC-V toolchain has proven to be difficult in a myriad of applications especially since this project is using a specific configuration of the toolchain created by the vendor of the Ibex core. In combination with the focus of this project being to gain familiarity with the RTL-to-GDSII process, focusing on solutions with RTL would be preferred thus making RDII undesirable.

3.6.3 Generating Randomness in circuits

In the hiding and masking sections, there was a frequent need for randomness and when looking at the implementation of these techniques the realization is that there is a need for consistent and fresh random numbers. The need for randomness is to integral to these mitigation techniques as randomness make it harder for adversaries to determine and take advantage patterns, functionality, and side channels happening within the circuit. This section will detail the different ways random numbers can be generated and assess which will be best suited for this project.

3.6.3.1 Linear Feedback Shift Registers (LFSRs)

Linear Feedback Shift Registers are pseudo-random number generator (PRNG) that are attractive because their low hardware overhead. LFSRs consists of a shift register whose the input of the first shift register is a linear combination of XORs of bits stored in other registers in the LFSR starting from an initial seed. If the XOR taps are set for the maximum period, for a n sized LFSR the outputs from the LFSR will appear random and the LFSR will output all $2^n - 1$ possibilities before repeating the initial seed once again. Once the initial seed is repeated, it will repeat the same set of $2^n - 1$ possibilities which is why LFSRs are considered to be pseudo-random number generators (PRNG). The period of the LFSR, the amount of numbers generated before going back to the original seed, can be made shorter by manipulating the XOR taps on the LFSR. The length of the sequence before repetition is a crucial factor in applications requiring high-quality randomness, such as in cryptographic key generation or stream cipher implementations.

The advantages of LFSRs are clear in terms of both their efficiency and simplicity. They are computationally inexpensive and can be easily implemented in hardware using simple logic gates, making them ideal for resource-constrained environments such as embedded systems or low-power devices. Moreover, calculating the next number from the LFSR are fast, and the design of an LFSR is straightforward, requiring minimal resources.

However, LFSRs have significant drawbacks, especially in the context of cryptography. The most critical issue is their predictability. Since the output of an LFSR is deterministic, if an attacker can deduce or discover the initial state (or seed) of the

register, they can easily predict the entire sequence of random numbers or keystream bits. This makes LFSRs inherently vulnerable to attacks, particularly in systems where the key or the state may be exposed to an adversary. LFSRs are also prone to known-plaintext attacks, where the attacker can derive the keystream if enough corresponding plaintext and ciphertext pairs are available. Moreover, because the output of an LFSR is a linear function of its state, it is vulnerable to cryptanalytic techniques such as the Berlekamp-Massey algorithm, which can reverse-engineer the feedback polynomial and recover the internal state of the LFSR.

In addition to these security weaknesses, the periodicity of LFSRs can pose a problem. While an LFSR can generate sequences of high length, the sequence will eventually repeat. In cryptographic applications, a short period can lead to repeating patterns in the keystream, which can be exploited by attackers to break the cipher. Even with a long period, the linear nature of the LFSR can make it easier for attackers to detect and exploit weaknesses, particularly when combined with other vulnerabilities in the cryptographic system.

LFSRs are a valuable tool in cryptographic systems, particularly for applications that require fast, efficient, and simple random number generation. However, their predicability as a result of their pseudo-random nature, lead it to be vulnerable to side channel analysis techniques. By itself, LFSRs are not attractive as something to implement in this project, but in conjunction with other techniques it proves to be promising. Specifically, when introducing entropy sources into it's number generation to increase its randomness.

3.6.3.2 White Noise Random Number Generators

White Noise Random Number Generators (RNGs) are a type of true random number generator (TRNG) that relies on physical, unpredictable noise to generate random numbers. Unlike pseudo-random number generators (PRNGs), which use deterministic algorithms to simulate randomness, white noise-based RNGs exploit inherent unpredictability in natural phenomena to produce random sequences.

The basic principle behind white noise RNGs is the use of physical noise sources that exhibit random fluctuations. White noise, specifically, refers to a type of signal that contains equal power across all frequencies, producing a signal that is essentially unpredictable over time. This randomness can be sourced from various phenomena, such as thermal noise (also known as Johnson-Nyquist noise) in resistors, shot noise in semiconductor devices, or even atmospheric noise. These sources of physical randomness can be captured using sensors or amplifiers, converted into digital form through an analog-to-digital converter (ADC), and processed to generate random numbers. This process of measuring, digitizing, and post-processing noise results in high-quality random data.

However, white noise RNGs also come with certain drawbacks that must be considered, especially in cryptographic contexts. One of the main challenges is the need for specialized hardware to generate and measure the noise. While thermal or shot noise is widely available in electronic systems, it still requires sensors, amplifiers, and ADC circuitry to convert the physical noise into usable digital data. This adds

complexity and cost to the system, making white noise RNGs less suitable for low-cost or resource-constrained applications. Moreover, the quality of the noise source itself is crucial. If the noise is contaminated by external factors or does not exhibit sufficient randomness, the resulting random numbers may contain biases or patterns that compromise security.

Another challenge is the post-processing of the noise data. Although white noise is generally random, it may still require additional processing to ensure that the output is uniformly distributed and free from any statistical biases. This adds complexity to the system and can impact throughput. For example, if the entropy of the noise source is low, the system may need to gather a larger amount of noise data before generating usable random numbers, which can slow down the overall process.

In conclusion, white noise RNGs offer distinct advantages for cryptographic applications that require true randomness. Their ability to produce random numbers based on physical noise sources ensures a high level of unpredictability, making them highly suitable for cryptographic key generation, secure communication protocols, and other security-related tasks. However, the need for specialized hardware, potential post-processing requirements, and limitations in output rate mean that white noise RNGs are best suited for high-security applications where randomness quality is paramount, and the overhead in hardware.

3.6.3.3 Ring-Oscillators

A ring oscillator is a fundamental electronic circuit where the output of the last inverter is fed back into the input of the first. This feedback loop causes the circuit to oscillate, producing a periodic waveform. This generates a periodic oscillation at a certain frequency. However, the frequency at which the oscillator operates is not perfectly deterministic. This is primarily due to factors such as thermal noise, voltage fluctuations, and process variations that affect the individual components of the circuit.

Jitter refers to the small, random fluctuations in the timing of these oscillations. In other words, it is the variation in the periods of oscillations from their expected values. The jitter in a ring oscillator is primarily caused by random variations in these environmental and physical factors. By sampling the phase of the ring oscillator at very short intervals. These samples are recorded as binary values, and the variations in their timing are used as entropy. The randomness of the jitter makes it difficult for an attacker to predict future samples or reverse-engineer the process.

Ring Oscillators don't fall for the two pitfalls as the LFSR and White Noise Generator face. Ring oscillators generate true random out without the need for specialized hardware. This doesn't make LFSRs obsolete, as ring oscillators don't solely generate random numbers but use the randomness outputted from it to make pseudo-random process more random!

3.6.4 Process Node

The process node defines the transistor dimensions and the overall device scaling of an integrated circuit. Smaller nodes generally yield faster and more power-efficient chips; however, for side-channel attack (SCA)-resistant designs, a smaller process does not automatically result in a more secure implementation. As transistor dimensions shrink, leakage mechanisms, coupling effects, and transient glitches become more complex, which can actually increase vulnerability to SCAs and fault-injection attacks.

Selecting the appropriate process node is therefore a strategic balance between security robustness, performance, cost, and manufacturability. It is one of the most critical early design decisions because it directly impacts not only speed and power consumption but also the feasibility of validating and maintaining security features. In general, smaller nodes (around 7 nm) offer greater transistor density, higher operating frequencies, and lower dynamic power consumption. These characteristics make them ideal for high-performance computing applications but introduce challenges for secure hardware. At such small geometries, signal coupling, power noise, and glitch propagation become more pronounced, making it difficult to preserve the independence of masked signals and to verify side-channel resistance. Furthermore, the extremely low signal amplitudes in deep-submicron technologies make it harder to measure and analyze leakage behavior during testing [33].

Conversely, larger process nodes, such as around 130 nm, tend to be more robust and easier to analyze for security purposes. Their higher supply voltages and wider device spacing reduce crosstalk and simplify the modeling of power and electromagnetic emissions. However, these benefits come at the expense of area efficiency and performance: larger transistors increase chip area and power consumption while limiting achievable clock speeds. As a result, while they are suitable for proof-of-concept or radiation-hardened designs, they may not meet modern performance or cost constraints for embedded cryptographic processors.

A mid-range process node provides an effective compromise between security and performance. At this scale, designers can achieve clock frequencies in the range of 500 MHz to 1 GHz, which is sufficient for RISC-V-based workloads while maintaining manageable leakage behavior and realistic side-channel evaluation conditions. Selection of process node size would entail leveraging of the pros and cons of larger vs smaller node size.

In this project, a mid-scale process node of approximately **45 nm** was selected to balance performance, security, and design practicality. At 45 nm, the device geometry supports moderate clock frequencies and competitive power consumption while maintaining sufficient signal margins for accurate leakage analysis and side-channel countermeasure validation. In addition, mature 45nm design kits and support for EDA tools enable reliable physical implementation without the extreme variability or cost of the process associated with advanced nodes, making it a strategic choice for secure, research-oriented ASIC development.

3.6.4.1 Cadence GSCLIB045

The GSCLIB045 standard cell library is a generic 45-nanometer CMOS library designed primarily for educational and research use in digital integrated circuit design. Developed to complement the Cadence GPDK045 process design kit, GSCLIB045 provides a complete suite of front-end and back-end design views that support the full ASIC design flow, including logic synthesis, placement, routing, and timing verification. While not based on proprietary foundry data, it accurately models the structure and performance of a modern 45 nm process, enabling designers and students to perform realistic design and analysis exercises within a consistent technology framework.

The library includes a wide range of standard digital cells such as inverters, buffers, logic gates, multiplexers, latches, and flip-flops. These are provided in multiple drive-strength variants (e.g., X1, X2, X4, X8) to allow design flexibility in timing optimization and power management. GSCLIB045 also offers multiple threshold-voltage options, including standard-, high-, and low-VT versions, which facilitate trade-offs between speed and leakage power. Each cell is characterized with timing, area, and power parameters, and accompanied by corresponding layout, symbol, and abstract views.

The organization of GSCLIB045 follows a modular structure, where the main library binds to a technology library (gsclib045_tech) and the underlying GPDK045 process definitions. This structure ensures seamless integration with Cadence Virtuoso and Encounter tools. Although GSCLIB045 is not intended for fabrication or production use, it serves as an effective educational and prototyping resource, allowing users to gain practical experience in the ASIC design flow. Its use is widespread in university laboratories and training environments, where it provides a balance between realistic design behavior and accessibility for academic experimentation. Thus, choosing this standard cell library is ideal for this project as it is built to integrate with the Cadence tools.

3.6.5 Operating Voltage Range

When determining the operating voltage range for a secure core, timing reliability, power integrity, and fault/attack behavior are some of many factors that need to be considered. Though this design in its current application will not reach tapeout, it is still crucial to analyze its functionalities as if it would be.

In VLSI, Process, Voltage, and Temperature (PVT) is a common analysis point to determine on-chip performance. Chips are modeled at different PVT corners in order to make them function after manufacturing in all scenarios.

When determining an appropriate operating voltage range for a chip, it's important to understand how voltage interacts with the other two primary sources of hardware variation, process and temperature, to influence circuit behavior. Voltage deviations across the die or between operating modes can significantly affect speed, power consumption, and reliability. In real systems, the supply voltage (VDD) rarely remains perfectly constant- it fluctuates due to resistive and inductive losses in the power delivery network, as well as dynamic switching activity. These fluctuations,

known as IR drop and Ldi/dt noise, cause local variations in transistor drive strength and timing. A higher VDD increases transistor drive current and improves switching speed, while a lower VDD slows operation and lengthens propagation delay. This means that if the voltage drops too far below its nominal value, certain timing paths may fail, while excessive voltage can overstress devices and increase leakage power.

Process variations amplify this challenge. Because no two transistors are exactly alike, especially at smaller technology nodes, the optimal voltage for one region of the chip might not be ideal for another. Some transistors may require slightly higher voltages to switch reliably, while others can operate efficiently at lower levels. As a result, the operating voltage range must provide enough margin to ensure that even the slowest devices on the chip still meet timing across all process corners. Designers typically define a minimum operating voltage (Vmin) based on these worst-case conditions and a maximum operating voltage (Vmax) constrained by reliability limits such as electro migration, gate-oxide stress, and thermal dissipation.

Temperature variations further complicate voltage selection. As a chip heats up during operation, transistor mobility decreases while threshold voltage also drops, changing the balance of performance and power. At high temperatures, a slightly higher supply voltage may be needed to compensate for slower switching, whereas at low temperatures, the same voltage can cause excessive current and potential overstress. For this reason, the operating voltage range must remain stable across the full temperature spectrum expected in the device's environment.

Increasing the voltage can boost the processor's operating frequency, as it provides more drive strength to the transistors, which in turn reduces switching delays. However, this comes at the cost of higher power consumption, which is a major design consideration.

Ultimately, defining the correct operating voltage range is a process of balancing timing robustness, power efficiency, reliability, and security. Too narrow a range risks functional instability under real-world fluctuations, while too wide a range increases design complexity and vulnerability to side-channel or fault-injection attacks. Establishing a tightly controlled voltage band, helps to ensure that the system performs predictably and securely under all process and temperature conditions [34].

3.6.6 IR Drop

IR drop refers to the voltage drop that occurs as current flows through the resistive elements of the power distribution network (PDN). For a side-channel-attack extension that implements high order masking techniques, the power distribution network becomes a security asset as well as a functional resource. IR drop is a crucial aspect to consider when designing a security feature because it can disrupt the reliable operation of masked logic. Voltage fluctuations across the chip may cause logic errors or timing variations that compromise the independence of masked signals. Although masking is typically applied at the circuit or instruction level, its secure and reliable execution depends on stable hardware operation, which can be degraded by IR-drop-induced voltage instability.

Masking works by splitting sensitive data into multiple random "shares" that

should operate independently. If one share's power rail drops more than another, the transistors will switch at different speeds or draw different currents. This imbalance can cause correlated power leakage between the shares, allowing an attacker to detect patterns in the power consumption that reveal information about the secret encryption. In other words, voltage instability directly weakens the masking and increases side-channel vulnerability.

For this reason, secure hardware typically requires tighter IR-drop limits than conventional digital designs. While standard logic circuits can tolerate larger voltage drop margins, masked or cryptographic sections should ideally stay below 4-5%. Maintaining a more stable voltage ensures all masked shares behave consistently, preserving statistical independence and reducing the chance of data-dependent leakage.

Several design techniques can help reduce IR drop in the physical layout of a secure chip:

- Dedicated power rails or domains for masked logic, preventing interference from high-activity functional blocks
- On-chip decoupling capacitors placed near sensitive regions to smooth fast transient voltage dips caused by switching activity
- Low-impedance routing, achieved by using wider metal traces and multiple vias to improve current delivery and reduce resistive losses
- Voltage monitors and protection circuits that detect large droops or spikes and can halt cryptographic operations before leakage or data corruption occurs

3.6.7 Timing Failure

Timing failures occur when a signal in a digital circuit arrives too early or too late to be captured correctly by the next logic stage. These can result from IR drop, temperature variation, process variation, excessive logic depth, or clock jitter. In security-critical designs, even small timing inconsistencies can be exploited. Because every change in delay affects the moment when current is drawn, such fluctuations can leak information about the data being processed. This link between circuit timing and power behavior makes cryptographic hardware especially vulnerable to timing attacks—a form of side-channel attack in which an adversary infers secret information by analyzing execution time or power traces [35].

Timing attacks are a category of side-channel-attacks that target indirect information leaked during computation such as timing variations and power consumption. Timing attacks exploit the fact that many cryptographic operations take slightly different amounts of time depending on the input data or the secret key.

To mitigate timing-based attacks, a combination of algorithmic, microarchitectural, and physical-design countermeasures can be used:

- Constant-time algorithms: These algorithms ensure that all execution paths take the same amount of time regardless of input or key values. This means no data-dependent branches, early exits, or variable-length loops. Examples include using Montgomery multiplication or RSA blinding, which randomize computation order while maintaining fixed latency

- Instruction and data flow balancing: This ensures that all operations, including memory access, occur in a uniform sequence and timing. Also, avoids data-dependent cache behavior by disabling caching for sensitive routines
- Randomization techniques: Adding controlled randomness to execution timing to obscure correlations. This includes random instruction insertion, dummy operations, or randomized clock periods

3.6.8 Operating Frequency

Clock frequency influences power behavior, timing alignment, and leakage characteristics in digital circuits. A higher frequency equates to more switching events per second, which affects the shape and amplitude of the power trace. At lower frequencies, each operation lasts longer and produces a clearer, more distinct power signature. This can make it easier for an attacker to separate individual operations or correlate power spikes with cryptographic steps. In contrast, higher frequencies cause operations to overlap more, compressing the power trace and making it appear noisier. This can make simple visual analysis harder, though not necessarily more secure.

However, a noisy trace is not automatically more secure. Even when peaks blur together, the leakage may still correlate with the processed data. A higher clock frequency may blur visible peaks but can still leak through aggregate correlation across many traces. Research has shown that clock jitter may be observed and measured, then converted into an analog signal for analysis. This would allow adverse personnel to gain information on secret data. A study showed that "a secret key in a target chip running the Advanced Encryption Standard (AES) crypto algorithm could be extracted in around 50k measurements. That is considered strong leakage in the side channel research community." [36] When selecting the operating frequency for a secure processor core, both performance and side-channel resilience must be considered. High-quality clock sources, increased slew rate, and low-jitter buffers improve signal integrity and reduce unintentional phase variations between masked logic domains. In masked or constant-time cryptographic designs, maintaining tight synchronization between clock domains is crucial; desynchronization can cause shares to misalign in time, producing measurable correlations.

Additionally, most modern 32-bit embedded cores (such as RISC-V) are designed to operate efficiently in the 250MHz-1GHz range. Running significantly below this range can degrade performance and timing uniformity. For side-channel-resistant designs, operating near the lower end of this range (e.g., 250-400 MHz) provides a balance between performance, power integrity, and security, ensuring that masked shares switch simultaneously and minimizing leakage caused by timing skew or transient voltage dips.

3.6.9 Instructions Per Cycle

A system's Instructions Per Cycle (IPC) is a key performance metric that measures how many instructions a processor can complete per clock cycle. A higher IPC indicates that the processor is executing more work per cycle, improving overall

throughput and responsiveness. Conversely, a lower IPC often signals stalls, cache misses, or pipeline inefficiencies that slow performance. Several architectural features- such as caches, branch predictors, out-of-order execution, and speculative execution- directly influence IPC. While these mechanisms are designed to increase performance, their measurable effects on timing and behavior can inadvertently expose a system to side-channel attack (SCA) exploitation.

Although IPC itself is not inherently a "vulnerability" in side-channel attacks, variations in IPC can reveal subtle details about a processor's internal state, which attackers can analyze to infer sensitive information. Because IPC is closely tied to how efficiently instructions flow through the processor pipeline and memory hierarchy, even small timing differences can leak data-dependent behaviors. Attackers can exploit these IPC fluctuations through several key mechanisms:

3.6.9.1 Micro-architectural Contention and Timing Leakage

When multiple threads or processes share hardware resources- such as caches, memory buses, or execution units- resource contention can change the IPC of each process. For example, when a victim process accesses a shared cache line, the attacker's own process may experience reduced IPC due to delayed cache access. By observing these performance variations over time, an attacker can infer when and where shared resources are being accessed, forming the basis of cross-core timing attacks. These techniques allow adversaries to extract information about memory access patterns, control flow, or even cryptographic key usage across cores.

3.6.9.2 Speculative Execution and IPC Variability

Attacks such as Spectre and Meltdown exploit speculative execution- the processor's ability to execute instructions ahead of time before confirming whether they are needed. While this boosts IPC by maximizing instruction throughput, it can also modify the cache state in ways that encode secret data. When speculative instructions are later discarded, the architectural state is restored, but the cache remains altered. This change manifests as measurable timing differences that correlate with sensitive data. In such cases, the very optimizations that raise IPC also amplify the side channels through which attackers extract information.

3.6.9.3 Cache Misses and IPC Fluctuations

IPC can drop sharply when cache misses occur, as the CPU stalls while waiting for data to arrive from main memory. These stalls create timing discrepancies that are directly observable at the microarchitectural level. If an attacker can measure these fluctuations through execution time, performance counters, or cache probing, they can determine which memory addresses were accessed, effectively reconstructing a victim's data access patterns. Over repeated measurements, these patterns can leak information such as cryptographic operations, user input timing, or instruction traces. The tighter the correlation between IPC and cache activity, the more exploitable the side channel becomes.

3.6.9.4 Mitigating IPC-related SCA Risks

Mitigating IPC-related side-channel attacks requires both architectural and software-level defenses. At the hardware level, techniques such as cache partitioning, performance counter isolation, and deterministic execution can reduce leakage opportunities. Cache partitioning assigns private cache regions to specific processes or security domains, preventing untrusted software from influencing or monitoring shared cache behavior. Performance counter isolation restricts unprivileged access to hardware performance counters, which might otherwise expose IPC, cache misses, or branch prediction statistics.

A critical consideration is enforcing deterministic behavior- ensuring that a system's timing and IPC remain consistent regardless of input or execution path. Deterministic or constant-time execution helps prevent attackers from detecting data-dependent IPC variations. On the software side, secure coding practices, such as implementing constant-time algorithms and avoiding data-dependent branching, further limit leakage pathways.

3.6.10 Operating Temperature Range

Temperature is a critical factor in defining and validating the operating voltage range of an integrated circuit. During normal operation, the internal temperature of a chip can fluctuate widely due to switching activity, leakage currents, and environmental conditions. These thermal variations directly affect transistor performance by altering mobility and threshold voltage. As temperature rises, carrier mobility decreases, which slows transistor switching speed. At the same time, the threshold voltage tends to drop- increasing leakage and static power consumption. This complex interplay can shift timing margins and cause circuits operating near their voltage limits to fail. Conversely, at very low temperatures, reduced leakage and higher threshold voltages can lead to slower operation or even incomplete switching in marginal cells. Therefore, both high- and low-temperature corners must be considered when defining a safe voltage range.

Industry design standards often specify temperature classifications to ensure reliability across operating environments. Commercial-grade devices typically operate within a range of 0°C to 70°C, suitable for consumer electronics and general computing. Industrial-grade parts extend this range to –40°C to 85°C, accommodating outdoor and factory conditions, while automotive-grade devices may reach up to 125°C to withstand engine-bay environments. Military- and aerospace-grade components may further expand this window to –55°C to 150°C, demanding rigorous testing and validation. The chosen temperature class directly influences voltage margins. Devices expected to run across broader temperature ranges must allow more voltage range to guarantee stable operation under all conditions.

Thermal management also plays a major role in maintaining voltage stability and side-channel robustness. As temperature rises, resistance in the power delivery network increases, exacerbating IR drop and voltage noise. This can lead to transient undervoltage conditions that degrade timing or expose sensitive masked operations

to leakage variations. For this reason, voltage and temperature must be co-validated during characterization. Testing across PVT (Process, Voltage, Temperature) corners ensures that even under the hottest or coldest operating conditions, the system remains both functionally correct and secure. Additionally, temperature monitoring circuits, on-chip sensors, and dynamic thermal throttling can be implemented to keep the device within safe voltage–temperature limits during operation.

In summary, temperature effects cannot be considered in isolation when establishing the operating voltage range. They directly influence transistor performance, power integrity, and leakage behavior- factors that in turn affect both timing and security margins. Adhering to standardized temperature classes and validating performance across all thermal extremes ensures that the hardware maintains consistent, reliable behavior under any operating condition [37].

3.6.11 Cache Size

CPU cache size plays a crucial role in overall processor performance by storing frequently accessed data and instructions closer to the CPU, thereby reducing the time spent fetching information from slower main memory. This proximity allows the processor to operate more efficiently, as it can quickly retrieve the information it needs without repeatedly waiting for memory transfers. However, this performance gain is not linear. Beyond a certain point, increasing the cache size can introduce higher access latency, higher power consumption, and more complex management logic. Therefore, selecting the optimal cache size requires careful balancing- too small a cache can lead to frequent cache misses and performance bottlenecks, while an excessively large cache may yield diminishing returns due to longer access delays and increased silicon cost.

Modern CPUs often implement a multi-level cache hierarchy to address this balance between speed and capacity. Typically, caches are organized into three levels: Level 1 (L1), Level 2 (L2), and Level 3 (L3). The L1 cache is the smallest but fastest, designed to store the most frequently used data and instructions that the CPU needs almost instantaneously. The L2 cache serves as an intermediary buffer- larger and slower than L1 but still significantly faster than main memory. It helps capture data that does not fit into L1, preventing excess memory accesses. The L3 cache can be shared across multiple cores, and provides the largest storage capacity and acts as a communication bridge between cores. By maintaining a common data repository, it reduces the frequency of main memory accesses and improves coordination between cores through cache coherency protocols, which ensure all cores have consistent and up-to-date information [38].

While larger caches enhance performance by reducing memory latency and improving throughput, they also introduce potential vulnerabilities in the form of cache-based side-channel attacks (SCAs). These attacks exploit subtle variations in cache behavior- such as access times, eviction patterns, and shared resource contention- to infer sensitive information from other processes running on the same system. Attackers can measure differences in how long it takes to access specific memory addresses and use those timing discrepancies to deduce whether certain data reside in the cache.

Over time, these measurements can reveal confidential information such as encryption keys, password hashes, or user activity. The larger and more shared a cache is, the more susceptible it becomes, since a greater number of processes may compete for cache lines, and timing patterns become easier to detect.

For example, in a multi-core processor with a large shared L3 cache, an attacker running on one core might monitor changes in cache occupancy caused by a victim process on another core. This type of attack, often referred to as a Prime+Probe attack, relies on detecting which cache sets have been accessed, thus indirectly learning about the victim's data usage or control flow. Increasing cache size can exacerbate this problem, as more sets and associativity levels introduce richer timing behavior that attackers can analyze [39].

To mitigate these risks, cache design must consider both performance and security constraints. Several strategies can be applied. Hardware-level techniques include cache partitioning, which divides the cache into dedicated regions for different cores or security domains, reducing unwanted interference. Randomized cache indexing can make cache set mappings unpredictable, disrupting the patterns that side-channel attacks rely on. Software-based countermeasures, such as constant-time cryptographic implementations and cache flushing between context switches, can further obscure cache behavior. Additionally, using smaller or distributed caches at higher security levels can limit the exposure of shared resources while still maintaining reasonable performance.

Ultimately, determining the ideal cache size involves a multidimensional trade-off among speed, power efficiency, cost, and security. While larger caches generally improve performance, they may simultaneously expand the attack surface for side-channel exploits. A balanced approach- combining thoughtful cache sizing with architectural safeguards and secure software design- can significantly reduce vulnerability while preserving the performance benefits that make caching a cornerstone of modern processor architecture.

3.7 FuseSoC

3.7.1 Introduction and Motivation

FuseSoC is an open-source tool that does the jobs of a package manager and build system for HDL code (Verilog, SystemVerilog, VHDL, etc.), meant to assist in modern ASIC and FPGA development. This Python-based tool allows HDL designers and engineers to treat IP cores like software libraries with respect to how they can be discovered, declared, packaged, built, and simulated across different tools and targets. As stated in its documentation, the goal of FuseSoC is to "increase reuse of IP cores and be an aid for creating, building, and simulating SoC solutions." This means that instead of hardware projects being repetitive and reinventing things like build flows and scripts, Fuse can create an extra layer of abstraction for describing cores and their connections. This abstraction allows Fuse to automatically handle tasks like simulation, synthesis, and generating FPGA images.

As with most tools in software/hardware development, FuseSoc attempts to emphasize the reuse, portability, and abstraction of resources.

- Reuse: With how complex modern IC designs are, trying to reuse IP blocks without a structured mechanism for packaging/managing IP cores, code can get very unorganized and inefficient. This issue is addressed by Fuse's core file and metadata framework, which allows engineers to declare and version cores, as well as list and include their dependencies.
- Portability: IC design projects often need to target a variety of different vendors and boards, so a single IP core may need to be reused for different applications like Xilinx Vivado or Intel Quartus, for example. FuseSoc can abstract over multiple targets to generate flows for simulation and synthesis for different tools and devices. Without it, we developers would need to repeat the process of generating scripts and writing project files, on top of having to satisfy vendor-specific core requirements. This issue makes large-scale maintenance across builds very difficult. FuseSoC is able to fill this gap by translating a single project description into whatever toolchain is needed.
- Abstraction: While it is common practice for software development to integrate package managers and build systems into the workflow, HDL developers lacked a tool that would fulfill these roles. FuseSoC can offer a "package manager" style interface and a unified build system that creates a consistent layer of abstraction, hiding all the complexities and tool-specific details, allowing hardware developers and engineers to put more effort into actual IC designs rather than building the same designs in different environments.

3.7.2 Core Concepts and Motivation

As stated previously, a core is a reusable hardware block (IP blocks or subsystems) that can be packaged with metadata to use with a variety of different projects. To declare metadata, they are usually included in a file with the ".core" extension. These files define the fields like the core name, version, description, HDL source files, build targets, dependencies, tool flows, and parameters. This is very important as the .core files enable Fuse to build cores, understand their dependencies, and interface with designs of higher levels of abstractions.

Understandably so, dependency listing is an especially important job since, in most applications, cores depend on other cores in the system to carry out their functionality. For example, a bus master peripheral would need help from a bus interconnect core and the low-level glue logic. Relationships like this get only more important when considering the type of complex designs necessary in this day in age of IC design. To resolve these dependencies, FuseSoc resolves these dependencies by pulling in all required cores and metadata before building. The main benefit of using this method of building is the fact that it is possible to make hierarchical systems of IP modules without manual wiring of each source file or dependency chain. The metadata may also include versioning information, mappings for tools and filters, and conditions(target device or synthesis tool). This allows developers to fine-tune their

designs and makes it easier for them to have more control without spending much more effort and time.

FuseSoc also has a similar function to a traditional package manager due to its ability to discover, list, and add libraries of cores. These cores may be stored in libraries from either local(on your machine) or remote repositories like GitHub, for example. This tool includes commands like "fusesoc library add" to add a library to a workspace and "fusesoc core list" to view available cores. Once a library is added, FuseSoc can automatically look through its core files and metadata, and allows the developer to select them once it is time to build the design. This method of library abstraction allows teams to share groups of IP cores in a structured way instead of loosely copying portions of code. Furthermore, FuseSoC is known as a non-intrusive application, which means that designs do not need to undergo any changes to be compatible, further emphasizing its modular nature and the ability to integrate into projects with ease.

3.7.3 Workflow and Usage

FuseSoc can be installed by using the Python package manager with the 'pip install fusesoc' command and works on most prominent operating systems (Windows, Linux, macOS). After installation, users may configure their remote or local core libraries using the fusesoc command mentioned earlier. Configuration details are stored in a YAML configuration file, allowing persistent setup of multiple libraries. This modular approach allows users to smoothly mix official, organizational, and personal IP repositories.

To create a new core, a developer must first define its metadata in a .core file using YAML syntax. Important fields to include in the file are CAPI (Core API version), name, description, filesets (listing HDL files), parameters, dependencies, and targets. For example, a simulation target would specify source files along with the simulation tool, while a synthesis target would specify the constraints and top module for FPGA implementation. When declaring dependencies, they are referenced as core names and automatically include their respective lower-level modules. Versioning fields such as `core_version` allow structured tracking, evolution, and controlled distribution of hardware cores. This structure makes it possible to reproduce builds and maintain traceability.

Once the core and libraries are set up, builds are initiated by using the "fusesoc run" command. After this initial statement, it is meant to be followed up by "–target=" to specify the target type and its name. The type could be synth(synthesis), sim(simulation), or formal(formal verification). FuseSoC supports specifying things like parameters, build directories, and backend overrides from the command line. Each run is also self-contained, meaning that they are repeatable and create isolated builds.

FuseSoc supports both open source(Verilator, Icarus Verilog,etc.) and proprietary(Vivado, Cadence Xcelium, etc.) EDA tools, and abstracts their usage through a common interface. Because of this, users can switch tools without modifying scripts or HDL sources.

3.8 LiteX

LiteX is a Python-based framework for the rapid assembling of FPGA SoC fabrics [40]. Rather than acting primarily as a package manager, like FuseSoC [41], LiteX works as an SoC generator. You are able to describe a system in Python, target specific boards/other targets, and LiteX will use whatever existing components of it's library in order to stitch together a CPU. This stitching also includes interconnects, memory structures, and peripherals defined within the system description. From here, a configurable toolchain flow is executed in order to generate and load a bitstream.

Common subsystems are provided as reusable "LiteX cores", such as LiteDRAM (memory controller), LiteEth (ethernet), LitePCIe, LiteSDCard, and so on. Systems typically connect over a wishbone or AXI-Lite interface, with an exposed CSR register map for platform bring-up.

These features allow for LiteX to emphasize fast FPGA prototyping. Significant board support is provided in their repository/documentation, allowing for users to generate and load a working SoC on a wide variety of dev boards, with tutorials/blogs/discsusions regarding how to extend a "BaseSoC".

We opted to not use LiteX for this project as our deliverables target an ASIC-first design flow, with a strong emphasis on IP packaging, dependency mangaement, and tool neutrality across multiple EDA back-ends. LiteX's strengs and documentation skew toward FPGA board targets and bitstream generation. On the other hand, FuseSoC aligns with our need to treat IP like reusable packages, driving varied toolchains through a neutral levl of abstraction. Also vital for our design choice, the lowRISC Ibex documentation explicitly uses and recommends the usage of FuseSoC, showcasing commands and core generation for a "simply system" bring-up and co-simulation workflows. Therefore, adopting FuseSoC allowed us to keep aligned with the core's documented flow and ready-made configurations.

LiteX would make sense for us to use if we re to pivot and target rapid FPGA bring-up, board demos, or a SoC oriented around a soft RISC-V core with significant peripheral attachment. LiteX's relatively swift concept-to-firmware path makes this all a great strength. With that all said, our ASIC-centered workflow allowed for us to see FuseSoC as the better fit for our needs.

3.9 Bender (ETH Zürich)

3.10 Chipyard

Table 3.7: *Build/SoC Framework Comparison*

Build/SoC Framework Selection Comparison					
Tool	**Role / Category**	**Implement In**	**Backends / Outputs**	**Strengths**	**Drawbacks**
FuseSoC	IP package manager + unified build / orchestration for HDL projects	Python (with `.core` YAML metadata)	Wraps simulators and synth/P&R via Edalize (e.g., Verilator, Icarus, Questa/Xcelium; Vivado, Quartus, etc.); generates project files, runs flows	Non-intrusive; versioned IP cores; dependency resolution; tool-neutral abstraction; repeatable builds; aligns with Ibex docs/flows	Not an SoC generator; learning curve for `.core` metadata; relies on backend tool availability/config; less opinionated—needs project structure
LiteX	Python SoC generator (stitches CPU, interconnects, DRAM / PCIe / Eth, peripherals; fast FPGA bring-up)	Python (Migen / Amaranth ecosystem)	Generates gateware and board projects for many FPGAs; integrates LiteDRAM, LitePCIe, LiteEth, etc.; produces bitstreams via vendor/open flows	Rapid FPGA prototyping; rich reusable "LiteX cores"; large board coverage; cohesive SoC composition	Skews FPGA / bitstream-centric; less tool-neutral for ASIC; opinionated stack; less suited to IP packaging as a first-class artifact
Bender					
Chipyard					

Chapter 4 - Standards and Design Constraints

4.1 Industry Standards

4.2 Design and Architectural Standards

4.3 Practical and Security Constraints

4.3.1 Design Constraints

4.3.2 Practical Constraints

4.3.3 Security Specific Constraints

4.4 RISC-V Architecture

Due to its modularity and adaptability, the RISC-V ISA is widely known as one of the most popular and impactful instruction set architectures of our generation. A major cause of its success is its openness in nature. Due to ISAs like x86 and ARM being proprietary in nature, using their designs can lead to a lack of transparency and flexibility for engineers and developers. This is where RISC-V steps in, as it is open source, encouraging a collaborative global community dedicated to improving, modifying, and teaching the architecture. In addition, organizations such as RISC-V International have accelerated the adoption of the standard in new markets.

RISC-V was first introduced in May 2010 at UC Berkeley with funding from DARPA. As the name suggests, it is the fifth version of the Reduced Instruction Set Computing (RISC) standard, which emphasizes the use of simpler and shorter instructions. These operations allow processors to reach higher speeds and power efficiency compared to earlier designs.

The RISC philosophy greatly contrasts with its predecessor, CISC. Complex Instruction Set Computing(CISC) emphasized a large number of complex instructions that could perform multiple low-level operations in a single command. Although this structure enabled simplified software to run on the cores, it placed more responsibility on the hardware, as it had to support many more complex instructions and types of

memory accesses. This aspect of CISC came with some costly consequences: slower execution, reduced energy efficiency, and more complicated hardware design.

Although proprietary architectures like ARM and x86 are currently dominating fields like mobile computing, Internet of Things(IoT), and high-performance computing, RISC-V is quickly catching up with its growing dedicated community and abundance of resources. The following sections discuss the architecture of RISC-V and its extensions, its design options, and real-world applications.

4.4.1 Basic Architecture

The RISC-V base instruction set architecture (ISA) usually takes the forms of RV32I, RV64I, or RV128I, which represent 32-bit, 64-bit, and 128-bit variants, respectively. The "I" at the end of these names means that a core comes with the base integer instructions. This is necessary because it includes the arithmetic, logic, immediate, load/store, and control-flow instructions that every RISC-V core needs to function. A notable feature of RISC-V is its ability to include the RVC, or compressed extension, which allows instructions to be represented in just 16 bits, rather than the full 32 bits. This format allows a processor to use up less memory, which has proven to be incredibly beneficial in embedded applications where it is important to minimize the amount of space and resources necessary for tasks. For the rest of this discussion, we will be referring to the RV32(32-bit) base ISA format, which has qualities and characteristics that should carry over to most other formats.

Without any extensions, the base RISC-V ISA contains only a few dozen instructions. These instructions cover the most essential categories like arithmetic/logical, immediates, load/store, control flow, and system operations. Unlike the more complex ISAs, this minimal base design offers a simple and efficient set of operations that can be expanded upon modularly as needed through the instruction set's extensions. The heart of the ISA is its register file: RISC-V provides 31 general-purpose registers in addition to a special hardwired register, x0, which always holds the value zero. This constant register is very useful for simplifying operations and eliminating a few unnecessary instructions, making the job of developers easier. Among the 31 available registers, some are assigned special tasks such as maintaining the stack, global, and thread pointers, while others are divided into temporary and saved registers for use during function calls. This organization allows for an environment where registers can be used efficiently and minimizes any confusion on the side of engineers and developers. Each instruction in the base RISC-V ISA adheres to a fixed 32-bit format divided into the following fields: opcode, destination register(rd), source registers(rs1 and rs2), and function codes. Once again, this uniformity greatly simplifies instruction decoding hardware and control complexity, especially when compared to its CISC counterparts.

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

Figure 4.1: *The RISC-V Integer Register Set (x0–x31)*

One of the defining characteristics of the base RISC-V design is that it is a load-store architecture. In this design, only the load and store instructions are allowed to access memory outside the CPU's registers. All other operations occur strictly within the registers themselves. This decision greatly simplifies hardware implementation, since the processor only needs to support two types of memory interactions rather than a wide variety of instructions that each directly access memory. From an architectural standpoint, this uniformity also improves pipeline design, allowing the instruction pipeline to be more predictable, consistent, and efficient. By enforcing a clean boundary between register operations and memory access, RISC-V reduces complexity and ensures greater scalability in more advanced processor designs.

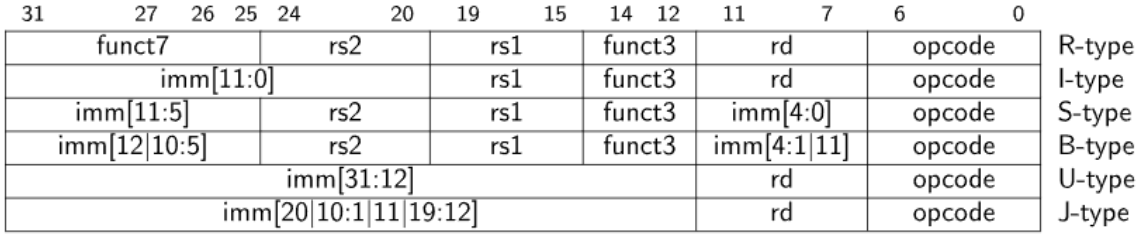


Figure 4.2: *Instruction Types for RV32I ISA*

4.4.2 ISA Extensions

Contrary to how it sounds, the simplicity and rather bare-bones nature of the base 32-bit ISA is one of RISC-V's greatest strengths. This is because there are many

approved extensions to give extra functionality to the base ISA mentioned above. This modular approach empowers developers to pick and choose what functionalities they want to be a part of their designs with no wasted resources. Implementing these extensions is straightforward, as each new instruction has its own dedicated encoding within the opcode field. When combining extensions, developers refer to their design as a ISA string showing the supported instruction sets. For example, a core labeled as RV32IMC would be a 32-bit design with the integer (I), multiplication/division(M), and compressed(C) extensions. Correctly implementing these tools allows RISC-V to be incredibly efficient and versatile.

4.4.2.1 M-type

A key example is the M extension, which enables a core to process multiplication and division instructions. As per the rules of multiplication, when two numbers of equal length are multiplied, a product of twice the original bit length is returned. This means that when we are working with 2 32-bit numbers, we need 2 different registers to hold the entire 64-bit value. To hold true to the RISC straightforward philosophy, there are two instructions: MUL and MULH. These instructions calculate the lower and upper bits, respectively. Similarly, division needs separate registers to hold the quotient and remainder values returned from the DIVW and REMW instructions. These instructions are critical for fields like digital signal processing, graphics rendering, and machine learning, where multiplication and division at high speeds are necessary.

4.4.2.2 A-type

Another critical extension is the A extension, which enables atomic instructions that are essential for building reliable multithreading systems. The name "atomic" refers to their indivisible nature, meaning that once these instructions begin, they either complete completely or not at all. This characteristic guarantees that certain memory operations cannot be interrupted or corrupted by other threads on the same system. Atomic instructions can come in two forms: load-reserved/store-conditional (LR/SC) pairs and atomic memory operations (AMOs).

The LR instruction is essentially the same as the base lw(load word) operation, but it uses an extra register to create a reservation in the core's memory. This reservation indicates that the memory loaded from this address is valid and has not been modified by other threads. Once the core is ready to store the data in the same address, it will use the atomic SC instruction to invalidate other cores' reservations on the memory, then write back to it. If a core attempts to write to an address while its reservation is invalid, the SC instruction will fail, and the processor retries the operation starting from its corresponding LR. Although this process could cause multiple retries for memory addresses with high traffic, it ensures that every thread is working on valid data.

On the other hand, atomic operations(AMO) can offer a simpler approach to memory operations in multithreaded environments. For example, an atomic add

instruction can increment a shared counter in memory without the risk of another thread reading or modifying it in the middle of the update. These instructions are special because they are able to read, modify, and write in a single atomic step. They include the functionality of basic instructions like SWAP, AND, OR, and XOR. The use of AMOs prevents race conditions and enables efficient synchronization between multiple cores.

Without atomic instructions, writing correct and reliable multithreaded programs would be significantly more difficult. Developers need to rely on cumbersome software mechanisms to coordinate access to shared memory, leading to inefficiency and potential errors. By including hardware support for atomic operations, RISC-V not only simplifies programming but also makes it possible to build robust, high-performance systems with multiple cores.

4.4.2.3 F/D-type

The F and D extensions add support for floating-point arithmetic in single precision (32-bit, F) and double precision (64-bit, D). For the sake of ensuring compatibility with software, each of the instructions in these extensions follows the IEEE-754 floating point standard(1 sign bit, 8 exponent bits, and 23 mantissa(fractional) bits). They each contain instructions for tasks like addition, subtraction, multiplication, division, and square root. A unique aspect of these extensions is that they add 32 different registers (f0-f31) reserved for floating point values.

Clearly, floating-point support is indispensable in applications such as scientific simulations, graphics rendering, and machine learning, where fractional values are necessary. Without these extensions, programs would need to emulate floating-point arithmetic using integer instructions, leading to slower performance. By providing native hardware support for floating-point arithmetic, RISC-V can efficiently handle workloads that demand high numerical accuracy.

f0–7	ft0–7	FP temporaries
f8–9	fs0–1	FP saved registers
f10–11	fa0–1	FP arguments/return values
f12–17	fa2–7	FP arguments
f18–27	fs2–11	FP saved registers
f28–31	ft8–11	FP temporaries

Figure 4.3: *The RISC-V Floating-Point Register Set (f0–f31)*

4.4.2.4 C-Type

By using the RISC-V compressed extension (C-type), developers can reduce code size by implementing 16-bit encodings for commonly used instructions. Instead of introducing new functionality to the RISC-V core, the C-type extension focuses on improving the efficiency of instructions like loads, stores, and arithmetic. This

improvement effectively shrinks the program's memory footprint and improves cache efficiency and memory usage. Upgrades like these are perfect for embedded systems and microcontrollers, where resources are often limited. With both the C-type extension and base 32-bit ISA, most RISC-V cores can work with both bit lengths simultaneously since the compressed instructions can be placed on any 16-bit boundary. This gives engineers the flexibility to save small amounts of space throughout their designs without extra effort. In the case of the C-type extension, only registers x8 through x15 are used in many of the compressed instructions, simplifying decoding while maintaining compatibility with the full register set.

For example, the compiler can switch to 16-bit values when the entire 32 bits aren't necessary, like when using the hardcoded zero register(x0) or when including a small immediate in a calculation. This, of course, comes with the downside of needing additional alignment logic and decode paths to expand the compressed instructions before execution. Although implementing these compressed instructions does add slight complexities to the hardware design, the positives usually outweigh the negatives, as in most cases, more than half of a program's instructions can be replaced by their 16-bit counterpart.

To integrate the C-type extension into a core, it needs to support a dual instruction encoding scheme. During the decoding stage in the pipeline, the instruction can be either left alone or extended, which is signified by the last 2 bits in the opcode. Doing the extension this way minimizes the additions necessary and keeps the rest of the pipeline relatively unchanged.

4.4.2.5 B-Type

The B-type extension adds a set of bit manipulation instructions that includes shifts, rotates, bitfield extract and insert, population count, and leading/trailing zero counts. At first glance, these additions may seem trivial, but they are critical for performance in applications like cryptography and compression. Although the functionalities introduced in the extension are possible in the base ISA, they would require multiple instructions to emulate them. This, in turn, would increase the instruction count, thereby increasing execution time. Because the B extension provides direct hardware support for these operations, compilers can generate more efficient code for bit-level algorithms. Since some instructions included in the B-type extension are more specialized than others, the extension itself is split into different submodules to avoid complicating the ISA and adding bloat to the design.

For instance, to compute a population count without the extension would normally need loops of shifts and masks, taking up a lot of time. However, with the cpop B-type instruction, the processor can go through the whole operation in just one cycle and free up the pipeline. Luckily, compilers also are able to automatically detect bit manipulation patterns and replace them, simplifying software integration.

4.4.3 Memory

The RISC-V ISA uses a memory consistency model referred to as RVWMO (RISC-V Weak Memory Ordering). In the RVWMO model, memory load and stores are only controlled by a "global memory order". The global memory order is an absolute ordering of memory accesses for all threads in a system. Other than maintaining the integrity of the global order, the instructions are free to be moved around in any sequence. This gives developers a lot of flexibility to reorder memory operations internally to improve performance, which further emphasizes the principles that RISC-V was built on. Unlike other more strict models like Total Store Order(TSO), RVWMO minimizes excessive restrictions, allowing the hardware to optimize memory access latency and instruction-level parallelism. Furthermore, this model is designed to be compatible across both single-processor and multiprocessor systems, adding to its versatility.

When it comes to the virtual and physical memory boundary, RISC-V uses multi-level page tables managed by an MMU, controlling the virtual address used in software applications to the physical ones used by hardware. Although the architecture usually uses a flat, linear virtual address, the implementations need to enforce isolation, access permissions, and translation correctness.

On the security side, the RISC-V privileged architecture defines a physical memory protection (PMP) mechanism, which developers have the option to implement within their design. PMP essentially allows programmers to define regions with configurable read, write, and execute permissions, as well as locking behavior.

4.4.4 Pipelining

The standard RISC pipeline that is also used in RISC-V implementations is made up of the usual stages: fetch, decode, execute, memory accesses, and writeback. As stated multiple times, the RISC-V instruction set is made up of many simple instructions, allowing them to be mapped onto this structure very cleanly. However, when multiple instructions have to share resources and dependencies, the cores must still manage hazards correctly and efficiently.

Data hazards occur when a register that hasn't finished being computed is needed for another instruction. If this issue is not addressed in a design, it can result in instructions operating on incorrect data, leading to invalid outputs. Luckily, the RISC-V standard supports solutions like forwarding and precise exceptions. By implementing the forwarding method, instructions can send data directly to another before getting to its writeback stage. This allows implementers to avoid stalling the pipeline for multiple cycles and maximizes uptime. With precise exceptions, a processor can continue with instructions that rely on data being operated on. In the situation where the result of the prior instruction differs from the data that was operated on, the will go back to the last known valid state, not committing any of the later instructions. These two methods of minimizing the effects of data hazards can also be used together.

Control hazards occur during the cycles of branches and jumps, where the next

instructions to be executed are not known. Handling these situations incorrectly can cause pipeline flushes, which greatly degrade the performance of the system. RISC-V does not specify any branch prediction mechanisms and gives the hardware designers complete freedom to implement their own mechanisms. One of these mechanisms is the early branch resolution, which simply moves the branch comparison to earlier in the pipeline (from EX to ID, for example) to greatly reduce the flush penalty. This is a very common method in simpler cores like Ibex.

Lastly, structural hazards occur when multiple instructions compete for the same hardware resources. These hazards can be easily avoided due to RISC-V's simple design that ensures consistent and predictable timing behavior.

In classic RISC-V fashion, the standards also allow processors to be designed with different pipeline depths depending on the developers' goals. First is the most commonly used, 5-stage pipeline with distinct IF, ID, EX, MEM, and WB stages used in industry and universities. This configuration works great when trying to maximize performance and the complexity of designs, since splitting the stages gives developers more options for optimizations like choosing different forwarding paths and having multiple ways to configure interlocking. Furthermore, with the stages split up like this, it shortens the critical path of the processor, allowing for a higher clock frequency.

Another common pipelining configuration is the 3-stage pipeline, where the 5 original stages are grouped into fetch, decode/execute, and memory/writeback. Grouping stages like this allows for limited forwarding paths and makes avoiding hazards much easier. Although the 3-stage makes a much simpler core, there are fewer options for optimizations, and the critical path is slightly longer than the aforementioned 5-stage pipeline. Lastly, the 2-stage pipeline splits fetch from the rest and groups the other 4 together. The vastly reduced complexity of this design makes it great for ultra-low-power designs and embedded cores due to the fewer control paths and pipeline registers needed. Although this is great for minimizing resources the necessary, the overall performance of the core takes a hit, as the critical path is much longer than the 3 and 5-stage configurations.

4.4.5 Privileges

Security and privilege management are also the principles that the architecture of RISC-V looks to uphold, ensuring that software with varying levels of trust can co-exist. Through its privileged architecture, it extends the base ISA with mechanisms that manage access to system resources, enforce isolation, and protect firmware from malicious entities and glitches.

RISC-V utilizes a privileged architecture that incorporates multiple privilege levels to ensure security and stability. Different configurations may include 3 main modes: Machine, Supervisor, and User. Machine mode (M-mode) is the only mandatory privilege level and is the highest level. This is due to the fact that it is responsible for firmware and device control, and to boot up the processor, it needs full access to the hardware. Supervisor mode (S-mode) is optional and supports operating systems by managing virtual memory and system calls. This mode is useful for processors that incorporate Unix-like operating systems since it helps manage virtual memory. The

least privileged mode is the User mode(U-mode), which is preserved for application code. These layers, together, ensure that applications cannot access higher-privileged CSRs and memory regions, thereby maintaining a strong separation between software levels.

4.5 Universal Verification Methodology (UVM)

Universal Verification Methodology, often referred to as UVM, is an IEEE standard that defines a SystemVerilog class library and APIs for building reusable, coverage-driven, constrained-random verification environments for digital designs. The exact specification for this is IEEE 1800.2. It standardizes the base classes, phasing, configuration, reporting, interfaces, sequences, and various mechanisms that make up UVM. Alongside IEEE, Accellera maintains a reference implementation of UVM that is aligned to IEEE specifications [42].

The IEEE Standard's stated goals are to improve interoperability between EDA tools, reduce the cost of using intellectual property for new projects, and lower verification costs through a common methodology. Practically, UVM provides a team with a consistent way to derive drivers, monitors, scoreboards, and sequences, as well as ways for them to be shared across projects and vendors.

More specifically, the components that make up a UVM testbench (driver, sequencer, monitor, agent, environment, scoreboard, subscriber, virtual sequencer/sequence, test, and Register Abstraction Layer [43]) are all outlined in the sections below:

4.5.0.1 uvm_driver

Converts high-level sequence items into pin-/protocol-level activity on a virtual interface. The driver pulls items from a sequencer via seq_item_port, drives the DUT per protocol timing/handshakes, and returns responses,if applicable.

4.5.0.2 uvm_sequencer

Arbitrates and dispatches sequence items from one or more sequences to a driver. It connects to the driver through seq_item_export and seq_item_port, handling item requests and grants.

4.5.0.3 uvm_monitor

The UVM monitor is a form of passive sampler that observes an interface (from the driver) and converts any activity into transaction objects. Decoded transactions are pushed to scoreboards, subscribers, or predictors, through a connection known as uvm_analysis_port.

4.5.0.4 uvm_agent

Bundles the pieces for one protocol/interface, typically a sequencer, driver, and monito, behind a single component. Agents can be active (drive + monitor) or passive (monitor-only). Monitor analysis ports usually fan out to subscribers and scoreboards.

4.5.0.5 uvm_env

A top-level "container" for one or more agents plus scoreboards, register models, coverage collectors, and the cross-component connections. Good practice is to keep environments as DUT-agnostic as possible and push per-test tweaks into config/factory overrides.

4.5.0.6 uvm_scoreboard

Checks correctness by comparing observed transactions against a reference/prediction model. Keep prediction (what should happen) separate from comparison (what did happen) so you can swap predictors or change check strategies without touching the DUT plumbing.

4.5.0.7 uvm_subscriber

A generic analysis consumer that implements a write method. Commonly used for functional coverage or lightweight checkers that need a clean, single-purpose tap on the data stream.

4.5.0.8 virtual_sequencer & virtual_sequence

A virtual sequencer holds handles to multiple interface-level sequencers. A virtual sequence runs on that virtual sequencer and orchestrates coordinated stimulus across agents; useful for SoC-level scenarios that span buses and subsystems.

4.5.0.9 uvm_test

The top-level component that builds the environment, applies configuration (e.g., via uvm_config_db), sets up factory overrides, and kicks off top sequences. Keep tests thin and robust: tailor behavior via config/overrides rather than custom wiring.

4.5.0.10 uvm_reg

The RAL provides abstract register/memory models (uvm_reg, uvm_mem, uvm_reg_block) plus plumbing for frontdoor (via the bus) and backdoor (via HDL paths) access. Two key helpers are the uvm_reg_adapter, which translates abstract register ops to bus sequence items (and back), as well as uvm_reg_predictor, which updates the mirrored register model from observed bus traffic, keeping the predictive model consistent with the DUT

4.5.0.11 UVM Wrap-up

It is worth noting that this list is non-exhaustive, though it is fairly comprehensive of the most critical components. Specialized testbenches may introduce things such as TLM analysis FIFOs, protocol checkers/assertions, coverage-only subscribers, and utilities like callbacks or report catchers [42].

4.6 OpenTitan

Briefly mentioned within the introduction of OpenTitan in the research chapter, OpenTitan also functions as a standards-shaping effort.[44] The Comportable IP (CIP) specification defines common interfaces and requirements for IP integration, provides a CIP UVM library to standardize IP-level testbench architecture, clarifies coding styles within the definition of the SystemVerilog style, and documents DV methodology/hardware development stages that can be adopted. There are also requirements regarding licensing and copyright to ensure fair and valid usage of the OpenTitan name.

4.6.1 Comportable IP (CIP) Architecture

4.6.2 CIP Library Classes

4.6.3 Security Verification in CIP Library

4.7 Engineering Standards Selection

4.7.1 Instruction Set

The RV32I instruction set was selected because it represents the 32-bit base integer subset of the RISC-V ISA, offering deterministic behavior and low implementation complexity. Its simplicity reduces transistor count and power leakage, both of which directly minimize potential side-channel leakage vectors. By excluding floating-point or vector extensions, RV32I provides a compact and predictable execution profile that facilitates secure masking and formal verification.

4.7.2 Operating Temperature Range

The 0 °C to 120 °C temperature range ensures the processor operates reliably under industrial and defense-grade environments while preserving consistent electrical characteristics for side-channel mitigation. Temperature fluctuations influence leakage current, transistor threshold voltage, and propagation delay- all of which can distort the uniformity of power and EM signatures. By bounding the thermal envelope to this range, the design minimizes variation in dynamic power behavior and ensures that masking correlations and noise amplitude remain stable during operation. The upper limit of 120 °C provides a safety margin for stress testing and reliability qualification

without introducing excessive thermal noise or timing drift that could compromise secure execution.

4.7.3 Operating Voltage Range

The selected voltage window of 0.95 V to 1.25 V provides a stable operational range that balances energy efficiency, timing reliability, and side-channel consistency. Voltages below 0.95 V increase the risk of timing violations and signal degradation, while those above 1.25 V can amplify dynamic power consumption and electromagnetic emissions- both of which enhance leakage observability. Constraining voltage within this moderate band ensures deterministic switching behavior across all process corners, reducing power variance and signal amplitude deviations. This tight regulation also supports reliable operation of masking and noise injection circuits, which depend on steady supply conditions to maintain synchronization and statistical independence.

4.7.4 Process Node

A mid-range process node provides an effective compromise between security and performance. At this scale, designers can achieve clock frequencies in the range of 500 MHz to 1 GHz, which is sufficient for RISC-V-based workloads while maintaining manageable leakage behavior and realistic side-channel evaluation conditions. This balance makes the 45 nm node an attractive choice for implementing secure, masked, or instruction-set-extended processor cores.

4.7.5 IR Drop

The maximum allowable IR drop of 5% was chosen to maintain voltage uniformity across the chip's power distribution network, preserving both timing integrity and side-channel resistance. Excessive voltage drop introduces local delay variations that can desynchronize masked shares or distort the spectral balance of injected noise, creating exploitable leakage paths. A 5% threshold ensures that voltage remains sufficiently stable under peak dynamic conditions, preventing data-dependent fluctuations in current draw. This constraint directly supports the core's goal of power predictability, ensuring that the observable power signature remains as uncorrelated to secret data as possible while maintaining robust performance and timing closure.

4.7.6 Average Frequency

The Ibex security-enhanced core targets an average operating frequency of at least 250 MHz to achieve sufficient throughput for embedded cryptographic workloads while maintaining controlled power behavior. Operating at or above this threshold ensures that masking and noise-injection logic meet real-time performance requirements without forcing excessively high dynamic switching, which could increase EM emissions or thermal leakage. The 250 MHz target was selected as the optimal point between energy efficiency and predictable timing margins- fast enough for secure computation,

but conservative enough to retain stable operation across process corners. Maintaining this frequency floor also simplifies clock-domain synchronization, ensuring that noise and masking operations remain statistically balanced within each clock cycle.

4.7.7 Masking Order

A first-order masking scheme was chosen as the baseline level of protection for the Ibex security extension, as it provides the most substantial reduction in side-channel leakage per unit of hardware cost and design complexity. In first-order masking, each sensitive variable is split into two statistically independent shares, ensuring that any single power or EM trace reveals no direct information about the true data. This approach effectively protects against simple and first-order differential power analysis (SPA/DPA), which are the most common and practically exploitable forms of side-channel attacks in embedded processors.

From a hardware design perspective, first-order masking strikes the optimal balance between security, performance, and verifiability. Implementing higher-order masking (second- or third-order) significantly increases circuit area and routing overhead due to the exponential growth in required shares, random bits, and register duplication.

Furthermore, a first-order design serves as a verifiable security baseline within the chosen 45 nm process node. This level of assurance allows subsequent design iterations to scale security upward (e.g., higher masking orders or hybrid schemes) while maintaining proven leakage resistance at the fundamental level. Thus, selecting first-order masking ensures that the SCALER core achieves strong practical resistance to side-channel attacks while preserving timing determinism and engineering feasibility.

4.7.8 Instructions Per Cycle

Limiting the core's IPC to a maximum of 1.2 preserves a balance between throughput and security. High IPC values often require aggressive out-of-order execution, deep speculation, and complex instruction scheduling- all of which introduce microarchitectural timing variations that can be exploited through side-channel attacks. By capping IPC near a single-issue threshold, the Ibex core maintains predictable and deterministic timing behavior, making it easier to validate constant-time execution. The slight allowance above 1.0 provides flexibility for benign dual-issue scenarios that improve performance without adding speculative complexity. This design decision ensures stable timing across instruction types while minimizing data-dependent performance fluctuations that could correlate with secret information.

4.7.9 Cache Size

A cache capacity of at least 16 MB was selected to provide sufficient temporal and spatial locality for the Ibex core under realistic workloads, while supporting isolation and partitioning strategies essential for side-channel attack (SCA) resistance. From a performance standpoint, internal simulations demonstrated that hit-rate

improvements begin to plateau around 16 MB, indicating that this capacity represents an efficient point of diminishing returns for most embedded and security-oriented workloads.

From a security standpoint, a larger shared cache also enables effective cache partitioning and randomization, both of which are critical in mitigating cache-based SCAs such as Prime+Probe and Flush+Reload. With 16 MB of total capacity, the cache can be divided into independently managed ways or regions, allowing secure processes to operate within dedicated partitions isolated from untrusted software. Additionally, this size supports hardware-based random indexing or set-mapping functions, which make cache state changes less predictable to an attacker. These techniques are only feasible when the cache provides adequate space for both performance-critical and security-critical operations to coexist without thrashing.

4.7.10 Timing Failure Count

A zero-fault tolerance requirement ensures fully deterministic timing across all process, voltage, and temperature (PVT) conditions. Even a single setup or hold-time violation can produce non-uniform delays that reveal secret-dependent data flow or cause masked shares to lose alignment, undermining the integrity of Boolean masking and noise injection schemes. By enforcing strict zero-fault operation, the design guarantees consistent signal propagation and cycle-accurate behavior, essential for constant-time execution. This standard also improves long-term reliability and verification confidence, as it eliminates timing ambiguity that could otherwise translate into measurable side-channel leakage or functional instability under environmental stress.

Chapter 5 - Application of ChatGPT and Similar Platforms

Large Language Models (LLMs) like ChatGPT have quickly become everyday tools for writing, developing, and searching. Their value lies in domains where common truths are textual, loosely structured, and tolerant of stylistic variance, such as email drafts, simple programs/scripts, or API boilerplate. With this in mind, Electronic Design Automation (EDA) differs significantly. Hardware design tends to exist under unforgiving constraints: correctness over all input traces, synthesizability of RTL code, verification coverage, timing closure, and specific toolchain version quirks are only to name a few. Generative AI in this space seems promising as an extremely open area of research, but immature for many practical applications.

For our Senior Design Project, ChatGPT can be seen as a useful companion for navigating documentation in open-source software(s) that we leverage, checking for syntax validity in various design implementations, or even providing alternative conceptual approaches that we may not have initially considered. When used interactively, it can be a powerful catalyst to propel our efforts into a functional baseline.

However, we must recognize the limits. LLMs can confidently produce HDL that is syntactically valid but functionally incorrect (e.g., introducing clock-domain crossing issues, mishandling reset behavior, or inferring unintended latches), propose verification scenarios that miss corner cases, or suggest flows that don't match real tool behavior. Critically, LLMs have no direct notion of PPA or timing; they cannot see post-synthesis or post-route consequences without human-driven experiments.

Confidentiality is another concern. While our RTL builds upon open-source designs, many resources we use to learn the flow are licensed training materials from Cadence Design Systems. If we encounter an issue that resembles content in those materials, it may be tempting to paste excerpts into a prompt, risking disclosure of confidential information and potential license violations. The following subsections provide various examples of how generative AI models may benefit or harm our learning experience throughout this project.

5.1 Example 1 - Sourcing RTL files for the small configuration of the Ibex Core

The Ibex Core by LowRISC provides multiple configurations within its repository, which are managed through FuseSoC. FuseSoC is a build and package management tool designed to simplify the creation, configuration, and integration of HDL projects. It enables users to generate project files, manage file hierarchies, and configure design parameters for different EDA tools, streamlining simulation and synthesis workflows.

For this project, the small configuration of the Ibex core was selected. One of the initial objectives was to use FuseSoC to generate the appropriate RTL for this configuration. However, the official Ibex documentation offered limited guidance on preparing these files with FuseSoC. The available examples only demonstrated how to instantiate the default configuration and an integrated system called simple_system that included an Ibex core.

After multiple unsuccessful attempts and iterations, ChatGPT was prompted to identify the correct command needed to generate the RTL using FuseSoC. The suggested command combined syntax elements from both documented examples but failed when executed. To troubleshoot further, GitHub Copilot and GPT-5's agentic reasoning model were utilized in an attempt to debug the issue. At one point, the model attempted to manually reconstruct the SystemVerilog files for the small configuration without using FuseSoC, based on repository information. However, since any AI-generated modifications to SystemVerilog could compromise design integrity and debugging accuracy, this approach was discontinued.

Upon closer inspection of the provided example commands, a new hypothesis was tested. ChatGPT was asked whether the default configuration might correspond to the small configuration. Initially, it incorrectly asserted that they were distinct. Further investigation with Agentic GPT-5 revealed that the "default" configuration was not explicitly defined in the repository's README.md. The model had incorrectly inferred it as a separate configuration. Once this discrepancy was identified, a comparison of the parameter sets confirmed that the default and small configurations were indeed identical a finding verified both by the model and through manual inspection.

This experience highlights the importance of critical evaluation when collaborating with large language models (LLMs). Blindly trusting generated outputs can lead to significant delays; in this case, the debugging process extended over a week. A strong understanding of project context and documentation proved essential in resolving the issue. This situation underscores the limitations of LLMs when working with incomplete or ambiguous technical documentation and emphasizes the need for human oversight in AI-assisted engineering workflows.

5.2 Example 2 - Setting up LowRisc Toolchain

Alex had some interactions with GPT regarding RV32/RV64 Compiler

5.3 Example 3 - ...

placeholder, up for grabs

5.4 Example 4 - ...

placeholder, up for grabs

Chapter 6 - Hardware & Software Design

This chapter documents the hardware and software design of our RISC-V subsystem built around the Ibex core. The Register-Transfer Level (RTL) sources were obtained and instantiated by FuseSoC, which produced a self-contained set of Verilog/SystemVerilog files, supporting IP, and simulation collateral for our chosen target, the small configuration. Seeing as FuseSoC resolves dependencies and manages reusable "cores", the export includes both directly instantiated modules and more utility-focused/supportive library files. In this chapter we focus primarily on the modules that are actually instantiated within our build of the small Ibex system and are architecturally relevant to our design.

It is worth recognizing that FuseSoC does not trim these files in a way that makes maneuverability simple, so some module descriptions may be more applicable to core instantiations other than the small design.

Our goals here are threefold: (1) to explain the function and interfacing of each module, such that a reader can understand the data and control flows without having to parse the RTL code themselves. (2) To show how these modules compose into the top-level system architecture, enabling our end-goal functionality. (3) Connecting each module's responsibilities to the verification and firmware implications that will be addressed later. Most modules can be referred back to a single overall block diagram within Chapter 2.6 (Hardware & Software Block Diagrams).

Each following subsection will cover one module. For each module, we aim to describe the purpose, parameters, interfaces (signals and protocols), known dependencies, configurability, and any design or verification notes specific to our project design. Anywhere a module might be seen as a wrapper, we aim to point out what logic is pass-through (buses, validating logic) versus transformative behavior (combinational and sequential logic on data). We may note timing-critical paths or resource trade-offs that influence any design choices on modules that are foreign to the original Ibex RTL.

6.1 Core Architecture and Top-Level

These top-level modules define and outline the overall structural organization of the Ibex RISC-V core and act as the foundation for how the various subsystems interact. In this level, the processor's pipeline, control logic, memory interfaces,

and debug features are all combined into a cohesive design. As with most other hardware designs, the top-level module acts as the entry and exit point for all external communication, as it exposes interfaces for things like instruction memory, interrupts, and system configuration. Moreover, these modules control how internal modules communicate with one another and ensure proper synchronization throughout the design.

They are also responsible for tasks affecting the whole system, such as clock and reset distribution, parameterization, and integrating features like branch prediction. This level sets the foundation for the high flexibility and modularity that the Ibex core represents. It also allows the core to be portable across different ASIC and FPGA targets. Organizing code this way allows for clear separation between computational logic and system-level control, making it easier for developers who want to verify and reuse the design. The clear signal boundaries between the modules for the pipeline, memory, and peripheral interfaces encourage clean hardware abstraction.

6.1.1 ibex_top.sv

ibex_top functions as the central top-level wrapper of the Ibex core. It integrates all major internal components and exposes the standardized interfaces to external systems. Within the file, it instantiates the ibex_core module with its connections to instruction and data memory buses, clock and reset inputs. The debug and interrupt lines are also established. A benefit of this design is the fact that it provides a clean boundary between the Ibex core and the rest of the system. This is important as it allows it to be easily integrated into SoC designs and prototypes. Since this module defines the external interfaces and hierarchy of the design, it acts as the processor's entry point for synthesis and simulation. Furthermore, it ensures the coordination of signals between the other main modules, like the instruction fetch unit, load-store unit, register file, and control logic. Overall, it plays a key role in providing the structural foundation for both functional operation and integration into larger hardware systems.

6.1.2 ibex_top_tracing.sv

ibex_top_tracing acts as a specialized variant of the ibex_top.sv module that incorporates instruction-level tracing. While it has the same functionality and interconnections as the original top module, it brings in extra ports and logic that enable the observation of the processor's internal behavior. With this module, developers are able to collect runtime information like instruction execution, pipeline stage transitions, and register modifications. It goes without saying that this information is key during simulation and verification, since it allows engineers to track performance in fine detail. During the debugging process, the usage of this module can identify pipeline hazards and profile performance bottlenecks. Although this module is irreplaceable when it comes to verification environments, it is not supposed to be synthesized in hardware.

6.1.3 ibex_core.f

Unlike the rest, ibex_core.f is not an RTL module, but a file list used by tools like FuseSoc, Verilator, and commercial EDA environments. It names all source files needed to compile the Ibex core, making sure that the design can be built consistently across different platforms. This file also specifies the correct compilation order, defines dependency paths, and states which configuration files are necessary for simulation and synthesis. By maintaining the file list in ibex_core.f developers can manage sources reliably and avoid mismatches with versions or dependencies. This file greatly simplifies the automation of build processes and upholds portability across projects and their design environments.

6.1.4 ibex_core.sv

ibex_core is the top-level that ties together the fetch (IF), decode (ID), execute (EX), load/store (LSU), CSR, and optional blocks. In the small configuration, many features are compiled out or tied off, yielding a compact 2-stage pipeline (IF/ID+EX) with no separate writeback stage. Instruction fetch uses the prefetch buffer rather than the I-cache (ICache=0), and branch prediction is disabled (BranchPredictor=0). The branch target ALU is also disabled (BranchTargetALU=0), so branches and jumps are resolved over two cycles through the main ALU path.

The core instantiates ibex_if_stage feeding an IF/ID pipeline register into ibex_id_stage, which combines decode, register file, and execute issue. The ibex_ex_block hosts the ALU and the M-extension unit. With RV32M=RV32MFast, ibex_multdiv_fast is selected; RV32BNone removes all bitmanipulation datapaths and opcodes. The optional ibex_wb_stage is bypassed entirely (WritebackStage=0), so EX results and LSU load data write directly to the register file. CSR integration uses ibex_cs_registers; in this configuration, PMP is disabled (PMPEnable=0), debug triggers are off, SecureIbex hardening paths are inactive, and only base architectural counters (mcycle/minstret) are present because MHPMCounterNum=0.

Externally, the core exposes separate instruction and data memory interfaces (simple request/grant/valid protocol), interrupt lines, debug request, and alert/status wiring. Exception and interrupt vectors are managed through CSRs; without I-cache and branch predictor, the control flow is straightforward: IF fetches through ibex_prefetch_buffer, ID/EX decodes and executes, LSU arbitrates data bus accesses, and completed results write back in-place. This setup emphasizes simplicity and area efficiency while keeping fast M operations via the fast mult/div unit and the compressed (C) extension fully supported.

6.1.5 ibex_pkg.sv

ibex_pkg defines a collection of SystemVerilog packages for definitions, enumerations, parameters, and constants shared across the Ibex design. This module could be treated as a repository for configuration settings used across the whole design, like instruction widths, ALU opcodes, control and status register(CSR) identifiers, and

pipeline configuration options. Keeping these definitions in this file further emphasizes Ibex's focus on design consistency, modularity, and readability. If one chooses to modify these architectural parameters like managing instruction extensions or adjusting datapath widths, it can easily be done by editing this package.

6.2 Pipeline Stages and Datapath

These modules, included in the pipeline and datapath group, do most of the computation during execution, as they define how instructions and data move through the processor. Depending on synthesis parameters, Ibex can implement a two-stage or three-stage pipeline configuration. Both of these configurations include the functionalities of the fetch, decode, execute, and write-back stages in some fashion to ensure a steady flow of instructions while minimizing latency and control hazards.

Within this group, the datapath modules take care of the actual computational logic in the core. Things like integer arithmetic, logical operations, and branch condition evaluations are carried out by the arithmetic logic unit (ALU), register file, and optional multiplier/divider units. These components were designed with timing and resource utilization in mind, as most common operations can be executed in a single cycle, while others are pipelined/multi-cycle. The datapath also includes forwarding logic to minimize the effects of data hazards and keep efficiency at a maximum.

6.2.1 ibex_alu.sv

The ALU implements arithmetic, comparison, shift, and boolean operations and is optionally extended by RV32B bitmanip and ternary operations. In the small configuration RV32B=RV32BNone, so all bitmanip instructions (Zba/Zbb/Zbc/Zbs/Zbf/Zbp, etc.) and ternary paths are compiled out. The remaining repertoire covers ADD/SUB, SLT/SLTU, logical shifts and arithmetic shift right, XOR/OR/AND, and comparator outputs used for branch decisions. Operand selection supports immediates (I/S/U/J/B), the current PC, forwarded LSU address, zeros, and register values, controlled by decoder outputs.

The ALU exposes an adder result and a wider extended sum (for div/mul reuse), comparison outputs, and an "imd_val" pair of registers to hold intermediate values for multi-cycle sequences (e.g., rotate or CRC in other configs, or mult/div assistance). In small config, most multicycle ALU ops are absent; however, branch targets without BTALU are computed in a second cycle using the ALU add path, and LSU addresses use ALU add with immediate.

Design-wise, the module carefully manages fan-out by taking a replicated instruction image for ALU control decode and keeps barrel shifters and comparators on critical paths. When RV32B is disabled, related muxing and rs3 paths collapse, improving timing and area. The intermediate value registers (imd_val) act like tiny scratchpads that allow multi-cycle sequences without bloating the pipeline depth. Together with ibex_ex_block, the ALU provides all integer datapath needs for RV32I

and assists the multiplier/divider. Its outputs feed LSU (address adders), the controller (branch decision), and the register file writeback, making it the central arithmetic hub in the small 2-stage pipeline.

6.2.2 ibex_ex_block.sv

ibex_ex_block hosts the main ALU and the multiplier/divider and generates branch decisions and targets. With RV32M=RV32MFast, ibex_multdiv_fast is instantiated (the slow variant is unused); with RV32BNone, the ALU is configured without bitmanip operations. Since BranchTargetALU=0, the branch target output is sourced from the ALU adder result (computed during the second cycle of branches/jumps) rather than a dedicated BTALU path; the BTALU input ports are tied off to avoid lint issues.

The block arbitrates access to a pair of "intermediate value" registers shared between ALU and mult/div to support multicycle operations. It multiplexes those registers and the write-enable strobes depending on whether a mult/div operation is active. The result path selects between ALU and mult/div result based on mult/div selection. Branch decision comes from the ALU comparator outputs and is flopped/used per the ID FSM's needs (DataIndTiming=0 here, so only taken branches force a second cycle).

For the fast mult/div, the module supplies operands and receives readiness/valid signals so ID can stall appropriately when a result isn't ready in the first cycle. Adder results are exposed for LSU address calculation and to assist the divider. In the small config, the overall behavior is: single-cycle ALU for RV32I; multi-cycle for branches/jumps (two cycles), load/store (till LSU responds), and some M ops (depending on fast multiplier internal sequencing). The absence of BTALU and WB stage keeps EX lean; only essential datapaths remain, reducing complexity and area without sacrificing RV32M throughput.

6.2.3 ibex_id_stage.sv

ibex_id_stage covers instruction decode, register file reads/writes, execute issue, and most pipeline control. In the small configuration, the ID stage integrates tightly with EX because there is no separate writeback stage (WritebackStage=0). The stage instantiates ibex_decoder to classify instructions, compute immediates, and drive ALU/LSU/CSR control. With RV32BNone, ternary or bitmanip paths are disabled; with RV32MFast, multiply/divide operations are enabled and routed to the fast mult/div unit in EX.

Without BranchTargetALU, branches and jumps are two-cycle operations. The first cycle evaluates the branch condition (or prepares JAL/JALR), and the second calculates the target using the main ALU. With BranchPredictor=0, there is no predicted-taken plumbing: the not-taken mispredict path is tied off, and branch_not_set is unused. DataIndTiming defaults to 0 here, so only genuinely taken branches force a second cycle; otherwise the instruction retires in one cycle. Load/store requests are initiated from ID and, since there's no writeback stage, the ID stage stalls on the first

cycle of a memory operation and remains stalled until lsu_resp_valid_i returns.

Hazard handling is simple in this setup. With no WB stage, there's no late forwarding: register read data comes directly from the register file, and loads always stall until data is available. CSR accesses are gated by csr_op_en_o only when an instruction is actually executing, avoiding combinational loops with exception flushes. The ID FSM flips between FIRST_CYCLE and MULTI_CYCLE based on instruction type and stall reasons (LSU, mult/div, branch/jump, ALU multicycle). Performance outputs flag branch, taken branch, d-side waits, and mul/div wait cycles. Overall, ID_stage orchestrates a compact, deterministic 2-stage flow tuned for simplicity and area.

6.2.4 ibex_if_stage.sv

ibex_if_stage implements instruction fetch, next-PC selection, and delivery of a decoded/expanded instruction to ID. In the small configuration: ICache=0 selects the ibex_prefetch_buffer path rather than ibex_icache; BranchPredictor=0 removes the skid buffer and prediction plumbing; DummyInstructions=0 disables injection of randomized dummy ops; MemECC=0 disables instruction bus ECC checking; and PCIncrCheck=0 bypasses the optional sequential PC consistency alerting. The stage accepts PC control from the ID/controller (pc_set_i with a pc_mux_i reason), NMI/exception vectors, and jump/branch targets from EX.

Fetched words flow through either the prefetch buffer or, when aligned and available, a direct bypass, then into the compressed decoder to produce a 32-bit instruction image plus a "compressed/illegal" classification. The stage tracks unaligned 32-bit instructions that straddle two words and propagates err_plus2_o to associate a second-half bus error with the correct instruction address. With no branch predictor, fetch_valid is gated only by the prefetch buffer. FENCE.I is treated as a jump to PC+4 and issues a fetch buffer flush; since ICache=0, all tag/data ECC/scramble signals are tied off.

Not-taken mispredict handling is compiled out (there's no prediction), but the IF interface still supports a mispredict input for configurations where BranchPredictor=1. In small config, branch_req simply follows pc_set_i. The IF/ID pipeline register is written when both a valid instruction and ID readiness coincide, carrying two copies of the instruction (for timing fan-out), the compressed 16-bit image (for mtval), error flags, and the current PC. Overall, this stage provides a clean, low-latency fetch path that pairs well with a 2-stage pipeline.

6.2.5 ibex_wb_stage.sv

The optional writeback stage provides a third pipeline stage to hold results and await LSU responses; it also centralizes register file writes and performance accounting. In the small configuration (WritebackStage=0), the module is synthesized into a simple bypass: ready_wb_o is constant 1, rf_write_wb_o is 0, and data flows directly from ID/EX and LSU into the register file mux without a holding stage. The speculative performance outputs are zeroed because the raw counters in ID are sufficient when

there is no WB.

In full WB configurations, the stage latches the instruction type (OTHER/LOAD/STORE), the PC and compressed flag (for performance counters), controls "outstanding" load/store tracking, and forwards writeback data to ID for hazard resolution. It also arbitrates the RF write data between EX results and LSU read data and holds RF write strobes until the LSU response arrives for loads. Since small Ibex removes this, hazards are handled conservatively in ID: loads stall until data arrives and there's no late forwarding. This simplifies timing and area at the cost of slightly higher load-use latency and fewer opportunities for single-cycle resolution of load-use cases. The trade-off aligns with the small configuration's goals: minimal area and straightforward timing closure.

6.2.6 ibex_multdiv_fast.sv

This unit implements the M-extension with a "fast" multiplier and an iterative divider. In RV32MFast mode (small config), multiply is done with either three 17-bit multipliers (single-cycle MUL and two-cycle MULH variant when RV32MSingleCycle is selected) or a 17-bit reuse scheme (3–4 cycles) depending on the RV32M parameter; the default small build uses the fast multi-cycle path integrated here. It sequences partial products and accumulators through a small FSM, using the shared intermediate registers from EX. MUL/MULH variants are handled by different state flows and may hold ID if the result isn't ready immediately.

The divider performs long division with sign handling, running for 32 steps. It computes ABS(a) and ABS(b) (or short-cuts when equal_to_zero indicates division by zero if DataIndTiming is off) and then iteratively subtracts/accumulates quotient bits using the ALU adder inputs routed from EX. Sign correction is applied at the end. Flow control supports holding the final result until ID/EX is ready to accept it, preventing combinational feedthrough paths.

In small Ibex, this block is the only M-extension hardware: it receives enables from the decoder (mult_en/div_en), selector hints, and signed mode. With RV32MNone it would synthesize away; with RV32MFast it's fully included. Combined with the absence of WB stage, results are visible to the register file as soon as EX asserts valid and the instruction completes.

6.2.7 ibex_multdiv_slow.sv

The slow multiplier/divider provides an alternative M-extension implementation using a classic Baugh-Wooley multiplication scheme and an iterative divider that shares a compact datapath. It is not used in the small configuration (RV32MFast selects ibex_multdiv_fast), but it remains in the tree for other build profiles. The multiplier assembles partial products over multiple cycles by shifting operands and accumulating results via the shared ALU adder interface. MULH variants take an extra cycle to combine high partial products. The divider mirrors the fast divider conceptually (ABS, iterative subtract/compare, sign fix-up) but reuses more state, trading latency for area.

Interface-wise, it matches the fast unit: it consumes mult/div enables, operators, signed mode, and operands; it emits valid/data and uses the shared "imd_val" registers. When unselected, synthesis eliminates its logic entirely. From a system perspective, documenting it here matters because CSR MISA reports M if RV32MFast is enabled, and the decode will generate mult/div ops which target whichever unit EX instantiated. In small Ibex, any references to this file are effectively dead code, but it documents the alternative timing/area points supported by the design.

6.3 Control and Decode Logic

The control and decode logic group manages how the core interprets instructions as they pass through the pipeline. This group includes components like the instruction decoder, control signal generator, and logic that supports branching, stalling, and exception handling. On top of the basic decoding, these modules also manage pipeline control mechanisms to ensure correct instruction sequencing, which include the detection of data and control hazards, as well as the insertion of stalls and bubbles. Additionally, the branch control logic predicts future instruction addresses, reducing the chances of wasted cycles from changes in the control flow.

The synchronization between the decode logic and the rest of the pipeline is extremely important when considering the overall performance of the system. The modular structure of this group lets extensions be easily added to the instruction set, allowing for Ibex to be a valid option for a multitude of different applications. By maintaining a clean separation between decoding, control generation, and execution, this group ensures the pipeline remains deterministic, efficient, and easily verifiable.

6.3.1 ibex_branch_predict.sv

This file implements a simple static branch predictor that recognizes unconditional jumps (JAL, compressed J and JAL), indirect JALR patterns, and conditional branches with a negative-offset-taken heuristic. It outputs two pieces of information to IF: a predicted taken flag and a predicted target PC. In the small configuration, BranchPredictor=0 disables its integration: ibex_if_stage won't instantiate either the predictor or the associated skid buffer, so fetch proceeds strictly in program order, and ibex_controller does not receive predicted-taken feedback. Even so, understanding this module is helpful if the design is later scaled up.

Internally, the predictor inspects the raw 32-bit fetch word (before decompression) and the fetch PC. It extracts immediates for B-type and J-type encodings and detects compressed control-flow forms. For conditional branches, it applies a classic static policy: backward (negative offset) branches are predicted taken; forward branches are not. For JAL/JALR/compact jumps, it predicts taken and computes a target as PC+imm (J/JAL) or an early guess for JALR (exact only once rs1 data is known in ID). These predictions are only consumed when IF successfully captures an instruction; a skid buffer can hold predicted-taken instructions if ID is not ready, decoupling ready/valid paths and avoiding a comb loop from data bus grants back

into IF request.

Mispredict recovery is also defined here. When ID determines a predicted-taken branch is actually not taken, it raises nt_branch_mispredict to IF. The prefetch buffer then discards the predicted path and continues from the fall-through PC, which IF can supply immediately if the buffer already fetched it. Assertions constrain timing: IF must see the mispredict before accepting the instruction after a predicted branch, and the next PC on correction must match PC+2/4 based on compressed state. In the small Ibex build the entire block synthesizes away, eliminating its area and timing impact while leaving the interfaces intact for future enablement.

6.3.2 ibex_compressed_decoder.sv

he compressed decoder expands 16-bit RVC instructions into equivalent 32-bit RV32 encodings and flags illegal encodings. It covers all three compressed quadrants (C0/C1/C2), mapping to ALU-immediates (C.ADDI, C.SLLI/SRLI/SRAI), loads/stores (C.LW/C.SW and their stack-relative forms), control flow (C.J, C.JAL, C.JR, C.JALR), and integer register moves/LI/LUI. It also checks architectural constraints like disallowing writes to x0 when the compressed form requires a non-zero rd/rs. The module is fully combinational and sits in IF, after fetch alignment and before the main decoder. For the small configuration, there are no feature gates here: all valid RVC forms are emitted, and B-extension-specific compressed forms are not relevant since RV32B=RV32BNone.

Inputs are the raw instruction word, a valid qualifier, and the reset. Outputs include the expanded 32-bit instruction, a boolean that the source was compressed, and an "illegal" flag for malformed encodings. The "compressed" bit propagates into ID and the controller for accurate mtval reporting and correct PC increment (2 vs 4). Because compressed instructions alter instruction boundaries, this decoder works closely with ibex_fetch_fifo, which may assemble a 32-bit instruction from two halves across adjacent words. The decoder itself is not responsible for alignment; it only interprets the 16 or 32 bits presented from the IF aligner.

Error handling is explicit: if the input was flagged as valid and corresponds to a bad compressed opcode, illegal_instr_o is asserted and the downstream decoder will treat the instruction as illegal, disabling side effects. This prevents any architectural state change on malformed encodings. In practice, enabling RVC significantly improves code density and fetch efficiency. Even without an I-cache, the prefetch buffer benefits because more useful work fits in the same bus bandwidth when instruction streams consist of a mix of 16- and 32-bit operations. Assertions in the IF/ID path ensure the replicated instruction copies match exactly, avoiding X propagation that could obscure decoder behavior in simulation.

6.3.3 ibex_controller.sv

The controller is the central FSM for instruction flow, exceptions, interrupts, debug entry, and PC control. It consumes decoder flags (illegal, system ops, wfi, mret/dret/ecall/ebreak), IF/ID instruction validity, fetch errors, and LSU excep-

tion indicators. It generates pc_set and pc_mux selections, exception PC and cause, flushes, and IF/ID stage gating. In small Ibex: WritebackStage=0 simplifies exception prioritization and "writeback-pending" cases; BranchPredictor=0 removes predicted-taken and not-taken mispredict paths; MemECC is off, so internal NMI paths for instruction/data integrity errors are inactive.

The controller sequences reset/boot to the first fetch, handles synchronous exceptions (illegal instruction, ecall/ebreak, misaligned or erroring fetch) and asynchronous interrupts based on CSR masking, and controls sleep via WFI. For jumps/branches, it listens for one-shot jump_set/branch_set from ID and intentionally holds these to a single cycle to avoid repeated flushes. With no BTALU, taken branches are two-cycle operations; otherwise, most non-memory instructions complete in a single ID/EX cycle.

Debug entry is supported via debug_req_i, with DmHaltAddr and DmExceptionAddr managed by IF, but triggers are disabled in this configuration. CSR save/restore signaling (mepc/depc, cause, mtval) is coordinated here; ibex_cs_registers holds the architectural state. The lack of a writeback stage means many "pending memory access" interlocks vanish, making the controller compact. It still interfaces cleanly with the LSU for precise exceptions.

6.3.4 ibex_decoder.sv

ibex_decoder is a fully combinational decoder producing ALU operations, immediate selections, register file addresses/enables, LSU control, CSR operations, and special signals for exceptions and flow control. With RV32E=0 no GPR narrowing checks apply; with RV32BNone, all bitmanip encodings are flagged illegal, and ternary instructions are removed; with BranchTargetALU=0, JAL/JALR/BRANCH target computation is split across cycles and uses the main ALU rather than the BTALU.

The decoder covers full RV32I plus RV32C (via a separate compressed decoder in IF) and RV32M. It emits mult_sel/div_sel and the md operator/signed modes for M-extension operations while guarding them with RV32M!=None. Load/store instructions set data_req/we/type/sign_ext appropriately and rely on the LSU to handle misalignment and sign extension. CSR instructions generate csr_access and csr_op, with additional operand "cleanups" to force CSRRS/CSRRC reads when rs1/uimm is zero as per spec. The decoder asserts ecall/ebreak/mret/dret/wfi flags for the controller and issues icache_inval_o on FENCE.I (leading IF to flush the prefetch buffer).

On any illegal instruction (including illegal compressed opcodes propagated from IF) the decoder clears write enables and side effects to avoid touching architectural state. It also provides two separate instruction replicas (instr vs instr_alu) to reduce fan-out along the critical ALU timing path. For branches, it selects ALU comparators (EQ/NE/LT/GE/LTU/GEU) and, with no BTALU, directs a second ALU cycle for target add. In the small configuration, this module cleanly strips out RV32B and keeps only what's needed for RV32I+M, ensuring tight decode timing and minimal area.

6.3.5 ibex_dummy_instr.sv

This module implements pseudo-random dummy instruction insertion for control-flow obfuscation and side-channel hardening. It uses a configurable LFSR (seed and permutation constants provided by parameters and CSRs) to generate a small record that selects an instruction type (ADD/AND/MUL/DIV), two small operand indices, and a counter threshold that spaces out insertions. When enabled via CSR, the IF stage asks this block each cycle whether to insert. If the counter equals the threshold and ID is ready, insert_dummy_instr_o asserts and a synthesized 32-bit instruction is emitted. The instruction encodes an intentionally harmless operation (e.g., ADD x0, rs1, rs2) or one whose result is ignored by design.

Integration details matter: IF stalls for one cycle when inserting to ensure the dummy executes between the current ID instruction and the next fetched real instruction, preserving architectural ordering and precise exceptions. The dummy's PC equals the fall-through PC from the prefetch buffer so debug/trace remain coherent. CSR controls allow software to enable/disable dummy insertion, adjust the rate via a mask, and reseed the LFSR (XOR-based reseed to avoid trivial zeros). The design also ensures that insertion doesn't propagate bus or PMP errors (those are suppressed for dummies) and that instruction legality is guaranteed (no compressed form is used; dummy is emitted as a 32-bit OP encoding).

In the small configuration, DummyInstructions=0 in IF, so the entire datapath is compiled out and these CSRs have no effect. However, the file documents a drop-in path to introduce unpredictability in instruction streams, which can help mitigate timing or power analysis in security-sensitive deployments. Because the dummy instruction set is limited and side-effect free, functional correctness is unaffected; at worst, it increases runtime in proportion to the configured insertion rate.

6.4 Memory System and Bus Logic

This group of components manages how the Ibex core deals with the instruction and data memories, as well as external system buses. These modules instantiate the necessary logic to create and handle memory requests, buffer data, and interface with standard bus protocols. The Ibex core usually follows the Harvard Architecture, meaning that it employs separate instruction and data buses. This allows for concurrent access and fewer memory access bottlenecks. Each bus interface handles address generation, read/write control, and handshaking signals for communication with memory or peripheral devices. Moreover, these modules make sure that data integrity and coherence are upheld when multiple memory operations are active. In summary, this group acts as the communication backbone between the processor and the external world, bridging the computational core with its data environment.

6.4.1 ibex_fetch_fifo.sv

ibex_fetch_fifo is a specialized FIFO buffering instruction words and aligning halfword-based compressed instructions. Because RVC allows 16-bit opcodes, a 32-bit fetch can contain one 32-bit instruction or two 16-bit ones, and an unaligned 32-bit instruction may straddle two adjacent words. The FIFO stores word-aligned fetch data and exposes a halfword-aware read port that reconstructs the next instruction boundary based on the current PC LSBs (pc[1]).

In practice, the module computes aligned and unaligned instruction candidates: for unaligned cases, it concatenates the upper half of the previous word with the lower half of the current word and gates validity on the presence of both halves. It also tracks error sources across boundaries: err_unaligned merges error indications from either contributing word and sets an "err_plus2" bit when the second half of a 32-bit instruction caused the error. Validity for unaligned 32-bit instructions is only true when both halves are available; compressed instructions can be emitted as soon as their halfword is present.

Internally, a small array of entries stores rdata/err/valid flags. The FIFO writes to the lowest free entry and pops entries as the IF stage consumes instructions, but popping may leave an entry partially valid when an unaligned 32-bit instruction spans two words. The FIFO also keeps a rolling PC (instr_addr_q) that increments by 2 for compressed or 4 for uncompressed instructions, reflecting the output stream's actual instruction addresses. In the small config, this is the key alignment mechanism feeding the compressed decoder: it ensures correct instruction assembly independent of bus timing and fetch burst alignment, preserving precise exception behavior.

6.4.2 ibex_icache.sv

ibex_icache implements an instruction cache subsystem meant to improve the processor's efficiency by reducing the latency from instruction fetch and the number of main memory accesses. The cache holds the recently used instruction lines and tries to provide faster access for fetches that come after. It contains components similar to those of caches for memory in commercial cores. This module includes logic for cache line fills, tag comparison, and miss detection. When misses inevitably occur, the module has the logic necessary to coordinate with the external memory interface to fetch required instructions and update its cache contents. Moreover, it supports cache invalidation as well as prefetching mechanisms to maintain consistency. The use of this module plays a major role in improving execution throughput by avoiding slower memory accesses.

6.4.3 ibex_load_store_unit.sv

The LSU arbitrates and sequences data bus transactions, handling byte/halfword/word accesses, misaligned operations (by splitting into two aligned requests), byte enables, sign/zero extension, and error reporting. In small Ibex, MemECC is off, so there's no SECDED decode/encode; the bus data path is 32 bits.

The LSU computes byte enables based on the data type and address offset, realigns store data with a rotate-by-byte scheme, and reconstructs load data with appropriate sign extension. For misaligned word or halfword accesses, it issues two back-to-back aligned transactions, capturing intermediate data and tracking errors across both halves.

The FSM covers IDLE, WAIT_GNT (waiting for grant), WAIT_RVALID_MIS (await first response while issuing second half), and WAIT_RVALID_MIS_GNTS_DONE (first rvalid arrived, second already granted). It records PMP and bus errors; in this configuration, PMPEnable=0 means upstream PMP checks are tied off, but the LSU still wires inputs for pmp_err to support other builds. The LSU signals lsu_req_done_o when an issued request has progressed such that only the response is outstanding; in small config with no WB stage, ID stalls on any outstanding access and resumes when lsu_resp_valid_o asserts. Integrity error outputs are distinct to avoid comb paths into request generation.

Overall, LSU integrates cleanly with the 2-stage pipeline: ID issues the request, EX provides the computed address, LSU manages bus protocol and alignment, and the result/write completes in ID once the response returns. This keeps precise exceptions (load/store/align/PMP) and mtval reporting correct, with minimal hardware.

6.4.4 ibex_pmp.sv

ibex_pmp is a generic PMP implementation supporting TOR/NA4/NAPOT modes, MSECCFG features (MML/MMWP/RLB), and priority resolution across multiple regions and channels (I/D). In the small configuration PMPEnable=0, so this module is not instantiated and CSR writes to PMP CSRs have no effect on access gating. Nevertheless, the module defines how addresses are matched (including configurable granularity), how permissions are checked (original PMP behavior versus Smepmp MSECCFG rules), and how debug mode access into the DM address range is always allowed.

When enabled, IF and LSU send addresses and access types (EXEC/READ/WRITE) through channels; the PMP compares against each region (lowest index wins) and returns an error if permissions disallow the access. The module supports TOR (top-of-range) regions (bounded by the previous address and current address), NA4 (4-byte naturally aligned), and NAPOT (power-of-two naturally aligned) encodings with a configurable minimum granularity. Smepmp extensions (MSECCFG.MML/MMWP) alter permission semantics, especially the meaning of the lock bit in M-mode and default deny behavior in non-matching cases, which this RTL captures in separate helper functions. It also whitelists Debug Module accesses while in Debug Mode regardless of PMP entries, as mandated by the RISC-V Debug Spec.

In small Ibex, access errors (pmp_err) are tied low at the sources. Nevertheless, the IF and LSU already route PMP error inputs into their exception machinery, and ibex_cs_registers exposes PMP CSRs, allowing you to flip PMPEnable later with minimal wiring changes. If you plan to enable PMP, consider how many regions you need (PMPNumRegions), granularity constraints imposed by your memory map, and

whether Smepmp semantics are desired. The module's static priority (lowest index wins) and per-channel evaluation make its timing predictable; synthesis prunes unused channels or regions automatically based on parameters, keeping it suitable for small and large builds alike.

6.4.5 ibex_prefetch_buffer.sv

The prefetch buffer is a small, request-smoothing fetch front-end for a 32-bit instruction memory interface. In small Ibex, it replaces the I-cache and aims to keep IF supplied without long critical paths. It issues aligned word requests, tracks outstanding grants/responses, and pushes completed words into an internal FIFO (ibex_fetch_fifo). A simple "branch" input clears the FIFO and resets the internal address tracking, which FENCE.I also relies on to force flushing and restart fetch from the new PC.

The buffer maintains two addresses: a stored "in-flight" bus address (stable until grant) and a next fetch address that increments by 4 on each issued request. When a branch occurs, both the prefetch address and the FIFO are reset to the branch target, and any in-flight responses corresponding to the old stream are discarded based on branch_discard tracking. The design supports up to two outstanding requests (configurable via NUM_REQS in the FIFO), overlaying FIFO fill state with outstanding request slots to decide when it's safe to issue another request.

On response, valid words are forwarded to the FIFO unless canceled by a branch. Errors propagate to IF along with an err_plus2 indicator to mark faults on the second half of an unaligned instruction. With ICache=0 and BranchPredictor=0, this module carries the full burden of keeping fetch steady; there's no prediction feedback loop, so valid_o simply reflects availability of the next instruction word. The prefetch buffer excels in area-limited builds: it avoids tag/data RAMs, integrates easily with simple busses, and keeps IF decoupled from bus timing while preserving exact exception semantics and PC tracking.

6.5 System and Debug Architecture

This group includes modules that manage debugging, exceptions, and system control, extending Ibex beyond basic instruction execution. It allows developers to do things like monitor the processor state, set breakpoints, and step through code. These system-level components also handle interrupts and exceptions, making sure that faults and asynchronous events are dealt with properly. They also include performance counters and system control features to provide insights for optimization.

6.5.1 ibex_counter.sv

ibex_counter implements a parameterized up-counter used across the design for architectural and performance counters. It supports an optional "ProvideValUpd" output to surface the next value combinationally, which can help inference of hardware

carry chains/DSP blocks on some FPGAs, improving Fmax. Width is parameterized; in Ibex, counters can be 40+ bits internally and exposed as 64-bit CSRs via pairs. Reset behavior and enable gating are explicit, and the design favors simple adders without exotic control logic.

In the small configuration, MHPMCounterNum=0 leaves only the base mcycle and minstret implemented in ibex_cs_registers. mcycle increments unconditionally each cycle the core is running; minstret increments when an instruction retires. With WritebackStage=0, retirement is qualified by the ID stage's instr_perf_count_id_o and en_wb_o, ensuring traps and illegal instructions aren't counted. The module's minimalistic interface fits this usage well: it can be clock-gated externally if needed, or run free with a simple increment enable.

A practical concern is how these counters interact with CSR inhibit bits and low-power modes. ibex_cs_registers can gate their enables (e.g., mhpminhibit) or snapshot values on reads; ibex_counter just holds the state. Because it's reused, it also covers potential future expansion (adding mhpmcounters when MHPMCounterNum¿0) without changes to the primitive. In bring-up and benchmarking, these counters are vital for measuring CPI, loops, and latency hotspots. Even in the feature-reduced small build, having a robust, timing-friendly counter primitive avoids pitfalls like long ripple carry chains that could spill onto critical paths.

6.5.2 ibex_cs_registers.sv

ibex_cs_registers implements CSR state and decode for mstatus/misa/mie/mip/mtvec/mepc/mcause/mtval and optional blocks like PMP CSRs, debug CSRs, and mhpmcounters. In the small configuration, PMPEnable=0 ties off PMP CSR effects (the ibex_pmp block isn't instantiated); MHPMCounterNum=0 disables implementation of extra mhpmcounters, leaving only mcycle and minstret; DbgTriggerEn=0 removes trigger CSRs; I-cache controls and dummy instruction CSRs exist but have no effect since those features are off. MISA reflects RV32IMC (E=0, B=0, M=1, C=1) for our parameters.

The block arbitrates CSR reads/writes, privilege checks, and side effects (like flushing IF on certain CSR writes). It provides outputs to the controller for exception/interrupt handling, save/restore of mepc/depc and cause, and wiring for performance counters with inhibit bits. Shadowed CSR storage (via ibex_csr) can be used for integrity checking, but SecureIbex=0 means broader hardening paths are inactive. WFI trapping obeys mstatus.TW, enforced in ID.

In this build, the integration is straightforward: minimal counters reduce area, PMP is unused, and all interface signals for disabled features are statically tied. The module remains central for interrupt enable/mask, mtvec base, trap vectoring, and capture of mtval on exceptions. Its clean parameterization lets the same RTL scale up to feature-rich builds without code divergence.

6.5.3 ibex_csr.sv

This is a primitive CSR storage element with optional shadow copy for error detection. It parameterizes width, reset value, write mask, and whether to use a redundant shadow register that can detect mismatches on update, setting an error bit. Many CSRs in ibex_cs_registers are composed from these primitives (one per field or per CSR), which keeps the top-level CSR logic compact and uniform. The interface usually takes a write enable and data, applies a mask, and exposes the stored value along with a shadow error indicator if shadowing is enabled.

In the small configuration, behavior is straightforward: SecureIbex=0 implies no global hardening mandate, so most CSRs are single-flop storage with side-effects implemented in ibex_cs_registers. Still, the primitive supports useful patterns: WRITE, SET, and CLEAR operations are handled by composing next-state logic externally; this cell just captures the result. For fields that are WARL (write-any, read-legal), the enclosing CSR logic can sanitize on write and feed the accepted value here. The shadow option, when enabled for a given CSR, doubles the storage and checks consistency when writes occur, which can be used to detect glitch or fault injection in security-hardened builds.

From a verification and synthesis perspective, the module includes simple assertions to guard against unknowns on control inputs and ensures reset behavior is well-defined. Because it's a leaf cell with minimal logic, it won't impact timing in the small pipeline, and its reuse avoids hand-rolled flops scattered throughout the CSR file. If you later enable security features, many critical CSRs can be switched to shadowed mode without refactoring the broader design, since all call sites already funnel through this primitive. That separation of concerns is the main value here, beyond basic state storage.

6.5.4 ibex_lockstep.sv

ibex_lockstep implements redundancy features for fault-tolerant and safety-critical applications. In the current configuration, two identical Ibex cores are kept in "lockstep" mode, meaning that they execute the same instructions simultaneously. This is meant to allow the module to continuously compare their outputs to detect possible mismatches, signifying something went wrong. If a mismatch does occur, the system can trigger exceptions or error signals. This approach is also common in industries like automotive, aerospace, and medicine, as there are many safety standards that they must adhere to in which error detection is paramount. The ibex_lockstep file contains the comparison logic, synchronization structures, and configuration parameters needed to enable this redundancy system.

6.5.5 ibex_register_file_ff.sv

The ibex_register_file_ff is one of the versions of the core's register file. This implementation is created using flip-flops. This makes it optimized for ASIC synthesis, as it provides high performance and predictable timing characteristics. The register

file contains all 32 general-purpose registers listed in the RISC-V ISA while also supporting functionalities like dual read ports and a single write port, enabling reads and writes to be executed simultaneously during instruction execution. Because this module is based on using flip-flops, it offers great performance in custom silicon, but consumes more area than other approaches. This implementation is used to emphasize deterministic timing and high clock speeds.

6.5.6 ibex_register_file_fpga.sv

ibex_register_file_fpga implements the same register file interface as the flip-flop module mentioned above, but is optimized for FPGAs. This module relies on the FPGA's embedded block RAM rather than completely on flip-flops. This implementation conserves logic resources and improves timing closure on FPGA devices. These devices also usually have dedicated memory blocks optimized for this purpose. Although this is slightly slower than the flip-flop design, it has a good balance between performance, area, and resource utilization, making it good for things like hardware prototyping and low-cost implementations.

6.5.7 ibex_register_file_latch.sv

ibex_register_file_latch provides the last file register implementation that uses latches instead of flip-flops or block RAM. The main advantage this version offers is the reduced power consumption and clock tree load, making it very appealing for low-power applications. The latch-based register file can manage this because it leverages techniques like clock gating and transparent latch behavior to minimize dynamic power usage without sacrificing correct read/write functionality. The main downside to this implementation is the fact that timing verification may be more complex and less portable across technologies. Despite this, the latch register file is still valuable, as it can significantly contribute to overall energy savings.

6.5.8 ibex_tracer.sv

ibex_tracer implements the instruction-level tracing and event recording for the Ibex core. It measures multiple internal signals like instruction fetches, program counter updates, register writes, and memory transactions. This is necessary because observing these signals in real time makes it possible for developers and verification engineers to gain more insight into the processor's internal behavior. This role is invaluable in the debugging process, especially when trying to verify complex software interactions. Furthermore, ibex_tracer enables developers to collect detailed performance metrics like branch frequency, stall cycles, and pipeline utilization.

6.5.9 ibex_tracer_pkg.sv

The ibex_tracer_pkg file assists the tracer module by defining the data types, constants, and structures needed for it to consistently represent trace events. It

maintains all of the format definitions, signal encodings, and configuration parameters necessary for the tracer to function. Instead of combining both into a single module, this organization promotes clean separation between logic implementation and data specification. This package enforces standardization, simplifying integration with external analysis scripts or automated verification environments. Combined with the functionality of ibex_tracer, it forms a cohesive subsystem that offers invaluable visibility into the Ibex core's runtime activity. Below is a photo of the core in its entirety.

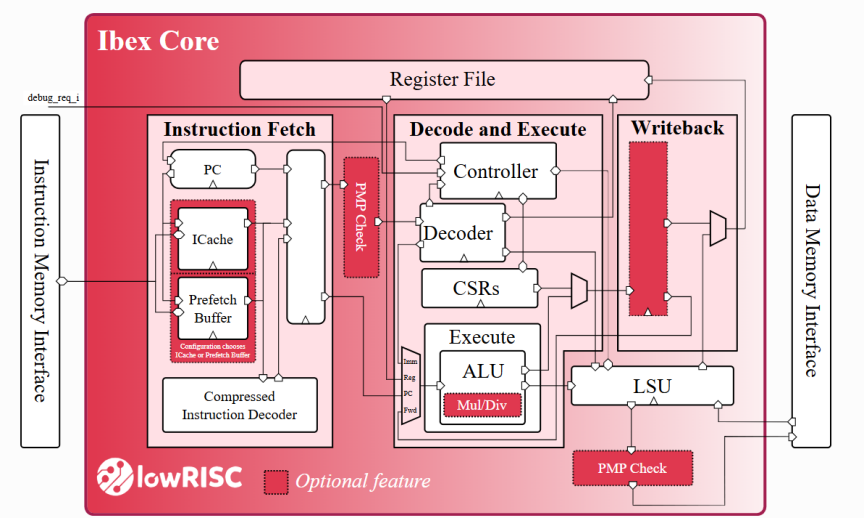


Figure 6.1: *Diagram of Ibex Core*

6.6 Core Redesign for Security

This section will cover which methods discussed earlier in the report to mitigate side channel analysis attacks will be implemented. As many of these techniques were used in past research papers, choosing techniques that were agnostic of the design was ideal. As these techniques are implemented, special consideration and evaluation has to be made in order to ensure that the techniques being implemented are compatible with the core. The specific section, included in this section will be related to Boolean Masking, clock randomization, and random-noise generation.

6.6.1 Randomization

True randomness comes from physical processes such as thermal noise, quantum effects, or radioactive decay, which are inherently unpredictable and continuous in nature. RTL, or Register Transfer Level, representations describe digital logic using flip-flops, combinational gates, and sequential circuits with precise state transitions. Synthesis converts RTL into a fixed gate-level netlist, which behaves deterministically given specific inputs and clocking. Therefore, any random behavior implemented

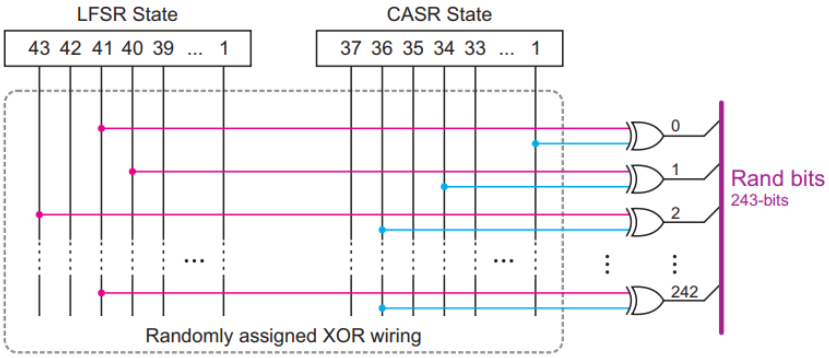


Figure 6.2: *Pseudo-random Number Generator made with LFSR and CASR[1]*

purely in RTL is pseudo-random, generated by algorithms rather than actual physical unpredictability. Without incorporating analog or physical entropy sources, RTL and synthesis cannot produce true stochastic behavior and are limited to approximations.

Given that this project is focused on the RTL-to-GDSII flow, pseudo-random number generators are the ideal method to generate fresh new random numbers. Specifically, using a pseudo-random number generator with a long periodicity as it ensures that the sequence of generated numbers does not repeat for a very long time which effectively maximizes randomness over extended use.

The psuedo-random number generator chosen for this project is a mixture of a 43-bit LFSR and 37-bit CASR whose bits will be XORed with each other to form a 243-bit pseudo-random value. The order in which the bits are XORed and written to one of the 243 bits can vary and is randomly assigned during the design phase. A diagram of this is presented above.

6.6.2 Boolean Masking to Mitigate First Order Attacks

As stated in the research section, Boolean Masking seeks to transform data being operated in the circuit by splitting it into shares that are independent of the data and operating on these newly generated shares. To protect against an n^{th} order attack, $n+1$ shares need to represent the data in the circuit. Given that our goal is to mitigate first order attacks, we require two shares.

Since the data is split into two shares, how the shares are operated on to ensure that it can be used to build the original data is important. Thus, the standard AND and OR gates must be converted into an equivalent masking/secure gates that take in the two shares as input. This project opted to use the masking method presented by Biruykov [29] as it reduces the amount of random numbers need for nonlinear operations (AND and OR). The figure below shows the masked versions of these non-linear operations as well as how shares are generated and used to rebuild the original data. Linear operations such as XOR, shifts, and permutations do not require modification and are applied to both shares.

Implementing these new masked gates can be accomplished by updating the

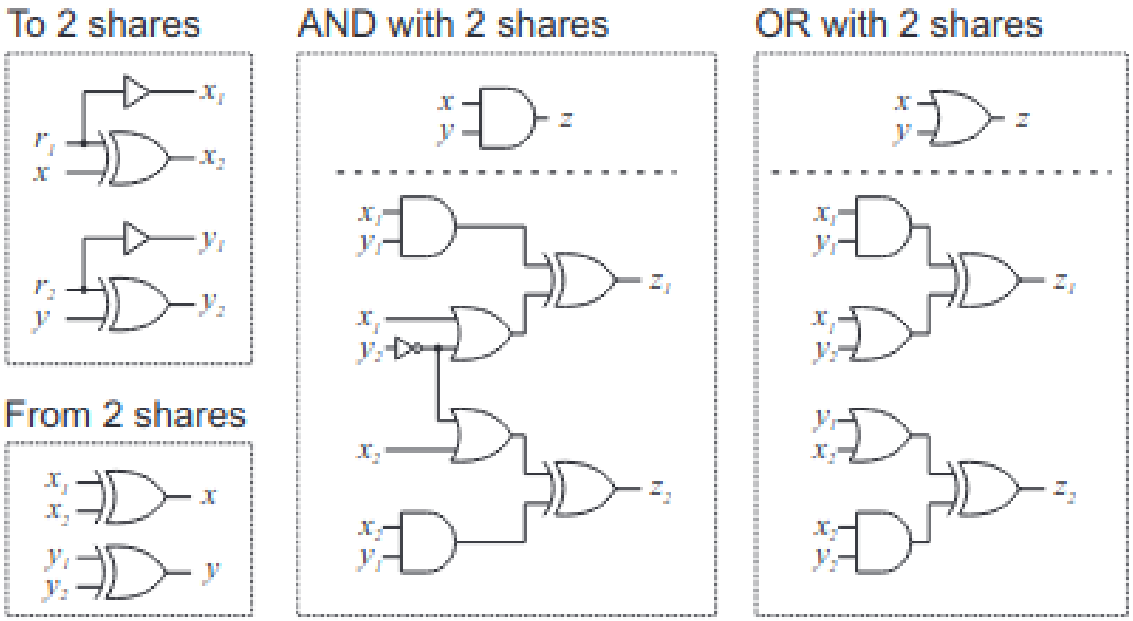


Figure 6.3: *Placeholder for Masking versions for the non-linear gates*

Tcl files that control the steps of synthesis in the Cadence Tools. Instead of having Cadence go through the full-synthesis steps (syn_generic, syn_map, and syn_opt), the syn_opt step is replaced with a swapping step that replaces the original gates into their masked/secure versions. The syn_opt step is skipped because this optimizes the circuit which may modify the secure gates opening up new leakages in the circuit. Work done by Stangherlin[1], has provided an outline for these Tcl scripts which will be used as a framework for this project.

The synthesis and Boolean masking flow is implemented through two interconnected `.tcl` scripts provided by the CMO framework. The first script initializes the synthesis environment by defining the technology libraries, LEF and capacitance tables, and setting the process node to 65nm. It then specifies the design hierarchy by identifying the top-level module, in this case the SERV-based Ibex configuration, and disables automatic optimizations such as clock-gating insertion and hierarchy ungrouping to preserve the original module structure during masking.

After setting up the environment, the unmasked RTL source files are read and elaborated, followed by a generic and mapped synthesis stage that produces an initial unmasked gate-level netlist. This intermediate netlist serves as the basis for the masking transformation. The second phase, invoked through the masking script, begins by creating shared top-level and hierarchical ports to enable the propagation of masked signal shares throughout the design hierarchy.

Next, the total number of sequential elements (flip-flops) is determined. Using this count, a new bus signal called `randbits` is created. This bus represents entropy inputs generated externally by a pre-synthesized pseudo-random number generator (PRNG). The `randbits` signal is inserted at the top level and propagated down through each module instance, providing unique randomness to every masked register

and combinational gate.

The masking transformation proceeds by replacing all standard logic cells—such as inverters, AND, OR, XOR gates, and D-type flip-flops—with their corresponding secure Boolean-masked cell implementations. The substitution process uses pin mapping definitions to maintain functional equivalence while introducing first-order masking at the gate level. This ensures that each logic operation operates on two statistically independent shares of the sensitive data.

Following cell replacement, the scripts connect the shared masked signals (`_s1` suffix) and the random entropy bus to the appropriate input and output pins across the design hierarchy. This hierarchical connection mechanism guarantees consistent signal propagation between modules and prevents partial unmasking due to unconnected ports.

Finally, the flow generates synthesis reports for area and gate count, along with header files defining the number of random bits and maximum circuit depth. These outputs facilitate both design verification and subsequent integration with the PRNG subsystem. The resulting netlist thus represents a fully Boolean-masked version of the Ibex-based processor, automatically derived from the unmasked design while preserving modularity and timing integrity.

6.6.3 Random Noise Injection Implementation

As discussed in the research section of this paper, random noise injection is a hardware-level countermeasure designed to obscure the correlation between a circuit's power consumption and the sensitive data being processed. This technique is particularly relevant in mitigating side-channel attacks, where adversaries attempt to extract confidential information by analyzing power traces. In the implementation described here, a Gaussian Noise Generator (GNG) is employed to produce a sequence of random bits. These bits as well as control signals in the circuit are used to generate the select signal for a multiplexer, which in turn enables one of three high-activity circuits: a second Gaussian Noise Generator, a ring oscillator, and another LFSR. The selection mechanism ensures that only one of these circuits is active at a given time, introducing variability in the power consumption pattern of the overall /6 system.

The choice to utilize circuits with high switching activity is deliberate. Circuits that toggle frequently exhibit increased dynamic power dissipation, which in turn produces larger and less predictable fluctuations in the instantaneous power consumption of the device. By injecting such high-activity components, the deterministic patterns in the power traces that might otherwise correlate with the processed data are effectively masked, making it significantly more difficult for an attacker to infer sensitive information. This approach leverages the principle that higher toggling rates generate more pronounced and randomized variations in the power profile, thereby enhancing the effectiveness of noise-based obfuscation techniques in protecting against power analysis attacks.

6.7 Verification Architecture

6.7.1 Objectives and Approach

Our initial strategy is to first reproduce the behavior of the unmodified Ibex core in its intended verification environment/flow and establish this as our regression anchor. Once this baseline is established, we will incrementally extend the environment to cover the various features (Boolean masking, clock randomization, new instructions/extensions) whilst preserving reuse overall regression stability. The guiding principle is separation of concerns: architectural correctness will be checked independently of timing properties, such that each change to the RTL has a clear verification target and unambiguous pass/fail criteria.

6.8 Post-Synthesis Verification

6.8.1 Masking after Applied Synthesis

As established in §2.2.1.3 and §3.6.1.6, our Boolean masking is inserted after RTL synthesis via SERV-style Tcl edits to the synthesized netlist [29]. Because the masking logic does not exist in the RTL source, an RTL-only verification workflow cannot validate the presence, placement, or timing behavior of the final masked datapaths. Consequently, we need to verify at the netlist level to handle a few scenarios: (i) the masking structures were actually instantiated and wired as intended, (ii) synthesis and downstream optimizations did not fold away or reshare/leak mask signals, and (iii) the resulting gate-level implementation still satisfies our architectural checks and timing/side-channel constraints (constant-time windows or bounded randomized latency).

From a practical standpoint, this means that we aim to reuse the baseline Ibex UVM environment, but swap the Device Under Test (DUT) for the synthesized (and potentially SDF-annotated, depending on toolchain capability [45] [46]) netlist that includes the Boolean masking implementation. From here, we can address the three concerns mentioned prior: (i) we can demonstrate architectural and logical equivalence in system behavior (LEC [47] or observational equivalence, where applicable), (ii) run zero-delay and SDF gate-level simulations to confirm reset and CSR/PMP behaviors, as well as instruction behavior under realistic delays, and (iii) collect latency distributions from our UVM subscribers, verifying whether constant-time paths remain uniform and that randomized paths meet statistical bounds.

This post-synthesis step is not a typical sign-off activity; it is the only place where we can prove the implemented SERV-style masking matches the design intent and remains intact through the tool flow.

Chapter 7 - System Testing and Evaluation

This section explains various testing methods that can be used on the Ibex core before and after adding our SCALER extension. The main method used would be to collect measurements that specifically show changes in leakage rather than a pass/fail test on functionality. Although we will be using ASIC design for our project, physical measurements on an FPGA would enable us to see power-based leakage on a live system, something that is more difficult to observe in simulation. We will first characterize the unmodified Ibex, then repeat the same procedures after integrating SCALER and modifying the core, using identical firmware and capture settings for fair comparison.

7.1 Testing Methods

For testing, we will measure the behavior of power-analysis side-channel attacks and implement techniques to measure timing leakage. The timing measurements will be used to confirm minimal timing variations in memory access patterns, which makes the processor less susceptible to timing-based attacks. Our core measurements will use Simple Power Analysis (SPA), Differential Power Analysis (DPA), and Correlation Power Analysis (CPA) on AES workloads that are executed by the Ibex core. Both CPA and DPA require traces collected from encryption, and the traces are used to correlate the measured power with the values of the cipher.

7.1.1 Capture Setup

To make these measurements consistent, we will run AES-128 with a fixed key per capture session and a known plaintext list. The firmware will raise a small GPIO "trigger" right before the AES rounds start so the oscilloscope can align each capture window the same way every time. Each encryption gives one power trace. We will start with a short pilot run to tune sample rate, vertical scale, and trigger offset, and then record the large dataset used for the statistics. The pilot is also where we make sure the AES window actually sits in the middle of the capture so SPA can show the repeatable round structure we expect.

7.1.2 SPA, DPA, and CPA Procedures

For SPA, we will stack a small set of traces and visually check for consistent features that line up with the round operations. This is a quick read on whether the core is exposing obvious patterns or if the traces are already noisy. For DPA, we will group traces by a predicted bit, subtract group averages, and look for a stable difference curve at the right sample region. For CPA, we will use a standard leakage model and compute the Pearson correlation for each key guess across the capture window. A correct guess should create a correlation peak at specific samples tied to the round computation. The main numbers we will report are how many traces it takes before a correct key-byte "pops out," the height and sharpness of the best correlation peaks, and how often we get false positives at the wrong guesses.

We will keep the capture conditions identical before and after adding our leakage-reduction extension so the comparison is fair. That means the same clock frequency, the same trigger protocol, and the same capture window and sampling configuration. If the extension introduces intentional desynchronization or noise, we will widen the capture window slightly and use software re-alignment (cross-correlation on a short sub-window) so CPA can still evaluate the same operation points. The goal is to see whether the same analysis scripts need more traces or stop producing stable peaks after the modification.

7.1.3 Timing Checks and Data Control

For timing leakage, we will record a cycle count for each AES run and build a simple histogram. The expectation is that the distribution stays tight and does not depend on the plaintext or key. If we see a pattern (for example, certain plaintext bytes cause a longer path), we will track down whether it comes from memory access or a branch and fix it in the test harness. We are not treating timing as the main attack channel in this project, but we still want to confirm that nothing obvious is leaking through cycle-level variation.

Data handling will be straightforward so we can re-run analysis without guessing: each trace file will be stored with a small metadata record (board, bitstream hash, firmware hash, clock, shunt value, sample rate, trigger offsets, plaintext list, key ID). For pre-modification and post-modification, we will keep the same plaintext schedule and the same count targets so "traces to first key-byte hit" is directly comparable. If needed, we will also try a second plaintext pattern (like a byte-toggling set) as a sensitivity check and label those runs separately so they don't mix with the main dataset.

Finally, we will document any adjustments we make that could affect the results, like changing the shunt value, moving the probe points, or altering the capture window. If those change for stability reasons, we will rerun the corresponding baseline so that the comparison still uses matched conditions. The end result of this section will be a clean set of SPA plots and DPA/CPA curves that show how the unmodified Ibex behaves and how the behavior shifts after we add the extension, using the exact same AES workload and trace-processing steps.

7.2 FPGA Implementation

To properly test power-based attacks on the Ibex core, we will run tests with an FPGA hosting the Ibex core. This will require testing and measuring the power-based attacks on the base Ibex core before modification and adding the SCALER extension, and testing after modifying the Ibex core. We will use the same program, the same capture settings, and the same measurement points in both phases so the results are directly comparable.

7.2.1 Minimal Hardware and Firmware Loop

First we build the plain Ibex system and add necessary components to the core for the extension: a small on-board memory block so the program runs locally, a simple serial print as a pulse, and one spare output pin as a trigger. The firmware can raise that trigger pin, perform one encryption, lower the trigger, and repeat with a list of known messages. We will choose a moderate board clock so the oscilloscope can sample cleanly; staying in the tens of megahertz makes alignment easier and reduces noise from very fast edges. In the power line that feeds the logic fabric, we place a small series resistor so a tiny voltage drop appears whenever the current changes. A two-lead probe measures that drop while the board runs. One encryption equals one captured trace. We will do a short practice run to tune alignment and vertical scale, and then capture the large set used for statistics. After we add SCALER, we load the new file onto the board and repeat the exact same capture process.

7.2.2 Power Line Selection and Probing

Selection of the FPGA board and power line traces for measurement will determine the accuracy of the test. A board like Basys 3 has multiple power traces: one that feeds the logic inside the chip, another for helper circuits, and one for the input/output pins. The line that powers the logic is the one most tied to switching inside Ibex, so that is our first choice even if it is slightly harder to reach. If the board already has a removable jumper or a small built-in sense part on that line, we use that spot for measurement. If not, we insert a low-value precision resistor in series with that line (ex. around 0.1 ohm). We keep all wires short and put the probe directly across that resistor. If we cannot safely reach the internal logic line, we will measure on the main input line instead and accept that it is noisier; to compensate, we will capture more traces and be careful with alignment during analysis. Either way, we will write down exactly where we probed and which parts remained in the power path so the setup can be repeated later.

7.2.3 Sampling, Noise, and Drift

Picking the resistor and the sampling settings is about balance. The resistor must be small so it does not starve the chip, but not so small that the oscilloscope cannot see anything. For the oscilloscope, we want several samples per clock edge

inside the encryption window. If our board clock is around 20-50 MHz, starting with a sample rate in the low hundreds of millions of samples per second is reasonable. We will use a capture mode that stores many short segments back-to-back, one segment per encryption, with a little time before the trigger so every segment includes a baseline. We set the vertical scale so the signal never clips but is not too tiny, and we keep that scale fixed for both the before and after runs.

Triggering and alignment will be kept simple. The firmware raises the trigger pin right before the round sequence starts and lowers it after the encryption finishes. That pin connects to a separate scope channel and acts as the trigger source. We place the trigger so the first round sits well inside the capture window, and we keep a small pre-trigger section for baseline. If SCALER adds small timing movement on purpose, we keep the same trigger but widen the window slightly so the whole operation still fits. After capture, we do a light software alignment step: we look for a short, known "bump" in the traces and nudge each trace so that bump lines up across all traces. We save both raw and aligned versions.

We will also control noise and drift. We keep the probe ground lead short and avoid long clip leads. We route power and probe cables away from the board's clock source and any switching power parts. If the scope gets power from a noisy laptop port, we switch to a separate power brick. Some people remove certain decoupling parts to make the signal larger, but that can make the board unstable; we will only change decoupling if the design allows it safely, and if we do, we will document exactly what changed. Temperature drift matters for long sessions, so we let the board warm up for a few minutes, then start the long capture so the baseline is steady.

7.2.4 Workload, Keys, and Reproducibility

Proper management of messages and keys is essential for proper testing. For baseline runs, we keep the key fixed for that session and go through a known list of messages. We log every message and a key label next to the trace number so that the analysis can rebuild the predictions exactly. Between long runs, we can rotate to a different key to confirm the behavior is not a one-off tied to a single key. If we want to stress a particular part of the computation like flipping only one message byte, we will do that as a separate and clearly labeled run so that it does not mix with the main data set.

The clock and steady operation will remain constant across the comparisons. We pick one clock for the whole dataset and we do not change it between the before and after runs. Even if the modified design could run at a different speed, we keep the test clock fixed so the traces are comparable. If the internal clock block of the board introduces random movement that we cannot fully control, we lock its settings in the design file. If needed, we can feed a clean external clock and make a note of that setting.

The scope stores traces as many small segments; we export them to a simple format our scripts can read easily, like comma-separated values or NumPy arrays. In the same folder, we put a short notes file in plain text or JSON listing the board type, the exact file we loaded onto the board, the program version, clock speed, resistor

value, scope sample rate, vertical scale, trigger offsets, the list of messages, and the key label. Using the same notes layout before and after SCALER lets us re-run analysis quickly and keeps the comparison fair.

7.2.5 Capture and Comparison

The setup will first be validated with a small practice capture. Around two hundred traces is enough to check that the trigger edge is clean, that the main part of the encryption sits inside the window, and that a quick "stack" plot shows repeatable bumps where we expect them. If we are capturing too much idle time or cutting off part of the round, we adjust the window. We also check that the signal is not clipped and that the baseline is flat inside each segment. Only after these checks do we run the long capture with thousands of traces.

All of these steps collectively create a more reliable comparison for the pre and post modified Ibex core through utilizing all of the same tools and systems. If the extension reduces the information leak, it should take more traces to recover any key bytes, and the correlation curves should drop or flatten compared to the base case. If that does not happen, the setup has sufficient detail that we can figure out why and adjust the approach.

7.3 SCA Attack Design

There are several established side-channel attack designs, and a focused choice is needed to evaluate the pre-modified and post-modified Ibex core specifically for side-channel leakage rather than only meeting general technical constraints or standards. This section specifies the workload, attacker model, leakage models, analysis windows, statistical procedures, decision metrics, and control conditions used to quantify changes in leakage attributable to the SCALER extension.

7.3.1 Attacker Model

AES-128 encryption on an Ibex-based system serves as the target workload. Each capture session fixes the secret key and uses a known plaintext schedule. Firmware raises a trigger signal immediately before the round sequence to align traces to the same operation window; one encryption corresponds to one short trace segment. The attacker is assumed to know the plaintexts and observe the trace per encryption while not knowing the key—standard for embedded SCA evaluation and directly compatible with SPA/DPA/CPA analysis.

7.3.2 Analysis and Leakage Models

The primary point of interest is the first-round S-box, which is both sensitive and well documented in the literature. A Hamming-weight model of the S-box output provides the initial leakage hypothesis; Hamming-distance across state updates is

considered if needed. The analysis window is selected around early-round activity using the trigger and a small overlay of traces to confirm alignment. If SCALER introduces intentional desynchronization, the window is widened and light software re-alignment is applied so the evaluation still targets equivalent internal events. Consistency of windows and models across "before" and "after" runs is maintained to keep comparisons fair.

7.3.3 Procedures for SPA, DPA, and CPA

Simple Power Analysis (SPA) provides a quick stability check by overlaying a handful of traces and confirming visible, repeatable features at expected round boundaries. Differential Power Analysis (DPA) partitions traces by a predicted bit (e.g., a bit of the first S-box output derived from plaintext and a key-byte guess) and inspects the difference of group means at relevant indices. Correlation Power Analysis (CPA) sweeps all key-byte guesses and computes Pearson correlation between model predictions and measured samples; correct guesses typically produce a distinct peak at specific sample locations. The same procedures, windows, and parameters are applied to the baseline and SCALER datasets to avoid analysis bias.

7.3.4 Metrics and Conditions

Primary metrics include traces-to-first correct key-byte under CPA, peak height and sharpness of the best correlation per byte, and transient false-positive behavior when incorrect guesses momentarily peak. For DPA, the stability of difference-of-means curves at expected indices and the traces required to reach stability are recorded. An effective countermeasure shifts these curves, either through increasing the traces required beyond the planned budget, or preventing stable peaks from forming under identical conditions. Observable changes in traces are noted as supporting evidence.

7.3.5 Controls

Conditions are held constant across datasets: same board, power measurement point, clock, trigger protocol, capture window, plaintext schedule, and analysis scripts. Each run carries a compact metadata record (bitstream/firmware versions, clock, sense resistor, sampling configuration, trace counts) to support exact repetition. If any stability fix requires changing the measurement point or capture window, the corresponding baseline is re-taken so comparisons remain matched.

7.3.6 Interpreting Outcomes

Outcome statements emphasize practical resistance. If early correlation peaks observed on the baseline do not appear under the same trace count after SCALER, the extension is credited with lowering leakage under the stated attacker model. Mixed

outcomes trigger model sensitivity checks and, if indicated, a note on whether higher-order analysis would be necessary to make further progress. Findings are tied back to SCALER's intended mechanism to connect measurements with design intent.

7.4 Base Ibex Core Testing

This phase establishes the baseline leakage behavior of the unmodified Ibex core for meaningful comparisons for the post-modification core. This plan covers functional testing, stable trace collection on an FPGA board during AES encryptions, timing sanity checks, and data logging for repeatability. The same firmware, trigger protocol, clock, and capture windows used here are reused after the modification so differences are attributable to the extension and not the setup.

The base SoC is programmed with a small test application that repeatedly encrypts known plaintexts under a fixed key per session. A dedicated trigger pin is toggled immediately around the AES round sequence to align the oscilloscope captures. Before collecting large datasets, we confirm correct AES outputs against known values, check that the trigger edge is clean, and verify that the expected round activity sits inside the capture window. A short pilot capture is used to finalize sample rate, scale, and pre/post-trigger margins.

7.4.1 Power Trace Capture

With alignment confirmed, the board collects thousands of short segments using a series resistor in the logic power path and a differential measurement across it. The capture conditions are kept constant across runs: same clock in the tens of megahertz, same window, same probe placement, and the same firmware loop. A quick SPA overlay verifies that the trace features are repeatable enough for downstream DPA/CPA. The larger dataset is then exported together with a compact metadata file.

7.4.2 Leakage Checks and Controls

Cycle counts are recorded for each encryption across large batches. The expectation is a tight, key-independent distribution. Any outliers are investigated for causes such as I/O interactions or unintended branches in the test harness. While timing is not the primary attack channel in this project, confirming constant-time behavior avoids misinterpreting power-analysis results.

A run is considered valid when AES outputs match the reference for all sampled messages, SPA shows stable structure without clipping or baseline drift, and trigger alignment requires only small software adjustments. Known pitfalls such as differences in temperature and noise are mitigated through various methods to retain proper collection of leakage data.

7.5 SCALER Extension Testing

Post-modification testing evaluates leakage after integrating the SCALER extension into Ibex. Side-channel checkpoints appear throughout the flow: RTL simulation for functional and timing sanity, gate-level power estimates to understand switching trends; post-layout reports for area, timing, and power overheads up to GDSII, and on-board power trace measurements using the same FPGA capture flow as the baseline. Together, these form the narrative of local changes, implementation costs, and whether the traces reflect reduced side-channel leakage.

After we integrate our leakage reduction extension for the power-analysis attacks, we will host the modified RISC-V core on the FPGA and repeat the same attacks performed prior to the Ibex core modification. This will include the same methods of measuring the clock frequency, high-frequency changes during operation, and utilizing an oscilloscope that can measure the voltage drop.

7.5.1 Testing and Configuration

For fair comparison, the firmware, trigger protocol, clock, and capture window are unchanged. SCALER-specific settings are documented per run. Each configuration in the matrix produces its own dataset with identical trace counts and analysis scripts. If desynchronization is part of the design, the capture window is widened slightly and a light software re-alignment is applied before DPA/CPA. Any change that could influence measurement is mirrored by re-taking the corresponding baseline slice so the comparison stays matched.

7.5.2 Capture on FPGA

The same board and measurement point are used on the Basys 3 FPGA. Stable triggering is confirmed and SPA overlays remain interpretable, even if round features appear blurrier due to the countermeasure. Once stable, the long capture proceeds to produce the main dataset. Initial plots provide an early signal for alterations in the traces to emerge. Any new periodic features are noted and cross-checked with a control to ensure they are design-related and not a board artifact.

7.5.3 Comparing Results

Results comparison between pre-modified core and post-modification core.

Chapter 8 - Administrative Content

8.1 Bill of Materials

This project centers primarily on software-based development and simulation rather than physical prototyping or fabrication. As a result, there are no tangible materials or hardware components to include in the Bill of Materials (BOM). Instead, the focus is on the software tools, intellectual resources, and computing infrastructure that enable the complete digital ASIC design process. The following table summarizes the essential resources required to perform RTL design, synthesis, physical implementation, and verification of the SCALER RISC-V security extension, effectively capturing all critical components necessary to execute the ASIC flow end-to-end.

Table 8.1: *Bill of Materials (BOM)*

Bill of Materials (BOM)		
Category	**Item / Resource**	**Source / Cost**
EDA Tools	Cadence Xcelium, Genus, and Innovus – used for simulation, synthesis, and physical design throughout the ASIC flow.	Provided via UCF Academic License
HDL Design Environment	SystemVerilog and UVM – for RTL design, testbench development, and functional verification.	Open-source / No Cost
Open-Source Reference Core	Ibex RISC-V Core (OpenHW Group) – baseline RTL for security extension development.	Open-source / No Cost
RISC-V Toolchain	GCC / LLVM with custom ISA extensions for program compilation and validation.	Open-source / No Cost
Computing Resources	UCF HPC Servers and Department Workstations – used for simulation, synthesis, and place-and-route workloads.	Provided by University

Continued on next page

Bill of Materials (BOM) (continued)		
Category	**Item / Resource**	**Source / Cost**
Documentation Tools	Overleaf (LaTeX) and GitHub – used for collaborative documentation, version control, and project management.	Free Academic Access
Training Resources	Cadence Training Portal and online ASIC flow modules for tool proficiency and methodology learning.	Provided through Academic Program

8.2 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.

8.3 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.

8.3.1 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
Start 40-Page Draft	9/10/25	9/18/25	1 Week	Write 40-page draft ($\frac{1}{4}$ 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
Midterm Report	9/26/25	10/31/25	4 weeks	Write Midterm Report (2/3 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
Midterm Report Revision	10/31/25	11/10/25	1 Week	Revise and organize Midterm Report for final paper
Final Report Draft	11/10/25	11/19/25	1.5 Weeks	Write remaining pages for final report (target 150 pages)

Continued on next page

Senior Design 1 Timeline (continued)				
Task	**Start Date**	**End Date**	**Duration**	**Description**
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

8.3.2 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
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

8.3.3 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
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

8.3.4 Project Milestone Summary

The project milestones for SCALER are structures around the two semester Senior Design sequence. The focus for Senior Design 1 (SD1) is research, design definition, RTL development, and verification. Senior Design 2 (SD2) focuses on synthesis, physical design, and validation of the ASIC flow.

8.3.4.1 Senior Design 1

Objective: Research and learn in-depth concepts of the ASIC design flow, become familiar with the Cadence Design Suite, Establish the architectural baseline, develop custom ISA instructions, and validate the RTL.

Key Deliverables:

- Completion of SD1 Final Report, including all required content and incorporating all necessary revisions
- Implementation of noise-injection and masking logic in RTL
- Verification of instruction correctness through simulation test benches
- Documentation of success and performance metrics

8.3.4.2 Senior Design 2

Objective: Translate the verified RTL into a gate-level netlist, perform synthesis, physical implementation, and validate engineering requirements.

Key Deliverables:

- Gate-level synthesis using Cadence Genus
- Place-and-route and synthesis using Cadence Innovus
- Static timing and power validation to engineering standards

- Design rule and layout-versus-schematic (DRC & LVS) checks for sign off
- Generation of a final GDS-II mock tapeout for verification report

Chapter 9 - Conclusion

Final Conclusion

Chapter A - References

Bibliography

[1] K. Stangherlin and M. Sachdev, "Design and implementation of a secure risc-v microprocessor," 2022.

[2] J. Mishra and S. K. Sahay, "Modern hardware security: A review of attacks and countermeasures," *arXiv preprint*, 2025.

[3] M. Lipp *et al.*, "Meltdown." `https://arxiv.org/pdf/1801.01207`, 2018. Accessed: Sept. 11, 2025.

[4] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors." `https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf`, 2014. ISCA 2014, Accessed: Oct. 28, 2025.

[5] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms." `https://vvdveen.com/publications/drammer.pdf`, 2016. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), Accessed: Oct. 28, 2025.

[6] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript." `https://arxiv.org/pdf/1507.06955`, 2015. arXiv:1507.06955, DIMVA 2016 (accepted), Accessed: Oct. 28, 2025.

[7] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*, pp. 695–711, IEEE, May 2020.

[8] H. Luo, A. Olgun, A. G. Yaglikci, Y. C. Tugrul, S. Rhyner, M. B. Cavlak, J. Lindegger, M. Sadrosadati, and O. Mutlu, "Rowpress: Amplifying read disturbance in modern dram chips," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, (Orlando, FL, USA), ACM, June 2023. 18 pages.

[9] H. Luo, A. Olgun, A. G. Yaglikci, Y. C. Tugrul, S. Rhyner, M. B. Cavlak, J. Lindegger, M. Sadrosadati, and O. Mutlu, "Rowpress vulnerability in modern dram chips." arXiv:2406.16153 [cs.AR], June 2024. To appear in *IEEE Micro* Top Picks Special Issue (Jul.–Aug. 2024).

[10] H. Luo, I. E. Yuksel, A. Olgun, A. G. Yaglikci, M. Sadrosadati, and O. Mutlu, "An experimental characterization of combined rowhammer and rowpress read disturbance in modern dram chips." arXiv:2406.13080 [cs.AR], June 2024. To appear at DSN Disrupt 2024.

[11] M. Xue, Y. Liu, R. Karri, D. Forte, and S. Bhunia, "Ten years of hardware trojans: A survey from the attacker's perspective," *IET Computers & Digital Techniques*, vol. 14, Sep 2020.

[12] E. D. Mulder, S. Gummalla, and M. Hutter, "INVITED: Protecting risc-v against side-channel attacks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, (Las Vegas, NV, USA), pp. 1–4, 2019.

[13] M. M. Ahmadi, F. Khalid, and M. Shafique, "Side-channel attacks on risc-v processors: Current progress, challenges, and opportunities," *arXiv preprint*, Jun 2021.

[14] A. A. Circuits, "Hardware description language: Getting started with vhdl for digital circuit design." `https://www.allaboutcircuits.com/technical-articles/hardware-description-langauge-getting-started-vhdl-digital-circuit-design/`. Accessed: Sep. 4, 2025.

[15] GeeksforGeeks, "Getting started with verilog." `https://www.geeksforgeeks.org/electronics-engineering/getting-started-with-verilog/`. Accessed: Sep. 4, 2025.

[16] chipsalliance, "rocket-chip (rocket chip generator)." `https://github.com/chipsalliance/rocket-chip`. GitHub, Accessed: Sep. 4, 2025.

[17] Chipyard, "3.2 rocket core — chipyard 1.11.0 documentation." `https://chipyard.readthedocs.io/en/stable/Generators/Rocket.html`. Chipyard Documentation, Accessed: Sep. 4, 2025.

[18] stnolting, "neorv32-verilog: Convert the neorv32 processor into a synthesizable plain-verilog netlist module using ghdl." `https://github.com/stnolting/neorv32-verilog`. GitHub, Accessed: Sep. 4, 2025.

[19] OpenXiangShan, "Xiangshan: Open-source high-performance risc-v processor." `https://github.com/OpenXiangShan/XiangShan`. GitHub, Accessed: Sep. 4, 2025.

[20] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution." `https://spectreattack.com/spectre.pdf`, 2018. Accessed: Sept. 4, 2025.

[21] XiangShan, "Xiangshan official documents (latest)." `https://docs.xiangshan.cc/zh-cn/latest/`. Accessed: Sep. 4, 2025.

[22] YosysHQ, "picorv32 – a size-optimized risc-v cpu." https://github.com/YosysHQ/picorv32. GitHub, Accessed: Sept. 4, 2025.

[23] lowRISC, "Ibex: A small 32-bit risc-v cpu core previously known as zero-riscy." https://github.com/lowRISC/ibex. GitHub, Accessed: Sept. 4, 2025.

[24] O. Kindgren, "Serv – the serial risc-v cpu." https://github.com/olofk/serv. GitHub repository, Accessed: Sept. 4, 2025.

[25] C. Zeoli, "How synopsys and cadence are fueling the semiconductor industry's growth engine." https://www.wing.vc/content/how-synopsys-and-cadence-are-fueling-the-semiconductor-industrys-growth-engine, May 2024. Accessed: Oct. 10, 2025.

[26] O. Project, "About opentitan." https://opentitan.org/book/doc/sections/opentitan.html. OpenTitan Documentation, Accessed: Oct. 21, 2025.

[27] O. Project, "Earl grey (top_earlgrey) chip specification." https://opentitan.org/book/hw/top_earlgrey/doc/specification.html. OpenTitan Documentation, Accessed: Oct. 21, 2025.

[28] A. Unknown, "Document from hal — "hal-04132900"." https://hal.science/hal-04132900/document. Accessed: Oct. 30, 2025.

[29] A. Biryukov, D. Dinu, Y. L. Corre, and A. Udovenko, "Optimal first-order boolean masking for embedded iot devices," 2018. Accessed: 2025-10-20.

[30] A. R. Shahmirzadi and M. Hutter, "Efficient boolean-to-arithmetic mask conversion in hardware." https://eprint.iacr.org/2024/1633.pdf, 2024. IACR Cryptology ePrint Archive Report 2024/1633, 2024.

[31] J.-S. Coron, J. Großschädl, and P. K. Vadnala, "Secure conversion between boolean and arithmetic masking of any order," 2014.

[32] H. Gross, S. Mangard, and T. Korak, "Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order," 2016. Accessed: 2025-10-21.

[33] R. Corporation, "2 nm semiconductor challenges: Exploring rapidus' technological breakthroughs." https://www.rapidus.inc/en/tech/te0006/, 2025. Accessed: Oct. 30, 2025.

[34] M. Abhinav, "What is pvt in vlsi?." https://chipedge.com/resources/what-is-pvt-in-vlsi/. ChipEdge, Accessed: Oct. 30, 2025.

[35] M. Katerbarg, "How timing attacks threaten post-quantum cryptography." https://www.sectigo.com/resource-library/how-timing-attacks-threaten-postquantum-cryptography. Accessed: Oct. 30, 2025.

[36] M. Witteman, "Security highlight: Clock jitter and side channel leakage." https://www.keysight.com/blogs/en/tech/nwvs/2023/10/03/security-highlight-clock-jitter-and-side-channel-leakage, 2023. Accessed: Oct. 30, 2025.

[37] U. S. Patent, P. T. Trademark Office, and A. Board, "Petition documents — case no. ipr 2020-01692, petition 1541825." https://ptacts.uspto.gov/ptacts/public-informations/petitions/1541825/download-documents?artifactId=Y-PxPacZ40fhnMDD7m7afzlyQPrcLieH3UYDZu7sBx5OJfXY-c5NcQM, 2021. Accessed: Oct. 30, 2025.

[38] D. Blog, "What is cpu cache and how does it impact performance?." https://directmacro.com/blog/post/what-is-cpu-cache-and-how-does-it-impact-performance#_toc_What_is_CPU_Cache, 2023. Accessed: Oct. 30, 2025.

[39] A. Pulkit, S. Naval, and V. Laxmi, "A survey of side-channel attacks in context of cache — taxonomies, analysis and mitigation." https://arxiv.org/abs/2312.11094, 2023. arXiv preprint, Accessed: Oct. 30, 2025.

[40] enjoy digital, "Litex wiki." https://github.com/enjoy-digital/litex/wiki. GitHub Wiki, Accessed: Oct. 28, 2025.

[41] O. Kindgren, "Fusesoc — package manager and build tool for fpga/asic." https://github.com/olofk/fusesoc. GitHub repository, Accessed: Oct. 28, 2025.

[42] A. S. Initiative, "Universal verification methodology (uvm) 1.2 class reference manual." https://www.accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf. Accellera, Accessed: Oct. 26, 2025.

[43] ChipVerify, "Uvm introduction — major uvm class categories." https://www.chipverify.com/uvm/uvm-introduction#major-uvm-class-categories. ChipVerify, Accessed: Oct. 26, 2025.

[44] O. Project, "Comportability definition and specification." https://opentitan.org/book/doc/contributing/hw/comportability/. OpenTitan Documentation, Accessed: Oct. 21, 2025.

[45] I. S. Association, "Ieee std 1497™ – standard delay format (sdf) for the electronic design process." https://standards.ieee.org/ieee/1497/2235/, 2001. Accessed: Nov. 6, 2025.

[46] ChipVerify, "Standard delay format (sdf)." https://www.chipverify.com/verilog/verilog-sdf. Accessed: Nov. 6, 2025.

[47] I. Cadence Design Systems, "Conformal equivalence checker — datasheet." https://www.cadence.com/en_US/home/resources/datasheets/conformal-equivalence-checker-ds.html. Accessed: Nov. 6, 2025.

Chapter B - Copyright Permission

Insert permission letters/emails.

Chapter C - Datasheets (if needed)

Chapter D - Software Code (if needed)

Chapter E - ChatGPT Prompts and Outcomes

E.1 Case Study 1:

Can You Cause Jitter with an RTL Version of a Ring Oscillator?

Yes, you can observe **jitter-like behavior** in an RTL implementation of a ring oscillator, but it is important to note that this does **not correspond to real physical jitter**. It only occurs if you explicitly model it in your simulation.

1. What a Ring Oscillator Is

A ring oscillator is a chain of an odd number of inverters (or any circuit with an odd number of logical inversions) whose output feeds back into its input. In hardware, it oscillates because of the **propagation delay** through each inverter, which causes the signal to continuously flip.

2. RTL Simulation Characteristics

At the **Register Transfer Level (RTL)**, simulators such as ModelSim or Synopsys VCS typically assume **zero or fixed delays** for combinational logic unless you explicitly specify them.

For example, consider the following Verilog snippet:

```
assign a = ~b;
assign b = ~a;
```

Because there is no modeled delay, most simulators will either find this configuration **undefined (X)** or resolve it to a stable value. It will *not oscillate*.

Even if you introduce artificial fixed delays, e.g.,

```
assign #1 a = ~b;
assign #1 b = ~a;
```

the simulation will produce a perfectly deterministic square wave with no variation in timing—hence, **no jitter**.

3. Sources of Jitter in Real Hardware

In real silicon, jitter arises due to several physical effects:

- Thermal and supply noise
- Device mismatch (variation in inverter delays)
- Temperature and voltage fluctuations
- Random transistor-level timing variations

These effects introduce **random timing noise**, resulting in **cycle-to-cycle jitter** in the oscillator output.

4. Modeling Jitter in RTL

If you wish to emulate jitter behavior in an RTL simulation, you must inject *random delay variations* explicitly. For example:

```
real delay;
always begin
  delay = 1.0 + ($urandom_range(-10,10) * 0.01); // vary +/-10%
  #delay clk_out = ~clk_out;
end
```

This approach produces an artificial oscillation with pseudo-random period variation—useful for system-level or verification purposes, but not physically accurate.

5. Summary

Environment	**Oscillation?**	**Jitter?**	**Notes**
RTL (no delay)	No	No	Loop collapses to X or stable value
RTL (fixed delay)	Yes	No	Perfect periodic signal
RTL (randomized delay)	Yes	Yes	Artificial jitter model
Gate-level / SPICE	Yes	Yes	Real physical jitter sources

6. Conclusion

In conclusion, an RTL-level ring oscillator will not exhibit real jitter unless you explicitly introduce it through random delay modeling. To analyze true timing noise, you must simulate at the **gate level** or use a **SPICE** simulation, where physical transistor effects and noise sources are included.